

CP2410 - Assignment 2

(Evaluating the efficiency and effectiveness of data structures and algorithms)

Objective

Efficient searching of databases can be done with the use of high performance data structures which are often used to efficiently perform the necessary operations in a timely manner. We can look at hash tables and trees as some of the most commonly used data structures suitable for this purpose. The objective of this assignment is to analyze these two forms of data structures, make comparisons and explain their significance in their real-world usage.

We evaluate the efficiency of these data structures under set conditions and further explore their implementation and uses.

Data structures to be analyzed

1. Probing Hash Table
2. AVL tree
3. Modified Probing Hash Table

PART 1 - Experimentations and Results

1. Probing Hash Table

We first look at the probing hash table and perform separate set of experiments to analyze its efficiency in terms of time complexity. To begin with, a hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty.

1.1 Database used:

We use a student database dictionary to insert and delete the items from the hash table. The database consists of 11 items with corresponding keys and values. The key denotes the student ID and the value indicates the corresponding name of the student. There a total of 11 items in this dictionary.

```
student_dict = {13368171: "Uday", 13354612: "Albert", 13345678: "Lisha",  
                13345789: "Sharath", 13345890: "Susan", 13345202: "Samantha", 13321111: "Francesca", 13321212: "Jin",  
                13312121: "Vinay", 13312456: "Sudheer", 13358717: "Sandeep"}
```

Figure 1 – Showing the database chosen for experimentation

Initially, the probing hash table is empty. This is denoted by the 'None' values in the table. The initial table size is set to its default value '11'. The hash value for each item is based on the hash function which uses a prime number p for MAD compression (Multiply-Add-and-Divide method).

1.2 What is a hash function?

The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and $m-1$, where m is the size of the table. This value is obtained based on a simple modulus calculation. For the probing hash table, the hash value is calculated by the following function.

```
def _hash_function(self, k):  
    return (hash(k)*self._scale + self._shift) % self._prime % len(self._table)
```

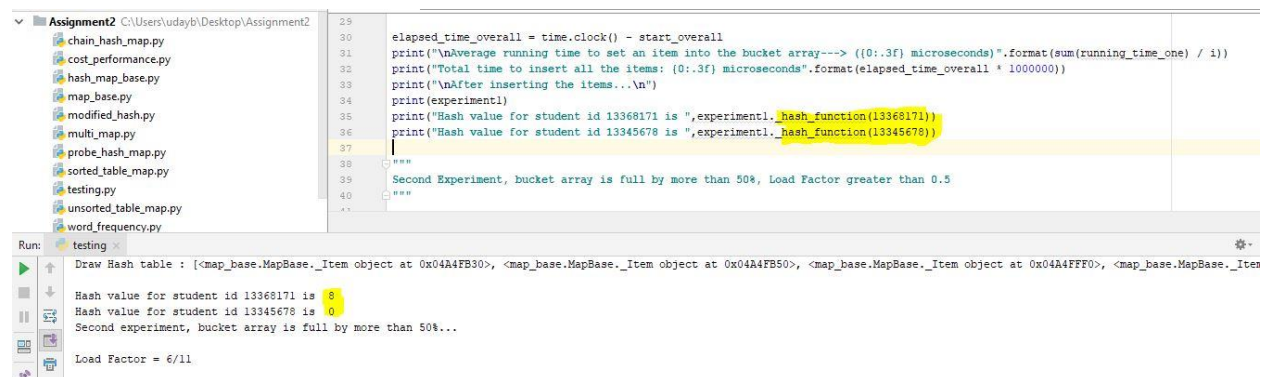
Figure 2 – Showing the hash function used to get the hash value

prime denotes the prime number $p = 109345121$

scale can be a value from 1 to $p-1$ for MAD compression

shift can be a value from 0 to $p-1$ for MAD compression

To find an item in the array, we can employ a brute force approach such as a linear search. But this would involve checking each item in turn. For a very big array, this could take a very long time. Suppose, if we want to retrieve a value from an element in the array and we know the index number, we can look up for the value very quickly. The time taken to look up for a value in an array is independent of the size of the array and its position, if we know the index number. But how do we know the index of the value? The answer is each index number can be calculated using the value itself. This can be done with the help of the hash function. The hash function calculates an index number based on the 'key' given. For example, we can run the hash function and check the output to know the hash value as shown below.



The screenshot shows a Python IDE with a file explorer on the left containing files like `chain_hash_map.py`, `cost_performance.py`, `hash_map_base.py`, `map_base.py`, `modified_hash.py`, `multi_map.py`, `probe_hash_map.py`, `sorted_table_map.py`, `testing.py`, `unsorted_table_map.py`, and `word_frequency.py`. The main editor displays a Python script with the following code:

```
25  
30 elapsed_time_overall = time.clock() - start_overall  
31 print("\nAverage running time to set an item into the bucket array--> {(0:.3f) microseconds}.format(sum(running_time_one) / i))  
32 print("\nTotal time to insert all the items: {(0:.3f) microseconds}.format(elapsed_time_overall * 1000000))  
33 print("\nAfter inserting the items...\n")  
34 print(experiment1)  
35 print("Hash value for student id 13368171 is ",experiment1._hash_function(13368171))  
36 print("Hash value for student id 13345678 is ",experiment1._hash_function(13345678))  
37  
38  
39 Second Experiment, bucket array is full by more than 50%, Load Factor greater than 0.5  
40  
41
```

The output window at the bottom shows the following results:

```
Run: testing  
Draw Hash table : [<map_base.MapBase._Item object at 0x04A4FB30>, <map_base.MapBase._Item object at 0x04A4FB50>, <map_base.MapBase._Item object at 0x04A4FFD0>, <map_base.MapBase._Item  
Hash value for student id 13368171 is 8  
Hash value for student id 13345678 is 0  
Second experiment, bucket array is full by more than 50%...  
Load Factor = 6/11
```

Figure 3 – Showing the hash values calculated for different keys

We can see that, the index number for the student id 13368171 is calculated as 8 and 13345678 is calculated as 0.

A hash table of key, value pairs is sometimes referred to as a hash map as it maps every key to its value. A hashing function is basically a calculation applied to a key, which may be a very large number or a very long string, to transform it into a relatively small index number that corresponds to its position in the hash table. There are several kinds of hashing algorithms we can choose from depending on the nature of data available. There are some problems that need to be addressed with the generation of hash values like collisions which will be covered in the later sections.

1.3 Load Factor:

Another factor we consider while performing our experiments is the load factor. Load factor is obtained by dividing the number of items inserted into the array by the total array size. Technically, we are looking at how much accessibility is there in the array to fill up the empty slots. In the case of probing hash table, the load factor is kept at 50%. If the size of entries into the bucket array is greater than half the size of the table, then the table size is increased to $(2 * x - 1)$ and all items are rehashed to this table (x is the size of the table). This is done to accommodate more number of entries into the array. As long as the load factor is $O(1)$, the core operations on the hash table will run in $O(1)$ expected time.

1.4 Operations:

Our experiments include measuring the time efficiency of these three operations.

1. Set item – inserting an item into the bucket array
2. Get item – Getting an item from the bucket array
3. Delete item – Deleting an item from the bucket array

We check the running times of these operations and analyse our findings.

1.5 Linear Probing:

The Probe hash map class is implemented using open addressing with linear probing. We already know that our bucket array is implemented as a python list using the hash map base class. We also talked about collisions when generating the hash values. A collision occurs when two items/values get the same slot/index, i.e. the hashing function generates same slot number for multiple items. In an open addressing system, all the data must go inside the table. If collision occurs, alternative cells are tried until an empty cell is found.

Simply put,

Linear Probing is a strategy for resolving collisions of keys that map to the same index in a hash table.

1. Use a hash function to find the index for a key
2. If that slot contains a value, use the next available slot, "a higher index". If you reach the end of the array, go back to the front.

