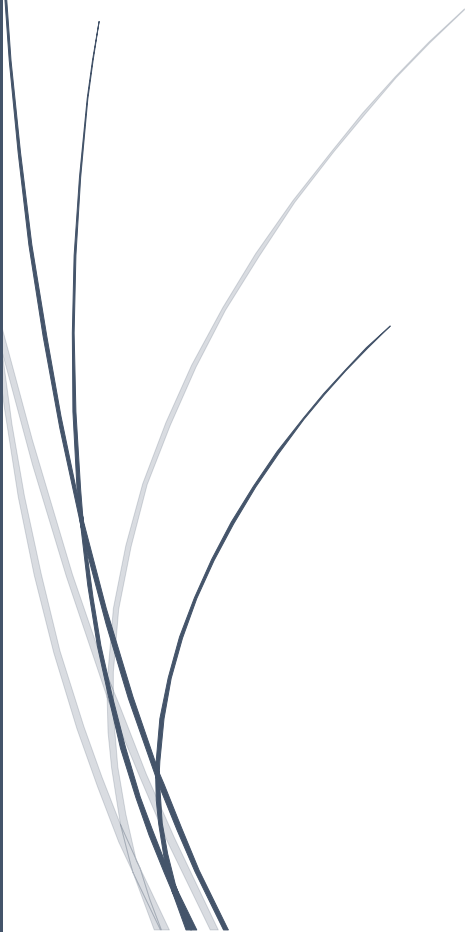


A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

12/6/2016

# Algorithms and Data Structures in Object Oriented Framework

Mini-Project 2016/2017

Several thin, curved lines in dark blue and light grey originate from the bottom left and sweep upwards and to the right.

Musab Adam  
150310350

## Introduction

In this report, I will be demonstrating the advantages and disadvantages of certain data structures. Data structures are ways of storing data into memory. Data structures must be able to add data, find specific or groups of data and lastly remove data from the memory. The best data structures can be efficient by storing data in such a way that it minimizes the cost of memory and can be quick in locating data by storing it in such a way that it would cost less time to remove data from the memory.

Here I will be demonstrating two well-known methods of storing data into memory. The first data structure I will be considering is the “array and count” data structure.

## Hash Tables

Hash tables is another common data structure in programming. It's a combination of ways of indexing and a common method of structuring data called linked list. Its unique indexing method works by calculating an index based on a predefined formula. For example, string type data can be indexed by the number of vowels so the string word “alpha” would be given the index number two since it has two vowels, thus being stored into index two in the array.

Every time the program wants to store a string (or any other data types) into the array, it will calculate the index using the formula otherwise known as hash function, however if there was a string that was given the same index as the previous string, it would cause a collision. To reduce the risk of having collisions, having efficient hash function is vital, for instance the hash function I mentioned earlier of using the vowels to produce a hash index is very inefficient since there are many words in the dictionary that can have two vowels. An example of efficient hash indexing would be to sum the ASCII number of each character in the string then mod arrays length. This would reduce the risk of a collision significantly. However even if you had the best hash function, a collision will always occur so the objective is to reduce the number of collisions as much as possible.

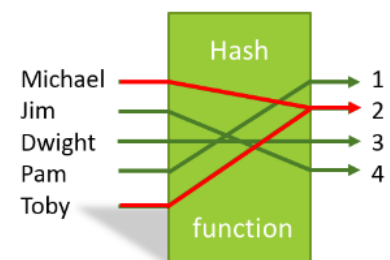


Figure 1 Showing a collision in Hash tables

But when a collision does occur when adding new data, the program must deal with it to not cause any exception errors, this is where linked list comes in. Linked list is a data structure that utilises a two-cell structure where the first cell would contain the value (actual data) and the second cell

would contain the reference of the next two cell structure.

