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.

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

a)
b)
c) Priority Queue

The difference between Linear queue is at the time when the queue is full i.e. one of the pointe

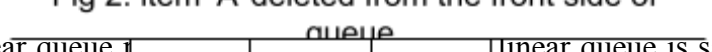


Fig 2: item 'A' deleted from the front side of queue

In the linear queue i linear queue is said to be empty whenever rear < front and the number of ue 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

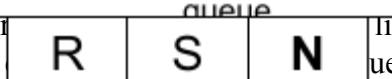


Fig 3: item 'N' being inserted from rear side of the queue. On an attempt to insert the eleme queue.

Fig: Queue

4					4			4	M	Rear =4
3			3			3			3	N	
2			2	S	Rear = 2	2	S	Front =	2	S	
1			1	R		1			1		Front =2
0		Rear = -1 Front = 0	0	A	Front = 0	0		Rear = 2	0		

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.

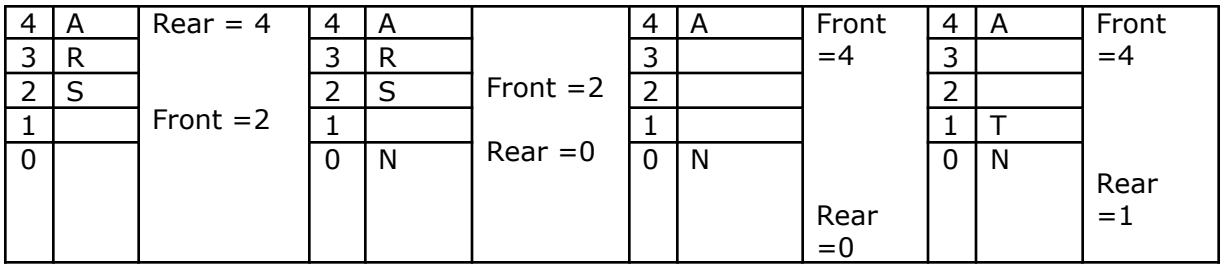


Figure: Elements being inserted and deleted from Circular Queue

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

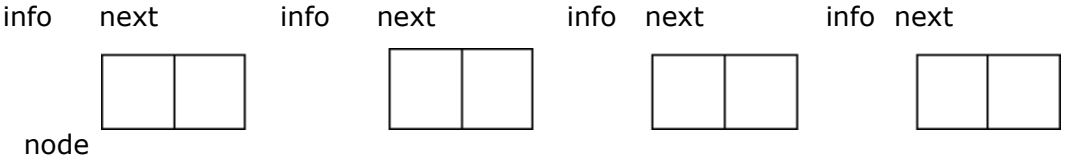
- Doubly Linked List
  - Linear Doubly Linked List
  - Circular doubly Linked List

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.

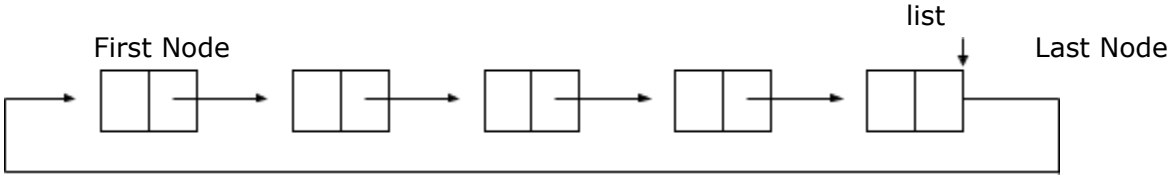


Figure 16.2: Circular List

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:

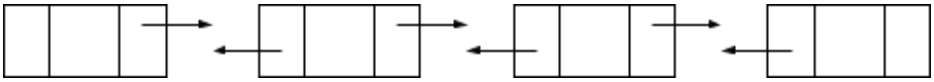


Figure 16.3: Linear doubly linked list

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:

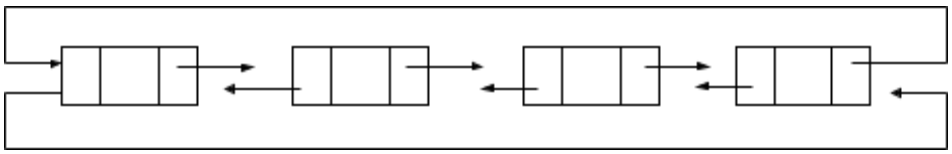


Figure 16.4: Circular doubly linked list

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.

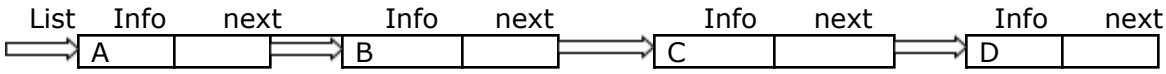
In another word, the linked list does not allocate storage memory for variables, until it is needed. As nodes are needed, the system is called upon to provide them. Any storing space not used for one type of node may be used for another. Thus, as long as sufficient storage is available for the nodes 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.

