# I. Binary Tree

## Ex1:

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

Node \*createBinaryTree(int arr[], int n, int index)

{

    if (index >= n)

    {

        return nullptr;

    }

    Node \*root = new Node(arr[index]);

    root->left = createBinaryTree(arr, n, index \* 2 + 1);

    root->right = createBinaryTree(arr, n, index \* 2 + 2);

    return root;

}

double averageNodeValue(Node \*root, int &count, int &sum)

{

    if (root == nullptr)

    {

        return 0;

    }

    sum += root->data;

    count++;

    averageNodeValue(root->left, count, sum);

    averageNodeValue(root->right, count, sum);

    return (double)sum / count;

}

double averagePositiveNodeValue(Node \*root, int &count, int &sum)

{

    if (root == nullptr)

    {

        return 0;

    }

    if (root->data > 0)

    {

        sum += root->data;

        count++;

    }

    averagePositiveNodeValue(root->left, count, sum);

    averagePositiveNodeValue(root->right, count, sum);

    return (double)sum / count;

}

double averageNegativeNodeValue(Node \*root, int &count, int &sum)

{

    if (root == nullptr)

    {

        return 0;

    }

    if (root->data < 0)

    {

        sum += root->data;

        count++;

    }

    averageNegativeNodeValue(root->left, count, sum);

    averageNegativeNodeValue(root->right, count, sum);

    return (double)sum / count;

}

double ratioPosToNeg(Node \*root, int &sumPos, int &sumNeg)

{

    if (root == nullptr)

    {

        return 0;

    }

    if (root->data > 0)

    {

        sumPos += root->data;

    }

    if (root->data < 0)

    {

        sumNeg += root->data;

    }

    ratioPosToNeg(root->left, sumPos, sumNeg);

    ratioPosToNeg(root->right, sumPos, sumNeg);

    return (sumNeg != 0) ? (double)sumPos / sumNeg : 0;

}

int main()

{

    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, -12, -13, 14};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n, 0);

    int count = 0, sum = 0;

    double avgAll = averageNodeValue(root, count, sum);

    std::cout << "Average value of all nodes in the tree: " << avgAll << std::endl;

    int posCount = 0, posSum = 0;

    double avgPos = averagePositiveNodeValue(root, posCount, posSum);

    std::cout << "Average value of positive nodes in the tree: " << avgPos << std::endl;

    int negCount = 0, negSum = 0;

    double avgNeg = averageNegativeNodeValue(root, negCount, negSum);

    std::cout << "Average value of negative nodes in the tree: " << avgNeg << std::endl;

    int sumPos = 0, sumNeg = 0;

    double ratio = ratioPosToNeg(root, sumPos, sumNeg);

    std::cout << "Ratio of positive sum to negative sum: " << ratio << std::endl;

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, -12, -13, -14] | 3.92857 |
| 6.66667 |
| -12.5 |
| -3.2 |

## Ex2:

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

void modifyNode(Node \*root)

{

    if (root == nullptr)

    {

        return;

    }

    Node \*temp = root->left;

    root->left = root->right;

    root->right = temp;

    temp = root->left->left;

    root->left->left = root->left->right;

    root->left->right = temp;

}

Node \*createBinaryTree()

{

    Node \*root = new Node(10);

    root->left = new Node(5);

    root->right = new Node(15);

    root->left->left = new Node(3);

    root->left->right = new Node(9);

    root->right->left = new Node(12);

    root->right->right = new Node(18);

    root->right->right->left = new Node(20);

    root->left->right->left = new Node(7);

    return root;

}

void printTree(Node \*root)

{

    if (root == nullptr)

    {

        return;

    }

    std::cout << root->data << " ";

    printTree(root->left);

    printTree(root->right);

}

int main()

{

    Node \*root = createBinaryTree();

    std::cout << "Original tree: ";

    printTree(root);

    std::cout << std::endl;

    modifyNode(root);

    std::cout << "Modified tree: ";

