Sashi Jagadam

jsashikala@gmail.com

Non-linear data structures

Binary Trees, Binary Tree Search, In-Order, Pre-Order, Post-Order Traversal, AVL Trees, Red-Black Trees, B-Tree, Graphs

Table of Contents

[Non-Linear Data Structures 3](#_Toc184905278)

[Binary Trees 3](#_Toc184905279)

[Why a Tree Data structure 3](#_Toc184905280)

[Tree Terminology 3](#_Toc184905281)

[Node 3](#_Toc184905282)

[Edge 3](#_Toc184905283)

[Root 4](#_Toc184905284)

[Height of a Node 4](#_Toc184905285)

[Depth of a Node 4](#_Toc184905286)

[Height of a Tree 4](#_Toc184905287)

[Degree of a Node 4](#_Toc184905288)

[Forest 5](#_Toc184905289)

[Code Explanation: 13](#_Toc184905290)

[Binary Tree Traversal 13](#_Toc184905291)

[1. Inorder Traversal (Left, Root, Right) 14](#_Toc184905292)

[2. Preorder Traversal (Root, Left, Right) 14](#_Toc184905293)

[3. Postorder Traversal (Left, Right, Root) 14](#_Toc184905294)

[4. Level Order Traversal (Breadth-First Traversal) 14](#_Toc184905295)

[Binary Search Tree 15](#_Toc184905296)

[Search Operation 15](#_Toc184905297)

[Full Binary Tree 21](#_Toc184905298)

[Complete Binary Tree 25](#_Toc184905299)

[Perfect Binary Tree 28](#_Toc184905300)

[Threaded Binary Tree 31](#_Toc184905301)

[One Way Threaded Binary Trees 32](#_Toc184905302)

[Two way Threaded Binary Trees 33](#_Toc184905303)

[Two way Threaded Binary Trees with header Node 34](#_Toc184905304)

[Representing a Threaded Binary Tree 36](#_Toc184905305)

[Single Threaded Binary Tree 36](#_Toc184905306)

[Reverse Morris Traversal 38](#_Toc184905307)

[Algorithm for Reverse Morris Traversal 38](#_Toc184905308)

[AVL Tree 42](#_Toc184905309)

[What is an AVL Tree? 42](#_Toc184905310)

[Implementation of AVL Tree in C++ 43](#_Toc184905311)

[AVL Tree Rotations 43](#_Toc184905312)

[Right Rotation (RR) 44](#_Toc184905313)

[Left Rotation (LL) 44](#_Toc184905314)

[Left-Right Rotation (LR) 45](#_Toc184905315)

[Right-Left Rotation (RL) 45](#_Toc184905316)

[Representation of AVL Tree in C++ 46](#_Toc184905317)

[Implementation of Insert Function 47](#_Toc184905318)

[Implementation of Delete Node Function 47](#_Toc184905319)

[Implementation of Search Function 48](#_Toc184905320)

[Implementation of Rotate Left Function 48](#_Toc184905321)

[Implementation of Rotate Right Function 48](#_Toc184905322)

[Red Black Tree 54](#_Toc184905323)

[How the red-black tree maintains the property of self-balancing? 55](#_Toc184905324)

[Operations on a Red-Black Tree 55](#_Toc184905325)

[Rotating the subtrees in a Red-Black Tree 55](#_Toc184905326)

[Left Rotate 56](#_Toc184905327)

[Right Rotation: 56](#_Toc184905328)

[Algorithm for Insertion 58](#_Toc184905329)

[Algorithm for Deletion 58](#_Toc184905330)

[Black Height Property 59](#_Toc184905331)

[B-Tree 68](#_Toc184905332)

[Properties of B-Tree: 69](#_Toc184905333)

[B-Tree Characteristics: 70](#_Toc184905334)

[B-Tree Properties: 70](#_Toc184905335)

[B-Tree Algorithm: 70](#_Toc184905336)

[1. B-tree Insertion Algorithm: 70](#_Toc184905337)

[2. B-tree Search Algorithm: 71](#_Toc184905338)

[3. B-tree Deletion Algorithm: 71](#_Toc184905339)

[Rules for root node and internal nodes 72](#_Toc184905340)

[Rules for Internal Nodes 72](#_Toc184905341)

[Rules for Root Node 72](#_Toc184905342)

[Rules for Leaf Nodes 73](#_Toc184905343)

[Summary of rules 73](#_Toc184905344)

[Rules for Children in a B-Tree 73](#_Toc184905345)

[Child Count per Node 73](#_Toc184905346)

# Non-Linear Data Structures

## Binary Trees

Binary trees are hierarchical data structures in which each node has at most two children, commonly referred to as the left child and right child. Below is a comprehensive explanation and implementation of a binary tree in C++.

Key Components of a Binary Tree

* Node Structure:
* Contains data.
* Has pointers to its left and right child.
* Tree Operations:
* Insertion: Add nodes to the tree.
* Traversal: Visit all nodes in a specific order (Inorder, Preorder, Postorder).
* Search: Find a node in the tree.
* Deletion: Remove a node.

### Why a Tree Data structure

* Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
* In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size which is not acceptable in today's computational world.
* Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

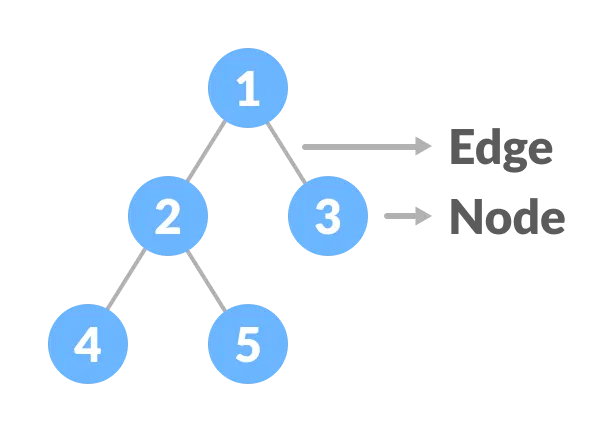
### Tree Terminology

#### Node

1. A node is an entity that contains a key or value and pointers to its child nodes.
2. The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.
3. The node having at least a child node is called an **internal node**.

#### Edge

1. It is the link between any two nodes.



#### Root

1. It is the topmost node of a tree.

#### Height of a Node

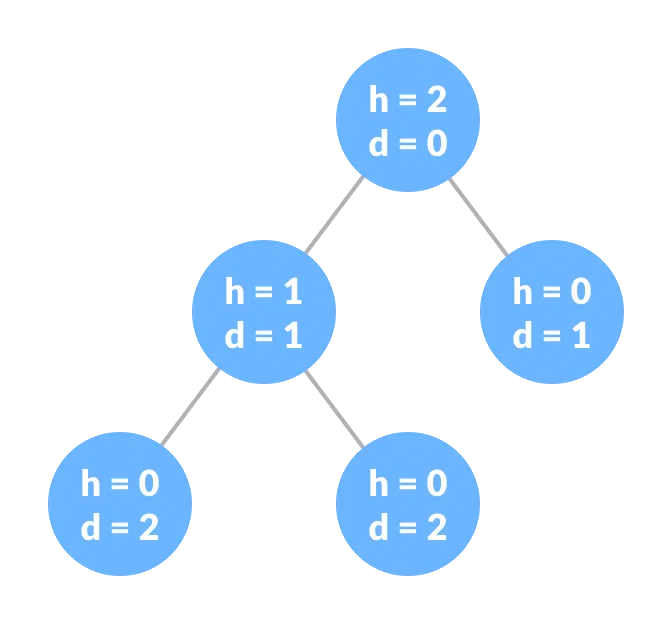
1. The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

#### Depth of a Node

1. The depth of a node is the number of edges from the root to the node.

#### Height of a Tree

1. The height of a Tree is the height of the root node or the depth of the deepest node.

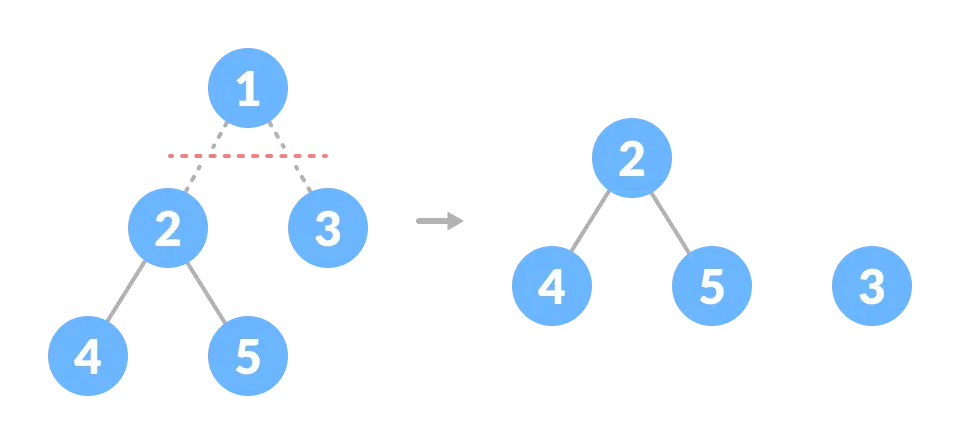


#### Degree of a Node

1. The degree of a node is the total number of branches of that node.

#### Forest

1. A collection of disjoint trees is called a forest.
2. You can create a forest by cutting the root of a tree.



Example: Binary Tree with insert, delete, delete deepest node, get deepest node, post-order, pre-order and in-order traversal

#include <iostream>

#include <queue>

using namespace std;

// Define a node structure for the binary tree

struct Node {

int value; // Data stored in the node

Node\* left; // Pointer to the left child

Node\* right; // Pointer to the right child

// Constructor to initialize a node

Node(int val) : value(val), left(nullptr), right(nullptr) {}

};

// Class for an Unsorted Binary Tree

class BinaryTree

{

private:

Node\* root; // Root of the binary tree

public:

BinaryTree()

{

root = nullptr;

}

// Helper function for in-order traversal

void inOrderTraversal(Node\* node) {

if (node == nullptr) return;

inOrderTraversal(node->left);

cout << node->value << " ";

inOrderTraversal(node->right);

}

// Helper function for pre-order traversal

void preOrderTraversal(Node\* node) {

if (node == nullptr) return;

cout << node->value << " ";

preOrderTraversal(node->left);

preOrderTraversal(node->right);

}

void postOrderTraversal(Node\* node)

{

if (node == nullptr) return;

postOrderTraversal(node->left);

postOrderTraversal(node->right);

cout << node->value << " ";

}

void deleteDeepest(Node\* deepNode)

{

if(root == nullptr)

return;

queue<Node\*> q;

q.push(root);

while(!q.empty())

{

Node\* current = q.front();

q.pop();

if(current == deepNode)

{

cout<<"Current val "<<current->value;

delete deepNode;

return;

}

if(current->left)

{

if(current->left == deepNode)

{

delete current->left;

current->left = nullptr;

return;

}

else

{

q.push(current->left);

}

}

if(current->right)

{

if(current->right == deepNode)

{

delete current->right;

current->right = nullptr;

return;

}

else

{

q.push(current->right);

}

}

}

}

// Helper function to get the deepest node

Node\* getDeepestNode() {

queue<Node\*> q;

q.push(root);

Node\* current;

while (!q.empty()) {

current = q.front();

q.pop();

if (current->left) q.push(current->left);

if (current->right) q.push(current->right);

}

return current;

}

void deleteNode(int value)

{

if(root == nullptr)

{

cout<<"Empty tree"<<endl;

return;

}

if(root->left == nullptr && root->right == nullptr)

{

if(root->value == value)

{

delete root;

root = nullptr;

return;

}

else

{

cout<<"Node with value "<<value<<" not found";

return;

}

}

queue<Node\*> q;

q.push(root);

Node\* keyNode = nullptr;

Node\* current = nullptr;

while(!q.empty())

{

current = q.front();

q.pop();

if(current->value == value)

{

keyNode = current;

break;

}

if(current->left)

q.push(current->left);

if(current->right)

q.push(current->right);

}

if(keyNode != nullptr)

{

Node\* deepNode = getDeepestNode();

cout<<"Key node value "<<keyNode->value<<" deep node value "<<deepNode->value<< endl;

keyNode->value = deepNode->value;

deleteDeepest(deepNode);

}

else

{

cout<<"Node with value "<<value<<" not found"<<endl;

