Social Media Manager

Experiment Report

Team Members:

Katie Hammer

CSC 316 - 001

Name of Person Submitting the Report: Katie Hammer

4/11/24

Implementation provided by:

Ryder Fuscellaro

Methodology

Measuring Runtimes

In this experiement, the runtimes were measured by incorporating specific methods within the main method of the UI package. Initially, the experiment was set up by defining the person and connections files; an example setup was demonstrated for an input size of 2^18 and
using a skip list as the data structure. Therefore, the program executed the getConnectionsByPerson method, utilizing the specified input files and data structure. The program then captured the runtime by starting and stopping a timer around the method execution, subsequently recording the duration and displaying it on the console. The entire process was encapsulated within a try-catch block, ensuring that the program would not execute or record time if the files failed to open. The runtime values were measured in milliseconds using the currentTimeMillis method from the Java library.

```
    public static void main(String[] args) {

    try {
        // Specific file names
        String personFile = "input/experimentFiles/people_18.csv";
        String connectionsFile = "input/experimentFiles/connections_18.csv";
        long start = System.currentTimeMillis();
        ReportManager manager = new ReportManager(personFile, connectionsFile, DataStructure.SKIPLIST);
        manager.getConnectionsByPerson();
        long stop = System.currentTimeMillis();
        long duration = stop - start;
        System.out.print("Elapsed Time: ");
        System.out.println(duration);
    } catch (Exception e) {
        System.out.print("One (or more) of your files caused an error, re-run and try again.");
        System.exit(0);
    }

}
```

Results

Hardware

We conducted the experiment using the following hardware:

- Operating system version: Windows 10, 64-bit operating system, x64-based processor
- Amount of RAM: 16.0 GB (15.7 GB usable)
- Processor Type & Speed: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz 3.00 GHz

Table of Actual Runtimes

Input Size (2^x)	Unordered Linked Map (ms)	Search Table Map (ms)	Skip List Map (ms)	Splay Tree Map (ms)	Red-Black Tree Map (ms)	Linear Probing Hash Map (ms)
6	192	172	176	144	143	119
8	260	239	228	186	191	190
10	490	351	367	316	301	281
12	2414	649	576	501	505	465
14	75981	1621	1294	1124	969	904
16	1636787	2469	3489	2489	2361	1951
18	>30 min	7700	13319	8377	7430	5319
20	>30 min	35534	54537	36809	30446	19410
22	>30 min	235275	340896	246460	184515	109589