Another key thing to look at is the "is_available" method in Probe Hash map class. In order to support deletions, a special marker is placed in a table location at which an item has been deleted, so that we can distinguish between it and a location that has always been empty. When attempting to retrieve the value associated with a given key, we must continue probing until we find the key, or until we reach a table slot with the None value. We cannot stop the search upon reaching an AVAIL sentinel, because it represents a location that may have been filled when the desired item was once inserted.

```
def _is_available(self, j):  
    """Return True if index j is available in table."""  
    return self._table[j] is None or self._table[j] is ProbeHashMap._AVAIL
```

Figure 4 – Showing the is_available method that creates a sentinel AVAIL

1.6 Experiments on inserting the items from the student dictionary

1.6.1 - Experiment 1

Testing conditions:

Empty bucket array with a load factor of 0/11

Initial number of items in the array = 0

Size of the table = 11

We calculate the running times for setting the items from the student dictionary. The results are shown in the figure below.

```
Run: testing x
Experiment 1 (Empty bucket array: )

Number of entries in the map: 0
Size of the table: 11
Draw Hash table : [None, None, None, None, None, None, None, None, None, None, None]

Efficiency to set item 0- running time (5.136 microseconds)
Efficiency to set item 1- running time (3.556 microseconds)
Efficiency to set item 2- running time (2.370 microseconds)
Efficiency to set item 3- running time (1.975 microseconds)
Efficiency to set item 4- running time (1.975 microseconds)
Efficiency to set item 5- running time (1.975 microseconds)
Efficiency to set item 6- running time (1.975 microseconds)
Efficiency to set item 7- running time (1.975 microseconds)
Efficiency to set item 8- running time (1.975 microseconds)
Efficiency to set item 9- running time (1.975 microseconds)
Efficiency to set item 10- running time (1.975 microseconds)

Average running time to set an item into the bucket array---> (2.442 microseconds)
Total time to insert all the items: 126.815 microseconds

After inserting the items...

Number of entries in the map: 11
Size of the table: 11
Draw Hash table : [<map_base.MapBase._Item object at 0x02C8FB30>, <map_base.MapBase._Item object at 0x02C8FB50>,
```

Figure 5 – Showing the running time for inserting an item

From the figure above, we can see that the average time taken to insert the items into the empty hash table is very less, 2.442 micro seconds. Also, the total time taken to insert all the items is approx. 126.815 micro seconds. We can also notice the initial hash table which is represented as a python list with all “None” values.

1.6.2 - Experiment 2

Initial testing conditions:

More than half full bucket array with a load factor of 6/11

Initial number of items in the array = 6

Size of the table = 11

After insertion:

Number of items in the array = 17

Size of the table = 21

```

Second experiment, bucket array is full by more than 50%...

Load Factor = 6/11

Efficiency to set item 0- running time (1.580 microseconds)
Efficiency to set item 1- running time (3.556 microseconds)
Efficiency to set item 2- running time (3.160 microseconds)
Efficiency to set item 3- running time (4.346 microseconds)
Efficiency to set item 4- running time (3.951 microseconds)
Efficiency to set item 5- running time (3.951 microseconds)
Efficiency to set item 6- running time (3.951 microseconds)
Efficiency to set item 7- running time (3.951 microseconds)
Efficiency to set item 8- running time (4.346 microseconds)
Efficiency to set item 9- running time (4.741 microseconds)
Efficiency to set item 10- running time (4.346 microseconds)

Average running time to set an item into the bucket array----> (3.807 microseconds)

Efficiency to set overall time: 129.580 microseconds

After inserting the items...

Number of entries in the map: 17
Size of the table: 21
Draw Hash table : [<map_base.MapBase._Item object at 0x05278AF0>, <map_base.MapBase._Item object at 0x05278B10>,

```

Figure 6 – Showing the running time for inserting an item

From the figure above, we can notice, that the size of the table is increased to 21 slots to accommodate the 11 entries from the student dictionary. The total number of entries is 17. We can notice that there is a marginal increase in the time taken to insert each item (average = 3.807 micro seconds). This can be attributed to the fact that with an increase in the load factor, a new array of larger size was created and all the elements rehashed individually before insertion. The total taken for the whole process also shows a marginal increase. If we look at a real-world scenario with very large databases, the time difference can be a significant attribute to consider.

With the given code for the probing hash table, the hash value must be recomputed each time a new array is created further increasing the time complexity. However, it needs to be noted from both the experiments, that the time taken to set an item into the bucket array is relatively small. We only calculate the hash value to determine the index of an item and then perform insertion into the identified slot. We can also say it is of constant time or on average $O(1)$. In some cases, where an array is full and we must traverse its whole length, the time can be $O(n)$ as well.

1.6.3 - Experiment 3

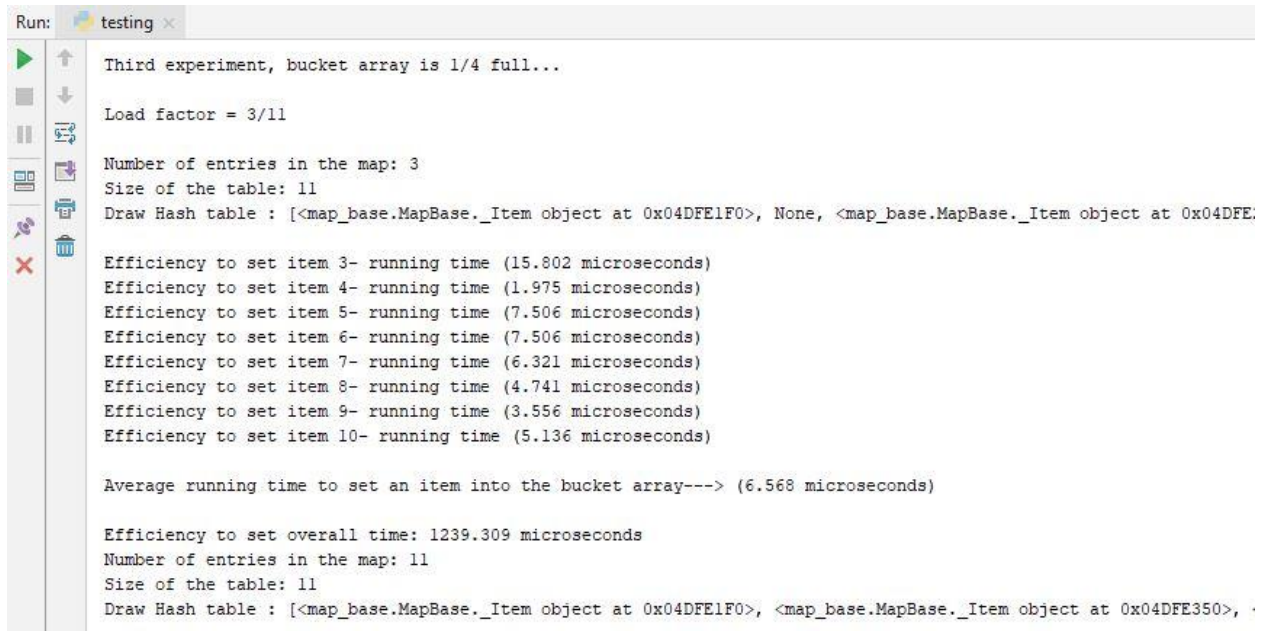
Initial testing conditions:

Bucket array is one fourth full, load factor of 3/11

Initial number of items in the array = 3

Size of the table = 11

We now consider a condition where the bucket array is one-fourth full. The probing hash code does not allow for a resize of the bucket array if the load factor is less than 0.5. So, we must draw only the allowed number of entries to fill in the bucket array based on linear probing.



```
Run: testing x
Third experiment, bucket array is 1/4 full...

Load factor = 3/11

Number of entries in the map: 3
Size of the table: 11
Draw Hash table : [<map_base.MapBase._Item object at 0x04DFE1F0>, None, <map_base.MapBase._Item object at 0x04DFE...

Efficiency to set item 3- running time (15.802 microseconds)
Efficiency to set item 4- running time (1.975 microseconds)
Efficiency to set item 5- running time (7.506 microseconds)
Efficiency to set item 6- running time (7.506 microseconds)
Efficiency to set item 7- running time (6.321 microseconds)
Efficiency to set item 8- running time (4.741 microseconds)
Efficiency to set item 9- running time (3.556 microseconds)
Efficiency to set item 10- running time (5.136 microseconds)

Average running time to set an item into the bucket array---> (6.568 microseconds)

Efficiency to set overall time: 1239.309 microseconds
Number of entries in the map: 11
Size of the table: 11
Draw Hash table : [<map_base.MapBase._Item object at 0x04DFE1F0>, <map_base.MapBase._Item object at 0x04DFE350>, .
```

Figure 7 – Showing the running time for inserting an item

From the figure above, we can see that there is a relatively higher increase in time taken to set each item in this array. The overall time taken to perform the whole process is also significantly higher (1239.309 micro seconds).

