Data Structure

4. Stack & Queue

Handwritten by pankaj kumar

Stack & Queue

Date

Page No. 181.

Date

Page No. 182.

(*) Stack Introduction & Implementation:

Stack is a linear data structure that follows LIFO (Last in First out) or FILO (First in Last out) order.

* Operation:-

Push: Adds an item to the stack, if stack is full (overflow condition)

```
begin
  if stack is full
     return
  endif
  else
     Increment top
     stack[top] assign value
  end else
end procedure.
```

Pop: Remove an item from the stack. The item are poped in reversed order in which they are pushed, if stack is empty [Underflow condition]

```
begin
  if stack is empty
     return
  endif
  else
     store value of stack[top]
     decrement top
     return value
  end else
end procedure.
```

Peek or Top: return top element

```
begin
   return stack[top]
end procedure
```

Is Empty: return true if stack is empty, else false

```
begin
   if top<1
      return true
   else
      return false
end procedure.
```

Note:- all push(), pop(), isEmpty(), peek() will take O(1) time.

* Applications of the stack:
- Balancing of symbols
- infix to postfix/prefix
- Redo-undo features at many places like editors, photoshop
- Forward and backward features in web browsers
- used in many algorithm like Tower of hanoi, tree traversals, stock span problems, and histogram problems.
- Backtracking algorithm, recursion all uses stack features.
- In memory management. etc.

* Implementation:
① using array ② using linked list

Implementing stack using array.

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000
class stack{
    int top;
  public:
    int a[MAX];
    stack() { top=-1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

bool stack::push(int x){
    if (top>=(MAX-1)){
        cout<<"stack overflow";
        return false;
    }else{
        a[++top]=x;
        return true
    }
}
```

```
int Stack::pop(){
    if (top<0){
        cout<<"stack underflow";
        return 0;
    }
    else{
        int x=a[top--];
        return x;
    }
}
int Stack::peek(){
    if (top<0){
        cout<<"stack is Empty";
        return 0;
    }
    else{
        int x=a[top];
        return x;
    }
}
bool Stack::isEmpty(){ return (top<0); }

int main(){
    class Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout<<s.pop()<<"\n";
    cout<<"Elements in stack:";
    while(!s.isEmpty()){
        cout<<s.peek()<<" ";
        s.pop();
    }
    return 0;
}
```

Op:- 30
Element in stack: 20 10

Note:- It is not dynamic, doesn't grow and shrink depending on needs of runtime.

Implementing stack using Linked list

```
class StackNode{
    public:
        int data;
        StackNode* next;
};

StackNode* newNode(int data){
        StackNode* stackNode = new StackNode();
        stackNode->data = data;
        stackNode->next = NULL;
        return stackNode;
}

int isEmpty(StackNode* root){ return !root;}
Void push(StackNode** root, int data){
        StackNode* stackNode = newNode(data);
        stackNode->next = *root;
        *root = stackNode;
}
int pop(StackNode** root){
        if(isEmpty(*root)) return INT_MIN;
        StackNode* temp = *root;
        *root = (*root)->next;
        int popped = temp->data;
        free(temp);
        return popped;
}
int main(){
        StackNode* root = NULL;
        push(&root, 10);
        push(&root, 20);
        push(&root, 30);
        cout<<pop(&root);
        cout<<"Element present in stack:";
        while(!isEmpty(root)){
                cout<<peek(root)<<" ";
                pop(&root);
        }
}
```

