MODULE 5: GRAPHS

DEFINITIONS

A graph, G, consists of two sets V and E. V is a finite non-empty set of vertices. E is a set of pairs of vertices, these pairs are called edges. V(G) and E(G) will represent the sets of vertices and edges of graph G. We will also write G = (V, E) to represent a graph. In an undirected graph the pair of vertices representing any edge is unordered. Thus, the pairs (v1, v2) and (v2, v1) represent the same edge. In a directed graph each edge is represented by a directed pair (v1, v2). v1 is the tail and v2 the head of the edge. Therefore <v2, v1> and <v1, v2> represent two different edges.
Figure 6.2 shows three graphs G1, G2 and G3

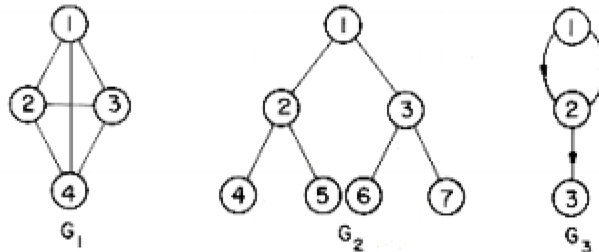


Figure 6.2 Three sample graphs.

The graphs G1 and G2 are undirected. G3 is a directed graph.

V (G1) = {1,2,3,4}; E (G1) = {(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)}

V (G2) = {1,2,3,4,5,6,7}; E (G2) = {(1,2), (1,3), (2,4), (2,5), (3,6), (3,7)}

V (G3) = {1,2,3}; E (G3) = {<1,2>, <2,1>, <2,3>}.

TERMINOLOGIES

If (v1, v2) is an edge in E(G), then we shall say the vertices v1 and v2 are **adjacent** and that the edge (v1, v2) is **incident** on vertices v1 and v2. The vertices adjacent to vertex 2 in G2 are 4, 5 and 1. The edges incident on vertex 3 in G2 are (1,3), (3,6) and (3,7). If <v1, v2> is a directed edge, then vertex v1 will be said to be **adjacent** to v2 while v2 is **adjacent from** v1.
The edge <v1, v2> is incident to v1 and v2. In G3 the edges **incident** to vertex 2 are <1,2>, <2,1> and <2,3>.

Multigraph: **Multi Graph**: Any **graph** which contain some parallel edges but doesn't contain any self-loop is called **multi graph**.

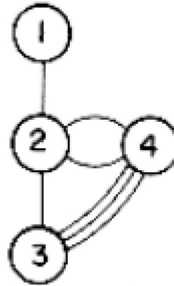
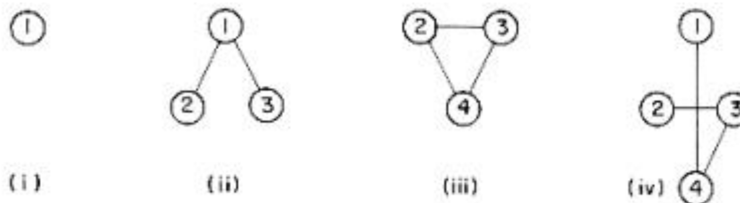


Figure 6.3 Example of a multigraph that is not a graph.

A subgraph of G is a graph G' such that V(G') ⊆V(G) and E(G') ⊆ E(G). Figure 6.4 shows some of the subgraphs of G1 and G3.



(a) Some of the subgraphs of G_1

(b) Some of the subgraphs of G_3

A *path* from vertex v_p to vertex v_q in graph G is a sequence of vertices v_p,v_i1,v_i2, ...,v_{in},v_q such that (v_p,v_i1),(v_i1,v_i2), ...,(v_{in},v_q) are edges in $E(G)$. If G' is directed then the path consists of <v_p,v_i1>,<v_i,v_i2>, ...,<v_{in},v_q>, edges in $E(G')$.

The length of a path is the number of edges on it. A simple path is a path in which all vertices except possibly the first and last are distinct. A path such as (1,2) (2,4) (4,3) we write as 1,2,4,3. Paths 1,2,4,3 and 1,2,4,2 is both of length 3 in G1.

The first is a **simple path** while the second is not. 1,2,3 is a simple directed path in G3. 1,2,3,2 is not a path in G3 as the edge <3,2> is not in E(G3).

A ***cycle*** is a simple path in which the first and last vertices are the same. 1,2,3,1 is a cycle in *G*1. 1,2,1 is a cycle in *G*3.

In an undirected graph, *G*, two vertices *v*1 and *v*2 are said to be ***connected*** if there is a path in *G* from *v*1 to *v*2 (since *G* is undirected, this means there must also be a path from *v*2 to *v*1).

A ***connected component*** or simply a component of an undirected graph is a *maximal* connected subgraph. *G*4 has two components *H*1 and *H*2.

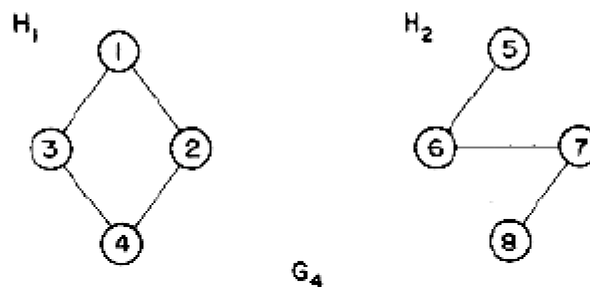


Figure 6.5 A graph with two connected components.

A ***tree*** is a connected acyclic (i.e., has no cycles) graph. A directed graph *G* is said to be ***strongly connected*** if for every pair of distinct vertices *vi*, *v*j in *V*(*G*) there is a directed path from *vi* to *vj* and also from *vj* to *vi*. The graph *G*3 is not strongly connected as there is no path from *v*3 to *v*2.

