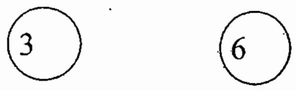
Graphs

Definitions

Vertex – It is a synonym for a node and is generally represented by a circle.
Example:



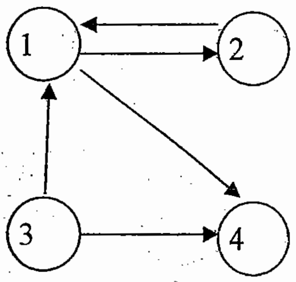
Edge – If 'u' and 'v' are two vertices, then an arc or a line joining the two vertices 'u' and 'v' is called an edge.
Example:

Undirected Edge – An edge without any direction is called an undirected edge and is denoted by an ordered pair (u, v).
Example: (3, 6)

Directed Edge – An edge with a direction is called an directed edge and is denoted by an directed pair <u, v>.
Example: <4, 7>

Graph – A graph G consists of two sets V and E and is denoted as G = (V, E), where,
V is finite, nonempty set of vertices
E is finite set of edges.

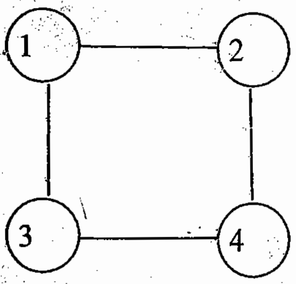
Directed Graph – A graph G = (V, E) is called a directed graph if all the edges are directed. It is also called as a digraph.
Example:



Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

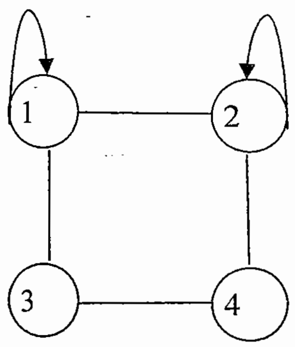
The graph G = (V, E), where
V = {1, 2, 3, 4}
E = { <1,2>, <1,4>, <2,1>, <3,1>, <3,4> }

Undirected Graph – A graph G = (V, E) is called a undirected graph if all the edges are undirected.
Example:



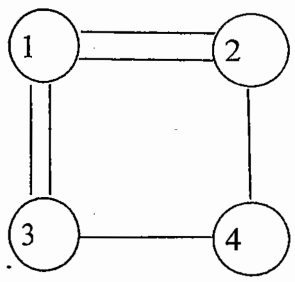
The graph G = (V, E), where
V = {1, 2, 3, 4}
E = { (1,2), (2,4), (1,3), (3,4)}

Self Loop – A loop is an edge which starts and ends on the same vertex. It is represented by an ordered pair (i,i).
Example:



In the above graph the self loops are (1, 1) and (2, 2)

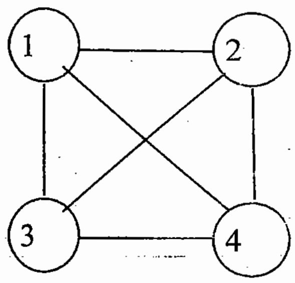
Multigraph – A graph with multiple occurrence of the same edge between any two vertices is called as a multigraph.
Example:



Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

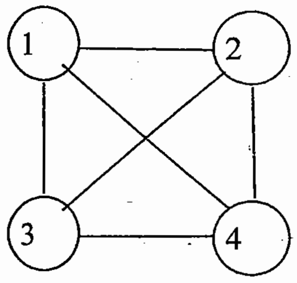
In the above graph, (1,2) and (1,3) are the multiple edges.

Complete Graph – A graph G = (V,E) is said to be a complete graph, if there exists an edge between every pair of vertices.
Example:

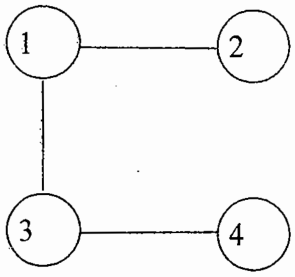


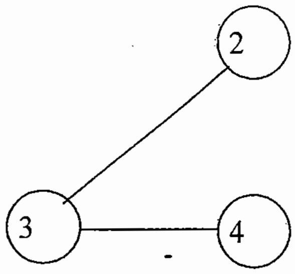
Note: In a complete graph with 'n' vertices there will be n(n-1)/2 edges.

Subgraph – A subgraph G' of a graph G is graph such that V(G') ⊆ V(G) and E(G') ⊆ E(G).
Example: Consider the graph G

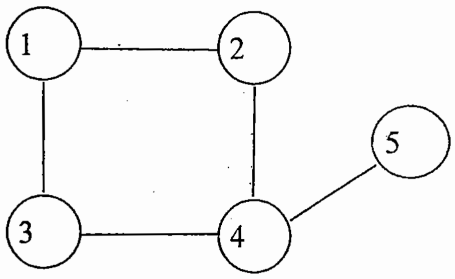


Some of the subgraphs of G are





Path – A path from a vertex 'u' to a vertex 'v' in a graph G is a sequence of vertices u, i_1, i_2,........i_k, v such that (i_1, i_2), (i_2, i_3), ...(i_k, v) are edges in E(G).
Example: Consider the graph shown below



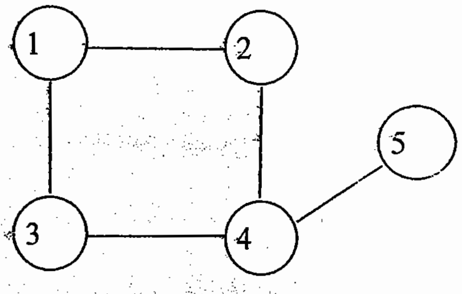
A path from vertex 1 to vertex 5 is given by 1, 3, 4, 5

Length of the path – It is the number of edges in the path.
Example: In the above graph, the length of the path from vertex 1 to vertex 5 is 3.

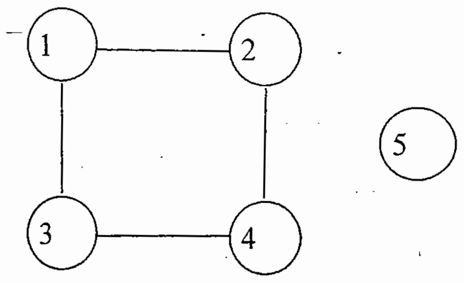
Cycle – A cycle is a simple path in which the first and last vertices are the same.
Example: In the above graph the path 1 2 4 3 1 represents a cycle.

Note: A graph with atleast one cycle is called as a cyclic graph and a graph with no cycles is called as an acyclic graph.

Connected Graph – A graph G is said to be a connected graph if and only if there exists a path between every pair of vertices
Example: The following is a connected graph

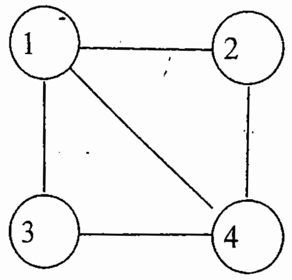


Disconnected Graph – A graph G is said to be a disconnected graph if there exists atleast one vertex in the graph such that it cannot be reached from any of the other vertices in the graph.
Example: The following is a disconnected graph



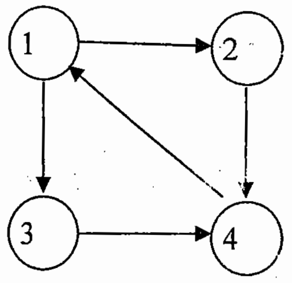
Degree, In-degree and Out-degree
The degree of a vertex is the number of edges incident to that vertex.

Example: Consider the graph G



Degree of vertex 1 is 3
Degree of vertex 2 is 2
Degree of vertex 3 is 2
Degree of vertex 4 is 3

If G is a directed graph, the In-degree of a vertex 'v' is the number of edges for which 'v' is the head and Out-degree of vertex 'v' is the number of edges for which 'v' is the tail.
Example: Consider the graph G



In-degree of vertex 1 is 1
Out-degree of vertex 1 is 2
Degree of vertex 1 is 3

Graph Representations

The following are the different representations of a graph

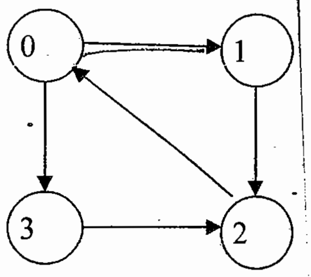
Adjacency Matrix

Let G=(V,E) be a graph, where V is the set of vertices and E is the set of edges. The adjacency matrix for the graph G with 'n' vertices is a "n x n" two dimensional array a[ ][ ] such that,

a[i][j] = 1 if there is an edge from vertex 'i' to vertex 'j' and
a[i][j] = 0 if there is no edge from vertex 'i' to vertex 'j'.

Example:

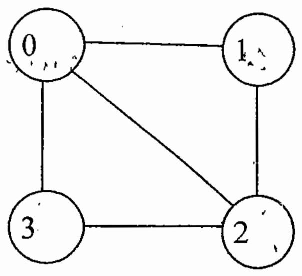
The following is a directed graph and its adjacency matrix



	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

The following is a undirected graph and its adjacency matrix



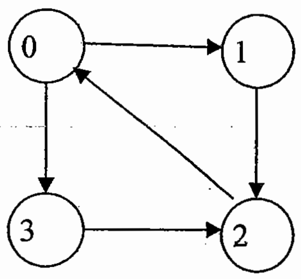
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

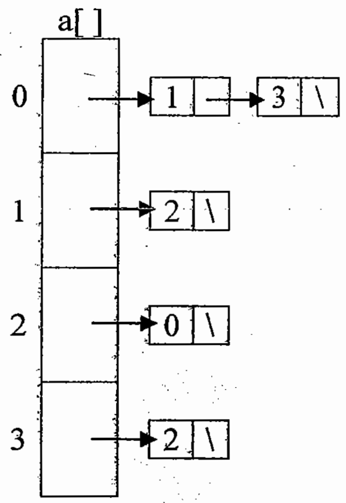
Adjacency Lists

Let G = (V,E) be a graph. An adjacency list is an array of 'n' linked lists where 'n' is the number of vertices in the graph G. Each location of the array represents a vertex of the graph.

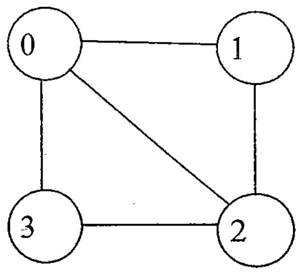
Example:

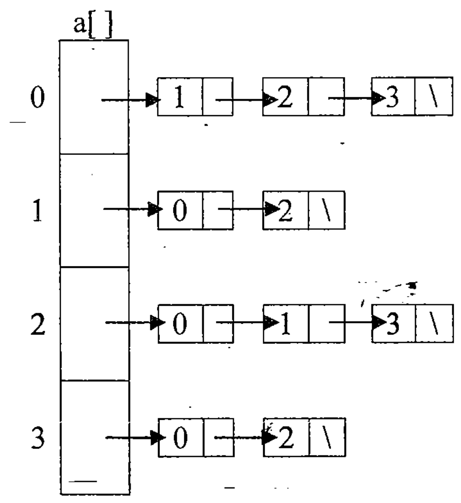
The following is a directed graph and its adjacency list





The following is a undirected graph and its adjacency list



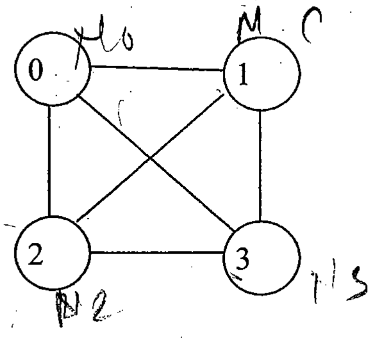


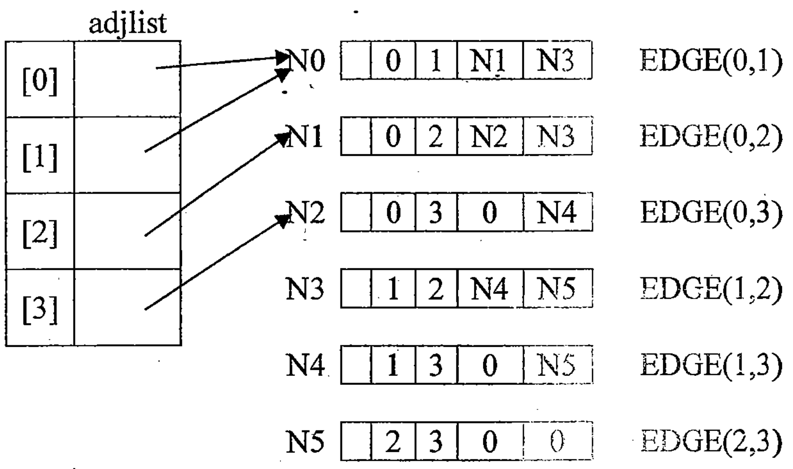
Adjacency Multilists

In a adjacency list representation of a undirected graph, each edge (u,v) is represented by two entries, one on the list for 'u' and the other on the list for 'v'. This can be avoided easily if adjacency list are actually maintained as multilists (i.e. lists in which nodes may be shared among several lists)

Here for each edge there will be exactly one node, but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident)

Example: The following is a undirected graph and its adjacency multilist





N0	0	1	N1	N3	EDGE(0,1)
N1	0	2	N2	N3	EDGE(0,2)
N2	0	3	0	N4	EDGE(0,3)
N3	1	2	N4	N5	EDGE(1,2)
N4	1	3	0	N5	EDGE(1,3)
N5	2	3	0	0	EDGE(2,3)

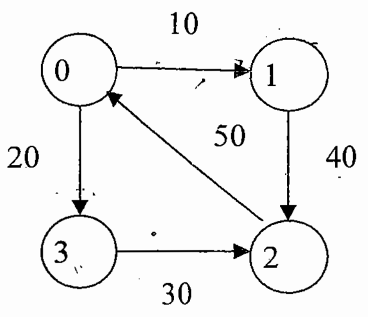
Edges from vertex 0 are (0,1) (0,2)
and (0,3) represented by the nodes N0, N1 and N2
Edges from vertex 1 are (1,0) (1,2) and (1,3) represented by the nodes N0, N3 and N4
Edges from vertex 2 are (2,0) (2,1) and (2,3) represented by the nodes N1, N3 and N5
Edges from vertex 3 are (3,0) (3,1) and (3,2) represented by the nodes N2, N4 and N5

Weighted Edges and Weighted Graph

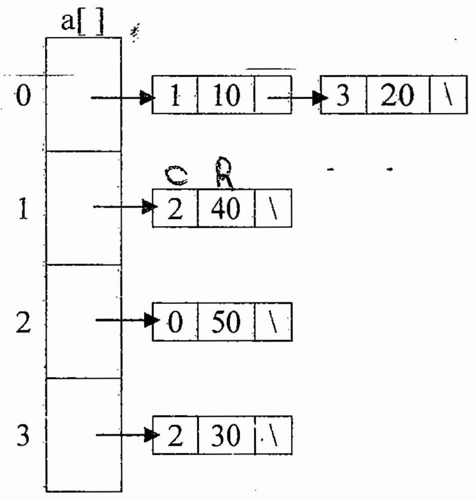
In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. Such edges with weights associated to them are called as weighted edges.

