Assignment 2 – Stacks

Write pseudo-code, not Java, for problems requiring code. You are responsible for the appropriate level of detail.

1. a) Use the operations push, pop, peek and empty to construct an operation which sets *i* to the bottom element of the stack, leaving the stack unchanged. (hint: use an auxiliary stack.)

```
public static Stack setBottom(int i, Stack stack) { // assume i is of type int
   Stack auxStack = new Stack(); // auxiliary stack to temporarily store stack data
   while stack.empty == false {
      auxStack.push(stack.pop()); // empty stack in to aux stack
   }
   stack.push(i); // add i at bottom
   while auxStack.empty == false {
      stack.push(auxStack.pop()); // fill stack with stored data from aux stack
   }
   return stack; // return stack with i at bottom
}
```

b) Use the operations push, pop, peek and empty to construct an operation which sets *i* to the third element from the bottom of the stack. The stack may be left changed.

```
public static Stack setBottom(int i, Stack stack) { // assume i is of type int
   try {
      Stack auxStack = new Stack(); // auxiliary stack to temporarily store stack data
      while stack.empty == false {
         auxStack.push(stack.pop()); // empty stack in to aux stack
      }
      stack.push(auxStack.pop());
      stack.push(auxStack.pop());
      stack.push(i); // add i at third from bottom
      while auxStack.empty == false {
         stack.push(auxStack.pop()); // fill stack with stored data from aux stack
      }
   catch emptyStackException {
      raise Exception("The given stack was of size less than 2!");
     }
      return stack;
}
```

2. Simulate the action of the algorithm for checking delimiters for each of these strings by using a stack and showing the contents of the stack at each point. Do not write an algorithm.

a) {[A+B]-[(C-D)]

Character	Action	Stack
{	Push({)	{

[	**Push([)**	**{[**
A		**{[**
+		**{[**
B		**{[**
]	**Pop([)**	**{**
-		**{**
[	**Push([)**	**{[**
(	**Push(()**	**{[(**
C		**{[(**
-		**{[(**
D		**{[(**
)	**Pop(()**	**{[**
]	**Pop([)**	**{**

Since the stack at the end of the operations is not empty, this is not a valid expression

b) ((H) * {([J+K])})

Character	Action	Stack
(	**Push(()**	**(**
(	**Push(()**	**((**
H		**((**
)	**Pop(()**	**(**
*****		**(**
{	**Push({)**	**({**
(	**Push(()**	**({(**
[	**Push([)**	**({([**
J		**({([**
+		**({([**
K		**({([**
]	**Pop([)**	**({(**
)	**Pop(()**	**({**
}	**Pop({)**	**(**
)	**Pop(()**	

Since the stack is empty at the end of operations and never tries to pop when empty, this is a valid expression

3. Write an algorithm to determine whether an input character string is of the form

x C y

where *x* is a string consisting only of the letters '*A*' and '*B*' and *y* is the reverse of the *x* (i.e. if *x="ABABBA"* then *y* must equal *"ABBABA"*). At each point you may read only the next character in the string, i.e. you must process the string on a left to right basis. You may not use string functions.

```
public static Boolean doHWProbThree(String string) {
   Stack xStack = new Stack;
```

```
   boolean returnValue = true; // assume true (lets be optimistic!)
   int i = 0; // iterator to access place in string
   while string[i] != "C" {
      xStack.push(string[i]); // push string value to top of xStack
      i++; // move on to next letter
   }
   i++; // need to move to next value of string after "C"
   while xStack.empty() == false {
      if xStack.pop() != string[i] { // top of xStack should match next value of string
         returnValue = false;
      }
      i++ // move on to next letter
   }
   return returnValue;
}
```

4. Write an algorithm to determine whether an input character string is of the form

a D b D c D … D z

Where each string *a, b, …z* is of the form of the string defined in problem 4. (Thus a string is in the proper form if it consists of any number of such strings from problem 4, separated by the character '*D*', e.g. *ABBCBBADACADBABCBABDAABACABAA*.) At each point you may read only the next character in the string, i.e. you must process the string on a left to right basis. You may not use string functions..

```
public static bool doHWProbFour(String string) {
   Stack leftSide = new Stack(); // represent the left side of string
   Stack leftSideBackup = new Stack;
   Stack rightSide = new Stack(); //
   String middleChar = "";

   if string[0] == "D" || string[1] == "D" {
      return false; // this would be invalid
   }

   leftSide.push(string[0]);
   middleChar = string[1]; // initialize variables ready for for{} loop

   for(n = 2, n < string.size(), n++) {
      leftSideBackup = leftSide; // keep a backup if we destroy the left side stack prematurely

      if string[n] == "D" {
         if leftSide != rightSide { // or some algorithm to figure it out
            return false;

      if string[n] == leftSide.peek() { // this may be the right side of the string
```

```
            rightSide.push(string[n]);
            leftSide.pop(); // if this is not the right side of the string we have a backup
        }
        else {
            leftSide.push(middleChar);
            leftSide.pushStack(rightSide);
            // assume this is set up, and it maintains the order of rightSide (and does nothing if empty)
            middleChar = string[n];
            rightSide.clear() // assume this is set up
            leftSideBackup = leftSide;
        }
    }
    if leftSide == rightSide { // there will be one more string to compare after the for loop
        return true;
    }
    else {
        return false;
    }
}
```

5. Design and implement a stack in which each item on the stack is a varying number of integers. Choose a Java data structure to implement your stack and design push and pop methods for it. You may not use library functions. Hint: Make it as simple as possible.

```
public class Stack {
    private int topIndex; // keep track of list index of stack
    private stack[]; // each item in stack is an array

    init() {
        this.topIndex = 0;
        this.stack = new Array[ pick your stack size ]; // you can choose size of stack
    }

    private static void push(Array toPush) {
        stack[topIndex] = toPush;
        topIndex ++;
    }

    private static Array pop() {
        topIndex--;
        Array toPop = stack[topIndex];
        stack[topIndex] = []; // clear out value in stack
        return toPop;
```

