Data Types and Data Structures

Collections

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. [LIFO = last in first out]
- Remove the item most recently added.
- Ex: cafeteria trays, Web surfing.

Queue. [FIFO = first in, first out]
- Remove the item least recently added.
- Ex: Registrar’s line.

Symbol table.
- Remove the item with a given key.
- Ex: Phone book.

Stacks
Stack API

```java
public class *StackOfStrings {
    *StackOfStrings() create an empty stack
    boolean isEmpty() is the stack empty?
    void push(String item) push a string onto the stack
    String pop() pop the stack
}
```

Stack Client Example 1: Reverse

```java
public class Reverse {
   public static void main(String[] args) {
      StackOfStrings stack = new StackOfStrings();
      while (!StdIn.isEmpty()) {
         stack.push(StdIn.readString());
      }
      while (!stack.isEmpty()) {
         StdOut.println(stack.pop());
      }
   }
}
```

Stack Client Example 2: Test Client

```java
public static void main(String[] args) {
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty()) {
        String s = StdIn.readString();
        if (s.equals("-"))
            StdOut.println(stack.pop());
        else
            stack.push(s);
    }
}
```

Stack: Array Implementation

```java
public class ArrayStackOfStrings {
    private String[] a;
    private int N = 0;
    public ArrayStackOfStrings(int max) { a = new String[max]; }  
    public boolean isEmpty() { return (N == 0); }
    public void push(String item) { a[N++] = item; }
    public String pop() { return a[--N]; }
}
```

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

How big to make array? [stay tuned]

Stack contents just before first pop operation

Temporary solution: make client provide capacity

Stack and array contents after 4th push operation

Stack contents when standard input is empty

% more tiny.txt
it was the best of times
% java Reverse < tiny.txt
times of best the was it
### Array Stack: Test Client Trace

<table>
<thead>
<tr>
<th>StdIn</th>
<th>StdOut</th>
<th>N</th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>push</td>
<td>to</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>be</td>
<td>2</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>3</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
<td>5</td>
<td>be</td>
</tr>
<tr>
<td>pop</td>
<td>to</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>be</td>
<td>5</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>-be</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>-not</td>
<td>3</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>-be</td>
<td>1</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>is</td>
<td>2</td>
<td>is</td>
</tr>
</tbody>
</table>

### Array Stack: Performance

**Running time.** Push and pop take constant time.

**Memory.** Proportional to client-supplied capacity, not number of items.

**Problem.**
- API does not call for capacity (bad to change API).
- Client might use multiple stacks.
- Client might not know what capacity to use.

**Challenge.** Stack implementation where size is not fixed ahead of time.

### Linked Lists

#### Sequential vs. Linked Allocation

**Sequential allocation.** Put object one after another.
- TOY: consecutive memory cells.
- Java: array of objects.

**Linked allocation.** Include in each object a link to the next one.
- TOY: link is memory address of next object.
- Java: link is reference to next object.

**Key distinctions.**
- Array: random access, fixed size.
- Linked list: sequential access, variable size.

![Linked List Diagram]
Linked list.
- A recursive data structure.
- An item plus a pointer to another linked list (or empty list).
- Unwind recursion: linked list is a sequence of items.

Node data type.
- A reference to a String.
- A reference to another Node.

```java
public class Node {
    private String item;
    private Node next;
}
```

Building a Linked List

```
Node third = new Node();
third.item = "Carol";
third.next = null;
Node second = new Node();
second.item = "Bob";
second.next = third;
Node first = new Node();
first.item = "Alice";
first.next = second;
```

Stack Push: Linked List Implementation

```
first = new Node();
best = first;
the = second;
was = third;
```

Stack Pop: Linked List Implementation

```
String item = first.item;
first = first.next;
```

```java
Node second = first;
```
public class LinkedStackOfStrings {
    private Node first = null;

    private class Node {
        private String item;
        private 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;
    }
}

Linked List Stack: Test Client Trace

Running time. Push and pop take constant time.

Memory. Proportional to number of items in stack.

Stack Data Structures: Tradeoffs

Two data structures to implement Stack data type.

Array.
- Every push/pop operation takes constant time.
- But... must fix maximum capacity of stack ahead of time.

Linked list.
- Every push/pop operation takes constant time.
- But... uses extra space and time to deal with references.
List Processing Challenge 1

Q. What does the following code fragment do?

```java
for (Node x = first; x != null; x = x.next) {
    StdOut.println(x.item);
}
```

Parameterized Data Types

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, ...