Chart 1: Log-Log Chart of Actual Runtimes

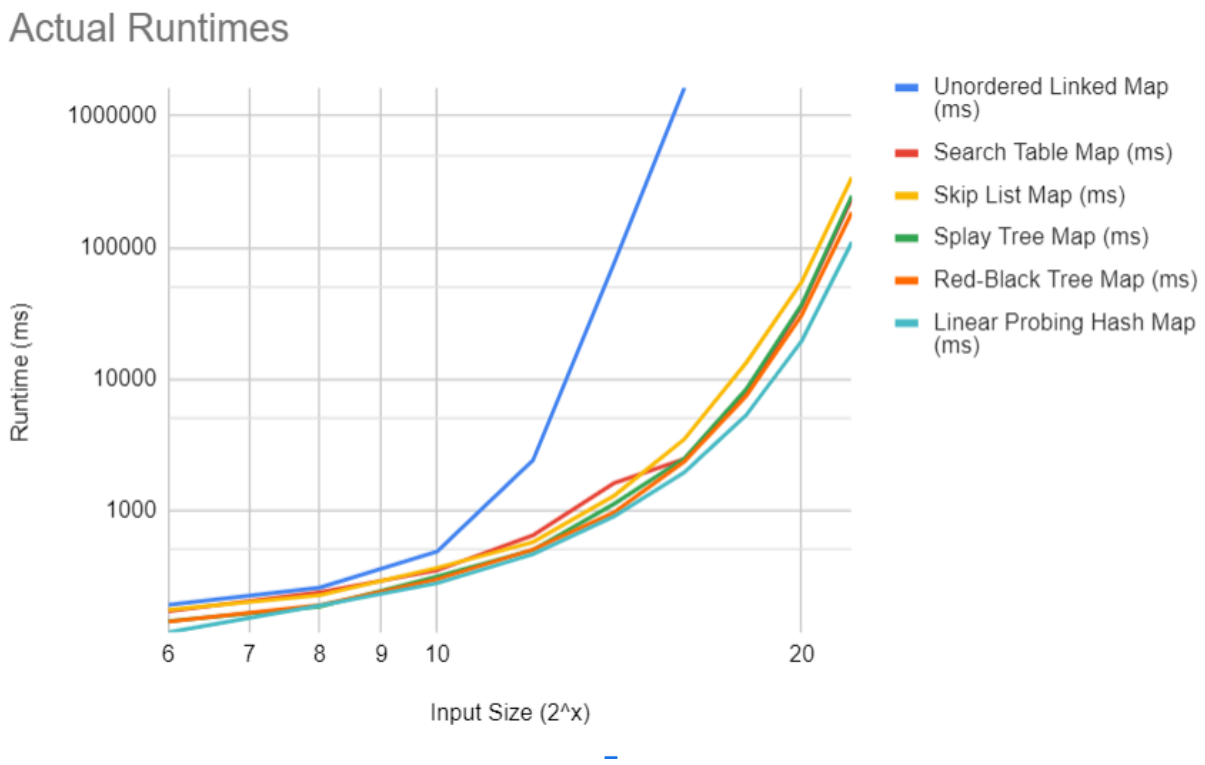


Chart 2: Log-Log Chart of