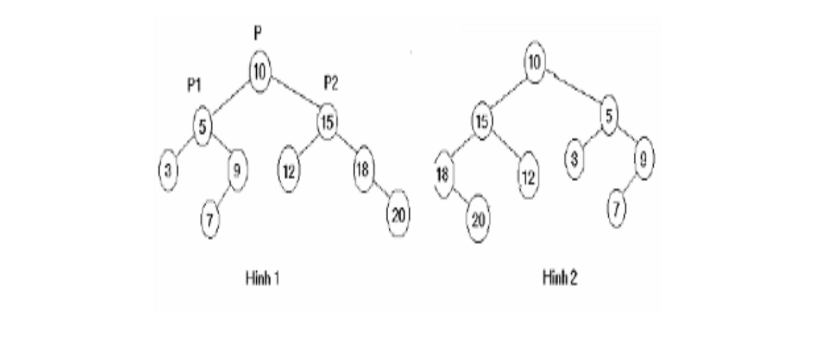
    printTree(root);

    std::cout << std::endl;

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| Original tree: 10 5 3 9 7 15 12 18 20 | Modified tree: 10 15 18 20 12 5 3 9 7 |



## Ex4:

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

Node \*createBinaryTree(int arr[], int n)

{

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    return root;

}

Node \*findMin(Node \*root)

{

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

    {

        root = root->left;

    }

    return root;

}

Node \*findMax(Node \*root)

{

    while (root != nullptr && root->right != nullptr)

    {

        root = root->right;

    }

    return root;

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n);

    Node \*minNode = findMin(root);

    Node \*maxNode = findMax(root);

    if (minNode != nullptr)

    {

        std::cout << "Minimum value in the tree: " << minNode->data << std::endl;

    }

    else

    {

        std::cout << "Tree is empty." << std::endl;

    }

    if (maxNode != nullptr)

    {

        std::cout << "Maximum value in the tree: " << maxNode->data << std::endl;

    }

    else

    {

        std::cout << "Tree is empty." << std::endl;

    }

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Minimum value in the tree: 3  Maximum value in the tree: 20 |

## Ex4:

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

Node \*createBinaryTree(int arr[], int n)

{

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    return root;

}

int totalOneChildNode(Node \*root)

{

    if (root == nullptr)

    {

        return 0;

    }

    int count = 0;

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

    {

        count = 1;

    }

    count += totalOneChildNode(root->left);

    count += totalOneChildNode(root->right);

    return count;

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n);

    int count = totalOneChildNode(root);

    std::cout << "Total nodes with one child: " << count << std::endl;

    return 0;

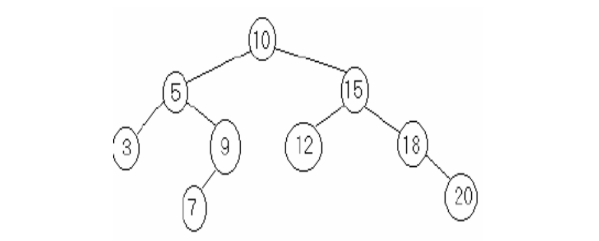
}

* Công thức đệ quy

- T(n) = 0, nếu n là NULL

- T(n) = 1 + T(left) + T(right), nếu n có một nhánh con (con trái hoặc con phải)

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Total nodes with one child: 2 |



## Ex5:

1. **Công thức đệ quy để tính số nút trong cây nhị phân tìm kiếm**:

* **Công thức**:
* **T(n) = 0, nếu n là NULL**: Nếu nút là NULL (tức là không có nút), thì không có nút nào.
* **T(n) = 1 + T(left) + T(right), nếu n không phải là NULL**: Nếu nút không phải là NULL, ta tính tổng số nút trong cây con trái và cây con phải, cộng với 1 cho chính nút đó.

int countNodes(Node\* root) {

    if (root == nullptr) {

        return 0;

    }

    return 1 + countNodes(root->left) + countNodes(root->right);

}

2. **Công thức đệ quy để tính tổng giá trị các nút trong cây**:

* **Công thức**:
  + **T(n) = 0, nếu n là NULL**: Nếu nút là NULL, tức là không có nút, thì không có giá trị nào.
  + **T(n) = giá trị của nút + T(left) + T(right), nếu n không phải là NULL**: Nếu nút không phải là NULL, ta tính tổng giá trị cây con trái và cây con phải, cộng với giá trị của chính nút đó.

int sumValues(Node\* root) {

    if (root == nullptr) {

        return 0;

