Data Structures and Algorithms Assignment

Submitted to: Mam Irsha

Submitted by: Muhammad Faisal Kamran

Roll Number: 2023-BS-AI-025

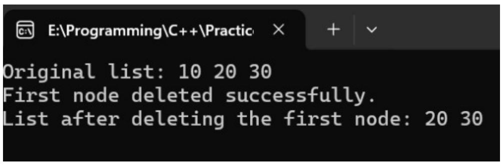
Section: A

1. Doubly Linked List Codes

- **Deletion at First**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* prev;
    Node* next;
    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};
void deleteFirstNode(Node*& head) {
    if (head == nullptr) {
        cout << "The list is already empty." << endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    if (head != nullptr) {
        head->prev = nullptr;
    }
    delete temp;
    cout << "First node deleted successfully." << endl;
}
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

Output



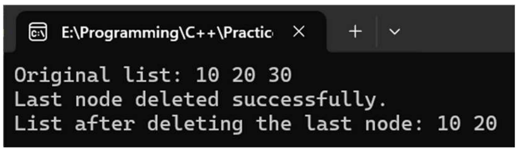
```
Original list: 10 20 30
First node deleted successfully.
List after deleting the first node: 20 30
```

```
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);

    if (head == nullptr) { // If the list is empty
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}
int main() {
    Node* head = nullptr;
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    cout << "Original list: ";
    displayList(head);
    deleteFirstNode(head);
    cout << "List after deleting the first node: ";
    displayList(head);
    return 0;
}
```

- **Deletion at Last**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* prev;
    Node* next;
    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};
void deleteLastNode(Node*& head) {
    if (head == nullptr) {
        cout << "The list is already empty." << endl;
        return;
    }
```

Output



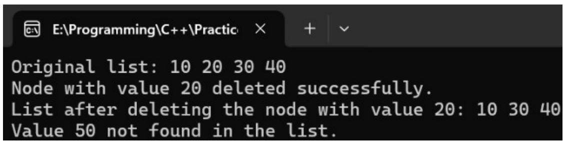
```
Original list: 10 20 30
Last node deleted successfully.
List after deleting the last node: 10 20
```

```
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        cout << "Last node deleted successfully." << endl;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->prev->next = nullptr;
    delete temp;
    cout << "Last node deleted successfully." << endl;
}
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
int main() {
    Node* head = nullptr;
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    cout << "Original list: ";
    displayList(head);
    deleteLastNode(head);
    cout << "List after deleting the last node: ";
    displayList(head);
    return 0;
}
```

- ## Deletion by value

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* prev;
    Node* next;
    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};
void deleteNodeByValue(Node*& head, int value) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr && temp->data != value) {
        temp = temp->next;
    }
    if (temp == nullptr) {
        cout << "Value " << value << " not found in the list." << endl;
        return;
    }
    if (temp == head) {
        head = head->next;
```

Output



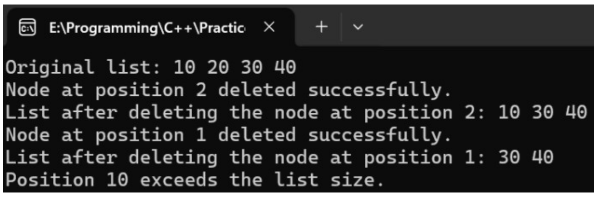
```
Original list: 10 20 30 40
Node with value 20 deleted successfully.
List after deleting the node with value 20: 10 30 40
Value 50 not found in the list.
```

```
        if (head != nullptr) {
            head->prev = nullptr;
        }
    } else if (temp->next == nullptr) {
        temp->prev->next = nullptr;
    } else {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }
    delete temp;
    cout << "Node with value " << value << " deleted successfully." << endl;
}
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
int main() {
    Node* head = nullptr;
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);
    cout << "Original list: ";
    displayList(head);
    deleteNodeByValue(head, 20);
    cout << "List after deleting the node with value 20: ";
    displayList(head);
    deleteNodeByValue(head, 50);
    return 0;
}
```

