Data Structures for Data Storing

Yakub Yakubov, Tarlan Sultanov, Daler Zakirov

April 2023

1. Introduction (Task 1 – Part A)

In the paper we approach several problems related to Balanced Binary Search Trees, trie, hash maps and some graphs algorithms. BBST, Trie, several versions of Hash Maps and DFS, BFS algorithms were implemented, and the running times were measured.

1.1. Naïve

For the naïve approach linked list is used giving the time complexity of insertion of O(1), search complexity to be $O(n)$.

1.2. BBST (Balanced Binary Search Tree)

It is a self-balancing binary search tree in which the heights of the left and right subtrees of every node differ by at most 1. The balancing property ensures efficient search, insertion, and deletion operations. The BBST maintains its balance by performing rotations when necessary. A rotation is a local transformation that adjusts the structure of the tree while preserving the ordering property of the binary search tree. The time complexity of search, insertion, and deletion is $O(\log n)$, where n in the number of nodes. BBST is powerful data structure which combines the benefits of a binary search tree with auto balancing, making it useful for search and modification

1.3. Trie

It is also known as a prefix tree or digital tree, is a tree-based data structure used to efficiently store and retrieve strings or sequences of characters. It is primarily used for efficient string matching and prefix-based operations. The key feature of a Trie is that it allows for fast insertion, deletion, and retrieval of strings. It achieves this by exploiting the common prefixes among strings. By storing strings as prefixes, Trie can effectively reduce the search space and optimize string operations. Time complexity of insertion is $O(n)$, deletion $O(n)$, search is $O(n)$.

1.4. HashMap

Hash table or associative array is a data structure that allows efficient insertion, deletion, and retrieval of key-value pairs. It provides a way to map keys to their corresponding values using a hash function. hash table or associative array, is a data structure that allows efficient insertion, deletion, and retrieval of key-value pairs. It provides a way to map keys to their corresponding values using a hash function. A hash function is used to convert a key into an index in the array. It takes the key as input and computes a hash code, which is an integer value. The hash code is then mapped to a valid index within the array using a modulus operation. Efficient hash function is needed to avoid collisions. Collisions can occur when multiple keys generate the same hash code or map to the same index. In average the time complexity of insertion, search and delete is $O(1)$, in the worst case with many collisions $O(n)$.

2. Methodology

Problem statement:

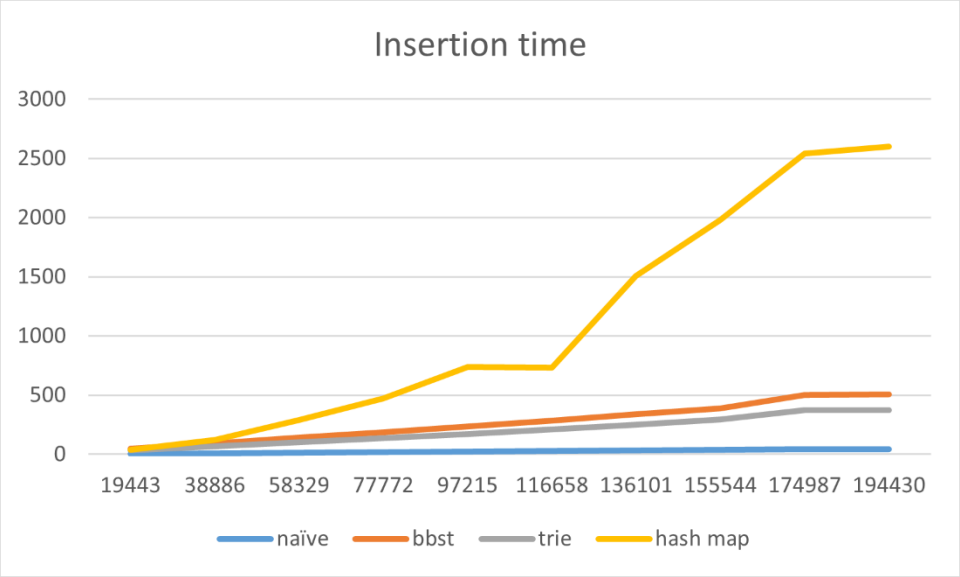
The goal of this project is to compare the performance of the operations on bbst, trie, hash map, as well as modify them and implement some solution to the problems using graphs algorithms such as dfs and bfs.

