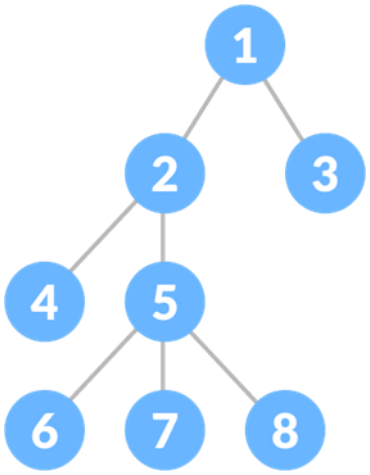
MODULE-4

Trees

In computer science, trees are extensively employed and essential to many algorithms and data storage uses. The terms that are essential to understanding tree data structures are listed below.



Tree

Nodes are 1, 2, 3, 4, 5, 6, 7, and 8. A root node is 1. 2 and 3 are parents of 1.

2, 3, are brothers and sisters. The leaf nodes are 4, 6, 7, and 8. The internal, or non-leaf, nodes are 2, 5. The tree's height or depth is three.

A tree with two root nodes is called the left subtree. There is just one node in the Right-Sub-Tree, which is node 3.

1. **Tree:** An edge-connected hierarchical data structure with nodes that are each connected by at least one child node.

2. **Node:** A single link in a tree data structure that is composed of linkages to its child nodes as well as data.

3. **Root:** The highest point in a tree, which is where a tree journey begins. It is parentless.

4. **Parent Node:** In a tree data structure, a parent link is a node that is connected to one or more child nodes. In the hierarchy, it is positioned above its progeny.

5. **Child Node:** In a tree, nodes that are related to a parent node. In the hierarchy, they are beneath the parent node.

6. **Leaf Node (Terminal Node):** An unbroken link in a tree or a node with a zero out-degree.

7. **An internal or non leaf link** in a tree is one that has one or more child nodes.

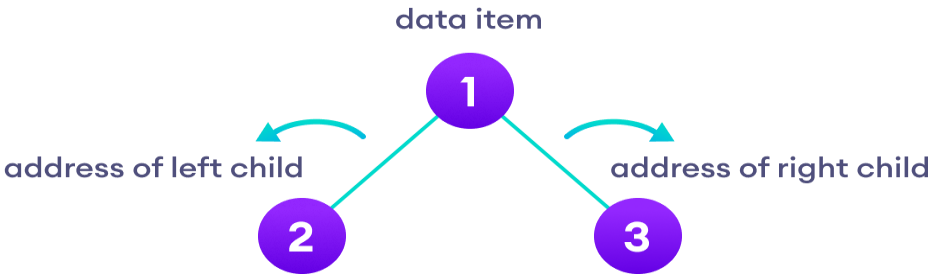
8. **Siblings:** In a tree, siblings are nodes that have the same parent node.

9. **Depth:** A node's elevation or separation from the root node. The root node has a height of zero.

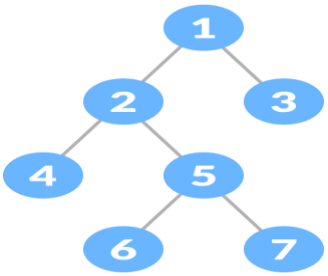
10. **Height or Depth of Tree:** The maximum distance that a tree's root can go to reach a leaf node. It stands for the tree's deepest point.

11. **Subtree:** A tiny tree contained within a bigger tree. It is made up of a node and every child that it has.

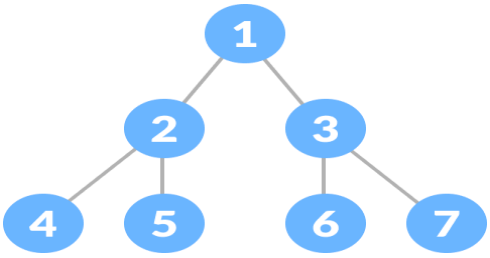
12. **Binary Tree:** A tree where each node has a maximum of two children, known as the left and right children.



Full-Binary Tree :



Perfect Binary-Tree:



13. **Balanced Tree:** Also known as a height-balanced binary tree, a balanced binary tree is one in which the difference in height between a node's left and right subtrees is limited to 1.

