Data Structures Lecture-1

1. Static vs. *Dynamic* memory allocation

A program requires memory to store its data. Memory for a program can be allocated either at compile time or at run time. Memory allocated during compile time is called static (compile time) memory allocation. Memory allocated during run time of program is called dynamic (run time) memory allocation.

Memory allocated during compile time is automatically released upon program termination. Dynamically allocated memory need to be released forcibly when it is no longer in use.

Dynamic memory allocation in C-language

C-language offers following three functions to allocate memory at run time…

i. malloc(). This function is important as we will use this in our linked implementation of data-structures.

ii. calloc()

iii.realloc()

The malloc() function

malloc() function in c-language is used to allocate memory when the program is running. It takes size of memory block to be allocated as argument and returns the address of very first byte of allocated block(if malloc() get success in allocating memory). If malloc() fails to allocate memory then it returns a NULL.

The type of pointer that malloc() returns is void, so we can convert it into any type ofpointer as needed.

Syntax of malloc().

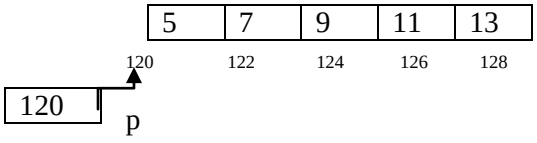
Ptr_var=(type_caste*)malloc(size_of_block);

Example: To allocate memory for storing 5 integers---

int *p;

p=(int*)malloc(5*sizeof(int);

This is what happens in memory. (addresses are assumed)



p+0 means 120

p+1 means 122

p+2 means 124

p+3 means 126

p+4 means 128

*(p+0)=5; *(p+1)=7; *(p+2)=9; *(p+3)=11; *(p+4)=13;

Note:

free() function is used to release memory allocated during run time.

We use the following to release memory….

free(p); //will delete memory block pointed by pointer p.

Program example 1:

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <process.h>
main()
{
  char *p;
  /* allocate memory for string */
  p = (char *) malloc(10);
  if(p==NULL)
  {
    printf("Not enough memory to allocate buffer\n");
    exit(1);  /* terminate program if malloc() fails */
  }
  /* copy "RIYAZ" into string */
  strcpy(p, "RIYAZ");
  /* display string */
  printf("String is %s\n", p);
  /* free memory */
  free(p);
}
```

Program example 2:

```
#include <stdio.h>
#include <malloc.h>
#include <process.h>
main()
{
  int  *p;
  /* allocate memory for a single integer */
  p = (int *) malloc(sizeof(int));
  if(p==NULL)
  {
    printf("Not enough memory to allocate buffer\n");
    exit(1);  /* terminate program if malloc() fails */
  }
  /* assign 25 to memory */
  *p=25;
  /* display string */
  printf("The number is  %d\n",* p);
  /* free memory */
  free(p);
}
```

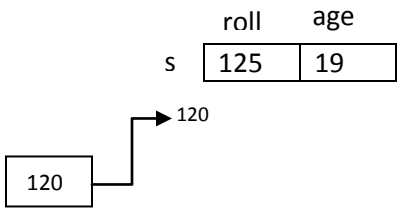
Accessing structure members using pointer:

Syntax:

Pointer_name-> member_name;

Example:

```
struct student
{
int roll;
int age;
};
main()
{
struct student s1;
struct student *p;
p=&s1;
p->roll=125;
p->age=19;
printf("%d %d",p->roll,p->age);
}
```



Note:

A field in structure may be pointer too.

Exaample:

```
Struct record
{
int roll;
struct record *link;
};
```

Here, the link filed is a pointer of **record** type, which can store address of any memory block of **record** type.

