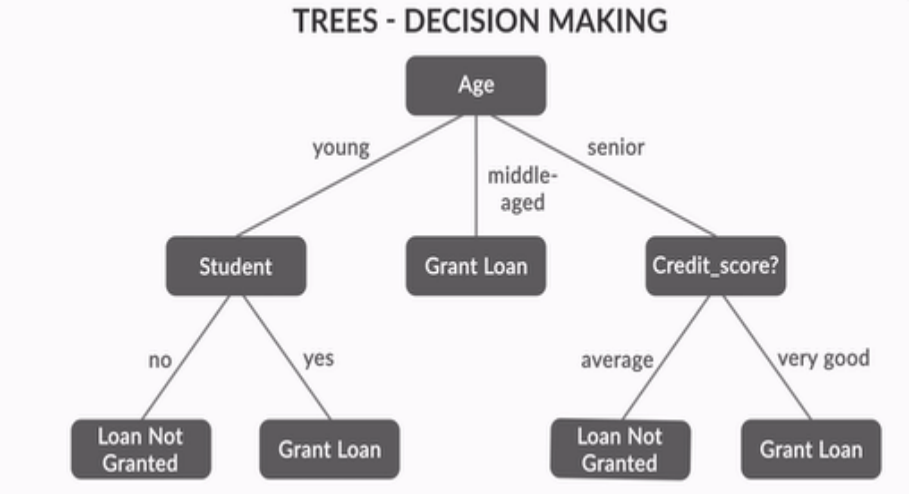
Lecture Notes

Binary Trees and BSTs

So, in this module you learnt about a non- linear data structure called Trees. The first session focused on a special variant of trees called binary trees. The second session dealt majorly with a special variant of Binary Trees called Binary Search Trees.

Introduction to Trees

In this segment you learnt about the structure of trees which can store non-linear data with various data points having specific relation with each other. You learnt that trees naturally convey the information concealed within a data set. All this was discussed with the help of few examples. One of the examples that was discussed was a decision-making tree representing the criteria used to grant loan to a bank customer.



Components of a Tree

Here you learnt the following

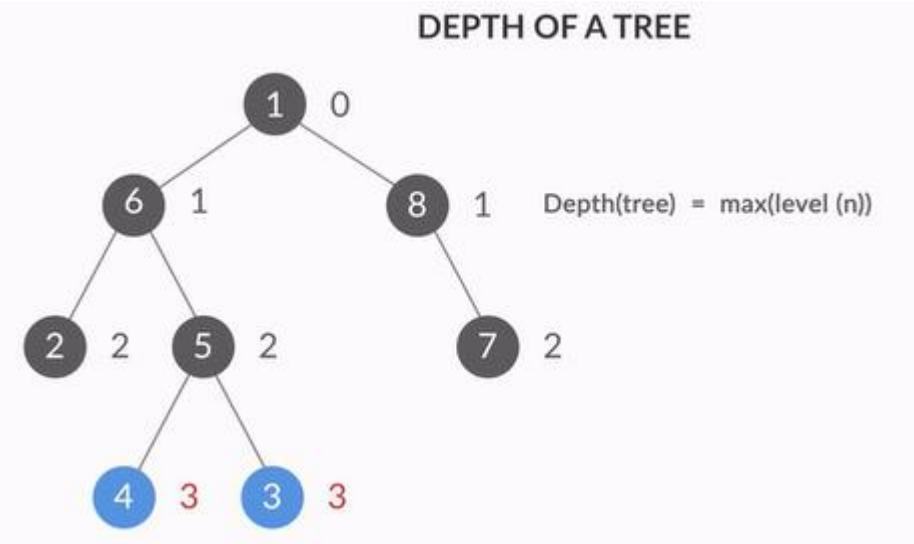
- Nodes of a Tree - Each data point in the tree with some value is the node of a tree
- Root Node – A node which does not have any parent is called the root node of the tree
- Leaf Node – A node without any child/children is the leaf node
- Internal Node – Any node which is not the leaf node is internal node
- Edges – The connection between 2 nodes. An edge connects one node to another node
- Definition of a tree –
    - It should have only one root node
    - A node should have exactly one parent
    - No cycles are allowed in a tree

