2.6 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.

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.

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?

LIFO = "last in first out"

FIFO = "first in first out"
Stack operations:

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

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

<table>
<thead>
<tr>
<th>( N = 1 \text{ million} )</th>
<th>Inserting ( N ) elements: time proportional to ( 1 + 2 + \ldots + N )</th>
<th>( N^2/2 ).</th>
</tr>
</thead>
</table>

How to resize array? Increase size \( s \) 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 \( 1 + 2 + \ldots + N = N^2/2 \).
  \( N = 1 \text{ million} \) is infeasible.

Stack Array Implementation: Resizing

How to resize array? Increase size \( s \) 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 \( 1 + 2 + \ldots + N = N^2/2 \).
  \( N = 1 \text{ million} \) is infeasible.

Stack Array Implementation: Dynamic Resizing

How to resize array? Use repeated doubling: if \( s \) is not big enough, create new array of twice the size, and copy items.

Consequence. Inserting \( N \) items takes time proportional to \( N \) (not \( N^2 \)).
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.

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

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

Dequeue: Linked List Implementation

item = first.item;

first = first.next;

return item;

Enqueue: Linked List Implementation

x = new Node();
x.item = item;
x.next = null;

last = x;

first = first.next;

last.next = x;

last = 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.
**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.

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

**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.

```
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;
    }
}
```

**Guiding principle.** Run-time errors are much harder to identify and fix than compile-time errors.

```
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];
    }
}
```
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 = new Object[capacity];
    }

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

    public void push(Item item) {
        a[N++] = item;
    }

    public Item pop() {
        return (Item) a[--N];
        // the ugly cast
    }
}
```

Generic stack implementation. Allows objects, not primitive types.

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

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

```
% java Postfix
1 2 3 4 5 * + 6 * * +
277  infix expression: (1+(((2*((3+(4*5))*6)))

% java Postfix
7 16 * 5 + 16 * 3 + 16 * 1 +
30001 convert 7531 from hex to decimal using Horner's method
```

Note. Postfix notation.
- Systematic way to save intermediate results and avoid parentheses.
- Simple way to evaluate expressions.

Infix to Postfix

```
% java Postfix
1 2 3 4 5 * + 6 * * +
277  infix expression: (1+(((2*((3+(4*5))*6)))

% java Infix
{ 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) } *
2 3 4 + 5 6 * * +
```

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

```
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("")") System.out.print(" ");
            else if (s.equals("*")) stack.push(s);
            else if (s.equals("")") stack.push(s);
            else System.out.print(s + ");
        }
    }
}
```

Postfix to Infix

```
% java Infix
{ 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) } *
2 3 4 + 5 6 * * +
```
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.

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.

public class Josephus {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int M = Integer.parseInt(args[1]);
        Queue<Integer> q = new ArrayDeque<>();
        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

M/M/1 Queuing Model

M/M/1 queue.
- Customers arrive at rate of λ per minute.
- Customers are serviced at rate of µ per minute.
- Inter-arrival time obeys exponential distribution: Pr[X ≤ x] = 1 − e^−λx

Q. How long does a customer wait in queue?
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) {
                q.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            } else {
                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.