- **Deletion at position**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
    if (position <= 0) {
```

Output



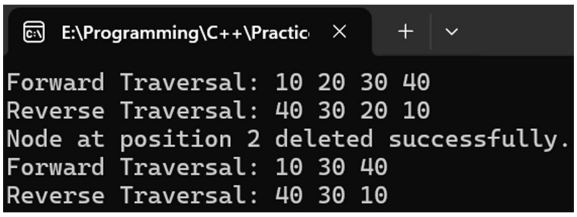
```
Original list: 10 20 30 40
Node at position 2 deleted successfully.
List after deleting the node at position 2: 10 30 40
Node at position 1 deleted successfully.
List after deleting the node at position 1: 30 40
Position 10 exceeds the list size.
```

```
        cout << "Invalid position. Position should be greater than 0." << endl;
        return;
    }
    Node* temp = head;
    int currentIndex = 1;
    while (temp != nullptr && currentIndex < position) {
        temp = temp->next;
        currentIndex++;
    }

    if (temp == nullptr) {
        cout << "Position " << position << " exceeds the list size." << endl;
        return;
    }
    if (temp == head) {
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
    } else if (temp->next == nullptr) {
        temp->prev->next = nullptr;
    } else {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }
    delete temp;
    cout << "Node at position " << position << " deleted successfully." << endl;
}
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}
int main() {
    Node* head = nullptr;
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);
    cout << "Original list: ";
    displayList(head);
    deleteNodeAtPosition(head, 2);
    cout << "List after deleting the node at position 2: ";
    displayList(head);
    deleteNodeAtPosition(head, 1);
    cout << "List after deleting the node at position 1: ";
    displayList(head);
    deleteNodeAtPosition(head, 10);
    return 0;
}
```

- ## Forward and Reverse Traversal

```
#include <iostream>

using namespace std;

struct Node {

    int data;

    Node* prev;

    Node* next;

    Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

void forwardTraversal(Node* head) {

    cout << "Forward Traversal: ";

    Node* temp = head;

    while (temp != nullptr) {
```

Output



```
Forward Traversal: 10 20 30 40
Reverse Traversal: 40 30 20 10
Node at position 2 deleted successfully.
Forward Traversal: 10 30 40
Reverse Traversal: 40 30 10
```

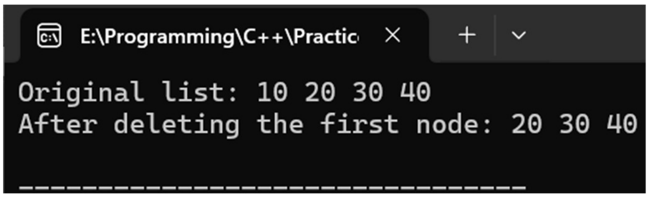
```
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
void reverseTraversal(Node* head) {
    if (head == nullptr) {
        cout << "Reverse Traversal: List is empty." << endl;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    cout << "Reverse Traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}
void appendNode(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) { /
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
    if (position <= 0) {
        cout << "Invalid position. Position should be greater than 0." << endl;
        return;
    }
    Node* temp = head;
    int currentIndex = 1;
    while (temp != nullptr && currentIndex < position) {
        temp = temp->next;
        currentIndex++;
    }

    if (temp == nullptr) {
        cout << "Position " << position << " exceeds the list size." << endl;
        return;
    }
    if (temp == head) {
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
    } else if (temp->next == nullptr) {
        temp->prev->next = nullptr;
    } else {
        temp->prev->next = temp->next;
        temp->next->prev = temp->prev;
    }
    delete temp;
    cout << "Node at position " << position << " deleted successfully." << endl;
}
int main() {
    Node* head = nullptr;
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);
    forwardTraversal(head);
    reverseTraversal(head);
    deleteNodeAtPosition(head, 2);
    forwardTraversal(head);
    reverseTraversal(head);
    return 0;
}
```

2. Circular Linked List Codes

• Deletion at First

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
void deleteFirstNode(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }
    if (head->next == head) {
        delete head;
        head = nullptr;
        return;
    }
    Node* last = head;
    while (last->next != head) {
        last = last->next;
    }
    Node* temp = head;
    head = head->next;
    last->next = head;
    delete temp;
}
void insert(Node*& head, int data) {
    Node* newNode = new Node();
```