}

}

// Helper function to display the tree structure

void displayTree(Node\* node)

{

if (node == nullptr)

{

cout<<"Empty tree."<<endl;

return;

}

queue<Node\*> q;

q.push(node);

while(!q.empty())

{

int lvlSize = q.size();

for(int i=0; i< lvlSize; ++i)

{

Node\* current = q.front();

q.pop();

if(current)

{

cout<<current->value<<" ";

q.push(current->left);

q.push(current->right);

}

else

{

cout<<"Null";

}

}

}

cout<<endl;

}

// Insert a node into the tree

void insert(int value)

{

Node\* newNode = new Node(value);

if (root == nullptr) {

root = newNode;

cout << "Inserted " << value << " as root"<<endl;

return;

}

queue<Node\*> q;

q.push(root);

while (!q.empty()) {

Node\* current = q.front();

q.pop();

if (current->left == nullptr) {

current->left = newNode;

cout << "Inserted " << value << " as left child of " << current->value << endl;

return;

} else {

q.push(current->left);

}

if (current->right == nullptr) {

current->right = newNode;

cout << "Inserted " << value << " as right child of " << current->value << endl;

return;

} else {

q.push(current->right);

}

}

}

// Perform in-order traversal

void inOrder() {

cout << "In-order Traversal: ";

inOrderTraversal(root);

cout << endl;

}

// Perform pre-order traversal

void preOrder() {

cout << "Pre-order Traversal: ";

preOrderTraversal(root);

cout << endl;

}

// Perform post-order traversal

void postOrder() {

cout << "Post-order Traversal: ";

postOrderTraversal(root);

cout << endl;

}

// Display the tree structure

void display() {

cout << "Tree Structure:\n";

displayTree(root);

cout << endl;

}

};

int main() {

BinaryTree tree;

// Insert nodes into the tree

tree.insert(10);

tree.insert(20);

tree.insert(30);

tree.insert(50);

tree.insert(60);

tree.insert(40);

tree.insert(70);

// Display the tree structure

tree.display();

// Perform traversals

tree.inOrder();

tree.preOrder();

tree.postOrder();

// Delete a node

cout << "Deleting node 20...\n";

tree.deleteNode(20);

tree.deleteNode(40);

tree.deleteNode(50);

// Display the tree structure after deletion

tree.display();

// Perform traversals after deletion

tree.inOrder();

tree.preOrder();

tree.postOrder();

return 0;

}

### Code Explanation:

* deleteDeepest function
  + deletes the deepest node in the tree by performing a level-order traversal and freeing the memory of the node
  + updates the parent node’s pointer (left or right) to nullptr
* deleteNode function
  + finds the node with the specific value
  + replaces the value of the target node with the value of the deepest node
  + deletes the deepest node using the deleteDeepest function
  + the getDeepestNode function is called to find the deepest node
  + its value replaces the value of the node to be deleted
* level-order traversal
  + the above functions use a queue for level-order traversal to locate the node to delete and the deepest node
* getDeepestNode function
  + traverses the tree level by level by using a queue
  + returns the last node visited, which is the deepest node

### Binary Tree Traversal

Binary tree traversal is the process of visiting all the nodes of a binary tree in a systematic order. Traversal methods ensure that each node is visited once, and they differ in the sequence of visiting the root, left subtree, and right subtree.

Types of Binary Tree Traversal

#### 1. Inorder Traversal (Left, Root, Right)

**Steps**:

* Visit the left subtree recursively.
* Visit the root node.
* Visit the right subtree recursively.

**Result**: If the tree is a Binary Search Tree (BST), the nodes are visited in ascending order.

**Example**:

4

/ \

2 6

/ \ / \

1 3 5 7

Inorder traversal gives: 1, 2, 3, 4, 5, 6, 7.

#### 2. Preorder Traversal (Root, Left, Right)

**Steps**:

* Visit the root node.
* Visit the left subtree recursively.
* Visit the right subtree recursively.

**Result**: Captures the structure of the tree and is used for tasks like copying the tree.

**Example**:

For the same tree:

Preorder traversal gives: 4, 2, 1, 3, 6, 5, 7.

#### 3. Postorder Traversal (Left, Right, Root)

**Steps**:

* Visit the left subtree recursively.
* Visit the right subtree recursively.
* Visit the root node.

**Result**: Used in applications like deleting a tree (process children before parent).

**Example**:

For the same tree:

Postorder traversal gives: 1, 3, 2, 5, 7, 6, 4.

#### 4. Level Order Traversal (Breadth-First Traversal)

**Steps**:

* Visit the root node.
* Visit all nodes at depth 1 (children of the root).
* Visit all nodes at depth 2, and so on.

**Implementation**: Typically uses a queue to manage nodes at each level.

**Example**:

For the same tree:

Level order traversal gives: 4, 2, 6, 1, 3, 5, 7.

## Binary Search Tree

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

* It is called a binary tree because each tree node has a maximum of two children.
* It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

* All nodes of left subtree are less than the root node
* All nodes of right subtree are more than the root node
* Both subtrees of each node are also BSTs i.e. they have the above two properties

### Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

#include <iostream>

#include <queue> //For level-order traversal

using namespace std;

// Node structure

class Node

{

public:

int value; // Value of the node

Node\* left; // Pointer to left child

Node\* right; // Pointer to right child

Node(int val) : value(val), left(nullptr), right(nullptr) {}

};

// Binary Search Tree

class BST {

private:

Node\* root; // Root of the tree

//Insert node

Node\* insert(Node\* node, int value)

{

if (node == nullptr)

return new Node(value);

if (value < node->value)

node->left = insert(node->left, value);

else if (value > node->value)

node->right = insert(node->right, value);

return node;

}

//find the minimum value node

Node\* findMin(Node\* node)

{

while (node && node->left != nullptr)

{

node = node->left;

}

return node;

}

//Delete node

Node\* deleteNode(Node\* node, int value)

{

if (node == nullptr)

return nullptr;

if (value < node->value)

{

node->left = deleteNode(node->left, value);

}

else if (value > node->value)

{

node->right = deleteNode(node->right, value);

}

else

{

// Node with one child or no child

if (node->left == nullptr)

{

Node\* temp = node->right;

delete node;

return temp;

}

else if (node->right == nullptr)

{

Node\* temp = node->left;

delete node;

return temp;

}

// Node with two children: Get the inorder successor (minimum in the right subtree)

Node\* temp = findMin(node->right);

// Replace current node's value with the successor's value

node->value = temp->value;

// Delete the inorder successor

node->right = deleteNode(node->right, temp->value);

}

return node;

}

//search node

bool search(Node\* node, int value)

{

if (node == nullptr)

return false;

if (node->value == value)

return true;

if (value < node->value)

return search(node->left, value);

else

return search(node->right, value);

}

//Traversal functions: Inorder, preOrder, postOrder, levelOrder

void inOrder(Node\* node)

{

if (node == nullptr) return;

inOrder(node->left);

cout << node->value << " ";

inOrder(node->right);

}

void preOrder(Node\* node)

{

if (node == nullptr)

return;

cout << node->value << " ";

preOrder(node->left);

preOrder(node->right);

}

void postOrder(Node\* node)

{

if (node == nullptr)

return;

postOrder(node->left);

postOrder(node->right);

cout << node->value << " ";

}

void levelOrder(Node\* node)

{

if (node == nullptr)

return;

queue<Node\*> q;

q.push(node);

while (!q.empty())

{

Node\* current = q.front();

q.pop();

cout << current->value << " ";

if (current->left != nullptr)

q.push(current->left);

if (current->right != nullptr)

q.push(current->right);

}

}

public:

// Constructor

BST() : root(nullptr) {}

// Insert a node

void insert(int value)

{

root = insert(root, value);

}

// Delete a node

void deleteNode(int value)

{

root = deleteNode(root, value);

}

// Search for a value

bool search(int value)

{

return search(root, value);

}

// Traversals

void inOrder()

{

cout << "In-order Traversal: ";

inOrder(root);

cout << endl;

}

void preOrder()

{

cout << "Pre-order Traversal: ";

preOrder(root);

cout << endl;

}

void postOrder()

{

cout << "Post-order Traversal: ";

postOrder(root);

cout << endl;

}

void display()

{

cout << "Level-order Traversal: ";

levelOrder(root);

cout << endl;

}

};

int main()

{

BST bst;

// Insert nodes

bst.insert(50);

bst.insert(30);

bst.insert(70);

bst.insert(20);

bst.insert(40);

bst.insert(60);

bst.insert(80);

// Display traversals

bst.inOrder();

bst.preOrder();

bst.postOrder();

bst.display();

// Search for values

cout << "Search 40: " << (bst.search(40) ? "Found" : "Not Found") << endl;

cout << "Search 25: " << (bst.search(25) ? "Found" : "Not Found") << endl;

// Delete a node

bst.deleteNode(50);

bst.display();

bst.deleteNode(70);

// Display traversals after deletion

//bst.inOrder();

//bst.preOrder();

//bst.postOrder();

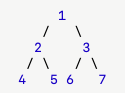
bst.display();

return 0;

}

## Full Binary Tree

A **Full Binary Tree** is a type of binary tree in which every node has either 0 or 2 children. This means no node in the tree has only one child.



#include <iostream>

using namespace std;

// Define a node structure for the binary tree

struct TreeNode {

int value; // Data stored in the node

TreeNode\* left; // Pointer to the left child

TreeNode\* right; // Pointer to the right child

// Constructor to initialize a node

TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}

};

// Class for a Full Binary Tree

class FullBinaryTree {

private:

TreeNode\* root; // Root of the binary tree

// Helper function for in-order traversal

void inOrderTraversal(TreeNode\* node) {

if (node == nullptr) return;

inOrderTraversal(node->left); // Visit left subtree

cout << node->value << " "; // Visit current node

inOrderTraversal(node->right); // Visit right subtree

}

// Helper function to check if the tree is a full binary tree

bool isFullTree(TreeNode\* node) {

if (node == nullptr) return true; // An empty tree is a full binary tree

if ((node->left == nullptr && node->right != nullptr) ||

(node->left != nullptr && node->right == nullptr)) {

return false; // Node has only one child, not a full binary tree

}

return isFullTree(node->left) && isFullTree(node->right);

}

public:

// Constructor to initialize the tree

FullBinaryTree() : root(nullptr) {}

// Function to set the root of the tree

void setRoot(int value) {

if (root == nullptr) {

root = new TreeNode(value);

cout << "Root set with value " << value << ".\n";

} else {

cout << "Root is already set.\n";

}

}

// Function to manually add children to a node

void addChildren(int parentValue, int leftValue, int rightValue) {

TreeNode\* current = findNode(root, parentValue);

if (current == nullptr) {

cout << "Parent node with value " << parentValue << " not found.\n";

return;

}

if (current->left == nullptr && current->right == nullptr) {

current->left = new TreeNode(leftValue);

current->right = new TreeNode(rightValue);

cout << "Added children " << leftValue << " and " << rightValue << " to node " << parentValue << ".\n";

} else {

cout << "Node " << parentValue << " already has children.\n";

}

}

// Helper function to find a node by value

TreeNode\* findNode(TreeNode\* node, int value) {

if (node == nullptr) return nullptr;

if (node->value == value) return node;

TreeNode\* leftSearch = findNode(node->left, value);

if (leftSearch != nullptr) return leftSearch;

return findNode(node->right, value);

}

// Perform in-order traversal

void inOrder() {

cout << "In-order Traversal: ";

inOrderTraversal(root);

cout << endl;

}

// Check if the tree is a full binary tree

void checkIfFullBinaryTree() {

if (isFullTree(root)) {

cout << "The tree is a full binary tree.\n";

} else {

cout << "The tree is not a full binary tree.\n";

}

}

};

int main() {

FullBinaryTree tree;

// Set the root of the tree

tree.setRoot(1);

// Add children to the root

tree.addChildren(1, 2, 3);

// Add children to the left child of the root

tree.addChildren(2, 4, 5);

// Add children to the right child of the root

tree.addChildren(3, 6, 7);

// Perform in-order traversal

tree.inOrder();

// Check if the tree is a full binary tree

tree.checkIfFullBinaryTree();

return 0;

}

Code Explanation:

**TreeNode Structure**:

* Each node contains a value and pointers to its left and right children.
* A constructor initializes the node.

**FullBinaryTree Class**:

* Provides functionality to set the root, add children, and check if the tree is a full binary tree.

