Stacks and queues

Rasmus Pagh

Based on slides by Kevin Wayne, Princeton

Fundamental data types.

- Values: sets of objects.
- Operations: insert, remove, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

**Stack.** Remove the item most recently added.

**Analogy.** Cafeteria trays, Web surfing.

**Queue.** Remove the item least recently added.

**Analogy.** Line in the canteen.
Separate interface and implementation.
Ex: stack, queue, priority queue, symbol table, union-find, ...

Benefits.
- Client can't know details of implementation ⇒
  client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒
  many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

**Client:** program using operations defined in interface.
**Implementation:** actual code implementing operations.
**Interface:** description of data type, basic operations.
Warmup. Stack of strings objects.

```java
public class StackOfStrings {
    StackOfStrings() {
        // create an empty stack
    }
    void push(String s) {
        // insert a new item onto stack
    }
    String pop() {
        // remove and return the item most recently added
    }
    boolean isEmpty() {
        // is the stack empty?
    }
    int size() {
        // number of items on the stack
    }
}
```

Challenge. Reverse sequence of strings from standard input.
public class StackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    {
        return first == null;
    }

    public void push(String item)
    {
        ...
    }

    public String pop()
    {
        ...
    }
}
Stack pop: linked-list implementation

save item to return

String item = first.item;

delete first node

first = first.next;

return saved item

return item;
Stack push: linked-list implementation

**save a link to the list**

```java
Node oldfirst = first;
```

**create a new node for the beginning**

```java
first = new Node();
```

**set the instance variables in the new node**

```java
first.item = "not";
first.next = oldfirst;
```
public class StackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    {  return first == null;  }

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
Stack: array implementation

Array implementation of a stack.

• Use array $s[]$ to store $N$ items on stack.
• $\text{push}()$: add new item at $s[N]$.
• $\text{pop}()$: remove item from $s[N-1]$.

Defect. Stack overflows when $N$ exceeds capacity. [stay tuned]
public class StackOfStrings
{
    private String[] s;
    private int N = 0;

    public StackOfStrings(int capacity)
    {
        s = new String[capacity];
    }

    public boolean isEmpty()
    {
        return N == 0;
    }

    public void push(String item)
    {
        s[N++] = item;
    }

    public String pop()
    {
        return s[--N];
    }
}
Stack considerations

Overflow and underflow.

- **Underflow**: throw exception if pop from an empty stack.
- **Overflow**: use resizing array for array implementation. [stay tuned]

Loitering. Holding a reference to an object when it is no longer needed.

```java
public String pop()
{
    String item = s[--N];
    s[N] = null;
    return item;
}
```

Loitering

```java
public String pop()
{
    String item = s[--N];
    s[N] = null;
    return item;
}
```

this version avoids "loitering":
garbage collector reclaims memory
only if no outstanding references

Null items. We allow null items to be inserted.
› stacks
› resizing arrays
› queues
› generics
› iterators
› applications
Problem. Requiring client to provide capacity does not implement API!
Q. How to grow and shrink array?

Problem session: Think about this!
(And try not to look at the next few slides.)
Problem. Requiring client to provide capacity does not implement API!
Q. How to grow and shrink array?

First try.
• \texttt{push()}: increase size of \texttt{s[]} by 1.
• \texttt{pop()}: decrease size of \texttt{s[]} by 1.

Too expensive.
• Need to copy all item to a new array.
• Inserting first \( N \) items takes time proportional to \( 1 + 2 + \ldots + N \sim \frac{N^2}{2} \).

Challenge. Ensure that array resizing happens infrequently.
Stack: dynamic-array implementation

Q. How to grow array?
A. If array is full, create a new array of twice the size, and copy items.

```
public StackOfStrings() {  s = new String[1];  }

public void push(String item)
{
    if (N == s.length) resize(2 * s.length);
    s[N++] = item;
}

private void resize(int capacity)
{
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
       copy[i] = s[i];
    s = copy;
}
```

Consequence. Inserting first $N$ items takes time proportional to $N$ (not $N^2$).
Cost of inserting first $N$ items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N$. 

