Name: Waleed Amjad

Roll No: 2023-BS-Ai-054

Department: AI

Sec: A

Semester: 3rd

Final Assignment

Doubly Linked List

Problem1: Write a program to delete the first node in a doubly linked list.

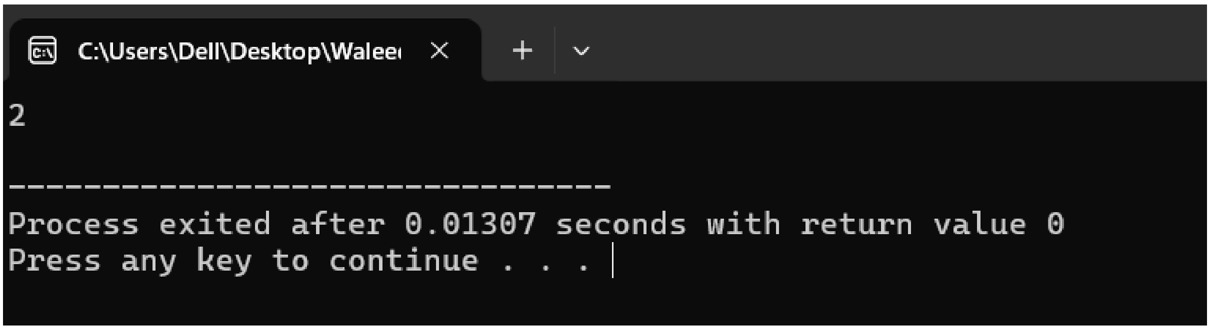
Code:

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

class Node {
public:
    int val;
    Node* next;
    Node* prev;

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

class DOUBLELINKLIST {
public:
    Node* head;
    Node* tail;

    DOUBLELINKLIST() {
        head = NULL;
        tail = NULL;
    }

    void insert(int val) {
```

```
    Node* new_node = new Node(val);
    if (head == NULL) { // If the list is empty
        head = new_node;
        tail = new_node;
    } else {
        tail->next = new_node; // Link the current tail to the new node
        new_node->prev = tail; // Link the new node back to the current tail
        tail = new_node;       // Update the tail to the new node
    }
}


    void deleteAThead() {
        if (head == NULL) { // If the list is empty
            return;
        }
        Node* temp = head;
        head = head->next;
        if (head == NULL) { // If the list becomes empty after deletion
            tail = NULL;
        } else {
            head->prev = NULL;
        }
        delete temp;
    }

    void display() {
        Node* temp = head;
        while (temp != NULL) {
            cout << temp->val;
            if (temp->next != NULL) { // Only add <-> between nodes
                cout << " <-> ";
```

```
            }
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    DOUBLELINKLIST dll;
    dll.insert(3);
    dll.insert(2);
    dll.deleteAThead();
    dll.display();
    return 0;
}
```

Output:



Problem 2: How can you delete the last node in a doubly linked list? Write the code

Code:

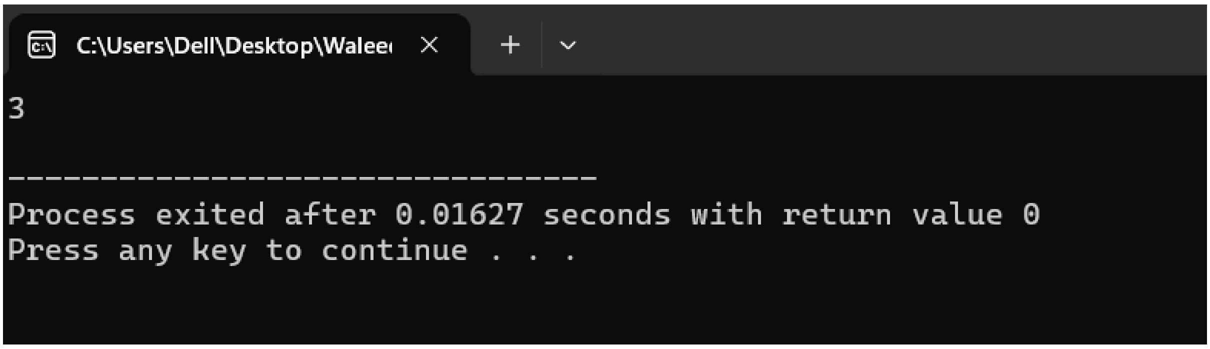
```
#include <iostream>
using namespace std;

class Node {
public:
```

```
    int val;
    Node* next;
    Node* prev;

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

class DOUBLELINKLIST {
public:
    Node* head;
    Node* tail;

    DOUBLELINKLIST() {
        head = NULL;
        tail = NULL;
    }

    void insert(int val) {
    Node* new_node = new Node(val);
    if (head == NULL) { // If the list is empty
        head = new_node;
        tail = new_node;
    } else {
        tail->next = new_node; // Link the current tail to the new node
        new_node->prev = tail; // Link the new node back to the current tail
        tail = new_node;       // Update the tail to the new node
    }
}
```

```
    void del(){
         if (head==NULL){
                return;
                }
                Node* temp = tail;
                tail = tail->prev;
                if(head==NULL){
                        tail=NULL;
                }
                else{
                        tail->next = NULL;
                }
                delete temp;
         }

    void display() {
        Node* temp = head;
        while (temp != NULL) {
            cout << temp->val;
            if (temp->next != NULL) { // Only add <-> between nodes
                cout << " <-> ";
            }
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    DOUBLELINKLIST dll;
```

