UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES

Searching- Linear Search - Binary Search. Sorting - Bubble sort - Selection sort - Insertion sort - Shell sort – Radix sort. Hashing- Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

5.1 SEARCHING:

Searching is a technique of finding an element from a given data list or set of elements like an array, linked list or tree. It is a technique to find out an element in the sorted or unsorted list.

Three Cases in Finding the Element During Search:

- Best case: Element is found in first comparison.
- Worst Case: Element is found in last comparison
- Average Case: Number of comparisons is greater than best case and less than the worst case.

Classification of Searching:

- **Linear Search or sequential search** – it is a method of searching the elements in a list sequence. The element is searched in sequential order from the first to last element.
- **Binary search-** It divides the list into two partitions and then the element is searched in the list.

5.1.1 LINEAR SEARCH

Implementation of Linear Search:

The linear or sequential means the items are stored in systematic manner. Linear search can be applied either in sorted or unsorted linear data structure. The expected element is to be searched in the entire data structure in a sequential method from starting to the last element. It consumes more time for searching.

Routine:

```
int linearsearch(int a[], int n, int key)
{
int i;
for(i=0;i<n;i++)
{
if(a[i]==key)
if(s==a[i])
```

```
{
printf("Element %d is found in position %d",s,i+1);
f=2;
break;
}
if(f==1)
printf("The element is not found in the array");
}
```

Advantage:

- It is simple to code and easy to understand
- It is suited to all data sets
- The data need not be pre sorted

Disadvantage:

- The worst case performance scenario for a linear search is that it needs to loop through the entire collection; either because the item is the last one, or because the item is not found.

Time Complexity:

- Best case : O (1)
- Average case : O (N)
- Worst case : O (N)

Example Program

```
#include<stdio.h>
#include<conio.h>

void main()
{
int i,j,n,s,a[20],f=2;
clrscr();
printf("Enter the size of the array");
scanf("%d",&n);

printf("Enter the element in array");
for(i=0;i<n;i++)
scanf("%d",&a[i]);

printf("Enter the element to be search");
scanf("%d",&s);

for(i=0;i<n;i++)
if(s==a[i])
{
```

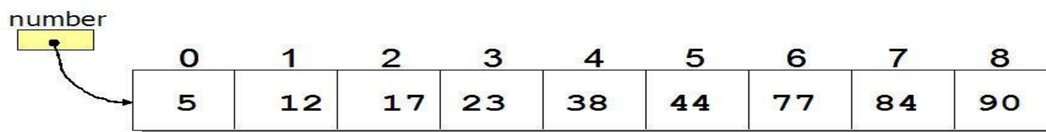
OUTPUT:

```
Enter the size of the array 5
Enter the element in array 3 2 1 5 6
Enter the element to be search 5
Element 5 is in the position 4
```

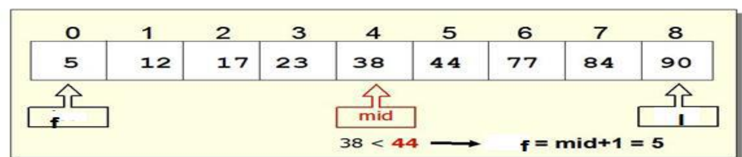
```
printf("Element %d is found in position %d",s,i+1);
f=2;
break;
}
if(f==1)
printf("The element is not found in the array");
getch();
}
```

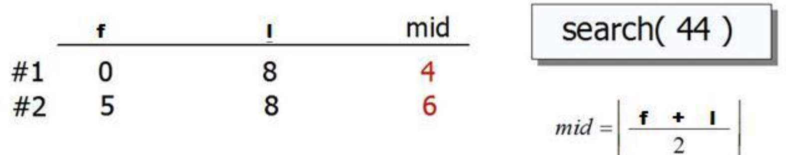
5.1.2 BINARY SEARCH

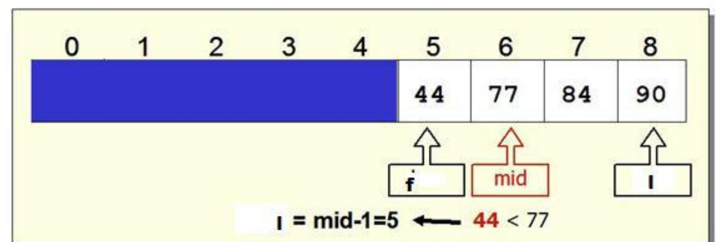
Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array. Binary search is the **divide and conquer** strategy.

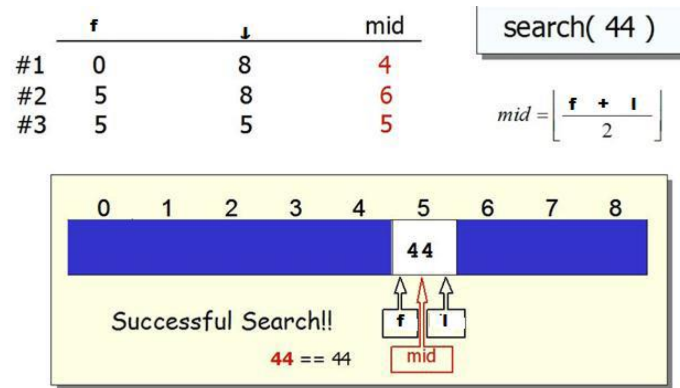



```
int bsearch(int a,int n, int search)
{
int first = 0, last = n - 1, middle;
middle = (first+last)/2;
while( first <= last )
{
if ( a[middle] < search )
first = middle + 1;
else if ( a[middle] >search )
{
last=middle+1;
}
else
printf("Element Found:");
}
if ( first > last )
printf("Not found!";
}
```









C Program to search the location of given data in the array / Binary Search.

```
#include <stdio.h>
#include<conio.h>
int main()
{
int i, first, last, middle, n, search, array[100];
printf("Enter number of elements\n");
scanf("%d",&n);
printf("Enter %d integers\n", n);
for ( i = 0 ; i < n ; i++ )
scanf("%d",&array[i]);
printf("Enter value to find\n");
scanf("%d",&search);
first = 0;
last = n - 1;
middle = (first+last)/2;
while( first <= last )
{
if ( array[middle] < search )
first = middle + 1;
else if ( array[middle] >search )
{
last=middle+1;
}
else
printf("%d found at location %d.\n", search, middle+1);
}
if ( first > last )
printf("Not found! %d is not present in the list.\n", search);
return 0;
}
```

Output:

Enter the number of elements: 7

Enter integers: 10, 15, 17, 19, 21, 26, 50

Enter the value to find: 15

15 is found at location 1.

