UNIT V
SEARCHING AND SORTING ALGORITHMS

Linear Search – Binary Search. Bubble Sort, Insertion sort – Merge sort – Quick sort – Hash tables – Overflow handling.

SEARCHING

- Linear Search
- Binary Search

LINEAR SEARCH

A linear search scans one item at a time, without jumping to any item.

```
#include<stdio.h>
int main( )
{
      int a[20],n,x,i,flag=0;
      printf("How many elements?");
      scanf("%d",&n);
       printf("\nEnter elements of the array\n");
      for(i=0;i<n; i++)
         scanf("%d",&a[i]);
      printf("\nEnter element to search:");
      scanf("%d",&x);
      for(i=0;i<n;i++)
      {
            if(a[i]==x)
          {
                flag=1;
                 break;
          }
      }
      if(flag)
         printf("\nElement is found at position %d ",i+1);
      else
         printf("\nElement not found");
      return 0;
}
```

The worst case complexity is O(n), sometimes known an O(n) search.

Time taken to search elements keep increasing as the number of elements are increased.

BINARY SEARCH

A binary search however, cut down your search to half as soon as you find middle of a sorted list. The middle element is looked to check if it is greater than or less than the value to be searched. Accordingly, search is done to either half of the given list.

```
#include<stdio.h>
int main()
{
      int n,i,a[100],f=0,l,h;
      printf("Enter the no. of Elements:");
```

```
        scanf("%d",&n);
        printf("\nEnter Elements of Array in Ascending order\n");
        for(i=0;i<n;++i)
        {
                scanf("%d",&a[i]);
        }
        printf("\nEnter element to search:");
        scanf("%d",&e);
        l=0;
        h=n-1;
        while(l<=h)
        {
                m=(l+h)/2;
                if(e==a[m])
                 {
                        f=1;
                        break;
          }
       else
          if(e>a[m])
             l=m+1;
          else
             h=m-1;
    }
   if(f==1)
       cout<<"\nElement found at position "<<m+1;
    else
      cout<<"\nElement is not found....!!!";
    return 0;
}
```

Differences

- Input data needs to be sorted in Binary Search and not in Linear Search
- Linear search does the sequential access whereas Binary search access data randomly.
- Time complexity of linear search -O(n) , Binary search has time complexity O(log n).
- Linear search performs equality comparisons and Binary search performs ordering comparisons.

SORTING

Sorting is linear ordering of list of items. Different types of sorting are

1. Bubble Sort
2. Insertion sort
3. Merge sort
4. Quick sort

BUBBLE SORT

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on. This process will be repeated for n-1 times.(n- total elements)

