**K L EDUCATIONAL FOUNDATION**

**Department of Computer Science Engineering**

**A Project Based Lab Report**

**On**

**PERFECT HASH TABLE-BASED TELEPHONE DIRECTORY**

**By**

U. MAHESH – 170031326

P. GEETA SANDEEP – 170031004

B. SAM KUMAR – 170031598

B. LOHITH BABU – 170030103

**Batch No:13**

**Under the Esteemed Guidance of**

Mr. Mohammed A R Quadri

Assistant Professor

**A close up of a sign

Description generated with high confidence**

Green fields, Vaddeswaram, Guntur Dist-522502, Andhra Pradesh, India

K L EDUCATIONAL FOUNDATION

**DEPARTMENT COMPUTER SCIENCE ENGINEERING**

**A close up of a sign

Description generated with high confidence**

***DECLARATION***

We hereby declare that this project-based lab report titled **“PERFECT HASH TABLE BASED TELEPHONE DIRECTORY”** has been prepared by us in partial fulfilment of the requirements for the award of degree “**BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE ENGNEERING**” during the Academic year 2018-2019.

We also declare that this project-based lab report is of our own efforts and it has not been submitted to any other university for the award of any degree.

U. MAHESH – 170031326

P. GEETA SANDEEP – 170031004

B. SAM KUMAR – 170031598

B. LOHITH BABU –170030103

K L EDUCATIONAL FOUNDATION

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**logo9**

**CERTIFICATE**

This is to certify that the project based laboratory report entitled ” **PREFECT HASH TABLE BASED TELEPHONE DIRECTORY** “ submitted U. Mahesh (170031326), P. Geeta Sandeep (170031004), B. Sam Kumar (170031598), B. Lohith Babu (170030103), to the Department of **Computer Science Engineering**, KL Educational Foundation in partial fulfilment of the requirements for the completion of a project based Laboratory in “**DATA STRUCTURES (17CS1102)**” course in II B Tech I Semester, is a bonafide record of the work carried out by us during the academic year 2018 – 2019.

**Project guide Head of the Department**

Mr. Mohammed A R Quadri Dr. V. Hari Kiran

**ACKNOWLEDGMENTS**

It is great pleasure for me to express my gratitude to our honourable President **Sri. Koneru Satya Narayana,** for giving the opportunity and platform with facilities in accomplishing the project-based laboratory report.

I express the sincere gratitude to our principal **Dr. K. Subba Rao** for his administration towards our academic growth.

I express sincere gratitude to our Coordinator and **HOD-BES Dr. HARI KIRAN VEGE** for her leadership and constant motivation provided in successful completion of our academic semester. I record it as my privilege to deeply thank for providing us the efficient faculty and facilities to make our ideas into reality

I express my sincere thanks to our faculty in-charge Mr. Mohammed A R Quadri and project supervisor Mr. Mohammed A R Quadri for her/his novel association of ideas, encouragement, appreciation and intellectual zeal which motivated us to venture this project successfully.

Finally, it is pleased to acknowledge the indebtedness to all Teaching and Non-teaching faculty those who devoted themselves directly or indirectly to make this project report success

By

U. MAHESH – 170031326

P. GEETA SANDEEP – 170031004

B. SAM KUMAR – 170031598

B. LOHITH BABU –170030103

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Topic** | **Page No** |
| 1 | ABSTRACT | 6 |
| 2 | INTRODUCTION | 7 |
| 3 | DESCRIPTION | 8-10 |
| 4 | SYSTEM REQUIREMENTS | 11 |
| 5 | FUNCTIONAL REQUIREMENTS | 12 |
| 6 | ADVANTAGES & DIS-ADVANTAGES | 13 |
| 7 | FUTURE ENHANCEMENT | 14 |
| 8 | SOURCE CODE | 15-55 |
| 9 | OUTPUT | 56-58 |
| 10 | CONCLUSION & REFERENCE | 59 |

**1.ABSTRACT**

This project is designed to develop and demonstrate perfect hash table data structure and its applications for Telephone Directory with appropriate algorithms having following tasks. A hash table for integers and strings with a universal hash function, A perfect hash table for integers and create a set of programs that looks up telephone numbers. Given input is a simple text file which contains names and phone numbers. When the program begins, it reads the file and loads the data into two files outflies and overflow file. A hash is used to find phone numbers. The outflies is used here as the hash table. First, this file is prepared by filling it with table Size \* bucket Size empty records (one record is simply a certain number of bytes). Next, all entries of names are transferred to outflies to buckets indicated by the hash function. This transfer is performed by the function insert () which includes the hashed item in the bucket indicated by the hash function or in overflow if the bucket is full. Elements that cannot be hashed to the corresponding bucket in this file are stored in the file overflow. At the end of the session both files are combined and sorted to replace the contents of the original file names. The running time/Time complexity of the routines developed are compared with other Hash Tables and proved to be efficient.

2. **INTRODUCTION**

A hash table, or a hash map, is a data structure that associates keys with values. The primary operation it supports efficiently is a lookup: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number). It works by transforming the key using a [hash function](https://en.wikipedia.org/wiki/hash_function) into a hash, a number that the hash table uses to locate the desired value. This hash maps directly to a bucket in the array of key/value pairs, hence the name hash map. The mapping method lets us directly access the storage location for any key/value pair.

A good hash function is essential for good hash table performance. A poor choice of hash function is likely to lead to clustering behaviour, in which the probability of keys mapping to the same hash bucket (i.e. a collision) is significantly greater than would be expected from a random function. A nonzero probability of collisions is inevitable in any hash implementation, but the number of operations to resolve collisions usually scales linearly with the number of keys mapping to the same bucket, so excess collisions will degrade performance significantly. In addition, some hash functions are computationally expensive, so the amount of time (and, in some cases, memory) taken to compute the hash may be burdensome.

Choosing a good hash function is tricky. The literature is replete with poor choices, at least when measured by modern standards. For example, the very popular multiplicative hash advocated by Knuth in The Art of Computer Programming (see reference below) has particularly poor clustering behaviour. However, since poor hashing merely degrades hash table performance for particular input key distributions, such problems go undetected far too often.

The literature is also sparse on the criteria for choosing a hash function. Unlike most other fundamental algorithms and data structures, there is no universal consensus on what makes a "good" hash function. The remainder of this section is organized by three criteria: simplicity, speed, and strength, and will survey algorithms known to perform well by these criteria.

3. **DESCRIPTION**

In this project we will design, develop and demonstrates prefect hash table and its applications. Student has to implement the following tasks - A hash table for integers and integers with a universal hash function, A perfect hash table for integers, sort parallel arrays, count the number of times each word appears, Merging Ordered Lists, Threaded Binary Tree ADT.

We are performing this project in 4 different modules. i.e.,

**Module1**:- A hash table for integers with a universal hash function.

**Module2:-** A hash table for strings with a universal hash function.

**Module3**:- A perfect hash table for integers.

**Module4**:-A Sorting techniques, Binary trees.

These functions will be handled with the help of following sub functions: -

**Module: 1**

**A hash table for integers with a universal hash function.**

1. Function: get\_node()

2. Function: return\_node

3. Function: create\_hashtable

4. Function: universalhashfunction

5. Function: find

6. Function: insert

7. Function: delete

8. Function: find\_mtf

9. Function: list\_table

10. Function: main()

**Module: 2**

**A hash table for strings with a universal hash function.**

1. Function: get\_node()

2. Function: return\_node

3. Function: create\_hashtable

4. Function: universalhashfunction

5. Function: find

6. Function: insert

7. Function: delete

8. Function: find\_mtf

9. Function: list\_table

10. Function: main()

**Module: 3**

**A perfect hash table for integers.**

1. Function: create\_perf\_hash

2. Function: find

3. Function: main()

4. Function: compute secondary table sizes and their offset

5. Function: mark bucket as defect

6. Function: secondary hash tables

**Module: 4**

1. Rewrite insertionSort3 to illustrate how to sort parallel arrays: Each time a name is moved during the sorting process, the corresponding ID number must also be moved. Since the name and ID number must be moved “in parallel,” we say we are doing a *parallel sort* or we are sorting *parallel arrays*.

2. Write a program to read an English passage and count the number of times each word appears. The output consists of an alphabetical listing of the words and their frequencies. This is a typical “search and insert” situation.

3. Merging Ordered Lists: Merging is the process by which two or more ordered lists are combined into one ordered list.

4. Threaded Binary Tree ADT

5. Function: Insert

6. Function: Delete

7. Function: Search

8. Function: Inorder traversal

**4. SYSTEM REQUIREMENTS**

* **SOFTWARE REQUIREMENTS:**

The major software requirements of the project are as follows:

Language : C Language

Operating system **:**  windows 10

C Compiler : Dev C++

* **HARDWARE REQUIREMENTS:**

The hardware requirements that map towards the software are as follows:

RAM : 8 GB

Processor : Intel Core i7

Hard Disk : 1 TB

**5.FUNCTIONAL REQUIREMENTS**

**Purpose:**

The hash function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There's no need to "reverse engineer" the hash function by analyzing the hashed values. In fact, the ideal hash function can't be derived by such analysis. A good hash function also should not produce the same hash value from two different inputs. If it does, this is known as a *collision*.

