CS 112 Introduction to Programming (Spring 2012)

Lecture #26: Stacks and Queues

Zhong Shao
Department of Computer Science
Yale University
Office: 314 Watson
http://flint.cs.yale.edu/cs112

Acknowledgements: some slides used in this class are taken directly or adapted from those accompanying the textbook, Introduction to Programming in Java, by Robert Sedgewick and Kevin Wayne (Copyright 2002-2010).

Data Types and Data Structures

Data types:
- Set of values and operations on those values.
- Some are built into the Java language: int, double[], String, ...
- Most are not: Complex, Picture, Stack, Queue, ST, Graph, ...

Data structures:
- Represent data or relationships among data.
- Some are built into Java language: arrays.
- 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: Commons line at 12:30 MW.

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

Stacks
Stack API

```java
public class *StackOfStrings*

*StackOfStrings* create a new 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()) {
            String s = StdIn.readString();
            stack.push(s);
        }
        while (!stack.isEmpty()) {
            String s = stack.pop();
            StdOut.println(s);
        }
    }
}
```

% 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);
    }
}
```

% more test.txt
to be or not to - that - - - is
% java StackOfStrings < test.txt
to be not that or be

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]; }
}
```

Temporary solution: make client provide capacity

stack and array contents after 4th push operation
3/18/12
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>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>push</td>
<td>1</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>be</td>
</tr>
<tr>
<td>or</td>
<td>3</td>
<td>3</td>
<td>be</td>
</tr>
<tr>
<td>not</td>
<td>4</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td>to</td>
<td>5</td>
<td>5</td>
<td>be</td>
</tr>
<tr>
<td>pop</td>
<td>4</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>5</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>3</td>
<td>be</td>
</tr>
<tr>
<td>that</td>
<td>4</td>
<td>4</td>
<td>to</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>be</td>
</tr>
<tr>
<td>-</td>
<td>be</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td>is</td>
<td>2</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 take capacity as argument (bad to change API).
- Client might not know what capacity to use.
- Client might use multiple stacks.

Challenge. Stack where capacity is not known ahead of time.

Linked Lists

Sequential vs. Linked Allocation

Sequential allocation. Put items one after another.
- Java: String; array of objects.

Linked allocation. Include in each object a link to the next one.
- Java: link is reference to next item.

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.

Singly-Linked Data Structures

Linked Lists

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;
}

from

Building a Linked List

Stack Push: Linked List Implementation
17 Stack Pop: Linked List Implementation

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

18 Stack: Linked List Implementation

```java
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;
    }
}
```

19 Linked List Stack: Test Client Trace

20 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.
  - Memory is proportional to number of items on stack.
  - **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, StackOfVans, 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.

```
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.

Parameterized type

Parameterized type name (chosen by programmer)

Generic Stack: Linked List Implementation

```
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

Parameterized type name (chosen by programmer)

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.

**Auto-unboxing.** Automatic cast from wrapper type to primitive type.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);  // autobox (int -> Integer)
int a = stack.pop();  // auto-unbox (Integer -> int)
```

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.
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.

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.

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.

1 + (2 - 3 + 4) * 5 = sqrt(6*6 + 7*7)
Stack-Based Programming Languages

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, ...

Summary

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

Many applications.

Queues

Queue API

```java
public class Queue<Item>

    public Queue() create an empty queue
    boolean isEmpty() is the queue empty?
    void enqueue(Item item) enqueue an item
    Item dequeue() dequeue an item
    int length() queue length
```

```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

1. first
2. last
3. oldlast

last = new Node();
last.item = "of";
last.next = null;
oldlast.next = last;

Dequeue: Linked List Implementation

1. first
2. last
3. oldlast

String item = first.item;
first = first.next;
first.item = item;
first.next = null;
return item;

Queue: Linked List Implementation

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

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 Queueing 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.

**Arrival rate $\lambda$**  
**Departure rate $\mu$**

![Queueing Diagram](image)

**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();
                // 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 high.

**Queueing theory.**

\[
W = \frac{1}{2\mu(\mu - A)} + \frac{1}{\mu} \quad L = \lambda W
\]

**Little's law**
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];
    }
}