



RV Institute of Technology and Management®

**Rashtreeya Sikshana Samithi Trust**  
**RV Institute of Technology and Management®**  
(Affiliated to VTU, Belagavi)  
**JP Nagar, Bengaluru - 560076**  
**Department Computer Science and Engineering**  
**&**  
**Department Information Science and Engineering**



**Course Name : DATA STRUCTURES AND APPLICATIONS**  
**Course Code: BCS304**  
**III Semester**  
**2022 Scheme**

## Module-5

### Graph

#### 5.1 Introduction

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets  $(V, E)$ , where  $V$  is the set of vertices and  $E$  is the set of edges, connecting the pairs of vertices. Look at the following graph:

In the figure 5.1 graph,  $V = \{a, b, c, d, e\}$

$E = \{ab, ac, bd, cd, de\}$

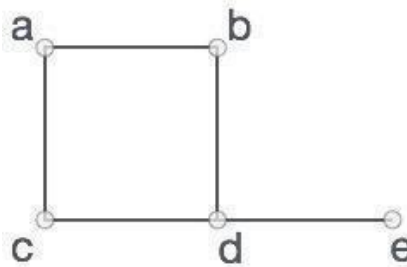


Figure: 5.1 pictorial representation of graph

#### Graph Data Structure

Mathematical graphs can be represented in data-structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms:

- **Vertex**

Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So, A to G are vertices. We can represent them using an array as shown in the below image. Here A can be identified by index 0, B can be identified using index 1 and so on.

- **Edge**

Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two-dimensional array to represent array as shown in the below image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency**

Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.

- **Path**

Path represents a sequence of edges between two vertices. In example given figure 5.2, ABCD represents a path from A to D.

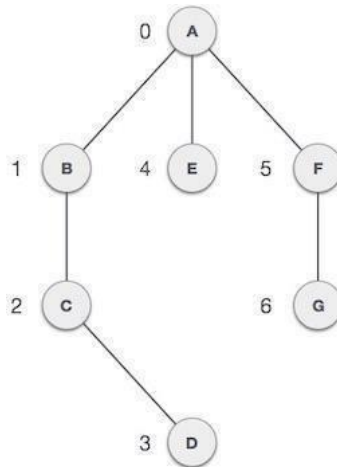


Figure:5.2 Graph Data Structure

### Basic Operations

Following are basic primary operations of a Graph which are following.

- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

### Traversal methods

## 5.2 Breadth First Search

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

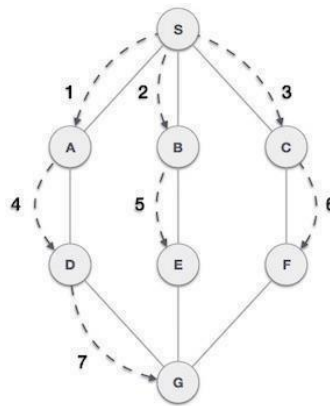
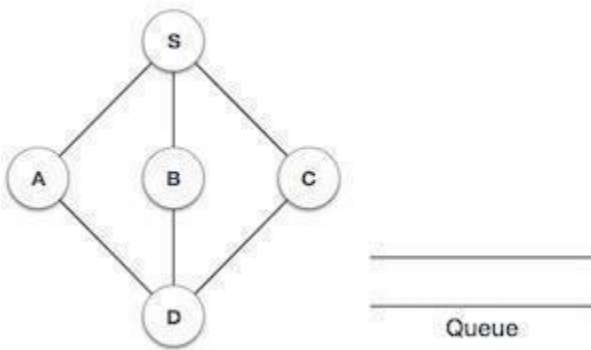


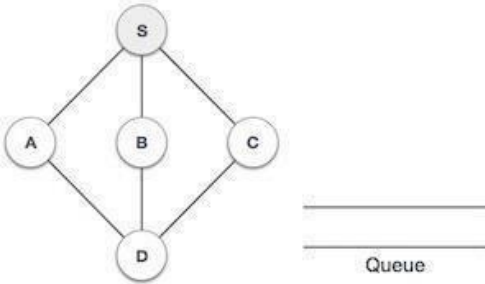
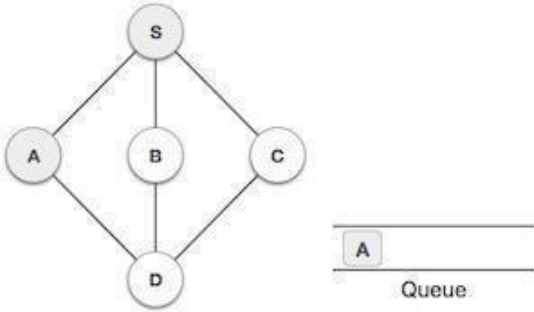
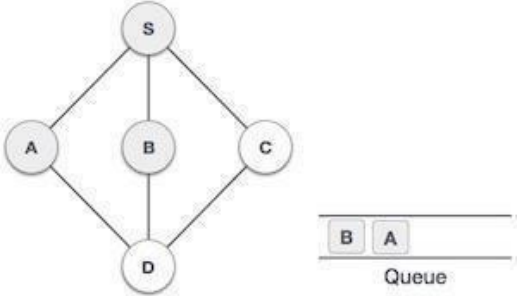
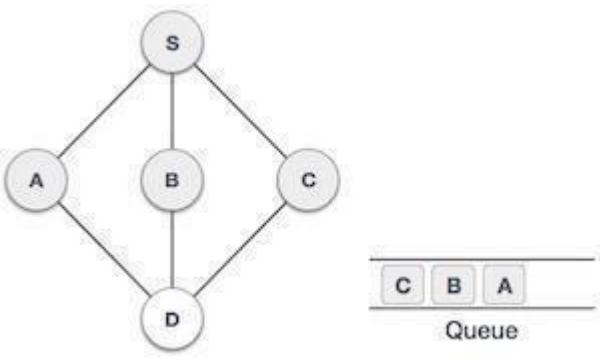
Figure:5.3 BFS algorithm

As in example given in the figure 5.3, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

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

Table 5.1 shows BFS algorithm implementation and traverse and perform certain operation.

Step	Traversal	Description
1.		Initialize the queue.

2.		We start from visiting <b>S</b> (starting node), and mark it visited.
3.		We then see unvisited adjacent node from <b>S</b> . In this example, we have three nodes but alphabetically we choose <b>A</b> mark it visited and enqueue it.
4.		Next unvisited adjacent node from <b>S</b> is <b>B</b> . We mark it visited and enqueue it.
5.		Next unvisited adjacent node from <b>S</b> is <b>C</b> . We mark it visited and enqueue it.

