Ryan Gorman (N0809433)

Advanced software engineering

Assessment One

Abstract

An investigation into the time complexity of various binary search tree implementations, including a time complexity analysis of the great wall problem and the varying approaches that can be implemented to solve it.

Contents

[1. Complexity Analysis of a BST Implementation 4](#_Toc97326400)

[1.1 Time Complexity 4](#_Toc97326401)

[1.2 Amortised Time Complexity 4](#_Toc97326402)

[1.3 BST Time Complexity 5](#_Toc97326403)

[2. The Great Wall Problem 8](#_Toc97326404)

[2.1 Scenario 8](#_Toc97326405)

[2.2 Justifying Implementation Choices 8](#_Toc97326406)

[2.3 Typical Implementations 9](#_Toc97326407)

[2.4 Complexity of Implementation Choices 9](#_Toc97326408)

[2.5 Solution Approach 10](#_Toc97326409)

[3. The Royal Software Engineers Algorithm 13](#_Toc97326410)

[3.1 Measuring the Royal Software Engineers Algorithm 13](#_Toc97326411)

[3.2 Evaluating the Royal Software Engineers Algorithm 13](#_Toc97326412)

[Bibliography 16](#_Toc97326413)

[Appendix A Graphs & Tables 17](#_Toc97326414)

[Appendix B Typical Implementations 20](#_Toc97326415)

[Appendix C Solutions 24](#_Toc97326416)

List of Figures

[Figure 1. Comparison of Common Time Complexities 4](file:///C:\Users\ryang\Documents\Uni\Year%20Three\Advanced%20SE\Assessment\Assessment%201\Submission\Report.docx#_Toc97326339)

[Figure 2. Solution Efficiency Comparison with 20 Entries 13](#_Toc97326340)

[Figure 3. Solution Comparison with 200 Entries 14](#_Toc97326341)

[Figure 4. Solution Efficiency Comparison with 200 Entries 14](#_Toc97326342)

[Figure 5. Significantly Unbalanced BST 17](file:///C:\Users\ryang\Documents\Uni\Year%20Three\Advanced%20SE\Assessment\Assessment%201\Submission\Report.docx#_Toc97326343)

[Figure 6. Average Insertion for Vector and List 17](#_Toc97326344)

[Figure 7. Solution Comparison with 200 Entries 18](#_Toc97326345)

[Figure 8. Map, Hash Table and Unbalanced BST Lookups 18](#_Toc97326346)

[Figure 9. Lookup Comparison of Vector and List 18](#_Toc97326347)

[Figure 10. Insertion Time for Map, BST and Unordered\_Map 19](#_Toc97326348)

[Figure 11. Typical Unordered\_Map Insertion 20](#_Toc97326349)

[Figure 12. Typical Unordered\_Map Find 20](#_Toc97326350)

[Figure 13. Typical Map Insert 21](#_Toc97326351)

[Figure 14. Typical Map Find 21](#_Toc97326352)

[Figure 15. Typical List Insert 22](#_Toc97326353)

[Figure 16. Typical List Find 22](#_Toc97326354)

[Figure 17. Typical Vector Insert 22](#_Toc97326355)

[Figure 18. Typical Vector Find 23](#_Toc97326356)

[Figure 19. Quadratic Solution 24](file:///C:\Users\ryang\Documents\Uni\Year%20Three\Advanced%20SE\Assessment\Assessment%201\Submission\Report.docx#_Toc97326357)

[Figure 20. Linear Solution 24](file:///C:\Users\ryang\Documents\Uni\Year%20Three\Advanced%20SE\Assessment\Assessment%201\Submission\Report.docx#_Toc97326358)

[Figure 21. Most Efficient Solution 25](#_Toc97326359)

List of Tables

[Table 1. Time Complexity Comparison for Balanced and Unbalanced BSTs 6](#_Toc97326324)

[Table 2. Implemented BST Time Complexity 7](#_Toc97326325)

[Table 3. Time Efficiency for Each BST Method 7](#_Toc97326326)

[Table 4. Varying Data Structures Time Complexities 10](#_Toc97326327)

[Table 5. Insertion and Lookup Time for Different Data Structures 11](#_Toc97326328)

[Table 6. Combined Data Structures Complexity 12](#_Toc97326329)

# Complexity Analysis of a BST Implementation

## Time Complexity

Time complexity is the concept of determining a program’s ability to scale and is useful for comparing different approaches to problems where the amount of users or data can vary. This is different to measuring a program’s efficiency to complete a problem, which tells us the rate at which a fixed data size can be solved. This means that, one solution could be more efficient than another at solving a solution involving one hundred entries, but slower at solving a solution involving one thousand entries, and this is why, not only program efficiency, but time complexity are crucial factors to consider when approaching a problem.