Output



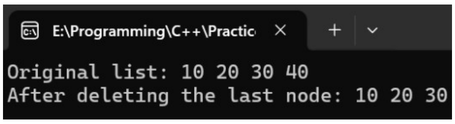
```
Original list: 10 20 30 40
After deleting the first node: 20 30 40

--------------------------------
```

```
    newNode->data = data;
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
void display(Node* head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
int main() {
    Node* head = nullptr;
    insert(head, 10);
    insert(head, 20);
    insert(head, 30);
    insert(head, 40);
    cout << "Original list: ";
    display(head);
    deleteFirstNode(head);
    cout << "After deleting the first node: ";
    display(head);
    return 0;
}
```

- ## Deletion at Last

```
#include <iostream>using namespace std;
struct Node {
    int data;
    Node* next;
};
void deleteLastNode(Node*& head) {
    if (head == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }
    if (head->next == head) {
        delete head;
        head = nullptr;
        return;
    }
    Node* current = head;
    while (current->next->next != head) {
        current = current->next;
```

Output



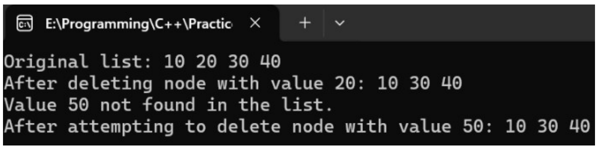
```
Original list: 10 20 30 40
After deleting the last node: 10 20 30
```

```
    }
    Node* last = current->next;
    current->next = head;
    delete last;
}
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
void display(Node* head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
int main() {
    Node* head = nullptr;
    insert(head, 10);
    insert(head, 20);
    insert(head, 30);
    insert(head, 40);
    cout << "Original list: ";
    display(head);
    deleteLastNode(head);
    cout << "After deleting the last node: ";
    display(head);
    return 0;
}
```

- **Deletion at value**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
void deleteNodeByValue(Node*& head, int value) {
    if (head == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }
```

Output



```
Original list: 10 20 30 40
After deleting node with value 20: 10 30 40
Value 50 not found in the list.
After attempting to delete node with value 50: 10 30 40
```

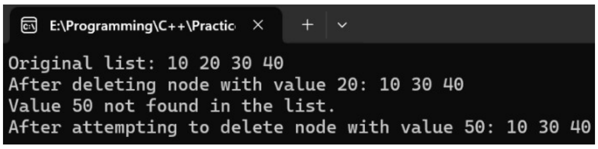
```
    Node* current = head;
    Node* previous = nullptr;
    if (head->data == value && head->next == head) {
        delete head;
        head = nullptr;
        return;
    }
    if (head->data == value) {
        while (current->next != head) {
            current = current->next;
        }
        Node* temp = head;
        head = head->next;
        current->next = head;
        delete temp;
        return;
    }
    do {
        previous = current;
        current = current->next;

        if (current->data == value) {
            previous->next = current->next;
            delete current;
            return;
        }
    } while (current != head);
    cout << "Value " << value << " not found in the list." << endl;
}
```

```
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
void display(Node* head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
int main() {
    Node* head = nullptr;
    insert(head, 10);
    insert(head, 20);
    insert(head, 30);
    insert(head, 40);
    cout << "Original list: ";
    display(head);
    deleteNodeByValue(head, 20);
    cout << "After deleting node with value 20: ";
    display(head);
    deleteNodeByValue(head, 50);
    cout << "After attempting to delete node with value 50: ";
    display(head);
    return 0;
}
```

- ## Deletion at value

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
void deleteNodeByValue(Node*& head, int value) {
    if (head == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }
    Node* current = head;
    Node* previous = nullptr;
```

