Unit - 2 Advanced Data Structures

Trees:

What is Tree?

The tree structure consists of the following components:

- Node: A node is a fundamental unit of a tree that contains some data or information. It may also have a link or reference to its child nodes.
- Edge: An edge is a connection or link between two nodes. It represents the relationship between the nodes.
- Root: The root is the topmost node of the tree. It serves as the starting point or the parent node for all other nodes in the tree.
- Parent and Child: Nodes in a tree can have a hierarchical relationship. A node that is connected to another node through an edge and is closer to the root is considered the parent of the connected node, while the connected node is referred to as the child of the parent node.
- Leaf: A leaf node is a node that does not have any child nodes. It represents the end or termination point of a branch in the tree.
- Subtree: A subtree is a smaller tree that is part of a larger tree. It consists of a node and all its descendants.
- Level: The level of a node represents the generation of a node in the tree. The root node is at level 0, its children are at level 1, and so on.
- Height: The height of a tree is the length of the longest path from the root node to any leaf node in the tree.
- Degree: The degree of a node is the number of its children.
- Siblings: Nodes that have the same parent are called siblings.
- Ancestor: An ancestor of a node is any node that is on the path from the root node to the parent node of the node.
- Descendant: A descendant of a node is any node that is on the path from the node to one of its leaf nodes.
- Path: A path is a sequence of nodes that are connected by edges. It represents the traversal from one node to another.

Types of Trees

There are many different types of trees. Some of the most common types of trees are as follows:

- *Binary Tree*: A binary tree is a tree in which each node has at most two children. The children of a node are referred to as the left child and the right child. The left child is the first child of the node, and the right child is the second child of the node.
- *Complete Binary Tree*: A complete binary tree is a binary tree in which all the levels of the tree are completely filled except possibly the last level. In the last level, all the nodes are as far left as possible.
- *Binary Search Tree*: A binary search tree is a binary tree in which the value of each node is greater than or equal to the value of all the nodes in its left subtree and less than or equal to the value of all the nodes in its right subtree.
- *AVL Tree*: An AVL tree is a self-balancing binary search tree. It is named after its inventors, Adelson-Velsky and Landis. It is a binary search tree in which the heights of the left and right subtrees of each node differ by at most one.
- *Red-Black Tree*: A red-black tree is a self-balancing binary search tree. It is named after its inventors, Rudolf Bayer and Rudolf Mehl. It is a binary search tree in which each node has a color either red or black. The root node is always black. The children of a red node are always black. The children of a black node can be either red or black.
- *B-Tree*: A B-tree is a self-balancing tree. It is named after its inventors, Bayer and McCreight. It is a tree in which each node can have more than two children. It is commonly used in databases and file systems.
- *B+ Tree*: A B+ tree is a self-balancing tree. It is named after its inventors, Bayer and McCreight. It is a tree in which each node can have more than two children. It is commonly used in databases and file systems.
- *Threaded Binary Tree*: A threaded binary tree is a binary tree in which each node has a link to its inorder successor and inorder predecessor. It is used to make inorder traversal faster and more efficient.

Binary Search Tree

While creating a BST, we need to follow the following rules:

```
1. The value of the left child of a node should be less than or equal to
the value of the node.
2. The value of the right child of a node should be greater than or equal
to the value of the node.
3. If an equal value is added to the right side of the root node then it
is a right biased BST.
4. If an equal value is added to the left side of the root node then it is
a left biased BST.
```

In a BST if you traverse to the left most node of the BST you will find out that it is has the lowest values in the BST. Similarly, if you traverse to the right most node of the BST you will find out that it is has the highest values in the BST.

