Surveillance Manager

Experiment Report

Implementation provided by:

Brandon Wroblewski

Methodology

To measure the runtime, I developed a user interface that will run the operation “getPeopleCoveredByWarrent” using the given testing files on a specified map type. The timer begins prior to creating an instance of ReportManager, and ends once the operation “getPeopleCoveredByWarrent” finishes, where it will then print out the time taken. getPeopleCoveredByWarrent will always be called with a hop count of 100,000 and a phone number of “123-456-7890123”. The timer uses ms to measure time taken, and uses a threshold of 30 minutes where if it fails to finish a file in that timezone then I put “>30 min” for the data, and if a error occurred where it failed, I would write, “error” in the test data, and in these situations the data would be omitted from the charts. Since my code has no real way of knowing when the 30 minute threshold is met, I would set a 30 minute timer on my phone and click start immediately after running the program. For the testing, I am using a stack size of 10 megabytes, and a heap size of 800 megabytes, the same stack and heap size given were the same for all test performed. One limitation of my testing code is that I can only test one structure at a time for a given set of files, this is because I have no way to have the program realize that it reached the 30 minute threshold and force stop a test when the issue occurs. This is possible using imports, but in CSC316 and prior classes, no import of the nature were taught, as such I opted out of using any and instead decided to test each structure manually and using my phone timer to know when to force end a test.

Code for testing the files:

```
        /**
         * Main method that runs the program
         *
         * @param args arguments from command line, should be none
         * @throws FileNotFoundException
         */
  public static void main(String[] args) throws FileNotFoundException {
        // represents the current data structure
        DataStructure currentStructure = DataStructure.UNORDEREDLINKEDMAP;
        // represents the input files
        String peopleFile = "input/people_8.csv";
        String callFile = "input/calls_8.csv";
        // starts timer then makes a report instance
       long beginTimer = System.currentTimeMillis();
       ReportManager surveillanceManager = new ReportManager(peopleFile, callFile, DataStructure.UNORDEREDLINKEDMAP);
       // runs getPeopleByHop and ends the timer when it finishes
       surveillanceManager.getPeopleCoveredByWarrant(100000, "123-456-7890123");
       long endTimer = System.currentTimeMillis();
       // calculates duration and prints result
       long duration = endTimer - beginTimer;
       System.out.println("Data Structure: " + currentStructure);
       System.out.println("Time taken: " + duration + " milliseconds");
  }
```

Example of output for it:

```
Data Structure: UNORDEREDLINKEDMAP
Time taken: 352 milliseconds
```

Results

Hardware

We conducted the experiment using the following hardware:

- Operating system version: Pop!_OS 22.04 LTS
- Amount of RAM: Total: 62Gi, Available: 48Gi, Free:34Gi
- Processor Type & Speed: 13th Gen Intel(R) Core(TM) i9-13900HX

  CPU max MHz: 5400.0000 MHz (5.4 GHz)

  CPU min MHz: 800.0000 MHz (0.8 GHz)

  Adjusted Stack Size Used: 10 megabytes

  Adjusted Heap Size Used: 800 megabytes

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	215	210	222	211	220	221
8	352	353	332	349	297	288
10	781	594	633	614	574	568
12	3277	1831	1984	1964	1502	1396
14	36607	8623	6273	5790	6226	5711
16	1252010	59152	25513	24925	23513	21332
18	> 30 min	1411995	106322	100241	94963	86530
20	> 30 min	> 30 min	error	error	error	error

Chart 1: Log-Log Chart 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	215	210	222	211	220	221
8	352	353	332	349	297	288
10	781	594	633	614	574	568
12	3277	1831	1984	1964	1502	1396
14	36607	8623	6273	5790	6226	5711
16	1252010	59152	25513	24925	23513	21332

