Adam Rich

EN.605.202.87.SP18 Data Structures

Module 2, Homework Assignment 2

February 13, 2018

**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 second element from the top of the stack, leaving the stack unchanged.**

method peek\_2(stack s)

init i

init j //temp variable

if (!s.Empty()) j = s.pop() else throw error

if (!s.Empty()) i = s.peek() else throw error

s.push(j)

return(i)

end-method

**b) Use the operations push, pop, peek and empty to construct an operation which sets *i* to the *nth* element from the top of the stack, leaving the stack without its top *n* elements. You are given integer *n.***

// assume "nth element from the top" is interpreted as

// "top = element 1, remove elements 1, 2, 3, ... n"

method pop\_n(stack s, int n)

init i

init int j = 0 // index

loop while j < n

if (!s.Empty()) i = s.pop() else throw error

j++

end-loop

return(i)

end-method

OR

// assume "nth element from the top" is interpreted as

// "top = element 0, remove elements 0, 1, 2, 3, ... n - 1"

// but return value of element n, leaving it on the stack

method pop\_n(stack s, int n)

init i

init int j = 0 // index

loop while j < n

if (!s.Empty()) i = s.pop() else throw error

j++

end-loop

if (!s.Empty()) i = s.peek() else throw error

return(i)

end-method

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.)**

method peek\_bottom(stack s)

init stack t

init i

loop while !s.Empty

t.push(s.pop)

end-loop

i = t.peek

loop while !t.Empty

s.push(t.pop)

end-loop

return(i)

end-method

**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.**

method peek\_third\_from\_bottom(stack s)

init stack t

init i

init j = 0 // counter

loop while !s.Empty

t.push(s.pop)

end-loop

loop while j < 3

if (t.Empty) throw error

else s.push(t.pop)

j++

end-loop

i = s.peek

loop while !t.Empty

s.push(t.pop)

end-loop

return(i)

end-method

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)]**

Iterate through each character... push open delimiters to stack, pop matching closing delimiters off stack

char 01: { stack: {

char 02: [ stack: { [

char 03: A stack: { [

char 04: + stack: { [

char 05: B stack: { [

char 06: ] stack: {

char 07: - stack: {

char 08: [ stack: { [

char 09: ( stack: { [ (

char 10: C stack: { [ (

char 11: - stack: { [ (

char 12: D stack: { [ (

char 13: ) stack: { [

char 14: ] stack: {

Since we have a delimiter left it means that the expressions is malformed...

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

Iterate through each character... push open delimiters to stack, pop matching closing delimiters off stack

char 01: ( stack: (

char 02: ( stack: ( (

char 03: H stack: ( (

char 04: ) stack: (

char 05: \* stack: (

char 06: { stack: ( {

char 07: ( stack: ( { (

char 08: [ stack: ( { ( [

char 09: J stack: ( { ( [

char 10: + stack: ( { ( [

char 11: K stack: ( { ( [

char 12: ] stack: ( { (

char 13: ) stack: ( {

char 14: } stack: (

char 15: ) stack: null

We have no delimiters left, so parenthesis are well-formed (but that doesn't mean that the rest of the expression is!)

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.**

// get\_next\_char will depend on implementation

// return statements also halt execution

// the string 'C' is valid

// string '' is not

method test\_xCy(string x)

init stack s

init char c

loop

c = get\_next\_char(x)

if c is null return(false) // for string to be valid xCy it must have a C

s.push(c)

repeat-loop while c != 'C'

// If we get this far 'C' is on top of the stack

s.pop

loop

c = get\_next\_char(x)

if s.Empty then

if c is null return(true)

else return(false)

end-if

if c is null return(false)

if c != s.pop return(false)

repeat-loop

// Will NEVER get here

end-method

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 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..**

// get\_next\_char will depend on implementation

// return statements also halt execution

// assume that a sting that is valid xCy with \*no\* D is still valid

method test\_aCbDyCz(string x)

init stack s

init char c

loop

// Before we find a 'C', '' is not valid xCy

loop

c = get\_next\_char(x)

if c is null OR c == 'D' return(false)

s.push(c)

repeat-loop while c != 'C'

// If we get this far 'C' is on top of the stack

s.pop

// test after 'C' until 'D' or null

loop

c = get\_next\_char(x)

if s.Empty then

if c is null OR c == 'D' ***exit-loop***

else return(false)

end-if

if c is null OR c == 'D' return(false)

if c != s.pop return(false)

end-loop

repeat-loop while c == 'D'

// if we get here it is because everything passed

return(true)

end-method

1. **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.**

// Not Java, pseudocode

// using Java integer array

// takes jagged arrays to push

// but would need to get actual user requirements

// Also, in practice would need to check if there is room in the private array

// maybe an "autoGrow" method could help?

