DS. FINAL. ASSINGMENT

Doubly Linked List:

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

CODE:

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

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

// Function to insert a node at the front
void insertFront(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    newNode->prev = nullptr;

    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
}

// Function to delete the first node
void deleteFirstNode(Node*& head) {
    if (head == nullptr) {  // List is empty
        cout << "List is already empty.\n";
        return;
    }

    Node* temp = head;  // Store the node to delete
    head = head->next;  // Move head to the next node

    if (head != nullptr) {
        head->prev = nullptr;  // Remove the backward link
```

```
    }

    delete temp;  // Free memory
    cout << "First node deleted.\n";
}

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes
    insertFront(head, 10);
    insertFront(head, 20);
    insertFront(head, 30);

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

    // Delete the first node
    deleteFirstNode(head);

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

    return 0;
}
```

OUTPUT:

```
Original list: 30 20 10
First node deleted.
Updated list: 20 10
```

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

CODE:

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

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

// Function to insert a node at the end
void insertEnd(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        newNode->prev = nullptr;
        head = newNode;
        return;
    }

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

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

// Function to delete the last node
void deleteLastNode(Node*& head) {
```

```
    if (head == nullptr) {  // List is empty
        cout << "List is already empty.\n";
        return;
    }

    if (head->next == nullptr) {  // Only one node in the list
        delete head;
        head = nullptr;
        cout << "Last node deleted.\n";
        return;
    }

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

    temp->prev->next = nullptr;  // Remove the link to the last node
    delete temp;  // Free memory
    cout << "Last node deleted.\n";
}

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes
    insertEnd(head, 10);
    insertEnd(head, 20);
    insertEnd(head, 30);

    cout << "Original list: ";
```

```
    displayList(head);

    // Delete the last node
    deleteLastNode(head);

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

    return 0;
}
```

OUTPUT:

```
Original list: 10 20 30
Last node deleted.
Updated list: 10 20
```

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

<u>CODE:</u>

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

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

// Function to insert a node at the end
void insertEnd(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        newNode->prev = 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 by its value
void deleteNodeByValue(Node*& head, int value) {
    if (head == nullptr) {  // List is empty
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;

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

    if (temp == nullptr) {  // Value not found
        cout << "Node with value " << value << " not found.\n";
        return;
    }

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

```
    delete temp;  // Free memory
    cout << "Node with value " << value << " deleted.\n";
}

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes
    insertEnd(head, 11);
    insertEnd(head, 22);
    insertEnd(head, 33);

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

    // Delete a node by value
    deleteNodeByValue(head, 22);

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

    return 0;
}
```

OUTPUT:

```
Original list: 11 22 33
Node with value 22 deleted.
Updated list: 11 33
```

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;

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

// Function to insert a node at the end
void insertEnd(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        newNode->prev = 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
void deleteNodeAtPosition(Node*& head, int position) {
```

```
    if (head == nullptr) {  // List is empty
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;

    // Traverse to the desired position
    for (int i = 0; temp != nullptr && i < position; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) {  // Position is out of range
        cout << "Position is out of range.\n";
        return;
    }

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

    delete temp;  // Free memory
    cout << "Node at position " << position << " deleted.\n";
}

// Function to display the list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
```

```
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes
    insertEnd(head, 1);
    insertEnd(head, 2);
    insertEnd(head, 3);
    insertEnd(head, 4);
    insertEnd(head, 5);

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

    // Delete a node at a specific position (e.g., position 2)
    deleteNodeAtPosition(head, 3);

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

    return 0;
}
```

OUTPUT:

```
Original list: 1 2 3 4 5
Node at position 3 deleted.
Updated list: 1 2 3 5
```

5. *After deleting a node, how will you write the forward and reverse traversal Functions?*

CODE:

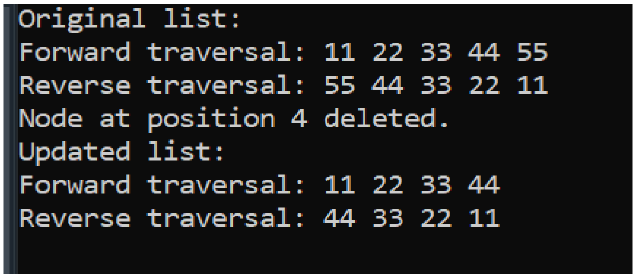
```
#include <iostream>
using namespace std;

// Node structure
struct Node {
```

```
    int data;
    Node* next;
    Node* prev;
};