**Use of universal hashing:**

In [mathematics](https://en.wikipedia.org/wiki/Mathematics) and [computing](https://en.wikipedia.org/wiki/Computing), universal hashing (in a [randomized algorithm](https://en.wikipedia.org/wiki/Randomized_algorithm) or data structure) refers to selecting a [hash function](https://en.wikipedia.org/wiki/Hash_function) at random from a family of hash functions with a certain mathematical property (see definition below). This guarantees a low number of collisions in [expectation](https://en.wikipedia.org/wiki/Expected_value), even if the data is chosen by an adversary. Universal hashing has numerous uses in computer science, for example in implementations of [hash tables](https://en.wikipedia.org/wiki/Hash_table), [randomized algorithms](https://en.wikipedia.org/wiki/Randomized_algorithm), and [cryptography](https://en.wikipedia.org/wiki/Cryptography).

**Purpose of universal hashing with integers:**

Usually, the goal of hashing is to obtain a **low** number of collisions (keys from that land in the same bin). A deterministic hash function cannot offer any guarantee in an adversarial setting if the size of is greater than, since the adversary may choose to be precisely the preimage of a bin.

**Reduces the time complexity for larger programs:**

Hash tables are among the most important data structures known to mankind. Through hashing, the address of each stored object is calculated as a function of the object's contents. Because they do not require exorbitant space and, in practice, allow for constant-time dictionary operations (insertion, lookup, deletion), hash tables are often employed in the indexation of large amounts of data.

**Use of perfect hashing with strings**

This refers to hashing a *variable-sized* vector of machine words. If the length of the string can be bounded by a small number, it is best to use the vector solution from above (conceptually padding the vector with zeros up to the upper bound). The space required is the maximal length of the string, but the time to evaluate {\displaystyle h(s)}is just the length of {\displaystyle s}. As long as zeroes are forbidden in the string, the zero-padding can be ignored when evaluating the hash function without affecting universality.

**6.ADVANTAGES &DISADVANTAGES**

**1.Advantages: -**

1.The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more).

2.Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.  
3. If the set of key-value pairs is fixed and known ahead of time (so insertions and deletions are not allowed).

4.one may reduce the average lookup cost by a careful choice of the hash function, bucket table size, and internal data structures.

5.In particular, one may be able to devise a hash function that is collision-free, or even perfect. In this case the keys need not be stored in the table.

**2.DISADVANTAGES: -**

1.Hash tables can be more difficult to implement than self-balancing binary search trees. Choosing an effective hash function for a specific application is more an art than a science. In open-addressed hash tables it is fairly easy to create a poor hash function.

2.The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order.

3.Therefore, there is no efficient way to efficiently locate an entry whose key is nearest to a given key.

4.Listing all n entries in some specific order generally requires a separate sorting step, whose cost is proportional to log(n) per entry.

**7. FUTURE ENHANCEMENT**

FUTURE ENHANCEMENT In e-commerce, many types of users can access the website. But this work focuses only on identification of potential users. In future, three 162 types of users can be identified from weblog files such as frequent users, Synthetic user and Potential user. It helps to improve the browsing behaviour of user and predict their interest. Heuristic based user and session algorithm can be used to optimize the pre-processing procedure and produce the optimized and cleaned weblog. This improved technique can be implemented in online phase of the website. WUM has become a well-established field of research, where WUM processes can be applied to integrate the semantics within website design and aim to improve the results of WUM applications. Efforts in this direction seem to be a productive way to create much more effective WUM based systems that are consistent with the emergence and proliferation of the semantic web. Furthermore, the web mining seems to be adequate for applying other mining techniques in parallel with the current algorithms. The system extensions for augmentative and detailed analysis could be the use of queries in one-dimensional representations, persistent data structures and multidimensional data structures in order to apply queries based on time.

We have now seen two approaches to implementing collections classes:

● Dynamic arrays: allocating space and doubling it as needed.

●Linked lists: Allocating small chunks of space one at a time.

● These approaches are good for linear structures, where the elements are stored in some order. Associative Structures

● Not all structures are linear.

● How do we implement Map, Set, and Lexicon?

● There are many options, as you'll see in the next two weeks:

● Hash tables. ● Binary search trees. ● Tries. ● DAWGs.

**8. SOURCE CODE**

//MODULE1  
  
#include <stdio.h>  
#include <stdlib.h>  
#define BLOCKSIZE 256  
typedef int object\_t;  
typedef int key\_t;  
#define MAXP 46337 /\* prime, and 46337\*46337 < 2147483647 \*/  
typedef struct l\_node { key\_t key;  
 object\_t \*obj;  
 struct l\_node \*next; } list\_node\_t;  
typedef struct { int a; int b; int size; } hf\_param\_t;  
typedef struct { int size;  
 list\_node\_t \*\*table;   
 int (\*hash\_function)(key\_t, hf\_param\_t);   
 hf\_param\_t hf\_param; } hashtable\_t;  
list\_node\_t \*currentblock = NULL;  
int size\_left;  
list\_node\_t \*free\_list = NULL;  
list\_node\_t \*get\_node()  
{ list\_node\_t \*tmp;  
 if( free\_list != NULL )  
 { tmp = free\_list;  
 free\_list = free\_list -> next;  
 }  
 else  
 { if( currentblock == NULL || size\_left == 0)  
 { currentblock =   
 (list\_node\_t \*) malloc( BLOCKSIZE \* sizeof(list\_node\_t) );  
 size\_left = BLOCKSIZE;  
 }  
 tmp = currentblock++;  
 size\_left -= 1;  
 }  
 return( tmp );  
}  
void return\_node(list\_node\_t \*node)  
{ node->next = free\_list;  
 free\_list = node;  
}  
hashtable\_t \*create\_hashtable(int size)  
{ hashtable\_t \*tmp; int i;  
 int a, b;  
 int universalhashfunction(key\_t, hf\_param\_t);  
 if( size >= MAXP )  
 exit(-1); /\* should not be called with that large size \*/  
 /\* possibly initialize random number generator here \*/  
 tmp = (hashtable\_t \*) malloc( sizeof(hashtable\_t) );  
 tmp->size = size;  
 tmp->table = (list\_node\_t \*\*)malloc( size\*sizeof(list\_node\_t \*));  
 for(i=0; i<size; i++)  
 (tmp->table)[i] = NULL;  
 tmp->hf\_param.a = rand()%MAXP;  
 tmp->hf\_param.b = rand()%MAXP;  
 tmp->hf\_param.size = size;  
 tmp->hash\_function = universalhashfunction;  
 return( tmp );  
}  
int universalhashfunction(key\_t key, hf\_param\_t hfp)  
{ return( ((hfp.a\*key + hfp.b)%MAXP)%hfp.size );  
}  
object\_t \*find(hashtable\_t \*ht, key\_t query\_key)  
{ int i; list\_node\_t \*tmp\_node;  
 i = ht->hash\_function(query\_key, ht->hf\_param );  
 tmp\_node = (ht->table)[i];  
 while( tmp\_node != NULL && tmp\_node->key != query\_key )  
 tmp\_node = tmp\_node->next;  
 if( tmp\_node == NULL )  
 return( NULL ); /\* not found \*/  
 else  
 return( tmp\_node->obj ); /\* key found \*/  
}  
void insert(hashtable\_t \*ht, key\_t new\_key, object\_t \*new\_obj)  
{ int i; list\_node\_t \*tmp\_node;  
 i = ht->hash\_function(new\_key, ht->hf\_param );  
 tmp\_node = (ht->table)[i];  
 /\* insert in front \*/  
 (ht->table)[i] = get\_node();  
 ((ht->table)[i])->next = tmp\_node;  
 ((ht->table)[i])->key = new\_key;  
 ((ht->table)[i])->obj = new\_obj;  
}  
  
object\_t \*delete(hashtable\_t \*ht, key\_t del\_key)  
{ int i; list\_node\_t \*tmp\_node; object\_t \*tmp\_obj;  
 i = ht->hash\_function(del\_key, ht->hf\_param );  
 tmp\_node = (ht->table)[i];  
 if( tmp\_node == NULL )  
 return( NULL ); /\* list empty, delete failed \*/  
 if( tmp\_node->key == del\_key ) /\* if first in list \*/  
 { tmp\_obj = tmp\_node->obj;  
 (ht->table)[i] = tmp\_node->next;  
 return\_node( tmp\_node );  
 return( tmp\_obj );  
 }   
 /\* list not empty, delete not first in list \*/  
 while( tmp\_node->next != NULL   
 && tmp\_node->next->key != del\_key )  
 tmp\_node = tmp\_node->next;  
 if( tmp\_node->next == NULL )  
 return( NULL ); /\* not found, delete failed \*/  
 else  
 { list\_node\_t \*tmp\_node2; /\* unlink node \*/  
 tmp\_node2 = tmp\_node->next;  
 tmp\_node->next = tmp\_node2->next;  
 tmp\_obj = tmp\_node2->obj;  
 return\_node( tmp\_node2 );  
 return( tmp\_obj );  
 }  
}  
object\_t \*find\_mtf(hashtable\_t \*ht, key\_t query\_key)  
{ int i; list\_node\_t \*front\_node, \*tmp\_node1, \*tmp\_node2;  
 i = ht->hash\_function(query\_key, ht->hf\_param );  
 front\_node = tmp\_node1 = (ht->table)[i]; tmp\_node2 = NULL;  
 while( tmp\_node1 != NULL && tmp\_node1->key != query\_key )  
 { tmp\_node2 = tmp\_node1;  
 tmp\_node1 = tmp\_node1->next;  
 }  
 if( tmp\_node1 == NULL )  
 return( NULL ); /\* not found \*/  
 else /\* key found \*/  
 { if( tmp\_node1 != front\_node ) /\* move to front \*/  
 { tmp\_node2->next = tmp\_node1->next; /\* unlink \*/  
 tmp\_node1->next = front\_node;   
 (ht->table)[i] = tmp\_node1;  
 }  
 return( tmp\_node1->obj );   
 }  
}  
void list\_table(hashtable\_t \*ht)  
{ int i; list\_node\_t \*tmp\_node;  
 for(i = 0; i< ht->size; i++ )  
 { printf("|");  
 tmp\_node = (ht->table)[i];  
 while( tmp\_node != NULL )  
 { printf("%d ", (tmp\_node->key) );   
 tmp\_node = tmp\_node->next;  
 }  
 }  
}