1.6.4 - Experiment 4

Initial testing conditions:

Bucket array contains a few entries which are available in the student dictionary

Initial number of items in the array = 3

Size of the table = 11

After insertion:

Number of items in the array = 10

Size of the table = 11


```
Run: testing x
Size of the table: 11
Draw Hash table : [<map_base.MapBase._Item object at 0x05C5E150>, <map_base.MapBase._
Fourth experiment, Same entries
Load Factor = 3/11
Number of entries in the map: 3
Size of the table: 11
Draw Hash table : [None, None, None, None, <map_base.MapBase._Item object at 0x05C5E2
Efficiency to set item 3- running time (3.556 microseconds)
Efficiency to set item 4- running time (5.531 microseconds)
Efficiency to set item 5- running time (6.321 microseconds)
Efficiency to set item 6- running time (6.321 microseconds)
Efficiency to set item 7- running time (6.321 microseconds)
Efficiency to set item 8- running time (6.321 microseconds)
Efficiency to set item 9- running time (5.926 microseconds)
Efficiency to set item 10- running time (6.716 microseconds)
Average running time to set an item into the bucket array---> (5.877 microseconds)
Efficiency to set overall time: 146.963 microseconds
After insertion...
Number of entries in the map: 11
Size of the table: 11
Draw Hash table : [<map_base.MapBase._Item object at 0x05C5E3B0>, <map_base.MapBase._
```

Figure 8 – Showing the running time for inserting an item

In this scenario, we look at the bucket which already contains three items which have the same key, value as in the student dictionary. As new entries are inserted into the bucket array, a new hash value is computed and the existing value will be replaced by a similar value from the dictionary for the same key. The average time taken for this whole process is marginally higher as can be noted from the figure above.

Analysis:

From the above experiments it can be clearly analysed that the time taken to insert an item into the bucket array using linear probing is most of time constant and can be represented as $O(1)$. However, in some cases where we must traverse the whole length of the array to insert the item at the last index, the time complexity can be represented using $O(n)$, where n is the size of the array.

1.7 - Summarizing collision resolution

Open addressing uses probing, has clustering issues as it gets full but still has reasons for its use:

- Easier data representation
- Less memory allocation
- Run-time overhead for list nodes (but an array implementation could be faster)

1.8 - A word on hashing

The hash table is one of the most important data structures

- Efficient find, insert, and delete
- Operations based on sorted order are not so efficient
- Useful in many, many real-world applications
- Popular topic for job interview questions

Important to use a good hash function

- Good distribution of key hashes
- Not overly expensive to calculate (bit shifts good!)

Important to keep hash table at a good size

- Keep Table Size a prime number
- Set a preferable λ depending on type of hash table

1.9 - Experiments on searching/retrieving items from the probing hash table

We now conduct experiments to analyse the efficiency of the probing hash map when retrieving an item.

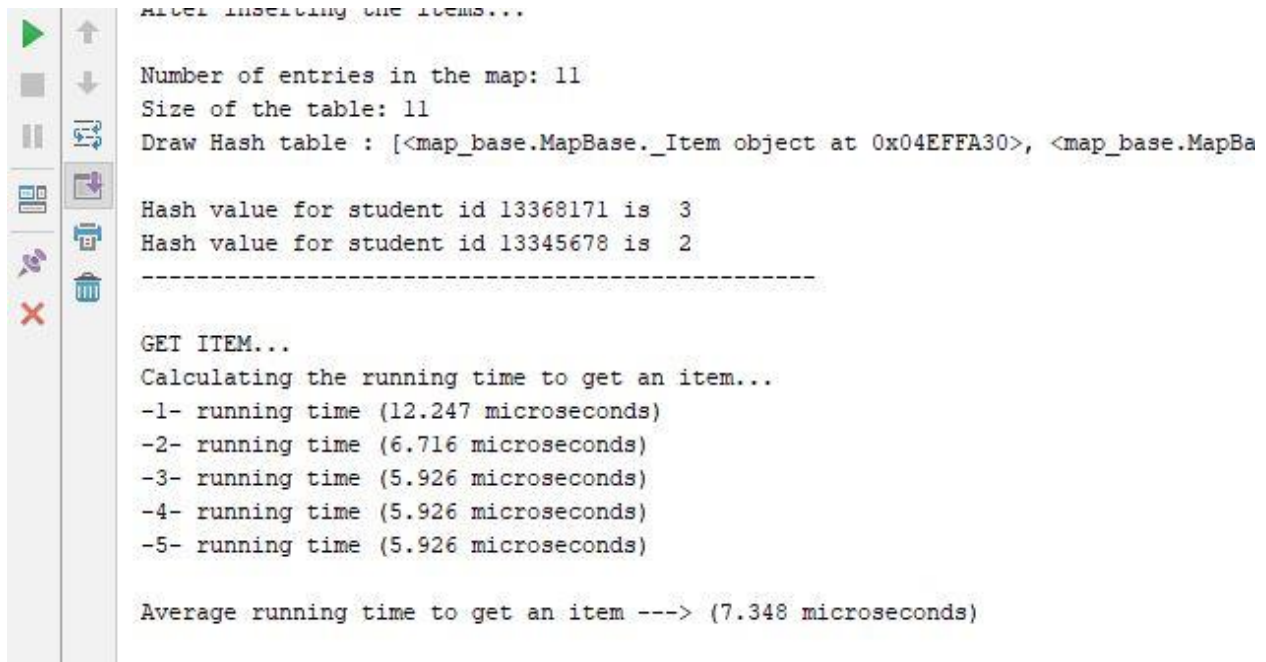
Initially, the probing hash table is empty with all None values. We try to run the code to get an item and get a 'Key Error'. Based on the experiments we conducted for inserting an item into the array, we also look at the running time to get the value from a specified index.

1.9.1 - Experiment 1:

Size of the array = 11

Array is full.

We conduct this analysis based on the first experiment we previously conducted for inserting an item. The resultant array is filled with all the values from the student dictionary. We now try to retrieve an item from this array and calculate the average running times.



```
After inserting the items...

Number of entries in the map: 11
Size of the table: 11
Draw Hash table : [<map_base.MapBase._Item object at 0x04EFFA30>, <map_base.MapBa

Hash value for student id 13368171 is 3
Hash value for student id 13345678 is 2
-----

GET ITEM...
Calculating the running time to get an item...
-1- running time (12.247 microseconds)
-2- running time (6.716 microseconds)
-3- running time (5.926 microseconds)
-4- running time (5.926 microseconds)
-5- running time (5.926 microseconds)

Average running time to get an item ---> (7.348 microseconds)
```

Figure 9 – Showing the running times to get an item

As we can see, the average running time to get an item from this bucket array is 7.348 micro seconds, which is slightly higher than the time taken to insert an item (2.442 micro seconds). Still, this value is significantly lower and we can analyse that we can retrieve a value from an index of a bucket array in very short time. Technically, if we know the index number of the array, we get the value in a constant time represented by $O(1)$. As mentioned above, the index number can be calculated by simply using the hash function. This mechanism is extremely useful in case of very large arrays.

