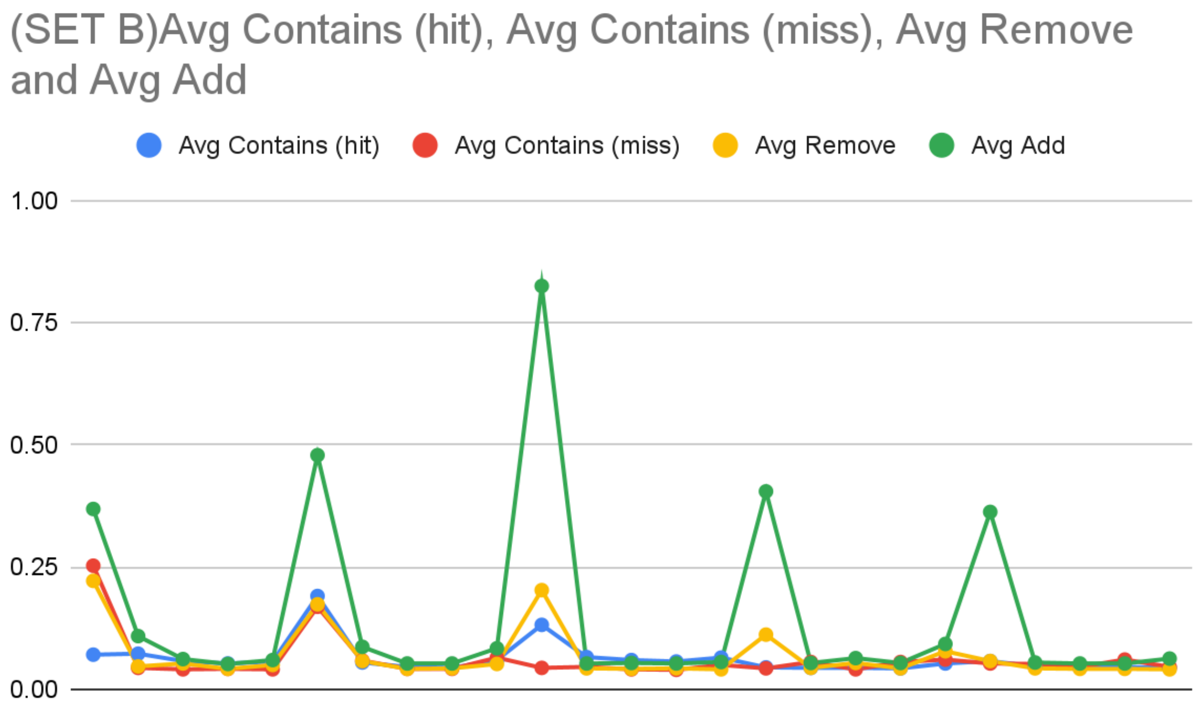
