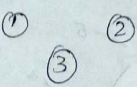
MODULE 5

MODULE : 5 GRAPHS, SORTING & SEARCHING, HASHING AND FILE ORGANIZATION

Graph Terminologies :-

① **Vertex** : A vertex is a synonym for a node. A vertex is represented by a circle.

Eg:



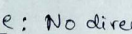
Nodes 1, 2, 3 are vertices

② **Edge** : An arc or a line joining two vertices say U and V is called an edge.

Eg:

a) **Undirected edge** : No direction b/w 2 vertices.

Eg:



* Denoted by an ordered pair (1,2).
* (1,2) is same as (2,1) in an undirected edge.

b) **Directed edge** : a direction exists b/w 2 vertices. Eg:

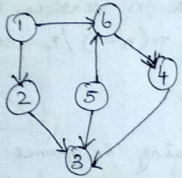
* Denoted by the directed pair <1,2> where 1 is called tail of the edge and 2 is the head of the edge. Hence <1,2> is not same as <2,1>.

③ **Graph** :- Formally, a graph G is defined as a pair of two sets V and E, denoted by

$G = (V, E)$

where V is the set of vertices and E is the set of edges.

Eg:



$V = \{1, 2, 3, 4, 5, 6\}$

$E = \{ \langle 1,6 \rangle, \langle 1,2 \rangle, \langle 2,3 \rangle, \langle 4,3 \rangle, \langle 5,3 \rangle, \langle 5,6 \rangle, \langle 6,4 \rangle \}$

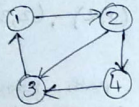
$|V| = 6$ → no. of vertices in graph

$|E| = 7$ → no. of edges in graph

(4) **Directed Graph:** A graph $G = (V, E)$ in which every edge is directed is called a directed graph.

* Also called as digraph.

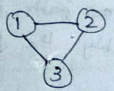
Example:



$V = \{1, 2, 3, 4\}$

$E = \{ \langle 1,2 \rangle, \langle 2,4 \rangle, \langle 3,1 \rangle, \langle 4,3 \rangle, \langle 2,3 \rangle \}$

(5) **Undirected Graph:** A graph $G = (V, E)$ in which every edge is undirected is called an Undirected graph.



$V = \{1, 2, 3\}$

$E = \{ (1,2), (2,1), (1,3), (3,1), (2,3), (3,2) \}$

(6) **Self loop (self edge):**

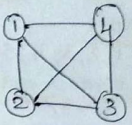
* A self loop is an edge which starts and ends on the same vertex.

(7) **Complete Graph:**

* A Graph $G = (V, E)$ is said to be a complete graph, if there exists an edge b/w every pair of

Vertices.

Eg:

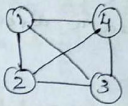


Note: For a complete graph with n vertices, there will be $n(n-1)/2$ edges.

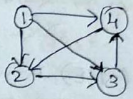
(8) Path:

* A path is denoted using sequence of vertices and there exists an edge from one vertex to next vertex.

Eg:

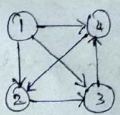


The path from vertex 1 to 4 is denoted by: 1, 2, 3, 4 which can also be written as; (1,2), (2,3), (3,4).



The path from vertex 1 to 3 is denoted by 1, 4, 2, 3 which can also be written as; <1,4>, <4,2>, <2,3>

(9) Simple path: A simple path is a path in which all vertices except possibly the first and last are distinct.



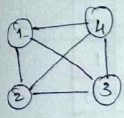
* In the graph, the path 1, 4, 2, 3 is a simple path since each node in the sequence is distinct. (appears only once).

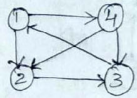
(10) Length of the path: no. of edges in the path gives the length of the path.

Eg: <1, 4, 2, 3> = 3

(11) **Connected graph:** In a graph G (directed or undirected) is said to be connected if and only if there exists a path b/w every pair of vertices.

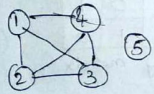
Eg:

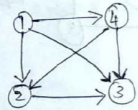




(12) **Disconnected Graph:** A Graph G is said to be disconnected, if there exists atleast one vertex in a graph that cannot be reached from other vertices

Eg:





Graph Representations:-

* There are two methods:
  a) Adjacency Matrix
  b) Adjacency Linked List

a) Adjacency Matrix:-

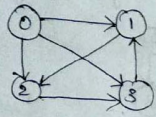
* Let G = (V, E) be a graph. Let N be the no. of vertices in Graph G. The Adjacency matrix A of a graph G is defined as:

$$A[i][j] = \begin{cases} 1 & \text{if there exists an edge from vertex } i \text{ to } j. \\ 0 & \text{if there is no edge from vertex } i \text{ to } j. \end{cases}$$

* It is a boolean square matrix with 'n' rows and 'n' columns, with entries 1's and 0's.

Example :

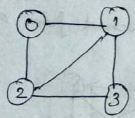
① Directed graph.