Actual Runtimes Unordered Linked List-based Map

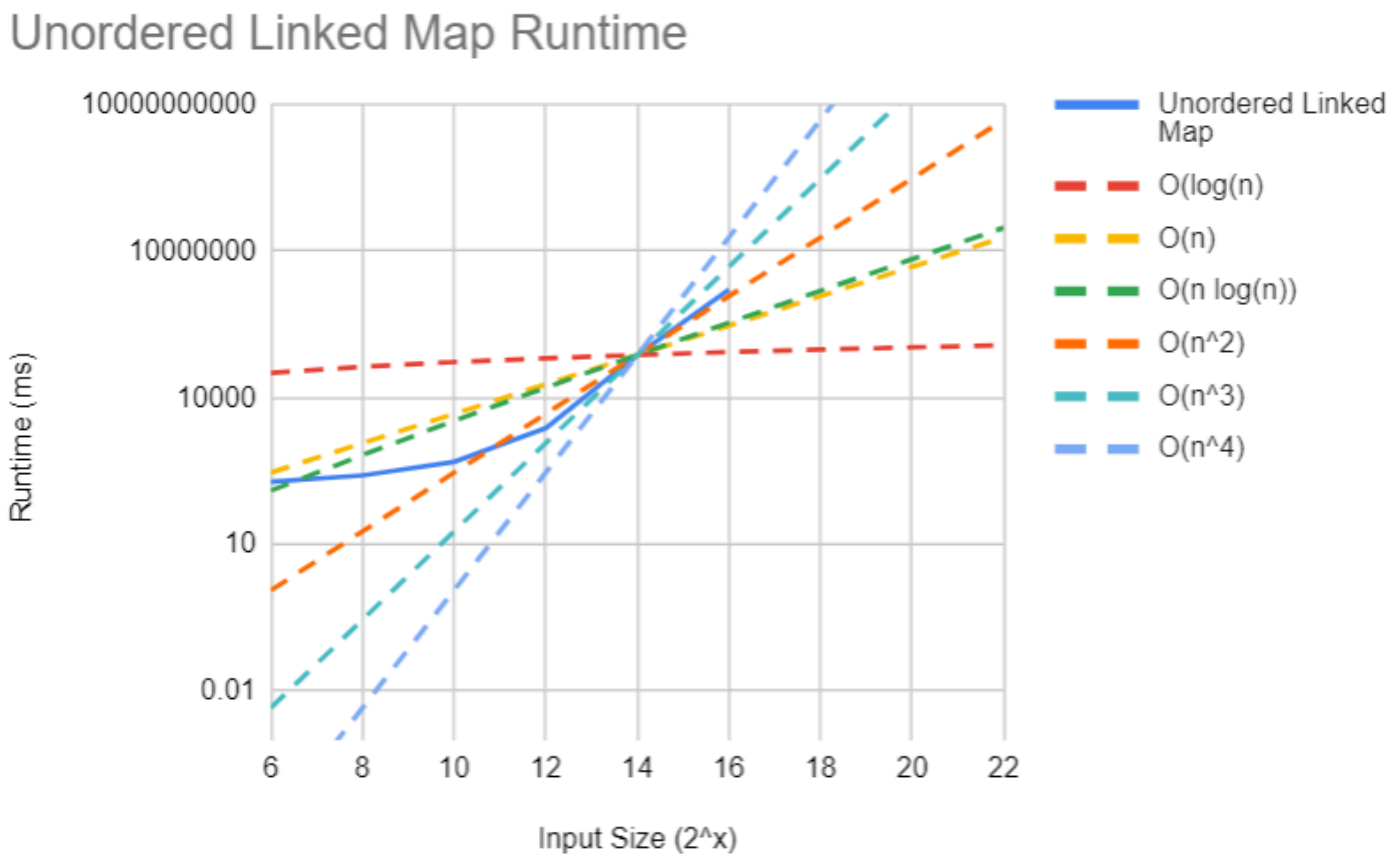


Chart 3: Log-Log