The C5 Generic Collection Library for C# and CLI

Version 1.0.1 of 2006-06-26

Niels Kokholm
Peter Sestoft

IT University Technical Report Series **TR-2006-76**

ISSN 1600–6100 **January 2006**

ISSN 1600–6100

ISBN 87-7949-114-6

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaardsvej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web `www.itu.dk`

Preface

This book describes the C5 library of generic collection classes (or container classes) for the C# programming language and other generics-enabled languages on version 2.0 of the CLI platform, as implemented by Microsoft .NET and Mono. The C5 library provides a wide range of classic data structures, rich functionality, the best possible asymptotic time complexity, documented performance, and a thoroughly tested implementation.

Goals of the C5 library The overall goal is for C5 to be a generic collection library for the C# programming language [11, 17, 27] and the Common Language Infrastructure (CLI) [12] whose functionality, efficiency and quality meets or exceeds what is available for similar contemporary programming platforms. The design has been influenced by the collection libraries for Java and Smalltalk and the published critique of these. However, it contains functionality and a regularity of design that considerably exceeds that of the standard libraries for those languages.

Why yet another generic collection library? There are already other generic collection libraries for C#/CLI, including the System.Collections.Generic namespace of the CLI or .NET Framework class library (included with Microsoft Visual Studio 2005), and the PowerCollections library developed by Peter Golde [15].

The CLI collection library as implemented by Microsoft .NET Framework 2.0 provides a limited choice of data structures. In general, the CLI Framework library has a proliferation of method variants and rather poor orthogonality. Collection implementations such as array lists and linked lists have much the same functionality but do not implement a common interface. This impairs the learnability of the library. Some of these design decisions are well-motivated by the use of the CLI class library in contexts where nano-second efficiency is more important than rich functionality, and the need to support also rather resource-constrained run-time systems.

The PowerCollections library by Peter Golde augments the CLI version 2.0 collection library with various data structures and algorithms. However, it accepts the basic design of the CLI collection classes and therefore suffers from some of the same shortcomings, and also does not provide most of the advanced functionality (updatable list views, snapshots, directed enumeration, priority queue handles, . . .)

of C5.

Thus, in our opinion, C5 provides the most powerful, well-structured and scalable generic collections library available for C#/CLI. However, although the size of the compiled `C5.dll` is only about 300 KB, you probably would not choose to use it on your .NET 2.0 Compact Framework wristwatch just now.

What does the name C5 stand for? This is not entirely clear, but it may stand for *Copenhagen Comprehensive Collection Classes for C#*, although the library may be used from VB.NET, F# and other CLI languages, not just C#. It has nothing to do with a Microsoft Dynamics product that used to be called Concorde C5/Damgaard C5/Navision C5, nor a real-time operating system called C5 (or Chorus), nor the C5 Corporation (system visualization), nor an Eclipse plug-in called C5, nor with cars such as Citroën C5 or Corvette C5 or Clive Sinclair's ill-fated C5 concept vehicle. The name may be inspired by the versatile C4 plastic explosive known from e.g. James Bond movies. All trademarks belong to their owners.

State of completion At the time of writing, library design and implementation are complete, and extensive unit tests have been written and applied systematically. Most of the library API documentation is in place but requires proof-reading.

The C5 implementation uses most of the new features of C# 2.0: generic types and methods, type parameter constraints, iterator blocks, anonymous methods, and nullable value types. It was developed using alpha, beta and final releases of Microsoft .NET 2.0, but uses only standard libraries and CLI features, and the library builds and passes the units test on the Mono 1.1.15 implementation on CLI.

This book The present book is a guide to effective use of C5. It gives an overview of the library and its design rationale, describes the entire API in detail, including the time complexity of all operations, presents more than a hundred small usage patterns (idioms), and several larger, complete examples. Finally it explains some of the techniques used in the implementation.

C5 availability and license The complete C5 library implementation, including documentation, is available in binary and source form from the IT University of Copenhagen:

`http://www.itu.dk/research/c5/`

The library is copyrighted by the authors and distributed under a BSD-style license:

So you can use and modify it for any purpose, including commerce, without a license fee, but the copyright notice must remain in place, and you cannot blame us or our employers for any consequences of using or abusing the library.

The authors Niels Kokholm holds an MSc and a PhD in mathematics and an MSc in information technology and software development. He makes software for the insurance industry at Edlund A/S in Copenhagen. Peter Sestoft holds an MSc and a PhD in computer science, is a member of the Ecma C# and CLI standardization committees, and the author of *C# Precisely* [27] and *Java Precisely* (MIT Press). He is professor at the IT University of Copenhagen and the Royal Veterinary and Agricultural University (KVL), Denmark.

Acknowledgements A visit to Microsoft Research, Cambridge UK, in late 2001 permitted Peter to write a first rudimentary generic collection library for C#, which was considerably redesigned, extended and improved by Niels during 2003–2004.

We thank Microsoft Research University Relations for a grant that enabled us to complete, improve and document the implementation in 2004–2005.

Thanks to Daniel Morton and Jon Jagger for comments on the design and implementation of C5, and to the IT University of Copenhagen and the Royal Veterinary and Agricultural University (KVL), Denmark, for their support.

Notational conventions

Symbol	Use	Type	Section
act	action delegate	Act<A1>	3.6.1
arr	array	T[]	
cmp	comparer	SCG.IComparer<T>	2.6
cq	circular queue	IQueue<T>	6.1
csn	comparison delegate	System.Comparison<T>	2.6
eqc	equality comparer	SCG.IEqualityComparer<T>	2.3
f	function delegate	Fun<A1,R>	3.6.2
h	priority queue handle	IPriorityQueueHandle<T>	1.4.12
i, j	offset or index into collection	int	
k	key in dictionary	K	
ks	sequence of keys in dictionary	SCG.IEnumerable<K>	
kv	(key,value) pair, entry	KeyValuePair<K,V>	
kvs	sequence of (key,value) pairs		3.5
m, n, N	count, length, or size	int	
p	predicate delegate	Fun<T, bool>	3.6.2
rnd	random number generator	System.Random	3.8
T, U, K, V	generic type parameter		
v	value of key in dictionary	V	
w, u	view	IList<T>	8.1
x, y	collection item		
xs, ys	item sequence, enumerable	SCG.IEnumerable<T>	

The abbreviation SCG stands for System.Collections.Generic, that is, the CLI class library namespace for generic collection classes. This namespace is standardized as part of Ecma CLI [12].

Common example declaration header

All program fragments and code examples shown in this book are supposed to be preceded by these declarations:

```
using System;
using C5;
using SCG = System.Collections.Generic;
```

Contents

Chapter 1

Collection concepts

This section gives a simple example using the C5 generic collection library, describes the design goals of the library, and introduces some basic notions such as item equality, hashing and comparers for item ordering. The rest of the chapter briefly presents each collection interface in C5 and thus gives an overview of the available collection concepts.

1.1 Getting started

Here is a small but complete example using the C5 collection library:

```
using System;
using C5;
using SCG = System.Collections.Generic;
namespace GettingStarted {
  class GettingStarted {
    public static void Main(String[] args) {
      IList<String> names = new ArrayList<String>();
      names.AddAll(new String[] { "Hoover", "Roosevelt",
                                  "Truman", "Eisenhower", "Kennedy" });
      // Print list:
      Console.WriteLine(names);
      // Print item 1 ("Roosevelt") in the list:
      Console.WriteLine(names[1]);
      // Create a list view comprising post-WW2 presidents:
      IList<String> postWWII = names.View(2, 3);
      // Print item 2 ("Kennedy") in the view:
      Console.WriteLine(postWWII[2]);
      // Enumerate and print the list view in reverse chronological order:
      foreach (String name in postWWII.Backwards())
        Console.WriteLine(name);
    }
  }
}
```

The example illustrates a simple use of the ArrayList<T> class through the IList<T> interface. It shows how to print the entire list; how to index into the list; how to create a list view comprising `Truman`, `Eisenhower` and `Kennedy`; how to index into that view; and how to enumerate the items in the view in reverse order. The output of the program should look like this:

```
[ 0:Hoover, 1:Roosevelt, 2:Truman, 3:Eisenhower, 4:Kennedy ]
Roosevelt
Kennedy
Kennedy
Eisenhower
Truman
```

To compile and run this example program, a compiled version of the library C5 must be installed, in the form of the file `C5.dll`. You also need a C# 2.0 compiler, such as Microsoft's `csc` or the Mono project's `mcs`.

Assume that the example program is saved to a C# source file `GettingStarted.cs`. Then you can compile and run it using Microsoft's command line compiler from a Command Prompt like this:

```
csc /r:C5.dll GettingStarted.cs
GettingStarted.exe
```

Using the Mono project's command line compiler, you can compile and run the example program like this:

```
mcs /r:C5.dll GettingStarted.cs
mono GettingStarted.exe
```

Using an integrated development environment such as Microsoft Visual Studio 2005 or Visual C# Express 2005, you can create a project and then add `C5.dll` as a project reference: In the `Project` menu, select the `Add Reference` item to open a dialog; select the `Browse` tab; find `C5.dll` in the file dialog; and click `OK`.

1.2 Design goals of C5

The overall goal is for C5 to be a generic collection library for C#/CLI whose functionality, efficiency and quality meets or exceeds what is available for similar contemporary programming platforms, most notably Java. More precisely, our goals in developing the C5 library are:

- To provide a collection class library for C# that is as comprehensive as those of comparable languages, such as Java [3] and Smalltalk [14], while learning from the critique of these [7, 13, 21].
- To be tested and documented well enough that it will be widely usable.
- To provide a full complement of well-known abstractions such as lists, sets, bags, dictionaries, priority queues, (FIFO) queues and (LIFO) stacks.
- To provide well-known data structure implementations (array-based, linked list-based, hash-based, tree-based).
- To fit existing C# patterns (`SCG.IEnumerable<T>`, the `foreach` statement, events), and to enable good use of the new C# 2.0 features: generic types and methods, anonymous delegates, and iterator methods (the `yield` statement).
- To provide convenient but hard-to-implement features such as multiple updatable list views, hash-indexed lists, persistent trees, reversible enumeration, events on collection modifications, and priority queue items that can be accessed by handle.
- To describe functionality by interfaces: "program to an interface, not an implementation".
- To provide orthogonal capabilities, such as enumeration directions and subranges, to avoid a proliferation of overloaded method variants.
- To document the asymptotic run-time complexity of all implementations.
- To make the implementation available under a liberal open source license.

The C5 library separates functionality (interface hierarchy) from data structures (class hierarchy). Interfaces should fully express the functionality of collection classes: what can be done to the collection. For instance, the IList<T> interface describes indexed sequences of items. Clearly, different concrete implementations of the interface have different tradeoffs in speed and space. For instance, there are both array list and linked list implementations of IList<T>. Whereas array lists have constant-time indexing and linear-time insertion (at the front of the list), linked lists have the opposite characteristics.

The design has been influenced by the collection libraries for Java and Smalltalk and the published critique of these. However, it has functionality and a regularity of design that considerably exceeds that of the standard libraries for those languages.

1.3 Overview of collection concepts

A *collection* contains zero or more *items*. The main collection interfaces, listed below, are further described in section 1.4, and chapter 4 describes their methods in detail.

- An SCG.IEnumerable<T> can have its items enumerated. All collections and dictionaries are enumerable.
- An IDirectedEnumerable<T> is an enumerable that can be reversed, giving a "backwards" enumerable that enumerates its items in the opposite order.
- An ICollectionValue<T> is a collection *value*: it does not support modification, is enumerable, knows how many items it has, and can copy them to an array.
- An IDirectedCollectionValue<T> is a collection value that can be reversed into a "backwards" collection value.
- An IExtensible<T> is a collection to which items can be added.
- An IPriorityQueue<T> is an extensible whose least and greatest items can be found (and removed) efficiently.
- An ICollection<T> is an extensible from which items can also be removed.
- An ISequenced<T> is a collection whose items appear in a particular sequence (determined either by insertion order or item ordering).
- An IIndexed<T> is a sequenced collection whose items are accessible by index.
- An ISorted<T> is a sequenced collection in which items appear in increasing order; item comparisons determine the item sequence. It can efficiently find the predecessor or successor (in the collection) of a given item.
- An IIndexedSorted<T> is an indexed and sorted collection. It can efficiently determine how many items are greater than or equal to a given item x.
- An IPersistentSorted<T> is a sorted collection of which one can efficiently make a snapshot, that is, a read-only copy that will remain unaffected by updates to the original collection.
- An IQueue<T> is a first-in-first-out queue that in addition supports indexing.
- An IStack<T> is a last-in-first-out stack.
- An IList<T> is an indexed and therefore sequenced collection, where item order is determined by insertions and deletions. Derives from SCG.IList<T>.

All C5 collection and dictionary classes classes are serializable and cloneable. They have `ToString` methods to facilitate program debugging and understanding of error messages. All C5 exception classes are serializable.

1.4 The collection interfaces

An interface describes operations supported by a collection class. For instance, interface ISequenced<T> describes collections whose items should appear in some order, as determined by the way they were inserted or determined by a comparer that says that one item is less than (precedes) another.

The interfaces are organized in a hierarchy, describing the relations between them. For instance, a list collection supports indexing, and its items are sequenced, so interface IList<T> extends IIndexed<T> and ISequenced<T>.

The functionality of a collection class is fully described by the collection interfaces it implements. The interface hierarchy is shown in figure 1.1, and the interface hierarchy with implementing classes is shown in figure 6.2.

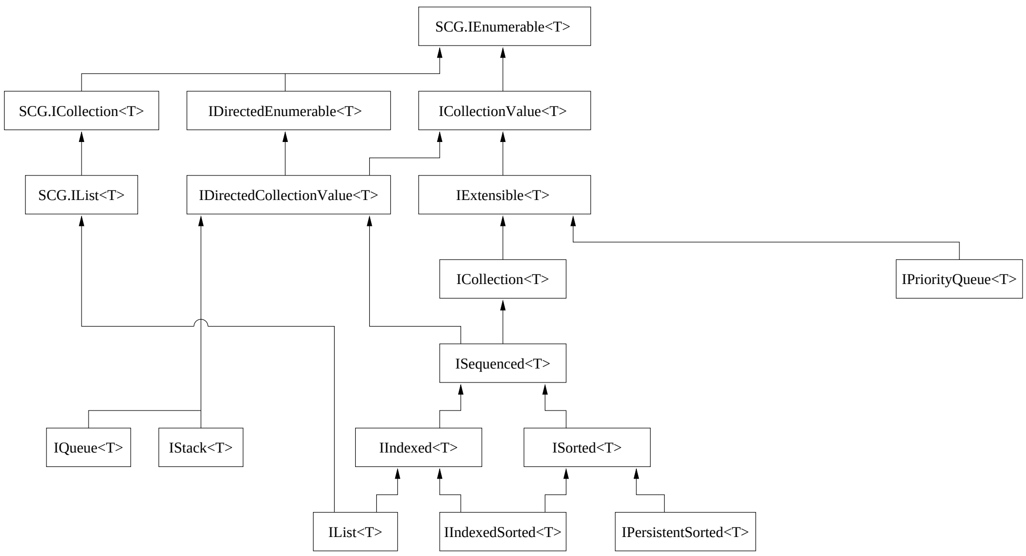


Figure 1.1: The collection interface hierarchy. See figure 6.2 for the classes implementing these interfaces. SGC is the System.Collections.Generic namespace.

1.4.1 Enumerables and directed enumerables

An *enumerable* implements interface SCG.IEnumerable<T> and is a collection of possibly unknown size. It describes method `GetEnumerator()` that returns a generator of T values in the form of an SCG.IEnumerator<T> object; this generator can also be considered a lazy stream of T values. All collection classes are enumerable. Their enumerators produce the collection's items in some order, called the collection's *enumeration order*. Therefore the `foreach` statement of C# can be used to iterate over a collection's items. The interfaces SCG.IEnumerable<T> and

SCG.IEnumerator<T> are from the CLI (or .NET Framework) class library namespace System.Collections.Generic, here abbreviated SCG.

A *directed enumerable* implements interface IDirectedEnumerable<T> and is an enumerable with an additional method `Backwards()`. This method returns a new IDirectedEnumerable<T> which when used as an enumerable will enumerate its items in the reverse order of the given one. Sequenced collections such as lists, trees, stacks and queues (but not hash sets and hash bags) implement IDirectedEnumerable<T>. Also list views and subranges of tree sets are directed enumerables and therefore can have their items enumerated backwards.

1.4.2 Collection values and directed collection values

A collection value implements interface ICollectionValue<T> and is a collection of known size but possibly not permitting any update or extension (hence the "value" in the name). It extends SCG.IEnumerable<T> with the properties `Count` and `IsEmpty` and methods `CopyTo(arr, i)` and `ToArray()` to copy the collection's items to a given array or a new array. Since the number of items is known, such methods can be implemented efficiently.

Furthermore it describes a method `Choose()` to get an arbitrary item from the collection, and methods `Apply`, `Exists`, `All` and `Filter` iteration, quantifier evaluation and filtering in the style of functional programming languages.

All standard collection classes implement ICollectionValue<T>.

A *directed collection* implements interface IDirectedCollectionValue<T> and is a collection value that also admits backwards enumeration and therefore implements IDirectedEnumerable<T>. Note for instance that if `coll` is such a collection, then `coll.Backwards().Apply(act)` will apply delegate `act` to the collection's items in reverse order.

1.4.3 Extensibles

An *extensible collection* implements interface IExtensible<T> and is a collection value to which items may be added. It extends ICollectionValue<T> with methods `Add` and `AddAll` to add more items to the collection. It also has a property `AllowsDuplicates` that tells whether the collection allows duplicate items and behaves like a bag (or multiset), or disallows duplicate items and behaves like a set. For collections that allow duplicates, the `DuplicatesByCounting` property says whether the collection actually stores the duplicate items, or just keeps a count of them; see section 1.4.14. The `EqualityComparer` property is the equality comparer used by this collection to define item equality and a hash function.

1.4.4 Collections

An (unsequenced) *collection* implements interface ICollection<T> and is an extensible collection that furthermore offers well-known collection operations. These include methods to clear the collection (remove all items), a method to test whether the collection contains a given item, methods to find, remove or update an item, methods to test whether the collection contains all items from a given enumerable, and methods to remove or retain all items from a given enumerable. There is also a property to determine whether the collection is read-only.

In addition, interface ICollection<T> describes methods `UnsequencedEquals` and `GetUnsequencedHashCode` for comparing collection objects (not the individual items), based on non-sequenced item comparison. This allows collections to be used as items in other collections.

The primary implementations of ICollection<T> are hash sets and hash bags (sections 6.10 and 6.11) in which items are added, retrieved and removed by equality and hash value. Hash sets and hash bags do not implement more specific interfaces (lower in the hierarchy) than ICollection<T>, as can be seen from figure 6.2.

The reader may wonder whether there are any reasonable “collection” classes that implement IExtensible<T> but not ICollection<T>? There is at least one, the priority queue. Items can be added, so it is extensible, but the standard remove methods (for instance) make little sense: Items are organized by comparison, but several distinct items may compare equal (by having the same priority), so how can one specify precisely which one to remove? Sections 1.4.12 and 6.12 explain the capabilities of our priority queues.

1.4.5 Sequenced collections

A *sequenced collection* implements interface ISequenced<T> and is a collection whose items are kept in some order. The order may be determined either by the order in which the items were inserted, or by a comparer on the items themselves. The former is the case for array lists and linked lists, and the latter is the case for sorted arrays, tree sets, tree bags and tree dictionaries.

Hash sets, hash bags and hash dictionaries are unsequenced and do not keep their items in any such order.

Sequenced collections also differ from (unsequenced) collections in what notions of equality are meaningful. Unsequenced collections should be compared only using unsequenced equality comparers; see section 2.3. Sequenced collections can be compared by sequenced as well as unsequenced equality comparers: The sequenced collections $< 2,3 >$ and $< 3,2 >$ are equal under unsequenced equality and different under sequenced equality. Unsequenced equality comparison is efficient on all C5 collection implementations, but allocates some auxiliary data when used on collections that does not have fast item membership test.

A sequenced collection is directed (section 1.4.2): it permits enumeration of its items in sequence order and in reverse sequence order.

1.4.6 Indexed collections

An *indexed collection* implements interface IIndexed<T> and is a sequenced collection whose items can be accessed, updated, removed and enumerated by index, that is, by position $(0,1,\ldots)$ within the sequenced collection. In addition is describes methods for finding the least or greatest index at which a given item occurs, and an indexer for obtaining a directed collection value representing the subsequence in a particular index range. Primary implementations are array lists, linked lists, sorted arrays, and ordered trees (but not hash sets and hash bags).

1.4.7 Sorted collections and indexed sorted collections

A *sorted collection* implements interface ISorted<T>. It keeps the items in the order specified by an explicit comparer or by the items being comparable. A sorted collection provides methods to find or delete the least and greatest items, to find the (weak) successor or predecessor item of a given item, to obtain a directed enumerable corresponding to a particular item value range, and to remove all items in a particular item value range. The primary implementations of this interface are sorted arrays (section 6.7) and ordered trees (sections 6.8 and 6.9).

An indexed sorted collection implements interface IIndexedSorted<T> and is a collection that is both indexed (items can be accessed and inserted by index) and sorted (items are kept and enumerated in an order determined by a comparer). The primary implementations of this interface are sorted arrays (section 6.7) and ordered trees (sections 6.8 and 6.9).

1.4.8 Persistent sorted collections

A *persistent sorted collection* implements interface IPersistentSorted<T> and is a sorted collection of which one can efficiently make a read-only snapshot, or a "copy" that is not affected by updates to the original collection. It describes a method `Snapshot()` that returns a sorted collection with exactly the same items as the persistent sorted collection. The TreeSet<T> and TreeBag<T> collection classes are persistent sorted collections; their `Snapshot()` methods take constant time, but subsequent updates to the given tree take more time and space. Nevertheless, this is usually far better than making a copy of the entire collection (tree).

1.4.9 Stacks: last in, first out

A *stack* implements interface IStack<T> and is a directed collection that behaves like a stack, that is, a last-in-first-out data structure. It has methods to push an item onto the stack top and to pop (get and remove) from the stack the most recently pushed item; that is the item on the stack top. In addition, it has an indexer that addresses relative to the bottom of the stack. Primary implementations include circular queues (section 6.1), array lists (section 6.2), and linked lists (section 6.3).

1.4.10 Queues: first in, first out

A *queue* implements interface IQueue<T> and is a directed collection that behaves like a queue, that is, a first-in-first-out data structure. It has methods to enqueue an item at the end of the queue and to dequeue (get and remove) from the queue the oldest of the remaining items; that is the item at the front of the queue. In addition, it has an indexer that addresses relative to the queue's front. Primary implementations include linked lists and CircularQueue<T> (section 6.1).

1.4.11 Lists and views

A *list* implements interfaces IList<T> and SCG.IList<T> and is an indexed collection and in addition is a queue and a stack. It has a read-write property `FIFO` that indicates whether the `Add` and `Remove` methods give queue behavior (true) or stack behavior (false), and properties to get the first and last items. It has methods to insert or remove items first, last, at a specified index, and at a specified list view. It has methods to reverse the items in the list, to sort them, to shuffle them using pseudo-random permutation, and to test whether items are sorted according to a specified order. It has a method to create a new list containing only those items satisfying a given predicate (of type Fun<T,bool>), and methods to create a new list by applying a delegate (of type Fun<T,V>) to all items in the given lists.

Indexed deletions from a list happen *at an item* in the list, whereas indexed insertion happens *at an inter-item space* in the list. Hence there are n possible deletion-points but $n+1$ possible insertion-points in an n-item list.

From a list one can create a *view* of a possibly empty contiguous segment of the list. A list can support multiple updatable views at the same time. A view refers to the items of the underlying list, any updates to those items are immediately visible through the view, and any updates through a view are immediately visible on the underlying list. A view can be slid left or right by a specified number of items, and can be extended or shrunk. There are methods to return the first (or last) one-item view that contains a specified item x. This is especially convenient and efficient if the concrete collection has fast lookup, as for hash-indexed array lists and hash-indexed linked lists, as subsequent operations can refer to that view rather than to list indexes. A view supports all the operations of a list, so one can access, update, remove and insert the items of a view by index, one can enumerate the items of a view (backwards if desired), insert or remove items in the view, and so on. In particular, a one-item view can be used to point at an individual item of a list, and a zero-item view can be used to point between (or before or after) items in a list. Views and their use are described in more detail in section 8.1.

The primary list implementations include array lists (section 6.2), linked lists (section 6.3), hash-indexed versions of these (sections 6.4 and 6.5), and wrapped arrays (section 6.6) which are array lists created from a given underlying array.

Although the C5 list classes implement SCG.IList<T>, some of their methods throw different exceptions than mandated by the SCG.IList<T>. This discrepancy exists already for some of the CLI and .NET implementations of SCG.IList<T>.

1.4.12 Priority queues and priority queue handles

A *priority queue* implements interface IPriorityQueue<T> and is an extensible collection that can hold comparable items and permits efficient retrieval (and removal) of the least item and greatest item. This is useful for implementing scheduling of events and processes in simulation systems, and in many algorithms to find the next object (graph node, path, configuration, and so on) to process.

A priority queue does not permit retrieval, removal or update of items by item identity. Instead a *priority queue handle* of type IPriorityQueueHandle<T> can be associated with an item when it is inserted into the priority queue, and then this handle can be used to efficiently retrieve or remove that item, or to update it as long as it is in the priority queue. See the example in section 11.10.

1.4.13 Sets

There is no interface describing plain sets with item type T. This is because the methods described by ICollection<T> and its base interfaces IExtensible<T> and ICollectionValue<T> are sufficient to implement sets, provided the `AllowsDuplicates` property is false, as for HashSet<T> (section 6.10) and TreeSet<T> (section 6.8).

1.4.14 Bags or multisets

There is no interface describing just bags (multisets) with item type T. This is because the methods described by ICollection<T> and its base interfaces IExtensible<T> and ICollectionValue<T> are sufficient to implement bags, provided the `AllowsDuplicates` property is true. Bag implementations include HashBag<T> (section 6.11) and TreeBag<T> (section 6.9).

When representing bags, the `DuplicatesByCounting` property is important. When the property is true, as for HashBag<T> and TreeBag<T>, then the collection does not store copies of duplicate items, but simply keep count of the number of duplicates. When the property is false, then the collection stores the actual duplicates of each item.

The former approach is necessary when there may be millions of duplicates of a given item; it is preferable in general because it prevents the collection from keeping objects live; and it loses no information when the items are simple types. The latter approach is advantageous or necessary when equal items still have some form of distinct identity.

To illustrate the difference, the shelves of a library may be considered a bag of books since the library may have several copies of a book. The equals function or comparer should consider the copies of a book equal as items. When duplicates are kept by counting, there is no way to distinguish one copy of a book from another; we only know the number of copies currently present. When duplicates are not kept by counting, we know exactly which book objects are present, which may be useful because each copy has its own history of who has borrowed it, and a borrower may have written an important note in the margin in a particular copy.

1.4.15 Summary of collection interfaces and members

Figures 1.2 and 1.3 show properties and methods for each collection interface.

Name	Kind	SCG.IEnumerable<T>	IDirectedEnumerable<T>	ICollectionValue<T>	IDirectedCollectionValue<T>	IExtensible<T>	ICollection<T>	ISequenced<T>	IIndexed<T>	ISorted<T>	IIndexedSorted<T>	IQueue<T>	IStack<T>	IList<T>	IPersistentSorted<T>	IPriorityQueue<T>
`GetEnumerator`	m	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
`Backwards`	m	-	+	-	+	-	-	+	+	+	+	+	+	+	+	-
`Direction`	p	-	+	-	+	-	-	+	+	+	+	+	+	+	+	-
`ActiveEvents`	p	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`All`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`Apply`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`Choose`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`CopyTo`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`Count`	p r	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`CountSpeed`	p r	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`Exists`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`Filter`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`Find`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`IsEmpty`	p r	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`ListenableEvents`	p	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`ToArray`	m	-	-	+	+	+	+	+	+	+	+	+	+	+	+	+
`FindLast`	m	-	-	-	+	-	-	+	+	+	+	+	+	+	+	-
`Add`	m	-	-	-	-	+	+	+	+	+	+	-	-	+	+	+
`AddAll`	m	-	-	-	-	+	+	+	+	+	+	-	-	+	+	+
`AllowsDuplicates`	p r	-	-	-	-	+	+	+	+	+	+	-	-	+	+	+
`Clone`	m	-	-	-	-	+	+	+	+	+	+	-	-	+	+	+
`DuplicatesByCounting`	p r	-	-	-	-	+	+	+	+	+	+	-	-	+	+	+
`EqualityComparer`	p r	-	-	-	-	+	+	+	+	+	+	-	-	+	+	+
`IsReadOnly`	p r	-	-	-	-	+	+	+	+	+	+	-	-	+	+	+
`Clear`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`Contains`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`ContainsAll`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`ContainsCount`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`ContainsSpeed`	p r	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`Find`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`FindOrAdd`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`GetUnsequencedHashCode`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`ItemMultiplicities`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`Remove`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`RemoveAll`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`RemoveAllCopies`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`RetainAll`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`UniqueItems`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`UnsequencedEquals`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`Update`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`UpdateOrAdd`	m	-	-	-	-	-	+	+	+	+	+	-	-	+	+	-
`GetSequencedHashCode`	m	-	-	-	-	-	-	+	+	+	+	-	-	+	+	-
`SequencedEquals`	m	-	-	-	-	-	-	+	+	+	+	-	-	+	+	-
`FindIndex`	m	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`FindLastIndex`	m	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`IndexingSpeed`	p r	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`IndexOf`	m	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`LastIndexOf`	m	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`RemoveAt`	m	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`RemoveInterval`	m	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`this[i]`	i r	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`this[i,n]`	i r	-	-	-	-	-	-	-	+	-	+	-	-	+	+	-
`Dequeue`	m	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-
`Enqueue`	m	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-
`this[i]`	i r	-	-	-	-	-	-	-	-	-	-	+	-	-	-	-
`Pop`	m	-	-	-	-	-	-	-	-	-	-	-	+	-	-	-
`Push`	m	-	-	-	-	-	-	-	-	-	-	-	+	-	-	-
`this[i]`	i r	-	-	-	-	-	-	-	-	-	-	-	+	-	-	-

Figure 1.2: Properties (p), methods (m) and indexers (i) in collection interfaces. For properties and indexers, r=read-only and rw=read-write. Part 1: The interfaces SCG.IEnumerable<T>, IDirectedEnumerable<T>, ICollectionValue<T>, IDirectedCollectionValue<T>, IExtensible<T>, ICollection<T>, ISequenced<T>, IIndexed<T>, IQueue<T> and IStack<T>.

Name	Kind	SCG.IEnumerable<T>	IDirectedEnumerable<T>	ICollectionValue<T>	IDirectedCollectionValue<T>	IExtensible<T>	ICollection<T>	ISequenced<T>	IIndexed<T>	ISorted<T>	IIndexedSorted<T>	IQueue<T>	IStack<T>	IList<T>	IPersistentSorted<T>	IPriorityQueue<T>
`FIFO`	p rw	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`First`	p r	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`FindAll`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Insert`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`InsertAll`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`InsertFirst`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`InsertLast`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`IsFixedSize`	p r	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`IsSorted`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`IsValid`	p r	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Last`	p r	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`LastViewOf`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Map`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Offset`	p r	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Remove`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`RemoveFirst`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`RemoveLast`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Reverse`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Shuffle`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Slide`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Sort`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`this[i]`	i rw	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`TrySlide`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`Underlying`	p r	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`View`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`ViewOf`	m	-	-	-	-	-	-	-	-	-	-	-	-	+	-	-
`AddSorted`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`Comparer`	p r	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`Cut`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`DeleteMax`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`DeleteMin`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`FindMax`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`FindMin`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`Predecessor`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`RangeAll`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`RangeFrom`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`RangeFromTo`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`RangeTo`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`RemoveRangeFrom`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`RemoveRangeFromTo`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`RemoveRangeTo`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`Successor`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`WeakPredecessor`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`WeakSuccessor`	m	-	-	-	-	-	-	-	-	+	+	-	-	-	+	-
`CountFrom`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`CountFromTo`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`CountTo`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`FindAll`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`Map`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`RangeFrom`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`RangeFromTo`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`RangeTo`	m	-	-	-	-	-	-	-	-	-	+	-	-	-	-	-
`Snapshot`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	+	-
`Add`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`Comparer`	p r	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`Delete`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`DeleteMax`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`DeleteMin`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`Find`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`FindMax`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`FindMin`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`Replace`	m	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+
`this[h]`	i rw	-	-	-	-	-	-	-	-	-	-	-	-	-	-	+

Figure 1.3: Properties (p), methods (m) and indexers (i) in collection interfaces. For properties and indexers, r=read-only and rw=read-write. Part 2: The interfaces IList<T>, ISorted<T>, IIndexedSorted<T>, IPersistentSorted<T>, and IPriorityQueue<T>.

1.5 Dictionary interfaces

A dictionary is a special kind of collection that associates a value with a given key, much in the same way an array associates a value with an integer (the index), so a dictionary is sometimes called an *associative array*. However, in a dictionary, keys need not be integers but can have any type that has a hash function or a comparer. The (small) hierarchy of dictionary interfaces is shown in figure 1.4.

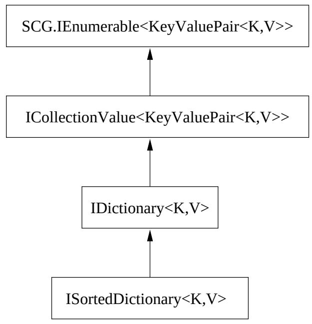


Figure 1.4: The dictionary interface hierarchy.

The two dictionary interfaces are:

- An IDictionary<K,V> is a dictionary with keys of type K and values of type V, where no particular enumeration order can be assumed.

- ISortedDictionary<K,V> is a dictionary with keys of type K and values of type V, where the keys are ordered by a comparer, so that enumeration happens in the order determined by the key comparer.

1.5.1 IDictionary<K,V>

A dictionary implements interface IDictionary<K,V> where K is the type of keys in the dictionary and V is the type of values (or data items) associated with the keys. A dictionary may be thought of as a collection of (key,value) pairs, but there can be only one value associated with a given key.

A dictionary supports addition of a new (key,value) pair, retrieval of the value associated with a given key, update of the value associated with a given key, removal of the (key,value) pair for a given key, and some efficient and convenient combinations of these operations.

A dictionary permits enumeration of all its (key,value) pairs, where a (key,value) pair is represented by type KeyValuePair<K,V>, described in section 3.5. A dictionary also permits separate enumeration of its keys and values, by means of read-only properties `Keys` and `Values` of type ICollectionValue<K> and ICollectionValue<V> respectively.

A dictionary may be *hash-based* if there is an appropriate hash function and equality predicate for the key type, or *sorted* (section 1.5.2) if there is an appropriate comparer. The primary implementation of hash-based dictionaries is HashDictionary<K,V>; see section 7.1.

1.5.2 ISortedDictionary<K,V>

A sorted dictionary implements interface ISortedDictionary<K,V> where K is the type of keys in the dictionary and V is the type of values (or data items) associated with the keys. A sorted dictionary needs an appropriate comparer for the key type K. This comparer may be explicitly given when the dictionary is created or it may be the key type's natural comparer.

A sorted dictionary supports all the operations of a dictionary (section 1.5.1 and in addition permits finding the (key,value) pair whose key is the predecessor or successor of a given key. Enumeration of the (key,value) pairs or of the keys or the values in a sorted dictionary is done in the order determined by the comparer.

The primary implementation of sorted dictionaries is TreeDictionary<K,V>; see section 7.2.

1.6 Comparison with other collection libraries

Figure 1.6 compares C5 and other collection libraries.

	C5	.NET SCG	PowerCollections	Java java.util	Smalltalk
			Collection implementations		
Array list	ArrayList	List	(SCG.List)	ArrayList	OrderedCollection
Linked list	LinkedList	(LinkedList)	(SCG.LinkedList)	LinkedList	LinkedList
Hash-indexed array list	HashedArrayList	(none)	(none)	(none)	(none)
Hash-indexed linked list	HashedLinkedList	(none)	(none)	(LinkedHashSet)	(none)
Fast-index linked list	(none)	(none)	BigList	(none)	(none)
Hash set	HashSet	(Dictionary)	Set	HashSet	Set
Hash bag	HashBag	(Dictionary)	Bag	(HashMap to ints)	Bag
Order-based set	TreeSet	(SortedDictionary)	OrderedSet	TreeSet	SortedCollection
Order-based bag	TreeBag	(SortedDictionary)	OrderedBag	(TreeMap to ints)	SortedCollection
Stack	ArrayList	Stack	(SCG.Stack)	Stack	OrderedCollection
Queue	CircularQueue	Queue	Dequeue	Queue	OrderedCollection
Priority queue	IntervalHeap	(none)	(none)	PriorityQueue	(none)
			Dictionary implementations		
Hash-based dictionary	HashDictionary	Dictionary	(SCG.Dictionary)	HashMap	Dictionary
Same, multivalued	(dictionary of sets)	(none)	MultiDictionary	(HashMap of sets)	(dictionary of sets)
Ordered tree dictionary	TreeDictionary	SortedDictionary	OrderedDictionary	TreeMap	(none)
Same, multivalued	(dictionary of sets)	(none)	OrderedMultiDictionary	(TreeMap of sets)	(none)
Ordered array dictionary	(none)	SortedList	(none)	(none)	(none)
			Other features		
Range queries	Yes	No	Yes	Yes	No
Mutable list views	Yes	No	No	No	No
Priority queue handles	Yes	No	No	No	No
Collection snapshots	Yes	No	No	No	No
Listenable events	Yes	No	No	No	No
Collection formatting	Yes	No	No	No	No

Figure 1.5: A summary comparison of C5 and other collection libraries: The generic collection classes of the .NET Framework Library [12, 22], Peter Golde's PowerCollections library [15], the Java collection library [3], and the Smalltalk 80 collection library [14].

Chapter 2

Equality and comparison

Any collection that supports search for a given item, for instance via the `Contains` method, must be able to determine whether two items are equal. Similarly, a dictionary must be able to determine whether two keys are equal.

Every collection class in the C5 library implements this equality test in one of two ways: either the class has an *item equality comparer*, or it has an *item comparer*, also called an item ordering.

- An *item equality comparer* is an object of type SCG.IEqualityComparer<T>, which uses an equality predicate `bool Equals(T, T)` to determine whether two items are equal and a hash function `int GetHashCode(T)` to quickly determine when they are definitely not equal. Thus a hash function is basically an approximation to the equality predicate that can be computed quickly.

- An *item comparer* is an object of type SCG.IComparer<T>, which uses an ordering relation `int Compare(T, T)` to determine whether one item is less than, equal to, or greater than another.

Collections such as LinkedList<T>, HashSet<T>, HashedLinkedList<T> and so on that use an item equality comparer are said to be *hash-based*. Collections such as TreeSet<T> that use an item comparer are said to be *comparison-based*. Figure 2.1 show which of C5's collection classes and dictionary classes use an equality comparer and which use a comparer.

A collection's item equality comparer or item comparer is fixed when the collection instance is created. Either it is given explicitly as an argument to the constructor that creates the collection instance, or it is manufactured automatically by the C5 library. In the latter case, the interfaces implemented by the item type T determine what item equality comparer or item comparer is created for the collection; see sections 2.4 and 2.7.

Similarly, every dictionary either uses a *key equality comparer* or a *key comparer* to determine when two keys are equal. Either it is given explicitly as an argument

Class	Equality comparer	Comparer
CircularQueue<T>	−	−
ArrayList<T>	+	−
LinkedList<T>	+	−
HashedArrayList<T>	+	−
HashedLinkedList<T>	+	−
WrappedArray<T>	+	−
SortedArray<T>	(+)	+
TreeSet<T>	(+)	+
TreeBag<T>	(+)	+
HashSet<T>	+	−
HashBag<T>	+	−
IntervalHeap<T>	−	+
HashDictionary<K,V>	+	−
TreeDictionary<K,V>	(+)	+

Figure 2.1: Collection classes needing equality comparers or comparers. Some collections take an optional (+) equality comparer, to determine equality and hashing when using the entire collection as an item in an "outer" collection.

to the constructor that creates the dictionary instance, or it is manufactured automatically by the C5 library. In the latter case, the interfaces implemented by the key type K determine what key equality comparer or key comparer is created for the dictionary; see sections 2.4 and 2.7.

2.1 Natural equality

In C#, all types inherit from class Object, and therefore every item type has the following two methods which define the type's *natural equality*:

- `bool Equals(Object y)` compares the item to some other object.
- `int GetHashCode()` finds the item's hash code.

For primitive value types such as `int`, natural equality is based on the values. For user-defined struct types, default natural equality compares two struct values field by field. For type String, natural equality compares the contents of the strings. For other reference types, default natural equality compares the object references.

Note that when the argument has value type, a call to `bool Equals(Object)` causes boxing of the argument, which is slow. Also, default natural equality for struct types uses reflection, which is slow.

2.2 Equatable types

Interface IEquatable<T> from CLI namespace System describes a single method:

- `bool` **`Equals`**`(T x)` returns true if this item equals item `x`.

This is different from the `Equals` method inherited from Object, whose argument type is Object, not T. Implementations of System.IEquatable<T> must satisfy:

- The equality function is
  reflexive so `x.Equals(x)` is true;
  symmetric so `x.Equals(y)` implies `y.Equals(x)`; and
  transitive so `x.Equals(y)` and `y.Equals(z)` implies `x.Equals(z)`.
- The type's hash function `GetHashCode()`, possibly inherited from Object, is an *approximation of equality*:
  If `x.Equals(y)` then `x.GetHashCode()==y.GetHashCode()`.
- The equality function and the hash function must be *total*: they must never throw exceptions, not even on a `null` reference argument. Throwing an exception in these methods may corrupt the internal state of the collection.

Note that it is difficult to satisfy these requirements both for a class and for its derived classes. See Bloch [6, chapter 3] for a discussion.

Many built-in types in C#, such as `long`, do implement IEquatable<`long`>, and hence support equality comparison with themselves.

2.3 Equality comparers

Interface IEqualityComparer<T> from CLI namespace System.Collections.Generic describes two methods:

- `bool` **`Equals`**`(T x, T y)` returns true if item `x` equals item `y`, and must be defined for all `x` and `y`, even `null` references.
- `int` **`GetHashCode`**`(T x)` returns the hash code for `x`, and must be defined for all `x`, even `null` references.

Implementations of SCG.IEqualityComparer<T> must satisfy:

- The equality function is
  reflexive so `Equals(x,x)` is true;
  symmetric so `Equals(x,y)` implies `Equals(y,x)`; and
  transitive so `Equals(x,y)` and `Equals(y,z)` implies `Equals(x,z)`.
- The hash function must be an approximation of equality:
  If `Equals(x,y)` then `GetHashCode(x)==GetHashCode(y)`.

- The equality function and the hash function must be *total*: they must never throw exceptions, not even on a `null` reference argument. Throwing an exception in these methods may corrupt the internal state of the collection.

The equality and hash functions should be unaffected by modifications to items, so these functions should depend only on immutable (read-only) fields of items. If an item is modified in a way that affects its equality or hashcode while stored in a collection, then the collection will not work and may fail in arbitrary ways; see anti-pattern 130.

2.4 Creating equality comparers

The static class C5.EqualityComparer<T> is a factory class, used to produce a default equality comparer for a given type T. The class has one static property:

- `static EqualityComparer<T>.`**`Default`** of type SCG.IEqualityComparer<T> is the default equality comparer for type T; see figure 2.2. For a given type `T`, every evaluation of `EqualityComparer<T>.Default` will return the same object.

When creating a hash-based collection such as a hash set, hash bag or hash dictionary, one may give an explicit equality comparer of type SCG.IEqualityComparer<T> that will be used when hashing items and when comparing them for equality. If one does not give such an equality comparer explicitly, then a standard one returned by `EqualityComparer<T>.Default` will be used by the collection, as shown in figure 2.2. This standard equality comparer is created once for each type T, so multiple calls to the `EqualityComparer<T>.Default` property for a given T will return the same object.

Item type T	Value of `EqualityComparer<T>.Default`
`byte`	ByteEqualityComparer
`char`	CharEqualityComparer
`double`	DoubleEqualityComparer
`int`	IntEqualityComparer
implements ISequenced<W>	SequencedCollectionEqualityComparer<T,W>
implements ICollectionValue<W>	UnsequencedCollectionEqualityComparer<T,W>
implements IEquatable<T>	EquatableEqualityComparer<T>
does not implement IEquatable<T>	NaturalEqualityComparer<T>

Figure 2.2: Finding the default hash function for various types. The alternatives are tried in order from top to bottom, and the first match is used. W is any type.

When creating a hash-based collection of collections, an equality comparer for the “outer” collection’s items (which are themselves “inner” collections) should be created and given explicitly; see section 8.3 and 9.17.

The equality comparer classes listed in figure 2.2 define item equality and item hash functions that satisfy the requirements on `Equals(T,T)` and `GetHashCode(T)` mentioned above, and behave as follows:

- The classes ByteEqualityComparer, CharEqualityComparer, DoubleEqualityComparer IntEqualityComparer, and so on define equality as the built-in equality for those types, with the corresponding hash functions.

- Otherwise, when the item type T implements ISequenced<W>, so each outer item is a sequenced collection of inner items of type W, the default outer item comparer is a SequencedCollectionEqualityComparer<T,W>. This equality comparer implements interface IEqualityComparer<T> and defines the equality `Equals(T,T)` of two collections by comparing their items of type W, using the `Equals(W,W)` method of their equality comparers, in the order in which they appear in the collections. Correspondingly, it defines the hash function `GetHashCode(T)` for a collection using the `GetHashCode(W)` method of their equality comparers in a way that depends on the order in which the items appear.

- Otherwise, when the item type T implements ICollection<W>, so each outer item is an unsequenced collection of inner items of type W, the default outer item comparer is an UnsequencedCollectionEqualityComparer<T,W>. This equality comparer implements interface SCG.IEqualityComparer<T> and defines the equality `Equals(T,T)` of two collections by comparing their items using the `Equals(W,W)` method of their equality comparers, disregarding the order in which the items appear. Correspondingly, it defines the hash function `GetHashCode(T)` of a collection using the `GetHashCode(W)` method of their equality comparers in a way that does not depend on the order in which the items appear.

- An EquatableEqualityComparer<T>, where T must implement IEquatable<T>, uses T's method `IEquatable<T>.Equals(T)` to define equality, and it uses T's `GetHashCode()` as hash function.

- A NaturalEqualityComparer<T> uses T's method `Equals(Object)` to define equality and uses T's `GetHashCode()` method compute hash codes.

  If T is a value type, these methods may have been overridden in type T or else are the default methods inherited from System.ValueType. In the latter case, the `Equals(Object)` method uses reflection to compare the fields of the value for equality, and the `GetHashCode()` method similarly uses one or more fields of the value type to compute a hash value. For efficiency and correctness it is advisable to override `Equals(Object)` and `GetHashCode()` in a value type T.

  If T is a reference type, these methods may have been overridden in type T or a base class, or else are the default methods inherited from class Object. The default `Equals(Object)` method from Object for a reference type simply compares object references.

- A ReferenceEqualityComparer<T>, where T must be a reference type, defines equality for reference type T as object reference equality, using class Object's static method `ReferenceEquals(Object,Object)`, and defines its hash code using the original `GetHashCode()` method from class Object, even if `GetHashCode()` has been overridden by T or one of its base classes.

- A KeyValuePairEqualityComparer<K,V> implements the interface IEqualityComparer<KeyValuePair<K,V>>, defines equality of two (key,value) pairs as equality of their keys, and defines the hash code of the (key,value) pair as the hash code of the key.

The `Equals(Object)` and `GetHashCode()` methods for a collection class are the standard ones inherited from class Object. In particular, `coll1.Equals(coll2)` tests whether collections `coll1` and `coll2` are the same object reference, not whether they contain the same items.

To test whether collections `coll1` and `coll2` contain the same items in some order, use `UnsequencedCollectionEqualityComparer<T,W>.Equals(coll1, coll2)`. Here T is the type of the collections `coll1` and `coll2`, and W is their item type. To test whether sequenced collections `coll1` and `coll2` contain the same items in the same order, use `SequencedCollectionEqualityComparer<T,W>.Equals(coll1, coll2)`.

2.5 Comparable types

Interface IComparable<T> from CLI namespace System describes a single method:

- `int` **`CompareTo`**`(T y)` returns a negative number if the given value is less than `y`, zero if it is equal to `y`, and a positive number if it is greater than `y`.

To describe the requirements on method `CompareTo`, let us define that "−" and "+" are the opposite signs of each other and that 0 is the opposite sign of itself. Then the `CompareTo` must satisfy:

- As an ordering relation it must be *reflexive* so that `x.CompareTo(x)` is zero.
- As an ordering relation it must be *transitive* so if `x.CompareTo(y)` has a given sign, and `y.CompareTo(z)` have the same sign or is zero, then `x.CompareTo(z)` has the same sign.
- As an ordering relation it must be *anti-symmetric* so `x.CompareTo(y)` and `y.Compare(x)` must have opposite signs.
- It must be *total* so it must never throw an exception. Throwing an exception in `CompareTo` may corrupt the internal state of a collection.

The result of `CompareTo` should be unaffected by modifications to items. This can be ensured by letting the comparer depend only on immutable (read-only) fields of items. If an item is modified in a way that affects the results of comparisons while it is stored in a collection, then the collection will not work properly and may fail in arbitrary ways.

Many built-in types in C#, such as `long`, do implement IComparable<`long`>, and hence support comparison with themselves.

Interface System.IComparable is a legacy version of IComparable<T>; it describes a single method:

- `int` **`CompareTo`**`(Object y)` returns a negative number if the given value is less than `y`, zero if it is equal to `y`, and a positive number if it is greater than `y`.

Use IComparable<T> whenever possible for better type safety, and to avoid the overhead of boxing when T is a value type.

2.6 Comparers

Interface IComparer<T> from CLI namespace System.Collections.Generic describes one method:

- `int` **`Compare`**`(T x, T y)` must return a negative number when `x` is less than `y`, zero when they are equal, and a positive number when `y` is greater than `x`.

To describe the requirements on method `Compare`, let us define that "–" and "+" are the opposite signs of each other and that 0 is the opposite sign of itself. Then the `Compare` method in an implementation of SCG.IComparer<T> must satisfy:

- As an ordering relation it must be *reflexive* so that `Compare(x,x)` is zero.
- As an ordering relation it must be *transitive* so if `Compare(x,y)` has a given sign, and `Compare(y,z)` have the same sign or is zero, then `Compare(x,z)` has the same sign.
- As an ordering relation it must be *anti-symmetric* so `Compare(x,y)` and `Compare(y,x)` must have opposite signs.
- It must be *total* so it must never throw an exception. Throwing an exception in a comparer may corrupt the internal state of a collection.

A comparer should be unaffected by modifications to items. This can be ensured by letting the comparer depend only on immutable (read-only) fields of items. If an item is modified in a way that affects the results of comparisons while it is stored in a collection, then the collection will not work properly and may fail in arbitrary ways.

2.7 Creating comparers

The static class C5.Comparer<T> is a factory. One cannot create object instances of it, but can use it to produce a default comparer (ordering) for a given type T. The class has one static property:

- `static Comparer<T>.`**`Default`** of type SCG.IComparer<T> is the default comparer for type T; see figure 2.3.

When creating a sorted collection of collections, a comparer for the "outer" collection's items (which are themselves "inner" collections) must often be created explicitly; see section 8.3 and 9.17. Since every evaluation of `Comparer<T>.Default` for a given `T` returns the same comparer object, reference comparison of comparers can be used to check whether two (inner) collections have the same comparers, provided the comparers were created by the Comparer<T> class.

Type T	`Comparer<T>.Default`
implements System.IComparable<T>	NaturalComparer<T>
implements System.IComparable	NaturalComparerO<T>

Figure 2.3: Default comparers for various types.

The comparer classes listed in figure 2.3 define comparer methods that behave as follows:

- A NaturalComparer<T> simply uses the `Compare(T,T)` method from type T, which must implement System.IComparable<T>; otherwise NotComparableException is thrown.

- A NaturalComparerO<T> simply uses the `Compare(Object,Object)` method from type T, which must implement System.IComparable; otherwise NotComparableException is thrown.

A KeyValuePairComparer<K,V> implements SCG.IComparer<KeyValuePair<K,V>> and defines comparison of two (key,value) pairs by comparison of their keys only. The class has a single constructor:

- `KeyValuePairComparer<K,V>(SCG.IComparer<K> cmp)` creates a new (key, value) pair comparer whose `Compare(K x,K y)` method just calls `cmp.Compare(x,y)` to compare the keys of two (key, value) pairs.

A DelegateComparer<T> encapsulates a delegate of type System.Comparison<T> and uses that to implement SCG.IComparer<T>. For that purpose it has a single constructor:

- `DelegateComparer<T>(System.Comparison<T> csn)` creates a new comparer whose `Compare(T x,T y)` method just calls the delegate `csn(x,y)` and returns its result. The given delegate `csn` must satisfy the requirements on a comparer mentioned above. In particular, it must never throw an exception.

This is useful for creating comparers inline, for instance when creating a sorted collection or when sorting arrays or lists. For an example, see pattern 92.

Chapter 3

Auxiliary types

In addition to the interfaces describing the collection concepts and dictionary concepts, and the classes implementing the collections and dictionaries, there are several auxiliary enum types, struct types, delegate types and exception types. Additional auxiliary types that are primarily of interest to library developers are described in chapter 14.

3.1 Enum type EventTypeEnum

The enum type EventTypeEnum is used to report which event handlers are currently associated with a collection or dictionary, and to report which events can be listened to at all. See section 8.8.5 for events and event handlers, and see properties `ActiveEvents` and `ListenableEvents` in ICollectionValue<T> (section 4.2). The enum type has the following values which can be combined, like flags, using the bitwise operator "or" (|) to form other values of type EventTypeEnum:

Enum value	Corresponding events
`Added`	ItemsAdded
`All`	`Basic` | `Inserted` | `RemovedAt`
`Basic`	`Added` | `Changed` | `Cleared` | `Removed`
`Changed`	CollectionChanged
`Cleared`	CollectionCleared
`Inserted`	ItemInserted
`None`	None
`Removed`	ItemsRemoved
`RemovedAt`	ItemRemovedAt

For instance, to test whether a CollectionChangedHandler is associated with collection `coll`, evaluate `(coll.ActiveEvents & EventTypeEnum.Changed) != 0`. To test for no event handler, evaluate `coll.ActiveEvents == EventTypeEnum.None`.

