**Task 1b**

Constructor

Time complexity, best, average and worst case: O(1) – Constant

Reason: The function has no parameters and its running time is independent of the size of the linked list.

Destructor

Time complexity, best, average and worst case: O(n) — Linear

Reason: The recursive function ‘deepDelete’ goes through each of the items in the dictionaries linked list to delete the items. This means the functions execution time is dependant of the size of the number of items in the dictionaries linked list.

Move constructor

Time complexity, best and worst case: O(n) — Linear

Reason: The recursive function ‘deepCopy’ performs actions on each of the items in the dictionaries linked list. This means the functions execution time is dependant of the size of the number of items in the dictionaries linked list.

Copy assignment operator

Time complexity, best and worst case: O(n) — Linear

Reason: The recursive function ‘deepCopy’ performs actions on each of the items in the dictionaries linked list. This means the functions execution time is dependant of the size of the number of items in the dictionaries linked list.

Move assignment operator

Time complexity, best and worst case: O(n) — Linear

Reason: The recursive function ‘deepCopy’ performs actions on each of the items in the dictionaries linked list. This means the functions execution time is dependant of the size of the number of items in the dictionaries linked list.

Insert

Average and worst-case time complexity: O(n) — Linear

Reason: The nodes of the linked list need to be iterated through in order to find the position of insertion. This means the functions execution time is dependant of the size of the number of items in the dictionaries linked list. While inserting may occur half-way through insertion of the list for example, this is still considered in terms of time complexities to be dependant upon the number of items in the list.

Best case time complexity: O(1) – Constant

Reason: If the key to be inserted happens to be in the root position then regardless of the number of elements in the list, the operation will take a constant time.

Lookup

Average and worst-case time complexity: O(n) — Linear

Reason: The nodes of the linked list need to be iterated through in order to find the position of the search node. This means the functions execution time is dependant of the size of the number of items in the dictionaries linked list.

Best case time complexity: O(1) – Constant

Reason: If the node to be searched is the root node, then finding it will always have a constant time complexity since it is the first node to be searched from.

Remove

Time complexity, best and average case: O(n) — Linear

Reason: The nodes of the linked list need to be iterated through in order to find the key of the node to remove. This means the functions execution time is dependant upon the size of the number of items in the dictionaries linked list.

Best case time complexity: O(1) – Constant

Reason: If the node to be removed is the root node, then finding it will always have a constant time complexity since it is the first node to be searched from.

RemoveIf

Worst and average case time complexity: O(n^2) — Quadratic

Reason: The nodes are iterated through in order to be passed to the higher order function. Nodes which then need to be removed are once again iterated through in the remove function. This means that the function has operations which are dependent on the size of the list of the dictionary, each of which is also dependant on the size of the list, (O(n \* n) ).

It would be worth noting however that the higher order function, which is passed a nodes key, could call other functions of the dictionary, which have other time complexities.

Best case time complexity: O(n) — Linear

Reason: If the higher order function is never to return true, none of the nodes will be deleted. Therefore, the nodes will only be iterated through once when passed to the higher order function, an action with a running time dependant on the size of the linked list.

**Task 2a.**

std::list

A doubly linked list is used to implement the std::list container. This data structure is composed of a list of nodes each of which is linked to its previous and preceding node within the list. This allows for bi-directional traversal through the list.

Both searching (through iterators) and inserting (list.push\_back, list.push\_front, list.insert) operations on an std::list require a pointer to the node containing the stored value. Best and worst cases for locating this node have the time-complexity of O(n), as the list needs to be traversed, meaning the operation depends on the number of nodes in the list. However, this is not the case for the best case time complexity of the data structure since it has a head and a tail pointer, thanks to it being implemented with a doubly linked list, these operations at the front and back of the list have a constant time-complexity (O(1)).

std::map

The container std::map has the underlying data structure of a red-black tree. A red-black tree is a self-balancing binary search tree (BST). BST’s are composed of nodes which hold values, each node has a left and right child node. Left child nodes have a lower value than that of their parent, while right child nodes have a higher value. This means nodes are kept in sorted order by their values. The self-balancing aspect of a red-black tree attempts to ensure there are as many nodes on the left of the root node as there are on the right. This allows for the fastest average search and insertion time.

Thanks to layout of nodes, searching skips about half of the tree so that each lookup (map.find()), insertion (map.insert()) or deletion takes time proportional to the logarithm of the number of nodes stored in the tree; O((n)). This is often, especially with large numbers of elements, a much quicker approach to operations than that of the general std::list operations. However, this scaling is not always accurate, as a red-black tree, as discussed above is self-balancing meaning it attempts to keep the same number of nodes on each side of each node if possible. Due to this insertion operations occasionally take longer.

std::unordered\_map

A hash table data structure is used in the implementation of std::unordered\_map. A hash table consists of a group of unordered buckets or slots in which values are stored. A hash function is used to calculate the index of these buckets or slots in order to access their stored values. Thanks to the calculation of the index, searching (map.find()), and insertion (map.insert()) operations have a constant time complexity.