The prerequisites for a height-balanced binary tree are as follows:

1. For each node, there is only one difference between the left and right sub trees.

2. The balance of the left-sub-tree.

3. A balanced sub tree is selected.

Binary Tree in Balance:

Complete Binary Tree: In a complete binary tree, every level of the tree is filled to the brim, maybe with the exception of the final level, which is filled from left to right. Alternatively put, in an entire binary tree:

1. There are nodes on every level, possibly with the exception of the final one.

2. Nodes are added from left to right if the final level is not entirely occupied.

A complete binary tree has the following essential features:

1. It is a binary tree.

2. Because it is perfectly balanced, the tree's height is kept to a minimum relative to the number of nodes it has.

3. Because the structure is complete, adding and removing nodes from a binary tree is usually efficient.

1. **Traversal:** Also known as in-order, pre-order, or post-order traversal, this is the process of going through every node in a tree data structure in a particular order.

2. **In-order Traversal:** This method visits nodes in a binary tree depth-first, visiting the left subtree, the current node, and the right subtree in that order.

3. **Pre-order Traversal:** This is a depth-first binary tree traversal in which the current node, the left subtree, and the right subtree are visited in that order.

4. **Post-order Traversal:** A depth-first traversal of a binary tree in which the current node, the left subtree, and the right subtree are visited in that order.

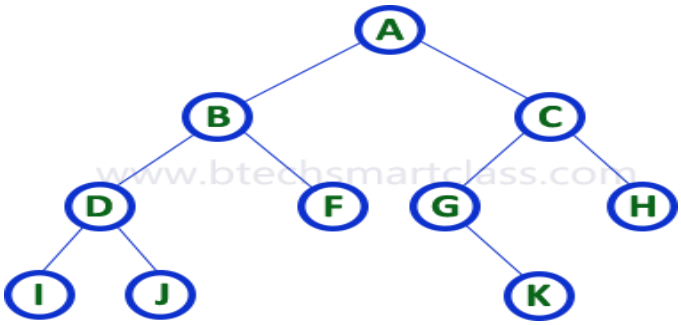
5. **Order of Levels Traversal:** A breadth-first approach to examining a tree's nodes level by level, beginning at the root.

Binary tree representations: There are two ways to represent a binary tree data structure. These techniques are listed below.

1. The Array Diagram

Linked List Illustration No. 2

Take a look at the binary tree below.



Binary Tree Array Representation

One-dimensional arrays, or 1-D arrays, are used in array representations of binary trees. Let's say that the array index is 1. Examine the binary tree example above, which is shown in the tailing manner.

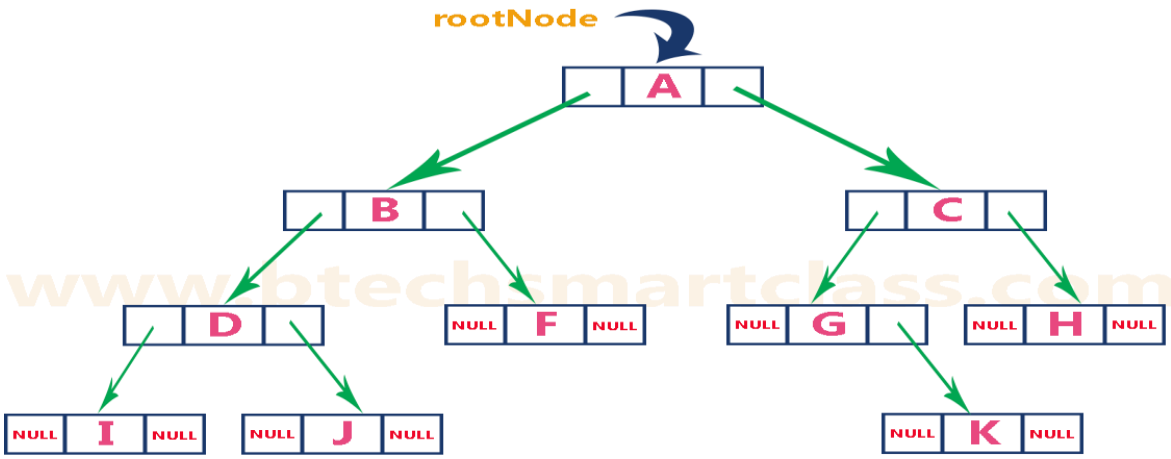
An array is utilised to store a tree's root[1]. The left child of a node, let's say m, is stored in array[2*k], and the right child of a node, let's say m, is stored in array[2*k+1].

One-dimensional arrays with maximum sizes of 2n + 1 are required in order to use array representation to mean a binary tree of level 'n'.

