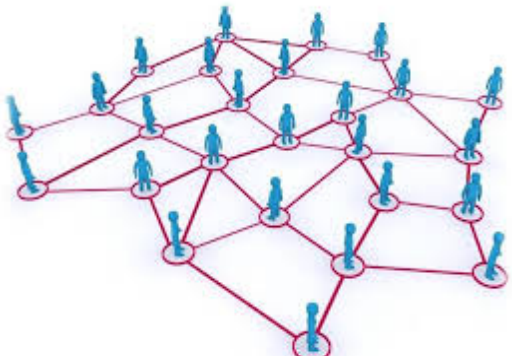
CONTENTS

Data Structure & Applications

Module V : Graphs, Files, Sorting Hashing



Dr. Ganga Holi,
Professor & Head, Dept. of ISE,
GLOBAL ACADEMY OF TECHNOLOGY, BENGALURU.

MODULE IV : TREES

MODULE V

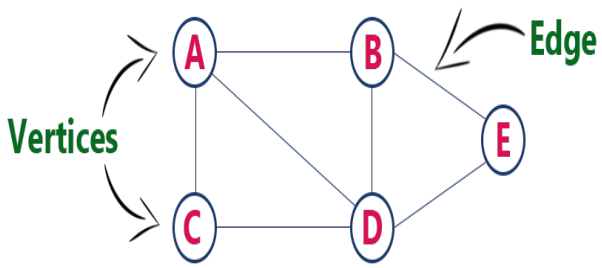
1. GRAPH

Graph is a nonlinear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is defined as follows...

Graph is a collection of vertices/nodes and arcs which connects vertices in the graph.

Generally, a graph **G** is represented as **G = (V, E)**, where **V is set of vertices** and **E is set of edges**.

Example: The following is a graph with 5 vertices and 6 edges. This graph G can be defined as G = (V, E), where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



1.1. Graph Terminology

We use the following terms in graph data structure...

Vertex: A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

Edge: An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Edges are three types.

1. **Undirected Edge -** An undirected egde is a bidirectional edge. If there is a undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. **Directed Edge -** A directed egde is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. **Weighted Edge -** A weighted egde is an edge with cost on it.

Undirected Graph: A graph with only undirected edges is said to be undirected graph.

Directed Graph: A graph with only directed edges is said to be directed graph.

Mixed Graph: A graph with undirected and directed edges is said to be mixed graph.

End vertices or Endpoints: The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.

Origin: If an edge is directed, its first endpoint is said to be origin of it.

Destination: If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Adjacent: If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, Two vertices A and B are said to be adjacent if there is an edge whose end vertices are A and B.

Incident: An edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.

Outgoing Edge: A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge: A directed edge is said to be incoming edge on its destination vertex.

Degree: Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree: Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree: Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

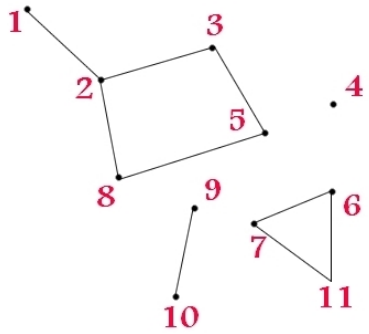
Parallel edges or Multiple edges: If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop : An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph: A graph is said to be simple if there are no parallel and self-loop edges.

Path: A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Cycle: A closed (simple) path, with no other repeated vertices or edges other than the starting and ending vertices.



In the above figure, 2, 3, 6, 8 & 6,7,11 form a cycle.

2. Graph Representation

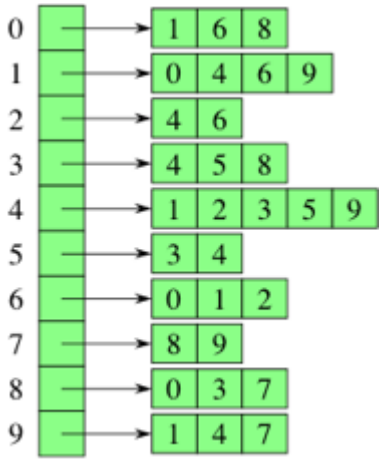
2.1. Adjacency Matrix

For a graph with |V||V||V| vertices, an **adjacency matrix** is a |V|×|V| matrix of 0s and 1s, where the entry in row i and column j is 1 if and only if the edge (i,j) is in the graph. If you want to indicate an edge weight, put it in the row i, column j entry, and reserve a special value (perhaps null) to indicate an absent edge. Zero can be used to represent no edge between two vertices. Here's the adjacency matrix for the graph:

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

2.2. Adjacency List

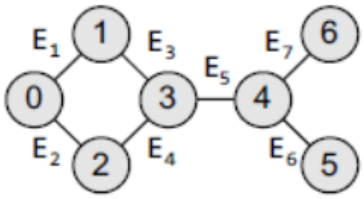
Representing a graph with **adjacency lists** combines adjacency matrices with edge lists. For each vertex i, store an array of the vertices adjacent to it. We typically have an array of |V| adjacency lists, one adjacency list per vertex. Here's an adjacency-list representation of the social network graph:



2.3. Adjacency Multilist

Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists. Adjacency multi-list is an edge-based rather than a vertex-based representation of graphs. A multi-list representation basically consists of two parts—a directory of nodes' information and a set of linked lists storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). Consider the undirected graph given below. The adjacency multi-list for the graph can be given as:

Edge1	→ 0→ 1 Edge 2 Edge 3
Edge2	→ 0→ 2 NULL Edge 4
Edge3	→ 1→ 3 NULL Edge 4
Edge4	→ 2→ 3 NULL Edge 5
Edge5	→ 3→ 4 NULL Edge 6
Edge6	→ 4→ 5 Edge 7 NULL
Edge7	→ 4→ 6 NULL NULL



Using the adjacency multi-list given above, the adjacency list for vertices can be constructed as shown below:

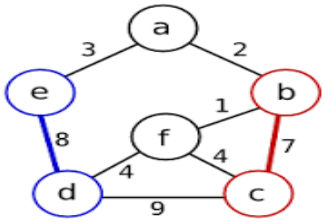
VERTEX	LIST OF EDGES

0	0 Edge 1, Edge 2
1	1 Edge 1, Edge 3
2	2 Edge 2, Edge 4
3	3 Edge 3, Edge 4, Edge 5
4	4 Edge 5, Edge 6, Edge 7
5	5 Edge 6
6	6 Edge 7

2.4. Weighted edges.

In many applications, each edge of a graph has an associated numerical value, called a weight. Usually, the edge weights are non-negative integers.
Weighted graphs may be either directed or undirected. These edges may represent distance from one vertex to another.

Adjacency matrix will have the weights for the edge (I,j) and ∞ for no edge



	a	b	c	d	e	f
a	0	2	∞	∞	3	∞
b	2	0	7	∞	∞	1
c	∞	7	0	9	∞	4
d	∞	∞	8	0	8	4
e	3	∞	∞	8	0	∞
f	∞	1	4	4	0	∞

2.5. Elementary Graph Operations

Graphs can traversed using Depth first search & Breadth first search. Depth First Search

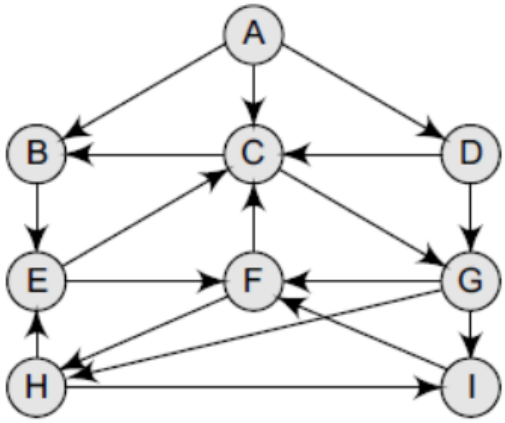
2.5.1. Depth First Search

The depth-first search algorithm (Fig. 13.22) progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.
In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbor of neighbor of A, and so on. During the execution of the

algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.
Depth First Search is done in the following manner.

```
dfs(vertex v)
        {
          visit(v);
           for each neighbor w of v
              if w is unvisited
              {
                        dfs(w);
               }
        }
```

Print all the nodes reachable from H for the given graph using DFS.