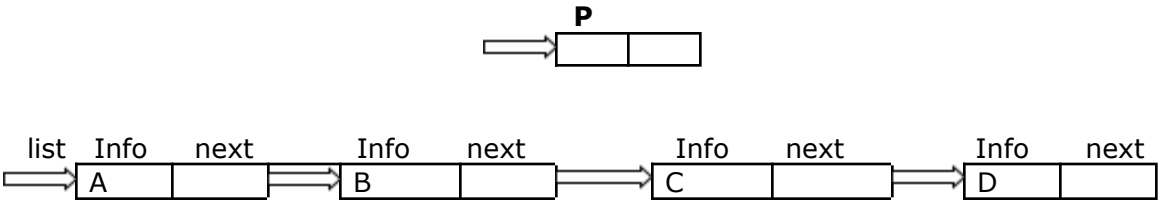
Let us discuss about the insertion of nodes in a list. For this let us consider a list as shown below.

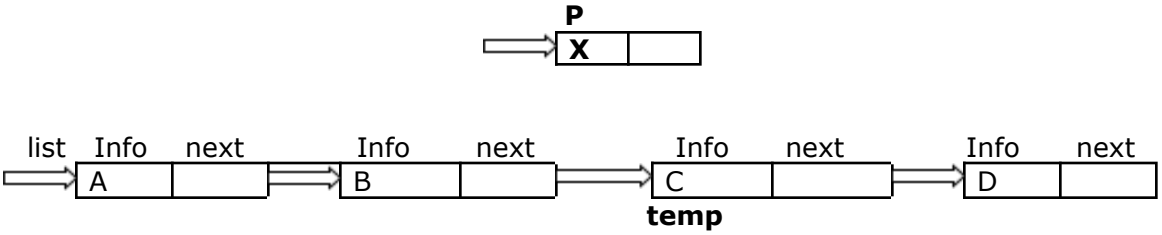


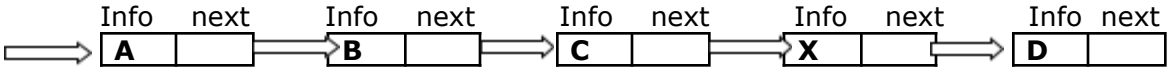
If we have to insert a node with content 'X' after the content 'C' then for this we will first check the content of each node of a list until the content 'C' is not found. Then a new node is allocated by the function getnode() and place the element 'X' in the info field of the new node and the pointer 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;
```

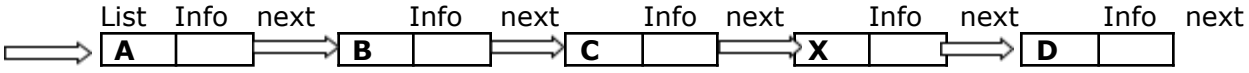
```
        p=getnode();
        p->info='X';
        p->next =temp->next;
        temp->next=p;
}
```





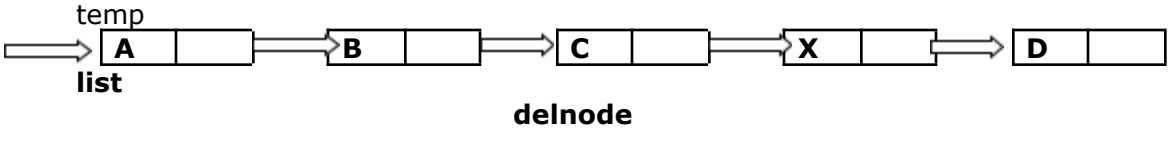


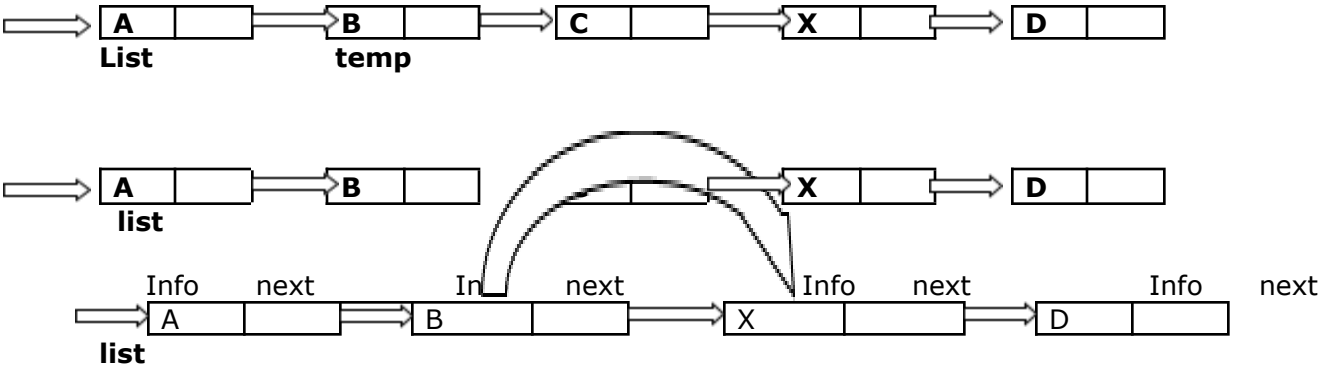
Now let us discuss about the deletion of the nodes from a list. For this, let us consider the above example, again which is as given below:



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);
}
```

