DATA STRUCTURES FINAL ASSIGNMENT

DOUBLY LINKED LIST:

- **Write a program to delete the first node in a doubly linked list**

Code:

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

// Definition of a doubly linked list node
struct Node {
    int data;
    Node* prev;
    Node* next;

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

// Function to delete the first node in a doubly linked list
void deleteFirstNode(Node*& head) {
    if (!head) {
        cout << "List is already empty!" << endl;
        return;
    }

    Node* temp = head;  // Temporary pointer to the node being deleted
    head = head->next;  // Move the head pointer to the next node

    if (head) {
        head->prev = nullptr;  // Set the previous pointer of the new head to nullptr
    }

    delete temp;  // Free memory of the old head
    cout << "First node deleted successfully." << endl;
}

// Function to print the doubly linked list
void printList(Node* head) {
    Node* current = head;
    while (current) {
        cout << current->data << " ";
        current = current->next;
```

```
    }
    cout << endl;
}

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

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

    Node* current = head;
    while (current->next) {
        current = current->next;
    }

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

int main() {
    Node* head = nullptr;

    // Create a sample doubly linked list
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);

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

    // Delete the first node
    deleteFirstNode(head);

    cout << "After deleting the first node: ";
    printList(head);

    // Clean up remaining nodes
    while (head) {
```

```
        deleteFirstNode(head);
    }

    return 0;
}
```

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

Code:

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

// Definition of a doubly linked list node
struct Node {
    int data;
    Node* prev;
    Node* next;

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

// Function to delete the last node in a doubly linked list
void deleteLastNode(Node*& head) {
    if (!head) {
        cout << "List is already empty!" << endl;
        return;
    }

    if (!head->next) {
        // If there's only one node in the list
        delete head;
        head = nullptr;
        cout << "Last node deleted successfully." << endl;
        return;
    }

    Node* temp = head;

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

```
    // Update the second-to-last node's next pointer
    temp->prev->next = nullptr;

    delete temp;  // Free memory of the last node
    cout << "Last node deleted successfully." << endl;
}

// Function to print the doubly linked list
void printList(Node* head) {
    Node* current = head;
    while (current) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}

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

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

    Node* current = head;
    while (current->next) {
        current = current->next;
    }

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

int main() {
    Node* head = nullptr;

    // Create a sample doubly linked list
    appendNode(head, 10);
    appendNode(head, 20);
```

```
    appendNode(head, 30);

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

    // Delete the last node
    deleteLastNode(head);

    cout << "After deleting the last node: ";
    printList(head);

    // Clean up remaining nodes
    while (head) {
        deleteLastNode(head);
    }

    return 0;
}
```

- **Write code to delete a node by its value in a doubly linked list.**

Code:

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

// Definition of a doubly linked list node
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) {
        cout << "List is empty! Cannot delete value " << value << "." << endl;
        return;
    }

    Node* current = head;
```

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

    // If the value is not found
    if (!current) {
        cout << "Value " << value << " not found in the list." << endl;
        return;
    }

    // If the node to delete is the head
    if (current == head) {
        head = head->next;  // Move the head to the next node
        if (head) {
            head->prev = nullptr;  // Update the new head's prev pointer
        }
    } else {
        // Update the previous node's next pointer
        current->prev->next = current->next;

        // Update the next node's prev pointer, if it exists
        if (current->next) {
            current->next->prev = current->prev;
        }
    }

    delete current;  // Free the memory of the deleted node
    cout << "Value " << value << " deleted successfully." << endl;
}

// Function to print the doubly linked list
void printList(Node* head) {
    Node* current = head;
    while (current) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}
```

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

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

    Node* current = head;
    while (current->next) {
        current = current->next;
    }

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

int main() {
    Node* head = nullptr;

    // Create a sample doubly linked list
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);

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

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

    cout << "After deleting value 20: ";
    printList(head);

    // Attempt to delete a non-existent value
    deleteNodeByValue(head, 50);

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

```
    cout << "After deleting value 10: ";
    printList(head);

    // Delete the last node
    deleteNodeByValue(head, 40);

    cout << "After deleting value 40: ";
    printList(head);