int main()  
{ hashtable\_t \*ha;  
 char nextop;  
 ha = create\_hashtable(20);  
 printf("Made Hashtable\n");  
 while( (nextop = getchar())!= 'q' )  
 { if( nextop == 'i' )  
 { int inskey, \*insobj;  
 insobj = (int \*) malloc(sizeof(int));  
 scanf(" %d", &inskey);  
 \*insobj = 10\*inskey+2;  
 insert( ha, inskey, insobj );  
 printf(" inserted key = %d, object value = %d\n", inskey, \*insobj);  
 }   
 if( nextop == 'f' )  
 { int findkey, \*findobj;  
 scanf(" %d", &findkey);  
 findobj = find( ha, findkey);  
 if( findobj == NULL )  
 printf(" basic find failed, for key %d\n", findkey);  
 else  
 printf(" basic find successful, found object %d\n", \*findobj);  
 findobj = find\_mtf( ha, findkey);  
 if( findobj == NULL )  
 printf(" find (mtf) failed, for key %d\n", findkey);  
 else  
 printf(" find (mtf) successful, found object %d\n", \*findobj);  
 }  
 if( nextop == 'd' )  
 { int delkey, \*delobj;  
 scanf(" %d", &delkey);  
 delobj = delete( ha, delkey);  
 if( delobj == NULL )  
 printf(" delete failed for key %d\n", delkey);  
 else  
 printf(" delete successful, deleted object %d\n", \*delobj);  
 }  
 if( nextop == '?' )  
 { printf(" Checking table\n");   
 list\_table(ha);  
 printf(" Finished Checking table\n");   
 }  
 }  
 return(0);  
}

//MODULE2

#include <stdio.h>

#include <stdlib.h>

#define BLOCKSIZE 256

typedef char object\_t;

#define MAXP 46337 /\* prime, and 46337\*46337 < 2147483647 \*/

typedef struct l\_node { char \*key;

object\_t \*obj;

struct l\_node \*next; } list\_node\_t;

typedef struct htp\_l\_node { int a;

struct htp\_l\_node \*next; } htp\_l\_node\_t;

typedef struct { int b; int size;

struct htp\_l\_node \*a\_list;} hf\_param\_t;

typedef struct { int size;

list\_node\_t \*\*table;

int (\*hash\_function)(char \*, hf\_param\_t);

hf\_param\_t hf\_param; } hashtable\_t;

list\_node\_t \*currentblock = NULL;

int size\_left;

list\_node\_t \*free\_list = NULL;

list\_node\_t \*get\_node()

{ list\_node\_t \*tmp;

if( free\_list != NULL )

{ tmp = free\_list;

free\_list = free\_list -> next;

}

else

{ if( currentblock == NULL || size\_left == 0)

{ currentblock =

(list\_node\_t \*) malloc( BLOCKSIZE \* sizeof(list\_node\_t) );

size\_left = BLOCKSIZE;

}

tmp = currentblock++;

size\_left -= 1;

}

return( tmp );

}

void return\_node(list\_node\_t \*node)

{ node->next = free\_list;

free\_list = node;

}

hashtable\_t \*create\_hashtable(int size)

{ hashtable\_t \*tmp; int i;

int universalhashfunction(char \*, hf\_param\_t);

if( size >= MAXP )

exit(-1); /\* should not be called with that large size \*/

tmp = (hashtable\_t \*) malloc( sizeof(hashtable\_t) );

tmp->size = size;

tmp->table = (list\_node\_t \*\*)malloc( size\*sizeof(list\_node\_t \*));

for(i=0; i<size; i++)

(tmp->table)[i] = NULL;

tmp->hf\_param.b = rand()%MAXP;

tmp->hf\_param.size = size;

tmp->hf\_param.a\_list = (htp\_l\_node\_t \*) get\_node();

tmp->hf\_param.a\_list->next = NULL;

tmp->hash\_function = universalhashfunction;

return( tmp );

}

int universalhashfunction(char \*key, hf\_param\_t hfp)

{ int sum;

htp\_l\_node\_t \*al;

sum = hfp.b;

al = hfp.a\_list;

while( \*key != '\0' )

{ if( al->next == NULL )

{ al->next = (htp\_l\_node\_t \*) get\_node();

al->next->next = NULL;

al->a = rand()%MAXP;

}

sum += ( (al->a)\*((int) \*key))%MAXP;

key += 1;

al = al->next;

}

return( sum%hfp.size );

}

object\_t \*find(hashtable\_t \*ht, char \*query\_key)

{ int i; list\_node\_t \*tmp\_node;

char \*tmp1, \*tmp2;

i = ht->hash\_function(query\_key, ht->hf\_param );

tmp\_node = (ht->table)[i];

while( tmp\_node != NULL )

{ tmp1 = tmp\_node->key; tmp2 = query\_key;

while( \*tmp1 != '\0' && \*tmp2 != '\0' && \*tmp1 == \*tmp2 )

{ tmp1++; tmp2++; }

if( \*tmp1 != \*tmp2 ) /\*strings not equal \*/

tmp\_node = tmp\_node->next;

else /\* strings equal: correct entry found \*/

break;

}

if( tmp\_node == NULL )

return( NULL ); /\* not found \*/

else

return( tmp\_node->obj ); /\* key found \*/

}

void insert(hashtable\_t \*ht, char \*new\_key, object\_t \*new\_obj)

{ int i; list\_node\_t \*tmp\_node;

i = ht->hash\_function(new\_key, ht->hf\_param );

tmp\_node = (ht->table)[i];

/\* insert in front \*/

(ht->table)[i] = get\_node();

((ht->table)[i])->next = tmp\_node;

((ht->table)[i])->key = new\_key;

((ht->table)[i])->obj = new\_obj;

}

object\_t \*delete(hashtable\_t \*ht, char \* del\_key)

{ int i; list\_node\_t \*tmp\_node; object\_t \*tmp\_obj;

char \*tmp1, \*tmp2;

i = ht->hash\_function(del\_key, ht->hf\_param );

tmp\_node = (ht->table)[i];

if( tmp\_node == NULL )

return( NULL ); /\* list empty, delete failed \*/

/\* test first item in list \*/

tmp1 = tmp\_node->key; tmp2 = del\_key;

while( \*tmp1 != '\0' && \*tmp2 != '\0' && \*tmp1 == \*tmp2 )

{ tmp1++; tmp2++; }

if( \*tmp1 == \*tmp2 )/\* strings equal: correct entry found \*/

{ tmp\_obj = tmp\_node->obj; /\* delete first entry in list \*/

(ht->table)[i] = tmp\_node->next;

return\_node( tmp\_node );

return( tmp\_obj );

}

/\* list not empty, delete not first in list \*/

while( tmp\_node->next != NULL )

{ tmp1 = tmp\_node->next->key; tmp2 = del\_key;

while( \*tmp1 != '\0' && \*tmp2 != '\0' && \*tmp1 == \*tmp2 )

{ tmp1++; tmp2++; }

if( \*tmp1 != \*tmp2 ) /\* strings not equal \*/

tmp\_node = tmp\_node->next;

else /\* strings equal: correct entry found \*/

break;

}

if( tmp\_node->next == NULL )

return( NULL ); /\* not found, delete failed \*/

else

{ list\_node\_t \*tmp\_node2; /\* unlink node \*/

tmp\_node2 = tmp\_node->next;

tmp\_node->next = tmp\_node->next->next;

tmp\_obj = tmp\_node2->obj;

return\_node( tmp\_node2 );

return( tmp\_obj );

}

}

void list\_table(hashtable\_t \*ht)

{ int i; list\_node\_t \*tmp\_node;

for(i = 0; i< ht->size; i++ )

{ printf("|");

tmp\_node = (ht->table)[i];

while( tmp\_node != NULL )

{ printf("%s ", (tmp\_node->key) );

tmp\_node = tmp\_node->next;

}

}

}

int main()

{ hashtable\_t \*ha;

char nextop;

ha = create\_hashtable(20);

printf("Made Hashtable\n");

while( (nextop = getchar())!= 'q' )

{ if( nextop == 'i' )

{ char \*ins\_key, \*ins\_obj; int success;

ins\_key = (char \*) malloc(100\*sizeof(char));

ins\_obj = (char \*) malloc(100\*sizeof(char));

scanf(" %s %s", ins\_key, ins\_obj);

insert( ha, ins\_key, ins\_obj );

printf(" inserted key = %s, object = %s\n", ins\_key, ins\_obj);

}

if( nextop == 'f' )

{ char find\_key[100], \*find\_obj;

scanf(" %s", find\_key);

find\_obj = find( ha, find\_key);

if( find\_obj == NULL )

printf(" find failed, for key %s\n", find\_key);

else

printf(" find successful, found object %s\n", find\_obj);

}

if( nextop == 'd' )

