# C5: Copenhagen Comprehensive C# Collection Classes

# and

# Experience with Generic C#

Niels Jørgen Kokholm and Peter Sestoft

IT University of Copenhagen, Denmark

and

Royal Veterinary and Agricultural University, Denmark

**Outline**

- Design goals and history of the C5 collection class library
- Design principles and implementation principles
- Collection classes in Smalltalk-80, Java 1.5, C# 2.0 and C++
- Capability (interface) hierarchy
- Data structure and algorithm (implementation) hierarchy
- Some feature highlights
- Some implementation highlights
- Assessment and remaining work
- Further experience using C# generics and other new features

## Goals of the C5 collection class library

- Provide a collection class library for C# that is as a comprehensive as those of comparable languages.
- Test and document well enough that it will be widely usable.

## History

- Experimented with C# generics during visit to MSR Cambridge in late 2001.

  In particular, created `GCollections.cs` to resemble Java 1.2 collections, with generics.

  Developed using MSR-internal builds of CLR and C# compiler.
- Niels Jørgen vastly extended and improved the library in his 2004 MSc thesis.

  Developed using Whidbey alpha release from August 2003.

  Recently converted to March 2004 Community Technology Preview version.
- Funding has been applied for from MSR Unirel for finishing, polishing, testing and documenting the library.

**Design principles**

- Provide well-known abstractions: lists, priority queues, sets, bags, dictionaries.
- Provide well-known data structure implementations (array-based, list-based, hash-based, order-based).
- Should fit with existing C# patterns (`IEnumerable<T>`, `foreach` statement, . . . )
- Provide convenient but hard-to-implement features; e.g. updatable views, persistence, reversible enumeration.
- Functionality should be described by interfaces: ‘program to an interface, not an implementation’.
- Capabilities — e.g. subrange and enumeration direction — should be orthogonal to avoid method proliferation.
- Document the asymptotic run-time complexity of all implementations.

**Implementation principles**

- Use best known data structures and algorithms, even if cumbersome to implement.
- Asymptotics (scalability) are more important than nanosecond efficiency.
- It is OK to sacrifice a constant-factor space overhead to support richer functionality.
- Concurrent read-accesses, including iteration, must be naturally thread-safe (no synchronization overhead).
- Have the machine model in mind. E.g. avoid splay trees: they incur many write-barrier checks.
- Measure performance of implementation alternatives; e.g. whether to use objects or structs internally.
- Use only managed code, no unsafe operations.
- Test carefully using Nunit, and document test coverage.
- Write all code from scratch and release under an MIT-style license.

## Other collection libraries

- Smalltalk-80 comprehensive library: sets, bags, lists, sequences, hash-based and tree-based dictionaries.

  Critique by Cook (1992): capabilities and implementation are mixed; functionality is (unsystematically) duplicated.

  (Smalltalk has no interfaces and no generic types.)

- Java 1.2 has comprehensive library: lists, sets and dictionaries; array-based, linked, hashtables, binary trees.

  Critique by Evered and Menger (1998): Separation into interface hierarchy and class hierarchy a step forward, but not complete: interfaces do not describe all capabilities of implementations.

  Java 1.5 generic collection library is based directly on the non-generic library.

- C#/.Net Framework classes version 1.1 has no linked lists and no tree-based dictionaries.

  Oddly named SortedList is an array-based comparison-based dictionary called ‘a HashTable/Array hybrid’.

  Version 2.0 generic collection library is less weird (SortedDictionary) and has LinkedList, but still not trees.

- C++ Standard Template Library

  Linked lists, array-based lists, sets, multisets, maps and multimaps as binary trees (no standard hashtables?).

