Top of Form

**TREES:**

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

****

**Important Terms**

Following are the important terms with respect to tree.

* Path − Path refers to the sequence of nodes along the edges of a tree.
* Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
* Parent − Any node except the root node has one edge upward to a node called parent.
* Child − The node below a given node connected by its edge downward is called its child node.
* Leaf − The node which does not have any child node is called the leaf node.
* Subtree − Subtree represents the descendants of a node.
* Visiting − Visiting refers to checking the value of a node when control is on the node.
* Traversing − Traversing means passing through nodes in a specific order.
* Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
* keys − Key represents a value of a node based on which a search operation is to be carried out for a node.

**Binary Search Tree Representation**

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

**Tree Node**

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

In a tree, all nodes share common construct.

**BST Basic Operations**

The basic operations that can be performed on a binary search tree data structure, are the following −

* Insert − Inserts an element in a tree/create a tree.
* Search − Searches an element in a tree.
* Preorder Traversal − Traverses a tree in a pre-order manner.
* Inorder Traversal − Traverses a tree in an in-order manner.
* Postorder Traversal − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

**Insert Operation**

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

**Algorithm**

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

goto right subtree

else

goto left subtree

endwhile

insert data

end If

**Search Operation:**

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

**Algorithm**

If root.data is equal to search.data

return root

else

while data not found

If data is greater than node.data

goto right subtree

else

goto left subtree

If data found

return node

endwhile

return data not found

end if

**Tree Traversal:**

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree−

* In-order Traversal
* Pre-order Traversal
* Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

**In-order Traversal**

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

*D → B → E → A → F → C → G*

**Algorithm**

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Visit root node.

**Step 3** − Recursively traverse right subtree.

**Pre-order Traversal**

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

*A → B → D → E → C → F → G*

**Algorithm**

Until all nodes are traversed −

**Step 1** − Visit root node.

**Step 2** − Recursively traverse left subtree.

**Step 3** − Recursively traverse right subtree.

**Post-order Traversal**

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

*D → E → B → F → G → C → A*

**Algorithm**

Until all nodes are traversed −

**Step 1** − Recursively traverse left subtree.

**Step 2** − Recursively traverse right subtree.

**Step 3** − Visit root node.Bottom of Form

Top of Form

**Introduction of B-Tree:**

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses.

Most of the tree operations (search, insert, delete, max, min, ..etc ) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

**Properties of B-Tree**

**1)** All leaves are at same level.

**2)** A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.

**3)** Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

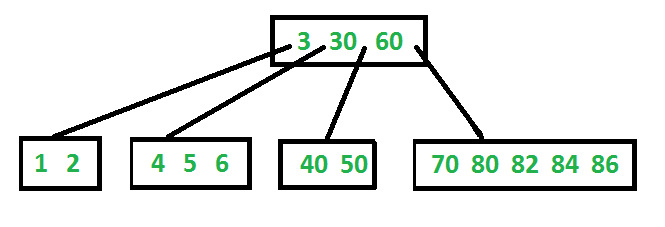
**4)** All nodes (including root) may contain at most 2t – 1 keys.

**5)** Number of children of a node is equal to the number of keys in it plus 1.

**6)** All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

**7)** B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

**8)** Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.

**Search**

Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don’t find k in the leaf node, we return NULL.

**Traverse**

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.Bottom of Form

**RELEVANT READING MATERIAL AND REFERENCES:**

**Source Notes:**

1. <https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm>
2. <https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm>
3. <https://www.geeksforgeeks.org/introduction-of-b-tree-2/>

**Lecture Video:**

1. <https://www.youtube.com/watch?v=qH6yxkw0u78>
2. <https://www.youtube.com/watch?v=OrhgXVwbSPY>

**Online Notes:**

1. <http://www.crectirupati.com/sites/default/files/lecture_notes/ds%20ln.pdf>
2. <http://www.vssut.ac.in/lecture_notes/lecture1428550942.pdf>

**Text Book Reading:**

1. Cormen, Leiserson, Rivest, Stein, “*Introduction to Algorithms*”, Prentice Hall of India, 3rd edition 2012. problem, Graph coloring.
2. Lipschutz, S., “*Data Structures, Schaum's Outline Series*”, Tata McGraw Hill.

**Online Book Reference:**

1. <https://www.edutechlearners.com/download/books/DS.pdf>

**In addition: PPT can be also be given.**