**Key Methods**:

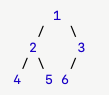
* setRoot(int value): Sets the root node of the tree.
* addChildren(int parentValue, int leftValue, int rightValue): Adds left and right children to a specified parent node.
* inOrderTraversal: Recursively visits the nodes in in-order (left, root, right).
* isFullTree(TreeNode\* node): Checks if the tree satisfies the full binary tree property.

**Main Function**:

* Demonstrates creating a tree, adding children, and checking if it is a full binary tree.

## Complete Binary Tree

A **Complete Binary Tree** is a type of binary tree in which all levels except possibly the last are fully filled, and all nodes are as far left as possible.



#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

class CompleteBinaryTree {

private:

vector<int> tree; // Vector to store the tree nodes in array representation

// Helper function for in-order traversal

void inOrderTraversal(int index) {

if (index >= tree.size() || tree[index] == -1) return;

inOrderTraversal(2 \* index + 1); // Left child

cout << tree[index] << " "; // Current node

inOrderTraversal(2 \* index + 2); // Right child

}

// Helper function for level-order traversal

void levelOrderTraversal() {

for (int i = 0; i < tree.size(); ++i) {

if (tree[i] != -1) {

cout << tree[i] << " ";

}

}

cout << endl;

}

public:

// Constructor to initialize the tree

CompleteBinaryTree() {}

// Insert a node into the complete binary tree

void insert(int value) {

tree.push\_back(value); // Add the value to the end of the array

cout << "Inserted " << value << " into the tree.\n";

}

// Perform in-order traversal

void inOrder() {

cout << "In-order Traversal: ";

inOrderTraversal(0); // Start from the root at index 0

cout << endl;

}

// Perform level-order traversal

void levelOrder() {

cout << "Level-order Traversal: ";

levelOrderTraversal();

cout << endl;

}

// Check if the tree is a complete binary tree

bool isCompleteTree() {

for (int i = 0; i < tree.size(); ++i) {

if (tree[i] == -1) return false; // A "gap" would indicate the tree is not complete

}

return true;

}

};

int main() {

CompleteBinaryTree tree;

// Insert nodes into the tree

tree.insert(1);

tree.insert(2);

tree.insert(3);

tree.insert(4);

tree.insert(5);

tree.insert(6);

// Perform traversals

tree.levelOrder(); // Output: 1 2 3 4 5 6

tree.inOrder(); // Output: 4 2 5 1 6 3

// Check if the tree is complete

if (tree.isCompleteTree()) {

cout << "The tree is a complete binary tree.\n";

} else {

cout << "The tree is not a complete binary tree.\n";

}

return 0;

}

**Explanation:**

**Tree Representation**:

* The tree is stored as an array/vector, where:
* The root is at index 0.
* The left child of a node at index i is at 2\*i + 1.
* The right child of a node at index i is at 2\*i + 2.

**Insertion**:

* Nodes are inserted into the vector sequentially to maintain the “complete” property.

**Traversals**:

* **In-order**: Recursively visits the left child, current node, and then the right child.
* **Level-order**: Simply prints the nodes in the vector.

**Check for Completeness**:

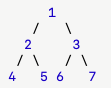
* Ensures there are no “gaps” (e.g., -1 or empty spaces) in the array representation.

**Main Function**:

* Inserts nodes, performs traversals, and checks if the tree is a complete binary tree.

## Perfect Binary Tree

A **Perfect Binary Tree** is a type of binary tree in which all internal nodes have exactly two children, and all leaf nodes are at the same level



#include <iostream>

#include <cmath>

using namespace std;

// Define a node structure for the binary tree

struct TreeNode {

int value; // Data stored in the node

TreeNode\* left; // Pointer to the left child

TreeNode\* right; // Pointer to the right child

// Constructor to initialize a node

TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}

};

// Class for a Perfect Binary Tree

class PerfectBinaryTree {

private:

TreeNode\* root; // Root of the binary tree

int height; // Height of the tree

// Helper function to build a perfect binary tree

TreeNode\* buildTree(int currentHeight) {

if (currentHeight > height) return nullptr; // Base case: beyond tree height

TreeNode\* node = new TreeNode(0); // Create a node with default value 0

node->left = buildTree(currentHeight + 1); // Build left subtree

node->right = buildTree(currentHeight + 1); // Build right subtree

return node;

}

// Helper function for in-order traversal

void inOrderTraversal(TreeNode\* node) {

if (node == nullptr) return;

inOrderTraversal(node->left); // Visit left subtree

cout << node->value << " "; // Visit current node

inOrderTraversal(node->right); // Visit right subtree

}

// Helper function to check if the tree is perfect

bool isPerfect(TreeNode\* node, int currentDepth, int targetDepth) {

if (node == nullptr) return true;

if (node->left == nullptr && node->right == nullptr) {

return currentDepth == targetDepth; // Check if leaf is at the correct depth

}

if (node->left == nullptr || node->right == nullptr) {

return false; // Internal node must have both children

}

return isPerfect(node->left, currentDepth + 1, targetDepth) &&

isPerfect(node->right, currentDepth + 1, targetDepth);

}

public:

// Constructor to initialize the tree with a given height

PerfectBinaryTree(int h) : root(nullptr), height(h) {

root = buildTree(1); // Build the tree starting from height 1

}

// Perform in-order traversal

void inOrder() {

cout << "In-order Traversal: ";

inOrderTraversal(root);

cout << endl;

}

// Check if the tree is perfect

void checkIfPerfect() {

int targetDepth = height;

if (isPerfect(root, 1, targetDepth)) {

cout << "The tree is a perfect binary tree.\n";

} else {

cout << "The tree is not a perfect binary tree.\n";

}

}

// Set values for all nodes in the tree (for demonstration)

void setValues(int startValue) {

int currentValue = startValue;

// Helper function to set values in level-order

function<void(TreeNode\*)> setValuesHelper = [&](TreeNode\* node) {

if (node == nullptr) return;

node->value = currentValue++;

setValuesHelper(node->left);

setValuesHelper(node->right);

};

setValuesHelper(root);

}

};

int main() {

int height = 3; // Example: Height of the perfect binary tree

PerfectBinaryTree tree(height);

// Set values for nodes

tree.setValues(1);

// Perform in-order traversal

tree.inOrder(); // Output: Nodes in in-order traversal

// Check if the tree is perfect

tree.checkIfPerfect();

return 0;

}

## Threaded Binary Tree

**Defining Threaded Binary Trees**

* In a binary search tree, there are many nodes that have an empty left child or empty right child or both.
* You can utilize these fields in such a way so that the empty left child of a node points to its inorder predecessor and empty right child of the node points to its inorder successor.

**Threaded binary Tree**

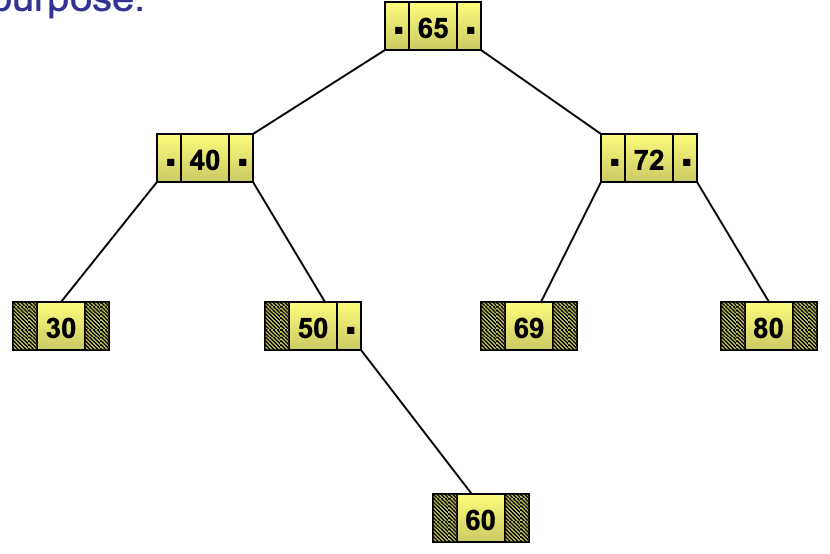
* One way threading:- A thread will appear in a right field of a node and will point to the next node in the inorder traversal.
* Two way threading:- A thread will also appear in the left field of a node and will point to the preceding node in the inorder traversal.

**Defining Threaded Binary Trees**

Consider the following binary search tree.

Most of the nodes in this tree hold a NULL value in their left or right child fields.

In this case, it would be good if these NULL fields are utilized for some other useful purpose.

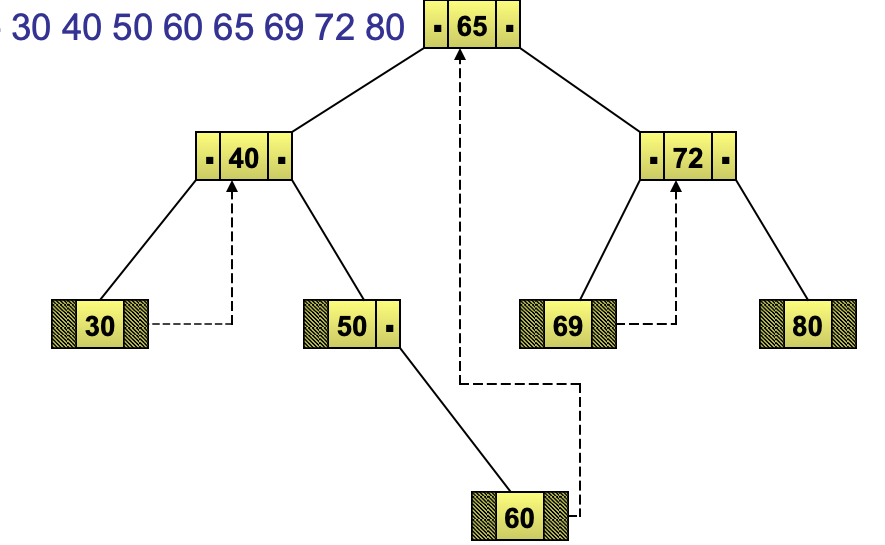


### One Way Threaded Binary Trees

The empty left child field of a node can be used to point to its inorder predecessor.

Similarly, the empty right child field of a node can be used to point to its in-order successor.  
Such a type of binary tree is known as a one way threaded binary tree.

A field that holds the address of its in-order successor is known as thread. In-order :- 30 40 50 60 65 69 72 80



### Two way Threaded Binary Trees

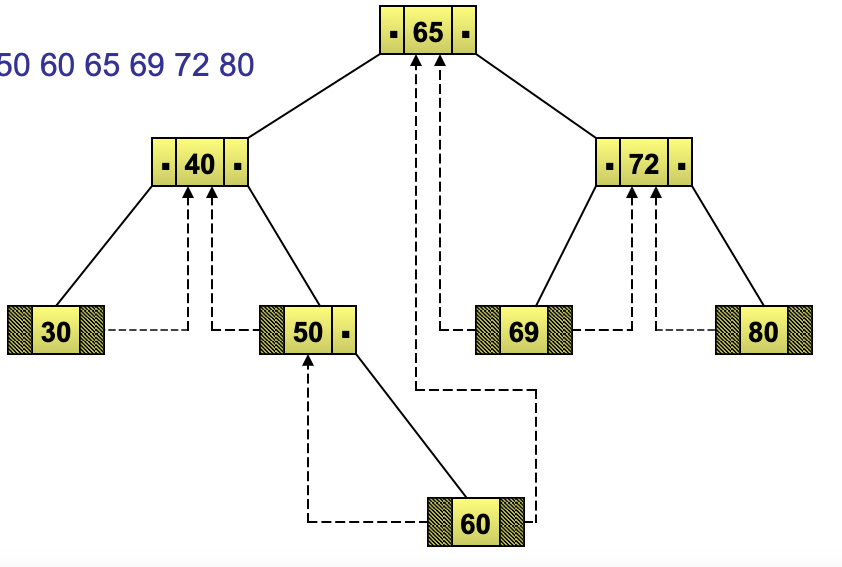
Such a type of binary tree is known as a threaded binary tree.

A field that holds the address of its inorder successor or predecessor is known as thread.

The empty left child field of a node can be used to point to its inorder predecessor.

