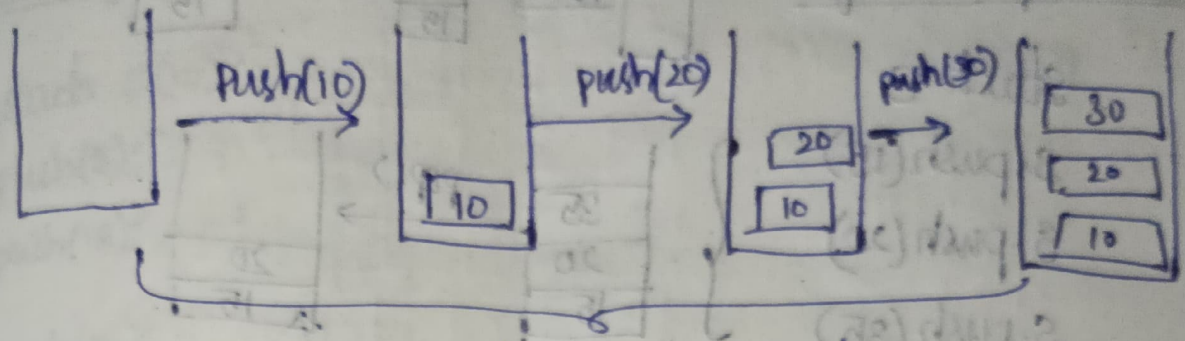
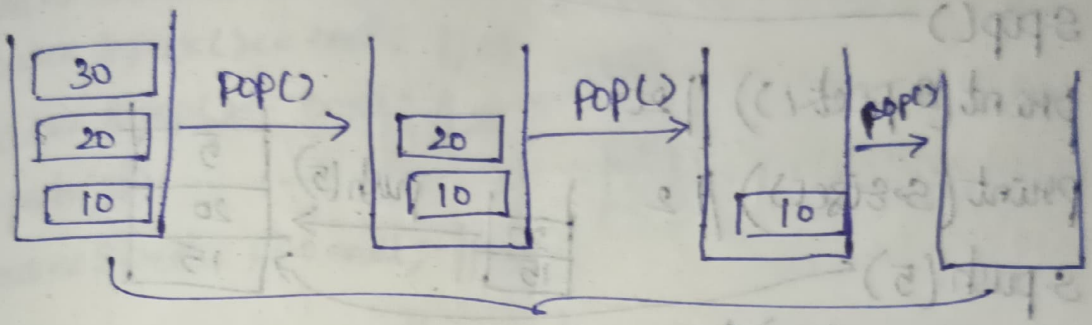
Stack Data Structure



push(x) Examples



pop() Examples.

Stack operations: (LIFO) principal

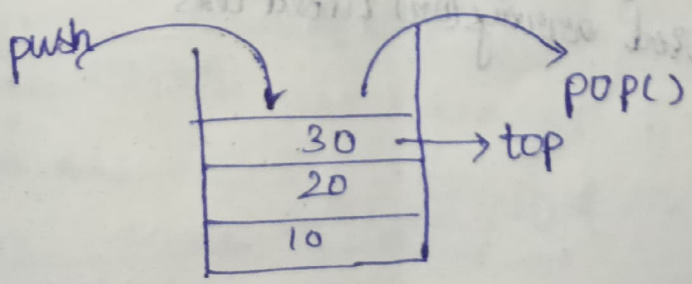
isEmpty(): Returns true if stack is empty, else false.

push(x): Inserts an item to the top of the stack

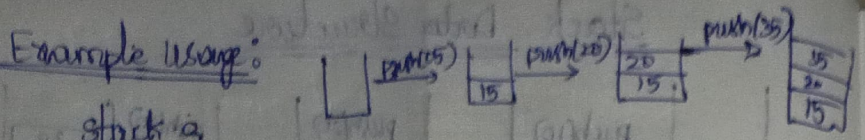
pop(): Remove an item from the top.

peek(): Returns the top item.

size(): Returns the size of stack.

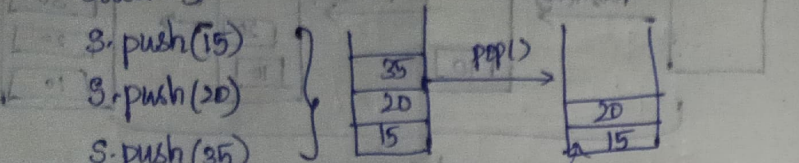


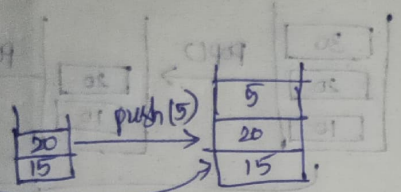
Example Usage:



Stack S

S.push(15)
S.push(20)
S.push(35)
print(S.peek()) // 35
S.pop()
print(S.peek()) // 20
print(S.size()) // 2
S.push(5)
print(S.peek()) // 5
print(S.isEmpty()) => // false





very common

Underflow: when pop() or peek() called on empty stack

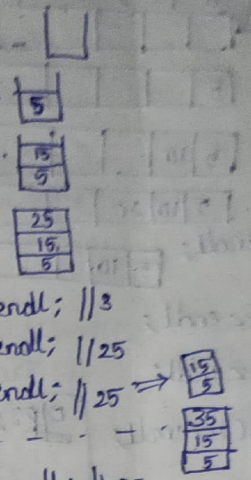
Overflow: when push called on a full stack

not very common by using dynamically sized arrays (or) Linked lists

Array Implementation of stack in C++:

```
Mystack S(10);
S.push(5);
S.push(15);
S.push(25);

cout << S.size() << endl; // 3
cout << S.peek() << endl; // 25
cout << S.pop() << endl; // 25
S.push(35);
cout << S.peek() << endl; // 35
cout << S.isEmpty() << endl; --- // 0
```