There are six, time complexities that are considered the most common to come up when analysing the scalability of code. As you can see from the comparison made in figure 1, the time complexities listed below, are listed in order of scalability, where “n” is the size of data impacting the system, whether that is the number of entries within a data structure, or the number of users concurrently accessing a server.

**Chart, line chart

Description automatically generated**

* Constant O(1)
* Logarithmic O(log2(n))
* Linear O(n)
* Loglinear O(n × log2(n))
* Quadratic O(n2)
* Exponential O(2n)

Figure . Comparison of Common Time Complexities

## Amortised Time Complexity

Time complexity in itself, is the complexity measurement of a singular operation i.e. a singular insertion or lookup. Amortised complexity, however, is the concept of measuring complexity for a series of operations, to provide a more accurate understanding of worst-case complexity per operation.

A simple example of this, is a lookup within a balanced binary search tree. If the worst case for a singular lookup is O(log2 n), then the worst case amortised complexity is O(m\*log2(n)) where “m” is the number of operations. However, in more complex scenarios, it is not this straight forward to calculate the complexity, as unpredictable behaviour can occur at unexpected times.

For example, an unordered\_map within the standard C++ library, with balanced binary search trees implemented as buckets. The typical average case insertion cost would be O(1) as all the entered values would be inserted into almost empty buckets if not empty buckets. However, the worst case scenario would involve inserting each value into the same bucket, meaning a necessary traversal of the tree, leading to a complexity of O(log2 n).

If an insertion into the unordered\_map led to reaching the load factor threshold, the unordered\_map would be trigged to dynamically resize, typically doubling in size. Requiring all entries “n” to be rehashed and reinserted. However, with this being a worst-case complexity, all reinserted hash values could be reinserted into the same bucket, meaning each insertion would require another traversal of the tree, implying linear complexity. This means, the actual worst-case complexity for single insertion into a hash table is O(n\*log2(n)).

Considering the fact, each insertion into the hash table cannot trigger a resizing until another insertion of “n” is made, where “n” is the size of the tree, this is not the complexity for each worst-case insertion, meaning the worst-case insertion including a resize is O(n\*log2(n)), but a worst-case insertion not including a resize, is O(log2 n). This means the worst-case complexity for a singular insertion is actually O(n\*log2(n) + (n-1) + log2(n)), which breaks down to O(n\*log2(n)).

Finally, to calculate amortised complexity in this more complex scenario, as amortised complexity is the concept of worst-case analysis over a series of operations, we divide the worst-case complexity by the number of operations, which in this case is the values in the tree. Therefore, the amortised complexity for a hash table insertion is O(n\*log2(n)) / n = O(log2(n)).

## BST Time Complexity

Binary search trees are a well-known data structure used within software development, and because of this, the time complexity is well documented for each individual method of the tree. Binary search trees typically consist of a core foundation of methods required to compute “typical” operations, such as inserting new nodes into the tree, removing these nodes, and performing a lookup to check the tree for specific key values.

There are however, two forms of binary search tree, balanced, and unbalanced. A perfectly balanced binary search tree is when the same number of nodes exist on the right side of the tree as on the left, typically allowing for more efficient computing. An unbalanced binary search tree allows for any number of nodes to exist on one side of the tree in comparison to zero on the other side, a great example of which is a singly linked list that has formed from several binary search tree insertions. An example of which can be seen in Appendix A, Figure 5.