Experiment execution:

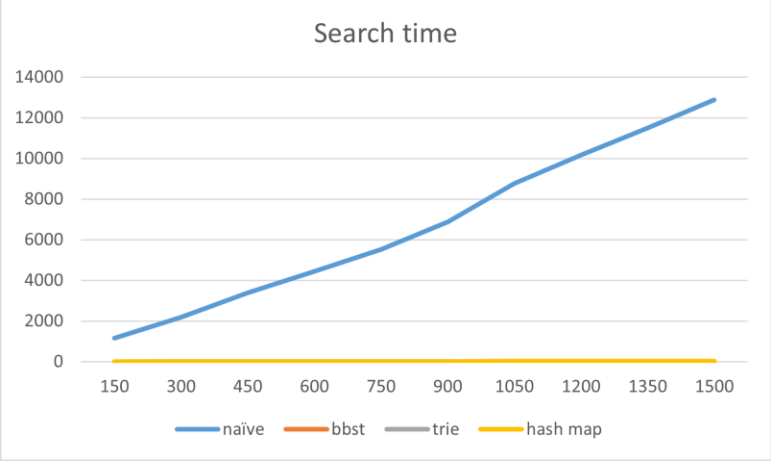
The experiments will be conducted in a Visual Studio Code. The performance metrics will be measured using the C++ time profiler libraries.

3. Results

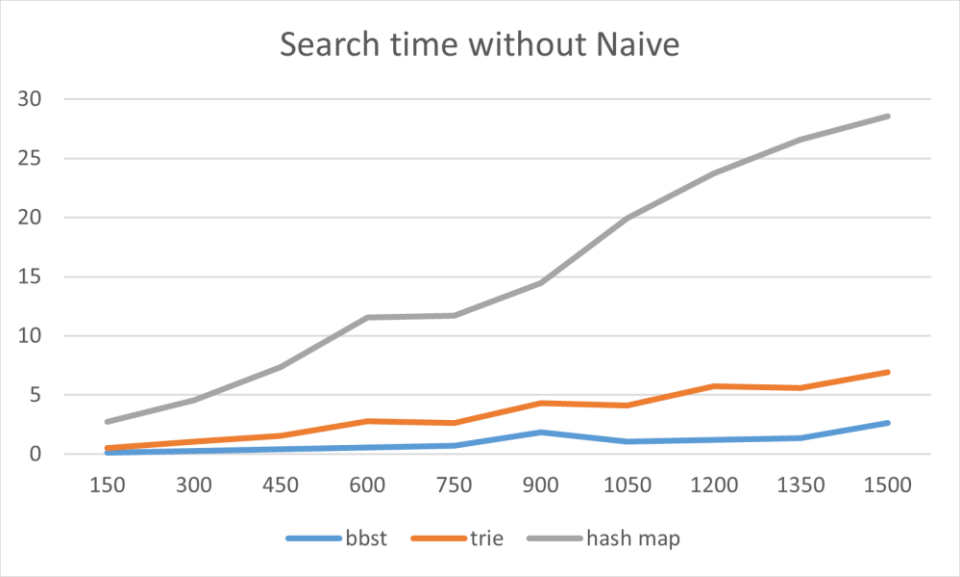
As it clearly seen from the graphs, that naïve algorithm performs the best for the insertion as the data structured that was used is a linked list which has a constant time insertion.



If we look at this graph, the gradual increase of time for trie and bbst is visible. A hash map function performed poorly, and it is assumed that it had many collisions so the time complexity became linear being several times slower for the big pattern.