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

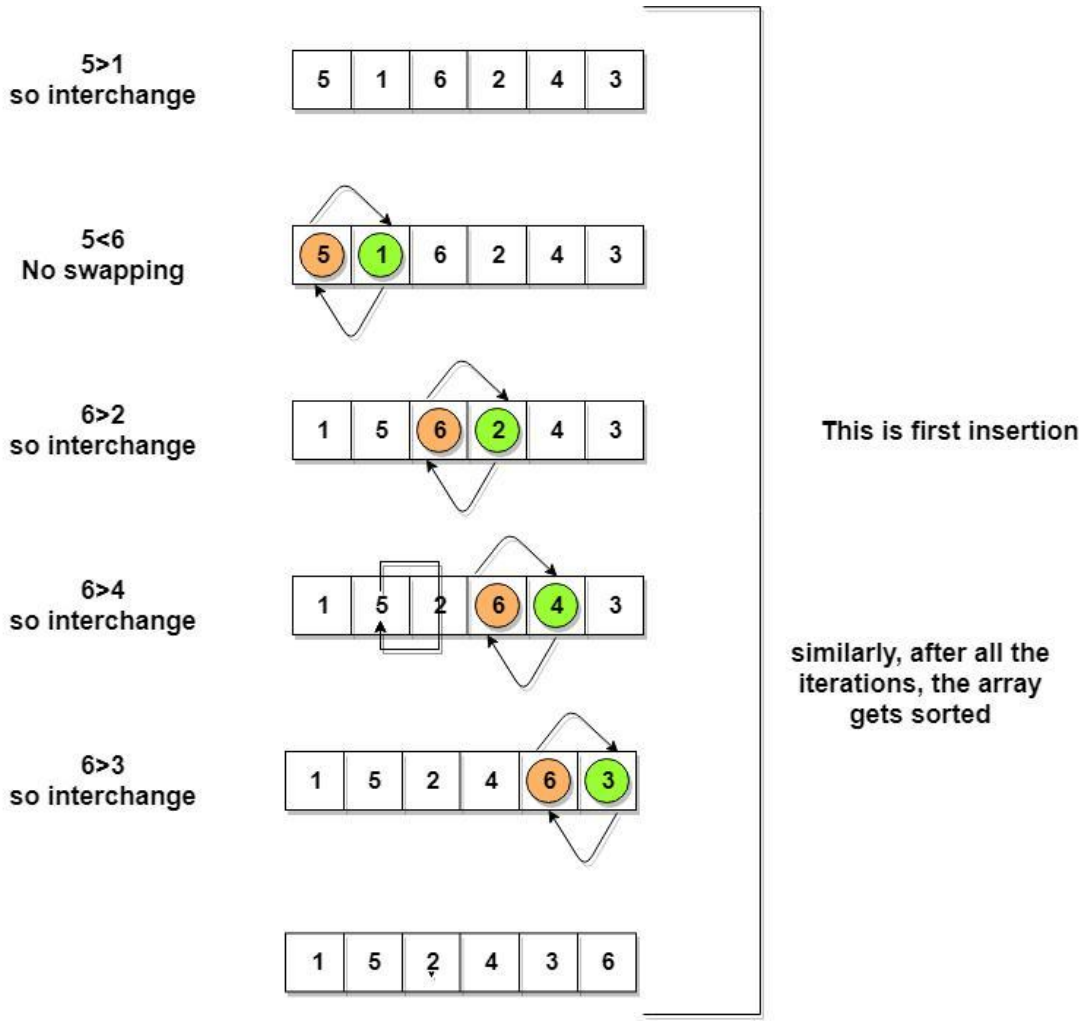
Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Bubble Sort:

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
   int i, j, temp;
   for(i = 0; i < n; i++)
   {
      for(j = 0; j < n-i-1; j++)
      {
         if( arr[j] > arr[j+1])
         {
            // swap the elements
            temp = arr[j];
            arr[j] = arr[j+1];
             arr[j+1] = temp;
         }
      }
   }
   // print the sorted array
   printf("Sorted Array: ");
   //  for(i = 0; i < n; i++)
   {
      printf("%d ", arr[i]);
   }
}

int main( )
{
   int arr[100], i, n, step, temp;
   printf("Enter the number of elements to be sorted: ");
   scanf("%d", &n);
   for(i = 0; i < n; i++)
   {
      printf("Enter elements\n");
      scanf("%d", &arr[i]);
   }
   // call the function
   bubbleSort bubbleSort(arr, n);
   return 0;
}
```

Example: Let's consider an array with values {5, 1, 6, 2, 4, 3}



As shown above, after the first iteration, 6 is placed at the last index, which is the correct position for it. Similarly after the second iteration, 5 will be at the second last index, and so on.

INSERTION SORT

1. 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.
2. It has one of the simplest implementation
3. It is efficient for smaller data sets, but very inefficient for larger lists.
4. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
5. It is better than Selection Sort and Bubble Sort algorithms.
6. Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.
7. 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...

INSERTION SORT ROUTINE