Binary Tree Representation in Linked Lists

We represent a binary tree using a double-linked list. Each node in a double-linked list has three fields. The left child address is stored in the first field, the actual data is stored in the second, and the right child address is stored in the third.

A node in this representation of a linked list has the structure shown below.



The binary tree example above that uses a linked-list internal representation is displayed as follows.

C **Tree Traversal Algorithms:**

A binary-tree must be traversed by visiting each node in a certain order. In-order traversal, pre-order traversal, and post-order traversal are the three common techniques for navigating a binary tree. Every method specifies a distinct order for visiting nodes.

```
//#include <stdio.h>
//#include <stdlib.h>
// Define a basic binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
// In-order traversal: left, root, right
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}
// Pre-order traversal: root, left, right
void preOrderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}
```

```
// Post-order traversal: left, right, root
void postOrderTraversal(struct Node* root) {
    if (root != NULL) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        printf("%d ", root->data);
    }}
int main() {
    // Creating a sample binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
     // Perform traversals
     printf("In-order traversal: ");
     inOrderTraversal(root);
     printf("\nPre-order traversal: ");
     preOrderTraversal(root);
     printf("\nPost-order traversal: ");
     postOrderTraversal(root);
     return 0;
}
```

1. **Recursion-Based In Order Traversal Implementation**

a. First visit the left node.

b. Then visit the root node.

c. Then visit the right node.

In the above order specified when performing an in order traversal.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def inorder_traversal(node):
    if node:
        inorder_traversal(node.left)
        print(node.value, end=" ")
        inorder_traversal(node.right)
# Example usage:
# Construct a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
# Perform in-order traversal  print("In-order Traversal:")
inorder_traversal(root)
```

This will output:

In-order Traversal:4 2 5 1 3

2. Using Recursion to Implement Pre Order Traversal

a. First traverse the root node.

b. Then visit the left node.

c. Then visit the right node.

In the above order specified when performing a pre order traversal.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def preorder_traversal(root):
    if root is not None:
        # Visit the root node
        print(root.value, end=" ")
        # Recur on the left subtree
        preorder_traversal(root.left)
        # Recur on the right subtree
        preorder_traversal(root.right)
# Example usage:
# Create a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
# Perform pre-order traversal preorder_traversal(root)
```

3. Using Recursion to Implement Post-Order Traversal

a. First visit the left node.

b. Then visit the right node.

c. Then traverse the current node.

In the above order specified when performing a post-order traversal.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def postorder_traversal(node):
    if node:
        postorder_traversal(node.left)
        postorder_traversal(node.right)
        print(node.value, end=' ')