Chart of Actual Runtimes for Search Table Map

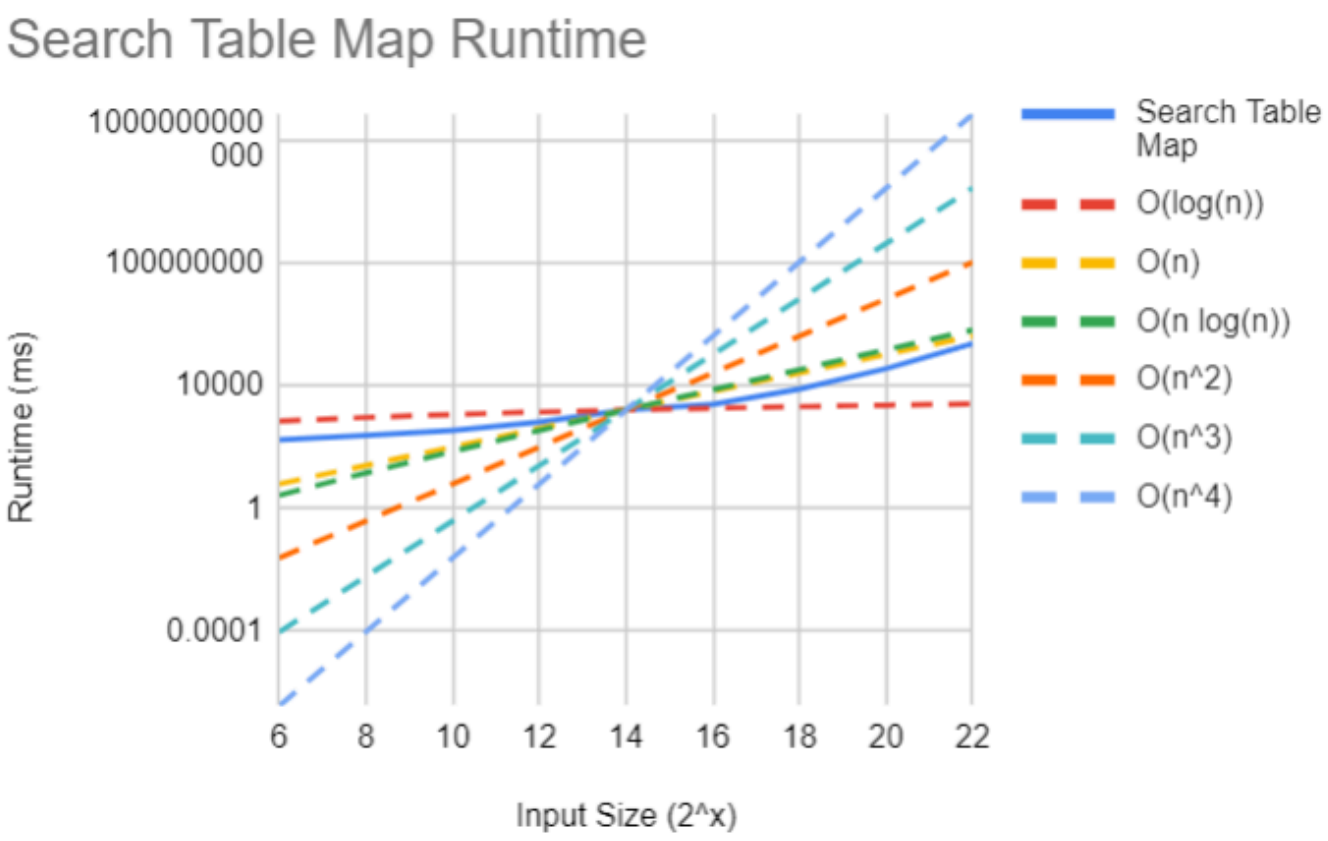


Chart 4: Log-Log Chart