25/7/18
Wednesday

Data Structure

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Basically, data structure is a logical or mathematical model of a particular organization of data.

Data Structure also defines the way in which data is efficiently stored, processed and retrieved from the computer's memory.

The choice of a particular data structure model depends on 2 considerations -

1. It must be rich enough in structure to mirror the actual relationships of the data in the real world.
2. The structure should be simple enough that one can effectively process the data when necessary.

(III) Need of Data Structure :-

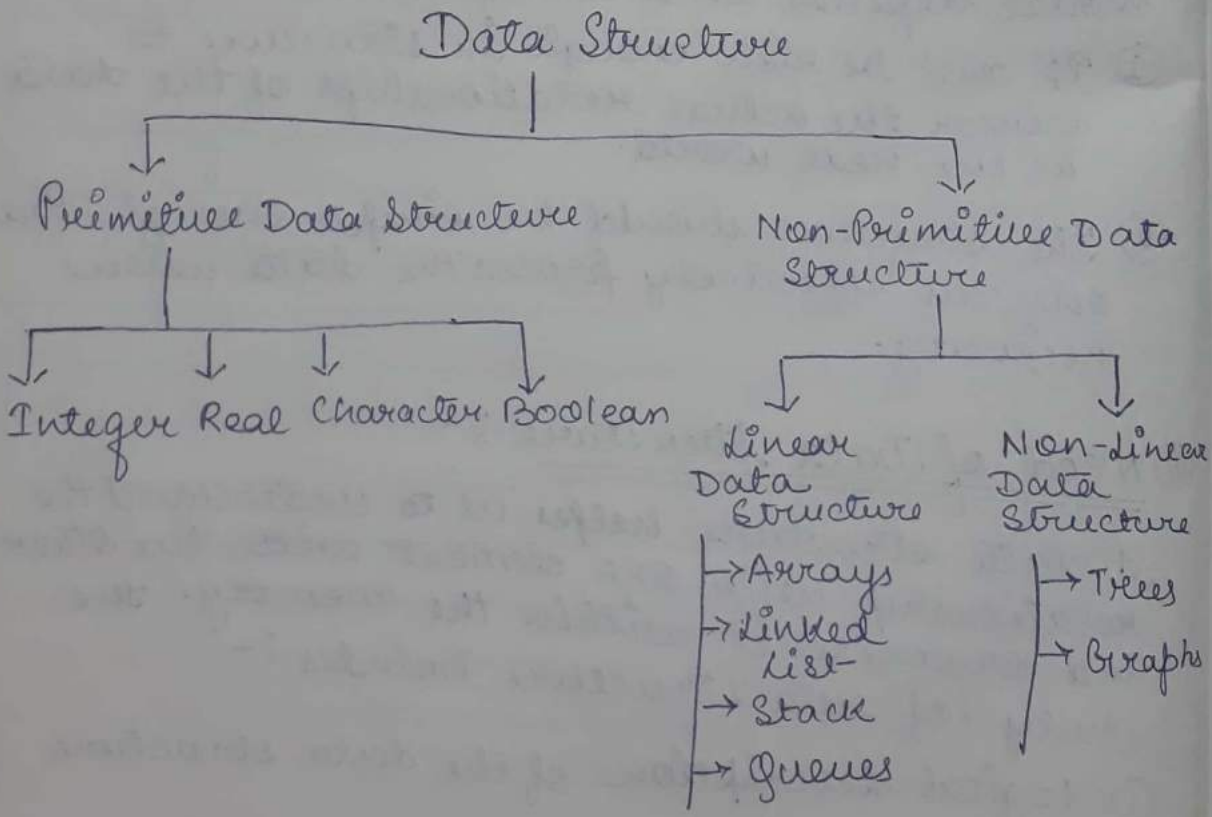
A data structure helps us to understand the relationship with one element with the other and organise it within the memory. The study of data structure includes :-

1. Logical description of the data structure
2. Implementation of the data structure
3. Quantitative analysis of data structure which includes determining the amount of memory needed to store the data structure & the time required for processing it.

Data structures are applied extensively in the following areas :-

1. DBMS
2. Compiler Design
3. Network
4. Numerical Analysis
5. Artificial Intelligence
6. Statistical Analysis Package
7. Simulation
8. Operating System
9. Graphics

(III) Types of Data Structure :-



(III) Data Structure Operations :-

In data structure the following operations are used to process data :-

(1) Creating → This is the first operation to create a data structure. This is just declaration and initialization of the data structure and reserved memory locations for data elements.

(2) Inserting → Adding new records to the structure.

(3) Deleting → Removing a record from the structure.

(4) Updating → It changes data values of the data structure.

(5) Traversing → Accessing each record exactly once so that certain items in the record may be processed.

(6) Searching → Finding the location of the record with a given key value.

(7) Sorting → Arranging the data elements in some logical order (ascending or descending order).

(8) Merging → Combining the data elements in two different sorted sets into a single sorted set.

(9) Destroying → This must be the last operation of the data structure and apply this operation when no longer needs of the data structure.

27/7/18
(Friday)

(III) Algorithm :-

An algorithm is a step by step finite sequence of instructions to solve a well defined computational problem ie; in practice to solve any real life problems, first we have to define the problems.

30/7/18
(Monday)

And then choose the better way to solve the problem from various alternative solutions.

Basically, an algorithm takes zero or more set up values as input and produces a value or set up values as output.

Functional algorithm representation of an algorithm:

Problem
↓
Algorithm
↓
Input → [Computer] → Output.

29/9/18
(Saturday)

(III) Algorithm Analysis :-

Algorithms are designed using basic programming structure structure such as :-

1. Sequence
2. Decision
3. Repeatation

It is very common that to solve a problem more than one algorithms are available as solution. But is very essential to choose very good and appropriate algorithm and which will be the most effective approach of solution to solve a problem. So in that case analysing of algorithms is the best way to select the perfect algo for a particular problem.

Analysing an algorithm has two approaches. The first one is to check the correctness of the algorithm and the second one is to check the simplicity of the algorithm. It is right that correct and simpliest is not the best one always but which one the best algo - that depends on another factor which is known as complexity of algorithm. Now question is what is complexity of an algorithm.

(III) Complexity :-

Complexity is a function of the amount of input data (input size or problem size) of the algorithm. Complexity measures how much resources is required for execution of the algorithm ie; complexity is nothing but the measurement of efficiency of an algorithm.

Complexity functions are designed to measure two parameters :-

(a) Space Complexity
(b) Time complexity.

31/7/18
(Tuesday)

(III) Stack :-

A stack is a non-primitive linear data structure, in which both Insertion and Deletion operations are performed only at one end which is called the top of the stack. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

3/8/18 (Friday)

(III) Operations on Stack :-

The basic operations that can be performed on stack are as follows :-

(1) PUSH :→ This process is used to add a new element to the top of stack. After pushing a new element at the top position, the value of top will be incremented by 1. That means after every successful PUSH operation, top is incremented by 1.

In case the array is full and no new element can be accomodated, it is called Stack full condition or stack overflow.

(2) POP :→ The process of deleting an element from the top of stack is called POP operation. After every successful POP operation the value of top will be decremented by 1. If there is no element in the stack and the POP operation is performed then this condition is known as stack underflow condition.

(III) Stack Terminology :-

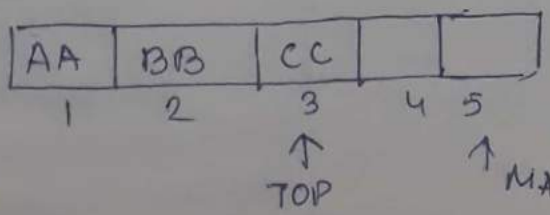
- **Context** :- The environment in which a function executes : includes argument-values, local variables and global variables.
- **Stack Frames** :- The data structure containing all the data needed each time a procedure / a function is called.
- **Max Size** :- This term is not a standard one, we use this term to refer to the maximum size of Stack.
- **Top** :- In stack top is a very important variable because the stack top is used to check stack overflow or underflow conditions. Initially top stores 0. When element is inserted then top is incremented by 1 and when element is deleted top is decremented by 1. Top is also known as **Top of Stack (TOS)**.
- **Stack** :- It is an array of size max size.
- **Stack Empty / Underflow** :- This is the situation when the stack contains no elements.
- **Stack Overflow** :- This is a situation when the stack becomes full & no more elements can be pushed on to the stack.

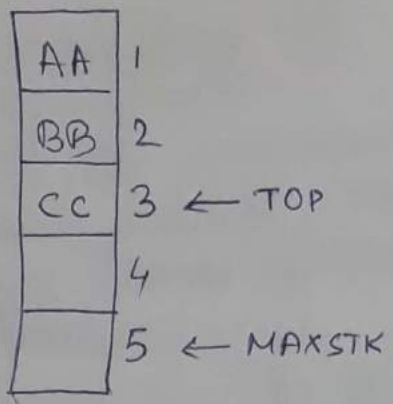
(III) Array Representation of Stack :-

Stack may be represented in the computer memory in various method, usually by means of a 1 way list or a linear array. In case of linear array STACK, a pointer variable TOP which contains the location of the top element of the stack and a variable MAXSTK which gives the maximum no. of elements that can be held by the Stack. The condition TOP = 0 or TOP = Null will indicate that the stack is empty.

6/8/18 (Monday).

The array representation of a stack is as follows :-





In this stack Top = 3, that means already 3 elements are present in the stack — AA, BB, CC Since MAX = 5, there is room for 2 more elements in the stack.

At this situation, user can implement PUSH operation as well as POP operation.

Algorithm for PUSH operation :-

PUSH (STACK, TOP, MAXSTK, ITEM)
This procedure Pushes ITEM onto a stack.

1. [Stack already Filled?]
   If TOP = MAXSTK, then: Print: OVERFLOW and Return
2. Set TOP = TOP + 1 [Increase TOP by 1].
3. Set STACK [TOP] = ITEM [insert ITEM in new TOP position]
4. Return.

(II) Algorithm for POP operation :-

POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ~~item~~ ITEM.

1. [Stack has an element / ITEM to ~~the room~~ removed ?]
   If TOP = 0, then, Print : UNDERFLOW and Return
2. Set ITEM = STACK [TOP]. [Assigns TOP element to ITEM]
3. Set TOP = TOP - 1 [Decreases TOP by 1]
4. Return.

(III) Application of stack :-

1. Evaluation of arithmatic expression
2. Converting of infix expression to postfix expression.
3. Converting of infix expression to prefix expression.
4. Evaluation of postfix expression.
5. Evaluation of prefix expression
6. Implementation of recursion expression.

(III) Evaluation of Arithmatic expression :-

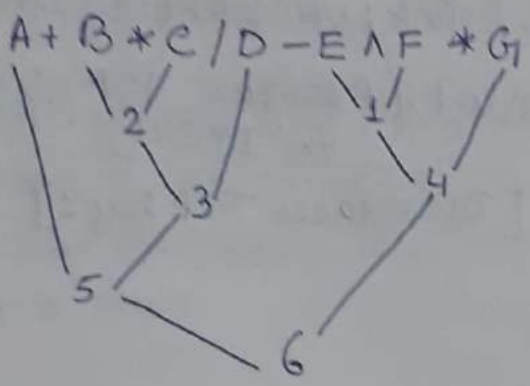
(i) Operator Precedence :-

Operator	Precedence	Associativity
-(unary), +(unary), NOT	6	
^ (Exponential)	6	Right to left
*, /	5	Left to right
+, -	4	Left to right
<, >, <=, >=, !=	3	Left to right

AND	2	Left to right
OR, XOR	1	Left to right

7/8/18
(Tuesday)

Example 1



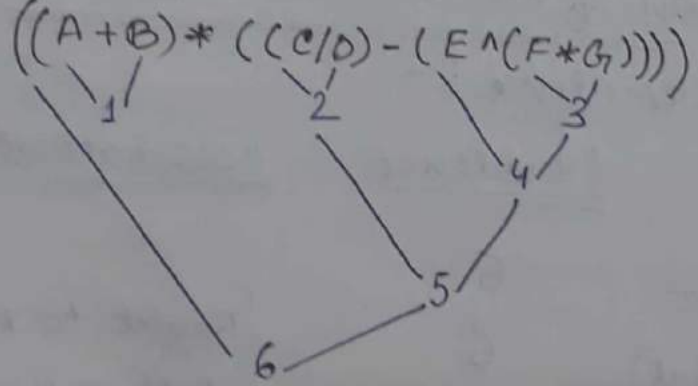
Ex 2:

Evaluate the following mathematical expressio

(a) 2^4+6*2^2-12/4

⇒
1. 2^4+6*4-12/4
2. 16+6*4-12/4
3. 16+24-12/4
4. 16+24-3
5. 40-3
6. 37

Ex 3:-



(IIII) Notation for arithmatic Expression :-

There are 3 notation to represent arithmatic expression :-

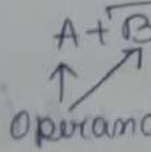
(1) Infix Notation
(2) Prefix Notation
(3) Postfix Notation

(I) Infix Notation :-

It is conventional way of writing arithmatic expression. Here ~~+ - * /~~ ^ notation is written as :-

<Operand> <operator> <operand>

Eg :-



, B/C , A*C

(II) Prefix Notation :-

In prefix notation, operator comes before the operands. The notation is written as :-

<Operator> <operand> <Operand>

Eg :-

+AB , *EF , /BC

(III) Postfix Notation :-

In postfix notation the operator come after the operands. This notation is also called suffix notation or Reverse Polish Notation. This notation is written as :-

<Operand> <Operand> <Operator>

Eg :-

AB+ , BD/ , EF*

(III) Notation Conversion :-

Conversion of Infix expression to postfix expression

Eg4:-

$A+[(B+C)+(D+E)*F]/G_1$

$\Rightarrow A+[(BC+)+(DE+)*F]/G_1$ [let $(BC+)=X$ & $(DE+)=Y$]

$\Rightarrow A+[X+Y*F]/G_1$

$\Rightarrow A+[X+YF*]/G_1$

$\Rightarrow A+[X+Z]/G_1$ [let $(YF*)=Z$]

$\Rightarrow A+[XZ+]/G_1$

$\Rightarrow A+W/G_1$ [let $[XZ+]=W$]

$\Rightarrow A+WG_1/$

~~$\Rightarrow AWG_1/+$~~

$\Rightarrow A+M$ [let $M=WG_1/$]

$\Rightarrow AM+$

$\Rightarrow AWG_1/+$

$\Rightarrow$ ~~$AXZG_1/+$~~ $AXZ+G_1/+$

$\Rightarrow AXYF*+G_1/+$

$\Rightarrow ABC+DE+F*+G_1/+$

Eg5:-

$A+B-C$

$\Rightarrow AB+-C$ [let $AB+=X$]

$\Rightarrow X-C$

$\Rightarrow XC-$

$\Rightarrow AB+C-$

Eg 6 :-

$A*B+C|D$

~~$\Rightarrow AB*$~~

~~$\Rightarrow (A*B)+(C|D)$~~

~~$\Rightarrow AB*$~~

$\Rightarrow (A*B)+C|D$

$\Rightarrow (AB*)+C|D$ [Let $T=AB*$]

$\Rightarrow T+C|D$

$\Rightarrow T+CD|$ [Let $S=CD|$]

$\Rightarrow T+S$

$\Rightarrow TS+$

$\Rightarrow AB*CD|+$

~~0 8 9~~

~~(Excellent)~~

Eg 7 :-

$(A+B)*C|D+E\wedge F|G$.

