Nottingham Trent University
Department of Computer Science
Advanced Software Engineering
SOFT37002

November 2024

Complexity Analysis of Binary Search Trees

Word count: 1916

Hannah Jones
N1039942

Contents

1 Introduction

This report presents a detailed complexity analysis of the public operations of the Binary Search Tree (BST) implemented in C++. The time complexities of operations such as insertion, lookup, deletion and tree traversal are analysed using Big-O and other asymptotic notations. Each operation is analysed for its best, average and worst-case scenarios, with particular attention to their computational efficiency. Additionally, an operation to calculate the height of the tree is developed and examined for its time complexity.

2 Analysis

2.1 Overview

	Method declaration	Best case	Average case	Worst case
Constructor	`Dictionary()`	$\Omega(1)$	$\Theta(1)$	$O(1)$
Deconstructor	`~Dictionary()`	$\Omega(n)$	$\Theta(n)$	$O(n)$
Iterative insertion	`void insert(int, string)`	$\Omega(\log_2(n))$	$\Theta(\log_2(n))$	$O(n)$
Recursive insertion	`void insert_rec(int, string)`	$\Omega(\log_2(n))$	$\Theta(\log_2(n))$	$O(n)$
Iterative look-up	`string* lookup(int)`	$\Omega(1)$	$\Theta(\log_2(n))$	$O(n)$
Recursive look-up	`string* lookup_rec(int)`	$\Omega(1)$	$\Theta(\log_2(n))$	$O(n)$
Removal	`void remove(int)`	$\Omega(1)$	$\Theta(\log_2(n))$	$O(n)$
Display tree	`void displayTree()`	$\Omega(n)$	$\Theta(n)$	$O(n)$
Display entries	`void displayEntries()`	$\Omega(n)$	$\Theta(n)$	$O(n)$
Copy constructor	`Dictionary(const Dictionary&)`	$\Omega(n)$	$\Theta(n)$	$O(n)$
Copy assignment operator	`Dictionary & operator=(const Dictionary&)`	$\Omega(1)$	$\Theta(n)$	$O(n)$
Move constructor	`Dictionary(Dictionary&&)`	$\Omega(1)$	$\Theta(1)$	$O(1)$
Move assignment operator	`Dictionary & operator=(Dictionary&&)`	$\Omega(1)$	$\Theta(n)$	$O(n)$
Right rotation	`void rotateRightOn()`	$\Omega(1)$	$\Theta(\log_2(n))$	$O(n)$
Left rotation	`void rotateLeftOn()`	$\Omega(1)$	$\Theta(\log_2(n))$	$O(n)$
Height calculation	`int calculateHeight()`	$\Omega(n)$	$\Theta(n)$	$O(n)$

Figure 1 - A table showing the time complexities of the public functions of the binary search tree data structure. The best, average and worst cases are represented using asymptotic notation, where n represents the number of nodes.

2.2 Constructor

	Best case	Average case	Worst case
`Dictionary()`	$\Omega(1)$	$\Theta(1)$	O(1)

Figure 2 - A table showing the time complexities of the binary search tree constructor, where n represents the number of nodes.

The time complexity of the BST constructor is constant in all cases (best, average and worst) because it always performs only one operation: setting the root pointer to null. This operation does not depend on any factors, making it a fixed-time process.

2.3 Deconstructor

	Best case	Average case	Worst case
`~Dictionary()`	$\Omega(n)$	$\Theta(n)$	O(n)

Figure 3 - A table showing the time complexities of the binary search tree deconstructor, where n represents the number of nodes.

The time complexity of the destructor is O(n) in all cases (best, average and worst). This is because the destructor must traverse and deallocate every node in the tree, which involves visiting each node exactly once. Regardless of the tree's structure, whether it is balanced or unbalanced, the destructor will still perform n operations for n nodes. This results in a linear relationship between the number of nodes and the time required to deallocate memory, making the time complexity consistently O(n).

