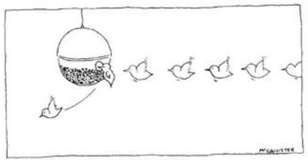
2.6 Stacks and Queues



Drawing by McCallister; © 1977 *The New Yorker* Magazine, Inc.

Robert Sedgewick and Kevin Wayne · · http://www.Princeton.EDU/~cos226

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?

LIFO = "last in first out"

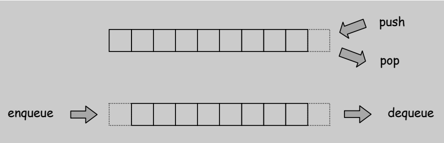
Stack.

- Remove the item most recently added.
- Analogy: cafeteria trays, Web surfing.

FIFO = "first in first out"

Queue.

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



Client, Implementation, Interface

Separate interface and implementation so as to:

- Build layers of abstraction.
- Reuse software.
- Ex: stack, queue, symbol table.

Interface: description of data type, basic operations.
Client: program using operations defined in interface.
Implementation: actual code implementing operations.

Client, Implementation, Interface

Benefits.

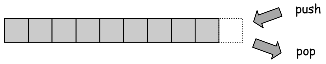
- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ many clients can re-use the same implementation.
- Design: creates modular, re-usable libraries.
- Performance: use optimized implementation where it matters.

Interface: description of data type, basic operations.
Client: program using operations defined in interface.
Implementation: actual code implementing operations.

Stack

Stack operations.

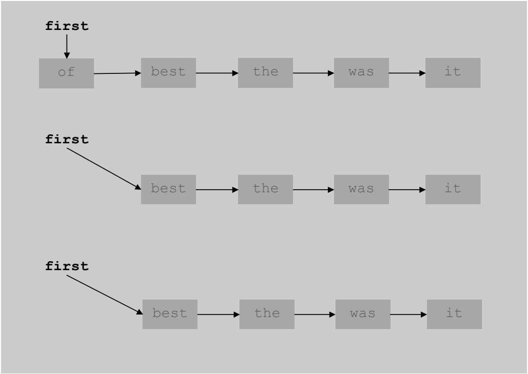
- `push()` Insert a new item onto stack.
- `pop()` Delete and return the item most recently added.
- `isEmpty()` Is the stack empty?



```
public static void main(String[] args) {
   StringStack stack = new StringStack();
   while(!StdIn.isEmpty()) {
      String s = StdIn.readString();
      stack.push(s);
   }
   while(!stack.isEmpty()) {
      String s = stack.pop();
      System.out.println(s);
   }
}
```

a sample stack client

Stack Pop: Linked List Implementation

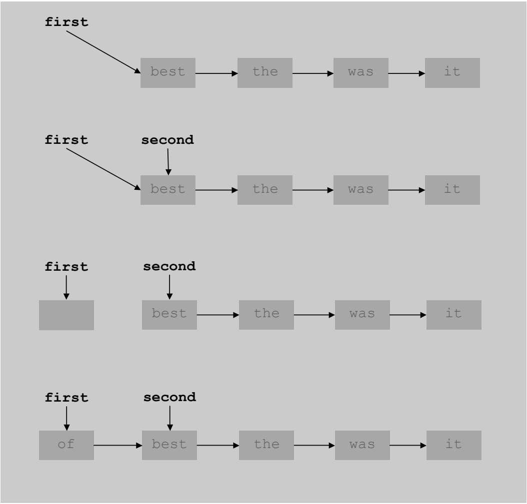


```
item = first.item;
```

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

```
return item;
```

Stack Push: Linked List Implementation



```
second = first;
```

```
first = new Node();
```

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

Stack: Linked List Implementation

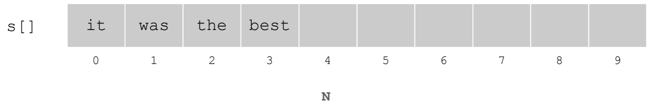
```
public class StringStack {
   private Node first = null;

   private class Node {
      String item;
      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"

Stack: Array Implementation

Array implementation of a stack.

- Use array `s[]` to store `N` items on stack.
- `push()` add new item at `s[N]`.
- `pop()` remove item from `s[N-1]`.



Stack: Array Implementation

