Linear Hashing

GACS-7102-001

The final project Report of Web document and databases

Submitted to

Dr. Yangjun chen

in partial fulfilment for the degree of Master of Science

By

Hardeep singh

3089450

Singh-h18@webmail.uwinnipeg.ca

April 15, 2017

Contents

Introduction

Several dynamic hashing techniques for different external files have been developed over the last few years. These type of techniques allow the data file size to grow and shrink, which depend on the number of records actually stored in the file. Any one of the techniques can be used for internal hash tables as well. However, the technique best fit for internal tables is linear hashing, it is easy to implement and use little extra storage. This paper shows how to do searching and insertion in linear hashing. Linear hashing is faster and also easier to implement. A basic characteristic of hashing technique is that a higher load on the table increases the cost of all basic operations such as insertion and searching. If the performance of a hash table is to remain within limits when the number of records increases, additional storage must somehow be allocated to the table. The traditional solution is to be built a new, larger hash table and rehash all the records into the new table. The details of how and when this is done can vary accordingly. Linear hashing allows a smooth growth. As the number of records increases, the table grows one bucket at a time. When a new bucket is added to the address space, a limited local reorganization is performed.

Linear hashing is a dynamic hash table algorithm invented by Witold Litwin (1980), and later popularized by Paul Larson. The original technique was constructed for external files. Several upgraded versions of linear hashing have been proposed. Consider a hash table consisting of N buckets with addresses 0, 1, . . . , N - 1. Linear hashing increases the address space gradually by splitting the buckets in a predetermined order like, first bucket 0, then bucket 1, and so on, up to and including bucket N - 1. Splitting a bucket involves moving the records from the bucket to a new bucket at the end of the table.

Linear Hashing is a dynamically updateable disk-based index structure which implements a hashing technique and which grows or shrinks one bucket at a time. If we Compared it with the B+ tree index, which also supports exact match queries, where Linear Hashing has better expected query cost O(1) I/O. Moreover, Compared with Extendible Hashing technique, Linear Hashing does not use a bucket directory, and

when an overflow occurs it is not always the overflown bucket that is split. The name Linear Hashing is used because the number of buckets grows or shrinks in a linear fashion. Overflows are handled by creating a chain of pages under the overflown bucket.
The hashing function changes dynamically and at any given instant there can be at most two hashing functions used by the scheme.

Historical background

A hash table is an in-memory data structure (Array) that associates keys with values. The main operations it support easily and efficiently is a search: given a key, and the corresponding value and insertion of data files. It works by transforming the key using a hash function into a hash, a number that is used as an index in an array to locate the desired location where the values should be. Multiple keys may be hashed to the same bucket, and all keys in a bucket should be searched upon a query. Hash tables are often used to implement associative arrays, sets and caches. Like arrays, hash tables have O(1) lookup cost on average [4]. Whereas, the other hashing techniques like static hashing, in which the insertion is based on inserting the data files at the given keys by the users and overflown is handled by the chaining process. However, drawback of linear hashing is the large overflowing buckets which slows the hashing functions. Apart from this, the other technique such as extendible hashing use bucket directory, where overflown is handled but taking lot of space. To overcome this, linear hashing was introduced to dynamic allocation of the data files in the hash table.

Basic Idea:

The basic idea is to split buckets when overflows occur but not necessarily the page with the overflow. Splitting occurs in turn, in a round robin fashion. One by one from the first bucket to the last bucket.

- Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
- Each function's range is twice that of its predecessor.
- When all the pages at one phase (the current hash function) have been split, a new phase is applied.

- Primary pages are allocated in order & consecutively.

Insertion Algorithm

1. Initially the hash file comprises M primary buckets numbered 0, 1, … M-1.
2. The hashing process is divided into several phases (phase 0, phase 1, phase 2, …). In phase j, records are hashed according to hash functions h_j(key) and h_{j+1}(key).
3. h_j(key) = key mod (2^j*M)
4. Phases in Linear hashing

Phase 0: h_0(key) = key mod (2^0*M), h_1(key) = key mod (2^1*M)
Phase 1: h_1(key) = key mod (2^1*M), h_2(key) = key mod (2^2*M)
Phase n: h_n(key) = key mod (2^n*M), h_{n+1}(key) = key mod (2^{n+1}*M)

5. Bucket Splitting

- If a bucket overflows its primary page is chained to an overflow page (same as in static hashing).
- Also when a bucket overflows, some bucket is split.
- The first bucket to be split is the first bucket in the file (*not* necessarily the bucket that overflows).
- The next bucket to be split is the second bucket in the file and so on until the *N*th has been split.
- When buckets are split their entries (including those in overflow pages) are distributed using hash function h_1.
- To access split buckets the next level hash function (h_1) is applied.
- h_1 maps entries to $2N_0$ (or N_1)buckets.
- splitting occurs according to a specific rule such as
    1. An overflow occurring, or
    2. The load factor reaching a certain value, etc.
    3. A split pointer (n) keeps track of which bucket to split next.
    4. Split pointer goes from 0 to 2^j*M - 1 during the j^{th} phase, j= 0, 1,2, … ...