```
void insert(int a[],int n)
{
int i,j,temp;
for(i=0;i<n;i++)
{
temp=a[i];
for(j=0;j>0&&a[j-1]>temp;j--)
a[j]=a[j-1];
a[j]=temp;
}
```

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 : $O(1)$

QUICK SORT

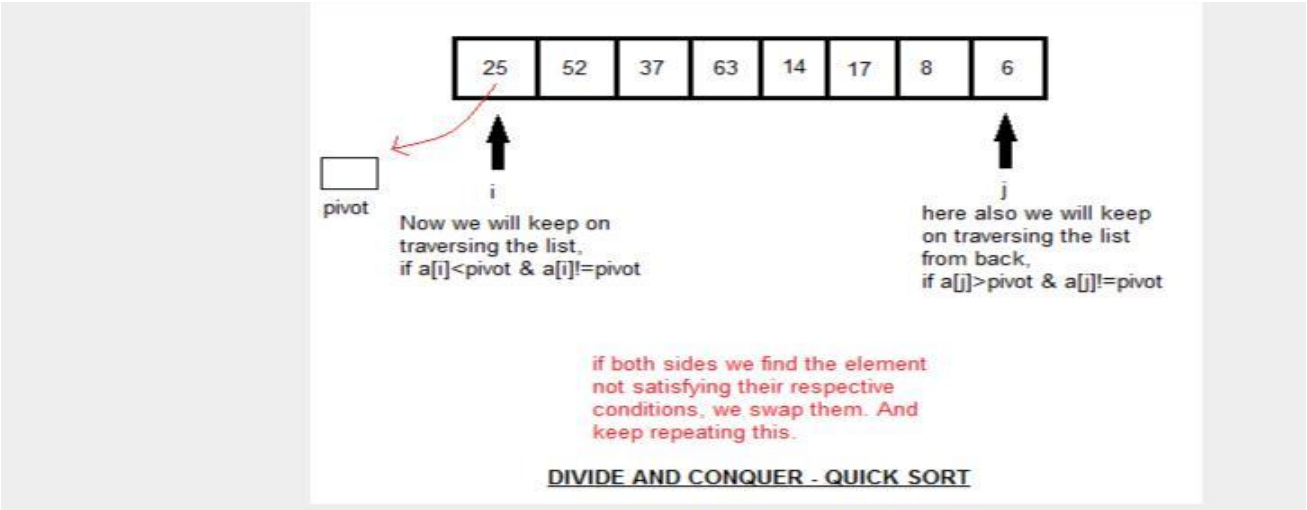
Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less aditional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 **25** 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.



How Quick Sorting Works

```
void qsort(int arr[], int left, int right)
{
    int i,j,pivot,tmp;
    if(left<right)

    {
        pivot=left;
        i=left+1;
        j=right;
        while(i<j)
        {
            while(arr[i]<=arr[pivot] && i<right)
                i++;
            while(arr[j]>arr[pivot])
                j--;
            if(i<j)
            {
                tmp=arr[i];
                arr[i]=arr[j];
                arr[j]=tmp;
            }
        }
```

```
            tmp=arr[pivot];
            arr[pivot]=arr[j];
            arr[j]=tmp;
            qsort(arr,left,j-1);
            qsort(arr,j+1,right);
        }
    }
```

Complexity Analysis of Quick Sort

Worst Case Time Complexity : $O(n^2)$
Best Case Time Complexity : O(n log n)
Average Time Complexity : O(n log n)
Space Complexity : O(n log n)

Space required by quick sort is very less, only O(n log n) additional space is required.

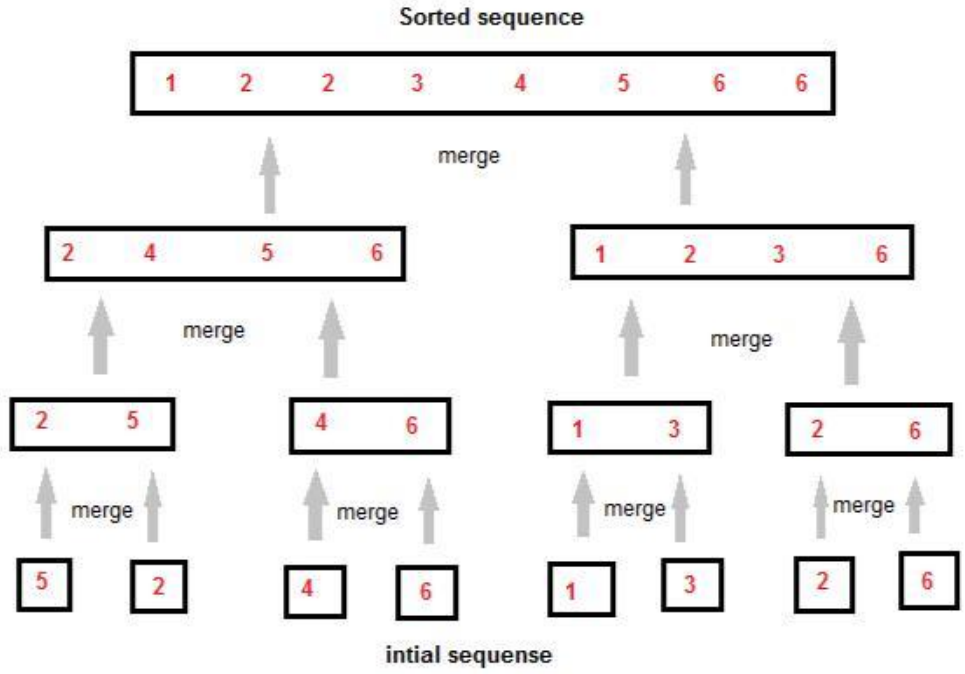
Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

MERGE SORT

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divides the list into two halves. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

How Merge Sort Works



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, and then keep merging these sublists, to finlly get the complete sorted list.

Merge Sort Algorithm

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


void merge(int [],int ,int ,int );
void part(int [],int ,int );

int main( )
{
        int arr[30];
        int i,size;
        printf("\n\t------- Merge sorting method -------\n\n");
        printf("Enter total no. of elements : ");
        scanf("%d",&size);
        for(i=0; i<size; i++)
        {
                printf("Enter %d element : ",i+1);
                scanf("%d",&arr[i]);
        }
        part(arr,0,size-1);
        printf("\n\t-------     Merge sorted elements                                   -------\n\n");
        for(i=0; i<size; i++)
        printf("%d ",arr[i]);
        getch();

        return 0;
 }
void part(int arr[],int min,int max)
{
        int mid;
        if(min<max)
        {
                mid=(min+max)/2;
                part(arr,min,mid);
                part(arr,mid+1,max);
                merge(arr,min,mid,max);
        }
}

void merge(int arr[],int min,int mid,int max)
{
        int tmp[30];
        int i,j,k,m;
        i=min;
        j=mid+1;k=0;
        while(i<=mid && j<=max )
        {
                if(arr[i]<=arr[j])
                        tmp[k++]=arr[i++];
                else
```

```
                tmp[k++]=arr[j++];
        }
    while(i<=mid)
            tmp[k++]=arr[i++];
    while(j<=max)
            tmp[k++]=arr[j++];


    for(k=min; k<=max; k++)
        arr[k]=tmp[k];
}
```