A graph in which weights (a number) are assigned to each edge is called as a weighted graph.

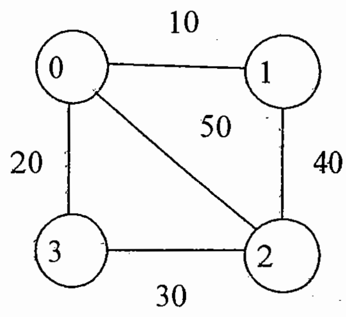
Weighted graphs can be represented using adjacency matrix (known as cost adjacency matrix) or adjacency list (known as cost adjacency list)
Example:
The following is a directed graph with its cost adjacency matrix and cost adjacency list.



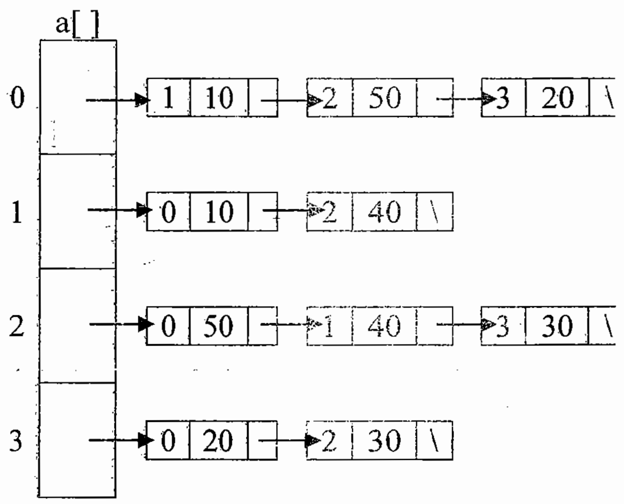
	0	1	2	3
0	∞	10	∞	20
1	∞	∞	40	∞
2	50	∞	∞	∞
3	∞	∞	30	∞



The following is a undirected graph with its cost adjacency matrix and cost adjacency list.



	0	1	2	3
0	∞	10	50	20
1	10	∞	40	∞
2	50	40	∞	30
3	20	∞	30	∞

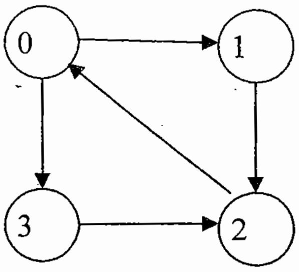


Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Elementary Graph Operations

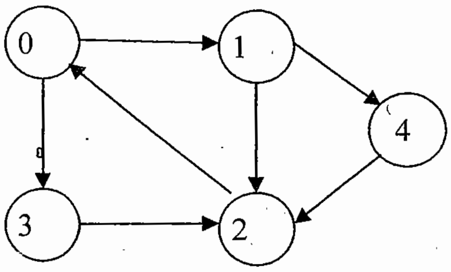
Insert operation in Graph

1) Vertex Insertion - This operation is done by adding the vertex name in the row and column of the adjacency matrix, and then mark the edges of the newly added vertex in the matrix.
Example:
Consider the following directed graph and its adjacency matrix



	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

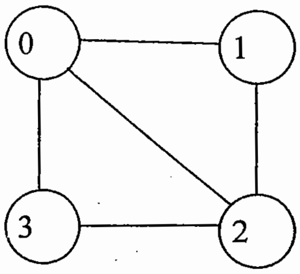
· Now if we insert a vertex 4, by attaching it to vertices 1 and 2, then the graph becomes



	0	1	2	3	**4**
0	0	1	0	1	**0**
1	0	0	1	0	**1**
2	1	0	0	0	**0**
3	0	0	1	0	**0**
4	**0**	**0**	**1**	**0**	**0**

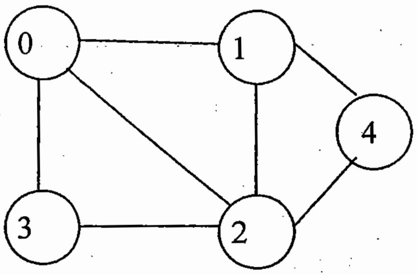
Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Consider the following undirected graph and its adjacency matrix



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

Now if we insert a vertex 4, by attaching it to vertices 1 and 2, then the graph becomes

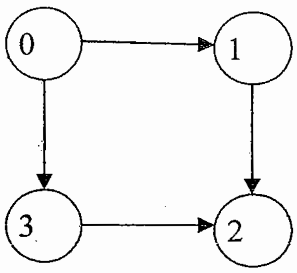


	0	1	2	3	**4**
0	0	1	1	1	**0**
1	1	0	1	0	**1**
2	1	1	0	1	**1**
3	1	0	1	0	**0**
4	**0**	**1**	**1**	**0**	**0**

2) Edge Addition - Suppose the newly added edge corresponds to the vertices v_i and v_j, then this operation is done by marking

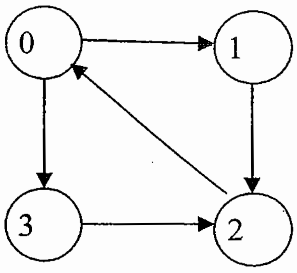
- $a[v_i][v_j]$ to 1 in case of a directed graph
- Both $a[v_i][v_j]$ and $a[v_j][v_i]$ to 1 in case of a undirected graph

Example:
Consider the following directed graph and its adjacency matrix



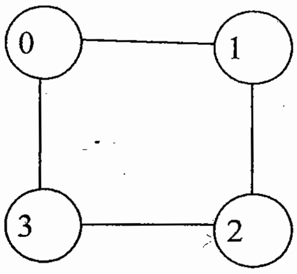
	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	0	0	0	0
3	0	0	1	0

Now if we insert a edge 2-0, then the graph becomes



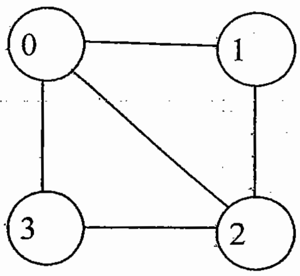
	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

Consider the following undirected graph and its adjacency matrix



	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	1
3	1	0	1	0

Now if we insert a edge 2-0, then the graph becomes



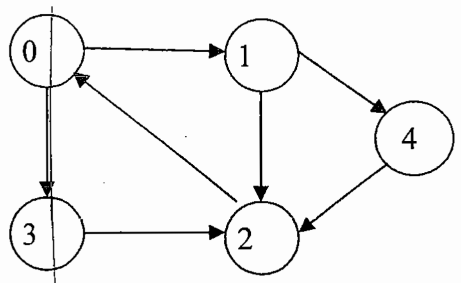
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

Delete operation in Graph

1) Vertex Deletion - This operation is done by unmarking all the edges corresponding to the vertex to be deleted in the adjacency matrix.

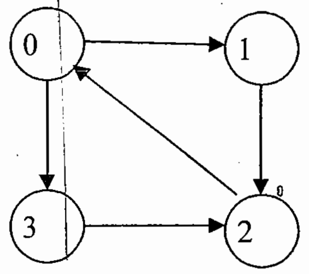
Example:

Consider the following directed graph and its adjacency matrix



	0	1	2	3	4
0	0	1	0	1	0
1	0	0	1	0	1
2	1	0	0	0	0
3	0	0	1	0	0
4	0	0	1	0	0

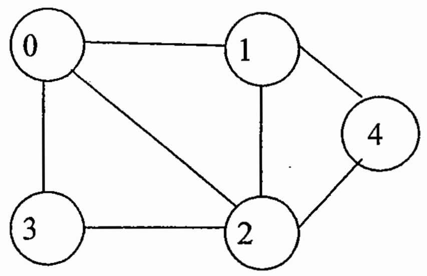
Now if we delete vertex 4, then the graph becomes



	0	1	2	3	4
0	0	1	0	1	0
1	0	0	1	0	**0**
2	1	0	0	0	0
3	0	0	1	0	0
4	0	0	**0**	0	0

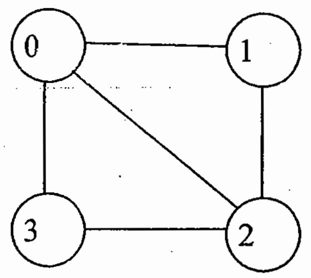
Consider the following undirected graph and its adjacency matrix

Dr.Mahesh G Assoc. Prof. BMSIT & M
Dr.Harish G Assoc. Prof. Dr. AIT



	0	1	2	3	4
0	0	1	1	1	0
1	1	0	1	0	1
2	1	1	0	1	1
3	1	0	1	0	0
4	0	1	1	0	0

Now if we delete vertex 4, then the graph becomes

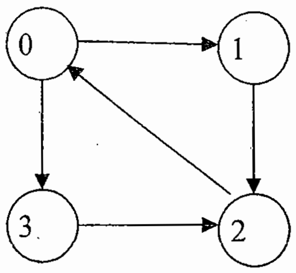


	0	1	2	3	4
0	0	1	1	1	0
1	1	0	1	0	**0**
2	1	1	0	1	**0**
3	1	0	1	0	0
4	0	**0**	**0**	0	0

2) Edge Deletion - Suppose the deleted edge corresponds to the vertices v_i and v_j, then this operation is done by marking

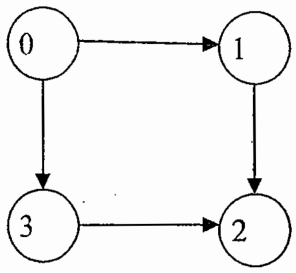
- $a[v_i][v_j]$ to 0 in case of a directed graph
- Both $a[v_i][v_j]$ and $a[v_j][v_i]$ to 0 in case of a undirected graph

Example:
Consider the following directed graph and its adjacency matrix



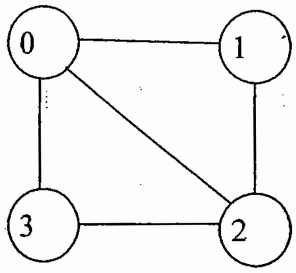
	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	1	0	0	0
3	0	0	1	0

Now if we delete edge 2-0, then the graph becomes



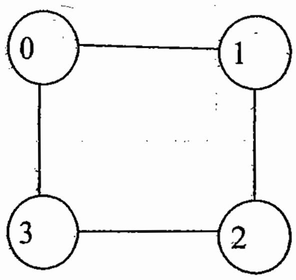
	0	1	2	3
0	0	1	0	1
1	0	0	1	0
2	**0**	0	0	0
3	0	0	1	0

Consider the following undirected graph and its adjacency matrix



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

Now if we delete edge 2-0, then the graph becomes



	0	1	2	3
0	0	1	**0**	1
1	1	0	1	0
2	**0**	1	0	1
3	1	0	1	0

Search operation in Graph
1) Find Vertex - This operation is done by scanning the complete graph using any traversal techniques. If the vertex is found, then search is successful else display a message stating the vertex is not found.
Example of graph traversal techniques include BFS and DFS.

10

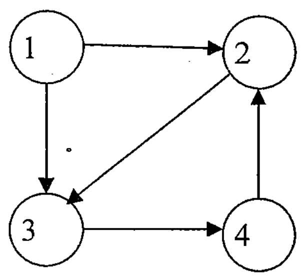
Warshalls Algorithm

This algorithm is used to find the path matrix P of a graph G.

Path Matrix

Let G=(V,E) be a graph, where V is the set of vertices and E is the set of edges. The path matrix for the graph G with 'n' vertices is a "n x n" two dimensional array a[ ][ ] such that,
a[i][j] = 1 if there is a path from vertex 'i' to vertex 'j' and
a[i][j] = 0 if there is no path from vertex 'i' to vertex 'j'.

