# **Advanced Software Engineering**

# **Coursework 1**

# **Code Analysis**

# **Jared Awiti (N0992216)**

# **Course Leader: Niel Sculthrope**

**Contents**

Part A……………………………….3

Part B3…………………………………8

Part B4…………………………………..11

Part B5………………………………….14

Part C1…………………………………15

**PART A**

**Insert function.**

Best case

O (1)

The best-case time complexity for the insert function occurs when the value inserted has the same key as the node. Regardless of the shape of the tree, if the item has the same key as the item currently at the root node, the value of the node will be replaced. This results in an O(1) time complexity.

Average case

O log(n)

The average time complexity for the insert function is O log(n). This is because if the tree is somewhat balanced, the function will not have to visit every single node to find where to insert the new node. This results in the O log(n) time complexity.

Worse case

O (n)

The worst-case time complexity for the insert function occurs when the tree is completely unbalanced. This means it is like a linked list and does not reap the benefits of a binary tree. If the value that needs to be looked up is at the bottom of the tree, the function must go through every single node to find the value. This means the worst-case time complexity is O(n), where n is the number of nodes.

**Lookup function.**

Best case

O (1)

The best-case time complexity for the lookup function occurs when the value looked up is at the root node. Regardless of the tree shape if the value looked up is at the root the algorithm will not have to move and will return the value. This results in an O(1) time complexity.

Average case

O (log n)

The average time complexity for the lookup function is O log(n). This is because if the tree is somewhat balanced, the function will not have to visit every single node to find where to look up the node. This results in the O log(n) time complexity.

Worse case

O (n)

The worst-case time complexity for the lookup function occurs when the tree is completely unbalanced. This means it is like a linked list and does not reap the benefits of a binary tree. If the value that needs to be looked up is at the bottom of the tree, the function must go through every single node to find the value. This means the worst-case time complexity is O(n), where n is the number of nodes.

**displayEntries function.**

Best case

O (n)

The best case occurs when the binary tree is fully balanced. The best-case time complexity for the displayEntries function is O(n). This is because the function needs to step through every node once and display it in either post-order, pre-order, or in-order traversal. This means the best case is O(n), where n is the number of nodes.

Average case

O (n)

The average case time complexity for the displayEntries function is O(n). This can occur when the tree is somewhat balanced. This is because the function needs to step through every node once and display it in either post-order, pre-order, or in-order traversal. This means the average case is O(n), where n is the number of nodes.

Worst case

O (n)

The worst case can occur when the tree is completely unbalanced. The tree will have to step through every node and display it. This means that the worst-case time complexity is O(n).

**Destructor.**

Best case

O(n)

The best-case time complexity for the destructor is O(n) and may occur when the tree is completely balanced. This is because the function needs to step through every node once and delete it using post-order traversal. This means the best case is O(n), where n is the number of nodes.

Average case

O(n)

The average case time complexity for the destructor is O(n) and can occur when the tree is somewhat unbalanced. This is because the function needs to step through every node once and delete it using post-order traversal. This means the average case is O(n), where n is the number of nodes.

Worst case

O(n)

The worst-case time complexity for the destructor is O(n) and may occur when the tree is completely unbalanced. This is because the function needs to step through every node once and delete it using post-order traversal. This means the worst-case is O(n), where n is the number of nodes.

**Remove function.**

Best case

O (1)

The best-case time complexity for the remove function occurs when the tree is completely unbalanced. This is because if the tree is completely unbalanced, for example, the root in key 5 and the tree is skewed to the right and a value is removed which is less than the root key. The function will simply look at the root left pointer and see null. This means it will return “Key not in the tree”. This will be a constant time operation, O(1).

Average case

O (log n)

The average case time complexity for the remove function occurs when the tree is either completely or somewhat balanced. The function will traverse the height of the tree to find the node. If the node has no children nodes, then it will be removed. If the node has children nodes then it will be removed and pointers will be rearranged, rearrangement of pointers is O(1) time complexity. This means either way the time complexity of remove is O(log(n)).