The understood and expected time complexity for each method within a typical balanced binary search tree and an unbalanced binary search tree can be seen in table 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Balanced BST | Best Case | Average Case | Worst Case | Amortised Worst Case |
| Insertion | O(1) | O(log2 n) | O(log2 n) | O(m\*log2(n)) |
| Removal | O(log2 n) | O(log2 n) | O(log2 n) | O(m\*log2(n)) |
| Lookup | O(1) | O(log2 n) | O(log2 n) | O(m\*log2(n)) |
| Unbalanced BST |  |  |  |  |
| Insertion | O(1) | O(log2 n) | O(n) | O(m\*n) |
| Removal | O(log2 n) | O(log2 n) | O(n) | O(m\*n) |
| Lookup | O(1) | O(log2 n) | O(n) | O(m\*n) |

Table 1. Time Complexity Comparison for Balanced and Unbalanced BSTs

The insertion of nodes in a balanced BST has a constant best-case complexity as the tree does not need to traversed beyond the first node, and therefore always takes the same amount of time. The tree has logarithmic worst case complexity when the node needs to be inserted at the very bottom of the tree, this is not linear time however, as the tree is balanced, and therefore would not directly increase in time as more nodes are inserted, as these nodes would not necessarily need to be visited.

A balanced BST removal has a best-case logarithmic complexity, as unlike an insertion, any removal from the tree, balanced or not, requires the determination of a replacement node, even if the node being removed is the root node. The worst case remains logarithmic however, as regardless of the removal, a replacement node is still required, with no additional operations.

Searching a balanced BST has a best case of constant complexity, as the tree would not need to be traversed beyond the first node. The worst-case complexity on the other hand is linear, as the traversal of the tree would involve checking every single node present and is therefore directly influenced by the number of nodes, as apposed to the height of the tree.

Insertion within an unbalanced BST has a best case of constant complexity similarly to a balanced insertion, as the complexity of inserting the first node in the tree, is not impacted by how balanced the tree is, as the tree is already perfectly balanced. However, the worst-case complexity of this insertion is linear, as the tree exists as Appendix A, Figure 5 demonstrates, where the entire length of the tree including all nodes, would need to be traversed to insert the node.

The time complexity of a removal from an unbalanced BST, however, is linear, unlike a balanced BST, as previously mentioned, an unbalanced tree in it’s worst state can be represented as a singly linked list, shown in Appendix A, Figure 5. Meaning all nodes would be traversed to remove the desired node.

Searching an unbalanced BST however, similarly to the removal of a node from an unbalanced tree, in its worst case, would require the traversal of all nodes within the tree, therefore becoming directly impacted by the number of nodes present, and therefore maintains a linear complexity.

Further analysis of an unbalanced binary search tree that has been implemented, has generated several tables and graphs to better understand the specific time complexity of each member function developed.

The unbalanced binary search tree implementation contains several methods, including the three common practice methods mentioned previously. Table 2 contains the time complexity of an implemented unbalanced BST in practice.

|  |  |  |  |
| --- | --- | --- | --- |
| BST Implementation | Best Case | Average Case | Worst Case |
| Insertion | O(1) | O(Log2 n) | O(n) |
| Removal | O(log2 n) | O(Log2 n) | O(n) |
| Lookup | O(1) | O(Log2 n) | O(n) |
| In-order Traversal | O(1) | O(n) | O(n) |
| Pre-order Traversal | O(1) | O(n) | O(n) |
| Post-order Traversal | O(1) | O(n) | O(n) |
| Constructor | O(1) | O(1) | O(1) |
| Destructor | O(1) | O(n) | O(n) |

Table 2. Implemented BST Time Complexity

When comparing the expected typical implementation time complexity of an unbalanced BST, with that of my implementation, we can see the complexity has not varied, as the best, average and worst-case complexities are the same for Insertion, Removal, and Lookup. indicating the methods for each component were implement to at least a typical standard.

The three types of traversals, all carry a best case complexity of O(1), average case complexity of O(n) and a worst case complexity of O(n), as all the nodes need to be traversed to produce the required traversals, however, the best case scenario only involves a singular node, and therefore no matter the traversal, there is still only one node to traverse.

The BST constructor method has a best case, average case, and worst-case constant complexity, as the number of nodes to be passed as argument, or the height of the tree is irrelevant to the operation of constructing an empty tree. However, the Destructor method only has a best case of constant complexity, when only one node need’s to be deleted to deep delete the entire tree. However, the average and worst-case complexity for destructing a tree, directly depends on the number of nodes present, as each node needs to be visited to be deleted, therefore having linear complexity.

To further analyse and understand the performance of the implemented BST, outside of the best, average and worst-case complexity, table 3 records the time measurement to complete a sample number of operations for each method. As you can see, it takes on average, 2540.2 nanoseconds to complete a singular insertion, however, it only takes 1906 nanoseconds to compute a lookup. This is because a lookup identifies the required node, and does nothing more, whereas an insertion identifies the appropriate insertion position, and then coordinates the construction of either new pointers, or the repositioning of existing ones.

