GeeksMan

Data Structure

Lesson - 1

Trilok Kaushik

Founder of GeeksMan

Chirag Soni Nupur Pahuja Jessica Mishra

Team Co-ordinators

STACK

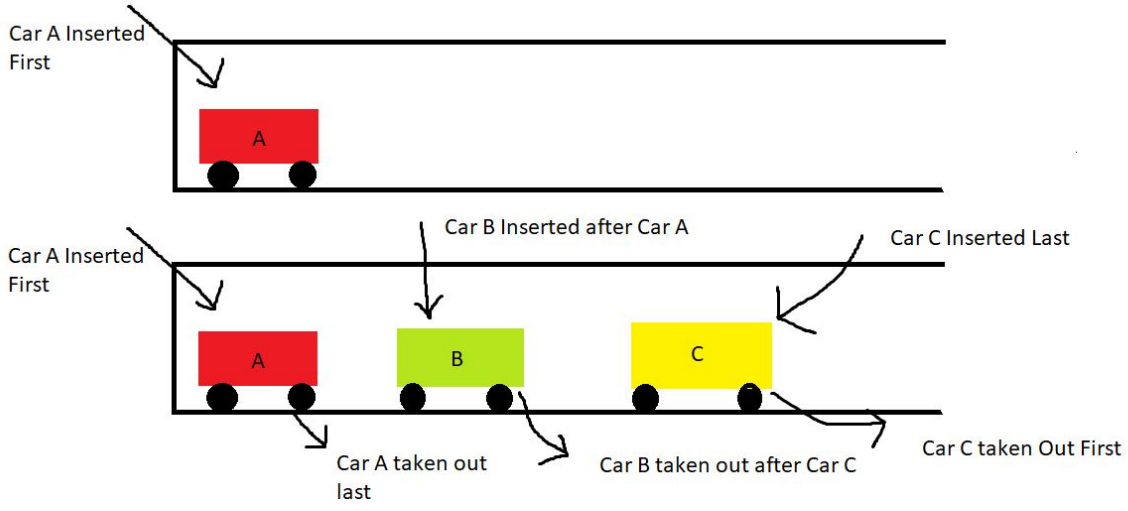
Part-1

Stack is a Data Structure which works on the Principle of **LIFO**.

Now, what do we mean by LIFO.LIFO stands for Last In First Out. As the name suggests , LIFO means the element which is Inserted last is taken out first. This concept of LIFO is used by us numerous times in Real life.

Let us consider a Simple real life Situation. Suppose there is a Parking slot with space for 3 cars and till now no car is parked in that slot. Now Person A comes and finds the Parking Slot empty and Parks his car Inside the Parking slot. After he had parked the car , Person B came and parked his car behind the Car of Person A and after some time Person C came and found that there was space for 1 Car , and so he parked his Car behind Person's B car. Now if Person A wants to take his Car out from that Slot, he cannot do this Until Person B takes out his car and Person B can't take out his car until Person C has taken out his Car. So, Person A who parked his Car first will be able to take out his Car last And Person C who parked his Car last will be able to take it first. This is LIFO.

Stack also works on the Same principle. The Insertion and Deletion operation on stack elements are done on the principle of LIFO. The Element inserted (generally called '**pushed**') first will be removed (generally called '**popped**') from the stack last and vice-versa.



Now, Considering this Interesting LIFO principle on which Stack is Based, we can think of a very Interesting application of stack which is Reversing Strings. Let us Consider a String **'Geeksman'**.There are many ways to reverse this string, but the LIFO Principle of STACK makes the task much easier.

We will start storing individual characters of this string in our stack.Consider stack as an array placed vertical on a floor. Now, 'G' being the first character of this string will come first in the stack and hence will occupy the bottom most position in the stack and 'n' being the last character of the string will be placed last in the stack and hence will occupy the top-most position in the Stack. As we start taking out elements from that stack the topmost character i.e 'n' will be taken out first and will become the first character of our output String and 'G' being the bottom-most element of stack will be taken out last and will be the last character of our Output String and hence by using stack we have reversed our string from **'Geeksman'** to **'namskeeG'**. Hence, by using stack and utilizing its LIFO principle we can do various tasks easily.