```
struct record *p;                    //Declaring pointer p of struct record type.
```

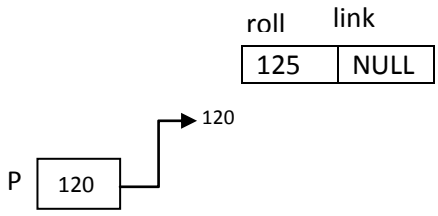
p=(struct record*)malloc(sizeof(struct record)); //allocating memory

Above line will allocate a memory block of type **struct record** and return its address to pointer p.

p->roll=125;

p->link=NULL;

The memory representation is shown below.



Now

struct record *p1; //Declaring pointer p1 of struct record type.

p1=(struct record*)malloc(sizeof(struct record)); //allocating memory

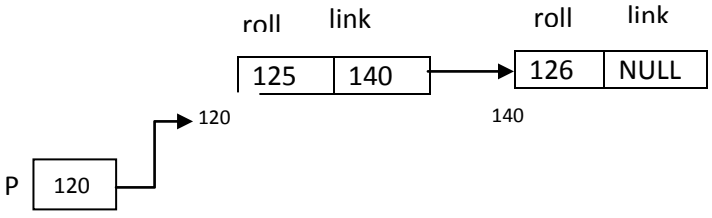
Above line will allocate a memory block of type **struct record** and return its address to pointer p.

p1->roll=126;

p1->link=NULL;

p->link=p1;

Now,the memory representation with two records(node) is shown below.



This way we can create a list of roll numbers of students in memory. Each record contains the address of its immediate next record. And the last record (node) contains a NULL in its link field.

And such a list is known as linked list. And this is how we can create dynamic list in which memory (node) is allocated when the program is running.

Data Structures Lecture-2(Linked list)

Linked list

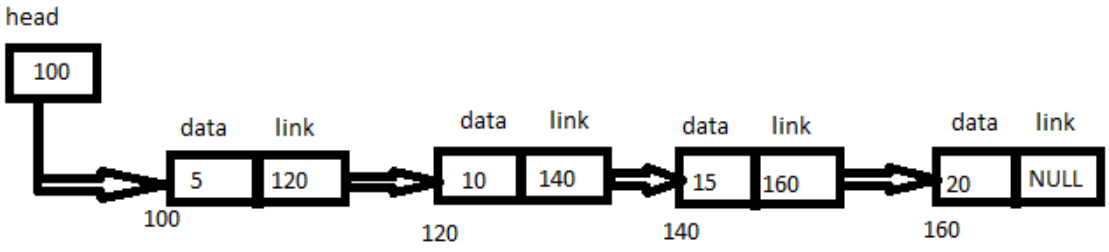
A linked list is a collection of nodes (record) each pointing to the next. A node of linked list has two fields, first is **data** field and second is **link** field. The data field contains information to be stored and the link field contains the memory address of immediate next node. The link filed of last node contains NULL. To maintain integrity of linked list we established an extra pointer called head to keep address of very first node of linked list.

Structure of node of linked list

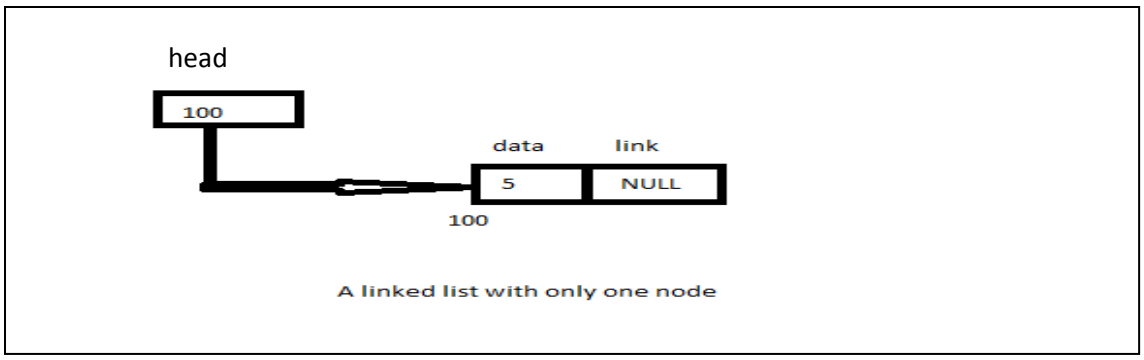
```
Struct node
{
int data;
struct node *link;
};
```

The type of link pointer is struct node as it has to store memory address of another node which is also of struct node type. And hence we call this a self referencing structure.

A node of linked list in memory



A linked list



Printing Linked List

Void display()

{

Struct node *t;

If(head==NULL)

{

printf("Linked list is empty");

getch();

return;

}

t=head; **//head is a global pointer which is pointing to first node if linked list is not empty.**

while(t!=NULL)

{

printf("%d\n",t->data);

t=t->link;

}

getch();

}

If head equals NULL then, linked list is empty.

We write t=head,

and repeatedly print **data** part of pointer **t** and then move **t** to

ual to head.

Operations on linked list

The operations that can be performed on linked list are:

1.Creation of linked list.

2. Display/traversal of linked list.

3. Insertion of a new node at the beginning of the linked list.

4. Insertion of a new node at the end of the linked list.

5. Insertion of a new node at a given position of linked list.

6. Deletion of an existing node from the beginning of linked list.

7. Deletion of an existing node from the end of linked list.

8. Merge two linked list.

9. Searching.

10. Sorting.

Program to demonstrate creation, insertion, deletion and display operation on linked list.

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>
struct node
{
int data;
struct node *link;
};
struct node *head=NULL;
void insert_last(int n)
```

```
{
struct node *newnode,*temp;
newnode=(struct node*)malloc(sizeof(struct node));
newnode->data=n;
newnode->link=NULL;
if(head==NULL)
{
head=newnode;
return;
}
temp=head;
while(temp->link!=NULL)
temp=temp->link;
temp->link=newnode;
}

void insert_begin(int n)
{
struct node *newnode;
newnode=(struct node*)malloc(sizeof(struct node));
newnode->data=n;
newnode->link=head;
head=newnode;
}
```

```
void insert_specific(int n,int pos)
{
struct node *newnode,*temp;
int i;
temp=head;
for(i=1;i<pos-1;i++)
{
temp=temp->link;
if(temp==head)
{
printf("Position out of range\n");
getch();
return;
}
}
newnode=(struct node*)malloc(sizeof(struct node));
newnode->data=n;
newnode->link=temp->link;
temp->link=newnode;
}

void del_begin()
{
int n;
struct node *temp;
```

```
if(head==NULL)
{
printf("Linked list empty\n");
getch();
return;
}
temp=head;
n=temp->data;
head=head->link;
printf("%d is deleted\n",n);
free(temp);
getch();
}

void del_last()
{
int n;
struct node *temp,*p;
if(head==NULL)
{
printf("Linked list empty\n");
getch();
return;
}
temp=head;
```

```
while(temp->link!=NULL)
{
p=temp;
temp=temp->link;
}
n=temp->data;
p->link=NULL;
printf("%d is deleted\n",n);
free(temp);
getch();
}

void del_specific(int pos)
{
int n,i;
struct node *temp,*p;
if(head==NULL)
{
printf("Linked list empty\n");
getch();
return;
}
temp=head;
for(i=1;i<n;i++)
{
```

```
p=temp;
temp=temp->link;
if(temp==NULL)
{
printf("Position out of range\n");
getch();
return;
}
}
n=temp->data;
p->link=temp->link;
printf("%d is deleted\n",n);
free(temp);
getch();

}

void display()
{
struct node *temp;
if(head==NULL)
{
printf("Linked list empty");
getch();
return;
```

```
}
temp=head;
while(temp!=NULL)
{
printf("%d\n",temp->data);
temp=temp->link;
}
getch();
}
main()
{
int n,pos,ch;
while(1)
{
clrscr();
puts("1.Insertion begin");
puts("2.Insertion last");
puts("3.Insertion at a given position");
puts("4.Deletion begin");
puts("5.Deletion last");
puts("6.Deletion from given position");
puts("7.Display");
puts("8.Exit");
printf("Enter your choice...");
scanf("%d",&ch);
```

```
switch(ch)
{
case 1:
printf("Enter an item");
scanf("%d",&n);
insert_begin(n);
break;
case 2:
printf("Enter an item");
scanf("%d",&n);
insert_last(n);
break;
case 3:
printf("Enter item");
scanf("%d",&n);
printf("Enter position");
scanf("%d",&pos);
insert_specific(n,pos);
break;
case 4:
del_begin();
break;
case 5:
del_last();
break;
```