3.2 Enum type EnumerationDirection

The enum type EnumerationDirection is used to report the enumeration direction of a directed collection or directed enumerable; for instance by property `Direction` in interface IDirectedEnumerable<T> (section 4.4). The enum type has two values which are mutually exclusive:

Enum value	Meaning
`Forwards`	The natural enumeration order
`Backwards`	The opposite of the natural enumeration order

3.3 Enum type Speed

Enum type Speed has values `Constant`, `Log`, `Linear` and `PotentiallyInfinite` and is used internally in the C5 library. This type and the properties `ContainsSpeed` (section 4.1), `CountSpeed` (section 4.2) and `IndexingSpeed` (section 4.6) are used to determine the most efficient way to test equality of two collections.

For instance, if one needs to determine whether collections `coll1` and `coll2` are equal, and `coll1.ContainsSpeed` is `Linear` and `coll2.ContainsSpeed` is `Constant`, then it is faster to enumerate all items `x` of `coll1` and test whether `coll2.Contains(x)` than to do the opposite. This is exploited in C5's collection equality comparers synthesized by the C5.Comparer<T> class (section 2.7).

The possible values of enum type Speed are:

- **`PotentiallyInfinite`** means that the operation may not terminate. For instance, counting the number of items in a hypothetical lazily generated and potentially infinite collection may not terminate. Note that all operations on C5's collection classes or dictionary classes do terminate.

- **`Linear`** means that the operation takes time $O(n)$, where n is the size of the collection. For instance, this is the case for item lookup with `Contains(x)` on non-hashed lists.

- **`Log`** means that a the operation takes time $O(\log n)$, where n is the size of the collection. For instance, this is the case for item lookup with `Contains(x)` on tree sets, tree bags, and sorted arrays.

- **`Constant`** means that the operation takes time $O(1)$, that is, constant time. For instance, this is the case for item indexing `this[i]` on array lists, and is the expected time for item lookup with `Contains(x)` on hash sets, hash bags, and hashed lists.

3.4 Record struct types Rec<T1,T2>, ...

In generic function libraries it is convenient to have ways to represent records, such as pairs, triples, and quadruples. This purpose is served by the Rec family of record struct types, declared as shown below.

```
public struct Rec<T1,T2> : IEquatable<Rec<T1,T2>>, IShowable {
  public readonly T1 X1;
  public readonly T2 X2;
  public Rec(T1 x1, T2 x2) {
    this.X1 = x1; this.X2 = x2;
  }
  public override int GetHashCode() { ... }
  public override bool Equals(Object o) { ... }
  public override bool Equals(Rec<T1,T2> that) { ... }
  public static bool operator==(Rec<T1,T2> r1, Rec<T1,T2> r2) { ... }
  public static bool operator!=(Rec<T1,T2> r1, Rec<T1,T2> r2) { ... }
  public bool Show(StringBuilder sb, ref int rest, IFormatProvider fmp) { ... }
}
public struct Rec<T1,T2,T3> : IEquatable<Rec<T1,T2,T3>>, IShowable { ... }
public struct Rec<T1,T2,T3,T4> : IEquatable<Rec<T1,T2,T3,T4>>, IShowable { ... }
```

Note that the fields `X1`, `X2`, ... are public but read-only, to prevent the confusion that results from updating a field of a copy of a struct value, rather than the original struct value.

The `Equals(Object o)` method of a record type returns false when `o` is `null` or is not a boxed instance of the record type; otherwise compares the records field by field using their `Equals` methods.

The `Equals(Rec<T1,T2>)` method in type Rec<T1,T2> and the operators (==) and (!=) compare the record fields using the `Equals` methods for the field types T1 and T2.

3.5 Struct type KeyValuePair<K,V>

A struct of type KeyValuePair<K,V> is used represent pairs of a key of type K and an associated value of type V. It has public fields `Key` and `Value`. It is similar to Rec<K,V>; the main difference is that the fields are called `Key` and `Value` rather than `X1` and `X2`. Also, the formatting by `ToString` is different.

3.6 Delegate types

The delegate type families Act and Fun described below are used by several methods in interface ICollectionValue<T> and elsewhere. For event handler types, which are delegate types too, see section 8.8.

3.6.1 Action delegate types Act<A1>, Act<A1,A2>, ...

Delegate type Act<A1> is the type of functions or methods from A1 to `void`, used to perform some side-effect or action for a given collection item `x`. Type Act<A1> is part of a family of such action delegate types with zero to four arguments, declared as follows:

```
public delegate void Act();
public delegate void Act<A1>(A1 x1);
public delegate void Act<A1,A2>(A1 x1, A2 x2);
public delegate void Act<A1,A2,A3>(A1 x1, A2 x2, A3 x3);
public delegate void Act<A1,A2,A3,A4>(A1 x1, A2 x1, A3 x3, A4 x4);
```

The type Act<T> corresponds to System.Action<T> in in the CLI or .NET class library.

3.6.2 Function delegate types Fun<A1,R>, Fun<A1,A2,R>, ...

Delegate type Fun<A1,R> is the type of functions or methods from A1 to R, used to compute some transformation for a given collection item `x`. Type Fun<A1,R> is part of a family of such function delegate types with zero to four arguments, declared as follows:

```
public delegate R Fun<R>();
public delegate R Fun<A1,R>(A1 x1);
public delegate R Fun<A1,A1,R>(A1 x1, A2 x2);
public delegate R Fun<A1,A2,A3,R>(A1 x1, A2 x2, A3 x3);
public delegate R Fun<A1,A2,A3,A4,R>(A1 x1, A2 x1, A3 x3, A4 x4);
```

The type Fun<A1,R> is the type of *functions* or *converters* from type A1 to type R, and corresponds to System.Converter<A1,R> in the CLI or .NET class library. The type Fun<T,bool> is the type of *predicates* for values of type T and corresponds to type System.Predicate<T> in the CLI or .NET class library.

3.7 Exception types

3.7.1 General exceptions used by C5

- A System.ArgumentException is thrown when the argument to an operation does not satisfy its preconditions. For instance, it is thrown if the enumerable `xs` passed in `AddSorted(xs)` does not produce items in strictly increasing order.
- A System.ArgumentOutOfRangeException is thrown by subrange (view) operations if the given index and length arguments are illegal for the collection.
- A System.IndexOutOfRangeException is thrown by indexers `this[i]` and indexing operations such as `Insert(i, x)` when integer `i` is outside the range of legal indexes for the operation. An indexer that does not in general take integer arguments throws other exceptions: `this[k]` on a dictionary throws NoSuchItemException when key `k` is not in the dictionary, and `this[h]` on a priority queue throws InvalidHandleException when handle `h` is not associated with the priority queue.

3.7.2 Exception types particular to C5

The C5 collection library uses several exception classes to provide more specific information about operations that went wrong. All derive from System.Exception, and the general rules for throwing them are these:

- A CollectionModifiedException is thrown if the collection underlying an enumerator has been modified while the enumerator is in use. The exception is thrown at the first use of the enumerator ("fail-early") after a modification.
- A DuplicateNotAllowedException is thrown when one attempts to add an item `x` to a collection that already contains an item equal to `x`, and that collection does not allow duplicates. Whether a collection allows duplicates can be determined by inspecting the collection's `AllowsDuplicates` property; see section 4.5.
- A FixedSizeCollectionException is thrown when an attempt is made to extend (for instance, by `Add`) or shrink (for instance, by `Remove`) a fixed-size list: one for which the `IsFixedSize` property is true.
- An IncompatibleViewException is thrown when a list operation such as `u.Insert(w, x)` or `u.Span(w)` is applied to a list view `w` (section 8.1) that has a different underlying list than that of `u`.
- An InternalException is thrown in the unlikely event of an inconsistency in the C5 library's internal data structures. The library developers (kokholm@itu.dk and sestoft@itu.dk) would very much appreciate being notified of such problems, preferably with example code that provokes the problem.

- An IntervalHeap.InvalidHandleException is thrown when one attempts to use a priority queue handle (sections 4.10 and 9.20) in an indexer or `Delete` or `Replace` operation on a priority queue, and the handle is not currently associated with this priority queue, or when one attempts to set (re-use) an existing handle in an `Add` operation and the handle is already in use.
- A NoSuchItemException is thrown when no item can be returned from an operation, such as when `Choose()` is applied to an empty collection, or `First` or `Last` are applied to an empty list, or `Predecessor` is applied to a value less than or equal to all items in a sorted collection, or a dictionary indexer is used to look up a key that is not in the collection.
- A NotComparableException is thrown when an attempt is made to construct a NaturalComparer<T> or NaturalComparerO<T> (section 2.6) for a type T that does not implement System.IComparable<T> or System.IComparable. This typically happens when one creates a comparer-based collection without giving an explicit comparer.
- A NotAViewException is thrown when a view operation such as `Slide` or `TrySlide` is applied to a proper list, not a list view (sections 4.8 and 8.1). One can determine whether `list` is a view by evaluating `list.Underlying != null`.
- A ReadOnlyCollectionException is thrown if any modification is attempted on a collection that is read-only, such as a guarded collection or a snapshot of a tree set or tree bag. One can determine whether a collection is read-only by inspecting its `IsReadOnly` property; see section 4.5.
- An UnlistenableEventException is thrown if an attempt is made to attach an event handler to a collection that does not support that event type. Whether a collection supports a given event type can be determined by inspecting the collection's `ListenableEvents` property; see section 4.2.
- A ViewDisposedException is thrown if an operation other than `IsValid` or `Dispose` is called on a list view that has been invalidated. One can determine whether a view has been disposed by inspecting its `IsValid` property; see section 4.8.

3.8 Pseudo-random number generators

A deck of 52 cards can be shuffled (permuted) in $52! \approx 8.1 \cdot 10^{67}$ different ways, but a pseudo-random number generator with a 32 bit seed can generate only $2^{32} \approx 4 \cdot 10^{9}$ of these permutations. Proper shuffling or permutation requires very large seeds and very long periods.

The C5 library provides a pseudo-random number generator in class C5Random which is a subclass of System.Random. The C5Random class provides an implementation of George Marsaglia's "complimentary multiply with carry" (CMC) pseudo-random number generator which has an extremely long period [19, 20]. The implementation in C5 accepts a seed consisting of 16 unsigned 32-bit integers, which gives a period of $2^{16 \cdot 32} = 2^{512} \approx 1.3 \cdot 10^{154}$. Marsaglia's paper [20] includes advice on how to obtain multiple seed values for pseudo-random number generators such as C5Random.

The C5Random class provides the following methods:

- `int` **`Next`**`()` returns a pseudo-random integer.
- `int` **`Next`**`(int max)` returns a pseudo-random integer in the range `0 ...max-1`. Throws ArgumentException if `max < 0`.
- `int` **`Next`**`(int min, int max)` returns a pseudo-random integer in the range `min ...max-1`. Throws ArgumentException if `max <= min`.
- `void` **`NextBytes`**`(byte[] arr)` fills array a with pseudo-random bytes. Throws NullReferenceException if `arr` is `null`.
- `double` **`NextDouble`**`()` returns a pseudo-random floating-point number greater than or equal to 0 and less than 1.

3.9 The IShowable interface

The collection and dictionary classes implement the IShowable interface to permit output-limited formatting of collections and dictionaries. Interface IShowable derives from the System.IFormattable interface. See also section 8.7.

Methods

- `bool` **`Show`**`(StringBuilder sb, ref int rest, IFormatProvider formatProvider)` appends a formatted version of the given collection to `sb`, using at most approximately `rest` characters, and using the `formatProvider` if non-`null` to format collection items. If the collection cannot not be formatted within `rest` characters, then ellipses "..." are used to indicate missing pieces in the resulting output. Subtracts the number of characters actually used from `rest`. Returns true if `rest > 0` on return; otherwise false.

Chapter 4

Collection interface details

This chapter gives a detailed description of the collection interfaces shown in figure 4.1. Since all important functionality of the collection implementation classes is described by the interfaces, this is the main documentation of the entire library (also available online as HTML help files).

The interfaces are presented in alphabetical order in sections 4.1 to 4.14.

The dictionary interfaces are presented separately in chapter 5.

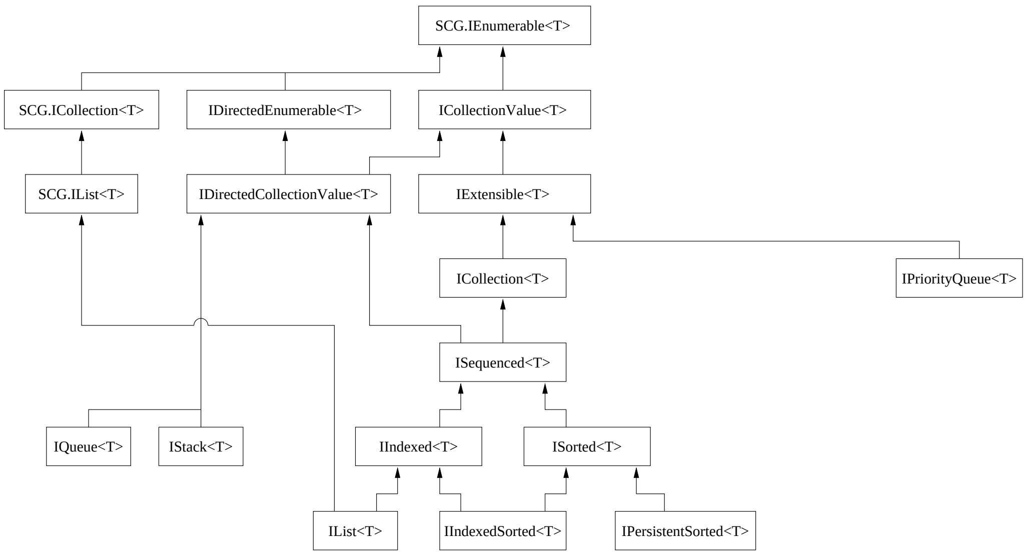


Figure 4.1: The collection interface hierarchy (same as figure 1.1). See figure 6.2 for the classes implementing these interfaces. SGC is the System.Collections.Generic namespace.

4.1 Interface ICollection<T>

Inherits from: IExtensible<T>.

Implemented by: ArrayList<T> (section 6.2), HashBag<T> (section 6.11), HashSet<T> (section 6.10), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5), LinkedList<T> (section 6.3), SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), TreeSet<T> (section 6.8), and WrappedArray<T> (section 6.6).

Properties

- Read-only property EventTypeEnum **ActiveEvents**, see page 46.
- Read-only property bool **AllowsDuplicates**, see page 52.
- Read-only property Speed **ContainsSpeed** is the guaranteed run-time of the Contains method; see section 3.3.
- Read-only property int **Count**, see page 46.
- Read-only property Speed **CountSpeed**, see page 46.
- Read-only property bool **DuplicatesByCounting**, see page 52.
- Read-only property SCG.IEqualityComparer<T> **EqualityComparer**, see page 52.
- Read-only property bool **IsEmpty**, see page 46.
- Read-only property bool **IsReadOnly**, see page 52.
- Read-only property EventTypeEnum **ListenableEvents**, see page 46.

Methods

- bool **Add**(T x), see page 52.
- void **AddAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 53.
- bool **All**(Fun<T,bool> p), see page 46.
- void **Apply**(Act<T> act), see page 46.
- bool **Check**(), see page 53.
- T **Choose**(), see page 47.
- void **Clear**() removes all items from the collection. Raises events CollectionCleared and CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only.

- `Object` **`Clone`**`()`, see page 53.
- `bool` **`Contains`**`(T x)` returns true if the collection contains an item equal to x.
- `bool` **`ContainsAll`**`<U>(SCG.IEnumerable<U> xs) where U:T` returns true if the collection contains (items equal to) all the items in xs. For collections with bag semantics, item multiplicity is taken into account: each item x must appear in the collection at least as many times as in xs. The method is generic so that it can be applied to enumerables with any item type U that is a subtype of T; see section 8.4.
- `int` **`ContainsCount`**`(T x)` returns the number of occurrences of x in the collection; `Contains(x)` is equivalent to, but faster than, `ContainsCount(x)>0`.
- `void` **`CopyTo`**`(T[] arr, int i)`, see page 47.
- `bool` **`Exists`**`(Fun<T,bool> p)`, see page 47.
- `SCG.IEnumerable<T>` **`Filter`**`(Fun<T,bool> p)`, see page 47.
- `bool` **`Find`**`(Fun<T,bool> p, out T res)`, see page 47.
- `bool` **`Find`**`(ref T x)` returns true if the collection contains an item equal to x, and in that case binds one such item to the ref parameter x; otherwise returns false and leaves x unmodified.
- `bool` **`FindOrAdd`**`(ref T x)` returns true if the collection contains an item equal to x, and in that case binds one such item to the ref parameter x; otherwise returns false and adds x to the collection. If the item was added, it raises events ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only.
- `int` **`GetUnsequencedHashCode`**`()` returns the unsequenced or order-insensitive hash code of the collection. This is the sum of a transformation of the hash codes of its items, each computed using the collection's item equality comparer. The collection's hash code is cached and thus not recomputed unless the collection has changed since the last call to this method.
- `ICollectionValue<KeyValuePair<T,int>>` **`ItemMultiplicities`**`()` returns a new collection value whose items are pairs (x, n) where x is an item in the given collection and n is the multiplicity of x in the collection: the number of times x appears. For collections with set semantics, n=1 always, and for collections with bag semantics, n >= 1 always.
- `bool` **`Remove`**`(T x)` attempts to remove an item equal to x from the collection. If the collection has bag semantics, this means reducing the multiplicity of x by one. Returns true if the collection did contain an item equal to x, false if it did not. If an item was removed, it raises events ItemsRemoved and CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only.

- bool **Remove**(T x, out T xRemoved) attempts to remove an item equal to x from the collection. If the collection has bag semantics, this means reducing the multiplicity of x by one. Returns true if the collection did contain an item equal to x and if so it binds one such item to xRemoved; returns false if it did not. If an item was removed, it raises events ItemsRemoved and CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only.

- void **RemoveAll**<U>(SCG.IEnumerable<U> xs) where U:T attempts to remove (items equal to) all items in xs from the collection. If the collection has bag semantics, this means reducing the item multiplicity of x in the collection by at most the multiplicity of x in xs. If any items were removed, then events ItemsRemoved and CollectionChanged are raised. Throws ReadOnlyCollectionException if the collection is read-only. The method is generic so that it can be applied to enumerables with any item type U that is a subtype of T; see section 8.4.

- void **RemoveAllCopies**(T x) attempts to remove all copies of items equal to x from the collection, reducing the multiplicity of x to zero. It has no effect if the collection does not contain an item equal to x. If any items were removed, then events ItemsRemoved and CollectionChanged are raised. Throws ReadOnlyCollectionException if the collection is read-only.

- void **RetainAll**<U>(SCG.IEnumerable<T> xs) where U:T retains every item that is equal to some item in xs. Equivalently, removes from the collection any item y not equal to some item from xs. If the collection has bag semantics, then this reduces the multiplicity of each item x in the collection to the minimum of its multiplicity in the given collection and in xs. If any items were removed, then events ItemsRemoved and CollectionChanged are raised. Throws ReadOnlyCollectionException if the collection is read-only. The method is generic so that it can be applied to enumerables with any item type U that is a subtype of T; see section 8.4.

- T[] **ToArray**(), see page 47.

- ICollectionValue<T> **UniqueItems**() returns a collection value which is the given collection with duplicate items removed. If the given collection allows duplicates, a new collection is created and returned; if not, the given collection is returned. The items of the returned collection are the key components of the collection returned by method ItemMultiplicities.

- bool **UnsequencedEquals**(ICollection<T> coll) returns true if this collection contains the same items as coll with the same multiplicities, but possibly in a different order. More precisely, for each item in this collection there must be one equal to it in coll with the same multiplicity, and vice versa.

- bool **Update**(T x) returns true if the collection contains an item equal to x, in which case that item is replaced by x; otherwise returns false without modifying the collection. If any item was updated, and the collection has set semantics or `DuplicatesByCounting` is false, then only one copy of `x` is updated; but if the collection has bag semantics and `DuplicatesByCounting` is true, then all copies of the old item are updated. If any item was updated, then events ItemsRemoved, ItemsAdded and CollectionChanged are raised. Throws ReadOnlyCollectionException if the collection is read-only.

- `bool` **`Update`**`(T x, out T xOld)` returns true if the collection contains an item equal to `x`, in which case that item is replaced by `x` and the old item is bound to `xOld`; otherwise returns false and binds the default value for T to `xOld` without modifying the collection. The collection is updated and events raised as for `Update(T)`. Throws ReadOnlyCollectionException if the collection is read-only.
- `bool` **`UpdateOrAdd`**`(T x)` returns true and updates the collection if the collection contains an item equal to `x`; otherwise returns false and adds `x` to the collection. In the first case (return value is true), if the collection has set semantics or `DuplicatesByCounting` is false, then one copy of the old item is updated; but if the collection has bag semantics and `DuplicatesByCounting` is true, then all copies of the old item are updated. Also, events ItemsRemoved, ItemsAdded and CollectionChanged are raised. In the second case (return value is false), `x` is added to the collection and events ItemsAdded and CollectionChanged are raised. Throws ReadOnlyCollectionException if the collection is read-only.
- `bool` **`UpdateOrAdd`**`(T x, out T xOld)` returns true if the collection contains an item equal to `x`, in which case that item is replaced by `x` and the old item is bound to `xOld`; otherwise returns false, adds `x` to the collection, and binds the default value for T to `xOld`. The collection is updated and events are raised as for `UpdateOrAdd(T)`. Throws ReadOnlyCollectionException if the collection is read-only.

Events

- `event CollectionChangedHandler<T>` **`CollectionChanged`**, see page 48.
  Raised by `Add`, `AddAll`, `Clear`, `FindOrAdd`, `Remove`, `RemoveAll`, `RemoveAllCopies`, `RetainAll`, `Update` and `UpdateOrAdd`.
- `event CollectionClearedHandler<T>` **`CollectionCleared`**, see page 48.
  Raised by `Clear`.
- `event ItemInsertedHandler<T>` **`ItemInserted`**, see page 48.
- `event ItemRemovedAtHandler<T>` **`ItemRemovedAt`**, see page 48.
- `event ItemsAddedHandler<T>` **`ItemsAdded`**, see page 48.
  Raised by `Add`, `AddAll`, `FindOrAdd`, `Update` and `UpdateOrAdd`.
- `event ItemsRemovedHandler<T>` **`ItemsRemoved`**, see page 48. Raised by `Remove`, `RemoveAll`, `RemoveAllCopies`, `RetainAll`, `Update` and `UpdateOrAdd`.

4.2 Interface ICollectionValue<T>

Inherits from: SCG.IEnumerable<T>, System.IFormattable, and IShowable.

Implemented by: ArrayList<T> (section 6.2), HashBag<T> (section 6.11), HashDictionary<K,V> (section 7.1), HashSet<T> (section 6.10), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5), IntervalHeap<T> (section 6.12), LinkedList<T> (section 6.3), SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), TreeDictionary<K,V> (section 7.2), TreeSet<T> (section 6.8), and WrappedArray<T> (section 6.6).

Properties

- Read-only property `EventTypeEnum` **`ActiveEvents`** is the set of events for which there are active event handlers attached to this collection. More precisely, it is the of bitwise "or" of the EventTypeEnum values for those events; see section 3.1.

- Read-only property `int` **`Count`** is the number of items in the collection value; this is the number of items that enumeration of the collection value would produce.

- Read-only property `Speed` **`CountSpeed`** is the guaranteed run-time of the `Count` property; see section 3.3.

- Read-only property `bool` **`IsEmpty`** is true if `Count` is zero, otherwise false.

- Read-only property `EventTypeEnum` **`ListenableEvents`** is the set of events for which handlers can be attached to this collection, or more precisely, the bitwise "or" of the EventTypeEnum values; see section 3.1. For instance, to test whether an ItemInsertedHandler can be attached, evaluate `(ListenableEvents & EventTypeEnum.Inserted) != 0`. An attempt to attach an event handler on an event that is not listenable will throw UnlistenableEventException.

Methods

- `bool` **`All`**`(Fun<T,bool> p)` applies delegate `p` to each item `x` of the collection in enumeration order until `p(x)` evaluates to false or until there are no more items. Returns false if `p(x)` returned false for some item; otherwise returns true. If the delegate `p` modifies the given collection, then a CollectionModifiedException may be thrown.

- `void` **`Apply`**`(Act<T> act)` applies delegate `act` to each item `x` of the collection in enumeration order. If the delegate `act` modifies the given collection, then a CollectionModifiedException may be thrown.

- `T` **`Choose`**`()` returns an arbitrary item from the collection, or throws NoSuchItemException if the collection is empty. Multiple calls to `Choose()` may return the same item or distinct items, at the collection's whim. For collections that also implement ICollection<T> it is guaranteed that if `Choose()` returns item `x`, then `Remove(x)` on the same unmodified collection will be efficient.

- `void` **`CopyTo`**`(T[] arr, int i)` copies the collection's items to array `arr` in enumeration order, starting at position `i` in `arr`. Throws ArgumentOutOfRangeException if `i < 0` or `Count+i > arr.Length`, and if so does not modify `arr`. Throws NullReferenceException if `arr` is `null`. Throws ArrayTypeMismatchException if some collection item is not assignable to the array's element type, after copying all items until, but not including, the offending one.

- `bool` **`Exists`**`(Fun<T,bool> p)` applies delegate `p` to each item `x` of the collection in enumeration order until `p(x)` evaluates to true or until there are no more items. Returns true if `p(x)` returned true for some item; otherwise returns false. If the delegate `p` modifies the given collection, then a CollectionModifiedException may be thrown.

- `SCG.IEnumerable<T>` **`Filter`**`(Fun<T,bool> p)` creates an enumerable whose enumerators apply delegate `p` to each item `x` of the collection in enumeration order, yielding those items for which `p(x)` evaluates to true. Applies `p` to the collection's items only to the extent that items are requested from the enumerator. In particular, predicate `p` will not be called until the first call of the `MoveNext()` method in an enumerator created from the enumerable. Applies `p` over again from the beginning of the collection for each enumerator created from the enumerable. If the delegate `p` modifies the given collection, then a CollectionModifiedException may be thrown.

- `bool` **`Find`**`(Fun<T,bool> p, out T res)` applies predicate `p` to each item `x` of the collection in enumeration order until `p(x)` evaluates to true or until there are no more items. Returns true if `p(x)` returned true for some item `x` and in that case binds `res` to that item; otherwise returns false and binds the default value for T to `res`. In case of success, `res` is the first item `x` in the collection for which `p(x)` is true. If the delegate `p` modifies the given collection, then a CollectionModifiedException may be thrown.

- `T[]` **`ToArray`**`()` creates a new array that contains the collection's items in enumeration order.

Events

Section 8.8.5 describes the event handler types and section 8.8.6 describes the event argument types. One cannot add event listeners to a list view (section 8.1), only to the underlying list.

- `event CollectionChangedHandler<T>` **`CollectionChanged`** is raised to signal the end of a modification to the collection. The event argument is the collection that was modified.

- `event CollectionClearedHandler<T>` **`CollectionCleared`** is raised after the collection or part of it was cleared by `Clear` or `RemoveInterval`. The event arguments are the collection itself and a description of what part of the collection was cleared.

- `event ItemInsertedHandler<T>` **`ItemInserted`** is raised after an item was added to the indexed collection by `Enqueue`, `Insert`, `InsertAll`, `InsertFirst`, `InsertLast`, `Push` or the set accessor of an indexer `this[i]=e`. The event arguments are the collection, the item that was added, and the position at which it was added.

- `event ItemRemovedAtHandler<T>` **`ItemRemovedAt`** is raised after an item was removed from the collection by `Dequeue`, `Pop`, `RemoveAt`, `RemoveFirst`, `RemoveLast`, or the set accessor of an indexer `this[i]=e`. The event arguments are the collection and the item that was removed.

- `event ItemsAddedHandler<T>` **`ItemsAdded`** is raised after an item was added to the collection by `Add`, `Insert`, `Update` or similar, or by the set accessor of an indexer `this[i]=e`. The event arguments are the collection itself, the item `x` that was added, and the number of copies of x that were added (always 1 when the collection has set semantics).

- `event ItemsRemovedHandler<T>` **`ItemsRemoved`** is raised after an item was removed from the collection by `Remove`, `RemoveAt`, `RemoveRangeFromTo`, `Replace`, `RetainAll`, `Update` or similar, or by the set accessor of an indexer `this[i]=e`. The event arguments are the collection itself, the item `x` that was removed, and the number of copies of x that were removed (always 1 when the collection has set semantics).

4.3 Interface IDirectedCollectionValue<T>

Inherits from: ICollectionValue<T> and IDirectedEnumerable<T>.

Implemented by: ArrayList<T> (section 6.2), CircularQueue<T> (section 6.1), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5), LinkedList<T> (section 6.3), SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), TreeSet<T> (section 6.8), and WrappedArray<T> (section 6.6).

Properties

- Read-only property EventTypeEnum **ActiveEvents**, see page 46.
- Read-only property int **Count**, see page 46.
- Read-only property Speed **CountSpeed**, see page 46.
- Read-only property EnumerationDirection **Direction**, see page 51.
- Read-only property bool **IsEmpty**, see page 46.
- Read-only property EventTypeEnum **ListenableEvents**, see page 46.

Methods

- bool **All**(Fun<T,bool> p), see page 46.
- void **Apply**(Act<T> act), see page 46.
- IDirectedEnumerable<T> **IDirectedEnumerable<T>.Backwards**(), see page 51.
- IDirectedCollectionValue<T> **Backwards**() returns a new directed collection value that has the opposite enumeration order of the given one.
- T **Choose**(), see page 47.
- void **CopyTo**(T[] arr, int i), see page 47.
- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.
- bool **FindLast**(Fun<T,bool> p, out T res) applies predicate p to each item x of the collection in reverse enumeration order until p(x) evaluates to true or until there are no more items. Returns true if p(x) returned true for some item x and in that case binds res to that item; otherwise returns false and binds the default value for T to res. In case of success, res is the last item

x in the collection for which p(x) is true. Equivalent to, but potentially more efficient than, Backwards().Find(p, out res). If the delegate p modifies the given collection, then a CollectionModifiedException may be thrown.

- T[] **ToArray**(), see page 47.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48.
- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.
- event ItemInsertedHandler<T> **ItemInserted**, see page 48.
- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.
- event ItemsAddedHandler<T> **ItemsAdded**, see page 48.
- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48.

4.4 Interface IDirectedEnumerable<T>

Inherits from: SCG.IEnumerable<T> and System.IFormattable.

Implemented by: ArrayList<T> (section 6.2), CircularQueue<T> (section 6.1), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5), LinkedList<T> (section 6.3), SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), TreeSet<T> (section 6.8), and WrappedArray<T> (section 6.6).

Properties

- Read-only property `EnumerationDirection` **`Direction`** returns `Forwards` if the enumeration direction is unchanged from the original, otherwise `Backwards`.

Methods

- `IDirectedEnumerable<T>` **`Backwards`**`()` returns a new directed enumerable that has the opposite enumeration order of the given one.

4.5 Interface IExtensible<T>

Inherits from: ICollectionValue<T>, System.ICloneable<T>.

Implemented by: ArrayList<T> (section 6.2), HashBag<T> (section 6.11), HashSet<T> (section 6.10), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5), IntervalHeap<T> (section 6.12), LinkedList<T> (section 6.3), SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), TreeSet<T> (section 6.8), and WrappedArray<T> (section 6.6).

Properties

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`** is true if the collection has *bag semantics*: if it may contain two items that are equal by the collection's comparer or equality comparer. Otherwise false, in which case the collection has *set semantics*.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `bool` **`DuplicatesByCounting`** is true if only the number of duplicate items is stored, not the individual duplicate values themselves; otherwise false. Namely, values may be equal by the collection's comparer or equality comparer, yet distinct objects. Relevant only for collections with bag semantics; true by convention for collections with set semantics.
- Read-only property `SCG.IEqualityComparer<T>` **`EqualityComparer`** is the item equality comparer used by this collection.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `bool` **`IsReadOnly`** is true if the collection is *read-only*, that is, if all attempts at structural modification (`Add`, `Clear`, `Insert`, `Remove`, `Update` and so on) will throw an exception; false if the collection admits such modifications. In particular, it is true for all guarded collections; see section 8.2.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.

Methods

- `bool` **`Add`**`(T x)` attempts to add item `x` to the collection. Returns true if the item was added; returns false if it was not, for instance because the collection has set semantics (`AllowsDuplicates` is false) and already contains an item equal to `x`. If the item was added, it raises events ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only.

- void **AddAll**<U>(SCG.IEnumerable<U> xs) where U:T attempts to add the items xs to the collection, in enumeration order. If AllowsDuplicates is false, then items in xs that are already in the collection, and duplicate items in xs, are ignored. If any items were added, it raises event ItemsAdded for each item added and then raises CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only. The method is generic so that it can be applied to enumerables with any item type U that is a subtype of T; see section 8.4.

- bool **All**(Fun<T,bool> p), see page 46.

- void **Apply**(Act<T> act), see page 46.

- bool **Check**() performs a comprehensive integrity check of the collection's internal representation. Relevant only for library developers.

- T **Choose**(), see page 47.

- Object **Clone**() creates a new collection as a shallow copy of the given one, as if by creating an empty collection newcoll and then doing newcoll.AddAll(this). See section 8.9.

- void **CopyTo**(T[] arr, int i), see page 47.

- bool **Exists**(Fun<T,bool> p), see page 47.

- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.

- bool **Find**(Fun<T,bool> p, out T res), see page 47.

- T[] **ToArray**(), see page 47.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48. Raised by Add and AddAll.

- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.

- event ItemInsertedHandler<T> **ItemInserted**, see page 48.

- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.

- event ItemsAddedHandler<T> **ItemsAdded**, see page 48. Raised by Add and AddAll.

- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48.

4.6 Interface IIndexed<T>

Inherits from: ISequenced<T>.

Implemented by: ArrayList<T> (section 6.2), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5), LinkedList<T> (section 6.3), SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), TreeSet<T> (section 6.8), and Wrapped-Array<T> (section 6.6).

Properties and indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`**, see page 52.
- Read-only property `Speed` **`ContainsSpeed`**, see page 42.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `EnumerationDirection` **`Direction`**, see page 51.
- Read-only property `bool` **`DuplicatesByCounting`**, see page 52.
- Read-only property `SCG.IEqualityComparer<T>` **`EqualityComparer`**, see page 52.
- Read-only property `Speed` **`IndexingSpeed`** is the guaranteed run-time of the collection's indexer `this[int]`; see section 3.3.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `bool` **`IsReadOnly`**, see page 52.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.
- Read-only indexer `T` **`this`**`[int i]` returns the `i`'th item of this indexed collection. Throws IndexOutOfRangeException if `i < 0` or `i >= Count`.
- Read-only indexer `IDirectedCollectionValue<T>` **`this`**`[int i, int n]` returns a new directed collection value containing those items of the given collection that have indexes `i`, `i+1`, ..., `i+n-1`, in that order. Does not create a new collection, but provides read-only access to the collection subsequence. Throws ArgumentOutOfRangeException if `i < 0` or `n < 0` or `i+n > Count`.

Methods

- bool **Add**(T x), see page 52.
- void **AddAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 53.
- bool **All**(Fun<T,bool> p), see page 46.
- void **Apply**(Act<T> act), see page 46.
- IDirectedEnumerable<T> **IDirectedEnumerable<T>.Backwards**(), see page 51.
- IDirectedCollectionValue<T> **Backwards**(), see page 49.
- bool **Check**(), see page 53.
- T **Choose**(), see page 47.
- void **Clear**(), see page 42.
- Object **Clone**(), see page 53.
- bool **Contains**(T x), see page 43.
- bool **ContainsAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 43.
- int **ContainsCount**(T x), see page 43.
- void **CopyTo**(T[] arr, int i), see page 47.
- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.
- bool **Find**(ref T x), see page 43.
- int **FindIndex**(Fun<T,bool> p) finds the position of the first item x that satisfies predicate p, if any. More precisely, applies predicate p to each item x of the collection in enumeration order until p(x) evaluates to true or until there are no more items. Returns the index of the x for which p(x) returned true, if any; otherwise returns −1. If the delegate p modifies the given collection, then a CollectionModifiedException may be thrown.
- bool **FindLast**(Fun<T,bool> p, out T res), see page 49.
- int **FindLastIndex**(Fun<T,bool> p) finds the position of the last item x that satisfies predicate p, if any. More precisely, applies predicate p to each item x of the collection in reverse enumeration order until p(x) evaluates to true or until there are no more items. Returns the index of the x for which p(x) returned true, if any; otherwise returns −1. If the delegate p modifies the given collection, then a CollectionModifiedException may be thrown.

- bool **FindOrAdd**(ref T x), see page 43.
- int **GetSequencedHashCode**(), see page 82.
- int **GetUnsequencedHashCode**(), see page 43.
- int **IndexOf**(T x) returns the least index i >= 0 such that this[i] equals x, if any. Otherwise returns i < 0 such that the collection's Add operation would put x at position ~i, the one's complement of i.
- ICollectionValue<KeyValuePair<T,int>> **ItemMultiplicities**(), see page 43.
- int **LastIndexOf**(T x) returns the greatest index i >= 0 such that this[i] equals x, if any. Otherwise returns i < 0 such that the collection's Add operation would put x at position ~i, the one's complement of i.
- bool **Remove**(T x), see page 43.
- bool **Remove**(T x, out T xRemoved), see page 44.
- void **RemoveAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 44.
- void **RemoveAllCopies**(T x), see page 44.
- T **RemoveAt**(int i) removes and returns the item at position i in the collection. Raises events ItemRemovedAt, ItemsRemoved and CollectionChanged. Throws IndexOutOfRangeException if i < 0 or i >= Count. Throws ReadOnlyCollectionException if the collection is read-only.
- void **RemoveInterval**(int i, int n) removes those items from the collection that have positions i..(i+n-1). Raises events CollectionCleared and CollectionChanged. Throws ArgumentOutOfRangeException if i < 0 or n < 0 or i+n > Count. Throws ReadOnlyCollectionException if the collection is read-only.
- void **RetainAll**<U>(SCG.IEnumerable<T> xs) where U:T, see page 44.
- bool **SequencedEquals**(ISequenced<T> coll), see page 82.
- T[] **ToArray**(), see page 47.
- ICollectionValue<T> **UniqueItems**(), see page 44.
- bool **UnsequencedEquals**(ICollection<T> coll), see page 44.
- bool **Update**(T x), see page 44.
- bool **Update**(T x, out T xOld), see page 45.
- bool **UpdateOrAdd**(T x), see page 45.
- bool **UpdateOrAdd**(T x, out T xOld), see page 45.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48.
  Raised by Add, AddAll, Clear, FindOrAdd, Remove, RemoveAll, RemoveAllCopies, RemoveAt, RemoveInterval, RetainAll, Update and UpdateOrAdd.
- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.
  Raised by Clear and RemoveInterval.
- event ItemInsertedHandler<T> **ItemInserted**, see page 48.
- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.
  Raised by RemoveAt.
- event ItemsAddedHandler<T> **ItemsAdded**, see page 48.
  Raised by Add, AddAll, FindOrAdd, Update and UpdateOrAdd.
- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48. Raised by Remove, RemoveAll, RemoveAllCopies, RemoveAt, RetainAll, Update and UpdateOrAdd.

4.7 Interface IIndexedSorted<T>

Inherits from: IIndexed<T> and ISorted<T>.

Implemented by: SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), and TreeSet<T> (section 6.8).

Properties and indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`**, see page 52.
- Read-only property `SCG.IComparer<T>` **`Comparer`**, see page 84.
- Read-only property `Speed` **`ContainsSpeed`**, see page 42.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `EnumerationDirection` **`Direction`**, see page 51.
- Read-only property `bool` **`DuplicatesByCounting`**, see page 52.
- Read-only property `SCG.IEqualityComparer<T>` **`EqualityComparer`**, see page 52.
- Read-only property `Speed` **`IndexingSpeed`**, see page 54.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `bool` **`IsReadOnly`**, see page 52.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.
- Read-only indexer `T` **`this`**`[int i]`, see page 54.
- Read-only indexer `IDirectedCollectionValue<T>` **`this`**`[int i, int n]`, see page 54.

Methods

- `bool` **`Add`**`(T x)`, see page 52.
- `void` **`AddAll`**`<U>(SCG.IEnumerable<U> xs) where U:T`, see page 53.
- `void` **`AddSorted`**`<U>(SCG.IEnumerable<U> xs) where U:T`, see page 84.
- `bool` **`All`**`(Fun<T,bool> p)`, see page 46.
- `void` **`Apply`**`(Act<T> act)`, see page 46.

- IDirectedEnumerable<T> **IDirectedEnumerable<T>.Backwards**(), see page 51.
- IDirectedCollectionValue<T> **Backwards**(), see page 49.
- bool **Check**(), see page 53.
- T **Choose**(), see page 47.
- void **Clear**(), see page 42.
- Object **Clone**(), see page 53.
- bool **Contains**(T x), see page 43.
- bool **ContainsAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 43.
- int **ContainsCount**(T x), see page 43.
- void **CopyTo**(T[] arr, int i), see page 47.
- int **CountFrom**(T x) returns the number of items in the sorted collection that are greater than or equal to x. Equivalent to RangeFrom(x).Count but potentially much faster.
- int **CountFromTo**(T x, T y) returns the number of items in the sorted collection that are greater than or equal to x and strictly less that y. Equivalent to RangeFromTo(x, y).Count but potentially much faster.
- int **CountTo**(T y) returns the number of items in the sorted collection that are strictly less that y. Equivalent to RangeTo(y).Count but potentially much faster.
- bool **Cut**(System.IComparable<T> c, out T cP, out bool cPValid, out T cS, out bool cSValid), see page 85.
- T **DeleteMax**(), see page 85.
- T **DeleteMin**(), see page 85.
- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.
- bool **Find**(ref T x), see page 43.
- IIndexedSorted<T> **FindAll**(Fun<T,bool> p) applies delegate p to the items of the sorted collection in increasing item order and returns a new indexed sorted collection containing those items x for which p(x) is true. It holds that FindAll(p).Count <= Count. If the delegate p modifies the given collection, then a CollectionModifiedException may be thrown.

- int **FindIndex**(Fun<T,bool> p), see page 55.
- bool **FindLast**(Fun<T,bool> p, out T res), see page 49.
- int **FindLastIndex**(Fun<T,bool> p), see page 55.
- T **FindMax**(), see page 87.
- T **FindMin**(), see page 87.
- bool **FindOrAdd**(ref T x), see page 43.
- int **GetSequencedHashCode**(), see page 82.
- int **GetUnsequencedHashCode**(), see page 43.
- int **IndexOf**(T x), see page 56.
- ICollectionValue<KeyValuePair<T,int>> **ItemMultiplicities**(), see page 43.
- int **LastIndexOf**(T x), see page 56.
- IIndexedSorted<V> **Map**<V>(Fun<T,V> f, SCG.IComparer<V> cmp) applies delegate f to the items x of the sorted collection in increasing order and returns a new indexed sorted collection whose items are the results returned by f(x) and whose item comparer is cmp. Throws ArgumentException if f is not strictly increasing; that is, if cmp.Compare(f(x1),f(x2)) >= 0 for any two consecutive items x1 and x2 from the given sorted collection. If the delegate f modifies the given collection, then a CollectionModifiedException may be thrown.
- T **Predecessor**(T x), see page 87.
- IDirectedCollectionValue<T> **RangeAll**(), see page 87.
- IDirectedEnumerable<T> **ISorted<T>.RangeFrom**(T x), see page 87.
- IDirectedCollectionValue<T> **RangeFrom**(T x) returns a directed collection value that is a read-only view, in enumeration order, of all those items in the sorted collection that are greater than or equal to x.
- IDirectedEnumerable<T> **ISorted<T>.RangeFromTo**(T x, T y), see page 87.
- IDirectedCollectionValue<T> **RangeFromTo**(T x, T y) returns a directed collection value that is a read-only view, in enumeration order, of all those items in the sorted collection that are greater than or equal to x and strictly less than y.
- IDirectedEnumerable<T> **ISorted<T>.RangeTo**(T y), see page 87.
- IDirectedCollectionValue<T> **RangeTo**(T y) returns a directed collection value that is a read-only view, in enumeration order, of all those items in the sorted collection that are strictly less than y.

- bool **Remove**(T x), see page 43.
- bool **Remove**(T x, out T xRemoved), see page 44.
- void **RemoveAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 44.
- void **RemoveAllCopies**(T x), see page 44.
- T **RemoveAt**(int i), see page 56.
- void **RemoveInterval**(int i, int n), see page 56.
- void **RemoveRangeFrom**(T x), see page 88.
- void **RemoveRangeFromTo**(T x, T y), see page 88.
- void **RemoveRangeTo**(T y), see page 88.
- void **RetainAll**<U>(SCG.IEnumerable<T> xs) where U:T, see page 44.
- bool **SequencedEquals**(ISequenced<T> coll), see page 82.
- T **Successor**(T x), see page 88.
- T[] **ToArray**(), see page 47.
- ICollectionValue<T> **UniqueItems**(), see page 44.
- bool **UnsequencedEquals**(ICollection<T> coll), see page 44.
- bool **Update**(T x), see page 44.
- bool **Update**(T x, out T xOld), see page 45.
- bool **UpdateOrAdd**(T x), see page 45.
- bool **UpdateOrAdd**(T x, out T xOld), see page 45.
- T **WeakPredecessor**(T x), see page 88.
- T **WeakSuccessor**(T x), see page 88.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48.
  Raised by Add, AddAll, AddSorted, Clear, DeleteMax, DeleteMin, FindOrAdd, Remove, RemoveAll, RemoveAllCopies, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, RetainAll, Update and UpdateOrAdd.
- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.
  Raised by Clear.
- event ItemInsertedHandler<T> **ItemInserted**, see page 48.

- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.
- event ItemsAddedHandler<T> **ItemsAdded**, see page 48.
  Raised by Add, AddAll, AddSorted, FindOrAdd, Update and UpdateOrAdd.
- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48.
  Raised by DeleteMax, DeleteMin, Remove, RemoveAll, RemoveAllCopies, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, RetainAll, Update and UpdateOrAdd.

4.8 Interface IList<T>

Inherits from: IIndexed<T>, SCG.IList<T> and System.IDisposable.

Implemented by: ArrayList<T> (section 6.2), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5) and LinkedList<T> (section 6.3), and WrappedArray<T> (section 6.6).

Properties and indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`**, see page 52.
- Read-only property `Speed` **`ContainsSpeed`**, see page 42.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `EnumerationDirection` **`Direction`**, see page 51.
- Read-only property `bool` **`DuplicatesByCounting`**, see page 52.
- Read-only property `SCG.IEqualityComparer<T>` **`EqualityComparer`**, see page 52.
- Read-write property `bool` **`FIFO`** is true if the methods `Add` and `Remove` behave like a first-in-first-out queue, that is, if method `Remove` removes and returns the first item in the list; false if they behave like a last-in-first-out stack. By default false for ArrayList<T> and true for LinkedList<T>; on all lists, items are added at the end of the list.
- Read-only property `T` **`First`** is the first item in the list or view, if any. Throws NoSuchItemException if the list is empty. Otherwise equivalent to `this[0]`.
- Read-only property `Speed` **`IndexingSpeed`**, see page 54.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `bool` **`IsFixedSize`** is true if the size of the collection cannot be changed. Any read-only list has fixed size, but a fixed-size list such as WrappedArray<T> need not be read-only: the operations `Reverse`, `Shuffle` and `Sort` are still applicable. Operations that attempt to change the size of a fixed-size list throw FixedSizeCollectionException.
- Read-only property `bool` **`IsReadOnly`**, see page 52.
- Read-only property `bool` **`IsValid`** is true if the list is a proper list or a valid view; false if the list is an invalidated view. A newly created view is valid, but may be invalidated by multi-item operations such as `Clear`, `Reverse`, `Shuffle` and `Sort` on the underlying list (see section 8.1.6) and by the `Dispose()` method.

- Read-only property T **Last** is the last item, if any, in the list or view. Throws NoSuchItemException if the list is empty. Otherwise equivalent to this[Count-1].

- Read-only property EventTypeEnum **ListenableEvents**, see page 46.

- Read-only property int **Offset** is the offset relative to the underlying list if this list is a view (see section 8.1); or zero if this list is a proper list (not a view).

- Read-write indexer T **this**[int i] is the i'th item in the list or view, where the first item is this[0] and the last item is this[Count-1]. The set accessor of the indexer raises events ItemRemovedAt, ItemsRemoved, ItemInserted, ItemsAdded and CollectionChanged. Throws IndexOutOfRangeException if i < 0 or i >= Count. Throws ReadOnlyCollectionException if the set accessor is used and the list is read-only.

- Read-only indexer IDirectedCollectionValue<T> **this**[int i, int n], see page 54.

- Read-only property IList<T> **Underlying** is the underlying list if this list is a view (see section 8.1); or null if this is a proper list (not a view). The expression xs.Underlying == null can be used to test whether xs is a proper list. The expression xs.Underlying ?? xs always returns a proper list: either xs or its underlying list.

Methods

- bool **Add**(T x), see page 52. Additionally throws FixedSizeCollectionException if the list has fixed size.

- void **SCG.ICollection<T>.Add**(T x) attempts to add item x to the list by calling Add(x). Provided to implement SCG.IList<T> and SCG.ICollection<T>.

- void **AddAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 53. Additionally throws FixedSizeCollectionException if the list has fixed size.

- bool **All**(Fun<T,bool> p), see page 46.

- void **Apply**(Act<T> act), see page 46.

- IDirectedEnumerable<T> **IDirectedEnumerable<T>.Backwards**(), see page 51.

- IDirectedCollectionValue<T> **Backwards**(), see page 49.

- bool **Check**(), see page 53.

- T **Choose**(), see page 47.

- `void` **`Clear`**`()`, see page 42. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `Object` **`Clone`**`()` creates a new list as a shallow copy of the given list or view; see page 53. Cloning of a view does not produce a new view, but a list containing the same items as the view, where the new list is of the same kind as that underlying the view.

- `bool` **`Contains`**`(T x)`, see page 43.

- `bool` **`ContainsAll`**`<U>(SCG.IEnumerable<U> xs) where U:T`, see page 43.

- `int` **`ContainsCount`**`(T x)`, see page 43.

- `void` **`CopyTo`**`(T[] arr, int i)`, see page 47.

- `void` **`Dispose`**`()` invalidates the given view; or invalidates all views of the given proper list and then clears it. More precisely, if the given list is a view, this operation frees all auxiliary data structures used to represent the view (except for the underlying list); this raises no events. If the given list is a proper list, then all views of that list are invalidated and the list itself is cleared but remains valid; this raises events CollectionCleared and CollectionChanged. Subsequent operations on invalidated views, except for `IsValid` and `Dispose`, will throw ViewDisposedException. This method is from interface System.IDisposable.

- `bool` **`Exists`**`(Fun<T,bool> p)`, see page 47.

- `SCG.IEnumerable<T>` **`Filter`**`(Fun<T,bool> p)`, see page 47.

- `bool` **`Find`**`(Fun<T,bool> p, out T res)`, see page 47.

- `bool` **`Find`**`(ref T x)`, see page 43.

- `IList<T>` **`FindAll`**`(Fun<T,bool> p)` applies delegate `p` to the items of the list or view in index order and returns a new list containing those items `x` for which `p(x)` is true. It holds that `FindAll(p).Count <= Count`. If the delegate `p` modifies the given collection, then a CollectionModifiedException may be thrown.

- `int` **`FindIndex`**`(Fun<T,bool> p)`, see page 55.

- `bool` **`FindLast`**`(Fun<T,bool> p, out T res)`, see page 49.

- `int` **`FindLastIndex`**`(Fun<T,bool> p)`, see page 55.

- `bool` **`FindOrAdd`**`(ref T x)`, see page 43. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `int` **`GetSequencedHashCode`**`()`, see page 82.

- int **GetUnsequencedHashCode**(), see page 43.
- int **IndexOf**(T x), see page 56.
- void **Insert**(int i, T x) inserts item x at position i in the list or view. After successful insertion all items at position i and higher have had their position increased by one, and it holds that this[i] is x. Raises events ItemInserted, ItemsAdded and CollectionChanged. Throws IndexOutOfRangeException if i < 0 or i > Count. Throws DuplicateNotAllowedException if AllowsDuplicates is false and x is already in the underlying list. Throws ReadOnlyCollectionException if the list or view is read-only, and FixedSizeCollectionException if the list has fixed size.
- void **Insert**(IList<T> u, T x) inserts item x into the given list or view, at the end of list or view u. Raises events ItemInserted, ItemsAdded and CollectionChanged. Throws DuplicateNotAllowedException if AllowsDuplicates is false and x is already in the underlying list. Throws ReadOnlyCollectionException if the given list or view (not u) is read-only, and FixedSizeCollectionException if it has fixed size. The view or list u must be non-null, and the given view or list must have the same underlying list, or one or both may be that underlying list; otherwise IncompatibleViewException is thrown.
- void **InsertAll**<U>(int i, SCG.IEnumerable<U> xs) where U:T inserts items from xs at position i in enumeration order. If any items were inserted, it raises events ItemInserted and ItemsAdded for each item and then raises CollectionChanged. Throws IndexOutOfRangeException if i < 0 or i > Count. Items from xs that are already in the underlying list, and duplicate items in xs, are ignored if AllowsDuplicates is false. Throws ReadOnlyCollectionException if the list or view is read-only, and FixedSizeCollectionException if it has fixed size.
- void **InsertFirst**(T x) inserts item x as the first item in the list or view, at position zero. Raises events ItemInserted, ItemsAdded and CollectionChanged. Throws DuplicateNotAllowedException if AllowsDuplicates is false and x is already in the underlying list. Throws ReadOnlyCollectionException if the list or view is read-only, and FixedSizeCollectionException if it has fixed size. Equivalent to Insert(0, x).
- void **InsertLast**(T x) inserts item x as the last item in the list or view, at position Count. Raises events ItemInserted, ItemsAdded and CollectionChanged. Throws DuplicateNotAllowedException if AllowsDuplicates is false and x is already in the underlying list. Throws ReadOnlyCollectionException if the list or view is read-only, and FixedSizeCollectionException if it has fixed size. Equivalent to Insert(Count, x).
- bool **IsSorted**() applies the default comparer for type T to pairs of neighbor items x and y from the list or view in index order until it returns a positive

number (which indicates that x is greater than y, and so the list is not sorted), or until the end of the list or view is reached. Returns false if the comparer was positive for any neighbor items x and y; otherwise returns true.

