E/15/280

Premathilaka M.P.U

HASHING

INTRODUCTION

Open hashing is a method of storing data relative to some keys generated by a function. This function is usually called the hash function. In typical open hashing it is used an array with specified number of buckets. In this lab collisions are handled using linked lists and these linked lists are connected to buckets in the array. Depending on the hash values generated by the hash functions data is distributed among the buckets

PROCEDURE

A good hash function should be complex enough such that it does not produce the same hash value from two different inputs. An ideal hash function should distribute the keys evenly for all the buckets. With an ideal hash function search time would be M/N where M is the number of keys and N is the number of buckets. We can manipulate the search time by changing the number of buckets(N) because, N is inversely proportional to the search time.

After implementing the specified data structure, different hashing algorithms/implemented hash functions are tested for given text file inputs.

- hashFun_1()

Pseudo code

$for\ i = 1 : length(word)$

$$\text{hashval} = \text{hashval} + 31^{i} * \text{charAt(i)}$$

Function

```
int hashFun_1(String key, int buckets){

  int val = 0;
  int g = 31;
  for (int i = 0; i < key.length(); i++) {
      int unicode = key.charAt(i)-‘a’;
      val += Math.pow(g, i) * unicode;
  }
  return val % buckets;
}
```

- hashFun_2()

  Pseudo code

  $for\ i = 1 : length(word)$
  $\text{hashval} = \text{hashval} + \text{charAt(i)}$

  Function

```
int hashFun_2(String key, int buckets){
        String lowerCase=key.toLowerCase();
        int val=0;
        for(int i=0;i<lowerCase.length();i++){
                val+=(lowerCase.charAt(i)-'a')%buckets;
        }
}
```

- Fowler-Nell-Vo hash algorithm (FNV)

IMPLEMENTATION

On the java file name HashTableImp.java there is a java class HashTableImp.class . The hash table is implemented using array and it is co-operated by linked lists. There are 3 implementations of the interface using 3 different hash functions. HashTableImp.class uses the implemented hash function.

Array is built with Chain objects. Chain objects are built by Node objects. For each bucket in the array a Chain is connected to it.

When a word is adding to the hash table,

- First it calculates the hash value or relevant index on the bucket array
- Check whether a Node is already created for that particular word in the bucket
- If yes, increment the data attribute on the node
  If no, add a new node to that word on the bucket

In this particular application, for a selected chain, number of adding Nodes to the chain is greater than number of retrieving values from nodes.
Since, collisions could affect to the complexity of the programme, inserting a new node to the bucket chain is implemented as complexity is of inserting Node is O(1).

How to run the programme:

1) Use javac complier. Need to Run.java file with Command line arguments
   Argument 1 = text file going to analyze
   Argument 2 = number of buckets on the bucket array
   Ex:

   javac Run.java sample-text1.txt 100

RESULTS

Data is represented considering following criteria.

(In this this implementation similar words are put to a node in the bucket and word count is incremented. For the calculation similar words has considered as one word count when counting number of collisions)

1. Min-Max comparison

   For a given bucket size if the range,

   range =maximum number of collisions – minimum number of collisions

   is small, that means the keys are distributed among all the buckets and almost all the buckets have occurred same amount of collisions.

Bucket size = 10

	sample-text1.txt			*sample-text2.txt*		
	Minimum collisions	Maximum collisions	Range	Minimum collisions	Maximum collisions	Range
hashFun_1	31	526	495	48	205	157
hashFun_2	75	104	29	57	78	21
FNV	73	106	33	57	80	23

Bucket size = 50

	sample-text1.txt			*sample-text2.txt*		
	Minimum collisions	Maximum collisions	Range	Minimum collisions	Maximum collisions	Range
hashFun_1	4	486	482	4	165	161
hashFun_2	8	27	19	7	23	16
FNV	8	29	21	8	20	12

Bucket size = 500

	sample-text1.txt			*sample-text2.txt*		
	Minimum collisions	Maximum collisions	Range	Minimum collisions	Maximum collisions	Range
hashFun_1	0	6	6	0	5	5
hashFun_2	0	17	17	0	18	18
FNV	0	6	6	0	5	5

2. Average

   Average number of collisions for different hash functions for a specified bucket size