Stack: amortized cost of adding to a stack

Cost (array accesses) per push

$1$ array accesses 

$\uparrow$

$k$ array accesses 

to double to size $k$ 

one gray dot for each operation 

red dots give cumulative average 

number of push() operations
Q. How to shrink array?

First try.

• push(): double size of s[] when array is full.
• pop(): halve size of s[] when array is one-half full.

Too expensive.

• Consider push-pop-push-pop-... sequence when array is full.
• Takes time proportional to $N$ per operation in worst case.
Q. How to shrink array?

Efficient solution.

- `push()`: double size of `s[]` when array is full.
- `pop()`: halve size of `s[]` when array is one-quarter full.

```java
public String pop()
{
    String item = s[--N];
    s[N] = null;
    if (N > 0 && N == s.length/4) resize(s.length / 2);
    return item;
}
```

Invariant. Array is between 25% and 100% full.
Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from empty stack (with dynamic resizing), any sequence of $M$ push and pop operations takes time proportional to $M$. 

<table>
<thead>
<tr>
<th></th>
<th>best</th>
<th>worst</th>
<th>amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>construct</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>push</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>pop</td>
<td>1</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

running time for doubling stack with N items
Tradeoffs. Can implement a stack with either dynamic array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.
• Every operation takes constant time in the worst case.
• Uses extra time and space to deal with the links.

Dynamic-array implementation.
• Every operation takes constant amortized time.
• Less wasted space.
• stacks
• resizing arrays
• queues
• generics
• iterators
• applications
**Queue API**

public class QueueOfStrings

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueueOfStrings()</td>
<td>create an empty queue</td>
</tr>
<tr>
<td>void enqueue(String s)</td>
<td>insert a new item onto queue</td>
</tr>
<tr>
<td>String dequeue()</td>
<td>remove and return the item least recently added</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the queue empty?</td>
</tr>
<tr>
<td>int size()</td>
<td>number of items on the queue</td>
</tr>
</tbody>
</table>
Queue dequeue: linked-list implementation

**Remark.** Identical code to linked-list stack \texttt{pop()}. 

```
String item = first.item;

first = first.next;
```

```
return item;
```
Queue enqueue: linked-list implementation

save a link to the last node

```java
Node oldlast = last;
```

create a new node for the end

```java
Node last = new Node();
last.item = "not";
last.next = null;
```

link the new node to the end of the list

```java
oldlast.next = last;
```
public class QueueOfStrings
{
    private Node first, last;

    private class Node
    {  /* same as in StackOfStrings */  }

    public boolean isEmpty()
    {  return first == null;  }

    public void enqueue(String item)
    {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else oldlast.next = last;
    }

    public String dequeue()
    {
        String item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
Queue: dynamic array implementation

Array implementation of a queue.
- Use array $q[]$ to store items in queue.
- enqueue(): add new item at $q[tail]$.
- dequeue(): remove item from $q[head]$.
- Update head and tail modulo the capacity.
- Add dynamic resizing.
- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, etc.? 

Attempt 1. Implement a separate stack class for each type.
• Rewriting code is tedious and error-prone.
• Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5.
We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, etc.?

**Attempt 2.** Implement a stack with items of type `Object`.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```java
StackOfObjects s = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, etc.?

Attempt 3. Java generics.
• Avoid casting in client.
• Discover type mismatch errors at compile-time instead of run-time.

```java
Stack<Apple> s = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

Guiding principles. Welcome compile-time errors; avoid run-time errors.
Generic stack: linked-list implementation

public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }  

    public void push(String item)
    {  
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}

public class Stack<Item>
{
    private Node first = null;
    private class Node
    {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }  

    public void push(Item item)
    {  
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.

Java solution. Make stack implement the Iterator interface.
Iterators

Q. What is an **Iterable**?
A. Has a method that returns an **Iterator**.

Q. What is an **Iterator**?
A. Has methods `hasNext()` and `next()`.

Q. Why make data structures **Iterable**?
A. Java supports elegant client code.

