STACK AND QUEUE

Problem Solving with Computers-II

```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

C++STL

- The C++ Standard Template Library is a very handy set of three built-in components:
  - Containers: Data structures
  - Iterators: Standard way to search containers
  - Algorithms: These are what we ultimately use to solve problems

C++ STL container classes

```
         array
        vector
  forward_list
          list
           set
         stack
         queue
priority_queue
multiset (non unique keys)
         deque
 unordered_set
           map
 unordered_map
      multimap
        bitset
```

Stacks – container class available in the C++ STL

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

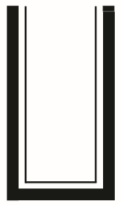
i. push()
ii. pop()
iii. top()
iv. empty()

Lab05: Evaluate a fully parenthesized infix expression

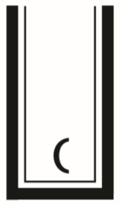
```
( 4 * ( ( 5 + 3.2 ) / 1.5 ) ) // okay

( 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
```

Checking if the parenthesis are balanced

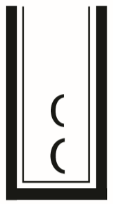
Initial empty stack



Read and push first (

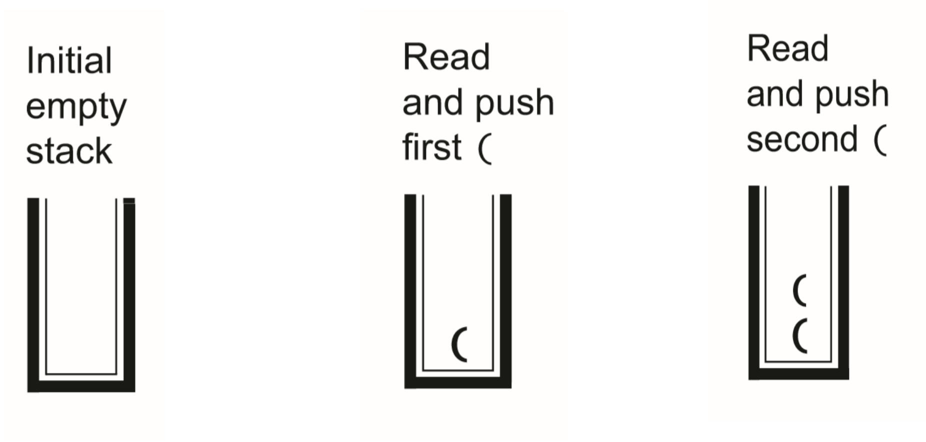


Read and push second (



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

Checking if the parenthesis are balanced



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

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 (

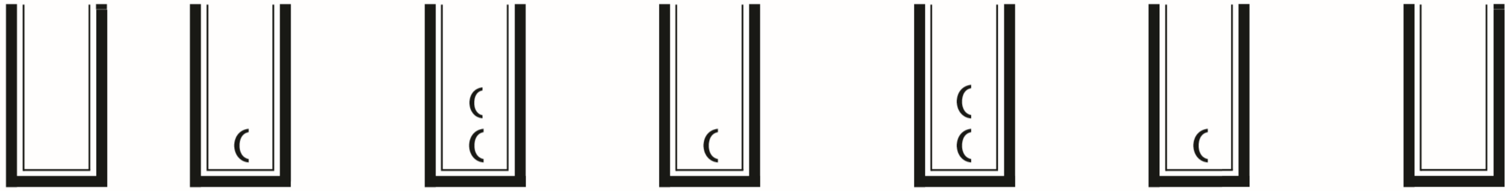
Read and push second (

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

evaluate

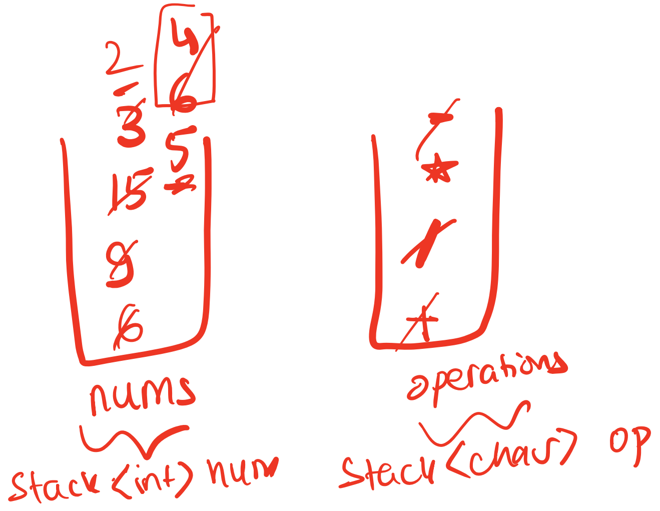
(((6 + 9)/3)*(6 - 4))

0 1 2 3 4 5 6

A. One stack

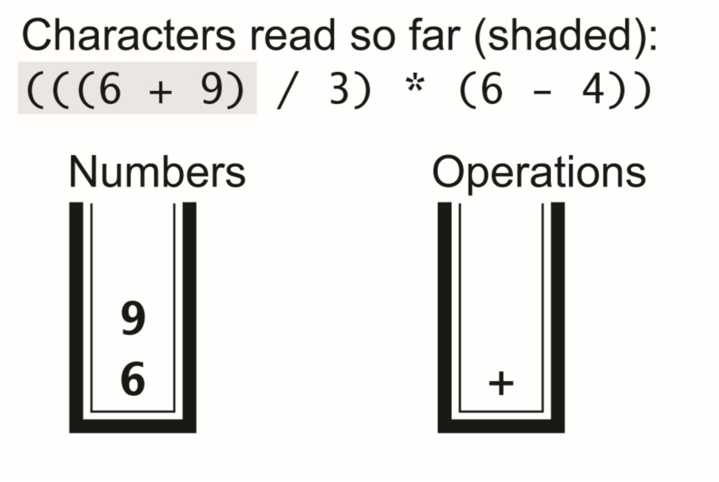
B. Two stacks

C. Three or more



$\boxed{10}$

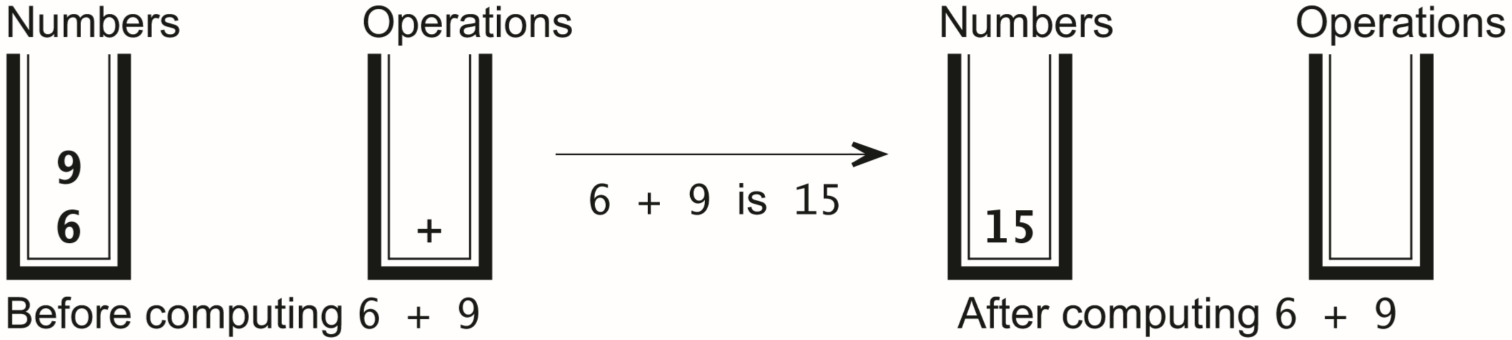
Evaluating a fully parenthesized infix expression



Evaluating a fully parenthesized infix expression

Characters read so far (shaded):

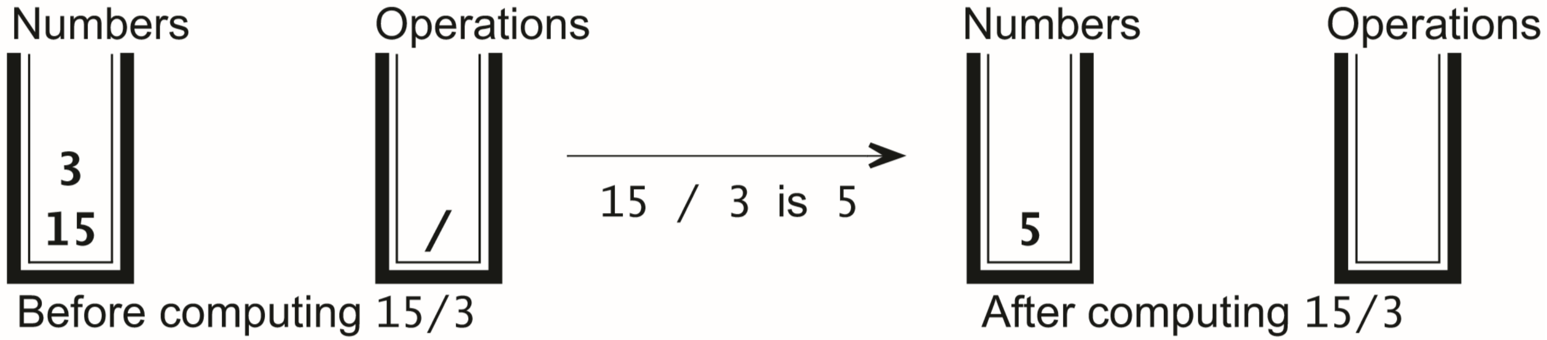
```
(((6 + 9) / 3) * (6 - 4))
```



Evaluating a fully parenthesized infix expression

Characters read so far (shaded):

