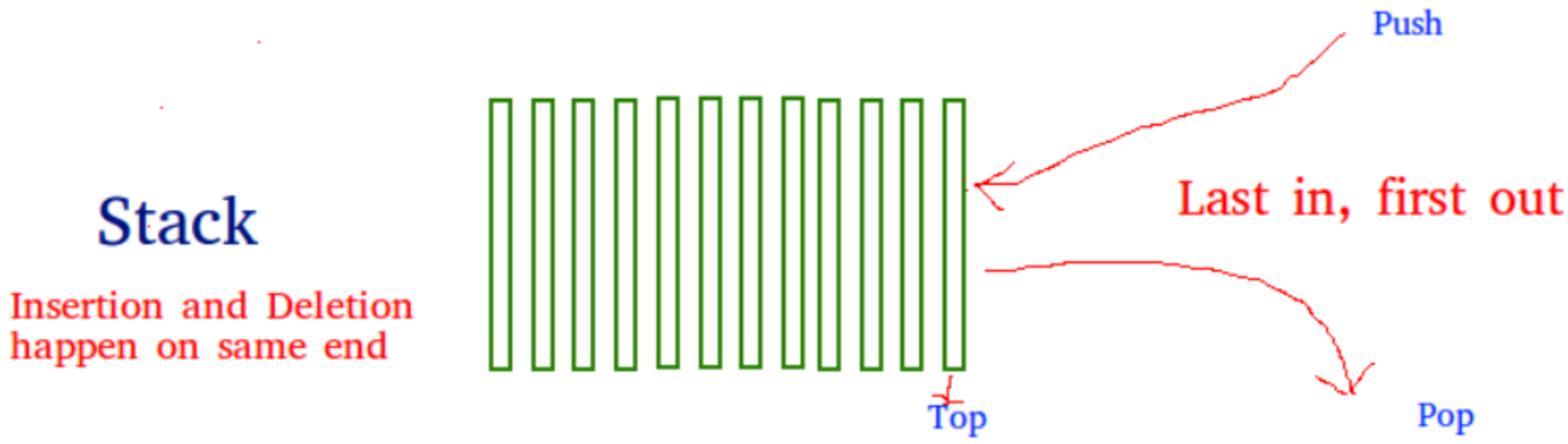
Stack in C++ STL

In general, Stack is a linear data structure where elements are inserted in the **LIFO** order. LIFO stands for **L**ast **I**n **F**irst **O**ut, which means that the last element to be inserted in the stack will be the first one to be removed.

C++ STL has an in-built container adaptor which implements the stack data structure internally. We can directly use this container to implement stacks in our programs and perform a number of operations available.

Insertion and Deletion in a stack are done from a single end which is the rear end of the stack data structure.



Example:

```
Let's say we inserted 4 items, {10, 20, 30, 40}
in a stack:

The stack will look like:
++++++++
+  40  +
++++++++
+  30  +
++++++++
+  20  +
++++++++
+  10  +
++++++++

On deleting element 40, stack will be:
++++++++
+  30  +
++++++++
+  20  +
++++++++
+  10  +
++++++++

Again, on deleting 30, the stack will be:
++++++++
+  20  +
++++++++
+  10  +
++++++++

Now, if we insert an element 40 again, it will
be inserted again at the top:
++++++++
+  40  +
++++++++
+  20  +
++++++++
+  10  +
++++++++
```

Syntax:

```
stack<data_type> stack_name;
```

Note: The stack class is available in the header file . Therefore, one must include this header file in programs while using the stack class of C++ STL.

Some important functions of Stacks in C++ STL: Below is a list of some of the most commonly used functions which are available with Stack in C++ STL.

The functions associated with stack are:

- **push(g)**: The push function adds an element 'g' passed as parameter to it at the top of the stack.
- **pop()**: The pop() function deletes the topmost element of the stack.
- **top()**: This function returns a reference to the topmost element of the stack.
- **size()**: The size() function returns the size of the stack, i.e., the number of elements present in the Stack.
- **empty()**: This function is used to check whether the stack is empty. If the stack is empty it will return True otherwise it will return False.