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

Adjacency matrix

② Undirected graph



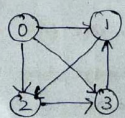
	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

Adjacency matrix

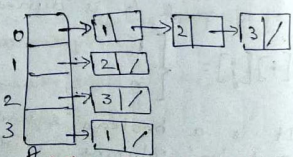
b.) Adjacency Linked List :-

* Let $G = (V, E)$ be a graph. An adjacency linked list is an array of 'n' linked lists where n is the no. of vertices in graph G.
* Each location of the array gives a vertex of the graph.
* For each vertex $U \in V$, a linked list consisting of all the vertices adjacent to U is created and stored in A[U].

Example:

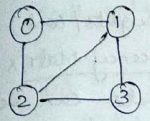


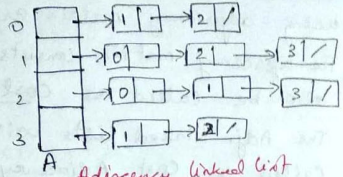
i) directed graph.



A Adjacency Linked list

ii) Undirected graph.





Adjacency linked list

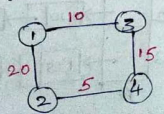
➡ Note; Which Graph Representation is best?

* Depends on the following factors:
  i) Nature of the problem
  ii) Algorithm used for solving.
  iii) Type of input
  iv) No. of vertices and edges.
    * If graph is sparse (less no. of edges) then adjacency list is best, since uses lesser space compared to adj. matrix.
    * If graph is dense (more no. of edges) then adjacency matrix is best.

Weighted Graphs:- (costs)

* A graph in which a number is assigned to each edge in a graph is called weighted graph.
* The weights may represent the cost involved or length or capacity depending on the problem.

Eg:



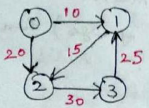
* **Weighted Graph representation**: can be represented using adjacency matrix or adjacency linked list. The adjacency matrix consists of weights (cost) can also be called as **Cost Adjacency Matrix**. The Adj. Linked lists with costs can be called as **Cost Adjacency Linked List**.

Defn: Let $G = (V, E)$ be a graph, with 'n' no. of vertices. The cost adj. matrix A of a graph is formally defined as;

$$A[i][j] = \begin{cases} w & \text{if there is a weight associated from vertex } i \text{ to } j, \\ \infty & \text{if no edge from } i \text{ to } j. \end{cases}$$

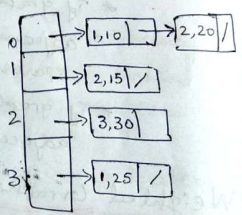
Example: Using Cost Adjacency matrix and Cost Adj. L.L

a) For directed graph.



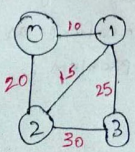
	0	1	2	3
0	0	10	20	∞
1	∞	0	15	∞
2	∞	∞	0	30
3	∞	25	∞	0

Cost Adj. Matrix



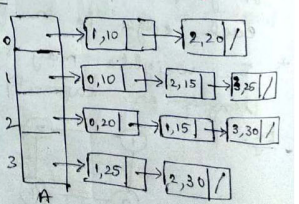
Cost Adj. List

b) For Undirected weighted graph



	0	1	2	3
0	0	10	20	∞
1	10	0	15	25
2	20	15	0	30
3	∞	25	30	0

Cost Adj. Matrix



Cost Adj. Linked list

Function to Read Adj. Matrix & Adj. Linked List:

```
void read_matrix(int a[][], int n)
{
    int i, j;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            scanf("%d", &a[i][j]);
}

Void read_List(NODE a[], int n)
{
    int i, j, m, item;
    for(i=0; i<n; i++)
    {
        Printf("Enter No. of nodes adjacent to %d", i);
        scanf("%d", &m);
        if(m==0) continue;
        Printf("Enter Nodes adj. to %d", i);
        for(j=0; j<m; j++)
        {
            scanf("%d", &item);
            a[i] = insert_rear(item, a[i]);
        }
    }
}
```


Graphs Traversals:- The process of visiting each node of a graph systematically in some order is called graph Traversal.

* Two Graph Traversal technique :-

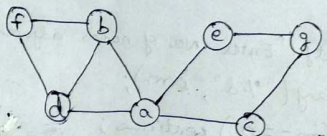
① Breadth First Search (BFS)

② Depth First Search (DFS)

① Breadth First Graph (BFS) :

* BFS is a method of traversing the graph from an arbitrary vertex say u. First, visit the node U. Then we visit all neighbors of U. Then we visit the neighbors of U and so on ie; we visit all the neighbouring nodes first before moving to next level neighbors.
* The search will terminate when all the vertices have been visited.
* BFS can be implemented using a queue.

