Q.2 Write short note on Extended Binary Tree.

Extended Binary Tree
- Extended binary tree consists of replacing every null sub tree of the original tree with special nodes.
- Empty circle represents internal node and filled circle represents external node.
- The nodes from the original tree are internal nodes and the special nodes are external nodes.
- Every internal node in the extended binary tree has exactly two children and every external node is a leaf. It displays the result which is a **complete binary tree**.

![Fig. Extended Binary Tree](image)

Q.3 Write shorts notes on threaded binary Tree.

- A threaded binary tree defined as follows:
- "A binary tree is threaded by making all right child pointers that would normally be null point to the in-order successor of the node (if it exists), and all left child pointers that would normally be null point to the in-order predecessor of the node."[1]
- This definition assumes the traversal order is the same as in-order traversal of the tree. However, pointers can instead (or in addition) be added to tree nodes, rather than
replacing linked lists thus defined are also commonly called "threads", and can be used to enable traversal in any order(s) desired. For example, a tree whose nodes represent information about people might be sorted by name, but have extra threads allowing quick traversal in order of birth date, weight, or any other known characteristic.

- A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
- There are two types of threaded binary trees.
  - **Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)
  - **Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.
- Let's make the Threaded Binary tree out of a normal binary tree:

![Diagram](image)

The in-order traversal for the above tree is — D B A E C. So, the respective Threaded Binary tree will be --
Q.4. Explain Huffman Algorithm with example.

Huffman coding is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters.

There are mainly two parts. First one to create a Huffman tree, and another one to traverse the tree to find codes.

Huffman Algorithm was developed by David Huffman in 1951.

This is a technique which is used in a data compression or it can be said that it is a coding technique which is used for encoding data.

This technique is a mother of all data compression scheme.

This idea is basically dependent upon the frequency, i.e. the frequency of the corresponding character which needs to be compressed, and by that frequency, only Huffman code will be generated.

In case of Huffman coding, the most generated character will get the small code and least generated character will get the large code.

Huffman tree is a specific method of representing each symbol.

This technique produces a code in such a manner that no codeword is a prefix of some other code word. These codes are called as prefix code.

Example:

Let obtain a set of Huffman code for the message \((m_1.....m_7)\) with relative frequencies \((q_1.....q_7) = (4,5,7,8,10,12,20)\). Let us draw the Huffman tree for the given set of codes.

**Step 1)** Arrange the data in ascending order in a table.

4,5,7,8,10,12,20

**Step 2)** Combine first two entries of a table and by this create a parent node.
Step 3)

A) Remove the entries 4 and 5 from the table and insert 9 at its appropriate position. \(7,8,9,10,12,20\)

Combine minimum value of table and create a parent node.

```
    15
   / \  
  7   8
```

B) Now remove the entries 7 and 8 from the table and insert 15 at its appropriate position. \(9,10,12,15,20\)

Combine minimum value of two blocks and create a parent node.

```
    19
   /  
  9   10
   /   
  4   5
```

C) Remove the entries 9 and 10 from the table and insert 19 at its proper position. \(12,15,19,20\).

Combine minimum value of two blocks and create parent node.
D) Remove the entries 15 and 12 from the table and insert 27 at its appropriate position. **19, 20, 27**

Combine minimum value of two blocks and create parent node.

E) Remove the entries 19 and 20 from the table and insert 39 in the table. **27, 39**

Combine minimum value of two blocks and create parent node.
Step 4) Now assign left child as 0 and right child as 1 to encode the frequencies.

Now, codes for the given frequencies are given below:
Q.5 **How a binary tree is different from binary search tree.**

Binary tree is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as **Left child** and **Right Child**. It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM).

**Definition:**

"A tree in which every node can have maximum of two children is called as Binary Tree."
The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively. The above figure shows how a binary tree is represented as an array. Value '7' is the total number of nodes. If any node does not have any of its child, null value is stored at the corresponding index of the array.

