Data Structure And Algorithm
Course Material

1. Define data structure and algorithm. Why is it important to study?

Data Structure is an organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree, or conceptual unity, such as the name and address of a person. It may include redundant information, such as length of the list or number of nodes in a sub tree. And algorithm is a computable set of steps to achieve a desired result.

It is important to study due to following facts:

- To identify and develop useful mathematical entities and operation to determine what classes of problems can be solved by using these entities and operations.
- To determine representations for those abstract entities and to implement the abstract operations on these concrete representations.
- To save computer memory because the memory of the computer is limited.
- To maintain the execution time of the program, which determines the efficiency?

2. What is ADT? Write down the method of specifying ADT. How do you define Stack and Queue as an ADT?

A useful tool for specifying the logical properties of a data type is the ADT (Abstract Data Type). A data type is a collection of values and a set of operations on those values. The term "abstract data type" refers to the basic mathematical concept that defines the data type.

There are a numbers of methods for specifying an ADT**.** The method that we use is semiformal and borrows heavily from C notation but extends those notations where necessary. The operations on real numbers that we define are the creation of a rational numbers from two integers, addition, multiplication, and testing for equality. The following is an initial specification of this ADT.

```
/* value  definitions */
      abstract typedef <integer, integer>RATIONAL;
      condition RATIONAL[1]!=0;
/*operator definition*/
      abstract  RATIONAL makerational(a,b)
            int a,b;
            precondition b!=0;
            post condition make rational[0]= =a;
                           makerational[1]==b;

      abstract add(a,b)       /*written a+b*/

            Post condition add[1]= = a[1]*b[1];
                           add[0]= = a[0]*b[1] +b[0]*a[1];
      abstract mult(a,b)      /*written a*b*/

            Post condition mult[1]== a[1]*b[1];
                           mult[0]== a[0]*b[0];

      abstract equal(a,b)   /* written a==b*/
            Post condition equal==(a[0]*b[1]= = a[1]*b[0]);
```

STACK AS AN ABSTRACT DATA TYPE

A stack is an ordered collection of items into which new items may be inserted and from which items may be removed at one end called the *top* of stack. The representation of a stack as an abstract data type is straight forward. We use eltype to denote the type of the stack element and parameterize the stack data type with eltype :

```
abstract typedef<<eltype>>  STACK(eltype)
abstract empty(s)
      STACK(eltype) s;
      Post condition empty = = ( len(s)= =0);

abstract eltype pop(s)
      STACK(eltype)s;
           pre condition empty(s) = =FALSE;
           post condition pop = = first(s)
                         s= = sub(s,1,len(s)-1);

abstract push(s,elt)
      STACK(s,elt)
           eltype elt;
           post condition s = = <elt>+s;
```

QUEUE AS AN ADT

```
Abstract typedef<<eltype>> queue(eltype)
abstract empty(q)
      queue (eltype)q;
      Post condition empty = = ( len(q)= =0);
abstract queue insert (q,elt)
     queue (eltype)q;
     post condition q = p +<elt>

abstract queue remove(q,elt)
      queue(empty)q;
      pre condition empty(q)== FALSE;
      post condition remove == first(q)
                      q == sub(q,1,len(q)-1)
```

3. Define array as an ADT. Explain the concept of Multi-Dimensional Array.

We can represent an array as an abstract data type with a slight extension of the convention and notation. The elements of an array can themselves be arrays. This is a special case of having arrays of Objects. But in this case the original array is said to be a multidimensional array. The following example declares and creates a rectangular integer array with 10 rows and 20 columns:

```
int a[][] = new int[10][20];
```

It can be initialized by code such as

```
for (int i = 0; i < a.length; i++)
   for (int j = 0; j < a[i].length; j++)
       a[i][j] = 0;
```

The elements are accessed as a[i][j]. This is a consistent notation since a[i][j] is element j of the array a[i]. This element is said to be in row i and column j. Note that a.length has the value 10 and each a[i].length has the value 20. Two-dimensional arrays can, of course, be square e.g.

```
  int a[][] = new int[10][10];
```

They can also have different numbers of columns in each row. For example, the following code declares a (lower) triangular array in which row a[i] has i+1 element.

```
int a[][] = new int[10][];
for (int i = 0; i < a.length; i++)
    a[i] = new int[i+1];
```

Such arrays can be useful for representing symmetric matrices, `a[i][j] == a[j][i]`, where it is only necessary to operate on, and store, one copy of the off-diagonal elements. The following code initializes such an array to the unit matrix, with `1`'s on the diagonal and `0`'s elsewhere:

```
for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < i; j++) {
        a[i][j] = 0;
    }
    a[i][i] = 1;
}
```

The result is

```
1
0 1
0 0 1
if a.length == 3.
```

Multi-dimensional arrays can also be created and initialized using initialize lists as in

```
int a[][] = {{54, 64}, {98, 12, 67}};
```

4. What is Stack? How stack can be implemented?

5. What are the primitive operations of stack? Write down the implementations in C.

A stack is an ordered collection of the items into which new items may be inserted and from which items may be deleted at one end, called the top of the stack. A stack is also called a *Last-In-First-Out* (LIFO) list.

The two changes, which can be made to a stack, are given special names. When an item is added to stack, it is pushed onto the stack, and when the item is removed, it is popped from the stack. Given a stack s, and an item i, performing the operation **push(s,i)** adds the item i, to the top of the stack s. Similarly, the operation **pop(s)** removes the top element and returns it as a function value. Thus the assignment operation

```
i= pop(s);
```

removes the element at the top of s and assigns its value to i.

Because of the push operation, which adds elements to a stack, a stack is sometimes called a pushdown list. If a stack contains a single item and the stack is popped, the resulting stack contains no items and is called the *empty* stack. Although the push operation is applicable to any stack, the pop operation cannot be applied to empty stack because such a stack has no element to pop. Therefore, before applying the pop operator to a stack, we must ensure that the stack is not empty. The operation **empty(s)** determines whether or not a stack s is empty. If the stack is empty, empty(s) returns the value TRUE; otherwise it returns the value FALSE. The **stacktop(s)** returns the top element of the stack s. The operation stacktop(s) can be decomposed into a pop and push.

```
i = stacktop(s);
```

is equivalent to

```
i = pop(s);
push(s,i);
```

The result of an illegal attempt to pop or access, an item from an empty stack is called underflow. Underflow can be avoided by ensuring that empty(s) is false before attempting the operation pop(s)or stacktop(s).

The implementation in C for the primitive operation of stack of character is given bellow:

```
struct stack{
      int top;
      char items[size];
      };

void push(struct stack *ps, char x)
{
      if(ps->top==size-1){
            printf("%s","stack overflow");
            exit(1);
      }
      else
            ps->items[++(ps->top)]=x;
      return; }


char pop(struct stack *ps)
{
     if(empty(ps))
      {
            printf("%s","stack underflow");
            exit(1);
      }
      return(ps->items[ps->top--]);
}

int empty(struct stack *ps)
{
      if(ps->top==-1)
       return 1;
      else
       return 0;
}

int stacktop(struct stack *ps)
{
if(empty(ps)){
      printf("%s","stack underflow");
      exit(1);
}
else
      return(ps->items[ps->top]);
}
```

6. Write down the algorithm of evaluating postfix operation using stack.

In postfix notations the operator follows the two operands. The postfix notations are not really as awkward to use as they might at first appear. Each operator in a postfix string refers to the previous two operands in the string.

Each time we read an operand we push it into a stack. When we reach an operator, its operands will be the top of two elements on the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack so that it will be available for use as an operand of the next operator. The following algorithm evaluates an expression in postfix using this method:

```
Opndstk = the empty stack;
/* scan the input string reading one */
```

```
/* element at a time into symb       */
while (not end of input) {
      symb = next input character;
      if (symb is an operand)
            push(opndstk, symb);
      else {
            /* symb is an operator */
            opnd2 = pop(opndstk);
            opnd1 = pop(opndstk);
value = result of applying symb to opnd1 and        opnd2;
push(opndstk, value);
      } /* end else */
} /* end while */
return (pop(opndstack));
```

Let us now consider an example. Suppose that we are asked to evaluate the following postfix operation:

6 2 3 + - 3 8 2 / + * 2 $ 3 +

Here we show the content of the stack opndstk and the variables symb, opnd1, opnd2 and value after each successive iteration of the loop. The top of stack opndstk is to the right.

symb	***opnd1***	***opnd2***	***value***	***opndstk***
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Each operand is pushed into the operand stack as it is encountered. Therefore the maximum size of the stack is the number of operands that appear in the input expression. However, in dealing with most postfix expressions the actual size of the stack needed is less than this theoretical maximum, since an operator removes operands from the stack. In the previous example the stack never contain more than four elements, despite the fact eight operands appeared in the postfix expression.

7. How do you convert an expression from infix to postfix using stack. Explain with the help of an example.

8. What is Queue? What are the different types of Queue?

A queue is an ordered collection of the items from which the items may be inserted at one end and also can be deleted form the other end. Queue, one of the important parts of the data structure has two parts: Front and Rear. The items that are newly created are inserted from the rear side of the queue and the items to be removed are deleted from the front side of the queue. A queue is also called as a FIFO (first-

in-first-out) list because the items that are inserted first are deleted or removed from the queue first. Examples of the queue can be taken in account to the real world.
Example: line of people at bank, bus stop etc. The fig below illustrates the queue and the ways of inserting and deleting the items form the queue.

Fig 1: A queue containing three elements A, R, S.

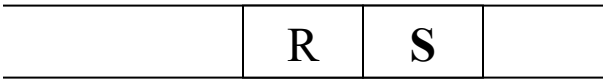


Fig 2: item 'A' deleted from the front side of queue.

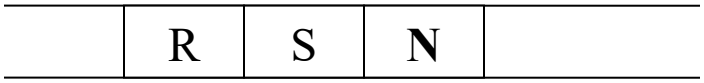


Fig 3: item 'N' being inserted from rear side of queue.

Fig: Queue

Depending upon the ways of inserting and deleting the queue it is of three types:

a) Linear Queue
b) Circular Queue
c) Priority Queue

The difference between Linear queue and circular queue is at the time when the queue is full i.e. one of the pointer has reached to the top of the queue.

In the linear queue rear = -1 and front = 0. The linear queue is said to be empty whenever rear < front and the number of elements present in the queue is given by rear – front +1. The main thing is that though the queue is empty still but rear has reached to the location size -1 (size denotes the number of elements that the queue can hold), no new elements can be inserted to the queue. On an attempt to insert the elements in this case, it would result in an overflow error.

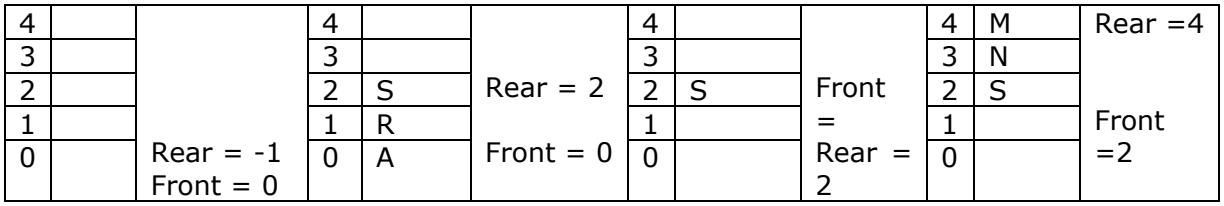


Figure: Elements being inserted and deleted from Linear Queue

If another element, say 'T' is tried to insert then it will result in an overflow error though the queue is still empty.
But in the case of circular queue both rear and front = -1 initially. In the circular queue, the first element of the array is immediately followed by the last element. This implies that even if the last element is

occupied, a new value can be inserted behind it in the first element of the array as long as that first element is empty.

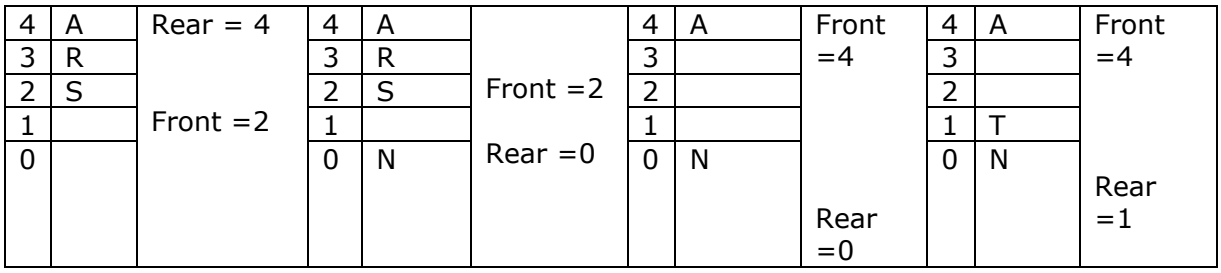


Figure: Elements being inserted and deleted from Circular Queue

If a new element say 'T' is if now tried to insert then it will be inserted at the rear side as shown in above figure.

A priority queue is a homogeneous data structure of variable size, initially empty, but allowing the insertion and extraction of elements during the execution of a program. In a priority queue each element carries with it a value - its priority - by which the extraction order is determined: An element cannot leave a priority queue until after all the elements of greater priority have been extracted. A priority queue is like a waiting line in which a high-priority element can ``pull rank'' on elements with lower priorities, cutting in line ahead of them. But priority queues are generally used in cases where the priorities are independent of time. The ordering of the elements in the priority queue can be done in two ways: Ascending and Descending.

An ascending priority queue is a collection of items into which the elements can be inserted arbitrarily and from which the smallest item can be removed first whereas in DPQ it allows the deletion of the largest element first.

9. Write down the algorithm of En-queue and De-queue in Linear and Circular Queue separately.

Algorithm of Enqueue and Dequeue in linear queue:

Algorithm for main function

1. Start
2. Define size 10
3. Declaration of necessary variables a, I, x, y
4. Declare integer variable named items [size].
5. Declare a structure named queue *q
6. Assign q->front=0 and q->rear=-1.
7. Use for loop from i=0 to i<size. Inside the loop ask a value to enter which represents 1 for insert and 2 for delete.
    7. i. Compare the value of a, if a=1, go to step 8, if a=2 go to step 9 else print invalid.
8. Ask the value to be inserted i.e. x, then call function insert.
9. Call function rem (q) and assign it to y. Display y.
11. Stop

Algorithm for different functions

Insert Function:

1. Start

2. Check if q->rear = q->front, if true print queue is overflow and exit else go to step3.
3. Increase q->rear by 1.
4. Insert the value of x to q->items [q->rear].
5. Stop.

Remove function

1. Declare an integer variable a.
2. Call and check function empty. If empty (q) is equal to 1, display queue overflow, then exit. Else go to step3
3. Assign a=p->items [p->front]
4. Increase p->front by 1
5. Return a
6. Stop

Empty function

1. Check if q->rear <q->front, if yes go to step2 else go to step3.
2. Return 1. (True)
3. Return 0 (False)
4. Stop

Algorithm of Enqueue and Dequeue in circular queue

Algorithm for Main function:

1. Start
2. Define size 10
3. Declaration of necessary variables a, I, x, y
4. Declare integer variable named items [size].
5. Declare a structure named queue *q
6. Assign q->front= -1 and q->rear= -1.
7. Use for loop from i=0 to i<size. Inside the loop ask a value to enter which represents 1 for insert and 2 for delete.
    7. i. Compare the value of a, if a=1, go to step 8, if a=2 go to step 9 else print invalid.
8. Ask the value to be inserted i.e. x, then call function insert.
9. Call function rem (q) and assign it to y. Display y.
11. Stop

Algorithm for Different functions

Insert function

1. Start
2. Check if q->rear = =size-1, assign q->rear=0 else increase q->rear by 1.
3. Check if q->rear= =q->front, display queue overflow, then exit
4. Assign q->items [q->rear] = x
5. Stop

Remove function

1. Start
2. Declare an integer variable a
3. If empty (q) is equal to 1, display queue overflow, then exit
4. If q->front is equal to size-1, assign q->front=0 else go to step 5
5. Increase q->front by 1
6. Assign a=q->items [q->front]
7. Return a
8. Stop

Empty function

1. Start

2. If q->front is equal to q->rear go to step 3 else go to step 4
3. Return 1
4. Return 0
5. Stop

10. Explain the types of priority queue? Is it always necessary that the order of priority queue be in terms of values? If not, give some examples.

The priority queue is a data structure in which the intrinsic ordering of the elements determines results of the basic operation of the queue. Priority queues are useful data structures in simulations, particularly for maintaining a set of future events ordered by time so that we can quickly retrieve what the next thing to happen is. They are called ``priority" queues because they enable you to retrieve items not by the insertion time (as in a stack or queue), nor by a key match (as in a dictionary), but by which item has the highest priority of retrieval. Generally,there are two types of prority queues:

***Ascending priority queue**:*

An ascending prority queue is the queue in which elements are stored in such a way that while inserting elements they can be inserted randomly. But while removing the items, only smallest item can be removed. i.e. in ascending order. So, a priority queue is said to be an ascending-priority queue if the item with smallest key has the highest priority .Let the APQ be the ascending priority queue. It can have three operations like the ordinary queue:

I. Insert element
II. Remove element
III. To check whether the queue is empty or not

If y is the element to be inserted, then **insert(APQ, y)** inserts element y into APQ and **mindelete(APQ)** removes the minimum element from it and returns its value. The operation **empty(APQ)** checks whether the priority queue is empty or not. It is important because mindelete can be applied to only non-empty queue. So, for removing elements from the queue, this operation is used. Once mindelete has been applied to retrieve the smallest element of an ascending priority queue, it can be applied again to retrieve the next smallest element and so on. Thus the operation successively retrieves elements of the priority queue in ascending order.