# Example usage:
# Construct a sample binary tree
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
```

Binary Search Structure:

A particular kind of binary tree data structure called a Binary Search Tree (BST) is especially effective at finding, adding, and removing elements because it adheres to a predetermined set of rules. Each node in a BST can have a maximum of two offspring: a left and a right child. The values (or keys) of the nodes in the left subtree are less than the value of the current node, and the values in the right subtree are greater, which is the essential characteristic of a BST.

The following are the essential traits and guidelines of a Binary Search Tree:

1. Structure of Binary Trees: A BST is a binary tree, as the name implies, meaning that each node can have a maximum of two children: a left child,
2. Performing a Binary Search Tree Search:

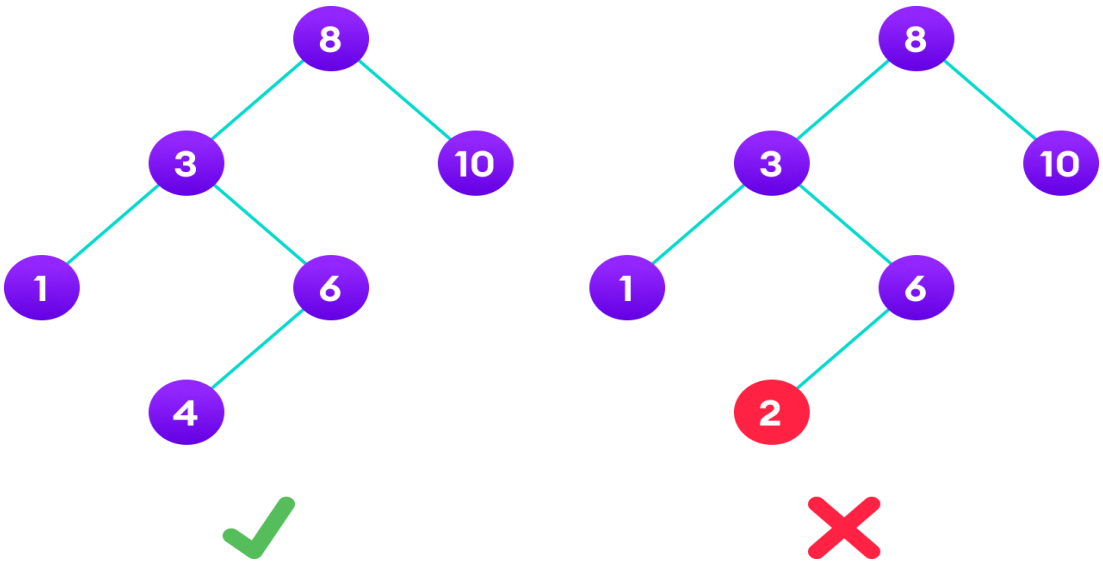
In order to find a particular value in a Binary Search Tree (BST), one must traverse the tree using its ordering property as a guide. Here is an explanation and a C algorithm for searching in a BST:

Performing a Binary-Search-Tree Search:

In order to find a particular value in a Binary Search Tree (BST), one must traverse the tree using its ordering property as a guide. This is an explanation and a C algorithm for searching in a BST.

1. The search function requires two inputs: the value we wish to search for (key) and a pointer to the BST root (root).
2. The function first determines whether the data of the root node matches the key or whether the tree is empty (root is NULL). The root node is returned, signifying that the key has been located, if either of these circumstances holds true.
3. Because the key needs to be on the left side according to the BST's ordering property, it recursively calls search on the left subtree if the key is smaller than the root's data.
4. Because the key must be on the right subtree, it recursively calls search on the right subtree if the key value is larger than the roots data.

In a Binary-Search-Tree (BST), adding a new node is as follows:



Finding the ideal location for a new node while preserving the ordering property of the Binary Search Tree (BST) is the process of inserting a new node. This is an example of a C algorithm for a BST insertion operation.

1. To create a new BST node with the given key, use the create Node function. The node's memory is allocated, its data is set to the specified key, and the left and right pointers are initialized to NULL.
2. The insert function requires two inputs: the value to be inserted (key) and a pointer to the BST root (root).
3. The insert function creates a new node with the key and returns it as the new root if the tree is initially empty (root is NULL).
4. The function compares the key with the data of the current node if the tree is not empty.
5. It recursively calls insert on the left subtree if the key is smaller than the data of the current node.
6. It recursively calls insert on the appropriate subtree if the key is greater than or equal to the data of the current node since the ordering property requires that the key be inserted in the appropriate subtree.
7. After inserting the key, the function returns the updated root.

Create an empty BST, use the insert function to add multiple values to it, and then run an in order traversal to copy the contents of the Tree in the main function.

In a Binary-Search-Tree (BST), deletion:

1. The delete function requires two inputs: the value to be deleted (key) and a pointer to the BST root (root).
2. Since there is nothing to delete, it returns NULL if the tree is initially empty (root is NULL).
3. It recursively calls delete on the left subtree if the key is less than the data of the current node.
4. It recursively calls delete on the appropriate subtree if the key exceeds the size of the data on the current node.
5. The node has been determined to be deleted if the key corresponds with the data currently on file.

Three cases are examined for deletion:

Node without children: To update the parent's pointer, we just release the node and return NULL.

When deleting a node with two children, it is necessary to locate the minimum value node in the appropriate subtree, which is done using the findMin function.

Create a sample BST in the main function, use the delete function to remove a particular value (key), and then print the tree before and after the deletion using an in-order traversal to show how the deletion operation was carried out.

The vertical or horizontal depth of a tree:

The longest downward path from a node to a leaf in a tree determines that node's height. The tree's height equals the height of its roots. Therefore, we must consider each link in the tree in-order to regulate its height.