of Actual Runtimes for Skip List Map

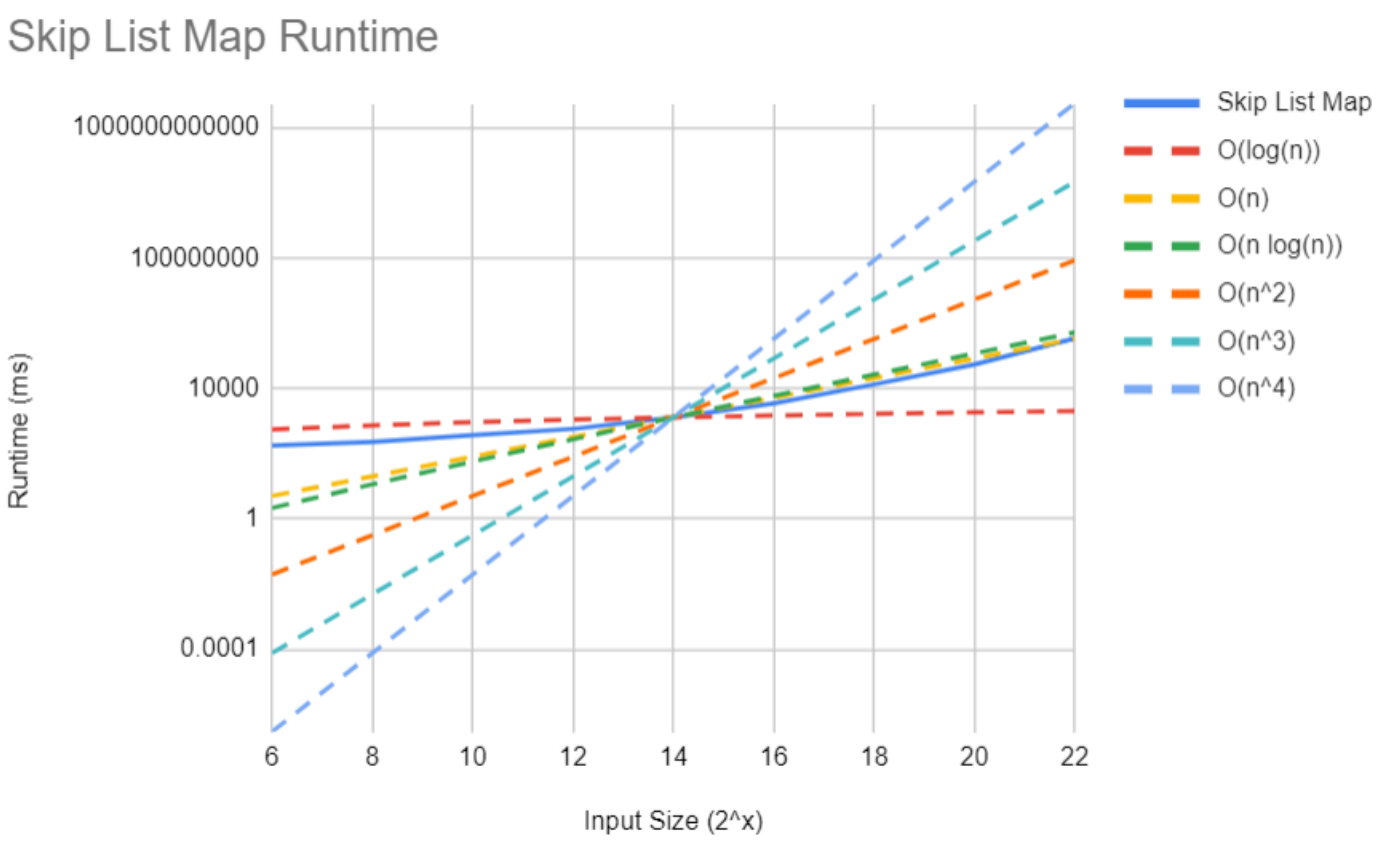


Chart 5: Log-Log Chart of Actual