Descending prority queue:

Descending the priority queue is just reversed to the ascending priority queue.In this queue,the items can be inserted in any order (randomly), but only the largest item can be removed from it. Priority queue is said to be a descending-priority queue if the item with largest key has the highest priority. The operations to the descending priority queue are also same i.e.

I. Insert element
II. Remove element
III. To check whether the queue is empty or not

The difference is only in the remove portion. In this case when an element is removed from the descending priority queue, the largest number is removed i.e. **maxdelete(DPQ)** remove the maximum element from DPQ and returns its value (DPQ be the descending priority queue). The insert and empty processes are logically same as that of ascending priority queue. So, maxdelete also retrieves element of descending priority queue in the descending order. Hence, this time also it is necessary to check whether the queue is empty or not using operation empty (DPQ) before deletion of the element from the queue.

The above description explains the designation of the priority queue as either ascending or descending. The above discussion of the priority queue has been made considering the elements of the queue as the numbers (values). But it is **NOT** always necessary that the priorities to be made on the basis of the numbers. Unlike value different other factors can also be considered in the

priority queue such as time, some external values specifically for the purpose of ordering on one or several fields. A priority queue whose elements are ordered by time of insertion becomes a stack.

11. What are the issues for the deletion of items in an array implementation of priority queue? What could be the possible solutions?

Priority queue is special type of queue. A queue can be implemented in an array so that each insertion or deletion involves accessing only a single element of the array. However, such operation is not possible with the priority queue. This is because while implementing the queue in array: under the insertion method the elements of the priority queue are not kept in order. This won't effect the operation of the priority queue because in the queue the elements can be entered randomly. So, as long as only insertions take place the array implementation woks well. This can be shown in the following pseudo code.

```
If ( DPQ.rear >= maxpq) {
      Print overflow
}
DPQ.items [DPQ.rear]=x;/*x is an element*/
Increase rear by 1;
```

But while removing or deletion of the elements there arise problems as the elements are to be removed either in ascending order or in the descending order. Let us consider descending priority queue. For deletion of the elements from the queue, first it is necessary to find the largest element in the array. For this every element of the array from dpq.items[0] to dpq[dpq.rear-1] have to be examined. Thus, in order to delete an element from a priority queue, it requires accessing of every element of the priority queue. Another problem may arise while deletion of the element at the middle of the array, because insertion is made randomly and there is maximum possibility to have largest element at the middle of the array. So, the deletion of elements in the priority queue under the array implementation requires both searching of element that is to be deleted and the removal of an element in the middle of an array.

So, for implementing the priority queue these issues should be removed.There are several possilble solutions,but are not entirely satisfactory.They may be:

I. Place a splecial indicator "empty" on the deleted portion. However, This requires to examine all the deleted array positions also.

II. To overcome the above mentioned problem, the insertion operation can be slightly modified to insert new element on the first empty location.

III. Compact the array by shifting all the elements past the deleted element one position and decrement the rear position by 1. This increase the inefficiency in deletion.

IV. A slight improvement can be made by shifting either all preceding elements forward or all succeeding elements backward, depending upon which group is smaller maintaining the front and rear indicator while treating array as a circular structure.

V. Another approach may be to use priority queue as an ordered array rather than unordered. However, the insertion requires locating the proper position of the new element and shifting the preceding or succeeding elements. Other solutions involve leaving gaps in the array between the elements of the priority queue to allow for subsequent insertion.

Hence, the solution given above help to remove the difficulties involves in the deletion of the element in the priority queue, but need some restrictions.

12. What is circular queue? How can you implement a circular queue? Explain empty queue and full queue.

A queue is a sequential list in which items are inserted into the tail end of the queue and taken from the front i.e. FIFO (first in first out). When we add an element into the queue this item is now considered the item at the end of the queue. When we remove an item from the queue the

second item in the queue becomes the front element. Frequently the rear element is referred to as the "tail" and the front element is referred to as the "head" so we can say that the head is chasing the tail. Through the magic of modulus at some point we wrap around and the head will start at position 0 again. This is convenient. There is no marker to mark that we are at the end of our array. When we reach the end and need to store another variable we will overwrite the item in position 0.

Implementation of circular queue

Circular queue can be implemented in the following way:

- Use linear model, but wrap around from end of array to beginning
- Never need to move entries while on queue
- Doesn't waste storage on unusable entries
- Variables:
  - *MAX* is the number of entries in the array
  - *Front* is index of front queue entry in array
  - *Rear* is index of rear queue entry in array
- Wrap around condition:
  - Conditional: `I=(I <max-1) ? I+1 :0;`
  - Arithmetic: `I=(I+1)%max;`

It is known that queue contains two pointers, front through which the elements are removed and rear through which the elements are inserted. If there is no elements in the queue i.e. if the queue is *empty* then both of them are initialized to –1.So if rear of the queue equals to the front of the queue, then the queue is called empty queue. This can be shown as:

```
Int empty(queue *pq)
{
        return((pq->front= =pq->rear)?TRUE:FALSE);
}
```

This shows that if the condition is true it return that the queue is empty else it does not return.

If the rear of the queue points at the (maximum size of queue-1) then it indicates that the queue is *full*. If this condition is statisfied,then the queue is known as the full queue. If elements inserted with this condition the there occurs queue overflow. This can be shown as:

```
If(pq->rear==maxqueue-1)
{
        printf("queue overflow");
}
```

If the queue is empty it returns NULL, otherwise it returns the key of the item at the front. The queue is not affected.

alloc creates a new queue with n usable spaces and initializes all spaces to Nulls free frees a queue. En-queue adds an item and a key to the back of the line, used for FILO lists. It returns the total number of usable spaces remaining in the queue, or –1 if the routine just overwrote an old element. If the queue is full, qlength returns 0, since it can't distinguish a full queue from an empty one.

13. What are the conditions of overflow and underflow in a queue and stack? Explain with suitable example.

Unlike that of array, the definition of the stack provides for the insertion and deletion of items, so that stack is dynamic, constantly changing object. Hence, in the stack there is no limitation on the number of items that may be inserted on the stack, thus condition of overflow doesn't generally occur in case of stack. But during the *array* representation of the stack, when the stack contains as many elements as the

array and if an attempt is made to push yet another element onto the stack, this results in an overflow condition.
The result of an illegal attempt to pop or access an item from an *empty* stack results in underflow condition.

The queue is based on the FIFO principle, in which an item is inserted from one end (called front) and an item is removed from the other end (called rear).

Initially, the front of the queue is always assigned to zero (0) and the rear of the queue -1.The insert operation can always be performed since there is no limit to the number a queue may contains. But during the array representation of a linear queue, when the front of the queue is at the end of the array and rear = = front of the queue, though the queue is empty, the attempt to insert an item into the queue may result in overflow condition.

An illegal attempt to remove an item from an empty queue results in underflow condition. It occurs when rear = = front and rear is null.

14. What are the drawbacks of using sequential storage to represent stack and queue? What are the possible alternatives and its advantages?

The drawbacks of using sequential storage to represent stack and queue is listed below:

1. In a queue and stack, there is maximum possibility of the overflow and underflow condition to occur frequently.
2. In case of a queue, there is no distinct identification when the overflow and underflow condition occurs.
3. During the array representation of the stack and the queue, there is pre allocation of the memory, which may be used or mayn't be used during the stack or queue operations. When it is not used, it remains ideal and can't be used by any other processes. Moreover, the allocated memory may run out of the space if the items are large, causing overflow condition.
4. Since the stack is based on FILO and the queue on FIFO principle, it is very inefficient to insert or remove an item from the intermediate position of the stack or the queue.
5. Underflow and Overflow condition of the stack and the queue may cause the system to hang up.
6. It is very difficult to do searching and sorting of the items of the queue and the stack.

Linked list is the best possible alternatives to get rid of the flaw of the sequential storage of the queue and the stack. Since the linked list is the dynamic list, whenever an item is inserted, dynamically memory is allocated at that instant and whenever an item is removed, dynamically the unused memory is freed. Hence, the wastage of the unused memory is saved and hence the very memory can be used for other processes.
The next advantage is that we can efficiently insert and remove an item from any intermediate position of the list.

15. What is list? Write about the array implementation of list.

16. What are the different types of list? Explain.

The different types of list are as follows:

- Singly Linked List

- Linear Linked List
- Circular List

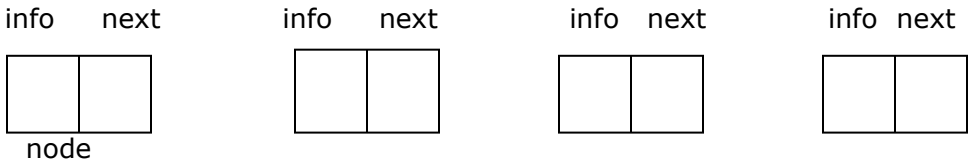
❖ Doubly Linked List
- Linear Doubly Linked List
- Circular doubly Linked List

Singly Linked List:

Singly Linked List is a sequential storage techniques introduced to fulfill the drawbacks of stacks and queues. In such list the items of a stack or a queue were explicitly ordered, that is, each item contained within itself the address of the next item.

Linear Linked List:

In such kind of list the items of a stack or a queue were explicitly ordered, that is, each item contained within itself the address of the next item. Such explicit ordering gives rise to a data structure as shown in the figure below which is known as a linear linked list.



Each item in the list is called a node and contains two fields, an information field and a next address field. The information field holds the actual element on the list. The next address field contains the address of the next node in the list. Such an address, which is used to access a particular node, is known as a pointer. The entire list is accessed from an external pointer list that points to the first node in the list. The next address field of the last node in the list contains a special value, known as null, which is not a valid address. This null pointer is used to signal the end of a list. The list with no nodes on it is called the empty list or the null list. The value of the external pointer list to such a list is the null pointer.

Circular List:

If the next field in the last node contains a pointer back to the first node rather than the null pointer then such a list is called a circular list and is illustrated in figure shown below.

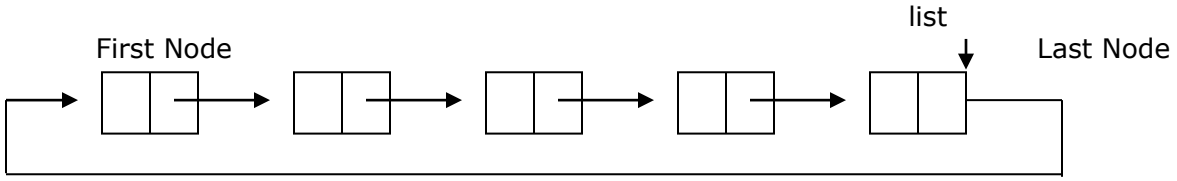


Figure 16.2: Circular List

From any point in such a list it is possible to reach any other point in the list. If we begin at a given node and traverse the entire list, we ultimately end up at the starting point. A circular list does not have a natural "first" or "last" node. Therefore, we must establish a first and last node by convention. One useful convention is to let the external pointer to the circular list point to the last node, and to allow the following node to be the first node as shown above. If p is an external pointer to a circular list, this convention allows access to the last node of the list by referencing node(p) and to the first node of the list by referencing node(next(p)). This convention provides the advantage of being able to add or remove an element conveniently from either the front or the rear of a list.

Doubly Linked List:

In Doubly Linked List each of its node contains two pointers, one to its predecessor and successor. In Doubly Linked List the terms predecessor and successor does not have real meanings because this list in entirely symmetric. Doubly Linked List is introduced which one more field in the structure has named

'prev' in dynamic implementation and 'left' in array implementation. Doubly Linked Lists may or may not contain a header node.

Linear Doubly Linked List:

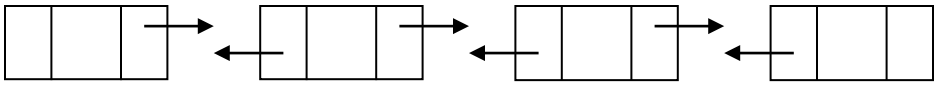


Figure 16.3: Linear doubly linked list

Such list consists of three fields: "Info" field that contains the information stored in the node, and "left" and "right" fields that contain pointers to the nodes on either side.

Circular Doubly Linked List:

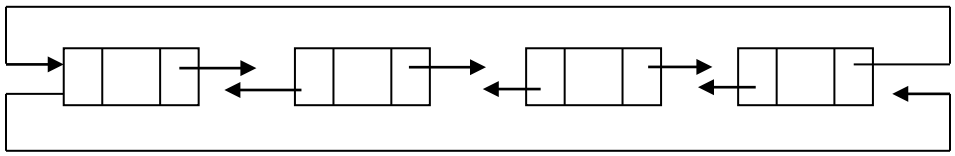


Figure 16.4: Circular doubly linked list

If the next field in the last node contains a pointer back to the first node and first node contains a pointer to the last node rather than the null pointer. Then such list is called circular doubly linked list.

17. Compare Singly Linked List with Doubly Linked List.

Singly Linked List is a sequential storage techniques introduced to overcome the drawbacks of stacks and queues. In such list the items of a stack or a queue were explicitly ordered, that is, each item contained within itself the address of the next item. The major drawbacks is the fixed amount of the storage, allocated to the stack or queue whether the structure is using smaller amount of storage or no storage at all and also no more than fixed amount of the storage may be allocated. Singly Linked List has number of nodes that are linked as a chain and the node contains two fields: - one is 'info' field, which contains the element of the list whereas another is 'next' field, which contains the address of the next node. The last node is determined by the 'next' field, which contains 'null'. So, as the one node points another node through pointer, it is of dynamic type so any number of the node can be inserted at any position and also any element of any node can be removed easily and efficiently. Singly Linked List can also be used easily for forward searching i.e. it can traverse in forward. But, it has some major drawbacks that are it can't traverse backward nor a node can be deleted from a linked list given only a pointer to that node.

In Doubly Linked List each of its node contains two pointers, one to its predecessor and successor. In Doubly Linked List the terms predecessor and successor does not have real meanings because this list in entirely symmetric. Doubly Linked List is introduced which one more field in the structure has named 'prev' in dynamic implementation and 'left' in array implementation. Also, we can say that in doubly linked list, each node has two pointers: - one to its predecessor and another to its successor. So, it is capable of pointing two nodes i.e. previous node as well as next node simultaneously. Thus, it provides facility of backward traverse and of deleting node given only a pointer to that node. As it is capable of traversing in both the direction, it is also used in adding of long integers.

Both the Doubly linked list and singly linked list has the application of linear and circular list. In Singly linked list although a circularly linked list has advantages over a linear list, it still has several drawbacks. One cannot traverse such a list backward, nor can a node be deleted from a circularly linked list, given only a pointer to that node.

18. Explain why Linked List is called Dynamic List?

Since linked list has two field, they are: 'info' field and a pointer to the '*next*' node. Also, in addition, an external pointer points to the first node in the list. Now, the programmer need not be concerned with managing available storage so if a new information is to be inserted then the memory is dynamically allocated for the new node by the "*malloc*" function and the pointer of 'next' field of the preceding node points to the new formed node. Similarly, when any information or element is to be deleted, then that particular node is freed and the pointer pointing to that node is moved to the succeeding node. Thus, here we can see that a set of nodes need to be reserved in advance for use by a particular group of lists and can allocate or free storing space as per users requirement. Because of this reason linked list is also known as Dynamic List.

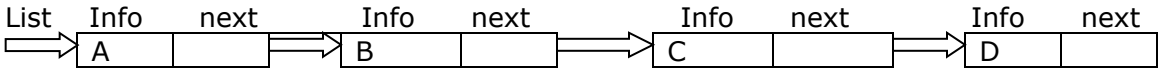
In another word, the linked list does not allocate storage memory for variables, until it is needed. As nodes are needed, the system is called upon to provide them. Any storing space not used for one type of node may be used for another. Thus, as long as sufficient storage is available for the nodes actually present in the lists, no overflow occurs, which is because of the dynamical memory allocation. Thus, linked list is also called Dynamic List.

19.

i. Write an algorithm for insertion and deletion of nodes from a list. Explain with figure.

A list is a dynamic data structure. The no. of nodes on a list may vary dramatically as elements are inserted and deleted. The dynamic nature of a list may be contrasted with the static nature of an array, whose size remains constant.

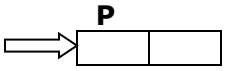
Let us discuss about the insertion of nodes in a list. For this let us consider a list as shown below.

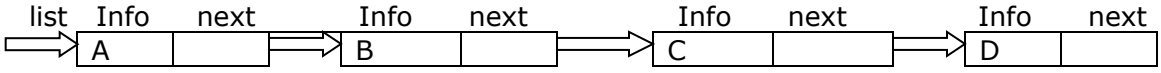


If we have to insert a node with content 'X' after the content 'C' then for this we will first check the content of each node of a list until the content 'C' is not found. Then a new node is allocated by the function getnode() and place the element 'X' in the info field of the new node and the pointer in the next field is made to point to the node that is followed by the node containing 'C' previously. And the node with content 'C' is made to point the newly formed node. In this way a node can be inserted. The algorithm and figure are as follows:

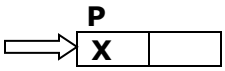
```
Insert()
{
        temp=list;
        while(temp.info!='C')
                temp=temp->next;
```

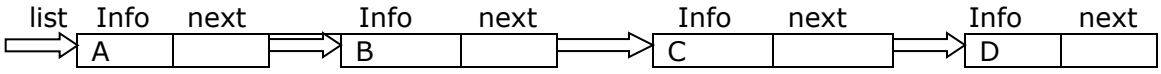
```
        p=getnode();
        p->info='X';
        p->next =temp->next;
        temp->next=p;
}
```



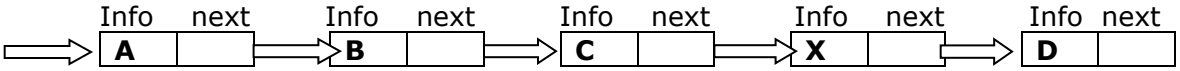


temp

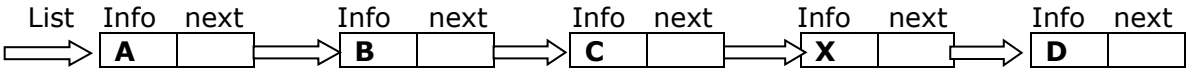




temp

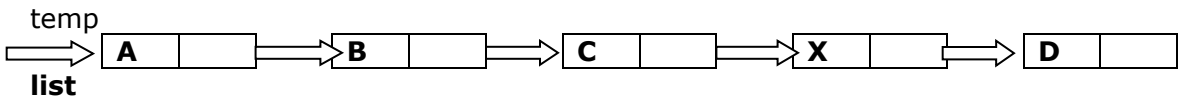


Now let us discuss about the deletion of the nodes from a list. For this, let us consider the above example, again which is as given below:



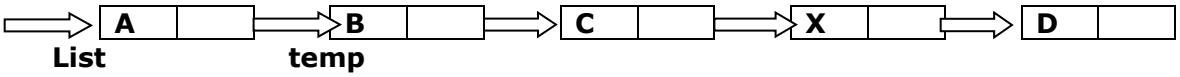
Let us consider we have to delete the node with the element 'C' in its info field. Then first of all the node with the content 'C' is scanned then the pointer pointing the node containing 'C' is made to point to the node which is pointed by the node containing 'C'. then the memory is freed by the use of function freenode(). The algorithm and figure can be summarized as below.

