**Characteristics of a Good Hash Function**

There are four main characteristics of a good hash function: 1) The hash value is fully determined by the data being hashed. 2) The hash function uses all the input data. 3) The hash function "uniformly" distributes the data across the entire set of possible hash values. 4) The hash function generates very different hash values for similar strings

Let's examine why each of these is important: Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values. Rule 2: If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions. Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table. Rule 4: In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

# **Double Hashing**

**Double hashing** is a collision resolving technique in [**Open Addressed** Hash tables](https://www.geeksforgeeks.org/hashing-set-3-open-addressing/). Double hashing uses the idea of applying a second hash function to key when a collision occurs.

*Double hashing can be done using:****(hash1(key) + i \* hash2(key)) % TABLE\_SIZE*** *Here hash1() and hash2() are hash functions and TABLE\_SIZE  
is size of hash table.  
(We repeat by increasing i when collision occurs)*

First hash function is typically hash1(key) = key % TABLE\_SIZE

A popular second hash function is: **hash2(key) = PRIME – (key % PRIME)** where PRIME is a prime smaller than the TABLE\_SIZE.

A good second Hash function is:

* It must never evaluate to zero
* Must make sure that all cells can be probed

Linear probing for strings :

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <stdint.h>

#include <stdbool.h>

#define max\_name 256

#define tablesize 10

typedef struct

{

char name [max\_name];

}person;

//making a hash table its an array of pointers to people struct , pointers because for pointers i don't need space for tables means i directly fill the table

// 2nd reason to use the pointers that it makes easy for me to tell when a spot in the table is empty because i can just set the pointer to null

//making a function to setup the table

person \* hash\_table[tablesize];

void init\_hashtable()

{

//initially i want all the entries to be empty so :

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

{

hash\_table [i] = NULL;

}

//table is empty

}

// i am declaring hash func which takes a input name and map it to a location in a table which is an unsigned int:

unsigned int hash (char \*name )

{

int length = strlen(name ); //it woudl calculate the length of the string

//now i would go through the string at one character at a time and add the ASCII values of char

unsigned int hash\_value = 0;

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

{

hash\_value += name [i]; //we would add each char ascii value to the sum

hash\_value = ( hash\_value \* name[i] ) ; //for getting more random number

hash\_value = hash\_value % tablesize ; //keeping the num bewtween 0 and table size

}

return hash\_value ; //we can return the sum when we are done

}

//making a function that would insert something in the table

//this func would return a boolean true if we successfully inserted otherwise false

bool hash\_table\_insert(person \*p)

{

if( p == NULL) //if it is null value

{

return false ;

}

int index = hash(p->name); //index or location where we are going to try the index of a person .

//making linear probing :

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

int key = (i+index) % tablesize ;

if(hash\_table[key] == NULL)

{

hash\_table[key] = p;

return true;

}

} return false ;

}

void print\_table ()

{

printf("Start \n");

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

{

if(hash\_table[i] == NULL)

{

printf(" %i \t-----\n" ,i);

}

else

{

printf(" %i\t%s\n",i,hash\_table[i]->name);

}

}

printf("End \n \n");

}

//the main concept of the table is too lookup people

//so now lets make a lookup function

//it find a person on the table by their name

person \*hash\_table\_lookup (char \*name)

{

int index = hash(name);

//adding linear logic in the lookup

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

{

int key = (index +i) %tablesize;

if(hash\_table[key] != NULL && strncmp(hash\_table[key]->name , name , 50)== 0)

{ //strncmp() func is used to compare two strings with each other to limited no of char passed

return hash\_table[key];

}

}

return NULL ;

}

int main(){

init\_hashtable();

print\_table();

//lets make some people

person jacob = {.name = "jacob"};

person kate = {.name = "kate"};

person mpho = {.name = "mpho"};

person Sarah = {.name = "Sarah"};

person Edna = {.name = "Edna"};

person Eliza = {.name = "Eliza"};

person Robert = {.name = "Robert"};

person Jane = {.name = "Jane"};

//inserting these people in the hashtable

hash\_table\_insert(&jacob);

hash\_table\_insert(&kate);

hash\_table\_insert(&mpho);

hash\_table\_insert(&Sarah);

hash\_table\_insert(&Edna);

hash\_table\_insert(&Eliza);

hash\_table\_insert(&Robert);

hash\_table\_insert(&Jane);

//print

print\_table();

//looking up a name in the table :

person \*tmp = hash\_table\_lookup("mpho");

if(tmp == NULL )

{

printf("we don't found it \n ");

}

else

{

printf("Found %s \n",tmp ->name);

}

//lookin up a value which is not present in the table

tmp = hash\_table\_lookup("george");

if(tmp == NULL )

{

printf("we don't found it \n");

}

else

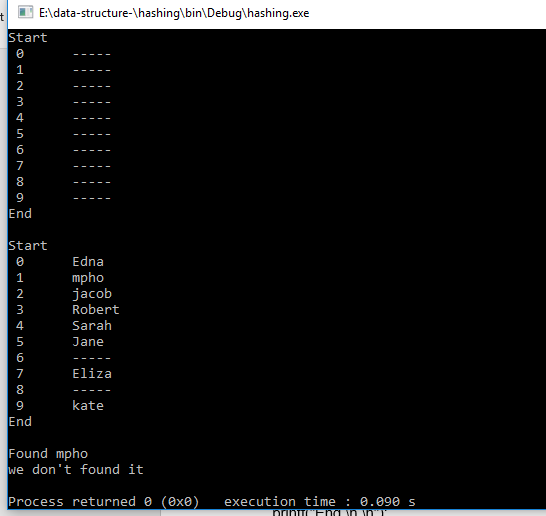
{

printf("Found %s \n",tmp ->name);

}

return 0 ;

}



Linear probing for strings :

#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

float arr[3];

int count=0;

float ftvalue;

//MAIN METHDOD :

int main(int argc, char \*argv[])

{

for(int i = 0 ; i<3 ; i++)

{

printf("enter the value u want to insert : ");

scanf("%f",&ftvalue);

int val = ftvalue;

printf("\n");

int Key=val%3;

if(count==3){

printf("array is full.\n");

}else if(arr[Key]==0){

arr[Key]= ftvalue;

count++;

printf("fgdfgdfgd");

printf("value inserted at : %d no values inserted in the array = %d \n \n", Key,count );

}else{

int increment=Key+1;

int Newkey;

if(Key==3-1){

increment=0;

}

for(Newkey=increment;Newkey!=Key;Newkey++){

if(arr[Newkey]==0){

arr[Newkey]=ftvalue;

count++;

printf("value inserted at : %d no values inserted in the array = %d \n \n", Newkey,count );

break;

}

}

}

}

int i;

for(i=0;i<3;i++)

{

printf("arr[%d]: %f \n \n",i,arr[i]);

}

return 0;

}