// Function to insert a node at the end
void insertEnd(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;

    if (head == nullptr) {
        newNode->prev = 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
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {  // List is empty
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;

    // Traverse to the desired position
    for (int i = 0; temp != nullptr && i < position; i++) {
        temp = temp->next;
    }

    if (temp == nullptr) {  // Position is out of range
```

```
        cout << "Position is out of range.\n";
        return;
    }

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

    delete temp;  // Free memory
    cout << "Node at position " << position << " deleted.\n";
}

// Function for forward traversal (left to right)
void forwardTraversal(Node* head) {
    Node* temp = head;
    cout << "Forward traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << "\n";
}

// Function for reverse traversal (right to left)
void reverseTraversal(Node* head) {
    // Traverse to the last node first
    if (head == nullptr) return;
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
```

```
    // Now traverse backwards from the last node
    cout << "Reverse traversal: ";
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->prev;
    }
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes
    insertEnd(head, 11);
    insertEnd(head, 22);
    insertEnd(head, 33);
    insertEnd(head, 44);
    insertEnd(head, 55);

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

    // Delete a node at a specific position (e.g., position 4)
    deleteNodeAtPosition(head, 4);

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

    return 0;
}
```

OUTPUT:



Circular Linked List:

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

CODE:

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

// Node structure for Circular Linked List
struct Node {
    int data;
    Node* next;
};

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

    if (head == nullptr) {  // If the list is empty
        head = newNode;
        newNode->next = head;  // Point to itself (circular)
    } else {
        Node* temp = head;
        while (temp->next != head) {  // Traverse to the last node
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;  // Connect the new node to the head (circular)
    }
}

// Function to delete the first node in the circular linked list
```

```
void deleteFirstNode(Node*& head) {
    if (head == nullptr) {  // List is empty
        cout << "List is already empty.\n";
        return;
    }

    if (head->next == head) {  // Only one node in the list
        delete head;
        head = nullptr;
        cout << "First node deleted, list is now empty.\n";
        return;
    }

    // If more than one node, delete the first node
    Node* temp = head;
    while (temp->next != head) {  // Find the last node
        temp = temp->next;
    }

    temp->next = head->next;  // Last node now points to second node
    delete head;  // Delete the first node
    head = temp->next;  // Update head to the second node
    cout << "First node deleted.\n";
}

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

    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << "\n";
}

int main() {
```

```
    Node* head = nullptr;

    // Insert nodes into the circular linked list
    insertEnd(head, 1);
    insertEnd(head, 2);
    insertEnd(head, 3);

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

    // Delete the first node
    deleteFirstNode(head);

    cout << "Updated circular linked list: ";
    displayList(head);

    return 0;
}
```

OUTPUT:

```
Original circular linked list: 1 2 3
First node deleted.
Updated circular linked list: 2 3
```

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

CODE:

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

// Node structure for Circular Linked List
struct Node {
    int data;
    Node* next;
};

// Function to insert a node at the end of the circular linked list
void insertEnd(Node*& head, int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = nullptr;
```

```
    if (head == nullptr) {  // If the list is empty
        head = newNode;
        newNode->next = head;  // Point to itself (circular)
    } else {
        Node* temp = head;
        while (temp->next != head) {  // Traverse to the last node
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;  // Connect the new node to the head (circular)
    }
}

// Function to delete the last node in the circular linked list
void deleteLastNode(Node*& head) {
    if (head == nullptr) {  // List is empty
        cout << "List is already empty.\n";
        return;
    }

    if (head->next == head) {  // Only one node in the list
        delete head;
        head = nullptr;
        cout << "Last node deleted, list is now empty.\n";
        return;
    }

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

    // Delete the last node
    Node* lastNode = temp->next;
    temp->next = head;  // Second last node now points to the head
    delete lastNode;    // Free memory for the last node

    cout << "Last node deleted.\n";
}
```

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

    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes into the circular linked list
    insertEnd(head, 100);
    insertEnd(head, 200);
    insertEnd(head, 300);

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

    // Delete the last node
    deleteLastNode(head);

    cout << "Updated circular linked list: ";
    displayList(head);

    return 0;
}
```

OUTPUT:

```
Original circular linked list: 100 200 300
Last node deleted.
Updated circular linked list: 100 200
```

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

CODE:

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

// Node structure for Circular Linked List
struct Node {
    int data;
    Node* next;
};

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

    if (head == nullptr) {  // If the list is empty
        head = newNode;
        newNode->next = head;  // Point to itself (circular)
    } else {
        Node* temp = head;
        while (temp->next != head) {  // Traverse to the last node
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;  // Connect the new node to the head (circular)
    }
}