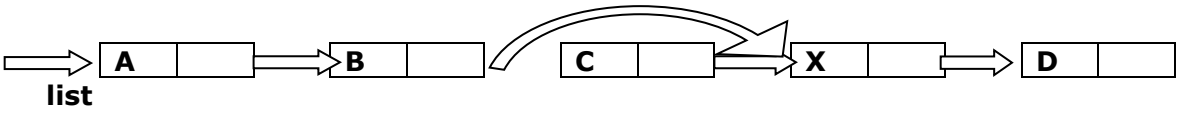
```
Delete(char value)              //eg delete(C)
{
        temp=list;
        while((temp->next)->info!=value)
                temp=temp->next;
        delnode=temp->next;
        temp->next=delnode->next;
        free(delnode);
}
```

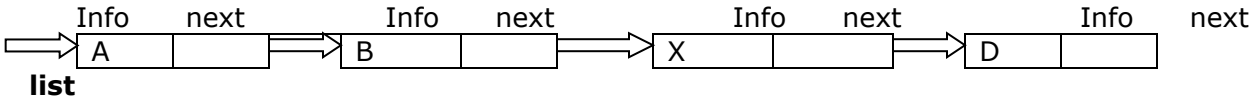


list

delnode



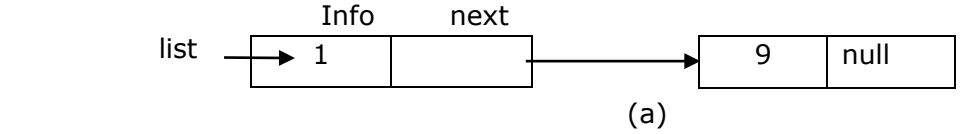




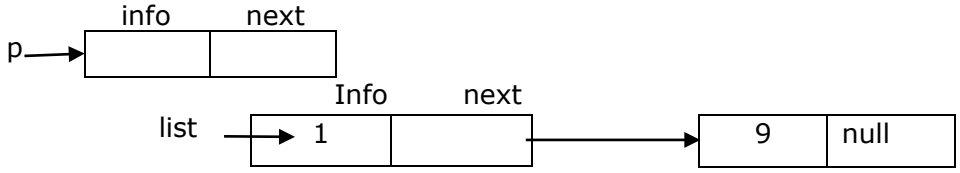
ii. Write an algorithm for insertion and deletion of nodes from a list. Explain with figure.

A list is a dynamic data structure of items called nodes. Each list in node consists of two fields, information field that holds actual element and an address field that contains the address of the next node in the list .the null pointer in a node signal the end of the list .The number of nodes in a may vary dramatically as a elements are inserted and removed.

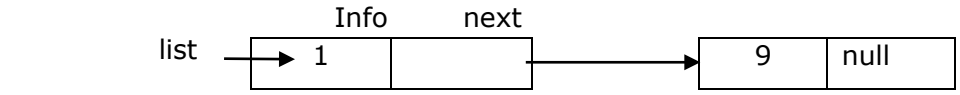
Now, let us consider we want to add int 3 in front of the list .The procedure for this is as shown in the figure 19. a



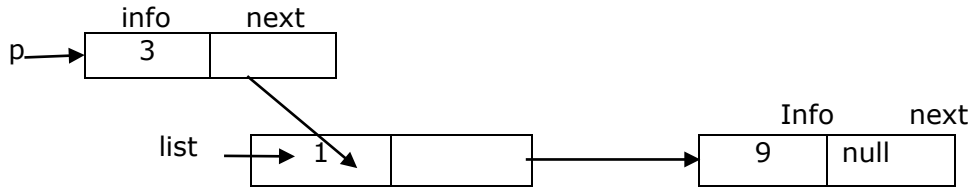
(a)



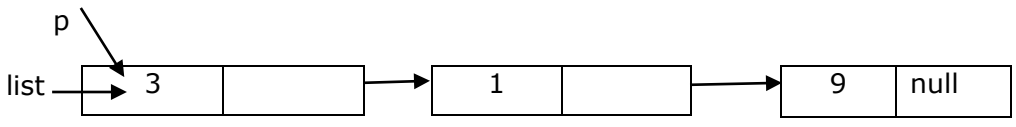
(b)



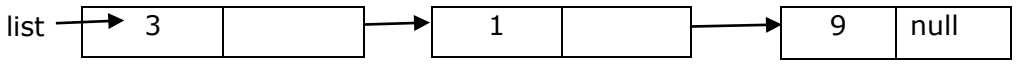
(c)



(d)



(e)



(f)

fig 19. a.Adding an element to the front of a list.

At first, we need to get an empty node using function get node () that returns a node p with fields info and next .The next is to insert 3 into info portion of newly allocated node i.e. info (P) =3 and let the next portion of that node at the front of he list such that the current first node on he list follows it. Since the variable list contains the address of the first node, node (p) list contains the address of the first node, node (p) can be added to the list by performing the operation.

next (p) list

The operation places the values of the list onto the next field of the node (P0.

At this point, p points to the list into the next item included .However, since list with additional item included .However, since list is the external pointer to the desired list, its value must be modified to the address of the near first node of the list .the list by performing operation

List=p;

which changes the value of the list to the value p.

The figure 1, e and f are similar except the auxiliary variable p is not shown in f Since its value is irrelevant to the list before and after the process.

Finally putting all steps together

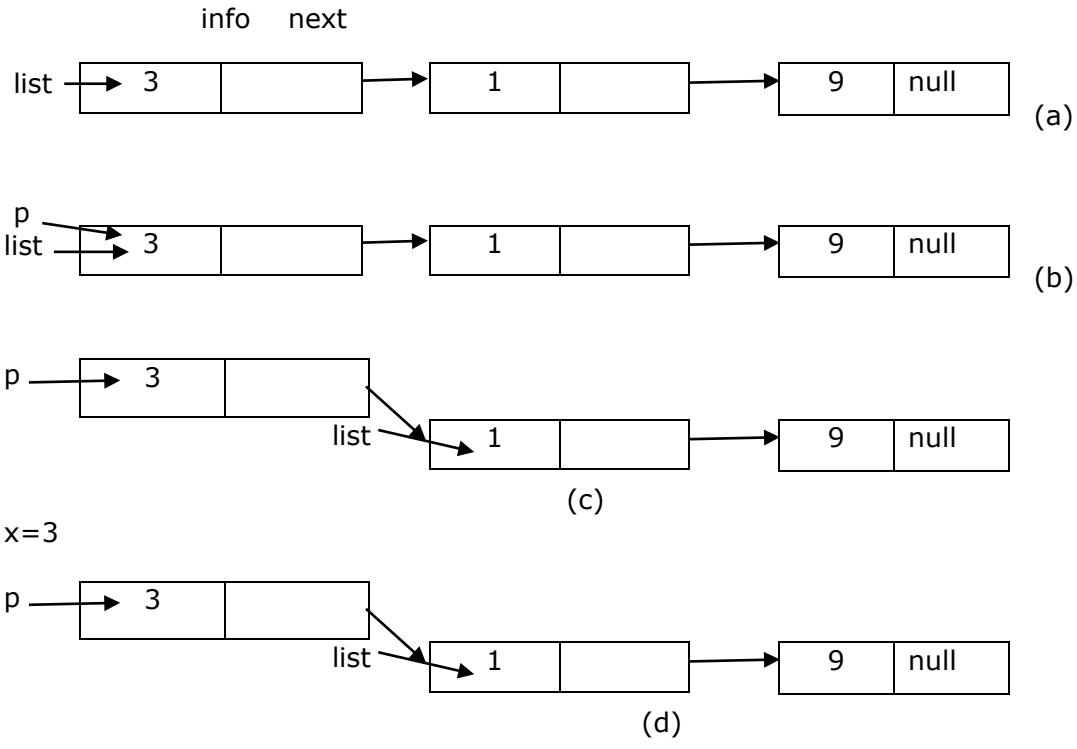
P=getnode()

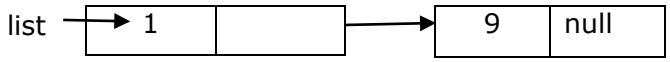
info (p) =3

next (p) =list;

list=p;

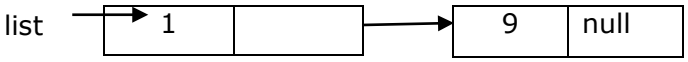
Now, let us consider we want to remove the first node of a non empty list and storing the value of its info a variable x. The process is almost the exact opposite of the process to a node to the front of a list and shown in figure 19. b.





(e)

x=3



(f)

figure 19. b removing a node from the front of a list

The steps required to performed are as follows
p=list
list=next (p);
x=info;
Free node (p);
In this process, p is used as auxiliary variable that points to the first element on the list .After x has been set to the desired value and list is made to new first element, p is made free since it becomes useless.

The four primitive operations of linked list required to insert and delete a node from a list are as follows:

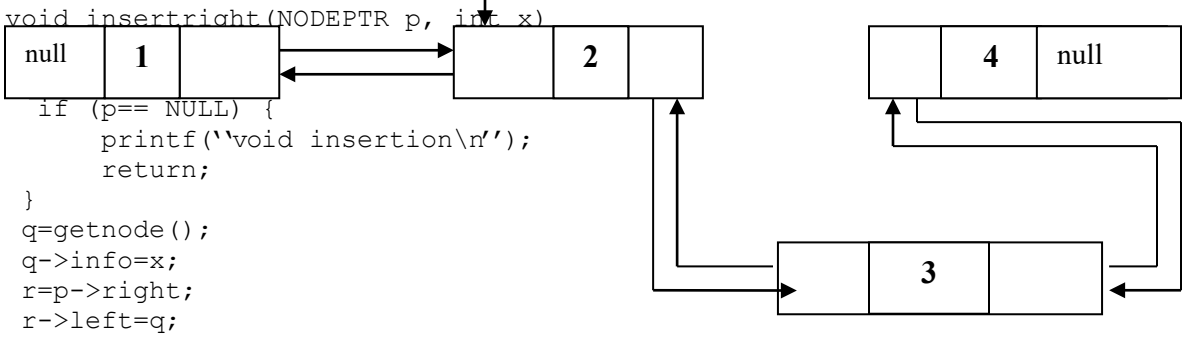
```
struct node{
   int info;
   struct node *next;
};
typedef struct node *NODEPTR;

NODEPTR getnode(){
   NODEPTR p;
   p=(NODEPTR)malloc(sizeof(struct node));
   return(p);
}
void freenode(NODEPTR p){
   free(p);
}
NODEPTR insert(NODEPTR list,int x){
   NODEPTR p;
   if(p!=NULL){
          p=getnode();
          p->info=x;
          p->next=p;
          list=p;
   }
   return p;
}

NODEPTR remove(NODEPTR list){
   NODEPTR p;
   int x;
   p=list;
   list=p->next;
   x=p->info;
   printf("\nRemoved item=%d",x);
```

```
  freenode(p);
  return list;
}
```

20. Write down the steps involved in inserting and deleting of a node in a doubly linked list.

A doubly linked list is a dynamic data structure consisting of items called nodes. Each node in a doubly linked list consists of three fields; an info field that contains information stored in the node, and left and right fields that contain pointers to the nodes on the either side.
Inserting and deleting a node in a doubly linked list involves a series of pointers rearrangements. While inserting a node to the right, pointers rearrangements takes place as shown in the figure 20. a

The routine that inserts a node with information field x to the right of node (p) in a doubly linked list is as follows:




```
void insertright(NODEPTR p, int x)
{
  if (p== NULL) {
       printf(''void insertion\n'');
       return;
  }
  q=getnode();
  q->info=x;
  r=p->right;
  r->left=q;
  q->right=r;
  q->left=p;
  p->right=q;
  return;
}/*end insertion;
```

Figure: 20.a

A routine insertleft to insert a node with information field x to the left of node(p) in a doubly linked list is as follows:

```
void insertleft(NODEPTR p, int x)
{
  NODEPTR q,r;
  if (p== NULL) {
       printf(''void insertion\n'');
       return;
  }
  q=getnode();
  q->info=x;
  r=p->left;
  r->right=q;
  q->left=r;
  q->right =p;
  p->left=q;
  return;
}/*end insertion;/
```

Now the routine that deletes the node pointed to by p from a doubly linked list and stores the contents in x is as follows:

```
void delete(NODEPTR p, int *px)
{
```

```
NODEPTR q,r;
If (p= = NULL)
   {
     printf(''void deletion''\n'');
     return;
  }/*end if*/
*px=p->info;
 q=p->left;
 r=p->right;
 q->right=r;
 r-> left=q;
 freenode(p);
 return;
}
```

21.i. Write the merits and demerits of contiguous list and linked list.

The word contiguous means in contact, touching, adjoining. For e.g. the entries in an array are contiguous. Firstly, let us cite to assess some relative advantages of linked and of contiguous implementation of lists.
The foremost advantage of dynamic storage for linked lists is flexibility. Overflow is no problem until the computer memory is actually exhausted. Especially when the individual structures are quite large, it may be difficult to determine the amount of contiguous static storage that might be needed for the required arrays, while keeping free for other needs.
Changes, especially insertions and deletions can be made in the middle of a linked list more easily than in the middle of a contiguous list. Even queues are easier to handle in linked storage. If the structures are large, then it is much quicker to change the values of a few pointers than to copy the structures themselves from one location to another.
The first drawback of linked lists is that the links themselves take space, space that might otherwise be needed for additional data. In most systems, a pointer requires the same amount of storage (one word) as does an integer. Thus a list of integers will require double the space in linked storage that it would require in contiguous storage. On the other hand, in many practical applications, the nodes in the list are quite large, with data fields taking hundreds of words altogether. If each node contains 100 words of data, then using linked storage will increase the memory requirement by only one percent, an insignificant amount. In fact, if extra space is allocated to arrays holding contiguous lists to allow for additional insertions, then linked storage will probably require less space altogether. If each item takes 100 words, then contiguous storage will save space only if all the arrays can be filled to more than 99 percent of capacity.

The major drawback of linked list is that they are not suited to random access. With contiguous storage, the program can refer to any position within a list as quickly as to any other position. With a linked list, it may be necessary to traverse a long path to reach the desired node.

Finally, access to a node in linked storage may take slightly more computer time, since it is necessary, first, to obtain the pointer and then go to the address. This consideration, however, is usually no importance. Similarly, we may find that writing functions to manipulate linked lists takes a bit more programming effort.

Therefore, we can conclude that contiguous storage is generally preferable when the structures are individually very small, when few insertions or deletions need to be made in the middle of a list, and when random access is important, linked storage proves superior when the structures are large and flexibility is needed in inserting, deleting, and rearranging the nodes.

21 .ii. Write the merits and demerits of contiguous list and linked list.

The word *contiguous* means in contact, touching, adjoining. For e.g. the entries in an array are contiguous. Firstly, let us cite to assess some relative advantages of linked and of contiguous implementation of lists.

The foremost advantage of dynamic storage for linked lists is flexibility. Overflow is no problem until the computer memory is actually exhausted. Especially when the individual structures are quite large, it may be difficult to determine the amount of contiguous static storage that might be needed for the required arrays, while keeping free for other needs.
Changes, especially insertions and deletions can be made in the middle of a linked list more easily than in the middle of a contiguous list. Even queues are easier to handle in linked storage. If the structures are large, then it is much quicker to change the values of a few pointers than to copy the structures themselves from one location to another.
The first drawback of linked lists is that the links themselves take space, space that might otherwise be needed for additional data. In most systems, a pointer requires the same amount of storage (one word) as does an integer. Thus a list of integers will require double the space in linked storage that it would require in contiguous storage. On the other hand, in many practical applications, the nodes in the list are quite large, with data fields taking hundreds of words altogether. If each node contains 100 words of data, then using linked storage will increase the memory requirement by only one percent, an insignificant amount. In fact, if extra space is allocated to arrays holding contiguous lists to allow for additional insertions, then linked storage will probably require less space altogether. If each item takes 100 words, then contiguous storage will save space only if all the arrays can be filled to more than 99 percent of capacity.

The major drawback of linked list is that they are not suited to random access. With contiguous storage, the program can refer to any position within a list as quickly as to any other position. With a linked list, it may be necessary to traverse a long path to reach the desired node.
Finally, access to a node in linked storage may take slightly more computer time, since it is necessary, first, to obtain the pointer and then go to the address. This consideration, however, is usually no importance. Similarly, we may find that writing functions to manipulate linked lists takes a bit more programming effort.
Therefore, we can conclude that contiguous storage is generally preferable when the structures are individually very small, when few insertions or deletions need to be made in the middle of a list, and when random access is important, linked storage proves superior when the structures are large and flexibility is needed in inserting, deleting, and rearranging the nodes.