Thus, creating a chain of objects together and each object consisting the data and the pointer to its successor. Based on the diagram, each object consists of the two cell, the second cell linking to the next object, it will keep repeating until an object's second cell will contain “null”, stating that this object is the last object in the chain.

So, when a collision does occur in a hash table, the program will detect the collision and create a linked list structure where the new data that has caused the collision will be put into a two-cell based object and then linked to its predecessor object.

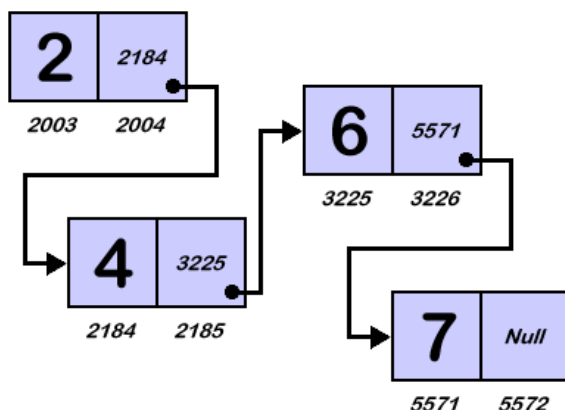


Figure 2 Showing a diagram demonstrating how linked list work

This means that the data structure can continue growing without fault unlike normal array structure where the size is fixed. Also with the use of linked list, data can be removed destructively rather than constructively therefore less harsh on memory.

To demonstrate the effectiveness of hash tables in java, I have created a class which implements this data structure. The java class has been designed to handle a collection of data of a specific type (e.g. String, Integer, Double) and consist of four instance methods. These are “add”, “remove”, “count” and “extend”.

### Class constructor

The class begins with a constructor which initialises the array “words” as type “Cell” and setting the size based on the argument that is sent through. The array will be initialised to contain only Cell instances.

```
public class WordStoreImp implements WordStore {  
    //creates an array of type Cell which will hold the linkedlists  
    Cell[]map;  
    public WordStoreImp(int n){  
        //linkedlist.first stores the string word  
        //linkedlist.next stores the next linkedlist object as type Cell  
        map=new Cell[n];  
    }  
}
```

### Nested Cell class

The Cell class is where it puts together the data that is being stored under the variable type string called “first” and the instance successor as the variable type “Cell” called “next”. If I wanted to add a string into the array, the method will create a new instance of type “Cell” and will have the parameter set to “String variable” which would create an instance and set the String first with the String variable that was passed through, and since the string was the only parameter, it will set the “next” variable to null to state that there is no other existing “Cell” object after this.

```
private class Cell  
{  
    String first;  
    Cell next;  
    //holds the next linkedlist or if there isn't another list then it holds null which means the end of the linkedlist  
    Cell(String f)  
    {  
        first=f;  
        next=null;  
    }  
    Cell(String f,Cell n)  
    {  
        first=f;  
        next=n;  
    }  
}
```

## Hash Generator

The hash generator method has the role of determining where the soon to be added String data will be stored in the array. The method has String variable in its argument which will contain a copy of the String that is going to be added into the array. The method will go through each character in the String "word" and convert it into an ASCII number, The ASCII number will then be added into the variable count. The piece of code within the loop will keep repeating until it has gone through each character and adding the ASCII of each character into the count variable. Once the loop has been complete, the method will then do the value of count mod the length of the array which will divide the count value by the length of the array being used in the class and will return remainder to the method that called the hash generator method.

```
public int IndexGen(String word){
    //creates a hash based on the number of vowels in the String word
    int count=0;
    char letter;
    for (int i=0;i<word.length();i++){
        count=count+(int)word.charAt(i);
    }
    count=count%map.length;
    return count;
}
```