```
    dll.insert(3);
    dll.insert(2);
    dll.del();
    dll.display();
    return 0;
}
```

Output:



Problem 3: Write code to delete a node by its value in a doubly linked list.

Code:

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

// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;

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

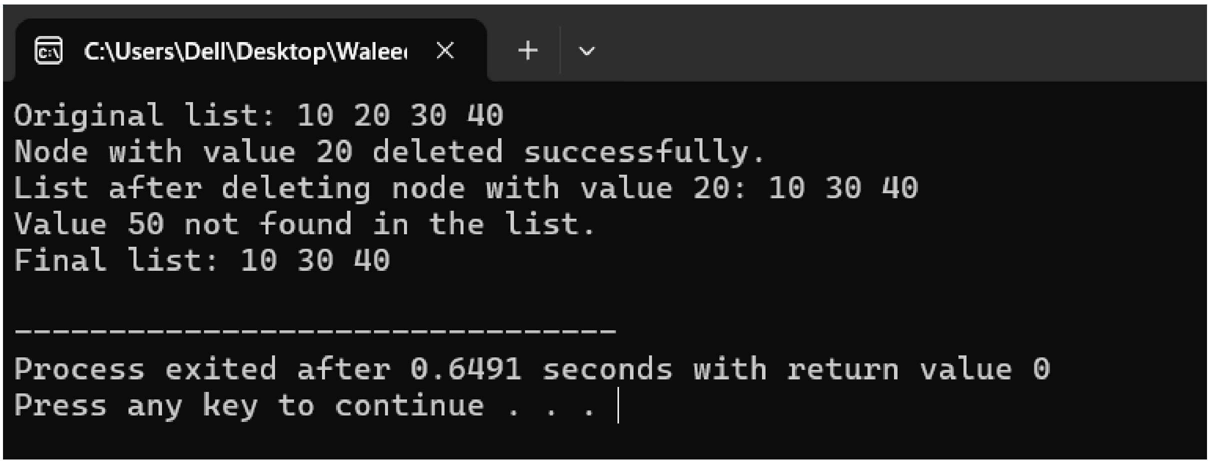
```
// Function to delete a node by its value
void deleteNodeByValue(Node*& head, int value) {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete.\n";
        return;
    }

    Node* current = head;

    // Traverse the list to find the node with the given value
    while (current != nullptr && current->data != value) {
        current = current->next;
    }

    // If the value is not found
    if (current == nullptr) {
        cout << "Value " << value << " not found in the list.\n";
        return;
    }

    // If the node to be deleted is the head node
    if (current == head) {
        head = current->next; // Move head to the next node
        if (head != nullptr) {
            head->prev = nullptr; // Update the new head's prev pointer
        }
    } else {
        // Update the pointers of the previous and next nodes
        if (current->prev != nullptr) {
            current->prev->next = current->next;
        }
        if (current->next != nullptr) {
```

```
            current->next->prev = current->prev;
        }
    }

    delete current; // Free the memory of the deleted node
    cout << "Node with value " << value << " deleted successfully.\n";
}

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

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

// Function to append a new node to the end of the list
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(value);

    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;

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

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

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

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

    // Attempt to delete a value not in the list
    deleteNodeByValue(head, 50);

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

```
    return 0;
}
```

Output:



Problem 4: How would you delete a node at a specific position in a doubly linked list? Show it in code.

Code:

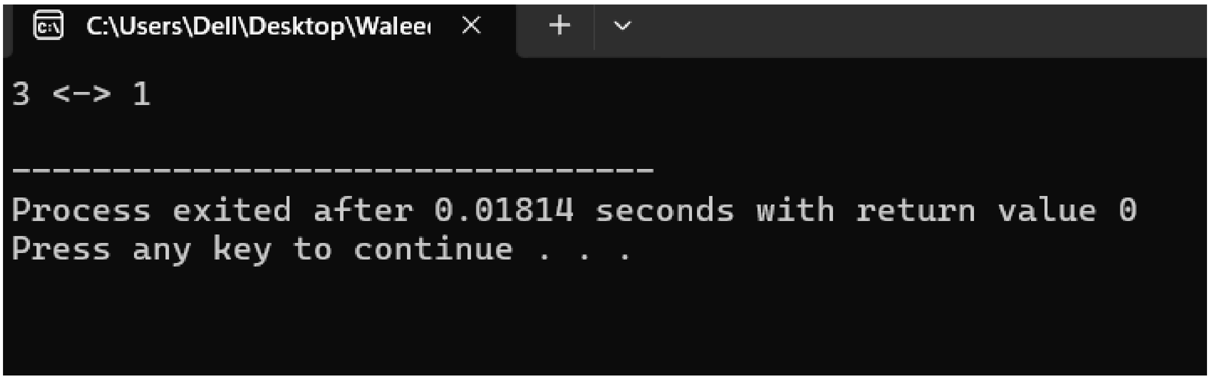
```
#include <iostream>
using namespace std;

class Node {
public:
    int val;
    Node* next;
    Node* prev;

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

```
        prev = NULL;
    }
};

class DOUBLELINKLIST {
public:
    Node* head;
    Node* tail;

    DOUBLELINKLIST() {
        head = NULL;
        tail = NULL;
    }

    void insert(int val) {
        Node* new_node = new Node(val);
        if (head == NULL) {
            head = new_node;
            tail = new_node;
        } else {
            tail->next = new_node;
            new_node->prev = tail;
            tail = new_node;
        }
    }

    void del(int p) {
        if (head == NULL) {
            cout << "List is empty." << endl;
            return;
        }
```

```
    Node* temp = head;
    int count = 1;

    while (temp != NULL && count < p) {
        temp = temp->next;
        count++;
    }

    if (temp == NULL) {
        cout << "Position out of bounds." << endl;
        return;
    }

    if (temp->prev != NULL)
        temp->prev->next = temp->next;

    if (temp->next != NULL)
        temp->next->prev = temp->prev;

    if (temp == head)
        head = temp->next;

    if (temp == tail)
        tail = temp->prev;

    delete temp;
}

void display() {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->val;
```

```
            if (temp->next != NULL) {
                cout << " <-> ";
            }
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    DOUBLELINKLIST dll;
    dll.insert(3);
    dll.insert(2);
    dll.insert(1);
    dll.del(2);
    dll.display();
    return 0;
}
```

Output:



Problem 5: After deleting a node, how will you write the forward and reverse traversal functions?

Code:

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

// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;

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

// Function to delete a node at a specific position
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete.\n";
        return;
    }

    if (position <= 0) {
        cout << "Invalid position. Position must be greater than 0.\n";
        return;
    }

    Node* current = head;

    // Traverse to the desired position
    int index = 1; // 1-based index
    while (current != nullptr && index < position) {
        current = current->next;
        index++;
```