{ char del\_key[100], \*del\_obj;

scanf(" %s", del\_key);

del\_obj = delete( ha, del\_key);

if( del\_obj == NULL )

printf(" delete failed for key %s\n", del\_key);

else

printf(" delete successful, deleted object %s,\n", del\_obj);

}

if( nextop == '?' )

{ printf(" Checking Hashtable: listing keys\n");

list\_table(ha);

printf("\n Finished Checking Hashtable\n");

}

}

return(0);

}

//MODULE 3

#include <stdio.h>  
#include <stdlib.h>  
#define BLOCKSIZE 256  
typedef int object\_t;  
typedef int key\_t;  
#define MAXP 46337 /\* prime, and 46337\*46337 < 2147483647 \*/  
typedef struct { int size;  
 int primary\_a;  
 int \*secondary\_a;  
 int \*secondary\_s;  
 int \*secondary\_o;  
 int \*keys;  
 object\_t \*objs; } perf\_hash\_t;  
perf\_hash\_t \*create\_perf\_hash(int size, int keys[], object\_t objs[])  
{ perf\_hash\_t \*tmp;   
 int \*table1, \*table2, \*table3, \*table4;  
 int i, j, k, collision, sq\_bucket\_sum, sq\_sum\_limit, a;   
 object\_t \*objects;  
 tmp = (perf\_hash\_t \*) malloc( sizeof(perf\_hash\_t) );  
 table1 = (int \*) malloc( size \* sizeof(int) );  
 table2 = (int \*) malloc( size \* sizeof(int) );  
 table3 = (int \*) malloc( size \* sizeof(int) );  
 sq\_sum\_limit = 5\*size;  
 sq\_bucket\_sum = 100\*size;  
 while(sq\_bucket\_sum > sq\_sum\_limit) /\* find primary factor \*/  
 { a = rand()%MAXP;  
 for(i=0; i<size; i++)  
 table1[i] = 0;  
 for(i=0; i<size; i++)  
 table1[ (((a\*keys[i])%MAXP)% size) ] +=1;  
 sq\_bucket\_sum = 0;  
 for(i=0; i<size; i++)  
 sq\_bucket\_sum += table1[i]\*table1[i];  
 }  
 /\* compute secondary table sizes and their offset \*/  
 for(i=0; i< size; i++ )   
 { table1[i] = 2\*table1[i]\*table1[i];   
 table2[i] = (i>0) ? table2[i-1] + table1[i-1] : 0;  
 }  
 table4 = (int \*) malloc( 2\*sq\_bucket\_sum \* sizeof(int) );  
 for( i=0; i< 2\*sq\_bucket\_sum; i++ )  
 table4[i] = MAXP; /\* different from all keys \*/  
 collision = 1;  
 for( i=0; i< size; i++ )  
 table3[i] = rand()%MAXP; /\* secondary hash factor \*/  
 while( collision )  
 { collision = 0;  
 for( i=0; i< size; i++ )  
 { j = ((keys[i]\*a)% MAXP) % size;  
 k = ((keys[i]\*table3[j])% MAXP) % table1[j] + table2[j];  
 if( table4[k] == MAXP || table4[k] == keys[i] )  
 table4[k] = keys[i]; /\* entry up to now empty \*/   
 else /\* collision \*/  
 { collision = 1;  
 table3[i] = 0; /\* mark bucket as defect \*/  
 }  
 }  
 if( collision )  
 { for( i=0; i< size; i++)  
 { if( table3[i] == 0 ) /\* defect bucket \*/  
 { table3[i] = rand()%MAXP; /\* choose new factor \*/  
 for( k= table2[i]; k< table2[i]+table1[i]; k++)  
 table4[k] = MAXP; /\* clear i-th secondary table \*/  
 }  
 }  
 }  
 } /\* now the hash table is collision-free \*/  
 /\* keys are in the right places, now put objects there \*/  
 objects =(object\_t \*)malloc(2\*sq\_bucket\_sum\*sizeof(object\_t) );  
 for( i=0; i< size; i++ )  
 { j = ((keys[i]\*a)% MAXP) % size;  
 k = ((keys[i]\*table3[j])% MAXP) % table1[j] + table2[j];  
 objects[k] = objs[i];  
 }  
 tmp->size = size;  
 tmp->primary\_a = a; /\* primary hash table factor \*/  
 tmp->secondary\_a = table3; /\* secondary hash table factors \*/  
 tmp->secondary\_s = table1; /\* secondary hash table sizes \*/  
 tmp->secondary\_o = table2; /\* secondary hash table offsets \*/  
 tmp->keys = table4; /\* secondary hash tables \*/  
 tmp->objs = objects;  
 return( tmp );  
}  
  
  
object\_t \*find(perf\_hash\_t \*ht, int query\_key)  
{ int i, j;  
 i = ((ht->primary\_a\*query\_key)% MAXP)%ht->size;  
 if( ht->secondary\_s[i] == 0 )  
 return( NULL ); /\* secondary bucket empty \*/  
 else  
 { j = ((ht->secondary\_a[i]\*query\_key)% MAXP)%ht->secondary\_s[i]  
 + ht->secondary\_o[i];  
 if( ht->keys[j] == query\_key )  
 return( (ht->objs)+j ); /\* right key found \*/  
 else  
 return( NULL ); /\* query\_key does not exist. \*/   
 }  
}  
int main()  
{   
 char nextop;  
 int keys[1000]; int objects[1000]; int size = 0; int i;  
 perf\_hash\_t \*ht;  
 printf("Enter Keys (here we choose object 10\*k for key k)\n");  
 while( (nextop = getchar())!= 'q' )  
 { if( nextop == 'i' )  
 { int inskey;  
 scanf(" %d", &inskey);  
 keys[size] = inskey;  
 objects[size] = 10\*inskey;  
 size += 1;  
 }  
 }  
 printf("\nList of keys:\n");  
 for( i =0; i < size ; i++ )  
 printf(" %d", keys[i] );  
 printf("\n");  
 ht = create\_perf\_hash( size, keys, objects );  
 printf("created perfect hash table\n");  
 while( (nextop = getchar())!= 'q' )  
 {   
 if( nextop == 'f' )  
 { int findkey, \*findobj;  
 scanf(" %d", &findkey);  
 printf(" looking for key %d\n", findkey);  
 findobj = find( ht, findkey);  
 if( findobj == NULL )  
 printf(" find failed, for key %d\n", findkey);  
 else  
 printf(" find successful, found object %d\n", \*findobj);  
 }}return(0);}

//MODULE 4

4.1:-Merging ordered list

#include<bits/stdc++.h>

using namespace std;

struct Node

{

int data;

struct Node \*next;

};

Node \*newNode(int key)

{

struct Node \*temp = new Node;

temp->data = key;

temp->next = NULL;

return temp;

}

void printList(Node \*node)

{

while (node != NULL)

{

printf("%d-> ", node->data);

node = node->next;

}

}

Node \*merge(Node \*h1, Node \*h2)

{

if (!h1)

return h2;

if (!h2)

return h1;

if (h1->data < h2->data)

{

h1->next = merge(h1->next, h2);

return h1;

}

else

{

h2->next = merge(h1, h2->next);

return h2;

}

}

int main()

{

Node \*head1 = newNode(1);

head1->next = newNode(3);

head1->next->next = newNode(5);

Node \*head2 = newNode(0);

head2->next = newNode(2);

head2->next->next = newNode(4);

Node \*mergedhead = merge(head1, head2);

printList(mergedhead);

return 0;

}

4.2:-Most frequent words

#include <stdio.h>

#include <string.h>

#include <ctype.h>

# define CHAR 26

# define WORD 30

struct Node

{

bool isEnd;

unsigned frequency;

int indexMinHeap;

Node\* child[CHAR];

};

struct MinHeapNode

{

Node\* root;

unsigned frequency;

char\* word;

};

struct MinHeap

{

unsigned capacity;

int count;

MinHeapNode\* array;

};

Node\* newNode()

{

Node\* node = new Node;

node->isEnd = 0;

node->frequency = 0;

node->indexMinHeap = -1;

for( int i = 0; i < CHAR; ++i )

node->child[i] = NULL;

return node;

}

MinHeap\* createMinHeap( int capacity )

{

MinHeap\* minHeap = new MinHeap;

minHeap->capacity = capacity;

minHeap->count = 0;

minHeap->array = new MinHeapNode [ minHeap->capacity ];

return minHeap;

}

void swapMinHeapNodes ( MinHeapNode\* a, MinHeapNode\* b )

{

MinHeapNode temp = \*a;

\*a = \*b;

\*b = temp;

}

void minHeapify( MinHeap\* minHeap, int idx )

{ int left, right, smallest;

left = 2 \* idx + 1;

right = 2 \* idx + 2;

smallest = idx;

if ( left < minHeap->count &&

minHeap->array[ left ]. frequency <

minHeap->array[ smallest ]. frequency

)

smallest = left;

if ( right < minHeap->count &&

minHeap->array[ right ]. frequency <

minHeap->array[ smallest ]. frequency

)

smallest = right;

if( smallest != idx )

{

minHeap->array[ smallest ]. root->indexMinHeap = idx;

minHeap->array[ idx ]. root->indexMinHeap = smallest;

swapMinHeapNodes (&minHeap->array[ smallest ], &minHeap->array[ idx ]);

minHeapify( minHeap, smallest );

}

}

void buildMinHeap( MinHeap\* minHeap )

{

int n, i;

n = minHeap->count - 1;

for( i = ( n - 1 ) / 2; i >= 0; --i )

minHeapify( minHeap, i );