2.4 Iterative insertion

	Best case	Average case	Worst case
`void insert(int, string)`	$\Omega(\log_2(n))$	$\Theta(\log_2(n))$	O(n)

Figure 4 - A table showing the time complexities of the iteration insertion function, where n represents the number of nodes.

The iterative insertion method involves searching through the tree to find the appropriate position of the new node and then inserting it.

In the best case, the tree would have a perfectly balanced structure. This balance allows for efficient insertion, as each operation requires traversing from the root to the appropriate leaf node, making decisions that halve the remaining possibilities at each level. As a result, a balanced BST has a logarithmic height relative to the number of nodes and the expected number of comparisons from root to leaf is approximately $\log_2(n)$. The insertion itself is a fixed-time operation, meaning the overall best-case time complexity for insertion in a balanced BST is $\Omega(1) + \Omega(\log_2(n)) = \Omega(\log_2(n))$.

According to Sedgewick R. & Wayne, K. (2011), in average cases, when nodes are inserted randomly, the tree tends to maintain a balanced structure. Therefore, we can use the same logic to say the average case complexity is $\Theta(\log_2(n))$.

In contrast, the worst-case scenario occurs when the tree becomes unbalanced due to sequential insertions, leading to a structure resembling a linked list. In this case, each new node must be traversed through all existing nodes to find the appropriate position for insertion. This traversal requires visiting each node in the tree, resulting in a linear time complexity of O(n).

2.5 Recursive insertion

	Best case	Average case	Worst case
`void insert_rec(int, string)`	$\Omega(\log_2(n))$	$\Theta(\log_2(n))$	O(n)

Figure 5 - A table showing the time complexities of the recursive insertion function, where n represents the number of nodes.

The recursive insertion method shares the same time complexity as its iterative counterpart because both methods follow the same traversal logic from root to the appropriate leaf to find the correct position for the new node. While one employs a loop and the other relies on recursive function calls, the core operations, comparing nodes and navigating through the tree, remain unchanged. Therefore, the time complexity scales in the same manner across all cases.

2.6 Iterative look-up

	Best case	Average case	Worst case
`string* lookup(int)`	$\Omega(1)$	$\Theta(\log_2(n))$	O(n)

Figure 6 - A table showing the time complexities of the iterative look-up function, where n represents the number of nodes.

In the best-case scenario, the node being searched for is located at the root of the tree. This results in a time complexity of $\Omega(1)$ because the search operation can be completed in constant time regardless of the total number of nodes, as no traversal is required.

In the average case, assuming the tree is balanced, the method will have to search through the tree, narrowing down the possible location of the node by approximately half with each comparison. The height of a balanced BST is proportional to $\log_2(n)$ and the average number of comparisons needed to locate a node corresponds to the tree's height. Therefore, the average time complexity for is $\Theta(\log_2(n))$.

In the worst-case scenario, the tree can degenerate into a linked list, such as when nodes are inserted in a strictly increasing or decreasing order. This would result in a height of n, leading to a linear time complexity of O(n).

2.7 Recursive look-up

	Best case	Average case	Worst case
`string* lookup_rec(int)`	$\Omega(1)$	$\Theta(\log_2(n))$	O(n)

Figure 7 - A table showing the time complexities of the recursive look-up function, where n represents the number of nodes.

The recursive look-up method shares the same time complexity as the iterative look-up method because both algorithms follow the same traversal path from the root to the target node. In both approaches, whether through looping (iterative) or function calls (recursive), the process involves comparing the target key to the current node and deciding whether to traverse left or right based on the BST structure.

Therefore, the number of comparisons made, and hence the time complexity, is identical between the two methods.

2.8 Removal

	Best case	Average case	Worst case
`void remove(int)`	$\Omega(1)$	$\Theta(\log_2(n))$	O(n)

Figure 8 - A table showing the time complexities of the removal function, where n represents the number of nodes.

The removal operation involves searching through the BST to find the desired node, removing it from the tree and restructuring if required. The time complexity of this function depends on three factors: the structure of the tree (balanced or unbalanced), the location of the node to be removed and if the tree needs to be restructured after removal.