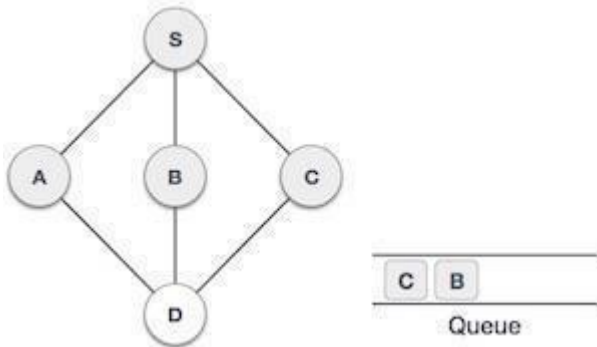
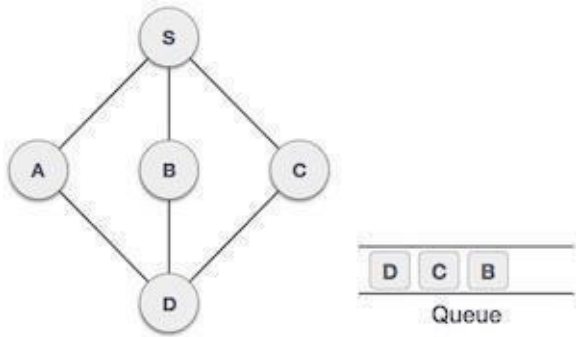
6 .		Now <b>S</b> is left with no unvisited adjacent nodes. So we dequeue and find <b>A</b> .
7 .		From <b>A</b> we have <b>D</b> as unvisited adjacent node. We mark it visited and enqueue it.

Fig:5.3 BFS algorithm

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.

### 5.3 Depth First Search

Depth First Search algorithm (DFS) traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

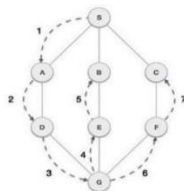


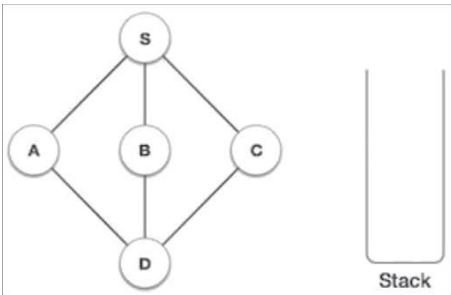
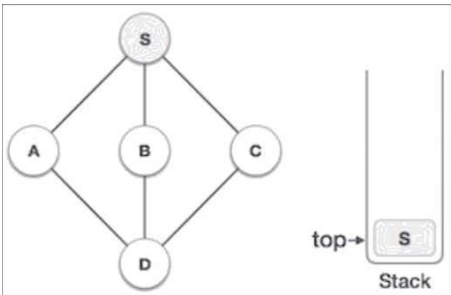
Figure 5.4: Depth First Search algorithm

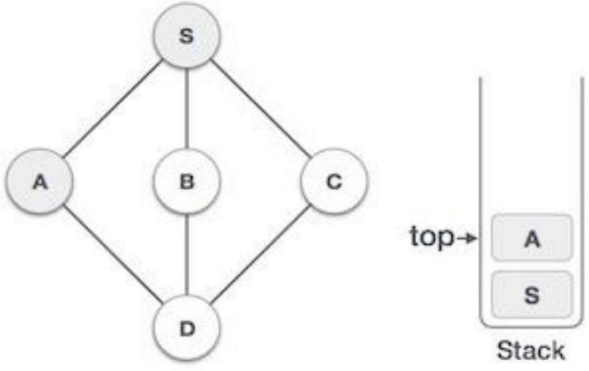
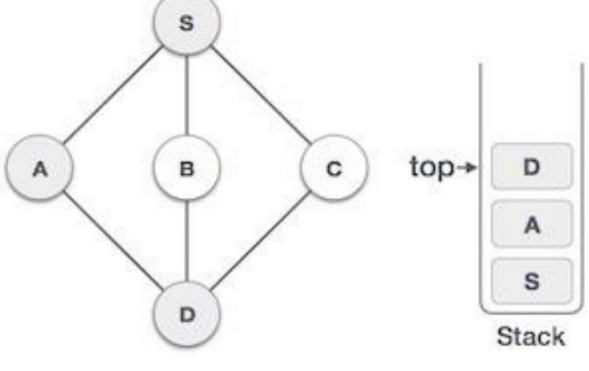
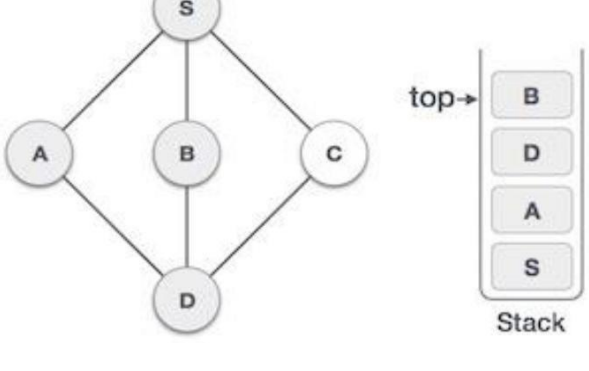
As in example given in figure 5.4 shows DFS algorithm traverses from A to B to C to D first

then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

Fig 5.3 shows DFS algorithm implementation and certain operation

Step	Traversal	Description
1		Initialize the stack
2		Mark <b>S</b> as visited and put it onto the stack. Explore any unvisited adjacent node from <b>S</b> . We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.

3.		<p>Mark <b>A</b> as visited and put it onto the stack. Explore any unvisited adjacent node from A.</p> <p>Both <b>S</b> and <b>D</b> are adjacent to <b>A</b> but we are concerned for unvisited nodes only.</p>
4.		<p>Visit <b>D</b> and mark it visited and put onto the stack. Here we have <b>B</b> and <b>C</b> nodes which are adjacent to <b>D</b> and both are unvisited. But we shall again choose in alphabetical order.</p>
5.		<p>We choose <b>B</b>, mark it visited and put onto stack. Here <b>B</b> does not have any unvisited adjacent node. So we pop <b>B</b> from the stack.</p>



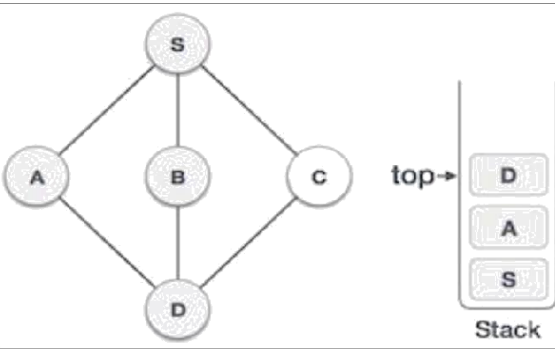
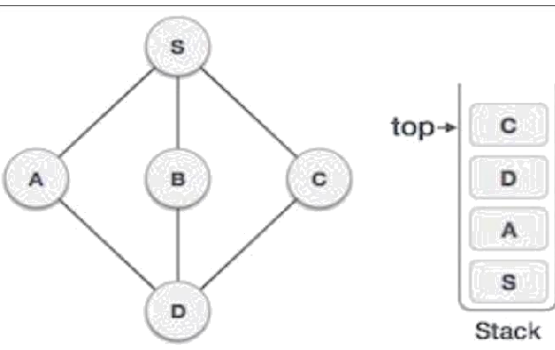
6.		<p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find <b>D</b> to be on the top of stack.</p>
7.		<p>Only unvisited adjacent node is from <b>D</b> is <b>C</b> now. So we visit <b>C</b>, mark it visited and put it onto the stack.</p>

Fig: 5.3 DFS algorithm

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

## 5.4 Sorting and Searching

### 5.4.1 Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list, must find its appropriate place and insert it there. Hence the name **insertion sort**.

