4.3 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.

FIFO = "first in first out"
LIFO = "last in first out"

Stack API

```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());
    }
}
```

```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

    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty())
        stack.push(StdIn.readString());
    while (!stack.isEmpty())
        StdOut.println(stack.pop());
}
```

Stacks

Operations on a pushdown stack.
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: 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.

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

Linked Lists

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;

Traversing a Linked List

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

```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;
    }
}
```

Stack Pop: Linked List Implementation

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

Linked List Stack: Trace
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

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();
```
### 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;
    }
}
```

### 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.

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

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.

Stack-Based Programming Languages

Observation 1. Remarkably, the 2-stack algorithm computes the same value if the operator occurs after the two values.

Observation 2. All of the parentheses are redundant!

Bottom line. Postfix or "reverse Polish" notation.

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

public class Evaluate {
    public static void main(String[] args) {
        ...       
        StdOut.println(vals.pop());
    }
}

1 2 3 + 4 5 * * +

Jan Lukasiewicz
Queues

public class Queue<Item> {
    boolean isEmpty()
    void enqueue(Item item)
    Item dequeue()
    int 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");
    while (!q.isEmpty())
        StdOut.println(q.dequeue());
}

Enqueue: Linked List Implementation

first

Dequeue: Linked List Implementation

first
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.
- Inter-arrival time has exponential distribution:
  $$P[X \leq s] = 1 - e^{-\lambda s}$$

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 h***.

Queueing theory.

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

Little's law

see ORFE 309

**Summary**

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

Many applications.