}

void insertInMinHeap( MinHeap\* minHeap, Node\*\* root, const char\* word )

{

if( (\*root)->indexMinHeap != -1 )

{

++( minHeap->array[ (\*root)->indexMinHeap ]. frequency );

minHeapify( minHeap, (\*root)->indexMinHeap );

}

else if( minHeap->count < minHeap->capacity )

{

int count = minHeap->count;

minHeap->array[ count ]. frequency = (\*root)->frequency;

minHeap->array[ count ]. word = new char [strlen( word ) + 1];

strcpy( minHeap->array[ count ]. word, word );

minHeap->array[ count ]. root = \*root;

(\*root)->indexMinHeap = minHeap->count;

++( minHeap->count );

buildMinHeap( minHeap );

}

else if ( (\*root)->frequency > minHeap->array[0]. frequency )

{

minHeap->array[ 0 ]. root->indexMinHeap = -1;

minHeap->array[ 0 ]. root = \*root;

minHeap->array[ 0 ]. root->indexMinHeap = 0;

minHeap->array[ 0 ]. frequency = (\*root)->frequency;

delete [] minHeap->array[ 0 ]. word;

minHeap->array[ 0 ]. word = new char [strlen( word ) + 1];

strcpy( minHeap->array[ 0 ]. word, word );

minHeapify ( minHeap, 0 );

}

}

void insertUtil ( Node\*\* root, MinHeap\* minHeap,

const char\* word, const char\* dupWord )

{

if ( \*root == NULL )

\*root = newNode();

if ( \*word != '\0' )

insertUtil ( &((\*root)->child[ tolower( \*word ) - 97 ]),

minHeap, word + 1, dupWord );

else

{

if ( (\*root)->isEnd )

++( (\*root)->frequency );

else

{

(\*root)->isEnd = 1;

(\*root)->frequency = 1;

}

insertInMinHeap( minHeap, root, dupWord );

}

}

void insertTrieAndHeap(const char \*word, Node\*\* root, MinHeap\* minHeap)

{

insertUtil( root, minHeap, word, word );

}

void displayMinHeap( MinHeap\* minHeap )

{

int i;

for( i = 0; i < minHeap->count; ++i )

{

printf( "%-4s : %d\n", minHeap->array[i].word,

minHeap->array[i].frequency );

}

}

void printKMostFreq( FILE\* fp, int k )

{

MinHeap\* minHeap = createMinHeap( k );

Node\* root = NULL;

char buffer[WORD];

while( fscanf( fp, "%s", buffer ) != EOF )

insertTrieAndHeap(buffer, &root, minHeap);

displayMinHeap( minHeap );

}

int main()

{

int k ;

printf("enter number of words to be searched\n");

scanf("%d",&k);

FILE \*fp = fopen ("file.txt", "r");

if (fp == NULL)

printf ("File doesn't exist ");

else

printKMostFreq (fp, k);

return 0;

}

4.3:-Parallel arrays

#include <iostream>

using namespace std;

int partition(string first\_name[], string last\_name[],

int height[], int low, int high)

{

int pivot = height[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (height[j] <= pivot) {

i++;

string temp = first\_name[i];

first\_name[i] = first\_name[j];

first\_name[j] = temp;

temp = last\_name[i];

last\_name[i] = last\_name[j];

last\_name[j] = temp;

int temp1 = height[i];

height[i] = height[j];

height[j] = temp1;

}

}

string temp = first\_name[i + 1];

first\_name[i + 1] = first\_name[high];

first\_name[high] = temp;

temp = last\_name[i + 1];

last\_name[i + 1] = last\_name[high];

last\_name[high] = temp;

int temp1 = height[i + 1];

height[i + 1] = height[high];

height[high] = temp1;

return (i + 1);

}

void quickSort(string first\_name[], string last\_name[],

int height[], int low, int high)

{

if (low < high) {

int pi = partition(first\_name, last\_name,

height, low, high);

quickSort(first\_name, last\_name, height,

low, pi - 1);

quickSort(first\_name, last\_name, height,

pi + 1, high);

}

}

void binarySearch(string first\_name[], string

last\_name[],

int height[], int value, int n)

{

int low = 0, high = n - 1;

int index;

while (low <= high) {

index = (high + low) / 2;

if (height[index] == 158) {

cout << "Person having height 158"

" cms is "

<< first\_name[index]

<< " " << last\_name[index] << endl;

return;

}

else if (height[index] > 158)

high = index - 1;

else

low = index + 1;

}

cout << "Sorry, no such person with"

" height 158 cms";

cout << "is found in the record";

}

void printParallelArray(string first\_name[],

string last\_name[], int height[], int n)

{

cout << "Name of people in increasing";

cout << "order of their height: " << endl;

for (int i = 0; i < n; i++) {

cout << first\_name[i] << " "

<< last\_name[i] << " has height "

<< height[i] << " cms\n";

}

cout << endl;

}

int main()

{

int n = 10;

string first\_name[] = { "Bones", "Welma",

"Frank", "Han", "Jack", "Jinny", "Harry",

"Emma", "Tony", "Sherlock" };

string last\_name[] = { "Smith", "Seger",

"Mathers", "Solo", "Jackles", "Weasly",

"Potter", "Watson", "Stark", "Holmes" };

int height[] = { 169, 158, 201, 183, 172,

152, 160, 163, 173, 185 };

quickSort(first\_name, last\_name, height,

0, n - 1);

printParallelArray(first\_name, last\_name,

height, n);

cout << "Name of the second tallest person"

" is "

<< first\_name[n - 2] << " "

<< last\_name[n - 2] << endl;

cout << "Name of the third shortest person is "

<< first\_name[2] << " " << last\_name[2]

<< endl;

binarySearch(first\_name, last\_name, height,

158, n);

return 0;

}

4.4:-Threaded binary tree

// Complete C++ program to demonstrate deletion

// in threaded BST

#include<bits/stdc++.h>

using namespace std;

struct Node

{

struct Node \*left, \*right;

int info;

// True if left pointer points to predecessor

// in Inorder Traversal

bool lthread;

// True if right pointer points to predecessor

// in Inorder Traversal

bool rthread;

};

// Insert a Node in Binary Threaded Tree

struct Node \*insert(struct Node \*root, int ikey)

{

// Searching for a Node with given value

Node \*ptr = root;

Node \*par = NULL; // Parent of key to be inserted

while (ptr != NULL)

{

// If key already exists, return

if (ikey == (ptr->info))

{

printf("Duplicate Key !\n");

return root;

}

par = ptr; // Update parent pointer

// Moving on left subtree.

if (ikey < ptr->info)

{

if (ptr -> lthread == false)

ptr = ptr -> left;

else

break;

}

// Moving on right subtree.

else

{

if (ptr->rthread == false)

ptr = ptr -> right;

else

break;

}

}

// Create a new Node

Node \*tmp = new Node;

tmp -> info = ikey;

tmp -> lthread = true;

tmp -> rthread = true;

if (par == NULL)

{

root = tmp;

tmp -> left = NULL;

tmp -> right = NULL;

}

else if (ikey < (par -> info))

{

tmp -> left = par -> left;

tmp -> right = par;

par -> lthread = false;

par -> left = tmp;

}

else

{

tmp -> left = par;

tmp -> right = par -> right;

par -> rthread = false;

par -> right = tmp;

}

return root;

}

// Returns inorder successor using left

// and right children (Used in deletion)

struct Node \*inSucc(struct Node \*ptr)

{

if (ptr->rthread == true)

return ptr->right;

ptr = ptr -> right;

while (ptr->right)

ptr = ptr->left;

return ptr;

}

// Returns inorder successor using rthread

// (Used in inorder)

struct Node \*inorderSuccessor(struct Node \*ptr)

{

// If rthread is set, we can quickly find

if (ptr -> rthread == true)

return ptr->right;

// Else return leftmost child of right subtree

ptr = ptr -> right;

while (ptr -> lthread == false)

ptr = ptr -> left;

return ptr;

}

// Printing the threaded tree

void inorder(struct Node \*root)

{

if (root == NULL)

printf("Tree is empty");

// Reach leftmost Node

struct Node \*ptr = root;

while (ptr -> lthread == false)

ptr = ptr -> left;

// One by one print successors

while (ptr != NULL)

{

printf("%d ",ptr -> info);

ptr = inorderSuccessor(ptr);

}

}

struct Node \*inPred(struct Node \*ptr)

{

if (ptr->lthread == true)

return ptr->right;

ptr = ptr->left;

while (ptr->rthread);

ptr = ptr->right;

return ptr;

}

// Here 'par' is pointer to parent Node and 'ptr' is

// pointer to current Node.

struct Node \*caseA(struct Node \*root, struct Node \*par,

struct Node \*ptr)

{

// If Node to be deleted is root

if (par == NULL)

root = NULL;

// If Node to be deleted is left

// of its parent

else if (ptr == par->left)

{

par->lthread = true;

par->left = ptr->left;

}

else

{

par->rthread = true;

par->right = ptr->right;

}

// Free memory and return new root

free(ptr);

return root;

}

// Here 'par' is pointer to parent Node and 'ptr' is

// pointer to current Node.

struct Node \*caseB(struct Node \*root, struct Node \*par,

struct Node \*ptr)

{

struct Node \*child;

// Initialize child Node to be deleted has

// left child.

if (ptr->lthread == false)

child = ptr->left;

// Node to be deleted has right child.

else

child = ptr->right;

// Node to be deleted is root Node.

