KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY

COLLEGE OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

THESIS TOPIC: AN ENHANCED NON- CRYPTOGRAPHIC HASH FUNCTION

BY

AKOTO-ADJEPONG VIVIAN

AN ENHANCED NON- CRYPTOGRAPHIC HASH FUNCTION

A THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE, KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI GHANA

BY

AKOTO-ADJEPONG VIVIAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF MASTER OF PHILOSOPHY DEGREE IN COMPUTER SCIENCE AT THE DEPARTMENT OF COMPUTER SCIENCE, COLLEGE OF SCIENCE, KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI GHANA

SEPTEMBER, 2016

DECLARATION

I hereby declare that this submission is my own work towards the Master of Philosophy degree in computer science and that, to the best of my knowledge, it contains no material previously published by another person nor material which has been accepted for the award of any other degree of the university, except where due acknowledgement has been in the text.

Akoto-Adjepong Vivian …….. …….

(PG8136112) Signature Date

Certified by:

Dr. Michael Asante …….. …….

Supervisor Signature Date

Certified by:

Dr.J.B Hayfron-Acquah…….. …….

Head of Department Signature Date

ABSTRACT

How to store information for it to be searched and retrieved efficiently is one of the fundamental problems in computer science. There exists sequential search that support operation such as INSERT, DELETE and RETRIVAL in O (n log (n)) expected time in operations. Therefore in many applications where we need these operations, hashing provides a way to reduce expected time to O (1). There are many different types of hashing algorithms or functions such as cryptographic hash functions, non cryptographic hash function, checksums and cyclic redundancy checks. Non-cryptographic hash functions (NCHFs) take a string as input and compute an integer output (hash index) representing the position in memory the string is going to be stored. The desirable property of a hash function is that the outputs are evenly distributed across the domain of possible outputs, especially for inputs that are similar. Non-cryptographic hash functions have an immense number of applications, ranging from compilers and databases to videogames, computer networks, etc. A suitable hash function and strategy must be used for specific applications. This will help efficient use of memory space and access time.

The most essential features of non cryptographic hash functions is its % distribution, number of collisions, performance, % avalanche and quality which are the properties of the hash function. Basing on the properties assessed using the VHASHER developed; the results clearly demonstrated that; Vivian hash function that was developed had better properties as compared to other hash functions.

ACKNOWLEDGEMENTS

No amount of words is enough to express my gratitude to God for his grace to do this work. And to my supervisor Dr. Michael Asante, am highly honored to study under your feet. I would like to also acknowledge all my lecturers, you have really been of help, especially Mr. Asamoah and Mr. J. K. Panford .

Finally, I would like to acknowledge all hash function researchers whose work inspired me to undertake this research.

DEDICATION

This thesis is dedicated to my son Emmanuel Okyere-Gyamfi

TABLE OF CONTENT

CHAPTER ONE

INTRODUCTION

1.0 Background of study

One basic problem in computer science is how to efficiently search and retrieve stored information. There exists sequential search that support INSERTION, DELETION and RETRIVAL operations in expected time of O (n log (n)) (Singh and Garg 2009). Therefore in applications where INSERTION, DELETION and RETRIVAL are needed, using hash algorithms help tominimize expected time to O (1).

Hashing is used to store and retrieve information in databases. This deals with key attributes or properties andmake use of each individual character numbers in the data or key. To implement keyed tables, hashing is a recommended technique (Walker, 1988).

Algorithm for lists, trees and stacks takes time proportional to the data size, i.e., O (n).

In order to locate and retrieve information, hashing is a recommended scheme because is effective and efficient.

A suitable hash function and strategy must be used to solve particular problems or for specific application. This will help efficient use of memory space and access time.

There exist different types of hash algorithms such as non-cryptographic hash algorithms or functions, cryptographic hash algorithm or functions, checksums and cyclic redundancy checks.

The main focus of study is non-cryptographic hash functions.

Non cryptographic hash algorithms or functions (NCHFs) take its input as string and compute an integer output (hash index) which represent the position in memory the string is to be stored. One of the important properties of hash functions is the even distribution of outputs across the space allocated or domain, especially for similar inputs.

NCHF'sare functions that are designed not to withstand an attackerseffort unlike cryptographic hash functionsbut are much slower. Therefore,NCHF'sare roughly 33x fasterat the expense of it not able to withstand attackers' effort. NCHFs are used in a number of applications, ranging from databases and compilers to videogames, dictionaries, computer networks, etc.

Such hash functions as stated above are: Pearson hashing, FNV hash, Bob Jenkins hash, murmur hash, city hash, buz hash etc. ETC (Zobel et al2001).

In computing memory usage and return time are very important resources to consider in running an application. This is dependent on the particular hash function one chooses to solve a problem.

1.1 Research field and subject of the study

The research field is computer science and the subject of study is non-cryptographic hash functions. Wikipedia defines hashing as transformation of characters in a string into a frequently shorter fixed-length that represents the original data or key. A hash algorithm or function transformsa key into a hash index of an array element (bucket) where the value it corresponds to is to be found. This is a subject area where much research is needed to be done to seek improved techniques and make it usage efficient, cost effective and accurate.

1.2 Research objectives

The following are the objectives of the research:

- To develop a new efficient and effective hash function.
- To create the awareness of the most efficient hash.
- To analyze if there is a defect with any of the hash functions.
- To differentiate between different hash functions
- Know various hash functions and understand how they work
- Know when to use a particular hash function

Research problem and research questions

1.3 Research problem

Using an efficient and effective non-cryptographic hash function in running an application is very important. The usage of memory space and the return time are very important resources to consider when choosing a particular non-cryptographic hash function to run an

application. Many people just choose any known hash function to solve any problem resulting in wasting memory space, increasing return time and not getting the efficient results expected.

Therefore making it necessary to develop an efficient and effective non-cryptographic hash function analyze the various common non-cryptographic hash functions so that users will choose suitable functions to run their applications.

Hence developing, analyzing and evaluating the performance and efficiency of various non-cryptographic hash functions.

1.4 Research questions

- Why developing a new hash function important?
- Why is selecting a suitable hash function for a particular application important?
- What hash function is used for solving particular problem?
- What contributions does choosing a suitable hash function provide to the system and user?
- How is selecting a suitable hash function for a particular application useful in the hashing process?
- How can the analysis help users to choose suitable hash functions?

1.5 Summary and presentation of thesis

This research is structured in five chapters. These are introduction which deals with the foundation of the research. The second chapter establishes the conceptual foundations for the research by reviewing prior studies relevant to the research topic of study. Chapter three deals with the methodology of the research, the research data and experiments employed in this study. The fourth chapter presents the summaries and conclusions of the study. Finally, the fifthchapter discusses the conclusions and recommendation issues and directions onnon-cryptographic hash functions.

CHAPTER TWO

LITERATURE REVIEW

2.0 Introduction

Most people who uses the computer to run applications just uses the interface of their program not considering what goes on at the background. A hash function takes this

process a step further, allowing user data to be stored and retrieved efficiently. An essential operation in retrieving information is accessing the information based on a key. Given a particular value, a system for retrieving information must be able to determine the location of information that is relevant to the value given. The challenge or most important thing is to store information or data for it to be searched and retrieved efficiently. This chapter involves a study on the general overview of hash functions, including the various non-cryptographic hash functions and how they work.

2.1 Hashing

Is a procedure for storing and retrieving of information at a faster rate in main memory and used insearching and retrieving of information because of its better retrievals, increased speed, reduced overhead, optimized searching of data, etc. The basicaim isefficient use of memory space and reduced return time during operations such as insertion, finding, deletion and updating of data (Cormen et al, 2001).

2.2 Overview of hashing

The word hash technically means to chop and mix. In using operators such as the mod operator, it chops the input data into sub domains; these get mixed up to provide an output that help to improve uniform key distribution.

By Donald Knuth. The term was a jargon used by Hans, Peter, Luhn of IBM but Robert Morris turned it into a formal term in a survey paper in CASM (Knuth and Wesley, 1998).

Hashing

In databases, hashing means to index and store data. A lot of information can be searched and listed easily when using hashing. In security, hashing deals with the encryption of data to produce an output that is not easily reversible (Pieprzyk et al, 2000).

Transformation of characters in a string into a short fixed length value which is a representation of the original string is done during hashing.

Finding items using shorter keys are much easier than using original key.

This makes hashing very useful in locating and retrieving information in databases and encryption algorithms (Cormen et al,2001).

Data structures like stacks, binary trees or lists are different from hashing. This is because data in hash table need not to be reorganized before it is inserted or retrieved.

For items saved or stored in tree or list form,this can be a problem, for large sets of data all nodes of a tree or elements of a list must be travelled through during searching and retrieval. This makes the return time for operations such as insertion and retrieval in such data structures relative to the data set size.

Items (string or integer) are tackled in different ways when they are to be stored in a hash table. Integers are directly hashed to get a key whereas strings need to be converted by using ASCII conventions or the preconditioning technique into an integer value.

The process of preconditioning is the transformation into an integer a string by the use of ASCII conventions. The 26 letters of the alphabets starting from 11 are assigned ASCII values. Numbers starting from 0 to 10 are first.Therefore, the letter A is assigned 11, B is assigned 12 and the letter C is assigned 13 till Z which is assigned 36. Special characters are assigned the numbers 37 and upwards. Such special characters are addition sign (+), subtraction sign (-), equal to sign (=), division sign (/), multiplication sign (*) etc. As such, the conversion of ABC is111213.

It becomes difficult to store in a hash table the value gotten after preconditioning because; the integer value is very large. For this problem to be resolved, a hash process is used to get a shorter integer, and another method is used to map the results to the hash table (Zobel et al, 2001).

2.3Four main components involve in hashing

There are 4 key components involved in hashing:

1. Hash tables
2. Hashing technique and hash functions
3. Collisions
4. Collision resolution strategies

2.3.1 Hash table

This is location on disk or in memory where data records that are hashed using a hash function are kept.The hash table is a data structure divided into equal sized buckets to help permit fast access to elements. Basically, it isa single array dimension indexed by an integer value known as hash index.This is calculated by a function known as hash function. The function help to create a hash index which help to find out which bucket data is to be stored in(Singh and Garg 2009).

Basic problems with hash table creation are;

- How to get a good hash function to distribute data to be inserted uniformly in the hash table.
- How to get an effective collision resolution strategy to resolve collision by computing a different hash index for a data that has the same hash index as an already stored data in the hash table (Wiener and Pinson,1987).

Initializing, inserting, retrieving and deleting of data are typical operations in hash tables.

Implementing hashing

The implementation of hash table is done as array of buckets. If the size of the bucket is *N*, then it numbered 0 to *N* - 1.

In the implementation of hashing, the following operations are performed; **1.** Initialization: to show elements are in the hash table.

2. Insertion: this means tostore information in a hash table by using a hash index. If the data already exist in the hash table, it means the data cannot be saved again. (Some few algorithms permit the replacement of the existing information with the new one).

This brings about the idea of collision resolution strategies.

3. Retrieval: to retrieve information associated with a key given a key *k.*

4. Deletion and updating: removes a key and its information or removes information thatcomes with a keyfrom the hash table.

Hash tables can be used for symbol table in a parser, a dictionary or spellchecker, IP address table for filtering network traffic etc.

2.3.2Hash function

A key is transformed into a hash index (bucket) which represent the position where the value is to be placed or sought by using a hash function.

Hash functions are used in mapping different keys to different storage locations as shown in figure 2.1. But it's practically impossible to write or create a hash function without collision. In a lot of cases, hash tables work well than other data structures.

This is why hash tables are used in most computer software's such as sets, indexing databases, caches etc(Wiener and Pinson1987).

Basically, hash functions are used to transform data into a short length output, which represent the original data's reference. This is important when the entire data is cumbersome to be used.

Linear searching in a list for person's information becomes difficult as the list increases in length. But with hash values, the original data's reference is stored and this helps to retrieve the original data in constant time.

Hash functions are used in cryptography where hash values are generated from an input key. It is very easy to check that the key corresponds to the hash value, but very difficult to create a fake hash value for malicious attack. PGP algorithm uses this principle for its data validation.

Hash functions are also used frequently in compares' of information (detecting duplicate record in a large file, finding liked stretches in a DNA sequence), looking for data in a database etc.

A hash function must be deterministic; this means when used twice on the same information, the hash value produced must be the same. This shows how correct a hash function is. To find an item in a hash table, the same hash value that was generated by the insertion function is needed for the table lookup because it need to find the data in the same bucket it was stored.

Characteristics of a good hash function

The characteristics below are exhibited by a good hash function:

- Minimize collision
- Be easy and quick to compute

- Distribute key values evenly in the hash tables
- Use all the information provided by the key

It is impossible to rehash the hash value to produce the original data.

When keys that is different hashes to the very same address location, it is known as hash collision.

Collision is not preventable but in most special cases, the primary or basic goal is to develop a hash algorithm or function that minimizes collisions because, more collisions cause the application to be slow (Singh and Garg 2009).

Hash functions are functions that can be compared with checksums, error-correcting codes, fingerprints, check digits, randomization functions, ciphers etc. These algorithms work somehow similar, but each of them are designed differently optimized differently and have different requirements.Some uses of hash functions are in Caches, Hash tables, Bloom filters, Protecting data,Finding similar records, Finding duplicate records, Finding similar substrings etc(Wiener and Pinson 1987).

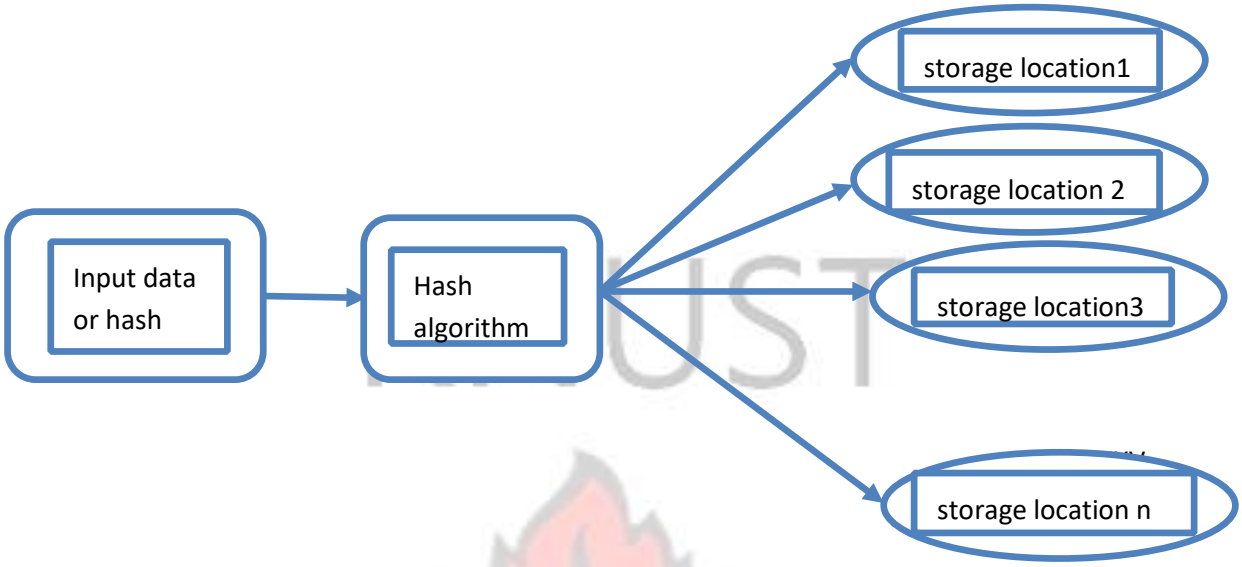


Figure 2.1Hash functions and hash keys hash table

2.3.3Collision

When keys that is different hashes to the very same address location, it is known as hash collision. These objects cannot be placed in the same bucket of the hash table. Hence a collision resolution strategy is used to reposition the second key or object in a different location (Dehne et al, 2006).

2.3.4 Collision resolution strategies

When two records happen to have the same hash value, it is known as collision. But the storage of different records in the same storage space is not possible, hence using collision resolution strategies to find an alternative address space for the different data.

There are several collision resolution strategies. Some common ones are open addressing (linear probing, quadratic probing and double hashing) and chaining (separate chaining and coalesced hashing).

Open addressing

This procedure is again called closed hashing and this is done by probing. Here records are stocked up straight inside an array. When collision occurs, it is resolute by searching or probing through the array locations awaiting the establishment of the record that is sought or an empty slot, which is a sure sign that there is no such record in the table (Main and Wesley, 1999).

- **Linear probing (linear search)**

Here records are stored directly in an array.When collision occurs, it is resolute by searching or probing through the array locations awaiting the establishment of the record that is sought or an empty slot, which is a sure sign that there is no such record in the table.

If a key hashes into aspace which is occupied, what is done is to find a location next to the stored item to store the next item.For example, ifthere exist a bunch of keys such as (89, 18, 49, 58, 9) and there is a need to put these keys in a hash table of size say 10. Figure 2.2 demonstrates the process.

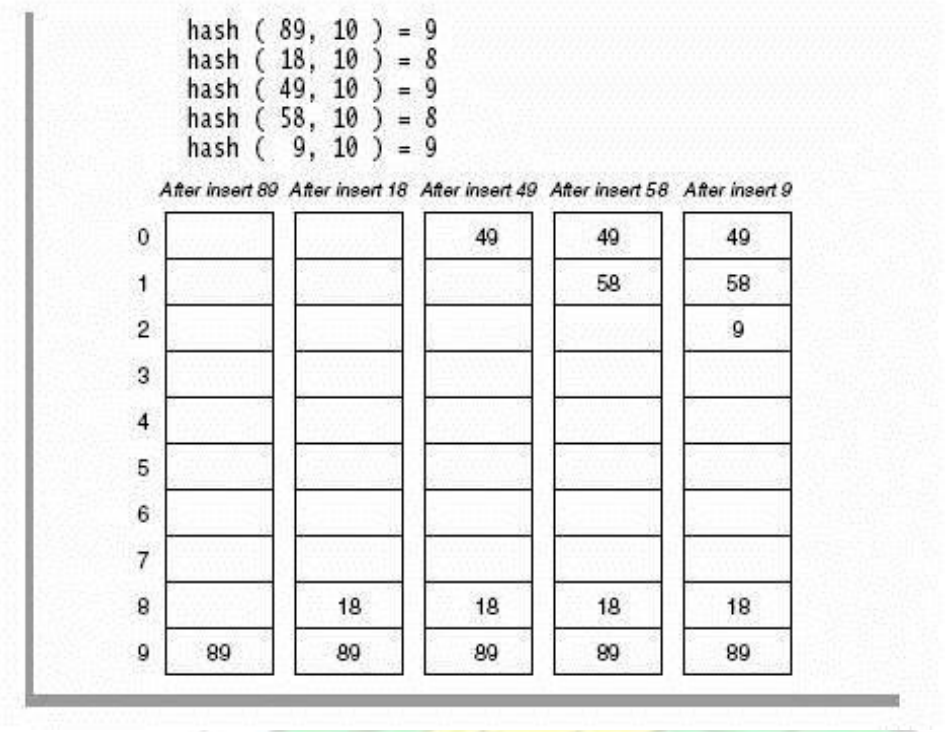


Figure2.2 An example of linear probing collision resolution strategy

The very first collision occurs when the value 49 hashes into a hash index of 9, which is the location where the value 89 is already located. This brings about the need to put the value 49 in a next location that is available. If the array is considered to be in a circle form, the next available location will be 0.This is done by performing the operation (9+1) mod 10. Therefore the value 49 is placed in address location 0. For any other collision that occurs, the next location that is available receives the value.If an element is to be found, for example, 49, the hash code (9) is computed first andcheck in address location 9 will be done. Since the value is not at location 9, there will be a check in address location [(9+1) %

10] = A[0], the number 49 will be found there. If we are looking for 79, the hashcode of 79 = 9 will be computed. We look in A[9], A[(9+1)%10]=A[0], A[(9+2)%10]=A[1], A[(9+3)%10]=A[2], A[(9+4)%10]=A[3] etc. Since A[3] = null, this help us to know the value 79 is not part of the set of numbers(Feldman, 2007).

- **Quadratic probing (nonlinear search)**

The next empty location for items to be stored is easy to calculate in linear probing, this makes linear probing method simple. But when key values computed are that close, it contributes to clustering in the table.

On the occurrence of collisions, quadratic probing help to distribute data well in a hash table. For example, for a hash value H, the next address is calculated by the use of the sequence H + 1, H + 4, H + 9 etc.The hash index is calculated by a quadratic function and the intervals linearly increase(Feldman, 2007).

The figure 2.3 shows the collision resolution using quadratic probing.

