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, ...

this lecture  TSP assignment  next lecture
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. \([\text{LIFO = last in first out}]\)
- Remove the item most recently added.
- Ex: cafeteria trays, Web surfing.

Queue. \([\text{FIFO = first in, first out}]\)
- 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 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()) {
            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 contents when standard input is empty
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 -- be -- that -- -- is
```
% java StackOfStrings < test.txt
to be not that or be

stack contents just before first pop operation
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: 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></td>
</tr>
</tbody>
</table>

**push**

- to 1 to
- be 2 to be
- or 3 to be or
- not 4 to be or not
- to 5 to be or not to

**pop**

- - 4 to be or not to
- be 5 to be or not be
- - be 4 to be or not be
- - not 3 to be or not be
- that 4 to be or that be
- - that 3 to be or that be
- - or 2 to be or that be
- - be 1 to be or that be
- is 2 to be or not to
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

Official Florida Presidential Ballot
Follow the arrow and Punch the appropriate dot.

Bush
Buchanan
Gore
Nader
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.

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
</table>
| B0   | "Alice"
| B1   | "Bob"
| B2   | "Carol"
| B3   | -     |
| B4   | -     |
| B5   | -     |
| B6   | -     |
| B7   | -     |
| B8   | -     |
| B9   | -     |
| BA   | -     |
| BB   | -     |

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
</table>
| C0   | "Carol"
| C1   | null  |
| C2   | -     |
| C3   | -     |
| C4   | "Alice"
| C5   | CA    |
| C6   | -     |
| C7   | -     |
| C8   | -     |
| C9   | -     |
| CA   | "Bob"
| CB   | C0    |
From the point of view of a particular object:
all of these structures look the same.

Multiply-linked data structures. Many more possibilities.
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.

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

```
first
Alice <-> Bob <-> Carol <-> null

special pointer 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;
Stack Push: Linked List Implementation

```
Node second = first;

first = new Node();

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

```
String item = first.item;
first = first.next;
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;
    }
}
Linked List Stack: Test Client Trace

```
push
- be
- or
- not
- to

pop
- not
- be
- to
- be
- or
- be
is
```
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.
- Memory is proportional to number of items on stack.
- But... uses extra space and time to deal with references.
Q. What does the following code fragment do?

```java
for (Node x = first; x != null; x = x.next) {
    StdOut.println(x.item);
}
```
Q. What does the following code fragment do?

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;
}

```
first
  Alice
    item  next
  → Bob
  → Carol
  → null
```
Parameterized Data Types
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. 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();
```
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)
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(); // 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.

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

**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());
    }
}

% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
Correctness

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

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

So it's as if the original input were:

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

Repeating the argument:

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

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

\[1 + (2 - 3 - 4) \times 5 \times \sqrt{6\times6 + 7\times7}\]
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
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

define the class Queue with methods

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 = "of";
last.next = null;
oldlast.next = last;
```

```
Node oldlast = last;
```

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

```java
String item = first.item;

first = first.next;

return item;
```
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.

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

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

Inter-arrival time has exponential distribution
\[ \Pr[X \leq x] = 1 - e^{-\lambda x} \]
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;
            }
        }
    }
}
**Observation.** As service rate approaches arrival rate, service goes to h***.

**Queueing theory.**

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

*Little’s law*
Extra Slides
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.
Queue: Array Implementation

Array implementation of a queue.
- Use array $q[]$ to store items on queue.
- enqueue(): add new object at $q[tail]$.
- dequeue(): remove object from $q[head]$.
- Update head and tail modulo the capacity.

<table>
<thead>
<tr>
<th>q[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>capacity = 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>