It is also worthy to note that unordered maps occasionally require a re-hashing. This causes operations to have a time complexity of O(n). This is due to all of the ‘buckets’ within the data structure being assigned a new index due to a change in how the hash function works, and occurs due to a size change of the structure.

**Task 2b.**

As a result of the large number of pairs of names the algorithm will be created for, analysing time complexities of the data structures used is important in determining its performance consequences. The search-based algorithm requires at least two data structures; one for loading the file contents and searching from and one to store the results.

Loading file contents and searching

No matter which container is used to store the file contents before it is searched from, the time complexity of the operation will be at least O(n). This is due to the file needing to be read from line by line, a task dependant upon the number of lines in the file. Both std::list and std::unordered\_map seem like good choices for storing the data due to their insertion time complexities of O(1) (this is the case for std::list since insertion will amend the end of the list).

Like loading from the file, searching from the container used will also carry a time complexity of at least O(n), this is because each element needs to be searched for. The most optimal data structure for searching from in this context is the unordered\_map. This is because of its search time complexity of O(1) compared to that of std::list with O(n) and std::map with O((n)).

As a result of analysis of the time-complexities of each of the container types on both insertion and searching, std::unordered\_map seems the best to use in order to fulfil the task of loading the file contents and being searched from.

Storing results

Since the container used to store results requires inserting elements at its front and back, of the three containers std::list seems a perfect choice. This is because of the doubly linked list data type used to implement the container which requires head and tail pointers, meaning performing the insertion operations on this data type would have a constant time complexity which is of use due to the large number of elements needing to be processed.

While std::unordered\_map also has a time complexity of O(1) for its insertion operations, the execution time for its hash function to retrieve the required index value means insertion operations are predictably longer than that of the std::list data structure.

Due to the large number of elements needing to be processed, std::map is a less preferable choice than std::list for the storing of results, thanks to it’s is O((n)) insertion time.

Memory

Using the containers to implement the algorithm will have significant memory use implications. This is because the entire file contents will need to be stored in working memory for them to be searched from. Also storing the results of the algorithm in a container will mean the data of that container will also be in memory.

Conclusion

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Container used to store file contents and search from | Container used to store results | Time-complexity of reading file and loading container | Time-complexity of searching each element from container | Time-complexity of storing in results container | Overall complexity |
| list | list | O(n) |  | O(1) |  |
| list | map | O(n) |  | O((n)) | O((n)) |
| list | unordered\_map | O(n) |  | O(1) |  |
| map | list | O((n)) | O((n)) | O(1) | O((n)) |
| map | map | O((n)) | O((n)) | O((n)) | O((n)) |
| map | unordered\_map | O((n)) | O((n)) | O(1) | O((n)) |
| unordered\_map | list | O(n) | O(n) | O(1) | O(n) |
| unordered\_map | map | O(n) | O(n) | O((n)) | O((n)) |
| unordered\_map | unordered\_map | O(n) | O(n) | O(1) | O(n) |

As a result of this analysis, the best combination of data structures for the algorithm seems to be std::unordered\_map for insertion and searching of the data, and std::list for storage of the results. It is however important to note that the time-complexity of the containers operations does not equate to speed at which the operations are performed, but only how these operations scale with the number of elements they deal with. Time complexity is however a good rule of thumb to use when deciding the implementation of the algorithm due to the large number of elements to be used by the algorithm.

**Task 3b**

The two combinations of containers used to implement the algorithm were:

* unordered\_map for storing and searching the data combined with a list for storing its results
* map for storing and searching combined with a list for storing the results

Both implementations of the algorithm were subject to 11 input files of differing sizes in which their execution time was recorded in order to analyse how this time scaled with input size. Below are the results of the unordered\_map and list combination.

|  |  |
| --- | --- |
| **unordered\_map with list algorithm** | |
| **Number of names in input file** | **Algorithm completion time (micro seconds)** |
| 1000 | 69933 |
| 2000 | 133898 |
| 5000 | 352969 |
| 20000 | 1426618 |
| 50000 | 3430850 |
| 100000 | 6954964 |
| 200000 | 14018093 |
| 500000 | 34075173 |
| 1000000 | 71495230 |
| 2000000 | 143713844 |
| 3000000 | 225855834 |

The results of the performance data suggest that, like the analysis made in task 2b, the algorithm implemented with an unordered\_map in combination with a list has a linear time complexity. A slight curvature upwards is seen towards the later stages of the algorithm, however this could be as a result of the larger number of input values requiring a more time-consuming hash function, or causing more hash collisions.

Below are the results of the analysis of the algorithms implementation with a map for loading the file contents and searching from combined with a list for storing results.

|  |  |
| --- | --- |
| **Map with list algorithm** | |
| **Number of names in input file** | **Algorithm completion time (micro seconds)** |
| 1000 | 89201 |
| 2000 | 150436 |
| 5000 | 397685 |
| 20000 | 1553380 |
| 50000 | 3954307 |
| 100000 | 8269785 |
| 200000 | 17007800 |
| 500000 | 44149317 |
| 1000000 | 94291178 |
| 2000000 | 198769552 |
| 3000000 | 307914784 |

The results of the performance data above, while curving slightly, does resemble that of linear time complexity scaling. This was unexpected as the predicted time complexity made in task 2b was O((n)).