Create a BST with following keys (4, 2, 3, 6, 5, 7, 1):

```
      4
     / \
    2   6
   / \ / \
  1   3 5   7
```

Inorder of the BST created: 1, 2, 3, 4, 5, 6, 7 *Preorder of the BST created:* 4, 2, 1, 3, 6, 5, 7 *Postorder of the BST created:* 1, 3, 2, 5, 7, 6, 4

Deletion in BST:

Given BST:

```
           25
          /    \
        10      35
      /    \      \
     5      20     40
    / \    /  \    /
   3   6  15   21  37
         /
        13
```

1. If the node we want to delete is a `leaf node`, then we can simply delete the node.
2. If the node we want to delete has only one child, then we can simply delete the node and replace it with its child.
3. If the node we want to delete has two children, then there are two methods:-

   ```
    a. Find the largest element in the left subtree of the node and
   replace the node with it. (Predecessor of the node)
    b. Find the smallest element in the right subtree of the node and
   replace the node with it. (Successor of the node)
   ```

4. If the node we want to delete is a root node, then we can simply delete the node and replace it with its successor or predecessor.

Question: *The `Preorder traversal` of BST is (30, 20, 10, 15, 25, 23, 39, 35, 42). Which of the following is `Postorder traversal` of same tree?*

```
a. 10, 15, 20, 23, 25, 30, 35, 39, 42
b. 15, 10, 23, 25, 20, 35, 42, 39, 30
```

```
c. 15, 10, 25, 23, 20, 42, 35, 39, 30
d. 15, 10, 23, 25, 20, 39, 35, 42, 30
```

So, according to the `preorder traversal` given to us in the question the tree is:

```
           30
          /  \
         20   39
        / \   / \
       10 25 35 42
        \  /
        15 23
```

Now, the `postorder traversal` of the tree is:

```
15, 10, 23, 20, 25, 35, 42, 39, 30
```

Inorder Preorder Postorder

- Preorder Traversal: In preorder traversal, the root node is visited first, followed by the left subtree and then the right subtree. Root -> Left -> Right
- Inorder Traversal: In inorder traversal, the left subtree is visited first, followed by the root node and then the right subtree. Left -> Root -> Right
- Postorder Traversal: In postorder traversal, the left subtree is visited first, followed by the right subtree and then the root node. Left -> Right -> Root

Important Question for BST

Question: A BST sotres vaules in the range 37-573. Consider following sequence of Keys.

```
1. 87, 537, 102, 439, 285, 376, 305
2. 52, 97, 121, 195, 242, 381, 472
3. 142, 248, 520, 386, 345, 270, 307
4. 550, 149, 507, 395, 463, 402, 270
```

Suppose the BST has been unsuccessfully searched for key 273. Which of the above sequence list nodes in the order in which we could have encountered them in the search?

Solution: The BST for the first sequence will be

```
      87
    /   \
537     102
        /  \
      439  285
           /  \
         376  305
```

The BST for the second sequence will be

```
      52
    /     \
  97       121
 /  \     /  \
195 242 381 472
```

The BST for the third sequence will be

```
         142
        /   \
      248   520
      /       /
    386     345
   /   \
270   307
```

The BST for the fourth sequence will be

```
    550
   /   \
149   507
      /  \
    395  463
           \
           402
           /
         270
```

AVL Tree

An `AVL tree` is a `self-balancing binary search tree` in which the height difference between its left and right subtrees is always kept at most 1. This balance condition ensures efficient `search`, `insertion`, and `deletion` operations with a time complexity of `O(log n)`, where n is the number of elements in the tree. `AVL trees` automatically perform rotations to maintain balance when necessary, making them a reliable data structure for storing and retrieving ordered data.

Advantages of AVL Trees:

- *Efficient operations:*AVL trees maintain a balance condition, which guarantees that the height of the tree is logarithmic in the number of elements. This ensures efficient search, insertion, and deletion operations with a time complexity of O(log n).
- *Self-balancing:* AVL trees automatically perform rotations to maintain balance after insertions or deletions. This self-balancing property ensures that the tree remains relatively balanced, preventing the tree from becoming highly skewed and maintaining optimal performance.
- *Predictable performance:* Due to their self-balancing nature, AVL trees provide predictable performance for various operations. This makes them suitable for real-time applications and situations where a consistent time complexity is essential.