Adjacency lists

A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Solution

(a) Push H onto the stack.
STACK: H
(b) Pop and print the top element of the STACK, that is, H. Push all the neighbours of H onto the stack that are in the ready state. The STACK now becomes
PRINT: H STACK: E, I
(c) Pop and print the top element of the STACK, that is, I. Push all the neighbours of I onto the stack that are in the ready state. The STACK now becomes
PRINT: I STACK: E, F
(d) Pop and print the top element of the STACK, that is, F. Push all the neighbours of F onto the stack that are in the ready state. (Note F has two neighbours, C and H. But only C will be added, as H is not in the ready state.) The STACK now becomes
PRINT: F STACK: E, C
(e) Pop and print the top element of the STACK, that is, C. Push all the neighbours of C onto the stack that are in the ready state. The STACK now becomes
PRINT: C STACK: E, B, G
(f) Pop and print the top element of the STACK, that is, G. Push all the neighbours of G onto the

stack that are in the ready state. Since there are no neighbours of G that are in the ready state, no push operation is performed. The STACK now becomes
PRINT: G STACK: E, B
(g) Pop and print the top element of the STACK, that is, B. Push all the neighbours of B onto the stack that are in the ready state. Since there are no neighbours of B that are in the ready state, no push operation is performed. The STACK now becomes
PRINT: B STACK: E
(h) Pop and print the top element of the STACK, that is, E. Push all the neighbours of E onto the stack that are in the ready state. Since there are no neighbours of E that are in the ready state, no push operation is performed. The STACK now becomes empty.
PRINT: E STACK:
Since the STACK is now empty, the depth-first search of G starting at node H is c omplete and the nodes which were printed are: H, I, F, C, G, B, E. These are the nodes which are reachable from the node H.

Using Linked List for graph(adjacency List) :

```
#define FALSE 0
#define TURE 1
int visited[MAX];
void dfs(int v)
{       NODE *w;
        visited[v]=TRUE;
        printf("%d  ", v);
        for(w=graph[v];w; w→link)
        {
                if(!visited[w→vertex])
                        dfs(w→vertex);
    }
```

Applications of Depth-First Search Algorithm

Depth-first search is useful for:
- ✓ Finding a path between two specified nodes, u and v, of an unweighted graph.
- ✓ Finding a path between two specified nodes, u and v, of a weighted graph.
- ✓ Finding whether a graph is connected or not.
- ✓ Computing the spanning tree of a connected graph.

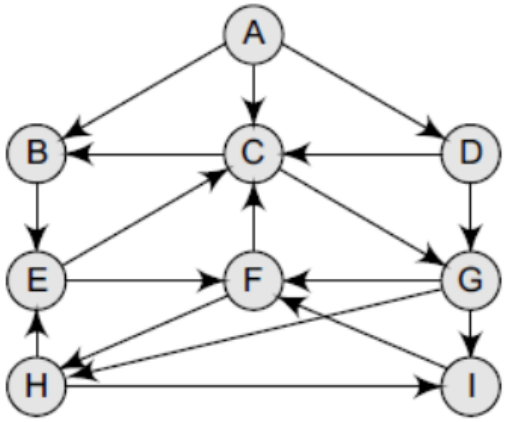
2.5.2. Breadth First Search

Breadth First Search (BFS) algorithm traverses a graph in a breadthwise motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and

explores all the neighbouring nodes. That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable STATUS to represent the current state of the node.

It employs the following rules.
Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.
Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.



Adjacency lists
A: B, C, D
B: E
C: B, G
D: C, G
E: C, F
F: C, H
G: F, H, I
H: E, I
I: F

Algorithm/function for BFS

```
void bfs(int s)
{
        NODE w; front=rear=NULL;
        printf(" %d",p);
        addq(s);
        visited[s]= TRUE;
        while(front)
        {       s=deleteq();
                for(w=graph[s];w;w→link)
                {       if((!visited[w→vertex])
```

```
                    {
                        addq(w→vertex);
                        visited[w→vertex]= TRUE;
                    }
                }
            }

        }
```

Print all the nodes reachable from A for the given graph using BFS.

Solution

The minimum path P can be found by applying the breadth-first search algorithm that begins at city A and ends when I is encountered. During the execution of the algorithm, we use two arrays: QUEUE and ORIG. While QUEUE is used to hold the nodes that have to be processed, ORIG is used to keep track of the origin of each edge. Initially, FRONT = REAR = –1. The algorithm for this is as follows:

(a) Add A to QUEUE and add NULL to ORIG.

(b) Dequeue a node by setting FRONT = FRONT + 1 (remove the FRONT element of QUEUE) and enqueuer the neighbours of A. Also, add A as the ORIG of its neighbours.

FRONT = 1 QUEUE = A B C D
REAR = 3 ORIG = \0 A A A

(c) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of B. Also, add B as the ORIG of its neighbours.

FRONT = 2 QUEUE = A B C D E
REAR = 4 ORIG = \0 A A A B

(d) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of C. Also, add C as the ORIG of its neighbours. Note that C has two neighbours B and G. Since B has already been added to the queue and it is not in the Ready state, we will not add B and only add G.

FRONT = 3 QUEUE = A B C D E G
REAR = 5 ORIG = \0 A A A B C

(e) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of D. Also,

add D as the ORIG of its neighbours. Note that D has two neighbours C and G. Since both of them have already been added to the queue and they are not in the Ready state, we will not add them again.

FRONT = 4 QUEUE = A B C D E G
REAR = 5 ORIG = \0 A A A B C

(f) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of E. Also, add E as the ORIG of its neighbours. Note that E has two neighbours C and F. Since C has already been added to the queue and it is not in the Ready state, we will not add C and add only F.

FRONT = 5 QUEUE = A B C D E G F
REAR = 6 ORIG = \0 A A A B C E

(g) Dequeue a node by setting FRONT = FRONT + 1 and enqueue the neighbours of G. Also, add G as the ORIG of its neighbours. Note that G has three neighbours F, H, and I.

FRONT = 6 QUEUE = A B C D E G F H I
REAR = 9 ORIG = \0 A A A B C E G G

Since F has already been added to the queue, we will only add H and I. As I is our final destination, we stop the execution of this algorithm as soon as it is encountered and added to the QUEUE.

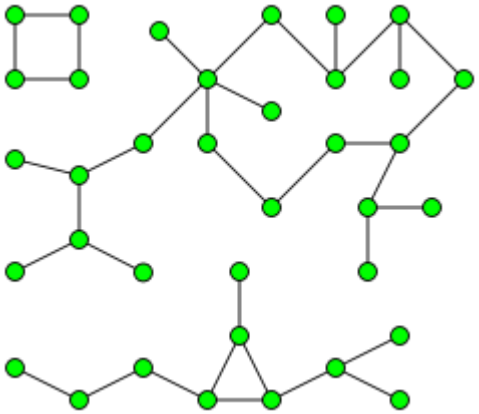
Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- ✓ Finding all connected components in a graph G.
- ✓ Finding all nodes within an individual connected component.
- ✓ Finding the shortest path between two nodes, u and v, of an unweighted graph.
- ✓ Finding the shortest path between two nodes, u and v, of a weighted graph.