ADT of Stack

Now, let's talk about ADT of Stack. ADT stands for **Abstract Data Type**.

It provides 2 Important Information about any Data Structure as mentioned below:

1)How to represent our Data Structure in Memory.

2)List of Operations on that Data Structure.

So, whenever we say ADT of stack ,we are talking about what things we need to represent our Stack in Memory and what operations we can do on Stack once we have represented it in our Memory.So let's talk about Data representation and operations on Stack one by one.

Data Representation:-

So to represent Stack in our Memory ,we need to know about the following points:

1)*Space for Storing Elements:-*

Since stack is a data structure which stores elements, we need some empty space in our memory so that we can store our elements inside stack.

2)*Top Pointer:-*

As we have seen in our previous examples, elements are always taken out from the top of stack.Hence we require a top pointer which will always point to the top-most element of the stack and will keep track of the current element on top which has to be removed(**popped**).

Operations on Stack:-

Following are some of the basic operations applied on stack:-

1) **push(x)** Inserting an element x on top of Stack.
2) **pop()** Removing the top-most element from the Stack.
3) **peek(index)** Looking at a value on Particular index of the stack.
4) **Stacktop()** Looking at the topmost value of the Stack.
5) **isEmpty()** Finding out whether the Stack is Empty or not.
6) **isFull()** Finding out whether the Stack is Full or not.

We will Study the Implementation of all these operations Later in the Chapter in Detail.

Part-2

STACK IMPLEMENTATION

Stack can be Implemented using 2 Data Structures:-

1) Array 2) Linked List

Stack using Array:-

Array is one of the most Common Data Structure which is used to Implement Stack since Insertion and deletion of elements in an array is easy and fast. To Implement stack using array we require three things:-

1)**An Array of fixed size** to store elements.

2)**A top pointer**:-Since all insertion and deletion operations are performed on top of the stack, we require a top pointer which will always point to the top of the stack.

3)**A size variable** which will store the size of the array which we are using. This size variable will help us to know whether the stack is full or not.If the value of top pointer becomes equal to the value of size variable, it means the stack is full now and hence we cannot insert any more elements in the stack and in order to add more elements inside the stack,we have to pop out at least one element from the stack.

An array has 2 ends i.e. **Front End** and **Rear End** and element can be inserted from either of the end. But,we have to insert the element from that end which is most optimal in terms of time taken to insert the element.

When we insert an element from the rear end of an array, it takes just O(1) time complexity to insert the element since the top pointer is always pointing

to the empty space available on top of the array and we all know that inserting an element at a particular index in array just takes **O(1)** time Complexity.