Example: Consider the following graph with adjacency matrix



	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

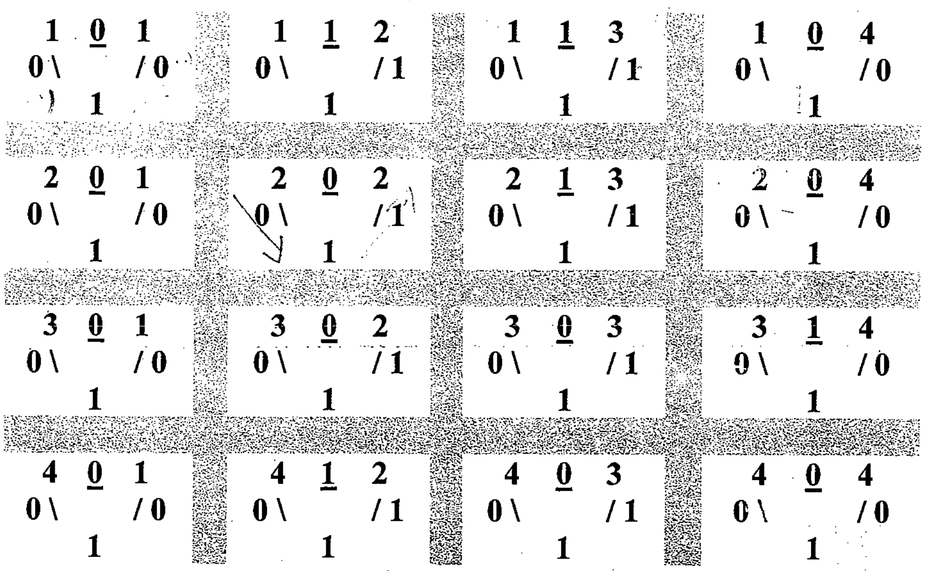
Copy the adjacency matrix to path matrix

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

Path Matrix P

Let K=1, we find whether a path exists from i to j through 1and the path matrix obtained through node 1 is



	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

Path Matrix P

Let K=2, we find whether a path exists from i to j through 2 and the path matrix obtained through node 2 is

```
  1  0  1        1  1  2        1  1  3        1  0  4
 1\     /0      1\     /0      1\     /1      1\     /0
    2              2              2              2

  2  0  1        2  0  2        2  1  3        2  0  4
 0\     /0      0\     /0      0\     /1      0\     /0
    2              2              2              2

  3  0  1        3  0  2        3  0  3        3  1  4
 0\     /0      0\     /0      0\     /1      0\     /0
    2              2              2              2

  4  0  1        4  1  2        4  0  3        4  0  4
 1\     /0      1\     /0      1\     /1      1\     /0
    2              2              2              2
```

	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	1
4	0	1	1	0

Path Matrix P

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

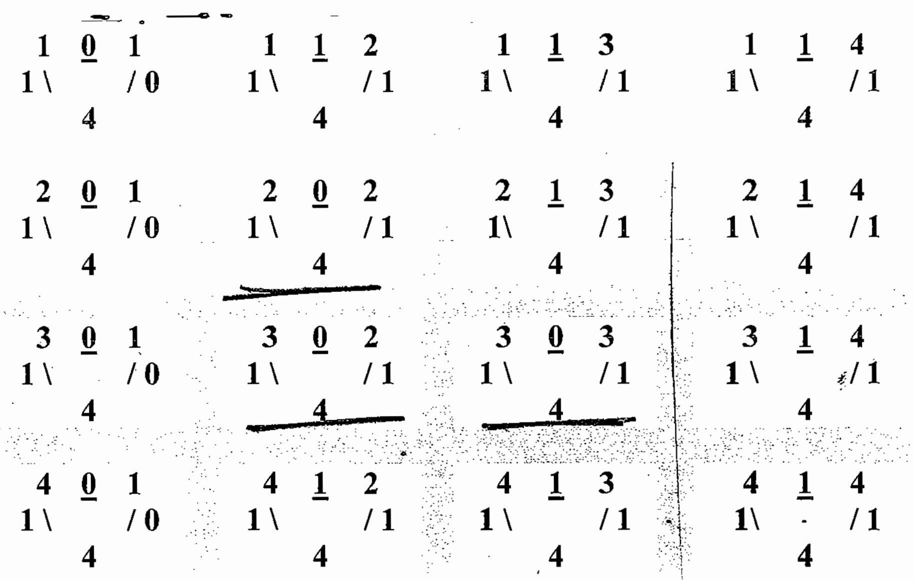
Let K=3, we find whether a path exists from i to j through 3 and the path matrix obtained through node 3 is

```
  1  0  1        1  1  2        1  1  3        1  0  4
 1\     /0      1\     /0      1\     /0      1\     /1
    3              3              3              3

  2  0  1        2  0  2        2  1  3        2  0  4
 1\     /0      1\     /0      1\     /0      1\     /1
    3              3              3              3

  3  0  1        3  0  2        3  0  3        3  1  4
 0\     /0      0\     /0      0\     /0      0\     /1
    3              3              3              3

  4  0  1        4  1  2        4  1  3        4  0  4
 1\     /0      1\     /0      1\     /0      1\     /1
    3              3              3              3
```

	1	2	3	4
1	0	1	1	1
2	0	0	1	1
3	0	0	0	1
4	0	1	1	1

Path Matrix P

Let K=4, we find whether a path exists from i to j through 4 and the path matrix obtained through node 3 is



	1	2	3	4
1	0	1	1	1
2	0	1	1	1
3	0	1	1	1
4	0	1	1	1

Path Matrix P

Algorithm

Read n

Read the adjacency matrix

Copy the adjacency matrix to path matrix

Find the path matrix

```
for( k=1; k<=n; k++ )
{
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        if(p[i][j] = = 0)
                        {
                                if(p[i][k] = = 1 && p[k][j] = = 1)
                                {
                                        p[i][j]) = 1;
                                }
                        }
                }
        }
}
```

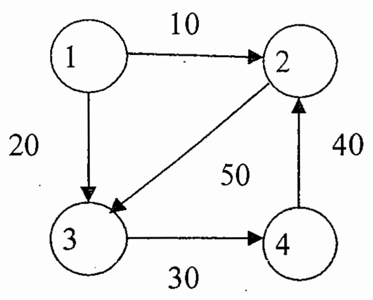
Write the path matrix

```
#include<stdio.h>
void main( )
{
        int i, j, k, n, a[20][20], p[20][20];
        printf("Enter the number of vertices\n");
        scanf("%d", &n);
        // Read the adjacency matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        scanf("%d", &a[i][j]);
                }
        }
        // Copy the adjacency matrix to path matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        p[i][j] = a[i][j];
                }
        }
        // Warshalls algorithm
        for( k=1; k<=n; k++ )
        {
                for( i=1; i<=n; i++ )
                {
                        for( j=1; j<=n; j++ )
                        {
                                if(p[i][j] = = 0)
                                {
                                        if(p[i][k] = = 1 && p[k][j] = = 1)
                                        {
                                                p[i][j]) = 1;
                                        }
                                }
                        }
                }
        }
        // Write the path matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        printf("%d\t", p[i][j]);
                }
                printf("\n");
        }
        getch( );
}
```

All Pairs Shortest Path / Floyds Algorithm

This algorithm is used to find the shortest distance from all nodes to all other nodes. We modify the warshalls algorithm to obtain the solution. The input for this is the cost adjacency matrix.

Example: Consider the following graph with cost adjacency matrix



	1	2	3	4
1	0	10	20	∞
2	∞	0	50	∞
3	∞	∞	0	30
4	∞	40	∞	0

Copy the adjacency matrix to distance matrix

	1	2	3	4
1	0	10	20	∞
2	∞	0	50	∞
3	∞	∞	0	30
4	∞	40	∞	0

Distance Matrix d

Dr.Mahesh G Assoc. Prof. BMSIT & M
Dr.Harish G Assoc. Prof. Dr. AIT

Let K=1, we find the distance matrix 'd' through node 1

$\begin{array}{ccc} 1 & \underline{0} & 1 \\ 0\backslash & & /0 \\ & 1 & \end{array}$	$\begin{array}{ccc} 1 & \underline{10} & 2 \\ 0\backslash & & /10 \\ & 1 & \end{array}$	$\begin{array}{ccc} 1 & \underline{20} & 3 \\ 0\backslash & & /30 \\ & 1 & \end{array}$	$\begin{array}{ccc} 1 & \underline{\infty} & 4 \\ 0\backslash & & /\infty \\ & 1 & \end{array}$
$\begin{array}{ccc} 2 & \underline{\infty} & 1 \\ \infty\backslash & & /0 \\ & 1 & \end{array}$	$\begin{array}{ccc} 2 & \underline{0} & 2 \\ \infty\backslash & & /10 \\ & 1 & \end{array}$	$\begin{array}{ccc} 2 & \underline{50} & 3 \\ \infty\backslash & & /30 \\ & 1 & \end{array}$	$\begin{array}{ccc} 2 & \underline{\infty} & 4 \\ \infty\backslash & & /\infty \\ & 1 & \end{array}$
$\begin{array}{ccc} 3 & \underline{\infty} & 1 \\ \infty\backslash & & /0 \\ & 1 & \end{array}$	$\begin{array}{ccc} 3 & \underline{\infty} & 2 \\ \infty\backslash & & /10 \\ & 1 & \end{array}$	$\begin{array}{ccc} 3 & \underline{0} & 3 \\ \infty\backslash & & /30 \\ & 1 & \end{array}$	$\begin{array}{ccc} 3 & \underline{30} & 4 \\ \infty\backslash & & /\infty \\ & 1 & \end{array}$
$\begin{array}{ccc} 4 & \underline{\infty} & 1 \\ \infty\backslash & & /0 \\ & 1 & \end{array}$	$\begin{array}{ccc} 4 & \underline{40} & 2 \\ \infty\backslash & & /10 \\ & 1 & \end{array}$	$\begin{array}{ccc} 4 & \underline{\infty} & 3 \\ \infty\backslash & & /30 \\ & 1 & \end{array}$	$\begin{array}{ccc} 4 & \underline{0} & 4 \\ \infty\backslash & & /\infty \\ & 1 & \end{array}$

	1	2	3	4
1	0	10	20	∞
2	∞	0	50	∞
3	∞	∞	0	30
4	∞	40	∞	0

Distance Matrix d

Let K=2, we find the distance matrix 'd' through node 2

```
    1   0   1        1  10   2        1  20   3        1   ∞   4
  10 \     / ∞     10 \     / 0     10 \     / 50    10 \     / ∞
        2                2                2                2

    2   ∞   1        2   0   2        2  50   3        2   ∞   4
   0 \     / ∞      0 \     / 0      0 \     / 50     0 \     / ∞
        2                2                2                2

    3   ∞   1        3   ∞   2        3   0   3        3  30   4
   ∞ \     / ∞      ∞ \     / 0      ∞ \     / 50     ∞ \     / ∞
        2                2                2                2

    4   ∞   1        4  40   2        4   ∞   3        4   0   4
  40 \     / ∞     40 \     / 0     40 \     / 50    40 \     / ∞
        2                2                2                2
```

$$\begin{array}{c|c|c|c|c}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 10 & 20 & \infty \\
2 & \infty & 0 & 50 & \infty \\
3 & \infty & \infty & 0 & 30 \\
4 & \infty & 40 & 90 & 0
\end{array}$$

Distance Matrix d

Dr.Mahesh G Assoc. Prof. BMSIT & M **Dr.Harish G** Assoc. Prof. Dr. AIT

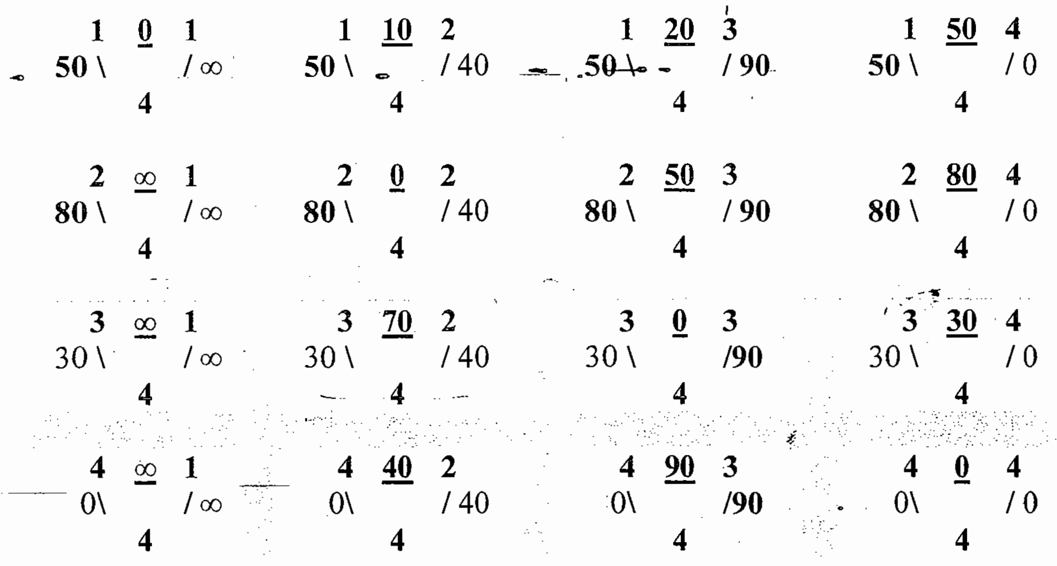
Let K=3, we find the distance matrix 'd' through node 3

```
    1   0   1        1  10   2        1  20   3        1   ∞   4
  20 \     / ∞     20 \     / ∞     20 \     / 0     20 \     / 30
        3                3                3                3

    2   ∞   1        2   0   2        2  50   3        2   ∞   4
  50 \     / ∞     50 \     / ∞     50 \     / 0     50 \     / 30
        3                3                3                3

    3   ∞   1        3   ∞   2        3   0   3        3  30   4
   0 \     / ∞      0 \     / ∞      0 \     / 0      0 \     / 30
        3                3                3                3

    4   ∞   1        4  40   2        4  90   3        4   0   4
  90 \     / ∞     90 \     / ∞     90 \     / 0     90 \     / 30
        3                3                3                3
```

$$\begin{array}{c|c|c|c|c}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 10 & 20 & \mathbf{50} \\
2 & \infty & 0 & 50 & \mathbf{80} \\
3 & \infty & \infty & 0 & 30 \\
4 & \infty & 40 & 90 & 0
\end{array}$$

Distance Matrix d

Let K=4, we find the distance matrix 'd' through node 4



	1	2	3	4
1	0	10	20	50
2	∞	0	50	80
3	∞	70	0	30
4	∞	40	90	0

Distance Matrix d

Algorithm

Read n

Read the cost adjacency matrix

Copy the cost adjacency matrix to distance matrix 'd'

Find the shortest distance matrix

```
for( k=1; k<=n; k++ )
{
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        d[i][j] = min( d[i][j], d[i][k] + d[k][j] )
                }
        }
}
```

Write the shortest distance matrix

```
#include<stdio.h>
int min(int x, int y)
{
        if(x<y)
                return x;
        else
                return y;
}
void main( )
{
        int i, j, k, n, a[20][20], d[20][20];
        printf("Enter the number of vertices\n");
        scanf("%d", &n);
        // Read the cost adjacency matrix (Enter 999 or a big number for infinity)
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        scanf("%d", &a[i][j]);
                }
        }
        // Copy the adjacency matrix to distance matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        d[i][j] = a[i][j];
                }
        }
        // Floyds algorithm
        for( k=1; k<=n; k++ )
        {
                for( i=1; i<=n; i++ )
                {
                        for( j=1; j<=n; j++ )
                        {
                                d[i][j] = min( d[i][j], d[i][k] + d[k][j] )
                        }
                }
        }
        // Write the shortest distance matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        printf("%d\t", d[i][j]);
                }
                printf("\n");
        }
}
```