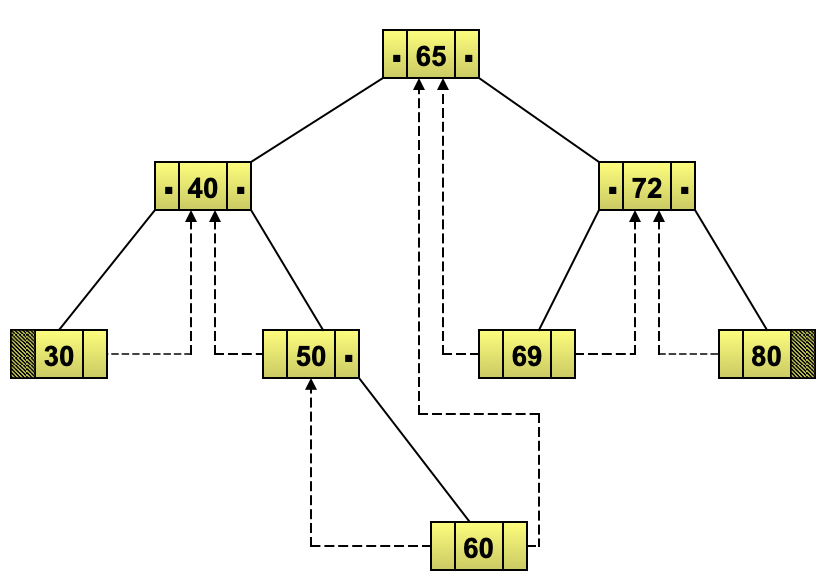
Similarly, the empty right child field of a node can be used to point to its inorder successor.

Inorder :- 30 40 50 60 65 69 72 80



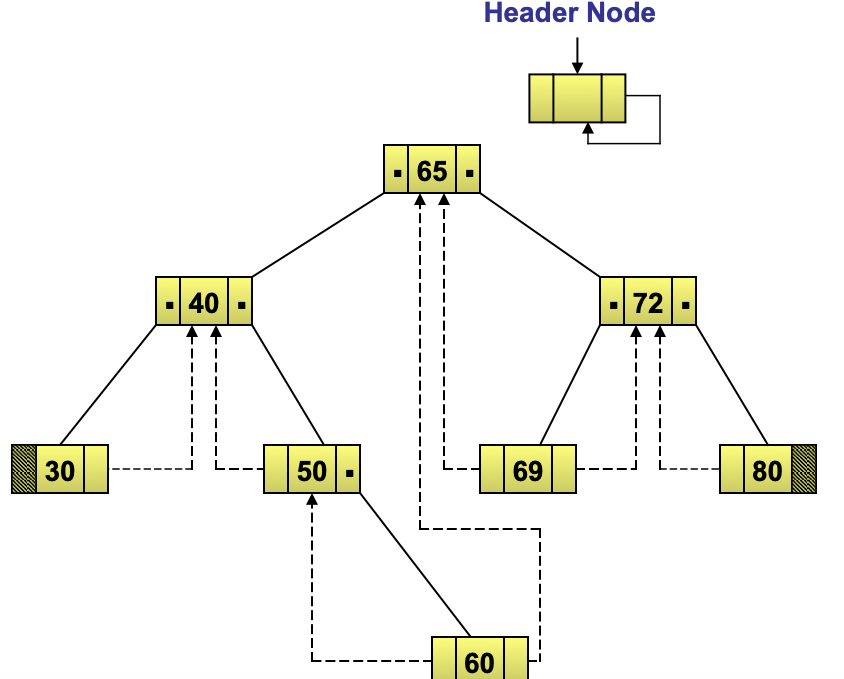
Node 30 does not have an inorder predecessor because it is the first node to be traversed in inorder sequence.

Similarly, node 80 does not have an inorder successor.

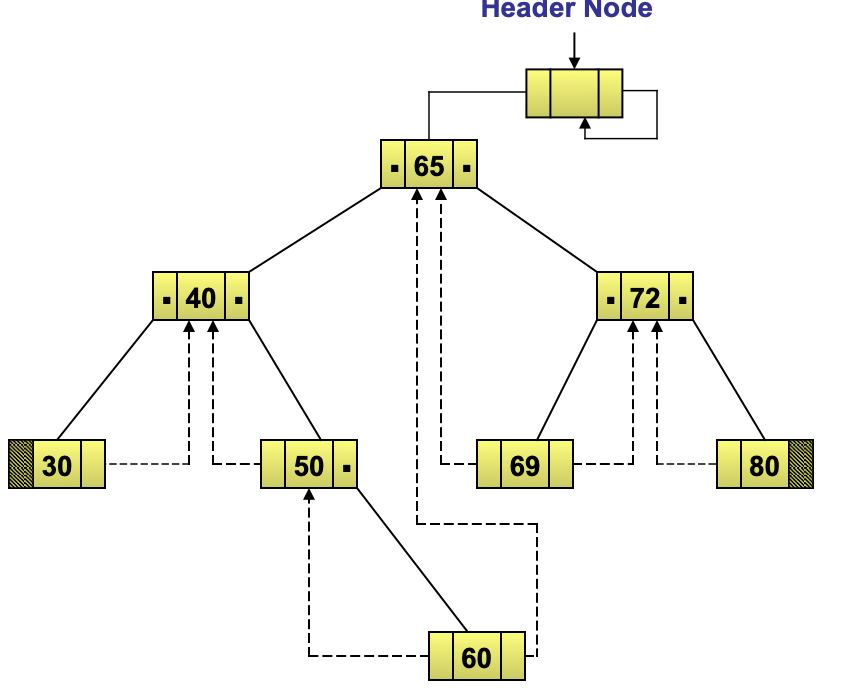


### Two way Threaded Binary Trees with header Node

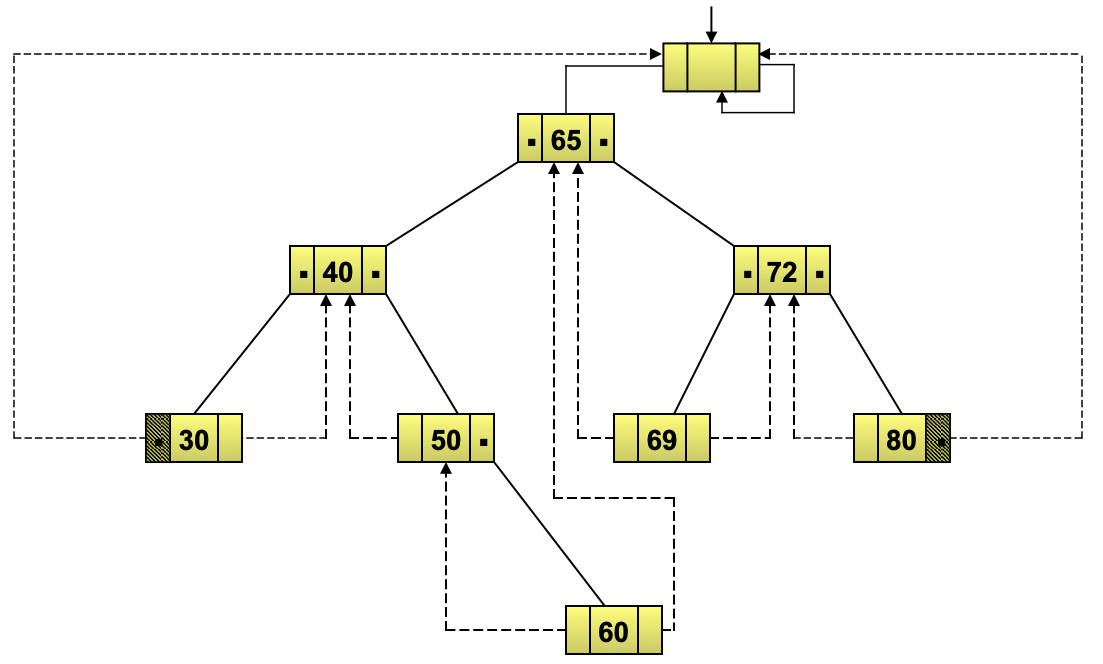
The right child of the header node always points to itself. Therefore, you take a dummy node called the header node.



The threaded binary tree is represented as the left child of the header node.



The left thread of node 30 and the right thread of node 80 point to the header node.



### Representing a Threaded Binary Tree

The structure of a node in a threaded binary tree is a bit different from that of a normal binary tree.

Unlike a normal binary tree, each node of a threaded binary tree contains two extra pieces of information, namely left thread and right thread.

The left and right thread fields of a node can have two values:

1: Indicates a normal link to the child node

0: Indicates a thread pointing to the inorder predecessor or inorder successor

Left Address of Data Address of Right Thread Left Child Right Child Thread

## Single Threaded Binary Tree

The **principle behind a single-threaded binary tree** is to efficiently utilize null pointers in binary tree nodes to store **threads**. These threads provide a way to directly access the **inorder predecessor** or **inorder successor** of a node, facilitating easy traversal of the tree without the need for recursion or a stack.

**Key Principles**

1. **Thread Concept**:

* In a standard binary tree, many nodes have null pointers for their left or right children.
* A **threaded binary tree** converts these null pointers into links (threads) to other nodes in the tree, typically the **inorder predecessor** or **inorder successor**.

1. **Traversal Without Recursion/Stack**:

* Standard traversal methods like **inorder traversal** rely on recursion or a stack to backtrack after visiting a node.
* By introducing threads, a node can directly “point” to the next node in the traversal sequence, eliminating the need for auxiliary memory structures.

**Single Threading**:

* A **single-threaded binary tree** adds threads for **one type of null pointer** (either left or right).
* **Right-threaded binary tree**: Threads point to the **inorder successor**.
* **Left-threaded binary tree**: Threads point to the **inorder predecessor**.

**Efficient Space Usage**:

* By reusing null pointers for threads, the tree reduces wasted memory while maintaining fast traversal capabilities.

**Working of a Single Threaded Binary Tree**

1. **Inorder Successor Threads**:

* If a node has no right child, its right pointer is repurposed to point to its **inorder successor**.

20

/ \

10 30

For node 10 (left child of 20), the right pointer will thread to 20 (inorder successor).

1. **Traversal**:

* Start at the leftmost node.
* Print the node’s value.
* Move to the next node in inorder sequence:
* Follow the thread if it exists.
* Otherwise, move to the leftmost node in the right subtree.

**Advantages**

**Space Efficiency**:

* Utilizes existing pointers in nodes, avoiding the need for an explicit stack or recursion overhead.

**Simplified Traversal**:

* Traversals (like inorder) become simpler and faster since backtracking is handled via threads.

**No Additional Structures**:

* Unlike recursion or iterative methods, threaded trees don’t require a stack or other storage for backtracking.

**Limitations**

**Insertion and Deletion Complexity**:

* Maintaining threads during insertion or deletion is more complex compared to a standard binary tree.

**Restricted to Traversal**:

* Threads are primarily useful for traversal and provide limited benefits for other tree operations.

**Applications**

**Inorder Traversal**:

* Quickly and efficiently traverse the tree in **O(n)** time without recursion or stack.

**Low-Memory Environments**:

* Useful in memory-constrained systems where additional storage for recursion or stacks is undesirable.

**Tree-based Data Structures**:

* Can be utilized in applications like expression trees or Huffman encoding trees to simplify traversal logic.

Comparison with other trees

|  |  |  |
| --- | --- | --- |
| **Feature** | **Standard Binary Tree** | **Single Threaded Binary Tree** |
| Traversal Mechanism | Uses recursion/stack | Uses threads for traversal |
| Memory Usage | May leave null pointers | Utilizes null pointers for threads |
| Traversal Time Complexity | O(n) | O(n) |
| Implementation Overhead | Simple | Slightly complex due to threads |

The single-threaded binary tree is an elegant way to achieve efficient traversal by leveraging unused pointers in the tree structure.

## Reverse Morris Traversal

**Reverse Morris Traversal** is an efficient way to perform a reverse inorder traversal (right-root-left) of a binary tree using the threading technique, which modifies the tree during the traversal but restores it afterward. This approach avoids recursion or the use of an auxiliary stack, achieving O(1) additional space complexity.

### Algorithm for Reverse Morris Traversal

The algorithm performs the following steps:

* Start at the root of the tree and traverse to the **rightmost node** (instead of the leftmost node in normal Morris traversal).
* While traversing:
* If the node’s right child is nullptr:
  + Process the current node.
  + Move to its left child.
* If the node’s right child is not nullptr:
  + Find the **inorder predecessor** (leftmost node in the left subtree or thread).
* If the predecessor’s right pointer is nullptr:
  + Set it to the current node (create a thread).
  + Move to the left child.
* If the predecessor’s right pointer points to the current node:
  + Remove the thread.
  + Process the current node.
  + Move to the left child.