In the best-case scenario, the node to be removed is found at the root and has only one child. The 'search' part of the operation would have a constant time complexity, $\Omega(1)$, as no traversal is required. If the root has only one child, it can simply be replaced by its child, which has a time complexity of $\Omega(1)$. This means the overall best-case time complexity is $\Omega(1) + \Omega(1) = \Omega(1)$.

In the average case, assuming the tree is relatively balanced, the time complexity to search for the node to remove is $\Theta(\log_2(n))$ (see sections 2.6 and 2.7). If the node to be removed has no children, it is simply deleted, requiring $\Theta(1)$. If it has one child, its child takes its place, also requiring $\Theta(1)$. If it has two children, additional work is needed to find the in-order successor (the smallest node in the right subtree) to replace it. This additional step also takes logarithmic time on average in a balanced tree.

This means that regardless of the number of children, the overall average time complexity is $\Theta(\log_2(n))$:

- 0 children: $\Theta(\log_2(n)) + \Theta(1) = \Theta(\log_2(n))$
- 1 child: $\Theta(\log_2(n)) + \Theta(1) = \Theta(\log_2(n))$
- 2 children: $\Theta(\log_2(n)) + \Theta(\log_2(n)) = \Theta(\log_2(n))$

In the worst case, the tree is unbalanced and degenerates into a linked list. In this scenario, the height of the tree is n where every node has only one child. Finding the node to be removed could now require traversing through all n nodes, taking O(n) time. As each node has only one child, the time complexity to remove the node would be O(1), as it just involves replacing the node with its child. This gives an overall worst-case time complexity of $O(n) + O(1) = O(n)$.

2.9 Display tree

	Best case	Average case	Worst case
`void displayTree()`	$\Omega(n)$	$\Theta(n)$	O(n)

Figure 9 - A table showing the time complexities of the tree display function, where n represents the number of nodes.

This function uses an in-order traversal to recursively visit each node in the tree while printing its key-value pair.

The time complexity of this operation is O(n) in all cases. This is because it must traverse and display every node in the tree, which involves visiting each node exactly once. Regardless of the tree's structure, it will still perform n operations for n nodes, resulting in a linear relationship between the number of nodes and the time required.

2.10 Display entries

	Best case	Average case	Worst case
`void displayEntries()`	Ω(n)	Θ(n)	O(n)

Figure 10 - A table showing the time complexities of the entry display function, where n represents the number of nodes.

This operation also performs an in-order traversal of the BST to display all nodes, but instead applies additional formatting to visually represent the tree's structure. For the same reasoning as in 2.9, the time complexity is O(n) in all cases, because each node must be visited exactly once.

2.11 Copy constructor

	Best case	Average case	Worst case
`Dictionary(const Dictionary&)`	Ω(n)	Θ(n)	O(n)

Figure 11 - A table showing the time complexities of the copy constructor, where n represents the number of nodes.

The time complexity of the copy constructor is O(n) in all cases. This is because it calls the deepCopy method, which recursively traverses the entire BST starting from the root. Each node is visited and copied exactly once, leading to a linear relationship between the number of nodes and the time taken, regardless of the structure of the BST.

2.12 Copy assignment operator

	Best case	Average case	Worst case
`Dictionary & operator=(const Dictionary&)`	Ω(1)	Θ(n)	O(n)

Figure 12 - A table showing the time complexities of the copy assignment operator, where n represents the number of nodes.

In the best-case scenario, the copy assignment operator involves self-assignment. In this case, only a pointer check is required, resulting in a constant time complexity of Ω(1).

Otherwise, this operation involves destructing the current tree and performing a deep copy of the source tree. First, the destructor must visit every node in the current tree to deallocate memory, which takes O(n) time. Afterward, the deep copy operation recursively visits each node of the source tree to recreate it, which also requires O(n) time. Since these two processes occur sequentially, the total time complexity becomes O(n) + O(n) = O(2n), which simplifies to O(n), as constant factors are omitted in asymptotic notation because they do not affect the growth rate as n increases.

