NAME : LAIBA FATIMA

REG # : 2023-BS-AI-047

SECTION : AI SECTION A

INSTRUCTER : MISS IRSHA QURESHI

DEPARTMENT OF COMPUTER SCIENCES

FINAL ASSIGNMENT

Doubly Linked List

PROGRAM 1

Statement : Write a program to delete the first node in a doubly linked list.

```
#include <iostream>
using namespace std;
// Node structure for the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) {
        data = val;
        prev = nullptr;
        next = nullptr;
    }
};

// Function to insert a node at the end of the doubly linked list
void insertAtEnd(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;
}

// Function to delete the first node in the doubly linked list
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;
}
```

```
// Function to display the doubly linked list
void displayList(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main()
{
    Node* head = nullptr;
    // Inserting nodes into the doubly linked list
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
    cout << "Original list: ";
    displayList(head);
    // Deleting the first node
    deleteFirstNode(head);
    cout << "List after deleting the first node: ";
    displayList(head);
    return 0;
```

```
}
```

OUTPUT

Original list: 10 20 30

First node deleted successfully.

List after deleting the first node: 20 30

PROGRAM 2

Statement : How can you delete the last node in a doubly linked list? Write the code.

```
#include <iostream>
using namespace std;
// Node structure for the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;
    Node(int val) {
        data = val;
        prev = nullptr;
        next = nullptr;
    }
};
// Function to insert a node at the end of the doubly linked list
void insertAtEnd(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;
}
// Function to delete the first node in the doubly linked list
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;
}
// Function to delete the last node in the doubly linked list
void deleteLastNode(Node*& head) {
    if (head == nullptr) {
        cout << "The list is already empty." << endl;
```

```
        return;
    }
    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;
}
// Function to display the doubly linked list
void displayList(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
```

```
    cout << endl;
}
int main() {
    Node* head = nullptr;
    // Inserting nodes into the doubly linked list
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);

 cout << "Original list: ";
    displayList(head);
    // Deleting the first node
    deleteFirstNode(head);
    cout << "List after deleting the first node: ";
    displayList(head);
    // Deleting the last node
    deleteLastNode(head);
    cout << "List after deleting the last node: ";
    displayList(head);
    return 0;
}
```

OUTPUT

Original list: 10 20 30

First node deleted successfully.

List after deleting the first node: 20 30

Last node deleted successfully.

List after deleting the last node: 20

PROGRAM 3

Statement : Write code to delete a node by its value in a doubly linked list.

```
#include <iostream>

using namespace std;

// Node structure for the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) {
        data = val;
        prev = nullptr;
        next = nullptr;
    }
};

        }

// Function to delete a node by its value in the doubly linked list
void deleteNodeByValue(Node*& head, int value) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
```

```
        return;
    }

    Node* temp = head;

    // Traverse the list to find the node with the given value
    while (temp != nullptr && temp->data != value) {
        temp = temp->next;
    }
// Function to display the doubly linked list
void displayList(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;

    // Inserting nodes into the doubly linked list
```

```
: ";

    // Deleting a node by its value
    deleteNodeByValue(head, 20);

    cout << "List after deleting the node with value 20: ";
    displayList(head);

    deleteNodeByValue(head, 10);

    cout << "List after deleting the node with value 10: ";
    displayList(head);

    deleteNodeByValue(head, 50);

    cout << "Final list: ";
    displayList(head);

    return 0;
}
```

OUTPUT

Original list: 10 20 30 40

Node with value 20 deleted successfully.

List after deleting the node with value 20: 10 30 40

Node with value 10 deleted successfully.

List after deleting the node with value 10: 30 40

Value 50 not found in the list.