Analysis:

Binary search is faster than linear search but list should be sorted, hashing is faster than binary search and performs searches in constant time.

- Worst case performance : **O (log n)**
- Best case performance : **O (1)**
- Average case performance : **O (log n)**

Application: Searching in a telephone directory

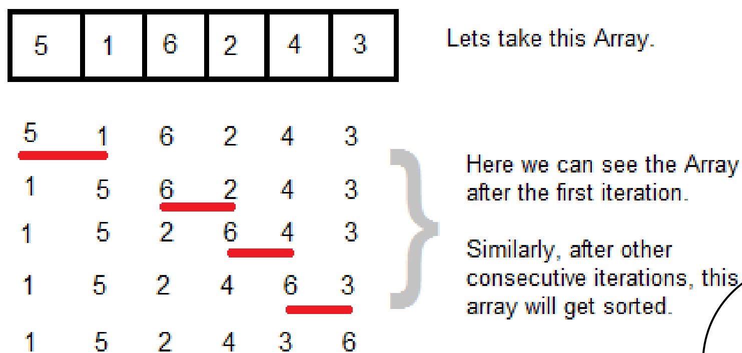
5.2 SORTING

5.2.1 BUBBLE SORTING

Bubble Sort is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares the entire element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.



Advantage:

- Easy to use and implement.
- It is an in place sort.

Disadvantage:

- Bubble sort is not efficient on larger lists.
- More number of comparisons and swapping is done in bubble sort.

Analysis of Bubble Sort:

- Best case analysis : **$O(N^2)$**
- Average case analysis : **$O(N^2)$**
- Worst case Analysis : **$O(N^2)$**

Sorting using Bubble Sort Algorithm

```
Void bubblesort(int a[],int n)
{
int i,j,temp;
for(i=0;i<n-1;i++)
{
for(j=0;j<n-1;j++)
{
if(a[j]>a[j+1])
{
```

```
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}
}
```

5.2.2 SELECTION SORTING

Selection sorting is conceptually the simplest sorting algorithm. These algorithms first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

How Selection Sorting Works

Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.

Sorting using Selection Sort Algorithm

```
void selectionSort(int a[], int n)
{
 int i, j, t;
 for(i=0; i < n; i++ )
 {
  for(j=i+1; j < n; j++)
```

```
    {
      if(a[i] > a[j])
      {
        temp = a[i];
                a[i] = a[min];
                a[min] = temp;
      }
    }
  }
}
```

Program:

```
#include<stdio.h>
#include<conio.h>
void selectionsort(int a[10],int n);
void main()
{
int a[10],n,i;
printf("\n\t\t Enter the Limit");
scanf("%d",&n);
printf("\n\t\t Enter the element");
for(i=0;i<n;i++)
scanf("\n%d",&a[i]);
selectionsort(a,n);
}
void selectionSort(int a[], int n)
{
  int i, j, t;
  for(i=0; i < n; i++ )
  {
    for(j=i+1; j < n; j++)
    {
      if(a[i] > a[j])
      {
        temp = a[i];
                a[i] = a[min];
                a[min] = temp;
      }
    }
  }
```

```
 printf("\n The sorted list of element is..\n");
 for(i=0;i<n;i++)
 printf("\n%d",a[i]);
}
```

Advantage:

- It performs well on small list
- It is an in-place sorting algorithm, so on additional storage is required beyond what is needed to hold the original list.

Disadvantage:

- It is inefficient when dealing with a huge list of items.
- This requires n-squared number of steps for sorting n elements.
- The selection sort is only suitable for a list of few elements that are in random order.

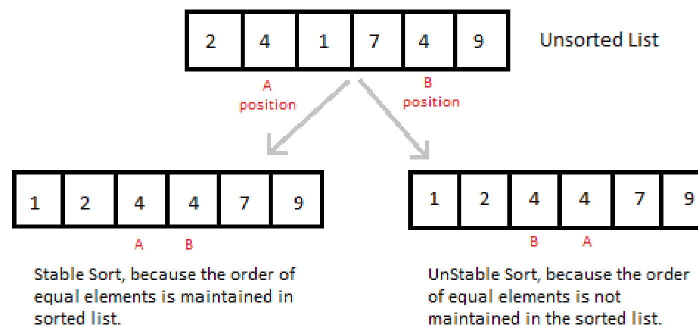
Complexity Analysis of Selection Sorting

- Worst Case Time Complexity : $\mathbf{O(n^2)}$
- Best Case Time Complexity : $\mathbf{O(n^2)}$
- Average Time Complexity : $\mathbf{O(n^2)}$

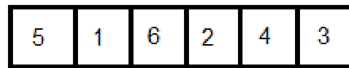
5.2.3 INSERTION SORTING

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

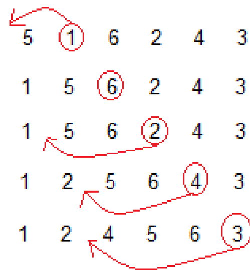
1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys



How Insertion Sorting Works



Lets take this Array.



(Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

Program:

```
#include<stdio.h>
#include<conio.h>
void Insertsort(int a[10],int n);
void main()
{
int a[10],n,i;
clrscr();
printf("\n\t\t Enter the Limit");
scanf("%d",&n);
printf("\n\t\t Enter the element");
for(i=0;i<n;i++)
scanf("\n%d",&a[i]);
Insertsort(a,n);
getch();
}

void Insertsort(int a[10],int n)
{
int i,j,t;
```

Complexity Analysis of Insertion Sorting

- Worst Case Time Complexity : **$O(n^2)$**
- Best Case Time Complexity : **$O(n)$**
- Average Time Complexity : **$O(n^2)$**
- Space Complexity : **$O(1)$**

Output:

Enter the Limit 5
Enter the element
30 20 10 40 50
The sorted list of element is..
10
20
30
40
50

```
for(i=1;i<n-1;i++)
{
t=a[i];
j=i-1;
while( (j >=0) && (a[i]>t))
{
a[j+1]=a[j];
j=j-1;
}
a[j+1]=t;
}
printf("\n The sorted list of element is..\n");
for(i=0;i<n;i++)
printf("\n%d",a[i]);
}
```

5.2.4 SHELL SORT

Shell sort was invented by Donald Shell. It improves upon bubble and insertion sort by moving out of order element more than one position at a time. In shell sort the whole array is first fragmented into K segments, where K is preferably a prime number. After the first pass the whole array is partially sorted.

Example:

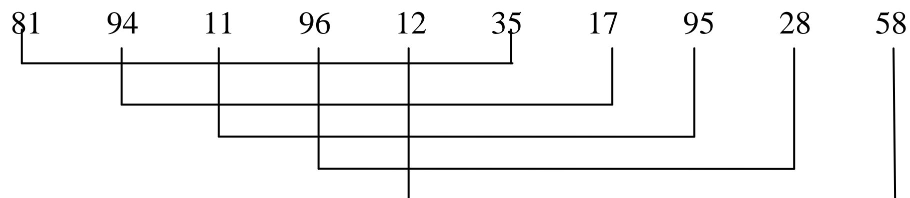
Consider an unsorted array as follows

81 94 11 96 12 35 17 95 28 58

Here N=10, the first pass as k=5 (10/2)

81 94 11 96 12 35 17 95 28 58

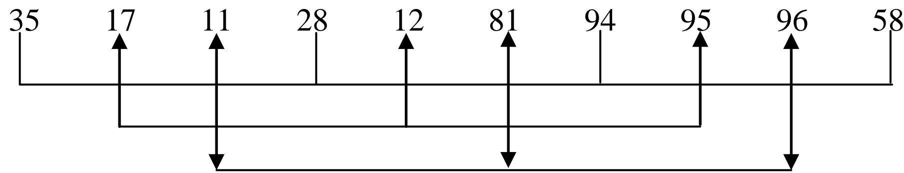
81 94 11 96 12 35 17 95 28 58



After First pass

35 17 11 28 12 81 94 95 96 58

In second pass, K is reduced to 3



After second pass

28 12 11 35 17 81 58 95 96 94

In third pass, k is reduced to 1

28 12 11 35 17 81 58 95 96 94

The final sorted array is

11 12 17 28 35 58 81 94 95 96

Routine:

```
void shellsort(int a[], int n)
{
int i,j,k,temp;
for(k=n/2;k>0;k=k/2)
for(i=k;i<n;i++)
{
temp=a[i]);
for(j=i;j>=k&&a[j-k]>temp;j=j-k)
{
a[j]=a[j-k];
}
a[j]=temp;
}
}
```

Advantage:

- Sorting a group of elements is more easy and faster than sorting the entire list.
- It is an in place sort.

Disadvantage:

- Extra overhead to divide the list into sub list.
- No universally accepted formula for calculating step size. It depends on the data to be sorted.

Program:

```
#include<stdio.h>
void shellsort(int a[10],int n);
void main()
{
```

Analysis of Shell Sort

- Best case analysis : **O(N log N)**
- Average case analysis : **$O(N^{1.5})$**
- Worst case Analysis : **$O(N^2)$**

```
int a[10],n,i;
printf("\n\t\t Enter the Limit");
scanf("%d",&n);
printf("\n\t\t Enter the element");
for(i=0;i<n;i++)
scanf("\n%d",&a[i]);
shellsort(a,n);
}
void shellsort(int a[], int n)
{
int I,j,k,temp;
for(k=n/2;k>0;k=k/2)
for(i=k;i<n;i++)
{
temp=a[i]);
for(j=i;j>=k&&a[j-k]>temp;j=j-k)
{
a[j]=a[j-k];
}
a[j]=temp;
}
printf("\n The sorted list of element is..\n");
for(i=0;i<n;i++)
printf("\n%d",a[i]);
}
```

Example 2: [Apr/May 2015]

35 12 14 9 15 45 32 95 40 5

After Pass1: 35, 12, 14, 9, 5, 45, 32, 95, 40, 15

After Pass2: 9, 5, 14, 32, 12, 40, 15, 95, 40, 15

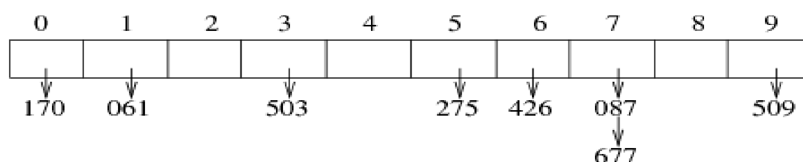
After Pass3: 9, 5, 12, 32, 14, 40, 15, 35, 45, 95

After Pass4: 5, 9, 12, 14, 15, 32, 35, 40, 45, 95

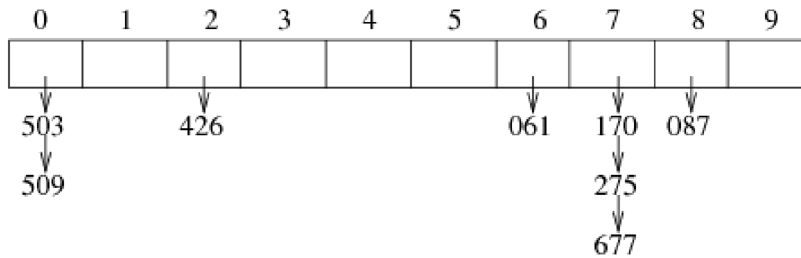
5.2.5 RADIX SORT

Radix sort is one of the linear sorting algorithms for integers. It works by sorting the input based on each digit. In first pass all the elements are sorted accordingly to the least significant digit. In second pass, the elements are arranged according to next least significant digit and so on till the most significant digit. The number of passes in a radix sort depends upon the number of digits in the given number.

