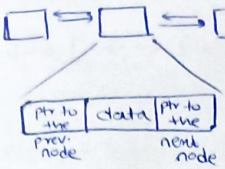
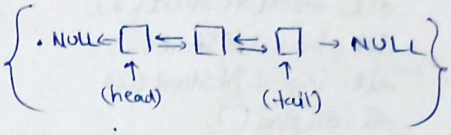
DOUBLY LINKED LIST

- Str:





- Advantages over singly linked list:
  - → Traversal can be done both ways.
  - → Insertion & deletion becomes more efficient
  - → Disadvantage: extra space for previous ptr.

- Implementation of a node in a doubly linked list:

```
class Node{
    public:
    int val;
    Node* prev;
    Node* next;
    Node(int data){
        val=data;
        prev=NULL;
    }   next=NULL;
};
```

(CN)

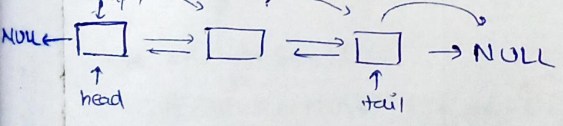
```
class DoublyLL{
    public:
    Node* head;
    Node* tail;
(*) DoublyLL(){
        head=NULL;
        tail=NULL;
    }
};
```

(CN)

```
int main(){
    Node* new_node= new Node(3);
    DoublyLL dll;
    dll.head=new_node;
    dll.tail=new_node;
    cout<<dll.head->val;
    return 0;
}
```

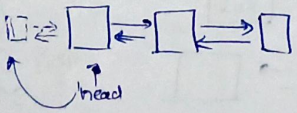
↳ op: 3

- Traversal in a doubly linked list:



(or) (we can start from tail & traverse backwards.)

- Insertion at k^{th} position in a doubly linked list.
  - Add a node at the start:



Code:

```
class DoublyLL{
    (*)
    void insertAtStart(int val){
        Node* new_node = new Node(val);
        if(head==NULL){ //means LL is empty
            head=new_node;
            tail=new_node;
            return;
        }
        new_node->next = head;
        head->prev = new_node;
        head = new_node;
        return;}
```

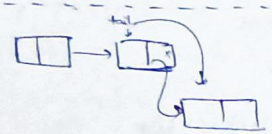
```
    void display(){
        Node* temp = head;
        while(temp!=NULL){
            cout<<temp->val<<"<->";
            temp = temp->next;
        }cout<<endl;
    }
};
```

(All codes in class DoublyLL)

```
int main(){
    DoublyLL dll;
    dll.insertAtStart(1);
    dll.insertAtStart(2);
    dll.insertAtStart(3);
    dll.display();
    return 0;
}
```

o/p: 3 ⟷ 2 ⟷ 1 ⟷

⇒ Add a node at the end:



∴ 1) Create
2) tail→next = val & tail→prev = tail.
3) tail = val

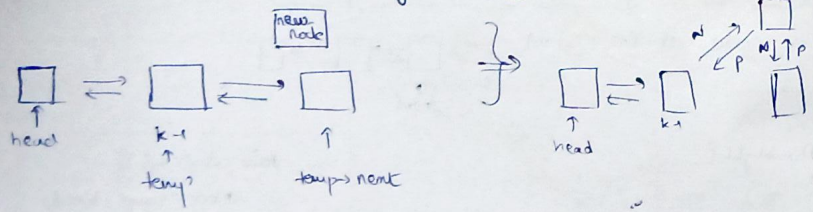
Code:

```
class DoublyLL{
    // cout<<temp->val<<" ";
    void insertAtEnd(int val){
        Node* new_node = new Node(val);
        if(tail==NULL){
            head = new_node;
            tail = new_node;
            return;
        }
        tail->next = new_node;
        new_node->prev = tail;
        tail = new_node;
        return;
    }
};
```

```
int main(){
    dll.insertAtEnd(1);
    dll.insertAtEnd(2);
    dll.insertAtEnd(3);
    dll.display();
}
```

o/p: 1 2 3

⇒ Add a node at an arbitrary position:



Let K=3

Here $T_c = O(k)$ as we need to traverse 'k' elements.

Code:

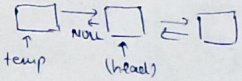
```
class DoublyLL{
    // assuming K < length of linked list
    void insertAtPosition(int val, int k){
        Node* temp = head;
        int count = 1;
        while(count<(k-1)){
            temp = temp->next;
            count++;
        }
        // temp now points to node at (k-1)th position
        Node* new_node = new Node(val);
        new_node->next = temp->next;
        temp->next = new_node;
        new_node->prev = temp;
        new_node->next->prev = new_node;
        return;
    }
};
```

```
int main(){
    dll.insertAtPosition(4,3);
    dll.display();
}
```

o/p: 1 2 4 3

Deletion at Kth position in a doubly linked list.

⇒ Delete a node at the start:



, $T_c = O(1)$

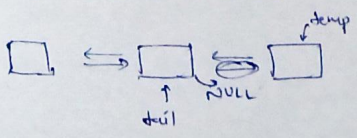
Code:

```
class DoublyLL{
    void deleteAtStart(){
        if(head==NULL) return; // if LL is empty
        Node* temp = head;
        head = head->next;
        if(head==NULL){ // if dLL has only 1 node
            tail=NULL;
        }else{
            head->prev = NULL;
        }
        free(temp);
    }
};
```

```
int main(){
    dll.deleteAtStart();
    dll.display();
}
```

o/p: 2 4 3

⇒ Delete a node at the end:



, $T_c = O(1)$

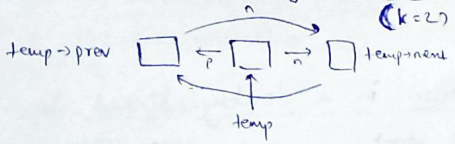
Cont.

```
class DoublyLL {
    Node* head;
    Node* tail;
    int size;
public:
    void deleteAtEnd(){
        if(head==NULL) return;
        Node* temp = tail; tail=tail->prev;
        if(tail==NULL){
            head = NULL;
        }else{
            tail->next = NULL;
        }
        free(temp);
        return;
    }
};
```

```
int main(){
    dll.deleteAtEnd();
    dll.display();
}
```

o/p: 2 4

⇒ Delete node at an arbitrary position:



, $T_c = O(k)$

Code:

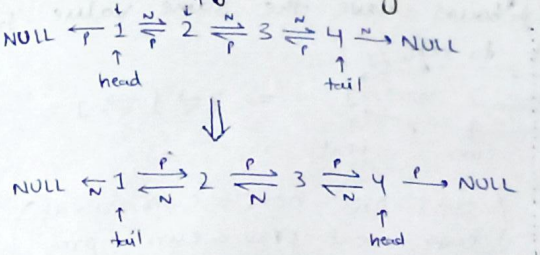
```
class DoublyLL{
    Node* head;
    Node* tail;
    int size;
public:
    void deleteAtPosition(int k){
        //assuming k <= length of dll
        Node* temp = head;
        int counter = 1;
        while(counter < k){
            temp = temp->next;
            counter++;
        }
        //now temp is pointing to node at k'th position
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
        free(temp);
        return;
    }
};
int main(){
    (1); //insert
    (2);
    (3);
    dll.deleteAtPosition(2);
    dll.display();
}
```

o/p: 1 2 3

1 3

Q) Given the head of a doubly linked list, reverse it.

Ex: NULL ⇐ 1 ⇔ 2 ⇔ 3 ⇔ 4 ⇒ NULL



In end → swap(head, tail).

{N→next, P→prev}

curr→next = curr→prev

NULL ⇐ 1 ⇒ 2

curr→prev = nextptr

NULL ⇐ 1 ⇔ 2

curr++; nextptr++;

Code:

passed by reference

```
void reverseDll(Node* &head, Node* &tail){
    Node* currPtr = head;
    while(currPtr){
        Node* nextPtr = currPtr->next;
        currPtr->next = currPtr->prev;
        currPtr->prev = nextPtr;
        currPtr = nextPtr;
    }
    //swapping head & tail pointers
    Node* newhead = tail;
    tail = head;
    head = newhead;
}
```

```
int main(){
    (1);
    (2);
    (3);
    (4);
    dll.display();
    reverseDll(dll.head, dll.tail);
    dll.display();
}
```

o/p: 1 2 3 4

4 3 2 1

Q) Given head of a doubly linked list, find if it's palindrome or not?

Ex: 1 ⇌ 2 ⇌ 3 ⇌ 4 ⇌ 2 ⇌ 1 (head, tail) } we will check if head == tail, then head++; tail--;

Code:

```
bool isPalindrome(Node* head, Node* tail){
    while(head!=tail && tail!=head->prev){
        if(head->val != tail->val){
            return false;
        }
        head = head->next;
        tail = tail->prev;
    }
    return true;
}
```

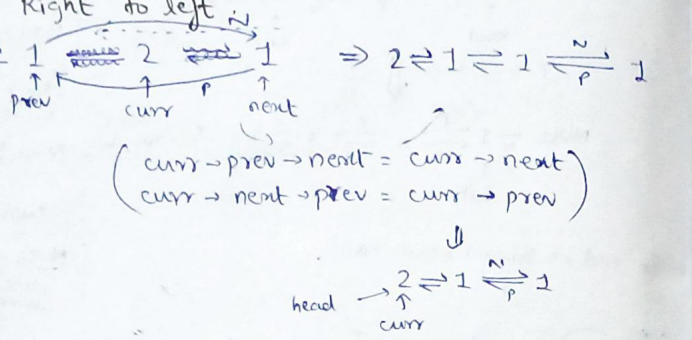
//bool is not written inside 'class'

```
int main(){
    1;
    2;
    3;
    3;
    2;
    1;
    dll.display();
    cout<<isPalindrome(dll.head, dll.tail);
}
```

o/p: 1 2 3 3 2 1

1

Q) Given the head of a doubly linked list, delete the nodes whose neighbours have the same value. Traverse the list from Right to left.

A) 

Code

```
void deleteSameNeigNode(Node* &head, Node* &tail){
    Node* currNode = tail->prev;
    while(currNode != head){
        Node* prevNode = currNode->prev;
        Node* nextNode = currNode->next;
        if(prevNode->val == nextNode->val){
            prevNode->next = nextNode;
            nextNode->prev = prevNode;
            free(currNode);
        }
        currNode = prevNode;
    }
```

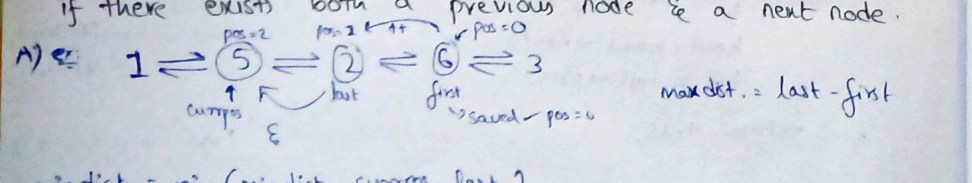
```
int main(){
    2;
    1;
    1;
    2;
    1;
    ---
    deleteSameNeigNode(dll.head, dll.tail);
    dll.display();
}
```

Op: 2 1 1 2 1

2 1 1

Q) A critical point in a linked list is defined either a local maxima or a local minima. Given a linked list head, return an array of length 2 containing [minDistance, maxDistance] where minDistance is the minimum distance between any 2 distinct critical points & maxDistance is the maximum distance between any 2 distinct critical points. If there are fewer than 2 critical points, return [-1,-1].

Note that a node can only be a local maxima/minima if there exists both a previous node & a next node.

A) 

Code:

```
bool isCriticalPoint(Node* &currNode){
    if(currNode->prev->val < currNode->val and currNode->next->val < currNode->val)
    { return true; } // local maxima
    if(currNode->prev->val > currNode->val and currNode->next->val > currNode->val)
    { return true; } // local minima
    return false;
}

vector<int> distBtwCriticalPoints(Node* head, Node* tail){
    vector<int> ans(2, INT_MAX);
    int firstCP = -1, lastCP = -1;
    Node* currNode = tail->prev;
    if(currNode == NULL){
        ans[0] = ans[1] = -1;
        return ans;
    }
    int currPos = 0;
    while(currNode->prev != NULL){
        if(isCriticalPoint(currNode)){
            if(firstCP == -1){
                firstCP = lastCP = currPos;
            }
            else{
                ans[0] = min(ans[0], currPos - lastCP); // min distance
                ans[1] = currPos - firstCP; // max distance
                lastCP = currPos;
            }
        }
        currPos++; // see diagram left side, we are setting pos=0 from right to left.
        currNode = currNode->prev;
    }
    if(ans[0] == INT_MAX){
        ans[0] = ans[1] = -1;
    }
    return ans;
}
```

```
int main(){
    1;
    5;
    4;
    2;
    6;
    3;
    ---
    vector<int> ans = distBtwCriticalPoints(dll.head, dll.tail);
    cout << ans[0] << " " << ans[1];
}
```

Op: 1 5 4 2 6 3

1 3

Q) Given the head of a doubly linked list. The values of the list are sorted in non-decreasing order. Find if there exists a pair of distinct nodes such that the sum of their values is x. Return the pair in the form of a vector [l, r], where l & r are the values stored in the 2 nodes pointed by the pointers. If there are multiple such pairs, return any of them. If there is no such pair return [-1, -1].

A) ex: We did this kind of prob. in arrays (type, target sum)

2 ⇌ 5 ⇌ 6 ⇌ 8 ⇌ 10

head, tail

if x = 11

head + tail > 11 → tail--;

head + tail < 11 → head++;

Code:

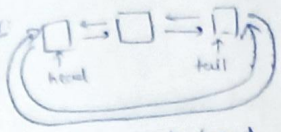
```
vector<int> pairSumDLL(Node* head, Node* tail, int x){
    vector<int> ans(2, -1);
    while(head != tail){
        int sum = head->val + tail->val;
        if(sum == x){
            ans[0] = head->val;
            ans[1] = tail->val;
            return ans;
        }
        if(sum > x){ // need smaller values
            tail = tail->prev;
        }
        else{ // need bigger values
            head = head->next;
        }
    }
    return ans;
}
int main(){
    2;
    5;
    6;
    8;
    10;
    vector<int> ans = pairSumDLL(dll.head, dll.tail, 11);
    cout << ans[0] << " " << ans[1];
}
```

2 5 6 8 10

5 6

CIRCULAR LINKED LIST

- Str:



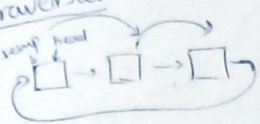
tail->next = head

- Advantages of CLL over singly LL

→ We can start traversing from any node to print all nodes until we reach visited node.