2.5.3. Connected components

In graph theory, a **connected component** (or just **component**) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. For example, the graph shown in the illustration on the right has three connected components. A vertex with no incident edges is itself a connected component. A graph that is itself connected has exactly one connected component, consisting of the whole graph.

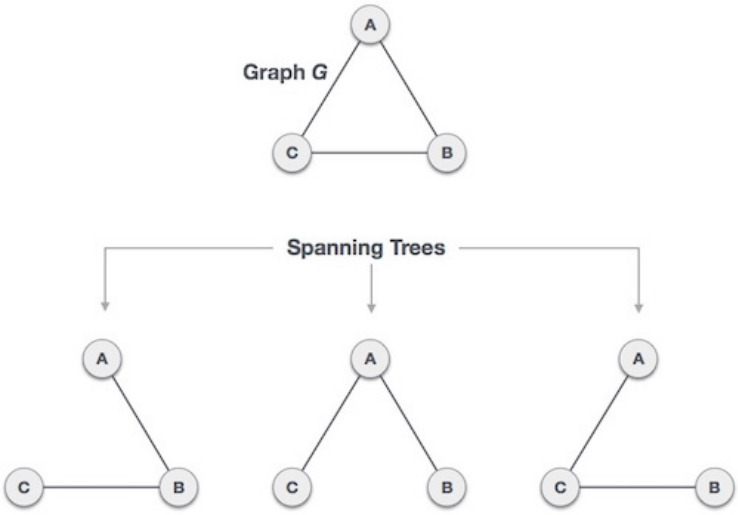


Graph with three connected components.

2.5.4. Spanning Trees

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected..

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.



2.5.4.1. General Properties of Spanning Tree

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

2.5.4.2. Application of Spanning Tree

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

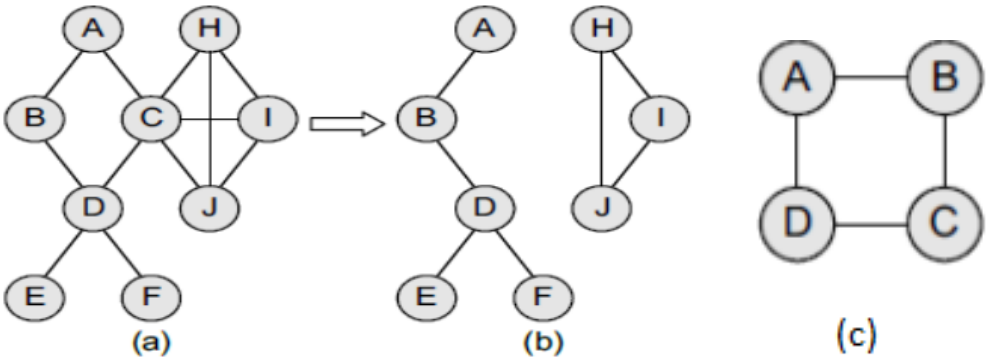
2.5.5. Biconnected Components

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.

A bi-connected graph is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected. By definition,

- ✓ A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.
- ✓ In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.

Note that the graph shown in Fig. (a) is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph Fig. (b).



Graph shown in Fig (C) is Bi-connected Graph.

2. SORTING

2.1. Insertion sort

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending). Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using following steps...

- **Step 1:** Assume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.
- **Step 2:** Consider first element from the unsorted list and insert that element into the sorted list in order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted list aremoved into the sorted list.

Sorting algorithm : Following is the sample code for insertion sort

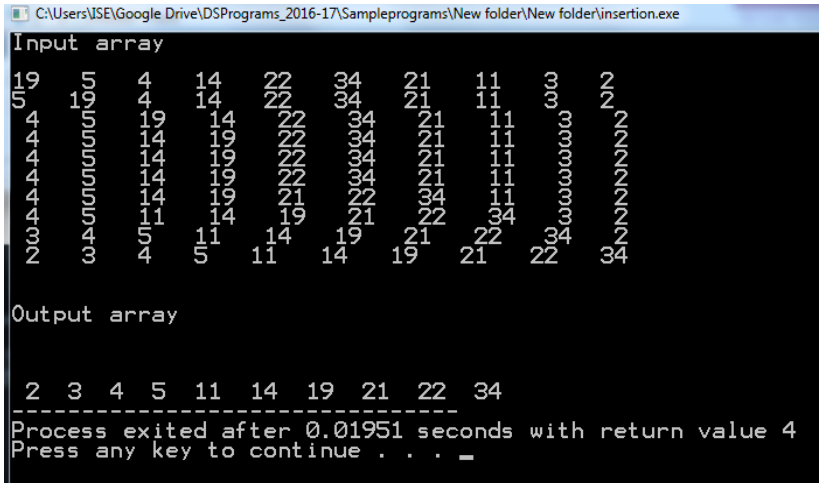
```
//Insertion sort logic
Void Insertion_sort(int a[], int n)
{
  for(i = 1; i<=n; i++)
  {
    v = a[i];
    j = i-1;
    while ((v < a[j]) && (j > 0))
    {  a[j+1] = a[j];
      j = j - 1;
    }
    a[j+1] = v;
  }
}
```

Complete C Program for the insertion sort is as follows

```
#include<stdio.h>
int main()
{    int i,j,v,k;
     int a[]={19,5,4,14, 22, 34, 21, 11, 3,2 };
     printf("Input array\n\n");
     for(k=0;k<10;k++)
     {
                          printf("%d   ",a[k]);
```

```
    }
    printf("\n");
  for(i =1; i<10; i++)
    {
    v = a[i];
    j = i-1;
    while ((v < a[j]) && (j >= 0))
    {
            a[j+1] = a[j];
            j = j - 1;
    }
    a[j+1] = v;
            for(k=0;k<10;k++)
            {
                    printf("%d   ",a[k]);
            }
            printf("\n ");
  }
  printf("\n\nOutput array\n\n");
  printf("\n\n ");
    for(i=0;i<10;i++)
    {
            printf( "%d  ",a[i]);
    }
}
```

Example: Input array elements are: 19, 5, 4, 14, 22, 34, 21, 11, 3, 2



Insertion sort algorithm tracing for the given data 39 9 45 63 18 81 1 8 54 72 36 9 39 45 63 18 81 1 8 54 72 36

Solution

39	9	45	63	18	81	1	8	54	72	36

A[0] is the only element in sorted list

39	9	45	63	19	81	1	8	54	72	36

After 1st Pass

9	39	45	63	19	81	1	8	54	72	36

9	39	45	63	19	81	1	8	54	72	36

After 2nd Pass

9	39	45	63	19	81	1	8	54	72	36

9	39	45	63	19	81	1	8	54	72	36

After 3rd Pass

9	39	45	63	19	81	1	8	54	72	36

9	39	45	63	19	81	1	8	54	72	36

After 4th Pass

9	19	39	45	63	81	1	8	54	72	36

9	19	39	45	63	81	1	8	54	72	36

After 5th Pass

9	19	39	45	63	81	1	8	54	72	36

After 6th Pass

1	9	19	39	45	63	81	8	54	72	36

After 7th Pass

1	8	9	19	39	45	63	81	54	72	36

1	8	9	19	39	45	63	81	54	72	36

After 8th Pass

1	8	9	19	39	45	54	63	81	72	36

After 9th pass

1	8	9	19	39	45	54	63	72	81	36

1	8	9	19	36	39	45	54	63	72	81

2.2. Radix Sort

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. (Here Radix is 26, 26 letters of alphabet). Specifically, the list of names is first sorted according to the first letter of each names, that is, the names are arranged in 26 classes. Intuitively, one might want to sort numbers on their most significant digit. But Radix sort do counter-intuitively by sorting on the least significant digits first. On the first pass entire numbers sort on the least significant digit and combine in an array. Then on the second pass, the entire numbers are sorted again on the second least-significant digits and combine in an array and so on.