Complexity Analysis of Merge Sort

Worst Case Time Complexity	: O(n log n)
Best Case Time Complexity	: O(n log n)
Average Time Complexity	: O(n log n)
Space Complexity	: O(n)

Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves. It requires equal amount of additional space as the unsorted list. Hence it is not at all recommended for searching large unsorted lists. It is the best Sorting technique for sorting Linked Lists.

HASHING

Hashing

Hashing is a process which uses a function to get the key and using the key it quickly identifies the record, without much strain. The values returned by a hash function are called hash values. Hash table is data structure in which key values are place in array location

Hash Table

The hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

There are two types of hashing :

1. ***Static hashing***: In static hashing, the hash function maps search-key values to a fixed set of *locations*.
2. ***Dynamic hashing***: In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

A ***hash function***, **h**, is a function which transforms a key from a set, **K**, into an index in a table of size n:

h: K -> {0, 1, ..., n-2, n-1}

- A key can be a number, a string, a record etc.The size of the set of keys, |**K**|, to be relatively very large. It is possible for different keys to hash to the same array location.This situation are called *collision* and the colliding keys are called *synonyms*.

A good hash function should:

- *Minimize* collisions.
- Be *easy* and *quick* to compute.
- Distribute key values *evenly* in the hash table.
- Use *all the information* provided in the key.

Various types of Hash Functions:

Type 1: Truncation Method
Type 2: Folding Method
Type 3: Midsquare Method

Type 4: Division Method (Modulo Division)

1 Truncation Method

The Truncation Method truncates a part of the given keys, depending upon the size of the hash table.

1. Choose the hash table size.

2. Then the respective right most or left most digits are truncated and used as hash code| value.

Ex: 123,42,56 Table size = 9

H(123)=1

H(42)=4

H(56)=5

10

0	
1	123
2	
3	
4	42
5	56
6	
7	
8	
9	

2 Mid square Method :

It is a Hash Function method.

1. Square the given keys.
2. Take the respective middle digits from each squared value and use that as the hash value | address | index | code, for the respective keys.

H(123)=1 [123^2 = 15129]

H(42)=7 [42^2 = 17 64]

H(56)=3 [56^2 = 3136]

0	
1	123
2	
3	56
4	
5	
6	
7	42
8	
9	

3 Folding Method:

Partition the key K into number of parts, like K1,K2,.....Kn, then add the parts together and ignore the carry and use it as the hash value.

