**HASHING**

**EXPT NO: 10**  **DATE: 3/1/22**

**AIM**

**A)** WAP to implement closed hashing with linear probing.  
   I) Insert II) Delete III) Search IV) Display table

**B)** WAP to implement closed hashing with quadratic probing.  
    I) Insert II) Delete III) Search IV) Display table

**C)** WAP to implement closed hashing with double hashing.  
   I) Insert II) Delete III) Search IV) Display table  
  
**D)** WAP to implement separate chaining  
   I) Insert II) Delete III) Search IV) Display table

**THEORY**

Hashing is a procedure in which the key is converted into and integer value within a range, and this converted value can be used as an index of an array. The process of converting a key into an address is known as Hashing or key to address transformation.

A hashing function is used to generate an address from a key. The key may be of any type like integer or string but the resulting hash value will always be an integer.



We represent this as

**h(k)=a**

**h: -** Hash Function

**k: -** Key

**a: -** has value of k

Now the key can be stored at array index a, which is known as the home address. The key and the in which the insertion and searching is done through hashing is called hash table.

**Hash Function**

If the size of hash table is m, then we need a hash function that can generate addresses between 0 to n-1.

Two criteria to create a good has function

1) It should be easy to computer

2) It should generate addresses with minimum collision i.e., it should distribute the keys as uniformly as possible in the hash table.

**Collison Resolution**

An ideal hash function should perform one to one mapping between set of all possible keys and all hash table addresses but this is almost impossible and no hash function can totally prevent collisions. A collision occurs whenever a key is mapped to an address that is already occupied, and the different resolution suggest for an alternate place where this key can be placed.

The two collision resolution techniques are: -

**1)** Open Addressing (Closed Hashing)

**2)** Separate chaining (Open Hashing)

In the process of searching, the given key is compared with many keys and each key comparison is known as a probe. The efficiency of a collision technique is defined in terms of the number of probes required to find a record within a given key.

**Open Addressing (Closed Hashing)**

In open addressing, initially a key value is mapped to a particular address in the hash table. If that address id occupied then we will try to insert the key in some other empty location inside the table. The array is assumed to be closed and hence this method is named as closed hashing.

The three methods are: -

1) Linear Probing

2) Quadratic Probing

3) Double Hashing

**Linear Probing**

If the address given by the hash function is already occupied, then the key will be inserted in the next empty location in the hash table. If the address given by the hash function is ‘a’ and it is not empty, then we will try to insert the key in the next location. i.e., address a+1. If address a+1 is also occupied then then wee will try to insert in the next location (a+2) and we will keep on trying successive locations till we find an empty location where the can be inserted.