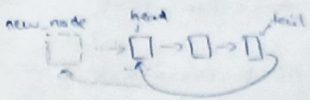
→ When we have traverse linked list in circular fashion multiple times

- Traversal



- Insertion at k^th position in a circular linked list:

⇒ Add a node at the start:



Code:

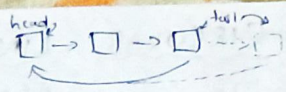
```
class Node{
    public:
    int val;
    Node* next;
    Node(int data){
        val = data;
        next = NULL;
    }
};

int main(){
    CircularLinkedList cll;
    cll.insertAtStart(3);
    cll.insertAtStart(2);
    cll.insertAtStart(1);
    cll.display();
    return 0;
}
```

O/P: 1 -> 2 -> 3 ->

- T_c = O(n)

```
class CircularLinkedList{
    public:
    Node* head;
    CircularLinkedList(){
        head = NULL;
    }
    void display(){
        Node* temp = head;
        do{
            cout << temp->val << " ->";
            temp = temp->next;
        } while(temp != head);
        cout << endl;
    }
    void insertAtStart(int val){
        Node* new_node = new Node(val);
        if(head == NULL){
            head = new_node;
            new_node->next = head;
            return;
        }
        Node* tail = head;
        while(tail->next != head){
            tail = tail->next;
        }
        tail->next = new_node;
        new_node->next = head;
        head = new_node;
    }
```

⇒ Add a node at the end:



Code:

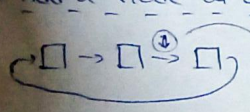
```
void insertAtEnd(int val){
    Node* new_node = new Node(val);
    if (head==NULL){
        head = new_node;
        new_node->next = head; //circular LL
        return;
    }
    Node* tail = head;
    while (tail->next != head){
        tail = tail->next;
    }
    // here tail will be pointing to last node
    tail->next = new_node;
    new_node->next = head;
    tail = new_node;
}
```

```
int main(){
    ...
    cll.insertAtEnd(3);
    cll.display;
    ...
}
```

Op: 1→2→3→8→

• $T_c = O(n)$

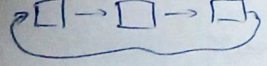
⇒ Add a node at an arbitrary position:



Suppose we want an element here, then we write the same code which we wrote for singly linked list!

Code only changes if we were to insert at start (or) end & that we have already discussed.

⇒ Updation at K^th position in a Circular linked list:



We just need to know how to traverse the 'cll', rest is same as singly LL.

• Deletion at K^th position in a circular linked list:

⇒ Delete a node at the start:

Code:

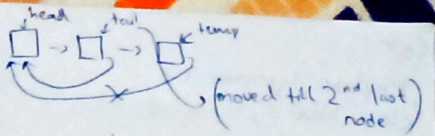
```
void deleteAtStart(){
    if(head==NULL) return;
    Node* temp=head;
    Node* tail = head;
    while(tail->next != head){
        tail = tail->next;
    }
    head = head->next;
    tail->next = head;
    free(temp);
}
```

```
int main(){
    ...
    cll.deleteAtStart();
    cll.display();
    ...
}
```

Op: 2→3→8→

• $T_c = O(n)$

⇒ Delete a node at the end:



• $T_c = O(n)$

Code:

```
void deleteAtEnd(){
    if(head==NULL) return;
    Node* tail = head;
    while(tail->next->next != head){
        tail = tail->next;
    }
    // now tail is pointing to 2nd last node
    Node* temp = tail->next;
    tail->next = head;
    free(temp);
}
```

```
int main(){
    ...
    cll.deleteAtEnd();
    cll.display();
    ...
}
```

Op: 2→3→

⇒ Delete a node at an arbitrary position:

Same as for Singly linked lists!

• Checking if our LL is circular:

```
void printCircular(){
    Node* temp=head;
    for(int i=0; i<15; i++){
        cout<<temp->val<<"->";
        temp=temp->next;
    }cout<<endl;
}
```

(we can choose any no.)

```
int main(){
    ...
    cll.printCircular();
    ...
}
```

Op: 2→3→2→3→2→3→2→3→2→3→2→3→2→3→2→

• Template Classes:

They allow us to pass data type as parameter

Syntax:

```
template <typename T>
class Node{
    T val;
    Node* next;
};
```

(T: name we want to give; val, next: decided at compile time)

```
Node <int> node1;
```

```
class Node{
    int val;
    Node* next;
};
```

```
Node <char> node2;
```

```
class Node{
    char val;
    Node* next;
};
```

```
template<typename T>
class Node {
    public:
    T val;
    Node* next;
    Node(T data){
        val = data;
        next = NULL;
    }
};
```

```
int main(){
    Node<int>* node1 = new Node<int>(3);
    cout<<node1->val<<endl;

    Node<char>* node2 = new Node<char>('a');
    cout<<node2->val<<endl;

    return 0;
}
```

o/p: 3
a

STL: {STANDARD TEMPLATE LIBRARY}

Set of template classes for implementing commonly used data structures & functions.

ex: vector, lists, set, map, queue, stack, ...

We have 3 major components in STL: → containers, → iterators, → algorithms

→ **Containers**: used to hold data of same type.

ex: vector → class that defines a dynamic array.
list → class for doubly linked list
map → used to store unique key-value pairs
set → used to store unique values

→ **Iterators**: → pointer-like entities
↳ used to iterate / traverse the container

ex:
```
vector<int> v = {1,2,3,4,5};
vector<int>::iterator itr;
itr.begin() (→ will give me output) = 1
```

→ **Algorithms**: func.'s that are used to manipulate data in the containers

ex: sort() ⇒ sort(v.begin(), v.end());
min() ⇒ min(a,b) , max() ⇒ max(a,b)

List: → template class in STL for implementing doubly linked list

→ Declaration of a list:

```
#include<list>
list<data_type> list_name;
list<int> roll_no;
list<int> list1{1,2,3};
list<int> list2 = {4,5,6,7};
```
} 2 ways of initialising list

→ Advantages of List in C++ STL: implementation becomes easy.

→ Iterator func.'s:

⇒ list.begin() → returns iterator for the first element

⇒ list.end() → returns iterator for the position after the last element

⇒ list.rbegin() → returns iterator for the first ele. in reverse iteration

ex: {1,2,3,4}

⇒ list.rend() → returns iterator for the position after last element in reverse function.

ex: {1,2,3,4}

⇒ advance(itr,n) → advances the iterator by 'n' places.

ex: {1,2,3,4}
itr

advance(itr, 1)

Code:

```
int main(){
    list<int> l1 = {1,2,3,4};
    auto itr = l1.begin();
    cout<<*itr<<endl;
    auto rev_itr = l1.rbegin();
    cout<<*rev_itr<<endl;
    advance(itr, 2);
    cout<<*itr;
    return 0;
}
```

⇒ allows compiler to automatically deduce the datatypes without mentioning it in the code.

o/p:
1
4
3

→ Traversal in a list:

⇒ //using range-based loop

```
for(auto num: l1){
    cout<<num<<" ";
}
```

↳ 1 2 3 4

⇒ //using iterators

```
for(auto itr = l1.begin(); itr != l1.end(); itr++){
    cout<<*itr<<" ";
}
```

↳ 1 2 3 4

→ //reverse traversal

```
for(auto itr = l1.rbegin(); itr != l1.rend(); itr++)
{ cout<<*itr<<" "; }
```

↳ 4 3 2 1

→ Inserting elements into a list:

⇒ list.insert(itr, value) → insert value before the position of the itr.

⇒ list.insert(itr, count, value) → insert value count no. of times before itr.

Ex. {1,2,4} —l.insert(itr,3)→ {1,2,3,4} (itr changed) —l.insert(itr,2,5)→ {1,2,5,5,3,4}

⇒ list.insert(itr, str_itr, end_itr) → insert values from str_itr to end_itr before itr

Ex. {1,2,3,4} → {1,2,1,2,3,4}

[l, r)

Code:

```
// list = {1,2,3,4}
auto itr = l1.begin();
advance(itr,2);
auto l = l1.begin();
auto r = l1.begin();
advance(r,2);
l1.insert(itr,l,r);
print→
```

O/P: 1 2 1 2 3 4

→ Deleting elements from a list:

⇒ list.erase(itr) → delete the element pointed by the itr

Ex. {1,2,3,4} —l1.erase(itr)→ {1,2,4}

⇒ list.erase(str_itr, end_itr) → delete elements from (str_itr ---- end_itr)

Code:

```
auto s_itr = l1.begin();
advance(s_itr,2);
auto e_itr = l1.begin();
advance(e_itr,4);
l1.erase(s_itr,e_itr);
print→
```

O/P: {1 2 1 2 3 4}

1 2 3 4

→ Other member functions:

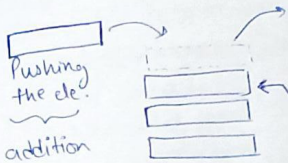
push_front(val): insert val in front of list

pop_front(): removes val from front of list

push_back(val): insert val in the back of the list

pop_back(): removes val from back of the list

STACKS - 1



Our Stack works on LIFO principle!

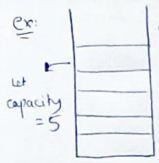
Last In First Out

Diff. properties: push → insert at top, pop → remove from top

isEmpty → checks if stack is empty, isFull

size → gives stack size, top → returns topmost ele

We can create stack by using → array / linked list

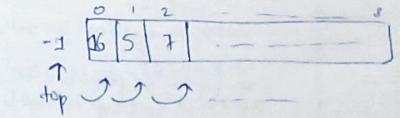


Overflow condn.: When we try to add more elements i.e. when capacity of stack exceeds, overflow condn. occurs.

Underflow condn.: Suppose our stack is empty & we try to 'pop()', then it will give us underflow condn.

Array Implementation of Stacks:

We can initialise our top = -1, to check overflow condn.



push(16) → top++ → arr[top] = data; size = top+1;

isfull() → top == arr.size-1

isempty() → top == -1;

pop() → top--;

Now user can't access its next value. If he wants to access, then value will be overwritten!

Code:

```
class Stack{
    int capacity;
    int* arr;
    int top;
    public:
    Stack(int c){
        this->capacity=c;
        arr= new int[c];
        this->top=-1;
    }
    void push(int data){
        if(this->top==this->capacity-1){
            cout<<"Overflow\n";
            return;
        }
        this->top++;
        this->arr[this->top]=data;
    }
    void pop(){
        if(this->top==-1){
            cout<<"Underflow\n";
            return;
        }
        this->top--;
    }
    int getTop(){
        if(this->top==-1){
            cout<<"Underflow\n";
            return INT16_MIN;
        }
        return this->arr[this->top];
    }
    bool isEmpty(){
        return this->top==-1;
    }
    int size(){
        return this->top+1;
    }
    bool isFull(){
        return this->top==this->capacity-1;
    }
};
```

by default private (capacity, arr, top)

arr declared as pointer because we need to be able to dynamically allocate the array. Size of array is not known at compile time so we need to use a pointer so that we can allocate the array on the heap.

this: keyword that refers to current obj. It's used in member func.'s to access data members & member func.'s of the current object.

* If we don't declare them as private then we will expose the internal state of 'Stack' class to external code. It can cause several issues like:
  - data integrity issues: someone might set our

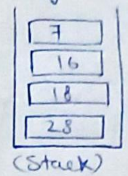
```
int main(){
    Stack st(5);
    st.push(1);
      "  (2);
      "  (3);
      "  (4);
      "  (5);
    cout<<st.getTop()<<endl;
    st.push(6);
    st.pop(); st.pop();
    cout<<st.getTop()<<endl;
```

o/p: 5
Overflow
3

- $T_c = O(1)$ for each operation

Linked List Implementation of Stacks:

7 → 16 → 18 → 28 → NULL ~



Position: ~ LIFO
1
2 We can keep our
3 'top' at head!
4

push() → newnode→next=head;
head=newnode

pop() → newhead=head→next
head→next=NULL
delete --

Code:

```
class Node{
    public:
    Node* next;
    int data;
    Node(int d){
        this->data=d;
        this->next=NULL;
    }
};
int main(){
    //same as prev.
}
```

o/p: 5
Overflow
3

```
class Stack{
    Node* head;
    int capacity;
    int currsize;
    public:
    Stack(int c){
        this->capacity=c;
        this->currsize=0;
        head=NULL;
    }
    bool isEmpty(){return this->head==NULL;}
    bool isFull(){return this->currsize==this->capacity;}

    void push(int data){
        if(this->currsize==this->capacity){
            cout<<"Overflow\n";
            return;
        }
        Node* new_node = new Node(data);
        new_node->next = this->head;
        this->head=new_node;
        this->currsize++;
    }
    int getTop(){
        if(this->head==NULL){
            cout<<"Underflow\n";
            return INT16_MIN;
        }
        return this->head->data;
    }
    int size(){
        return this->currsize;
    }
    int pop(){ //here we are returning value of 'deleted' result!
        if(this->head==NULL){
            cout<<"Underflow\n"; return;}
        Node* new_head = this->head->next;
        this->head->next=NULL;
        Node* tobeRemoved = this->head;
        int result = tobeRemoved->data;
        delete tobeRemoved;
        this->head = new_head;
        return result;}
};
```

```
void pop(){
    (OR)
    Node* temp=this->head;
    free(temp);
    head=new_head;
    return;
}
```

SHORTCUT:

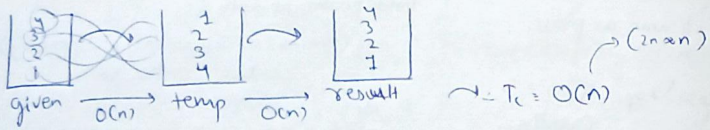
```
#include<iostream>
#include<stack>
using namespace std;
int main(){
    stack<int> st;
    st.push(1);
      :  (2);
      :  (3);
    cout<<st.top()<<endl;
      :  (4);
      :  (6);
    cout<<st.top()<<endl;
    st.pop();
    st.pop();
    cout<<st.top()<<endl;
    cout<<st.empty()<<endl;
    return 0;
}
```

→ o/p: 3 6 3 0

→ variable size

Q) Copy Stack: copy contents of one stack to another in same order.