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

```
    }

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

    // If the node to be deleted is the head node
    if (head->data == value) {
        if (head->next == head) {  // Only one node in the list
            delete head;
            head = nullptr;
            cout << "Node with value " << value << " deleted, list is now empty.\n";
            return;
        } else {
            // Find the last node
            while (temp->next != head) {
                temp = temp->next;
            }
            // Last node now points to the second node
            temp->next = head->next;
            delete head;
            head = temp->next;  // Update head to the second node
            cout << "Node with value " << value << " deleted.\n";
            return;
        }
    }

    // Traverse the list to find the node with the given value
    do {
        prev = temp;
        temp = temp->next;
        if (temp->data == value) {
            prev->next = temp->next;  // Bypass the node
            delete temp;  // Free memory
            cout << "Node with value " << value << " deleted.\n";
            return;
        }
    } while (temp != head);

    cout << "Node with value " << value << " not found.\n";
}
```

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

    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes into the circular linked list
    insertEnd(head, 2);
    insertEnd(head, 4);
    insertEnd(head, 6);
    insertEnd(head, 8);
    insertEnd(head, 10);

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

    // Delete a node by value
    deleteNodeByValue(head, 6);

    cout << "Updated circular linked list: ";
    displayList(head);

    // Attempt to delete a node that doesn't exist
    deleteNodeByValue(head, 100);

    return 0;
}
```

OUTPUT:

```
Original circular linked list: 2 4 6 8 10
Node with value 6 deleted.
Updated circular linked list: 2 4 8 10
Node with value 100 not found.
```

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 for Circular Linked List
struct Node {
    int data;
    Node* next;
};

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

    if (head == nullptr) {  // If the list is empty
        head = newNode;
        newNode->next = head;  // Point to itself (circular)
    } else {
        Node* temp = head;
        while (temp->next != head) {  // Traverse to the last node
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;  // Connect the new node to the head (circular)
    }
}

// Function to delete a node at a specific position in the circular linked list
void deleteNodeAtPosition(Node*& head, int position) {
    if (head == nullptr) {  // List is empty
        cout << "List is empty.\n";
```

```
    return;
}

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

// If the node to be deleted is the head node
if (position == 0) {
    if (head->next == head) {  // Only one node in the list
        delete head;
        head = nullptr;
        cout << "Node at position " << position << " deleted, list is now empty.\n";
        return;
    } else {
        // Find the last node
        while (temp->next != head) {
            temp = temp->next;
        }
        // Last node now points to the second node
        temp->next = head->next;
        delete head;
        head = temp->next;  // Update head to the second node
        cout << "Node at position " << position << " deleted.\n";
        return;
    }
}

// Traverse to the node at the specified position
for (int i = 0; temp->next != head && i < position; i++) {
    prev = temp;
    temp = temp->next;
}

if (temp == head) {
    cout << "Position is out of range.\n";
    return;
}

// Delete the node at the specified position
prev->next = temp->next;  // Bypass the node
delete temp;  // Free memory
```

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

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

    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes into the circular linked list
    insertEnd(head, 10);
    insertEnd(head, 20);
    insertEnd(head, 30);
    insertEnd(head, 40);
    insertEnd(head, 50);

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

    // Delete a node at a specific position (e.g., position 4)
    deleteNodeAtPosition(head, 4);

    cout << "Updated circular linked list: ";
    displayList(head);

    return 0;
}
```

OUTPUT:

```
Original circular linked list: 10 20 30 40 50
Node at position 4 deleted.
Updated circular linked list: 10 20 30 40
```

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 for Circular Linked List
struct Node {
    int data;
    Node* next;
};

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

    if (head == nullptr) {  // If the list is empty
        head = newNode;
        newNode->next = head;  // Point to itself (circular)
    } else {
        Node* temp = head;
        while (temp->next != head) {  // Traverse to the last node
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;  // Connect the new node to the head (circular)
    }
}

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

```
    }

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

    // If the node to be deleted is the head node
    if (head->data == value) {
        if (head->next == head) {  // Only one node in the list
            delete head;
            head = nullptr;
            cout << "Node with value " << value << " deleted, list is now empty.\n";
            return;
        } else {
            // Find the last node
            while (temp->next != head) {
                temp = temp->next;
            }
            // Last node now points to the second node
            temp->next = head->next;
            delete head;
            head = temp->next;  // Update head to the second node
            cout << "Node with value " << value << " deleted.\n";
            return;
        }
    }

