Homework 2: Stacks
Alexander Shah
EN.605.202.81 Section 84

Table of Contents

1.

a)

// Sets I to bottom element of stack

Define stack as Stack

Define tempStack as Stack

//while the stack has elements in it, pop them off and put them on tempStack

while stack is not empty

   pop data off stack

   push data to tempStack

//put I at the bottom of the now empty stack

push i on stack

//put back the elements popped off from tempStack

while tempstack isn't empty

     pop data off tempstack

     push data on stack

b)

//Sets I to third most bottom element of stack

Define stack as Stack

Define tempStack as Stack

//while the stack has elements in it, pop them off and put them on tempStack

while stack is not empty

   pop data off stack

   push data to tempStack

//put back 2 elements and insert I at 3rd from the bottom of the stack

count = 0

if tempstack isn't empty and count < 3

pop data off tempstack

push data on stack

increment count

push i on stack

//put back the rest of the elements popped off from tempStack

while tempstack isn't empty

pop data off tempstack

push data on stack

note: if is_empty is not a given method, we can still use 'peek' to see if there is a value on the stack

2.

a.

Check delimiters with a stack

At each delimiter we either push or pop an item on the stack.

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

Item	Action	Stack
{	Push	{
[	Push	[{
]	Pop [	{
[	Push	[{
(	Push	([{
)	Pop (	[{
]	Pop [	{

The delimiters do not match, at the end we are left with "{" in the stack.

b.

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

Item	Action	Stack
(	Push	(
(	Push	((
)	Pop (	(
{	Push	{(
(	Push	({(
[	Push	[({(
]	Pop [	({(
)	Pop (	{(
}	Pop {	(
)	Pop (	

There are no items left on the stack so the delimiters match.

3.

//Read A's and B's into a stack, when you come to a C character, switch to pop and compare, if you have any remaining items, string doesn't match.

Define fwd as Stack

Define compare as Boolean

compare = False

//add A's and B's to a stack until we reach C

for character in inputString

   if compare = False

     if character != 'C'

       push character on to fwd stack

     else

       //then switch over to compare x to y

       compare = True

else

    //check if x matches y, if y is longer than x it will also fail

    if pop(fwd) != character

        return False

//also false if x is longer than y, finally true

if !is_empty(stack)

    return False

else

    return True

4.

//Read in AB's into a stack, when you come to a C, compare stack to rest of string until D, repeat with new stack for new sequence after D

Define fwd as Stack

Define compare as Boolean

compare = False

//add A's and B's to a stack and compare x to y for each xcy between D's

for character in inputString

    // between Ds

    if character != 'D'

        //add X elements before the C to stack

        if compare = False

            if character != 'C'

                push(fwd, character)

            else

                //switch over to compare x to y by comparing stack to rest of string

                compare = True

        else

            //check if x matches y, if y is longer than x it will also fail

            if pop(fwd) != character

return False

    else

        //when we reach a D, reset the stack

        empty(stack)

//xcy is also false if x is longer than y, finally true

if !is_empty(stack)

    return False

else

    return True

5.

Implementing a 1d array with 2 stacks

To initialize such an array we need only one stack, s1, and later s2 can handle temporary storage of elements. We push the elements we want on to s1.

To read elements at a particular index i, we can pop from s1 until we reach i and return its data, by popping the elements of s1 and pushing them onto s2. This will reverse the elements in s1, so we need to push them back onto s1 from s2 to put them back in order.

To insert new elements we can add them onto the end by pushing to the stack, giving the new data an index at the end. Or we can insert elements to a particular index I by first popping the elements off s1 onto s2 until we reach the desired index, inserting the new element(s), and then popping from s2 to push back on s1.

6.

Using a single array to emulate 2 stacks

We can use a single array to hold the data of two stacks without overflow by assigning the tops of the two stacks as the beginning and end of the memory space and having them grow toward each other. We can then check whether each push and pop would put the tops at the same element, in which case we have run out of room when the memory is out of room. We assign s1 to have a top1 at the beginning of the memory space (0 or -1 to start); we assign s2 to have a top2 at the end of the memory space (SPACESIZE or SPACESIZE-1 to start). We can tell when the stacks would overlap by checking whether the tops are overlapping, e.g. top1 + 1 == top2. If the next push wouldn't put the tops in the

same place, we can push some data and increment or decrement the location of the new top element on either stack.