```
int peek(StackNode* root)
{
    if(isEmpty(root))
        return INT_MIN;
    return root->data;
}
```

O/P:

30
Element present in stack:
20 20

Date

Page No. 186.

(*) Queue Introduction & Implementation:

- Queue follows the First in First Out (FIFO) rule - the item that goes in First is the item that comes out First

(*) Basic Operations of Queue.

A queue is an object (an abstract data structure - ADT) that allows the following operations.

- Enqueue: Add an element to the end of the queue.
- Dequeue: Remove an element from the front of the queue.
- isEmpty: check if the queue is empty.
- isFull: check if the queue is Full.
- Peek: Get the value of the the front of queue without removing it.

(*) Working

Queue work as follows:

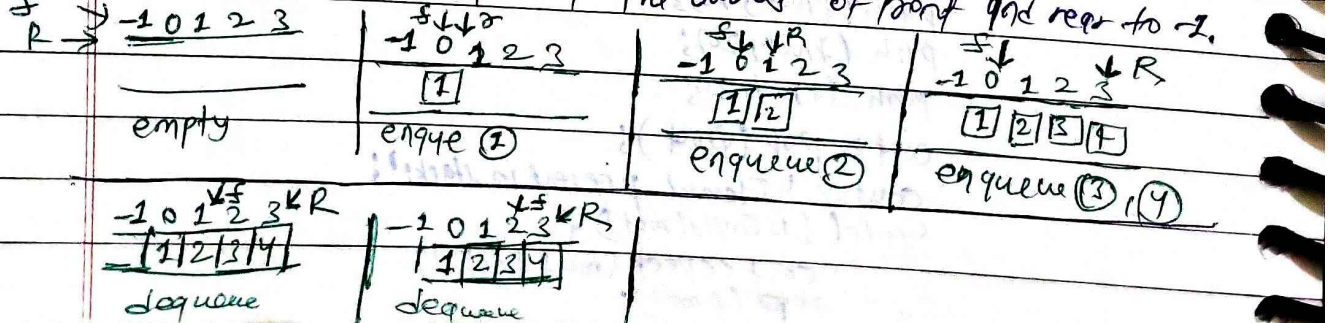
- two pointers FRONT & REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue.
- initially, value of the FRONT and REAR is -1

Enqueue:

- Check if the queue is not Full
- For First element set value of [Front] to 0.
- Increase [REAR] index by 1.
- Add the new element at position [REAR]

Dequeue:

- Check if the queue is not empty.
- return value pointed by FRONT
- increase the FRONT index by 1
- For the last element reset the values of Front and rear to -1.



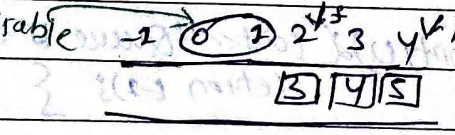
⊛ Implementation :

```
#include <iostream>
#define SIZE 5
using namespace std;
class Queue {
    private:
    int items[SIZE], front, rear;
    public:
     Queue () { front = -1; rear = -1; }
    bool isFull() {
        if (front == 0 && rear == SIZE-1) { return true; }
        return false;
    }
    bool isEmpty () {
        if (front == -1) { return true; }
        return false;
    }
    void enqueue (int ele) {
        if (isFull()) { cout << "Queue is full "; }
        else {
            if (front == -1) front = 0;
            items[++rear] = ele;
            cout << endl << "Inserted " << ele << endl;
        }
    }
    int deQueue () {
        int ele;
        if (isEmpty()) { cout << "Queue is empty" << endl;
                         return (-1); }
        else {
            ele = items[front];
            if (front >= rear) { front = -1; rear = -1; }
            else { front ++; }
            cout << endl << "Deleted -> " << ele << endl;
        }  return (ele);
}
```

```
void display(){
    int i;
    if (isEmpty()) { cout<<endl<<"Empty Queue"<<endl; }
    else {
        for(i=front; i<=rear; i++)
            cout<<items[i]<<" ";
        cout<<endl;
    }
}
int main(){
    Queue q;
    q.dequeue(); //empty Queue
    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);
    q.enqueue(4);
    q.enqueue(5);
    q.enqueue(6); //queue is Full
    q.display(); // 1 2 3 4 5
    q.dequeue(); // remove 1
    q.display(); // 2 3 4 5
    return 0;
}
```

Limitations of Queue

- after a bit of enqueing and dequeuing, the size of queue is reduced
- and we can only use index 0 and 1 only when queue is reset



- After REAR reaches the last index, if we can store extra ele in the empty spaces (0 and 1), we can make use of the empty spaces. This is done by using circular queue.

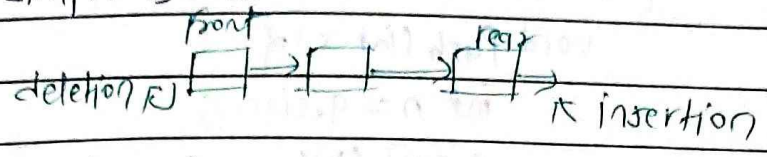
Note :- Dequeue & enqueue → O(1) time

Applications

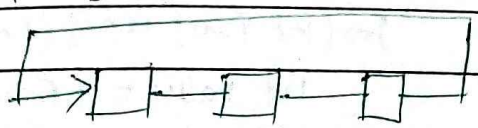
- CPU scheduling, Disk scheduling
- IO Buffers, pipes, File I/O.
- Handling of interrupts in real-time systems.

✱ Types of Queues

① Simple Queue



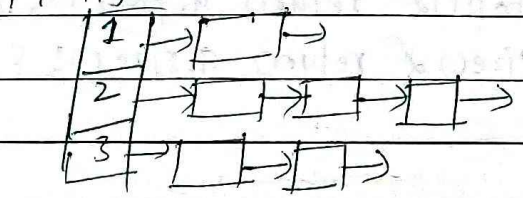
② Circular Queue



- It is better memory utilization

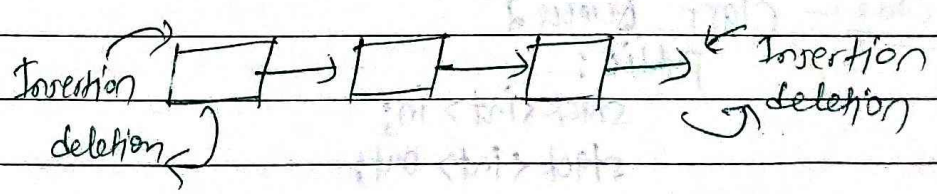
③ priority Queue.

- a special type of queue in which each element is associated with a priority and is served according to its priority. If the element with same priority occurs, they are served according to their order in the queue.



④ Deque (Double Ended Queue)

- insertion and removal of elements can be performed from either from the front or rear. it doesn't follow FIFO



(3) Implement stack using Queue :-

if push :- push into queue from rear and pop all the element, again push them into sequence.

if pop :- pop from queue from front.

Code :-