// Code assumes that arrays of zero length still deserve place on stack!

class StackOfJaggedArrays

private int[PREDEFINED\_INIT\_SIZE] s

private int pos = -1

method push(int[] a)

for(int i = a.length; i > 0; i--)

pos++

s[pos] = a[i - 1]

end-for

pos++

s[pos] = a.length

end-method

method pop()

if pos == -1 return error

init int size

size = s[pos]

pos--

init int[size] a

for(int i = 0; i < size; i++)

a[i] = s[pos]

pos--

end-for

return(a)

end-method

method Empty

return(pos == -1)

end-method

end class

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 and delete into such an array.**

// This assumes that you can push null on to the stack

// If NOT then there would have to be another 2 stacks

// of true and false to track if something is null

class Array

private stack s1 // front of array

private stack s2 // back of array

private int i1 = 0

private int i2 = 0

private int asize = 0

constructor(int size)

asize = size

for(int i = 0; i < size; i++)

s1.push(null)

i1++

end-for

end-constructor

private method rewind(int n)

if n > i1 throw error

for (int i = 0; i < n; i++)

s2.push(s1.pop))

i1--

i2++

next

end-method

private method fast\_forward(int n)

if n > i2 throw error

for (int i = 0; i < n; i++)

s1.push(s2.pop))

i2--

i1++

next

end-method

private method goto\_position(int n)

if n < i1 this.rewind(i1 - n)

if n > i1 this.fast\_forward(n - i1)

end-method

method read(int pos)

// standard `[` method

// pos is zero-indexed (standard)

this.goto\_position(pos + 1)

return(s1.peek)

end-method

method assign(int pos, value)

// standard `[<-` method

// pos is zero-indexed (standard)

this.goto(pos + 1)

s1.pop

s1.push(value)

end-method

method insert(int pos, value)

// put value at position, shifting all other values

// pos is zero-indexed (standard)

this.goto(pos)

s1.push(value)

i1++

asize++

end-method

method delete(int pos)

// delete value at pos, shift others, reduce array size by 1

// pos is zero-indexed (standard)

this.goto(pos + 1)

s1.pop

i1--

asize--

end-method

end class

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.)**

class TwoStacks

private s[SPACESIZE] // type-unknown for now

private int i1 = 0

private int i2 = 0

method push1(val)

if i1 + i2 + 1 > SPACESIZE throw error

i1++

s[i1 - 1] = val

end-method

method push2(val)

if i1 + i2 + 1 > SPACESIZE throw error

i2++

s[SPACESIZE - i2] = val

end-method

method pop1

i1--

return(s[i1])

end-method

method pop2

i2--

return(s[SPACESIZE - i2 - 1])

end-method

end class

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

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

POSTFIX

AB+ \* (C$( DE- )+F)-G

AB+ \* ( CDE-$ )+F)-G

AB+ \* CDE-$F+ - G

AB+CDE-$F+\* - G

AB+CDE-$F+\*G-

PREFIX

+AB \* (C$( -DE ) + F) - G

+AB \* ($C-DE + F) - G

+AB \* +$C-DEF - G

\*+AB+$C-DEF - G

-\*+AB+$C-DEFG

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

POSTFIX

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-$

ABC-DE-\*F+G/$HJ-+

PREFIX

A + (( -BC \* -DE + F) / G)$( -HJ )

A + (( \*-BC-DE + F) / G)$( -HJ )

A + (( +\*-BC-DEF ) / G)$( -HJ )

A + ( /+\*-BC-DEFG )$( -HJ )

A + $/+\*-BC-DEFG-HJ

+A$/+\*-BC-DEFG-HJ

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

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

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

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

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

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

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

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

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

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

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

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

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

A$B - C + D\*E\*F\*G

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

AB-C+DEF-+$

(A - B)C+D(E - F)+$

(A - B + C)(D + (E - F))$

(A - B + C)$(D + E - F)

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

ABCDE-+$\*EF\*-

ABC(D-E)+$\*(E\*F)-

AB(C+D-E)$\*(E\*F)-

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

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

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

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

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$-**

Using method illustrated on slide 27 of the lecture deck

A B + C - B A + C $ -

1 2 3 2 1 3

Operand Stack Operation

1

1 2 +

3

3 3 -

0

0 2

0 2 1 +

0 3

0 3 3 $

0 27 -

-27

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

Using method illustrated on slide 27 of the lecture deck

A B C + \* C B A - + \*

1 2 3 3 2 1

Operand Stack Operation

1

1 2

1 2 3 +

1 5 \*

5

5 3

5 3 2

5 3 2 1 -

5 3 1 +

5 4 \*

20

1. **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.**

// get\_next\_char will depend on implementation

// for this example, get\_next\_char works from right to left

method in\_to\_pre(string x)

init stack s

init string out to empty string

init char c

init char t

if x has non-matching parens throw error and stop

c = get\_next\_char(x)

loop while c is not null

if c is NOT an operator {

prepend "c" to "out"

} else {

// c IS an operator

t = s.peek or NULL if stack is empty

loop while **pop\_arg1?**(t, c) **// def'n next page**

if c is open-paren and t is close-paren then

// matter and anti-matter...

s.pop

exit loop

end-if

prepend s.pop to "out"

t = s.peek or NULL if stack is empty

end-loop

s.push(c)

} // end c IS an operator

c = get\_next\_char(x)

end-loop

loop while stack "s" is not empty

prepend s.pop to "out"

end-loop

return(out)

end-method

// pop\_arg1?(arg1, arg2)

//

// Follows normal precedence rules most of the time

// i.e. if arg1 is higher than arg2 in list below, return TRUE

// $

// \* /

// + -

// NULL

//

// There are some special cases...

// The way I have written this, "(" can never be on the stack

// The only thing that can pull ")" off the stack is "("

// "(" has lower precedence than anything except ")"

// ANY, ")" -> FALSE -> push ")" to stack always

//

// ANY, "(" -> TRUE -> pulls everything off the stack

// ")", Any non-paren -> FALSE

// ")", "(" -> TRUE -> but stop comparing after they obliterate each other

// "(", ")" -> ERROR -> can NEVER happen

// "-", "+" -> TRUE -> + pulls – off the stack

// "+", "-" -> FALSE -> - does not pull + off the stack

// "/", "\*" -> TRUE -> \* pulls / off the stack

// "\*", "/" -> FALSE -> / does not pull \* off the stack

//