Class Notes: Stacks, CS 3137

1 Stacks

- Stacks are a data structure that conform to the Last-In, First-Out protocol (LIFO)
- Stacks are often used to hold information about "postponed" operations that need to be done later.
- Operating systems often use stacks to keep track of recursive function calls
- The standard operations on stacks are:
  - PUSH: pushes a data element onto the stack
  - POP: takes the top the element off the stack, and usually returns that element to the calling function
  - TOP: Returns the top item of the stack, leaving the stack intact
  - ISEMPTY: Boolean to see if stack is empty
  - ISFULL: Boolean to see if stack is full
  - INITIALIZE: Create an empty stack
- Stacks can be implemented with linked lists or arrays, depending on whether the stack size (number of elements which the stack can hold) is known ahead of time or not.
- Whether we want to PUSH a double or a character string onto a stack, it requires much of the same code. Object oriented languages permit us to write a generic set of stack functions that work on different data types. Our stack methods should allow arbitrary objects to be placed on the stack.
- The linked list implementation of the stack is very simple. A PUSH operation is no more than a linked list insert at the beginning of the list. A POP is a delete from the begining of the list.

```
// simple stack class, implemented as an array, adapted from Data Structures
// by Bruno Preiss, code chapter 6

public class StackLinkedList
{
    protected SimpleLinkedList list;
    protected int count;

    public StackLinkedList ()
    { list=new SimpleLinkedList();}

    public void push (Object object)
    {
        list.addfront(object);
    }

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

    public Object pop ()
    {
        if (list.isEmpty()){
            System.out.println("error - pop of empty stack");
            return null;
        }
        Object result=list.retrievefront();
        list.deletefront();
        return result;
    }

    public Object getTop ()
    {
        if (list.isEmpty()){
            System.out.println("error - getTop of empty stack");
            return null;
        }
        Object result=list.retrievefront();
        return result;
    }
}
```

Here is a LinkedList class with a resticted set of methods that can be used to implement the stack:

```
//simple LinkedList class for use with a stack

public class SimpleLinkedList {

    protected Node header;

    public SimpleLinkedList() {
      header=null;
    }

    public boolean isEmpty() {
        if(header==null) return true;
        else return false;
    }

    public void addfront(Object obj) {
        Node tmp= new Node(obj,null);
        tmp.next=header;
        header=tmp;
    }
    public Object retrievefront() {
        return header.data;
    }

    public void deletefront() {
        if(isEmpty()){
            System.out.println("ilegal delete from empty list");
        } else header=header.next;
    }
}

public class Node{
   protected Object data;
   protected Node next;

   public Node(Object x, Node n){
     data=x;
     next=n;
   }
   public Node(){
       data=null;
       next=null;
   }
}
```

You can use the Java Collections interface to easily create a stack. Use a LinkedList class, which allows you to do insertions and deletions from the front of the list - essentially a stack. The code below also uses the generic typing facility of Java 1.5

```
import java.util.*;

public class StackGeneric<anytype>
{
    public LinkedList<anytype> list;
    public  int count;

    public StackGeneric ()
    { list=new LinkedList<anytype>();}

    public void push(anytype item)
    {
       list.addFirst(item);
    }

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

    public anytype pop ()
    {
       if (list.isEmpty()){
           System.out.println("error - pop of empty stack");
            return null;
        }
        anytype result=list.getFirst();
       list.removeFirst();
       return result;
    }

    public anytype getTop ()
    {
       if (list.isEmpty()){
           System.out.println("error - getTop of empty stack");
            return null;
        }
        anytype result=list.getFirst();
       return result;
    }
```

- You can also implement stacks using arrays. This simply uses an array and an index top to indicate which array element is the top element. If we start with top = 0, this means an empty stack (the array has nothing in it). Each $push$ operation increments top by 1. A pop decrements top by 1. $getTop$ is simply reading $StackArray[top - 1]$. OVERFLOW is if we try to add more elements than the array can hold. UNDERFLOW is trying to POP an empty stack.

```
// simple stack class, implemented as an array, adapted from Data
//   structures and Algorithms by Bruno Preiss, code chapter 6

public class StackArray
{
    protected Object[] array;
    protected int top;
    protected static final int DEFAULT_SIZE=10;

    public StackArray (int size)
    { array = new Object [size];
      top=0;
    }

    public StackArray ()
    { array = new Object [DEFAULT_SIZE];
      top=0;
    }

    public void purge ()
    {
        while (top > 0)
            array [--top] = null;
    }
    public void push (Object object)
    {
        if (isFull()){
            System.out.println("OVERFLOW - push of full stack");
        } else  array [top++] = object;
    }

    public Object pop ()
    {
        if (top == 0){
            System.out.println("UNDERFLOW - pop of empty stack");
            return null;
        }
        Object result = array [--top];
        array [top] = null;
        return result;
    }

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

    public boolean isFull() {
        return(top==array.length);
    }

    public Object getTop ()
    {
        if (top == 0){
            System.out.println("error - pop of empty stack");
            return null;
        }
        return array [top - 1];
    }

    public static void main(String[] Args) {

        int i;
        StackArray s= new StackArray();
        for(i=0;i<DEFAULT_SIZE;i++){
            s.push(new Integer(i));
        }
        while(!s.isEmpty()){
            System.out.println(s.pop());
        }
    }
}
```