$\Rightarrow (AB+)*C|D+E\wedge F|G$ [Let $AB+=X$]

$\Rightarrow X*C|D+E\wedge F|G$

$\Rightarrow X*C|D+(EF\wedge)|G$ [Let $EF\wedge=Y$]

$\Rightarrow X*C|D+Y|G$ [Let $X*C=Z$]

$\Rightarrow (Z*)|D+Y|G$

$\Rightarrow (Z*D|)+Y|G$ [Let $Z*D|=P$]

$\Rightarrow P+Y|G$ [Let $Y|G=Q$]

$\Rightarrow P+QG|$

~~$\Rightarrow P+Q$~~ [Let $P+Q=R$]

~~$\Rightarrow PQ+$~~

$\Rightarrow Z*D|Y|G+$

$\Rightarrow X*C*D|EF\wedge G$.

10/8/18
(Friday)

Eg:-

$(A+B) * C|D+E\wedge F|G$.

$= (AB+) * C|D+E\wedge F|G$.

$= X * C|D+E\wedge F|G$. [Let $AB+ = X$]

$= X * C|D+(EF\wedge)|G$

$= X * C|D+Q|G$ [Let $EF\wedge = Q$]

$= (XC*)|D+Q|G$.

$= Z|D+Q|G$ [Let $XC* = Z$]

~~$= P+Q|G$~~ ~~[Let $Z|D = P$.~~

~~$= P$~~

$= ZD| +Q|G$

$= P+Q|G$ [Let $ZD| = P$]

$= P+QG|$

$= P+R$ [$QG| = R$]

$= PR+$

$= PQG|+$

$= ZD|QG|+$

$= XC*D|QG|+$

$= XC*D|EF\wedge G|+$

$= AB+C*D|EF\wedge G|+$

Eg 8

$A+(B*C-(D/E\wedge F)*G)*H$

$= \cancel{A+(B*C-(D/EF\wedge)}$

$= A+(B*C-(D/EF\wedge)*G)*H$ [Let $EF\wedge = S$]

$= A+(B*C-(D/S)*G)*H$

$= A+(B*C-(DS/)*G)*H$ [Let $DS/ = Q$]

$= A+(B*C-Q*G)*H$

$= A+(BC*-Q*G)*H$ [Let $BC* = R$]

$= A+(R-Q*G)*H$

$= A+(R-QG*)*H$ [Let $QG* = P$]

$= A+(R-P)*H$

$= A+(RP-)*H$ [Let $RP- = T$]

$= A+T*H$

$= A+TH*$ [$TH* = X$]

$= A+X$

$= \cancel{A+TH}$

$= AX+$

$= ATH*+$

$= ARP-H*+$

$= ARQG*-H*+$

$= ABC*QG*-H*+$

$= ABC*DS/G*-H*+$

$= ABC*DEF\wedge/G*-H*+$

(III) Infix to Prefix Conversion :-

(1) $A*B+C$

$= (A*B)+C$

$= (*AB)+C$ [Let $*AB = T$]

$= T+C$

$= +TC$

$= +*ABC$

(2) $A/B\wedge C+D$

$= A/(B\wedge C)+D$

$= A/(\wedge BC)+D$

$= A/P+D$ [Let $\wedge BC = P$]

$= (/AP)+D$

$= Q+D$ [Let $/AP = Q$]

$= +QD$

$= +/APD$

$= +/A\wedge BCD$

(3) $A+(B*C-(D/(E\wedge F)*G)*H$

$= A+(B*C-(D/(E\wedge F))*G)*H$

$= A+(B*C-(D/(\wedge EF))*G)*H$

$= A+(B*C-(D/P)*G)*H$ [Let $\wedge EF = P$]

$= A+(B*C-(/DP)*G)*H$

$= A+(B*C-Q*G)*H$ [Let $/DP = Q$]

$= A+(*BC-Q*G)*H$

$= A+(R-Q*G)*H$ [Let $*BC = R$]

$= A+(R-*QG)*H$

$= A+(R-S)*H$ [Let $*QG = S$]

$= A + (-RS) * H$

$= A + T * H$ [Let $-RS = T$]

$= A + (*TH)$.

$= A + M$. [Let $*TH = M$]

$= +AM$.

$= +A*TH$

$= +A*-RSH$

$= +A*-R*QGH$.

$= +A*-*BC*QGH$.

$= +A**\bar{*}BC*/DPGH$.

$= +A**\bar{*}*BC*/D\wedge EFGH$

(III) Algorithm to convert Infix expression to postfix expression :-

Let Q be an arithmatic Expression written in infix notation. The following algorithm transforms the infix ~~expression~~ expression Q into its equivalent postfix expression P.

POLISH (Q, P)

1. Push "(" onto STACK and add ")" to the end of Q.
2. Scan Q from left to right and Repeat Steps 3 to 6 For each element of Q until the STACK is empty.
3. If an operand is encountered, add it to P
4. If a left paranthesis is encountered, Push it onto STACK.
5. If an operator $\otimes$ is encountered, then :

a) Repeatedly Pop from STACK and add to P each operator (On the top of STACK) which has the same precedence as or higher precedence than ⊗.

b) Add ⊗ to the STACK.

[End of if structure]

6. If a right paranthesis is encountered, then:

a) Repeatedly Pop from STACK and add to P each operator (on the top of STACK) until a left Paranthesis is encountered.

b) Remove the left paranthesis. [DO not add the left paranthesis to P]

[End of If structure]

[end of Step 2 loop]

7. Exit

(III) Consider the following arithmatic Infix expression Q: A+(B*C-(D/E^F)*G)*H.

Convert Q to its equivalent Polish notation or postfix expression using STACK table.

⇒ According to algorithm at first Push "(" onto STACK and then add ")" to the end of Q. Now the sequence of scanned symbol of Q is —

A	+	(	B	*	C	-	(	D	/	E	^	F	)	*	G	)	*	H	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Now the tabular format of STACK Status of conversion of each scanned symbol of Q to P.

Symbol Scanned	STACK	Expression P
	(	
1. A	(	A
2. +	(+	A
3. (	(+(	A
4. B	(+(	A B
5. *	(+(*	A B
6. C	(+(*	A B C
7. -	(+(-	A B C *
8. (	(+(-(	A B C *
9. D	(+(-(	A B C * D
10. /	(+(-(/	A B C * D
11. E	(+(-(/	A B C * D E
12. ^	(+(-(/^	A B C * D E
13. F	(+(-(/^	A B C * D E F
14.)	(+(-	A B C * D E F ^ /
15. *	(+(-*	A B C * D E F ^ /
16. G	(+(-*	A B C * D E F ^ / G
17.)	(+	A B C * D E F ^ / G * -
18. *	(+*	A B C * D E F ^ / G * -
19. H	(+*	A B C * D E F ^ / G * - H
20.)		A B C * D E F ^ / G * - H * +

H.W

1) $A + [(B+C) + (D+E) * F] / G$

Symbol	STACK	EXPRESSION
	(	
1. A	(	A
2. +	(+	A
3. [	(+[	A
4. (	(+[(	A
5. B	(+[(	AB
6. +	(+[(+	AB
7. C	(+[(+	ABC
8.)	(+[	ABC+
9. +	(+[+	ABC+
10. (	(+[+(	ABC+
11. D	(+[+(	ABC+D
12. +	(+[+(+	ABC+D
13. E	(+[+(+	ABC+DE
14.)	(+[+	ABC+DE+
15. *	(+[+*	ABC+DE+
16. F	(+[+*	ABC+DE+F
17.]	(+	ABC+DE+F*+
18. /	(+/	ABC+DE+F*+
19. G	(+/	ABC+DE+F*+G
20.)		ABC+DE+F*+G/+

2) (A+B)*C/D+E^F/G

Symbol	STACK	Expression
	(	
1. (	((	
2. A	((	A
3. +	((+	A
4. B	((+	AB
5.)	(	AB+
6. *	(*	AB+
7. C	(*	AB+C
8. /	(/	AB+C*
9. D	(/	AB+C*D
10. +	(+	AB+C*D/
11. E	(+	AB+C*D/E
12. ^	(+^	AB+C*D/E
13. F	(+^	AB+C*D/EF
14. /	(+/	AB+C*D/EF^
15. G	(+/	AB+C*D/EF^G
16.)		AB+C*D/EF^G/+

3) A*B+C

Symbol	STACK	Expression
	(	
1. A	(	A
2. *	(*	A
3. B	(*	AB
4. +	(+	AB*
5. C	(+	AB*C
6.)		AB*C+

4) A/B^C+D

Symbol	STACK	Expression
C	C	
1. A	C	A
2. /	C/	A
3. B	C/	AB
4. ^	C/^	AB
5. C	C/^	ABC
6. +	C/+	ABC
		ABC..^
		...
7. D	C+	ABC^D/
8.)		ABC^D/+

13/8/18 (Monday)

(III) Evaluation of a Postfix expression:-

Suppose P is an arithmatic expression written in Postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

1. Add a right Parenthesis ")" at the end of P [this acts as a sentinel]
2. Scan P from left to right and repeat Step 3, and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator ⊗ is encountered, then:
   a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   b) Evaluate B ⊗ A

c) Place the result of (b) back on STACK
[End of if structure]
[End of step 2 loop]

5. Set VALUE equal to the top element on STACK.

6. Exit

Eg

Consider the following arithmatic expression P written in postfix notation :-

P: 5, 6, 2, +, *, 12, 4, /, -

Ans:- At first we add a sentinel right parantheis at the end of P to obtain the sequence of scanned element

P: 5 6 2 + * 12 4 / -)
1 2 3 4 5 6 7 8 9 10

Step	Scanned element	Operation	Stack	Result
1.	5	Push 5	5	
2.	6	Push 6	5, 6	
3.	2	Push 2	5, 6, 2	
4.	+	Pop 2, 6	5	6+2 = 8
		Push result 8	5, 8	
5.	*	Pop 8, 5	#empty	5 * 8 = 40
		Push 40	40	
6.	12	Push 12	40, 12	
7.	4	Push 4	40, 12, 4	
8.	/	Pop 4, 12	40	12/4 = 3
		Push 3	40, 3	
9.	-	Pop 3, 40	#empty	40 - 3 = 37
		Push 37	37	
10.	)	Pop 37	#empty	37

QUEUE

14/8/18 (Tuesday)

Queue is a non-primitive linear data structure. It is a homogeneous collection of elements, in which new elements are added at one end called the Rear end and the existing elements are deleted from other end called the Front end.

Logically queue supports FIFO (First In First Out) process. That means the first element in the queue will be the first element out of the queue. In other words, the order in which elements enter ~~in QUEUE~~ a queue is the order in which they leave.

(III) Representation of queue :-

In computer queue is representated in 2 ways:

(I) Using linear array
(II) Using linked list

(III) Array representation of queue :-

In data structure queue can be implemented by using linear array. Each queue has 2 pointer variables :-

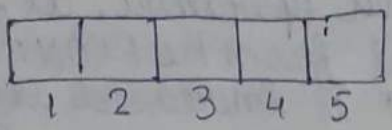
(I) FRONT - which contains the location of the front element of the queue.

(II) REAR - it contains the location of the REAR element of the queue.

17/8/18
(Friday)

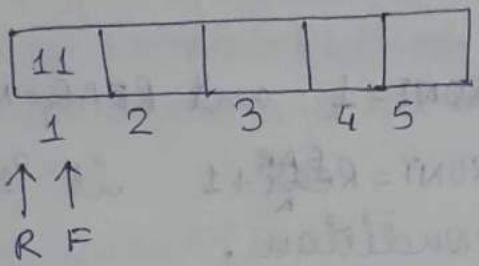
Now consider the following diagram to understand the queue status:-

1. Rear = 0
   Front = 0



Queue is empty.

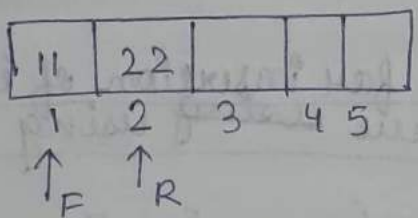
2. Rear = 1.
   Front = 1



R → Rear
F → Front

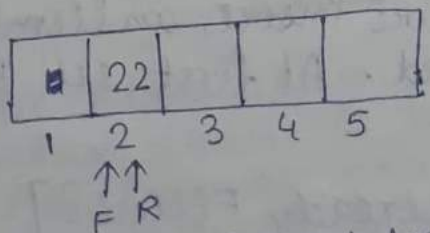
Queue with one element-
11 is inserted at Rear end

3. Rear = 2
   Front = 1



Queue with two elements

4. Rear = 2
   Front = 2



~~One~~ element 11 is deleted from Frontend

- It is clear that during insertion time at first the value of REAR is increased by 1 ie;
  REAR = REAR + 1 then the new value is inserted at the position of REAR.

- When first element is inserted into the queue then

  REAR = FRONT = 1

- During deletion operation, at first the item is deleted from the FRONT end and then value of FRONT is increased by 1 ie;

  FRONT = FRONT + 1

- Suppose FRONT = 1 and REAR = MAX or FRONT = REAR + 1 it is called overflow condition.

- Suppose FRONT = NULL or 0 it is called Underflow

(III) Algorithm for insertion of items in the linear queue using array:-

Queue - Insert (QUEUE [MAX], ITEM, FRONT, REAR)

This procedure inserts an item into the queue at REAR end. At first set REAR = 0 and FRONT = 0.

1. [Queue already Filled?]

   If FRONT = 0 and REAR = MAX or

   if FRONT = REAR + 1 then

   print: OVERFLOW and Return

2. If FRONT = 0 and REAR = 0 then set FRONT = REAR

   else

   REAR = REAR + 1

   [End of If structure]

3. Set QUEUE[REAR] = ITEM

4. Return

(ii) Algorithm for deletion :

Queue - Delete (QUEUE[MAX], ITEM, FRONT, REAR)

This algorithm deletes an item from the linear queue at FRONT end.