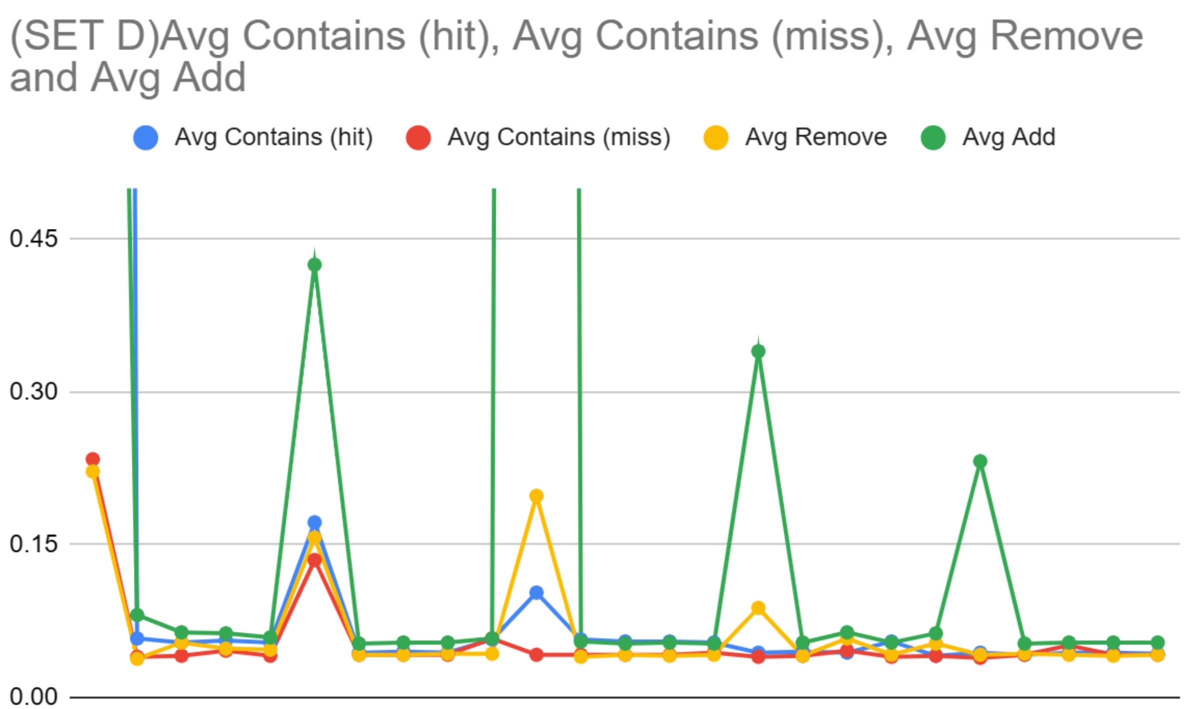
Hashes - Research Report