This mirrors Morris Traversal but starts with the **right subtree** and traverses **backward**.

// Online C++ compiler to run C++ program online

#include <iostream>

using namespace std;

// Node structure for the binary tree

class Node

{

public:

int value; // Data value

Node\* left; // Pointer to the left child

Node\* right; // Pointer to the right child

// Constructor to initialize a node

Node(int val) : value(val), left(nullptr), right(nullptr) {}

};

// Class for the Binary Tree

class BinaryTree {

private:

Node\* root; // Root of the tree

// Helper function to insert a node into the tree

Node\* insertNode(Node\* current, int value)

{

if (current == nullptr)

{

return new Node(value);

}

if (value < current->value)

{

current->left = insertNode(current->left, value);

}

else if (value > current->value)

{

current->right = insertNode(current->right, value);

}

return current;

}

void inorderDisplay(Node\* current)

{

if (current == nullptr)

{

return;

}

inorderDisplay(current->left); // Visit left subtree

cout << current->value << " "; // Process the current node

inorderDisplay(current->right); // Visit right subtree

}

public:

// Constructor to initialize the BinaryTree

BinaryTree() : root(nullptr) {}

// Public function to insert a value

void insert(int value) {

root = insertNode(root, value);

}

// Reverse Morris Traversal for reverse inorder (right-root-left)

void reverseMorrisTraversal()

{

Node\* current = root;

cout << "Reverse Inorder Traversal: ";

while (current != nullptr)

{

if (current->right == nullptr)

{

// No right child, process the current node

cout << current->value << " ";

current = current->left; // Move to the left child

}

else

{

// Find the inorder predecessor in the left subtree of the right child

Node\* predecessor = current->right;

while (predecessor->left != nullptr && predecessor->left != current)

{

predecessor = predecessor->left;

}

if (predecessor->left == nullptr)

{

// Create a temporary thread to the current node

predecessor->left = current;

current = current->right; // Move to the right child

}

else

{

// Thread exists, remove it and process the current node

predecessor->left = nullptr;

cout << current->value << " ";

current = current->left; // Move to the left child

}

}

}

cout << endl;

}

void display()

{

cout << "Inorder Traversal: ";

inorderDisplay(root);

cout << endl;

}

};

int main()

{

// Create a BinaryTree object

BinaryTree tree;

// Insert nodes into the binary tree

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

tree.display();

// Perform Reverse Morris Traversal

tree.reverseMorrisTraversal();

return 0;

}

## AVL Tree

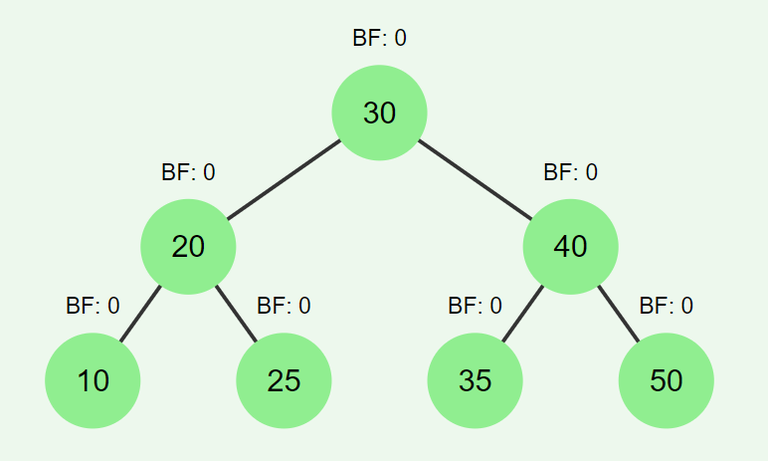
* AVL Tree, named after its inventors Adelson-Velsky and Landis, is a self-balancing binary search tree.
* In an AVL tree, the heights of the two child subtrees of any node differ by at most one, which ensures that the tree remains approximately balanced, providing efficient search, insertion, and deletion operations.

### What is an AVL Tree?

* An AVL tree maintains balance by performing rotations during insertions and deletions to ensure the height difference between the left and right subtrees of any node is no more than one.
* This property guarantees that the tree's height remains O(logn), ensuring efficient operations.
* An AVL Tree is a binary search tree with the following properties:
  + The heights of the left and right subtrees of every node differ by at most one.
  + Every subtree is an AVL tree.
  + For every node, its balance factor (height of left subtree - height of right subtree) is -1, 0, or 1.

### Implementation of AVL Tree in C++

* An AVL tree can be implemented using a binary tree structure where each node will have left and right pointers and key values to store the data but along with that, we will store the height for each node so that the balance factor can be calculated easily.
* The balance factor of a node will be calculated for each node as the difference between the heights of its left and right subtrees.
* When the balance factor for any node is not in the allowed limits, rotations are preformed to balance it

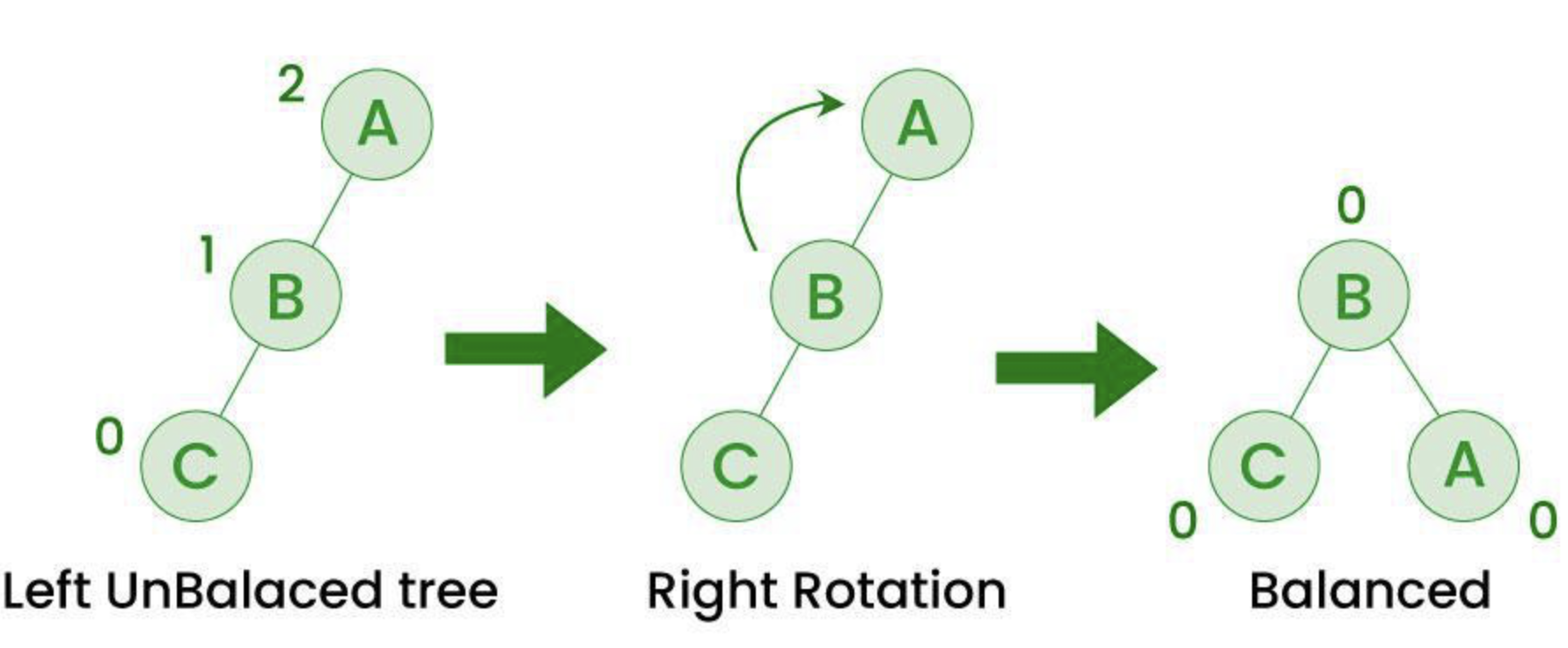


### AVL Tree Rotations

* Rotations are the most important part of the working of the AVL tree.
* They are responsible for maintaining the balance in the AVL tree.
* There are 4 types of rotations based on the 4 possible cases:
  + Right Rotation (RR)
  + Left Rotation (LL)
  + Left-Right Rotation (LR)
  + Right-Left Rotation (RL)

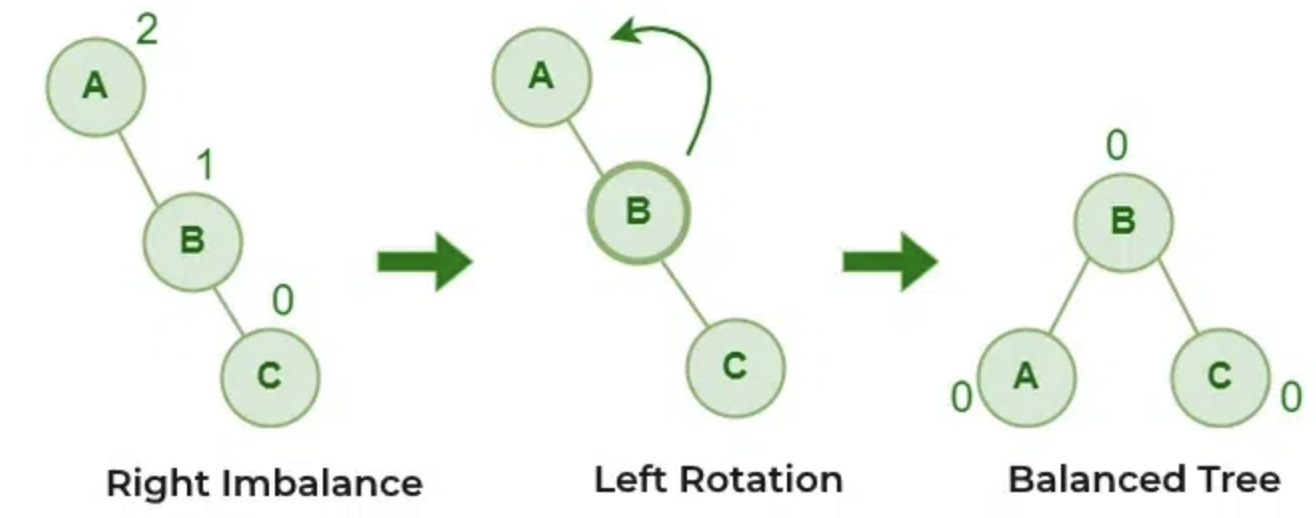
#### Right Rotation (RR)

* The Right Rotation (RR) is applied in an AVL tree when a node becomes unbalanced due to an insertion into the right subtree of its right child, leading to a Left Imbalance.
* To correct this imbalance, the unbalanced node is rotated 90° to the right (clockwise) along the top edge connected to its parent.



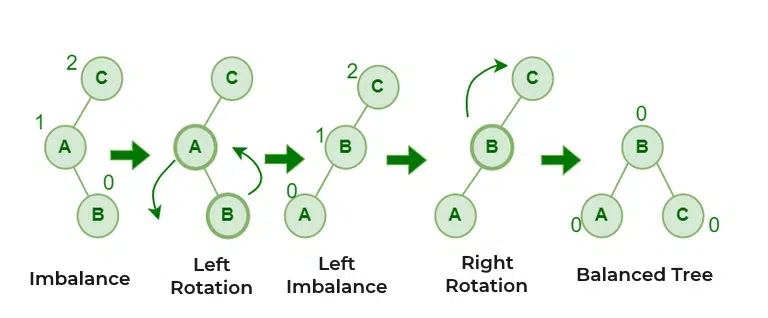
#### Left Rotation (LL)

* The Left Rotation (LL) is used to balance a node that becomes unbalanced due to an insertion into the left subtree of its left child, also resulting in a Left Imbalance.
* The solution is to rotate the unbalanced node 90° to the left (anti-clockwise) along the top edge connected to its parent.