As for the searching, it took much longer to search in linked list than in the rest of the containers. The naive algorithm in the worst time scenario will go along all the element and compare them having linear time complexity.



The rest 3 container showing really good time results searching fast for the words, with bbst being the fastest and hash map slowest. From the measurements we can conclude that hash function could be designed better.

4. The Triwizard Tournament (Task 1 – Part B)

The task was to implement the algorithm which will mark the first person to escape the labyrinth. The algorithm utilizes a graph traversal technique to efficiently navigate through the labyrinth and determine the optimal path for each player. The algorithm considers the players' speeds and calculates the minimum time required to reach the exit. The proposed algorithm is based on a graph traversal technique known as breadth-first search (BFS). The labyrinth is represented as a graph, where each node corresponds to a position in the game and each edge represents a valid move between positions. The algorithm starts from the designated starting position and explores neighboring positions in a breadth-first manner, keeping track of visited nodes, using queue for efficient traversal, updating previous node pointer to construct the shortest way. The winner is known by the time comparisons. The code is efficient in handling change of the speeds and finding optimal path.

5. Aunt’s Namesday (Task 2)

The code of this task implements a table assignment algorithm for a party where guests are represented as nodes in a graph. The goal was to assign guests to tables such that no two guests who do not like each other are seated at the same table. The algorithm uses a depth-first search (DFS) approach to assign tables to the guests. The unordered_map container from the C++ Standard Template Library (STL) with the key to be a string to represent a graph structure. The special function was created which adds both vertices to each other's neighbor list by pushing the corresponding string into the vector for each vertex. And special container(set) is also used for keeping track of “visited” guests. If the guest is not yet met, DFS is run for this guest, then assign a table to a guest, and run DFS on the neighbors, if conflict is detected we detect that we assigned table is not valid. Otherwise, run recursively assigning table function on visited neighbors.

6. Full House (Task 2)

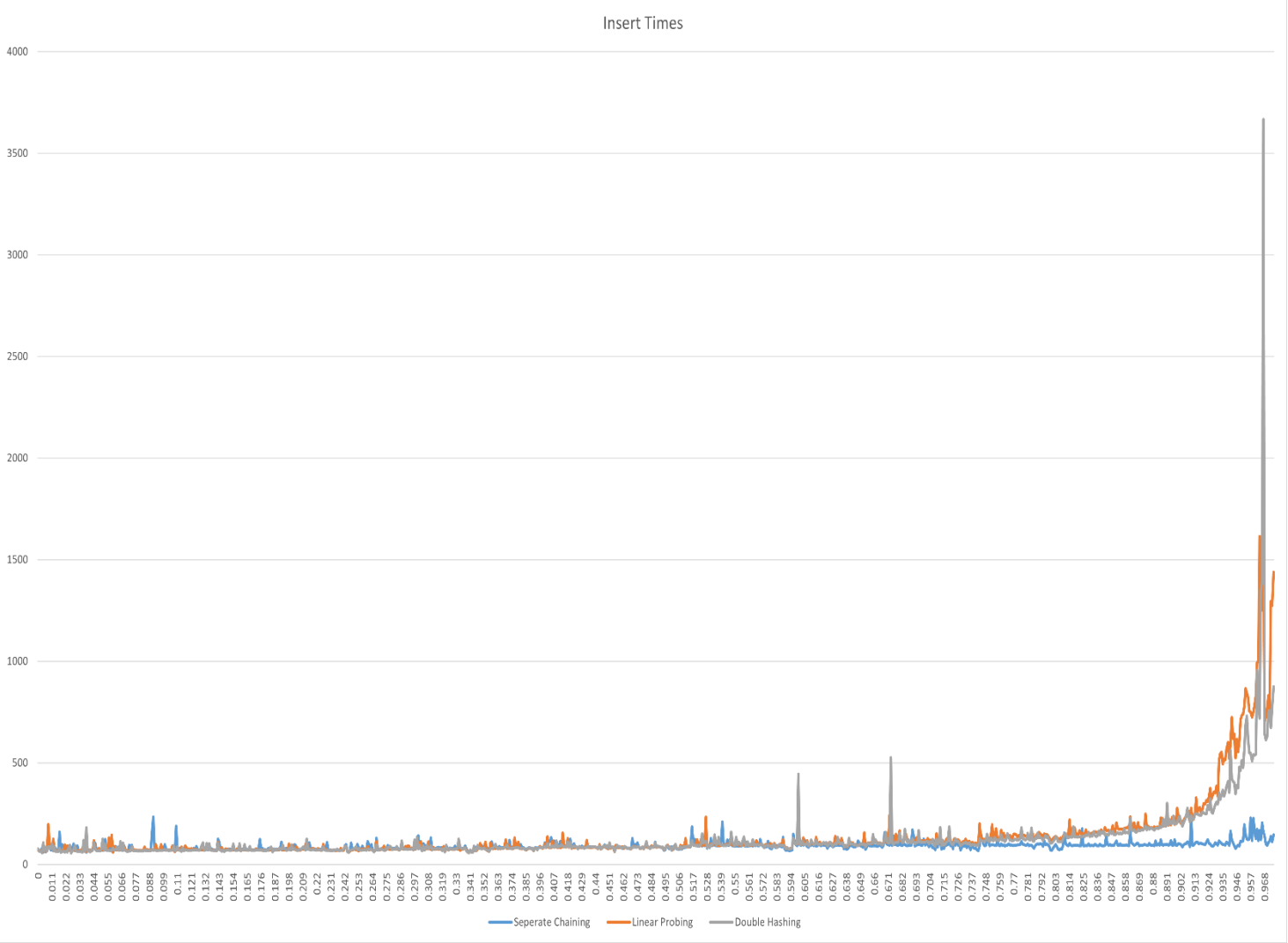
Hash map is a data structure used for efficient storage and usage of data stored as key value pairs. Well implemented hash table provide constant time performance for insertion, deletion, and search operations in average case. To handle collisions, several techniques exist, open addressing, linear probing, and double hashing.