Worse case

O(n)

The worst-case time complexity for the remove function occurs when the tree is completely unbalanced. This means it is like a linked list and does not reap the benefits of a binary tree. If the value that needs to be removed is at the bottom of the tree, the function must go through every single node to remove the value. This means the worst-case time complexity is O(n), where n is the number of nodes.

**displayTree function.**

Best case

O(n)

The best-case time complexity, when the tree is completely balanced, for the displayTree function is O(n). This is because the function needs to step through every node once and display it. This means the best case is O(n), where n is the number of nodes.

Average case

O(n)

The average case time complexity for the displayTree function is O(n). This is because the function needs to step through every node once and display it. This means the average case is O(n), where n is the number of nodes.

Worst-case

O(n)

The worst-case time complexity, when the tree is completely unbalanced, for the displayTree function is O(n). This is because the function needs to step through every node once and display it. This means the worst-case is O(n), where n is the number of nodes.

**RotateLeft and rotateRight functions.**

Best case

O(1)

The best-case time complexity of the rotateRight or rotateLeft function occurs when the tree is balanced, and rotation is on the root. This is because the function will not have to traverse the tree and instead will need to adjust the root pointer and one other pointer to achieve the rotation, which are O(1) operations. This results in the function being a constant time complexity, O(1).

Average case

O(log n)

The average-case time complexity of the rotateRight or rotateLeft function occurs when the function has to traverse the tree to find the pivot node, which is O(log(n)) time complexity. Regardless of if the tree is completely balanced or not completely balanced, this is an O(log(n)) time complexity.

Worse case

O(n)

The worst-case time complexity of the rotateRight or rotateLeft function occurs when the tree is completely unbalanced, and the pivot node is the node before the last node. This means the function will have to travel to every single node until it finds the pivot node, this is an O(n) time complexity. Only then it can rearrange the points which is an O(1) time complexity. Overall, this function is an O(n) time complexity worst-case.

**Copy constructor.**

Best case

O(n)

The best-case time complexity for the Copy constructor is O(n), for example when the tree is completely balanced. This is because the constructor is completing a deep copy. This requires every node to be visited once and copied to the new object. This means the best case is O(n), where n is the number of nodes.

Average case

O(n)

The average case time complexity for the Copy constructor is O(n). This is because the constructor is completing a deep copy. This requires every node to be visited once and copied to the new object. This means the average case is O(n), where n is the number of nodes.

Worst case

O(n)

The worst-case time complexity for the Copy constructor is O(n), for example when the tree is completely unbalanced. This is because the constructor is completing a deep copy. This requires every node to be visited once and copied to the new object. This means the worst case is O(n), where n is the number of nodes.

**Move constructor.**

Best case

O(1)

The best-case time complexity, for example, when the tree is completely balanced, for the Move constructor is O(1). This is because the constructor is completing a shallow copy. This requires the new object to point to the root node and the original object to point away from the root node. This means the best case is O(1), constant time.

Average case

O(1)

The average case time complexity for the Move constructor is O(1). This is because the constructor is completing a shallow copy. This requires the new object to point to the root node and the original object to point away from the root node. This means the average case is O(1), constant time.

Worst case

O(1)

The worst-case time complexity, for example, when the tree is completely balanced, for the Move constructor is O(1). This is because the constructor is completing a shallow copy. This requires the new object to point to the root node and the original object to point away from the root node. This means the worst case is O(1), constant time.

**Copy assignment Operator.**

Best-case

O(n)

The best case occurs when the tree is completely balanced. The best-case time complexity is still determined by the deep copy operation, which traverses every node, and the best scenario is when the tree is balanced. Therefore, the best-case time complexity is O(n), where n is the number of nodes in the tree.

Average-case

O(n)

The average case depends on the depth of the tree. The average-case time complexity is still determined by the deep copy operation. Therefore, the average-case time complexity is O(n), where n is the number of nodes in the tree.

