Tamia Naeem
AI-004

Exercise

1. Implement methods for the following operations in Binary tree class given in example 01.

o Searching a node based on a given value.

o Pre-order traversal

o Post-order traversal

Source code

```
#include <bits/stdc++.h>
using namespace std;
class Node {
public:
   int value;
   Node* left;
   Node* right;
// constructor
   Node(int val) {
      value = val;
      left = right = nullptr;
   }
};
class Tamia_004 {
public:
   Node* root;
   Tamia_004() {
      root = nullptr;
   }
```

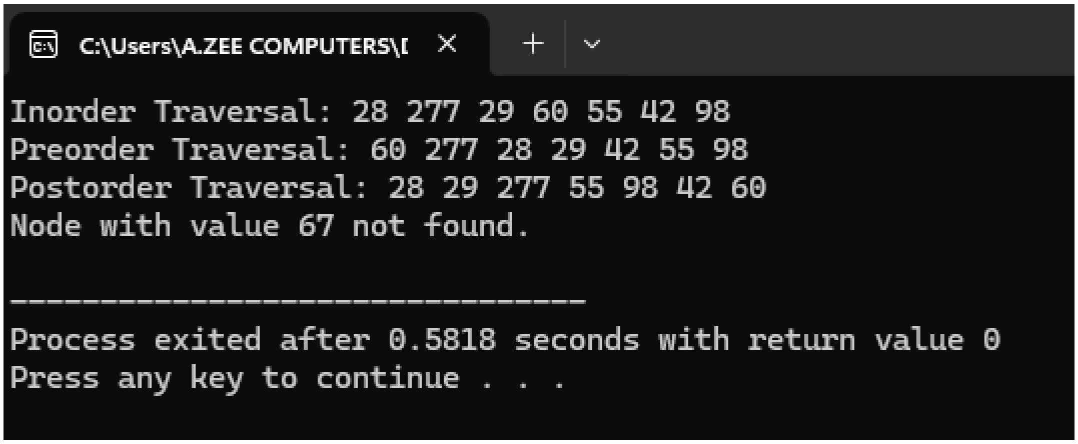
```
    void add(int n) {
        Node* newNode = new Node(n);
        if (root == nullptr) {
            root = newNode;
            return;
        }
  // BFS for level order insertion
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node* temp = q.front();
            q.pop();

            if (temp->left == nullptr) {
                temp->left = newNode;
                return;
            } else {
                q.push(temp->left);
            }

            if (temp->right == nullptr) {
                temp->right = newNode;
                return;
            } else {
                q.push(temp->right);
            }
```

```
        }
    }

    bool searchNode(int n) {
        return find(root, n);
    }

    void traverse(string order) {
        if (order == "inorder")
            doInorder(root);
        else if (order == "preorder")
            doPreorder(root);
        else if (order == "postorder")
            doPostorder(root);
        cout << endl;
    }

private:
// inorder traversal
    void doInorder(Node* node) {
        if (node == nullptr) return;
        doInorder(node->left);
        cout << node->value << " ";
        doInorder(node->right);
    }
// preorder traversal
```

```
    void doPreorder(Node* node) {
        if (node == nullptr) return;
        cout << node->value << " ";
        doPreorder(node->left);
        doPreorder(node->right);
    }
// postorder traversal
    void doPostorder(Node* node) {
        if (node == nullptr) return;
        doPostorder(node->left);
        doPostorder(node->right);
        cout << node->value << " ";
    }
// search function
    bool find(Node* node, int val) {
        if (node == nullptr) return false;
        if (node->value == val) return true;
        return find(node->left, val) || find(node->right, val);
    }
};


int main() {
    Tamia_004 T;
      int n = 67;
    T.add(60);
    T.add(277);
```

```
    T.add(42);

    T.add(28);

    T.add(29);

    T.add(55);

    T.add(98);


    cout << "Inorder Traversal: ";

    T.traverse("inorder");


    cout << "Preorder Traversal: ";

    T.traverse("preorder");


    cout << "Postorder Traversal: ";

    T.traverse("postorder");


    if (T.searchNode(n)) {

        cout << "Node with value " << n << " found." << endl;

    } else {

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

    }

    return 0;

}
```