Open addressing is the way of dealing with collisions where all key-value pairs are stored directly in the hash table itself, on collision instead of putting the value into the separate container, open addressing searches for the next free slot in its array. The stages are following: first hash code is computed, if the index is taken, the algo uses open addressing looking for next free place, various techniques can be used to locate the value and search for the spot.

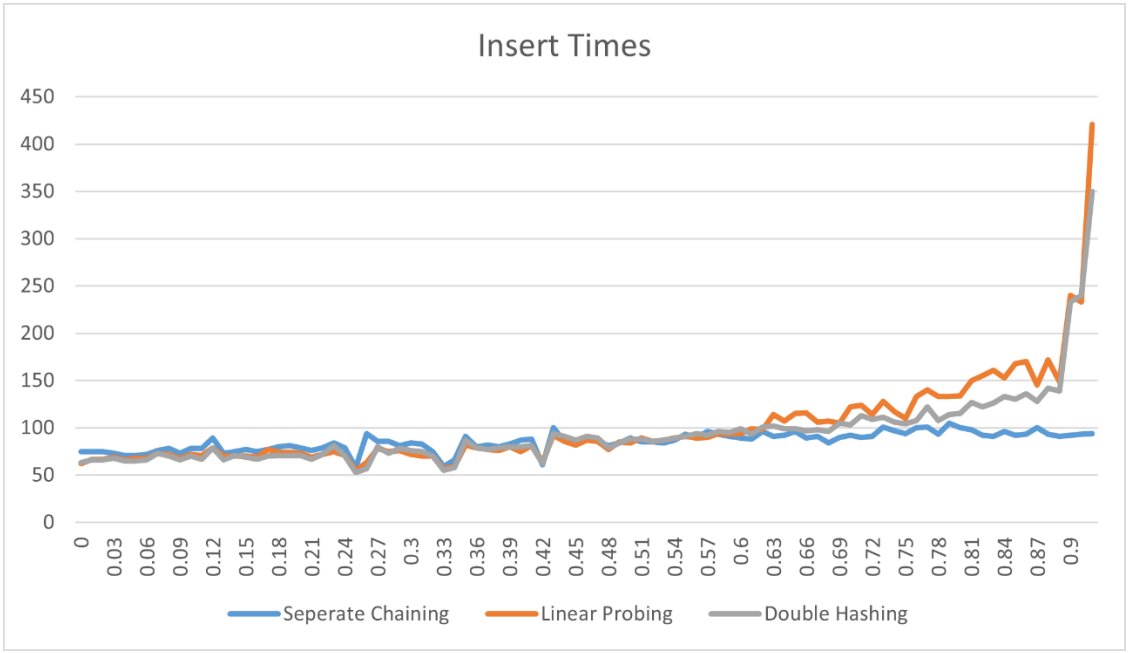
Double hashing is a method that addresses collision by probing for alternative hash codes when a collision occurs, it uses two hash functions to calculate the next available index in the hash table. The primary hash function generates the initial hash code, if collision happened than second hash function is taken to calculate the offset value, which used to determine the next available index to probe. If that index is taken already, second hash function reapplied with incremented offset to find empty slot. If slot is found value is inserted to the slot.

