Lesson:

Java

Binary Trees

PW SKILLS

Pre Requisites:

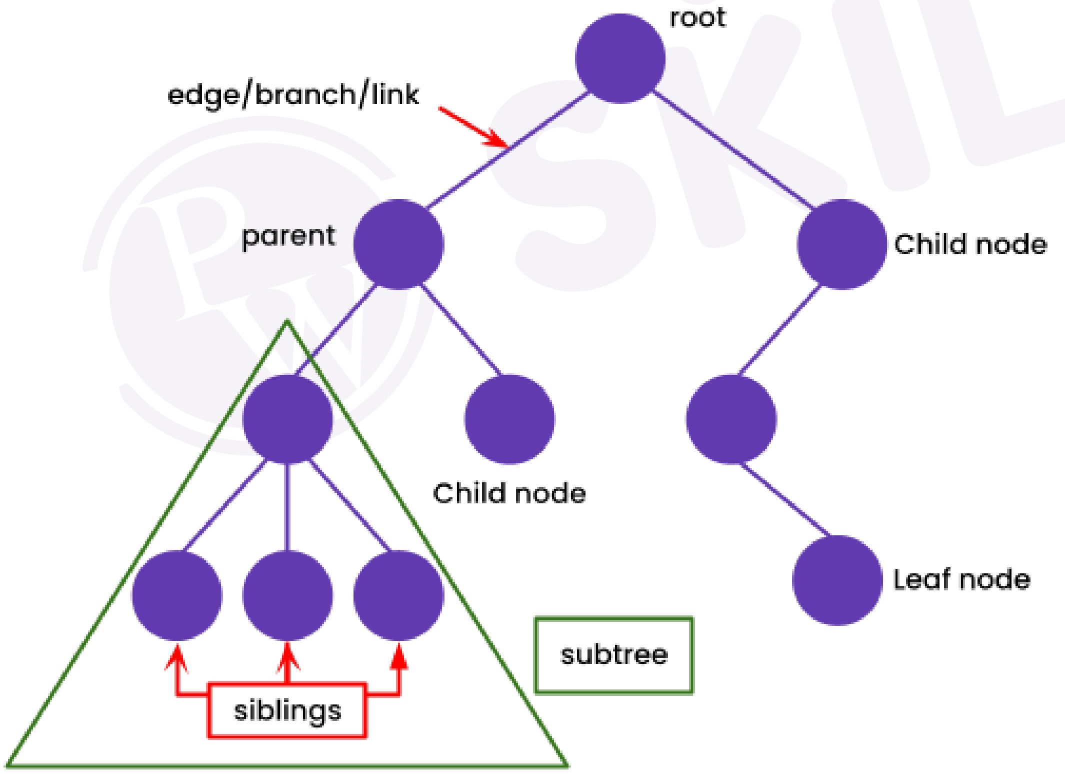
- Basic Java
- Recursion in Java

List of concepts involved :

- Introduction To Trees
- Introduction to Binary Trees
- Introduction to N-ary trees
- Traversal of tree
- Pre-order, Inorder, post-order
- Interview Problem: Height of Tree
- Interview Problem: Level of Tree

Introduction to Trees

A Tree is a non-linear data structure where each node is connected to a number of nodes with the help of pointers or references.



Basic Tree Terminologies:

- **Root:** The root of a tree is the first node of the tree.
- **Edge:** An edge is a link connecting any two nodes in the tree.
- **Siblings:** The children nodes of the same parents are called siblings of each other. That is, the nodes with the same parents are called siblings.
- **Leaf Node:** A node is said to be the leaf node if it has no children.
- **Height of a Tree:** Height of a tree is defined as the total number of levels in the tree or the length of the path

from the root node to the node present at the last level.

Introduction To Binary Tree

A Tree is said to be a Binary Tree if all of its nodes have at most 2 children. That is, all of its nodes can have either no child, 1 child or 2 child nodes.

