1. **Parenthesis Checker**

Given an expression string **exp**. Examine whether the pairs and the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp.  
For example, the program should print 'balanced' for exp = “[()]{}{[()()]()}” and 'not balanced' for exp = “[(])”

**Input:**  
The first line of input contains an integer T denoting the number of test cases.  Each test case consists of a string of expression, in a separate line.

**Output:**  
Print 'balanced' without quotes if the pair of parenthesis is balanced else print 'not balanced' in a separate line.

**Constraints:**  
1 ≤ T ≤ 100  
1 ≤ |s| ≤ 105

**Example:  
Input:**  
3  
{([])}  
()  
([]

**Output:**  
balanced  
balanced  
not balanced

Code:

using namespace std;

bool balance(string str)

{

stack<char> st;

for(int i=0;i<str.length();i++)

{

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

st.push(str[i]);

else

{

if(st.empty())

return false;

char top=st.top();

if(top=='(' && str[i]!=')')

return false;

if(top=='[' && str[i]!=']')

return false;

if(top=='{' && str[i]!='}')

return false;

st.pop();

}

}

if(!st.empty())

return false;

return true;

}

int main()

{

int t;

cin>>t;

while(t--)

{

string str;

cin>>str;

if(balance(str))

cout<<"balanced"<<endl;

else

cout<<"not balanced"<<endl;

}

return 0;

}

2. **Next larger element**

Given an array **A** of size **N** having distinct elements, the task is to find the next greater element for each element of the array in order of their appearance in the array. If no such element exists, output **-1**

**Input:**  
The first line of input contains a single integer **T** denoting the number of test cases.Then **T** test cases follow. Each test case consists of two lines. The first line contains an integer **N** denoting the size of the array. The Second line of each test case contains**N**space separated positive integers denoting the values/elements in the array **A**.

**Output:**  
For each test case, print in a new line, the next greater element for each array element separated by space in order.

**Constraints:**  
1<= T <= 100  
1 <= N <= 107  
1 <= Ai <= 1018  
**Example:  
Input**  
2  
4  
1 3 2 4  
4  
4 3 2 1  
**Output**  
3 4 4 -1  
-1 -1 -1 -1

**Code:**

using namespace std;

#define ll long long

int main()

{

ll t;

cin>>t;

while(t--)

{

ll n;

cin>>n;

ll arr[n];

for(ll i=0;i<n;i++)

cin>>arr[i];

ll ans[n];

stack<ll> st;

for(ll i=n-1;i>=0;i--)

{

while(!st.empty() && arr[i]>=st.top())

st.pop();

ans[i]=st.empty()?-1 : st.top();

st.push(arr[i]);

}

for(ll i=0;i<n;i++)

cout<<ans[i]<<" ";

cout<<endl;

}

return 0;

}

3. **Queue using two Stacks**

Implement a Queue using 2 stacks**s1** and**s2** .  
A Query **Q** is of 2 Types  
**(i)** 1 x (a query of this type means  pushing **'x'** into the queue)  
**(ii)** 2   (a query of this type means to pop element from queue and print the poped element)

**Example 1:**

**Input:**

Q = 5

Queries = 1 2 1 3 2 1 4 2

**Output:** 2 3

**Explanation:** In the first testcase

1 2 the queue will be {2}

1 3 the queue will be {2 3}

2   poped element will be 2 the queue

  will be {3}

1 4 the queue will be {3 4}

2   poped element will be 3.

**Example 2:**

**Input:**

Q = 4

Queries = 1 2 2 2 1 4

**Output:** 2 -1

**Explanation:** In the second testcase

1 2 the queue will be {2}

2   poped element will be 2 and

  then the queue will be empty

2   the queue is empty and hence -1

1 4 the queue will be {4}.

**Your Task:**  
You are required to complete the two methods **push** which take one argument an integer **'x'** to be pushed into the queue and **pop** which returns a integer poped out from other queue(-1 if the queue is empty). The **printing** is done **automatically**by the**driver code**.

