BINARY TREE

TREES

- Trees are hierarchical data structures.
- The topmost node is called the root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent.

WHY TREES?

- One reason to use trees might be because you want to store information that naturally forms a hierarchy.
- provide moderate access/search (quicker than Linked List and slower than arrays).
- provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
- Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Application of Trees

- Manipulate hierarchical data.
- Make information easy to search (see tree traversal).
- Manipulate sorted lists of data.
- Router algorithms

BINARY TREE

The tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Binary Tree Representation

https://github.com/jhavidit/DSA/blob/master/binaryTree/introduction/BinaryTreeRepresentation.java

Binary Tree Properties

- The maximum number of nodes at level 'l' of a binary tree is **2l**.
- The maximum number of nodes in a binary tree of height 'h' is **2^(h+1) – 1**.
- In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is **|Log2(N+1)|**.
- In a Binary tree where every node has 0 or 2 children, the number of leaf nodes is always one more than nodes with two children.
- A Binary Tree with L leaves has at least **| Log2(L) |+ 1** levels.

Types of Binary Tree

Full Binary Tree
A Binary Tree is a full binary tree if every node has 0 or 2 children.

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

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.

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 longest path from the root node to any leaf node in the tree, height of root node is 1) has 2h – 1 node.

Example - Ancestor tree

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.

Example - AVL Tree, Red-Black Tree

Enumeration of Binary Tree

Labeled tree - nodes of the tree are labeled.
Unlabeled tree - nodes of the tree are unlabeled.

Different unlabeled binary trees with n node

T(0) = 1 [There is only 1 empty tree]
T(1) = 1
T(2) = 2

T(3) = T(0)*T(2) + T(1)*T(1) + T(2)*T(0) = 1*2 + 1*1 + 2*1 = 5

T(4) = T(0)*T(3) + T(1)*T(2) + T(2)*T(1) + T(3)*T(0)
= 1*5 + 1*2 + 2*1 + 5*1
= 14

$$T(n) = \sum_{i=1}^{n} T(i-1)T(n-i) = \sum_{i=0}^{n-1} T(i)T(n-i-1) = C_n$$

Different labeled binary trees with n node

The number of Labelled Trees = (Number of unlabelled trees) * n!

Traversal

DFS(Depth First Search)

Inorder
Traverse left -> root - > right

T.C. - O(n)
S.C. - O(n)

Preorder

Traverse root->left->right
T.C. - O(n)
S.C. - O(n)

Postorder

Traverse left->right->root
T.C. - O(n)
S.C. - O(n)

https://github.com/jhavidit/DSA/blob/master/binaryTree/traversal/Traversal.java

Height(Depth) of binary tree

Following the recursive approach for every node the height till that point we calculate by finding the maximum height of left child subtree and right child subtree then add 1 to it.

https://github.com/jhavidit/DSA/blob/master/binaryTree/basic/HeightOfBinaryTree.java

Print Nodes at K distance

Recursively traverse to the left and right child nodes and decrease k when k==0 print that node value.

https://github.com/jhavidit/DSA/blob/master/binaryTree/basic/NodesAtKDistance.java

Level Order Traversal

Use queue data structure add root to queue iterate till queue is not empty pop from queue print the data and then push its left child and right child to the queue.

Time Complexity - O(n) number of nodes
Space Complexity - O(w) width of the tree

https://github.com/jhavidit/DSA/blob/master/binaryTree/basic/LevelOrderTraversal.java

Reverse Left Order Traversal

Similarly like left order traversal instead of printing the popped element from the queue we push it into the stack and after all, children have been pushed into the queue we pop elements from the stack and append to the front of the array list

https://github.com/jhavidit/DSA/blob/master/binaryTree/basic/ReverseLevelOrderTraversal.java

Iterative PreOrder Traversal

We use stack and push root nodes. We continue to pop and print and add its right and then left child node until the stack is empty.

https://github.com/jhavidit/DSA/blob/master/binaryTree/basic/IterativePreOrder.java

Iterative Inorder Traversal

We use stack and a node pointer we will push the pointer to stack go left till we do not find null and update the pointer when finding null if the stack is empty stop or pointer goes to right.

https://github.com/jhavidit/DSA/blob/master/binaryTree/basic/IterativeInOrder.java

Iterative PostOrder Traversal using 2 stack

Iterative PostOrder Traversal