Following example shows how Radix sort operates on seven 3-digits number.

Ist pass	1	2	3
329, 457, 657, 839, 436,720, 355, 322, 431, 641			
	720		
1	431	641	
2	322		
3			

4			
5	355		
6	436		
7	457	657	
8			
9	329	839	

2^{nd} pass	1	2	3
329, 457, 657, 839, 436,720, 355, 322, 431, 641			
0			
1			
2	720	322	329
3	431	436	839
4	641		
5	355	457	657
6			
7			
8			
9			

3^{rd} pass	1	2	3	4
329, 457, 657, 839, 436,720, 355, 322, 431, 641				
0				
1				
2				
3	322	329	355	
4	431	436	457	

5				
6	641	657		
7	720			
8	839			
9				

Sorted array is 322, 329, 355, 431, 436, 457, 641, 657, 720, 839.

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

In the above example, the first column is the input. The remaining shows the list after successive sorts on increasingly significant digits position. The code for Radix sort assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and d is the highest-order digit.

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

void countSort(int arr[], int n, int exp)
{    int output[n]; // output array
     int i, count[10] = { 0 };
```

```
        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
                count[(arr[i] / exp) % 10]++;
        for (i = 1; i < n; i++)
                count[i] += count[i - 1];
        // Build the output array
         for (i = n - 1; i >= 0; i--)
        {
                  output[count[(arr[i] / exp) % 10] - 1] = arr[i];
                count[(arr[i] / exp) % 10]--;
        }
    for (i = 0; i < n; i++)
            arr[i] = output[i];
 }

// The main function to that sorts arr[] of size n using Radix Sort

  void radixsort(int arr[], int n)
  {
        int m = getMax(arr, n);
        int exp;
        for (exp = 1; m / exp > 0; exp *= 10)
                countSort(arr, n, exp);
   }

  void print(int arr[], int n)
  {
      int i;
      for (i = 0; i < n; i++)
          printf("%d ", arr[i]);
   }


int main()
{
      int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
      int n = sizeof(arr) / sizeof(arr[0]);
      radixsort(arr, n);
      print(arr, n);
      return 0;
}
```

2.3. Address Calculation Sort

The Address Calculation Sorting makes use of **Hash Function Algorithm** for sorting a set of elements. These types of functions are generally known as Order Preserving Hashing Function. This function is

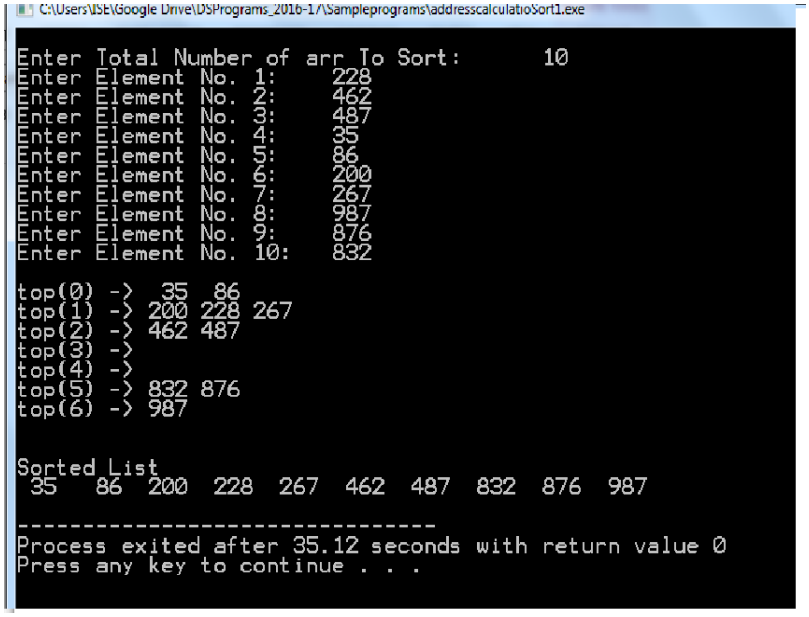
applied to all the elements to be sorted. According to the value of the hashing function, every element to be sorted is to be placed in a pre-defined set.

Every set is represented by a linked list. The starting address of each linked list can be maintained by an array of pointers. In every set, the elements have to be inserted in sorted order and, therefore, sorted linked lists needs to be taken. The time is dependent on the insertion time of elements in the sorted linked list. This algorithm is not an in-place sort but it is a **stable sort**.

```
#include<stdio.h>
#include<stdlib.h>
 #define MAX 55
 struct node
{       int data;
        struct node *link;
};
struct node *top[6];
int n, arr[MAX];
int maximum;

void address_sorting()
{       int i, temp;
        struct node *p;
        int address;
        for(i = 0; i <= 6; i++)
        {       top[i] = NULL;
        }
        for(i = 0; i < n; i++)
        {       address = hash_function(arr[i]);
                insert(arr[i], address);
        }
        printf("\n");
        for(i = 0; i <= 6; i++)
        {
                printf("top(%d) -> ", i);
                p = top[i];
                while(p != NULL)
                {       printf("%3d ", p->data);
                        p = p->link;
                }
                printf("\n");
        }
        printf("\n");     i = 0;
        for(temp = 0; temp <= 6; temp++)
        {       p = top[temp];
                while(p != NULL)
                {       arr[i++] = p->data;
                        p = p->link;
                }
        }
```

```
}

void insert(int num, int address)
{       struct node *q, *temp;
        temp = malloc(sizeof(struct node));
        temp->data = num;
        if(top[address] == NULL || num < top[address]->data)
        {       temp->link = top[address];
                top[address] = temp;
                return;
        }
        else
        {       q = top[address];
                while(q->link != NULL && q->link->data < num)
                {       q = q->link;    }
                temp->link = q->link;
                q->link = temp;
        }
}
int hash_function(int number)
{       int address;    float temp;
        temp = (float)number / maximum;
        address = temp * 6;
        return(address);
}