**Expected Time Complexity** : O(1) for both **push()**and O(N) for **pop().**  
**Expected Auxilliary Space**: O(N).

**Constraints:**  
1 <=Q <= 100  
1 <= x <= 100

Code:

#include<bits/stdc++.h>

using namespace std;

class StackQueue{

private:

stack<int> s1;

stack<int> s2;

public:

void push(int B);

int pop();

};

int main()

{

int T;

cin>>T;

while(T--)

{

StackQueue \*sq = new StackQueue();

int Q;

cin>>Q;

while(Q--){

int QueryType=0;

cin>>QueryType;

if(QueryType==1)

{

int a;

cin>>a;

sq->push(a);

}else if(QueryType==2){

cout<<sq->pop()<<" ";

}

}

cout<<endl;

}

}

// } Driver Code Ends

/\* The structure of the class is

class StackQueue{

private:

// These are STL stacks ( http://goo.gl/LxlRZQ )

stack<int> s1;

stack<int> s2;

public:

void push(int);

int pop();

}; \*/

/\* The method push to push element into the queue \*/

void StackQueue :: push(int x)

{

s1.push(x);

// Your Code

}

/\*The method pop which return the element poped out of the queue\*/

int StackQueue :: pop()

{

if(s1.empty())

return -1;

//empty s1 in s2

while(!s1.empty())

{

s2.push(s1.top());

s1.pop();

}

int ans=s2.top();

s2.pop();

//again make s2 empty and push all elements in s1

while(!s2.empty())

{

s1.push(s2.top());

s2.pop();

}

return ans;

// Your Code

}

4. **Stack using two queues**

Implement a Stack using two queues**q1** and**q2**.

**Example 1:**

**Input:**

push(2)

push(3)

pop()

push(4)

pop()

**Output:** 3 4

**​Explanation:**

push(2) the stack will be {2}

push(3) the stack will be {2 3}

pop() poped element will be 3 the

  stack will be {2}

push(4) the stack will be {2 4}

pop()   poped element will be 4

**Example 2:**

**Input:**

push(2)

pop()

pop()

push(3)

**Output:** 2 -1

**Your Task:**

Since this is a function problem, you don't need to take inputs. You are required to complete the two methods **push()** which takes an integer **'x'** as input denoting the element to be pushed into the stack and **pop()** which returns the integer poped out from the stack(**-1** if the stack is empty).

**Expected Time Complexity:**O(1) for **push()**and O(N) for **pop()**(or vice-versa).  
**Expected Auxiliary Space:**O(1) for both **push()**and **pop()**.

**Constraints:**  
1 <=Number of queries <= 100  
1 <= values of the stack <= 100

Code:

#include<bits/stdc++.h>

using namespace std;

class QueueStack{

private:

queue<int> q1;

queue<int> q2;

public:

void push(int);

int pop();

};

int main()

{

int T;

cin>>T;

while(T--)

{

QueueStack \*qs = new QueueStack();

int Q;

cin>>Q;

while(Q--){

int QueryType=0;

cin>>QueryType;

if(QueryType==1)

{

int a;

cin>>a;

qs->push(a);

}else if(QueryType==2){

cout<<qs->pop()<<" ";

}

}

cout<<endl;

}

}

// } Driver Code Ends

/\* The structure of the class is

class QueueStack{

private:

queue<int> q1;

queue<int> q2;

public:

void push(int);

int pop();

};

\*/

/\* The method push to push element into the stack \*/

void QueueStack :: push(int x)

{

q1.push(x);

// Your Code

}

/\*The method pop which return the element poped out of the stack\*/

int QueueStack :: pop()

{

if(q1.empty())

return -1;

int n=q1.size()-1;

while(n--)

{

q2.push(q1.front());

q1.pop();

}

int ans=q1.front();

q1.pop();

while(!q2.empty())

{

q1.push(q2.front());

q2.pop();

}

return ans;

}

5. **Get minimum element from stack**

You are given **N** elements and your task is to Implement a Stack in which you can get minimum element in O(1) time.

**Example 1:**

**Input:**

push(2)

push(3)

pop()

getMin()

push(1)

getMin()

**Output:** 3 2 1

**Explanation:** In the first test case for

query

push(2)  the stack will be {2}

push(3)  the stack will be {2 3}

pop() poped element will be 3 the

  stack will be {2}

getMin() min element will be 2

push(1) the stack will be {2 1}

getMin() min element will be 1

**Your Task:**  
You are required to complete the three methods **push()** which take one argument an integer **'x'** to be pushed into the stack, **pop()** which returns a integer poped out from the stack and **getMin()** which returns the min element from the stack. (-1 will be returned if for **pop() and getMin()**the stack is empty.)

**Expected Time Complexity** : O(1) for all the 3 methods.  
**Expected Auixilliary Space** : O(1) for all the 3 methods.