    // Clean up the remaining list
    while (head) {
        deleteNodeByValue(head, head->data);
    }

    return 0;
}
```

- **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;

// Definition of a doubly linked list node
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) {
        cout << "List is empty! Cannot delete position " << position << "." << endl;
        return;
    }

    if (position < 1) {
        cout << "Invalid position! Position must be >= 1." << endl;
```

```
        return;
    }

    Node* current = head;

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

    // If the position is out of bounds
    if (!current) {
        cout << "Position " << position << " is out of bounds." << endl;
        return;
    }

    // If the node to delete is the head
    if (current == head) {
        head = head->next;  // Move the head to the next node
        if (head) {
            head->prev = nullptr;  // Update the new head's prev pointer
        }
    } else {
        // Update the previous node's next pointer
        current->prev->next = current->next;

        // Update the next node's prev pointer, if it exists
        if (current->next) {
            current->next->prev = current->prev;
        }
    }

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

// Function to print the doubly linked list
void printList(Node* head) {
    Node* current = head;
    while (current) {
        cout << current->data << " ";
```

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

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

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

    Node* current = head;
    while (current->next) {
        current = current->next;
    }

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

int main() {
    Node* head = nullptr;

    // Create a sample doubly linked list
    appendNode(head, 10);
    appendNode(head, 20);
    appendNode(head, 30);
    appendNode(head, 40);

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

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

    cout << "After deleting node at position 2: ";
    printList(head);
```

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

    cout << "After deleting node at position 1: ";
    printList(head);

    // Delete the last node
    deleteNodeAtPosition(head, 2);

    cout << "After deleting node at position 2: ";
    printList(head);

    // Try to delete a node at an out-of-bounds position
    deleteNodeAtPosition(head, 5);

    // Clean up remaining list
    while (head) {
        deleteNodeAtPosition(head, 1);
    }

    return 0;
}
```

- **After deleting a node, how will you write the forward and reverse traversal functions?**

Forward Traversal

```
void forwardTraversal(Node* head) {
    Node* current = head;
    cout << "Forward traversal: ";
    while (current) {
        cout << current->data << " ";
        current = current->next;
    }
    cout << endl;
}
```

Reverse Traversal

```
void reverseTraversal(Node* head) {
    if (!head) {
        cout << "Reverse traversal: List is empty." << endl;
        return;
    }
```

```
    // Find the last node
    Node* current = head;
    while (current->next) {
        current = current->next;
    }

    // Traverse backward using prev pointers
    cout << "Reverse traversal: ";
    while (current) {
        cout << current->data << " ";
        current = current->prev;
    }
    cout << endl;
}
```

Circular Linked List:

- **Write a program to delete the first node in a circular linked list.**

Code:

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

// Node definition for a circular linked list
struct Node {
    int data;
    Node* next;

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

// Function to delete the first node in a circular linked list
void deleteFirstNode(Node*& head) {
    if (!head) {
        cout << "The list is empty! Cannot delete the first node." << endl;
        return;
    }

    if (head->next == head) { // Only one node in the list
        delete head;
        head = nullptr;
        cout << "The only node in the list has been deleted." << endl;
        return;
    }

    // More than one node in the list
    Node* last = head;
    while (last->next != head) { // Find the last node
        last = last->next;
    }

    Node* temp = head; // Store the node to be deleted
    head = head->next; // Update the head to the next node
    last->next = head; // Maintain the circular structure
    delete temp;       // Free the memory of the old head
    cout << "The first node has been deleted." << endl;
}
```

```
// Function to print the circular linked list
void printList(Node* head) {
    if (!head) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* current = head;
    do {
        cout << current->data << " ";
        current = current->next;
    } while (current != head); // Loop until we circle back to the head
    cout << endl;
}

int main() {
    Node* head = nullptr;

    // Manually create a circular linked list
    head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);
    head->next->next->next = new Node(40);
    head->next->next->next->next = head;  // Last node points back to head

    cout << "Initial list: ";
    printList(head);

    // Delete the first node
    deleteFirstNode(head);
    cout << "After deleting the first node: ";
    printList(head);