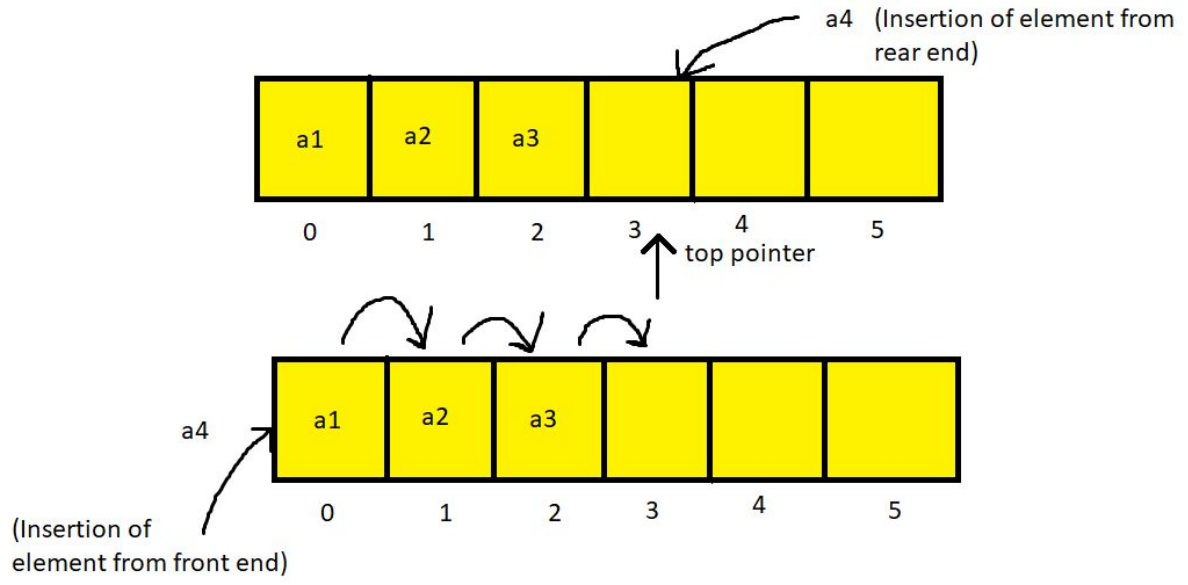


Fig. 1.2(a)

But when we insert an element from the front side of an array, the element is always inserted at index 0, but as we can see the index 0 will always be occupied with some element and hence we cannot directly insert an element at index 0. To insert an element at index 0, we have to shift all elements present in the array by one place so that an empty space can be created at index 0 and our new element can be inserted there. So ,suppose if we have 'N' number of elements present in the array and we have to insert a new element , we have to make N number of shifts to create an empty space at index 0. So the average time Complexity to insert an element from the front side of an array is **O(N).**

Since , **O(N)>O(1)** , inserting an element from the rear end of an array is faster than inserting an element from the front side of an array and hence we always insert the element from the rear side of an array.

To make visualization easy , whenever we will talk about stack , we will visualize stack as a vertical array because in Real life whenever we talk about a stack of something,we usually talk about a vertical pile of some objects.