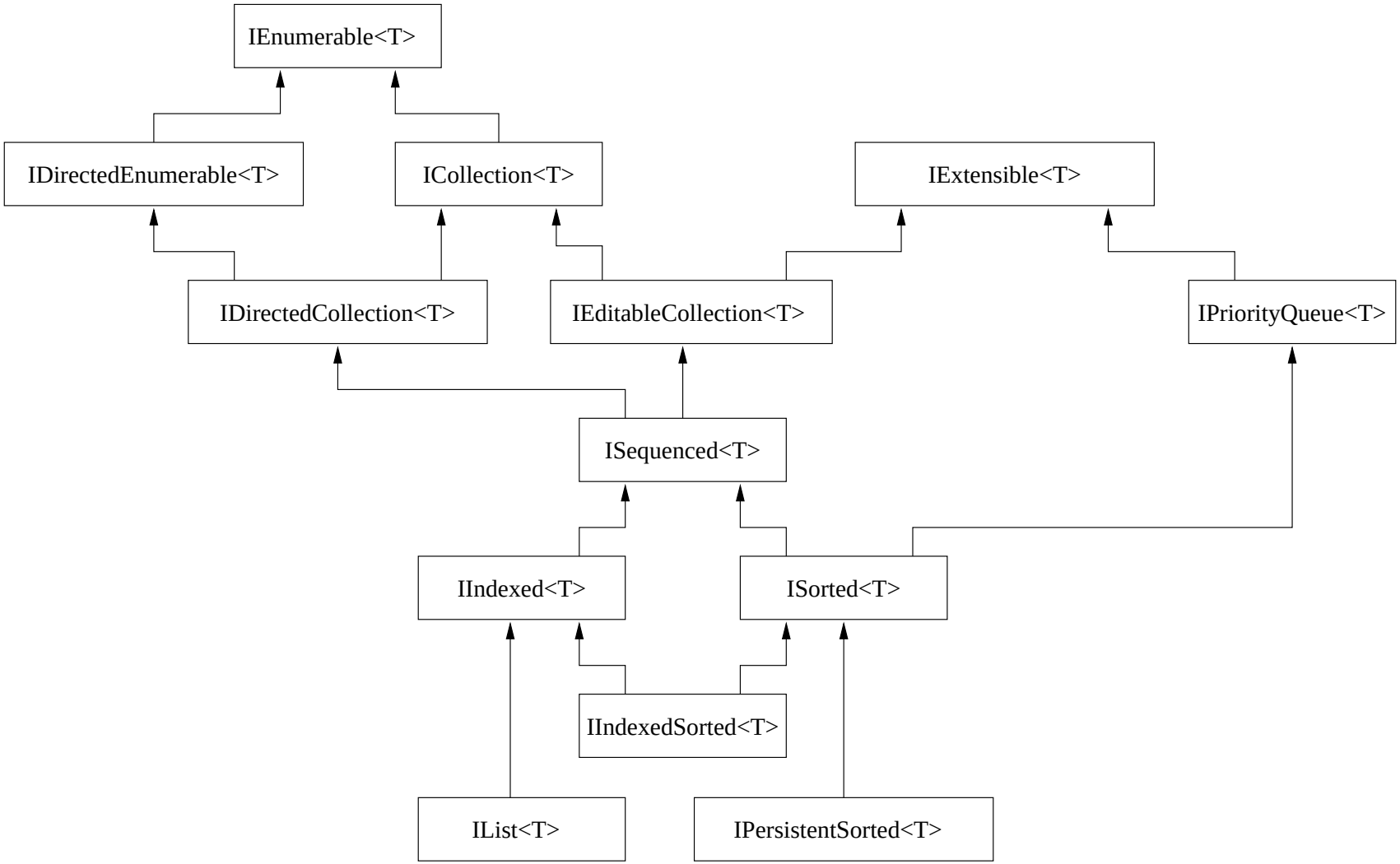
  Provides only efficient operations, and e.g. linked-list and arraylist do not derive from same base.

  Therefore no clear separation between capabilities and implementation.

**C5 supported concepts**

- Sequences — implemented by array-based lists and doubly-linked lists.
- Sequences without duplicates — implemented by array-based lists and doubly-linked lists with hash index.
- Sets — implemented by hashtables and ordered red-black binary trees.
- Bags or multisets — implemented by hashtables and ordered red-black binary trees.
- Dictionaries — implemented by hashtables and ordered red-black binary trees.
- Priority queues — implemented by interval heaps.
- Snapshots of collections and dictionaries — implemented by persistent red-black binary trees.
- Subrange views of sequenced collections
- Reversible enumeration of sequenced collections

## Collection capabilities: Interface hierarchy



| Collection Concept | Supported Operations |
|---|---|
| Enumerable | get enumerator; apply function; exists and for all quantifiers |
| Collection | enumerable; element count |
| DirectedEnumerable | enumerable that can produce reversed enumerable (and enumerator) |
| DirectedCollection | collection that can produce reversed enumerable |
| Extensible | accepts adding of elements; emptiness test |
| EditableCollection | extensible collection; contains; find; remove; update; clear; to array |
| Sequenced | directed editable collection; equality comparison in item order |
| Indexed | *by integer index*: sequenced collection with element access, insertion, removal |
| List | indexed collection that supports sorting, updatable views, `FindAll(p)` and `Map(f)` with list result |
| PriorityQueue | extensible that supports efficient `FindMin` and `FindMax` |
| Sorted | *by element ordering*: sequenced that has predecessor, successor, cut by element, element subrange queries |
| IndexedSorted | sorted and indexed collection; element subrange counts and queries |
| PersistentSorted | sorted collection that supports constant-time snapshots |

**C5 enumerables, collections, and directed enumerables and collections**

```
interface IEnumerable<T> {
  System.Collections.Generic.IEnumerator<T> GetEnumerator();
  void Apply(Applier<T> applier);
  bool Exists(Filter<T> filter);
  bool All(Filter<T> filter);
}

interface IDirectedEnumerable<T> : IEnumerable<T> {
  IDirectedEnumerable<T> Backwards();
  EnumerationDirection Direction { get; }
}

interface ICollection<T> : IEnumerable<T> {
  int Count { get; }
  void CopyTo(T[] a, int i);
}

interface IDirectedCollection<T> : ICollection<T>, IDirectedEnumerable<T> {
  new IDirectedCollection<T> Backwards();
}
```

A recent .Net CLI proposal makes `S.C.Generic.ICollection<T>` a mix of collection and editable.

Not really desirable, as some implementation of `Add(T x)` will be by throwing NotSupportedException.