```
class Stack {
    queue<int> q;
public:
    void Push(int x) {
        int n = q.size();
        q.push(x);
        for(int i=0; i<n; i++) {
            int value = q.front();
            q.pop();
            q.push(value);
        }
    }
    int Pop() {
        int value = q.front();
        q.pop();
        return value;
    }
    int Top() { return q.front(); }
    int Size() { return q.size(); }
};
```

(4) Implement a Queue using Stack :-

→ use 2 stacks.

→ while pop(), shift all element in 2nd stack and pop from 2nd stack and again shift all element to 2nd when the 2nd got empty.

Code :-

```
class Queue {
public:
    stack<int> in;
    stack<int> out;
    void Push(int x) {
        in.push(x);
    }
```

```
int pop(){
    if (out.empty()){
        while (in.size()){
            out.push (in.top());
            in.pop();
        }
    }
    int x = out.top();
    out.pop();
    return x;
}

int Top(){
    if (out. empty()){
        while (in. size()){
            out. push (in.top());
            in.pop();
        }
    }
    return out.top();
}
int size(){
    return in.size() + out.size();
}
};
```

(5) Check for Balanced Parentheses

ex:- str = "[()]" | str = "()[{}()]"

False | True

Code:-

```
bool isValid (string s){
    stack<char> st;
    for(auto it: s){
        if(it == '(' || it == '{' || it == '[') st.push(it);
        else{
            if (st.size()==0) return false;
            char ch = st.top();
            st.pop();
            if((it==')' && ch=='(') || (it==']' && ch=='[')
                || (it=='}' && ch=='{'))
```

```
            continue;
            else return false;
        }
    }
    return st.empty();
}
```

// Time: O(n)
// Space: O(n)

(6) Next Greater element:

- return the next greater element for every element in an array.
- next greater element is first greater element in an array while traversing.

ex:- arr[4,5,2,25]

4→5	2→25
5→25	25→-1

res=[5,25,25,-1]

ex:- arr[2,4,1,3,1,6]

2→4	3→6
4→6	1→6
1→3	6→-1

res=[4,6,3,6,6,-1]

*Approach :-

(i) stack=[ ], ele=25] NGE=-1

(ii) stack=[25], ele=2] NGE=25

res=[5,25,25,-1]

(iii) stack=[25,2], ele=5
∵ 2<5, pop(2)
∴ NGE=25

(iv) stack=[25,5], ele=4
NGE=5

*Code :-

```
class Solution{
    public:
    vector<int> nextGreaterElement(vector<int> arr, int n){
        stack<int> st;
        vector<int> res(n);
        for(int i=n-1; i>=0; i--){
            int currVal=arr[i];
            while(!st.empty() && st.top()<=currVal)
                st.pop();
            res[i]=st.empty()?-1:st.top();
            st.push(currVal);
        }
        return res;
    }
};
```

// TC=O(n)
// SC=O(n)

(7) Next Greater Element in Circular Array:

ex:- A[] = {5, 7, 1, 7, 6, 0}
O/P = [7, -1, 7, -1, 7, 5]

Approach:- The only difference b/w a circular and non-circular array is that while searching for the next greater element in a non-circular array we don't consider the elements left to the concerned element. This can be easily done by inserting the elements of the array A at the end of A, thus making its size double. But we actually don't require any extra space. We can just traverse the array twice. i.e 2*N times.

Code:-