Runtimes for Splay Tree Map

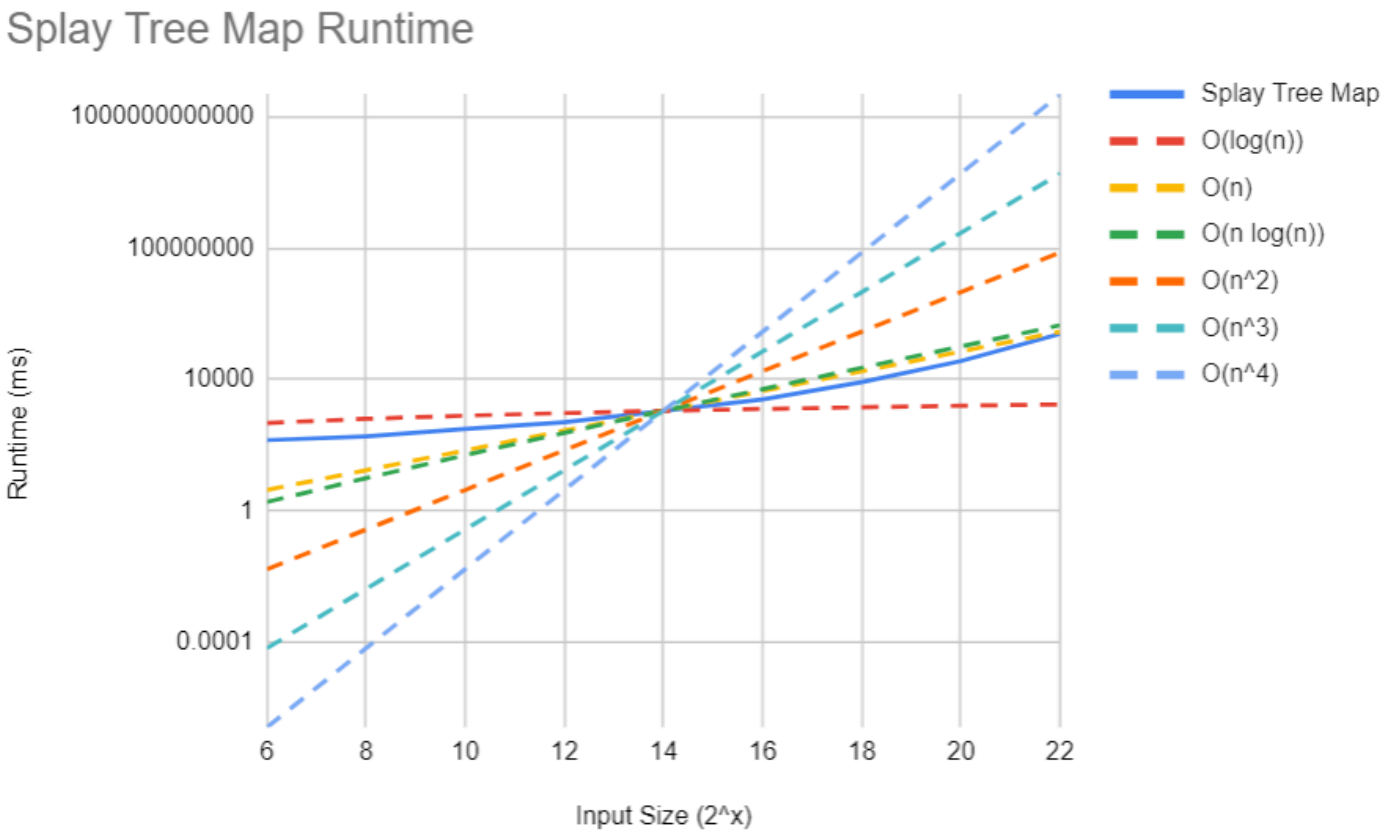


Chart 6: Log-Log Chart of Actual Runtimes