- bool **IsSorted**(SCG.IComparer<T> cmp) applies the comparer cmp to pairs of neighbor items x and y from the list or view in index order until cmp(x, y) returns a positive number (which indicates that x is greater than y, and so the list is not sorted), or until the end of the list or view is reached. Returns false if cmp(x, y) was positive for any neighbor items x and y; otherwise returns true.
- ICollectionValue<KeyValuePair<T,int>> **ItemMultiplicities**(), see page 43.
- int **LastIndexOf**(T x), see page 56.
- IList<T> **LastViewOf**(T x) returns a new list view that points at the last occurrence of x, if any, in the given list or view. More precisely, the new list view w has length 1 and w.Offset is the largest index for which this[w.Offset] is equal to x. Note also that w[0] equals w.First equals w.Last equals x. Returns null if no item in the given list or view equals x.
- IList<V> **Map**<V>(Fun<T,V> f) applies delegate f to the items x of the list or view in index order and returns a new list whose items are the results f(x). The new list is of the same kind (for instance, array list, linked list, hash-indexed array list, or hash-indexed linked list) as the given one. The new list will use the default item equality comparer for type V; see section 2.3. Throws DuplicateNotAllowedException if the (new) list does not allow duplicates and applying f to the given list produces two new items that are equal by the new item equality comparer. It holds that Map(f).Count equals Count. If the delegate f modifies the given collection, then a CollectionModifiedException may be thrown.
- IList<V> **Map**<V>(Fun<T,V> f, SCG.IEqualityComparer<V> eqc) applies delegate f to the items x of the list or view in index order and returns a new list whose items are the results f(x). The new list must be of the same kind (for instance, array list, linked list, hash-indexed array list, or hash-indexed linked list) as the given one. The new list will use the given item equality comparer eqc for type V. Throws DuplicateNotAllowedException if the (new) list does not allow duplicates and f produces two new items that are equal by eqc. It holds that Map(f,eqc).Count equals Count. If the delegate f modifies the given collection, then a CollectionModifiedException may be thrown.
- T **Remove**() removes and returns the first item from the list or view if FIFO is true, or removes and returns the last item if FIFO if false. Raises events ItemRemovedAt, ItemsRemoved and CollectionChanged. Throws ReadOnlyCollectionException if the list or view is read-only, throws FixedSizeCollectionException if it has fixed size, and throws NoSuchItemException if the list or view is empty.

- `bool` **`Remove`**`(T x)`, see page 43. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `bool` **`Remove`**`(T x, out T xRemoved)`, see page 44. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `void` **`RemoveAll`**`<U>(SCG.IEnumerable<U> xs) where U:T`, see page 44. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `void` **`RemoveAllCopies`**`(T x)`, see page 44. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `T` **`RemoveAt`**`(int i)`, see page 56. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `void` **`SCG.IList<T>.RemoveAt`**`(int i)` removes the item at position `i` in the list by calling `RemoveAt(i)`. Provided to implement `SCG.IList<T>`.

- `T` **`RemoveFirst`**`()` removes and returns the first item from the list or view. Raises events ItemRemovedAt, ItemsRemoved and CollectionChanged. The methods `Add` and `RemoveFirst` together behave like a first-in-first-out queue (section 9.22). Throws ReadOnlyCollectionException if the list or view is read-only, throws FixedSizeCollectionException if it has fixed size, and throws NoSuchItemException if the list or view is empty.

- `void` **`RemoveInterval`**`(int i, int n)`, see page 56.

- `T` **`RemoveLast`**`()` removes and returns the last item from the list or view. Raises events ItemRemovedAt, ItemsRemoved and CollectionChanged. The methods `Add` and `RemoveFirst` together behave like a last-in-first-out stack (section 9.22). Throws ReadOnlyCollectionException if the list or view is read-only, throws FixedSizeCollectionException if it has fixed size, and throws NoSuchItemException if the list or view is empty.

- `void` **`RetainAll`**`<U>(SCG.IEnumerable<T> xs) where U:T`, see page 44. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `void` **`Reverse`**`()` reverses the items in the list or view: an item that was at position `j` before the operation is at position `Count-1-j` after the operation, for `0 <= j < Count`. For its effect on views of the list, see section 8.1.6. Raises event CollectionChanged. Throws ReadOnlyCollectionException if the list or view is read-only.

- `bool` **`SequencedEquals`**`(ISequenced<T> coll)`, see page 82.

- `void` **`Shuffle`**`()` randomly permutes the items of the list or view using the library's pseudo-random number generator; see section 3.8. Throws InvalidOperationException if the list is read-only. Equivalent to `Shuffle(new C5Random())`. For its effect on views of the list, see section 8.1.6. Raises event CollectionChanged. Throws ReadOnlyCollectionException if the list or view is read-only.

- `void` **`Shuffle`**`(System.Random rnd)` randomly permutes the items of the list using the given random number generator `rnd`. Throws InvalidOperationException if the list is read-only. For its effect on views of the list, see section 8.1.6. Raises event CollectionChanged. Throws ReadOnlyCollectionException if the list or view is read-only. Can be applied also to instances of C5Random (section 3.8) which is a subclass of System.Random.

- `IList<T>` **`Slide`**`(int i)` slides the given view by `i` items, to the left if `i < 0` and to the right if `i > 0`. Returns the given view. Throws ArgumentOutOfRangeException if the operation would bring either end of the view outside the underlying list; or more precisely, if `i+Offset < 0` or `i+Offset+Count > Underlying.Count`. Throws ReadOnlyCollectionException if the view is read-only, and throws NotAViewException if the current list is not a view.

- `IList<T>` **`Slide`**`(int i, int n)` slides the given view by `i` items, to the left if `i < 0` and to the right if `i > 0`, and sets the length of the view to `n`. Returns the given view. Throws ArgumentOutOfRangeException if the operation would bring either end of the view outside the underlying list; or more precisely, if `i+Offset < 0` or `i+Offset+n > Underlying.Count`. Throws ReadOnlyCollectionException if the view is read-only, and throws NotAViewException if the list is not a view.

- `void` **`Sort`**`()` sorts the list or view using the default comparer for the item type; see section 2.6. Throws ReadOnlyCollectionException if the list or view is read-only. For its effect on views of the list, see section 8.1.6. Raises event CollectionChanged.

- `void` **`Sort`**`(SCG.IComparer<T> cmp)` sorts the list or view using the given item comparer; see section 2.6. Throws ReadOnlyCollectionException if the list is read-only. For its effect on views of the list, see section 8.1.6. Raises event CollectionChanged.

- `IList<T>` **`Span`**`(IList<T> w)` returns a new view, if any, spanned by two existing views or lists. The call `u.Span(w)` produces a new view whose left endpoint is the left endpoint of `u` and whose right endpoint is the right endpoint of `w`. If the right endpoint of `w` is strictly to the left of the left endpoint of `u`, then `null` is returned. The views or lists `u` and `w` must have the same underlying list, or one or both may be that underlying list, and `w` must be non-`null` otherwise IncompatibleViewException is thrown.

  When `list` is the underlying list, then `list.Span(w)` is a view that spans from the beginning of the list to the right endpoint of `w`; and `u.Span(list)` is a view that spans from the left endpoint of `u` to the end of the list.

- `T[]` **`ToArray`**`()`, see page 47.

- `bool` **`TrySlide`**`(int i)` returns true if the given view can be slid by `i` items, and in that case slides it exactly as `Slide(i)`; otherwise returns false and does not modify the given view. More precisely, returns true if `i+Offset >= 0` and `i+Offset+Count <= Underlying.Count`. Throws ReadOnlyCollectionException if the view is read-only, and throws NotAViewException if the list is not a view.

- `bool` **`TrySlide`**`(int i, int n)` returns true if the given view can be slid by `i` items and have its length set to `n`, and in that case slides it exactly as `Slide(i, n)`; otherwise returns false and does not modify the given view. More precisely, returns true if `i+Offset >= 0` and `i+Offset+n <= Underlying.Count`. Throws ReadOnlyCollectionException if the view is read-only, and throws NotAViewException if the list is not a view.

- `ICollectionValue<T>` **`UniqueItems`**`()`, see page 44.

- `bool` **`UnsequencedEquals`**`(ICollection<T> coll)`, see page 44.

- `bool` **`Update`**`(T x)`, see page 44. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `bool` **`Update`**`(T x, out T xOld)`, see page 45. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `bool` **`UpdateOrAdd`**`(T x)`, see page 45. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `bool` **`UpdateOrAdd`**`(T x, out T xOld)`, see page 45. Additionally throws FixedSizeCollectionException if the list has fixed size.

- `IList<T>` **`View`**`(int i, int n)` returns a new view `w` with offset `i` relative to the given list or view, and with length `n`. More precisely, `w.Offset` equals `Offset+i` and `w.Count` equals `n`. Throws ArgumentOutOfRangeException if the view would not fit inside given list or view; that is, if `i < 0` or `n < 0` or `i+n > Count`. A view of a read-only list or view is itself read-only. Note that a view created from a view is itself just a view of the underlying list. Views are not nested inside each other; for instance, a view created from another view `w` is not affected by subsequent sliding of `w`.

- `IList<T>` **`ViewOf`**`(T x)` returns a new list view that points at the first occurrence of `x`, if any, in the list or view. More precisely, the new list view `w` has length 1 and `w.Offset` is the least index for which `this[w.Offset]` is equal to `x`. Note also that `w[0]` equals `w.First` equals `w.Last` equals `x`. Returns `null` if no item in the list equals `x`.

Events

- `event CollectionChangedHandler<T>` **`CollectionChanged`**, see page 48.
  Raised by `Add`, `AddAll`, `Clear`, `Dispose`, `FindOrAdd`, `Insert`, `InsertAll`, `InsertFirst`, `InsertLast`, `Remove`, `RemoveAll`, `RemoveAllCopies`, `RemoveAt`, `RemoveFirst`,

`RemoveInterval`, `RemoveLast`, `RetainAll`, `Reverse`, `Shuffle`, `Sort`, `Update` and `UpdateOrAdd`, and by the set accessor of the indexer `this[int]`.

- `event CollectionClearedHandler<T>` **CollectionCleared**, see page 48.
  Raised by `Clear`, `Dispose` and `RemoveInterval`.
- `event ItemInsertedHandler<T>` **ItemInserted**, see page 48.
  Raised by `Insert`, `InsertAll`, `InsertFirst`, `InsertLast` and by the set accessor of the indexer `this[int]`.
- `event ItemRemovedAtHandler<T>` **ItemRemovedAt**, see page 48.
  Raised by `RemoveAt` and by the set accessor of the indexer `this[int]`.
- `event ItemsAddedHandler<T>` **ItemsAdded**, see page 48.
  Raised by `Add`, `AddAll`, `FindOrAdd`, `Insert`, `InsertAll`, `InsertFirst`, `InsertLast`, `Update` and `UpdateOrAdd`, and by the set accessor of the indexer `this[int]`.
- `event ItemsRemovedHandler<T>` **ItemsRemoved**, see page 48.
  Raised by `Remove`, `RemoveAll`, `RemoveAllCopies`, `RemoveAt`, `RetainAll`, `Update` and `UpdateOrAdd`, and by the set accessor of the indexer `this[int]`.

4.9 Interface IPersistentSorted<T>

Inherits from: ISorted<T> and System.IDisposable.

Implemented by: TreeBag<T> (section 6.9) and TreeSet<T> (section 6.8).

Properties

- Read-only property EventTypeEnum **ActiveEvents**, see page 46.
- Read-only property bool **AllowsDuplicates**, see page 52.
- Read-only property SCG.IComparer<T> **Comparer**, see page 84.
- Read-only property Speed **ContainsSpeed**, see page 42.
- Read-only property int **Count**, see page 46.
- Read-only property Speed **CountSpeed**, see page 46.
- Read-only property EnumerationDirection **Direction**, see page 51.
- Read-only property bool **DuplicatesByCounting**, see page 52.
- Read-only property SCG.IEqualityComparer<T> **EqualityComparer**, see page 52.
- Read-only property bool **IsEmpty**, see page 46.
- Read-only property bool **IsReadOnly**, see page 52.
- Read-only property EventTypeEnum **ListenableEvents**, see page 46.

Methods

- bool **Add**(T x), see page 52.
- void **AddAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 53.
- void **AddSorted**<U>(SCG.IEnumerable<U> xs) where U:T, see page 84.
- bool **All**(Fun<T,bool> p), see page 46.
- void **Apply**(Act<T> act), see page 46.
- IDirectedEnumerable<T> **IDirectedEnumerable<T>.Backwards**(), see page 51.
- IDirectedCollectionValue<T> **Backwards**(), see page 49.
- bool **Check**(), see page 53.

- T **Choose**(), see page 47.
- void **Clear**(), see page 42.
- Object **Clone**(), see page 53.
- bool **Contains**(T x), see page 43.
- bool **ContainsAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 43.
- int **ContainsCount**(T x), see page 43.
- void **CopyTo**(T[] arr, int i), see page 47.
- bool **Cut**(System.IComparable<T> c, out T cP, out bool cPValid, out T cS, out bool cSValid), see page 85.
- T **DeleteMax**(), see page 85.
- T **DeleteMin**(), see page 85.
- void **Dispose**() disposes the persistent sorted collection if it is a snapshot, releasing any internal data and preventing it from holding on to external data. Calling Dispose on a persistent sorted collection that is not a snapshot will dispose all snapshots made from the collection and then clear the collection, raising events CollectionCleared and CollectionChanged. Subsequent operations on disposed snapshots, except for Dispose, will fail. This method is inherited from System.IDisposable.
- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.
- bool **Find**(ref T x), see page 43.
- bool **FindLast**(Fun<T,bool> p, out T res), see page 49.
- T **FindMax**(), see page 87.
- T **FindMin**(), see page 87.
- bool **FindOrAdd**(ref T x), see page 43.
- int **GetSequencedHashCode**(), see page 82.
- int **GetUnsequencedHashCode**(), see page 43.
- ICollectionValue<KeyValuePair<T,int>> **ItemMultiplicities**(), see page 43.
- T **Predecessor**(T x), see page 87.

- IDirectedCollectionValue<T> **RangeAll**(), see page 87.
- IDirectedEnumerable<T> **RangeFrom**(T x), see page 87.
- IDirectedEnumerable<T> **RangeFromTo**(T x, T y), see page 87.
- IDirectedEnumerable<T> **RangeTo**(T y), see page 87.
- bool **Remove**(T x), see page 43.
- bool **Remove**(T x, out T xRemoved), see page 44.
- void **RemoveAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 44.
- void **RemoveAllCopies**(T x), see page 44.
- void **RemoveRangeFrom**(T x), see page 88.
- void **RemoveRangeFromTo**(T x, T y), see page 88.
- void **RemoveRangeTo**(T y), see page 88.
- void **RetainAll**<U>(SCG.IEnumerable<T> xs) where U:T, see page 44.
- bool **SequencedEquals**(ISequenced<T> coll), see page 82.
- ISorted<T> **Snapshot**() returns a snapshot of the persistent sorted collection. The snapshot is read-only (so property IsReadonly is true) and is unaffected by subsequent updates to the original collection, but all such updates become slightly slower. See sections 8.5 and 12.6. Any number of snapshots can be made from a given collection. There is no point in making a snapshot from a snapshot, and an attempt to do so will throw InvalidOperationException.

  Note that a snapshot has type ISorted<T>, not IIndexedSorted<T>. This is because item access by index on a snapshot would be inefficient in many plausible implementations of snapshotting, including the one used in the C5 library; see sections 12.6 and 13.10.
- T **Successor**(T x), see page 88.
- T[] **ToArray**(), see page 47.
- ICollectionValue<T> **UniqueItems**(), see page 44.
- bool **UnsequencedEquals**(ICollection<T> coll), see page 44.
- bool **Update**(T x), see page 44.
- bool **Update**(T x, out T xOld), see page 45.
- bool **UpdateOrAdd**(T x), see page 45.
- bool **UpdateOrAdd**(T x, out T xOld), see page 45.
- T **WeakPredecessor**(T x), see page 88.
- T **WeakSuccessor**(T x), see page 88.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48.
  Raised by Add, AddAll, AddSorted, Clear, DeleteMax, DeleteMin, Dispose, FindOrAdd, Remove, RemoveAll, RemoveAllCopies, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, RetainAll, Update and UpdateOrAdd.

- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.
  Raised by Clear and Dispose.

- event ItemInsertedHandler<T> **ItemInserted**, see page 48.

- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.

- event ItemsAddedHandler<T> **ItemsAdded**, see page 48.
  Raised by Add, AddAll, AddSorted, FindOrAdd, Update and UpdateOrAdd.

- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48.
  Raised by DeleteMax, DeleteMin, Remove, RemoveAll, RemoveAllCopies, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, RetainAll, Update and UpdateOrAdd.

4.10 Interface IPriorityQueue<T>

Inherits from: IExtensible<T>.

Implemented by: IntervalHeap<T> (section 6.12).

Properties and indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`**, see page 52.
- Read-only property `SCG.IComparer<T>` **`Comparer`** returns the item comparer used by this priority queue.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `bool` **`DuplicatesByCounting`**, see page 52.
- Read-only property `SCG.IEqualityComparer<T>` **`EqualityComparer`**, see page 52.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `bool` **`IsReadOnly`**, see page 52.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.
- Read-write indexer `T` **`this`**`[IPriorityQueueHandle<T> h]` gets or sets the item with handle `h` in this priority queue. The indexer's set accessor raises events ItemsRemoved, ItemsAdded and CollectionChanged. Throws InvalidHandleException if the handle is not currently associated with this priority queue: the item with that handle may have been removed or the handle may be currently associated with a different priority queue.

Methods

- `bool` **`Add`**`(T x)`, see page 52.
- `bool` **`Add`**`(ref IPriorityQueueHandle<T> h, T x)` adds item `x` to the priority queue, passing back a handle for that item in `h`. If `h` was `null` before the call, then a new handle is created and bound to `h`; otherwise the handle given in `h` must be unused (no longer associated with a priority queue) and then is reused and associated with `x` in this queue. Returns true because addition of `x` always succeeds. Raises events ItemsAdded and CollectionChanged. Throws InvalidHandleException if `h` is non-`null` and already in use, that is, already associated with some priority queue. Throws ReadOnlyCollectionException if the collection is read-only.

- void **AddAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 53.
- bool **All**(Fun<T,bool> p), see page 46.
- void **Apply**(Act<T> act), see page 46.
- bool **Check**(), see page 53.
- T **Choose**(), see page 47.
- Object **Clone**() creates a new priority queue as a shallow copy of the given one; see page 53. No handles are associated with the new priority queue.
- void **CopyTo**(T[] arr, int i), see page 47.
- T **Delete**(IPriorityQueueHandle<T> h) removes and returns the item with the given handle h. Raises events ItemsRemoved and CollectionChanged. Throws NullReferenceException if the handle is null, and throws InvalidHandleException if it is not associated with this priority queue. Throws ReadOnlyCollectionException if the collection is read-only.
- T **DeleteMax**() removes and returns a maximal item from the priority queue. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the priority queue is empty. Throws ReadOnlyCollectionException if the collection is read-only.
- T **DeleteMax**(out IPriorityQueueHandle<T> h) removes and returns a maximal item from the priority queue, assigning its now unused handle, if any, to variable h. Assigns null to h if no handle was associated with the returned item. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the priority queue is empty. Throws ReadOnlyCollectionException if the collection is read-only.
- T **DeleteMin**() removes and returns a minimal item from the priority queue. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the priority queue is empty. Throws ReadOnlyCollectionException if the collection is read-only.
- T **DeleteMin**(out IPriorityQueueHandle<T> h) removes and returns a minimal item from the priority queue, assigning its now unused handle, if any, to variable h. Assigns null to h if no handle was associated with the returned item. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the priority queue is empty. Throws ReadOnlyCollectionException if the collection is read-only.
- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.

- bool **Find**(IPriorityQueueHandle<T> h, out T x) returns true if the handle h is associated with an item in the priority queue, and if so, assigns that item to x; otherwise returns false and assigns the default value for T to x.

- T **FindMax**() returns the item with the given handle. Throws NoSuchItemException if the priority queue is empty.

- T **FindMax**(out IPriorityQueueHandle<T> h) returns a maximal item from the priority queue, assigning its handle, if any, to variable h. Assigns null to h if no handle was associated with the returned item. Throws NoSuchItemException if the priority queue is empty.

- T **FindMin**() returns a minimal item from the priority queue.
  Throws NoSuchItemException if the priority queue is empty.

- T **FindMin**(out IPriorityQueueHandle<T> h) returns a minimal item from the priority queue, assigning its handle, if any, to variable h. Assigns null to h if no handle was associated with the returned item. Throws NoSuchItemException if the priority queue is empty.

- T **Replace**(IPriorityQueueHandle<T> h, T x) replaces the priority queue item associated with handle h with item x, and returns the item previously associated with h. Raises events ItemsRemoved, ItemsAdded and CollectionChanged. Throws InvalidHandleException if the handle is not associated with any item in the priority queue. Throws ReadOnlyCollectionException if the collection is read-only.

- T[] **ToArray**(), see page 47.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48.
- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.
- event ItemInsertedHandler<T> **ItemInserted**, see page 48.
- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.
- event ItemsAddedHandler<T> **ItemsAdded**, see page 48.
- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48.

4.11 Interface IQueue<T>

Inherits from: IDirectedCollectionValue<T>.

Implemented by: ArrayList<T> (section 6.2), CircularQueue<T> (section 6.1 and LinkedList<T> (section 6.3).

Properties and indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`** is true if the queue can contain two items that are equal by the queue's item equality comparer; false otherwise.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `EnumerationDirection` **`Direction`**, see page 51.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.
- Read-only indexer `T` **`this`**`[int i]` returns the `i`'th oldest item from the queue, where the oldest item `this[0]` is at the front of the queue, and the most recently enqueued item `this[Count-1]` is at the end of the queue. Throws IndexOutOfRangeException if `i < 0` or `i >= Count`.

Methods

- `bool` **`All`**`(Fun<T,bool> p)`, see page 46.
- `void` **`Apply`**`(Act<T> act)`, see page 46.
- `IDirectedEnumerable<T>` **`IDirectedEnumerable<T>.Backwards`**`()`, see page 51.
- `IDirectedCollectionValue<T>` **`Backwards`**`()`, see page 49.
- `T` **`Choose`**`()`, see page 47.
- `void` **`CopyTo`**`(T[] arr, int i)`, see page 47.
- `T` **`Dequeue`**`()` removes and returns the item at the front of the queue, that is, the oldest item remaining in the queue, if any. Raises events ItemRemovedAt, ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the queue is empty. Throws ReadOnlyCollectionException if the collection is read-only.

- void **Enqueue**(T x) adds item x at the end of the queue. Raises events ItemInserted, ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only.
- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.
- bool **FindLast**(Fun<T,bool> p, out T res), see page 49.
- T[] **ToArray**(), see page 47.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48.
  Raised by Dequeue and Enqueue.
- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.
- event ItemInsertedHandler<T> **ItemInserted**, see page 48.
  Raised by Enqueue.
- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.
  Raised by Dequeue.
- event ItemsAddedHandler<T> **ItemsAdded**, see page 48.
  Raised by Enqueue.
- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48.
  Raised by Dequeue.

4.12 Interface ISequenced<T>

Inherits from: ICollection<T> and IDirectedCollectionValue<T>.

Implemented by: ArrayList<T> (section 6.2), HashedArrayList<T> (section 6.4), HashedLinkedList<T> (section 6.5), LinkedList<T> (section 6.3), SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), TreeSet<T> (section 6.8), and Wrapped-Array<T> (section 6.6).

Properties

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`**, see page 52.
- Read-only property `Speed` **`ContainsSpeed`**, see page 42.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `EnumerationDirection` **`Direction`**, see page 51.
- Read-only property `bool` **`DuplicatesByCounting`**, see page 52.
- Read-only property `SCG.IEqualityComparer<T>` **`EqualityComparer`**, see page 52.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `bool` **`IsReadOnly`**, see page 52.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.

Methods

- `bool` **`Add`**`(T x)`, see page 52.
- `void` **`AddAll`**`<U>(SCG.IEnumerable<U> xs) where U:T`, see page 53.
- `bool` **`All`**`(Fun<T,bool> p)`, see page 46.
- `void` **`Apply`**`(Act<T> act)`, see page 46.
- `IDirectedEnumerable<T>` **`IDirectedEnumerable<T>.Backwards`**`()`, see page 51.
- `IDirectedCollectionValue<T>` **`Backwards`**`()`, see page 49.
- `bool` **`Check`**`()`, see page 53.
- `T` **`Choose`**`()`, see page 47.

- void **Clear**(), see page 42.
- Object **Clone**(), see page 53.
- bool **Contains**(T x), see page 43.
- bool **ContainsAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 43.
- int **ContainsCount**(T x), see page 43.
- void **CopyTo**(T[] arr, int i), see page 47.
- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.
- bool **Find**(ref T x), see page 43.
- bool **FindLast**(Fun<T,bool> p, out T res), see page 49.
- bool **FindOrAdd**(ref T x), see page 43.
- int **GetSequencedHashCode**() returns the sequenced, that is, item order sensitive, hash code of the collection: a transformation of the hash codes of its items, each computed using the collection's item equality comparer.
- int **GetUnsequencedHashCode**(), see page 43.
- ICollectionValue<KeyValuePair<T,int>> **ItemMultiplicities**(), see page 43.
- bool **Remove**(T x), see page 43.
- bool **Remove**(T x, out T xRemoved), see page 44.
- void **RemoveAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 44.
- void **RemoveAllCopies**(T x), see page 44.
- void **RetainAll**<U>(SCG.IEnumerable<T> xs) where U:T, see page 44.
- bool **SequencedEquals**(ISequenced<T> coll) returns true if this collection contains the same items as coll with same multiplicities and in the same order. More precisely, enumeration of this collection and of coll must yield equal items, place for place.
- T[] **ToArray**(), see page 47.
- ICollectionValue<T> **UniqueItems**(), see page 44.
- bool **UnsequencedEquals**(ICollection<T> coll), see page 44.

- bool **Update**(T x), see page 44.
- bool **Update**(T x, out T xOld), see page 45.
- bool **UpdateOrAdd**(T x), see page 45.
- bool **UpdateOrAdd**(T x, out T xOld), see page 45.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48. Raised by Add, AddAll, Clear, FindOrAdd, Remove, RemoveAll, RemoveAllCopies, RetainAll, Update and UpdateOrAdd.
- event CollectionClearedHandler<T> **CollectionCleared**, see page 48. Raised by Clear.
- event ItemInsertedHandler<T> **ItemInserted**, see page 48.
- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.
- event ItemsAddedHandler<T> **ItemsAdded**, see page 48. Raised by Add, AddAll, FindOrAdd, Update and UpdateOrAdd.
- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48. Raised by Remove, RemoveAll, RemoveAllCopies, RetainAll, Update and UpdateOrAdd.

4.13 Interface ISorted<T>

Inherits from: ISequenced<T>.

Implemented by: SortedArray<T> (section 6.7), TreeBag<T> (section 6.9), and TreeSet<T> (section 6.8).

Properties

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`**, see page 52.
- Read-only property `SCG.IComparer<T>` **`Comparer`** is the item comparer used by this sorted collection. Always non-`null`.
- Read-only property `Speed` **`ContainsSpeed`**, see page 42.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `EnumerationDirection` **`Direction`**, see page 51.
- Read-only property `bool` **`DuplicatesByCounting`**, see page 52.
- Read-only property `SCG.IEqualityComparer<T>` **`EqualityComparer`**, see page 52.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `bool` **`IsReadOnly`**, see page 52.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.

Methods

- `bool` **`Add`**`(T x)`, see page 52.
- `void` **`AddAll`**`<U>(SCG.IEnumerable<U> xs) where U:T`, see page 53.
- `void` **`AddSorted`**`<U>(SCG.IEnumerable<U> xs) where U:T` adds all items from `xs` to the sorted collection, ignoring items already in the collection. If any items were added, it raises event ItemsAdded for each item added, and then raises event CollectionChanged. The items from `xs` must appear in increasing order according to the sorted collection's item comparer, otherwise ArgumentException is thrown. Throws ReadOnlyCollectionException if the collection is read-only.
- `bool` **`All`**`(Fun<T,bool> p)`, see page 46.

- void **Apply**(Act<T> act), see page 46.
- IDirectedEnumerable<T> **IDirectedEnumerable<T>.Backwards**(), see page 51.
- IDirectedCollectionValue<T> **Backwards**(), see page 49.
- bool **Check**(), see page 53.
- T **Choose**(), see page 47.
- void **Clear**(), see page 42.
- Object **Clone**(), see page 53.
- bool **Contains**(T x), see page 43.
- bool **ContainsAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 43.
- int **ContainsCount**(T x), see page 43.
- void **CopyTo**(T[] arr, int i), see page 47.
- bool **Cut**(System.IComparable<T> c, out T cP, out bool cPValid, out T cS, out bool cSValid) returns true if the sorted collection contains an item x such that c.CompareTo(x) is zero, otherwise false. If the sorted collection contains an item x such that c.CompareTo(x) is positive and so c greater than x, then cP is the greatest such item and cPValid is true; otherwise cPValid is false and cP is the default value for type T. Symmetrically, if the sorted collection contains an item x such that c.CompareTo(x) is negative and so c less than x, then cS is the least such item and cSValid is true; otherwise cSValid is false and cS is the default value for type T. Never throws exceptions.

  The method int c.CompareTo(T x) need not be the item comparer for type T, but its graph must pass from positive to zero at most once and from zero to negative at most once. Then cP is the last x value before reaching zero, if any, and cS is the first x value after reaching zero, if any. See figures 4.2, 4.3 and 4.4.

  If c is of type T and the collection's item comparer is the natural comparer for T, then cP is the predecessor and cS is the successor of c in the sorted collection, if any, and cPValid and cSValid report whether these values are defined.
- T **DeleteMax**() removes and returns the maximal item from the sorted collection, if any. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the collection is empty. Throws ReadOnlyCollectionException if the collection is read-only.
- T **DeleteMin**() removes and returns the minimal item from the sorted collection, if any. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the collection is empty. Throws ReadOnlyCollectionException if the collection is read-only.

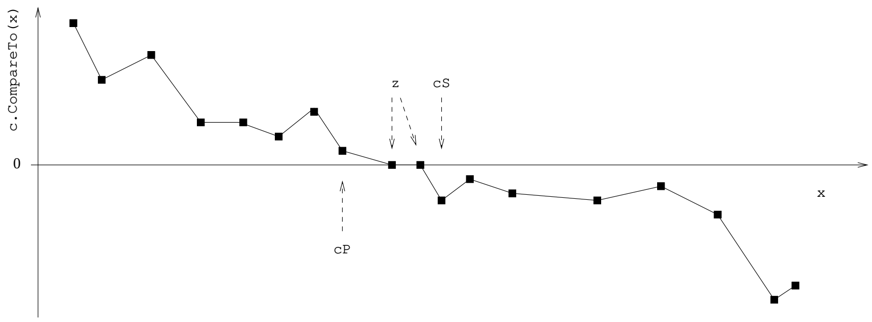


Figure 4.2: `Cut(c,...)` when `cP` and `cS` exist, and `c.CompareTo(z)=0` for some `z`.

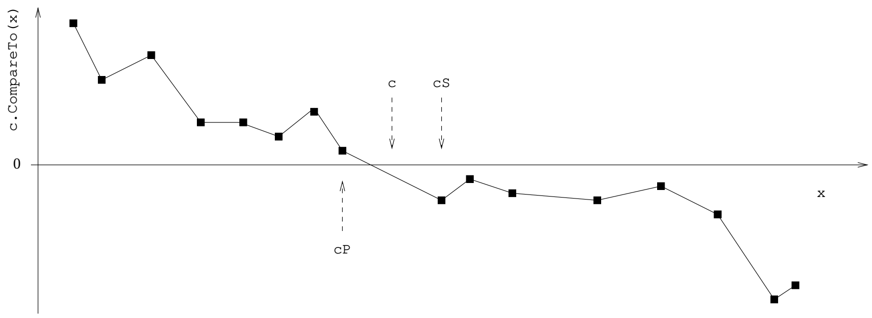


Figure 4.3: `Cut(c,...)` when `cP` and `cS` exist, but `c.CompareTo(x)=0` for no `x`.

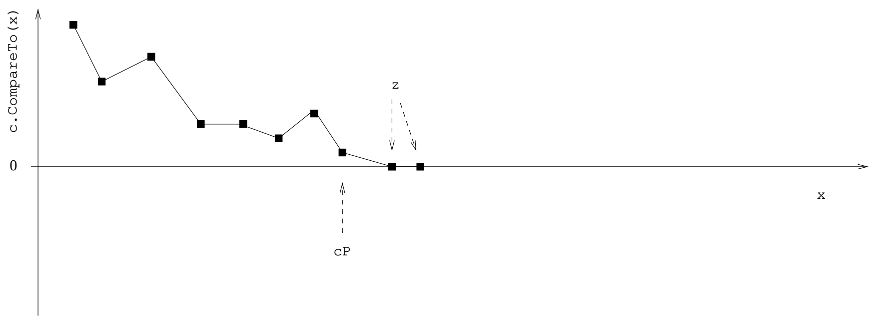


Figure 4.4: `Cut(c,...)` when `cP` but not `cS` exists, and `c.CompareTo(z)=0` for some `z`.

- bool **Exists**(Fun<T,bool> p), see page 47.
- SCG.IEnumerable<T> **Filter**(Fun<T,bool> p), see page 47.
- bool **Find**(Fun<T,bool> p, out T res), see page 47.
- bool **Find**(ref T x), see page 43.
- bool **FindLast**(Fun<T,bool> p, out T res), see page 49.
- T **FindMax**() returns the maximal item from the sorted collection, if any. Throws NoSuchItemException if the collection is empty.
- T **FindMin**() returns the minimal item from the sorted collection, if any. Throws NoSuchItemException if the collection is empty.
- bool **FindOrAdd**(ref T x), see page 43.
- int **GetSequencedHashCode**(), see page 82.
- int **GetUnsequencedHashCode**(), see page 43.
- ICollectionValue<KeyValuePair<T,int>> **ItemMultiplicities**(), see page 43.
- T **Predecessor**(T x) returns the predecessor of x, if any. The *predecessor* is the greatest item in the sorted collection that is less than x according to the item comparer. Throws NoSuchItemException if x does not have a predecessor; that is, no item in the collection is less than x.
- IDirectedCollectionValue<T> **RangeAll**() returns a directed collection value that is a read-only view, in enumeration order, of all the items in the sorted collection.
- IDirectedEnumerable<T> **RangeFrom**(T x) returns a directed enumerable whose enumerators yield, in increasing order, all those items in the sorted collection that are greater than or equal to x.
- IDirectedEnumerable<T> **RangeFromTo**(T x, T y) returns a directed enumerable whose enumerators yield, in increasing order, all those items in the sorted collection that are greater than or equal to x and strictly less than y.
- IDirectedEnumerable<T> **RangeTo**(T y) returns a directed enumerable whose enumerators yield, in enumeration order, all those items in the sorted collection that are strictly less than y.
- bool **Remove**(T x), see page 43.
- bool **Remove**(T x, out T xRemoved), see page 44.
- void **RemoveAll**<U>(SCG.IEnumerable<U> xs) where U:T, see page 44.
- void **RemoveAllCopies**(T x), see page 44.

- void **RemoveRangeFrom**(T x) deletes every item in the sorted collection that is greater than or equal to x. If any items were removed, raises ItemsRemoved for each removed item and then raises CollectionChanged once. Throws ReadOnlyCollectionException if the collection is read-only.
- void **RemoveRangeFromTo**(T x, T y) deletes every item in the sorted collection that is greater than or equal to x and strictly less than y. If any items were removed, raises ItemsRemoved for each removed item and then raises CollectionChanged once. Throws ReadOnlyCollectionException if the collection is read-only.
- void **RemoveRangeTo**(T y) deletes every item in the sorted collection that is strictly less than y. If any items were removed, raises ItemsRemoved for each removed item and then raises CollectionChanged once. Throws ReadOnlyCollectionException if the collection is read-only.
- void **RetainAll**<U>(SCG.IEnumerable<T> xs) where U:T, see page 44.
- bool **SequencedEquals**(ISequenced<T> coll), see page 82.
- T **Successor**(T x) returns the successor of x, if any. The *successor* is the least item in the sorted collection that is greater than x according to the collection's item comparer. Throws NoSuchItemException if x does not have a successor; that is, no item in the collection is greater than x.
- T[] **ToArray**(), see page 47.
- ICollectionValue<T> **UniqueItems**(), see page 44.
- bool **UnsequencedEquals**(ICollection<T> coll), see page 44.
- bool **Update**(T x), see page 44.
- bool **Update**(T x, out T xOld), see page 45.
- bool **UpdateOrAdd**(T x), see page 45.
- bool **UpdateOrAdd**(T x, out T xOld), see page 45.
- T **WeakPredecessor**(T x) returns the weak predecessor of x, if any. The *weak predecessor* is the greatest item in the sorted collection that is less than or equal to x according to the collection's item comparer. Throws NoSuchItemException if x does not have a weak predecessor; that is, no item in the collection is less than or equal to x.
- T **WeakSuccessor**(T x) returns the weak successor of x, if any. The *weak successor* is the least item in the sorted collection that is greater than or equal to x according to the item comparer. Throws NoSuchItemException if x does not have a weak successor; that is, no item in the collection is greater than or equal to x.

Events

- event CollectionChangedHandler<T> **CollectionChanged**, see page 48.
  Raised by Add, AddAll, AddSorted, Clear, DeleteMax, DeleteMin, FindOrAdd, Remove, RemoveAll, RemoveAllCopies, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, RetainAll, Update and UpdateOrAdd.

- event CollectionClearedHandler<T> **CollectionCleared**, see page 48.
  Raised by Clear.

- event ItemInsertedHandler<T> **ItemInserted**, see page 48.

- event ItemRemovedAtHandler<T> **ItemRemovedAt**, see page 48.

- event ItemsAddedHandler<T> **ItemsAdded**, see page 48.
  Raised by Add, AddAll, AddSorted, FindOrAdd, Update and UpdateOrAdd.

- event ItemsRemovedHandler<T> **ItemsRemoved**, see page 48. Raised by DeleteMax, DeleteMin, Remove, RemoveAll, RemoveAllCopies, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, RetainAll, Update and UpdateOrAdd.

4.14 Interface IStack<T>

Inherits from: IDirectedCollectionValue<T>.

Implemented by: ArrayList<T> (section 6.2), CircularQueue<T> (section 6.1) and LinkedList<T> (section 6.3).

Properties and indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `bool` **`AllowsDuplicates`** is true if the stack can contain two items that are equal by the stack's item equality comparer; false otherwise.
- Read-only property `int` **`Count`**, see page 46.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `EnumerationDirection` **`Direction`**, see page 51.
- Read-only property `bool` **`IsEmpty`**, see page 46.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.
- Read-only indexer `T` **`this`**`[int i]` returns the `i`'th item from the stack, where `this[0]` is the oldest item, at the bottom of the stack, and `this[Count-1]` is the most recently pushed item, at the top of the stack. Throws IndexOutOfRangeException if `i < 0` or `i >= Count`. To index relative to the stack top (the youngest item) in a stack `st`, use `st[st.Count-j-1]` where `j` is the desired offset relative to the stack top, `0 <= j < st.Count`.

Methods

- `bool` **`All`**`(Fun<T,bool> p)`, see page 46.
- `void` **`Apply`**`(Act<T> act)`, see page 46.
- `IDirectedEnumerable<T>` **`IDirectedEnumerable<T>.Backwards`**`()`, see page 51.
- `IDirectedCollectionValue<T>` **`Backwards`**`()`, see page 49.
- `T` **`Choose`**`()`, see page 47.
- `void` **`CopyTo`**`(T[] arr, int i)`, see page 47.
- `SCG.IEnumerable<T>` **`Filter`**`(Fun<T,bool> p)`, see page 47.
- `bool` **`Find`**`(Fun<T,bool> p, out T res)`, see page 47.

- `bool` **`FindLast`**`(Fun<T,bool> p, out T res)`, see page 49.
- `T` **`Pop`**`()` removes and returns the top item from the stack, that is, the youngest item remaining on the stack, if any. Raises events ItemRemovedAt, ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the stack is empty. Throws ReadOnlyCollectionException if the collection is read-only.
- `void` **`Push`**`(T x)` pushes item `x` onto the stack top. Raises events ItemInserted, ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the collection is read-only.
- `T[]` **`ToArray`**`()`, see page 47.

Events

- `event CollectionChangedHandler<T>` **`CollectionChanged`**, see page 48.
  Raised by `Pop` and `Push`.
- `event CollectionClearedHandler<T>` **`CollectionCleared`**, see page 48.
- `event ItemInsertedHandler<T>` **`ItemInserted`**, see page 48.
  Raised by `Push`.
- `event ItemRemovedAtHandler<T>` **`ItemRemovedAt`**, see page 48.
  Raised by `Pop`.
- `event ItemsAddedHandler<T>` **`ItemsAdded`**, see page 48.
  Raised by `Push`.
- `event ItemsRemovedHandler<T>` **`ItemsRemoved`**, see page 48.
  Raised by `Pop`.

Chapter 5

Dictionary interface details

This chapter gives a detailed description of the dictionary interfaces shown in figure 5.1. Since all important functionality of the dictionary implementation classes is described by the interfaces, this is the main documentation of the entire library (also available online as HTML help files).

The dictionary interfaces are presented in alphabetical order.

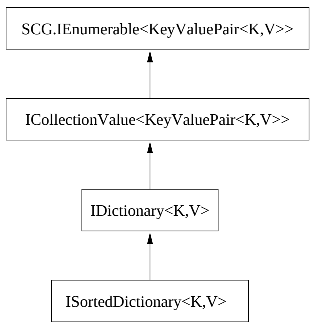


Figure 5.1: The dictionary interface hierarchy (same as figure 1.4).

5.1 Interface IDictionary<K,V>

Inherits from: ICollectionValue<KeyValuePair<K,V>>, System.ICloneable, IShowable.

Implemented by: HashDictionary<K,V> (section 7.1) and TreeDictionary<K,V> (section 7.2).

Properties and Indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `int` **`Count`**, see page 46. This is the number of (key,value) pairs in the dictionary.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `SCG.IEqualityComparer<K>` **`EqualityComparer`** returns the key equality comparer used by this dictionary, or a compatible equality comparer if the dictionary is comparer-based.
- Read-only property `Fun<K,V>` **`Fun`** returns a new delegate `f` that represents the dictionary as a finite function; more precisely, `f(k)` is computed as `this[k]`.
- Read-only property `bool` **`IsEmpty`**, see page 46. Returns true if the dictionary contains no (key,value) pairs.
- Read-only property `bool` **`IsReadOnly`** is true if the dictionary does not admit addition, deletion or update of entries.
- Read-only property `ICollectionValue<K>` **`Keys`** returns a collection value which is a view of the dictionary's keys. Its enumeration order is the same as that of the dictionary and as that of the `Values` property.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.
- Read-write indexer `V` **`this`**`[K k]` gets or sets the value associated with the given key. The indexer's set accessor `this[k] = v` replaces the value associated with `k` if there already is an entry whose key equals `k`, otherwise adds a new entry that associates `v` with `k`. It raises events ItemsRemoved, ItemsAdded and CollectionChanged in the first case; raises ItemsAdded and CollectionChanged in the second case. It throws ReadOnlyCollectionException if the collection is read-only. The indexer's get accessor throws NoSuchItemException if there is no entry whose key equals `k`.
- Read-only property `ICollectionValue<V>` **`Values`** returns a collection value which is a view of the dictionary's values. Its enumeration order is the same as that of the dictionary and as that of the `Keys` property.

Methods

- `void` **`Add`**`(K k, V v)` adds a new entry with key `k` and associated value `v` to the dictionary. Raises events ItemsAdded and CollectionChanged. Throws DuplicateNotAllowedException if the dictionary already has an entry with key equal to `k`. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `void` **`AddAll`**`<U,W>(SCG.IEnumerable<KeyValuePair<U,W>> kvs) where U:K where W:V` adds all entries from `kvs` to the dictionary. Raises events ItemsAdded and CollectionChanged. Throws DuplicateNotAllowedException if `kvs` contains duplicate keys or a key that is already in the dictionary. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `bool` **`All`**`(Fun<KeyValuePair<K,V>,bool> p)`, see page 46.

- `void` **`Apply`**`(Act<KeyValuePair<K,V>> act)`, see page 46.

- `bool` **`Check`**`()` performs a comprehensive integrity check of the dictionary's representation. Relevant only for library developers.

- `KeyValuePair<K,V>` **`Choose`**`()`, see page 47.

- `void` **`Clear`**`()` removes all entries from the dictionary. Raises events CollectionCleared and CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `Object` **`Clone`**`()` creates a new dictionary as a shallow copy of the given one, as if by creating an empty dictionary `newdict` and then doing `newdict.AddAll(this)`. See section 8.9.

- `bool` **`Contains`**`(K k)` returns true if the dictionary contains an entry whose key equals `k`; otherwise false.

- `bool` **`ContainsAll`**`<U>(SCG.IEnumerable<U> ks) where U:K` returns true if the dictionary contains keys equal to all the keys in `ks`; otherwise false.

- `void` **`CopyTo`**`(KeyValuePair<K,V>[] arr, int i)`, see page 47.

- `bool` **`Exists`**`(Fun<KeyValuePair<K,V>, bool> p)`, see page 47.

- `SCG.IEnumerable<KeyValuePair<K,V>>` **`Filter`**`(Fun<KeyValuePair<K,V>, bool> p)`, see page 47.

- `bool` **`Find`**`(Fun<KeyValuePair<K,V>,bool> p, out KeyValuePair<K,V> res)`, see page 47.

- `bool` **`Find`**`(K k, out V v)` returns true if the dictionary contains an entry whose key equals `k` and if so assigns the associated value to `v`; otherwise returns false and assigns the default value for T to `v`. This method provides an exception-free variant of `v = this[k]`.

- `bool` **`Find`**`(ref K k, out V v)` returns true if the dictionary contains an entry whose key equals `k` and if so assigns that key to `k` and assigns the associated value to `v`; otherwise returns false and assigns the default value for T to `v`. This method provides an exception-free variant of `v = this[k]`. In addition it binds the actual key to parameter `k`; the actual key may be distinct from the given `k` but is necessarily equal to it by the dictionary's key equality comparer.

- `bool` **`FindOrAdd`**`(K k, ref V v)` returns true if the dictionary contains an entry whose key equals `k` and if so assigns the associated value to `v`; otherwise returns false and adds a new entry with key `k` and associated value `v` to the dictionary. Raises event ItemsAdded and CollectionChanged in the latter case. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `bool` **`Remove`**`(K k)` returns true if the dictionary contains an entry whose key equals `k` and if so removes that entry; otherwise returns false. If an entry was removed, it raises ItemsRemoved and CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `bool` **`Remove`**`(K k, out V v)` returns true if the dictionary contains an entry whose key equals `k` and if so removes that entry and assigns the associated value to `v`; otherwise returns false and assigns the default value for T to `v`. If an entry was removed, it raises ItemsRemoved and CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `KeyValuePair<K,V>[]` **`ToArray`**`()`, see page 47.

- `bool` **`Update`**`(K k, V v)` returns true if the dictionary contains an entry whose key equals `k` and if so replaces the associated value with `v`; otherwise returns false without modifying the dictionary. In the former case it raises events ItemsRemoved, ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `bool` **`Update`**`(K k, V v, out V vOld)` returns true if the dictionary contains an entry whose key equals `k` and if so replaces the associated value with `v` and then assigns the old value to `vOld`; otherwise returns false and assigns the default value for T to `vOld` without modifying the dictionary. In the former case it raises events ItemsRemoved, ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `bool` **`UpdateOrAdd`**`(K k, V v)` returns true if the dictionary contains an entry whose key equals `k` and if so replaces the associated value with `v`; otherwise returns false and adds a new entry with key `k` and associated value `v` to the dictionary. In the first case, raises events ItemsRemoved, ItemsAdded and CollectionChanged; and in the second case raises events ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `bool` **`UpdateOrAdd`**`(K k, V v, out V vOld)` returns true if the dictionary contains an entry whose key equals `k` and if so replaces the associated value with `v` and then assigns the old value to `vOld`; otherwise returns false and adds a new entry with key `k` and associated value `v` to the dictionary. In the first case, raises events ItemsRemoved, ItemsAdded and CollectionChanged; and in the second case raises events ItemsAdded and CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

Events

- `event CollectionChangedHandler<KeyValuePair<K,V>>` **`CollectionChanged`**, see page 48. Raised by `this[k]=v`, `Add`, `AddAll`, `Clear`, `FindOrAdd`, `Remove`, `Update` and `UpdateOrAdd`.

- `event CollectionClearedHandler<KeyValuePair<K,V>>` **`CollectionCleared`**, see page 48. Raised by `Clear`.

- `event ItemInsertedHandler<KeyValuePair<K,V>>` **`ItemInserted`**, see page 48. Not raised.

- `event ItemRemovedAtHandler<KeyValuePair<K,V>>` **`ItemRemovedAt`**, see page 48. Not raised.

- `event ItemsAddedHandler<KeyValuePair<K,V>>` **`ItemsAdded`**, see page 48. Raised by `this[k]=v`, `Add`, `AddAll`, `FindOrAdd`, `Update` and `UpdateOrAdd`.

- `event ItemsRemovedHandler<KeyValuePair<K,V>>` **`ItemsRemoved`**, see page 48. Raised by `this[k]=v`, `Remove`, `Update` and `UpdateOrAdd`.

5.2 Interface ISortedDictionary<K,V>

Inherits from: IDictionary<K,V>.

Implemented by: TreeDictionary<K,V> (section 7.2).

Properties and Indexers

- Read-only property `EventTypeEnum` **`ActiveEvents`**, see page 46.
- Read-only property `SCG.IComparer<K>` **`Comparer`** returns the key comparer used by this sorted dictionary. Always non-`null`.
- Read-only property `int` **`Count`**, see page 46. This is the number of (key,value) pairs in the dictionary.
- Read-only property `Speed` **`CountSpeed`**, see page 46.
- Read-only property `SCG.IEqualityComparer<K>` **`EqualityComparer`**, see page 94.
- Read-only property `Fun<K,V>` **`Fun`**, see page 94.
- Read-only property `bool` **`IsEmpty`**, see page 46. Returns true if the dictionary contains no (key,value) pairs.
- Read-only property `bool` **`IsReadOnly`**, see page 94.
- Read-only property `ISorted<K>` **`ISortedDictionary<K,V>.Keys`** returns a read-only sorted collection which is a view of the dictionary's keys. Its enumeration order is the same as that of the dictionary and as that of the `Values` property, namely increasing key order.
- Read-only property `EventTypeEnum` **`ListenableEvents`**, see page 46.
- Read-write indexer `V` **`this`**`[K k]`, see page 94.
- Read-only property `ICollectionValue<V>` **`Values`**, see page 94.

Methods

- `void` **`Add`**`(K k, V v)`, see page 95.
- `void` **`AddAll`**`<U,W>(SCG.IEnumerable<KeyValuePair<U,W>> kvs) where U:K where W:V`, see page 95.
- `void` **`AddSorted`**`(SCG.IEnumerable<KeyValuePair<K,V>> kvs)` adds all entries from `kvs` to the sorted dictionary, ignoring any entry from `kvs` whose key is already in the dictionary. If any entries were added, it raises event Items-Added for each item added, and then raises event CollectionChanged. The

entries from kvs must appear in increasing key order according to the sorted dictionary's key comparer, otherwise ArgumentException is thrown. Throws ReadOnlyCollectionException if the dictionary is read-only.

- bool **All**(Fun<KeyValuePair<K,V>,bool> p), see page 46.
- void **Apply**(Act<KeyValuePair<K,V>> act), see page 46.
- bool **Check**(), see page 95.
- void **Clear**(), see page 95.
- KeyValuePair<K,V> **Choose**(), see page 47.
- Object **Clone**() creates a new sorted dictionary as a shallow copy of the given one, as if by creating an empty sorted dictionary newdict and then executing newdict.AddAll(this). See section 8.9.
- bool **Contains**(K k), see page 95.
- bool **ContainsAll**<U>(SCG.IEnumerable<U> ks) where U:K, see page 95.
- void **CopyTo**(KeyValuePair<K,V>[] arr, int i), see page 47.
- bool **Cut**(IComparable<K> c, out KeyValuePair<K,V> cP, out bool cPValid, out KeyValuePair<K,V> cS, out bool cSValid) returns true if the sorted dictionary contains an entry (k,v) such that c.CompareTo(k) is zero, otherwise false. If the sorted dictionary contains an entry (k,v) such that c.CompareTo(k) is positive and so c greater than k, then cP is the entry with the greatest such k and cPValid is true; otherwise cPValid is false and cP is the default value for type KeyValuePair<K,V>. Symmetrically, if the sorted dictionary contains an entry (k,v) such that c.CompareTo(k) is negative and so c less than k, then cS is the entry with the least such k and cSValid is true; otherwise cSValid is false and cS is the default value for type KeyValuePair<K,V>. Never throws exceptions.

  The method int c.CompareTo(K k) need not be the comparer for key type k, but its graph must pass from positive to zero at most once and from zero to negative at most once. See figures 4.2, 4.3 and 4.4.