Below program illustrate the above-discussed functions of Stacks in C++ STL:

```cpp

// C++ program to illustrate Stack in STL

#include<iostream>
#include<stack>

using namespace std;

int main()
{
    // Creating a Stack using STL
    stack<int> s;

    // The push function inserts elements
    // at the top of Stack
    s.push(10);
    s.push(20);
    s.push(30);

    // After above operations, stack will
    // look like:
    //     ******
    //     * 30 *  <- Stack TOP
    //     * 20 *
    //     * 10 *
    //     ******

    // This will print the current size
    // of stack which is 3
    cout<<s.size()<<endl;
```

Run

Output:

```
3
30
20
```

Stack Traversal

We cannot access stack elements randomly as they are not stored at contigous memory locations. Hence, we cannot use loops or iterators to traverse stacks.

However, in some cases, if we want to remove a few elements from stack or empty a stack we can do that by using some built-in functions we discussed above.

The idea is:

- While, stack is not empty, i.e., there are some elements in stack.
    1. Print the top element.

    2. Pop the top element.

  Repeat the above 2 steps until stack is not empty.

Below program illustrate traversal of the stack:

```cpp

// C++ program to illustrate Stack traversal
```

```cpp

#include<iostream>
#include<stack>

using namespace std;

int main()
{
    // Creating a Stack using STL
    stack<int> s;

    // The push function inserts elements
    // at the top of Stack
    s.push(10);
    s.push(20);
    s.push(30);

    // After above operations, stack will
    // look like:
    //     ******
    //     * 30 *  <- Stack TOP
    //     * 20 *
    //     * 10 *
    //     ******

    // While stack is not empty
    while(!s.empty())
    {
```

Run

Output:

```
30 20 10
```

Note: Elements in the above traversal will be printed in reverse order of their insertion.

Time Complexities and Internal Working

All of the above five functions we discussed works in constant time complexity.

```
push()    -----
pop()     -----  \
top()     -----    ---  O(1) Time Complexity.
size()    -----  /
empty()   -----
```

Container Adaptor: In C++ STL, container adaptor is a term used for containers that can be built upon other containers. That is, they can be implemented using other containers already available in C++ STL.

The **stack** is a container adaptor and can be implemented using any other container which supports below operations:

- Inserting elements to the end.
- Deleting elements from the end.
- Returning size of the container.
- Returning element from the end.
- Checking if the container is empty.

Sample Problem: Next greater element

Problem: Given an array, print the Next Greater Element(NGE) for every element. The Next greater Element for an element x is the first greater element on the right side of x in the array. Elements for which no greater element exist, consider next greater element as -1.

Example:

```
Input: arr[] = {4, 5, 2, 25}
Output:
```

Input: arr[] = {4, 5, 2, 25}
Output:

```
Element       NGE
   4      -->   5
   5      -->   25
   2      -->   25
   25     -->   -1
```

Input: arr[] = {1, 2, 4, 8, 6, 10}
Output:

```
Element       NGE
   1      -->   2
   2      -->   4
   4      -->   8
   6      -->   10
   8      -->   10
   10     -->   -1
```

Approaches

Method 1: A naive method.
Use of two loops: The outer loop picks all the elements one by one. The inner loop looks for the first greater element for the element picked by the outer loop. If a greater element is found then that element is printed as next, otherwise, -1 is printed.

Time complexity: Time taken by above algorithm is $O(N^2)$

Method 2: Using of Stack.

Step 1: Push the first element to stack.
Step 2: Pick rest of the elements one by one and follow the following steps in loop.

a. Mark the current element as *next*.
b. If stack is not empty, compare top element of stack with *next*.
c. If next is greater than the top element, Pop element from stack. *next* is the next greater element for the popped element.
d. Keep popping from the stack while the popped element is smaller than *next*. *next* becomes the next greater element for all such popped elements

Step 3: Finally, push the next in the stack.
After the loop in step 2 is over, pop all the elements from stack and print -1 as next element for them.