AVL Tree:

The AVL tree is a type of binary search tree wherein the height difference between a node's left and right subtrees is either equal to or less than one. The term "AVL tree," or balanced binary tree, refers to the method of balancing the height of binary trees that was created by Adelson, Velsky, and Landis.

The factor of Balance in AVL Tree:

Height(left subtree) – height(right subtree) equals the balance factor.
Either -1, 0, or +1 should be the balanced factor. If not, the tree will be regarded as being out of balance.
The definition of an AVL tree is as follows: Let T be a non-empty binary tree, and its left and right subtrees are TL and TR.
If both TL and TR have the same height, then the tree is height-balanced.
• hL - hR<= 1, where TL and TR's heights are represented by hL - hR.
Depending on whether the height of a node's left subtree is greater than, less than, or equal to the height of its right subtree, the balance factor of that node in a binary tree can have a value of 1, -1, or 0.

AVL Tree Representation

AVL Rotations: In order to create a balanced tree, rotations are the mechanisms that move some of the unbalanced tree's subtrees. An AVL tree can rotate in any of the following four ways to balance itself:

1. Rotation to the left
2. Rotation to the right
3. Rotating left to right
4. Rotating right to left

The following two rotations are double rotations after the initial two single rotations.

Rotation to the left

We execute a single left rotation when a node is added to the right subtree of the right subtree if the tree becomes unbalanced.

In this case, a node has been inserted into the right subtree of node A, causing node A to become unbalanced. Making A the left subtree of B allows us to execute the left rotation.

Rotation to the right

AVL tree:The imbalanced node turns to the right and becomes the left child's right child, as seen.

Turning from the Left to the Right

A somewhat more complex variation of the rotations that have already been covered is called double rotations. To understand them better, we should watch each and every action taken during rotation. Let's practice turning from left to right first. A left-right rotation is produced by combining the rotations of the left and right.

The fundamental functions of AVL trees are as follows: 1. Locate a specific node; 2. Add a node; and 3. Remove a node.

Looking around

1. Begin at the root node in order to search for a node with a specific value x.
2. Examine this root node's value in relation to x.
3. Return a pointer to this node and declare the node found if the two are equal.
4. If the two do not equal, determine if x is greater or less than the root node's value.
5. Repeat the search for the correct subtree if x is greater than the value of the root node.
6. If the value of x is less than that of the root node, repeat the search for its Search for the left subtree of the root node again if x is less than the value of the root node.
7. Come to the conclusion that there are no nodes in the AVL tree with values equal to x if you come across a NULL at any stage of the search.
8. Return NULL

Insertion Algorithm:

To carry out the insertion operation in an AVL Tree, the following procedures must be followed.

Step 1: Establish a node

Step 2: Determine whether the tree is empty

Step 3: The newly created node will become the AVL Tree's root node if the tree is empty.

Step 4: In the event that the tree is not empty, we insert nodes using the Binary Search Tree and determine the node's balancing factor.

Step 5: If the balancing factor is greater than ±1, we rotate the aforementioned node appropriately and proceed with the insertion from Step 4.

Removal of a Node from an AVL Tree:

There are three distinct circumstances in which deletions occur in AVL Trees:
In case the node to be deleted is a leaf node, it can be deleted without any replacement since it doesn't affect the binary search tree property. This is scenario number one. Rotations are used to restore equilibrium because it can become upset.

• Scenario 2 (Deletion of a node with one child): If the node that is to be deleted has a single child, substitute the value from its child node for the node's value. Next, remove the child

In the third scenario, which involves deleting a node with two children, locate the node's in order successor and substitute its value with the in order successor value. Attempt to remove the in order successor node after that. Use balance algorithms if, after deletion, the balance factor is greater than 1.