  If c is of type K and the dictionary's key comparer is the natural comparer for K, then cP is the predecessor entry and cS is the successor entry of the entry with key c in the sorted dictionary, if any, and cPValid and cSValid report whether these values are defined.
- KeyValuePair<K,V> **DeleteMax**() removes and returns the entry with maximal key from the sorted dictionary, if any. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the dictionary is empty. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `KeyValuePair<K,V>` **`DeleteMin`**`()` removes and returns the entry with minimal key from the sorted dictionary, if any. Raises events ItemsRemoved and CollectionChanged. Throws NoSuchItemException if the dictionary is empty. Throws ReadOnlyCollectionException if the dictionary is read-only.
- `bool` **`Exists`**`(Fun<KeyValuePair<K,V>, bool> p)`, see page 47.
- `SCG.IEnumerable<KeyValuePair<K,V>>` **`Filter`**`(Fun<KeyValuePair<K,V>, bool> p)`, see page 47.
- `bool` **`Find`**`(Fun<KeyValuePair<K,V>,bool> p, out KeyValuePair<K,V> res)`, see page 47.
- `bool` **`Find`**`(K k, out V v)`, see page 95.
- `bool` **`Find`**`(ref K k, out V v)`, see page 96.
- `KeyValuePair<K,V>` **`FindMax`**`()` returns the entry with maximal key from the sorted dictionary, if any. Throws NoSuchItemException if the dictionary is empty.
- `KeyValuePair<K,V>` **`FindMin`**`()` returns the entry with minimal key from the sorted dictionary, if any. Throws NoSuchItemException if the dictionary is empty.
- `bool` **`FindOrAdd`**`(K k, ref V v)`, see page 96.
- `KeyValuePair<K,V>` **`Predecessor`**`(K k)` returns the entry that is the predecessor of `k`, if any. The *predecessor* of `k` is the entry in the sorted dictionary with the greatest key strictly less than `k` according to the key comparer. Throws NoSuchItemException if `k` does not have a predecessor entry; that is, no key is less than `k`.
- `IDirectedCollectionValue<KeyValuePair<K,V>>` **`RangeAll`**`()` returns a read-only directed collection value that is a view, in increasing key order, of all the entries in the sorted dictionary.
- `IDirectedEnumerable<KeyValuePair<K,V>>` **`RangeFrom`**`(K x)` returns a directed enumerable whose enumerators yield, in increasing key order, all those entries in the sorted dictionary whose keys are greater than or equal to `x`.
- `IDirectedEnumerable<KeyValuePair<K,V>>` **`RangeFromTo`**`(K x, K y)` returns a directed enumerable whose enumerators yield, in increasing key order, all those entries in the sorted collection whose keys are greater than or equal to `x` and strictly less than `y`.
- `IDirectedEnumerable<KeyValuePair<K,V>>` **`RangeTo`**`(K y)` returns a directed enumerable whose enumerators yield, in increasing key order, all those entries in the sorted collection whose keys are strictly less than `y`.

- `bool` **`Remove`**`(K k)`, see page 96.

- `bool` **`Remove`**`(K k, out V v)`, see page 96.

- `void` **`RemoveRangeFrom`**`(K x)` deletes every entry in the sorted dictionary for which the key is greater than or equal to `x`. If any entries were removed, it raises ItemsRemoved for each removed item and then raises CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `void` **`RemoveRangeFromTo`**`(K x, K y)` deletes every entry in the sorted collection for which the key is greater than or equal to `x` and strictly less than `y`. If any items were removed, raises ItemsRemoved for each removed entry and then raises CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `void` **`RemoveRangeTo`**`(K y)` deletes every entry in the sorted dictionary for which the key is strictly less than `y`. If any entries were removed, raises ItemsRemoved for each removed item and then raises CollectionChanged. Throws ReadOnlyCollectionException if the dictionary is read-only.

- `KeyValuePair<K,V>` **`Successor`**`(K k)` returns the entry that is the successor of `k`, if any. The *successor* of `k` is the entry in the sorted dictionary with the least key that is strictly greater than `k` according to the key comparer. Throws NoSuchItemException if `k` does not have a successor; that is, no entry in the dictionary has a key that is greater than `k`.

- `KeyValuePair<K,V>[]` **`ToArray`**`()`, see page 47.

- `bool` **`Update`**`(K k, V v)`, see page 96.

- `bool` **`Update`**`(K k, V v, out V vOld)`, see page 96.

- `bool` **`UpdateOrAdd`**`(K k, V v)`, see page 96.

- `bool` **`UpdateOrAdd`**`(K k, V v, out V vOld)`, see page 97.

- `KeyValuePair<K,V>` **`WeakPredecessor`**`(K k)` returns the entry that is the weak predecessor of `k`, if any. The *weak predecessor* of `k` is the entry in the sorted dictionary with the greatest key less than or equal to `k` according to the key comparer. Throws NoSuchItemException if `k` does not have a weak predecessor entry; that is, no key is less than or equal to `k`.

- `KeyValuePair<K,V>` **`WeakSuccessor`**`(K k)` returns the entry that is the weak successor of `k`, if any. The *weak successor* of `k` is the entry in the sorted collection with the least key that is greater than or equal to `k` according to the key comparer. Throws NoSuchItemException if `k` does not have a weak successor; that is, no entry in the dictionary has a key that is greater than or equal to `k`.

Events

- event CollectionChangedHandler<KeyValuePair<K,V>> **CollectionChanged**, see page 48. Raised by this[k]=v, Add, AddAll, AddSorted, Clear, DeleteMax, DeleteMin, FindOrAdd, Remove, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, Update and UpdateOrAdd.

- event CollectionClearedHandler<KeyValuePair<K,V>> **CollectionCleared**, see page 48. Raised by Clear.

- event ItemInsertedHandler<KeyValuePair<K,V>> **ItemInserted**, see page 48. Not raised.

- event ItemRemovedAtHandler<KeyValuePair<K,V>> **ItemRemovedAt**, see page 48. Not raised.

- event ItemsAddedHandler<KeyValuePair<K,V>> **ItemsAdded**, see page 48. Raised by this[k]=v, Add, AddAll, AddSorted, FindOrAdd, Update and UpdateOrAdd.

- event ItemsRemovedHandler<KeyValuePair<K,V>> **ItemsRemoved**, see page 48. Raised by this[k]=v, DeleteMax, DeleteMin, Remove, RemoveRangeFrom, RemoveRangeFromTo, RemoveRangeTo, Update and UpdateOrAdd.

Chapter 6

Collection implementations

The preceding chapter introduced the concept hierarchy, or interface hierarchy. This chapter describes their actual implementations: data structures and algorithms for collections and dictionaries. The collection classes and the interfaces they implement are shown in figure 6.2 which is an extension of figure 4.1.

A collection class is partially characterized by its properties: does it allow duplicates, are duplicates stored explicitly, how fast is membership test (`Contains`), and so on. Some of the these properties are accessible as C# properties called `AllowsDuplicates`, `DuplicatesByCounting`, `IsFixedSize`, and so on. An overview of the collection classes from this perspective is given in figure 6.1.

Collection class	`AllowsDuplicates`	`DuplicatesByCounting`	`IsFixedSize`	`FIFO` default	`ContainsSpeed`	`IndexingSpeed`
CircularQueue<T>	`true`	`false`	`false`	–	–	–
ArrayList<T>	`true`	`false`	`false`	`false`	`Linear`	`Constant`
LinkedList<T>	`true`	`false`	`false`	`true`	`Linear`	`Linear`
HashedArrayList<T>	`false`	`true`	`false`	`false`	`Constant`	`Constant`
HashedLinkedList<T>	`false`	`true`	`false`	`true`	`Constant`	`Linear`
WrappedArray<T>	`true`	`false`	`true`	(throws)	`Linear`	`Constant`
SortedArray<T>	`false`	`true`	`false`	–	`Logarithmic`	`Constant`
TreeSet<T>	`false`	`true`	`false`	–	`Logarithmic`	`Logarithmic`
TreeBag<T>	`true`	`true`	`false`	–	`Logarithmic`	`Logarithmic`
HashSet<T>	`false`	`true`	`false`	–	`Constant`	–
HashBag<T>	`true`	`true`	`false`	–	`Constant`	–
IntervalHeap<T>	`true`	`false`	`false`	–	–	–

Figure 6.1: Properties of C5 collection classes. In all cases, property `CountSpeed` has the value `Constant`.

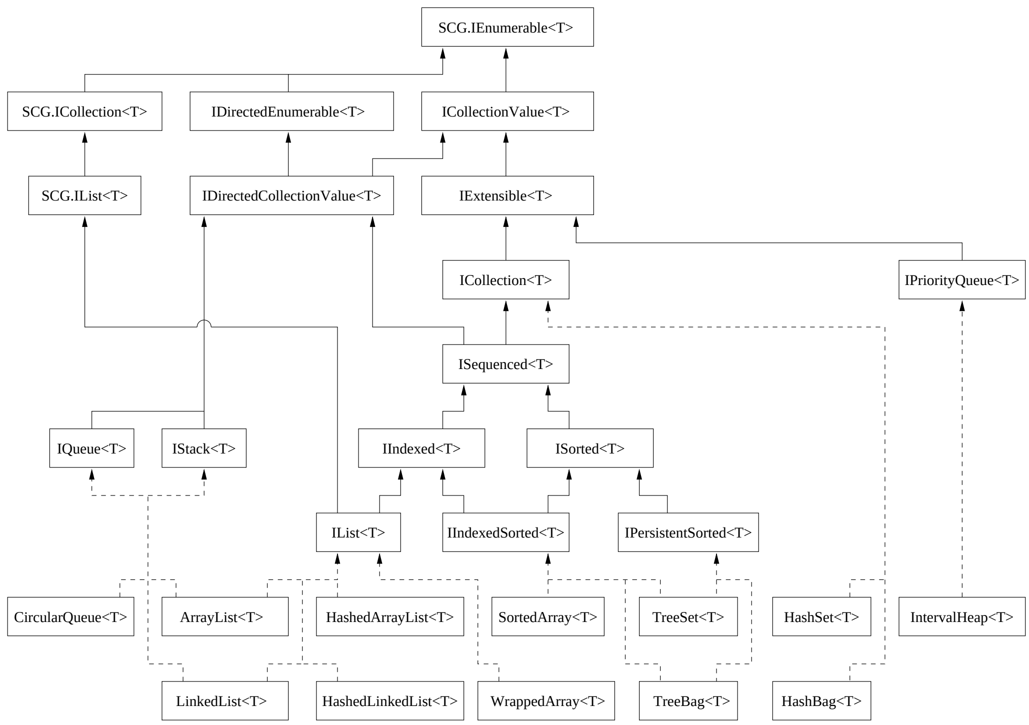


Figure 6.2: The collection classes and interfaces. Solid lines indicate a subinterface relation, and dashed lines indicate an implementation relation.

6.1 Circular queues

Class CircularQueue<T> implements the interfaces IQueue<T> (see sections 1.4.10 and 4.11) and IStack<T> (sections 1.4.9 and 4.14) and uses an underlying array to implement a first-in-first-out (FIFO) queue by methods `Enqueue` and `Dequeue`, or a last-in-first-out (LIFO) stack by methods `Push` and `Pop`. In fact, `Push` has the same effect as `Enqueue` whereas `Pop` and `Dequeue` remove items from opposite ends of the circular queue (or stack). A circular queue has efficient $O(1)$ item access by index in contrast to queues based on linked lists. For a circular queue `cq`, the item `cq[0]` is the first (oldest) item in the queue, the one that will be returned by the next call to `cq.Dequeue()`.

Class CircularQueue<T> has two constructors:

- `CircularQueue<T>()` creates a circular queue with a default initial capacity.
- `CircularQueue<T>(int capacity)` creates a circular queue with at least the given initial capacity. The circular queue will grow as needed when items are added.

6.2 Array lists

Class ArrayList<T> implements interfaces IList<T> (see sections 1.4.11 and 4.8), IStack<T> (sections 1.4.9 and 4.14) and IQueue<T> (sections 1.4.10 and 4.11) using an internal array to store the list items. Item access by index `i` takes time $O(1)$, or constant time, regardless of `i`, but insertion of one or more items at position `i` takes time $O(\texttt{Count} - \texttt{i})$, proportional to the number of items that must be moved to make room for the new item.

The default value of the `FIFO` property is false, so the `Remove()` method by default removes and returns the list's last item. An array list allows duplicates, so `AllowsDuplicates` is true, and stores duplicates explicitly, so `DuplicatesByCounting` is false. The `IsFixedSize` property is false, so insertion and deletion is supported. The `Sort` method on ArrayList<T> is introspective quicksort, a version of quicksort guaranteed to be efficient.

Class ArrayList<T> has four constructors:

- `ArrayList<T>()` creates an array list with a default initial capacity and a default item equality comparer.
- `ArrayList<T>(SCG.IEqualityComparer<T> eqc)` creates an array list with a default initial capacity and the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.
- `ArrayList<T>(int capacity)` creates an array list with at least the given initial capacity and a default item equality comparer.

- `ArrayList<T>(int capacity, SCG.IEqualityComparer<T> eqc)` creates an array list with at least the given initial capacity and with the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.

6.3 Linked lists

Class LinkedList<T> implements interfaces IList<T> (sections 1.4.11 and 4.8), IStack<T> (sections 1.4.9 and 4.14) and IQueue<T> (sections 1.4.10 and 4.11) using a doubly-linked list of internal nodes to store the list items. Insertion of an item at a given point in the list takes time $O(1)$, or constant time, but getting to that point by indexing may take linear time. Item access by index `i` takes time proportional to the distance from the nearest end of the list, more precisely $O(\min(\mathtt{i}, \mathtt{Count} - \mathtt{i}))$, and so is fast only near the ends of the list.

The default value of the `FIFO` property is true, so the `Remove()` method by default removes and returns the list's first item. A linked list allows duplicates, so `AllowsDuplicates` is true and method `Add` always returns true. Duplicates are stored explicitly, so property `DuplicatesByCounting` is false. The `IsFixedSize` property is false, so insertion and deletion is supported. The `Sort` method on LinkedList<T> is an in-place merge sort, which is stable.

Class LinkedList<T> has two constructors:

- `LinkedList<T>()` creates a linked list with a default item equality comparer.
- `LinkedList<T>(SCG.IEqualityComparer<T> eqc)` creates a linked list with the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.

6.4 Hashed array lists

Class HashedArrayList<T> implements IList<T> (section 1.4.11) and is very similar to an ordinary array list (section 6.2), but in addition maintains a hash table so that it can quickly find the position of a given item in the array list. Item lookup by item value using the `IndexOf(x)` and `ViewOf(x)` methods take time $O(1)$, or constant time, thanks to this hash table.

In contrast to an ordinary array list, a hashed array list does not allow duplicates, so property `AllowsDuplicates` is false. As for an ordinary array list, the default value of the `FIFO` property and the `IsFixedSize` property are false.

Class HashedArrayList<T> has four constructors:

- `HashedArrayList<T>()` creates an array list with a default initial capacity and a default item equality comparer.
- `HashedArrayList<T>(SCG.IEqualityComparer<T> eqc)` creates an array list with a default initial capacity and the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.

- `HashedArrayList<T>(int capacity)` creates an array list with at least the given initial capacity and a default item equality comparer.

- `HashedArrayList<T>(int capacity, SCG.IEqualityComparer<T> eqc)` creates an array list with at least the given initial capacity and the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.

6.5 Hashed linked lists

Class HashedLinkedList<T> implements IList<T> (sections 1.4.11 and 4.8) and is very similar to an ordinary linked list (section 6.3), but in addition maintains a hash table so that it can quickly find the position of a given item in the linked list. Item lookup by item value using the `ViewOf(x)` method takes time $O(1)$, or constant time, thanks to this hash table. Finding the index of a given item using the `IndexOf(x)` method takes time $O(n)$, proportional to the length of the list.

In contrast to an ordinary linked list, a hashed linked list does not allow duplicates, so property `AllowsDuplicates` is false. As for an ordinary linked list, the default value of the `FIFO` property is true and the `IsFixedSize` property is false.

Class HashedLinkedList<T> has two constructors:

- `HashedLinkedList<T>()` creates a linked list with a default item equality comparer.

- `HashedLinkedList<T>(SCG.IEqualityComparer<T> eqc)` creates a linked list with the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.

6.6 Wrapped arrays

Class WrappedArray<T> implements IList<T> (section 1.4.11 and 4.8) and provides a way to wrap an ordinary one-dimensional array of type T[] as a fixed-size array list (section 6.2). The primary use of wrapped arrays is to allow all IList<T> functionality, in particular views, search and reordering, to be used on ordinary one-dimensional arrays.

Wrapping an array as a WrappedArray<T> does not copy the array, and is an $O(1)$, or constant time, operation. Subsequent updates to the underlying array show through to the WrappedArray<T>, and vice versa. Updates to the underlying array do not raise events, whereas updates through the wrapper do. Most IList<T> operations, including the creation of views, work also for WrappedArray<T>, but not those methods that would cause the collection to grow or shrink. In particular, `Add`, `Clear`, `FIFO`, `Insert`, `Remove`, and `RemoveAt` do not work, but will throw FixedSizeCollectionException. By contrast, operations such as `Reverse`, `Shuffle` and `Sort` do work: although they change the collection, they do not change its size. All supported operations have the same running time as for ArrayList<T>.

The properties AllowsDuplicates and IsFixedSize are true.

Class WrappedArray<T> has one constructor:

- WrappedArray<T>(T[] arr) creates a wrapped array list whose underlying array is arr. It is not read-only, but it has fixed size: its Count is fixed and equals arr.Length. Throws NullReferenceException if arr is null.
- WrappedArray<T>(T[] arr, SCG.IEqualityComparer<T> eqc) creates a wrapped array list whose underlying array is arr. It is not read-only, but it has fixed size: its Count is fixed and equals arr.Length. Throws NullReferenceException if arr or eqc is null.

6.7 Sorted arrays

Class SortedArray<T> implements interface IIndexedSorted<T> (see sections 1.4.6 and 4.7). A sorted array is similar to an array list because it supports indexing, FindIndex, and so on, but different because it does not support list views, First, Last, Shuffle and so on. In contrast to an array list, a sorted array keeps its items sorted and does not allow duplicates, so property AllowsDuplicates is false.

Class SortedArray<T> has the following constructors:

- SortedArray<T>() creates a new sorted array with a default initial capacity, a default item comparer, and a default item equality comparer. The default item comparer uses the natural comparer for T; if T does not implement IComparable<T> or IComparable, then NotComparableException is thrown.
- SortedArray<T>(int capacity) creates a new sorted array with at least the given initial capacity, a default item comparer, and a default item equality comparer.
- SortedArray<T>(SCG.IComparer<T> cmp) creates a new sorted array with a default capacity, the given item comparer, and a default item equality comparer. Throws NullReferenceException if cmp is null.
- SortedArray<T>(int capacity, SCG.IComparer<T> cmp) creates a new sorted array with at least the given capacity, the given item comparer, and a default item equality comparer. Throws NullReferenceException if cmp is null.
- SortedArray<T>(int capacity, SCG.IComparer<T> cmp, SCG.IEqualityComparer<T> eqc) creates a new sorted array with at least the given capacity, the given item comparer, and the given item equality comparer. Throws NullReferenceException if cmp or eqc is null.

The item comparer is used to determine item equality and item order in the sorted array. The item equality comparer is used only to define an equality comparer for the entire sorted array. This is useful, for instance, if the sorted array is used as an item in a hash-based collection: An equality comparer may be much faster than a comparer.. For instance, two strings of different length are clearly not equal, but if they have a long prefix in common, it may be inefficient for a comparer to determine which one is less than the other one.

Efficiency warning: Random insertions and deletions on a large SortedArray<T> are slow because all items after the insertion or deletion point must be moved at each insertion or deletion. Use a TreeSet<T> if you need to make random insertions and deletions on a sorted collection. Use SortedArray<T> only if the number of items is small or if the sorted array is modified rarely but is accessed frequently by item index or item value.

6.8 Tree-based sets

Class TreeSet<T> implements interfaces IIndexedSorted<T> (sections 1.4.6 and 4.7) and IPersistentSorted<T> (sections 1.4.8 and 4.9) and represents a set of items using a balanced red-black tree; see section 13.10. Item access by index or by item value takes time $O(\log n)$, and item insertion and item deletion take time $O(\log n)$ amortized. A tree set does not allow duplicates, so the property `AllowsDuplicates` is false. From a tree-based set, one can efficiently make a snapshot, or persistent copy; see section 8.5.

Class TreeSet<T> has three constructors:

- `TreeSet<T>()` creates a new tree set with a default item comparer and a default item equality comparer. The default item comparer uses the natural comparer for T; if T does not implement IComparable<T> or IComparable, then NotComparableException is thrown.

- `TreeSet<T>(SCG.IComparer<T> cmp)` creates a new tree set with the given item comparer and a default item equality comparer. Throws NullReferenceException if `cmp` is `null`.

- `TreeSet<T>(SCG.IComparer<T> cmp, SCG.IEqualityComparer<T> eqc)` creates a new tree set with the given item comparer and the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.

The item comparer is used to determine item equality and item order in the tree set. The item equality comparer is used only to define an equality comparer for the entire tree set. This is needed, for instance, if the tree set is used as an item in a hash-based collection.

6.9 Tree-based bags

Class TreeBag<T> implements interfaces IIndexedSorted<T> (sections 1.4.6 and 4.7) and IPersistentSorted<T> (sections 1.4.8 and 4.9) and represents a bag, or multiset, of items using a balanced red-black tree. Item access by index or by item value takes time $O(\log n)$, and item insertion and item deletion take time $O(\log n)$ amortized.

A tree bag allows duplicates, so the property `AllowsDuplicates` is true, and only one representative for each item equivalence class is stored, so `DuplicatesByCounting` is true.

From a tree-based bag, one can efficiently make a snapshot, or persistent copy; see section 8.5.

Class TreeBag<T> has three constructors:

- `TreeBag<T>()` creates a new tree bag with a default item comparer and a default item equality comparer. The default item comparer uses the natural comparer for T; if T does not implement IComparable<T> or IComparable, then NotComparableException is thrown.
- `TreeBag<T>(SCG.IComparer<T> cmp)` creates a new tree bag with the given item comparer and a default item equality comparer. Throws NullReferenceException if `cmp` is `null`.
- `TreeBag<T>(SCG.IComparer<T> cmp, SCG.IEqualityComparer<T> eqc)` creates a new tree bag with the given item comparer and the given item equality comparer, which must be compatible. Throws NullReferenceException if `eqc` is `null`.

The item comparer is used to determine item equality and item order in the tree bag. The item equality comparer is used only to define an equality comparer for the entire tree bag. This is needed, for instance, if the tree bag is used as an item in a hash-based collection.

6.10 Hash sets

Class HashSet<T> implements interface IExtensible<T> (sections 1.4.3 and 4.5) and represents a set of items using a hash table with linear chaining; see section 13.4. Item access by item value takes time $O(1)$, and item insertion and item deletion take time $O(1)$ amortized. A hash set does not allow duplicates, so the property `AllowsDuplicates` is false.

Class HashSet<T> has four constructors:

- `HashSet<T>()` creates a new hash set with a default initial table size (16), a default fill threshold (0.66), and a default item equality comparer.
- `HashSet<T>(SCG.IEqualityComparer<T> eqc)` creates a new hash set with a default initial table size (16), a default fill threshold (0.66), and the given item equality comparer. Throws NullReferenceException if `eqc` is `null`.

- HashSet<T>(int capacity, SCG.IEqualityComparer<T> eqc) creates a new hash set with the given initial table size, a default fill threshold (0.66), and the given item equality comparer. Throws ArgumentException if capacity <= 0. Throws NullReferenceException if eqc is null.
- HashSet<T>(int capacity, double fill, SCG.IEqualityComparer<T> eqc) creates a new hash set with the given initial table size, the given fill threshold, and the given item equality comparer. Throws ArgumentException if capacity <= 0 or if fill < 0.1 or fill > 0.9. Throws NullReferenceException if eqc is null.

In addition to the methods described by the interfaces implemented by HashSet<T>, it has one method:

- ISortedDictionary<int,int> **BucketCostDistribution**() returns a new table bcd reporting the current bucket cost distribution in the internal representation of the hash set. The value of bcd[c] is the number of buckets that have cost c. The bcd table can be used to diagnose whether a performance problem may be due to a bad choice of hash function.

  A *bucket* in a hash table is an equivalence class of items whose hash values map to the same entry in the hash table, for the current table size. The *cost* of a bucket is the number of equality comparisons that must be performed by the item equality comparer to determine that an item is not a member of the bucket. This is the number of equality comparisons made when a lookup fails.

  The largest bucket cost of a hash table gives an upper bound on the execution time of a hash table lookup. A large bucket cost is caused by a poor choice of hash function for the given distribution of items. In particular, if the hash function gives all items the same constant hash code, then the table will have a single bucket with a very large cost.

  For good performance, a hash table should only have buckets with low bucket cost. Use BucketCostDistribution.Keys.FindMax() to find the largest bucket cost in the hash set. As a rule of thumb this number should not be more than 25. See antipattern 129 for typical bucket cost distributions resulting from good and bad hash functions.

6.11 Hash bags

Class HashBag<T> implements interface IExtensible<T> (sections 1.4.3 and 4.5) and represents a bag (multiset) of items using a hash table with linear probing. Item access by item value takes time $O(1)$, and item insertion and item deletion take time $O(1)$ amortized. A hash bag allows duplicates, so the property AllowsDuplicates is true, but duplicate items are not stored explicitly, only the number of duplicates, so DuplicatesByCounting is true.

Class HashBag<T> has four constructors:

- HashBag<T>() creates a new hash bag with a default initial table size (16), a default fill threshold (0.66), and a default item equality comparer.
- HashBag<T>(SCG.IEqualityComparer<T> eqc) creates a new hash bag with a default initial table size (16), a default fill threshold (0.66), and the given item equality comparer. Throws NullReferenceException if eqc is null.
- HashBag<T>(int capacity, SCG.IEqualityComparer<T> eqc) creates a new hash bag with the given initial table size, a default fill threshold (0.66), and the given item equality comparer. Throws ArgumentException if capacity <= 0. Throws NullReferenceException if eqc is null.
- HashBag<T>(int capacity, double fill, SCG.IEqualityComparer<T> eqc) creates a new hash bag with the given initial table size, the given fill threshold, and the given item equality comparer. Throws ArgumentException if capacity <= 0 or if fill < 0.1 or fill > 0.9. Throws NullReferenceException if eqc is null.

6.12 Interval heaps or priority queues

Class IntervalHeap<T> implements interface IPriorityQueue<T> (sections 1.4.12 and 4.10) using an interval heap stored as an array of pairs. The FindMin and FindMax operations, and the indexer's get-accessor, take time $O(1)$. The DeleteMin, DeleteMax, Add and Update operations, and the indexer's set-accessor, take time $O(\log n)$. In contrast to an ordinary priority queue, an interval heap offers both minimum and maximum operations with the same efficiency. The AllowsDuplicates property is true, and DuplicatesByCounting is false. See section 13.11 for implementation details.

Class IntervalHeap<T> has four constructors:

- IntervalHeap<T>() creates a new interval heap with a default capacity (16) and a default item comparer. The default item comparer uses the natural comparer for T; if T does not implement IComparable<T> or IComparable, then NotComparableException is thrown.
- IntervalHeap<T>(SCG.IComparer<T> cmp) creates a new interval heap with a default capacity (16) and the given item comparer. Throws NullReferenceException if cmp is null.
- IntervalHeap<T>(int capacity) creates a new interval heap with at least the given capacity and a default item comparer. The default item comparer uses the natural comparer for T; if T does not implement IComparable<T> or IComparable, then NotComparableException is thrown.
- IntervalHeap<T>(int capacity, SCG.IComparer<T> cmp) creates a new interval heap with at least the given capacity and with the given item comparer. Throws NullReferenceException if cmp is null.

Chapter 7

Dictionary implementations

As shown in figure 7.1 there are two dictionary implementations, namely HashDictionary<K,V> and TreeDictionary<K,V>. The former uses a key equality comparer to compare dictionary keys, and the latter uses a key comparer, or ordering, to compare dictionary keys. The latter implements the ISortedDictionary<K,V> interface; both implement the IDictionary<K,V> interface.

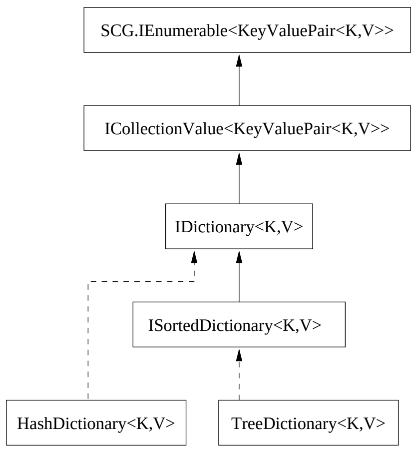


Figure 7.1: The dictionary classes and interfaces. Solid lines indicate a subinterface or subclass relation, and dashed lines indicate an implementation relation.

7.1 Hash-based dictionaries

Class HashDictionary<K,V> implements interface IDictionary<K,V> and represents a dictionary of (key,value) pairs, or entries, using a hash table with linear chaining. Entry access and entry deletion by key takes expected time $O(1)$, and entry insertion takes expected time $O(1)$ also. Enumeration of the keys, values or entries of a hash dictionary does not guarantee any particular enumeration order, but key enumeration order and value enumeration order follow the (key,value) pair enumeration order.

Class HashDictionary<K,V> has three constructors:

- `HashDictionary<K,V>()` creates a new hash dictionary with a default initial table size (16), a default fill threshold (0.66), and a default key equality comparer for key type K, see section 2.3.
- `HashDictionary<K,V>(SCG.IEqualityComparer<K> eqc)` creates a new hash dictionary with a default initial table size (16), a default fill threshold (0.66), and the given key equality comparer. Throws NullReferenceException if `eqc` is `null`.
- `HashDictionary<K,V>(int capacity, double fill, SCG.IEqualityComparer<K> eqc)` creates a new hash dictionary with the given initial table size, the given fill threshold (0.66), and the given key equality comparer. Throws NullReferenceException if `eqc` is `null`.

In addition to the methods described by interface IDictionary<K,V>, class HashDictionary<K,V> has one method:

- `ISortedDictionary<int,int>` **`BucketCostDistribution`**`()` returns a new table `bcd` reporting the current bucket cost distribution in the internal representation of the hash dictionary. The value of `bcd[c]` is the number of buckets that have cost `c`. The `bcd` table can be used to determine whether a performance problem may be due to a bad choice of hash function. For further information, see the description of method `BucketCostDistribution` in section 6.10.

7.2 Tree-based dictionaries

Class TreeDictionary<K,V> implements interface ISortedDictionary<K,V> and represents a dictionary of (key,value) pairs, or entries, using an ordered balanced red-black binary tree. Entry access, entry deletion, and entry insertion take time $O(\log n)$. Enumeration of the keys, values or entries of a tree dictionary follow the key order, as determined by the key comparer

Class TreeDictionary<K,V> has two constructors:

- `TreeDictionary<K,V>()` creates a new tree dictionary with a default key comparer for key type K; see section 2.6. The default key comparer uses the natural comparer for K; if K does not implement IComparable<K> or IComparable, then NotComparableException is thrown.
- `TreeDictionary<K,V>(SCG.IComparer<K> cmp)` creates a new tree dictionary with the given key comparer. Throws NullReferenceException if `cmp` is `null`.

- `TreeDictionary<K,V>(SCG.IComparer<K> cmp, SCG.IEqualityComparer<K> eqc)` creates a new tree dictionary with the given key comparer and the given key equality comparer. Throws NullReferenceException if `cmp` or `eqc` is `null`.

The key comparer is used to determine item equality and item order in the tree dictionary. The key equality comparer is used only to define an equality comparer for the entire dictionary. This is needed if the tree dictionary is used as an item in a hash-based collection.

Chapter 8

Advanced functionality

8.1 List views

A list view represents a contiguous segment of an underlying list and may be used to perform local operations on that list. There may be several, possibly overlapping, views on the same list. All list operations from interface IList<T> (section 1.4.11) apply to views as well, and the term “list” is usually taken to mean “list or view”, whereas a A *proper list* is a list that is not a view. A view has a unique *underlying list* that is a proper list.

Using list views, many operations can be concisely expressed in a way that is efficient both on array lists and linked lists, as shown by the patterns in chapter 9. Any list operation (enumeration, item search, indexing, insertion, removal, sorting, reversal, ...) can be applied also to a list view, thus restricting the operations to a contiguous subsequence of the list. This improves orthogonality and predictability and reduces the number of overloaded method variants needed in the library.

8.1.1 Basic concepts of list views

A view is delimited by a left endpoint and a right endpoint, both of which are at inter-item space in the underlying list. Figure 8.1 shows a list with four items (a, b, c, d) and seven views (A–G).

The *offset* of a view is the number of items before its left endpoint in the underlying list. Property `Offset` reports the offset of a view. In the example, views A and G have offset 0, view F has offset 1, views B and C have offset 2, view D has offset 3, and view E has offset 4.

The *length* of a view is the number of items between the left and right endpoints. Property `Count` reports the length of a view. For a zero-item view, also called an *empty* view, both endpoints are in the same inter-item space. In the example, views A and F have length 2, views B and E have length 0, views C and D have length 1, and view G has length 4.

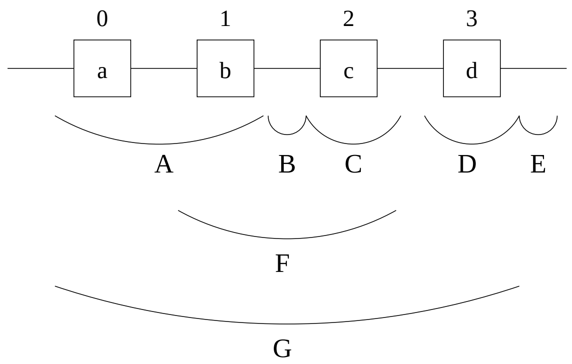


Figure 8.1: Seven views of the same underlying four-item list.

Two views `u` and `w` of the same underlying list *overlap* if they have one or more items in common or if one is a zero-item view strictly inside the other. Equivalently, the left endpoint of one is strictly before the right endpoint of the other, and vice versa. Formally, `u` and `w` overlap if `u.Offset < w.Offset+w.Count` and `w.Offset < u.Offset+u.Count`. It follows that the overlaps relation is symmetric, any non-empty view overlaps with itself, and an empty (zero-item) view overlaps only with views of length two or more. In the example, views A, B and C overlap with F and G but not with each other; view D overlaps with G; and views F and G overlap. Each non-empty view A, C, D, F and G overlaps with itself, but the empty view B overlaps only with the strictly enclosing views F and G, and E does not overlap with any view. The length of an overlap is the number of items the two views have in common; this may be zero but only if one view is empty and strictly inside the other.

A view `u` *contains* a view `w` of the same underlying list if `w` is non-empty and the left and right endpoints of `w` are inside `u`, or if `w` is empty and its endpoints are strictly inside `u`. Formally, if `w.Count > 0` it must hold that `u.Offset <= w.Offset` and `w.Offset+w.Count <= u.Offset+u.Count`, or if `w.Count = 0` it must hold that `u.Offset < w.Offset < u.Offset+u.Count`. It follows that a non-empty view contains itself, that an empty view contains no views, that if two views contain each other then they have the same offset and length, and that a view contains any empty views strictly inside it but not the empty views at its ends. In particular, a proper list contains all its views except the empty views at its ends. Also, if `u` contains `w` then they overlap. In fact, `u` contains `w` if and only if they overlap and the length of the overlap equals the length of `w`. In the example, each of the views A, C and D contains itself only; view F contains B, C and F; and view G contains the views A, B, C, D, F and G but not E.

Section 9.4 presents C# methods corresponding to the above definitions.

8.1.2 Operations on list views

Operations on views include:

- Updates, insertions or deletions of items within a view. These operations immediately affect the underlying list and possibly other views, according to the rules in section 8.1.5.
- All other list properties and methods may be used on a view and will operate on the items within the view only. In particular, one may perform enumeration (and backwards enumeration), search, reversal, sorting, shuffling, clearing, and so on. Again these operations work on the underlying list and affect that list and possibly other views of the list.
- The **`Count`** property is the length of the view, that is, the number of items it contains.
- The **`Offset`** property of a view gives the index of the view's beginning within the underlying list.
- The **`Slide`**`(i)` method slides the view left (if `i<0`) or right (if `i>0`) by the given number of items. Throws ArgumentOutOfRangeException if this would move either endpoint of the view outside the underlying list, that is, if `i+Offset < 0` or `i+Offset+Count > Underlying.Count`.
- The **`Slide`**`(i,n)` method slides the view by `i` items and also sets its size to `n`, thus extending the view to include more items or shrinking it to include fewer. Throws ArgumentOutOfRangeException if this would move either endpoint of the view outside the underlying list, that is, if `i+Offset < 0` or `i+Offset+n > Underlying.Count`.
- `IList<T>` **`Span`**`(IList<T> w)` returns a new view spanned by two existing views, if any. The call `u.Span(w)` produces a new view whose left endpoint is the left endpoint of `u` and whose right endpoint is the right endpoint of `w`. If the right endpoint of `w` is strictly to the left of the left endpoint of `u`, then `null` is returned. The views `u` and `w` must have the same underlying list, or one or both may be that underlying list, and `w` must be non-`null`; otherwise IncompatibleViewException is thrown.

  When `list` is the underlying list of view `u`, then `u.Span(list)` is a view that spans from the left endpoint of `u` to the end of the list, and `list.Span(u)` is a view that spans from the beginning of the list to the right endpoint of `u`.
- `bool` **`TrySlide`**`(i)` returns true if the given view can be slid by `i` items, and in that case slides it exactly as `Slide(i)`; otherwise returns false and does not modify the given view. More precisely, returns true if `i+Offset >= 0` and `i+Offset+Count <= Underlying.Count`. Throws ReadOnlyCollectionException if the view is read-only.

- bool **TrySlide**(i, n) returns true if the given view can be slid by i items and have its length set to n, and in that case slides it exactly as Slide(i, n); otherwise returns false and does not modify the given view. More precisely, returns true if i+Offset >= 0 and i+Offset+n <= Underlying.Count. Throws ReadOnlyCollectionException if the view is read-only.

- The property **Underlying** returns the view's underlying list, or returns null if applied to a proper list. The expression (list.Underlying != null) is true when list is a view and false when it is a proper list. Similarly, the expression (list.Underlying ?? list) evaluates to the underlying list regardless of whether list is a view or itself a proper list.

8.1.3 Creating list views

There are several ways to create a view from a list or view list:

- Method list.**View**(i,n) creates and returns a new view with offset i and length n. Throws ArgumentOutOfRangeException if the view would not fit inside list; or more precisely, if i < 0 or n < 0 or i+n > list.Count.

- Method list.**ViewOf**(x) creates and returns a new view of length 1 containing the first occurrence of item x in list, if any; otherwise returns null.

- Method list.**LastViewOf**(x) creates and returns a new view of length 1 containing the last occurrence of item x in list, if any; otherwise returns null.

These methods work whether list is a view or a proper list. They always produce a view of the underlying list, so the Offset of the resulting view will be relative to that underlying list, not to any view.

8.1.4 One-item views and zero-item views

One-item and zero-item views are surprisingly useful as different kinds of cursors on a list:

- An n-item list list has n distinct one-item views, created by list.View(i,1) where $0 \leq$ i $<$ list.Count. A one-item view points *at* a particular item.

- An n-item list list has $n+1$ distinct zero-item views, created by list.View(i,0) where $0 \leq$ i $\leq$ list.Count. A zero-item view u has u.Count = 0 and represents a position *between*, or before or after, list items.

Zero-item view are conceptually and practically useful as *inter-item cursors*, precisely because there are enough of them to unambiguously point at a pre-item, inter-item, or post-item space.

8.1.5 The effect of inserting or deleting an item

The effect on a view of an operation, such as insertion or deletion, depends on whether the operation is performed on another view or the underlying list, or is performed explicitly on that view. Insertions and deletions to the underlying list or to *another* view before or inside a given view immediately show through the view.

The following rules apply when the operation is performed *on another view or on the underlying list*:

- An insertion at the underlying list's position `i` increases by 1 the offset of view `u` if `i <= u.Offset`.
- An insertion at the underlying list's position `i` increases by 1 the length of view `u` if `u.Offset < i < u.Offset+Count`. It follows that the length of a zero-item or one-item view is never affected by insertions into another view or into the underlying list. This contributes to their usefulness as cursors.
- A deletion at the underlying list's position `i` decreases by 1 the offset of view `u` if `i < u.Offset`.
- A deletion at the underlying list's position `i` decreases by 1 the length of view `u` if `u.Offset <= i < u.Offset+Count`. It follows that deletion from another view or from the underlying list never affects the length of a zero-item view, whereas it might remove the sole item in a one-item view.

Operations performed *explicitly on a given view* `w` follow these rules:

- Insertion explicitly performed on a view increases its length by 1 and leaves its offset unaffected.
- A deletion explicitly performed on a view decreases its length by 1 and leaves its offset unaffected.

For instance, an insertion at the beginning of the underlying list in figure 8.1 is at position `i=0`; it increases the offset of all views A–G by 1 but does not affect their length. However, an insertion explicitly performed at the beginning of view A increases the length of A by 1 and leaves its offset unaffected at 0, and increases the offset of all views B–G by 1 but does not affect their length.

An insertion at the end of the underlying list in figure 8.1 is at position `i=4`; its only effect on the views A–G is to increase the offset of the zero-item view E. In particular, it does not affect the whole-list view G. However, an insertion into view E will increase the length of E but not affect its offset, nor will it affect any of the views A, B, C, D, F, G.

A newly created view is *valid*, but some operations on a list or another view of the list may invalidate the view; see section 8.1.6. All operations on invalid views, except for `IsValid` and `Dispose`, will throw ViewDisposedException.

8.1.6 List views and multi-item list operations

Operations such as `Reverse`, `Sort` and `Shuffle` on a list or a list view `u` potentially affect many items and therefore many views at once.

- `u.Clear()` where `u` is a list or list view affects another view `w` of the same underlying list as follows:
  - If `w` contains `u`, then `w.Count` is reduced by `u.Count`.
  - Otherwise, if `u` contains `w` (but not vice versa), then `w` is cleared: `w.Count` becomes zero, and `w.Offset` becomes `u.Offset`.
  - Otherwise, if `w` is completely to the left of `u`, then `w` is unchanged.
  - Otherwise, if `w` is completely to the right of `u`, then `w.Offset` is reduced by `u.Count`.
  - Otherwise, if `u` and `w` overlap (but neither contains the other) and `w.Offset < u.Offset`, then `w.Count` is reduced by the length of the overlap.
  - Otherwise, if `u` and `w` overlap (but neither contains the other) and `w.Offset > u.Offset`, then `w.Offset` is reduced to `u.Offset` and `w.Count` is reduced by the length of the overlap.

  These cases are exhaustive. For instance, if `w.Offset = u.Offset`, then either `w` is an empty view to the left of `u`, or `w` contains `u`, or `u` contains `w`. The intention is that `u.Clear()` affects `u` and other views in the same way as removing the items of `u` one by one, except that the two procedures will raise different events. In particular, if `u` is a zero-item view, then `u.Clear()` affects no other view, and if `u` is a one-item view, then `u.Clear()` has the same effects as deleting the item at position `u.Offset` by `u.Remove()`.

- `u.Reverse()`, where `u` is a list or view, affects another view `w` of the same underlying list as follows:
  - If `w` contains `u`, then the offset and length of `w` are unchanged.
  - Otherwise, if `u` contains `w` (but not vice versa), then `w` is "mirrored" inside `u` by setting `w.Offset` to `2 * u.Offset + u.Count - v.Offset - v.Count`.
  - Otherwise, if `u` and `w` do not overlap, then `w` is unchanged.
    In particular, any zero-item views at the ends of `u` are unchanged.
  - Otherwise, if `u` and `w` overlap (but none is contained in the other), then `w` is invalidated.
    It follows that `u.Reverse()` does not invalidate any zero-item or one-item views. Also, when `u` is a proper list, then `u.Reverse()` does not invalidate any views at all.

  These cases are exhaustive. Also, when `u` is a zero-item or one-item view, then `u.Reverse()` affects no other view.

- `u.Sort(...)` and `u.Shuffle(...)` where `u` is a list or list view, affect another view `w` of the same underlying list as follows:
  - If `w` contains `u`, then the offset and length of `w` are unchanged.
  - Otherwise, if `u` contains `w` (but not vice versa), then `w` is invalidated.

    It follows that when `u` is a proper list, then all views of `u` except zero-item views at its ends are invalidated.
  - Otherwise, if `u` and `w` do not overlap, then `w` is unchanged.

    In particular, any zero-item views at the ends of `u` are unchanged.
  - Otherwise, if `u` and `w` overlap but none is contained in the other, then `w` is invalidated.

  These cases are exhaustive. Also, when `u` is a zero-item or one-item view, then `u.Sort(...)` and `u.Shuffle(...)` affect no other view.

8.1.7 List views and event handlers

One cannot attach an event handler (section 8.8) to a view, only to the underlying list. Consequently, the item positions reported via the event arguments when an event is raised are the absolute positions in the underlying list, even if the event was raised as a consequence of calling a method on some view.

An attempt to add or remove an event handler on a view will throw InvalidOperationException.

One can determine whether an event such as `ItemInserted(coll, args)` has affected a given view `u` by examining `args` which is an object of class `ItemAtEventArgs`; see section 8.8.6. Let `uo` be the `Offset` and `uc` the `Count` of view `u` *before* the event. Then the event affected the view if `uo <= args.Index && args.Index < uo + uc`.

Also, when an operation such as `Clear` is applied to a view `u`, it raises a CollectionCleared event (and no ItemsRemoved events) on the underlying list. The event argument will be a `ClearedRangeEventArgs` object whose `Full` field is false, whose `Start` field is `uo` and whose `Count` field is `uc`, where `uo` is the `Offset` and `uc` the `Count` of view `u` *before* the `Clear` operation.

8.1.8 Views of a guarded list, and guarded views of a list

One can create a view `vg` of a GuardedList<T> (section 8.2). Clearly, such a view does not permit updates of the underlying list, but one can slide the view, and one can obtain and inspect (but not update) the underlying list `vg.Underlying`. Hence one cannot use a view of a guarded list to hide parts of that list from a client.

Conversely one can put a GuardedList<T> wrapper around a list view to obtain a guarded view `gv` of a list. A guarded view does not permit updating nor sliding, but one can still use it to obtain and inspect (but not update) the underlying list `gv.Underlying`. Hence one cannot use a guarded view of a list to hide parts of that list from a client.

8.2 Read-only wrappers

To provide read-only access to a collection, wrap it as a *guarded* collection. The `IsReadOnly` property of a guarded collection is true. Structural modifications such as adding, inserting, updating or removing items will throw ReadOnlyCollectionException. However, a guarded collection cannot itself prevent modifications inside each of the collection's items. A guarded collection is not a copy of the underlying collection `coll`, just a view of it, so modifications to the underlying collection will be visible through the guarded-collection wrapper.

Typically the guarded collection will be passed to a method `DoWork` or some other consumer that must be prevented from modifying the collection. For instance, for a HashSet<T> collection the appropriate wrapper class is GuardedCollection<T>, and the wrapper could be created and used like this:

```
HashSet<T> coll = ...;
DoWork(new GuardedCollection<T>(coll));
```

The method `DoWork` can be declared to take as argument an ICollection<T>; that interface describes all the members of the GuardedCollection<T> class:

```
void DoWork<T>(ICollection<T> gcoll) { ... }
```

Similar read-only wrappers exist for the other collection classes and for the dictionary classes. For each class the relevant read-only wrapper and the corresponding interface are shown in figure 8.2. To create a read-only list view (section 8.1) use the GuardedList<T> wrapper; the resulting view cannot be slid and its `Underlying` property returns a read-only list.

Class	Read-only wrapper classes	Interface
HashSet<T>	GuardedCollection<T>	ICollection<T>
HashBag<T>	GuardedCollection<T>	ICollection<T>
TreeSet<T>	GuardedIndexedSorted<T>	IIndexedSorted<T>
TreeBag<T>	GuardedIndexedSorted<T>	IIndexedSorted<T>
SortedArray<T>	GuardedIndexedSorted<T>	IIndexedSorted<T>
ArrayList<T>	GuardedList<T>	IList<T>
HashedArrayList<T>	GuardedList<T>	IList<T>
LinkedList<T>	GuardedList<T>	IList<T>
HashedLinkedList<T>	GuardedList<T>	IList<T>
CircularQueue<T>	GuardedQueue<T>	IQueue<T>
HashDictionary<K,V>	GuardedDictionary<K,V>	IDictionary<K,V>
TreeDictionary<K,V>	GuardedSortedDictionary<K,V>	ISortedDictionary<K,V>

Figure 8.2: Read-only wrappers for collection and dictionary classes.

There is no wrapper class for priority queues; a read-only priority queue is not very useful. A read-only FIFO queue, on the other hand, can at least be indexed into. Some additional read-only wrapper classes can be used when developing custom collection classes as described in section 14.1.

8.3 Collections of collections

Sometimes it is useful to create a collection whose items are themselves collections. For instance, one may use a dictionary whose keys are bags of characters (section 11.4), or a dictionary whose keys are sets of integers (section 11.5). The collection that contains other collections as items will be called the *outer* collection and the collections that appear as items in the outer collection will be called *inner* collections.

It is important that the inner collections have appropriate comparers or equality comparers, including hash functions. Assume that the inner collection type is S and that its item type is W; for instance, S may be HashSet<W> or ISorted<W>:

- When the outer collection is hash-based, an item equality comparer of type SCG.IEqualityComparer<S> for the inner collections is created automatically; see section 2.3. Usually this default equality comparer is appropriate.

  For instance, when the inner collection type S implements ISequenced<W>, the default equality comparer is a SequencedCollectionEqualityComparer<S,W> whose equality and hash function take into account the order of items in the inner collections. A sequenced equality comparer will be used when inner collections of type LinkedList<W>, ArrayList<W>, SortedArray<W>, TreeSet<W>, or TreeBag<W> are used as items in a hash-based outer collection.

  When the inner collection type S is unsequenced, that is, implements ICollectionValue<W> but not ISequenced<W>, the default equality comparer will be an UnsequencedCollectionEqualityComparer<T,W> whose equality and hash function ignore the order of items in the inner collections. An unsequenced equality comparer will be used when inner collections of type HashSet<W> or HashBag<W> are used as items in a hash-based outer collection.

- Correctness Warning: Modifications to an inner collection after it has been inserted as an item in an outer collection should be avoided. If the inner collection's equality function, hash function or comparer are affected by the modification, the inner collection may be "lost" and no longer retrievable from the outer collection, which will most likely cause strange errors; see antipattern 130. A safe way to avoid such problems is make a read-only copy of the inner collection before storing it in the outer collection. Tree snapshots are ideal for this purpose; see pattern 86 and section 8.5.

- Correctness Warning: It is possible for a collection to be a member of itself, but equality functions, hash functions and comparers are likely to go into an infinite loop or overflow the call stack when applied to such a collection.

8.4 Generic bulk methods

You may wonder why some methods take a generic type parameter where one does not seem to be needed. For instance, interface IExtensible<T> describes this generic method that takes a type parameter U:

```
void AddAll<U>(SCG.IEnumerable<U> xs) where U : T;
```

One might think that this simpler non-generic method would suffice:

```
void AddAll(SCG.IEnumerable<T> xs);
```

However, the version with the additional generic type parameter is more generally applicable. To see this, consider a class Vehicle with a subclass Car, and assume you have two variables `vehicles` and `cars`, bound to lists of vehicles and cars, respectively:

```
IList<Vehicle> vehicles = ...;
IList<Car> cars = ...;
```

Further suppose you want to add all items from `cars` to the `vehicles` list. Clearly you could use `vehicles.Add(car)` to add a single Car to the `vehicles` list. However, if only the second, non-generic version of `AddAll` is available, you cannot apply `AddAll` to the `cars` list like this:

```
vehicles.AddAll(cars);         // Illegal: cars not of type SCG.IEnumerable<Vehicle>
```

The problem is that although Car is a subtype of Vehicle, IList<Car> is not a subtype of IList<Vehicle> and does not implement SCG.IEnumerable<Vehicle>.

The first, generic version of `AddAll` solves the problem by introducing some extra flexibility. Namely, the additional type parameter U may be instantiated to Car. First, this satisfies the type parameter constraint `U : T` since Car is a subclass of Vehicle, and secondly, it makes `cars` a legal first argument for `AddAll<Car>`:

```
vehicles.AddAll<Car>(cars);    // Legal: cars has type SCG.IEnumerable<Car>
```

The extra type parameter U on `AddAll` partially compensates for the fact that the IList<T> type is neither covariant nor contravariant in the type parameter T.

The following generic bulk methods are described by interfaces IExtensible<T>, ICollection<T>, IList<T> and ISorted<T>:

- `void AddAll<U>(SCG.IEnumerable<U> xs) where U : T`
- `void AddSorted<U>(SCG.IEnumerable<U> xs) where U : T`
- `bool ContainsAll<U>(SCG.IEnumerable<U> xs) where U : T`
- `bool ContainsAny<U>(SCG.IEnumerable<U> xs) where U : T`
- `void InsertAll<U>(int i, SCG.IEnumerable<U> xs) where U : T`
- `void RemoveAll<U>(SCG.IEnumerable<U> xs) where U : T`
- `void RetainAll<U>(SCG.IEnumerable<U> xs) where U : T`

8.5 Snapshots of tree-based collections

Tree-based sets and tree-based bags in C5 support efficient snapshots. In constant time, regardless of the size of the set or bag, one can obtain a persistent (read-only) copy the set or bag. Subsequent modification of the original set or bag will be slightly slower and will use slightly more space; see sections 12.6 and 13.10.

Nevertheless, constant-time snapshots offer several advantages:

- One can iterate (using foreach) over a snapshot of a set or bag while updating the set or bag. Recall that usually one cannot update a collection while iterating over it.
- Snapshots can be used to easily implement rather advanced and very efficient algorithms, such as point location in the plane; see section 11.9.

8.6 Sorting arrays

Class Sorting provides several generic methods for sorting one-dimensional arrays:

- static void **IntroSort**<T>(T[] arr, int i, int n, SCG.IComparer<T> cmp) sorts arr[i..(i+n-1)] using introspective quicksort and comparer cmp. This sorting algorithm is not stable but guaranteed efficient, with worst-case execution time $O(n \log n)$. It must hold that 0 <= i and 0 <= n and i+n <= arr.Length.
- static void **IntroSort**<T>(T[] arr) sorts arr using introspective quicksort and the default comparer for type T. This sorting algorithm is not stable but guaranteed efficient, with execution time $O(n \log n)$ where n is the length of arr.
- static void **InsertionSort**<T>(T[] arr, int i, int n, SCG.IComparer<T> cmp) sorts arr[i..(i+n-1)] using insertion sort and comparer cmp. This method should be used *only* on short array segments, when n is small; say, less than 20.
- static void **HeapSort**<T>(T[] arr, int i, int n, SCG.IComparer<T> cmp) sorts arr[i..(i+n-1)] using heap sort and comparer cmp. This sorting algorithm is not stable but guaranteed efficient, with execution time $O(n \log n)$. In practice it is somewhat slower than introspective quicksort.

In addition to the above array sort methods, the library provides an implementation of merge sort for linked lists, via the Sort method; see page 69.

8.7 Formatting of collections and dictionaries

For debugging purposes it is convenient to format or print collections and dictionaries as strings. However, it is inefficient and inconvenient to print a collection with 50,000 items to the console. Therefore the collection classes provide facilities for limiting the amount of output, displaying ellipses "..." in the resulting string to indicate that some items are not shown. They do so by implementing the C5.IShowable interface (section 3.9), which derives from System.IFormattable.