for Red-Black Tree Map

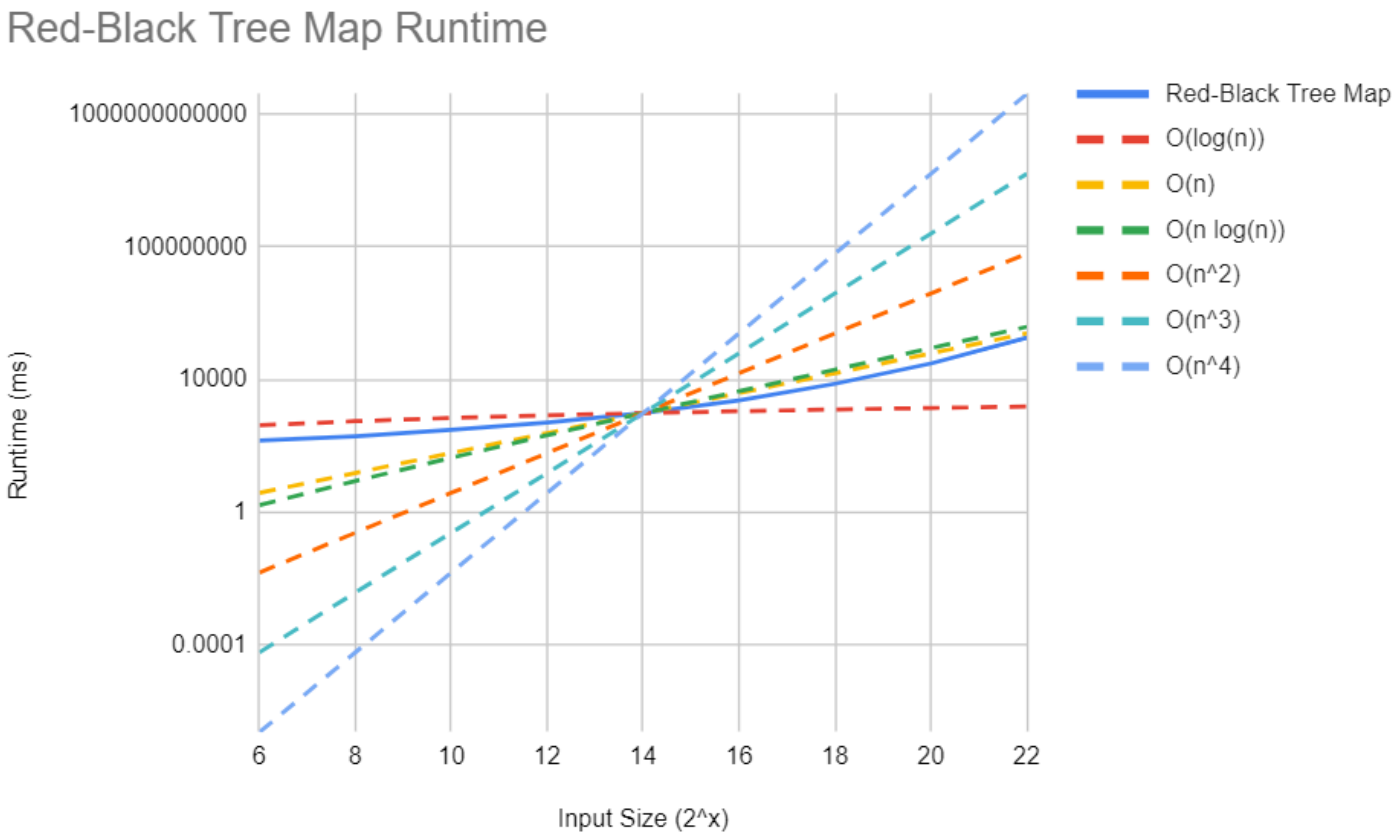
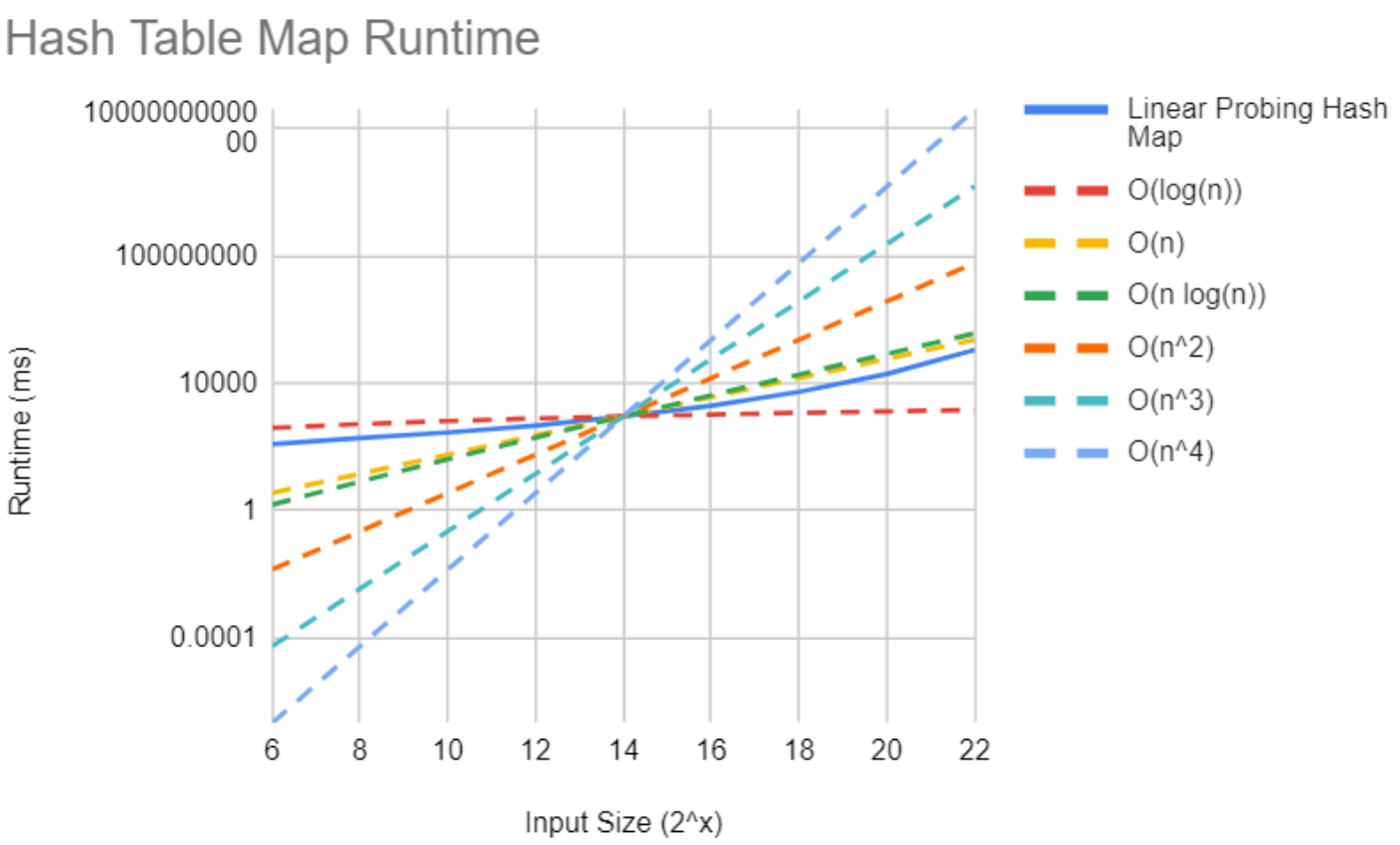