Example :



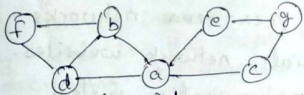
	u = del(Q)	v = adj. to U	Nodes Visited S	Queue (Q)
Initial	—	—	a	a
1	a	b, c, d, e	a, b, c, d, e	b, c, d, e
2	b	a, d, f	a, b, c, d, e, f	c, d, e, f
3	c	a, g	a, b, c, d, e, f, g	d, e, f, g
4	d	a, b, f	a, b, c, d, e, f, g	e, f, g
5	e	a, g	a, b, c, d, e, f, g	f, g
6	f	b, d	a, b, c, d, e, f, g	g
7	g	c, e	a, b, c, d, e, f, g	—

Thus, the nodes that are reachable from source a = a b c d e f g

(2) Depth First Search (DFS) :-

* DFS is a method of traversing the graph by visiting each node of the graph in a systematic order ie; to search deeper in the graph.
* In DFS, a vertex U is picked as a source and is visited. The vertex U at this point is unexplored. The exploration of vertex U is postponed and a vertex V adjacent to U is picked and is visited. Now the search begins at the vertex V.
* There may be still some nodes which are adjacent to U but not visited. When the vertex V is completely examined, then only U is examined.
* The search will terminate when all the vertices have been examined.
* DFS is implemented using stacks.

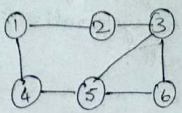
Example;



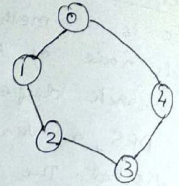
	Stack	v = adj.(S[top])	Nodes visited S	Pop(stack)
Initial	a	–	a	–
1	a	b	a,b	
2	a,b	d	a,b,d	
3	a,b,d	f	a,b,d,f	
4	a,b,d,f	–	a,b,d,f	f
5	a,b,d	–	a,b,d,f	d
6	a,b	–	a,b,d,f	b
7	a	c	a,b,d,f,c	
8	a,c	g	a,b,d,f,c,g	
9	a,c,g	e	a,b,d,f,c,g,e	
10	a,c,g,e	–	a,b,d,f,c,g,e	e
11	a,c,g	–	a,b,d,f,c,g,e	g
12	a,c	–	a,b,d,f,c,g,e	c
13	a	–	a,b,d,f,c,g,e	a

→ Solve using BFS and DFS

(i)



(ii)



* C Program to implement BFS and DFS graph traversal → Refer Lab. Program 11.

Applications Of BFS and DFS :-

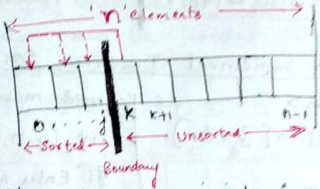
BFS:
1. To find shortest path from a vertex to another vertex in an unweighted graph.
2. To find if a graph contains cycles.
3. In peer-peer networks.
4. Social network websites.
5. GPS navigation system.

DFS:
1. Produces Minimum spanning tree and all shortest path tree.
2. To detect cycles
3. Topological Sorting.
4. To find connected graphs.

Sorting :-

1.) Insertion Sort :-

Procedure: The given list is divided into 2 parts: sorted part and unsorted part.



* All the elements from 0 to j are sorted and elements from k till n-1 are unsorted.
* The k^{th} item can be inserted into any of the positions from 0 to j, so that elements towards left of boundary are sorted.
* As each item is inserted towards left, the boundary moves to the right by one position, thus decreasing the unsorted list.
* Once the boundary reaches the last position the list is finally sorted completely.

Example: Sort the elements 25, 75, 40, 10, 20 using insertion sort.

Given:

0	1	2	3	4
25	75	40	10	20

Step 1:

25	75	40	10	20
0	1	2	3	4

k^{th} item = 75

Step 2:

25	75	40	10	20

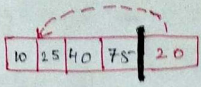
k^{th} item = 40

Step 3:

25	40	75	10	20

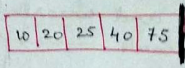
k^{th} item = 10

Step 4:



k^{th} item = 20

Step 5.



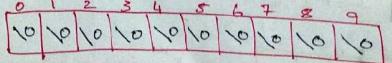
Final sorted list

C Program to implement Insertion Sort

```
#include<stdio.h>
void Insert(int a[], int n)
{
    int i, j, item;
    for(i=1; i<n; i++)
    {
        item = a[i];
        j = i-1;
        while(item<a[j] && j>=0)
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = item;
    }
}
void main()
{
    int i, j, n, item, a[20];
    printf("Enter No. of elements");
    scanf("%d", &n);
    printf("Enter the n elements");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    Insert(a, n);
    printf("The sorted Items:\n");
    for(i=0; i<n; i++)
        printf("%d\n", a[i]);
}
```

2.) Radix Sort :-

* Consider a list P: which contains 10 packets starting from 0 to 9.
* Initially all 10 packets are filled with a NULL.