```
Class Solution {
    public:
        vector<int> nextGreaterElements(vector<int>& nums) {
            int n=nums.size();
            vector<int> nge(n,-1);
            Stack<int> st;
            for(int i=2*n-1; i>=0; i--) {
              while(!st.empty() && st.top()<=nums[i%n])
                    st.pop();
              if(i<n) {
                  if(!st.empty()) nge[i]=st.top();
              }
              st.push(nums[i%n]);
            }
            return nge;
        }
};
```

// TC: O(N)
// SC: O(N)

(8) Next Smaller element:-

- entire approach is similar to next greater element except for comparison.

ex: A=[2 1 4 3], O/P=[1, -1, 3, -1]
A=[1 3 2], O/P=[-1, 2, -1]

```
vector<int> nextSmallerElement(vector<int> &arr, int n){
    stack<int> st;
    vector<int> res(n);
    for(int i=n-1; i>=0; i--){
        int currVal = arr[i];
        while(!st.empty() && st.top() >= currVal)
            st.pop();
        res[i] = st.empty() ? -1 : st.top();
        st.push(currVal);
    }
    return res;
}
```

// TC → O(n)
// SC → O(n)

(9) Stock span problem :- given price quotes of stock for n days. We need to find span of stock on any particular day.

- The span of the stock's price today is defined as the max no. of consecutive days (starting from today and going back) for which the stock price was less than or equal to today's price

ex:- A = [100, 80, 60, 70, 60, 75, 85], o/p = [1, 1, 1, 2, 1, 4, 6]
(indices 0 1 2 3 4 5 6)

Approach :- stack = [stores index]

span = [0|0|0|0|0|0|0] (0 1 2 3 4 5 6)

while (currEle >= A(stack.top)) pop stack

span = currEle (pos) − stack.top

- 100 : stack = [ ] ∴ span[0] = 1
  push (0)
- 80 : stack = [0] (top)
  ∵ 80 ≯ 100 ∴ span[1] = 1 − 0 = 1
  push (1)
- 60 : stack = [0, 1] (top)
  ∵ 60 ≯ 80 ∴ span[2] = 2 − 1 = 1
  push (2)
- 70 : stack = [0, 1, 2] (top)
  ∵ 70 > 60 ↑ pop(2)
  ∵ 70 ≯ 80 ∴ span[3] = 3 − 1 = 2 (idx of 80 top of stack)
  push (3) → stack = [0, 1, 3]

and so on

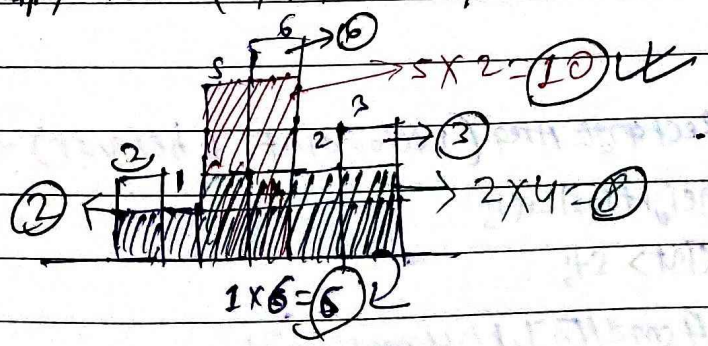
code:-