The Formula for Linear Probing is: -

**H(k,i)=(h(k)+i) mod Tsize (**Here i varies from 0 to Tsize-1)

Consider the Following Example

Insert Keys: 4, 15, 17, 32, 43, 21

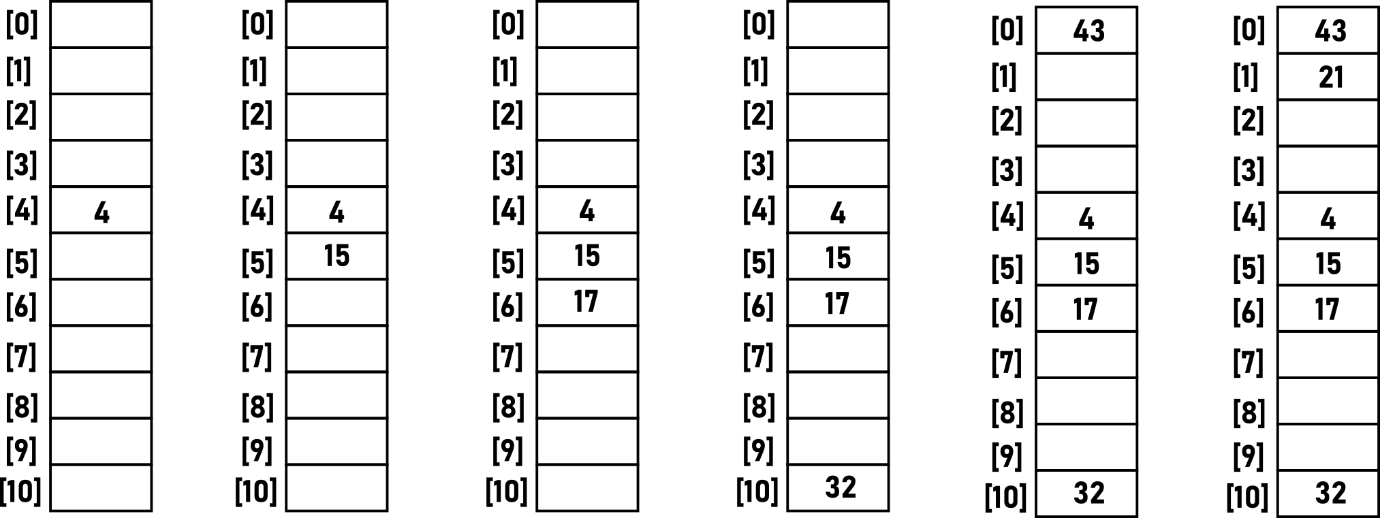
h(4)=4%11=4

h(15)=15%11=4 \*Collision 1

h(17)=17%11=6

h(32)=32%11=10

h(43)=43%11=10 \*Collision 2

h(21)=21%11=10 \*Collision 3

h(15)=15%11=4 \*Collision 1

H(15)=(h(15)+1)%11=5

h(43)=43%11=10 \*Collision 2

H(43)=(h(43)+1)%11=0

h(21)=21%11=10 \*Collision 3

H(21)=(h(21)+1)%11=0

H(21)=(h(21)+2)%11=1

**Quadratic Probing**

In Linear Probing, the colliding keys are stored near the initial collision point, resulting in the formation of clusters. In quadratic this problem is solved by storing the colliding keys away from the initial collision point.

The Formula for Quadratic Probing is: -

**H(k,i)=(h(k)+i2) mod Tsize (**Here i varies from 0 to Tsize-1)

Consider the Following Example

Insert Keys: 4, 15, 17, 32, 43, 21

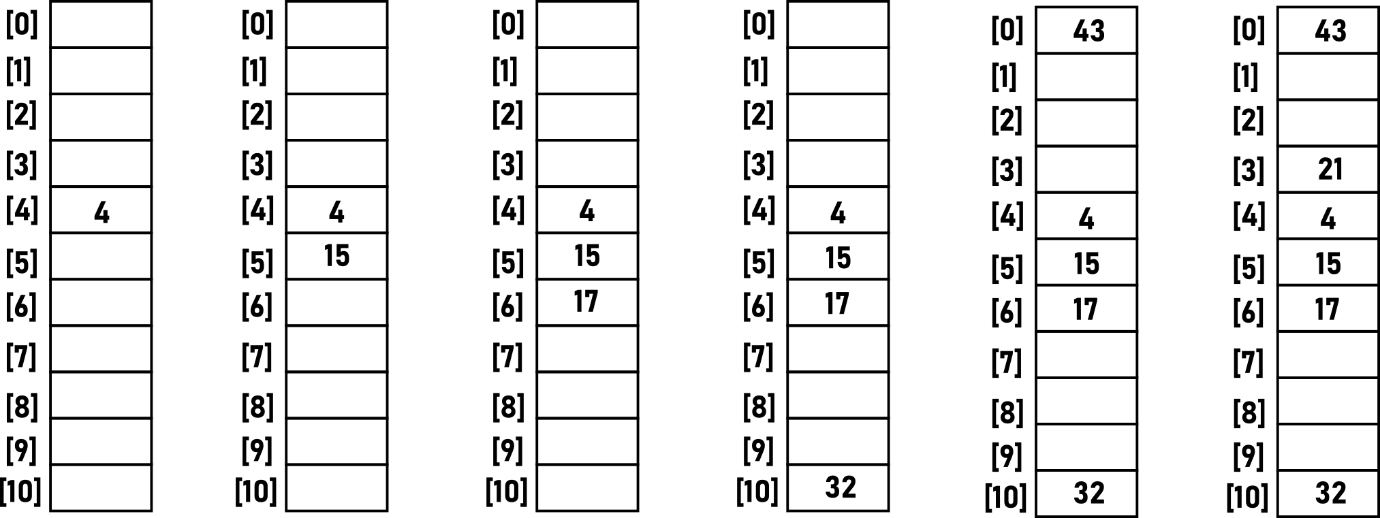
h(4)=4%11=4

h(15)=15%11=4 \*Collision 1

h(17)=17%11=6

h(32)=32%11=10

h(43)=43%11=10 \*Collision 2

h(21)=21%11=10 \*Collision 3

h(15)=15%11=4 \*Collision 1

H(15)=(h(15)+12)%11=5

h(43)=43%11=10 \*Collision 2

H(43)=(h(43)+12)%11=0

h(21)=21%11=10 \*Collision 3

H(21)=(h(21)+1)%11=0

H(21)=(h(21)+22)%11=3

**Double Hashing**

In double hashing, the increment factor is not constant as in linear or quadratic probing, but it depends on the key. The increment factor is another hash function and hence the nae double hashing.