2.13 Move constructor

	Best case	Average case	Worst case
`Dictionary(Dictionary&&)`	Ω(1)	Θ(1)	O(1)

Figure 13 - A table showing the time complexities of the move constructor, where n represents the number of nodes.

The move constructor transfers ownership of resources from the source object to a new object without requiring deep copying of the tree's nodes. The move constructor doesn't have to traverse the tree or copy nodes, it simply assigns the source object's root pointer to the new object's root and sets the source object's root to nullptr. This pointer reassignment is a constant time operation, O(1), regardless of the number of nodes in the tree.

2.14 Move assignment operator

	Best case	Average case	Worst case
`Dictionary & operator=(Dictionary&&)`	$\Omega(1)$	$\Theta(n)$	O(n)

Figure 14 - A table showing the time complexities of the move assignment operator, where n represents the number of nodes.

In the best-case scenario, the move assignment operator involves self-assignment. In this case, only a pointer check is required, resulting in a constant time complexity of $\Omega(1)$.

In contrast, the average and worst cases consider a situation where the object being assigned already contains a tree that must be deleted to avoid memory leaks. The deletion process requires a complete traversal of the existing tree to deallocate all nodes, resulting in a time complexity of O(n). After the deletion, the new tree is assigned to the root pointer, which is an O(1) operation. O(n) + O(1) simplifies to O(n) because the linear term dominates the constant term in asymptotic notation.

2.15 Rotation

	Best case	Average case	Worst case
`void rotateRightOn()`	$\Omega(1)$	$\Theta(\log_2(n))$	O(n)
`void rotateLeftOn()`	$\Omega(1)$	$\Theta(\log_2(n))$	O(n)

Figure 15 - A table showing the time complexities of the rotation operation, where n represents the number of nodes.

The rotation operations involve locating the node to rotate and performing the rotation itself.

The rotation itself is a constant-time operation because it involves a fixed number of pointer adjustments, regardless of the size of the tree. A rotation in a BST, whether left or right, only affects a local portion of the tree: the node being rotated, its child and potentially one of its grandchildren. For example, for a right rotation, it adjusts the pointers between the root, its left child and the left child's right subtree. These are simple, constant-time pointer manipulations, making the rotation itself O(1).

In the best-case scenario, the node to be rotated is found immediately, such as when the node is the root. In this case, the search is a fixed-time operation and the rotation is performed in constant time. Therefore, the overall best-case time complexity is $\Omega(1) + \Omega(1) = \Omega(1)$.

In the average-case scenario, assuming the tree is balanced, the height of the tree is logarithmic relative to the number of nodes, i.e., $\log_2(n)$. To locate the node that needs to be rotated, the function traverses down the tree, which in a balanced tree takes $\Theta(\log_2(n))$. Since the rotation itself requires $\Theta(1)$, the overall average-case time complexity is $\Theta(\log_2(n)) + \Theta(1) = \Theta(\log_2(n))$.

In the worst-case scenario, the tree is unbalanced, resembling a linear structure (degenerate tree). In this case, locating the node could take O(n) time. The rotation remains O(1), so the overall worst-case time complexity is O(n) + O(1) = O(n).

2.16 Height calculation

	Best case	Average case	Worst case
`int calculateHeight()`	$\Omega(n)$	$\Theta(n)$	O(n)

Figure 15 - A table showing the time complexities of the height calculation, where n represents the number of nodes.

This method calculates the height of the BST by recursively traversing each node to find the maximum depth of its subtrees. Its time complexity is O(n) in all cases because the algorithm must visit each node exactly once during the recursive traversal, regardless of the tree's shape or balance.

3 Conclusion

This report has provided an in-depth analysis of the time complexities associated with BST operations. The findings highlight the importance of tree structure on the performance of operations, reinforcing the role of balanced trees in ensuring optimal efficiency. This analysis also signifies the value of careful algorithm selection in software engineering to achieve high performance.

4 References

Sedgewick R. & Wayne, K., 2011. 3.2 Binary Search Trees. In: *Algorithms.* 4th edition. Boston: Pearson Education, Inc., 2011, pp, 396-458.