4.3 Stacks and Queues
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.
- Ex: cafeteria trays, Web surfing.

Queue.
- Remove the item least recently added.
- Ex: Registrar's line.

LIFO = "last in first out"

FIFO = "first in first out"
Stacks

Operations on a pushdown stack
Stack API

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

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());
    }
}
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 $a[N-1]$.

```
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: 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></td>
</tr>
<tr>
<td><strong>push</strong></td>
<td>to</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>be</td>
<td>2</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>3</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>to</td>
<td>5</td>
<td>to</td>
</tr>
<tr>
<td><strong>pop</strong></td>
<td>-</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>be</td>
<td>5</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>3</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>that</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>3</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>2</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>2</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>is</td>
<td>2</td>
<td>to</td>
</tr>
</tbody>
</table>
Array Stack: Performance

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

**Memory.** Proportional to $\text{max}$. 

**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 Lists

Linked list.
- A recursive data structure.
- A 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;
}
```

```plaintext
first

Alice | Bob | Carol | null

item next
```

special value null terminates list
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;

Address
first  second  third
Alice   Bob     Carol   null

Value
addr  Value
C0     "Carol"
C1     null
C2     "Alice"
C3     CA
C4     C0
C5     CA
C6     "Bob"
C7     C0
C8     -
C9     -
CA     -
CB     -
CC     -
CD     -
CE     -
CF     -

main memory
**Iteration.** Idiom for traversing a null-terminated linked list.

```
for (Node x = first; x != null; x = x.next) {
    StdOut.println(x.item);
}
```
Stack Push: Linked List Implementation

first

second = first;

first

second

first = new Node();

first

second

first.item = item;
first.next = second;
Stack Pop: Linked List Implementation

first

of → best → the → was → it

item = first.item;

first = first.next;

garbage-collected

first

best → the → was → it

return item;
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;
    }
}

"inner class"
Linked List Stack: Trace

push

be
or
not
to

pop

be
not
that

Stack Implementations: Tradeoffs

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.
Parameterized Data Types
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
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;
    }
}
**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)
```
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.

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

```
gcd (216, 192)
gcd (192, 24)
gcd (24, 0)
p = 216, q = 192
p = 192, q = 24
p = 24, q = 0
```

```
p = 24, q = 0
p = 192, q = 24
p = 216, q = 192
```
**Goal.** Evaluate infix expressions.

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

**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.

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

So it's as if the original input were:

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

Repeating the argument:

\[
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
\]

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, ...
Queue API

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

create an empty queue
is the queue empty?
enqueue an item
dequeue an item
queue length

```
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");
    q.enqueue("Pieces of Me");
    while(!q.isEmpty())
        StdOut.println(q.dequeue());
}
```
Enqueue: Linked List Implementation

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

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

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

Q. What is average wait time \( W \) of a customer?
Q. What is average number of customers \( L \) in system?
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;
            }
        }
    }
}

Event-Based Simulation
**Observation.** As service rate approaches arrival rate, service goes to h***.

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

Little’s law

**Queueing theory.**

see ORFE 309
Summary

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

Many applications.
Extra Slides
Doug’s first calculator

ENTER means push

No parens!
**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];
    }
}
```

@#$%^! generic array creation not allowed in Java
Generic Stack: Array Implementation

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

```java
public class ArrayStack<Item> {
    private Item[] a;
    private int N;

    public ArrayStack(int capacity) {
        a = (Item[]) new Object[capacity];
    }

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

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

    public Item pop() {
        return a[--N];
    }
}
```
Array implementation of a queue.

- Use array $q[]$ to store items on queue.
- $\text{enqueue}()$: add new object at $q[\text{tail}]$.
- $\text{dequeue}()$: remove object from $q[\text{head}]$.
- Update head and tail modulo the capacity.
Linked Stuff
Linked Structures Overview

Linked structures. Simple abstraction for customized access to data.

Singly linked structures.
- Linked list.
- Circular linked list.
- Parent-link tree.

Doubly linked structures.
- Binary tree.
- Patricia tries.
- Doubly linked circular list.
Conclusions

Sequential allocation: supports indexing, fixed size.
Linked allocation: variable size, supports sequential access.

Linked structures are a central programming abstraction.
- Linked lists.
- Binary trees.
- Graphs.
- Sparse matrices.

Alice should have done this!
Conclusions

Whew, lots of material in this lecture!

- Pointers are useful, but can be confusion.
- Study these slides and carefully read relevant material.