Unit 3

Introduction to Tree Data Structure

Tree is a hierarchical data structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the **root**, and the nodes below it is called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

Why Tree is considered a non-linear data structure?

Data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. Hence, the tree is considered to be a non-linear data structure. An example tree is shown in Fig 1 and Fig 2.

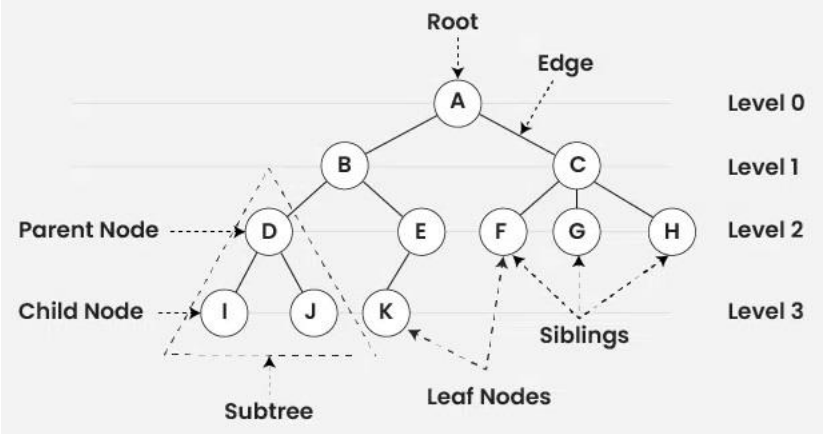


Fig 1: Tree with levels

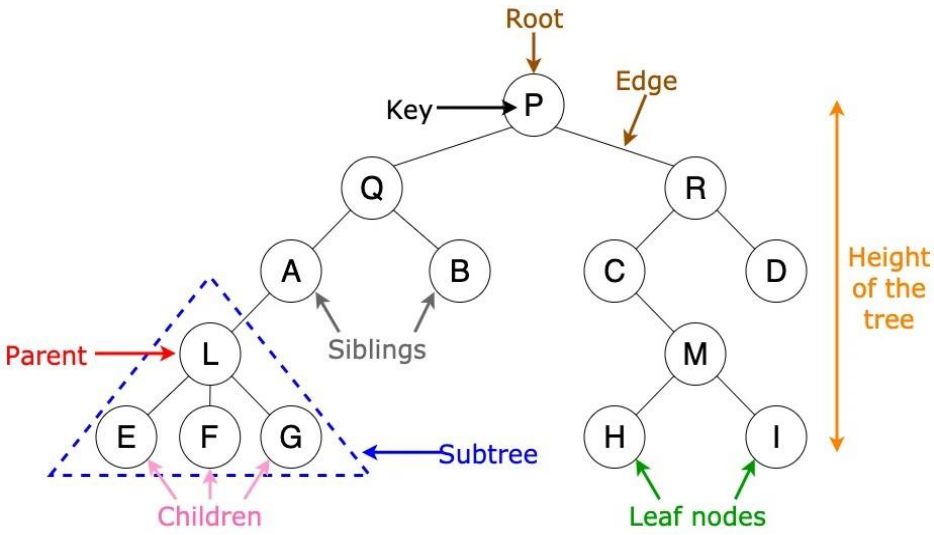


Fig 2: Labelling of entities in a tree

Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbours of that node.
- **Subtree**: Any node of the tree along with its descendant.

Representation of Tree Data Structure:

A tree consists of a root node, and zero or more subtrees T_1, T_2, … , T_k such that there is an edge from the root node of the tree to the root node of each subtree. Subtree of a node X consists of all the nodes which have node X as the ancestor node. A tree can be represented using a collection of nodes. Fig 3 illustrates this. Each of the nodes can be represented with the help of class or *structs*.

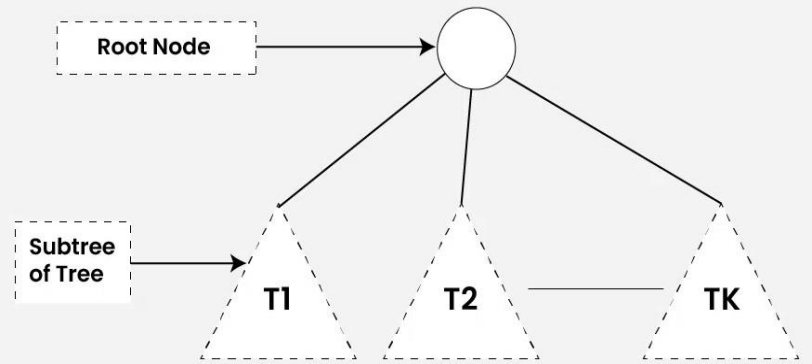


Fig 3: Subtree of Tree

```
struct Node {
  int data;
  struct Node * first_child;
  struct Node * second_child;
  struct Node * third_child;
  .
  .
  .
  struct Node * nth_child;
};
```

Importance of Tree Data Structure:

One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer: The DOM model of an HTML page is also tree where we have html tag as root, head and body its children and these tags, then have their own children.