An unbalanced AVL tree may result from a deletion operation. Rotations classified as L and R are required for rebalancing the tree following deletion. The R category is further divided into R0, R1, and R-1 rotations, whereas the L category is further divided into L0, L1, and L-1 rotations. When a node is deleted, the balance factors of both the parent and the parent's ancestor are altered. The parent's balancing factor could be zero.

• **Red -Black Trees**

An additional attribute added to each node in a red-black tree is its color, which can be either red or black. This makes the tree a binary search tree. It is also necessary to monitor each node's parent.

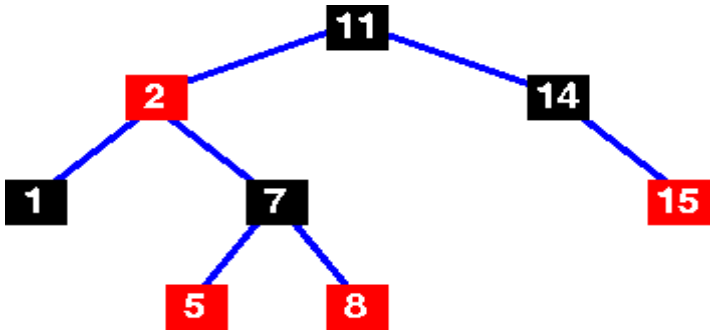
Red-black tree definition / Red-Black Tree Properties:

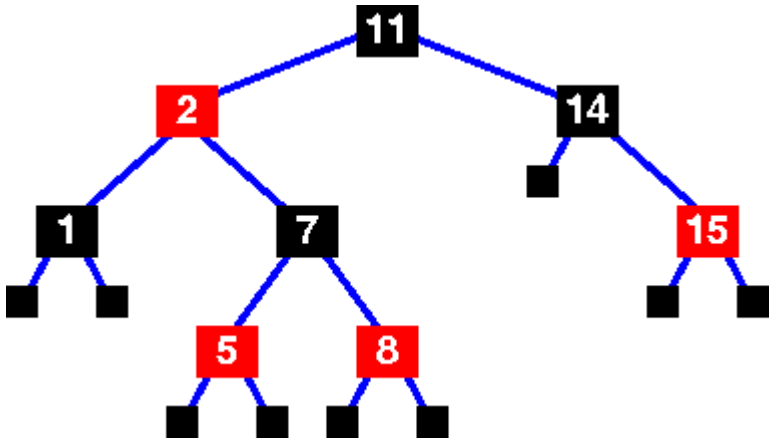
A binary search tree with the following red-black properties is called a red-black tree:

1. All nodes are either black or red.

2. There is a black root node.
3. There are no black leaves (NULL).

1. A node's two offspring are black if it is red. It means that red nodes cannot be next to each other on any path that leads from the root to a leaf. Nonetheless, a sequence of black nodes could contain any number of them. We refer to this as red condition.





Advantages:

1. The time complexity of Red-Black Trees for simple operations such as searching, insertion, and deletion is guaranteed to be O(log n).
2. Red Black Trees can balance themselves.

3. Red Black Trees' adaptability and effective performance make them suitable for a variety of uses.

4. Red Black Trees have a reasonably straightforward and understandable balance system.

Disadvantages:

1. One drawback of Red-Black Trees is that each node needs an additional bit of storage to store its color—red or black.

2. Implementation Complexity.

3. While Red Black Trees offer effective performance for fundamental tasks, they might not be the ideal option for particular data kinds or use cases.

Working with Red-Black Trees:

1. Type in a crucial value (insert).
2. Use a lookup or search to see if a key value is present in the tree.
3. Delete the key value from the tree.
4. Print every key value in a sorted list (print)
5. Change the color

In Red-Black Trees, Rotation / Restructure Operations:

A Node is Added to a Red Black Tree:

The process for inserting a key into a red-black tree is exactly the same as that of a binary search tree. On the other hand, giving the inserted node a color could go against the red-black tree's characteristics and result in imbalances. Classifications for the imbalances include LLb, LLr, RLb, RLr, LRb, LRr, RRb, and RRr. The type LLr, RLr, LRr, and RRr imbalances with "r" as their suffix only call for.

Removing a Node from a Red Black Tree:

The process for removing a key from a red-black tree is precisely the same as that for a binary search tree. There is no way that the black condition will be broken if the deleted node is red. In the event that the deleted node, v, is a black node, there is a chance that the black condition will be broken. This throws the red-black tree out of balance.

Depending on whether the deleted node (v) occurs to the right or left of its parent, the imbalance is categorized as Left (L) or Right (R). If sibling_v, v's black sibling, is present, the imbalance is further categorized as either Lb or Rb. Should v's sibling, sibling_v, be red, then

Splay Trees :

Research on node access has demonstrated that once a node or piece of information is accessed, it is likely to be accessed again. Binary-search-trees with a self adjusting mechanism are called Splay Trees. The Splay Tree data structure drastically alters its shape when a node is accessed, pushing that link in the direction of the base node.

This modification improves the efficiency of subsequent accesses to the node. As a result, the most recent node that is accessed—whether for insertion or search—is moved closer to the root each time. Methodically, splay rotations—which are essentially AVL tree rotations—are followed in order to pushing the node upwards toward the base.

The other nodes would be forced to move away from the root as a result.

This modification improves the efficiency of subsequent accesses to the node. As a result, the most recent node that is accessed—whether for insertion or search—is moved closer to the root each time. Methodically, splay rotations—which are essentially AVL tree rotations—are followed in order to pushing the node upwards toward the base.

The other nodes would be forced to move away from the root as a result.

Below are the different rotations that need to be made in the splay tree.

- Rotating rightward, or zigzag
- "Zig zig"—two rotations to the right
- Left rotation, or zigzag rotation
- [Two Left Rotations] Zagzag
- The zigzag [Zig came next to Zag]
- The zig zig [Zig came next to Zag]