**C5 extensible and editable collections**

```
public interface IExtensible<T> {
  bool NoDuplicates { get; }
  bool IsEmpty { get; }
  bool Add(T item);
  void AddAll(IEnumerable<T> items);
}

interface IEditableCollection<T> : ICollection<T>, IExtensible<T> {
  bool IsReadOnly { get; }
  Speed ContainsSpeed { get; }                    // for fast Equals
  int GetHashCode();
  bool Equals(IEditableCollection<T> that);
  bool Contains(T item);
  int ContainsTimes(T item);                      // bag
  bool ContainsAll(IEnumerable<T> items);
  bool Find(ref T item);
  bool FindOrAdd(ref T item);
  bool Update(T item);
  bool UpdateOrAdd(T item);
  bool Remove(T item);
  bool RemoveWithReturn(ref T item);
  void RemoveAllCopies(T item);                   // bag
  void RemoveAll(IEnumerable<T> items);
  void Clear();
  void RetainAll(IEnumerable<T> items);
  T[] ToArray();
}
enum Speed: short { Linear = 1, Log = 2, Constant = 3 }
```

**C5 sequenced (by item position) and indexed**

Comparison of sequenced collections takes item order into account:

```
interface ISequenced<T> : IEditableCollection<T>, IDirectedCollection<T> {
  new int GetHashCode();
  bool Equals(ISequenced<T> that);
}
```

A subsequence of an indexed collection is a directed collection:

```
interface IIndexed<T> : ISequenced<T> {
  T this[int i] { get; }
  IDirectedCollection<T> this[int start, int end] { get; }
  int IndexOf(T item);
  int LastIndexOf(T item);
  T RemoveAt(int i);
  void RemoveInterval(int start, int count);
}
```

So one can enumerate a subsequence backwards:

```
foreach (T x in coll[10, 15].Backwards()) {
  ...
}
```

**C5 lists**

```
interface IList<T> : IIndexed<T> {
  T First { get; }
  T Last { get; }
  bool FIFO { get; set; }
  new T this[int i] { get; set; }
  void Insert(int i, T item);
  void InsertFirst(T item);
  void InsertLast(T item);
  void InsertBefore(T item, T target);
  void InsertAfter(T item, T target);
  void InsertAll(int i, IEnumerable<T> items);
  IList<T> FindAll(Filter<T> filter);
  IList<V> Map<V>(Mapper<T,V> mapper);
  T Remove();
  T RemoveFirst();
  T RemoveLast();
  IList<T> CreateView(int start, int count);
  IList<T> Base { get; }
  int Offset { get; }
  void Slide(int offset);
  void Slide(int offset, int size);
  void Reverse();
  void Reverse(int start, int count);
  bool IsSorted(IComparer<T> c);
  void Sort(IComparer<T> c);
  void Shuffle();
  void Shuffle(Random rnd);
}
```

**C5 priority queues**

```
interface IPriorityQueue<T> : IExtensible<T>, IComparer<T> {
  T FindMin();
  T DeleteMin();
  T FindMax();
  T DeleteMax();
}
```

Sometimes only `FindMin` or `FindMax` is needed.

But although it is trivial to turn a min-problem into a max-problem, it causes confusion in practice.

And since interval heaps are just as fast as single-ended heaps, we provide both functionalities in one.

**C5 sorted — by element ordering**

```
interface ISorted<T> : ISequenced<T>, IPriorityQueue<T> {
  T Predecessor(T item);                    //  < item
  T Successor(T item);                      //  > item
  T WeakPredecessor(T item);                // <= item
  T WeakSuccessor(T item);                  // >= item
  bool Cut(IComparable<T> c, out T low,  out bool lowIsValid,
                             out T high, out bool highIsValid);
  IDirectedEnumerable<T> RangeFrom(T bot);
  IDirectedEnumerable<T> RangeFromTo(T bot, T top);
  IDirectedEnumerable<T> RangeTo(T top);
  IDirectedCollection<T> RangeAll();
  void AddSorted(IEnumerable<T> items);
  void RemoveRangeFrom(T low);
  void RemoveRangeFromTo(T low, T hi);
  void RemoveRangeTo(T hi);
}
```

Cut finds the greatest `low <= c` and the least `high > c`, if any.

A range query produces a directed enumerable. Hence it can be enumerated (or processed) backwards:

```
foreach (Talk t in talks.RangeTo(nexttalk).Backwards()) {
  ...
}
```