Types of Tree data structures:

Tree data structure can be classified into three types based upon the number of children each node of the tree can have. The types are:

- **Binary tree:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees. Examples of Binary Tree are Binary Search Tree and Binary Heap.
- **Ternary Tree:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

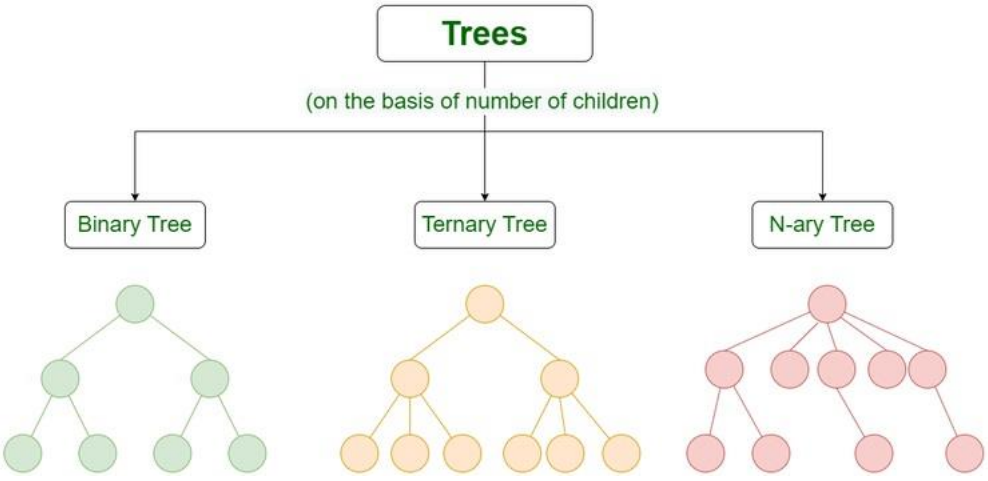


Fig 4: Types of Trees

Basic Operations Of Tree Data Structure:

- **Create** – create a tree in the data structure.
- **Insert** − Inserts data in a tree.
- **Delete**- deletes a specific node
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Traversal**: Movement across nodes in specific order

Properties of Tree Data Structure:

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree. Refer Fig 5.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

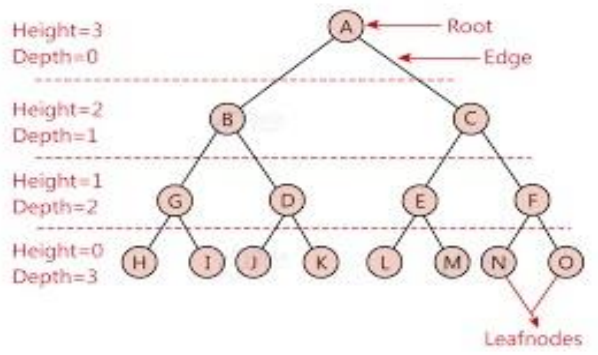


Fig 5: Height and Depth illustration of Tree

Types of Binary Tree

Types of Binary Tree based on the number of children:

1. Full Binary Tree
2. Degenerate Binary Tree
3. Skewed Binary Trees

Full Binary Tree

A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children. It is also known as a proper binary tree.

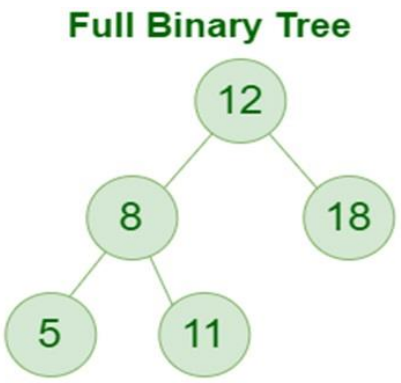


Fig 6

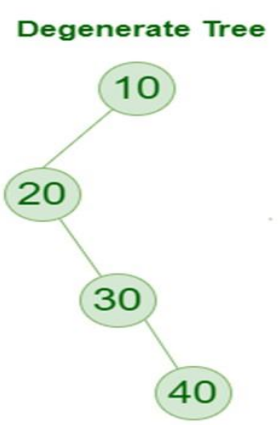


Fig 7

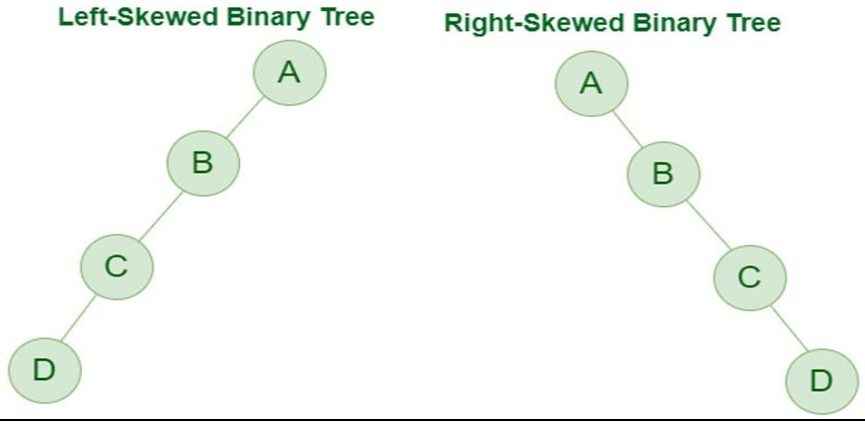


Fig 8

Degenerate (or pathological) tree

A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.

Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.

Types of Binary Tree On the basis of the completion of levels:

1. Complete Binary Tree
2. Perfect Binary Tree
3. Balanced Binary Tree

Complete Binary Tree

A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

A complete binary tree is just like a full binary tree, but with two major differences:

- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

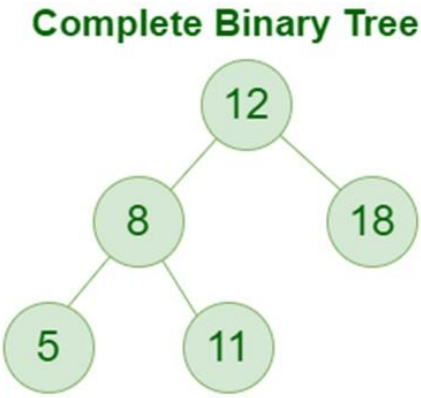


Fig 9: Complete Binary Tree

Perfect Binary Tree

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level. Examples of Perfect Binary Trees is shown in Fig 10.

