4.3 Stacks and Queues

Data Types and Data Structures

Data types.
- Set of values.
- Set of operations on those values.
- Some are built in to Java: int, double, char, ...
- Most are not: Complex, Picture, Stack, Queue, Graph, ...

Data structures.
- Represent data or relationships among data.
- Some are built into Java: arrays, String, ...
- Most are not: linked list, circular list, tree, sparse array, graph, ...

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

Operations on a pushdown stack
Stack API

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

Stack Client Example 1: Reverse

```java
% more tiny.txt
it was the best of times
% java Reverse < tiny.txt
times of best the was it
```

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

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

```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 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></td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>push</td>
<td>to</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>or</td>
<td>3</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>not</td>
<td>4</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>to</td>
<td>5</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>pop</td>
<td>–</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td>be</td>
<td>5</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>–</td>
<td>-</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td>not</td>
<td>3</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>that</td>
<td>4</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>that</td>
<td>3</td>
<td>to</td>
</tr>
<tr>
<td>-</td>
<td>or</td>
<td>2</td>
<td>to</td>
</tr>
<tr>
<td>-</td>
<td>be</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
<td>to</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.
From the point of view of a particular object:
all of these structures look the same.

Multiply-linked data structures. Many more possibilities.

Linked Structures can Become Intricate

Building a Linked List

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.

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

Alice ← Bob ← Carol ↷ null
item next

special pointer value null terminates list

first C4 "Alice"
second CA C0 null
third CB CD CE CF

main memory

official Florida presidential ballot
follow the arrow and punch the appropriate dot.

bush buchanan gore nader

first second third

13

15

16

14

15

16
Stack Push: Linked List Implementation

first
  
  best
  the
  was
  it

Node second = first;

first
  
  second
  
  best
  the
  was
  it

first = new Node();

first
  
  second
  
  of
  best
  the
  was
  it

Stack Pop: Linked List Implementation

first
  
  best
  the
  was
  it

String item = first.item;

first = first.next;

Stack: Linked List Implementation

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
Linked List Stack: Performance

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 take 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);
}
```

List Processing Challenge 2

Q. What does the following code fragment do?

```java
Node last = new Node();
last.item = StdIn.readString();
last.next = null;
Node first = last;
while (!StdIn.isEmpty()) {
    last.next = new Node();
    last = last.next;
    last.item = StdIn.readString();
    last.next = null;
}
```
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

Generics. Parameterize stack by a single type.

```
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;
   }
}
```

Generic Stack: Linked List Implementation

```
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();
```

parameterized type name (chosen by programmer)

can't push an orange onto a stack of apples
**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.

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

**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.

**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!
public class Evaluate {
    public static void main(String[] args) {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if (s.equals("("))
            else if (s.equals("+")) ops.push(s);
            else if (s.equals("*")) ops.push(s);
            else if (s.equals("") { String op = ops.pop();
                if (op.equals("+")) vals.push(vals.pop() + vals.pop());
                else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
                else vals.push(Double.parseDouble(s));
            }
            StdOut.println(vals.pop());
        }
    }
}

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.

Queues
Queue API

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

    private class Node {
        Item item;
        Node next;
    }

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

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

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

Dequeue: Linked List Implementation

```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());
}
```

Queue: Linked List Implementation
Queue Applications

Some 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

M/D/1 queue.
- 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.
  - $\Pr[X \leq x] = 1 - e^{-\lambda x}$

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

Event-Based Simulation

```java
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();
            if (q.isEmpty()) nextService = nextArrival + 1/mu;
            else nextService = nextService + 1/mu;
         }
      }
   }
}
```
Observation. As service rate approaches arrival rate, service goes to h***.

Queueing theory.

\[ W = \frac{\lambda}{2\mu(\mu - \lambda)} + \frac{1}{\mu}, \quad L = \frac{\lambda}{W} \]
Little's law

see ORFE 309

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

Many applications.