A ***strongly connected component*** is a maximal subgraph that is strongly connected. *G*3 has two strongly connected components.

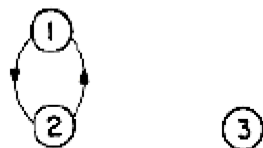


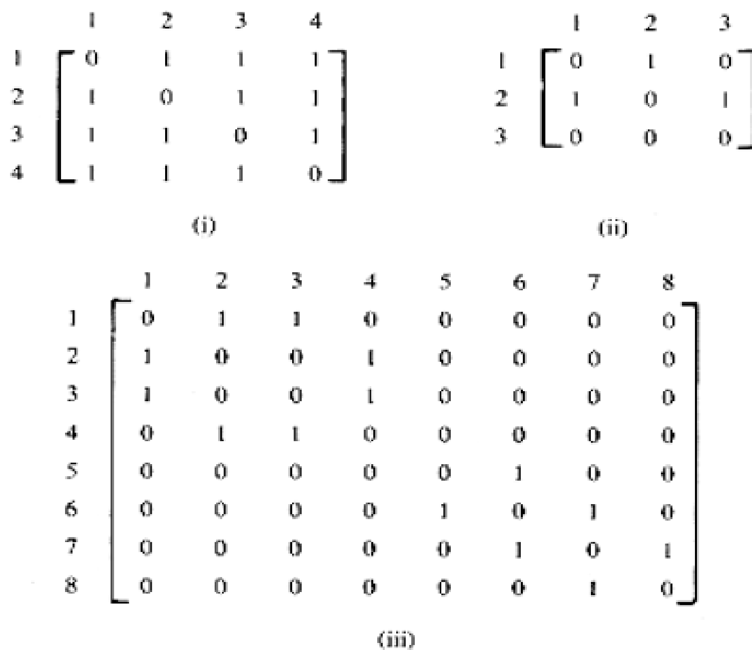
Figure 6.6 Strongly connected components of G_3.

Graph Representations

1. Adjacency Matrix
2. Adjacency Lists
3. Adjacency Multilists.

Adjacency Matrix.

Let G = (*V*, *E*) be a graph with *n* vertices, *n* 1. The adjacency matrix of *G* is a 2-dimensional nxn array, say *A*, with the property that *A* (*i*, *j*) = 1 iff the edge (v*i*, v*j*) (<v*i*,vj>for a directed graph) is in *E*(*G*). *A*(*i*,*j*) = 0 if there is no such edge in *G*. The adjacency matrices for the graphs *G*1, *G*3 and *G*4 are shown in figure 6.7.



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

(i)

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

(ii)

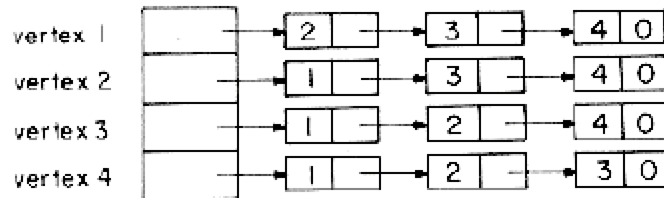
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	0	1	0	0	0	0
3	1	0	0	1	0	0	0	0
4	0	1	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	1	0
7	0	0	0	0	0	1	0	1
8	0	0	0	0	0	0	1	0

(iii)

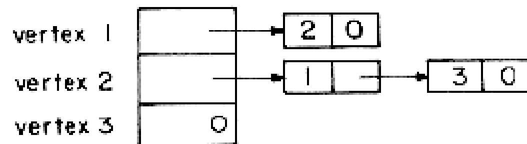
Figure 6.7 Adjacency matrices for (i) G_1, (ii) G_3 and (iii) G_4.

Adjacency Lists

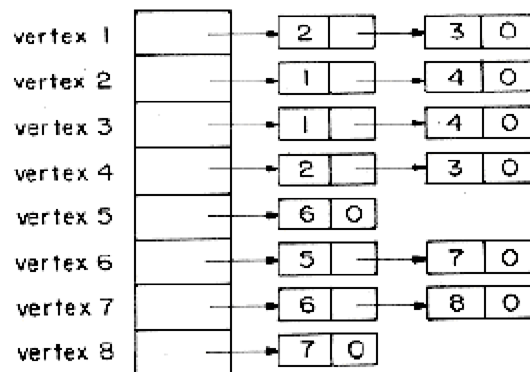
In this representation the *n* rows of the adjacency matrix are represented as *n* linked lists. There is one list for each vertex in *G*. The nodes in list *i* represent the vertices that are adjacent from vertex *i*. Each node has at least two fields: VERTEX and LINK. The VERTEX fields contain the indices of the vertices adjacent to vertex *i*. The adjacency lists for *G*1, *G*3 and *G*4 are shown in figure 6.8. Each list has a head node.



(i) Adjacency list for G_1



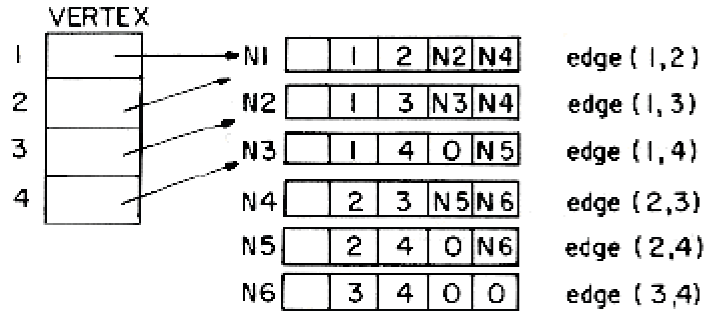
(ii) Adjacency lists for G_3



(iii) Adjacency list for G_4

Figure 6.8 Adjacency Lists

Adjacency Multilists




```
The lists are:  vertex 1:  N1 ← N2 ← N3

                vertex 2:  N1 ← N4 ← N5

                vertex 3:  N2 ← N4 ← N6

                vertex 4:  N3 ← N5 ← N6
```

Figure 6.12 Adjacency Multilists for G_1.