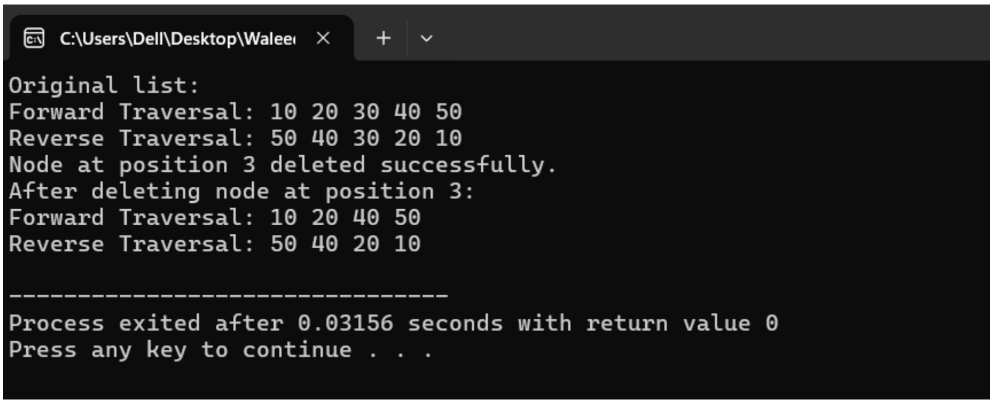
```
    }

    // If position is out of bounds
    if (current == nullptr) {
        cout << "Position " << position << " is out of bounds.\n";
        return;
    }

    // Update pointers to exclude the current node
    if (current->prev != nullptr) {
        current->prev->next = current->next;
    } else {
        head = current->next; // Update head if the first node is being deleted
    }
    if (current->next != nullptr) {
        current->next->prev = current->prev;
    }

    delete current; // Free memory of the deleted node
    cout << "Node at position " << position << " deleted successfully.\n";
}

// Function to append a new node to the end of the list
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(value);

    if (head == nullptr) {
        head = newNode;
        return;
    }

    Node* temp = head;
```

```cpp
    while (temp->next != nullptr) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}

// Forward traversal function
void forwardTraversal(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty.\n";
        return;
    }

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

// Reverse traversal function
void reverseTraversal(Node* head) {
    if (head == nullptr) {
        cout << "The list is empty.\n";
        return;
    }

    // Find the tail of the list
```

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

    // Traverse backward from the tail to the head
    cout << "Reverse Traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;

    // Append some nodes to the list
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);
    appendNode(head, 50);

    cout << "Original list:\n";
    forwardTraversal(head);
    reverseTraversal(head);

    // Delete a node at position 3
    deleteNodeAtPosition(head, 3);

    cout << "After deleting node at position 3:\n";
```

```
    forwardTraversal(head);
    reverseTraversal(head);

    return 0;
}
```

Output:



Circular Linked List

Problem 1: Write a program to delete the first node in a circular linked list.

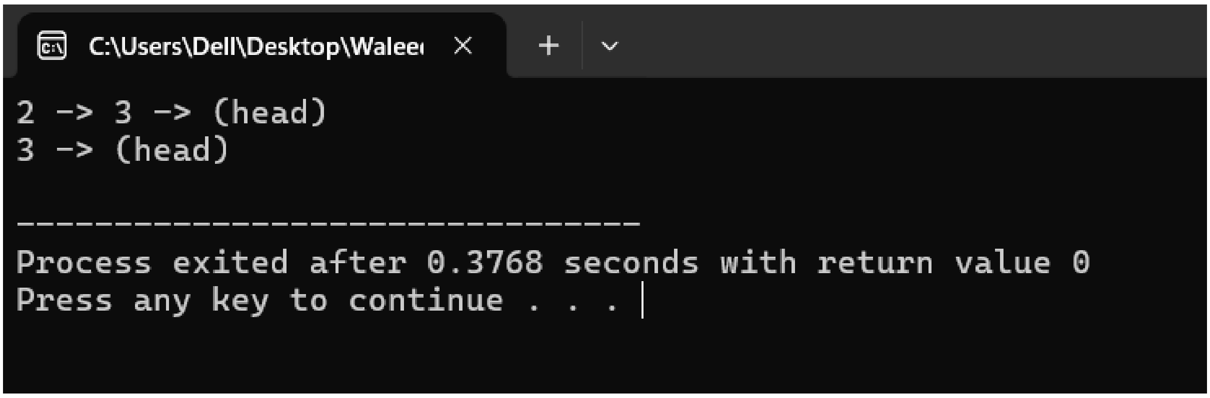
Code:

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

class Node {
public:
    int val;
    Node* next;
    Node(int data) {
        val = data;
        next = NULL;
    }
```

```
};

class Circular {
public:
    Node* head;
    Circular() {
        head = NULL;
    }

    void insert(int data) {
        Node* newNode = new Node(data);
        if (head == NULL) {
            head = newNode;
            newNode->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
            head = newNode;
        }
    }
    void deleteATstart(){
         if (head == NULL){
                    return;
                    }
                    Node* temp = head;
                    Node* tail = head;
                    while (tail->next != head){
                              tail = tail->next;
```

```
            }
            head = head->next;
            tail->next = head;
            delete temp;
        }
    void display() {
        if (head == NULL) {
            cout << "The list is empty." << endl;
            return;
        }

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

int main() {
    Circular cc;
    cc.insert(3);
    cc.insert(2);
    cc.display();
    cc.deleteATstart();
    cc.display();
    return 0;
}
```

Output:



Problem 2: How can you delete the last node in a circular linked list? Write the code.

Code:

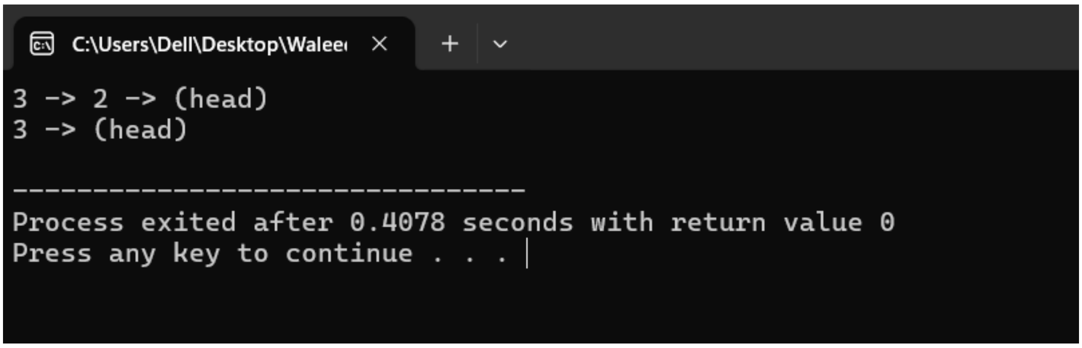
```
#include <iostream>
using namespace std;

class Node {
public:
    int val;
    Node* next;
    Node(int data) {
        val = data;
        next = NULL;
    }
};

class Circular {
public:
    Node* head;
    Circular() {
        head = NULL;
```

```
}

void insert(int data) {
    Node* newNode = new Node(data);
    if (head == NULL) {
        head = newNode;
        newNode->next = head;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;

    }
}
void deleteATstart(){
         if (head == NULL){
                    return;
                    }
                    Node* temp = head;
                    Node* tail = head;
                    while (tail->next != head){
                              tail = tail->next;
                    }
                    head = head->next;
                    tail->next = head;
                    delete temp;
         }
         void deleteTail(){
                    if (head == NULL){
```

```
                return;
            }
            Node* tail = head;
            while (tail->next->next != head){
                tail = tail->next;
            }
            Node* temp = tail->next;
            tail->next = head;
            delete temp;
         }
    void display() {
        if (head == NULL) {
            cout << "The list is empty." << endl;
            return;
        }

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

int main() {
    Circular cc;
    cc.insert(3);
    cc.insert(2);
    cc.display();
    cc.deleteTail();
    cc.display();
```

```
    return 0;
}
```

Output:



Problem 3: Write a function to delete a node by its value in a circular linked list.

Code:

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

// Node structure
struct Node {
    int data;
    Node* next;

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

// Function to delete a node by its value
void deleteNodeByValue(Node*& head, int value) {
```

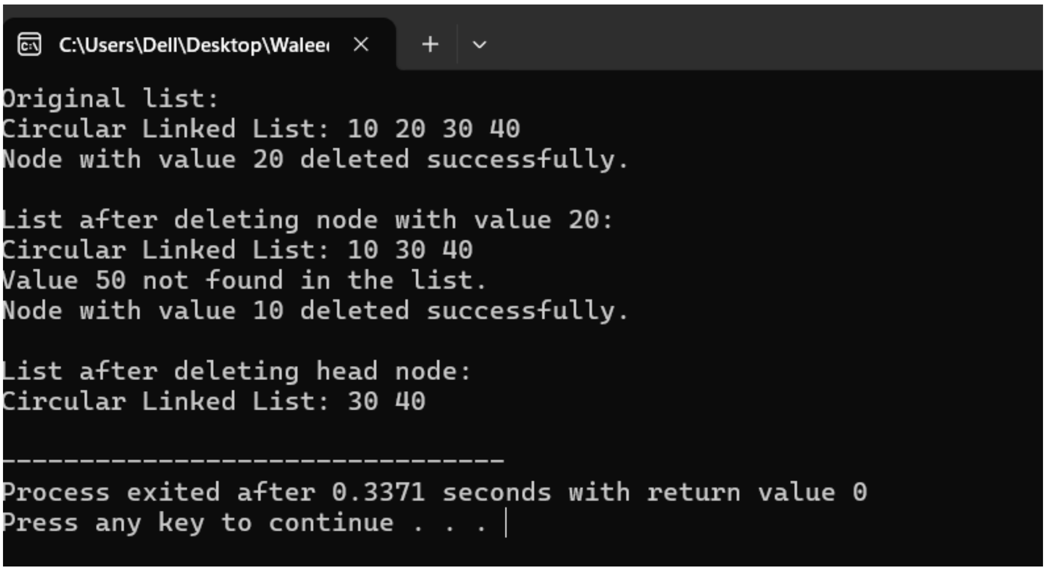
```
if (head == nullptr) {
    cout << "The list is empty. Nothing to delete.\n";
    return;
}

Node* current = head;
Node* prev = nullptr;

// If the list contains only one node
if (head->data == value && head->next == head) {
    delete head;
    head = nullptr;
    cout << "Node with value " << value << " deleted successfully.\n";
    return;
}

// Traverse the list to find the node with the given value
do {
    if (current->data == value) {
        if (prev == nullptr) { // Node to delete is the head
            Node* tail = head;
            while (tail->next != head) { // Find the tail node
                tail = tail->next;
            }

            // Update head and tail pointers
            tail->next = head->next;
            Node* temp = head;
            head = head->next;
            delete temp;
        } else { // Node to delete is not the head
            prev->next = current->next;
```

```cpp
            delete current;
        }

        cout << "Node with value " << value << " deleted successfully.\n";
        return;
    }
    prev = current;
    current = current->next;
  } while (current != head);

  cout << "Value " << value << " not found in the list.\n";
}

// Function to append a node to the circular linked list
void appendNode(Node*& head, int value) {
  Node* newNode = new Node(value);

  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;
}
```

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

    Node* temp = head;
    cout << "Circular Linked List: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

int main() {
    Node* head = nullptr;

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

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

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

    cout << "List after deleting node with value 20:\n";
```

```
    displayList(head);

    // Attempt to delete a value not in the list
    deleteNodeByValue(head, 50);

    // Delete the head node
    deleteNodeByValue(head, 10);

    cout << "List after deleting head node:\n";
    displayList(head);

    return 0;
}
```

Output:



Problem 4: How will you delete a node at a specific position in a circular linked list? Write code for it.

Code:

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

```
// Node structure
struct Node {
    int data;
    Node* next;

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

// Function to delete a node at a specific position
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {
        cout << "The list is empty. Nothing to delete.\n";
        return;
    }

    if (position <= 0) {
        cout << "Invalid position. Position must be greater than 0.\n";
        return;
    }

    Node* current = head;
    Node* prev = nullptr;

    // Case 1: Deleting the head node
    if (position == 1) {
        // Find the last node
        Node* tail = head;
        while (tail->next != head) {
            tail = tail->next;
        }
```

```
    // Update head and adjust pointers
    if (head->next == head) { // If only one node in the list
        delete head;
        head = nullptr;
    } else {
        tail->next = head->next;
        Node* temp = head;
        head = head->next;
        delete temp;
    }
    cout << "Node at position 1 deleted successfully.\n";
    return;
}

// Case 2: Deleting a node at another position
int index = 1;
while (current->next != head && index < position) {
    prev = current;
    current = current->next;
    index++;
}

if (index < position || current == head) { // Position out of bounds
    cout << "Position " << position << " is out of bounds.\n";
    return;
}

// Update pointers and delete the node
prev->next = current->next;
delete current;

cout << "Node at position " << position << " deleted successfully.\n";
```

```
}

// Function to append a node to the circular linked list
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(value);

    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;
}

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

    Node* temp = head;
    cout << "Circular Linked List: ";
    do {
        cout << temp->data << " ";
```

```
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

int main() {
    Node* head = nullptr;

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

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

    // Delete a node at position 2
    deleteNodeAtPosition(head, 2);

    cout << "\nList after deleting node at position 2:\n";
    displayList(head);

    // Attempt to delete a node at an invalid position
    deleteNodeAtPosition(head, 10);

    // Delete the head node
    deleteNodeAtPosition(head, 1);

    cout << "\nList after deleting head node:\n";
    displayList(head);
```

```
    return 0;
}
```

Output:

```
C:\Users\Dell\Desktop\Walee
Original list:
Circular Linked List: 10 20 30 40
Node at position 2 deleted successfully.

List after deleting node at position 2:
Circular Linked List: 10 30 40
Position 10 is out of bounds.
Node at position 1 deleted successfully.

List after deleting head node:
Circular Linked List: 30 40

--------------------------------
Process exited after 0.02414 seconds with return value 0
Press any key to continue . . .
```

Problem 5: Write a program to show forward traversal after deleting a node in a circular linked list.

Code:

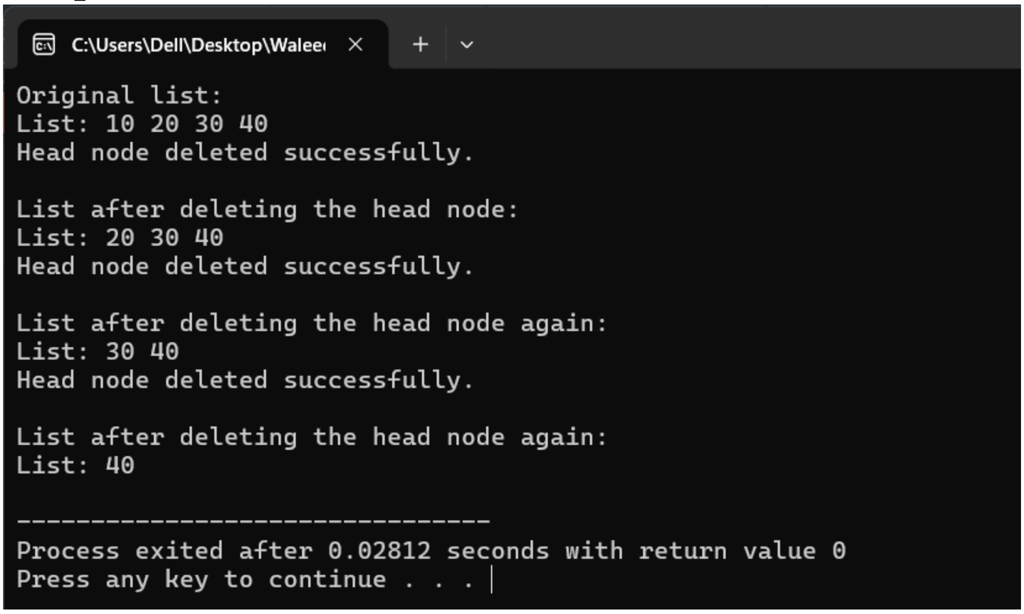
```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* next;

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

// Function to delete the head node
void deleteHeadNode(Node*& head) {
    if (!head) {
        cout << "The list is empty. Nothing to delete.\n";
```

```
        return;
    }

    if (head->next == head) { // Single node case
        delete head;
        head = nullptr;
    } else {
        Node* tail = head;
        while (tail->next != head) tail = tail->next; // Find the tail node
        tail->next = head->next;
        Node* temp = head;
        head = head->next;
        delete temp;
    }
    cout << "Head node deleted successfully.\n";
}

// Function to append a node to the circular linked list
void appendNode(Node*& head, int value) {
    Node* newNode = new Node(value);
    if (!head) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) temp = temp->next;
    temp->next = newNode;
    newNode->next = head;
}

// Function to display the circular linked list
```

```
void displayList(Node* head) {
    if (!head) {
        cout << "List is empty.\n";
        return;
    }
    Node* temp = head;
    cout << "List: ";
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

// Main function
int main() {
    Node* head = nullptr;

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

    // Display the original list
    cout << "Original list:\n";
    displayList(head);

    // Delete the head node and display the list after each deletion
    deleteHeadNode(head);
    cout << "\nList after deleting the head node:\n";
    displayList(head);
```

```
    deleteHeadNode(head);
    cout << "\nList after deleting the head node again:\n";
    displayList(head);

    deleteHeadNode(head);
    cout << "\nList after deleting the head node again:\n";
    displayList(head);

    return 0;
}
```

Output:



Binary Search Tree

Problem 1: Write a program to count all the nodes in a binary search tree.

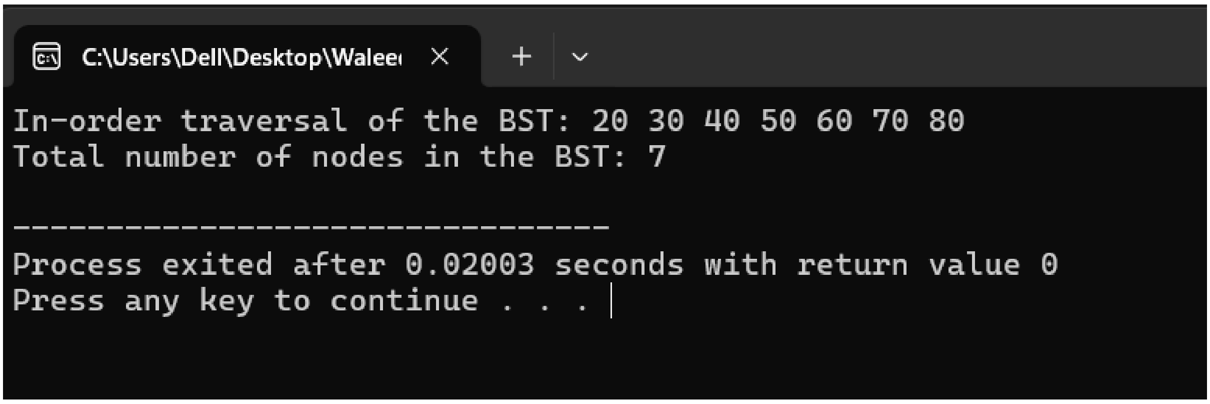
Code:

```
#include <iostream>
```

```
using namespace std;

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node into the BST
Node* insertNode(Node* root, int value) {
    if (!root) return new Node(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else
        root->right = insertNode(root->right, value);

    return root;
}

// Function to count the nodes in the BST
int countNodes(Node* root) {
    if (!root) return 0; // Base case: empty tree

    // Count the current node + left subtree + right subtree
    return 1 + countNodes(root->left) + countNodes(root->right);
}

// Function to display the BST (In-order Traversal)
```

```
void inOrderTraversal(Node* root) {
    if (!root) return;

    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 70);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Display the BST
    cout << "In-order traversal of the BST: ";
    inOrderTraversal(root);
    cout << endl;

    // Count the nodes
    int totalNodes = countNodes(root);
    cout << "Total number of nodes in the BST: " << totalNodes << endl;

    return 0;
}
```

Output:



Problem 2: How can you search for a specific value in a binary search tree? Write the code

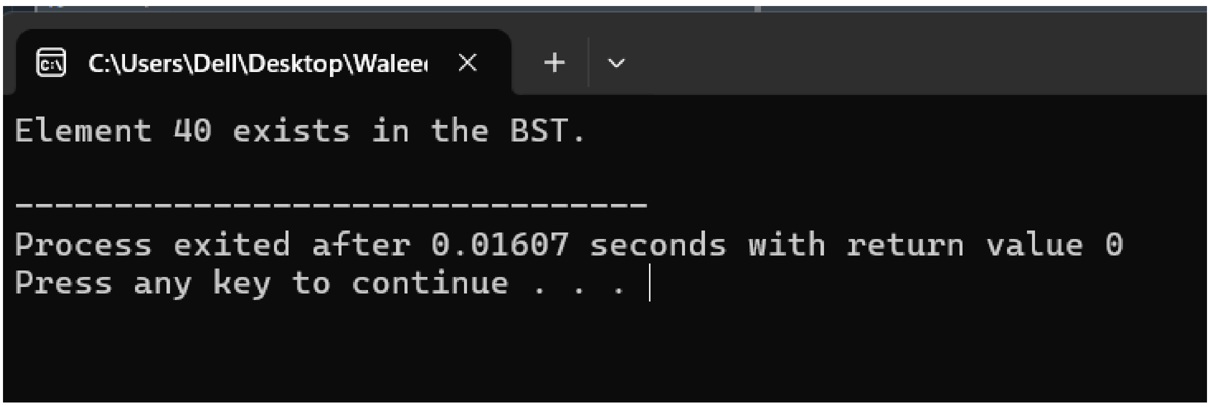
Code:

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

// Define a node of the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

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

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

```
    if (value < root->data) {
        root->left = insertNode(root->left, value);
    } else {
        root->right = insertNode(root->right, value);
    }

    return root;
}

// Function to search for an element in the binary search tree
bool searchNode(Node* root, int value) {
    if (root == nullptr) {
        return false;  // Element not found, root is nullptr
    }

    if (root->data == value) {
        return true;  // Element found
    }

    if (value < root->data) {
        return searchNode(root->left, value);  // Search in left subtree
    } else {
        return searchNode(root->right, value); // Search in right subtree
    }
}

int main() {
    Node* root = nullptr;

    // Insert elements into the binary search tree
    root = insertNode(root, 50);
```

```
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Search for an element
    int searchValue = 40;
    bool result = searchNode(root, searchValue);

    if (result) {
        cout << "Element " << searchValue << " exists in the BST." << endl;
    } else {
        cout << "Element " << searchValue << " does not exist in the BST." << endl;
    }

    return 0;
}
```

Output:



Problem 3: Write code to traverse a binary search tree in in-order, pre-order, and post order.

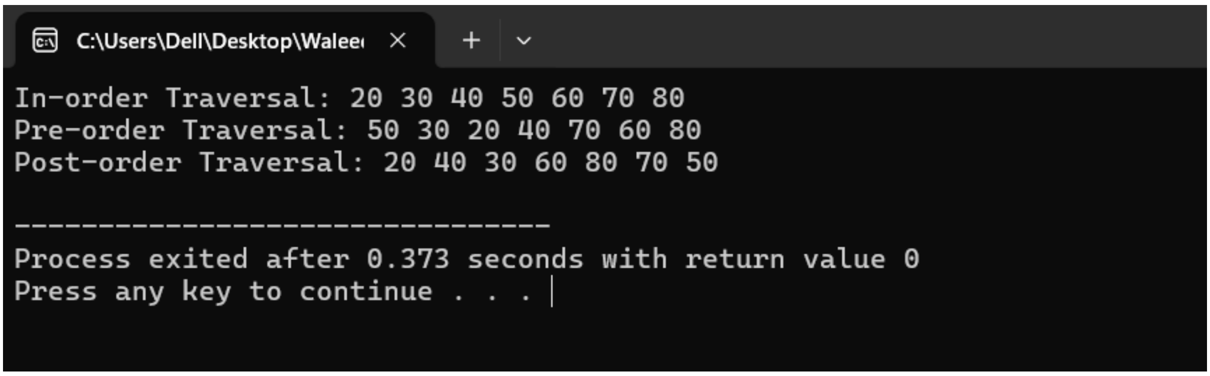
Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node into the BST
Node* insertNode(Node* root, int value) {
    if (!root) return new Node(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else
        root->right = insertNode(root->right, value);

    return root;
}

// In-order Traversal: Left -> Root -> Right
void inOrderTraversal(Node* root) {
    if (!root) return;
```

```
    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}

// Pre-order Traversal: Root -> Left -> Right
void preOrderTraversal(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    preOrderTraversal(root->left);
    preOrderTraversal(root->right);
}

// Post-order Traversal: Left -> Right -> Root
void postOrderTraversal(Node* root) {
    if (!root) return;
    postOrderTraversal(root->left);
    postOrderTraversal(root->right);
    cout << root->data << " ";
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 70);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 60);
```

```
    root = insertNode(root, 80);

    // Perform and display all three traversals
    cout << "In-order Traversal: ";
    inOrderTraversal(root);
    cout << endl;

    cout << "Pre-order Traversal: ";
    preOrderTraversal(root);
    cout << endl;

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

    return 0;
}
```

Output:



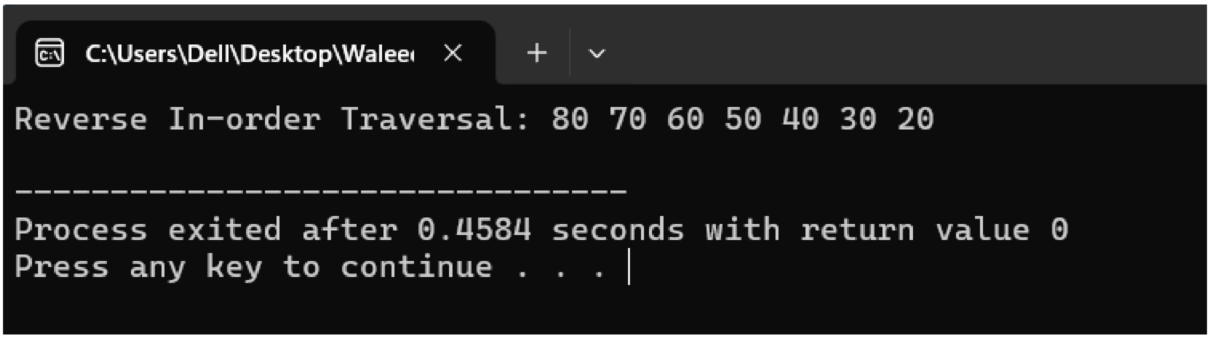
Problem 4: How will you write reverse in-order traversal for a binary search tree? Show it in code.

Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node into the BST
Node* insertNode(Node* root, int value) {
    if (!root) return new Node(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else
        root->right = insertNode(root->right, value);

    return root;
}

// Reverse In-order Traversal: Right -> Root -> Left
void reverseInOrderTraversal(Node* root) {
    if (!root) return;
    reverseInOrderTraversal(root->right);  // Visit right subtree
    cout << root->data << " ";              // Visit root
    reverseInOrderTraversal(root->left);   // Visit left subtree
}
```

```
// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 70);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Perform and display the reverse in-order traversal
    cout << "Reverse In-order Traversal: ";
    reverseInOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:



Problem 5: Write a program to check if there are duplicate values in a binary search tree.

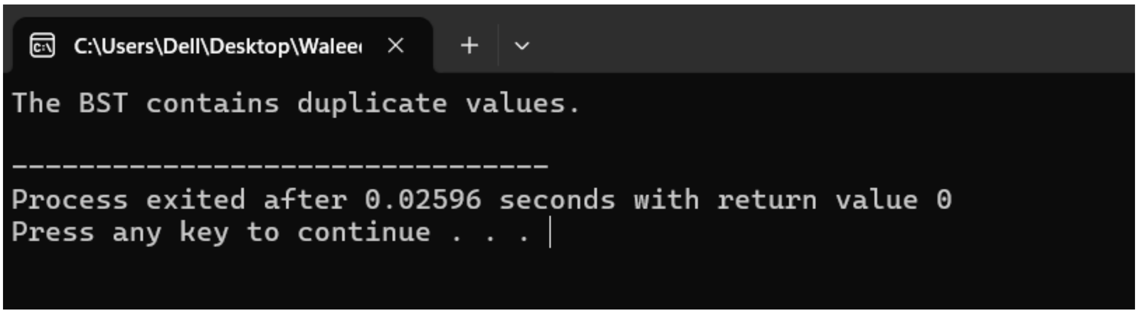
Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node into the BST
Node* insertNode(Node* root, int value) {
    if (!root) return new Node(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else
        root->right = insertNode(root->right, value);

    return root;
}

// Function to check for duplicates using in-order traversal
bool checkForDuplicates(Node* root, int& prevValue) {
    if (!root) return false;

    // Check the left subtree
    if (checkForDuplicates(root->left, prevValue)) return true;
```

```
    // Check the current node for duplicate
    if (root->data == prevValue) return true;

    // Update the previous value with the current node's value
    prevValue = root->data;

    // Check the right subtree
    return checkForDuplicates(root->right, prevValue);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 70);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    // Inserting a duplicate value to test
    root = insertNode(root, 40);  // Duplicate value

    int prevValue = -1;  // Initialize previous value to a value that can't be in the tree
    bool hasDuplicates = checkForDuplicates(root, prevValue);

    if (hasDuplicates)
        cout << "The BST contains duplicate values." << endl;
    else
```

```
        cout << "The BST does not contain duplicate values." << endl;

    return 0;
}
```

Output:



Problem 6: 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.

Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

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

```
// Function to insert a node into the BST
Node* insertNode(Node* root, int value) {
    if (!root) return new Node(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else
        root->right = insertNode(root->right, value);

    return root;
}

// Function to find the minimum node in a subtree (used for finding in-order successor)
Node* findMin(Node* root) {
    while (root && root->left)
        root = root->left;
    return root;
}

// Function to delete a node in a BST
Node* deleteNode(Node* root, int value) {
    if (!root) return root;

    // If value to be deleted is smaller than root's value, it lies in the left subtree
    if (value < root->data)
        root->left = deleteNode(root->left, value);

    // If value to be deleted is greater than root's value, it lies in the right subtree
    else if (value > root->data)
        root->right = deleteNode(root->right, value);

    // If value is the same as root's value, this is the node to be deleted
```

```
    else {
        // Case 1: Node has no children (leaf node)
        if (!root->left && !root->right) {
            delete root;
            root = nullptr;
        }
        // Case 2: Node has one child
        else if (!root->left) {
            Node* temp = root;
            root = root->right;
            delete temp;
        } else if (!root->right) {
            Node* temp = root;
            root = root->left;
            delete temp;
        }
        // Case 3: Node has two children
        else {
            Node* temp = findMin(root->right); // Get the in-order successor (smallest in right subtree)
            root->data = temp->data; // Replace root's value with in-order successor's value
            root->right = deleteNode(root->right, temp->data); // Delete the in-order successor
        }
    }

    return root;
}

// Function to perform in-order traversal
void inOrderTraversal(Node* root) {
    if (!root) return;
    inOrderTraversal(root->left);
    cout << root->data << " ";
```

```
    inOrderTraversal(root->right);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 70);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    cout << "In-order traversal of the BST before deletion: ";
    inOrderTraversal(root);
    cout << endl;

    // Delete nodes and perform in-order traversal after each deletion

    // Deleting a leaf node (e.g., 20)
    root = deleteNode(root, 20);
    cout << "In-order traversal after deleting leaf node 20: ";
    inOrderTraversal(root);
    cout << endl;

    // Deleting a node with one child (e.g., 30)
    root = deleteNode(root, 30);
    cout << "In-order traversal after deleting node with one child (30): ";
    inOrderTraversal(root);
```

```
    cout << endl;

    // Deleting a node with two children (e.g., 50)
    root = deleteNode(root, 50);
    cout << "In-order traversal after deleting node with two children (50): ";
    inOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Code:

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

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;

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

// Function to insert a node into the BST
Node* insertNode(Node* root, int value) {
    if (!root) return new Node(value);

    if (value < root->data)
        root->left = insertNode(root->left, value);
    else
        root->right = insertNode(root->right, value);
```

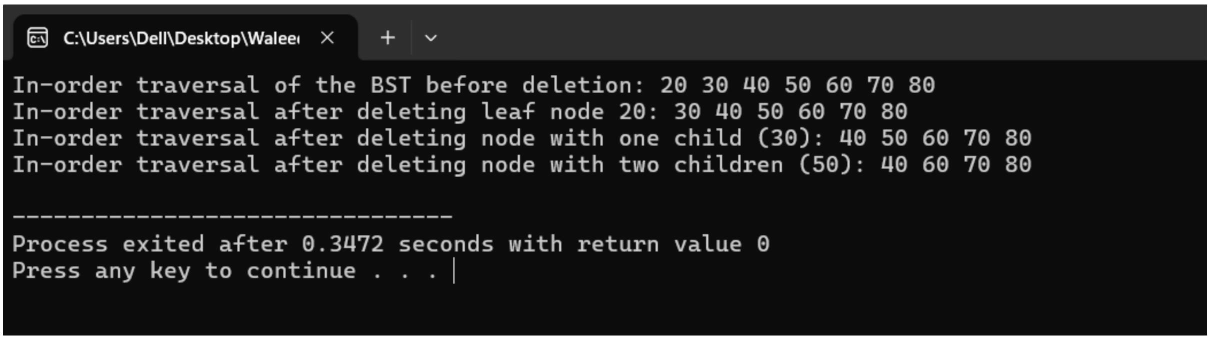
```
    return root;
}

// Function to find the minimum node in a subtree (used for finding in-order successor)
Node* findMin(Node* root) {
    while (root && root->left)
        root = root->left;
    return root;
}

// Function to delete a node in a BST
Node* deleteNode(Node* root, int value) {
    if (!root) return root;

    // If value to be deleted is smaller than root's value, it lies in the left subtree
    if (value < root->data)
        root->left = deleteNode(root->left, value);

    // If value to be deleted is greater than root's value, it lies in the right subtree
    else if (value > root->data)
        root->right = deleteNode(root->right, value);

    // If value is the same as root's value, this is the node to be deleted
    else {
        // Case 1: Node has no children (leaf node)
        if (!root->left && !root->right) {
            delete root;
            root = nullptr;
        }
        // Case 2: Node has one child
        else if (!root->left) {
```

```
            Node* temp = root;
            root = root->right;
            delete temp;
        } else if (!root->right) {
            Node* temp = root;
            root = root->left;
            delete temp;
        }
        // Case 3: Node has two children
        else {
            Node* temp = findMin(root->right); // Get the in-order successor (smallest in right subtree)
            root->data = temp->data; // Replace root's value with in-order successor's value
            root->right = deleteNode(root->right, temp->data); // Delete the in-order successor
        }
    }

    return root;
}

// Function to perform in-order traversal
void inOrderTraversal(Node* root) {
    if (!root) return;
    inOrderTraversal(root->left);
    cout << root->data << " ";
    inOrderTraversal(root->right);
}

// Main function
int main() {
    Node* root = nullptr;

    // Insert nodes into the BST
```

```
root = insertNode(root, 50);
root = insertNode(root, 30);
root = insertNode(root, 70);
root = insertNode(root, 20);
root = insertNode(root, 40);
root = insertNode(root, 60);
root = insertNode(root, 80);

cout << "In-order traversal of the BST before deletion: ";
inOrderTraversal(root);
cout << endl;

// Delete nodes and perform in-order traversal after each deletion

// Deleting a leaf node (e.g., 20)
root = deleteNode(root, 20);
cout << "In-order traversal after deleting leaf node 20: ";
inOrderTraversal(root);
cout << endl;

// Deleting a node with one child (e.g., 30)
root = deleteNode(root, 30);
cout << "In-order traversal after deleting node with one child (30): ";
inOrderTraversal(root);
cout << endl;

// Deleting a node with two children (e.g., 50)
root = deleteNode(root, 50);
cout << "In-order traversal after deleting node with two children (50): ";
inOrderTraversal(root);
cout << endl;
```

```
    return 0;
}
```

Output:



End of Assignment