H(123)= [1+2+3 =6]

H(43)= [4+3 = 7]

H(56)= [5+6 = 11]

0	
1	56
2	
3	
4	
5	
6	123
7	43

4 Division Method :

Choose a number m, larger than the number of keys. The number m is usually chosen to be a prime number. The formula for the division method :

Hash(key)= key % tablesize

Tablesize : 10 20,21,24,26,32,34

H(20)= 20 %
10 = 0

H(21)= 21 %
10 = 1

H(24)=24 %
10 = 4

H(26)= 26 % 10 = 6

H(32) = 32 % 10 = 2

H(34)=34%10=4 34 IS COLLISION

0	20
1	21
2	32
3	
4	24
5	
6	26
7	42
8	
9	

Applications

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.
- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.
- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.
- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.
- **Browser Cashes**: Hash tables are used to implement browser cashes.

HASH COLLISION:

A hash collision or hash clash is a situation that occurs when two distinct inputs into a hash function produce identical outputs.

COLLISION RESOLUTION TECHNIQUES

The techniques are:

Closed Addressing

1. Separate Chaining.

Open Addressing

2. Linear Probing.
3. Quadratic Probing.
4. Double Hashing.

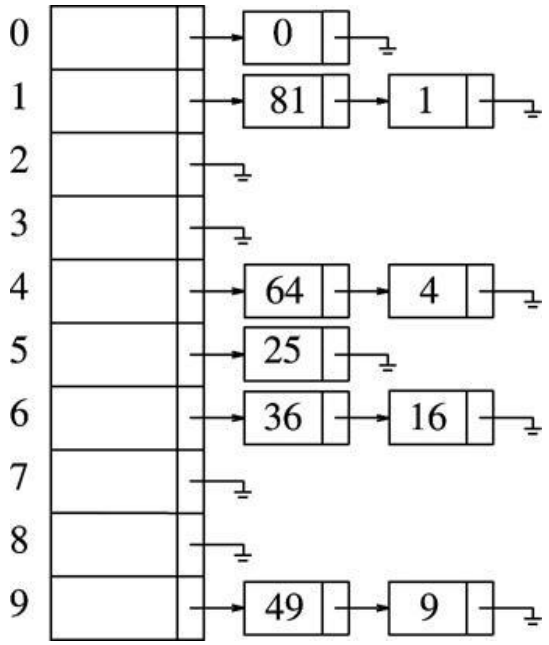
Dynamic Hashing

5. Rehashing.
6. Extendable Hashing.
7. Linear Hashing.

1. Separate Chaining

It is to keep a list of all elements that hash to the same value. An alternative to open addressing as a method of collision resolution is **separate chaining** hashing. This uses an array as the primary hash table, except that the array is an array of **lists** of entries, each list initially being empty.

If in a set of elements, if an element hashes to the same value as that of an already inserted element then we have a collision and We need to resolve it. In separate chaining ,each slot of the bucket array is a pointer to a linked list that contains key-value pairs that are hashed to the same location. It is otherwise called as direct chaining or simply chaining. An array of linked list implementation is used here.



SEPARATE CHAINING PROCEDURE :

TO FIND AN ELEMENT

- To perform a Find, we use the hash function to determine which list to traverse.
- We then traverse it in normal manner, returning the position where the item is found.
- Finding an element in separate chaining is very much similar to the find operation performed in the case of lists.

If the Element Type is a string then a comparison and assignment must be done with *strcmp* and *strcpy* respectively.

TO INSERT AN ELEMENT

- To perform an Insert, we traverse down the appropriate list to check whether the element is already in place .
- If it is new then it is either inserted at the front or at the end.

- If the item to be inserted is already present, then we do not perform any operation; otherwise we place it at the front of the list. It is similar to the insertion Operation that takes place in the case of linked lists.
- The disadvantage is that it computes the hash function twice.

TO DELETE AN ELEMENT:

- To delete we find the cell P prior to the one containing the element to be deleted.
- Make the cell P to point to the next cell of the element to be deleted.
- Then free the memory space of the element to be deleted.