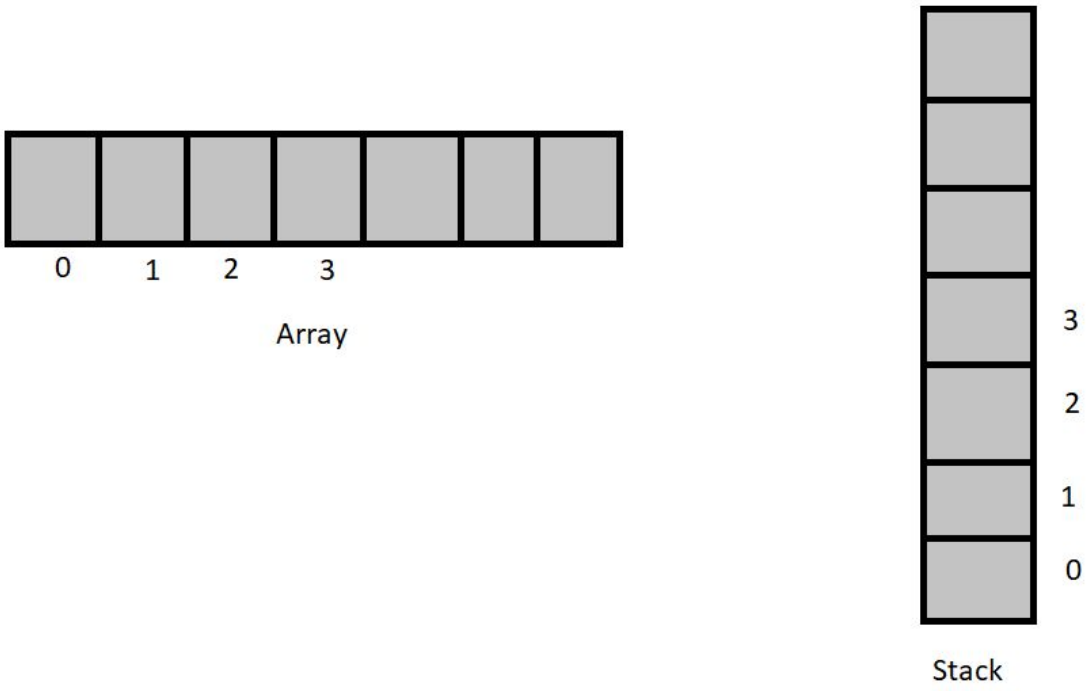


Fig. 1.2(b)

Visual Representation of Stack

As we discussed about an array , a size variable and a top pointer to implement a stack using array, but these three are of different data types. Size and top are integer variables and a pointer to an array. So , to combine data items of different datatypes used for same purpose there is a user defined datatype called structure . So the structure for stack is as follows:-

```
struct stack
{
int size; //for storing size of stack
int top;//top variable for keeping track of topmost element of stack
```

```
int *s;// a pointer pointing to an array we have created
}
```

Now , we have collected the basic knowledge and hence , we can create our program.

```
#include<iostream>
using namespace std;
struct stack
{
int size;
int top; // We will use it as pointer to top of the stack
int *s;
} st; // Here pointer is struct variable, you can see here how to implement structure in C

 void push(); // To push elements in stack
 int  pop();  // To Pop elements in stack
 void display();
 void peek();
 int main()
 {
 int c;
 st.top=-1; //Set pointer to -1
 cout<<"enter the size of stack\n";
 cin>>st.size;
 st.s=(int *)malloc(st.size* sizeof(int));
 while(1)  //While loop to keep program in loop
  {
```

```
23.  cout<<"\n Pointer is at : \t"<<st.top+1<<endl;
  //this line is to test, the position of pointer
24.  cout<<"\n Please enter your choice
  :"<<endl<<"1: Push"<<endl<<"2: Pop"<<endl<<"3:
  Peek"<<endl<<"4: Display"<<endl<<"5:
  Exit"<<endl;
25.  cin>>c;
26.     switch(c)
27.     {
28.     case 1:
29.         push();
30.         break;
31.     case 2:
32.         pop();
33.         break;
34.     case 3:
35.         peek();
36.         break;
37.     case 4:
38.         display();
39.         break;
40.     case 5:
41.         return 0;
42.     default:cout<<"invalid input\n";
43.     }
44.   }
45.  }
46.  void push()
47.  {
48.     int num;
```

```
49.     if(st.top==(st.size-1))
50.     {
51.         cout<<"\n Sorry stack overflow";
52.     }
53.     else
54.     {
55.     cout<<"\n Enter the number to push into
  stack";
56.     cin>>num;
57.     st.top+=1;
58.     st.s[st.top]= num;
59.     }
60. }
61.
62. int pop()
63. {
64.     int num;
65.     if(st.top==-1)
66.     {
67.     cout<<"\nstack is empty "<<endl;
68.     }
69.     else
70.     {
71.     num=st.top;
72.     cout<<"\n Poped number is : "<<st.s[num];
73.     st.top-=1;
74.     }
75.     return num;
76.
77. }
```

```
78.  void peek()
79.  {
80.                                          int pos;
81.                                          int x=-1;
82.
  cout<<"Enter the position you want to see\n";
83.                                          cin>>pos;
84.
  if(st.top-pos+1<0)
85.
  cout<<"Invalid Position\n";
86.                                          else
87.
  x=st.s[st.top-pos+1];
88.     cout<<x<<"\n";
89.  }
90.  void display()
91.  {
92.     if(st.top==-1)
93.     {
94.     cout<<"\n Stack is empty"<<endl;
95.     }
96.     else
97.     {  cout<<"stack entries--------\n";
98.     for(int i=st.top;i>=0;i--)
99.         {
100.        cout<<st.s[i]<<" "<<endl;
101.        }
102.    }
```

Stack using array https://ideone.com/4wlmFN

Balanced Brackets

In this section , we will see how stack can be used to solve an interesting programming problem quite easily called Balanced Brackets.