```
class Solution{
    public:
    vector<int> calculateSpan(int price[], int n){
        vector<int> span(n);
        Stack<int> st;
        st.push(0);
        span[0]=1;
        for(int i=1; i<n; i++){
            int currprice=price[i];
            while(!st.empty() && currprice>=price[st.top()])
                st.pop();
            if(st.empty()){
                span[i]=i+1;
            }else{
                span[i]=i-st.top();
            }
            st.push(i);
        }
        return span;
    }
};
```

(10.) Area of Largest rectangle in Histogram:

Given an array of integers heights representing the histogram's bar height where the width of each bar is 1 return the area of largest rectangle in histogram.

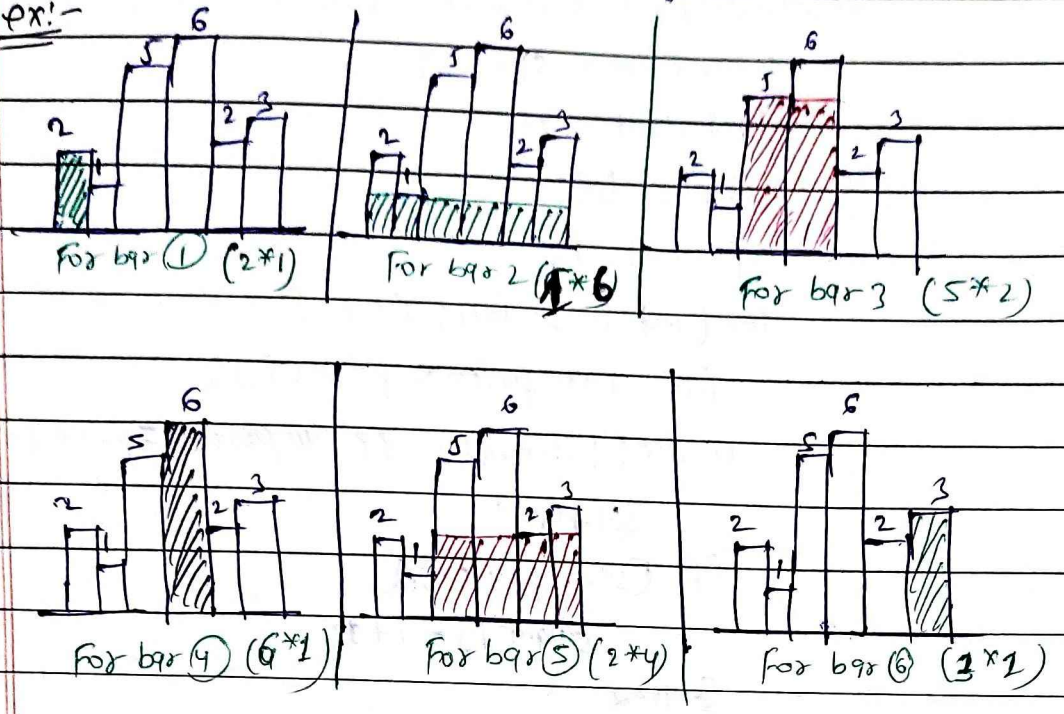
Ip:- h[]={2,1,5,6,2,3}, O/P:- 10



Approach 1 :- Brute Force Approach :-

the institution behind the approach is taking diff bars and finding the maxm width possible using the bar.

ex:-



- The approach is to find the right next smaller and left next smaller element and find the larsest rectangle area in Histogram
- we can find this by using two loop. Time = O(N*N)
  space = O(1)

Approach 2 :-

The instuition behind the approach is the same as finding the smaller element on both side but in an optimized way using the concept of next greater element and the next smaller element.

Code :-

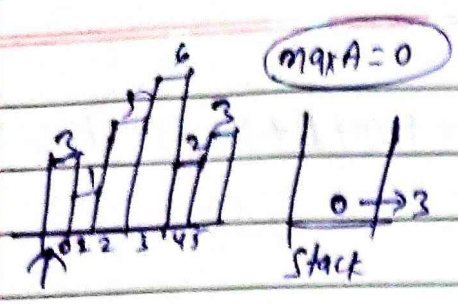
```
class Solution {
    public:
        int largestRectangleArea(vector<int>& heights) {
            int n=heights.size();
            stack<int> st;
            int leftsmall[n],rightsmall[n];
```

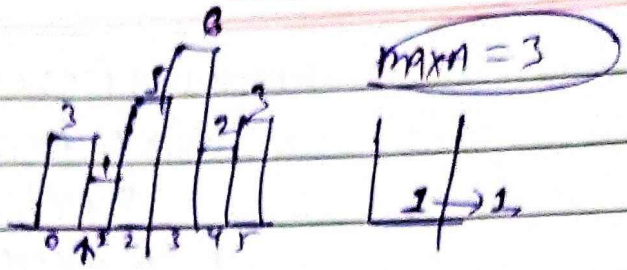
```
for(int i=0; i<n; i++){
    while(!st.empty() && height[st.top()] >= heights[i])
        st.pop();
    if (st.empty())
        leftsmall[i]=0;
    else
        leftsmall[i]= st.top()+1;
    st.push(i);
}