Graph Traversals

Traversing a graph is the systematic approach of visiting each node of the graph and do the processing (Ex: Printing the nodes information).
The two traversal techniques of graphs are
1) Breadth First Search (BFS)
2) Depth First Search (DFS)

Breadth First Search (BFS)

In this traversal technique, the vertices are visited in the order of their levels from the source node i.e. all the vertices at distance 'k' from the source vertex are visited before visiting the nodes at distance 'k+1'.
BFS can be used

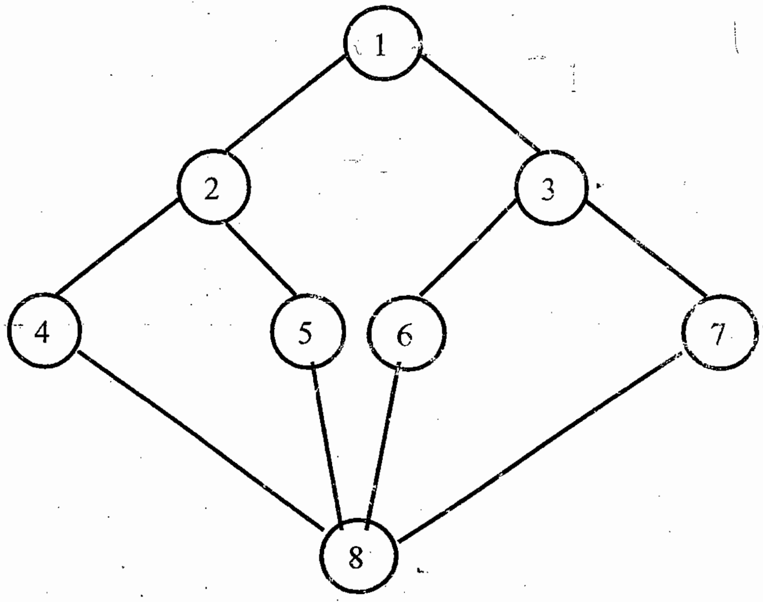
1. To check if the graph is connected or not.
2. To find the vertices which are reachable from a given vertex.
3. To find the spanning tree if it exists.
4. To find whether a graph is acyclic or not.

Algorithm

Initialize the visited[ ], f and r of q[ ]

```
Step 1: Visit the source vertex and insert the source vertex to queue.
Step 2: As long as the queue is not empty
        Step 2a: Delete a vertex 'u' from the queue
        Step 2b: Find all the nodes which are adjacent to the deleted vertex 'u'
                 and are not visited earlier 'v'.
                   ✓ Visit these nodes 'v'
                   ✓ Insert them to queue from left to right
                   ✓ Store the edges (from deleted vertex 'u' to adjacent nodes 'v')
```

Example: Consider the graph shown below



	-1	0	1	2	3	4	5	6	7	8
q[ ]										
	r	f								

	0	1	2	3	4	5	6	7	8
visited[ ]		0	0	0	0	0	0	0	0

Step 1: Assuming source vertex is 1

	-1	0	1	2	3	4	5	6	7	8
q[ ]		1								
		f, r								

	0	1	2	3	4	5	6	7	8
visited[ ]		1	0	0	0	0	0	0	0

Step 2:

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

// 'u' will be the element deleted from the queue
// 'v' will the nodes adjacent to 'u' and not visited
// SPTE will be the edges of the Spanning Tree if the graph is connected i.e. all nodes are reachable

Visited[ ]									q[ ]										u	v	e[ ][ ]
1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7	8				u-v
1	0	0	0	0	0	0	0		1										1	2 3	1-2 1-3
1	1	1	0	0	0	0	0		2	3									2	4 5	2-4 2-5
1	1	1	1	1	0	0	0			3	4	5							3	6 7	3-6 3-7
1	1	1	1	1	1	1					4	5	6	7					4	8	4-8
1	1	1	1	1	1	1	1					5	6	7	8				5	-	-
1	1	1	1	1	1	1	1						6	7	8				6	-	-
1	1	1	1	1	1	1	1							7	8				7	-	-
1	1	1	1	1	1	1	1								8				8	-	-
List of nodes visited									Queue is empty stop												SPTE

```
#include<stdio.h>
void main( )
{
        int n, i, j, a[20][20], source, visited[20], e[20][2];
        printf("Enter the number of vertices\n");
        scanf("%d", &n);
        // Read the adjacency matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        scanf("%d", &a[i][j]);
                }
        }
        printf("Enter the source vertex\n");
        scanf("%d", &source);
        for(i=1; i<=n; i++)
        {
                visited[i] = 0;
        }
        bfs( n,  a, source, visited, e );
        // To print the nodes which are reachable and not reachable from source
        flag = 0;
        for(i=1; i<=n; i++)
        {
                if(visited[i] = = 0)
                {
                        printf("%d is not reachable \n", i);
                        flag = 1;
                }
                else
                {
                        printf("%d is reachable \n", i);
                }
        }
        // To check if the graph is connected and for printing spanning tree
        if(flag = = 1)
        {
                printf("The graph is not connected\n");
        }
        else
        {
                printf("The graph is connected\n");
                printf("The spanning tree or BFS traversal is\n");
                for(i=1; i < = n-1; i++)
                {
                        printf("Edge from %d to %d\n", e[i][1], e[i][2]);
                }
        }
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

lab

```
void bfs(  int n, int a[20][20], int source, int visited[ ], int e[20][2] )
{
        int f = 0, r = -1, q[20]; // queue components

        int u, v, k = 1;

        // visit the source node and insert it into the queue
        visited[source] = 1;
        r = r + 1;
        q[r] = source;

        while( f < = r) // as long as the queue is not empty
        {
                u = q[f];          // delete an element from the queue
                f = f + 1;

                // find the nodes adjacent to 'u' and not visited
                for( v=1; v<=n; v++)
                {
                        if( a[u][v] = = 1 && visited[v] = = 0)
                        {
                                visited[v] =1;  // visit the node
                                r = r + 1;
                                q[r] = v;        // insert the node to queue
                                e[k][1] = u;
                                e[k][2] = v;    // store the edges
                                k = k + 1;
                        }
                }
        }
}
```

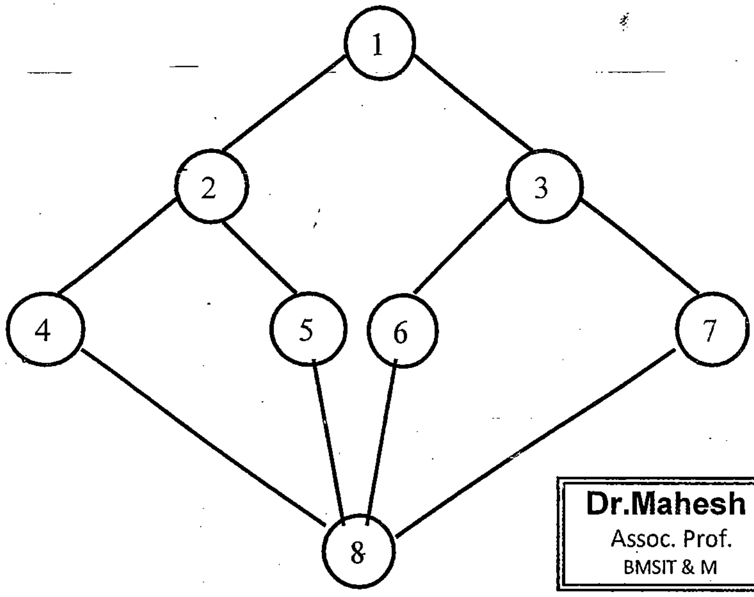
Dr.Mahesh G	Dr.Harish G
Assoc. Prof.	Assoc. Prof.
BMSIT & M	Dr. AIT

Depth First Search

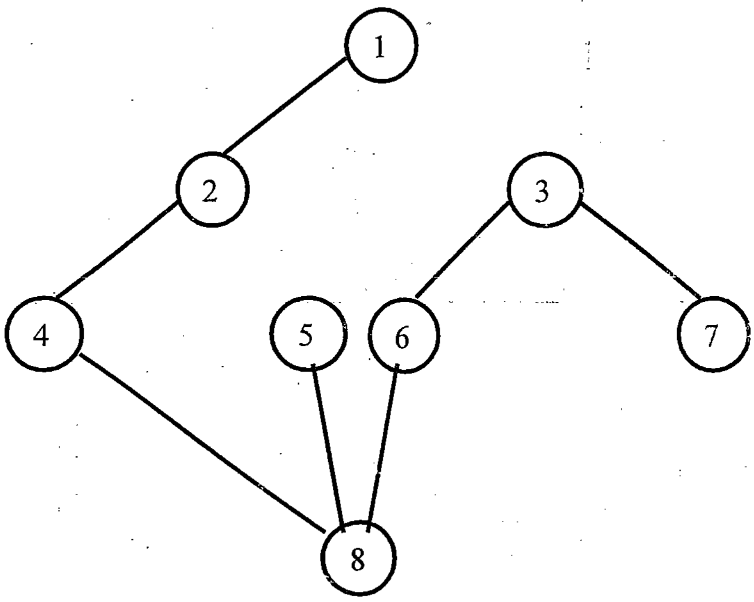
In this traversal technique, a node 's' is picked as a start node and marked visited. An unmarked adjacent node to 's' is selected to become the new start node and marked, possibly leaving the original start node with unexplored edges for the time being.
The search continues in the graph until the current path ends or until all the adjacent nodes are already marked. Then the search returns to the previous nodes which still had unmarked adjacent nodes and continues marking until all nodes are marked.

Note: We use a recursive call to the DFS function, hence the data structure in use for DFS is stack.

Example: Consider the graph shown below



Assuming source = 1, the depth first traversal for the above graph is



Algorithm

```
Initialize the visited[ ]
Step 1: Visit the source vertex 'u'
Step 2: For every node 'v' adjacent to vertex 'u'
            If 'v' is not visited earlier
                  Store the edges (from source vertex 'u' to adjacent nodes 'v')
                  Repeat through Step 1 with source as 'v'
```

```
#include<stdio.h>

void dfs( int n, int a[20][20], int u, int visited[ ], int e[20][2] )
{
      int v;
      static int k = 1;

      // visit the source vertex 'u'
      visited[u] = 1;

      // find the nodes adjacent to 'u' and not visited
      for( v=1; v<=n; v++)
      {
            if( a[u][v] = = 1 && visited[v] = = 0)
            {
                  e[k][1] = u;
                  e[k][2] = v;    // store the edges
                  k = k + 1;
                  dfs( n, a, v, visited, e);
            }
      }
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

```
void main( )
{
        int n, i, j, a[20][20], source, visited[20], e[20][2];
        printf("Enter the number of vertices\n");
        scanf("%d", &n);
        // Read the adjacency matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        scanf("%d", &a[i][j]);
                }
        }
        printf("Enter the source vertex\n");
        scanf("%d", &source);
        for(i=1; i<=n; i++)
        {
                visited[i] = 0;
        }

        dfs( n,  a, source, visited, e );
        // To print the nodes which are reachable and not reachable from source
        flag = 0;
        for(i=1; i<=n; i++)
        {
                if(visited[i] = = 0)
                {
                        printf("%d is not reachable \n", i);
                        flag = 1;
                }
                else
                {
                        printf("%d is reachable \n", i);
                }
        }
        // To check if the graph is connected and for printing spanning tree
        if(flag = = 1)
        {
                printf("The graph is not connected\n");
        }
        else
        {
                printf("The graph is connected\n");
                printf("The spanning tree or DFS traversal is\n");
                for(i=1; i < = n-1; i++)
                {
                        printf("Edge from %d to %d\n", e[i][1], e[i][2]);
                }
        }
}
```

Lab Program

Design, Develop and Implement a Program in C for the following operations on **Graph(G)** of Cities
a. Create a Graph of **N** cities using Adjacency Matrix.
b. Print all the nodes **reachable** from a given starting node in a digraph using DFS/BFS method

```
#include<stdio.h>

void bfs( int n, int a[20][20], int source, int visited[ ], int e[20][2] )
{
        int f = 0, r = -1, q[20]; // queue components
        int u, v, k = 1;
        // visit the source node and insert it into the queue
        visited[source] = 1;
        r = r + 1;
        q[r] = source;
        while( f < = r) // as long as the queue is not empty
        {
                u = q[f];         // delete an element from the queue
                f = f + 1;
                // find the nodes adjacent to 'u' and not visited
                for( v=1; v<=n; v++)
                {
                        if( a[u][v] = = 1 && visited[v] = = 0)
                        {
                                visited[v] =1;  // visit the node
                                r = r + 1;
                                q[r] = v;          // insert the node to queue
                                e[k][1] = u;
                                e[k][2] = v;     // store the edges
                                k = k + 1;
                        }
                }
        }
}
```

```
void dfs(  int n, int a[20][20], int u, int visited[ ], int e[20][2] )
{
        int v;
        static int k = 1;
        // visit the source vertex 'u'
        visited[u] = 1;
        // find the nodes adjacent to 'u' and not visited
        for( v=1; v<=n; v++)
        {
                if( a[u][v] = = 1 && visited[v] = = 0)
                {
                        e[k][1] = u;
                        e[k][2] = v;      // store the edges
                        k = k + 1;
                        dfs( n, a, v, visited, e);
                }
        }
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

```
void main( )
{
        int n, i, j, a[20][20], source, visited[20], e[20][2],choice,flag;

        printf("Enter the number of vertices\n");
        scanf("%d", &n);

        // Read the adjacency matrix
        for( i=1; i<=n; i++ )
        {
                for( j=1; j<=n; j++ )
                {
                        scanf("%d", &a[i][j]);
                }
        }
        printf("Enter the source vertex\n");
        scanf("%d", &source);

        for(i=1; i<=n; i++)
        {
                visited[i] = 0;
        }
        printf("1.BFS  2.DFS\n");
        printf("Enter ur choice\n");
        scanf("%d", &choice);

        if(choice = = 1)
                bfs( n,  a, source, visited, e );
        else
                dfs( n,  a, source, visited, e );
```

// To print the nodes which are reachable and not reachable from source