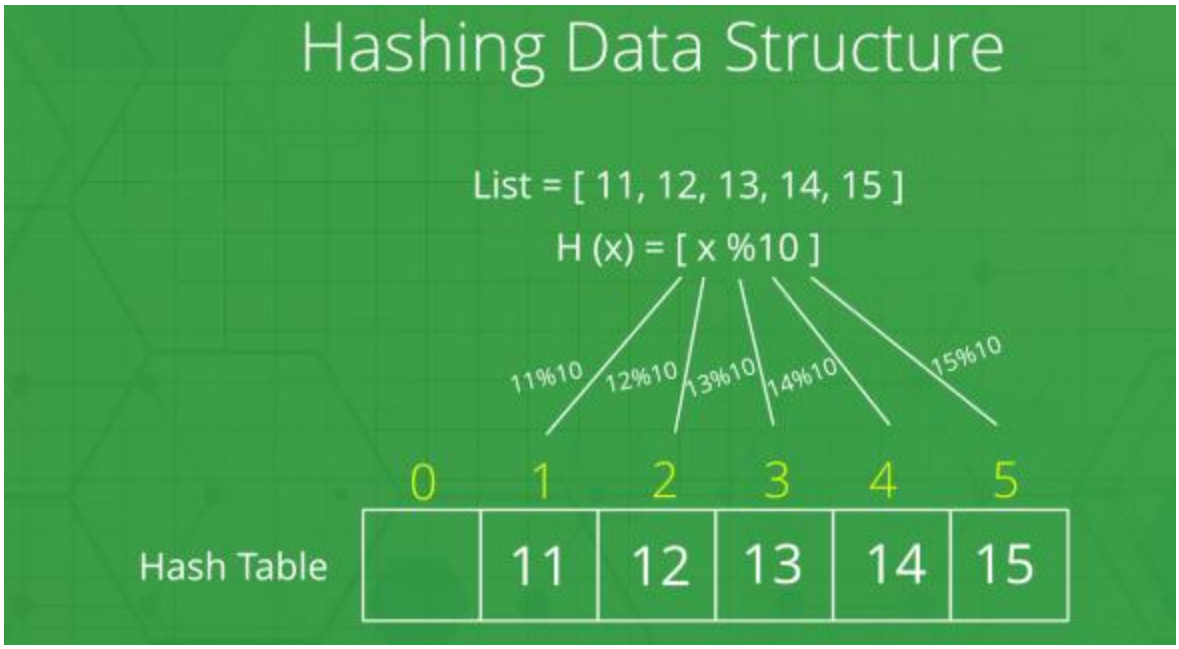
```
case 6:
del_specific(pos);
break;
case 7:
display();
break;
case 8:
exit(0);
}
}
}
```