The graph below is of both versions of the algorithm, which allows for a comparison to be drawn between the two. It can be seen, like the prediction made in task 2b that the unordered\_map implementation scaled the best of the two. For the large input of 3 million names that the algorithm was intended for, this more efficient scaling meant that the unordered\_map implementation, as predicted, was also the quickest.

* A graph of both versions of the algorithm (input file size by algorithm completion time in micro seconds)

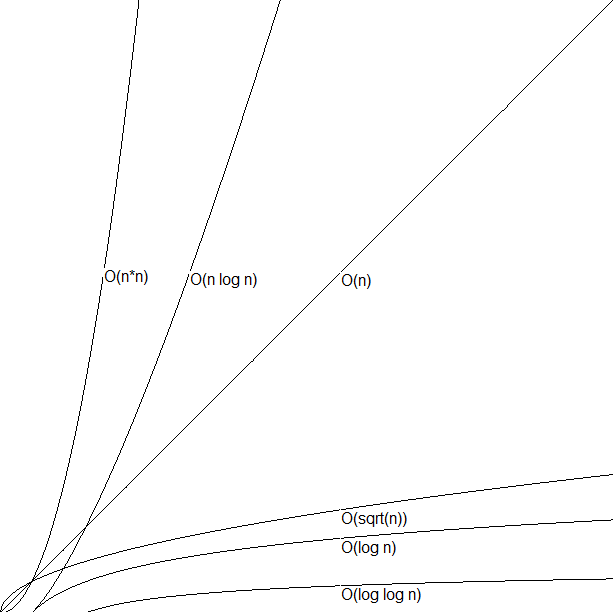


Figure 1 – A rough sketch of various time complexities.

**Task 4a**

Splitting up the algorithm solution into three separate parts allows for better analysis of its time complexity. The two parts are; the (xi, xi+2) file creation, the inductive process, and the sorting of G.

During the first section of the algorithm the file contents are read, sorted in two ways and put in two separate files respectively, then these two files are used to create a third file. For each of these operations, apart from the sorting, each element of a file is read once and used to produce the operations result. It can therefore be concluded that each of these operations, apart from the sorting, have a time complexity of O(N), as the scaling of the time of each operation is dependent upon the number of elements within a file.

The sorting of the files will have a different time complexity to that of the other operations within the first section of the algorithm. If std::sort were to be used for example, then the time complexity would be O(n log n), this complexity scales slower in time (faster on a graph) than that of the other operations within the first section of the algorithm. Because of this, and the fact that none of the operations described are nested within each other, the time complexity for the first part of the algorithm can be considered O(n log n). This is because the time complexity of an algorithm with separate parts, is equal to that of the part with the worst time complexity.

The second part of the algorithm consists of two main parts. Firstly, filling the files F’ and G’, and secondly, sorting files H, F and G as well as repeating the first process until file F is depleted. The claimed complexity for the first process of the second part of the algorithm is O(log n). Since this is repeated for log n times the claimed complexity for this stage of the algorithm is . This seems to make sense, however before the running of each iteration of the first process, the sorting of files H, F and G occurs. Normally a sorting process would have a time complexity of O(n log n), which scales slower than the O(log n) time complexity of the filling of the files F’ and G’.

Since the first part of the algorithm however is considered to have a time complexity of O(n log n), depending on the sorting process used of course, regardless of weather or not the second part of the algorithm has the quicker scaling time complexity of , the algorithm as a whole can still be considered to have a time complexity of O(n log n) thanks to its first section.

The final part of the algorithm is to sort the file G. This considering G contains the number of elements we started with, has a time complexity of O(n log n), which is equal to that of the rest of the algorithm which has been considered so far. As a result, the predicted time complexity of the algorithm is O(n log n).

**Task 4c**

Below are the timing results of the running of an implementation with std::containers and the std::sort function of the serial algorithm on differing input file sizes. 12 files were used in the results analysis.

|  |  |
| --- | --- |
| **Serial algorithm** | |
| **Number of names in input file** | **Algorithm completion time (micro seconds)** |
| 500 | 4185 |
| 1000 | 12907 |
| 2000 | 16074 |
| 5000 | 54651 |
| 20000 | 177857 |
| 50000 | 761398 |
| 200000 | 1695673 |
| 500000 | 5155818 |
| 500000 | 19916940 |
| 1000000 | 50392708 |
| 2000000 | 121596962 |
| 3000000 | 206146345 |

From the data above we can see that the results of the performance data match that of the predicted time complexity of O(n log n) of the algorithm in section 4a. This is due to the curvature of the graph matching that of the O(n log n) curvature in figure 1.

**References**

Figure 1 - <https://stackoverflow.com/questions/20512642/big-o-confusion-log2n-vs-log3n>