Formula for load factor

Load factor = n/m
Where,

n is total number of entries in hash table.
m is total number of buckets in hash table.

If the load factor grows bigger, the hash table becomes slower, and it is possibility there that it may even fail to work. The expected constant time property of a hash table assumes that the load factor is kept below some bound. For a *fixed* number of buckets, the time for a search grows with the number of entries and therefore the desired constant time is not achieved.

Second to that, one can examine the number of entries per bucket. For example, two tables both have 500 entries and 500 buckets, one has exactly one entry in each bucket, the other has all entries in the same bucket. Clearly the hashing is not working in the second one.

A low load factor is not especially beneficial. As the load factor approaches 0, the proportion of unused areas in the hash table increases, but there is not necessarily any reduction in search cost. This results in wasted memory.[1]

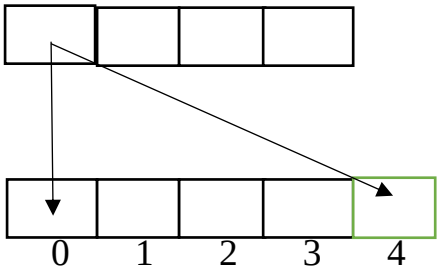
Insertion Example

Initially suppose M = 4

h_0(key) = key mod (M), i.e key mod 4 (rightmost 2 bits)

h_1(key) = key mod 2*M

Capacity of bucket is 2



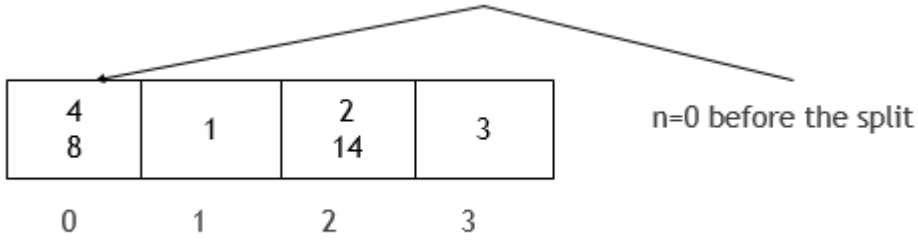
As the file grows bucket split and records are using

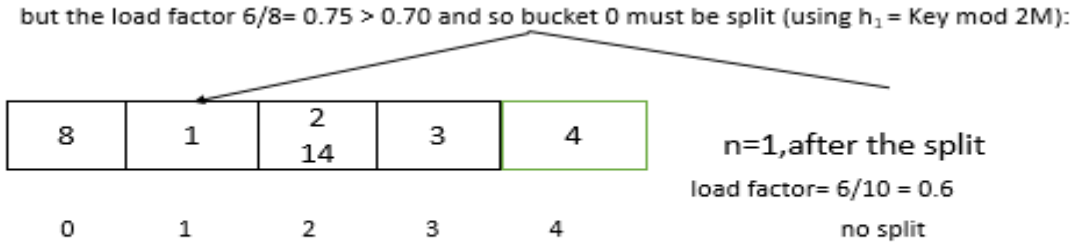
h_1(key) = key mod 2*M

Insert the records with key values:

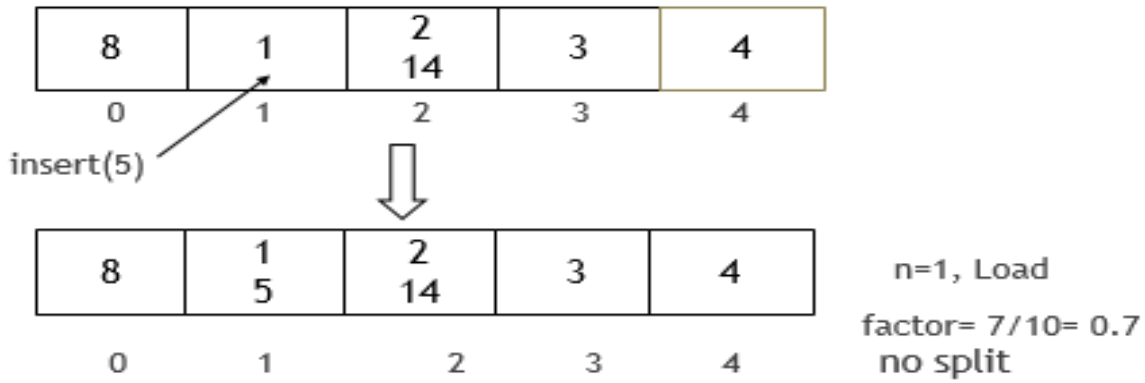
3,2,4,1,8,14,5,10,7,12

When inserting the sixth record (using h_0(key) = key mod (M),) we would have

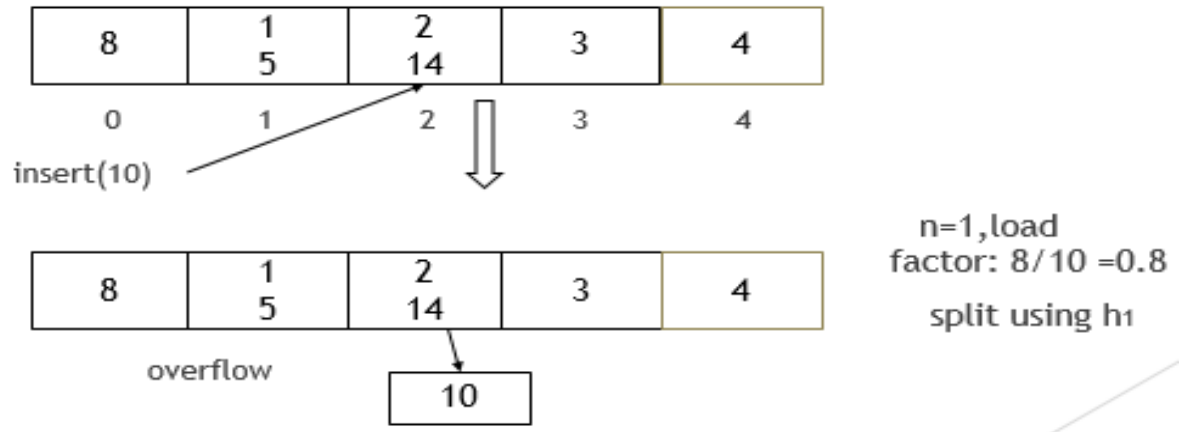




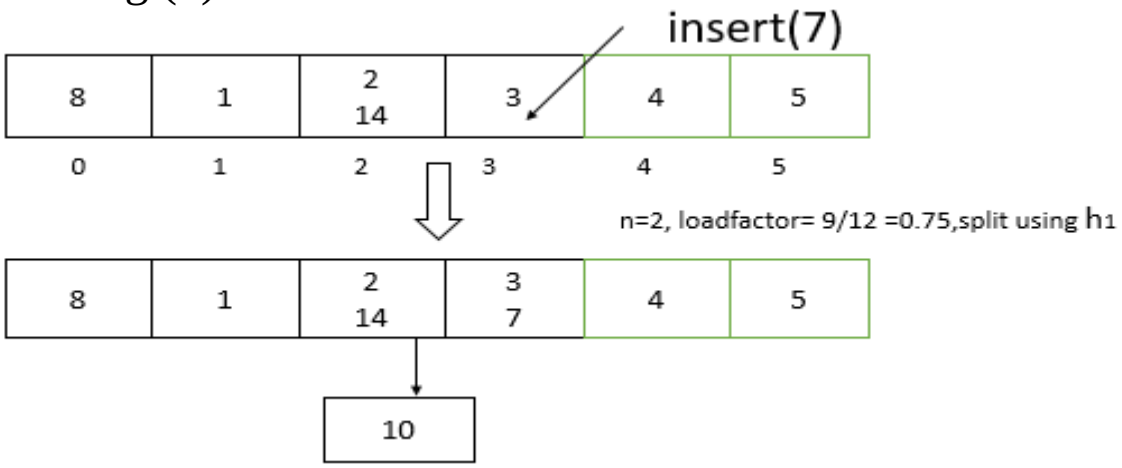
Inserting (5)



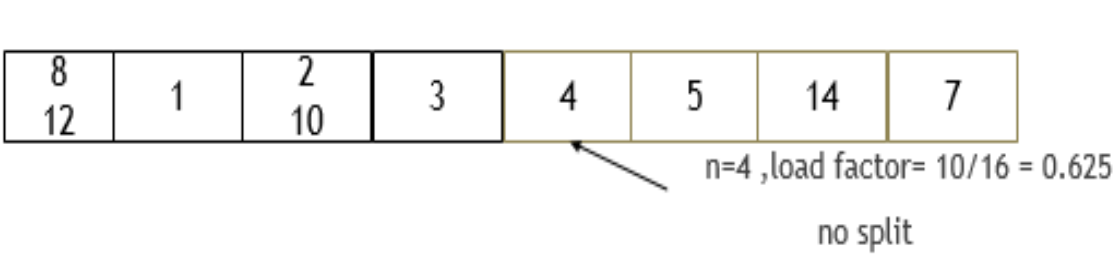
Inserting (10)



Inserting (7)



After inserting all elements hash table will look as
Final Hash table



- At this point, all the 4 (M) buckets are split. n should be set to 0.
- It begins a second phase.
- In the second phase, we will use h_1 to insert records and h_2 to split a bucket.
- $h_1(K) = K \bmod 2M$ and $h_2(K) = K \bmod 4M$.

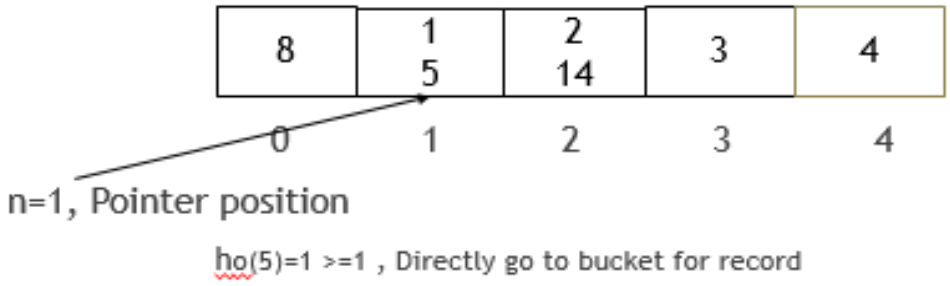
Searching Algorithm

A search scheme is needed to map a key *k* to a bucket, either when searching for an existing record or when inserting a new record. The search scheme works as follows:

- If $h_i(key) >= n$, choose bucket $h_i(key)$ since the bucket has not been split yet in the current round.
- If $h_i(key) <$, choose bucket $h_i+1(key)$, which can be either $hi(key)$ or its split image .

- Where n is the pointer position.

Searching Example



In above Figure, n = 1. To search for record 5, since $h_0(5) = 1 \geq n$, one directly goes to bucket to find the record. But to search for record 4, since $h_0(4) = 0 < n$, one needs to use h_1 to decide the actual bucket. In this case, the record should be searched in bucket $h_1(4) = 4$.[4]

Implementation Details

This section includes the hardware specification section along with software specification and code diagrams. Implementation of linear hashing can be easily done by using various header files in C++.

Hardware specification:

- Intel core i5
- 4 GB RAM
- 750 Hard Disk

Software specification:

- Visual Studio 2013
- Microsoft window 10

Language Used: C++

Code_Diagrams

```
#include "stdafx.h"
#include <cstring>
#include <vector>
#include <math.h>
#include <iomanip>
#include <fstream>
#include<stdio.h>
#include <stdlib.h>
#include <string>
#include<iostream>

using namespace std;

int power(int a, int b)
{
    for (int i = 0; i<b; i++)
    {
        a *= a;
    }
    return a;
};

struct Bucket
{
string  word;
};

const int size = 100;

class LProbing
{
private:
    int a; //a constant which is used in hashing
    int cursize;  //current size of hash table
    Bucket *Table;  //pointer to array of struct
    int loadfactor; //ratio of number of elements entered over size of hashtable
    int n;  //number of elements entered
    Bucket table[size];   //array of structs
public:
    LProbing(int A);  //constant is decided by user
    void resize();
    void insert(string word);
    void Lookup(string word);
};

LProbing::LProbing(int A)
{
    cursize = size;
    a = A;
    Table = table;
    loadfactor = 0;  //initially loadfactor is 0 as number of elements entered are 0
    n = 0;
```