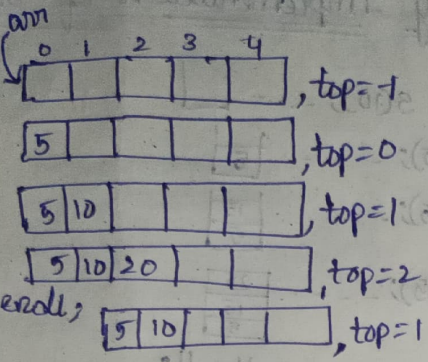



```
struct Mystack
{ int * arr;
  int cap;
  int top;
  Mystack(int c)
  { cap = c;
    arr = new int[cap];
    top = -1;
  }
  void push(int x)
  { if (top == cap-1) { --- }
    top++;
    arr[top] = x;
  }
  int pop()
  { if (top == -1) { --- }
    int res = arr[top];
    top--;
    return res;
  }
  int peek()
  { if (top == -1) { --- }
    return arr[top];
  }
  int size()
  { return (top+1);
  }
  bool isEmpty()
  { return (top == -1); } }; // End of structure mystack.
```

problems handled.

return INT_MAX (or) INT_MIN

```
int main()
{
    MyStack S(5);
    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;
}
```



O/P: 20
2
10
0

```
int main()
{
    MyStack S(5);
    cout << S.pop() << endl;
    cout << S.peek() << endl;
    S.push(10);
    S.push(20);
    S.push(30);
    S.push(40);
    S.push(50);
    S.push(60);
    return 0;
}
```

pop() and peek() on empty stack;

→ Adding 6 items to a stack of capacity 5.

problems with this code:

① Does not handle overflow and underflow.

② We need to provide capacity initially. No dynamic resizing.

Dynamically resizable stack:

```
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 size()
    {
        return v.size();
    }
    bool isEmpty()
    {
        return v.empty();
    }
    int peek()
    {
        return v.back();
    }
};// End of structure
```

T.C = O(1)

push & pop
↓
amortized → O(1)

Linked List implementation of stack in C++

```
Struct Node
{
  int data;
  Node *next;
  Node (int d)
  {
    data=d;
    next = NULL;
  }
};
Struct mystack
{
  Node *head;
  int ss;
  mystack()
  {
    head=NULL;
    ss=0;
  }
  Void push (int x)
  {
    Node *temp= new Node (x);
    temp->next = head;
    head = temp;
    ss++;
  }
  int pop()
  {
    if (head==NULL)
      return INT_MAX;
    int res = head->data;
    node *temp=head;
    head =head->next;
    delete temp;
    ss--;
    return res;
}
int size()
{ return ss;
}
bool isEmpty
{ return (head == NULL);
}
int peek()
{
  if (head == NULL)
    return INT_MAX;
  return head->data;
}
} // End of structure.
```

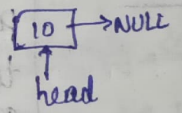
Worst case
T.C = O(1)

⇒ mystack s;

ss = 0 head = NULL

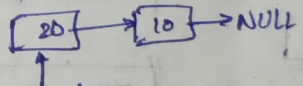
s.push(10)

ss = 1



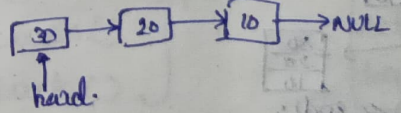
s.push(20)

ss = 2



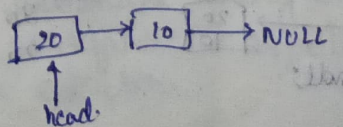
s.push(30)

ss = 3



s.pop()

ss = 2



Applications of stack

① function calls
fun1()
↳ fun2()
↳ fun3()

② Balanced paranthesis
[(a+b) * {c+(d-e)}]

③ Reversing Items

④ Infix to postfix / prefix
a+b → ab+
→ +ab
to get rid of order of evaluation by precendence associativity etc

⑤ Evaluation of postfix / prefix

⑥ stack span problem & its variants

⑦ undo / Redo (ctrl) forward / backward

---

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;
    return 0;
}
```

push()
pop()
top()
size()
empty()

T.C = O(1)

o/p: 3
30
20

```
int main()
{
    stack <int> s;
    s.push(10);
    s.push(20);
    s.push(30);
    while (s.empty() == false)
    {
        cout << s.top() << " ";
        s.pop();
    }
}
```

o/p: 30 20 10

push()
pop()
top()
size()
empty()
→ O(1)

Time Complexity & Internal working

stack can be implemented on any underlying container that provides following operations

- back()
- size()
- empty()
- push_back()
- pop_back()

provides ⇒ list, vector, deque

stack implemented using ⇒ by default deque

stack is implemented using other containers & work as interface it is also called as container adaptor.

Check for Balanced parenthesis

Six possible characters on I/p string:
(,), {, }, [and]

I/p: str = "([])"
O/p: yes

I/p: str = "(())"
O/p: NO

I/p: str = "([)]"
O/p: NO

I/p: str = "{}([()])"
O/p: yes

I/p: str = "(()))"
O/p: NO

bracket has opened latest has to be closed first
↓
LIFO

for opening bracket push into stack, for closing bracket compare with top of stack. If matched, pop() the top else return NO.

HINT: we are going to use a stack here, because of LIFO nature.

