**Stack**

The ***Stack*** is a linear data structure, which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

* The LIFO order says that the element which is inserted at the last in the Stack will be the first one to be removed. In LIFO order, the insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.
* The FILO order says that the element which is inserted at the first in the Stack will be the last one to be removed. In FILO order, the insertion takes place at the rear end of the stack and deletion occurs at the front of the stack.

Mainly, the following three basic operations are performed in the stack:

* **Push:**Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
* **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they were pushed. If the stack is empty, then it is said to be an Underflow condition.
* **Peek or Top:** Returns the top element of the stack.
* **isEmpty:**Returns true if the stack is empty, else false.

**Time Complexities of operations on stack:** The operations push(), pop(), isEmpty() and peek() all take O(1) time. We do not run any loop in any of these operations.

**Implementation:** There are two ways to implement a stack.

* Using array
* Using linked list

**Pros:** The linked list implementation of stack can either grow or shrink according to the needs at runtime.  
**Cons:** Requires extra memory due to involvement of pointers.

**Applications of stack:**

* Stacks can be used to check for the balancing of paranthesis in an expression.
* Infix to Postfix/Prefix conversion.
* Redo-undo features at many places such as editors, photoshop, etc.
* Forward and backward feature in web browsers.

And Many More...

Stack in C++ STL

The C++ STL offers a built-in class named **stack** for implementing the stack data structure easily and efficiently. This class provides almost all functions needed to perform the standard stack operations like push(), pop(), peek(), remove() etc..

Some Basic functions of Stack class in C++:

* **empty()** – Returns whether the stack is empty.
* **size()** – Returns the size of the stack.
* **top()** – Returns a reference to the topmost element of the stack.
* **push(g)** – Adds the element ‘g’ at the top of the stack.
* **pop()** – Deletes the topmost element of the stack.

*All of the above functions work in O(1) time complexity.*

**Infix to postfix conversion using stack**

**Infix expression:** The expression of the form *a op b*. When an operator is in-between every pair of operands.  
  
**Postfix expression:** The expression of the form *a b op*. When an operator is followed for every pair of operands.  
  
**Why postfix representation of the expression?** The compiler scans the expression either from left to right or from right to left.  
  
Consider the below expression:

a op1 b op2 c op3 d  
  
If op1 = +, op2 = \*, op3 = +

The compiler first scans the expression to evaluate the expression b \* c, then again scan the expression to add a to it. The result is then added to d after another scan.  
  
The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.  
  
The corresponding expression in postfix form is:**abc\*+d+**. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.  
  
**Algorithm to Convert an Infix expression to Postfix:**

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,  
   * If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '(' ), push it.
   * Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop the stack and and output it until a ‘(‘ is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output.
8. Pop and output from the stack until it is not empty.

**Evaluating postfix expression using stack**

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have already discussed the conversion of infix to postfix expressions. In this post, the next step after that, that is evaluating a postfix expression is discussed.  
  
  
Following is the algorithm for evaluation of postfix expressions:

1. Create a stack to store operands (or values).
2. Scan the given expression and do following for every scanned element.  
   * If the element is a number, push it into the stack.
   * 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.
3. When the expression is ended, the number in the stack is the final answer.

**Example:** Let the given expression be "***2 3 1 \* + 9 -***". We will first scan all elements one by one.

1. Scan '2', it's a number, so push it to stack. Stack contains '2'
2. Scan '3', again a number, push it to stack, stack now contains '2 3' (from bottom to top)
3. Scan '1', again a number, push it to stack, stack now contains '2 3 1'
4. Scan '\*', it's an operator, pop two operands from the stack, apply the \* operator on operands, we get 3\*1 which results in 3. We push the result '3' to stack. Stack now becomes '2 3'.
5. Scan '+', it's an operator, pop two operands from the stack, apply the + operator on operands, we get 3 + 2 which results in 5. We push the result '5' to stack. Stack now becomes '5'.
6. Scan '9', it's a number, we push it to the stack. Stack now becomes '5 9'.
7. Scan '-', it's an operator, pop two operands from the stack, apply the - operator on operands, we get 5 - 9 which results in -4. We push the result '-4' to stack. Stack now becomes '-4'.
8. There are no more elements to scan, we return the top element from the stack (which is the only element left in the stack).