    // Delete the first node again
    deleteFirstNode(head);
    cout << "After deleting the first node again: ";
    printList(head);

    // Clean up the remaining nodes
    while (head) {
        deleteFirstNode(head);
```

```
    }

    cout << "After deleting all nodes: ";
    printList(head);

    return 0;
}
```

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

Code:

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

// Node definition for a circular linked list
struct Node {
    int data;
    Node* next;

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

// Function to delete the last node in a circular linked list
void deleteLastNode(Node*& head) {
    if (!head) {
        cout << "The list is empty! Cannot delete the last node." << endl;
        return;
    }

    if (head->next == head) { // Only one node in the list
        delete head;
        head = nullptr;
        cout << "The only node in the list has been deleted." << endl;
        return;
    }

    // More than one node in the list
    Node* secondLast = head;
    while (secondLast->next != head && secondLast->next->next != head) { // Find second last node
        secondLast = secondLast->next;
    }
```

```
    Node* last = secondLast->next; // Last node
    secondLast->next = head; // Update second last node's next to head (circular structure)
    delete last; // Free the memory of the last node
    cout << "The last node has been deleted." << endl;
}

// Function to print the circular linked list
void printList(Node* head) {
    if (!head) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* current = head;
    do {
        cout << current->data << " ";
        current = current->next;
    } while (current != head); // Loop until we circle back to the head
    cout << endl;
}

// Main function to test the implementation
int main() {
    Node* head = nullptr;

    // Manually create a circular linked list
    head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);
    head->next->next->next = new Node(40);
    head->next->next->next->next = head;  // Last node points back to head

    cout << "Initial list: ";
    printList(head);

    // Delete the last node
    deleteLastNode(head);
    cout << "After deleting the last node: ";
    printList(head);

    // Delete the last node again
```

```
    deleteLastNode(head);
    cout << "After deleting the last node again: ";
    printList(head);

    // Clean up the remaining nodes
    while (head) {
        deleteLastNode(head);
    }

    cout << "After deleting all nodes: ";
    printList(head);

    return 0;
}
```

- **Write a function to delete a node by its value in a circular linked list.**

Code:

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

// Node definition for a circular linked list
struct Node {
    int data;
    Node* next;

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

// Function to delete a node by its value in a circular linked list
void deleteNodeByValue(Node*& head, int value) {
    if (!head) {
        cout << "The list is empty! Cannot delete a node with value " << value << "." << endl;
        return;
    }

    Node* current = head;
    Node* previous = nullptr;

    // Handle the case when the node to delete is the only node in the list
    if (head->data == value && head->next == head) {
        delete head;
        head = nullptr;
```

```
        cout << "The node with value " << value << " has been deleted." << endl;
        return;
    }

    // If the node to delete is the first node (head node)
    if (head->data == value) {
        // Traverse to find the last node
        Node* last = head;
        while (last->next != head) {
            last = last->next;
        }

        // Update the head and adjust the last node's next pointer
        last->next = head->next;
        Node* temp = head;
        head = head->next;
        delete temp;
        cout << "The node with value " << value << " has been deleted." << endl;
        return;
    }

    // Traverse the list to find the node to delete
    current = head;
    while (current != head || (previous == nullptr && current != head)) {
        if (current->data == value) {
            previous->next = current->next;
            delete current;
            cout << "The node with value " << value << " has been deleted." << endl;
            return;
        }
        previous = current;
        current = current->next;
    }

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

// Function to print the circular linked list
void printList(Node* head) {
    if (!head) {
        cout << "The list is empty." << endl;
```

```
        return;
    }

    Node* current = head;
    do {
        cout << current->data << " ";
        current = current->next;
    } while (current != head); // Loop until we circle back to the head
    cout << endl;
}

// Main function to test the implementation
int main() {
    Node* head = nullptr;

    // Manually create a circular linked list
    head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);
    head->next->next->next = new Node(40);
    head->next->next->next->next = head;  // Last node points back to head

    cout << "Initial list: ";
    printList(head);