```
bool matching (char a, char b)
{
    return ((a == '(' && b == ')') ||
            (a == '{' && b == '}') ||
            (a == '[' && b == ']'));
}

bool isBalanced (String &str)
{
    stack < char > s;
    for (char x : str)
    {
        if (x == '(' || x == '{' || x == '[')
            s.push (x);
        else
        {
            if (s.empty() == true)
                return false;
            if (matching (s.top(), x) == false)
                return false;
            else
                s.pop();
        }
    }
    return (s.empty() == true);
}
```

T.C = O(n) (|str|)
A.S = O(n)

str = "[()]"

s

(I) [(II) [((III) [(IV)

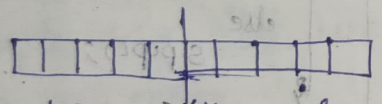
Implementation of two stacks in array

```
class TwoStacks
{ int *arr[];
  int cap;
  - - - -

public:
  TwoStacks(int n)
  { arr = new int[n];
    cap = n;
    - - - -
    - - - -
  }
  void push1(int x) { - - - }
  void push2(int x) { - - - }
  int pop1() { - - - }
  int pop2() { - - - }
  int size1() { - - - }
  int size2() { - - - }
}
```

T.C = O(1)

Naive solution:

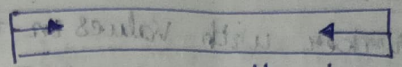


=> We divide the array from middle, use first half for stack 1 and second half for stack 2

=> Inefficient use of space

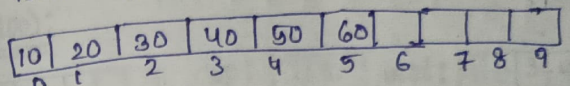
If we add 3 items to stack 1 and no items to stack 2, then we cannot add any more items to stack 1, even if we have space in the array.

Idea for the efficient solution:



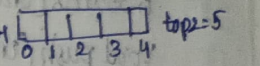
=> Begin both stacks from the two corners of the array.

=> Now we can insert items in any stack as long as we have space

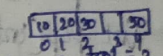


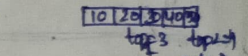
top1 = 5 , top2 = 10

```
class twoStacks
{ int *arr;
  int cap, top1, top2;
public:
  twoStacks(int n)
  { cap = n;
    top1 = -1;
    top2 = n;
    arr = new int[n];
  }
  bool push1(int x)
  { if (top1 < (top2-1))
    { top1++;
      arr[top1] = x;
      return true;
    }
    return false;
  }
  bool push2(int x)
  { if (top1 < top2-1)
    { top2--;
      arr[top2] = x;
      return true;
    }
    return false;
  }
```




```
int pop1()
{ if (top1 >= 0)
  { int x = arr[top1];
    top1--;
    return x;
  }
  else
  { cout << "stack underflow";
    exit(1); } return -1;
}
int pop2()
{ if (top2 < cap)
  { int x = arr[top2];
    top2++;
    return x;
  }
  else
  { cout << "stack underflow";
    exit(1); } return -1;
}
int size1()
{ return (top1+1); }
int size2()
{ return (cap-top2); }
}; // End of class.
```





Implement k stacks in an Array:

sn : stack Number with values in the below range [0, 1, 2, ... k-1]

```
Struct kStacks
{ int *arr;
  int k, cap;
  kStacks(int n, int k) {...}
  void push(int sn, int x) {...}
  int pop(int sn) {...}
  bool isEmpty() {...}
};
```

T.C = O(1)

```
Struct kStacks
{ int *arr, *top, *next;
  int k, free_top, cap;
  kStacks(int k1, int n)
  { free_top=0;
    cap=n;
    k=k1;
    arr=new int[n];
    top=new int[k];
    fill(top, top+k, -1);
    for(int i=0; i<cap-1; i++)
      next[i]=i+1;
    next[cap-1]=-1;
  }
  bool isEmpty(int sn)
  { return (top[sn]==-1);
  }
```

```
void push(int sn, int x)
{ if(free_top==-1) // overflow
    return;
  else
  { int i=free_top;
    free_top=next[i];
    next[i]=top[sn];
    top[sn]=i;
    arr[i]=x;
  }
}
int pop(int sn)
{ if(top[sn]==-1) // underflow;
    return INT_MAX;
  int i=top[sn];
  top[sn]=next[i];
  next[i]=free_top;
  free_top=i;
  return arr[i];
}
};
```

n=5
k=2

arr: [| | | |], top: [-1 | -1], next: [1 | 2 | 3 | 4 | -1] (0 1 2 3 4), free_top=0

push(1,100):

arr: [100 | | | |], top: [-1 | 0], next: [-1 | 2 | 3 | 4 | -1], free_top=1

push(1,200):

arr: [100 | 200 | | |], top: [-1 | 1], next: [-1 | 0 | 3 | 4 | -1], free_top=2

pop(1):

arr: [100 | | | |], top: [-1 | 0], next: [-1 | 2 | 3 | 4 | -1], free_top=1

Stock span problem

No of consecutive days on the left side including the current day, which have values smaller or equal

I/p: arr[] = [13, 15, 12, 14, 16, 8, 6, 4, 10, 30]
O/p: 1 2 1 2 5 1 1 1 4 10

I/p: arr[] = [10, 20, 30, 40]
O/p: 1 2 3 4

I/p: arr[] = [40, 30, 20, 10]
O/p: 1 1 1 1

I/p: arr[] = [30, 20, 25, 28, 27, 29]
O/p: 1 1 2 3 1 2

Naive solution:

```
for (int i=0; i<n; i++)
{ int span=1;
  for (int j=i-1; j>=0 && arr[j]<=arr[i]; j--)
    span++;
  print(span);
}
```

T.C = $O(n^2)$
A.S = $O(1)$

arr[] = [18, 12, 13, 14, 11, 16]

i=0 : span: 1
i=1 : span: 1
i=2 : span: 2
i=3 : span: 3
i=4 : span: 1
i=5 : span: 5

Efficient solution

Span = (Index of current element) − (Index of closest greater element on left side)

= (Index of current element) + 1 — If there is a greater element on left side

otherwise

[60, 10, 20, 40, 35, 30, 50, 70, 65]
index ⇒ 0 1 2 3 4 5 6 7 8
1 (1-0) (2-0) (3-0) (4-3) (5-4) (6-0) (7-1) (8-~~0~~)

$x_0, x_1, x_2, x_3 \dots x_i \mid x_{i+1} \dots x_{n-1}$

[60, 10, 20, 40, 35 | 30, 50, 70, 65]
x_i | x_{i+1}

greater element

$x_{i+1} < 35$ (previous G.E = 35)

$(x_{i+1} > 35) \& (x_{i+1} < 40)$ (" = 40)

$(x_{i+1} > 40) \& (x_{i+1} < 60)$ (" = 60)

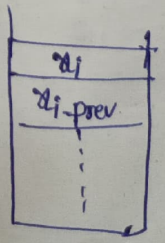
$(x_{i+1} > 60)$ (" = arr[i+1])

So we need to store ⇒ ~~35, 40, 60~~

60 40 35 (order) } Stack

previous greater ← previous greater

LIFO



```
Void printSpan(int arr[], int n)
{
    stack S; // create a stack
    S.push(0); // process First Item
    print(1);

    for (int i=1; i<n; i++)
    {
        while (S.isEmpty() == false && arr[S.top()] <= arr[i])
        {
            S.pop();
        }
        Span = S.isEmpty() ? i+1 : i-S.top();
        print(span);
        S.push(i);
    }
}
```

// process Remaining Items.

push, pop O(1)

T.C = O(n)
A.S = O(n)

arr[] = [60, 10, 20, 15, 35, 50]
        0   1   2   3   4   5

Before Loop: print(1)

Loop:

i=1: print(2)

i=2: print(2)

i=3: print(1)

i=4: print(4)

i=5: print(5)

previous Greater Element

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

I/p: arr[] = [15, 10, 18, 12, 4, 6, 2, 8]
O/p: -1 15 -1 18 12 12 6 12

I/p: arr[] = [8, 10, 12]
O/p: -1 -1 -1

I/p: arr[] = [12, 10, 8]
O/p: -1 12 10

Naive Solution:

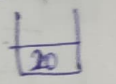
```
Void printPrevGreater(int arr[], int n)
{
    for (int i=0; i<n; i++)
    {
        int j;
        for (j=i-1; j>=0; j--)
        {
            if (arr[j] > arr[i])
            {
                cout << arr[j] << " ";
                break;
            }
        }
        if (j==-1)
            cout << -1 << " ";
    }
}
```

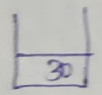
T.C = O(n²)

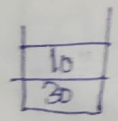
A.S = O(n)
T.C = O(n)

```
Void printPrevGreater(int arr[], int n)
{
    stack<int> s;
    s.push(arr[0]); cout << -1 << " ";
    for (int i=1; 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]);
    }
}
```

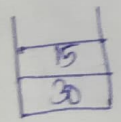
I/p: [20, 30, 10, 5, 15]
O/p: -1 -1 30 10 30

before:  print(-1)

i=1:  print(-1)

i=2:  print(30)

i=3:  print(10)

i=4:  print(30)

Next Greater Element:

Given an array of integers, find Next greater (position-wise closest and on right side) for every array element

I/p: arr[] = [5, 15, 10, 8, 6, 12, 9, 18]
O/p: 15 18 12 12 12 18 18 -1

I/p: arr[] = [10, 15, 20, 25]
O/p: 15 20 25 -1

I/p: arr[] = [25, 20, 15, 10]
O/p: -1 -1 -1 -1

Naive solution:

```
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 << " ";
    }
}
```

T.C = O(n²)

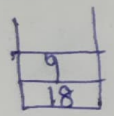
```
Vector<int> nextGreater(int arr[], int n)
{
    Stack<int> S; Vector<int> V;
    S.push(arr[n-1]); //cout<<"-1"<<" ";
    for(int i=n-2; i>=0; i--)
    {
        while(S.empty()==false && S.top()<=arr[i])
            S.pop();
        int ng = S.empty()?-1:S.top();
        //cout<<ng<<" ";
        S.push(arr[i]);
        V.push_back(ng);
    }
    reverse(V.begin(), V.end());
    return V;
}
```

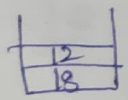
T.C = O(n)
A.S = O(n)

// prints output in reverse order" → to get correct O/P reverse the ans

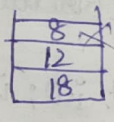
Ip: 0 1 2 3 4 5 6 7
[5, 15, 10, 8, 6, 12, 9, 18]
15 18 12 12 12 18 18 -1

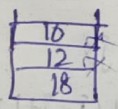
i=7:  print(-1)

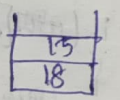
i=6:  print(18)

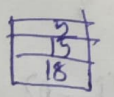
i=5:  print(18)

i=4:  print(12)

i=3:  print(12)

i=2:  print(12)

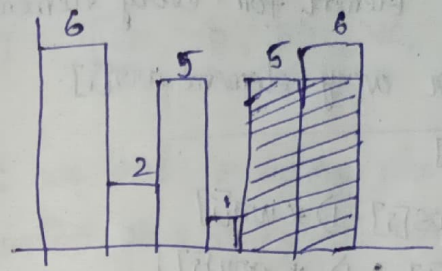
i=1:  print(18)

i=0:  print(15)

Largest Rectangle area in a Histogram

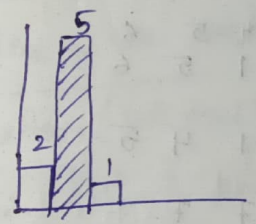
Ip: arr[] = [6, 2, 5, 4, 1, 5, 6]

Op: 10



Ip: arr[] = [2, -5, 1]

Op: 5



T.C = O(n²) A.S = O(1)

naive: idea: for every element check both on LHS & RHS for >= elements

```
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 breaks;
        }
        for(int j=i+1; j<n; j++)
        {
            if(arr[j]>=arr[i]) curr+=arr[i];
            else breaks;
        }
        res = max(res, curr);
    }
}
```

0 1 2 3 4 5 6
[6, 2, 5, 4, 1, 5, 6]

i=0: curr=6, res=6
i=1: curr=8, res=8
i=2: curr=5,
i=3: curr=8,
i=4: curr=7,
i=5: curr=10, res=10
i=6: curr=6,

Better Solution (O(n) Time)

① Initialize: res = 0

② find previous Smaller Element for every element ⇒ TC=O(n) AS=O(n)

③ find next Smaller Element for every element ⇒ TC=O(n) AS=O(n)

④ Do following for every element arr[i].

```
curr = arr[i]
curr += (i - ps[i] - 1) * arr[i]
curr += (ns[i] - i - 1) * arr[i]
res = max(res, curr)
```

⑤ return res.

	0	1	2	3	4	5	6
arr[] =	6	2	5	4	1	5	6
ps[] =	-1	-1	1	1	-1	4	5
ns[] =	1	4	3	4	7	7	7

i=0 : curr = 6 + 0*6 + 0*6

i=1 : curr = 2 + 1*2 + 2*2

i=2 : curr = 5 + 0*5 + 0*5

i=3 : curr = 4 + 1*4 + 0*4

Efficient solution for the Largest Area in a Histogram

arr[] = [60, 20, 50, 40, 10, 50, 60]
0 1 2 3 4 5 6

Idea: we are computing the largest area in Histogram with current item being the smallest bar,

when we pop the item from the stack, the element just below the stack ⇒ gives the previous smaller element & element ⇒ i.e., current element of the stack gives the index of the next smaller element.

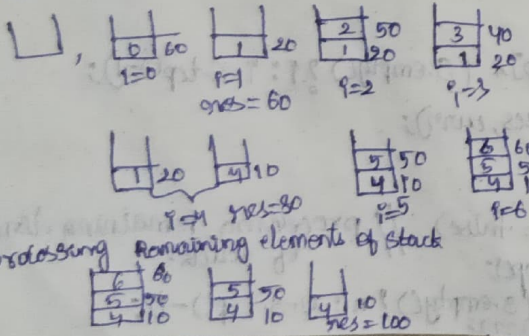
```
int getMaxArea(int arr[], int n)
{ stack S;  // Empty stack.
  int res=0;  int tp;  int curr;
  for(int i=0; i<n; i++)
  { while (!S.empty() && arr[st.top()] >= arr[i])
    { tp = st.pop();
      curr = arr[tp]
      tp = S.top();
      S.pop();
      curr = arr[tp] * (S.empty() ? i : i - S.top() - 1);
      res = max(res, curr);
    }
    S.push(i);
  } // End of for loop
  while (S.empty() == false)  // processing remaining elements of stack.
  { tp = S.top(); S.pop();
    curr = arr[tp] * (S.empty() ? n : n - S.top() - 1);
    res = max(res, curr);
  }
```

```
return res;
}
```

```
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) // processing remaining elements of stack
    {
        tp=s.top();
        s.pop();
        curr=arr[tp]*(s.empty()?n : n-s.top()-1);
        res=max(res,curr);
    }
    return res;
}
```

T.C = O(n)
A.S = O(n)

arr = [60, 20, 50, 40, 10, 50, 60]
0 1 2 3 4 5 6



Largest Rectangle with all 1s

I/p: mat[][] = [[0, 1, 1, 0],
[1, 1, 1, 1],
[1, 1, 1, 1],
[1, 1, 0, 0]]

O/p: 8

I/p: mat[][] = [[0, 1, 1],
[1, 1, 1],
[0, 1, 1]]

O/p: 6

I/p: mat[][] = [[0, 0],
[0, 0]]

O/p: 0

I/p: mat[][] = [[0, 0],
[0, 1]]

O/p: 1

I/p: mat[][] = [[1, 1, 1]]

O/p: 3

Naive solution:
⇒ Consider every cell as a start point O(R×C)
⇒ Consider all sizes of rectangle with current cell as a starting point. O(R×C)
⇒ for the current rectangle, check if it has all 1s
If yes, then update the res, if the current size is more. O(R×C)

Time: O(R³ × C³)

Effecient solution (Idea)

T.C = $\theta(R \times C)$
A.S = $O(C)$

Run a loop from 0 to R-1
⇒ update the histogram for the current row.
⇒ find the largest area in the histogram and update the result if required.

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

[0, 1, 1, 0], res = 2
[1, 2, 2, 1], res = 4
[2, 3, 3, 2], res = 8
[3, 4, 0, 0], "