22.i. What is the role of Header node in linked list? Explain.

Header node is an extra node placed at the front of a list. Alternately, the header for a linked list is a pointer variable that locates the beginning of the list. The header will usually be a static variable, and by using its value we can arrive at the first (dynamic) node of the list. The header is also sometimes called the ***base*** or the ***anchor*** of the list.
When execution of the program starts we shall wish to initialize the linked list to be empty; with a header pointer. The header is a static pointer; so it exists when the program begins, and to set its value to indicate that its list is empty, we need only the assignment:

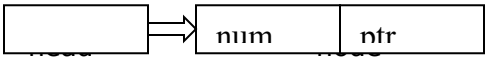
```
header=NULL;
```

We consider head as an external pointer. This helps in creating and accessing other nodes in the linked list. Consider the following structure definition and head creation.

```
struct node
        {
                int num;
                struct node *ptr;
        }
        typedef struct node NODE;              /*type definition making it abstract    data type */
        NODE *head;             /*pointer to the node of linked list*/
        head=(NODE*)malloc(sizeof(NODE));
```

When the statement
head=(NODE*)malloc(sizeof(NODE));

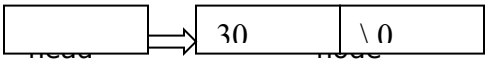
is executed, a block of memory sufficient to store the NODE is allocated and assigns head as the starting address of the NODE(Now, head is an external pointer). This activity can be pictorially shown as follows;



Now, we can assign values to the respective fields of NODE.

```
head ->number=30;   /*data fields contains value 30*/
head ->ptr='\0'; /*Null pointer assignment */
```

Thus, new vision of the NODE would be like this:



22.ii.What is the role of Header node in linked list? Explain.

Header node is an extra node at the front of a list which does not represent any item in the list. Info portion of the header node is used to keep global information about the entire list (such as the number of nodes in it, or a pointer to its last node). Also, the number of items in the list may be obtained directly from the header node without traversing the entire list.

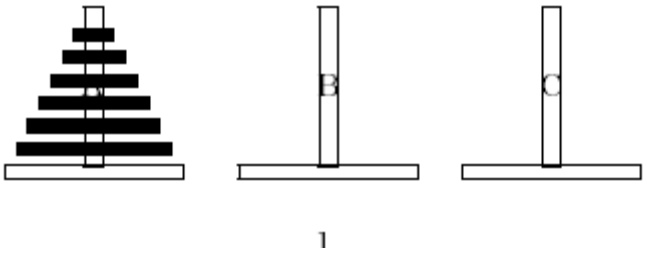
Another use of info portion of the list header (header node) is as a pointer to a "current" node in the list during the traversal process. It eliminates the need for an external pointer during traversal.

23. i. What is Recursion? Explain with Towers Of Hanoi example.

Recursion is one of the most powerful programming technique. It can also be defined as a method which includes an invocation of itself.

Let us clarify with the problem of Towers Of Hanoi.

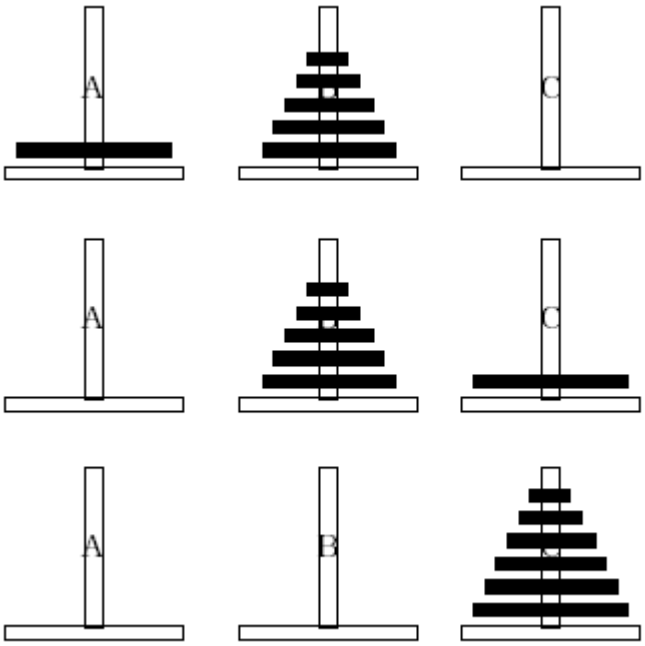
The Towers of Hanoi problem was invented by the French mathematician Edouard Lucas 1883. The problem is to transfer a tower of discs from one
peg to another moving one disc at a time and never placing a disc on top of a smaller one Here is the starting state of the problem, with all of the discs in order
on peg A.



1

The problem is solved by moving every disc except the largest one onto the spare peg; then move the largest across; then move the others back on top. In the case where we are moving six discs from peg A to peg C we must reach an intermediate stage where we have only the largest disc on peg A and we have five discs on peg B. At this point we can move the largest disc on to peg C and we then move the five discs on top of it(in many steps, observing the rules that we can move only

one disc at time and that we can never place a larger disc on top of a smaller



one.
Of course, moving every disc but the largest is nothing other than a smaller instance of the problem so the solution naturally lends itself to a recursive description.

24. How Recursive algorithm makes program effective? Write the merits and demerits of recursion in Programming.

The recursive algorithm makes the program effective in a sense that the solution of a certain real world problem can be visualized naturally by the recursive technique rather than iterative technique. As an example 'The Towers of Hanoi' problem can be solved easily by the recursive technique because the problem can be visualized more technically by the recursion. The iterative solution of this problem can be very much complex and can not be error free. If we use the iterative solution then there will be a burden to the compiler since we have to deal with the simulation of the problem in lots of codes. The above-mentioned point is also a merit of the recursive algorithm.

The demerits of the recursive definition can be explained by its use in finding the factorial of a number and a Fibonacci of a number. The recursive definition of a factorial of a number can be done sequentially by a for loop and doesn't need a recursive function due to which the load to the compiler is reduced a lot. Factorial, whose non recursive version does not need a stack, and calculation of Fibonacci numbers, which contains an unnecessary second recursive call (and does not need a stack either), are examples where recursion should be avoided in a practical implementation. The Fibonacci sequence of the number can also be determined easily without using the recursive technique by the use of simple for loop. If that is done by the recursively then there will be the burden to the compiler.

The ideas and transformations that we have put forward in presenting the factorial function and in the Towers of Hanoi problem can be applied to more complex problems whose non-recursive solution is not readily apparent. The extent to which a recursive solution (actual or simulated) can be transformed into a direct solution depends on the particular problem and ingenuity of the programmer.

25. i. Write down the recursive definition of algebraic Expression Multiplication.

1. An expression is a term followed by an Asterisk sign followed by a term, or a term alone.
2. A factor is either a letter or an expression enclosed in parentheses.

An expression is defined in terms of a term, a term in terms of factor, and a factor in terms of an expression. Similarly, a factor is defined as in terms of an expression, which is defined in terms of term, which is defined in terms of a factor. Thus an entire set of definitions forms a recursive chain. Let us take an example: Since A is an expression, (A) is a factor and therefore a term as well as an expression. A*B is an example of expression that is neither a term nor a factor. (A*B), however is all three. A+B is a term and therefore an expression, but it is not a factor. A+B*C is an expression that is neither a term nor a factor. A+ (B*C) is a term and an expression but not a factor.

25. ii. Write down the recursive definitions of Algebraic expression multiplication?

1. An expression is a term followed by an Asterisk sign followed by a term, or a term alone.
2. A factor is either a letter or an expression enclosed in parentheses.

An expression is defined in terms of a term, a term in terms of factor, and a factor in terms of an expression. Similarly , a factor is defined as in terms of an expression, which is defined in terms of term, which is defined in terms of a factor. Thus an entire set of definitions forms a recursive chain. Let us take an example: Since A is an expression, (A) is a factor and therefore a term as well as an expression. A*B is an example of expression that is neither a term nor a factor. (A*B), however is all three. A+B is a term and therefore an expression, but it is not a factor. A+B*C is an expression that is neither a term nor a factor. A+ (B*C) is a term and an expression but not a factor.

6. **26. i. Write down the function to compute the nth Fibonacci number using recursion. Explain the working mechanism with the help of recursion.**

The function to compute the nth Fibonacci number using recursion is as follows.

```
int fib (int n)
{
        int x,y;
        if (n<=1)
                return (n);
        x=fib(n-1);
        y=fib(n-2);
return (x+y);
}
```

A Fibonacci sequence is the sequence of integers 0,1,1,2,3,5….. in which each element is the sum of the preceding elements. Let us find the action of the function written above in computing the sixth Fibonacci number. When the program is first called the variable n,x,y are allocated and n is set 6. Since n>1, n-1 is evaluated and fib is called recursively . A new set of n,x,y are allocated and n is set to 5.this process continues until each successive value of n being one less than its predecessor, until fib return 1 to its caller, so that the fifth allocation of x is set to 1.The next sequential statement y=fib(n-2) is then executed. The value of n that is used is the most recently allocated one which is 2.Thus we again call on fib with an argument 0. The value of 0is immediately returned, so that y in fib(2) is set to 0. We should note that each recursive call results in a return to the point of call, so that the call fib(1) returns to the assignment to x, and the call of fib(0) returns to the assignment to y. The next statement to be executed in fib(2) is the statement that returns x+y=1+0=1 to the statement that calls fib(2) in the generation of

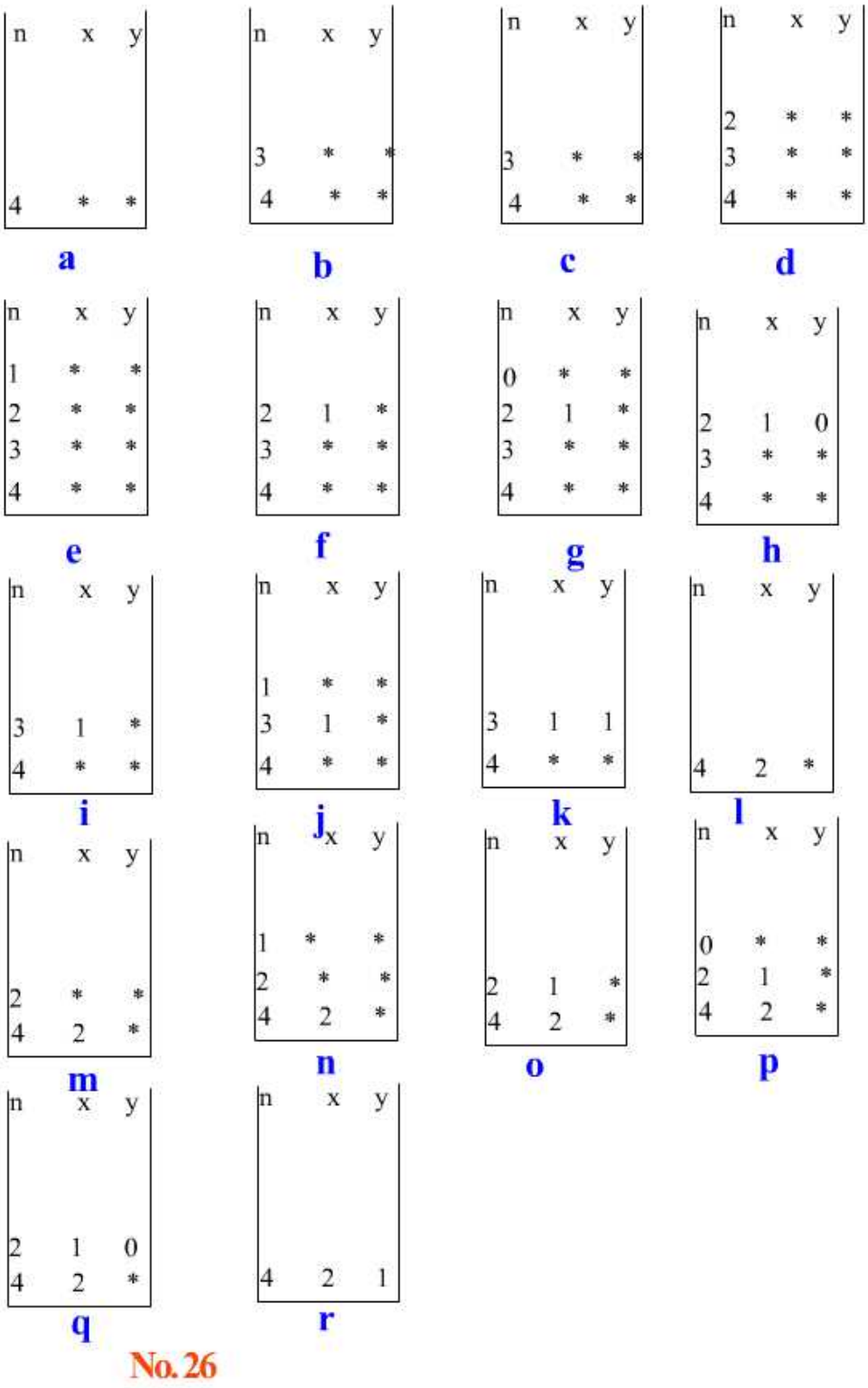
the function calculating fib(3).This is the assignment t x, so that x in fib(3)is given the value fib(2)=1.This process of calling and pushing and returning and popping continues until finally the routine returns for the last time to the main program with the value 8.

26.ii. Write a function to compute the nth Fibonacci number using recursion. Explain the working mechanism with the help of recursion stack.

The required function to compute the nth Fibonacci number is as follows:

```
int fib(int n){
   int x,y;
 if(n<=1)
    Return (n);
x=fib(n-1);
y=fib(n-2);
return (x+y);
}
```

When the program first calls the fib() function, the variable n, x and y are allocated, and n is set to 4. It checks the condition if(n<=1), in the first case the condition is not fulfilled and the function fib() is called recursively. Each time the function is called the new set of variables n, x and y is being allocated. The process continuous until fib() is called with n equal to 1. this process can be explained in detail by the following illustration of recursion stack.



The above illustration shows the snapshots of the stack formed during the call to the fourth Fibonacci number sequence. In the first call the value of n is 4. The function fib(n) is called until

the value of n is not equal or less than 1. In this case the value of n is called until it becomes 1. The figure *'e'* shows this moment of stack. When the value of n becomes 1 then 1 is returned to the variable x as shown in fig *'f'*. Now the function fib(n-2) is called for n=2 i.e. fib(0) is called and 0 is returned to y as shown in fig. *'h'*. Now the return statement comes into action so that the sum '0+1' is returned to x as shown in fig.*'i'*. In this case the value of n is 3 that cause the fib(1) to execute as shown in fig. *'j'* and 1 is returned to y. This process goes on up to fig. *'r'.* This figure is the last stage of the stack from where the new variables will not be formed. The last result '2+1' i.e. 3 is returned to the function. In this way the pushing and popping action continuous in the stack while computing the n^{th} Fibonacci number.

27i Explain the advantage of recursion in series of function calls.

The non-recursive program executes more efficiently in terms of time and space than a recursive program. This is because the overhead involved in entering and exiting a block is avoided in non-recursive version. But the recursive function serves the most suitable for certain problems such as Towers of Hanoi, conversion of postfix to prefix etc.

In some problems the recursive solution is the most natural and logical way of finding the solution. We have also used the non-recursive technique in converting the prefix to postfix by using the stack. This causes the overload to the compiler and the program is usually not error free. By using the recursive technique the problem of using the stack can be solved. There are certain problems of whose the non-recursive solution is very hard to design leading to various errors. Thus it depends on the problem and the ingenuity of the programmer to make a program a recursive or non-recursive.

2**7iiExplain the advantage of recursion in series of function calls.**

Recursion of function refers to chain calling of the same function repeatedly. Simply, a function that makes a call to itself is said to be *recursive*. Many problems in mathematics and computer science are naturally recursive. This just means that a problem can be broken down into smaller instances of itself, solved, then put back together

The non-recursive program executes more efficiently in terms of time and space than a recursive program. This is because the overhead involved in entering and exiting a block is avoided in non-recursive version. But the recursive function serves the most suitable for certain problems such as Towers of Hanoi, conversion of postfix to prefix etc.

In some problems the recursive solution is the most natural and logical way of finding the solution. We have also used the non-recursive technique in converting the prefix to postfix by using the stack. This causes the overload to the compiler and the program is usually not error free. By using the recursive technique the problem of using the stack can be solved. There are certain problems of whose the non-recursive solution is very hard to design leading to various errors. Thus it depends on the problem and the ingenuity of the programmer to make a program a recursive or non-recursive.

A recursive function must have two parts:

- One or more *base cases*. This solves a "trivial" instance of the problem, like finding the factorial of zero.
- The recursive case. This is where the function breaks up the problem, solves the parts recursively, and then comes up with an answer to the whole problem.

This often leads to code that is more brief and more clear. For example, here is an *iterative* version of factorial ("iterative" in this context means "not recursive," i.e. using explicit iteration to do many things):

```
int factorial (int n) {
        int     product, i;

        product = 1;
        for (i=2; i<=n; i++)
                product = product * i;
        return product;
}
```

Here is a recursive version of factorial:

```
int factorial (int n) {
        if (n == 0) return 1;
        return n * factorial (n-1);
}
```

The recursive version is much cleaner and retains the *functional* definition of factorial.

28.i. Write down the recursive solution to translate an expression from prefix to postfix?

Recursion solution is the most direct and elegant method to translate from prefix to postfix. Briefly, prefix and postfix method are methods of writing mathematical expression without parentheses. In prefix notation each operator immediately precedes its operands. In postfix notation each operator immediately follows its operands . The most convenient way to define postfix and prefix by using recursion .