    // Delete a node by its value
    deleteNodeByValue(head, 20);
    cout << "After deleting node with value 20: ";
    printList(head);

    // Delete a node by its value
    deleteNodeByValue(head, 10);
    cout << "After deleting node with value 10: ";
    printList(head);

    // Try to delete a node that doesn't exist
    deleteNodeByValue(head, 100);
    cout << "After attempting to delete node with value 100: ";
    printList(head);

    // Clean up the remaining nodes
```

```
    while (head) {
        deleteNodeByValue(head, head->data);
    }

    cout << "After deleting all nodes: ";
    printList(head);

    return 0;
}
```

- **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 definition for a circular linked list
struct Node {
    int data;
    Node* next;

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

// Function to delete a node at a specific position in a circular linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (!head) {
        cout << "The list is empty! Cannot delete a node at position " << position << "." << endl;
        return;
    }

    // Case 1: If position is 0, delete the head node
    if (position == 0) {
        if (head->next == head) { // Only one node in the list
            delete head;
            head = nullptr;
            cout << "The only node in the list has been deleted." << endl;
            return;
        }

        // More than one node in the list
        Node* last = head;
```

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

        Node* temp = head;
        head = head->next;  // Update head to the next node
        last->next = head;  // Maintain the circular structure
        delete temp;        // Free the memory of the old head
        cout << "Node at position 0 has been deleted." << endl;
        return;
    }

    // Case 2: If position is greater than 0, traverse the list to find the node to delete
    Node* current = head;
    Node* previous = nullptr;
    int currentPos = 0;

    while (currentPos < position && current->next != head) {
        previous = current;
        current = current->next;
        currentPos++;
    }

    // If the position is out of bounds
    if (currentPos != position) {
        cout << "Position " << position << " is out of bounds." << endl;
        return;
    }

    // If we're deleting the last node
    if (current->next == head) {
        previous->next = head;
    } else {
        previous->next = current->next;
    }

    delete current; // Free the memory of the node
    cout << "Node at position " << position << " has been deleted." << endl;
}

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

```
void printList(Node* head) {
    if (!head) {
        cout << "The list is empty." << endl;
        return;
    }

    Node* current = head;
    do {
        cout << current->data << " ";
        current = current->next;
    } while (current != head); // Loop until we circle back to the head
    cout << endl;
}

// Main function to test the implementation
int main() {
    Node* head = nullptr;

    // Manually create a circular linked list
    head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);
    head->next->next->next = new Node(40);
    head->next->next->next->next = head;  // Last node points back to head

    cout << "Initial list: ";
    printList(head);

    // Delete a node at a specific position
    deleteNodeAtPosition(head, 2);
    cout << "After deleting node at position 2: ";
    printList(head);

    // Delete a node at position 0 (head)
    deleteNodeAtPosition(head, 0);
    cout << "After deleting node at position 0: ";
    printList(head);

    // Try to delete a node at an out-of-bounds position
    deleteNodeAtPosition(head, 5);
    cout << "After attempting to delete node at position 5: ";
```

```
    printList(head);

    // Clean up the remaining nodes
    while (head) {
        deleteNodeAtPosition(head, 0); // Delete all nodes by position 0 (head)
    }

    cout << "After deleting all nodes: ";
    printList(head);

    return 0;
}
```

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

Code:

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

// Node definition for a circular linked list
struct Node {
    int data;
    Node* next;

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

// Function to delete a node by its value in a circular linked list
void deleteNodeByValue(Node*& head, int value) {
    if (!head) {
        cout << "The list is empty! Cannot delete the node with value " << value << "." << endl;
        return;
    }

    // Case 1: If position is 0, delete the head node
    if (head->data == value && head->next == head) { // Only one node in the list
        delete head;
        head = nullptr;
        cout << "The only node in the list with value " << value << " has been deleted." << endl;
        return;
    }
```

```
    // Case 2: If the node to delete is the head node
    if (head->data == value) {
        // Traverse to find the last node
        Node* last = head;
        while (last->next != head) {
            last = last->next;
        }