Worst-case

O(n)

The worst case occurs when the tree is completely unbalanced. The worst-case time complexity is still determined by the deep copy operation, and the worst scenario is when the tree is unbalanced. Therefore, the worst-case time complexity is O(n), where n is the number of nodes in the tree.

**Move assignment Operator.**

Best-case

O(n)

The best case occurs when the tree is completely balanced. The best-case time complexity is still determined by the deep delete operation, which traverses every node, and the best scenario is when the tree is balanced. Therefore, the best-case time complexity is O(n), where n is the number of nodes in the tree.

Average-case

O(n)

The average case depends on the depth of the tree. The average-case time complexity is still determined by the deep delete operation which traverses every node. Therefore, the average-case time complexity is O(n), where n is the number of nodes in the tree.

Worst-case

O(n)

The worst case occurs when the tree is completely unbalanced. The worst-case time complexity is still determined by the deep delete operation, which traverses every node, and the worst scenario is when the tree is unbalanced. Therefore, the worst-case time complexity is O(n), where n is the number of nodes in the tree.

**Part B3**

**Efficient Implementation**

The overall time complexity of the efficient Implementation of the Domino game is O(n). the overall reason is due to reading the file. Each domino must be read one by one and then added to the unordered dominoes vector. This implementation uses unordered\_maps to search dominoes within this vector which makes searching more efficient. However, this results in the overall domino game being O(n), n being the number of dominoes.

For the solutions vectors, pairs and an unordered map were used. Vectors were used because it was unknown how many dominos would be loaded into the program. Std::vector is a dynamic array in C++, which means the vector can grow or shrink depending on the number of dominoes inserted. The standard container pairs were used because the domino is split on the left and right sides. Pairs allowed the sides to be distinguished easily and easily access one part, using either .first or .second, which is a constant time operation. An unordered map was used for constant time searching. An unordered map is a hash table and uses key-value pairs to achieve constant time lookups. This allows for the program to run at an O(1) time complexity.

**Constructor**

The time complexity of the constructor is O(n). This is due to the function having to read all the dominoes from the file sequentially. This means according to the number of dominoes the time will increase. The populating of the unordered maps is also an O(n) time complexity as each domino will need to be assigned a key-value pair and stored within the unordered map. This results in the overall time complexity of the function being O(n).

**Read dominoes function.**

The time complexity of the read dominoes function is O(n). This is due to the function having to run through each line of the file, which is a domino, described as n. This means the bigger the file the more time taken for the read domino function. Resulting in an O(n) time complexity.

**Completed function.**

The time complexity of the completed function is O(1). The .empty() function is from the std::vector standard library and checks if the size of the vector is 0. This does not depend on the size of the vector hence the time complexity of the function is O(1).

**Output line function**

The time complexity of the function output line is O(n), where n is the number of dominos in the vector domino\_line. The output of the function e.g. cout is an O(1) operation. Therefore, the overall time complexity of the function is O(n).

**Get right domino function.**

The time complexity of this function is O(1). This is due to the function finding the domino using a hash map. By using the key to search for a domino the domino can be easily found, this is a constant time operation, hence O(1).

**Get left domino function.**

The time complexity of this function is O(1). This is due to the function finding the domino using a hash map. By using the key to search for a domino the domino can be easily found, this is a constant time operation, hence O(1).

**Next domino right function**

The time complexity of this function is O(n). The function uses an unordered\_map from the get\_right\_domino function to find the domino which is a constant time operation. The push\_back function is similarly constant time. However, the erase and std::Remove functions are an O(n) time complexity. These functions iterate over the unordered\_dominoes to remove it from the vector. As a result, the function is an O(n) time complexity. Where n is the number of dominoes in the vector.

**Next domino left function.**

The time complexity of this function is O(n). the function uses an unordered\_map from the get\_left\_domino function to find the domino which is a constant time operation. The push\_back function is similarly constant time. However, the erase and std::Remove functions are an O(n) time complexity. These functions iterate over the unordered\_dominoes to remove it from the vector. As a result, the function is an O(n) time complexity. Where n is the number of dominoes in the vector.