Final list: 30 40

PROGRAM 4

Statement : How would you delete a node at a specific position in a doubly linked list?

```
#include <iostream>

using namespace std;

// Node structure for the doubly linked list
struct Node {
    int data;
    Node* prev;
    Node* next;

    Node(int val) {
        data = val;
        prev = nullptr;
        next = nullptr;
    }
};

// Function to insert a node at the end of the doubly linked list
void insertAtEnd(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;
}

// Function to delete a node at a specific position in the doubly linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr || position < 1) {
        cout << "Invalid position or the list is empty." << endl;
        return;
    }

    Node* temp = head;

    // Traverse to the node at the given position
    for (int i = 1; i < position && temp != nullptr; ++i) {
        temp = temp->next;
    }

    // If the position is beyond the list length
    if (temp == nullptr) {
        cout << "Position " << position << " does not exist in the list." << endl;
        return;
```

```
    }

    // If the node to delete is the head
    if (temp == head) {
        head = head->next;
        if (head != nullptr) {
            head->prev = nullptr;
        }
    } else {
        if (temp->next != nullptr) {
            temp->next->prev = temp->prev;
        }
        if (temp->prev != nullptr) {
            temp->prev->next = temp->next;
        }
    }

    delete temp;
    cout << "Node at position " << position << " deleted successfully." << endl;
}

// Function to display the doubly linked list
void displayList(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
```

```
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;

    // Inserting nodes into the doubly linked list
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
    insertAtEnd(head, 40);

    cout << "Original list: ";
    displayList(head);

    // Deleting nodes at specific positions
    deleteNodeAtPosition(head, 2);

    cout << "List after deleting node at position 2: ";
    displayList(head);

    deleteNodeAtPosition(head, 1);
```

```
    cout << "List after deleting node at position 1: ";
    displayList(head);

    deleteNodeAtPosition(head, 5);

    cout << "Final list: ";
    displayList(head);

    return 0;
}
```

OUTPUT

```
Original list: 10 20 30 40
Node at position 2 deleted successfully.
List after deleting node at position 2: 10 30 40
Node at position 1 deleted successfully.
List after deleting node at position 1: 30 40
Position 5 does not exist in the list.
Final list: 30 40
```

PROGRAM 5

Statement : After deleting a node, how will you write the forward and reverse traversal functions

Forward Traversal

```
// Function to traverse the list forward
void forwardTraversal(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
```

```
        return;
    }

    Node* temp = head;
    cout << "Forward traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

Reverse Traversal

```
// Function to traverse the list in reverse
void reverseTraversal(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    // Move to the last node
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    // Traverse backward
    cout << "Reverse traversal: ";
    while (temp != nullptr) {
```

```
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}
```

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
```

Circular Linked List

PROGRAM 1

Statement : Write a program to delete the first node in a circular linked list.

```
#include <iostream>

using namespace std;

// Node structure for the circular linked list
struct Node {
    int data;
    Node* next;
```

```
    Node(int val) {
        data = val;
        next = nullptr;
    }
};

// Function to insert a node at the end of a circular linked list
void insertAtEnd(Node*& tail, int data) {
    Node* newNode = new Node(data);
    if (tail == nullptr) {
        // If the list is empty, initialize with one node pointing to itself
        tail = newNode;
        tail->next = tail;
        return;
    }

    // Insert the new node after the tail and update the tail pointer
    newNode->next = tail->next;
    tail->next = newNode;
    tail = newNode;
}

// Function to delete the first node in a circular linked list
void deleteFirstNode(Node*& tail) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }
```

```
    Node* head = tail->next;

    // If there's only one node in the list
    if (tail == head) {
        delete tail;
        tail = nullptr;
        cout << "First node deleted successfully. The list is now empty." << endl;
        return;
    }

    // For a list with more than one node
    tail->next = head->next; // Update tail's next to point to the second node
    delete head;             // Delete the first node
    cout << "First node deleted successfully." << endl;
}

// Function to display the circular linked list
void displayList(Node* tail) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = tail->next; // Start from the head (next of tail)
    cout << "Circular list: ";
    do {
        cout << temp->data << " ";
```

```
        temp = temp->next;
    } while (temp != tail->next); // Loop until back to the head
    cout << endl;
}

int main() {
    Node* tail = nullptr;

    // Inserting nodes into the circular linked list
    insertAtEnd(tail, 10);
    insertAtEnd(tail, 20);
    insertAtEnd(tail, 30);
    insertAtEnd(tail, 40);

    cout << "Original list: ";
    displayList(tail);

    // Deleting the first node
    deleteFirstNode(tail);

    cout << "List after deleting the first node: ";
    displayList(tail);

    // Deleting the first node again
    deleteFirstNode(tail);

    cout << "List after deleting the first node again: ";
    displayList(tail);
```

```
    return 0;
}
```