Strawman. 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

Parameterize stack by a single type.

```java
Stack<Apple> stack = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b); // compile-time error
a = stack.pop();
```

Sample client

"stack of apples" can't push an orange onto a stack of apples

Generic Stack: Linked List Implementation

```java
public class Stack<Item> {
    private Node first = null;

    private class Node {
        private Item item;
        private 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;
    }
}
```

Parameterized type name (chosen by programmer)

Stack Applications

Real world applications.

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

Autoboxing

Generic stack implementation. Only permits reference types.

Wrapper type.

- Each primitive type has a wrapper reference type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast from primitive type to wrapper type.

Autounboxing. Automatic cast from wrapper type to primitive type.

```java
Stack<Integer> stack = new Stack<Integer>();
stack.push(17); // autobox (int -> Integer)
int a = stack.pop(); // autounbox (Integer -> int)
```
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 an explicit stack to remove recursion.

gcd (216, 192)

static int gcd (int p, int q)
{
    if (q == 0) return p;
    else return gcd(q, p % q);
}
gcd (192, 24)

static int gcd (int p, int q)
{
    if (q == 0) return p;
    else return gcd(q, p % q);
}
gcd (24, 0)

Arithmetic Expression Evaluation

Goal. Evaluate infix expressions.

Two stack algorithm. [E. W. Dijkstra]
- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

Extensions. More ops, precedence order, associativity, whitespace.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + ( (2 + 3) * (4 * 5) )</td>
<td>101.0</td>
</tr>
<tr>
<td>1 + ( (2 + 3) * (4 * 5) )</td>
<td>101</td>
</tr>
<tr>
<td>1 + (5 * (4 * 5))</td>
<td>101</td>
</tr>
<tr>
<td>1 + (5 * 20)</td>
<td>101</td>
</tr>
<tr>
<td>1 + 100</td>
<td>101</td>
</tr>
</tbody>
</table>

Correctness

Why correct? When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

So it's as if the original input were:

Repeating the argument:

Extensions. More ops, precedence order, associativity, whitespace.
Observation 1. Remarkably, the 2-stack algorithm computes the same value if the operator occurs after the two values.

\[
(1 \ (2 \ 3 +) \ (4 \ 5 \ *) \ *) +)
\]

Observation 2. All of the parentheses are redundant!

\[
1 \ 2 \ 3 + 4 \ 5 * * +
\]

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, ...

---

Queue API

```java
public class Queue<Item>
{
    Queue<Item>()
    boolean isEmpty()
    void enqueue(Item item)
    Item dequeue()
    int length()
}
```

enqueue \[ \rightarrow \] dequeue

```java
public static void main(String[] args)
{
    Queue<String> q = new Queue<String>();
    q.enqueue("Vertigo");
    q.enqueue("Just Lose It");
    q.enqueue("Pieces of Me");
    while(!q.isEmpty())
    {
        StdOut.println(q.dequeue());
    }
}
```

---

Enqueue: Linked List Implementation

```
Node oldlast = last;
Node last = new Node();
last.item = "of";
last.next = null;
oldlast.next = last;
```
Dequeue: Linked List Implementation

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

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

```java
public Item dequeue() { 
    Item item = first.item; 
    first = first.next; 
    if (isEmpty()) last = null; 
    return item; 
}
```

Queue: Linked List Implementation

```java
public class Queue<Item> { 
    private Node first, last; 
    private class Node { Item item; Node next; }
```

Queue 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.
- Guitar string.
- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

M/D/1 Queuing Model

- Customers are serviced at fixed rate of $\mu$ per minute.
- Customers arrive according to Poisson process at rate of $\lambda$ per minute.

inter-arrival time has exponential distribution

\[
Pr[X \leq x] = 1 - e^{-\lambda x}
\]

Arrival rate $\lambda$  
Infinite queue  
Departure rate $\mu$  
Server

Q. What is average wait time $W$ of a customer?
Q. What is average number of customers $L$ in system?
Event-Based Simulation

public class MD1Queue {
   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 = StdRandom.exp(lambda);
      double nextService = nextArrival + 1/mu;
      while(true) {
         if (nextArrival < nextService) {
            q.enqueue(nextArrival);
            nextArrival += StdRandom.exp(lambda);
         } else {
            double wait = nextService - q.dequeue();
            // add waiting time to histogram
            if (q.isEmpty()) nextService = nextArrival + 1/mu;
            else nextService = nextService + 1/mu;
         }
      }
   }
}

M/D/1 Queue Analysis

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

Queueing theory. W = λ μ(μ−λ) + 1 μ, L = λ W

Summary

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

Many applications.