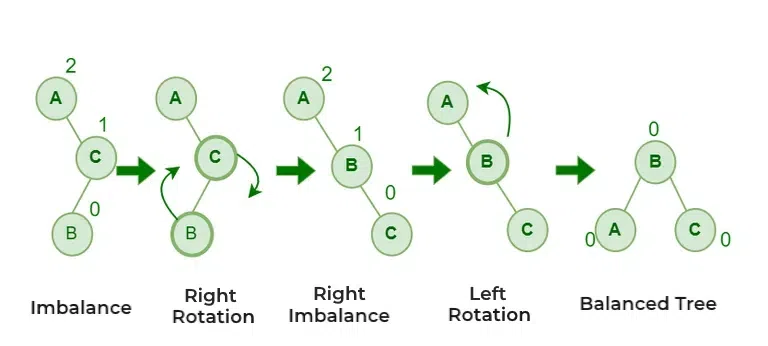
#### Left-Right Rotation (LR)

* The Left-Right Rotation (LR) is necessary when the left child of a node is right-heavy, creating a double imbalance.
* This situation is resolved by performing a left rotation on the left child, followed by a right rotation on the original node.



#### Right-Left Rotation (RL)

* The Right-Left Rotation (RL) is used when the right child of a node is left-heavy.
* This imbalance is corrected by performing a right rotation on the right child, followed by a left rotation on the original node.



### Representation of AVL Tree in C++

* The following diagram represents the structure of an AVL Tree where the balance factor of each node is 0 and the tree is balanced:
* To represent an AVL Tree in C++, we will use a class AVLNode to define the node structure and a class AVLTree to implement the AVL tree operations.

class **AVLNode** {

public:  
 int key;  
 AVLNode\* left;  
 AVLNode\* right;  
 int height;  
}

* **key:**represents the value stored inside the node.
* **left &right:** are pointers to the left and right node.
* **height:**represents the height of each subtree starting from each node.

**Implementation of Basic Operations of an AVL Tree in C++**

Following are the basic operations that are required to work with an AVL tree:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation Name | Description | Time Complexity | Space Complexity |
| Insert | Inserts a new element into the tree | O(log n) | O(log n) |
| Delete Node | Removes an element from the tree | O(log n) | O(log n) |
| Search | Searches for an element in the tree | O(log n) | O(log n) |
| Rotate Left | Performs left rotation to balance the AVL tree | O(1) | O(1) |
| Rotate Right | Performs right rotation to balance the AVL tree | O(1) | O(1) |

### Implementation of Insert Function

1. *Start at the root.*
2. *Compare the new value with the current node.*
3. *If less, move to the left child. If greater, move to the right child.*
4. *Repeat until reaching a null position.*
5. *Insert the new node at this position.*
6. *Update the height of the current node.*
7. *Calculate the balance factor of the current node.*
8. *If the balance factor is >1 or <-1, perform necessary rotations:*
   * *Left-Left case: Right rotation*
   * *Left-Right case: Left rotation on left child, then right rotation.*
   * *Right-Right case: Left rotation*
   * *Right-Left case: Right rotation on right child, then left rotation.*
9. *Repeat steps 6-8 while moving back up to the root.*

### Implementation of Delete Node Function

1. *Start at the root.*
2. *Search for the node to delete.*
3. *If the node is a leaf, simply remove it.*
4. *If the node has one child, replace it with its child.*
5. *If the node has two children:*
   * *Find the in-order successor (minimum value in right subtree).*
   * *Replace the node to be deleted with the in-order successor.*
   * *Delete the in-order successor from its original position.*
6. *Update the height of the current node.*
7. *Calculate the balance factor of the current node.*
8. *If the balance factor is >1 or <-1, perform necessary rotations:*
   * *Left-Left case: Right rotation*
   * *Left-Right case: Left rotation on left child, then right rotation*
   * *Right-Right case: Left rotation*
   * *Right-Left case: Right rotation on right child, then left rotation*
9. *Repeat steps 6-8 while moving back up to the root.*

### Implementation of Search Function

* *Start from the root.*
* *Compare the value with the current node.*
* *If equal, return true.*
* *If less, move to the left child.*
* *If greater, move to the right child.*
* *Repeat until found or reached a leaf node.*

### Implementation of Rotate Left Function

* *Start with a node A that has a right child B.*
* *Make B's left subtree the right subtree of A.*
* *Make A the left child of B.*
* *Update the heights of A and B.*
* *Return B as the new root of this subtree.*

### Implementation of Rotate Right Function

* *Start with a node A that has a left child B.*
* *Make B's right subtree the left subtree of A.*
* *Make A the right child of B.*
* *Update the heights of A and B.*
* *Return B as the new root of this subtree.*

//AVL Tree Implementation

// Online C++ compiler to run C++ program online

#include <iostream>

using namespace std;

class AVLNode

{

public:

AVLNode\* left;

AVLNode\* right;

int height;

int data;

AVLNode(int value) : data(value), left(nullptr), right(nullptr), height(1){} //new node is initially added at height 1

};

class AVLTree

{

private:

int getHeight(AVLNode\* node)

{

return node ? node->height : 0;

}

int getBalance(AVLNode\* node)

{

return node ? getHeight(node->left) -

getHeight(node->right) : 0;

}

AVLNode\* rightRotate(AVLNode\* rt)

{

AVLNode\* x = rt->left;

AVLNode\* temp = x->right;

//perform rotation

x->right = rt;

rt->left = temp;

//update height

rt->height = max(getHeight(rt->left), getHeight(rt->right)) + 1;

x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

return x; //return new root

}

AVLNode\* leftRotate(AVLNode\* lt)

{

AVLNode\* y = lt->right;

AVLNode\* temp = y->left;

//perform roration

y->left = lt;

lt->right = temp;

//update height

lt->height = max(getHeight(lt->left), getHeight(lt->right)) + 1;

y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

return y; //return new root

}

//find node with min value

AVLNode\* getMinVal(AVLNode\* node)

{

AVLNode\* current = node;

while(current->left)

{

current = current->left;

}

return current;

}

//delete node in AVL tree

AVLNode\* deleteNode(AVLNode\* root, int val)

{

//perform BST delete

if(!root)

return root;

if(val < root->data)

root->left = deleteNode(root->left, val);

else if(val > root->data)

root->right = deleteNode(root->right, val);

else

{

//Node with one child or no child

if(!root->left || !root->right)

{

AVLNode\* temp = root->left ? root->left : root->right;

if(!temp)

{

temp = root;

root = nullptr;

}

else

{

//copy contents of non-empty child

\*root = \*temp;

}

delete temp;

}

else //node with two children

{

//get inorder successor

AVLNode\* temp = getMinVal(root->right);

root->data = temp->data; //copy inorder successor data to this node

root->right = deleteNode(root->right, temp->data); //delete inorder successor

}

}

//if tree has only one node

if(!root)

return root;

root->height = 1+ max(getHeight(root->left), getHeight(root->right)); //update heigght of current node

int balance = getBalance(root); //get balance factor

//balance if the node if it has become unbalanced

//1: left-left case

if(balance > 1 && getBalance(root->left) >= 0)

{

return rightRotate(root);

}

//2: left-right case

if(balance > 1 && getBalance(root->left) < 0)

{

root->left = leftRotate(root->left);

return rightRotate(root);

}

//3: right-right case

if(balance < -1 && getBalance(root->right) <= 0)

{

return leftRotate(root);

}

//4: right-left case

if(balance < -1 && getBalance(root->right) > 0)

{

root->right = leftRotate(root->right);

return leftRotate(root);

}

return root;

}

void inOrderTraversal(AVLNode\* root)

{

if(root)

{

inOrderTraversal(root->left);

cout<<"Data: "<<root->data<<", Height: "<<root->height<<", Balance: "<<getBalance(root)<<endl;

inOrderTraversal(root->right);

}

}

public:

AVLNode\* root;

AVLTree() : root(nullptr){};

void insert(int val)

{

root = insertNode(root, val);

}

AVLNode\* insertNode(AVLNode\* node, int value)

{

if(!node)

{

return new AVLNode(value);

}

if(value < node->data)

node->left = insertNode(node->left, value);

else if(value > node->data)

node->right = insertNode(node->right, value);

else

return node;

//update height of node

node->height = 1+ max(getHeight(node->left), getHeight(node->right));

int balance = getBalance(node); //get balance factor

//if node is unbalanced, perform rotations

//1: left-left rotation

if(balance >1 && value < node->left->data)

return rightRotate(node);

//2: right-right rotation

if(balance < -1 && value > node->right->data)

return leftRotate(node);

//3: left-right case

if(balance >1 && value > node->left->data)

{

node->left = leftRotate(node->left);

return rightRotate(node);

}

//4: right-left case

if(balance < -1 && value < node->right->data)

{

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

void removeNd(int val)

{

root = deleteNode(root, val);

}

void inOrder()

{

inOrderTraversal(root);

cout<<endl;

}

};

int main() {

AVLTree tree;

tree.insert(10);

tree.insert(20);

tree.insert(30);

tree.insert(40);

tree.insert(50);

tree.insert(60);

tree.insert(25);

cout<<"Inorder traversal of AVL tree: "<<endl;

tree.inOrder();

tree.removeNd(30);

cout<<"Inorder traversal of AVL tree after deleting 30: "<<endl;

tree.inOrder();

return 0;

}

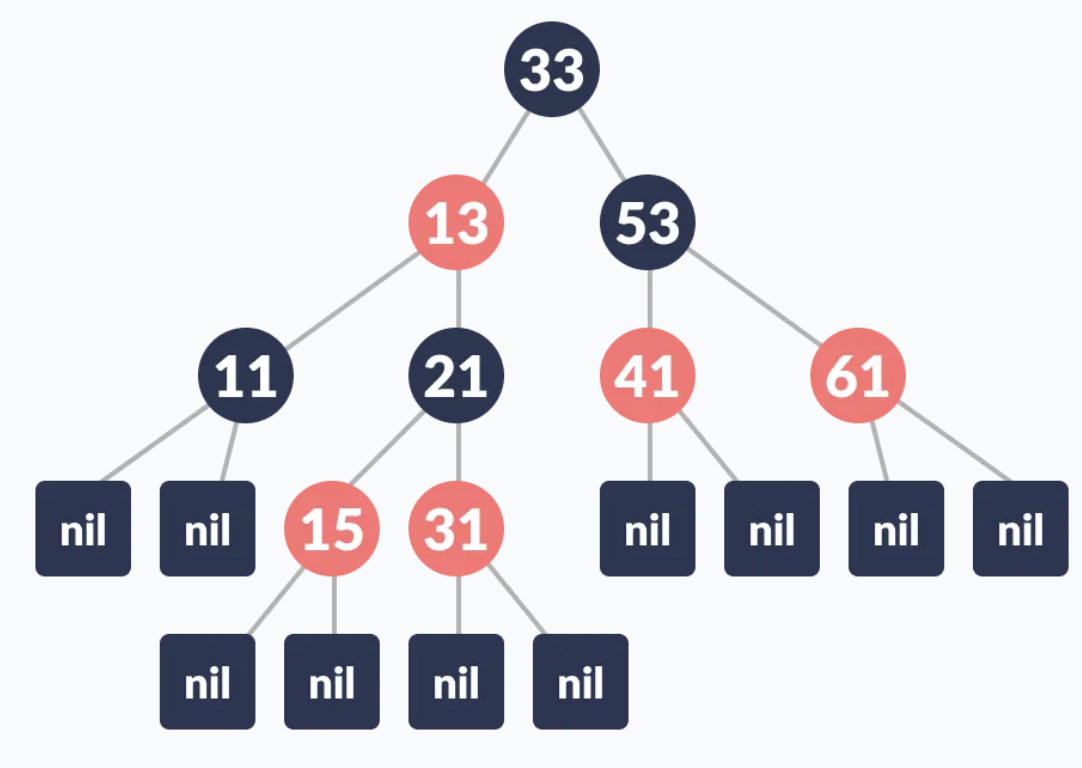
## Red Black Tree

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

* **Red/Black Property:** Every node is colored, either red or black.
* **Root Property:** The root is black.
* **Leaf Property:** Every leaf (NIL) is black.
* **Red Property:** If a red node has children then, the children are always black.
* **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

An example of a red-black tree is:



Each node has the following attributes:

* color
* key
* leftChild
* rightChild
* parent (except root node)

### How the red-black tree maintains the property of self-balancing?

* The red-black color is meant for balancing the tree.
* The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such path. It helps in maintaining the self-balancing property of the red-black tree.