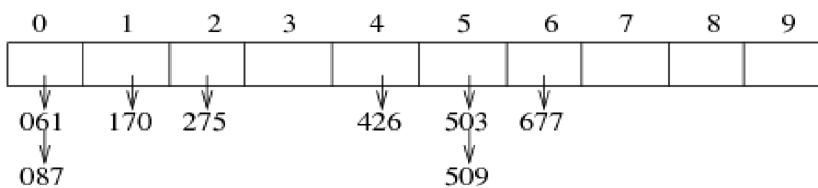
Example: 275, 087, 426, 061, 509, 170, 677, 503



2nd pass



3rd pass



in sorted order

ROUTINE:

```
int getMax(int arr[], int n)
{
   int mx = arr[0];
   for (int i = 1; i < n; i++)
      if (arr[i] > mx)
         mx = arr[i];
   return mx;
}

// A function to do counting sort of arr[] according to the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
   int output[n]; // output array
   int i, count[10] = {0};
    // Store count of occurrences in count[]
   for (i = 0; i < n; i++)
      count[ (arr[i]/exp)%10 ]++;
    // Change count[i] so that count[i] now contains actual position of this digit in output[]
   for (i = 1; i < 10; i++)
      count[i] += count[i - 1];
```

```
    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }
    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);
// Do counting sort for every digit. Note that instead of passing digit number, exp is passed.
//exp is 10^I  where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}


// A utility function to print an array
void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    printf("%d",arr[i]);
}
```

Advantage

- No comparison operation
- This appears to be better than quick sort for a wide range of input numbers.
- It is stable sort.

Disadvantage:

- The worst case and the average case all will take the same number of moves.
- Extra space needed for counter and temporary variables.

Time Complexity:

- Best case : O (1)
- Average case : O (N)
- Worst case : O (N)

5.3 HASHING

Basic Concept:

Hashing is an effective way to store the elements in some data structure. It allows to reduce the number of comparisons. Using the hashing technique we can obtain the concept of direct access of stored record. There are two concepts used regarding the hashing and those are – **hash table and hash function.**

5.3.1 Hash Table:

- Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the kae value. Hence every entry in the hash table is associated with some key. For example for storing an employee record in the hash table the employee ID will work as a key.
- Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table.
- The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

5.3.2 Hash Function:

- Hash Function is a function which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table.
- The integer returned by the hash function is called hash key.

For example:

- Consider that we want to place some employee record in the hash table. The record of employee is placed with the help of key: employee Id. The employee ID is a 7 digit number for placing the record in the hash table. To place the record, the key 7 digit number is converted into 3 digits by taking only last three digits of the key.
- If the key is 496700 it can be stored at 0th position. The second key is 8421002, the record of this key is placed at 2nd position in the array.

Hence the hash function will be

H(key)=key%1000.

Where key%1000 is a hash function and key obtained by hash function is called hash key.

The hash table will be

	Employee ID	Record
0	4968000	
1		
2	7421002	
4	.	.
396	44618396	
397	4957397	
	. . .	. . .
998		
999	0001999	

Bucket and home bucket: The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.
The function H(key) is home bucket for the dictionary with pair whose value is key.

Types of hash function:

There are various types of hash functions that are used to place the record in the hash table-

1. Division method: The hash function depends upon the remainder of division. Typically the divisor is table length. For example:

If the record 54, 72, 89,37 is to be placed in the hash table and if the table size is 10 then

H(key) = record % table size

4=54 % 10
2=72 % 10
9=89 % 10
7=37 % 10

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

2. Mid square: In the mid square method, the key is squared and the middle or mid part of the result is used as the index.
If the key is a string, it has to be preprocessed to produce a number
Consider that if we want to place a record 3111 then

3111^2 = 96**783**21

For the hash table of size 1000
H(3111) = 783 (the middle 3 digits)

3. Multiplicative hash function: The given record is multiplied by some constant value. The formula for computing the hash key is

H(key)=floor(p *(fractional part of key*A))

where p is integer constant and A is constant real number.

Donald knuth suggested to use constant A=0.61803398987

If key 107 and p = 50 then

H(key) = floor(50*107*0.61803398987)

=floor(3306.4818458045)

=3306

At 3306 location in the hash table the record 107 will be placed.

4. Digit folding: The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.

For example, consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together.

H(key) = 123+654+12

=0789

The record will be placed at location 789 in the hash table.

5. Radix Transformation: In this method, a key is transformed into another number base to obtain the hash value. For example:

Map the key $(8465)_{10}$ in the range 0 to 9999 using base 15.

$(8465)_{10} = (2795)_{15}$

The key 8465 is placed in the hash address 2795.

Characteristics of a good hash function:

- Easily computed.
- Key has to be distributed evenly in the hash table.
- Should avoid collision.

Application-Hash Table

- Database System.
- Data Dictionary.
- Browser Cache.

5.4 HASHING AND THE VARIOUS COLLISION RESOLUTION TECHNIQUES

COLLISION RESOLUTION TECHNIQUE

Situation in which, the hash function returns the same hash key for more than one record/key value is called Collision.