**Implementing two stack using one array**

The task is to create a data structure *twoStacks*that represents two stacks. Implementation of *twoStacks*should use only one array, i.e., both stacks should use the same array for storing elements. Following functions must be supported by *twoStacks*.

* push1(int x) --> pushes x to first stack.
* push2(int x) --> pushes x to second stack.
* pop1() --> pops an element from the first stack and return the popped element.
* pop2() --> pops an element from the second stack and return the popped element.

**Note**: Implementation of *twoStack*should be space efficient.  
  
  
**Method 1 (Divide the space in two halves)**: A simple way to implement two stacks is to divide the array into two halves and assign the half space to the first stack and the other half to the second stack, i.e., use arr[0] to arr[n/2] for stack1, and arr[(n/2) + 1] to arr[n-1] for stack2 where arr[] is the array to be used to implement two stacks and size of array be **n**.  
  
The problem with this method is an inefficient use of array space. A stack push operation may result in stack overflow even if there is space available in arr[]. For example, say the array size is 6 and we push 3 elements to stack1 and do not push anything to the second stack2. When we push the 4th element to stack1, there will be overflow even if we have space for 3 more elements in the array.  
  
**Method 2 (A space-efficient implementation)**: This method efficiently utilizes the available space. It doesn't cause an overflow if there is space available in arr[]. The idea is to start two stacks from two extreme corners of arr[]. The first stack, *stack1* starts from the leftmost element, the first element in *stack1* is pushed at index 0. The second stack, *stack2* starts from the rightmost corner, the first element in stack2 is pushed at index (n-1). Both stacks grow (or shrink) in opposite direction. To check for overflow, all we need to check is for space between top elements of both stacks.  
  
Below the space-efficient implementation(Method 2) of the above task:

### Array implementation of stack :

### Using array :

#include<iostream>

using namespace std;

struct MyStack

{

    int \*arr;

    int cap;

    int top;

    MyStack(int c)

    {

        cap=c;

        arr=new int [cap];

        top=-1;

    }

    void push(int x)

    {

        if(top==cap-1)

        {

            cout<<"Stack is Full"<<endl;

            return;

        }

        top++;

        arr[top]=x;

    }

    int pop()

    {

        if(top==-1)

        {

            cout<<"Stack is Empty"<<endl;

            return INT32\_MIN;

        }

        int res=arr[top];

        top--;

        return res;

    }

    int peak()

    {

        if(top==-1)

        {

            cout<<"Stack is Empty"<<endl;

            return INT32\_MIN;

        }

        return arr[top];

    }

    int size()

    {

        return (top+1);

    }

    bool isEmpty()

    {

        return top==-1;

    }

};

int main()

{

    MyStack s(5);

    s.push(5);

    s.push(10);

    s.push(20);

    cout<<s.pop()<<endl;

    cout<<s.size()<<endl;

    cout<<s.peak()<<endl;

    cout<<s.isEmpty()<<endl;

    return 0;

}

### OUTPUT :

### 20

### 2

### 10

### 0

### Array implementation of stack :

### Using vector :

#include<iostream>

#include<vector>

using namespace std;

struct MyStack

{

    vector<int> v;

    void push(int x)

    {

        v.push\_back(x);

    }

    int pop()

    {

        int res=v.back();

        v.pop\_back();

        return res;

    }

    int peek()

    {

        return v.back();

    }

    int size()

    {

        return v.size();

    }

    bool isEmpty()

    {

        return v.empty();

    }

};

int main()

{

    MyStack s;

    s.push(5);

    s.push(10);

    s.push(20);

    cout<<s.pop()<<endl;

    cout<<s.size()<<endl;

    cout<<s.peek()<<endl;

    cout<<s.isEmpty()<<endl;

    return 0;