By Maddox Krape

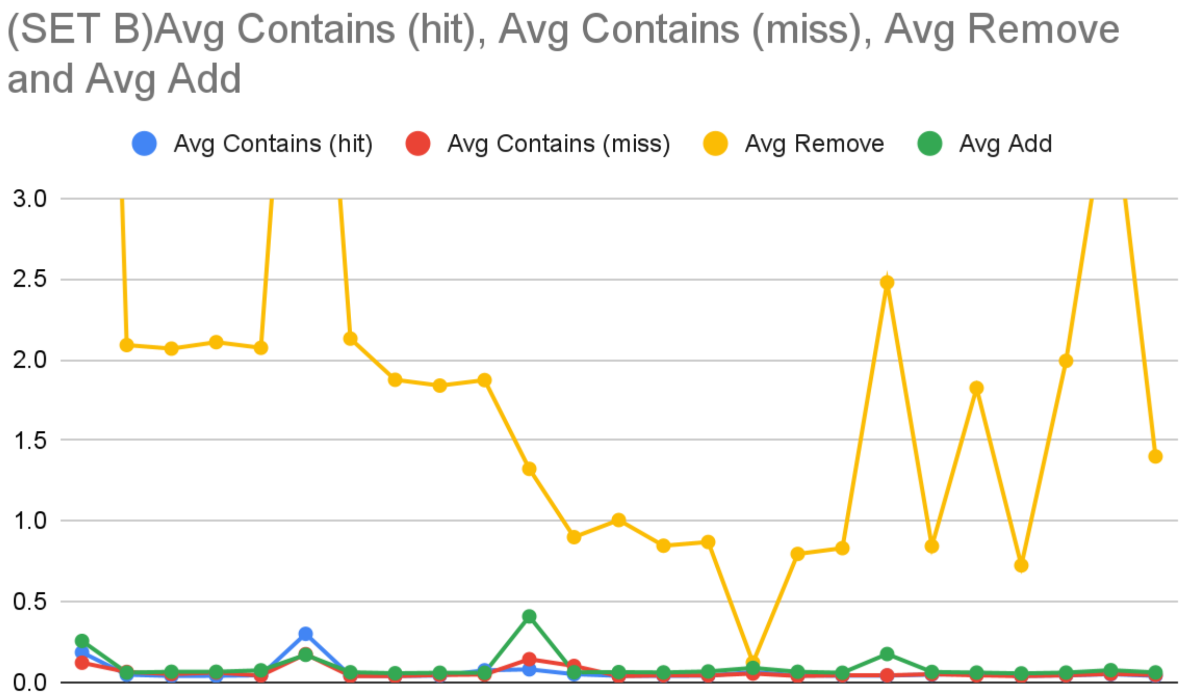
Spreadsheet Source

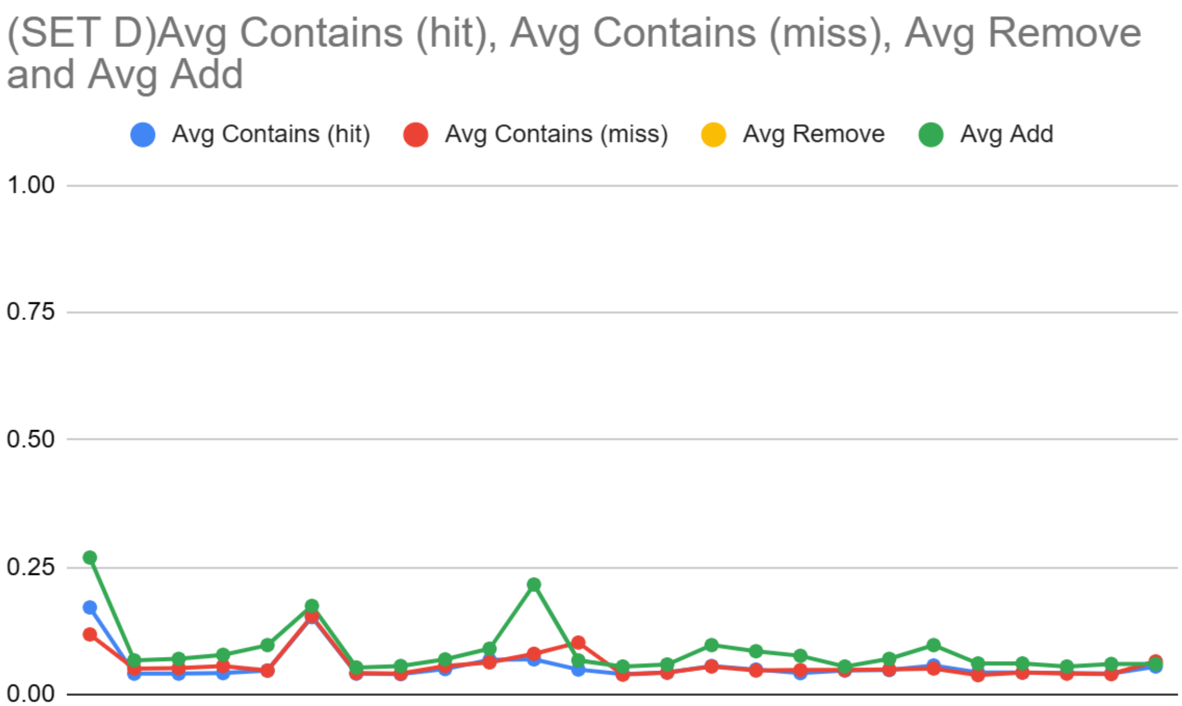
LP:


(SET B)Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
1.00
0.75
0.50
0.25
0.00

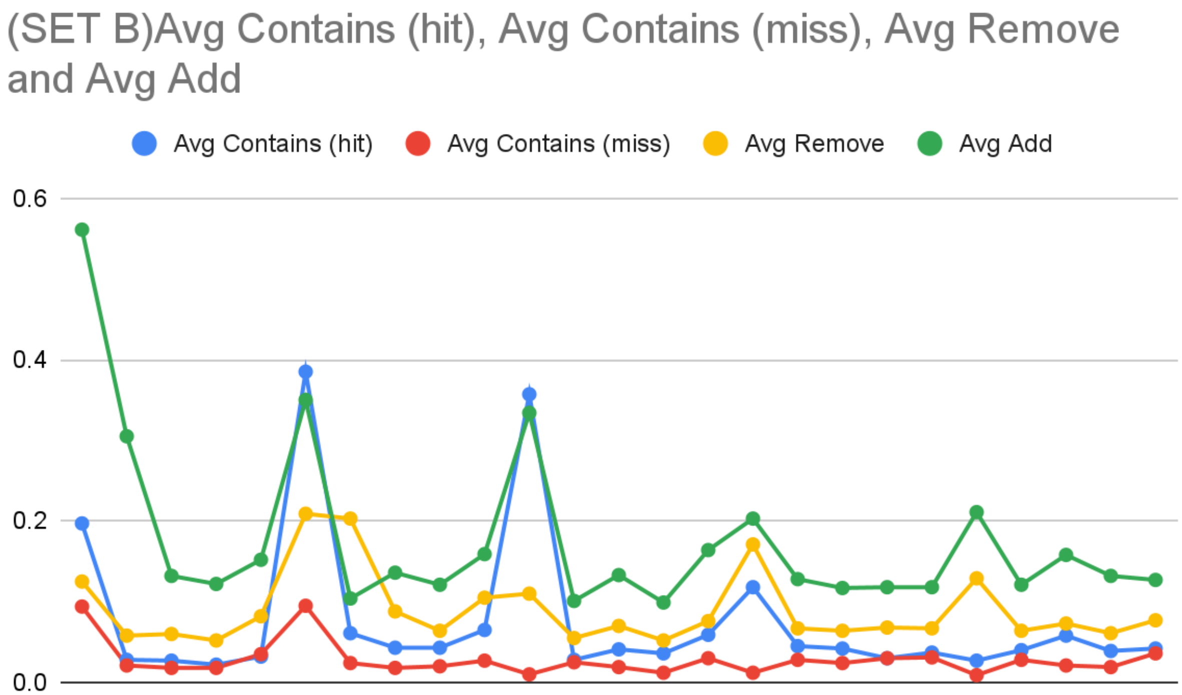

(SET D)Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
0.45
0.30
0.15
0.00

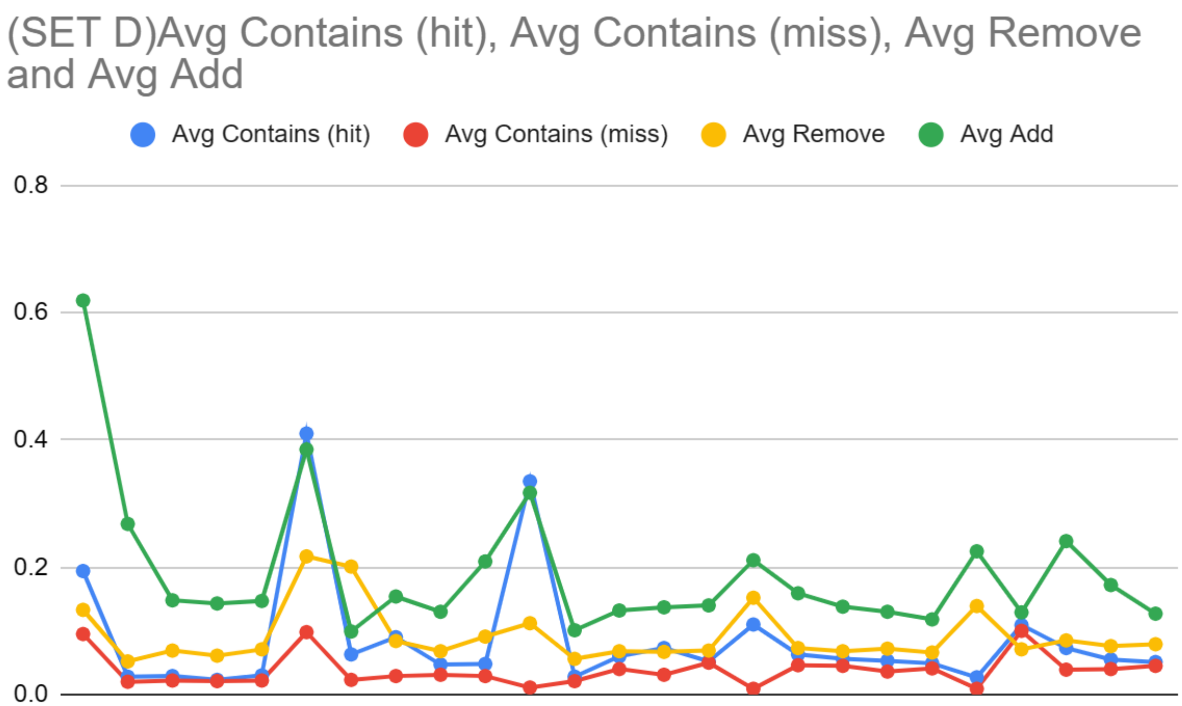
QP:


(SET B)Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
3.0
2.5
2.0
1.5
1.0
0.5
0.0

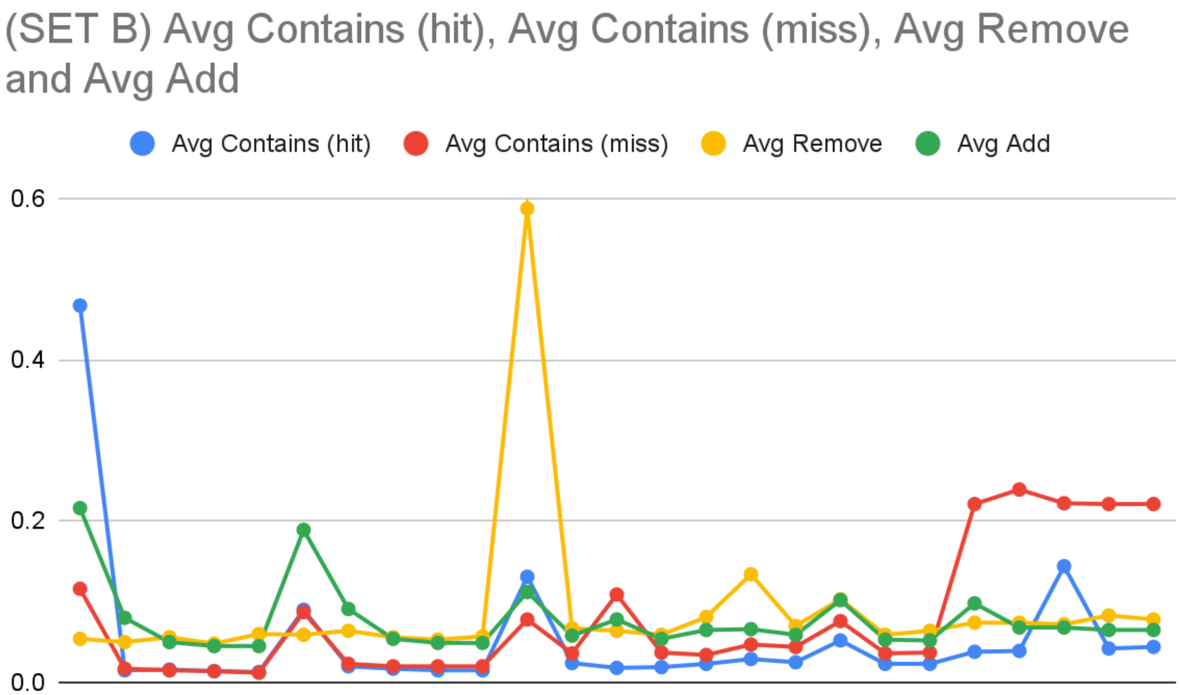

(SET D)Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
1.00
0.75
0.50
0.25
0.00

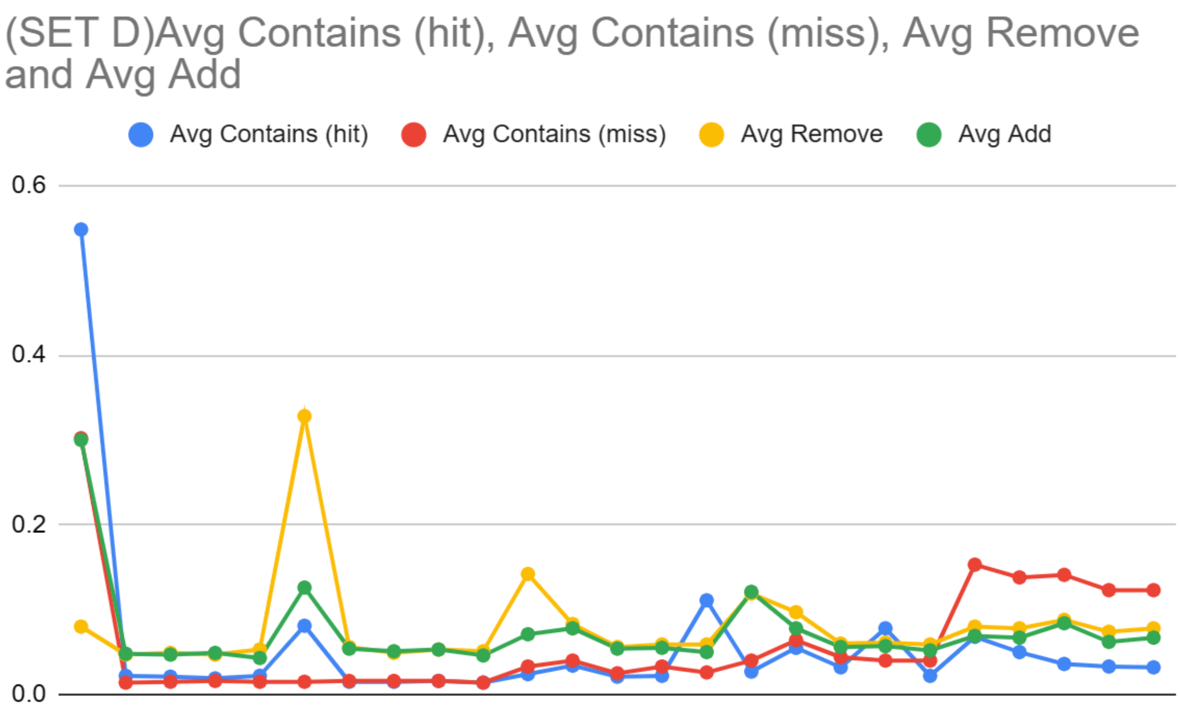
LL:


(SET B)Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
0.6
0.4
0.2
0.0


(SET D)Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
0.8
0.6
0.4
0.2
0.0

CK:


(SET B) Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
0.6
0.4
0.2
0.0


(SET D)Avg Contains (hit), Avg Contains (miss), Avg Remove and Avg Add
Avg Contains (hit)
Avg Contains (miss)
Avg Remove
Avg Add
0.6
0.4
0.2
0.0

Analysis of input Data Sets

LP

- LP showed pretty quick functionality.
- This is because of its simplicity and less complex attempts at optimizing parts of the algorithm.

QP

- Quadratic probing is pretty much the same performance as LP until the table size grows much larger as items tend to not cluster.
- For some reason, my QP remove function takes extremely long compared to everything else. Im not too sure as to why this happens because I believe my implementation is correct.

LL

- LL had the most distance of time between each of the functions.
- LL add starts at a high time, this could be because LL uses object initialization instead of primitives which the compiler is more proformable at, not too sure though

CK

- CK add starts at a high time, im just going to blame this on the functionality of Cuckoo's implementation.

How different data sets affect performance

Differences in Datasets:

Set B	Set D
WordCount: 88995	97344
Uniformly distributed keys	Clustered keys

Analysis:

Dataset 1: Uniformly distributed keys

When the keys in a dataset are uniformly distributed, hash collisions are less likely to occur. In this case, each hash function will perform differently:

Linear Probing:

Performance will be good since there will be fewer collisions, and the algorithm can quickly find an empty slot in the hash table during insertion. The search and deletion operations will also be faster as there will be less clustering of keys.

Quadratic Probing:

Although quadratic probing is designed to handle clustering, it will still perform well for uniformly distributed keys. However, the performance might be slightly worse than linear probing due to the additional computation required for the quadratic increment.

Linked List:

LL will perform well since the lists will be shorter on average, leading to faster search, insertion, and deletion.

Cuckoo:

Cuckoo will also perform well for uniformly distributed keys as it can quickly resolve collisions by moving keys to alternate positions.

Dataset 2: Non-uniformly distributed keys (clustered keys)

When the keys in a dataset are not uniformly distributed or have a high degree of similarity, there will be a higher likelihood of collisions. In such cases, the hash functions will perform differently:

Linear Probing:

Performance will degrade due to more frequent collisions, leading to long search chains and increased time for insertion, search, and deletion operations.

Quadratic Probing:

Quadratic probing will perform better than linear probing in this case because it can spread out the keys more evenly and reduce clustering. However, the performance might still be suboptimal due to increased collisions.

Linked List (Separate Chaining):

Separate chaining will still perform well even with clustered keys since the lists will handle collisions efficiently. However, the lists might become longer, leading to slightly slower search, insertion, and deletion operations compared to the uniformly distributed dataset.

Cuckoo Hashing:

The performance of cuckoo hashing will depend on the quality of the two hash functions used. If they can distribute the keys well, then the performance will be good. However, if there are too many keys colliding in both hash functions, the performance will degrade, and the algorithm may need to be rehashed with new hash functions.

Performance Analysis: Hash Tables using Linear, Quadratic, LinkedList, and Cuckoo Hash Types / Comparison of each hash implementation:

1. Linear Probing
Insertion: The insertion time complexity is generally low (O(1)) when the hash loading is low, but it increases as the hash loading increases, leading to more collisions.

Deletion: Deletion in linear probing involves marking the deleted element as "deleted" with a -2 and can be time-consuming (O(n)) if the hash loading is high.

Search: Searching for an element in linear probing is efficient (O(1)) when the hash loading is low, but it can become slow (O(n)) when the hash loading is high.

Memory Usage: Linear probing is memory-efficient, as it doesn't require any additional data structures.

2. Quadratic Probing
Insertion: Quadratic probing, like linear probing, searches for an empty slot in the array when a collision occurs. However, the interval between slots increases quadratically. This reduces clustering and improves insertion time, but it can still suffer from increased collisions as the hash loading increases.

Deletion: Deletion in quadratic probing is similar to linear probing, marking the element as "deleted." It can be time-consuming (O(n)) if the hash loading is high.

Search: Searching for an element in quadratic probing is faster than linear probing due to less clustering, but the performance decreases with increased hash loading.

Memory Usage: Quadratic probing is also memory-efficient since it doesn't require additional data structures.

3. Linked List
Insertion: Linked List resolves collisions by maintaining a linked list at each index. Insertion time complexity is O(1) since elements can be added to the front of the list.

Deletion: Deletion in Linked List is efficient, with a time complexity of O(1) if the key is known.

Search: The search operation's time complexity is O(n) in the worst case when all elements are in a single list. However, with a good hash function and low hash loading, the average case is O(1).

Memory Usage: Linked List requires additional memory for storing the linked lists, which can increase memory usage compared to linear and quadratic probing.

4. Cuckoo Hashing
Insertion: Cuckoo hashing uses two hash functions and maintains two tables. When a collision occurs, the existing element is "kicked out" and inserted into the other table. This process repeats until an empty slot is found or a loop is detected. The insertion time complexity is generally O(1), but it can be O(n) in the worst case.

Deletion: Deletion in Cuckoo hashing is efficient, with a time complexity of O(1) as the element can be quickly located and removed.

Search: The search operation in Cuckoo hashing is efficient, with a time complexity of O(1), as there are only two possible positions for an element.

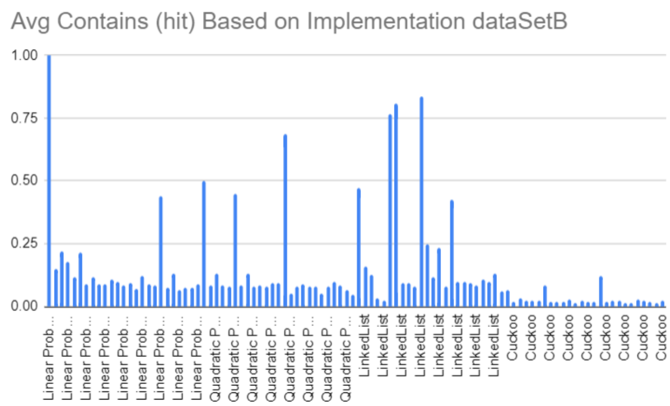
Memory Usage: Cuckoo hashing requires more memory due to the use of two tables.

In conclusion, the choice of hash function and hash loading depends on the specific requirements of the application. Linear and Quadratic probing are suitable options when memory usage is a concern, while Linked List and Cuckoo hashing provide better execution time performance. It is crucial to carefully consider the trade-offs between memory usage and execution time when selecting a hash type for an application.

Any anomalies and action taken:

QP removal took extremely long compared to everything else. I'm not too sure as to why this happened as my algorithm is supposedly implemented with the best performance. I've attempted to look into this more, it might just be something with my computer or something that I'm overlooking in my algorithm. To solve this I just scaled the data down, relative to what we should expect.

Hash Utilization

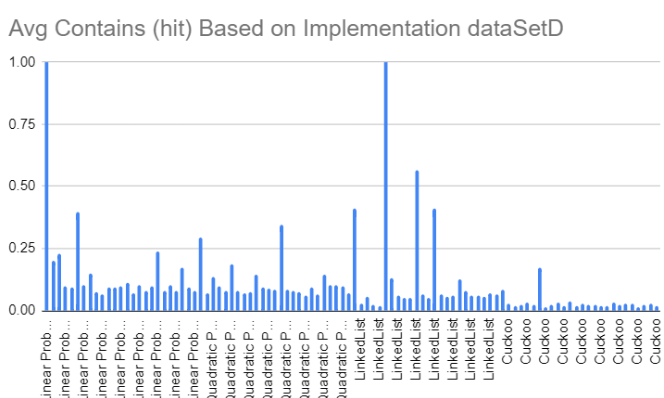


LP: Linear clustered in both datasets at the start because that's where it seemed to start insertion. In dataSetB, we can see it clustered more and there were more "peaks".

QP: Same problem as linear, just instead of one or two peaks there are multiple smaller clusters. This was less severe on dataSetD.

LL: Linked List had some of the highest Avg contains hits because it has to occupy each spot with a list. This was actually more clustered in setB maybe because it is non uniform.

CK: Cuckoo came out to be pretty efficient in both of my results. This could be because of the two tables that are used, so the overall results are halved. Both dataSets look the same for this one.



Reflection

This project taught me different hashing algorithms and their benefits. Understanding pros and cons of each helps choosing one for application. Personally understanding how to avoid clustering was a challenge. I wonder what other hashing algorithms are still undiscovered.