STUDENT OUTLINE

Lesson 38 – Stacks

INTRODUCTION: When studying recursion you were introduced to the concept of a stack. A stack is a linear data structure with well-defined insertion and deletion routines. The stack abstraction has been implemented for you in the *ArrayStack* class. After covering the member methods available in implementing a stack interface, the lab exercise will use stacks to solve a non-recursive *inorder* tree traversal problem.

The key topics for this lesson are:

A. The Stack Abstract Data Type
B. Implementation Strategies for a Stack Type

VOCABULARY:

STACK	POP
PUSH	TOP

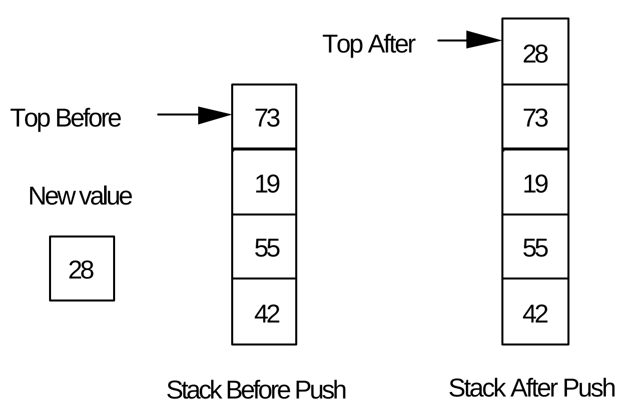
DISCUSSION: A. <u>The Stack Abstract Data Type</u>

1. A stack is a linear data structure, with each node or cell holding the same data type.

2. All additions to and deletions from a stack occur at the top of the stack. The last item pushed onto the stack will be the first item removed. A stack is sometimes referred to as a LIFO structure, which stands for Last-In, First-Out.

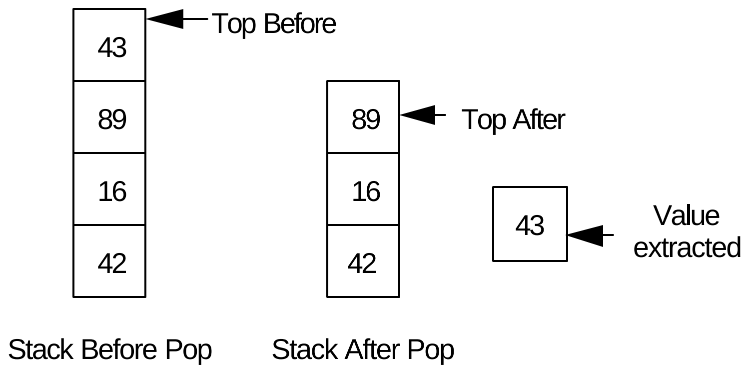
3. Two of the more important stack operations involve *pushing* data onto a stack and *popping* data off the stack.

4. The *push* operation will look like this:



<u>Push Operation</u>

5. The *pop* operation will look like this:



Pop Operation

B. Implementation Strategies for a Stack Type

See Handout H.A.38.1, *Stack Interface*.

1. A `Stack` interface is defined to formalize the stack methods. See Handout H.A.38.1, *Stack Interface*[*] for the details.

```
public interface Stack
{
  boolean isEmpty();
  void push(Object x);
  Object pop();
  Object peekTop();
}
```

2. The `Stack` interface above specifies the `push` and `pop` methods, the **`boolean`** method `isEmpty`, and an additional method `peekTop` that returns the value of the top element without removing it from the stack.

3. The following listing shows the `Stack` interface implemented in the `ArrayStack` class:

```
public class ArrayStack implements Stack
{
  private java.util.ArrayList array;

  public ArrayStack()
  { array = new java.util.ArrayList(); }
  public boolean isEmpty() { return array.size() == 0; }
  public void push(Object obj) { array.add(obj); }
  public Object pop() { return array.remove(array.size() - 1); }
  public Object peekTop() { return array.get(array.size() - 1); }
}
```

[*] Adapted from the College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

4. The data structure used in the `ArrayStack` class is an `ArrayList`. This allows for resizing of the stack as needed to make it larger.

