Report

1B:

Dictionary() = default;

The best and worst time complexity for the default constructor will be O(1) .

Dictionary(const Dictionary&); (Copy)

The best and worst time complexity for copy constructor will be O(n) since all it will have another recursive worker function doing all the coping of the nodes.

Dictionary& operator=(const Dictionary&); (copy)

Copy assignment will also have O(n) complexity as it also has a worker function which also has to copy the nodes 1 by 1 as well as delete them from the previous tree.

Dictionary(Dictionary&& original) (move)

The best and worst time complexity for move constructor will be O(N) because it doesn't matter how big the tree or how many entries there are, as it will just assign the pointer of the root from the given containers to the new container's root and assign given containers point to null pointer.

Dictionary& operator=(Dictionary&&); (move)

The best and worst time complexity for move assignment will be O(N) because similarly to the move constructor, it just changes the pointers, but also deletes the other dictionary.

bool insert(Key, Item);

The worst time complexity for insert function in the O(n) because it will take longer to insert into the container as it grows because it will need to compare all the keys and values to the ones given as arguments to see if they key already exists, and if it does, then it will replace the value, if it doesn't, then it will add the new node (key and value). Therefore, it will take longer as the container size increases since it needs to check all the nodes. The best time complexity would be O(1) if there is no nodes in the container or the key matches with the first key on the list.

Item* lookup(Key);

The worst time complexity this function has is O(n), as it has to make many comparisons when trying to find and match the keys, so if the container was very big, it would take a long time. The best time complexity would be O(1) when the key is the root.

bool remove(Key);

The worst time complexity for this function would be O(n) because it searches the container to find the key and then delete that node it the keys match. The best time complexity would be O(1) if then key to remove is the root.

void removeIf(F);

The worst and best time complexity for this function would be O(n) because it has a recursive worker function which removes certain nodes.

2.

std::list

The data structure to implement a std::list is by using doubly linked list because it always keeps track of the head and tail pointer. The most suitable way for searching in a standard list data structure is to use **std::find** by making use of begin and end operator. The first 2 values are iterators and the 3rd one is the value you wish to find. It will iterate through all the elements between the 2 given iterators and compare it with the given value. If it matches with the value then it will return that iterator, otherwise it will return the iterator pointing at the end of the list. This means the searching algorithm will have O(n) time complexity for best and worst case. The most suitable way for insertion for the list data structure is by using the **std::push_back** which is used to insert the value to the end of the list container. Alternatively, you can use **std::push_front** to insert values to the front of the list container. This means for insertion, the time complexity would be O(1) as it is consistent for worst and best case. However, if you wish to insert the values to a specific position in the container, you can use the **list::insert** function. It takes 3 values as arguments, the first being the list variable, second being value you wish to insert, the third being the position you wish to insert it to.

std::map

The data structure used to implement std::map is by using binary search tree with self balancing features (red black tree) which means both insertion and search have O(log(n)) time complexity. The most suitable way for searching in a standard map data structure is by using **map::find** function which will take the value you want as an argument as search through the container (using map::iterator) comparing the keys, if a match is found, the comparison object will return false. This means Red Black Tree (std::map) will have O(log n) complexity for worst and best case as it will take longer as the size of the container increases. The most suitable way for inserting in a map data structure is by using the **map::insert** function which extends the container by inserting new elements. Map containers keep all their elements sorted and equally balanced as it has self-balancing features. You insert a key and item in the map container as pairs using **std::pair.** For inserting, the time complexity is also O(log n) for worst and best case.

std::unordered_map

The data structure used to implement std::unordered_map is hash table which has O(1) time complexity for best case and O(n) complexity for worst case. Similarly to std::map, you can also use the **map::find** function to search through the container using map::iterator. So the time complexity would also be O(log(n)). The most suitable way for inserting in a unordered map would be my using the **unordered_map::insert** function which works same way map::insert function works. The time complexity for hash table is O(log n).