Define s as array

top1 = 0

top2 = size

Method full

   return top1 +1 == top2

Method Push1(data)

  if !full

    s[top1] = data

    top1+=1

Method push2

  if !full

    s[top2] = data

    top2-=1

method pop1

  if top1 == 0

    //empty

  else

    data = s[top1]

    top1 -=1

method pop2

  if top2 == size

    //empty

  else

    data = s[top2]

    top2 +=1

7.

a.

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

convert infix to prefix:

First we reverse the expression into: G -) F +) E - D ($ C (*) B + A (

Then we use a stack

Item	Action	Stack	Prefix
G			G
-	Push	-	G
)	Push	-)	G
F		-)	GF
+	Push	-)+	GF
)	Push	-)+)	GF
E		-)+)	GFE
-	Push	-)+)-	GFE
D		-)+)-	GFED
(	Pop	-)+	GFED-
$	Push	-)+$	GFED-
C		-)+$	GFED-C
(	Pop	-	GFED-C$+
*	Push	-*	GFED-C$+
)	Push	-*)	GFED-C$+
B		-*)	GFED-C$+B
+	Push	-*)+	GFED-C$+B
A		-*)+	GFED-C$+BA
(	Pop	-*	GFED-C$+BA+
[empty]	Pop		GFED-C$+BA+*-

Reverse again to get our answer:

-*+AB+$C-DEFG

converting infix to postfix:

Item	Action	Stack	Postfix
(	push	(	
A		(	A
+	Push	(+	A
B		(+	AB
)	Pop		AB+
*	Push	*	AB+
C		*	AB+C
$	Push	*$	AB+C
(	Push	*$(	AB+C
D		*$(	AB+CD
-	Push	*$(-	AB+CD
E		*$(-	AB+CDE
)	Pop	*$	AB+CDE-
+	Push	*$+	AB+CDE-
F		*$+	AB+CDE-F
)	Pop	*	AB+CDE-F+$
-	Push	*-	AB+CDE-F+$
G		*-	AB+CDE-F+$
[empty]	Pop		AB+CDE-F+$G-*

Which gets us our answer:

AB+CDE-F+$G-*

b .

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

converting infix to prefix

reverse expression:) J – H ($) G /) F +) E – D (*) C – B (((+ A

then use stack

Item	Action	Stack	Prefix
)	Push	)	
J		)	J
-	Push	)-	J
H		)-	JH
(	Pop		JH-
$	Push	$	JH-
)	push	$)	JH-
G		$)	JH-G
/	Push	$)/	JH-G
)	Push	$)/)	JH-G
F		$)-)	JH-GF
+	Push	$)-)+	JH-GF
)	Push	$)/)+)	JH-GF
E		$)/)+)	JH-GFE
-	Push	$)/)+)-	JH-GFE
D		$)/)+)-	JH-GFED
(	Pop	$)/)+	JH-GFED-
*	Push	$)/)+*	JH-GFED-
)	Push	$)/)+*)	JH-GFED-
C		$)/)+*)	JH-GFED-C
-	Push	$)/)+*)-	JH-GFED-C
B		$)/)+*)-	JH-GFED-CB
(	Pop	$)/)+*	JH-GFED-CB-

(	Pop	$)/	JH-GFED-CB-*+
(	Pop		JH-GFED-CB-*+/$
+	Push	+	JH-GFED-CB-*+/$
A		+	JH-GFED-CB-*+/$A
[empty]	Pop		JH-GFED-CB-*+/$A+

Reverse again to get our answer:

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

Converting infix to postfix: A + (((B - C) * (D - E) + F) / G) $ (H - J)