Data structure Lecture-9

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 (find) 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 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 are Hash Functions and How to choose a good Hash Function?

What is a Hash Function?

A function that converts a given big phone number(any big number) to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as the index in the hash table.

What is meant by Good Hash Function?

A good hash function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

For example: For phone numbers, a bad hash function is to take the first three digits. A better function is considered the last three digits. Please note that this may not be the best hash function. There may be better ways.

In practice, we can often employ ***heuristic techniques*** to create a hash function that performs well. Qualitative information about the distribution of the keys may be useful in this design process. In general, a hash function should depend on every single bit of the key, so that two keys that differ in only one bit or one group of bits (regardless of whether the group is at the beginning, end, or middle of the key or present throughout the key) hash into different values.

Thus, a hash function that simply extracts a portion of a key is not suitable. Similarly, if two keys are simply digited or character permutations of each other *(such as 139 and 319)*, they should also hash into different values.
The two heuristic methods are:

i. ***hashing by division*** and

ii. ***hashing by multiplication***

which are as follows:

1. **The mod method:**
   - In this method for creating hash functions, we map a key into one of the slots of table by taking the remainder of key divided by table_size. That is, the hash function is
   - h(key) = key **mod** table_size
   - i.e. key % table_size
   - Since it requires only a single division operation, hashing by division is quite fast.
   - When using the division method, we usually avoid certain values of table_size like table_size should not be a power of a number suppose **r**, since if ***table_size = r^p***, then h(key) is just the p lowest-order bits of key. Unless we know that all low-order p-bit patterns are equally likely, we are better off designing the hash function to depend on all the bits of the key.
   - It has been found that the best results with the division method are achieved when the table size is prime. However, even if table_size is prime, an additional restriction is called for. If **r** is the number of possible character codes on an computer, and if **table_size** is a prime such that r % table_size equal 1, then hash function **h(key) = key % table_size** is simply the *sum of the binary representation of the characters* in the key mod table_size.

   Example:
   - Suppose **r** = 256 and **table_size** = 17, in which r % table_size i.e. 256 % 17 = 1.
   - So for **key = 37596**, its hash is
     37596 % 17 = 12
   - But for **key = 573**, its hash function is also
     573 % 12 = 12
   - Hence it can be seen that by this hash function, many keys can have the same hash. This is called Collision.
   - A prime not too close to an exact power of 2 is often good choice for table_size.
2. **The multiplication method:**
   - In multiplication method, we multiply the key **k** by a constant real number **c** in the range **0 < c < 1** and extract the *fractional part of* ***k * c***.
   - Then we multiply this value by table_size **m** and take the floor of the result. It can be represented as
   - **h(k) = floor (m * (k * c mod 1))**
   - **or**
   - **h(k) = floor (m * frac (k * c))**

where the function **floor(x)**, available in standard library *math.h*, yields the integer part of the real number x, and **frac(x)** yields the fractional part. **[frac(x) = x – floor(x)]**