Binary Trees

In this segment, you learnt about a special variant of trees called Binary Trees. You learnt that a tree with a degree 2 is called a Binary Tree. Degree is defined as the maximum number of child nodes a node can have. So, a tree where a node can have maximum of 2 child nodes is called a binary tree. We also discussed that unlike lists, trees do not have any natural ordering within its nodes. So we had to define certain traversal techniques based on which we could allot some ordering within the nodes of a tree

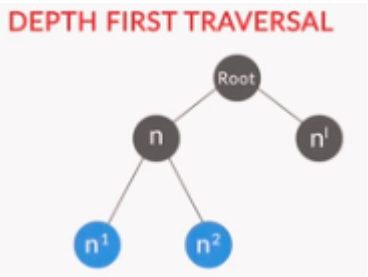
Depth First Traversal

Next, we moved onto a traversal technique called Depth First Traversal. Before learning what DFS is, we defined Depth of a Tree as the maximum level a node can have.

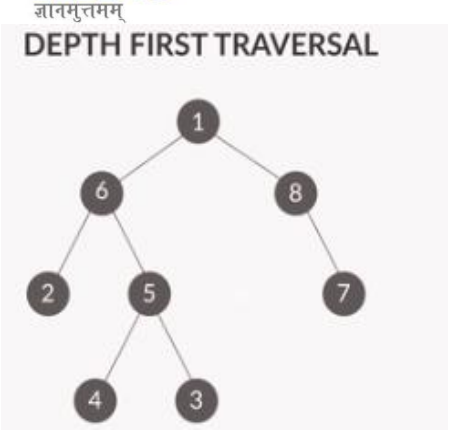


In the image above, nodes 3 and 4 have the maximum level in the tree which is 3. So the depth of this three is 3.

The basic ideology of Depth first traversal (DFS) was explained with the help of the following image



It was discussed that if the node n is visited and its sibling node n' and its child nodes n^1 and n^2 are not visited, then before visiting n' n^1 and n^2 are visited. We move along the depth first and then the breadth is traversed.
Following example was used in the video to explain the same



The DFS for the tree comes out to be 1,6,2,5,4,3,8.

Following was the pseudocode that was discussed for DFS

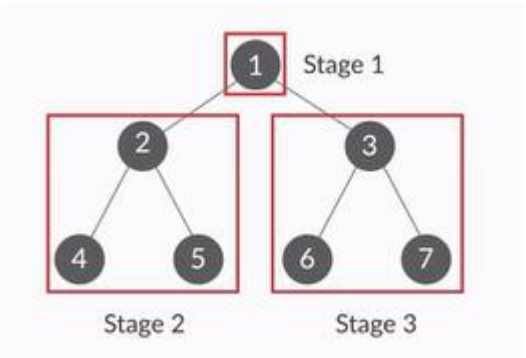
```
dfs (tree)
dfs_node(start(tree))
dfs_node(n)
  if (left (n) exists)
    dfs (left (n))
  if (right (n) exists)
    dfs (right (n))
```

Variants of DFS

You learnt about 3 variants of DFS namely

- Post Order Traversal
- Pre Order Traversal
- In Order Traversal

We discussed that any visit to a node in a tree has 3 stages as depicted in the figure given below.



Stage 1: When none of the child nodes of a node is visited
Stage 2: When the Depth First Traversal of the left child is taken care of
Stage 3: When the Depth First Traversal of the right child is taken care of.

Depending upon at which of these stages the action is performed on a node, we defined the particular variant of DFS.

When the action is performed during the stage 1 we call it Pre Order Traversal

When the action is performed during the stage 2 we call it In Order Traversal

When the action is performed during the stage 3 i.e. once the right child has been taken care of we call it Post Order Traversal.