OUTPUT

Original list: Circular list: 10 20 30 40

First node deleted successfully.

List after deleting the first node: Circular list: 20 30 40

First node deleted successfully.

List after deleting the first node again: Circular list: 30 40

PROGRAM 2

Statement : **How can you delete the last node in a circular linked list? Write the code.**

```
#include <iostream>

using namespace std;

// Node structure for the circular linked list
struct Node {
    int data;
    Node* next;

    Node(int val) {
        data = val;
        next = nullptr;
    }
```

```
};

// Function to insert a node at the end of a circular linked list
void insertAtEnd(Node*& tail, int data) {
    Node* newNode = new Node(data);
    if (tail == nullptr) {
        // If the list is empty, initialize with one node pointing to itself
        tail = newNode;
        tail->next = tail;
        return;
    }

    // Insert the new node after the tail and update the tail pointer
    newNode->next = tail->next;
    tail->next = newNode;
    tail = newNode;
}

// Function to delete the last node in a circular linked list
void deleteLastNode(Node*& tail) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* head = tail->next;

    // If there's only one node in the list
```

```
    if (tail == head) {
        delete tail;
        tail = nullptr;
        cout << "Last node deleted successfully. The list is now empty." << endl;
        return;
    }

    // Traverse to the second last node
    Node* temp = head;
    while (temp->next != tail) {
        temp = temp->next;
    }

    // Update the second last node to point to the head
    temp->next = tail->next;
    delete tail;  // Delete the last node
    tail = temp;  // Update the tail pointer
    cout << "Last node deleted successfully." << endl;
}

// Function to display the circular linked list
void displayList(Node* tail) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = tail->next; // Start from the head (next of tail)
```

```
    cout << "Circular list: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != tail->next); // Loop until back to the head
    cout << endl;
}

int main() {
    Node* tail = nullptr;

    // Inserting nodes into the circular linked list
    insertAtEnd(tail, 10);
    insertAtEnd(tail, 20);
    insertAtEnd(tail, 30);
    insertAtEnd(tail, 40);

    cout << "Original list: ";
    displayList(tail);

    // Deleting the last node
    deleteLastNode(tail);

    cout << "List after deleting the last node: ";
    displayList(tail);

    // Deleting the last node again
    deleteLastNode(tail);
```

```
    cout << "List after deleting the last node again: ";
    displayList(tail);

    return 0;
}
```

OUTPUT

Original list: Circular list: 10 20 30 40

Last node deleted successfully.

List after deleting the last node: Circular list: 10 20 30

Last node deleted successfully.