The results of formatting various kinds of collections and dictionaries are outlined in figure 8.3.

Class	Output format	Note
Array list or sorted array	`[ 0:`x_0`, ⋯, n:`x_n` ]`	
Linked list	`[ `x_0`, ⋯, `x_n` ]`	
Circular queue	`[ `x_0`, ⋯, `x_n` ]`	
Priority queue	`{ `x_0`, ⋯, `x_n` }`	
Hash set or tree set	`{ `x_0`, ⋯, `x_n` }`	
Hash bag or tree bag	`{{ `x_0`(*`m_0`), ⋯, `x_n`(*`m_n`) }}`	if x_i has multiplicity m_i
Tree dictionary	`[ `k_0` => `v_0`, ⋯, `k_n` => `v_n` ]`	
Hash dictionary	`{ `k_0` => `v_0`, ⋯, `k_n` => `v_n` }`	
Record	`( `x_0`, ⋯, `x_n` )`	

Figure 8.3: Formatting of collections with items $x_0, \cdots, x_n$, and of dictionaries with (key,value) pairs $(k_0, v_0), \cdots, (k_n, v_n)$.

In general, `[⋯]` is a sequenced indexed collection; `{⋯}` is an unsequenced collection with set semantics; `{{⋯}}` is an unsequenced collection with bag semantics; and `(⋯)` is a record: a pair, triple or quadruple.

The formatting of a collection or dictionary `coll` as a string can be requested in several different ways:

- `String.Format("{0}", coll)` returns a formatted version of `coll`; this is equivalent to `coll.ToString()`.
- `String.Format("{0:L327}", coll)` returns a formatted version of `coll` using up to roughly 327 characters. Items omitted from the output because of the space limit are displayed using an ellipsis "...". The formatting specifier `"L327"` can be used also in calls to the methods `Console.WriteLine`, `String.Format` and `StringBuilder.AppendFormat`.

Both the above formatting methods draw on the same underlying formatting routines, so the formatted result will be the same. Formatting is applied recursively, so formatting a linked list of sets of integers may produce a result of the form `[{1,2},{1,3}]`.

All collection classes implement the interface System.IFormattable to permit the use of formatting specifications as shown above. The actual implementation of output limitation is specified by interface IShowable; see section 3.9.

8.8 Events: Observing changes to a collection

Often one part of a system needs to be notified when another part of the system is modified in some way. For instance, a graphical display may need to be updated when items are added to or removed from a list of possible choices. To obtain notification about such modifications of a collection, one associates an *event handler*, which is a delegate, with a particular event on the collection. When the collection is modified by an insertion, say, the relevant event is *raised* by calling the associated event handler (if any).

8.8.1 Design principles for collection events

- The events raised by modifying a collection should permit observers to efficiently discover in what way the collection was modified.

  In principle, it would suffice to report that the collection changed somehow and leave it to the observer to determine what happened by inspection of the collection. But that may be very inefficient if the change is the removal of a single item from a million-item collection. Hence C5 provides events that report precisely the item(s) affected by the change.

- It should be safe for the event handler to inspect the collection.

  Therefore events are raised, and event handlers called, only when the collection is in a consistent state, and only after an update has been successfully performed on the collection.

- The sequence of events raised by a collection update should accurately reflect the resulting state of the collection.

  This holds even if the update fails by throwing an exception (other than by a comparer or equality comparer), and it holds even for a bulk update that partially updates the collection and then throws an exception.

- Events should be precise: an event is raised only if there is an actual change to the collection, not just as a consequence of calling a method such as `AddAll(xs)`. Such a call will actually cause no change if `xs` is empty, or if the collection has set semantics and contains all items of `xs` already.

- The use or non-use of events should not change the asymptotic run-time or space complexity of an operation.

  As a consequence of this principle, the `Clear` method does not generate an ItemsRemoved event for every item removed, only a CollectionCleared event and a CollectionChanged event. Note that this holds even for a `Clear` operation on a list view. So `list.View(0,10).Clear()`, which removes the first 10 items from `list`, will not generate ten ItemsRemoved events, but one CollectionCleared event that carries a description of the range of indexes cleared.

Similarly, let `bag` be a bag collection for which `DuplicatesByCounting` is true. Then `bag.Update(x)` raises the three events ItemsRemoved, ItemsAdded and CollectionChanged once each with an event argument giving the multiplicity of `x` in the bag, regardless of that multiplicity, and even though all copies of `x` are updated.

- The `Reverse`, `Shuffle`, and `Sort` operations on a list generate only a CollectionChanged event, no ItemsAdded or ItemsRemoved events. Therefore the precise effect of such multi-item operations cannot be tracked using events; one must inspect the collection to establish its state.
- A sequence of descriptive events signaling the concrete changes is always followed by a CollectionChanged event that signals the end of information about this update.
- A CollectionChanged event is preceded by one or more descriptive events, except in the case of multi-item updates caused by `Reverse`, `Shuffle` or `Sort`, where a CollectionChanged event may follow another CollectionChanged event without any intervening events.

8.8.2 Interfaces, events and event handlers

An event handler for an event X is a delegate of type XHandler. When an event is raised, the handlers currently associated with that event are called. Figure 8.4 shows the events supported by the C5 collection library, as well as the event handler type (delegate type, see section 8.8.5) and event argument type for each event. All events are declared in the ICollectionValue<T> interface.

When an event handler is called, the internal state of the collection is consistent, so it is safe for the event handler to use the collection, for instance query its size, or enumerate its items. However, the event handler should not modify the collection. Such modification may cause confusion at the code that caused the original modification and may lead to an infinite chain of collection changes and event handler calls. Moreover, the collection class implementations do not protect themselves against such changes, so they may lead to inexplicable failures.

For efficiency, an observer should add handlers only for events that are interesting to the observer. Section 9.23 shows some usage patterns for event handlers.

Event	Event handler type	Page	Event argument type	Page
CollectionChanged	CollectionChangedHandler<T>	132	(none)	
CollectionCleared	CollectionClearedHandler<T>	132	ClearedEventArgs	134
ItemsAdded	ItemsAddedHandler<T>	133	ItemCountEventArgs<T>	135
ItemInserted	ItemInsertedHandler<T>	133	ItemAtEventArgs<T>	134
ItemsRemoved	ItemsRemovedHandler<T>	133	ItemCountEventArgs<T>	135
ItemRemovedAt	ItemRemovedAtHandler<T>	133	ItemAtEventArgs<T>	134

Figure 8.4: Events, event handler types, and event argument types.

8.8.3 Methods and the event sequences they raise

Figure 8.5 shows for each method the sequence of events it may raise. A (+) after an event sequence means that the event sequence may be repeated one or more times. Square brackets [. . .] around an event means that it may be omitted. If a call to `AddAll`, `Clear`, `RemoveAll`, `RemoveAllCopies`, `RetainAll` or `RemoveInterval` does not modify the collection, then no events are raised, not even CollectionChanged.

Methods	Event sequence
`Add`	ItemsAdded, CollectionChanged
`AddAll`	ItemsAdded+, CollectionChanged
`Clear`	CollectionCleared, CollectionChanged
`Insert`, `InsertFirst`, ...	ItemInsertedAt, ItemsAdded, CollectionChanged
`InsertAll`	(ItemInsertedAt, ItemsAdded)+, CollectionChanged
`Remove`	ItemsRemoved, CollectionChanged
`RemoveAt`, `RemoveFirst`, ...	ItemRemovedAt, ItemsRemoved, CollectionChanged
`RemoveAll`, `RetainAll`	ItemsRemoved+, CollectionChanged
`RemoveInterval`	CollectionCleared, CollectionChanged
`RemoveAllCopies`	ItemsRemoved, CollectionChanged
`Update`	ItemsRemoved, ItemsAdded, CollectionChanged
`UpdateOrAdd`	[ItemsRemoved,] ItemsAdded, CollectionChanged
`this[i]=x`	ItemRemovedAt, ItemsRemoved, ItemInserted, ItemsAdded, CollectionChanged
`Push`, `Enqueue`	ItemInserted, ItemsAdded, CollectionChanged
`Pop`, `Dequeue`	ItemRemovedAt, ItemsRemoved, CollectionChanged

Figure 8.5: Methods and their event sequences. An event sequence followed by (+) may be raised one or more times. An event within [. . .] may or may not be raised.

8.8.4 Listenable events

All six events are defined on interface ICollectionValue<T> (section 4.2) but some collection implementations support only a subset of the events, as shown in figure 8.6. The results are those reported by property `ListenableEvents` (see page 46) using type EventTypeEnum (section 3.1).

8.8.5 The event handler types

An event handler is a delegate. All event handlers in C5 have return type `void`, but each event handler takes a specific argument that provides information about the event. In all cases, the collection that was modified (the so-called *sender* of the event) is passed to the event handler as the first argument. For list collections, the sender is always the underlying list, even if the modification was performed by calling a method on a view of that list.

Class	Listenable events
`CircularQueue<T>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved
`ArrayList<T>`	All
`LinkedList<T>`	All
`HashedArrayList<T>`	All
`HashedLinkedList<T>`	All
`SortedArray<T>`	None
`WrappedArray<T>`	None
`TreeSet<T>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved
`TreeBag<T>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved
`HashSet<T>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved
`HashBag<T>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved
`IntervalHeap<T>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved
`HashDictionary<K,V>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved
`TreeDictionary<K,V>`	CollectionChanged, CollectionCleared, ItemsAdded, ItemsRemoved

Figure 8.6: Listenable events on collection and dictionary classes.

As explained below, additional event-specific information such as the item that was inserted or deleted, or the position at which the insertion or deletion happen, may also be passed. For list collections, such position information is always the absolute position in the underlying list, even if the modification was performed by calling a method on a view of that list.

Event handler type CollectionChangedHandler<T>

This handler type is declared as

```
delegate void CollectionChangedHandler<T>(Object coll)
```

It is called as `collectionChanged(coll)` when all other events caused by a change to collection `coll` have been raised.

Event handler type CollectionClearedHandler<T>

This handler type is declared as

```
delegate void CollectionClearedHandler<T>(Object coll, ClearedEventArgs args)
```

It is called as `collectionCleared(coll, new ClearedEventArgs(true, n))` when `Clear()` is called on an entire collection `coll`, that is, not on a list view or index range. In this case `n` equals `coll.Count`, the number of items in the collection before the modification.

It is called as `collectionCleared(coll, new ClearedRangeEventArgs(false, i, n))` when `Clear()` is called on some view `u` whose underlying list is `coll`, in which case `i = u.Offset` and `n = u.Count`; or when `coll.RemoveInterval(i,n)` is called and `coll` is an indexed collection and not a list view; or when `u.RemoveInterval(j,n)` is called and `u` is a view with underlying list `coll`, in which case `i = u.Offset+j`.

Event handler type ItemsAddedHandler<T>

This handler type is declared as

```
delegate void ItemsAddedHandler<T>(Object coll, ItemCountEventArgs<T> args)
```

It is called as itemsAdded(coll, new ItemCountEventArgs<T>(x,n)) when n>0 copies of item x are added to collection coll. In the frequent case where just a single copy of x is added, n will be 1. When AddAll(xs) is used to add items to the collection, then the ItemsAddedHandler<T> is called repeatedly, once for each item that is actually added to the collection, followed by a final call to a CollectionChangedHandler<T> if any items were added.

Event handler type ItemInsertedHandler<T>

This handler type is declared as

```
delegate void ItemInsertedHandler<T>(Object coll, ItemAtEventArgs<T> args)
```

It is called as itemInserted(coll, new ItemAtEventArgs<T>(x, i)) when an item x is inserted at position i in collection coll, by Insert, InsertFirst or InsertLast or by aliases such as Push, Pop, Enqueue or Dequeue. When AddAll(xs) is used to add items starting at a particular position in the collection, then the ItemsAdded-Handler<T> is called repeatedly, once for each item that is actually added to the collection, followed by a final call to a CollectionChangedHandler<T> if any items were added.

Event handler type ItemsRemovedHandler<T>

This handler type is declared as

```
delegate void ItemsRemovedHandler<T>(Object coll, ItemCountEventArgs<T> args)
```

It is called as itemsRemoved(coll, new ItemCountEventArgs<T>(x, n)) when n > 0 copies of item x are removed from collection coll, whether by Remove, Update, Update-OrAdd or similar, or by overwriting using an indexer. In the frequent case where just a single copy of x is removed, n is 1.

Event handler type ItemRemovedAtHandler<T>

This handler type is declared as

```
delegate void ItemRemovedAtHandler<T>(Object coll, ItemAtEventArgs<T> args)
```

It is called as itemRemovedAt(coll, new ItemAtEventArgs<T>(x, i)) when item x is removed from position i in collection coll, by RemoveAt, RemoveFirst or RemoveLast. Every ItemRemovedAt event is always followed by an ItemsRemoved event with the same collection argument coll and the item argument x.

8.8.6 The event argument types

These four classes are used to provide information about collection events. They all derive from System.EventArgs in the CLI (or .NET Framework) class library.

Event argument type ClearedEventArgs

This class and its subclasses describe a CollectionCleared event that concerns all or part of a collection. The class derives from System.EventArgs and has two public fields and one constructor:

- Read-only field `int` **`Count`** is the number of items cleared by the operation.
- Read-only field `bool` **`Full`** is true if the entire collection (as opposed to a list view or an index range) was cleared. It is false if a list view or index range was cleared, even if the view or index range comprised the entire collection.
- Constructor **`ClearedEventArgs`**`(bool full, int n)` creates a new cleared event argument object with `Full` equal to `full` and count `n`.

Event argument type ClearedRangeEventArgs

This class describes a CollectionCleared event that concerns part of a collection, such as a list view or index range. The class derives from ClearedEventArgs from which it inherits fields `Full` and `Count`; in addition it has one field and one constructor:

- Read-only field `int?` **`Start`** is the position of the first item in the range deleted, when known. The position is known if `Start.HasValue` is true, or equivalently, `Start != null`. The `Start` position may be unknown if an operation is performed through a view on a HashedLinkedList<T>, but is always known for operations on array lists, hashed array lists and linked lists.
- Constructor **`ClearedRangeEventArgs`**`(int i, int n)` creates a cleared range event argument object with `Full` being false, offset `i` and count `n`.

Event argument type ItemAtEventArgs<T>

This class derives from System.EventArgs and has two public fields and one constructor:

- Read-only field `int` **`Index`** is the index at which the insertion or deletion occurred.
- Read-only field `T` **`Item`** is the item that was inserted or deleted.
- Constructor **`ItemAtEventArgs`**`<T>(T x, int i)` creates a new event argument object with item `x` and known index `i`.

Event argument type ItemCountEventArgs<T>

This class derives from System.EventArgs and has two public fields and one constructor:

- Read-only field `int` **`Count`** is the multiplicity with which the insertion or deletion occurred. The multiplicity will always be greater than 0 for events raised by the C5 collection classes; it is 1 when only a single copy of an item was added or removed; and it may be greater than 1 when manipulating collections that have bag semantics and for which `DuplicatesByCounting` is true,
- Read-only field `T` **`Item`** is the item that was inserted or removed.
- Constructor **`ItemCountEventArgs`**`<T>(T x, int n)` creates a new event argument object with item `x` and count `n`.

8.9 Cloning of collections

Collections that implement IExtensible<T> are *cloneable*, and so are all dictionaries. In particular, they implement the interface System.ICloneable which describes this method:

- `Object` **`Clone`**`()` creates and returns a shallow copy of the given collection or dictionary.

Cloning returns a new collection or dictionary containing the same items as the given one, and having the same comparer, equality comparer and so on. Cloning produces a shallow copy: it does not attempt to clone the collection's items themselves.

In general, when `coll` is an extensible collection or a dictionary, then

```
newColl = coll.Clone();
```

is equivalent to

```
newColl = ... new collection of appropriate type ...;
newColl.AddAll(coll);
```

but the former is usually more efficient. As a consequence of this, the clone of a list view is not a new list view, but a new list containing the same items as the given view.

The clone of a guarded collection is a guarded collection of the same kind.

8.10 Thread safety and locking

The C5 collection and dictionary implementations are not thread-safe: modifying the same collection or dictionary from multiple concurrent threads may corrupt internal data structures and produce arbitrary behavior. This is the case also for the Java's collection classes and for the standard C#/.NET collection classes in System.Collections.Generic.

For safe concurrent use of a collection or dictionary, lock on an object `sync` before manipulating the collection(s):

```
private static readonly Object sync = new Object();

public static void SafeAdd<T>(IExtensible<T> coll, T x) {
  lock (sync) {
    coll.Add(x);
  }
}
```

The `sync` object should be private and not exposed to clients, because one client thread could block other threads by — intentionally or unintentionally — locking on that object.

Locking is even more important if an update affects more than one collection or dictionary:

```
private static readonly Object sync = new Object();

public static void SafeMove<T>(ICollection<T> from, ICollection<T> to) {
  lock (sync)
    if (!from.IsEmpty) {
      T x = from.Choose();
      Thread.Sleep(0);       // yield processor to other threads
      from.Remove(x);
      to.Add(x);
    }
}
```

Preferably lock on a single object as shown above. If multiple locks need to be obtained, there is a risk of introducing a deadlock, unless the locks are always consistently obtained in the same order.

While multi-threaded updates require locking, read access does not: multiple threads can query the same collection or dictionary, or enumerate it, at the same time without locking.

However, recall that while enumerating a collection or dictionary (with `foreach` or similar), one must not be modify it, neither from the enumerating thread nor from a different one. Doing so will throw CollectionModifiedException at the first subsequent step of the enumeration; see section 3.7.2. A tree-based collection can be safely enumerated while modifying it; see pattern 66.

Chapter 9

Programming patterns in C5

9.1 Patterns for read-only access

A collection or dictionary can be made read-only by wrapping it as a *guarded* collection; see section 8.2 for an overview. Then the guarded collection can be passed to other methods or objects without risk of their directly modifying it.

Pattern 1 Read-only access to a hash-based collection
Typically a guarded collection will be passed to a method `DoWork` or some other consumer that must be prevented from modifying the collection. For a HashSet<T> or HashBag<T> collection the appropriate wrapper class is GuardedCollection<T>, which implements interface ICollection<T>:

```
ICollection<T> coll = new HashSet<T>();
...
DoWork(new GuardedCollection<T>(coll));
```

The method `DoWork` can be declared to take as argument an ICollection<T>; that interface describes all the members of the GuardedCollection<T> class:

```
static void DoWork<T>(ICollection<T> gcoll) { ... }
```

Pattern 2 Read-only access to an indexed collection
The GuardedIndexedSorted<T> wrapper class implements all members of the interface IIndexedSorted<T> and is used to provide read-only access to a TreeSet<T>, TreeBag<T> or SortedArray<T>.

```
IIndexedSorted<T> coll = new TreeSet<T>();
...
DoWork(new GuardedIndexedSorted<T>(coll));

static void DoWork<T>(IIndexedSorted<T> gcoll) { ... }
```

Pattern 3 Read-only access to a list
The GuardedList<T> wrapper class implements the IList<T> interface and is used to provide read-only access to an ArrayList<T>, HashedArrayList<T>, LinkedList<T>, HashedLinkedList<T>, or WrappedArray<T>.

```
IList<T> coll = new ArrayList<T>();
...
DoWork(new GuardedList<T>(coll));

static void DoWork<T>(IList<T> gcoll) { ... }
```

Pattern 4 Read-only access to a hash dictionary
The GuardedDictionary<K,V> wrapper class implements the IDictionary<K,V> interface and is suitable for providing read-only access to a HashDictionary<K,V>.

```
IDictionary<K,V> dict = new HashDictionary<K,V>();
...
DoWork(new GuardedDictionary<K,V>(dict));

static void DoWork<K,V>(IDictionary<K,V> gdict) { ... }
```

Pattern 5 Read-only access to a tree dictionary
The GuardedSortedDictionary<K,V> wrapper class implements the interface ISortedDictionary<K,V> and is used to provide read-only access to a TreeDictionary<K,V>.

```
ISortedDictionary<K,V> dict = new TreeDictionary<K,V>();
...
DoWork(new GuardedSortedDictionary<K,V>(dict));

static void DoWork<K,V>(ISortedDictionary<K,V> gdict) { ... }
```

9.2 Patterns using zero-item views

A zero-item view can be used to point *between* any two list items, or *before* or *after* a list item, rather than point *at* a list item. This is useful for indicating where to insert new items; see pattern 12. Let `view` be a zero-item view of a list `list`.

Pattern 6 Get the number of list items before (zero-item) view

```
view.Offset
```

Pattern 7 Get the number of list items after zero-item view

```
view.Underlying.Count - view.Offset
```

Pattern 8 Move the view one item to the left
The first alternative throws ArgumentOutOfRangeException if there is no item before the view, whereas the second alternative just returns false.

```
view.Slide(-1)                view.TrySlide(-1)
```

Pattern 9 Move the view one item to the right
The first alternative throws ArgumentOutOfRangeException if there is no item after the view, whereas the second alternative just returns false.

```
view.Slide(+1)                view.TrySlide(+1)
```

Pattern 10 Test whether (zero-item) view is at beginning of underlying list

```
view.Offset == 0
```

Pattern 11 Test whether zero-item view is at end of underlying list

```
view.Offset == view.Underlying.Count
```

Pattern 12 Insert item at position indicated by zero-item view
An item `x` can be inserted at the inter-item space pointed to by a zero-item view in two ways. This will insert `x` into the underlying list and into the view, which will become a one-item view pointing *at* the item, provided insertion succeeds and returns true:

```
view.Add(x)
```

This will insert `x` into the underlying list (and into any overlapping views), but the given view will remain a zero-item view that points before the newly inserted item:

```
list.Insert(view, x)
```

Pattern 13 Delete the item before the view
This throws ArgumentOutOfRangeException if there is no item before the view. The call to `RemoveFirst` is possible because `Slide` returns the view after sliding it.

```
view.Slide(-1,1).RemoveFirst()
```

Pattern 14 Delete the item after the view
This throws ArgumentOutOfRangeException if there is no item after the view. The call to `RemoveFirst` is possible because `Slide` returns the view after sliding it.

```
view.Slide(0,1).RemoveFirst()
```

Pattern 15 Get a zero-item view at the left or right end of a list or view
These operations succeed whenever `list` is a proper list or a valid view.

```
public static IList<T> LeftEndView<T>(IList<T> list) {
  return list.View(0,0);
}
public static IList<T> RightEndView<T>(IList<T> list) {
  return list.View(list.Count,0);
}
```

9.3 Patterns using one-item views

A one-item view can be used to point *at* any list item, rather than *between* items as does a zero-item view. This can be used to delete the item, replace the item, insert new items before or after it, and in general to reach nearby items. For instance, the `IList<T>` methods

```
IList<T> ViewOf(T y)
IList<T> LastViewOf(T y)
```

return a one-item view of `y`, or return `null` if `y` is not in the list. If the list is a HashedArrayList<T> or a HashedLinkedList<T> then these operations are fast; if the list is just an ArrayList<T> or a LinkedList<T>, they require a linear search and will be slow unless the list is short.

Note that a one-item view is a kind of *item cursor* on a list, pointing *at* an item in a list or view. There can be several cursors on the same list at the same time. Insertion into a one-item view increases its length (`Count`), but insertion into other views or into the underlying list never affects the length (but possibly the offset) of a one-item view. Deletion from another view or from the underlying list may delete the sole item in a one-item view and thus reduce its length.

When `list` is an IList<T>, the following code patterns may be used.

Pattern 16 Get the sequence predecessor of `y`
This throws NullReferenceException if `y` is not in the `list` and throws ArgumentOutOfRangeException if `y` has no predecessor.

```
public static T SequencePredecessor<T>(IList<T> list, T y) {
  return list.ViewOf(y).Slide(-1)[0];
}
```

Pattern 17 Get the sequence predecessor `x` of `y` without exceptions
This returns true and sets `x` to the predecessor of `y` if `y` is in the list and has a predecessor; otherwise returns false.

```
public static bool SequencePredecessor<T>(IList<T> list, T y, out T x) {
  IList<T> view = list.ViewOf(y);
  bool ok = view != null && view.TrySlide(-1);
  x = ok ? view[0] : default(T);
  return ok;
}
```

Note the use of `TrySlide` to determine whether there is a predecessor for `y`. Using `view.Offset >= 1` or similar would needlessly compute the precise offset of `y`, when only one bit of information is needed: whether the view can be slid left by one item.

Pattern 18 Get the sequence successor of first (leftmost) `y`
This throws NullReferenceException if `y` is not in the `list` and throws ArgumentOutOfRangeException if `y` has no successor.

```
public static T SequenceSuccessor<T>(IList<T> list, T y) {
  return list.ViewOf(y).Slide(+1)[0];
}
```

Pattern 19 Get the sequence successor `x` of first `y`, without exceptions
This returns true and sets `x` to the successor of `y` if `y` is in the list and has a successor; otherwise returns false. It uses `TrySlide` for the same reason as pattern 17.

```
public static bool SequenceSuccessor<T>(IList<T> list, T y, out T x) {
  IList<T> view = list.ViewOf(y);
  bool ok = view != null && view.TrySlide(+1);
  x = ok ? view[0] : default(T);
  return ok;
}
```

Pattern 20 Insert a new item `x` as sequence successor to first `y`
This operation works both when `list` is a proper list and when it is a view. It succeeds provided `list` contains `y` and is not read-only, and throws NullReferenceException if `y` is not in `list`.

```
public static void InsertAfterFirst<T>(IList<T> list, T x, T y) {
  list.Insert(list.ViewOf(y), x);
}
```

The call `ViewOf(y)` finds the first (leftmost) one-item view that contains `y` (if any), and then `Insert` inserts `x` into `list` at the end of that view, that is, after `y`. Using `LastViewOf(y)` one can insert after the last (rightmost) occurrence of `y` instead.

When `list` is a view, this is subtly different from `list.ViewOf(y).InsertLast(x)`, which will insert `x` into the underlying proper list, but not into the view `list`.

Pattern 21 Insert a new item x as sequence predecessor to first y
This operation works both when `list` is a proper list and when it is a view. It succeeds provided `list` contains y and is not read-only, and throws NullReferenceException if y is not in `list`.

```
public static void InsertBeforeFirst<T>(IList<T> list, T x, T y) {
  list.Insert(list.ViewOf(y).Slide(0, 0), x);
}
```

The call `ViewOf(y)` finds the first (leftmost) one-item view that contains y (if any); then `Slide(0, 0)` makes it a zero-item view before the occurrence of y; and then `Insert` inserts x at that position. Using `LastViewOf(y)` one can insert before the last (rightmost) occurrence of y instead.

Pattern 22 Delete the sequence predecessor of y
This operation throws NullReferenceException is y is not in the list.

```
public static T RemovePredecessorOfFirst<T>(IList<T> list, T y) {
  return list.ViewOf(y).Slide(-1).Remove();
}
```

Pattern 23 Delete the sequence successor of y
This operation throws NullReferenceException is y is not in the list.

```
public static T RemoveSuccessor<T>(IList<T> list, T y) {
  return list.ViewOf(y).Slide(+1).Remove();
}
```

9.4 Patterns using views

Pattern 24 Allocating a view with the `using` statement
This method replaces the first occurrence of each x from xs by y in `list`. For each x in xs that appears in `list`, a new view is created that contains an item equal to x. That item is removed (so the view becomes empty) and y is inserted instead.

```
public static void ReplaceXsByY<T>(HashedLinkedList<T> list, T[] xs, T y) {
  foreach (T x in xs) {
    using (IList<T> view = list.ViewOf(x)) {
      if (view != null) {
        view.Remove();
        view.Add(y);
      }
    }
  }
}
```

By allocating the view variable in the using statement we make sure that the view is disposed (invalidated) immediately after its last use. This is a sensible thing to do, because the time to insert or remove items in a list grows with the number of live views on that list.

Pattern 25 Index of left or right endpoint of a view
The right and left endpoints of a view is the index of its first and last item in the underlying list.

```
public static int LeftEndIndex<T>(IList<T> view) {
  return view.Offset;
}
public static int RightEndIndex<T>(IList<T> view) {
  return view.Offset + view.Count;
}
```

Pattern 26 Determine whether views overlap
Two views u and w of the same underlying list overlap if they have some item in common or one is an empty view strictly inside the other.

```
public static bool Overlap<T>(IList<T> u, IList<T> w) {
  if (u.Underlying == null || u.Underlying != w.Underlying)
    throw new ArgumentException("views must have same underlying list");
  else
    return u.Offset < w.Offset+w.Count && w.Offset < u.Offset+u.Count;
}
```

Pattern 27 Determine length of overlap of views
The length of the overlap of views u and w is the number of items they have in common. If one view is a zero-item view inside another view, the length of the overlap may be zero even though the views overlap.

```
public static int OverlapLength<T>(IList<T> u, IList<T> w) {
  if (Overlap(u, w))
    return Math.Min(u.Offset+u.Count, w.Offset+w.Count)
         - Math.Max(u.Offset, w.Offset);
  else
    return -1; // No overlap
}
```

Pattern 28 Determine whether one view contains another view
View u contains view w of the same underlying list if w is non-empty and both the left and right endpoints of w are within u, or if w is empty and strictly inside u.

```
public static bool ContainsView<T>(IList<T> u, IList<T> w) {
  if (u.Underlying == null || u.Underlying != w.Underlying)
    throw new ArgumentException("views must have same underlying list");
  else
    if (w.Count > 0)
      return u.Offset <= w.Offset && w.Offset+w.Count <= u.Offset+u.Count;
    else
      return u.Offset < w.Offset && w.Offset < u.Offset+u.Count;
}
```

Pattern 29 Determine whether two lists or views have the same underlying list
The requirement of u and w having the same underlying list may be relaxed by permitting u or w or both to be proper lists, not views. Then we get: Two lists or views u and w have the same underlying list if both are the same proper list, or both are views with the same underlying list, or one is a view whose underlying list is the other one. This method intentionally throws NullReferenceException if either u or w is null.

```
public static bool SameUnderlying<T>(IList<T> u, IList<T> w) {
  return (u.Underlying ?? u) == (w.Underlying ?? w);
}
```

Pattern 30 Use views to find index of the first item that satisfies a predicate
This pattern shows a possible implementation of method FindFirstIndex(p) from interface IList<T> using views. It is efficient for array lists as well as linked lists. This method creates a view and slides it over the list, from left to right. For each list item x it evaluates the predicate p(x) and if the result is true, returns the offset of the view, which is the index of the first item satisfying p.

Note how the view is initially empty and points at the beginning of the list, before the first item (if any). Then the view is extended to length 1 if possible, and p is applied to the item within the view. If the result is false, the view is slid right by one position and shrunk to length 0 at the same time (always possible at that point), and the procedure is repeated.

```
public static int FindFirstIndex<T>(IList<T> list, Fun<T,bool> p) {
  using (IList<T> view = list.View(0, 0)) {
    while (view.TrySlide(0, 1)) {
      if (p(view.First))
        return view.Offset;
      view.Slide(+1, 0);
    }
  }
  return -1;
}
```

Pattern 31 Use views to find index of the last item that satisfies a predicate
This pattern shows a possible implementation of method `FindLastIndex(p)` from interface IList<T> using views, which is efficient for array lists as well as linked lists. The method creates a view and slides it over the list, from right to left. For each list item `x` it evaluates the predicate `p(x)` and if the result is true, returns the offset of the view, which is the index of the last item satisfying `p`. Note how the view is initially empty and points at the end of the list, after the last item (if any). Then the view is slid left by 1 position and extended to length 1, if possible, and `p` is applied to the item at that position.

```
public static int FindLastIndex<T>(IList<T> list, Fun<T,bool> p) {
  using (IList<T> view = list.View(list.Count, 0)) {
    while (view.TrySlide(-1, 1)) {
      if (p(view.First))
        return view.Offset;
    }
  }
  return -1;
}
```

Pattern 32 Use views to find the indexes of all items equal to a given one
This method uses `ViewOf` on ever shorter tails (views) of a list to find all indexes of items equal to `x`. At any time, the view `tail` represents the list suffix that begins at `tail.Offset`. Using `ViewOf`, the suffix is searched for the first occurrence of `x`, if any; the `tail` is updated to point at that occurrence, the left endpoint of `tail` is slid past the occurrence, `tail` is extended to span the rest of the list, and the search is repeated.

```
public static SCG.IEnumerable<int> IndexesOf<T>(IList<T> list, T x) {
  IList<T> tail = list.View(0, list.Count);
  tail = tail.ViewOf(x);
  while (tail != null) {
    yield return tail.Offset;
    tail = tail.Slide(+1,0).Span(list);
    tail = tail.ViewOf(x);
  }
}
```

9.5 Patterns for item search in a list

Let `list` be an IList<T>, such as ArrayList<T>, LinkedList<T>, HashedArrayList<T>, HashedLinkedList<T>, or WrappedArray<T>.

Pattern 33 Find the first (leftmost) position of item `x` in `list`

```
int j = list.IndexOf(x);
if (j >= 0) {
  // x is a position j in list
} else {
  // x is not in list
}
```

Pattern 34 Find the last (rightmost) position of item `x` in `list`

```
int j = list.LastIndexOf(x);
if (j >= 0) {
  // x is at position j in list
} else {
  // x is not in list
}
```

Pattern 35 Find the first (leftmost) index of item `x` in the sublist `list[i..i+n-1]`

```
int j = list.View(i,n).IndexOf(x);
if (j >= 0) {
  // x is at position j+i in list
} else {
  // x is not in list[i..i+n-1]
}
```

Pattern 36 Find the last (rightmost) index of item `x` in the sublist `list[i..i+n-1]`

```
int j = list.View(i,n).LastIndexOf(x);
if (j >= 0) {
  // x is at position j+i in list
} else {
  // x is not in list[i..i+n-1]
}
```

Let `n` be the length of the of the list (or view).

- If `list` is an ArrayList<T>, a LinkedList<T>, or a WrappedArray<T>, the search takes time $O(n)$.
- If `list` is a HashedArrayList<T> or a HashedLinkedList<T>, the search takes time $O(1)$.

9.6 Item not found in indexed collection

When the value `x` searched for in an indexed collection is not found, the `IndexOf(x)` and `LastIndexOf(x)` methods return a number `j` < 0.

In the case of a sorted collection, the negative return value is `~k`, the one's complement of `k`, where `k >= 0` is an index such that if `x` is inserted at position `k` then the collection remains sorted.

In the case of a non-sorted collection, the negative return value is `~n`, where `n` equals `Count`, the number of items in the collection.

Hence the pattern below works regardless of the implementation of the indexed collection.

Pattern 37 Inserting item in indexed collection if not found
If `x` is not in the collection, then `coll.Insert(~j, x)` will be equivalent to `coll.Add(x)` on unsorted list implementations such as ArrayList<T> and LinkedList<T>, and therefore efficient. On SortedArray<T> it will insert `x` where it belongs to keep the array sorted.

```
int j = coll.IndexOf(x);
if (j >= 0) {          // x is at coll[j]
  ...
} else {               // x is not in coll, but belongs at ~j
  coll.Insert(~j, x);
  ...
}
```

9.7 Patterns for removing items from a list

Let `list` be an IList<T>, such as ArrayList<T>, HashedArrayList<T>, LinkedList<T>, or HashedLinkedList<T>.

Pattern 38 Removing all items of the list

```
list.Clear()
```

Pattern 39 Removing items with indexes `i...i+n-1`
There are two obvious ways to remove all items in the range of indexes `i...i+n-1` from a list, shown below. Both have the same effect and both raise a CollectionCleared event followed by a CollectionChanged event, unlike multiple `Remove` operations, which would raise multiple ItemsRemoved events followed by a CollectionChanged event (see section 8.8.1).

```
list.RemoveInterval(i,n)                list.View(i,n).Clear()
```

Pattern 40 Removing items with indexes i and higher
As in the previous pattern, there are two obvious ways to remove all items with indexes i and higher. Both have the same effect and both raise a CollectionCleared event followed by a CollectionChanged event.

```
list.RemoveInterval(i, list.Count-i)          list.View(i, list.Count-i).Clear()
```

The Clear operations themselves take time $O(1)$ for LinkedList<T> and takes time $O(\texttt{list.Count} - n)$ for ArrayList<T>. View creation is inefficient for a LinkedList<T> unless the beginning or the end of the view is near the beginning or end of list. In general, index-based operations on linked lists are slow and should be avoided.

9.8 Patterns for predecessor and successor items

A sorted collection, such as a SortedArray<T>, TreeSet<T> or TreeBag<T>, makes it easy and efficient the predecessor or successor item of a given value. That is, given a value y, to find the collection's least item greater than y, or the collection's greatest item less than y.

Pattern 41 Find the least item x greater than or equal to y
Let coll be an ISorted<T> and y a T value. This expression returns the least item in coll greater than or equal to y, if any; otherwise throws NoSuchItemException.

```
coll.WeakSuccessor(y)
```

Pattern 42 Find the greatest item x less than or equal to y
Let coll be an ISorted<T> and y a T value. This expression returns the greatest item in coll less than or equal to y, if any; otherwise throws NoSuchItemException.

```
coll.WeakPredecessor(y)
```

Pattern 43 Find the least item x greater than y
Let coll be an ISorted<T> and y a T value. This expression returns the least item in coll greater than y, if any; otherwise throws NoSuchItemException.

```
coll.Successor(y)
```

Pattern 44 Find the greatest item x less than y
Let coll be an ISorted<T> and y a T value. This expression returns the greatest item in coll less than y, if any; otherwise throws NoSuchItemException.

```
x = coll.Predecessor(y);
```

The patterns 41 to 44 above all throw an exception in case the (weak) predecessor or successor does not exist. Using the `Cut` method on ISorted<T> and a `y` value of type T, where T implements IComparable<T>, one can avoid this exception, using instead a boolean success indicator. This is preferable because the code to catch the exception is cumbersome, and throwing it is slow.

Pattern 45 Find the least item x greater than or equal to y, without exceptions
This method returns true and assigns to `ySucc` the least item in `coll` greater than or equal to `y`, if any; otherwise returns false.

```
public static bool WeakSuccessor<T>(ISorted<T> coll, T y, out T ySucc)
  where T : IComparable<T>
{
  T yPred;
  bool hasPred, hasSucc,
    hasY = coll.Cut(y, out yPred, out hasPred, out ySucc, out hasSucc);
  if (hasY)
    ySucc = y;
  return hasY || hasSucc;
}
```

Pattern 46 Find the greatest item x less than or equal to y, without exceptions
This method returns true and assigns to `yPred` the greatest item in `coll` less than or equal to `y`, if any; otherwise returns false.

```
public static bool WeakPredecessor<T>(ISorted<T> coll, T y, out T yPred)
  where T : IComparable<T>
{
  T ySucc;
  bool hasPred, hasSucc,
    hasY = coll.Cut(y, out yPred, out hasPred, out ySucc, out hasSucc);
  if (hasY)
    yPred = y;
  return hasY || hasPred;
}
```

Pattern 47 Find the least item x greater than y, without exceptions
This method returns true and assigns to `ySucc` the least item in `coll` greater than `y`, if any; otherwise returns false.

```
public static bool Successor<T>(ISorted<T> coll, T y, out T ySucc)
  where T : IComparable<T>
{
  bool hasPred, hasSucc;
  T yPred;
  coll.Cut(y, out yPred, out hasPred, out ySucc, out hasSucc);
  return hasSucc;
}
```

Pattern 48 Find the greatest item x less than y, without exceptions
This method returns true and assigns to yPred the greatest item in coll less than y, if any; otherwise returns false.

```
public static bool Predecessor<T>(ISorted<T> coll, T y, out T yPred)
  where T : IComparable<T>
{
  bool hasPred, hasSucc;
  T ySucc;
  coll.Cut(y, out yPred, out hasPred, out ySucc, out hasSucc);
  return hasPred;
}
```

9.9 Patterns for subrange iteration

A sorted collection — one whose items are ordered by an item comparer — can be enumerated in two ways: *forwards* (in ascending item order) and *backwards* (in descending item order). In C5, sorted collections implement IDirectedEnumerable and therefore easily support enumeration in either direction. Furthermore, enumeration may start in three different ways: at the collection's least item, at a specified item x1 (inclusive), or at x1 (exclusive). Similarly, enumeration can end in three different ways: at the collection's greatest item, at x2 (inclusive), or at x2 (exclusive). Thus there are nine possible subranges and two directions. All nine forwards and two of the backwards possibilities are illustrated by patterns below; an overview is given in figure 9.1.

In the following patterns, let coll be an ISorted<T>, for instance a SortedArray<T> or a TreeSet<T> or TreeBag<T>.

Pattern 49 Iteration over sorted collection
Let coll be an ISorted<T>. To iterate forwards over all items of coll:

```
foreach (T x in coll) {
  ... use x ...
}
```

Pattern 50 Descending-order iteration over sorted collection
Let coll be an ISorted<T>. To iterate backwards over all items of coll:

```
foreach (T x in coll.Backwards()) {
  ... use x ...
}
```

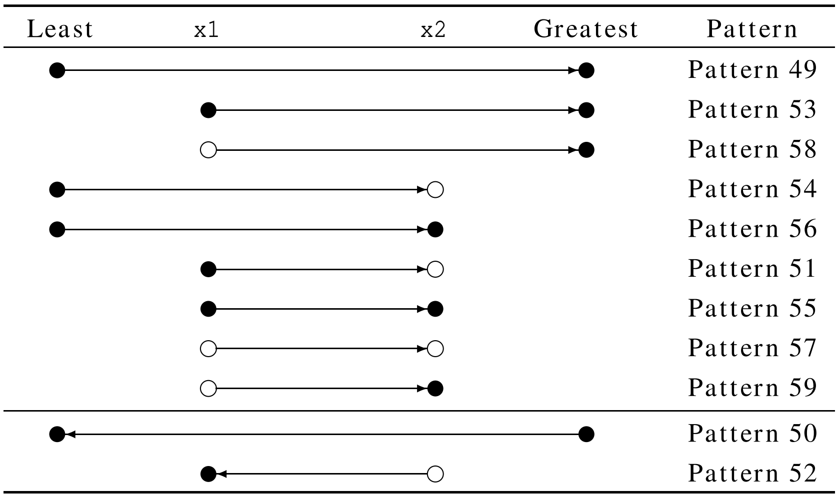


Figure 9.1: All nine forwards and two examples of backwards iteration patterns for sorted collections. The filled-in disk • means endpoint included, the open circle ∘ means endpoint excluded.

Pattern 51 Iteration from `x1` (inclusive) to `x2` (exclusive)
Let `coll` be an ISorted<T>. To iterate forwards over all items between `x1` inclusive and `x2` exclusive, do as below. This does nothing if `x1` is greater than or equal to `x2`, that is, `x1.CompareTo(x2) >= 0`, or if `x1` is greater than the greatest item or `x2` less than or equal to the least item in the collection.

```
foreach (T x in coll.RangeFromTo(x1, x2)) {
  ... use x ...
}
```

Pattern 52 Descending-order iteration from `x2` (exclusive) to `x1` (inclusive)
Let `coll` be an ISorted<T>. To iterate backwards over all items between `x1` inclusive and `x2` exclusive, do as follows. Like pattern 51, this one does nothing if `x1` is greater than or equal to `x2`, or if `x1` is greater than the greatest item or `x2` less than or equal to the least item in the collection.

```
foreach (T x in coll.RangeFromTo(x1, x2).Backwards()) {
  ... use x ...
}
```

Pattern 53 Iteration from `x1` (inclusive) to end
Let `coll` be an ISorted<T>. Iterate forwards over all items between `x1` (inclusive) and the end of the collection. Does nothing if `x1` is greater than the greatest item in the collection.

```
foreach (T x in coll.RangeFrom(x1)) {
  ... use x ...
}
```

Pattern 54 Iteration from beginning to x2 (exclusive)
Let coll be an ISorted<T>. Iterate forwards over all items between the beginning of the collection and x2 (exclusive). Does nothing if x2 is less than or equal to the least item in the collection.

```
foreach (T x in coll.RangeTo(x2)) {
  ... use x ...
}
```

Pattern 55 Iteration from x1 (inclusive) to x2 (inclusive)
Let coll be an ISorted<T>. If x2 has a successor x2Succ, then iteration to x2 inclusive can be done by iteration to x2Succ exclusive. If x2 has no successor, then iteration to x2 inclusive is just iteration to the end of the collection. The directed enumerable range is determined according to these two cases. The x2Succ is computed using method Successor from pattern 47. For descending-order iteration, simply use range.Backwards() in the foreach statement.

```
T x2Succ;
bool x2HasSucc = Successor(coll, x2, out x2Succ);
IDirectedEnumerable<T> range =
  x2HasSucc ? coll.RangeFromTo(x1, x2Succ) : coll.RangeFrom(x1);
foreach (T x in range) {
  ... use x ...
}
```

Pattern 56 Iteration from beginning to x2 (inclusive)
Let coll be an ISorted<T>. This is similar to the previous pattern, except that iteration starts at the beginning of the collection.

```
T x2Succ;
bool x2HasSucc = Successor(coll, x2, out x2Succ);
IDirectedEnumerable<T> range =
  x2HasSucc ? coll.RangeTo(x2Succ) : coll.RangeAll();
foreach (T x in range) {
  ... use x ...
}
```

Pattern 57 Iteration from x1 (exclusive) to x2 (exclusive)
Let coll be an ISorted<T>. If x1 has no successor, then the iteration is empty, and is equivalent to an iteration over an ArrayList<T> containing no items. The successor of x1, if any, is found by method Successor from pattern 47.

```
T x1Succ;
bool x1HasSucc = Successor(coll, x1, out x1Succ);
IDirectedEnumerable<T> range =
  x1HasSucc ? coll.RangeFromTo(x1Succ, x2) : new ArrayList<T>();
foreach (T x in range) {
  ... use x ...
}
```

Pattern 58 Iteration from `x1` (exclusive) to end
Let `coll` be an ISorted<T>. This is similar to the previous pattern, except that iteration ends at the end of the collection.

```
T x1Succ;
bool x1HasSucc = Successor(coll, x1, out x1Succ);
IDirectedEnumerable<T> range =
  x1HasSucc ? coll.RangeFrom(x1Succ) : new ArrayList<T>();
foreach (T x in range) {
  ... use x ...
}
```

Pattern 59 Iteration from `x1` (exclusive) to `x2` (inclusive)
Let `coll` be an ISorted<T>. This is a combination of patterns 55 and 57.

```
T x1Succ, x2Succ;
bool x1HasSucc = Successor(coll, x1, out x1Succ),
     x2HasSucc = Successor(coll, x2, out x2Succ);
IDirectedEnumerable<T> range =
  x1HasSucc ? (x2HasSucc ? coll.RangeFromTo(x1Succ, x2Succ)
                         : coll.RangeFrom(x1Succ))
            : new ArrayList<T>();
foreach (T x in range) {
  ... use x ...
}
```

9.10 Patterns for indexed iteration

An indexed collection — one whose items are indexed by consecutive integers — can be enumerated in two ways: *forwards* (in ascending index order) and *backwards* (in descending index order). In C5, indexed collections implement IDirectedEnumerable and therefore easily support enumeration in either direction. These patterns show how to iterate forwards or backwards over an indexed collection, while keeping track of the item's index `j`.

In the patterns below, let `list` be an IList<T>, such as ArrayList<T>, HashedArrayList<T>, LinkedList<T>, HashedLinkedList<T>, or WrappedArray<T>.

Pattern 60 Iterate forwards over the indexed list

```
int j = 0;
foreach (T x in list) {
  ... x ... at index j ...
  j++;
}
```

Pattern 61 Iterate backwards over the indexed list

```
int j = list.Count;
foreach (T x in list.Backwards()) {
  j--;
  ... x ... at index j ...
}
```

Pattern 62 Iterate forwards over a sublist, visiting items at indexes `i...i+n-1`

```
int j = i;
foreach (T x in list.View(i, n)) {
  ... x ... at index j ...
  i++;
}
```

Pattern 63 Iterate backwards over a sublist, visiting indexes `i+n-1...i`

```
int j = i+n;
foreach (T x in list.View(i, n).Backwards()) {
  j--;
  ... x ... at index j ...
}
```

Pattern 64 Iterate forwards over a sublist, visiting indexes from `i` to end of list

```
int j = i;
foreach (T x in list.View(i, list.Count-i)) {
  ... x ... at index j ...
  j++;
}
```

Pattern 65 Iterate forwards from beginning of list to index `i` inclusive

```
int j = 0;
foreach (T x in list.View(0, i+1)) {
  ... x ... at index j ...
  j++;
}
```

9.11 Patterns for enumerating a tree snapshot

It is illegal to modify a collection while enumerating it. For instance, execution of

```
foreach (T x in coll) {
  ... use x and modify coll ...
}
```

would throw CollectionModifiedException if `coll` were modified while the `foreach` loop is being executed.

Pattern 66 Enumerating a collection while modifying it
If `tree` is an IPersistentSorted<T>, for instance a TreeSet<T> or TreeBag<T>, one can take a snapshot of the tree and then enumerate the items of that snapshot, while modifying the original tree at the same time:

```
using (ISorted<T> snap = tree.Snapshot()) {
  foreach (T x in snap) {
    ... use x while possibly modifying tree ...
  }
  ...
}
```

The snapshot is allocated in a `using` statement to ensure that its `Dispose` method is called immediately after it is no longer needed. The iteration would be correct also without this safeguard, but the snapshot might be kept alive longer and that would cause a time and space overhead for subsequent updates to the `tree`. See section 12.6.

9.12 Patterns for segment reversal and swapping

Let `list` be an IList<T>, such as ArrayList<T>, HashedArrayList<T>, LinkedList<T>, HashedLinkedList<T>, or WrappedArray<T>.

Pattern 67 Reverse all items in the list

```
list.Reverse()
```

Pattern 68 Reverse the list segment `list[i...i+n-1]`
The list segment `i...i+n-1` is reversed by reversing a list view covering that segment. Throws ArgumentOutOfRangeException if `i < 0` or `n > 0 i+n > list.Count`.

```
public static void ReverseInterval<T>(IList<T> list, int i, int n) {
  list.View(i,n).Reverse();
}
```

Pattern 69 Swap initial segment `list[0..i-1]` with `list[i..Count-1]`
An initial segment `list[0..i-1]` and the corresponding final segment `list[i..n-1]`, where `n` is the length of the list, can be swapped without using any auxiliary space by first reversing the segments separately and then reversing the whole list. This procedure moves each list item twice; using a far more intricate scheme it is possible to use only one move per item, but that is not faster in practice. Throws ArgumentOutOfRangeException if `i < 0` or `i >= Count`.

```
public static void SwapInitialFinal<T>(IList<T> list, int i) {
  list.View(0,i).Reverse();
  list.View(i,list.Count-i).Reverse();
  list.Reverse();
}
```

9.13 Pattern for making a stream of item lumps

Pattern 70 Make a stram of `n`-item lumps
Given an SCG.IEnumerable<T>, create an SCG.IEnumerable<IQueue<T>> that contains all the (possibly overlapping) `n`-item lumps or subsequences of items.

```
public static SCGIEnumerable<IQueue<T>> Lump<T>(SCG.IEnumerable<T> xs, int n) {
  if (n <= 0)
    throw new ArgumentException("must be positive", "n");
  else {
    IQueue<T> res = new CircularQueue<T>();
    int i=0;
    foreach (T x in xs) {
      res.Enqueue(x);
      i++;
      if (i>=n) {
        yield return res;
        res.Dequeue();
      }
    }
  }
}
```

The `Lump` method returns an empty sequence of lumps if `xs.Count` $<$ `n`. The method can be used as follows:

```
int n = ...;
foreach (IQueue<T> lump in Lump<T>(coll, n)) {
  for (int i=0; i<n; i++) {
    ... lump[i] ...
  }
}
```

Note that modifications to a lump returned by method Lump will affect subsequent lumps returned by the method. In fact, the res.Dequeue operation may throw an exception in case the consumer has emptied lump res. To prevent the consumer from modifying the lumps, one can wrap the returned lump in a GuardedQueue<T> when yielding it from method Lump:

```
if (i>=n) {
  yield return new GuardedQueue<T>(res);
  res.Dequeue(); // Succeeds because n>0 and so res nonempty
}
```

A note on efficiency: Getting all lumps from the sequence generated by Lump(xs,n) takes time $O(|\mathrm{xs}|)$ when res is a CircularQueue<T> and the same if res were a LinkedList, but takes time $O(n|xs|)$ if res were an ArrayList. Random indexing into one of the lumps takes time $O(1)$ when res is a CircularQueue<T> and the same if res were an ArrayList, but takes time $O(n|xs|)$ if res were a LinkedList. So CircularQueue<T> is the only choice that offers fast lump creation as well as fast lump indexing.

9.14 Patterns for arbitrary and random items

Let coll be an ICollectionValue<T> for which coll.IsEmpty is false.

Pattern 71 Get an arbitrary item x from a collection coll

```
T x = coll.Choose();
```

Note that x is an arbitrary item, not a randomly chosen one, so the Choose() method cannot be used to generate a random sequence of items from coll. With some collection classes, subsequent calls to Choose() will return the same item x, with others it will not. To choose a random item, use pattern 73.

Pattern 72 Iterate over a collection in an undetermined order
When coll is a C5 collection class, it is guaranteed that an item x newly returned by coll.Choose() can be efficiently removed from coll, so that a sequence of n Choose and Remove operations as implemented below takes time $O(n)$. (This would not be the case if x were chosen at random).

```
while (!coll.IsEmpty) {
  T x = coll.Choose();
  coll.Remove(x);
  ... process x, possibly adding further items to coll ...
}
```

Pattern 73 Get a random item `x` from an indexed collection
Let `coll` be an IIndexed<T>, preferably one that permits fast indexing, such as ArrayList<T>, HashedArrayList<T>, SortedArray<T>, TreeSet<T>, TreeBag<T>, or WrappedArray<T>. Let `rnd` be a random number generator of class System.Random or C5Random. To get a random item from the collection, choose a random index and return the item at that index. To draw multiple random items with or without replacement, see section 9.18.

```
T x = coll[rnd.Next(coll.Count)];
```

9.15 Patterns for set operations on collections

Sets `s1` and `s2` of items of type T can be represented by HashSet<T> or TreeSet<T> objects. *Destructive* set operations, that is, operations that modify one of the sets involved in the operation, are easily implemented using collection operations as shown below. *Functional* or versions of these operations, that is, operations that return a new set instead of modifying any of the given ones, can be implemented using overloaded operators in a subclass of HashSet<T> or TreeSet<T> as shown in section 11.11.

