**Software Development: Data Structures and Performance (SCQF Level 8)**

**HL9T 35**

**Assessment**

**Outcome 1 Part 1 of 2**

#### Creative Industries

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Student** | | Kamil Milej | | | | | | |
| **I.D.** | | 2264629 | | | | | | |
| **Date** | | 09/05/2025 | | | | | | |
|  | **Pass** | |  | **Fail** |  | **Remediation** |  |  |
| **Tutor** | |  | | | | | | |

#### Creative Industries

***Binary Search Tree Implementation Documentation***

* *Algorithm Description*

This Assessment was about the correct implementation of a binary search tree, where we use a dataset which have 31 values, one optimal set and one worst-case set. To do that we use one header class called BSTree.h where we implement one class called BSTree and two normal file BSTree.cpp and main.cpp. Goal of this structure is to generate 31 value put this value in a variable, next use it as a parameter in the insertDataSet method with tree. Tree is our most important object which was declare here BSTree tree;. When our node in bst have a value and pointers to left and right children (left, right). The constructor sets the value and initializes the pointers as nullptr.Then set a root from the insert function with value of 1st of our dataset.

Insert() public function to insert value, starts recursion from root value. Recursive function navigates left/right to find correct place for the new value and inserts it.

Search() These two search functions recursively check if a given value exists in the binary search tree, counting the number of steps taken to find it or determine it's not present.

Clear() added to clean BSTree after use on optional – not needed

inOrderTraversal() and InOrder() which is inside inordertraversal() These two functions perform an in-order traversal of the binary search tree, which recursively visits the left subtree, prints the current node’s value, then visits the right subtree and resulting in the values being printed in sorted order.

* *Data Structure used*

The data structure required for this task is a Binary Search Tree. It is a dynamic data structure where each node has up to two children. The left child has value less than that of the parent. And the right child value is higher the the parents value. This ordering allows for efficient searching, insertion, and deletion operations. Also, BST maintain elements in a sorted order, enabling in-order traversal to access elements sequentially

* *Test complexity analysis*

to evaluate the performance of our binary search tree, we create two data sets, optional and worstcare.  
1. Optional is just rundom number to see how it will work in normal flow.  
2. WorstCase where we use value from 1 to 31 when we know all will go in right and and much steps its needed to do this.

* *Time Complexity Analysis*

The time complexity depends on the dataset and how many steps are needed to insert values into the BSTree. With the Optimal Dataset, only 129 steps are required for insertion and search, which reflects efficient O(log n) behavior thanks to a balanced tree. In the Worst Case (sorted input), the tree becomes a right-skewed list, and insertion takes 496 steps, resulting in O(n) time complexity. This shows that BST performance strongly depends on input order — unbalanced trees are inefficient for sorted data.

* *Space Complexity*

Space complexity of the BSTree is as each inserted value creates a new node. Also, recursive operations like insert and search use stack memory. In the worst case unbalanced tree, recursion can reach 0(n), while in the best case balanced value tree it’s only 0(log n).