Output



2. Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

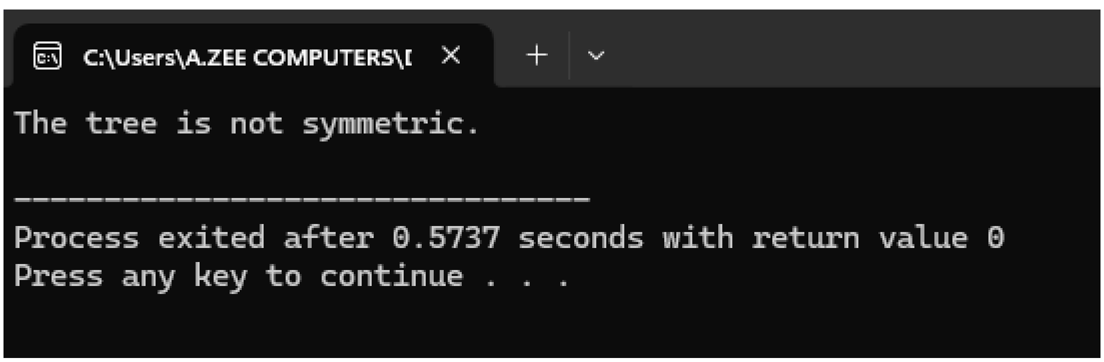
Source code

```
#include <bits/stdc++.h>
using namespace std;
class Node {
public:
   int value;
   Node* left;
   Node* right;
// constructor
   Node(int val) {
      value = val;
      left = right = nullptr;
   }
};
class Tamia_004 {
public:
   Node* root;
   Tamia_004() {
      root = nullptr;
   }

   void add(int n) {
      Node* newNode = new Node(n);
      if (root == nullptr) {
         root = newNode;
         return;
```

```
        }
    // BFS for level order insertion
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node* temp = q.front();
            q.pop();

            if (temp->left == nullptr) {
                temp->left = newNode;
                return;
            } else {
                q.push(temp->left);
            }

            if (temp->right == nullptr) {
                temp->right = newNode;
                return;
            } else {
                q.push(temp->right);
            }
        }
    }

// SYMMETRIC FUNCTION
    bool isMirror(Node* left, Node* right) {
        if (left == nullptr && right == nullptr) return true;
        if (left == nullptr || right == nullptr) return false;
        return (left->value == right->value) &&
               isMirror(left->left, right->right) &&
               isMirror(left->right, right->left);
    }
// EMPTY TREE IS SYMMETRIC
    bool isSymmetric() {
        if (root == nullptr) return true;
        return isMirror(root->left, root->right);
    }
};
```

```
int main() {
  Tamia_004 T;
  T.add(45);
  T.add(145);
  T.add(245);
  T.add(3);
  T.add(245);
  T.add(145);
  T.add(45);
  if (T.isSymmetric()) {
    cout << "The tree is symmetric." << endl;
  } else {
    cout << "The tree is not symmetric." << endl;
  }
  return 0;
}
```

Output



3. Given a binary tree, determine if it is height-balanced i.e., the absolute difference between the left and right subtree of each node is not greater than 1.