In all cases, the exact same code can be used to implement operations on bags (multisets). Only one must use HashBag<T> and TreeBag<T> instead of HashSet<T> and TreeSet<T> to represent bags.

For set operations using TreeSet<T> or TreeBag<T> it would make sense to check that their Comparers are identical (same object). This check can usually be implemented as `s1.Comparer == s2.Comparer` because the default comparers created by `Comparer<T>.Default` are singletons; see section 2.6.

Pattern 74 Determine whether `s1` is the empty set

```
s1.IsEmpty
```

Pattern 75 Determine whether `s1` is a subset of `s2`

```
s2.ContainsAll(s1);
```

Pattern 76 Compute the intersection of sets `s1` and `s2` in `s1`

```
s1.RetainAll(s2);
```

Pattern 77 Compute the union of sets `s1` and `s2` in `s1`

```
s1.AddAll(s2);
```

Pattern 78 Compute the difference between sets s1 and s2 in s1

```
s1.RemoveAll(s2);
```

Pattern 79 Compute the symmetric difference between sets s1 and s2
The symmetric difference is the set of elements in one set but not in the other; it equals $(\mathtt{s1} \cup \mathtt{s2}) \setminus (\mathtt{s1} \cap \mathtt{s2})$ and equals $(\mathtt{s1} \setminus \mathtt{s2}) \cup (\mathtt{s2} \setminus \mathtt{s1})$. This computes the symmetric difference in a new set su.

```
HashSet<T> su = new HashSet<T>();
su.AddAll(s1);
su.AddAll(s2);
HashSet<T> si = new HashSet<T>();
si.AddAll(s1);
si.RetainAll(s2);
su.RemoveAll(si);
```

For sorted sets the symmetric difference can be computed more efficiently by traversing the ordered sets in synchrony, in a way similar to a list merge.

9.16 Patterns for removing duplicates

Pattern 80 Keep only the first occurrence in an enumerable
This method takes an enumerable and returns an array list with the items in the same order, but keeping only the first occurrence of each item. Pattern 81 shows how to keep only the last occurrence of each item.

```
public static IList<T> RemoveLateDuplicates<T>(SCG.IEnumerable<T> enm) {
  IList<T> res = new HashedArrayList<T>();
  res.AddAll(enm);
  return res;
}
```

Pattern 81 Keep only the last occurrence in a directed enumerable
This method takes a directed enumerable and returns an array list with the items in the same order, but keeping only the last occurrence of each item. Pattern 80 shows how to keep only the first occurrence of each item.

```
public static IList<T> RemoveEarlyDuplicates<T>(IDirectedEnumerable<T> enm) {
  IList<T> res = new HashedArrayList<T>();
  res.AddAll(enm.Backwards());
  res.Reverse();
  return res;
}
```

If the given enumerable enm is not directed, then one can create an ArrayList<T> of its items and obtain a directed enumerable from that.

9.17 Patterns for collections of collections

Here are some patterns for creating a collection whose items are themselves collections. See also section 8.3.

Pattern 82 Create a hash set of lists with default (sequenced) equality
The default equality comparer created for an "outer" collection of "inner" collections is a sequenced collection equality comparer or an unsequenced collection equality comparer, depending on the type of the inner collection; see section 2.3. For lists, the default is a sequenced equality comparer, which takes item order into account.

```
HashSet<IList<int>> hs1 = new HashSet<IList<int>>();
```

Hence the default equality comparer `hs1.EqualityComparer` created for the hash set `hs1` above would consider the two lists `coll1` and `coll2` equal because they have the same items in the same order, whereas `coll3` is not equal to any of the others: although it contains the same items, they do not appear in the same order. The result is that `hs1` contains two items (list objects): one of `coll1` and `coll2`, as well as `coll3`:

```
IList<int> coll1 = new LinkedList<int>(),
           coll2 = new LinkedList<int>(),
           coll3 = new LinkedList<int>();
coll1.AddAll(new int[] { 7, 9, 13 });
coll2.AddAll(new int[] { 7, 9, 13 });
coll3.AddAll(new int[] { 9, 7, 13 });
hs1.Add(coll1); hs1.Add(coll2); hs1.Add(coll3);   // hs1.Count is 2
```

Pattern 83 Explicitly create a hash set of lists with sequenced equality
If desired, one can explicitly create a sequenced collection equality comparer `seqc` (section 2.3) and pass it to the constructor when creating the hash set. In fact, this is exactly what happens implicitly in pattern 82.

```
SCG.IEqualityComparer<IList<int>> seqc
  = new SequencedCollectionEqualityComparer<IList<int>,int>();
HashSet<IList<int>> hs2 = new HashSet<IList<int>>(seqc);
Equalities("Sequenced equality", hs2.EqualityComparer);
hs2.Add(coll1); hs2.Add(coll2); hs2.Add(coll3);
```

To avoid taking list item order into account, one could create an UnsequencedCollectionEqualityComparer<IList<int>,int> object and pass it to the constructor when creating the hash set. But this is a bad idea, as the equality function will be very slow except for short lists; see anti-pattern 122.

Pattern 84 Create a hash set of lists using reference equality
To use reference equality and a corresponding hash function for a collection of collections, one must explicitly create a reference equality comparer (section 2.3) and pass it to the outer collection when it is created.

```
SCG.IEqualityComparer<IList<int>> reqc
  = new ReferenceEqualityComparer<IList<int>>();
HashSet<IList<int>> hs4 = new HashSet<IList<int>>(reqc);
```

The equality comparer `hs4.EqualityComparer` used by the hash set `hs4` created above would consider the three lists `coll1`, `coll2` and `coll3` below unequal: they are distinct objects, created by distinct applications of `new`. The result is that `hs4` contains three items: the three list objects:

```
hs4.Add(coll1); hs4.Add(coll2); hs4.Add(coll3);
```

Pattern 85 Check that all inner collections use the same equality comparer
When using a collection of collections it is crucial that the inner collections all use the same equality comparer. Provided all equality comparers are obtained from `EqualityComparer<T>.Default`, one can just check that the `EqualityComparer` property for all the inner collections are the same object. The two generic type parameters and the type parameter constraint permits this method to work on any collection whose items are extensible collections (and which therefore have the `EqualityComparer` property); it introduces a measure of co-variant subtyping otherwise not available in C#.

```
public static bool HasherSanity<T,U>(ICollectionValue<U> colls)
  where U : IExtensible<T>
{
  SCG.IEqualityComparer<T> hasher = null;
  foreach (IExtensible<T> coll in colls) {
    if (hasher == null)
      hasher = coll.EqualityComparer;
    if (hasher != coll.EqualityComparer)
      return false;
  }
  return true;
}
```

Pattern 86 Adding snapshot of inner collection to an outer collection
An inner collection should never be modified after it has been added to an outer collection. One way to ensure this is to create a copy of the inner collection and wrap it as a guarded collection (to prevent modification of an inner collection retrieved from an outer collection) before adding it to the outer collection. Snapshots of tree sets and tree bags are read-only lightweight copies of the tree set and therefore ideal for this purpose; see sections 4.9 and 8.5.

```
ICollection<ISequenced<int>> outer = new HashSet<ISequenced<int>>();
IPersistentSorted<int> inner1 = new TreeSet<int>();
inner1.AddAll(new int[] { 2, 3, 5, 7, 11 });
// Take a snapshot and add it to outer:
outer.Add(inner1.Snapshot());
// Modify inner1, but not the snapshot contained in outer:
inner1.Add(13);
```

Pattern 87 Create a set of sets of integers
After this piece of code, the set bound to `outer` contains five sets of integers, namely the sets { } and {2} and {2,3} and {2,3,5} and {2,3,5,7}. In cases like this where the inner sets being added are modifications of each other, it is absolutely essential that only snapshots of the collection bound to `inner`, not that collection itself, are added to the outer collection.

```
ICollection<ISequenced<int>> outer = new HashSet<ISequenced<int>>();
int[] ss = { 2, 3, 5, 7 };
TreeSet<int> inner = new TreeSet<int>();
outer.Add(inner.Snapshot());
foreach (int i in ss) {
  inner.Add(i);
  outer.Add(inner.Snapshot());
}
```

Pattern 88 Create a set of bags of characters
It is straightforward to create a set of bags of characters as a hash set of hash bags of characters. As further elaborated in the section 11.4 example, this can be used to find anagrams. Note that a new bag object is created for each word added to the set of bags; hence there is no need to create a copy For defensive programming, one might wrap each hash bag as a GuardedCollection<`char`> before storing it in the set of bags.

```
String text =
  @"three sorted streams aligned by leading masters ... algorithm.";
String[] words = text.Split(' ', '\n', '\r', ';', ',', '.');
ICollection<ICollection<char>> anagrams = new HashSet<ICollection<char>>();
int count = 0;
foreach (String word in words) {
  if (word != "") {
    count++;
    HashBag<char> anagram = new HashBag<char>();
    anagram.AddAll(word.ToCharArray());
    anagrams.Add(anagram);
  }
}
Console.WriteLine("Found {0} anagrams", count - anagrams.Count);
```

9.18 Patterns for creating a random selection

Pattern 89 Create random selection with replacement
Let `coll` be an efficiently indexable indexed collection, such as ArrayList<T>, and let $N \geq 0$ be an integer. Then this code obtains `N` random items from `coll`, with replacement.

```
public static SCG.IEnumerable<T> RandomWith<T>(IIndexed<T> coll, int N) {
  for (int i=N; i>0; i--) {
    T x = coll[rnd.Next(coll.Count)];
    yield return x;
  }
}
```

Pattern 90 Create a (large) random selection without replacement
Let `list` be a list collection, and let `N` where $0 \leq N <$ `list.Count` be an integer. Then `N` random items from `coll` may be selected, without replacement, as shown below. Note that this modifies the given list.

```
public static SCG.IEnumerable<T> RandomWithout1<T>(IList<T> list, int N) {
  list.Shuffle(rnd);
  foreach (T x in list.View(0, N))
    yield return x;
}
```

Pattern 91 Create a (small) random selection without replacement
Pattern 90 is simple and quite efficient when `N` and `list.Count` are of the same order of magnitude. When `N` is much smaller than `list.Count`, the following is faster but requires the list to be an ArrayList<T> so that it supports fast indexing. Note that this too modifies the given list.

```
public static SCG.IEnumerable<T> RandomWithout2<T>(ArrayList<T> list, int N) {
  for (int i=N; i>0; i--) {
    int j = rnd.Next(list.Count);
    T x = list[j], replacement = list.RemoveLast();
    if (j < list.Count)
      list[j] = replacement;
    yield return x;
  }
}
```

9.19 Patterns for sorting

9.19.1 Quicksort for array lists

Let `enm` be an SCG.IEnumerable<T> and `cmp` an SCG.IComparer<T>. An array list containing `enm`'s items sorted according to `cmp` can be obtained as follows, preserving duplicates of items that compare equal:

```
IList<T> result = new ArrayList<T>();
result.AddAll(enm);
result.Sort(cmp);
```

9.19.2 Merge sort for linked lists

Let `enm` be an SCG.IEnumerable<T> and `cmp` an SCG.IComparer<T>. A linked list containing `enm`'s items sorted according to `cmp` can be obtained as follows:

```
IList<T> result = new LinkedList<T>();
result.AddAll(enm);
result.Sort(cmp);
```

This operation preserves duplicates of items that compare equal, and preserves the order of equal items: two items that compare equal will appear in the same order in the result as in `enm`. This takes time $O(n\log n)$, but is likely to be somewhat slower than creating a sorted array list above.

9.19.3 Heap sort using a priority queue

Let `enm` be an SCG.IEnumerable<T>, `cmp` an SCG.IComparer<T> and N an integer. A sequence of the N least items of `enm` (in increasing order) can be created as follows:

```
IPriorityQueue<T> queue = new IntervalHeap<T>(enm);
queue.AddAll(enm);
IList<T> result = new ArrayList<T>();
int k = N;
while (!queue.IsEmpty && k-- > 0)
  result.Add(queue.DeleteMin());
```

Using `queue.DeleteMax()` instead of `queue.DeleteMin()` gives a list (in descending order) of the N greatest items of enm.

When N equals the initial `queue.Count`, this will implement heapsort, but although it will have guaranteed worst-case execution time $O(n\log n)$ it will be slower than the introspective quicksort implemented in C5. When N is much smaller than the initial `queue.Count`, the above procedure may be faster, but it is unlikely.

9.19.4 Sorting arrays

Class Sorting provides several methods for sorting a one-dimensional array or a of segment of one; see section 8.6.

9.19.5 Sorting by bulk insertion into a SortedArray<T>

Let `enm` be an SCG.IEnumerable<T> and `cmp` an SCG.IComparer<T>. A sorted-array list containing `enm`'s items sorted according to `cmp` can be obtained as follows:

```
IList<T> res = new SortedArray<T>(cmp);
res.AddAll(enm);
```

After this operation items are stored in increasing order in `res`, but duplicate items (items that compare equal) have been discarded. The `AddAll` operation on SortedArray<T> is guaranteed efficient: the above operation takes time $O(n \log n)$. In particular, `AddAll()` does not insert the items of `enm` one by one into the sorted array.

9.19.6 Finding a permutation that would sort a list

Sometimes (1) the permutation that would sort a collection is more interesting than the sorted collection itself; or (2) the items in the collection appear in some order that should be preserved, while at the same time the must be accessed in sorted order; or (3) the items to be sorted are very large structs, and it would be good to avoid the copying of items implied by standard sorting algorithms.

Pattern 92 achieves goals (1), (2) and (3) and applies to collections with fast indexing, such as array lists, whereas pattern 93 achieves only goals (1) and (2) but applies also to collections without fast indexing, such as linked lists.

Pattern 92 Finding a permutation that would sort an array list
Given a list `list` of comparable items, this method creates and returns an array list `res` of distinct integer indexes such that the functional composition of `list` and `res` is sorted. In other words, whenever `i<=j` it holds that `list[res[i]]<=list[res[j]]`.

It works by sorting the array list `res` of indexes using a comparer that compares integers `i` and `j` by comparing the items `list[i]` and `list[j]` in the given list. That is, it rearranges the indexes in `res` but does not move the items in `list`, which may actually be a read-only list.

This method is fast when `list` is an array list and similar, but not stable. It is slow for linked lists because it performs random access to list items by index.

```
public static ArrayList<int> GetPermutation1<T>(IList<T> list)
  where T : IComparable<T>
{
  ArrayList<int> res = new ArrayList<int>(list.Count);
  for (int i = 0; i < list.Count; i++)
    res.Add(i);
  res.Sort(new DelegateComparer<int>
           (delegate(int i, int j) { return list[i].CompareTo(list[j]); }));
  return res;
}
```

Pattern 93 Find the permutation that would stably sort a linked list
Like the preceding pattern 92 this method creates and returns an array list `res` of distinct integer indexes such that the functional composition of `list` and `res` is sorted. In other words, whenever `i<=j` it holds that `list[res[i]]<=list[res[j]]`.

It works by creating a new linked list of pairs, each pair consisting of an item from the given list and the index of that item. Then it sorts the list of pairs, based on comparison of the items only. Finally it extracts the (now reordered) list of indexes `res`. Note that the given list may be read-only, as only the items of the newly constructed list of pairs are reordered.

This method performs a stable sort (which means that the indexes of equal items will come in increasing order in the result) and is fairly fast both for array lists and linked lists, but in contrast to pattern 92 it does copy the given list's items.

```
public static ArrayList<int> GetPermutation2<T>(IList<T> list)
  where T : IComparable<T>
{
  int i = 0;
  IList<KeyValuePair<T,int>> zipList =
    list.Map<KeyValuePair<T,int>>
        (delegate(T x) { return new KeyValuePair<T,int>(x, i++); });
  zipList.Sort(new KeyValueComparer<T>(list));
  ArrayList<int> res = new ArrayList<int>(list.Count);
  foreach (KeyValuePair<T, int> p in zipList)
    res.Add(p.value);
  return res;
}

private class KeyValueComparer<T> : SCG.IComparer<KeyValuePair<T,int>>
  where T : IComparable<T>
{
  private readonly IList<T> list;
  public KeyValueComparer(IList<T> list) {
    this.list = list;
  }
  public int Compare(KeyValuePair<T,int> p1, KeyValuePair<T,int> p2) {
    return p1.key.CompareTo(p2.key);
  }
}
```

These methods to find the permutation that would sort a list can be used also to efficiently find the quantile rank of all items in a list; see pattern 106.

9.20 Patterns using priority queue handles

A priority queue is a collection that permits efficient retrieval of a minimal or maximal item from the collection. In addition, C5's priority queue implementation permits retrieval, deletion and updating of items using a so-called *handle*, which is a unique identification of the item. The comparer of the items cannot be used for this purpose because the priority queue may contain multiple items that have the same priority, so they compare equal by the comparer used in the priority queue.

These patterns show some typical ways to use priority queue handles. Let `pq` be an IPriorityQueue<T>, such as IntervalHeap<T>, and let `h` be an IPriorityQueueHandle<T>.

Pattern 94 Add item `x` into a priority queue, getting a new handle for it in `h`

```
h = null;
pq.Add(ref h, x)
```

Pattern 95 Add item `x` into a priority queue, reusing handle `h` for it

```
pq.Add(ref h, x)
```

Pattern 96 Check whether the item with handle `h` is (still) in the priority queue, and if so, get it in variable `x`

```
bool isInPq = pq.Find(h, out x)
```

Pattern 97 Get the item `x` with handle `h`

```
T x = pq[h]
```

Pattern 98 Get and delete the item `x` with handle `h`

```
T x = pq.Delete(h)
```

Pattern 99 Replace the item having handle `h` with `x`
This can be done in two ways:

```
pq[h] = x                              pq.Replace(h, x)
```

Pattern 100 Get the current least item into `min` and its handle into `h`
To get the greatest item, use `FindMax` instead.

```
T min = pq.FindMin(out h)
```

Pattern 101 Get and delete the current least item and its handle

```
T min = pq.DeleteMin(out h)
```

To get and delete the greatest item, use `DeleteMax` instead.

Typically, P is the type of some items that may have associated updatable priorities, times, weights or similar, and P is a reference type. Then one may define a struct type T that holds a P reference and the priority, so that T implements IComparable<T> by comparing P item priorities. If further one declares operators

```
public static T operator +(T t, int delta)
public static T operator -(T t, int delta)
```

in T, then one can change the priority of a given T item with handle h using statements such as these:

```
pq[h] += 17
pq[h] -= 17
```

They are equivalent to the following more explicit statements:

```
pq.Replace(h, pq[h] + 17)
pq.Replace(h, pq[h] - 17)
```

9.21 Patterns for finding quantiles

Assume we have a non-empty ordered data set B of observations, possibly with duplicates (so B is a bag), from some stochastic process in nature. For instance, we may have observed the following $10 = |B|$ grades awarded to students in a class:

$$0, 3, 5, 6, 6, 7, 8, 8, 9, 11$$

This example uses the Danish grading scale which has the grades 00, 03, 4, 5, 6, 7, 8, 9, 10, 11, 13, with 5 and below being failed, 8 being the average grade, and 13 exceptional. Such a set of observations can be represented by a TreeBag<`int`> which implements IIndexedSorted<`int`>.

The *quantile rank* of x in such a data set is defined as the ratio p of observations that are less than or equal to x. For instance, the quantile rank of 5 is 30%, the quantile rank of 6 is 50%, the quantile rank of 8 is 80%, and the quantile rank of 13 is 100%. Conversely, the p-quantile is the least observation $x \in B$ whose quantile rank is at least p. In other words, the p-quantile is the first observation in the ordered data set if $p = 0$ or and it is the observation at position $\lceil p \cdot |B| \rceil$, counting from 1, when $p > 0$.

The first, second and third *quartile* are the 25%-quantile, the 50%-quantile and the 75%-quantile. The *median* is the 50%-quantile. In the example above, the first quartile is 5, the median is 6, and the third quartile is 8.

In the following patterns, let `obs` be a non-empty IIndexedSorted<T>, such as a TreeBag<T>, representing a collection of observations of some ordered type T.

Pattern 102 Quantile rank in observations with duplicates
This computes the quantile rank as a number between 0 and 1 (inclusive). The quantile rank is also called the empirical cumulative distribution function.

```
double rank = (obs.FindMax().CompareTo(x) <= 0) ? 1
              : (obs.CountTo(obs.Successor(x)))/(double)(obs.Count);
```

Pattern 103 The observation at a given quantile
Given a floating-point number p between 0 and 1 (inclusive), this finds the p-quantile.

```
double quantile = p==0 ? B[0] : B[(int)(Math.ceiling(p*B.Count))-1];
```

The above works whether or not duplicate observations are allowed.

If there are no duplicates in the data set *B*, then it can be represented as a sorted array, and `IndexOf(x)`, which performs a binary search, can be used to find the rank. This does not work when duplicates are allowed, as the search will just find one of the duplicate items, not necessarily the last one.

Pattern 104 Percentile rank in observations with no duplicates
This computes the quantile rank as a number between 0 and 1 (inclusive) when the data set has no duplicates and is stored in a non-empty sorted array `sarr`. When `x` is not `sarr`, then `sarr.IndexOf(x)` returns the one's complement `j` of the index `i` at which `x` belongs. This is position 0 if all items in `sarr` are greater than `x`, or else the least position whose left neighbor is smaller than `x`.

```
int j = sarr.IndexOf(x);
double rank = (j >= 0 ? j : ~j)/(double)(obs.Count);
```

Pattern 105 The observation at a given quantile
Given a floating-point number p between 0 and 1 (inclusive), this finds the p-quantile in sorted array `sarr`:

```
double quantile = p==0 ? sarr[0] : sarr[(int)(Math.ceiling(p*sarr.Count))-1];
```

Pattern 106 Finding the rank of every item in a list x
Given a list of ordered observations, one can find the rank within the data set of each item by first finding a permutation that would sort the list, and then finding the permutation that would sort that permutation. More precisely, let the resulting array list be `res`; then the item `list[i]` at position `i` in the given list ranks as number `res[i]` among all the list items (counting from 0). Equal items will be assigned consecutive ranks in the order in which they appear in the `list`. The methods `GetPermutation1` and `GetPermutation2` are from patterns 92 and 93.

```
ArrayList<int> res = MySort.GetPermutation1(MySort.GetPermutation2(list));
foreach (int i in res)
  Console.Write("{0} ", i);
```

9.22 Patterns for stacks and queues

The classes ArrayList<T>, CircularQueue<T> and LinkedList<T> implement interface IList<T> and hence the methods `Add` and `Remove`. The `Add` method always adds the new item as the last one in the a sequenced collection.

On ArrayList<T>, the `Remove` method removes the last item, so the pair `Add`/`Remove` behaves like `Push`/`Pop` on a stack, and the FIFO property of an ArrayList<T> is false by default. In fact, ArrayList<T> implements IStack<T> and hence `Push` and `Pop` methods that make it behave like a stack. Although ArrayList<T> implements also IQueue<T>, its `Dequeue` operation is inefficient so ArrayList<T> should not be used to implement a queue; use CircularQueue<T> or LinkedList<T> instead.

On LinkedList<T>, the `Remove` method removes the first item from the sequenced collection, so the pair `Add`/`Remove` behaves like `Enqueue`/`Dequeue` on a queue: the FIFO property of a LinkedList<T> is true by default. In fact, LinkedList<T> implements IQueue<T> and hence `Enqueue` and `Dequeue` methods that make it behave like a queue.

Class CircularQueue<T> implements a queue (first in, first out) using an array as a circular buffer, so in addition to efficient queue operations it permits efficient indexing relative to the queue's first item (which is its oldest item).

The characteristics of the three classes ArrayList<T>, LinkedList<T> and CircularQueue<T> as stacks and queues are summarized in figure 9.2.

	IExtensible<T>	IList<T>		IStack<T>		IQueue<T>	
Class	`Add`	`Remove`	`this[i]`	`Push`	`Pop`	`Enqueue`	`Dequeue`
ArrayList<T>	last	last	$O(1)$	last	last	last	first†
LinkedList<T>	last	first	$O(i)$†	last	last	last	first
CircularQueue<T>	–	–	$O(1)$	–	–	last	first

Figure 9.2: Stack and queue operations implemented by the classes ArrayList<T>, LinkedList<T> and CircularQueue<T>. Avoid operations marked (†); they are slow.

In practice, stack operations on ArrayList<T> and queue operations on CircularQueue<T> are probably somewhat faster than on LinkedList<T>. Hence these recommendations:

- To implement a pure stack, use ArrayList<T> as in pattern 108.
- To implement a pure queue, use CircularQueue<T> as in pattern 107.
- If you need to sometimes pass a stack and sometimes a queue to a method, then declare the method to accept an IList<T> and then pass either an ArrayList<T> or a LinkedList<T>, as shown in pattern 109.

9.22.1 Breadth-first and depth-first traversal

There are two main orders in which to traverse the nodes of a tree or graph: breadth-first and depth-first. In the former, all nodes near the root are traversed before nodes far from the root. In the latter, complete paths from root to leaves are visited before visiting other nodes near the root of the tree. Using the queue, stack and list interfaces one can write clear, efficient and flexible tree traversal code. For graph traversal one in addition needs to keep track of the which nodes have been visited already; a hashset of nodes references (with reference equality) may be used for that purpose.

Binary trees with node contents of type T can be represented by class Tree<T> like this:

```
class Tree<T> {
  private T val;
  private Tree<T> t1, t2;
  public Tree(T val) : this(val, null, null) { }
  public Tree(T val, Tree<T> t1, Tree<T> t2) {
    this.val = val; this.t1 = t1; this.t2 = t2;
  }
}
```

The expression `new Tree<T>(x)` produces a leaf node that carries value `x`; the expression `new Tree<T>(x, t1, t2)`, where `t1` or `t2` is non-`null`, produces an internal that carries value `x` and has subtrees `t1` and `t2`.

Pattern 107 Breadth-first traversal of a binary tree
This method uses a queue (a first-in-first-out data structure) to hold those tree nodes that still need to be visited. A node removed from the queue will be the oldest one on the queue, that is, the one closest to the tree root. This gives breadth-first traversal. The queue implementation used here is a circular queue. A linked list would do fine as well, but might be slightly slower in practice.

At each tree node, the function `act` is applied to the value `val` in the node.

```
public static void BreadthFirst(Tree<T> t, Act<T> act) {
  IQueue<Tree<T>> work = new CircularQueue<Tree<T>>();
  work.EnQueue(t);
  while (!work.IsEmpty) {
    Tree<T> cur = work.DeQueue();
    if (cur != null) {
      work.EnQueue(cur.t1);
      work.EnQueue(cur.t2);
      act(cur.val);
    }
  }
}
```

Pattern 108 Depth-first traversal of a binary tree
This method uses a stack (a last-in-first-out data structure) to hold those tree nodes that still need to be visited. A node removed from the stack will be newest one in the stack, that is, the one furthest from the tree root. This gives depth-first traversal. Note that to obtain left-right traversal order, the right subtree must be pushed before the the left subtree. The stack implementation is an array list. A linked list whose `FIFO` property had been set to false could be used instead, but would be slightly slower.

```
public static void DepthFirst(Tree<T> t, Act<T> act) {
  IStack<Tree<T>> work = new ArrayList<Tree<T>>();
  work.Push(t);
  while (!work.IsEmpty) {
    Tree<T> cur = work.Pop();
    if (cur != null) {
      work.Push(cur.t2);
      work.Push(cur.t1);
      act(cur.val);
    }
  }
}
```

Pattern 109 Parametrized depth-first/breadth-first traversal of a binary tree
This method uses a list `work` to hold those tree nodes that still need to be visited. If `work` is a linked list (whose `FIFO` property is true), then the method performs a breadth-first traversal. If `work` is an array list (whose `FIFO` property is false), then the method performs a depth-first traversal. Hence this method can do the work of those shown in the previous two patterns.

```
public static void Traverse(Tree<T> t, Act<T> act, IList<Tree<T>> work) {
  work.Clear();
  work.Add(t);
  while (!work.IsEmpty) {
    Tree<T> cur = work.Remove();
    if (cur != null) {
      if (work.FIFO) {
        work.Add(cur.t1);
        work.Add(cur.t2);
      } else {
        work.Add(cur.t2);
        work.Add(cur.t1);
      }
      act(cur.val);
    }
  }
}
```

9.22.2 Double-ended queues

For a double-ended queue or dequeue — one in which insertions and removal can be done efficiently at either end — use a LinkedList. In particular, use the methods InsertFirst(x), InsertLast(x) for inserting items into the dequeue, the methods RemoveFirst() and RemoveLast() for extracting items from the dequeue, and the properties First and Last to inspect the items at either end.

Assume `deq` is LinkedList<T>, then:

Operation	Method call
enqueue at end	`deq.InsertLast(x)`
dequeue from beginning	`deq.RemoveFirst()`
enqueue at beginning	`deq.InsertLast(x)`
dequeue from end	`deq.RemoveLast()`
look at beginning	`deq.First`
look at end	`deq.Last`

By LinkedList<T> default, `InsertLast(x)` is the same as `Add(x)` and `RemoveFirst()` is the same as `Remove()`.

9.23 Patterns for collection change events

Recall that in C# terminology, an *event* on a class is a field of delegate type. An event handler `h`, which is a delegate, can be attached to an event `e` on collection `coll` by executing `coll.e += h` and it can be detached again by executing `coll.e -= h`.

The event and handler mechanisms of C5 are described on section 8.8. Here we show some standard usage patterns for events and event handlers. Note that some modifications to a collection causes several events; for instance `coll.InsertFirst(x)` will raise three events: ItemInserted, ItemsAdded, and CollectionChanged.

Pattern 110 Observing when a collection has been changed
The collection object `c` affected is passed to the event handler, but not used here.

```
coll.CollectionChanged
  += delegate(Object c)  {
       Console.WriteLine("Collection changed");
     };
```

Pattern 111 Observing when a collection has been cleared

```
coll.CollectionCleared
  += delegate(Object c, ClearedEventArgs args) {
       Console.WriteLine("Collection cleared");
     };
```

Pattern 112 Observing when an item has been added to a collection
The event argument also contains the multiplicity `args.Count`, which is 1 if the collection has set semantics but may be greater if it has bag semantics; see pattern 113.

```
coll.ItemsAdded
  += delegate(Object c, ItemCountEventArgs<int> args) {
       Console.WriteLine("Item {0} added", args.Item);
     };
```

Pattern 113 Observing the number of items being added to a collection
A call to `AddAll` or similar methods results in a sequence of ItemsAdded events followed by a CollectionChanged event (if anything was added). Note that a CollectionChanged event is raised also after remove operations; hence the event handler attached to CollectionChanged should test whether any ItemsAdded event preceded the CollectionChanged event.

```
private static void AddItemsAddedCounter<T>(ICollection<T> coll) {
  int addedCount = 0;
  coll.ItemsAdded
    += delegate(Object c, ItemCountEventArgs<T> args) {
         addedCount += args.Count;
       };
  coll.CollectionChanged
    += delegate(Object c) {
         if (addedCount > 0)
           Console.WriteLine("{0} items were added", addedCount);
         addedCount = 0;
       };
}
```

A call to `AddAll` that actually adds no items will not cause a CollectionChanged event. Note that there will be an instance of the `addedCount` local variable for each call to the `AddItemsAddedCounter` method, so the method can be used to attach a counting handler for any number of collections.

Pattern 114 Observing the number of items being removed from a collection
This is the removal count pattern corresponding to the add count in pattern 113.

```
private static void AddItemsRemovedCounter<T>(ICollection<T> coll) {
  int removedCount = 0;
  coll.ItemsRemoved
    += delegate(Object c, ItemCountEventArgs<T> args) {
         removedCount += args.Count;
       };
  coll.CollectionChanged
    += delegate(Object c) {
         if (removedCount > 0)
           Console.WriteLine("{0} items were removed", removedCount);
         removedCount = 0;
       };
}
```

Pattern 115 Observing when an item has been added to an indexed collection

```
coll.ItemInserted
  += delegate(Object c, ItemAtEventArgs<int> args)  {
       Console.WriteLine("Item {0} inserted at {1}", args.Item, args.Index);
     };
```

Pattern 116 Observing when an item has been removed from an indexed collection

```
coll.ItemRemovedAt
  += delegate(Object c, ItemAtEventArgs<int> args)  {
       Console.WriteLine("Item {0} removed at {1}",
                         args.Item, args.Index);
     };
```

Pattern 117 Observing when items (with multiplicity) have been added to a bag

```
bag.ItemsAdded
  += delegate(Object c, ItemCountEventArgs<int> args) {
       Console.WriteLine("{0} copies of {1} added", args.Count, args.Item);
     };
```

Pattern 118 Observing when items (with multiplicity) have been removed

```
bag.ItemsRemoved
  += delegate(Object c, ItemCountEventArgs<int> args) {
       Console.WriteLine("{0} copies of {1} removed",
                         args.Count, args.Item);
     };
```

Pattern 119 Observing when a collection has been updated
There is no specific event associated with an item update, as performed for instance by the methods `Update` (see page 44) and `UpdateOrAdd` (see page 45), or by indexers `coll[i] = x`. Hence an update must be recognized as an ItemsRemoved event followed by an ItemsAdded event followed by a CollectionChanged event, without other intervening CollectionChanged events. This can be done using a small automaton with three states as shown below. The CollectionChanged event is raised at the end of every change precisely to enable this kind of composite-event discovery.

```
private static void AddItemUpdatedHandler<T>(ICollection<T> coll) {
  State state = State.Before;
  T removed = default(T), added = default(T);
  coll.ItemsRemoved
    += delegate(Object c, ItemCountEventArgs<T> args) {
         if (state==State.Before) {
           state = State.Removed;
           removed = args.Item;
         } else
           state = State.Before;
       };
  coll.ItemsAdded
    += delegate(Object c, ItemCountEventArgs<T> args) {
         if (state==State.Removed) {
           state = State.Updated;
           added = args.Item;
         } else
           state = State.Before;
       };
  coll.CollectionChanged
    += delegate(Object c)  {
         if (state==State.Updated)
           Console.WriteLine("Item {0} was updated to {1}",
                             removed, added);
         state = State.Before;
       };
}
```

Chapter 10

Anti-patterns in C5

10.1 Efficiency anti-patterns

The C5 library supports efficient implementation of a large number of common operations in advanced software. Unfortunately it is also possible to use the library in ways that are far from optimal.

This chapter lists some common performance mistakes: solutions that apparently work for small data sets but are far less efficient for large data sets than they could be. We call these common mistakes *performance anti-patterns*: what not to do if you want to write fast programs.

The chapter also mentions a single *correctness anti-pattern*, namely the updating of an inner collection (or other item) after it has been added to an outer collection.

Anti-pattern 120 Many insertions into or deletions from a sorted array
It is very inefficient to insert many items into a sorted array: at each insertion, all items at higher indexes must be moved. Hence n random insertions will take time $O(n^2)$.

```
SortedArray<double> sarr = new SortedArray<double>();
foreach (double d in randomDoubles)
  sarr.Add(d);
```

Instead, use a sorted tree (TreeSet<T>, section 6.8) which like SortedArray<T> implements interface IIndexedSorted<T>; then n random insertions will take only time $O(n \log n)$.

Anti-pattern 121 Using bulk operations `RemoveAll` and `RetainAll` on lists
The bulk operations `RemoveAll` and `RetainAll` are inefficient on non-hashed array lists and non-hashed linked lists; to remove m items from an n-item list takes time $O(mn)$.

```
ArrayList<double> list1 = ..., list2 = ...;
list1.RemoveAll(list2);
```

If item order need not be preserved, use hash sets or hash bags instead of lists. If item order must be preserved and all items have multiplicity at most one (so the collection has set semantics), use hashed linked lists instead on non-hashed lists.

Anti-pattern 122 Using an unsequenced collection equality comparer on lists
An unsequenced collection equality comparer determines equality (whether two collections contain the same items) by disregarding the order in which the items appear. Using an unsequenced equality comparer on a non-hashed array list or non-hashed linked list is slow because there is no efficient way to check whether a particular item is in a list. So performing unsequenced comparison of two lists of length n will take $O(n^2)$.

```
SCG.IEqualityComparer<IList<int>> unseqEqualityComparer
  = new UnsequencedCollectionEqualityComparer<IList<int>,int>();
HashSet<IList<int>> hs3 = new HashSet<IList<int>>(unseqEqualityComparer);
hs3.Add(coll1); hs3.Add(coll2); hs3.Add(coll3);
```

If item order need not be preserved, use hash sets or hash bags instead of lists. If item order must be preserved but all items have multiplicity at most one (so the collection has set semantics), use hashed linked lists instead on non-hashed lists. If item order must be preserved and items may have multiplicity higher than one (so the collection has bag semantics), create hash bags from the lists and perform unsequenced comparison on them. Any of these approaches should reduce the execution time to $O(n)$.

Anti-pattern 123 Using a loop that indexes into a linked list

```
for (int i=0; i<list.Count; i++)
  ... list[i] ...
```

Instead, if you need to update at position `i` but need not insert into or delete from the list, use an ArrayList<T> which has efficient indexing:

```
int i=0;
foreach (T x in list) {
  ... x ...
  i++;
}
```

Or, if you need not update at position `i`, use `foreach` with an explicit counter `i`:

```
int i=0;
foreach (T x in list) {
  ... x ...
  i++;
}
```

Or, if you need to update at position `i` and need to insert into or delete from the list, use a linked list and a view (as in pattern 30):

```
IList<T> view = list.View(0,0);
while (view.TrySlide(0, 1))) {
  ... view[0] ...
  view.Slide(+1, 0);
}
```

Anti-pattern 124 Using `IndexOf` repeatedly on array list or linked list
If `list` is an ArrayList<T> or LinkedList<T> or WrappedArray<T>, then `list.IndexOf(x)` and `list.ViewOf(x)` and the corresponding `Last`-prefixed methods must perform a linear search of the lists. Hence this is likely to be very slow unless `list` is short:

```
// list is a LinkedList<T> or ArrayList<T>
foreach (T x in ...) {
  int i = list.IndexOf(x);
  ... i ...
}
```

Instead, if the list has set semantics, use a hashed array list or hashed linked list, which gives constant-time item lookup:

```
// list is a HashedLinkedList<T> or ArrayList<T>
foreach (T x in ...) {
  int i = list.IndexOf(x);
  ... i ...
}
```

Anti-pattern 125 Using `IndexOf` and then indexing into hashed linked list
If `list` is a hashed linked list, then `list.IndexOf(x)` is fast, but indexing remains slow. Hence this way of updating items equal to `x` is likely to be very slow:

```
// list is a HashedLinkedList<T>
foreach (T x in ...) {
  int i = list.IndexOf(x);
  if (i >= 0) {
    T y = list[i];
    list[i] = ... y ...;
  }
  ...
}
```

Instead use `list.ViewOf(x)` to get a one-item view, and work on the item through that view, for constant-time indexing:

```
// list is a HashedLinkedList<T>
foreach (T x in ...) {
  using (IList<T> view = list.ViewOf(x)) {
    if (view != null) {
      T y = view[0];
      view[0] = ... y ...;
    }
  }
  ...
}
```

Or, if you do not need to insert or delete items, use a hashed array list and `IndexOf`, instead of a hashed linked list and `ViewOf`, to avoid creating the view objects.

Anti-pattern 126 Using an array list as a FIFO queue
An array list `list` can be used as a FIFO queue by enqueueing (adding) items at the end of the list and dequeueing (removing) items from the beginning of the list. But the latter operation is slow if there are more than a few items in the array list, so this can be very slow:

```
// list is an ArrayList<T>
while (!list.IsEmpty) {
  T next = list.Dequeue();
  ... list.Enqueue(...) ...
}
```

Instead use a CircularQueue (or a linked list, see section 9.22) which has constant-time enqueueing and dequeueing operations:

```
// queue is a CircularQueue<T> or LinkedList<T>
while (!list.IsEmpty) {
  T next = queue.Dequeue();
  ... queue.Enqueue(...) ...
}
```

Anti-pattern 127 Using a sorted array as priority queue
A sorted array `sarr` of type `SortedArray<T>` can be used as a priority queue by enqueueing items using `Add` and by retrieving minimal or maximal items using `sarr.DeleteMin()` or `sarr.DeleteMax()`. But insertion into and deletion from a sorted array are slow (linear time) if there are more than a few items in the array, so this can be very slow:

```
// sarr is a SortedArray<T>
while (!sarr.IsEmpty) {
  T min = sarr.DeleteMin();
  ... sarr.Add(...) ...
}
```

Instead, use an IntervalHeap<T> in which insertion and deletion are fast (logarithmic time) operations. Moreover, in contrast to a sorted array, an interval heap can contain multiple occurrences of items that have the same priority (that is, are equal by the item comparer):

```
// pq is an IntervalHeap<T>
while (!pq.IsEmpty) {
  T min = sarr.DeleteMin();
  ... pq.Add(...) ...
}
```

Anti-pattern 128 Using lists to represent sets or bags
An array list or linked list may be used to represent a set or bag (multiset) of items. This may be tempting if you have seen examples in Lisp or Scheme or another functional language, but should be avoided except for very small sets. So these implementations of set operations may be very slow, except for small sets `s0`, `s1`, `s2` and `s3`:

```
// s0, s1, s2, s3 are LinkedList<T> or ArrayList<T>
s1.AddAll(s0);              // s1 = s1 union s0
s2.RemoveAll(s0);           // s2 = s2 minus s0
s3.RetainAll(s0);           // s3 = s3 intersect s0
```

Instead use hash sets, hash bags, tree sets or tree bags, as shown by the patterns in section 9.15 and by the example in section 11.11; or use hashed array lists or hashed linked lists (for sets) if you need to maintain item insertion order.

Anti-pattern 129 Using a bad hash function
This is an artificial example of a bad hash function GetHashCode. It maps all items (which themselves happen to be integers) to just seven different integers: 0, 1, 2, 3, 4, 5 and 6.

```
private class BadIntegerEqualityComparer : SCG.IEqualityComparer<int> {
  public bool Equals(int i1, int i2) { return i1 == i2; }
  public int GetHashCode(int i) { return i % 7; }       // BAD
}
```

Here are some statistics from inserting 50000 random items into a hashset using the bad hash function. The time per inserted item is 104 microseconds. The bucket cost distribution (see section 6.10) shows that there are 7 buckets — one for each of the hash codes 0, 1, 2, 3, 4, 5 and 6 — with a very high cost, and 131065 buckets that are empty. The average non-empty bucket cost is 6968.

```
Bad hash function:  (5.1974736 sec, 48779 items)
 131065 bucket(s) with cost     0
      1 bucket(s) with cost  6808
      1 bucket(s) with cost  6891
      1 bucket(s) with cost  6903
      1 bucket(s) with cost  6950
      1 bucket(s) with cost  7027
      1 bucket(s) with cost  7068
      1 bucket(s) with cost  7132
```

Instead of a bad one, one should of course use a good hash function: one that maps items to hash codes that are evenly distributed over all int values. For items that are integers, that is easy:

```
private class GoodIntegerEqualityComparer : SCG.IEqualityComparer<int> {
  public bool Equals(int i1, int i2) { return i1 == i2; }
  public int GetHashCode(int i) { return i; }
}
```

The statistics from inserting 50000 items into a hash set using the good hash function show that the time per insertion is insertion is 1.0 microsecond, or 100 times better than with the bad hash function. The average non-empty bucket cost is 1.17, so the average lookup time is very low. Moreover, no bucket has cost higher than 5, so the worst-case lookup time is low too.

```
Good hash function: (0.050072 sec, 48771 items)
  89515 bucket(s) with cost     0
  35003 bucket(s) with cost     1
   5938 bucket(s) with cost     2
    573 bucket(s) with cost     3
     42 bucket(s) with cost     4
      1 bucket(s) with cost     5
```

10.2 Correctness anti-patterns

Anti-pattern 130 Modifying a collection that is an item in another collection
As shown by the examples in section 11.4 and 11.5, it is often useful to work with collections of collections. It is easy to get this wrong by creating an inner collection, adding it to the outer collection and then modifying the inner collection. Most likely this will cause it to be "lost" from the outer collection and therefore may produce strange results. What is worse, code like this may accidentally work when the outer collection contains only a few items, then fail as more items are added:

```
ICollection<ISequenced<int>> outer = new HashSet<ISequenced<int>>();
ISequenced<int>
  inner1 = new TreeSet<int>(),
  inner2 = new TreeSet<int>(),
  inner3 = new TreeSet<int>();
inner1.AddAll(new int[] { 2, 3, 5, 7, 11 });
inner2.AddAll(inner1); inner2.Add(13);
inner3.AddAll(inner1);
outer.Add(inner1);
// Now inner1 equals inner3 but not inner2
// and outer contains inner1 and inner3 but not inner2
inner1.Add(13);                      // BAD IDEA
// Now inner1 equals inner2 but not inner3
// and we would expect inner1 or inner3 to be in outer
// but outer apparently contains none of inner1, inner2 and inner3!
```

It is immaterial that the inner collection is a tree set; the same problems appear when it is a hash set (or a bag). The morale is: Never modify an inner collection after it has been added to an outer collection. One way to guard against such modification is to create a read-only copy of the inner collection and then adding that copy to the outer collection. Snapshots of tree sets and tree bags are ideal for this purpose; see sections 4.9 and 8.5 and pattern 86. It is *not* sufficient to wrap the inner collection as a guarded collection and then adding it; it could still be modified via the original reference.

Chapter 11

Application examples

Here we present a number of larger applications of the C5 library, each with some background explanation, code fragments, and discussion. The examples are:

- Recognition of keywords. This example uses a hashset of strings.
- Building a concordance: a sorted listing of words and the line numbers on which they occur. This example uses a tree dictionary and tree-based sets.
- Computing the convex hull of a set of points in the plane. This example uses linked-lists and list views.
- Finding anagram classes in a set of words. This example uses a dictionary from strings to bags of characters.
- Constructing a deterministic finite automaton from a regular expression. This example uses, among other things: sets of sets of integers, and a dictionary from sets of integers to a dictionary from strings to sets of integers.
- Topological sort. This example uses hashed linked lists and list views.
- Graph copying. This example hash dictionaries with custom equality comparers, and stacks.
- A library of general graph algorithms. This example uses hash sets, linked lists, priority queues, and priority queue handles.
- Efficient point location in the plane. This example uses tree sets, tree set snapshots, and tree-based dictionaries.
- A batch job queue simulation. This example uses priority queues, priority queue handles, and hash dictionaries.
- A hash-based implementation of functional set operations.
- A multidictionary, in which a key can be associated with multiple values.
- Finding the *k* most common words in a file.

11.1 Recognizing keywords

In many applications one needs to decide whether a given word (string) is a member of a fixed collection of words. For instance, one wants to ignore frequent and therefore insignificant words (so-called stop words) when automatically indexing text, and one wants to ignore keywords (such as `if`, `while`) when building a cross-reference of identifiers in a program text. In this example we consider the fixed set of 77 keywords from C#:

```
abstract as base bool break byte case catch char checked class const
continue decimal default delegate do double else enum event explicit
extern false finally fixed float for foreach goto if implicit in int
interface internal is lock long namespace new null object operator out
override params private protected public readonly ref return sbyte sealed
short sizeof stackalloc static string struct switch this throw true try
typeof uint ulong unchecked unsafe ushort using virtual void volatile while
```

Such word recognition can be performed efficiently using at least three different kinds of collections: Hash sets, tree sets, and sorted arrays. This example is in file `KeywordRecognition.cs`.

To use a hash set, build it once and for all, then use it in a `bool` method. For instance, it may be bound to a static read-only field `kw1`, initialized in a static constructor, and then used in a method `IsKeyword1`:

```
class KeywordRecognition {
  static readonly String[] keywordArray = { "abstract", ..., "while" };
  private static readonly ICollection<String> kw1;
  static KeywordRecognition() {
    kw1 = new HashSet<String>();
    kw1.AddAll(keywordArray);
  }
  public static bool IsKeyword1(String s) {
    return kw1.Contains(s);
  }
}
```

In this code HashSet<String> could be replaced with TreeSet<String> or SortedArray<String>. By far the fastest solution is to use a hash set. To a large extent that is because the string comparison is quite slow (due to locales or cultures) so the string comparer used by a tree set or sorted array is much slower than the string hash function and equality predicate used by the hash set.

11.2 Building a concordance for a text file

This example builds and prints a concordance from a text file: for every word in the file it finds the line numbers on which the word occurs. The example is in file `Fileindex.cs`. We can use a dictionary to map each word to a set of the numbers of lines on which it occurs; a set avoids duplicate line numbers for each word. The dictionary is a tree dictionary to make sure the words come out sorted when the dictionary is printed, and the set of line numbers is a tree set to make sure the line numbers for each word are sorted. For each word the line numbers are represented by a TreeSet<int>, and hence the entire concordance is represented by a TreeDictionary<String, TreeSet<int>>.

Method `IndexFile` builds the concordance for a given file name:

```
static IDictionary<String, TreeSet<int>> IndexFile(String filename) {
  IDictionary<String, TreeSet<int>> index
    = new TreeDictionary<String, TreeSet<int>>();
  Regex delim = new Regex("[^a-zA-Z0-9]+");
  using (TextReader rd = new StreamReader(filename)) {
    int lineno = 0;
    for (String line = rd.ReadLine(); line != null; line = rd.ReadLine()) {
      String[] res = delim.Split(line);
      lineno++;
      foreach (String s in res)
        if (s != "") {
          if (!index.Contains(s))
            index[s] = new TreeSet<int>();
          index[s].Add(lineno);
        }
    }
  }
  return index;
}
```

Instead of a tree dictionary, one might have used a hash dictionary and then sort the entries before printing them. Instead of a tree set of line numbers one might have used a list (because the lines are scanned in order of increasing line numbers anyway) and either eliminate duplicates afterwards (see pattern 80), or use a hashed (array or linked) list to avoid duplicated during the construction. It is not clear that these alternatives would have been any more efficient. Using a tree dictionary and tree sets naturally achieves the absence of duplicates, and the result `index` returned by method `IndexFile` above will print in alphabetical order without further ado:

```
foreach (String word in index.Keys) {
  Console.Write("{0}: ", word);
  foreach (int ln in index[word])
    Console.Write("{0} ", ln);
  Console.WriteLine();
}
```

11.3 Convex hull in the plane

The *convex hull* is the least convex set that encloses a given set of points. A point set is convex if every point between to points that belong to the set also belongs to the set: the set has no holes or inward dents. Figure 11.1 shows an example of several points and their convex hull, enclosed by a solid line.

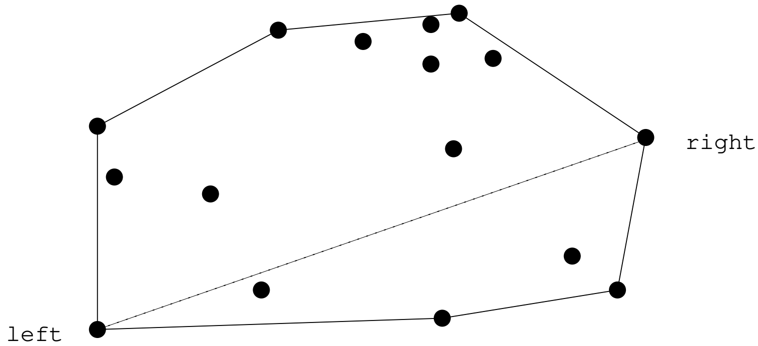


Figure 11.1: A point set and its convex hull in the plane.

Graham's algorithm [16] for finding the convex hull in the plane, as modified by Andrew [2], consists of four steps, implemented in file `GConvexHull.cs`:

1. Sort the points lexicographically by increasing (x,y), thus finding also a left-most point `left` and a rightmost point `right`. Several points may have the same minimal or maximal x coordinate; any such point will do.
2. Partition the point set into two lists, `upper` and `lower`, according as point is above or below the line segment from `left` to `right`, shown by a dotted line in figure 11.1. The upper list begins with point `left` and ends with `right`; the lower list begins with point `right` and ends with `left`.
3. Traverse the point lists clockwise, eliminating all but the extreme points (thus eliminating also duplicate points). This is Graham's point elimination scan.
4. Join the point lists (in clockwise order) in an array, leaving out point `left` from the `lower` list and point `right` from the `upper` list.

Using C5, these steps can be implemented as follows:

```
// 1. Sort points lexicographically by increasing (x, y)
int N = pts.Length;
Array.Sort(pts);
Point left = pts[0], right = pts[N - 1];
// 2. Partition into lower hull and upper hull
IList<Point> lower = new LinkedList<Point>(),
  upper = new LinkedList<Point>();
lower.InsertFirst(left); upper.InsertLast(left);
```

```
for (int i = 0; i < N; i++) {
  double det = Point.Area2(left, right, pts[i]);
  if (det < 0)
    lower.InsertFirst(pts[i]);
  else if (det > 0)
    upper.InsertLast(pts[i]);
}
lower.InsertFirst(right);
upper.InsertLast(right);
// 3. Eliminate points not on the hull
Eliminate(lower);
Eliminate(upper);
// 4. Join the lower and upper hull, leaving out lower.Last and upper.Last
Point[] res = new Point[lower.Count + upper.Count - 2];
lower[0, lower.Count - 1].CopyTo(res, 0);
upper[0, upper.Count - 1].CopyTo(res, lower.Count - 1);
```

Graham's point elimination scan, which is step (3) above, works as follows:

- Consider three consecutive points p_0, p_1, p_2.
- If they make a right turn, then p_1 is not in the convex set spanned by p_0, p_2 and other points; keep it.
- Otherwise eliminate p_1 and go back to reconsider p_0.

Using C5 list views, it can be implemented as follows:

```
public static void Eliminate(IList<Point> list) {
  IList<Point> view = list.View(0, 0);
  int slide = 0;
  while (view.TrySlide(slide, 3))
    if (Point.Area2(view[0], view[1], view[2]) < 0)   // right turn
      slide = 1;
    else {                                            // left or straight
      view.RemoveAt(1);
      slide = view.Offset != 0 ? -1 : 0;
    }
  }
}
```

In every iteration of the `while` loop, the `view` focuses on three consecutive points. If they form a right turn, slide `view` to the right (by 1); else remove the middle point, and slide `view` to the left (by -1) unless at the beginning of the list.

All the above operations, including `RemoveAt(1)`, are efficient when `list` is a linked list, and considerably clearer than implementations using explicit manipulation of linked list nodes. Also note that the point deletion operation `RemoveAt(1)` cannot be efficiently implemented when using array lists.

11.4 Finding anagram classes

Two words are anagrams of each other if one can be obtained from the other by rearranging its letters. For instance "generate" and "teenager" are anagrams of each other. From a collection point of view, two words are anagrams of each other if they contain the same bag of letters. For the above example, this bag is { a, e, e, e, g, n, r, t }.

