**Assignment 2 – Stacks**

*Write pseudo-code for problems requiring code. Do not write Java, Python or C++. 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.)**

inputStack //the input stack

workingStack //init to empty stack

while inputStack is not empty

push the return from pop(inputStack) to workingStack

push i to inputStack

while workingStack is not empty

push the return from pop(workingStack) to inputStack

return inputStack

**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.**

inputStack //the input stack

workingStack //init to empty stack

counter //init to 0

placeholder //some default data

while inputStack is not empty

push the return from pop(inputStack) to workingStack

increment counter by 1

if counter >= 3

for 3 iterations

push the return from pop(workingStack) to inputStack

else

for counter iterations

push the return from pop(workingStack) to inputStack

for (3 – counter)

push placeholder to inputStack

push i to inputStack

while workingStack is not empty

push the return from pop(workingStack) to inputStack

return inputStack

1. **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.**
2. **{[A+B]-[(C-D)]**

|  |  |  |  |
| --- | --- | --- | --- |
| Item | Action | Stack | Comment |
| { | push | { |  |
| [ | push | {[ |  |
| A | none | {[ | unchanged |
| + | None | {[ | unchanged |
| B | None | {[ | unchanged |
| ] | pop | { |  |
| - | None | { | unchanged |
| [ | push | {[ |  |
| ( | Push | {[( |  |
| C | None | {[( | unchanged |
| - | none | {[( | unchanged |
| D | none | {[( | unchanged |
| ) | pop | {[ |  |
| ] | pop | { | Error...stack is not empty |

1. **((H) \* {([J+K])})**

|  |  |  |  |
| --- | --- | --- | --- |
| Item / next input | Action | Stack | Comments |
| ( | Push | ( |  |
| ( | push | (( |  |
| H | none | (( | unchanged |
| ) | pop | ( |  |
| \* | none | ( | unchanged |
| { | push | ({ |  |
| ( | push | ({( |  |
| [ | push | ({([ |  |
| J | none | ({([ | unchanged |
| + | none | ({([ | unchanged |
| K | none | ({([ | unchanged |
| ] | pop | ({( |  |
| ) | pop | ({ |  |
| } | pop | ( |  |
| ) | pop |  | Success...stack empty |

1. **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.**

inputStack //the input stack xCy where each node corresponds to one letter ‘A’,’B’, or ‘C’

workingStack //init to empty stack which will represent ‘x’

while inputStack is not empty and peak(inputStack) does not equal C

push the return from pop(inputStack) to workingStack

if inputStack is not empty

pop(inputStack) //clears ‘C’ from top of inputStack

while inputStack is not empty

if workingStack is empty OR pop(inputStack) does not equal pop(workingStack)

return false //inputStack did not have correct ‘y’ for input ‘x’

else

return false //inputStack did not contain letter ‘C’

return true //no error conditions met. Form is correct

1. **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 3. (Thus a string is in the proper form if it consists of any number of such strings from problem 3, 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.**

inputStack //the input stack xCy where each node corresponds to one letter ‘A’,’B’, ‘C’, or ‘D’

workingStack //init to empty stack

while inputStack is not empty

if peak(inputStack) does not equal C or D

push the return from pop(inputStack) to workingStack

else if peak(inputStack) equals C and workingStack is not empty

pop(inputStack) //clears ‘C’

while working stack is not empty

if inputStack is not empty and pop(inputStack) does not equal pop(workingStack)

return false //inputStack did not have correct form

else if peak (inputStack) equals D

if workingStack is not empty

return false //last group of characters not correct xCy form

pop(inputStack) //clear D

if peak(inputStack) is D, return false //sequential D

else

return false

return true //no error conditions met

1. **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 into and read from such an array.**

Array would have to start at the bottom of the stack. It is easiest to start counting from the bottom of the stack similar to one of the first questions

//insert – define ADT

// input – inputIndex, inputData, inputArray

// precondition – inputIndex exists

// process – insert the inputData into the inputIndex index of an array

// postcondition – inputData inserted at inputIndex of the inputArray

// output – returns stack with new insert if successful, otherwise returns null pointer

inputStack = inputArray //the input stack where the bottom of the stack corresponds to index 0 of inputArrary

workingStack //init to empty stack

inputIndex

inputData

counter //init to 0

placeholder //some default data

while inputStack is not empty

push the return from pop(inputStack) to workingStack

increment counter by 1

if counter >= inputIndex

for inputIndex iterations

push the return from pop(workingStack) to inputStack

else

return null //inputIndex exceeds the length of the inputArray

push inputDATA to inputStack

while workingStack is not empty

push the return from pop(workingStack) to inputStack

return inputStack

//read – define ADT

// input – inputIndex, inputArray

// precondition – inputIndex exists

// process – read the array until inputIndex found and return its contents

// postcondition – none

// output – returns data contents of inputIndex of inputArray if successful, otherwise returns null pointer

inputStack = inputArray //the input stack where the bottom of the stack corresponds to index 0 of inputArrary

workingStack //init to empty stack

inputIndex

counter //init to 0

while inputStack is not empty

push the return from pop(inputStack) to workingStack

increment counter by 1

if counter >= inputIndex

for inputIndex iterations

pop(workingStack)

return pop(workingStack)

else

return null

1. **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.)**

precondition to all is that SPACESIZE > stack1.size + stack2.size

push1 (inputData, s[SPACESIZE])

counter //init to 0

while s[counter] is not empty and counter is less than SPACESIZE

counter ++

if counter < SPACESIZE

s[counter] = inputData

else

throw expception //stack 1 exceeds the size of array ‘s’

push2 (inputData, s[SPACESIZE])

counter //init to SPACESIZE - 1

while s[counter] is not empty and counter is greater or equal to 0

counter – -

if counter > = 0

s[counter] = inputData

else

throw exception //stack 2 exceeds the size of array ‘s’

pop1(s[SPACESIZE])

counter //init to 0

while s[counter] is not empty and counter is less than SPACESIZE

counter ++

if counter < SPACESIZE

return s[counter – 1] //subtract 1 otherwise will return empty node

else

return null //end of array ‘s’ reached

pop2(s[SPACESIZE])

counter //init to SPACESIZE – 1

while s[counter] is not empty and counter is greater than or equal to 0

counter - -

if counter >= 0

return s[counter]

else

return null //end of array ‘s’ reached due to stack1 or stack2 being too large

1. **Transform each of the following expressions to prefix and postfix expressions. $ is exponentiation.**

