2.5 Stacks and Queues

Client, Implementation, Interface

Separate interface and implementation so as to:
- Build layers of abstraction.
- Reuse software.
- Ex: stack, queue, symbol table.

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

Stacks and Queues

Fundamental data types.
- Set of operations (add, remove, test if empty) on generic data.
- 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: Registrar’s line.

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

- **push()**: Insert a new item onto stack.
- **pop()**: Delete and return the item most recently added.
- **isEmpty()**: Is the stack empty?

```
public static void main(String[] args) {
    StringStack stack = new StringStack();
    while(!StdIn.isEmpty()) {
        String s = StdIn.readString();
        stack.push(s);
    }
    while(!stack.isEmpty()) {
        String s = stack.pop();
        System.out.println(s);
    }
} // a sample stack client
```

Stack Push: Linked List Implementation

```
public class StringStack {
    private Node first = null;

    private class Node {
        String item;
        Node next;
    }

    public boolean isEmpty() { return first == null; }
    public void push(String item) {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }
    public String pop() {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```
Stack: Array Implementation

Array implementation of a stack:
- Use array to store N items on stack.
- push() add new item at s[N].
- pop() remove item from s[N-1].

<table>
<thead>
<tr>
<th>s[]</th>
<th>it</th>
<th>was</th>
<th>the</th>
<th>best</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Stack Array Implementation: Resizing

How to resize array? Increase size by one if the array is full.

Thrashing.
- Increasing the size of an array involves copying all of the elements to a new array.
- Inserting N elements: time proportional to 1 + 2 + ... + N = N²/2. 
  N = 1 million ⇒ infeasible.

Stack: Array Implementation

public class StringStack {
    private String[] s;
    private int N = 0;

    public StringStack() {
        this(8);  
    }

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

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

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

    public String pop() {
        String item = s[N-1];
        s[N-1] = null;
        N--;
        return item;
    }

    private void resize() {
        String[] dup = new String[2*N];
        for (int i = 0; i < N; i++)
            dup[i] = s[i];
        s = dup;
    }
}

Consequence. Inserting N items takes time proportional to N (not N²).
Stack Implementations: Array vs. Linked List

Stack implementation tradeoffs. Can implement with either array or linked list, and client can use interchangeably. Which is better?

Array.
- Most operations take constant time.
- Expensive re-doubling operation every once in a while.
- Any sequence of N operations (starting from empty stack) takes time proportional to N. "amortized" bound

Linked list.
- Grows and shrinks gracefully.
- Every operation takes constant time.
- Uses extra space and time to deal with references.

Queue

Queue operations.
- enqueue() Insert a new item onto queue.
- dequeue() Delete and return the item least recently added.
- isEmpty() Is the queue empty?

```java
public static void main(String[] args) {
    StringQueue q = new StringQueue();
    q.enqueue("Vertigo");
    q.enqueue("Just Lose It");
    q.enqueue("Pieces of Me");
    q.enqueue("Pieces of Me");
    System.out.println(q.dequeue());
    q.enqueue("Drop It Like It's Hot");
    while(!q.isEmpty())
        System.out.println(q.dequeue());
}
```

Dequeue: Linked List Implementation

```java
first = first.next;
item = first.item;
return item;
```

Enqueue: Linked List Implementation

```java
first = first.next;
x = new Node();
x.item = item;
x.next = null;
last = x;
last.next = x;
```
Parameterized Data Types

We implemented: StringStack, StringQueue.

We also want: URLStack, CustomerQueue, 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.
### Generics

**Generics.** [since Java 1.5] Parameterize stack by a single type.
- Avoid casting in both client and implementation.
- 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 principle.** Run-time errors are much harder to identify and fix than compile-time errors.

---

### Stack of Objects

**We implemented:** `StringStack, StringQueue`.

**We also want:** `URLStack, CustomerQueue, 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
Stack s = new Stack();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) s.pop();  // compile-time error
```

---

### Generic Stack: Linked List Implementation

```java
public class Stack<Item> {   private Node first;
    
    private class Node {  
        Item item;
        Node next;
    }   
    
    public boolean isEmpty() { return first == null; }
    
    public void push(Item item) {   
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
    }   
    
    public Item pop() {   
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

---

### Generic Stack: Array Implementation

The way it should be.

```java
public class ArrayStack<Item> {   private Item[] a;
    private int N;
    
    public ArrayStack(int capacity) {     
        a = new Item[capacity];
    }   
    
    public boolean isEmpty() { return N == 0; }
    
    public void push(Item item) {   
        a[N++] = item;
    }   
    
    public Item pop() { 
        return a[--N];
    }
}
```

---

### Generics

**Generics.** Avoid parameterized type name, but must use consistently.
Generic Stack: Array Implementation

The way it is: **an ugly cast** in the implementation.

