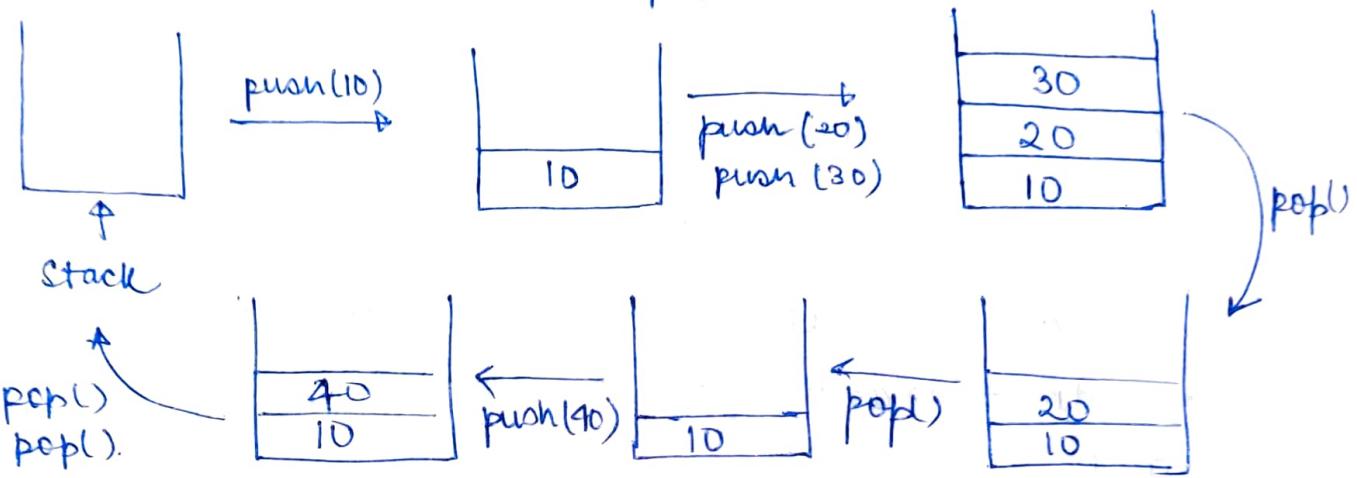


Stacks (Last In first Out)

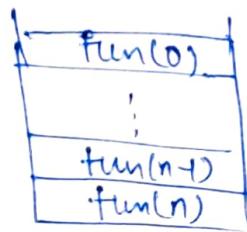
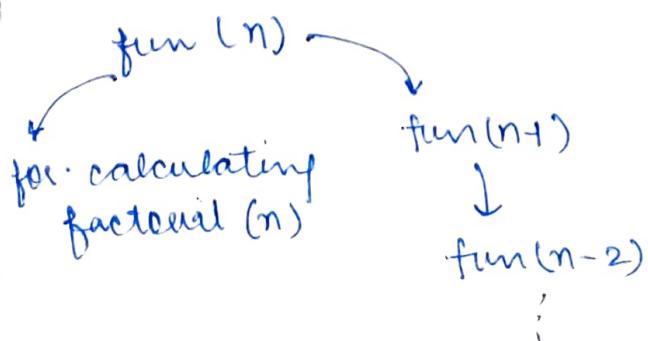
- can be imagined as a str box closed at one end
- Insertion op^{er}ation is called push operation and removal is called pop



Stack Operations

- isEmpty() : Returns true if stack is empty
- push(x) : Inserts an item x to the stack.
- pop() : Removes an item from the top of stack.
- peek() : Returns the top item. (similar can be achieved by top())
- size() : Returns the size of stack.
- * Linear Data Structure
- * Abstract Data type → The behaviour of operations is fixed. For ex- in stacks element must be pushed and popped in the end only.

Ex:- Recursion uses stack



Corner conditions

(a) underflow: when stack is empty and user calls `pop()` or `peek() / top()` functions.

(b) overflow: when push is called on a full stack.

Array Implementation using Array

Functions required → (a) `push(x)`
(b) `pop()`

- (c) `top() / peek()`
- (d) `size()`
- (e) `isEmpty()`

• We use class to implement this functionality
• As we don't want user to ~~know~~ access the full data structure
• We make the array private and make some operations on it, so that user can implement some functionality

• If we're working with array, we have to mention the its size.

• So in this implementation user has to specify the size of stack.

• How to maintain the pos of last item?
We'll create a variable which always points to the last element of stack.

• We'll dynamically create our array.

Ex :- `int * arr;`

After user has defined the size we just create an array of defined size dynamically

`arr = new int [size]`

First we have to define constructor

`MyStack (int capacity){`

```
    cap = capacity;
    arr = new int [cap];
    top = -1;
```

}

Functions

(a) pushing an element

```
void push(int x){  
    top++;  
    arr [top] = x;  
}
```

For defining the size of stack

`cap` → private member
`top` → To keep track to last element
`arr` → To keep track to last element

(d) size of stack

```
int size(){  
    return (top+1);  
}
```

(e) check if Empty()

```
return (top == -1);
```

(c) peeking an element

```
int peek(){  
    return arr [top];  
}
```

`if (top == -1){`
`cout << "Array is empty";`
`return;`

* The function does not include overflow and underflow conditions

② Red shows underflow and overflow conditions.