We can also see, that on average, it takes 400,000 nanoseconds to compute a single traversal of the tree, this is because all current nodes within the tree need to be visited to display the result. Whereas a singular lookup, insertion, or removal, on average will only visit half of the nodes.

|  |  |
| --- | --- |
| Method | Nanoseconds |
| Insertion | 500K (2540.2) |
| Removal | 500K (1355) |
| Lookup | 500K (1906) |
| In-order Traversal | 10 (412512.8) |
| Pre-order Traversal | 10 (420334.4) |
| Post-order Traversal | 10 (418125.8) |
| Constructor | 5M (20) |
| Destructor | 5M(240) |

Table 3. Time Efficiency for Each BST Method

# The Great Wall Problem

## Scenario

“A long time ago, a great wall was built along the northern border of an ancient kingdom. After the wall was finished, an artist walked along the wall from west to east, decorating the southern side of each top brick with a unique symbol.

The artist's apprentice was instructed to follow and copy each symbol onto the northern side of each brick. However, the apprentice made a mistake. Instead of copying each symbol onto the back of the same brick, the apprentice actually copied the symbol onto the back of the next brick along to the east. That is, he drew the symbol from southern side of the first brick onto the northern side of the second brick, the symbol from the southern side of the second brick onto the northern side of the third brick, and so forth all the way along the wall. When he reached the eastern end, he realised his mistake, as there was no brick on which to draw the final symbol. In panic, he removed the first brick from the wall, and destroyed it.

The years passed, and the local people gave names to the symbols decorating the wall. They carved these names beneath the symbol on (both sides of) each brick. Many years later, an earthquake shook the kingdom, and the wall came crashing down. Saddened, the King ordered all of the decorated bricks to be brought to his palace. Upon examining the heap of bricks, the Royal Data Scientist observed that it was readily apparent which was the north and south side of each brick, as exposure to sunlight had caused the symbols on one side of the wall to fade more than the other. Thus there was enough information to efficiently determine the original sequence of symbols.”

## Justifying Implementation Choices

The solution to the above scenario, involves following the below steps.

1. Load the information from each brick into main memory, organising it in a manner suitable for efficient searching.
2. Arbitrarily choose one of the bricks as a starting point.
3. Taking the two symbol names from the starting brick, start constructing a result sequence elsewhere in main memory, northern name followed by southern name.
4. Repeatedly, until no matching brick is found:
   1. Search for the brick with a northern symbol that matches the back (easternmost) symbol in the result sequence.
   2. Add the southern symbol name from that brick to the back of the result sequence.
5. Repeatedly, until no matching brick is found:
   1. Search for the brick with a southern symbol that matches the front (westernmost) symbol in the result sequence.
   2. Add the northern symbol name from that brick to the front of the result sequence.

This solution can be implemented through several combinations of data structures including arrays, vectors, binary search trees, hash maps, linked lists etc. However, only a number of these solutions are efficient and scalable, therefore, only a select group of varying combinations of data structures will be considered to determine the ideal solution.

All considered data structures are within the std standard library and the combinations being analysed includes vector and unordered\_map, vector and map, list and unordered\_map, and list and map. For clarification, std::unordered\_map is implemented as a hash table, and std::map is implemented as a self-balancing binary search tree. They have been chosen through suggestion but are also useful as they share similar implementations in terms of written code and are appropriate for this problem, for reasons discussed later. It is also worth noting, list is being used due to suggestion, but vector is being used as it also provides a comparison point as well as a similar implementation.

The most significant reason for vectors being, they allow for random access, as they are implemented through contiguous memory. Whereas, lists are implemented through pointers, and therefore do not allow random access.

## Typical Implementations

A typical implementation for an insertion and lookup using the vector class can be seen in Appendix B, Figures 17 and 18. As you can see, the sample implementation demonstrates the use of the “push\_back” method to allow for constant insertion as discussed in table 4, storing the current value at the end of the data structure, and the sample lookup implementation demonstrates the use of the “find” method along with an iterator structure to loop through the vector demonstrating the output of all values, however, simply outputting a singular value, would not require the iterator class.

