# Task 1 – Complexity Analysis of BST Implementation

For my implementation of a Binary Search Tree, I have completed all of the core functionality required in Labs 2-4, as well as all of the advanced functionality in Lab 2, as well as the advanced tree traversals in Lab 3. Hence, I will be analysing the time complexities of the lookup, insert, displayEntries, displayTree, and deepDelete functions that I have written.

The “core” version of my Binary Search Tree defines the Tree as a class, and the Nodes as a separate Struct. The Node has 4 data members, the “key” integer and “item” string, which are defined in the constructor, as well as the left and right children of the node, which are pointers and set to null by default. The basic version of the BST class provides 2 members, lookup, and insert. Lookup searches through the tree to find a given key, and insert recursively inserts a new node into the tree. The lab 3 activities introduced a displayEntries function, which recursively traverses the tree and outputs each key and item to the console. Finally, Lab 4 introduces a deepDelete function, and overwrites the class destructor to call on it, which recursively traverses the tree and deletes each node.

The lookup function works iteratively, utilising a while loop to search through the tree until the correct node is found. Until the correct node is reached, it updates the current node to become either it’s left or right child, until either the correct key or a null value is found. The function returns a reference of the item. As a binary search tree effectively halves the amount of data to look through on each iteration, assuming it is fully balanced. In this assumption, the worst case time complexity of the lookup function is logarithmic, O(log2(n)). The best case for such a tree would be that the sought key is the root node, and therefore would have a constant time complexity, O(1), as this is always the first node that is looked at. The implementation of the search tree is not self-balancing, however, and it is likely the tree may be skewed. In the case of an unbalanced tree, the worst case time complexity is linear, O(n), as the amount of time it will take to search for an item at the bottom of an unbalanced tree will be proportional to the height of the tree, and we cannot assume that half of the dataset is removed in each iteration as the tree is not balanced. The best case time complexity will remain to be constant, O(1).

The insert function I have written works iteratively, with a public function that will call a recursive private function, passing the root node and the node to be inserted. The private function will check if a root exists, and insert the node into the root if not. Otherwise it will check whether the key of the new node is greater than or less than the current node. If the corresponding child of the current node is null, the new node will be set to the child, otherwise the function will recrsively call itself, passing in the child of the current node. This will happen until a null value is found, or if a node with the key to be added already exists and is found. In this case, the children of that node will be detached, and added as the children to the new node. The new node will then be set to that node. If the tree is perfectly balanced, the worst case time complexity for this function will be logarithmic, O(log2(n)), as with each recursion, half of the possible dataset will be discarded when searching for the location to insert into. The best case time complexity will be constant, O(1), as if the root node needs to be inserted into, the algorithm will not need to recursively search through the tree. In the case of an unbalanced tree, the worst case time complexity for an insertion will be linear, O(n), as if the insertion occurs at the bottom of the tree, the time taken will be proportional to the height of the tree. The best case will remain to be constant, O(1), as the root will remain to be the first node that is looked at.

The displayEntries function consists of a public member that will call the private recursive displayEntriesRec function, passing in the root as the first node. The purpose of this function is to display all of the values within the tree in key order. This function recursively performs an in-order traversal of the tree, by first traversing all left sub-trees, outputting the key and value, and then traversing all of the right sub-trees. As this tree traversal must go through all of the nodes in the tree, the time complexity will be proportional to the amount of nodes in the tree, and therefor the best and worst case time complexities will be linear, O(1).

Finally, the deepDelete function works recursively to delete all nodes in the tree. The default destructor of the class is overwritten to instead call the deepDelete function, passing in the root node. This will then perform a post-order traversal of the tree, and if the node is a leaf, it will be deleted. Otherwise it will recursively call itself until it finds a leaf, first traversing all left sub-trees, then all right sub-trees, until it eventually deletes the root node. As the function will need to traverse all nodes in the tree, the best and worst case time complexities for this function will be linear, O(1), as the amount of time taken will be proportional to the amount of nodes within the tree.