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

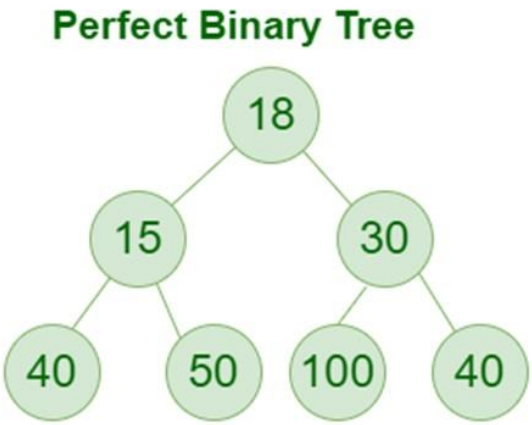


Fig 10:Perfect Binary Tree

In a Perfect Binary Tree, the number of leaf nodes is the number of internal nodes plus 1

L = I + 1 where L = Number of leaf nodes, I = Number of internal nodes.

A Perfect Binary Tree of height h (where the height of the binary tree is the number of edges in the longest path from the root node to any leaf node in the tree, height of root node is 0) has $2^{h+1} - 1$ node.
An example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree
A binary tree is balanced if the height of the tree is O(log n) where n is the number of nodes. For Example, the AVL tree maintains O(log n) height by making sure that the difference between the heights of the left and right subtrees is at most 1. Balanced Binary trees are performance-wise good as they provide O(log n) time for search, insert and delete.

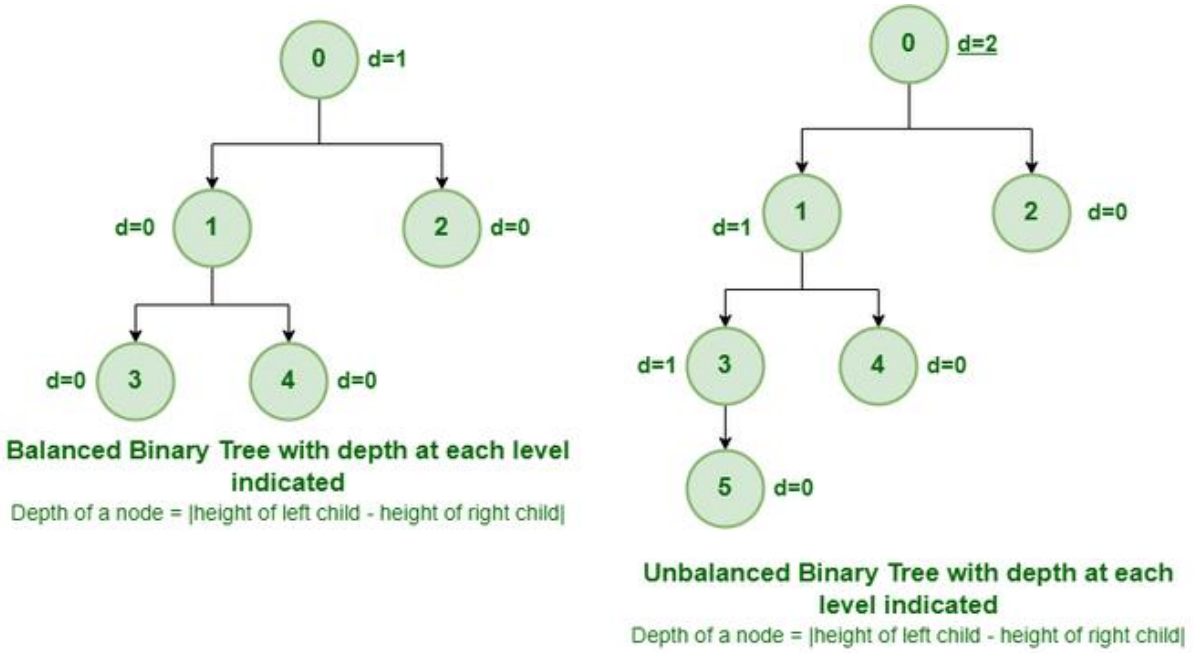


Fig 11: Example of Balanced and Unbalanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1. In the figure above, the root node having a value 0 is unbalanced with a depth of 2 units.

Some Special Types of Trees on the basis of node values:
Binary Tree can be classified into the following special types:

1. Binary Search Tree
2. AVL Tree
3. Red Black Tree
4. B Tree
5. B+ Tree
6. Segment Tree

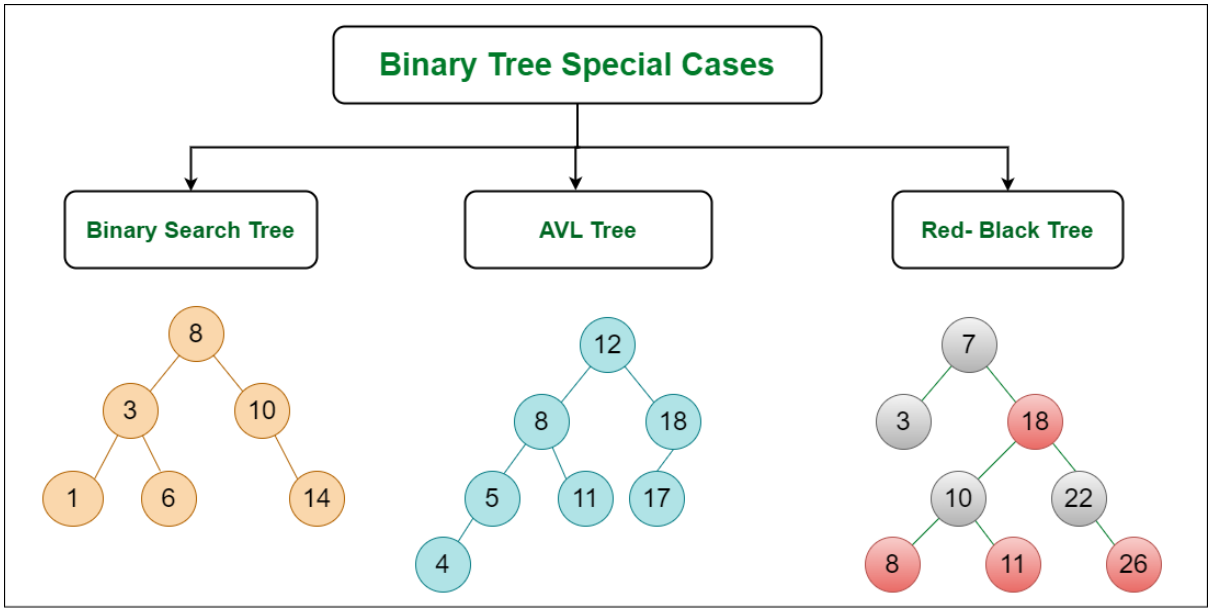


Fig 12 : Important Special cases of binary Trees

Binary Search Tree

Binary Search Tree is a node-based binary tree data structure that has the following properties:

- Left subtree of a node contains only nodes with keys lesser than the node's key.
- Right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Tree traversal

The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

- Preorder traversal
- Inorder traversal
- Postorder traversal

Note: Traversal can be applied to any binary tree in general. However when applied to BST, it gives certain pattern.

Preorder traversal

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

So, in a preorder traversal, each node is visited before both of its subtrees.

The applications of preorder traversal include -

- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

Inorder traversal

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

So, in the inorder traversal, each node is visited in between of its subtrees.

The applications of Inorder traversal includes -

- It is used to get the BST nodes in increasing order.
- It can also be used to get the prefix expression of an expression tree.

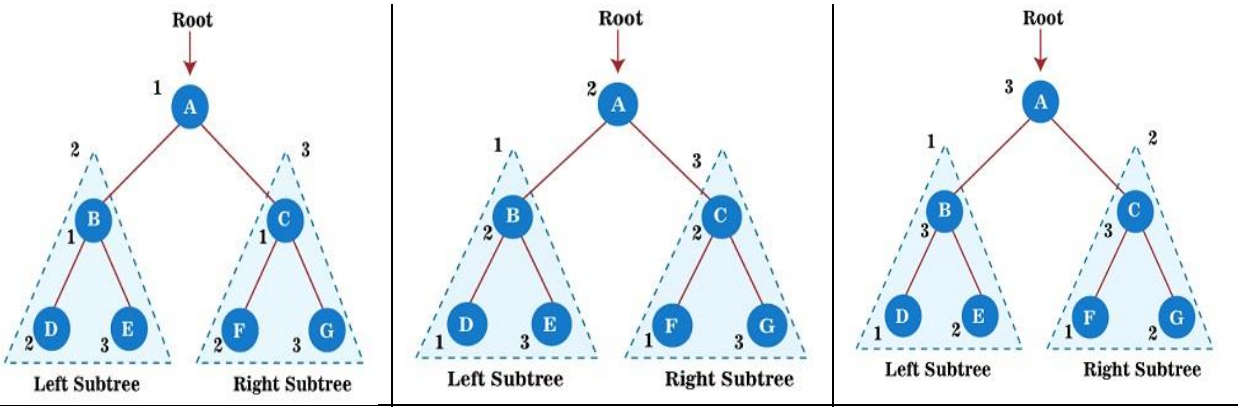
Postorder traversal

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

So, in a postorder traversal, each node is visited after both of its subtrees.

The applications of postorder traversal include -

- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.



Preorder	Inorder	Postorder
A → B → D → E → C → F → G	D → B → E → A → F → C → G	D → E → B → F → G → C → A

Fig 13: Tree traversal

Traversal Algorithms are summarized in Table 1.

Preorder Algorithm
Until all nodes of the tree are not visited 1. Step 1 - Visit the root node 2. Step 2 - Traverse the left subtree recursively. 3. Step 3 - Traverse the right subtree recursively.

Inorder Algorithm

Until all nodes of the tree are not visited

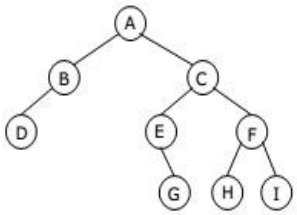
1. Step 1 - Traverse the left subtree recursively.
2. Step 2 - Visit the root node.
3. Step 3 - Traverse the right subtree recursively.

Postorder Algorithm

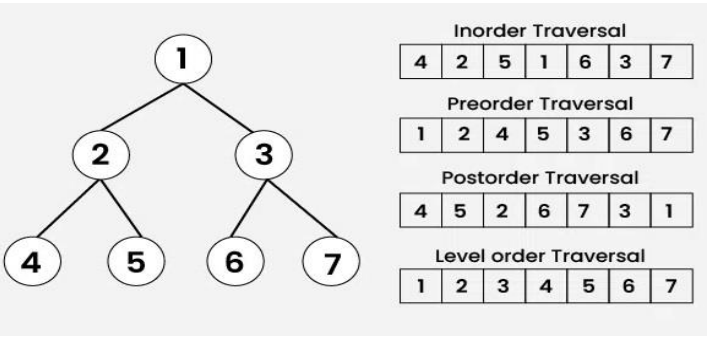
Until all nodes of the tree are not visited

1. Step 1 - Traverse the left subtree recursively.
2. Step 2 - Traverse the right subtree recursively.
3. Step 3 - Visit the root node.

Some solved examples are shown below.