Fig 1. Code

```
void LProbing::resize()
{
    //cout << "resize" << endl;
    printf("\nresize");
    loadfactor = n / cursize;   //ensuring if resize needs to be done
    if (loadfactor <= 0.7)
    {
        return;
    }
    const int s = 2 * cursize;
    Bucket PTable[100];//change
    for (int i = 0; i<cursize; i++)
    {
        if (Table[i].word.empty())
            continue;

        //rehashing the word onto the new array
        string w = Table[i].word;
        int key = 0;
        for (int j = 0; j<w.size(); j++)
        {
            unsigned char b = (unsigned char)w[j];
            key += (int)power(a, i)*b;
        }
        key = key % (2 * cursize);
        PTable[key].word = w;  //entering the word in the new array
    }
    Table = PTable;  //putting pointer equal to new array
    cursize = 2 * cursize;  //doubling the current size of array
}

void LProbing::insert(string word)
{
    //cout << "1" << endl;
    printf("\nIteration : %d",n);
    n++;  //incrementing the number of elements entered with every call to insert

    //if loadfactor is greater than 0.5, resize array
    loadfactor = n / cursize;
    if (loadfactor>0.5)
    {
        resize();
    }
    //hashing the word
    int k = 0;
    for (int i = 0; i<word.size(); i++)
    {
        unsigned char b = (unsigned char)word[i];
        int c = (int)((power(a, i))*b);
        k += c;
        //cout << c << endl;
        printf("\nHash Value: %d",c);
```

Fig 2. Code

```
    int key = 0;
    key = k%cursize;
    //cout << key << endl;
    printf("\nKey: %d", key);
    //if the respective key index is empty enter the word in that slot
    if (Table[key].word.empty() == 1)
    {
        //cout << "initial empty slot" << endl;
        printf("\n\ninitial empty slot");
        Table[key].word = word;
    }
    else  //otherwise enter in the next slot
    {
        //searching array for empty slot
        while (Table[key].word.empty() == 0)
        {
            k++;
            key = k%cursize;
        }
        //when empty slot found,entering the word in that bucket
        Table[key].word = word;
        //cout << "word entered" << endl;
        printf("\nValue Inserted");
    }
}

int main()
{
    char a;
    LProbing H(35);
    ifstream fin;
    fin.open("dict.txt");
    vector<string> D;

    string d;
    while (getline(fin, d))
    {
        if (!d.empty())
        {
            D.push_back(d);
        }
    }
    fin.close();
    for (int i = 0; i<D.size(); i++)
    {
        printf("\nInserting Element %S:", D[i]);
        cout << D[i];
        H.insert(D[i]);
    }

    //system("PAUSE");
```