The Formula for Double Hashing is: -

**H(k,i)=(h(k)+i h’(k)+) mod Tsize (**Here i varies from 0 to Tsize-1)

Consider the Following Example

Insert Keys: 46, 28, 21, 35, 57, 15

h(46)=46%11=2

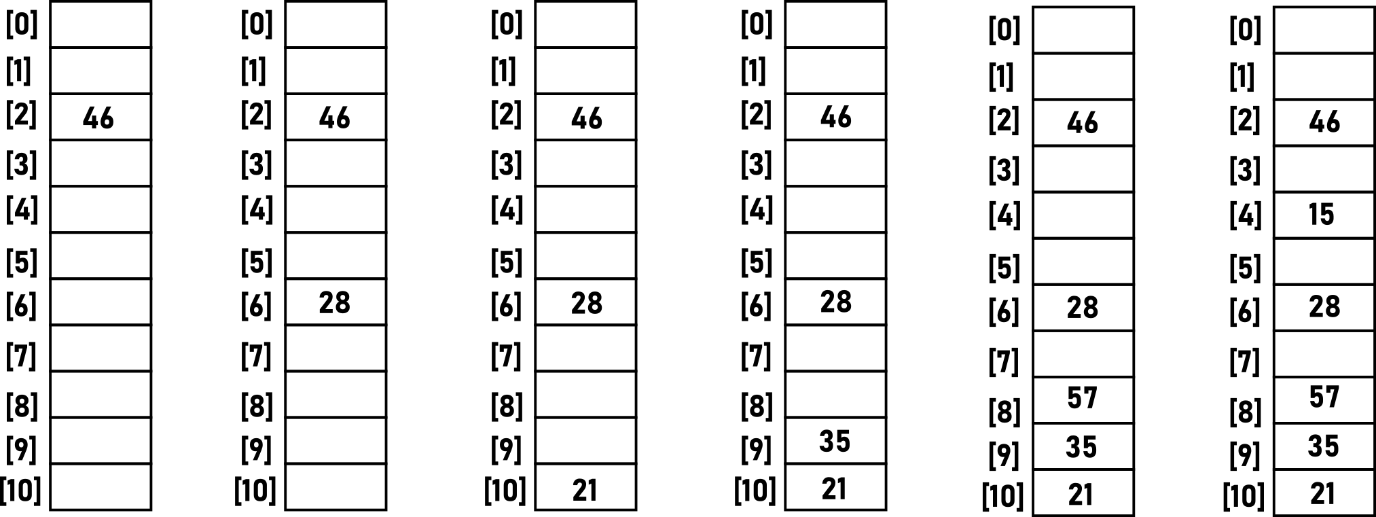
h(28)=28%11=6

h(21)=21%11=10

h(35)=35%11=2 \*Collision 1

h(57)=57%11=2 \*Collision 2

h(15)=15%11=4



h(35)=35%11= 2 \*Collision 1

H(35)=(h(35)+(7+(35%7)))= 9

h(57)=57%11= 2 \*Collision 2

H(57)=(h(57)+(7+(57%7)))= 8

**Advantages and Disadvantages**

|  |  |
| --- | --- |
| **LINEAR PROBING** | |
| **Advantage** | **Disadvantage** |
| 1.No extra space | 1. Primary Clustering |

|  |  |
| --- | --- |
| **QUADRATIC PROBING** | |
| **Advantage** | **Disadvantage** |
| 1.No extra space  2. Primary Clustering Resolved | 1. Secondary Clustering  2. No guarantee of finding space from some elements |

|  |  |
| --- | --- |
| **DOUBLE HASHING** | |
| **Advantage** | **Disadvantage** |
| 1. No extra space  2. No Primary Clustering  3. No Secondary Clustering | 1. Slower than Linear and Quadratic Probes |

**Separate Chaining**

In this method, linked lists are maintained for elements that have the same address. Here the hash table does not contain actual keys and records but it just an array of pointers, where each pointer points to a linked list. All the elements having the same address i will be stored in a separate linked list, and the starting address of that linked list will be stored in the index i of the hash table. So, array index i of the hash table contains a pointer to the list of all elements that share the has address i. theses linked lists are referred as chains and hence the method is named as separate chaining.

If the linked lists are short, performance if goof but if lists becomes long then it takes time to search a given key in any list.

**Advantage:**

Implementation of insertion and deletion is simple in separate chaining.

**Disadvantage:**

It needs extra space for pointers. If there are n records and the table size is m, then wee nee space for n+m pointers. If the records are very small then this extra space can be expensive.

