4.3 Stacks and Queues
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: Hoagie Haven line.

Symbol table.
- Remove the item with a given key.
- Ex: Phone book.
Stacks
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 \texttt{a[]} to store \texttt{N} items on stack.
- \texttt{push()} add new item at \texttt{a[\texttt{N}]}.
- \texttt{pop()} remove item from \texttt{a[\texttt{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]; }
}
```
## 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>0</td>
</tr>
<tr>
<td>push</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>be</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>to</td>
<td>5</td>
<td>be</td>
</tr>
<tr>
<td>pop</td>
<td>to</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td>be</td>
<td>5</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>be</td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>not</td>
<td>3</td>
<td>be</td>
</tr>
<tr>
<td>that</td>
<td></td>
<td>4</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>that</td>
<td>3</td>
<td>be</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>be</td>
</tr>
<tr>
<td>is</td>
<td></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.
- Original 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 allocation. Put items 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 item.
- **Java**: link is reference to next item.

Key distinctions.

- **Array**: random access, fixed size.
- **Linked list**: sequential access, variable size.
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
private class Node {
    private String item;
    private Node next;
}
```

why private? stay tuned

special pointer value null terminates 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;

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;
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;  // This line updates the 'last' pointer to point to the newly created node.
    last.item = StdIn.readString();
    last.next = null;
}
```
Q. What does the following code fragment do?

```java
for (Node x = first; x != null; x = x.next) {
    StdOut.println(x.item);
}
```
Enough with the Idioms

How about this idea:

- Use a linked list to implement a stack
Stack Push: Linked List Implementation

first

Node second = first;

second

first = new Node();

first

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

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

garbage-collected
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: Linked List Implementation

inner class
stack and linked list contents after 4th push operation
special reserved name
Linked List Stack: Test Client Trace
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.
Parameterized Data Types
public class LinkedStackOfString {  
   private Node first = null;

   private class Node {  
      private String item;
      private Node next;
   }  
   "inner class"

   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;
   }
}
Parameterized Data Types

We just implemented: StackOfStrings.

We also want: StackOfInts, StackOfURLs, StackOfVans, ...

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();
```

“stack of apples”

parameterized type

can’t push an orange onto a stack of apples

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(); // 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.
Goal. Evaluate infix expressions.

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

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 value 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.

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

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.

\[
1 + ( 2 - 3 - 4 ) * 5 * \sqrt{6*6 + 7*7}
\]
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, ...
Queues
Queue API

public class Queue<Item>

    Queue<Item>()       create an empty queue
    boolean isEmpty()   is the queue empty?
    void enqueue(Item item)    enqueue an item
    Item dequeue()        dequeue an item

    enqueue → [ ] [ ] [ ] [ ] [ ]        dequeue

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

first

String item = first.item;

first = first.next;

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.
From the point of view of a particular object: all of these structures look the same.

Multiply-linked data structures. Many more possibilities.
Conclusions

Stacks and Queues are fundamental ADTs.
Sequential allocation: supports indexing, fixed size.
Linked allocation: variable size, supports sequential access.

Linked structures are a central programming tool.
  • Linked lists.
  • Binary trees.
  • Graphs.
  • Sparse matrices.
  ...