**Constraints:**  
1 <= Number of queries <= 100  
1 <= values of the stack <= 100

Code:

#include<bits/stdc++.h>

using namespace std;

class \_stack{

stack<int> s;

int minEle;

public :

int getMin();

int pop();

void push(int);

};

int main()

{

int t;

cin>>t;

while(t--)

{

int q;

cin>>q;

\_stack \*a = new \_stack();

while(q--){

int qt;

cin>>qt;

if(qt==1)

{

//push

int att;

cin>>att;

a->push(att);

}

else if(qt==2)

{

//pop

cout<<a->pop()<<" ";

}

else if(qt==3)

{

//getMin

cout<<a->getMin()<<" ";

}

}

cout<<endl;

}

}

// } Driver Code Ends

/\*

The structure of the class is as follows

class \_stack{

stack<int> s;

int minEle;

public :

int getMin();

int pop();

void push(int);

};

\*/

/\*returns min element from stack\*/

int \_stack :: getMin()

{

if(s.empty())

return -1;

return minEle;

//Your code here

}

/\*returns poped element from stack\*/

int \_stack ::pop()

{

if(s.empty())

return -1;

//poping will not effect minEle value

else if(s.top()>minEle)

{

int ans=s.top();

s.pop();

return ans;

}

else

{

int ans=minEle;

//update minEle variable

minEle=2\*minEle-(s.top());

s.pop();

return ans;

}

}

/\*push element x into the stack\*/

void \_stack::push(int x)

{

if(s.empty())

{

minEle=x;

s.push(x);

return;

}

else

{

if(x<minEle)

{

s.push(2\*x-minEle);

minEle=x;

}

else

{

s.push(x);

}

return;

}

//Your code here

}

6. **Rotten Oranges**

Given a matrix of dimension **r**\***c** where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:  
**0**: Empty cell  
**1** : Cells have fresh oranges  
**2** : Cells have rotten oranges

So, we have to determine what is the minimum time required to rot all oranges. A rotten orange at index [i,j] can **r**ot other fresh orange at indexes [i-1,j], [i+1,j], [i,j-1], [i,j+1] (**up**, **down**, **left** and **right**) in unit time. If it is impossible to rot every orange then simply return -1.

**Input:**  
The first line of input contains an integer T denoting the number of test cases. Each test case contains two integers r and c, where r is the number of rows and c is the number of columns in the array a[]. Next line contains space separated r\*c elements each in the array a[].

**Output:**  
Print an integer which denotes the minimum time taken to rot all the oranges (-1 if it is impossible).

**Constraints:**  
1 <= T <= 100  
1 <= r <= 100  
1 <= c <= 100  
0 <= a[i] <= 2

**Example:  
Input:**  
2  
3 5  
2 1 0 2 1 1 0 1 2 1 1 0 0 2 1  
3 5  
2 1 0 2 1 0 0 1 2 1 1 0 0 2 1  
  
**Output:**  
2  
-1

Code:

using namespace std;

bool issafe(vector<vector<int> > &grid,int i,int j,int n,int m)

{

if(i<0||i>=n||j<0||j>=m) return false;

else if(grid[i][j]!=1) return false;

return true;

}

bool invalid(vector<vector<int> > &grid,int n,int m)

{

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

{

for(int j=0;j<m;j++)

if(grid[i][j]==1) return true;

}

return false;

}

int rotOranges(vector<vector<int> > &grid, int n, int m)

{

queue<pair<int,int> > q;

// int n=grid.size(),m=grid[0].size();

int time=0;

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

{

for(int j=0;j<m;j++)

{

if(grid[i][j]==2) q.push({i,j});

// cout<<i<<" "<<j<<endl;}

}

}

while(!q.empty())

{

int size=q.size();

time++;

while(size--)

{

pair<int,int> p=q.front(); q.pop();

int i=p.first;

int j=p.second;

if(issafe(grid,i-1,j,n,m)) {grid[i-1][j]=2;q.push({i-1,j}); }

if(issafe(grid,i+1,j,n,m)) {grid[i+1][j]=2;q.push({i+1,j});}

if(issafe(grid,i,j-1,n,m)) {grid[i][j-1]=2;q.push({i,j-1});}

if(issafe(grid,i,j+1,n,m)) {grid[i][j+1]=2;q.push({i,j+1});}

}

}

if(time>0) time--;

if(invalid(grid,n,m)) return -1;

return time;

}

void solve(void)

{

int r,c;

cin>>r>>c;

vector<vector<int>> grid(r,vector<int>(c));

for(int i=0;i<r;i++)

{

for(int j=0;j<c;j++)

cin>>grid[i][j];

}

cout<<rotOranges(grid,r,c)<<endl;

}

int main()

{

int t;

cin>>t;

while(t--)

solve();

return 0;

}

7. **Maximum of all subarrays of size k**

Given an array **A** and an integer **K**. Find the maximum for each and every contiguous subarray of size K.