**C5 indexed and sorted, and persistent sorted**

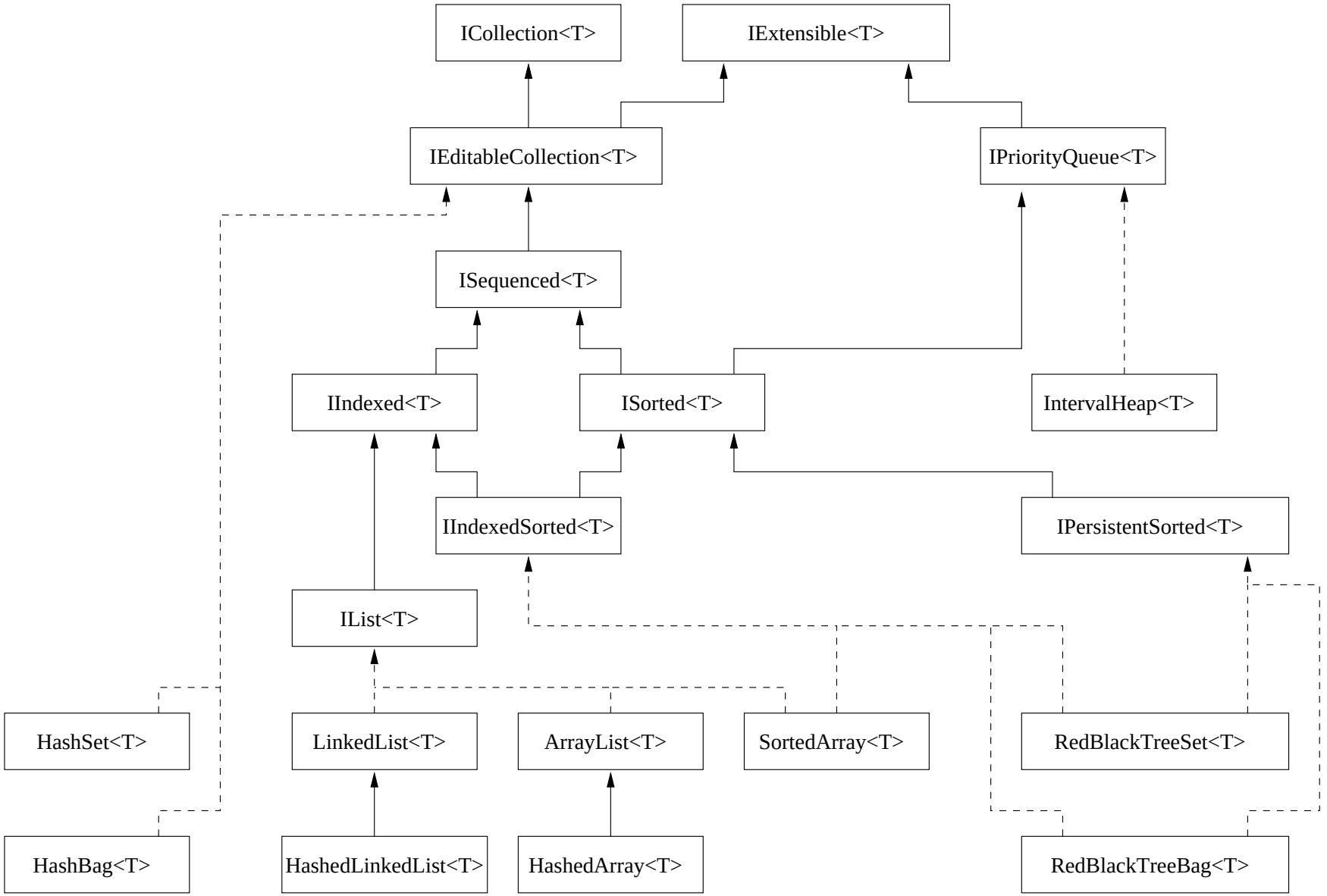
```
interface IIndexedSorted<T> : ISorted<T>, IIndexed<T> {
  int CountFrom(T bot);
  int CountFromTo(T bot, T top);
  int CountTo(T top);
  new IDirectedCollection<T> RangeFrom(T bot);
  new IDirectedCollection<T> RangeFromTo(T bot, T top);
  new IDirectedCollection<T> RangeTo(T top);
  IIndexedSorted<T> FindAll(Filter<T> f);
  IIndexedSorted<V> Map<V>(Mapper<T,V> m, IComparer<V> c);
}

interface IPersistentSorted<T> : ISorted<T> {
  ISorted<T> Snapshot();
}
```

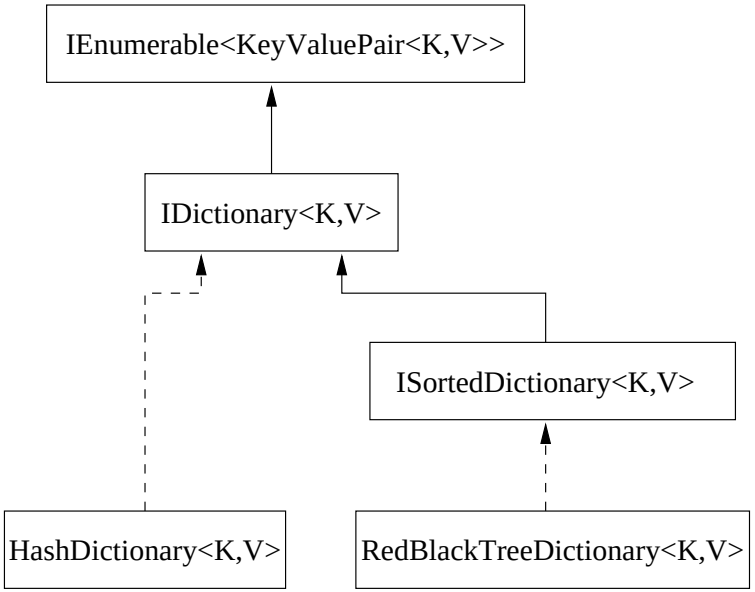
A persistent sorted collection supports making a snapshot (but not snapshots of snapshots).

In principle one could have persistent non-sorted collections, but there are no efficient implementations.