Eg: 132, 45, 44, 79, 20, 37, 58, 78 m=10

At 8th position Collision Occurs, 78 mod 10 =8

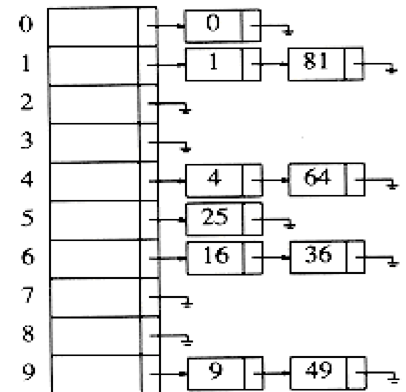
0	20
1	
2	132
3	
4	44
5	45
6	
7	37
8	58
9	79

There are various Collision Resolution Techniques:

- Separate Chaining/Open Hashing.
- Open Addressing/Closed Hashing.
- Rehashing.
- Extendible Hashing.

5.4.1 SEPARATE CHAINING/OPEN HASHING

- The first strategy, commonly known as either open hashing, or separate chaining, is to keep a list of all elements that hash to the same value.
- For convenience, our lists have headers. A pointer field is added to each record location, when an overflow occurs, this pointer is set to overflow blocks making a Linked list.
- Example: 0, 4, 1, 16, 25, 9, 36, 49, 64. 81 m=10



- To perform a find, we use the hash function to determine which list to traverse. We then traverse this list in the normal manner, returning the position where the item is found.
- To perform an insert, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match).
- If the element turns out to be new, it is inserted either at the front of the list or at the end of the list, whichever is easiest. This is an issue most easily addressed while the code is being written. Sometimes new elements are inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.

Advantages:

- More number of elements can be inserted than the table size.
- Simple.

Disadvantages:

- Requires pointers.

- Elements are not evenly distributed

5.4.2 OPEN ADDRESSING/CLOSED HASHING

- Alternative way to resolve collision.
- If a collision occurs, alternative cells are tried until an empty cell is found. Cells h0(x), h1(x), h2(x), … are tried in succession, where

hi(x)=((h(x)+f(i)) mod table_size where f(0)=0.

f ->collision resolution strategy

Three Types of Open Addressing

- Linear Probing.
- Quadratic Probing.
- Double Hashing.

1. Linear probing:

- Basically, Probing is a process of getting next available hash table array cell.
- Whenever collision occurs, cells are searched sequentially for an empty cell.
- When 2 record/key value has the same hash key, then 2nd record is placed linearly down (wrap around), whenever the empty index is found.

$$H_1(X) = (Hash\ (X) + i)\ mod\ Table_size.$$

Example: Insert 89, 18, 49, 58, 69 m=10

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

- In the above example, first collision occurs when 69 is inserted, which is placed in the next available spot namely spot 0 which is open.
- The next collision occurs when 71 is inserted, which is placed in the next available spot namely spot3.
- The collision for 33 is handled in a similar manner.

Advantage:

- IT doesn’t require pointers.

Disadvantages:

- Suffers from primary clustering.
- Increased search time.

2. Quadratic probing:

- Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. If Collision Occurs, new hash values is computed:
  Hash(X)=Keyvalue % Table_size
  H_i(X)=(Hash(X)+f(i)) mod Table_size
  where f(i)=i^2 such that i=1, 2,…., table_size-1.

Example:
H_0(X) = Hash(X) mod Table_size
H_1(X) = (Hash (X) + 1^2)mod Table_size
H_2(X) = (Hash (X) + 2^2)mode Table_size

H_0(89) = 89 % 10 =9
H_0(18) = 18 %10 =8
H_0(49) = 49 % 10 =9; Collide with 89

H_i(X)=(Hash (X) + i^2) mod Table_size
$H_{1(49)}$ = (9+1^2)%10 =0

$H_{0(58)}$ = (58)%10 =8 Collides with 18
$H_{1(58)}$ = (8+1^2)%10 =9 Collides with 89
$H_{2(58)}$ = (8+2^2)%10 =2

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

On the 1st collision, it looks ahead 1 position. On the 2nd collision, it looks ahead 4 position and 3rd, it looks ahead 9 position

Disadvantages:

- Key will not be inserted evenly.
- At most half of the table can be used as alternative locations to resolve collisions.
- It is difficult to find an empty slot, once the table is more than half full. This problem is known as secondary clustering because elements that hash to the same hash key will always probe the same alternative cells.

3. DOUBLE HASHING:

- Second hash function is applied to the key when a collision occurs.

By applying the 2nd hash function, we'll get the number of position from the point of collision to insert, where f(i)=i*h_d(X).

2 Rules:

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

$$H_d(X)=R-(X \bmod R)$$

Here R is a prime number that is smaller than the size of the table.

Insert: 89, 18, 49, 58, 69 using

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

[The first collision occurs when 49 is inserted. $h_2(49) = 7 - 0 = 7$, so 49 is inserted in position 6. h2(58) = 7 - 2 = 5, so 58 is inserted at location 3. Finally, 69 collides and is inserted at a distance h2(69) = 7 - 6 = 1 away.]

Hash(X) = X % 10

$Hash_2(X)$ = 7-(X % 7)

Hash (89) = 89 % 10 =9

Hash (18) = 18 % 10 =8

Hash (49) = 49 % 10 =9 Collides with 89

Therefore $Hash_2(49)$ = 7-(49 % 7) = 7

$H_i(49)=(Hash(49)+I*Hash_2(49)\%Tablesize)$

$H_1(49)$ = (9+1*7) mod 10

= 16 mod 10 = 6

Hash (58) = 58 % 10 = 8 Collides with 18

$Hash_2(58)$ = 7-(58 % 7) = 7 – 2 = 5

$H_i(58)=(Hash(58)+I*Hash_2(58)\%Tablelsize)$

$H_1(58)$ = (8+1*5) % 10

= 13 % 10 = 3

Hash (69) = 69 % 10 = 9 Collides with 89

$Hash_2(69)$ = 7-(69 % 7) = 7 – 6 = 1

$H_i(69)=(Hash(69)+I*Hash_2(69)\%Tablesize)$

$H_1(69)$ = (9+1*1) % 10

= 10 % 10

= 0

5.4.3 REHASHING

- When the hash table gets almost full, insert operation may take quite some time.
- Sometime collision resolution technique may fail to locate an empty slot.
- If the hash table gets full, then the rehashing method builds a new table (twice as big as old) just scans the keys of old hash table and compute the new hash value and insert into the new hash table.