```
public class StringStack {
   private String[] s;
   private int N = 0;

   public StringStack(int capacity) {
      s = new String[capacity];
   }

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

   public void push(String item) {
      s[N++] = item;
   }

   public String pop() {
      String item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }
}
```

garbage collector only reclaims memory if no outstanding references

Stack Array Implementation: Resizing

How to resize array? Increase size of `s[]` by one if the array is full.

Thrashing.

- Increasing the size of an array involves copying all of the elements to a new array.
- Inserting N elements: time proportional to $1 + 2 + \ldots + N \approx N^2/2$.
  N = 1 million ⇒ infeasible.

Stack Array Implementation: Dynamic Resizing

How to resize array? Use repeated doubling: if `s[]` not big enough, create a new array of twice the size, and copy items.

```
public StringStack() { this(8); }

public void push(String item) {
   if (N >= s.length) resize();
   s[N++] = item;
}

private void resize() {
   String[] dup = new String[2*N];
   for (int i = 0; i < N; i++)
      dup[i] = s[i];
   s = dup;
}
```

no-argument constructor

double the size of the array

Consequence. Inserting N items takes time proportional to N (not N^2).

Stack Implementations: Array vs. Linked List

Stack implementation tradeoffs. Can implement with either array or linked list, and client can use interchangeably. Which is better?

Array.

- Most operations take constant time.
- Expensive re-doubling operation every once in a while.
- Any sequence of N operations (starting from empty stack) takes time proportional to N. ← "amortized" bound

Linked list.

- Grows and shrinks gracefully.
- Every operation takes constant time.
- Uses extra space and time to deal with references.

Queue

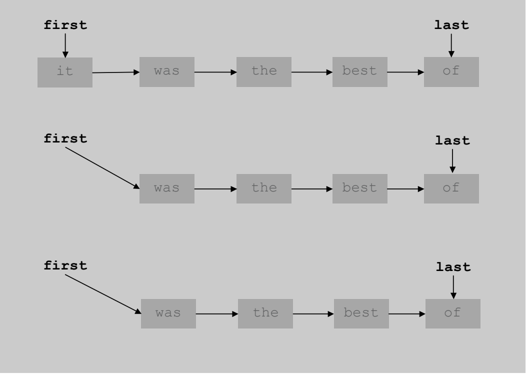
Queue operations.

- `enqueue()` Insert a new item onto queue.
- `dequeue()` Delete and return the item least recently added.
- `isEmpty()` Is the queue empty?

```java
public static void main(String[] args) {
   StringQueue q = new StringQueue();
   q.enqueue("Vertigo");
   q.enqueue("Just Lose It");
   q.enqueue("Pieces of Me");
   q.enqueue("Pieces of Me");
   System.out.println(q.dequeue());
   q.enqueue("Drop It Like It's Hot");
   while(!q.isEmpty())
      System.out.println(q.dequeue());
}
```

(CNN.COM GRPAHIC)

Dequeue: Linked List Implementation

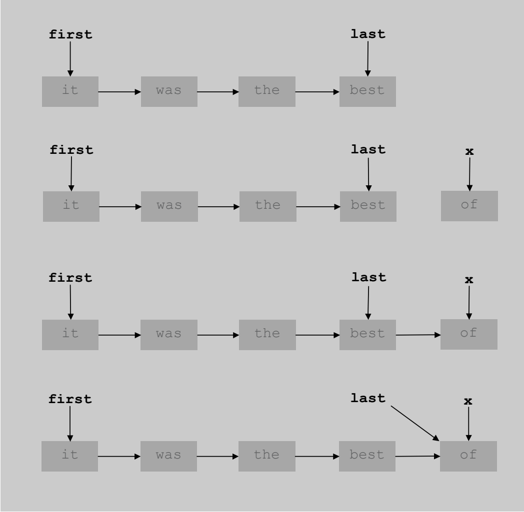