    }

    return root->value + sumValues(root->left) + sumValues(root->right);

}

## Ex6:

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

Node \*createBinaryTree(int arr[], int n)

{

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    return root;

}

int countNodes(Node \*root)

{

    if (root == nullptr)

    {

        return 0;

    }

    return 1 + countNodes(root->left) + countNodes(root->right);

}

int sumValues(Node \*root)

{

    if (root == nullptr)

    {

        return 0;

    }

    return root->data + sumValues(root->left) + sumValues(root->right);

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n);

    std::cout << "Total nodes: " << countNodes(root) << std::endl;

    std::cout << "Sum of values: " << sumValues(root) << std::endl;

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Total nodes: 9  Sum of values: 99 |

## Ex8:

#include <iostream>

#include <fstream>

struct Node

{

    double data;

    Node \*left;

    Node \*right;

    Node(double val)

    {

        data = val;

        left = nullptr;

        right = nullptr;

    }

};

Node \*insertNode(Node \*root, double value)

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

Node \*createBinaryTree(double arr[], int n)

{

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    return root;

}

void inOrderTraversal(Node \*root, std::ofstream &outFile)

{

    if (root == nullptr)

    {

        return;

    }

    inOrderTraversal(root->left, outFile);

    outFile.write((char \*)&root->data, sizeof(root->data));

    inOrderTraversal(root->right, outFile);

}

int main()

{

    double arr[] = {20.5, 10.3, 30.7, 5.1, 15.2, 25.6, 35.4};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n);

    std::ofstream outFile("data\_ex8.out", std::ios::binary);

    if (!outFile)

    {

        std::cerr << "Error opening file!" << std::endl;

        return 1;

    }

    inOrderTraversal(root, outFile);

    outFile.close();

    std::cout << "Cây nhị phân đã được xuất ra tập tin data\_ex8.out" << std::endl;

    return 0;

}

## Ex9:

#include <iostream>

#include <fstream>

struct Node

{

    double data;

    Node \*left;

    Node \*right;

    Node(double val)

    {

        data = val;

        left = nullptr;

        right = nullptr;

    }

};

Node \*insertNode(Node \*root, double value)

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

Node \*createBinaryTree(double arr[], int n)

{

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    return root;

}

void preOrderTraversal(Node \*root, std::ofstream &outFile)

{

    if (root == nullptr)

    {

        return;

    }

    outFile.write((char \*)&root->data, sizeof(root->data));

    preOrderTraversal(root->left, outFile);

    preOrderTraversal(root->right, outFile);

}

int main()

{

    double arr[] = {20.5, 10.3, 30.7, 5.1, 15.2, 25.6, 35.4};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n);

    std::ofstream outFile("data\_ex9.out", std::ios::binary);

    if (!outFile)

    {

        std::cerr << "Error opening file!" << std::endl;

        return 1;

    }

    preOrderTraversal(root, outFile);

    outFile.close();

    std::cout << "Cây nhị phân đã được xuất ra tập tin data\_ex9.out" << std::endl;

    return 0;

}

## Ex10:

#include <iostream>

#include <fstream>

struct Node

{

    double data;

    Node \*left;

    Node \*right;

    Node(double val)

    {

        data = val;

        left = nullptr;

        right = nullptr;

    }

};

Node \*insertNode(Node \*root, double value)

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

Node \*createBinaryTree(double arr[], int n)

{

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    return root;

}

void postOrderTraversal(Node \*root, std::ofstream &outFile)

{

    if (root == nullptr)

    {

        return;

    }

    postOrderTraversal(root->left, outFile);

    postOrderTraversal(root->right, outFile);

    outFile.write((char \*)&root->data, sizeof(root->data));

}

int main()

{

    double arr[] = {20.5, 10.3, 30.7, 5.1, 15.2, 25.6, 35.4};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n);

    std::ofstream outFile("data\_ex10.out", std::ios::binary);

    if (!outFile)

    {

        std::cerr << "Error opening file!" << std::endl;

        return 1;

    }

    postOrderTraversal(root, outFile);

    outFile.close();

    std::cout << "Cây nhị phân đã được xuất ra tập tin data\_ex10.out" << std::endl;

    return 0;

}

## Ex13:

#include <iostream>

#include <fstream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

Node \*createBinaryTree(int arr[], int n)

