**Godavari College Of Engineering, Jalgaon.**

**Subject Name:** Data Structure. **Teacher Name:** Prof. S.S.Shete

**Practical No. :**  11 **Date:**

**Class:** S.E **Roll No:**

**Title :-**  Write a program to implement hashing with (a) Separate Chaining and (b) Open addressing methods.

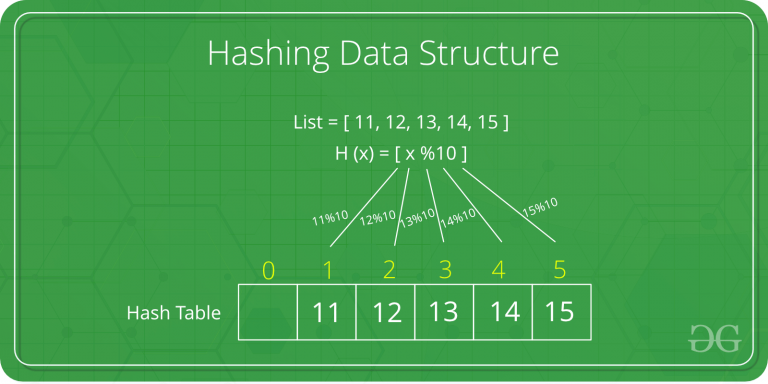
**Aim :-** To implement hashing with (a) Separate Chaining and (b) Open addressing methods.

**Theory:-**

**Hashing Data Structure:-**

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Let a hash function H(x) maps the value **x’th** at the index (x%10) in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



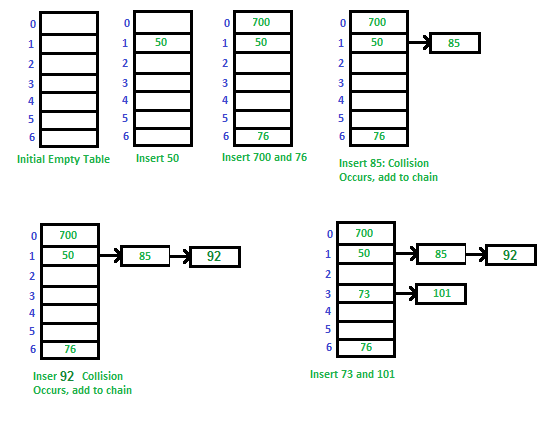
What is Collision?  
Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

What are the chances of collisions with large table?  
Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox, With only 23 persons, the probability that two people have the same birthday is 50%.

How to handle Collisions?  
There are mainly two methods to handle collision:  
1) Separate Chaining  
2) Open Addressing

1) Separate Chaining:-  
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

**Algorithm to search a value in separate chaining**

Hashtable is an array of pointers. All pointers are initialized to NULL head[ TABLE\_SIZE] = NULL)

Step1: Read the value to be searched

step 2: compute the index

index = value % TABLE\_SIZE

step 5: if head[ index] is NULL then print “search element not found” and STOP

step 6: else

step 6.1: store the first node address in pointer c (c = head [ index ])

step 6.2 : if value is equal to data in c then print “search element found” and STOP

step 6.3: else move c to next node ( c = c->next) and if c != NULL go to step 6.2

step 7: if search element is not found in entire linked list (c is NULL) then print “ search element not found”

2) Open Addressing  
Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot’s key doesn’t become equal to k or an empty slot is reached.

Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as “deleted”.

Insert can insert an item in a deleted slot, but the search doesn’t stop at a deleted slot.

**Open Addressing is done following ways:**

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.  
Let hash(x) be the slot index computed using hash function and S be the table size

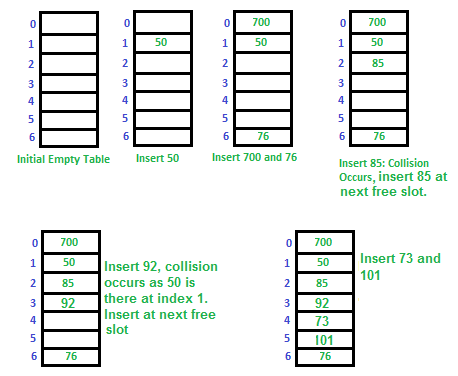
If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

....

Eg. - Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

b) Quadratic Probing :- We look for i2‘th slot in i’th iteration.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S

If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S

If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S

......

c) Double Hashing. :- We use another hash function hash2(x) and look for i\*hash2(x) slot in i’th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*hash2(x)) % S

If (hash(x) + 1\*hash2(x)) % S is also full, then we try (hash(x) + 2\*hash2(x)) % S

If (hash(x) + 2\*hash2(x)) % S is also full, then we try (hash(x) + 3\*hash2(x)) % S

.....

| **S.No.** | **Seperate Chaining** | **Open Addressing** |
| --- | --- | --- |
| 1. | Chaining is Simpler to implement. | Open Addressing requires more computation. |
| 2. | In chaining, Hash table never fills up, we can always add more elements to chain. | In open addressing, table may become full. |
| 3. | Chaining is Less sensitive to the hash function or load factors. | Open addressing requires extra care for to avoid clustering and load factor. |
| 4. | Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted. | Open addressing is used when the frequency and number of keys is known. |
| 5. | Cache performance of chaining is not good as keys are stored using linked list. | Open addressing provides better cache performance as everything is stored in the same table. |
| 6. | Wastage of Space (Some Parts of hash table in chaining are never used). | In Open addressing, a slot can be used even if an input doesn’t map to it. |
| 7. | Chaining uses extra space for links. | No links in Open addressing |