Liner probing deals with collisions by sequentially searching for the next available slot in the hash table. The first hash code is calculated using a hash function, if the received index is taken, linear probing is used to find the next free index. Starting from the initial index, the algo one by one checks the next indices in the array until empty one is found, inserting the value inside

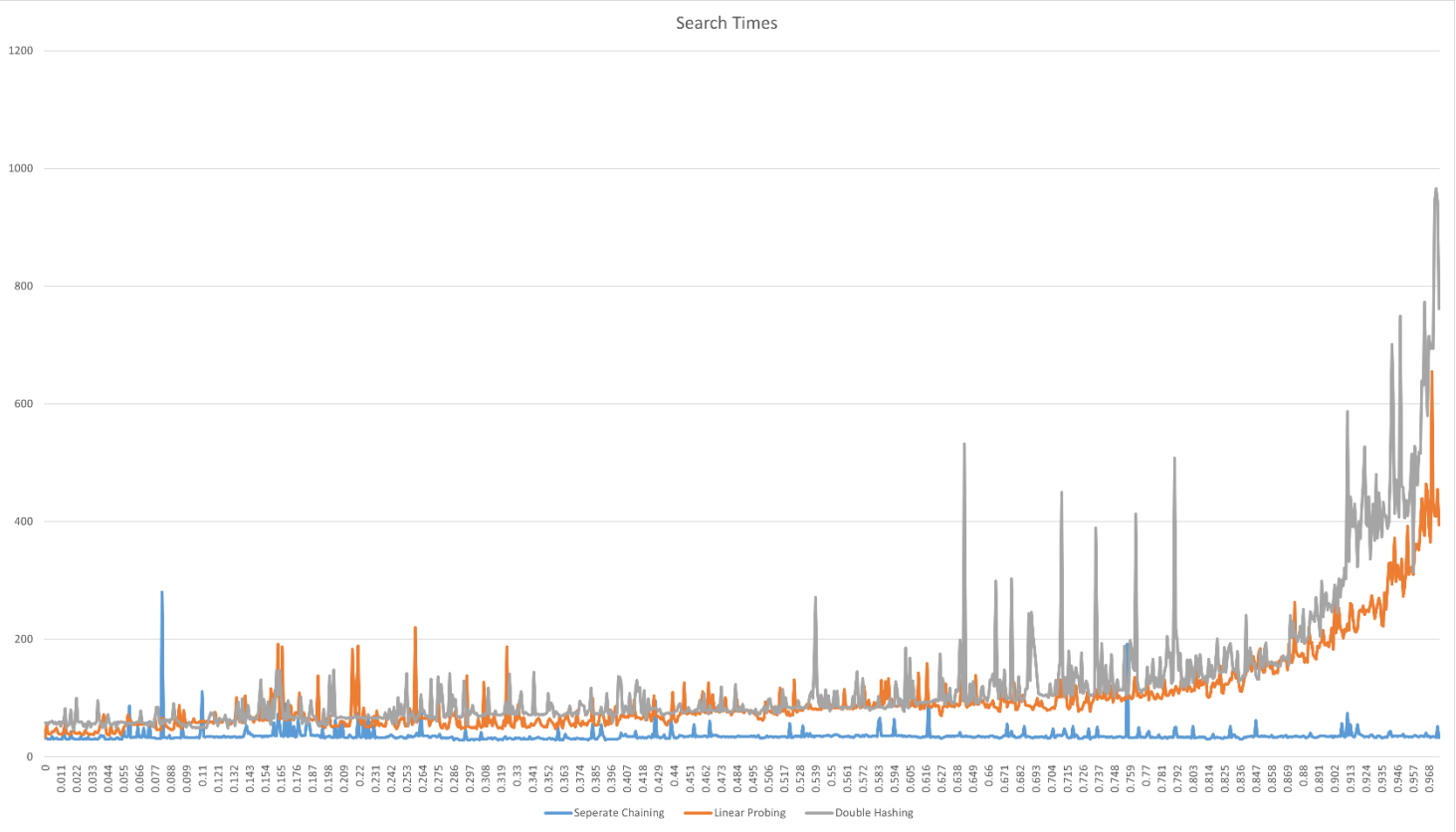
Comparing the insertion, the hash map with the separate chaining performs worse than two others. The reason for that is that traversal of the internal linked list and poor cache performance.