**Worse Case Implementation**

The overall time complexity of the worst-case Implementation of the Domino game is O(n). the overall reason is due to reading the file. Each domino must be read one by one and then added to the unordered dominoes vector. This implementation uses maps to search dominoes within this vector which makes searching more efficient. There are not many changes that can enable the efficient implementation to be changed to the most efficient worst case. The one major way to change this is changing the way the user searches using maps instead. However, this results in the overall domino game being O(n), n being the amount of dominoes.

For the solutions vectors, pairs and a map were used. Vectors were used because it was unknown how many dominos would be loaded into the program. Std::vector is a dynamic array in C++, which means the vector can grow or shrink depending on the number of dominoes inserted. The standard container pairs were used because the domino is split on the left and right sides. Pairs allowed the sides to be distinguished easily and easily access one part, using either .first or .second, which is a constant time operation. A map was used for O(log(n)) searching. A map is a red-black binary tree which traverses down the tree to find its values. A red-black tree is a self-balancing binary search tree. This allows for the program to run at an average case of O(log(n)) time complexity.

**Get right domino function.**

The time complexity of this function is O(log(n)). This is due to the function finding the domino using a red-black binary tree. By using the key to search for a domino the domino can be easily found by traversing the tree, this is a constant time operation, hence O(log(n)). Furthermore, the red-black tree is self-balancing so it will not be unbalanced at any time.

**Get left domino function.**

The time complexity of this function is O(log(n)). This is due to the function finding the domino using a red-black binary tree. By using the key to search for a domino the domino can be easily found by traversing the tree, this is a constant time operation, hence O(log(n)). Furthermore, the red-black tree is self-balancing so it will not be unbalanced at any time.

**Part B4**

**Efficient 1000 inputs**

|  |  |  |
| --- | --- | --- |
| **Dominos placed** | **Mean time per operation (microseconds, 3 results)** | **Mean (0dp)** |
| 200 | 1010,867,894 | 924 |
| 400 | 843,843,783 | 823 |
| 600 | 624,674,576 | 624 |
| 800 | 534,548,529 | 537 |
| 1000 | 506,506,430 | 480 |

**Worse case 1000 inputs**

|  |  |  |
| --- | --- | --- |
| **Dominos placed** | **Mean time per operation (microseconds, 3 results)** | **Mean (0dp)** |
| 200 | 949,907,901 | 919 |
| 400 | 785,750,881 | 805 |
| 600 | 758,673,590 | 674 |
| 800 | 621,612,602 | 612 |
| 1000 | 572,507,489 | 523 |

A graph with a line and a red line

Description automatically generated

In part B3 it was stated that both Worst-case and efficient implementations have a n O(n) time complexity. This means the more dominoes placed the longer it will take. In the 1000 dominoes inputs above we can see the more dominoes placed the shorter the meantime per operation. This does not prove an O(n) time complexity. This is most likely due to the small number of dominoes in this test, and the time may be influenced by other factors such as overheads. However, we can see as the number of dominoes placed increases the efficient implementation is quicker per operation than the worse-case implementation.

**Efficient 6k inputs**

|  |  |  |
| --- | --- | --- |
| **Dominos placed** | **Mean time per operation (microseconds, 3 results)** | **Mean (0dp)** |
| 1000 | 5543,5438,4983 | 5321 |
| 2000 | 5063,4329,4279 | 4457 |
| 3000 | 4105,3961,4034 | 4033 |
| 4000 | 3737,3470,3522 | 3576 |
| 5000 | 3205,3040,3351 | 3199 |
| 6000 | 2627,2359,2259 | 2415 |

**Worse case 6k inputs**

|  |  |  |
| --- | --- | --- |
| **Dominos placed** | **Mean time per operation (microseconds, 3 results)** | **Mean (0dp)** |
| 1000 | 7440,7647,7863 | 7650 |
| 2000 | 6706,6155,6308 | 6390 |
| 3000 | 5832,5755,6176 | 5695 |
| 4000 | 5221,5078,5216 | 5172 |
| 5000 | 4233,4376,4503 | 4371 |
| 6000 | 2684,3624,3837 | 3382 |