GRAPHS TRAVERSALS

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows.

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

Depth First Search

Depth first search of an undirected graph proceeds as follows. The start vertex *v* is visited. Next an unvisited vertex *w* adjacent to *v* is selected and a depth first search from *w* initiated. When a vertex *u* is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited which has an unvisited vertex *w* adjacent to it and initiate a depth first search from *w*. The search terminates when no unvisited vertex can be reached from any of the visited one. This procedure is best described recursively as in

```
procedure DFS(v)
//Given an undirected graph G = (V,E) with n vertices and an array
V1S1TED(n) initially set to zero, this algorithm visits all vertices
reachable from v. G and VISITED are global.//
VISITED (v) ← 1
for each vertex w adjacent to v do
if VIS1TED(w) = 0 then call DFS(w)
end
end DFS
```

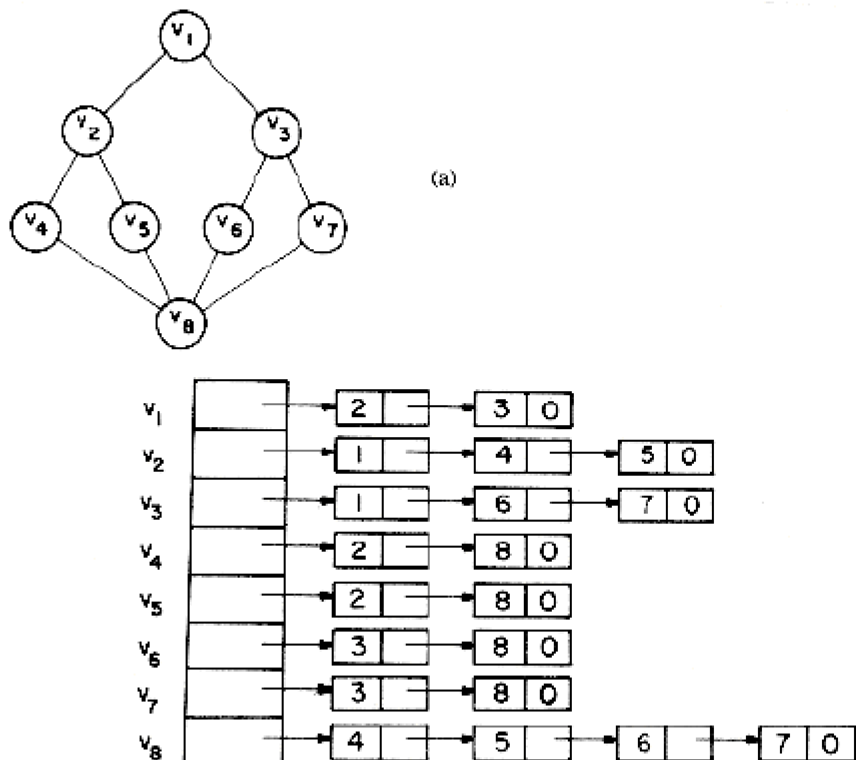


Figure 6.13 Graph G and Its Adjacency Lists.

The graph G of figure 6.13(a) is represented by its adjacency lists as in figure 6.13(b). If a depth first search is initiated from vertex $v1$, then the vertices of G are visited in the order: $v1$, $v2$, $v4$, $v8$, $v5$, $v6$, $v3$, $v7$. One may easily verify that DFS ($v1$) visits all vertices connected to $v1$. So, all the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

Breadth First Search

Starting at vertex v and marking it as visited, breadth first search differs from depth first search in that all unvisited vertices adjacent to v are visited next. Then unvisited vertices adjacent to these vertices are visited and so on. A breadth first search beginning at vertex $v1$ of the graph in figure 6.13(a) would first visit $v1$ and then $v2$ and $v3$. Next vertices $v4$, $v5$, $v6$ and $v7$ will be visited and finally $v8$. Algorithm BFS gives the details.

```
procedure BFS(v)
//A breadth first search of G is carried out beginning at vertex v.
All vertices visited are marked as VISITED(i)= 1. The graph G
and array VISITED are global and VISITED is initialized to zero.//
VISITED (v) ← 1
initialize Q to be empty          //Q is a queue//
loop
for all vertices w adjacent to v do
if VISITED(w) = 0            //add w to queue//
then [call ADDQ(w,Q); VISITED(w)←1]
//mark w as VISITED//
end
if Q is empty then return
call DELETEQ(v,Q)
forever
end BFS
```

Sorting and Searching:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order. **Searching in data**-structure refers to the process of finding a desired element in set of items. The set of items to be searched in, can be any **data-structure** like − list, array, linked-list, tree or graph. Search refers to locating a desired element of specified properties in a collection of items.