A) ex:



Code:

```
stack<int> copyStack(stack<int> &input){
    stack<int> temp;
    while(not input.empty()){//doing until input stack becomes empty
        int curr = input.top();
        input.pop();
        temp.push(curr);
    }
    stack<int> result;
    while(not temp.empty()){
        int curr = temp.top();
        temp.pop();
        result.push(curr);
    }
    return result;
}
```

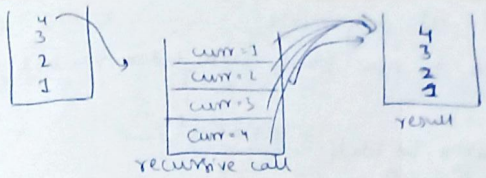
curr → top value is stored & then removed

```
int main(){
    stack<int> st;
    st.push(1);
      :  (2);
      :  (3);
      :  (4);
    stack<int> res = copyStack(st);
    // printing the stack
    while(not res.empty()){
        int curr = res.top();
        res.pop();
        cout<<curr<<"\n";
    }
    return 0;
}
```

o/p: 4 3 2 1

Here we are copying & then printing. But we can also do this recursively!

Like:



Code:

```
void f(stack<int> &st, stack<int> &result){
    if(st.empty()) return;
    int curr = st.top();
    st.pop();
    f(st, result);
    result.push(curr);
}
```

```
int main(){
      :1
      :2
      :3
      :4
    stack<int> res;
    f(st, res);
    while(not res.empty()){
    }
```

o/p: 4 3 2 1

Q) Insert at bottom/any index?

A) • If we want to insert at bottom:

→ once stack is empty, we push our value.

Code:

```
void insertAtBottom(stack<int> &st, int x){
    stack<int> temp;
    while(not st.empty()){
        int curr = st.top();
        st.pop();
        temp.push(curr);
    }
    st.push(x);
    while(not temp.empty()){
        int curr = temp.top();
        temp.pop();
        st.push(curr);
    }
}
```

x → no. to be inserted

- $T_c = O(n)$
- $S_c = O(n)$

Similarly, we can do this recursively!

Code:

```
void f(stack<int> &st, int x){
    if(st.empty()){
        st.push(x);
        return;
    }
    int curr = st.top();
    st.pop();
    f(st, x);
    st.push(curr);
}
```

- If we want to insert at any index;
  ↳ we will move ele. to new stack until index reached, push ele & fill it back

Code:

```
void insertAt(stack<int> &st, int x, int idx){
    stack<int> temp;
    int n = st.size();
    int count = 0;
    while(count < n-idx){
        count++;
        int curr = st.top();
        st.pop();
        temp.push(curr);
    }
    st.push(x);
    while(not temp.empty()){
        int curr = temp.top();
        temp.pop();
        st.push(curr);
    }
}
```

(element; at index you want to enter)

recursively!

Code: (size of org. stack)

```
void f(stack<int>&st, int n, int x, int idx){
    if(idx==n){
        st.push(x);
        return;
    }
    if(st.empty()||idx>n){
        return;
    }
    int curr=st.top();
    st.pop();
    f(st, n, x, idx+1);
    st.push(curr);
}
```

Q) Remove from bottom/any index?

A) · From bottom ~ when size = 1, remove, then refill stack

Code:

```
void removeAtBottom(stack<int>&st){
    stack<int> temp;
    int n=st.size();
    int count=0;
    while(st.size()!=1){
        int curr=st.top();
        st.pop();
        temp.push(curr);
    }
    st.pop();
    while(not temp.empty()){
        int curr=temp.top();
        temp.pop();
        st.push(curr);
    }
}
```

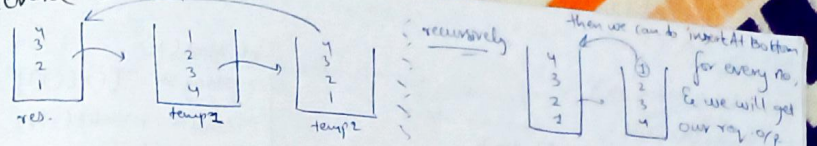
recursive code:

```
void f(stack<int> &st){
    if(st.size()==1){
        st.pop();
        return;
    }
    int curr=st.top();
    st.pop();
    f(st);
    st.push(curr);
}
```

If we want to remove at any index, then we can do it similar to prev. que!

Q) Reverse stack.

A)



Code:

```
void reverse(stack<int> &st){
    stack<int> t1, t2;
    while(not st.empty()){
        int curr=st.top();
        st.pop();
        t1.push(curr);
    }
    while(not t1.empty()){
        int curr=t1.top();
        t1.pop();
        t2.push(curr);
    }
    while(not t2.empty()){
        int curr=t2.top();
        t2.pop();
        st.push(curr);
    }
}
```

recursive code:

```
void insertAtBottom(stack<int>&st, int x){

}
void f(stack<int> &st){
    if(st.empty()) return;
    int curr=st.top();
    st.pop();
    f(st);
    insertAtBottom(st, curr);
}
```

STACKS - IMPORTANT PROBLEMS

BALANCED BRACKET SEQUENCE

Q) Check whether a given bracket sequence is balanced or not?

A) ( ) → parenthesis
{ } → curly braces
[ ] → square bracket

eg: (())() — balanced; eg: ([)] — not balanced

counter ~ how many opening parenthesis are still waiting to get a closed one.

eg: (())(())

counter: 0 1 2 1 2 1 2 1 0

, but it shouldn't be:)(unbalanced

so counter > 0 condn.

b1+b2 → if we concatenate 2 string which are balanced then result will be balanced

(b) ~ if we wrap a balanced string with a pair of bracket() then result is also balanced

eg: ({{()}[]}(()))

on detection, we will check the top ele. of '(' → if yes, then pop.! (stored in stack)

$T_c = O(n)$
$S_c = O(n)$

Code:

```
bool isValid(string str){
    stack<char> st;
    for(int i=0; i<str.size(); i++){
        char ch = str[i]; // current char
        if(ch=='[' or ch=='{' or ch=='('){
            st.push(ch);
        }
        else{ // closing bracket
            if(ch==')' and !st.empty() and st.top()=='('){
                st.pop();
            }
            else if(ch==']' and !st.empty() and st.top()=='['){
                st.pop();
            }
            else if(ch=='}' and !st.empty() and st.top()=='{'){
                st.pop();
            }
            else{
                return false;
            }
        }
    }
    return st.empty();
}
```

```
int main(){
    string str = "[()(())]";
    cout << isValid(str);
    return 0;
}
```

↳ o/p: 1

a → b → c → d

if we needed min. no. of brackets to be removed to make it balance:
- a → change to int
- b → initialise int count = 0;
- c → replace it with: count++;
- d → return count;
- e → change str of int main & cross-check

9) NEXT GREATER ELEMENT

ex: [4,3,9,1,6,8,2] —nge array→ [9,9,-1,6,8,-1,-1] ← o/p array

(if no. ele, > is present on rhs)

Brute force:

A) [4,3,9,1,6,8,2]

i, j

[i+1, n-1]

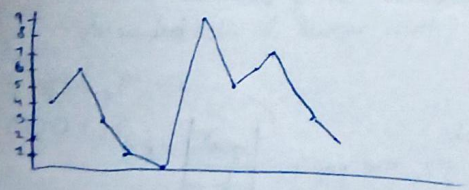
if arr[i] < arr[j] → then arr[i] = arr[j]; i++; break;

else → arr[i] = -1;

∴ Double for loops ↝ $T_c = O(n^2)$

Optimising:

[4,6,3,1,0,9,5,6,7,3,2]



if we notice, → on ↑ing curve → we get our answer quickly

↓ing curve → we have to 'wait' until we get high value.

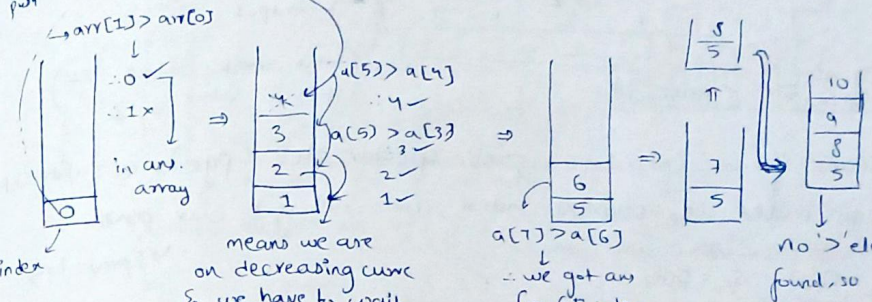
(we can use stack approach) → we use 'index'

ex: [4,6,3,1,0,9,5,6,7,3,2] (0 1 2 3 4 5 6 7 8 9 10)

o/p: [6 9 9 9 9 -1 6 7 -1 -1 -1]

push

arr[1] > arr[0]



- $T_c = O(n)$, $S_c = O(n)$

Code:

```
#include <stack>
    <vector>
    <iostream>
using namespace std;

vector<int> nge(vector<int> &arr){
    int n = arr.size();
    vector<int> output(n, -1);
    stack<int> st;
    st.push(0);
    for(int i=1; i<n; i++){
        while(!st.empty() and arr[i] > arr[st.top()]){
            output[st.top()] = arr[i];
            st.pop();
        }
        st.push(i);
    }
    return output;
}
```

```
int main(){
    int n; cin>>n;
    vector<int> v;
    while(n--){
        int x; cin>>x;
        v.push_back(x);
    }
    vector<int> res = nge(v);
    for(int i=0; i<res.size(); i++){
        cout<<res[i]<<" ";
    }
    return 0;
}
```

indexes ↑ — size 'n' initialised with -1

o/p:
10
4 6 3 1 0 9 5 6 7 3
6 9 9 9 9 -1 6 7 -1 -1

- "if we need → next smaller element" → keep '<' instead of '>' at (a).
- "If we need → prev greater element" (or) "prev smaller element" → reverse arr → get nge/nse → reverse it

Q) STOCK SPAN

Given a series of N daily price quotes for a stock, we need to calculate the span of stock's price for all N days. The span of stock's price in one day is the max. no. of consecutive days (starting from that day & going backward) for which the stock price was less than or equal to the price of that day.

A) eg: [100, 80, 60, 70, 60, 75, 85]

(3-2) (only 1) (5-2) (6-0)

1 1 1 2 1 4 6 → output

Span ~ ≤ price(i)

(total stock value less than i^{th} value)

Means if we calculate prev. greater ele. (pge), & subtract pge index & current index, we will get our ans.

- $T_c = O(n)$, $S_c = O(n)$ ↳ // prev. logic

Code:

```
vector<int> pge(vector<int> &arr){
    int n=arr.size();
    reverse(arr.begin(),arr.end());
    vector<int> output (n,-1);
    stack<int> st;
    st.push(0);
    for(int i=1; i<n; i++){
        while(!st.empty() and arr[i]>arr[st.top()]){
            output[st.top()] = n-i-1;
            st.pop();
        }
        st.push(i);
    }
    reverse(output.begin(),output.end());
    reverse(arr.begin(), arr.end());
    return output
}
int main(){
    int n; cin>>n;
    vector<int> v;
    while(n--){
        int x; cin>>x; v.push_back(x);
    }
    vector<int> res = pge(v);
    for(int i=0; i<res.size(); i++){ cout<<i-res[i]<<" "; }
    return 0;
}
```

85 75 60 70 60 80 100

0 1 3 1 1 0 -1

-1 0 1 1 3 1 0

100 80 60 70 60 75 85

(but after reversing arr it will go to 0th. So we write like this)

→ if = i → for reversed arr.

then our output vector:

85 75 60 70 60 80 100

6 5 3 5 5 6 -1

(means ele. at 6th pos. is > than current ele.)

→ because after reverse, index changes

---

eg:

7

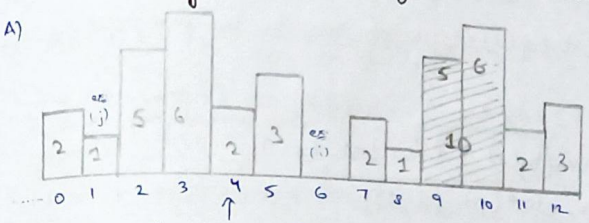
100 80 60 70 60 75 85

1 1 1 2 1 4 6

(0-(-1)) (1-0) (2-1) (3-1) (4-3) (5-1) (6-0)

Q) Given an array of integer heights representing the histogram's bar height where the width of each bar is 1, return the area of largest rectangle in the histogram?

A)



(consider this) ~ we will find → next smaller ele. → $T_c = O(n)$

↳ prev. smaller ele. → $T_c = O(n)$

$\therefore$ Area = height × (i-j-1)

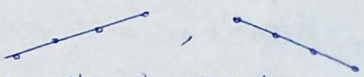
nse: 1 6 4 4 6 6 13 8 13 11 11 13 13

$T_c = O(n)$

$S_c = O(n)$

Method 2: (if we want to solve in single pass)

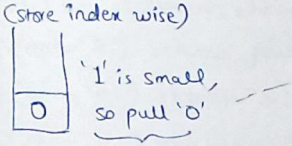
we know ↑ing & ↓ing curve.



On ↓ing, 'nse' we will get, so we can store in stack!

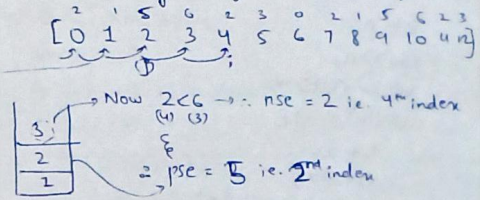
And automatically if for a ele. we get nse, then we will also get 'pse'! We can update our 'ans' → if we get new 'nse, pse'.

