Trees

Introduction -

A tree is recursively defined as a set of one or more nodes where one node is designated as the root of the tree and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.

Basic Terminology -

Root node The root node R is the topmost node in the tree. If R = NULL, then it means the tree is empty.
Sub-trees If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.
Leaf node A node that has no children is called the leaf node or the terminal node.
Path A sequence of consecutive edges is called a path. For example, in Fig. 9.1, the path from the root node A to node I is given as: A, D, and I.
Ancestor node An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors.
Descendant node A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants.
Level number Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1. Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.
Degree of a node is equal to the number of children that a node has. The degree of a leaf node is zero.
In-degree of a node is the number of edges arriving at that node.
Out-degree of a node is the number of edges leaving that node.

Applications of Trees -

- One reason to use trees might be because we want to store information that naturally forms a hierarchy. For example, the file system on a computer: file system
- If we organize keys in the form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Log n) for search.
- We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists).

- Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Log n) for insertion/deletion.
- Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Types of Trees -

- **Binary Tree -** A binary tree is a tree such that every node has at most 2 children and each node is labeled as being either a left child or a right child.
- **Binary Search Tree -** a node-based binary tree data structure which has the following properties:
    - The left subtree of a node contains only nodes with keys lesser than the node's key.
    - The 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.
- **AVL Tree**
- **Expression Tree**
- **Huffman Tree -** A full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet.

Binary Tree -

Basic Operations -

- Create
- isleftchild()
- isrightchild()
- Tree Traversal
    - inorder()
    - preorder()
    - postorder()

Declaration -

A node in a binary tree will have following three elements:

- A Node value
- A left child pointer
- A right child pointer

Node Structure

```
struct tree { int data; struct tree *left, struct tree *right}
```

Traversal -

1. PreOrder -

Root node first, the left sub-tree next, and then the right sub-tree. Pre-order traversal is also called depth-first traversal. In this algorithm, the left sub-tree is always traversed before the right sub-tree. The word ‘pre’ in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees.
Pre-order traversal algorithms are used to extract a prefix notation from an expression tree.

ALGORITHM -

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: Write TREE -> DATA
Step 3: PREORDER(TREE -> LEFT)
Step 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END

2. InOrder -

Left sub-tree first, the root node next, and then the right sub-tree. In-order traversal is also called symmetric traversal. In this algorithm, the left sub-tree is always traversed before the root node and the right sub-tree. The word ‘in’ in the in-order specifies that the root node is accessed in between the left and the right sub-trees.
In-order traversal algorithm is usually used to display the elements of a binary search tree.

ALGORITHM -

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: INORDER(TREE -> LEFT)
Step 3: Write TREE -> DATA
Step 4: INORDER(TREE -> RIGHT)
[END OF LOOP]
Step 5: END

3. PostOrder -

Left sub-tree first, the right sub-tree next, and finally the root node. In this algorithm, the left sub-tree is always traversed before the right sub-tree and the root node. The word ‘post’ in the post-order specifies that the root node is accessed after the left and the right sub-trees.
Post-order traversals are used to extract postfix notation from an expression tree.

ALGORITHM -

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: POSTORDER(TREE -> LEFT)
Step 3: POSTORDER(TREE -> RIGHT)
Step 4: Write TREE -> DATA
[END OF LOOP]
Step 5: END

Expression Tree -

Expression tree is a binary tree in which each internal node corresponds to an operator and each leaf node corresponds to operand.

Construction of Expression Tree:

For constructing expression trees we use a stack. We loop through input expressions and do the following for every character.

- If a character is an operand push that into stack by creating a node.
- If a character is an operator pop two values (node) from stack make them its child and push current node again.

At the end only the element of stack will be the root of the expression tree.

AVL Tree -

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtree cannot be more than one for all nodes.
Balance Factor = Height of Left Subtree - Height of Right Subtree

Insertion -

- Check is tree balanced after every insertion.
- Insert in Balanced AVL tree – Nothing to be done (Heavy Tree)
- Insert in a non heavy part of heavy tree – Nothing to be done (Balance)
- Insert in a heavy part of the heavy tree – Need to balance the tree. (Unbalance)
- To balance the UNBALANCE AVL TREE we have to use the rotation.
- There are 4 different rotations.
    - LL Rotation - When a new node is inserted in the Left subtree of the Left child of critical node.
    - RR Rotation - When a new node is inserted in the Right subtree of Right child of critical node.
    - LR Rotation - When a new node is inserted in the Right subtree of Left child of the critical node.
    - RL Rotation - When a new node is inserted in the Left subtree of Right child of critical node.

- The rotation is selected based on the insertion and critical node.
- Critical node is the nearest ancestor node on the path from the root to the inserted node whose balance factor is neither -1, 0 nor 1.