**CODES**

**1)**

#include<stdio.h>

#include<string.h>

#define MAX 5

#define HASH(x)(x%MAX)

#define DUPLICATE printf("DUPLICATE ELEMENT\n")

#define SPACE printf(" ")

#define LINE printf("----------------------------------")

enum RecordType{EMPTY,OCCUPIED,DELETED};

struct candidate

{

char name[20];

int idd;

};

struct record

{

struct candidate info;

enum RecordType status;

};

void insert(struct candidate x,struct record table[])

{

int location;

int i,key,h;

key=x.idd;

h=HASH(x.idd);

location=h;

for(i=1;i!=MAX;i++)

{

if((table[location].status==EMPTY)||(table[location].status==DELETED))

{

table[location].info.idd=x.idd;

strcpy(table[location].info.name,x.name);

printf("RECORD ENTERED SUCCESSFULLY\n\n");

table[location].status=OCCUPIED;

return;

}

if(table[location].info.idd==key)

{DUPLICATE;return;}

location=(h+i)%MAX;

}

printf("TABLE OVERFLOW\n");

}

void display(struct record table[])

{

int i;

printf("\n\_\_\_\_\_\_\_\_\_CANDIDATE LIST\_\_\_\_\_\_\_\_\_\_\_\n");

printf("HASH ");SPACE;

printf("ID");SPACE;

printf("NAME\n");

for(i=0;i!=MAX;i++)

{

printf("%d ",i+1);

if(table[i].status==OCCUPIED)

{

printf("%14.d",table[i].info.idd);SPACE;

printf(" %s",table[i].info.name);

printf("\n");

}

else if(table[i].status==DELETED)

{

SPACE;SPACE;

printf(" DELTED\n");

}

else

{

SPACE;SPACE;

printf(" EMPTY\n");

}

}

LINE;

}

int search(struct record table[],int key)

{

int i;

int h,location;

h=HASH(key);

location=h;

for(i=1;i!=MAX;i++)

{

if(table[location].info.idd==key)

return location;

else if(table[location].status==EMPTY)

return -1;

location=(h+i)%MAX;

}

return -1;

}

void del(struct record table[])

{

int i,key;

int location;

printf("ENTER CANDIDATE ID NUMBER TO BE DELETED: ");

scanf("%d",&key);

location=search(table,key);

if(location==-1)

printf("CANDIDATE NAME NOT PRSENT IN THE LIST\n");

else

{

table[location].status=DELETED;

printf("RECORD DELETED\n");

}

}

int main()

{

int i,c,l,z;

struct record table[MAX];

struct candidate x;

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

table[i].status=EMPTY;

while(1)

{

printf("\n\_\_\_\_ELECTION CANDIDATE LIST\_\_\_\_\_\_\n");

printf("\n1: INSERT\n");

printf("2: DISPLAY:\n");

printf("3: SEARCH\n");

printf("4: DELETE\n");LINE;

printf("\nENTER CHOICE: ");

scanf("%d",&c);

printf("\n");

switch(c)

{

case 1:

printf("ENTER CANDIDATE NAME: ");

scanf("%s",&x.name);

printf("ENTER CANDIDATE ID: ");

scanf("%d",&x.idd);

insert(x,table);

break;

case 2:

display(table);

break;

case 3:

printf("ENTER ID: ");

scanf("%d",&l);

z=search(table,l);

if(z==-1)

printf("CANDIDATE DEETS NOT PRESENT: ");

else

{

printf("NAME: %s\tID: %d\n",table[z].info.name,table[z].info.idd);

}

break;

case 4:

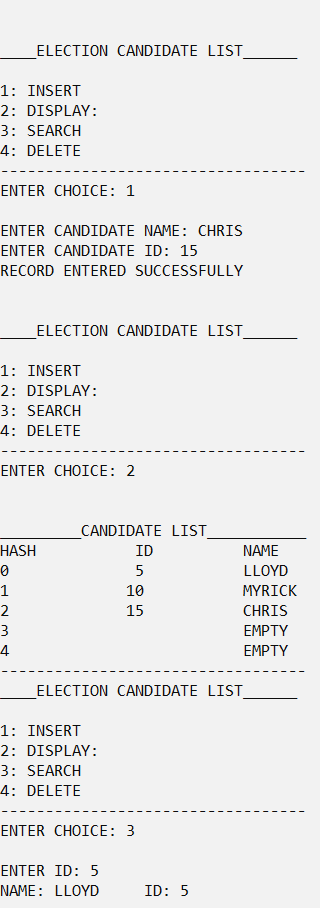
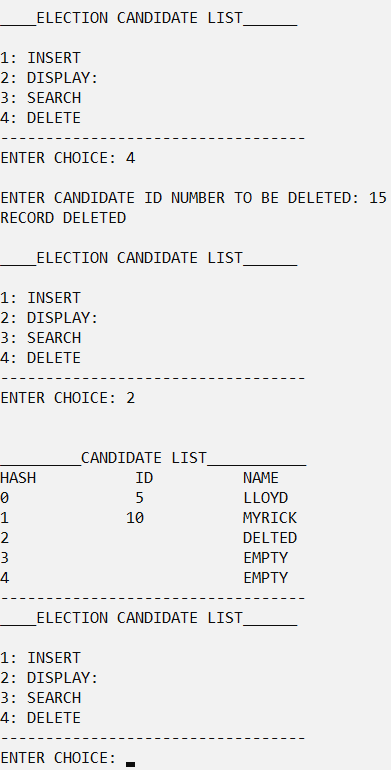
del(table);