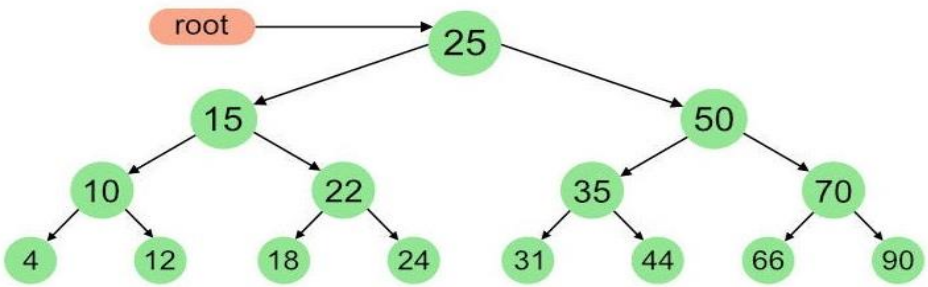
- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I



InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Implementation of Tree traversal

```
// C program to implement binary search tree
#include <stdio.h>
#include <stdlib.h>

// Define a structure for a binary tree node
struct BinTree {
   int key;
   struct BinTree *left, *right;
};

// Function to create a new node with a given value
struct BinTree* newNodeCreate(int value)
{
   struct BinTree* temp = (struct BinTree*) malloc(sizeof(struct BinTree));
   temp->key = value;
   temp->left = temp->right = NULL;
   return temp;
}

// Function to insert a node with a specific value in the tree
struct BinTree*
insertNode(struct BinTree* node, int value)
{
   if (node == NULL) {
      return newNodeCreate(value);
   }
   if (value < node->key) {
      node->left = insertNode(node->left, value);
   }
   else if (value > node->key) {
      node->right = insertNode(node->right, value);
   }
   return node;
}

// Function to perform post-order traversal
void postOrder(struct BinTree* root)
{
   if (root != NULL) {
      postOrder(root->left);
      postOrder(root->right);
      printf(" %d ", root->key);
   }
}

// Function to perform in-order traversal
void inOrder(struct BinTree* root)
{
   if (root != NULL) {
      inOrder(root->left);
```

```
        printf(" %d ", root->key);
        inOrder(root->right);
    }
}

// Function to perform pre-order traversal
void preOrder(struct BinTree* root)
{
    if (root != NULL) {
        printf(" %d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Function to find the minimum value
struct BinTree* findMin(struct BinTree* root)
{
    if (root == NULL) {
        return NULL;
    }
    else if (root->left != NULL) {
        return findMin(root->left);
    }
    return root;
}

// Function to search for a node with a specific key in the tree
struct BinTree *searchNode(struct BinTree* root, int target)
{
    if (root == NULL || root->key == target)
        return root;
    if (root->key < target)
        return searchNode(root->right, target);
    return searchNode(root->left, target);
}

// Function to delete a node from the tree
struct BinTree* delete (struct BinTree* root, int x)
{
    if (root == NULL)       return NULL;
    if (x > root->key)
        root->right = delete (root->right, x);

    else if (x < root->key)
        root->left = delete (root->left, x);

    else {
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
```

```
        }
        else if (root->left == NULL || root->right == NULL)
        {
            struct BinTree* temp;
            if (root->left == NULL)
                temp = root->right;
            else
                temp = root->left;
                free(root);
            return temp;
        }
        else {
            struct BinTree* temp= findMin(root->right);
            root->key = temp->key;
            root->right = delete (root->right, temp->key);
        }
    }
    return root;
}

int main()
{
    // Initialize the root node
    struct BinTree* root = NULL;

    // Insert nodes into the binary search tree
    root = insertNode(root, 45);
    insertNode(root, 30);    insertNode(root, 20);    insertNode(root, 40);
    insertNode(root, 70);    insertNode(root, 60);    insertNode(root, 80);

    // Search for a node with key 80
    if (searchNode(root, 80) != NULL) {
        printf("element 80 found");
    }
    else {
        printf("element 80 not found");
    }
    printf("\n\n postorder is :");
    postOrder(root);
    printf("\n preorder is :");
   preOrder(root);
    printf("\n   inorder is :");
    inOrder(root);

// Perform delete the node (50)
    struct BinTree* temp = delete (root, 50);
    printf("\n\nAfter Delete: \n");
    inOrder(root);
    return 0;
}
```

Construct A Binary Tree from Inorder and Preorder Traversal

Problem Statement: Given the Preorder and Inorder traversal of a Binary Tree, construct the Unique Binary Tree represented by them.

Input: Inorder: [9, 3, 15, 20, 7], Preorder: [3, 9, 20, 15, 7]
Output : Shown in Fig 14.

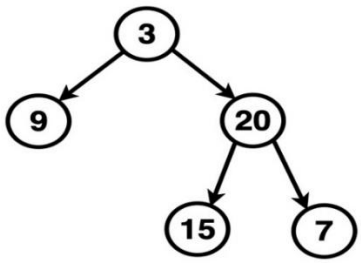


Fig 14

Algorithm / Intuition

Before we dive into the algorithm, it's essential to grasp the significance of inorder and preorder traversals. Inorder traversal allows us to identify a node and its left and right subtrees, while preorder traversal ensures we always encounter the root node first. Leveraging these properties, we can uniquely construct a binary tree. The core of our approach lies in a recursive algorithm that creates one node at a time. We locate this root node in the inorder traversal, which splits the array into the left and right subtrees.

The inorder array keeps getting divided into left and subtrees hence to avoid unnecessary array duplication, we use variables (inStart, inEnd) and (preStart, preEnd) on the inorder and preorder array respectively. These variables effectively define the boundaries of the current subtree within the original inorder and preorder traversals. Everytime we encounter the root of a subtree via preorder traversal, we locate its position in the inorder array to get the left and right subtrees. So to save complexity on the linear look up, we employ a hashmap to store the index of each element in the inorder traversal. This transforms the search operation into a constant-time lookup.

Process is elaborated in Fig 15.