Deletion -

- Check is tree balanced after every deletion.
- Delete from Balanced AVL tree – Nothing to be done (Heavy Tree)
- Delete from a heavy part of heavy tree – Nothing to be done (Balance)
- Delete from a non heavy part of the heavy tree – Need to balance the tree. (Unbalance)
- If a node is deleted from the Left Subtree of Critical node then we use L Rotation.
- If a node is deleted from the Right Subtree of Critical node then we use R Rotation.
- There are three different variations in L & R rotation such as L-1, L0 and L1 and R-1,R0 and R1 respectively.
- If the left child of the critical node has balance factor 0 then R0 rotation is applied.
- If the left child of the critical node has balance factor -1 then R-1 rotation is applied.
- If the left child of the critical node has balance factor 1 then R1 rotation is applied.
- If the right child of the critical node has balance factor 0 then L0 rotation is applied.
- If the right child of the critical node has balance factor -1 then L-1 rotation is applied.
- If the right child of the critical node has balance factor 1 then L1 rotation is applied.

Huffman Tree -

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:

- Create a Huffman tree
- Traverse the tree to find codes.

Create -

- Given a tree with a ‘n’ characters and their frequencies, the Huffman algorithm is used to find a tree with a minimum-weighted path length.

- The process essentially begins with the leaf nodes containing the frequency of the nodes.
- Then a new node whose children are the two nodes with the smallest frequency is created, such that the new node's frequency is equal to the sum of the children's frequency.
- That is two nodes are merged together in one node. This process is repeated until the tree has only one node.
- The Huffman tree algorithm can be implemented using a priority queue in which all the nodes are placed in such a way that the node with the lowest weight is given the highest priority.

ALGORITHM -

Step 1: Create a leaf node for each character. Add the character and its frequency of occurrence to the priority queue.
Step 2: Repeat step 3 to 5 while the total number of nodes in the queue is greater than 1.
Step 3: Remove two nodes that have the highest priority.
Step 4: Create a new internal node by merging these two nodes as children and with frequency equal to the sum of the two nodes's frequency.
Step 5: Add a newly created node to the queue.

Data Coding -

In data coding, every left branch is coded with "0" and every right branch is coded with "1". To find a code for a character traverse it from the root node and concatenate the sequence of 0's and 1's.

B Tree -

A B tree is special type of M-way search tree that is widely used for disk access. B Tree has all the properties of M-way search tree and few below mentioned:

- Every node in the B tree has at most m children.
- Every node in the B tree except the root has at least m/2 children where m is the degree of the B tree. (Reason: Bushy Tree)
- The root node has at least two children if it is not a terminal node.
- All the leaf nodes are at the same level.

Operations -

While performing insert or delete operations in B-Tree the number of child nodes may change. So, in order to maintain a minimum number of children, the internal node may be joined or split.

- Search

- Insert
- Delete

Insert -

- Search the B tree to find the leaf node where the new key value should be inserted.
- If the leaf node is not full, that is, it contains less than M-1 key values, then insert the new element in the node keeping the node's elements ordered.
- If the leaf node is full, that is the leaf nodes already has m-1 key elements, then
    - Insert the new value in order into the existing set of keys.
    - Split the node at its median into two nodes
    - Push the median element up to its parent's node. If the parent's node is already full then split the parent node by following the same step.

Deletion

- Leaf Node : Directly delete the value from leaf node using given algorithm

1. Locate the leaf node which is to be deleted.
2. If the leaf node contains more than the minimum required number of keys then delete the value.
3. Else, If the leaf node does not contain the even m/2 elements, then fill the node by taking an element either from the left or right sibling.
    a. If the left sibling has more than the minimum number of keys then push its largest key into the parent's node and pull down the interleaving element from parent node to the leaf node where the key is deleted.
    b. Else, If the right sibling has more than the minimum number of keys then push its smallest key into the parent's node and pull down the interleaving element from parent node to the leaf node where the key is deleted.
4. If both the left and right sibling contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the interleaving elements of the parent node. If pulling the interleaving from the parent nodes leave it with minimum number of keys in the node then the propagate the process upwards, thereby reducing the height of the B-tree

- Internal Node : Replace it with successor value from the leaf node and then delete value from leaf node using above process.

B+ Tree -

A B tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, deletion, retrieval of records, each of which is identified by a key. Unlike B tree, in B+ tree keys are stored in internal nodes while records are stored in the leaf

node only. The leaf nodes of the B+ tree are often linked to one another in a linked list. With B+ tree the internal nodes of the tress are stored in a main memory while leaf nodes are stored in a secondary memory. The internal node is also called as i-nodes or index node.

Points	B Tree	B+ Tree
1	All internal and leaf nodes have data pointers	Only leaf nodes have data pointers
2	Since all keys are not available at leaf, search often takes more time.	All keys are at leaf nodes, hence search is faster and accurate.
3	No duplicate of keys is maintained in the tree.	Duplicates of keys are maintained and all nodes are present at the leaf.
4	Insertion takes more time and it is not predictable sometimes.	Insertion is easier and the results are always the same.
5	Deletion of internal nodes is very complex and the tree has to undergo a lot of transformations.	Deletion of any node is easy because all nodes are found at leaf.
6	Leaf nodes are not stored as a structural linked list.	Leaf nodes are stored as a structural linked list.
7	No redundant search keys are present.	Redundant search keys may be present.