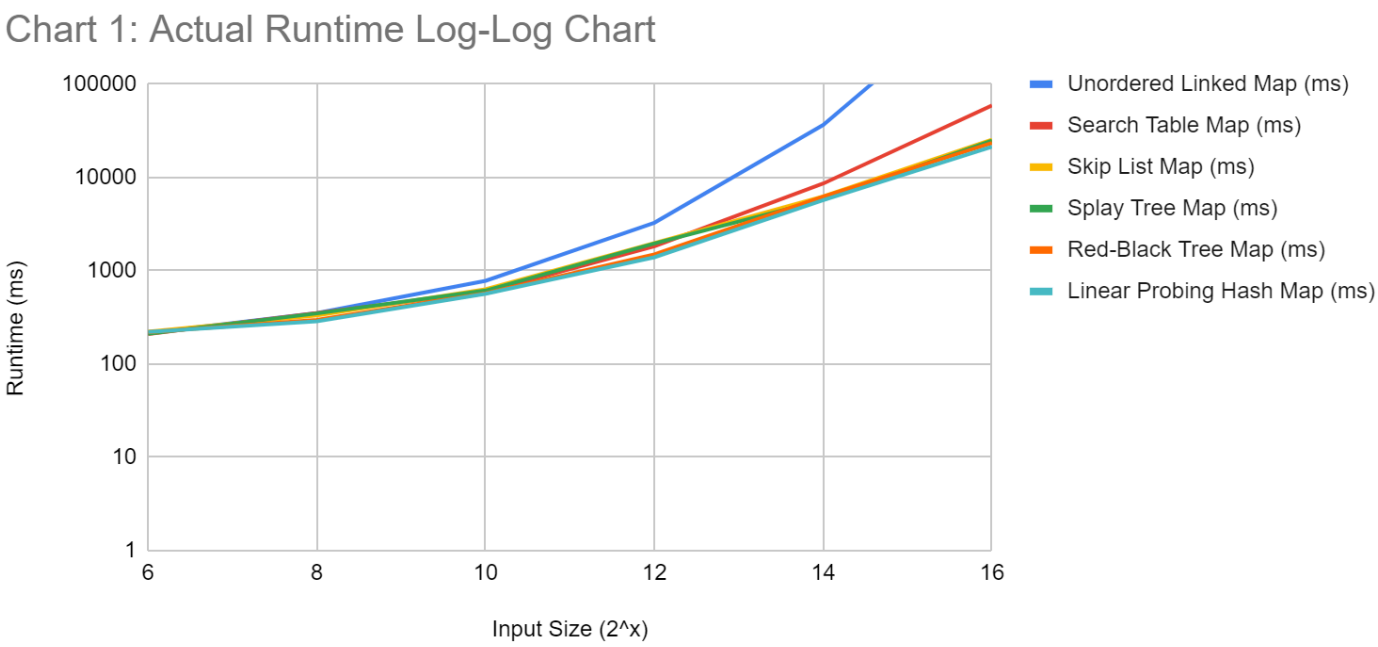


Chart 2: Log-Log Chart of Actual Runtimes Unordered Linked List-based Map

Midpoint: 2^12

Input Size (2^x)	Runtime (ms)	y = alog(x) + b	y = ax + b	y = axlog(x) + b	y = ax^2 + b	y = ax^3 + b	y = ax^4 + b
6	215	1638.5	51.203125	25.6015625	0.8000488281	0.0125007629	0.0001953244209
8	352	2184.666667	204.8125	136.5416667	12.80078125	0.8000488281	0.0500030517
10	781	2730.833333	819.25	682.7083333	204.8125	51.203125	12.80078125
12	3277	3277	3277	3277	3277	3277	3277
14	36607	3823.166667	13108	15292.66667	52432	209728	838912
16	1252010	4369.333333	52432	69909.33333	838912	13422592	214761472