//clear stack to reused
while (!st.empty())
    st.pop();
for (int i=n-1; i>=0; i--){
    while(!st.empty() && heights[st.top()] >= heights[i])
        st.pop();
    if (st.empty())
        rightsmall[i]=n-1;
    else
        rightsmall[i]= st.top() -1
    st.push(i);
}
int maxA=0;
for(int i=0; i<n;i++){
    maxA= max(maxA, heights[i]*(rightsmall[i]-leftsmall[i]
                                +1));
}
return maxA;
}
};
```

Time: O(n)
Space: O(3N)

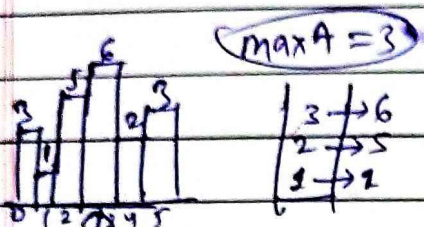
Approach 3:- This approach is a single pass approach instead of a two-pass approach. When we traverse the array by finding the next greater element, we found that some elements were inserted into the stack which signifies that after them the smallest element is themselves.

So, area = arr[i] * (right smaller - left smaller - 1)

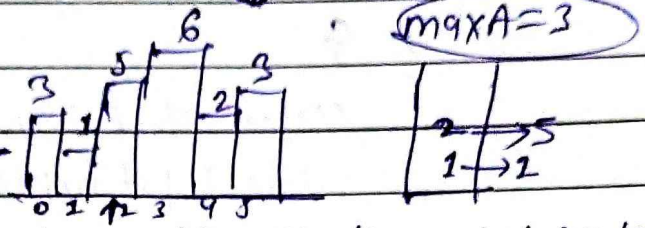




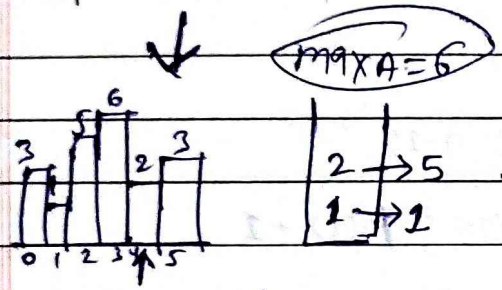
since 1 is smaller than 3 this means right smaller for 3 is 1 and left smaller is someone before 3 in stack.



since, 5 is smaller than 6, but we don't know right smaller, so, push it directly.

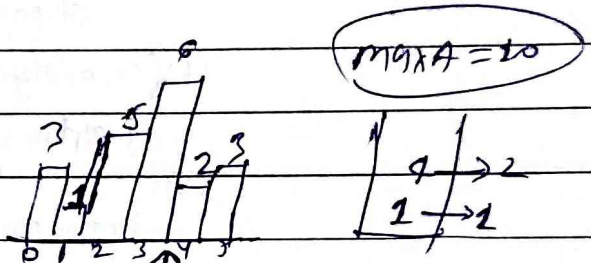


since 1 is smaller than 5. but we don't know right smaller so, push it



∵, 2 is smaller than 6, so it ~~means it is right smaller, and popping out 6 from the stack, 5~~ is left smaller for '6'

$\therefore maxA = max((6*(4-2-1)), 3) = 6$



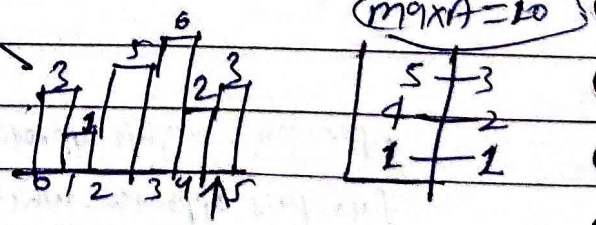
∵ 2 is smaller than 5 so it means it is right smaller, and popping out 5 from the stack, 1 is left smaller for 5.

$\therefore maxA = max((5*(4-1)), 6) = 10$

now, 2 is smaller than 2 so push it (2)

(I) since now, there is no ele. so consider 6 index as right smaller for '5', so pop out '3' and consider 2 as left smaller.

$maxA = 3*(6-4-2) = 3$



(II) consider 6 index as right smaller for '4' index and 1 as left smaller for '4' index and calculate maxA.

∵ 3 is greater than 2 we will push directly '3'

(III) same for '1' index.