PROGRAM

HEADER FILE FOR SEPARATE CHAINING

```
Typedef int elementtype;
Typedef struct listnode *position;
Typedef struct hashtbl *hashtable;
Typedef position list;
Struct hashtbl
{
        int tablesize;
        list *thelists;
};
```

IMPLEMENTATION OF SEPARATE CHAINING

The Lists will be an array of list.

```
Struct listnode
{
        Elementtype element;
        Position next;
};
Hashtable initialize table(int tablesize)
{
        Hashtable H;
        int i;
/ *Allocate Table * /
        H=malloc(sizeof(struct hashtable));
        If(H==NULL)
                Fatalerror("Out of Space");
                H-->tablesize=nextprime(tablesize);
/*Allocate array of list */
                H-->thelist=malloc(sizeof(list)*H-->tablesize);
        If(H-->thelist==NULL)
                Fatalerror("Out of Space");
/*Allocate list header */
                For(i=0;i<H-->tablesize;i++)
                {
                        H-->thelists[i]=malloc(sizeof(struct listnode));
        If(H-->thelists[i]==NULL)
              Fatalerror("Out of Space");
              Else
              H-->thelists[i]-->next=NULL;
              }
              return H;
              }
```

```
Hash(char *key, int tablesize)
{
int hashvalue=0;


  while(*key!='\0')
  hashvalue=hashvalue+ *key++;
  return(hashvalue % tablesize);
  }
  Position find(Elementtype key, hashtable H)
  {
  Position P;
  List L;
  L=H-->thelists(Hash(key,H-->tablesize));
  P=L-->next;
  While(P!=NULL && P-->element ! =key)
  P=P-->next;
  Return P;
  }
  void insert(elementtype key, hashtable H)
  {
  Position pos,newcell;
  List L;
  Pos=find(key,H);
  If(pos==NULL)
  {
  Newcell=malloc(sizeof(struct listnode));
  If(newcell==NULL)
  Fatalerror("Out of Space");
  else
  {
  L=H-->thelists(hash(key,H-->tablesize));
  Newcell-->next=L-->next;
  Newcell-->element=key;
  L-->next=Newcell;
  }
  }
  }
```

ADVANTAGES:

- Separate chaining is used when memory space is a concern.
- It can be very easily implemented.

DISADVANTAGES:

- It requires pointers which causes the algorithm to slow down a bit.
- Unevenly distributed keys-long lists-search time increases.

Open Addressing

In an open addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found.

A bigger table is needed for open addressing hashing, than for separate chaining.

Types of Open Addressing :
1. Linear Probing.
2. Quadratic Probing.
3. Double Hashing.

Linear Probing

It is a kind of Open Addressing. In Linear probing, F is a linear function of i, typically F(i)=i. In linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found.

If the end of the table is reached and no empty cell have been found, then the search is continued from the beginning of the table. It has a tendency to create cluster in the table.

In linear probing we get primary clustering problem. Primary Clustering Problem

If the Hashtable becomes half full and if a collision occurs, it is difficult to find an empty location in the hash table and hence an insertion or the deletion process takes a longer time. Hash function

hi(key)=(Hash(key)+ F(i)) % Tablesize F(i)=i Hash(key)=key % tablesize

Ex: 89,18,49,58,69

-> Hashtable

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	

1.Hash(89)=89 % 10=9

2.Hash(18)=18 % 10= 8

3.Hash(49)=49 % 10=9 -> collision occurs for first time i=1 h1(49)= (9+1) % 10=10 % 10 =0

4.Hash(58)= 58 % 10 =8 -> collision occurs for first time i=1 h1(58)= (8+1) % 10 =9 -> collision occurs for first time i=2 h2(58)= (8+2) % 10 =0 -> collision occurs for first time i=3 h3(58)= (8+3) % 10 =1

5.Hash(69)=69%10=9 -> collision occurs for first time i=1 h1(69)= (9+1) % 10 =9 -> collision occurs for first time i=2 h2(69)= (9+2) % 10 =1 -> collision occurs for first time i=3 h3(69)= (9+3) % 10 =2

Advantage:

It does not require pointers.

Disadvantage:

It forms clusters, which degrades the performance of the hash table for storing and retrieving data.

Quadratic Probing

It is a kind of open addressing technique.It is a collision resolution method that eliminates the primary clustering problem of linear probing.