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figure 2.3 An example of Quadratic probing collision resolution strategy

Double hashing

Here when collision occurs, the hash value gotten is hashed with another or second hash function to obtain a new hash value where the information will be stored (Kruse and Hall, 1987).

Chaining

Chaining occasionally known as direct chaining; this procedure or method consist of an array reference that have each slot consisting of a link list of records inserted.

 Separate chaining

Data that needs to be at the same address position can be in a link list.

An imitation of the link list is done by the use of record numbers other than actual pointers.

Collision is resolved by placing elements in a link list as in figure 2.4.

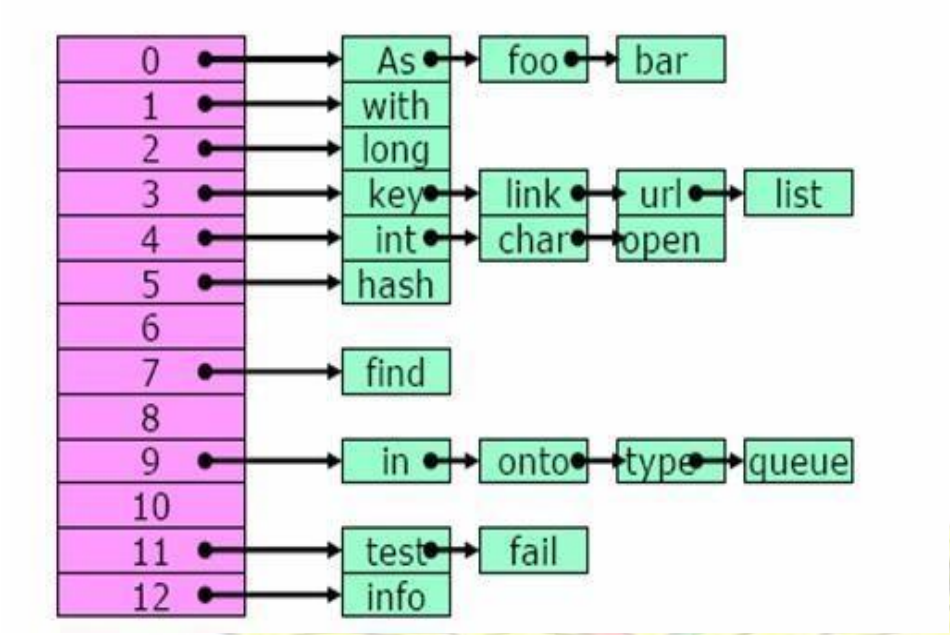


Figure 2.4 An example of separate chaining collision resolution strategy

In figure 2.4,{As, foo, and bar} hashes into address space A[0] of the hash table. A creation of a list of elements which hashes into the same address space is done.

In the same way, other lists in the figure above shows keys which hashes into the same address space. For even distribution of data, a good hash algorithm is obviously in need(Kruse and Hall, 1987).

 coalesced hashing

For this technique, every slot carries an item which was hashed into it, and an item number of another record which hashed to the same address location.

Here, the link list used is stored inside the hash table as shown in figure 2.5, whiles the above methods link list is external and stores only the front pointerof individual list (Kruse and Hall 1987).

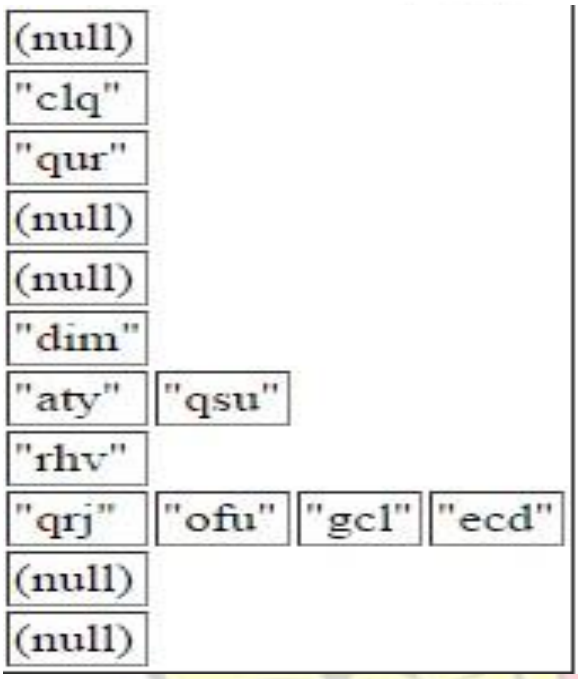


Figure 2.5An example of coalesced chaining collision resolution strategy

This technique is easy to implement,efficient and effective.Open addressing has setbackswhich causeslow performance, both secondary and primary clustering, which results from a search operation which accesses buckets with items of different hash address locations. The searching process can be prolonged by objects with the same address location.

A solution to these problems is coalesced hashing. This collision resolution strategy makesuse of a similar procedure as in separate chaining.Buckets in the actual table are usedrather than the allocation of new linked list nodes. A slot is taken to be the collision slot when is the first free bucket when collision occurs in the hash table.The collision slot and the chain are then linked (Vitter and Chen1987).

2.4 Types of hash algorithms or functions

Many types of hashing functions exist, such as:

- Cyclic redundancy checks e.g. BSD checksum,crc16, crc32peg2, crc64, crc32, SYSV checksum.
- Non cryptographic hash functions (NCHF's) e.g. Pearson hashing, Buz hash function, Jenkins hash, FNV hash, city hash, murmur hash etc. ETC
- Cryptographic hash functions e.g. SHA1, MD5, MD2, , RIPEMD320, SHA-256, SHA-512, TIGER, SPECTRAL HASH, WHIRLPOOL.
- Checksums e.g. Fletcher-8, Fletcher-32, sum8, sum16, sum32, Verhoeffalgorithm, Dammalgoritm etc.

All these functions are grouped into two as shown in figure 2.6.

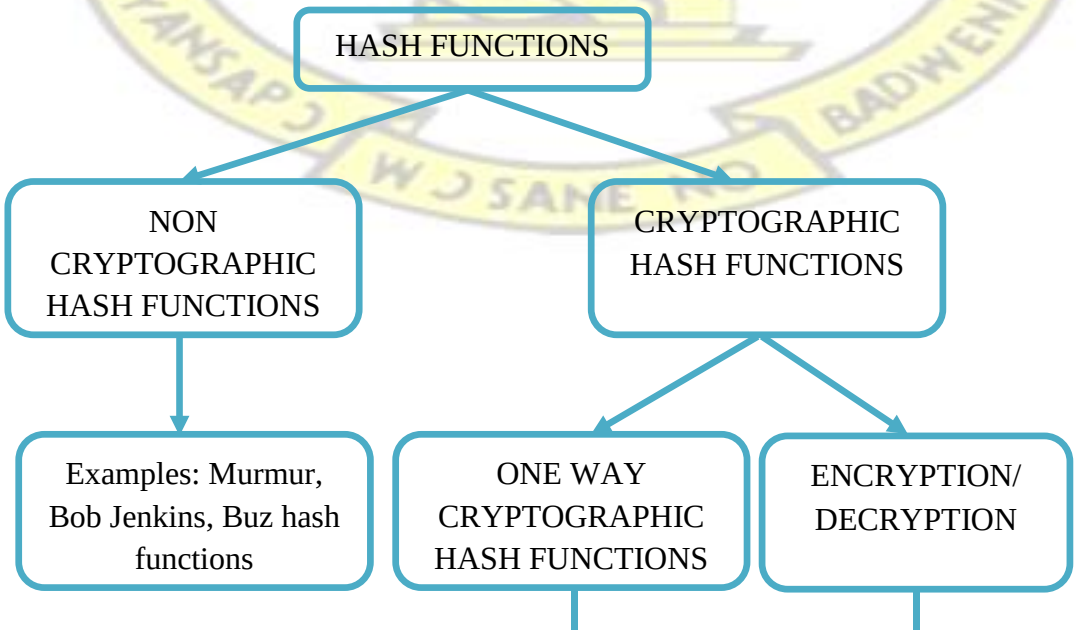


Figure 2.6: Depiction of the various types of Hash Functions

2.5Cryptographic hash functions

Cryptographic hash functions are functions that link a key whose size is arbitrary to a fixed size string. These functions are one way algorithms (they are infeasible to invert).

A message is inputted and a message digest which is represented by the hash index is outputted (Bruse, 2006). There are four basic attributes of a these functions, such as;the quickness to compute a hash index for a message given, the infeasibility to produce the message hashed, based on the hash index created, impossible getting different messages that have equal hash indexand making a little change in a message must cause an extensive change in the hash index in such a way that the latest hash index is not correlated to the earlier hash index.Examples of these functions are SHA 1, MD5, SHA256, HAVALETC,etc (Chad, 2007).

2.6 Non cryptographic hash functions

NCHF's, an integer outputis computed by takinga string as input. Even distribution of data in memory location allocated especially for similar data is a cherished property of this hash function. These hash functions are *not* written in such a way that an attacker can not find collision.In NCHF's, collision must be minimized, computation must be quick and easy, there must be even distribution of keys in the table and the use of all key bits provided in the key.

Cryptographic hash functions are much slower but have this property. Even though NCHF's are fast, is at the expense of not able to stand an attackers attack. These functions are commonly used in hash tables.

Examples of non-cryptographic functions are Pearson hashing, FNV hash, murmur hash, Jenkins hash, buz hash, city hash etc. (Goodrich and Tamassia, 2013).

2.6.1 Bob Jenkins hash function

Bob Jenkins is known to be designing hash functions for table lookup. Bob Jenkins created a multi byte keyed function which is made up of a collection of non-cryptographic hash functions. This function can be used to detect data that are similar in a database and as checksums(Bob, 1997).

There exist variants of bob Jenkins hash functions such as Jenkins's one at a time hash, lookup2, lookup3 and spookyHash.

There are three fundamental stages in Bob Jenkins hash function:

• Combining key length and initialization value to set up an initial state.

• mixing of bits of the keys in 12 byte increments.

• processing of remaining bytes of the key.

Jenkins's one at a time hash

This hash function is the first variant of Bob Jenkins hash function. It was formally published in 1997. The function has three stages as stated above. The One-at-a-Time hash is a considerably simpler algorithm of his design.It quickly reaches avalanche and performs very well. It has been used in several high level scripting languages for their associative array data type as the hash function. Some few bits are mixed weakly in the input data as compared to bits that made up the output hash.

By default, the programming language Perl uses is Bob Jenkins one at a time hash but can be implemented by using Sip hash or Jenkins one at a time hash (Bob, 1997)

Lookup2

This function succeeds Bob Jenkins one at a time. The **lookup2** function is also known as (My Hash). This function is now obsolete because of the other functions that Jenkins has released. It is used in many applications (Bob, 1997).

Lookup2 is found in the following applications;

SPIN model checker- this checker is for detecting error. Researchers Dillinger and Manolios in a paper about this program noted that lookup2 hash function is commonly used in implementing bloom filters and hash tables(Dillinger et al, 2004).

- Netfilterfirewall component of Linux, this has taken the place of a collision sensitive function that existed earlier(*Ayuso, 2006)*.
- Jenkins hash function was used in solving the kalah game application, instead of a more commonly used Zobrist hashing technique that was used; the speed of Jenkins hash on kalahboards and the rule of kalah game which causes a radical alter of the board when performance is low negates the importance ofZobrist incremental hash function (*Irving et al, 2008).*

Lookup3

Lookup3 hash takes in input data in 12 byte chunks. This is very useful when the simplicity of the function is not as useful as speed. For large data, improved speed will be very useful but how complex a function is can cause consequences in speed.

The hashlittle function for a given length and initialization value provided, computes a hash of a single key. For each mixing iteration, the function reduces the key length by 12 bytes. A part of the function that is most computationally expensive is the mixing of the bits of a given key. When a key reaches a length less or equal than 12 bytes the remaining bits are mixed within the hash function after it is extracted (Lalanne et al, 2015).

SpookyHash

In 2011, Bob Jenkins brought into the system a 128 bit hash known as SpookyHash.

SpookyHash is faster compared to lookup3.SpookyHash is a non-cryptographic hash function released into the public domain. It produces 128 bit keys for array byte of any length. 64-bit and 32-bit hash values can also be produce at a similar speed.

The Spooky Hash allows a 128 bit seed. It is given a name SpookyHash because it was released on Halloween.

Reasons to use spookyHash

• spookyHash is fast - For keys that are short it is one byte per cycle, this comes with 30 cycles of cost for startup. For long keys, it is threebytes per cycle, this occupies only one core.

• spookyHash is good - avalanche is achieved for one(1)-bit and two(2)-bit inputs. It is designed to work for any kind or type of key that is made to be like list of arrays of bytes or array of bytes.

When not to use spookyHash

 When there involve an attacker: The reason is that, spookyHash is not cryptographic. When a digest is given, an attacker who is resourceful can create a tool which can give a message that is modified having an equal hash as the message originally sent.Such tools written can be used by an opponent who is not resourceful to perform their actions. Another case not to use spookyHash is that Big-endian machines are not in support of itsnew implementations. Good result can be produced when run on big endians but the

results will differ from the use of little endian machines.By default, machines which do not read unaligned data cannot also run spooky hash.

For keys that are long, the inner loop of the SpookyHash:Mix(),takes in 8 byte input data, performsxor operation, and another xor operation, rotation, and then addition.

Whereas Spooky::Mix() handles keys that are long well, it needs four repetitions formixing finally, and contribute to huge cost of starting cost which makes keys that are short costive. Therefore ShortHash helps in producing a shorter key of 128 bit hash which has a small cost of startup, and Spooky::End() help minimize the mixing costthat is final (Bob, 2012).

2.6.2 Murmur hash function

This is a function that is generally suitable for table lookup (Couceiro et al,2012). This exist in various variants and was created in 2008 by Austin Appleby.

The name is from operations which is simple in sequence and performs a thorough mixing of the bits of a given value - x *= m; x = rotate_left(x,r), performs multiplication and then rotation. This is repeated for about 15 times by using good values for m and r and the value of x will then pseusorandomize.The use of multiplication and rotation has some small

weakness when they are used in creating hash functions. Therefore multiplication, shift and xor operators are used but Murmur is still used instead of Musxmusx (Tanjent,2008).

Whencomparism was made between MurmurHash and other common hash functions, murmur hash had a good performance in the distribution of keys (Tanjent,2008). The current version of MurmurHash is MurmurHash3, this produces 32bit or 128bit hash value.The earlier MurmurHash2 produces 32bit or 64bit hash value.

Two variations of Murmur hash generate 64bit values. That is MurmurHash64A, designed for 64bit processors, and MurmurHash64B, designed for 32bit processors. (Adam, 2012).

Someone who uses MurmurHash2A saw a little bug in the C++ implementation, which causes the C and C++ variants to give different hashes for keys, which the size does not conform to multiples of four. The bug was fixed and the code was updated.

Other people also saw a bug in MurmurHash2; this was that because the 4byte code was repeated a lot of times, it had a high collision chance. This problem was not a problem that could be fixed but does not cause much problem. While this bug was under investigation there emerged a new mix function that was an improvement on the earlier function and was published as MurmurHash3.

Murmurhash3's performance is better than MurmurHash2. There was no repetitive flaw, had variants in 32bits, 64bits and 128bits for x86 and x64 machines. The 128bit x64 variant is much faster (that is over five gigabytes per second on 3 gigahertz core 2).

Murmurhash3 passed all test but failed avalanche (Couceiro et al, 2012).

2.6.3 Buz Hash function

Buz hash function produces up to 2^{32} hash values that are different, but this function uses a lot of Pascal initialization code. In some programming languages, the pascal initialization is done in the code itself;therefore, there will be no need for an initialization call.

For Buz hash function as in pkphash function, added noise is from a random table, but in *pkphash*function, it is from characters in an array. *Buzhash*uses set of 32bit random aliases for every character bit. Because of this, each bit location have one-half of its aliases having one and the other having zero, and these are saved in a table. During hashing, the random aliases of every character bit is xor-ed.Xor-ing have the chance of inverting every character bit by 50%.

For the Java`buzhash, it works for keys that have short length than 65 key bits; this is because it was designed for such keys. Most programs are not limited much because of this because most programs uses character bits less than 64` (Uzgalis,1995).

Buz function was improved by Robert Uzgalis. This hash function is effective and efficient and was developed on the bases of BuzHash function by RobertUzgalis(Uzgalis,2009).

2.7Summary and conclusion

The research on the existing NCHF's shows that, all the hash functions have not been compared together before hence the need to compare these hash functions together with mine in order to contribute to the enhancement of hash functions.

CHAPTER THREE

RESEARCH METHODOLOGY

3.0 Introduction

Methodology refers to techniques that are used to conduct research.

This research is needed because much research has not been done on non-cryptographic hash functions based on recent study done.

NCHFs have a lot of applications, ranging from dictionary lookup, databases and compilers to computer networks and videogames. Major corporations use most NCHF in developing commercial products. This success demonstrates that, hashsystems provide extreme efficient search over an unsorted set. But little research has been done on these functions.

This bring about the need to evaluate and enhance common NCHF's using attributes such as distribution of outputs, collision resistance, speed (usage of CPU time), avalanche effect, and quality of function (Goodrich and Tamassia,2013). Anew and efficient hash function will be developed and strengths and weaknesses identified.The results of this research must helpengineers and practitioners in making decisions regarding the function to choose and use.

3.1 Research strategy

The empirical research in this study is interested in evaluating various popular non cryptographic hash functions in terms of; distribution of outputs, collision resistance, speed (usage of CPU time), avalanche effect, and quality of function; and the outcome will be observed and collected for further analysis.

The implementation and development of algorithms will be done at the laboratory by making use of MySQL and Netbeans 6.9 (JAVA programming Language).

Experimental research strategy will be employed for this research.

An experiment is a procedure that is orderly carried out with the aim of verification, refutation, or establishment ofa hypotheses' validity. Basing on the above definition, research of experimentation has its goal as testing a new or existing hypothesis, which

means expectation of the way particular phenomenon or process works. The results of a careful and well conducted experiment help disprove or support the hypothesis. As such, if the experiment is well and carefully conducted, the result will either support or disprove the hypothesis.

3.2 Research Approach

This study or research is quantitative. Quantitative Research refers to researches that emphasize on quantities and measurements.

3.3 Data Collection: Experiment and Observation

A TEST SUITE that was developed using Netbeans 6.9 (JAVA programming Language) and MySQL will be used to conduct the research on a computer system.

In this research,non-Cryptographichash functions such as Bob Jenkins Spookyhash which will be referred to as Bob Jenkins hash, Murmur3 which will be referred to as Murmur hash

and an improved Buz function which will be referred to us Buz hash function throughout this document will be run on the VIVIAN TEST SUITE using data (keys) to checkpopular hash functions;distribution of outputs, collision resistance, speed (usage of CPU time), avalanche effect, and quality of functionto prove the performance and efficiencyof the various NCHF's.This test suite uses the separate chaining collision resolution strategy to resolve collisions. In order to prove the stated hypothesis, data will then run several times and collected for further analysis.

Observation technique will be employed in data collectionof this research. Observation means taking a careful or critical look.

Therefore, by the definitions above, Experimentation and Observation will be the best data collection methods to be used in this study or research. Experimentwill be done ondistribution of output, collision resistance, speed (usage of CPU time), avalanche effect, and quality of some common non Cryptographic hash functions, with different key sizes

and types by testing them on the VIVIAN TEST SUITE and afterwards make a full observation of the outcome and collect that data for further analysis.

3.4 Framework for Data Analysis

This research analyzes collected data from the conducted experiment, by doing comparing and contrasting of the obtained results from the various non-cryptographic hash functions. The experiment will be conducted using VIVIAN TEST SUITE (VHASHER). The tool to be used to critically and statistically analyze various non-cryptographic hash functions will be VIVIAN TEST SUITE.

3.5 Vivian hash function

Vivian hash function is named after its inventor. This hash function requires a common initial value and an offset. It uses bitwise operators such as shift, bitwise AND, bitwise XOR and bitwise OR. All these are mixed up with the individual characters of the word to be hashed.

Steps for mixing individual characters of a word

Forthe mixing of individual characters, the ASCII value of the new character (ANC) is left shift with the hash value from previous mix (HPM) which initially holds the offset, and the result is stored as intermediate result 1 (IR1).

The initial value is mixed with the HPM using the AND operator and the result is stored as intermediate result 2 (IR2).

IR1 exclusive-OR (XOR) IR2 and the result is stored as intermediate result 3 (IR3).

HPM is left shift with the initial value and the result is stored as intermediate result 4 (IR4).

IR3 OR IR4 and the resulting value is stored as intermediate result 5 (IR5).

ASCII value of the new character (ANC) is XOR-ed with the initial value and the resulting value is stored as intermediate result 6 (IR6).