The ”advanced” version of the binary search tree declares the Node struct as a private member of the BST class, instead of seperately to the class. It also introduces an extra member function to check whether the node it is passed is a leaf or not, returning a Boolean value. In addition to this, the Lab 2 advanced tasks introduce a recursive implementation of the lookup function, and the Lab 3 advanced tasks introduce a recursive function that displays the structure of the tree, called displayTree.

The best and worst case time complexity for the isLeaf function would be constant O(1), as the amount of children a node has does not change, it simply checks whether those children are null or not.

The recursive lookup has a public function that calls the recursive private function, passing the sought key and the root node. The private function then checks if the current node’s key is the sought key, and returns it if true. Otherwise it checks if the current node is a leaf, and returns null if this is the case. If the node has children, it compares the sought key to the key of the current node, to identify which direction to traverse in, and recursively calls itself, passing in the sought key and the corresponding child of the current node. As with the iterative lookup function, the worst case time complexity would be logarithmicO(log2 (n)) if the tree is fully balanced, as half of the dataset is removed with each recursion. If the tree is unbalanced, the worst case time complexity would be linear, O(n), as if the sought key is at the bottom of the tree, the amount of time taken to find it would be proportional to the height of the tree. The best case time complexity for both scenario’s would remain to be constant, O(1), as if the sought key is the root, it will be found after the first lookup and no further traversal is needed.

Finally, the displayTree function is another recursive function that performs a pre-order traversal of the binary search tree, in order to output the structure of the entire tree, outlined by indenting the outputs to show levels. As with other functions that perform a full traversal of the binary search tree, the best and worst case time complexities are linear, O(n), as the amount of time taken to traverse each node is proportional to the amount of nodes in the tree.

# Task 2 – Analysis of the Solution

## 3a - Understanding Performance Guarantees – Standard Library Components

Some containers in the C++ standard library that could be used to implement the Royal Software Engineers Algorithm are the list, map, and unordered map.

A list container in C++ is implemented as a doubly linked list, and allows for insertion and deletion operations anywhere in the container, as well as iteration in both directions of the container. A linked list is a linear data structure where the elements are not stored in one, contiguous location in memory. Instead they contain pointers to the next element in the list, allowing for elements to be stored in smaller, separate memory locations, leading to increased space efficiency. The elements of a doubly linked list also contain pointers to the previous element in the sequence, meaning it can be traversed in both directions. For insertion into the list, the C++ standard library provides the push\_front() and push\_back() functions, which allow an element to be added directly to either the front or the back of the list. This would be useful for the algorithm as if the container is used to build and store the result sequence, the item could be added directly to the front or back of the container, depending on if it is a southern or northern name. The best, worst, and average case time complexity for this operation is constant, O(1), as the functions directly access the location where the insertion is taking place, rather than traversing through the list, therefore the size of the list does not affect performance. There is no difference between the worst and average case as the data within the list does not affect the insertion algorithm. For searching operations, the standard library list does not directly provide a searching function. However, a find() function is available in the standard library, compatible with lists. The list also offers functions to return iterator objects. The find() operation will have a best case time complexity of constant, or O(1), if the element is the first in the list. The worst case time complexity will be O(n), as all elements will need to be searched through if the sought element is the final one or not present. The average case time complexity will also be O(n), as if the sought item is in the middle of the list, the time taken to find it will still be proportional to the size of the list.

A map container in C++ is implemented as a binary search tree, storing elements that are formed as a combination of a key and a corresponding value. A binary search tree is a data structure that stores elements as nodes, containing their key and value, and pointers to their children nodes. If the key of the child is greater than the key of the node, it is stored as a right child. Otherwise it is stored as a left child. The standard library provides an insert() operator for inserting a new key/value pair into the container. The best case time complexity for this operation would be constant, as if there are no elements in the container, the first element would be the root node of the tree, and only the root node would need to be accessed. The worst case time complexity would be O(log2(n)). This is because the implementation of the map is self-balancing, so if the location to insert into is at the bottom of the tree, half of the dataset does not need to be looked through with each iteration. The average case time complexity is also O(log2(n)), as if the location to insert into is halfway down the tree, the height of the tree will still have an impact, but half of the tree will still be discarded when finding the location. For searching operations, the find() function is available. This returns a reference to the value at a given key, or an exception if the key is not present. As insertion into a binary search tree requires searching through the tree, the time complexities will mostly remain the same, for the same reasons. Best case is O(1), while worst and average cases are O(log2(n)).