6. Consider a language that does not have arrays but does have stacks defined as a data type. That is, one can declare

```
stack s;
```

The push, pop, empty, and peek operations are defined. Show how a one-dimensional array can be implemented by using these operations on two stacks. In particular, show how you can insert and delete into such an array.

```
public stack array; // array is public variable accessible by either function

insert(item, int index) { // item can be whatever type you specify
   stack overflowStack;
   i = 0;
   while i < index {
      overFlowStack.push(array.pop()); // store items in overflow
      i++;
   }
   array.push(item); // insert item
   while i >= 0 {
      array.push(overFlowStack.pop()); // put overflow items back into array
      i--;
   }
   return array;
}

delete(int index) {
   stack overflowStack;
   i = 0;
   while i < index {
      overFlowStack.push(array.pop()); // store items in overflow
      i++;
   }
   array.pop(); // delete item
   while i > 0 {
      array.push(overFlowStack.pop()); // put overflow items back into array
      i--;
   }
   return array;
```

7. Design a method for keeping two stacks within a single linear array s[SPACESIZE] so that neither stack overflows until all of memory is used and an entire stack is never shifted to a different location within the array. Write methods *push1, push2, pop1, and pop2* to manipulate the two stacks. (Hint: the two stacks grow toward each other.)

```
public stack stack1; // public stack accessible to all functions
public stack stack2;

public static void push1(item) { // item can be any type
   if stack1.size() >= stack2.size() { // assume size is all set up
      stack2.push(item);
   }
   else {
      stack1.push(item);
```

```
   }
}

public static void push2(item) { // item can be any type
   if stack2.size() >= stack1.size() { // assume size is all set up
      stack1.push(item);
   }
   else {
      stack2.push(item);
   }
}

public static item pop1() { // return type item
   if stack1.size() >= stack2.size() { // assume size is all set up
      stack2.pop();
   }
   else {
      stack1.pop();
   }
}

public static item pop2() { // return type item
   if stack2.size() >= stack1.size() { // assume size is all set up
      stack1.pop();
   }
   else {
      stack2.pop();
   }
}
```

8. **Transform each of the following expressions to prefix and postfix expressions.**

a. (A+B)*(C$(D-E)+F)-G
b. A+(((B-C)*(D-E)+F)/G)$(H-J)

prefix
a. -*+AB+$C-DEFG
b. $+A/+*-BC-DEFG-HJ
postfix
a. AB+CDE-$F+G-*
b. ABC-DE-*F+G/*HJ-$

9. **Transform each of the following expressions to infix expressions.**

a. ++A-*$BCD/+EF*GHI
b. +-$ABC*DEFG**

c. AB-C+DEF-+$
d. ABCDE-+$*EF*-

answers
a. A+((B$C)*D)-((E+F)/(G*H))+I
b. (A$B)*(C+(D-E))-(E*F)
c. ((A-B)+C)(D+(E-F))$
d. ((A-B)+C)(D+(E-F))$

10. Apply the evaluation algorithm in the text to evaluate the following postfix expressions, where A=1, B=2, and C=3.

a. AB+C-BA+C$-
b. ABC+*CBA-+*

ansers:
a. -27
b. 20

11. Write a prefix method to accept an infix string and create the prefix form of that string, assuming that the string is read from right to left and that the prefix string is created from right to left.

```
public static bool isOpenParentheses(String char) { // give string of length 1
   if char in [‘(‘, ‘[‘, ‘{‘] { // I believe that is python notation
      return true;
   }
   else {
      return false;
   }

public static bool isClosedParentheses(String char) // give string of length 1
   if char in [‘)‘, ‘]‘, ‘}‘] { // I believe that is python notation
      return true;
   }
   else {
      return false;
   }

public static bool isOperator(String char) // give string of length 1
   if char in [‘+’, ‘-‘, ‘/’, ‘*’, ‘$’] { // I believe that is python notation
      return true;
   }
   else {
      return false;
   }

public static int getPrecedence(String char) { // give string of length 1
   if char in [‘+’, ‘-‘] {
```

```
        return 0;
    }
    else if char in [‘*’, ‘/’] {
        return 1;
    }
    else if char == ‘$’ {
        return 2;
    }
    return -1;
}

public static bool isVar(String char) { // give string of length 1
    if isOpenParentheses(char) == false && isClosedParentheses(char) == false &&
       isOperator(char) == false {
        return true}
    }
    else {
        return false;
    }

public static String infixToPrefix(String infix) {
    int stringSize = infix.size();

    Stack operatorStack = new Stack();
    String prefixString = “”;

    for(n = stringSize - 1, n >= 0; n--) { // iterate over each value in infix
        if isVar(infix[n]) {
            prefixString.insert(0, infix[n]); // insert to the front of prefix string if variable
        }
        else if isOpenParenthesis(infix[n]) {
            operatorStack.push(infix[n]);
        }
        else if isClosedParenthesis(infix[n]) {
            prefixString.insert(0, operatorStack.pop()); // move operator in parentheses to prefix
            operatorStack.pop(); // clear open parentheses from operator stack
        }
        else if isOperator(infix[n]) {
            while getPrecedence(operatorStack.peek()) > infix[n].peek { //keep precedence order
                prefixString.insert(0, operatorStack.pop());
            }
            operatorStack.push(infix[n]);
        }
    }
    while operatorStack.pop() != nil { //clear out operator stack (.pop() on an empty stack gives nil)
        prefixString.insert(0, operatorStack.pop());
    }
    return prefixString;
}
```