List after deleting the last node again: Circular list: 10 20

PROGRAM 3

Statement : Write a function to delete a node by its value in a circular linked list.

```
#include <iostream>

using namespace std;

// Node structure for the circular linked list
struct Node {
    int data;
    Node* next;

    Node(int val) {
        data = val;
```

```
        next = nullptr;
    }
};

// Function to insert a node at the end of a circular linked list
void insertAtEnd(Node*& tail, int data) {
    Node* newNode = new Node(data);
    if (tail == nullptr) {
        // If the list is empty, initialize with one node pointing to itself
        tail = newNode;
        tail->next = tail;
        return;
    }

    // Insert the new node after the tail and update the tail pointer
    newNode->next = tail->next;
    tail->next = newNode;
    tail = newNode;
}

// Function to delete a node by its value in a circular linked list
void deleteNodeByValue(Node*& tail, int value) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* curr = tail->next;
```

```
Node* prev = tail;

// If the list contains only one node
if (tail == tail->next && tail->data == value) {
    delete tail;
    tail = nullptr;
    cout << "Node with value " << value << " deleted. The list is now empty." << endl;
    return;
}

// Traverse the list to find the node with the given value
do {
    if (curr->data == value) {
        // If the node to be deleted is found
        prev->next = curr->next;
        if (curr == tail) {
            // If the node to be deleted is the tail, update the tail pointer
            tail = prev;
        }
        delete curr;
        cout << "Node with value " << value << " deleted successfully." << endl;
        return;
    }
    prev = curr;
    curr = curr->next;
} while (curr != tail->next);

// If the node with the value is not found
```

```
    cout << "Node with value " << value << " not found in the list." << endl;
}

// Function to display the circular linked list
void displayList(Node* tail) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = tail->next; // Start from the head (next of tail)
    cout << "Circular list: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != tail->next); // Loop until back to the head
    cout << endl;
}

int main() {
    Node* tail = nullptr;

    // Inserting nodes into the circular linked list
    insertAtEnd(tail, 10);
    insertAtEnd(tail, 20);
    insertAtEnd(tail, 30);
    insertAtEnd(tail, 40);
```

```
    cout << "Original list: ";
    displayList(tail);

    // Deleting a node by its value
    deleteNodeByValue(tail, 20);

    cout << "List after deleting the node with value 20: ";
    displayList(tail);

    deleteNodeByValue(tail, 10);

    cout << "List after deleting the node with value 10: ";
    displayList(tail);
    deleteNodeByValue(tail, 50);

    cout << "Final list: ";
    displayList(tail);

    return 0;
}
```

OUTPUT

Original list: Circular list: 10 20 30 40

Node with value 20 deleted successfully.

List after deleting the node with value 20: Circular list: 10 30 40

Node with value 10 deleted successfully.

List after deleting the node with value 10: Circular list: 30 40

Node with value 50 not found in the list.

Final list: Circular list: 30 40

PROGRAM 4

Statement : **How will you delete a node at a specific position in a circular linked list? Write code for it.**

```
#include <iostream>

using namespace std;

// Node structure for the circular linked list
struct Node {
    int data;
    Node* next;

    Node(int val) {
        data = val;
        next = nullptr;
    }
};

// Function to delete a node at a specific position in a circular linked list
void deleteNodeAtPosition(Node*& tail, int position) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* curr = tail->next; // Head node
    Node* prev = tail;
```

```
// If the list contains only one node
if (tail == tail->next && position == 1) {
    delete tail;
    tail = nullptr;
    cout << "Node at position " << position << " deleted. The list is now empty." << endl;
    return; }

// Traverse to the node at the specified position
int count = 1;
while (count < position && curr != tail) {
    prev = curr;
    curr = curr->next;
    count++;
}

// If the position is invalid (greater than the number of nodes)
if (count < position || curr == tail->next) {
    cout << "Invalid position. No node deleted." << endl;
    return;
}

// If the node to be deleted is the tail
if (curr == tail) {
    tail = prev; // Update the tail pointer
}

prev->next = curr->next; // Bypass the node to be deleted
```

```
    delete curr;
    cout << "Node at position " << position << " deleted successfully." << endl;
}

// Function to display the circular linked list
void displayList(Node* tail) {
    if (tail == nullptr) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* temp = tail->next; // Start from the head (next of tail)
    cout << "Circular list: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != tail->next); // Loop until back to the head
    cout << endl;
}

int main() {
    Node* tail = nullptr;

    // Inserting nodes into the circular linked list
    insertAtEnd(tail, 10);
    insertAtEnd(tail, 20);
    insertAtEnd(tail, 30);
    insertAtEnd(tail, 40);
```

```
    cout << "Original list: ";
    displayList(tail);

    // Deleting a node at specific positions
    deleteNodeAtPosition(tail, 2);

    cout << "List after deleting the node at position 2: ";
    displayList(tail);

    deleteNodeAtPosition(tail, 1);

    cout << "List after deleting the node at position 1: ";
    displayList(tail);
    deleteNodeAtPosition(tail, 5);

    cout << "Final list: ";
    displayList(tail);

    return 0;
```