An unordered map container in C++ also uses a key/value pairing, like with the map. However instead of implementing a binary search tree, the unordered map utilises hash tables and buckets, enabling fast access to an element based on its key. A hash function is used to convert the key into a hash value, which is converted to an array index, which allows for accessing its’ corresponding cell in the array. This also means that, unlike a map, the unordered map is not sorted in any way. For insertion, the unordered map offers the insert() member function. The best case time complexity for this is O(1), if there are no collisions for the hash value of the key. The worst case time complexity for insertion is O(n), as if if all an insertion triggers a hash collision, checking the linked list within the bucket will be proportional to the amount of elements in the list. The average time complexity for an unordered map is O(1) as hashing collisions can be rare. For searching operations, the find() member function is provided. This accesses the hash value of the key in the bucket and returns the stored value. The best case time complexity for this is O(1), as if no collisions occur, the element can be directly accessed. The worst case time complexity is O(n), as if collisions have occurred, the values in that index in the bucket will be stored as a linked list, which will need to be traversed. The amount of time to travers the linked list is proportional to the items in the list. Finally, the average time complexity will be O(1), as hash collisions are very rare.

## 3b - Analysis of the Royal Software Engineers Algorithm

The algorithm will need to load the contents of a text file into data container A. Then perform a single insertion into data container B, which will build the sequence. Then, searching through all items in A, it will perform a single insertion into B, and repeat this until no matches are found for the western sequence. It will then repeat this for the eastern sequence.

A combination of a list and map could be used in the algorithm. A map could be implemented to load all of the bricks into the memory, storing the northern name as a key and the southern name as a value. These would be stored as a binary search tree, and would have a worst and average time complexity for both insertion and searching of O(log2(n)). A list can be used to build and store the result sequence, as one item can be added to the list from the map, and push\_back can be used to build the eastern part of the sequence, while push\_front can be used to build the western part of the sequence. The worst and average time complexity for insertion is O(1). For searching, the worst and average case time complexities will be O(n). Assuming the average case for each operation, as the algorithm will be performing many binary tree searches within a loop, the average time complexity of the entire algorithm will become O(n Log2(n)).

A combination of the list and unordered map could be implemented in a similar way, however an unordered map could be used for loading the northern and southern names into memory instead of the map, as it is not important that these values are ordered. This uses a hash table instead of the binary search tree, and the worst case time complexity for both insertion and searching will be O(n), while the average case time complexity will be O(1). The list would have the same functionality as listed above, and have the worst and average time complexity for insertion and searching of O(n). Assuming the average case, the time complexity of the algorithm would be O(n), as the data would need to be iterated over a number of times.

Finally, a combination of two lists could be used for the implementation. The data could be loaded into a list, line by line, with each entry containing a secondary list containing the northern name in the first index and the southern name in the second. The implementation of a list for storing the result sequence would remain the same as previously mentioned. For the first list, as there will be multiple lists nested within the primary list, the average and worst time complexities becomes O(n2). The average time complexity for the algorithm would be O(n2), due to the need to search through nested lists, causing nested loops.

Overall, I feel that the list and unordered map combination is the best. The use of a map would not provide advantages over an unordered map, as the data does not need to be ordered when being loaded into the memory, and this implementation appears to have the most efficient average time complexity. I would consider the list and map combination to be second best.

# Task 4b – Evaluation of Results

I decided to implement both of the list and unordered map, and the list and map combinations of the algorithm, in order to have a comparison to identify whether my choices of containers were correct.