int main()
{       int i;
        printf("\nEnter Total Number of arr To Sort:\t");
        scanf("%d", &n);
        for(i = 0; i < n; i++)
        {       printf("Enter Element No. %d:\t", i + 1);
                scanf("%d", &arr[i]);
        }
        for(i = 0; i < n; i++)
        {       if(arr[i] > maximum)
                {
                        maximum = arr[i];
                }
        }
        address_sorting();
        printf("\nSorted List\n");
        for(i = 0; i < n; i++)
        {       printf("%3d  ", arr[i]);
        }
        printf("\n");
        return 0;
}
```

Output screen:



3. HASHING

3.1. Static Hashing

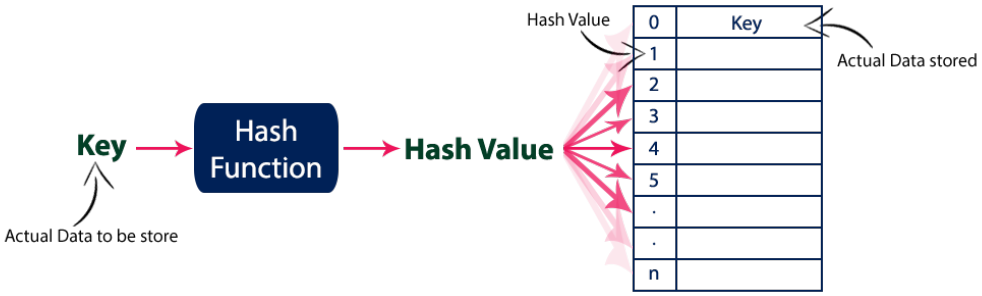
In all search techniques like linear search, binary search and search trees, the time required to search an element is depends on the total number of element in that data structure. In all these search techniques, as the number of element are increased the time required to search an element also increased linearly. Hashing is another approach in which time required to search an element doesn't depend on the number of element. Using hashing data structure, an element is searched with **constant time complexity**. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure. Hashing is defined as follows.

Hashing is the process of indexing and retrieving element (data) in a data structure to provide faster way of finding the element using the hash key.

Here, hash key is a value which provides the index value where the actual data is likely to store in the data structure.

In this data structure, we use a concept called **Hash table** to store data. All the data values are inserted into the hash table based on the hash key value. Hash key value is used to map the data with index in the hash table. And the hash key is generated for every data using a **hash function**. That means every entry in the hash table is based on the key value generated using a hash function.

Hash Table is defined as follows...



Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations can be performed with constant time complexity (i.e. O(1)).

Hash tables are used to perform the operations like insertion, deletion and search very quickly in a data structure. Using hash table concept insertion, deletion and search operations are accomplished in constant time. Generally, every hash table make use of a function, which we'll call the **hash function** to map the data into the hash table.

A hash function is defined as follows...

Hash function is a function which takes a piece of data (i.e. key) as input and outputs an integer (i.e. hash value) which maps the data to a particular index in the hash table.

HASH TABLEs

Hash table is a data structure in which keys are mapped to array positions by a hash function. In the example discussed here we will use a hash function that extracts the last two digits of the key. Therefore, we map the keys to array locations or array indices. A value stored in a hash table can be searched in O(1) time by using a hash function which generates an address from the key (by producing the index of the array where the value is stored).

HASH Functions

A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table. The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed. It produces a unique set of integers within some suitable range in order to reduce the number of collisions. In practice, there is no hash function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

3.2. Different Hash Functions

3.2.1. Division method

It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

h(x) = x mod M. The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M. For example, suppose M is an even number then h(x) is even if x is even and h(x) is odd if x is odd. If all possible keys are equi-probable, then this is not a problem. But if even keys are more likely than odd keys, then the division method will not spread the hashed values uniformly.

Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values. M should also be not too close to the exact powers of 2. If we have h(x) = x mod 2k then the function will simply extract the lowest k bits of the binary representation of x.
The division method is extremely simple to implement. The following code segment illustrates how to do this:

```
int const M = 97; // a prime number
int h (int x)
{ return (x % M); }
```

A potential drawback of the division method is that while using this method, consecutive keys

map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

Example 1 Calculate the hash values of keys 1234 and 5462.

Solution Setting M = 97, hash values can be calculated as:

h(1234) = 1234 % 97 = 70

h(5642) = 5642 % 97 = 16

3.2.2. Multiplication Method

The steps involved in the multiplication method are as follows:

Step 1: Choose a constant A such that 0 < A < 1.

Step 2: Multiply the key k by A.

Step 3: Extract the fractional part of kA.

Step 4: Multiply the result of Step 3 by the size of hash table (m).

Hence, the hash function can be given as:

$$h(k) = \lfloor m (kA \bmod 1) \rfloor$$

where (kA mod 1) gives the fractional part of kA and m is the total number of indices in the hash table. The greatest advantage of this method is that it works practically with any value of A. Although the algorithm works better with some values, the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is (sqrt5 – 1) /2 = 0.6180339887

Example 2 Given a hash table of size 1000, map the key 12345 to an appropriate location in the hash table.

Solution We will use A = 0.618033, m = 1000, and k = 12345

h(12345) = ⌊1000 (12345 * 0.618033 mod 1) ⌋

= ⌊1000 (7629.617385 mod 1 ⌋

= ⌊1000 (0.617385) ⌋

= ⌊617.385) ⌋

= 617

3.2.3. Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits

of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value. In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as: h(k) = s, where s is obtained by selecting r digits from k2.

Example 3 Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.

Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99. This means that only two digits are needed to map the key to a location in the hash table, so r = 2.

When k = 1234, k2 = 1522756, h (1234) = 27

When k = 5642, k2 = 31832164, h (5642) = 21

Observe that the 3rd and 4th digits starting from the right are chosen.

3.2.4. Folding Method

The folding method works in the following two steps:
Step 1: Divide the key value into a number of parts. That is, divide k into parts k_1,k_2, ..., k_n where each part has the same number of digits except the last part which may have lesser digits than the other parts.
Step 2: Add the individual parts. That is, obtain the sum of $k_1 + k_2 + . + k_n$. The hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table. For example, if the hash table has a size of 1000, then there are 1000 locations in the hash table. To address these 1000 locations, we need at least three digits; therefore, each part of the key must have three digits except the last part which may have lesser digits.

Example 4. Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.
Solution
Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678,	321,	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	

3.2.5. Digit Analysis

This method is particularly useful in the case of static file where all the keys in the table are known in advance. Each key is interpreted as a number using some radix r. The same radix is

used for all the keys in the table. Using this radix, the digits of each keys are examined. Digits having the most skewed distributions are deleted. Enough digits are deleted so that the number of remaining digits is small enough to give an address in the range of the hash table.

3.2.6. Converting Keys to Integers

To use some of the described hash functions, keys need to first be converted to nonnegative integers. Since all hash functions have several keys into the same home bucket, it is not necessary to convert keys into unique nonnegative integers.

Algm for converting Strings to Integers:

```
unsigned int stringToInt(char *key)
{
        int number=0;
        while(*key)
                number+=*key++;
        return number;
}
```

Alternative way to convert a string to non-negative integer

```
unsigned int stringToInt(char *key)
{
        int number=0;
        while(*key)
        {       number+=*key++;
                if(*key) number+=((int) *key++)<<8;
        }
return number;
}
```

2.1. Collisions

As discussed earlier in this chapter, collisions occur when the hash function maps two different keys to the same location. Obviously, two records cannot be stored in the same location. Therefore, a method used to solve the problem of collision, also called *collision resolution technique*, is applied. The two most popular methods of resolving collisions are:

1. Open addressing
2. Chaining

2.1.1. 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 even a single 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.

*2.1.2.*Linear Probing

The simplest approach to resolve a collision is 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] mod 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.

Note Linear probing is known for its simplicity. When we have to store a value, we try the slots: [h′(k) mod m, [h′(k) + 1]mod m, [h′(k) + 2]mod m, [h′(k) + 3]mod m, [h′(k) + 4]mod m, [h′(k) + 5]mod m, and so no, until a vacant location is found.

Example 15.5 Consider a hash table of size 10. Using linear probing, insert the keys 72, 36, 24, 63, 81, 92, and 101 into the table.

Let h′(k) = k mod m, m = 10

Initially, the hash table can be given as:

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) mod 10
= (2) mod 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
0	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.

Key = 92
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.

Key = 92
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.

Key = 92
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 = 101
h(101, 0) = (101 mod 10 + 0) 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) mod 10
          = (1 + 1) mod 10
          = 2
```

T[2] is also occupied, so we cannot store the key in this location. The procedure will be repeated until the hash function generates the address of location 8 which is vacant and can be used to store the value in it.

2.1.3. 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. The search time in this case is given as O(1). 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.

In the worst case, the search operation may have to make n–1 comparisons, and the running time of the search algorithm may take O(n) time. The worst case will be encountered when after scanning all the n–1 elements, the value is either present at the last location or not present in the table. Thus, we see that with the increase in the number of collisions, the distance between the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Pros and Cons

Linear probing finds an empty location by doing a linear search in the array beginning from position h(k). Although the algorithm provides good memory caching through good locality of reference, the drawback of this algorithm is that it results in clustering, and thus there is a higher risk of more collisions where one collision has already taken place. The performance of linear probing is sensitive to the distribution of input values. As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. In addition to this, when a new value has to be inserted into the table at a position which is already occupied, that value is inserted at the end of the cluster, which again increases the length of the cluster. Generally, an insertion is made between two clusters that

are separated by one vacant location. But with linear probing, there are more chances that subsequent insertions will also end up in one of the clusters, thereby potentially increasing the cluster length by an amount much greater than one. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called primary clustering. To avoid primary clustering, other techniques such as quadratic probing and double hashing are used.