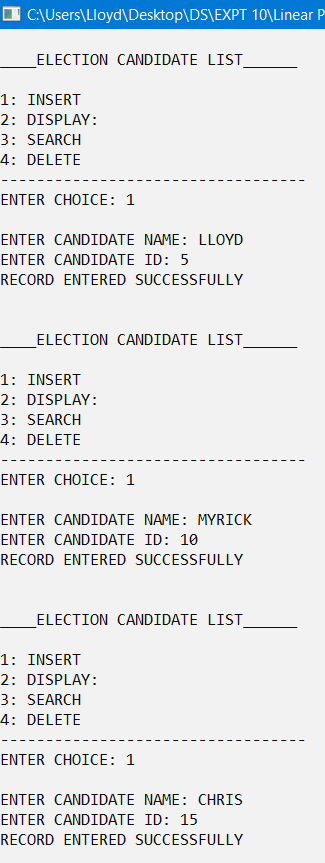
break;

}

}

}

**OUTPUT**

****

**2)**

#include<stdio.h>

#include<string.h>

#define MAX 5

#define HASH(x)(x%MAX)

#define DUPLICATE printf("DUPLICATE ELEMENT\n")

#define SPACE printf(" ")

#define LINE printf("----------------------------------")

enum RecordType{EMPTY,OCCUPIED,DELETED};

struct candidate

{

char name[20];

int idd;

};

struct record

{

struct candidate info;

enum RecordType status;

};

void insert(struct candidate x,struct record table[])

{

int location;

int i,key,h;

key=x.idd;

h=HASH(x.idd);

location=h;

for(i=1;i!=MAX;i++)

{

if((table[location].status==EMPTY)||(table[location].status==DELETED))

{

table[location].info.idd=x.idd;

strcpy(table[location].info.name,x.name);

printf("RECORD ENTERED SUCCESSFULLY\n\n");

table[location].status=OCCUPIED;

return;

}

if(table[location].info.idd==key)

{DUPLICATE;return;}

location=(h+(i\*i))%MAX;

}

printf("TABLE OVERFLOW\n");

}

void display(struct record table[])

{

int i;

printf("\n\_\_\_\_\_\_\_\_\_CANDIDATE LIST\_\_\_\_\_\_\_\_\_\_\_\n");

printf("HASH ");SPACE;

printf("ID");SPACE;

printf("NAME\n");

for(i=0;i!=MAX;i++)

{

printf("%d ",i+1);

if(table[i].status==OCCUPIED)

{

printf("%14.d",table[i].info.idd);SPACE;

printf(" %s",table[i].info.name);

printf("\n");

}

else if(table[i].status==DELETED)

{

SPACE;SPACE;

printf(" DELTED\n");

}

else

{

SPACE;SPACE;

printf(" EMPTY\n");

}

}

LINE;

}

int search(struct record table[],int key)

{

int i;

int h,location;

h=HASH(key);

location=h;

for(i=1;i!=MAX;i++)

{

if(table[location].info.idd==key)

return location;

else if(table[location].status==EMPTY)

return -1;

location=(h+(i\*i))%MAX;

}

return -1;

}

void del(struct record table[])

{

int i,key;

int location;

printf("ENTER CANDIDATE ID NUMBER TO BE DELETED: ");

scanf("%d",&key);

location=search(table,key);

if(location==-1)

printf("CANDIDATE NAME NOT PRSENT IN THE LIST\n");

else

{

table[location].status=DELETED;

printf("RECORD DELETED\n");

}

}

int main()

{

int i,c,l,z;

struct record table[MAX];

struct candidate x;

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

table[i].status=EMPTY;

while(1)

{

printf("\n\_\_\_\_ELECTION CANDIDATE LIST\_\_\_\_\_\_\n");

printf("\n1: INSERT\n");

printf("2: DISPLAY:\n");

printf("3: SEARCH\n");

printf("4: DELETE\n");LINE;

printf("\nENTER CHOICE: ");

scanf("%d",&c);

printf("\n");

switch(c)