No matter the length of the array, if we know the index number, we can quickly retrieve any data from the hash table. We can reconfirm this fact by searching for an item from an array of double the size. For this purpose, we use the second experiment as a basis to conduct further experimentation of the get item functionality.

1.9.2 - Experiment 2:

Size of the table = 21

Number of slots filled = 17

As we can see, we are trying to retrieve an item from a bucket array of size 21 and about 80% full. The results can be observed below.

```
Run: testing x
Number of entries in the map: 17
Size of the table: 21
Draw Hash table : [<map_base.MapBase._Item object at 0x056D9AF0>, <map_bas

-----

GET ITEM...
Calculating the running time to get an item...
-1- running time (10.667 microseconds)
-2- running time (6.321 microseconds)
-3- running time (6.321 microseconds)
-4- running time (7.506 microseconds)
-5- running time (5.926 microseconds)

Average running time to get an item ---> (7.348 microseconds)
```

Figure 10 – Showing the running times to get an item

The average running time is almost the same as from the previous experiment with a lesser size array. This indicates that getting an item from a bucket array is independent of the size of the array in a linear probing hash table. And it takes a constant time of $O(1)$ to retrieve the item irrespective of its location.

1.10 - Experiments on deleting the item

We also look at the third operation to delete an item from the array and calculate the time efficiency of the probing hash table. We use the first experiment as a basis for conducting this test.

```
Run: testing x
Hash value for student id 13345678 is 5
-----

GET ITEM...
Calculating the running time to get an item...
-1- running time (7.901 microseconds)
-2- running time (5.531 microseconds)
-3- running time (5.136 microseconds)
-4- running time (5.136 microseconds)
-5- running time (5.136 microseconds)

Average running time to get an item ---> (5.768 microseconds)

-----

DELETE ITEM...
Calculating the running time to delete an item...
Running time to delete an item : 3.160 microseconds
-----
```

Figure 11 – Showing the running times to delete an item

As can be observed from the figure above, the time taken to delete a value from the bucket array is very less. Even in comparison to the get item, it is marginally faster. This is another key factor which signifies the importance of having a probing hash table. We can retrieve and delete an item from the hash table with a time complexity of $O(1)$. Deleting also follows the same principle as retrieving. Irrespective of the size of the array, if we know the index of the item we can delete or retrieve it in a very quick time. The efficiency of these operations has remained constant irrespective of the size of the array.

1.11 - Time efficiency as per the experiments conducted:

Pros

Efficiency of the operations

1. Insertion: $O(1)$, however worst case can be $O(n)$
2. Searching: $O(1)$
3. Deleting: $O(1)$

This is the theoretical minimum access time. Actual time is the (constant) time of the hashing function and the length of the probe.

Usage of a simple hashing function which doesn't take too much CPU cycles

Cons

Uses more memory than is actually required to maintain efficiency.

Usage of a complex (multiplication method) would have produce more random distribution values in the index (at the expense of CPU cycles of course)

As the probe length depends on the ratio of number of items on the table and the size of the table (i.e. load factor), during design every step has been taken to reduce performance degrading due to collisions considering the resource vs. efficiency factors

Unsuccessful search will take more time than a successful search since the whole index has to be probed. But this isn't a problem since unsuccessful searches are occasional.

2. AVL Tree

In this section, we analyze the time complexity of AVL tree for different operations. We will base our experiments on the given code and try to find the efficiency of using an AVL tree structure for search, insert and delete operations.

2.1 - What is an AVL Tree?

Before we understand what an AVL tree is, we must become familiar with the concepts of binary tree data structures. In a binary tree, a node can have a maximum of two children. BST is a special type of binary tree in which left child of a node has value less than the parent and right child has value greater than parent. AVL tree is binary search tree with additional property that the difference between height of left sub-tree and right sub-tree of any node can't be more than 1. In other words, a perfectly balanced tree is called an AVL tree. The basic operations of an AVL tree generally involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the sub trees.

2.2 - Why AVL trees?

Most of the BST operations (e.g., search, max, min, insert, delete) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of an AVL tree is always $O(\log n)$ where n is the number of nodes in the tree.

2.3 - Operations:

Our experiments include measuring the time efficiency of these three operations.

1. Set item – inserting an item into the bucket array
2. Get item – Getting an item from the bucket array
3. Delete item – Deleting an item from the bucket array

We check the running times of these operations and analyse our findings.

2.4 - Experiments

2.4.1 - Experiment 1 – Inserting items into an empty AVL Tree

In this experiment, we start out with an initial AVL tree with no nodes. We had used the student dictionary to analyse the time efficiency of the probing hash table. We use the same database to test on the AVL tree code and analyse its time complexity with regards to inserting, searching and deleting an item. The experimental results are analysed below.

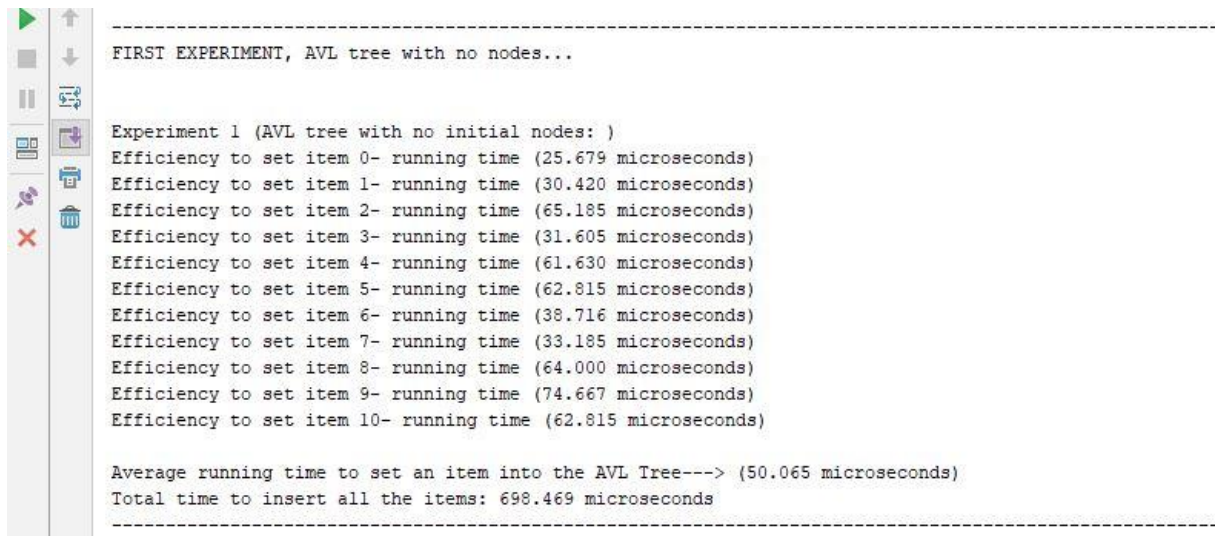


Figure 12 – Showing the running times to insert an item into the AVL tree

From the figure above, we can clearly observe that the time taken to insert an item into the AVL tree is significantly higher than the probing hash table. The average time taken here is 50.065 micro seconds, as compared to 2.442 micro seconds we found for the hash table. The overall time taken to complete the process is also much higher (698.469 micro seconds). The reasons for this can be many.

As we know, an AVL tree is a self-balancing data structure. With every insertion, the tree balances itself. The rotation operations for balancing takes a constant time as only a few pointers are being changed. Updating the height and getting the balance factor also takes constant time. So, the time complexity of AVL insert remains same as any binary search tree insert which is $O(h)$ where h is the height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So, time complexity of AVL insert is $O(\log n)$. As compared to the time complexity of a hash table which is only $O(1)$, this is significantly higher

The basic steps involved in insertion are explained below.

