# Task 1

## Constructor

This is the default constructor, it has the default time complexity (assuming it’s O=1)

## Destructor

The destructor loops through all elements, this gives it O=n being n the amount of elements

## Lookup

The lookup loops through all elements trying to find the correct one and automatically discards half the set, it’s similar to a bubble sort and its complexity is O=log n

## Insert

For the insert it is very similar as with the lookup but with the difference once we find the current element associated to the key or find a leaf, an element is created. It’s algorithmic complexity is the same, O=log n

## DisplayEntries

Display entries is like a lookup with another loop to display the items, its complexity is O=2log(n)as it’s got to traverse the amount of elements twice.

A very similar algorithm has been introduced for all traversals required across the functions, it might not be the most efficient one but it does work and by using this, code can be made function-independent in the future and reuse certain parts of the algorithm into a better function structure.

# Task 2

a)

Std::list Is internally structured as a doubly-linked list, which means it has a pointer to both the next and the previous element along with the data payload contained on each entry. The class is also traversable both backwards and frontwards as it is iterable. This means that items can be added to the list using the native functions insert() and emplace() for the front-iteration as well as push\_back() and emplace\_back() for the backwards one, this is important as it lets us decide exactly the order in which we’d like to insert the data, though it is not the fastest one for traversing as no “Jumping” between items can be done, the list must be followed thoroughly from beginning to end or vice-versa until the desired item is found. Inserting in a list gives a complexity of O(1) as you can indicate the position in which it is to be stored or O(n) in the worst case scenario if list traversing needs to be done. For the lookup, as we need to loop through the list, again it’s O(1) in the best case to O(n) in the worst. Also do note that as std::lists do not store information contiguously (like std::vectors do) they are much slower to traverse.

Std::map are internally structured in the old-fashioned key-value pair list and internally uses a Binary Search Tree to keep itself sorted. These have a series of very useful functions such as the begin() and end() which return an iterator with the first and las position in the map, which is used when iterating through it to lookup for values. It also has other functions such as size() and empty() which are used in various contexts to assess how to handle the structure effectively. This structure is automatically sorted by key, which makes it good for The Royal Software Engineer’s problem. However, it’s not the best structure for searching keys. It uses insert() for insertion as well as the operator [], main difference is that the operator will automatically replace if an item already exists. When finding one particular key it uses the function find() which provides an iterator to the element or the end iterator of the map. In terms of time-complexity guarantees, the map has log(n) for searching as a binary tree does, which is the same for insertion but with the difference it needs to balance/sort itself again after each operation, which adds an overhead.

Std::unordered\_map is very similar to a map but it is not sorted by default. It uses hash tables instead of a linked structure and this allows for much faster searching. Provided that order is not of concern, this structure performs better than its simple brother, the std::map, particularly when finding keys all along the structure, but is not so optimal for iterating only a subset of the elements when compared to the map. In terms of time-complexity guarantees, this type of map has O(1) for the average scenario and o(n) on the worst case assuming the element is the very last to be accessed. For insertion it has the same complexity as for searching, this is due to the fact that it does not need to sort itself after the insertion.

b)

One of the possibilities could be to implement a solution which used an unordered map for storing all the bricks and lookup the map to find each brick, then storing its information in the relevant position on a list (with pointers to the next and previous bricks once discovered). This would benefit from the fast mesh-search capabilities of the unordered map but storing in the list could be less efficient and neat.

Another possibility would be to use an unordered map again for storing all the bricks, since its lookup proves to be the fastest of all these structures but instead storing in a map, which is sorted by default and is an associative structure, like the unordered map, which makes processing data more time-efficient as well. Hence why I believe this is the best approach for the Royal Software Engineer’s problem. However, this does not use the algorithm as suggested but instead uses the fact that data is sorted by key on a map, which means that every brick will fall into its desired place automatically without the need to find them one by one (as the keys will get sorted). This should get the same final result in a much smaller time if I’m right, it will only work in a real-life situation provided that the keys are sorted before the brick wall is destroyed (let’s say bricks were 4-5-6-7 west-to-east bound, in this case it would work but if bricks were layered out in a random order it would not)

To implement the actual algorithm proposed I will use a different combination of structures as it makes more logical sense. This uses an unordered map for storing the pile of bricks and a list to rebuild it. I chose this mainly because the list is capable of emplacing to the front and back which is indispensable in order to implement the algorithm as it has been specified.