Time complexity → $O(1)$ in insertion deletion & popping

array implementation

```

class MyStack {
private: int *arr;
int cap;
int top;
MyStack(int c) {
    cap = c;
    top = -1;
    arr = new int [c];
}

public: void push(int x) {
    if (top == cap) {
        cout << "Stack is full";
        return;
    }
    top++;
    arr[top] = x;
}

void pop() {
    if (top == -1) {
        cout << "Stack is empty";
        return;
    }
    int res = arr[top];
    top--;
    return res;
}
}

summary the remaining func.

```

vector implementation

```

class MyStack {
private:
    vector<int> v;
public:
    void push(int x) {
        v.push_back(x);
    }

    int pop() {
        int res = v.back();
        v.pop_back();
        return res;
    }

    int size() {
        return v.size();
    }

    int peek() {
        return v.back();
    }

    bool isEmpty() {
        return v.empty();
    }
}

```

linked list implementation of stack

- In case of arrays all the operations i.e. push, pop, peek... were performed in $O(1)$ complexity except for 1 operation in case of dynamic arrays which includes copying of data from old to new array.

implementation of dynamic sized stack using array

- The only change will be
 - we don't need to input stack size from the user
 - we have to modify the push function as there will be no overflow condition

push function

```

void push(int data) {
    if (top == cap-1) {
        int *newData = new int[cap*2];
        for (int i=0; i<cap; i++) {
            newData[i] = arr[i];
        }
        delete [] arr;
        cap *= 2;
        arr = newData;
    }
}

```

constructor modified
`stacks()` {
 cap = 4;
 ;
}

defined by us

copying element to the another array of size doubled

```

top++;
arr[top] = data;
}

```

Implementation using linked list

(i) using standard technique of insertion and deletion in linked list

we'll maintain a tail pointer.
so that we can perform
insertion operation in $O(1)$ time

pop() function

maintaining a
previous pointer

$25 \rightarrow 35 \rightarrow 45 \rightarrow \text{null}$
↑ ↑
prev tail

pop();
↓
 $25 \rightarrow 35 \rightarrow \text{null}$
↑
tail
prev

✓ previous is at same
because we can't
traverse backwards in
linked list



Wrong approach

traversing from the
node just behind
tail

$25 \rightarrow 35 \rightarrow 45 \rightarrow \text{null}$
↑ ↑
head tail

using temp.

temp = 35
by the help of
temp we can
delete 45 &
update tail also

This states that,
pop function will
be implemented
in $O(n)$ time

which is very
costly for
stack.

Effective Approach

* Insert element at beginning

we know that insertion and deletion at
head of linked list is $O(1)$. so we can
use this concept here.

Ex:-

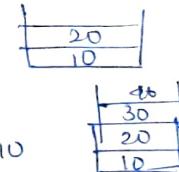
• push(10)



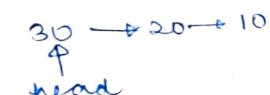
• push(20)



• push(30)
• push(40)



• pop()



* Every operation is of $O(1)$

① For the size() function we have two approaches.

→ null to get size $O(n)$

and just return size
when user calls size
function

maintaining a
private data variable
size and increase
and decrease at
push and pop
respectively

```
void push(int x){  
    Node *newNode = new Node(x);  
    newNode->next = head;  
    head = newNode;  
    size++;
```

pop(),
top(),
isEmpty(),
size()

all can be
implemented
in $O(1)$.

Templates

- Templates can be used when we want to use the same piece of function / class for different data types.

```
template <typename T>
class Pair {
    T x;
    T y;
public:
    void setX(T x) {
        this->x = x;
    }
}
```

To avoid compilation error

- whenever we've created a class using templates, we have to specify the 'T' while creating objects of that class.

* while creating object we'll specify the datatype

For ex:

```
Pair<int> p1;
Pair<double> p2;
Pair<float> p3;
```

* If we want to use more than 1 datatype in a class we can assign different variable names to each of them

```
template <typename T1, typename T2>
class Pair {
    T1 x;
    T2 y;
public:
    ;
    func
    ;
}
```

while creating objects

Pair<int, double> p1

creates a class with
x as int +
y as double.