## Collection implementations



## Dictionary capabilities and implementations



**C5 dictionaries and dictionaries with sorted keys**

```
interface IDictionary<K,V> : IEnumerable<KeyValuePair<K,V>> {
  V this[K key] { get; set; }
  int Count { get; }
  bool IsReadOnly { get; }
  ICollection<K> Keys { get; }
  ICollection<V> Values { get; }
  void Add(K key, V val);
  bool Remove(K key);
  bool Remove(K key, out V val);
  void Clear();
  bool Contains(K key);
  bool Find(K key, out V val);
  bool Update(K key, V val);          //no-adding
  bool FindOrAdd(K key, ref V val);   //mixture
  bool UpdateOrAdd(K key, V val);
}

interface ISortedDictionary<K,V> : IDictionary<K,V> {
  KeyValuePair<K,V> Predecessor(K key);
  KeyValuePair<K,V> Successor(K key);
  KeyValuePair<K,V> WeakPredecessor(K key);
  KeyValuePair<K,V> WeakSuccessor(K key);
  IEnumerable<KeyValuePair<K,V>> Snapshot();
}
```

**Some feature highlights**

- Updatable views of linked lists.
- Range queries by index (indexed collections) and by elements (sorted collections).
- Reversible enumerations.
- Snapshots of red-black trees (persistent trees).

**Some implementation highlights**

- How to support both hash-indexes and views of a linked list.
- How to support constant-time snapshots of tree-based dictionaries.
- Introspective quicksort for arrays; worst-case running-time logarithmic.
- In-place smooth stable mergesort for doubly-linked lists.

**Feature highlight: Updatable views of lists**

A view of a list is a sublist of the list. All views work on the same underlying list.

Updates to a view will update the underlying list. Other updates to the underlying list invalidate the view(s).

Common applications:

- Orthogonality: Make operation, such as `Contains`, independent of the list subrange operated on.

  There are many operations: `Find`, `IndexOf`, `LastIndexOf`, `Update`, `FindAll`, `CopyTo`, ...

  In the Smalltalk and .Net libraries, some of these exist in three versions each, causing method proliferation.

- Enumeration and reverse enumeration of sublists.

- Point into a list (without exposing its implementation by array or linked-list nodes):

  A one-element view points to a particular list item.

  A zero-element view points to the space between (or before or after) list items.

  In an $n$-element list there are $n$ one-element views and $n + 1$ zero-element views: spaces between items.

**Application of list views: convex hull algorithm**

Convex hull: Least convex set that encloses a given set of points.

Sort points $(x, y)$ lexicographically, and separate in upper and lower set.

Then perform Graham's clockwise point elimination scan:

- 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 $p_0$.

Implementation of Graham's scan using views (of linked lists, else inefficient):

```
IList<Point> view = lst.CreateView(0, 0);
int slide = 0;
while (view.Offset + slide + 3 <= lst.Count) {
  view.Slide(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;
  }
}
```

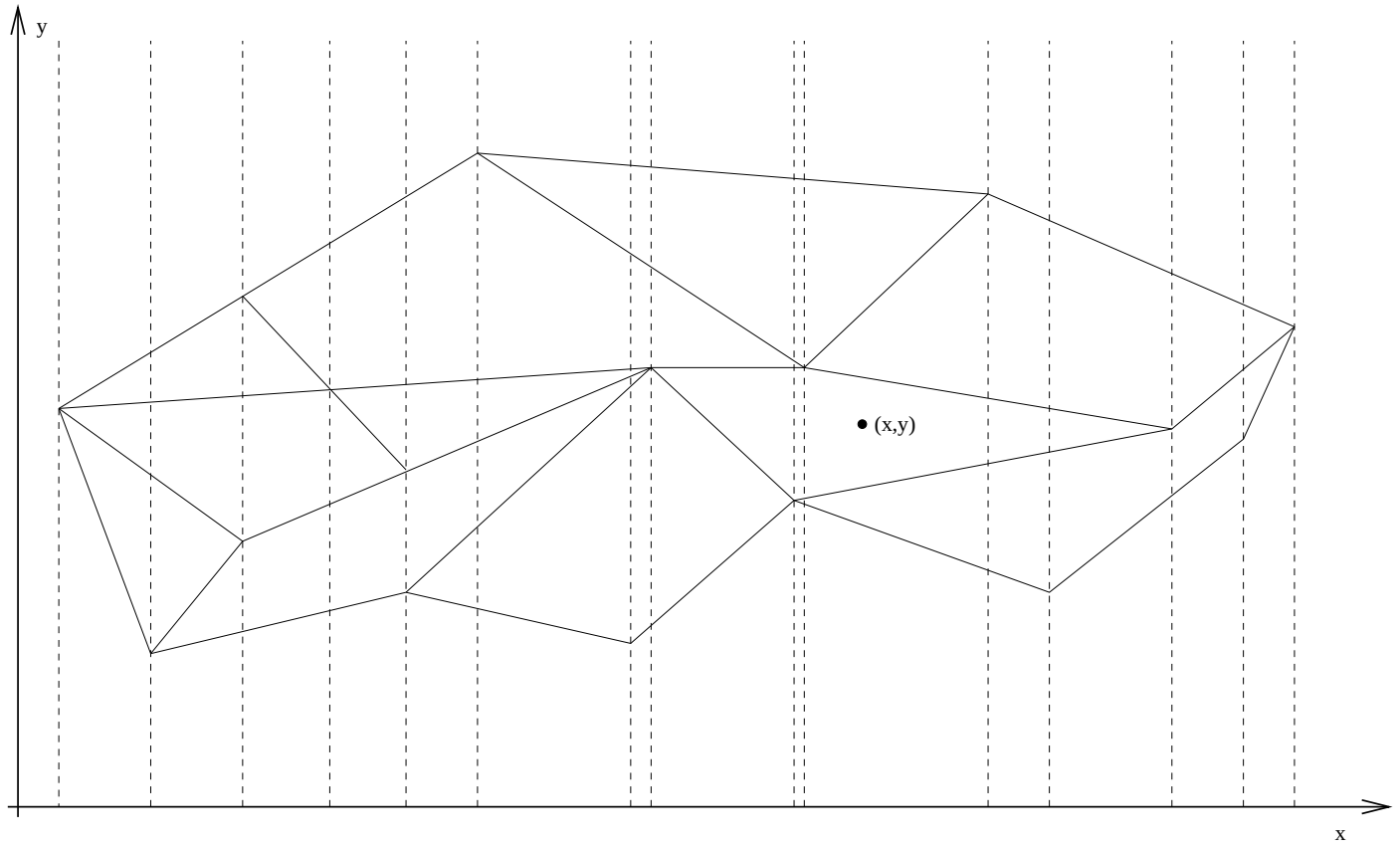
Considerably clearer than previous implementation using explicit linked list nodes.