Clearly, a collection of words can be divided into disjoint *anagram classes* such any two words in an anagram class are anagrams of each other, and no two words from different anagram classes are anagrams of each other. Each anagram class is characterized by a bag of letters (characters). An anagram class is trivial if it contains only one word. This example is in file `Anagrams.cs`.

Consider the following task: Divide a given word list into anagram classes, for each non-trivial anagram class, print the words it contains. The word list is presented as an enumerable of strings and may contain duplicates.

Given a word (string) we can find its anagram class (as a bag of characters) straightforwardly like this:

```
public static HashBag<char> AnagramClass(String s) {
  HashBag<char> anagram = new HashBag<char>();
  foreach (char c in s)
    anagram.Add(c);
  return anagram;
}
```

The default item equality comparer used by the `anagram` hash bag is CharEqualityComparer; see section 2.3. To manage the anagram classes and the words that belong to them, we use a dictionary that maps an anagram class to the set of the words in that class. Then we simply go through the given word list, find the anagram class of each word, and check whether we have seen that anagram class before, and if not, create a new entry for that anagram class in the dictionary and associate it with a new empty set of words. Then add the word to the set associated with its anagram class.

Method AnagramClasses below takes as argument a stream of words and returns a stream of the non-trivial anagram classes, each being a stream of words. A stream is represented as an enumerable of type SCG.IEnumerable<T>, where SCG abbreviates the System.Collections.Generic namespace. The variable classes holds the anagram classes found so far and has type IDictionary<HashBag<char>, TreeSet<String>>.

```
public static SCG.IEnumerable<SCG.IEnumerable<String>>
  AnagramClasses(SCG.IEnumerable<String> ss)
{
  IDictionary<HashBag<char>, TreeSet<String>> classes
    = new HashDictionary<HashBag<char>, TreeSet<String>>();
  foreach (String s in ss) {
    HashBag<char> anagram = AnagramClass(s);
    TreeSet<String> anagramClass;
    if (!classes.Find(anagram, out anagramClass))
      classes[anagram] = anagramClass = new TreeSet<String>();
    anagramClass.Add(s);
  }
  foreach (TreeSet<String> anagramClass in classes.Values)
    if (anagramClass.Count > 1)
      yield return anagramClass;
}
```

Note that we use collection types both for the key type (HashBag<char>) and the value type (TreeSet<String>) in the dictionary. In particular, the HashBag<char> key items must be compared by contents: two bags are equal if they contain the same characters with the same multiplicity. The key equality comparer used by the classes hash dictionary is the default equality comparer for the key type HashBag<char>: an unsequenced collection equality comparer; see section 2.3.

The words of each non-trivial anagram class found by method AnagramClasses(ss) can be printed, one anagram class per line, as shown below. The parameter ss has type SCG.IEnumerable<String>:

```
foreach (SCG.IEnumerable<String> anagramClass in AnagramClasses(ss)) {
  foreach (String s in anagramClass)
    Console.Write(s + " ");
  Console.WriteLine();
}
```

The anagram class problem could also be solved by using a TreeBag<char> anywhere a HashBag<char> is used above. This solution has similar efficiency for medium-size problems (100 000 anagram classes, of which 10 000 non-trivial), but uses slightly more memory and so is slower on systems that do not have sufficient RAM.

11.5 Finite automata

This example shows how to use sets, dictionaries and queues from the C5 library to systematically construct a deterministic finite automaton from a given nondeterministic finite automaton. The example is in file `GNfaToDfa.cs`.

A *nondeterministic finite automaton* (NFA) is a directed graph, as shown in figure 11.2, with *states* shown as ovals, and *transitions* shown as arrows from one state to another. A transition is labelled with a letter or with the special symbol epsilon (ε), written eps in the figure. An NFA has a single start state and one or more accept states, the latter shown as double ovals.

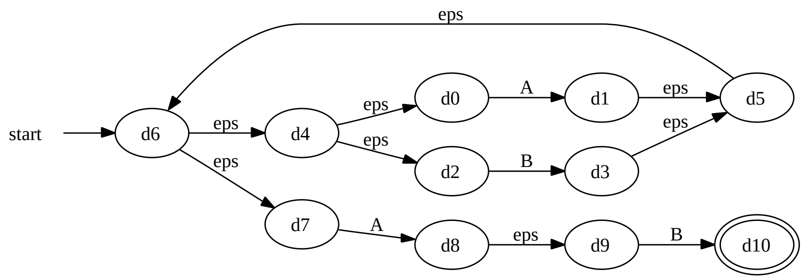


Figure 11.2: A nondeterministic finite automaton for $(A|B)*AB$.

A nondeterministic finite automaton is said to *recognize* a string such as *AAABAB* if there is a sequence of transitions from the start state to an accept state, such that the concatenation of the transitions' labels equals the string, ignoring epsilons. The NFA in figure 11.2 recognizes the strings *AB*, *AAB*, *BAB*, and infinitely many others; in fact, exactly those strings of *A*'s and *B*'s that end in *AB*. On the other hand, it does not recognize the strings *AAA* or *BA* or *B* or *A*.

A *deterministic finite automaton* (DFA) is an NFA that satisfies: (1) no state has more than one transition with the same label, and (2) there are no epsilon transitions. Figure 11.3 gives an example DFA.

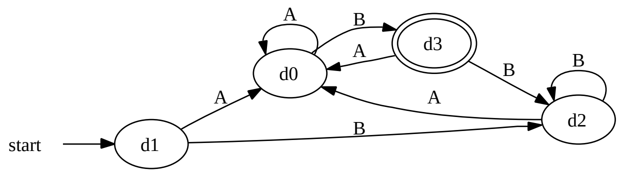


Figure 11.3: A deterministic finite automaton equivalent to the NFA in figure 11.2.

From any NFA one can systematically construct an equivalent DFA; in fact the DFA in figure 11.3 has been constructed from the NFA in figure 11.2. This construction is useful because there is a simple way to make an NFA from a given regular expression, and there is an efficient way to check whether a DFA recognizes a given string. Together, this gives an efficient way to check whether a particular string matches a given regular expression.

A nondeterministic finite automaton can be represented by a start state, an acceptance state (only one for simplicity), and its transition relation: a dictionary that maps a state (an `int`) to an array list of Transitions, where a Transition is a pair of a label `lab` and a `target` state. A label is a string, and we use the label `null` to denote epsilon transitions.

```
class Nfa {
  private readonly int startState;
  private readonly int exitState;    // This is the unique accept state
  private readonly IDictionary<int, ArrayList<Transition>> trans;
  ...
}

public class Transition {
  public readonly String lab;
  public readonly int target;
  public Transition(String lab, int target) {
    this.lab = lab; this.target = target;
  }
}
```

Given a nondeterministic finite automaton (NFA) we can construct a deterministic finite automaton (DFA) in two main steps:

1. Construct a DFA each of whose states is composite, namely a set of NFA states. This is done by methods `CompositeDfaTrans` and `EpsilonClose` below.

2. Replace each composite state (a Set<int>) by a simple state (an `int`). This is done by method `MkRenamer`, which creates a renamer, and method `Rename`, which applies the renamer to the composite-state DFA created in step 1.

The above steps can be implemented as follows:

```
// 1. Construct composite-state DFA
IDictionary<Set<int>, IDictionary<String, Set<int>>>
  cDfaTrans = CompositeDfaTrans(startState, trans);
// 2. Replace composite states with simple (int) states
ICollectionValue<Set<int>> cDfaStates = cDfaTrans.Keys;
IDictionary<Set<int>, int> renamer = MkRenamer(cDfaStates);
IDictionary<int, IDictionary<String, int>> simpleDfaTrans =
  Rename(renamer, cDfaTrans);
```

Method `EpsilonClose`, shown below, computes and returns the *epsilon-closure* of a given NFA state `s`, that is, the set of all NFA states that are reachable from `s` by epsilon transitions. This is done as follows: Let a set `S` of states be given. Put the states of `S` on a worklist. Repeatedly choose and remove a state `s` from the worklist, and consider all epsilon-transitions $s \xrightarrow{\varepsilon} s'$ from `s` to some state `s'`. If `s'` is in `S` already, then do nothing; otherwise add `s'` to `S` and to the worklist. When the worklist is empty, `S` is epsilon-closed; return `S`.

```
static Set<int>
  EpsilonClose(Set<int> S, IDictionary<int, ArrayList<Transition>> trans) {
  // The worklist initially contains all S members
  IQueue<int> worklist = new CircularQueue<int>();
  S.Apply(worklist.Enqueue);
  Set<int> res = new Set<int>(S);
  while (!worklist.IsEmpty) {
    int s = worklist.Dequeue();
    foreach (Transition tr in trans[s]) {
      if (tr.lab == null && !res.Contains(tr.target)) {
        res.Add(tr.target);
        worklist.Enqueue(tr.target);
      }
    }
  }
  return res;
}
```

Method `CompositeDfaTrans`, shown below, builds and returns the transition relation of a composite-state DFA equivalent to transition relation of the given NFA. This is done as follows: Create the epsilon-closure `S0` (a Set<`int`>) of the NFA's start state `s0`, and put it in a worklist (an IQueue<Set<`int`>>). Create an empty DFA transition relation `res`, which is a dictionary that maps a composite state (an epsilon-closed set of `int`s) to a dictionary that maps a label (a non-null string) to a composite state.

Repeatedly choose a composite state `S` from the worklist. If `S` is not already in the key set of the DFA transition relation, compute for every non-epsilon label `lab` the set `T` of states reachable by that label from some state `s` in `S`. Compute the epsilon-closure `Tclose` of every such state `T` and put it on the worklist. Then for every `lab`, add the transition $\texttt{S} \xrightarrow{\texttt{lab}} \texttt{Tclose}$ to the DFA transition relation.

```
static IDictionary<Set<int>, IDictionary<String, Set<int>>>
  CompositeDfaTrans(int s0, IDictionary<int, ArrayList<Transition>> trans) {
  Set<int> S0 = EpsilonClose(new Set<int>(s0), trans);
  IQueue<Set<int>> worklist = new CircularQueue<Set<int>>();
  worklist.Enqueue(S0);
  // The transition relation of the DFA
  IDictionary<Set<int>, IDictionary<String, Set<int>>> res =
    new HashDictionary<Set<int>, IDictionary<String, Set<int>>>();
  while (!worklist.IsEmpty) {
    Set<int> S = worklist.Dequeue();
    if (!res.Contains(S)) {
      // The S -lab-> T transition relation being constructed for a given S
      IDictionary<String, Set<int>> STrans =
        new HashDictionary<String, Set<int>>();
      // For all s in S, consider all transitions s -lab-> t
      foreach (int s in S) {
        // For all non-epsilon transitions s -lab-> t, add t to T
        foreach (Transition tr in trans[s]) {
          if (tr.lab != null) {       // Non-epsilon transition
            Set<int> toState;
            if (STrans.Contains(tr.lab))  // Already a transition on lab
              toState = STrans[tr.lab];
            else {                        // No transitions on lab yet
              toState = new Set<int>();
              STrans.Add(tr.lab, toState);
            }
            toState.Add(tr.target);
          }
        }
      }
      // Epsilon-close all T such that S -lab-> T, and put on worklist
      IDictionary<String, Set<int>> STransClosed =
        new HashDictionary<String, Set<int>>();
      foreach (KeyValuePair<String, Set<int>> entry in STrans) {
        Set<int> Tclose = EpsilonClose(entry.Value, trans);
        STransClosed.Add(entry.Key, Tclose);
        worklist.Enqueue(Tclose);
      }
      res.Add(S, STransClosed);
    }
  }
  return res;
}
```

Method `MkRenamer`, shown below, creates and returns a renamer, a dictionary that maps Set<int> to `int`. This is done as follows: Given a collection of Set<int>, create a new injective dictionary that maps from Set<int> to `int`, by choosing a fresh `int` for every key in the given dictionary. The collection of Set<int> is actually the key set of the composite-state DFA's transition relation, as computed by method `CompositeDfaTrans` above.

```
static IDictionary<Set<int>, int> MkRenamer(ICollectionValue<Set<int>> states) {
  IDictionary<Set<int>, int> renamer = new HashDictionary<Set<int>, int>();
  int count = 0;
  foreach (Set<int> k in states)
    renamer.Add(k, count++);
  return renamer;
}
```

Method `Rename`, shown below, creates and returns a DFA whose states are simple `int`s. This is done as follows: Given a renamer as constructed by method `MkRenamer`, and given the composite-state DFA's transition relation, create and return a new transition relation as a dictionary in which every Set<int> has been replaced by an `int`, as dictated by the renamer.

```
static IDictionary<int, IDictionary<String, int>>
  Rename(IDictionary<Set<int>, int> renamer,
         IDictionary<Set<int>, IDictionary<String, Set<int>>> trans)
{
  IDictionary<int, IDictionary<String, int>> newtrans =
    new HashDictionary<int, IDictionary<String, int>>();
  foreach (KeyValuePair<Set<int>, IDictionary<String, Set<int>>> entry
     in trans) {
    Set<int> k = entry.Key;
    IDictionary<String, int> newktrans = new HashDictionary<String, int>();
    foreach (KeyValuePair<String, Set<int>> tr in entry.Value)
      newktrans.Add(tr.Key, renamer[tr.Value]);
    newtrans.Add(renamer[k], newktrans);
  }
  return newtrans;
}
```

11.6 Topological sort

This example concerns topological sorting; it is in file `Toposort.cs`. A topological sort of a directed acyclic graph is a linear ordering of the graph's nodes, such that if node a points to node b in the graph, noted $a \to b$, then $a < b$ in the linear ordering. Note that there may be many different such linear orderings for a given graph. For instance, consider the graph in figure 11.4.

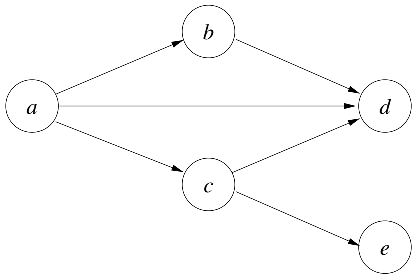


Figure 11.4: A directed acyclic graph with nodes a, b, c, d and e.

The graph has linear orderings $a < b < c < d < e$ but also $a < b < c < e < d$ and $a < c < b < d < e$ and $a < c < b < e < d$ and $a < c < e < b < d$.

Topological sort may be used in a spreadsheet implementation. Each cell formula in the spreadsheet is a node, and its children are the cells that the formula refers to, that is, the cells on which its value depends. A topological sort of the cell formulas can be used to determine the order in which to evaluate the cell formulas after a change.

11.6.1 Representation of graph nodes

Assume that a graph node has a name `id` and zero or more child nodes. A graph consists of a set of such nodes, together with a set of some of the nodes called the start set. More concretely, let a node with id of type T be represented in C# by an object of type Node<T>:

```
public class Node<T> {
  public readonly T id;
  public readonly Node<T>[] children;        // All non-null
  public Node(T id, params Node<T>[] children) {
    this.id = id; this.children = children;
  }
  public Node(Node<T> node) : this(node.id, new Node<T>[node.children.Length])
  { }
}
```

Thus `new Node(id,` n_1`, ...,` n_k`)` creates a node with name id and children $n_1, \ldots, n_k$. For instance, if node names are represented as strings, the graph in figure 11.4 can be created using this C# declaration:

```
Node<String>
  d = new Node<String>("d"),
  e = new Node<String>("e"),
  c = new Node<String>("c", d, e),
  b = new Node<String>("b", d),
  a = new Node<String>("a", b, c, d);
```

11.6.2 Standard depth-first topological sort

The standard algorithm for topological sort places the graph nodes in the order of increasing finish time in a depth-first search of the graph from the nodes in the start set [9, section 22.4].

This is easily implemented using the HashedLinkedList<T> class of the C5 library. We need two methods, `Toposort0` and `AddNode0`, where method `Toposort0` takes as argument the start set, creates a new empty list of sorted nodes, and then for each start node adds all its descendants:

```
public static IList<Node<T>> Toposort0<T>(params Node<T>[] starts) {
  HashedLinkedList<Node<T>> sorted = new HashedLinkedList<Node<T>>();
  foreach (Node<T> start in starts)
    if (!sorted.Contains(start))
      AddNode0(sorted, start);
  return sorted;
}
```

Method `AddNode0` takes as argument a node which is not yet in the sorted list. For each child of the given node it adds all the child's descendants by calling itself recursively, and finally adds the node:

```
private static void AddNode0<T>(IList<Node<T>> sorted, Node<T> node) {
  SDD.Assert(!sorted.Contains(node));
  foreach (Node<T> child in node.children)
    if (!sorted.Contains(child))
      AddNode0(sorted, child);
  sorted.InsertLast(node);
}
```

Provided the graph contains no cycles, method `Toposort0` returns a sorted list of nodes with the following property: every node appears strictly after all its descendants. If the algorithm terminates, it takes time linear in the number of nodes in the graph. However, if the graph contains a cycle, then there will be an infinite sequence of calls to `AddNode0` and the program will fail with stack overflow.

The call `SDD.Assert(...)` is to method `Assert` in class System.Diagnostics.Debug. It is used to assert a precondition for method `AddNode0`, and has no effect at runtime unless the example is compiled with option `/d:DEBUG`.

11.6.3 Improved depth-first algorithm for topological sort

The implementation of topological sort shown above exploits that in a hashed linked list, it is very fast to determine (using Contains) *whether* a node is in the list. By exploiting also that a hashed linked list knows *where* a node is in the list, we get a slightly better non-standard algorithm for topological sort: one that terminates even if the graph contains cycles. The main difference is that the above standard algorithm adds a node after adding its descendants, whereas the non-standard algorithm below adds a node before addings its descendants.

Method Toposort1 that takes as argument the set of start nodes, creates a new empty list of sorted graph nodes, and then adds each start node and its descendants:

```
public static IList<Node<T>> Toposort1<T>(params Node<T>[] starts) {
  HashedLinkedList<Node<T>> sorted = new HashedLinkedList<Node<T>>();
  foreach (Node<T> start in starts)
    if (!sorted.Contains(start)) {
      sorted.InsertLast(start);
      AddNode1(sorted, start);
    }
  return sorted;
}
```

Method AddNode1 takes as argument a node which is already in the sorted list. For each child of the given node it adds that child just before the given node, and then adds all the child's descendants by calling itself recursively.

```
private static void AddNode1<T>(IList<Node<T>> sorted, Node<T> node) {
  SDD.Assert(sorted.Contains(node));
  foreach (Node<T> child in node.children)
    if (!sorted.Contains(child)) {
      sorted.ViewOf(node).InsertFirst(child);
      AddNode1(sorted, child);
    }
}
```

Provided the graph contains no cycles, method Toposort1 returns a sorted list of nodes in which every node appears strictly after all its descendants. Also, it terminates even in case the graph contains cycles, but in that case, there will be at least one node in the sorted list that does not appear strictly after all its descendants. The algorithm takes time linear in the number of nodes in the graph.

Note that the precondition of AddNode1 is the logical negation of that of AddNode0.

The main shortcoming of the above depth-first algorithms is that they may perform long chains of recursive method calls, and thus may overflow the method call stack. Namely, if the graph contains a chain of the form $a_1 \to a_2 \to a_3 \to \cdots \to a_n$ where a_1 has child a_2, and a_2 and child a_3 so on, then there may be a chain of calls of the AddNode methods that has depth n.

This shortcoming can be addressed by keeping an explicit stack (using an array list, for instance) of node children yet to visit. Alternatively, one may consider the sorted list itself as a "worklist" of nodes whose children should be considered, using a list view as a cursor. This is at the expense of some extra work, especially if some nodes may have many children.

11.6.4 Topological sort by rescanning the sorted list

The last version of topological sort considered here uses a one-item list view as a cursor to scan the sorted list. When a new node is inserted into the list, the cursor points at that node. If the node under the cursor has a pending child — a child not yet on the sorted list — that child is inserted in at the left end of the cursor, the cursor is moved left by `Slide(0,1)`, and the same process is repeated. Otherwise, the cursor is moved right by `TrySlide(+1)` until a node is found that has a pending child, or until the right end of the list is reached. The loop invariant is that no node strictly to the left of the cursor has a pending child.

```
public static IList<Node<T>> Toposort2<T>(params Node<T>[] starts) {
  HashedLinkedList<Node<T>> sorted = new HashedLinkedList<Node<T>>();
  foreach (Node<T> start in starts)
    if (!sorted.Contains(start)) {
      sorted.InsertLast(start);
      using (IList<Node<T>> cursor = sorted.View(sorted.Count-1,1)) {
        do {
          Node<T> child;
          while (null != (child = PendingChild(sorted, cursor.First))) {
            cursor.InsertFirst(child);
            cursor.Slide(0,1);
          }
        } while (cursor.TrySlide(+1));
      }
    }
  return sorted;
}
static Node<T> PendingChild<T>(IList<Node<T>> sorted, Node<T> node) {
  foreach (Node<T> child in node.children)
    if (!sorted.Contains(child))
      return child;
  return null;
}
```

The main drawback of this procedure is that the child list of a node may be scanned repeatedly, which means that the running time for each node is quadratic in its number of children. If the maximal number of children for a node is bounded (say, 3), then this is not a cause for concern.

11.7 Copying a graph

Assume we have an arbitrary graph represented as in section 11.6.1, and that we want to make an exact shallow copy of that graph. The new copy must consist of fresh Node<T> objects, and its structure, as expressed by the `children` references, must be the same as that of the given old graph, preserving sharing and cycles. This example is in file `Graphcopy.cs`.

11.7.1 Graph copying using a recursive helper method

One way to create such a copy is to build a dictionary that maps each old node to a new node, using a reference equality comparer to distinguish nodes by their object identity. In fact, the resulting dictionary is an isomorphism between the old and the new graph. Initially the dictionary is empty, and new graph's start is the copy of the old graph's start node.

```
public static Node<T> CopyGraph0<T>(Node<T> start) {
  IDictionary<Node<T>,Node<T>> iso
    = new HashDictionary<Node<T>,Node<T>>
          (ReferenceEqualityComparer<Node<T>>.Default);
  return CopyNode0(iso, start);
}
```

The real work is done in method `CopyNode0`, which returns the new node corresponding to a given old node. If the old node is already in the dictionary's key set, then the corresponding new node is obtained from the dictionary. Otherwise a new node is created as a copy of the old node, the dictionary is extended to map the old node to the new one, and the new node's children are set to be the copies of the old node's children:

```
private static Node<T> CopyNode0<T>(IDictionary<Node<T>,Node<T>> iso,
                                    Node<T> old) {
  Node<T> copy;
  if (!iso.Find(old, out copy)) {
    copy = new Node<T>(old);
    iso[old] = copy;
    for (int i=0; i<copy.children.Length; i++)
      copy.children[i] = CopyNode0(iso, old.children[i]);
  }
  return copy;
}
```

It is important that dictionary is updated before the children of the old node is considered; otherwise the copying could fail on cyclic graphs while attempting to unroll a cycle.

11.7.2 Graph copying using an explicit stack

The straightforward recursive algorithm above may fail with a stack overflow exception if applied to a graph that has a very long path. To avoid that, one may use a stack or queue of pending nodes: nodes whose children still need to be copied, as in the following algorithm:

```
public static Node<T> CopyGraph1<T>(Node<T> start) {
  IDictionary<Node<T>,Node<T>> iso
    = new HashDictionary<Node<T>,Node<T>>
          (ReferenceEqualityComparer<Node<T>>.Default);
  IStack<Node<T>> work = new ArrayList<Node<T>>();
  iso[start] = new Node<T>(start);
  work.Push(start);
  while (!work.IsEmpty) {
    Node<T> node = work.Pop(), copy = iso[node];
    for (int i=0; i<node.children.Length; i++) {
      Node<T> child = node.children[i];
      Node<T> childCopy;
      if (!iso.Find(child, out childCopy)) {
        iso[child] = childCopy = new Node<T>(child);
        work.Push(child);
      }
      copy.children[i] = childCopy;
    }
  }
  return iso[start];
}
```

As before, the dictionary maps an old node to its new copy. The work stack holds nodes that have already been copied, but whose children may still need to be copied. Every node in the stack is also in the key set of the dictionary.

11.8 General graph algorithms

This section gives a rudimentary description of a more general graph library, implemented in file `Graph.cs`. The graph library consists of an interface for edge-weighted graphs, an implementation based on an adjacency list representation using hash dictionaries, and these example algorithms:

- Breadth-First-Search and Depth-First-Search, with an interface based on actions to be taken as edges are traversed. Applications are: checking for connectedness, and counting components. This illustrates the use of hash sets and of LinkedList<Edge<V,E>> to determine breadt-first or depth-first search.
- Priority-First-Search, where edges are traversed according to either weight or accumulated weight from the start of the search. An application of the non-accumulating version is the construction of a minimal spanning tree, used in the approximate algorithm for the Traveling Salesman Problem below. Applications of the accumulating version are: the construction of a shortest path and the computation of the distance from one vertex to another one. This illustrates the use of hash dictionaries, priority queues and priority queue handles.
- An approximation algorithm for the Traveling Salesman Problem, when the edge weights satisfies the triangle inequality [32].

The interfaces and classes of the graph library uses these generic parameters:

- Generic parameter V stands for the type of a vertex in a graph. Vertices are identified by the `Equals` method inherited from object (or overridden in the vertex class).
- Generic parameter E stands for the type of additional data associated with edges in a graph.
- Generic parameter W stands for the type of weights on edges in a weighted graph, in practice usually `int` or `double`. Values of type W must be comparable, that is, `W : IComparable<W>`, and there must be given a compatible way to add values of type W.

Interface IGraph<V,E,W> is the interface for a graph implementation with vertex type V, edge data type E and edge weight values of type W.

Interface IWeight<E,W> describes a weight function that maps an edge data value of type E to a weight value of type W, and an operation to add two weight values of type W giving a value of type W.

Class HashGraph<V,E,W> provides an implementation of IGraph<V,E,W> based on adjacency lists represented as hash dictionaries.

Class CountWeight<E> implements IWeight<E,`int`>, class IntWeight implements IWeight<`int,int`>, and class DoubleWeight implements IWeight<`double,double`>, and they are used to represent commonly occurring types of edge weights.

Struct type Edge<V,E> is the type of an edge in a graph with vertices of type V and edge data of type E.

Delegate type EdgeAction<V,E,U> is the type of an action to perform on each edge when traversing a graph with edges of type Edge<V,E> and with additional edge information (e.g. weight data) of type U.

11.9 Point location in the plane

Assume that the plane is divided into cells, delimited by edges (straight line segments) such as the solid lines in figure 11.5. For instance, such a division may represent plots of land, the fields of a farm, or the rooms and corridors of a complex building.

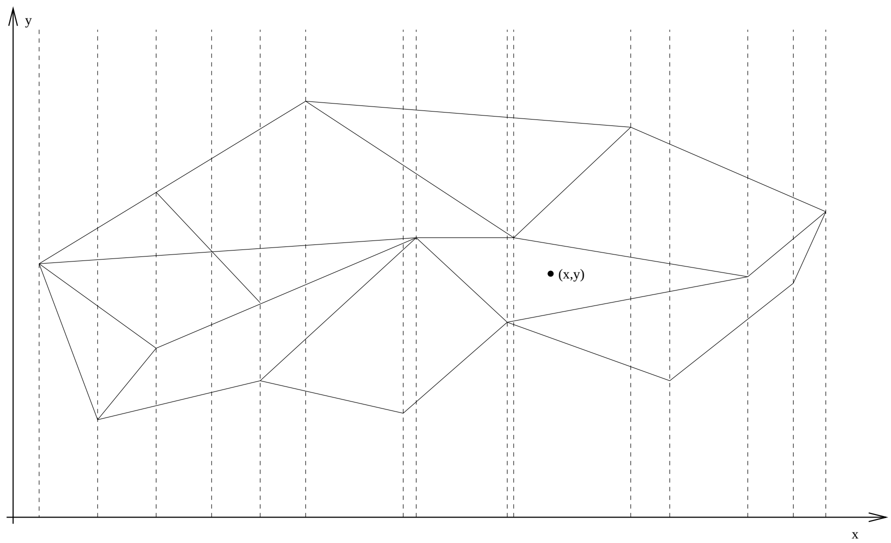


Figure 11.5: Dividing the plane into slices for efficient point location.

Then one can ask the question of any given point (x,y) in the plane: which cell does this point belong to? This question can be answered efficiently by using tree-based collections and taking snapshots (sections 6.8 and 8.5). This is a classical example of using persistent trees by Sarnak and Tarjan [25]. The example is in file `PointLocation.cs`.

The solution is to use an outer sorted dictionary that maps each x coordinate of an edge endpoint to an inner sorted set of the edges that cross or touch the vertical line at that x coordinate. The edges contained in the inner sorted set are ordered by their y coordinates to the immediate right of x. In the figure, the positions of the inner sorted sets are shown as vertical dashed lines. Note that there is an inner sorted set for each edge endpoint in the plane.

To look up a point (x, y) one first finds the predecessor of x in the outer sorted dictionary, searching on x only. The result is an inner sorted set, in which one then locates the edges above and below (x, y) by searching on the y coordinate only. The complete lookup takes time $O(\log n)$ where n is the number of edges.

The whole data structure can be built efficiently by letting the inner sorted sets be snapshots of the same sorted set, created in order of increasing x coordinate. These snapshots are created by maintaining a sorted tree set of edges, inserting and deleting edges during a horizontal (x axis) sweep, taking a snapshot of the inner tree set and saving it in the outer sorted dictionary at each x coordinate that causes one or more edges to be added or deleted. Thus snapshots are taken exactly at the edge endpoints.

If there are n edges, there will be $2n$ updates to the inner sorted tree, and in the worst case, the inner tree will have size $\Omega(n)$ for $\Omega(n)$ snapshots. We will use $O(n \log n)$ time and $O(n)$ space for sorting the endpoints. Since C5 uses node copy persistence (section 13.10) for snapshots of the inner sorted trees, we will use $O(n)$ space and $O(n)$ time to build the data structure. This is far better than the naive approach of making full copies of non-persistent inner trees, which would use up to $O(n^2)$ space and time to build the datastructure.

Lookup will take $O(\log n)$ time in any case, but taking the memory hierarchy into consideration, a low space use is very beneficial for large problems.

The code for this example present demonstrates the use of snapshots as well as the `Cut` method from interface ISorted<T> (section 4.13).

11.10 A batch job queue

This example concerns management of computer batch jobs using priority queues. The example is in file `Jobqueue.cs`.

The Blast server at the National Center for Biotechnology Information in Maryland, USA, can search a gene database for DNA sequences that are similar to a given DNA sequence. Such searches are performed from a batch job queue. To prevent a single user from stealing all cpu power, a search job submitted at time t is scheduled to be executed at time $t+60n$ where n is the number of search jobs from that uses already in the job queue. Hence a single search job will be allowed to run immediately (if there is a processor available), but if a second search job is submitted immediately after the first one, it will have to wait at least 60 seconds before it gets executed (if there is other work to do with an earlier scheduled execution time). Submitted jobs are identified unique by a request id (RID), and here we assume users are identified by the submitting machine's IP address. Submitted jobs can be retracted (deleted) using their RID.

Let Ip be the type of IP numbers, let Rid be the type of request ids, and let Job be the type of search jobs. A Job object should contain at least an Ip, a Rid, and the job's scheduled execution time and should implement IComparable<Job> based on scheduled time. We can implement the policy described above by maintaining three data structures:

- An IPriorityQueue<Job> called `jobQueue` that contains the jobs, ordered according to scheduled execution time.
- An IDictionary<Rid,IPriorityQueueHandle<Job>> called `jobs` that maps a request id to the associated handle in the priority queue.
- A HashBag<Ip> called `userJobs` containing the IP numbers of users, with the same multiplicity that that user's jobs appear in the priority queue. There is no efficient way to find this number using only the priority queue.

The following operations can be performed on the batch job system:

- `Submit(ip, t)` should find the number of existing jobs from the same user `ip` by `userJobs.ContainsCount(ip)`, compute the scheduled execution time, create a Rid object `rid`, create a Job object `job` and insert it into the priority queue obtaining a handle `h` in return by `jobQueue.Add(job, out h)`; add the `ip` to the hash bag by `userJobs.Add(ip)`; and insert the rid-to-handle mapping in the dictionary by `jobs[rid] = h`. Total time: $O(\log(n))$ where n is the number of pending jobs.
- `ExecuteOne()` should find a job with minimal scheduled execution time and remove it from the priority queue by `job = jobQueue.DeleteMin()`, remove the job's IP once from the hash bag by `userJobs.Remove(job.ip)`, and remove the job's RID from the dictionary by `jobs.Remove(job.rid)`. Total time: $O(\log(n))$ where n is the number of pending jobs.

- `Cancel(rid)` should find the handle corresponding to `rid` in the dictionary and remove it (if it is present) by `present = jobs.Find(rid, out h)`; find the job corresponding to the handle in the priority queue and remove it by `job = jobQueue.Delete(h)`; and remove the job's IP once from the hash bag by `jobs.Remove(job.ip)`. Total time: $O(\log(n))$ where n is the number of pending jobs.

In reality these three operations are likely to happen asynchronously, since new search jobs (and possibly cancellation requests) come in via the web while at the same time several processors are removing jobs from the job queue to execute them.

Clearly some form of synchronization is needed to keep the three data structures consistent with each other. One way is for each of the three above operations to lock on some object (for instance, `jobQueue`) for the duration of the operation. This avoids scenarios such as overlapped execution of scheduling and cancelling: `ExecuteOne` might remove a job from the job queue (but not get to remove of from the dictionary) and at the same time `Cancel` gets and removed the same job from the dictionary. When subsequently `ExecuteOne` tries to remove the job from the dictionary, it fails; and so does `Cancel` when it tries to remove the job from the job queue. For simplicity this synchronization has been left out of the example.

Here are the auxiliary classes for representing Jobs, Rids and Ip numbers:

```
class Job : IComparable<Job> {
  public readonly Rid rid;
  public readonly Ip ip;
  public readonly int time;
  public Job(Rid rid, Ip ip, int time) { ... }
  public int CompareTo(Job that) {
    return this.time - that.time;
  }
  public override String ToString() { ... }
}
class Rid {
  private readonly int ridNumber;
  private static int nextRid = 1;
  public Rid() { ... }
  public override String ToString() { ... }
}
class Ip {
  public readonly String ipString;
  public Ip(String ipString) { ... }
  public override int GetHashCode() {
    return ipString.GetHashCode();
  }
  public override bool Equals(Object that) {
    return that != null
           && that is Ip
           && this.ipString.Equals(((Ip)that).ipString);
  }
}
```

The job queue class and its operations can be implemented like this:

```
class JobQueue {
  private readonly IPriorityQueue<Job> jobQueue;
  private readonly IDictionary<Rid,IPriorityQueueHandle<Job>> jobs;
  private readonly HashBag<Ip> userJobs;

  public JobQueue() {
    this.jobQueue = new IntervalHeap<Job>();
    this.jobs = new HashDictionary<Rid,IPriorityQueueHandle<Job>>();
    this.userJobs = new HashBag<Ip>();
  }

  public Rid Submit(Ip ip, int time) {
    int jobCount = userJobs.ContainsCount(ip);
    Rid rid = new Rid();
    Job job = new Job(rid, ip, time + 60 * jobCount);
    IPriorityQueueHandle<Job> h = null;
    jobQueue.Add(ref h, job);
    userJobs.Add(ip);
    jobs.Add(rid, h);
    Console.WriteLine("Submitted {0}", job);
    return rid;
  }

  public Job ExecuteOne() {
    if (!jobQueue.IsEmpty) {
      Job job = jobQueue.DeleteMin();
      userJobs.Remove(job.ip);
      jobs.Remove(job.rid);
      Console.WriteLine("Executed {0}", job);
      return job;
    } else
      return null;
  }

  public void Cancel(Rid rid) {
    IPriorityQueueHandle<Job> h;
    if (jobs.Remove(rid, out h)) {
      Job job = jobQueue.Delete(h);
      userJobs.Remove(job.ip);
      Console.WriteLine("Cancelled {0}", job);
    }
  }
}
```

A possible sequence of operations is:

```
JobQueue jq = new JobQueue();
// One user submits three jobs at time=27
Rid rid1 = jq.Submit(new Ip("62.150.83.11"), 27),
    rid2 = jq.Submit(new Ip("62.150.83.11"), 27),
    rid3 = jq.Submit(new Ip("62.150.83.11"), 27);
// One job is executed
jq.ExecuteOne();
// Another user submits two jobs at time=55
Rid rid4 = jq.Submit(new Ip("130.225.17.5"), 55),
    rid5 = jq.Submit(new Ip("130.225.17.5"), 55);
// One more job is executed
jq.ExecuteOne();
// The first user tries to cancel his first and last job
jq.Cancel(rid1);
jq.Cancel(rid3);
// The remaining jobs are executed
while (jq.ExecuteOne() != null) { }
```

This will produce the following output:

```
Submitted rid=1
Submitted rid=2
Submitted rid=3
Executed rid=1
Submitted rid=4
Submitted rid=5
Executed rid=4
Cancelled rid=3
Executed rid=2
Executed rid=5
```

11.11 A functional hash-based set implementation

This section shows how to declare a class Set<T> with functional or declarative set operations as a simple subclass of HashSet<T>. This approach has the advantage that the set implements ICollectionValue<T> and hence SCG.IEnumerable<T>, but it has the disadvantage that a Set<T> object is not protected against destructive modification. The example is in file `Sets.cs`.

The operators +, - and * are overloaded to implement set union ($A \cup B$), set difference ($A \setminus B$) and set intersection ($A \cap B$).

```
public class Set<T> : HashSet<T> {
  public static Set<T> operator +(Set<T> s1, Set<T> s2) {
    if (s1 == null || s2 == null)
      throw new ArgumentNullException("Set+Set");
    else {
      Set<T> res = new Set<T>(s1);
      res.AddAll(s2);
      return res;
    }
  }
  public static Set<T> operator -(Set<T> s1, Set<T> s2) {
    if (s1 == null || s2 == null)
      throw new ArgumentNullException("Set-Set");
    else {
      Set<T> res = new Set<T>(s1);
      res.RemoveAll(s2);
      return res;
    }
  }
  public static Set<T> operator *(Set<T> s1, Set<T> s2) {
    if (s1 == null || s2 == null)
      throw new ArgumentNullException("Set*Set");
    else {
      Set<T> res = new Set<T>(s1);
      res.RetainAll(s2);
      return res;
    }
  }
  ...
  public Set(SCG.IEnumerable<T> enm) : base() {
    AddAll(enm);
  }
  public Set(params T[] elems) : this((SCG.IEnumerable<T>)elems) { }
}
```

The Set<T> class has a constructor that creates a set from an enumerable (which may be another set) and a variable-arity constructor that creates a set from a list of items.

The subset and superset relations can be defined quite efficiently as follows:

```
public static bool operator <=(Set<T> s1, Set<T> s2) {
  if (s1 == null || s2 == null)
    throw new ArgumentNullException("Set<=Set");
  else
    return s1.ContainsAll(s2);
}
public static bool operator >=(Set<T> s1, Set<T> s2) {
  if (s1 == null || s2 == null)
    throw new ArgumentNullException("Set>=Set");
  else
    return s2.ContainsAll(s1);
}
```

The equality operator (==) could be defined as inclusion both ways, but another and more efficient version is obtained by using the `Equals(Set<T>,Set<T>)` method from the default equality on the Set<T> collection. Since the default equality comparer correctly handles `null` values, it can be called directly to compare the sets `s1` and `s2`:

```
public static bool operator ==(Set<T> s1, Set<T> s2) {
  return EqualityComparer<Set<T>>.Default.Equals(s1, s2);
}
public static bool operator !=(Set<T> s1, Set<T> s2) {
  return !(s1 == s2);
}
```

One would expect that if `s1<=s2` and `s2<=s1` then `s1==s2`. In fact, the default equality comparer in question is UnsequencedCollectionEqualityComparer<T,Set<T>>, whose equality works by checking that `s1` and `s2` have equally many items and that every item in one set is also in the other. Hence this expected property holds.

Also, once one has overloaded the operators (==) and (!=) one should also override the `Equals(Object)` and `GetHashCode()` methods. These can simply call the `Equals(Set<T>,Set<T>)` and `GetHashCode(Set<T>)` methods of the default equality comparer.

```
public override bool Equals(Object that) {
  return this == (that as Set<T>);
}
public override int GetHashCode() {
  return EqualityComparer<Set<T>>.Default.GetHashCode(this);
}
```

To illustrate the use of Set<T>, consider some simple computations on sets of integers, and sets of sets of integers:

```
Set<int> s1 = new Set<int>(2, 3, 5, 7, 11);
Set<int> s2 = new Set<int>(2, 4, 6, 8, 10);
Console.WriteLine("s1 + s2 = {0}", s1 + s2);
```

```
Console.WriteLine("s1 * s2 = {0}", s1 * s2);
Console.WriteLine("s1 - s2 = {0}", s1 - s2);
Console.WriteLine("s1 - s1 = {0}", s1 - s1);
Console.WriteLine("s1 + s1 == s1 is {0}", s1 + s1 == s1);
Console.WriteLine("s1 * s1 == s1 is {0}", s1 * s1 == s1);
Set<Set<int>> ss1 = new Set<Set<int>>(s1, s2, s1 + s2);
Console.WriteLine("ss1 = {0}", ss1);
```

As a more advanced application of the Set<T> class, consider computing the intersection closure of a finite set ss of finite sets. The *intersection closure* of ss is the smallest set tt containing ss such that whenever two sets s and t are in tt, then their intersection s ∩ t is in tt also.

The intersection closure can be computed by maintaining a worklist of sets, initially containing the sets from ss. Then one repeatedly selects a set s from the worklist, adds it to tt, and for each t in tt adds the set s ∩ t to the worklist unless it is already in tt. When the worklist is empty, tt is the intersection closure of ss.

The intersection closure is computed by the generic method IntersectionClose below. The input and output sets are represented as objects of type Set<Set<T>>, but internally in the method a HashSet<Set<T>> is used to hold the result tt while it is being computed. This is because the intermediate sets of sets are not of interest; it suffices to build up the result set by creating and modifying a single set object tt. In the end, a new declarative set is created that contains all the sets from tt, and that set is returned.

```
static Set<Set<T>> IntersectionClose<T>(Set<Set<T>> ss) {
  IQueue<Set<T>> worklist = new CircularQueue<Set<T>>();
  foreach (Set<T> s in ss)
    worklist.Enqueue(s);
  HashSet<Set<T>> tt = new HashSet<Set<T>>();
  while (worklist.Count != 0) {
    Set<T> s = worklist.Dequeue();
    foreach (Set<T> t in tt) {
      Set<T> ts = t * s;
      if (!tt.Contains(ts))
        worklist.Enqueue(ts);
    }
    tt.Add(s);
  }
  return new Set<Set<T>>((SCG.IEnumerable<Set<T>>)tt);
}
```

Here are two example computations of intersection-closed sets:

```
ss1 = {{6,4,10,2,8},{3,6,4,7,10,2,5,8,11},{3,7,2,5,11}}
IntersectionClose(ss1) = {{6,4,10,2,8},{3,6,4,7,10,2,5,8,11},{3,7,2,5,11},{2}}
ss2 = {{3,2},{3,1},{1,2}}
IntersectionClose(ss2) = {{},{3,2},{1},{3},{2},{3,1},{1,2}}
```

11.12 Implementing multidictionaries

A multidictionary or multi-valued dictionary is a dictionary in which each key can be associated with a (non-empty) collection of values rather than just a single value. The example implementations of a multidictionary below are in example file `MultiDictionary.cs`.

11.12.1 Basic implementation

A class MultiHashDictionary<K,V> of hash-based multidictionaries with keys of type K and values of type V can be based on a HashDictionary<K,ICollection<V>>:

```
public class MultiHashDictionary<K,V> : HashDictionary<K, ICollection<V>> {
  public virtual void Add(K k, V v) {
    ICollection<V> values;
    if (!base.Find(k, out values) || values == null) {
      values = new HashSet<V>();
      Add(k, values);
    }
    values.Add(v);
  }

  public virtual bool Remove(K k, V v) {
    ICollection<V> values;
    if (base.Find(k, out values) && values != null) {
      if (values.Remove(v)) {
        if (values.IsEmpty)
          base.Remove(k);
        return true;
      }
    }
    return false;
  }

  public override bool Contains(K k) {
    ICollection<V> values;
    return base.Find(k, out values) && values != null && !values.IsEmpty;
  }

  public override ICollection<V> this[K k] {
    get {
      ICollection<V> values;
      return base.Find(k, out values) && values!=null ? values : new HashSet<V>();
    }
    set { base[k] = value; }
  }

  ... implement property Count, see below ...
}
```

The new method `Add(k, v)` just adds `v` to the value collection associated with `k`, if there is one; otherwise it creates a new collection containing only `v` and associates it with `k`.

The new method `Remove(k, v)` removes the single (key,value) pair consisting of `k` and `v`, if any, from the multidictionary. To do so, it must first check that any value collection is associated with `k`, and if so, remove `v` from it; then if the resulting collection becomes empty, it removes that collection from the base hash dictionary.

The override of method `Contains(k)` must not only check whether the base hash dictionary associates a value collection with key `k`, but also that the value collection is non-`null` and non-empty. Namely, a client could use methods inherited from the base dictionary to associate a `null` or empty collection with a key.

The get accessor of the indexer `this[k]` returns the value collection associated with `k`, if any, or else a fresh empty collection.

What is missing at this point is an implementation of the `Count` property, which should return the total number of (key,value) pairs in the multidictionary; the one inherited from the base class returns just the number of keys. The simplest approach is to sum the number of items in all the value collections at each use of `Count`, but that makes it a linear-time operation:

```
public new virtual int Count {
  get {
    int count = 0;
    foreach (KeyValuePair<K,ICollection<V>> entry in this)
      if (entry.Value != null)
        count += entry.Value.Count;
    return count;
  }
}

public override Speed CountSpeed {
  get { return Speed.Linear; }
}
```

To do better than this, we need to track all changes to the base dictionary and to the value collections. We cannot rely on the multidictionary methods to do that, because the individual value collections may be modified directly, without involving the multidictionary.

11.12.2 Making `Count` a constant-time operation

Fortunately, in the C5 collection library one can add event listeners to dictionaries and collections, and thus track modifications. The idea is to add a private field `count` to the multidictionary, define event listeners that maintain the value of this field by incrementing and decrementing it, and associate these event listerners with each value collection.

```
private int count = 0;      // Cached value count, updated by events only
```

```
private void IncrementCount(Object sender, ItemCountEventArgs<V> args) {
  count += args.Count;
}
private void DecrementCount(Object sender, ItemCountEventArgs<V> args) {
  count -= args.Count;
}
private void ClearedCount(Object sender, ClearedEventArgs args) {
  count -= args.Count;
}
```

In turn, to make sure that each value collection gets the appropriate event listeners, we add two event listeners to the base dictionary in the multidictionary's constructor:

```
public MultiHashDictionary() {
  ItemsAdded +=
    delegate(Object sender, ItemCountEventArgs<KeyValuePair<K,ICollection<V>>> args) {
      ICollection<V> values = args.Item.Value;
      if (values != null) {
        count += values.Count;
        values.ItemsAdded += IncrementCount;
        values.ItemsRemoved += DecrementCount;
        values.CollectionCleared += ClearedCount;
      }
    };
  ItemsRemoved +=
    delegate(Object sender, ItemCountEventArgs<KeyValuePair<K,ICollection<V>>> args) {
      ICollection<V> values = args.Item.Value;
      if (values != null) {
        count -= values.Count;
        values.ItemsAdded -= IncrementCount;
        values.ItemsRemoved -= DecrementCount;
        values.CollectionCleared -= ClearedCount;
      }
    };
}
```

With this machinery in place, an addition to a value collection will raise an ItemsAdded event, which will invoke the `IncrementCount` method and adjust the `count` field of the multidictionary correctly; and analogously for ItemsRemoved events and CollectionCleared events. This works even if the value collection is associated with more than one multidictionary, or associated multiple times with the same multidictionary (through multiple keys).

The implementations of `Add`, `Remove`, `this[k]` and so on are unaffected by these changes, but the implementation of `Count` is now a trivial constant-time operation:

```
public new virtual int Count {
  get { return count; }
}
```

Also, the multidictionary's `Clear()` method must be overridden so that it decrements the multidictionary's count field and removes the event listeners from value collections. Otherwise updates to value collections would continue to affect the multidictionary after its was cleared, which would be wrong:

```
public override void Clear() {
  foreach (ICollection<V> values in Values)
    if (values != null) {
      count -= values.Count;
      values.ItemsAdded -= IncrementCount;
      values.ItemsRemoved -= DecrementCount;
      values.CollectionCleared -= ClearedCount;
    }
  base.Clear();
}
```

11.12.3 Using the multidictionary

To illustrate the use of the multidictionary, we create a multidictionary `mdict` that maps an integer key to one or more strings — the names of that integer in various languages.

```
MultiHashDictionary<int,String> mdict = new MultiHashDictionary<int,String>();
mdict.Add(2, "to");
mdict.Add(2, "deux");
mdict.Add(2, "two");
mdict.Add(20, "tyve");            // #1
mdict.Add(20, "twenty");
mdict.Remove(20, "tyve");
mdict.Remove(20, "twenty");       // #2
ICollection<String> zwei = new HashSet<String>();
zwei.Add("zwei");
mdict[2] = zwei;
mdict[-2] = zwei;                 // #3
zwei.Add("kaksi");                // #4
ICollection<String> empty = new HashSet<String>();
mdict[0] = empty;                 // #5
mdict.Remove(-2);                 // #6
zwei.Remove("kaksi");             // #7
zwei.Clear();                     // #8
```

The contents and `Count` of the multidictionary at the indicated program points are:

```
#1 { 20 => { tyve, twenty }, 2 => { two, deux, to } }
   mdict.Count is 5
   mdict[2].Count is 3
#2 { 2 => { two, deux, to } }
   mdict.Count is 3
```

```
#3 { -2 => { zwei }, 2 => { zwei } }
   mdict.Count is 2
#4 { -2 => { zwei, kaksi }, 2 => { zwei, kaksi } }
   mdict.Count is 4
#5 { 0 => {  }, -2 => { zwei, kaksi }, 2 => { zwei, kaksi } }
   mdict.Count is 4
   mdict contains key 0: False
#6 { 0 => {  }, 2 => { zwei, kaksi } }
   mdict.Count is 2
#7 { 0 => {  }, 2 => { zwei } }
   mdict.Count is 1
#8 { 0 => {  }, 2 => {  } }
   mdict.Count is 0
```

11.12.4 Choosing the value set representation

The multidictionary class in section 11.12.1 used hash sets for the value collections, but it might as well have used hash bags, tree sets, tree bags, hashed array lists, hashed linked lists, or some other collection. Moreover, the underlying dictionary might have been a tree dictionary instead of a hash dictionary, so one can image at least 12 possible multidictionary implementations; clearly too many to provide all of them explicitly.

Fortunately, one can use an additional type parameter `VC` to generalize the multidictionary so that two implementations suffice: one based on a hash dictionary (shown below) and one based on a tree dictionary.

The additional type parameter `VC` stands for the desired type of value collection, and therefore has a constraint that requires it to be a collection with item type `V`, and to have an argumentless constructor:

```
public class MultiHashDictionary<K,V,VC> : HashDictionary<K, VC>
  where VC : ICollection<V>, new()
{ ... }
```

In the body `{ ... }` of the multidictionary we just need to replace `new HashSet<V>()` with `new VC()`, and voilà, we have a typesafe and general multidictionary implementation that works for any collection type `VC`. In particular, multidictionary created in section 11.12.3, which maps an integer to a hash set of strings, can be obtained as follows:

```
MultiHashDictionary<int,string,HashSet<string>> mdict
  = new MultiHashDictionary<int,string,HashSet<string>>();
```

Actually there are at least two other ways in which the hard-coding of the value collection type as HashSet<V> can be avoided. That gives a total of three ways to present general typesafe multidictionaries with flexible value set type:

Our first MultiDictionary implementation hardcoded the use of HashSet<V> for the value collections. This hardcoding can be avoided in three ways:

(1) The one shown above, where the value collection type VC is exposed to the client of the dictionary:

```
public class MultiHashDictionary<K,V,VC>
             : HashDictionary<K, VC>
  where VC : ICollection<V>, new()
{ ... }
```

(2) A variant of this that exposes the ICollection<V> interface to the client:

```
public class MultiHashDictionary<K,V,VC>
             : HashDictionary<K, ICollection<V>>
  where VC : ICollection<V>, new()
{ ... }
```

(3) A third possibility is to use a delegate of type Fun<ICollection<V>> to create new value collections. This delegate must be passed to the multidictionary constructor when making a multidictionary instance:

```
public class MultiHashDictionary<K,V>
             : HashDictionary<K, ICollection<V>>
{
   private readonly Fun<ICollection<V>> vcFactory;
   public MultiHashDictionary(Fun<ICollection<V>> vcFactory) {
     this.vcFactory = vcFactory;
   }
   ...
}

MultiHashDictionary<int, String> mdict
  = new MultiHashDictionary<int,String>(
        delegate{ return new HashSet<String>(); });
```

All three approaches provide equally good typesafety inside the multidictionary, but (1) provides the best external typesafety, ensuring that *only* instances of VC can be added to the multidictionary. With (2), this restriction is gone, but on the other hand allows the created multidictionary to implement an interface IMultiDictionary<K,V> that derives from IDictionary<K,ICollection<V>>:

```
public interface IMultiDictionary<K,V> : IDictionary<K,ICollection<V>>
{ ... }
```

The main shortcoming of (1) and (2) is that only the collection's default constructor can be used. No parameters can be passed to the constructor to specify a particular hash function or comparer to use. That can be done with method (3), which on the other hand requires some redundant type specifications when creating a multidictionary object.

11.12.5 An attempt to avoid slow value collections

Note that some multidictionary operations will be rather slow if one instantiates VC with a collection class, such as LinkedList<V>, whose Contains operation is slow. If desirable, one can use the static constructor reject construction of types whose ContainsSpeed (see page 42) is not Constant or Log; see section 3.3:

```
public class MultiHashDictionary<K,V,VC> : HashDictionary<K, VC>
  where VC : ICollection<V>, new()
{
  static MultiHashDictionary() {
    Speed speed = new VC().ContainsSpeed;
    if (speed != Speed.Constant && speed != Speed.Log)
      throw new ArgumentException("Attempt to use slow value collection");

  }
  ...
}
```

It is not obvious that this is a desirable improvement. First, a fast ContainsSpeed does not guarantee fast insertion or deletion; consider ArrayList<T>. Secondly, the user of the multidictionary may know that there are never more than 20 items in each value collection, so that linear-time operations are perfectly acceptable.