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

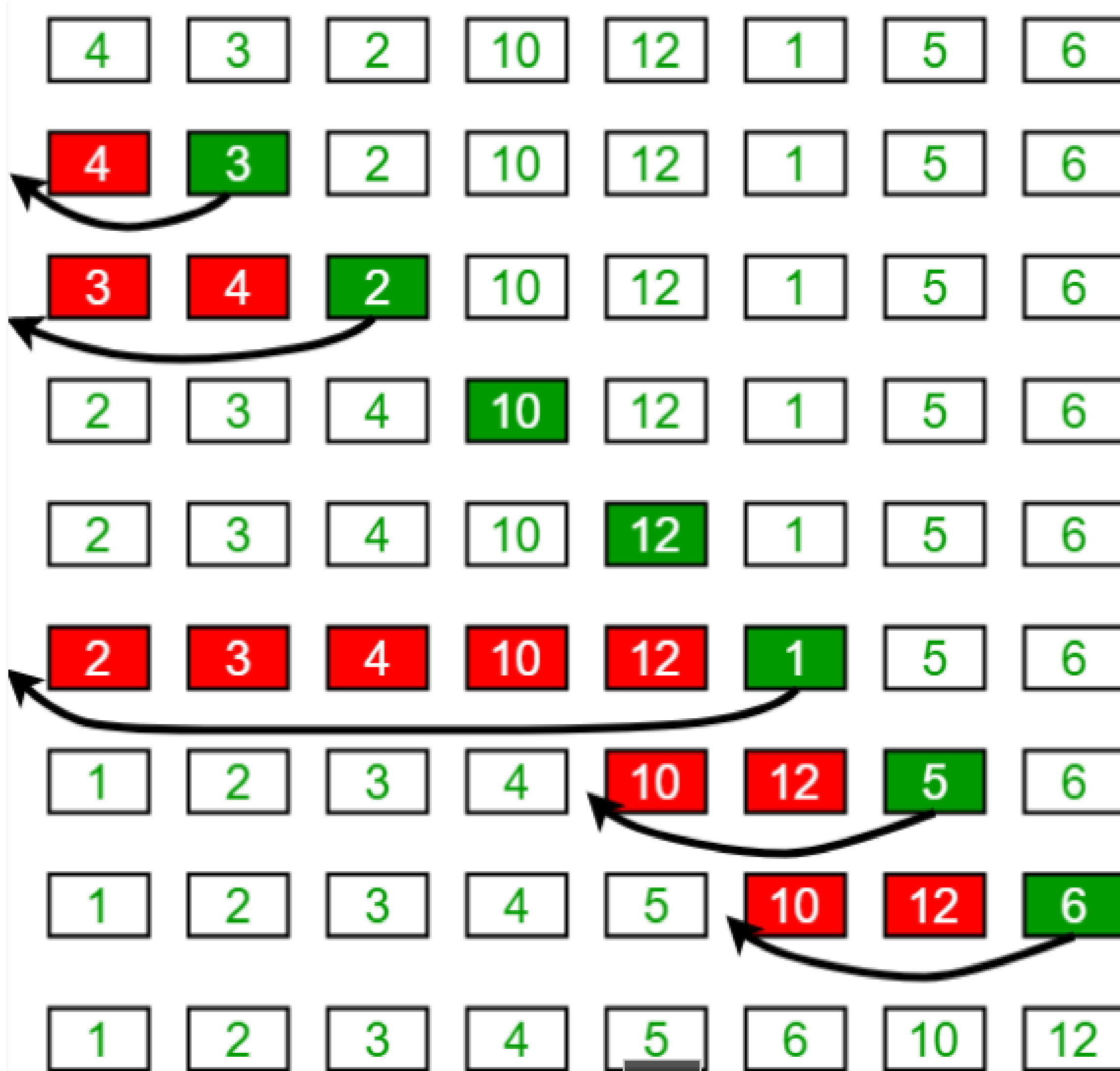
To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Insertion Sort Example:



```
/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Radix sort

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from **least significant digit to the most significant digit**. Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3-digit number then that list is sorted with 3 passes.

The Radix sort algorithm is performed using the following steps.

Step 1 - Define 10 queues each representing a bucket for each digit from 0 to 9.

Step 2 - Consider the least significant digit of each number in the list which is to be sorted.

Step 3 - Insert each number into their respective queue based on the least significant digit.

Step 4 - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.

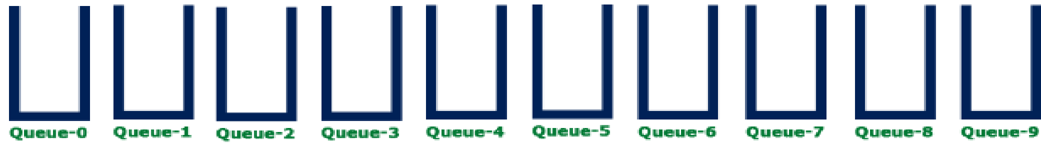
Step 5 - Repeat from step 3 based on the next least significant digit.

Step 6 - Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

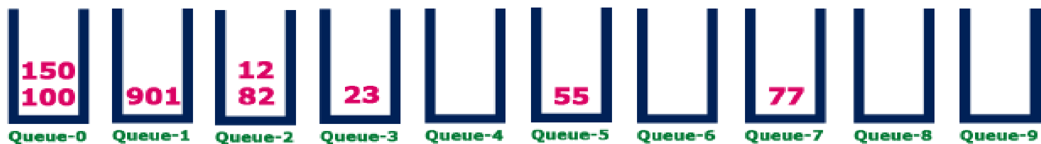
Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.



Pass - 1

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

8<u>2</u>, 90<u>1</u>, 10<u>0</u>, 1<u>2</u>, 15<u>0</u>, 7<u>7</u>, 5<u>5</u> & 2<u>3</u>



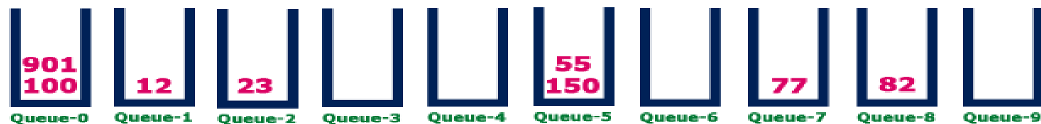
Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Pass - 2

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

1<u>0</u>0, 1<u>5</u>0, 9<u>0</u>1, <u>8</u>2, <u>1</u>2, <u>2</u>3, <u>5</u>5 & <u>7</u>7



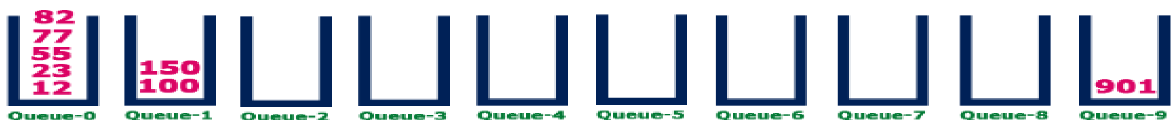
Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

<u>1</u>00, <u>9</u>01, 12, 23, <u>1</u>50, 55, 77 & 82

<u>1</u>00, <u>9</u>01, 12, 23, <u>1</u>50, 55, 77 & 82



Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

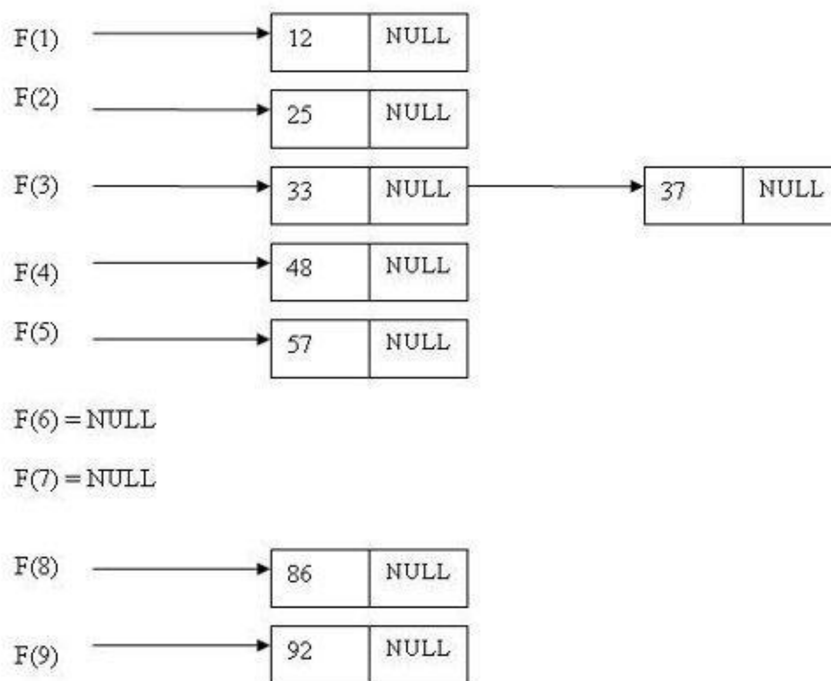
Address Calculation Sort.