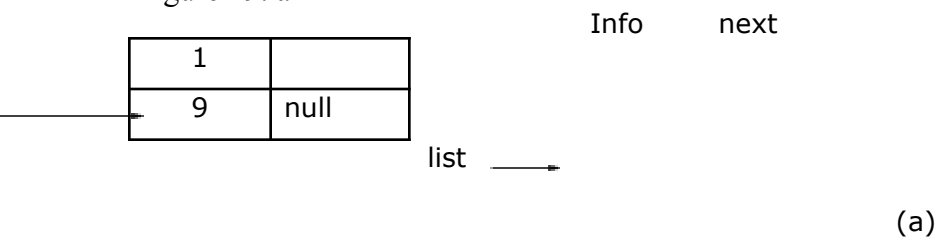




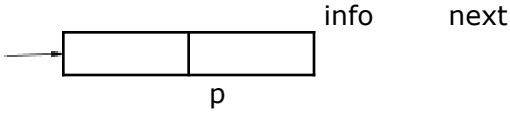
ii. Write an algorithm for insertion and deletion of nodes from a list. Explain with figure.

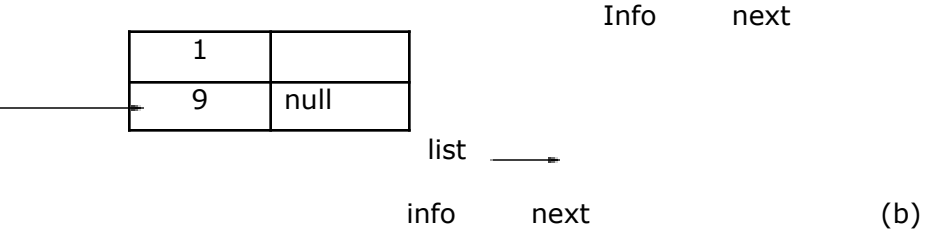
A list is a dynamic data structure of items called nodes. Each list in node consists of two fields, information field that holds actual element and an address field that contains the address of the next node in the list .the null pointer in a node signal the end of the list .The number of nodes in a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

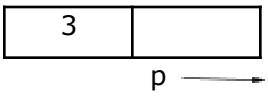


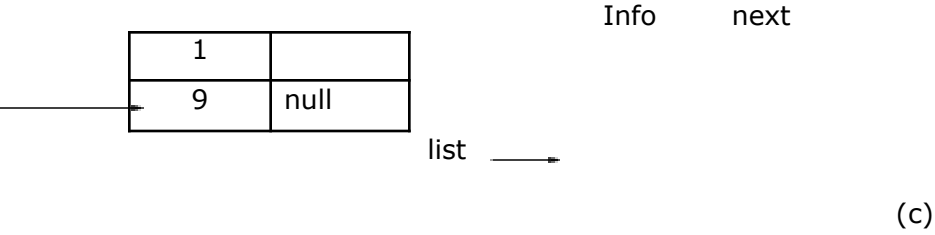
(a)



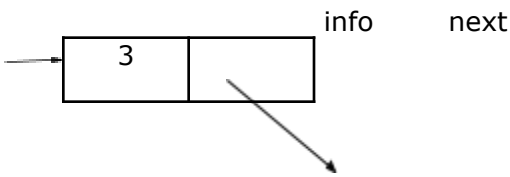


(b)





(c)