- An advantage of the multiplication method is that the value of *m is not critical*, we typically choose it to be a power of 2 (**m = 2^p** for some integer **p**), since we can then easily implement the function on most computers
- Suppose that the word size of the machine is **w** bits and that key fits into a single word.
- We restrict **c** to be a fraction of the form **s / (2^w)**, where s is an integer in the range **0 < s < 2^w**.
- Referring to figure, we first multiply key by the w-bit integer **s = c * 2^w**. The result is a 2w-bit value
- **r1 * 2^w + r0**
-
- where **r1** = high-order word of the product
- **r0** = lower order word of the product
- Although this method works with any value of the constant **c**, it works better with some values than the others.

  c ~ (sqrt (5) – 1) / 2 = 0.618033988 . . .

  is likely to work reasonably well.

3. **Example:**
   - Suppose **k** = 123456, **p** = 14,
   - **m** = 2^14 = 16384, and **w** = 32.
   - Adapting **Knuth's suggestion**, **c** to be fraction of the form **s / 2^32**.
   - Then **key * s** = 327706022297664 = (76300 * 2^32) + 17612864,
   - So **r1** = 76300 and **r0** = 176122864.
   - The 14 most significant bits of r0 yield the value **h(key) = 67.**

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%.

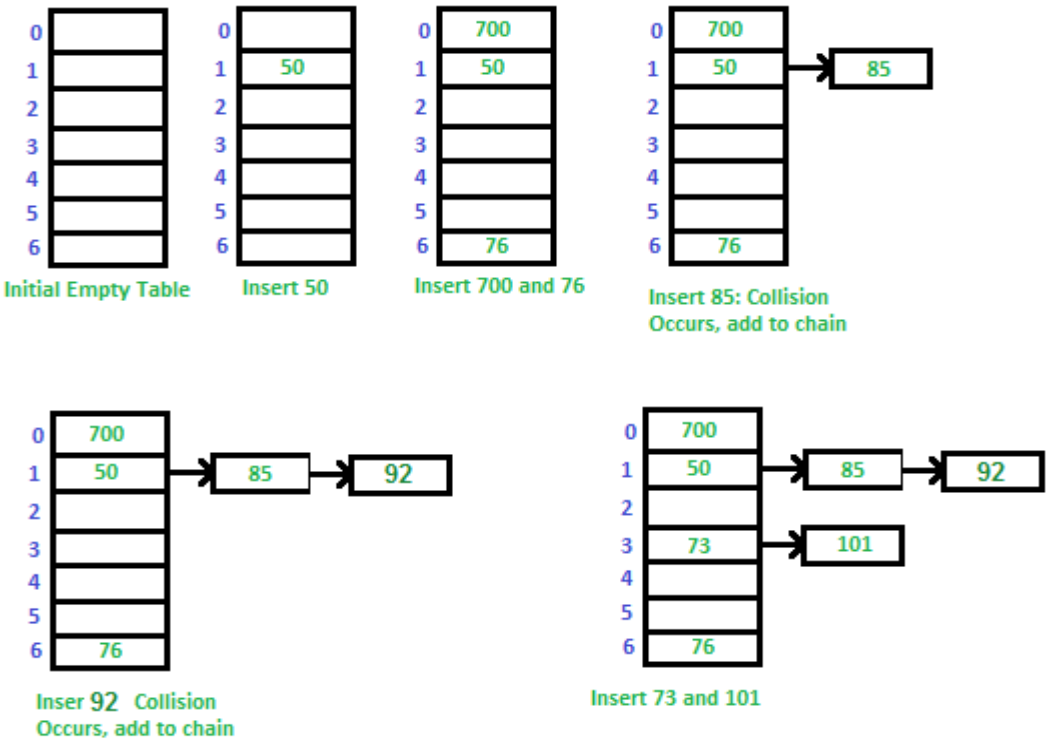
How to handle Collisions?
There are mainly two methods to handle collision:

1) Separate Chaining (open hashing)

2) Open Addressing (close hashing)

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.

Advantages:
1) Simple to implement.
2) Hash table never fills up, we can always add more elements to the chain.
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:
1) Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
2) Wastage of Space (Some Parts of hash table are never used)
3) If the chain becomes long, then search time can become O(n) in the worst case.
4) Uses extra space for links.