A typical implementation for inserting and searching a list class, also referred to as a doubly linked list, is shown in Appendix B, Figures 15 and 16, where the “push\_back” and “find” method are demonstrated again. The methods are similar as a doubly linked list allows for the insertion of values to the front or back of the structure, just like a vector. Due to the similarities in coding to implement both data structures, it allows for easier implementation of both, allowing for ease of testing their efficiency and complexity.

A standard implementation for inserting and searching a map, also known as a balanced binary search tree, demonstrates the use of the “insert” method and the “find” method in Appendix B, Figures 13 and 14, which will work the same way as an insertion or search within an unbalanced binary search tree, like the binary search tree implemented and discussed previously.

Lastly, and typical insertion and lookup with the unordered\_map class can be seen in Appendix B, Figures 11 and 12, showing the use of the “insert” method for insertion, and the “find” method for lookups. Although the method is the same, the functionality of the find method differs with an unordered\_map as it is a hash table, containing a key and a value, similar to a binary search tree, but dissimilar to a single value within a vector or list. The find method takes a key as argument, and returns the associated value, or a pointer to the end of the data structure if there is no match.

## Complexity of Implementation Choices

As shown in table 4, for an unordered\_map insertion, the worst-case complexity is O(m\*log2(n)), as the hash table will be full, and will need to resize, typically involving a resize of double the initial size of the table, meaning all current values will be iterated through to get the size, then new space will be designated. The removal for a hash table shares the same complexity, as all of the values within the hash table are within the same bucket, meaning the hash function used to narrow down the number of values to be checked, actually returns every value within the table, and because of this, a lookup within a hash table also has the same complexity, as all values need to be visited.

A Map insertion has a worst-case logarithmic complexity as a balanced binary search tree, will never need to visit all nodes to find the correct insertion spot for any node. This is also the case for a removal, with worst case logarithmic complexity, as the tree is balanced, not all nodes will need to be visited to determine the maximum or minimum node. Lastly, the case is the same for a Map lookup, as not all nodes will need to be visited to determine the correct position of the node being looked for.

A list insertion has worst case complexity of linear time, as a doubly linked list can represent that of Appendix A Figure 5, where all nodes within the list will need to be visited to insert a node at the desired location. This is also the case for a lookup or removal within a list, as removal from the middle of a list, or performing a lookup on the middle of a list, will require the list to be iterated over, requiring linear time.

A vector insertion has a worst-case linear complexity, as inserting into a vector, causing it to be full, will require the repositioning of all the existing data within the vector to another memory location. Removal from a vector, however, is worst-case constant complexity, as vectors allow for random access, meaning the length of the vector, or the node being removed, has no bearing on the complexity of the operation. Lastly, searching a vector has worst case constant complexity like a removal, as it allows for random access, meaning the size of the vector will not impact the operation.

|  |  |  |  |
| --- | --- | --- | --- |
| Unordered\_map | Best Case | Average | Worst Case |
| Insertion | O(1) | O(1) | O(m\*log2(n)) |
| Removal | O(1) | O(1) | O(m\*log2(n)) |
| Lookup | O(1) | O(1) | O(m\*log2(n)) |
| Map |  |  |  |
| Insertion | O(1) | O(log2 n) | O(log2 n) |
| Removal | O(log2 n) | O(log2 n) | O(log2 n) |
| Lookup | O(1) | O(log2 n) | O(log2 n) |
| List |  |  |  |
| Insertion | O(1) | O(1) | O(n) |
| Removal | O(log2 n) | O(log2 n) | O(n) |
| Lookup | O(1) | O(1) | O(n) |
| Vector |  |  |  |
| Insertion | O(1) | O(1) | O(n) |
| Removal | O(1) | O(1) | O(1) |
| Lookup | O(1) | O(1) | O(1) |

Table 4. Varying Data Structures Time Complexities

## Solution Approach

Due to the nature of the scenario, and the proposed solution, one data structure will be used to store the original sequence and will also need to be traversed in order to solve the sequence, and a second data structure will be used to store the solved sequence but will not need to be traversed. Therefore, the data structure which is most scalable and most efficient for lookups should be the best choice for the data type storing the original sequence, and the data structure most scalable and efficient for insertions, should be the best choice for inserting the solved sequence.

As the ability to efficiently perform lookup operations at a scalable rate is the priority for the solution, the data structures with the most scalable time complexities are most appropriate.