IR5 and IR6 are XOR-ed and the resulting value is kept in the HPM.

This mixing of individual characters is done until the length of the array is reached, and the results represent the final hash used to determine the memory location where the data is to be saved or stored.

This memory address or location is calculated by finding modulo of the hash value using the hash table size.

The individual characters are mixed well enough to help achieve avalanche, better distribution, reduce collision, quality and better the performance of the hash function.

Generally, this algorithm or function in terms of distribution, collisions, performance, avalanche and quality,should be the first used for applications involved in using hash

function to perform table lookup because of its better properties. The following abbreviations are used in the module and flowchart below; HPM : Hash value from previous mix. This initially holds the offset.

ANC: ASCII value for the next character

IR : Intermediate representation

VICM : Module of Vivian hash function individual character mixing (VICM)

FHV : Final hash value

MI : Memory index

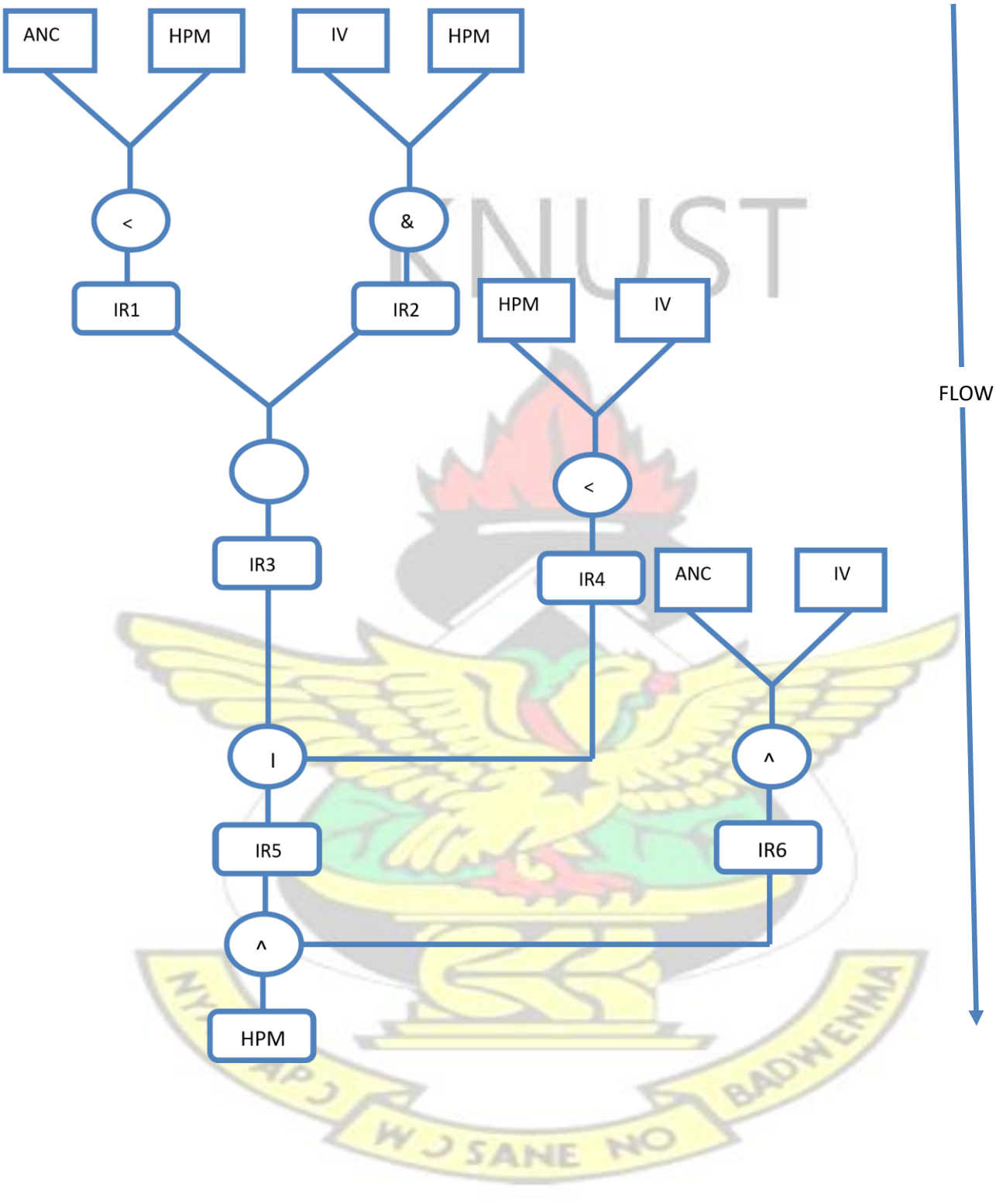


Figure 3.1A module of Vivian hash function individual character mixing (VICM)

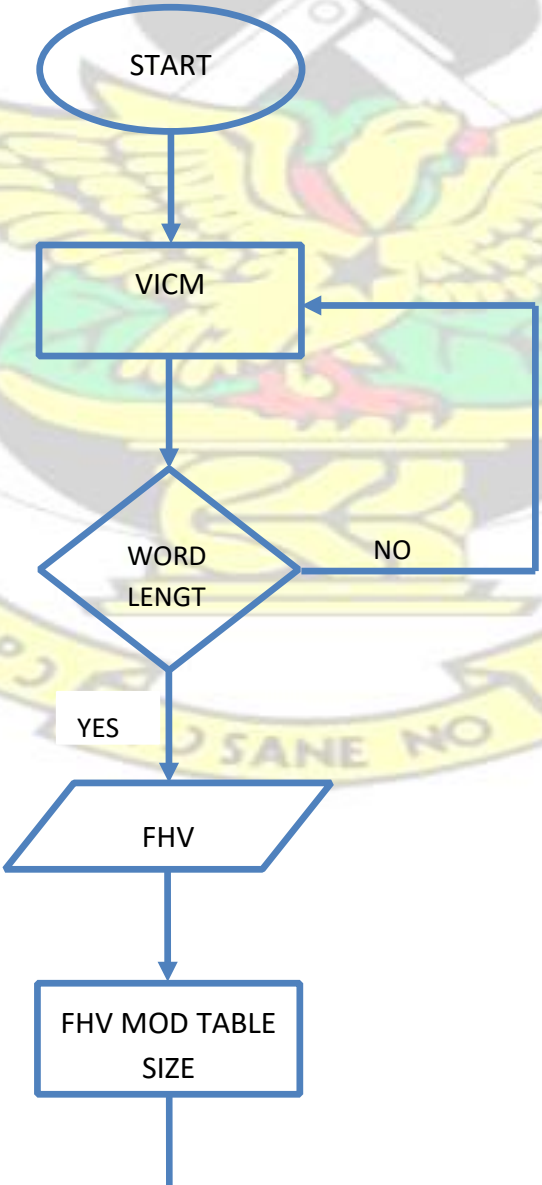


Figure 3.2A flowchart of how Vivian hash function works.

3.6 Vivian test suite (VHASHER)

VHASHER is a java-based program that was designed to test properties such as percentage distribution, number of collisions, performance, qualityand percentage avalanche of Bob Jenkins, Murmur, Buz and Vivian hash functions.

- Percentage distribution

This shows how evenly data is spread out in the memory space allocated for the data to be saved or stored.

- Number of collisions

The number of keys or data that hashes to the same address space.

- Performance or speed

How fast the hash function can consume input.

- Quality

This test the quality of hash function based on the various properties.

- Percentage avalanche

Deals with how each individual key bit, contribute to a change in hash value produced. The VHASHER works by using hash table to implement a dictionary. The dictionary stores words and their synonyms.

Operations such as inserting or saving, updating, finding and deleting of data are performed by using the various hash functions to hash the data or key or word, and the resulting hash value becomes the memory index or location where that particular data is to be stored.

MySQL was used to create database for the various hash functions. Each database consist of an array of separately chained memory indexes, where data is stored (data is stored in these memory by the various hash functions hashing the key or data provided and the resulting hash index represent space in memory where data is to be stored.) , avalanche table which contains percentage avalanche of each data entered and the statistical table which contains memory indexes, return time of data operated upon (this is gotten from the

system time spent to hash words and perform operations such as inserting, updating, finding and deleting) and the number of words stored in the various separately chained memory indexes (which can help know the number of words that hashed to the same memory location).

How the VHASHER works is explained on the next page.

3.6.1 How the v-hasher works

Main form

The VHASHER has a main form, where the user can choose operations to perform operations such as save, update, find and delete. The user can also choose properties to view properties of Bob Jenkins, Murmur, Buz and Vivian hash functions for previous data stored in the hash table (dictionary). The properties form consists of percentage distribution, number of collisions, performance or speed, percentage avalanche and quality.

Operations form

The operations form can help save, update, find and delete data from the hash table or dictionary. These operations are performed by clicking the save, update, find and delete buttons.

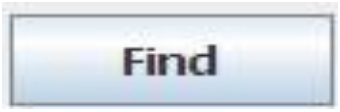
Save button

This button when clicked help save data (a word and its synonym) in the hash table or dictionary after entering the word and its synonym in the textbox provided.

Update button

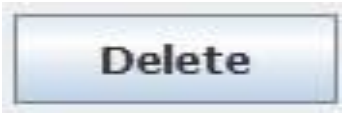
This button when clicked help update the synonym of a word in the dictionary, by entering the word to be updated in the word textbox and the update of the word's synonym in the synonym textbox.

Find button



This button when clicked after entering a word to be found in the word text box just above the button shows the synonym of the word entered in a message box if only the word exist in the dictionary. Else a message box display that, the word does not exist in the dictionary.

Delete button



This button when clicked after entering a word to be deleted in the word text box just above the button, the word will be deleted from the dictionary or hash table if it exists in there.

Properties form

The properties form shows the percentage distribution, number of collisions, performance, quality, and percentage avalanche properties of Bob Jenkins, Murmur, Buz and Vivian hash functions. Percentage distribution property shows how evenly data is spread out in the memory space allocated for the data to be saved or stored, the number of collisions property shows the number of keys or data that hashes to the same address space, the performance or speed property shows how fast the hash function can consume input , percentage avalanche property deals with how each individual key bit, contribute to a change in hash value produced and the quality property test the quality of hash function based on the other various properties.

Statistical details form

This shows the statistical details of Bob Jenkins, Murmur, Buz and Vivian hash functions and was loaded from the statistical tables of the various databases.

This contains the content of the statistical tables from the various databases of the hash functions (Bob Jenkins, Murmur, Buz and Vivian). Columns that can be found in here are the memory locations in the hash tables, the number of words each hash index or memory location contains and the time spent in hashing words by the hash functions and also performing operations such as save, update, find and delete. These data is used to determine properties such as percentage distribution, number of collisions, performance and quality of the various hash functions.

Avalanche details form

This shows the avalanche details of Bob Jenkins, Murmur, Buz and Vivian hash functions. The content of this form is loaded from the avalanche tables in the various databases of Bob Jenkins, Murmur, Buz and Vivian hash functions.

It contains the words entered in the dictionary and its percentage avalanche.

3.6.2 How operations are performed using the VHASHER

Saving

When the save button is clicked, it checks whether the textboxes are not empty or the word is not already in the dictionary. After this, each hash function calculates the memory index where data is to be stored. Based on the memory index, the percentage avalanche tables, the number of words columns and time spent columns of the statistical tables of Bob Jenkins, Murmur, Buz and Vivian hash functions databases are updated. Otherwise a message is popped up to show the word already exists or the textboxes are empty.

Updating

When the update button is clicked, it checks whether the word is in the dictionary. Based on the memory index calculated by each hash function, the synonym of the word is updated and the time spent column of the statistical tables of Bob Jenkins, Murmur, Buz and Vivian hash functions databases are updated. If the word is not in the dictionary or textboxes were empty, a message pops up to show that.

Finding

When the find button is clicked, it checks whether the word is in the dictionary. If the word is found in the dictionary, the synonym of the word is shown in a message box and the time spent columns of the statistical tables of Bob Jenkins, Murmur, Buz and Vivian hash functions databases are updated based on the memory indices calculated by the various

hash functions. If the word is not in the dictionary or textbox is empty, a message pops up to show that.

Deleting

When the delete button is clicked, it checks whether the word is in the dictionary. If the word is found in dictionary, both the word and its synonym are deleted from the dictionary. The number of words and time-spent columns of the statistical tables of Bob Jenkins, Murmur, Buz and Vivian hash functions databases are updated based on the memory indices calculated by the various hash functions. If the word was not in the dictionary or textbox was empty, a message pops up to show that.

3.7 Various test performed using the VHASHER

3.7.1 Percentage distribution test

To calculate percentage distribution, the number of words column of the statistical table of each functions database is loaded into an array .A counter is set to zero which is increased by 1 whenever the value in an array index is greater than zero (0). The data in the array is then added together and after the summation, the value of the counter is used to find how a hash function have distributed data in its memory space allocated by taking the length of the array as 100% as seen in equation 1.

Counter=0. count=$\sum_{i=0}^{m-1} counter + 1$

,where x[i] >0.

Therefore percentage distribution=$\frac{count}{length\ of\ array}$*100%

equation 1

Take a bunch of keys, hash them and store in the database using VHASHER which uses Bob Jenkins, Murmur, Buz and Vivian hash functions. Data will be stored in the various

memory locations based on the memory index or hash index calculated by the hash functions. This shows how evenly the hash values are spread over the memory space allocated for the hash table. Sets of keys are hashed a lot of times as shown in table 3.1 and average percentage distribution calculated. When a hash function is able to distribute data in memory space allocated well, it will help in efficient use of memory space allocated to be used. On the other hand, when data is not well distributed, it will result in wasting the memory space allocated.

Table 3.1 Percentage distribution table

Bob jenkins	Words less than 7 characters	Up to half of table	12.9	31.7	37.6	39.6	40.6	Avg= 40.5	Favg= 41.0
		Up to full table	41.6	44.6	46.5	48.5	61.4		
	Words more than 7 characters	Up to half of table	24.8	27.7	32.7	36.6	40.6	Avg= 41.3	
		Up to full table	44.6	48.5	50.5	48.5	58.4		
	Mixed length of words	Up to half of table	20.8	26.7	34.7	35.6	37.6	Avg= 41.1	
		Up to full table	38.6	47.5	51.5	55.4	62.4		
Murmur	Words less than 7 characters	Up to half of table	11.9	26.7	31.7	34.7	35.6	Avg= 35.6	Favg= 34.9
		Up to full table	37.6	38.6	41.6	42.6	54.5		
	Words more than 7 characters	Up to half of table	22.8	24.8	25.7	25.7	27.7	Avg= 30.5	
		Up to full table	34.7	34.7	36.6	33.7	38.6		
	Mixed length of words	Up to half of table	20.8	26.7	30.7	34.7	36.6	Avg= 38.6	
		Up to full table	37.6	44.6	49.5	49.5	55.4		

Buz	Words less than 7 characters	Up to half of table	12.9	30.7	36.6	40.6	41.6	Avg= 40.5	
		Up to full table	42.6	42.6	46.5	49.5	61.4		
	Words more than 7 characters	Up to half of table	25.7	27.7	31.7	35.6	38.6	Avg= 40.1	Favg= 40.3
		Up to full table	44.6	45.5	48.5	47.5	55.4		
	Mixed length of words	Up to half of table	18.8	25.7	32.7	34.7	38.6	Avg= 40.4	
		Up to full table	39.6	47.5	51.5	53.5	61.4		
Vivian	Words less than 7 characters	Up to half of table	13.9	32.7	39.6	40.6	42.6	Avg= 42.5	Favg= 42.3
		Up to full table	42.6	45.5	48.5	51.5	67.3		
	Words more than 7 characters	Up to half of table	26.7	29.7	34.7	37.6	39.6	Avg= 40.6	
		Up to full table	43.6	45.5	48.5	46.5	53.5		
	Mixed length of words	Up to half of table	21.8	29.7	36.6	38.6	40.6	Avg= 43.8	
		Up to full table	42.6	49.5	55.4	57.4	65.3		

3.7.2 Number of collisions test

Collision occurs when keys that are different hashes to the very same address location.

Collision is not preventable but in most special cases, the primary or basic goal is to develop a hash algorithm or function that minimizes collisions.

To get the number of collisions that occur when using a particular hash algorithm or function, the number of words column of the statistical table of each functions database is loaded into an array. The value in each array location where the value in the array index is

greater than 1 is summed up. A counter is set to zero and increased whenever the value in an array index is greater than 1(this is done because the first time an item or data was stored at that memory location, there is no collision. Therefore that needs to be subtracted to get the actual number of collisions) as seen in equation 2.

Counter=0

$$sum = \sum_{i}^{m-1} x[i], counter + 1, where\ x[i] > 1$$

Number of collisions=sum-counter equation 2

The smaller the number of collisions a hash function generates the faster and more efficient it is. When there is less number of collisions, it will help result in efficient use of memory space and also contribute to reduced return time when operations such as saving, updating, finding and deleting are performed. This is because the time needed to resolve collision using collision resolution strategy will reduce, and time needed to search for data when separate chaining collision resolution strategy is used will reduce.

But when there are more collisions, more time will be needed to perform the operations above, which will result in increased return time. Different sets of keys are hashed at different times and the average number of collisions calculated as shown in table 3.2.

Table 3.2 Number of collisions table

Bob jenkins	Words less than 7 characters	Up to half of table	1	6	10	13	14	Avg= 17	Favg= 16
		Up to full table	15	16	22	27	42		
	Words more than 7 characters	Up to half of table	6	6	8	10	12	Avg= 13	
		Up to full table	16	16	18	15	23		
		Up to half of table	3	6	9	12	14	Avg=	

	Mixed length of words	Up to full table	15	22	28	29	44	18	
Murmur	Words less than 7 characters	Up to half of table	2	11	16	18	19	Avg= 22	Favg= 22
		Up to full table	19	22	27	33	50		
	Words more than 7 characters	Up to half of table	7	8	14	20	24	Avg= 23	
		Up to full table	25	29	31	29	41		
	Mixed length of words	Up to half of table	3	6	12	12	14	Avg= 20	
		Up to full table	16	24	29	34	53		
Buz	Words less than 7 characters	Up to half of table	1	7	11	12	13	Avg= 17	Favg= 17
		Up to full table	14	18	22	26	42		
	Words more than 7 characters	Up to half of table	5	6	9	11	14	Avg= 14	
		Up to full table	16	19	20	16	26		
	Mixed length of words	Up to half of table	5	7	11	13	13	Avg= 19	
		Up to full table	15	22	28	31	45		
Vivian	Words less than 7 characters	Up to half of table	0	5	8	12	12	Avg= 15	Favg= 15
		Up to full table	14	15	20	24	36		
	Words more than 7 characters	Up to half of table	4	4	6	9	13	Avg= 14	
		Up to full table	17	19	20	17	28		
	Mixed length of words	Up to half of table	2	3	7	9	11	Avg= 16	
		Up to full table	12	20	24	27	41		

3.7.3 Performance or speed Test

How fast can the hash function consume input i.e. speed

This ismeasured by using the systems time to determine how fast a hash function consumes data and the time the various hash functions take to perform operations such as save, update, find and delete in milliseconds and this is stored in the time column of the statistical table depending on the memory locations a particular data. This is summed up using equation 3 by loading the time spent column values in an array. The average time is calculated and that is loaded on the properties form.

To determine how fast a hash function can consume data, create a large block of random data and measure how long it takes to hash it. Due to caching and interrupts and other sources of noise in an average PC, repeat the test a large number of times to ensure we get a clean run.Keys are hashed number of times and average speed for various hash functions calculated as shown in table 3.3. When more time is spent by a hash function in hashing a word and performing operations (save, update, find and delete), it will result in increasing return time when it is used in running applications but when less time is used in hashing a word and perform operations by a hash function, it will result in reducing return time which will contribute to increased productivity or work done. The equation below is used to calculate the return time.

$$sum = \sum_{i}^{m-1} x[i] \qquad \text{Equation 3}$$

Table 3.3 Performance (t/ms) table

Bob jenkins	Words less than 7 characters	Up to half of table	11047	32094	39140	43156	45156	Avg= 47459	Favg= 46041
		Up to full table	47156	51156	58172	62218	85296		
	Words more than 7 characters	Up to half of table	24156	27156	33187	38202	40264	Avg= 40781	
		Up to full table	43346	45376	48391	49296	58437		
	Mixed length of words	Up to half of table	22031	30047	37141	40157	44157	Avg= 49884	
		Up to full table	47157	57236	66251	70267	84391		
Murmur	Words less than 7 characters	Up to half of table	12031	29141	36188	39220	41220	Avg= 44220	Favg= 45132
		Up to full table	43220	47220	53252	57331	83376		
	Words more than 7 characters	Up to half of table	24095	27095	34095	39111	45111	Avg= 45117	
		Up to full table	49765	52780	53827	55127	70159		
	Mixed length of words	Up to half of table	17158	25173	33204	36219	38251	Avg= 46059	
		Up to full table	41251	53298	61330	65345	89362		
Buz	Words less than 7 characters	Up to half of table	391	954	1154	1279	1310	Avg= 1364	Favg= 1289
		Up to full table	1341	1402	1590	1761	2462		
	Words more than 7 characters	Up to half of table	750	843	982	1089	1229	Avg= 1309	
		Up to full table	1518	1595	1657	1494	1930		
	Mixed length of words	Up to half of table	423	643	877	910	1001	Avg= 1194	
		Up to full table	1064	1349	1520	1630	2519		
Vivian	Words less than 7 characters	Up to half of table	294	855	1058	1184	1216	Avg= 1292	
		Up to full table	1263	1358	1575	1731	2388		

	Words more than 7 characters	Up to half of table	657	703	904	1042	1151	Avg= 1249	Favg= 1249
		Up to full table	1498	1592	1655	1400	1885		
	Mixed length of words	Up to half of table	477	681	897	960	1022	Avg= 1207	
		Up to full table	1069	1364	1519	1612	2470		

3.7.4 Avalanche Test

Do all key bits affect all hash bits equally? Does flipping a key bit cause all hash bits to flip with a 50/50 probability?

The percentage avalanche was calculated by storing each hash value a word or data produces whenever it add each character during mixing in an array. After the final hash value, the array is compared to find how many characters caused a change in its preceding hash value. This value is stored by setting a counter to zero and increasing whenever a succeeding hash value is different from its preceding hash value. By taking the length of the array (which is equal to the length of data) to be 100%, equation 4 is then used to find the percentage of the value in the counteror sum. This value is then stored in the avalanche table andthe average avalanche is calculated from this table.

100% indicates perfect 50/50 probability, 0% indicates that the output bit either never or always flipped. Higher the percentage avalanche contributes to reduced returnand efficient use of memory space. This is because, it help spread data in memory and time needed to resolve collision and perform operations such as save, update, find and delete will reduce.

The average percentage avalanche of hashes of various keys is calculated as shown in table

3.4.

$$sum = \sum_{i}^{m-1} x[i], where\ x[i] is\ not\ equal\ to\ x[i+1]$$

Therefore percentage avalanche=$\frac{sum}{length\ of\ array}$*100%

Equation 4

Table 3.4Percentage avalanche table

Bob jenkins	Words less than 7 characters	Up to half of table	99.1	97.8	97.5	97.5	97.6	Avg= 97.7	Favg= 85.6
		Up to full table	97.7	97.6	97.4	97.3	97.7		
	Words more than 7 characters	Up to half of table	69.8	70.1	68.3	68.9	69.6	Avg= 69.7	
		Up to full table	70.3	69.8	69.8	70.2	70.3		
	Mixed length of words	Up to half of table	92.0	92.2	89.2	87.5	86.5	Avg= 89.3	
		Up to full table	86.6	86.3	87.2	87.2	98.0		
Murmur	Words less than 7 characters	Up to half of table	34.7	35.8	34.6	34.5	34.7	Avg= 34.6	Favg= 27.9
		Up to full table	34.7	34.4	33.9	33.8	34.5		
	Words more than 7 characters	Up to half of table	19.9	20.0	19.5	19.6	19.8	Avg= 19.8	
		Up to full table	20.0	19.8	19.8	19.9	20.0		
	Mixed length of words	Up to half of table	31.0	30.4	29.0	28.2	27.7	Avg= 29.3	
		Up to full table	27.6	27.4	28.3	28.3	34.9		
Buz		Up to half of table	100	100	100	100	100	Avg=	

	Words less than 7 characters	Up to full table	100	100	100	100	100	100	
	Words more than 7 characters	Up to half of table	100	100	100	100	100	Avg= 100	Favg= 100
		Up to full table	100	100	100	100	100		
	Mixed length of words	Up to half of table	100	100	100	100	100	Avg= 100	
		Up to full table	100	100	100	100	100		
Vivian	Words less than 7 characters	Up to half of table	100	100	100	100	100	avg= 100	
		Up to full table	100	100	100	100	100		
	Words more than 7 characters	Up to half of table	100	100	100	100	100	avg= 100	Favg= 100
		Up to full table	100	100	100	100	100		
	Mixed length of words	Up to half of table	100	100	100	100	100	avg= 100	
		Up to full table	100	100	100	100	100		

3.7.5 Quality test

To test for quality of the hash functions, the number of words column of the statistical table of each functions database is loaded into an array and a formula proposed by Red Dragon Book was used for evaluating hash function quality as seen in equation 5:

$$\sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)}$$

Equation 5

Where b_j represents number of data or items in j-th slot, m represents the total number of slots, and n represents the total number of data or items. The sum of $b_j(b_j + 1) / 2$ gives the number of buckets the program should visit to find the required value. The denominator (n/ $2m$) ($n + 2m - 1$) is the number of slots visited for an ideal function that puts each item into a random slot.

Actually, a good hash function has quality between 0.95 and 1.05. If the quality is high, it means the function has a degraded performance and is not efficient. If the quality is less, it means the function has a good performance and is more efficient.The lesser the number of collisions, the more efficient the hash function. This is because, it will contribute to even distribution of data in memory space allocated and this means there is efficient use of memory and also contribute to reduced return time because, the time that will be spent to resolve collision using collision resolution strategy and perform operations such as save, update, find and delete will reduce. But when there is more collisions, memory space allocated will not be efficiently used and return time will increase because time to resolve collisions and perform operations as stated above will increase. Sets of keys are run and the average quality of various hash functions calculated as shown in table 3.5. The resulting quality of a hash function is shown of the properties form of the VHASHER.

Table 3.5 Quality table

Bob jenkins	Words less than 7	Up to half of table	1.01	1.00	1.00	1.01	1.02	Avg=	

	characters	Up to full table	1.02	1.00	1.04	1.07	1.02	1.02	
	Words more than 7 characters	Up to half of table	1.07	1.04	1.02	1.01	0.99	Avg= 1.00	Favg= 1.02
		Up to full table	0.99	0.96	0.97	0.98	0.97		
	Mixed length of words	Up to half of table	1.01	1.07	1.03	1.05	1.04	Avg= 1.03	
		Up to full table	1.03	1.04	1.04	1.01	1.00		
Murmur	Words less than 7 characters	Up to half of table	1.14	1.20	1.20	1.17	1.16	Avg= 1.18	Favg= 1.23
		Up to full table	1.14	1.14	1.16	1.20	1.25		
	Words more than 7 characters	Up to half of table	1.14	1.18	1.40	1.56	1.55	Avg= 1.40	
		Up to full table	1.35	1.39	1.40	1.49	1.53		
	Mixed length of words	Up to half of table	1.01	1.02	1.10	1.06	1.07	Avg= 1.10	
		Up to full table	1.09	1.08	1.11	1.17	1.26		
Buz	Words less than 7 characters	Up to half of table	1.01	1.00	1.01	0.99	0.99	Avg= 1.02	Favg= 1.04
		Up to full table	1.02	1.05	1.04	1.05	1.03		
	Words more than 7 characters	Up to half of table	1.04	1.04	1.10	1.07	1.08	Avg= 1.04	
		Up to full table	1.01	1.03	1.01	1.01	1.00		
	Mixed length of words	Up to half of table	1.12	1.07	1.07	1.06	1.03	Avg= 1.05	
		Up to full table	1.03	1.03	1.03	1.04	1.05		
Vivian	Words less than 7 characters	Up to half of table	0.94	0.98	1.00	1.04	1.02	Avg= 1.00	Favg= 0.99
		Up to full table	1.03	1.02	1.02	1.01	0.97		
	Words more than 7 characters	Up to half of table	0.98	0.96	0.96	0.99	1.04	Avg= 1.01	
		Up to full table	1.05	1.06	1.04	1.00	1.04		
		Up to half of	0.97	0.94	0.96	0.98	0.98		

	Mixed length of words	table						Avg= 0.97	
		Up to full table	0.98	0.99	0.96	0.97	0.99		

NOTE

Avg as found in tables 3.1, 3.2, 3.3, 3.4 and 3.5 represents intermediate averages for values got from running different length of words.

Favg represents final averages for values got from finding averages of the various Avg's.

CHAPTER FOUR

DISCUSSIONS

4.0 Introduction

Chapter four deals with findings, explanations and comparisons of the hash function developed and other hash functions that already exist.

A summary of results in tables 3. 1, 3.2, 3.3, 3.4 and 3.5 which can be found in table 4.1 is used for the discussion.

Table 4.1 A table of hash functions and the various properties

		Properties of hash functions				
		Percentage distribution (%)	Number of collisions	Performance or speed (ms)	Percentage avalanche (%)	quality
hash functions	Bob Jenkins	42.0	16	46041	85.6	1.02
	Murmur	34.9	22	45134	27.9	1.23
	Buz	40.3	17	1289	100	1.04
	Vivian	42.3	15	1246	100	0.99

4.1 Percentage Distribution

When the hash table is well distributed, it helps in efficient use of memory space allocated for the hash table. As proportion of areas that are unused in the hash table increases, it does not necessarily reduce search cost. This results in wasting memory.

This means that, Vivian hash function efficiently make use of memory space allocated to the hash table with an average % distribution of 42.3%, followed by Bob Jenkins of 41.0%, 40.3% for Buz and 34.9% hash table distribution for Murmur hash function as shown in figure 4.1. Murmur hash had less distribution as can be seen in the figures above.

To make efficient use of memory space when running applications, Vivian hash will be the best option to use followed by Bob Jenkins, Buz and Murmur hash function.

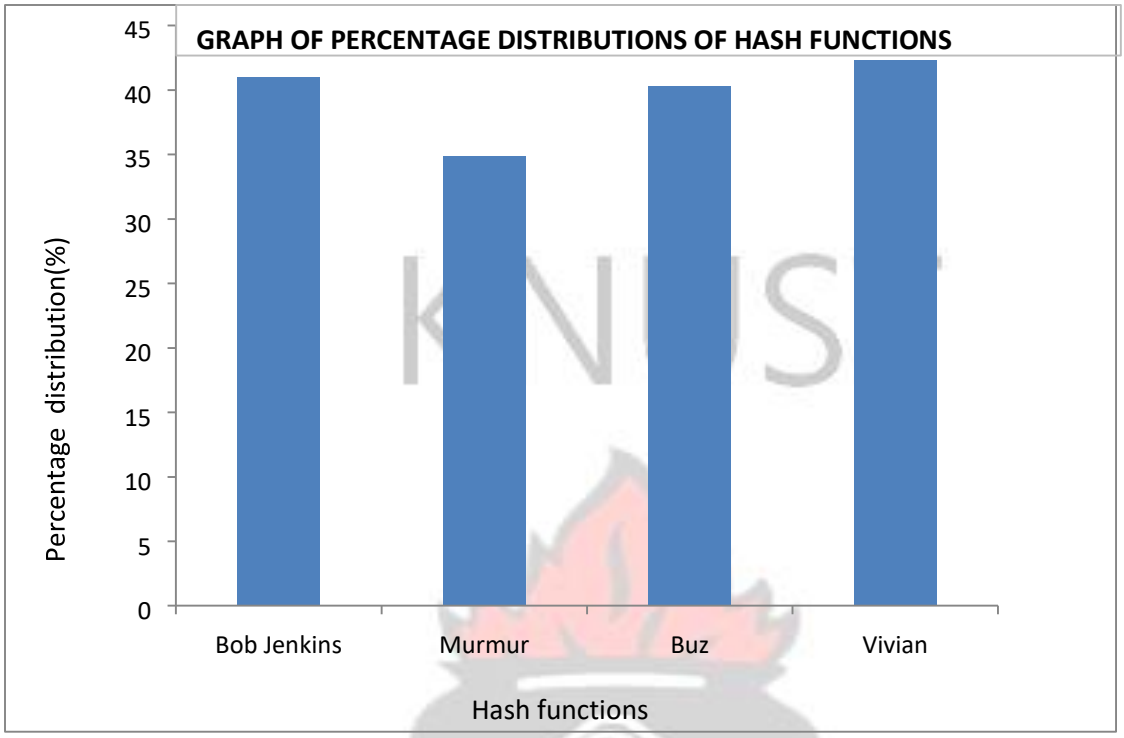


Figure 4.1 A graph of percentage distribution of hash functions

4.2 Number of Collisions

The total number of operations that is required to resolve collision (i.e. collision resolution strategies) linearly scales to the number of keys mapping to the same slot or bucket. More collisions results in degrading the performance or the efficiency of the hash function significantly. Non cryptographic hash functions deals with operations such as insertion, find or look up, delete and update of data. When there are much collisions, there will be much time involve in performing these operations which will surely degrade the efficiency and effectiveness of the hash function.

The smaller the number of collisions a hash function generates the faster and more efficient it is. When there is less number of collisions, it will help result in efficient use of memory

space and also contribute to reduced return time when operations such as saving, updating, finding and deleting are performed. This is because the time needed to resolve collision using collision resolution strategy will reduce, and time needed to search for data when separate chaining collision resolution strategy is used will reduce. But when there are more collisions, more time will be needed to perform the operations above, which will result in increased return time and wasting memory space.

On the average, Vivian hash function had the lowest number of collisions of 15, followed by Bob Jenkins with 16 number of collisions, 17 number of collisions for Buz and Murmur hash function recording the highest number of collisions of 22 as shown in figure 4.2. This shows that Vivian hash function will help make efficient use of memory space and reduce return time when used to run applications. This will help increase work output or productivity followed by Bob Jenkins, Buz and then Murmur hash function.

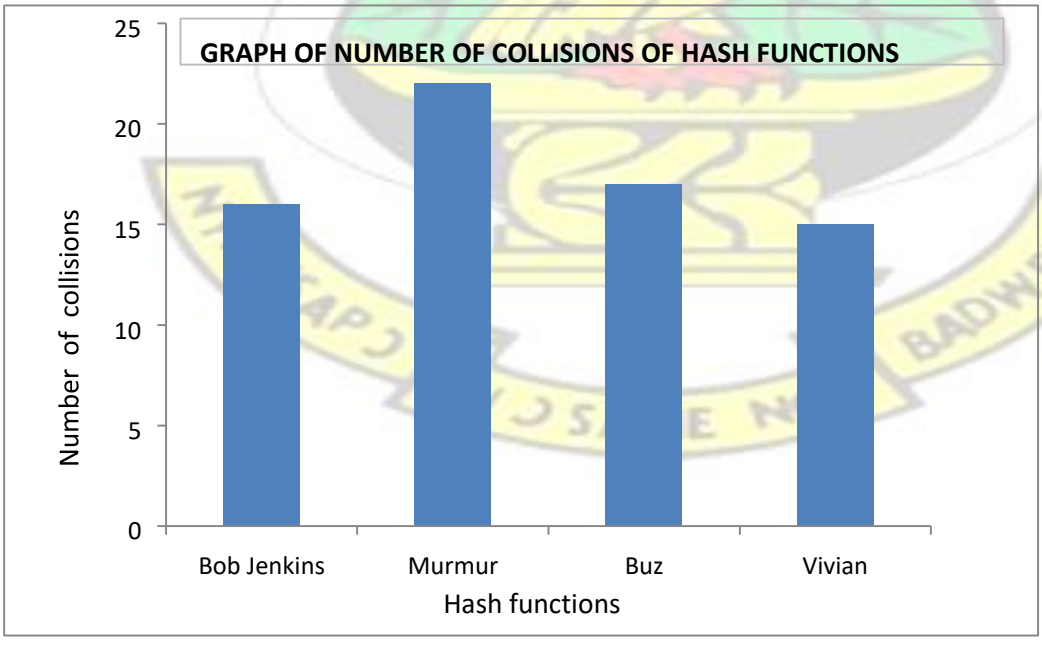


Figure 4.2 A graph of number of collisions of hash function

4.3 Performance or speed

Some hash algorithms or functions are cumbersome ie. Computationally expensive, the amount of time (and, in some cases, memory) taken to compute the hash may be burdensome. Speed is measured objectively by using number of lines of code and CPU benchmark. Also, when there are a lot of collisions and operations are performed in that particular location, it can contribute to increase in time spent in performing the operations and this contribute to reduced work output or productivity. But when collisions are less

when a particular hash function is used to store data, it helps to reduce return time and this contributes to increase in productivity or work output.

On the average, Vivian hash function had a better performance of 1249ms, followed by Buz hash function with 1289ms, 45132ms for Murmur hash function and 46041ms for Bob Jenkins hash function as shown in figure 4.3.

Therefore when speed is a priority in running applications that uses hash functions, Vivian hash function is the best, followed by Buz, Murmur and then Bob Jenkins hash function.

This is because Bob Jenkins hash function consists of an offset, initial value and a lot of loops for mixing (individual characters of the key). This consists of a lengthy code than any of the hash functions resulting in spending a lot of time to hash a key.

Murmur hash function consist of an offset, initial value and loops for mixing (individual characters of the key) but not that lengthy as Bob Jenkins hash function.

Buz hash function consist of an offset and requires more Pascal initialization code that comes from a randomized table and mixing (individual characters of the key)using two's compliment and bitwise operatorswhich is quite simple.

Vivian hash function consist of an offset, an initial value and mixing (individual characters of the key) using two's compliment which is more simpler and will require less time to hash a key.

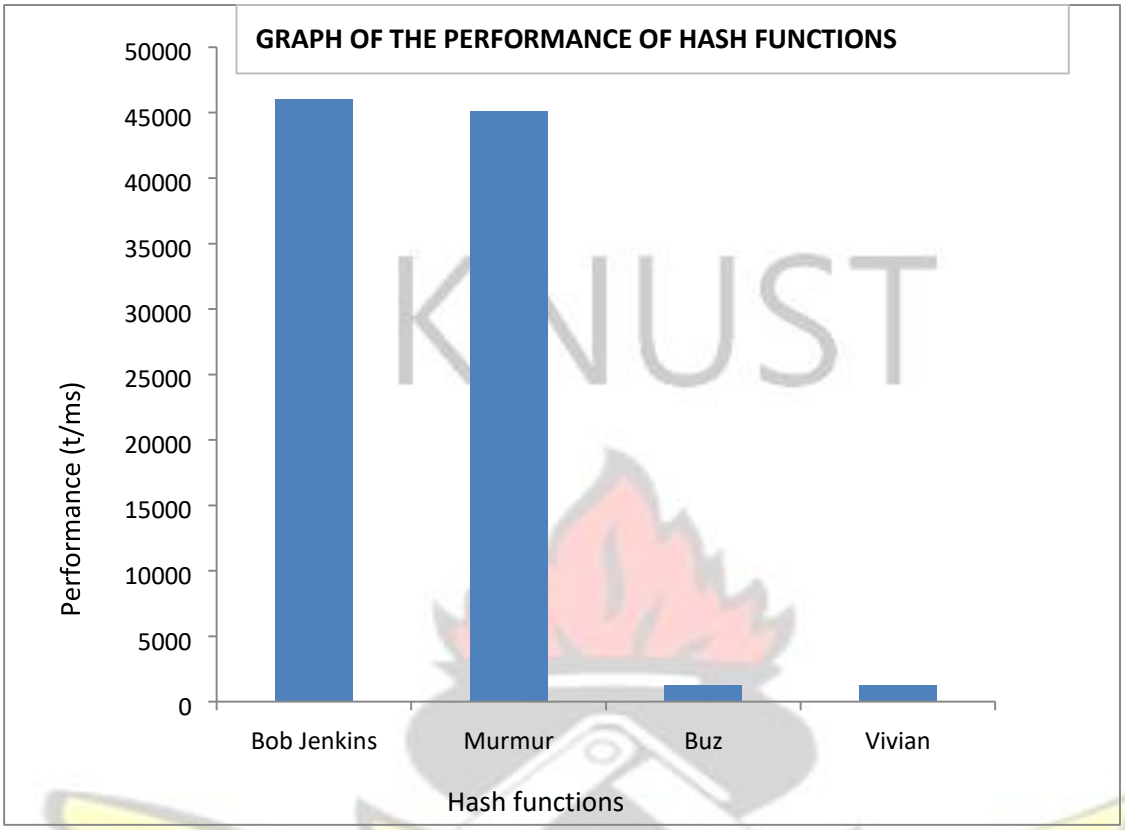


Figure 4.3 A graph of the performance of hash functions

4.4 Percentage Avalanche

A hash function achieve avalanche if the resulting hash index or value is widely different if a single key bit differs. Percentage avalanche aids distribution of data because keys that are similar will not end up having similar or same hash values. Hash function that have good

percentage avalanche distributes hash values in a uniform manner and this will help minimize the number collisions and fill the hash table more evenly.

Higher the percentage avalanche contribute to reduced return time and efficient use of memory space. This is because, it help spread data in memory and time needed to resolve collision and perform operations such as save, update, find and delete will reduce.

Percentage avalanche, both Buz and Vivian hash function had 100% throughout this means that, every bit of the key changed the hash value, followed by Bob Jenkins hash function with percentages less than that of the two above with an average of 85.6% and Murmur hash function with 27.9% which is the lowest percentage avalanche for all length of characters as shown in figure 4.4.This means that both Buz and Vivian hash function can help reduce return time more than Murmur and Bob Jenkins hash.

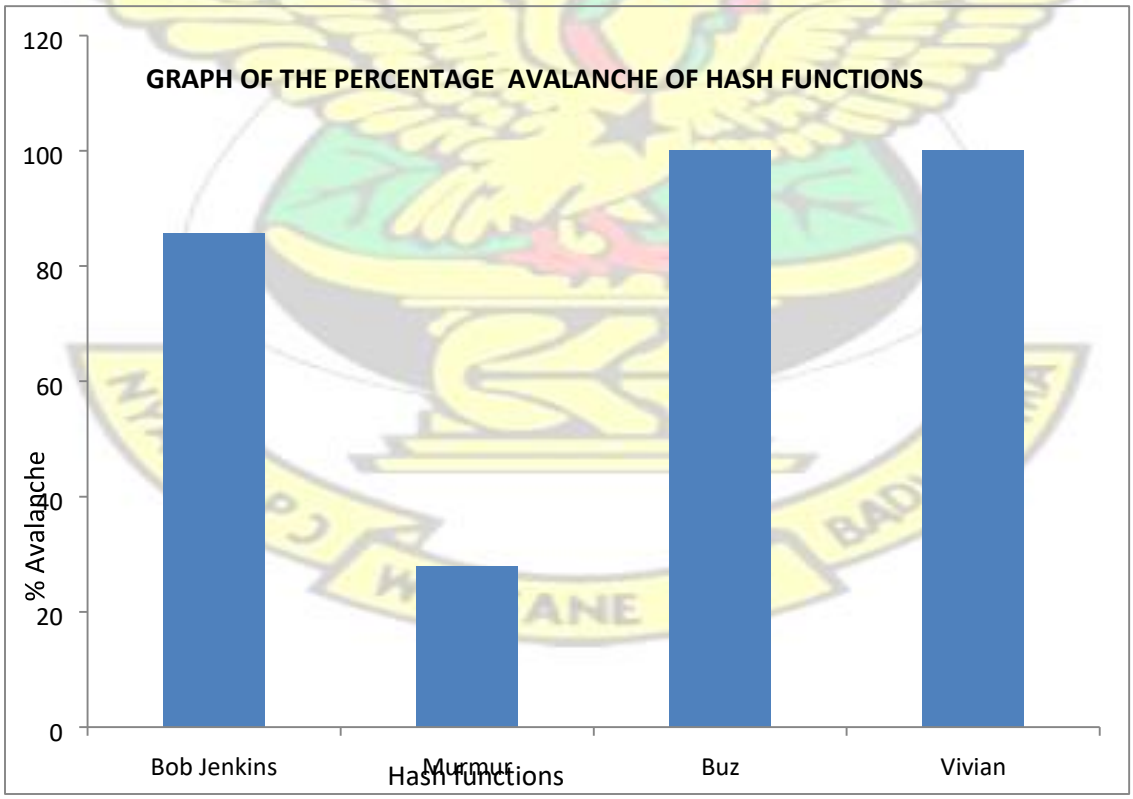


Figure 4.4A graph of the percentage avalanche of hash functions

4.5 Quality

Actually, a good hash function has quality between 0.95 and 1.05. If the quality is high, it means the function has a degraded performance and is not efficient. If the quality is less, it means the function has a good performance and is more efficient.

The lesser the number of collisions, the more efficient the hash function. This is because, it will contribute to even distribution of data in memory space allocated and this means there is efficient use of memory and also contribute to reduced return time because, the time that will be spent to resolve collision using collision resolution strategy and perform operations such as save, update, find and delete will reduce. But when there is more collisions,

memory space allocated will not be efficiently used and return time will increase because time to resolve collisions and perform operations as stated above will increase

On the average, Vivian hash function was more quality with a value of 0.99, followed by Bob Jenkins hash function with 1.02, Buz hash function with 1.04 and 1.23 for Murmur hash function as shown in figure 4.5.

Based on the values recorded, Vivian hash function can help reduce return time and make efficient use of memory space hence help increase work output.

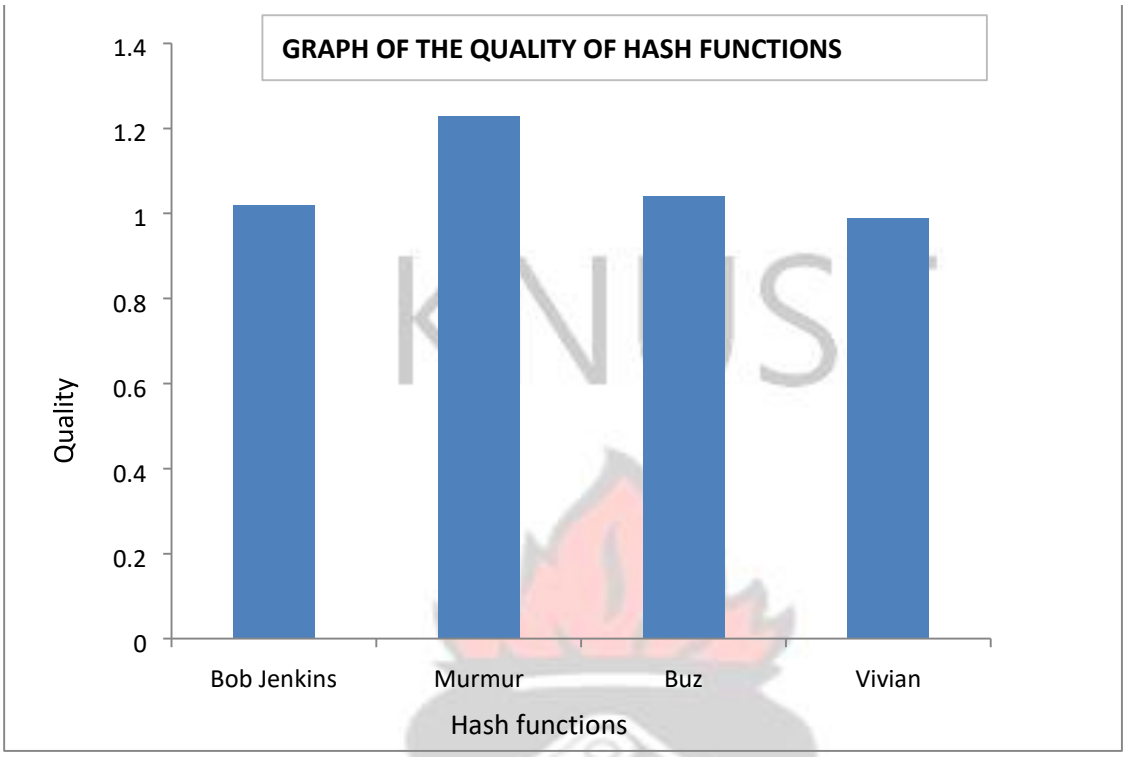


Figure 4.5 A graph of the quality of hash functions **CHAPTER FIVE**

CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

The most essential features of non-cryptographichash functions is its percentage distribution (which shows how evenly data is spread out in the memory space allocated for the data to be stored or the hash table), number of collisions (which shows the number of keys or data that hashes to the same address space), performance or speed (which shows how fast the hash function can consume input), % avalanche (Deals with how each individual key bit, contribute to a change in hash value produced) and quality (which tests the quality of hash function based on the various properties), which are the properties of the hash function.

For percentage distributions that help to make efficient use of memory space when running applications when data is well distributed in memory space allocated, Vivian hash had the best distribution followed by Bob Jenkins, Buz and Murmur hash function. This means that

Vivian hash makes efficient use of memory space when running applications followed by Bob Jenkins, Buz and Murmur hash function

For number of collisions which when less help to make efficient use of memory space and reduce return time when used to run applications and contribute to increased work output or productivity, Vivian hash has the lowest followed by Bob Jenkins, Buz and then Murmur hash function. This means that Vivian hash make efficient use of memory space and reduce return time when used to run applications and contribute to increased work output or productivity followed by Bob Jenkins, Buz and then Murmur hash function.

For performance or speed which when less when a particular hash function is used to store data help to reduce return time and contribute to increase in productivity or work output. Vivian hash function had a better performance by Buz hash function, Murmur and Bob Jenkins hash function.

Therefore when speed is a priority in running applications that uses hash functions, Vivian hash function is the best, followed by Buz, Murmur and then Bob Jenkins hash function.

For percentage avalanche which when high avalanche contribute to reduced return time and efficient use of memory space. Buz and Vivian hash function had the highest. This means that both Buz and Vivian hash function can help reduce return time more than Murmur and Bob Jenkins hash.

For quality which is dependent on all the other properties. The lower the value, the better the hash function. It depicts how effective and efficient a hash function is in terms of reduced return time and efficient use of memory space, which help to increase work output when used to run applications, Vivian hash function had the lowest value, followed by Bob Jenkins hash function, Buz hash function and 1.23 for Murmur hash function. This means that Vivian hash function is the most effective and efficient followed by Bob Jenkins hash function, Buz hash function and 1.23 for Murmur hash function

Based on the properties examined using tables and graphs, the results clearly demonstrated that; Vivian hash function hadbetter properties which mean that is more effective and efficient to use to run applications as compared to Bob Jenkins, Murmur and Buz hash functions.

5.2 Recommendations

Basing on the conclusions and findings of the research, it is recommended that Vivian hash function should be used when working with hash tables because of its better percentage distribution and avalanche, reduced number of collisions, its high speed and quality. This can help reduce return time when used to run applications and also make efficient use of memory space allocated.

5.3 Limitations of the study and suggestions for future reference

The key limitation of this research was the ability to measure the properties of the various java-based hash functions.

I suggest that Vivian hash function should be taken up and improve upon so that it will have better properties than it has now.

REFERENCES

"Couceiro et al." (PDF) (in (Portuguese)). 13 January 2012.

Adam,(2010). "MurmurHash2-160". Simonhf.wordpress.com. Retrieved 13 January 2012.

Ayuso,N., Pablo (2006). "Netfilter's connection tracking system" *(PDF).* ;login: *31 (3).*

Bar-Yosef, N., Wool, A., (2007). Remote algorithmic complexity attacks against randomized hash tables Proc. International Conference on Security and Cryptography (SECRYPT) *(PDF). pp. 117–124.*

Bob, J., (1997). "Hash functions". *Dr. Dobbs Journal*.

*Bob, J.,(*2012) "SpookyHash: a 128-bit noncryptographic hash".*Dr. Dobbs Journal*

Bruce, S.,(2016). "Cryptanalysis of MD5 and SHA: Time for a New Standard". Computerworld. Retrieved 2016-04-20. Much more than encryption algorithms, one-way hash functions are the workhorses of modern cryptography.

Chad, P., (2007). "Use MD5 hashes to verify software downloads". TechRepublic. Retrieved March 2, 2013.

Cormen, T. H., Leiserson, C.E., Rivest, R.L. and Stein, C., (2001). Introduction to Algorithms. MIT Press and McGraw-Hill. ISBN 0-262-03293-7, Second Edition, Chapter 11: Hash Tables, pp.221–252.

Dehne, F., Sack, J., Smid, M., (2006). "Data Structures and algorithms in java"., Springer.

Dillinger, Peter, C.; Manolios, Panagiotis, (2004). Fast and accurate bitstate verification for SPIN. Proc. 11th International SPIN Workshop. pp. 57–75. CiteSeerX*:* 10.1.1.4.6765.

Feldman, M. B., (1885). “Data structures with ADA”, Reston Pub. Co., original from the University of Michigann Digitized Jan 23, 2007, pg 244.

Goodrich, M.T. and Tamassia, R., (2013). Data Structures and Algorithms in Java, 4th edition. John Wiley & Sons, Inc. ISBN 0-471-73884-0. Chapter 9: Maps and Dictionaries. Pp.369–418.

Irving, G., Donkers, J., Uiterwijk, J., (2008) "Solving kalah" *(PDF). ICGA Journal.*

Knuth, D., Wesley, A., (1998). The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition, ISBN 0-201-89685-0. Section 6.4: Hashing, pp.513–558.

Kruse, R. L., Hall, P., (1987). Data Structures & Program Design, 2nd ed.. There is also a third edition (1994) as well as a new text, Data Structures and Program Design in C++, authored by Kruse and Alexander J. Ryba (1999).

Lalanne, C., Muralidharan, S., and Lysaght, M., (2015). An OpenCL design of the Bob Jenkins lookup3 hash function using the Xilinx TM SDAccel Development Environment. CHEC White Paper, July 16.

Main M., Wesley,A.,(1999). “Data structures and other objects using java”, Original from the University of Michagan Digitized Nov 17, 2007.

Pieprzyk, J., Hardjono T., Seberry, J., (2000). ‘’fundamentals of computer security’’ spinger, pg 242-285.

Sedgewick, R., Wesley, A.,(1998). ‘Algorithms in C’. 3rd edition, chapter14.

Singh, M., Garg, D.,(2009) ‘’choosing best hashing strategies, IEEE International advanced computing conference (IACC’09).

Tanjent, (2008). "MurmurHash first announcement". Tanjent.livejournal.com. Retrieved 13 January 2012.

Uzgalis,R.,(1995). Contact: buz@cs.aukuni.ac.nz

Uzgalis,R.,(2009).A very efficient java hash algorithm, based on the BuzHash algoritm by R obertUzgalis (http://www.serve.net/buz/hash.adt/java.000.html

Vitter, J. S., and Chen, W.C.,(1987). Design and Analysis of Coalesced Hashing, Oxford University Press, New York, NY, , ISBN 0-19-504182-8

Walker,H. M.,(1998). ‘Abstract Data Types.’ Clarendon Press, 4th Edition, pg 129143.

Wiener, R., Pinson, L.J,. (1987).‘’fundamentals of OOP and data structures in java’’, Cambridge university press, 367-393.

Zobel, J., Heinz, S. and Williams, H. E.,(2001). ‘’in-memory hash tables for accumulating text vocabularies’’, information processing letters, pg 271-277.

APPENDIX

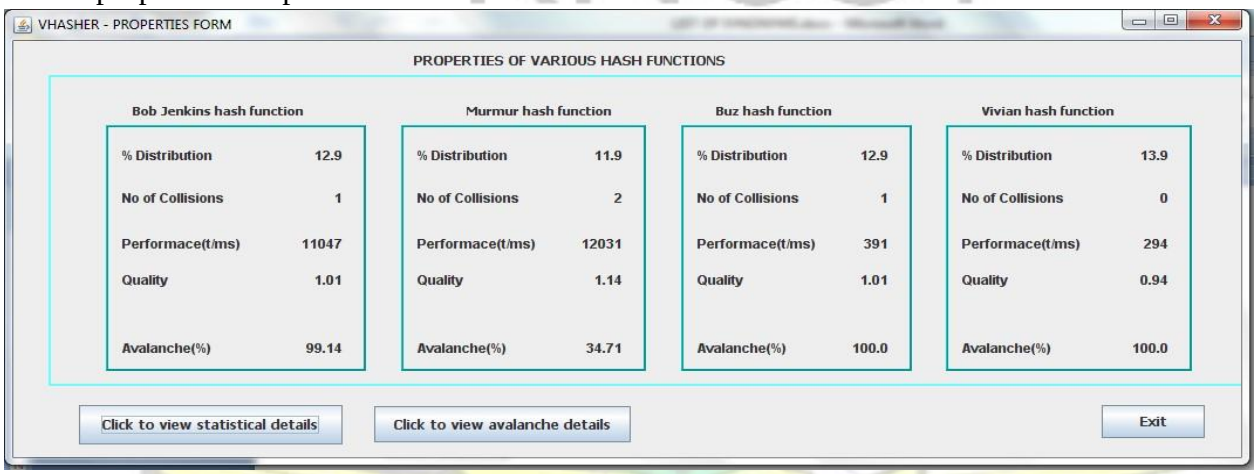
Properties of various hash function at different stages of entry

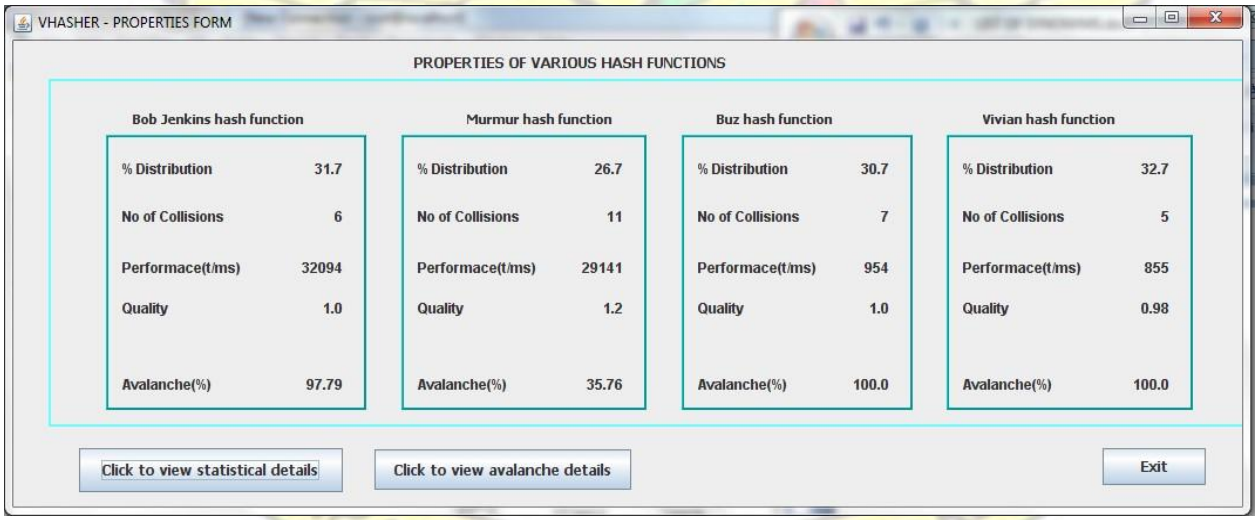
These are the properties of the various hash functions at different stages of entry.

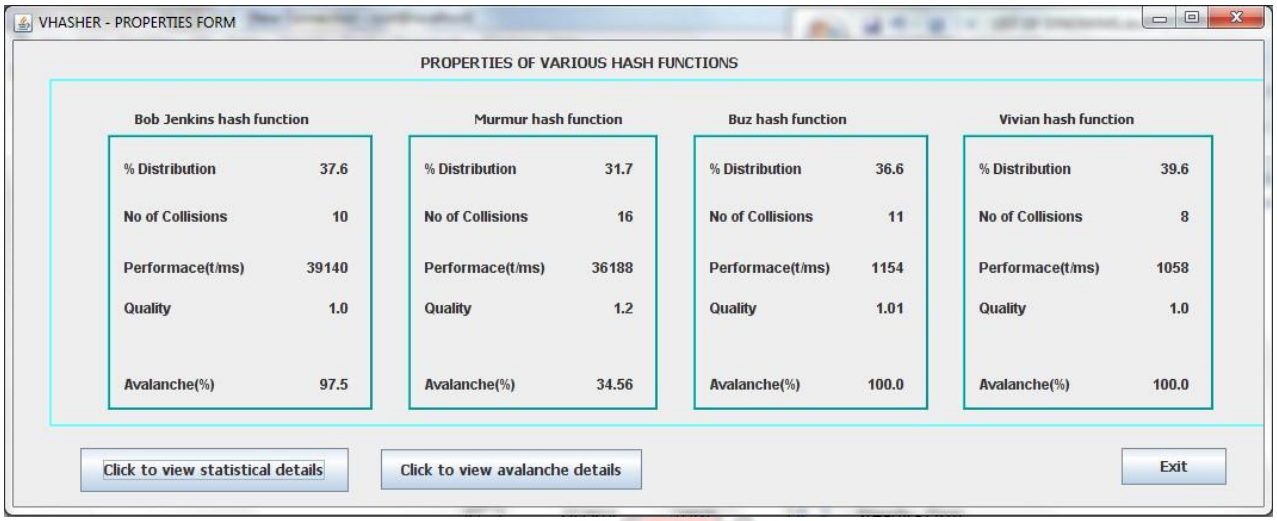
Sample properties forms;

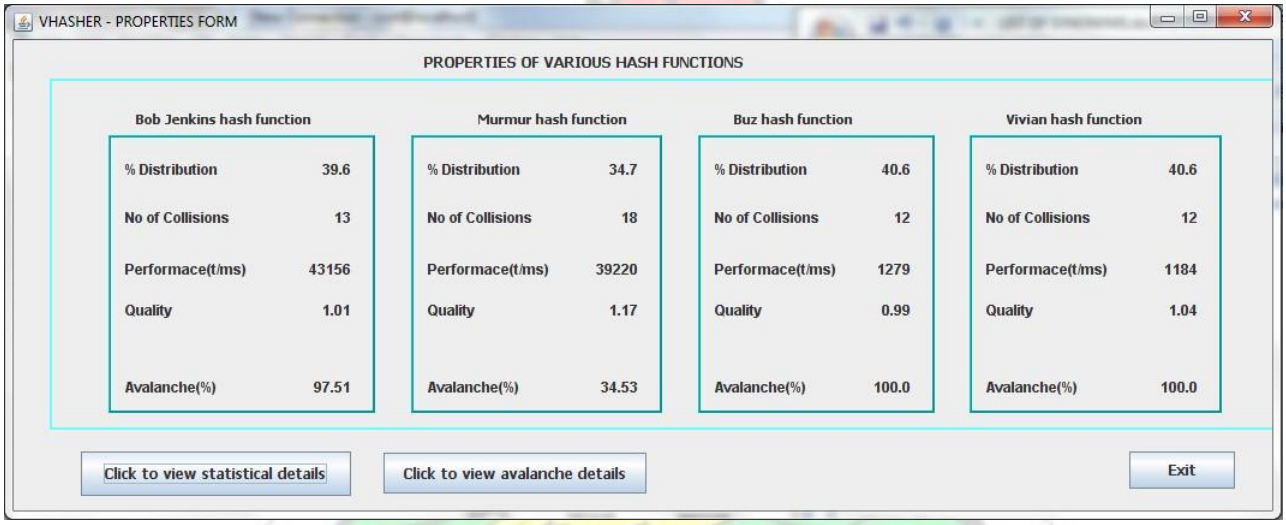
Characters not more than 7

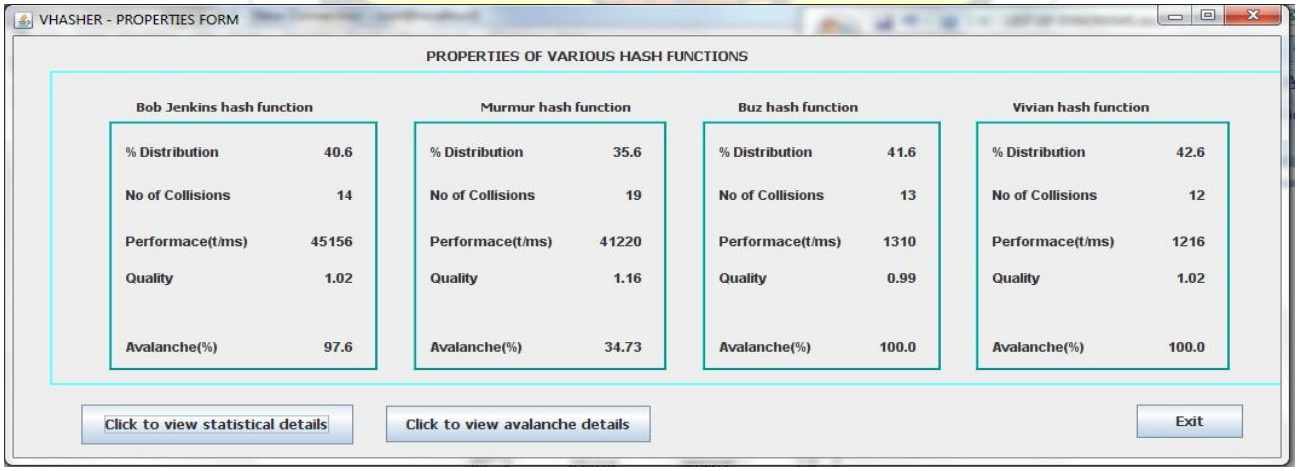
a. properties of up to 55 entries for characters less than 7

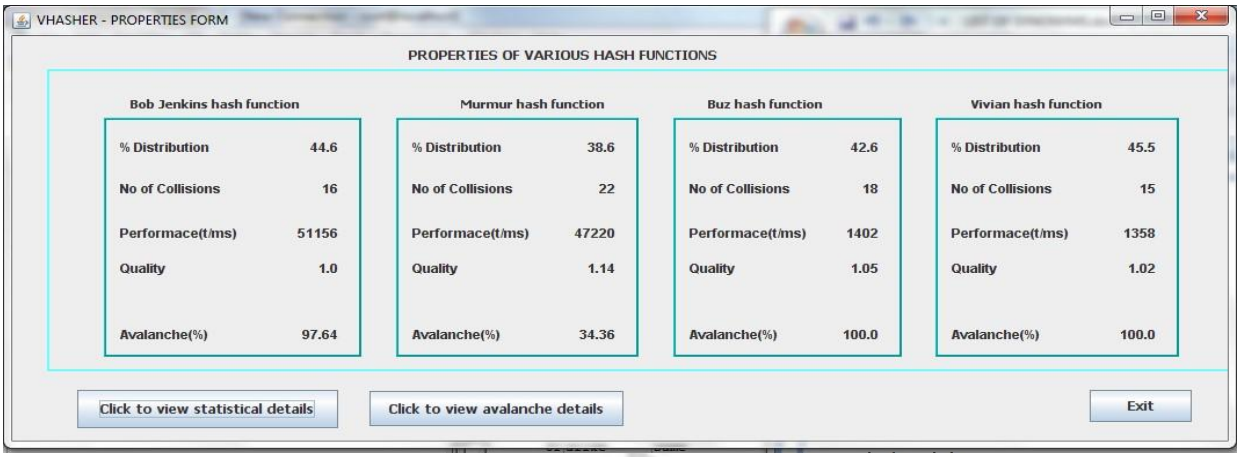




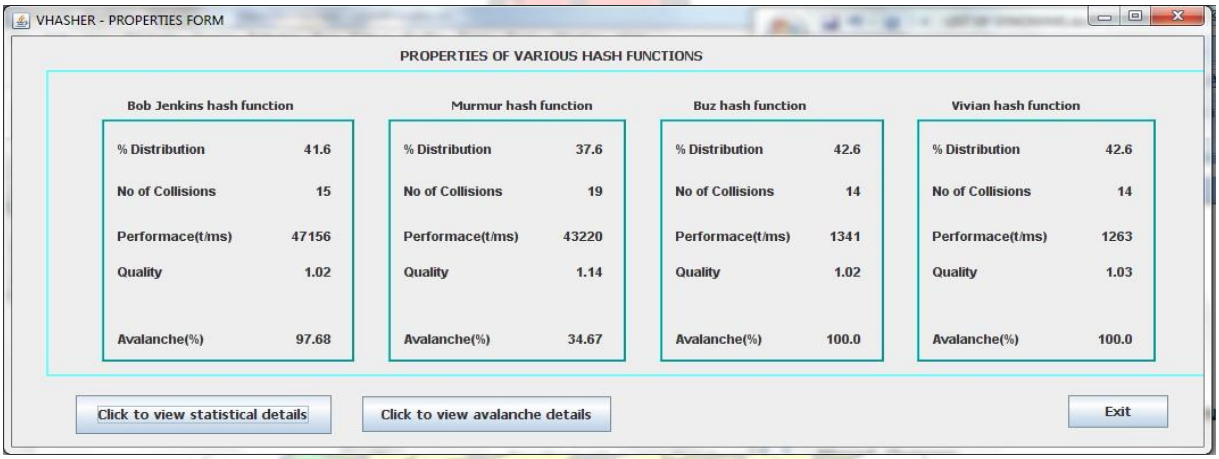

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 37.6
No of Collisions 10
Performace(t/ms) 39140
Quality 1.0
Avalanche(%) 97.5
Murmur hash function
% Distribution 31.7
No of Collisions 16
Performace(t/ms) 36188
Quality 1.2
Avalanche(%) 34.56
Buz hash function
% Distribution 36.6
No of Collisions 11
Performace(t/ms) 1154
Quality 1.01
Avalanche(%) 100.0
Vivian hash function
% Distribution 39.6
No of Collisions 8
Performace(t/ms) 1058
Quality 1.0
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

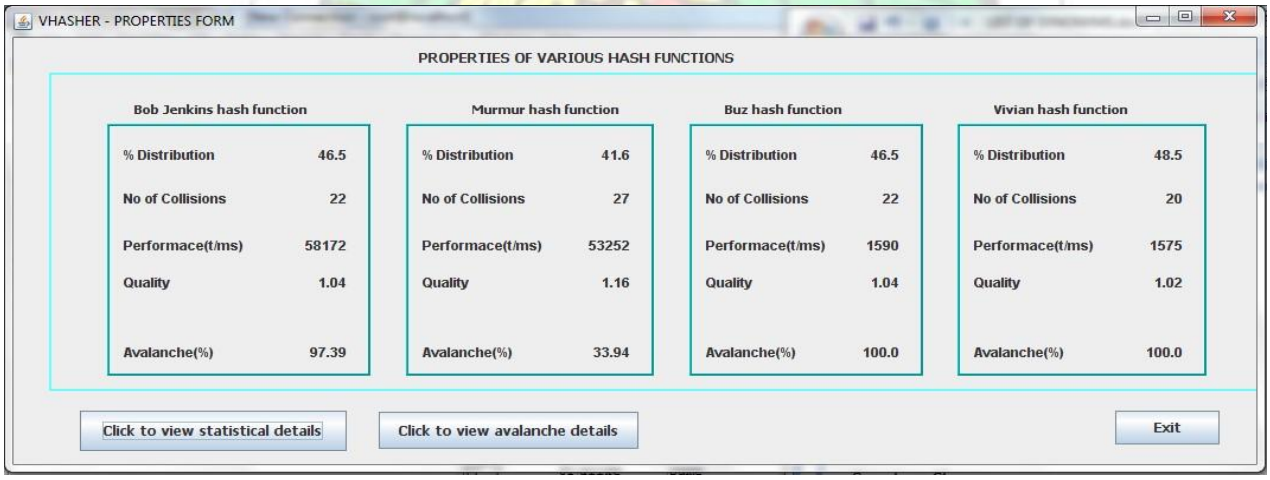

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 39.6
No of Collisions 13
Performace(t/ms) 43156
Quality 1.01
Avalanche(%) 97.51
Murmur hash function
% Distribution 34.7
No of Collisions 18
Performace(t/ms) 39220
Quality 1.17
Avalanche(%) 34.53
Buz hash function
% Distribution 40.6
No of Collisions 12
Performace(t/ms) 1279
Quality 0.99
Avalanche(%) 100.0
Vivian hash function
% Distribution 40.6
No of Collisions 12
Performace(t/ms) 1184
Quality 1.04
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

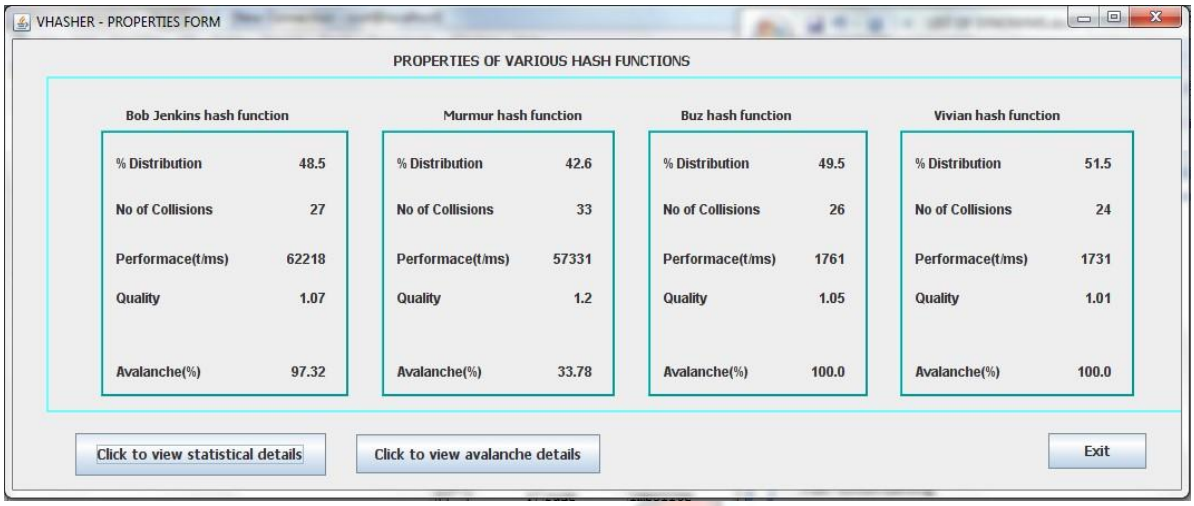

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 40.6
No of Collisions 14
Performace(t/ms) 45156
Quality 1.02
Avalanche(%) 97.6
Murmur hash function
% Distribution 35.6
No of Collisions 19
Performace(t/ms) 41220
Quality 1.16
Avalanche(%) 34.73
Buz hash function
% Distribution 41.6
No of Collisions 13
Performace(t/ms) 1310
Quality 0.99
Avalanche(%) 100.0
Vivian hash function
% Distribution 42.6
No of Collisions 12
Performace(t/ms) 1216
Quality 1.02
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

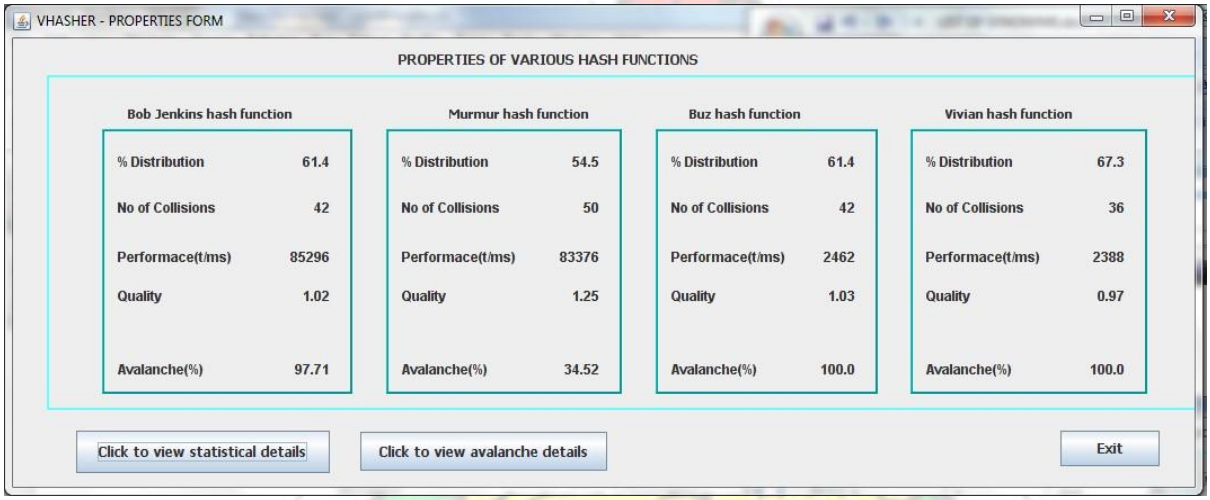


b. properties of above 55 entries for words not more than 7



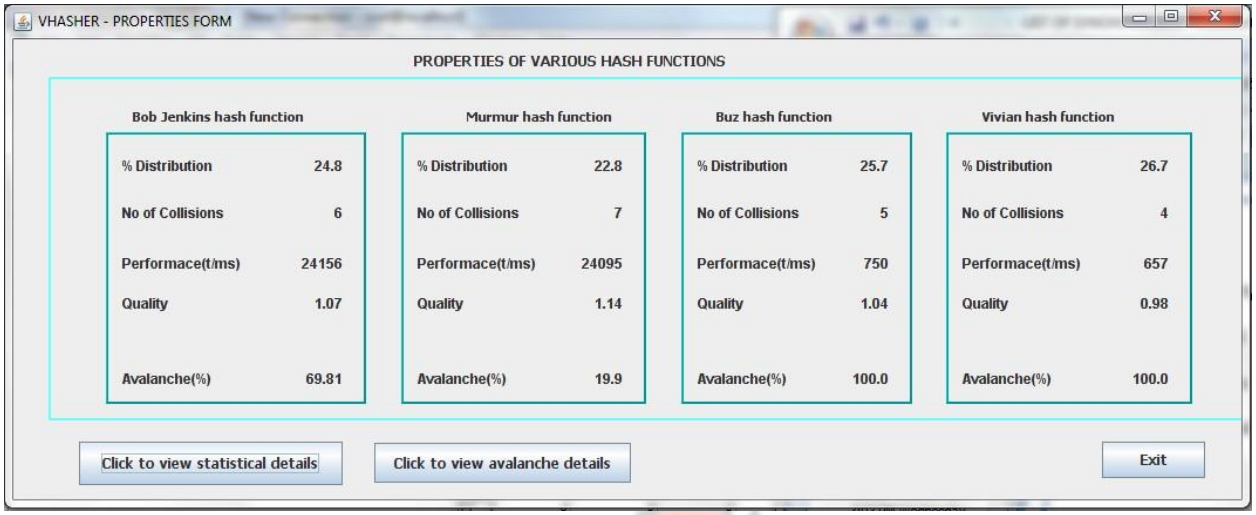


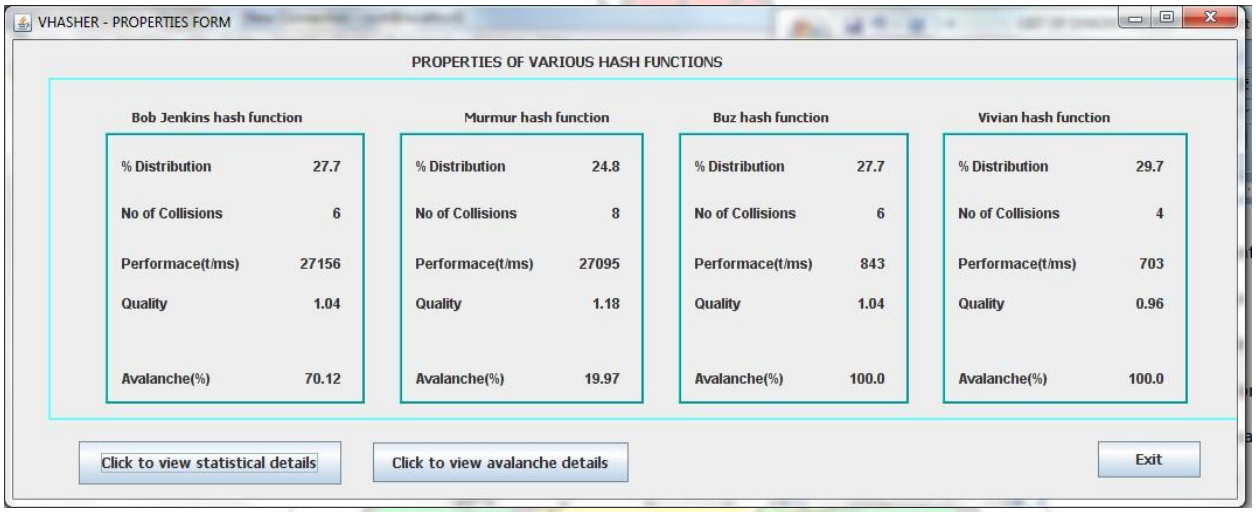


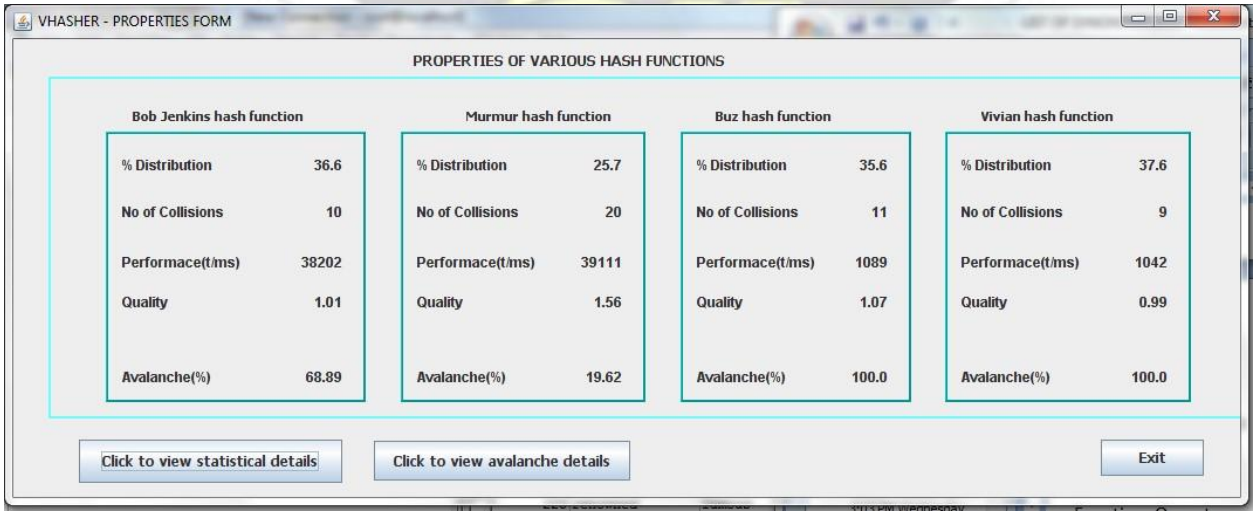


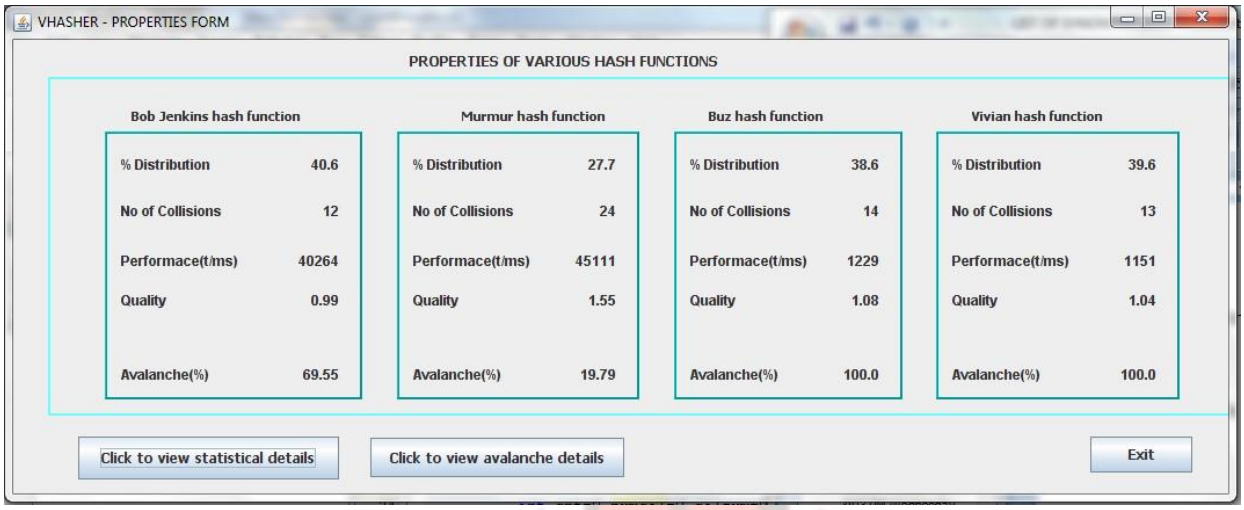
Characters more than 7

a. properties of less than 55 entries for words more than 7

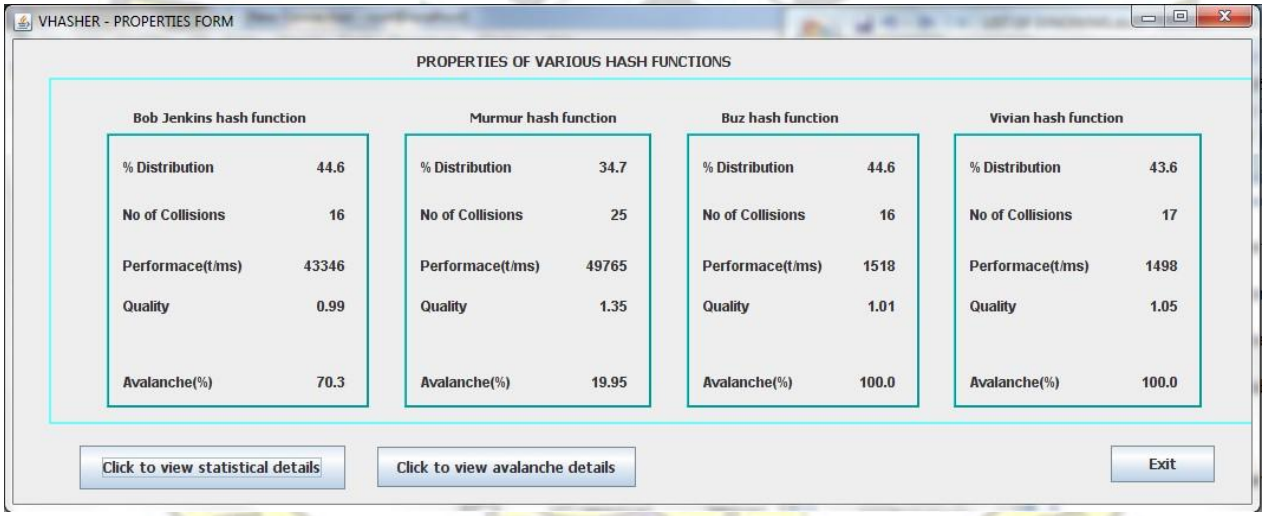

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 24.8
No of Collisions 6
Performace(t/ms) 24156
Quality 1.07
Avalanche(%) 69.81
Murmur hash function
% Distribution 22.8
No of Collisions 7
Performace(t/ms) 24095
Quality 1.14
Avalanche(%) 19.9
Buz hash function
% Distribution 25.7
No of Collisions 5
Performace(t/ms) 750
Quality 1.04
Avalanche(%) 100.0
Vivian hash function
% Distribution 26.7
No of Collisions 4
Performace(t/ms) 657
Quality 0.98
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

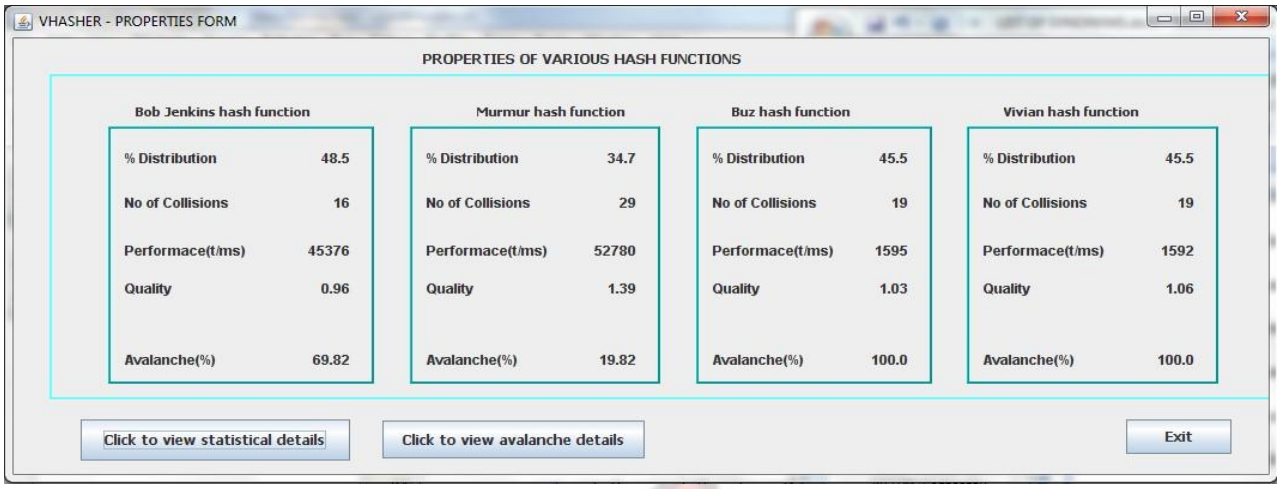

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 27.7
No of Collisions 6
Performace(t/ms) 27156
Quality 1.04
Avalanche(%) 70.12
Murmur hash function
% Distribution 24.8
No of Collisions 8
Performace(t/ms) 27095
Quality 1.18
Avalanche(%) 19.97
Buz hash function
% Distribution 27.7
No of Collisions 6
Performace(t/ms) 843
Quality 1.04
Avalanche(%) 100.0
Vivian hash function
% Distribution 29.7
No of Collisions 4
Performace(t/ms) 703
Quality 0.96
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

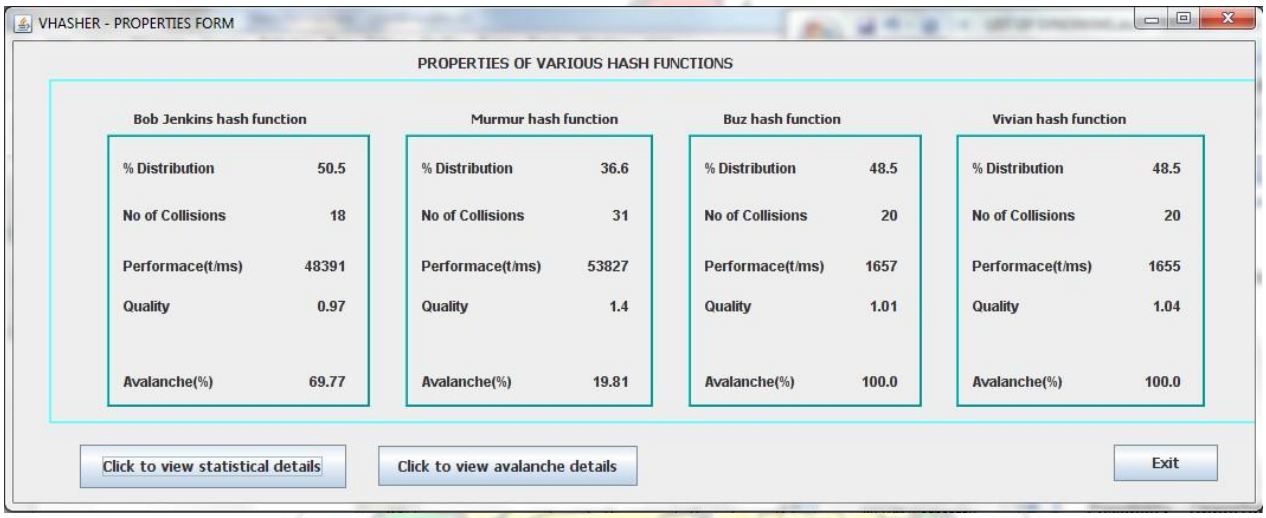

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 36.6
No of Collisions 10
Performace(t/ms) 38202
Quality 1.01
Avalanche(%) 68.89
Murmur hash function
% Distribution 25.7
No of Collisions 20
Performace(t/ms) 39111
Quality 1.56
Avalanche(%) 19.62
Buz hash function
% Distribution 35.6
No of Collisions 11
Performace(t/ms) 1089
Quality 1.07
Avalanche(%) 100.0
Vivian hash function
% Distribution 37.6
No of Collisions 9
Performace(t/ms) 1042
Quality 0.99
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

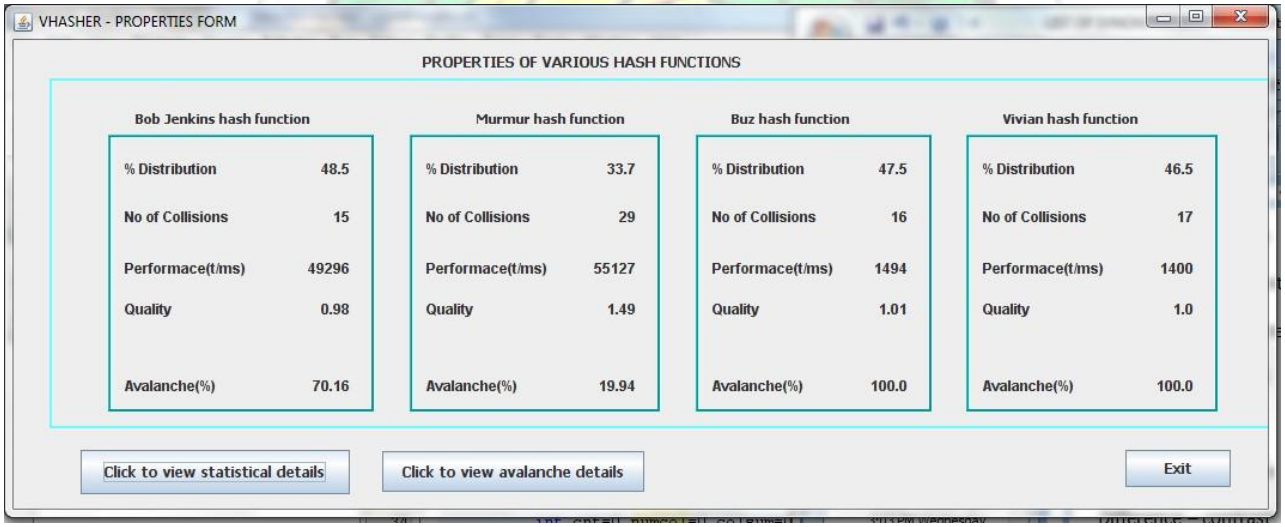


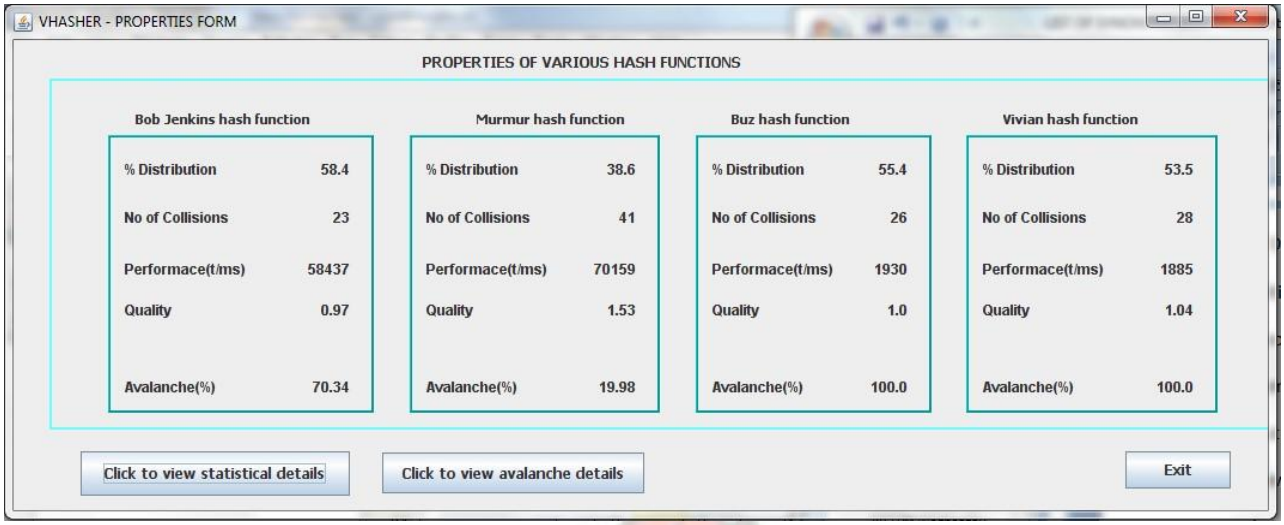
b. properties of more than 55 entries for words more than 7




VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 48.5
No of Collisions 16
Performace(t/ms) 45376
Quality 0.96
Avalanche(%) 69.82
Murmur hash function
% Distribution 34.7
No of Collisions 29
Performace(t/ms) 52780
Quality 1.39
Avalanche(%) 19.82
Buz hash function
% Distribution 45.5
No of Collisions 19
Performace(t/ms) 1595
Quality 1.03
Avalanche(%) 100.0
Vivian hash function
% Distribution 45.5
No of Collisions 19
Performace(t/ms) 1592
Quality 1.06
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

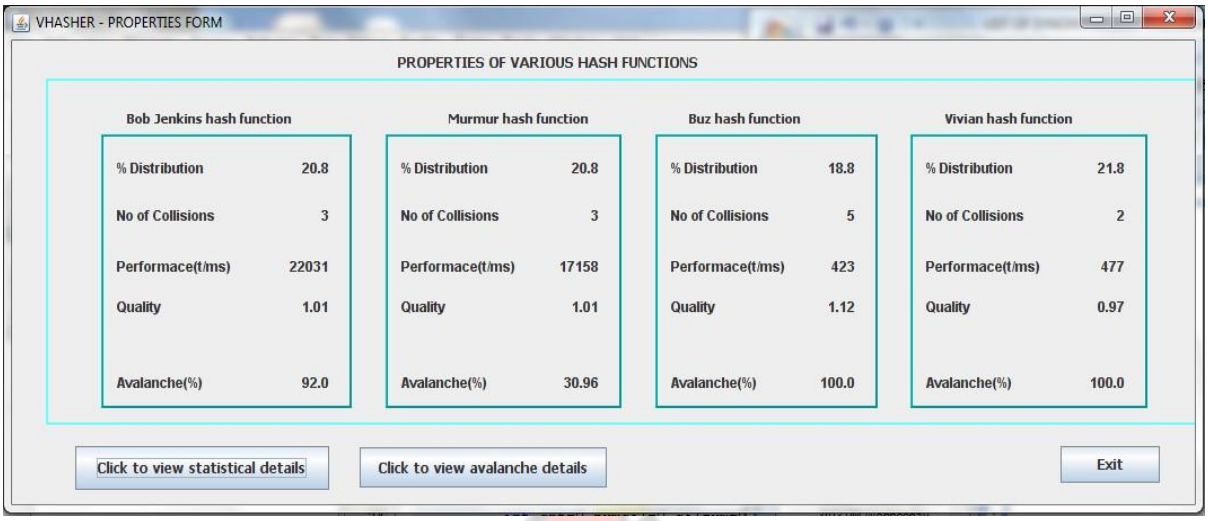

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 50.5
No of Collisions 18
Performace(t/ms) 48391
Quality 0.97
Avalanche(%) 69.77
Murmur hash function
% Distribution 36.6
No of Collisions 31
Performace(t/ms) 53827
Quality 1.4
Avalanche(%) 19.81
Buz hash function
% Distribution 48.5
No of Collisions 20
Performace(t/ms) 1657
Quality 1.01
Avalanche(%) 100.0
Vivian hash function
% Distribution 48.5
No of Collisions 20
Performace(t/ms) 1655
Quality 1.04
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

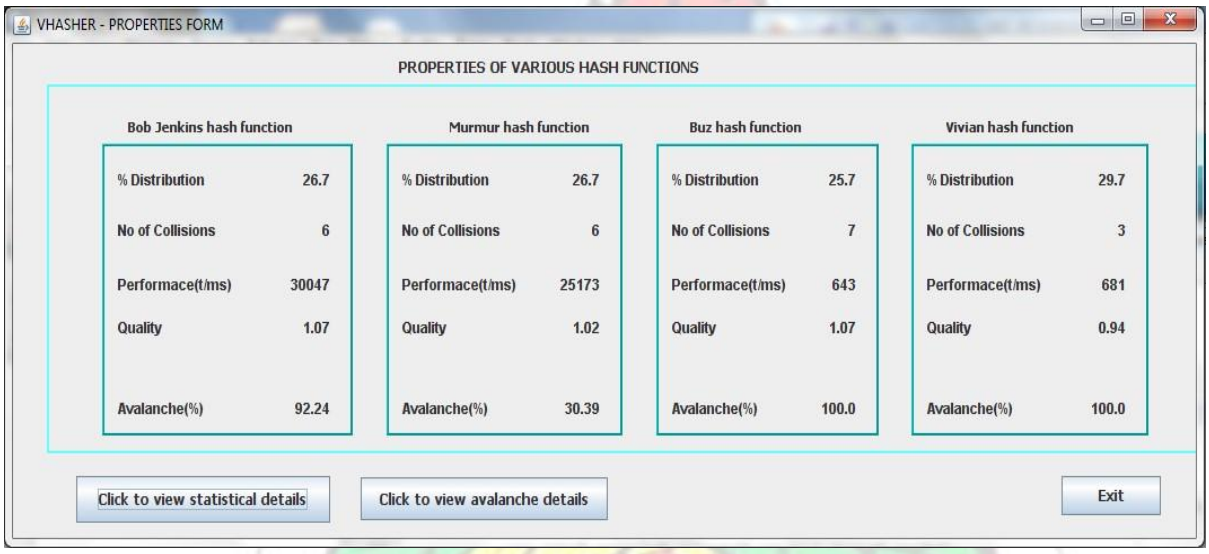

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 48.5
No of Collisions 15
Performace(t/ms) 49296
Quality 0.98
Avalanche(%) 70.16
Murmur hash function
% Distribution 33.7
No of Collisions 29
Performace(t/ms) 55127
Quality 1.49
Avalanche(%) 19.94
Buz hash function
% Distribution 47.5
No of Collisions 16
Performace(t/ms) 1494
Quality 1.01
Avalanche(%) 100.0
Vivian hash function
% Distribution 46.5
No of Collisions 17
Performace(t/ms) 1400
Quality 1.0
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

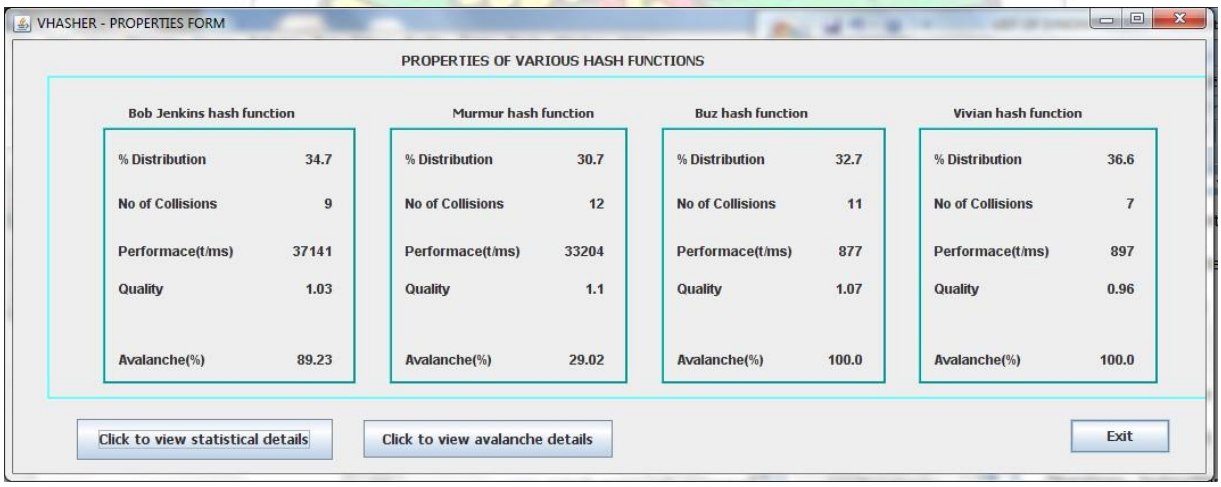


Mixed length of words

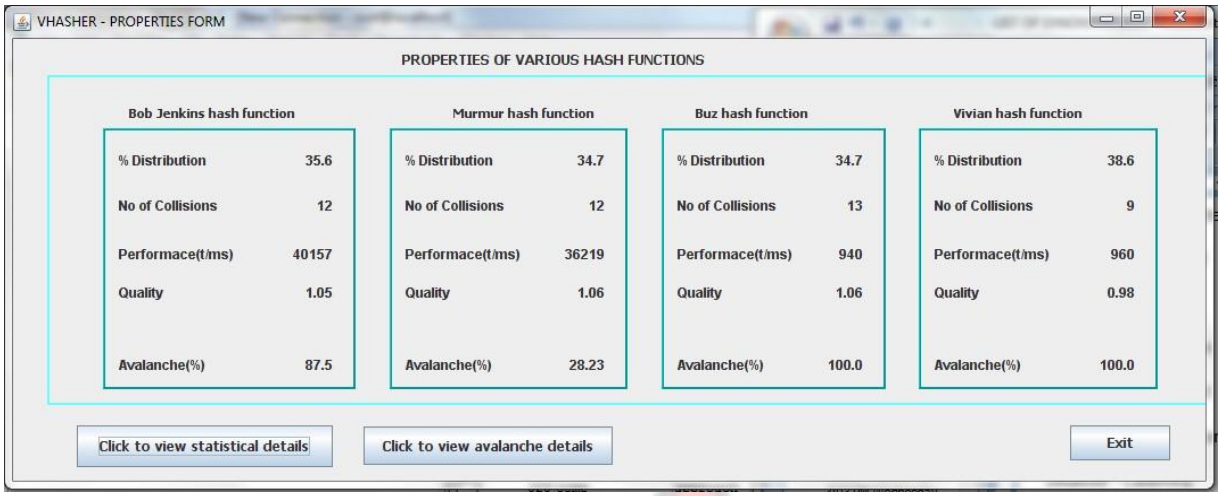
a. properties of less than 55 entries for mixed length of words

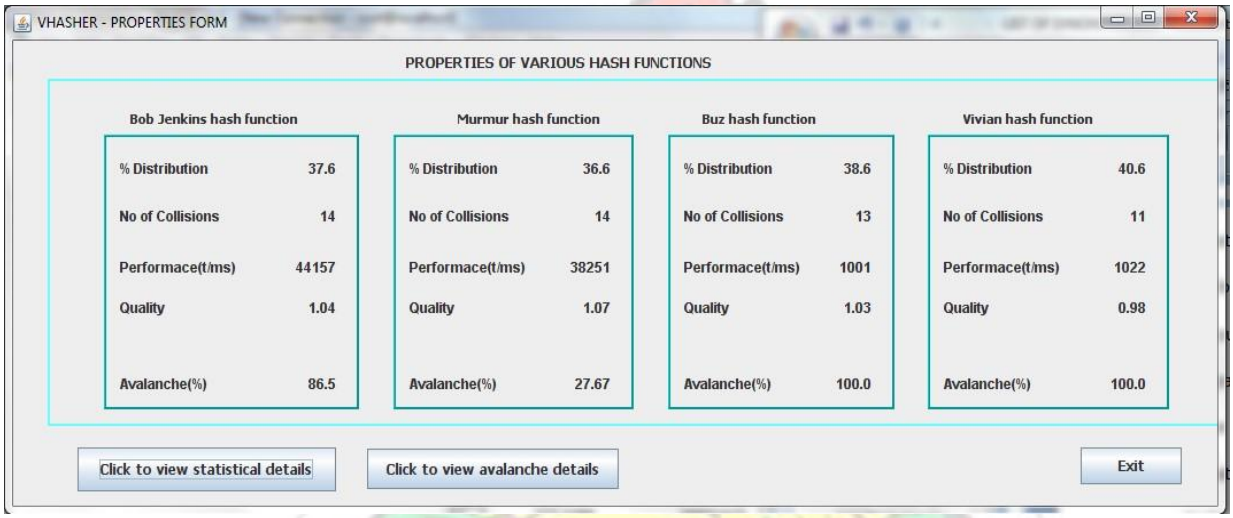

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 20.8
No of Collisions 3
Performace(t/ms) 22031
Quality 1.01
Avalanche(%) 92.0
Murmur hash function
% Distribution 20.8
No of Collisions 3
Performace(t/ms) 17158
Quality 1.01
Avalanche(%) 30.96
Buz hash function
% Distribution 18.8
No of Collisions 5
Performace(t/ms) 423
Quality 1.12
Avalanche(%) 100.0
Vivian hash function
% Distribution 21.8
No of Collisions 2
Performace(t/ms) 477
Quality 0.97
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit


VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 26.7
No of Collisions 6
Performace(t/ms) 30047
Quality 1.07
Avalanche(%) 92.24
Murmur hash function
% Distribution 26.7
No of Collisions 6
Performace(t/ms) 25173
Quality 1.02
Avalanche(%) 30.39
Buz hash function
% Distribution 25.7
No of Collisions 7
Performace(t/ms) 643
Quality 1.07
Avalanche(%) 100.0
Vivian hash function
% Distribution 29.7
No of Collisions 3
Performace(t/ms) 681
Quality 0.94
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

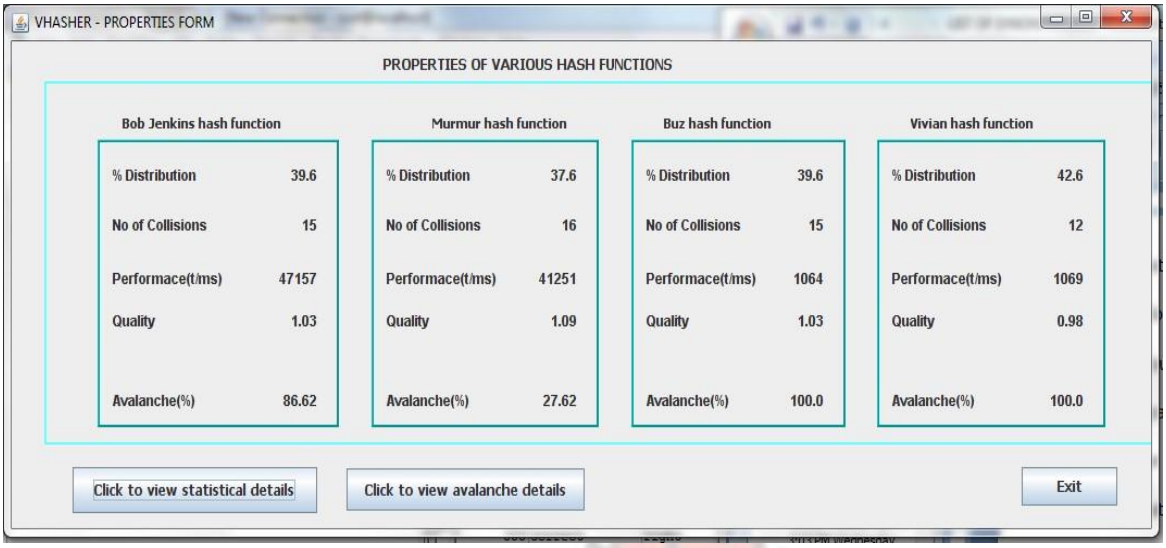

VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 34.7
No of Collisions 9
Performace(t/ms) 37141
Quality 1.03
Avalanche(%) 89.23
Murmur hash function
% Distribution 30.7
No of Collisions 12
Performace(t/ms) 33204
Quality 1.1
Avalanche(%) 29.02
Buz hash function
% Distribution 32.7
No of Collisions 11
Performace(t/ms) 877
Quality 1.07
Avalanche(%) 100.0
Vivian hash function
% Distribution 36.6
No of Collisions 7
Performace(t/ms) 897
Quality 0.96
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit

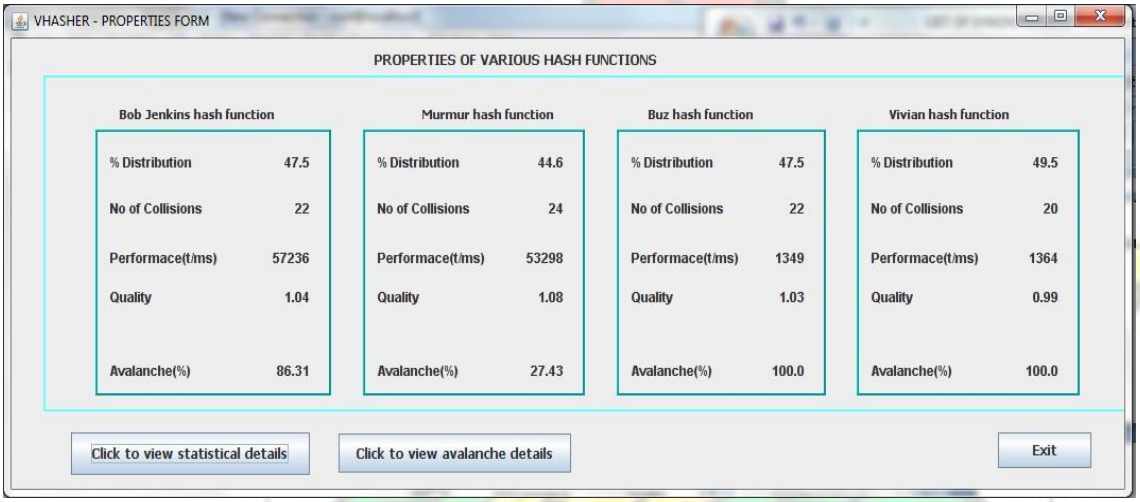
C

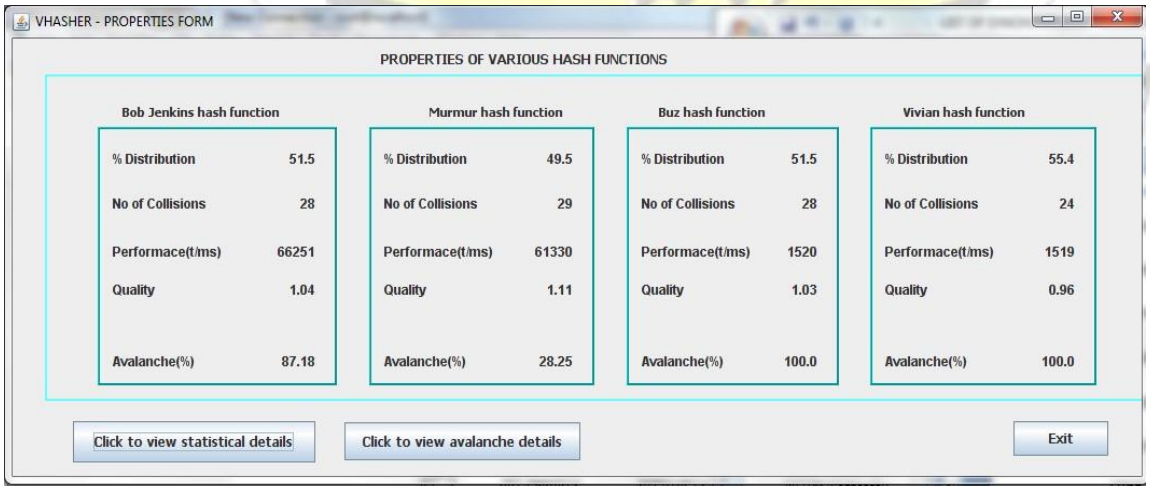


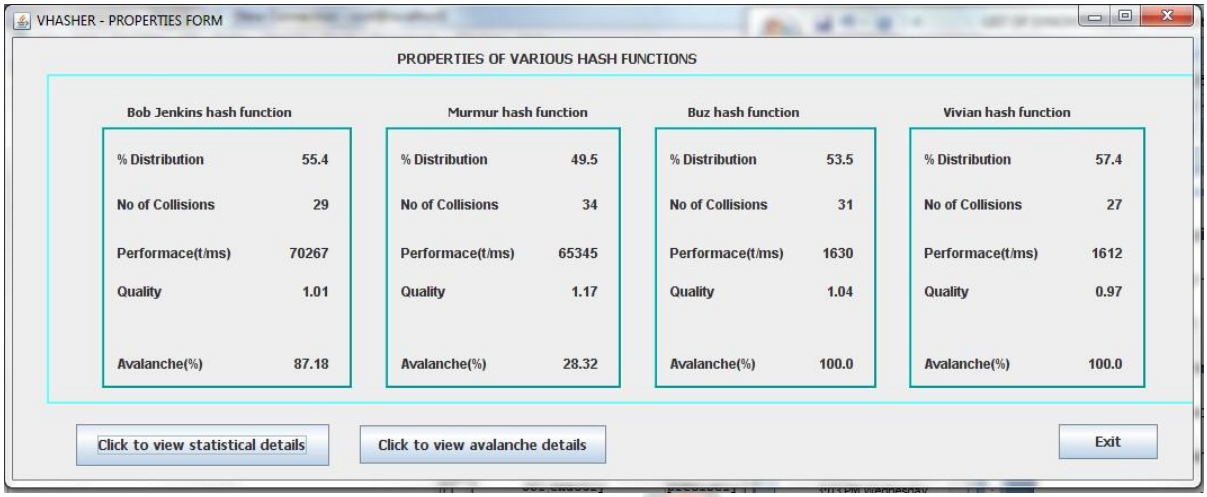


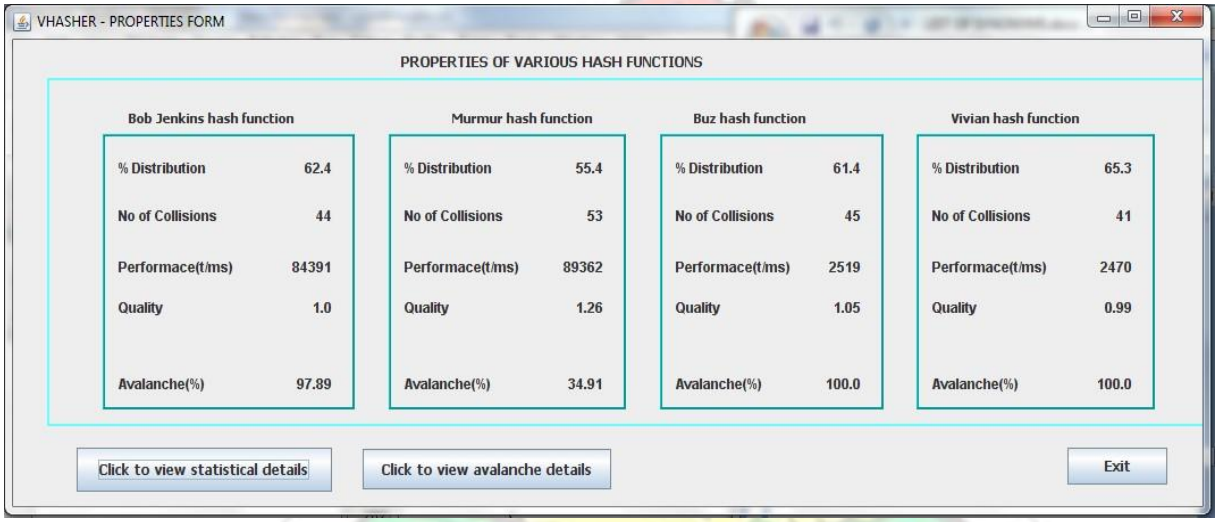
b. properties of more than 55 entries for mixed length of words


VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 39.6
No of Collisions 15
Performace(t/ms) 47157
Quality 1.03
Avalanche(%) 86.62
Murmur hash function
% Distribution 37.6
No of Collisions 16
Performace(t/ms) 41251
Quality 1.09
Avalanche(%) 27.62
Buz hash function
% Distribution 39.6
No of Collisions 15
Performace(t/ms) 1064
Quality 1.03
Avalanche(%) 100.0
Vivian hash function
% Distribution 42.6
No of Collisions 12
Performace(t/ms) 1069
Quality 0.98
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit


VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 47.5
No of Collisions 22
Performace(t/ms) 57236
Quality 1.04
Avalanche(%) 86.31
Murmur hash function
% Distribution 44.6
No of Collisions 24
Performace(t/ms) 53298
Quality 1.08
Avalanche(%) 27.43
Buz hash function
% Distribution 47.5
No of Collisions 22
Performace(t/ms) 1349
Quality 1.03
Avalanche(%) 100.0
Vivian hash function
% Distribution 49.5
No of Collisions 20
Performace(t/ms) 1364
Quality 0.99
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit


VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 51.5
No of Collisions 28
Performace(t/ms) 66251
Quality 1.04
Avalanche(%) 87.18
Murmur hash function
% Distribution 49.5
No of Collisions 29
Performace(t/ms) 61330
Quality 1.11
Avalanche(%) 28.25
Buz hash function
% Distribution 51.5
No of Collisions 28
Performace(t/ms) 1520
Quality 1.03
Avalanche(%) 100.0
Vivian hash function
% Distribution 55.4
No of Collisions 24
Performace(t/ms) 1519
Quality 0.96
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit


VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 55.4
No of Collisions 29
Performace(t/ms) 70267
Quality 1.01
Avalanche(%) 87.18
Murmur hash function
% Distribution 49.5
No of Collisions 34
Performace(t/ms) 65345
Quality 1.17
Avalanche(%) 28.32
Buz hash function
% Distribution 53.5
No of Collisions 31
Performace(t/ms) 1630
Quality 1.04
Avalanche(%) 100.0
Vivian hash function
% Distribution 57.4
No of Collisions 27
Performace(t/ms) 1612
Quality 0.97
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit


VHASHER - PROPERTIES FORM
PROPERTIES OF VARIOUS HASH FUNCTIONS
Bob Jenkins hash function
% Distribution 62.4
No of Collisions 44
Performace(t/ms) 84391
Quality 1.0
Avalanche(%) 97.89
Murmur hash function
% Distribution 55.4
No of Collisions 53
Performace(t/ms) 89362
Quality 1.26
Avalanche(%) 34.91
Buz hash function
% Distribution 61.4
No of Collisions 45
Performace(t/ms) 2519
Quality 1.05
Avalanche(%) 100.0
Vivian hash function
% Distribution 65.3
No of Collisions 41
Performace(t/ms) 2470
Quality 0.99
Avalanche(%) 100.0
Click to view statistical details
Click to view avalanche details
Exit