The array is searched sequentially, and unsorted items are moved and inserted into sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-

case complexity are of  $O(n^2)$  where  $n$  is no. of items.

How insertion sort works?

We take an unsorted array for our example figure 5.5.a.



Figure: 5.5.a Insertion Sort

Insertion sort compares the first two elements shown in figure 5.5.b.



Figure: 5.5.b Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list shown in figure 5.5.c.



Figure: 5.5.c Insertion Sort

Insertion sort moves ahead and compares 33 with 27 shown in figure 5.5.d.



Figure: 5.5.d Insertion Sort

And finds that 33 is not in correct position shown in figure 5.5.e.



Figure: 5.5.e Insertion Sort

It swaps 33 with 27. Also, it checks with all the elements of sorted sub list. Here we see that sorted sub-list has only one element 14 and 27 is greater than 14. Hence sorted sub-list remain sorted after swapping shown in figure 5.5.f.



Figure: 5.5.f Insertion Sort

By now we have 14 and 27 in the sorted sub list. Next it compares 33 with 10 shown in the figure 5.5.g,



Figure: 5.5.g. Insertion Sort

These values are not in sorted order.



Figure: 5.5.h Insertion Sort

So, we swap them.



But swapping makes 27 and 10 unsorted.



So, we swap them too.



Again, we find 14 and 10 in unsorted order shown in figure 5.5.j.

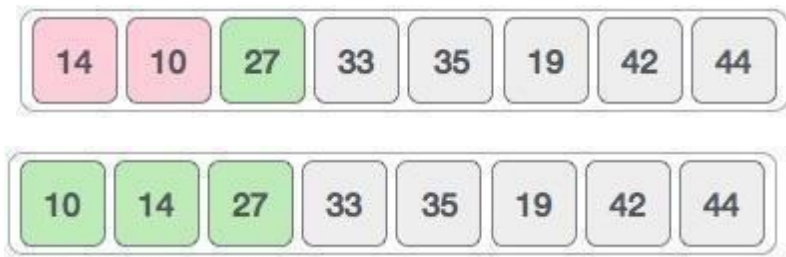


Figure: 5.5.j Insertion Sort

And we swap them. By the end of third iteration we have a sorted sub list of 4 items.

This process goes until all the unsorted values are covered in sorted sub list.

## 5.4.2 Radix sort

Consider the following 9 numbers:

493 812 715 710 195 437 582 340 385

We should start sorting by comparing and ordering the **one's** digits:

Fig 5.6.a unsorted list

Digit	Sub list
0	340 710
1	
2	812 582
3	493
4	
5	715 195 385
6	
7	437
8	
9	

Notice that the numbers were added onto the list in the order that they were found, which is why the numbers appear to be unsorted in each of the sub lists in table 5.3. Now, we gather the sub lists (in order from the 0-sub list to the 9-sub list) into the main list again:

340 710 812 582 493 715 195 385 437

Note: The **order** in which we divide and reassemble the list is **extremely important**, as this is one of the foundations of this algorithm.

Now, the sub lists are created again, this time based on the **ten's** digit in the table 5.3 b:

Fig 5.6b sub list on ten's digit

Digit	Sub list
0	
1	710 812 715
2	
3	437
4	340
5	
6	
7	
8	582 385
9	493 195

Now the sub lists are gathered in order from 0 to 9:

10 812 715 437 340 582 385 493 195

Finally, the sub lists are created according to the **hundred's** digit in the table 5.3.c:

Fig 5.6 c: sublist on hundred's digit

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	582
6	
7	710 715
8	812
9	

At last, the list is gathered up again:

195 340 385 437 493 582 710 715 812

And now we have a fully sorted array! Radix Sort is very simple, and a computer can do it fast. When it is programmed properly, Radix Sort is in fact **one of the fastest sorting algorithms** for numbers or strings of letters.

### 5.4.3 Address Calculation Sort

- In this method a function  $f$  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: if  $x \leq y$ ,  $f(x) \leq f(y)$ , Such a function is called order preserving.
- An item is placed into a sub file in correct sequence by placing sorting method – simple insertion is often used shown in figure 5.6.

*Example:*

25    57    48    37    12    92    86    33

Let us create 10 sub files. Initially each of these sub files is empty. An array of pointer  $f$  (10) is declared, where  $f(i)$  refers to the first element in the file, whose first digit is  $i$ . The number is passed to hash function, which returns its last digit (ten 's place digit), which is placed at that position only, in the array of pointers.

num=	25	-	$f(25)$	gives	2
57		-	$f(57)$	gives	5
48		-	$f(48)$	gives	4
37		-	$f(37)$	gives	3
12		-	$f(12)$	gives	1
92		-	$f(92)$	gives	9
86		-	$f(86)$	gives	8
33	-		$f(33)$ gives 3 which is repeated.		

Thus it is inserted in 3<sup>rd</sup> sub file (4<sup>th</sup>) only, but must be checked with the existing elements for its proper position in this sub file.

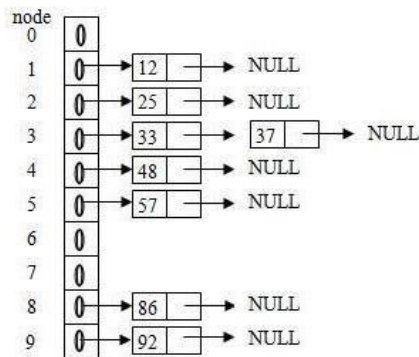


Figure 5.7. Memory Representation

## 5.5 Hashing

### 5.5.1 The Hash Table organizations

If we have a collection of  $n$  elements whose key are unique integers in  $(1, m)$ , where  $m \geq n$ , then we can store the items in a *direct address* table shown in figure 5.7.1,  $T[m]$ , where  $T_i$  is either empty or contains one of the elements of our collection.

Searching a direct address table is clearly an  $O(1)$  operation: for a key,  $k$ , we access  $T_k$ ,

5.5.1.1 if it contains an element, return it,

5.5.1.2 if it doesn't then return a NULL.