(store index wise)



'1' is small, so pull '0'

$(1-(-1)-1) \times 2 = 2$ ans



$\therefore$ ans = $6 \times (4-2-1) = 6$

Now 2nd index = '5'

$\therefore$ nse = 4th index = 2

pse = 1st index = 1

$\therefore$ ans = $5 \times (4-1-1) = 10$

Similarly we can do like this —

$T_c = O(n)$

Method-2:

Code (M2)

```
int histogram(vector<int> &arr){
    int n = arr.size();
    stack<int> st;
    int ans = INT16_MIN;
    st.push(0);
    for(int i=1; i<n; i++){
        while(!st.empty() and arr[i] < arr[st.top()]){
            int ele = arr[st.top()]; //current bar to be removed & whose ans will be calc.
            st.pop();
            int nsi = i;
            int psi = (st.empty()) ? (-1) : st.top();
            ans = max(ans, ele*(nsi-psi-1));
        }
        st.push(i);
    }
    while(not st.empty()){
        int ele = arr[st.top()];
        int nsi = n;
        int psi = st.empty()?(-1):st.top();
        st.pop();
    }
    return ans;
}

int main(){
    int n;
    cin>>n;
    vector<int> v;
    while(n--){
        int x;
        cin>>x;
        v.push_back(x);
    }
    int ans = histogram(v);
    cout<<ans;
    return 0;
}
```

for 'nse'

removing that ele.

if true

'-1' when initial part of stack is empty.

if false

if not removed at first, then it will alter working of "int psi".

not needed

if our stack is present even after traversing whole hist., then we remove ele. one by one!

we can also directly pop. ele.

op:
13
2 1 5 6 2 3 0 2 1 5 6 2 3

10

STACKS_ INTERVIEW QUESTIONS

INFIX EXPRESSIONS: way of writing mathematical expressions in which the operators are placed btw the operands.

ex: → 2+3
→ (4*5)-6
→ 10/2+3)

POSTFIX EXPRESSIONS: mathematical notation in which the operators come after their operands. (Reverse Polish Notation)

ex: ABCD+-+ //operator applied on last 2 operands.

(solving) AB(C+D)-+ (=) ABE-+ (=) A(B-E)+ (=) AF+ (=) A+F (=) G

E F G

PREFIX EXPRESSIONS: Here operands come after their operators (Polish Notation)

ex: +-ABC (solving) +(A-B)C (=) +DC (=) D+C (=) E

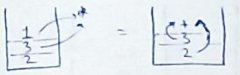
D E

Evaluation of Postfix Expressions: (L → R)

ex: 2 3 1 * + 9 -

→ if no. → store in stack → move to next
↘ if operator → solve prev 2 values ↗

After solving we will be left with only 1 ele., so return st.top().



= 5-9 = -4

- $T_c = O(n)$
- $S_c = O(n)$

Code:

```
int calc(int v1, int v2, char op){
    if(op=='^') return pow(v1,v2);
    if(op=='*') return v1*v2;
    if(op=='/') return v1/v2;
    if(op=='+') return v1+v2;
    if(op=='-') return v1-v2;
}

int main(){
    string str = "231*+9-";
    cout<<eval(str);
    return 0;
}
```

op: -4

```
int eval(string &str){
    stack<int> st;
    for(int i=0; i<str.size(); i++){ (1)
        char ch = str[i];
        if(isdigit(ch)){
            st.push(ch-'0');
        }
        else{
            int v2 = st.top();
            st.pop();
            int v1 = st.top();
            st.pop();
            st.push(calc(v1,v2,ch));
        }
    }
    return st.top();
}
```

we will get no. in ASCII form, so we subtract ASCII of '0' to get original no.

Evaluation of Prefix Expressions: (R → L)

eg: ←
- 9 + * 1 3 2
1 x 3 = 3 → 3 + 2 = 5 → 5 - 9 = -4

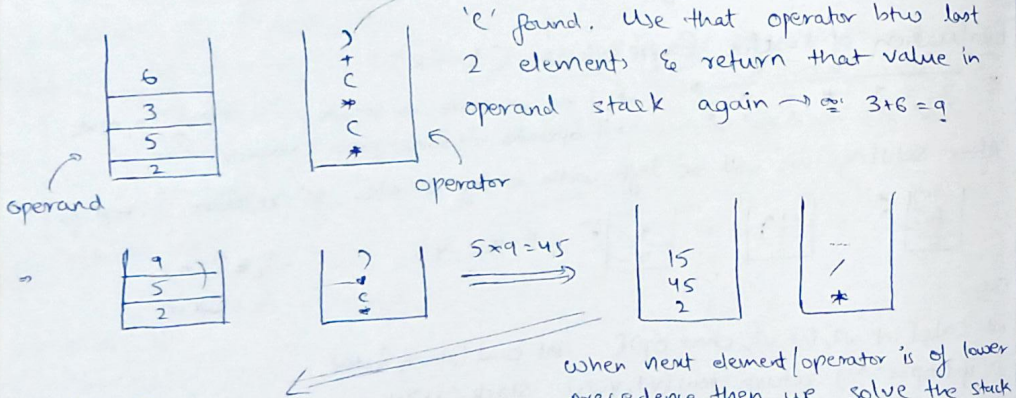
Code:
We need to change prev. code little bit → (only in 'for' loop)

```
for(int i = str.size() - 1; i >= 0; i--){
:
}
```

Evaluation of Infix expressions:

ex: 2 * (5 * (3+6)) / 15 - 2

We maintain 2 stacks



Here $T_c = O(n)$, $S_c = O(n)$

Code:

```
int calc(int v1, int v2, char op){
    if(op=='^') return pow(v1,v2);
    if(op=='*') return v1*v2;
    if(op=='/') return v1/v2;
    if(op=='+') return v1+v2;
    if(op=='%') return v1%v2;
    if(op=='-') return v1-v2;
}

int precedence(char ch){
    if(ch=='^') return 3;
    else if(ch=='*' or ch=='/') return 2;
    else if(ch=='+' or ch=='-') return 1;
    else return -1;
}

int main(){
    string str = "2*(5*(3+6))/15-2";
    cout<<eval(str);
    return 0;
}
```

op: +4

```
int eval(string &str){
    stack<int> num;
    stack<char> ops;
    for(int i=0; i<str.size(); i++){
        if(isdigit(str[i])){
            int operand=0;
            while(i<str.size() and isdigit(str[i])){   // if double digits are there
                operand = operand*10 + (str[i]-'0');
                i++;
            }
            i--;
            num.push(operand);
        }
        else if(str[i]=='('){
            ops.push('(');
        }
        else if(str[i]==')'){
            while(not ops.empty() and ops.top()!='('){
                char op1 = ops.top();
                ops.pop();
                int v2 = num.top();
                num.pop();
                int v1 = num.top();
                num.pop();
                num.push(calc(v1,v2,op1));
            }
            if(not ops.empty()){
                ops.pop();
            }
        }
        else{
            while(!ops.empty() and
                  precedence(ops.top())>=precedence(str[i])){
                char op1 = ops.top(); ops.pop();
                int v2 = num.top(); num.pop();
                int v1 = num.top(); num.pop();
                num.push(calc(v1,v2,op1));
            }
            ops.push(str[i]);
        }
    }
    while(not ops.empty()){   // if still few ele. left in stack
        char op1 = ops.top(); ops.pop();
        int v2 = num.top(); num.pop();
        int v1 = num.top(); num.pop();
        num.push(calc(v1,v2,op1));
    }
    return num.top();
}
```

Conversion of a Prefix expression to a Postfix expression

eg: *+32-15 → Prefix → (3+2)*(1-5)

32+15-* → Postfix → (3+2)*(1-5)

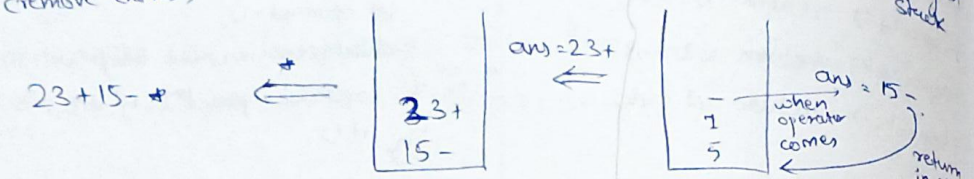
eg: *+32-15

reverse

51-23+*

→ storing in stack

(remove entries)



→ because in postfix, operators come at end, so just to make it easy, we did this.

Code:

```
string eval(string &pre){
    stack<string> st;
    reverse(pre.begin(), pre.end());
    for(int i=0; i<pre.size(); i++){
        if(isdigit(pre[i])){ → character
            st.push(to_string(pre[i]-'0'));
        }
        else{
            string v1 = st.top();
            st.pop();
            string v2 = st.top();
            st.pop();
            string newexp = v1+v2+pre[i];
            st.push(newexp);
        }
    }
    return st.top();
}
```

```
int main(){
    String str = "*+32-15";
    cout<<eval(str);
    return 0;
}
```

∴ o/p → 32+15-*

(1) Design a stack that supports push, pop, top & retrieving the minimum element in const. time.

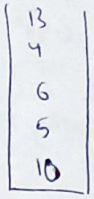
Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on top of the stack
- int top() gets the top element of the stack
- int getMin() retrieves the min. element in the stack

You must implement a soln. with O(1) time complexity for each func.

Assumption: Methods pop, top & getMin operations will always be called on non-empty stacks.

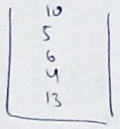
A) (1) eg: push(10), push(5), push(6), push(4), push(13)



Brute force

We call elements one by one & find out min.

(initially = +∞)



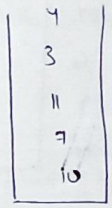
min = ~~∞~~ 13 4

∴ Time = O(n) } for any query

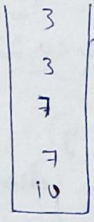
Space = O(n)

(2)

eg: push(10), push(-1), push(11), push(3), push(4)



Now in parallel, calculate min. while pushing ele.'s & store it in another stack



→ stack top

min. of all ele. stored in the stack.

So now if we pop, we must simultaneously pop from min. stack also, to get min. of ele. present in original stack

∴ Space = O(n)

Time = O(1)

(3)

Lets assume we are getting only (+)ve elements

x → incoming value to be added

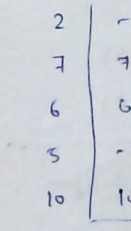
if x < min → we update 'min' to x

x - min = y

new, old, (-)ve no.

{old min. = x - y}

eg:



min: ~~10~~ 5 2

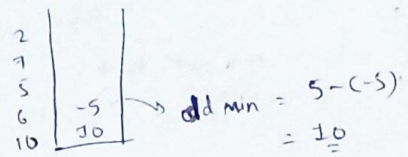
∴ If we pop now ~ ∴ y = -3

↳ old min = 2 - (-3)

= 5

min. = 10 ~~5~~ x

If value is not (-)ve, then we can simply pop it out



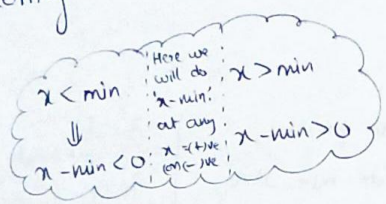
old min = 5 - (-5)

= 10

But here we have a constraint of considering only (+)ve no.s!

(4)

Considering x = (+)ve & (-)ve !



(x - (old min.) = y)

Now we pop. elements one by one!

eg:

(x)	(y)	
-5	-4	-5 - (-1)
-1	-6	-1 - 5
12	7	12 - 5
6	1	6 - 5
5	-5	5 - 10
10	10	

min. = ~~10~~ ~~5~~ ~~-1~~ ~~-5~~ x 5 10

pop() ~ -5 - (-4)
= -5 + 4
= -1 (min.)
] -5 -4

pop() ~ -1 - (-6)
= +5 (min.)
] -1 -6

pop() → 12 (7) (+)ve so we can pop it directly

pop() → 6 1

pop() ~ 5 - (-5)
= +10 (min.)
] 5 -5

∴ 'x - min' ✓

∴ $T_c = O(1)$

$S_c = O(1)$

Shortcut

#define — short form — original form

→ Now we can use shortform in our code rather than writing long code!

Code:

```
#include<bits/stdc++.h>
#define ll long long int
using namespace std;
class MinStack{
public:
    stack<ll> st;
    ll mn;
    MinStack(){
        this->mn = INT_MAX;
    }
    void push(int val){
        if(this->st.empty()){
            this->mn = val;
            this->st.push(val - this->mn);
        }else{
            this->st.push(val - this->mn);
            if(this->mn > val){
                this->mn = val;
            }
        }
    }
    void pop(){
        if(!this->st.empty()){
            if(this->st.top()>=0){
                this->st.pop();
            }else{
                this->mn = this->mn - this->st.top();
                this->st.pop();
            }
        }
    }
    int getMin(){
        return this->mn;
    }
    int top(){
        if(this->st.size()==1){
            return this->st.top();
        }else if(this->st.top()<0){
            return this->mn;
        }else{
            return this->mn + this->st.top();
        }
    }
};
int main(){
    MinStack s;
    s.push(10);
    s.push(5);
    s.push(6);
    s.push(12);
    s.push(-1);
    s.push(5);
    cout<<s.getMin()<<endl;
    s.pop();
    cout<<s.getMin()<<endl;
    s.pop();s.pop();s.pop();
    cout<<s.getMin()<<endl;
    return 0;
}
```

o/p: -5
-1
5

QUEUES - 1 - BASICS, STL AND IMPLEMENTATION

Based on FIFO or FCFS principle.

(first in first out) (first come first serve)

→ Queue is a data structure that supports FIFO or FCFS

→ Queue is a linear data structure

Types of Operation on Queues:

1) Enqueue: this operation will help us to add a new element in the queue.

2) Dequeue: this operation will help us to remove a new element in the queue

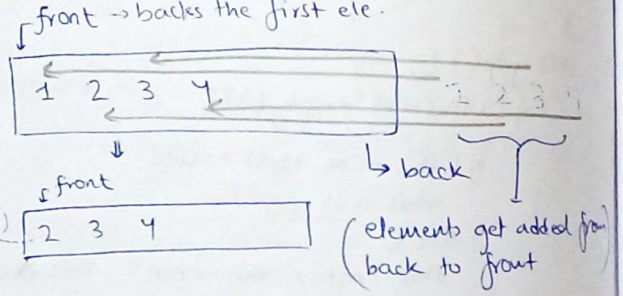
3) isFull: checks whether queue is full or not.

4) isEmpty: checks whether queue is empty or not.

5) front: gives us the element which came first.