```
        flag = 0;

        for(i=1; i<=n; i++)
        {
                if(visited[i] = = 0)
                {
                        printf("%d is not reachable \n", i);
                        flag = 1;
                }
                else
                {
                        printf("%d is reachable \n", i);
                }
        }
```

// To check if the graph is connected and for printing spanning tree

```
        if(flag = = 1)
        {
                printf("The graph is not connected\n");
        }
        else
        {
                printf("The graph is connected\n");
                printf("The spanning tree or traversal is\n");
                for(i=1; i < = n-1; i++)
                {
                        printf("Edge from %d to %d\n", e[i][1], e[i][2]);
                }
        }
        getch( );
}
```

Insertion Sort

Let the number of elements to be sorted be 'n'. To sort 'n' elements using insertion sort,

Step 1: The second element of the array is compared with the elements that appears before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.

Step 2: The third element of the array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.

Step 3: Similarly, the fourth element of the array is compared with the elements that appears before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted.

If there are n elements to be sorted. Then, this procedure is repeated n-1 times to get sorted list of array.

Example: Consider the array a[ ] = { **25**, 75, 40, 10, 20, 5, 7}

Step 1: Item to be inserted a[1] = 75

Compare item to be inserted with the elements that appear before it to find and insert the item at appropriate.

Resulting array a[ ] = { **25, 75**, 40, 10, 20, 5, 7}

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Step 2: Item to be inserted a[2] = 40

Compare item to be inserted with the elements that appear before it to find and insert the item at appropriate.

Resulting array a[ ] = { **25, 40, 75**, 10, 20, 5, 7}

Step 3: Item to be inserted a[3] = 10

Compare item to be inserted with the elements that appear before it to find and insert the item at appropriate.

Resulting array a[ ] = { **10, 25, 40, 75**, 20, 5, 7}

Step 4: Item to be inserted a[4] = 20

Compare item to be inserted with the elements that appear before it to find and insert the item at appropriate.

Resulting array a[ ] = { **10, 20, 25, 40, 75**, 5, 7}

Step 5: Item to be inserted a[5] = 5

Compare item to be inserted with the elements that appear before it to find and insert the item at appropriate.

Resulting array a[ ] = { **5,10, 20, 25, 40, 75**, 7}

Step 6: Item to be inserted a[6] = 7

Compare item to be inserted with the elements that appear before it to find and insert the item at appropriate.

Resulting array a[ ] = { **5,7,10, 20, 25, 40, 75**}

Note:

1) For 'n' elements item to be inserted range from a[1] to a[n-1]

2) To find appropriate place to insert item = a[i]

as long as item < elements to the left of a[i], shift elements by one position

3) Insert the item in the appropriate position found.

```
#include<stdio.h>

void insertion_sort( int a[ ], int n)
{
        int i, j, item;

        for( i = 1; i <= n-1; i++)
        {
                item = a[ i ];  // item to be inserted

                // find appropriate place for a[i]
                j = i - 1;
                while( item < a[ j ] && j >= 0)
                {
                        a[ j + 1 ] = a[ j ]
                        j = j - 1;
                }

                // Insert item at appropriate position
                a[ j + 1] = item;
        }
}

void main( )
{
        int i, n, a[20];

        printf("Enter the number of elements\n");
        scanf("%d", &n);

        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
        }

        insertion_sort( a, n);

        printf("The sorted elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d", a[i]);
        }
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Radix Sort

Step 1: Find the biggest of 'n' elements in the array and also find the number of digits in it (m). This position corresponds to the number of passes to be executed.

Example:

If 97 is found to be the biggest number, then m = 2

If 189 is found to be the biggest number, then m = 3

If 1923 is found to be the biggest number, then m = 4

Note: To find value m

m = log10(biggest number) + 1

Example:

log10(97) = 1.986 m = 1 + 1 = 2

log10(189) = 2.276 m = 2 + 1 = 3

log10(1923) = 3.283 m = 3 + 1 = 4

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Step 2: Repeat the following Steps 3 to 5 'm' times (j = 1 to m)

Step 3: Create an array of linked list with index from 0 to 9 (as these are the only possible integers)

Step 4: Obtain the jth least significant digit for each element in the given array to get the index of the linked list where the element needs to be stored and Store the element using insert_rear function of linked list.

Note: To find the jth least significant digit use the formula

item / pow(10, j-1) % 10

Example: If item = 1923

for j = 1, we get 1923 / pow(10, 0) % 10 = 1923 / 1 % 10 = 1923 % 10 = 3

for j = 2, we get 1923 / pow(10, 1) % 10 = 1923 / 10 % 10 = 192 % 10 = 2

for j = 3, we get 1923 / pow(10, 2) % 10 = 1923 / 100 % 10 = 19 % 10 = 9

for j = 4, we get 1923 / pow(10, 3) % 10 = 1923 / 1000 % 10 = 1 % 10 = 1

Step 5: Copy the elements from the linked list inorder from 0 to 9 into an array.

Step 6: Display the array elements.

Example:

Consider the array elements 28, 43, 55, 14, 88, 39, 91, 33, 49, 57, 45, 79, 62, 86

Step 1: Biggest number = 91 and m = 2

j = 1

Step 3:

First[0] = NULL
First[1] = NULL
First[2] = NULL
First[3] = NULL
First[4] = NULL
First[5] = NULL
First[6] = NULL
First[7] = NULL
First[8] = NULL
First[9] = NULL

Step 4:

F(28) = 8, F(43) = 3, F(55) = 5, F(14) = 4, F(88) = 8, F(39) = 9, F(91) = 1, F(33) = 3, F(49) = 9, F(57) = 7, F(45) = 5, F(79) = 9, F(62) = 2, F(86) = 6

First[0]			
First[1]	91		
First[2]	62		
First[3]	43	33	
First[4]	14		
First[5]	55	45	
First[6]	86		
First[7]	57		
First[8]	28	88	
First[9]	39	49	79

Step 5: a[ ] = { 91, 62, 43, 33, 14, 55, 45, 86, 57, 28, 88, 39, 49, 79}

j = 2

Step 3:

First[0] = NULL
First[1] = NULL
First[2] = NULL
First[3] = NULL
First[4] = NULL
First[5] = NULL
First[6] = NULL
First[7] = NULL
First[8] = NULL
First[9] = NULL

Step 4:

F(91) = 9, F(62) = 6, F(43) = 4, F(33) = 3, F(14) = 1, F(55) = 5, F(45) = 4, F(86) = 8, F(57) = 5, F(28) = 2, F(88) = 8, F(39) = 3, F(49) = 4, F(79) = 7

First[0]			
First[1]	14		
First[2]	28		
First[3]	33	39	
First[4]	43	45	49
First[5]	55	57	
First[6]	62		
First[7]	79		
First[8]	86	88	
First[9]	91		

Step 5: a[ ] = { 14, 28, 33, 39, 43, 45, 49, 55, 57, 62, 79, 86, 88, 91}

```
#include<stdio.h>

struct node
{
        int info;
        struct node *link;
};
typedef struct node * NODE;

NODE insert_rear(int item,NODE first)
{
        NODE temp, cur;
        temp=(NODE)malloc(sizeof(struct node));
        temp->info = item;
        temp->link = NULL;
        if(first = = NULL)    //no node
        {
                return temp;
        }
        cur = first;              // node exists
        while(cur->link != NULL)
        {
                cur = cur->link;
        }
        cur->link = temp;
        return first;
}

int largest(int a[ ], int n)
{
        int big, i;
        big = a[0];
        for(i=1; i<n; i++)
        {
                if(a[i] > big)
                        big = a[i];
        }

        return big;
}

void radix_sort( int a[ ], int n)
{
        NODE first[10], temp;
        int i, j, k, m, big, index;

        big = largest(a, n);
        m = log10(big) + 1;
        for( j=1; j<=m; j++ )
        {
                for(i=0; i<10; i++)
                {
                        first[i] = NULL;
                }


                for(i=0; i<n; i++)
                {
                        index = a[i] / pow(10, j-1) %10;
                        first[index] = insert_rear( a[i], first[index]);
                }

                k=0;
                for(i=0; i<10; i++)
                {
                        temp = first[i];
                        while(temp != NULL)
                        {
                                a[ k ] = temp->info;
                                k = k + 1;
                                temp = temp->link;
                        }
                }
        }
}
```

```
void main( )
{
        int i, n, a[20];

        printf("Enter the number of elements\n");
        scanf("%d", &n);

        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
        }

        radix_sort( a, n);

        printf("The sorted elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d", a[i]);
        }
}
```

Address Calculation Sort

Step 1: Find the biggest of 'n' elements in the array and also find the position (tens, hundreds, thousand etc) of the biggest number. This position corresponds to the hash value.

Example:

If 97 is found to be the biggest number, then hash value = 10

If 189 is found to be the biggest number, then hash value = 100

If 1923 is found to be the biggest number, then hash value = 1000

Note: To find hash value

Let i = log10(biggest number) hash value = pow(10, i);

Example:

log10(97) = 1.986	hash value = pow(10, 1) = 10
log10(189) = 2.276	hash value = pow(10, 2) = 100
log10(1923) = 3.283	hash value = pow(10, 3) = 1000

Step 2: Create an array of linked list with index from 0 to 9 (as these are the only possible integers)

Step 3: Divide each element in the given array by the hash value to get the index of the linked list where the element needs to be stored and Store the element using insert_order function of linked list.

Step 4: Copy the elements from the linked list inorder from 0 to 9 into an array.

Step 5: Display the array elements.

Example:

Consider the array elements 28, 43, 55, 14, 88, 39, 91, 33, 49, 57, 45, 79, 62, 86

Step 1: Biggest number = 91 and hash value = 10

Step 2:

First[0] = NULL
First[1] = NULL
First[2] = NULL
First[3] = NULL
First[4] = NULL
First[5] = NULL
First[6] = NULL
First[7] = NULL
First[8] = NULL
First[9] = NULL

Dr.Mahesh G	**Dr.Harish G**
Assoc. Prof.	Assoc. Prof.
BMSIT & M	Dr. AIT

Step 3:

F(28) = 2, F(43) = 4, F(55) = 5, F(14) = 1, F(88) = 8, F(39) = 3, F(91) = 9, F(33) = 3, F(49) = 4, F(57) = 5, F(45) = 4, F(79) = 7, F(62) = 6, F(86) = 8

First[0]			
First[1]	14		
First[2]	28		
First[3]	33	39	
First[4]	43	45	49
First[5]	55	57	
First[6]	62		
First[7]	79		
First[8]	86	88	
First[9]	91		

Step 4: a[ ] = { 14, 28, 33, 39, 43, 45, 49, 55, 57, 62,79, 86, 88, 91}

```
#include<stdio.h>
struct node
{
        int info;
        struct node *link;
};
typedef struct node * NODE;

NODE insert_order(int item, NODE first)
{
        NODE temp, cur, prev;
        temp=(NODE)malloc(sizeof(struct node));
        temp->info = item;
        temp->link = NULL;
        if(first = = NULL)  // no elements in the list
                return temp;
        if(item <= first->info)         // insert at the beginning
        {
                temp->link = first;
                return temp;
        }

        prev = NULL;        // insert at middle or end
        cur = first;

        while(cur != NULL && item > cur->info)
        {
                prev = cur;
                cur = cur->link;
        }

        prev->link = temp;
        temp->link = cur;
        return first;
}

int hash(int a[ ], int n)
{
        int big, i;
        big = a[0];
        for(i=1; i<n; i++)
        {
                if(a[i] > big)
                        big = a[i];
        }
        i = log10(big);
        return( pow(10, i);
}
```

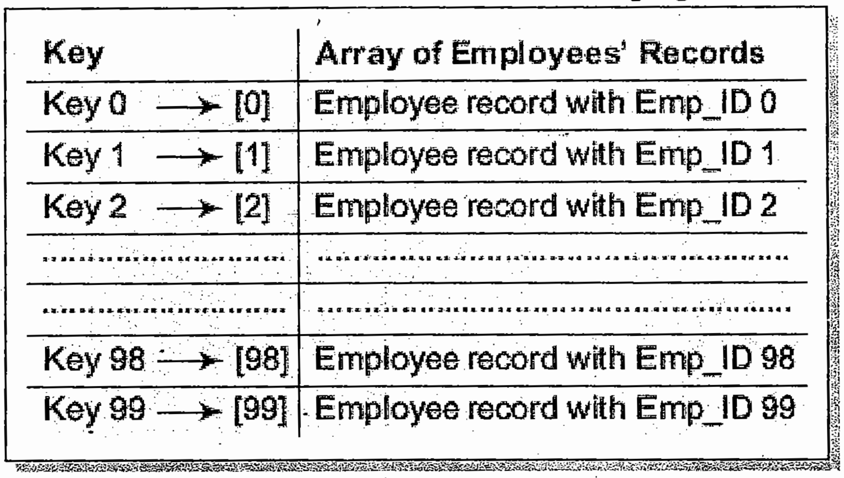
Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

```
void address_calculation_sort( int a[ ], int n)
{
        NODE first[10], temp;
        int i, k, index, hash_value;

        hash_value = hash( a, n);

        for(i=0; i<10; i++)
        {
                first[i] = NULL;
        }


        for(i=0; i<n; i++)
        {
                index = a[i] / hash_value;
                first[index] = insert_order( a[i], first[index]);
        }

        k=0;
        for(i=0; i<10; i++)
        {
                temp = first[i];
                while(temp != NULL)
                {
                        a[k ] = temp->info;
                        k = k + 1;
                        temp = temp->link;
                }
        }
}

void main( )
{
        int i, n, a[20];
        printf("Enter the number of elements\n");
        scanf("%d", &n);
        printf("Enter the elements\n");
        for(i=0; i<n; i++)
        {
                scanf("%d", &a[i]);
        }

        address_calculation_sort( a, n);
        printf("The sorted elements are\n");
        for(i=0; i<n; i++)
        {
                printf("%d", a[i]);
        }
}
```

Hashing

Linear search and binary search are the two well known searching techniques. In both the techniques the data to be searched for goes through several comparisons. For example, in linear search the data to be searched for is compared with elements starting from first to last one by one until the key is found or the search is unsuccessful. Similarly in binary search the data to be searched for is compared with the middle element, if it is less than the middle element, then searching is continued in the first half else search is continued in the second half.
The search time for both techniques depends on the number of elements. However, if we need a search technique to perform the search operation in O(1) time, then we need to use hashing. The objective of hashing is to minimize the comparisons. The basic idea is not to search for the correct position of data with comparisons but to compute the position / index of data within the hash table using the hash function.