1. [Queue already empty ?]
   if FRONT = 0 or FRONT > REAR then
   Print : UNDERFLOW and Return
   else
   Set ITEM = QUEUE[FRONT]
   Set FRONT = FRONT + 1
   [End of if structure]

2. Return

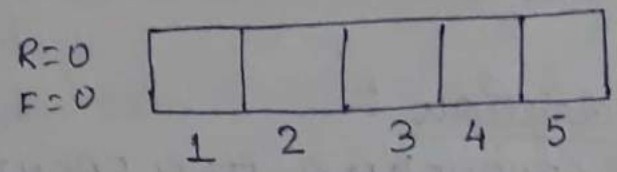
21/8/18 (Tuesday)
(~~Monday~~)

(ii) Limitations of linear queue :

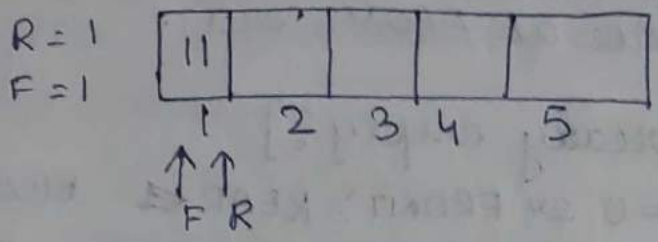
There are certain problems associated with a simple queue when queue is implemented using arrays. Consider an example of a linear queue Q(5) which is initially empty. We analyse the problem with a series of ~~instructions~~ insertion & deletion operations performed on the queue. ~~The insertion & deletion operations performed on the queue.~~

The insertion & deletion process are shown in the following figures :-

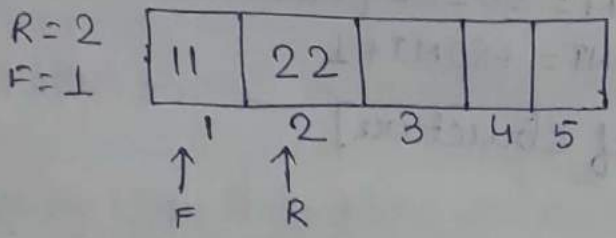
Q(5)



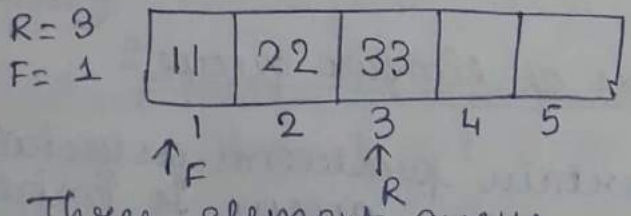
(a) Initial Queue



(b) One element queue.

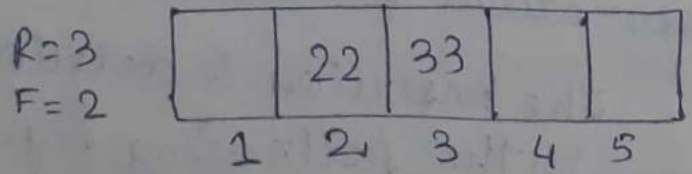


(c) Two element queue.

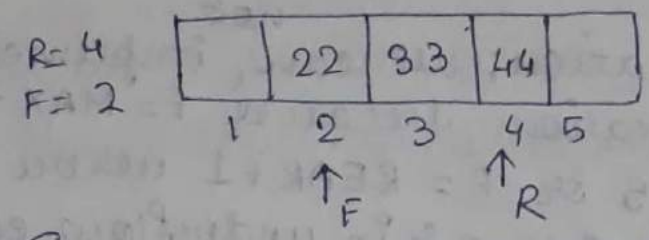


(d) Three element queue

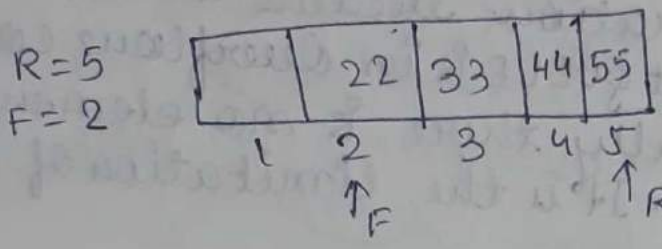
Upto that there is no problem. Till now we can ~~apply~~ implement insertion & deletion process.



(e) After delete one element

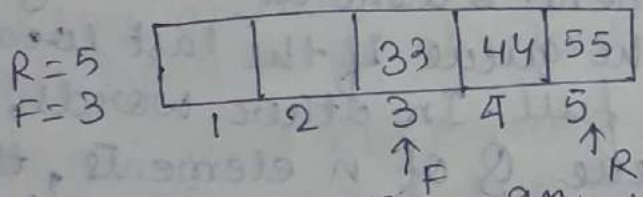


(f) Three element queue after inserting one element.

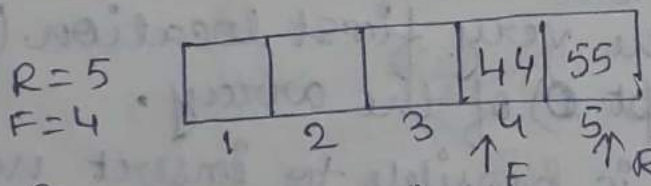


(g)

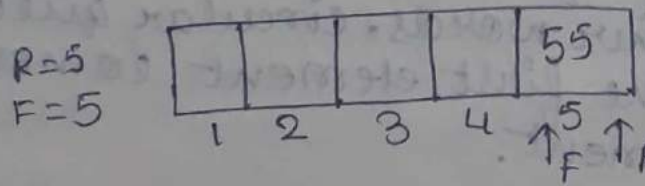
At that position queue is full because REAR=5. We cannot implement insertion but we can implement deletion operation.



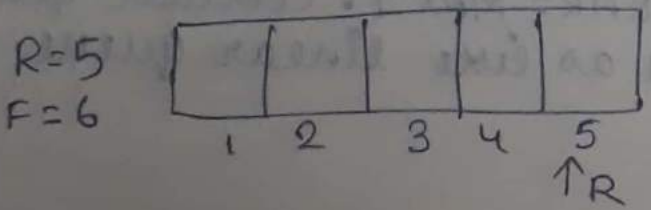
(h) After deleting ~~one~~ an element



(i) After deleting an element



(j) Again deleting an element



At that situation, we can not implement deletion operation because F = MAX + 1 = 6 where MAX = 5 or F = REAR + 1 where REAR = 5, that means QUEUE is in underflow condition.

Besides that we cannot implement Insertion operation because R = MAX = 5 which indicates that QUEUE is in overflow condition but practically there is no element in that QUEUE. It is the limitation of linear queue.

24/8/18 (Friday)

Circular Queue :-

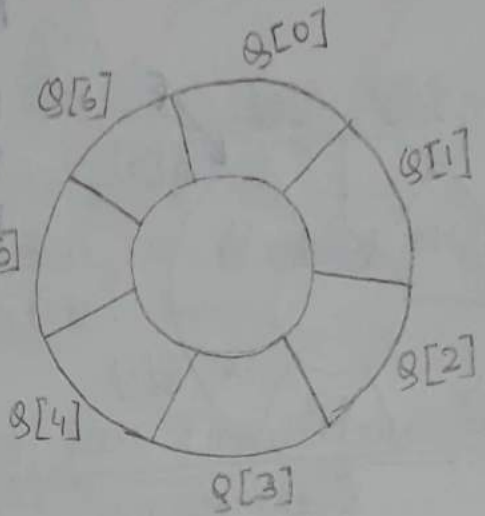
A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location of the queue is full. In other words, suppose there is a queue Q of n elements, then after inserting an element at the last location (n-1) of the array, the next element will be inserted at the very first location (location with subscript 0) of the array.

It is possible to insert new eleme if and only if those locations / slots are empty. In other words, circular queue is one in which the first element comes just after the last element.

The circular queue will be full when FRONT = 0, REAR = MAX-1. Circular queue is implemented as like linear queue but

difference in the process of insertion, ~~dation~~ deletion in the circular queue.

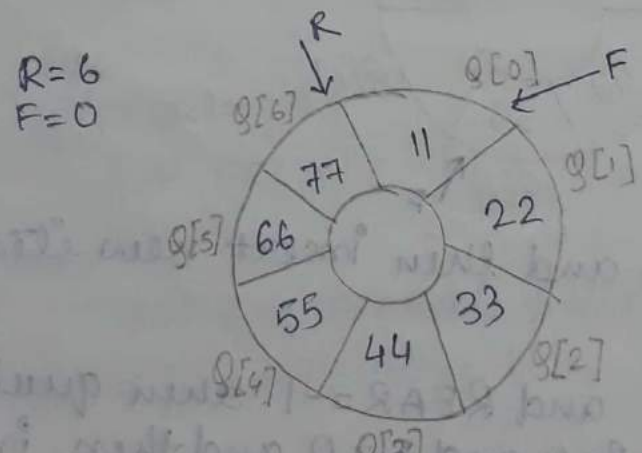
consider the following conceptual diagram:



Circular Queue.

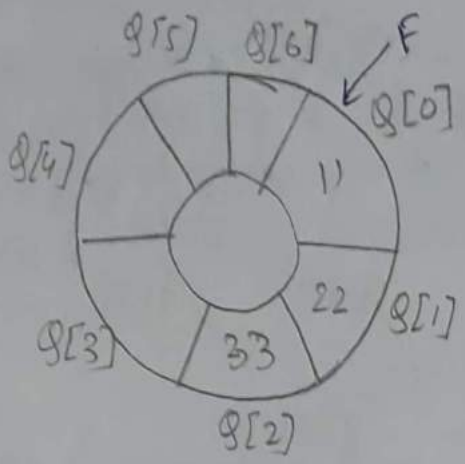
If we want to insert an element into the circular queue, the following 4 conditions have to be checked :-

(1) If FRONT = 0 and REAR = MAX-1 then print: The circular queue is full. Item cannot be inserted.



(2) If FRONT = 0 and REAR ≠ MAX-1 then the value of REAR will be incremented by 1 ie; REAR = REAR + 1 and then item is inserted at REAR.

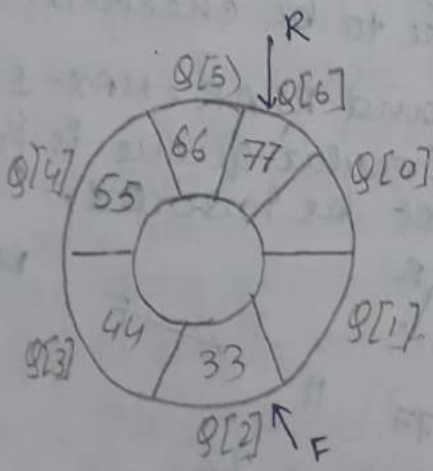
MAX = 7



Here REAR can be anywhere b/w Q[0] to Q[5]

(3) If FRONT ≠ 0 and REAR = MAX-1 then QUEUE is not full. Reset REAR = 0 and then insert new item.

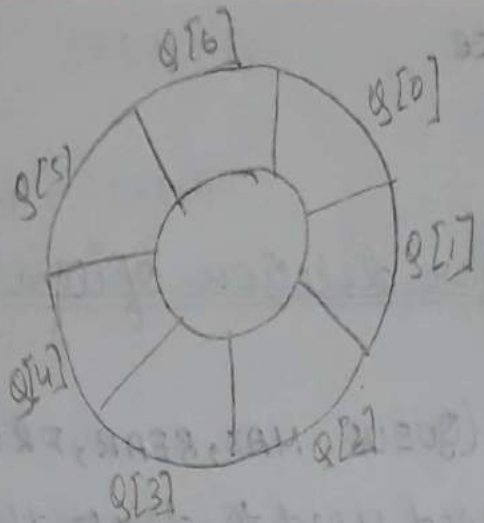
R = 0
R = 2.



Reset R = 0 and then insert new item

(4) If FRONT = -1 and REAR = -1 then queue is empty. Set R = 0 and F = 0 and then insert new item at REAR position

R=-1
F=-1



Reset R = 0, F = 0 and then insert new item at R.

(III) Algorithm for insertion in a circular Queue : (using array)

#Queue_insert (QUEUE, MAX, REAR, FRONT, ITEM)

The algo is used to insert the ITEM at the position of REAR.

1. [QUEUE already filled ?]
   if [FRONT == (REAR+1) % MAX)
      Print : Queue OVERFLOW and Return

   else
   if (FRONT == -1)
      Set FRONT = REAR = 0

   else
      REAR = ((REAR+1) % MAX)
   QUEUE [REAR] = ITEM

2. Exit

~~# Queue - delete~~

~~Algorithm~~.

(III) Algorithm for deletion of item in a circular queue :-

Queue - delete (QUEUE, MAX, REAR, FRONT, ITEM)

This algo is used used to delete the ITEM from the position of FRONT.

1. ~~If~~ [Queue already empty ?]

   If (FRONT == -1)

      Print : Queue Empty and Return

   else

      ITEM = QUEUE [FRONT]

   if (FRONT == REAR)

      Set FRONT = REAR = -1

   else

      FRONT = ((FRONT + 1) % MAX)

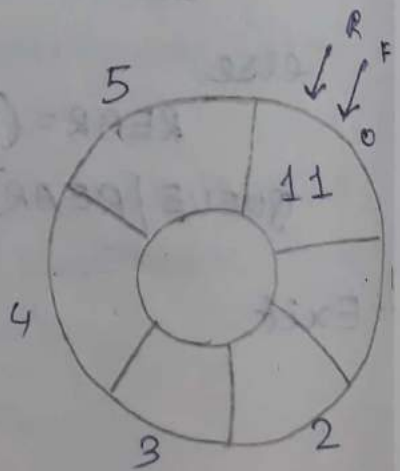
   [End of if structure]

2. Exit

(IIII) Trace Of Algorithm :-

(1) REAR = -1 ITEM = 11
FRONT = -1

REAR = (REAR+1) % MAX
= (-1+1) % 6
= 0 % 6
= 0



(2) REAR = 0 ITEM = 22

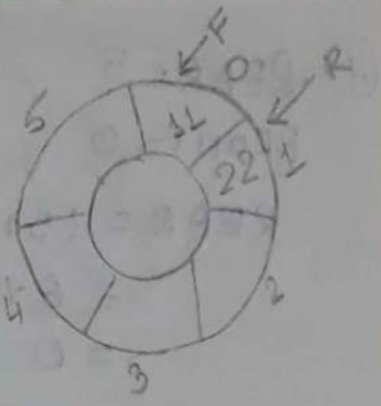
FRONT = 0

REAR = (REAR+1) % MAX

= (0+1) % 6

= 1 % 6

= 1



(3) REAR = 1 ITEM = 33.