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is **0 or 1 or -1** then go for next operation.

Step 4: If the Balance Factor of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

2.4.2 - Experiment 2 – Insertion, AVL tree with three nodes

We now look at another case with an AVL tree with three nodes and run the same experiments. By doing this, we analyse the addition of nodes to an already existing tree.

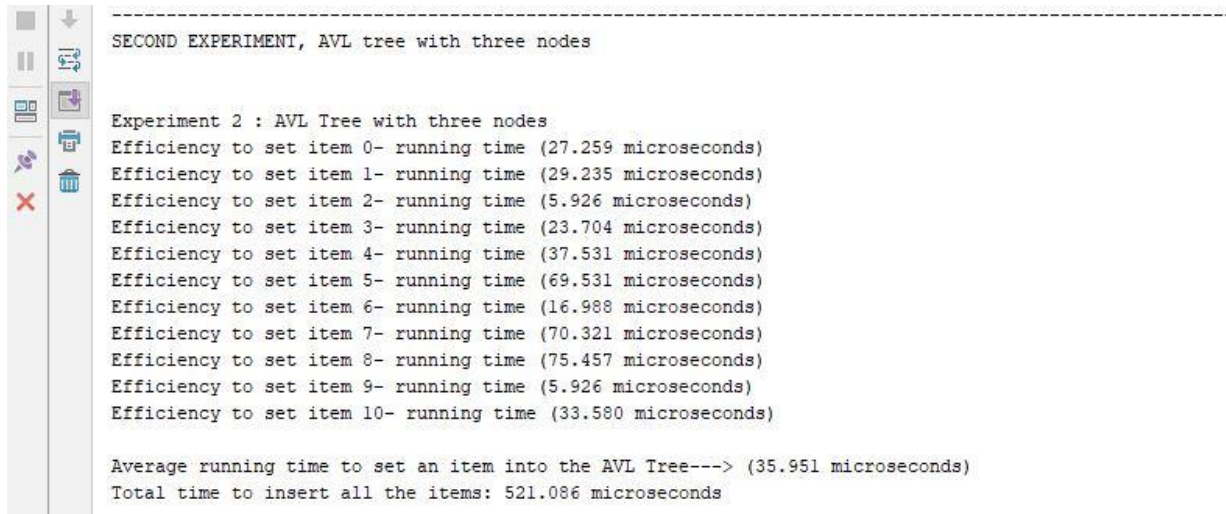


Figure 13 – Showing the running times to insert an item into an AVL tree

As we can see, the time complexity for this operation is almost the same as experiment 1. As more number of items are added to the tree, it balances itself so that the differences between heights of left and right subtrees for every node is less than or equal to 1.

2.4.3 - Experiment 3 – Searching for an item in the AVL Tree

We now look at the next operation – Search. For this purpose, we test the code and try to find out the time taken to get an item from the AVL tree. We use the first experiment as a basis to perform this operation. The results are detailed below.

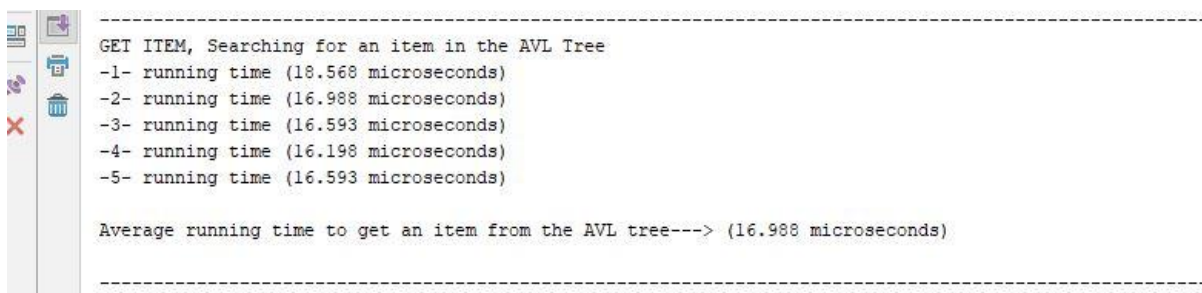


Figure 14 – Showing the running times to get an item from the AVL tree

As we can see, the time taken to search for an item in an AVL tree is 16.988 micro seconds which is higher than the time taken to search the probing hash table (7.348 micro seconds). The

search operation also follows a similar pattern to the insert operation for the AVL tree with a $O(\log n)$ time complexity.

2.4.4 - Experiment 4 – Deleting an item from the AVL tree

We also look at the delete operation that can be performed in an AVL tree. We use the same test code as the above case for this purpose.

```
DELETE ITEM, Searching for an item in the AVL Tree
Running time to delete an item : 38.716 microseconds
```

Figure 15 – Showing the running times to delete an item from the AVL tree

As in case of the get item, the delete operation also takes a longer time in comparison with the hash table. This goes in accordance with the analysis that the delete operation takes $O(\log n)$ time complexity to perform its operation on the AVL tree data structure.

2.4.5 - Analysis:

Based on the experiments conducted, we can analyze the performance of various operations on the AVL tree as below.

Operation	Complexity	Case
Insertion	$O(\log n)$	For Avg and Worst case
Deletion	$O(\log n)$	For Avg and Worst case
Search	$O(\log n)$	For Avg and Worst case

Table 1 – Showing the comparisons between the time complexity of the different operations on AVL tree

The AVL tree algorithms of the operations are more complex than the hash tables. The main algorithmic complexity comes due to the rebalancing requirement of the AVL tree. This makes understanding and coding bit difficult.

2.5 - Comparison of Probing hash table and AVL tree

We now compare the time taken by the probing hash table and the AVL tree to perform various operations.

2.5.1 - Insert item:

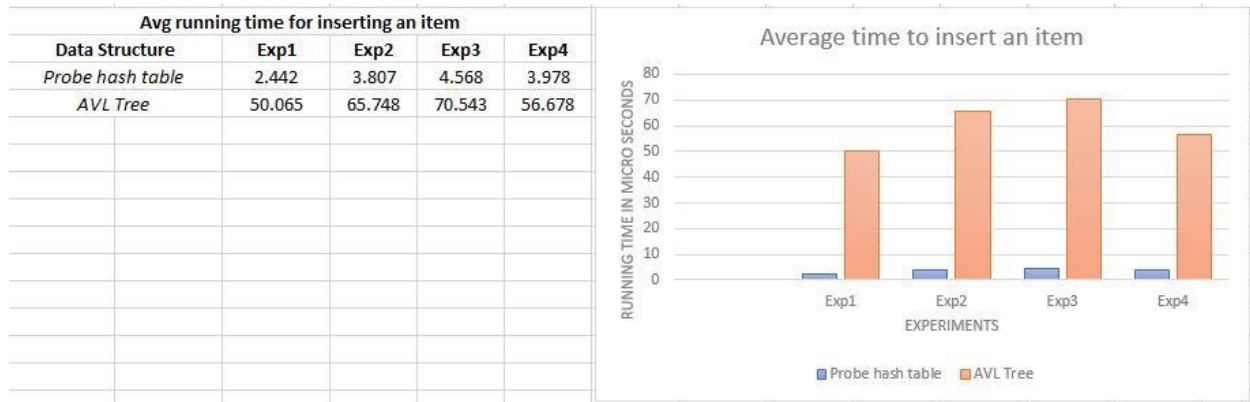


Figure 16 – Comparing insertion between Probe hash map and AVL tree

We can clearly observe from the above graph that the hash table outperforms the AVL tree in terms of the inserting an item.