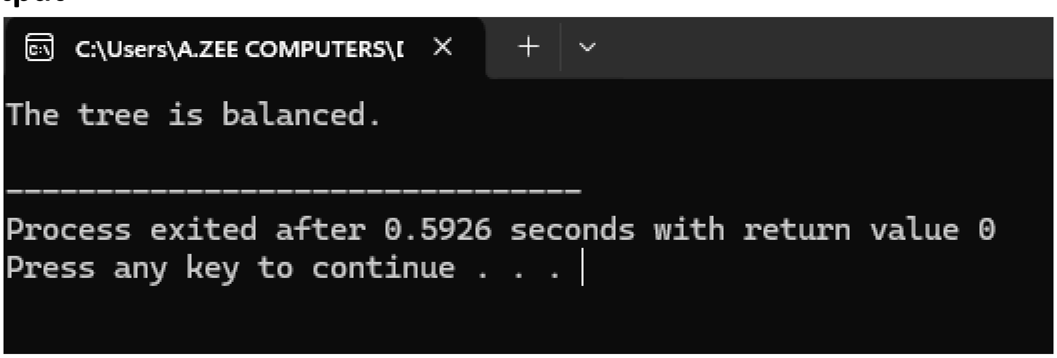
Source code

```
#include <bits/stdc++.h>
using namespace std;
class Node {
public:
  int value;
  Node* left;
  Node* right;
  // constructor
  Node(int val) {
    value = val;
```

```
        left = right = nullptr;
    }
};
class Tamia_004 {
public:
    Node* root;
    Tamia_004() {
        root = nullptr;
    }
    void add(int n) {
        Node* newNode = new Node(n);
        if (root == nullptr) {
            root = newNode;
            return;
        }
// BFS for level order insertion
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node* temp = q.front();
            q.pop();

            if (temp->left == nullptr) {
                temp->left = newNode;
                return;
            } else {
                q.push(temp->left);
            }

            if (temp->right == nullptr) {
                temp->right = newNode;
                return;
            } else {
                q.push(temp->right);
            }
        }
    }
// Function to calculate height
    int height(Node* node) {
        if (node == nullptr) return 0;
        return 1 + max(height(node->left), height(node->right));
    }
```

```
// Function to check if the tree is height-balanced
  bool isBalanced(Node* node) {
    if (node == nullptr) return true;

    int leftHeight = height(node->left);
    int rightHeight = height(node->right);
    return abs(leftHeight - rightHeight) <= 1 && isBalanced(node->left) && isBalanced(node-
>right);
  }
};

int main() {
  Tamia_004 T;
  T.add(45);
  T.add(145);
  T.add(245);
  T.add(3);
  T.add(245);
  T.add(145);
  T.add(45);
  if (T.isBalanced(T.root)) {
    cout << "The tree is balanced." << endl;
  } else {
    cout << "The tree is not balanced." << endl;
  }
  return 0;
}
```

Output



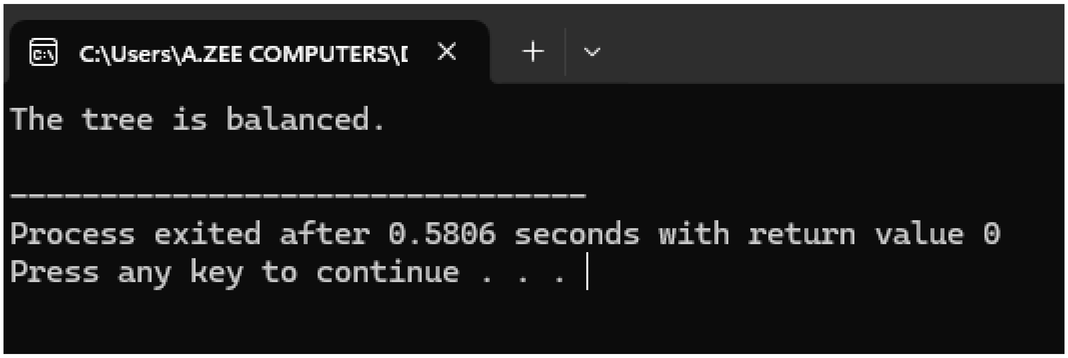
4. Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Source code

```
#include <bits/stdc++.h>
using namespace std;
class Node {
public:
   int value;
   Node* left;
   Node* right;
// Constructor
   Node(int val) {
      value = val;
      left = right = nullptr;
   }
};
class Tamia_004 {
public:
   Node* root;
   Tamia_004() {
      root = nullptr;
   }
// BFS for level order insertion
   void add(int n) {
      Node* newNode = new Node(n);
      if (root == nullptr) {
         root = newNode;
         return;
      }
      queue<Node*> q;
      q.push(root);
      while (!q.empty()) {
         Node* temp = q.front();
         q.pop();

         if (temp->left == nullptr) {
            temp->left = newNode;
            return;
         } else {
            q.push(temp->left);
         }

         if (temp->right == nullptr) {
            temp->right = newNode;
            return;
```

```
            } else {
                q.push(temp->right);
            }
        }
    }

// function to calculate the height of the tree
    int height(Node* node) {
        if (node == nullptr) return 0;
        return 1 + max(height(node->left), height(node->right));
    }
// Function to check if the tree is balanced
    bool isBalanced(Node* node) {
        if (node == nullptr) return true;

        int leftHeight = height(node->left);
        int rightHeight = height(node->right);
        return abs(leftHeight - rightHeight) <= 1 && isBalanced(node->left) && isBalanced(node-
>right);
    }
};

int main() {
    Tamia_004 T;
    T.add(45);
    T.add(145);
    T.add(245);
    T.add(3);
    T.add(245);
    T.add(145);
    T.add(45);
    if (T.isBalanced(T.root)) {
        cout << "The tree is balanced." << endl;
    } else {
        cout << "The tree is not balanced." << endl;
    }
    return 0;
}
```