```java
public class ArrayStack<Item> {
    private Object[] a;
    private int N;
    public ArrayStack(int capacity) {
        a = ... a[--N];
    }
    public boolean isEmpty() { return N == 0; }
    public void push(Item item) {
        a[N++] = item;
    }
    public Item pop() {
        return (Item) a[--N];
    }
}
```

Syntactic sugar.

Wrapper type.
- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Casts are still done behind the scenes.

```java
Stack<Integer> s = new Stack<Integer>();
s.push(17); // s.push(new Integer(17));
int a = s.pop(); // int a = ((Integer) s.pop()).intValue();
```

Stack Applications

Real world applications.
- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.
Function Calls

How a compiler implements functions.
- 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 a stack to remove recursion.

Postfix Evaluation

```java
public class Postfix {
    public static void main(String[] args) {
        Stack<Integer> st = new Stack<Integer>();
        while (!StdIn.isEmpty()) {
            String string = StdIn.readString();
            if (s.equals("+")) st.push(st.pop() + st.pop());
            else if (s.equals("*")) st.push(st.pop() * st.pop());
            else st.push(Integer.parseInt(s));
        }
        System.out.println(st.pop());
    }
}
```

Postfix Notation

- Put operator after operands in expression.
- Use stack to evaluate.
  - operand: push it onto stack
  - operator: pop operands, push result
- Systematic way to save intermediate results and avoid parentheses.

Infix to postfix algorithm.
- Left parentheses: ignore.
- Right parentheses: pop and print.
- Operator: push.
- Integer: print.

Infix to Postfix

```java
public class Infix {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if (s.equals("+")) System.out.print(stack.pop() + " ");
            else if (s.equals("*")) stack.push(s);
            else if (s.equals("^")) System.out.print("\n");
            else if (s.equals("/")) stack.push(s);
            else if (s.equals("-")) stack.push(s);
            else System.out.print(s + " ");
        }
    }
}
```
Josephus Problem

Flavius Josephus. [first century]
- Band of 41 Jewish rebels trapped in cave by Romans.
- Preferring suicide to capture, rebels formed a circled and killed every 3rd remaining person until no one was left.

Q. Where should you stand to be the last survivors?
A. 31.

M/M/1 Queueing Model

M/M/1 queue.
- Customers arrive at rate of \( \lambda \) per minute.
- Customers are serviced at rate of \( \mu \) per minute.
- Inter-arrival time obeys exponential distribution: \( \Pr[X \leq x] = 1 - e^{-\lambda x} \)

Q. How long does a customer wait in queue?

Queue Applications

Some applications.
- iTunes playlist.
- Breadth first search.
- Data buffers (iPod, TiVo).
- Graph processing (stay tuned).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.
- Traffic analysis of Lincoln tunnel.
- Waiting times of customers in McDonalds.
- Determining number of cashiers to have at a supermarket.

public class Josephus {
    public static void main(String[] args) {
        int M = Integer.parseInt(args[0]);
        int N = Integer.parseInt(args[1]);
        Queue<Integer> q = new Queue<Integer>();
        for (int i = 1; i <= N; i++)
            q.enqueue(i);
        while (!q.isEmpty()) {
            for (int i = 0; i < M - 1; i++)
                q.enqueue(q.dequeue());
            System.out.print(q.dequeue() + " ");
        }
    }
}

% java Josephus 5 9
5 1 7 4 3 6 9 2 8
Summary

Stacks and queues are fundamental ADTs.
- Array implementation.
- Linked list implementation.
- Different performance characteristics.
- Many applications.

M/M/1 Queue: Implementation

```java
public class MM1Queue {
    public static void main(String[] args) {
        double lambda = Double.parseDouble(args[0]);
        double mu = Double.parseDouble(args[1]);
        Queue<Double> q = new Queue<Double>();
        double nextArrival = Random.exponential(lambda);
        double nextDeparture = Random.exponential(mu);
        while (true) {
            if (nextArrival < nextDeparture) { // arrival
                q.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            } else { // departure
                if (!q.isEmpty()) {
                    double wait = nextDeparture - q.dequeue();
                    // add waiting time to histogram
                }
                nextDeparture += Random.exponential(mu);
            }
        }
    }
}
```

M/M/1 Queue Analysis

**Remark.** As service rate approaches arrival rate, service goes to h***.

**Theorem.** Average time a customer spends in system = 1 / (μ - λ).

Summary

ADTs enable modular programming.
- Separate compilation.
- Split program into smaller modules.
- Different clients can share the same ADT.

ADTs enable encapsulation.
- Keep modules independent (include `main` in class for testing).
- Can substitute different classes that implement same interface.
- No need to change client.

Issues of ADT design.
- Feature creep.
- Formal specification problem.
- Implementation obsolescence.