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
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

Stack Client Example 1: Reverse

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 Client Example 2: Test Client

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 -- that -- is not or be
% java StackOfStrings < test.txt
to be not that or be

Stack: Array Implementation

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].

ArrayStackOfStrings

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]; }
}

Temporary solution: make client provide capacity

Stack and array contents after 4th push operation

how big to make array? [stay tuned]
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></td>
<td>0 1 2 3 4</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>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>pop</td>
<td>- to</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>- be</td>
<td>4</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>- not</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>- that</td>
<td>3</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>- or</td>
<td>2</td>
<td>to</td>
</tr>
<tr>
<td></td>
<td>- be</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 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 vs. Linked Allocation

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.

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>Alice</td>
</tr>
<tr>
<td>B1</td>
<td>Bob</td>
</tr>
<tr>
<td>B2</td>
<td>Carol</td>
</tr>
<tr>
<td>B3</td>
<td>C3</td>
</tr>
<tr>
<td>B4</td>
<td>C4</td>
</tr>
<tr>
<td>B5</td>
<td>C5</td>
</tr>
<tr>
<td>B6</td>
<td>C6</td>
</tr>
<tr>
<td>B7</td>
<td>C7</td>
</tr>
<tr>
<td>B8</td>
<td>C8</td>
</tr>
<tr>
<td>B9</td>
<td>C9</td>
</tr>
<tr>
<td>C0</td>
<td>Carol</td>
</tr>
<tr>
<td>C1</td>
<td>null</td>
</tr>
<tr>
<td>C2</td>
<td>-</td>
</tr>
<tr>
<td>C3</td>
<td>-</td>
</tr>
<tr>
<td>C4</td>
<td>Alice</td>
</tr>
<tr>
<td>C5</td>
<td>CA</td>
</tr>
<tr>
<td>C6</td>
<td>C6</td>
</tr>
<tr>
<td>C7</td>
<td>C7</td>
</tr>
<tr>
<td>C8</td>
<td>C8</td>
</tr>
<tr>
<td>C9</td>
<td>C9</td>
</tr>
<tr>
<td>CA</td>
<td>Bob</td>
</tr>
<tr>
<td>CB</td>
<td>-</td>
</tr>
</tbody>
</table>

array (B0) linked list (C4)
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;
}
```

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;
    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

```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;
    }

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

Stack Push: Linked List Implementation

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

Stack Pop: Linked List Implementation

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

Stack: Linked List Implementation

- `first = first.next;`
- `return item;`
Linked List Stack: Test Client Trace

Stack Data Structures: Tradeoffs

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

Stack: Linked List Implementation

```java
public class LinkedStackOfStrings {
    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

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();
```

53

Generic Stack: Linked List Implementation

```
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;
    }
}
```

54

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.

```
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.

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!
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

```java
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
}
```

```java
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

```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;
    }
}
```

Dequeue: Linked List Implementation

```java
public class Queue<Item> {
    private Node first, last;

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

    public void enqueue(Item item) {
        Node oldlast = last;
        last = new Node();
        last.item = "of";
        last.next = null;
        if (isEmpty()) first = last;
        else oldlast.next = last;
    }

    public Item dequeue() {
        String 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.
Singly-Linked Data Structures

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.
• . . .