To make a decision about which two data structures are most appropriate for the problem, I will be comparing and discussing the findings of table 5, with Appendix A, Figures 6, 8, 9, and 10. In table 5, as well as the figures aforementioned, we can see that vector and list, have significantly worse time measurements when performing lookups when compared to that of unordered\_map and map. Because of this, and the fact vector and list have a significantly lower time taken to perform insertions, vector and list are much more suited to store the final solution, as apposed to being traversed to solve the sequence.

With that in mind, when once again comparing the insertion time and lookup time between vector and list, and unordered\_map and map, with lookup time being the priority, it is clear to see, that unordered\_map and map are better solutions for storing the original sequence, and to be traversed when solving the sequence.

I made the decision to not include an unbalanced binary search tree, as a map is already acting as a balanced tree. I felt it was necessary to include in my investigation to identify the time difference between average case balanced and unbalanced, but don’t see the use to include the structure any further when solving the problem, as it will hold no additional benefit.

|  |  |  |
| --- | --- | --- |
| Data Structure | Insertion (500K) | Lookup |
| List | 467.4 | 891833 |
| Vector | 178.8 | 53635.6 |
| Map | 4039.6 | 1603.8 |
| Unordered\_map | 3142.6 | 547 |
| Unbalanced BST | 1767.2 | 1906 |

Table 5. Insertion and Lookup Time for Different Data Structures

To calculate the complexity for a combination of data structures, they are added them together and any constant values are dropped. For example, the worst-case complexity for an unordered\_map with vector insertion is linear, as the worst-case complexity for an unordered\_map insertion is O(n\*log2(n)), and the worst-case insertion for a vector is O(n). Therefore, O(n) + O(n\*log2(n)) with dropped constants is equal to O(n\*log2(n)), and the same applies for lookup with this combination.

|  |  |  |  |
| --- | --- | --- | --- |
| Unordered\_Map + Vector | Best Case | Average | Worst Case |
| Insertion | O(1) | O(1) | O(n\*log2(n)) |
| Lookup | O(1) | O(1) | O(n\*log2(n)) |
| Unordered\_Map + List |  |  |  |
| Insertion | O(1) | O(1) | O(n\*log2(n)) |
| Lookup | O(1) | O(1) | O(n\*log2(n)) |
| Map + Vector |  |  |  |
| Insertion | O(1) | O(log2 n) | O(n\*log2(n)) |
| Lookup | O(1) | O(log2 n) | O(n\*log2(n)) |
| Map + List |  |  |  |
| Insertion | O(1) | O(log2 n) | O(n\*log2(n)) |
| Lookup | O(1) | O(log2 n) | O(n\*log2(n)) |

Table . Combined Data Structures Complexity

As another example, the worst-case complexity for an unordered\_map with a list, follows the same rules as above, as the worst-case insertion for a vector is O(n), and the complexity for an unordered\_map is O(m\*log2(n)). Therefore the worst-case insertion complexity for an unordered\_map and vector is O(m\*log2(n)). The same applies for the lookup complexity, when combining the two worst case complexities, the result is O(n\*log2(n)).

As shown in table 6, each combination of data structures has the same worst-case lookup and insertion complexity. However, when comparing the average case complexity of each approach, we can see that unordered\_map in any combination, has the more scalable complexity, and therefore would be the more appropriate and efficient solution as the data size increases.

From tables 5 and 6, and Appendix A, Figures 6, 8, 8, and 10, as well as the fact it allows for random access leading to better time measurements, we can assume the combination of unordered\_map for main storage of the wall as it has a more scalable average case insertion and lookup as well as a lower time per insertion and lookup compared to map, and vector for the secondary storage containing the solution sequence as it has the same worst case complexity for insertions, but when timing an average case insertion, vector clearly has a more efficient insert, allowing me to make the prediction that these two structures would provide the most efficient result, as well as assuming it will be the joint most scalable solution.

# The Royal Software Engineers Algorithm

## Measuring the Royal Software Engineers Algorithm

To introduce my evaluation of the Great Wall Problem, I will first reiterate, that from my investigation into the complexity and efficiency of different combinations of data structures, I expect the combination of Unordered\_map and vector to be the most scalable and efficient approach to the problem.

There are many approaches that can be taken to solve the Great Wall Problem, two great examples of which produce two different time complexities, clearly indicating the more scalable and efficient choice.