Output



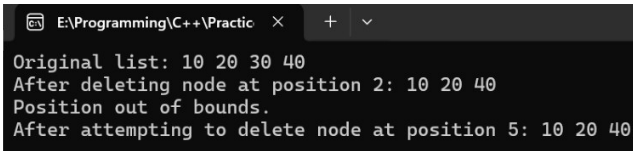
```
Original list: 10 20 30 40
After deleting node with value 20: 10 30 40
Value 50 not found in the list.
After attempting to delete node with value 50: 10 30 40
```

```
    if (head->data == value && head->next == head) {
        delete head;
        head = nullptr;
        return;
    }
    if (head->data == value) {
        while (current->next != head) {
            current = current->next;
        }
        Node* temp = head;
        head = head->next;
        current->next = head;
        delete temp;
        return;
    }
    do {
        previous = current;
        current = current->next;

        if (current->data == value) {
            previous->next = current->next;
            delete current;
            return;
        }
    } while (current != head);
    cout << "Value " << value << " not found in the list." << endl;
}
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
void display(Node* head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
int main() {
    Node* head = nullptr;
    insert(head, 10);
    insert(head, 20);
    insert(head, 30);
    insert(head, 40);
    cout << "Original list: ";
    display(head);
    deleteNodeByValue(head, 20);
    cout << "After deleting node with value 20: ";
    display(head);
    deleteNodeByValue(head, 50);
    cout << "After attempting to delete node with value 50: ";
    display(head);
    return 0;
}
```

- ## Deletion at Position

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }
    if (position == 0) {
        if (head->next == head) {
            delete head;
```

Output



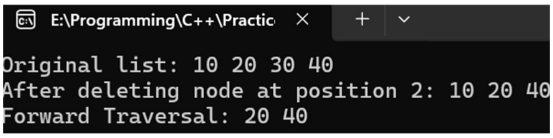
```
Original list: 10 20 30 40
After deleting node at position 2: 10 20 40
Position out of bounds.
After attempting to delete node at position 5: 10 20 40
```

```
}
            head = nullptr;
        } else {
            Node* last = head;
            while (last->next != head) {
                last = last->next;
            }
            Node* temp = head;
            head = head->next;
            last->next = head;
            delete temp;
        }
        return;
    }
    Node* current = head;
    Node* previous = nullptr;
    int count = 0;
    while (current->next != head && count < position) {
        previous = current;
        current = current->next;
        count++;
    }
    if (current->next == head && count < position) {
        cout << "Position out of bounds." << endl;
        return;
    }
    previous->next = current->next;
    delete current;
}
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
void display(Node* head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
int main() {
    Node* head = nullptr;
    insert(head, 10);
    insert(head, 20);

    insert(head, 30);

    insert(head, 40);

    cout << "Original list: ";

    display(head);

    deleteNodeAtPosition(head, 2);

    cout << "After deleting node at position 2: ";

    display(head);

    deleteNodeAtPosition(head, 5);

    cout << "After attempting to delete node at position 5: ";

    display(head);

    return 0;

}
```

- **Forward Traversal**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) { // List is empty
        cout << "List is empty. Nothing to delete." << endl;
        return;
    }
    if (position == 0) {
        if (head->next == head) {
            delete head;
            head = nullptr;
        } else {
            Node* last = head;
            while (last->next != head) {
                last = last->next;
            }
            Node* temp = head;
            head = head->next;
```