Pair<int, double> p1

Pair<char, int> p2

x → int
y → double

x → char
y → int

④ How to create a triplet with a pair class

pair<int, int> p1 →

x	y
p1	

pair<pair<int, int>, int> p2 →

pair<int, int>	int
p2	

pair<pair<int, int>, int> p2;
p2.getY(); → gives value of p2.y (simple).
p2.setX(p)
expecting argument
of type pair.

p2.setX(p4);

creating triplet of

- ① int
- ② double
- ③ char

pair<int, pair<double, char> p3;

STACK IN STL (Container adapter)

- we have to mention the datatype, the stack will be storing

```
stack<datatype> s;
           |
stack<int>    stack<char>,    stack<float>.
```

O(1)

functions

- | | | |
|------------------|---------------|----------------|
| (a) s.push(data) | (c) s.pop(); | (e) s.empty(), |
| (b) s.top(); | (d) s.size(); | |

- The difference is that the built-in pop function of stack class is of return type void

```
void pop(){  
}.
```

Balanced Parenthesis

I/P : {a + (b - c) * d}, → True

I/P : {a + (b - c + [e - f])} → False

I/P : ((()) → False

I/P : { } [()] → True.

• we can use stack here, just storing the opening brackets in stack and then popping the top element whenever we see a closing one.

bool isBalanced(string str){

```
stack<int> s;  
for (int i = 0; i < str.length(); i++) {  
    if (str[i] == ')' || str[i] == '}' ||  
        str[i] == ']') {  
        s.pop();  
    } else {  
        s.push(str[i]);  
    }  
}
```

O(1)

else {

```
if (s.empty() == true)  
    return false;
```

```
else if (!matching(s.top(), str[i]))  
    return false;
```

else

```
s.pop();
```

}

```
}  
return (s.empty == true); // extra opening  
brackets
```

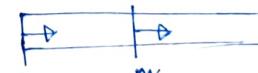
bool matching(char a, char b) {

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

Implementing two stacks with one array

1st Approach

divide the array in two half and then keep 0, m (middle) as starting points of two stacks



Disadvantages

even if we have 0 element in stack 2 i.e right half is empty, on filling left half we can't push more elements in S1

class TwoStacks {

int arr[];

```
void pushX(){}  
void pushY(){}  
int pop1(){}  
int pop2(){}  
int size1(){}  
int size2(){}  
int top1(){}  
int top2(){}  
All  
these func  
are to be  
implemented
```

Method-2



class TwoStacks {

int *arr, top1, top2, cap;

public:

TwoStacks (int n) {

cap = n;

top1 = -1;

top2 = cap;

arr = new int [cap];

}

void push1 (int x) {

if (top1 < top2 - 1) {

top1++;

arr [top1] = x;

}

void push2 (int x) {

if (top1 < top2 - 1) {

top2++;

arr [top2] = x;

}

int pop1 () {

if (top1 >= 0) {

int x = arr [top1];

top1--;

return x;

}

constructor

pushing element to 1st stack

pushing element to 2nd stack

popping element

}

Implement 'k' stacks in an array

• we will use a variable 'sn' to identify the stack we want to operate on

If there are k stacks then the value of sn will be from $0 \leq sn \leq k-1$

Approach-1



divide array into k equal parts (static partitioning)

① space inefficient

Approach-2

• we'll create one more array as nextIndex -

Initially nextIndex is filled as:-

arr [] \rightarrow {0, 0, 0, 0, 0, 0}

nextIndex \rightarrow {1, 2, 3, 4, 5, 6}

Overflow

top1 \geq top2

Underflow

top1 < 0

top2 \geq cap

Ex:- at MI[1], initially the next available space is '2'

similarly for MI[5], there is no free space, hence -1

we have another array of size 'k' storing the top index of each stack

topOfStack = {-1, -1, -1};

topStack[i] \rightarrow tells the index of top of ith stack

Next available serves two purpose

① gives the next available index

② provides the previous element's index

① initially we have our data array which serves as a container to implement k stacks -

② we have an array called nextIndex of size same as data array which initially points to the next free slot in the data array, hence initially

$$\text{nextIndex} = \{1, 2, 3, 4, 5, -1\}$$

③ NextIndex also serves another purpose at any index 'i' it points to the previous index filled by that stack

For ex push(2, 7)

suppose nextAvailable = 3.

$$\text{nextIndex}[3] = \text{top}(2)$$

④ nextAvailable :- It points to the next free slot available in the array.

⑤ top of stack :- It is an array of size ' k ' it points to the top of i^{th} stack

$\text{top of stack}[i] = m$ \rightarrow index of data[i] where top of i^{th} stack is stored.

$\text{freetop} = \text{nextAvailable}$

Ex- steps while we push anything to stack

① check for nextAvailable slot

② put the data into $\text{arr}[\text{nextAvailable}]$.

③ now modify freetop to $\text{next}[\text{freetop}]$.

④ $\text{next}[i] = \text{top}[sn]$ (change $\text{next}[i]$ to point sn)

⑤ $\text{top}[sn] = i$

while pop()

① get the top index from the topOfStack array (i)

② get the previous of the stack using $\text{next}[i]$

now $\text{next}[i]$ becomes top

$$\text{top}[sn] = \text{next}[i]$$

$\text{next}[i] = \text{freetop}$ (fill with next available location)

$$\text{freetop} = i$$

class kstacks {

int *arr, *top, *next;

int R, freetop, cap;

KStacks (int k1, int n) {

R = k1;

cap = n;

arr = new int [cap];

next = new int [cap];

top = new int [k];

for (int i = 0; i < R; i++) {
 top[i] = -1
}

freetop = 0;

for (int i = 0; i < cap - 1; i++) {
 next[i] = i + 1;
}

next[cap - 1] = -1;

}

void push (sn, x) {

arr[freetop] = x;

int n = next[freetop] = top[sn],

top[sn] = freetop

freetop = next[freetop];

next[freetop] = n;

Push Function

```
void push (int x) {
    int i = freeTop;
    freeTop = next[i];
    next[i] = top[sn];
    top[sn] = i;
    arr[i] = x;
}
```

Pop function

```
int pop (int sn) {
    int i = top[sn];
    top[sn] = next[i];
    next[i] = freeTop;
    freeTop = i;
    return arr[i];
}
```

Stock Span Problem

I/P → arr[] = {13, 15, 12, 14, 16, 8, 6, 4, 10, 30}
O/P → {1, 2, 1, 2, 1, 5, 1, 1, 1, 4, 10}

Span → no of consecutive days
with values smaller
or same.

I/P → arr[] = {10, 20, 30, 40} (increasing order)
O/P → {1, 2, 3, 4}

I/P → {40, 30, 20, 10} (decreasing order)
O/P → {1, 2, 1, 1, 1}

I/P → {30, 20, 25, 28, 27, 29}
O/P → {1, 0, 2, 3, 4, 5}

Naive Solution

For every element we go to the left side of it
and count the number of elements smaller than
the cur element

Efficient Solution

$\text{span}[i] = \text{index of curr Element}(i) -$
 $(\text{index of closest greater element on left side}).$

{ 60, 10, 20, 40, 35, 30, 50, 40, 65 }	↓	↓	↓	↓	↓	↓	↓	↓
Index → 0	1	2	3	4	5	6	7	8
Formula - (i)	(1-0)	(2-0)	(3-0)	(4-3)	(5-4)	(6-0)	(7-0)	(8-7)
Ans - ↓	1	2	3	4	5	6	7	8

Also $\text{span}[i] = i+1$ (when there is no greater
element on left side)

$x_0 \ x_1 \ x_2 \ x_3 \dots x_i \ | \ x_{i+1} \ x_{i+2} \ \dots \ x_n$

We have all
the info regarding
that element
(spans)

For x_{i+1} we have

to check the closest
greater element
at left

- we have to store the chain of elements greater than x_{i+1} on left

For $\underline{x_i}$

60, 10, 20, 40, 35 | 30, 50, 70, 65

For 30.

elements on left

35, 40, 60

There may be
3 possibilities

- element may be smaller than element must be between the 1st element of chain and the last element of chain.
- element may be greater than the last element of chain.
- element may be equal to the last element of chain.

80
60, 10, 20, 32, 40, 35, 30

at any point in
the computer

part of array, we
have to find the
nearest greater than

an element on the left to form chain

Let the chain of elements is

$a_0 \dots a_r a_{r+1} \dots a_{n-1}$

- If x_{n-1} is smaller than a_r , then previous greater element would be simply a_r (last)
- If x_{n-1} is between $a_{r+1} \dots a_{n-1}$ then previous greater element would be $x_{[last]}$ (case-2)

- If x_{n-1} is greater than a_r , then there is no previous greater element, $\underline{\text{span}} = n-1$ (case-3)

④ Processing starts from right to left. (LIFO)

1- example. (60, 10, 20, 40, 35, 38, 50, ...)

For element 50

60 40 38

chain

Now we have to process '38' first then 40 and then 60.

suppose the element whose span is to be calculated is x

- Now if $x < 38$ span becomes index of x - index of 38

and 38 is of no use for us, the chain will be updated to 60, 40, x

- else if $x < 60$ span = index of x - index of 60

or chain \rightarrow 60, x

35

40

60

Example

30, 20, 25, 28, 27, 29.
 w or → 0 1 2 3 4 5

at $i=0$ as stack = empty
 $span = 0+1 = 1$
 $i+1$

* element pushed to stack

at $i=1$
 $20 < 30$

$$span = 1 - 0 = 1$$

20 is pushed in stack

at $i=2$
 $25 > 20$

20 is popped

$$30 < 30$$

$$span = 2 - 0 = 2$$

at $i=3$

$$28 > 25$$

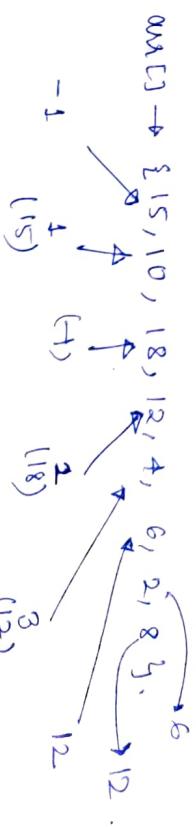
25 popped

$$28 < 30 \quad span = 3 - 0$$



Previous Greater Element

Given an array, we have to find (position-wise) greater element on left. If there is no element (greater) on left then print (-1).



$29 > 28$
 28 popped
 $29 < 30 \quad span = 5 - 0 = 5$

Implementation

```
void printspan(int arr[], int n) {
    stack s;
    s.push(0); // processing first element
    print(-1);
    for (int i = 1; i < n; i++) {
        while (s.isEmpty() == false && arr[s.top()] < arr[i]) {
            popping until { arr[i] < arr[s.top()]} } s.pop();
        span = s.isEmpty() ? i + 1 : i - s.top();
        calculating span & pushing element
        span = s.isEmpty() ? i + 1 : i - s.top();
        print(span);
    }
}
```

Time complexity $\rightarrow O(n)$

Aux - space $\rightarrow O(n) \rightarrow$ Ascending $\} O(n)$
 $\Theta(n) \rightarrow$ Descending $\} O(n)$

In new 18 is greater than 12 on left but 12 is not valid greater.

This question is a subset of previous question.

void getGreaterPrev(int arr[], int n)

```
stack<int> s;
```

```
s.push(0);
```

```
s.push(-1);
```

```
for (int i=0; i<n; i++) {
```

```
while (arr[i] < arr[s.top()]) {
```

```
if (s.isEmpty() == true) {
```

```
s.pop();
```

```
} else if (s.isEmpty()) {
```

```
print(-1)
```

```
else {
```

```
print(arr[s.top()]);
```

```
s.pop();
```

```
} else {
```

Next Greater Element

(position wise
elements)

I/P: {5, 10, 8, 6, 12, 9, 18}

O/P: {10, 12, 12, 12, 18, 18, -1}

Naive solution

for every element, traverse ~~left~~ to right and find

- the closest greater element

$O(n^2)$

Efficient solution

Same approach, just we have to traverse from right

void nextGreater (int arr[], int n)

```
stack<int> s;
```

```
s.push(arr[n-1]); cout << "-1";
```

```
for (int i = n-2; i >= 0; i--) {
```

```
while (s.isEmpty() == false &&
```

```
s.top() <= arr[i]) {
```

```
s.pop();
```

```
} else {
```

```
int nextG = s.empty() ? -1 : s.top();
```

```
cout << nextG;
```

```
s.push(arr[i]);
```

```
} else {
```

- This code produces output in reverse order, we can just store the answers in stack and pop them in the end of function.

- Or we can use a vector and reverse it in the end using reverse (v.begin(), v.end());

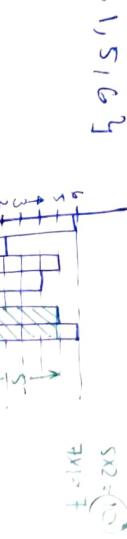
Highest Rectangular Area in Histogram

I/P \rightarrow arr \rightarrow {6, 2, 5, 4, 1, 5, 6}

O/P \rightarrow 10

I/P \rightarrow arr \rightarrow {2, 1, 5, 1}

O/P \rightarrow 5



$3 \times 1 + 3$

$7 \times 1 + 7$

$5 \times 2 + 5$

Initial Solution

For every bar we calculate the area of rectangle (max-area) having that bar as smallest width

$\text{area} = \text{height} \times \text{width}$

area[i] → to increase this width, we move in left and right to find till we encounter a smaller element

```

int getMaxRes (int arr[], int n){
    int res = 0;
    for (int i=0; i<n; i++) {
        for (int curw= arr[i];
            going to cover right boundary of rectangle;
            curw += arr[i]) {
            for (int j=i+1; j<n; j++) {
                if (arr[j] > arr[i]) {
                    curw += arr[j];
                }
                else break;
            }
            for (int j=i-1; j>0; j--) {
                if (arr[j] > arr[i])
                    curw += arr[j];
                else break;
            }
            res = max(res, curw);
        }
    }
    return res;
}

```

Steps

- ① Initialize res = 0.
- ② Find previous smaller & next smaller element for every element

- ③ For every element arr[i].
curw = arr[i].

curw += (i - ps[i] - 1) * arr[i]; // ($i - (ps + 1)$)
curw += (ns[i] - (i + 1)) * arr[i]; // ($(ns - (i + 1))$).
res = max(res, curw);

- ④ return res.

Time complexity → $O(3n)$ space → $O(n)$
 $\rightarrow O(n)$

Efficient Solution :-

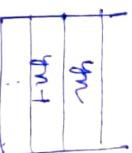
- We want to get the previous and next smaller element in one loop.
- We maintain the stack in such a way so that when we push an element i in the stack the item just below it will be previous greater element.
- When we process the next element, and if the condition ~~not~~ satisfies and we pop some element, then for ex:-

Better Solution $O(n)$

We can precompute next & prev smaller element using the same logic (next & prev greater element).

- We will push element i in the stack and will pop element until $s.top() < arr[i]$ or $s.isEmpty() = \text{true}$.
- If previous value does not exist → \ominus
- If next smaller doesn't exist → \oplus

Let's suppose our element $\rightarrow x$
and stack has elements



then if we pop $ym \rightarrow$ for element ym next smaller

$$= x$$

area of rectangles
formed by ym

ym (index of x)
index of $ym-1$)

area of x

process

- ① Create a stack s .
- ② int res = 0;
- ③ for (int i = 0; i < m; i++) {

while ($s.empty() == \text{false}$ & $s.top() \geq s[i]$) {

top = $s.pop()$;

curr = curr + [top] * (s.empty ? 0 : i - s.top());

top = $s.pop()$;

res = max (res, curr);

s.push (top);
}

```

} while (s.empty() == false) {
    top = s.pop();
    curr = curr + [top] * (s.empty ? 0 : (n - top));
    res = max (res, curr);
    s.push (top);
}
  
```

This will loop "i" for element which do not have any next smaller elements, and this is the reason why may now pop out from the stack.

For these elements, we consider the previous smaller and find the area

$$\text{curr} = (n - \text{stop} - 1) * \text{arr}[\text{stop}];$$

get all the previous smaller elements on right side

Largest rectangular submatrix with all 1's

Input: mat[3][3] = { {0, 1, 1}, {1, 1, 1}, {1, 1, 1} }
 ① $\rightarrow 2 \times 3 = 6$
 ② $\rightarrow 2 \times 4 = 8$
 ③ $\rightarrow 2 \times 3 = 6$
 $6 + 8 + 6 = 20$
 $O/P = 20$

Input: mat[2][2] = { {0, 1}, {1, 1} }
 2x3 = 6
 $1 \times 3 = 3$
 $2 \times 1 = 2$
 $O/P = 0$

Input: mat[1][1] = { {0} }
 $0 \times 0 = 0$

We can use the largest area of histogram problem to solve this.

We can consider every row and calculate the max area formed by bars of all sizes.

for ex

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \rightarrow [3, 4, 0, 0]$$

$[0, 1, 1, 0]$
 $[1, 2, 2, 1]$
 $[2, 3, 3, 2]$

If we start by the last row, we have to compute the height for each column that again calls for an RxC loop.

Efficient solution is to start by the top and then pass the 2d row array to the function while calculating the longest histogram for 2nd row.

We can have something like this →

- ① **longesthistogram (array)** → gives the value of max. area formed by arr val.

- ② **surrounds (vector a, vector b)** :

$y_{blk} = 0$
 $a[i] = 0$
 over $a[i]++;$

$y -$

- ③ call longest histogram for arr[1] to arr[i+1] ← surrounds (arr[1], arr[i+1])

Problem—85 (vector)

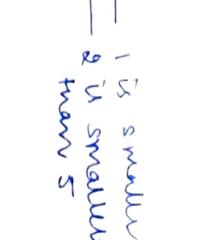
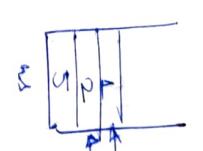
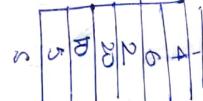
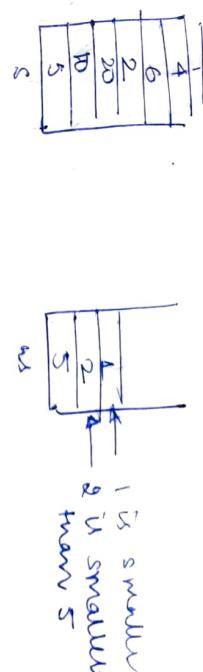
Design a stack that supports getmin().

I/P → push(20), push(10), getmin(), push(5), getmin()

O/P → 10, 5

We maintain an auxiliary array where we will push only those elements who is minimum.

Ex:- 5, 10, 20, 2, 6, 4, 1



push() → $\star s \cdot \text{empty}() \ni s \cdot \text{push}(x)$

$s \cdot \text{push}(x);$
 as $\cdot \text{push}(x);$
 $\star s \cdot \text{push}(x);$

pop() → $\star s \cdot \text{is_top}() == \text{as_top}()$

as $\cdot \text{pop}();$

as $\cdot \text{pop}();$

as $\cdot \text{space} - O(n)$
 time $\text{comp} - O(1)$

* having blk fix space

- The task is we have to store prev minimum elements and current minimum without using any data structure
- we have to store 2 variables values in 1 space i.e. 1 variable.

ans

We will use a variable $\min \rightarrow$ stores the current min elem.

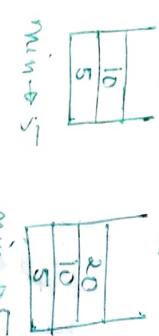
whenever we push(x) and $x < \min$, we will store the $x - \min$ in stack
and \min becomes x

$\text{push}(5)$
 $\text{push}(10)$
 $\text{push}(20)$
 $\text{push}(2)$

$\min \rightarrow 5$



$\min \rightarrow 5$



$\min \rightarrow 5$

$2 - 5 = -3$
we'll push
 t_2



$\min \rightarrow 5$

$2 - 5 = -3$

$\text{push}(x)$

$\text{if } (\text{s.empty}()) \{$
 $\min = x;$

$\text{s.push}(x);$

$\text{else if } (x < \min) \{$

$\text{s.push}(x - \min);$
 $\min = x;$

$\text{else } \{$

$\text{return } t;$

$\}$

$\text{int getmin}()$

$\text{return } \min$

$\}$

$\text{int peek}()$

$t = s.peek();$

$\text{return } (t < 0) ?$

$\min : t$

Second method - Handling Negatives
Initially when we were storing, the special value is $\text{stop}() \leftarrow -\infty$

Here special value will be $\text{stop} \leq \min$
 $2x - \min = (x + (\min - x)) \leq x$

special case \rightarrow $(2x - \min)$
where $x \rightarrow$ new min

$\min \rightarrow$ prev min

Also we are pushing $(2x - \min)$
where $x \rightarrow$ new min

$\min \rightarrow$ prev min

$2x \min - \text{prev-min} = \text{top of stack}$

$\text{prev-min} = 2x \min - \text{top-stack}$

$\text{void push}(int x)$

$\text{if } (\text{s.isEmpty}()) \{$
 $\min = x;$

$\text{s.push}(x);$

$\text{else if } (x < \min) \{$

$\text{s.push}(2 * x - \min);$
 $\min = x;$

$\}$

$\text{else } \{$

$\text{return } \min;$

$\}$

$\text{void pop}()$

$t = s.pop();$

$\text{if } (t < \min) \{$

$\text{res} = \min;$
 $\min = 2 * \min - t;$

$\}$

$\text{else } \{$

$\text{return } \text{res};$

$\}$

$\text{return } t;$

$\text{void pop}()$

$t = s.pop();$

$\text{if } (t < \min) \{$

$\text{res} = \min;$
 $\min = 2 * \min - t;$

$\}$

$\text{else } \{$

$\text{return } \text{res};$

$\}$

$\text{return } t;$

Infix, Prefix and Postfix

$$\begin{array}{ccc} \downarrow & \downarrow & \downarrow \\ x+y & xy+ & +xy \end{array}$$

prefix & postfix are better because

- ① Do not require parenthesis, precedence rules and associativity rules

- ② can be evaluated by writing a program that traverses the given expression exactly once

operators	Associativity
\wedge	R to L
$*, /$	L to R
$+, -$	L to R

cloning bracket
(bottommost priority)

Steps to convert infix to postfix

- ① Parenthesise expression using associativity and precedence rules
- ② convert the innermost to postfix

Ex-1 $x+y * z \rightarrow x + (y * z)$
 $\rightarrow x + (yz*)$
 $xyz*+$

Ex-2 $(x+y)*z \rightarrow ((x+y)*z)$
 $\rightarrow ((xy+)*z)$
 $\rightarrow xy+ z *$

Infix to Postfix

operators	I/P \rightarrow	O/P \rightarrow
-	$a+b*c$	$b*c+a*$
*	$(a+b)*c$	$ab+c*$

Naive Approach

- Fully parenthesise the expression
- And then start with innermost expression

① $a \wedge b \wedge c \rightarrow (a \wedge (b \wedge c))$
 $\rightarrow (a \wedge (bc \wedge)) \rightarrow abc \wedge \wedge$

② $((a+b)* (c+d)) \rightarrow (ab+)* (cd+)$
 $\rightarrow ab+cd+*$

③ $a + b * (c-d) \rightarrow a + (b * (c-d))$
 $\rightarrow a + (b * (cd-))$
 $\rightarrow a + (bcd-*)$
 $\rightarrow abc d - * +$

④ $a + b * c/d + e \rightarrow a + ((b * c)/d) + e$
 $\rightarrow a + ((bc*)/d) + e$
 $\rightarrow a + (bc*d/1) + e$
 $\rightarrow (abc*d/+)+e$
 $\rightarrow abc*d/+e+$

Efficient Solution

Steps

- ① Create an empty stack S.
- ② For every character x from left to right
 - (a) operand : print it
 - (b) left parenthesis :- push to the stack
 - (c) right parenthesis :- pop elements from the stack until left par. is found.

(d) operation -
 (i) if S is empty, push x to S
 (ii) else compare x with $S.pop()$

higher precedence
than $S.top()$

push to S

(i)

push to S
higher precedence
than $S.pop()$

(ii)

push to S
higher precedence
operator is found.

(iii)

If associative
in L to R

R-to-L
(iv)

*

*

c

*

ab + c

ab + c

ab * c

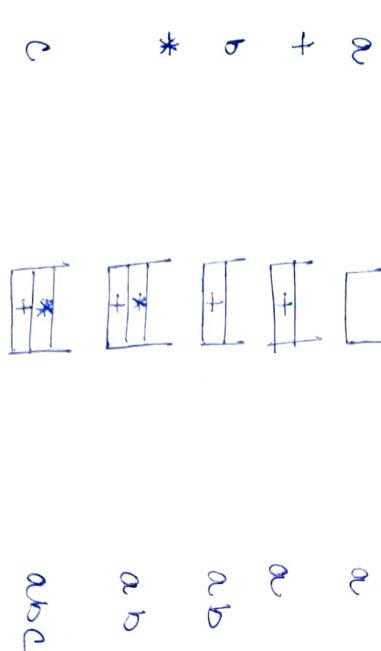
ab * c

Input : $a+b*c$

stack

stack

Result (Postfix)



pop everything
from stack

abc * +

Input symbol

[]

a

[]

a

[]

a

[]

ab

[]

ab

[]

ab*

[]

ab*

[]

ab*c

[]

ab*c/

Result

a

a

a

ab

ab

ab*

ab*

ab*c

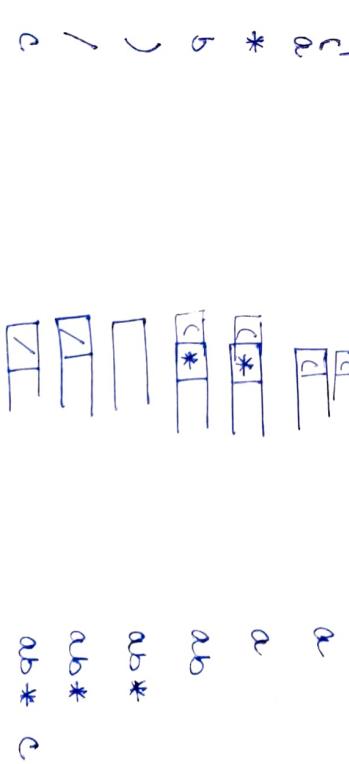
ab*c/

Example-3 $(a*b)/c$

stack

stack

Result



EX-4

$a*b/c$

stack

Result



Example-5 - $a + b/c - d * e$

Input

a

+

b

/

c

-

d

*

e

stack

Output

a

a

ab

ab

abc

abc / +

abc / + d

abc / + de

abc / + de

pop everything

Ex Evaluation of Postfix

I/P $\rightarrow 10\ 2\ * \ 3\ +$

O/P $\rightarrow 23$

I/P $\rightarrow 10\ 2\ + \ 3\ *$

O/P $\rightarrow 36$

I/P $\rightarrow 10\ 2\ * \ 3\ +$

O/P $\rightarrow 36$

I/P $\rightarrow 10\ 2\ + \ 3\ *$

O/P $\rightarrow 36$

using stacks

① operand \rightarrow push in stack
 ② operator \rightarrow pop n elements and apply that operator on n elements & push result to stack.

I/P $\rightarrow x + y * (z - w)$
 O/P $\rightarrow x + (y * (-z w))$
 $x + (* y - z w)$
 $+ x * y - z w$
 $(+ x * y z w) + w$
 $+ + x | * y z w w$

I/P $\rightarrow x + y * z / w + w$
 $x + ((y * z) / w) + w$
 $(x + ((y * z) / w)) + w$
 $x + (1 * y z w) + w$
 $(+ x | * y z w) + w$

(d) operator-

(i) if s is empty, then push x
 (ii) else compare it with s-top

Infix to Prefix conversion

$x+y \rightarrow +xy$

I/P $\rightarrow x + y * z$

O/P $\rightarrow x + (y * z) \Rightarrow x + (* y z)$

I/P \rightarrow

O/P \rightarrow

I/P $\rightarrow (x+y)*z$

O/P $\rightarrow (+xy)*z$

$\rightarrow * + xy z$

I/P $\rightarrow 4 \wedge 5 \wedge 3$

O/P $\rightarrow 4 \wedge (5 \wedge 3) \Rightarrow 4 \wedge (\wedge 5 3)$

$\Rightarrow \wedge 4 \wedge 5 3$

infix \rightarrow $x + y / z - w * u$

Input	Stack	prefix
u		u
*	*	u
w	*w	uw
-	*w-	uw*
z	*w-z	uw*z
/	*w-z/	uw*z
y	*w-z/y	uw*zy
+	*w-z/y+	uw*zy/
x	*w-z/y+x	uw*zy/x
<u>pop()</u>		uw*zy/x + -
		<u>Reverse</u>
		- + * / y z * w u

Evaluation of Prefix expression

I/P \rightarrow $+ * 10 2 3$

O/P \rightarrow 23

- scan \rightarrow from right to left
- whenever operator appears pop last two elements from stack push & push the op1 operator op2 to the stack.
- s.top() \rightarrow result