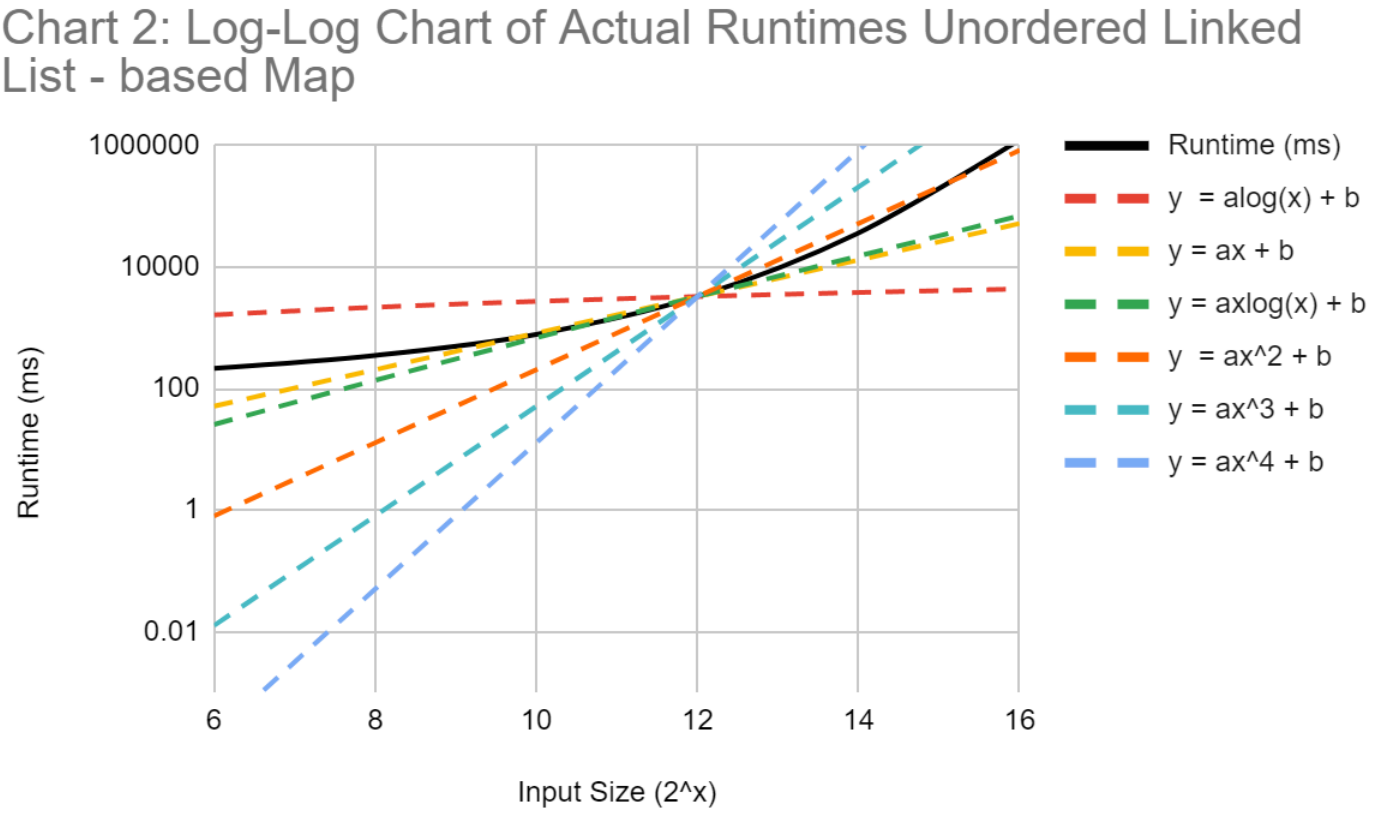


Chart 3: Log-Log Chart of Actual Runtimes for Search Table Map

Midpoint: 2^12

Input Size (2^x)	Runtime (ms)	y = alog(x) + b	y = ax + b	y = axlog(x) + b	y = ax^2 + b	y = ax^3 + b	y = ax^4 + b
6	210	915.5	28.609375	14.3046875	0.4470214844	0.006984710693	0.000109136104 6
8	353	1220.666667	114.4375	76.29166667	7.15234375	0.4470214844	0.027938842 77
10	594	1525.833333	457.75	381.4583333	114.4375	28.609375	7.15234375
12	1831	1831	1831	1831	1831	1831	1831
14	8623	2136.166667	7324	8544.666667	29296	117184	468736
16	59152	2441.333333	29296	39061.33333	468736	7499776	119996416