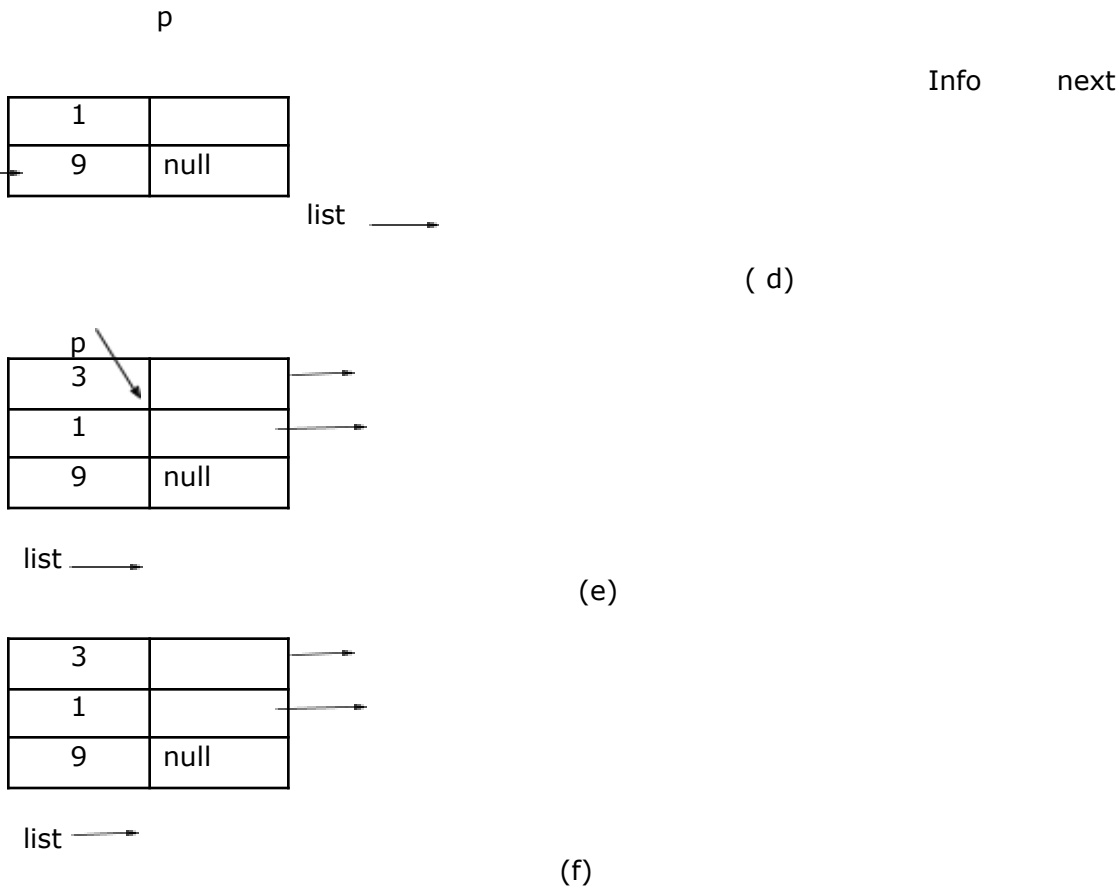


fig 19. a.Adding an element to the front of a list.

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

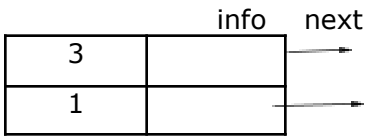
P=getnode()

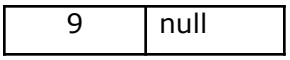
info (p) =3

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.



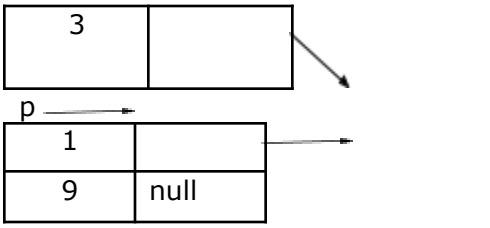


list →

(a)

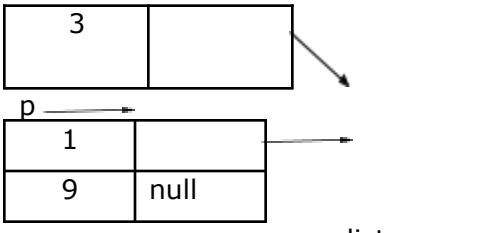
p →
list →

(b)



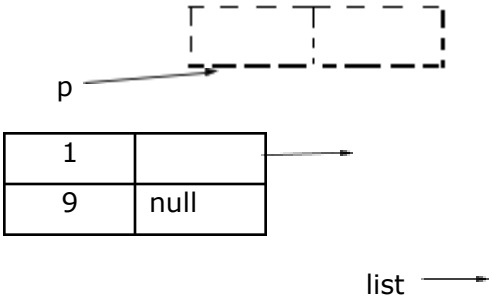
(c)

x=3



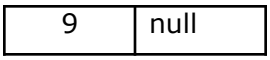
(d)

x=3



(e)

x=3



list

(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

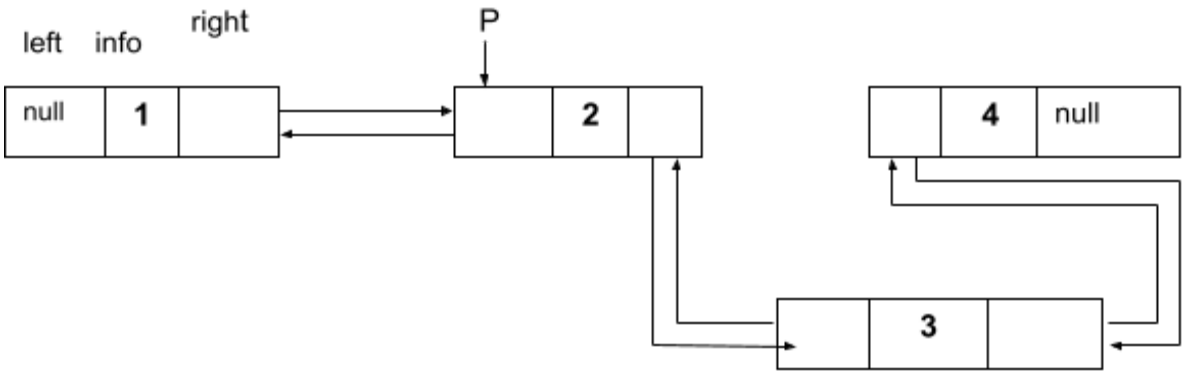


Figure: 20.a

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)
{
  NODEPTR q,r;
  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;/
```

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.

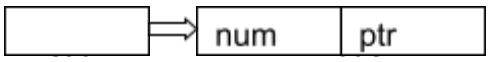
Header node is an extra node placed at the front of a list. Alternately, the header for a linked list is a pointer variable that locates the beginning of the list. The header will usually be a static variable, and by using its value we can arrive at the first (dynamic) node of the list. The header is 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));
```

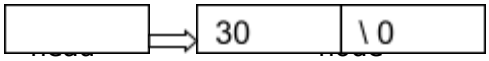
When the statement
head=(NODE*)malloc(sizeof(NODE));
is executed, a block of memory sufficient to store the NODE is allocated and assigns head as the starting address of the NODE(Now, head is an external pointer). This activity can be pictorially shown as follows;



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.