Properties of Binary trees:

- The maximum number of nodes at level 'l' of a binary tree is 2^(i-1) Level of root is 1.
- Maximum number of nodes in a binary tree of height 'h' is (2^(h - 1)).
- In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is Log2(N+1).
- A Binary Tree with L leaves has at least (Log2L + 1) levels.
- 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.

Traversal of Tree

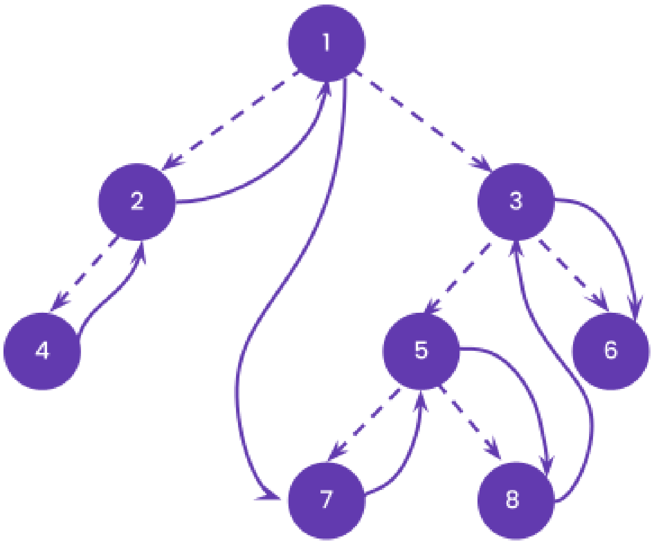
Unlike linked lists, one-dimensional arrays, and other linear data structures, which are traversed in linear order, trees can be traversed in multiple ways in **depth-first order** (**preorder**, **inorder**, and **postorder**) or **breadth-first order** **(level order traversal).**

Traversing a tree involves iterating over all nodes in some manner. As the tree is not a linear data structure, there can be more than one possible next node from a given node, so some nodes must be deferred, i.e., stored in some way for later visiting. The traversal can be done iteratively where the deferred nodes are stored in the **stack**, or it can be done by **recursion,** where the deferred nodes are stored implicitly in the **call stack.**

Inorder Traversal of tree

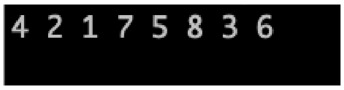
For traversing a (non-empty) binary tree in an inorder fashion, we must do these three things for every node n starting from the tree's root:
(L) Recursively traverse its left subtree. When this step is finished, we are back at n again.
(N) Process n itself.
(R) Recursively traverse its right subtree. When this step is finished, we are back at n again.

In normal inorder traversal, we visit the left subtree before the right subtree.



LP_Code1.java

Output



```
4 2 1 7 5 8 3 6
```

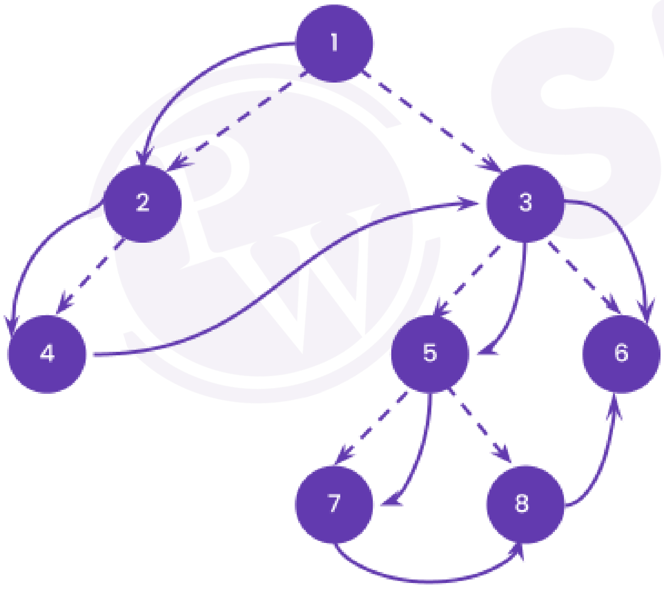
Approach:

- As we can see, before processing any node, the left subtree is processed first, followed by the node, and the right subtree is processed at last.
- These operations can be defined recursively for each node. The recursive implementation is referred to as a Depth–first search (DFS), as the search tree is deepened as much as possible on each child before going to the next sibling.

