CS 6968 Final Project Report – Graveyard Hashing

Pranjal Patil, Zixiao Chen

September 13, 2023

1 Project Summary

This project focuses on the implementation and evaluation of three different hashing techniques: Graveyard Hashing, Robinhood with tombstones, and Robinhood without tombstones ([PCM])in C++. Graveyard Hashing is a variant of Linear Probing[Wika] Hash Table, where artificial tombstones are inserted during resizing to improve performance.

The primary goal of this project is to demonstrate how Graveyard Hashing can outperform Linear Probing in terms of reducing primary clustering[Wikc]. As per the research in [BKK], data points tend to cluster together, which limits the availability of free slots for new data insertion.

The performance of the three hash tables will be compared by analyzing the amortized performance of insertion, deletion, and queries. We will be examining the effects of artificial tombstones and aging, which involves consecutively inserting and deleting elements in a hash table that is already filled to its load factor.

Overall, this project aims to provide insights into the performance of different hashing techniques(Graveyard, Robinhood Tombstones) and how they can be optimized to achieve better efficiency and reduced primary clustering.

Cuckoo Hashing ([Wikb]) and Quadratic Probing ([Gee]) has also been implemented and the evaluation numbers are present in the result.

For the Cuckoo hashing and quadratic probing, the hash tables supporting these data structures are implemented with the pointers as a representation of array, parameters such as seed, initial size, and maximum number of iterations to support the timing experiments.

The quadratic probing has a limited size of the table. Therefore, elements ranked the least recently inserted will be thrown out of the hashing table and

new items will be inserted instead. Whenever a collision happens between the old and new inserted items, a swapping will take place among them. These techniques work the same for Cuckoo hashing.

2 Items proposed in the initial Report

- In the initial report, we proposed to implement hash tables with different hashing techniques including quadratic hashing, cuckoo hashing with quadratic probing technique, Robinhood Hashing, Robinhood Tombstone and graveyard hashing with basic functionalities such as insert, query and delete.
- These basic techniques will also be utilized to test the time performance of different hashing techniques and prove how the difference of utilizing spaces, distribution parameters, rebuilding can make a difference in the time performance of the above mentioned Hash Table. Particularly, the relationship between the time performance of each operation and the number of iterations will also be evaluated for Cuckoo hashing with quadratic probing.
- Last but not least, an experimental and theoretical analysis will be conducted for different hashing techniques to understand how the structural difference can lead to variation of time complexity.

3 Items Accomplished

- Implemented the Robinhood, Robinhood tombstone and Graveyard Hashing techniques. We studied its working and structure based on different parameters like redistribution frequency, rebuilding frequency, load factor etc.
- Evaluation based on these above factors were performed and complexity were observed. In addition to that we also tried evaluating standard unordered map and compared its performance of operations with the above three classes that we implemented.
- Implemented hashing with quadratic probing and Cuckoo Hashing. Specially, implement with various parameters including initial size, seed, and maximum number of iterations allowed within the hash table for insertion, query and lookup.
- Besides, debug code is also implemented for ensuring the basic function under basic case won't fail. Timing code for both hashing functions is set up with the variation of number of maximum iterations. The timing of all the operations for the hash tables are obtained to get a basic sense of algorithmitic behavior.

- Makefile is also built to ensure a more user-friendly process of generating and removing executables.

4 Techniques of testing/benchmarking implementation

All the tests were ran on cade machine.
We created a Hash Table of 25 Million for each of the classes(RobinHood, RobinHood Tombstone and Graveyard Hashing, Unordered map). First we inserted elements such that it filled to its load factor(90 percent). We inserted 22500K elements in each of the Hash Table.
Now to understand aging in these we alternatively inserted, deleted, queried approx 1.4 percent of 25M in these table. This will be 360900 elements.

Each of these sets are denoted as a iteration.

For the case of Graveyard Hashing we redistributed(x is 12 one in every x is tombstone) the tombstones in the rebuild function. This was done once in every k iterations. The average time in redistribution was considered in the results. Here k in results are 4, 8, 16, 32.

For the case of Robinhood with Tombstones we did the rebuilding same as in Graveyard Hashing. But in this case rebuilding means removing all tombstones so that all elements are in their correct places (case when tombstones never existed).

During insertion in all the cases the number of positions we had to iterate to find the free slot was also calculated and understood. All the evaluations done were amortized.

Unordered map performed better than all the other hash tables. Reason is that the built in Hash Table is highly optimized with its algorithm and code.

As for the hashing with quadratic probing and the Cuckoo hashing with quadratic probing, the performance is evaluated in terms of the relationship between time complexity of each operation and the number of maximum iterations that the hash table can tolerate. The number of iterations is evaluated in terms of the number of loops an inserted or swapped item goes through.