**Input:**  
The first line of input contains an integer T denoting the number of test cases. The first line of each test case contains a single integer N denoting the size of array and the size of subarray K. The second line contains N space-separated integers A1, A2, ..., AN denoting the elements of the array.

**Output:**  
Print the maximum for every subarray of size k.

**Constraints:**  
1 ≤ T ≤ 200  
1 ≤ N ≤ 107  
1 ≤ K ≤ N  
0 ≤ A[i] <= 107

**Example:  
Input:**  
2  
9 3  
1 2 3 1 4 5 2 3 6  
10 4  
8 5 10 7 9 4 15 12 90 13

**Output:**  
3 3 4 5 5 5 6  
10 10 10 15 15 90 90

Code:

using namespace std;

int main()

{

int t;

cin>>t;

while(t--)

{

int n,k;

cin>>n>>k;

int a[n];

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

cin>>a[i];

deque<int> q;

int i=0;

for(;i<k;i++)

{

while(!q.empty()&&a[i]>=a[q.back()])

q.pop\_back();

q.push\_back(i);

}

for(;i<n;i++)

{

cout<<a[q.front()]<<" ";

while(!q.empty()&&q.front()<=i-k)

q.pop\_front();

while(!q.empty()&&a[i]>=a[q.back()])

q.pop\_back();

q.push\_back(i);

}

cout<<a[q.front()]<<endl;

q.pop\_front();

}

return 0;

}

**8. First non-repeating character in a stream**

Given an input stream of **N** characters consisting only of lower case alphabets. The task is to find the first non repeating character, each time a character is inserted to the stream. If no non repeating element is found print -1.

**Input:**  
The first line of input contains an integer **T** denoting the no of test cases. Then T test cases follow. Each test case contains an integer N denoting the size of the stream. Then in the next line are **x** characters which are inserted to the stream.

**Output:**  
For each test case in a new line print the first non repeating elements separated by spaces present in the stream at every instinct when a character is added to the stream, if no such element is present print -1.

**Constraints:**  
1 <= T <= 200  
1 <= N <= 500

**Example:  
Input:**  
2  
4  
a a b c  
3  
a a c

**Output:**  
a -1 b b  
a -1 c

CODE:

using namespace std;

int main()

{

int t;

cin>>t;

while(t--)

{

int n;

cin>>n;

vector<char> v;

deque<char> dq;

map<char,int> m;

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

{

char x;

cin>>x;

v.push\_back(x);

m[x]=0;

}

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

{

char x=v[i];

if(m[x]==0)

{

if(dq.size()==0)

{

cout<<x<<" ";

dq.push\_front(x);

m[x]=-1;

}

else

{

cout<<dq.front()<<" ";

dq.push\_back(x);

m[x]=-1;

}

}

else

{

deque<char>::iterator it=find(dq.begin(),dq.end(),x);

if(it!=dq.end())

dq.erase(it);

if(dq.size()==0)

cout<<"-1"<<" ";

else

cout<<dq.front()<<" ";

}

}

cout<<endl;

}

return 0;

}

**9. Circular tour**

Suppose there is a circle. There are **N** petrol pumps on that circle. You will be given two sets of data.  
**1.** The amount of petrol that every petrol pump has.  
**2.** Distance from that petrol pump to the next petrol pump.  
Find a starting point where the truck can start to get through the complete circle without exhausting its petrol in between.  
**Note :**  Assume for 1 litre petrol, the truck can go 1 unit of distance.

**Example 1:**

**Input:**

N = 4

Petrol = 4 6 7 4

Distance = 6 5 3 5

**Output:** 1

**Explanation: T**here are 4 petrol pumps with

amount of petrol and distance to next

petrol pump value pairs as {4, 6}, {6, 5},

{7, 3} and {4, 5}. The first point from

where truck can make a circular tour is

2nd petrol pump. Output in this case is 1

(index of 2nd petrol pump).