```
int maxRectangle(int mat[R][C])
{ int res = getMaxArea(mat[0], C);
  for (int i=1; i<R; i++)
  { for (int j=0; j<C; j++)
    { if (mat[i][j] == 1)
        mat[i][j] += mat[i-1][j];
    }
    res = max(res, getMaxArea(mat[i], C));
  }
  return res;
}
```

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$ res = 2

i=1: $\begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$ res = 4

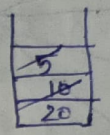
i=2: $\begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 1 & 1 & 1 & 3 & 3 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix}$ res = 6

i=3: $\begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 2 & 2 \\ 1 & 1 & 1 & 3 & 3 \\ 0 & 2 & 2 & 4 & 4 \end{bmatrix}$ res = 8

Design a stack that supports getMin()

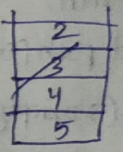
I/p: push(20), push(10), getmin(), push(5), getMin(), pop(), getmin(), pop(), getMin()

o/p: 10 5 10 20



I/p: push(5), push(4), push(3), getmin(), pop(), getmin(), push(2), getMin()

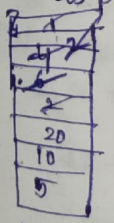
o/p: 3 4 2



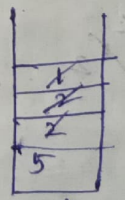

```
Stack ms, as;
Void push(int x)
{ if (ms.empty())
  { ms.push(x);
    as.push(x);
    return;
  }
  ms.push(x);
  if (as.top() >= ms.top())
    as.push(x);
}
```

```
Void pop()
{ if (as.top() == ms.top())
    as.pop();
  ms.pop();
}
int top()
{ return ms.top();
}
int getMin()
{ return as.top();
}
```

T.C = O(1)
A.S = O(n)



main Stack (ms)

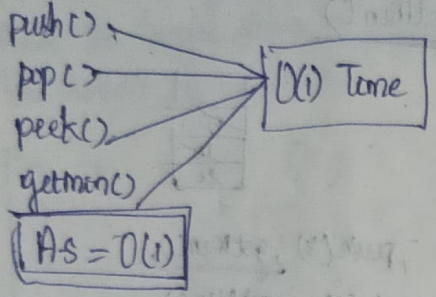


Auxiliary Stack (as)

push(5)	pop()
push(10)	pop()
push(20)	push(2)
push(2)	pop()
push(6)	push(1)
push(4)	pop()
	pop()

Design a stack with getMin() in O(1) space

First method (Assuming all elements positive)



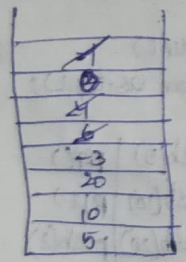

```
Stack S;
int min;
Void push(int x)
{
  if(S.isEmpty())
  {
    min=t;
    S.push(x);
  }
  else if(x<=min)
  { S.push(x-min);
    min=x;
  }
  else
  S.push(x)
}
```

T.C = O(1)
A.S = O(1)

```
int pop()
{t=S.pop();
 if(t<=0) S.pop()
 { res=min;
   min=min-t;
   return res;
 }
 else
 { return t;
 }
```

```
int peek()
{
 t=S.peek();
 return (t<=0)?min:t;
}
int getmin()
 return min
```

push(5)
push(10)
push(20)
push(2)
push(6)
push(4)
pop()//4
pop()//6
push(2)
pop()//2
push(1)
pop()
pop()



min = 5 / 7 / 2

second method (Handles Negatives)

```
Void push(int x)
{ if(S.isEmpty())
  { min=t;
    S.push(x);
  }
  else if(x<=min)
  { S.push(2*x-min);
    min=x;
  }
  else
  S.push(x);
}
```

```
int pop()
{ t=S.pop();
  if(t<=min)
  { res=min;
    min=2*min-t;
    return res;
  }
  else
  return t;
}
```