OUTPUT

Original list: Circular list: 10 20 30 40

Node at position 2 deleted successfully.

List after deleting the node at position 2: Circular list: 10 30 40

Node at position 1 deleted successfully.

List after deleting the node at position 1: Circular list: 30 40

PROGRAM 5

Statement : **Write a program to show forward traversal after deleting a node in a circular linked list.**

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

struct Node {
    int data;
    Node* next;
};

// Function to add a node to the circular linked list
void append(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        head = newNode;
        newNode->next = head;  // Circular reference
    } else {
        Node* temp = head;
        // Traverse to the last node
        while (temp->next != head) {
            temp = temp->next;
        }
```

```
        temp->next = newNode;
        newNode->next = head;  // Circular reference
    }
}

// Function to delete a node by value
void deleteNode(Node*& head, int value) {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    Node* temp = head;
    Node* prev = nullptr;

    // If the node to delete is the head node
    if (head->data == value) {
        if (head->next == head) {  // Only one node in the list
            delete head;
            head = nullptr;
            return;
        }

        // Traverse to the last node
        while (temp->next != head) {
            temp = temp->next;
        }
```

```
        // Update head and delete the old head
        temp->next = head->next;
        Node* oldHead = head;
        head = head->next;
        delete oldHead;
        return;
    }

    // Traverse to find the node to delete
    while (temp->next != head && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    // If node is not found
    if (temp->data != value) {
        cout << "Node not found!" << endl;
        return;
    }

    // Delete the node
    prev->next = temp->next;
    delete temp;
}

// Function to traverse the list
void traverse(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;

    // Append nodes to the list
    append(head, 10);
    append(head, 20);
    append(head, 30);
    append(head, 40);

    cout << "Original List: ";
    traverse(head);

    // Delete a node
    int valueToDelete = 20;
    cout << "Deleting node with value " << valueToDelete << endl;
```

```
    deleteNode(head, valueToDelete);

    // Traverse the list after deletion
    cout << "List after deletion: ";
    traverse(head);

    return 0;
}
```

OUTPUT

Original List: 10 20 30 40

Deleting node with value 20

List after deletion: 10 30 40

Binary Search Tree

PROGRAM 1

Statement : **Write a program to count all the nodes in a binary search tree.**

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

// Structure to represent a node in the binary search tree
struct Node {
    int data;
    Node* left;
```

```
    Node* right;

    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a new node into the binary search tree
Node* insert(Node* root, int value) {
    // If the tree is empty, create a new node
    if (root == nullptr) {
        return new Node(value);
    }

    // Otherwise, recur down the tree
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }

    return root;
}

// Function to count the nodes in the binary search tree
```

```
int countNodes(Node* root) {
    // Base case: if the tree is empty, return 0
    if (root == nullptr) {
        return 0;
    }

    // Recur for left and right subtrees and add 1 for the current node
    return 1 + countNodes(root->left) + countNodes(root->right);
}

int main() {
    Node* root = nullptr;

    // Inserting nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    // Counting the nodes in the binary search tree
    int nodeCount = countNodes(root);

    // Printing the result
    cout << "Total number of nodes in the BST: " << nodeCount << endl;
```

```
    return 0;
}
```

OUTPUT

Total number of nodes in the BST: 7

PROGRAM 2

Statement : How can you search for a specific value in a binary search tree? Write the code.