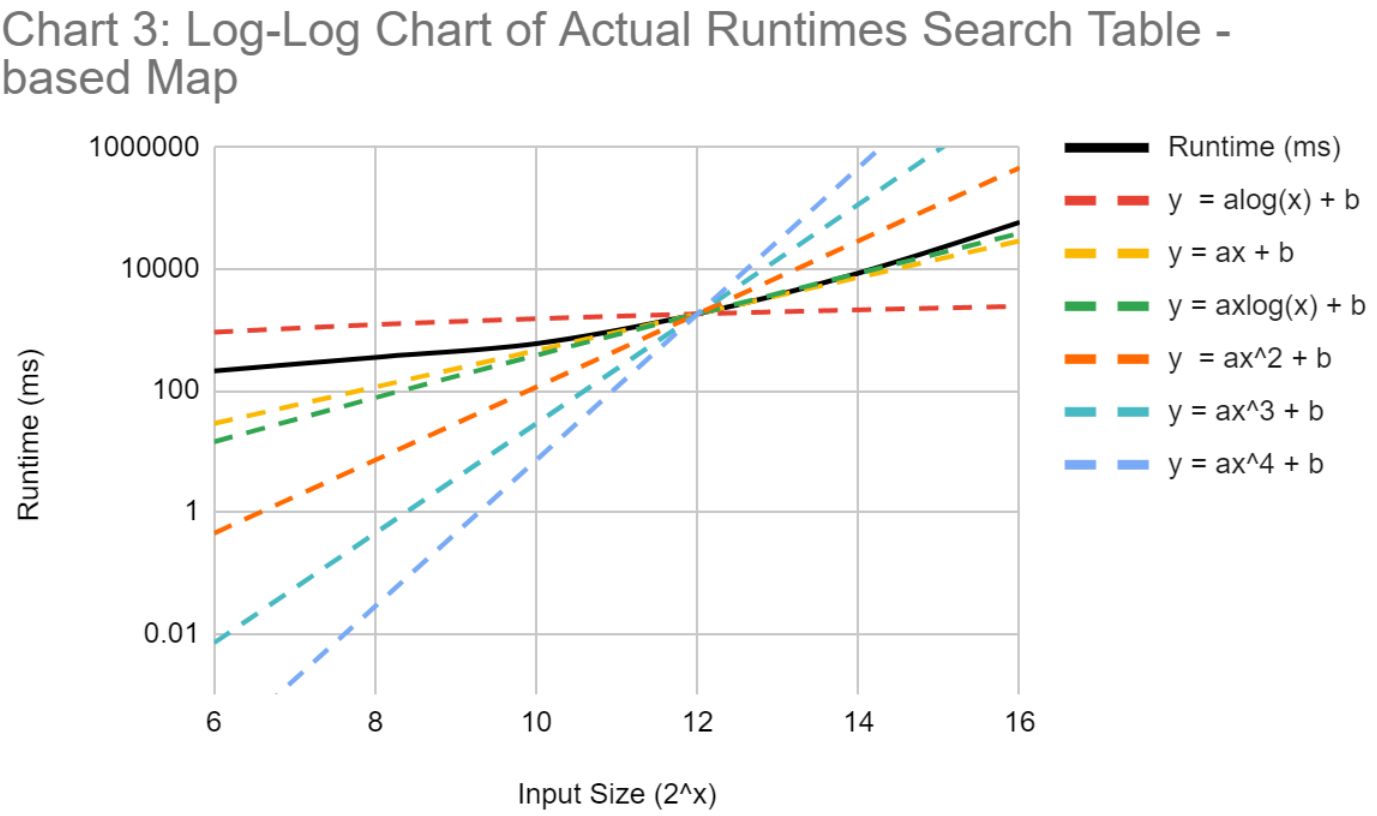


Chart 4: Log-Log Chart of Actual Runtimes for Skip List Map

Midpoint: 2^12

Input Size (2^x)	Runtime (ms)	y = alog(x) + b	y = ax + b	y = axlog(x) + b	y = ax^2 + b	y = ax^3 + b	y = ax^4 + b
6	222	992	31	15.5	0.484375	0.007568359375	0.000118256152
8	332	1322.666667	124	82.66666667	7.75	0.484375	0.0302734375
10	633	1653.333333	496	413.3333333	124	31	7.75
12	1984	1984	1984	1984	1984	1984	1984
14	6273	2314.666667	7936	9258.666667	31744	126976	507904
16	25513	2645.333333	31744	42325.33333	507904	8126464	130023424

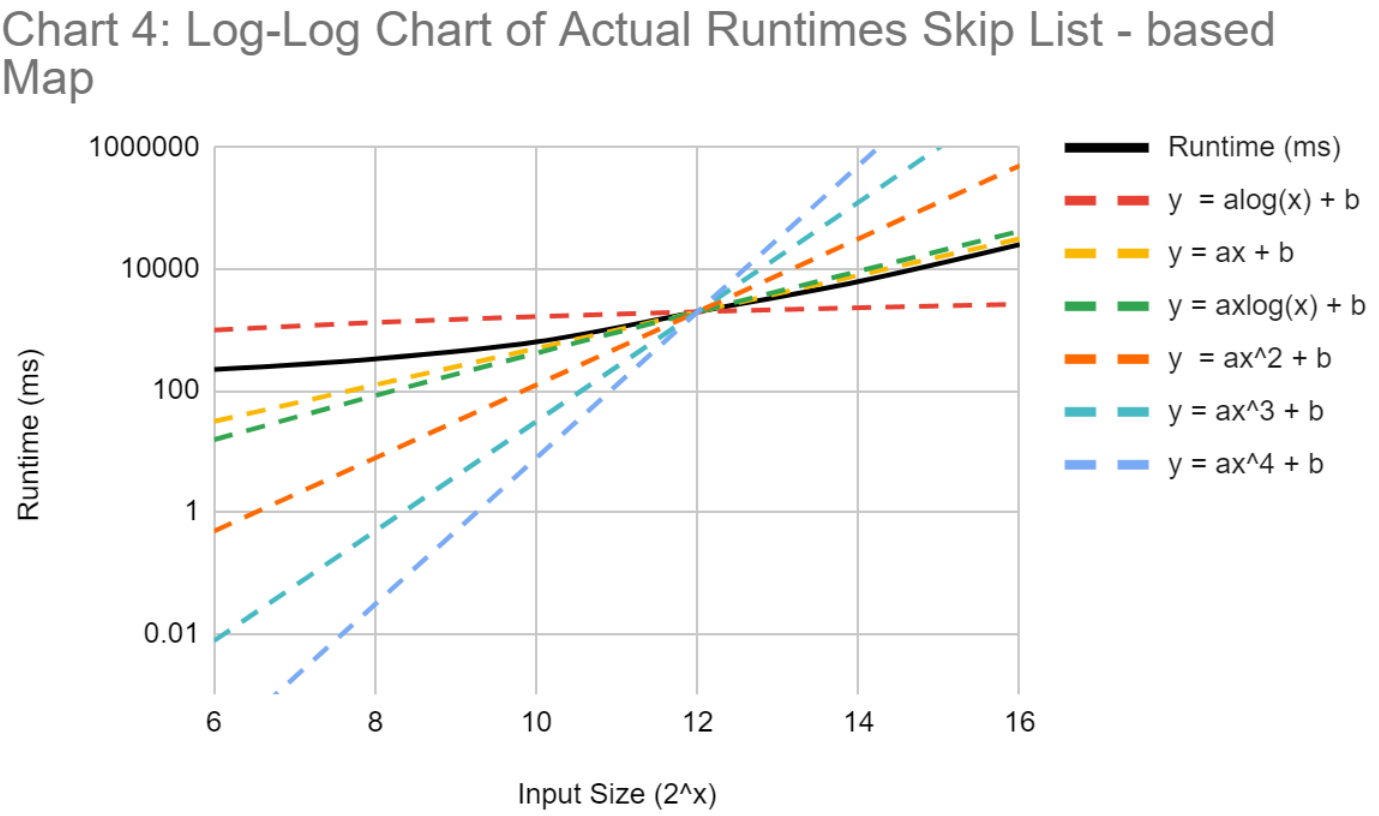


Chart 5: Log-Log Chart of Actual Runtimes for Splay Tree Map

Midpoint: 2^12

Input Size (2^x)	Runtime (ms)	y = alog(x) + b	y = ax + b	y = axlog(x) + b	y = ax^2 + b	y = ax^3 + b	y = ax^4 + b
6	211	982	30.6875	15.34375	0.4794921875	0.0074920654 3	0.0001170635223
8	349	1309.333333	122.75	81.83333333	7.671875	0.4794921875	0.0299682617 2
10	614	1636.666667	491	409.1666667	122.75	30.6875	7.671875
12	1964	1964	1964	1964	1964	1964	1964
14	5790	2291.333333	7856	9165.333333	31424	125696	502784
16	24925	2618.666667	31424	41898.66667	502784	8044544	128712704

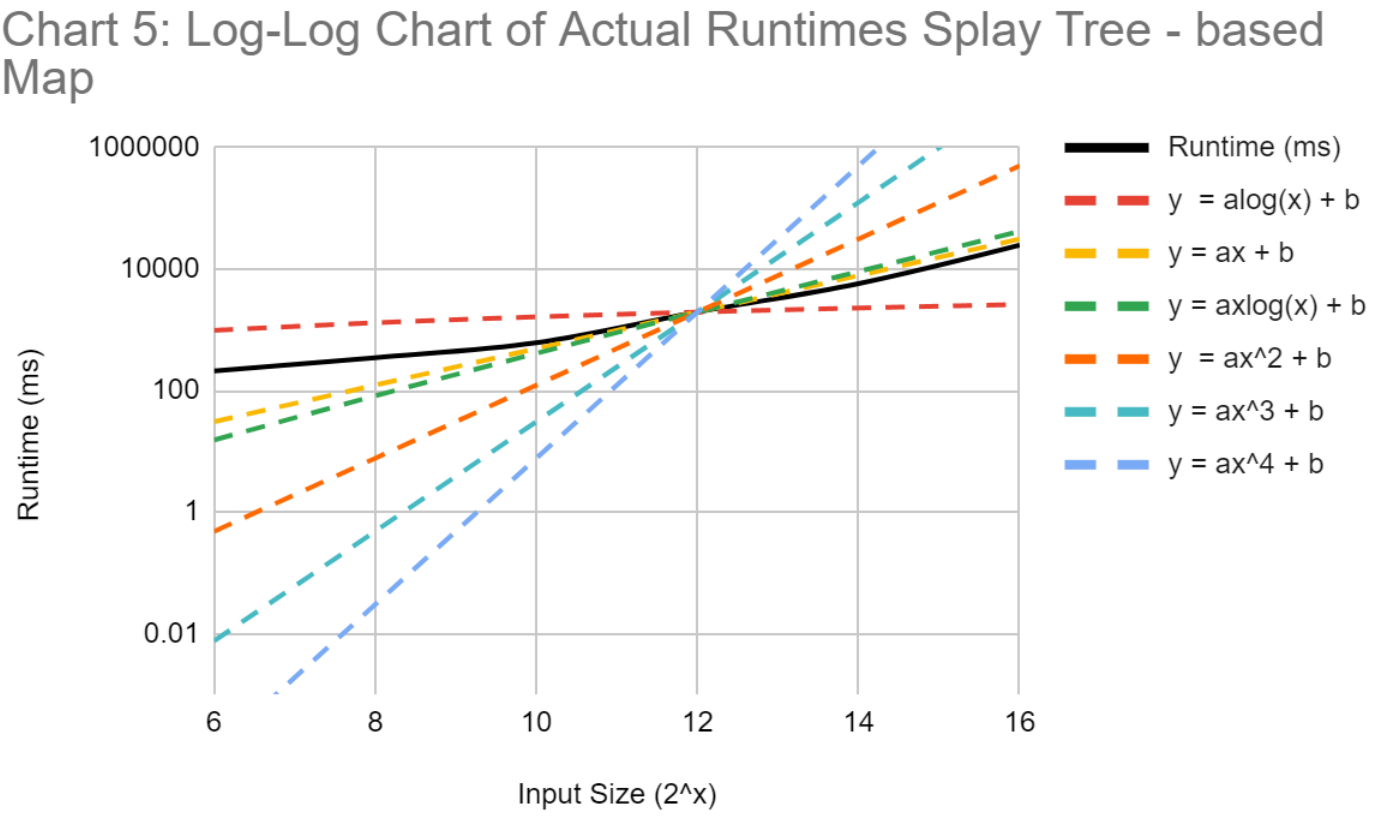


Chart 6: Log-Log Chart of Actual Runtimes for Red-Black Tree Map

Midpoint: 2^12

Input Size (2^x)	Runtime (ms)	y = alog(x) + b	y = ax + b	y = axlog(x) + b	y = ax^2 + b	y = ax^3 + b	y = ax^4 + b
6	220	751	23.46875	11.734375	0.3666992188	0.005729675293	0.00008952617645
8	297	1001.333333	93.875	62.58333333	5.8671875	0.3666992188	0.02291870117
10	574	1251.666667	375.5	312.9166667	93.875	23.46875	5.8671875
12	1502	1502	1502	1502	1502	1502	1502
14	6226	1752.333333	6008	7009.333333	24032	96128	384512
16	23513	2002.666667	24032	32042.66667	384512	6152192	98435072

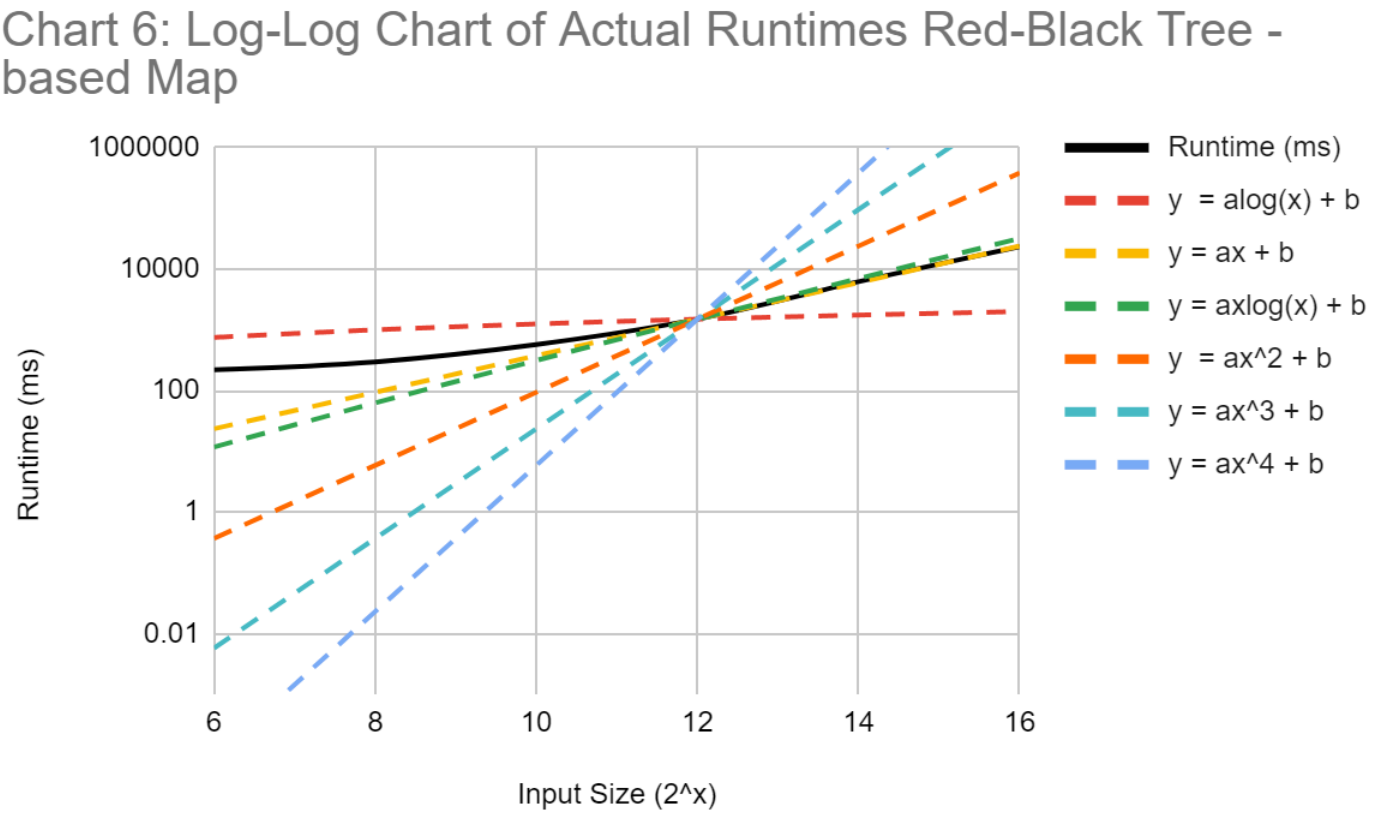
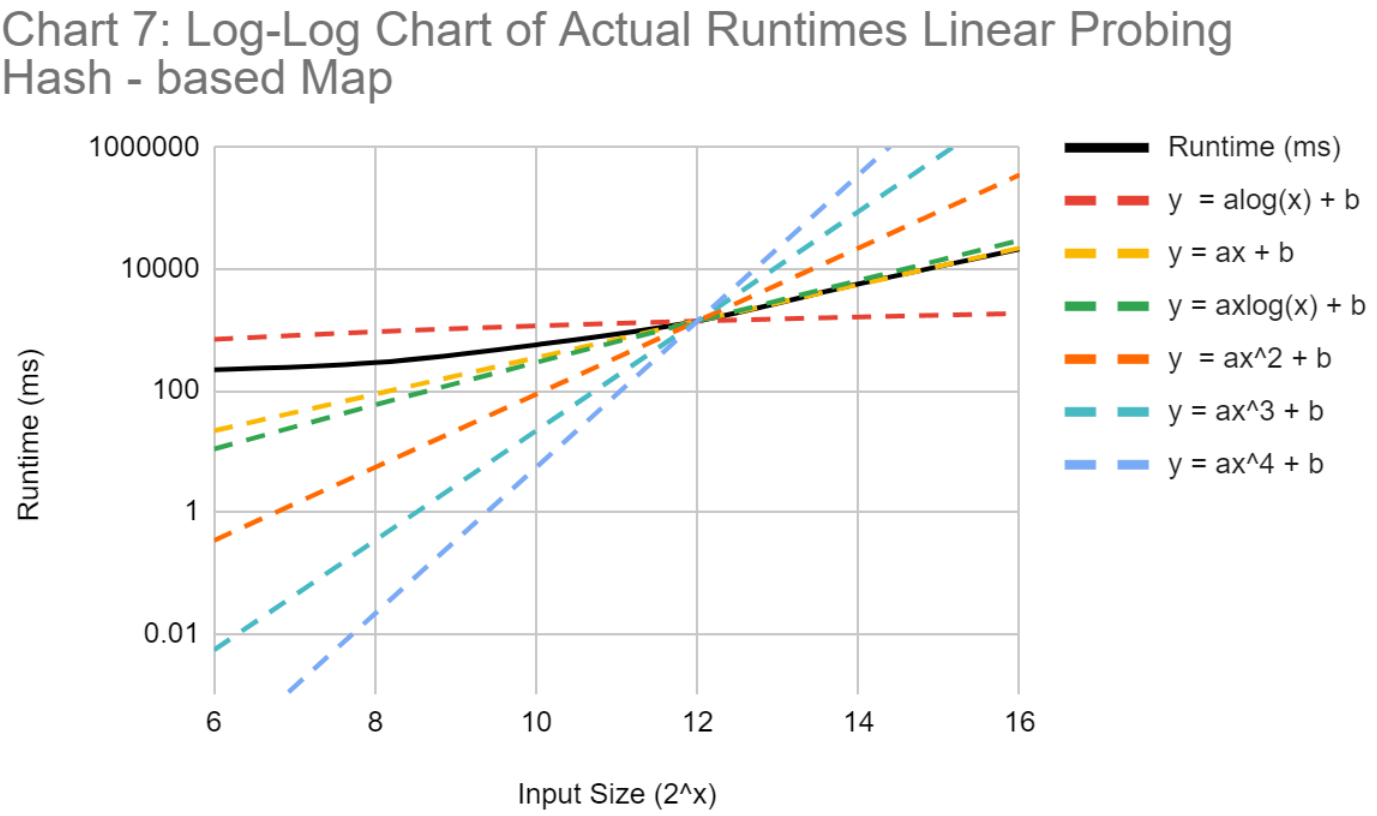


Chart 7: Log-Log Chart of Actual Runtimes for Linear Probing Hash Map

Midpoint: 2^12

Input Size (2^x)	Runtime (ms)	y = alog(x) + b	y = ax + b	y = axlog(x) + b	y = ax^2 + b	y = ax^3 + b	y = ax^4 + b
6	221	698	21.8125	10.90625	0.3408203125	0.005325317383	0.0000832080841 1
8	288	930.6666667	87.25	58.16666667	5.453125	0.3408203125	0.0213012695 3
10	568	1163.333333	349	290.8333333	87.25	21.8125	5.453125
12	1396	1396	1396	1396	1396	1396	1396
14	5711	1628.666667	5584	6514.666667	22336	89344	357376
16	21332	1861.333333	22336	29781.33333	357376	5718016	91488256



Discussion

Reflection on Theoretical Analysis

My algorithm design for getPeopleByHop() originally predicted a runtime of O(n^5) and a average case of O(n^4log(n)). The logic was wrong, since I originally claimed the while loop and first-inner for loop would have a worst case of O(n) each, which is true if the worst case could occur for both, but since they contradict eachother where the larger the n of one of the loops, the smaller it will be for the other, this would actually make it O(n) for these together rather than O(n^2). The next mistake I made is that the second inner loop is not actually dependent on n, the number of people, so it should have a new variable, which we can use m for now. The third issue is that I failed to explain how the third inner loop is dependent on the number of people involved in the call which can have a worst case of n; I did calculate this in my math, but I failed to explain it in my explanation which can cause confusion for someone analyzing my theoretical analysis. With the math for the while loop, and first two inner for-loops fixed, the worst case is O(m * n^3) with an average case of m*n^2log(n) which is alot more efficient than my wrong analysis of O(n^5) worst case and average case of n^4. For my project proposal I decided to use Skip List for the Map ADT because the get, put, and size were the only operations I needed from the map ADT, and size is always O(1), so I only had to consider the get and put. For a SkipList, it had a average expected runtime of log(n) for get and put, which appeared to be the best option, since other operations were O(n), except the search table which had guaranteed O(log(n)) for get, but O(n) for put. So using the Skip List, my corrected theoretical analysis was I could expect a time of m * n^2log(n), my actual runtime, however, followed the y = ax + b and the y = axlog(x) slope, which is below the expected case. There are a few reasons I believe this occurred, the first is that the worst case is not very realistic, this is because the complexity is multiplied by n, since one of the scenarios is that everybody is in every call. Realistically, a call is between one or two people, so a better estimate would be m * nlog(n) expected. Another unrealistic assumption is that each new person's contacts that are checked would have all the contacts that have been previously examined where it has to use the get operation when is expected log(n) for each contact checked which would n(n+1)log(n)/2, this also ties in with the previous point where this would result in an excess amount of people in single calls, so part of the first reduction would also correlate with this one. The third likely reason it performed better than the theoretical analysis is due to good probability, skiplist are RNG based, so the efficiency can be better, or worse than the expected O(log(n)). Due to these reasons, even the corrected theoretical analysis ended up being way off, this is why it is important to recognize the difference between worst case and realistic scenarios.

Reflection on Data Structure Selection

My data structures were appropriate for the given project, for the MapADT I selected a Skip List Map, and for the List ADT, I selected the Singly Linked List with Head and Tail. The reason I made the right choice with the list ADT structure was because in the project, the only list operations I used were size(), addLast(), and removeFirst(), which all are O(1) in the chosen structure. The Map ADT selection was a bit more complicated, for this ADT I needed to use the operations, get(k), put(k, v), and size(), but size() is O(1) in all implementations so it can be ignored. The two underlying structures that stood out were the Search Table implementation which was O(log(n)) for the get operation, and O(n) for the put operation, and the SkipList which was expected log(n) for get and put, but O(n) for both of them also. My final decision was to go with the search table, since when dealing with large amounts of data, the worst case scenerios become less and less realistic, while it will slowly skew to the average case. My choice proved correct when I tested the files, where the table below shows the result. As the data set grew larger, the skip list became more efficient than the search table.

Input Size (2^x)	**Unordered Linked Map (ms)**	**Search Table Map (ms)**	**Skip List Map (ms)**
6	215	210	222
8	352	353	332
10	781	594	633
12	3277	1831	1984
14	36607	8623	6273
16	1252010	59152	25513
18	> 30 min	1411995	106322
20	> 30 min	> 30 min	error

Improving Efficiency

The main issue with my efficiency was that even though in my project proposal I selected a Skip List as my Map ADT, when it came to selecting my default Map Structure, I ended up using the UnorderedLinkedMap structure. This resulted in O(n) get and put rather than a expected O(log(n)), which hurt my project during large test, which is why I believe I failed the time portion. Below shows a table of the efficiency difference between a Skip List and a Unordered Linked Map implementation which shows the impact of my decision.

Input Size (2^x)	**Unordered Linked Map (ms)**	**Skip List Map (ms)**
6	215	222
8	352	332
10	781	633
12	3277	1984
14	36607	6273
16	1252010	25513
18	> 30 min	106322
20	> 30 min	error

Lessons Learned

The main lesson I took from this project is the true impact of choosing the most efficient data structure for your algorithms when it comes to large input sizes. Even though all the structured performed with similar efficiencies for small input sizes like 2^6 and 2^8, the difference between them changed drastically as the input sizes would increase. For 2^16, the Linear Probing Hash Map took 1.7% of the time as the Unordered Linked Map, and by 2^18. The Unordered Linked Map wasn't even able to finish within the 30 minutes. Another lesson I learned was the difference between theoretical analysis and reality; where the theoretical analysis, which assumed worst cases, resulted in a way worse efficiency compared to the results. When considering worst case, some of the situations that come up are unfeasible, like every single call in the data involving every single person. So a big lesson from this portion of the project is that when considering worst case scenarios, sometimes it is also important to consider how realistic they actually are.