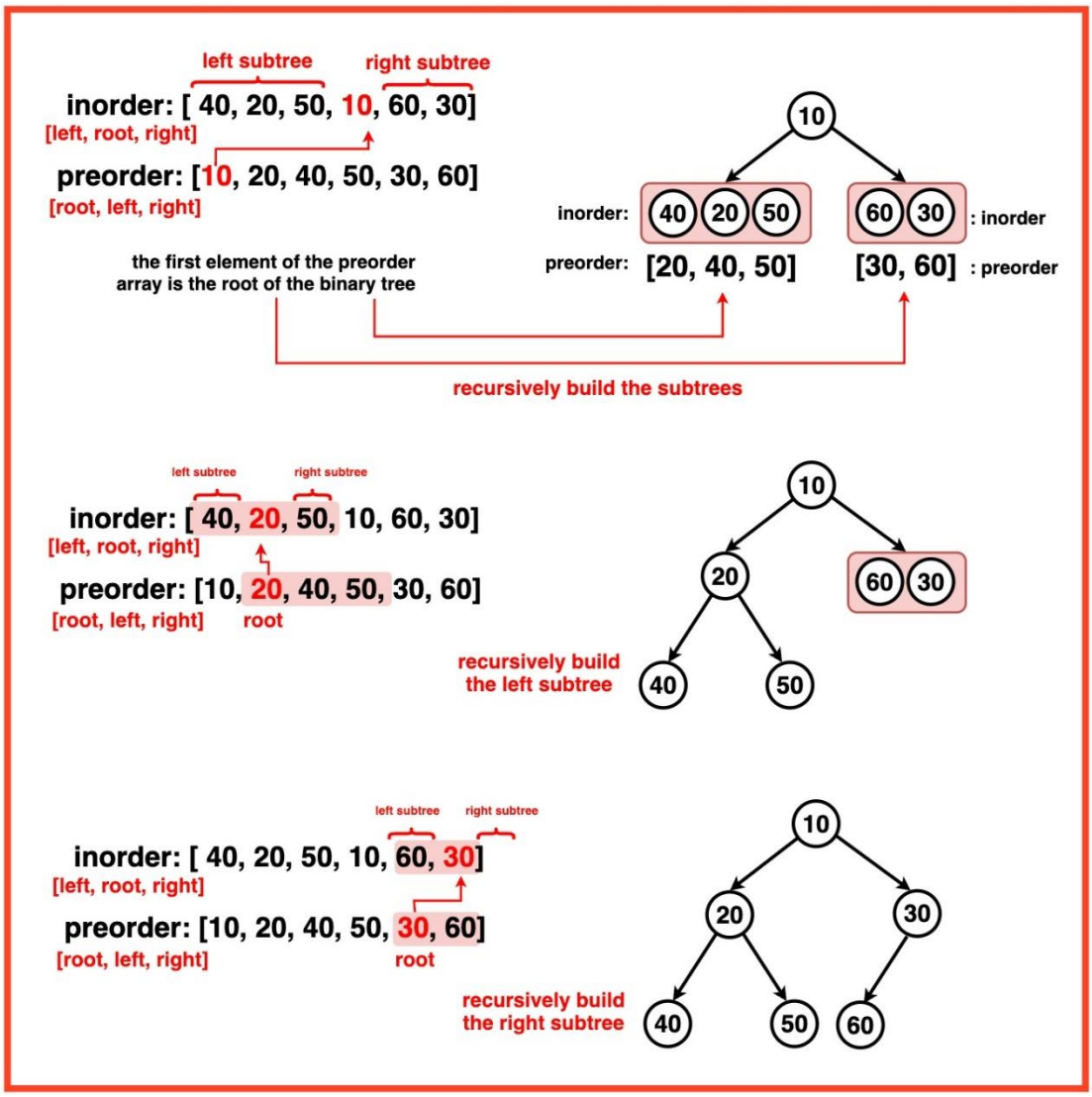


Fig 15: Construction of TREE when 2 orders given

Example 2:
Inorder = {2, 5, 6, 10, 12, 14, 15}
Preorder = {10, 5, 2, 6, 14, 12, 15}

Steps of construction:

1. First element in Preorder will be the root of the tree, here its 10.
2. Now the search element 10 in inorder[], say you find it at position i, once you find it, make note of elements which are left to i (this will construct the leftsubtree) and elements which are right to i (this will construct the rightsubtree).
3. See this step above and recursively construct left subtree and link it root.left and recursively construct right subtree and link it root.right. Pictorial Representation of above questions