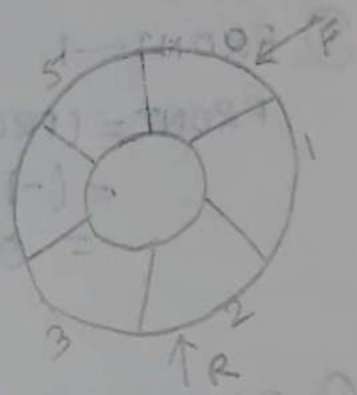
FRONT = 0

REAR = (REAR+1) % MAX

= (1+1) % 6

= 2 % 6

= 2



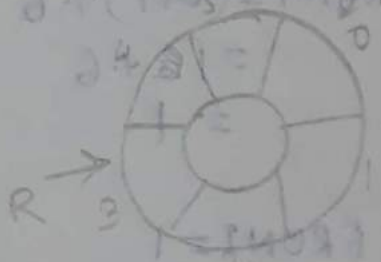
(4) REAR = 2 ITEM = 44

FRONT = 0

REAR = (2+1) % 6

= 3 % 6

= 3.



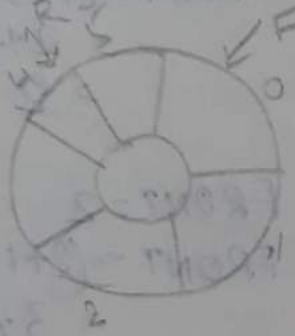
(5) REAR = 3. ITEM = 55

FRONT = 0

REAR = (3+1) % 6

= 4 % 6

= 4



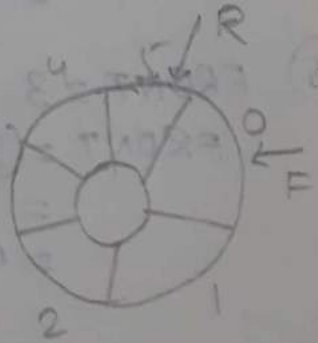
(6) REAR = 4 ITEM = 66.

FRONT = 0

REAR = (4+1) % 6

= 5 % 6

= 5



(7) REAR = 5 ITEM = 77

FRONT = 0

REAR = (5+1) % 6

= 6 % 6

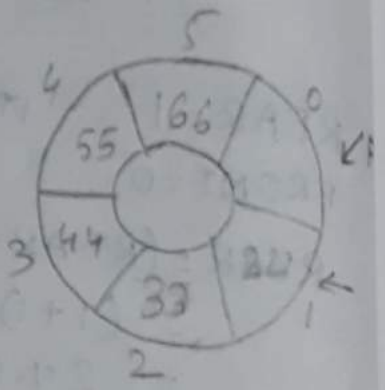
= 0

(1) FRONT = -1

FRONT = (FRONT+1) % MAX

= (-1+1) % 6

= 0

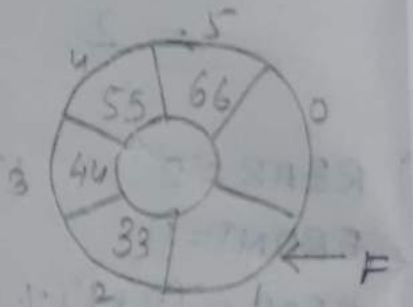


(2) FRONT = 0

FRONT = (0+1) % 6

= 1 % 6

= 1

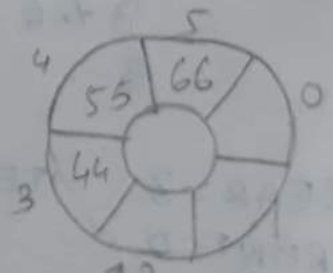


(3) FRONT = 1

FRONT = (1+1) % 6

= 2 % 6

= 2

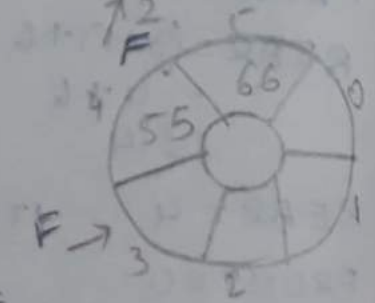


(4) FRONT = 2

FRONT = (2+1) % 6

= 3 % 6

= 3

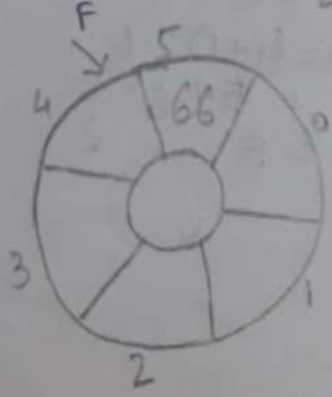


(5) FRONT = 3

FRONT = (3+1) % 6

= 4 % 6

= 4

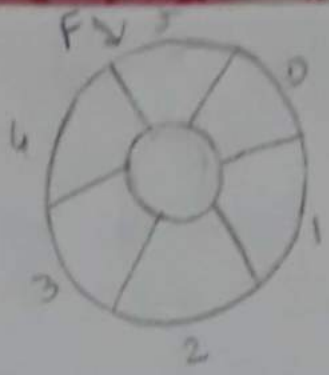


⑥ FRONT = 4

FRONT = (4+1) % 6

= 5 % 6

= 5



⑦ FRONT = 5

FRONT = (5+1) % 6

= 6 % 6

= 0

Searching

27/8/18
(Monday)

Searching is a process to find an element which is present in a list of sorted (in ascending or in descending order) or unsorted elements. Elements may be alphabetical or numerical or alpha numerical. A data structure searching is 2 types:

1. Linear Search
2. Binary Search

1. Linear Search :-

In data structure, linear search is very easy method of searching an element. In this method programmers access each element of the list one by one sequentially and then check whether it is desired element or not. Searching will be successful when the desired element is found in the list of element otherwise it is called unsuccessful search.

As the searching is done sequentially, linear search is also called sequential searching method. Linear search can be applicable for sorted as well as unsorted list of elements. In this method each and every element of the list is successfully compared to the desired element (which has to be found) and then successful searching message is displayed or not.

Linear search can be implemented by using array data structure or linked list. Consider the linear search algorithm using array :-

Linear_Search (DATA, N, ITEM, LOC)

Here DATA is a linear array with N elements and ITEM is the given element.

This algo finds the LOC of ITEM in DATA or set LOC = 0 if the search is unsuccessful

1. [Insert ITEM at the end of DATA]
   Set DATA[N+1] = ITEM
2. [Initialize counter]
   LOC = 0
3. [Search for ITEM]
   Repeat while DATA[LOC] ≠ ITEM
   Set LOC = LOC + 1
4. [End of Loop Structure]
4. [Successful ?]
   If LOC = N+1, then:
   Set LOC = -1
5. Exit

28/8/18
(Tuesday)

(III) Binary Search :-

As a searching technique, binary search algorithm is an extremely efficient algorithm. Using this algorithm the given item can be searched using minimum possible comparisons. It is very important that binary search technique can be implemented on a sorted list of elements only, that means unsorted list of elements is not applicable for binary search and for this at first you have to sort the elements using any sorting algorithm and then implement binary search to search a desired element.

~~The logic behind the binary search is as follows :-~~

~~(1) Fi~~

Binary Search algorithm can be implemented on a sorted linear array or on a sorted linked list. Assume that there is a sorted array. We want to search an element in this array. The logic behind the binary search is as follows:

(1) First find the middle element of the array. It is called mid element.

(2) Compare the mid element with the desired element (which has to be searched)

(3) Doing this comparison one of the following 3 cases can arrive :-

(a) If it is the desired element then search is successful.

(b) If it is less than desired element then it is clear that the desired element may be in the second half of the array. So search only the second half of the array.

(c) If it is greater than the desired element then it is possible that the desired element may be present in the first half of the array. So search only the first half of the array.

(*) First half of the array means —

First element or Lower Bound (LB) to Mid-1

Second half means —

Mid+1 to Last element / Upper Bound (UB)

(iii) Algorithm for Binary Search :-

Binary_Search (DATA, LB, UB, ITEM, LOC)

Here DATA is a sorted array with LB and UB and ITEM is a given item of information. The variables ~~Beg~~ BEG, END and MID denote the beginning, end and middle locations of a segment of elements of DATA respectively. This algorithm finds the LOC of ITEM in DATA or set LOC = NULL.

1. [Initialize segment's variables]
   Set BEG = LB, END = UB and ~~MID =~~
   MID = INT((BEG + END)/2)

2. Repeat step 3 and 4 while BEG <= END
   and DATA[MID] ≠ ITEM

3. If ITEM < DATA[MID], then
   Set END = MID - 1

   Else
   Set BEG = MID + 1

   [End of If structure]

4. Set MID = INT((BEG + END)/2)
   [End of step 2 loop]

5. If DATA[MID] = ITEM, then:
   Set LOC = MID

   Else
   Set LOC = NULL
   [End of If structure]

6. Exit

(11) Selection Sort :-

Selection sort is another sorting algorithm which is also called exchange sort. In this method user have to find the smallest value present in the list & this no. is put in the first position of the list. Then the second smallest no. is found among the remaining unsorted element and it is put in the second position of the list. This process is repeated upto the last element.

Procedure :-

(a) **Pass-1** :- Find the location LOC of the smallest in the list of N elements - A[0], A[1], A[2], ... A[N-1]. Then interchange A[LOC] and A[0]. Now A[0] is sorted.

(b) **Pass-2** :- Find location LOC of the smallest in the sub list N-1 element - A[1], A[2], ... A[N-1]. Then interchange A[LOC] and A[1]. Now A[1] is sorted.
Here A[0] ≤ A[1]

(c) **Pass-3** :- Find the location LOC of the smallest in the sub list N-2 element - A[2], A[3], ... A[N-1]. Now interchange A[LOC] and A[2]. So A[0], A[1], A[2] are sorted.
Here A[0] ≤ A[1] ≤ A[2]

- - - - - - - - - - - - - -

- - - - - - - - - - - - - -

Pass-(N-2) :- Find the location LOC of the smallest of the sublist A[N-2], A[N-1]. Then interchange A[LOC] and A[N-2]. Now A[0], A[1], A[2], ... A[N-1] are sorted.
Here A[0] ≤ A[2] ≤ .... ≤ A[N-1].

Then the list is sorted after (N-2) pass.

(1) Trace of algorithm:

		A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
K=0	LOC=3	(77)	33	44	(11)	88	22	66	55
K=1	LOC=5	11	(33)	44	77	88	(22)	66	55
K=2	LOC=5	11	22	(44)	77	88	(33)	66	55
K=3	LOC=5	11	22	33	(77)	88	(44)	66	55
K=4	LOC=7	11	22	33	44	(88)	77	66	(55)
K=5	LOC=6	11	22	33	44	55	(77)	(66)	88
K=6	LOC=6	11	22	33	44	55	66	77	88
		11	22	33	44	55	66	77	88

Algorithm to find smallest in the list

SMALLEST (A, K, LOC, N)

In this algorithm A is an array. This procedure finds the location - LOC of the smallest element among A[K], A[K+1], A[K+2]..... A[N]

① [initialize]
Set SMALLEST = A[K]
LOC = K

② Repeat for J = K+1 to N
if SMALLEST > A[J] then
Set SMALLEST = A[J]
Set LOC = J
[End of loop]

③ return.

Algorithm to sort elements :-

SELECTION SORT (A, N)

① Repeat Step 2 and 3 For K = 1 to N-1

② Called SMALLEST (A, K, LOC, N)

③ Inter-change A[K] with A[LOC]

Set temp = A[K]

Set A[K] = A[LOC]

Set A[LOC] = temp

[End of step1 loop]

④ Exit

31/8/18
(Friday)

TREE

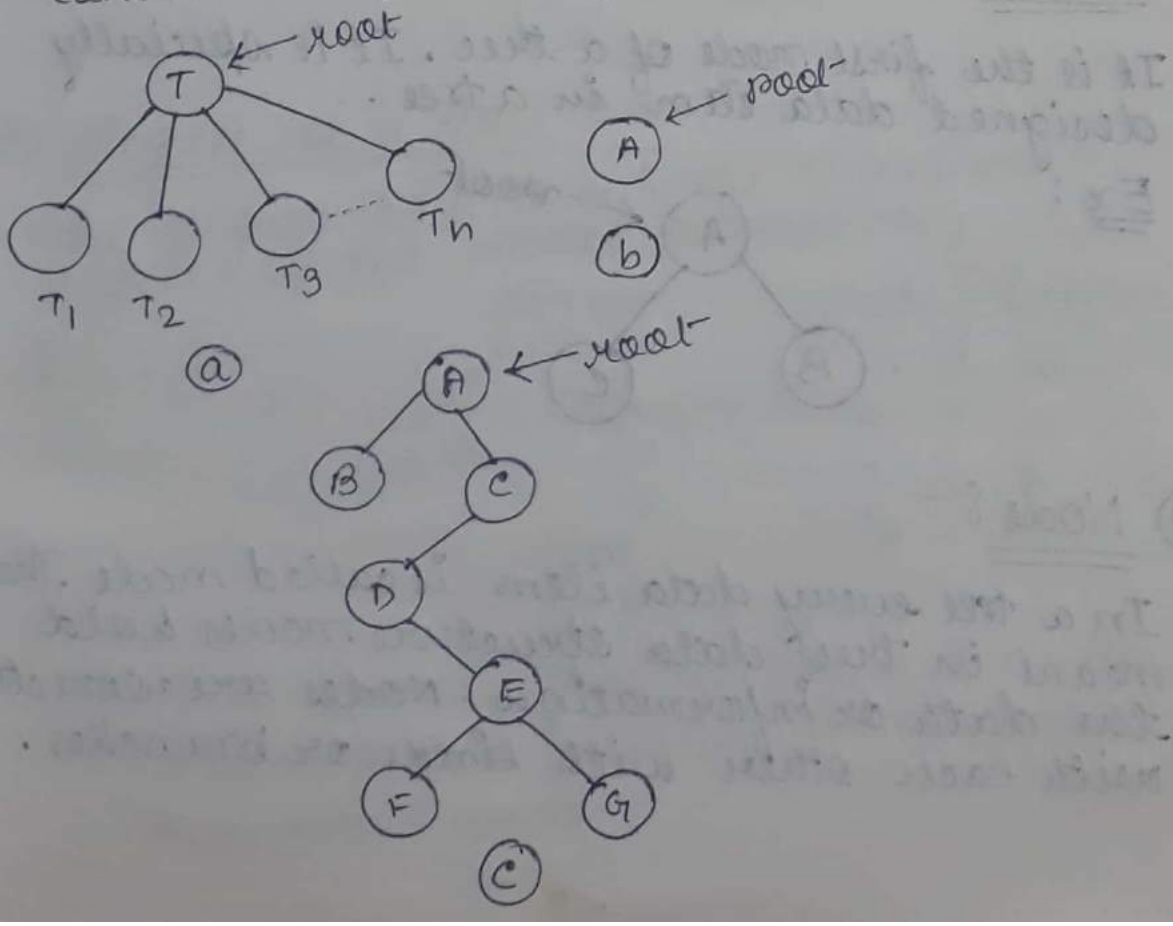
A tree is a non-linear data structure in which items are arranged in a sorted sequence. It is used to represent hierarchial relationship existing among several data items.