The solution shown in Appendix C, Figure 20, demonstrates a worst-case linear solution, and the solution in Appendix C, Figure 21, demonstrates another worst-case linear solution, however, the solution in figure 21, is much more efficient, so while it will be just as scalable, it will still solve the problem at a more efficient rate. This can be seen when comparing figures 2 and 4 below, the difference between twenty and two hundred entries.

Chart, bar chart

Description automatically generated

Figure 2. Solution Efficiency Comparison with 20 Entries

As shown in figure 2, unordered\_map combined with a list, or combined with a vector returned very similar times between 120,000 nanoseconds and 150,000 nanoseconds. This is relatively surprising considering the insertion time difference between list and vector on their own. Although, it is worth considering that 20 entries is simply not enough to see the insertion difference between list and vector, outside of a slight indication.

I found when comparing these results figure 2 and figure 4 that unordered\_map with list, is significantly more efficient per solution than unordered\_map with vector, which is surprising considering the insertion time difference identified earlier in the document, going against my expectation that vector would be the more efficient solution storage.

## Evaluating the Royal Software Engineers Algorithm

An evaluation of my four different implementations of two separate solutions, I have found that the solution involving unordered\_map with vector, and unordered\_map with list (hash + list), is the most efficient solution, and I have found that for entries above 20, unordered\_map with list is the most efficient overall solution at all levels of “wall size” aka text file entries.

Chart, bar chart

Description automatically generated

Figure 3. Solution Comparison with 200 Entries

When comparing three different solutions in figure 3, this does not indicate or expand on the time complexity of each solution, but it does highlight the difference in efficiency between them. There is a clear efficiency difference shown, where the quadratic complexity solution is “Map + List”, producing a time measurement of 24,440,710 nanoseconds to solve a sequence of two hundred entries shown in Appendix C, Figure 19, the most up to date Map inclusive linear solution is “Map + List New” with 19,144,807 nanoseconds to solve the same sequence, the code for which is shown in Appendix C, Figure 20, and lastly, the most efficient solution is unordered\_map combined with a list, solving the same sequence in 1,290,158 nanoseconds, which can be seen in Appendix C, Figure 21.

This can also be seen when plotting average case time measurements across five instances for each of the four final approaches, each with a different combination of data structures. Although this figure compares the latest solution for each combination, as appose to the different versions of solutions over the course of development, the figure does still demonstrate the efficiency difference between an unordered\_map and a map.

Chart, bar chart

Description automatically generated

Figure 4. Solution Efficiency Comparison with 200 Entries

In order to create an even more efficient solution, this would require further analysis of the unordered\_map being implemented, as a deeper understanding behind the bucket organisation, and how the hash function and load factor being used impacts the efficiency of searching, inserting and removing data from the table, as having a deeper understanding of these factors would allow for a more fine-tuned solution to be developed specifically for this problem.

# Bibliography

En.cppreference.com. 2022. *std::vector - cppreference.com*. [online] Available at: <https://en.cppreference.com/w/cpp/container/vector> [Accessed 20 February 2022].

En.cppreference.com. 2022. *std::unordered\_map<Key,T,Hash,KeyEqual,Allocator>::insert - cppreference.com*. [online] Available at: <https://en.cppreference.com/w/cpp/container/unordered\_map/insert> [Accessed 26 February 2022].

GeeksforGeeks. 2022. *std::find in C++ - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/std-find-in-cpp/> [Accessed 17 February 2022].

GeeksforGeeks. 2022. *map find() function in C++ STL - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/map-find-function-in-c-stl/> [Accessed 2 March 2022].

En.cppreference.com. 2022. std::map - cppreference.com. [online] Available at: <https://en.cppreference.com/w/cpp/container/map> [Accessed 29 February 2022].

En.cppreference.com. 2022. std::list - cppreference.com. [online] Available at: <https://en.cppreference.com/w/cpp/container/list> [Accessed 30 February 2022].

En.cppreference.com. 2022. std::unordered\_map<Key,T,Hash,KeyEqual,Allocator>::insert - cppreference.com. [online] Available at: <https://en.cppreference.com/w/cpp/container/unordered\_map/insert> [Accessed 2 March 2022].

En.cppreference.com. 2022. std::hash<Key>::operator() - cppreference.com. [online] Available at: <https://en.cppreference.com/w/cpp/utility/hash/operator()> [Accessed 2 March 2022].