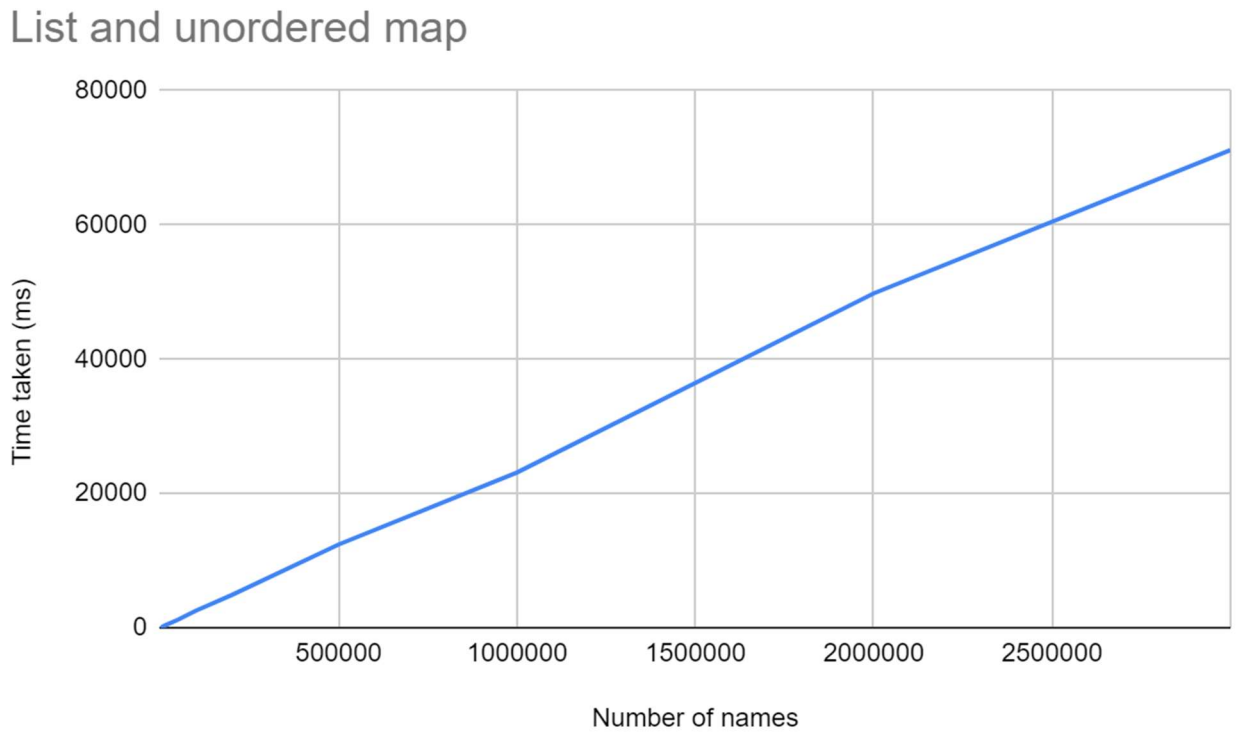
Best combination:

The best combination would be to use std::list to store the results and use the std::unordered_map for the container for the basket of names. The reason to use std::list for storing the result data is because it is implemented using doubly linked list so its easier to add the data in the front of the container or the back, also it internally keeps the ordering of the data unlike the other 2 containers. The reason to use std::unordered_map for the container to handle the basket of names is because they are implemented using hash table which is made up of a combination of key value and map value so it will be easier to store the

name, and their neighbours; compared to std::map, unordered map allow fast access to individual elements whereas std::map doesn't. The time complexity for the algorithm as a whole would be O(n).

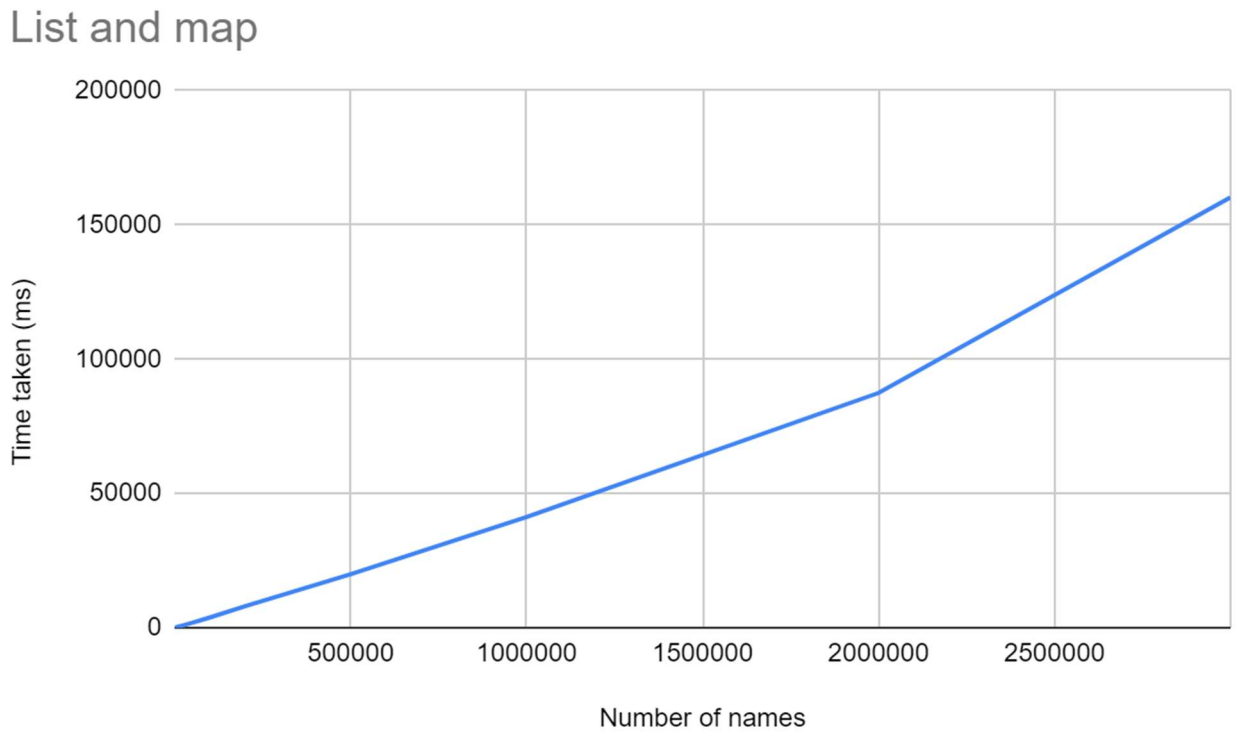
3B:

Number of names	Time taken (ms)
20	1
50	3
100	6
200	10
500	14
1000	23
2000	42
5000	109
10000	241
20000	478
50000	1203
100000	2527
200000	4863
500000	12392
1000000	23116
2000000	49802
3000000	71135

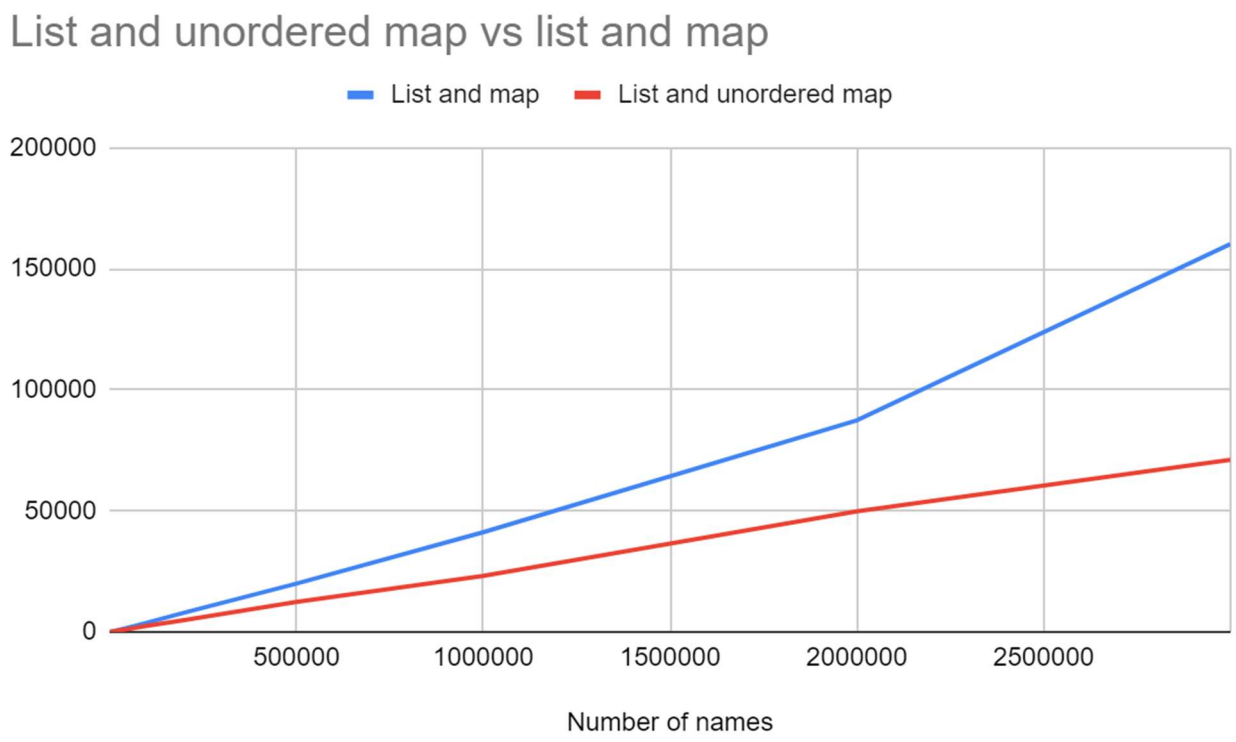


The graph represents the performance data of the list and unordered map combination and it has a linear line so it is O(n) time complexity as predicted.

Number of names	Time taken (ms)
20	1
50	3
100	6
200	9
500	17
1000	33
2000	75
5000	171
10000	360
20000	704
50000	1827
100000	3812
200000	7952
500000	19924
1000000	41194
2000000	87408
3000000	160213



This graph represents the performance data results of list and map container combination and as you can see the it has a slight curve which resembles the O(x log(n)) time complexity.



This graph represents the performance data results of both combinations and as you can see the unordered map and list has better results compared to map and list combination, as predicted.