{

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    return root;

}

void saveTree(Node \*root, std::ofstream &outFile)

{

    if (root == nullptr)

    {

        int nullValue = -1;

        outFile.write((char \*)&nullValue, sizeof(nullValue));

        return;

    }

    outFile.write((char \*)&root->data, sizeof(root->data));

    saveTree(root->left, outFile);

    saveTree(root->right, outFile);

}

Node \*loadTree(std::ifstream &inFile)

{

    int value;

    if (!inFile.read((char \*)&value, sizeof(value)))

    {

        return nullptr;

    }

    if (value == -1)

    {

        return nullptr;

    }

    Node \*root = new Node(value);

    root->left = loadTree(inFile);

    root->right = loadTree(inFile);

    return root;

}

void deleteTree(Node \*root)

{

    if (root == nullptr)

        return;

    deleteTree(root->left);

    deleteTree(root->right);

    delete root;

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = createBinaryTree(arr, n);

    std::ofstream outFile("tree\_data.dat", std::ios::binary);

    if (!outFile)

    {

        std::cerr << "Error opening file!" << std::endl;

        return 1;

    }

    saveTree(root, outFile);

    outFile.close();

    std::cout << "The binary tree has been saved to the file 'tree\_data.dat'." << std::endl;

    std::ifstream inFile("tree\_data.dat", std::ios::binary);

    if (!inFile)

    {

        std::cerr << "Error opening file!" << std::endl;

        return 1;

    }

    Node \*newRoot = loadTree(inFile);

    inFile.close();

    std::cout << "The binary tree has been read from the file." << std::endl;

    deleteTree(root);

    deleteTree(newRoot);

    return 0;

}

## Ex14:

#include <iostream>

struct NodeTree

{

    int data;

    NodeTree \*left;

    NodeTree \*right;

    NodeTree(int value) : data(value), left(nullptr), right(nullptr) {}

};

struct Node

{

    int data;

    Node \*next;

    Node(int value) : data(value), next(nullptr) {}

};

NodeTree \*insertNodeTree(NodeTree \*root, int value)

{

    if (root == nullptr)

    {

        return new NodeTree(value);

    }

    if (value < root->data)

    {

        root->left = insertNodeTree(root->left, value);

    }

    else if (value > root->data)

    {

        root->right = insertNodeTree(root->right, value);

    }

    return root;

}

NodeTree \*createBST(int arr[], int n)

{

    NodeTree \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNodeTree(root, arr[i]);

    }

    return root;

}

void insertNode(NodeTree \*root, Node \*&head)

{

    if (root == nullptr)

    {

        return;

    }

    insertNode(root->left, head);

    Node \*newNode = new Node(root->data);

    newNode->next = head;

    head = newNode;

    insertNode(root->right, head);

}

Node \*convertBSTtoLinkedList(NodeTree \*root)

{

    Node \*head = nullptr;

    insertNode(root, head);

    return head;

}

void printList(Node \*head)

{

    Node \*current = head;

    while (current != nullptr)

    {

        std::cout << current->data << " ";

        current = current->next;

    }

    std::cout << std::endl;

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    NodeTree \*root = createBST(arr, n);

    Node \*head = convertBSTtoLinkedList(root);

    printList(head);

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | 20 18 15 12 10 9 7 5 3 |

## Ex16:

#include <iostream>

struct Node

{

    int data;

    int count;

    Node \*left;

    Node \*right;

    Node(int value) : data(value), count(1), left(nullptr), right(nullptr) {}

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    else

    {

        root->count++;

    }

    return root;

}

Node \*search(Node \*root, int value)

{

    if (root == nullptr || root->data == value)

    {

        return root;

    }

    if (value < root->data)

    {

        return search(root->left, value);

    }

    else

    {

        return search(root->right, value);

    }

}

void deleteNode(Node \*&root, int value)

{

    Node \*foundNode = search(root, value);

    if (foundNode != nullptr)

    {

        foundNode->count--;

        if (foundNode->count == 0)

        {

            std::cout << "Node with value " << value << " is completely removed from the tree." << std::endl;

        }

        else

        {

            std::cout << "Decremented count for node with value " << value << " to " << foundNode->count << std::endl;