```
item = first.item;
```

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

```
return item;
```

Enqueue: Linked List Implementation



```
x = new Node();
x.item = item;
x.next = null;
```

```
last.next = x;
```

```
last = x;
```

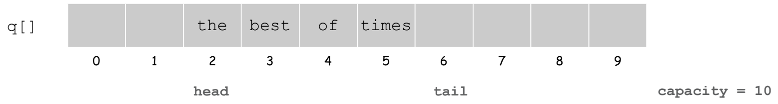
Queue: Linked List Implementation

```
public class StringQueue {
   private Node first;
   private Node last;

   private class Node { String item; Node next; }

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

   public void enqueue(String item) {
      Node x = new Node();
      x.item = item;
      x.next = null;
      if (isEmpty()) { first      = x; last = x; }
      else           { last.next = x; last = x; }
   }

   public String dequeue() {
      String item = first.item;
      first       = first.next;
      return item;
   }
}
```

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`.



q[]			the	best	of	times				
	0	1	2	3	4	5	6	7	8	9
			head				tail			

capacity = 10

Parameterized Data Types

Parameterized Data Types

We implemented: `StringStack`, `StringQueue`.

We also want: `URLStack`, `CustomerQueue`, etc?

Attempt 1. 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.

Stack of Objects

We implemented: `StringStack`, `StringQueue`.

We also want: `URLStack`, `CustomerQueue`, etc?

Attempt 2. Implement a stack with items of type `Object`.
- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
Stack  s = new Stack();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error

Generics

Generics. Parameterize stack by a single type.
- Avoid casting in both client and implementation.
- Discover type mismatch errors at compile-time instead of run-time.

parameter

```
Stack<Apple> s = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

compile-time error

no cast needed in client

Guiding principle. Run-time errors are much harder to identify and fix than compile-time errors.

Generic Stack: Linked List Implementation

```
public class Stack<Item> {
   private Node first;

   private class Node {
      Item item;
      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;
   }
}
```

arbitrary parameterized type name, but must use consistently

Generic Stack: Array Implementation

The way it should be.

```
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.

```
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];
   }
}
```

↖ the ugly cast

Autoboxing

Generic stack implementation. Allows objects, not primitive types.

Wrapper type.

- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Casts are still done behind the scenes.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17);         // s.push(new Integer(17));
int a = s.pop();    // int a = ((Integer) s.pop()).intValue();
```

Stacks and Queues: Applications

Stack Applications

Real world applications.

- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

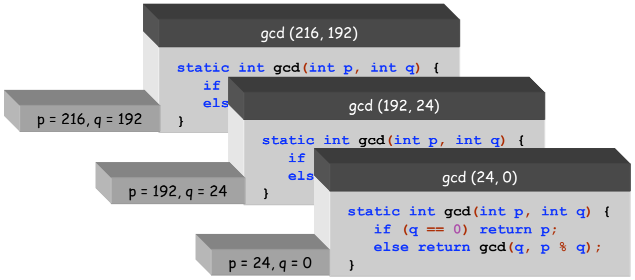
Function Calls

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 a stack to remove recursion.



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

Postfix Notation

Postfix notation.

- Put operator after operands in expression.
- Use stack to evaluate.
  - operand: push it onto stack
  - operator: pop operands, push result
- Systematic way to save intermediate results and avoid parentheses.

```
% java Postfix
1 2 3 4 5 * + 6 * * +
277    infix expression: (1+(((2*((3+(4*5))*6)))

% java Postfix
7 16 * 5 + 16 * 3 + 16 * 1 +
30001  convert 7531 from hex to decimal using Horner's method
```


J. Lukasiewicz

Postfix Evaluation

```
public class Postfix {
   public static void main(String[] args) {
      Stack<Integer> st = new Stack<Integer>();
      while (!StdIn.isEmpty()) {
         String string = StdIn.readString();
         if (s.equals("+"))      st.push(st.pop() + st.pop());
         else if (s.equals("*")) st.push(st.pop() * st.pop());
         else st.push(Integer.parseInt(s));
      }
      System.out.println(st.pop());
   }
}
```

```
% java Postfix
1 2 3 4 5 * + 6 * * +
277

% java Postfix
7 16 * 5 + 16 * 3 + 16 * 1 +
30001
```

Infix to Postfix

Infix to postfix algorithm.