-> Hash function

hi(key)=(Hash(key)+ F(i)) % Tablesize

F(i)=i2 Hash(key)=key % tablesize

-> Hashtable Ex: 89,18,49,58,69

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

1.Hash(89)=89 % 10=9
2.Hash(18)=18 % 10= 8
3.Hash(49)=49 % 10=9 -> collision occurs for first time
i=1 h1(49)= (9+12) % 10=10 % 10 =0
4.Hash(58)= 58 % 10 =8 -> collision occurs for first time
i=1 h1(58)= (8+12) % 10 =9 -> collision occurs for first time i=2 h2(58)= (8+22) % 10 =2
5.Hash(69)=69%10=9 -> collision occurs for first time i=1
h1(69)= (9+12) % 10 =9 -> collision occurs for first time
i=2 h2(69)= (9+22) % 10 =3
Secondary Clustering:
Elements that Hash to the same position will probe the same alternative cells. This is known as secondary clustering.

Double Hashing

Double hashing is a technique which belongs to open addressing. Open addressing is a collision resolution technique which uses the concept of linked lists. Open addressing handles collision by trying out all the alternative cells until an empty cell is found. Collision is said to have occurred if there exists this situation. i.e., If an element inserted hashes to the same value as that of an already inserted element, then there is collision PROCEDURE:

- Compute the positions where the data elements are to be inserted by applying the first hash function to it.
- Insert the elements if the positions are vacant.
- If there is collision then apply the second hash function.
- Add the two values and insert the element into the appropriate position.
- Number of probes for the data element is 1 if it is inserted after calculating first hash function.
- Number of probes is 2 if it is inserted after calculating second hash function.

DEFINITION:
It is a collision resolution technique which uses two hash functions to handle collision.
The interval (i.e., the distance it probes) is decided by the second hash function which is independent.

REQUIREMENTS:

The table size should be chosen in such a way that it is prime so that all the cells can be inserted. The second hash function should be chosen in such a way that the function does not evaluate to zero. i.e., 1 can be added to the hash function(non-zero). To overcome secondary clustering, double hashing is used. The collision function is,

hi(key)=(Hash(key)+ F(i)) % Tablesize
F(i) = i * hash2 (X)

Where **hash2(X) = R – (X % R)** R is a prime number. It should be smaller than the table size

Example 1 :

89, 18, 49, 58, 69,60

0	69
1	
2	60
3	58
4	
5	
6	49
7	18
8	89
9	

49

h0(49)=9

h1(49) = (9+(1*7)) % 10 = (9+7) % 10 = 16 % 10 = 6

58

7 – (58 % 7) = 7 – 2 = 5

i=1

h1(58) = (8+(1*5)) % 10 = 13 % 10 = 3

69

7 – (69 % 7) = 7 – 6 = 1

h1(69) = (9+1) % 10 = 10 % 10 = 0

60

7 – (60 % 7) = 7 – 4 = 3

h1(60) = (0+1*3) % 10 = 3

h2(60) = (0+2*3) % 10 = 6

h3(60) = (0+9) % 10 = 9

h4(60)=(0+12)%10=2

APPLICATIONS:

- It is used in caches.
- It is used in finding duplicate records.
- Used in finding similar records.
- Finding similar substrings.

ADVANTAGES:

- No index storage is required.
- It provides rapid updates.

DISADVANTAGES:

- Problem arises when the keys are too small.
- There is no sequential retrieval of keys.
- Insertion is more random.

Re-Hashing:

It is a technique in which the table is re-sized i.e., the size of the table is doubled by creating a new table. Rehashing is a technique that is used to improve the efficiency of the closed hashing techniques. This can be done by reducing the running time. If the table gets too full, the running time for the operations will start taking too long and *inserts* might fail for closed hashing with quadratic resolution. This can happen if there are too many deletions intermixed with insertions. A solution, then, is to build another table that is about twice as big (with associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new tableRe-Hashing is required when, The table is completely filled in the case of double-Hashing. The table is half-filled in the case of quadratic and linear probing The insertion fails due to overflow.

IMPLEMENTATION:

Rehashing can be implemented in

1.Linear probing

2.Double hashing

3.Quadratic probing

- Rehashing can be implemented in several ways with quadratic probing.
- One alternative is to rehash as soon as the table is half full.
- The other extreme is to rehash only when an insertion fails.
- A third, middle of the road, strategy is to rehash when the table reaches a certain load factor.Since performance does degrade as the load factor increases, the third strategy, implemented with a good cutoff, could be best.