2.5.2 - Search item:

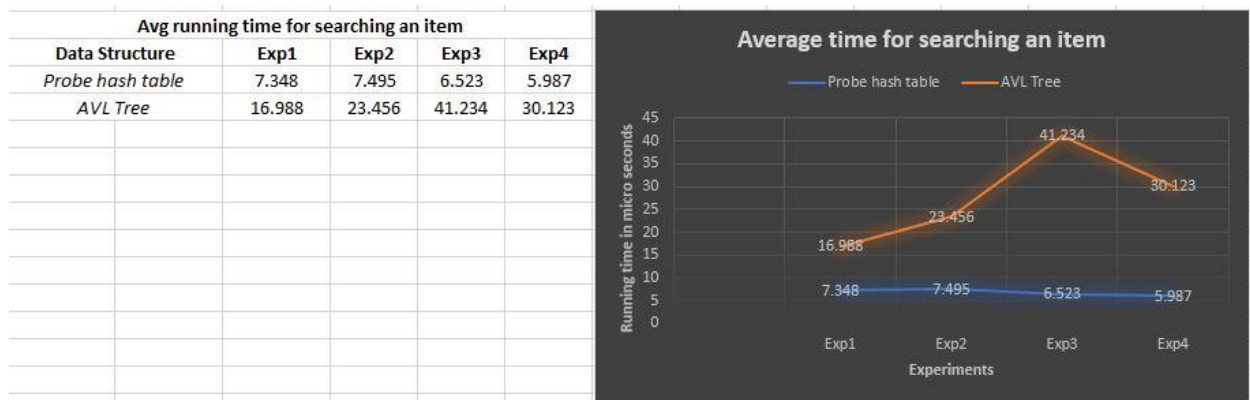


Figure 17 – Comparing search between Probe hash map and AVL tree

We can observe that the searching an item in a hash table is quicker than searching in the AVL tree.

2.5.3 - Delete item:

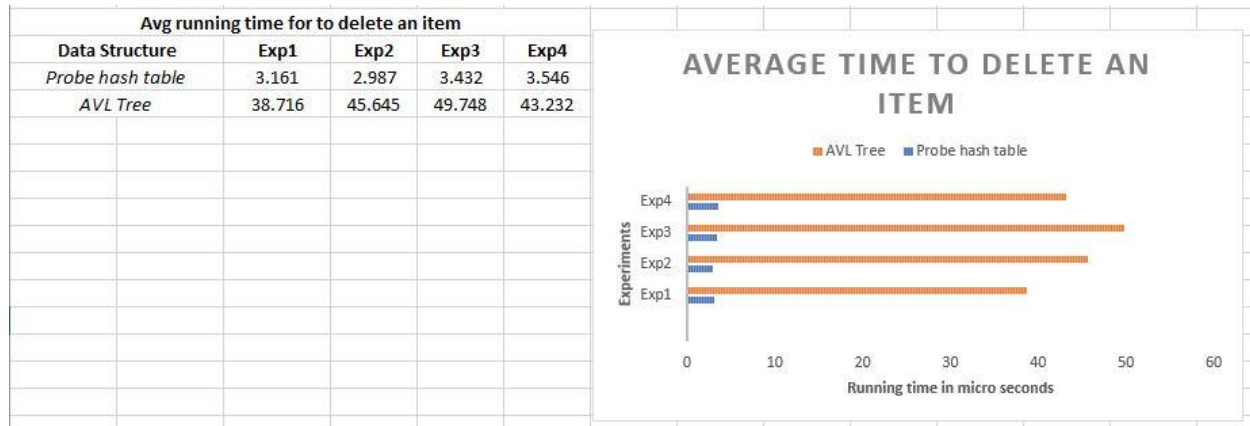


Figure 18 – Comparing deletion between Probe hash map and AVL tree

The average time to delete an item from a AVL tree is significantly higher than the time taken for the hash table.

3. Modified Hash Table

In our hash table implementations, we compute the hash code when first inserting an item, and re-compute each item's hash code each time we resize our table. The hash code is computed once, when an item is inserted, and the hash code is stored as an extra field of the item composite, so that it need not be recomputed.

In the code, the hash key is stored as a variable 'hashed_key'. The maximum load factor is set at 0.75. We use the same database student dictionary to perform the experiments and check the time efficiency of the modified hash table for the three operations, Insert, Delete and Search.

3.1 - Experiment 1 – Inserting an item into the modified hash table

In the modified hash table, in case of a resize, the already saved hash value is expected to alter the time taken to perform the whole operation. We run the experiment and compare the results with the original hash table.

```
C:\Python34\python.exe C:/Users/udayb/Desktop/Assignment2/testing_modified_hash_table.py
Experiments...
-----
FIRST EXPERIMENT (Insert an item )
Efficiency to set item 0- running time (1.580 microseconds)
Efficiency to set item 1- running time (2.370 microseconds)
Efficiency to set item 2- running time (15.407 microseconds)
Efficiency to set item 3- running time (1.975 microseconds)
Efficiency to set item 4- running time (1.975 microseconds)
Efficiency to set item 5- running time (2.765 microseconds)
Efficiency to set item 6- running time (1.580 microseconds)
Efficiency to set item 7- running time (1.975 microseconds)
Efficiency to set item 8- running time (4.346 microseconds)
Efficiency to set item 9- running time (25.284 microseconds)
Efficiency to set item 10- running time (1.580 microseconds)

Average running time to set an item into the bucket array---> (5.531 microseconds)
Total time to insert all the items: 159.210 microseconds
-----
```

Figure 19 – Showing the running times for inserting an item into the modified hash table

We can observe that the time taken to perform the insert operation into the modified hash table is also lower (5.531 micro seconds). However, the probing hash table delivered a better result of 3.807 micro seconds.

3.2 - Experiment 2 – Getting an item from the modified hash table

```
SECOND EXPERIMENT, GET an item from the modified hash table
-----
GET ITEM...
Calculating the running time to get an item...
-1- running time (3.160 microseconds)
-2- running time (1.580 microseconds)
-3- running time (1.185 microseconds)
-4- running time (1.185 microseconds)
-5- running time (1.580 microseconds)

Average running time to get an item ---> (1.738 microseconds)
```

Figure 20 – Showing the running times for retrieving an item from the modified hash table

As can be observed, searching for an item is more efficient in the modified hash table. The time taken, 1.738 micro seconds, is much lower than the probing hash table (7.348 micro seconds)

3.3 - Experiment 3 – Deleting an item from the modified hash table

```

Average running time to get an item ---> (1.738 microseconds)

-----

DELETE ITEM...
Calculating the running time to delete an item from the modified hash table

Running time to delete an item : 1.580 microseconds

```

Figure 21 – Showing the running times for deleting an item from the modified hash table

The running time to delete an item from the modified hash table, 1.580 micro seconds, is also slightly lower than the probing hash table (3.161 micro seconds).

3.4 - Analysis of the modified hash table:

From the above experiments, we can clearly analyze that the modified hash table performs marginally better than the probing hash table in terms of time complexity.

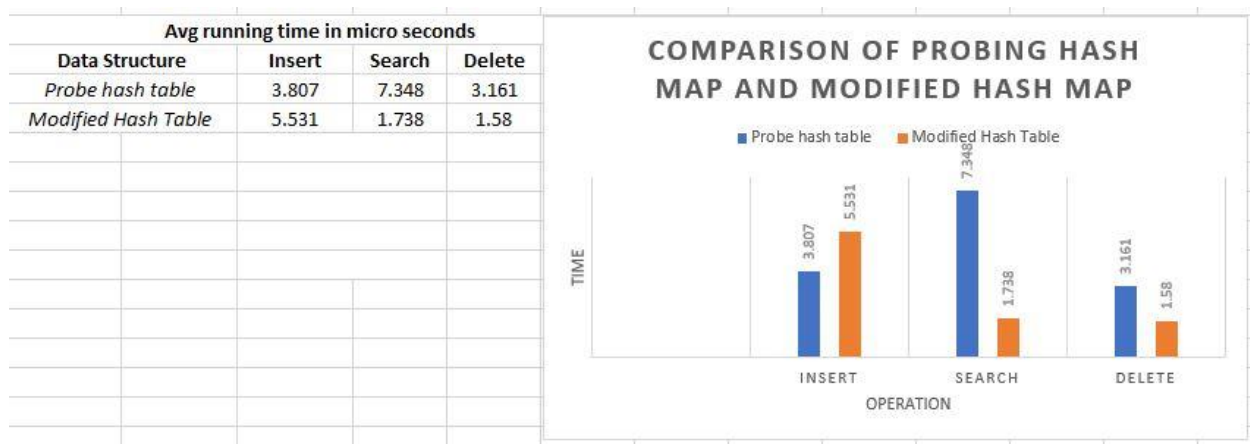


Figure 22 – Comparing the running times between probing hash map and modified hash map

PART II. Research into real world databases and their implementation issues:

As our daily dependence on applications grows, our expectations for those applications also grow. We want applications to be always up, bug free, easy to use, secure, and have high performance.