This algorithm uses an **address** table to store the values which is simply a list (or array) of Linked lists. The Hash function is applied to each value in the array to find its corresponding **address** in the **address** table. After insertion, the values at each **address** in the **address** table are **sorted**.

In this method, a function fn() is applied to each key. The result of this function determines into which of the several sub-files the record is to be placed. The function should have the property that **x <= y, fn (x) <= fn (y).** Such a function is called order preserving. Thus, all of the records in one sub-file will have keys that are less than or equal to the keys of the records in another sub-file. An item is placed into a sub-file in correct sequence by using any sorting method; simple insertion is often used. After all the items of the original file have been placed into sub-files, the sub-files may be concatenated to produce the sorted result.

For-example:An-array,
25 57 48 37 12 92 86 33

Let us create ten sub-files, one for each of the ten possible first digits. Initially, each of these sub-files is empty. Consider an array of pointers f [10], where f[i] points to the first element in the file whose digit is i. After scanning the first element (i.e., 25), it is placed into the file header by f [2]. Each of the sub-files is maintained as a sorted linked list of the original array elements. After processing each of the elements in the original file, the sub-files appear as in the figure shown below,



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 the hash function H(x) maps the value X at the index x % 10 in a way.

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 way or hash i.e.

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

0	
1	11
2	12
3	13
4	15
5	15
.	
.	
.	
.	

Hash Table

Note: 1,2,3,4,5 is 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 operation takes place with help of hash value.
- Hence every entry in the hash table is associated with some key. (As per 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 dependent upon the size of the hash table.

Hash Function

- Is a function which is used to put the data in hash table. The integer is returned by the hash function is called hash key.
- A good hash function should satisfy 2 criteria
  1. A hash function should hash address such that ll keys are distriuted as evenly as possible among the various cells of the hash table.
  2. Computation of key should be simple.

Types of Hash Functions

Division Method:

- The hash function depends upon the remainder of division. Typically the divisor is the table length.

Example: 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 i.e:

H (54) =54%10=4

H (72) =72%10=2

H (89) =89%10=9

H (37) =37%10=7

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

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

Example:

Let key=2345, its square is K^2=574525

H (2345) =45=>by discarding 57 and 25

Folding Method:

- Here, the key K is divided into number of parts; K_1, K_2, K_3-------, Kn of same length except the last part. Then all parts are added together.

$$H(K)= K_1+K_2+K_{3+}\text{-------}+K_n$$

Example:

Let key K=123987234876

The partitions are:

K1=12, K2=39, K3=87, K4=23, K5=48, K6=76

Therefore H(123987234876) = 12+39+87+23+48+76

= 285

By Converting Keys to Integer:

- Here, If key K is a string. The hash value is obtained by converting the string into integer. i.e. by adding all ASCII values of each character in the string.

Example: 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 hashing 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 and chaining
- As the size is fixed, this type of hashing consist handling overflow of elements (Collision) efficiently.

Example:

Elements to be stored: 24, 93, 45

H (24) = 24 % 10 = 4

H (93) = 93 % 10 = 3

H (45) = 45 % 10 = 5

Index	Value
0	
1	
2	
3	93
4	24
5	45
6	
7	
8	
9	

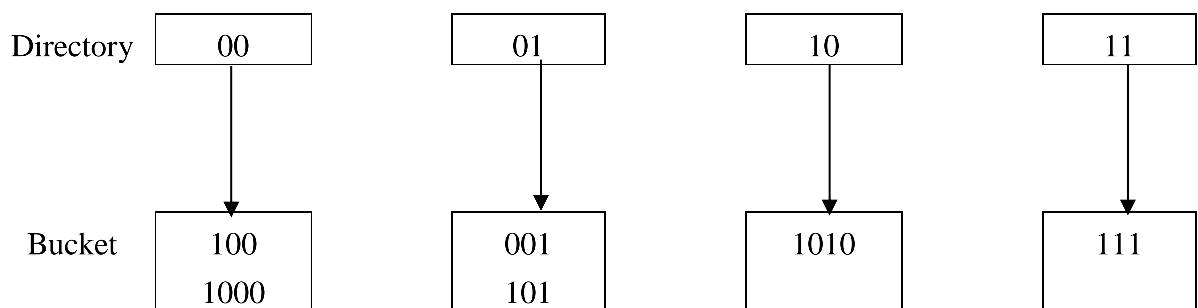
b. Dynamic Hashing:

- 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 referring to 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.

Example:

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

Hash function be K%10

0	
1	
2	
3	73
4	44
5	
6	
7	17
8	
9	

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

- The various technique using which collision can be avoided are:
  1. Open Addressing (Linear Probing)
  2. Chaining

1. Open Addressing (Linear Probing)

- Here, it involves static hashing, hash table size is fixed. In such a atble, 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, m is the size of hash table.

Items to be inserted: 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

0	1	2	……	34	……	48	……	56	……	98	99
				1234		2548		3256		1298	1299

- Collision is detected while inserting 1398 into 98th position. 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.

0	1	2	……	34	……	48	……	56	……	98	99
1398				1234		2548		3256		1298	1299

2. Chaining Method:

- Chaining technique avoids collision using an array of liked 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 and store the following words: like, a, tree, you, first, a, place, to

Let $H(str)=P_0+P_1+P_2+\ldots\ldots+P_{n-1}$; where P_i is position of letter in English alphabet series. Then calculate the hash address = Sum % 5

H(like) = 12 + 9 + 11 + 5 = 37 % 5 = 2

H(a) = 1 %5 =1

H(tree) = 20 + 18 + 5 + 5 = 48 % 5 = 3

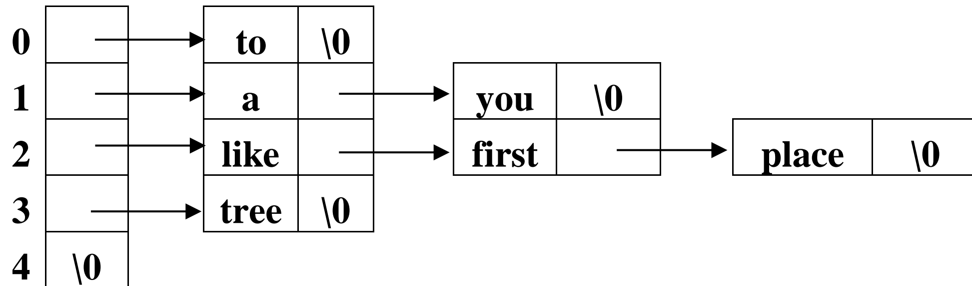
H(you) = 25 + 15 + 21 = 61 % 5 = 1

H(first) = 6 + 9 + 18 + 19 + 20 = 72 % 5 = 2

H(a) = 1 % 5 = 1

H(place) = 16 + 12 + 1 + 3 + 5 = 37 % 5 =2

H(to) = 20 + 15 = 35 % 5 = 0



Address Calculation Sort

- Uses Hashing technique.
- The hash function should have the property that x1 ≤ x2, the hash(x1) ≤ hash(x2). The function which exhibits this property is called order processing or Non-decreasing hashing function.

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

- Let us create 10 sub files, initially each of the sub files are empty.

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

- Here the largest number is 92, so we divide all the elements using the hash function h(k) = k/10.

H (25) = 25 / 10 = 2

H (57) = 57 / 10 = 5

H (48) = 48 / 10 = 4

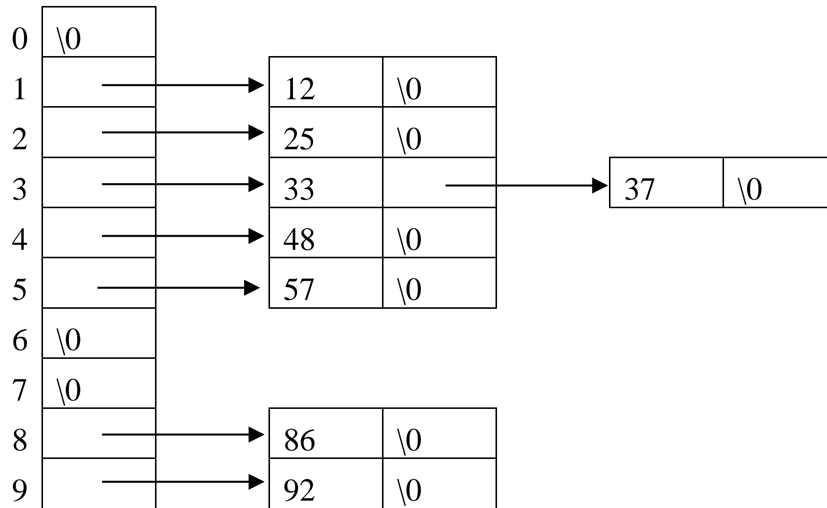
H (37) = 37 / 10 = 3

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 sub file only, but must be checked with the existing elements for its proper position in this sub file.\

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.

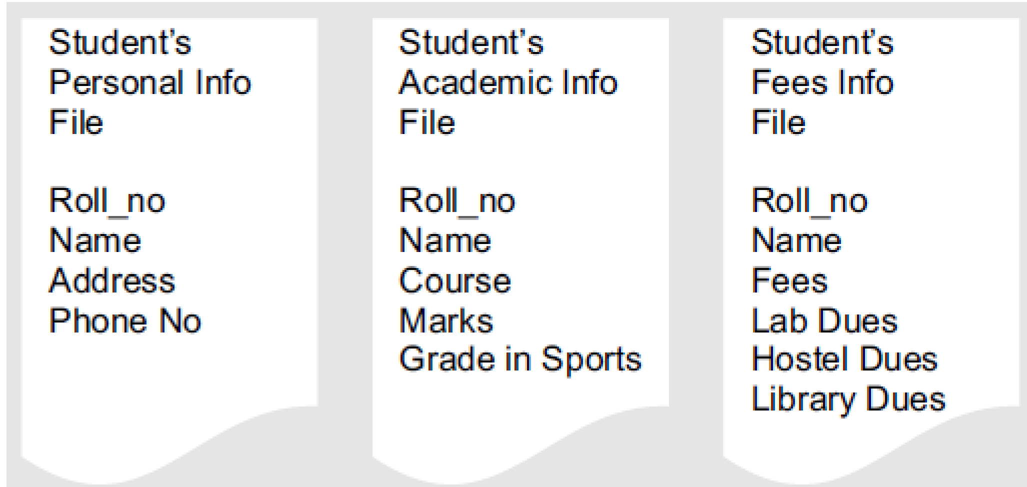


Figure Student directory

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

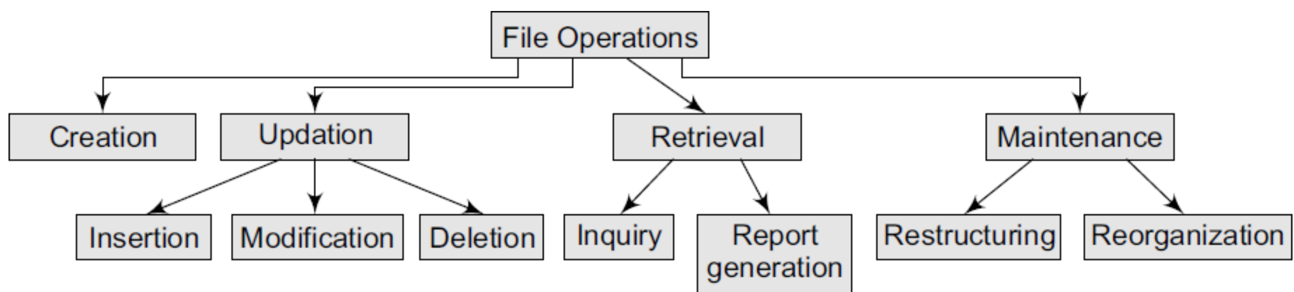


Figure File operations

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

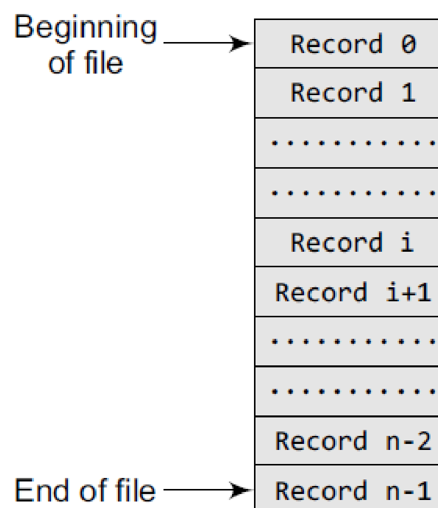


Figure Sequential file organization

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:

Address of ith record = base_address + (i–1) * 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

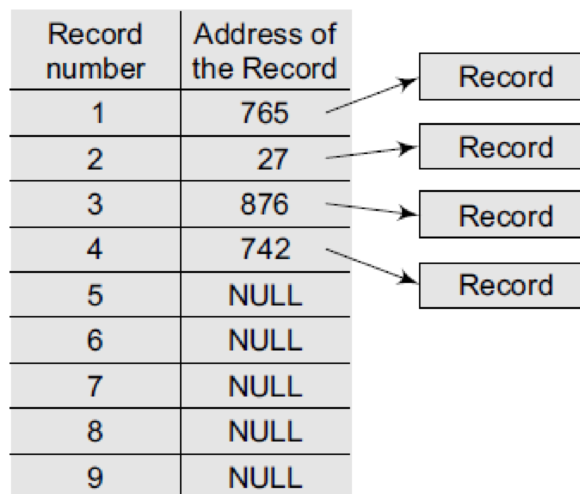


Figure Indexed sequential file organization

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.

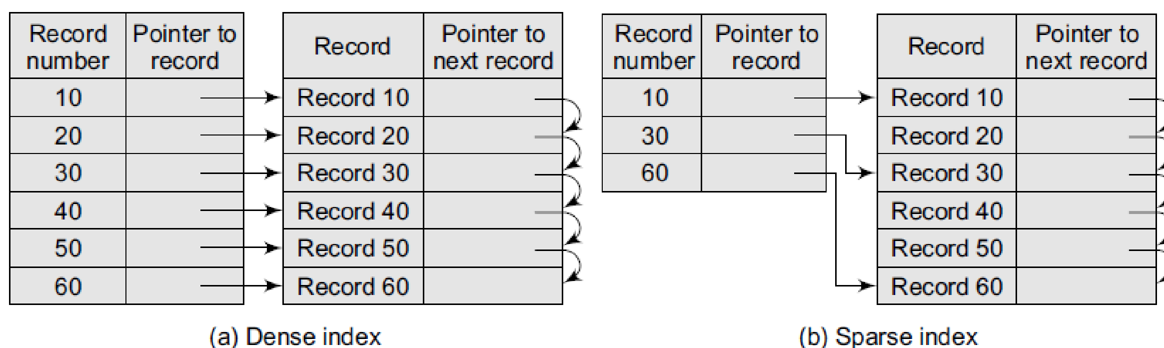


Figure Dense index and sparse index

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.

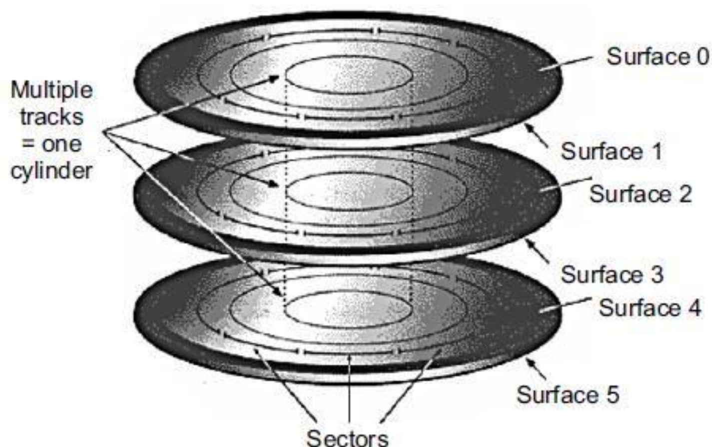


Figure Physical and logical organization of disk

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.

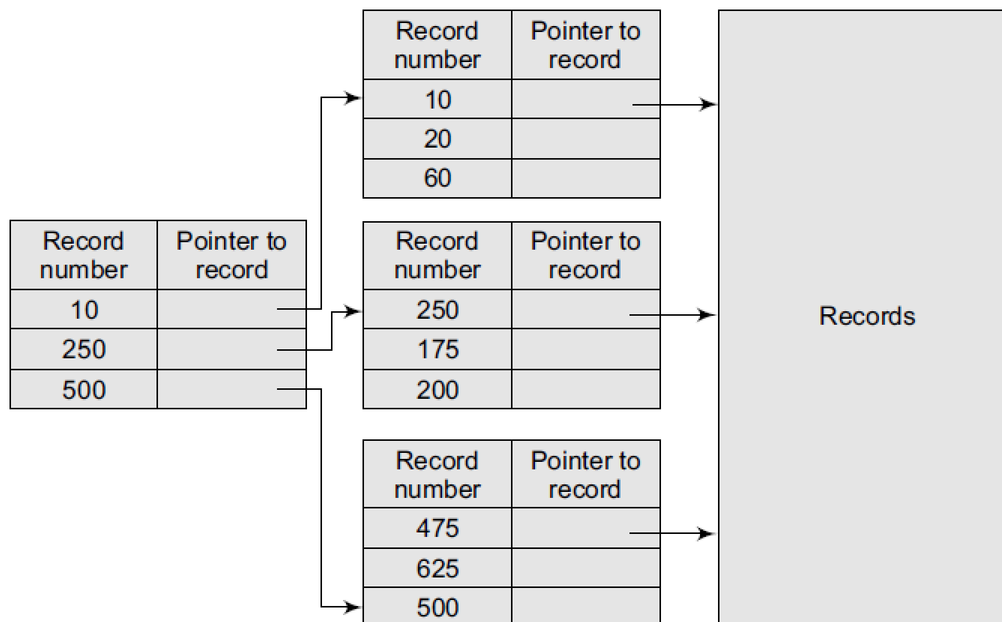


Figure Multi-level indices

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.

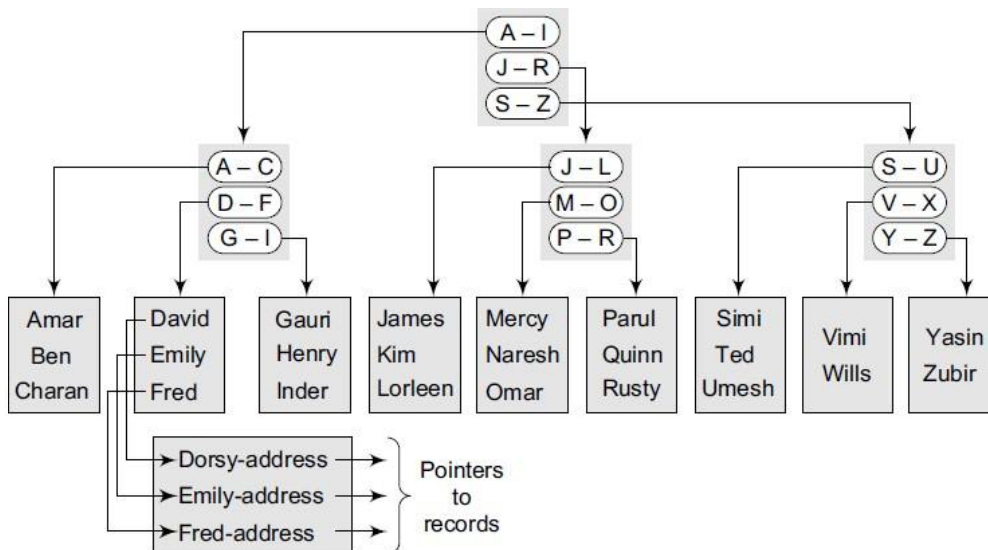


Figure B-tree index

- 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.

he 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.

he 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.