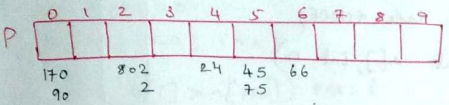


* Now let us sort the following elements:
  170, 45, 75, 90, 802, 24, 2, 66
* Firstly, find the largest number in the above given list and then check how many digits does it

contain. For example: The largest element is 802, and it has 3 digits. Thus, in our radix sort example we require totally 3 passes:

Pass 1: Scan each element in the list and obtain its LSD (Least Significant Digit ie; 1's place). Then each item can now be inserted into appropriate pocket as shown below: ie;

170, 45, 75, 90, 802, 24, 2, 66

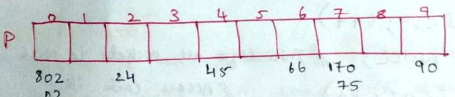


⇒ Now arrange the elements in order as shown below.

170, 90, 802, 2, 24, 45, 75, 66

Pass 2: Now scan each element and obtain the next LSD (least significant digit) ie; 2's place

170, 90, 802, 02, 24, 45, 75, 66

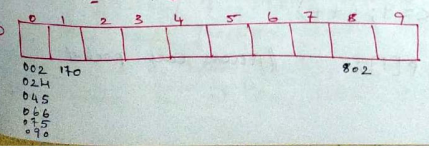


⇒ Now arrange the elements in order:

802, 02, 24, 45, 66, 170, 75, 90

Pass 3: Now scan each element & obtain the next LSD ie; 3's place:

802, 002, 024, 045, 066, 170, 075, 090



➡ Now arrange the elements in Order:

2, 24, 45, 66, 75, 90, 170, 802 ⟶ Sorted

C Function to sort numbers using radix sort

```
struct node
{
   int info;
   struct node *link;
};
typedef struct node *NODE;
Void radix (int a[], int n)
{
   int i,j,k,m, big, digit;
   NODE P[10], temp;
   big = Largest (a,n);        // find the largest number
   m = log 10 (big) +1;        // Find the No. of digits in big.
   for (j=1; j<=m; j++)
   {
     for(i=0; i<=9; i++)
         P[i] = NULL;          // Initialize all packets to NULL
     for(i=0; i<n; i++)        // Access each item
     {
       digit = a[i] / 10^(j-1) % 10;   // separate jth digit from a[i]
       P[digit]= insertrear(a[i], P[digit]);   Insert a[i] at end
     }
     k=0;
     for (i=0; i<=9; i++)
     {
         temp = P[i];          // Access each packet
```

```
while (temp != NULL)                    // copy items from each list
{
    s[k++] = temp->info;
    temp = temp->link;
  }
 }
}
void largest(int a[], int n)
{ int i, pos = 0;
  for(i=1; i<n; i++)
  {
     if( a[i] > a[pos]) pos = i;
  }
  return a[pos];
}
```

3.) Address calculation Sort :- // discussed later in Hashing

HASHING :-

* Hashing is an effective way to store and retrieve data in some data structure.
* Hashing technique is designed to use a special function called the hash function which is used to map a given value with a particular key for faster access of elements.
* The efficiency of mapping depends on the efficiency of the hash function used.

Example: Let a hash function H(x) maps the value the value x at the index x % 10, in an array.

For example if the list of values is {11, 12, 13, 14, 15} it will be stored at positions {1, 2, 3, 4, 5} in the array or hash table. ie;

$H(x) = x \% 10$; So:

$H(11) = 11 \% 10 = 1$

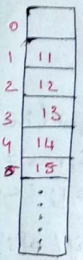
$H(12) = 12 \% 10 = 2$

$H(13) = 13 \% 10 = 3$

$H(14) = 14 \% 10 = 4$

$H(15) = 15 \% 10 = 5$

Hash Table ⇒



Hash Table

Note: Here 1, 2, 3, 4, 5 are called hash values and $H(x) = x \% 10$ is the Hash function used.

Hash Table Organization :-

- **Hash Table:** Hash table is a data structure used for storing and retrieving data very quickly.
  - * Insertion, deletion, or retrieval operations takes place with the help of hash value
  - * Hence every entry in the hash table is associated with some key. (Refer the above example).
  - * Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is dependant upon the size of the hash table.

Hash Function: is a function which is used to put the data in the hash table. The integer returned by the hash function is called hash key.

* A good hash function should satisfy 2 Criteria;

(i) A hash fn should generate the hash addresses such that all the keys are distributed as evenly as possible among the various cells of the hash table.

(ii) Computation of a key should be simple.

Types of hash Fns:

① Division method - The hash fn depends upon the remainder of division. Typically the divisor is the table length.

Ex: If the record to be stored 54, 72, 89, 37 is to be placed in the hash table and if the table size is 10.

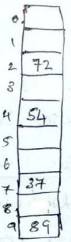
h(key) = record % SIZE ie;

h(54) = 54 % 10 = 4

h(72) = 72 % 10 = 2

h(89) = 89 % 10 = 9

h(37) = 37 % 10 = 7



② Mid-Square method: Here, the key k is squared. A number 'l' in the middle of k^2 is selected by removing the digits from both ends.

$h(k) = l$

Ex: key, k = 2345, its square is: $k^2 = 5745525$

$h(2345) = 45$ ⇐ By discarding 57 and 25.