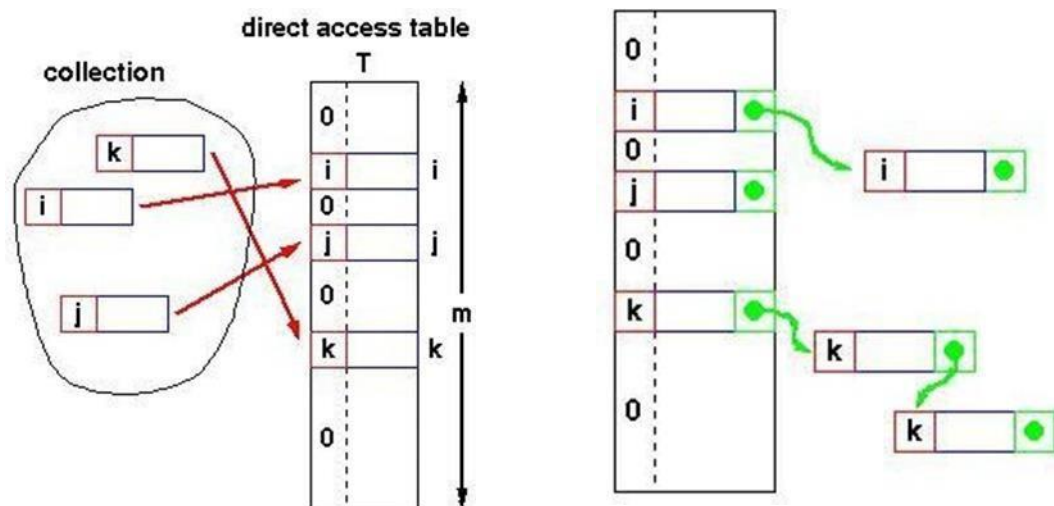


Figure 5.7.1: Hash Table

There are two constraints here:

1. the keys must be unique, and
2. the range of the key must be severely bounded.

If the keys are not unique, then we can simply construct a set of **m** lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be **O (1)**.

However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number of duplicates is **ndup<sup>max</sup>**, then searching for a specific element is **O(ndup<sup>max</sup>)**. If duplicates are the exception rather than the rule, then **ndup<sup>max</sup>** is much smaller than **n** and a direct address table will provide good performance. But if **ndup<sup>max</sup>** approaches **n**, then the time to find a specific element is **O(n)** and a tree structure will be more efficient.

The range of the key determines the size of the direct address table and may be too large to be practical. For instance, it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32- bit integers as their keys for a few years yet!

Direct addressing is easily generalized to the case where there is a function,

$$h(k) \Rightarrow (1, m)$$

which maps each value of the key, **k**, to the range (1, **m**). In this case, we place the element in **T[h(k)]** rather than **T[k]** and we can search in **O (1)** time as before.

### 5.5.2 Hashing Functions

The following functions map a single integer key (**k**) to a small integer bucket value **h(k)**. **m** is the size of the hash table (number of buckets).

**Division method** (Cormen) Choose a prime that isn't close to a power of 2.  $h(k) = k \bmod m$ . Works badly for many types of patterns in the input data.

**Knuth Variant on Division**  $h(k) = k(k+3) \bmod m$ . Supposedly works much better



than the raw division method.

**Multiplication Method** (Cormen). Choose  $m$  to be a power of 2. Let  $A$  be some random-looking real number. Knuth suggests  $M = 0.5 * (\sqrt{5} - 1)$ . Then do the following:

$s = k * A$

$x =$  fractional part of  $s$

$h(k) = \text{floor}(m * x)$

This seems to be the method that the theoreticians like.

To do this quickly with integer arithmetic, let  $w$  be the number of bits in a word (e.g. 32) and suppose  $m$  is  $2^p$ . Then compute:

$s = \text{floor}(A * 2^w)$   $x = k * s$

$h(k) = x \gg (w - p)$  // i.e. right shift  $x$  by  $(w - p)$  bits

// i.e. extract the  $p$  most significant

// bits from  $x$

### 5.5.3 Static and Dynamic Hashing

The good functioning of a hash table depends on the fact that the table size is proportional to the number of entries. With a fixed size, and the common structures, it is similar to linear search, except with a better constant factor. In some cases, the number of entries may be definitely known in advance, for example keywords in a language. More commonly, this is not known for sure, if only due to later changes in code and data. It is one serious, although common, mistake to not provide *any* way for the table to resize. A general-purpose hash table "class" will almost always have some way to resize, and it is good practice even for simple "custom" tables. An implementation should check the load factor, and do something if it becomes too large (this needs to be done only on inserts, since that is the only thing that would increase it).

To keep the load factor under a certain limit, e.g., under  $3/4$ , many table implementations expand the table when items are inserted. For example, in Java's Hash Map class the default load factor threshold for table expansion is  $3/4$  and in Python's dict., table size is resized when load factor is greater than  $2/3$ .

Since buckets are usually implemented on top of a dynamic array and any constant proportion for resizing greater than 1 will keep the load factor under the desired limit, the exact choice of the constant is determined by the same space-time tradeoff as for dynamic arrays.

Resizing is accompanied by a full or incremental table *rehash* whereby existing items are mapped to new bucket locations.

To limit the proportion of memory wasted due to empty buckets, some implementations also shrink the size of the table—followed by a rehash—when items are deleted. From the point of space-time tradeoffs, this operation is like the deallocation in dynamic arrays.

### Resizing by copying all entries

A common approach is to automatically trigger a complete resizing when the load factor exceeds some threshold  $r_{\max}$ . Then a new larger table is allocated, all the entries of the old table are removed and inserted into this new table, and the old table is returned to the free storage pool. Symmetrically, when the load factor falls below a second threshold  $r_{\min}$ , all entries are moved to a new smaller table.

For hash tables that shrink and grow frequently, the resizing downward can be skipped entirely. In this case, the table size is proportional to the maximum number of entries that ever were in the hash table at one time, rather than the current number. The disadvantage is that memory usage will be higher, and thus cache behavior may be worse. For best control, a "shrink-to-fit" operation can be if does this only on request.

If the table size increases or decreases by a fixed percentage at each expansion, the total cost of these resizing's, amortized over all insert and delete operations, is still a constant, independent of the number of entries  $n$  and of the number  $m$  of operations performed.

For example, consider a table that was created with the minimum possible size and is doubled each time the load ratio exceeds some threshold. If  $m$  elements are inserted into that table, the total number of extra re- insertions that occur in all dynamic resizing's of the table is at most  $m - 1$ . In other words, dynamic resizing roughly doubles the cost of each insert or delete operation.

#### ➤ Incremental resizing

Some hash table implementations, notably in real-time systems, cannot pay the price of enlarging the hash table all at once, because it may interrupt time-critical operations. If one cannot avoid dynamic resizing, a solution is to

perform the resizing gradually:

- ✓ During the resize, allocate the new hash table, but keep the old table unchanged.
- ✓ In each lookup or delete operation, check both tables.
- ✓ Perform insertion operations only in the new table.
- ✓ At each insertion also move  $r$  elements from the old table to the new table.
- ✓ When all elements are removed from the old table, deallocate it.

To ensure that the old table is completely copied over before the new table itself needs to be enlarged, it is necessary to increase the size of the table by a factor of at least  $(r + 1)/r$  during resizing.