{

case 1:

printf("ENTER CANDIDATE NAME: ");

scanf("%s",&x.name);

printf("ENTER CANDIDATE ID: ");

scanf("%d",&x.idd);

insert(x,table);

break;

case 2:

display(table);

break;

case 3:

printf("ENTER ID: ");

scanf("%d",&l);

z=search(table,l);

if(z==-1)

printf("CANDIDATE DEETS NOT PRESENT: ");

else

{

printf("NAME: %s\tID: %d\n",table[z].info.name,table[z].info.idd);

}

break;

case 4:

del(table);

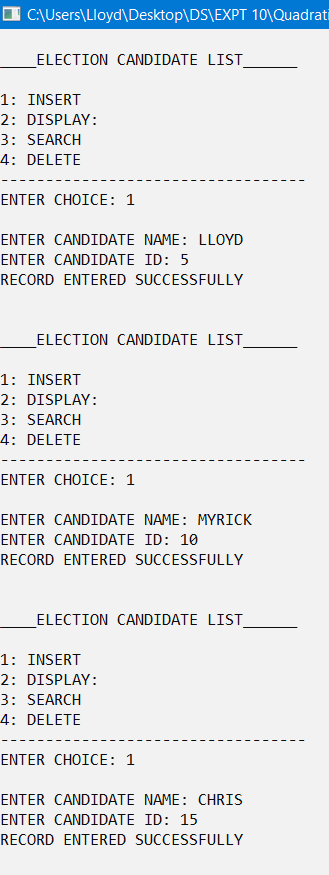
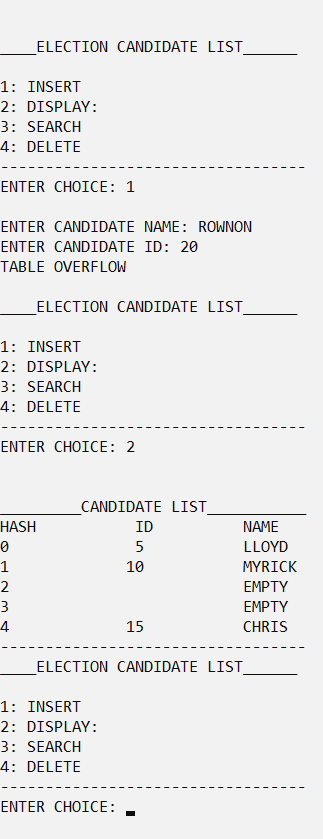
break;

}

}

}

**OUTPUT**

****

**3)**

#include<stdio.h>

#include<string.h>

#define MAX 5

#define HASH(x)(x%MAX)

#define HASH2(x)(7-(x%MAX))

#define DUPLICATE printf("DUPLICATE ELEMENT\n")

#define SPACE printf(" ")

#define LINE printf("----------------------------------")

enum RecordType{EMPTY,OCCUPIED,DELETED};

struct candidate

{

char name[20];

int idd;

};

struct record

{

struct candidate info;

enum RecordType status;

};

void insert(struct candidate x,struct record table[])

{

int location;

int i,key,h;

key=x.idd;

h=HASH(x.idd);

location=h;

for(i=1;i!=MAX-1;i++){

if((table[location].status==EMPTY)||(table[location].status==DELETED))

{

table[location].info.idd=x.idd;

strcpy(table[location].info.name,x.name);

printf("RECORD ENTERED SUCCESSFULLY\n\n");

table[location].status=OCCUPIED;

return;

}

if(table[location].info.idd==key)

{DUPLICATE;return;}

location=(h+(i\*HASH2(key)))%MAX;

}

printf("TABLE OVERFLOW\n");

}

void display(struct record table[])

{

int i;

printf("\n\_\_\_\_\_\_\_\_\_CANDIDATE LIST\_\_\_\_\_\_\_\_\_\_\_\n");

printf("HASH ");SPACE;

printf("ID");SPACE;

printf("NAME\n");

for(i=0;i!=MAX-1;i++)

{

printf("%d ",i+1);

if(table[i].status==OCCUPIED)

{ printf("%14.d",table[i].info.idd);SPACE;

printf(" %s",table[i].info.name);

printf("\n");

}

else if(table[i].status==DELETED)

{

SPACE;SPACE;

printf(" DELTED\n");

}

else

{

SPACE;SPACE;

printf(" EMPTY\n");

}

}

LINE;

}

int search(struct record table[],int key)

{

int i;

int h,location;

h=HASH(key);

location=h;

for(i=1;i!=MAX;i++)

{

if(table[location].info.idd==key)

return location;

else if(table[location].status==EMPTY)

return -1;

location=(h+(i\*HASH2(key)))%MAX;

}

return -1;

}

void del(struct record table[])