Description of the Problem:-

You are given a string containing a combination made up of 3 types of brackets { , } , [,] , (and).Now given a string made from the combination of these brackets ,we have to tell whether the string is balanced or not.

Example of Balanced String:- { ([ ]) }

Example of Unbalanced String:- { ( ]) }

Logic :-

We will create a stack and will start parsing strings from the left hand side. If we found any opening bracket, we will push that bracket into the stack. But if we found a closing bracket, we will look at the topmost bracket in the stack. If the topmost bracket matches the current bracket, we will pop that bracket from the stack and will move to the next bracket in the string (if any).But if the top bracket does not match the current bracket or if the stack is empty, then it is an **UNBALANCED String**. After the whole string is parsed, we look at the stack one final time, if the stack is empty, then we will finally print out that the given string is **BALANCED** but if there are some brackets left in the stack, then we print out that it is an unbalanced string.

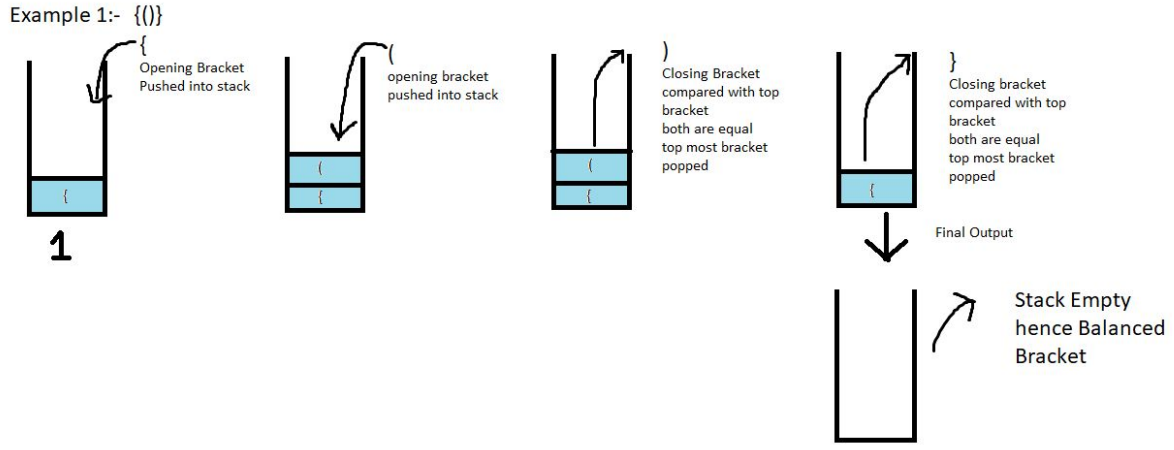


Fig. 1.2(c)

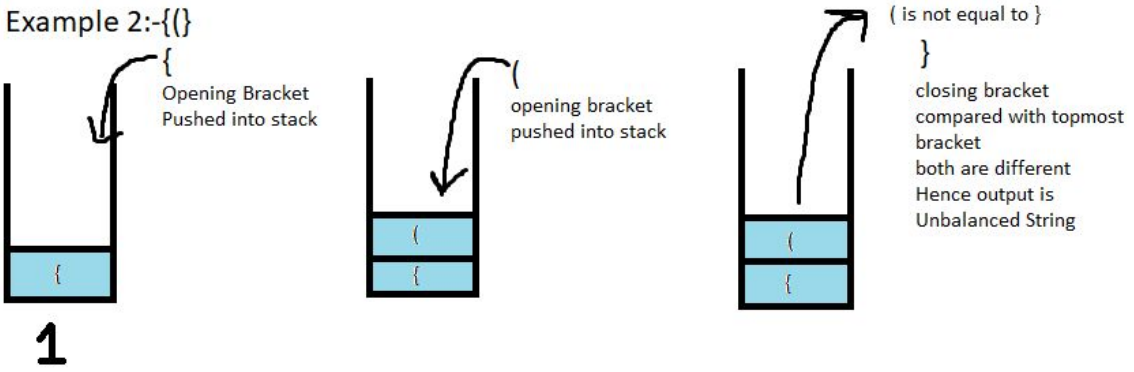


Fig. 1.2(d)

