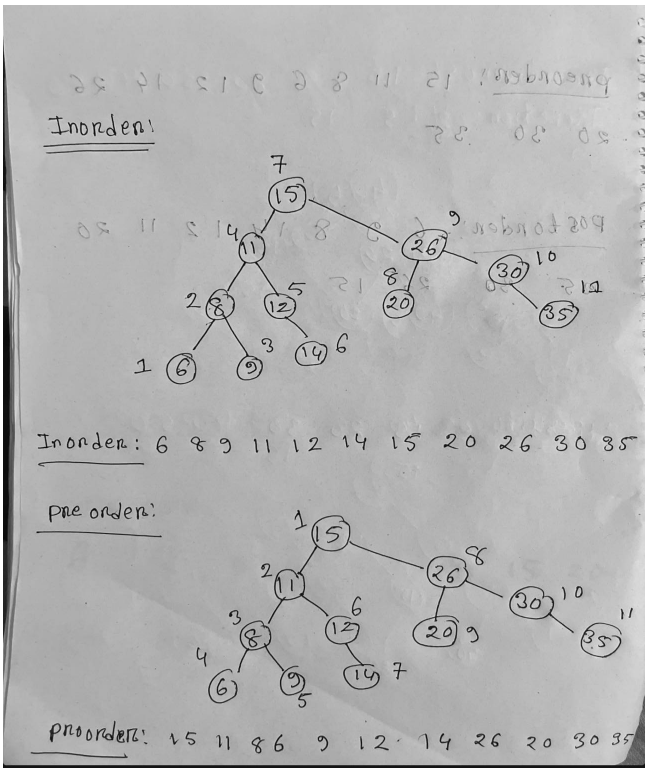
Ans to the question no : 01

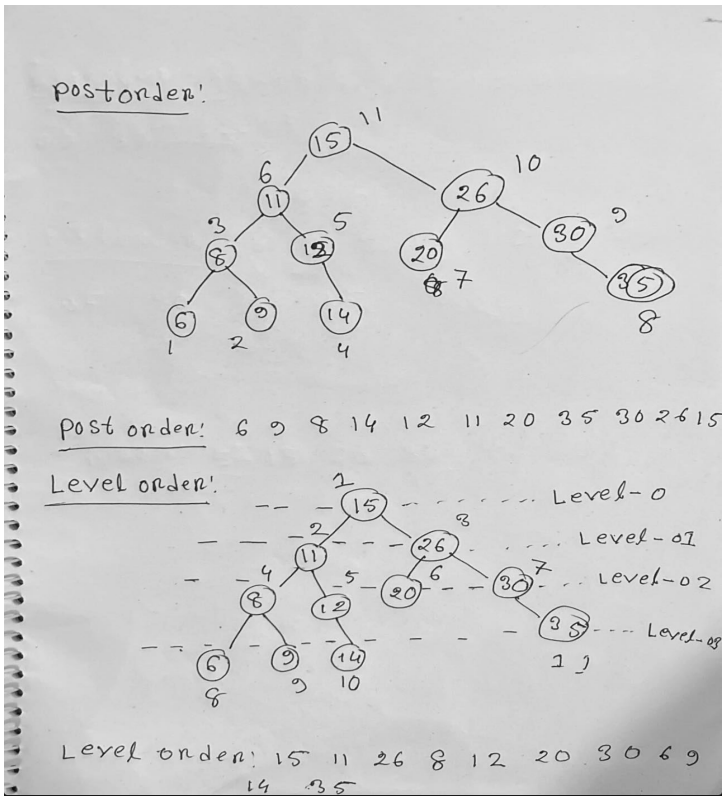
Preorder traversal of binary tree is
15 11 8 6 9 12 14 26 20 30 35
Inorder traversal of binary tree is
6 8 9 11 12 14 15 20 26 30 35
Postorder traversal of binary tree is
6 9 8 14 12 11 20 35 30 26 15
Level Order traversal of binary tree is
15 11 26 8 12 20 30 6 9 14 35

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

struct Node
{
  int data;
  struct Node *left, *right;
};