**Binary Search Tree**

- Binary search tree is a binary tree which has special property called BST.
- BST property is given as follows:
  
  **For all nodes A and B,**

  I. If B belongs to the left subtree of A, the key at B is less than the key at A.

  II. If B belongs to the right subtree of A, the key at B is greater than the key at A.

**Each node has following attributes:**

I. Parent (P), left, right which are pointers to the parent (P), left child and right child respectively.

II. Key defines a key which is stored at the node.

**Definition:**

"Binary Search Tree is a binary tree where each node contains only smaller values in its left subtree and only larger values in its right subtree."
The above tree represents binary search tree (BST) where left subtree of every node contains smaller values and right subtree of every node contains larger value.

- Binary Search Tree (BST) is used to enhance the performance of binary tree.
- It focuses on the search operation in binary tree.

Every binary search tree is a binary tree, but all the binary trees need not to be binary search trees.

**Q.6 Define Binary trees. How it can be represented in the memory (Linked and Array representation).**

Binary tree is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as Left child and Right Child. It is a method of placing and locating the records in a database, especially when all the data is known to be in random access memory (RAM).

**Definition:**

"A tree in which every node can have maximum of two children is called as Binary Tree."

![Fig. Binary Tree](image)

The above tree represents binary tree in which node A has two children B and C. Each children have one child namely D and E respectively.

**Representation of Binary Tree using Array**

Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.
Array index is a value in tree nodes and array value gives to the parent node of that particular index or node. Value of the root node index is always -1 as there is no parent for root. When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.

Location number of an array is used to store the size of the tree. The first index of an array that is '0', stores the total number of nodes. All nodes are numbered from left to right level by level from top to bottom. In a tree, each node having an index i is put into the array as its i th element.

Q.7. Draw a binary Tree for the expression : $A \times B - (C + D) \times (P / Q)$. 

Fig. Binary Tree using Array

Fig. Location Number of an Array in a Tree
Q.8 Define threaded binary tree.

A threaded binary tree defined as follows:

"A binary tree is *threaded* by making all right child pointers that would normally be null point to the in-order successor of the node (if it exists), and all left child pointers that would normally be null point to the in-order predecessor of the node."[1]

This definition assumes the traversal order is the same as in-order traversal of the tree. However, pointers can instead (or in addition) be added to tree nodes, rather than replacing linked lists thus defined are also commonly called "threads", and can be used to enable traversal in any order(s) desired. For example, a tree whose nodes represent information about people might be sorted by
name, but have extra threads allowing quick traversal in order of birth date, weight, or any other known characteristic.

A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.

**Single Threaded:** Where a NULL right pointers is made to point to the inorder successor (if successor exists)

**Double Threaded:** Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

Let's make the Threaded Binary tree out of a normal binary tree:

![Diagram of a threaded binary tree]

The **in-order** traversal for the above tree is — D B A E C. So, the respective Threaded Binary tree will be --

![Diagram of a threaded binary tree with threads marked]
Q.9 Write Preorder traversal algorithm for binary tree with example.

Preorder Traversal

Algorithm for preorder traversal

Step 1: Start from the Root.

Step 2: Then, go to the Left Subtree.

Step 3: Then, go to the Right Subtree.

The above figure represents how preorder traversal actually works.

Following steps can be defined the flow of preorder traversal:

Step 2: A + B + D (E + F) + C (G + H)

Step 3: A + B + D + E + F + C + G + H

Preorder Traversal: A B C D E F G H

Q.10 Write Post order traversal algorithm for binary tree with example.

Algorithm for postorder traversal

Step 1: Start from the Left Subtree (Last Leaf).

Step 2: Then, go to the Right Subtree.

Step 3: Then, go to the Root.
The above figure represents how postorder traversal actually works.

**Following steps can be defined the flow of postorder traversal:**

**Step 1:** As we know, preorder traversal starts from left subtree (last leaf) \(((\text{Postorder on E} + \text{Postorder on F}) + D + B)) + ((\text{Postorder on G} + \text{Postorder on H}) + C) + (\text{Root A})

**Step 2:** \((E + F) + D + B + (G + H) + C + A\)

**Step 3:** \(E + F + D + B + G + H + C + A\)

**Postorder Traversal:** \(E\ F\ D\ B\ G\ H\ C\ A\)

Q.11 Write Inorder traversal algorithm for binary tree with example.