Basically tree is a finite set of elements which are called nodes. In case of tree data structure the followings are considerable :-

(1) There is a specially designated node known as root node.

(2) The remaining nodes of a tree are partitioned into $n > 0$ disjoint sets $T_1, T_2, T_3, \ldots, T_n$ and each of these sets is a tree once again. Suppose a tree T, so $T_1, T_2, T_3, \ldots, T_n$ are called sub tree of the tree T.

Consider the following diagram of trees



(III) Tree Terminology :-

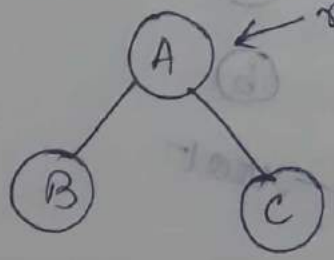
There are no. of terms associated with the Trees which are listed below :-

1. Root
2. Node
3. Degree of a node
4. Degree of a tree
5. Terminal node(s)
6. Non-Terminal nodes
7. Siblings
8. ~~to~~ level
9. Edge
10. Path
11. Depth
12. Forest

(III) Root :-

It is the first node of a tree. It is specially designed data item in a tree.

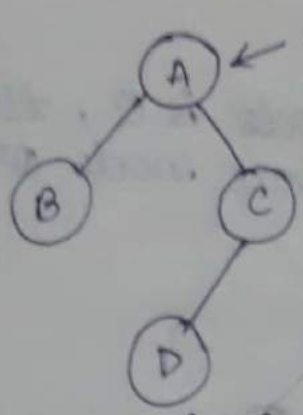
Ex :



(IIII) Node :-

In a tree every data item is called node. That means in tree data structure nodes hold the data or information. Nodes are connected with each other with links or branches.

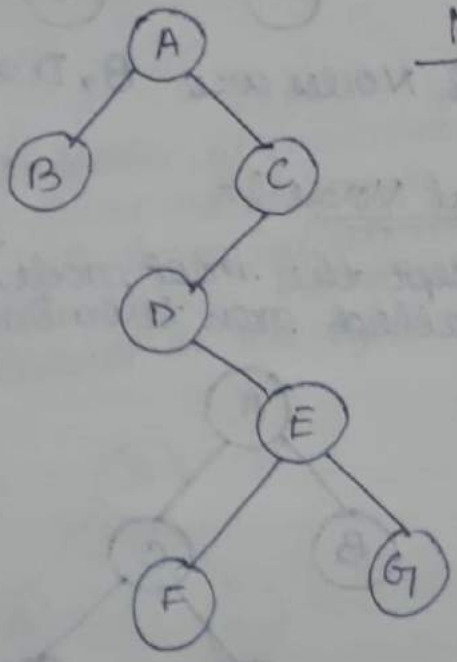
Ex



③ Degree of a node :-

The degree of a node is defined as the no. of subtrees of that node.

Ex

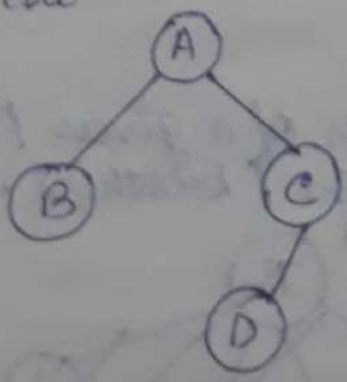


Node	Degree
A	2
B	0
C	1
D	1
E	2
F	0
G	0

(iii) Degree of a tree :-

The degree of a tree is the maximum degree of the nodes in the tree.

Ex



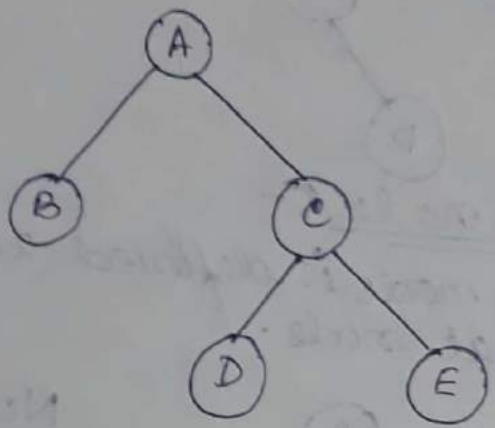
Node	Degree
A	2
B	0
C	1
D	0

So the degree of the tree is 2 as the degree of A is 2 which is maximum than other nodes.

(iii) Terminal Node :-

If the degree of a node is 0, that node is known as terminal node or leaf node.

Eg:-

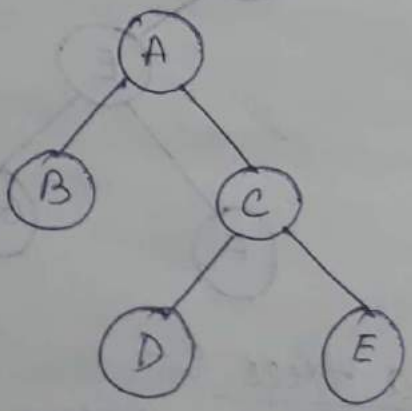


Terminal Nodes are B, D and E

(iii) Non terminal Node :-

Any node (except the root node) whose degree is not 0 is called non terminal node.

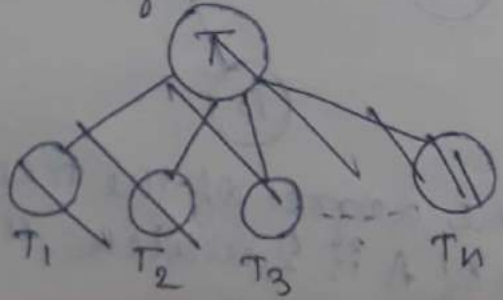
Eg:-

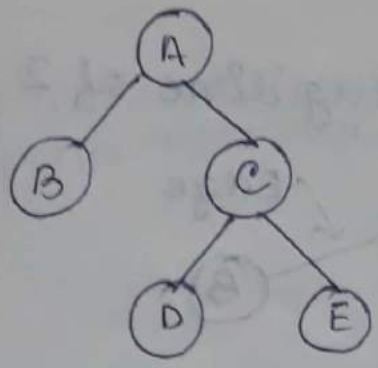


C is the non-terminal node.

(iii) Siblings :

The children nodes of a given parent node are called siblings / brothers.

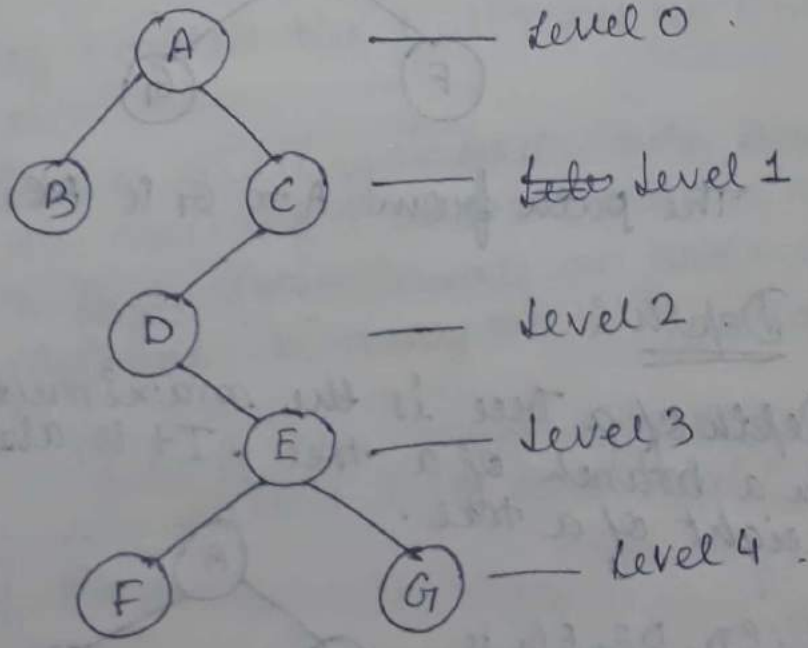




B, C are siblings as their parent is A and D, E are siblings as their parent is C.

(III) Level :-

The entire tree ~~data~~ structure is leveled in such a way that root node is always at level 0. Then its immidiate children are at level 1 and their immidiate children are at level 2 and so on upto the terminal nodes.

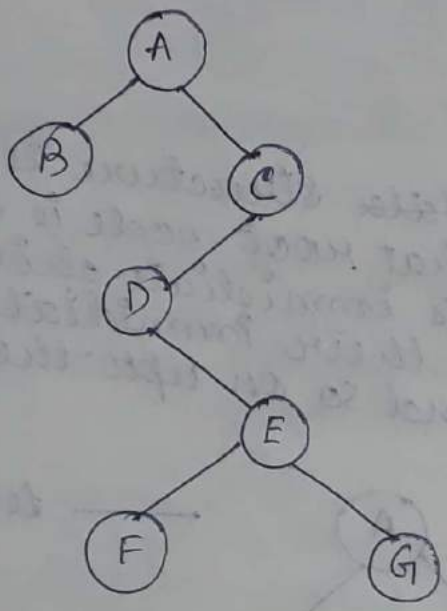


(ii) Edge:

It is the connecting line of 2 nodes.

(iii) Path:

It is the sequence of ~~consequ~~ consecutive edges from the source node to the destination node.



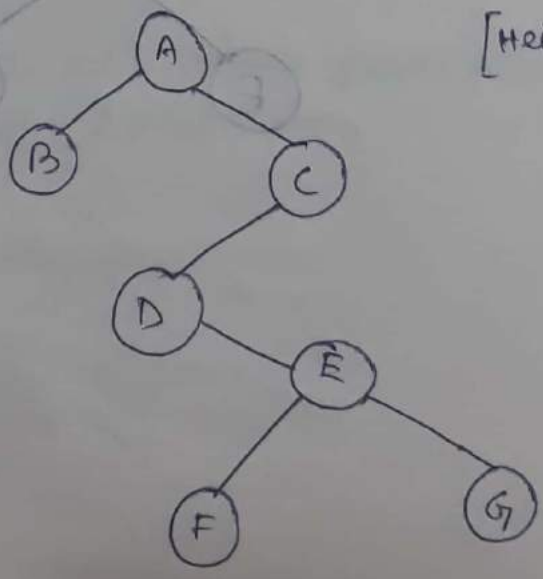
The path from A to G is AC, CD, DE, EG.

(iii) Depth:

Depth of a tree is the maximum no. of nodes in a branch of a tree. It is also called height of a tree.

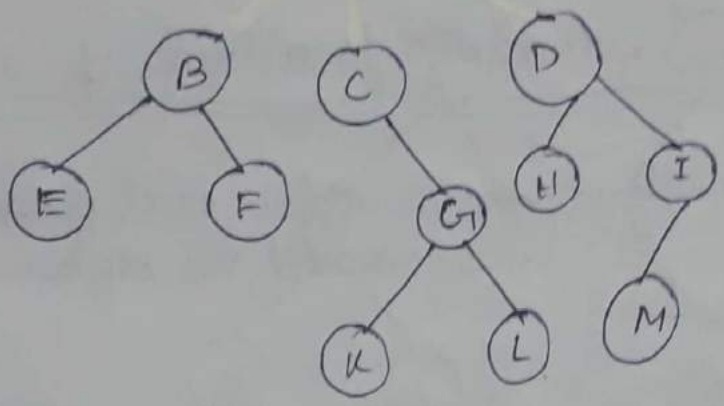
[Height = maximum level + 1]

AC, CD, DE, EG is the depth of the tree.



(IIII) Forest :-

It is a set of disjoint trees. In a given tree if we remove the root node then it becomes a forest.



(IIII) Various types of Trees :-

(1) Binary Tree - A binary tree T is defined as a finite set of elements, called nodes, such as :-

(a) T is empty (called the Null Tree or Empty Tree), or

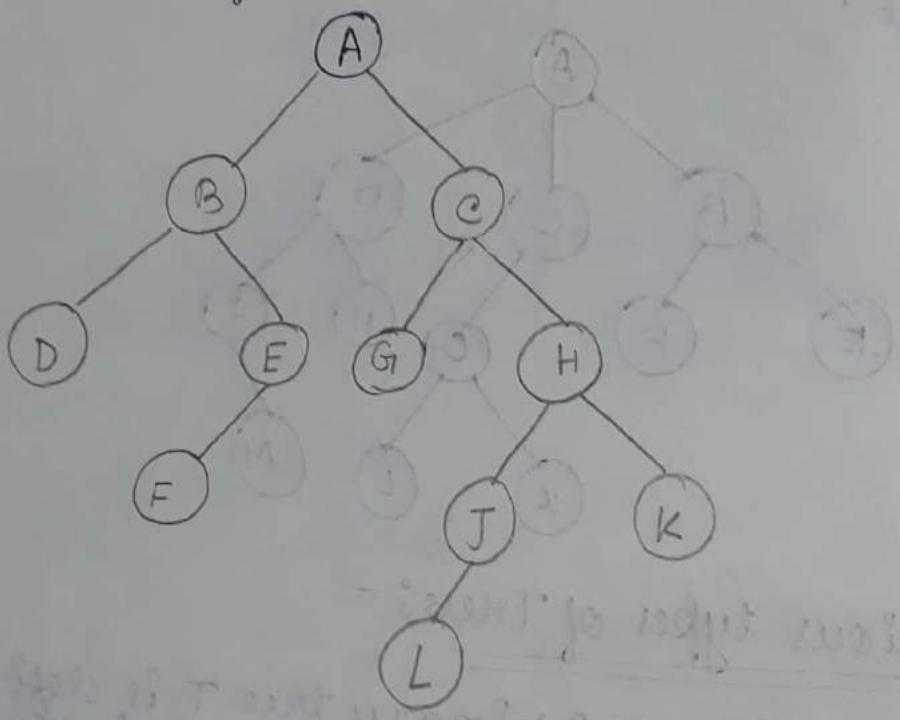
(b) T contains a distinguished node ~~are~~ R called the root of T and the remaining nodes of T ~~from~~ form/create an ordered pair of disjoint binary trees T_1 and T_2

If R is the root of T then T_1 and T_2 are called left and right sub trees of R.