### Operations on a Red-Black Tree

Various operations that can be performed on a red-black tree are:

#### Rotating the subtrees in a Red-Black Tree

* In rotation operation, the positions of the nodes of a subtree are interchanged.
* Rotation operation is used for maintaining the properties of a red-black tree when they are violated by other operations such as insertion and deletion.

There are two types of rotations:

#### Left Rotate

In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

Algorithm:

**Steps**

1. Let x be the node where the rotation is being performed.
2. Assign y = x->right (the right child of x).
3. Move y->left to x->right.
   1. If y->left exists, update its parent to x.
4. Update y->parent to x->parent.
   1. If x is the root, update the root pointer to y.
   2. Otherwise, update x->parent->left or x->parent->right to point to y.
5. Make x the left child of y.
6. Update x->parent to y.

Pseudocode

LeftRotate(Tree, x):

y = x.right // Set y

x.right = y.left // Turn y's left subtree into x's right subtree

if y.left ≠ NULL:

y.left.parent = x

y.parent = x.parent // Link x's parent to y

if x.parent == NULL: // If x is the root

Tree.root = y

else if x == x.parent.left:

x.parent.left = y

else:

x.parent.right = y

y.left = x // Put x on y's left

x.parent = y

#### Right Rotation:

Pivot the subtree to the right.

Make the left child the new root of the subtree, and the current root becomes the right child of the new root.

Algorithm

**Steps**

1. Let y be the node where the rotation is being performed.
2. Assign x = y->left (the left child of y).
3. Move x->right to y->left.
   1. If x->right exists, update its parent to y.
4. Update x->parent to y->parent.
   1. If y is the root, update the root pointer to x.
   2. Otherwise, update y->parent->left or y->parent->right to point to x.
5. Make y the right child of x.
   1. Update y->parent to x.

Pseudocode

RightRotate(Tree, y):

x = y.left // Set x

y.left = x.right // Turn x's right subtree into y's left subtree

if x.right ≠ NULL:

x.right.parent = y

x.parent = y.parent // Link y's parent to x

if y.parent == NULL: // If y is the root

Tree.root = x

else if y == y.parent.right:

y.parent.right = x

else:

y.parent.left = x

x.right = y // Put y on x's right

y.parent = x

*Time Complexity*: O(1) for a single rotation.

*Key Points*

* Red-Black Trees balance themselves automatically during insertion and deletion.
* They maintain a height of O(log n), ensuring efficient search, insert, and delete operations.
* Rotations and recolouring are the core mechanisms for maintaining balance.

This makes Red-Black Trees a robust and practical choice for real-world applications requiring balanced dynamic sets.

### Algorithm for Insertion

Insert(node, key):

1. Perform a standard BST insertion.

2. Color the newly inserted node RED.

3. Fix any violations:

a. If the parent of the node is BLACK:

i. Do nothing (Tree is still valid).

b. If the parent is RED:

i. Check the uncle node's color:

- If RED:

1. Recolor parent and uncle to BLACK.

2. Recolor the grandparent to RED.

3. Continue fixing at the grandparent.

- If BLACK:

1. Perform rotations to balance the tree.

2. Recolor nodes appropriately.

4. Ensure the root is BLACK.

### Algorithm for Deletion

Delete(node, key):

1. Perform a standard BST deletion.

2. Fix any violations:

a. If the node to be deleted is RED:

i. Delete directly (No violations occur).

b. If the node to be deleted is BLACK:

i. Fix the double-black violation:

- If the sibling is RED:

1. Recolor sibling and parent.

2. Perform a rotation.

- If the sibling is BLACK with BLACK children:

1. Recolor sibling to RED.

2. Move double-black up to the parent.

- If the sibling is BLACK with at least one RED child:

1. Perform a rotation and recolor nodes.

3. Ensure the root is BLACK.

### Black Height Property

The **black height property** is a fundamental characteristic of a Red-Black Tree (RBT) that ensures its balanced structure and logarithmic height. Here’s what it means and its implications:

**Definition of Black Height**

The **black height** of a node in a Red-Black Tree is the number of black nodes on the path from that node to any leaf, not counting the node itself. This property includes the following rules:

1. Every path from a node to its descendant leaf (null) nodes must contain the **same number of black nodes**.
2. This count is called the **black height (bh)** of the node.

**Significance of Black Height**

1. **Ensures Balance**:

* The black height property ensures that the tree remains balanced, with the longest path being at most twice as long as the shortest path.
* This guarantees a height of O(log n) for the tree.

1. **Consistency in Traversal**:

* Since every path to a leaf has the same black height, the black height property makes traversal and searches predictable and efficient.

**Implications**

1. If a Red-Black Tree has n nodes:
   1. The shortest possible path has only black nodes.
   2. The longest possible path alternates between red and black nodes.
   3. The height of the tree is at most 2 \* bh.
2. The black height provides a measure of the tree’s balance:
   1. The maximum height of the tree is logarithmic relative to the number of nodes.

// Online C++ compiler to run C++ program online

#include <iostream>

using namespace std;

enum Colour{Red, Black};

class RBNode

{

public:

int data;

Colour col;

RBNode\* left;

RBNode\* right;

RBNode\* parent;

RBNode(int value)

{

data = value;

left = right = parent = nullptr;

col = Red;

}

};

class RBTree

{

private:

RBNode\* root;

void leftRotate(RBNode\*& root, RBNode\*& node)

{

RBNode\* rtChild = node->right;

node->right = rtChild->left;

if(rtChild->left != nullptr)

rtChild->left->parent = node;

rtChild->parent = node->parent;

if(node->parent == nullptr)

root = rtChild;

else if(node == node->parent->left)

node->parent->left = rtChild;

else

node->parent->right = rtChild;

rtChild->left = node;

node->parent = rtChild;

}

void rightRotate(RBNode\*& root, RBNode\*& node)

{

RBNode\* ltChild = node->left;

node->left = ltChild->right;

if(ltChild->right != nullptr)

ltChild->right->parent = node;

ltChild->parent = node->parent;

if(node->parent == nullptr)

root = ltChild;

else if(node == node->parent->left)

node->parent->left = ltChild;

else

node->parent->right = ltChild;

ltChild->right = node;

node->parent = ltChild;

}

void fixInsert(RBNode\*& root, RBNode\*& node)

{

RBNode\* parent = nullptr;

RBNode\* gdParent = nullptr;

while(node!= root && node->col == Red && node->parent->col == Red)

{

parent = node->parent;

gdParent = parent->parent;

//A: parent is the left child of grand parent

if(parent == gdParent->left)

{

RBNode\* uNode = gdParent->right;

//1: uNode is red (recolouring)

if(uNode != nullptr && uNode->col == Red)

{

gdParent->col = Red;

parent->col = Black;

uNode->col = Black;

node = gdParent;

}

else

{

//2: node is right child (left rotate)

if(node == parent->right)

{

leftRotate(root, parent);

node = parent;

parent = node->parent;

}

//3: node is left child (right rotate)

rightRotate(root, gdParent);

swap(parent->col, gdParent->col);

node = parent;

}

}

else //B: parent is right child of grand parent

{

RBNode\* uNode = gdParent->left;

//1: uNode is red (recolourning)

if(uNode != nullptr && uNode->col == Red)

{

gdParent->col = Red;

parent->col = Black;

uNode->col = Black;

node = gdParent;

}

else //2: node is left child (right rotate)

{

if(node == parent->left)

{

rightRotate(root, parent);

node = parent;

parent = node->parent;

}

//3: node is right child (left rotate)

leftRotate(root, gdParent);

swap(parent->col, gdParent->col);

node = parent;

}

}

}

root->col = Black; //ensure root is always black

}

void inOrderTraversal(RBNode\* root)

{

if(root == nullptr)

return;

inOrderTraversal(root->left);

cout<<root->data<<" ("<<(root->col == Red ? "R" : "B")<<") ";

cout<<"Black height: "<<calculateBH(root)<<endl;

inOrderTraversal(root->right);

}

void fixDelete(RBNode\*& root, RBNode\*& node)

{

while(node != root && (node == nullptr || node->col == Black))

{

if(node == node->parent->left)

{

RBNode\* sibling = node->parent->right;

//1: red sibling

if(sibling-> col == Red)

{

sibling->col = Black;

node->parent->col = Red;

leftRotate(root, node->parent);

sibling = node->parent->right;

}

//2: sibling and its children are black

if((sibling->left == nullptr || sibling->left->col == Black) && (sibling->right == nullptr || sibling->right->col == Black))

{

sibling->col = Red;

node = node->parent;

}

else

{

//3: sibling's right child is black

if(sibling->right == nullptr || sibling->right->col == Black)

{

if(sibling->left != nullptr)

sibling->left->col = Black;

sibling->col = Red;

rightRotate(root, sibling);

sibling = node->parent->right;

}

//4: sibling's right child is red

sibling->col = node->parent->col;

node->parent->col = Black;

if(sibling->right != nullptr)

sibling->right->col = Black;

leftRotate(root, node->parent);

node = root;

}

}

else //symmetric case for right child

{

RBNode\* sibling = node->parent->left;

if(sibling->col == Red)

{

sibling->col = Black;

node->parent->col = Red;

rightRotate(root, node->parent);

sibling = node->parent->left;

}

if((sibling->left == nullptr || sibling->left->col == Black) && (sibling->right == nullptr || sibling->right->col == Black))

{

sibling->col = Red;

node = node->parent;

}

else

{

if(sibling->left == nullptr || sibling->left->col == Black)

{

if(sibling->right != nullptr)

sibling->right->col = Black;

sibling->col = Red;

leftRotate(root, sibling);

sibling = node->parent->left;

}

sibling->col = node->parent->col;

node->parent->col = Black;

if(sibling->left != nullptr)

sibling->left->col = Black;

rightRotate(root, node->parent);

node = root;

}

}

}

if(node!=nullptr)

node->col = Black;

}

RBNode\* deleteBST(RBNode\* root, int val)

{

if(root == nullptr)

return root;

if(val < root->data)

return deleteBST(root->left, val);

else if(val > root->data)

return deleteBST(root->right, val);

//node with one or no child

if(root->left == nullptr || root->right == nullptr)

return root;

//node with two children. get inorder successor

RBNode\* temp = minValue(root->right);

root->data = temp->data;

return deleteBST(root->right, temp->data);

}

//get node with min value

RBNode\* minValue(RBNode\* node)

{

RBNode\* current = node;

while(current->left != nullptr)

current = current->left;

return current;

}

public:

RBTree() : root(nullptr){}

void insert(int value)

{

RBNode\* node = new RBNode(value);

if(root == nullptr)

{

root = node;

root->col = Black;

return;

}

RBNode\* temp = root;

RBNode\* parent = nullptr;

while (temp!= nullptr)

{

parent = temp;

if(value < temp->data)

temp = temp->left;

else

temp = temp->right;

}

node->parent = parent;

if(value < parent->data)

parent->left = node;

else

parent->right = node;

fixInsert(root, node);

}

void inOrder()

{

inOrderTraversal(root);

cout<<endl;

}

void deleteNode(int value)

{

RBNode\* delNode = deleteBST(root, value);

if(delNode == nullptr)

return;

RBNode\* child = (delNode->left != nullptr) ? delNode->left : delNode->right;

if(delNode->parent == nullptr)

root = child;

else if(delNode == delNode->parent->left)

delNode->parent->left = child;

else

delNode->parent->right = child;

if(child != nullptr)

child->parent = delNode->parent;

if(delNode->col == Black)

fixDelete(root, child);

delete delNode;

}

int calculateBH(RBNode\* node)

{

if(node == nullptr)

return 1;

int leftHt = calculateBH(node->left);

int rightHt = calculateBH(node->right);

return leftHt + (node->col == Black ? 1 :0);

}

};

int main()

{

RBTree rb;

rb.insert(10);

rb.insert(20);

rb.insert(30);

rb.insert(15);

rb.insert(25);

rb.insert(5);

rb.inOrder();

rb.deleteNode(15);

rb.inOrder();

return 0;

}

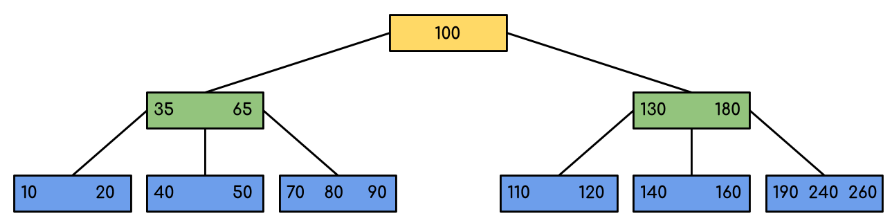
## B-Tree

* The limitations of traditional binary search trees can be frustrating. B-Tree, the multi-talented data structure that can handle massive amounts of data with ease.
* When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage.
* B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.
* Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees.
* Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height.
* This shallow height leads to less disk I/O, which results in faster search and insertion operations.
* B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.
* B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced.
* This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

### **Properties of B-Tree:**

* All leaf’s are at the same level.
* B-Tree is defined by the term minimum degree ‘**t**‘. The value of ‘**t**‘ depends upon disk block size.
* Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
* All nodes (including root) may contain at most (**2\*t – 1**) keys.
* Number of children of a node is equal to the number of keys in it plus **1**.
* All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
* B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
* Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O(log n).
* Insertion of a Node in B-Tree happens only at Leaf Node.