③ Folding method: Here, key k is divided into s number of parts; k1, k2, k3 .... kn of same length except the last part. Then all parts are added together.

$h(k) = k1 + k2 + k3 + \dots\dots + kn$

Ex: K = 123987234876. The partitions are:
K1 = 12, K2 = 39, K3 = 87, K4 = 23, K5 = 48, K6 = 76

∴ h(123987234876) = 12 + 39 + 87 + 23 + 48 + 76
= 285

(4) By converting keys to integers: Here, if key K is a string. The hash value is obtained by converting the string into integer. ie; by adding all ASCII values of each character in the string.

Ex: K = "ABCD" Then

h("ABCD") = 'A' + 'B' + 'C' + 'D'
= 65 + 66 + 67 + 68
= 266

Types of Hashing Techniques:-

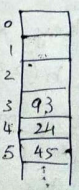
a) Static Hashing b) Dynamic Hashing

a) Static Hashing: is a technique in which the table (bucket) size remains the same (fixed during compilation time) is called static hashing.

* Various techniques of static hashing are linear probing, chaining.
* As the size is fixed, this type of hashing cannot handle overflow of elements (collision) effi. ely.

Eg: Elements to be stored: 24, 93, 45.

h(24) = 24%10 = 4
h(93) = 93%10 = 3
h(45) = 45%10 = 5



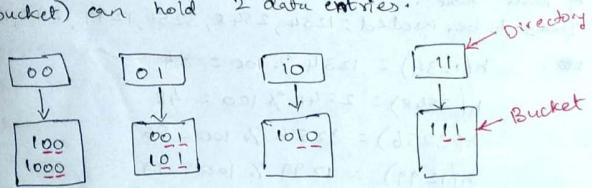
b) Dynamic Hashing:

* This is an hashing technique in which the bucket (table) size is not fixed. It can grow or shrink according to the increase or decrease of records.
* Typical example of dynamic hashing is Extensible hashing.

Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.

* In extensible hashing refering the size of directory the elements are to be placed in buckets.

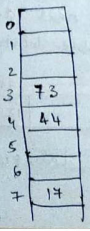
Example: To insert 1, 4, 5, 7, 8, 10. Assume each page (bucket) can hold 2 data entries.



Overflow Handling (Collision Resolution Techniques):-

* The phenomenon of two or more keys being hashed to the same location of the hash table (fixed size table) is called collision.

Ex: The data elements to be placed: 44, 73, 17, 77



If we try to place 77 in the hash table, we get the hash value 7 and at index 7, already 17 is placed. This situation is called as collision.

* The various technique using which collision can be avoided are:

① Open Addressing → Linear Probing

② Chaining.

1) Open Addressing: Here it involves static hashing - hash table size is fixed. In such a table, collision can be avoided by finding another, unoccupied location in the array. The collision can be avoided using Linear probing.

Example: Let size of hash table = 100. Let the hash function: $h(k) = k \% m$, where m is the size of hash table.

Items to be inseted: 1234, 2548, 3256, 1299, 1298, 1398

$h(1234) = 1234 \% 100 = 34$

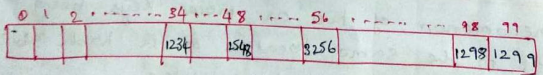
$h(2548) = 2548 \% 100 = 48$

$h(3256) = 3256 \% 100 = 56$

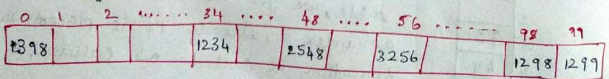
$h(1299) = 1299 \% 100 = 99$

$h(1298) = 1298 \% 100 = 98$

$h(1398) = 1398 \% 100 = 98$ // collision



* Collision is detected while inserting 1398 into 98th location. To overcome this, linear probing may be used. In Linear probing, it checks for the next available (empty) location. So, the next available location is 0.



→ C program on Linear probing – Refer Lab. Program 12.

2.) Chaining Method :-

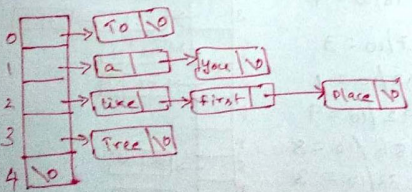
* Chaining technique avoids collision using an array of linked lists (run time).
* If more than one key has same hash value, then all the keys will be inserted at the end of the list (insert rear) one by one, and thus collision is avoided.

Example: Construct a hash table of size 5 and store the following words: like, a, tree, you, first, ~~first~~, a, place, to

⇒ Let $H(str) = P_0 + P_1 + \ldots + P_{n-1}$; where each P_i is position of letter in English alphabet series. Then calculate hash address = Sum % 5. So:

h(like) = 12 + 9 + 11 + 5 = 37
h(a) = 1
h(tree) = 20 + 18 + 5 + 5 = 48
h(you) = 25 + 15 + 21 = 61
h(first) = 6 + 9 + 18 + 19 + 20 = 72
h(a) = 1
h(place) = 16 + 12 + 1 + 3 + 5 = 37
h(to) = 20 + 15 = 35

37 % 5 = 2
1 % 5 = 1
48 % 5 = 3
61 % 5 = 1
72 % 5 = 2
1 % 5 = 1
37 % 5 = 2
35 % 5 = 0

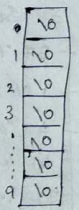


Explain Address calculation Sort with an example?