4/9/18 (Tuesday)

If T_1 is non empty, then its root is called the left successor of R. Similarly if T_2 is non-empty, then its root is called right-successor of R.

Consider the following example:-



Binary Tree-T

According to diagram of binary tree T,-

(1) T consists of 11 nodes.

(2) A is the root node.

(3) B is the left successor of A and C is the right successor of A.

(4) The left sub tree of the root A consists of the nodes B, D, E, F and the right sub tree of A consists of the nodes C, G, H, J, L, K.

(*) If root = NULL then the binary tree is empty.

(*) Any node in a binary tree has either 0, 1 or 2 successors. In the tree diagram nodes A, B, C and H have 2 successors,

the nodes E and J have only 1 successor. and the nodes B, F, G, L and K have no successor.

(*) The nodes with no successor are called terminal nodes.

(III) Some properties of a binary tree :-

(1) A binary tree with n nodes has exactly n-1 edges or branches.

Proof :-
We can prove the property by induction of n.

Induction base :
If $n = 1$, then the tree contains only one node and hence no. of edges is $n-1 = 1-1 = 0$

Induction hypothesis :
A tree with n nodes must have a root and m no. of children, where $m > 0$.
If n_k denotes the no. of nodes in the k^{th} child, then

$$n = 1 + \sum_{k=0}^{m-1} (n_k) \qquad 0 \le k \le m-1$$

The no. of edges in the k^{th} child of the root is $(n_k - 1)$

7/9/18 (Friday)

In all the children of the root has:

$$n = \sum_{K=0}^{m-1} (n_K - 1) \text{ edges.}$$

also the original tree has another m edges from the roots to its m children. Hence total no. of branches in the tree is:

$$n = \sum_{K=0}^{m-1} (n_K - 1) + m$$

$$= \sum_{K=0}^{m-1} (n_K - 1) - m + m$$

$$= (n-1) \text{ proved.}$$

(2) The maximum no. of nodes on level i of a binary tree is 2^i where $i \geq 0$.

We can prove this property by induction of i.

Induction base:

A binary tree on level $i=0$ contains only the root node because $2^i = 2^0 = 1$

Induction hypothesis:

Let $i > 0$ and assume that the maximum no. of nodes on level $i-1$ is 2^{i-1}

Induction Steps:

Since each node in a binary tree has maximum degree of 2, the maximum no. of node on level $i = 2 * 2^{i-1}$

(Proved)

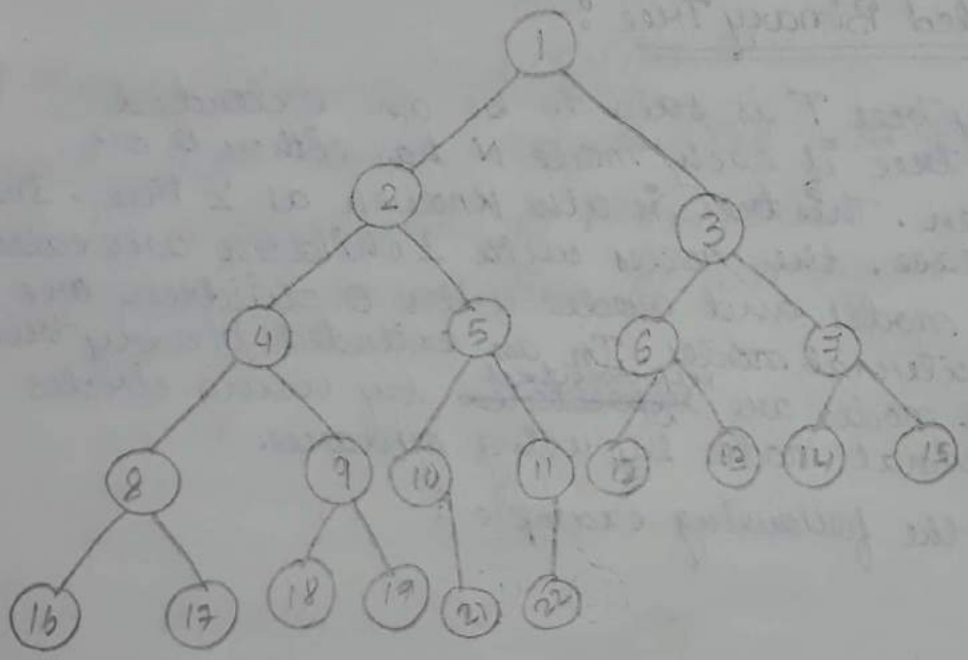
3) The maximum no. of nodes in a binary tree of height h is $2^h - 1$ where $h \geq 1$

(IIII) Complete Binary Tree :-

A complete binary tree is a binary tree with satisfied 2 properties :-

(1) In a complete binary tree, every level except possibly the last (may be) is completely filled.

(2) All nodes at the last level appear as far left as possible.



It is a complete binary tree and all nodes are labelled by integers 1, 2, 3, ... 22 from left to right, generation by generation. With this labelling any one can determine easily the children and parent of any node K in any complete binary tree.

Specifically, the left and right children of node K are respectively, 2*K and (2*K)+1 and the parent of K is the node [K/2]

For Eg:- Children of node 9 are nodes 18 and 19 and its parent is $[9/2] = 4$

The depth d_n of the complete binary tree T_n with n nodes is given by $d_n = \log_2 n + 1$

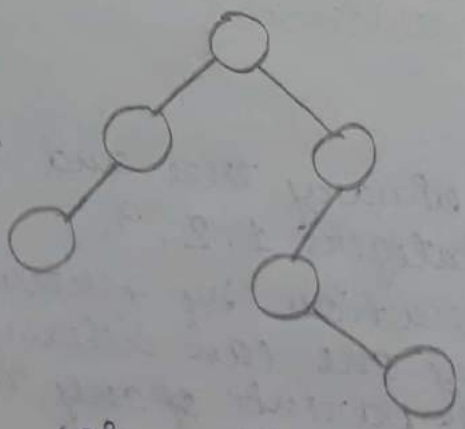
For Eg:- If the complete binary tree T_n has $n = 1000000$ nodes then its depth is $d_n = 21$

10/9/18
(Monday)

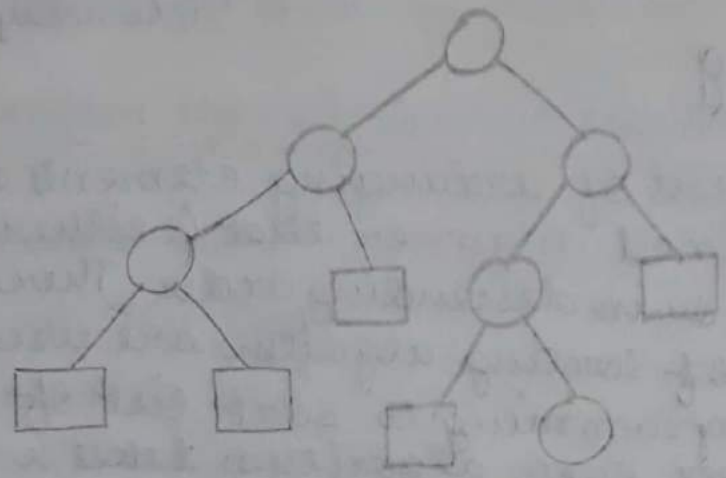
(III) Extended Binary Tree :-

A binary tree T is said to be an extended binary tree if each node N has either 0 or 2 children. This tree is also known as 2 tree. In such a case, the nodes with 2 children are called internal nodes and nodes with 0 children are called external nodes. In an extended binary tree internal nodes are ~~represented~~ represented by using circles and external nodes by using squares.

- Consider the following example :-



Binary Tree T



Extended Binary Tree / 2-Tree

(III) Binary Tree Representation :-

Binary Tree can be represented using 2 methods:

(1) Array representation of binary trees.

(2) Linked representation of binary trees.

Lab

10/9/18
(Monday)

Sorting

Sorting is a process of arranging elements of a list in a defined manner that is either in ascending order or in descending order. There are various type of sorting algorithm are used in computer programming to sort list of elements. However each algorithm takes a list of data or elements as random order and arrange them in a given sequence (it may be numerically or lexicographically or any other user defined order).

Suppose in a list there are n elements like $a_0, a_1, a_2, a_3, \ldots a_{n-1}$. After applying a sorting algorithm these elements are arranged in the list as like :-

$$a_0 > a_1 < a_2 < a_3 < \ldots < a_{n-2} < a_{n-1}$$

Sorting methods are 2 types :-

(1) Internal storage Sorting
(2) External storage Sorting

(I) Internal storage Sorting :-

If the file to be sorted is small enough, so that the entire sort can be carried out in the main memory, this type of sorting is known as Internal storage sorting.

(II) External Storage Sorting :-

If the file to be sorted is very large, requiring a large amount of physical memory, it may be sorted on secondary memory (hard disk etc.), this type of

sorting is known as external storage sorting.

(III) Various type of Sorting Algorithm :-

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Shell Sort
7. Radix Sort
8. Heap Sort
9. Tree Sort

(III) Bubble Sort :-

Bubble Sort algorithm sorts the list of elements by repeatedly moving ~~the more~~ the largest element of the entire list to the highest index position. In bubble sort method, each element is compared with its adjacent element. If the first element is larger than the second one then the position of the elements are interchanged otherwise it is not changed. Then the next element is compared with its adjacent element and the same process is repeated for all the elements in the array.

In bubble sort method, at the end of first cycle/~~or~~ first pass/first iteration, the largest element will be placed at the end of the sorted list. After the second cycle/pass the second largest element is placed at the second last position. This same process is repeated until no more elements are left for comparison.

(11) Trace of bubble sort algorithm :-

Assume that list X contains 8 elements. They are :

6 2 4 7 1 3 8 5

1st iteration / 1st Pass :

a) X[0] > X[1] ; ie; 6 > 2 - condition is true so swap 6 and 2.

Then the array becomes -

2 6 4 7 1 3 8 5

b) X[1] > X[2] ie; 6 > 4 - condition is true so swap 6 and 4.

Then the array becomes -

2 4 6 7 1 3 8 5

c) X[2] > X[3] ie; 6 > 7 - condition is false so no swapping.

The array is -

2 4 6 7 1 3 8 5

d) X[3] > X[4] ie; 7 > 1 - condition is true so swap 7 and 1.

Then array becomes -

2 4 6 1 7 3 8 5

e) X[4] > X[5] ie; 7 > 3 - condition is true so swap 7 and 3

Then the array becomes -

2 4 6 1 3 7 8 5

f) x[5] > x[6] ie; 7 > 8 - condition is ~~true so swap 7 and 8~~ false so no swapping

Then the array becomes -

2 4 6 1 3 7 8 5

g) x[6] > x[7] ie; 8 > 5 - condition is true so swap 8 and 5.

Then the array becomes -

2 4 6 1 3 7 5 8

~~**~~ * End of 1st Iteration * *

2nd iteration :-

2 4 1 3 6 5 7 8

3rd iteration :

~~2 1 3~~

2 1 3 4 5 6 7 8

4th iteration :

1 2 3 4 5 6 7 8

Tree

11/9/18
(Tuesday)

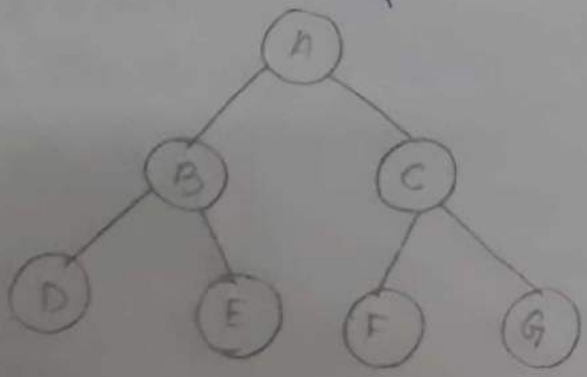
(III) Array Representation of a Binary Tree :-

In data structure, binary tree can be easily represented using linear array very easily. Each and every nodes of binary tree T are stored as an array element in the memory. These nodes are ~~accessible~~ accessable sequentially as like array elements in an array. Now, generally array starts with index 0 and upto maxsize-1. So in the binary tree, node's no. starts from 0 to maxsize-1.

Suppose array is TREE[MaxSize] and binary tree is T. Now,

1. The root node R is always at index 0. So R of T is stored in TREE[0].
2. If a node N occupies at TREE[K] then left child of N is stored in TREE[2*K+1] and the right child of N is stored in TREE[2*K+2]
3. The maximum no. of nodes is specified by maxsize.
4. NULL is used to indicate an empty sub tree. In particular, TREE[0] = NULL indicates that the tree is empty.

Consider the following example :



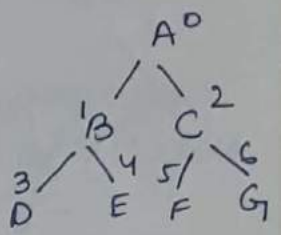
MaxSize = 7

TREE[7]

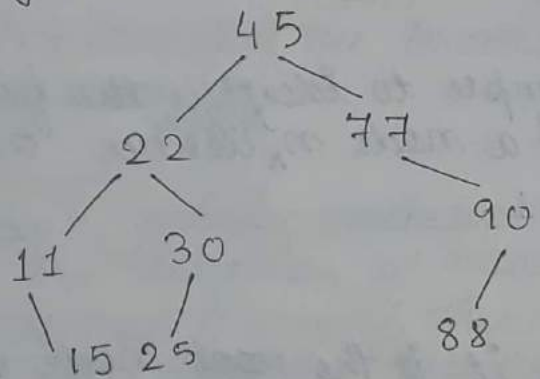
0	A
1	B
2	C
3	D
4	E
5	F
6	G

LC = 2K+1
RC = 2K+2

K → index of Parent Node



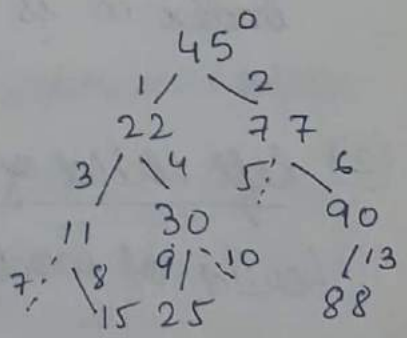
Q) Find out the array representation of the following tree :-

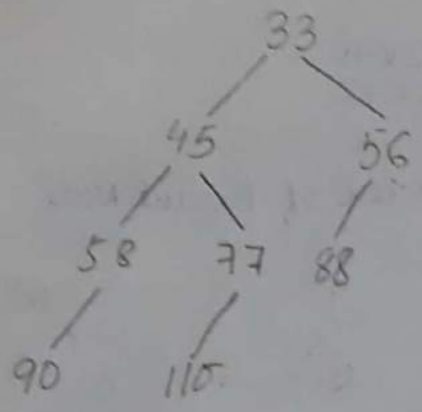


28/9/18
(Friday)

⇒

0	45
1	22
2	77
3	11
4	30
5	
6	90
7	
8	15
9	25
10	
11	
12	88





0	33
1	45
2	56
3	58
4	77
5	88
6	
7	90
8	
9	110

(iii) How to identify the father, the left child and the right child of an arbitary node in array representation?