}

### OUTPUT :

### 20

### 2

### 10

### 0

### Linked list implementation of stack

#include<iostream>

using namespace std;

struct Node

{

    int data;

    Node \*next;

    Node(int x)

    {

        data=x;

        next=NULL;

    }

};

struct MyStack

{

    Node \*head;

    int sz;

    MyStack()

    {

        head=NULL;

        sz=0;

    }

    void push(int x)

    {

        Node \*temp=new Node(x);

        temp->next=head;

        head=temp;

        sz++;

    }

    int pop()

    {

        if(head==NULL)

        {

            cout<<"Stack is Empty"<<endl;

            return INT32\_MAX;

        }

        int res=head->data;

        Node \*temp=head;

        head=head->next;

        delete(temp);

        sz--;

        return res;

    }

    int peak()

    {

        if(head==NULL)

        {

            cout<<"Stack is Empty"<<endl;

            return INT32\_MAX;

        }

        return head->data;

    }

    int size()

    {

        return sz;

    }

    bool isEmpty()

    {

        return head==NULL;

    }

};

int main()

{

    MyStack s;

    s.push(5);

    s.push(10);

    s.push(20);

    cout<<s.pop()<<endl;

    cout<<s.size()<<endl;

    cout<<s.peak()<<endl;

    cout<<s.isEmpty()<<endl;

    return 0;

}

### OUTPUT :

### 20

### 2

### 10

### 0

### Stack in C++ STL

#include<iostream>

#include<stack>

using namespace std;

int main()

{

    stack<int> s;

    s.push(10);

    s.push(20);

    s.push(30);

    cout<<s.size()<<endl;

    cout<<s.top()<<endl;

    s.pop();

    cout<<s.top()<<endl;

    s.push(5);

    cout<<s.top()<<endl;

    while (s.empty()==false)  //s.empty()!=true

    {

        cout<<s.top()<<endl;

        s.pop();

    }

     return 0;

}

### OUTPUT :

### 3

### 30

### 20

### 5

### 5

### 20

### 10

### Balance Parenthesis

//Given a string of parenthesis ({, }, (, ), [ and ]),

//we need to check if this string is balanced or not.

//time complexity O(n) and auxiliary space O(n)

#include<bits/stdc++.h>

using namespace std;

bool matching(char a, char b)

{

    return ((a=='(' && b==')') || (a=='[' && b==']')  || (a=='{' && b=='}'));

}

bool isBlanced(string str)

{

    stack<char> s;

    for(int i=0;i<str.length();i++)

    {

        if(str[i]=='{' || str[i]=='(' || str[i]=='[')

        {

            s.push(str[i]);

        }

        else{

            if(s.empty()==true)

                return false;

            else if(matching(s.top(),str[i])==false)

                return false;

            else

                s.pop();

        }

    }

    return (s.empty()==true);

}

int main()

{

    string str="{()}[]";

    if(isBlanced(str))

        cout<<"Balanced";

    else

        cout<<"Not Balanced";

    return 0;

}

### OUTPUT : Balanced

### Two Stack in Array

### Method 1(Divide the space in two halves)

// Method 1(Divide the space in two halves)

#include<iostream>

using namespace std;

struct TwoStack

{

    int \*arr;

    int cap;

    int top1,top2;

    TwoStack(int n)

    {

        cap=n;

        arr=new int[n];

        top1=n/2+1;

        top2=n/2;

    }

    void push1(int x)

    {

        if(top1>0){

            top1--;

            arr[top1]=x;

        }

        else{

            cout<<"stack Overflow"

                <<"By element :"<<x<<endl;

            return;

        }

    }

    void push2(int x)

    {

        if(top2<cap-1){

            top2++;

            arr[top2]=x;

        }

        else{

            cout<<"Stack Overflow"

                <<"By element :"<<x<<endl;

            return;

        }

    }

    int pop1()

    {

        if(top1<=cap/2)

        {

            int x=arr[top1];

            top1++;

            return x;

        }

        else{

            cout<<"Stack UnderFlow";

            exit(1);

        }

    }

    int pop2()

    {

        if(top2>=cap/2 +1)