0	6
1	
2	
3	3
4	4
5	5

Eg: 6, 5, 4, 6, 13 m=6

Insertion of 13 takes more time so we need to rehash

When m=10

0	
1	
2	
3	**3**
4	**4**
5	**5**
6	**6**
7	**13**
8	
9	

Advantages:

- Programmer doesn't worry about the table size.
- Simple to implement.

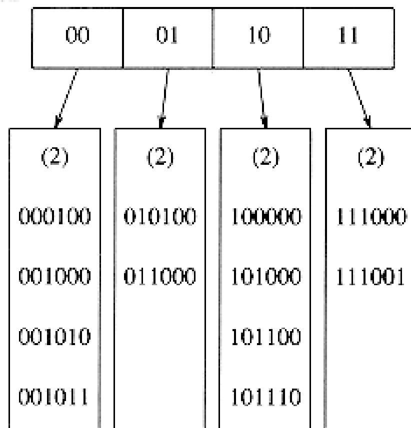
Drawbacks:

- Very Expensive.

5.4.4 EXTENDIBLE HASHING

- When open addressing/ separate chaining hashing is used, collisions could cause several blocks to be examined during a look-up. When hash table is too full, re-hashing must be performed [expensive].
- Hashing techniques discussed so far, when the data size is small. When data is bulky too many disk accesses in order to reduce the disk accesses we make use of any of extendible hashing.
- Suppose there are n records to place and we have m records space. We make use of any of collision technique, too many disk accesses. In extendible hashing, the information is maintained in the form of directory. Good for database that grows and shrinks in size. Allows the hash function to be modified dynamically.

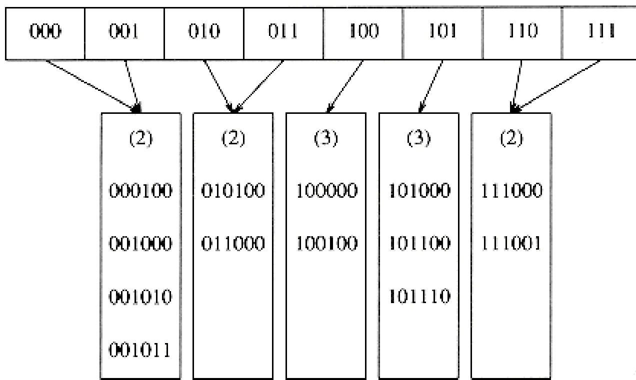
- Let us suppose, for the moment, that our data consists of several six-bit integers. Following Figure [Extendible hashing: original data] shows an extendible hashing scheme for this data.
- The root of the "tree" contains four pointers determined by the leading two bits of the data. Each leaf has up to m = 4 elements. It happens that in each leaf the first two bits are identical; this is indicated by the number in parentheses.



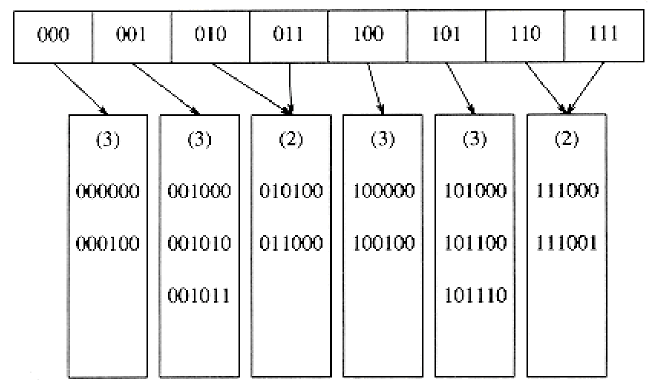
- To be more formal, D will represent the number of bits used by the root, which is sometimes known as the directory.
- The number of entries in the directory is thus 2D. dl is the number of leading bits that all the elements of some leaf 1 have in common. dl will depend on the particular leaf, and dl €D.

//Extendible hashing: original data

- Suppose that we want to insert the key 100100. This would go into the third leaf, but as the third leaf is already full, there is no room.
- We thus split this leaf into two leaves, which are now determined by the first three bits. This requires increasing the directory size to 3. These changes are reflected in Figure [after insertion of 100100 and directory split].
- Notice that all of the leaves not involved in the split are now pointed to by two adjacent directory entries. Thus, although an entire directory is rewritten, none of the other leaves are actually accessed.
- If the key 000000 is now inserted, then the first leaf is split, generating two leaves with dl = 3. Since D = 3, the only change required in the directory is the updating of the 000 and 001 pointers. See Figure [after insertion of 000000 and leaf split].
- This is very simple strategy provides quick access times for insert and find operations on large databases. There are a few important details we have not considered.
- First, it is possible that several directory splits will be required if the elements in a leaf agree in more than D + 1 leading bits. For instance, starting at the original example, with D = 2, if 111010, 111011, and finally 111100 are inserted, the directory size must be increased to 4 to distinguish between the five keys.
- This is an easy detail to take care of, but must not be forgotten. Second, there is the possibility of duplicate keys; if there are more than m duplicates, then this algorithm does not work at all. In this case, some other arrangements need to be made.
- These possibilities suggest that it is important for the bits to be fairly random. This can be accomplished by hashing the keys into a reasonably long integer; hence the reason for the name.