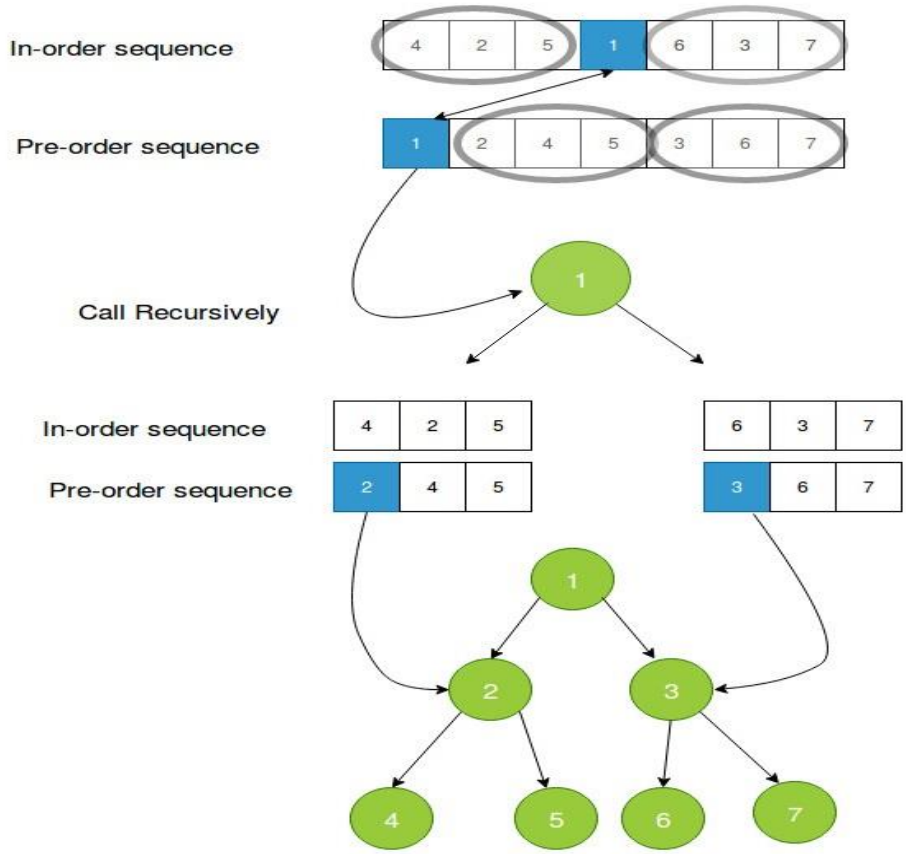


Fig 16: When preorder and inorder given

Algorithm for Tree construction from Inorder and Postorder

Consider the following traversals of the tree.
Inorder = { 4, 2, 5, 1, 6, 3, 7 }
Postorder = { 4, 5, 2, 6, 7, 3, 1 }

Steps of construction:

1. Last element in the postorder[] will be the root of the tree, here it is 1.
2. Now the search element 1 in inorder[], say you find it at position i, once you find it, make note of elements which are left to i (this will construct the leftsubtree) and elements which are right to i (this will construct the rightSubtree).
3. Suppose in previous step, there are X number of elements which are left of 'i' (which will construct the leftsubtree), take first X elements from the postorder[] traversal, this will be the post order traversal for elements which are left to i. similarly if there are Y number of elements which are right of 'i' (which will construct the rightsubtree), take next Y elements, after X elements from the postorder[] traversal, this will be the post order traversal for elements which are right to i.
4. From previous two steps construct the left and right subtree and link it to root.left and root.right respectively.

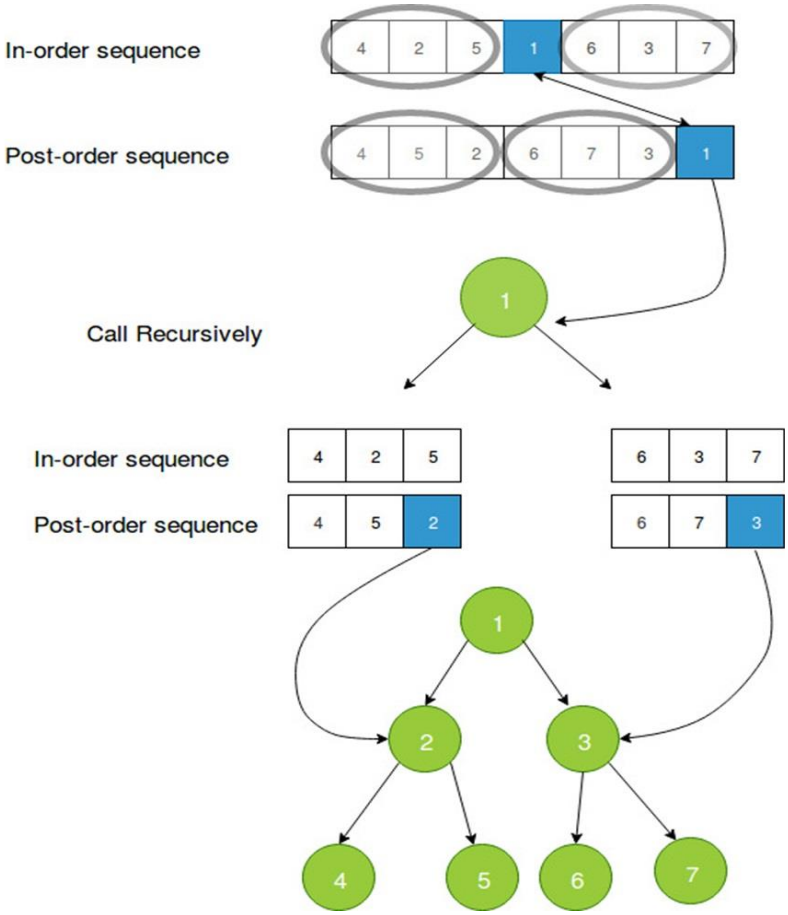


Fig 17: When post-order and inorder given

Expression Tree

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand. For example, expression tree for 3 + ((5+9)*2) is shown in Fig 18.

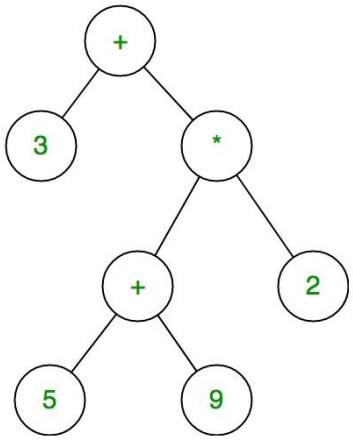


Fig 18: Expression tree

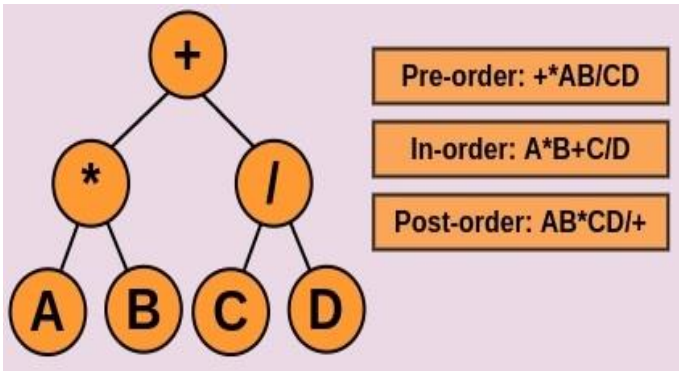


Fig 19: Traversal of expression tree

Inorder traversal of expression tree produces infix version of given postfix expression (same with postorder traversal it gives postfix expression)

Construction of Expression Tree:

Now for constructing an expression tree we use a stack. We loop through input expression (it should be in postfix form) and do the following for every character.