Now you are ready to write the program for this , Consider the following link for any help.

```
//Balanced Bracket problem...

#include <bits/stdc++.h>
#include <iostream>
using namespace std;

int main()
```

```
{
    string s;
    cin>>s;
    stack<char> st;
    int i,flag=0;
    for(i=0;i<s.length();i++)
    {
        if(s[i]=='('|| s[i]=='{' || s[i]=='[')
            st.push(s[i]);
        else
        {
            if(!st.empty())
            {
                if(s[i]==')'&&st.top()=='(' ||
s[i]=='}'&&st.top()=='{'|| s[i]==']'&&st.top()=='[')
                 st.pop();
             else
             {
                 flag=1;
                 break;
             }
            }
            else
            {
                flag=1;
                break;
            }
        }
    }
    if(st.empty()&&flag==0)
```

```
        cout<<"BALANCED BBRACKET";
    else
        cout<<"unbalanced bracket";
    return 0;

}
```

Balanced bracket:- https://sapphireengine.com/@/dpodt0

Part-3

INFIX , POSTFIX AND PREFIX CONVERSIONS

In this section we will implement the concept of stack for converting infix expression to postfix expression.
In c++ we already have stack containers in STL(standard template library)and its function are described below:-

- empty() → Returns whether the stack is empty
- size() → Returns the size of the stack
- top() → Returns a reference to the top most element of the stack
- push(g) → Adds the element ‘g’ at the top of the stack

- pop() → Deletes the top most element of the stack

ques=How to create a stack ???
ans= stack<char> st;
Using this we have created a stack st which can store character element in LIFO order.

Infix expression is an expression in which the operator is in the middle of operands, like **operand operator operand.**

Infix expression = x+y

Postfix expression is an expression in which the operator is after operands, like **operand operand operator.**

Postfix expression= xy+

One important question arises here is why do we need postfix expressions?

This is because the expressions written in postfix form are easily computed by the system as compared to infix notation as parenthesis are not required in postfix. Generally reading and editing by the user is done on infix notations as they are parentheses separated hence easily understandable for humans.

Infix to Postfix Conversion Algorithm:-

1. Print operands as they arrive.

2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.

3. If the incoming symbol is a left parenthesis, push it on the stack.

4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.

5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.

6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and

then push the incoming operator. If the association is right to left, push the incoming operator.

7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.

8. At the end of the expression, pop and print all operators on the stack

a - (*b* + *c* * *d*)/*e* to

ch	stack (bottom to top)	postfixExp
a		a
–	–	a
(	–(	a
b	–(	ab
+	–(+	ab
c	–(+	abc
*	–(+*	abc
d	–(+*	abcd
)	–(+	abcd*
	–(	abcd*+
	–	abcd*+
/	–/	abcd*+
e	–/	abcd*+e
		abcd*+e/–

Question for practice=
1.immediate-smaller-element :
https://practice.geeksforgeeks.org/problems/immediate-smaller-element/0
2.remove-repeated-digits-in-a-given-number :
https://practice.geeksforgeeks.org/problems/remove-repeated-digits-in-a-given-number/0
3.pairwise-consecutive-elements :
https://practice.geeksforgeeks.org/problems/pairwise-consecutive-elements/1
4.reverse-a-string-using-stack :
https://practice.geeksforgeeks.org/problems/reverse-a-string-using-stack/1
5.delete-middle-element-of-a-stack :
https://practice.geeksforgeeks.org/problems/delete-middle-element-of-a-stack/1