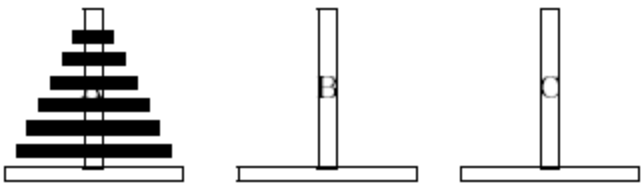
Header node is an extra node at the front of a list which does not represent any item in the list. Info portion of the header node is used to keep global information about the entire list (such as the number of nodes in it, or a pointer to its last node). Also, the number of items in the list may be 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.

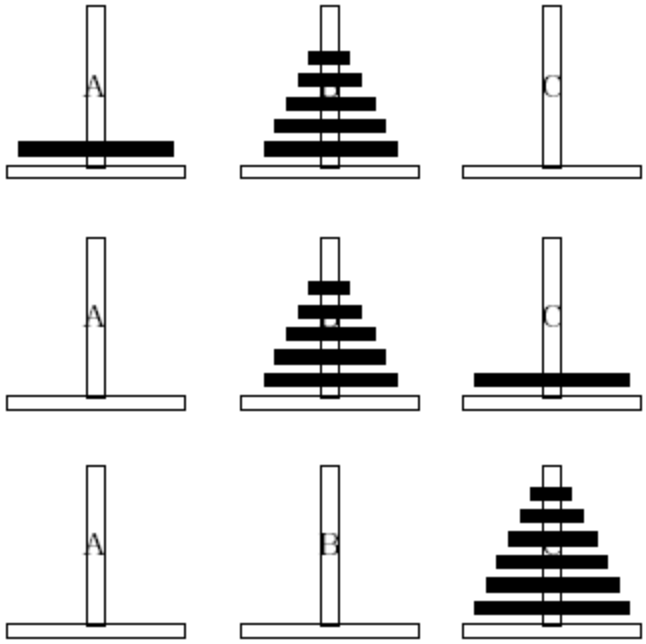
Recursion is one of the most powerful programming technique. It can also be defined as a method which includes an invocation of itself.

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.



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.

26. **26. i. Write down the function to compute the nth Fibonacci number using recursion. Ex 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

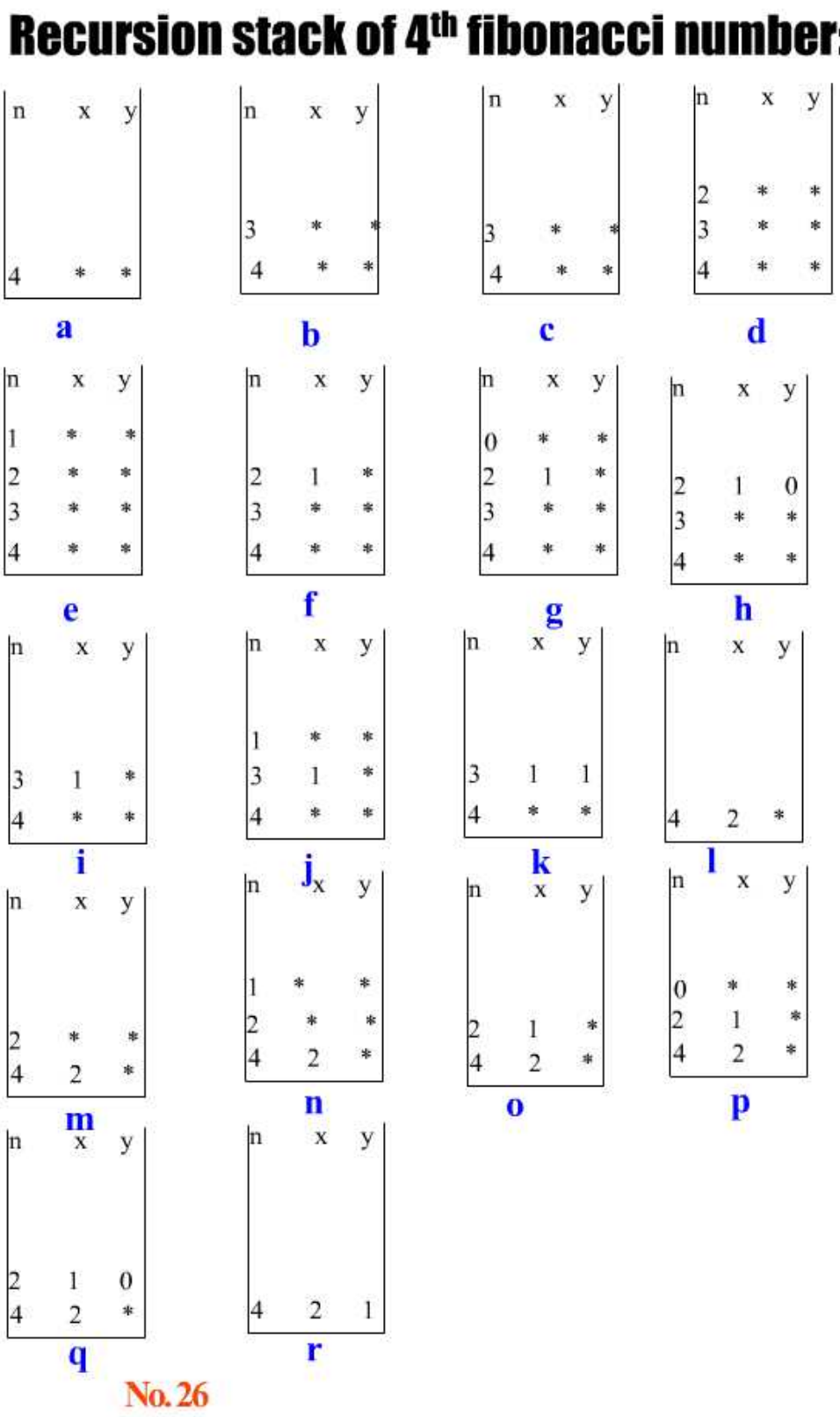
note that each recursive call results in a return to the point of call, so that the call fib(1) returns to the assignment to x, and the call of fib(0) returns to the assignment to y. The next statement to be executed in fib(2) is the statement that returns x+y=1+0=1 to the statement that calls fib(2) 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:

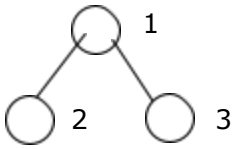
V L R	L V R	L R V
Preorder	In order	Post order

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.

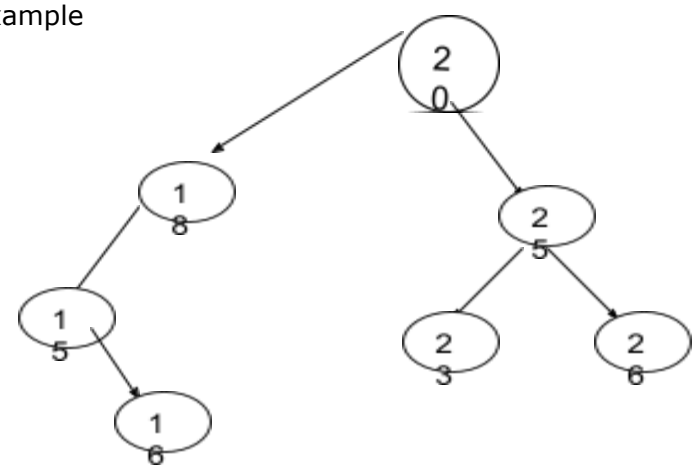
34.i. Explain how you can traverse in a tree to do sorting in descending order?

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. However, is no such "natural" linear order for the nodes of a tree. If we traverse the tree as mentioned as below then w 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.

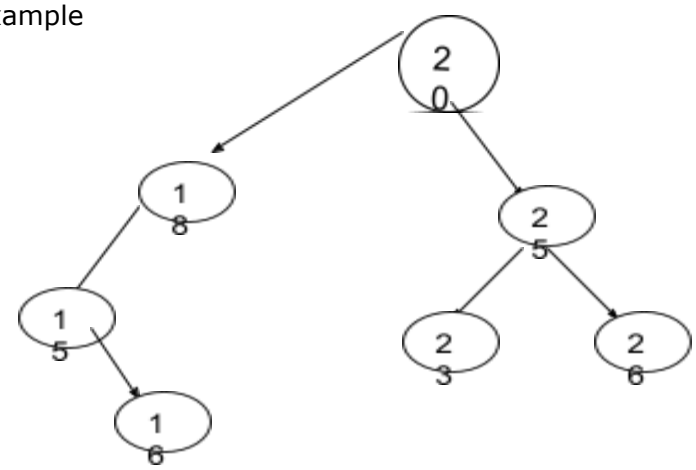
34. Explain how you can traverse in a tree to do sorting in descending order?

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)

