Analysis:

Insert:

	Ordered	**Unordered**
Best Case	**O(log n)**	**O(1)**
Worse Case	**O(n)**	**O(1)**

Search:

	Ordered	**Unordered**
Best Case	**O(log n)**	**O(n)**
Worse Case	**O(n)**	**O(n)**

Functions:

Creates Node and Defines Left and RIght Branch:

```
class Node {
public:
    int value;
    Node* left;
    Node* right;

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

Add function:

```
    BinarySearchTree() : root(nullptr) {}

    void add(int value) {
        if (root == nullptr) {
            root = new Node(value);
        } else {
            add(root, value);
        }
    }
```

Logic:

```
    void add(Node* node, int value) {
        if (value <= node->value) {
            if (node->left == nullptr) {
                node->left = new Node(value);
            } else {
                add(node->left, value);
            }
        } else {
            if (node->right == nullptr) {
                node->right = new Node(value);
            } else {
                add(node->right, value);
            }
        }
    }
```

Remove Function:

```
void remove(int value) {
    root = remove(root, value);
}
```

Logic:

```
Node* remove(Node* node, int value) {
    if (node == nullptr) {
        return node;
    }
    if (value < node->value) {
        node->left = remove(node->left, value);
    } else if (value > node->value) {
        node->right = remove(node->right, value);
    } else {
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        }
        Node* temp = findMin(node->right);
        node->value = temp->value;
        node->right = remove(node->right, temp->value);
    }
    return node;
}
```

Minimum Value Function:

```
int findMin() {
    Node* minNode = findMin(root);
    return minNode ? minNode->value : -1; // Return -1 if the tree is empty
}
```

Logic:

```cpp
    Node* findMin(Node* node) {
        Node* current = node;
        while (current && current->left != nullptr) {
            current = current->left;
        }
        return current;
    }
```

Inorder Transversal Function:

```cpp
std::vector<int> inOrderTraversal() {
    std::vector<int> result;
    inOrderTraversal(root, result);
    return result;
}
```

Logic:

```cpp
    void inOrderTraversal(Node* node, std::vector<int>& result) {
        if (node != nullptr) {
            inOrderTraversal(node->left, result);
            result.push_back(node->value);
            inOrderTraversal(node->right, result);
        }
    }
```

Tests:

Add:

Test 1 (add to empty tree):

```cpp
void testAddFunction() {
    BinarySearchTree bst;
    bst.add(10);
    assert(bst.inOrderTraversal() == std::vector<int>{10});
```

Test 2 (add three values and make BST):

```
    bst.add(5);
    bst.add(15);
    assert((bst.inOrderTraversal() == std::vector<int>{5, 10, 15}));

}
```

Remove:

Test 1 (Remove leaf):

```
    BinarySearchTree bst;
    bst.add(10);
    bst.add(5);
    bst.add(15);
    bst.remove(5);
    assert((bst.inOrderTraversal() == std::vector<int>{10, 15}));

```

Test 2 (remove node with one child):

```
    bst.add(5);
    bst.add(12);
    bst.remove(15);
    assert((bst.inOrderTraversal() == std::vector<int>{5, 10, 12}));

}
```

Min:

```cpp
void testFindMin() {
    BinarySearchTree bst;
    bst.add(10);
    bst.add(5);
    bst.add(15);
    bst.add(3);
    bst.add(7);
    assert(bst.findMin() == 3);

    bst.remove(3);
    assert(bst.findMin() == 5);

    bst.remove(5);
    assert(bst.findMin() == 7);
}
```

Inorder:

```cpp
void testInOrderTraversal() {
    BinarySearchTree bst;
    bst.add(10);
    bst.add(5);
    bst.add(15);
    assert((bst.inOrderTraversal() == std::vector<int>{5, 10, 15}));

    bst.add(3);
    bst.add(7);
    bst.add(12);
    bst.add(20);
    assert((bst.inOrderTraversal() == std::vector<int>{3, 5, 7, 10, 12, 15, 20}));
}
```