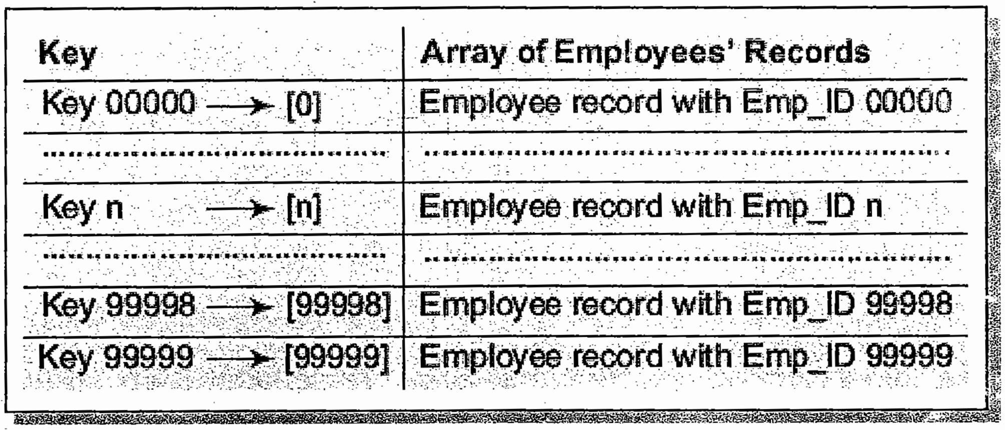
Consider a scenario for storing data about 100 employee records, where each record is uniquely identified by the field Emp_ID.
If the Emp_ID is in the range of 0 to 99, then to store the records, Emp_ID itself can be used as "index" to store the records. This is as shown in the following figure.



Key	Array of Employees' Records
Key 0 ⟶ [0]	Employee record with Emp_ID 0
Key 1 ⟶ [1]	Employee record with Emp_ID 1
Key 2 ⟶ [2]	Employee record with Emp_ID 2
....	....
....	....
Key 98 ⟶ [98]	Employee record with Emp_ID 98
Key 99 ⟶ [99]	Employee record with Emp_ID 99

An employee record for a particular Emp_ID can be easily accessed because index = Emp_ID. However, this would not be practically feasible.

Assume that the company uses a five digit Emp_ID. In this case the Emp_ID values range from 00000 to 99999. If we use the technique describe above, then an array of size 100000 is required to store 100 employee records. This is as shown below.



Key	Array of Employees' Records
Key 00000 ⟶ [0]	Employee record with Emp_ID 00000
....	....
Key n ⟶ [n]	Employee record with Emp_ID n
....	....
Key 99998 ⟶ [99998]	Employee record with Emp_ID 99998
Key 99999 ⟶ [99999]	Employee record with Emp_ID 99999

Whether we use a 2-digit Emp_ID (primary key) or a 5-digit key, the number of employees is limited in this case to 100 and hence it would be impractical to use so much of storage space for storing 100 employee records.

Another alternative to reduce the array size is to use the last 2-digits of the primary key (Emp_ID) to get the index of the array.

Example:

The employee with Emp_ID 59349 will be stored in the element of the array with index 49.
The employee with Emp_ID 12345 will be stored in the element of the array with index 45.

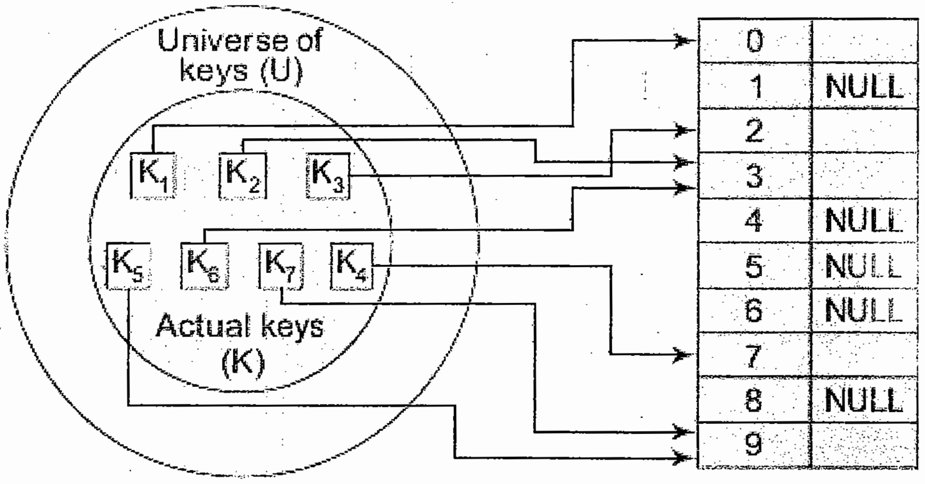
In this method the employee record is not stored according to the value of key (Emp_ID), instead, the 5-digit key is converted to 2-digit index.

The function ~~which~~ is used to convert the key to index is called **hash function** and the array which stores the records is called **hash table. Hashing** is the process of indexing the data so that sorting, searching, inserting and deleting data becomes fast.

Now, if we need to store an employee record with Emp_ID 23749, then index = 49. However, an employee record with Emp_ID 59349 would have been already stored at index = 49. This results in **collision**.

Hash Table

It is a data structure in which keys are mapped to array positions by a hash function.
Example: The key 23749 is mapped to the array position 49 using the hash function 23749 % 1000.
In a hash table, an element with key 'k' is stored at index h(k). The hash function 'h' is used to calculate the index at which the element with key 'k' will be stored.
The following shows the relationship between keys and the hash table index.



Note:
1) Keys K2 and K6 point to the same memory location, resulting in collision.
2) Keys K5 and K7 also point to the same memory location, and hence collide.

Hash Function

A hash function is a mathematical formula which when applied to a key, produces an integer which can be used as index for the key in the hash table.

Properties / Characteristics of Good Hash Function

1) **Low cost** - The function must be very easy and quick to compute.
2) **Determinism** - The function must produce the same hash value for a given input value.
3) **Uniformity** - The function must as far as possible, uniformly distribute the keys so that there are minimum number of collisions.

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Different Types of Hash Functions

1) Division Method
Choose a number 'm' larger than the number of keys 'n'. The hash function divides 'k' by 'm' and uses the remainder. It is defined by H(k) = k mod m
H(k) is the hash index generated for key 'k' and is the remainder when k is divided by m.
Example: Consider the keys to be stored in the hash table to be 64, 52, 79, 36. Then using the division method the keys are stored as shown below.
Choose a number m = 10

H(64) = 64 % 10 = 4	// 64 to be placed at address 4
H(52) = 52 % 10 = 2	// 52 to be placed at address 2
H(79) = 79 % 10 = 9	// 79 to be placed at address 9
H(36) = 36 % 10 = 6	// 36 to be placed at address 6

0	1	2	3	4	5	6	7	8	9
		52		64		36			79

2) Mid Square Method

Step 1: Square the value of the key i.e. find k^2
Step 2: Extract the middle 'r' digits of k^2

In this method the key to be stored 'k' is squared and the hash function is defined by
H(k) = mid part of k^2

Note: H(k) must be obtained by deleting digits from both ends of k^2 and the same position of k^2 must be used for all keys.

Example: Consider the data to be stored in the hash table to be 3111, 2345. Then using the mid square method data is stored as shown below.
$3111^2 = 9678321$
$2345^2 = 5499025$
Assuming the size of the hash table to be 1000, the middle 3 digits can be taken as hash address

H(3111) = 783	// 3111 to be placed at address 783
H(2345) = 990	// 2345 to be placed at address 990

Assuming the size of the hash table to be 100, the fourth and fifth digits counting from the right can be taken as hash address
H(3111) = 78 // 3111 to be placed at address 78
H(2345) = 99 // 2345 to be placed at address 99

3) Multiplication Method

Step 1: Choose a constant 'A' such that 0 < A < 1
Step 2: Multiply the key by A
Step 3: Extract the fractional part of "k*A"
Step 4: Multiply the result of Step 3 by the size of the hash table 'm'

The hash function is given by H(k) = | ((k * A) mod 1) * m |

Note : Common value used for A = (srqt(5) - 1) / 2 = 0.6180339887

Example: Consider the key to be stored is 12345, Assuming the size of hash table is 1000, the appropriate location for the key using multiplication method is
H(12345) = | ((12345 * 0.618033) mod 1) * 1000 |
= | (7629.617385 mod 1) * 1000 |
= | 0.617385 * 1000 |
= | 617.385 |
= 617

4) Folding Method

Step 1: Divide the key value into a number of parts i.e. divide 'k' into $k_1, k_2, \ldots k_n$ where each part, except possibly the last, has the same number of digits as the required hash address.

Step 2: Add the individual parts by ignoring the last carry to get the hash index
i.e. $H(k) = k_1 + k_2 + \ldots\ldots + k_n$ by ignoring the last carry.

There are two types of folding methods
a) Shift Folding - Shift all parts except for the last one, so that the least significant bit of each part lines up with corresponding bit of the last part.
b) Folding at the Boundaries - Reverses every other partition before adding

Example: Consider the data to be stored in the hash table to be 12320324111220. Then using the folding method data is stored as shown below.
Assuming the size of the hash table to be 1000, 12320324111220 is divided into the following parts
k_1 = 123
k_2 = 203
k_3 = 241
k_4 = 112
k_5 = 20
Shift Folding Method
H(k) = 123 + 203 + 241 + 112 + 20 = 699 // 12320324111220 to be placed at address 699
Folding at the Boundaries
H(k) = 123 + **302** + 241 + **211** + 20 = 897 // 12320324111220 to be placed at address 897

Collision Resolution Techniques

Collision occurs when the hash function maps two different keys to the same location. However, two records cannot be stored in the same location. Therefore a method to solve this collision called collision resolution technique is applied.

Collision Resolution by Open Addressing

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position. In this technique, all the values are stored in the hash table. The hash table contains two types of values: ***sentinel values*** **(e.g., –1) and *data values.*** The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If free location is not found, then we have an OVERFLOW condition.

The process of examining memory locations in the hash table is called ***probing.*** Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.

1) Linear Probing

In this technique, if a value is already stored at a location generated by H(k), then the following hash function is used to resolve the collision:

$H(k, i) = [H'(k) + i] \bmod m$

Where m is the size of the hash table, H'(k) = (k mod m), and 'i' is the probe number that varies from 0 to m–1.

Therefore, for a given key k, first the location generated by H'(k) mod m is probed because for the first time i=0. If the location is free, the value is stored in it, else the second probe generates the address of the location given by [H'(k) + 1]mod m. Similarly, if the location is occupied, then subsequent probes generate the address as [H'(k) + 2]mod m, [H'(k) + 3]mod m, [H'(k) + 4]mod m, [H'(k) + 5]mod m, and so on, until a free location is found.

Example: Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81, 92 and 104

The initial hash table is

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step 1 Key = 72

$H(72, 0) = (\overset{k}{72} \bmod \overset{M}{10} + 0) \bmod 10 = (2) \bmod 10 = 2$

Since T[2] is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 Key = 27
H(27, 0) = (27 mod 10 + 0) mod 10 = (7) mod 10 = 7
Since T[7] is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key = 36
H(36, 0) = (36 mod 10 + 0) mod 10 = (6) mod 10 = 6
Since T[6] is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 Key = 24
H(24, 0) = (24 mod 10 + 0) mod 10 = (4) mod 10 = 4
Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5 Key = 63
H(63, 0) = (63 mod 10 + 0) mod 10 = (3) mod 10 = 3
Since T[3] is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 Key = 81
H(81, 0) = (81 mod 10 + 0) mod 10 = (1) mod 10 = 1
Since T[1] is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7 Key = 92
H(92, 0) = (92 mod 10 + 0) mod 10 = (2) mod 10 = 2
Now T[2] is occupied, so we cannot store the key 92 in T[2].
Therefore, try again for the next location.
Thus probe, i = 1, this time.
H(92, 1) = (92 mod 10 + 1) mod 10 = (2 + 1) mod 10 = 3
Now T[3] is occupied, so we cannot store the key 92 in T[3].
Therefore, try again for the next location.
Thus probe, i = 2, this time.
H(92, 2) = (92 mod 10 + 2) mod 10 = (2 + 2) mod 10 = 4
Now T[4] is occupied, so we cannot store the key 92 in T[4].
Therefore, try again for the next location.
Thus probe, i = 3, this time.
H(92, 3) = (92 mod 10 + 3) mod 10 = (2 + 3) mod 10 = 5
Since T[5] is vacant, insert key 92 at this location.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	-1	-1

Step 8 Key = 104

H(104, 0) = (104 mod 10 + 0) mod 10 = (4) mod 10 = 4

Now T[4] is occupied, so we cannot store the key 104 in T[4].

Therefore, try again for the next location.

Thus probe, i = 1, this time.

H(104, 1) = (104 mod 10 + 1) mod 10 = (4 + 1) mod 10 = 5

Now T[5] is occupied, so we cannot store the key 104 in T[5].

Therefore, try again for the next location.

Thus probe, i = 2, this time.

H(104, 2) = (104 mod 10 + 2) mod 10 = (4 + 2) mod 10 = 6

Now T[6] is occupied, so we cannot store the key 104 in T[6].

Therefore, try again for the next location.

Thus probe, i = 3, this time.

H(104, 3) = (104 mod 10 + 3) mod 10 = (4 + 3) mod 10 = 7

Now T[7] is occupied, so we cannot store the key 104 in T[7].

Therefore, try again for the next location.

Thus probe, i = 4, this time.