It's a simple relationship: data performance powers application performance, which in turn powers business performance. And just like there are a growing number of applications that

process data, there are an equally growing number of ways to store the data. How we store and retrieve that data matters.

To get peak performance, it is important to understand the differences between storage engines. We have already analyzed the student database that we implemented to test the two types of data structures, Probing hash tables and AVL trees. For the database chosen, the probing hash table outperformed the AVL tree and delivered optimal results in relatively quicker time. But different types of data may require different storage mechanism based on their implementation and usage.

4.1 - Who uses data structures?

There are several real-world examples we can consider while discussing the usage of data structures. Software engineers at the top tier software companies use data structures and algorithms in their application development. It's rare that we will need to implement something standard and complex from scratch. However, we need to know of the algorithm and data structure existence, so that it can be factored into our final design.

For example, suppose I'm writing a web service which handles transactions from a front-end web service. I need to have local queues - possibly priority queues - to prioritize the transaction requests I get. I'll need locks and locking for any local data structures. I'll need reliable consensus algorithms (or libraries which implement these) to make persisted data reliable in the face of storage failures. I'll need to understand the scalability and computational complexity of my service and algorithms (in terms of big-O) to know when I need to start throttling my input.

Constructs like hash sets, maps, multi-maps, arrays, lists are used constantly. For specialized services we wind up digging deep into other domains, such as distributed hash tables for peer-to-peer services, zero-knowledge proofs for security, encryption choices for authentication and authorization. connected components analysis for touch tracking, machine learning for gesture recognition, and so on.

4.2 - Why should we know how a data structure is implemented?

In order to make good judgements about what data structures and algorithms to use when faced with a particular problem, we must have a basic understanding on how these data structures are implemented. A common case is that of sorting algorithms. If we know that Merge Sort and Heap Sort are faster than Insertion Sort for the average case, we will always end up using Heap Sort or Merge Sort. But suppose we know that our data will be almost sorted, then Heap or Merge will perform far worse than Insertion Sort. We could not have taken advantage of this fact if you didn't know the implementation details.

4.3 - Some real-world implementations

Learning "data structures and algorithms" is much more than just knowing how to implement a linked list and build a red-black tree (or at least it should in the minds of programmers).

The type of algorithmic problem-solving skill required in real-time application development is being able to see the new and unique data structure and computational tasks associated with your feature.

4.3.1 - Example 1:

Let's take the real-world example of Propeller head's music making program, **Reason**.

Reason offers us a virtual studio in a box where we can design our own virtual studio layout and connect our devices however we see fit.

If we are familiar with **graph theory** we will immediately ask ourselves, so what happens when we have created a dependency loop, whereby device A provides output to a chain of devices that eventually feeds back into device A? What order should we process them in?

With just one feedback loop it isn't that difficult. What makes the question more complex is that there may be multiple feedback loops intersecting each other, and the order we process devices will determine the total delay of the system since a feedback loop requires one of the connections to actually be delayed and processed on the next processing iteration.

You can go even further, as **Reason** now has "plugin delay compensation," which compensates for the varying delays introduced by devices to align different tracks with each other. This is important in music production as you do not want your drum track to be unaligned with your bass track because they have different effects in their processing chain. (Propellerheads.se. (2018))

4.3.2 - Example 2:

Another real-time problem is optimizing execution of applications on virtual machines.

For over 10 years JavaScript was slow to execute relative to other programming languages. That was until Google came out with V8 (which powers Node.js).

V8 runs a lot like the JVM (Java Virtual Machine), and makes use of methods like escape analysis that works out where it can eschew the use of the garbage collector for managing the lifetime of a variable, and compile the method to efficient machine code.

We won't solve these sorts of problems by reimplementing quicksort. We will need to have a grasp on **graph theory** and more complex algorithms such as min-flow and be able to spot NP-hard problems. We need the ability to perceive the structure and complexity of the desired feature and figure out the algorithms ourselves.

4.4 - Unexpected issues of using algorithms

Algorithms are often elegant and incredibly useful tools used to accomplish tasks. They are mostly invisible aids, augmenting human lives in increasingly incredible ways. However, sometimes the application of algorithms created with good intentions leads to unintended consequences. Recent news items tie to these concerns:

1. The British pound dropped 6.1% in value in seconds on Oct. 7, 2016, partly because of currency trades triggered by algorithms. (Buttonwood, 2016)
2. Microsoft engineers created a Twitter bot named “Tay” in an attempt to chat with Millennials by responding to their prompts, but within hours it was spouting racist, sexist, Holocaust-denying tweets based on algorithms that had it “learning” how to respond to others based on what was tweeted at it. (Gibbs, 2016)
3. Facebook tried to create a feature to highlight Trending Topics from around the site in people’s feeds. First, it had a team of humans edit the feature, but controversy erupted when some accused the platform of being biased against conservatives. So, Facebook then turned the job over to algorithms only to find that they could not discern real news from fake news. (Thielman, 2016)
4. Well-intentioned algorithms can be sabotaged by bad actors. An internet slowdown swept the East Coast of the U.S. on Oct. 21, 2016, after hackers bombarded Dyn DNS, an internet traffic handler, with information that overloaded its circuits, ushering in a new era of internet attacks powered by internet-connected devices. This after internet security expert Bruce Schneier warned in September that “Someone Is Learning How to Take Down the Internet.” And the abuse of Facebook’s News Feed algorithm and general promulgation of fake news online became controversial as the 2016 U.S. presidential election proceeded.

4.5 - A small research on the practical uses of Hash tables at the IBM

IBM has a long history of making fundamental contribution to Hash Functions and their applications. Carter and Wegman discovered the use of Universal Classes of Hash Functions

which have the property that when used for Hash Tables there are no inputs which are bad for too many of the functions within the class. Thus, this was one of the first important worst case random algorithms. This class can also give a secure one-time pad like authentication code.

(Carter, 1977)

4.6 - Relating research with our experimental findings

Here, we base our experimental results on some of the research findings that we have identified.

4.6.1 - Research finding:

"In [Friedman et al 2003], the use of hash tables in embedded real-time systems is considered. An incremental approach is proposed for the reorganization of hash tables that is similar to incremental garbage collection. The proposed approach is applied to hash tables using chaining. That work is motivated by the performance problems that appear when a hash table with chaining is very loaded, resulting in many collisions and linear search through long lists. The conventional solution is to build a new table, larger than the original, all at once, removing records from the original table and inserting them in the new table. This conventional approach makes the table unavailable while it is completely reconstructed. Consequently, this approach is not adequate to be used in real-time applications. The proposal in [Friedman et al 2003] makes insertions in an incremental way, so as to minimize the time the table is unavailable." (Oliviera – 2009)

4.6.2 - Experimental results:

We have identified these findings in the experiments we conducted. With an increased number of entries, a load factor of above 0.5, the probing hash table size is dynamically increased to two times and the hashing values are recomputed. This technically increases the memory allocation and the time taken.