5. Here is a short program illustrating usage of the `ArrayStack` class.

```
// Example program using the ArrayStack class

public static void main(String[] args)
{
  ArrayStack stack = new ArrayStack();

  for (int k = 1; k <= 5; k++)
    stack.push(new Integer(k));

  while (!(stack.isEmpty()))
  {
    System.out.print(stack.pop() + "  ");
  }
}
```

6. Another approach would be to use a linked list that would support true dynamic resizing. As you push data onto the stack another node is added to the appropriate end of the linked list. When data is popped from the stack, the linked list would be reduced in size. The following listing shows the `Stack` interface implemented in the `ListStack` class as a `java.util.LinkedList`:

```
public class ListStack implements Stack
{
  private java.util.LinkedList list;

  public ListStack() { list = new java.util.LinkedList(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public void push(Object obj) { list.addFirst(obj); }
  public Object pop() { return list.removeFirst(); }
  public Object peekTop() { return list.getFirst(); }
}
```

SUMMARY/ REVIEW: The stack ADT (*Abstract Data Type* – see Lesson 20) is what makes recursive algorithms possible. In the lab exercise you will gain a better understanding of the recursive *inorder* function used to traverse a binary tree.

ASSIGNMENT: Lab Exercise L.A.38.1, *Inorder*

LAB EXERCISE

Inorder

Background:

Any recursive algorithm can be reduced to a linear sequence of events. It will be longer and more difficult to follow, but recursive solutions can be rewritten as iterative solutions if a stack is available for use. In this lab exercise, you will implement a non-recursive `inorder` method now that we have a `stack` class to support stack operations. After completing the lab, you should have a greater appreciation for recursion and what it will accomplish for you.

You will work with the same binary tree code as implemented in Lessons 34-36. A non-recursive `inorder` method is summarized in pseudocode form below.

```
void inorder (TreeNode root)
{
  declare a stack of TreeNode, initialized as empty
  declare temp as a TreeNode

  start temp = root

  do
  {
    while moving temp as far left as possible,
      push tree references onto the stack

    if the stack is not empty
      reposition temp by popping the stack

    print the contents of tempgetValue()
    move temp one node to the right
  }
  while (the stack is not empty) or (temp != null)
}
```

Assignment:

1. Starting with an old binary tree lab, keep the code needed to read a data file (*file20.txt*) and build the binary tree.

2. Implement the `Stack` interface using either the `ArrayStack` class or the `ListStack` class described in the notes.

3. Solve the code for the non-recursive `inorder` method. Use (*file20.txt*) to test your program.

Instructions:

1. Turn in your source code and a run output. The run output should consist of the `inorder` output of the binary tree.

Stack Interface* and Implementation

```
public interface Stack
{
    // postcondition: returns true if stack is empty, false otherwise
  boolean isEmpty();

    // precondition:  stack is [e1, e2, ..., en] with n >= 0
    // postcondition: stack is [e1, e2, ..., en, x]
  void push(Object x);

    // precondition:  stack is [e1, e2, ..., en] with n >= 1
    // postcondition: stack is [e1, e2, ..., e(n-1)]; returns en
    //                 throws an unchecked exception if the stack is empty
  Object pop();

    // precondition:  stack is [e1, e2, ..., en] with n >= 1
    // postcondition: returns en
    //                 throws an unchecked exception if the stack is empty
  Object peekTop();
}


public class ArrayStack implements Stack
{
  private java.util.ArrayList array;

  public ArrayStack()
  {
    array = new java.util.ArrayList();
  }

  public void push(Object obj)
  {
    array.add(obj);
  }

  public Object pop()
  {
    return array.remove(array.size() - 1);
  }

  public Object peekTop()
  {
    return array.get(array.size() - 1);
  }

  public boolean isEmpty()
  {
    return array.size() == 0;
```

* Adapted from the College Board's *AP Computer Science AB: Implementation Classes and Interfaces*.

```
  }
}
```

```
public class ListStack implements Stack
{
  private java.util.LinkedList list;

  public ListStack()
  {
    list = new java.util.LinkedList();
  }

  public boolean isEmpty()
  {
    return list.isEmpty();
  }

  public void push(Object obj)
  {
    list.addFirst(obj);
  }

  public Object pop()
  {
    return list.removeFirst();
  }

  public Object peekTop()
  {
    return list.getFirst();
  }
}
```