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



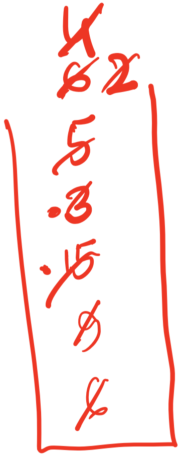
Notations for evaluating expression

- $\boxed{\text{Infix}}$ number operator number $7+5$
- (Polish) Prefix operators precede the operands $+\ 7\ \ 5$
- (Reverse Polish) Postfix operators come after the operands $7\ \ 5\ +$

Evaluating post fix expressions using a single stack

Postfix: 6, 9, +, 3, / 6, 4, -, *

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



$\boxed{10}$

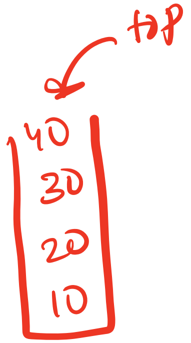
15 / 3

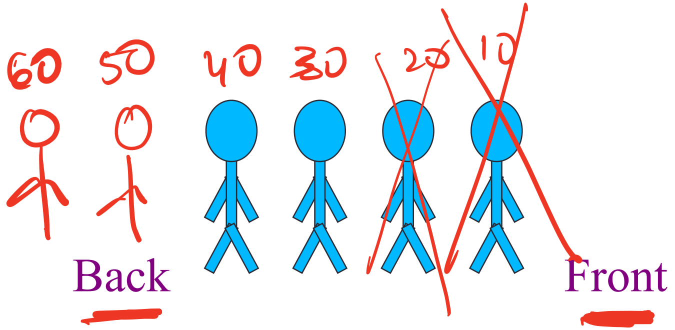
a b

Stack <int> nus

Queue Operations

- A queue is like a queue of people waiting to be serviced
- The queue has a **front** and a **back**.





STL

queue<int>q;

q.push(50)

q.push(60)

q.back()

q.front()

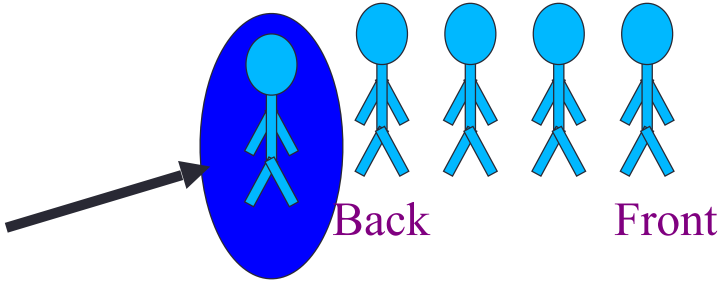
q.pop()

// deleting the front element

empty()

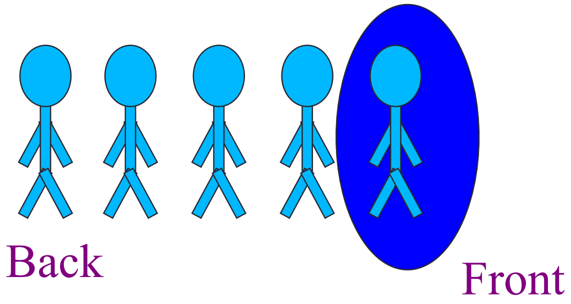
Queue Operations

- New people must enter the queue at the back. The C++ queue class calls this a **push**, although it is usually called an **enqueue** operation.