Disk-based hash tables almost always use some scheme of incremental resizing, since the cost of rebuilding the entire table on disk would be too high.

### ➤ Monotonic keys

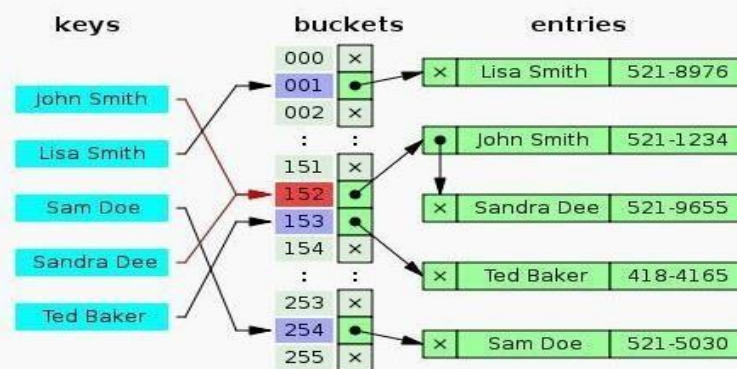


Fig 5.8 Hash collision resolved by separate chaining.

If it is known that key values will always increase (or decrease) monotonically, then a variation of consistent hashing can be achieved by keeping a list of the single most recent key value at each hash table resize operation (Refer Fig 5.8). Upon lookup, keys that fall in the ranges defined by these list entries are directed to the appropriate hash function—and indeed hash table—both of which can be different for each range. Since it is common to grow the overall number of entries by doubling, there will only be  $O(\log(N))$  ranges to check, and binary search time for the redirection would be  $O(\log(\log(N)))$ . As with consistent hashing, this approach guarantees that any key's hash, once issued, will never change, even when the hash table is later grown.

### ➤ Other solutions

Linear hashing is a hash table algorithm that permits incremental hash table

expansion. It is implemented using a single hash table, but with two possible lookup functions.

Another way to decrease the cost of table resizing is to choose a hash function in such a way that the hashes of most values do not change when the table is resized. This approach, called consistent hashing, is prevalent in disk-based and distributed hash tables, where rehashing is prohibitively costly.

## 5.6 Files and Their Organization

Nowadays, most organizations use data collection applications which collect large amounts of data in one form or other. For example, when we seek admission in a college, a lot of data such as our name, address, phone number, the course in which we want to seek admission, aggregate of marks obtained in the last examination, and so on, are collected. Similarly, to open a bank account, we need to provide a lot of input. All these data were traditionally stored on paper documents but handling these documents had always been a chaotic and difficult task. Similarly, scientific experiments and satellites also generate enormous amounts of data. Therefore, in order to efficiently analyse all the data that has been collected from different sources; it has become a necessity to store the data in computers in the form of files. In computer terminology, a file is a block of useful data which is available to a computer program and is usually stored on a persistent storage medium. Storing a file on a persistent storage medium like hard disk ensures the availability of the file for future use. These days, files stored on computers are a good alternative to paper documents that were once stored in offices and libraries.

### 5.6.1 DATA HIERARCHY

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

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

of all the suppliers, so on and so forth.

- A directory stores information of related files. A directory organizes information so that users can find it easily. For example, consider figure 5.9. below that shows how multiple related files are stored in a student directory.

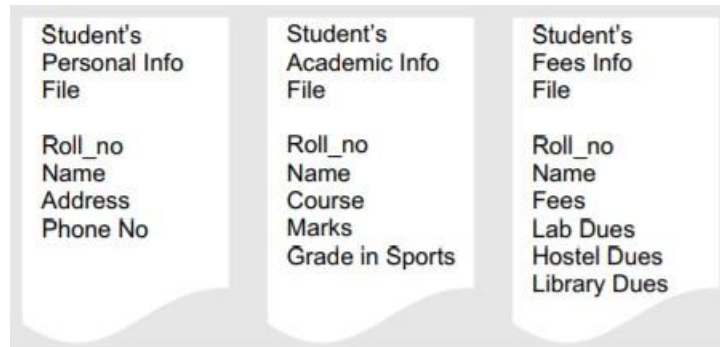


Figure 5.9: Student directory

## 5.6.2 FILE ATTRIBUTES

Every file in a computer system is stored in a directory. Each file has a list of attributes associated with it that gives the operating system and the application software information about the file and how it is intended to be used. A software program which needs to access a file looks up the directory entry to discern the attributes of that file. For example, if a user attempts to write to a file that has been marked as a read-only file, then the program prints an appropriate message to notify the user that he is trying to write to a file that is meant only for reading. Similarly, there is an attribute called hidden. When you execute the DIR command in DOS, then the files whose hidden attribute is set will not be displayed. These attributes are explained in this section.

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

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

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

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

**Attributes Flag** A file can have six additional attributes attached to it. These attributes are usually stored in a single byte, with each bit representing a specific attribute. If a particular bit is set to '1' then this means that the corresponding attribute is turned on. Table shows the list of attributes and their position in the attribute flag or attribute byte.

If a system file is set as hidden and read-only, then its attribute byte can be given as 00000111. We will discuss all these attributes here in this section shown in table 5.4. Note that the directory is treated as a special file in the operating system. So, all these attributes are applicable to files as well as to directories.

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

Table 5.4: Attribute flag

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

**Hidden** A file marked as hidden is not displayed in the directory listing.

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

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

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

**Archive** The archive bit is used as a communication link between programs that modify files and those that are used for backing up files. Most backup programs allow the user to do an incremental backup. Incremental backup selects only those files for backup which have been modified since the last backup.

When the backup program takes the backup of a file, or in other words, when the program archives the file, it clears the archive bit (sets it to zero). Subsequently, if any program modifies the file, it turns on the archive bit (sets it to 1). Thus, whenever the backup program is run, it checks the archive bit to know whether the file has been modified since its last run. The backup program will archive only those files which were modified.

### 5.6.3 TEXT AND BINARY 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, etc. However, 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 often denoted by placing a special character, called an end-of-file marker, after the last line in the text file.

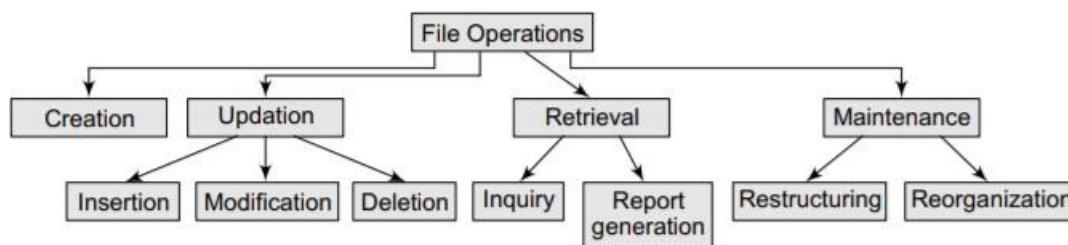
A binary file contains any type of data encoded in binary form for computer storage and processing purposes. A binary file can contain text that is not broken up into lines. A binary file stores data in a format that is like the format in which the data is stored in the main memory. Therefore, a binary file is not readable by humans and it is up to the program reading the file to make sense of the data that is stored in the binary file and convert it into something meaningful (e.g., a fixed length of record).

Binary files contain formatting information that only certain applications or processors can understand. It is possible for humans to read text files which contain only ASCII text, while 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. For example, only Microsoft Word can interpret the formatting information in a Word document.

Although text files can be manipulated by any text editor, they do not provide efficient storage. In contrast, binary files provide efficient storage of data, but they can be read only through an appropriate program.

### 5.6.4 BASIC FILE OPERATIONS

The basic operations that can be performed on a file are given in figure 5.9.



**Figure 5.9:** 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 (for example, changing the field width or adding/deleting fields). On the other hand, file reorganization may involve changing the entire organization of the file. We will discuss file organization in detail in the next section.

## 5.6.5 FILE ORGANIZATION

We know that a file is a collection of related records. The main issue in file management is the way in which the records are organized inside the file because it has a significant effect on the system performance. Organization of records means the logical arrangement of records in the file and not the physical layout of the file as stored on a storage media.

Since choosing an appropriate file organization is a design decision, it must be done keeping the priority of achieving good performance with respect to the most likely usage of the file.



Therefore, the following considerations should be kept in mind before selecting an appropriate file organization method:

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

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

In this section, we will discuss some of the techniques that are commonly used for file organization.

## Sequential Organization

A sequentially organized file stores the records in the order in which they were entered. That is, the first record that was entered is written as the first record in the file, the second record entered is written as the second record in the file, and so on. As a result, new records are added only at the end of the file.

Sequential files can be read only sequentially, starting with the first record in the file. Sequential file organization is the most basic way to organize a large collection of records in a file. Figure below shows  $n$  records numbered from 0 to  $n-1$  stored in a sequential file.

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

In sequential file organization, all the records have the same size and the same field format, and every field has a fixed size. The records are sorted based on the value of one field or a combination of two or more fields. This field is known as the key. Each key uniquely identifies a record in a file. Thus, every record has a different value for the key field. Records can be sorted in either ascending or descending order.

Sequential files shown in figure 5.10. are generally used to generate reports or to perform sequential reading of large amount of data which some programs need to do such as payroll processing of all the employees of an organization. Sequential files can be easily stored on both disks and tapes. Table 5.5 below (Sequential file organization summarizes the features, advantages, and disadvantages of sequential file organization.

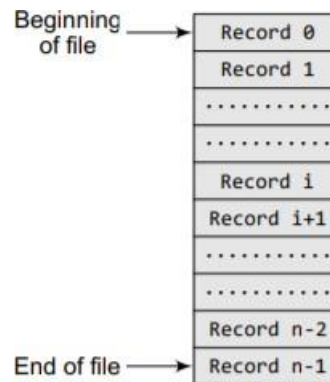


Figure 5.10: Sequential file organization

Features	Advantages	Disadvantages
<ul style="list-style-type: none"> <li>Records are written in the order in which they are entered</li> <li>Records are read and written sequentially</li> <li>Deletion or updation of one or more records calls for replacing the original file with a new file that contains the desired changes</li> <li>Records have the same size and the same field format</li> <li>Records are sorted on a key value</li> <li>Generally used for report generation or sequential reading</li> </ul>	<ul style="list-style-type: none"> <li>Simple and easy to handle</li> <li>No extra overheads involved</li> <li>Sequential files can be stored on magnetic disks as well as magnetic tapes</li> <li>Well suited for batch-oriented applications</li> </ul>	<ul style="list-style-type: none"> <li>Records can be read only sequentially. If <math>i^{\text{th}}</math> record has to be read, then all the <math>i-1</math> records must be read</li> <li>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</li> <li>Cannot be used for interactive applications</li> </ul>

## Relative File Organization

Relative file organization provides an effective way to access individual records directly. In a relative file organization, records are ordered by their relative key. It means the record number represents the location of the record relative to the beginning of the file. The record numbers range from 0 to  $n-1$ , where  $n$  is the number of records in the file. For example, the record with record number 0 is the first record in the file. The records in a relative file are of fixed length.

Therefore, in relative files, records are organized in ascending relative record number. A relative file can be thought of as a single dimension table stored on a disk, in which the relative record number is the index into the table. Relative files can be used for both random as well as sequential access. For sequential access, records are simply read one after another.

Relative files provide support for only one key, that is, the relative record number. This key must be numeric and must take a value between 0 and the current highest relative record number –1. This means that enough space must be allocated for the file to contain the records with relative record numbers between 0 and the highest record number –1. For example, if the highest relative record number is 1,000, then space must be allocated to store 1,000 records in the file. Figure below (Relative file organization) shows a schematic representation of a relative file which has been allocated enough space to store 100 records. Although it has space to accommodate 100 records, not all the locations are occupied. The locations marked as FREE are yet to store records in them. Therefore, every location in the table either stores a record or is marked as FREE.

Relative 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.5.11: Relative file organization

Relative file organization in figure 5.11 provides random access by directly jumping to the record which has to be accessed. If the records are of fixed length and we know the base address of the file and the length of the record, then any record  $i$  can be accessed using the following formula:

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

Note that the base address of the file refers to the starting address of the file. We took  $i-1$  in the formula because record numbers start from 0 rather than 1.

Table.5.6: Relative file organization

Features	Advantages	Disadvantages
<ul style="list-style-type: none"> <li>Provides an effective way to access individual records</li> <li>The record number represents the location of the record relative to the beginning of the file</li> <li>Records in a relative file are of fixed length</li> <li>Relative files can be used for both random as well as sequential access</li> <li>Every location in the table either stores a record or is marked as FREE</li> </ul>	<ul style="list-style-type: none"> <li>Ease of processing</li> <li>If the relative record number of the record that has to be accessed is known, then the record can be accessed instantaneously</li> <li>Random access of records makes access to relative files fast</li> <li>Allows deletions and updations in the same file</li> <li>Provides random as well as sequential access of records with low overhead</li> <li>New records can be easily added in the free locations based on the relative record number of the record to be inserted</li> <li>Well suited for interactive applications</li> </ul>	<ul style="list-style-type: none"> <li>Use of relative files is restricted to disk devices</li> <li>Records can be of fixed length only</li> <li>For random access of records, the relative record number must be known in advance</li> </ul>

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

Table 5.6 (Relative file organization) summarizes the features, advantages, and disadvantages of relative file organization.

### Indexed Sequential File Organization

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

The  $i^{\text{th}}$  entry in the index table points to the  $i^{\text{th}}$  record of the file. Initially, when the file is created, each entry in the index table contains NULL. When the  $i^{\text{th}}$  record of the file is written, free space is obtained from the free space manager and its address is stored in the  $i^{\text{th}}$  location of the index table. Now, if one has to read the 4<sup>th</sup> record, then there is no need to access the first three records. Address of the 4<sup>th</sup> record can be obtained from the index table and the record can be straightaway read from the specified address (742, in our example). Conceptually, the index sequential file organization can be visualized as shown in Fig below (Indexed sequential file organization).

Record number	Address of the Record	
1	765	Record
2	27	Record
3	876	Record
4	742	Record
5	NULL	
6	NULL	
7	NULL	
8	NULL	
9	NULL	

Figure.5.12: Indexed sequential file organization

An indexed sequential file shown in figure 5.12 uses the concept of both sequential as well as relative files. While the index table is read sequentially to find the address of the desired record, a direct access is made to the address of the specified record in order to access it randomly.

Indexed sequential files perform well in situations where sequential access as well as random access is made to the data. Indexed sequential files can be stored only on devices that support random access, for example, magnetic disks.

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

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

Fig 5.11 (Indexed sequential file organization) summarizes the features, advantages, and disadvantages of indexed sequential file organization

Features	Advantages	Disadvantages
<ul style="list-style-type: none"> <li>Provides fast data retrieval</li> <li>Records are of fixed length</li> <li>Index table stores the address of the records in the file</li> <li>The <i>i</i>th entry in the index table points to the <i>i</i>th record of the file</li> <li>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</li> <li>Indexed sequential files perform well in situations where sequential access as well as random access is made to the data</li> </ul>	<ul style="list-style-type: none"> <li>The key improvement is that the indices are small and can be searched quickly, allowing the database to access only the records it needs</li> <li>Supports applications that require both batch and interactive processing</li> <li>Records can be accessed sequentially as well as randomly</li> <li>Updates the records in the same file</li> </ul>	<ul style="list-style-type: none"> <li>Indexed sequential files can be stored only on disks</li> <li>Needs extra space and overhead to store indices</li> <li>Handling these files is more complicated than handling sequential files</li> <li>Supports only fixed length records</li> </ul>

## 5.7 INDEXING

An index for a file can be compared with a catalogue in a library. Like a library has card catalogues based on authors, subjects, or titles, a file can also have one or more indices.

Indexed sequential files are very efficient to use, but in real-world applications, these files are very large and a single file may contain millions of records. Therefore, in such situations, we require a more sophisticated indexing technique. There are several indexing techniques and each technique works well for a particular application. For a particular situation at hand, we analyse 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

### 5.7.1 Ordered Indices

Indices are used to provide fast random access to records. As stated above, a file may have multiple indices based on different key fields. 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. For 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. Indexed sequential files are a common example where a primary index is associated with the file.

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

### 5.7.2 Dense and Sparse Indices

In a dense index, the index table stores the address of every record in the file. However, in a sparse index, the index table stores the address of only some of the records in the file. Although sparse indices are easy to fit in the main memory, a dense index would be more efficient to use than a sparse index if it fits in the memory. Figure 5.13 (Dense index and sparse index) shows a dense index and a sparse index for an indexed sequential file.

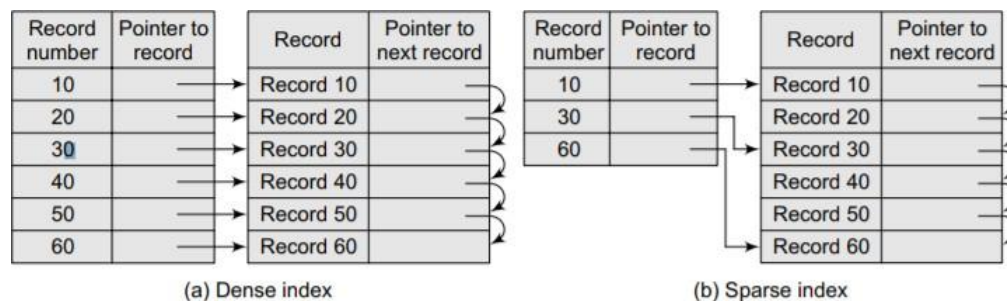


Figure 5.13: Dense index and sparse index

Note that the records need not be stored in consecutive memory locations. The pointer to the next record stores the address of the next record.

By looking at the dense index, it can be concluded directly whether the record exists in the file or not. This is not the case in a sparse index. In a sparse index, to locate a record, we 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, we 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. For 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.

Thus we see that sparse index takes more time to find a record with the given key. Dense indices are faster to use, while sparse indices require less space and impose less maintenance for insertions and deletions.

### 5.7.3 Cylinder Surface Indexing

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file. In a sequentially ordered file, the records are stored sequentially in the increasing order of the primary key. The index file will contain two fields—cylinder index and several surface indices. Generally, 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.

Each cylinder will have an entry corresponding to the largest key value into that cylinder. If the disk has  $n$  usable surfaces, then each of the surface indices will have  $n$  entries. Therefore, the  $i$ th entry in the surface index for cylinder  $j$  is the largest key value on the  $j$ th track of the  $i$ th surface. Hence, the total number of surface index entries is  $m.n$ . The physical and logical organization of disk is shown in figure 5.14 (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.



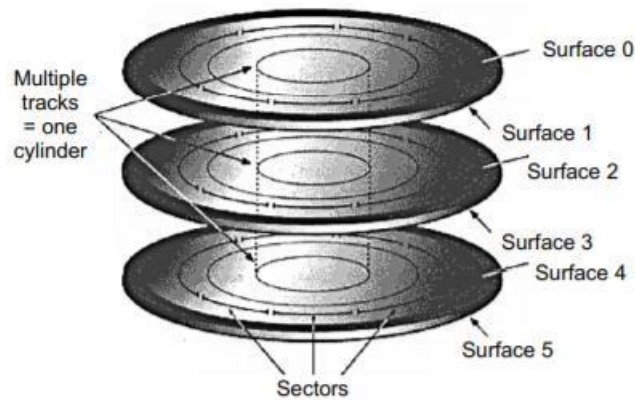


Figure 5.14: Physical and logical organization of disk

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. However, if track sizes are very large then it may not be a good idea to read the whole track at once. In such situations, we can also include sector addresses. But this would add an extra level of indexing and, therefore, the number of accesses needed to retrieve a record will then become four. In addition to this, when the file extends over several disks, a disk index will also be added.

The cylinder surface indexing method of maintaining a file and index is referred to as Indexed Sequential Access Method (ISAM). This technique is the most popular and simplest file organization in use for single key values. But with files that contain multiple keys, it is not possible to use this index organization for the remaining keys.

#### 5.7.4 Multi-level Indices

In real-world applications, we have 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. To understand this concept, consider a file that has 10,000 records. If we use simple indexing, then we need an index table that can contain at least 10,000 entries to point to 10,000 records. If each entry in the index table occupies 4 bytes, then we need an index table of  $4 \times 10000$  bytes = 40000 bytes. Finding such a big space consecutively is not always easy. So, a better scheme is to index the index table.

Figure. 5.15 (Multi-level indices) shows a two-level multi-indexing. We can continue further by having a three-level indexing and so on. But practically, we use two-level indexing. Note that

two and higher-level indexing must always be sparse, otherwise multi-level indexing will lose its effectiveness. 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.

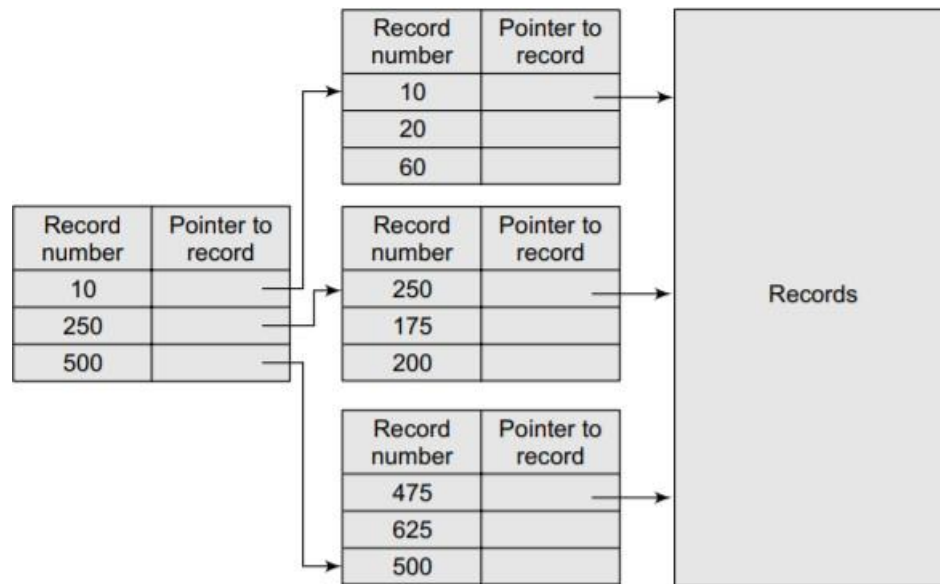


Figure 5.15: Multi-level indices

### 5.7.5 Inverted Indices

Inverted files are commonly 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.

For example, inverted files are widely used by bibliographic databases that may store author names, title words, journal names, etc. 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. Thus, 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 (also known as inverted file index or inverted file) stores a list of references to documents for each word

- A word-level inverted index (also known as 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. Although this technique needs more time and space, it offers more functionality (like phrase searches)

Therefore, the inverted file system consists of an index file in addition to a document file (also known as text file). It is this index file that contains all the keywords which may be used as search terms. For each keyword, an address or reference to each location in the document where that word occurs is stored. There is no restriction on the number of pointers associated with each word. For efficiently retrieving a word from the index file, the keywords are sorted in a specific order (usually alphabetically). However, the main drawback of this structure is that when new words are added to the documents or text files, the whole file must be reorganized. Therefore, a better alternative is to use B-trees.

### 5.7.6 B-Tree Indices

A database is defined as a collection of data organized in a fashion that facilitates updating, retrieving, and managing the data (that may include any item, such as names, addresses, pictures, and numbers). Most organizations maintain databases for their business operations. For example, an airline reservation system maintains a database of flights, customers, and tickets issued. A university maintains a database of all its students. These real-world databases may contain millions of records that may occupy gigabytes of storage space.

For a database to be useful, it must support fast retrieval and storage of data. Since it is impractical to maintain the entire database in the memory, B-trees are used to index the data in order to provide fast access

For example, searching a value in an un-indexed and unsorted database containing  $n$  key values may take a running time of  $O(n)$  in the worst case, but if the same database is indexed with a B-tree, the search operation will run in  $O(\log n)$  time.

Majority of the database management systems use the B-tree index technique as the default indexing method. This technique supersedes other techniques of creating indices, mainly due to its data retrieval speed, ease of maintenance, and simplicity. Figure 5.16 (B-tree index) shows a B-tree index.

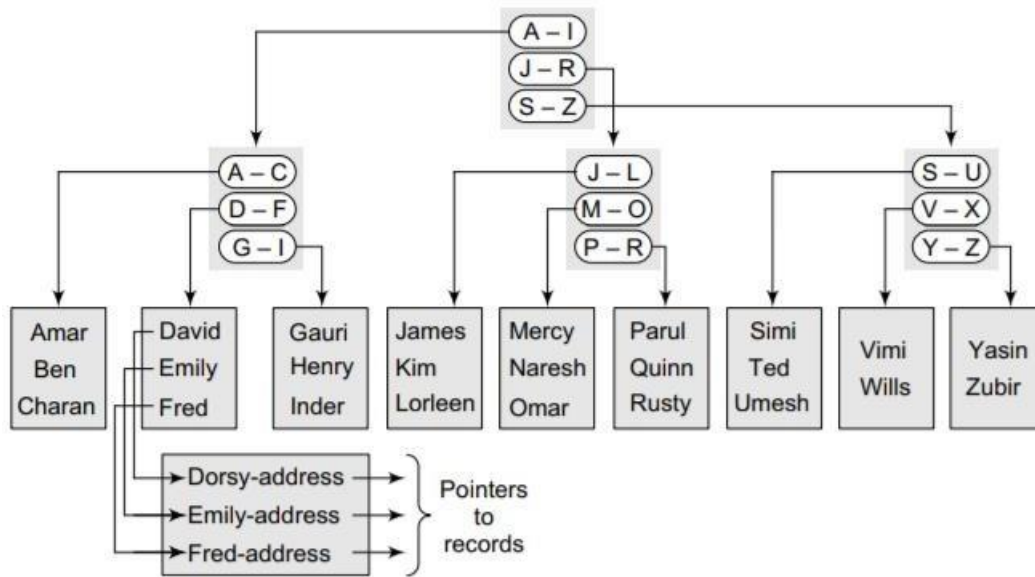


Figure 5.16: B-tree index

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. If a table has a column that has many unique values, then the selectivity of that column is said to be high. B-tree indices are most suitable for highly selective columns, but it causes a sharp increase in the size when the indices contain concatenation of multiple columns.

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 search 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.

### 5.7.7 Hashed Indices

So far, we have studied that hashing is used to compute the address of a record by using a hash function on the search key value. If at any point of time, 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, we mean two things. First, a good hash function, irrespective of the number of search keys, gives an average-case lookup that is a small constant. Second, the function distributes records uniformly and randomly among the buckets, where a bucket is defined as a unit of one or more records (typically a disk block). Correspondingly, the worst hash function is one that maps all the keys to the same bucket.

However, 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 file. This gives a good space–performance tradeoff. A hashed file organization uses hashed indices. Hashing is used to calculate the address of disk block where the desired record is stored. If  $K$  is the set of all search key values and  $B$  is the set of all bucket addresses, then a hash function  $H$  maps  $K$  to  $B$ . We can perform the following operations in a hashed file organization.

**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.

**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 (by comparing  $k_i$  with the key of every record) to finally retrieve the desired record with the given key value.

**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 (by comparing  $k_i$  with the key of every record). Then delete the record as we delete a node from a linear linked list.