H(104, 4) = (104 mod 10 + 4) mod 10 = (4 + 4) mod 10 = 8

Since T[8] is vacant, insert key 104 at this location.

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	92	36	27	104	-1

Searching a Value using Linear Probing

The procedure for searching a value in a hash table is same as for storing a value in a hash table. While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. If the key does not match, then the search function begins a sequential search of the array that continues until:

- ✓ the value is found, or
- ✓ the search function encounters a vacant location in the array, indicating that the value is not present, or
- ✓ the search function terminates because it reaches the end of the table and the value is not present.

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Disadvantages

- ✓ More complex insert, find, remove methods
- ✓ Primary clustering phenomenon - items tend to cluster together in the array, i.e. one part of the table gets quite dense, even though another part may have relatively few items. As clustering gets worse, inserting a key takes longer time because it must step all the way through a cluster to find a vacant position. Similarly as clustering gets worse, finding a key takes longer tie since elements get placed further and further from their correct hashed index.

Quadratic Probing

In this technique, if a value is already stored at a location generated by H(k), then the following hash function is used to resolve the collision:

$H(k, i) = [H'(k) + C_1 i + C_2 i^2] \bmod m$

where m is the size of the hash table, $H'(k) = (k \bmod m)$, i is the probe number that varies from 0 to m–1, and C_1 and C_2 are constants such that C_1 and $C_2 \neq 0$.
Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search. For a given key k, first the location generated by H'(k) mod m is probed. If the location is free, the value is stored in it, else subsequent locations probed are offset by factors that depend in a quadratic manner on the probe number i.

Example: Consider a hash table of size 10. Using linear probing, insert the keys 72, 27, 36, 24, 63, 81 and 101. Take $C_1 = 1$ and $C_2 = 3$
The initial hash table is

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step 1 Key = 72
H(72, 0) = [72 mod 10 + 1 * 0 + 3 * 0] mod 10 = [72 mod 10] mod 10 = 2 mod 10 = 2
Since T[2] is vacant, insert the key 72 in T[2].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 Key = 27
H(27, 0) = [27 mod 10 + 1 * 0 + 3 * 0] mod 10 = [27 mod 10] mod 10 = 7 mod 10 = 7
Since T[7] is vacant, insert the key 27 in T[7].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key = 36
H(36, 0) = [36 mod 10 + 1 * 0 + 3 * 0] mod 10 = [36 mod 10] mod 10 = 6 mod 10 = 6
Since T[6] is vacant, insert the key 36 in T[6].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 Key = 24
H(24, 0) = [24 mod 10 + 1 * 0 + 3 * 0] mod 10 = [24 mod 10] mod 10 = 4 mod 10 = 4
Since T[4] is vacant, insert the key 24 in T[4].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5 Key = 63
H(63, 0) = [63 mod 10 + 1 * 0 + 3 * 0] mod 10 = [63 mod 10] mod 10 = 3 mod 10 = 3
Since T[3] is vacant, insert the key 63 in T[3].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 Key = 81
H(81,0) = [81 mod 10 + 1 * 0 + 3 * 0] mod 10 = [81 mod 10] mod 10 = 81 mod 10 = 1
Since T[1] is vacant, insert the key 81 in T[1].

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7 Key = 101
H(101,0) = [101 mod 10 + 1 * 0 + 3 * 0] mod 10 = [101 mod 10 + 0] mod 10 = 1 mod 10 = 1
Since T[1] is already occupied, the key 101 cannot be stored in T[1].
Therefore, try again for the next location.

Thus probe, i = 1, this time.

H(101,1) = [101 mod 10 + 1 * 1 + 3 * 1] mod 10
= [101 mod 10 + 1 + 3] mod 10
= [101 mod 10 + 4] mod 10
= [1 + 4] mod 10
= 5 mod 10
= 5

Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	101	36	27	-1	-1

Searching a Value using Quadratic Probing
The procedure for searching a value in a hash table is same as for storing a value in a hash table. While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. If the key does not match, then the search function begins a sequential search of the array that continues until:

- ✓ the value is found, or
- ✓ the search function encounters a vacant location in the array, indicating that the value is not present, or
- ✓ the search function terminates because it reaches the end of the table and the value is not present.

Disadvantages

- ✓ One of the major drawbacks of quadratic probing is that a sequence of successive probes may only explore a fraction of the table, and this fraction may be quite small. If this happens, then we will not be able to find an empty location in the table despite the fact that the table is by no means full.
- ✓ Secondary clustering - It means that if there is a collision between two keys, then the same probe sequence will be followed for both. With quadratic probing, the probability for multiple collisions increases as the table becomes full.

Double Hashing

This technique uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function, hence the name *double hashing*. In double hashing, we use two hash functions rather than a single function. The hash function in the case of double hashing can be given as:
$H(k, i) = [H_1(k) + iH_2(k)] \bmod m$, where
'm' is the size of the hash table
$H_1(k) = k \bmod m$
$H_2(k) = k \bmod m'$
'i' is the probe number and m' is chosen to be m-1 or m-2

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

To insert a key k in the hash table, we first probe the location given by applying [H_1(k) mod m] because during the first probe, i = 0. If the location is vacant, the key is inserted into it, else subsequent probes generate locations that are at an offset of [H_2(k) mod m] from the previous location. Since the offset may vary with every probe depending on the value generated by the second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.

Pros and Cons
Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well as secondary clustering.

Example:
Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.
Take H_1 = (k mod 10) and H_2 = (k mod 8).

Here m=10, The initial hash table is

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Step 1 Key = 72
H(72, 0) = [72 mod 10 + (0 * 72 mod 8)] mod 10 = [2 + (0 * 0)] mod 10 = 2 mod 10 = 2
Since T[2] is vacant, insert the key 72 in T[2]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	-1	-1	-1

Step 2 Key = 27
H(27, 0) = [27 mod 10 + (0 * 27 mod 8)] mod 10 = [7 + (0 * 3)] mod 10 = 7 mod 10 = 7
Since T[7] is vacant, insert the key 27 in T[7].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	-1	27	-1	-1

Step 3 Key = 36
H(36, 0) = [36 mod 10 + (0 * 36 mod 8)] mod 10 = [6 + (0 * 4)] mod 10 = 6 mod 10 = 6
Since T[6] is vacant, insert the key 36 in T[6].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	-1	-1	36	27	-1	-1

Step 4 Key = 24
H(24, 0) = [24 mod 10 + (0 * 24 mod 8)] mod 10 = [4 + (0 * 0)] mod 10 = 4 mod 10 = 4
Since T[4] is vacant, insert the key 24 in T[4].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	-1	24	-1	36	27	-1	-1

Step 5 Key = 63
H(63, 0) = [63 mod 10 + (0 * 63 mod 8)] mod 10 = [3 + (0 * 7)] mod 10 = 3 mod 10 = 3
Since T[3] is vacant, insert the key 63 in T[3].

0	1	2	3	4	5	6	7	8	9
-1	-1	72	63	24	-1	36	27	-1	-1

Step 6 Key = 81
H(81, 0) = [81 mod 10 + (0 * 81 mod 8)] mod 10 = [1 + (0 * 1)] mod 10 = 1 mod 10 = 1
Since T[1] is vacant, insert the key 81 in T[1].

0	1	2	3	4	5	6	7	8	9
-1	81	72	63	24	-1	36	27	-1	-1

Step 7 Key = 92
H(92, 0) = [92 mod 10 + (0 * 92 mod 8)] mod 10 = [2 + (0 * 4)] mod 10 = 2 mod 10 = 2
Now T[2] is occupied, so we cannot store the key 92 in T[2].
Therefore, try again for the next location.

Thus probe, i = 1, this time.
Key = 92
H(92, 1) = [92 mod 10 + (1 * 92 mod 8)] mod 10 = [2 + (1 * 4)] mod 10 = (2 + 4) mod 10
= 6 mod 10 = 6
Now T[6] is occupied, so we cannot store the key 92 in T[6].
Therefore, try again for the next location.

Thus probe, i = 2, this time.
Key = 92
H(92, 2) = [92 mod 10 + (2 * 92 mod 8)] mod 10 = [2 + (2 * 4)] mod 10 = [2 + 8] mod 10
= 10 mod 10 = 0
Since T[0] is vacant, insert the key 92 in T[0].

0	1	2	3	4	5	6	7	8	9
92	81	72	63	24	-1	36	27	-1	-1

Step 8 Key = 101
H(101, 0) = [101 mod 10 + (0 * 101 mod 8)] mod 10 = [1 + (0 * 5)] mod 10 = 1 mod 10 = 1
Now T[1] is occupied, so we cannot store the key 101 in T[1].
Therefore, try again for the next location.

Thus probe, i = 1, this time.
Key = 101
H(101, 1) = [101 mod 10 + (1 * 101 mod 8)] mod 10 = [1 + (1 * 5)] mod 10
= [1 + 5] mod 10 = 6
Now T[6] is occupied, so we cannot store the key 101 in T[6].
Therefore, try again for the next location.

Thus probe, i = 2, this time.
Key = 101
H(101, 2) = [101 mod 10 + (2 * 101 mod 8)] mod 10 = [1 + (2 * 5)] mod 10
= [1 + 10] mod 10 = 1
Now T[1] is occupied, so we cannot store the key 101 in T[1].
Therefore, try again for the next location.

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Thus probe, i = 3, this time.
Key = 101
H(101, 3) = [101 mod 10 + (3 * 101 mod 8)] mod 10 = [1 + (3 * 5)] mod 10
= [1 + 15] mod 10 = 1

Now T[6] is occupied, so we cannot store the key 101 in T[6].
Therefore, try again for the next location.

Thus probe, i = 4, this time.
Key = 101
H(101, 4) = [101 mod 10 + (4 * 101 mod 8)] mod 10 = [1 + (4 * 5)] mod 10
= [1 + 20] mod 10 = 1
Now T[1] is occupied, so we cannot store the key 101 in T[1].
Therefore, try again for the next location.

Note: Continuing this way, it would be hard to find a vacant space for the key. Although double hashing is a very efficient algorithm, it always requires 'm' to be a prime number. In the example taken, m=10, which is not a prime number and hence the degradation.

Rehashing
When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.
All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table. Though rehashing seems to be a simple process, it is quite expensive and must therefore not be done frequently.

Consider the hash table of size 5 given below. The hash function used is H(x) = x % 5. Rehash the entries into to a new hash table.

0	1	2	3	4
-1	26	31	43	17

Note that the new hash table will have 10 locations.

0	1	2	3	4	5	6	7	8	9
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

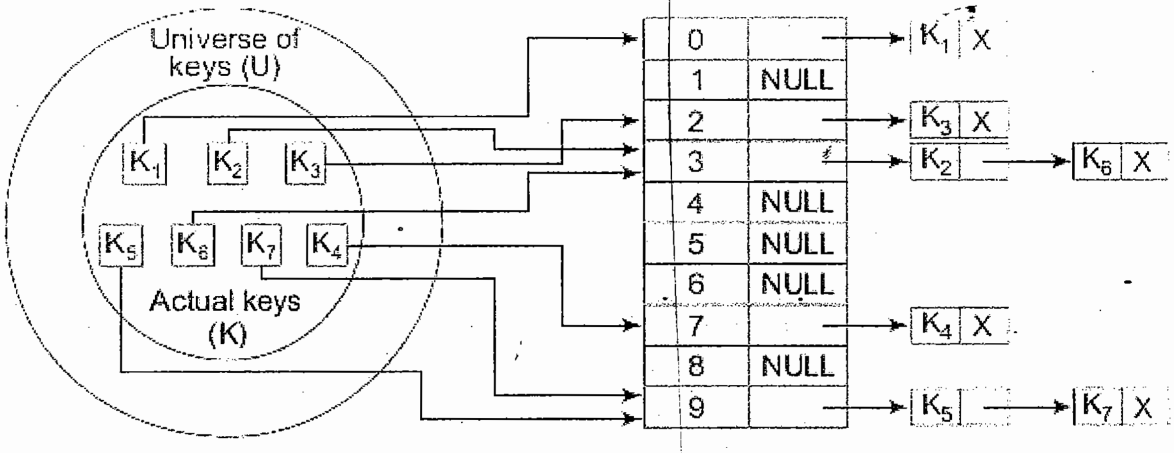
Now, rehash the key values from the old hash table into the new one using hash function H(x) = x % 10.

0	1	2	3	4	5	6	7	8	9
-1	31	-1	43	-1	-1	26	17	-1	-1

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Collision Resolution by Chaining

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. That is, location l in the hash table points to the head of the linked list of all the key values that hashed to l. However, if no key value hashes to l, then location l in the hash table contains NULL.
The following figure shows how the key values are mapped to a location in the hash table and stored in a linked list that corresponds to that location.



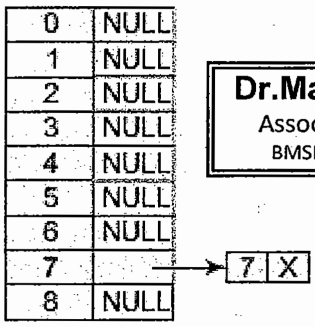
Example: Insert the keys 7, 24, 18, 52, 36, 54, 11, and 23 in a chained hash table of 9 memory locations. Use H(k) = k mod m.
In this case, m=9. Initially, the hash table can be given as:

Initial Hash Table

0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL

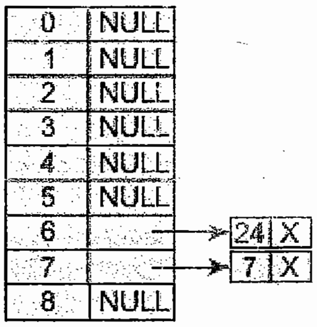
Step 1 Key = 7
h(k) = 7 mod 9
= 7

Create a linked list for location 7 and store the key value 7 in it as its only node.



Step 2 Key = 24
h(k) = 24 mod 9
= 6

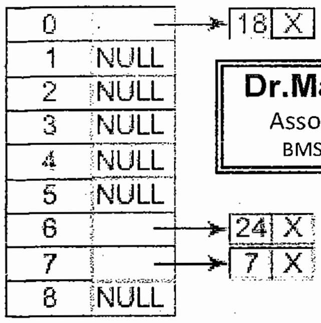
Create a linked list for location 6 and store the key value 24 in it as its only node.