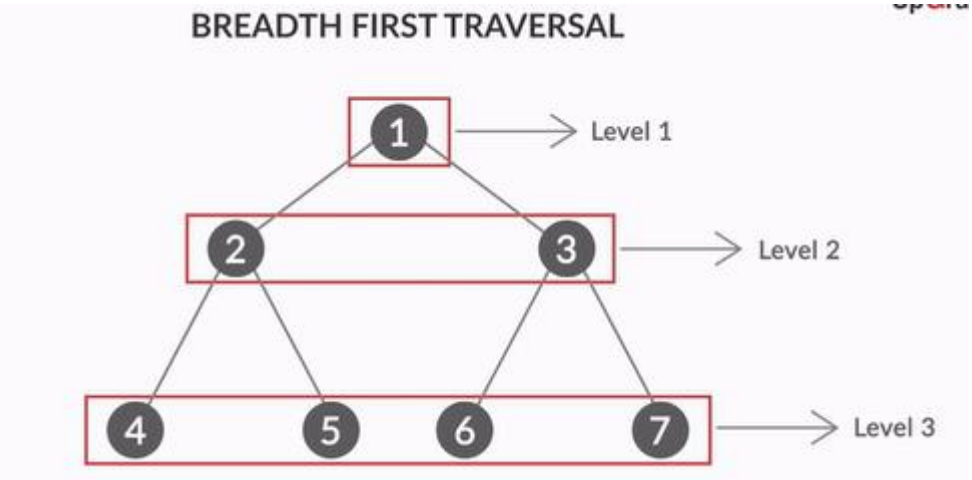
Following pseudocode was discussed for the same.

```
dfs (tree)
dfs_node(start(tree))

dfs_node(n)
  if (left (n) exists)          ---> Preorder
    dfs (left (n))              ---> Inorder
  if (right (n) exists)
    dfs (right (n))             ---> Postorder
```

Breadth First Traversal

The next traversal technique that was discussed was Breadth First Traversal (BFS). You learnt that in BFS, also known as Level Order Traversal, all the nodes of a level are visited first before we move onto the next level.



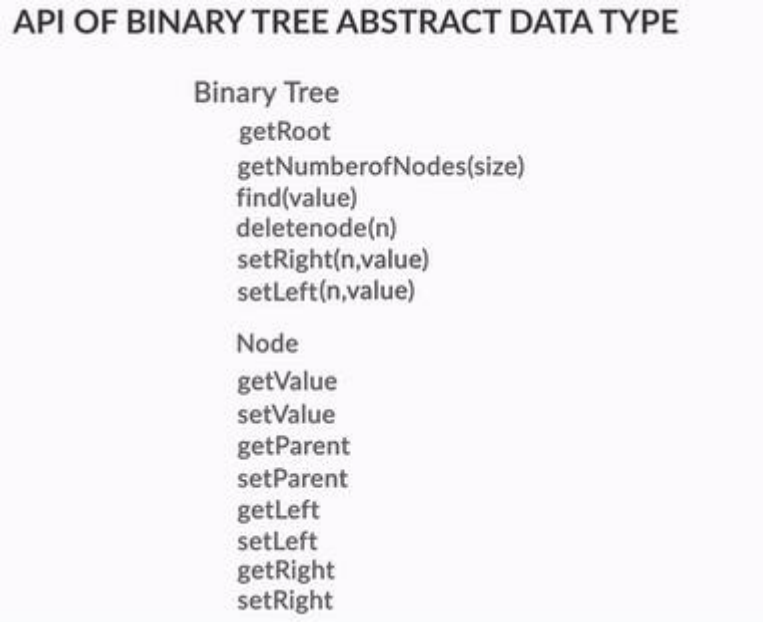
In the figure above, all the nodes of Level 1 are visited first. Then we move onto level 2 where all the nodes i.e node 2 and node 3 are visited before moving onto Level 3 where nodes 4,5,6 and 7 are visited.

The Java implementation of BFS was discussed using Queues. Following was the pseudocode discussed.

```
bfs(t)
 Q = new queue
  enqueue(Qroot(t))
  while(Q is not empty)
    n = dequeue(Q)
    action n
If(n has a left child)
 enqueue(left child)
If(n has a right child)
 enqueue(right child)
End while
end while
```

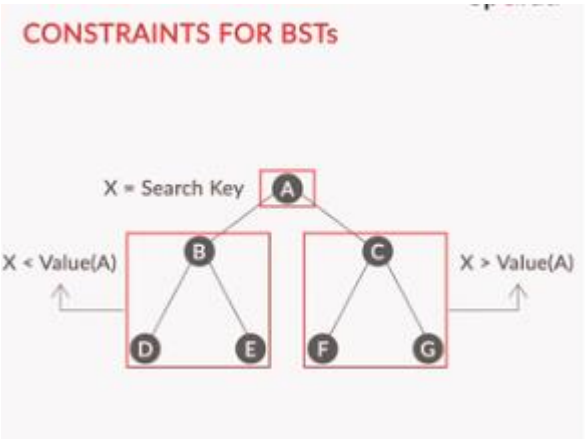
API Of Binary Abstract Data Type

To conclude the session, we discussed the API of Binary Abstract Data Type as can be seen in the image below.