Disadvantages of AVL Trees:

- *Additional overhead:* The self-balancing mechanism of AVL trees requires additional overhead in terms of memory and processing. Each node needs to store the balance factor, which adds extra memory requirements compared to simple binary search trees
- *Complex implementation:* Implementing and maintaining AVL trees can be more complex compared to basic binary search trees. The rotation operations and balance factor updates need to be carefully managed to ensure correctness and efficiency.
- *More restrictive than other balanced trees:* AVL trees maintain stricter balance conditions compared to some other self-balancing trees like red-black trees. This can lead to more frequent rotations and potential performance overhead in certain scenarios.

Creating an AVL Tree:

Create an AVL Tree for the given keys: 55, 25, 65, 9, 8, 15

Solution: 8 \ R 9 \ R 10

1st it is 0-2 = -2 so we need to do right rotation

```
  9
 / \
8   10
```

Now, it is 1 - 1 = 0 so it is balanced (RR rotation)

2nd it is 2 - 0 = 2, so we need to do left rotation

```
     10
 L   /
    9
L  /
 8
```

Now, it is 1 - 1 = 0 so it is balanced (LL rotation)

```
    9
   / \
  8  10
```

3rd, taking 10, 12, 11 the initial tree is:

```
    10
      \
      12
      /
    11
```

Now, swapping 11 and 12, the tree becomes:

```
    10
      \
      11
        \
         12
```

Now, applying LL rotation, the tree becomes:

```
      11
     /  \
   10    12
```

4th, taking 10, 8, 9 the initial tree is:

```
    10
   /
  8
   \
    9
```

Now, swapping 8 and 9, the tree becomes:

```
    10
   /
  9
 /
8
```

Now, performing RR rotation, the tree becomes:

```
    9
   / \
  8   10
```

Question: How many rotations are required during construction of an AVL ttree if the following elements are added in order given? `35, 50, 40, 25, 30, 60, 78, 20, 28`

- 2 left, 3 right
- 3 left, 2 right
- 2 left, 2 right
- 2 left, 1 right

Solution:

```
   35
     \
      50
     /
   40
```

Now as we see, the balance factor is not right so initially `Right Rotating` 40 and 50, the tree becomes:

```
   35
     \
     40
       \
       50
```

Now, performing `Left rotation` to resolve `RR unbalance`, the tree becomes:

```
     40
    /  \
  35    50
```

Now, adding further elements the tree becomes:

```
        40
       /  \
     35    50
    /
  25
    \
     30
```

Now, as we see another balance issue as arrived, so initially `Left Rotating` 25 and 30, the tree becomes:

```
        40
       /  \
     35    50
    /
  30
```

/ 25

Now, performing `Right rotation` to resolve `LL unbalance`, the tree becomes:

```
        40
       /  \
      30   50
     / \
    25 35
```

Now, adding further elements the tree becomes:

```
        40
       /  \
      30   50
     / \     \
    25 35     60
                \
                 78
```

Now, as we see another balance issue as arrived, and the issue is `RR issue` so doing `Left rotation`, the tree becomes:

```
        40
      /    \
     30    50
    / \    / \
   25 35  60  78
```

So, the total number of rotations are `3 left` and `2 right`.

Threaded Binary Tree

A `threaded binary tree` is a binary tree in which every node that does not have a right child has a `thread` (or a link) to its `inorder successor`. The threads are also known as `inorder successors`. The `inorder traversal` of a threaded binary tree is `non-recursive` and `non-morris`.

A Threaded Binary Tree consists of three fields `left`, `right`, and `thread` to store the left child, right child, and inorder successor of the current node respectively.

Example:

Consider the following binary tree:

```
       A
     /   \
    B     C
   / \   / \
  D   E F   G
```

The inorder traversal of the above tree is `D B E A F C G`. The inorder successor of `D` is `B`, the inorder successor of `B` is `E`, the inorder successor of `E` is `A`, and so on, similarly, the inorder predecessor of `A` is `E`, the inorder predecessor of `E` is `B`, and so on. Also, the inorder successor of `G` is `NULL` and the inorder predecessor of `D` is `NULL`.