Node *newNode(int data)
{
  Node *temp = new Node;
  temp->data = data;
  temp->left = temp->right = NULL;
  return temp;
}

void printPostorder(struct Node *node)
{
  if (node == NULL)
    return;

  printPostorder(node->left);

  printPostorder(node->right);
```

```
  cout << node->data << " ";
}

void printInorder(struct Node *node)
{
  if (node == NULL)
    return;

  printInorder(node->left);

  cout << node->data << " ";

  printInorder(node->right);
}

void printPreorder(struct Node *node)
{
  if (node == NULL)
    return;

  cout << node->data << " ";

  printPreorder(node->left);

  printPreorder(node->right);
}

int main()
{
  struct Node *root = newNode(15);
  root->left = newNode(11);
  root->right = newNode(26);

  root->left->left = newNode(8);
  root->right->left = newNode(20);

  root->right->right = newNode(30);
  root->left->right = newNode(12);
  root->left->left->left = newNode(6);
```

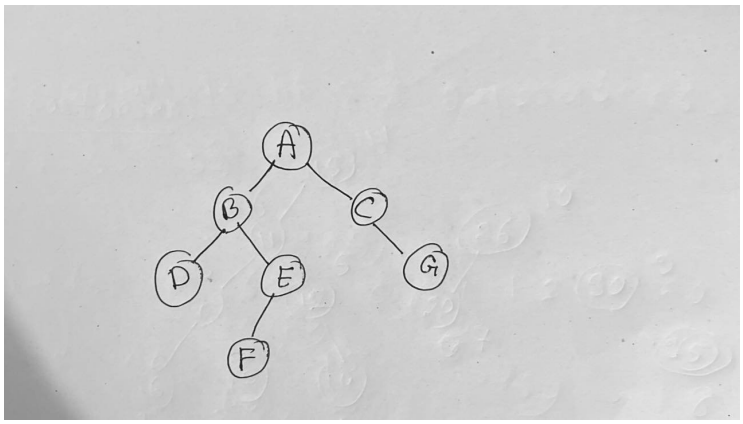
```
  root->left->left->right = newNode(9);
  root->right->right->right = newNode(35);
  root->left->right->right = newNode(14);

  cout << "\nPreorder traversal of binary tree is \n";
  printPreorder(root);

  cout << "\nInorder traversal of binary tree is \n";
  printInorder(root);

  cout << "\nPostorder traversal of binary tree is \n";
  printPostorder(root);