4.6.3 - Research finding 2:

Linux kernel is an excellent example of understanding the tree data structure.

Let's look at how the kernel puts some data structures through the paces.

Red-black trees

From the kernel docs:

“There are a number of red-black trees in use in the kernel. The deadline and CFQ IO schedulers employ rbtrees to track requests; the packet CD/DVD driver does the same. The high-resolution timer code uses an rbtree to organize outstanding timer requests. The ext3 filesystem tracks directory entries in a red-black tree. Virtual memory areas (VMAs) are tracked with red-black trees, as are epoll file descriptors, cryptographic keys, and network packets in the “hierarchical token bucket” scheduler.”

The kernel implementation is different from most naive implementations, and that teaches us something about performance and cache locality.

“The Linux rbtree implementation is optimized for speed, and thus has one less layer of indirection (and better cache locality) than more traditional tree implementations. Instead of using pointers to separate rb_node and data structures, each instance of struct rb_node is embedded in the data structure it organizes.”

Radix trees

Again, the kernel implementation is different from the standard one:

“A Linux radix tree is a mechanism by which a (pointer) value can be associated with a (long) integer key. It is reasonably efficient in terms of storage, and is quite quick on lookups. Additionally, radix trees in the Linux kernel have some features driven by kernel-specific needs, including the ability to associate tags with specific entries.”

“The most widespread use of radix trees, however, is in the memory management code. The address_space structure used to keep track of backing store contains a radix tree which tracks in-core pages tied to that mapping. Among other things, this tree allows the memory management code to quickly find pages which are dirty or under writeback.”

4.6.4 - Experimental finding 2:

We can base this idea and swap out a data structure with another data structure like the AVL tree and see if it can improve the performance. As we have noted in our experiments, the AVL tree offers efficient memory management as compared to the hash tables. And if we have to deal with sorted hierarchical structures, the AVL trees perform better with $O(\log n)$ time complexity.

4.7 - A generic understanding of hash table and AVL tree in a fictitious scenario

We can use a simple day to day example to understand the concept of hash table and AVL trees.

Hash Table:

“Suppose I stay in a hotel for a few days, because I attend a congress on hashing. At the end of the day, when I return to the hotel, I ask the desk clerk if there are any messages for me. Behind his back is a dovecot-like cupboard, with 26 entries, labeled A to Z. Because he knows my last name, he goes to the slot labeled W, and takes out three letters. One is for Robby Williams, one is for Jimmy Webb, and one is for me. The clerk only had to inspect three letters. How many letters would he have to inspect if there would have been only one letter box?”

AVL tree:

“You have a group of people lined up by height. You want to find the place to insert a new person into the line. You go to the median tallest person and compare them to the person you want to insert. If the person is shorter than the median, then you ignore people taller than the median and repeat the process with your new group. If the person is taller than the median, then you ignore people shorter than the median and repeat the process with your new group. Your search is over when you reach a group of \leq two people.”

4.8 - Some real-world examples of hash tables and AVL trees

Here, we look at some of the real-world applications of both the hash tables and the AVL tree. There is no hard and fast rule that needs to be applied while choosing a data structure. It all depends on the requirements a

Hash tables:

1. English dictionary
2. Credit card authentication
3. User login databases protected with passwords
4. Library Classification schemes

AVL trees:

Storing naturally hierarchical data

1. File system
2. City map - geometric map data on a 2-D plane, with scales and directions
3. Cashbook - tabular schema
4. Postal addresses
5. IP Domain names
6. Genealogies
7. Personnel and Institutional Organization charts

4.9 – Are hash tables always faster than AVL trees?

Hash tables are not always faster than AVL trees. It depends on many things, such as the size of the collection, the hash function, and for some hash table implementations - also the number of delete ops.

hash-tables are $O(1)$ per operation *on average* - but this is not always the case. They might be $O(n)$ in *worst cases*.

4.10 - Comparison of hash table and AVL tree – based on our research and experimental findings

For the chosen database of student dictionary, hash table performs better of the two data structures. Generally, because of the way hash table is designed, it has better performance however, at the expense of memory. The table size can be optimized by changing the code in case of a resize.

4.10.1 - Circumstances where the probing hash table is a better choice than AVL tree

1. Hash table requires less frequent insertions and deletions
2. Unsuccessful searches generally require more time than successful searches
3. Number of elements to be stored must be known accurately in advance. Hash cannot be expanded once implemented.
4. Rehashing into a larger table is expensive. (If the number of elements is not known in advance separate chaining can still be used. This is not covered in the probing hash table.
5. Simple and easy to understand data structure is needed
6. Application needs to be developed very quickly without regard any efficient memory utilization
7. Performance degrades dramatically when the table becomes near full.
8. Need high efficiency (more specifically constant time random access to data elements)
9. If the data is huge, latency is not an issue and collisions are improbable - hash-tables are asymptotically better than using a tree.

4.10.2 - Circumstances where the AVL tree is a better choice over Probing hash table

1. In case of table traversals (in order, preorder, post order), AVL tree is a better option
2. When we need to access the minimum or maximum items or finding items in a certain range.
3. Number of elements in the database changes frequently and unpredictably
4. Data structure that self maintains its efficiency without user/programmer intervention
5. If memory is often expensive/limited resource, AVL tree is a better option.
6. We can get all keys in sorted order by just doing In order Traversal of AVL tree. This is not a natural operation in Hash Tables and requires extra efforts.
7. Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with AVL trees. Like sorting, these operations are not a natural operation with Hash Tables.
8. Efficient utilization of memory. Hash tables require additional memory for resizing which is not the case with the AVL tree.
9. With Self-Balancing mechanism, all operations are guaranteed to work in $O(\log n)$ time. But with Hashing, $O(1)$ is average time and some particular operations may be costly, especially when table resizing happens.
10. Ordering is important. [hash-tables are not maintaining order, BST is sorted by definition]
11. Latency is an issue - and you cannot suffer the $O(n)$ that might occur. [This might be critical for real-time systems]
12. For relatively small collections - many times the hidden constant between hash table's $O(1)$ is much higher than the tree's - and using a tree might be faster for small collections.

4.11 - Conclusion

The study of the data structures thus far revealed that there is no hard and fast rule to choose a specific type of data structure to address our problems. As we have understood, the AVL trees are memory efficient, but the hash tables perform faster operations. Also, an AVL tree can be used for storing an ordered collection, whereas a hash table can be used to store an unordered collection. In the end, it all comes down to the data that is available to us to perform these operations in an effective and time efficient manner.

4.12 – References

1. Buttonwood. (2016, October). Why sterling suffered a “flash crash”. Retrieved from <https://www.economist.com/finance-and-economics>
2. Carter, J. L., & Wegman, M. N. (1977). Universal classes of hash functions (Extended Abstract). Proceedings of the Ninth Annual ACM Symposium on Theory of Computing - STOC 77. doi:10.1145/800105.803400
3. Gibbs, S. (2016, November). Opinion from the Guardian's US edition, *Microsoft's racist chatbot returns with drug-smoking Twitter meltdown* | Technology | The Guardian. Retrieved from <http://www.theguardian.com/technology/2016/mar/30/microsoft-racist-sexist-chatbot-twitter-drugs>
4. Oliveira, R. S., Montez, C., & Lange, R. (2009). On the use of hash tables in real-time applications. 2009 IEEE Conference on Emerging Technologies & Factory Automation. doi:10.1109/etfa.2009.5347067
5. Propellerheads.se. (2018). Reason by Propellerhead: the music-making software with everything you need to create amazing music. | Propellerhead. [online] Available at: <https://www.propellerheads.se/en/reason> [Accessed 1 Jun. 2018]
6. Thielman, S. (2016, May). *Facebook news selection is in hands of editors not algorithms, documents show* | Technology | The Guardian. Retrieved from <http://www.theguardian.com/technology/2016/may/12/facebook-trending-news-leaked-documents-editor-guidelines>