Performance of Chaining:
Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing).

m = Number of slots in hash table

n = Number of keys to be inserted in hash table

Load factor $\alpha = n/m$

Expected time to search = $O(1 + \alpha)$

Expected time to insert/delete = $O(1 + \alpha)$

Time complexity of search insert and delete is

$O(1)$ if α is $O(1)$

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

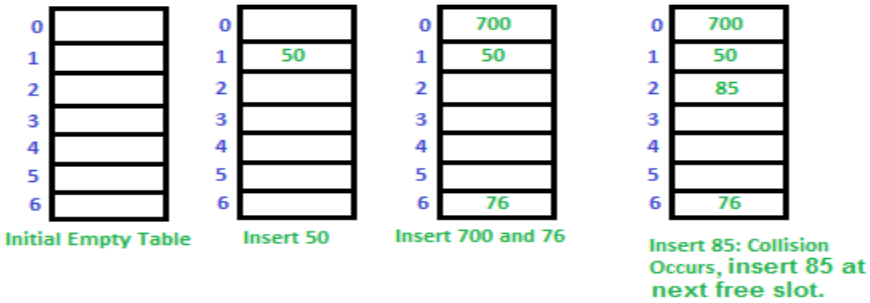
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

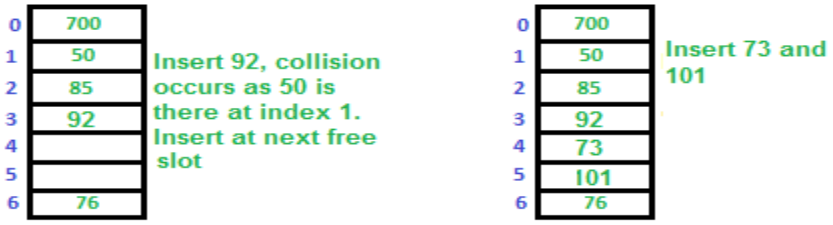
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

.....

.....

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 i^2'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

.....

.....

Comparison of above three:
Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	**Separate 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

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table

n = Number of keys to be inserted in the hash table

Load factor $\alpha = n/m$ (< 1)

Expected time to search/insert/delete $< 1/(1 - \alpha)$

So Search, Insert and De

Data Structures Lecture-10

Searching

Searching means to find the location of a given element in a list (array).

Searching operation requires tlowo things: (i). An array (list) of elements and

(ii). A key element to search on array.

Search operation is said to be **success** if given key element exists in the list.

Search operation is said to be fail if given key does not exist in the list.

Example: Consider the following array with 10 integer elements. And we want to search key 45.

12	19	17	25	10	45	24	15	40	35

For key=45, search is success with following output:

45 is present at position 6 and index 5.

For key=29, search is failed with following output:

29 does not exist in the list.

We have two searching techniques available:

i). Linear search.

ii). Binary search.

i).Linear search proceed by testing each element for a match with **key** element starting from 0^{th} index onward. And as soon an element matches to the key, the search stops with success. If element is not present then after testing **key** with all elements search fails and stop.

Program to implement linear search:

```
main()
{
int a[10],n=10,key,i,f=0;
clrscr();
printf("Enter %d numbers\n",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("Enter number to search:");
scanf("%d",&key);
for(i=0;i<n;i++)
{
        if(key==a[i])
        {
        f=1;
        break;
        }
}
if(f==1)
printf("Element found at position %d",i+1);
else
printf("Element not found");
getch();
}
```

ii).Binary search works only on sorted array (list). Suppose a list is sorted in ascending order, then binary search proceed by comparing the key with middle element of array and

i).if the key is equals to middle element then search immediately stop with success.

ii). if key is less than middle element then binary search is performed in left half of the array (excluding middle).

iii). if key is greater than middle element then binary search is performed in right half of the array (excluding middle).

The above process terminates upon, either success of search or certain condition of failure is satisfied.

Program to implement binary search:

```
main()
{
int a[10],n=10,low,high,mid,f=0,i,key;
clrscr();
printf("Enter %d numbers in ascending order",n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("Enter number to search:");
scanf("%d",&key);
```

```
//Binary search begins here...
low=0;
high=n-1;
while(low<=high)
{
mid=(low+high)/2;
if(key==a[mid])
{
f=1;
break;
}
else if(key<a[mid])
high=mid-1;
else
low=mid+1;
}
if(f==1)
printf("Element found at position %d",mid+1);
else
printf("Element does not exist");
getch();
}
```