1. If a character is an operand push that into the stack
2. If a character is an operator pop two values from the stack make them its child and push the current node again.

In the end, the only element of the stack will be the root of an expression tree.

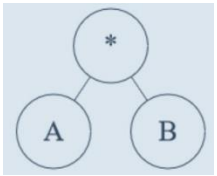
Example: A*B+C-D/E

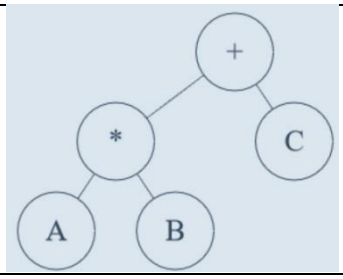
Postfix is AB*C+DE/-

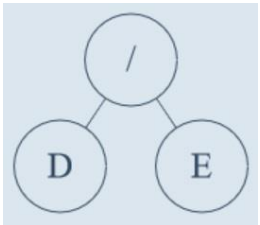
Stack status during construction is shown below and Final Tree is shown in Fig 20.

1	2	3	4	5	6	7	8	9
						E		
	B		C		D	D	D/E	
A	A	A*B	A*B	A*B+C	A*B+C	A*B+C	A*B+C	A*B+C-D/E

1. Scan A, push to stack
2. Scan B, push to stack
3. Scan *, pop two top elements, build tree A*B and push this back to stack
4. Scan C, push to stack
5. Scan +, pop two top elements, build tree A*B+C and push this back to stack
6. Scan D, push to stack
7. Scan E, push to stack
8. Scan /, pop two top elements, build tree D/E and push this back to stack
9. Scan -, pop top 2 and build final tree. Push it back







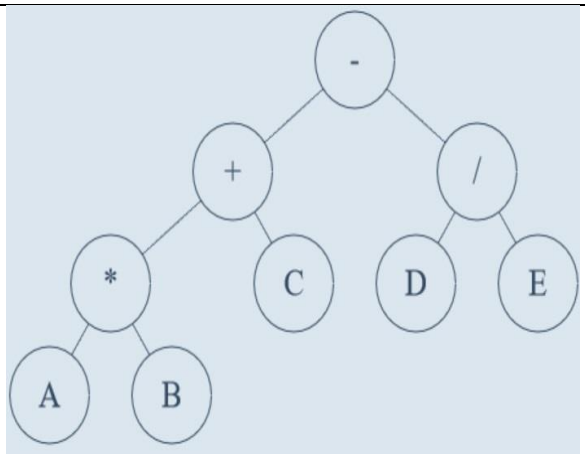
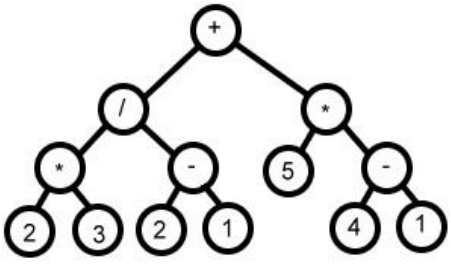
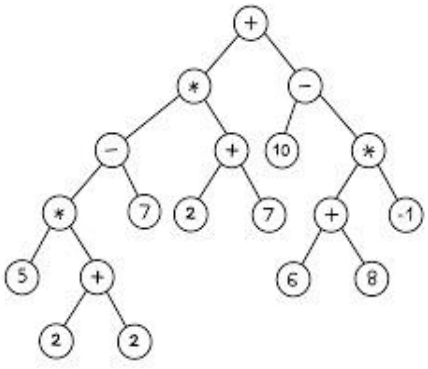


Fig 20:Expression Tree of A*B+C-D/E

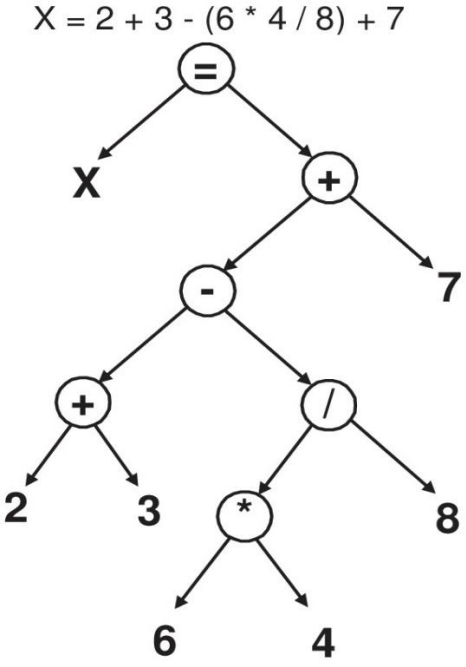
Some examples for expression trees are shown below.



Expression tree for 2*3/(2-1)+5*(4-1)



Exp: 5*(2+2)-7*(2+7)+10-(6+8*-1)



References

- https://builtin.com/software-engineering-perspectives/tree-traversal
- https://ds1-iiith.vlabs.ac.in/exp/tree-traversal/reconstructing-binary-tree/from-inorder-and-postorder.html
- https://expression-generator.netlify.app/