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

// Structure to represent a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};
```

```
// Function to insert a new node into the binary search tree
Node* insert(Node* root, int value) {
    // If the tree is empty, create a new node
    if (root == nullptr) {
        return new Node(value);
    }

    // Otherwise, recur down the tree
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }

    return root;
}

// Function to search for a value in the binary search tree
bool search(Node* root, int value) {
    // Base case: if the tree is empty or the value is found
    if (root == nullptr) {
        return false;  // Value not found
    }

    // If the value is found at the current node
    if (root->data == value) {
        return true;
    }
```

```
    // If the value is smaller than the current node's data, search in the left subtree
    if (value < root->data) {
        return search(root->left, value);
    }

    // If the value is greater than the current node's data, search in the right subtree
    return search(root->right, value);
}

int main() {
    Node* root = nullptr;

    // Inserting nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    // Searching for a value in the BST
    int valueToSearch = 40;
    if (search(root, valueToSearch)) {
        cout << "Value " << valueToSearch << " found in the BST." << endl;
    } else {
        cout << "Value " << valueToSearch << " not found in the BST." << endl;
```

```
    }

    return 0;
}
```

OUTPUT

Value 40 found in the BST.

PROGRAM 3

Statement : Write code to traverse a binary search tree in in-order, pre-order, and post order.

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

// Structure to represent a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};
```

```
// Function to insert a new node into the binary search tree
Node* insert(Node* root, int value) {
    // If the tree is empty, create a new node
    if (root == nullptr) {
        return new Node(value);
    }

    // Otherwise, recur down the tree
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }

    return root;
}

// In-order Traversal (Left, Root, Right)
void inOrder(Node* root) {
    if (root == nullptr) {
        return;
    }

    // Traverse the left subtree
    inOrder(root->left);

    // Visit the current node (root)
```

```
    cout << root->data << " ";

    // Traverse the right subtree
    inOrder(root->right);
}

// Pre-order Traversal (Root, Left, Right)
void preOrder(Node* root) {
    if (root == nullptr) {
        return;
    }

    // Visit the current node (root)
    cout << root->data << " ";

    // Traverse the left subtree
    preOrder(root->left);

    // Traverse the right subtree
    preOrder(root->right);
}

// Post-order Traversal (Left, Right, Root)
void postOrder(Node* root) {
    if (root == nullptr) {
        return;
    }
```

```
    // Traverse the left subtree
    postOrder(root->left);

    // Traverse the right subtree
    postOrder(root->right);

    // Visit the current node (root)
    cout << root->data << " ";
}

int main() {
    Node* root = nullptr;

    // Inserting nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    // Traversing the binary search tree
    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;

    cout << "Pre-order Traversal: ";
```

```
    preOrder(root);
    cout << endl;

    cout << "Post-order Traversal: ";
    postOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT

In-order Traversal: 20 30 40 50 60 70 80

Pre-order Traversal: 50 30 20 40 70 60 80

Post-order Traversal: 20 40 30 60 80 70 50

PROGRAM 4

Statement : How will you write reverse in-order traversal for a binary search tree? Show it in code.

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

// Structure to represent a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;
```

```
    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a new node into the binary search tree
Node* insert(Node* root, int value) {
    // If the tree is empty, create a new node
    if (root == nullptr) {
        return new Node(value);
    }

    // Otherwise, recur down the tree
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else {
        root->right = insert(root->right, value);
    }

    return root;
}

// Reverse in-order Traversal (Right, Root, Left)
void reverseInOrder(Node* root) {
```

```
    if (root == nullptr) {
        return;
    }

    // Traverse the right subtree first
    reverseInOrder(root->right);

    // Visit the current node (root)
    cout << root->data << " ";

    // Then traverse the left subtree
    reverseInOrder(root->left);
}

int main() {
    Node* root = nullptr;

    // Inserting nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    // Reverse In-order Traversal
    cout << "Reverse In-order Traversal: ";
```

```
    reverseInOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT

Reverse In-order Traversal: 80 70 60 50 40 30 20

PROGRAM 5

Statement : Write a program to check if there are duplicate values in a binary search tree.

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

// Structure to represent a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = nullptr;
```

```
        right = nullptr;
    }
};

// Function to insert a new node into the binary search tree
Node* insert(Node* root, int value) {
    // If the tree is empty, create a new node
    if (root == nullptr) {
        return new Node(value);
    }

    // Otherwise, recur down the tree
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    // If the value is equal to the current node's data, it's a duplicate
    return root;
}

// Function to check for duplicates using in-order traversal
bool checkDuplicates(Node* root, unordered_set<int>& values) {
    if (root == nullptr) {
        return false;
    }

    // Traverse the left subtree
```

```
    if (checkDuplicates(root->left, values)) {
        return true; // Duplicate found in left subtree
    }

    // Check if the value is already in the set (duplicate)
    if (values.find(root->data) != values.end()) {
        return true; // Duplicate found
    }

    // Add the current node's value to the set
    values.insert(root->data);

    // Traverse the right subtree
    return checkDuplicates(root->right, values);
}

int main() {
    Node* root = nullptr;

    // Inserting nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);
```

```
    // Introduce a duplicate value
    root = insert(root, 40);

    // Set to store visited values
    unordered_set<int> values;

    // Check for duplicates
    if (checkDuplicates(root, values)) {
        cout << "The BST contains duplicate values." << endl;
    } else {
        cout << "The BST does not contain duplicate values." << endl;
    }

    return 0;
}
```