Output



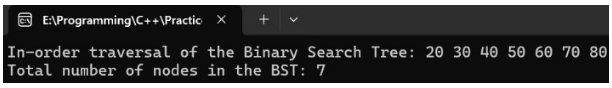
```
Original list: 10 20 30 40
After deleting node at position 2: 10 20 40
Forward Traversal: 20 40
```

```
            last->next = head;
            delete temp;
        }
        return;
    }
    Node* current = head;
    Node* previous = nullptr;
    int count = 0;
    while (current->next != head && count < position) {
        previous = current;
        current = current->next;
        count++;
    }
    if (current->next == head && count < position) {
        cout << "Position out of bounds." << endl;
        return;
    }
    previous->next = current->next;
    delete current;
}
void insert(Node*& head, int data) {
    Node* newNode = new Node();
    newNode->data = data;
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->next = head;
}
void display(Node* head) {
    if (head == nullptr) {
        cout << "List is empty." << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
int main() {
    Node* head = nullptr;
    insert(head, 10);
    insert(head, 20);
    insert(head, 30);
    insert(head, 40);
    cout << "Original list: ";
    display(head);
    deleteNodeAtPosition(head, 2);
    cout << "After deleting node at position 2: ";
    display(head);
    deleteNodeAtPosition(head, 0);
    cout << "Forward Traversal: ";
    display(head);
    return 0;
}
```

3. Binary Search Tree Codes

- **Count Number of nodes**

Output



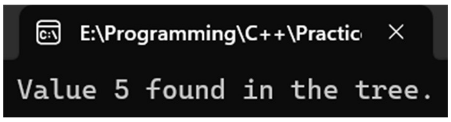
```
In-order traversal of the Binary Search Tree: 20 30 40 50 60 70 80
Total number of nodes in the BST: 7
```

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}
int countNodes(Node* root) {
    if (root == nullptr) {
        return 0;
    }
    return 1 + countNodes(root->left) + countNodes(root->right);
}
void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}
int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
    cout << "In-order traversal of the Binary Search Tree: ";
    inorderTraversal(root);
    cout << endl;
    int nodeCount = countNodes(root);
    cout << "Total number of nodes in the BST: " << nodeCount << endl;
    return 0;
}
```

- ## Searching in BST

```
#include <iostream>
using namespace std; struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
```

Output



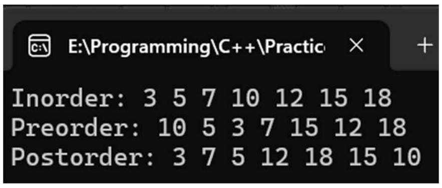
```
Value 5 found in the tree.
```

```
};
bool search(Node* root, int key) {
    if (root == NULL) return false;
    if (root->data == key) return true;
    if (key < root->data) return search(root->left, key);
    return search(root->right, key);
}
int main() {
    Node* root = new Node(10);
    root->left = new Node(5);
    root->right = new Node(15);
    int searchKey = 5;
    if (search(root, searchKey)) {
        cout << "Value " << searchKey << " found in the tree." << endl;
    } else {
        cout << "Value " << searchKey << " not found in the tree." << endl;
    }
    return 0;
}
```

- **Traversing(in-order, pre-order and post-order)**

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
    }
};
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}
void preorder(Node* root) {
    if (root == NULL) return;
    cout << root->data << " ";
```

Output



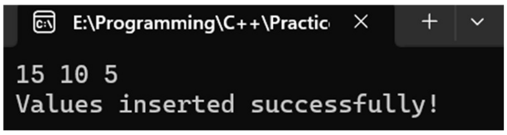
```
Inorder: 3 5 7 10 12 15 18
Preorder: 10 5 3 7 15 12 18
Postorder: 3 7 5 12 18 15 10
```

```
    preorder(root->left);
    preorder(root->right);
}
void postorder(Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}
int main() {
    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(7);
    root->right->left = new Node(12);
    root->right->right = new Node(18);
    cout << "Inorder: ";
    inorder(root);
    cout << endl;
    cout << "Preorder: ";
    preorder(root);
    cout << endl;
    cout << "Postorder: ";
    postorder(root);
    cout << endl;
    return 0;
}
```