2.1.4. 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 mod 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. Although quadratic probing performs better than linear probing, in order to maximize the utilization of the hash table, the values of c_1, c_2, and m need to be constrained.

Pros and Cons

Quadratic probing resolves the primary clustering problem that exists in the linear probing technique. Quadratic probing provides good memory caching because it preserves some locality of reference. But linear probing does this task better and gives a better cache performance. 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. In Example 15.6 try to insert the key 92 and you will encounter this problem.

Although quadratic probing is free from primary clustering, it is still liable to what is known as *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. This situation is usually encountered when the hash table is more than full. Quadratic probing is widely applied in the Berkeley Fast File System to allocate free blocks.

2.1.5. Double Hashing

To start with, double hashing 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:

$h(k, i) = [h1(k) + ih2(k)] \bmod m$

where m is the size of the hash table, h1(k) and h2(k) are two hash functions given as h1(k) = k mod m, h2(k) = k mod m', i is the probe number that varies from 0 to m−1, and m' is chosen to be less than m. We can choose m' = m−1 or m−2.

When we have to insert a key k in the hash table, we first probe the location given by applying [h1(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 [h2(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.

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.

2.2. Dynamic Hashing

2.3. Review on Hashing

- ✓ Hash table is a **data structure** in which keys are mapped to array positions by a hash function. A value stored in a hash table can be searched in **O(1)** time using a hash function which generates an address from the key.
- ✓ The storage requirement for a hash table is ***O(k),***where k is the number of keys actually used. In a hash table, an element with key k is stored at index ***h(k)***, not k. This means that a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or indices) in a hash table is called hashing.
- ✓ Popular hash functions which use numeric keys are division method, multiplication method, mid square method, and folding method.
- ✓ Division method divides x by M and then uses the remainder obtained. A potential drawback of this method is that consecutive keys map to consecutive hash values.
- ✓ Multiplication method applies the hash function given as ⌊ h (x) = I m (kA mod 1)⌋
- ✓ Mid square method works in two steps. First, it finds k2 and then extracts the middle r digits of the result.
- ✓ Folding method works by first dividing the key value k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts, and then obtaining the sum of $k_1 + k_2 + ... + k_n$. The hash value is produced by ignoring the last carry, if any.
- ✓ Collisions occur when a hash function maps two different keys to the same location.

Therefore, a method used to solve the problem of collisions, also **called *collision resolution technique*,** is applied. The two most popular methods of resolving collisions are: (a) open addressing and (b) chaining.

- ✓ Once a collision takes place, open addressing computes new positions using a probe sequence and the next record is stored in that position. In this technique of collision resolution, all the values are stored in the hash table. The hash table will contain two types of values—either sentinel value (for example, –1) or a data value.
- ✓ Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.
- ✓ In linear probing, 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] mod m Though linear probing enables good memory caching, the drawback of this algorithm is that it results in primary clustering.
- ✓ In quadratic probing, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision: $h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$ Quadratic probing eliminates primary clustering and provides good memory caching. But it is still liable to secondary clustering.
- ✓ 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)]** mod m The performance of double hashing is very close to the performance of the ideal scheme of uniform hashing. It minimizes repeated collisions and the effects of clustering.
- ✓ When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. So in rehashing, all the entries in the original hash table are moved to the new hash table which is double the size of the original hash table.
- ✓ 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. While the cost of inserting a key in a chained hash table is O(1), the cost for deleting and searching a value is given as O(m), where m is the number of elements in the list of that location. However, in the worst case, searching for a value may take a running time of O(n).

2.4. Review Questions

1. Define a hash table.
2. What do you understand by a hash function? Give the properties of a good hash function.
3. How is a hash table better than a direct access table (array)?
4. Write a short note on the different hash functions. Give suitable examples.
5. Calculate hash values of keys: 1892, 1921, 2007, and 3456 using different methods of hashing.
6. What is collision? Explain the various techniques to resolve a collision. Which technique do you think is better and why?
7. Consider a hash table with size = 10. Using linear probing, insert the keys 27, 72, 63, 42, 36, 18, 29, and 101 into the table.
8. Consider a hash table with size = 10. Using quadratic probing, insert the keys 27, 72, 63, 42, 36, 18, 29, and 101 into the table. Take c1 = 1 and c2 = 3.

9. Consider a hash table with size = 11. Using double hashing, insert the keys 27, 72, 63, 42, 36, 18, 29, and 101 into the table. Take h1 = k mod 10 and h2 = k mod 8.
10. What is hashing? Give its applications. Also, discuss the pros and cons of hashing.
11. Explain chaining with examples.
12. Write short notes on: Linear probing, Quadratic probing, Double hashing,

Multiple-choice Questions

1. In a hash table, an element with key k is stored at index
(a) k (b) log k (c) h(k) (d) k2
2. In any hash function, M should be a
(a) Prime number (b) Composite number (c) Even number (d) Odd number
3. In which of the following hash functions, do consecutive keys map to consecutive hash values?
(a) Division method (b) Multiplication method (c) Folding method (d) Mid-square method
4. The process of examining memory locations in a hash table is called (a) Hashing (b) Collision (c) Probing (d) Addressing
5. Which of the following methods is applied in the Berkeley Fast File System to allocate free blocks?
(a) Linear probing (b) Quadratic probing (c) Double hashing (d) Rehashing
6. Which open addressing technique is free from clustering problems? (a) Linear probing (b) Quadratic probing (c) Double hashing (d) Rehashing

True or False

1. Hash table is based on the property of locality of reference.
2. Binary search takes **O(n** log n) time to execute.
3. The storage requirement for a hash table is **O(k2)**, where k is the number of keys.
4. Hashing takes place when two or more keys map to the same memory location.
5. A good hash function completely eliminates collision.
6. ***M*** should not be too close to exact powers of 2.
7. A sentinel value indicates that the location contains valid data.
8. Linear probing is sensitive to the distribution of input values.
9. A chained hash table is faster than a simple hash table.

Fill in the Blanks

1. In a hash table, keys are mapped to array positions by a _____.
2. _____ is the process of mapping keys to appropriate locations in a hash table.
3. In open addressing, hash table stores either of two values _____ and _____.
4. When there is no free location in the hash table then _____ occurs.
5. More the number of collisions, higher is the number of _____ to find free location _____ which eliminates primary clustering but not secondary clustering.
6. _____ eliminates primary clustering but not secondary clustering.

Answers

Multiple-choice Questions
1. (c) 2. (a) 3. (a) 4. (c) 5. (b) 6. (c)
True or False
1. False 2. False 3. False 4. False 5. False 6. True 7. False 8. True 9. True

Fill in the Blanks

1. Hash function 2. Hashing 3. Sentinel value and a data value. 4. Collision 5. Probes 6. Quadratic probing

3. FILES

3.1. What is a File?

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

A file is defined as a collection of related data stored in auxiliary/secondary storage devices such as disks, CD, flash memories etc. The data is stored in the memory is recorded using 0's & 1's. When the data stored in auxiliary devices is read by the program, the data can be interpreted as either a text file or a binary file.

3.2. Data Hierarchy

Every file contains data which can be organized in a hierarchy to present a systematic organization. The data hierarchy includes data items such as fields, records, files, and database. These terms are defined below.