        {

            int x=arr[top2];

            top2--;

            return x;

        }

        else{

            cout<<"Stack UnderFlow";

            exit(1);

        }

    }

};

int main()

{

    TwoStack ts(5);

    ts.push1(5);

    ts.push2(10);

    ts.push2(15);

    ts.push1(11);

    ts.push2(7);

    cout<<"Popped element from stack1 is: "<<ts.pop1()<<endl;

    ts.push2(40);

    cout<<"Popped element from stack2 is: "<<ts.pop2()<<endl;

    return 0;

}

### OUTPUT :

### Stack OverflowBy element :7

### Popped element from stack1 is: 11

### Stack OverflowBy element :40

### Popped element from stack2 is: 15

**Method 2(A space efficient implementation)**

// Method 2(A space efficient implementation)

#include <iostream>

using namespace std;

struct TwoStack

{

    int \*arr;

    int cap;

    int top1, top2;

    TwoStack(int n)

    {

        cap = n;

        arr = new int[n];

        top1 = -1;

        top2 = cap;

    }

    void push1(int x)

    {

        if (top1 < top2 - 1)

        {

            top1++;

            arr[top1] = x;

        }

        else

        {

            cout << "Stack Overflow";

            exit(1);

        }

    }

    void push2(int x)

    {

        if (top1 < top2 - 1)

        {

            top2--;

            arr[top2] = x;

        }

        else

        {

            cout << "Stack Overflow";

            exit(1);

        }

    }

    int pop1()

    {

        if (top1 >= 0)

        {

            int x = arr[top1];

            top1--;

            return x;

        }

        else

        {

           cout<<"Stack Underflow";

           exit(1);

        }

    }

    int pop2()

    {

        if(top2<cap){

            int x=arr[top2];

            top2++;

            return x;

        }

        else{

            cout<<"Stack Underflow";

            exit(1);

        }

    }

};

int main()

{

    TwoStack ts(5);

    ts.push1(5);

    ts.push2(10);

    ts.push2(15);

    ts.push1(11);

    ts.push2(7);

    cout<<"Popped element from stack1 is "<<ts.pop1();

    ts.push2(40);

    cout<<"\nPopped element from stack2 is "<<ts.pop2();

    return 0;

}

### OUTPUT :

### Popped element from stack1 is 11

### Popped element from stack2 is 40

### K stack in an array

#include<iostream>

using namespace std;

struct kStacks

{

    int \*arr;

    int \*top;

    int \*next;

    int cap, k;

    int freeTop;

    kStacks(int k1, int n)

    {

        k=k1;

        cap=n;

        arr=new int[cap];

        top=new int[k];

        next= new int[cap];

        for(int i=0;i<k;i++)

            top[i]=-1;

        freeTop=0;

        for(int i=0;i<cap-1;i++)

            next[i]=i+1;

        next[cap-1]=-1;

    }

    bool isFull(){

        return (freeTop==-1);

    }

    bool isEmpty(int sn){ return (top[sn]==-1);}

    void push(int x, int sn)

    {

        if(isFull()){

            cout<<"\nStack Overflow\n";

            return;

        }

        int i=freeTop;

        freeTop=next[i];

        next[i]=top[sn];

        top[sn]=i;

        arr[i]=x;

    }

    int pop(int sn)

    {

        if(isEmpty(sn))

        {

            cout<<"\nStack Underflow\n";

            return INT32\_MAX;

        }

        int i=top[sn];

        top[sn]=next[i];

        next[i]=freeTop;

        freeTop=i;

        return arr[i];

    }

};

int main()

{

    int k=3, n=10;

    kStacks ks(k,n);

    ks.push(15,2);

    ks.push(45,2);

    ks.push(17,1);

    ks.push(49,1);

    ks.push(39,1);

    ks.push(11,0);

    ks.push(9,0);

    ks.push(7,0);

    cout<<"Popped element from stack 2 is "<<ks.pop(2)<<endl;

    cout<<"Popped element from stack 1 is "<<ks.pop(1)<<endl;

    cout<<"Popped element from stack 0 is "<<ks.pop(0)<<endl;

    return 0;