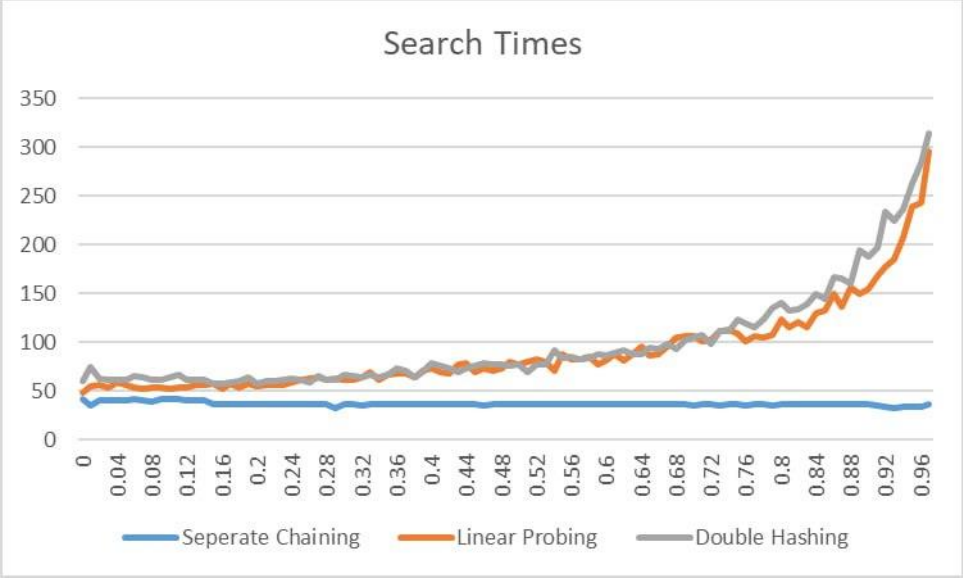


As it visible from the graphs all the methods to resolve the collisions are effective up to some load factor, where linear probing and double hashing starting to show linear time complexity, with double hashing performing worse for the big load factor.



As for the searching, the functions showing constant tendency up untill the load factor of 75%, after which the linear rate is demonstrated for double hashing and linear probing. With the separate chaining the situation is different as this technique showing the O(1) time on average with random spikes along the picture, independently on load factor.





Conclusion, two graph algorithms (DFS and BFS) were used for the specific tasks, and several complex modifications of the hash table were introduced.