OUTPUT

The BST contains duplicate values.

PROGRAM 6

Statement : How can you delete a node from a binary search tree? Write code for deleting a leaf, a node with one child, and a node with two children.

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

```
// Structure to represent a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

    // Constructor to create a new node
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

// Function to insert a new node into the binary search tree
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;
}
```

```
// Function to find the minimum node in the binary search tree
Node* findMin(Node* root) {
    while (root && root->left) {
        root = root->left;
    }
    return root;
}

// Function to delete a node from the binary search tree
Node* deleteNode(Node* root, int value) {
    // If the tree is empty
    if (root == nullptr) {
        return root;
    }

    // Traverse the tree
    if (value < root->data) {
        // If the value to be deleted is smaller than the root's data, go left
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        // If the value to be deleted is larger than the root's data, go right
        root->right = deleteNode(root->right, value);
    } else {
        // Node to be deleted is found

        // Case 1: Node has no children (leaf node)
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
```

```
            return nullptr;
        }

        // Case 2: Node has one child
        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;
        }

        // Case 3: Node has two children
        else {
            // Find the inorder successor (smallest node in the right subtree)
            Node* temp = findMin(root->right);
            root->data = temp->data;  // Copy the inorder successor's value to this node
            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->data);
        }
    }
    return root;
}

// Function to print the tree in-order (for testing)
void inOrder(Node* root) {
```

```
    if (root == nullptr) {
        return;
    }
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(root->right);
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the binary search tree
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    cout << "Original tree (in-order): ";
    inOrder(root);
    cout << endl;

    // Deleting a leaf node (20)
    root = deleteNode(root, 20);
    cout << "After deleting 20 (leaf node): ";
    inOrder(root);
```

```
    cout << endl;

    // Deleting a node with one child (30)
    root = deleteNode(root, 30);
    cout << "After deleting 30 (node with one child): ";
    inOrder(root);
    cout << endl;

    // Deleting a node with two children (70)
    root = deleteNode(root, 70);
    cout << "After deleting 70 (node with two children): ";
    inOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT

Original tree (in-order): 20 30 40 50 60 70 80

After deleting 20 (leaf node): 30 40 50 60 70 80

After deleting 30 (node with one child): 40 50 60 70 80

After deleting 70 (node with two children): 40 50 60 80