```
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    void remove();
}
```

```
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

```
for (String s : stack)
    StdOut.println(s);
```

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() { return current != null; }
        public void remove()     {
            /* not supported */
        }
        public Item next()
        {
            Item item = current.item;
            current   = current.next;
            return item;
        }
    }
}
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item> {
    ...

    public Iterator<Item> iterator() { return new ArrayIterator(); }

    private class ArrayIterator implements Iterator<Item> {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove() { /* not supported */ }
        public Item next() { return s[--i]; }
    }
}
- stacks
- resizing arrays
- queues
- generics
- iterators

- APIs and applications
Java collections library

List interface. `java.util.List` is API for ordered collection of items.

```java
public interface List<Item> implements Iterable<Item> {
    List() create an empty list
    boolean isEmpty() is the list empty?
    int size() number of items
    void add(Item item) append item to the end
    Item get(int index) return item at given index
    Item remove(int index) return and delete item at given index
    boolean contains(Item item) does the list contain the given item?
    Iterator<Item> iterator() iterator over all items in the list
    ...
}
```

Implementations. `java.util.ArrayList` uses dynamic array;
`java.util.LinkedList` uses linked list.
• Class LinkedList<E>: Generic class implementing doubly linked lists.

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

• Often linked lists are implemented directly in applications for transparency or to allow special features.
Consider the following code fragments accessing a list A.

1. for (int i=0; i<n; i++) A.add(i); // Add to end
2. Iterator<Integer> it = A.iterator();
   while (it.hasNext()) sum += it.next(); // Iterate
3. for (int i=0; i<n; i++) sum += A.get(i); // Iterate using get()
4. for (int i=0; i<n; i++) A.add(0,i); // Add to start

For each of the following implementations of A, how do you predict that the running time will differ (for large n)?

List<Integer> A = new LinkedList<Integer>();
List<Integer> A = new ArrayList<Integer>();
• Class `ArrayList<E>`: Generic class implementing unbounded arrays.

`Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.`

• Interface `RandomAccess`

`The best algorithms for manipulating random access lists (such as ArrayList) can produce quadratic behavior when applied to sequential access lists (such as LinkedList). Generic list algorithms are encouraged to check whether the given list is an instance of this interface.`
Stack application: Function calls

How a compiler implements a function.

- Function call: **push** local environment and return address.
- Return: **pop** return address and local environment.

Recursive function. Function that calls itself.

**Note.** Can always use an explicit stack to remove recursion.

```c
static int gcd(int p, int q) {
    if (q == 0) return p;
    else return gcd(q, p % q);
}
```

`gcd (216, 192)`

- `p = 216, q = 192`
- `p = 192, q = 24`
- `p = 24, q = 0`

`gcd (192, 24)`

`gcd (24, 0)`
Goal. Evaluate infix expressions.

Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parenthesis: ignore.
- Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!
Queue applications

Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.
Summary of points

• The choice of data structure can have a major impact on the complexity of an algorithm.
• Saw several representations of sequences (one more important representation to come later in course).
• Best choice depends on the usage pattern.
• Amortized analysis is needed to understand the performance of some data structures.
• You are only expected to be able to use amortized bounds in algorithm analysis.
Course goals

• The choice of data structure can have a major impact on the complexity of an algorithm.
  - Saw several representations of sequences (one more important representation to come later in course).
  - Best choice depends on the usage pattern.

• Amortized analysis is needed to understand the performance of some data structures.
  - You are only expected to be able to use amortized bounds in algorithm analysis, not derive amortized bounds on your own.
Next week

We will next consider fundamental algorithms for sorting.