  return 0;
}
```



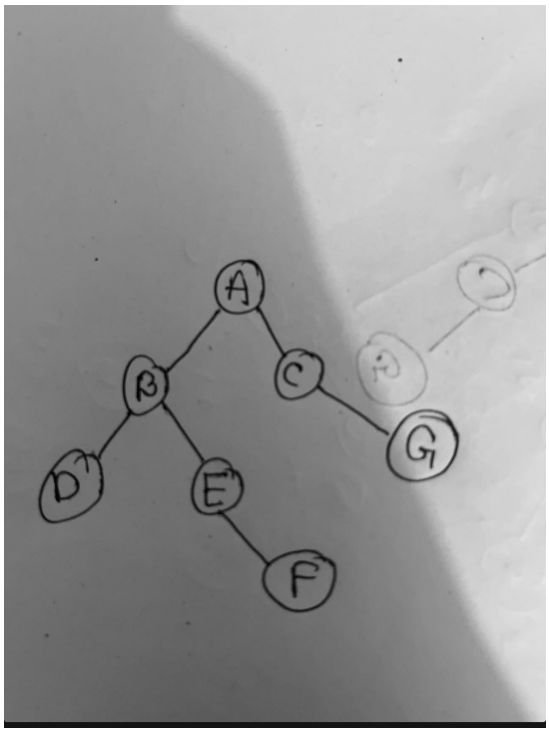


Ans to the question no : 2

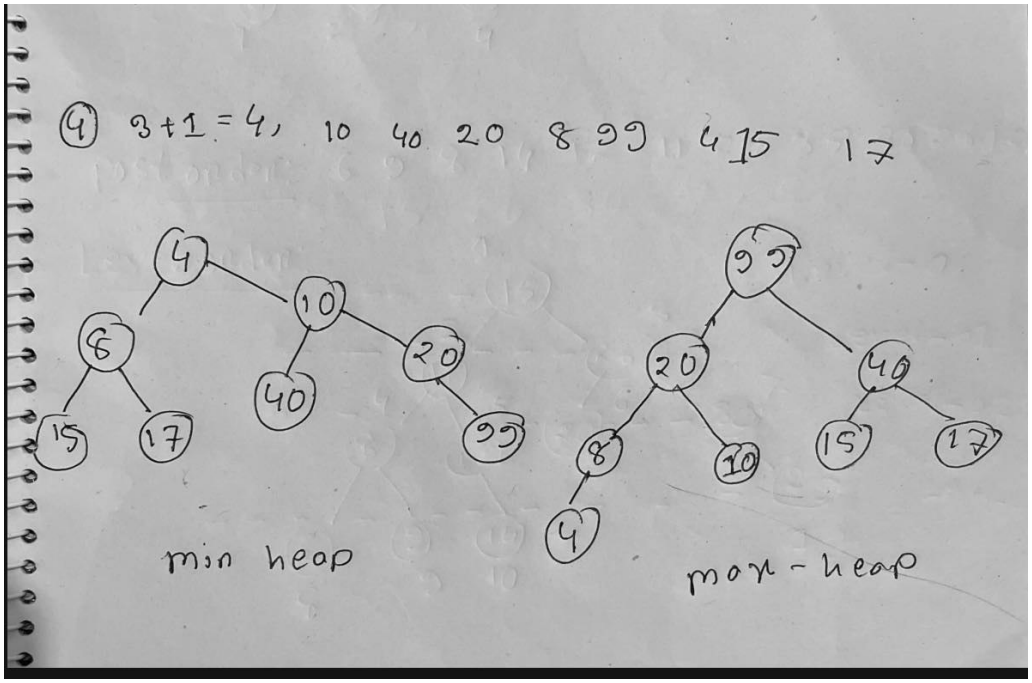
After traversing preorder and inorder the tree is same no change



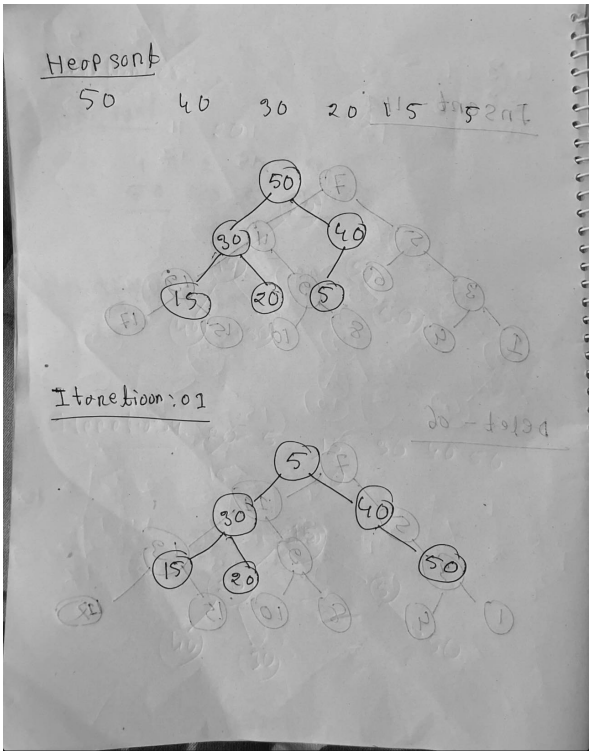
Ans to the question no : 3

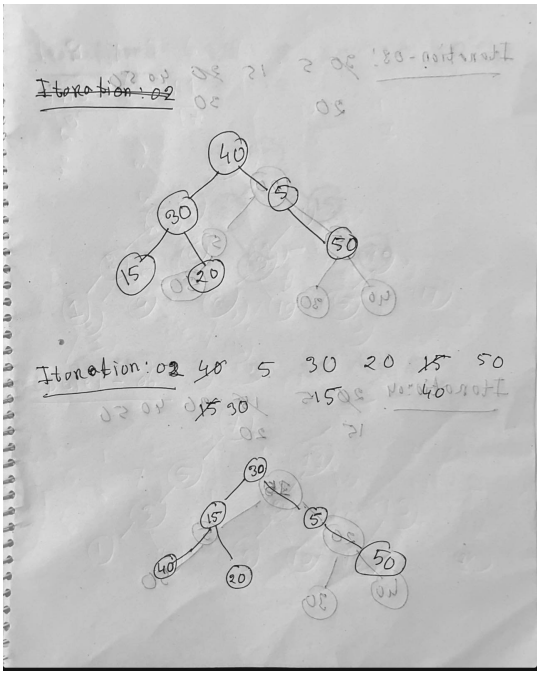


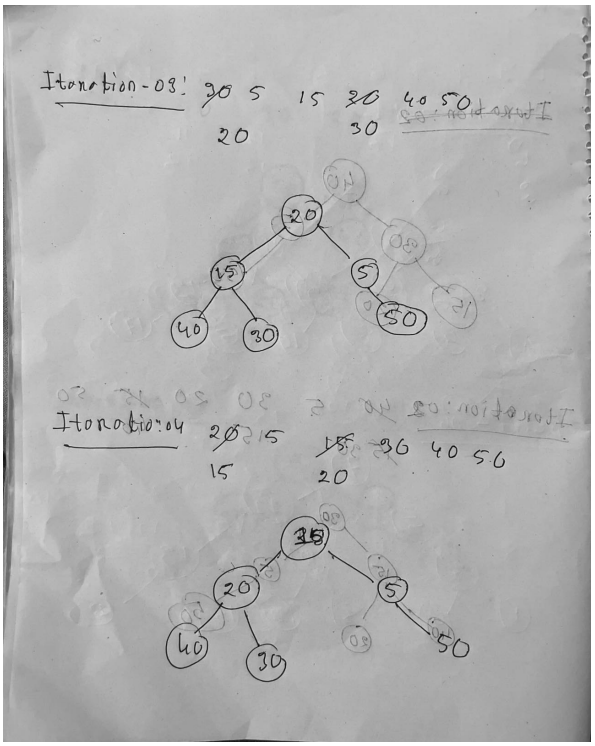
Ans to the question no : 4

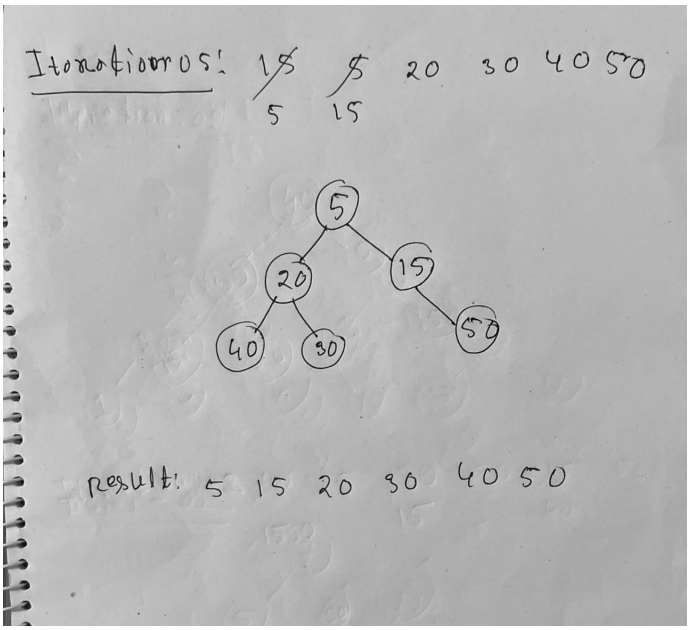


Ans to the question no : 5







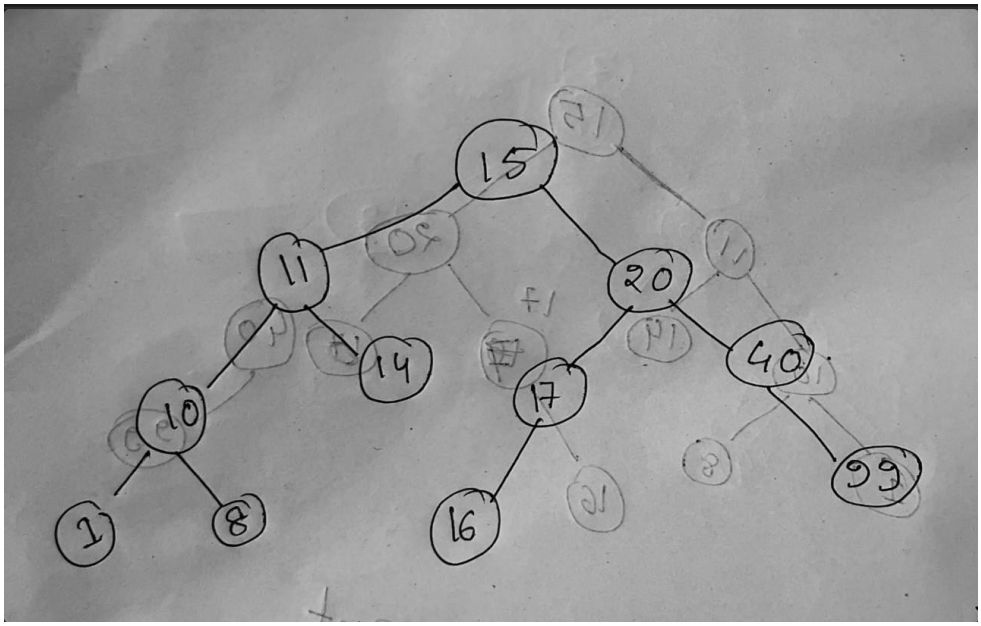


Ans to the question no : 6

In a Binary Search Tree (BST), all keys in the left subtree of a key must be smaller and all keys in the right subtree must be greater. So a Binary Search Tree by definition has distinct keys and duplicates in binary search trees are not allowed.

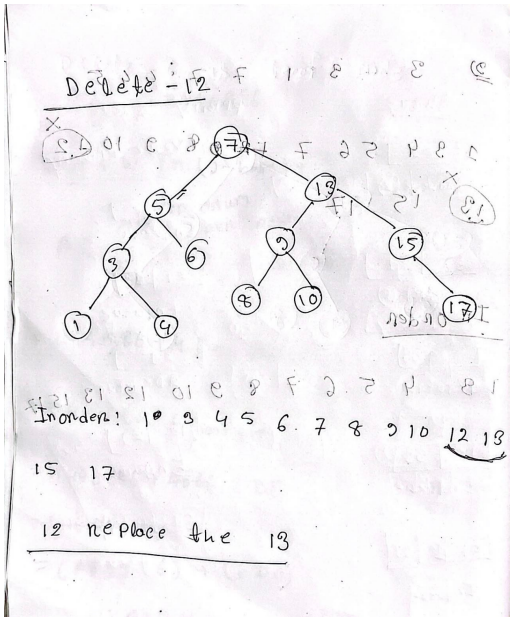
Given data : 10 40 20 8 99 16 15 17 11 14 1

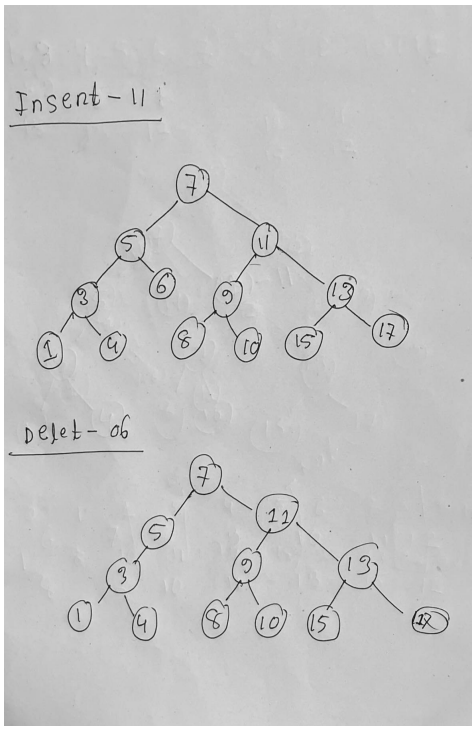
Sorted data : 1 8 10 11 14 15 16 17 20 40 99



Ans to the question no : 7

After Delete 12 inorder successor 13 replace the node position





After Insert 11 and Delete 6

Ans to the question no : 8

Ans to the question no : 9

STEP :

Before sort : 3 3 1 7 7 4 4 5

Maximum : 7

Frequency : 0 1 0 2 2 1 0 2

Cumulative Sum : 0 1 1 3 5 6 6 8 (i=i+1)

After sort : 1 3 3 4 4 5 7 7

The reason of backward traversing :

After Traversing the Original array , we prefer from last Since we want to add Elements in their proper position so when we subtract the index , the Element will be added to lateral position.
But if we start traversing from the beginning , then there will be no meaning for taking the cumulative sum since we are not adding according to the Elements placed. We are adding hap -hazardly which can be done even if we do not take their cumulative sum.

Ans to the question no : 10

Heap sort

- Worst case time complexity of O(n(log(n)) [all elements in the list are distinct]
- Best case time complexity of O(n) [all elements are same]