## Add method

The add method has the role of taking the String variable, calling the hash generator to determine where in the array the variable will be stored and will create an instance of "Cell" which will hold the String. Initially the method will call the hash generator method to pinpoint which part of the array the Cell instance will go into. The method will then assign the content inside the array to an uninitialized "Cell" instance to ensure that if there is an initialised "Cell" instance in the array, the method can detect a collision. However, if there is no "Cell" inside the array, the method will simple create a new Cell instance with the parameters containing the String. However, if there was a "Cell" instance that existed already in the array, the method will detect the collision and simply create a new "Cell" instance with the parameters containing the String and also the Cell instance that existed in the array therefore adding the new "Cell" to the front of the chain. Once the process of adding the Cell into the chain of Cells, the linked list data will be assigned back to the array.

```
public void add(String word){
    //generate a Hash Index by passing the String word into the method IndexGen()
    int index=IndexGen(word);
    //Retirieve the Linkedlist from the array cell specified by the index
    Cell list=map[index];

    //if the linkedlist.first inside the specified array has nothing(null) then add the String word into that linkedlist
    if (list==null){
        list=new Cell(word);
    }else{
        //else add the new linkedlist to the front of the linkedlist chain
    }
}
```

```
        list=new Cell(word,list);
    }
    //apply the newly constructed linkedlist back into the specified array
    map[index]=list;
}
```

### Count Method

The count method's role is to simply count the number of times a specific String occurs in the array. It will then return the total number back to the method that called the count method. The method will initially take the string that the method will look for into its arguments. It will then generate a hash index by calling the hash generator method to pin point the possible location the string might exist in the array. Once it finds the location where it could be stored, it will assign the linked list that is stored in the position (specified by the hash) into an uninitialized "Cell" instance. The method will then go through the chain of Cells inside the linked whilst checking if the "first" variable (which holds the string) equals the String specified by the method argument. If it does meet the condition, the count will increment by one and will keep looping until it finds the variable "next" which stores the pointer to the next "Cell", as null meaning it has reached the last "Cell". Once the loop terminates, the method will return the count back to the method that called the count method.

```
public int count(String word){
    int count=0;
    Cell list=map[IndexGen(word)];
    for(; list!=null; list=list.next){
        if(word.equals(list.first)){
            count++;
        }
    }
    return count;
}
```

### Remove Method

The remove method role is to find the specified String in the array and to remove the first occurrence of that string only. To do this, the method will find the possible location in the array where the string will be and will assign the content in that array into a "Cell" instance. It will first check if the "Cell" instance is empty, then the String does not exist in the array and will end the method. However, if the instance is not empty, the method will check if the initial Cell in the chain contains the string. If it does, the method will assign the "next" variable (successor to the initial Cell) to the first "Cell" (removing the first Cell instance in the process). If the first "Cell" contains null, then the method will go through each Cell in the linked list using a pointer by looping. In each iterative, the method will check if the "Cell" stored in the variable "next" contains the string. If that condition meets true, then the cell that contained the specified string will use its "next" variable and assign it to the "Cell" active in the current iterative.

```
public void remove(String word){
    Cell list=map[IndexGen(word)];
    if(null==list){
    }else{
        Cell temp=null;
        temp=list;
        if(temp.first.equals(word)){
            temp=temp.next;
        }else{
            for(;temp.next!=null; temp=temp.next){
                if(temp.next.first.equals(word)){
                    temp.next=temp.next.next;
                    break;
                }
            }
        }
        map[IndexGen(word)]=temp;
    }
}
```

## Performance of Hash Tables

To test how efficient and fast the Hash table is, it will be tested using 3 test classes which will stress test one of the three methods (add, count, and remove). Each test will initially add 10000000 Strings into the structure and then it will have the specified method process in its defined way. At the end of the execution, the test will output whether it was successful and will also output how long it took to execute a certain part of the code.

### Testing add method

To test the efficiency of the add method implementing array and count. I will execute the test class four times and in each instance, I will increment by  $n*10$ .

Based on the test and the graph, the hash table structure proved to be extremely fast in adding data into its structure. In every iterative, where the amount of data added increase, the method was still able to execute near instantly. The reason it was instant, was that the method didn't need to go through all the data, it simply added the data to the beginning of the linked list and moved on.

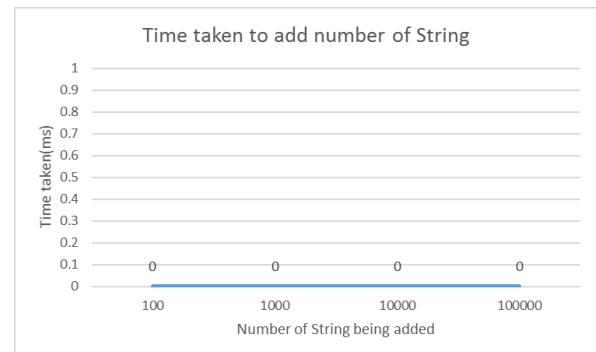


Figure 3 Showing timing figure of execution time of the add method

### Testing count method

To test the efficiency of the count method. I will execute the test four times and in each instance, I will have 10000000 Strings stored in the array and will search for  $n*10$  amounts of data per instance.

Based on the test and the graph, the hash tables performance seemed to diminish very rapidly the more String it searches. However, it is difficult to judge the performance without looking at how other data structures cope with this test

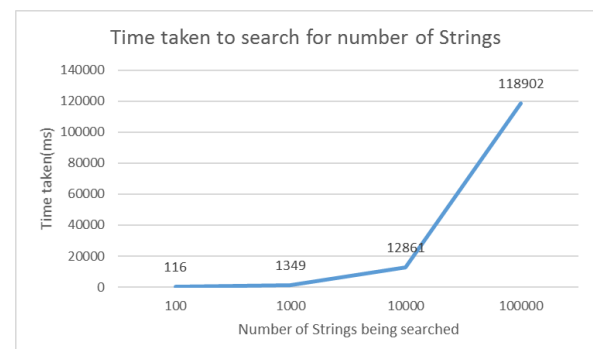


Figure 4 Showing timing figures of execution time of the count method

### Testing remove method

To test the efficiency of the remove method. I will execute the test four times and in each instance, I will have 10000000 strings stored in the array and will remove  $n*10$  amounts of data per instance.

Based on the test and the graph, the hash table structure again proved to be very fast in removing Strings from the structure. In every iterative of the test (whilst increasing the number of Strings to remove), the method managed to complete its execution of removing all the requested String in less than half a second.

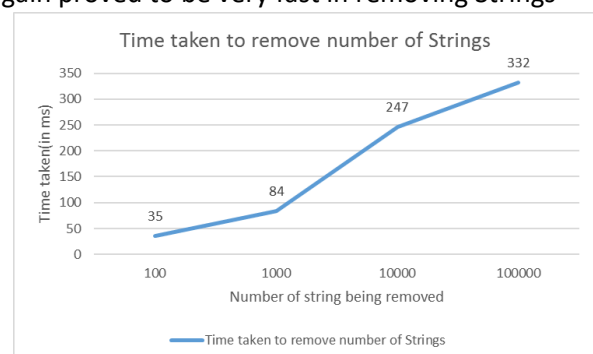


Figure 5 showing timing figures of execution time of the add method

## Comparisons with other data structures

To have a realistic idea of how efficient the hash table structure. I have created a duplicate class which implements the data structure called "array and count". The class that implements "array and count" will be tested using the same test classes used to test hash tables and in the same environment i.e. same hardware. no background programs running and power setting set to high performance.

### Array and Count

Array and count is a data structure that heavily relies on the use of arrays. Arrays is a storing data into indexed locations. However, arrays are fixed data length which means that the size must be set when initialising an array in programming. This means that when adding data into array, at some point the array will become full and therefore not be able to add more data until another data is deleted thus freeing space for additional data.

Single variable	1				
Array: Indexes	0	1	2	3	4
Values	1	3	8	23	99

Figure 6 demonstrating visually how array store data in ordered indexed cell

To get around this problem, programmers have adopted an approach where they monitor how many times the program is requested to add data by having a count variable that increments each time data is added into the array. Once the counter variable reaches the size-1 (its minus one due to java starting the index at zero) a new array will be initialised with an increased sized compared to the previous array size and each cell from the previous array are assigned to the cells of the newly created array.

### Performance of Array and Count

#### Testing add method

To test the efficiency of the add method implementing array and count. I will execute the test class four times and in each instance, I will increment the number of additional Strings added by  $n \times 10$ .

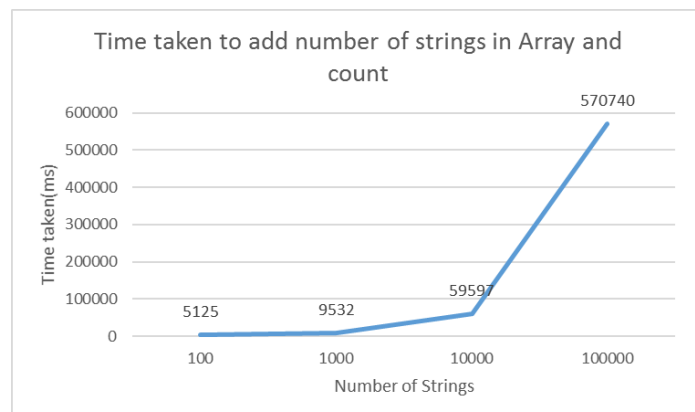


Figure 7 Showing timing figures of the add method using Array and count



### Testing count method

To test the efficiency of the count method. I will execute the test class four times and in each instance, I will search  $n \times 10$  number of Strings.

On this test, the last iterative of this test failed as the system ran out of memory therefore in this graph, the data will show 0.

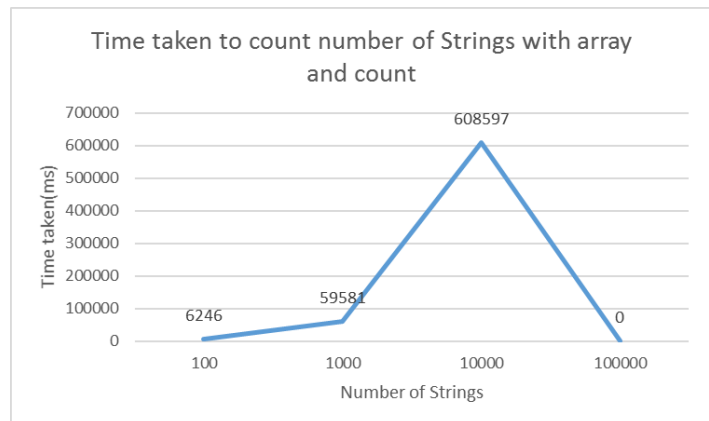


Figure 8 Showing timing figures of the count method using Array and count

### Testing remove method

To test the efficiency of the remove method. I will execute the test four times and in each instance, I will have 10000000 strings stored in the array and will remove  $n \times 10$  amounts of data per instance.

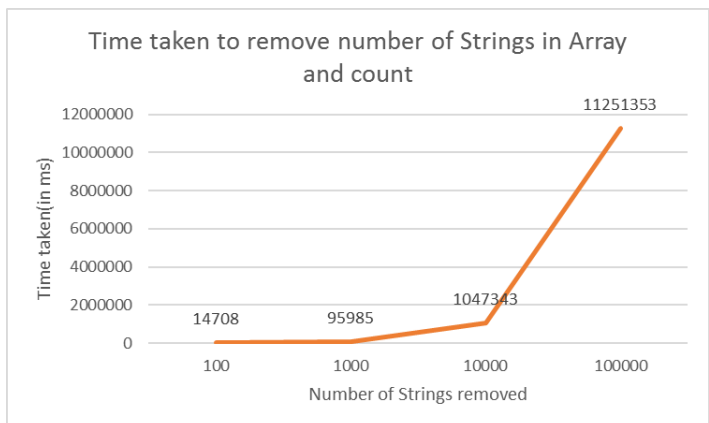


Figure 9 Showing timing figures of the remove method using array and count

### Comparison of Hash table and Array and count

Now that both structures have been tested and the performance have recorded, I have combined the results of both data structures (per method)

Based on the combined graph, the difference in performance was very clear. In each iterative of the add method using Hash tables, the program completed its execution instantly at 0ms consistently throughout the stress test. On the other hand, when testing the add method with "array and count", the results were extremely slow for instance to add just 100 Strings to the array took more than 5 seconds to complete compared to the 0 seconds taken by using Hash tables. However, when it came to adding 1000000 Strings into the array using "array and count", the program took near ten mins to complete its execution whereas with Hash tables, it was again zero seconds. The reason as to

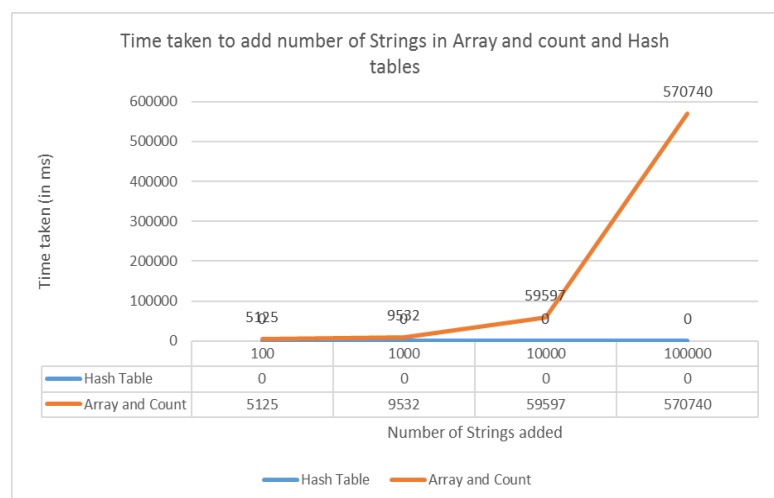


Figure 10 Showing the comparison of the timing figure of both data structures executing the add method

why the performance of Hash tables is significantly superior to Array and count is because where array and count must create new arrays to expand to allow more data to be added then adding all the data from the old array into the new array then finally adding the new string. With hash tables, it is simply adding a new cell and linking the existing cells with the new cell making it a two-step process hence why adding data is near instant.

The next two versions of count test showed that once again, using hash table structure was significantly quicker than using "array and count". For instance, when trying to search for 100 Strings through 10million stored Strings, it took the program near 100 milliseconds to complete its execution where in contrast, the program took more than 6 seconds to complete its execution of the method which x60 slower than the execution time with Hash tables. This is because with array and count implementation, the program must go through each data stored in the array until it finds the specified String. Whereas since data is categorised based on their hash values, the program can simply find what category the string would be classed as and then find the category which will consist of data closer or equal to the specified String without having to go through the 10million Strings stored in the structure.

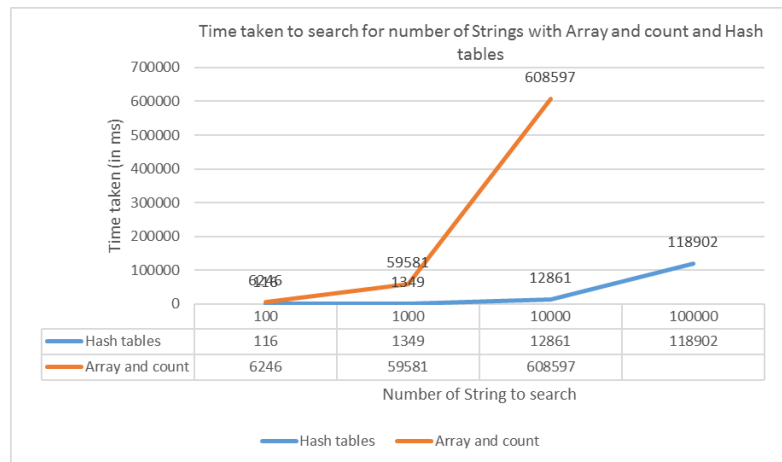


Figure 11 Showing the comparison of the timing figure of both data structures executing the count method

For the remove test, based on the combination of results from both Hash tables and "Array and count". The use of hash tables was clearly faster. From analysing the graph, the program was consistently able to complete the execution of removing the specified number of Strings in less than a second. In contrast to the program using "Array and count" it took more than a minute to complete its execution and at 1000000 Strings, the "Array and count" took approximately 3 hours to complete its

execution making "Array and count" 11000000x slower than the program implementing Hash tables. The reason as to why hash tables was significantly quicker in removing the items was simply because the program implementing "Array and count" had to first search for the data and based on the count test the recorded data showed that Array and count was extremely slow, when the program did find the data it was searching for, it removed the string constructively meaning that memory was being clogged with unused arrays. However, with Hash tables, as proved in the count test, searching is faster than "array and count" and when the program finds the String it's looking for, it simply removes the String in a destructive behaviour which means less memory usage and no time spent creating new arrays.

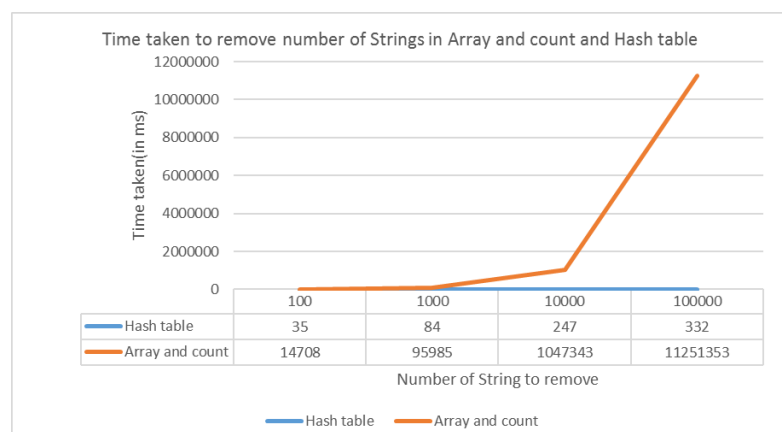


Figure 12 Showing the comparison of the timing figure of both data structures executing the add method

## Conclusion

Based on the comparison of the two tests, it was clear that using hash tables was the better option out of the two data structures. Using hash tables rather than Array and count was not only the faster option of the two (based on the tests) but in the long term, more memory efficient than array and count due to the way hash tables stores its data for instance with array and count, the program must create new arrays with bigger size to get around the issue with having fixed sized structure whereas with hash tables and linked lists, the program would simply add a new cell to the front thus allowing the structure to have a dynamic size. Also when it comes to dealing a very high number of Strings, the use of categorising the Strings in hash tables was a big advantage to the data structure, this is because it allows the system to pick out a group of strings from the list and check through the group of Strings rather than going through each String in the array from the beginning to the end of the array.

## References

CS50. (n.d.). *Hash Tables*. Retrieved from Youtube: [https://www.youtube.com/watch?v=h2d9b\\_nEzoA](https://www.youtube.com/watch?v=h2d9b_nEzoA)

Harmsen, B. (2014). *QlikView hash functions and collisions*. Retrieved from <http://www.qlikfix.com/2014/03/11/hash-functions-collisions/>

Parlante, N. (2001). *Linked List Basics*.