* Uses Hashing technique.
* The hash function should have the property that if $x1 \leq x2$, then hash(x1) $\leq$ hash(x2). The function which exhibits this property is called order preserving or Non-decreasing hashing function.

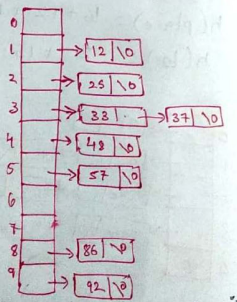
Example: 25, 57, 48, 37, 12, 92, 86, 33

* Let us create 10 subfiles. Initially each of the subfiles are empty.



* Here the largest Number is 92, so we divide all the elements using the hash function: H(k) = k/10.

ie: H(25) = 25/10 = 2
H(57) = 57/10 = 5
H(48) = 48/10 = 4
H(37) = 37/10 = 3
H(12) = 12/10 = 1
H(92) = 92/10 = 9
H(86) = 86/10 = 8
H(33) = 33/10 = 3



→ Here 3 which is repeated. It is inserted in 3rd subfile only, but must be checked with the existing elements for its proper position in this subfile.

MODULE 5 (contd.)

Files and Their Organization

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.

- *Data field:* A *data field* is an elementary unit that stores a single fact. A data field is usually characterized by its type and size.
  Example: student's name is a data field that stores the name of students.

- *Record:* A *record* is a collection of related data fields which is seen as a single unit from the application point of view.
  Example: The student's record may contain data fields such as name, address, phone number, roll number, marks obtained, and so on.

- *File:* A *file* is a collection of related records.
  Example: A file of all the employees working in an organization

- *Directory*: A *directory* stores information of related files. A directory organizes information so that users can find it easily.
  Example: Below fig. shows how multiple related files are stored in a student directoy.

Student's Personal Info File	Student's Academic Info File	Student's Fees Info File
Roll_no	Roll_no	Roll_no
Name	Name	Name
Address	Course	Fees
Phone No	Marks	Lab Dues
	Grade in Sports	Hostel Dues
		Library Dues

Figure Student directory

FILE ATTRIBUTES

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.

The attributes are explained below

- **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 are stored as a string of characters. A binary file 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.
  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 Attribute flag

Attribute	Attribute Byte
Read-Only	00000001
Hidden	00000010
System	00000100
Volume Label	00001000
Directory	00010000
Archive	00100000

Above figure shows the list of attributes and their position in the attribute flag or attribute byte.

- **Read-only:** A file marked as read-only cannot be deleted or modified. 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.
- **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.

TEXT AND BINARY FILES

Text Files

- A text file, also known as a flat file or an ASCII file, is structured as a sequence of lines of alphabet, numerals, special characters.
- The data in a text file, whether numeric or non-numeric, is stored using its corresponding ASCII code.
- The end of a text file is denoted by placing a special character, called an end-of-file marker, after the last line in the text file.
- It is possible for humans to read text files which contain only ASCII text.
- Text files can be manipulated by any text editor, they do not provide efficient storage.

Binary Files

- A binary file contains any type of data encoded in binary form for computer storage and processing purposes.
- A binary file can contain text that is not broken up into lines.
- A binary file stores data in a format that is similar to the format in which the data is stored in the main memory. Therefore, a binary file is not readable by humans.
- Binary files contain formatting information that only certain applications or processors can understand.
- Binary files must be run on an appropriate software or processor so that the software or processor can transform the data in order to make it readable.
- Binary files provide efficient storage of data, but they can be read only through an appropriate program.

BASIC FILE OPERATIONS

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

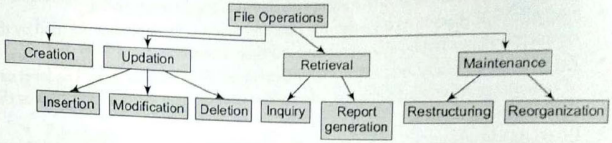


Figure File operations

Creating a 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.

Example: changing the field width or adding/deleting fields.

File reorganization may involve changing the entire organization of the file

FILE ORGANIZATION

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.