**Feature highlight: Constant-time snapshots of red-black trees**

Later updates to the tree do not affect the snapshot. A snapshot is not updatable (hence semi-persistent).

Application 1: Iteration over a (snapshot of a) collection while modifying the collection.

Application 2: Geometric algorithms such as planar point location use many almost-identical dictionaries:




Each dashed vertical line can be represented by a snapshot of a dictionary. Much faster than using copies.

**Implementation highlight: Constant-time snapshots of red-black trees**

Implementation uses node-copy persistence.

Initially a snapshot shares all tree nodes with the underlying red-black tree.

When the underlying tree is updated and any snapshot exists, tree nodes are copied lazily.

This requires extra data, including one reference, in each tree node.

Then all operations remain $O(\log n)$ and use only amortized constant extra space.

This uses ideas from Driscoll, Sarnak, Sleator, Tarjan (1989).

Also tried path copying persistence and other variations, but node copying persistence is faster.

Recommended idiom:

```
using (ISorted<T> snap = tree.Snapshot()) {
  ...
}
```

When all snapshots have been disposed (or finalized), the underlying tree stops making node copies at updates.

**Implementation highlight: Combining views and hashed indexes on linked lists**

Taken separately, views on linked lists and hash-indexes on linked lists are easy to implement.

But the combination is tricky: We want to use a single hashtable for a list and all its views.

How do we know whether a linked-list node found in the hashtable belongs to a given view?

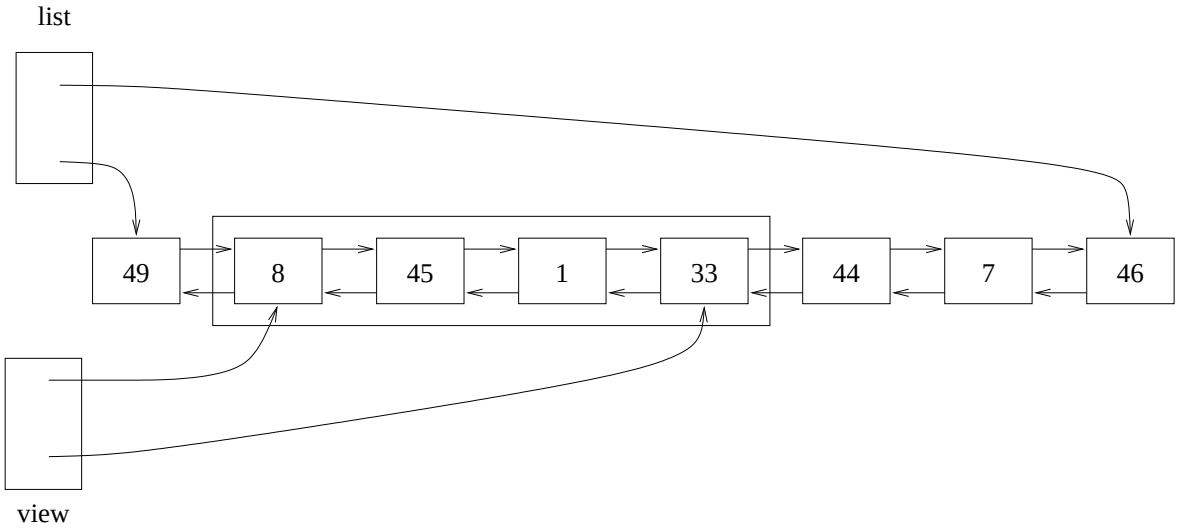
(In an array-based list, just check whether the found item index is within the view's index range.)

Put increasing integer tags on list nodes; a node's tag must be between the tags of the view's first and last node.

The tricky part is to maintain increasing tag order when inserting (and deleting) list nodes.

Uses ideas from Sleator and Tarjan (1987) and Bender *et al.* (2002) to do this in amortized constant time.

Further improved by organizing elements into sufficiently large tag groups; speeds up updates nearly twice.



**Implementation highlight: Sorting algorithms**

- Introspective quicksort for arrays (Musser 1997):

  Guaranteed $O(n \log n)$ worst-case run-time complexity.

  Faster than standard MS .Net array sort for moderate random data sets, and 2 % slower on large ones.

  But vastly better than the standard MS .Net array sort on 'killer sequences'.

  Core idea: If recursive partitioning proceeds beyond logarithmic depth, then use heapsort on that segment.

- Smooth stable in-place mergesort for doubly-linked lists:

  Smooth: near-linear time on nearly sorted data; guaranteed $O(n \log n)$ worst-case run-time complexity.

  Stable: preserves the order of equal elements.

  In-place: does not copy the data elements and uses no extract space.

  On random data, slower than array-based quicksort due to the many updates and write-barrier checks.

  Core idea: use the predecessor link in a doubly-linked list node to build contiguous non-decreasing sublists.

**C5 state of completion**

- Everything has been implemented and works (according to tests).
- The C5 source code (excluding test cases and comments) is 17,817 lines of C#.
- The compiled release build `C5.dll` is 155,648 bytes.
- Introductory overview of the library, and some documentation of the API exists.
- The unit test code (excluding comments) is 18,988 lines of C#, plus some examples.

**C5 assessment**

- Probably a mistake to leave out `Queue<T>` and `Stack<T>`.

  These are trivial subclasses of `LinkedList<T>` and `ArrayList<T>`, but irritating to implement.
- Maybe include initializing constructors `HashSet(IEnumerable<T> enm)` and similar.
- Should probably have range queries on sorted dictionaries too.
- Seems like a good design. Builds on lots of experience with libraries for Standard ML and Java.
- Non-trivial algorithms developed by leading experts and implemented by a very competent programmer.
- More experience with the learnability and practical performance of the code would be good.

**Remaining work on C5**

- Exploit new generics features, such as ‘naked’ type bounds:

```
interface IExtensible<T> {
  void AddAll<U>(IEnumerable<U> enm) where U : T;
  ...
}
```

  This partially compensates for the absence of covariance.

- Adapt to new standard .Net `ICompare<T>`, `IComparable<T>`, name scheme, and so on.

  Use (new) standard .Net delegate types for higher-order functions.

- Fix some mistakes, including:

  Consistent naming of types and operations; consistent argument semantics.

  Drop plain heaps (priority queues); interval heaps are just as fast.

  Reimplement `AddAll` in two cases to get optimal run-time complexity.

- Finish API documentation.

- Add a tutorial that provides examples of all features.

- Do more performance engineering.

- Create Web site from which the library can be downloaded.

**Experience with Whidbey**

- Generally the .Net CLR implementation August 2003 alpha was robust and supported the project very well.
- There was some flakiness in the Visual Studio IDE (but entirely forgivable in an alpha release).
- Difficult to predict efficiency of CIL code. Must measure, and then get some surprises:

  Static field access is very slow; yet another reason to avoid them.

  Sometimes method parameter access is fast and this is good:

```
public void M(int x) {
  ...
  while (...) {
    ... x ...
  }
}
```

  But sometimes a local variable is much better:

```
public void M(int x) {
  int y = x;
  while (...) {
    ... y ...
  }
}
```

- An attempt was made to use run-time code generation via System.Reflection.Emit (for generation of hashers and comparers) but that failed for lack of support of generics.

**Exercising C# generics: Polynomials over an arbitrary ring (Bertrand Meyer)**

One can add or multiply a value of type `AddMul<A,R>` with an `A`, given result of type `R`:

```
interface AddMul<A,R> {
  R Add(A e);                    // Addition with A, giving R
  R Mul(A e);                    // Multiplication with A, giving R
}
```

Can define polynomials over `E` if `E` supports addition and multiplication and has a zero (made by `new E()`):

```
class Polynomial<E> : AddMul<E,Polynomial<E>>,
                      AddMul<Polynomial<E>,Polynomial<E>>
  where E : AddMul<E,E>, new() {
  private readonly E[] cs;       // cs contains coefficients of xˆ0, xˆ1, ...


  public Polynomial<E> Add(Polynomial<E> that) { ... }
  public Polynomial<E> Add(E that) { ... }
  public Polynomial<E> Mul(Polynomial<E> that) { ... }
  public E Eval(E x) {
    E res = new E();             // Permitted by constraint E : new()
    for (int j=cs.Length-1; j>=0; j--)
      res = res.Mul(x).Add(cs[j]);
    return res;
  }
}
```

A `Polynomial<E>` supports addition and multiplication both with `E` and `Polynomial<E>`!

**Higher-order functional programming in C#**

Function types (as delegate types), and some higher-order functions:

```
public delegate R Fun1<A,R>(A x);                    //  A -> R
public delegate R Fun2<A1,A2,R>(A1 x1, A2 x2);       //  A1 * A2 -> R

// Compose1 : (I -> R) * (A -> I) -> (A -> R)
public static Fun1<A,R> Compose1<A,I,R>(Fun1<I,R> f, Fun1<A,I> g) {
  return delegate(A x) { return f(g(x)); };
}

// Curry : (A * B -> C) -> (A -> (B -> C))
public static Fun1<A,Fun1<B,C>> Curry<A,B,C>(Fun2<A,B,C> f) {
  return delegate(A x) {
    return delegate(B y) {
      return f(x, y);
    };
  };
}

// UnCurry : (A -> (B -> C)) -> (A * B -> C)
public static Fun2<A,B,C> UnCurry<A,B,C>(Fun1<A,Fun1<B,C>> f) {
  return delegate(A x, B y) {
    return f(x)(y);
  };
}
```

**Lazy streams as enumerables**

```
// Map : (A -> R) * A stream -> R stream
public static IEnumerable<R> Map<A,R>(Fun1<A,R> f, IEnumerable<A> xs) {
  foreach (A x in xs)
    yield return f(x);
}

// Memoize : T stream -> T stream
public static IEnumerable<T> Memoize<T>(IEnumerable<T> eble) {
  return new MemoizedEnumerable<T>(eble);
}

private class MemoizedEnumerable<T> : IEnumerable<T> {
  ... cache the elements of a sequence, lazy functional style ...
}

// Merge : int stream * int stream -> int stream
public static IEnumerable<int> Merge(IEnumerable<int> xEble,
                                     IEnumerable<int> yEble) {
  ... sorted merge of sorted sequences ...
}
```

**Computing the Hamming sequence**

Hamming: All products of 2, 3 and 5, in increasing order: $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, \ldots$

In lazy functional languages the sequence is recursively definable as

```
hamming = 1 : merge(map (*2) hamming, merge(map (*3) hamming, map (*5) hamming)
```

Possible in C# also (using an anonymous method to explicitly delay evaluation):

```
IEnumerable<int> hamming = null;
hamming =
  Memoize<int>(Prefix1(new Delay<IEnumerable<int>>
                           (delegate { return Hamming(hamming); })));
```

The method `Hamming` returns the 2-, 3- and 5-multiples of a given integer sequence `xs`:

```
public static IEnumerable<int> Hamming(IEnumerable<int> xs) {
  return Merge(Map<int,int>(delegate(int x) { return 2 * x; }, xs),
               Merge(Map<int,int>(delegate(int x) { return 3 * x; }, xs),
                     Map<int,int>(delegate(int x) { return 5 * x; }, xs)));
}
```

The `Prefix1` method prepends to 1 to a delayed `int` sequence:

```
public static IEnumerable<int> Prefix1(Delay<IEnumerable<int>> xs) {
  yield return 1;
  foreach (int x in xs())
    yield return x;
}
```

**Efficiency benefit of generics (2004): quicksort**

| Description | General | Typesafe | Generics | Ints | Strings |
|---|---|---|---|---|---|
| Object-based, interface IComparable | yes | no | no | 5.00 | 4.68 |
| Object-based, class OrderedInt/String | yes | no | no | 3.60 | 4.99 |
| Generic with untyped CompareTo | yes | no | yes | 3.51 | 4.96 |
| Generic with typed CompareTo | yes | yes | yes | 3.56 | 5.02 |
| Generic with Compare delegate | yes | yes | yes | 1.40 | 4.67 |
| Generic with Compare method | yes | yes | yes | 1.36 | 4.37 |
| Hand-specialized with Compare method | no | yes | no | 1.28 | 4.35 |
| Hand-specialized, inline $<$ | no | yes | no | 0.50 | 4.23 |

Random ints (1.000.000) or strings (200.000); average time/s of 20 runs; 850 MHz mobile P-III; Windows 2000; March 2004 CTP of Whidbey.

- Generics is the only way to have generality, type safety, and efficiency.
- The overhead in generics (1.36 vs 0.50) is due mostly to generality: passing the Compare method.
- The generics win is clearly larger for the value type `int` than for the reference type `string`.
- (String comparison is quite slow, so a dictionary with String keys should be hash-based not tree-based).

**General impression of C# 2.0**

Pleasant to use, with convenient syntax for many idioms.

The many features interact rather well; few surprises. (Value types and `readonly`).

Rather complicated to understand (and explain) in detail. Consider method calls:

- Kinds: static, instance, instance virtual, abstract, interface, and explicit interface member implementation.
- By-value and by-reference (`ref`/`out`) parameter passing; and value type and reference type arguments.
- Access modifiers; inherit from base class versus getting the method from an enclosing class.
- Parameter arrays; overloading resolution and ‘better conversion’ at compile-time.
- Run-time calls to virtual methods, respecting `new` in the subclass chain.
- Implicit argument conversions at run-time.
- And on top of this, generic type parameter inference.

Also, the autoboxing of simple type values may introduce performance surprises in connection with generics:

- Storing an `int` in an object-based `TreeSet` would require a single initial boxing.
- But accidentally using object-based comparisons `((Object)i1).CompareTo((Object)i2)` inside `TreeSet<int>` causes boxing at every comparison . . .