**Program:-**

**a) Seprate Chaining Hashing :-**

#include <stdio.h>

#include <stdlib.h>

#define TABLE\_SIZE 10

struct node

{

int data;

struct node \*next;

};

struct node \*head[TABLE\_SIZE]={NULL},\*c;

void insert()

{

int i,key;

printf("\nEnter a value to insert into hash table\n");

scanf("%d",&key);

i=key%TABLE\_SIZE;

struct node \* newnode=(struct node \*)malloc(sizeof(struct node));

newnode->data=key;

newnode->next = NULL;

if(head[i] == NULL)

head[i] = newnode;

else

{

c=head[i];

while(c->next != NULL)

{

c=c->next;

}

c->next=newnode;

}

}

void search()

{

int key,index;

printf("\nEnter the element to be searched\n");

scanf("%d",&key);

index=key%TABLE\_SIZE;

if(head[index] == NULL)

printf("\n Search element not found\n");

else

{

for(c=head[index];c!=NULL;c=c->next)

{

if(c->data == key)

{

printf("search element found\n");

break;

}

}

if(c==NULL)

printf("\n Search element not found\n");

}

}

void display()

{

int i;

for(i=0;i<TABLE\_SIZE;i++)

{

printf("\nEntries at index %d\n",i);

if(head[i] == NULL)

{

printf("No Hash Entry");

return;

}

else

{

for(c=head[i];c!=NULL;c=c->next)

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

}

}

}

int main()

{

int ch;

printf("\n\n\t\*\*\*\* Seprate Chaining Hashing. \*\*\*\*");

printf("\n\n\t 1) Insert. ");

printf("\n\t 2) Display. ");

printf("\n\t 3) SEarch. ");

printf("\n\t 4) Exit. ");

rep:

printf("\n=====================================================");

printf("\n\t Enter Your Choice :- ");

scanf("%d",&ch);

switch(ch)

{

case 1:

insert();

break;

case 2:

display();

break;

case 3:

search();

break;

case 4:

exit(0);

}

if(ch!=4)

{ goto rep; }

return 0;

}

b) Open Addressing. :-

#include <stdio.h>

int tsize;

int hasht(int key)

{

int i ;

i = key%tsize ;

return i;

}

//-------LINEAR PROBING-------

int rehashl(int key)

{

int i ;

i = (key+1)%tsize ;

return i ;

}

//-------QUADRATIC PROBING-------

int rehashq(int key, int j)

{

int i ;

i = (key+(j\*j))%tsize ;

return i ;

}

int main()

{

int key,arr[20],hash[20],i,n,op,j,k ;

printf ("Enter the size of the hash table: ");

scanf ("%d",&tsize);

printf ("\nEnter the number of elements: ");

scanf ("%d",&n);

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

hash[i]=-1 ;

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

{

printf ("\n\t Enter Elements: ");

scanf("%d",&arr[i]);

}

printf("\n\n=======================================================");

do

{

printf("\n\n1.Linear Probing\n2.Quadratic Probing \n3.Exit \nEnter your option: ");

scanf("%d",&op);

switch(op)

{

case 1:

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

hash[i]=-1 ;

for(k=0;k<n;k++)

{

key=arr[k] ;

i = hasht(key);

while (hash[i]!=-1)

{

i = rehashl(i);

}

hash[i]=key ;

}

printf("\nThe elements in the array are: ");

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

{

printf("\n Element at position %d: %d",i,hash[i]);

}

printf("\n\n=======================================================");

break ;

case 2:

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

hash[i]=-1 ;

for(k=0;k<n;k++)

{

j=1;

key=arr[k] ;

i = hasht(key);

while (hash[i]!=-1)

{

i = rehashq(i,j);

j++ ;

}

hash[i]=key ;

}

printf("\nThe elements in the array are: ");

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

{

printf("\n Element at position %d: %d",i,hash[i]);

}

printf("\n\n=======================================================");

break ;

}

}while(op!=3);

printf("\n\t Thanks For Visiting Program. \n\t NOw You Are EXiting From Program.");

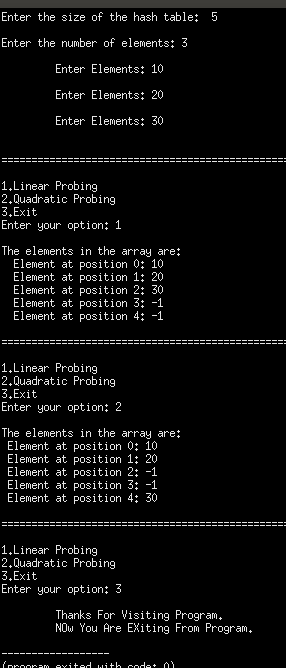
return 0;

}



**Output:-**

**1) Separate Chaining.**



**2) Open addressing.**

**Conclusion:-**