Pre-order of Tree

For traversing a (non-empty) binary tree in a preorder fashion, we must do these three things for every node n starting from the tree's root:
(N) Process n itself.
(L) Recursively traverse its left subtree. When this step is finished, we are back at n again.
(R) Recursively traverse its right subtree. When this step is finished, we are back at n again.

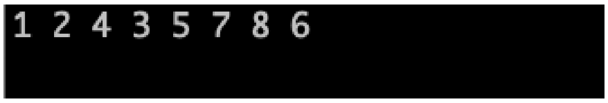
In normal preorder traversal, visit the left subtree before the right subtree.



Preorder: 1, 2, 4, 3, 5, 7, 8, 6

LP_Code2.java

Output:



```
1 2 4 3 5 7 8 6
```

Approach:

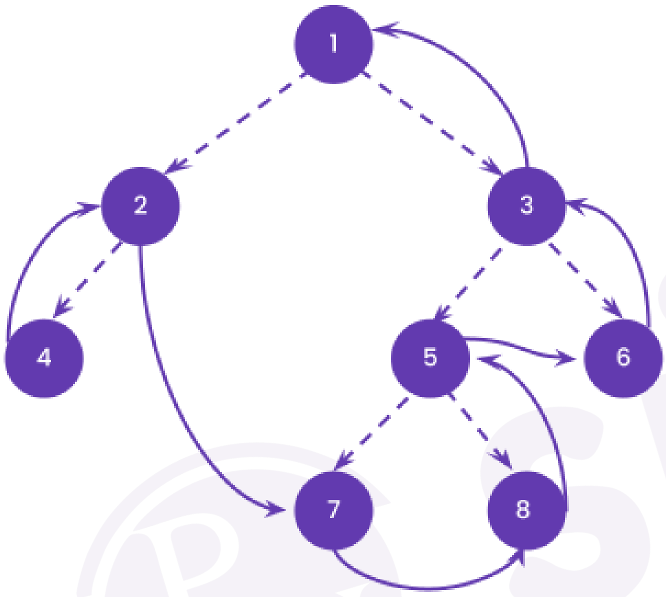
- As we can see, only after processing any node, the left subtree is processed, followed by the right subtree.
- These operations can be defined recursively for each node. The recursive implementation is referred to as a Depth–first search (DFS), as the search tree is deepened as much as possible on each child before going to

the next sibling.

Post Order of Tree

For traversing a (non-empty) binary tree in a postorder fashion, we must do these three things for every node n starting from the tree's root:
(L) Recursively traverse its left subtree. When this step is finished, we are back at n again.
(R) Recursively traverse its right subtree. When this step is finished, we are back at n again.
(N) Process n itself.

In normal postorder traversal, visit the left subtree before the right subtree.



Postorder: 4, 2, 7, 8, 5, 6, 3, 1

LP_Code3.java

Output:

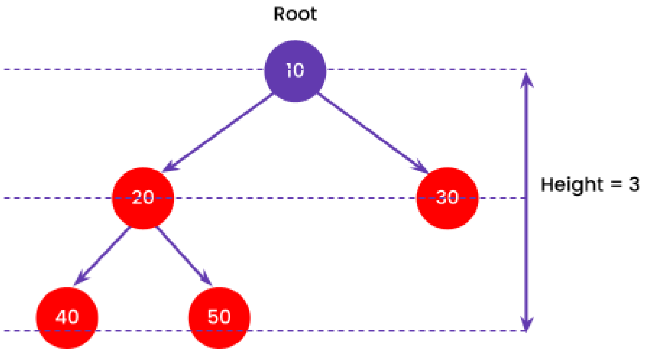
```
4 2 7 8 5 6 3 1
```

Approach

- As we can see, before processing any node, the left subtree is processed first, followed by the right subtree, and the node is processed at last.
- These operations can be defined recursively for each node. The recursive implementation is referred to as a Depth–first search (DFS), as the search tree is deepened as much as possible on each child before going to the next sibling.

