**Task 2a.**

For each

Typical data structures (describe):

Suitable functions for searching and insertion (time complexity guarantees, & how data structures used enable the guarantees):

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 and inserting 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), 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, insertion or deletion takes time proportional to the logarithm of the number of nodes stored in the tree; O(log2 (n)). This is much better than the general operation linear timing required by std::lists;

std::unordered\_map

A hash table data structure is used in the implementation of std::unordered\_map. A hash table uses a hash function to calculate the index of an array of buckets or slots in which values are stored. The buckets or slots are stored unordered. Thanks to the calculation of the index searching and insertion operations have a constant time complexity.

**Task 2b.**

Random order of pairs of names

Small amount of ram

Put in order

Data structure for inserting random list & then searching from (insertion & searching time important)

Data structure for holding resulting sequence (insertion time important)

std::list & std::map

std::list & std::unordered\_map

std::map & std::unordered\_map

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)

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.

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.

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.