11.13 Common words in a text file

Jon Bentley proposed the task of finding the k most common words in a text file and printing them (along with their frequencies) in order of decreasing frequency. Don Knuth presented his solution as a programming pearl in *Communications of the ACM* in 1986 [5]. Below is a much shorter and clearer solution using several features of C5. Of course, it is shorter and clearer chiefly because it exploits a pre-existing library and is less concerned with saving memory than was required two decades ago.

Method PrintMostCommon(maxWords, filename) reads words from the text file called filename, and then prints the maxWords most common of these words, along with the frequency of each word:

```
static void PrintMostCommon(int maxWords, String filename) {
  ICollection<String> wordbag = new HashBag<String>();
  Regex delim = new Regex("[^a-zA-Z0-9]+");
  using (TextReader rd = new StreamReader(filename)) {
    for (String line = rd.ReadLine(); line != null; line = rd.ReadLine())
      foreach (String s in delim.Split(line))
        if (s != "")
          wordbag.Add(s);
  }
```

```
    KeyValuePair<String,int>[] frequency
      = wordbag.ItemMultiplicities().ToArray();
    Sorting.IntroSort(frequency, 0, frequency.Length, new FreqOrder());
    int stop = Math.Min(frequency.Length, maxWords);
    for (int i=0; i<stop; i++) {
      KeyValuePair<String,int> p = frequency[i];
      Console.WriteLine("{0,4} occurrences of {1}", p.Value, p.Key);
    }
  }
```

The method builds a hashbag `wordbag` of all the words in the file, creates a dictionary that maps each word to the number of times it appears, creates an array `frequency` of (word, frequency) pairs from the dictionary, sorts the array, and prints the (at most) `maxWords` words with the highest frequency.

Sorting the array orders the (word, frequency) pairs lexicographically, first by decreasing frequency, then by increasing order of the words. The required lexicographic comparer can be implemented by a private nested class `FreqOrder` as follows:

```
class FreqOrder : SCG.IComparer<KeyValuePair<String,int>> {
  public int Compare(KeyValuePair<String,int> p1,
                     KeyValuePair<String,int> p2) {
    int major = p2.Value.CompareTo(p1.Value);
    return major != 0 ? major : p1.Key.CompareTo(p2.Key);
  }
}
```

Alternatively, one can can build the comparer inline from a suitable comparison delegate:

```
Sorting.IntroSort(frequency, 0, frequency.Length,
    new DelegateComparer<KeyValuePair<String,int>>
                (delegate(KeyValuePair<String,int> p1,
                          KeyValuePair<String,int> p2)
                 {
                   int major = p2.Value.CompareTo(p1.Value);
                   return major != 0 ? major : p1.Key.CompareTo(p2.Key);
                 }));
```

Not much elegance is achieved by inlining the comparer, though, due to the heavy syntax. Especially the verbose types on the constructor and the delegate parameters are heavy; more type inference would be welcome.

Chapter 12

Performance details

This chapter discusses the theoretical and practical performance of the collection operations and dictionary operations implemented by C5.

12.1 Performance of collection implementations

For each method and implementing collection class, the tables in figures 12.1, 12.2, 12.3, and 12.4 show the *time consumption* or running time of the method on an object of that class.

In the tables, n is the number of items in the given collection, m is the number of items in a collection or enumerable argument given to the operation, and r is the number of items affected (for instance, deleted) by an operation. For indexed operations, i is an integer index, and for list operations $|i|$ denotes the distance from an index i to the nearest end of the given list, that is, $\min(i,n-i)$. For instance, the `Insert(i)` line in figure 12.3 shows that adding an item to a LinkedList<T> is fast near either end of the list, where $|i|$ is small, but for an ArrayList<T> it is fast only near the back end, where $n-i$ is small.

The subscript a indicates *amortized complexity*: over a long sequence of operations, the average time per operation is $O(1)$, although any single operation could take time $O(n)$.

The subscript e indicates *expected complexity*: execution time is not a guaranteed upper bound, but an average over all possible (prior) input sequences. In practice, the expected complexity is the one you will experience in an application. However, there are some — very rare — input sequences for which the actual running time is higher. In particular, for most operations on hash-based collections and dictionaries, the time given is the expected complexity. When auxiliary hashsets are used in other collection operations, the time complexity will be expected complexity also, as indicated in the tables.

For the purpose of these tables, all user-supplied hash functions, comparers and delegates are assumed to be constant-time operations, that is, to take time $O(1)$.

Member	HashSet<T>	HashBag<T>	ArrayList<T>	LinkedList<T>	HashedArrayList<T>	HashedLinkedList<T>	TreeSet<T>	TreeBag<T>	SortedArray<T>	IntervalHeap<T>	CircularQueue<T>
`GetEnumerator()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`Backwards()`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	$O(1)$
`Direction`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	$O(1)$
`ActiveEvents`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`All(p)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Apply(act)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Choose()`	$O(1)_e$	$O(1)_e$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`CopyTo(arr,i)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Count`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`CountSpeed`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`Exists(p)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Filter(p)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Find(p, out res)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`IsEmpty`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`ListenableEvents`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`ToArray()`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`Backwards()`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	$O(1)$
`FindLast(p, out res)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	$O(n)$
`Add(x)`	$O(1)_{ae}$	$O(1)_{ae}$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(\log n)$	–
`AddAll(xs)`	$O(m)_{ae}$	$O(m)_{ae}$	$O(m)$	$O(m)$	$O(m)$	$O(m)$	$O(m \log n)$	$O(m \log n)$	$O(m \log m + n)$	$O(m \log n)$	–
`AllowsDuplicates`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–
`DuplicatesByCounting`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–
`EqualityComparer`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–
`IsReadOnly`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–

Figure 12.1: Performance of operations described by SCG.IEnumerable<T>, IDirectedEnumerable<T>, ICollectionValue<T>, IDirectedCollectionValue<T>, IExtensible<T>.

Member	HashSet<T>	HashBag<T>	ArrayList<T>	LinkedList<T>	HashedArrayList<T>	HashedLinkedList<T>	TreeSet<T>	TreeBag<T>	SortedArray<T>	IntervalHeap<T>	CircularQueue<T>
`Clear()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–
`Contains(x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`ContainsAll(xs)`	$O(m)$	$O(m)$	$O(m+n)$	$O(m+n)$	$O(m)$	$O(m)$	$O(m\log n)$	$O(m\log n)$	$O(m\log n)$	–	–
`ContainsCount(x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`ContainsSpeed`	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–
`Find(ref x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`FindOrAdd(ref x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`GetUnsequencedHashCode()`	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	–	–
`ItemMultiplicities()`	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–
`Remove(x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(n)$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`Remove(x, out y)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(n)$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RemoveAll(xs)`	$O(m)_e$	$O(m)_e$	$O(m+n)_e$	$O(m+n)_e$	$O(m+n)_e$	$O(m)_e$	$O(m\log n)$	$O(m\log n)$	$O(m\log n)$	–	–
`RemoveAllCopies(x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(n)$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RetainAll(xs)`	$O(m)_{ae}$	$O(m)_{ae}$	$O(m+n)_e$	$O(m+n)_e$	$O(m+n)_e$	$O(m)_e$	$O(m\log n)$	$O(m\log n)$	$O(m\log n)$	–	–
`UniqueItems()`	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–
`UnsequencedEquals(coll)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–
`Update(x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`Update(x, out y)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`UpdateOrAdd(x)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`UpdateOrAdd(x, out y)`	$O(1)_e$	$O(1)_e$	$O(n)$	$O(n)$	$O(1)_e$	$O(1)_e$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`GetSequencedHashCode()`	–	–	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	$O(1)_a$	–	–
`SequencedEquals(coll)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–
`FindIndex(p)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–
`FindLastIndex(p)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–
`IndexingSpeed`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–
`IndexOf(x)`	–	–	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`LastIndexOf(x)`	–	–	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RemoveAt(i)`	–	–	$O(n-i)$	$O(\|i\|)$	$O(n-i)$	$O(\|i\|)$	$O(\log n)$	$O(\log n)$	$O(n)$	–	–
`RemoveInterval(i,m)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	–	–
`this[i]`	–	–	$O(1)$	$O(\|i\|)$	$O(1)$	$O(\|i\|)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`this[i,m]`	–	–	$O(1)$	$O(i+m)$	$O(1)$	$O(i+m)$	$O(\log n)$	$O(\log n)$	$O(1)$	–	–

Figure 12.2: Performance of operations described by ICollection<T>, ISequenced<T>, and IIndexed<T>.

Member	HashSet<T>	HashBag<T>	ArrayList<T>	LinkedList<T>	HashedArrayList<T>	HashedLinkedList<T>	TreeSet<T>	TreeBag<T>	SortedArray<T>	IntervalHeap<T>	CircularQueue<T>
`AllowsDuplicates`	–	–	$O(1)$	$O(1)$	–	–	–	–	–	–	$O(1)$
`Dequeue()`	–	–	$O(n)$	$O(1)$	–	–	–	–	–	–	$O(1)$
`Enqueue(x)`	–	–	$O(1)$	$O(1)$	–	–	–	–	–	–	$O(1)$
`this[i]`	–	–	$O(1)$	$O(\|i\|)$	–	–	–	–	–	–	$O(1)$
`AllowsDuplicates`	–	–	$O(1)$	$O(1)$	–	–	–	–	–	–	$O(1)$
`Pop()`	–	–	$O(1)$	$O(1)$	–	–	–	–	–	–	$O(1)$
`Push(x)`	–	–	$O(1)$	$O(1)$	–	–	–	–	–	–	$O(1)$
`this[i]`	–	–	$O(1)$	$O(\|i\|)$	–	–	–	–	–	–	$O(1)$
`FIFO`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–	–	–	–
`First`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–	–	–	–
`FindAll(p)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`Insert(i,x)`	–	–	$O(n-i)$	$O(\|i\|)$	$O(n-i)$	$O(\|i\|)$	–	–	–	–	–
`Insert(w,x)`	–	–	$O(n-i)$*	$O(1)$	$O(n-i)$*	$O(1)$	–	–	–	–	–
`InsertAll(i,xs)`	–	–	$O(m+n)$	$O(m+n)$	$O(m+n)$	$O(m+n)$	–	–	–	–	–
`InsertFirst(x)`	–	–	$O(n)$	$O(1)$	$O(n)$	$O(1)$	–	–	–	–	–
`InsertLast(x)`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–	–	–	–
`IsSorted()`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`IsSorted(cmp)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`IsValid`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–	–	–	–
`Last`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–	–	–	–
`LastViewOf(x)`	–	–	$O(n)$	$O(n)$	$O(1)$	$O(1)$	–	–	–	–	–
`Map(f)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`Map(f, eqc)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`Offset`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)_a$	–	–	–	–	–
`Remove()`	–	–	$O(1)$$	$O(1)$	$O(1)$$	$O(1)$	–	–	–	–	–
`RemoveFirst()`	–	–	$O(n)$	$O(1)$	$O(n)$	$O(1)$	–	–	–	–	–
`RemoveLast()`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–	–	–	–
`Reverse()`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`Shuffle()`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`Shuffle(rnd)`	–	–	$O(n)$	$O(n)$	$O(n)$	$O(n)$	–	–	–	–	–
`Slide(d)`	–	–	$O(1)$	$O(d)$	$O(1)$	$O(d)$	–	–	–	–	–
`Slide(d,m)`	–	–	$O(1)$	$O(d+m)$	$O(1)$	$O(d+m)$	–	–	–	–	–
`Sort()`	–	–	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	–	–	–	–	–
`Sort(cmp)`	–	–	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	–	–	–	–	–
`Span(w)`	–	–	$O(1)$	$O(1)$†	$O(1)$	$O(n)$	–	–	–	–	–
`this[i]`	–	–	$O(1)$	$O(\|i\|)$	$O(1)$	$O(\|i\|)$	–	–	–	–	–
`TrySlide(d)`	–	–	$O(1)$	$O(d)$	$O(1)$	$O(d)$	–	–	–	–	–
`TrySlide(d,m)`	–	–	$O(1)$	$O(d+m)$	$O(1)$	$O(d+m)$	–	–	–	–	–
`Underlying`	–	–	$O(1)$	$O(1)$	$O(1)$	$O(1)$	–	–	–	–	–
`View(i,m)`	–	–	$O(1)$	$O(m)$	$O(1)$	$O(m)$	–	–	–	–	–
`ViewOf(x)`	–	–	$O(n)$	$O(n)$	$O(1)$	$O(1)$	–	–	–	–	–

Figure 12.3: Performance of operations described by IQueue<T>, IStack<T>, and IList<T>. Notes: ($) this becomes $O(n)$ if property `FIFO` is changed from its default; (*) `i` is the offset of `w`'s right endpoint; (†) actually $O(n)$ in C5 release 0.9.

Member	HashSet<T>	HashBag<T>	ArrayList<T>	LinkedList<T>	HashedArrayList<T>	HashedLinkedList<T>	TreeSet<T>	TreeBag<T>	SortedArray<T>	IntervalHeap<T>	CircularQueue<T>
`AddSorted(xs)`	–	–	–	–	–	–	$O(n\log n)$	$O(n\log n)$	$O(m+n)$	–	–
`Comparer`	–	–	–	–	–	–	$O(1)$	$O(1)$	$O(1)$	–	–
`Cut(cmp,,,,)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`DeleteMax()`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`DeleteMin()`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`FindMax()`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`FindMin()`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`Predecessor(x)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RangeAll()`	–	–	–	–	–	–	$O(1)$	$O(1)$	$O(1)$	–	–
`RangeFrom(x)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RangeFromTo(x,y)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RangeTo(y)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RemoveRangeFrom(x)`	–	–	–	–	–	–	$O(n\log n)*$	$O(n\log n)*$	$O(n)$	–	–
`RemoveRangeFromTo(x,y)`	–	–	–	–	–	–	$O(n\log n)*$	$O(n\log n)*$	$O(n)$	–	–
`RemoveRangeTo(y)`	–	–	–	–	–	–	$O(n\log n)*$	$O(n\log n)*$	$O(n)$	–	–
`Successor(x)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`WeakPredecessor(x)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`WeakSuccessor(x)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`CountFrom(x)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`CountFromTo(x,y)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`CountTo(y)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`FindAll(p)`	–	–	–	–	–	–	$O(n)$	$O(n)$	$O(n)$	–	–
`Map(f, cmp)`	–	–	–	–	–	–	$O(n)$	$O(n)$	$O(n)$	–	–
`RangeFrom(x)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RangeFromTo(x,y)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`RangeTo(y)`	–	–	–	–	–	–	$O(\log n)$	$O(\log n)$	$O(\log n)$	–	–
`Snapshot()`	–	–	–	–	–	–	$O(1)$	$O(1)$	–	–	–
`Add(ref h, x)`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–
`Comparer`	–	–	–	–	–	–	–	–	–	$O(1)$	–
`Delete(h)`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–
`DeleteMax()`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–
`DeleteMax(out h)`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–
`DeleteMin()`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–
`DeleteMin(out h)`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–
`Find(h, out x)`	–	–	–	–	–	–	–	–	–	$O(1)$	–
`FindMax()`	–	–	–	–	–	–	–	–	–	$O(1)$	–
`FindMax(out h)`	–	–	–	–	–	–	–	–	–	$O(1)$	–
`FindMin()`	–	–	–	–	–	–	–	–	–	$O(1)$	–
`FindMin(out h)`	–	–	–	–	–	–	–	–	–	$O(1)$	–
`Replace(h, x)`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–
`this[h] get`	–	–	–	–	–	–	–	–	–	$O(1)$	–
`this[h] set`	–	–	–	–	–	–	–	–	–	$O(\log n)$	–

Figure 12.4: Performance of operations described by ISorted<T>, IIndexedSorted<T>, IPersistentSorted, and IPriorityQueue<T>.

12.2 Performance of dictionary implementations

The tables below show the *time consumption* or running time of each method in the dictionary implementations. The table in figure 12.5 shows methods belonging to the interfaces ICollectionValue<KeyValuePair<K,V>> and IDictionary<K,V>, and the table in figure 12.6 shows the methods belonging to interface ISortedDictionary<K,V>.

Note that the HashDictionary<K,V> operations described by the ICollectionValue<> interface have the same asymptotic performance as those of HashSet<K>, and similarly the TreeDictionary<K,V> operations described by the ICollectionValue<> interface have the same asymptotic performance as those of TreeSet<K>.

Member	HashDictionary<K,V>	TreeDictionary<K,V>
`GetEnumerator`	$O(1)$	$O(1)$
`ActiveEvents`	$O(1)$	$O(1)$
`All(p)`	$O(n)$	$O(n)$
`Apply(act)`	$O(n)$	$O(n)$
`Choose()`	$O(1)_e$	$O(1)$
`CopyTo(arr,i)`	$O(n)$	$O(n)$
`Count`	$O(1)$	$O(1)$
`CountSpeed`	$O(1)$	$O(1)$
`Exists(p)`	$O(n)$	$O(n)$
`Filter(p)`	$O(n)$	$O(n)$
`Find(p, out res)`	$O(n)$	$O(n)$
`IsEmpty`	$O(1)$	$O(1)$
`ListenableEvents`	$O(1)$	$O(1)$
`ToArray()`	$O(n)$	$O(n)$
`Add(k, v)`	$O(1)_e$	$O(\log n)$
`AddAll(kvs)`	$O(m)_e$	$O(m \log n)$
`Clear()`	$O(1)$	$O(1)$
`Contains(k)`	$O(1)_e$	$O(\log n)$
`ContainsAll(ks)`	$O(m)_e$	$O(m \log n)$
`ContainsSpeed`	$O(1)$	$O(1)$
`Count`	$O(1)$	$O(1)$
`EqualityComparer`	$O(1)$	$O(1)$
`Find(k, out v)`	$O(1)_e$	$O(\log n)$
`Find(ref k, out v)`	$O(1)_e$	$O(\log n)$
`FindOrAdd(k, out v)`	$O(1)_e$	$O(\log n)$
`IsReadOnly`	$O(1)$	$O(1)$
`Keys`	$O(1)$	$O(1)$
`Remove(k)`	$O(1)_e$	$O(\log n)$
`Remove(k, out v)`	$O(1)_e$	$O(\log n)$
`this[k]`	$O(1)_e$	$O(\log n)$
`Update(k, v)`	$O(1)_e$	$O(\log n)$
`Update(k, v, out vOld)`	$O(1)_e$	$O(\log n)$
`UpdateOrAdd(k, v)`	$O(1)_e$	$O(\log n)$
`UpdateOrAdd(k, v, out vOld)`	$O(1)_e$	$O(\log n)$
`Values`	$O(1)$	$O(1)$

Figure 12.5: Performance of dictionary operations described by interfaces SCG.IEnumerable<KeyValuePair<K,V>>, ICollectionValue<KeyValuePair<K,V>> and IDictionary<K,V>.

Member	HashDictionary<K,V>	TreeDictionary<K,V>
`AddSorted(kvs)`	–	$O(m \log n)$
`Comparer`	–	$O(1)$
`Cut(cmp,,,,)`	–	$O(\log n)$
`DeleteMax()`	–	$O(\log n)$
`DeleteMin()`	–	$O(\log n)$
`FindMax()`	–	$O(\log n)$
`FindMin()`	–	$O(\log n)$
`Keys`	–	$O(1)$
`Predecessor(k)`	–	$O(\log n)$
`RangeAll()`	–	$O(\log n)$
`RangeFrom(k1)`	–	$O(\log n)$
`RangeFromTo(k1, k2)`	–	$O(\log n)$
`RangeTo(k2)`	–	$O(\log n)$
`RemoveRangeFrom(k1)`	–	$O(r \log n)$
`RemoveRangeFromTo(k1, k2)`	–	$O(r \log n)$
`RemoveRangeTo(k2)`	–	$O(r \log n)$
`Successor(k)`	–	$O(\log n)$
`WeakPredecessor(k)`	–	$O(\log n)$
`WeakSuccessor(k)`	–	$O(\log n)$

Figure 12.6: Performance of dictionary operations in ISortedDictionary<K,V>.

12.3 Performance of quicksort and merge sort

The sorting algorithm for array lists is *introspective quicksort*, whose implementation is described in section 13.2. It has the following performance properties:

- It is as fast as quicksort on random data.
- It is guaranteed fast: The worst-case running time is $O(n \log n)$, and therefore much faster than plain quicksort on bad data sets.
- It requires only $O(\log n)$ extra space (in addition to the array list).

The sorting algorithm for linked lists is a *stable in-place merge sort*, whose implementation is described in section 13.3. It has the following performance properties:

- It is guaranteed fast: The worst-case running time is $O(n \log n)$. In practice, it is slower than quicksort for arrays by a factor of two or so.
- It requires no extra space (in addition to the linked list).

12.4 Performance impact of list views

- Update overhead: Updates to a list may affect the `Offset` or `Count` of views on the list. To implement this, every update must check for affected views, so there is a runtime overhead that is proportional to the number of valid views on the list; see section 13.9. For this reason, it is important to invalidate views on lists as early as possible; see below.

- Space leaks: A range view of a sorted collection or a sorted dictionary contains a reference to the underlying collection or dictionary. This may cause a so-called space leak if a small range view refers to a large underlying collection. So long as the small range view is kept alive by the program, the underlying collection is kept alive as well, and the space it occupies cannot be reclaimed by the garbage collector. To avoid this, do not keep views alive unnecessarily.
- When a view is invalidated, it stops holding on to the underlying list, and no longer affects the execution time of updates. Invalidation of a view `u` can be requested by calling `u.Dispose()`. If a view variable `u` is allocated by C#'s `using` statement, then `u.Dispose()` is called automatically when the variable goes out of scope; see pattern 24.

12.5 Performance impact of event handlers

If each call to an event handler executes in constant time, then event handlers will not change the asymptotic running time of collection operations. That is, an $O(1)$ operation does not suddenly become an $O(n)$ operation just because an event handler has been attached to the collection on which the operation is performed.

This is a conscious design decision, one consequence of which is that the `Clear` method on a list view does not raise an ItemsRemoved event for each item removed. Namely, without event handlers, `list.View(i,n).Clear()` can be performed in constant time when `list` is a linked list, but if an ItemsRemoved event must be performed for each item, then it would become an $O(\mathtt{n})$ operation.

As described in section 13.12, the implementation of events is designed to minimize the runtime overhead in the frequent case where no event handlers are associated with a collection.

12.6 Performance impact of tree snapshots

Creating a tree snapshot itself takes constant time, independent of the size of the tree. Each modification to the original tree may take extra time and space as long as the snapshot is alive. For any single update to the original tree, the time and space overhead may be $O(\log n)$, where n is the size of the tree, but the amortized run-time and space cost is $O(1)$ per update to the original tree. Over a long sequence of updates, there is just a constant amount of extra work per update.

Once the snapshot has been deallocated by the garbage collector, further updates to the tree does not incur extra time or space costs. Therefore snapshots should preferably be allocated in a `using` statement so that they get released eagerly; see section 9.11.

In the worst case, where a snapshot is kept alive indefinitely and the original tree is updated aggressively, the snapshot will end up being a clone of the entire original tree, after which further updates to the original tree will incur no further overhead. See section 13.10 for more details on the implementation.

Chapter 13

Implementation details

13.1 Organization of source files

File or directory	Contents
`C5/AssemblyInfo.cs`	Version number and similar
`C5/BuiltIn.cs`	IntComparer, DoubleComparer, ...
`C5/Collections.cs`	Collection base classes (see chapter 14)
`C5/Comparer.cs`	Comparer<T>, NaturalComparer<T>, ...
`C5/Delegate.cs`	Delegates types Act<A1>, ..., Fun<A1,R>, ...
`C5/Dictionaries.cs`	Dictionary base classes (see chapter 14)
`C5/Enums.cs`	Enum types Speed, EnumerationDirection, ...
`C5/Events.cs`	Event handler types and event raising
`C5/Exceptions.cs`	Exception classes
`C5/Formatting.cs`	IShowable and formatting
`C5/Hashers.cs`	EqualityComparer<T>, ...
`C5/Interfaces.cs`	Collection and dictionary interfaces
`C5/Random.cs`	C5Random
`C5/Records.cs`	Record struct types Rec<T1,T2>, ...
`C5/Sorting.cs`	Introspective quicksort, heapsort, ...
`C5/WrappedArray.cs`	WrappedArray<T>
`C5/Wrappers.cs`	Guarded collections and dictionaries
`C5/arrays/ArrayList.cs`	Arraylist<T> and ⋆HashedArrayList<T>
`C5/arrays/CircularQueue.cs`	CircularQueue<T>
`C5/arrays/SortedArray.cs`	SortedArray<T>
`C5/hashing/HashBag.cs`	HashBag<T>
`C5/hashing/HashDictionary.cs`	HashDictionary<K,V>
`C5/hashing/HashTable.cs`	HashSet<T>
`C5/heaps/IntervalHeap.cs`	IntervalHeap<T>
`C5/linkedlist/LinkedList.cs`	LinkedList<T> and ⋆HashedLinkedList<T>
`C5/trees/RedBlackTreeDictionary.cs`	TreeDictionary<K,V>
`C5/trees/RedBlackTreeSet.cs`	TreeSet<T> and ⋆TreeBag<T>

The files above are found in the source distribution `C5.src.zip`. The source file for a class marked with an asterisk (⋆) is generated by a preprocessing tool found in file `PreProcess/Program.cs`. The overall structure is the source distribution is this:

File or directory	Contents
`BUILD.txt`	Instructions for building from source
`LICENSE.txt`	Library license, reproduced on page 3
`RELEASE-NOTES.txt`	List of changes in current release
`C5/`	Library source code; see above
`docNet/`	Files to build online documentation
`nunit/`	Files to build unit tests, to be run with NUnit [1].
`PreProcess/`	Preprocessor to build source for classes marked (⋆) above
`UserGuideExamples/`	Source for examples in chapter 11

The C5 library can be built from source on Microsoft's .NET 2.0 as well as Novell's Mono implementation. File `BUILD.txt` in the source distribution contains instructions for building the library, online documentation and unit tests.

13.2 Implementation of quicksort for array lists

This sorting algorithm works on array lists and is used in the implementation of the `Sort` methods in class ArrayList<T> and the static `IntroSort` methods in class Sorting. The algorithm has the following properties:

- It is as fast as quicksort on random data.
- It is guaranteed fast: The worst-case running time is $O(n \log n)$, and therefore much faster than quicksort on bad data sets.
- It requires only $O(\log n)$ extra space in addition to the array being sorted.
- It is not stable: two items in the given array that compare equal may be swapped in the sorted result. For a stable sorting algoritm, use merge sort on linked lists (section 13.3).

The algorithm is due to Musser [23]. It is basically a quicksort that keeps track of its own recursion depth. Then it switches to using heap sort when the recursion depth becomes "too large", which is a sign that the partitioning performed by the quicksort has repeatedly turned out bad.

13.3 Implementation of merge sort for linked lists

This sorting algorithm works on linked lists and is used in the implementation of the `Sort` methods in class LinkedList<T>. The algorithm has the following properties:

- It is stable: The order of equal items in the given list is preserved in the sorted list. This permits multi-key (lexicographic) sorting to be performed in several passes, starting with the least significant key in the lexicographic ordering and ending with the most significant key.

- It is in-place: It requires no extra space besides the linked list's nodes.
- It is guaranteed fast: The worst-case running time is $O(n \log n)$.

In practice, merge sort is two to three times slower than introspective quicksort (section 13.2). Our in-place merge sort algorithm builds an outer list of runs, where a run is an inner list of nodes whose items appear in (weakly) increasing order. Then the runs are repeatedly merged pairwise in order of appearance, creating longer and longer runs, until all nodes make up a single run: the sorted list.

The inner lists and the outer lists are singly-linked, because they need to be traversed in only one direction. The nodes of each inner list are linked together using the next-node reference of the participating LinkedList<T> nodes, terminating with a `null` next-node reference. The nodes of the outer list (which is a list of inner lists) are linked together using the previous-node reference in the first node of each inner list. Hence no extra space is required.

13.4 Implementation of hash-based collections

For experimental purposes the source code contains several variations of linear hashing, namely linear probing and linear chaining [9] in combination with both reference type bucket cells and value type bucket cells. The combination actually used is determined by preprocessor flags `LINEARPROBING`, which determines whether the code uses linear probing or linear chaining, and `REFBUCKET` which determines whether the first bucket is of reference type (pointed to from the collection's array) or value type (stored directly in that array).

The choice of trying these variations of linear hashing was based on the findings of Pagh and Rodler [24]. All implementations have the usual theoretical complexities: $O(1)$ expected time for lookups and $O(1)$ expected amortized time for update operations. There are other linear hashing strategies such as double hashing, but we have not implemented those.

The current compilation default is to use linear chaining and reference type buckets; over a range of experiments this seems to offer the best performance. A hash-based collection is therefore implemented by an array, each item of which contains `null` or a reference to a bucket. With linear chaining, a bucket contains an item, the item's cached hash code, and a reference that is `null` or refers to an overflow bucket.

The array size always is a power of two, and the index for an item (bucket) is computed from the hash value using a universal hash function family mentioned by Pagh and Rodler [24]:

```
index = (k * hashval) % tablesize
```

Here `k` is constant over at least the lifetime of a specific array. The preprocessor flags `INTERHASHER` and `RANDOMINTERHASHER` determine whether `k` is a large odd compile time constant or a random odd integer. The last choice should provide the good, randomized expected complexity asymptotics.

13.5 Implementation of array lists

An array list consists of an underlying extensible array and an indication of the number of items actually in the array list.

A view of an array list consists of a reference to its underlying array list, an offset relative to that array list, an item count, and an indication whether the view is valid. A view and its underlying array list have the same underlying extensible array.

A proper array list (not a view) has a list all its views, so that it can update any affected views whenever the array list is updated. For instance, insertions and deletions may affect the offset and the count of a view, and sorting, shuffling and subrange reversal may invalidate a view. The list contains weak references to the views, so that the views are not kept alive longer than necessary.

13.6 Implementation of hashed array lists

Class HashedArrayList<T> is implemented as an arraylist that additionally has a hash dictionary that maps an item to the unique index of that item in the list, if any. The hash dictionary is shared between the hashed arraylist and all views of it.

Still, `view.Contains(x)`, where `view` is a view on list `list`, can be implemented in expected constant time, as follows. First the index of `x` is looked up in the hash dictionary, and if it is there, then it is checked that the index is between the endpoints of `view`, using two integer comparisons.

13.7 Implementation of linked lists

A linked list is implemented as a doubly linked list of nodes, each node holding an item. A doubly linked list is preferred over a singly linked one because it permits efficient insertion and deletion at either end, and efficient insertion and deletion at views inside the list.

There is an artificial sentinel node at each end of the double linked list, and a LinkedList<T> object always has non-`null` references to these sentinel nodes. This significantly reduces the number of cases that must be considered in operations.

A view of a linked list consists of a reference to its underlying linked list, references to start and end sentinel nodes (which are simply nodes in the underlying list), an offset relative to the underlying list, an item count, and an indication whether the view is valid. All nodes of a view, including the sentinel nodes, belong also to the underlying linked list.

A proper linked list (not a view) further has a list all its views, so that it can update any affected views whenever the linked list is updated. For instance, insertions and deletions may affect the offset and the count of a view, and sorting, shuffling and subrange reversal may invalidate a view. The list uses weak references to the views, so that views are not kept alive longer than necessary.

13.8 Implementation of hashed linked lists

Class HashedLinkedList<T> is a linked list that additionally has a hash dictionary that maps an item to the unique list node containing that item, if any. The hash dictionary is shared between the hashed linked list and all views of it.

Obtaining the best asymptotic performance for the combination of list views and hash indexing is rather challenging. Namely, if `view` is a view on a list `list`, then `view.Contains(x)` should be fast and should return false if `x` is in the underlying list but not inside the view. Therefore, when the hash dictionary maps `x` to a list node we must be able to decide quickly whether that node is inside the view. For array lists this was easy: simply check whether the node's index is between the endpoints of the view, using integer comparisons. For linked lists, computing the index of `x`'s node, or traversing the view to see whether `x`'s node falls within the view would be slow and would completely defeat the purpose of the hash dictionary.

Our solution uses a list-ordering algorithm of Sleator and Dietz [29] in the version described by Bender et al. [4].

The basic idea is simple. We give each node an integer tag and make sure the tags are always in increasing order. First, whenever we insert a node in the list we give it a tag that is between its neighbors, say the mean of the two. If there is not enough room to make the tags distinct, we redistribute the tags of part of the list to make room. The only problem is how to choose the amount of renumbering to balance with the frequency of renumbering. The answer of Bender et al. [4] is the following. When we run out of room we follow the list both ways comparing binary prefixes of tags, counting the segments of the list that have common prefixes of length $w-1, w-2, \ldots, w-b$ where w is the word size (here 32). Let k be a parameter to be defined. For the least b where the segment is of size at most k^b, we redistribute the tags in that segment uniformly. We need $n \geq k^w$ to be sure such a b is found, and $k < 2$ to be sure there are sizable gaps after tag redistribution, so we choose $k = \sqrt[w]{n} < 2$.

The cost of this redistribution is $O(k^b)$. Note that before this redistribution the segment of common prefix length $w-(b-1)$ had at least k^{b-1} nodes, and after redistribution it has $k^b/2$. Thus there must be at least $k^{b-1} - k^b/2 = k^b(2-k)/(2k)$ insertions before we need to redistribute this segment again, and the amortized cost of redistributions of tags at a given b is $O(k/(2-k))$ per insertion into the list. As long as n is not close to 2^w, this is $O(1)$ per distinct value of b and hence the total amortized cost of redistributions of tags will be $O(w)$ per insertion into the list. Note, however, that the typical case of always inserting at the end of the list seems to be close to the worst-case scenario for maintaining the tags.

The tagging scheme used for hashed linked lists is a further refinement of this basic scheme. The reason is that although the basic scheme is simple and has low overhead on those operations that do not cause tag redistributions, in the worst case all tags must be redistributed and the complete list traversed, which is bad for locality of memory references.

To get to $O(1)$ amortized cost per list update of tag maintenance we use instead a two-level grouping scheme: Use the above basic scheme to maintain tags on groups of consecutive list nodes, and use a second tag inside each group. If the number of groups are kept at $O(n/w)$, say by making at least every other group be of size $\Omega(w)$, then the amortized cost of maintaining the top level tags will be $O(1)$. It is not hard to see that putting an extra $O(1)$ charge on insertions will pay for splitting and renumbering groups when we miss room in the low level tags; and an extra charge on deletions will pay for joining groups when a group that gets below the $\Omega(w)$ threshold has a neighbor also below that threshold. In total we get an $O(1)$ amortized cost per list update of tag maintenance at the cost of a much more complicated algorithm. Performance measurements indicated that this scheme nevertheless performs better at large list sizes, so this refined two-level grouping scheme is the one used in the library.

Note that HashedLinkedList<T> is not a subclass of LinkedList<T>, as that would require linked list nodes to carry fields used only in hashed linked list, incurring a considerable space overhead.

13.9 Implementation of list views

The implementation of views of array lists and of linked lists are described in sections 13.5 and 13.7 above. Here we describe how the `Offset` and `Count` properties of a view are maintained when the underlying list is updated, possibly through one of its views.

For this purpose, a proper list maintains a list of its views, in order of creation, using weak references so as not to keep the views live unnecessarily. Adding a newly created view to this list, or removing the view when disposed, takes constant time.

For ArrayList<T>, HashedArrayList<T> and LinkedList<T>, maintenance in connection with an update is done by going through the list of all live views, and determining from the each view's `Offset` and `Count` whether it is affected by the update, taking into account also whether the update was made through the view itself; see section 8.1.5. This is done after insertions and before removals.

For HashedLinkedList<T>, the `Offset` field of a view is not necessarily known. However, the tagging scheme described in section 13.8 can be used to determine in constant time whether one node precedes another, and hence to determine for a view whether it is affected by an update.

13.10 Implementation of tree-based collections

The tree-based collections TreeSet<T>, TreeBag<T> and TreeDictionary<K,V> are implemented as red-black trees, following Tarjan [31]. The operations are done in bottom-up style and there are no parent pointers in the tree nodes.

In contrast to tree-based collections from the standard libraries of C#, Java or Smalltalk, ours have support for persistence, using techniques from Driscoll et al. [10]. Namely, the `Snapshot` method creates a read-only "snapshot" of a tree in constant time, at the expense of making subsequent updates to the original tree somewhat slower.

Tree snapshots can be implemented in at least two ways, namely using *node copy persistence* or *path copy persistence*. To illustrate the difference, assume we have a tree, make a (read-only) snapshot of it, and then insert a new node in the tree.

- With path copy persistence, all nodes on the path from the root to the inserted node would be copied. Thus only completely unchanged subtrees are shared between a snapshot and the updated tree. This scheme is simple and has the advantage that each node can keep an accurate node count for the subtree of which it is a root. The main drawback is the aggressive copying of nodes; an update to the tree will on average allocate a number of new nodes that is logarithmic in the collection size.

- With node copy persistence, each node can have up to two left children or up to two right children: an old one (`oldref`, see below) belonging to a snapshot, and a new one belonging to the updated tree. Each node is created with an extra child reference, initially unused. At an insertion into the tree, one must consider the nodes on the path to the inserted node. A node that has an unused child reference need not be copied; one only needs to copy a node if it extra child reference is already in use. Thus a snapshot and the updated tree may share a node even though its children have been updated. The main advantage of this is efficiency: the amortized number of new nodes created per update is constant. The main disadvantages are that the scheme is more complicated, and that one cannot easily maintain subtree node counts in a snapshot. For that reason, snapshots do not support efficient item access by index, and the type of a snapshot is ISorted<T> even though the original collection was an IIndexedSorted<T>.

We chose node copy persistence for the C5 library after extensive performance studies [18]. Our implementation is based on ideas from Driscoll et al. [10]. Remember that we implement partial persistence; snapshots are read-only historic documents, not mutable clones.

To support persistence, a tree collection has the following additional fields:

- `bool isSnapShot` is false for a proper tree collection, true for snapshots.

- `int generation` is initially 0, and is incremented every time a snapshot is made from the tree. For a snapshot, this always equals the tree's `generation` at the time the snapshot was made; a snapshot cannot be made from a snapshot.

- `SnapRef snapList` for a tree registers all the tree's non-disposed snapshots, using weak references. For a snapshort, `snapList` refers to the snapshot's entry in the underlying tree's `snapList` so the snapshot can be disposed in constant time.

- `int maxsnapid` is the maximal generation number of any non-disposed snapshot of this tree, or -1 if there are no snapshots, so `snapList` is empty.

As in a standard red-black tree, a node has a color (red or black), an item, and optional left and right child node references. To support persistence, each node has the following additional fields:

- `int generation`, initialized from the tree's `generation` number when the node is created.
- `int lastgeneration`, initially -1, but updated to the current `maxsnapid` if the node is snapshotted and subsequently updated. That is, this is the generation of the most recent snapshot to which the node belongs.
- `Node oldref`, initially `null`, but non-`null` if the node has been snapshotted and subsequently updated.
- `bool leftnode` indicates whether the `oldref`, if non-`null`, refers to an overwritten left node (true) or right node (false).

A snapshot object is a shallow clone of a tree object (but not its nodes) having the tree's old generation number, and with `isSnapShot` set to true. If more than 2^{31} snapshots are made from a tree, the 32 bit signed generation number may wrap around, which will not be handled gracefully by the library.

We will now describe how the additional node fields are used for defining a specific generation snapshot, that is, how to enumerate a specific generation of the tree. Just after a node has been created, its generation number equals that of the tree, and such a node will never be encountered during enumeration of an older snapshot. The item of a node encountered during enumeration of a snapshot will always be valid, but the node's red-black color field is not necessarily valid for the snapshot; in fact, the color is only needed for updates, which do not happen for snapshots. If a node with `generation` equal to g_0, and `lastgeneration` equal to g_1 is encountered when enumerating a younger snapshot of generation $g_2 > g_1$, then both child references in the node are valid. If the snapshot being enumerated is older, so $g_0 \leq g_2 \leq g_1$, then one of the child references in the node is not valid for that snapshot; instead the valid child reference is in the `oldref` field, and the `leftnode` field tells which of the child references is held in `oldref`.

Now we will explain how updates are affected by node copy persistence, assuming that the reader is familiar with updates to non-persistent red-black trees. Assume we are about to update a node that belongs to a live snapshot, so its `generation` field is less than or equal to the `maxsnapid` field of the tree. If the update is a color change, just do it. If the update is a change of the item, make a copy of the node, update the item in the new node and update the reference in the parent node to point to the new node; this node will have the generation number of the tree we are updating. If the update is a change of a child reference and the `lastgeneration` field of the node is still -1, we update `lastgeneration` to `maxsnapid`, set the `leftnode` field to true if it is the left child we are updating, copy the old child reference to the `oldref` field and finally update the live child reference. If, finally, we must update a child reference in a node that has already had made one child reference update, we copy the node to a new one with whose `generation` will be that of the tree, and then update the child pointer in the parent node. In the last situation, if the child reference we are updating is the child that was updated before and the result of the old update cannot belong to any live snapshots because the tree's `maxsnapid` is less than or equal to the `lastgeneration` of the node, we just update the child reference instead of copying the node.

It should be clear from this description that the procedure is correct and does not influence enumerations and lookups in the tree and snapshots by more than $O(1)$ work per node encountered. Thus a single item operation will still use time $O(\log n)$. We now explain the crucial fact that the procedure will only produce $O(1)$ amortized new nodes per update operation on the tree. Note that a single update operation on the tree can result in cascading node copies all the way to the root. But since the cascade will stop when we use an unfilled `oldref` slot or reach the root, the number of `oldref` filled by each update operation is $O(1)$. Moreover, a node will only be copied once: after the copy it will no longer be part of the live tree and no longer subject to updates. Now, if we have performed m update operations on the tree since its creation, there can be at most $O(m)$ nodes with filled `oldref` slots. A node with an unfilled `oldref` slot either has been created directly by an insert operation on the tree or is the single copy of a Node with a filled `oldref` slot. In total we have at most $O(m)$ nodes of either kind, which means exactly that we will only produce $O(1)$ amortized new nodes per update operation on the tree.

The tree's `maxsnapid` field that holds the newest live snapshot is maintained as follows. When a snapshot is created, it is registered in the `snapList` of the original tree, and its `maxsnapid` field is updated. The storage element in the `SnapRef` class is a red-black tree itself. When a snapshot is disposed, either by an explicit call to `Dispose()` or by the garbage collector calling the finalizer of the tree class, that snapshot will be deregistered for the `SnapRef` object and `maxsnapid` will be updated if relevant. In particular, `maxsnapid` may be reset to -1 if there are no live snapshots. This means that if we take a single snapshot, keep it alive for some time, and then dispose it, we may avoid copying the whole tree if we only make a few updates while the snapshot is alive.

It would be possible, but expensive, to make a more thorough cleanup of nodes that become unreachable when a snapshot is disposed. We do not consider that worthwhile, because the typical usage scenarios would be:

- Make a snapshot, enumerate it while doing updates, and then dispose the snapshot, as in section 9.11.

- Build a data structure where we need all the snapshots until we just forget the whole thing, as in the example in section 11.9.

Both scenarios are handled well by the current implementation.

13.11 Implementation of priority queue handles

In C5 a priority queue is implemented as an interval heap, which permits efficient access to both least and greatest item. The choice of this data structure over Min-Max heaps was based upon results [28] from the Copenhagen STL project [8]. Our implementation of interval heaps is special in that a *handle* (at most one) can be associated with an item in the heap, permitting the item to be efficiently retrieved, updated or deleted.

An interval heap is represented as an array of intervals, where an interval is a pair of a first item a and a last item b, with $a \leq b$. The children (if any) of an interval at array index i are at array indexes $2i$ and $2i+1$. The interval heap invariant stipulates that an interval must contain the intervals of its children. This implies that the first and last items of the root interval (at array index 0) are the least and greatest items in the interval heap. At each item insertion, removal or update, at most $\log(n)$ intervals need to be updated, where n is the number of items in the heap.

A handle h for an item is implemented as an object that contains the array index i of the item, thus permitting constant time retrieval of the item. More precisely, $\lfloor i/2 \rfloor$ is the array index of an interval, and the handle points to the interval's last item if i is even, otherwise to the interval's last item.

Each of the first and last items of an interval may contain a reference to a handle. If an item has an associated handle h, then whenever the item is moved (from one array index to another or from first to last within an interval or vice versa), then the item index i in h is updated accordingly. When the item is removed from the heap, the handle is invalidated. These operations take constant time.

To prevent a handle h from being used to access a heap with which it is not associated, we check that the item at $h.i$ has a handle and that that handle is h.

13.12 Implementation of events and handlers

Event handlers are implemented using C# delegates and events. There are six different kinds of events in the C5 library; see figure 8.4. However, it is undesirable to add six event handler fields to every collection object, especially since in the most frequent case those fields will all be `null`. Similarly, it is inefficient to test all those fields for being non-`null` after operations, such as update, that may raise up to five events.

Therefore we have collected all event handlers in a reference type `EventBlock`, so that one field suffices, and so that this field if `null` in the frequent case where there are no event handlers at all. This also helps ensure consistency between the `ActiveEvents` property (see page 46) and the event handlers actually associated with a collection.

Chapter 14

Creating new classes

14.1 Implementing derived collection classes

A number of abstract base classes provide common functionality for the various concrete collection classes. The abstract base classes can be used also to derive new collection classes. Note: This chapter is currently incomplete.

14.1.1 ArrayBase<T>

Abstract base class for implementing sequenced collections that are represented using an array, such as ArrayList<T>, HashedArrayList<T>, and SortedArray<T>. The class has fields and methods from CollectionBase<T>, CollectionValueBase<T>, DirectedCollectionBase<T>, EnumerableBase<T> and SequencedBase<T>.

Further has protected fields to hold underlying array and view offset.

Further methods to manage the underlying array, create an enumerator, insert and remove items.

14.1.2 CollectionBase<T>

Abstract base class for implementing modifiable collection classes, such as most classes in this library except for the read-only wrappers. The class has fields and methods from CollectionValueBase<T> and EnumerableBase<T>.

Further has protected fields: read-only flag, size, item equality comparers, and update stamp (for enumerators).

Further has methods for range check against current size, for computing collection hash codes, checks for modification during enumeration, and for creating an enumerator.

14.1.3 CollectionValueBase<T>

Abstract base class for implementing modifiable and unmodifiable collection classes. The class has fields and methods from EnumerableBase<T>.

Further has events (fields of delegate type), and utilities for event firing.

Further has Count and CountSpeed properties.

Public methods `All`, `Apply`, `Exists`, `Filter`; `CopyTo`, `ToArray`.

14.1.4 DictionaryBase<K,V>

Abstract base class for implementing dictionary classes. The class has bases CollectionValueBase<KeyValuePair<K,V>> and EnumerableBase<KeyValuePair<K,V>>.

14.1.5 DirectedCollectionBase<T>

Abstract base class for implementing directed collections. The class has fields and methods from CollectionBase<T>, CollectionValueBase<T> and EnumerableBase<T>.

14.1.6 DirectedCollectionValueBase<T>

Abstract base class for implementing directed collection values, such as subranges of indexed and sorted collections. The class has fields and methods from CollectionValueBase<T> and EnumerableBase<T>.

14.1.7 EnumerableBase<T>

Abstract base class for implementing enumerable classes. The class has a public method `GetEnumerator()` and protected methods for counting (by enumeration) the number of items.

14.1.8 SequencedBase<T>

Abstract base for implementing sequenced collections. The class has fields, events and methods inherited from CollectionBase<T>, CollectionValueBase<T>, DirectedCollectionBase<T> and EnumerableBase<T>. Further has methods for computing the collection's hash code and for determining collection equality, respecting the item sequence.

14.1.9 SortedDictionaryBase<K,V>

Abstract base class for implementing sorted dictionary classes such as SortedDictionaryBase<K,V>. The class has fields and methods from CollectionValueBase<KeyValuePair<K,V>>, DictionaryBase<K,V>, and EnumerableBase<KeyValuePair<K,V>>.

14.1.10 Read-only wrappers for abstract base classes

The Guarded classes shown in figure 14.1 are mainly useful as base classes for creating read-only wrappers of derived classes.

Abstract base class	Read-only wrapper class
CollectionValueBase<T>	GuardedCollectionValue<T>
DirectedCollectionBase<T>	GuardedDirectedCollectionValue<T>
	GuardedDirectedEnumerable<T>
EnumerableBase<T>	GuardedEnumerable<T>
	GuardedEnumerator<T>
SequencedBase<T>	GuardedSequenced<T>
	GuardedSorted<T>

Figure 14.1: Read-only wrappers for abstract base classes. Read-only wrappers for non-abstract collection and dictionary classes are shown in figure 8.2.

Bibliography

[1] Nunit. Web site. At http://www.nunit.org/.

[2] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9:216–219, 1979.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, fourth edition, 2005.

[4] M. Bender et al. Two simplified algorithms for maintaining order in a list. In *10th Annual European Symposium on Algorithms (ESA 2002), Rome, Italy, September 2002. Lecture Notes in Computer Science, vol. 2461*, pages 152–164. Springer-Verlag, 2002.

[5] Jon Bentley, D. E. Knuth, and M. D. McIlroy. Programming pearls: A literate program. *Communications of the ACM*, 29(6):471–483, June 1986.

[6] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001.

[7] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1992). SIGPLAN Notices 27, 10*, pages 1–15, 1992.

[8] Copenhagen STL. Homepage. Web site. At http://www.cphstl.dk/.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[10] J. Driscoll, N. Sarnak, D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and Systems Sciences*, 38(1):86–124, 1989.

[11] Ecma TC39 TG2. *C# Language Specification. Standard ECMA-334, 3rd edition*. June 2005. At http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[12] Ecma TC39 TG3. *Common Language Infrastructure (CLI). Standard ECMA-335, 3rd edition*. June 2005. At http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[13] M. Evered and G. Menger. Eine Evaluierung des Java JDK 1.2 Collections Framework aus Sicht der Softwaretechnik. In C.H. Cap, editor, *Java-Informations-Tage 1998, Frankfurt, Germany, November 1998*, pages 11–12. Springer-Verlag, 1999. At http://www.informatik.uni-ulm.de/rs/projekte/proglang/jdk.ps.

[14] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, 1989.

[15] P. Golde. Power Collections for .NET. Web site. At http://www.wintellect.com/MemberOnly/PowerCollections.aspx.

[16] R. Graham. An efficient algorithm for determining the convex hull of a finite point set. *Information Processing Letters*, 1:132–133, 1972.

[17] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley, 2003.

[18] Niels Jørgen Kokholm. An extended library of collection classes for .NET. Master's thesis, IT University of Copenhagen, Denmark, 2004.

[19] George Marsaglia. Re: New random generator. Posting to newsgroup comp.lang.c, April 2003. Posted 2003-04-03.

[20] George Marsaglia. Seeds for random number generators. *Communications of the ACM*, 46(5):90–93, 2003.

[21] G. Menger et al. Collection types and implementations in object-oriented software libraries. In *Conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, 1998*, pages 97–109, 1998.

[22] Microsoft. Microsoft .NET framework developer center. Web site. At `http://msdn.microsoft.com/netframework/`.

[23] David R. Musser. Introspective sorting and selection algorithms. *Software-Practice and Experience*, 27(8):983–993, 1997.

[24] Rasmus Pagh and Flemming Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[25] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986.

[26] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[27] P. Sestoft and H. I. Hansen. *C# Precisely*. Cambridge, Massachusetts: The MIT Press, October 2004.

[28] Søren Skov and Jesper Holm Olsen. A comparative analysis of three different priority deques. CPH STL Report 2001-14, DIKU, University of Copenhagen, October 2001.

[29] D. Sleator and P. Dietz. Two algorithms for maintaining order in a list. In *19th ACM Symposium on the Theory of Computing (STOC'87)*, pages 365–372. ACM Press, 1987.

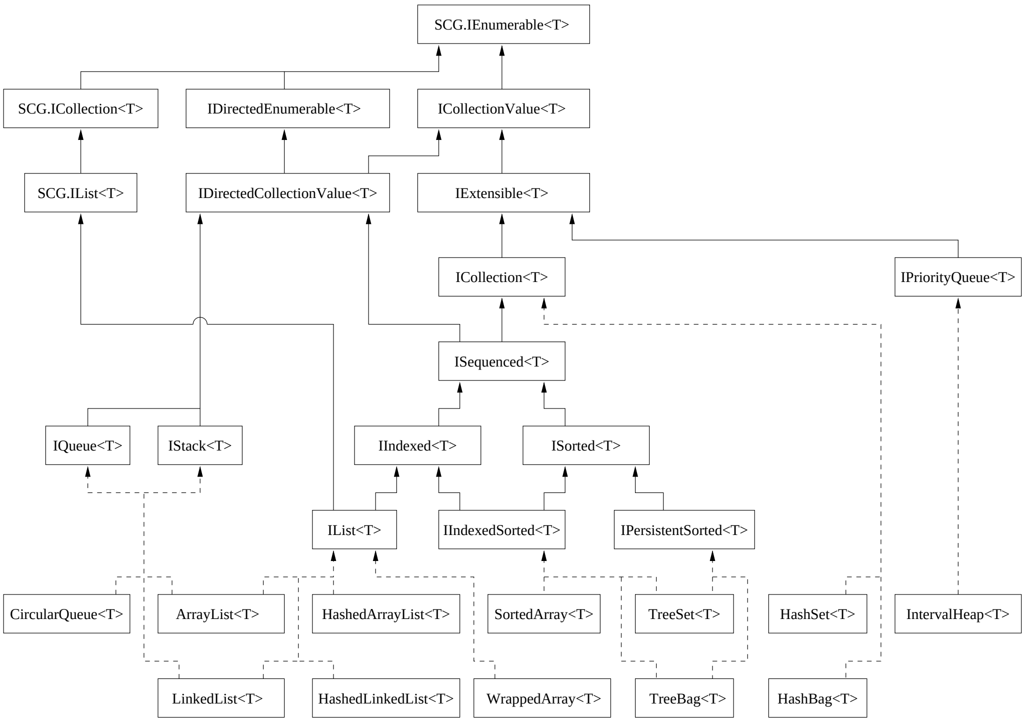
[30] D. Syme. F#. Web site. At `http://research.microsoft.com/projects/ilx/fsharp.aspx`.

[31] R.E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[32] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

Index

Collections



Dictionaries