- Average case time complexity of O(n(log(n))

- Space complexity of O(1)

Counting Sort

- Time complexity: O(N+K)
- Space Complexity: O(K)
- Worst case: when data is skewed and range is large
- Best Case: When all elements are same
- Average Case: O(N+K) (N & K equally domi

Conclusion : HeapSort is more efficient than Counting Sort for space complexity and memory complexity .

Ans to the question no : 11

By row major order:
If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

Given,
loc(A[0][0])= (AE92F6)H
decimal A[0][0] = 11440886

Address(a[i][j]) = Base_address + (i * n + j) * size
Given,
array = A[15] [20]
m = 15
n = 20
base = 11440886
data type = 4 bytes.

address of A[15] [20] = 11440886 + (0*20 + 0)*4 = 11440886 bytes.

By Column major order:
If array is declared by a[m][n] where m is the number of rows while n is the number of columns, then address of an element a[i][j] of the array stored in row major order is calculated as,

Address(a[i][j]) = Base_address + ((j*m) +i)*size
A[50][100]

address of A[50][100] = 11440886 + ((0 *50) + 0) *4 = 11440886 bytes.

Ans to the question no : 12

a)28

b)53

c)NULL

d)16

e)25

Ans to the question no : 13

Advantages Of Doubly Link List:

- Reversing the doubly linked list is very easy.
- It can allocate or reallocate memory easily during its execution.
- As with a singly linked list, it is the easiest data structure to implement.
- The traversal of this doubly linked list is bidirectional which is not possible in a singly linked list.
- Deletion of nodes is easy as compared to a Singly Linked List. A singly linked list deletion requires a pointer to the node and previous node to be deleted but in the doubly linked list, it only requires the pointer which is to be deleted.

Disadvantages Of Doubly Link List:

- It uses extra memory when compared to the array and singly linked list.

- Since elements in memory are stored randomly, therefore the elements are accessed sequentially no direct access is allowed.