{

int i,key;

int location;

printf("ENTER CANDIDATE ID NUMBER TO BE DELETED: ");

scanf("%d",&key);

location=search(table,key);

if(location==-1)

printf("CANDIDATE NAME NOT PRSENT IN THE LIST\n");

else

{

table[location].status=DELETED;

printf("RECORD DELETED\n");

}

}

int main()

{

int i,c,l,z;

struct record table[MAX];

struct candidate x;

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

table[i].status=EMPTY;

while(1)

{

printf("\n\_\_\_\_ELECTION CANDIDATE LIST\_\_\_\_\_\_\n");

printf("\n1: INSERT\n");

printf("2: DISPLAY:\n");

printf("3: SEARCH\n");

printf("4: DELETE\n");LINE;

printf("\nENTER CHOICE: ");

scanf("%d",&c);

printf("\n");

switch(c)

{

case 1:

printf("ENTER CANDIDATE NAME: ");

scanf("%s",&x.name);

printf("ENTER CANDIDATE ID: ");

scanf("%d",&x.idd);

insert(x,table);

break;

case 2:

display(table);

break;

case 3:

printf("ENTER ID: ");

scanf("%d",&l);

z=search(table,l);

if(z==-1)

printf("CANDIDATE DEETS NOT PRESENT: ");

else

{

printf("NAME: %s\tID: %d\n",table[z].info.name,table[z].info.idd);

}

break;

case 4:

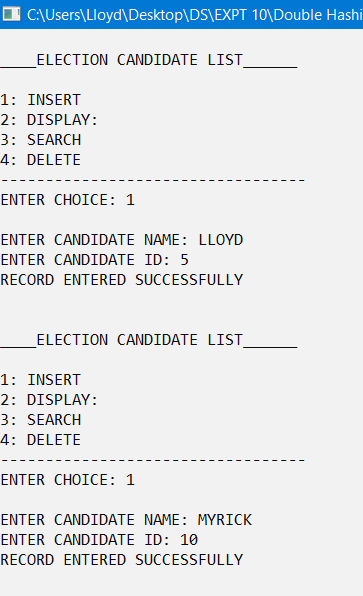
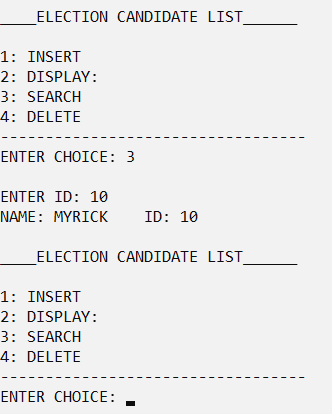
del(table);

break;

}

}

}

**OUTPUT**

**4)**

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#define MAX 7

#define HASH(x)(x%MAX)

#define DUPLICATE printf("DUPLICATE ELEMENT\n")

#define SPACE printf(" ")

#define LINE printf("----------------------------------")

struct candidate

{

char name[20];

int idd;

};

struct record

{

struct candidate info;

struct record \*link;

};

int search(struct record \*table[],int key)

{

int i;

int h,location;

struct record \*ptr;

h=HASH(key);

ptr=table[h];

while(ptr!=NULL)

{

if(ptr->info.idd==key)

return h;

ptr=ptr->link;

}

return -1;

}

void insert(struct candidate x,struct record \*table[])

{

int i,key,h;

struct record \*temp;

key=x.idd;

h=HASH(x.idd);

if(search(table,key)!=-1)

{

printf("DUPLICATE KEY\n");

return;

}

temp=(struct record \*)malloc(sizeof(struct record));

temp->info=x;

temp->link=table[h];

table[h]=temp;

}

void display(struct record \*table[])

{

int i;

struct record \*ptr;

printf("\n\_\_\_\_\_\_\_\_\_CANDIDATE LIST\_\_\_\_\_\_\_\_\_\_\_\n");

for(i=0;i!=MAX;i++)

{

printf("%d---->",i);

ptr=table[i];

while(ptr!=NULL)

{

printf("|%d %s|",ptr->info.idd,ptr->info.name);

if(ptr->link!=NULL)

printf("-->");

ptr=ptr->link;

}

printf("\n");

}

LINE;

}

void del(struct record \*table[])

{

int i,key,h;

int location;

struct record \*temp,\*ptr;

printf("ENTER THE KEY TO BE DELETED: ");

scanf("%d",&key);

h=HASH(key);

if(table[h]==NULL)

{

printf("KEY NOT FOUND\n");

return;

}

if(table[h]->info.idd==key)

{

temp=table[h];

table[h]=table[h]->link;

free(temp);

return;

}

ptr=table[h];

while(ptr!=NULL)

{

if(ptr->link->info.idd==key)

{

temp=ptr->link;

ptr->link=temp->link;

free(temp);

return;

}

ptr=ptr->link;

}

printf("KEY NOT FOUND\n");