if (par == NULL)

root = child;

// Node is left child of its parent.

else if (ptr == par->left)

par->left = child;

else

par->right = child;

// Find successor and predecessor

Node \*s = inSucc(ptr);

Node \*p = inPred(ptr);

// If ptr has left subtree.

if (ptr->lthread == false)

p->right = s;

// If ptr has right subtree.

else

{

if (ptr->rthread == false)

s->left = p;

}

free(ptr);

return root;

}

// Here 'par' is pointer to parent Node and 'ptr' is

// pointer to current Node.

struct Node \*caseC(struct Node \*root, struct Node \*par,

struct Node \*ptr)

{

// Find inorder successor and its parent.

struct Node \*parsucc = ptr;

struct Node \*succ = ptr -> right;

// Find leftmost child of successor

while (succ->left != NULL)

{

parsucc = succ;

succ = succ -> left;

}

ptr->info = succ->info;

if (succ->lthread == true && succ->rthread == true)

root = caseA(root, parsucc, succ);

else

root = caseB(root, parsucc, succ);

return root;

}

// Deletes a key from threaded BST with given root and

// returns new root of BST.

struct Node \*delThreadedBST(struct Node\* root, int dkey)

{

// Initialize parent as NULL and ptrent

// Node as root.

struct Node \*par = NULL, \*ptr = root;

// Set true if key is found

int found = 0;

// Search key in BST : find Node and its

// parent.

while (ptr != NULL)

{

if (dkey == ptr->info)

{

found = 1;

break;

}

par = ptr;

if (dkey < ptr->info)

{

if (ptr->lthread == false)

ptr = ptr -> left;

else

break;

}

else

{

if (ptr->rthread == false)

ptr = ptr->right;

else

break;

}

}

if (found == 0)

printf("dkey not present in tree\n");

// Two Children

else if (ptr->lthread == false && ptr->rthread == false)

root = caseC(root, par, ptr);

// Only Left Child

else if (ptr->lthread == false)

root = caseB(root, par, ptr);

// Only Right Child

else if (ptr->rthread == false)

root = caseB(root, par, ptr);

// No child

else

root = caseA(root, par, ptr);

return root;

}

// Driver Program

int main()

{

struct Node \*root = NULL;

root = insert(root, 20);

root = insert(root, 10);

root = insert(root, 30);

root = insert(root, 5);

root = insert(root, 16);

root = insert(root, 14);

root = insert(root, 17);

root = insert(root, 13);

root = delThreadedBST(root, 20);

inorder(root);

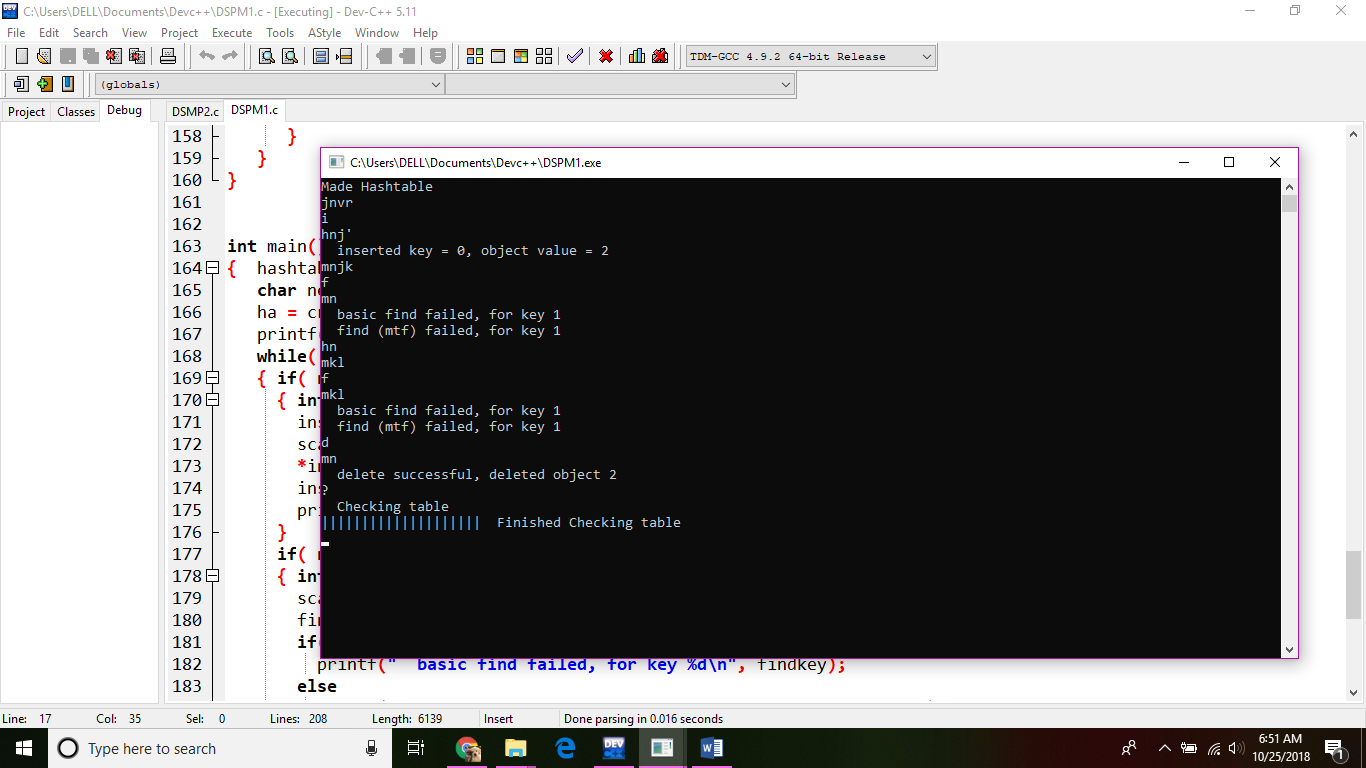
return 0;