The timing experiments borrowed some of the timing code from Assignment 1. Particularly, I used the rand_bytes method from test.cc to generate random numbers for insertion, query and lookup for 100000 items. The timing code is also run with a looping function regards to the maximum number of iterations that hashing function can tolerate.

5 Final sets of results and Analysis

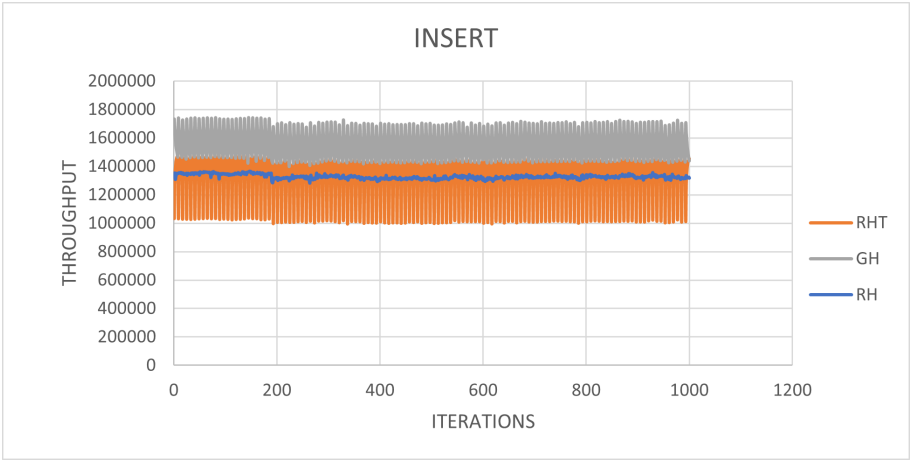
CASE 1 :
Let us understand the case when k is 8. That is once in every 8 iteration the Graveyard Hash Table is rebuilded(Redistribution of tombstones) and Robinhood Tombstone is rebuilded(all tombstones are removed).

Graphs: Here there are 4 kinds of graphs. Insertion, deletion, query and CDF(Cumulative Distribution Function).

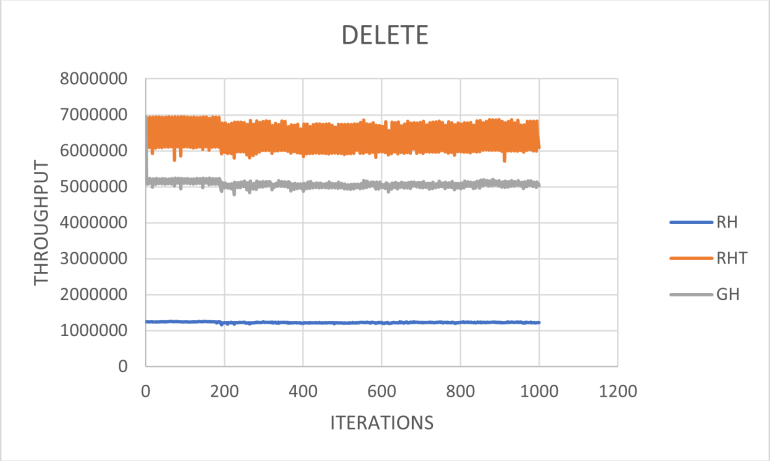
The insert, delete and query have same format. X axis is the number of iterations, y axis is the throughput and legends denote - RH(Robinhood), RHT(Robinhood Tombstone), GH(Graveyard Hahsing).

The CDF Graph has the following format- X axis the hops per insertions, Y axis is the frequency percentage.

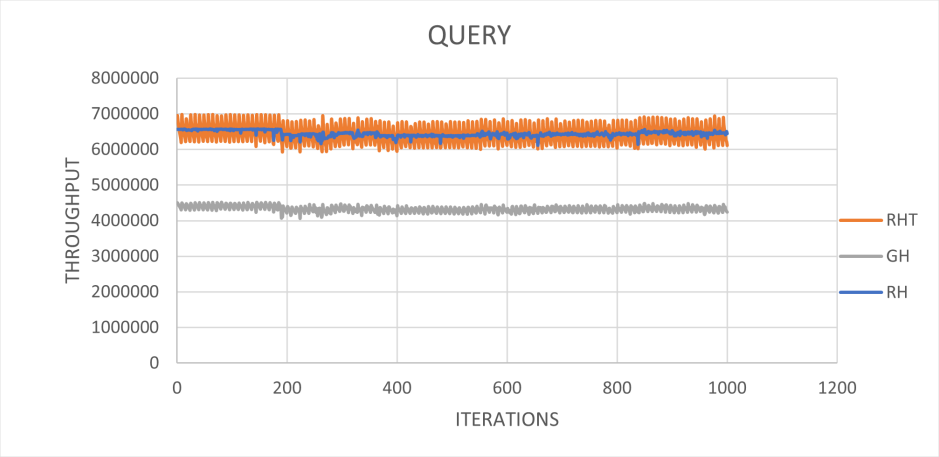
Insertion: We can observe that Graveyard has the best throughput(higher is good). This is because of tombstones. Due to tombstones the algorithm is able to find a free slot in linear time in terms of x(1 in every x slot is free). Here x is 12. Robinhood tomb performs better than robinhood in every 9 of 10 iterations. It the 10th iteration as the robinhood tomb rebuild its insertion will be same as robinhood and it will also consider time to rebuild. Hence in this case only the throughput of robinhood tomb is lower that robinhood.