        // Update the head and adjust the last node's next pointer
        last->next = head->next; // The last node's next should point to the new head
        Node* temp = head;
        head = head->next;  // Update head to the next node
        delete temp;        // Free the memory of the old head
        cout << "Node with value " << value << " has been deleted." << endl;
        return;
    }

    // Case 3: Deleting a node in the middle or end
    Node* current = head;
    Node* previous = nullptr;

    while (current->data != value && current->next != head) {
        previous = current;
        current = current->next;
    }

    if (current->data == value) {
        previous->next = current->next; // Skip the node to delete
        delete current;
        cout << "Node with value " << value << " has been deleted." << endl;
    } else {
        cout << "Node with value " << value << " not found in the list." << endl;
    }
}

// Function to print the circular linked list (forward traversal)
void printList(Node* head) {
    if (!head) {
        cout << "The list is empty." << endl;
        return;
    }
```

```
    Node* current = head;
    do {
        cout << current->data << " ";
        current = current->next;
    } while (current != head); // Loop until we circle back to the head
    cout << endl;
}

// Main function to test the implementation
int main() {
    Node* head = nullptr;

    // Manually create a circular linked list
    head = new Node(10);
    head->next = new Node(20);
    head->next->next = new Node(30);
    head->next->next->next = new Node(40);
    head->next->next->next->next = head;  // Last node points back to head

    cout << "Initial list: ";
    printList(head);

    // Delete a node by its value
    deleteNodeByValue(head, 30);
    cout << "After deleting node with value 30: ";
    printList(head);

    // Delete the head node
    deleteNodeByValue(head, 10);
    cout << "After deleting node with value 10 (head): ";
    printList(head);

    // Try to delete a node that doesn't exist
    deleteNodeByValue(head, 100);
    cout << "After attempting to delete node with value 100: ";
    printList(head);

    // Clean up the remaining nodes
    while (head) {
        deleteNodeByValue(head, head->data); // Delete all nodes by value
```

```
    }

    cout << "After deleting all nodes: ";
    printList(head);

    return 0;
}
```

Binary Search Tree:

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

Code:

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

// Definition of a node in the BST
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    // Constructor
    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Function to insert a new node into the BST
TreeNode* insert(TreeNode* root, int value) {
    if (root == nullptr) {
        return new TreeNode(value);
    }

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

    return root;
}

// Function to count all the nodes in the BST
int countNodes(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

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

// Main function
```

```
int main() {
    TreeNode* root = nullptr;

    // Insert values into the BST
    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);

    // Count and display the number of nodes
    cout << "Total number of nodes in the BST: " << countNodes(root) << endl;

    return 0;
}
```

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

Code:

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

// Definition of a node in the BST
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    // Constructor
    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Function to insert a new node into the BST
TreeNode* insert(TreeNode* root, int value) {
    if (root == nullptr) {
        return new TreeNode(value);
    }

    if (value < root->value) {
        root->left = insert(root->left, value);
```

```
    } else if (value > root->value) {
        root->right = insert(root->right, value);
    }

    return root;
}

// Function to search for a specific value in the BST
bool search(TreeNode* root, int value) {
    if (root == nullptr) {
        return false; // Value not found
    }

    if (value == root->value) {
        return true; // Value found
    } else if (value < root->value) {
        return search(root->left, value); // Search in the left subtree
    } else {
        return search(root->right, value); // Search in the right subtree
    }
}

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

    // Insert values into the BST
    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);

    // Search for specific values
    int searchValue = 40;
    if (search(root, searchValue)) {
        cout << "Value " << searchValue << " found in the BST." << endl;
    } else {
        cout << "Value " << searchValue << " not found in the BST." << endl;
```

```
    }

    return 0;
}
```

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

Code:

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

// Definition of a node in the BST
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    // Constructor
    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Function to insert a new node into the BST
TreeNode* insert(TreeNode* root, int value) {
    if (root == nullptr) {
        return new TreeNode(value);
    }

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

    return root;
}

// In-order Traversal (Left -> Root -> Right)
void inOrder(TreeNode* root) {
    if (root == nullptr) return;
    inOrder(root->left);
    cout << root->value << " ";
    inOrder(root->right);
```

```
}

// Pre-order Traversal (Root -> Left -> Right)
void preOrder(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->value << " ";
    preOrder(root->left);
    preOrder(root->right);
}

// Post-order Traversal (Left -> Right -> Root)
void postOrder(TreeNode* root) {
    if (root == nullptr) return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->value << " ";
}

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

    // Insert values into the BST
    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);

    // Perform tree traversals
    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;
}
```