```
int peek()
{
 t=S.peek();
 return (t<=min)?min:t;
}
```

```
int getmin()
{ return min;
}
```

t = 2*min - prevmin
prev_min = t - 2*min

How does this work?

we push 2*x-min only when x≤min
So 2*x-min is always going to be less than or equal to x,
& x is going to be our new min.

2x-min = x + (x-min) ⇒ <0

2x-min = x

x ≤ min ✓

Infix, postfix and prefix Introduction

Infix: x+y

postfix: xy+

prefix: +xy

Advantages of prefix & postfix:

* Do not require, paranthesis, precedence rules & associativity rules
* can be evaluated by writing a program that traverses the given expression exactly once

---

⇒ If there are two operations of diff precedence then we evaluate or we put paranthesis around the higher precedence operator first then lower - - - -

⇒ If there are two operations of same precedence in an expression, then we follow associativity.

If associativity is [L→R] [R→L] then we evaluate (Left side first)

---

^ ⇒ exponent

operators	associativity
^	Right to left
*, /	Left to Right
+, -	Left to Right

precedence ↑

precedence & Associativity

10 + 20 * 2

precedence: (10 + (20 * 2)) ⇒ 10 + 40 = 50

Associativity: (L-R)

6 10 + 2 - 3

((10+2)-3) ⇒ 12-3 = 9

Associativity (R-L): 2^1^2 ⇒ (2^(1^2)) = 2^1 = 2

prefix and postfix

Infix	prefix	postfix
x+y*z	+x*yz	xyz*+
(x+y)*z	*+xyz	xy+z*

Steps for postfix conversion:

x+y*z ⇒ (x+(y*z))
⇒ (x+(yz*))
⇒ xyz*+

(x+y)*z ⇒ ((x+y)*z)
⇒ ((xy+)*z)
⇒ xy+z*

Infix to postfix:

operators	Associativity
^	R-L
* /	L-R
+ -	L-R

precedence ↑

⇒ If precedence of operators is different, then we follow precedence rules

⇒ If precedence of operators is same, then we follow associativity rules

Ip: infix = "a+b*c" Op: postfix = "abc*+"

Ip: infix = "(a+b)*c" Op: postfix = "ab+c*"

Ip: infix = "a^b^c" Op: postfix = "abc^^"

Ip: infix: "(a+b)*(c+d)"

postfix: "ab+cd+*"

Naive approach:

make fully parenthesized →

Infix = a+b*c ⇒ (a+(b*c)) ⇒ (a+(bc*)) ⇒ abc*+

Infix = (a+b)*c ⇒ ((a+b)*c) ⇒ ((ab+)*c) ⇒ ab+c*

Infix = a^b^c ⇒ (a^(b^c)) ⇒ (a^(bc^)) ⇒ abc^^

Infix = (a+b)*(c+d) ⇒ ((a+b)*(c+d)) ⇒ ((ab+)*(cd+)) ⇒ ab+cd+*

a+b*(c-d) ⇒ (a+(b*(c-d)))

⇒ (a+(b*(cd-)))

⇒ (a+(bcd-*))

⇒ abcd-*+

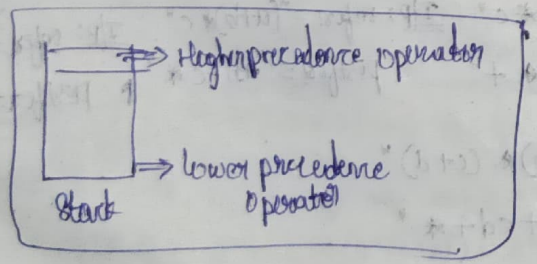
a+b*c/d+e ⇒ ((a+((b*c)/d))+e)

⇒ ((a+((bc*)/d))+e)

⇒ ((a+(bc*d/))+e)

⇒ ((abc*d/)+e)

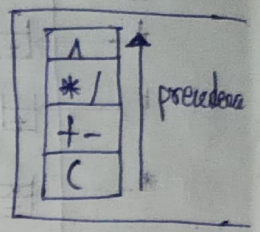
⇒ abc*d/+e+



Infix to postfix Using stack

① create an empty stack, st.

② Do following for every character x from - left to right



③ If x is

a) operand : output it

b) Left paranthesis : push to st.

c) Right paranthesis : pop from st until left paranthesis is found. output the popped operators.

d) operator : if st is empty, push x to st.

Else Compare with st top.

(i) Higher precedence (than st top), push to st

(ii) Lower precedence, pop st top and output until a higher precedence operator is found. then push x to st.

(iii) Equal precedence, use associativity.

④ pop and output everything from st.

T.C = O(n)

A.S = O(n).

If precedence of two operators, the earlier operator have higher precedence than current operator. So pop the earlier operator & push the current operator.

Eg-1

Input: a + b * c

Input Symbol a	stack	Result (postfix)
a	.	a
+	+	a
b	+	ab
*	+*	ab
c	+*	abc
pop out everything one by one & print	+	abc*
		abc*+

Eg-2: Input: (a+b)*c

Input Symbol a	stack	Result (postfix)
(	(	
a	(	a
+	(+	a
b	(+	ab
)		ab+
*	*	
c	*	ab+c
pop everything		ab+c*

Code leetcode

I/p: a * b / c

Input Symbol (a)	Stack	Result (postfix)
a		a
*	*	a
b	*	ab
/	/	ab*
c	/	ab*c
pop everything from stack & print		ab*c/

I/p: Infix = a + b / c - d * e

Input Symbol (a)	Stack (st)	output (postfix)
a		a
+	+	a
b	+	ab
/	+/	ab