**Inorder Traversal**

**Algorithm for inorder traversal**

**Step 1:** Start from the Left Subtree.

**Step 2:** Then, visit the Root.

**Step 3:** Then, go to the Right Subtree.
The above figure represents how inorder traversal actually works.

**Following steps can be defined the flow of inorder traversal:**

**Step 1:** B + (Inorder on E) + D + (Inorder on F) + (Root A) + (Inorder on G) + C (Inorder on H)

**Step 2:** B + (E) + D + (F) + A + G + C + H

**Step 3:** B + E + D + F + A + G + C + H

**Inorder Traversal:** B E D F A G C H

Q.12 WAP in C to implement a binary tree and its traversal in preorder.

```c
#include<stdio.h>
#include<stdlib.h>

struct node {
    int data;
    struct node *rlink;
    struct node *llink;
}*tmp=NULL;

typedef struct node NODE;
NODE *create();
void preorder(NODE *);
void inorder(NODE *);
void postorder(NODE *);
void insert(NODE *);
```
int main()
{
    int n,i,ch;
    do
    {
        printf("n1.Create\n2.Insert\n3.Preorder\n4.Postorder\n5.Inorder\n6.Exit\n\n"));
        printf("Enter Your Choice :");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                tmp=create();
                break;
            case 2:
                insert(tmp);
                break;
            case 3:
                printf("Display Tree in Preorder Traversal : ");
                preorder(tmp);
                break;
            case 4:
                printf("Display Tree in Postorder Traversal : ");
                postorder(tmp);
                break;
            case 5:
                printf("Display Tree in Inorder Traversal : ");
                inorder(tmp);
                break;
            case 6:
                exit(0);
                default:
                    printf("Invalid Choice..");
    }
    } while(n!=5);
}
void insert(NODE *root)
{
    NODE *newnode;
    if(root==NULL)
    {
        newnode=create();
        root=newnode;
    }
    else
{  
    newnode=create();  
    while(1)  
    {  
        if(newnode->data<root->data)  
        {  
            if(root->llink==NULL)  
            {  
                root->llink=newnode;  
                break;  
            }  
            root=root->llink;  
        }  
        if(newnode->data>root->data)  
        {  
            if(root->rlink==NULL)  
            {  
                root->rlink=newnode;  
                break;  
            }  
            root=root->rlink;  
        }  
    }  
}  

NODE *create()  
{  
    NODE *newnode;  
    int n;  
    newnode=(NODE *)malloc(sizeof(NODE));  
    printf("\n\nEnter the Data ");  
    scanf("%d",&n);  
    newnode->data=n;  
    newnode->llink=NULL;  
    newnode->rlink=NULL;  
    return(newnode);  
}  

void postorder(NODE *tmp)  
{  
    if(tmp!=NULL)  
    {  
        postorder(tmp->llink);  
        postorder(tmp->rlink);  
        printf("%d->",tmp->data);  
    }  
}
void inorder(NODE *tmp)
{
    if(tmp!=NULL)
    {
        inorder(tmp->llink);
        printf("%d-",tmp->data);
        inorder(tmp->rlink);
    }
}
void preorder(NODE *tmp)
{
    if(tmp!=NULL)
    {
        printf("%d-",tmp->data);
        preorder(tmp->llink);
        preorder(tmp->rlink);
    }
}

Q.13. WAP in C to find sum of all left leaves of a binary tree.