A graph with a red and blue line

Description automatically generated

Like the 1000 domino input case, the 6k domino inputs case does not follow an O(n) time complexity. However, it can be inferred from the graph above that the higher the amount of dominoes placed the better the efficient implementation is than the worse-case implementation. From the graph above we can see the efficient implementation is much faster than the worst-case implementation.

**Efficient 10k inputs**

|  |  |  |
| --- | --- | --- |
| **Dominos placed** | **Mean time per operation (microseconds, 3 results)** | **Mean (0dp)** |
| 2000 | 4407,4178,4448 | 4344 |
| 4000 | 5560,5562,5626 | 5583 |
| 6000 | 6672,6584,6362 | 6539 |
| 8000 | 7162,7062,7664 | 7296 |
| 10000 | 7591,7572,7760 | 7641 |

**Worse case 10k inputs**

|  |  |  |
| --- | --- | --- |
| **Dominos placed** | **Mean time per operation (microseconds, 3 results)** | **Mean (0dp)** |
| 2000 | 6070,5850,5865 | 5928 |
| 4000 | 6204,6292,6373 | 6290 |
| 6000 | 7018,6877,7189 | 7028 |
| 8000 | 8627,8552,8090 | 8423 |
| 10000 | 9666,9210,9253 | 9376 |

**A graph with a line and a red line

Description automatically generated**

The 10k domino input graph above truly shows an O(n) time complexity. This proves as the number of dominoes increases so does the time increase linearly. Furthermore, from the graph, it is shown that the efficient implementation is still quicker in operation time than the worst-case implementation. This is shown in the biggest test where the meantime for 10000 domino inputs for the efficient case was 7641 and the worst case was 9376.

Due to my personal laptop specs, CPU, RAM etc. The larger tests e.g. 30k, 60k could not be run. However, based on the graph above, it can be inferred that they would display similar qualities to the 10k implementation.

**Part B5**

The overall results reveal the time complexity of both worse case and efficient implementation is O(n) when the program is large enough. When the number of dominoes used is less than 10000 it may seem like both programs do not exhibit O(n) behaviours however this can be due to various other factors. However, when the program is over this threshold the program runs at an O(n) time complexity for both implementations. Unfortunately, due to my laptop specs, I could not run larger files however I believe the results would complement my analysis. The results above show that the efficient implementation is indeed faster than the worst-case implementation. This is due to the use of hash tables which means faster lookup times than a tree which is used in the worse-case scenario. Overall, the results support my case that the time complexity of both implementations is O(n), as the graphs show a gradual increase in time when more dominoes are placed.

**Part C1**

The convoluted algorithm starts storing the domino data as a list of pairs. This will be an O(n) time complexity as all items will need to be sequentially copied to the new list, List B. As the number of dominoes gets bigger the time taken will also increase.

In the second step, a single element from starting domino is added to list P, this can be done in constant time so O(1). Regardless of the number of dominoes only the first one will be taken; hence the number of dominoes does not affect the time.

For step three the lists are constructed, searched through, and sorted multiple times. This is until the distance variable is greater than the number of dominoes. Each sorting algorithm takes an O(n log(n)) time complexity. There is a possibility that each domino could be compared to multiple other dominoes. This can significantly increase the time complexity. As it means as the number of dominoes increases the time increases. Each actual operation in the loop is O(1) time complexity however due to the looping and searching this step is an O(n log(n)) operation.

For step four, List P is then sorted this is an O(n log(n)) time complexity as this is the time complexity of the inbuilt C++ sort. When n is the number of elements in list P.

Finally, the line is built using the pointer to construct each domino. This will mean traversing the whole list P, this is an O(n) time complexity. Hence the overall average time complexity is O(n log n).