    // Traverse the list to find the node with the given value
    do {
        prev = temp;
        temp = temp->next;
        if (temp->data == value) {
            prev->next = temp->next;  // Bypass the node
            delete temp;  // Free memory
            cout << "Node with value " << value << " deleted.\n";
            return;
        }
    } while (temp != head);

    cout << "Node with value " << value << " not found.\n";
}
```

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

    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);  // Traverse until we circle back to head
    cout << "\n";
}

int main() {
    Node* head = nullptr;

    // Insert nodes into the circular linked list
    insertEnd(head, 1);
    insertEnd(head, 2);
    insertEnd(head, 3);
    insertEnd(head, 4);
    insertEnd(head, 5);

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

    // Delete a node by value (e.g., delete node with value 4)
    deleteNodeByValue(head, 4);

    cout << "Updated circular linked list after deletion: ";
    displayList(head);

    return 0;
}
```

OUTPUT:

```
Original circular linked list: 1 2 3 4 5
Node with value 4 deleted.
Updated circular linked list after deletion: 1 2 3 5
```

Binary Search Tree:

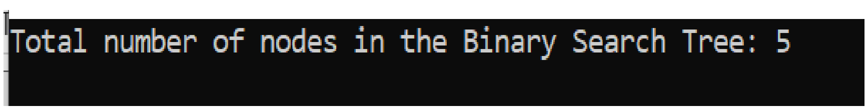
1. *Write a program to count all the nodes in a binary search tree.*

CODE:

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

// Node structure for 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 node in the Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // If the tree is empty, create a new node
    }

    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}
```

```
// Function to count the total number of nodes in the BST
int countNodes(Node* root) {
    if (root == nullptr) {
        return 0;  // If the node is null, return 0
    }

    // Count the current node + nodes in left and right subtrees
    return 1 + countNodes(root->left) + countNodes(root->right);
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the Binary Search Tree
    root = insert(root, 20);
    insert(root, 10);
    insert(root, 30);
    insert(root, 40);
    insert(root, 50);

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

    return 0;
}
```

OUTPUT:



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

<u>CODE:</u>

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

// Node structure for 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 node in the Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // If the tree is empty, create a new node
    }

    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}

// Function to search for a specific value in the BST
bool search(Node* root, int value) {
    if (root == nullptr) {
        return false;  // If the tree is empty or reached a leaf node
    }

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

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

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

    // Insert nodes into the Binary Search Tree
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    int value = 40;  // Value to search for
    if (search(root, value)) {
        cout << "Value " << value << " found in the Binary Search Tree." << endl;
    } else {
        cout << "Value " << value << " not found in the Binary Search Tree." << endl;
    }

    return 0;
}
```

OUTPUT:

```
Value 40 found in the Binary Search Tree.
```

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 for 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 node in the Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // If the tree is empty, create a new node
    }

    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}

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

    inOrder(root->left);    // Traverse left subtree
    cout << root->data << " ";  // Visit the root
    inOrder(root->right);   // Traverse right subtree
}

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

    cout << root->data << " ";  // Visit the root
    preOrder(root->left);    // Traverse left subtree
```

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

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

    postOrder(root->left);   // Traverse left subtree
    postOrder(root->right);  // Traverse right subtree
    cout << root->data << " ";  // Visit the root
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the Binary Search Tree
    root = insert(root, 500);
    insert(root, 100);
    insert(root, 200);
    insert(root, 400);
    insert(root, 600);

    // Perform In-Order Traversal
    cout << "In-Order Traversal: ";
    inOrder(root);
    cout << endl;

    // Perform Pre-Order Traversal
    cout << "Pre-Order Traversal: ";
    preOrder(root);
    cout << endl;

    // Perform Post-Order Traversal
    cout << "Post-Order Traversal: ";
    postOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT:

```
In-Order Traversal: 100 200 400 500 600
Pre-Order Traversal: 500 100 200 400 600
Post-Order Traversal: 400 200 100 600 500
```

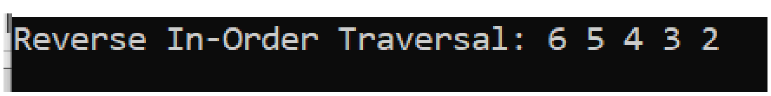
4. *How will you write reverse in-order traversal for a binary search tree? Show it in code.*

<u>CODE:</u>

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

// Node structure for 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 node in the Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // If the tree is empty, create a new node
    }

    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}
```

```
// Function for Reverse In-Order Traversal (Right, Root, Left)
void reverseInOrder(Node* root) {
    if (root == nullptr) {
        return;
    }

    reverseInOrder(root->right);  // Traverse right subtree
    cout << root->data << " ";    // Visit the root
    reverseInOrder(root->left);   // Traverse left subtree
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the Binary Search Tree
    root = insert(root, 5);
    insert(root, 3);
    insert(root, 2);
    insert(root, 4);
    insert(root, 6);

    // Perform Reverse In-Order Traversal
    cout << "Reverse In-Order Traversal: ";
    reverseInOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT:



5. *Write a program to check if there are duplicate values in a binary search Tree.*

CODE:

```
#include <iostream>
#include <unordered_set>  // For storing values to detect duplicates
using namespace std;

// Node structure for 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 node in the Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // If the tree is empty, create a new node
    }

    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}

// Function to check for duplicate values using in-order traversal
bool checkDuplicates(Node* root, unordered_set<int>& values) {
    if (root == nullptr) {
        return false;  // Base case: empty node, no duplicates
    }

    // Traverse the left subtree
```

```
    if (checkDuplicates(root->left, values)) {
        return true;  // If a duplicate is found in the left subtree
    }

    // Check if the current node's value already exists in the set
    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;

    // Insert nodes into the Binary Search Tree
    root = insert(root, 15);
    insert(root, 13);
    insert(root, 12);
    insert(root, 14);
    insert(root, 16);
    insert(root, 17);
    insert(root, 12);  // Duplicate value

    // Set to keep track of node values during traversal
    unordered_set<int> values;

    // Check if there are any duplicate values in the BST
    if (checkDuplicates(root, values)) {
        cout << "The Binary Search Tree contains duplicate values." << endl;
    } else {
        cout << "The Binary Search Tree does not contain duplicate values." << endl;
    }

    return 0;
}
```

OUTPUT:

```
The Binary Search Tree contains duplicate values.
```

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 for 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 node in the Binary Search Tree
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return new Node(value);  // If the tree is empty, create a new node
    }

    if (value < root->data) {
        root->left = insert(root->left, value);  // Insert in the left subtree
    } else {
        root->right = insert(root->right, value);  // Insert in the right subtree
    }

    return root;
}

// Function to find the minimum value node in the BST (in-order successor)
```

```
Node* findMin(Node* root) {
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}

// Function to delete a node in the Binary Search Tree
Node* deleteNode(Node* root, int value) {
    // Base case: if the tree is empty
    if (root == nullptr) {
        return root;
    }

    // If the value to be deleted is smaller than the root's data, go left
    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    }
    // If the value to be deleted is larger than the root's data, go right
    else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    }
    // If the value to be deleted is the root's data
    else {
        // Case 1: Node to be deleted has no children (leaf node)
        if (root->left == nullptr && root->right == nullptr) {
            delete root;
            return nullptr;
        }
        // Case 2: Node to be deleted 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 to be deleted has two children
        else {
```

```
            // Find the in-order successor (minimum value in the right subtree)
            Node* temp = findMin(root->right);
            // Copy the in-order successor's data to the root
            root->data = temp->data;
            // Delete the in-order successor
            root->right = deleteNode(root->right, temp->data);
        }
    }

    return root;
}

// Function for In-Order Traversal
void inOrder(Node* root) {
    if (root == nullptr) {
        return;
    }

    inOrder(root->left);    // Traverse left subtree
    cout << root->data << " ";  // Visit the root
    inOrder(root->right);   // Traverse right subtree
}

int main() {
    Node* root = nullptr;

    // Insert nodes into the Binary Search Tree
    root = insert(root, 22);
    insert(root, 11);
    insert(root, 33);
    insert(root, 44);
    insert(root, 55);

    cout << "Original BST (In-Order): ";
    inOrder(root);
    cout << endl;

    // Deleting a leaf node
    root = deleteNode(root, 22);  // Leaf node
    cout << "After deleting 20 (Leaf Node): ";
    inOrder(root);
```

```
    cout << endl;

    // Deleting a node with one child
    root = deleteNode(root, 44);
    cout << "After deleting 44 (Node with one child): ";
    inOrder(root);
    cout << endl;

    // Deleting a node with two children
    root = deleteNode(root, 33);
    cout << "After deleting 33 (Node with two children): ";
    inOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT:

```
Original BST (In-Order): 11 22 33 44 55
After deleting 20 (Leaf Node): 11 33 44 55
After deleting 44 (Node with one child): 11 33 55
After deleting 33 (Node with two children): 11 55
```

THE END.