Item	Action	Stack	Postfix
A			A
+	Push	+	A
(	Push	+(	A
(	Push	+((	A
(	Push	+(((	A
B		+(((	AB
-	Push	+(((-	AB
C		+(((-	ABC
)	Pop	+((	ABC-
*	Push	+((*	ABC-
(	Push	+((*(	ABC-
D			ABC-D
-	Push	+((*(-	ABC-D
E		+((*(-	ABC-DE
)	Pop	+((*	ABC-DE-
+	Push	+((*+	ABC-DE-
F		+((*+	ABC-DE-F
)	Pop	+(	ABC-DE-F+*

/	Push	+(/	ABC-DE-F+*
G		+(/	ABC-DE-F+*G
)	Pop	+	ABC-DE-F+*G/
$	Push	+$	ABC-DE-F+*G/
(	Push	+$(	ABC-DE-F+*G/
H		+$(	ABC-DE-F+*G/H
-	Push	+$(-	ABC-DE-F+*G/H
J		+$(-	ABC-DE-F+*G/HJ
)	Pop	+$	ABC-DE-F+*G/HJ-
[empty]	Pop		ABC-DE-F+*G/HJ-$+

So we get our answer:

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

8.

a.

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

Converting prefix to infix:

Item	Stack
I	I
H	I,H
G	I,H,G
*	I,(G*H)
F	I,(G*H),F
E	I,(G*H),F,E
+	I,(G*H),(F+E)
/	I,(G*H)/(F+E)
D	I,(G*H)/(F+E),D

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

Which gives us:

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

b .

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

Converting prefix to infix

Item	Stack
G	G
F	G,F
E	G,F,E
*	G,(F*E)
*	G*(F*E)
D	G*(F*E),D
*	D*(G*(F*E))
C	D*(G*(F*E)),C
B	D*(G*(F*E)),C,B
A	D*(G*(F*E)),C,B,A
$	D*(G*(F*E)),C,(B$A)
-	D*(G*(F*E)),C-(B$A)
+	D*(G*(F*E))+C-(B$A)

Which gives us our answer:

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

c.

A B - C + D E F - + $

Converting postfix to infix

Item	Stack
A	A
B	A,B
-	(A-B)
C	(A-B),C
+	(A-B)+C
D	(A-B)+C, D
E	(A-B)+C, D, E
F	(A-B)+C, D, E, F
-	(A-B)+C, D, (E-F)
+	(A-B)+C, D+(E-F)
$	((A-B)+C)$(D+(E-F)

So we get our answer :

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

d.

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

converting postfix to infix

Item	Stack
A	A,
B	A,B

C	A,B,C
D	A,B,C,D
E	A,B,C,D,E
-	A,B,C,(D-E)
+	A,B,(C+(D-E))
$	A,(B$(C+(D+E))
*	A*(B$(C+(D+E)))
E	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)

We get our answer

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

9.

a.

A B + C - B A + C $ -

Converting postfix to infix

Item	Stack
A	A
B	A,B
+	(A+B)
C	(A+B), C
-	(A+B)-C
B	(A+B)-C, B
A	(A+B)-C, B, A
+	(A+B)-C, (B+A)

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

So we get

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

If A=1, B=2, and C=3

then ((1+2)-3)-((2+1)^3) =

((3)-3)-((3)^3) =

0 – 3^3 =

 -27

b.

A B C + * C B A - + *

Converting postfix to infix

Item	Stack
A	A
B	A, B
C	A, B, C
+	A, (B+C)
*	A*(B+C)
C	A*(B+C), C
B	A*(B+C), C, B
A	A*(B+C), C, B, A
-	A*(B+C), C, (B-A)
+	A*(B+C), C+(B-A)
*	A*(B+C)*C+(B-A)

So we get

A*(B+C)*C+(B-A)

If A=1, B=2, and C=3

then 1*(2+3)*3+(2-1) =

 1*5*3+1 =

 16

10.

A method to transform infix to prefix expressions when read right to left, operator precedence may flip so higher ops will be appended before lower ones compared to when reading left to right.

Define s as stack

Define output as String

while reversed input is not empty, read character

 if operand, append operand to output

 if operator, pop and append operators on top of the stack until a lower precedence operator, left parentheses, or empty stack, then append operator

 if open parentheses "(", pop and append operators from stack until close parentheses, pop and append ")"

append any remaining items in stack