Step 3 Key = 18

h(k) = 18 mod 9 = 0

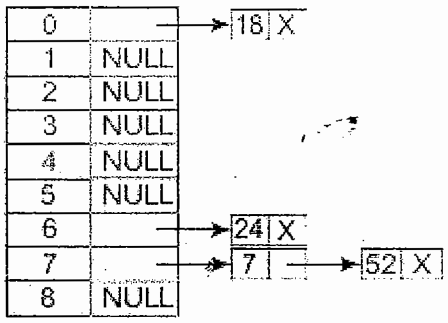
Create a linked list for location 0 and store the key value 18 in it as its only node.



Step 4 Key = 52

h(k) = 52 mod 9 = 7

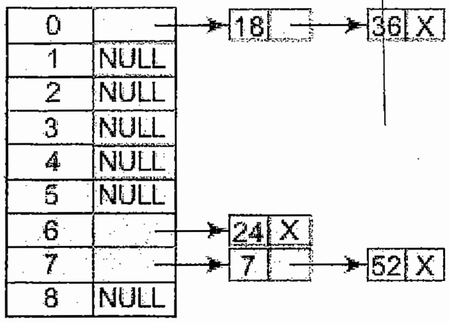
Insert 52 at the end of the linked list of location 7.



Step 5: Key = 36

h(k) = 36 mod 9 = 0

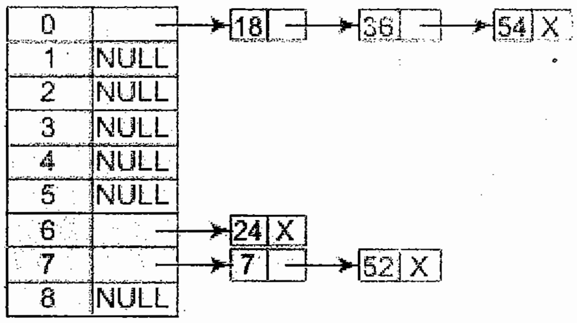
Insert 36 at the end of the linked list of location 0.



Step 6: Key = 54

h(k) = 54 mod 9 = 0

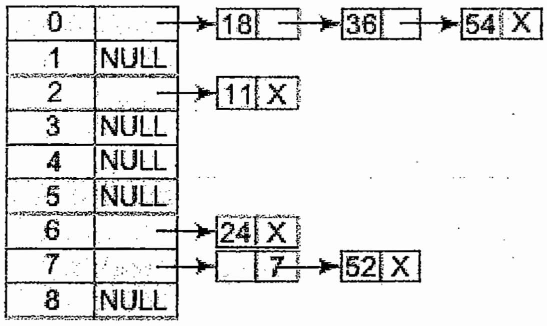
Insert 54 at the end of the linked list of location 0.



Step 7: Key = 11

h(k) = 11 mod 9 = 2

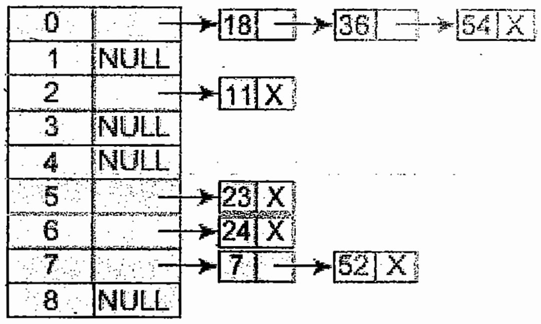
Create a linked list for location 2 and store the key value 11 in it as its only node.



Step 8: Key = 23

h(k) = 23 mod 9 = 5

Create a linked list for location 5 and store the key value 23 in it as its only node.



Pros and Cons

- ✓ The main advantage of using a chained hash table is that it remains effective even when the number of key values to be stored is much higher than the number of locations in the hash table. However, with the increase in the number of keys to be stored, the performance of a chained hash table does degrade gradually.

- ✓ The other advantage of using chaining for collision resolution is that its performance, unlike quadratic probing, does not degrade when the table is more than half full. This technique is absolutely free from clustering problems and thus provides an efficient mechanism to handle collisions.
- ✓ Chained hash tables inherit the disadvantages of linked lists.
    1. To store a key value, the space overhead of the next pointer in each entry can be significant.
    2. Traversing a linked list has poor cache performance, making the processor cache ineffective.

Pros and Cons of Hashing

- ✓ No extra space is required to store the index as in the case of other data structures.
- ✓ Hash table provides fast data access and has an added advantage of rapid updates.
- ✓ Inserting and retrieving data values usually lacks locality and sequential retrieval by key. This makes insertion and retrieval of data values even more random.
- ✓ Choosing an effective hash function is more of an art than a science. It is not uncommon to create a poor hash function.

Applications of Hashing

Hash tables are widely used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information.

Example:

1) Hashing is used for database indexing.
2) Hashing technique is used to implement compiler symbol tables in C++.
3) Hashing is also widely used for Internet search engines.

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Real World Applications of Hashing

1) CD Databases

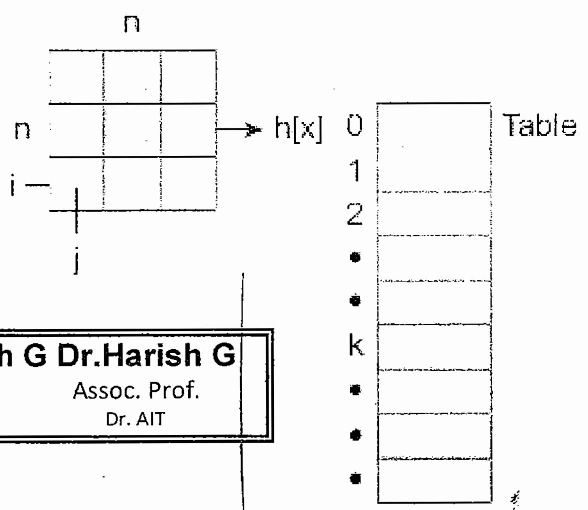
- ✓ It is desirable to have a world-wide CD database so that when users put their disk in the CD player, they get a full table of contents on their own computer's screen.
- ✓ These tables are not stored on the disks themselves, rather this information is downloaded from the database.
- ✓ Basically, a big number is created from the track lengths, also known as a 'signature'. This signature is used to identify a particular CD. The signature is a value obtained by hashing.

2) Drivers Licenses / Insurance Cards

- ✓ The driver's license numbers or insurance card numbers are created using hashing from data items that never change: date of birth, name, etc.

3) Sparse Matrix

- ✓ A sparse matrix is a two-dimensional array in which most of the entries contain a 0.
- ✓ Using a 2D array to store this would waste a lot of memory.
- ✓ The non zero elements of the sparse matrix can be stored in a 1D array using hashing.



- ✓ From the coordinates for every non zero element (i,j), we can determine the index 'k' by using a hash function k = H(i,j), for some function H.

4) File Signatures

- ✓ File signatures provide a compact means of identifying files.
- ✓ We use a function, h[x], the file signature, which is a property of the file.
- ✓ Although we can store files by name, signatures provide a compact identity to files.
- ✓ Since a signature depends on the contents of a file, if any change is made to the file, then the signature will change. In this way, the signature of a file can be used as a quick verification to see if anyone has altered the file, or if it has lost a bit during transmission.

5) Game Boards

- ✓ In the game board for tic-tac-toe or chess, a position in a game may be stored using a hash function.

6) Graphics

- ✓ In graphics, a central problem is the storage of objects in a scene or view. For this, we organize our data by hashing. Hashing can be used to make a grid of appropriate size, an ordinary vertical–horizontal grid.
- ✓ A grid is nothing but a 2D array, and there is a one-to-one correspondence when we move from a 2D array to a 1D array.
- ✓ We store the grid as a 1D array as we did in the case of sparse matrices.

Hashing example program for collision resolution and searching using linear probing

```
#include<stdio.h>

#define MAXADDR 100

struct employee
{
        int empid;      // 4-digit key to determine employee records uniquely
        int age;
        char name[20];
}ht[MAXADDR];

int hash(int key)       // Hash function to convert 4-digit key to 2-digit index (hash address)
{
        int index;
        index = key % MAXADDR;
        return ( index );
}

void main( )
{
        int i, choice, count, key, age, index;
        char name[20];

        count = 0;

        // Initialize all the empid in hash table ht to -1
        for( i=0; i<MAXADDR; i++)
        {
                ht[i].empid = -1;
        }

        for( ; ; )
        {
                printf("1. Insert Record        2. Search Record        3.Exit\n");

                printf("Enter your choice\n");
                scanf("%d", &choice);

                switch(choice)
                {

                        case 1: if(count = = MAXADDR)
                                {
                                        printf("No Space Available\n");
                                }
                                else
                                {
```

```
                printf("Enter the 4-digit unique key for employee\n");
                scanf("%d", &key);

                printf("Enter the Employee name\n");
                gets(name);

                printf("Enter the age\n");
                scanf("%d", &age);

                index = hash(key);

                if( ht[index].empid = = -1) // Found free location (No Collision)
                {
                        ht[index].empid = key;
                        strcpy(ht[index].name, name);
                        ht[index].age = age;
                        count = count + 1;
                        break;
                }
                else // Collision Resolution (Linear Probing)
                {
                        for( i=1; i<MAXADDR; i++ )
                        {
                                index = ( hash(key) + i ) % MAXADDR;

                                if( ht[index].empid = = -1) // Found free location
                                {
                                        ht[index].empid = key;
                                        strcpy(ht[index].name, name);
                                        ht[index].age = age;
                                        count = count + 1;
                                        break;
                                }
                        }
                }
        }

        break;

case 2: printf("Enter the 4-digit unique key of employee to search\n");
        scanf("%d", &key);

        index = hash(key);

        if( ht[index].empid = = key) // Found Successfully
        {
                printf("Successful Search\n");
                printf("Name = %s\n", ht[index].name);
                printf("Age = %d\n", ht[index].age);
```

```
                    break;
                }
                else if( ht[index].empid = = -1 ) // Found Vacant Position
                {
                    printf("Unsuccessful Search\n");
                    printf("Key not found\n");

                    break;
                }
                else // Search using Linear Probing
                {
                    for( i=1; i<MAXADDR; i++ )
                    {
                        index = ( hash(key) + i ) % MAXADDR;

                        if( ht[index].empid = = key) // Found Successfully
                        {
                            printf("Successful Search\n");
                            printf("Name = %s\n", ht[index].name);
                            printf("Age = %d\n", ht[index].age);

                            break;
                        }
                        else if( ht[index].empid = = -1 ) // Found Vacant Position
                        {
                            printf("Unsuccessful Search\n");
                            printf("Key not found\n");

                            break;
                        }
                    }
                }

                printf("Unsuccessful Search\n");
                printf("Key not found\n");
                break;

            default: Exit (0);
        }
    }
}
```

Given a File of **N** employee records with a set **K** of Keys(4-digit) which uniquely determine the records in file **F**. Assume that file **F** is maintained in memory by a Hash Table(HT) of **m** memory locations with **L** as the set of memory addresses (2- digit) of locations in HT. Let the keys in **K** and addresses in **L** are Integers. Design and develop a Program in C that uses Hash function **H: K→L** as H(**K**)=**K** mod **m** (**remainder** method), and implement hashing technique to map a given key **K** to the address space **L**. Resolve the collision (if any) using **linear probing**.

```
# include <string.h>
#include<stdio.h>
#define MAXADDR 100

struct employee
{
        int empid;      // 4-digit key to determine employee records uniquely
        int age;
        char name[20];
}ht[MAXADDR], temp;

int hash(int key)       // Hash function to convert 4-digit key to 2-digit index (hash address)
{
        int index;
        index = key % MAXADDR;
        return ( index );
}

void main( )
{
        int n, i, count,index;
        FILE *fp;
// Create a File with 'N' employee records
        printf("Enter the number of employees\n");
        scanf("%d", &n);
        fp = fopen("source.txt", "w");
        for(i=0; i<n; i++)
        {
                printf("Enter the 4-digit unique key for employee\n");
                scanf("%d", &temp.empid);
                printf("Enter the age\n");
                scanf("%d", &temp.age);
                printf("Enter the Employee name\n");
                fflush(stdin);
                gets(temp.name);
                fprintf(fp, "%d\t%d\t%s\n", temp.empid, temp.age, temp.name);
        }
        fclose(fp);
// Initialize all the empid in hash table ht to -1
        for( i=0; i<MAXADDR; i++)
        {
                ht[i].empid = -1;
        }
```

Dr.Mahesh G Assoc. Prof. BMSIT & M **Dr.Harish G** Assoc. Prof. Dr. AIT

```
// To store the records from file to hash table using remainder method of hashing
        count = 0;
        fp = fopen("source.txt", "r");
        while(fscanf(fp, "%d%d%s", &temp.empid, &temp.age, temp.name)!= EOF)
        {
                if(count = = MAXADDR)
                {
                        printf("No Space Available\n");
                        break;
                }
                index = hash(temp.empid);
                if( ht[index].empid = = -1) // Found free location (No Collision)
                {
                        ht[index].empid = temp.empid;
                        ht[index].age = temp.age;
                        strcpy(ht[index].name, temp.name);
                        count = count + 1;
                }
                else // Collision Resolution (Linear Probing)
                {
                        printf("Collision found for employee id %d\n", temp.empid);
                        for( i=1; i<MAXADDR; i++ )
                        {
                                index = ( hash(temp.empid) + i ) % MAXADDR;

                                if( ht[index].empid = = -1) // Found free location
                                {
                                        ht[index].empid = temp.empid;
                                        ht[index].age = temp.age;
                                        strcpy(ht[index].name, temp.name);
                                        count = count + 1;
                                        printf("Collision Resolved for empid %d \n", temp.empid);
                                        break;
                                }
                        }
                }
        }
        fclose(fp);

// Print the contents of  hash table ht
        printf("HA \t Empid \t Age \t Name\n");
        for( i=0; i<MAXADDR; i++)
        {
                if(ht[i].empid != -1)
                {
                        printf("%d\t%d\t%d\t%s\n", i, ht[i].empid, ht[i].age, ht[i].name);
                }
        }
        getch( );
}
```