#include <stdio.h>

/* A binary tree Node has key, pointer to left and right 
children */

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

/* Helper function that allocates a new node with the 
given data and NULL left and right pointer. */

Node *newNode(char k)
{
    Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
return node;
}

// A utility function to check if a given node is leaf or not
bool isLeaf(Node *node)
{
    if (node == NULL)
        return false;
    if (node->left == NULL && node->right == NULL)
        return true;
    return false;
}

// This function returns sum of all left leaves in a given
// binary tree
int leftLeavesSum(Node *root)
{
    // Initialize result
    int res = 0;

    // Update result if root is not NULL
    if (root != NULL)
    {
        // If left of root is NULL, then add key of
        // left child
        if (isLeaf(root->left))
            res += root->left->key;
        else // Else recur for left child of root
            res += leftLeavesSum(root->left);
    }
// Recur for right child of root and update res
res += leftLeavesSum(root->right);
}

// return result
return res;

/* Driver program to test above functions*/
int main()
{
    // Let us a construct the Binary Tree
    struct Node *root = newNode(20);
    root->left = newNode(9);
    root->right = newNode(49);
    root->right->left = newNode(23);
    root->right->right = newNode(52);
    root->right->right->left = newNode(50);
    root->left->left = newNode(5);
    root->left->right = newNode(12);
    root->left->right->right = newNode(12);
    sum= leftLeavesSum(root);
    printf("Sum of left leaves is %d", sum);
    return 0;
}

Q.14 WAP in C to Check whether a binary tree is a full binary tree or not.
To check whether a binary tree is a full binary tree we need to test the following cases:

1) If a binary tree node is NULL then it is a full binary tree.
2) If a binary tree node does have empty left and right sub-trees, then it is a full binary tree by definition.
3) If a binary tree node has left and right sub-trees, then it is a part of a full binary tree by definition. In this case recursively check if the left and right sub-trees are also binary trees themselves.
4) In all other combinations of right and left sub-trees, the binary tree is not a full binary tree.

Following is the implementation for checking if a binary tree is a full binary tree.

```c
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Tree node structure */
struct Node
{
    int key;
    struct Node *left, *right;
};

/* Helper function that allocates a new node with the given key and NULL left and right pointer. */
struct Node *newNode(char k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* This function tests if a binary tree is a full binary tree. */
bool isFullTree (struct Node* root)
{

```
// If empty tree
if (root == NULL)
    return true;
// If leaf node
if (root->left == NULL && root->right == NULL)
    return true;
// If both left and right are not NULL, and left & right subtrees
// are full
if ((root->left) && (root->right))
    return (isFullTree(root->left) && isFullTree(root->right));
// We reach here when none of the above if conditions work
return false;
}

// Driver Program
int main()
{
    struct Node* root = NULL;
    root = newNode(10);
    root->left = newNode(20);
    root->right = newNode(30);
    root->left->right = newNode(40);
    root->left->left = newNode(50);
    root->right->left = newNode(60);
    root->right->right = newNode(70);
    root->left->left->left = newNode(80);
    root->left->left->right = newNode(90);
root->left->right->left = newNode(80);
root->left->right->right = newNode(90);
root->right->left->left = newNode(80);
root->right->left->right = newNode(90);
root->right->right->left = newNode(80);
root->right->right->right = newNode(90);
if (isFullTree(root))
    printf("The Binary Tree is full\n");
else
    printf("The Binary Tree is not full\n");

return(0);

Q.15 WAP in C to find out Minimum Depth of a Binary Tree.

maxDepth()
1. If tree is empty then return 0
2. Else
   (a) Get the max depth of left subtree recursively i.e.,
       call maxDepth( tree->left-subtree)
   (a) Get the max depth of right subtree recursively i.e.,
       call maxDepth( tree->right-subtree)
   (c) Get the max of max depths of left and right
       subtrees and add 1 to it for the current node.
       max_depth = max(max dept of left subtree,
                       max depth of right subtree)
                 + 1
   (d) Return max_depth
See the below diagram for more clarity about execution of the recursive function maxDepth() for above example tree.

```
maxDepth('1') = max(maxDepth('2'), maxDepth('3')) + 1
    = 2 + 1
      /    \
    /      \
  /        \
 /          \
/            
maxDepth('2') = 1               maxDepth('3') = 1
= max(maxDepth('4'), maxDepth('5')) + 1
= 1 + 1   = 2
      /    
    /      
  /        
/          
/            
maxDepth('4') = 1               maxDepth('5') = 1
```

```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
    and a pointer to right child */

struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
   nodes along the longest path from the root node
```
```c
int maxDepth(struct node* node)
{
    if (node == NULL)
        return 0;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return(lDepth+1);
        else return(rDepth+1);
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
```
Q.16 WAP in C to check if two binary trees are identical.

