Data Structures

Data Structures and Data Types

Data types
• Set of values.
• Set of operations on those values.
• Some are built in to Java: int, double, char, ...
• Most are not: Complex, Picture, Charge, Stack, Queue, Graph, ...

Data structures.
• Represent data.
• Represent relationships among data.
• Some are built in to Java: arrays, string, ...
• Most are not: linked list, circular list, tree, sparse array, graph, ...

Design challenge for every data type: What data structure to use?
• Requirement 1: Space usage.
• Requirement 2: Time usage for data-type methods

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. (this lecture)
• Remove the item most recently added.
• Ex: cafeteria trays, Web surfing.

Queue. (see text)
• Remove the item least recently added.
• Ex: Registrar’s line.

Symbol Table. (next lecture)
• Remove item with a given key.
• Ex: Phone book

Stacks

LIFO = "last in first out"
FIFO = "first in first out"
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())
            stack.push(StdIn.readString());
        while (!stack.isEmpty())
            StdOut.print(stack.pop());
        StdOut.println();
    }
}
```

Stack Client Example 2: Test Client

```java
public static void main(String[] args) {
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty()) {
        String item = StdIn.readString();
        if (item.compareTo("-") != 0)
            stack.push(item);
        else
            System.out.print(stack.pop());
    }
    System.out.println();
}
```

Stack Client Example 3: Balanced Parentheses

```java
((a + b) * d) + (e * f)
```

Stack contents just before first pop() operation

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

% java StackOfStrings tiny.txt
it was the best of times
% more tiny.txt
% java Reverse tiny.txt
```
Stack Client Example 3: Balanced Parentheses

```java
public class Balanced {
    public static void main(String[] args) {
        StackOfStrings stack = new StackOfStrings();
        while (!StdIn.isEmpty()) {
            String item = StdIn.readString();
            if (item.compareTo('(') == 0) stack.push(item);
            if (item.compareTo(')') == 0) {
                if (stack.isEmpty()) { StdOut.println("Not balanced"); return; }
                stack.pop();
            }
        }
        if (!stack.isEmpty()) StdOut.println("Not balanced");
        else StdOut.println("Balanced");
    }
}
```

% java Balanced
( ( ( a + b ) * d ) + ( e * f ) )
Balanced

% java Balanced
( ( a + b ) * d ) + ( e * f )
Not balanced

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]`.

Temporary solution: Make client provide capacity.

Stack Challenge: Stack Sort?

Q. Can we always insert `pop` commands (-) to make strings come out sorted?

Ex 1: 6 5 4 3 2 1 - - - - -
Ex 2: 1 - 2 - 3 - 4 - 5 - 6 -
Ex 3: 4 1 - 3 2 - - - 6 5 - -
Array Stack: Performance

Running time. Push and pop take constant time. ✓

Memory. Proportional to client-supplied capacity, not number of items. X

What's the problem?
• API does not call for capacity (never good to change API)
• Client might have multiple stacks (stay tuned for an example)
• Client might not know what capacity to use (depends on its client)

Challenge. Stack implementation where space use is not fixed ahead of time.

Example: potential stack client

Possible implementation of Java memory management system (sketch)

Maintain N stacks
• stack i: blocks of contiguous $2^i$ byte chunks of memory
• new: pop from stack t, where $2^t$ is smallest block that will hold new object
• stack t empty? pop from t+1, split in half, push 2 blocks on stack t
• garbage collector: periodically finds unused memory blocks and pushes onto appropriate stack.

Properties
• many stacks
• stack size unpredictable

Stack implementation without capacity restriction (as in API) is a requirement

Sequential vs. Linked Data Structures

Sequential data structure. Put object one next to another.
• TOY: consecutive memory cells.
• Java: array of objects.

Linked data structure. Include in each object a link to the another one.
• TOY: link is memory address of next object.
• Java: link is reference to next object.

Key distinctions.
• Array: arbitrary access, fixed size.
• Linked list: sequential access, variable size.