- **Reverse in-order**

```
#include <iostream> using namespace std;

struct Node {

    int data;

    Node* left;

    Node* right;

    Node(int val) {

        data = val;

        left = NULL;

        right = NULL;
```

Output



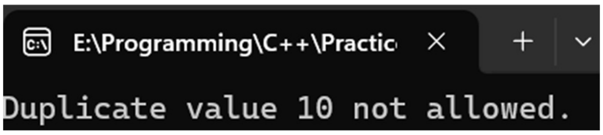
```
15 10 5
Values inserted successfully!
```

```
  }};
void reverseInorder(Node* root) {
  if (root == NULL) return;
  reverseInorder(root->right);
  cout << root->data << " ";
  reverseInorder(root->left);}
Node* insert(Node* root, int val) {
  if (root == NULL) {
    return new Node(val);
  }
  if (val < root->data) {
    root->left = insert(root->left, val);
  } else if (val > root->data) {
    root->right = insert(root->right, val);
  }
  return root;
}
int main() {
  Node* root = NULL;
  root = insert(root, 10);
  root = insert(root, 5);
  root = insert(root, 15);
  reverseInorder(root);
  cout << "\nValues inserted successfully!" << endl;
  return 0;
}
```

- **Duplication in BST**

```
#include <iostream>
using namespace std;
struct Node {
   int data;
   Node* left;
   Node* right;
   Node(int val) {
      data = val;
      left = NULL;
      right = NULL;
   }
};
Node* insert(Node* root, int val) {
   if (root == NULL) {
      return new Node(val);
   }
   if (val < root->data) {
      root->left = insert(root->left, val);
   } else if (val > root->data) {
      root->right = insert(root->right, val);
   } else {
      cout << "Duplicate value " << val << " not allowed." << endl;
   }
   return root;
}
int main() {
   Node* root = NULL;
```

Output

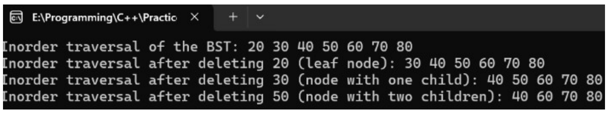


```
Duplicate value 10 not allowed.
```

```
    root = insert(root, 10);
    root = insert(root, 5);
    root = insert(root, 15);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 18);
    return 0;
}
```

- **Deletion in BST**

Output



```
Inorder traversal of the BST: 20 30 40 50 60 70 80
Inorder traversal after deleting 20 (leaf node): 30 40 50 60 70 80
Inorder traversal after deleting 30 (node with one child): 40 50 60 70 80
Inorder traversal after deleting 50 (node with two children): 40 60 70 80
```

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr) {
        current = current->left;
    }
    return current;
}
```

```
Node* deleteNode(Node* root, int key) {
    if (root == nullptr) return root;

    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } 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 = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
Node* insert(Node* root, int key) {
    if (root == nullptr) return new Node(key);

    if (key < root->data)
        root->left = insert(root->left, key);
    else if (key > root->data)
```

```
        root->right = insert(root->right, key);

    return root;
}
void inorder(Node* root) {
    if (root != nullptr) {
        inorder(root->left);
        cout << root->data << " ";
        inorder(root->right);
    }
}
int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);
    cout << "Inorder traversal of the BST: ";
    inorder(root);
    cout << endl;
    root = deleteNode(root, 20);
    cout << "Inorder traversal after deleting 20 (leaf node): ";
    inorder(root);
    cout << endl;
    root = deleteNode(root, 30);
    cout << "Inorder traversal after deleting 30 (node with one child): ";
    inorder(root);

    cout << endl;

    root = deleteNode(root, 50);

    cout << "Inorder traversal after deleting 50 (node with two children): ";

    inorder(root);

    cout << endl;

    return 0;

}
```