Ans:- It is very simple to identify the father & the children of a node n, where index $0 \leq n \leq maxSize - 1$.

(1) Father of n :

If $n = 0$ then it is the root node and has no father. Otherwise the father of node having index n is $\frac{(n-1)}{2}$

(2) Left child of n : (Lchild)

Lchild of node of n index no. is $2n+1$.

(3) Right child of n : (Rchild)

The Rchild of node of n index is $2n+2$.

(4) Siblings:

left sibling of a node of n index is $n-1$ and right sibling of a node of n index is $n+1$.

(II) Linked List representation of binary tree :-

Binary tree can be represented using linked list representation. Here every node consists of three fields :- ~~Info~~, Left

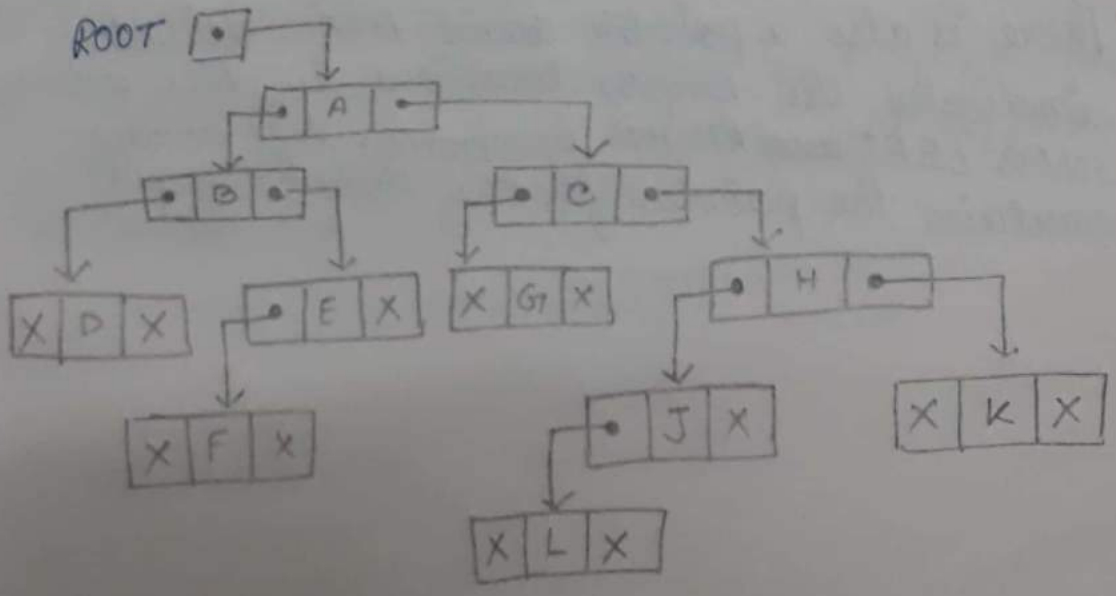
INFO, LEFT, RIGHT.

In memory these fields are 3 parallal arrays. Suppose T is the binary tree. Each node N of T will correspond to a location K such that :-

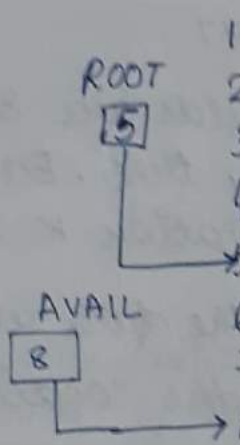
(1) INFO[K] contains the data at the node N.
(2) LEFT[K] contains the location of the left child of node N.
(3) RIGHT[K] contains the location of the right child of node N.

There is also a pointer variable ROOT which contains the location of root node R of T. If any sub tree is empty, then the corresponding pointer will contain the null value. If the tree T is empty then ROOT will contain the null value.

Consider the following linked representation of a binary tree :-



(ii) Memory representation of this tree :-



	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	

There is also a pointer AVAIL which indicates the empty locations in the arrays INFO, LEFT and RIGHT. Generally left array contain the pointers for the AVAIL list.

1/10/18
(Monday)

Traversing Binary Tree :-

Tree traversal is a way in which each and every node in the tree is visited exactly once in a systamatic manner. In data structure there are 3 standard ways of traversing a binary tree T with root R. These 3 algorithms are as follows :-

~~1. Tree order tra~~

1. Preorder traversal
2. Inorder Traversal
3. Postorder traversal

Preorder traversal :-

1. Process the root R
2. Traverse the left sub tree of R in preorder.
3. Traverse the right sub tree of R in preorder.

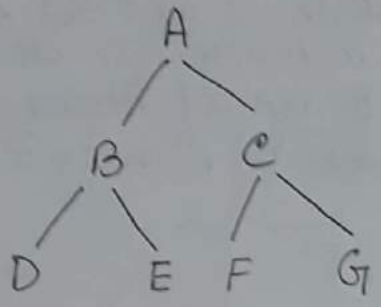
Inorder traversal :-

1. Traverse the left sub tree of R in inorder.
2. Process the root R.
3. Traverse the right sub tree of R in inorder.

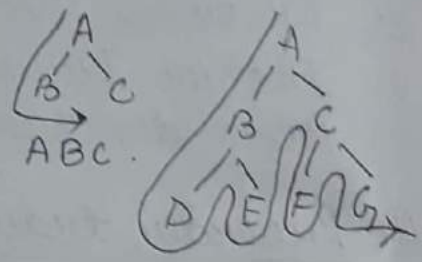
Postorder traversal :-

1. Traverse the left sub tree of R in postorder.
2. Traverse the right sub tree of R in postorder.
3. Process the root R.

Q1) Find out the sequence of nodes of the following binary trees using preorder, inorder and postorder traversal method.

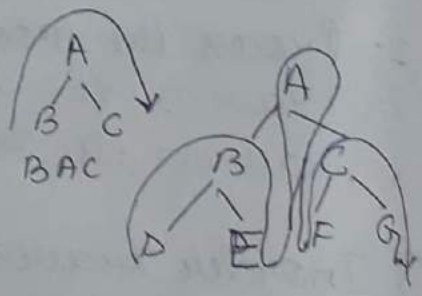


⇒ ① Preorder Traversal :



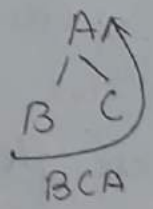
(A)BDECFG

② Inorder Traversal :



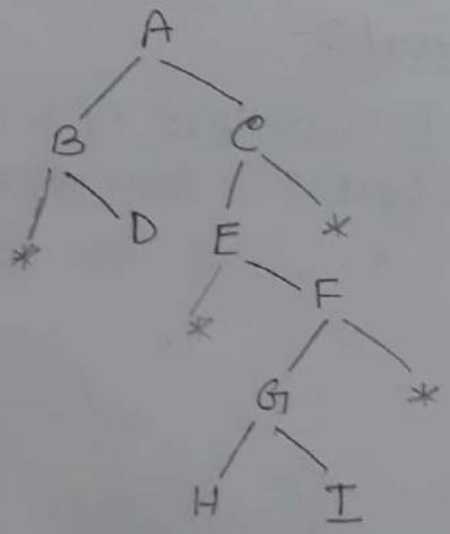
DBE(A)FCG

③ Postorder traversal :



DEBFGC(A)

Q2)



⇒ ① Preorder Traversal:

ABDCEFGHI

AB*DCE*FGHI
**

② Inorder Traverse :-

*BD A*E HGI F*C *

= ~~DD~~

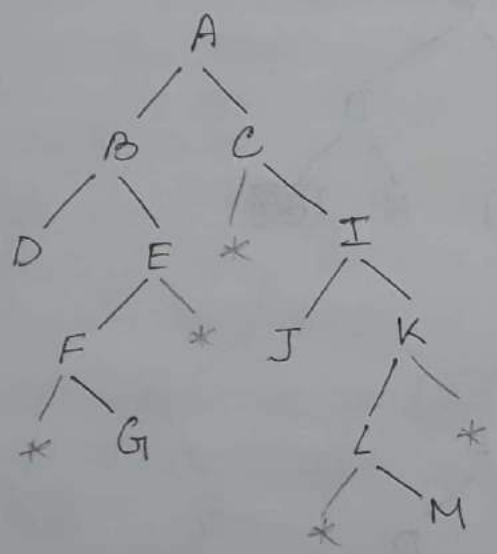
= BDAEHGIFC

③ Postorder Traverse :-

*DB*HIG* FE*C A

= DB HIGFECA

Q3)



⇒ ① Preorder Traverse :-

ABDEF*G* C *IJKL*M *

= A BDEF GC IJKLM .

② Inorder Traverse :-

DB*FGE* A*CJI*LMK*

= DBFGEACJILMK

③ Postorder Traverse :-

D*GF*EB* J*ML*KICA

: DGFEBJMLKICA

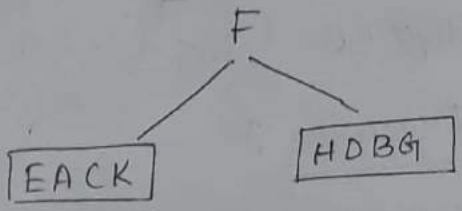
Q3) Construct a binary tree using the following sequence of node.

Inorder: EACKFHDBG

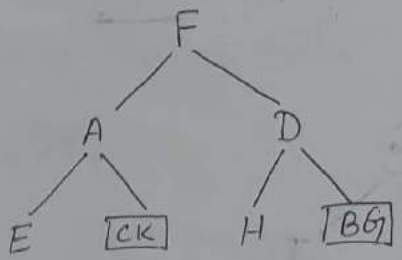
Preorder: FAEKCDHGB

⇒

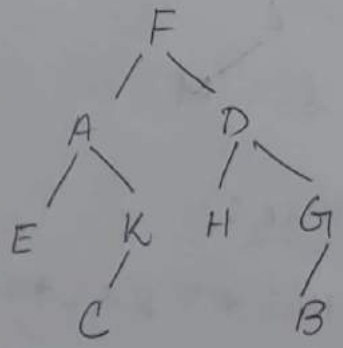
Step 1



Step 2

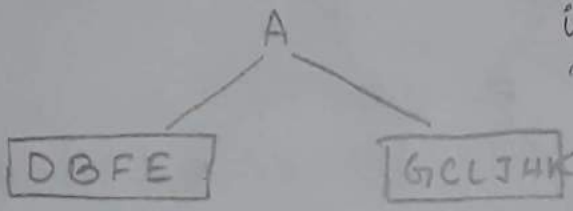


Step 3



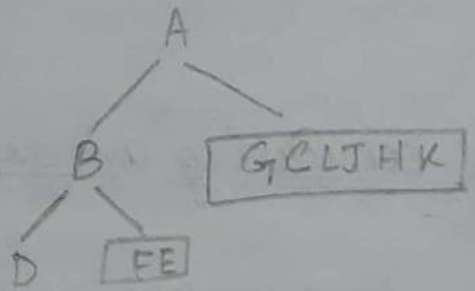
Q4) Inorder : D B F E (A) G C L J H K
Postorder : D F E B G L J K H C (A)

⇒ Step 1 :

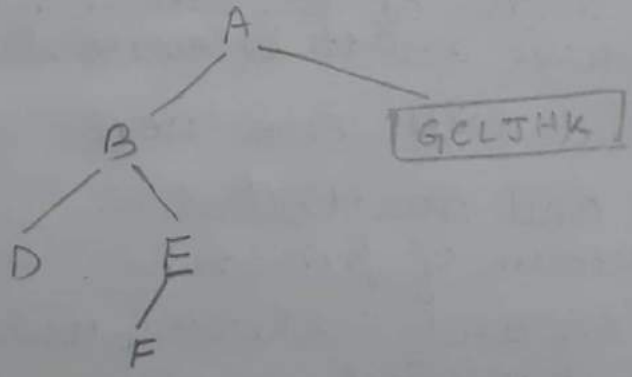


comparing the post-order, A is the root & in Inorder, A is in middle. So DBFE are the left child and GCLJHK are the right child

Step 2 :

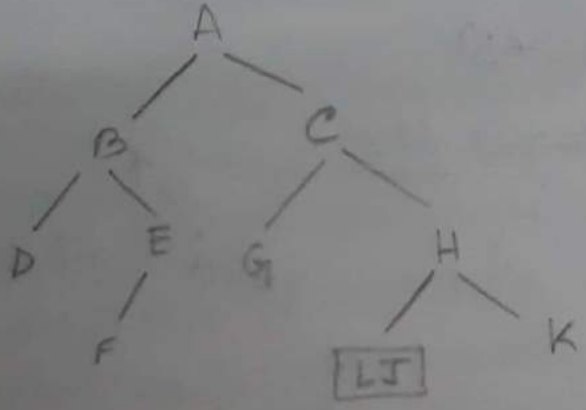


Step 3 :

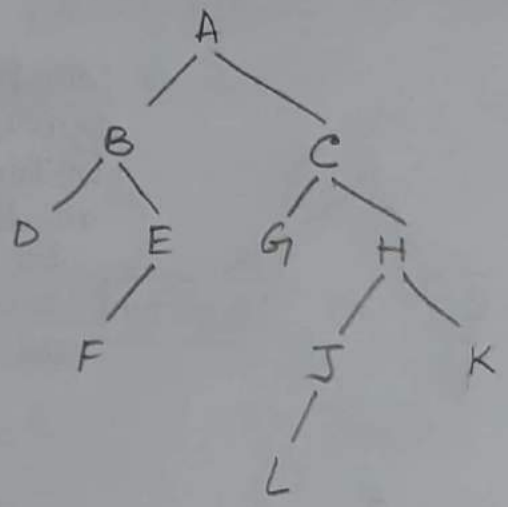


Since in inorder F is before E, So F is the left child of E

Step 4 :



Steps:



(III) Technique of converting an expression into binary tree:-

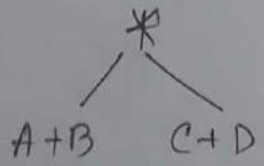
Divide and compare technique is used to convert an expression into binary tree. The rules are as follows:-

1. Note the order of precedence. All expression in paranthesis are to be evaluated first.
2. Exponential will come next.
3. Division and multiplication will be the next in order of precedence.
4. Subtraction and addition will be the last to be processed.