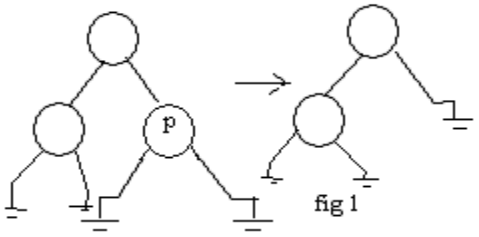


fig 1

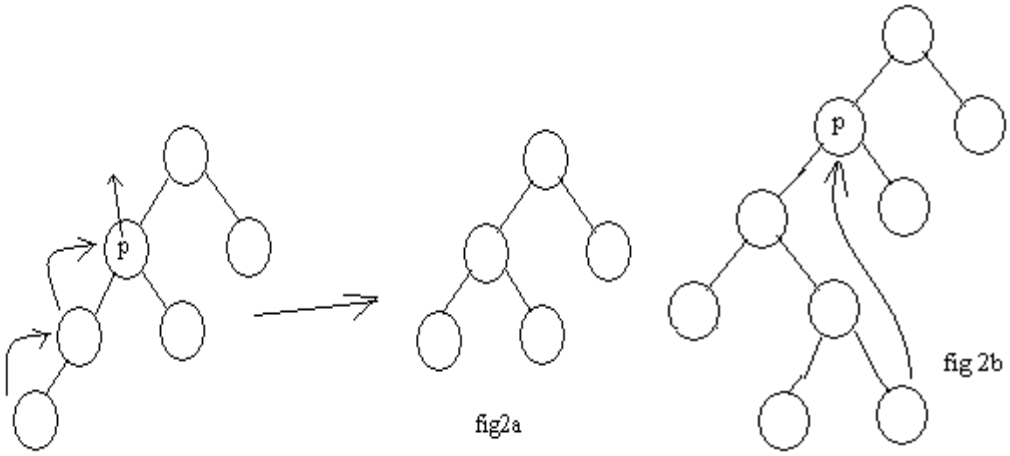


fig2a

fig 2b

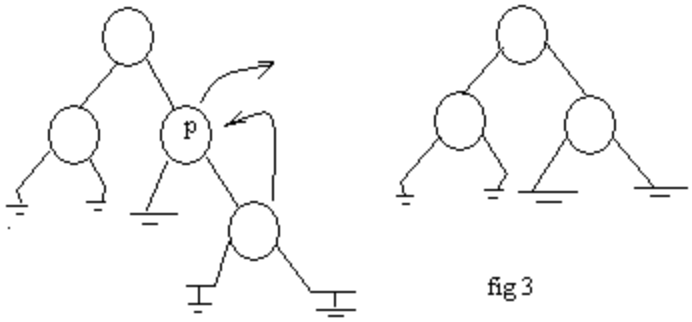


fig 3

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.

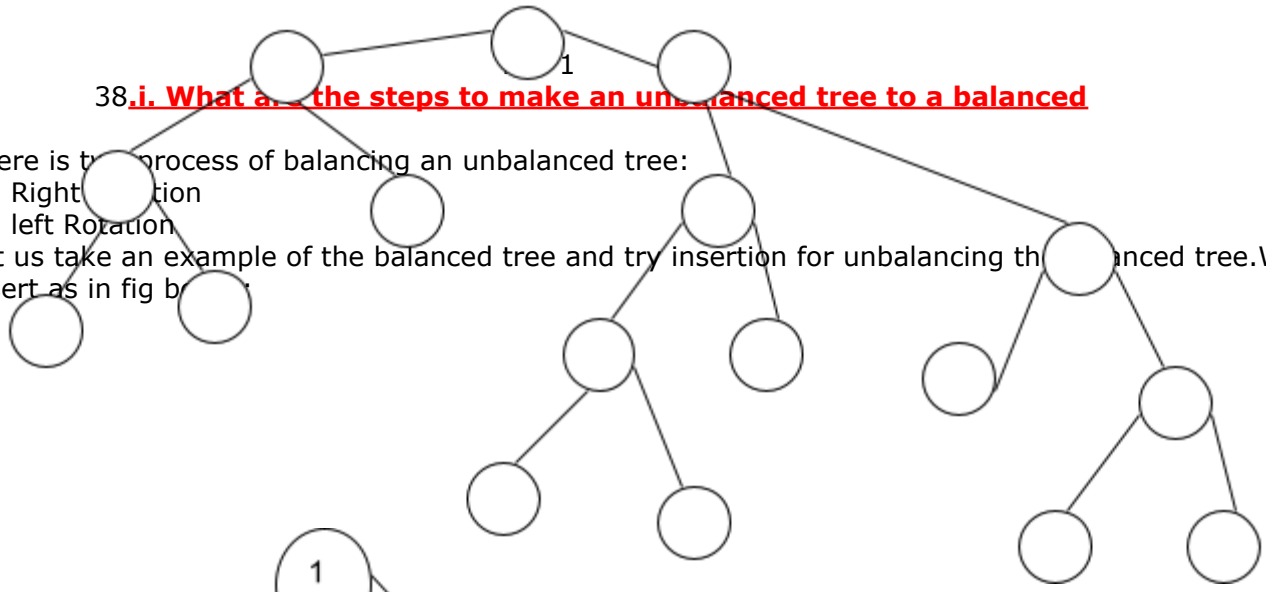
Where are AVL trees used?

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.



38.**i. What a[illegible] the steps to make an unbalanced tree to a balanced e?**

There is t[illegible]process of balancing an unbalanced tree:

1: Right[illegible]tion

2: left Rotation

Let us take an example of the balanced tree and try insertion for unbalancing th[illegible]anced tree.We insert as in fig b[illegible]

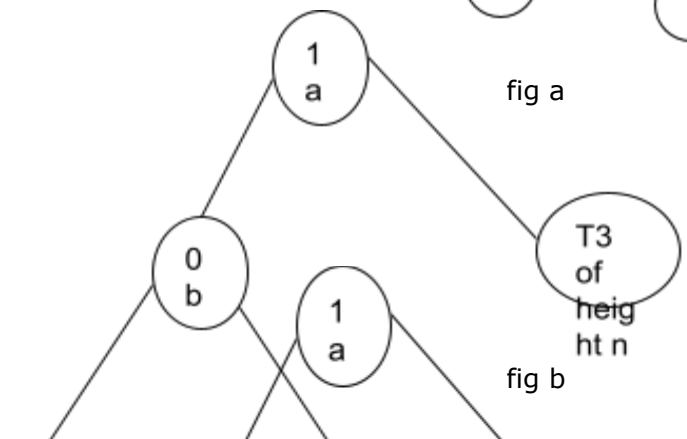
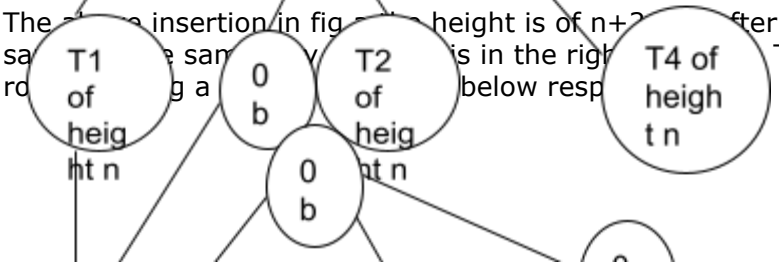