Session 2 – Binary Search Trees

In this session, you learnt about a special variant of Binary Trees called Binary Search Trees or BSTs. The following constraints make Binary Tree a Binary Search Tree



- The values of all the nodes in the left subtree are less than the root node.

- The values of all the nodes in the right subtree are greater than the root node.
- Again, each subtree behaves like a binary search tree

Search in a BST

You learnt that searching in a BST is a $\log_2 N$ operation. This is because every time we move onto the next node, we conclusively ignore the remaining half of the tree from our search space. Following was discussed

- We compare the search key with the root.
- If the search key is greater than the root, we move onto the right subtree of the root. Else we move onto the left subtree of the root, thus ignoring one half of the tree.
- In case search key = root, we return the root itself.
- When we move onto the left or right subtree, we again compare the search key with the node we are at.
- Depending upon the search key is greater or less than the current node, we move onto the left or right of this node and so on.
- So at every step the remaining half of the tree is ignored thus giving it a logN efficiency.

Following pseudocode was discussed for its Java implementation.

```
find(n,v)
If value(n) = v then
 Return n
If n is a leaf node
 Return null
If v<value(n) then
 Return find(left(n),v)
Else
 Return find(Right(n),v)

find(tree,v)
find(root(tree),v)
```

Adding a node to a BST

Adding a node to a BST in principle is very similar to the search operation. You learnt that while adding a node to a BST, you first must figure out the position at which the node can be added. It was discussed that a new node can only be added as a leaf node. So, adding a node with a given value is like finding the node with this value in the binary tree. Since this node does not exists in the tree, you will end up at a leaf node where you can add this node. This gives a fair intuition as to why adding a node is also a logN operation.

Following pseudocode was discussed for tis Java implementation.

```
Add Node(parent, node)
    If parent=node then
        return
    If parent > node then
            If (left(parent)exists) then
                add Node(left(parent), node)
             else,   /* no left child */
                make node the left child of parent
    If parent < node then
            If (right(parent)exists) then
                add Node(right(parent), node)
            else
                make node the right child of parent
```

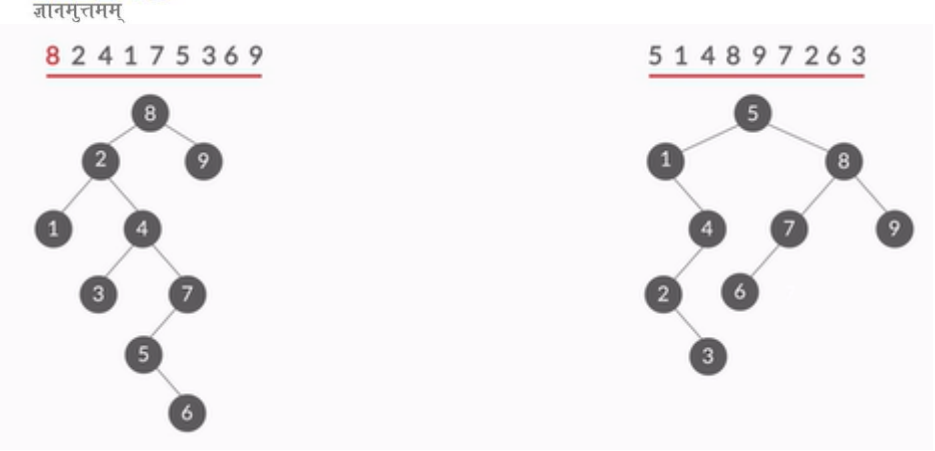
Deleting a Node

Next, you learnt about deleting a node from a BST. You learnt that the deletion of data from binary search trees can be carried out in three ways:

1. If the node to be deleted is a leaf node, it is directly removed from the tree.
2. If the node has one child, the node itself is deleted, and its child node is connected to its parent node.
3. If the node has two children, you find its successor to replace it. The successor node is the minimum node in the right subtree or the maximum node in the left subtree.

Balanced BSTs

Here you learnt that given a data set different BSTs can be created from it. It all depends on the sequence in which the elements are inserted. One such example can be seen in the image below.