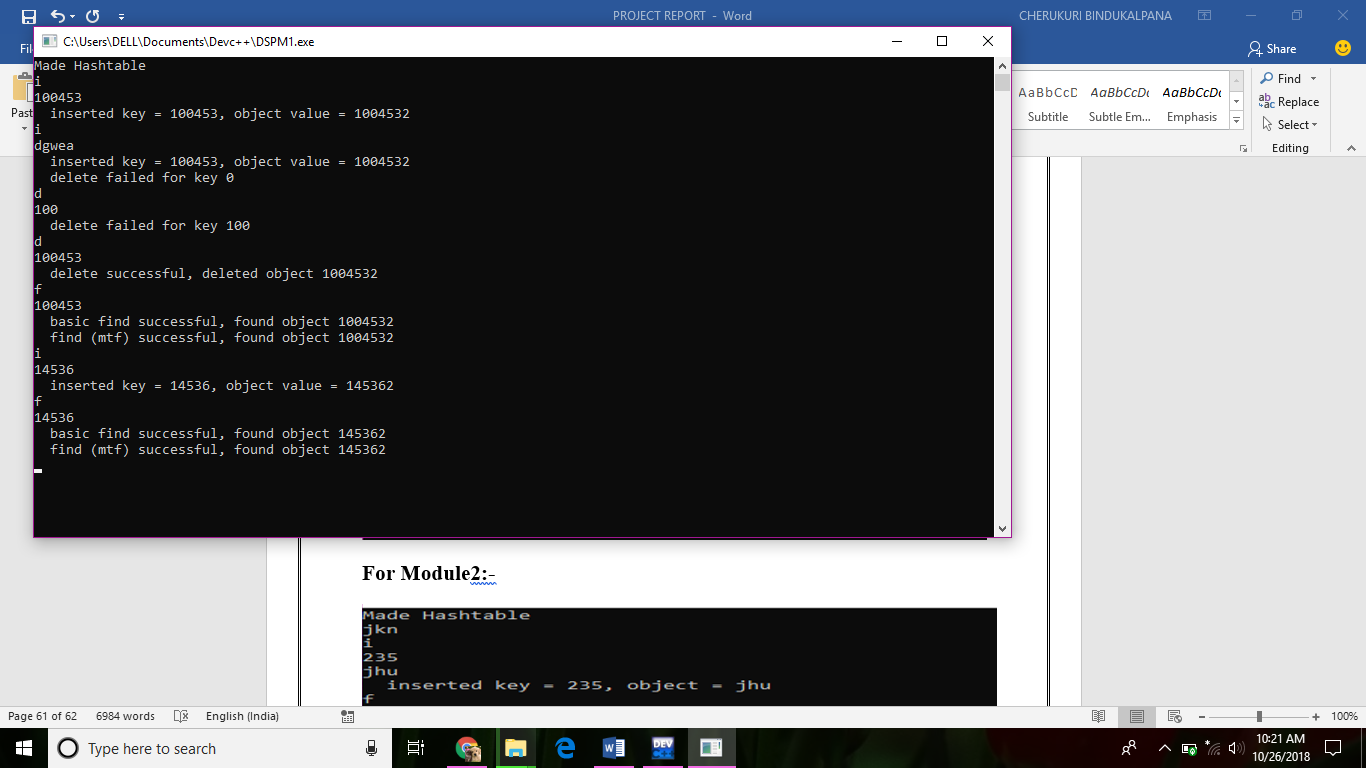
}

**9. OUTPUT**

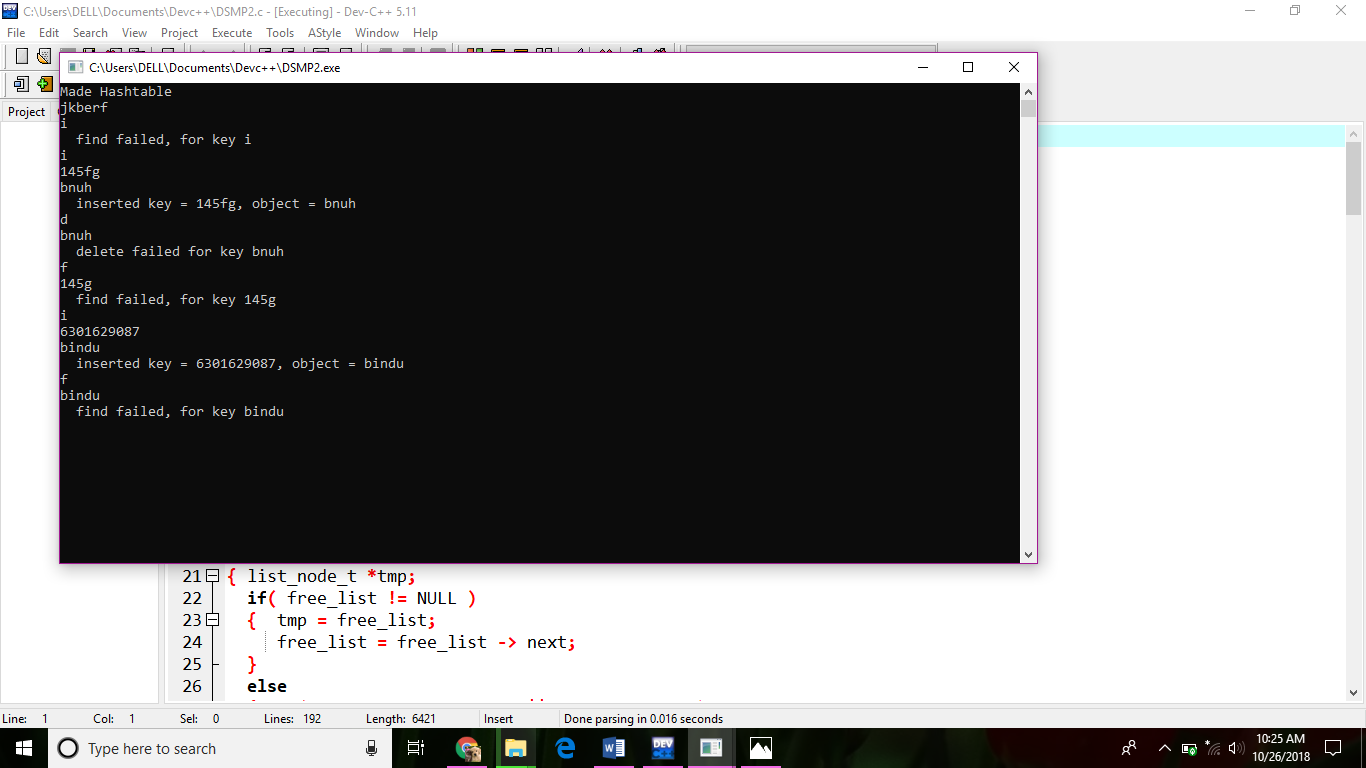
**SCREEN SHOTS:**

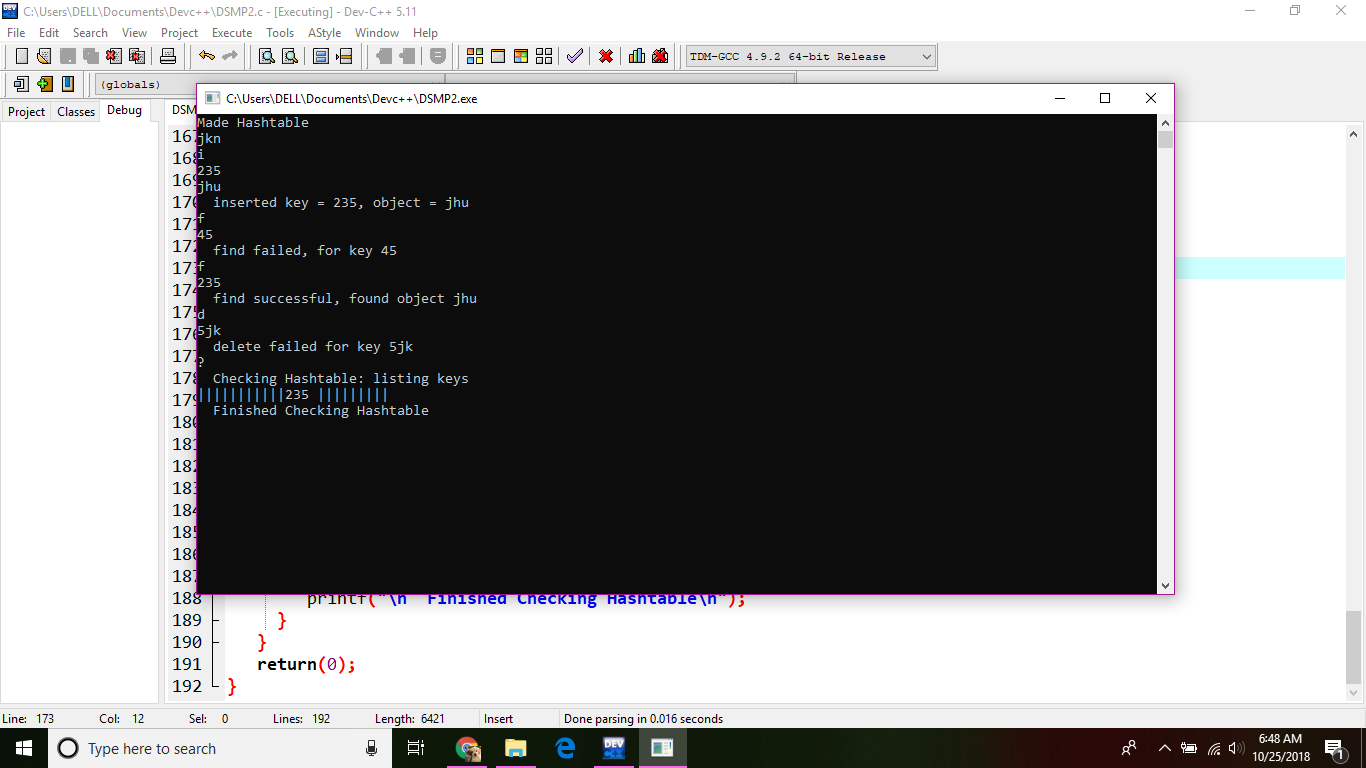
**Module 1: -**



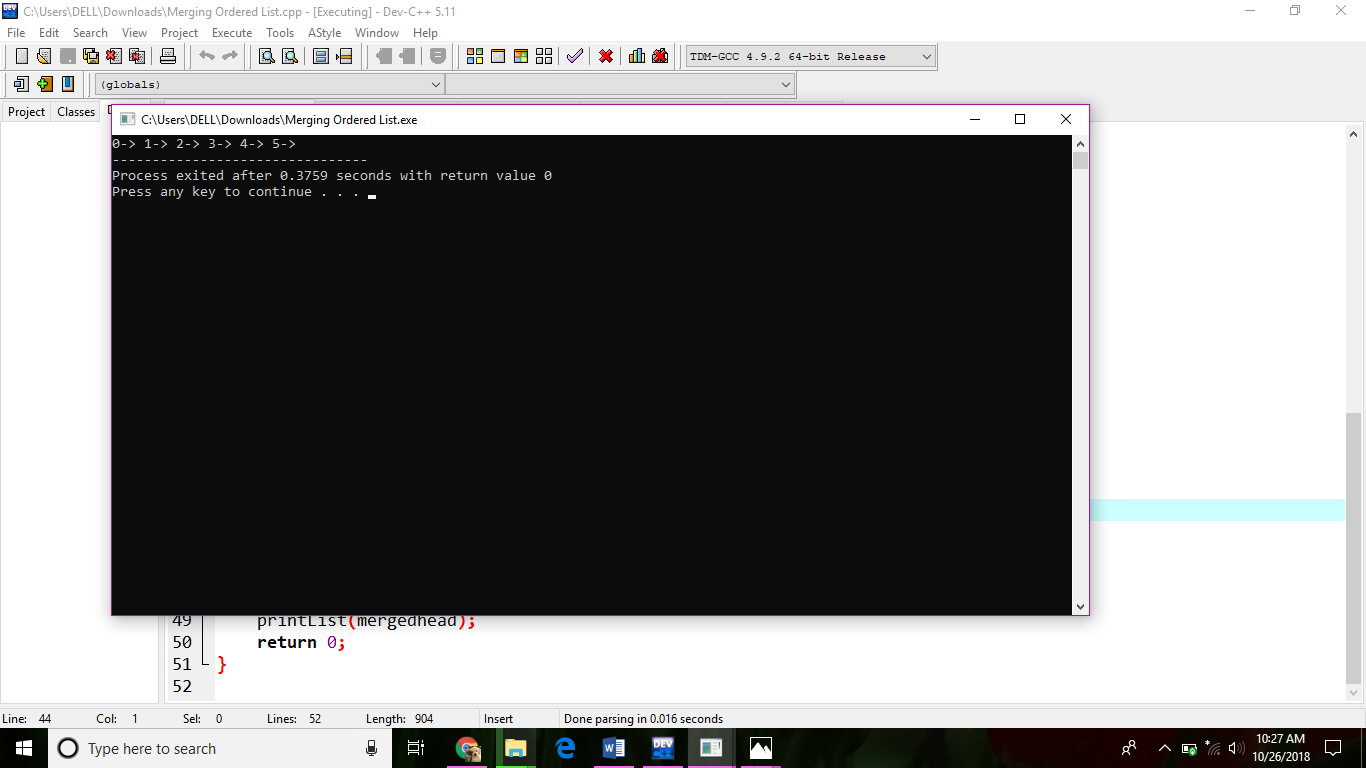


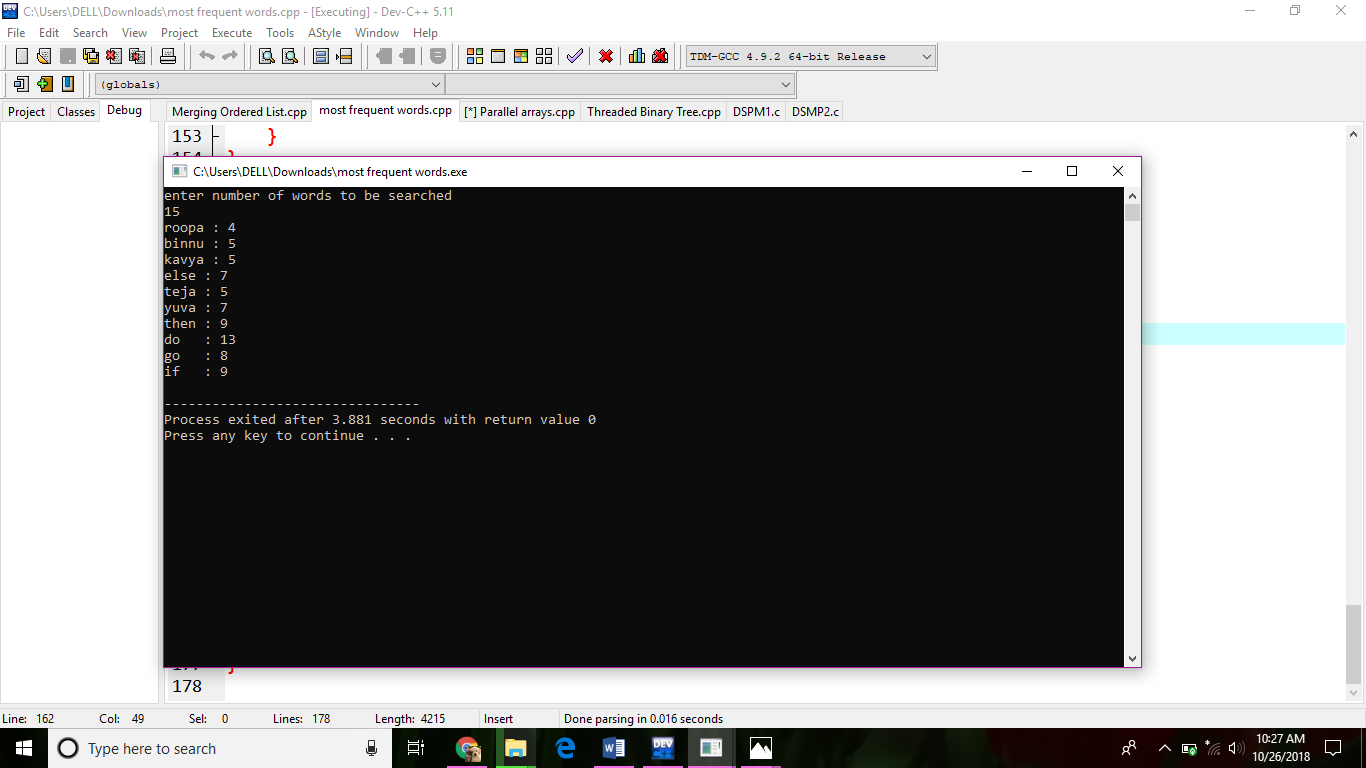
**Module 2: -**

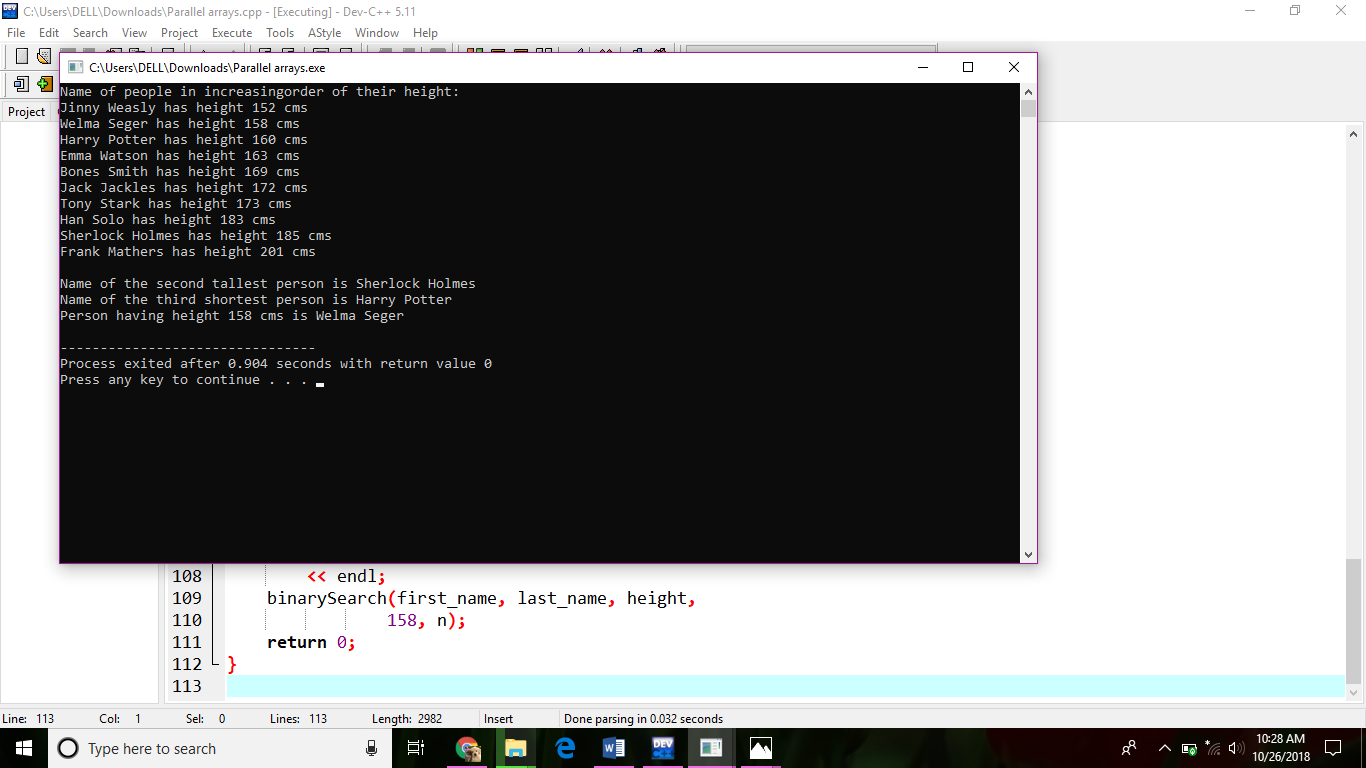




**Module 4:-**







**10. Conclusion**

By doing this project we understand about the hashing. when we want to use the hashing.

Why we have to use that. all these thigs we understood very much. And also, we understand the importance of hashing technique in data structures not only in data structures. If you want solve any larger problem easily and occupying less memory, we can use our hashing techniques anywhere.

**References**

1. Carter, Larry; [Wegman, Mark N.](https://en.wikipedia.org/wiki/Mark_N._Wegman) (1979). "Universal Classes of Hash Functions". Journal of Computer and System Sciences. 18 (2): 143–154. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):[10.1016/0022-0000(79)90044-8](https://doi.org/10.1016%2F0022-0000%2879%2990044-8). Conference version in STOC'77.
