UNIVERSITY OF DELHI

DEPARTMENT OF COMPUTER SCIENCE

NAME : Souptik Datta ROLL NO. : 55 Date : 19/02/23

Question 1)

Approach :

- Traverse the array from index 0 to end.
- For each element start another loop from index i+1 to end.
- If a greater element is found in the second loop then print it and break the loop, else print -1.

Below is the implementation of the above approach :

```
//C++ program to print next greater elements in a given array

#include <iostream>
using namespace std;
/* prints element and NGE pair
for all elements of arr[] of size n */
void printNGE(int arr[], int n)
{
    int next, i, j;
    for (i = 0; i < n; i++) {
        next = -1;
        for (j = i + 1; j < n; j++) {
            if (arr[i] < arr[j]) {
                next = arr[j];
                break;
            }
        }
        cout << arr[i] << " --> " << next << endl;
    }
}
// Driver Code
int main()
{
    int arr[] = {2,8,3,1,5,9};
    int n = sizeof(arr) / sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}
```

Output :

```
2 --> 8
8 --> 9
3 --> 5
1 --> 5
5 --> 9
9 --> -1
```

Time Complexity :

O(N^2)

Space Complexity :

O(1)

Question 2)

Approach :

We start by checking if the length of the input string is less than 2 . If it is , then we simply return the input string as there can be no adjacent duplicates in a string of length of 0 or 1.
If the length of the string is greater than or equal to 2 , we check if the first two characters of the string are equal . If they are , we remove those two characters and recursively call the function on the rest of

the string. If they are not equal, we recursively call the function on the rest of the string and concatenate the first character of the input string with the result of the recursive call.

Below is the implementation of the above approach :

```
#include <iostream>
#include <string>
using namespace std;

string removeAdjacentDuplicates(string s) {

    //return the input string as there can be no adjacent duplicates in a string of length 0 or 1.
    if (s.length() < 2) {
        return s;
    }

    // length of the string is greater than or equal to 2
    int i = 0;
    while (i < s.length() - 1 && s[i] == s[i+1]) {
        i++;
    }
    //if the first two characters of the string are equal,remove those two characters &
    //recursively call the function on the rest of the string
    if (i > 0) {
        s = s.substr(i+1);
        return removeAdjacentDuplicates(s);
    }
    //If not equal,recursively call the function on the rest of the string &
    //concatenate the first character of the input string with the result of the recursive call
    string rest = removeAdjacentDuplicates(s.substr(1));
    if (rest.length() > 0 && s[0] == rest[0]) {
        return rest.substr(1);
    } else {
        return s[0] + rest;
    }
}

//Driver code
int main() {

    string s1 = "Careermonk";
    cout << removeAdjacentDuplicates(s1) << endl;
    string s2 = "Mississippi";
    cout << removeAdjacentDuplicates(s2) << endl;
    return 0;
}
```

Output :

```
Camonk
M
```

Time Complexity :

O(N^2), where N is the length of the string.

Space Complexity :

O(N)

Question 3)

Approach :

1) Find the middle point using tortoise and hare method.
2) Split the linked list into two halves using found middle point in step 1.
3) Reverse the second half.
4) Do alternate merge of first and second halves.

Below is the implementation of the above approach :

```
// C++ program to reorder a linked list without using extra space
#include <bits/stdc++.h>
```

```
using namespace std;

// Linkedlist Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to create newNode in a linkedlist
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Function to reverse the linked list
void reverselist(Node** head)
{
    // Initialize prev and current pointers
    Node *prev = NULL, *curr = *head, *next;
    while (curr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    *head = prev;
}

// Function to print the linked list
void printlist(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        if (head->next)
            cout << "-> ";
        head = head->next;
    }
    cout << endl;
}

// Function to rearrange a linked list
void rearrange(Node** head)
{
    //Find the middle point
    Node *slow = *head, *fast = slow->next;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    //Split the linked list in two halves head1, head of first half 1 -> 2
    //and head2, head of second half 3 -> 4
    Node* head1 = *head;
    Node* head2 = slow->next;
    slow->next = NULL;

    //Reverse the second half, i.e., 4 -> 3
    reverselist(&head2);

    //Merge alternate nodes

    *head = newNode(0); // Assign dummy Node

    //curr is the pointer to this dummy Node, which will
    // be used to form the new list
    Node* curr = *head;

    while (head1 || head2) {
        //First add the element from list
        if (head1) {
            curr->next = head1;
            curr = curr->next;
            head1 = head1->next;
        }
        //Then add the element from the second list
        if (head2) {
            curr->next = head2;
            curr = curr->next;
            head2 = head2->next;
        }
```

```
    }
    //Assign the head of the new list to head pointer
    *head = (*head)->next;
}

// Driver program
int main()
{
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    printlist(head); // Print original list
    rearrange(&head); // Modify the list
    printlist(head); // Print modified list

    return 0;
}
```

Output :

```
1 -> 2 -> 3 -> 4 -> 5
1 -> 5 -> 2 -> 4 -> 3
```

Time Complexity :

O(N)

Space Complexity :

O(1)

Question 4)

Approach :

a. If an empty list return -1
b. Traverse the list and count the number of elements in it
c. Calculate the square root of the count (i.e. root(n)) .
d. Traverse the list again , stopping at the root(n)th element.
e. Return the value of the root(n)th element.

Below is the function of the above approach :

```
//Function to find the value of root(n)th element
int rootNthElement(ListNode* head) {

    // return an appropriate value for an empty list
    if (head == NULL) {
        return -1;
    }

    int count = 0;
    ListNode* current = head;

    // Traverse the list and count the number of elements
    while (current != NULL) {
        count++;
        current = current->next;
    }

    // Calculate root(n)
    int root = sqrt(count);

    // Traverse the list again, stopping at the root(n)th element
    current = head;
    for (int i = 0; i < root && current != NULL; i++) {
        current = current->next;
```

```
    }

    // Return the value of the root(n)th element
    return current->val;
}
```

Time Complexity :

O(n), where n is the number of elements in the linked list

Space Complexity :

O(1)

Question 5)

Below is the implementation of the above approach :

```
//Find the modular node from the end
#include<iostream>
using namespace std;
//structure of the node
struct node
{
    int data;
    struct node *next;
};
//fuction to create a node
struct node *newNode(int data)
{
    struct node *NODE = new struct node;
    NODE -> data = data;
    NODE -> next = NULL;

    return NODE;
}
//function to find the modular node from end
struct node *modul_node(struct node *head, int k)
{
    struct node *prev, *current;
    prev = current = head;
    int i = 1;
    /*in this loop at first we moving current to kth node from first and then moving both current
    and prev n-k times */
    while (current -> next != NULL)
    {
        current = current -> next;
        i++;
        if(i>k)
        {
            prev = prev -> next;
        }
    }
    return prev;

}

int main()
{
    // creating a linkedlist
    struct node *head = newNode(0);
    head -> next = newNode(1);
    head -> next -> next = newNode(2);
    head -> next -> next -> next = newNode(3);
    head -> next -> next -> next -> next = newNode(4);
    head -> next -> next -> next -> next -> next = newNode(5);
    head -> next -> next -> next -> next -> next -> next = newNode(6);
    head -> next -> next -> next -> next -> next -> next -> next = newNode(7);
    //let
    int k = 3;
    struct node *req_node = modul_node(head,k);
    cout<<"The value at modular node is "<<req_node -> data<<endl;
    return 0;
```

```
}
```

Output :

```
The value at modular node is 5
```

Time Complexity :

O(n), where n is the number of nodes in the linkedlist.

Space Complexity :

O(1)

Question 6)

Approach :

The idea is to sort the given array in ascending order and maintain search space by maintaining two indices (low and high) that initially points to two endpoints of the array. Then reduce the search space nums[low…high] at each iteration of the loop by comparing the sum of elements present at indices low and high with the desired sum. Increment low if the sum is less than the expected sum; otherwise, decrement high if the sum is more than the desired sum. If the pair is found, return it.

Below is the pseudo code of the algorithm :

```
// Function to find a pair in an array with a given sum using sorting
void findPair(int nums[], int n, int S)
{
    //sort the array in ascending order
    sort(nums, nums + n);

    //maintain two indices pointing to endpoints of the array
    int low = 0;
    int high = n - 1;

    //reduce the search space `nums[low…high]` at each iteration of the loop till the searchspace is exhausted
    while (low < high)
    {
        // sum found
        if (nums[low] + nums[high] == S)
        {
            cout << "Pair found is : (" << nums[low] << ", " << nums[high] << ")\n";
            return;
        }

        if (nums[low] + nums[high] < S) {
            // increment `low` index if the total is less than the desired sum;
            low++;
        }

        else {
        //decrement `high` index if the total is more than the desired sum
        high--;
        }
    }

    // we reach here if the pair is not found
    cout << "Pair not found";
}

int main()
{
    int nums[] = { 8, 7, 2, 5, 3, 1 };
    int S = 10;

    int n = sizeof(nums)/sizeof(nums[0]);

    findPair(nums, n, S);

    return 0;
}
```

Output :

```
Pair found is : (2, 8)
```

Time Complexity :

O(N*logN) , where N is the number of elements.

Space Complexity :

O(1).

Question 8)

Example :

Input: *N = 5 and k = 2*

Output: *3*

Explanation: *Firstly, the person at position 2 is dropped out,*

then the person at position 4 is dropped out, then the person at position 1 is dropped out.

Finally, the person at position 5 is dropped. So the person at position 3 will be last one remaining.

Approach :

The problem has the following recursive structure. josephus(N, m) = (josephus(N– 1, m) + m-1) % N + 1

and josephus(1, m) = 1

After the first person (mth from the beginning) is dropped out, N-1 persons are left. Make recursive call for

Josephus(N – 1, m) to get the position with N-1 persons. But the position returned by Josephus(N– 1, m)

will consider the position starting from m%N + 1. So make adjustments to the position returned by Josephus(N – 1, m).

Below is the implementation of the above approach :

```
// C++ code to implement the josephus circle problem

#include <iostream>
using namespace std;

// Recursive function to implement the Josephus circle problem
int josephus(int N, int m)
{
    //invalid inputs
    if (N == 0 || m == 0)
        return 0;
    //base case
    if (N == 1)
        return 1;
    else
        /*The position returned by josephus(N - 1, m)
            is adjusted because the recursive call
            josephus(N - 1, m) considers the
            original position m % N + 1 as position 1. */

        return (josephus(N - 1, m) + m - 1) % N + 1;
}

// Driver code
int main()
{
    int N;
    cout << "Enter the total number of people : ";
    cin >> N ;

    int m;
    cout << "Enter the position of the person you want to eliminate : ";
    cin >> m ;

    cout << "The chosen place is " << josephus(N, m);
    return 0;
}
```

Output :

```
Enter the total number of people : 5
Enter the position of the person you want to eliminate : 2
The chosen place is 3
```

Time Complexity :

O(N)

Space Complexity :

O(n), as space used in N recursive calls stack.

Question 9)

Approach :

- Created a dummy node and point it to the head of input.
- Calculate the length of the linked list which takes O(N) time, where N is the length of the linked list.
- Initialize three pointers prev, curr , next to reverse k elements from every group.
- Iterate over the linkedlist till next != NULL.
- Points curr to the prev -> next and next to the curr -> next.
- Then , using the inner for loop reverse the group using following four steps :
  curr -> next = next -> next
  next -> next = prev -> next
  prev -> next = next
  next = curr -> next
- This for loop runs for k-1 times for all groups except the last remaining element, for the last remaining element it runs for the remaining length of the linkedlist - 1.
- Decrement count after every for loop by count = count-k , to determine the length of the remaining linked list.
- change prev position to curr , prev = curr.

Below is the implementation of the above approach :

```
// C++ program to reverse a linked list in groups of given size

#include <bits/stdc++.h>
using namespace std;
class Node {
public:
    int data;
    Node* next;
};

/* Reverses the linked list in groups of size k and returns the pointer
to the new head node. */
Node* reverse(Node* head, int k)
{
    // If head is NULL or K is 1 then return head
    if (!head || k == 1)
        return head;
    // Create a dummy node
    Node* dummy = new Node();
    dummy->data = -1;
    dummy->next = head;
    // Initializing three points prev, curr, next
    Node *prev = dummy, *curr = dummy, *next = dummy;
    // Calculating the length of linked list
    int count = 0;
    while (curr) {
        curr = curr->next;
        count++;
    }
    // Iterating till next is not NULL
    while (next) {
        // Curr position after every reverse group
        curr = prev->next;
        // Next will always next to curr
        next = curr->next;
        // toLoop will set to count - 1 in case of remaining element
        int toLoop = count > k ? k : count - 1;

        for (int i = 1; i < toLoop; i++) {
            // 4 steps as discussed above
            curr->next = next->next;
            next->next = prev->next;
            prev->next = next;
            next = curr->next;
        }
        // Setting prev to curr
```

```
        prev = curr;
        // Update count
        count -= k;
    }
    // dummy -> next will be our new head for output linkedlist
    return dummy->next;
}

// Function to push a node
void push(Node** head_ref, int new_data)
{
    //allocate node
    Node* new_node = new Node();
    // put in the data
    new_node->data = new_data;
    // link the old list of the new node
    new_node->next = (*head_ref);
    // move the head to point to the new node
    (*head_ref) = new_node;
}

// Function to print linked list
void printList(Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
}

// Driver code
int main()
{
    // Start with the empty list
    Node* head = NULL;
    /* Created Linked list
    is 1->2->3->4->5->6->7->8->9 */
    push(&head, 9);
    push(&head, 8);
    push(&head, 7);
    push(&head, 6);
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);
    cout << "Given linked list \n";
    printList(head);

    //Reverse linked list in group of 3
    head = reverse(head, 3);
    cout << "\nReversed Linked list \n";
    printList(head);
    return (0);
}
```

Output :

```
Given linked list
1 2 3 4 5 6 7 8 9
Reversed Linked list
3 2 1 6 5 4 9 8 7
```

Time Complexity :

O(N), while loop takes O(N/k) times and inner for loop takes O(k) time. So O(N)

Space Complexity :

O(1), no extra space is needed.

Question 10)

Approach :

- Get the count of nodes in the first and second list as n and m respectively.
- Get the difference of counts d = abs(n-m)
- Now traverse the bigger list from the first node to d nodes so that from here both lists have equal no of nodes.
- Then we can traverse both lists in parallel till we come across a common node.(getting

a common node is done by comparing the address of the nodes)

Below is the implementation of the above approach :

```
// C++ program to get merge point of two linked list

#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;
};

/* Takes head pointer of the linked list and
returns the count of nodes in the list */
int getCount(Node* head)
{
    Node* current = head;
    // Counter to store count of nodes
    int count = 0;
    // Iterate till NULL
    while (current != NULL) {
        // Increase the counter
        count++;
        // Move the Node ahead
        current = current->next;
    }
    return count;
}

/* function to get the merge point of two linked
lists list1 and list2 where list1 has d more nodes than
list2 */
int _getMergeNode(int d, Node* list1, Node* list2)
{
    // Stand at the starting of the bigger list
    Node* current1 = list1;
    Node* current2 = list2;
    // Move the pointer forward
    for (int i = 0; i < d; i++) {
        if (current1 == NULL) {
            return -1;
        }
        current1 = current1->next;
    }
    // Move both pointers of both list till they
    // intersect with each other
    while (current1 != NULL && current2 != NULL) {
        if (current1 == current2)
            return current1->data;
        // Move both the pointers forward
        current1 = current1->next;
        current2 = current2->next;
    }
    return -1;
}

/* function to get the merge point of two linked
lists list1 and list2 */
int getMergeNode(Node* list1, Node* list2)
{
    // Count the number of nodes in
    // both the linked list
    int n = getCount(list1);
    int m = getCount(list2);
    int d;
    // If first is greater
    if (n > m) {
        d = n - m;
        return _getMergeNode(d, list1, list2);
    }
    else {
        d = m - n;
        return _getMergeNode(d, list2, list1);
    }
}

// Driver Code
int main()
{
```

```
    /*
        Create two linked lists

        1st 3->6->9->15->30
        2nd 10->15->30

        15 is the merge point
    */
    Node* newNode;

    // Addition of new nodes
    Node* list1 = new Node();
    list1->data = 10;
    Node* list2 = new Node();
    list2->data = 3;

    newNode = new Node();
    newNode->data = 6;
    list2->next = newNode;

    newNode = new Node();
    newNode->data = 9;
    list2->next->next = newNode;

    newNode = new Node();
    newNode->data = 15;
    list1->next = newNode;
    list2->next->next->next = newNode;

    newNode = new Node();
    newNode->data = 30;
    list1->next->next = newNode;
    list1->next->next->next = NULL;

    cout << "The node of merge is " << getMergeNode(list1, list2);
}
```

Output :

```
The node of merge is 15
```

Time Complexity :

O(N+M)

Space Complexity :

O(1)

Question 11)

Approach :

To check whether a linked list is NULL-terminated or cyclic, we can use Floyd's cycle-finding algorithm. This algorithm uses two pointers, a slow pointer that moves one node at a time, and a fast pointer that moves two nodes at a time. If the linked list is NULL-terminated, the fast pointer will reach the end of the list, and the algorithm will terminate. If the linked list is cyclic, the fast pointer will eventually catch up to the slow pointer, and the algorithm will detect the cycle.

Below is the implementation of the above approach :

```
// Q. Check whether the given linked list is either NULL-terminated or ends in cycle(cyclic).

#include<iostream>
using namespace std;

//structure of a node
struct node
{
    int data;
    struct node *next;
};

//function to create a node
struct node *newNode(int data)
{
    struct node *Node = new struct node;
    Node -> data = data;
    Node -> next = NULL;
```

```
    return Node;
}

/*This function takes head pointer as a input and return type is boolean.
  It returns true if the linkedlist is cyclic otherwise false */
bool checkCyclic(struct node *head)
{
    //if the linkedlist is empty then it is also a circular linkedlist
    if(head == NULL)
        return true;

    //for traversing the linklist
    struct node *temp;

    //next node of head
    temp = head -> next;

    //loop will terminated if the linkedlist is circular or it is null ended
    while (temp != NULL && temp != head)
    {
        temp = temp -> next;
    }
    //to check the reason of termination of the loop
    bool resn = temp == head;

    return resn;
}

int main()
{
    //for null linkedlist
    struct node *head = NULL;
    //for NULL terminated linkedlist
    head = newNode(5);
    head -> next = newNode(8);
    head -> next ->next = newNode(6);
    head -> next ->next -> next = newNode(14);
    //for circular linkedlist
    head -> next ->next -> next = head;

    bool check = checkCyclic(head);
    if(check)
        cout<<"This is a cyclic linkedlist."<<endl;

    else
        cout<<"The linkedlist is NULL terminated."<<endl;
    return 0;
}
```

Output :

```
This is a cyclic linkedlist.
```

Time Complexity :

O(N), where n is the number of nodes in the linked list.

Space Complexity :

O(1)

Question 12)

Approach :

start two indexes – one at the beginning of the string and other at the ending of the string. Each time compare whether the values at both the indexes are same or not. If the values are not same then we say that the given string is not a palindrome and break the loop. If the values are same then increment the starting index and decrement the ending index. Continue this process until both the indexes meet at the middle (at X) [this means that it is a palindrome].

Below is the implementation of the above approach :

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

//function to check if the string is Palindrome
int isPalindrome(string str)
```

```
{
    //the first index
    int start_index = 0;

    //the last index
    int last_index = str.length()-1;

    while(start_index < last_index && str[start_index] == str[last_index])
    {
        //increment start index and decrement last index
        start_index++;
        last_index--;
    }

    if(start_index < last_index)
    {
        //this means that we did not reach the center
        cout << "Not Palindrome." << endl;
        return 0;
    }
    else
    {
        //we reached the center
        cout << "Yes , Palindrome." << endl;
        return 1;
    }
}
int main()
{
    isPalindrome("ababXbaba");
    isPalindrome("ababaXaabbb");
    return 0;
}
```

Output :

```
Yes , Palindrome.
Not Palindrome.
```

Time Complexity :

O(N)

Space Complexity :

O(1)

Question 13)

Approach :

- Start with two indexes, one at the left end and other at the right end
- The left index simulates the first stack and the right index simulates the second stack.
- If we want to push an element into the first stack then put the element at left index.
- Similarly, if we want to push an element into the second stack then put the element at the right index.
- First stack grows towards the right and the second stack grows towards left.

Below is the implementation of the above approach :

```
//C++ program to implement 2 stacks in a single array

#include<iostream>
using namespace std;

class twostacks{
        int *arr;
        int top1;
        int top2;
        int size;
    public:
          twostacks(int n){
            this -> size = n;
            top1 = -1;
            top2 = size;
            arr = new int[n];
}
void push1(int element){
```

```
    if(top2 - top1 > 1){
        top1++;
        arr[top1] = element;
    }

}
    void push2(int element){
        if(top2 - top1 > 1){
            top2--;
            arr[top2] = element;
        }

    }
int pop1(){
    if (top1 >= 0){
        int pop_ele1 = arr[top1];
        top1--;
        return pop_ele1;
    }
    else{
        return -1;
    }
}
int pop2(){
    if(top2 < size){
        int pop_ele2 = arr[top2];
        top2--;
        return pop_ele2;
    }
}
};
int main(){
    twostacks st(10);
    st.push1(25);
    st.push1(46);
    st.push1(79);
    st.push1(17);

    st.push2(19);
    st.push2(7);
    st.push2(13);

    cout << "Popped element from stack 1 is " << st.pop1() << endl;
    cout << "Popped element from stack 1 is " << st.pop1() << endl;

    cout << "Popped element from stack 2 is " << st.pop2() << endl;
    return 0;
}
```

Output :

```
Popped element from stack 1 is 17
Popped element from stack 1 is 79
Popped element from stack 2 is 13
```

Time Complexity :

O(N)

Space Complexity :

O(N)

Question 14)

Approach :

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

- enQueue(q, x)
  1) Push x to stack1 (assuming size of stacks is unlimited).
  Here time complexity will be O(1)
- deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
  While stack1 is not empty, push everything from stack1 to stack2.

3) Pop the element from stack2 and return it.
Here time complexity will be O(n)

Below is the implementation of the above approach :

```
// C++ program to implement Queue using two stacks

#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s1, s2;
    // Enqueue an item to the queue
    void enQueue(int x)
    {
        // Push item into the first stack
        s1.push(x);
    }
    // Dequeue an item from the queue
    int deQueue()
    {
        // if both stacks are empty
        if (s1.empty() && s2.empty()) {
            cout << "Queue is empty";
            exit(0);
        }
        // if s2 is empty, move elements from s1
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }
        // return the top item from s2
        int x = s2.top();
        s2.pop();
        return x;
    }
};
// Driver code
int main()
{
    Queue q;
    q.enQueue(8);
    q.enQueue(11);
    q.enQueue(46);
    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
    return 0;
}
```

Output :

```
8
11
46
```

Time Complexity :

- Push Operation: O(1).
- Pop Operation : O(n). In the worst case we have to empty the whole of stack 1 into stack 2 so its O(N)

Space Complexity :

O(n).Use of stack for storing values.

Question 15)

Approach :

The new element is always enqueued to q1. In pop() operation , if q2 is empty then all the elements except the last , are moved to q2. Finally , the last element is dequeued from q1 and returned.

- Follow the below steps to implement the push(s , x) operation :
  Enqueue x to q1 (assuming the size of q1 is unlimited).
- Follow the below steps to implement the pop(s) operation :

a. One by one dequeue everything except the last element from q1 and enqueue to q2.
b. Dequeue the last item of q1 , the dequeued item is the result , store it.
c. Swap the names of q1 and q2.
d. Return the item stored in step 2.

Below is the implementation of the above approach :

```
// C++ Program to implement a stack using two queue

#include <bits/stdc++.h>
using namespace std;

class Stack {
    queue<int> q1, q2;
public:
    void pop()
    {
        if (q1.empty())
            return;
        // Leave one element in q1 and push others in q2.
        while (q1.size() != 1) {
            q2.push(q1.front());
            q1.pop();
        }
        // Pop the only left element from q1
        q1.pop();
        // swap the names of two queues
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
    }
    void push(int x) {
        q1.push(x);
    }
    int top()
    {
        if (q1.empty())
            return -1;
        while (q1.size() != 1) {
            q2.push(q1.front());
            q1.pop();
        }
        // last pushed element
        int temp = q1.front();
        // to empty the auxiliary queue after last operation
        q1.pop();
        // push last element to q2
        q2.push(temp);
        // swap the two queues names
        queue<int> q = q1;
        q1 = q2;
        q2 = q;
        return temp;
    }
    int size() {
        return q1.size();
    }
};

// Driver code
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);
    cout << "current size of stack : " << s.size() << endl;
    cout << "top of stack : " << s.top() << endl;
    s.pop();
    cout << "top of stack after pop : " << s.top() << endl;
    s.pop();
    cout << "top of stack after again pop : " << s.top() << endl;
    cout << "current size of stack after pop : " << s.size() << endl;

    return 0;
}
```

Output :

```
current size of stack : 3
top of stack : 3
```

```
top of stack after pop : 2
top of stack after again pop : 1
current size of stack after pop : 1
```

Time Complexity :

- Push Operation: O(1), as on each operation the new element is added at the end of the queue.
- Pop Operation : O(N). As,on each pop operation ,all the elements are popped out from the queue(q1) except the last element and pushed into the queue(q2).

Space Complexity :

O(N). Since 2 queues are used.

Question 16)

Approach :

a. Create a stack data structure of the integer type and insert/push the elements in it.
b. Create another auxiliary stack data structure of the integer type.
c. Traverse while the original stack is not empty and push the element at top of the auxiliary stack and pop the element from the top of the original stack.
d. Create a variable result of the boolean type and set its value as true.
e. Traverse through the auxiliary stack and pop the top 2 elements in the auxiliary stack and store them in 2 integer variables.
f. Check if the absolute difference of both the integer variables is not equal to 1, update the boolean variable result as false. Push / insert both the integer variables in the original stack.
g. Check if the size of the auxiliary stack is equal to 1, Push/insert the element at the top of the auxiliary stack in the original stack.
h. Return the boolean variable result.
i. Check if the returned value is equal to true, print "Yes" else print "No".
j. Print the original stack.

Below is the implementation of the above approach :

```
/* C++ program to check if successive pair of numbers in the stack are
 consecutive or not */

#include <bits/stdc++.h>
using namespace std;

// Function to check if elements are pairwise consecutive in stack
bool pairWiseConsecutive(stack<int> s)
{
    // Transfer elements of s to aux.
    stack<int> aux;
    while (!s.empty()) {
        aux.push(s.top());
        s.pop();
    }

    // Traverse aux and see if  elements are pairwise consecutive or not.
    bool result = true;
    while (aux.size() > 1) {
        // Fetch current top two elements of aux to check if they are consecutive.
        int x = aux.top();
        aux.pop();
        int y = aux.top();
        aux.pop();

        if (abs(x - y) != 1)
        result = false;

        // Push the elements to original stack.
        s.push(x);
        s.push(y);
    }
    if (aux.size() == 1)
        s.push(aux.top());
    return result;
}
// Driver program
int main()
```

```
{
    stack<int> s;
    s.push(4);
    s.push(5);
    s.push(-2);
    s.push(-3);
    s.push(11);
    s.push(10);
    s.push(5);
    s.push(6);
    s.push(20);

    if (pairWiseConsecutive(s))
        cout << "Yes" << endl;
    else
        cout << "No" << endl;

    cout << "Stack content (from top) after function call\n";

    while (s.empty() == false)
    {
    cout << s.top() << " ";
    s.pop();
    }
    return 0;
}
```

Output :

```
Yes
Stack content (from top) after function call
20 6 5 10 11 -3 -2 5 4
```

Time Complexity :

O(n), where n is the number of elements in the stack.

Space Complexity :

O(n), we have used a stack to store n elements.

Question 17)

Approach :

- Create a deque to store K elements.
- Run a loop and insert the first K elements in the deque. Before inserting the element, check if the element at the back of the queue is smaller than the current element, if it is so remove the element from the back of the deque until all elements left in the deque are greater than the current element. Then insert the current element, at the back of the deque.
- Now, run a loop from K to the end of the array.
- Print the front element of the deque.
- Remove the element from the front of the queue if they are out of the current window.
- Insert the next element in the deque. Before inserting the element, check if the element at the back of the queue is smaller than the current element, if it is so remove the element from the back of the deque until all elements left in the deque are greater than the current element. Then insert the current element, at the back of the deque.
- Print the maximum element of the last window.

Below is the implementation of the above approach :

```
#include <bits/stdc++.h>
using namespace std;

/* A Dequeue based method for printing maximum element of
 all subarrays of size k*/
void printKMax(int arr[], int N, int w)
{
    // Create a Double Ended Queue Qi that will store indexes
    // of array elements.The queue will store indexes
    // of useful elements in every window and it will
    // maintain decreasing order of values from front to rear in Qi,
    // arr[Qi.front[]] to arr[Qi.rear()] are sorted in decreasing order
    deque<int> Qi(w);

    //Process first k (or first window) elements of array
    int i;
    for (i = 0; i < w; ++i) {
```

```
        /* For every element, the previous smaller elements are useless so
         remove them from Qi */
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            // Remove from rear
            Qi.pop_back();

        // Add new element at rear of queue
        Qi.push_back(i);
    }

    // Process rest of the elements, from arr[k] to arr[n-1]
    for (; i < N; ++i) {

        //The element at the front of the queue is the largest element of previous window, so print it
        cout << arr[Qi.front()] << " ";

        // Remove the elements which are out of this window
        while ((!Qi.empty()) && Qi.front() <= i - w)

            // Remove from front of queue
            Qi.pop_front();

        // Remove all elements smaller than the currently being added element(remove useless elements)
        while ((!Qi.empty()) && arr[i] >= arr[Qi.back()])
            Qi.pop_back();

        // Add current element at the rear of Qi
        Qi.push_back(i);
    }
    // Print the maximum element of last window
    cout << arr[Qi.front()];
}
// Driver code
int main()
{
    int arr[] = {1, 3, -1, -3, 5, 3, 6, 7};
    int N = sizeof(arr) / sizeof(arr[0]);
    int w = 3;
    // Function call
    printKMax(arr, N, w);
    return 0;
}
```

Output :

```
3 3 5 5 6 7
```

Time Complexity :

O(N). It seems more than O(N) at first look. It can be observed that every element of the array is added and removed at most once.

Space Complexity :

O(K). Elements stored in the dequeue takes O(k) space.

Question 18)

If the queue is implemented as a circular array with n-1 locations available for storing elements , then the maximum number of elements that can be stored in the queue is n-1. We can use the following formula to calculate the number of elements in the queue:

Number of elements = (rear - front + n) % n

Here , "%" is the module operator, which gives the remainder when the first operand is divided by the second operand. The formula

works as follows :

- (rear - front gives the number of elements between the front and rear pointers .
- If rear >= front , this number is actual number of elements in the queue.
- If rear < front , then the rear pointer has wrapped around to the beginning of the array and the number of elements is n - (front - rear).
- Adding n to the result ensures that the number of elements is always positive , even if the rear pointer has wrapped around to the
- beginning of the array.
- Taking the result modulo n ensures that the number of elements is always 0 and n-1, inclusive.

Note that if the number of elements is n-1 , the queue is either full or empty, depending on whether the front pointer is one position behind the rear pointer or the front and rear pointers are equal.

Question 19)

Approach :

a. *Create an empty stack.*
b. *One by one dequeue first K items from given queue and push the dequeued items to stack*
c. *Enqueue the contents of stack at the back of the queue*
d. *Dequeue (size-k) elements from the front and enqueue them one by one to the same queue.*

Below is the implementation of the above approach :

```
//C++ Program to reverse the first K elements of Queue :
#include<bits/stdc++.h>
using namespace std;

 //Function to reverse the first K elements of Queue
void reverseFirstKelements(int k, queue<int>& q) {

    //Value of k should be positive
    if (k <= 0)
        return;
    // If Queue is empty or value of k is bigger than size of queue then return
    if (q.empty() ==true || k > q.size())
        return;
    stack<int>  s;
    //Pushing first k elements into a stack
    for (int i = 0; i < k; i++) {
        s.push(q.front());
        q.pop();
    }

    //Enqueue the elements at the back of the queue
    while (s.empty()==false) {
        q.push(s.top());
        s.pop();
    }
    //Remove the remaining elements and enqueue them at the end of the queue
    for (int i = 0 ; i < q.size() - k ; i++) {
        q.push(q.front());
        q.pop();
    }
}
//Function to print the queue
void Print(queue<int>& q) {
    while (q.empty()== false) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;
}
int main()
{
    queue<int> q;
    q.push(10);
    q.push(20);
    q.push(30);
    q.push(40);
    q.push(50);
    q.push(60);
    q.push(70);
    q.push(80);
    q.push(90);

    int k = 4;
    reverseFirstKelements(k, q);

    //Printing Queue
    Print(q);
    return 0;
}
```

Output :

```
40 30 20 10 50 60 70 80 90
```

Time Complexity :

O(n+k), where 'n' is the total number of elements in the queue and 'k' is the number of elements to be reversed. This is because firstly the whole queue is emptied into the stack and after that first k elements are emptied and enqueued in the same way.

Space Complexity :

O(k), where k is no of elements to be reversed since stack is being used to store values for the purpose of reversing.

Question 20)

Approach :

1) Get the Middle of the linked list and make it root.
2) Recursively do same for left half and right half.
   a) Get the middle of left half and make it left child of the root created in step 1.
   b) Get the middle of right half and make it right child of the root created in step 1.

Below is the algorithm of the above approach in pseudo code :

```
function findMiddleNode(head):
    slow = head
    fast = head

    // move fast pointer two steps at a time
    // and slow pointer one step at a time
    while fast.next is not null and fast.next.next is not null:
        fast = fast.next.next
        slow = slow.next

    // now slow points to middle node
    return slow


function convertDLLToBST(head):
    // base case: empty list
    if head is null:
        return null

    // find middle node
    mid = findMiddleNode(head)

    // create root node
    root = new TreeNode(mid.value)

    // recursively construct left and right subtrees
    root.left = convertDLLToBST(head)
    root.right = convertDLLToBST(mid.next)

    return root
```

Time Complexity :

O(NlogN), where N is the number of nodes in the linked list.

Space Complexity :

O(NlogN),because at each level of recursion, the algorithm creates a new node for the middle element of the linked list, and then recursively constructs two subtrees from the remaining elements. Therefore, the maximum number of nodes that can exist in the recursion call stack at any given time is O(log n), and the maximum number of nodes that can be created by the algorithm is O(n).

Question 21)

Approach :

1) Get the Middle of the linked list and make it root.

2) Recursively do same for left half and right half.
   a) Get the middle of left half and make it left child of the root created in step 1.
   b) Get the middle of right half and make it right child of the root created in step 1.

Below is the implementation of the above approach :

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

struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

TreeNode* sortedListToBST(ListNode* head) {
    if (!head) {
        return NULL;
    }
    // find middle node
    ListNode* slow = head;
    ListNode* fast = head;
    ListNode* prev = NULL;
    while (fast && fast->next) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }
    // create root node
    TreeNode* root = new TreeNode(slow->val);
    // handle left subtree
    if (prev) {
        prev->next = NULL;
        root->left = sortedListToBST(head);
    }
    // handle right subtree
    if (slow->next) {
        root->right = sortedListToBST(slow->next);
    }
    return root;
}

void preOrder(TreeNode* root){
    if(root == NULL) return;
    cout<< root-> val <<" ";
    preOrder(root->left);
    preOrder(root->right);
}

int main() {

    // create a sorted linked list
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);
    head->next->next->next->next->next = new ListNode(6);

    // convert linked list to binary search tree
    TreeNode* root = sortedListToBST(head);

    // print preorder traversal of binary search tree
    cout << "preorder traversal of binary search tree:" << endl;

    preOrder(root);

    cout << endl;
    return 0;
}
```

Output :

```
Preorder traversal of binary search tree:
4 2 1 3 6 5
```

Time Complexity :

O(NlogN), where N is the number of nodes in the linked list.

Space Complexity :

O(log N), where N is the number of nodes in the linked list.

Question 22)

Below is the pseudo code of the algorithm :

```
void Print(node *root, int k1, int k2)
{
	// base case
	if ( NULL == root )
		return;

	// Since the desired o/p is sorte, recurse for left subtree first
	Print(root->left, k1, k2);

	// if root's data lies in range, then prints root's data
	if ( k1 <= root->data && k2 >= root->data )
		cout<<root->data<<" ";

	// recursively call the right subtree
	Print(root->right, k1, k2);
}

// Utility function to create a new Binary Tree node
node* newNode(int data)
{
	node *temp = new node();
	temp->data = data;
	temp->left = NULL;
	temp->right = NULL;

	return temp;
}

// Driver code
int main()
{
	node *root = new node();
	int k1 = 10, k2 = 25;

	/* Constructing tree given
	in the above figure */
	root = newNode(20);
	root->left = newNode(8);
	root->right = newNode(22);
	root->left->left = newNode(4);
	root->left->right = newNode(12);

	Print(root, k1, k2);
	return 0;
}
```

Output :

```
12 20 22
```

Time Complexity :

O(N), where N is the total number of keys in the tree, A single traversal of the tree is needed.

Space Complexity :

O(H),. where H is the height of the tree in recursion call stack

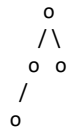
Question 23)

Below are the minimal AVL trees of height 0, 1, 2, 3, 4, and 5 :

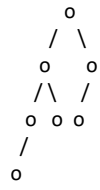
- **Height 0:**

- **Height 1 :**

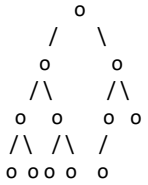
- **Height 2 :**



- **Height 3 :**



- **Height 4 :**



The number of nodes in a minimal AVL tree of height 6 :

The number of nodes in a minimal AVL tree of height h can be calculated using the recurrence relation:

- **N(h) = N(h-1) + N(h-2) + 1 , where N(1) = 2 , N(0) = 1.**

Substituting H = 6 in the recursive relation, we get-

N(6) = N(6-1) + N(6-2) + 1

N(6) = N(5) + N(4) + 1

N(6) = 20 + 12 + 1 (Using (3) and (4))

∴ N(6) = 33 .....(5)

So, minimum number of nodes required to construct AVL tree of height-6 = 33.

Question 24)

Approach :

To count the number of nodes in the range [a, b] in an AVL tree, we can use the following approach:

a. Traverse the AVL tree in in-order, which will give us the nodes in ascending order.
b. While traversing the tree, if we encounter a node with a value less than a, we can ignore its left subtree and continue to traverse the right subtree.
c. If we encounter a node with a value greater than b, we can ignore its right subtree and continue to traverse the left subtree.
d. If we encounter a node with a value between a and b (inclusive), we can increment the count and continue to traverse both its left and right subtrees.

Below is the algorithm of the above approach :

```
int countNodesInRange(AVLNode* root, int a, int b) {
    int count = 0;
    if (root == NULL) {
        return 0;
    }
    if (root->value < a) {
        count += countNodesInRange(root->right, a, b);
    }
    else if (root->value > b) {
        count += countNodesInRange(root->left, a, b);
    }
    else {
        count += 1;
        count += countNodesInRange(root->left, a, b);
        count += countNodesInRange(root->right, a, b);
    }
    return count;
}
int main() {
    AVLNode* root = new AVLNode{ 10, NULL, NULL, 1 };
    root->left = new AVLNode{ 5, NULL, NULL, 1 };
    root->right = new AVLNode{ 15, NULL, NULL, 1 };
    root->left->left = new AVLNode{ 3, NULL, NULL, 1 };
    root->left->right = new AVLNode{ 7, NULL, NULL, 1 };
    root->right->left = new AVLNode{ 12, NULL, NULL, 1 };
    root->right->right = new AVLNode{ 18, NULL, NULL, 1 };
    int a = 5, b = 15;
    int count = countNodesInRange(root, a, b);
    cout << "Number of nodes in the range [" << a << ", " << b << "]: " << count << endl;
}
```

Output :

```
Number of nodes in the range [5, 15]: 5
```

Time Complexity :

O(H + N) , where H is the height of BST and N is the number of nodes in the given range.

Space Complexity :

O(N)

Question 25)

Approach :

For each node there can be four ways that the max path goes through the node:

- Node only
- Max path through Left Child + Node
- Max path through Right Child + Node
- Max path through Left Child + Node + Max path through Right Child

The idea is to keep track of four paths and pick up the max one in the end. An important thing to note is, that the root of every subtree needs to return the maximum path sum such that at most one child of the root is involved. This is needed for the parent function call. In the below code, this sum is stored in 'max_single' and returned by the recursive function.

Below is the implementation of the above approach :

```
//C++ program to find maximum path sum in Binary Tree
```

```cpp
#include <bits/stdc++.h>
using namespace std;
// A binary tree node
struct Node {
    int data;
    struct Node *left, *right;
};
// A utility function to allocate a new node
struct Node* newNode(int data)
{
    struct Node* newNode = new Node;
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return (newNode);
}
// This function returns overall maximum path sum in 'res'
// And returns max path sum going through root.
int findMaxUtil(Node* root, int& res)
{
    // Base Case
    if (root == NULL)
        return 0;
    // l and r store maximum path sum going through left and
    // right child of root respectively
    int l = findMaxUtil(root->left, res);
    int r = findMaxUtil(root->right, res);
    // Max path for parent call of root. This path must
    // include at-most one child of root
    int max_single
        = max(max(l, r) + root->data, root->data);
    // Max Top represents the sum when the Node under
    // consideration is the root of the maxsum path and no
    // ancestors of root are there in max sum path
    int max_top = max(max_single, l + r + root->data);
    res = max(res, max_top); // Store the Maximum Result.
    return max_single;
}
// Returns maximum path sum in tree with given root
int findMaxSum(Node* root)
{
    // Initialize result
    int res = INT_MIN;
    // Compute and return result
    findMaxUtil(root, res);
    return res;
}
// Driver code
int main(void)
{
    struct Node* root = newNode(10);
    root->left = newNode(2);
    root->right = newNode(10);
    root->left->left = newNode(20);
    root->left->right = newNode(1);
    root->right->right = newNode(-25);
    root->right->right->left = newNode(3);
    root->right->right->right = newNode(4);
    // Function call
    cout << "Max path sum is " << findMaxSum(root);
    return 0;
}
```

Output :

```
Max path sum is 42
```

Time Complexity :

O(N), where N is the number of nodes in the Binary Tree.

Space Complexity :

O(N)

Question 26)

Approach :

To find the 10 maximum numbers from a big file containing billions of numbers, the approach could be :

- **Divide the file into smaller chunks:**
  We can divide the file into smaller chunks, each containing a fixed number of numbers. The size of the chunks will depend on the available memory and the total size of the file. We can read each chunk into memory, sort it, and write it to a temporary file.

- **Sort each chunk of numbers:**
  We can use an efficient sorting algorithm like QuickSort or MergeSort to sort each chunk of numbers in memory. After sorting, we can write the sorted numbers back to the temporary file.

- **Merge the sorted chunks:**
  We can merge the sorted chunks using an algorithm similar to the merge step of MergeSort. We can maintain a heap of the first element from each chunk and repeatedly extract the minimum element from the heap until all the numbers are merged into a single sorted list.

- **Extract the 10 highest numbers:**
  After merging all the sorted chunks into a single sorted list, we can extract the 10 highest numbers from the end of the list.

This approach is efficient because it minimizes the amount of data that needs to be held in memory at any given time, while still sorting the entire dataset. It can handle very large files, but the performance will depend on the size of the chunks and the available memory.

Time & Space Complexity :

The time and space complexity of the above approach can be analyzed as follows:

- Dividing the file into smaller chunks:

The time complexity of this step is O(N), where N is the total number of numbers in the file. The space complexity depends on the size of each chunk, which can be adjusted based on the available memory.

- Sort each chunk of numbers:

The time complexity of sorting each chunk is O(M log M), where M is the size of each chunk. The space complexity of this step is O(M), which is the size of the chunk.

- Merge the sorted chunks:

The time complexity of merging the sorted chunks is O(N log K), where K is the number of chunks. The space complexity of this step is O(M), which is the size of the heap.

- Extract the 10 highest numbers:

The time complexity of extracting the 10 highest numbers is O(1).

Overall, the time complexity of the entire approach is **O(N log K),** where K is the number of chunks, and the space complexity is O(M), where M is the size of each chunk. The actual time and space complexity will depend on the specific values of N, K, and M, as well as the available memory and the efficiency of the sorting algorithm used.

Question 30)

Total number of possible Binary Search Trees with n different keys = catalan(n) = **2nCn / (n + 1)**

For n = 1 --> 1 Binary Search Tree is possible.
For n = 2 --> 2 Binary Search Trees are possible.
For n = 3 --> 5 Binary Search Trees are possible.
For n = 4 --> 14 Binary Search Trees are possible.
For n = 5 --> 42 Binary Search Trees are possible.
For n = 6 --> 132 Binary Search Trees are possible.