Following is an example of a B-Tree of minimum order 5   
**Note:** that in practical B-Trees, the value of the minimum order is much more than 5.



* We can see in the above diagram that all the leaf nodes are at the same level and all non-leaf’s have no empty sub-tree and have keys one less than the number of their children.

### **B-Tree Characteristics:**

* Every node can contain multiple keys and have multiple children.
* The tree is balanced, meaning that all leaves are at the same level.
* Each node has a minimum and maximum number of keys it can hold.
* Search, insertion, and deletion operations in a B-tree have a time complexity of O(\log n).

### **B-Tree Properties:**

* **Order (t)**: A B-tree is defined by its order t, where t is the minimum degree of the tree.

Every node has at most 2t - 1 keys and at least t - 1 keys.

Every internal node has at least t children (except the root).

The root node can have as few as 1 key.

* **Height**: The height of the tree is logarithmic relative to the number of elements. This ensures balanced data access.

### B-Tree Algorithm:

Here is the general **algorithm** for **insertion** and **searching** in a B-tree:

#### B-tree Insertion Algorithm:

The B-tree insertion process involves:

• Searching for the appropriate position to insert the new key.

• If the node has space, insert the key.

• If the node is full, split it into two nodes, and propagate the middle key upward.

**Steps:**

1. **Search for the correct leaf node** where the new key should be inserted.

2. **Insert the key** into the node in sorted order.

3. **If the node is full** (i.e., it contains 2t - 1 keys):

• Split the node into two halves.

• Move the middle key up into the parent node.

4. **Repeat the splitting process** up the tree if necessary until the root is split, creating a new root.

#### B-tree Search Algorithm:

To search for a key in a B-tree, we:

• Traverse the tree, comparing the target key with the keys in the current node.

• Depending on the comparison, we move to the left or right child, repeating this process until the key is found or the leaf node is reached.

**Steps:**

1. **Start from the root node.**
2. **Compare the key** with the keys in the current node.

• If the key is found, return the key.

• Otherwise, move to the child node where the key would logically be, and repeat the process.

1. **If a leaf node is reached** and the key is not found, then the key does not exist in the tree.

#### B-tree Deletion Algorithm:

Deleting a key in a B-tree is more complex than insertion, as it requires rebalancing the tree if nodes end up with too few keys. Here’s how deletion works:

• **Find the node** that contains the key to be deleted.

• **If the key is in a leaf node**, simply delete the key.

• **If the key is in an internal node**, replace it with the largest key from the left child or the smallest key from the right child, and recursively delete the replacement key.

• **If a node has fewer than the minimum number of keys** after a deletion, perform a **rotation** or **merge** operation to restore the tree’s balance.

**Steps:**

1. **Locate the key** to delete by traversing the tree.

2. **If the key is in a leaf node**, remove it.

3. **If the key is in an internal node**:

• Replace the key with the in-order predecessor or successor.

• Recursively delete the predecessor or successor.

4. **If a node underflows** (has fewer than the minimum number of keys):

• Borrow a key from a sibling if possible.

• Otherwise, merge the node with a sibling and update the parent.

### Rules for root node and internal nodes

The rules for the **minimum** and **maximum** keys and children differ slightly between the **root node** and **internal nodes (non-root nodes)** in a B-tree. These differences exist because the root node has unique properties to allow for a smaller initial size during the creation of the tree.

#### Rules for Internal Nodes

1. **Keys**:

**Minimum Keys**:

* An internal node must have at least t - 1 keys, where **t** is the **minimum degree** of the tree.
* Example: For t = 3, an internal node must have at least 2 keys.

**Maximum Keys**:

* An internal node can have at most 2t - 1 keys.
* Example: For t = 3, an internal node can have at most 5 keys.

1. **Children**:

• **Minimum Children**:

• An internal node must have at least **t** children.

• **Maximum Children**:

• An internal node can have at most **2t** children.

#### Rules for Root Node

1. **Keys**:

**Minimum Keys**:

* The root node must have at least 1 key, regardless of **t**.
* This allows the tree to start small when it has just one key.

**Maximum Keys**:

* The root node can have at most **2t - 1** keys, just like any other node.

1. **Children:**

**Minimum Children**:

* The root must have at least 2 children if it is not a leaf node.
* Exception: If the tree has only one key (and thus only the root exists), the root will have no children.

**Maximum Children**:

* The root can have at most 2t children, like any other node.

#### Rules for Leaf Nodes

1. **Keys**:

Leaf nodes follow the same key rules as internal nodes:

* **Minimum Keys**: t - 1 keys.
* **Maximum Keys**: 2t - 1 keys.

Exception: When the root is also a leaf node (e.g., when the tree is very small), it may have fewer than t - 1 keys.

2. **Children**:

* **No Children**:
* Leaf nodes do not have children, so their child pointers are null or unused.

#### Summary of rules

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Node Type | Min Keys | Max Keys | Min Children | Max Children |
| **Root** | 1 | 2t - 1 | 2 (if not a leaf) | 2t |
| **Internal** | t - 1 | 2t - 1 | t | 2t |
| **Leaf** | t - 1 | 2t - 1 | 0 | 0 |

### Rules for Children in a B-Tree

In a B-tree, the relationship between nodes, keys, and their children is structured to maintain balance and the sorted order of keys. Here are the specific rules governing **children** in a B-tree:

#### Child Count per Node

**Minimum Number of Children**:

* Each internal node (non-root) must have at least t children, where t is the **minimum degree** of the B-tree.
* For example, if t = 3 , each internal node must have at least 3 children.
* The **root** node can have fewer than t children (special case).

**Maximum Number of Children**:

* Each node can have at most 2t children, corresponding to the maximum number of keys 2t - 1 plus one extra child pointer.

**Leaf Nodes**:

* Leaf nodes do not have any children. Their child pointers are null or unused.

**Relationship Between Keys and Children**

1. For a node with n keys:

• The node will have n + 1 children.

• Example: If a node has keys [k\_1, k\_2, k\_3], it will have 4 children:

• **Child 1** contains keys < k\_1 .

• **Child 2** contains keys \geq k\_1 and < k\_2 .

• **Child 3** contains keys \geq k\_2 and < k\_3 .

• **Child 4** contains keys \geq k\_3 .

2. **Keys Divide Children**:

• The keys act as separators for the child subtrees, ensuring that the child subtrees maintain the B-tree’s sorted property.

**3. Children During Splitting**

When a node becomes full ( 2t - 1 keys), it is split, and the children are redistributed:

* The middle key is promoted to the parent node.
* The left child receives the keys and children < the middle key.
* The right child receives the keys and children > the middle key.

**4. Children During Deletion**

1. If a node has fewer than t - 1 keys after deletion:

• Borrow a key from a sibling (if the sibling has \geq t keys), and adjust the child pointers accordingly.

• If borrowing is not possible, merge the node with a sibling, combining their children.

2. The child count must always conform to the t -based rules, even after deletion.

**Example**

For a B-tree of degree t = 3 :

1. **Internal Node**:

• Minimum keys: t - 1 = 2 , so minimum children: t = 3 .

• Maximum keys: 2t - 1 = 5 , so maximum children: 6 .

2. **Root Node**:

• Can have fewer than 2 keys and 3 children in special cases.

B-Tree Implementation

// Online C++ compiler to run C++ program online

#include <iostream>

#include <queue>

using namespace std;

class BTreeNode

{

int \*keys; // Array of keys

BTreeNode \*\*children; // Array of child pointers

int numKeys; // Current number of keys

bool isLeaf; // True if the node is a leaf

int t; // Minimum degree

public:

BTreeNode(int \_t, bool \_isLeaf);

void insertNonFull(int key);

void splitChild(int index, BTreeNode \*fullChild);

void traverse();

friend class BTree;

};

class BTree

{

BTreeNode \*root; // Pointer to root node

int t; // Minimum degree

public:

BTree(int \_t)

{

root = nullptr;

t = \_t;

}

void insert(int key);

void traverse()

{

if (root != nullptr) root->traverse();

}

void printLevelOrder();

};

// Constructor for BTreeNode

BTreeNode::BTreeNode(int \_t, bool \_isLeaf)

{

t = \_t;

isLeaf = \_isLeaf;

keys = new int[2 \* t - 1];

children = new BTreeNode \*[2 \* t];

numKeys = 0;

}

// Traverse the B-tree

void BTreeNode::traverse()

{

int i;

for (i = 0; i < numKeys; i++)

{

if (!isLeaf) children[i]->traverse();

cout << keys[i] << " ";

}

if (!isLeaf) children[i]->traverse();

}

// Insert a key into a non-full node

void BTreeNode::insertNonFull(int key)

{

int i = numKeys - 1;

if (isLeaf)

{

while (i >= 0 && keys[i] > key)

{

keys[i + 1] = keys[i];

i--;

}

keys[i + 1] = key;

numKeys++;

}

else

{

while (i >= 0 && keys[i] > key)

i--;

if (children[i + 1]->numKeys == 2 \* t - 1)

{

splitChild(i + 1, children[i + 1]);

if (keys[i + 1] < key) i++;

}

children[i + 1]->insertNonFull(key);

}

}

// Split a full child node

void BTreeNode::splitChild(int index, BTreeNode \*fullChild)

{

BTreeNode \*newNode = new BTreeNode(fullChild->t, fullChild->isLeaf);

newNode->numKeys = t - 1;

for (int j = 0; j < t - 1; j++)

newNode->keys[j] = fullChild->keys[j + t];

if (!fullChild->isLeaf)

{

for (int j = 0; j < t; j++)

newNode->children[j] = fullChild->children[j + t];

}

fullChild->numKeys = t - 1;

for (int j = numKeys; j >= index + 1; j--)

children[j + 1] = children[j];

children[index + 1] = newNode;

for (int j = numKeys - 1; j >= index; j--)

keys[j + 1] = keys[j];

keys[index] = fullChild->keys[t - 1];

numKeys++;

}

// Insert into the B-tree

void BTree::insert(int key)

{

if (root == nullptr)

{

root = new BTreeNode(t, true);

root->keys[0] = key;

root->numKeys = 1;

}

else

{

if (root->numKeys == 2 \* t - 1)

{

BTreeNode \*newRoot = new BTreeNode(t, false);

newRoot->children[0] = root;

newRoot->splitChild(0, root);

int i = 0;

if (newRoot->keys[0] < key)

i++;

newRoot->children[i]->insertNonFull(key);

root = newRoot;

}

else

{

root->insertNonFull(key);

}

}

}

// Print level-order traversal of the B-tree

void BTree::printLevelOrder()

{

if (!root)

{

cout << "The tree is empty."<<endl;

return;

}

queue<BTreeNode \*> q;

q.push(root);

int level = 0;

while (!q.empty())

{

int nodeCount = q.size();

cout << "Level " << level << ": ";

while (nodeCount > 0)

{

BTreeNode \*node = q.front();

q.pop();

// Print keys in the current node

for (int i = 0; i < node->numKeys; i++)

{

cout << node->keys[i] << " ";

}

cout << "| ";

// Push children of the current node

if (!node->isLeaf)

{

for (int i = 0; i <= node->numKeys; i++) {

q.push(node->children[i]);

}

}

nodeCount--;

}

cout << endl;

level++;

}

}

// Main function to test B-tree

int main() {

BTree btree(3);

// Insert values 1 to 10

for (int i = 1; i <= 10; i++) {

btree.insert(i);

}

cout << "Traversal of the B-tree: ";

btree.traverse();

cout << endl;

cout << "Level-order traversal of the B-tree: "<<endl;

btree.printLevelOrder();

return 0;

}