2. [Jump up^](https://en.wikipedia.org/wiki/Universal_hashing#cite_ref-Miltersen_2-0) Miltersen, Peter Bro. ["Universal Hashing"](http://www.daimi.au.dk/~bromille/Notes/un.pdf) (PDF). [Archived](https://www.webcitation.org/5hmOaVISI?url=http://www.daimi.au.dk/~bromille/Notes/un.pdf) (PDF) from the original on 24 June 2009. Retrieved 24 June 2009.
3. [Jump up^](https://en.wikipedia.org/wiki/Universal_hashing#cite_ref-3) Motwani, Rajeev; Raghavan, Prabhakar (1995). Randomized Algorithms. Cambridge University Press. p. 221. [ISBN](https://en.wikipedia.org/wiki/International_Standard_Book_Number) [0-521-47465-5](https://en.wikipedia.org/wiki/Special:BookSources/0-521-47465-5)
4. https://en.wikipedia.org/wiki/Universal\_hashing#References
5. http://shodhganga.inflibnet.ac.in/bitstream/10603/168210/16/16.%20chapter%20-%20vii.pdf
6. https://www.google.co.in/search?q=advantages+and+disadvantages+of+hashing+in+data+structure&oq=advantages+and+disadvantages+of+hashing+in+ds&aqs=chrome.1.69i57j0.26243j1j1&sourceid=chrome&ie=UTF-8