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
- Efficient storage of records
- Using redundancy to ensure data integrity

1. Sequential Organization

A sequentially organized file stores the records in the order in which they were entered. 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

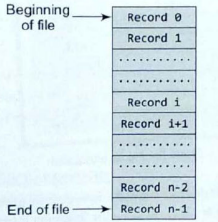


Figure Sequential file organization

Features

- Records are written in the order in which they are entered
- Records are read and written sequentially
- Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes
- Records have the same size and the same field format
- Records are sorted on a key value
- Generally used for report generation or sequential reading

Advantages

- Simple and easy to Handle
- No extra overheads involved
- Sequential files can be stored on magnetic disks as well as magnetic tapes
- Well suited for batch– oriented applications

Disadvantages

- Records can be read only sequentially. If i^{th} record has to be read, then all the i–1 records must be read
- Does not support update operation. A new file has to be created and the original file has to be replaced with the new file that contains the desired changes
- Cannot be used for interactive applications

2. Relative File Organization

Figure shows a schematic representation of a relative file which has been allocated space to store 100 records

Relative record number	Records stored in memory
0	Record 0
1	Record 1
2	FREE
3	FREE
4	Record 4
.....	.....
98	FREE
99	Record 99

Figure Relative file organization

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:

$$\text{Address of } i^{th} \text{ record} = \text{base\_address} + (i-1) * \text{record\_length}$$

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:

$$1000 + (5-1) * 20$$
$$= 1000 + 80$$
$$= 1080$$

Features

- Provides an effective way to access individual records
- The record number represents the location of the record relative to the beginning of the file
- Records in a relative file are of fixed length
- Relative files can be used for both random as well as sequential access
- Every location in the table either stores a record or is marked as FREE

Advantages

- Ease of processing
- If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously
- Random access of records makes access to relative files fast
- Allows deletions and updations in the same file
- Provides random as well as sequential access of records with low overhead
- New records can be easily added in the free locations based on the relative record number of the record to be inserted
- Well suited for interactive applications

Disadvantages

- Use of relative files is restricted to disk devices
- Records can be of fixed length only
- For random access of records, the relative record number must be known in advance

3. Indexed Sequential File Organization

The index sequential file organization can be visualized as shown in figure

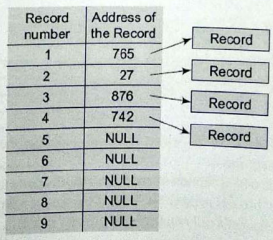


Figure Indexed sequential file organization

Features

- Provides fast data retrieval
- Records are of fixed length
- Index table stores the address of the records in the file
- The ith entry in the index table points to the ith record of the file
- 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

Advantages

- The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs
- Supports applications that require both batch and interactive processing
- Records can be accessed sequentially as well as randomly
- Updates the records in the same file

Disadvantages

- Indexed sequential files can be stored only on disks
- Needs extra space and overhead to store indices
- Handling these files is more complicated than handling sequential files
- Supports only fixed length records

INDEXING

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*

1. Ordered Indices

Indices are used to provide fast random access to records. An index of a file may be a primary index or a secondary index.

Primary Index

In a sequentially ordered file, the index whose search key specifies the sequential order of the file is defined as the primary index.

Example: suppose records of students are stored in a STUDENT file in a sequential order starting from roll number 1 to roll number 60. Now, if we want to search a record for, say, roll number 10, then the student's roll number is the primary index.

Secondary Index

An index whose search key specifies an order different from the sequential order of the file is called as the secondary index.

Example: If the record of a student is searched by his name, then the name is a secondary index. Secondary indices are used to improve the performance of queries on non-primary keys.

2. Dense and Sparse Indices

Dense index

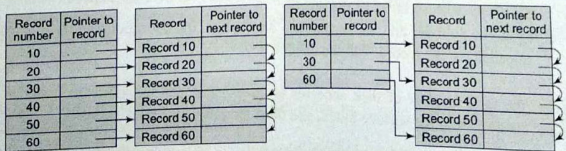
- In a dense index, the index table stores the address of every record in the file.
- Dense index would be more efficient to use than a sparse index if it fits in the memory
- By looking at the dense index, it can be concluded directly whether the record exists in the file or not.

Sparse index

- In a sparse index, the index table stores the address of only some of the records in the file.
- Sparse indices are easy to fit in the main memory,
- In a sparse index, to locate a record, first find an entry in the index table with the largest search key value that is either less than or equal to the search key value of the desired record. Then, start at that record pointed to by that entry in the index table and then proceed searching the record using the sequential pointers in the file, until the desired record is obtained.

Example: If we need to access record number 40, then record number 30 is the largest key value that is less than 40. So jump to the record pointed by record number 30 and move along the sequential pointer to reach record number 40.

Below figure shows a dense index and a sparse index for an indexed sequential file.



(a) Dense index (b) Sparse index

Figure Dense index and sparse index

3. Cylinder Surface Indexing

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file.
The index file will contain two fields—cylinder index and several surface indices.
There are multiple cylinders, and each cylinder has multiple surfaces. If the file needs m cylinders for storage then the cylinder index will contain m entries.

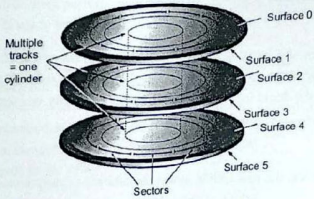


Figure Physical and logical organization of disk

When a record with a particular key value has to be searched, then the following steps are performed:

- First the cylinder index of the file is read into memory.
- Second, the cylinder index is searched to determine which cylinder holds the desired record. For this, either the binary search technique can be used or the cylinder index can be made to store an array of pointers to the starting of individual key values. In either case the search will take O (log m) time.
- After the cylinder index is searched, appropriate cylinder is determined.
- Depending on the cylinder, the surface index corresponding to the cylinder is then retrieved from the disk.
- Since the number of surfaces on a disk is very small, linear search can be used to determine surface index of the record.
- Once the cylinder and the surface are determined, the corresponding track is read and searched for the record with the desired key.