**Your Task:**  
Your task is to complete the function **tour**() which takes the required data as inputs and returns an integer denoting a point from where a truck will be able to complete the circle (The truck will stop at each petrol pump and it has infinite capacity). If there exists multiple such starting points, then the function must return the first one out of those. (return -1 otherwise)

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space**: O(N)

**Constraints:**  
1 <= N <= 10000  
1 <= petrol, distance <= 1000

CODE:

#include<bits/stdc++.h>

using namespace std;

struct petrolPump

{

int petrol;

int distance;

};

int tour(petrolPump [],int );

int main()

{

int t;

cin>>t;

while(t--)

{

int n;

cin>>n;

petrolPump p[n];

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

cin>>p[i].petrol>>p[i].distance;

cout<<tour(p,n)<<endl;

}

}

// } Driver Code Ends

/\*

The structure of petrolPump is

struct petrolPump

{

int petrol;

int distance;

};\*/

/\*You are required to complete this method\*/

int tour(petrolPump p[],int n)

{

int dis=0;

int pet=0;

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

{

dis+=p[i].distance;

pet+=p[i].petrol;

}

if(dis>pet)

return -1;

int i=0,j=0;

int len=0;

int ans=0;

while(!(i==j && len==n))

{

if(i==j)

{

ans+=(p[j].petrol-p[j].distance);

j++;

j=j%n;

len++;

}

else

{

if(ans<0)

{

ans-=(p[i].petrol-p[i].distance);

i++;

i=i%n;

len--;

}

else

{

ans+=(p[j].petrol-p[j].distance);

j++;

j=j%n;

len++;

}

}

}

return i;

//Your code here

}

**3. LRU Cache**

The task is to design and implement methods of an **LRU cache**. The class has two methods **get**() and **set**() which are defined as follows.  
**get**(x)   : Returns the value of the key **x** if the key exists in the cache otherwise returns **-1.**  
**set**(x,y) : inserts the value if the key **x** is not already present. If the cache reaches its capacity it should invalidate the least recently used item before inserting the new item.  
In the constructor of the class the size of the cache should be intitialized.

**Example 1:**

**Input:**

N = 2

Q = 2

Queries = SET 1 2 GET 1

**Output:** 2

**Explanation:** Cache Size = 2

SET 1 2 GET 1

SET 1 2 : 1 -> 2

GET 1 : Print the value corresponding

to Key 1, ie 2.

**Example 2:**

**Input:**

N = 2

Q = 7

Queries = SET 1 2 SET 2 3 SET 1 5

SET 4 5 SET 6 7 GET 4 GET 1

**Output:** 5 -1

**Explanation:** Cache Size = 2

SET 1 2 SET 2 3 SET 1 5 SET 4 5

SET 6 7 GET 4 GET 1

SET 1 2 : 1 -> 2

SET 2 3 : 1 -> 2, 2 -> 3 (the most

recently used one is kept at the

rightmost position)

SET 1 5 : 2 -> 3, 1 -> 5

SET 4 5 : 1 -> 5, 4 -> 5 (Cache size

is 2, hence we delete the least

recently used key-value pair)

SET 6 7 : 4 -> 5, 6 -> 7

GET 4 : Prints 5

GET 1 : No key-value pair having

key = 1. Hence prints -1.

**Your Task:**  
You only need to **complete**the provided functions **get()**and **set().**

**Expected Time Complexity:**O(1) for both **get()**and **set().**  
**Expected Auxiliary Space:**O(1) for both **get()**and **set().**(though, you may use extra space for cache storage and implementation purposes).

**Constraints:**  
1 <= N <= 1000  
1 <= Q <= 100000  
1 <= x, y <= 1000

CODE:

#include <bits/stdc++.h>

using namespace std;

// } Driver Code Ends

// design the class:

class LRUCache

{

private:

list<pair<int,int>> p;

unordered\_map<int,list<pair<int,int>>::iterator> mp;

int size;

public:

LRUCache(int cap)

{

size=cap;

p.clear();

mp.clear();

// constructor for cache

}

int get(int key)

{

if(mp.find(key)==mp.end())

return -1;

auto it=mp.find(key);

pair<int,int> pi =\*(it->second);

int ans=pi.second;

p.erase(it->second);

p.push\_front(make\_pair(key,ans));

mp[key]=p.begin();

return ans;

// this function should return value corresponding to key

}

void set(int key, int value)

{

if(mp.find(key)==mp.end())

{

if(size==p.size())

{

int last=p.back().first;

p.pop\_back();

mp.erase(last);

}

}

else

{

auto it=mp.find(key);

p.erase(it->second);

}

p.push\_front(make\_pair(key,value));

mp[key]=p.begin();

return;

// storing key, value pair

}

};

// { Driver Code Starts.

int main()

{

int t;

cin >> t;

while (t--)

{

int capacity;

cin >> capacity;

LRUCache \*cache = new LRUCache(capacity);

int queries;

cin >> queries;

while (queries--)

{

string q;

cin >> q;

if (q == "SET")

{

int key;

cin >> key;

int value;

cin >> value;

cache->set(key, value);

}

else

{

int key;

cin >> key;

cout << cache->get(key) << " ";

}

}

cout << endl;

}

return 0;

}