A conversion routine of prefix to postfix is as follows:

```
  Void convert (char prefix[ ] ,char postfix[ ])
{
        char opnd1[MAXLENGTH], opnd2[MAXLENGTH] ;
         char post1[MAXLENGTH], post2 [MAXLENGTH] ;
         char temp [MAXLENGTH] ;
          char op[1] ;
        int length;
        int I, j, m ,n ;

                if((length = strlen (prefix) ) ==1){
                        if (isalpha ( prefix[0] )){
                                postfix[0]=prefix[0];
                                postfix[1]=' \0 ' ;
                                return ;
                        }
                        printf(" \n illegal prefix string " ) ;
                        exit( );
                 }
op[ 0] = prefix[ 0];
op[1]= ` \0 `;
substr(prefix, m+1, length-1, temp );
m = find ( temp);
if(op[ 0]!=' +' &&  op[ 0] != ` –` && op[ 0] != ` * ' && op[0]!= ` / ' )
|| (m==0) || (n==0)          || ( m+n+1 != length)){
        printf( " \n illegal  prefix  string ");
```

```
        exit (1) ;
}
substr ( prefix , 1, m ,opnd1);
substr ( prefix ,m+1, n ,opnd2);
convert ( opnd1, post1);
convert ( post1, post2);
strcat ( opnd1,post2);
strcat ( post1,op);
substr ( post1, 0, length, postfix);
}
```

Implementation of find function:

```
 Int find(char str[ ] )
{
        char temp [MAXLENGTH] ;
        int length;
        int I, j, m ,n ;
        if((length = strlen (str) ) == 0 ) {
                return (0) ;
        if (isalpha ( str [0] ) !=0)
                return (1) ;
         if( strlen (str) ) <2  )
                return (0) ;
        substr ( str , 1, length-1 ,temp );
        m = find(temp) ;


if(m==0 || strlen (str)==m )
        return (0) ;
substr ( str , m+1, length –m - 1 ,temp );
        n = find ( temp) ;
if(n==0)
        return (0) ;
return ( m + n + 1) ;
}/* end find*/
```

28.ii. Write the recursive solution for to convert the prefix to postfix.

Prefix and postfix notation are methods of writing mathematical expressions without parentheses. In prefix notation each operator immediately precedes its operands. In postfix notation each operator immediately follows its operands.

The most convenient way to define postfix and prefix is by using recursive. If a prefix expression consists of only a single variable, that expression is its own postfix equivalent. That is, an expression such as A is valid as both a prefix and a postfix expression. Every prefix string longer than a single variable contains an operator, a first operand, and a second operand. Assume that we are able to identify the first and second operands, which are necessarily shorter than the original string. We can then convert the long prefix string to postfix by first converting the first operand to postfix, then converting the second operand to postfix and appending it to the end of the first converted operand, and finally appending the initial operator to the end of the resultant string. Thus we have developed a recursive algorithm for converting a prefix string to postfix, with the single provision that we must specify a method for identifying the operands in a prefix expression. We can summarize our algorithm as follows:

If the prefix string is a single variable, it is its own postfix equivalent.

Let op be the first operator of the prefix string.

Find the first operand, open1, of the string. Convert it to postfix and call it post1.

Find the second operand, open2, of the string. Convert it to postfix and call it post2.

Concatenate post1,post2, and op.

For the solution we wish to write a procedure convert that accepts a character string. This string represents a prefix expression in which all variables are single letters and the allowable operators are '+','-','*', and '/'. The procedure produces a string that is the postfix equivalent of the prefix parameter.

```
   Void convert (char prefix[], char postfix[])
   {
          char open1[MAXLENGTH],open2[MAXLENGTH];
      char post1[MAXLENGTH],post2[MAXLENGTH];
      char temp[MAXLENGTH];
      char op[1];
      char length;
      int I,j,m,n;

           if ((length=strlen(prefix))==1){

            if (isalpha(prefix[0]))}
                          postfix[0]=prefix[0];
            postfix[1]='\0';
return;
}
printf(''\nillegal prefix string'');
exit{1};
}
op[0]=prefix[0];
op[1]='\0';
substr(prefix, 1, length-1,temp);
m=find(temp);
substr(prefix, m+1, length-m-1, temp);
                        n=find(temp);
      if((op[0]!='+'&&op[0]                                                    !='-
'&&op[0]!='*'&&op[0]!='\')!!(m==0)!!(n==0)!!(m+n+1!=length)){
printf(''\n illegal prefix string'');
exit(1);
}
substr(prefix, 1, m open1);
substr(prefix, m+1,n, open2);
convert(opnd1, post1);
convert(opnd2, post2);
strcat(post1, post2);s
strcat(post1, op);
substr(post1,o,length,postfix);
}
```

The function convert also calls the library function isalpha , which determines if its parameter is a letter. And also we are using the function the find which accepts a string and returns an integer that is the length of the longest prefix expression contained within the input string that starts at the beginning of that string. We now incorporate this method into the function find .

```
 int find(char str[])
{
    char temp[MAXLENGTH];
    int length;
int i,j,m,n;
  if((length=strlen(str))==0)
return {0];
 if(isalpha(str[0]!=0)
return (1);
  if(strlen(str)<2)
return (0);
```

```
substr(str,1,length-1,temp);
m=find(temp);
if(m==0!!strlen(str)==m)
return (0);
substr,m+1,length-m-1;,temp);
n=find(temp);
if(n==0)
return (0);
return(m+n+1);
}
```

In this way the prefix expression can be converted into the postfix by using the concept of recursive .

29. i. Write down the Recursive definition of Tree.

Tree or simply the binary tree is a finite set of elements that is either or is portioned 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 or it contains the elements. Each element of a binary tree is called a node of the tree.

The right or left sub trees are also called the right and left child and the root node is called the father node. The left and right child nodes may also have the sub nodes, which are again the left and right child of the current node. These node's father is the child of the first root node. Also this is the grandchild of the root node. In this way the Tree can be defined recursively where the tree can be filled using recursive function. For each filling there will be a father node and it contains the two child left and right nodes, these nodes are filled with the elements .For the further filling the tree these child nodes will be the father by recursively and it contains the its own children.

33.i. What do you mean by traversing a tree? What are the different ways of traversing? Explain.

Traversing is one of the key operational features of a tree. In simple words, traversing means moving through all the nodes of the tree, visiting them one by one in turn. The visiting of the node depends from application to application. For lists, the nodes came in a natural order from list to last, and traversal followed the same order. For trees, however, there are many different orders in which we could traverse all the nodes. When we write an algorithm to traverse a binary tree, we shall almost always wish to proceed so that the same rules are applied at each node, and we thereby follow a general pattern.

At a given node, then, there are three tasks we shall wish to do in some order. We shall visit the node itself; we shall traverse its left sub tree; and we shall traverse its right sub tree. The key distinction in traversal orders is to decide if we are to visit the node itself before traversing either sub tree, between the suttees, or after traversing both sub trees.

If we name the tasks of visiting a node v, traversing the left sub tree L, and traversing the right sub tree R, then there are six ways to arrange them:

V L R L V R L R V V R L R V L R L V

By standard convention, these six are reduced to three by permitting only the ways in which the left sub tree is traversed before the right. The three ways with left before right are given special names that we shall use from now on:

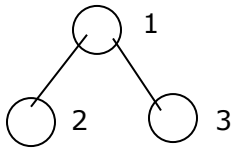
V L R	L V R	L R V
Preorder	In order	Post order

Therefore the different ways of traversing a tree are Preorder, In order and Post order.

These three names are chosen according to the steps at which the given node is visited. With preorder traversal, the node is visited before the sub trees; with in order traversal, it is visited

between them; and with post order traversal, the root is visited after both of the sub trees. In order traversal is also sometimes called symmetric order and post order traversal was once called endorser.

Let's consider the following binary tree:



Under preorder traversal, the root, labeled 1, is visited first. Then the traversal moves to the left sub tree. The left sub tree contains only the node labeled 2, and it is visited second. Then preorder traversal moves to the right sub tree of the root, finally visiting the node labeled 3. Thus preorder traversal visits the nodes in the order 1, 2, 3.

Before the root is visited under in order traversal, we must traverse its sub tree. Hence the node labeled 2 is visited first. This is the only node in the left sub tree of the root, so the traversal moves to the root, labeled 1, next, and finally to the right sub tree. Thus in order traversal visits the nodes in the order 2, 1, 3.

With post order traversal, we must traverse both the left and right sub trees before visiting the root. We first go to the left sub tree, which contains only the node labeled 2, and it is visited first. Next we traverse the right sub tree, visiting the node 3, and finally, we visit the root, labeled 1. Thus post order traversal visits the nodes in the order 2, 3, 1.

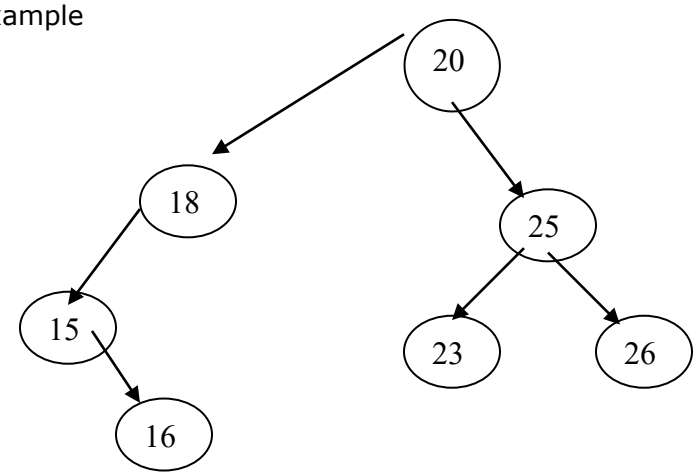
34.i. Explain how you can traverse in a tree to do sorting in descending order?

At first we are dealing with the binary tree. A binary tree is a finite set of elements that is either empty partitioned into three disjoint subsets. The first subset contains single elements called root of the tree. The two subsets are themselves binary trees, called the left and right sub tree of the original tree. A left or righ tree can be empty. Each elements of a binary tree is called a node of the tree.

The order in which the node of a linear list is visited in a traversal is clearly from first to last. How there is no such "natural" linear order for the nodes of a tree. If we traverse the tree as mentioned as below we can sort in descending order. For this purpose

1: Traverse the right sub tree
2: Visit the root
3: traverse the left sub tree

for example



If a binary search tree is traversed in the above mentioned order and the content of each node are printed as the node is visited, the numbers are printed in descending order. Following the above steps

26,25,23,20,18,16,15

Hence we can sort the tree in descending order.

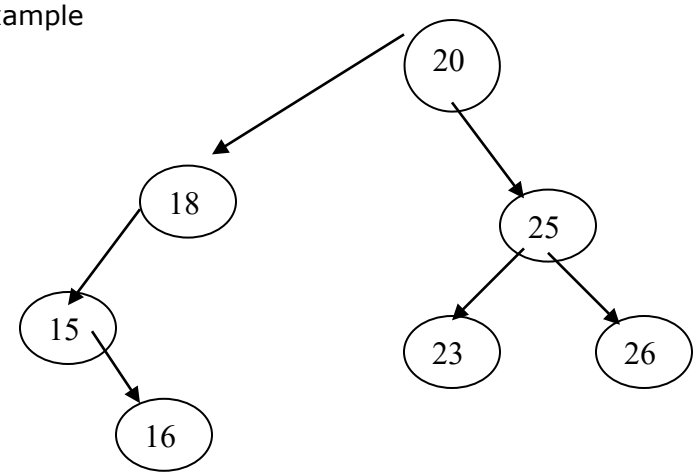
34. Explain how you can traverse in a tree to do sorting in descending order?

At first we are dealing with the binary tree. 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 elements called root of the tree. The o two subsets are themselves binary trees, called the left and right sub tree of the original tree. A left or right su tree can be empty. Each elements of a binary tree is called a node of the tree.

The order in which the node of a linear list are visited in a traversal is clearly from first to last. However, there such "natural" linear order for the nodes of a tree. If we traverse the tree as mentioned as below then we can in descending order. For this purpose

1: Traverse the right sub tree
2: Visit the root
3: traverse the left sub tree

for example



If a binary search tree is traversed in the above mentioned order and the content of each node are printed as the node is visited, the numbers are printed in descending order. Following the above steps

26,25,23,20,18,16,15

Hence we can sort the tree in descending order.

35. What is the application of binary tree?

Binary Trees are the most basic non linear structure in Computer Science. Their wide range of applicability derives from their fundamentally hierarchical nature, a property induced by their recursive definition. We can define a binary tree formally as a finite set of nodes that either is empty, or consists of a root and the elements of two disjoint binary trees, called the left and right sub-trees of the root or parent.

Binary Trees are used in many diverse applications. Some of them are mentioned here:

1. When two-way decisions must be made at each point in a process such as finding all duplicates in a list of numbers.
2. In Huffman Compression Algorithm where each time a 1 or 0 needs to be inserted going right and left of the root node respectively again involving decision making.

3. Sorting applications which requires a total ordering on key values to define a relevant notion of "less than or equal to" per application.
4. Producing efficient search mechanisms manipulating the structure of the tree.

In shared databases that need to maximize concurrency for good performance

36. How do you insert and delete nodes in binary tree?

A node is inserted or deleted in binary tree as described in the following section:

Insertion in Binary Tree:

The first number is the list is placed in the node that is established as the root of a binary tree with the empty left and right sub-trees.

Each successive number is compared to the number in the root. If it is matched we simply place that number in node as left child of the root node.

If it does not match and the sub-tree is empty, the number is placed into a new node as left child for smaller number and as right for larger one with due respect to the value at root. If again the sub-tree is not empty, and the value is smaller, then the value is compared to the left child of the root node as if it is the root node now. Same thing happens with larger value on right side of the node. This process repeats unless the value gets a proper node.

Algorithm:

```
/*
Helper function that allocates a new node
With the given data and NULL left and right
Pointers.
*/
struct node* New Node(int data) {
  struct node* node = new(struct node); // "new" is like "malloc"
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}
/*
Give a binary search tree and a number, inserts a new node
with the given number in the correct place in the tree.
Returns the new root pointer which the caller should
then use (the standard trick to avoid using reference
parameters).
*/
struct node* insert(struct node* node, int data) {

            // 1. If the tree is empty, return a new, single node
  if (node == NULL) {
    return(new Node(data));
  }
  else {
              // 2. Otherwise, recur down the tree
    if (data <= node->data) node->left = insert(node->left, data);
      else node->right = insert(node->right, data);
    return(node); // return the (unchanged) node pointer
  }
}
```

The shape of a binary tree depends very much on the order that the nodes are inserted. In particular, if the nodes are inserted in increasing order (1, 2, 3, 4), the tree nodes just grow to the right leading to a linked list shape where all the left pointers are NULL. A similar thing happens if the nodes are inserted in decreasing order (4, 3, 2, 1).

Deletion of a node in a binary tree:

In deletion we assume we have a pointer p to the item that we wish to delete. The pointer may be may be of the node that has:

1) no children
2) left child
3) right child but no left child.

1) If there are no children for the node that is to be deleted, it can be replaced by NULL. (see fig 1)
2) The node pointed by P ie. the node to be deleted has a l-child. In this case, we have a non NULL left sub-tree of the node to be deleted, so here we need to find the greatest node in that left sub tree. If the node pointed by left sub-tree of P has no right child, then the greatest in the left sub-tree of P is the left link itself otherwise we must follow the right branch leading from left link of P deeply as possible into the tree. (see fig 2)
3) The node pointed by P, which is to be deleted has r-child but no l-child. In this case, it is simpler and that the r-child can replace the node that is to be deleted. (see fig 3)

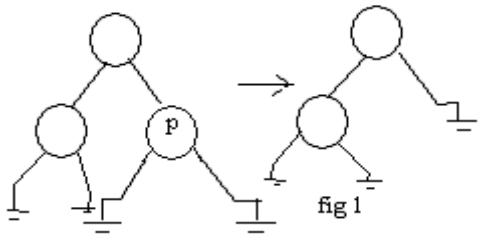


fig 1

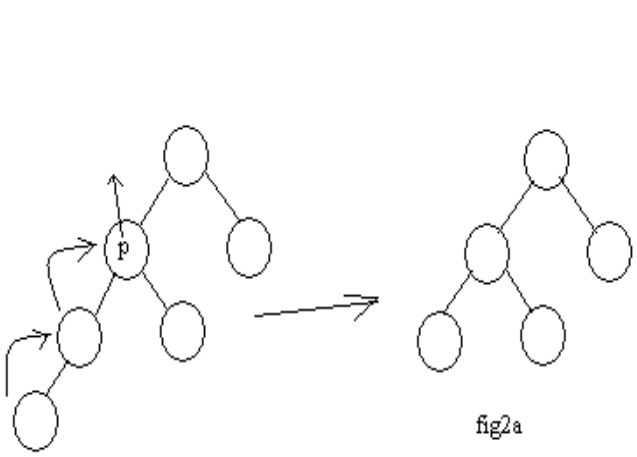


fig2a

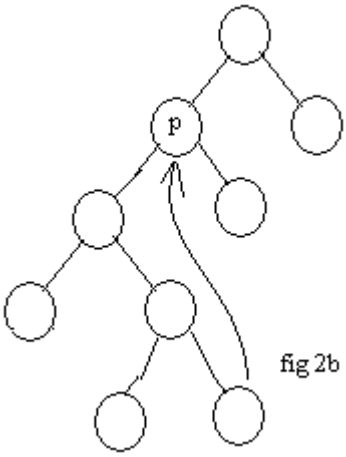
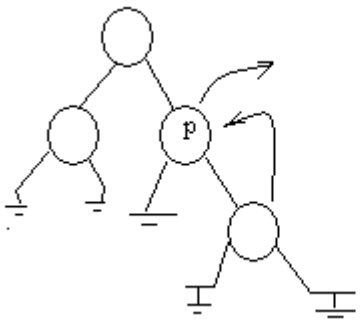


fig 2b



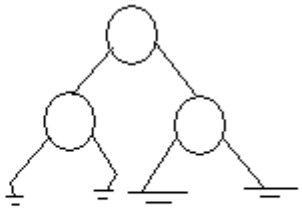


fig 3

37.i. Define AVL Balanced Tree. Where is it used?