En.cppreference.com. 2022. std::vector<T,Allocator>::push\_back - cppreference.com. [online] Available at: <https://en.cppreference.com/w/cpp/container/vector/push\_back> [Accessed 2 March 2022].

En.cppreference.com. 2022. std::iterator - cppreference.com. [online] Available at: <https://en.cppreference.com/w/cpp/iterator/iterator> [Accessed 1 March 2022].

# Appendix A Graphs & Tables

Diagram

Description automatically generated

Figure . Significantly Unbalanced BST

**Chart, line chart

Description automatically generated**

Figure . Average Insertion for Vector and List

Chart, line chart

Description automatically generated

Figure 7. Solution Comparison with 200 Entries

Chart, bar chart

Description automatically generated

Figure 8. Map, Hash Table and Unbalanced BST Lookups

Chart

Description automatically generated with medium confidence

Figure 9. Lookup Comparison of Vector and List

Chart, line chart

Description automatically generated

Figure 10. Insertion Time for Map, BST and Unordered\_Map

# Appendix B Typical Implementations

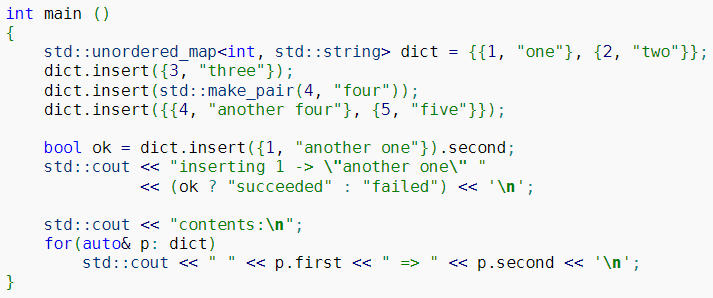


Figure 11. Typical Unordered\_Map Insertion

Text

Description automatically generated

Figure 12. Typical Unordered\_Map Find

Text

Description automatically generated

Figure 13. Typical Map Insert

Text

Description automatically generated

Figure 14. Typical Map Find

Graphical user interface, text, application

Description automatically generated

Figure 15. Typical List Insert

Graphical user interface

Description automatically generated with low confidence

Figure 16. Typical List Find

Graphical user interface, text, application

Description automatically generated

Figure 17. Typical Vector Insert

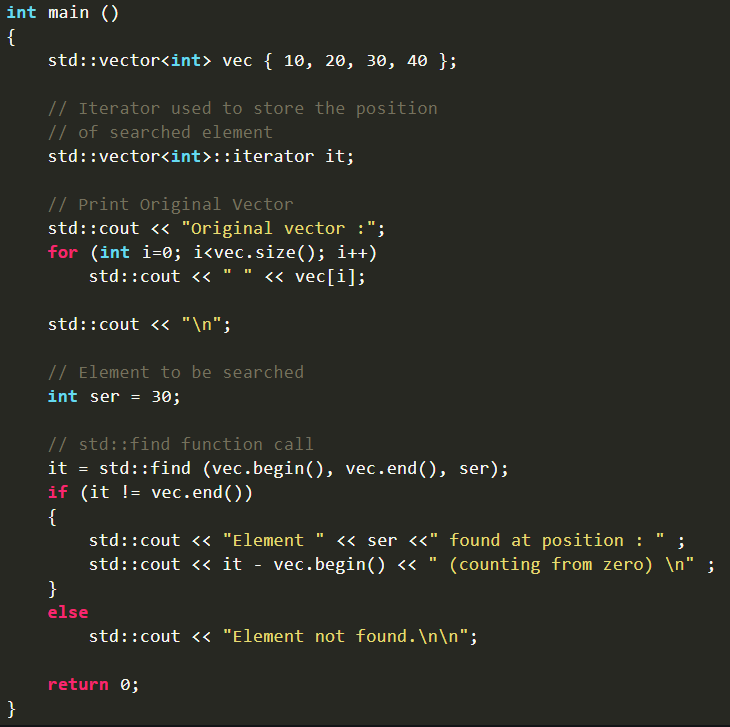


Figure 18. Typical Vector Find

# Appendix C Solutions

Text

Description automatically generated

Figure 19. Quadratic Solution

Text

Description automatically generated

Figure 20. Linear Solution

A screenshot of a computer

Description automatically generated with medium confidence

Figure 21. Most Efficient Solution