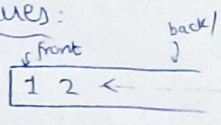
6) size

Ex. enqueue(1)
enqueue(2)
enqueue(3)
enqueue(4)
dequeue();



Types of Queues:

1) SIMPLE:



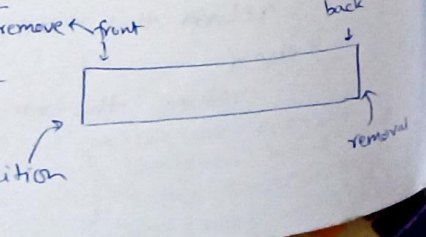
; Here element is based on time. If it came first, then it will be first.. (like that)

2) PRIORITY QUEUE: Queue based on some priority. Ex: We will dequeue an element which is largest in the queue.. (like that)

3) CIRCULAR QUEUE:

4) DOUBLE ENDED QUEUE / DEQUE:

Here we can add/remove from both front & back



Linked list Implementation of Queues:

ex: (front) head → 1 → 2 → 3 ← tail → NULL

• Space efficient (advantage)

enque → add at tail

deque → remove from head : front = head → data;

code:

```
class Node{
  public:
  int data;
  Node* next;
  Node(int data){
     this->data = data;
     this->next = NULL;
  }
};

int main(){
  Queue qu;
  qu.enqueue(10);
  qu.enqueue(20);
  qu.enqueue(30);
  qu.dequeue();
  qu.enqueue(40);
  while(not qu.isEmpty()){
     cout<< qu.front() << " ";
     qu.dequeue();
  }
  return 0;
}
```

o/p: 20 30 40

```
class Queue{
   Node* head;
   Node* tail;
   int size;
   public:
   Queue(){
      this->head = NULL;
      this->tail = NULL;
      this->size = 0;
   }
   void enqueue(int data){ // O(1) = Tc
      Node* newNode = new Node(data);
      if(this->head == NULL){ //LL is empty
         this->head = this->tail = newNode;
      }
      else{
         this->tail->next = newNode;
         this->tail = newNode;
      }
      this->size++;
   }
   void dequeue(){
      if(this->head == NULL) return; //LL is empty
      else{
         Node* oldHead = this->head;
         Node* newHead = this->head->next;
         this->head = newHead;
         if(this->head == NULL) this->tail = NULL;
         oldHead->next = NULL;
         delete oldHead;
      }
      this->size--;
   }
   int getSize(){
      return this->size;
   }
   bool isEmpty(){
      return this->head == NULL;
   }
   int front(){
      if(this->head == NULL) return -1;
      return this->head->data;
   }
};
```

Array Implementation of Queue

front → -1

10	20	30	40	50	60	70	80	90

back → 8

→ vector → v

v.push_back(x) (or) v.back() → inbuilt func.

Adding 1st ele. → front++, back++

(prev. ele. will be like that only, so memory wasted)

Adding ele's → back++ , removing ele.'s → front++

Code:

```
class Queue{
    int front;
    int back;
    vector<int> v;
    public:
    Queue(){
        this->back = -1;
        this->front = -1;
    }
    void enqueue(int data){
        this->v.push_back(data);
        this->back++;
        if(this->back == 0) this->front = 0;
    }                  ↳ when first ele. added
    void dequeue(){
        if(this->front == this->back){
            this->front = -1;
            this->back = -1;
            this->v.clear();
        }
        else this->front++;
    }
    int getfront(){
        if(this->front == -1) return -1;
        return this->v[this->front];
    }
    bool isEmpty(){
        return this->front == -1;
    }
};
```

```
int main(){
    Queue qu;
    qu.enqueue(10);
    qu.enqueue(20);
    qu.enqueue(30);
    qu.dequeue();
    qu.enqueue(40);
    qu.dequeue();
    qu.enqueue(400);
    while(not qu.isEmpty()){
        cout<<qu.getfront()<<" ";
        qu.dequeue();
    }
    return 0;
}
```

↳ o/p: 30 40 400

SHORTCUT:

We have inbuilt func. 'queue' in C++.

Code:

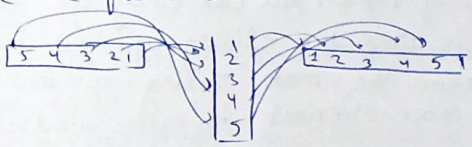
```
int main(){
    queue<int> qu;
    qu.push(10); //enqueue
    qu.push(20);
    qu.push(30);
    qu.push(40);
    qu.pop(); //dequeue
    while(not qu.empty()){
        cout<<qu.front()<<" ";
        qu.pop();
    }
    return 0;
}
```

↳ o/p: 20 30 40

Q) Reverse the elements of a queue?

A) ex: 5 4 3 2 1 → If we try to place in queue & again place in prev. queue (like we did in stack), it won't be useful here as order will remain same.

∴ we use queue - stack



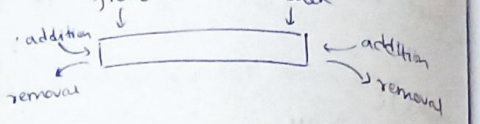
∴ Code:

```
int main(){
    queue<int> input;
    input.push(10);
    input.push(20);
    input.push(30);
    input.push(40);
    stack<int> st;
    while(not input.empty()){
        st.push(input.front()); //input in stack
        input.pop();
    }
    while(not st.empty()){
        input.push(st.top()); //input in queue
        st.pop();
    }
    while(not input.empty()){
        cout<<input.front()<<" ";
        input.pop();
    }
    return 0;
}
```

↳ o/p: 40 30 20 10

QUEUES - INTERVIEW QUESTIONS

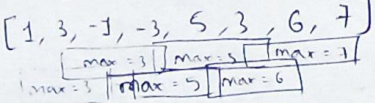
Deque: (double ended queue)
⇓
(we can add elements from both sides)



ex:
```
#include<deque>
:
int main(){
  deque<int> dq;
  dq.push_back(10);
        "  (20);
        "  (30);
  dq.push_front(0);
        "  (33);
  while(!dq.empty()){
    cout<<dq.front()<<" ";
    dq.pop_front();
  }
  return 0;
}
```
o/p: 33 0 10 20

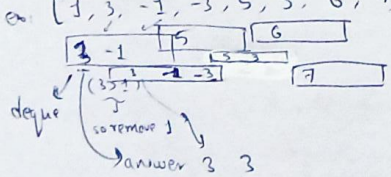
Q) You are given an array of integers nums, there is a sliding window of size k which is moving from very left of the array to the very right. You can only see the k no.s in the window. Each time the sliding window moves right by one position. Return the max sliding window which basically contains the max element in each window?

A) ex: [1, 3, -1, -3, 5, 3, 6, 7] , k = 3 (let)



Now as window moving, ele. is getting removed from front & getting added from back

ex: [1, 3, -1, -3, 5, 3, 6, 7]



∴ T_c = To add: O(1), To remove: O(1)

(if no. goes in ↓ order, we add to deque)
(if big no. found, we remove smaller elements)

Code:
```
vector<int> maxSlideWindow(vector<int>&arr, int k){
  deque<int> dq;
  vector<int> res;
  for(int i=0; i<k; i++){
    while(!dq.empty() and arr[dq.back()]<arr[i]){
      dq.pop_back();
    }
    dq.push_back(i);
  }
  res.push_back(arr[dq.front()]);
  for(i=k; i<arr.size(); i++){
    int curr = arr[i];
    if(dq.front()==(i-k)) dq.pop_front();
    while(!dq.empty() and arr[dq.back()]<arr[i]){
      dq.pop_back();
    }
    dq.push_back(i);
    res.push_back(arr[dq.front()]);
  }
  return res;
};

int main(){
  vector<int> arr={1,3,-1,-3,5,3,6,7};
  int k=3;
  vector<int> result = maxSlideWindow(arr,k);
  for(int i=0; i<result.size(); i++){
    cout<<result[i]<<" ";
  }
  return 0;
}
```
o/p: 3 3 5 5 6 7

Q) We are given a stack data str. with push & pop operations, the task is to implement a queue using instances of stack data structure & operations on them?

A) This que.: (stack → queue) ; here we have 2 variations,
(lifo) (fifo)

→ Pop efficient (dequeue) → O(1), push(enqueue) → can be bad
→ Push efficient (enqueue) → O(1), pop(dequeue) → can be bad.

Push efficient: → We push into stack & queue simul.
→ for pop: → we store all queue ele. in temp que until 1 ele. is left
↳ pop that 1 ele.
↳ copy from temp que. to org. queue

Code:

```
class Queue{
    stack<int> st;
public:
    Queue(){}
    void push(int x){ //queue.enqueue
        this->st.push(x); // O(1)
    }
    void pop(){ //queue.dequeue
        if(this->st.empty()) return;
                                          // O(n)
        stack<int> temp;
        while(this->st.size()>1){ ①
            temp.push(st.top());
            st.pop();
        }
        this->st.pop(); // now stack size=1 & we are at bottom ele.
        while(!temp.empty()){ ②
            this->st.push(temp.top());
            temp.pop();
        }
    }
    bool empty(){
        return this->st.empty();
    }
    int front(){ // O(n)
        ①
        int result = this->st.top();
        ②
        return result;
    }
};
```

```
int main(){
    Queue qu;
    qu.push(10); //enqueue
    qu.push(20);
    qu.push(30);
    qu.push(40);
    qu.pop(); // dequeue
    while(!qu.empty()){
        cout<<qu.front()<<" ";
        qu.pop();
    }
    return 0;
}
```

o/p: 20 30 40

Pop efficient: We need → 10 → [10] → 20 → [10 20] → 30 → [10 20 30] ← front/top

→ Here we are pushing: add at bottom!
→ So when new ele. comes all prev ele. go to temp stack, then new ele. added.
→ Again elements brought back from temp to org. stack
→ We can directly pop!

∴ Push → O(n), Pop → $T_c = O(1)$, Front → $T_c = O(1)$

Code:

```
class Queue{
    stack<int> st;
public:
    Queue(){}
    void push(int x){
        stack<int> temp;
        while(!this->st.empty()){
            temp.push(this->st.top());
            this->st.pop();
        }
        this->st.push(x);
        while(!temp.empty()){
            this->st.push(temp.top());
            temp.pop();
        }
    }
    void pop(){
        if(this->st.empty()) return;
        this->st.pop();
    }
    bool empty(){
        return this->st.empty();
    }
    int front(){
        if(this->st.empty()) return INT_MIN;
        return this->st.top();
    }
};
```

```
int main(){
    | same as prev one
}
```

o/p: 20 30 40

Q) Implement a stack using instances of queue data structure & operations on it?
A) This que. (Queue → stack); Here we have → push efficient / pop efficient

Pop efficient: → Make 2 queues
→ Pushing in 1 queue → transfer to other queue (element by element)
→ Exchange/Swap queues
→ for pop → we can remove from top of 2nd queue

Push efficient: → Add in 1st queue → transfer all ele. in 2nd queue until 1 ele. left
↓
pop it
↓
swap names.

Push efficient Code:

```
class Stack{
    queue<int> q1,q2;
public:
void push(int x){q1.push(x);} // Tc=O(1)
void pop(){ // Tc=O(n)
    if(q1.empty()) return;
    ① while(q1.size()!=1){          leave 1 ele. in q1 & push others in q2
        q2.push(q1.front());
        q1.pop();
      }
    q1.pop(); //pop the only element left in q1
    ② queue<int>q=q1;               } swap names of queues
      q1=q2;
      q2=q;
}
int size(){return q1.size();}
int top(){ // Tc=O(n)
    if(q1.empty()) return -1;
        ①
    int temp=q1.front(); //last pushed element
    q1.pop(); //emptying queue
    q2.push(temp); //pushing last ele. to q2
        ② ⟶ swaping names
    return temp;
}
};
```

```
int main(){
    Stack S;
    S.push(10);
      "  (20);
      "  (30);
    cout<<s.top()<<endl;
    S.pop();
    S.pop();
    cout<<s.top();
    return 0;
}
```

op: 30, 10

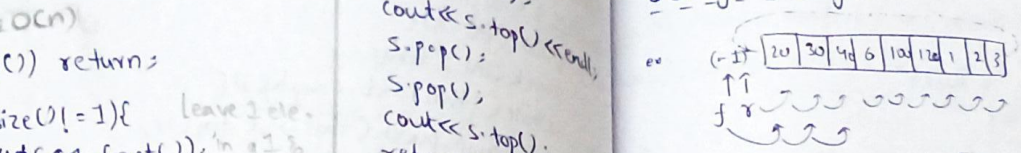
Pull efficient Code:

```
class Stack{
    queue<int> q1,q2;
public:
void push(int x){ // Tc=O(n)
    q2.push(x); //push 'x' in empty q2
    while(!q1.empty()){          } push all ele.'s in q1 to q2
        q2.push(q1.front());
        q1.pop();
    queue<int> q=q1;             } swap names
    q1=q2;
    q2=q;
}
void pop(){ // Tc=O(1)
    if(q1.empty()) return;
    q1.pop();
}
int size(){return q1.size();}
int top(){
    if(q1.empty()){
        return -1;
    }
    return q1.front();
}
};
int main(){
    : same as prev one
    :
}
```

op: 30, 10

Circular Queue:

Using Arrays:



→ When element is added ⇒ r++

→ When element is removed ⇒ f++

[0, n-1]
- → Once 'r' is at end then we need to go to start ⇒ we do (r+1)%n
- → Similarly we do ⇒ (f+1)%n
- → (or) we can keep 'r' at last index & 'f' at 0 ⇒ (r+1)%n = 0

Code:

```
class Queue{
    int front;
    int back;
    vector<int> v;
    int cs;
    int ts;
public:
    Queue(int n){
        v.resize(n);
        this->back=n-1;
        this->front=0;
        this->cs=0;
        this->ts=n;
    }
    bool isEmpty(){
        return this->cs==0;
    }
    bool isFull(){
        return this->cs==this->ts;
    }
    int getfront(){ if(isEmpty()) return -1;
        return this->v[this->front];
    }
    void enqueue(int data){
        if(isFull()) return;
        this->back=(this->back+1)%this->ts;
        this->v[this->back]=data;
        this->cs++;
    }
    void dequeue(){
        if(isEmpty()) return;
        else this->front=(this->front+1)%this->ts;
        this->cs--;
    }
};
```

```
int main(){
    Queue qu(3);
    qu.enqueue(10);
      "    (20);
      "    (30);
    qu.dequeue();
    qu.enqueue(40);
    qu.dequeue();

    cout<<qu.getfront()<<endl;
    while(!qu.isEmpty()){
        cout<<qu.getfront()<<" ";
        qu.dequeue;
    }
    return 0;
}
```

op: 40, 40

Above code is similar to array implementation of queues.

