STANDARD TEMPLATE LIBRARY STACKS

Problem Solving with Computers-II

C++STL

- The C++ Standard Template Library is a very handy set of three built-in components:
  - Containers: Data structures that are generic (store any kind of data)
  - Iterators: Standard way to search containers think of them as generic pointers
  - Algorithms: These are what we ultimately use to solve problems

The STL separates std. algorithms from containers.
The other option is that every container class implements its own copy of the algorithm.
With N containers, and M algorithms the second approach will result in NxM implementations. Not efficient!
That's why the STL separates some of the algorithms from the data structures

C++ STL container classes

```
array → arrays
vector → dynamic array
forward_list → singly linked list
list → doubly linked list
stack → stack
queue → queue
priority_queue → Heap
set → BST (balanced)
multiset (non unique keys)
deque
unordered_set
map
unordered_map
multimap
bitset
```

CS 32!

Stacks – container class available in the C++ STL

- Container class that uses the Last In First Out (LIFO) principle
- Methods

i. push() → push to the top of the stack

ii. pop() → delete the element on the top (does not return a value)

iii. top() → returns the element on the top (does not delete it)

iv. empty() → used to check if the stack is empty

~~Demo reversing a string~~

Notations for evaluating expression

- Infix number operator number
- Prefix operators precede the operands
- Postfix operators come after the operands

(7 + (3 * 5)) – (4 / 2)

operands

operator

/ 4 2

Infix	Prefix	Postfix
3*5	* 3 5	3 5 *
7 + (3*5)	+ 7 * 3 5	7 3 5 * +
4/2	/ 4 2	4 2 /
(7+(3*5)) - (4/2)	- + 7 * 3 5 / 4 2	7 3 5 * + 4 2 / -

Lab05 – part 1: Evaluate a fully parenthesized infix expression

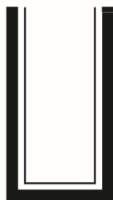
(((( ))) unbalanced

```
( 4 * ( ( 5 + 3.2 ) / 1.5 ) ) // okay

( 4 * ( ( 5 + 3.2 ) / 1.5 ) // unbalanced parens - missing last ')'

( 4 * ( 5 + 3.2 ) / 1.5 ) ) // unbalanced parens - missing one '('

4 * ( ( 5 + 3.2 ) / 1.5 ) // not fully-parenthesized at '*' operation

( 4 * ( 5 + 3.2 ) / 1.5 ) // not fully-parenthesized at '/' operation
```

))) (((not balanced

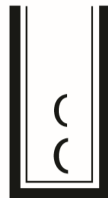
((2 * 2) + (8 + 4))

Initial empty stack



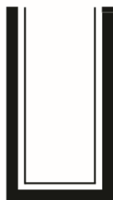
Read and push first (

Read and push second (



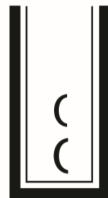
((2 * 2) + (8 + 4))

Initial empty stack



Read and push first (

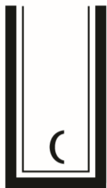
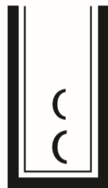
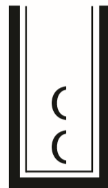
Read and push second (



What should **be the next step after the first right parenthesis is encountered**?

A. Push the right parenthesis onto the stack
B. If the stack is not empty pop the next item on the top of the stack
C. Ignore the right parenthesis and continue checking the next character
D. None of the above

((2*2)+(8+4))

Initial empty stack	Read and push first (	Read and push second (	Read first) and pop matching (	Read and push third (	Read second) and pop matching (	Read third) and pop the last (
						

Evaluating a fully parenthesized infix expression

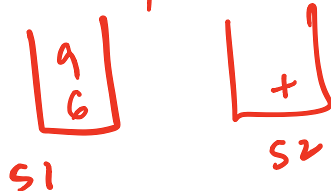
Assumes the expression has balanced parenthesis and is fully parenthesized

(((6 + 9)/3)*(6 - 4))