// Extendible hashing: after insertion of 100100 and directory split



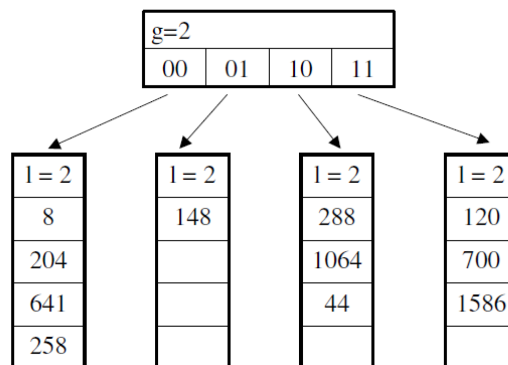
//Extendible hashing: after insertion of 000000 and leaf split

Extendible Hashing Example

- Suppose that g=2 and bucket size = 4.
- Suppose that we have records with these keys and hash function h(key) = key mod 64:

key	h(key) = key mod 64	bit pattern
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	50	110010
44	44	101010

Extendible Hashing Example – directory and bucket structure



- We have k bits to chosen in the header and number of entries at bucket will be 2k. When inserting a value, if the bucket is full after several splits (that is, i reaches some limit b) create an overflow bucket instead of splitting bucket entry table further.
- If suppose inserting : 00111, 00010 just split the 00 header into two as 000, 001.

Benefits of extendable hashing:

- Hash performance does not degrade with growth of file.
- Minimal space overhead.

Disadvantages of extendable hashing:

- Extra level of indirection to find desired record.
- Bucket address table may itself become very big (larger than memory).
- Need a tree structure to locate desired record in the structure!
- Changing size of bucket address table is an expensive operation.

5.5 APPLICATION OF HASHING

- In compilers to keep track of declared variables
- For online spelling checking the hashing function are used
- Hashing helps in Game Playing Programs to store the moves made
- For browser program while Caching the web pages, hashing is used

PART-A

SORTING

1. What is meant by sorting and searching? [L1]
2. **Give the types of sorting. Differentiate it. [Nov/Dec 2014] [Apr/May 2015]** [L1]
3. What are the types of sorting available in C? [L1]
4. **List the sorting algorithm which uses logarithmic time complexity. [May/Jun 2014]** [L1]

INSERTION SORTING

5. Define insertion sort. [L1]
6. What are the Advantages and disadvantages of insertion sort? [L1]
7. What is the time complexity of insertion sort? [L1]

SELECTION SORTING

8. Define selection sort. [L1]
9. What are the advantages and disadvantages of selection sort? [L1]

SHELL & BUBBLE SORTING

10. Define shell sort. [L1]
11. What is the time complexity of shell sort? [L1]

12. Define bubble sort. [L1]

RADIX SORTING

13. Define radix sort. Give its algorithm. [L1]

SEARCHING – LINEAR - BINARY

14. What are the types of searching? [L1]
15. Differentiate linear and binary search. [L1]
16. What is the time complexity of binary search? [May/Jun 2014] [L1]
17. Explain why binary search cannot be performed using linked list. [L2]
18. State the applications of linear and binary search technique. [Apr/May 2015] [L1]

HASHING - HASH FUNCTIONS

19. What is hashing? List its techniques. [L1]
20. What is hash function? [L1]
21. Define hash table. [L1]
22. What is overflow in hashing? [L1]

SEPARATE CHAINING – OPEN ADDRESSING - REHASHING – EXTENDIBLE HASHING

23. Define separate chaining. [L1]
24. What is open addressing? [L1]
25. Define probing. [L1]
26. What is rehashing? Give its advantages. [L1]
27. What is collision in hashing? [L1]
28. Define extendible hashing. [L1]

PART-B

SORTING

1. Write an algorithm to sort n numbers using insertion sort.**[Pg.No:147]** [L2]
2. Write an algorithm to sort n numbers using selection sort. **[Pg.No:144]** [L2]
3. Write an algorithm to sort n numbers using shell sort. [Apr/May-2015] **[Pg.No:149]** [L2]
4. Write an algorithm to sort n numbers using bubble sort. **[Pg.No:144]** [L2]

SEARCHING – LINEAR - BINARY

5. Explain in detail about linear search with example program. **[Pg.No:141]** [L2]
6. Write a C code to perform binary search. [Apr/May-2015] **[Pg.No:142]** [L2]

HASHING - HASH FUNCTIONS

7. Write short notes on hashing and the various types of hash function. [May/Jun -2014] **[Pg.No:154]** [L2]
8. Write short notes on hashing and the various collision resolution techniques. [May/Jun -2014] **[Pg.No:157]** [L2]
9. Explain the rehashing technique. [May/Jun -2014] **[Pg.No:160]** [L2]
10. Explain the extended hashing technique. **[Pg.No:161]** [L2]

PART-A

SORTING

1. What is meant by sorting and searching?

Sorting:

Sorting is a technique for arranging data in particular order. Order means the arrangement of data. The sorting order can be ascending or descending. The ascending order means arranging the data in increasing order where as descending order means arranging the data in decreasing order.

Searching:

Searching is the technique for finding particular element from the set.

2. Give the types of sorting. Differentiate it. [Nov/Dec 2014] [Apr/May 2015]

1. Internal Sorting	2. External Sorting
This is a type of sorting technique in which data resided on main memory of computer Example: Insertion Sort, Selection sort, shell sort, bubble sort, heap sort, quick sort, etc.	This is a sorting technique in which there is a huge amount of data and it resides on secondary devices while sorting. Example: Merge Sort, Multiway merge, Polyphase merge.

3. What are the types of sorting available in C?

- Insertion sort
- Merge sort
- Quick sort
- Radix sort
- Heap sort
- Selection sort

- Bubble sort.

4. **List the sorting algorithm which uses logarithmic time complexity. [May/Jun 2014]**
   Quick sort, merge sort, shell sort and heap sort has logarithmic time complexity

INSERTION SORTING

5. **Define insertion sort.**
   Successive element in the array to be sorted and inserted into its proper place with respect to the other already sorted element.
   We start with the second element and put it in its correct place, so that the first and second elements of the array are in order.