SET C++ IN 1 SHOT - STL AND TYPES

What is a Set?

→ STL container used to store unique values.
→ Values are stored in ordered state (↑ order/↓ order)
→ No indexing, elements are identified by their own values
→ Once value is inserted in a set, it can't be modified

Advantages of Set

→ Unique values → Ordered → Set = dynamic size → (so no overflow errors)
→ faster — insertion, deletion, search → $T_c = O(\log N)$ (binary search tree)
↳ (internally = binary search)

Disadvantages of Set

→ can't access elements using indexing
→ more memory than array
→ not suitable for large datasets

Declaration of a Set

```
#include<set>
set<data_type> set_name;
//initialising set
set<int> set1 = {1,3,2,4}; // * By default, values are stored in ↑ order in set.
```

If we want to store in ↓ order : set<data_type, greater<data_type>> (denotes ↓ order)

Inserting elements into a set

set_name.insert(value); ⇝ $T_c = O(\log N)$
ex: set1.insert(4);
set1.insert(3); ⇝ internally it will be stored in order

* returns an iterator (like pointer) to the inserted value.

Code

```
set<int> s1;
s1.insert(3);
      (2);
      (1);
cout<<s1.size()<<endl;
s1.insert(3);
```

o/p: 3 // size remains same as
3 // duplicate value is added
// & sets only contain unique

Traversal of a Set

We use iterator for traversing a set

⇒ set_name.begin() → iterator pointing to the first ele. of my set
⇒ set_name.end() → iterator pointing to the position after the last element of a set

ex: [1,2,3] (begin() at 1, end() after 3)
(here s1 = int, so value also int datatype)

(* 'auto' keyword automatically detects datatype of value)

Code

①
```
for(auto value: s1){
    cout<<value<<" ";
}
```
o/p: 1 2 3

②
```
set<int>::iterator itr;
for(itr=s1.begin(); itr!=s1.end(); itr++){
    cout<<*itr<<" ";
}
```
o/p: 1 2 3 (Here itr is giving position of an element)

(we can either create 'itr' specifically or mention 'auto' in for loop)

③
```
for(auto itr = s1.begin(); itr!=s1.end(); itr++){
    cout<<*itr<<" ";
}
```
o/p: 1,2,3

If we had mentioned initially ⇝ set<int, greater<int>> s1;
Then on traversal ⇝ we will get o/p: 3 2 1

Deleting elements from a Set

1) set_name.erase(value); ⇝ ex: [1,2,3] ↳ s1.erase(2)
2) set_name.erase(position); ⇝ ex: [1,2,3] (itr → we have itr pointing this) ↳ s1.erase(itr)
3) set_name.erase(start_pos, end_pos); ⇝ ex: [1,2,3,4,5] → [start, end)^e → [2,3] → deleted

$T_c = O(\log N)$ (1, 2) ; $T_c = O(N)$ (3)

Code: ex: [1,2,3,4,5]

①
```
s1.erase(4);
```
<for-each loop>
o/p: 1 2 3 5

②
```
auto itr = s1.begin();
advance(itr, 3);
s1.erase(itr);
```
moves itr by 3 positions
ex: 1 2 3 4 5 (itr +3 → new)
<for-each loop>
o/p: 1 2 3 5

③
```
auto s_itr = s1.begin();
s_itr++; //2
auto e_itr = s1.begin();
advance(e_itr, 3); //4
s1.erase(s_itr, e_itr);
```
<for-each loop>
o/p: 1 4 5

Member functions of a Set container

⇒ size() : gives no. of elements present in a set

⇒ max_size() : max no. of ele.'s set can hold

⇒ empty() : returns True if empty, else false

⇒ clear() : removes all elements from set

⇒ find() : returns position of element if present, else returns end

Code:
```
if(s1.find(4) != s1.end) cout<<'1': } → o/p: 1
else cout<<'0';
```

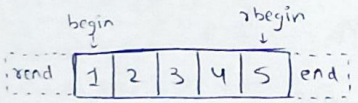
⇒ count() : returns no. of occurences of an element → Present → 1, absent → 0

⇒ lower_bound() : returns element if present, else returns just greater value.

ex: [10,20,30,40]
s.lower_bound(25) → o/p: 30

⇒ upper_bound() : returns the next greater value

ex: [10,20,30,40]
s.upper_bound(20) → o/p: 30



rbegin ⇒ return iterator to first ele. of set in reverse order

rend ⇒ return iterator to position after last element in reverse order.

Q) Cherry's Birthday is coming & Aashi is a random guy that keeps coming up with names of people randomly to add to invite list, even if name already on the list. Find final list that contains name without any repetition.

i/p: 1st line → no. of names that Aashi pops up with
o/p: o/p final list sorted lexographically in a new line.

A)
```
int main(){
    set<string> invitelist;
    int n; cin>>n;
    while(n--){
        string name; cin>>name;
        invitelist.insert(name);
    } cout<<"Printing invite list"<<endl;
    for(auto name: invitelist){
        cout<<name<<endl;
    } return 0;
}
```
// we use set to get all the i/p's only once & by default it's sorted

Q) Given 2 vectors v1 & v2. Find out common ele.'s btw the 2 & return the sum of them.

i/p: v1 = {1,1,2,3,3,3}
v2 = {5,6,7,5,2,3,6}
o/p: 5

Explanation: Values common are: 2,3
So, sum is 2+3 = 5

A) So, we can convert vector → set (to avoid repetition) & then compare ele.'s of set!

∴ $T_c = O(n \log t + m \log t) = O((n+m)\log t)$

(ele. in v1) (time to insert ele. from v1 to set of size t) (ele. in v2) (searching all ele of set in v2)

Code:
```
int main(){
    int n,m; cin>>n>>m;
    vector<int> v1(n); vector<int> v2(m);
    for(int i=0;i<n;i++) cin>>v1[i];
    for(int i=0;i<m;i++) cin>>v2[i];
    int ans=0;
    set<int> s1;
    for(auto ele: v1) s1.insert(ele);
    for(auto ele: v2){
        if(s1.find(ele) != s1.end()){ // checking if ele. of v2 is there in v1 or not!
            ans+=ele;
        }
    }
    cout<<"Ans is "<<ans<<endl;
    return 0;
}
```

Q) Check if the string has all english alphabets. Given a string, check if it has all english alphabets from a-z.

i/p 1: PW → o/p 1: NO
i/p 2: abcdEFGHIJklmnopQRSTUVWXYZ → o/p 2: Yes

Explanation: i/p 2 doesn't have all alphabets, hence NO.
i/p 1 has all alphabets irrespective of upper/lower case, so Yes!

A) Brute force: Checking each alphabet in whole string ∴ $T_c = O(26 \times n)$

Set soln.: If length < 26 → ✗
& we convert all to either lower/uppercase for simplicity → set → 26 ✓ / No

Code:
```
bool checkAlpha(string s){
    if(s.length() < 26) return false;
    transform(s.begin(), s.end(), s.begin(), ::tolower);
    set<char> s1;
    for(auto ch: s) s1.insert(ch);
    return (s1.size()==26);
}
```

```
int main(){
    string st;
    cin>>st;
    if(checkAlpha(st)){
        cout<<"True";
    }
    else cout<<"False";
    return 0;
}
```

'transform' keyword (_ , _ , _ , _)

- from starting position
- to ending position
- o/p iterator (where to store)
- (binary operation, type of operation)

Unordered_set :

→ values are stored in unordered fashion
→ unique values, values will be identified by itself, values cannot be modified inside set

→ Insertion, Deletion, Search → $T_c = O(1)$ (avg. time complexity) ~(using Hashing)

→ Code:
```
#include<unordered_set>
int main(){
unordered_set<int> s1;
s1.insert(1);
" (1);
" (1);
" (1);
for(auto val:s1)
cout<<val;
cout<<endl;
}
```
o/p: 2 1

Member function of unordered_set

⇒ insert() : single element → O(1) avg
↳ O(N) worst (size ≥ capacity) (then rehashing)

multiple element → O(N) avg → no. of ele.'s being inserted
↳ O(N(N+1)) worst (rehashing) ↳ size of unordered set

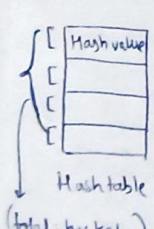
⇒ Deletion : erase(value), erase(iterator), erase(star_itr, end_itr)
→ single ele. → O(1) avg / O(N) worst
→ Multiple ele. → O(N) avg

⇒ find() : O(1) → avg, O(N) → worst

⇒ count() : O(1) → avg, O(N) → worst case

Here 'O' related to time complexity

⇒ Hashing:
⇒ Load factor : load_factor() → size of unordered set / bucket_count



⇒ rehash(x): sets the no. of bucket to 'x' or more
↳ O(N) → avg, O(N²) → worst

⇒ Rest all func.'s : size(), max_size(), empty(), clear(), begin(), end()
work in same way as an ordered set does

Multiset :

→ it can store duplicate values
→ ordered (asc./desc. order)
→ value will be identified by itself
→ value cannot be modified in multiset
→ Code: #include<set>

```
{ multiset<int> ms;   ~ if we want in descending
                        ↳ <int, greater<int>>
  (rest all same)
}
```

Member functions of Multiset :

⇒ insert() : $T_c = O(\log N)$
⇒ Deletion: erase(value), erase(position) } $T_c = O(\log N)$
erase(start_pos, end_pos) } $T_c = O(N)$

if (1,2,2,3,3) erase(2); then both val of '2' will be erased

⇒ find() : returns lowerbound of element searched if found, else end iterator

ex: 1 2 2 2 3 3 (2 pointed) ms.find(2); $T_c = O(\log N)$

⇒ count() : no. of occurences
$T_c = O(K + \log N)$ (K: no. of occurences)

⇒ lower_bound() : iterator pointing to first occurence of val if present else the position of next greater value. ($T_c = O(\log N)$)

⇒ upper_bound() : position of next greater value.

Unordered_Multiset :

→ allows duplicate values
→ values are not ordered
→ values will be identified by itself
→ values can't be modified

→ Code:
```
#include<unordered_set>
{ unordered_multiset<int> ms1;
  : (rest all same)
}
```

Member functions of unordered_multiset :

⇒ insert() : → single element – O(1) avg., O(N) worst
↳ multiple element – O(n) avg., O(n*(N+1)) worst
(rehashing) (size ≥ capacity)

⇒ Deletion : erase( ) → T_c = O(1) avg
O(N) worst

⇒ find( ) : T_c = O(1) avg
O(N) worst

⇒ count( ) : T_c = O(N) avg (No. of occurrences)
O(N) worst (size)

Q) Given n integers (can be duplicates), print second smallest integer. If it doesn't exist, print -1.

ip 1: n=4
1 2 2 -4

op 1: 1

ip 2: n=5
1 2 3 1 1

op 2: 2

A) We can use ordered set, so that we can all elements in ordered way & we can print the 2nd value of set.

Code:
```
int main(){
    int n; cin>>n;
    vector<int> v(n);
    for(int i=0; i<n; i++) cin>>v[i];
    set<int> s; int ans=0;
    for(auto val: v) s.insert(val);
    auto itr = s.begin();
    itr++;
    cout<<"2nd smallest int "<<*itr;
    return 0;
}
```

Q) Given no. of ques. 'n' & marks for correct ans as 'p' & for incorrect ans. One can either attempt to get 'p' marks if ans is right, or 'q' marks if answer is wrong, or leave the ques unattended & get 0 marks. Task is to find the count of all the diff. possible marks that one can score?

ip 1: n=2, p=1, q=-1 } explanation: diff. possible marks are: -2, -1, 0, 1, 2

op 1: 5

A) Explanation n=2:

	score
0 0	0
-1	-1
1	1
1 1	2
1 -1	0
-1 -1	-2

→ -1, 0, 1, 2, -2 → size = ans = 5

(we can use unordered set, as we don't need any order, & only size is req.)

score { (i) (0 to n) correct * p + (j) (0 to n) incorrect * q + (n-i-j) unattended * 0

Code:
```
int main(){
    int n,p,q; cin>>n>>p>>q;
    unordered_set<int> s;
    for(int i=0; i<=n; i++){
        for(int j=0; j<=n; j++){
            int correct = i;
            int incorrect = j;
            int unattended = n-(i+j);
            if(unattended>=0){
                int score = correct*p + incorrect*q;
                s.insert(score);
            }
            else{
                break;
            }
        }
    }
    cout<<"Ans : "<<s.size();
    return 0;
}
```

- $T_c = O(n^2)$
- $S_c = O(n)$ (size of set)

Introduction To Hashing And Hash Table

This concept helps us reduce searching to $T_c = O(1)$

ex: We have 682, 761, 494, 567, 869,... no.'s & we want any ele. to get in O(1).

So we can create arr[1000] size & arr[682] = 682, ...

But if ele. is 87926433, then it would be difficult. So we introduced hashing!

682, 761, 494, 567, 869 → suppose we want to store at max size = 10.

finding a (unique index) (hash value) for all elements to store them → Hashing

soln.

We make a "hash func." = [K mod 10] (reduces arr. no.)

∴ 682, 761, 494, 567, 869
Hash values { =2, =1, =4, =7, =9

We create table of size 10 : (10 buckets)

0	
1	761
2	682
3	
4	494
5	
6	
7	567
8	
9	869

Hash Table

Now if we want 761 → Hash func. → found ✓ (O(1))

(Similarly if we wanted in size 7 → Hash func = [K mod 7])

Hash Functions:

1) Division Method: $h(K) = K \bmod m$ → buckets

ex: m = 11

1276 mod 11 = 0 → stored in bucket 0

2) Mid Square Method: $h(K) = K^2$ & extract middle digits

ex: K = 13

$K^2 = 169$ → so stored at bucket 6

3) Digit Folding Method: we fold into equal size parts

ex: $K_1K_2K_3K_4K_5$ → we can: K_1K_2 + K_3K_4 + K_5

ex: K = 12345 → 12 + 34 + 5 = 51 (Hash value)

we can: $K_1K_2K_3$ + K_4K_5 = Hash value

4) Multiplication Method: $h(K) = \text{floor}(M(KA \bmod 1))$

(M: size of hash table; A: $0 < A < 1$)

ex: K = 12345

A = 0.01

∴ KA = 123.45

KA mod 1 = (123.45 mod 1) = 0.45

M = 0.45 × 10 (let) = 4.5 → ∴ floor(M(KA mod 1)) = floor(45) = 4

Collisions: (when 2 elements have same hash value)

We have 2 way to solve it: → Open Hashing
→ Closed Hashing

• Open Hashing (Closed Addressing):

We use method called "seperate chaining" {h(K) = K mod M}

ex: 682, 761, 494, 567, 869, 634, 794

h(K) = K mod 10 → 4 (Hash table size)

0	
1	761
2	682
3	
4	494 → 634 → 794
5	
6	
7	567
8	
9	869

* If we have same hash value, then add elements using linked lists

* $T_c = O(l)$ (l: length of linked list)

• Closed Hashing (Open Addressing):

Probes - no. of chances to find right position for our element

In this, we have 2

a) Linear Probing: ex: 682, 761, 494, 567, 869, 634, 794 (794: 3 probes; 634: 2 probes)

Let size = 10

∴ h(K) = K mod 10

∴ Here ⇒ $(h(K)+i) \bmod M \quad \{0 \le i \le M\}$

i.e. $(h(K)+i) \bmod 10 \quad \{0 \le i \le 9\}$

h(634) = (4+0) mod 10 = 4

= (4+1) mod 10 = 5 → i++

h(794) = (4+0) mod 10 = 4

= (4+1) mod 10 = 5

= (4+2) mod 10 = 6

0	
1	761
2	682
3	
4	494
5	634
6	794
7	567
8	
9	869

* No. is stored where bucket is empty.

* Now 4 is stored in row, so cluster will occur, so we use optimized Methods

b) Quadratic Probing: ex: 682, 761, 494, 567, 869, 634, 794

$(h(K) + i^2) \bmod M \quad ; 0 \le i < M$

$h(K) = K \bmod M$

h(634) = (4+0) mod 10 = 4

= (4+1²) mod 10 = 5

h(794) = (4+0) mod 10 = 4

= (4+1²) mod 10 = 5

= (4+2²) mod 10 = 8

0	
1	761
2	682
3	
4	494
5	634
6	
7	567
8	794
9	869

c) Double Hashing: $(h_1(K) + ih_2(K)) \bmod M \quad ; 0 \le i < M$

ex: 682, 761, 494, 567, 869, 634

$h_1(K) = K \bmod 10$, $h_2(K) = K \bmod 5$

∴ h(634) = $h_1(634) + 0 = 4$

= $[h_1(634) + 1 \times h_2(634)] \bmod 10$

= $[4 + 1 \times 4] \bmod 10 = 8$

0	
1	761
2	682
3	
4	494
5	
6	
7	567
8	634
9	869

Load Factor:

n → no. of elements, m → no. of buckets → Load factor = n/m {avg. entries in 1 bucket}

(we try to keep LF as low)

• Load Factor limit = 0.75 (max.)

Rehashing: If load factor (LF) > 0.75 (limit)

↳ we increase size of hash table & re-distributing elements in it.

If we don't do rehashing then collisions will increase resulting in more time complexity for searching. Therefore, it's necessary to rehash once LF > limit.

Q) Implement hash table with closed addressing?

```
#include<bits/stdc++.h>
using namespace std;
class Hashing{
    vector<list<int>> hashtable;
    int buckets;
public:
    Hashing(int size){
        buckets = size;
        hashtable.resize(size);
    }
    int hashvalue(int key){
        return key % buckets;
    }
    void addkey(int key){
        int idx = hashvalue(key);
        hashtable[idx].push_back(key); // adding at end of list
    }
    list<int>::iterator searchkey(int key){ //returns an iterator
        int idx = hashvalue(key);
        return find(hashtable[idx].begin(), hashtable[idx].end(), key);
    }
    void deletekey(int key){
        int idx = hashvalue(key);
        if(searchkey(key) != hashtable[idx].end()){ // returns end of hashtable
            hashtable[idx].erase(searchkey(key));
            cout<<key<<" is deleted "<<endl;
        }
        else{
            cout<<"Key is not present in hashtable "<<endl;
        }
    }
};
int main(){
    Hashing h(10);
    h.addkey(5);
    h.addkey(3);
    h.addkey(9);
    h.deletekey(3);
    h.deletekey(3);
    cout<<*(h.searchkey(5));
    return 0;
}
```

We need a table! So we create a table using vector (more friendly than arrays). Now in each bucket we must have a linked list, so we create a vector of linked lists!

push_back: most recent item added at end of LL & order of insertion is preserved

push_front: most recent item added at front of LL & order of insertion isn't preserved

↳ eg: 3 is deleted

$\therefore T_c = O(1)$ → search, add, delete

HASHMAP - STL & TYPES_IMPLEMENTATION

Map in C++ STL: (dynamic)

(can be vice versa too)

(Value) ← (Key)

→ STL container which stores key-value pairs eg: Phno's → Name (as pairs)

(by default)

→ These elements are stored in ascending/descending order

→ Maps cannot have duplicate keys.

→ Implemented through BST (Binary Search Tree)

- Header File Req.: #include<map>
- Declaration: map<key_datatype, value_datatype> map_name; (by default ascending)
  eg: map<string, int> directory;

If we want in descending order;

map<datatype1, datatype2, greater<datatype1>> map_name;

- Initialization: map<datatype1, datatype2> map_name = {{key1, val1}, {key2, val2}};
- Insertion: map_name.insert(make_pair(new_key, new_value)) (make_pair: keyword)
  $T_c = O(\log N)$
  (or)
  map_name[key] = value
- Printing the elements: We use → for each loop → each ele. is a key-value pair

```
for(auto element : map_name){
    key = element.first;
    value = element.second;
}
```

Code:

```
map<string, int> di;
di["naman"]=3876;
di["av"]=8555;
for(auto ele:di){
    cout<<ele.first<<" "<<ele.second;
    cout<<endl;
}cout<<endl;
// to add/insert
di["av"]=7777;
```

o/p: av 8555
naman 3876

↳ this updates "av" val. to 7777

o/p: av 7777
naman 3876

```
di.insert(make_pair("av", 8473));
di.insert(make_pair("chap", 9123));
```

if key name is already present, then no changes will be done.

if new key name is there, then it will be added to map.

o/p: av 8555
chap 9123
naman 3876

Member Functions:

{Iterators are used to traverse our STL containers}

⇒ erase(): → m.erase(itr) | m.erase(key) | m.erase(start_it, end_it)

→ $T_c = O(\log N)$

[start, end) → $T_c = O(N)$ ↳ length

⇒ swap(): m1, m2 → same type

m1.swap(m2) | swap(m1, m2)

⇒ clear(): m.clear() → clears all my elements in the map

⇒ empty(): m.empty() → returns 1 if map 'm' is empty

⇒ size(): m.size() → returns total no. of key-value pairs in the map

⇒ max_size(): returns max size allocated to map

⇒ find(): returns itr to element if present, else it returns map.end() itr

(m.find(key)) → $T_c = O(\log n)$

[ ], [ ], [ ], [ ] : m.begin() ↑ , m.end() ↑

⇒ count(): no. of occurences of key

↳ m.count(key)

⇒ upper_bound(): returns an itr of next greater element

↳ m.upper_bound(key)

⇒ lower_bound(): returns itr to element if present, else itr to next greater ele

⇒ begin(), end(), rbegin(), rend():

ex: m.rend() → [K_1, V_1 ← m.begin()
K_2, V_2
K_3, V_3
m.rbegin() → K_4, V_4]
← m.end()

⇒ Traversal: Code

```
map<string,int>::iterator itr;
for(itr = di.begin() ; itr != di.end() ; itr++){
    cout<< itr->first << " " << itr->second <<endl;
}
```

Output: av 8555
chap 9123
raman 3876

If we want to traverse in reverse;

```
map<string,int>::reverse_iterator itr;
for(itr = di.rbegin() ; itr != di.rend() ; itr++)
    // rest all same & o/p in ↓ order
```

Q) You are given 'n' → no. of ele's. Now you are given 'n' ele's. Return sum of repetitive elements that appear more than 1 time

if: n = 7
ele = [1,1,2,1,3,3,3]] o/p: 4
exp: Repetitive ele.'s = 1, 3 → sum = 1+3 = 4

Code:

```
int main(){
    int n; cin>>n;
    vector<int> v(n);
    for(int i=0;i<n;i++) cin>>v[i];
    map<int,int> m;
    for(int i=0;i<n;i++){
        m[v[i]]++;
    }
    int sum=0;
    for(auto pair:m){
        if(pair.second>1){
            sum+= pair.first;
        }
    }
    cout<<sum;
    return 0;
}
```

Storing freq. of every ele. in 'v' array

finding sum of repetitive elements

pair - element, frequency

We can solve this using map:

↳ element - freq.
↳ if freq. > 1
↳ add!

$T_c = O(n)$ → total ele

$S_c = O(K)$ → total unique ele's

Unordered Map:

→ STL container, stores key-value pairs

→ Elements are not ordered {→ Implemented using Hashing}

→ Keys will be unique

→ Insertion, Deletion, Search } $T_c = O(1)$ → avg. case
$\ll O(n)$ → worst case

Code

```
#include<unordered_map>
int main(){
    unordered_map<int,string> record;
    record.insert(make_pair(3,"ria"));
    record[1] = "bali";
    record[3] = "chimp"; → for updation
    record[2] = "hi";
    record.insert(make_pair(3,"ankit"));

    for(auto pair:record){
        cout<<"Roll "<<pair.first<<endl;
        cout<<"Name"<<pair.second<<endl;
    }
    return 0;
}
```

if key is present, then no changes done. If not present, then pair is added

o/p: Roll 2
Name hi
Roll 1
Name bali
Roll 3
Name chimp
} random order

Member Functions:

→ erase( ): → m.erase(itr) | m.erase(key) | m.erase(start_itr, end_itr)

→ find( ): → returns itr to ele. if present else it returns m.end()
↳ m.find(1);
↳ "key value"

→ count( ): → returns no. of occurences of key
m.count(key) ~ 1 or 0

→ begin( ): → returns itr to first element
↳ m.begin();

→ end( ): → returns itr to the position after the last element

[(1,0), (2,0), ...]
itr
advance(itr);

Multimap:

→ STL container, stores key-value pairs
→ Elements are stored in an order - ascending / descending
→ Duplicate keys are allowed ~ ex:

Key	Val
Name	Ph 1
Name	Ph 2

→ Implemented by using BST → (binary search tree)

→ Insert, Delete, Search } $T_c = O(\log n)$

Code

```
#include<map>
int main(){
    multimap <string, int> di;  // di = directory
    di.insert (make_pair("Hurra", 2394));
            "          ("Ankit", 8555));
            "          ("Hurra", 9539));
    // di["urvi"] = 2930; → not allowed in multimap! It will give error.
    for(auto p:di){
        cout<<"Name" << p.first <<endl;
        cout<<"No." << p.second <<endl;
    }
    return 0;
}
```

↳ o/p:
Ankit
8555
Hurra
2394
Hurra
9539

Member Functions:

→ m.insert
→ m.erase → erase(itr), erase(key) } $T_c = O(\log n)$
→ erase(start_itr, end_itr) → $T_c = O(n)$ (range of ele.)

m[key] = value

→ m.find: returns itr to ele. if present, else m.end()

→ m.begin(), m.end()

Unordered Multimap:

→ STL container that stores key-value pair
→ elements are not ordered
→ duplicate keys are allowed
→ Implemented using Hashing

→ Insert, Delete, Search } $T_c = O(1)$ → avg case
$T_c = O(n)$ → worst case

Code

```
#include <unordered_map>
int main(){
    unordered_multimap<string, int> fc; //fc = fruit count
    fc.insert(make_pair("apple", 3));
            "          ("Kiwi", 4));
            "          ("apple", 5));

    for(auto p:fc){
        cout<<"Name "<<p.first<<endl;
        cout<<"Count "<<p.second<<endl;
    }
    return 0;
}
```

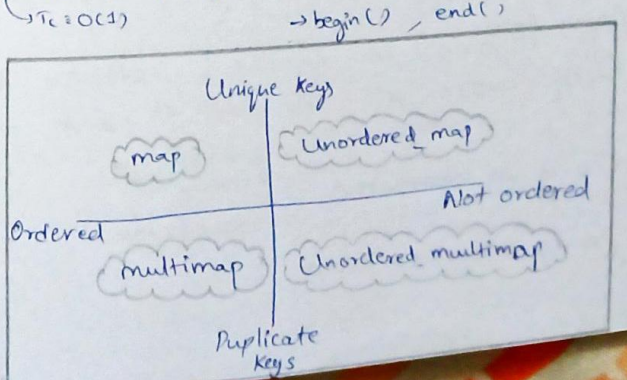
// fc["b"] = 4 } not possible

↳ o/p:
Name Kiwi
Count 4
Name apple
Count 5
Name apple
Count 3

Member Functions:

→ Insert : $T_c = O(1)$
→ Erase : $T_c = O(1)$
→ $T_c = O(n)$ } range
↳ $T_c = O(1)$
→ Find : $T_c = O(1)$
→ Count : can be more than 1 → $T_c = O(n)$ (no. of occurrences)
→ begin(), end()

*



Q) o/p :? → Multimap<int, int> m;

```
m.insert({1,1});
   "   {2,4};
   "   {3,9};
   "   {2,16};
   "   {2,20};
```

start_itr →
end_itr →

```
auto a = m.equal_range(4);
for(auto it = a.first ; it != a.second ; it++)
    cout << it->first << "=>" << it->second << endl;
```

A) equal_range(key): returns bound of range of elements with key

returns bounds: upper, lower

↳ 4=>1
4=>2

HASHMAP - IMPORTANT PROBLEMS IN STL - UNORDERED MAP

Q) Given array of strings, you can move any no. of characters from 1 string to any other string any no. of times. You just have to make all of them equal.
Print 'yes' if possible else 'No'.

i/p 1: ["collegeee", "coll", "collegge"]
o/p 1: Yes

i/p 2: ["wall", "ah", "wallahah"]
o/p 2: No

A) i) We don't need to transform strings by moving characters.
2) Each character should be present in multiple of length of array

c - [] {char, freq}

Code:

```
bool makeEqual(vector<string> &v){
    unordered_map<char,int> m;
    for(auto str : v){          // we get 1 string
        for(auto c : str){      // we get char of this string
            m[c]++;
        }
    }
    int n = v.size();
    for(auto ele : m){
        if(ele.second % n != 0){
            return false;
        }
    }
}
```

```
int main(){
    int n; cin>>n;
    vector<string> v(n);
    for(int i=0; i<n; i++){
        cin>>v[i];
    }
    cout<<(makeEqual(v)?"Yes":"No");
    return 0;
}
```

∴ T_c : O(n) → no. of all characters

S_c : O(m) → no. of unique characters

Q) Check whether 2 strings are anagram of each other. Return true if they are else return false.

i/p 1: triangle, integral
o/p 1: true

i/p 2: anagrams, grams
o/p 2: False

A) To check: ① same characters ② same freq

Approach 1: Make 2 freq. array of length 26 & compare them in last

Approach 2: Make 2 maps → <char, int> & compare

Approach 3: Make a map → store char & freq.!
Now while traversing 2nd str (via map) if any common ele. is present then do '-1'. In the end our map should have only 0's else False.