- Complete Interview Preparation - GFG
-

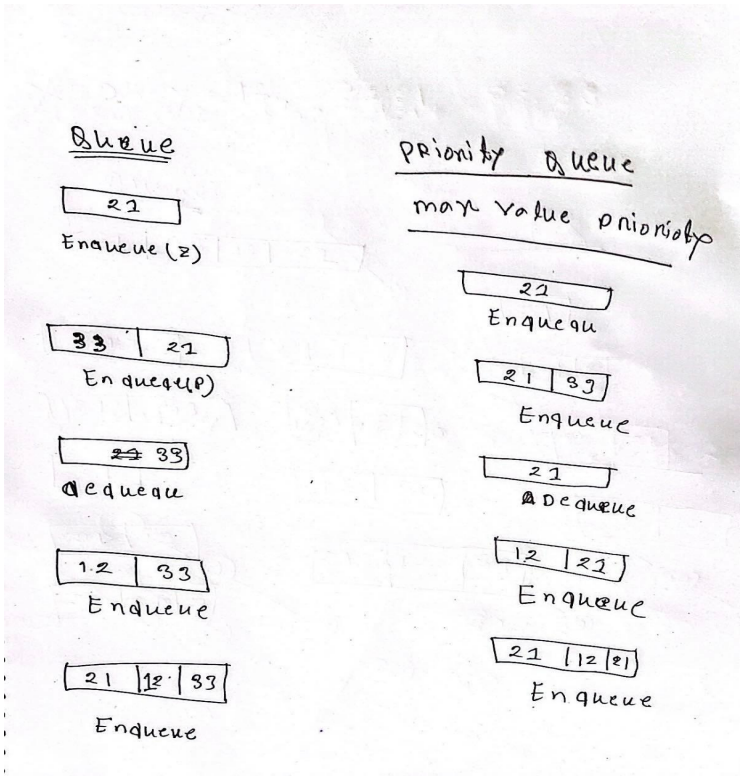
Uses Of Doubly Link List:

- It is used in the navigation systems where front and back navigation is required.
- It is used by the browser to implement backward and forward navigation of visited web pages that is a back and forward button.
- It is also used to represent a classic game deck of cards.
- It is also used by various applications to implement undo and redo functionality.
- Doubly Linked List is also used in constructing MRU/LRU (Most/least recently used) cache.
- Other data structures like stacks, Hash Tables, Binary trees can also be constructed or programmed using a doubly-linked list.
- Also in many operating systems, the thread scheduler(the thing that chooses what process needs to run at which time) maintains a doubly-linked list of all processes running at that time.
-

Ans to the question no : 14

Here ,X=9, Y=12, Z= 21, P=33

Show the queue



Ans to the question no : 15

We can implement circular queues of static or fix size using arrays. Here is the logic of how you can do it: Define a maximum size(N) of the queue and its two pointers- front pointer and rear pointer. For enqueue operation, we check if the circular queue is full. We exit if the queue is fully occupied. Next we check if the queue is empty and the element to be inserted will be the first element of the queue. If true, the front pointer is incremented to point to the index of the first element. As stated above, we increment the rear pointer and then insert the new element to the circular queue. For dequeue operation, we check if the circular queue is empty, that is if the front pointer is NULL. If the queue is empty we simply exit. Otherwise we fetch the rear element and store it in a temporary variable named deleted_item. We check if the element to be deleted is the last element in the queue. If true, we