}

### OUTPUT :

### Popped element from stack 2 is 45

### Popped element from stack 1 is 39

### Popped element from stack 0 is 7

### Stock span Problem

### Naive code :

// Stock span is defined as a number of consecutive days prior to the current day

// when the price of a stock was less than or equal to the price at current day

#include<iostream>

using namespace std;

void printSpan(int arr[], int n){

    for(int i=0;i<n;i++){

        int span=1;

        for(int j=i-1;j>=0 && arr[j]<=arr[i];j--)

            span++;

        cout<<span<<" ";

    }

}

int main()

{

    int arr[]={18,12,13,14,11,16};

    int n=6;

    printSpan(arr,n);

    return 0;

}

### OUTPUT :

### 1 1 2 3 1 5

### Efficient code for stock problem :

// Stock span is defined as a number of consecutive days prior to the current day

// when the price of a stock was less than or equal to the price at current day

//time complexity theta(n) and auxiliary space O(n)

#include<iostream>

#include<stack>

using namespace std;

void printSpan(int arr[],int n)

{

    stack<int>s;

    s.push(0);

    cout<<1<<" ";

    for(int i=1;i<n;i++){

        while(s.empty()==false && arr[s.top()]<=arr[i])

            s.pop();

        int span=s.empty()? i+1:i-s.top();

        cout<<span<<" ";

        s.push(i);

    }

}

int main()

{

    int arr[]={18,12,13,14,11,16};

    int n=6;

    printSpan(arr,n);

    return 0;

}

### OUTPUT :

### 1 1 2 3 1 5

### Previous Greater Element :

### Given an array of distinct integers, find the closest (positive wise) greater on left of every element. If there is no greater element on left, then print -1

### Naïve :

// Given an array of distinct integers, find the closest (positive wise)

// greater on left of every element. If there is no greater element on left,

//  then print -1

#include<iostream>

using namespace std;

void printPrevGreater(int arr[], int n)

{

    for(int i=0;i<n;i++){

        int j;

        for(j=i-1;j>=0;j--){

            if(arr[j]>arr[i]){

                cout<<arr[j]<<" ";

                break;

            }

        }

        if(j==-1)

            cout<<-1<<" ";

    }

}

int main()

{

    int arr[]={20,30,10,5,15};

    int n=5;

    printPrevGreater(arr,n);

    return 0;

}

### OUTPUT :

### -1 -1 30 10 30

### Efficient code for Previous Greater element:

// Given an array of distinct integers, find the closest (positive wise)

// greater on left of every element. If there is no greater element on left,

//  then print -1

#include<iostream>

#include<stack>

using namespace std;

void printPrevGreater(int arr[], int n)

{

    stack<int>s;

    s.push(arr[0]);

    for(int i=0;i<n;i++)

    {

        while (s.empty()==false && s.top()<=arr[i])

            s.pop();

        int pg=s.empty()?-1:s.top();

        cout<<pg<<" ";

        s.push(arr[i]);

    }

}

int main()

{

    int arr[]={20,30,10,5,15};

    int n=5;

    printPrevGreater(arr,n);

    return 0;

}

### OUTPUT :

### -1 -1 30 10 30

### Next Greater Element :

### Given an array of distinct integers, find the NextGreater(position-wise closest and on the right side) of every array elements.

### Naïve :

// Given an array of distinct integers, find the NextGreater

// (position-wise closest and on the right side) of every array elements.

#include<iostream>

using namespace std;

void nextGreater(int arr[], int n)

{

    for(int i=0;i<n;i++)

    {

        int j;

        for(j=i+1;j<n;j++)

        {

            if(arr[j]>arr[i])

            {

                cout<<arr[j]<<" ";

                break;

            }

        }

        if(j==n)

            cout<<-1<<" ";

    }

}

int main()

{

    int arr[]={5,15,10,8,6,12,9,18};

    int n=8;

    nextGreater(arr,n);

    return 0;

}

### OUTPUT :

### 15 18 12 12 12 18 18 -1

### Efficient code for Next Greater element:

// Given an array of distinct integers, find the NextGreater

// (position-wise closest and on the right side) of every array elements.