Output



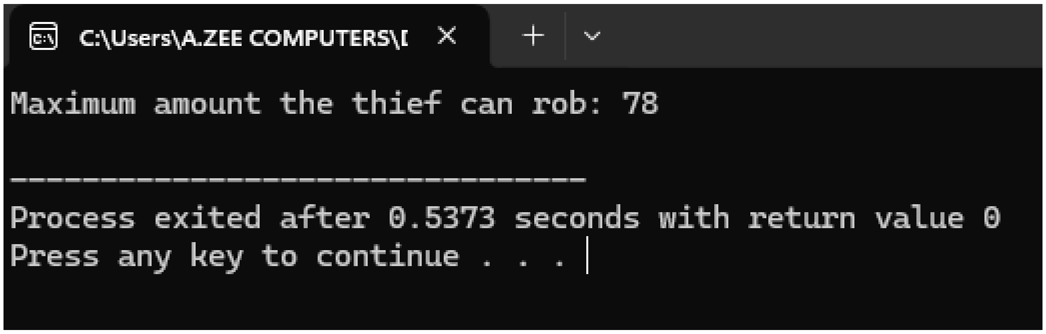
5. The thief has found himself a new place for his thievery again. There is only one entrance to this area, called root. Besides the root, each house has one and only one parent house. After a tour, the smart thief realized that all houses in this place form a binary tree. It will automatically contact the police if two directly-linked houses were broken into on the same night. Given the root of the binary tree, return the maximum amount of money the thief can rob without alerting the police.

Source code

```
#include <iostream>
using namespace std;
class Node {
public:
    int value;
    Node* left;
    Node* right;
    int includeRob;
    int excludeRob;
// Constructor
    Node(int val) {
        value = val;
        left = right = nullptr;
        includeRob = excludeRob = -1;
    }
};
class Tamia_004 {
public:
    Node* root;
    Tamia_004() : root(nullptr) {}
// Method to calculate the maximum amount of money
    int maxRobAmount(Node* node) {
        return robberyHelper(node);
    }

private:
```

```
// Helper function to calculate the max value based on the choice of robbing or not robbing a
node
  int robberyHelper(Node* node) {
    if (!node) return 0;

  // If this node's result has already been calculated, use it
    if (node->includeRob != -1) {
      return node->includeRob;
    }

  // Option 1: Rob this node, skip the immediate children
    int robCurrent = node->value;
    if (node->left) {
      robCurrent += robberyHelper(node->left->left) + robberyHelper(node->left->right);
    }
    if (node->right) {
      robCurrent += robberyHelper(node->right->left) + robberyHelper(node->right->right);
    }

  // Option 2: Do not rob this node, consider the immediate children
    int notRobCurrent = robberyHelper(node->left) + robberyHelper(node->right);

  // Choose the better option between robbing this node or not
    int result = max(robCurrent, notRobCurrent);

  // Store the result in the appropriate field
    node->includeRob = result;
    return result;
  }
};

int main() {
  Tamia_004 T;
  T.root = new Node(30);
  T.root->left = new Node(22);
  T.root->right = new Node(35);
  T.root->left->right = new Node(31);
  T.root->right->right = new Node(17);
  cout << "Maximum amount the thief can rob: " << T.maxRobAmount(T.root) << endl;
  return 0;
}
```

Output


C:\Users\A.ZEE COMPUTERS\[
Maximum amount the thief can rob: 78

Process exited after 0.5373 seconds with return value 0
Press any key to continue . . .