Deletion: Here it is observed that Robinhood tombstones has the best deletion throughput and then Graveyard and then Robinhood. There are two factors to be considered here. First is number of tombstones because tomstones make it slower to find an element. Second is shifting of element in case of Robinhood Hash Table. This takes time. Hence Robinhood takes longest time and as Graveyard has more Tombstones that Robinhood Tombstones it takes compartively longer time.

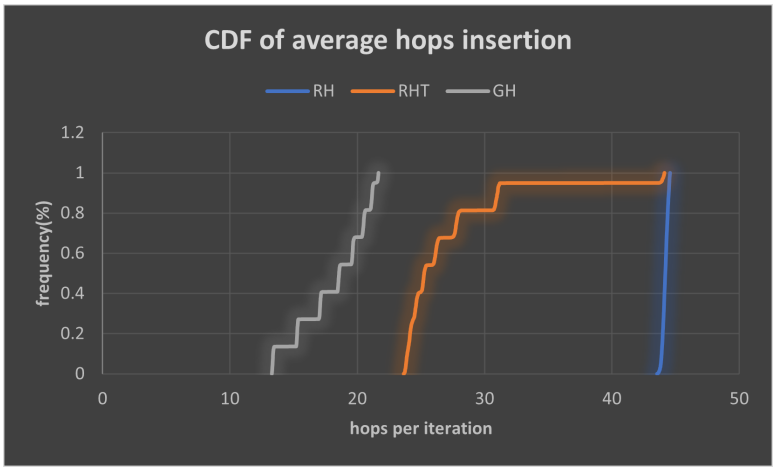


Query: Here it is observed that Graveyard Hash has lowest throughput. This is because queries are highly affected by tombstones. More tombstones more time to query. RHT oscillates above and below RH. This is because of rebuilding. When the tombstones are high in RHT it takes more time than RH.

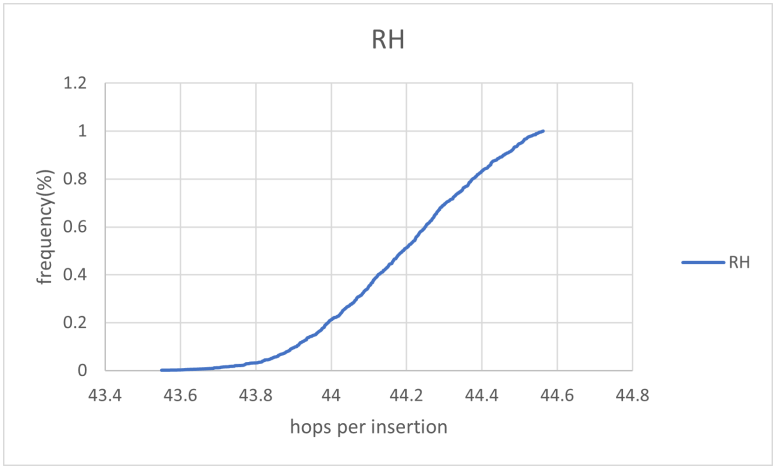


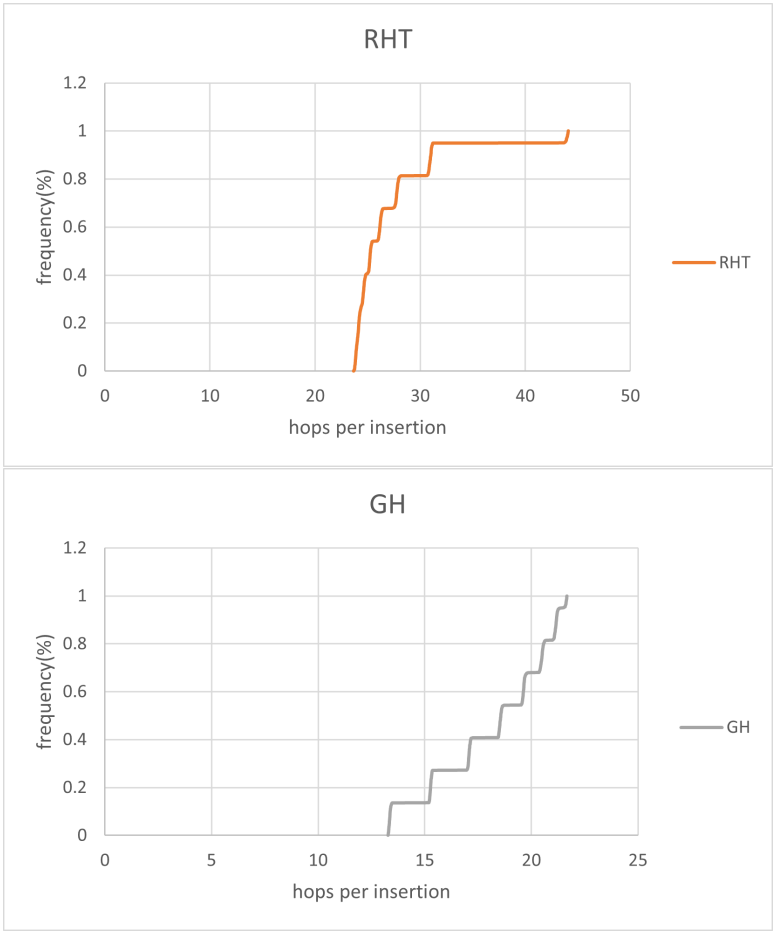
CDF: Here we try to understand the length of array iterated in insertion. The average has been taken into account. iteration length(hops) per insertion. As