#include<bits/stdc++.h>

using namespace std;

vector <int> nextGreater(int arr[], int n)

{

    vector<int> v;

    stack<int> s;

    s.push(arr[n-1]);

    v.push\_back(-1);

    for(int i=n-2;i>=0;i--){

        while (s.empty()==false && s.top()<=arr[i])

            s.pop();

        int ng=s.empty()?-1:s.top();

        v.push\_back(ng);

        s.push(arr[i]);

    }

    reverse(v.begin(),v.end());

    return v;

}

int main()

{

    int arr[]={5,15,10,8,6,12,9,18};

    int n=8;

    for(int x: nextGreater(arr,n))

        cout<<x<<" ";

    return 0;

}

### OUTPUT :

### 15 18 12 12 12 18 18 -1

### Largest Rectangular area in a histogram part-1:

### Naïve :

#include<iostream>

using namespace std;

int getMaxArea(int arr[], int n)

{

    int res=0;

    for(int i=0;i<n;i++)

    {

        int curr=arr[i];

        for(int j=i-1;j>=0;j--)

        {

            if(arr[j]>=arr[i])

                curr+=arr[i];

            else

                break;

        }

        for(int j=i+1;j<n;j++)

        {

            if(arr[j]>=arr[i])

                curr+=arr[i];

            else break;

        }

        res=max(res,curr);

    }

    return res;

}

int main()

{

    int arr[]={6,2,5,4,1,5,6};

    int n=7;

    cout<<"Maximum Area : "<<getMaxArea(arr,n);

    return 0;

}

### OUTPUT:

### Maximum Area : 10

### Better code for Largest rectangular area in histogram part-1 :

#include <bits/stdc++.h>

using namespace std;

int getMaxArea(int arr[],int n){

    int res=0;

    int ps[n],ns[n];

    stack <int> s;

    s.push(0);

    for(int i=0;i<n;i++){

        while(s.empty()==false && arr[s.top()]>=arr[i])

            s.pop();

        int pse=s.empty()?-1:s.top();

        ps[i]=pse;

        s.push(i);

    }

    while(s.empty()==false){

        s.pop();

    }

    s.push(n-1);

    for(int i=n-1;i>0;i--){

        while(s.empty()==false && arr[s.top()]>=arr[i])

            s.pop();

        int nse=s.empty()?n:s.top();

        ns[i]=nse;

        s.push(i);

    }

    for(int i=0;i<n;i++){

        int curr=arr[i];

        curr+=(i-ps[i]-1)\*arr[i];

        curr+=(ns[i]-i-1)\*arr[i];

        res=max(res,curr);

    }

    return res;

}

int main()

{

    int arr[]={6,2,5,4,1,5,6};

    int n=7;

    cout<<"Maximum Area: "<<getMaxArea(arr,n);

    return 0;

}

### OUTPUT :

### Maximum Area: 38532528

### Efficient code for Largest rectangular area in histogram part-2 :

#include<iostream>

#include<stack>

using namespace std;

int getMaxArea(int arr[], int n)

{

    stack<int> s;

    int res=0;

    int tp;

    int curr;

    for(int i=0;i<n;i++)

    {

        while(s.empty()==false && arr[s.top()]>=arr[i]){

            tp=s.top();

            s.pop();

            curr=arr[tp]\*(s.empty()? i:i-s.top()-1);

            res=max(res,curr);

        }

        s.push(i);

    }

    while (s.empty()==false)

    {

        tp=s.top();

        s.pop();

        curr=arr[tp]\*(s.empty()? n:n-s.top()-1);

        res=max(res,curr);

    }

    return res;

}

int main()

{

    int arr[]={6,2,5,4,1,5,6};

    int n=7;

    cout<<"Maximum Area: "<<getMaxArea(arr,n);

    return 0;

}

### OUTPUT :

### Maximum Area: 10