In a full binary search tree heights of the left and right sub-tree of any node are equal. Such trees are ideal for efficient searching because the height of such a tree with 'n' node is of O (logn). But when the insertion and deletion operations are performed randomly on a binary search tree, the binary tree often turns out to be far from ideal. A close approximation to an ideal binary search tree is achievable if it can be ensured that the difference between the heights of the left and right sub-trees of any node in the tree is at most 1. Such a binary tree in which the heights of two sub-tress of every node never differ by more than 1 is AVL tree named after Adelson Velskii Landis.

Where are AVL trees used?

AVL balanced tress may be used for efficient implementation of Priority Queues. A priority queue implemented using balanced tree can perform any sequence of n insertions and minimum deletion in O(nlogn) steps.

37ii.Define AVL Balance tree. Where it is used?

The height of a Binary tree is the maximum level of its leaves (This is also sometimes known as the depth of the tree).For convenience, the height of null tree is defined as -1. A balanced Binary tree (sometimes called an AVL Tree) is a binary tree in which the heights of the two sub trees of every node never differ by more then one. The balanced of a node in a binary tree is define as the height of its left sub tree minus the height of its right sub tree .Fig. 1 illustrates a balanced binary tree .Each node in a balanced binary tree has a balance of 1, -1or 0, depending on whether the height of its left sub tree is greater than, less than, or equal the height of its right sub tree. The balance of each node is indicated in fig. 1.

Balanced Tree may also be used for efficient implementation of priority queues.

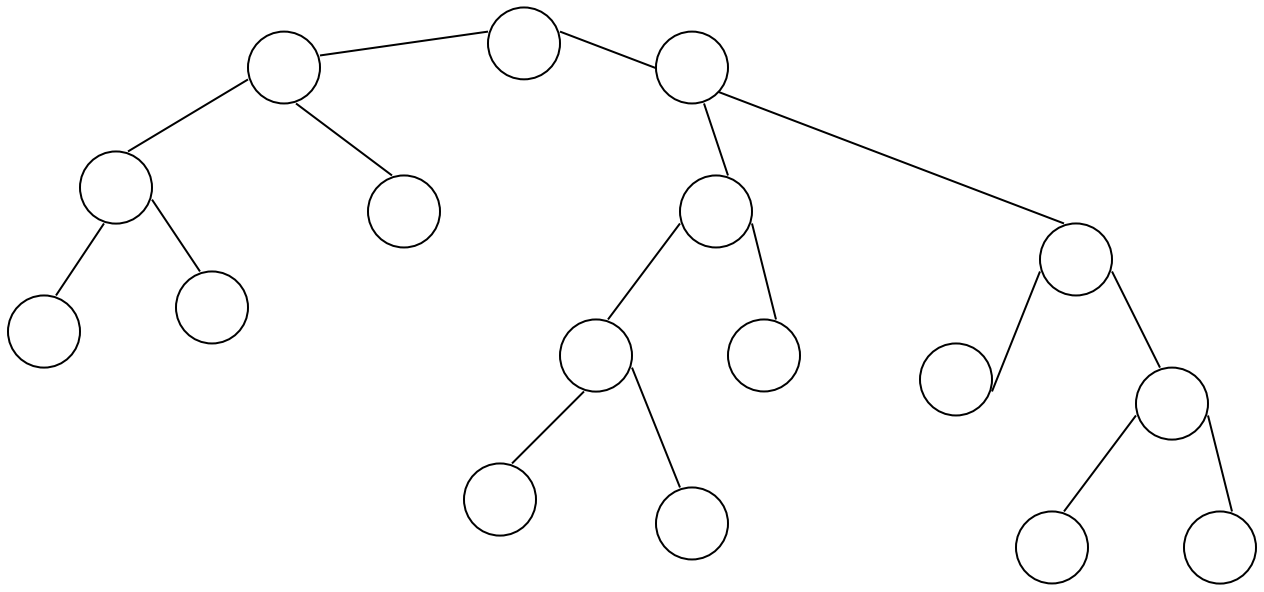


FIG 1

38**.i. What are the steps to make an unbalanced tree to a balanced one?**

There is two process of balancing an unbalanced tree:

1: Right Rotation

2: left Rotation

Let us take an example of the balanced tree and try insertion for unbalancing the balanced tree.We insert as in fig below:

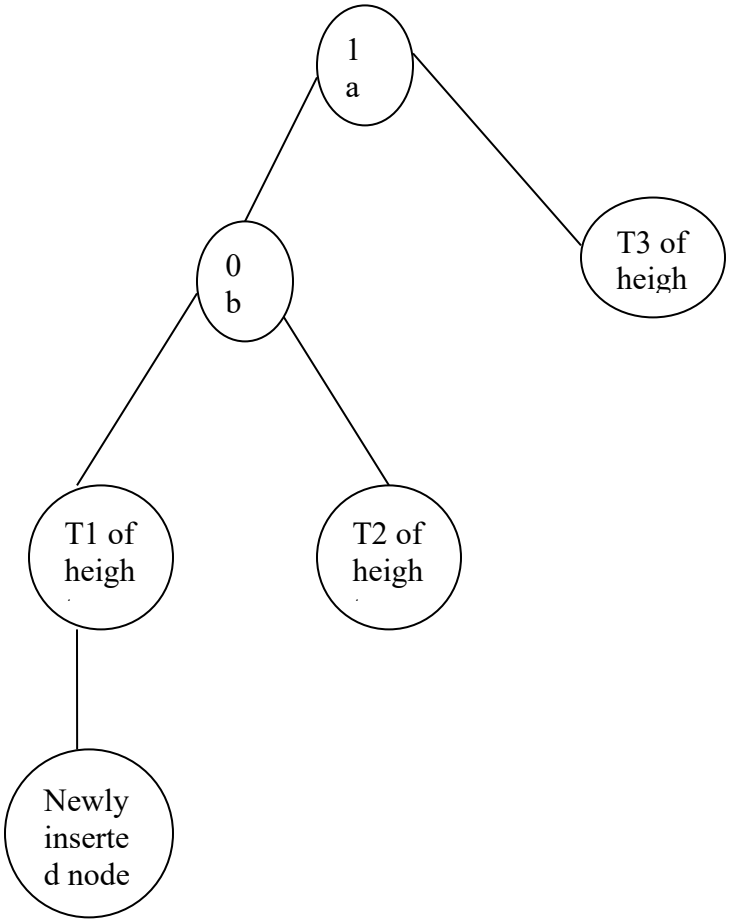


fig a

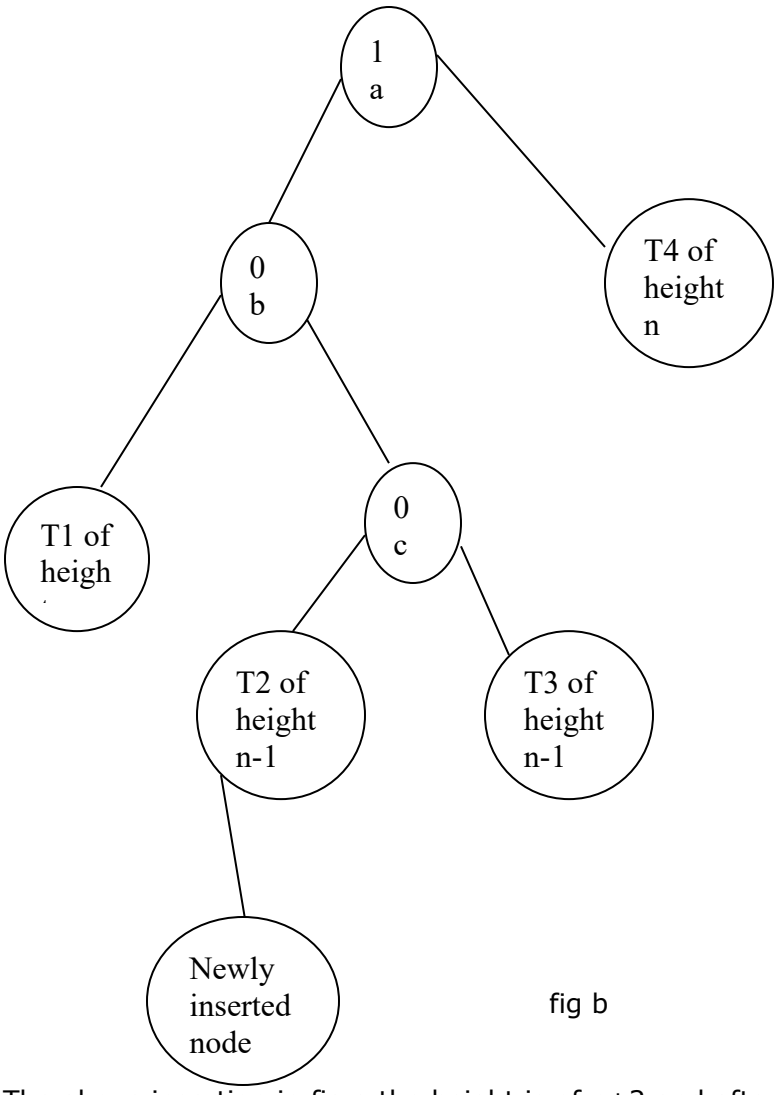
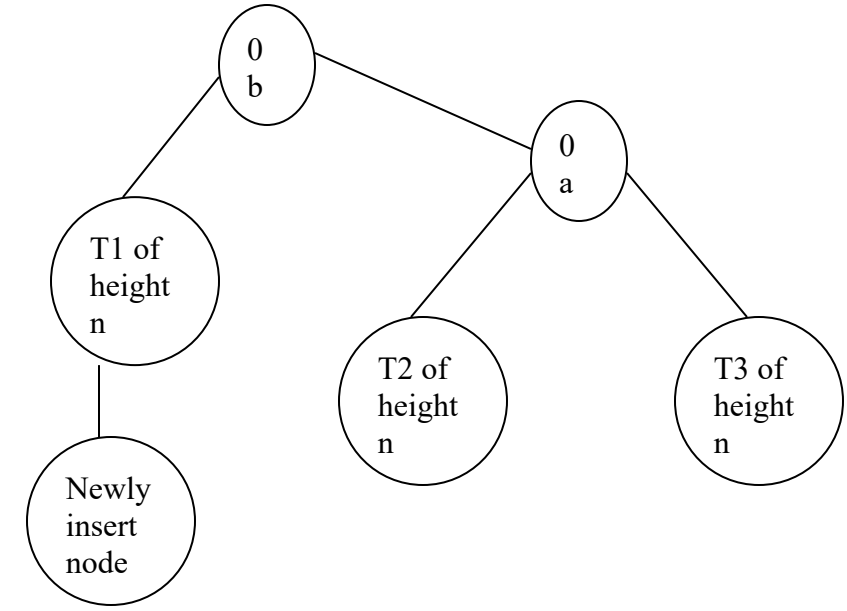


fig b

The above insertion in fig a the height is of n+2 and after rotation too it must remain same. In the same way the case is in the right insertion. The balanced tree after rotation of fig a and fig b is given below respectively:



Right rotation of fig a

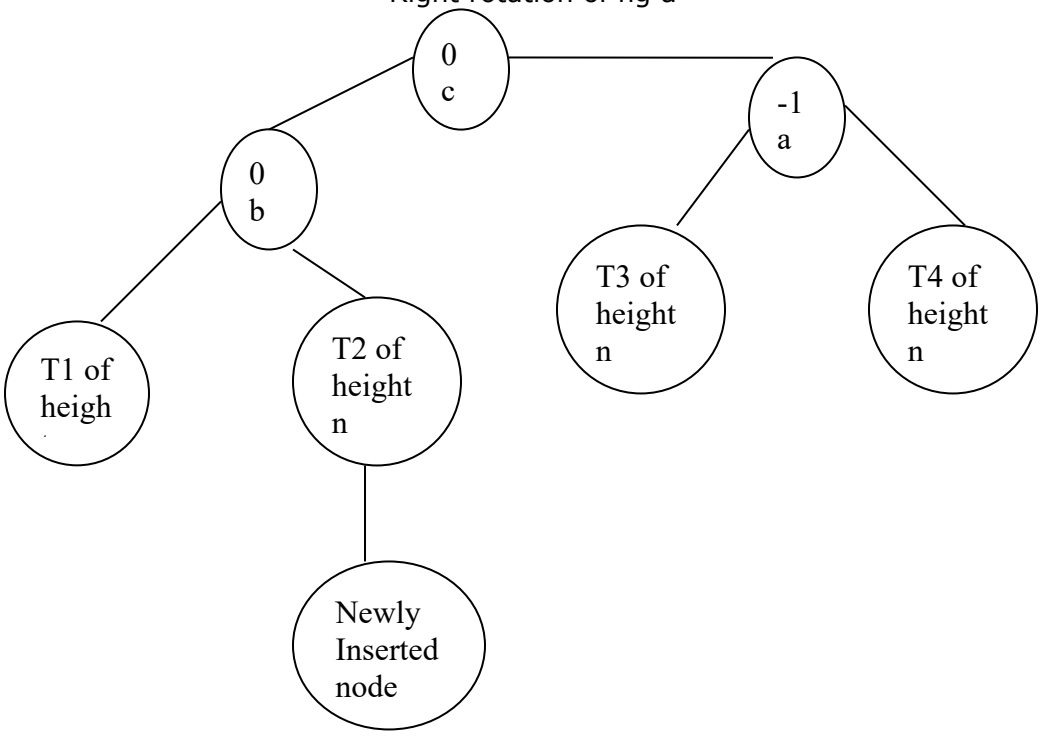


fig: Left Rotated Balanced tree of fig b

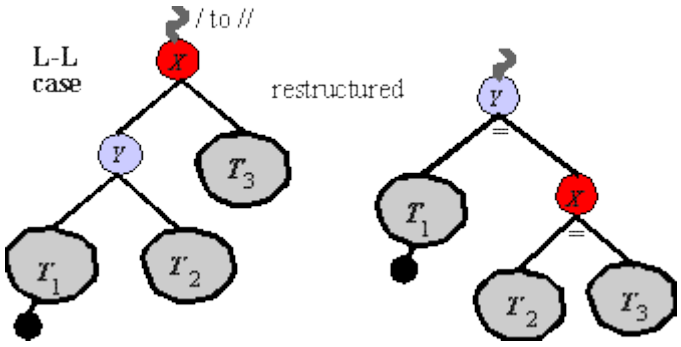
38.ii.What are the steps to make an unbalanced tree into a Balanced one? Explain.

Balanced tree is one in which the difference between the height of left child and the height of the right child is either 1or –1 or both the child are at the same level. Hence, to convert an unbalanced tree into a balanced one , the given previous condition must be fulfilled. This can be achieved by implementing the process of rotation.

Rotation can be of two types:-

a) single rotation
b) double rotation

Let us consider the case given below:



The L-L rebalancing case appears above. For an L-L restructuring to occur, a node is attached to the tree and all the nodes that are in the path back to the root are rebalanced following the rules described above, until a node with a '/' balance is encountered. From the viewpoint of the node that became unbalanced, the newly attached node was down a left sub tree, then another left sub tree, as indicated in the figure. If *X* is the node that went out of balance and *Y* is the left child of *X*, the nodes are rotated, as indicated in the figure. Node *Y* becomes the new root of the sub tree. The repositioning makes *X* the right child of Y and the right sub tree of *Y* becomes the left sub tree of *X*.

In the L-L case, both the *X* and *Y* nodes become balanced. The rebalancing is obtained by observing that before the new node was attached, if *n* is the length of the longest path in T_1, then the longest path in T_2 and T_3 are also *n*. Note that the new node might have been attached as the left child of *Y*, which implies that $n = 0$. The R-R case is a mirror of this case.

39.i.Write down the AVL Balancing algorithm.

1. The newly placed node being assigned a balance value of '='. Other nodes are rebalanced by traversing up the path from the new node to the root .

2.When the rebalancing algorithm traverses back toward the root along the left arc of a node, the rebalancing algorithm performs the following process on the node, depending on its current balance.

3.Restructure the tree as described in the AVL restructuring algorithm (described ahead) and terminate the rebalancing process, since 2.When the rebalancing algorithm traverses back toward the root along the left arc of a node, the rebalancing algorithm performs the following process on the node, depending on its current balance. no additional nodes need be rebalanced.

4.Rebalance the node as '=' and terminate the rebalancing procedure, since no additional nodes need rebalancing.

5.When the rebalancing algorithm traverses back toward the root along the right arc of a node, the rebalancing algorithm performs the following process on the node, depending on its current balance:

6.Rebalance the node as '=' and terminate the rebalancing, since no additional nodes need be rebalanced.
7.Rebalance the node as '+' and traverse to the parent and continue rebalancing.

8.Rebalance the node as '-' and continue the algorithm after traversing up to the parent.

9.Restructure the tree as described in the AVL restructuring algorithm and terminate the rebalancing, since no additional nodes need be rebalanced.

39.ii. Write down the AVL Balancing algorithm

*A*VL Trees -- Balancing Algorithms

An insertion can cause an AVL tree to become unbalanced in four cases:

1. An insertion into the left sub tree of the left child of node alpha.

2. An insertion into the right sub tree of the left child of node alpha.

3. An insertion into the left sub tree of the right child of alpha.

4. An insertion into the right sub tree of the right child of alpha.

Cases 1&4 are symmetrical, so the same algorithm (with minor variations) can be used to rebalance the tree. Likewise with cases 2&3.

40.i. Define Huffman Algorithm. What is the application of Huffman Algorithm?

Huffman algorithm is an algorithm, which is used to represent file, or it is used to represent the strings of input with the code. And code is a sequence of zeros and ones that can uniquely represent a character. The number of bits required to represent each character depends upon the number of characters that have to be represented. If the numbers of character to be represented is two then only 1 bit is sufficient for that. And as the number of character increases the bit required to represent that also increases. If we have to represent only two characters, lets say 'A' and 'B' then that can be represented by 1 bit i.e. we can use "0"to represent 'A' and "1" to represent 'B'. But if we need to represent three characters then 1 bit is not sufficient, we required 2 bits. Similarly we can represent 8 characters using 3bit but to represent 9 characters we need 4bit. That means n bit can represent at most 2^n characters.

Huffman Algorithm is used to compress the file or the strings of input. It does that by assigning the smaller code to the frequently used characters and with the longer code to the less frequently used character. For e.g. let us

take a string AAAAAAAAAABBBBBBBBCCCCCCDDDDDEE

Here the number of times each character occur in the given string is as follows:

A is 10

B is 8

C is 6

D is 5

E is 2

If each character is represented by using three bits then number of bits require to store this file will be

3*10 + 3*8 + 3*6 + 3*5 +3*2 = 93 Bits.

Now suppose if we represent the character

A by the code 11

B by the code 10

C by the code 00

D by the code 011

E by the code 010

then the size of the file becomes 2*10 + 2*8 + 2*6 + 3*5 + 3*2 = 69 bits. A certain amount of compression has been achieved. In general much higher compression ratios can be achieved by using this method. As we can see frequently used characters are assigned smaller codes while less frequently used characters are assigned larger codes. One of the difficulties of using a variable-length code knows when we reached the end of a character in reading a sequence of zeros and ones. This problem can be solved if we design the code in such a way that no complete code for any character is the beginning of the code for another character. In the above case 11 represent A. No other code begins with 11. Similarly B is assigned the code 00. No other code begins with 00. Such codes are known as prefix codes.

40.ii. Define Huffman Algorithm. What is the application of Huffman Algorithm?

41.i. Show with an example, How Huffman Algorithm works?

The Huffman algorithm is the basic data compression algorithm that is used to compress text files. The basic principle of this algorithm is the using the shortest symbol for the large occurring letter. This algorithm encodes the text by constructing the tree, which consists of a node having the frequency of occurrence of the symbol and the symbol itself. The process of assigning the code to the symbol can be done in the program by first calculating the frequency of occurrence of the alphabet and then combining the least frequency alphabet first and then again combining the least occurrence of the alphabet until all the alphabet are combined to become the single symbol. This can be done by constructing the tree. This tree is also referred to as a Huffman tree.

The action of combining two symbols into one suggests the use of a binary tree. Each node of the tree represents a symbol and each leaf represents a symbol of the original alphabet. Each node in the tree contains a symbol and its frequency.
Once the Huffman tree is constructed, the code of any symbol in the alphabet can be constructed by starting at the leaf representing that symbol and climbing up to the root. To decode the tree 0 is given each time a left branch is climbed and 1 is appended to the beginning of the code each time

right branch is climbed. In this way a code for any symbol can be found. As an example let us take the following case:
Consider the following message
aaaaaabbbooy
Here the number of a b o and y are repeated. The number of symbols here is four. For four symbols it is sufficient to use two bits. Since 2^n is four. Let us give the codes for the symbols as follows:

Symbol	**Frequency**	**Codes**
a	6	00
b	3	01
o	2	10
y	1	11

Giving the above codes the messages can be encoded as:
000000000000001010110101111
The total number of bits is then 2*(6+3+2+1) =24 bits.
But if we specify bits as follows:
a=>0
b=>10
o=>111
y=>110
Then total number of bits will be then 6*1+3*2+2*3+1*3=21bits.
Thus we see that the number of bits need will be decreased. In this case the reduction of bits has not been a great. But in a huge file the number of bits reduced will be of much importance. The Huffman algorithm gives the larger bit for low frequency symbol and smallest bit for the highest occurring symbol as we have seen in the example.
Using the Huffman algorithm the algorithm itself constructs a tree from the frequency distribution of the symbols and uses it to find the codes and to decode the code.

The tree constructed for the above example is shown below:

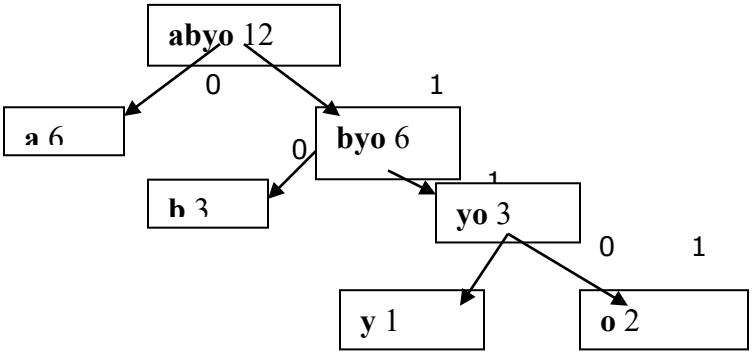


Figure Binary tree

The above tree is constructed form the example. We see from the tree that the actual symbols reside on the leaf of the tree and the non-leaf node of the tree consists of a total frequency of the left and right branch. In the tree above the lowest frequency are 1, and 2 of symbols 'y' and 'o' respectively. The minimum frequencies are summed up and made a new node having combination of two symbols 'yo' and the combined frequency 3. Similarly this is done for the symbols b and 'yo' and the sum is put to the new node. In the end of construction of the tree the total frequency of the symbols are in the root.

While decoding the tree it is decoded by climbing tree from root to the leaf. Begin at the root of the tree. Each time 0 is encountered, we move down to a left branch, and each time 1 is encountered,

we move to the right branch. This process is repeated until we encounter a leaf. The character of the original message is the symbol that corresponds to that leaf.

ge is the symbol that corresponds to that leaf.

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		3			A
d 2		R			R
d 3		A			S
d 4		N			3
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					d Pointer Table
	d 1 →	3		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.

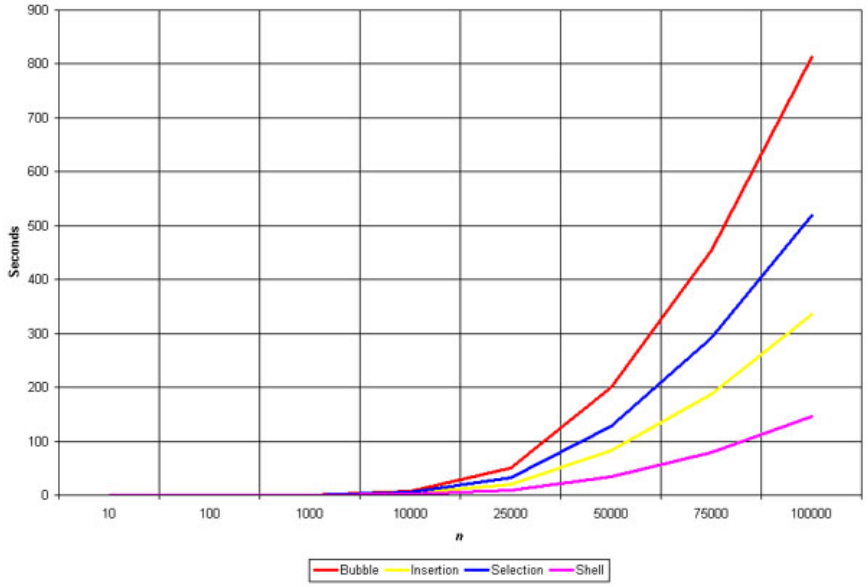
For example, *O* (*n*) means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items (10 * 10 = 100). If the complexity was *O* (n^2) (quadratic complexity), then it would take 100 times longer to operate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

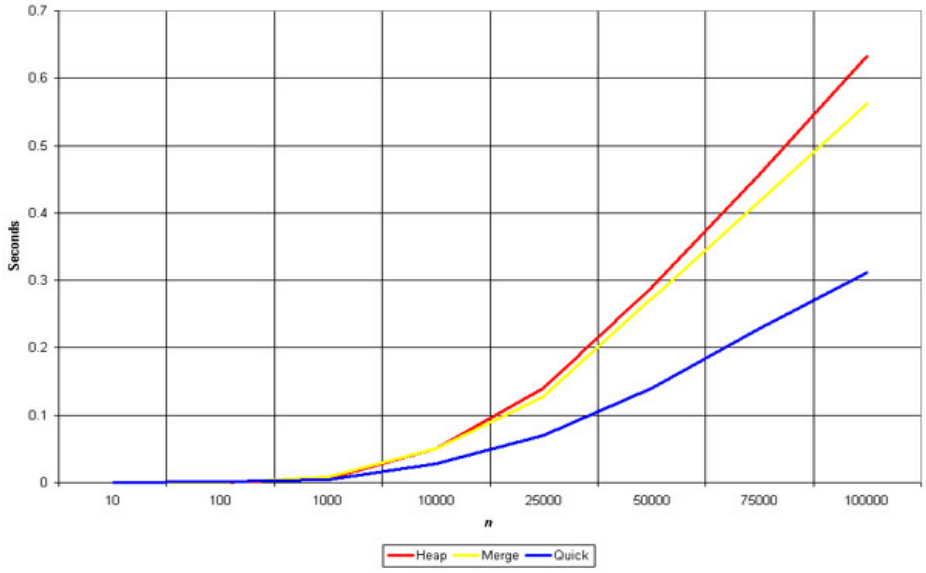
From the graph the bubble sort is inefficient and the shell sort is the best. On the other-side, all of these algorithms are simple and is useful for quick test programs, rapid prototypes, or internal-use software.

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 $\Theta(n)$ 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 $\Theta(n)$ 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:

$$T(n) = 1 + 2 + 3 + \ldots + (n-1)$$
$$= \sum_{i=1}^{n-1} i$$
$$= \frac{n}{2}(n-1)$$

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:

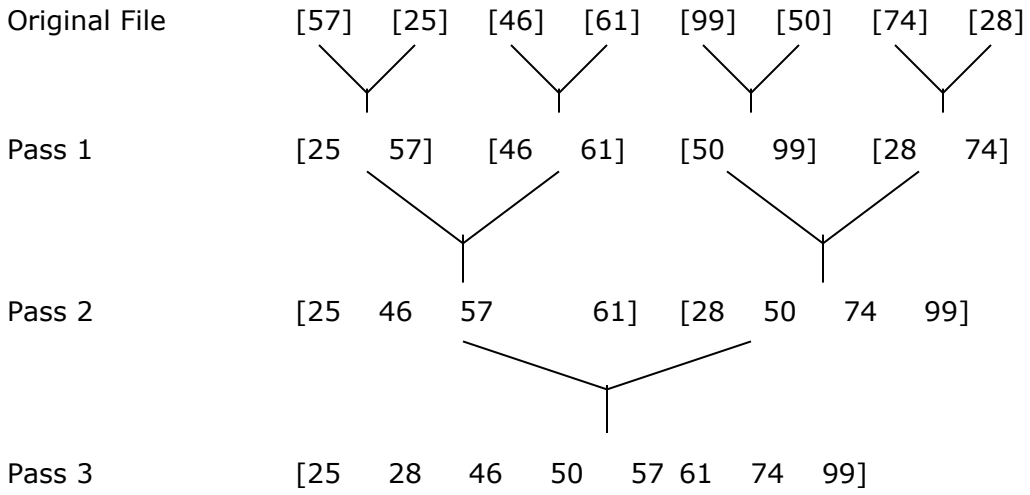
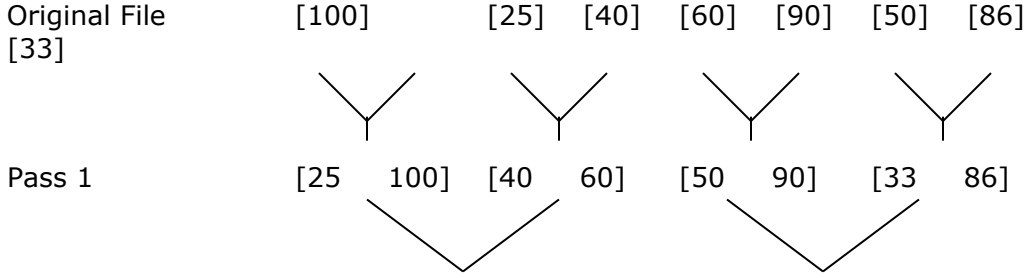


Figure 57.1: Successive Passes of the Merge Sort

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.



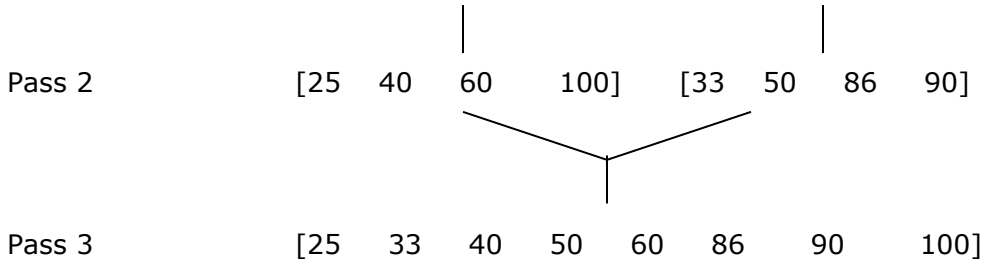


Fig. Successive Passes of the Merge Sort

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 i^{th} 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 $1/5^{th}$ of the elements of the original file are sorted. Increments are 5,3,1 on 1^{st} ,2^{nd} and 3^{rd} 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 j^{th} element of the k^{th} 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

1^{st} 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.