```
int largestRectangleArea(vector<int> &histo){
    stack<int> st;
    int maxA=0;
    int n=histo.size();
    for(int i=0; i<=n; i++){
        while(!st.empty() && (i==n || histo[st.top()]>=histo[i])){
            int height=histo[st.top()];
            st.pop();
            if(st.empty())
                width=i;
            else
                width=i-st.top()-1;
            maxA=max(maxA, width*height);
        }
        st.push(i);
    }
    return maxA;
}
```

TC = O(N) + O(N)
SC = O(N)

(11) Sliding Window Maximum :-

Given an array, and a window of size k, which is moving from the very left of the array to the very right. there can be only k numbers in window. Return maxm sliding window.

ex:- arr = [4, 0, -1, 3, 5, 3, 6, 8], k=3.

[4, 3, 5, 5, 6, 8]

Approach 1: (Brute Force):- we take a window of size k and calculate the maximum element in it. then shift our window to next position and do the same process untill reach end of array.

Time complexity: O(N²)
space complexity: O(k)

Approach 2: (Using dequeue):-

- We address this problem with the help of a data structure that keeps checking whether the incoming element is larger than the already present elements. This could be implemented with the help of a dequeue. When shifting our window, we push the new element in from the rear of our de-queue.
- Every time before entering a new element, we first need to check whether the element present at the front is out of bounds of our present window size. if so we need to pop out. Also we need to check from the rear that the element is smaller than the incoming ele. if yes, there is no point to store them so pop them out. Finally at the front be our largest element.

Code:-

```
vector<int> maxSlidingWindow(vector<int>& nums, int k){
    deque<int> dq;
    vector<int> ans;
    for(int i=0; i<nums.size(); i++){
        if(!dq.empty() && dq.front()==i-k) dq.pop_front();
        while(!dq.empty() && nums[dq.back()]<nums[i])
            dq.pop_back();
        dq.push_back();
        if(i>=k-1) ans.push_back(nums[dq.front()]);
    }
    return ans;
}
```

Time: O(N)
Space: O(k)

(12.) The Celebrity Problem

- A celebrity is a person who is known to all but does not know anyone at a party. If you go to a party of N people, find if there is a celebrity in the party or not

Ex:- N=3 0 1 2
m=0[[0,1,0],
1 [0,0,0],
2 [0,1,0]]
o/p = 1.
0th & 2nd person both know 1
∴, 1 is the celebrity.

N=2 0 1
m[][] = 0. [[0,1],
1 [1,0],
o/p = -1,
The two people both know each other at party so, None of them is celebrity.

① Using stack:

```
int celebrity (vector<vector<int>> &M, int n){
    stack<int> s;
    for(int i=0; i<n; i++){
        s.push(i);
    }
    while(s.size() > 1){
        int a = s.top();
        s.pop();
        int b = s.top();
        s.pop();
        if (M[a][b]) push(b);
        else push(a);
    }
    int ans = s.top();
    for(int i=0; i<n; i++){
        if (m[ans][i] != 0) return -1;
        [redacted]
        if (ans != i and m[i][ans] != 1) return -1;
    }
    return ans;
}
```

Time: O(n)
Space: O(n)

(2) Using Two Pointer :-