6. **What are the Advantages and disadvantages of insertion sort?**
   Advantage
   - Simple to implement
   - This method is efficient when we want to sort small number of elements and this method has excellent performance on almost sorted list of elements.
   - More efficient than most other simple O (n2) algorithms such as selection sort or bubble sort.
   - This is a stable.
   - It is called in-place sorting algorithm. The in-place sorting algorithm is an algorithm in which the input is overwritten by output and to execute the sorting method it does not require any more additional space.

   Disadvantage:
   - It is slow sorting algorithm
   - Shifting of elements is time consuming.
   - This sort works efficiently only in small lists.

7. **What is the time complexity of insertion sort?**
   - Best case : O (n)
   - Average case : O (n2)
   - Worst case : O (n^2)

SELECTION SORTING

8. **Define selection sort.**
   Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element scan the entire list to find the smallest element and swap it with the

second element. Then starting from the third element the entire list is scanned in order to find the next smallest element. Continuing in this fashion we can sort the entire list.

9. What are the advantages and disadvantages of selection sort?

Advantage:

- It performs well on a small list.
- It is an in-place sorting algorithm; so on additional temporary storage is required beyond what is needed to hold the original list.

Disadvantage:

- It is inefficient when dealing with a huge list of items.
- This requires n-squared number of steps for sorting n elements.
- The selection sort is only suitable for a list of few elements that are in random order.

SHELL & BUBBLE SORTING

10. Define shell sort.

In this method the elements at fixed distance are compared. The distance will then be decremented by some fixed amount and again the comparison will be made. Finally, individual elements will be compared. It uses an increment sequence. The increment size is reduced after each pass until increment size is 1.

11. What is the time complexity of shell sort?

- Best case : O (n log n)
- Average case : O ($n^{1.5}$)
- Worst case : O (n^2)

12. Define bubble sort.

Bubble sort is the one of the easiest sorting method. In this method each fate item is compared with its neighbor and if it is a descending order sorting, then the bigger element is moved to the top of all. The smaller numbers are slowly moved to the bottom position. Hence it is also called as the exchange sort.

RADIX SORTING

13. Define radix sort. Give its algorithm.

In radix sort the elements by processing its individual digits. It processes the digits either by least significant digit (LSD) method or by most significant digit (MSD) method.

Radix sort is a clever and intuitive little sorting algorithm; radix sort puts the elements in order by comparing the digits of the numbers.

SEARCHING – LINEAR - BINARY

14. What are the types of searching?

- Linear Search
- Binary Search

15. Differentiate linear and binary search.

S.No	Linear Search	Binary Search
1.	For searching the element by using linear search method it is not required to arrange the elements in some specific order	The elements need to be arranged either in ascending or descending order.
2.	Each and every element is compared with the key element from the beginning of the list.	The list is subdivided into two sub lists. The key element is searched in the sub list.
3.	Less efficient method.	Efficient method.
4.	Simple to implement.	Additional computation is required for computing mid element.

16. What is the time complexity of binary search? [May/Jun 2014]

- Best case : O (1)
- Average case : O (log N)
- Worst case : O (log N)

17. Explain why binary search cannot be performed using linked list.

In binary search algorithm, the mid element needs to be searched. If binary search is implemented using arrays then by simply saying a[mid] we can access the middle element of an array in constant time. But for finding the mid element of a linked list we have to execute separate algorithm and it cannot be done in constant time. Thus implementing binary search using linked list is very inefficient way. Hence it is not preferred to implement a binary search using linked list.

18. State the applications of linear and binary search technique. [Apr/May 2015]

Linear search is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an unordered list.

When many values have to be searched in the same list, it often pays to pre-process the list in order to use a faster method. For example, one may sort the list and use binary search,

or build any efficient search data structure from it. Should the content of the list change frequently, repeated re-organization may be more trouble than it is worth.

As a result, even though in theory other search algorithms may be faster than linear search (for instance binary search), in practice even on medium-sized arrays (around 100 items or less) it might be infeasible to use anything else. On larger arrays, it only makes sense to use other, faster search methods if the data is large enough, because the initial time to prepare (sort) the data is comparable to many linear searches

Application: Searching in a telephone directory. (Binary Search)

HASHING - HASH FUNCTIONS

19. What is hashing? List its techniques.

Hashing is a technique of storing the elements directly at the specific location in the hash table. The hashing makes use of hash function to place the record at its position. Using the same hash function the data can be retrieved directly from the hash table.

Techniques

- Division method
- Mid square method
- Multiplicative hash function.
- Digit folding
- Digit analysis.

20. What is hash function?

The mapping between a key and a bucket is called the hash function. A hash function is one which maps an element’s key into a valid hash table index.

21. Define hash table.

Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key. For example for storing an employee record in the hash table the employee ID will work as a key.

Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table.

The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

22. What is overflow in hashing?

The situation in which there is no room for a new pair in the hash table is called overflow.

SEPARATE CHAINING – OPEN ADDRESSING

23. Define separate chaining.

Separate chaining is a collection resolution strategy that maintains a linked list at every hash index for collided elements. This uses an array of linked lists as hash table.

24. What is open addressing?

Open addressing ensures that all element are stored directly into the hash table, thus it attempts to resolve collisions using various methods. In open addressing, the colliding elements are stored in other vacant buckets. This is a closed hashing technique.

25. Define probing.

Probing is the process of lookup and storage of the keys in hash table using open addressing.

REHASHING – EXTENDIBLE HASHING

26. What is rehashing? Give its advantages.

Rehashing is a technique in which the table is resized. That means the size of the table is doubled by creating a new table. The total size of the table is usually a prime number. Following are the situations in which the rehashing is required-

- When table is completely full.
- With quadratic probing the table is filled half.
- When insertions fail due to overflow.

Advantages:

- This technique provides the programmer a flexibility to enlarge the tale size if required.
- Only the space gets doubled with simple hash function which avoids occurrence of collisions.

27. What is collision in hashing?

The situation in which the hash function returns the same hash key for more than one record is called collision.

28. Define extendible hashing.

Extendible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits. In extendible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in parenthesis.

Example: