42.Define Multi-way Search Tree.

Multi-way search tree of order 'n' is a general tree in which each node has 'n' or fewer sub tree and contains one less key than the number of sub trees.

43.Define B-Tree. What is the application of B-Tree?

A b-tree of order m is an m-way tree such that

-all leaves are on the same level

-all internal nodes except the root are constrained to have at most m non empty children and at least m/2 non empty children . The root has at most m non empty children.

A balanced search tree in which every node has between m/2 and m children, where m>1 is a fixed integer. m is the order. The root may have as few as 2 children. This is a good structure if much of the tree is in slow memory (disk), since the height, and hence the number of accesses, can be kept small, say one or two, by picking a large m.

Also known as balanced multiway tree.

A B-tree is essentially just a sorted list of all the item identifiers from one of your data files. For example, if you have a customer file, and every customer item in the file uses a customer number as the item identifier, and if you use B-TREE-P to create a B-tree for the customer file in ZIP code order, then the resulting B-tree will simply be a list of all the customer numbers sorted by ZIP code. However, the B-TREE-P subroutines keep the sorted B-tree list structured in a special fashion that makes it very fast and easy to find any number in the list.

Just as there has to be a file to contain customer data, there has to be a file to contain a B-tree. Naturally, a good convention (and one followed in the examples already presented) is to create a file called `B-TREE` for keeping the B-tree data that the B-TREE-P subroutines create. Initially, the `B-TREE` file is completely empty. Then, each time the `BTPINS` subroutine is called by a program, another item identifier is inserted into the `B-TREE` file, and the file becomes a specially sorted and constructed list of identifiers.

The order in which item identifiers are sorted in a B-tree is controlled by the `BTPKEY` subroutine. Although the statements in `BTPKEY` may specify a very complicated sort for controlling how items in a B-tree are ordered (for example, by ZIP code by address by company by name), the only data actually saved in a B-tree are item identifiers. Therefore, it doesn't matter how complicated the sort is, since the size of the resulting B-tree file is always the same. As a very rough rule of thumb, a B-tree for a file takes approximately the same amount of space as about two SELECT lists of the file.

The actual structure of a B-tree consists of a number of *nodes* stored as items in the B-tree's file. Each node contains a portion of the sorted list of identifiers in the B-tree, along with pointers to other nodes in the B-tree. The number of identifiers and pointers stored in each node is controlled by a special *size* parameter that is passed as the second argument to the `BTPINS` and `BTPDEL` subroutines. The size parameter indicates the minimum number of identifiers in a node, and also half the maximum. For example, in the examples already presented, the node size used was 5, so each node contained from 5 to 10 item identifiers.

The B-tree node size can be any number from 1 up. Small sizes create B-trees that may be faster to search, but take up more disk space because there are more nodes with pointers. Extremely small nodes may cause very "deep" B-trees that end up being slow to search. Larger node sizes slow down searches, but take less disk space because there are fewer pointers. The disk space occupied by nodes also depends on the length of your data file's item identifiers. A node size of 50 is often a good, all-around, starting value. Once the B-tree is built, it can be examined using standard Pick file maintenance techniques to find the optimum node size that keeps items in the B-tree file nicely packed within the boundaries of Pick's frame structure. If desired, the B-tree can then easily be rebuilt with the optimum node size.

The first node created in a B-tree file is numbered 0, the next is 1 (even though the node might be for a different B-tree in the same file), and the next node is 2, and so on. As more identifiers are inserted into the file's B-trees, more nodes are created. The special item named `NEXT.ID`, which is automatically created in every B-tree file, contains the number of the next node that will be created.

A file can contain any number of B-trees, but each B-tree in the file must have a unique name, which can be any string of characters. Each B-tree name is saved as an item in the B-tree file, and contains the number of the *root* node in the B-tree. The root node for a given B-tree is where all searches through that tree happen to start. In the B-TREE-P examples, the `B-TREE` file contained three different B-trees, named `ZIP`, `COMP`, and `LNAME`, so the `B-TREE` file also contained three items with those names.

45. Is Binary Tree a B-Tree? If yes, explain how?

A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the **root** of the tree. The other two subsets are themselves binary trees, called the **left** and **right sub trees** of the original tree. A left or right sub tree can be empty. Each element of a binary tree is called a **node** of the tree.
A B- Tree is a method of placing and locating files (called records or keys) in a database. The B- Tree algorithm minimizes the number of times a medium must be accessed to locate a desired record, thereby speeding up the process.
Unlike a binary tree, each node of a B tree may have a variable number of keys and children. The keys are stored in non-decreasing order. Each key has an associated child that is root of a sub tree containing all nodes with keys less than or equal to the key but greater than the preceding key. A node also has an additional right most child that is the root for a sub tree containing all keys greater than any keys in the node. The decision process at each node is more complicated in a B-Tree compared with a binary tree.

46. Explain why a tree structure should be made bushy as possible? Justify your answer with example.

A tree structure should be made bushy. As for example the binary search tree is organized in such a ay that all of the items less than the item in the chosen node are contained in the left sub tree and all the items greater than or equal to the chosen node are contained in the right sub tree. In this manner one does not have to search the entire tree for a particular item in the manner of linked list traversals. If a binary search tree is traversed in order (left, root, right) and the content of each node are printed as the node is visited, the numbers are printed in ascending order.
Binary trees can therefore be considered a self-sorting data structure. Consequently, search times through the data structure are, on average, greatly reduced.

48. What are the different types of sorting in general? What are the basic operations involved in sorting, explain with example?

Sorting means arranging the unordered data into order. Sorting is done in order to simplify the manual retrieval of the information or to make the machine access to data more efficiently. There are many different sorting algorithms. Generally, we say a file with 'n' number of records is said to be sorted if the key i<j implies that k[i] precedes k[j] in some ordering of the keys.

The two general types of sorting are internal sorts and the external sorts. If the records being sort are inside the main memory then it is called as the internal sort but if the records are being sorted is in the auxiliary storage it is called as the external sort.
Also the records of the file can be sorted in different ways depending upon the conditions. Some types of sorting are:

i. Exchange Sorts
ii. Selection and Tree sorting
iii. Insertion Sorts
iv. Merge and Radix Sorts

The two basic operations that can be involved in the sorting can be either the sorting of the records themselves or sorting of the auxiliary pointers that represent those records.

		fields			Fields
d 1		B			A
d 2		R			R
d 3		A			S
d 4		N			B
d 5		S			N

i. Original File　　　　　　　　ii. Sorted File

Figure: Sorting Actual Records

In the first case a sort can take place on the actual records themselves. This type of sorting will be efficient if the records present in the file are in less numbers.

In the second case if the amount of the data stored in the files are in large numbers and moving of those records are prohibitive then an auxiliary table of pointers may be used so that these pointers will be moved instead of moving the actual data. This method is also called as sorting by address. The table in the center is the file, and the table at the left is the initial table of pointers. During the sorting process, the entries in the pointer table are adjusted so that the final table is shown at the right. Here no original file entries are moved.

al r table				Pointer Table
	d 1 →	B	d 3 →	
	d 2 →	R	d 2 →	
	d 3 →	A	d 5 →	
	d 4 →	N	d 1 →	
	d 5 →	S	d 4 →	

Figure: Sorting by using an auxiliary table of pointers.

49.i. What are the factors involved in efficiency consideration of a computer program. Explain.

As we know that there are the great numbers of sorting techniques to sort a file, a programmer must be careful in deciding which sorting technique is to be used. Three of the most important considerations involved during the consideration of the program are: length of the time that must be spent by the programmer in coding a particular sorting program, the amount of machine time necessary for running the program, and the amount of space necessary for the program.

If a file is small, sophisticated sorting techniques designed to minimize time and space requirements are usually worse. Also spending lots of time in the program that will be used only once is also useless. So a programmer must be able to recognize the use of a particular sort. This brings other two efficiency considering parameters: Time and Space.
The programmer must often optimize one of these at the expense of other. We do not measure the time efficiency of a sort by the number of time units required but rather by the number of critical operations performed. Examples: movement or records or pointers, interchange of records etc.

The efficiency of various sorting techniques could be compared from the O-notation, where the *O* represents the complexity of the algorithm and a value *n* represents the size of the set the algorithm is run against.

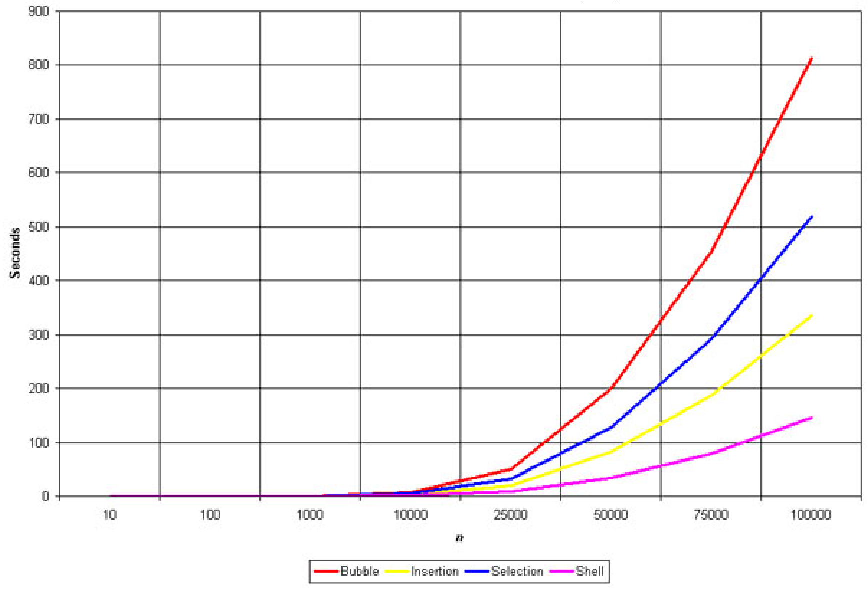
For example, *O* (*n*) means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items (10 * 10 = 100). If the complexity was $O\ (n^2)$ (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

The two classes of sorting algorithms are $O(n^2)$, which includes the bubble, insertion, selection, and shell sorts; and *O*(*n* log n) which includes the heap, merge, and quick sorts.

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together. In these empirical efficiency graphs the lowest line is the "best".

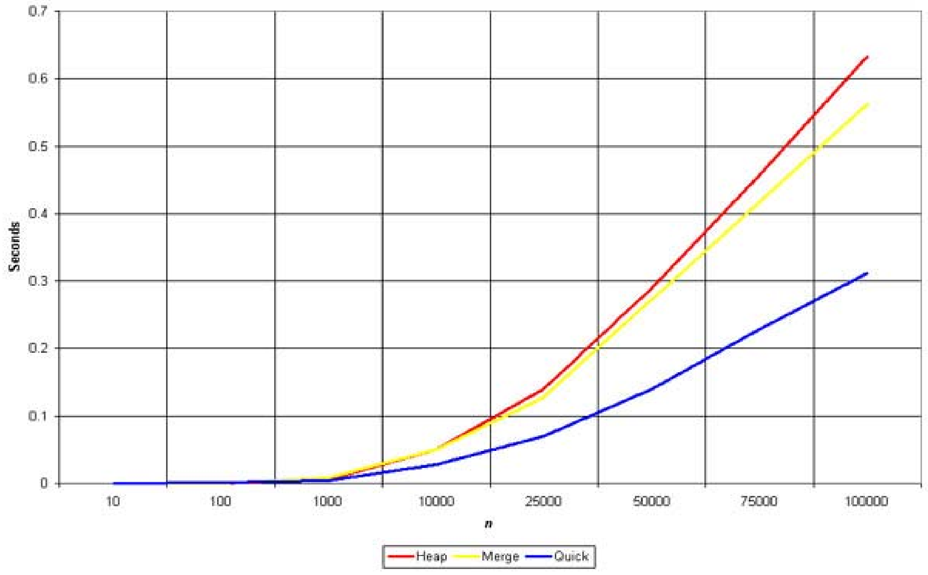
From the graph the bubble sort is inefficient and the shell sort is the best. On the other-side, all of these algorithms are simple and is useful for quick test programs, rapid prototypes, or internal-use software.

O (n^2) Sorts



In case of other sorts having the efficiency of O (n log n) notation:

O (n log n) Sorts



Notice that the time on this graph is measured in tenths of seconds, instead hundreds of seconds like the *O* (*n*^{2}) graph. But as with everything else in the real world, there are trade-offs. These algorithms are blazingly fast, but that speed comes at the cost of complexity. Recursion, advanced data structures, multiple arrays - these algorithms make extensive use of those nasty things.

49.ii.What are the factors involved in efficiency consideration of a computer program. Explain.

There are number of methods that can be used to sort a file. But the programmer must be aware of several inter-related and often conflicting efficiency considerations to make an intelligent choice about which sorting method is most appropriate to a particular problem. There are basically three most important factors involved in efficiency consideration of a computer program. These include:

- The length of time that must be spent by the programmer in coding a particular sorting program
- The amount of machine time necessary for running the program, and
- The amount of space necessary for the program.

If the file is small, the sophisticated sorting techniques designed to minimize space and time requirements are usually worst than simpler one. Similarly, if a particular sorting program is to be run only once and there is sufficient machine time and space in which to run it, it would be ridiculous to spend days in investigating the best methods. Therefore, a programmer must be able to recognize the fact that a particular sort is inefficient and must be able to justify its use in a particular situation.

Often we do not measure the time efficiency of a sort by the number of time units required but rather by number of critical operations performed. Examples of such critical operations are key comparisons(that is, the comparisons of the keys of two records in the file to determine which is greater)

- movement of records or pointers to records or
- Interchange of two records.

The critical operations chosen are those that take the most time.

50. I. Define Big 'O' Notation with some example.

If f(n) and g(n) be given two functions, we say that f(n) is on the order of g(n) or that f(n) is O(g(n)) if there exist positive integers a and b such that f(n)<= a*g(n) for all n>=b.

For example, if f(n)=n*n+100*n and g(n)=n*n, f(n) is O(g(n)), since n*n+100*n is less than or equal to 2*n*n for all n greater than or equal to 100. In this case a=2 and b=100. This same f(n) is also O(n*n*n), since n*n+100*n is less than or equal to 2*n*n*n for all n greater than or equal to 8. Given a function f(n), there may be many functions g(n) such that f(n) is O(g(n)).
If f(n) is O(g(n)),"eventually" (i.e., for n>=b) f(n) becomes permanently smaller or equal to some multiple of g(n). That is, f(n) is bounded by g(n) from above, or that f(n) is a "smaller" function than g(n). Another formal way of saying is that f(n) is asymptotically bounded by g(n). Yet another interpretation is that f(n) grows more slowly than g(n), since, proportionately(i.e., up to a factor of a), g(n) eventually becomes larger.

50.ii. Define Big 'O' notation with some example.

Big 'O' notation is a notation for measuring efficiency of some operation, especially in algorithm analysis and data structures. A Big O statement will denote rate of growth of the operation, which essentially indicates that operation's behavior in the worst case (say, a failed search of a binary tree). Typical growth rates can be c –constant,

log N- logarithmic ,log^2 N -log-squared, N –linear,N^2- quadratic ,N^3- cubic 2^N-exponentia.Some examples can be taken with the data structures:

- Binary Tree search is logarithmic (O = log N)
- Linked List search is linear (O = N)
- Skip List search is logarithmic (O = log N)

In another way, The *Big O* is the **upper bound** of a function. In the case of algorithm analysis, we use it to bound the worst-case running time, or the longest running time possible for *any* input of size *n*. We can say that the **maximum** running time of the algorithm is in the order of *Big O.*

For further explanation, let us consider two functions f(n) ad g(n),the f(n) is said to be the *order of* g(n)or that f(n) is *O*(g(n))if there exist positive integer s a and b such that f(n)<=a*g(n) for all n>=b. For example, f(n)=n^2+10n,and g(n)=n^2,then f(n) is *O*(g(n)) ,since n62+10n is less than or equal to2n^2 for all n greater than or equal to 10.In this case a equals 2 and b equals 10.However the same function f(n) is also order of *O*(n^3),as n62+10n is less than or equal to 2n^3 for all n greater or equal to 8.So,for a given function f(n),there may be more than one g(n) for *O*(g(n)) to be order of the function f(n)..Thus,n^2+10n is O(n^2)and n^2 is O(n^3).This property is called transitive property So, if f(n) is g(n) eventually f(n) becomes permanently smaller or equal to some multiple of g(n).Soft(n) is bounded by g(n),more formally f(n) is asymmetrically bounded by g(n).

51.i."There is a trade off between machine time and space required". Justify the statement.

In any process, generally there is a trade off between the machine time and the space required i.e. the improvement of machine time effect the space and vice versa. Taking sorting particularly, the amount of machine time necessary for running the sorting program and the amount of space necessary for the program are the main efficiency considerations of the sorting.

If a file or a program is small, then sophisticated sorting techniques can be designed in order to minimize the amount of space. But this makes the time requirements usually worse or marginally better in achieving efficiencies than that of the simpler techniques, but generally less efficient techniques. Similarly, if a particular sorting program is to be run only once then the machine time will be sufficient but the space in which the program is to run would be ludicrous .

It would be difficult for the programmer to spend days investigating the best methods of obtaining the last ounce of efficiency considering these considerations. However the programmer must be able to recognize the fact that a particular sort is inefficient and must be able to justify its use in a particular situation. Much time the designers and the planners are surprised at the inadequacy of their creation. To maximize the techniques and be cognizant of the advantages and disadvantages of each, so that when the need for a sort arises the programmer can supply the one which is most appropriate for the particular situation. This brings the considerations to the ***time*** and ***space*** while designing the sorting techniques.

In most of the computer applications, the programmer must often optimize either time or the space at the expense of the other. While considering the time required to sort a file of size ***n***, the actual time units are not concerned as these will vary from one machine to another. Instead, the corresponding change in the amount of time required to sort a file induced by a change in file size ***n*** is the matter of interest. This shows the relationship between the time and the space. Let us consider y is proportional to x such that multiplying x by a constant multiplies y by the same constant. Thus if y is proportional ton x, doubling x will double y, and multiplying x by 10 multiply y by 10.

There are different ways to determine the time requirements of a sort, neither of which yields that are applicable to all the cases. One is to go through a sometimes intricate and involved mathematical analysis of the various cases ,the result of which is often a formula giving the average time required for a particular sort as a function of file size that indicate the space required.

Considering these issues ,it can be noticed that there is a trade off between the machine time and the space both of which contribute equally to the efficiency of the process like sorting.

51.ii. "There is a trade off between machine time and space required". Justify the statement.

In any process, generally there is a trade off between the machine time and the space required i.e. the improvement of machine time effect the space and vice versa. Taking sorting particularly, the amount of machine time necessary for running the sorting program and the amount of space necessary for the program are the main efficiency considerations of the sorting.

If a file or a program is small, then sophisticated sorting techniques can be designed in order to minimize the amount of space. But this makes the time requirements usually worse or marginally betters in achieving efficiencies than that of the simpler techniques, but generally less efficient techniques. Similarly, if a particular sorting program is to be run only once then the machine time will be sufficient but the space in which the program is to run would be ludicrous .

It would be difficult for the programmer to spend days investigating the best methods of obtaining the last ounce of efficiency considering these considerations. However the programmer must be able to recognize the fact that a particular sort is inefficient and must be able to justify its use in a particular situation. Many times the designers and the planners are surprised at the inadequacy of their creation. To maximize the techniques and be cognizant of the advantages and disadvantages of each, so that when the need for a sort arises the programmer can supply the one, which is most appropriate for the particular situation. This brings the considerations to the **time** and **space** while designing the sorting techniques.

In most of the computer applications, the programmer must often optimize either time or the space at the expense of the other. While considering the time required to sort a file of size **n**, the actual time units are not concerned, as these will vary from one machine to another. Instead, the corresponding change in the amount of time required to sort a file induced by a change in file size **n** is the matter of interest. This shows the relationship between the time and the space. Let us consider y is proportional to x such that multiplying x by a constant multiplies y by the same constant. Thus if y is proportional ton x, doubling x will double y, and multiplying x by 10 multiply y by 10.

There are different ways to determine the time requirements of a sort, neither of which yields that are applicable to all the cases. One is to go through a sometimes intricate and involved mathematical analysis of the various cases, the result of which is often a formula giving the average time required for a particular sort as a function of file size that indicate the space required.

Considering these issues, it can be noticed that there is a trade off between the machine time and the space both of which contribute equally to the efficiency of the process like sorting.

52.What is exchange sort? Write an algorithm for binary sorting.

Comparing every two numbers in the list, and swapping them if the second number is less than the first do the exchange sort. This turns out to be a very inefficient N2 sort. It is done by comparing each adjacent pair of items in a *list* in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are done. *Complexity is $O(n^2)$ for arbitrary data, but approaches □(n) if the list is nearly in order at the beginning. Bi-directional bubble sort usually does better than bubble sort since at least one item is moved forward or backward to its place in the list with each pass. Bubble sort moves items forward into place, but can only move items backward one location each pass.* Since at least one item is moved into its final place for each pass, an improvement is to decrement the last position checked on each pass. one of the main characteristics of this sort is that it is easy to understand the program. but it is probably the least efficient. suppose x is an array of integer out of which the first n are to be sorted so that x[I]<=x[j] for 0<=I<j<n. it is straight forward to extend this simple format to one which is used in sorting n records , each with a sub field key k.

The basic idea underlying the bubble sort is to pass trough the file sequentially several times. Each pass consist of comparing the each element in the file with its successor (x[I] with x[I+1]) interchanging the two elements if they are not in the proper order.

The next sort we consider is the partition exchange sort or quick sort .let x be an array and n the number of elements in the array to be sorted .choose an element a from a specific position within the array (for e.g a can be chosen as the first element so that a=x[0]).suppose that the elements of x are portioned so that a is placed into the position j and the following condition hold .

52.What is exchange sort? Write an algorithm for binary sorting.

Comparing every two numbers in the list, and swapping them if the second number is less than the first do the exchange sort. This turns out to be a very inefficient N2 sort. It is done by comparing each adjacent pair of items in a *list* in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are done. *Complexity is $O(n^2)$ for arbitrary data, but approaches □(n) if the list is nearly in order at the beginning. Bi-directional bubble sort usually does better than bubble sort since at least one item is moved forward or backward to its place in the list with each pass. Bubble sort moves items forward into place, but can only move items backward one location each pass.* Since at least one item is moved into its final place for each pass, an improvement is to decrement the last position checked on each pass. one of the main characteristics of this sort is that it is easy to understand the program. but it is probably the least efficient. suppose x is an array of integer out of which the first n are to be sorted so that x[I]<=x[j] for 0<=I<j<n. it is straight forward to extend this simple format to one which is used in sorting n records , each with a sub field key k.

The basic idea underlying the bubble sort is to pass trough the file sequentially several times. Each pass consist of comparing the each element in the file with its successor (x[I] with x[I+1]) interchanging the two elements if they are not in the proper order. The next sort we consider is the partition exchange sort or quick sort .let x be an array and n the

number of elements in the array to be sorted .choose an element a from a specific position within the array (for e.g a can be chosen as the first element so that a=x[0]).suppose that the elements of x are portioned so that a is placed into the position j and the following condition hold

1.each of the elements in position 0 through j-1 is less tan or equal to a.

2.each of the elements in position j+1 through n-1 is greater than or equal to a.

so if this two condition hold for the particular a and j, a is the jth smallest element of x, so that a remains in position j when the array is completely sorted so if the forgoing process is repeated with the sub arrays x[0] through x[j-1]and x[j+1] through x[n-1] and any sub arrays created by this process in successive iteration finally we will get the sorted file.

ALGORITHM FOR BINARY SORTING

```
    1.    Input the first element.
    2.    Establish the first element as root as tree=maketree(x[0]).
    3.    Continue the process for successive element.
    4.    Assign I=1.
    5.    Check if I <the number of elements. If yes, goto step 6 else goto step 15.
    6.    Input the next element.
    7.    Keep the element in y as y=x[I].
    8.    Assign q=tree.
    9.    Assign p=q.
    10.   Check whether leaf has been reached or not.
    If yes, goto step 13,  else goto step 11.
    11.   Assign q=p.
    12.   Check if y<info(p).
    if yes, assign p=left(p)
    else assign p=right(p)
  12. Check if y<info(q)?
if yes,  set y to the left of q else set y to the right of q.
    13.   Increment I by 1
    14.   Goto step5
    15.   Transverse the tree in order
```

12 25 (48 37 33) 57 (92 86)

Where parenthesis encloses the sub arrays yet to be sorted. Repeating the process on the sub arrays further yields

12 25 (37 33) 48 57 (92 86)
12 25 (33) 37 48 57 (92 86)
12 25 33 37 48 57 (92 86)
12 25 33 37 48 57 (86) 92
12 25 33 37 48 57 86 92

Hence the final array is a sorted array.

In this way, the quick sort works.

53. i. Define how partition exchange sort works with an example.

Let us use the partition exchange sort (or quick sort) to the initial array given below:

25 57 48 37 12 92 86 33

The first element (25) placed in it proper position ,then the resulting array is

12 25 57 48 37 92 86 33

At this point, 25 are in its proper position in the array. It means each element in the array below that position is less than or equal to 25 and each element in the array above that position is greater than or equal to 25. Since 25 is in its final position, the original problem has been decomposed into the problem of sorting the two sub arrays

(12) And (57 48 37 92 86 33)

Since an array containing one element is already sorted, nothing has to be done to sort the first of the sub array. To sort the second sub array the process is repeated and the sub array is further subdivided. The entire array may now be viewed as

53.ii. Define how partition exchange sort works with an example.

Let us use the partition exchange sort (or quick sort) to the initial array given below:

26 57 48 37 12 92 86 33

The first element (25) placed in it proper position ,then the resulting array is

13 25 57 48 37 92 86 33

At this point, 25 are in its proper position in the array. It means each element in the array below that position is less than or equal to 25 and each element in the array above that position is greater than or equal to 25. Since 25 is in its final position, the original problem has been decomposed into the problem of sorting the two sub arrays

(12) And (57 48 37 92 86 33)

Since an array containing one element is already sorted, nothing has to be done to sort the first of the sub array. To sort the second sub array the process is repeated and the sub array is further subdivided. The entire array may now be viewed as

13 25 (48 37 33) 57 (92 86)

Where parenthesis encloses the sub arrays yet to be sorted. Repeating the process on the sub arrays further yields

12 25 (37 33) 48 57 (92 86)
12 25 (33) 37 48 57 (92 86)
12 25 33 37 48 57 (92 86)
12 25 33 37 48 57 (86) 92
12 25 33 37 48 57 86 92

Hence the final array is a sorted array. In this way, the quick sort works.

54. Compare the efficiency of the bubble sort and quick sort.

BUBBLE SORT:

Each of these algorithms requires ***n***-1 passes: each pass places one item in its correct place. (The ***n******th*** is then in the correct place also.) The ***i******th*** pass makes either ***I*** or ***n - i*** comparisons and moves. So:

$$\begin{aligned} T(n) &= 1+2+3+\ldots+(n-1) \\ &= \sum_{i=1}^{n-1} i \\ &= \tfrac{n}{2}(n-1) \end{aligned}$$

or **O(n^2)** Thus these algorithms are only suitable for small problems where their simple code makes them faster than the more complex code of the **O(*n* log*n*)** algorithm. It is likely that it is the number of interchanges rather than the number of comparisons that takes up the most time in the programs' execution.

QUICK SORT:
Assume that the file size n is a power of 2, say $n=2^m$, so that $m=\log_2 n$. Assume also the proper position for the pivot always turns out to be the exact middle of the sub array. In that case, there will be approximately n comparisons on the first pass, after which the file is split into two files of size n/2, approximately. For each of these files, there are approximately n/2 comparisons yielding four files each of size n/4 approximately. After halving the sub files m times, there are n files of size 1.Thus the total number of comparisons for the entire sort is approximately

n+2*n/2+4*n/4+…………………..+n*n/n

or, n+n+n+……………………….+n(m terms)

comparisons. There are m terms because the file is subdivided m times. Thus the total number of comparisons is O(n*m) or O(n log n).

Hence, the quick sort [**O(*n* log*n*)**] is more efficient than the bubble sort[**O(n^2)**] during the complex sorting.

56. Explain insertion sort. Calculate the efficiency of insertion sort for both worst case and best case.

An insertion is one that sorts a set of records by inserting records into existing sorted file. The simple insertion sort is viewed as a general selection sort in which the priority queue is implemented as an ordered array. Only the preprocessing phase of inserting the elements into the priority queue is necessary; once the elements have been inserted, they are already sorted, so that no selection is necessary.
The speed of the sort can be improved somewhat by using a binary search to find the proper positioning the sorted file. Another improvement to the simple insertion sort can be made by using list insertion. In this method there is an array link of pointers, one for each of the original array elements.

The efficiency of insertion sort for Worst case:

The array is in reversed sorted order, so f1=j for j=2………n.
Overall

$T(n) = an_2 + bn + c$ This is a quadratic function.

The efficiency of insertion sort for Best case:

The array is in sorted order, so $f_1=1$ for j=2………n.
Thus,

$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$

$T(n) = an + b$ This is a linear function.

57.i. What technique is used in Merge Sort? Explain.

Merging means the process of combining two or more things to form one. So, merge sort means the process of combining two or more sorted files to form a new complete file. Thus, the main technique used in Merge sort is dividing the file into 'n' number of sub-files of size 1 and merging adjacent pairs of files and perform sorting on them and makes n/2 files of size 2. Then, again merging adjacent pair of files and perform sorting on them which gives up with n/4 files of size 4. And, the same process is repeated until the final file does not become one file of size n.

For example:

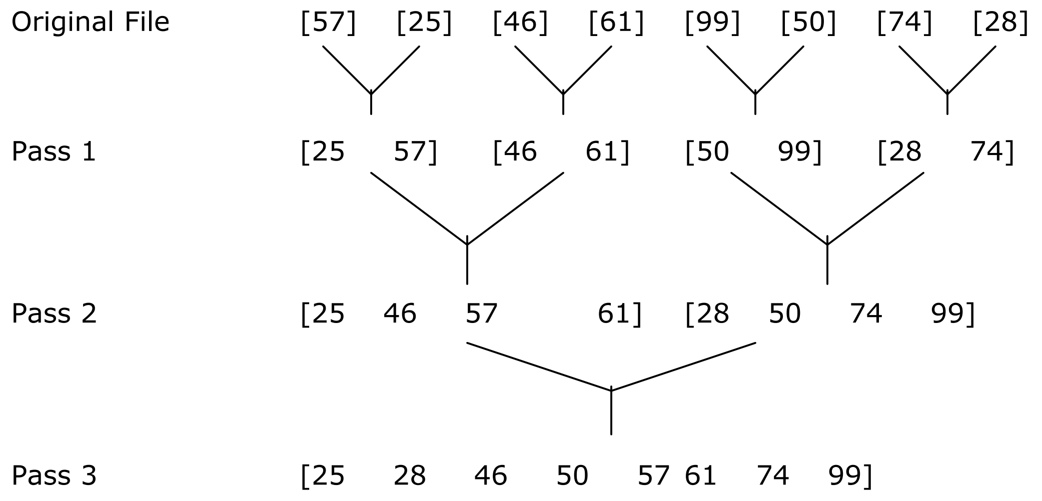

Original File [57] [25] [46] [61] [99] [50] [74] [28]
Pass 1 [25 57] [46 61] [50 99] [28 74]
Pass 2 [25 46 57 61] [28 50 74 99]
Pass 3 [25 28 46 50 57 61 74 99]

Figure 57.1: Successive Passes of the Merge Sort

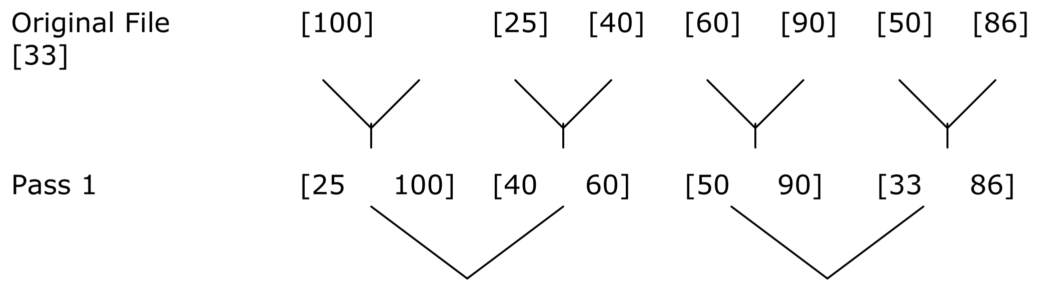
The above fig. illustrates the complete process operation on a sample file. Each file is contained within brackets.

In the Merge sort, the number of process does not exceed more than $\log_2(n)$ [E.g. $\log_2(8) = \log_{10}(8)/ \log_{10}(2) = 3$], each involving 'n' or few comparisons so the number of comparisons does not exceed more than $n * \log_2(n)$. And, Merge sort requires O(n) additional space for the auxiliary array, whereas quick sort requires only O(log n) additional space for the stack.

57.ii. What technique is used in Merge Sort? Explain.

Basically, merging means the process of combining two or more things to form one. So, **merge sort** means the process of combining two or more sorted files to form a new complete file. Thus, the main technique used in Merge sort is dividing the file into 'n' number of sub-files of size 1 and merging adjacent pairs of files and perform sorting on them and makes n/2 files of size 2. Then, again merging adjacent pair of files and perform sorting on them which gives up with n/4 files of size 4. And, the same process is repeated until the final file does not become one file of size n.

For e.g.


Original File [100] [25] [40] [60] [90] [50] [86]
[33]
Pass 1 [25 100] [40 60] [50 90] [33 86]

```
Pass 2        [25   40   60     100]   [33   50   86   90]

Pass 3        [25   33   40   50   60   86   90   100]
```

Fig. Successive Passes of the Merge Sort

The above fig. illustrates the complete process operation on a sample file. Each file is contained within brackets.
In the Merge sort, the number of process does not exceed more than $\log_2(n)$ [E.g. $\log_2(8) = \log_{10}(8)/ \log_{10}(2) = 3$], each involving 'n' or few comparisons so the number of comparisons does not exceed more than n* $\log_2(n)$. And, Merge sort requires O(n) additional space for the auxiliary array.

Q. 58 Define Radix Sort. Show the two different ways of implementing Radix Sort.
Radix Sort is defined as the process of sorting based on the values of the actual digits in the positional representations of the numbers being sorted. For e.g. the number 317 in decimal notation or in positional representation is written with a 3 in the hundreds position, a 1 in the tens position and a 7 in the units position.
Radix Sort can be implemented in two different ways: -
1) The first method is based on foregoing method. In this method, the given data's/ numbers are partitioned into ten groups (from 0 to 9) based on their most significant digit, but the necessary condition is that all the numbers should be of same size, if it not, it can be made by placing '0' i.e. zero in the front of the number. Thus, every element in the '0' group is less than every element in the '1' group, all of whose elements are less than every element in the '2' group and so on. Thus, we can sort within the individual groups based on the next significant digit and thus process is repeated till each sub-group has been sub-divided so the least significant digits are sorted. At this point, all the elements i.e. the original file is sorted. This method of sorting is also called Radix-Exchange Sort.

For e.g.
Original File: - 237 222 138 37 135 444 21
Each element is made of same size: -
237 222 138 037 135 444 021

Numbers are partitioned on the ten groups based on the MSD.

	Partitioned based on MSD	Sorting based on MSD	Sorting based on next to MSD i.e. LSD
Group 0	037 021	021 037	021 037
Group 1	138 135	138 135	135 138
Group 2	237 222	222 237	222 237
Group 3			
Group 4	444	444	444
Group 5			
Group 6			
Group 7			
Group 8			
Group 9			

After Sorting: - 021 037 135 138 222 237 444

2. The second method of Radix sorting is started with the least significant digit and ending with the most significant digit. Here, each number is taken in the order in which it appears in the file and is placed into a one of the ten queue, depending upon the value of digit currently being processed. Then, the numbers are restored into the original file from each queue starting with the queue of numbers with a '0' digit and ending with queue of number with a '9' digit. The process is repeated for each digit starting with the least significant digit and ending with the MSD and at last the file is sorted. This method is called Radix Sort.

For e.g.: -

Original File: - 25 19 17 26 33 49 30

Queue based on Least-Significant Digit : -

queue [0]	30	
queue [1]		
queue [2]		
queue [3]	33	
queue [4]		
queue [5]	25	
queue [6]	26	
queue [7]	17	
queue [8]		
queue [9]	19	49

Queue based on Most-Significant Digit : -

queue [0]		
queue [1]	17	19
queue [2]	25	26
queue [3]	30	33
queue [4]	49	
queue [5]		
queue [6]		
queue [7]		
queue [8]		
queue [9]		

Thus, sorted file is : - 17 19 25 26 30 33 49

58)Define Radix Sort. Show the two different ways of implementing Radix Sort.

Radix Sort is defined as the process of sorting based on the values of the actual digits in the positional representations of the numbers being sorted. For e.g. the number 589 in decimal notation or in positional representation is written with a 5 in the hundreds position, a 8 in the tens position and a 9 in the units position.

Radix Sort can be implemented in two different ways: -

1) The first method is based on foregoing method. In this method, the given data or numbers are partitioned into ten groups (from 0 to 9) using the decimal base. Thus, every element in the '0' group is less than every element in the '1' group, all of whose elements

are less than every element in the ‘2’ group and so on. Thus, we can sort within the individual groups based on the next significant digit and thus process is repeated till each sub-group has been sub-divided so the least significant digits are sorted. At this point, all the elements i.e. the original file is sorted. This method of sorting is also called Radix-Exchange Sort.

For e.g.

Original File: -   732   123   128   30   511   555   10

Each element is made of same size: -

732   123   198   030   511   555   010

Numbers are partitioned on the ten groups based on the MSD.

	Partitioned based on MSD	Sorting based on MSD	Sorting based on next to MSD i.e. LSD
Group 0	030 010	010 030	010 030
Group 1	128 123	128 123	123 198
Group 2	555 511	511 555	511 555
Group 3			
Group 4	732	732	732
Group 5			
Group 6			
Group 7			
Group 8			
Group 9			

After Sorting: -   010   030   123   128   511   555   732

2. The second method of Radix sorting is started with the least significant digit and ending with the most significant digit. Here, each number is taken in the order in which it appears in the file and is placed into a one of the ten queue, depending upon the value of digit currently being processed. Then, restore each queue to the original file starting with the queue of numbers with a ‘0’ digit and ending with queue of number with a ‘9’ digit. The process is repeated for each digit starting with the least significant digit and ending with the MSD and at last the file is sorted. This method is called Radix Sort.

For e.g.: -

Original File: -   25   57   48   37   12   92   86   33

Queue based on Least-Significant Digit : -

Fr.	Re.	
queue [0]		
queue [1]		
queue [2]	12	92
queue [3]	33	
queue [4]		
queue [5]	25	
queue [6]	86	
queue [7]	57	37
queue [8]	48	
queue [9]		

Queue based on Most-Significant Digit : -

queue [0]	
queue [1]	12

```
queue [2]          25
queue [3]          33      37
queue [4]          48
queue [5]          57
queue [6]
queue [7]
queue [8]          86
queue [9]          92
```

Thus, sorted file is : -   12   25   33   37   48   57   86   92

Q 59) Write an algorithm of shell short and explain with example.

Shell short is also known as Diminishing increment sort and is simply the improvement on simple insertion sort. In this method, several separate sub files of original file are sorted. These sub files contain every ith element of the original file. The value of i is called an increment. For e.g. if i=5, the sub files consists of x[0], x[5], x[10] is first sorted. So five sub files, each containing 1/5th of the elements of the original file are sorted. Increments are 5,3,1 on 1st ,2nd and 3rd iteration.

The algorithm can be written as:

```
Void shell sort(int x[], int n. int incrmnts[], int numic )
{
        int incr,j,k,span,y;
        for(incr=0;incr<numinc;incr++)
        {
                span=incrmnts[incr];
                for(j=span;j<n;j++)
                {
                        y=x[j];
                        for(k=j-1;k>=0 && y<x[k];k-=span)
                                x[k+span] =x[k];
                        x[k+span]=y;
                }
        }
}
```

In this method different increment i is chosen, i sub files are divided so that the jth element of the kth sub file is x[(j-1)*k+j-1]. After the first i sub files are sorted a new smaller value of increment is chosen and the file is again positioned into a new set of sub files. Each of these large sub files are sorted and process is repeated again with an even smaller value of i. The latest value in the sequence must be 1 so that the sub files consisting of the entire file be sorted.

For e.g.
Original file is

```
17     81     19     84     12     28     24     92     33
x[0]   x[1]   x[2]   x[3]   x[4]   x[5]   x[6]   x[7]   x[8]
```

let the sequence of the increment be 5,3,1 . then

```
1st iteration (increment =5) = (x[0],x[5])        (17   16)
                               (x[1],x[7])        (81   24)
```

```
(x[3],x[8])        (19   92)
(x[]4),..          (12)
sorting the sub files.
(17    26)         x[0]=17             x[5]=26
(24    81)    x[1]=24       x[6]=81
(15    92)    x[2]=15       x[7]=92
(34,   84)    x[3]=34       x[8]=33
(12)          x[4]=12
```

2nd iteration method(increment =3)

```
(x[0]  x[3]   x[6])        (17    34    81)
(x[1]  x[4]   x[7])        (24    12    92)
(x[2]  x[5]   x[8])        (19    26    84)

sorting the sub files.
(17    34    81)    x[0]=17      x[3]=34
x[6]=81
(12    24    92)    x[1]=12      x[4]=25
x[7]=92
(19    26    84)    x[2]=19      x[5]=26
x[8]=84
```

3rd iteration method(increment =1) (x[0] x[1] x[2] x[3] x[4] x[5] x[6] x[6] x[7] x[8])

```
(17 24  19  34  12  26  81  92  84)
sorting  (12  17  19  24  26  34  81  84  92)
```

in this way the shell sort sorts the original file.

59.ii. Write an algorithm of Shell Sort and explain with example.

Shell Sort is an improvement on simple insertion sort. This method sorts separate sub files of the original file. These sub files contain every kth elements of the original file. The value of k is called an increment. If the differential increment k is chosen, the k sub files are divided so that the ith element of the jth subfields x[(i-1)*k+j-1]. If k is 3, there will be three sub files and each contains one third of the original files sorted in these manner (reading across).

```
Sub file 1  ->x[0]    x[3]    x[6]    ……
Sub file 2 -> x[1]    x[4]    x[7]    ……
Sub file 3 -> x[2]    x[5]    x[8]    ……
```

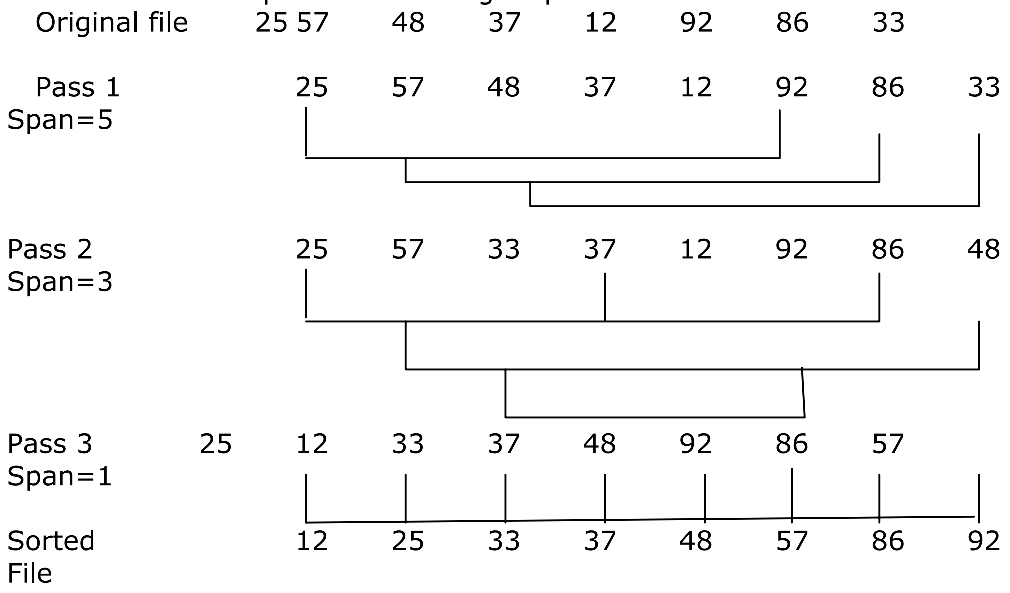
After the first k sub files are sorted (usually by simple insertion), a new smaller value of k is chosen and the file is sorted and the process is repeated yet again with an even smaller value of k. Eventually, the value of k is set to 1 so that the sub file consisting of the entire file is sorted . thus, a Shell Sort is also known as **diminishing increment sort**. A decreasing sequence of increments is fixed at the start of the entire process. The last value of this sequence must be 1.

The algorithm of the Shell Sort is as follows :

```
Void shell sort( int x[], int n, int incrmnts[], int numic)
{
        int incr, j, k, y;
        for(incr=0; incr<numic; incr++)
                {
                /*span is the size of the increment*/
                span=incmnts[incr];
                for(j=span; j<n; j++)
                        {
```

```
                                        /*insert element x[j] into its proper */
                                        /*position within its sub file*/
                                        y=x[j];
                                        for(k= j-span; k>=0 && y<x[k]; k-=span)
                                              x[k+span]=x[k];
                                        x[k+span]=y;
                                        }/*end for*/
                                  }/end for*/
                     }/*end shell sort*/
```

In addition to the standard parameters, x and n i.e. original file and its size it requires an array *incrmnts*, containing the diminishing increments of the sort and *numic,* the number elements in the array *incmnts.*
Let us consider incmnts[]={5, 3, 1}, numinc=3and original file with the elements as below. Then, the shell sort takes place in following steps.



Original file 25 57 48 37 12 92 86 33
Pass 1 Span=5 25 57 48 37 12 92 86 33
Pass 2 Span=3 25 57 33 37 12 92 86 48
Pass 3 Span=1 25 12 33 37 48 92 86 57
Sorted File 12 25 33 37 48 57 86 92

On the last iteration, where the span equals 1, the sort reduces to a simple insertion.

One thing that should be noted in Shell sort is that the elements of the *incrmnts* should be relatively prime for successful iteration and the entire file is indeed almost sorted when the span equals 1 on the last iteration.

60.i. What is heap? Write the conditions that should be satisfied to become a heap. Write the steps involved in a heap sort.

Heap is implicit data structure for the abstract data type, Priority Queue , and allow for fast insertions of new elements and fast deletions of the minimum element. In this respect, a heap is not an abstract data type, but an implementation of the priority queue.

We choose a representation of a priority queue as a complete binary tree in which each node contains an element and its associated key or priority. In addition, we assume that these keys satisfy the **heap condition**, that:

At each node, the associated key is larger than the keys associated with either child of that node.

This can be summaries as:

The structure of a heap is that of a complete binary tree in which satisfies the **heap ordering principle**. The conditions following this principle is:

1) The value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root
2) The value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

The steps involved in a heap sort are:
1.Heap sort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array.
2.After removing the largest item, it reconstructs the heap.
3.It then removes the largest remaining item and places it in the next open position from the end of the sorted array.
4.This is repeated until there are no items left in the heap and the sorted array is full.

60.ii.What is heap? Write the conditions that should be satisfied to become a heap. Write the steps involved in a heap sort.

Sorting algorithm, that works by first organizing the data to be sorted into a special type of binary tree called a *heap*. The heap itself has, by definition, the largest value at the top of the tree, so the heap sort algorithm must also reverse the order. It does this with the following steps:

1. Remove the topmost item (the largest) and replace it with the rightmost leaf. The topmost item is stored in an array.

2. Re-establish the heap.

3. Repeat steps 1 and 2 until there are no more items left in the heap.

The sorted elements are now stored in an array.

A heap sort is especially efficient for data that is already stored in a binary tree. In most cases, however, the *quick sort* algorithm is more efficient

Heap is implicit data structure for the abstract data type, Priority Queue , and allow for fast insertions of new elements and fast deletions of the minimum element. In this respect, a heap is not an abstract data type , but an implementation of the priority queue.

We choose a representation of a priority queue as a complete binary tree in which each node contains an element and its associated key or priority. In addition, we assume that these keys satisfy the heap condition, that:

At each node, the associated key is larger than the keys associated with either child of that node.

This can be summaries as:

The structure of a heap is that of a complete binary tree in which satisfies the heap ordering principle. The conditions following this principle is:

i. The value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root

ii. The value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.

The steps involved in a heap sort are:

- Heap sort begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array.
- After removing the largest item, it reconstructs the heap.
- It then removes the largest remaining item and places it in the next open position from the end of the sorted array.
- This is repeated until there are no items left in the heap and the sorted array is full.

61.i. Compare the different types of sorting for different types of data. Give suitable reason in support for your answer.

Sorting is the most common ingredients of programming systems. Sorting means putting unordered data into order. The typical situation is where we have items in an array and would like to sort them by some key so that we can e.g. print them out nicely or do fast binary searches on them. There are different sorting techniques, they are as follows:

BUBBLE SORT

This is one of the simplest sorting techniques. The idea is we go through the array, comparing adjacent elements. If they are out of order, we swap them. We keep repeating this process over and over until we can go through the array without doing any swaps. At this point, no elements are out of order and the array is sorted. There are n-1 passes and n-1 comparisons on each pass. Thus the total no. of comparisons is $(n-1)*(n-1)=n^2-2n+1$, which is $O(n^2)$. Of course, the number of interchanges depends on the original order of the file. However, the no. of interchanges cannot be greater then the no. of comparisons. The only redeeming features of the bubble sort are that it requires little additional space in comparison to other sorting techniques.

INSERTION SORT

An insertion sort is one that sorts a set of values by insertion values into an existing sorted file. Suppose an array a with n elements a[1],a[2]………,a[n] is in memory. The insertion sort algorithm scans a from a[1] to a[n], inserting each element a[k] into its proper position in the previously sorted sub array a[1],a[2],…….a[k-1]. Insertion sort has one major disadvantage with respect to other sorting. Even after more items have been sorted properly into the first part of the list, the insertion of a later item may require that many of them be moved. All the moves made by insertion sort are moves of only one position at a time. Thus to move an item 20 positions up the list requires 20 separate moves. If the items are small ,perhaps a key alone, or if the items are in linked storage, then that many moves may not require excessive time. But if the items are very large, structures containing hundreds of components like personnel files or student transcripts, and the structures must be kept in contiguous storage, then it would be far more efficient if, when it is necessary to move an item, it could be moved immediately to its final position. Now, this goal can be accomplished by sorting method called selection sort.

SELECTION SORT

The objective of this sort is to minimize data movement. The primary advantage of selection sort regards data movement. If an item is in its correct final position, then it will never be moved. Every time any pair of items is swapped ,then at least one of them moves into its final position, and therefore at most n-1 swaps are done altogether in sorting a list of n items. This is very best that we can expect from any method that relies entirely on swaps to move its items.

Let us pause for a moment to compare the counts for selection sort with those for insertion sort. The results are

	Selection	Insertion (average)
Assignments of items	3.0n+O(1)	$0.25n^2$+O(n)
Comparisons of keys	0.5n2+O(n)	$0.25n^2$+0(n)

Let us make relative comparisons between these two. When n becomes large, $0.25n^2$ becomes much larger than 3n, and if moving items is a slow process, then insertion sort will take far longer than will selection sort. But the amount of time taken for comparisons is, on average, only about half as much for insertion sort as for selection sort. Under other conditions, then, insertion sort will be better.

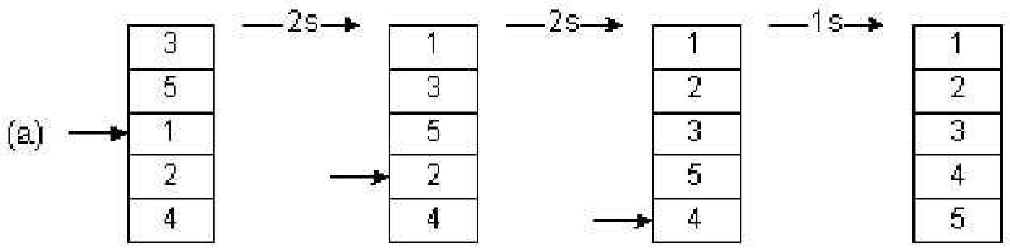
SHELL SORT

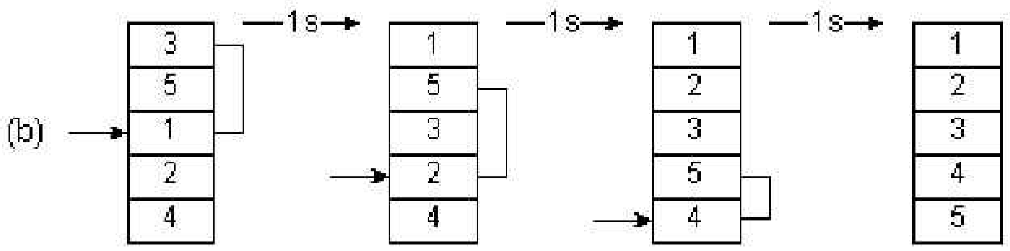
As we have seen that insertion sort and selection sort behave in opposite ways.
Selection sort moves the items very efficiently but does many redundant comparisons. In its best case, insertion sort does the minimum no. of comparisons, but is inefficient in moving items only one place at a time. Now, the problems with both of these are avoided by another method called shell sort. This method sorts separate sub files of the original file. These sub files contain every kth element of the original file. The value of k is called an increment. For example, if k is 5,the sub file consisting of x[0],x[5],x[10], ………is first stored. Five sub files; each containing one fifth of the elements of the original file are sorted in this manner.

Shell sort improves on the efficiency of insertion sort by quickly shifting values to their destination. This can be explained as:

In Figure (a) we have an example of sorting by insertion. First we extract 1, shift 3 and 5 down one slot, and then insert the 1, for a count of 2 shifts. In the next frame, two shifts are required before we can insert the 2. The process continues until the last frame, where a total of 2 + 2 + 1 =5 shifts have been made.

In Figure (b) an example of shell sort is illustrated. We begin by doing an insertion sort using a *spacing* of two. In the first frame we examine numbers 3-1. Extracting 1, we shift 3 down one slot for a shift count of 1. Next we examine numbers 5-2. We extract 2, shift 5 down, and then insert 2. After sorting with a spacing of two, a final pass is made with a spacing of one. This is simply the traditional insertion sort. The total shift count using shell sort is 1+1+1 = 3. By using an initial
spacing larger than one, we were able to quickly shift values to their proper destination.





QUICK SORT
Although the shell sort algorithm is significantly better than insertion sort, there is still room for improvement. One of the most popular sorting algorithms is quick sort. Quick sort executes in O(*n* log n) on average, and O(*n* 2) in the worst-case. However, with proper precautions, worst-case behavior is very unlikely.
It posses a very good average case behaviors among all the sorting techniques. It works by partitioning the array to be sorted. And each partition is in turn sorted recursively. In partition, one of the array elements is chosen as a key value. This key value can be the first element of an array. That is, if a is an array then key=a[0]. And rest of the array elements are grouped into two partitions such that

- one partition contains elements smaller than the key value.
- Another partition contains elements larger than the key value.

Now let us comp the sorting algorithms covered: insertion sort, shell sort, and quick sort. There are several factors that influence the choice of a sorting algorithm:
• Stable sort: A stable sort is a sort that will leave identical keys in the same relative position in the sorted output. Insertion sort is the only algorithm covered that is stable.
Space: An in-place sort does not require any extra space to accomplish its task. Both insertion sort and shell sort are in- place sorts. Quick sort requires stack space for recursion, and therefore is not an in-place sort. Tinkering with the algorithm considerably reduced the amount of time required.
Time: Table below shows the relative timings for each method.

Methods	Statements	Average time	Worst case time
Insertion sort 9	9	O(n2)	O(n2)
Shell sort	17	O(n7/6)	O(n4/3)
Quick sort	21	O(n log n)	O(n2)

61.ii. Compare the different types of sorting for different types of data. Give suitable reason in support for your answer.

Sorting means putting unordered data into order. The typical situation is where we have items in an array and would like to sort them by some key so that we can e.g. print them out nicely or do fast binary searches on them. There are different sorting techniques.

BUBBLE SORT

This is one of the simplest sorting techniques. The idea is we go through the array, comparing adjacent elements. If they are out of order, we swap them. We keep repeating this process over and over until we can go through the array without doing any swaps. At this point, no elements are out of order and the array is sorted. There are n-1 passes and n-1 comparisons on each pass. Thus the total no. of comparisons is $(n-1)*(n-1)=n^2-2n+1$, which is $O(n^2)$. Of course, the number of interchanges depends on the original order of the file. However, the no. of interchanges cannot be greater then the no. of comparisons. The only redeeming features of the bubble sort are that it requires little additional space in comparison to other sorting techniques.

INSERTION SORT

An insertion sort is one that sorts a set of values by insertion values into an existing sorted file. Suppose an array a with n elements a[1],a[2]………,a[n] is in memory. The insertion sort algorithm scans a from a[1] to a[n], inserting each element a[k] into its proper position in the previously sorted sub array a[1],a[2],…….a[k-1]. Insertion sort has one major disadvantage with respect to other sorting. Even after more items have been sorted properly into the first part of the list, the insertion of a later item may require that many of them be moved. All the moves made by insertion sort are moves of only one position at a time. Thus to move an item 20 positions up the list requires 20 separate moves. If the items are small ,perhaps a key alone, or if the items are in linked storage, then that many moves may not require excessive time. But if the items are very large, structures containing hundreds of components like personnel files or student transcripts, and the structures must be kept in contiguous storage, then it would be far more efficient if, when it is necessary to move an item, it could be moved immediately to its final position. Now, this goal can be accomplished by sorting method called selection sort.

SELECTION SORT

The objective of this sort is to minimize data movement. The primary advantage of selection sort regards data movement. If an item is in its correct final position, then it will never be moved. Every time any pair of items is swapped ,then at least one of them moves into its final position, and therefore at most n-1 swaps are done altogether in sorting a list of n items. This is very best that we can expect from any method that relies entirely on swaps to move its items.

Let us pause for a moment to compare the counts for selection sort with those for insertion sort. The results are

	Selection	Insertion (average)
Assignments of items	3.0n+O(1)	$0.25n^2$+O(n)
Comparisons of keys	0.5n2+O(n)	$0.25n^2$+0(n)

Let us make relative comparisons between these two. When n becomes large, $0.25n^2$ becomes much larger than 3n, and if moving items is a slow process, then insertion sort will take far longer than will selection sort. But the amount of time taken for comparisons is, on average, only about half as much for insertion sort as for selection sort. Under other conditions, then, insertion sort will be better.

SHELL SORT

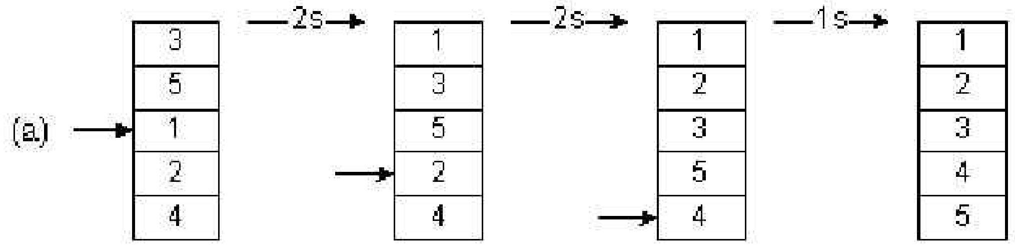
As we have seen that insertion sort and selection sort behave in opposite ways.
Selection sort moves the items very efficiently but does many redundant comparisons. In its best case, insertion sort does the minimum no. of comparisons, but is inefficient in moving

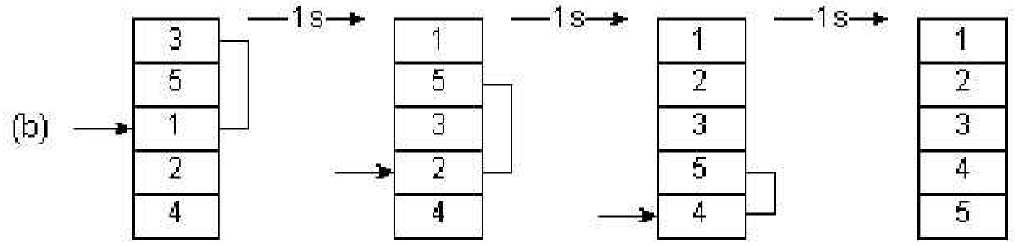
items only one place at a time. Now, the problems with both of these are avoided by another method called shell sort. This method sorts separate sub files of the original file. These sub files contain every kth element of the original file. The value of k is called an increment. For example, if k is 5,the sub file consisting of x[0],x[5],x[10], ………is first stored. Five sub files; each containing one fifth of the elements of the original file are sorted in this manner.

Shell sort improves on the efficiency of insertion sort by quickly shifting values to their destination. This can be explained as:

In Figure (a) we have an example of sorting by insertion. First we extract 1, shift 3 and 5 down one slot, and then insert the 1, for a count of 2 shifts. In the next frame, two shifts are required before we can insert the 2. The process continues until the last frame, where a total of 2 + 2 + 1 =5 shifts have been made.

In Figure (b) an example of shell sort is illustrated. We begin by doing an insertion sort using a *spacing* of two. In the first frame we examine numbers 3-1. Extracting 1, we shift 3 down one slot for a shift count of 1. Next we examine numbers 5-2. We extract 2, shift 5 down, and then insert 2. After sorting with a spacing of two, a final pass is made with a spacing of one. This is simply the traditional insertion sort. The total shift count using shell sort is 1+1+1 = 3. By using an initial
spacing larger than one, we were able to quickly shift values to their proper destination.





QUICK SORT
Although the shell sort algorithm is significantly better than insertion sort, there is still room for improvement. One of the most popular sorting algorithms is quick sort. Quick sort executes in O(*n* log n) on average, and O(*n* 2) in the worst-case. However, with proper precautions, worst-case behavior is very unlikely.
It posses a very good average case behaviors among all the sorting techniques. It works by partitioning the array to be sorted. And each partition is in turn sorted recursively. In partition, one of the array elements is chosen as a key value. This key value can be the first element of an array. That is, if a is an array then key=a[0]. And rest of the array elements are grouped into two partitions such that

- ⇨ one partition contains elements smaller than the key value.

⇨ Another partition contains elements larger than the key value.

Now let us comp the sorting algorithms covered: insertion sort, shell sort, and quick sort. There are several factors that influence the choice of a sorting algorithm:

• **Stable sor**t: A stable sort is a sort that will leave identical keys in the same relative position in the sorted output. Insertion sort is the only algorithm covered that is stable.

Space: An in-place sort does not require any extra space to accomplish its task. Both insertion sort and shell sort are in- place sorts. Quick sort requires stack space for recursion, and therefore is not an in-place sort. Tinkering with the algorithm considerably reduced the amount of time required.

Time: Table below shows the relative timings for each method.

Methods	Statements	Average time	Worst case time
insertion sort 9	9	O(n2)	O(n2)
shell sort	17	O(n7/6)	O(n4/3)
quick sort	21	O(n log n)	O(n2)

62.i. What is search? Discuss the binary search algorithm with suitable example.

Search is a technique used to find the location where the desired element is available. Computer systems are often used to store large amounts of data from which individual records must be retrieved according to some search criterion. Thus the efficient storage of data to facilitate fast searching is an important issue. Generally, there are two types of searching:

⇨ Linear or sequential searching

⇨ Binary searching

In linear search, we access each element of an array one by one sequentially and see whether it is desired element or not. A search will be unsuccessful if all the elements are accessed and the desired element is not found. In worst case, the number of average case we may have to scan half of the size of the array(n/2).

However, if we place our items in an array and sort them in either ascending or descending order on the key first, then we can obtain much better performance with an algorithm called ***binary*** search. **I**n binary search, we first compare the key with the item in the middle position of the array. If there's a match, we can return immediately. If the key is less than the middle key, then the item sought must lie in the lower half of the array; if it's greater then the item sought must lie in the upper half of the array. So we repeat the procedure on the lower (or upper) half of the array.

ALGORITHM

```
1) Initialize segment variables i.e. set low=0,hi=item-1 and mid=(hi+low)/2

2) Repeat steps 3 and 4 while low<=hi and [mid]!=item

3) If item<a[mid],then
```

```
     Set hi=mid-1
      Else set low=mid+1
4) Set mid=int ((hi+low)/2)
5) If a[mid]=item then
      Set location=mid
       Else location=NULL
     3)  Exit
```

Analysis

Suppose we have an array of 7 elements

9	12	24	30	36	45	70

0 1 2 3 4 5 6

The steps to search 45 using binary search in array a[7] are:

STEP 1: The given array is in ascending order; item to be searched for is 45.

Low=0, hi=6

mid=int(low+hi)/2= int(0+6/2)=int(3)=3

9	12	24	30	36	45	70

0 low 1 2 3 4 5
6 hi

STEP 2: a[mid] i.e. a[3] is 30

30<45 then low=mid+1=3+1=4

STEP 3: mid=int ((low+hi)/2) = int (4+6)/2= int 5=5

low	mid	hi
36	45	70

4 5 6

a[mid] i.e. a[5] is 45

45=45

Search successful!!! At location number 5 (element number 6).

62.ii.What is search? Discuss the binary search algorithm with suitable example.
Search is the process of finding particular elements of an array or list.
Binary Search: It is a method for searching through data where the algorithm decides which half of the data the value being searched for resides in, discards the other half, and repeats using the remaining half as the data set being searched.
Binary Search Algorithm can only be used if the table is stored as an array. For the binary search algorithm to work the List/Array must be sorted.
Each step of the algorithm divides the block of items being searched in half. We can divide a set of ***n*** items in half at most $\log_2 n$ times.
Thus the running time of a binary search is proportional to **log *n*** and we say this is a ***O(log n)*** algorithm.

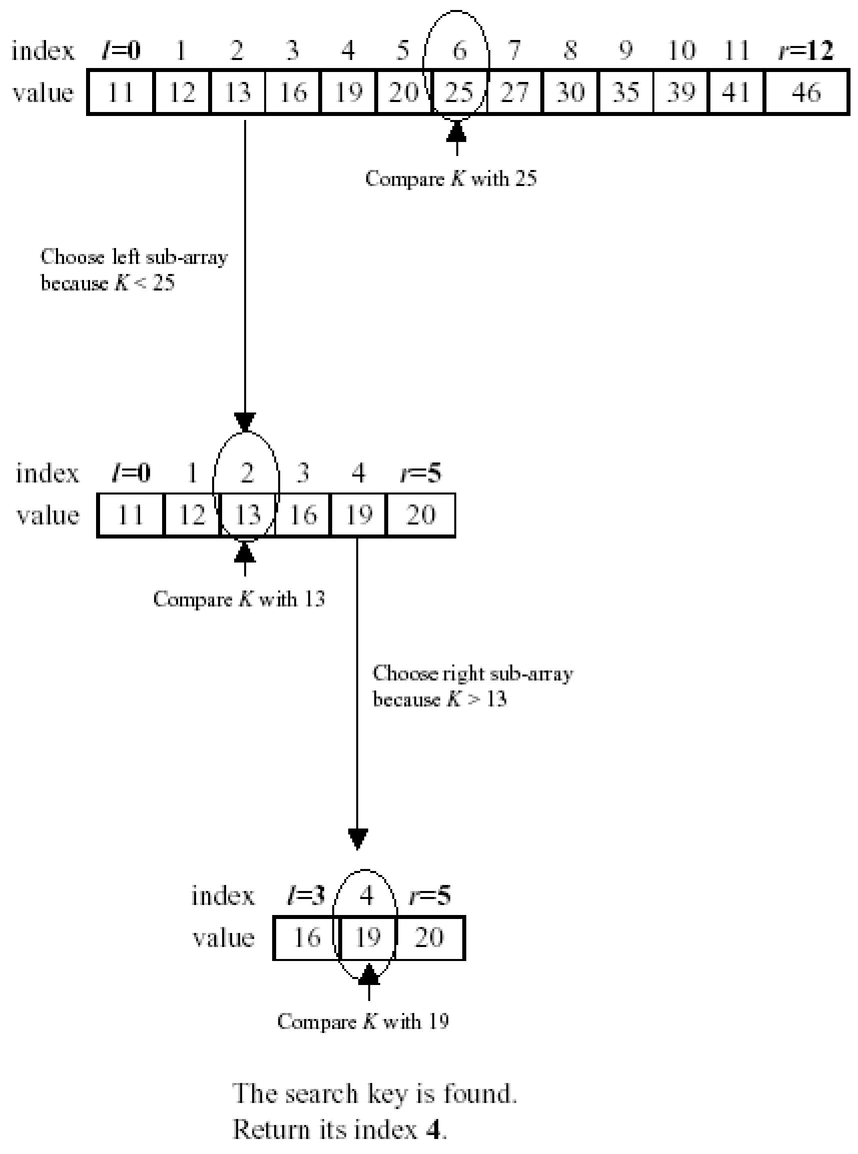
```
//The algorithm implements iterative binary search
//Input: an array A[l..r] sorted in ascending order, defined by its
// left and right indices l and r
// a search key K
//Output: an index of the array's element that is equal to K
// or −1 if there is no such element
while l □□r do
mid □□□(l+r)/2□
if K == A[mid] return mid
else
if K < A[mid] r □□mid-1
else l □□mid+1
return −1 //Search key not in array
```

Binary Search Algorithm

Explanation:

1. A midpoint pointer is initialized to point to the middle of the list. i.e the array is divided into two sub-arrays about half as large.
2. While the start pointer is LESS THAN OR EQUAL TO the end pointer AND the Item is NOT found.
2(a). if key k is LESS THAN value pointed to by the mid pointer
"right" pointer is set to point to mid index – 1 (i.e. ignore top half of current list)
2(b). Else item is GREATER THAN value pointed to by mid pointer
"left" pointer is set to point to mid index + 1 (I.e. ignore bottom half of current list)
2(c). Mid pointer is set to point to mid index of current new working list, loop back and do check again.
3. If mid pointer is EQUAL to our searched "Key", the algorithm stops.
4. Else returns –1 to indicate item not found.

Example.
Consider an array of 13 elements and search key K = 19. The operation of binary search is illustrated in the figure below.



63. Compare and contract the efficiency of three searching algorithms. (Sequential, Binary and Search Tree).

Sequential Search: The algorithm begins at the first element in the array and looks at each element in turn until the search argument is found.

Analysis of sequential search efficiency:

The ***worst-case efficiency*** of this search algorithm is its efficiency for the worst-case input of size *n*. In the worst case, ***sequential Search*** performs *n* comparisons (if key *K* is the last element in the array).So consumes more time.
The ***best-case efficiency*** of an algorithm is its efficiency for the best-case input of size *n*. In the best case, one comparison is done (if key *K* is the first element in the array).So requires less time.
The ***average-case efficiency*** of an algorithm is its efficiency for a "typical" or "random" input of size ***n***. In the average case, *n*/2 comparisons are done.
The time complexity of sequential search is ***O*(*n*)** in the worst case.

Sequential search is easier to implement than binary search, and does not require the list to be sorted.

Binary Search:
Binary search that requires ***O*(log2*n*)** time.
Binary search is much faster than sequential search, but it works only for **sorted** arrays.

Binary search is much better than sequential search. For example, □log21,000,000□□= 19, so
a sequential search of one million sorted elements can require one million comparisons while a binary search will require at most 20 comparisons. For large arrays, the binary search has enormous advantage over a sequential search.

64. Define the terms:

a. Record, key, Table:

A table or a file is a group of elements, each of which is called record. Associated with each record is a key, which is used to differentiate among different records**.**

b. Internal key, External key:

The association between a record and its key may be simple or complex. In simplest form, the key is contained within the record at a specific offset from the start of the record. Such a key is called an internal key or an embedded key.

In other cases there is a separate table of keys that includes pointers to the records. Such keys are called external keys.

c. Primary key, Secondary Key:

For every file there is at least one set of keys (possibly more) that are unique (that is, no two records have the same key value). Such a key is called a Primary key. For example, if the file is stored as an array, the index within the array of an element is a unique external key for that element.
If the state used is in such a way that the key for a particular search is not be unique. Since there may be two records with the same state in the file. Such a key is called Secondary key.
Some of the algorithms we present assume unique keys; others allow for duplicate keys. When adopting an algorithm for a particular application the programmer should know whether the keys are unique and make sure that the algorithm selected is appropriate.

65. I. Compare internal searching and external searching.

A file is also known as a Table and it is a group of elements each of which is called a record. Each record id differentiated with other different records by an association known as a key. The key may or may not be contained within a record. The key within a record is called internal key or embedded key. A separate table of keys that include pointers to the records and such are called external key.

For the organization of the file such keys are used. So while the file is needed to be searched different techniques are to be used. Such as, the file with internal key, while searching in such file the content of entire table maintained within the main memory is searched and such searching technique is called internal searching. But it is not the case of external searching. The file which has its table of keys is kept in the auxiliary storage, so while searching in such file the table is to be searched by going to the auxiliary storage and such searching techniques is called external searching.

So external searching is quite different from internal searching with a view of managing the table of keys. Since the table of key is in the auxiliary storage external searching is slower and inefficient because it requires external storage and the searching time is greater as compared to internal searching.

66.i. Write down sequential searching algorithm and also its efficiency?

The sequential searching algorithm is as follows.

```
for (i=0;i<n;i++)
        if (key == k(i))
                return (i);
        return (-1);
```

This algorithm examines each key in turn; upon finding one that matches the search argument, its index is returned. If no match is found than -1 is returned.
If we have number of comparisons to perform through a table of constant size n .The number of comparisons depends upon where the record with the argument key appears in the table .If the record is the first one in the table than only one comparison will do but if the record is the last one than the no of comparisons needed is n. If it is equally likely for the argument to be found at any table position, a successful search will take (on the average) (n+1)/2 comparisons and an unsuccessful search will take n comparisons. In any case the number of comparisons being 0(n).
Let p(i) be the probability that record i is retrieved. (P (i) is a number between 0 and 1 such that if m retrievals are made form the file , m*p(i) of them will be from r(i). Let us assume that p(0)+p(1)+………..+p(n-1) = 1, so that there is no possibility that an argument key is missing from the table. Then the average number of comparisons in searching for a record is

P(0) + 2*p(1)+3*p(2)+…..+n*p(n-1

Clearly this number is minimized if
P(0)>=p(1)>=p(2)>=…..>=p(n-1)

Thus if a large stable file, reordering the file in order of decreasing probability of retrieval achieves a greater degree of efficiency each time that the file is searched.

66.ii. Write down the sequential searching algorithm and also its efficiency.

Sequential searching is the simplest type of searching method. This search is applicable to a table organized either as an array or as a linked list. To write an algorithm we have to generalize some terms. r represents a record k represents a key so that k(I) is the key of r(i). i is the index.

The algorithm:

```
For(i=0;i<n;i++)

    If(key==k(i))

      Return (i);

Return –1;
```

If the required key is matched with the record key the index of that record is returned form the function otherwise –1 is returned. The algorithm examines each key in turn.

Efficiency of sequential search:

The efficiency of the sequential search depends upon the location of the record that is searched for i.e. nearer the record fastest is the search. The number of comparisons depends upon the location of the record with the argument key. If the record is in the 0th index then only one comparison is done and if the record is in the last position then n comparison is needed.

67.i. write down the two versions of Binary Search.(i.e. Recursive and Iterative).

The binary search refers to the searching techniques that divide the records into two parts while searching. In this search the argument compared with the key of the middle element of the table. If they are equal, the search ends successfully otherwise either the upper or lower half of the table must be searched in the similar manner until the required element is found.

The recursive version of the binary search is as follows:

```
int bsearch(int key,int a[]){
 If(I>j)
   Return –1;
mid=(I+j)/2;
if(key==a[mid])
      return mid;
if(key<a[mid])
  bsearch(key,a,I,mid-1)
else
bsearch(key,a,mid+1,j);
}
```

The iterative version of the binary search is as follows:

```
Low=0;
Hi=n-1;
While(low<=hi){
  mid=(low+hi)/2;

if(key==k[mid])
     return(mid);
if(key<k(mid))
    hi=mid-1;
else
   low=mid+1;
}
return –1;
```

Each comparison in the binary search reduces the number of possible candidates by a factor of 2. Therefore the maximum number of comparisons is approximately $\log_2 n$. Hence binary search is a better search for a large number of data.

67.ii. Write down the two versions of Binary Search.(i.e. Recursive and Iterative).

The binary search refers to the searching techniques that divide the records into two parts while searching. In this search the argument compared with the key of the middle element of the table. If they are equal, the search ends successfully otherwise either the upper or lower half of the table must be searched in the similar manner until the required element is found.

The recursive version of the binary search is as follows:

```
Int bsearch(int key,int a[]){
 If(I>j)
   Return –1;
mid=(I+j)/2;
if(key==a[mid])
      return mid;
if(key<a[mid])
  bsearch(key,a,I,mid-1)
else
bsearch(key,a,mid+1,j);
}
```

The iterative version of the binary search is as follows:

```
Low=0;
Hi=n-1;
While(low<=hi){
  mid=(low+hi)/2;

if(key==k[mid])
     return(mid);
if(key<k(mid))
    hi=mid-1;
else
   low=mid+1;
}
return –1;
```

Each comparison in the binary search reduces the number of possible candidates by a factor of 2. Therefore the maximum number of comparisons is approximately $\log_2 n$.

68.i. Explain the searching mechanism in a multi-way search tree?

A multi-way tree of order n is a general tree in which each node has n or fewer sub trees and one fewer key than it has sub trees .Now the searching mechanism is described below:
The algorithm to search a multiway search tree, regardless of whether it is
Topdown, balanced, or neither, is straightforward. Each node contains a single integer field ,a variable number of key fields.
If node (p) is a node, the integer field numtrees(p) equals the number of sub trees of node (p) .numtrees(p) is always less than or equals to the order of the tree, n. The pointer fields son (p,0) through son (p , numtrees(p)-1) point to the sub trees of node(p). the key field's k(p,0) through k(p , numtrees(p)-2) are the keys contained in node(P) in ascending order. The sub tree to which son(p,I) points contains all keys in the tree between k(p,I-1) and k(p,I).

We also assume a function node search(p, key) that returns the smallest integer j such that key <=k(p,j), or numtrees (p)-1 if key is greater than the keys in node(p) .
The following recursive algorithm is for the function search (tree) that returns a pointer to the node containing key and sets the global variable position to the position of key in that node :

```
       P=tree;
If(p==tree){
  Positin= -1;
  return(-1);
}
I=nodesearch(p, key);
If( I< numtrees ( p) – 1 && key == k(p,I)){
   Position  = I;
return (p);
}
return(search (son(p, I) ));
```

The function node search is responsible for locating the smallest key in a node greater than or equal to the search argument. The simplest technique or doing this is a sequential search through the ordered set of keys in the nodes. If all keys are of equal length , a binary search can also be used to locate the appropriate key. The decision whether to use a sequential or binary search depend upon the order of tree, which determine how many keys must be searched. Another possibility is to organize the keys within the node as a binary search tree.

68.ii. Explain the searching mechanism in a Multi-way search tree.

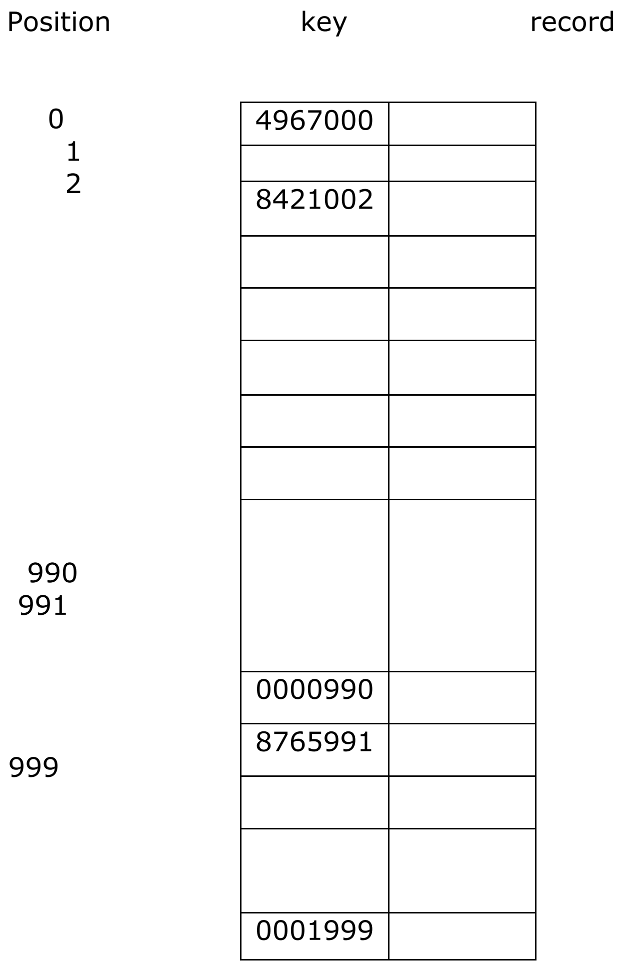
A multi-way search tree of order n is a general tree in which each node has n or fewer sub trees and contains one fewer key than it has sub trees. That is, if a node has four sub trees, it contains three keys .The searching mechanism for the multi-way search tree whether it is top-down, balanced, or neither ,is straightforward. Each node contains a single integer field, a variable number of pointer fields, and a variable number of key fields. If node (p) is a node, the integer field numtrees (p) equals the number of sub trees of node(p).numtrees (p) is always less than or equal to the order of the tree, n. The pointer fields son(p,0) through son(p, numtrees (p)-1) point to the sub trees of node(p) .The key fields k(p,0) through k(p ,numtrees (p)-2) are the keys contained in node(p) in ascending order. The sub tree to which son(p, i) points (for I between 1 and numtrees(p)-2 inclusive) contains all keys in the tree between k(p,i-1) and k(p,i).son(p,0) points to a sub tree containing only keys less than k(p,0) and son(p,numtrees(p)-1) points to a sub tree containing only keys greater than k(p ,numtrees(p)-2).
We also assume a function nodesearch(p, key) that returns the smallest integer j such that key<=k(p, j) , or numtrees (p)-1 if key is greater than all the keys in node(p).The following recursive algorithm is for a function search(tree) that returns a pointer to the node containing key(or –1[representing null] if there is no such node in the tree) and sets the global variable position to the position of key in that node:

```
          p=tree;
 if (p==null){
   position=-1;
return (-1);
}
i=nodesearch(p, key);
 if(i<numtrees (p)-1&& key ==k(p, i) {
```

```
          position =i;
         return (p);
        }
         return (search(son(p, i)));
```

69. Define Hashing? Why hashing is needed? Explain with suitable justification.

Hashing is the process of positioning or pointing the key of any number of digits by using the less number of elements. For example, suppose that the company has an inventory file of more than 100 items and the key to each record is a seven-digit part number. To use direct indexing using the entire seven-digit key, an array of 10 million elements would be required. This clearly wastes an unacceptably large amount of space because it is extremely unlikely that a company stocks more than a few thousand parts.
So, the hashing is used for the seven-digit number. If the company has fewer than 1000 parts and that there is only a single record for each part. Then an array of 1000 elements is sufficient to contain the entire file. The array is indexed by an integer between o and 999 inclusive. The last three digits of the part number are used as the index for the part's record in the array.



In the above figure the seven –digits numbers are represented by the 0-to 999-. The function that transforms a key into a table index is called a hash function. The seven –digits are not close to each other but while representing they are close. For example

0000990 and 8765991 are extremely different but the 990 and 991, which are close, represents them.

73. Discuss topological depth first Traversal with a suitable example (Assume a directed graph of eight vertices)

Depth-first traversal of a graph is roughly analogous to preorder traversal of an ordered tree. Depth-first traversal as its name indicates, traverse a single path of the graph as far as it can go i.e. until it visits a node with no successors or a node all of whose successor have already been visited. It then resume at the last node on the path just traversed that has an unvisited successor and begins traversing a new path emanating from that node. Spanning trees created by a depth-first traversal tends to be very deep. Depth-first traversal, like any other traversal method that creates a spanning forest, can be used to determine if an undirected graph is connected and to identify the connected component of an undirected graph. Depth-first traversal can also be used to determine if a graph is acyclic.

The depth-first traversal technique is best defined using an algorithm dftraverse(s) that visits all nodes reachable from s. This algorithm is presented shortly. We assume an algorithm visit (nd) that visits a node nd and a function visited (nd) that returns TRUE if nd has already been visited and FALSE otherwise. This is best implemented by a flag in each node. Visit sets the field to True. To execute the traversal, the field is first set false for all nodes. The traversal algorithm also assumes the function select with no parameter to select an arbitrary unvisited node. Select returns null if all nodes have been visited.

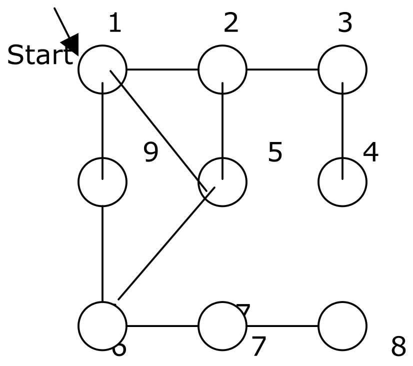
```
 for (every node nd)
     visited (nd)=FALSE;
s=a pointer to the starting node for the traversal
while (s=!null){
dftraverse(s);
s=select();
}
```

74.i. Write down the difference between depth first and breadth first traversal of a graph?

In many problems, we wish to investigate all the vertices in a graph in some systematic order, just as with binary trees, where we developed several systematic traversal methods. In tree traversal, we had a root vertex with which we generally started; in graphs, we often do not have any one vertex singled out as special, and therefore the traversal may start at an arbitrary vertex. Although there are many possible orders for visiting the vertices of the graph, two methods are of particular importance: Depth-first traversal and Breadth- first traversal.

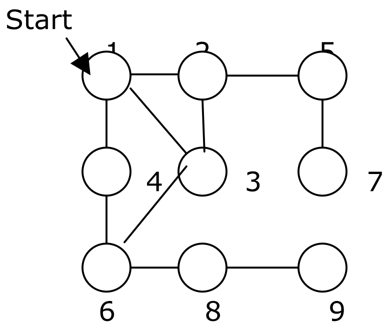
Depth-first traversal of a graph is roughly analogous to preorder traversal of an ordered tree. Depth-first traversal as its name indicates, traverse a single path of the graph as far as it can go i.e. until it visits a node with no successors or a node all of whose successor have already been visited. It then resume at the last node on the path just traversed that has an unvisited successor and begins traversing a new path emanating from that node. Spanning trees created by a depth-first traversal tends to be very deep. Depth-first traversal, like any other traversal method that creates a spanning forest, can be used to determine if an undirected graph is connected and to identify the connected component of an undirected graph. Depth-first traversal can also be used to determine if a graph is acyclic.

Suppose that the traversal has just visited a vertex v, and let w1,w2,….,wk be the vertices adjacent to v. Then we shall next visit w1 and keep w2,…,wk waiting. After visiting w1,we traverse all the vertices to which it is adjacent before returning to traverse w2,….,wk.



Depth-first traversal

Breadth-first traversal of a graph is roughly analogous to level-by-level traversal of an ordered tree. An alternative traversal method, breadth-first traversal (or breadth-first search), visits all the successors of a visited node before visiting any successors of any of those successors. This is the contradiction to depth-first traversal, which visits the successors of a visited node before visiting any of its 'brother'. Whereas depth-first traversal tends to create very long, narrow trees, breadth first traversal tends to create very wide, short trees.
If the traversal has just visited a vertex v, then it next visits all the vertices adjacent to v, putting the vertices adjacent to these in a waiting list to be traversed after all the vertices adjacent to v have been visited. Figure below shows breadth-first traversal.



Breadth-first traversal

In implementing depth-first traversal, each visited node is placed on a stack (either implicitly via recursion or explicitly), reflecting the fact that the last node visited is the first node whose successors will be visited. Breadth-first traversal is implemented using a queue, representing the fact that the first node visited is the first node whose successor are visited.

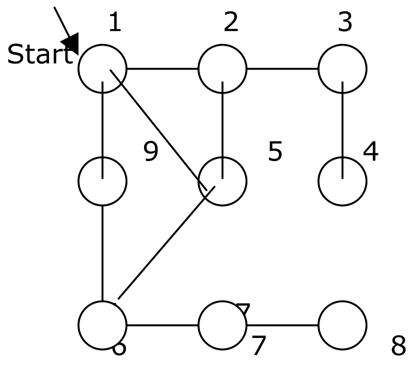
74: Write down the difference between depth first and breadth first traversal of a graph?

In many problems, we wish to investigate all the vertices in a graph in some systematic order, just as with binary trees, where we developed several systematic traversal methods.

In tree traversal, we had a root vertex with which we generally started; in graphs, we often do not have any one vertex singled out as special, and therefore the traversal may start at an arbitrary vertex. Although there are many possible orders for visiting the vertices of the graph, two methods are of particular importance: Depth-first traversal and Breadth- first traversal.

Depth-first traversal of a graph is roughly analogous to preorder traversal of an ordered tree. Depth-first traversal as its name indicates, traverse a single path of the graph as far as it can go i.e. until it visits a node with no successors or a node all of whose successor have already been visited. It then resume at the last node on the path just traversed that has an unvisited successor and begins traversing a new path emanating from that node. Spanning trees created by a depth-first traversal tends to be very deep. Depth-first traversal, like any other traversal method that creates a spanning forest, can be used to determine if an undirected graph is connected and to identify the connected component of an undirected graph. Depth-first traversal can also be used to determine if a graph is acyclic.

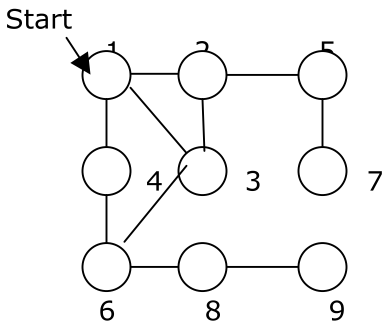
Suppose that the traversal has just visited a vertex v, and let w1, w2, wk be the vertices adjacent to v. Then we shall next visit w1 and keep w2, wk waiting. After visiting w1, we traverse all the vertices to which it is adjacent before returning to traverse w2, wk.



Depth-first traversal

Breadth-first traversal of a graph is roughly analogous to level-by-level traversal of an ordered tree. An alternative traversal method, breadth-first traversal (or breadth-first search), visits all the successors of a visited node before visiting any successors of any of those successors. This is the contradiction to depth-first traversal, which visits the successors of a visited node before visiting any of its 'brothers'. Whereas depth-first traversal tends to create very long, narrow trees, breadth first traversal tends to create very wide, short trees.

If the traversal has just visited a vertex v, then it next visits all the vertices adjacent to v, putting the vertices adjacent to these in a waiting list to be traversed after all the vertices adjacent to v have been visited. Figure below shows breadth-first traversal.



Breadth-first traversal

In implementing depth-first traversal, each visited node is placed on a stack (either implicitly via recursion or explicitly), reflecting the fact that the last node visited is the first node whose successors will be visited. Breadth-first traversal is implemented using a queue, representing the fact that the first node visited is the first node whose successor is visited.

75:-Define transitive closure. what is its application in graph? also define warshall's algorithm

Consider a directed graph G= (V, E), where V is the set of vertices and E is the set of edges. The transitive closure of G is a graph G+ = (V,E+) such that for all v,w in V there is an edge (v,w) in E+ if and only if there is a non-null path from v to w in G.
Finding the transitive closure of a directed graph is an important problem in many computational tasks. It is required, for instance, in the reach ability analysis of transition networks representing distributed and parallel systems and in the construction of parsing automata in compiler construction. Recently, efficient transitive closure computation has been recognized as a significant sub problem in evaluating recursive database queries, since almost all practical recursive queries are transitive.
Let us define the matrix path_k such that path_k[i][j] is true if and only if there is a path form node i to node j that does not pass through any nodes numbered higher than k. For any i and j such that path_k[i][j]=TRUE, path_{k+1}[i][j] must be true. The only situation in which path_{k+1} [i][j] can be true while path_k[i][j] equals false is if there is a path from i to j passing through node k+1, but there is no path form i to j passing through only nodes 1 through k. But this means that there must be a path from i to k+1 passing through k and a similar path from k+1 to j. Thus path_{k+1}[i][j] equals TRUE if and only if one of the following two conditions holds

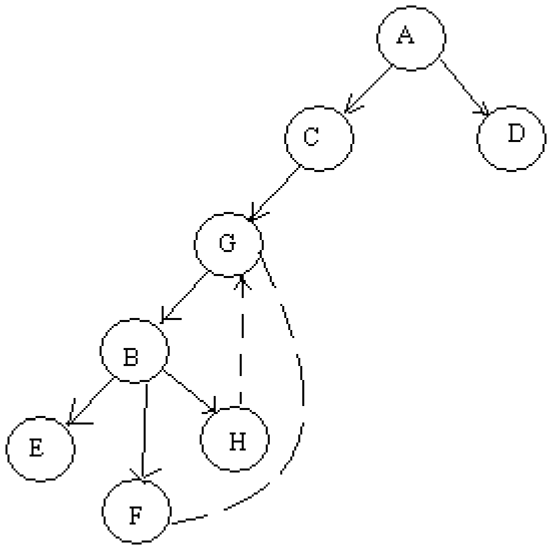
1:- path_k[i][j] ==true

2:-path_k[i][k+1]==true and path_k[k+1][j]==true

This method increases the efficiency of finding the transitive closure to $0(n^3)$. This method is called warshall's algorithm

76. Define the different types of Edges in Spanning tree.

A forest may be defined as an acyclic graph in which every node has one or no predecessors. A tree may be defined as a forest in which only a single node (called the root) has no predecessors. Any forest consists of a collection of trees. An ordered forest is one whose component trees are ordered.
Any spanning tree divides the edges (arcs) of a graph into four distinct groups:
Tree edges, forward edges, cross edges and back edges.
Tree Edges: Tree edges are arcs of the graph that are included in the spanning forest.
Forward Edges: Forward edges are arcs of the graph from a node to a spanning forest non-son descendant.
Cross Edges: Cross Edges are arcs from one node to another node that is not the first node's descendant or ancestor in the spanning forest.
Back Edges: Back edges are arcs from a node to a spanning forest ancestor.



ig a

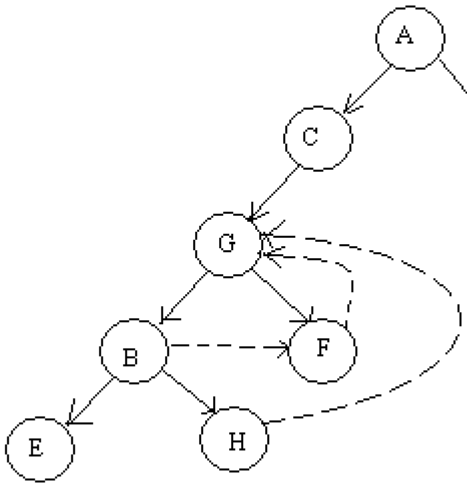


Fig b

Arc	Fig a	Fig b
<A,C>	Tree	Tree
<A,D>	Tree	Tree
<B,E>	Tree	Tree
<B,F>	Tree	Cross
<B,H>	Tree	Tree
<C,G>	Tree	Tree
<F,G>	Back	Back
<G,B>	Tree	Tree
<G,F>	Forward	Tree
<H,G>	Back	Back

77.i. Explain directed and undirected graph and also discuss about connected and not connected graphs.

Undirected Graph:

Undirected graph is represented by a diagram where vertices are represented by points or small circles, and edges by arcs joining the vertices of the associated path given by a mapping.

Figure below shows an undirected graph, thus the unordered pair (v_1,v_2) is associated with edge e_1; the pair (v_2,v_2) is associated with edge e6(a self loop).

fig

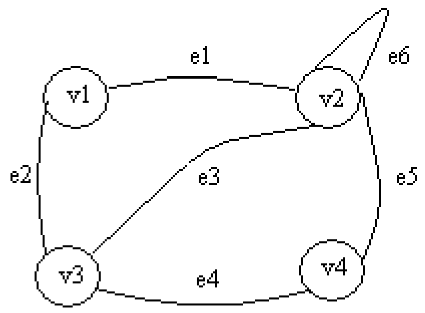


Fig: undirected graph

In an undirected graph, adjacency is both from and to for any edge.

Directed Graph:

A directed graph or digraph consists of:

(a) a non empty set V called the set of vertices,

(b) a set E called the set of edges, and

(c) a map Ø which assigns to every edge unique ordered pair of vertices.

In a directed graph, edges are represented by different arcs.

Figure below gives a directed graph. The order pair (v_2,v_3),(v_3,v_4),(v_1,v_3) is associated with the edges e_3,e_4,e_2, respectively.

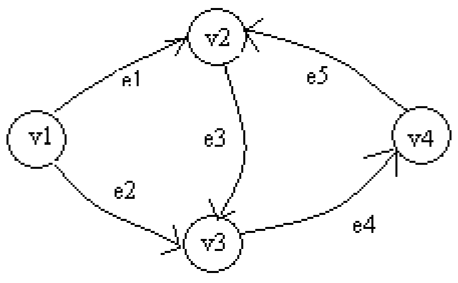


Fig: Directed Graph

Connected and Not-connected Graphs:

If every node in a graph is reachable from every other then the graph is known to be *Connected Graph* otherwise it is *Not-connected Graph.*

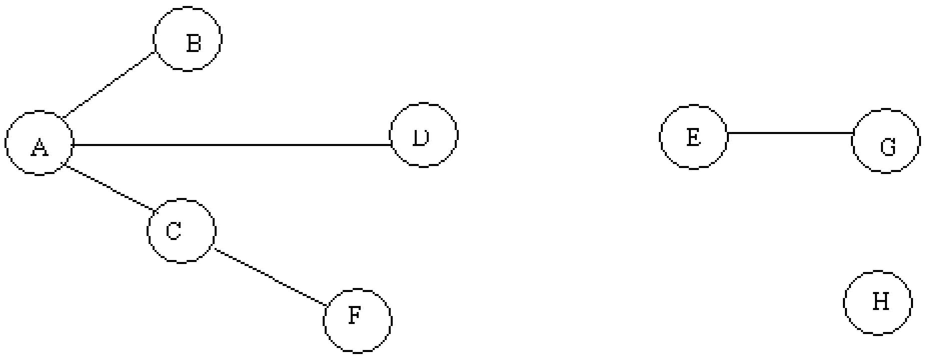


Fig: undirected and not-connected graph

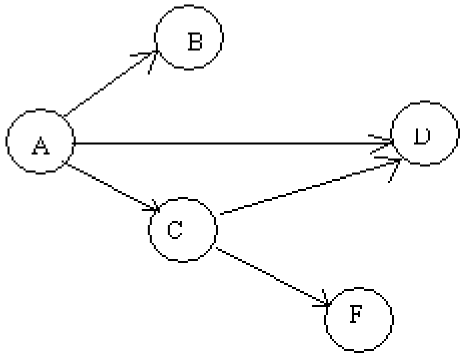


Fig: directed and connected graph

77.ii.Explain Directed and undirected graph and also connected and not connected graph?

A Graph consists of a set of nodes of (or vertices)and a set of arcs (or edges) .Each are in a graph in specified by a pair of nodes. Fig. 2 illustrates a Graph. The set of nodes is {a,b,c,d,e,f,g,h},an the set of arcs is a{(a,b),(a,d),(a,c),(c,d),(c,f),(e,g),(a,a)} if a pair of nodes that make up the arcs are ordered pairs,the graph is said to be a directed graph (or digraph) .Fig 3,4,5 illustrate three digraphs ,The arrows between nodes represent arcs .The head of each arrow represent the second node in the ordered pair of nodes making up an arc, and the tail of each arrow represent the first node in the pair. The set of arcs for the graph of fig 3 each { <a,b>,<a,c>,<a,d>,<c,d>,<f,c>,<e,g>,<a,a>}.We used parentheses to indicated an unordered pair and angled brackets to indicate an ordered pair .

An undirected graph may be considered a symmetric directed graph, that is ,one in which an arc <b,a> must exist whenever an arc <a,b>exists .The undirected arc (a,b) represents the two directed arcs <a,b> and <b,a>.An undirected graph may therefore be represented as a directed graph using either the adjacency matrix or adjacency list method .

In an undirected graph containing an arc (x,y),x and y must be part of the same tree. Since each is reachable from the other via that arc at least .A cross edge in an undirected graph is possible only if three nodes x,y ,and z are part of cycle and y and z are in separate sub trees of a sub tree whose root is x .The part between y and z must then include a cross edge between the two sub trees.

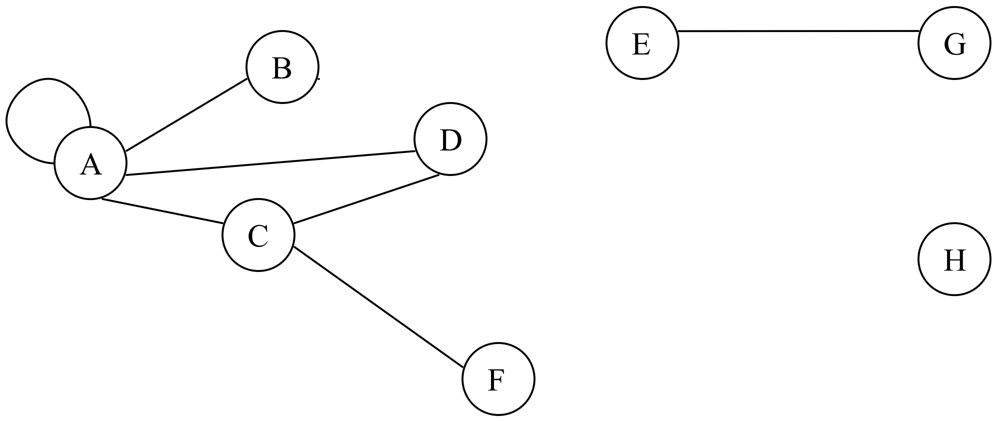


FIG 2

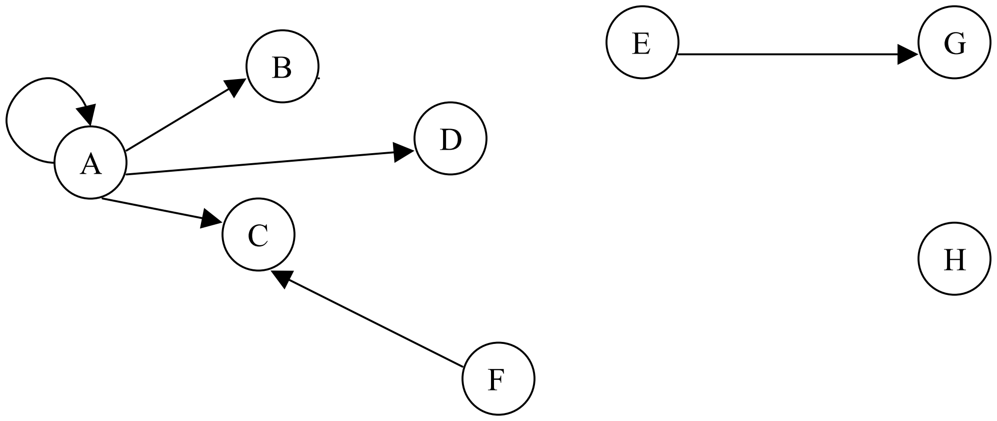


FIG 3

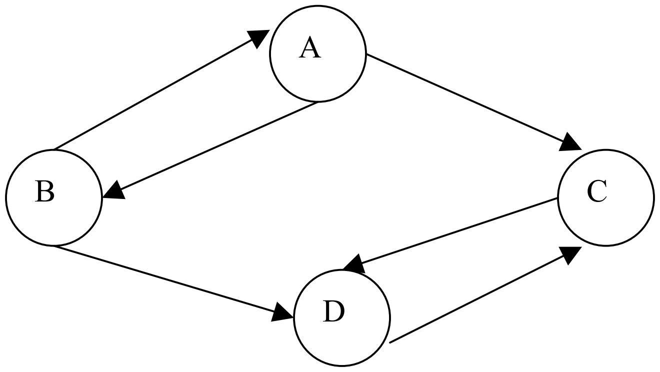


FIG 5

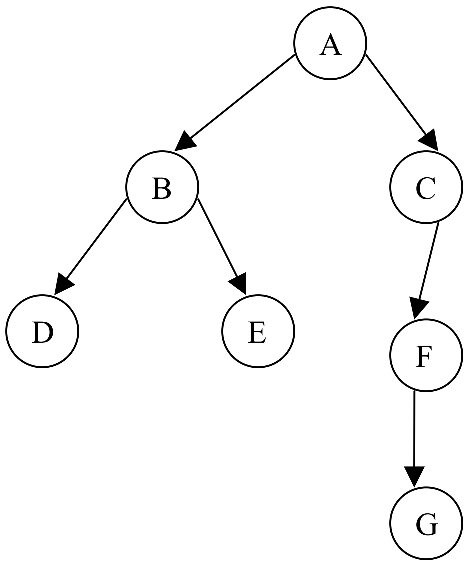


FIG 4

An undirected graph is termed connected if every node in it is reachable from every other .Pictorially, a connected graph has only one segment for e.g. the graph of fig 6 is a connected graph .The graph of fig 2 is not connected ,since node E is not reachable from node C ,for e,g.A connected component of an undirected graph is a connected sub graph is reachable from any node in a sub graph. For e.g., the sub graph fig. 1 has three connected component: nodes a,b,c,d,f ;nodes E and G ; and node H. A connected graph has a single connected component.

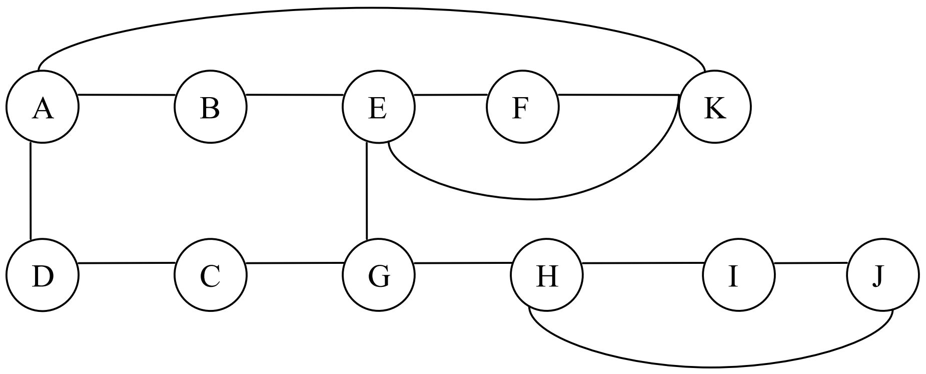


Fig 6

78.i.What is the application of Topological Sort? Explain.

The various application of topological sort are:-

1)topological sort is used to arrange items when some pairs of items have no comparison, that is, according to a partial order.

2) A topological sort is a permutation *p* of the vertices of a graph such that an edge{i,j} implies that *i* appears before *j* in *p* (Skiena 1990, p. 208). Only directed acyclic graphs can be topologically sorted. The topological sort of a graph can be computed using Topological Sort[*g*] in the *Mathematica* add-on package.

3) The topological sort algorithm creates a linear ordering of the vertices such that if edge *(u,v)* appears in the graph, then *v* comes before *u* in the ordering. The graph must be a directed acyclic graph (DAG). The implementation consists mainly of a call to depth_first_search().

4)topological sort command tsort reads input from *file* or from the standard input if you do not specify a *file* and produces a totally ordered list of items consistent with a partial ordering of items provided by the input.

Input to tsort takes the form of pairs of items (non-empty strings) separated by blanks. A pair of two different items indicates ordering. A pair of identical items indicates presence, but not ordering.

5) A topological sort attempts to make sense of a DAG. It returns a special sorting of the vertices so that all the directed edges go from a previous vertex in the list to a later vertex (no forward edges).

6) Topological sorting can be used to schedule tasks under precedence constraints. Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. These precedence constraints form a directed acyclic graph, and any topological sort (also known as a ***linear extension***) defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.

78.ii.What is the application of Topological Sort? Explain.

The various application of topological sort are:-

1)topological sort is used to arrange items when some pairs of items have no comparison, that is, according to a partial order.

2) A topological sort is a permutation *p* of the vertices of a graph such that an edge{i,j} implies that *i* appears before *j* in *p* (Skiena 1990, p. 208). Only directed acyclic graphs can be topologically sorted. The topological sort of a graph can be computed using TopologicalSort[*g*] in the *Mathematica* add-on package.

3) The topological sort algorithm creates a linear ordering of the vertices such that if edge *(u,v)* appears in the graph, then *v* comes before *u* in the ordering. The graph must be a directed acyclic graph (DAG). The implementation consists mainly of a call to depth_first_search().

4)topological sort command tsort reads input from *file* or from the standard input if you do not specify a *file* and produces a totally ordered list of items consistent with a partial ordering of items provided by the input.

Input to tsort takes the form of pairs of items (non-empty strings) separated by blanks. A pair of two different items indicates ordering. A pair of identical items indicates presence, but not ordering.

5) A topological sort attempts to make sense of a DAG. It returns a special sorting of the vertices so that all the directed edges go from a previous vertex in the list to a later vertex (no forward edges).

6) Topological sorting can be used to schedule tasks under precedence constraints. Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. These precedence constraints form a directed acyclic graph, and any topological sort (also known as a ***linear extension***) defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.

79.i..Explain the Kruskal's method of Spanning Tree. Compare this with Round Robin's Algorithm.

Kruskal's method of spanning tree is an algorithm for computing a *minimum spanning tree*. It maintains a set of partial minimum spanning trees, and repeatedly adds the shortest *edge* in the *graph* whose *vertexes* are in different partial minimum spanning trees.

Kruskal's algorithm for finding a minimal spanning tree in a connected graph is a greedy algorithm; that is, given a choice, it always processes the edge with the least weight.

This algorithm operates by considering edges in the graph in order of weight from the least weighted edge up to the most while keeping track of which nodes in the graph have been added to the spanning tree. If an edge being considered joins either two nodes not in the spanning tree, or joins a node in the spanning tree to one not in the spanning tree, the edge and its endpoints are added to the spanning tree. After considering one edge the algorithm continues to consider the next higher weighted edge. In the event that a graph contains equally weighted edges the order in which these edges are considered does not matter. The algorithm stops when all nodes have been added to the spanning tree.

Note that, while the spanning tree produced will be connected at the end of the algorithm, in intermediate steps Kruskal can be working on many independent, non-connected sections of the tree. These sections will be joined before the algorithm completes.

Often this algorithm is implemented using parent pointers and **equivalence classes**. At the start of the processing, each vertex in the graph is an independent equivalence class. Looping through the edges in order of weight, the algorithm groups the vertices together into one or more equivalence classes to denote that these nodes have been added to the solution minimal spanning tree.

It is a good idea to process the edges by putting them into a min-heap. This is usually much faster than sorting the edges by weight since, in most cases, not all the edges will be added to the minimal spanning tree.

Whereas ROUND ROBIN ALGORITHM provides even better performance than kruskal's algorithm when the number of edges are low. Kruskal's algorithm is o(e loge)Whereas Round robin algorithm requires only o(e log logn) operations.

The Kruskal Algorithm starts with a *forest* which consists of n trees. Each and everyone tree, consists only by one node and nothing else. In every step of the algorithm, two different trees of this *forest* are connected to a bigger tree. Therefore ,we keep having less and bigger trees in our *forest* until we end up in a tree which is the *minimum genetic tree (m.g.t.)* .In every step we choose the side with the least cost, which means that we are still under greedy policy. If the chosen side connects nodes which belong in the same tree the side is rejected, and not examined again because it could produce a circle which will destroy our tree. Either this side or the next one in order of least cost will connect nodes of different trees, and this we insert connecting two small trees into a bigger one. Whereas The Round-Robin algorithms for the short-term scheduler are the only preemptive algorithms that will be used. All partial trees are maintained in a queue ,q. associated with each partial treat, is a priority queue(t),of all arcs with exactly one incident node in the tree, ordered by the weights of the arcs. Initially, as in kruskal's , each node is a partial tree. A priority queue of all arcs incident to nd is created for each node nd, and the single-node trees are inserted into q in arbitrary order.

79.ii. Exlain Kruskal's method of Spanning Tree .Compare this with Round Robin's Algorithm.

Kruskal's method of spanning Tree is one of the methods of creating minimum spanning tree. In this method nodes of the graph are initially considered as ' n' distinct partial trees with one node each. Then two distinct partial trees are connected into a single partial tree by an edge of the graph. While connecting two distinct trees the arc of minimum weight should be used, for that arcs are placed in a priority queue on the basis of weight. Then the arc of lowest weight is examined to see if it connects two distinct trees. To determine if an arc (x, y) connects distinct trees, we can implement the trees with a father field in each node. Then we can traverse all ancestors of x and y to obtain the roots of the trees containing them. If the roots of the two trees are the same node, x and y are already in the same tree, so arc (x, y) is discarded, and the arc of next lowest weight is examined. Combining two trees simply involves setting the father of the root of one to the root of the other. This method requires O (e log e) operations

Round Robin's algorithm is another method of spanning tree, which provides better performance when the number of edges is low. This algorithm is similar to Kruskal's method except that there is a priority queue of arcs associated with each partial tree, rather than one global priority queue of all unexamined arcs. For this at first all partial trees are maintained in a queue,Q. Associated with each partial tree, T, is a priority queue ,P(T),of all arcs with exactly one incident node in the tree, ordered by the weights of the arcs. Then a priority queue of all arcs incident to 'nd' is created for each node 'nd', and the single–node trees are inserted into Q in arbitrary order. The algorithm proceeds by removing a partial tree,T1, from the front of Q; finding the minimum –weight arc a in P(T1);deleting from Q the tree ,T2,at the other end of arc a; combining T1 and T2 into a single new tree T3 [and at the same time combining P(T1) and P(T2),with a deleted ,into P(T3)];and adding T3 to the rear of Q. This continues until Q contains a single tree: the minimum spanning tree. This algorithm requires only O(e log n) operations if appropriate implementation of the priority queues is used .

80.i. Discuss about Greedy algorithms and Dijkstra's algorithm to find shortest path in a graph.

To find the shortest path between points, the weight or length of a path is calculated as the sum of the weights of the edges in the path. A path is a shortest path if there is no path from x to y with lower weight. Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects. It starts out at one vertex and branches out by selecting certain edges that lead to new vertices. Djikstra's algorithm solves the problem of finding the shortest path from a point in a graph (the source) to a destination. It turns out that one can find the shortest paths from a given source to all points in a graph in the same time, hence this problem is sometimes called the single-source shortest paths problem. This problem is related to the spanning tree one. The graph representing all the paths from one vertex to all the others must be a spanning tree - it must include all vertices. There will also be no cycles as a cycle would define more than one path from the selected vertex to at least one other vertex. For a graph,

- G=**(V,E)** where
- **V** is a set of vertices and
- **E** is a set of edges.

Dijkstra's algorithm keeps two sets of vertices:
S- the set of vertices whose shortest paths from the source have already been determined and
V-S the remaining vertices
The other data structures needed are:
d array of best estimates of shortest path to each vertex
pi an array of processors for each vertex

The basic mode of operation is:

1. Initialize **d** and **pi**,
2. Set **S** to empty,
3. While there are still vertices in **V-S**,
    i. Sort the vertices in **V-S** according to the current best estimate of their distance from the source,
    ii. Add **u**, the closest vertex in **V-S**, to **S**,
    iii. **Relax** all the vertices still in **V-S** connected to **u**

The **relaxation** process updates the costs of all the vertices, **v**, connected to a vertex, **u**, if we could improve the best estimate of the shortest path to **v** by including **(u,v)** in the path to **v**.

80.ii. discuss about Greedy and Dijkstra's algorithm to find shortest path in a graph.

In stead of building the tree from the top down, as in balancing method, the Greedy method builds the tree from the bottom up. The method uses a doubly linked linear list in which each list element contains four pointers, one key value and three probability values. The four pointers are left and right list pointers used to organize the doubly linked list and left and right sub tree pointers used to keep track of binary search sub trees containing keys less than and greater than the key value in the node. The three probability values are the sum of the probabilities in the left sub tree , called the left probability, the probability p(i) of the node's key value k(i), called the key probability, and the sum of the probabilities in the right sub tree , called the right probability . The total probability of a node is defined as sum of its left, key, and right probabilities. Initially there are n nodes in the list. The key value in the ith node is k (i), its left probability is q (i-1), its right probability is q (i), its key probability is p (i), and its left and right sub tree pointers are null.

Each iteration of the algorithm finds the first node on the list whose total probability is less than or equal to its successor's (if no node qualifies, nd is set to the last node in the list) .The key in nd becomes the root of a binary search sub tree whose left and right sub trees are the left and right sub trees of nd .nd is then removed from the list. The left sub tree pointer of its successor (if any) and the right sub tree pointer of its predecessor (if any) are reset to point to the new sub tree, and the left probability of its successor and the right probability of its predecessor are reset to the total probability of nd .This process is repeated until only one node remains on the list .when only one node remains on the list. Its key is placed in the root of the final binary search tree, with the left and right sub tree pointers of the node as the left and right sub tree pointers of the root.

Dijkstra's Algorithm:

In a weighted graph, if all weights [let weight (i, j) is the weight of the arc from i to j, if there is no arc from i to j, weight (i, j) is set to an arbitrarily large value to indicate the infinite cost (that is impossibility) of going directly from i to j.] are positive, the Dijkstra's algorithm is used to find the shortest path between two nodes, s and t (i.e. from s to t). Let the variable infinity hold the largest possible integer. Distance[i] keeps the cost of the shortest path known thus far from s to i. Initially, distance [s] =0 and distance[i] =infinity for all i! =s. A set perm contains all nodes whose minimal distance from s is known-that is, those nodes whose distance value is permanent and will not change. If a node i is member of perm, distance [i] is the minimal distance from s to i. Initially, the only member of perm is s. Once t becomes a member of perm, distance[t] is known to be the shortest distance from s to t, and the algorithm terminates. This implementation was O (n^2) , where n is the number of nodes in the graph. In most cases this algorithm can be implemented more efficiently using adjacency lists.