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

Code:

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

// Definition of a node in the BST
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    // Constructor
    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Function to insert a new node into the BST
TreeNode* insert(TreeNode* root, int value) {
    if (root == nullptr) {
        return new TreeNode(value);
    }

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

    return root;
}

// Reverse in-order traversal (Right -> Root -> Left)
void reverseInOrder(TreeNode* root) {
    if (root == nullptr) return;

    // Visit the right subtree first
```

```
    reverseInOrder(root->right);

    // Visit the root
    cout << root->value << " ";

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

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

    // Insert values into the BST
    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);

    // Perform reverse in-order traversal
    cout << "Reverse In-order Traversal: ";
    reverseInOrder(root);
    cout << endl;

    return 0;
}
```

- **Write a program to check if there are duplicate values in a binary search tree.**

Code:

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

// Definition of a node in the BST
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;
```

```cpp
    // Constructor
    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Function to insert a new node into the BST
TreeNode* insert(TreeNode* root, int value) {
    if (root == nullptr) {
        return new TreeNode(value);
    }

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

    return root;
}

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

    // Check the left subtree
    if (hasDuplicates(root->left, seen)) return true;

    // Check the current node
    if (seen.count(root->value)) {
        return true; // Duplicate found
    }
    seen.insert(root->value);

    // Check the right subtree
    return hasDuplicates(root->right, seen);
}

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

```
    // Insert values into the BST
    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);
    // Insert a duplicate value
    root = insert(root, 40);

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

    return 0;
}
```

- **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;

// Definition of a node in the BST
struct TreeNode {
    int value;
    TreeNode* left;
    TreeNode* right;

    // Constructor
    TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

// Function to insert a new node into the BST
TreeNode* insert(TreeNode* root, int value) {
    if (root == nullptr) {
        return new TreeNode(value);
```

```
    }

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

    return root;
}

// Helper function to find the minimum value in a subtree
TreeNode* findMin(TreeNode* root) {
    while (root && root->left != nullptr) {
        root = root->left;
    }
    return root;
}

// Function to delete a node from the BST
TreeNode* deleteNode(TreeNode* root, int value) {
    if (root == nullptr) {
        return root; // Node not found
    }

    if (value < root->value) {
        root->left = deleteNode(root->left, value); // Search in the left subtree
    } else if (value > root->value) {
        root->right = deleteNode(root->right, value); // Search in the right subtree
    } else {
        // Node to be deleted found
        if (root->left == nullptr && root->right == nullptr) {
            // Case 1: Leaf node
            delete root;
            return nullptr;
        } else if (root->left == nullptr) {
            // Case 2: Node with one child (right child)
            TreeNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
```

```
        // Case 2: Node with one child (left child)
        TreeNode* temp = root->left;
        delete root;
        return temp;
    } else {
        // Case 3: Node with two children
        TreeNode* temp = findMin(root->right); // Find in-order successor
        root->value = temp->value; // Replace value with successor's value
        root->right = deleteNode(root->right, temp->value); // Delete successor
    }
  }
  return root;
}

// In-order traversal to display the BST
void inOrder(TreeNode* root) {
  if (root == nullptr) return;
  inOrder(root->left);
  cout << root->value << " ";
  inOrder(root->right);
}

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

  // Insert values into the BST
  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 BST (In-order): ";
  inOrder(root);
  cout << endl;

  // Delete a leaf node
  root = deleteNode(root, 20);
```

```
    cout << "After deleting leaf node (20): ";
    inOrder(root);
    cout << endl;

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

    // Delete a node with two children
    root = deleteNode(root, 50);
    cout << "After deleting node with two children (50): ";
    inOrder(root);
    cout << endl;

    return 0;
}
```