Hence, the total number of disk accesses is three—first, for accessing the cylinder index, second for accessing the surface index, and third for getting the track address.

4. Multi-level Indices

Consider very large files that may contain millions of records. For such files, a simple indexing technique will not suffice. In such a situation, we use multi-level indices. Below figure shows a two-level multi-indexing. Three-level indexing and so, can also be used

In the figure, the main index table stores pointers to three inner index tables. The inner index tables are sparse index tables that in turn store pointers to the records.

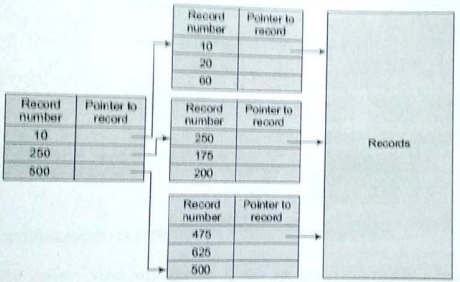


Figure Multi-level indices

5. Inverted Indices

- Inverted files are used in document retrieval systems for large textual databases.
- An inverted file reorganizes the structure of an existing data file in order to provide fast access to all records having one field falling within the set limits.
- When a term or keyword specified in the inverted file is identified, the record number is given and a set of records corresponding to the search criteria are created.
- For each keyword, an inverted file contains an inverted list that stores a list of pointers to all occurrences of that term in the main text. Therefore, given a keyword, the addresses of all the documents containing that keyword can easily be located.

There are two main variants of inverted indices:

- A record-level inverted index (inverted file index or inverted file) stores a list of references to documents for each word
- A word-level inverted index (full inverted index or inverted list) in addition to a list of references to documents for each word also contains the positions of each word within a document.

6. B-Tree (Balanced Tree) Indices

It is impractical to maintain the entire database in the memory, hence B-trees are used to index the data in order to provide fast access.
B-trees are used for its data retrieval speed, ease of maintenance, and simplicity.

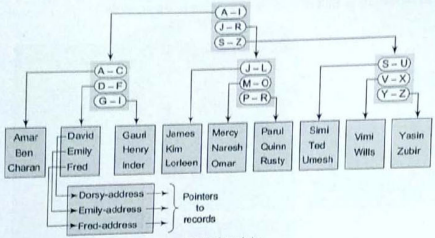


Figure B-tree index

- It forms a tree structure with the root at the top. The index consists of a B-tree (balanced tree) structure based on the values of the indexed column.
- In this example, the indexed column is name and the B-tree is created using all the existing names that are the values of the indexed column.
- The upper blocks of the tree contain index data pointing to the next lower block, thus forming a hierarchical structure. The lowest level blocks, also known as leaf blocks, contain pointers to the data rows stored in the table.

The B-tree structure has the following advantages:

- Since the leaf nodes of a B-tree are at the same depth, retrieval of any record from anywhere in the index takes approximately the same time.
- B-trees improve the performance of a wide range of queries that either searches a value having an exact match or for a value within specified range.
- B-trees provide fast and efficient algorithms to insert, update, and delete records that maintain the key order.
- B-trees perform well for small as well as large tables. Their performance does not degrade as the size of a table grows.
- B-trees optimize costly disk access.

7. Hashed Indices

Hashing is used to compute the address of a record by using a hash function on the search key value.
The hashed values map to the same address, then collision occurs and schemes to resolve these collisions are applied to generate a new address

Choosing a **good hash function** is critical to the success of this technique. By a good hash function, it mean two things.

1. First, a good hash function, irrespective of the number of search keys, gives an average-case lookup that is a small constant.
2. Second, the function distributes records uniformly and randomly among the buckets, where a bucket is defined as a unit of one or more records

The **worst hash function** is one that maps all the keys to the same bucket.

The drawback of using hashed indices includes:

- Though the number of buckets is fixed, the number of files may grow with time.
- If the number of buckets is too large, storage space is wasted.
- If the number of buckets is too small, there may be too many collisions.

The following operations are performed in a hashed file organization.

1. Insertion
To insert a record that has k_i as its search value, use the hash function $h(k_i)$ to compute the address of the bucket for that record.
If the bucket is free, store the record else use chaining to store the record.

2. Search
To search a record having the key value k_i, use $h(k_i)$ to compute the address of the bucket where the record is stored.
The bucket may contain one or several records, so check for every record in the bucket to retrieve the desired record with the given key value.

3. Deletion
To delete a record with key value k_i, use $h(k_i)$ to compute the address of the bucket where the record is stored. The bucket may contain one or several records so check for every record in the bucket, and then delete the record.