Bucket size = 10

	sample-text1.txt (AVG)	*sample-text2.txt (AVG)*
hashFun_1	92.0	68.0
hashFun_2	92.0	68.0
FNV	92.0	68.0

3. Standard Deviation

Bucket size = 10

	sample-text1.txt (STD)	*sample-text2.txt (STD)*
hashFun_1	144.88	45.49
hashFun_2	8.49	6.48
FNV	10.34	6.93

Bucket size = 50

	sample-text1.txt (STD)	*sample-text2.txt (STD)*
hashFun_1	66.87	21.84
hashFun_2	4.35	3.46
FNV	4.35	3.0

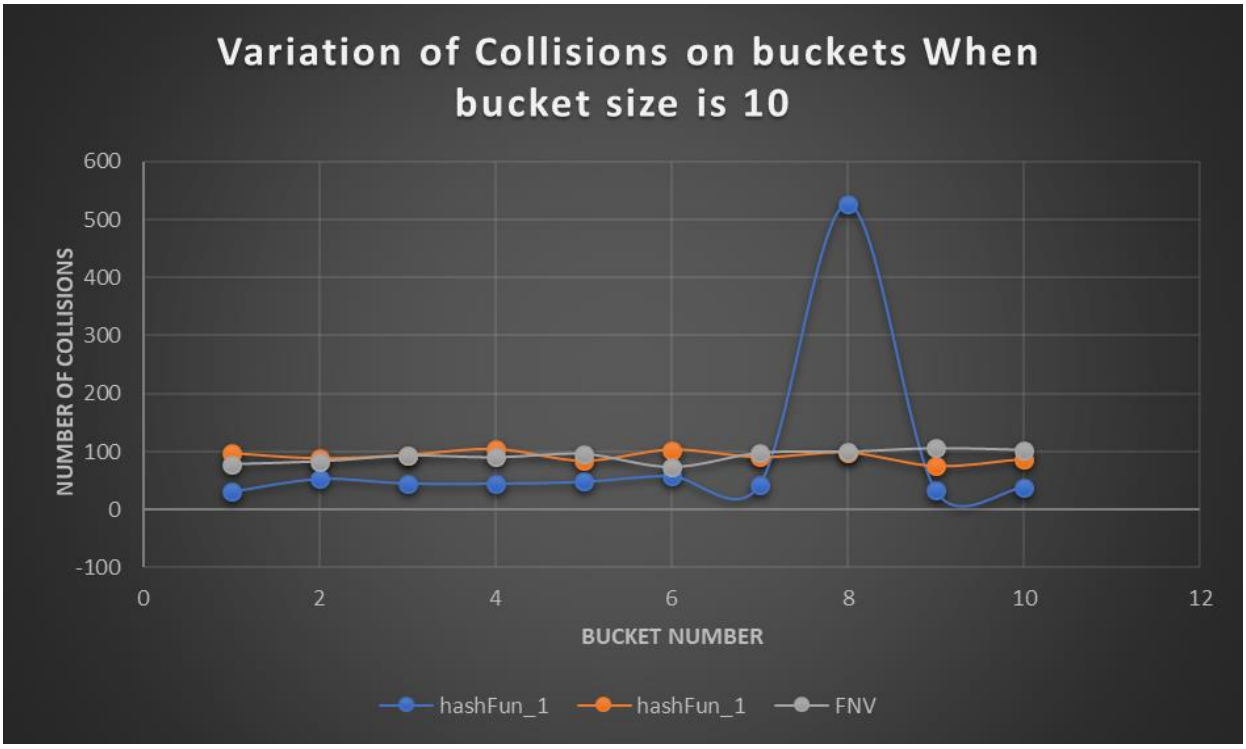

FIGURE 1

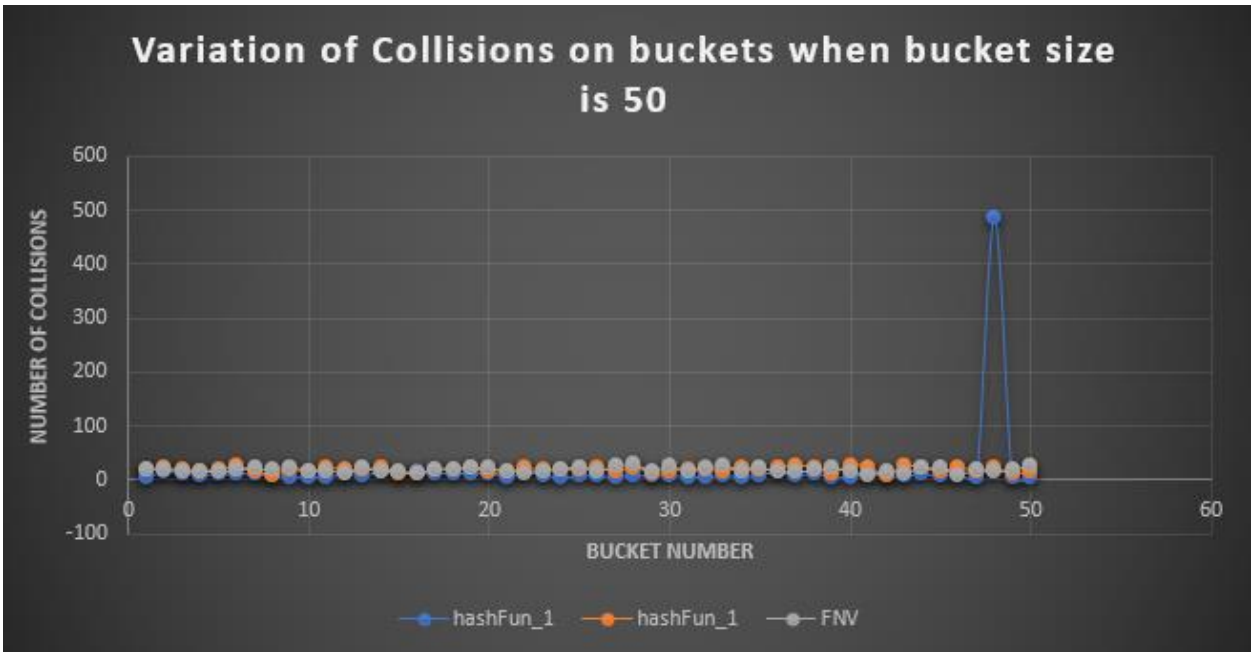

FIGURE 2

Distribution difference between sample-text1.txt & smple-text2.txt

Bucket size = 10

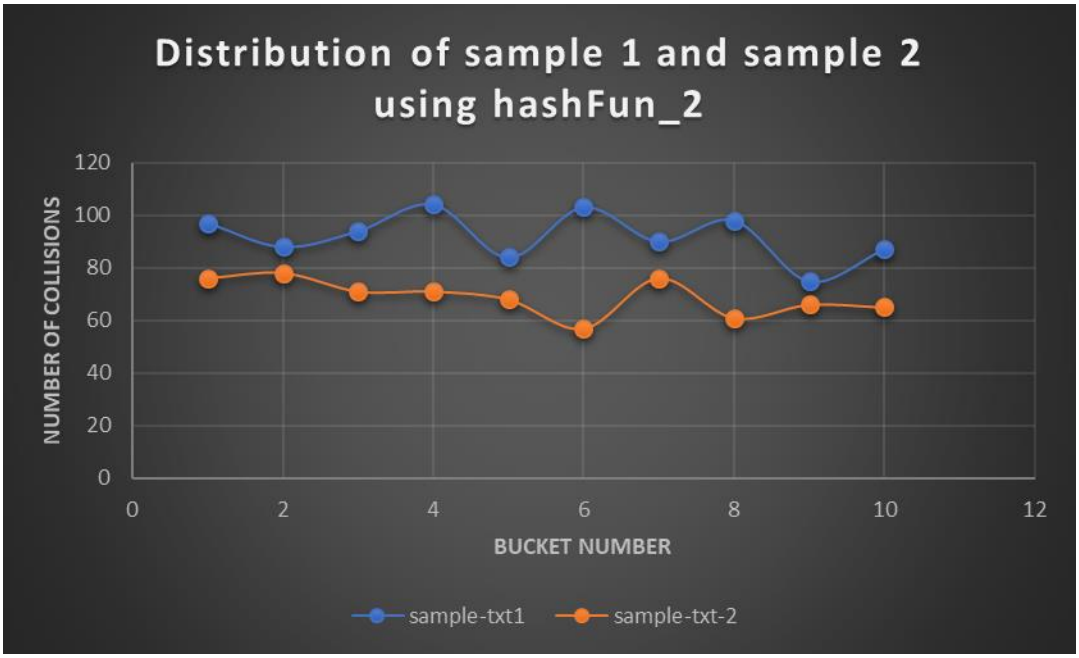

FIGURE 3

Bucket size =50

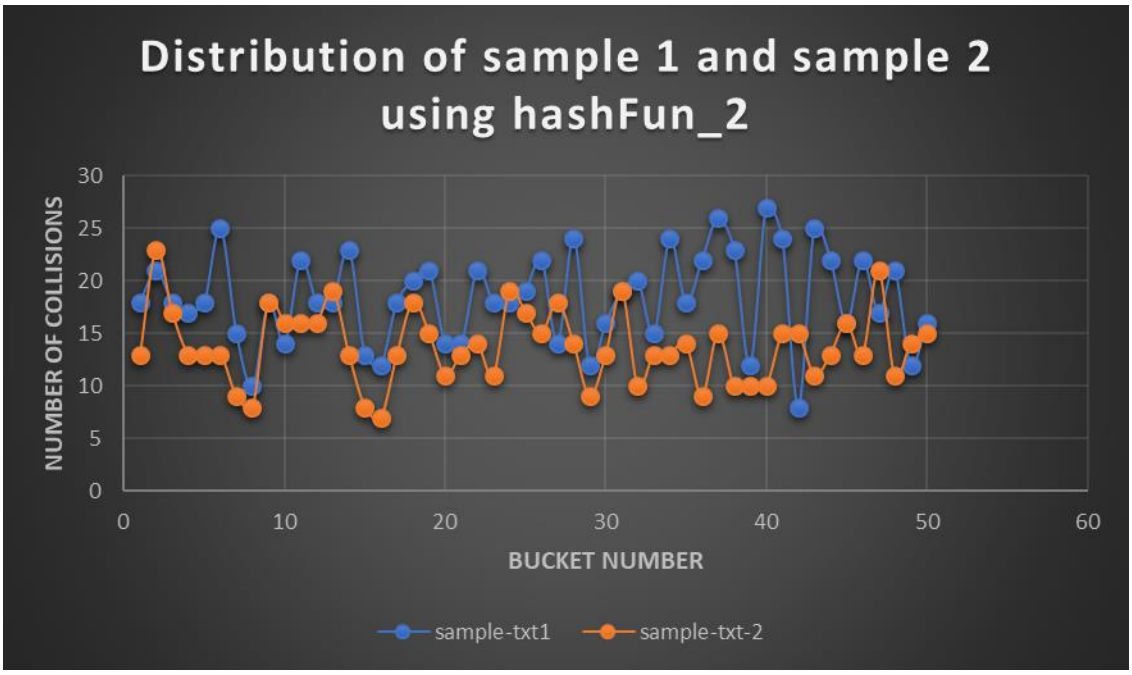

FIGURE 4

CONCLUSION

- Since the Average collisions for three different implementations are identical, best hash function cannot be determined by considering the Average collision.

- Two tested files (sample-text1.txt and sample-text2.txt) have words more than 500. Therefore, when the bucket size is selected as 10,50 and 500, by calculating the range in min-max approach we can determine the betterment of the hash functions. When the range is high in collisions, it can conclude that the hash function does not evenly distribute words among the bucket array. If the range is small, that specifies the hash function has evenly distributed words among the buckets. Considering this criteria hashFun_2 is better that hashFun_1.

- When we consider the standard deviation of collisions, the standard deviation emphasize how good words are distributed among the bucket array when the size of the bucket array is changed. When the standard deviation becomes low, the betterment of the hash function increases. Considering this criteria hashFun_2 is better that hashFun_1.

- As shown in Figure 3-4, for the same hash function (hashFun_2) number of collisions for a bucket in sample-text1.txt is grater than number of collisions in sample-text2.txt. Therefore, we can conclude that particular hash function works well in sample-text2.txt

- For this particular input files (sample-text1.txt and sample-text2.txt) hashFun_2 has performed really well when it compares with the Fowler-Nell-Vo hashing algorithm. Therefore, it can be concluded that hashFun_2 is the most suitable hash function as an implementation.