expected due to tombstones the number of hops in Graveyard in lowest and then Robinhood Tombstones has lower hops than Robinhood except in case of rebuilding where it is equal as observed in the graph.



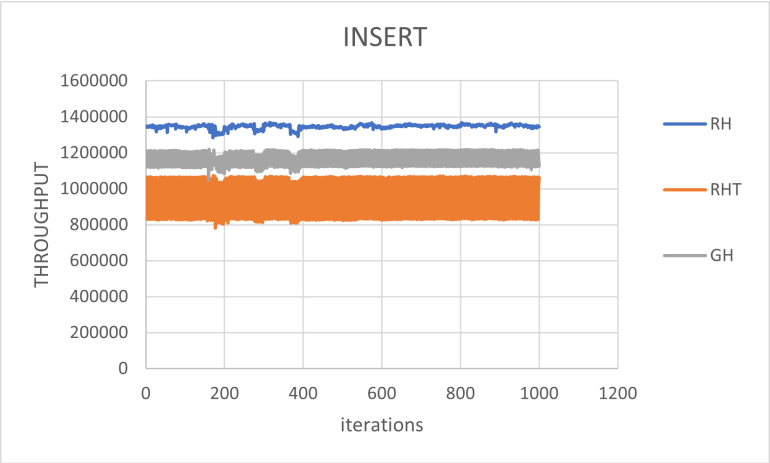
Below are the CDF of individual classes to understand it more clearly.





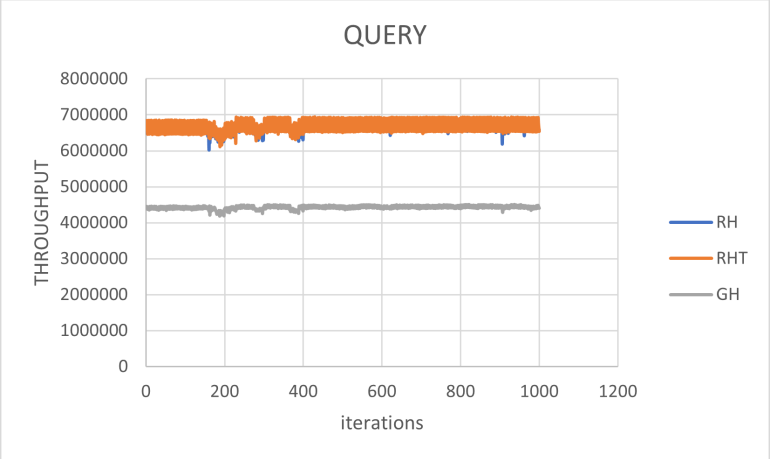
CASE 2: Let us understand the case when k is 4. That is once in every 4 iteration the Graveyard Hash Table is rebuilded(Redistribution of tombstones) and Robinhood Tombstone is rebuilded(all tombstones are removed).

Insertion: Unlike CASE 1 we observe that throughput of Robinhood is highest. This is because the time to rebuild/redistributed dominates over the time to insert. Hence in insertion Robinhood has best throughput, then Graveyard and then Robinhood Tombstones.



Deletion: The deletion behaviour is same as before.

Query: The query behaviour is also same as before. The major effect is only seen in insertion.



CASE 3 AND 4: Let us understand the case when k is 16 and 32. That is once in every 16/32 iteration the Graveyard Hash Table is rebuilded(Redistribution of tombstones) and Robinhood Tombstone is rebuilded(all tombstones are removed).