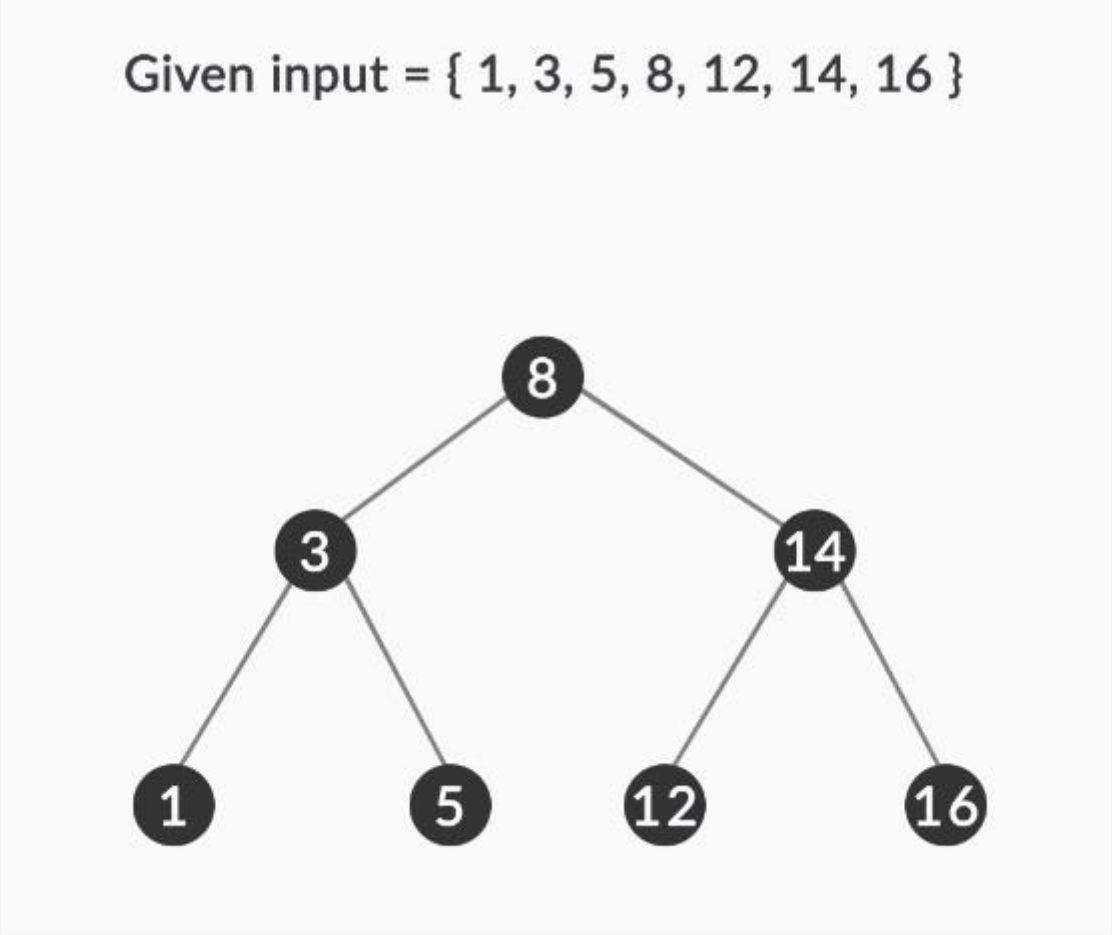


If the elements are inserted in the increasing order of their values, the tree essentially becomes a list and all the operations viz. adding a node, deleting a node or searching for an element become O(N) efficient.

Thus, to make sure that the operations are efficient the tree should be balanced i.e. the number of nodes in the left subtree should be comparable to the number of nodes in the right subtree. Ideally this number should be equal for both left and right subtrees.

So, pre-processing of data can improve the search efficiency of binary search trees.

Hence, the sorted data can be stored by choosing the median of all the elements as the root of the tree and iterating the same for all the subtrees, as shown in the figure below.



Here, 8 is the middle element. So, 8 is selected as the root of the BST. The middle element of the left part of the array is 3. So, 3 is the root of the left subtree. The elements 1 and 5 will become its left and right children, respectively. Similarly, 14 will be the middle element of the right part of the array, or the root node of the right subtree. Similarly, 12 and 16 will become its left and right children, respectively.

Storing the data in this way will always produce balanced binary search trees and, in turn, will enhance the search process efficiency as it reduces the number of comparisons required in the trees. You can read more about creating balanced BSTs from the data given in this link.

Additionally, you can read about self-balancing BSTs that will attempt to balance itself when new nodes are added to the tree. Find more information about self-balancing trees in the links provided below.

- Red-black trees
- AVL trees