fig a

fig b

The [illegible] insertion in fig [illegible] height is of n+2[illegible]fter rotation too it must remain sa[illegible]e sam[illegible]y [illegible]is in the righ[illegible] The balanced tree after ro[illegible]g a[illegible]below resp[illegible]



Right rotation of fig a

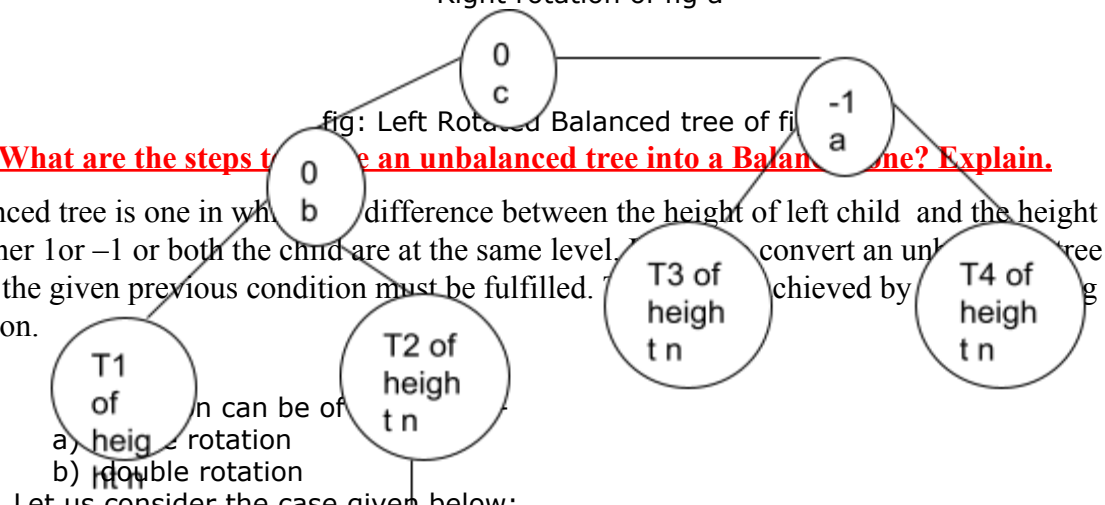


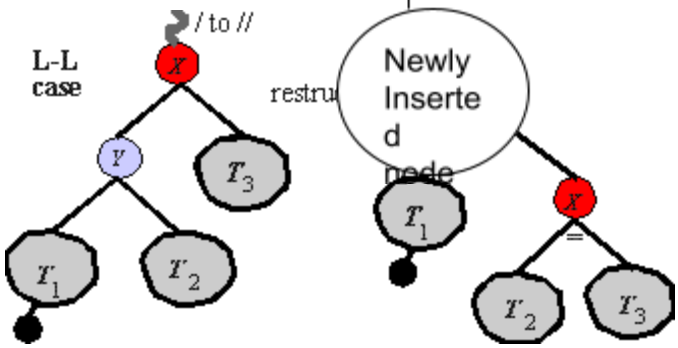
fig: Left Rotated Balanced tree of fi

38.ii.What are the steps t e an unbalanced tree into a Balan me? Explain.

Balanced tree is one in wh 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 convert an un ree into a balanced one , the given previous condition must be fulfilled. chieved by g the process of rotation.

n can be of
a) heig 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?

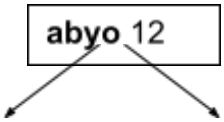
The Huffman algorithm is the basic data compression algorithm that is used to compress text files. The basic principle of this algorithm is the using the shortest symbol for the large occurring letter. This algorithm encodes the text by constructing the tree, which consists of a node having the frequency 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:



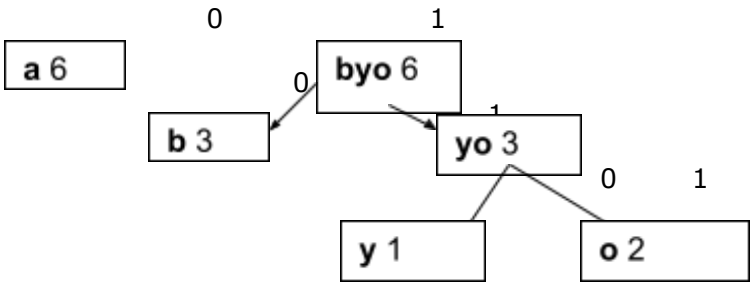


Figure Binary tree

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.