- Left parentheses: ignore.
- Right parentheses: pop and print.
- Operator: push.
- Integer: print.

```
% java Infix
( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )
* 2 3 4 + 5 6 * * +

% java Infix | java Postfix
( 2 + ( ( 3 + 4 ) * ( 5 * 6 ) ) )
212
```

```
public class Infix {
   public static void main(String[] args) {
      Stack<String> stack = new Stack<String>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals(")")) System.out.print(stack.pop() + " ");
         else if (s.equals("(")) System.out.print("");
         else if (s.equals("+")) stack.push(s);
         else if (s.equals("*")) stack.push(s);
         else                    System.out.print(s + " ");
      }
   }
}
```

Queue Applications

Some applications.

- iTunes playlist.
- Breadth first search.
- Data buffers (iPod, TiVo).
- Graph processing (stay tuned).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

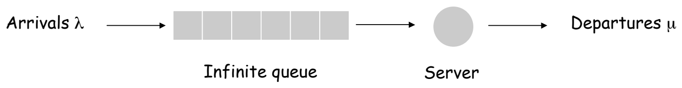
Simulations of the real world.

- Traffic analysis of Lincoln tunnel.
- Waiting times of customers in McDonalds.
- Determining number of cashiers to have at a supermarket.

M/M/1 Queuing Model

M/M/1 queue.

- Customers arrive at rate of λ per minute.
- Customers are serviced at rate of μ per minute.
- Inter-arrival time obeys exponential distribution: $\Pr[X \le x] = 1 - e^{-\lambda x}$



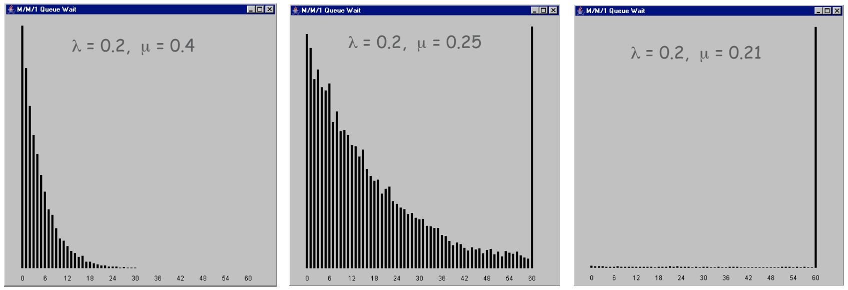
Q. How long does a customer wait in queue?

M/M/1 Queue: Implementation

```
public class MM1Queue {
   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 nextDeparture = StdRandom.exp(mu);
      while(true) {
         if (nextArrival < nextDeparture) {                arrival
            q.enqueue(nextArrival);
            nextArrival += StdRandom.exp(lambda);
         }
         else {                                          departure
            if (!q.isEmpty()) {
               double wait = nextDeparture - q.dequeue();
               // add waiting time to histogram
            }
            nextDeparture += StdRandom.exp(mu);
         }
      }
   }
}
```

M/M/1 Queue Analysis

Remark. As service rate approaches arrival rate, service goes to h***.



Theorem. Average time a customer spends in system = 1 / (μ - λ).

Summary

Stacks and queues are fundamental ADTs.

- Array implementation.
- Linked list implementation.
- Different performance characteristics.

Many applications.

Summary

ADTs enable modular programming.

- Separate compilation.
- Split program into smaller modules.
- Different clients can share the same ADT.

ADTs enable encapsulation.

- Keep modules independent (include `main()` in each class for testing).
- Can substitute different classes that implement same interface.
- No need to change client.

Issues of ADT design.

- Feature creep.
- Formal specification problem.
- Implementation obsolescence.