- A *data field* is an elementary unit that stores a single fact. A data field is usually characterized by its type and size. For example, student's name is a data field that stores the name of students. This field is of type *character* and its size can be set to a maximum of 20 or 30 characters depending on the requirement.
- A *record* is a collection of related data fields which is seen as a single unit from the application point of view. For example, the student's record may contain data fields such as name, address, phone number, roll number, marks obtained, and so on.
- A *file* is a collection of related records. For example, if there are 60 students in a class, then there are 60 records. All these related records are stored in a file. Similarly, we can have a file of all the employees working in an organization, a file of all the customers of a company, a file of all the suppliers, so on and so forth. A *directory* stores information of related files. A directory organizes information so that users can find it easily. Example shows how multiple related files are stored in a student directoy.

  Student's
  Personal Info File
  Roll_no
  Name
  Address
  Phone No
  Student's

3.3. FILE ATTRIBUTES

Every file in a computer system is stored in a directory. Each file has a list of attributes

associated with it that gives the operating system and the application software information about the file and how it is intended to be used.

A software program which needs to access a file looks up the directory entry to discern the attributes of that file. For example, if a user attempts to write to a file that has been marked as a read-only file, then the program prints an appropriate message to notify the user that he is trying to write to a file that is meant only for reading. Similarly, there is an attribute called *hidden*. When you execute the DIR command in DOS, then the files whose hidden attribute is set will not be displayed. These attributes are explained in this section.

File name It is a string of characters that stores the name of a file. File naming conventions vary from one operating system to the other.

File position It is a pointer that points to the position at which the next read/write operation will be performed.

File structure It indicates whether the file is a text file or a binary file. In the text file, the numbers (integer or floating point) are stored as a string of characters. A binary file, on the other hand, stores numbers in the same way as they are represented in the main memory.

File Access Method

It indicates whether the records in a file can be accessed sequentially or randomly. In sequential access mode, records are read one by one. That is, if 60 records of students are stored in the STUDENT file, then to read the record of 39th student, you have to go through the record of the first 38 students. However, in random access, records can be accessed in any order.

Attributes Flag

A file can have six additional attributes attached to it. These attributes are usually stored in a single byte, with each bit representing a specific attribute. If a particular bit is set to '1' then this means that the corresponding attribute is turned on. Table 16.1 shows the list of attributes and their position in the attribute flag or attribute byte. If a system file is set as hidden and read -only, then its attribute byte can be given as 00000111. We will discuss all these attributes here in this section. Note that the directory is treated as a special file in the operating system. So, all these attributes are applicable to files as well as to directories.

Read-only A file marked as read-only cannot be deleted or modified. For example, if an attempt is made to either delete or modify a read-only file, then a message 'access denied' is displayed on the screen.

Hidden A file marked as hidden is not displayed in the directory listing. ***System*** A file marked as a system file indicates that it is an important file used by the system and should not be altered or removed from the disk. In essence, it is like a 'more serious' read-only flag.

Volume Label Every disk volume is assigned a label for identification. The label can be assigned at the time of formatting the disk or later through various tools such as the DOS command LABEL.

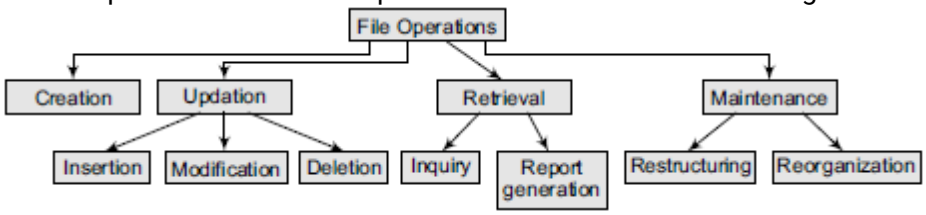
Directory In directory listing, the files and sub-directories of the current directory are differentiated by a directory-bit. This means that the files that have the directory-bit turned on are actually sub-directories containing one or more files.

Archive The archive bit is used as a communication link between programs that modify files

and those that are used for backing up files. Most backup programs allow the user to do an incremental backup. Incremental backup selects only those files for backup which have been modified since the last backup. When the backup program takes the backup of a file, or in other words, when the program archives the file, it clears the archive bit (sets it to zero). Subsequently, if any program modifies the file, it turns on the archive bit (sets it to 1). Thus, whenever the backup program is run, it checks the archive bit to know whether the file has been modified since its last run. The backup program will archive only those files which were modified.

3.4. BASIC FILE OPERATIONS

The basic operations that can be performed on a file are shown in figure



Creating File

A file is created by specifying its name and mode. Then the file is opened for writing records that are read from an input device. Once all the records have been written into the file, the file is closed. The file is now available for future read/write operations by any program that has been designed to use it in some way or the other.

Updating a File

Updating a file means changing the contents of the file to reflect a current picture of reality. A file can be updated in the following ways:

- Inserting a new record in the file. For example, if a new student joins the course, we need to add his record to the STUDENT file.
- Deleting an existing record. For example, if a student quits a course in the middle of the session, his record has to be deleted from the STUDENT file. \
- Modifying an existing record. For example, if the name of a student was spelt incorrectly, then correcting the name will be a modification of the existing record.

Retrieving from a File

It means extracting useful data from a given file. Information can be retrieved from a file either for an inquiry or for report generation. An inquiry for some data retrieves low volume of data, while report generation may retrieve a large volume of data from the file.

Maintaining a File

It involves restructuring or re-organizing the file to improve the performance of the programs that access this file. Restructuring a file keeps the file organization unchanged and changes only the structural aspects of the file (for example, changing the field width or adding/deleting fields). On the other hand, file reorganization may involve changing the entire organization of the file.

3.5. FILE ORGANIZATION

We know that a file is a collection of related records. The main issue in file management is the way in which the records are organized inside the file because it has a significant effect on the system performance. Organization of records means the *logical* arrangement of records in the file and not the physical layout of the file as stored on a storage media. Since choosing an appropriate file organization is a design decision, it must be done keeping the priority of achieving good performance with respect to the most likely usage of the file. Therefore, the following considerations should be kept in mind before selecting an appropriate file organization method:

- Rapid access to one or more records
- Ease of inserting/updating/deleting one or more records without disrupting the speed of accessing record(s)
- Efficient storage of records
- Using redundancy to ensure data integrity

Although one may find that these requirements are in contradiction with each other, it is the designer's job to find a good compromise among them to get an adequate solution for the problem at hand. For example, the ease of addition of records can be compromised to get fast access to data.

3.6. Sequential Organization

A sequentially organized file stores the records in the order in which they were entered. That is, the first record that was entered is written as the first record in the file, the second record entered is written as the second record in the file, and so on. As a result, new records are added only at the end of the file. Sequential files can be read only sequentially, starting with the first record in the file. Sequential file organization is the most basic way to organize a large collection of records in a file. Figure shows n records numbered from 0 to n–1 stored in a sequential file.

Once we store the records in a file, we cannot make any changes to the records. We cannot even delete the records from a sequential file. In case we need to delete or update one or more records, we have to replace the records by creating a new file.

In sequential file organization, all the records have the same size and the same field format, and every field has a fixed size. The records are sorted based on the value of one field or a combination of two or more fields. This field is known as the *key*. Each key uniquely identifies a record in a file. Thus, every record has a different value for the key field. Records can be sorted in either ascending or descending order. Sequential files are generally used to generate reports or to perform sequential reading of large amount of data which some programs need to do such as payroll processing of all the employees of an organization. Sequential files can be easily stored on both disks and tapes.

.

3.7. Relative File Organization

Relative file organization provides an effective way to access individual records directly. In a relative file organization, records are ordered by their *relative key.* It means the record number

represents the location of the record relative to the beginning of the file. The record numbers range from 0 to n–1, where n is the number of records in the file. For example, the record with record number 0 is the first record in the file. The records in a relative file are of fixed length.