```
int celebrity (vector<vector<int>>m, int n){
    int i=0,j=n-1;
    while (i<j){
        if (m[i][j]==1)
            j--;
        else
            i++;
    }
    int ans=i;
    for(i=0; i<n; i++){
        if(i!=ans){
            if (m[i][ans]==0 || m[ans][i]==1)
                return -1;
        }
    }
    return ans;
}
```

Time : O(n)
Space : O(1)

(13) LRU Cache :

- Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.
- Implement the LRUCache class:

[- LRUCache(int capacity)] initialize the LRU cache with +ve size (capacity)

[- int get(int key)] Return the value of the key if key exists, otherwise return -1.

[- void put(int key, int value)] Update the value of the key if the key exists. Otherwise, add the key-value pair to cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.

Note :- The function get and put must each run in O(1) average time complexity.

Date

Page No. 203.

Input :-

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1,1], [2,2], [1], [3,3], [2], [4,4], [1], [3], [4]]

Output :-

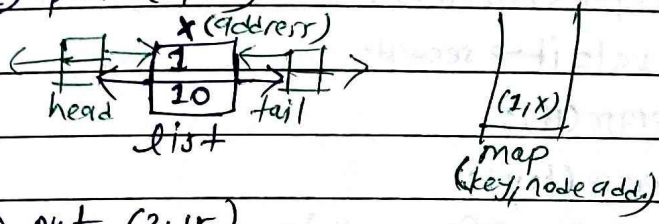
[null, null, null, 1, null, -1, null, -1, 3, 4]

Prerequisite :- Hashmaps & DLL (Doubly linked list) :

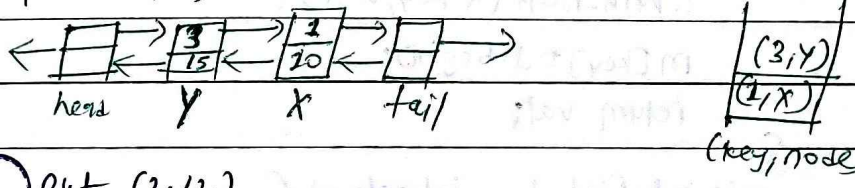
Dry run :- size = 3
put(1, 10)
put (3, 15)
put (2, 12)
get (3)
put (4, 25)

① create a DLL & hashmap.

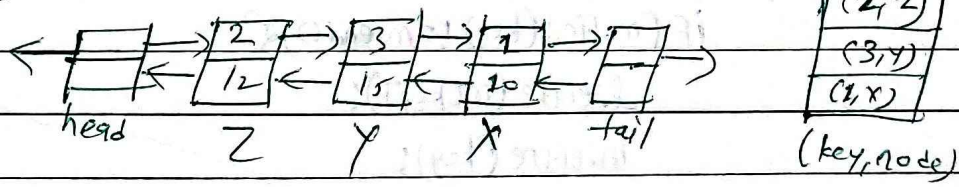
② put (1, 10)



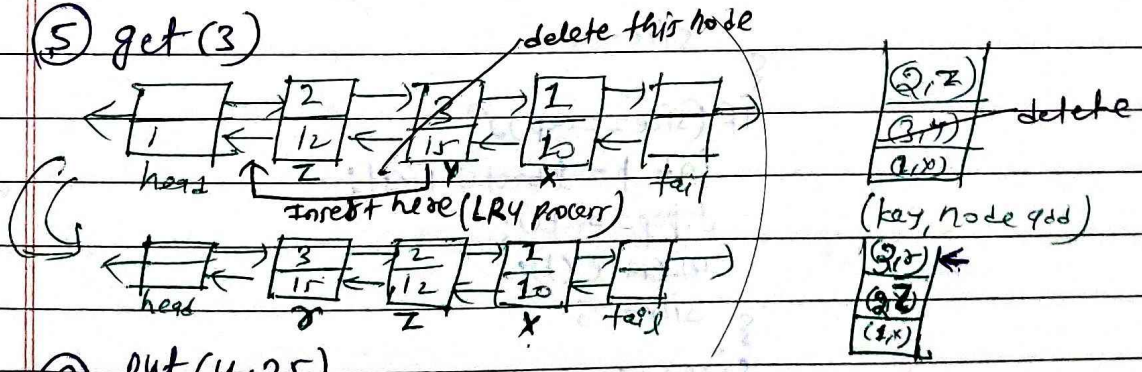
③ put (3, 15)



④ put (2, 12)

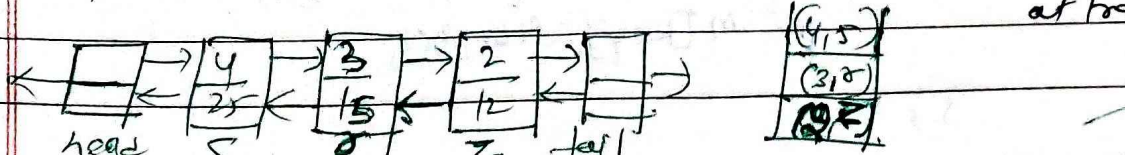


⑤ get (3)



⑥ put (4, 25)

∴ the size is full so delete the last (from back) and insert (4,25) at front)



Code:-

```
class LRUCache{
    int size;
    int cap;
    unordered_map<int, list<pair<int,int>>::iterator> m;
    list <pair<int,int>> l;
public:
    LRUCache(int capacity) {
        cap=capacity;
        size=0;
    }
    int get(int key) {
        if(m.find(key)==m.end()) return -1;
        list<pair<int,int>>::iterator it =m[key];
        int val=it->second;
        l.erase(it);
        m.erase(key);
        l.push_front({key,val});
        m[key]=l.begin();
        return val;
    }
    void put(int key, int value) {
        if(m.find(key)!=m.end()){
            l.erase(m[key]);
            m.erase(key);
            size--;
        }
        if(size==cap){
            int k=l.back().first;
            l.pop_back();
            m.erase(k);
            size--;
        }
        size++;
        l.push_front({key,value});
        m[key]=l.begin();
    }
};
```