Linked structures.
• Not intuitive, overlooked by naive programmers
• Flexible, widely used method for organizing data
Singly-linked data structures

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

Sequential list (this lecture)

Circular list (TSP)

Multiply linked structures: many more possibilities!

Linked structures can become intricate

Linked Lists

Linked list.
- Simplest linked structure.
- 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.

Building a Linked List

Confusing point:
Purpose of data structure is to represent data in a data type, but we also use data types to implement data structures
Stack Push: Linked List Implementation

```java
Stack: Linked List Implementation

```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;
    }
}

Note difference between first and second:
- first: an instance variable that retains state
- second: a local variable that goes out of scope

Stack Pop: Linked List Implementation

```java
Stack Pop: Linked List Implementation

```linkage = first.next;  node = first;  return item;  first = first.next;  first.item = item;  first.next = second;
```
Linked-List Stack: Performance

- Running time. Push and pop take constant time. ✓
- Memory. Always proportional to number of items in stack. ✓

Stack Data Structures: Tradeoffs

Two data structures to implement the 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.
- But... uses extra space and time to deal with references.

Client can evaluate performance tradeoffs to choose among implementations (implicitly choosing among underlying data structures)

List-Processing Challenge 1

What does the following code do?

```java
... 
Node list = null;
while (!StdIn.isEmpty())
{
    Node old = list;
    list = new Node();
    list.item = StdIn.readString();
    list.next = old;
}
for (Node t = list; t != null; t = t.next)
    StdOut.println(t.item);
... 
```

A. Reverses the strings in StdIn.

Note: Better to use a Stack, represented as a linked list.
In this course, we always do list processing in data-type implementations.

List-Processing Challenge 2

What does the following code do?

```java
... 
Node list = new Node();
list.item = StdIn.readString();
Node last = list;
while (!StdIn.isEmpty())
{
    last.next = new Node();
    last = last.next;
    last.item = StdIn.readString();
}
... 
```

A. Puts the strings on StdIn in a linked list, in the order they are read (assuming at least 1 string on StdIn).

Note: Could use a Queue, represented as a linked list (see text).
In this course, we always do list processing in data-type implementations.
This code might be the basis for an initialization method in a data type.
Parameterized Data Types

We implemented: StackOfStrings.

We also want: StackOfMemoryBlocks, 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

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();
```

String stack (for reference)

```
public class StackOfStrings {
  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;
  }
}
```

Generics 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;
  }
}
```
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.
• Wrapper type has larger set of operations than primitive type.
• Values of wrapper type are objects.

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.

Stack Client 4: 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.

```java
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("(")) {
                if (ops.equals("+")) vals.push(vals.pop() + vals.pop());
                else if (ops.equals("*")) vals.push(vals.pop() * vals.pop());
                else vals.push(Double.parseDouble(s));
            }
            else if (s.equals("")")
            else if (s.equals("*")) ops.push(s);
            else if (s.equals("+")
            else if (s.equals("")")
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

```java
% 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.

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

Postfix

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

\[(1 ((23 +) (45 *) *) +)\]

Observation 2. Now all of the parentheses are redundant!

\[123 + 4 5 * * +\]

Bottom line. Postfix or “reverse Polish” notation.

Real-World Stack Application: PostScript


- postfix program code
- add commands to drive virtual graphics machine
- add loops, conditionals, functions, types

Simple virtual machine, but not a toy.
- Easy to specify published page.
- Easy to implement on various specific printers
- Revolutionized world of publishing.
- Virtually all printed material is PostScript.

Context/Definitions/Summary

Interpreter.
- Takes a program as input
- Does what that program would do.
- Simulates a virtual machine.

Compiler.
- Takes a program as input
- Produces a program as output.
- Produces code for a (real) machine.

Data Type and Virtual Machine are the same thing!
- Set of values = machine state.
- Operations on values = machine operations.

Data Structure.
- Represent data and relationships among data in a data type.
- array, linked list, compound, multiple links per node.