**a. (A+B)\*(C$(D-E)+F)-G**

|  |  |
| --- | --- |
| AB+\*(C$DE-+F)-G | +AB\*(C$-DE+F)-G |
| AB+\*(CDE-$+F)-G | +AB\*($C-DE+F)-G |
| AB+\*CDE-$F+-G | +AB\*(+$C-DEF)-G |
| AB+CDE-$F+\*-G | \*+AB+$C-DEF-G |
| AB+CDE-$F+\*G- | -\*AB+$C-DEFG |
| ^postfix | ^prefix |

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

|  |  |
| --- | --- |
| A+((BC-\*DE-+F)/G)$HJ- | A+((-BC\*-DE+F)/G)$-HJ |
| A+((BC-DE-\*+F)/G)$HJ- | A+((\*-BC-DE+F)/G)$-HJ |
| A+((BC-DE-\*F+)/G)$HJ- | A+(+\*-BC-DEF/G)$-HJ |
| A+(BC-DE-\*F+G/)$HJ- | A+/+\*-BC-DEFG$-HJ |
| A+BC-DE-\*F+G/HJ-$ | A+$/+\*-BC-DEFG-HJ |
| ABC-DE-\*F+G/HJ-$+ | +A$/+\*-BC-DEFG-HJ |
| ^postfix | ^prefix |

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

**a. ++A-\*$BCD/+EF\*GHI**

|  |  |
| --- | --- |
| OPERANDS | OPERATOR |
| IHG | \* |
| I(G\*H)FE | + |
| I(G\*H)(E+F) | / |
| I[(E+F)/(G\*H)]DCB | $ |
| I[(E+F)/(G\*H)]D(B$C) | \* |
| I[(E+F)/(G\*H)][(B$C)\*D] | - |
| I{[(B$C)\*D]-[(E+F)/(G\*H)]}A | + |
| I(A+{[(B$C)\*D]-[(E+F)/(G\*H)]}) | + |
| (A+{[(B$C)\*D]-[(E+F)/(G\*H)]})+I | ← FINAL ANS |

**b. +-$ABC\*D\*\*EFG**

|  |  |
| --- | --- |
| OPERANDS | OPERATOR |
| GFE | \* |
| G(E\*F) | \* |
| [(E\*F)\*G]D | \* |
| (D\*[(E\*F)\*G])CBA | $ |
| (D\*[(E\*F)\*G])C(A$B) | - |
| (D\*[(E\*F)\*G])[(A$B)-C] | + |
| [(A$B)-C]+(D\*[(E\*F)\*G]) | ← FINAL ANS |

**c. AB-C+DEF-+$**

|  |  |
| --- | --- |
| OPERANDS | OPERATORS |
| A |  |
| AB | - |
| (A-B) |  |
| (A-B)C | + |
| ((A-B) + C)D |  |
| ((A-B) + C)DE |  |
| ((A-B) + C)DEF | - |
| ((A-B)+C)D(E-F) | + |
| ((A-B)+C)(D+(E-F)) | $ |
| ((A-B)+C)$(D+(E-F)) | ← FINAL ANS |

**d. ABCDE-+$\*EF\*-**

|  |  |
| --- | --- |
| OPERANDS | OPERATORS |
| A |  |
| AB |  |
| ABC |  |
| ABCD |  |
| ABCDE | - |
| ABC(D-E) | + |
| AB(C+(D-E)) | $ |
| A(B$(C+(D-E))) | \* |
| [A\*(B$(C+(D-E)))]EF | \* |
| [A\*(B$(C+(D-E)))][E\*F] | - |
| [A\*(B$(C+(D-E)))]-[E\*F] | ← FINAL ANS |

1. **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$-**

((A+B)-C)-(B+A)$C

3C-BA+C$-

0-3$3= -27

**b. ABC+\*CBA-+\***

[A\*(B+C)]\*[C+(B-A)]

5\*4=20

1. **Write a prefix function 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. Handle variables, +,-,/,+,$, (,).**

**To think about and to discuss in Office Hours: Design and implement a stack in which each item on the stack is a varying number of integers. Choose a native data structure to implement your stack and design push and pop methods for it, without using library functions. Hint: Make it as simple as possible.**

inputStack //a infix expression read from right to left, assume right most character at top of stack

outputStack //a prefix expression with the right most character at the top of the stack

workingStack//an empty stack

operatorHierarchy //mapping for operator comparison

// +,- = 1

// \*,/ = 2

// $ = 3

// (,) = 0

while inputStack is not empty

if peak(inputStack) is an operand (i.e. “+”, ”-”, ”/”, ”\*”, ”$”, ”(“, ”)”)

if peak(inputstack) is “(“

if peak(workingStack) is “)”

pop(workingStack)

else

return null //error condition

if workingStack is empty or peak(inputStack) greater than or equal to peak(workingStack)

push the return of pop(inputStack) to workingStack

while workingStack is not empty and peak(inputStack) is less than peak(workingStack)

push the return of pop(workingStack) to outputStack

else

push the return of pop(inputStack) to outputStack

return outputStack