ANS- #include <stdio.h>

// BST node
struct Node {  
    int data;  
    struct Node* left;  
    struct Node* right; 
};

// Utility function to create a new Node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)
        malloc(sizeof(struct Node));
node->data = data;
node->left = NULL;
node->right = NULL;

return node;
}

// Function to perform inorder traversal
void inorder(Node* root)
{
    if (root == NULL)
        return;

    inorder(root->left);

    printf("%d",root->data);

    inorder(root->right);
}

// Function to check if two BSTs are identical
// are identical
int isIdentical(Node* root1, Node* root2)
{
    // Check if both the trees are empty
    if (root1 == NULL && root2 == NULL)
        return 1;
    // If any one of the tree is non-empty
    // and other is empty, return false
    else if (root1 != NULL && root2 == NULL)
        return 0;
    else if (root1 == NULL && root2 != NULL)
        return 0;
    else {
        // Check if current data of both trees equal
        // and recursively check for left and right subtrees
        if (root1->data == root2->data && isIdentical(root1->left, root2->left)
            && isIdentical(root1->right, root2->right))
            return 1;
        else
            return 0;
    }
}

// Driver code
int main()
{
    struct Node* root1 = newNode(5);
    struct Node* root2 = newNode(5);
    root1->left = newNode(3);
    root1->right = newNode(8);
    root1->left->left = newNode(2);
    root1->left->right = newNode(4);
    root2->left = newNode(3);

    root2->left = newNode(3);
}
Q17. What is B+ tree?

B+ Tree

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.

![B+ Tree Diagram](image)

Advantages of B+ Tree
1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.

The structure of the leaf nodes of a B+ tree of order ‘b’ is as follows:
1. Each leaf node is of the form:
   \(<<K_1, D_1>, <K_2, D_2>, \ldots, <K_{c-1}, D_{c-1}>, P_{next}>\)
   where \(c \leq b\) and each \(D_i\) is a data pointer (i.e., points to actual record in the disk whose key value is \(K_i\) or to a disk file block containing that record) and, each \(K_i\) is a key value and, \(P_{next}\) points to next leaf node in the B+ tree (see diagram II for reference).
2. Every leaf node has: \(K_1 < K_2 < \ldots < K_{c-1}, c \leq b\)
3. Each leaf node has at least \(\lceil b/2 \rceil\) values.
4. All leaf nodes are at the same level.

Q.18 What is strictly binary tree?

- If every nonleaf node in a binary tree has nonempty left and right subtrees, the tree is called a strictly binary tree.
- A strictly binary tree with \(n\) leaves always contains \(2n - 1\) nodes.
- If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a strictly binary tree. Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one.

Q.19. Obtain an AVL tree starting from an empty tree on the following sequence of insertions:3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 9.
Draw AVL Tree for given data: 3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 9.

**Step 1:**
- Apply RR

**Step 2:**
- Balanced till the insertion of 2

**Step 3:**
- Taking RST from 11
- critical node
- 8 + 2
- 12
- Apply LL
- 5
Q.20 Describe B tree. Describe the method to delete one item from B tree.

B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

1. Locate the leaf node.
2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain \( \frac{m}{2} \) keys then complete the keys by taking the element from eight or left sibling.
   - If the left sibling contains more than \( \frac{m}{2} \) elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
   - If the right sibling contains more than \( \frac{m}{2} \) elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.

4. If neither of the sibling contain more than \( \frac{m}{2} \) elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.

5. If parent is left with less than \( \frac{m}{2} \) nodes then, apply the above process on the parent too.

If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

**Example 1**

Delete the node 53 from the B Tree of order 5 shown in the following figure.

![B Tree Diagram](image)

53 is present in the right child of element 49. Delete it.
Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. It is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.