2 Using a Stack for Recursion

Below is a simple recursive program that computes the factorial of a postive integer:

```
public class factorial{
public static int fact(int n)
{
  if(n==0) return 1;
      else return n*fact(n-1);
}
public static void main(String[] args){
  int num;
  num= Integer.parseInt(args[0]);
  if(num<0) System.out.println("error...factorial of negative number");
      else  System.out.println( num + "! = " + fact(num));

}
}
```

Each recursive call of the **fact()** method will generate an Operating System stack entry (known as a stack activation record) which remembers the "state" of each one of the recursive method calls so it can compute the final result.

If we try to compute **fact(4)** this is what happens:

Activation Stack	Activation Stack	Activation Stack	Activ. Stack	Activ. Stack
...	...	...	...	...
...	...	...	...	...
fact(0)	...	...	...	...
1 * fact(0)	1 * 1	...	...	...
2 * fact(1)	2 * fact(1)	2 * 1	...	...
3 * fact(2)	3 * fact(2)	3 * fact(2)	3 * 2	...
4 * fact(3)	4 * fact(3)	4 * fact(3)	4 * fact(3)	4 * 6

3 Using a Stack to Evaluate A Postfix Expression

- When we write an arithmetic expression, we normally use *infix* notation, in which the operator is in between the two operands: 5+3
- You can also write this a *postfix* expression, in which the operator is placed *after* the operands: 5 3 +.
- Without proving this, we will state that any infix arithmetic expression can be written as a postfix expression, and there is no need to use parentheses to establish precedence of arithmetric operations.
- Example: the infix expressions has an implied order of operations, which we can change with parentheses:

Infix	Postfix	Evaluation
2 - 3 * 4 + 5	2 3 4 * - 5 +	-5
(2 - 3) * (4 + 5)	2 3 - 4 5 + *	-9
2- (3 * 4 +5)	2 3 4 * 5 + -	-15

- We can use a simple stack algorithm to evaluate any postfix expression.
- The algorithm for postfix evaluation is simple:

```
While operands or operators still left do:
  Read in next token (token is operand or operator)
  If operand, PUSH on stack
     else {  /* operator */
        POP top two stack entries
        apply operator to these operands
        PUSH result back on stack
     }
Top of stack contains evaluated result
```

4 Converting Infix to Postfix

- Another common uses of stacks is in converting infix expressions to postfix. Suppose you want to generate the postifx equivalent of the infix expression:

  Infix: ((3-2)*8/4)+(7-3)∧(6/2)   Postfix: 3 2 - 8 * 4 / 7 3 - 6 2 / ∧ +
- There are 2 ways to do this: One is with a stack, and the other way is with a parser (more details on this in CS W4115!). We will outline the stack method here.
- A stack is needed because we are going to "postpone" putting each operator out in the postfix string until we are sure it is in the right place. Remember, we do not use parentheses in postifx notation to force the precedence of operators.
- We will use a stack called an *operator-stack* Our input here is assumed to be *tokens* from the input stream, reading the expression left to right. The *tokens* are the following:
  - Numbers
  - arithmetic binary operators: +,-,*,/,∧
  - Parentheses: (,)
  - Special Symbols: EOL (newline character)
- An important fact to note about postifx expressions is that the left to right ordering of the operands is the same for infix and postfix. Further, since any operator must logically come after its operands in postfix, we can automatically output an operand directly to the output stream when we see it. If we see an operator or parentheses, we have to decide whether to hold on to it for later use (why we have a stack) or output the operator.
- Outline of the Algorithm:
  1. We use a special symbol to represent the bottom of the stack (#). Push the bottom of stack symbol (#) onto *operator-stack*.
  2. Read in a token, and if its a number, just print it out directly
  3. If it is not a number, then we have to compare this new token with the token on the top of the *operator-stack*
  4.
     - If the new token has a higher precedence that the token on the top of the *operator-stack*, we push the new token onto the *operator-stack*.
     - If the new token's precedence is less than the token on the top, we **UNWIND** the *operator-stack* until we find a token on top whose precedence is less than the new token.
     - Once we have unwound the stack this far, we then place the new token on the *operator-stack*. The **UNWIND** simply pops operator's of the stack while the new input token's precedence is less than the top of the stack's precedence.
  5. The effect of all this is to pop operator's of the stack and output them. NOTE: you must continue to unwind as long as the new token's precedence is lower than the token on top of the *operator-stack*.
  6. We do this until we see EOL symbol, and at that point we pop all the operators in the *operator-stack* out.
- **PARENTHESES:** Parentheses are easy to deal with. Always push a left parentheses onto the *operator-stack*. If you see a right parentheses, simply unwind the stack until you see the corresponding left parentheses, and remember to pop this left parentheses off the operator stack. You will NEVER push a right parentheses onto the stack.
- **SPECIAL CASES :** There are a number of special cases, most dealing with illegal inputs, end of the input stream, or one of the stacks being empty. Suggestion: get the program working first without parentheses, then get it to accept parentheses, and finally add error tests for illegal inputs.
- **BUILDING A PRECEDENCE TABLE:** This is simply a a table that compares the top entry in the *operator-stack* with the next operator token from the input stream and tells you what to do. In the table, the symbol on the left hand column is the top of the stack entry, and the symbol on the top line is the next token read in.

Top of Stack			Input	Token				
	(	+	-	*	/	∧	)	EOL
(	PUSH	PUSH	PUSH	PUSH	PUSH	PUSH	MATCH	ERROR
+	PUSH	POP	POP	PUSH	PUSH	PUSH	POP	POP
-	PUSH	POP	POP	PUSH	PUSH	PUSH	POP	POP
*	PUSH	POP	POP	POP	POP	PUSH	POP	POP
/	PUSH	POP	POP	POP	POP	PUSH	POP	POP
∧	PUSH	POP	POP	POP	POP	PUSH	POP	POP
)	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR	ERROR
#	PUSH	PUSH	PUSH	PUSH	PUSH	PUSH	ERROR	DONE!

- Below is a *possible* implementation in JAVA (some table entries left out below). This allows for a **table-driven** algorithm:

```
final int POP=0;
final int PUSH=1;
final int MATCH=2;
final int ERROR=3;
final int EOL=012;
final int DONE=4;

/* wastes space but makes the table easy to generate */

int[][] table = new int[256][256];
....
table[’(’][’*’]=PUSH;
table[’(’][’/’]=PUSH;
table[’(’][’-’]=PUSH;
table[’(’][’+’]=PUSH;
table[’(’][’ˆ’]=PUSH;
table[’(’][’(’]=PUSH;
table[’(’][EOL]=ERROR;
table[’(’][’)’]=MATCH;

table[’+’][’*’]=PUSH;
table[’+’][’/’]=PUSH;
table[’+’][’-’]=POP;
table[’+’][’+’]=POP;
table[’+’][’ˆ’]=PUSH;
table[’+’][’(’]=PUSH;
table[’+’][EOL]=POP;
table[’+’][’)’]=POP;

table[’*’][’*’]=POP;
table[’*’][’/’]=POP;
table[’*’][’-’]=POP;
table[’*’][’-’]=POP;
table[’*’][’ˆ’]=PUSH;
table[’*’][’(’]=PUSH;
table[’*’][EOL]=POP;
table[’*’][’)’]=POP;

table[’ˆ’][’*’]=POP;
table[’ˆ’][’/’]=POP;
table[’ˆ’][’-’]=POP;
table[’ˆ’][’+’]=POP;
table[’ˆ’][’ˆ’]=PUSH;
table[’ˆ’][’(’]=PUSH;
table[’ˆ’][EOL]=POP;
table[’ˆ’][’)’]=POP;

table[’#’][’*’]=PUSH;
table[’#’][’/’]=PUSH;
table[’#’][’-’]=PUSH;
table[’#’][’+’]=PUSH;
table[’#’][’ˆ’]=PUSH;
table[’#’][’(’]=PUSH;
table[’#’][E0L]=DONE;
table[’#’][’)’]=ERROR;
```