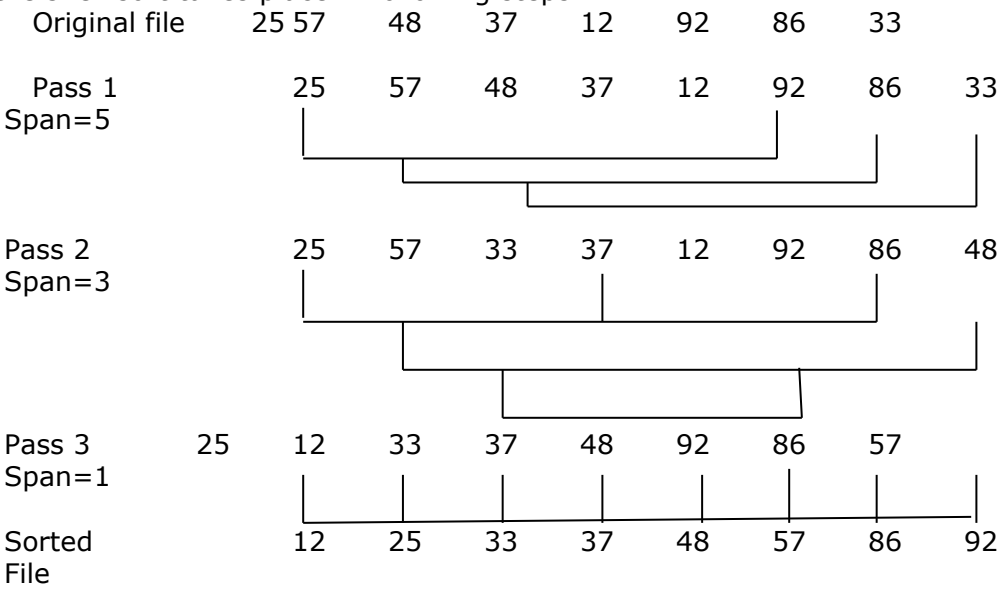
The algorithm of the Shell Sort is as follows :

```
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.



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.

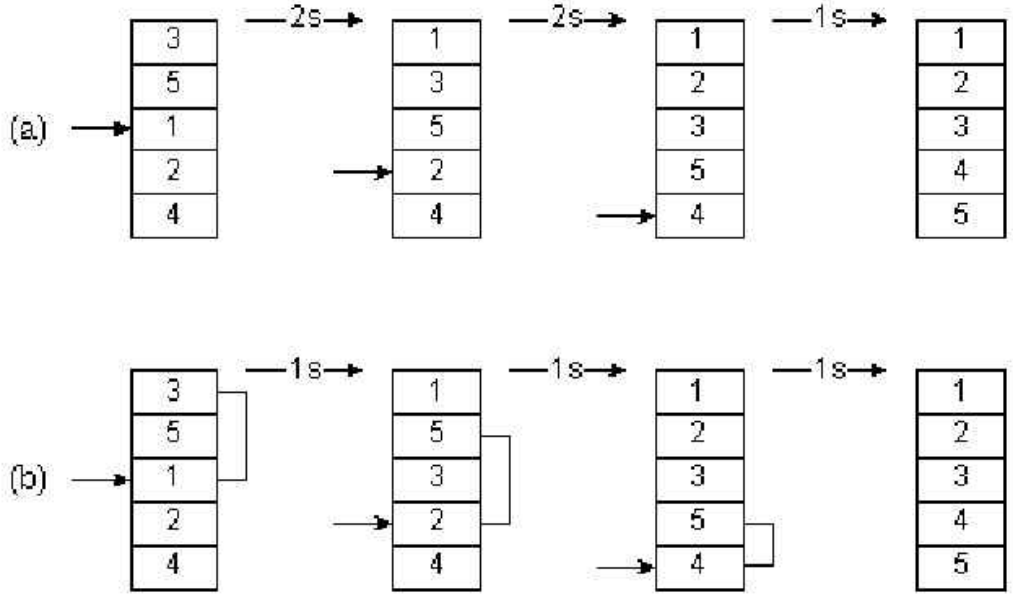
SHELL SORT

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

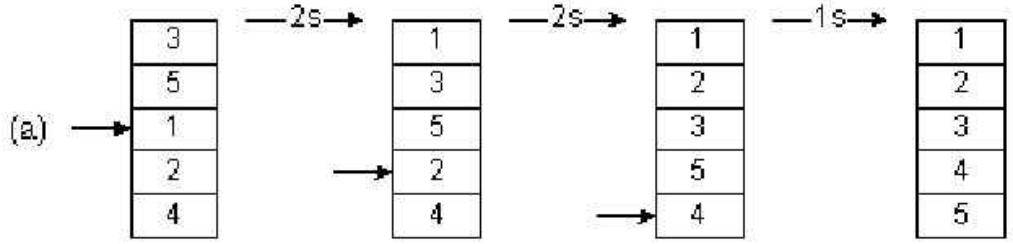
As we have seen that insertion sort and selection sort behave in opposite ways.
Selection sort moves the items very efficiently but does many redundant comparisons. In its best case, insertion sort does the minimum no. of comparisons, but is inefficient in moving

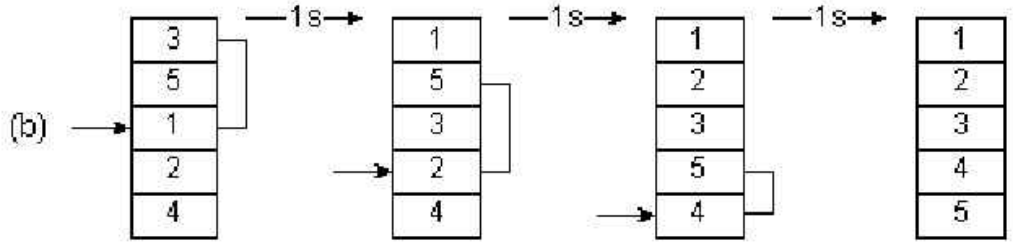
items only one place at a time. Now, the problems with both of these are avoided by another method called shell sort. This method sorts separate sub files of the original file. These sub files contain every kth element of the original file. The value of k is called an increment. For example, if k is 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

9	12	24	30	36	45	70

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

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.

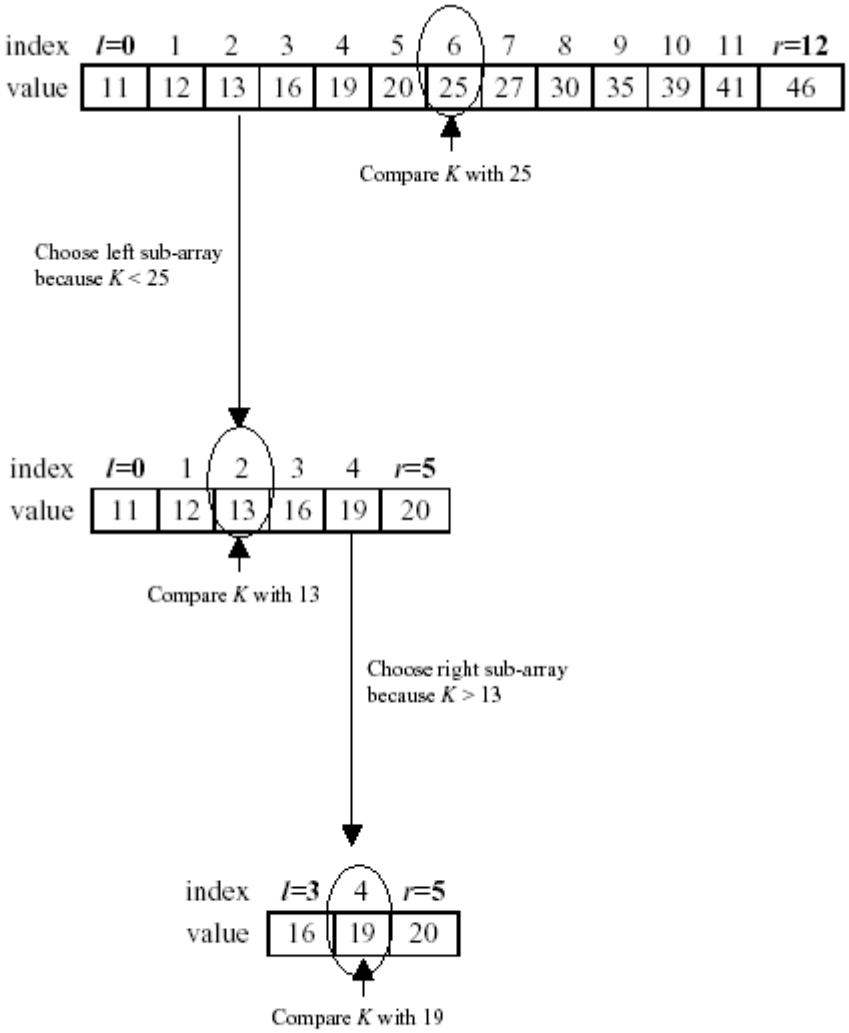
2(a). if key k is LESS THAN value pointed to by the mid pointer
"right" pointer is set to point to mid index − 1 (i.e. ignore top half of current list)
2(b). Else item is GREATER THAN value pointed to by mid pointer
"left" pointer is set to point to mid index + 1 (I.e. ignore bottom half of current list)
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 0^{th} 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.

Position	key	record
0	4967000	
1		
2	8421002	
990		
991		
	0000990	
	8765991	
999		
	0001999	

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.

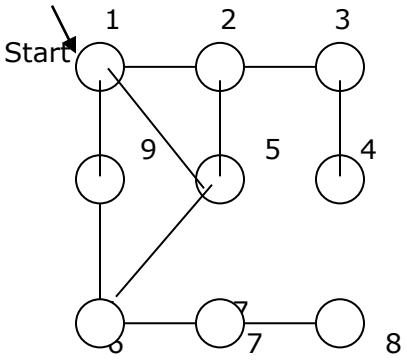
```
 for (every node nd)
     visited (nd)=FALSE;
s=a pointer to the starting node for the traversal
while (s=!null){
dftraverse(s);
s=select();
}
```

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

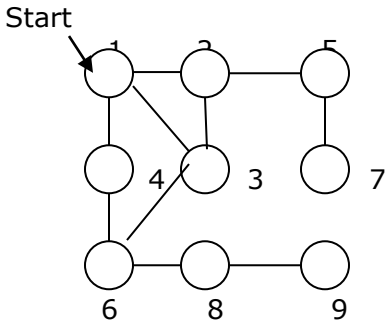
an undirected graph. Depth-first traversal can also be used to determine if a graph is acyclic.

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

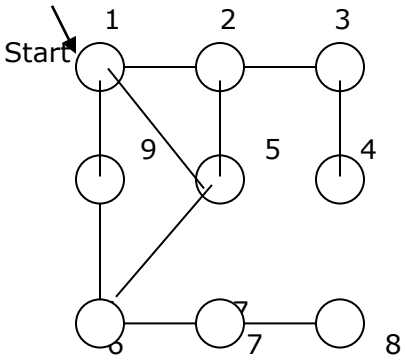
In implementing depth-first traversal, each visited node is placed on a stack (either implicitly via recursion or explicitly), reflecting the fact that the last node visited is the first node whose successors will be visited. Breadth-first traversal is implemented using a queue, representing the fact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

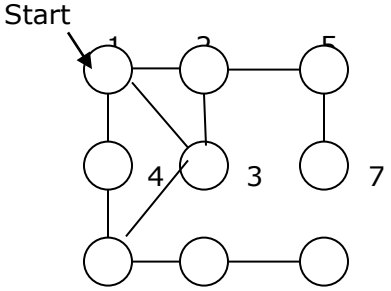
Suppose that the traversal has just visited a vertex v, and let w1, w2, wk be the vertices adjacent to v. Then we shall next visit w1 and keep w2, wk waiting. After visiting w1, we traverse all the vertices to which it is adjacent before returning to traverse w2, wk.



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.



6 8 9

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:

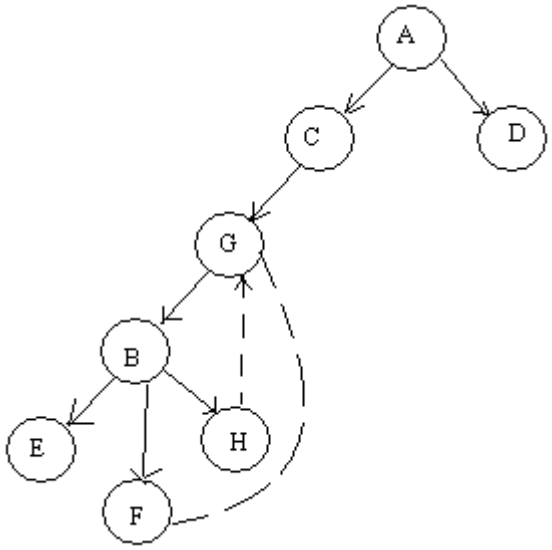
Tree edges, forward edges, cross edges and back edges.

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

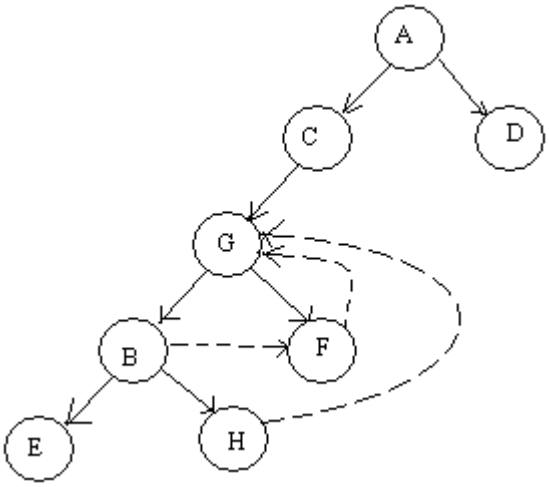


Fig b

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

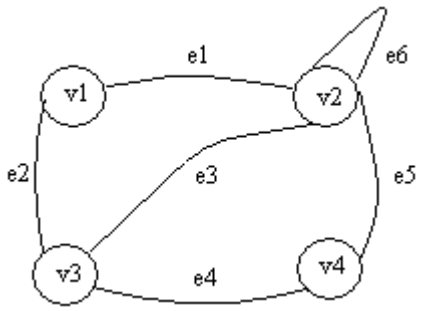


Fig: undirected graph

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.

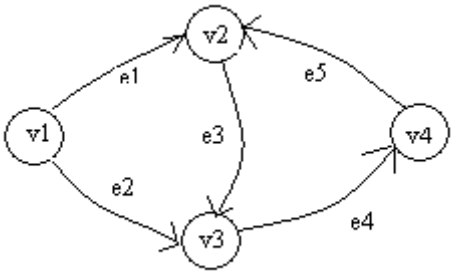


Fig: Directed Graph

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.*

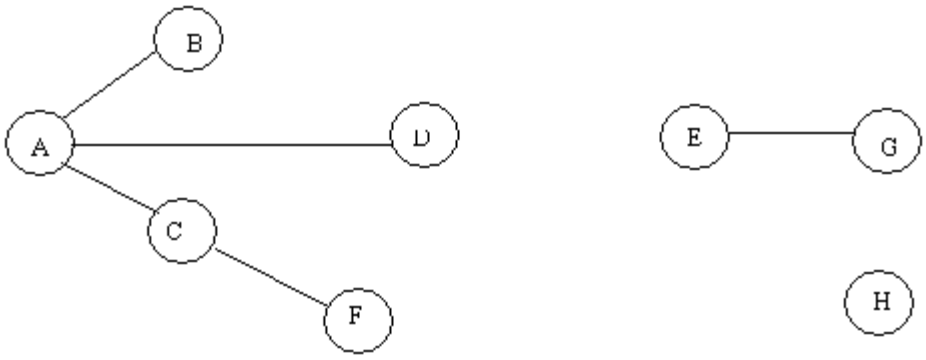


Fig: undirected and not-connected graph

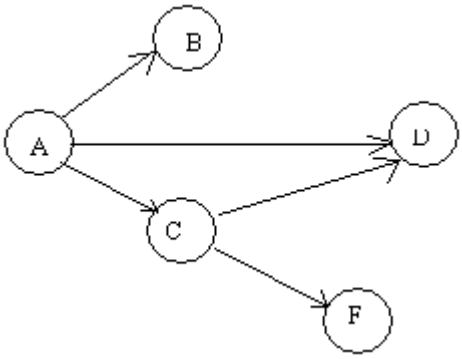


Fig: directed and connected graph

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.

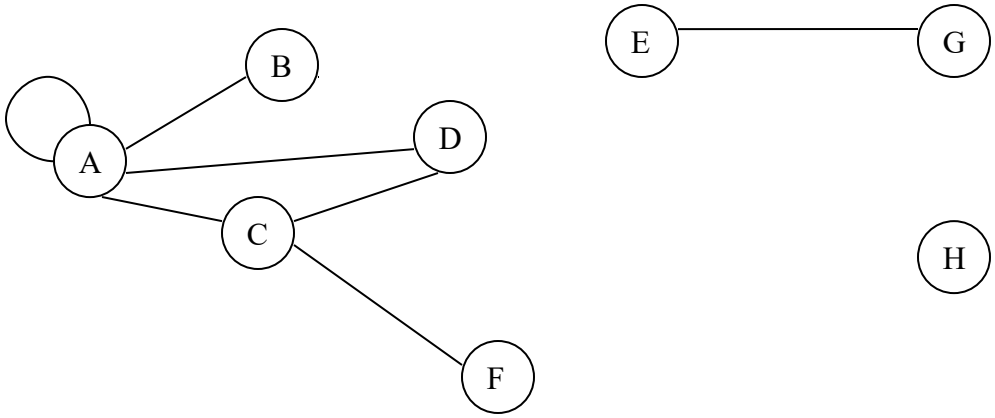


FIG 2

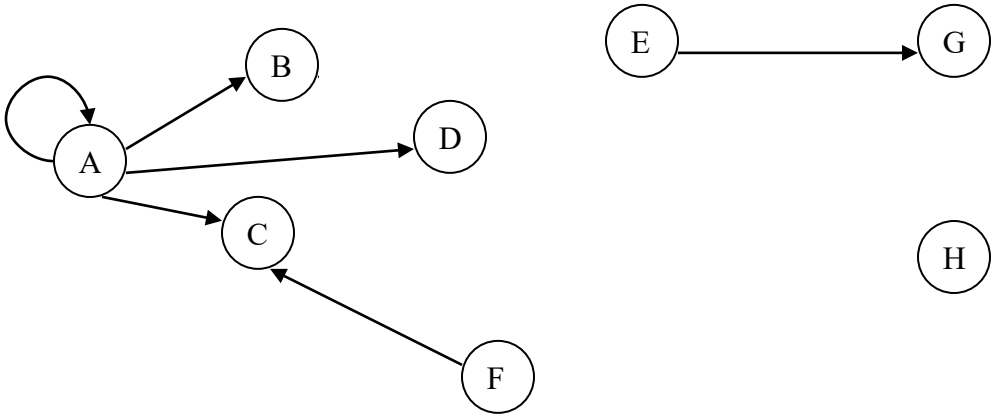


FIG 3

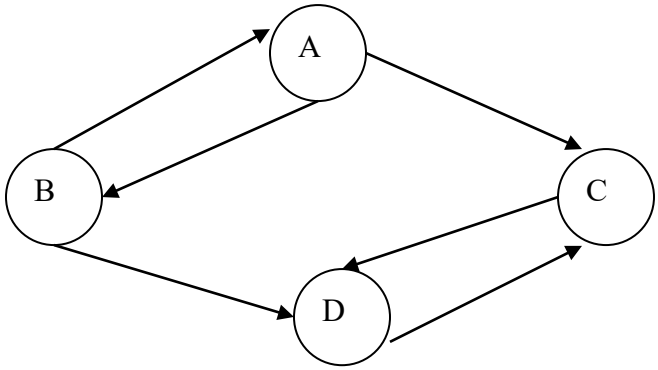


FIG 5

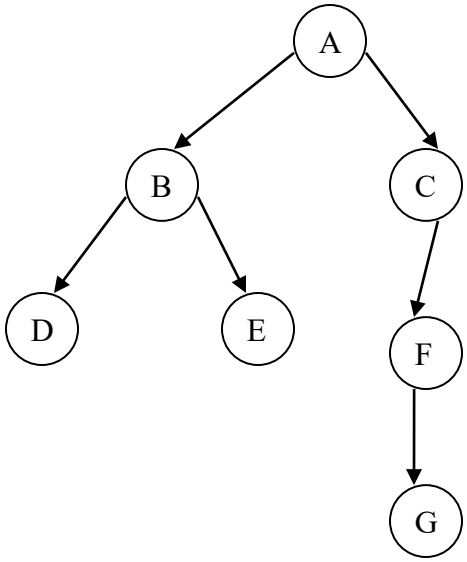


FIG 4

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.

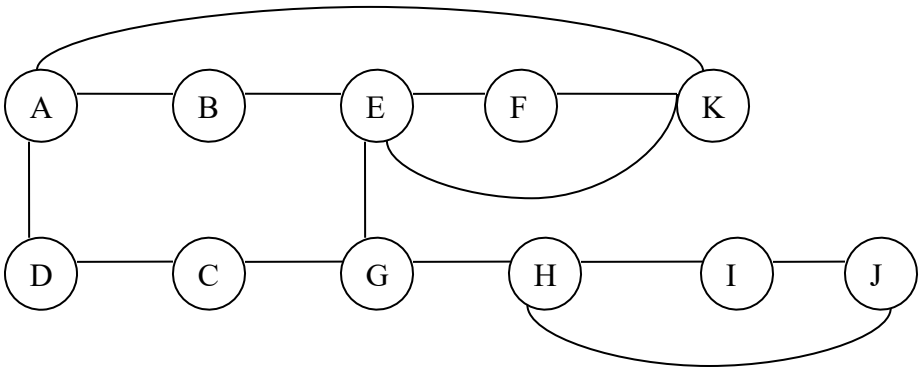


Fig 6

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.