Problems:

1. According to linear probing, 13, 15, 6, 24, 23

0	6
1	15
2	23
3	24
4	
5	
6	13

Rehashing

Size=7 double size=14 next prime =17

1. 16 % 17 = 6
2. 15 % 17 = 15
3. 23 % 17 = 6

h1(X) = 6 + 1 % 17 = 7

4. 24 % 17 = 7

h1(24) = 7 + 1 % 17 = 8

4. 13 % 17 = 13

0	
1	
2	
3	
4	
5	
6	1
7	6
8	23
9	24
10	
11	
12	
13	
13	13
14	
15	15
16	

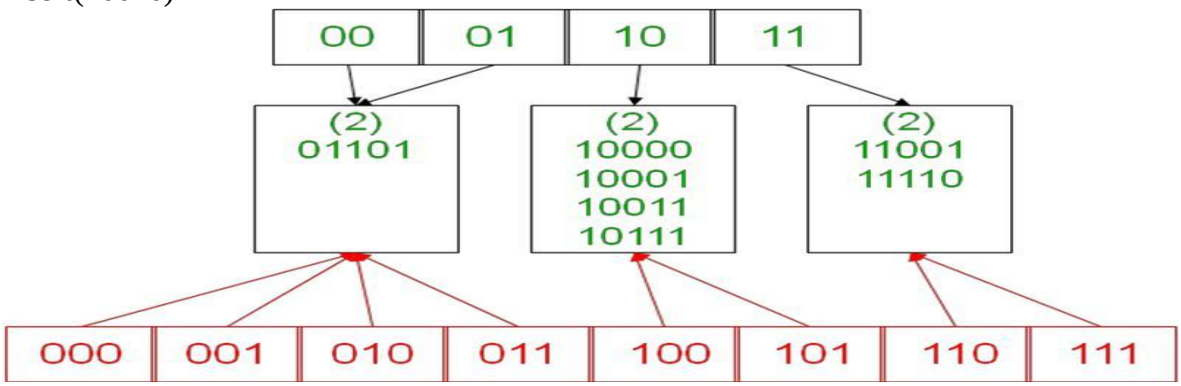
Advantage:
This technique provides the programmer the flexibility to enlarge the table size if required.
Disadvantages:
Transfer time is more.

Extensible Hashing
Hashing technique for huge data sets
- optimizes to reduce disk accesses
- each hash bucket fits on one disk block
- better than B-Trees if order is not important
- Table contains
  - buckets, each fitting in one disk block, with the data
  - a directory that fits in one disk block used to hash to the correct bucket

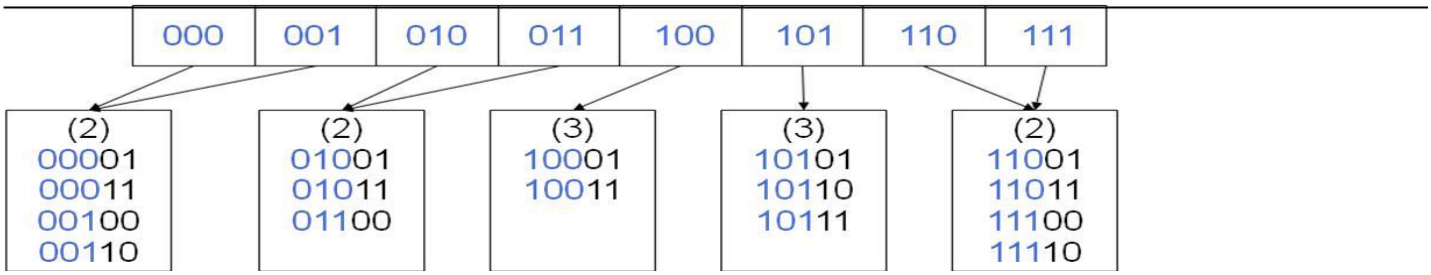
- Directory - entries labeled by *k* bits & pointer to bucket with all keys starting with its bits
- Each block contains keys & data matching on the first *j* □ *k* bits

1. **Advantages:**
   - Extendable hashing provides performance that does not degrade as the file grows.
   - Minimal space overhead - no buckets need be reserved for future use. Bucket address table only contains one pointer for each hash value of current prefix length.
2. **Disadvantages:**
   - Extra level of indirection in the bucket address table
   - Added complexity

- insert(10010)



directory for $k = 3$



insert(11011)