set the queue pointers to -1 which implies that the queue is now empty (empty condition: front == -1). Else we increment the front pointer and return the deleted item which we stored in the temporary variable, deleted_item. Please note, we do not need to delete values from the queue by assigning garbage values or shifting the elements to one side because only those which lie between the

front pointer and the rear pointer are considered as the elements of the queue and the rest which are left behind after deletion are garbage to us.

we check if the queue is Full , If the queue is full, we cannot insert a new element and we raise a Queue Overflow exception and exit.

deQueue(): This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position according to FIFO principle. Following are the steps for dequeue:
The first step is very obvious. We check if the queue is Empty, that is check if F==-1. If the circular queue is empty we cannot dequeue an element from it and we raise an exception for the Underflow condition and exit.

Ans to the question no : 16

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


struct Node
{
    int data;
    Node *left, *right;

    Node(int data)
    {
        this->data = data;
        this->left = this->right = nullptr;
    }
};


void preorder(Node* root)
{
    if (root == nullptr) {
        return;
    }
```

```
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}


void invertBinaryTree(Node* root)
{
    if (root == nullptr) {
        return;
    }


    swap(root->left, root->right);


    invertBinaryTree(root->left);


    invertBinaryTree(root->right);
}

int main()
{

    Node* root = new Node(4);
    root->left = new Node(2);
    root->right = new Node(7);
    root->left->left = new Node(1);
    root->left->right = new Node(3);
    root->right->left = new Node(6);
    root->right->right = new Node(9);

    invertBinaryTree(root);
    preorder(root);

    return 0;
}
```