}

int main()

{

int i,c,l,z;

struct record \*table[MAX];

struct candidate x;

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

table[i]=NULL;

while(1)

{

printf("\n\_\_\_\_ELECTION CANDIDATE LIST\_\_\_\_\_\_\n");

printf("\n1: INSERT\n");

printf("2: DISPLAY:\n");

printf("3: SEARCH\n");

printf("4: DELETE\n");LINE;

printf("\nENTER CHOICE: ");

scanf("%d",&c);

printf("\n");

switch(c)

{

case 1:

printf("ENTER CANDIDATE NAME: ");

scanf("%s",&x.name);

printf("ENTER CANDIDATE ID: ");

scanf("%d",&x.idd);

insert(x,table);

break;

case 2:

display(table);

break;

case 3:

printf("ENTER ID: ");

scanf("%d",&l);

z=search(table,l);

if(z==-1)

printf("CANDIDATE DEETS NOT PRESENT: ");

else

{

printf("NAME: %s\tID: %d\n",table[z]->info.name,table[z]->info.idd);

}

break;

case 4:

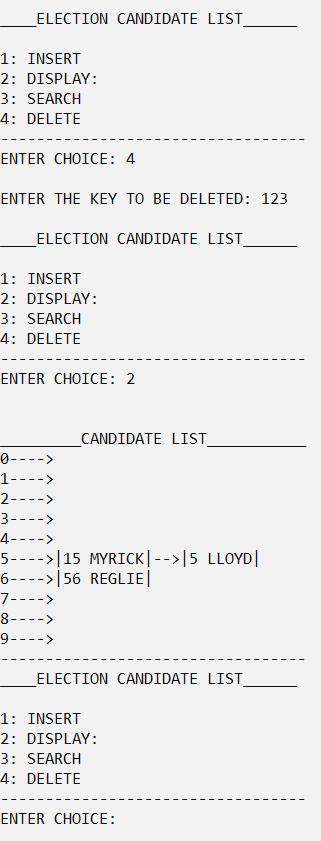
del(table);

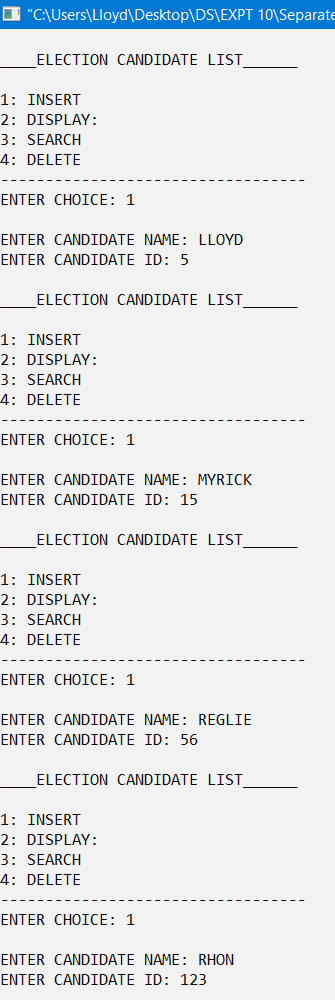
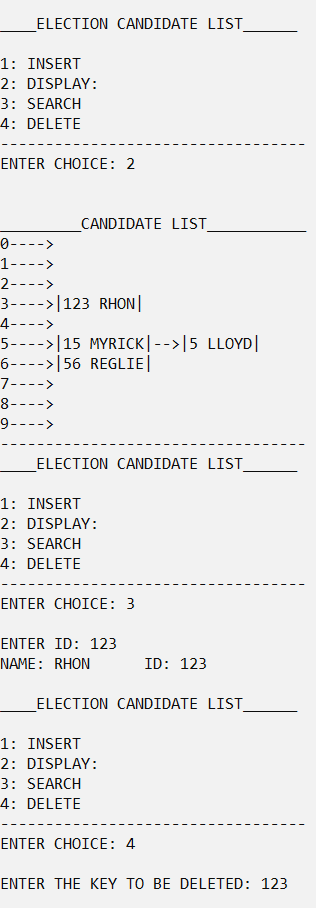
break;

}

}

}

**OUTPUT**

****

**CONCLUSION**

The given problem statements were successfully compiled and executed.

**LEARNINGS AND FINDINGS**

1. Concept of hashing.

2. Implementation of various hashing techniques.

Hashing is a technique in which a smaller number of comparisons are involved and searching can be performed in constant time.

However, there are some limitations in each hashing technique which may pose some problems.

|  |  |
| --- | --- |
| **SR. NO.** | **COMPILATION TIME** |
| 1 | 0.19 s |
| 2 | 0.20 s |
| 3 | 0.20 s |
| 4 | 0.21 s |