Code:

```
bool checkAnagram(string s1, string s2){
    if(s1.length() != s2.length()) return false;
    unordered_map<char,int> m;
    for(auto c1 : s1){
        m[c1]++;
    }
    for(auto c2 : s2){
        if(m.find(c2) == m.end()) return false;
        else m[c2]--;
    }
    for(auto ele : m){
        if(ele.second != 0) return false;
    }
    return true;
}
```

```
int main(){
    string s1,s2;
    cin>>s1>>s2;
    cout<<(checkAnagram(s1,s2)
        ? "yes" : "No");
    return 0;
}
```

T_c = O(n) (sum of length of s1 & s2), S_c : O(N) (length of s1 or s2)

Q) Check whether 2 strings are isomorphic of each other. Return true if they are else return false.

i/p 1: aab, xxy
o/p 1: True

i/p 2: abcdec, vicwog
o/p 2: False

A) ex: str1 → a b c d e m
str 2 → v i o u o g

map:
a → v
b → i
c → o
d → u
e → o
m → g
(key → value)

Now we find similar keys & check their ... ✓(passed) (we can only check from key)
{we can't check value}
↓
So next time, we pass str2 as key & other as val. & check!

Code:

```
bool check11Mapping(string s1, string s2){
    unordered_map<char,char> m;
    for(int i=0;i<s1.length();i++){
        if(m.find(s1[i])!=m.end()){
            if(m[s1[i]]!=s2[i]){      → if( c→e , c→d ) → false
                return false;
            }
        }
        else{ // if new element added
            m[s1[i]] = s2[i];
        }
    }
    return true;
}

bool checkIso(string s1, string s2){
    if(s1.length()!=s2.length()) return false;
    bool s1s2 = check11Mapping(s1,s2);   → check one to many map from s1 → s2
    bool s2s1 = check11Mapping(s2,s1);   → check one to many from s2 → s1
    return (s1s2 and s2s1);
}

int main(){
    string s1,s2;
    cin>>s1>>s2;
    cout<<(checkIso(s1,s2)? "Yes" : "No");
    return 0;
}
```

• $T_c = O(N)$ → sum of length of s1 & s2
$S_c = O(n)$ → unique characters in s1 & s2

Q) Given an array of length n & a target, return a pair whose sum is equal to the target. If there is no pair present, return -1.

i/p1: n = 7 → [1,4,5,11,13,10,2], target 13 | o/p1: [3,6]
i/p2: n=5 → [9,10,2,3,5], target = 15 | o/p2: [-1 ...

A) 1) Brute force: Nested loops → $O(n^2)$
2) Opt. → 2 pointer approach → $O(\log n)$ — T_c
3) Using map & storing prev. elements → $O(n)$

ex → i/p1:
1-0 → 13-1=12 is there in map? No, so save & go to next ele
4-1 → 13-4=9 is " " map? "
5-2 → 13-5=8 "
11-3 → 13-11=2 "
13-4 → 13-13=0 "
10-5 → 13-10=3 "
2-6 → 13-2=11 Yes it's there in map, so print indexes

Code:
(we always pass by reference for memory efficiency)

```
vector<int> tgtSumPair(vector<int>&v, int tgtSum){
    unordered_map<int,int> m;
    vector<int> ans{-1,-1};
    for(int i=0; i<v.size(); i++){
        if(m.find(tgtSum-v[i])!=m.end()){
            ans[0] = m[tgtSum-v[i]];
            ans[1] = i;
        }
        else{
            m[v[i]] = i;
        }
    }
    return ans;
}

int main(){
    int n; cin>>n;
    vector<int> v(n);
    for(int i=0;i<n;i++){
        cin>>v[i];
    }
    int targetSum;
    cin>>targetSum;
    vector<int> ans = tgtSumPair(v,targetSum);
    cout<<ans[0]<<" "<<ans[1];
    return 0;
}
```

Here $S_c = O(n)$ → total ele. in array.

Q) Given an array arr[] of length N, find the longest subarray with a sum equal to 0.

i/p1: n=8 → {15,-2,2,-8,1,7,10,23} | i/p2: n=3 → {1,2,3}
o/p1: 5 | o/p2: 0

A) ① Brute force = nested loops = $O(n^2)$ → T_c
② a[l, ... r] = 0
⇒ a[l] + a[l+1] + ... + a[r] = 0
⇒ (a[0] + ... + a[r]) - (a[0] + ... a[l-1]) = 0
⇒ $PrefixSum_r - PrefixSum_{l-1} = 0$ ⇒ $PrefixSum_r = PrefixSum_{l-1}$

15, 13, 15, 7, 8, 15, 25, 48 (in map)
ex: 15, -2, 2, -8, 1, 7, 10, 23
15-0 (we check length), 15-1, 15, 7-3, 8-7

-∴ If repeated ele, we check length!

; $T_c = O(n)$
$S_c = O(n)$

Code:

```
int maxLenZeroSumSubarray(vector<int> &v){
    unordered_map<int,int> m;
    int prefixSum=0, maxlen=0;
    for(int i=0;i<v.size();i++){
        prefixSum+=v[i];
        if(prefixSum==0){ // if our vector has all 0s only
            maxlen++;
        }
        if(m.find(prefixSum)!=m.end()){
            maxlen=max(maxlen, i-m[prefixSum]);
        }
        else{
            m[prefixSum]=i;
        }
    }
    return maxlen;
}
```

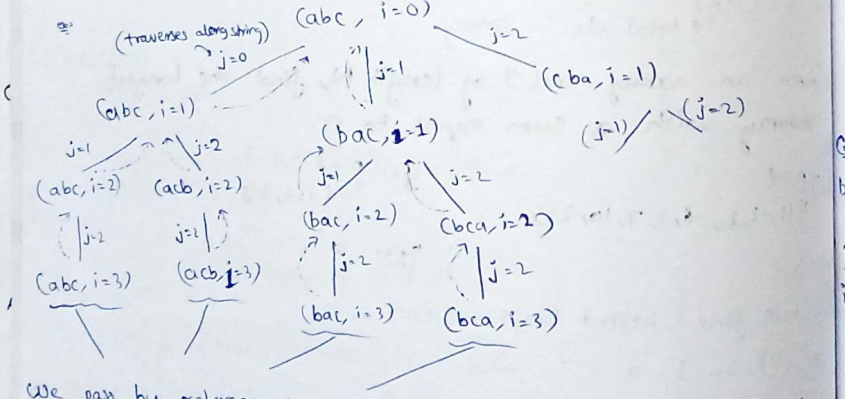
```
int main(){
    int n; cin>>n;
    vector<int> v(n);
    for(int i=0;i<n;i++) cin>>v[i];
    cout<<maxLenZeroSumSubarray(v);
    return 0;
}
```

BACKTRACKING - 1

Q) WAP to print all permutations of the given string s in lexographically sorted order.

e.g. PQR PQR PRQ QPR QRP RPQ RQP

A) 'PQR' → we make each character as first character & make permutations



We pass by reference to save space. Since the end string is different than "abc", so we swap it again as we move backwards through recursion.

Code:

```
void permutation(string str, int i){
    if(i==str.size()){ //base case
        cout<<str<<endl;
        return;
    }
    for(int j=i; j<str.size(); j++){
        swap(str[i], str[j]);
        permutation(str, i+1);
        swap(str[i], str[j]); // in this way
    }
}
```

```
int main(){
    string s="abc";
    permutation(s,0);
    return 0;
}
```

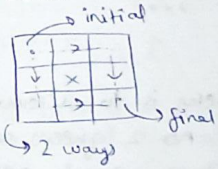
↳ op: abc
acb
bac
bca
cab
cba

- $T_c = O(n!)$ ↳ as: $f(n) = n \times f(n-1)$

Q) RAT IN A MAZE
A maze in N*N binary matrix of blocks where the upper left block is known as the source block, & the lower right most block is known as the destination block. If we consider the maze, then maze[0][0] is the source & maze[n-1][n-1] is the destination. Our main task is to reach the destination from the source. We have considered a rat as a character that can move either forward (or) downwards.

In the maze matrix, a few dead blocks will be denoted by 0 & active blocks will be denoted by 1. A rat can only move in the active blocks.

A) eg:



→ X → marking as '0'
Rest all blocks as '1'
Now as we move through maze/each box we change it to '2' so that (i.e. ↻) we don't get stuck in a loop. Once we get a way, we go back each step finding possibilities (making it 1) & getting ans.

Code:

```
bool canWeGo(int a, int b, vector<vector<int>> &grid){
    return (a<grid.size() and b<grid.size() and a>=0 and b>=0 and grid[a][b]==1);
}
int f(int i, int j, vector<vector<int>> &grid){
    int n=grid.size();
    if(i==n-1 and j==n-1){
        for(int i=0;i<n;i++){
            for(int j=0;j<n;j++){
                cout<<grid[i][j]<<" ";
            } cout<<endl;
        } cout<<"****"<<endl;
        return 1;
    }
```

If you want to see all paths, else not needed

```
int ans = 0;
grid[i][j] = 2; // setting (0,0) as 2
if(canWeGo(i,j+1,grid)) ans += f(i,j+1,grid); // right
if(canWeGo(i+1,j,grid)) ans += f(i+1,j,grid); // down
if(canWeGo(i,j-1,grid)) ans += f(i,j-1,grid); // left
if(canWeGo(i-1,j,grid)) ans += f(i-1,j,grid); // up
grid[i][j] = 1; // all paths checked, so now going 1 block back & changing current block from 2 -> 1.
return ans;
}

int main(){
    vector<vector<int>> grid = {
        {1,1,1,1},
        {0,1,0,1},
        {0,1,1,1},
        {0,1,1,1}
    };
    int res = f(0,0,grid);
    cout<<res;
    return 0;
}
```

op:

```
2 2 2 2
0 1 0 2
0 1 1 2
0 1 1 1
****
2 2 2 2
0 1 0 2
0 1 2 2
0 1 2 1
****
2 2 2 2
0 1 0 2
0 2 2 2
0 2 2 1
****
2 2 1 1
0 2 0 1
0 2 2 2
0 1 1 1
****
2 2 1 1
0 2 0 1
0 2 2 1
0 1 2 1
****
```

```
2 2 1 1
0 2 0 1
0 2 1 1
0 2 2 1
****
2 2 1 1
0 2 0 1
0 2 2 1
0 2 2 1
****
```

7

N QUEENS

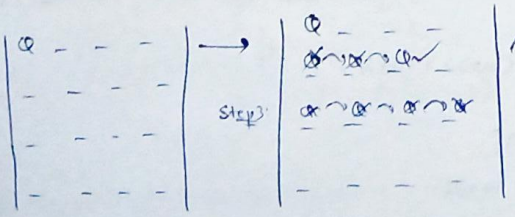
Consider an N*N chessboard. N Queen problem is to accomodate N queens on the N*N chessboard such that no 2 queens can attack each other.

ip: n = 4

op:
```
{0, 1, 0, 0}
{0, 0, 0, 1}
{1, 0, 0, 0}
{0, 0, 1, 0}
```

Each 1 = Queen can't attack any other queen.
↳ so obviously only 1 queen in a row/column.

ex:



So now we move 2nd queen to next position & check. Once all combo. done, we revert back to first queen & move her to next position & then check all possibilities.

Code:

```
bool canPlaceQueen(int row, int col, vector<vector<char>>& grid){
    // attack will only come from up & not down
    for(int i=row-1; i>=0; i--){ // if someone attacking from vertical up
        if(grid[i][col]=='Q') return false;
    }
    // checking diagonally row (right to left) {↖}
    for(int i=row-1, j=col-1; i>=0 and j>=0; i--,j--){
        if(grid[i][j]=='Q') return false;
    }
    // checking diagonally row (left to right) {↗}
    for(int i=row-1, j=col+1; i>=0 and j<grid.size(); i--,j++){
        if(grid[i][j]=='Q') return false;
    }
    return true;
}

void nqueen(int currRow, int n, vector<vector<char>>& grid){
    if(currRow==n){ // means we have checked all rows
        for(int i=0; i<n; i++){
            for(int j=0; j<n; j++){
                cout<<grid[i][j]<<" ";
            } cout<<endl;
        } cout<<"****"<<endl;
        return;
    }
    for(int col=0; col<n; col++){
        // we will go to all cols
        // lets check if we can queen at currRow & col
        if(canPlaceQueen(currRow, col, grid)){
            grid[currRow][col] = 'Q';
            nqueen(currRow+1, n, grid); // moving to next row
            grid[currRow][col] = '.'; // reverting the state
        }
    }
}

int main(){
    int n=4;
    vector<vector<char>> grid(n, vector<char>(n,'.'));
    nqueen(0,n,grid);
    return 0;
}
```

op:

```
. Q . .
. . . Q
Q . . .
. . Q .
****
. . Q .
Q . . .
. . . Q
. Q . .
****
```