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm 1 - List and Unordered Map | | | |
| **Size** | **Nanoseconds** | **Milliseconds** | **Seconds** |
| **20** | 65800 | 0.0658 | 0.0000658 |
| **50** | 103100 | 0.1031 | 0.0001031 |
| **100** | 150500 | 0.1505 | 0.0001505 |
| **200** | 265400 | 0.2654 | 0.0002654 |
| **500** | 627800 | 0.6278 | 0.0006278 |
| **1000** | 1095600 | 1.0956 | 0.0010956 |
| **2000** | 2334700 | 2.3347 | 0.0023347 |
| **5000** | 5674100 | 5.6741 | 0.0056741 |
| **10000** | 11068500 | 11.0685 | 0.0110685 |
| **20000** | 22461400 | 22.4614 | 0.0224614 |
| **50000** | 54965700 | 54.9657 | 0.0549657 |
| **100000** | 121395900 | 121.3959 | 0.1213959 |
| **200000** | 275594900 | 275.5949 | 0.2755949 |
| **500000** | 864470800 | 864.4708 | 0.8644708 |
| **1000000** | 1834896800 | 1834.8968 | 1.8348968 |
| **2000000** | 3821338800 | 3821.3388 | 3.8213388 |
| **3000000** | 5553298700 | 5553.2987 | 5.5532987 |

When running tests to identify the amount of time taken to run the List and Unordered Map implementation, it can be seen that generally, when the data doubles in size, the amount of time taken to run the algorithm also approximately doubles, with some variation. The variation could be due to the fact that the first item loaded into the result sequence is random, so it could be the first or last items in the sequence. On average, however, it appears that the time complexity of the algorithm is linear, O(n), as the amount of time seems to increase at approximately the same rate as the dataset.

When looking at a graph of the results we can see that the time complexity appears to be O(n). This had been corrrectly identified in the analysis of the algorithm, as the size of the dataset increases the amount of time to complete the algorithm increases at a very similar rate.

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm 2 - List and Map | | | |
| **Size** | **Nanoseconds** | **Milliseconds** | **Seconds** |
| **20** | 59200 | 0.0592 | 0.0000592 |
| **50** | 149400 | 0.1494 | 0.0001494 |
| **100** | 283100 | 0.2831 | 0.0002831 |
| **200** | 396800 | 0.3968 | 0.0003968 |
| **500** | 933100 | 0.9331 | 0.0009331 |
| **1000** | 1990000 | 1.99 | 0.00199 |
| **2000** | 4031700 | 4.0317 | 0.0040317 |
| **5000** | 11126100 | 11.1261 | 0.0111261 |
| **10000** | 22536000 | 22.536 | 0.022536 |
| **20000** | 47921700 | 47.9217 | 0.0479217 |
| **50000** | 128959900 | 128.9599 | 0.1289599 |
| **100000** | 284840400 | 284.8404 | 0.2848404 |
| **200000** | 659441400 | 659.4414 | 0.6594414 |
| **500000** | 1945005800 | 1945.0058 | 1.9450058 |
| **1000000** | 4133296500 | 4133.2965 | 4.1332965 |
| **2000000** | 9259596900 | 9259.5969 | 9.2595969 |
| **3000000** | 14665420300 | 14665.4203 | 14.6654203 |

When running tests on the List and Map implementation of the algorithm, a similar pattern can be identified. The amount of time taken appears to increase at a linear rate with the size of the dataset, however as the dataset gets larger, this begins to jump to a greater rate.

Looking at the graph of the timings, it appears to be very similar to the list and unordered map, however there is a slight curve at the first few plotted points, until 20K items in the dataset. At this point it begins to increase linearly. This suggests the graph is loglinear, O(n Log2(n)). This had been correctly identified in the analysis of the algorithm.

Looking at both of these algorithms together, we can see the List and Unordered map is much more efficient in regards to time complexity as opposed to the List and Map implementation. At 3 million data entries, the List and Unordered Map implementation takes 5.5 seconds, while the Map takes 14.6 seconds. We can also see that the Map grows at a much faster rate than the Unordered Map. This is due to the Map implementation being O(n log2(n)), on average, while the Unordered map being O(n) on average. Until 20K entries in the dataset, the Map implementation has a curve, while the Unordered Map implementation grows linearly in relation to the size. At this point both implementations grow linearly, however the Map grows at a much higher rate.

In conclusion, the List and Unordered Map has a time complexity of O(n), while the List and Map has a time complexity of O(n Log2(n)). The Unordered Map has a more efficient time complexity and is able to outperform the Map implementation.