Use two stacks

Stack S1 : stores the operands
Stack S2 : stores the operators

1) Read expression left to right.
Push operands in S1, push operators in S2



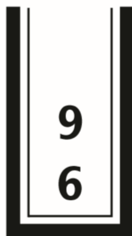
2) When you encounter a right parenthesis ')'
pop two operands and one operator ; evaluate, and store the sub expression back in S1

Evaluating a fully parenthesized infix expression

Characters read so far (shaded):

(((6 + 9) / 3) * (6 - 4))

Numbers



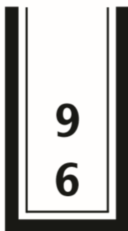
Operations

Evaluating a fully parenthesized infix expression

Characters read so far (shaded):

```
(((6 + 9) / 3) * (6 - 4))
```

Numbers



Operations

6 + 9 is 15

Numbers

Operations

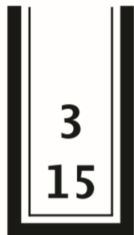
Before computing 6 + 9

After computing 6 + 9

Evaluating a fully parenthesized infix expression

Characters read so far (shaded):

(((6 + 9) / 3) * (6 - 4))



Before computing 15/3

15 / 3 is 5

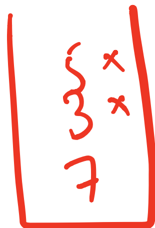
After computing 15/3

Lab 05, part2 :
Evaluating post fix expressions using a single stack

Postfix: 7 3 5 * + 4 2 / -

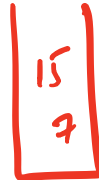
Infix: (7 + (3 * 5)) – (4 / 2)

Pop last two operands



5*3 = 15

when * is read

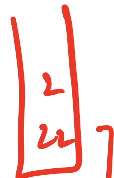


15+7

when + is read

4/2

when / is read



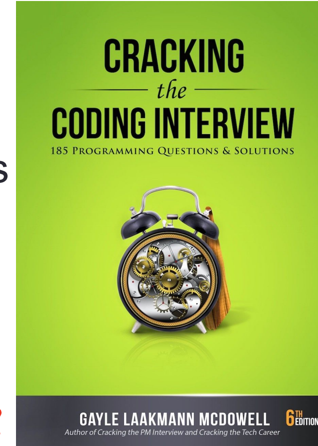
-

22-2

Small group exercise

Write a ADT called in minStack that provides the following methods

- push() // inserts an element to the "top" of the minStack
- pop() // removes the last element that was pushed on the stack
- top () // returns the last element that was pushed on the stack
- min() // returns the minimum value of the elements stored so far

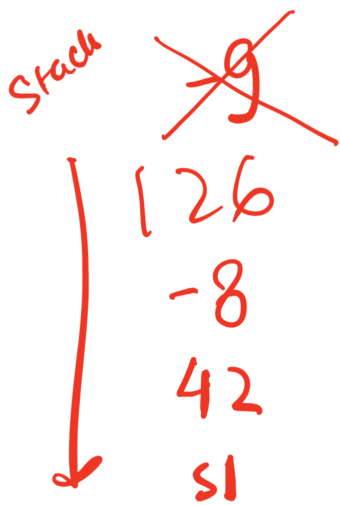


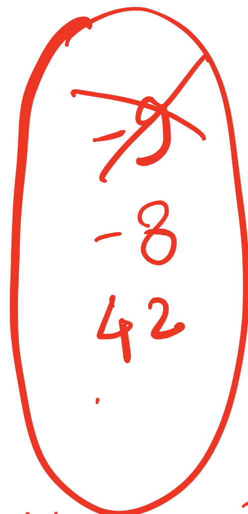
When we pop(), -9 is removed, the overall min should then be -8

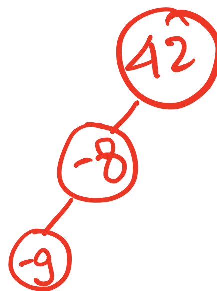
-9 min(-9)
126 min(-8)
-8 min(-8)
42 min(42)

You can use any of the container classes covered so far!

array, vector, list, set, stack!







Use a second stack to remember the min overall

→ could use a bst but in that case min is not guaranteed to be O(1) in the worst case