Queue Operations

- When an item is taken from the queue, it always comes from the front. The C++ queue calls this a **pop**, although it is usually called a **dequeue** operation.

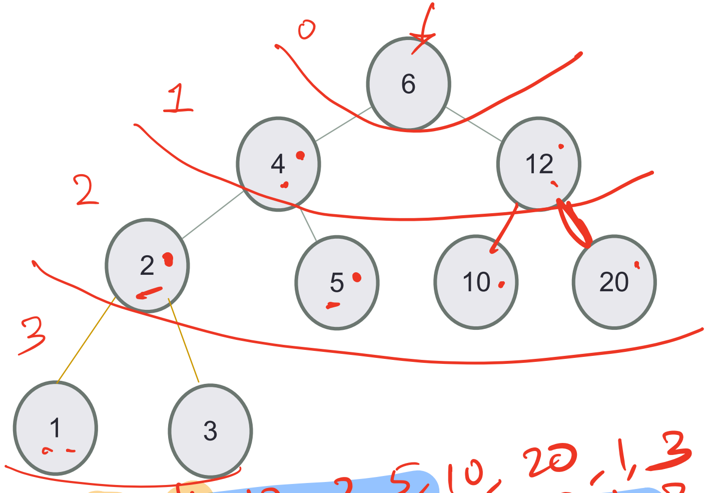


Queue Class

- The C++ standard template library has a queue template class.
- The template parameter is the type of the items that can be put in the queue.

```
template <class Item>
class queue<Item>
{
public:
        queue( );
        void push(const Item& entry);
        void pop( );
        bool empty( ) const;
        Item front( ) const;
        …
```

Breadth first traversal



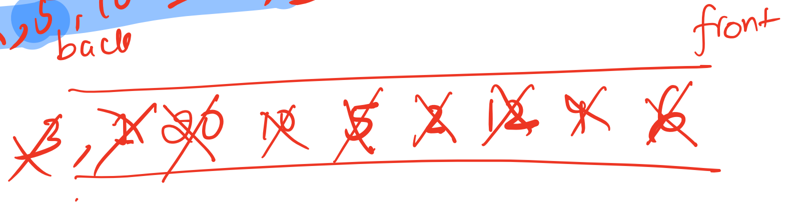
Algo BFS.

- Take an empty Queue.
- Start from the root, insert the root into the Queue.
- Now while Queue is not empty,
  - Extract the node from the Queue and insert all its children into the Queue. (if they exist)
  - Print the extracted node.

6, 4, 12, 2, 5, 10, 20, 1, 3

6, 4, 12, 2, 5, 10 20 1, 3

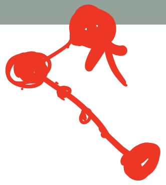
6, 4, 12, 2, 5, back



front
$\cancel{3}$, $\cancel{1}$ $\cancel{20}$ $\cancel{10}$ $\cancel{5}$ $\cancel{2}$ $\cancel{12}$ $\cancel{4}$ $\cancel{6}$

Summary of operations

$n \leftarrow$ no. of key



Operation	Sorted Array	Binary Search Tree		Linked List		stack	queue
	worst case	regular	balanced (set)	sorted	unsorted		
Min	$O(1)$	$O(n)$	$O(\log n)$			✗	✗
Max	$O(1)$	$O(n)$	$O(\log n)$			✗	✗
Median	$O(1)$	$O(n)$	$O(n)$			✗	✗
Successor	$O(1)$	$O(n)$	$O(\log n)$			✗	✗
Predecessor	$O(1)$	$O(n)$	$O(\log n)$			✗	✗
Search	$O(\log n)$ Binary Search	$O(n)$	$O(\log n)$			✗	✗
Insert	$O(n)$	$O(n)$	$O(\log n)$			$O(1)$	$O(1)$
Delete	$O(n)$	$O(n)$	$O(\log n)$			$O(1)$	$O(1)$