Chart 7: Log-Log Chart of Actual Runtimes for Linear Probing Hash Map



Discussion

Reflection on Theoretical Analysis

In this Java project, both my original proposal and the adopted code base utilized an Array-Based list with Merge Sort, targeting a theoretical runtime of O(n log n). However, significant differences in the implementation of the getConnectionsByPerson method influenced the actual outcomes. My approach proposed creating a new map, populating an ArrayList per person, and performing checks and sorts during each connection addition. This method, while thorough, potentially increased runtime due to the repetitive checks for existing connections. In contrast, the adopted code base streamlined this process. It involved populating an ArrayList directly from usernames and sorting connections as they were added. This method reduced overhead by eliminating repetitive checks and integrated sorting with data insertion, optimizing both time and efficiency. The practical application of integrating sorting with insertion in the adopted approach improved the overall efficiency.

Reflection on Data Structure Selection

In this Java project, we employed a linear probing hash map to efficiently manage usernames and their associated connections. This data structure was chosen for its ability to facilitate quick access to user profiles by using usernames as keys and their lists of connections as values. Our tests confirmed that the linear probing method was the fastest among the options we considered, supporting our initial hypothesis regarding its efficiency. Linear probing hash maps excel in scenarios involving key-value pairs, as demonstrated by our use case where quick and direct access to any user's data was crucial. The hash map efficiently computes the hash of a username to locate or insert the corresponding user profile, making it an ideal choice for handling dynamic and sizable datasets where performance is a priority.

Improving Efficiency

To further enhance the efficiency of the software, targeted improvements can be made to reduce actual runtimes. Addressing duplicate values in our connections dataset can significantly streamline operations. Currently, when a connection is established between two individuals, it is redundantly added to both parties' lists. By modifying the data structure to store each connection only once—perhaps by employing a set within a hash map that ensures unique entries—we can reduce memory usage and decrease the runtime for operations that traverse these lists. Additionally, implementing a pre-sorting step before any major methods are run can lead to more efficient data handling. This can be achieved by sorting the input file as it is loaded into the system. Maintaining this sorted order through subsequent operations would reduce the need for repeated sorting, thus optimizing both search efficiency and data retrieval. Pre-sorting can be efficiently executed using a robust algorithm like MergeSort, which can handle large volumes of data with greater speed, especially beneficial if connections are frequently accessed in a sorted order.

Lessons Learned

Through hands-on experimentation in my Java project, I gained profound insights into the complexities of selecting optimal data structures and algorithms. This process highlighted the importance of considering a multitude of factors, including the size and type of input, the presence of duplicate values, and the influence of external hardware—all of which significantly impact a program's runtime. Understanding these elements is crucial for making informed decisions to enhance program efficiency. Initially, I overlooked critical aspects such as input size and external hardware considerations, which impeded the progress of my project. This oversight necessitated the integration of another student's codebase to achieve completion. Throughout this project, I have significantly deepened my understanding of various sorting algorithms and developed the analytical skills necessary to determine the most appropriate algorithm for different scenarios. This experience has been invaluable in teaching me the nuances of algorithmic efficiency and decision-making in software development.