Time Complexity: O(n)
The worst case occurs when all elements are sorted in decreasing order. If elements are sorted in decreasing order, then every element is processed at most 4 times.

```cpp

// A Stack based C++ program to find next
// greater element for all array elements.
#include <bits/stdc++.h>
using namespace std;

// Prints element and NGE pair for all
// elements of arr[] of size n
void printNGE(int arr[], int n)
{
    stack<int> s;

    // Push the first element to stack
    s.push(arr[0]);

    // Iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;
        }

        // if stack is not empty, then
        // pop an element from stack.
```

```
26      // if the popped element is smaller
27      // than next, then
28      // a) print the pair
29      // b) keep popping while elements are
30      // smaller and stack is not empty
```

Run

Output:

```
4 --> 5
2 --> 25
5 --> 25
25 --> -1
```

Sample Problem: Check if a sequence is balanced or not

Problem: Given a sequence of small open and closed parenthesis, you need to check and return if the parenthesis is balanced or not.

```
Input: string = "(())(())"
Output: 1

Input: string = "(()))"
Output: 0
```

Approach:

Step1: Declare a stack of characters.

Step2: Traverse the given expression and if the character is open parenthesis then push it in the stack.

Step3: If the character is closed parenthesis then remove that character from stack but you also need to check whether stack is empty or not for avoiding empty exception and return 0 if it so.

Step4: Finally check of the stack is empty or not, if it is empty then return 1 otherwise 0 i.e. if containing open parenthesis return 0.

Time complexity: O(N)

Auxiliary space: O(N)

```
1
2   // Program to check whether the given
3   // bracket sequence is balanced or not.
4   #include<bits/stdc++.h>
5   using namespace std;
6
7   // Function to check the balance
8   bool isBalanced(string str)
9   {
10      // Creating a stack
11      stack<char> st;
12
13      // Iterating through the string
14      // of braces
15      for(int i = 0; i < str.length(); i++)
16      {
17          // Pusing the elements into stack
18          if(str[i] == '(')
19              st.push(str[i]);
20
21          // Popping the elements out
22          else
23          {
24              if(st.empty())
25                  return 0;
26              st.pop();
27          }
```

```
    }

    // Checking the balance
```

Run

Sample Problem: Reverse a String using stack

Problem: Given a String reverse it using Stack.

```
Input: string = "GeeksQuiz"
Output: "ziuQskeeG"

Input: string = "Welcome to GFG"
Output: "GFG ot emocleW"
```

Approach: Create a stack of characters and store all the characters of the string in it. After that, you can remove elements one by one from the stack and print it.

Time Complexity: The time taken for this algorithm is O(N) as we are inserting all the elements of string in the stack and then removing it from the stack.

```

// Program to print the reverse of string
#include <bits/stdc++.h>
using namespace std;

// Function to perform the
// reverse operation
void reverse(string str)
{
    // Creating a stack of characters
    stack<char> st;

    // Pushing elements into the stack
    for (int i = 0; i < str.length(); i++)
        st.push(str[i]);

    // Popping elements from the top
    // to get the reverse order
    while (!st.empty()) {
        cout << st.top();
        st.pop();
    }
}

// Driver Method
int main()
{
    // String to reverse
    string str = "GeeksQuiz";
    reverse(str);
```

Run

Output:

```
ziuQskeeG
```

Sample Problem: Evaluation of Postfix Expression

Problem: Given a string of expression, evaluate the eqivalent Postfix expression for it.

Examples:

```
Input: "231*+9-"
```

```
Output: -4

Input: "570*+6-"
Output: -1
```

Approach:

Step 1: Create a Stack to store operands(values)

Step 2: Scan the given expression and do following for every scanned element.

a. If the element is a number, push it into the stack

b. If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.

Step 3: When the expression ends, the number in the stack is the final answer

Time complexity: The evaluation of algorithm takes O(n) time, where n is the number of characters in input expression.

```cpp

// Program to evaluate the
// postfix expression
#include <bits/stdc++.h>
using namespace std;

// Function to evaluate postfix
// expressions
int evaluatePostfix(string str)
{
    // Creating an empty stack
    stack<int> st;
    int n = str.length();

    // Traversing the string
    for (int i = 0; i < n; i++) {
        // Pushing elements into stack
        // if element is a number
        if (isdigit(str[i]))
            st.push(str[i] - '0');

        // Evaluation of the expression
        else {
            int val1 = st.top();
            st.pop();
            int val2 = st.top();
            st.pop();
            char op = str[i];

            // Checking for the operators
```

Run

Output:

```
-4
```