c	+/	abc
-	-	abc/+
d	-	abc/+d
*	-*	abc/+d
e	-*	abc/+de
pop everything & print		abc/+de*-

Evaluation of postfix

I/P: 10 2 * 3 +
O/P: 23

[10*2+3 ⇒ ((10*2)+3)
⇒ ((10 2 *)+3)
⇒ 10 2 * 3 +

I/P: 10 2 + 3 *
O/P: 36

[(10+2)*3 ⇒ ((10+2) * 3)
⇒ ((10 2 +) * 3)
⇒ 10 2 + 3 *

I/P: 10 2 3 ^ ^
O/P: 100000000

[10 ^ 2 ^ 3 ⇒ (10^(2^3))
⇒ (10^(2 3 ^))
⇒ (10 2 3 ^ ^)]

Algorithm for evaluation of postfix:

1) Create an empty stack 'st'.
2) Traverse through every symbol 'x' of given postfix.
   1) If 'x' is an operand, push to 'st'
   2) Else ('x' is an operator)
      (i) op1 = st.pop();
      (ii) op2 = st.pop();
      (iii) Compute op2 x op1 and push the result to 'st'.
3) Return st.top().

I/P: 10 2 * 3 5 * + 9

[Infix: (10*2)+(3*5)-9]

Input symbol(x)	stack (st)
10	10
2	2, 10
*	20
3	3, 20
5	5, 3, 20
*	15, 20
+	35
9	9, 35
-	26

I/P: 10 2 3 ^ ^

Input symbol(x)	stack
10	10
2	2, 10
3	3, 2, 10
^	8, 10
^	100000000

Infix to prefix Conversion (Naive Approach)

Operators	Associativity
^	R-L
*, /	L-R
+, -	L-R

precedency ↑

Ex: Infix: x+y*z

prefix: +x*yz

Ex: Infix: (x+y)*z

prefix: *+xyz

Ex: Infix: x^y^z

prefix = ^x^yz

Ex: Infix = (x+y)*(z+w)

postfix = *+xy+zw

Two steps: ① Fully parenthesis
② start converting from innermost to outermost

→ x+y*z ⇒ (x+(y*z)) ⇒ (x+(*yz)) ⇒ +x*yz

→ (x+y)*z ⇒ ((x+y)*z) ⇒ ((+xy)*z) ⇒ *+xyz

→ x^y^z ⇒ (x^(y^z)) ⇒ (x^(^yz)) ⇒ ^x^yz

→ (x+y)*(z+w) ⇒ ((x+y)*(z+w)) ⇒ ((+xy)*(+zw)) ⇒
*+xy+zw

① x+y*(z-w) ⇒ (x+(y*(z-w)))
⇒ (x+(y*(-zw)))
⇒ (x+(*y-zw))
⇒ +x*y-zw

② x+y*z/w+u ⇒ ((x+((y*z)/w))+u)
⇒ ((x+((*yz)/w))+u)
⇒ ((x+(/*yzw))+u)
⇒ ((+x/*yzw)+u)
⇒ ++x/*yzwu

Infix to prefix using stack :-

precedence
^
* /
+ -
closing Bracket

① create an empty stack, st.

② create an empty string, prefix

③ Do following for every character c from right to left

④ If c is :

a) operand : push it to prefix.

b) Right paranthesis : push to st.

c) Left paranthesis : pop from st until right paranthesis is found. Append the popped character to prefix.

d) operator : If st is empty, push c to st.
else compare with st top.

(i) Higher precedence (than st top) : push c to st.

(ii) Lower precedence : pop st top and append the popped item to prefix until a higher precedence operator is found (or st becomes empty).
push c to st

(iii) Equal precedence : use Associativity.

⑤ pop everything from st and append to prefix.

⑥ Return reverse of prefix

T.C = O(n).
A.S = O(n)

ex-1 : I/p : x+y*z

I/p symbol	stack	prefix (reverse)
z		z
*	*	z
y	*	zy
+	+	zy*
x	+	zy*x
		zy*x+

prefix = reverse of "zy*x+"
= "+x*yz"

ex-2 : I/p : (x+y)*z

I/p symbol	stack	prefix (reverse)
z		z
*	*	z
)	*)	z
y	*)	zy
+	*)+	zy
x	*)+	zyx
(	*	zyx+
		zyx+*

output = *+xyz

I/p : ~~x+y/z-w*u~~

I/p symbol	stack	prefix (reverse)
x		x
+	+	x
y	+	xy
/	+/	xy
z	+/	xyz
-	-	xyz/+
w	-	xyz/+w
*	-*	xyz/+w
u	-*	xyz/+wu
	-	xyz/+wu*-

I/p : x^y^z

I/p symbol	stack	prefix (reverse)
z		z
^	^	z
y	^	zy
^	^^	zy
x	^^	zyx
	^	zyx^^
		zyx^^^

result = ^x^yz

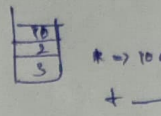
I/p: a + y / z - w * u

I/p Symbol (c)	stack	prefix reverse
u	u	u
*	* u	u
w	* u	uw
-	- * u	uw
z	- * u	uwz
/	- / uwz	uwz*
y	- / uwz*	uwz*y
+	- + uwz*y	uwz*y/
a	- + uwz*y/	uwz*y/a
	uwz*y/a+-	

result = -+a/y*zwu

evaluation of prefix

I/p: + * 10 2 3 [10 * 2+3 ⇒ (10*2)+3 ⇒ (* 10 2)+3 ⇒ + * 10 2 3

O/p: 23 ⇒ 

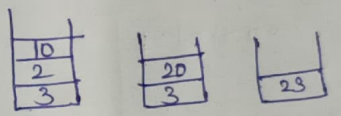
{ ~~Reverse of~~ (R-L evaluation same like postfix)

I/p: * + 10 2 3 [(10+2) * 3 ⇒ ((10+2) * 3) ⇒ ~~* * +~~
((+ 10 2) * 3)
* + 10 2 3

O/p: 36

I/p: ^ 10 ^ 2 3 ⇒ [10 ^ 2 ^ 3 ⇒ (10 ^ (2 ^ 3))
⇒ (10 ^ (^ 2 3))
⇒ ^ 10 ^ 2 3

I/p: + * 10 2 3



op1 = st-pop()
op2 = st-pop()
op1 a op2

a ⇒ operator
op1 and op2 ⇒ are operands.