Interview Problem: Height of Tree

As we know The height or depth of a binary tree is the total number of edges or nodes on the longest path from the root node to the leaf node.

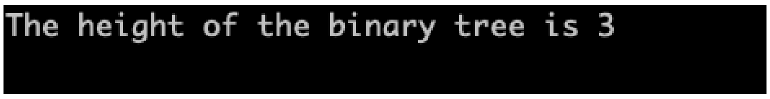


The Approach to solve the following problem is

- The idea is to traverse the tree in a postorder fashion and calculate the height of the left and right subtree.
- The height of a subtree rooted at any node will be one more than the maximum height of its left and right subtree.
- Recursively apply this property to all tree nodes in a bottom-up manner (postorder fashion) and return the subtree's maximum height rooted at that node.

LP_Code4.java

Output:



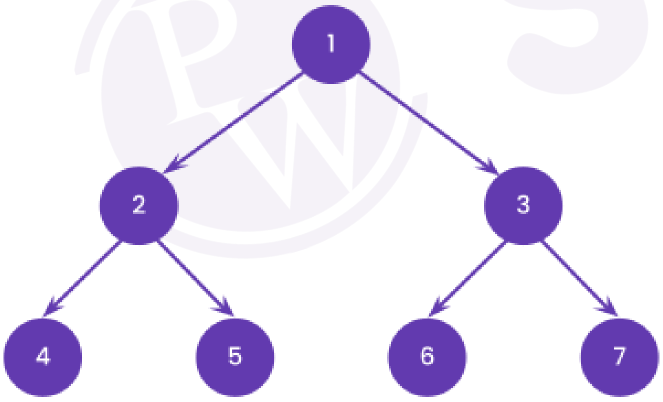
```
The height of the binary tree is 3
```

The time complexity of the above recursive solution is O(n), where n is the total number of nodes in the binary tree. The auxiliary space required by the program is O(h) for the call stack, where h is the height of the tree.

Interview problem: Level Order Traversal

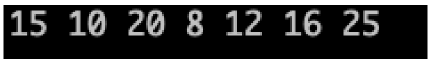
Trees can also be traversed in level order, where we visit every node on a level before going to a lower level. This search is referred to as level order traversal or Breadth–first search (BFS), as the search tree is broadened as much as possible on each depth before going to the next depth.
Level order traversal basically means traverse tree nodes level by level, i.e., print all nodes of level 1 first, followed by nodes of level 2 and so on...

For example, the level order traversal for the following tree is 1, 2, 3, 4, 5, 6, 7:



LP_Code5.java

Output:



```
15 10 20 8 12 16 25
```

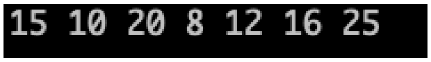
The time complexity of the above solution is O(n2), where n is the total number of nodes in the binary tree. The auxiliary space required by the program is O(h) for the call stack, where h is the height of the tree.
We can reduce the time complexity to O(n) by using extra space. Following is a pseudocode for a simple queue-based level order traversal, which requires space proportional to the maximum number of nodes at a given depth. It can be as much as half the total number of nodes.
levelorder(root)

```
• q —> empty queue
• q.enqueue(root)
• while (not q.isEmpty())
•   node —> q.dequeue()
•   visit(node)
•   if (node.left <> null)
•     q.enqueue(node.left)
```

```
 if (node.right <> null)
   q.enqueue(node.right)
```

LP_Code6.java

Output:



```
15 10 20 8 12 16 25
```

The time complexity of the above solution is O(n) and requires O(n) extra space, where n is the size of the binary tree.