**1b**

Constructor

Time complexity: 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: O(n) — Linear

Reason: The recursive function ‘deepDelete’ goes through each of the items in the dictionaries linked list.

Move constructor

Time complexity: O(n) — Linear

Reason: The recursive function ‘deepCopy’ performs actions off each of the items in the dictionaries linked list.

Copy assignment operator

Time complexity: O(n) — Linear

Reason: The recursive function ‘deepCopy’ performs actions off each of the items in the dictionaries linked list.

Move assignment operator

Time complexity: O(n) — Linear

Reason: The recursive function ‘deepCopy’ performs actions off each of the items in the dictionaries linked list.

Insert

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.

Lookup

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

Remove

Time complexity: 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.

RemoveIf

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.

**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. Generally, locating this node has the time-complexity of O(n), as the list needs to be traversed, therefore depends on the number of nodes in the list. However since the data structure 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.

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.

**Task 2b.**

searching and inserting; the data structure which the algorithm will run off;

std::unordered\_map seems the best of the three. Since:

searching through std::list is O(n), inserting the list is O(1)

inserting and searching through std::map is O(log2 (n))

but std::unordered\_map has search and insertion of O(1)

Every element is…

Searched for…

Stored…

As a result of the number of elements 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 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).

Similar to 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 at O(n) and std::map at is 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.

Storing results

Since the container used to store results requires inserting elements at its front and back, of the three 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.

Container used to load file, reading and loading file, searching each element from container, results container, storing in results container, overall complexity.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Container used to store file contents and search from | Container used to store results | Time-complexity of reading and loading file | 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.

**Task 3b**

Performance data of searched based algorithm

**Task 4a**

Analysis

**Task 4c**

Performance data