        }

    }

    else

    {

        std::cout << "Element " << value << " not found in the tree." << std::endl;

    }

}

void preOrder(Node \*root)

{

    if (root == nullptr)

    {

        return;

    }

    std::cout << "Value: " << root->data << ", Count: " << root->count << std::endl;

    preOrder(root->left);

    preOrder(root->right);

}

void deleteTree(Node \*root)

{

    if (root == nullptr)

        return;

    deleteTree(root->left);

    deleteTree(root->right);

    delete root;

}

int main()

{

    Node \*root = nullptr;

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    std::cout << "Binary Tree (Pre-order):" << std::endl;

    preOrder(root);

    std::cout << std::endl;

    int valueToDelete = 10;

    std::cout << "Deleting node with value " << valueToDelete << ":" << std::endl;

    deleteNode(root, valueToDelete);

    std::cout << "Binary Tree after deletion:" << std::endl;

    preOrder(root);

    deleteTree(root); // Clean up memory

    root = nullptr;   // Set root to nullptr after deletion

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Binary Tree (Pre-order):  Value: 10, Count: 1  Value: 5, Count: 1  Value: 3, Count: 1  Value: 9, Count: 1  Value: 7, Count: 1  Value: 15, Count: 1  Value: 12, Count: 1  Value: 18, Count: 1  Value: 20, Count: 1  Deleting node with value 10:  Node with value 10 is completely removed from the tree.  Binary Tree after deletion:  Value: 10, Count: 0  Value: 5, Count: 1  Value: 3, Count: 1  Value: 9, Count: 1  Value: 7, Count: 1  Value: 15, Count: 1  Value: 12, Count: 1  Value: 18, Count: 1  Value: 20, Count: 1 |

## Ex17:

#include <iostream>

struct Node

{

    int data;

    int count;

    Node \*left;

    Node \*right;

    Node(int value) : data(value), count(1), left(nullptr), right(nullptr) {}

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    else

    {

        root->count++;

    }

    return root;

}

Node \*findMin(Node \*root)

{

    while (root->left != nullptr)

    {

        root = root->left;

    }

    return root;

}

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

{

    if (root == nullptr)

    {

        return root;

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    else

    {

        if (root->left == nullptr)

        {

            Node \*temp = root->right;

            delete root;

            return temp;

        }

        else if (root->right == nullptr)

        {

            Node \*temp = root->left;

            delete root;

            return temp;

        }

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

        root->data = temp->data;

        root->right = deleteNode(root->right, temp->data);

    }

    return root;

}

void preOrder(Node \*root)

{

    if (root == nullptr)

    {

        return;

    }

    std::cout << "Value: " << root->data << ", Count: " << root->count << std::endl;

    preOrder(root->left);

    preOrder(root->right);

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    std::cout << "Tree before deletion:" << std::endl;

    preOrder(root);

    int valueToDelete = 10;

    root = deleteNode(root, valueToDelete);

    std::cout << "\nTree after deletion:" << std::endl;

    preOrder(root);

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Tree before deletion:  Value: 10, Count: 1  Value: 5, Count: 1  Value: 3, Count: 1  Value: 9, Count: 1  Value: 7, Count: 1  Value: 15, Count: 1  Value: 12, Count: 1  Value: 18, Count: 1  Value: 20, Count: 1  Tree after deletion:  Value: 12, Count: 1  Value: 5, Count: 1  Value: 3, Count: 1  Value: 9, Count: 1  Value: 7, Count: 1  Value: 15, Count: 1  Value: 18, Count: 1  Value: 20, Count: 1 |

## Ex19:

#include <iostream>

struct TreeNode

{

    int data;

    TreeNode \*left;

    TreeNode \*right;

    TreeNode(int x) : data(x), left(nullptr), right(nullptr) {}

};

TreeNode \*insert(TreeNode \*root, int data)

{

    if (root == nullptr)

    {

        return new TreeNode(data);

    }

    if (data < root->data)

    {

        root->left = insert(root->left, data);

    }

    else

    {

        root->right = insert(root->right, data);

    }

    return root;

}

TreeNode \*createBSTFromArray(const int a[], int n)

{

    TreeNode \*root = nullptr;

    for (int i = 0; i < n; i++)