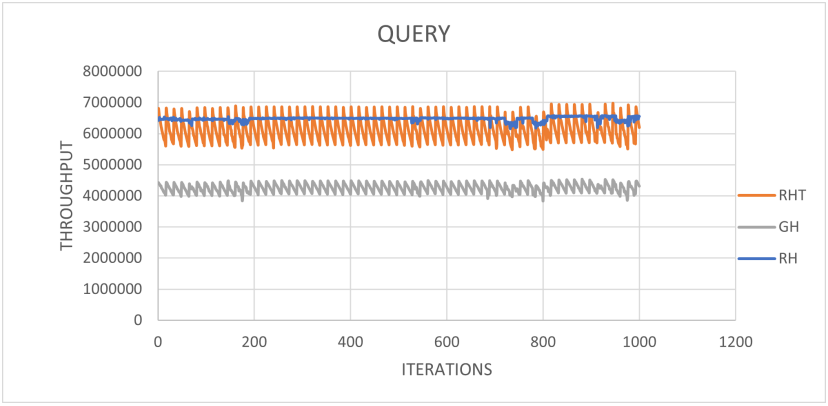
Insertion: Here we observe that as we are not rebuilding/redistributing tables as frequently the random formation of tombstones in table as iteration passes causes the insertion of graveyard to overlap with Robinhood tombstones. This make Graveyard insertion time more that Robinhood tomb in many cases. For robinhood and Robinhood tomb it is as expected as in case 1. As before also the tombstones in RHT were randomly spread the RHT vs RH has same behaviour. The insertion in k as 16 and k as 32 makes the above understanding more clear. In most cases the Graveyard takes longer time that RHT in case of k as 32.

Deletion: Due to higher randomization of position of tombstones in Graveyard (due to higher k). the performance of GH and RHT is closer than that of other cases. In k as 32 the RHT and GH is closer than k as 16 and followed by k as 8 and 4. In every case Robinhood takes the longest time hence lowest throughput.

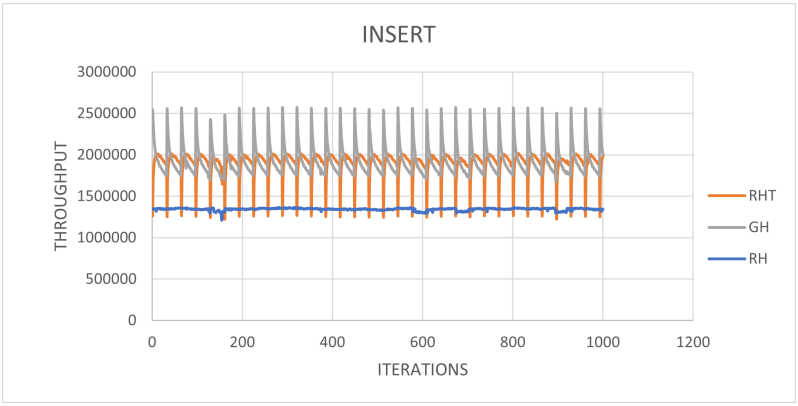
K=16

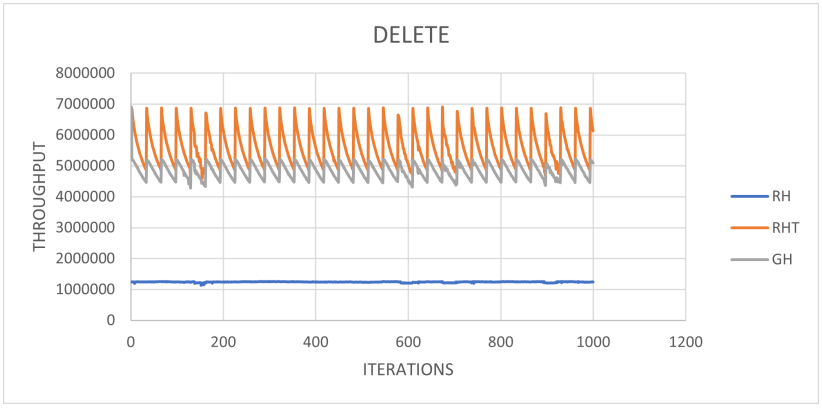

INSERT
2500000
2000000
1500000
1000000
500000
0
THROUGHPUT
0
200
400
600
800
1000
1200
ITERATIONS
RHT
GH
RH

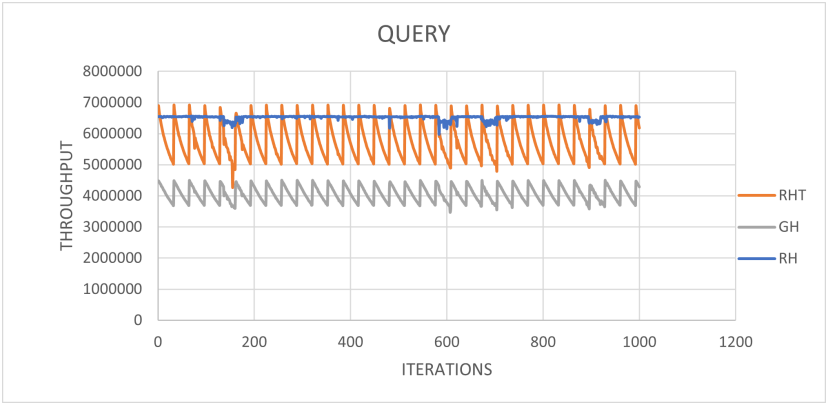

DELETE
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
THROUGHPUT
0
200
400
600
800
1000
1200
ITERATIONS
RH
RHT
GH


QUERY
8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
THROUGHPUT
0
200
400
600
800
1000
1200
ITERATIONS
RHT
GH
RH

K=32

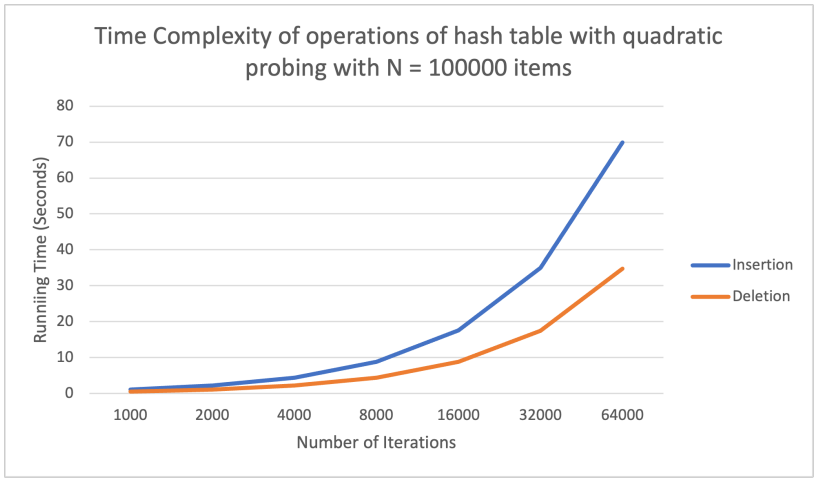






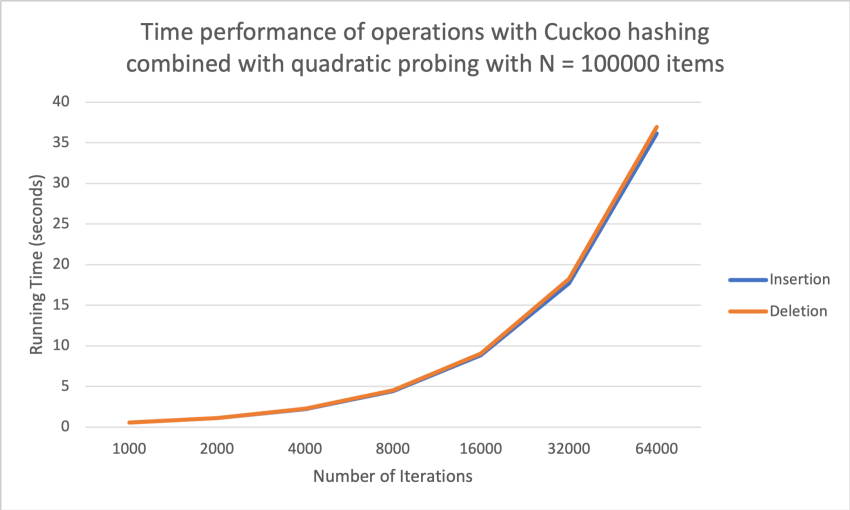
In order to illustrate how the performance of Graveyard Hashing has over the hashing with quadratic probing and the Cuckoo Hashing with quadratic probing, the timing experiments and the corresponding plots are produced.

Since the hash table won't be resized, the new item will be inserted and swap values with the items already inserted in the hash table. By doing so, the least recently inserted items in the hash table will be likely to be kicked out. The number of iterations is counted by the number of times the looping of the hash table happens.



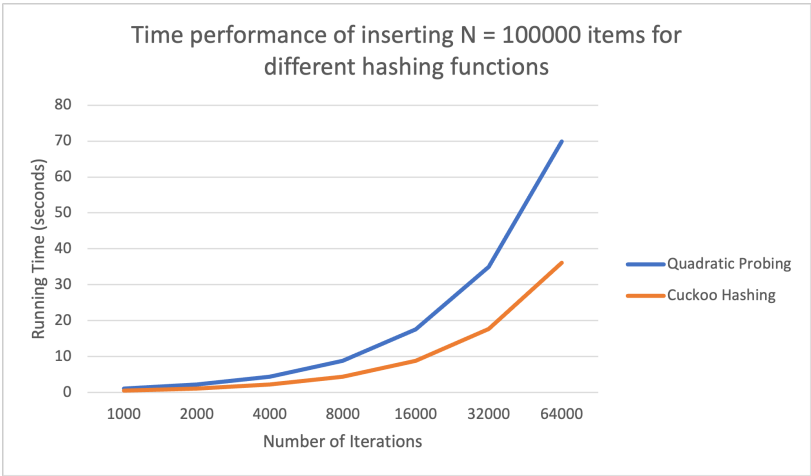
The graph above shows the relationship between time performance of insertion m deletion of quadratic probing with inserting and deleting $N = 100000$ items and the maximum number of iteration of hash table that the table can tolerate.

The hash table for this timing experiment is set up with the size of 10. Based on the above table, it can be inferred that the insertion for the hash table takes twice as much as the deletion does for dealing with 100000 items.



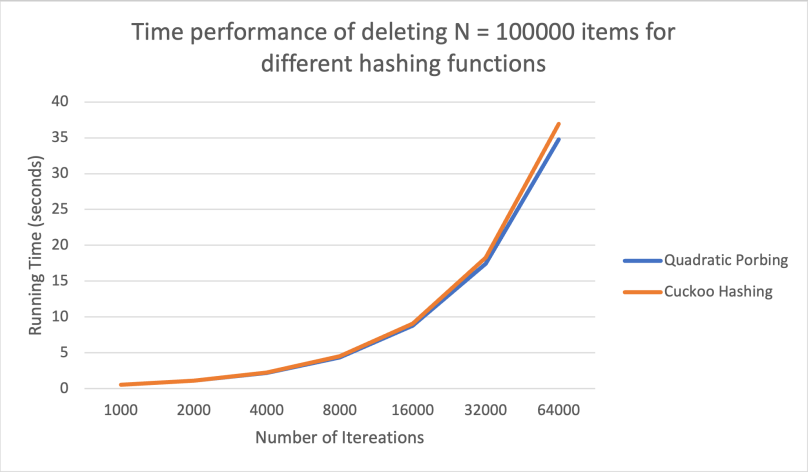
The graph above shows the relationship between time performance of insertion, deletion of cuckoo hashing with quadratic probing with $N = 100000$ items and the maximum number of iteration of hash table that the table can tolerate.

The hash table for this timing experiment is set up with the size of 10. Based on the above table, it can be inferred that the insertion for the hash table takes almost as much as the deletion does.



The graph above shows the relationship between time performance of insertion under quadratic hashing, cuckoo hashing with quadratic probing with $N = 100000$ items and the maximum number of iteration of hash table that the table can tolerate.

The hash table for this timing experiment is set up with the size of 10. Based on the above table, it can be inferred that the insertion under quadratic probing takes twice as long as the cuckoo hashing does.



The graph above shows the relationship between time performance of deletion under quadratic hashing, cuckoo hashing with quadratic probing with $N = 100000$ items and the maximum number of iteration of hash table that the table can tolerate.

The hash table for this timing experiment is set up with the size of 10. Based on the above table, it can be inferred that the deletion under quadratic probing almost takes as long as the cuckoo hashing does.

6 Things that we did not achieve and reasons

- One of the aspect that fails to achieve is trying to do a timing experiment of Cuckoo hashing based on the number of rows, i.e., number of different seeds applied. An important key factor of this phenomenon is the overestimation of the programming ability of tackling C++ languages. Since a considerable amount of time is built on testing the functionality, the time of implemented variation of Cuckoo hashing based on the number of rows is limited.

- Another aspect that doesn't make it is the bugs appeared in the code. Even though some of the code for Cuckoo hashing and hashing with quadratic probing is bug-free for the basic cases, there still appears to

be bugs affecting the time performance of insertion or deletion. Nonetheless, the affect on the time performance is minor.

- Finally, the time performance of the query function for hashing with quadratic probing and Cuckoo hashing can't be plotted in a way that the phenomenon is observable, even with 100000 items. Since the timing performance of the query function for both of these hashing functions took approximately 0 seconds, the plot itself will reveal little pattern about the difference of behaviour between Graveyard hashing, Cuckoo hashing or even hashing with quadratic probing.

7 Overall Analysis

Here for the case of Robinhood, Robinhood Tombstones and Graveyard it is observed that Graveyard performed the best for insertion taking the least time and high throughput. This proved that adding artificial tombstones can actually help in mitigating primary clustering problem. The complexity to insertion was linear in terms of x where x indicates 1 in every x elements is a tombstone.

Then we observed that for Robinhood tombstones as the deletion occurred randomness of tombstones was high so even-though it was slower than Graveyard due to tombstones it was faster than Robinhood.

In case of Robinhood the number of hops/travel in table was highest. This shows the bad effects of primary clustering in this Table.

We can clearly prove these numbers in the CDF plot.

In deletion and query if we compare the Robinhood and Robinhood Tombstones its observed that due to tombstones the process becomes slower, hence Graveyard is slower than Robinhood Tombstone. The Analysis of Deletion and query is done in the above section in detail.

Based on the plots generated from the timing experiments of both hashing via quadratic probing and Cucukoo hashing, the following patterns can be obtained:

- Since the quadratic probing will eventually lead to a higher likely of getting more primary clusters as eventually the quadratic difference will come in the period of the table size, in this case the period will be 10. Hence, the insertion inhabits linear complexity as the number of maximum iterations to be tolerated will come in $O(n)$ for inserting $O(n)$ many items, which is

approximately the average time complexity for insertion when the table becomes infinite.

- The deleting is somehow to be twice as fast as the insertion can be accounted for kicking out the least recently used item out of the hash table. As a matter of fact, the swapping makes some operations to be $O(1)$ in deleting items.
- For the Cuckoo hashing, the insertion and deletion are approximately the same. That can be accounting for an additional row of the hash table operating with a different seed, reducing the likelihood of collision.
- Meanwhile, the relationship between the running time and the maximum number of iterations also have $O(n)$ for both insertion and deletion operations. However, they are at approximately the same pace as the insertion will have a back up hast table at another row and the swapping will happen if the items at two different hash indices with different seeds happen.

On the other hand, if compared with the quadratic probing and the Cuckoo Hashing for the operations, the following can be suggested:

- Quadratic probing will be more costly in insertion than cuckoo hashing. One reason for such is the structural differences between the hash table with solely on the array itself for quadratic probing and the other one for Cuckoo hashing with additional hash table yet different seed in getting hash values.
- However, the deletion time doesn't make any different for either of the two hashing techniques. A possible reason for that is the swapping reduces the time to collide with each other and also gets rid of the least recently used items from the hash table.

Compared with the Graveyard Hashing with different measurement metric, it may seem that the insertion of the items will remain constant for the throughput and still gets $O(1)$ time complexity for having most of the insertions to be less costly than $O(n)$ even with a larger size of hash table. On the other hand, the quadratic probing and cuckoo hashing may see an increase in times complexity even with the max number of iterations increased or have a huge table size. Hence, it may be reasonable to say Graveyard hashing beats Cuckoo hahing and even quadratic probing in some ways for better utilization of invalid or deletion space.

However, it is still worth exploring whether the number of rows of the Cuckoo Hashing still has a effect on the performance such that the variation of Cuckoo hashing can beat Graveyard hashing in some ways. As indicated in the plots, the number of rows under the Cuckoo Hashing play a big role in keeping down the insertion time. Therefore, by utilziing the number of rows and even by using the invalid space as a backup, the Cucukoo Hashing may even perform better than the Graveyard Hashing.

8 Github repo link

The github link for the project is:

`https://github.com/PranjalJPatil/Graveyard_Hashing`

9 Contributions

Pranjal:

- I contributed in development of Graveyard Hashing. I developed the classes for Robinhood, Robinhood Tombstones and Graveyard Hashing. Each of these classes have insert, query, and delete method.
- I also coded the driver method that evaluated the performance of these classes. The Graphs, and CDF plots were evaluated.
- The evaluations and performance calculated are amortized. I also did Optimization of the code for these classes so that we get good evaluation numbers.
- I also performed evaluation on std unordered map of C++. In this I got unordered map to perform better than the graveyard.

Zixiao (Martin):

- I've implemented hash tables for both quadratic hashing and the cuckoo hashing with insert, query, and delete functionality. I've also implemented various parameters for both hash tables to ensure great flexibility in debugging code and timing experiments.
- In addition, I've also implemented debug code for testing purposes and timing code. The timing code borrows the timing program from the Assignment 1. To coordinate with multi-purposes with running multiple programs, I also included a Makefile responsible for compiling and cleaning the programs.
- To get an accurate estimate of the hash table, I also implemented the iterations of an item went through within the hash table. By including the maximum iterations that the hash table can contain and the swapping between inserted item and original items within the hash table, these hash tables also achieved the expected time complexity as mentioned in the report.

10 References

References

[BKK] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. Tombstones mark the death of primary clustering: https://arxiv.org/pdf/2107.01250.pdf.

[Gee] GeeksforGeeks. Geeksforgeeks contributors. quadratic probing: https://www.geeksforgeeks.org/quadratic-probing-in-hashing/.

[PCM] Per-Ake Larson Pedro Celis and J. Ian Munro. Robin hood hashing (preliminary report): https://ieeexplore.ieee.org/document/4568152.

[Wika] Wikipedia. Linear probing: https://en.wikipedia.org/wiki/linear_probing.

[Wikb] Wikpedia. Wikipedia contributors. cuckoo hashing: https://en.wikipedia.org/wiki/cuckoo_hashing.

[Wikc] Wikpedia. Wikipedia contributors. primary clustering: https://en.wikipedia.org/wiki/primary_clustering.