Ans to the question no : 17

a)

```
void boundaryTravarsal(treeNode *root)
{
    if (root == NULL)
        return;
    cout << root->data << " ";

    printLeftNonLeaves(root->leftchild);
    printLeaves(root->leftchild);
    printLeaves(root->rightchild);
    printRightNonLeaves(root->rightchild);
}

int main()
{

    int n;
    cin >> n;

    treeNode *allNodes[n];

    for (int i = 0; i < n; i++)
    {
        allNodes[i] = new treeNode(-1);
    }

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

        int value, left, right;
        cin >> value >> left >> right;
        allNodes[i]->data = value;

        if (left > n - 1 || right > n - 1)
```

```
        {
            cout << " Invalid Index " << endl;
            break;
        }
        if (left != -1)
        {
            allNodes[i]->leftchild = allNodes[left];
        }

        if (right != -1)
        {
            allNodes[i]->rightchild = allNodes[right];
        }
    }
```

b)

```
void zigzagTravarsal(treeNode *root)
{

    stack<treeNode *> currentLevel;
    stack<treeNode *> nextLevel;

    bool lefttoRight = true;

    currentLevel.push(root);

    while (!currentLevel.empty())
    {

        treeNode * x = currentLevel.top();
        currentLevel.pop();

        cout<<x->data << " ";

        if (lefttoRight)
        {
```

```
            if (x->leftchild)
            {
                nextLevel.push(x->leftchild);
            }
            if (x->rightchild)
            {
                nextLevel.push(x->rightchild);
            }
        }
        else
        {
            if (x->rightchild)
            {
                nextLevel.push(x->rightchild);
            }
            if (x->leftchild)
            {
                nextLevel.push(x->leftchild);
            }
        }
        if (currentLevel.empty())
        {
            lefttoRight != lefttoRight;
            swap(currentLevel, nextLevel);
        }
    }
}
```