    {

        root = insert(root, a[i]);

    }

    return root;

}

void inorder(TreeNode \*root, int result[], int &index)

{

    if (root == nullptr)

    {

        return;

    }

    inorder(root->left, result, index);

    result[index++] = root->data;

    inorder(root->right, result, index);

}

int main()

{

    int a[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(a) / sizeof(a[0]);

    TreeNode \*root = createBSTFromArray(a, n);

    int result[n];

    int index = 0;

    inorder(root, result, index);

    std::cout << "Sorted array: ";

    for (int i = 0; i < n; i++)

    {

        std::cout << result[i] << " ";

    }

    std::cout << std::endl;

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Sorted array: 3 5 7 9 10 12 15 18 20 |

## Ex32:

**<Stack>**

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

void preOrderWithoutRecursion(Node \*root)

{

    if (root == nullptr)

        return;

    Node \*stack[100];

    int top = -1;

    stack[++top] = root;

    while (top != -1)

    {

        Node \*currentNode = stack[top--];

        std::cout << currentNode->data << " ";

        if (currentNode->right)

        {

            stack[++top] = currentNode->right;

        }

        if (currentNode->left)

        {

            stack[++top] = currentNode->left;

        }

    }

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    std::cout << "Pre-order traversal without recursion: ";

    preOrderWithoutRecursion(root);

    std::cout << std::endl;

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Pre-order traversal without recursion: 10 5 3 9 7 15 12 18 20 |

**<Queue>**

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

void levelOrderWithoutRecursion(Node \*root)

{

    if (root == nullptr)

        return;

    Node \*queue[100];

    int front = 0, rear = 0;

    queue[rear++] = root;

    while (front < rear)

    {

        Node \*currentNode = queue[front++];

        std::cout << currentNode->data << " ";

        if (currentNode->left)

        {

            queue[rear++] = currentNode->left;

        }

        if (currentNode->right)

        {

            queue[rear++] = currentNode->right;

        }

    }

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    std::cout << "Level-order traversal without recursion: ";

    levelOrderWithoutRecursion(root);

    std::cout << std::endl;

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | Level-order traversal without recursion: 10 5 15 3 9 12 18 7 20 |

## Ex37:

#include <iostream>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

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

};

int max(int a, int b)

{

    return (a > b) ? a : b; // Manually computing max

}

int getHeight(Node \*p)

{

    if (p == nullptr)

        return 0;

    int leftHeight = getHeight(p->left);

    int rightHeight = getHeight(p->right);

    return max(leftHeight, rightHeight) + 1;

}

bool isBalanced(Node \*root)

{

    if (root == nullptr)

        return true;

    int leftHeight = getHeight(root->left);

    int rightHeight = getHeight(root->right);

    if (std::abs(leftHeight - rightHeight) > 1)

    {

        return false;

    }

    return isBalanced(root->left) && isBalanced(root->right);

}

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

{

    if (root == nullptr)

    {

        return new Node(value);

    }

    if (value < root->data)

    {

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

    }

    else if (value > root->data)

    {

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

    }

    return root;

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    if (isBalanced(root))

    {

        std::cout << "The tree is balanced." << std::endl;

    }

    else

    {

        std::cout << "The tree is not balanced." << std::endl;

    }

    return 0;

}

        return;

    Node \*queue[100];

    int front = 0, rear = 0;

    queue[rear++] = root;

    while (front < rear)

    {

        Node \*currentNode = queue[front++];

        std::cout << currentNode->data << " ";

        if (currentNode->left)

        {

            queue[rear++] = currentNode->left;

        }

        if (currentNode->right)

        {

            queue[rear++] = currentNode->right;

        }

    }

}

int main()

{

    int arr[] = {10, 5, 3, 9, 7, 15, 12, 18, 20};

    int n = sizeof(arr) / sizeof(arr[0]);

    Node \*root = nullptr;

    for (int i = 0; i < n; ++i)

    {

        root = insertNode(root, arr[i]);

    }

    std::cout << "Level-order traversal without recursion: ";

    levelOrderWithoutRecursion(root);

    std::cout << std::endl;

    return 0;

}

|  |  |
| --- | --- |
| Input | Output |
| 10 5 3 9 7 15 12 18 20 | The tree is balanced. |