Therefore, in relative files, records are organized in ascending *relative record number.* A relative file can be thought of as a single dimension table stored on a disk, in which the relative record number is the index into the table.

Relative files can be used for both random as well as sequential access. For sequential access, records are simply read one after another. Relative files provide support for only one key, that is, the relative record number. This key must be numeric and must take a value between 0 and the current highest relative record number –1. This means that enough space must be allocated for the file to contain the records with relative record numbers between 0 and the highest record number –1. For example, if the highest relative record number is 1,000, then space must be allocated to store 1,000 records in the file. Figure shows a schematic representation of a relative file which has been allocated enough space to store 100 records. Although it has space to accommodate 100 records, not all the locations are occupied. The locations marked as FREE are yet to store records in them. Therefore, every location in the table either stores a record or is marked as FREE. Relative file organization provides random access by directly jumping to the record which has to be accessed. If the records are of fixed length and we know the base address of the file and
the length of the record, then any record i can be accessed using the following formula:
Address of ith record = base_address + (i–1) * record_length
Note that the base address of the file refers to the starting address of the file. We took i–1 in the formula because record numbers start from 0 rather than 1.

Consider the base address of a file is 1000 and each record occupies 20 bytes, then the address of the 5th record can be given
as:
1000 + (5–1) * 20
= 1000 + 80
= 1080

3.8. Indexed Sequential File Organization

Indexed sequential file organization stores data for fast retrieval. The records in an indexed sequential file are of fixed length and every record is uniquely identified by a key field. We maintain a table known as the *index table* which stores the record number and the address of all the records. That is for every file, we have an index table. This type of file organization is called as indexed sequential file organization because physically the records may be stored anywhere, but the index table stores the address of those records.

The i^{th} entry in the index table points to the i^{th} record of the file. Initially, when the file is created, each entry in the index table contains NULL. When the i^{th} record of the file is written, free space is obtained from the free space manager and its address is stored in the i^{th} location of the index table. Now, if one has to read the 4th record, then there is no need to access the first three records. Address of the 4^{th} record can be obtained from the index table and the record can be straightaway read from the specified address (742, in our example). Conceptually, the index sequential file organization can be visualized as shown in figure. An indexed sequential file uses the concept of both sequential as well as relative files. While the index table is read

sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly. Indexed sequential files perform well in situations where sequential access as well as random access is made to the data. Indexed sequential files can be stored only on devices that support random access, for example, magnetic disks.

For example, take an example of a college where the details of students are stored in an indexed sequential file. This file can be accessed in two ways:

- *Sequentially*—to print the aggregate marks obtained by each student in a particular course or
- *Randomly*—to modify the name of a particular student.

3.9. Indexing

An index for a file can be compared with a catalogue in a library. Like a library has card catalogues based on authors, subjects, or titles, a file can also have one or more indices. Indexed sequential files are very efficient to use, but in real-world applications, these files are very large and a single file may contain millions of records. Therefore, in such situations, we require a more sophisticated indexing technique. There are several indexing techniques and each technique works well for a particular application. For a particular situation at hand, we analyze the indexing technique based on factors such as access type, access time, insertion time, deletion time, and space overhead involved. There are two kinds of indices:

- *Ordered indices* that are sorted based on one or more key values
- *Hash indices* that are based on the values generated by applying a *hash function*

3.10. Summary

- ✓ A file is a block of useful information which is available to a computer program and is usually stored on a persistent storage medium.
- ✓ Every file contains data. This data can be organized in a hierarchy to present a systematic organization. The data hierarchy includes data items such as fields, records, files, and database.
- ✓ A data field is an elementary unit that stores a single fact. A record is a collection of related data fields which is seen as a single unit from the application point of view. A file is a collection of related records. A directory is a collection of related files.
- ✓ A database is defined as a collection of data organized in a fashion that facilitates updating, retrieving, and managing the data.
- ✓ There are two types of computer files—text files and binary files. A text file is structured as a sequence of lines of alphabet, numbers, special characters, etc. However, the data in a text file is stored using its corresponding ASCII code. Whereas in binary files, the data is stored in binary form, i.e., in the format it is stored in the memory.
- ✓ Each file has a list of attributes associated with it which can have one of two states—*on* or *off*. These attributes are: read-only, hidden, system, volume label, archive, and directory. A file marked as read-only cannot be deleted or modified.
- ✓ A hidden file is not displayed in the directory listing.
- ✓ A system file is used by the system and should not be altered or removed from the disk. The archive bit is useful for communication between programs that modify files and programs that are used for backing up files.

- ✓ A file that has the directory bit turned on is actually a sub-directory containing one or more files.
- ✓ File organization means the logical arrangement of records in the file. Files can be organized as sequential, relative, or index sequential.
- ✓ A sequentially organized file stores records in the order in which they were entered.
- ✓ In relative file organization, records in a file are ordered by their relative key. Relative files can be used for both random access as well as sequential access of data. In an indexed sequential file, every record is uniquely identified by a key field. We maintain a table known as the index table that stores record number and the address of the record in the file.
- ✓ There are several indexing techniques, and each technique works well for a particular application.
- ✓ In a dense index, index table stores the address of every record in the file. However, in a sparse index, index table stores address of only some of the records in the file.
- ✓ Cylinder surface indexing is a very simple technique which is used only for the primary key index of a sequentially ordered file.

3.11. QUESTIONS ON FILE OPERATIONS

Review Questions

1. Why do we need files?
2. Explain the terms field, record, file organization, key, and index.
3. Define file. Explain all the file attributes.
4. How is archive attribute useful?
5. Differentiate between a binary file and a text file.
6. Explain the basic file operations.
7. What do you understand by the term file organization? Briefly summarize the different file organizations that are widely used today.
8. Write a brief note on indexing.

Multiple-choice Questions

1. Which of the following flags is cleared when a file is backed up?
(a) Read-only (b) System (c) Hidden (d) Archive
2. Which is an important file used by the system and should not be altered or removed from the disk?
(a) Hidden file (b) Archived file (c) System file (d) Read-only file
3. The data hierarchy can be given as
(a) Fields, records, files and database (b) Records, files, fields and database
(c) Database, files, records and fields (d) Fields, records, database, and files
4. Which of the following indexing techniques is used in document retrieval systems for large databases?
(a) Inverted index (b) Multi-level indices (c) Hashed indices (d) B-tree index

True or False

1. When a backup program archives the file, it sets the archive bit to one.
2. In a text file, data is stored using ASCII codes.
3. A binary file is more efficient than a text file.

4. Maintenance of a file involves re-structuring or re-organization of the file.
5. Relative files can be used for both random access of data as well as sequential access.
6. In a sparse index, index table stores the address of every record in the file.
7. Higher level indexing must always be sparse.
8. B-tree indices are most suitable for highly selective columns.

Fill in the Blanks
1. _____ is a block of useful information.
2. A data field is usually characterized by its _____ and _____.
3. _____ is a collection of related data fields.
4. _____ is a pointer that points to the position at which next read/write operation will be performed.
5. _____ indicates whether the file is a text file or a binary file.
6. Index table stores _____ and _____ of the record in the file.
7. In a sequentially ordered file the index whose search key specifies the sequential order of the file is defined as the _____ index.
8. _____ files are frequently used indexing technique in document retrieval systems for large textual databases.
9. _____ is a collection of data organized in a fashion that facilitates updating, retrieving, and managing the data.

Answers : Multiple-choice Questions
1. (d) 2. (c) 3. (a) 4. (a)
True or False
1. False 2. True 3. True 4. True 5. True 6. False 7. True 8. True
Fill in the Blanks
1. File 2. Type and size 3. Record 4. File position 5. File structure 6. Record number and address 7. Primary 8. Inverted 9. Database