Fig 3. Code

Output Diagram

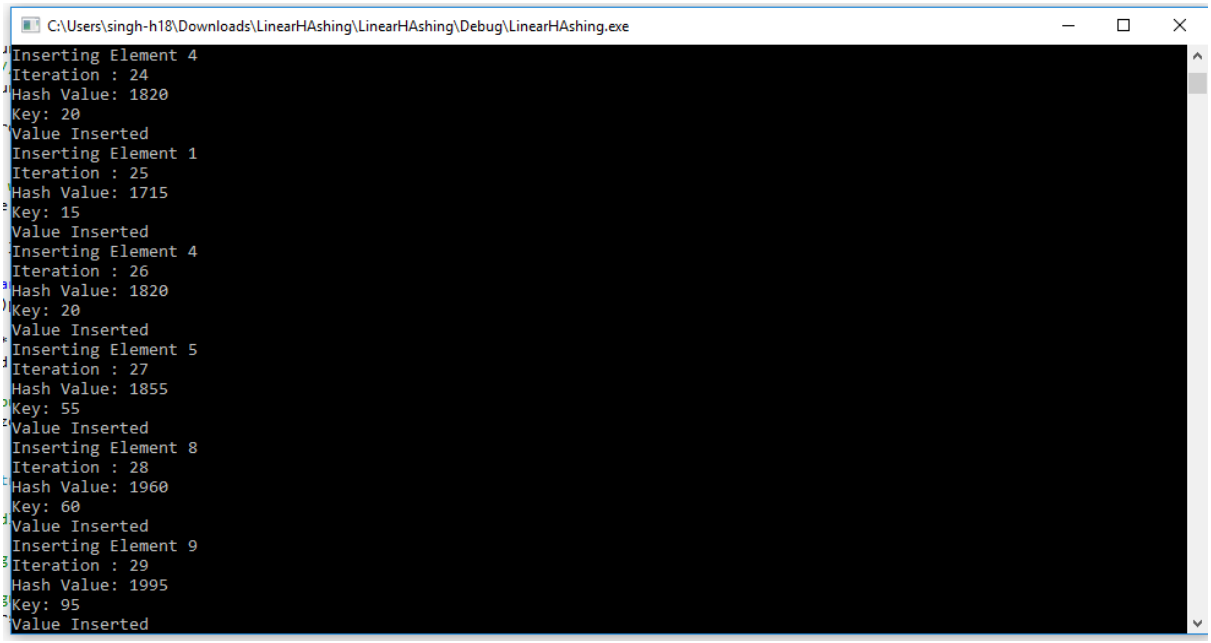


Fig 4. Output

Time Complexity

Time complexity is demonstrated in big O notation. Table no. 1 depicted about the average case complexity of insertion and search algorithm of linear hashing which is O(1) whereas the worst case complexity for both algorithm is O(n). Which clearly shows that linear hashing is faster than static and extendible hashing.

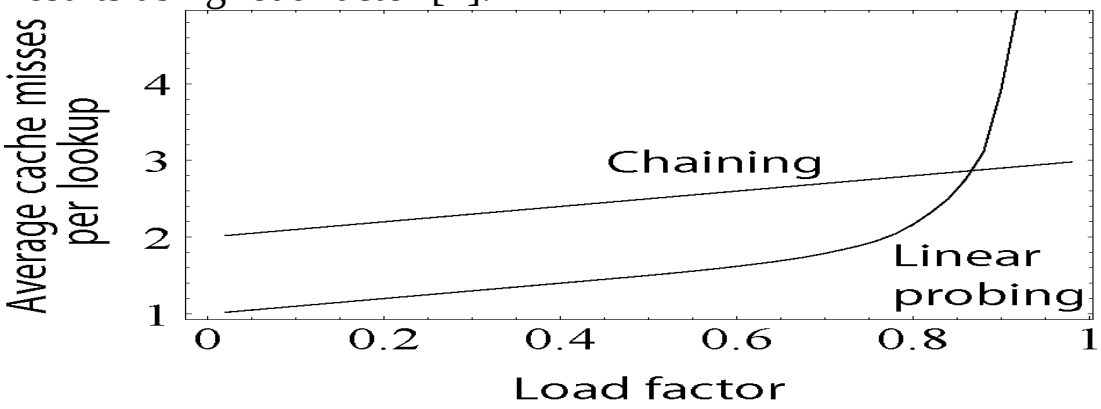
Table no. 1

Algorithm	Average Case	Worst Case
Insertion	O(1)	O(n)
Search	O(1)	O(n)

Test Results

This open addressing technique is that the number of inserted data files cannot exceed the number of slots in the bucket array. However, the hash functions in linear hashing, its performance dramatically degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing.

Results using load factor [2].



This above given graph compares the average number of cache misses required to look up elements in tables with chaining and linear hashing. As, the table passes the 0.8 mark load factor, after that Performance of linear hashing degrades drastically.
The results for time complexity is given in the above table no. 1. And The Output diagram shows the insertion of elements with their key description and hash value.
Apart from this, the results for time complexity is shown in the above section. Which shows that linear hashing is faster than the other hashing techniques such as static and extendible hashing.

Applications of linear hashing

- Linear Hashing has been implemented into commercial database systems. It is used in applications where exact match query is the most important. Query such as hash join.
- Compilers use hash tables to keep track of declared variables.
- A hash table can be used for on-line spelling checkers if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time.
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again.
- Hash functions can be used to quickly check for inequality if two elements hash to different values they must be different.
- Storing sparse data.

References

1. https://en.wikipedia.org/wiki/Linear_hashing
2. Cormen, Thomas H.*;* Leiserson, Charles E.*;* Rivest, Ronald L.*;* Stein, Clifford *(2009).* Introduction to Algorithms *(3rd ed.). Massachusetts Institute of Technology. pp. 253–280.* ISBN 978-0-262-03384-8
3. https://en.wikipedia.org/wiki/Hash_table
4. http://arl.wustl.edu/~sailesh/download_files/Limited_Edition/hash/Dynamic%20Hash%20Tables.pdf
5. http://hackthology.com/pdfs/Litwin-1980-Linear_Hashing.pdf
6. http://cgi.di.uoa.gr/~ad/MDE515/e_ds_linearhashing.pdf