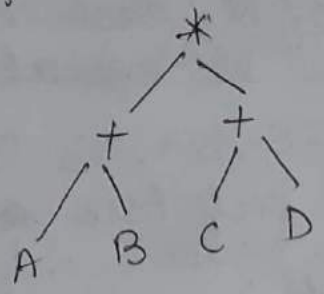
(1) $(A+B)*(C+D)$

$\Rightarrow$ ~~$(AB+)*(CD+)$~~ Step1

=



Step 2 :



Preorder : *+AB+CD

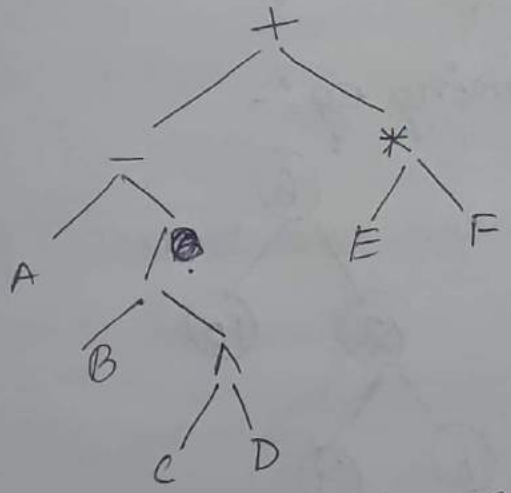
Postorder : AB+CD+*

(2) A-B/(C^D)+(E*F)

⇒ Preorder :- ~~+(C^D)~~

~~+-CADE*F~~

+-A/B^CD*EF



(iii) Binary Search Tree :- (BST)

In tree data structure, binary search tree is very important part. It is also known as ordered binary tree. In a BST all nodes are arranged in an order. A binary Search Tree / BST is also a binary tree but it has the following properties :-

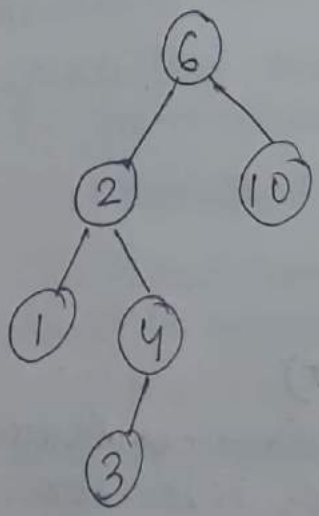
(1) The left sub tree of a node N contains values that each node value is less than the value of N in the left sub tree of N.

(2) In the right sub tree of N, each node's value is greater than or equal to the value of N.

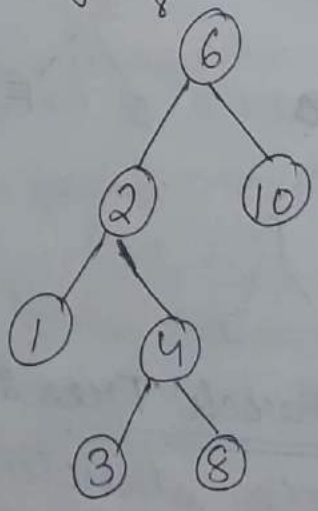
(3) Both left and right subtree ~~are~~ also satisfy these properties.

Suppose T is a binary tree. Now T is also called as Binary Search Tree (BST) if any node N of T satisfies the properties that the value of N is more than every value of node in the left sub tree and value of N is less than or equal to each value of nodes in the right sub tree.

Consider the following eg:-



BST



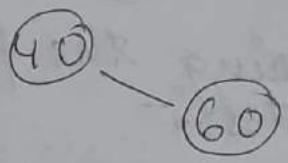
Non BST

9) Construct a binary search tree using the sequence of following nodes :-

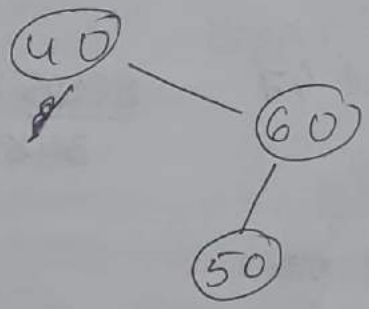
40, 60, 50, 33, 55, 11

⇒ Step 1 :- Insert 40.

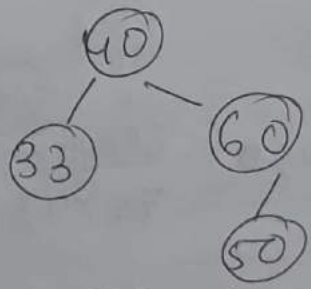
Step 2: Insert 60. Now 60 greater than 40.



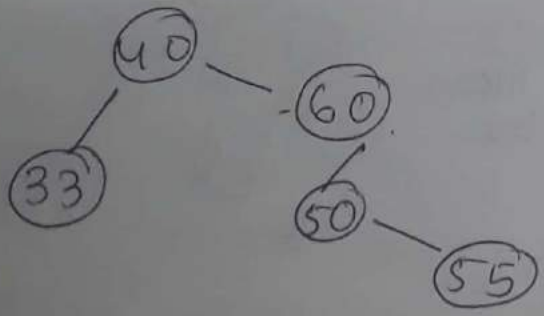
Step 3: Insert 50. Now 50 greater than 40 but 50 less than 60.



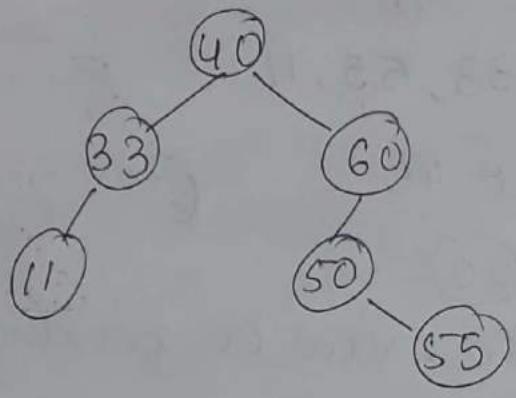
Step 4: Insert 33. Now 33 < 40



Step 5: Insert 55. Now 55 > 40 and 55 < 60 and also 55 > 50.



Step 6 : Insert 11. Now 11 < 40 & 11 < 33.

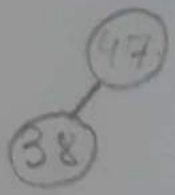


Q2) Construct BST using the following sequence of nodes:-

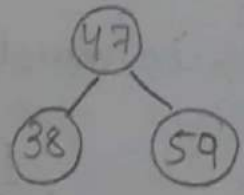
47, 38, 59, 12, 34, 78, 32, 10, 89, 52, 67, 81

⇒ Step 1 : Insert 47.

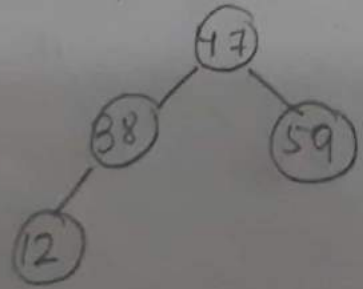
Step 2 : Insert 38. Now 38 < 47



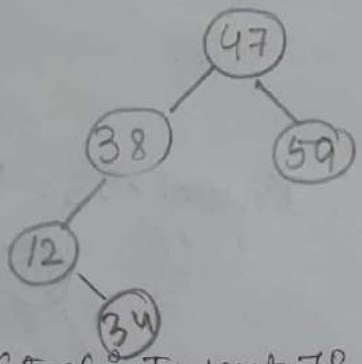
Step 3 : Insert 59. Now 59 > 47



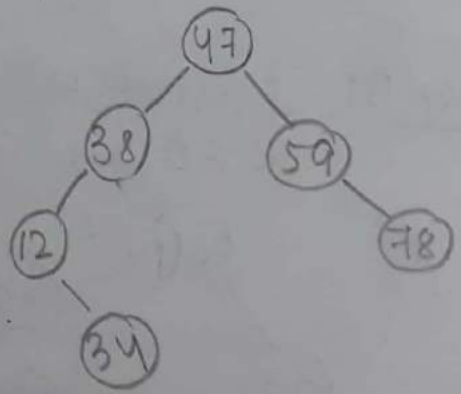
Step 4 : Insert 12. Now 12 < 47 and 12 < 38.



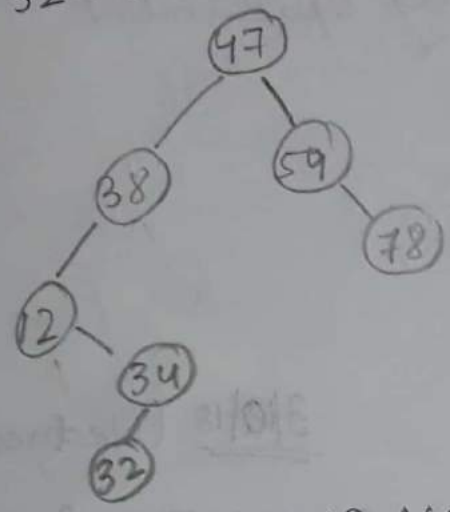
Step 5 : Insert 34. Now 34 < 47.



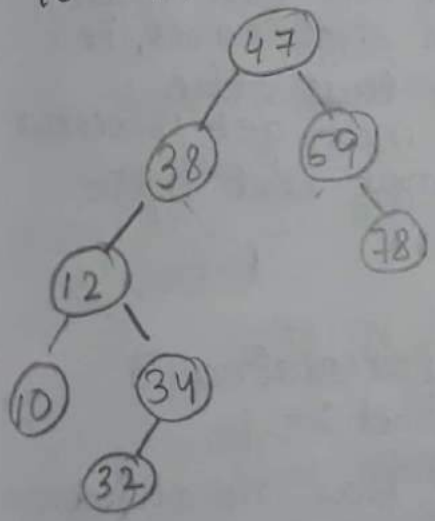
Step 6 : Insert 78. Now 78 > 47 and 78 > 59



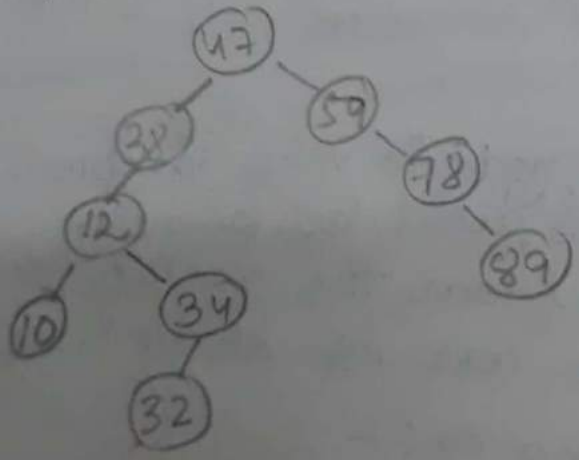
Step 7: Insert 32. Now
32 < 47



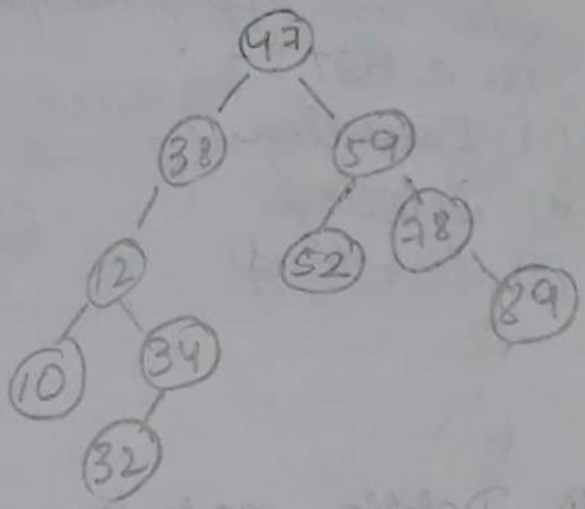
Step 8: Insert 10. Now
10 < 47 and 10 < 12



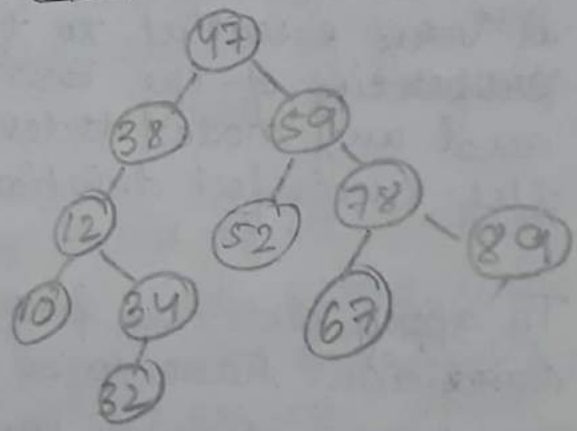
Step 9: Insert 89. Now
89 > 47 and 89 > 78.



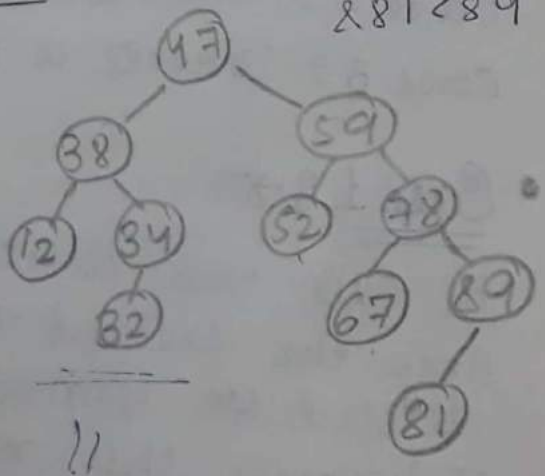
Step 10: Insert 52.
Now 52 > 47 & 52 < 59



Step 11: Insert 67.

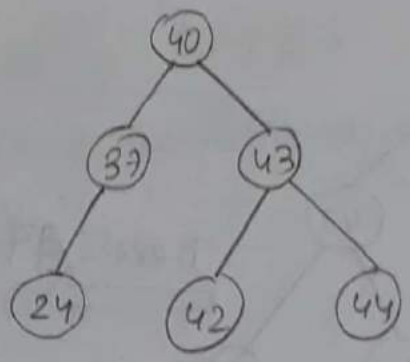


Step 12: Insert 81. 81 > 47
& 81 < 89



!!

Delete 22

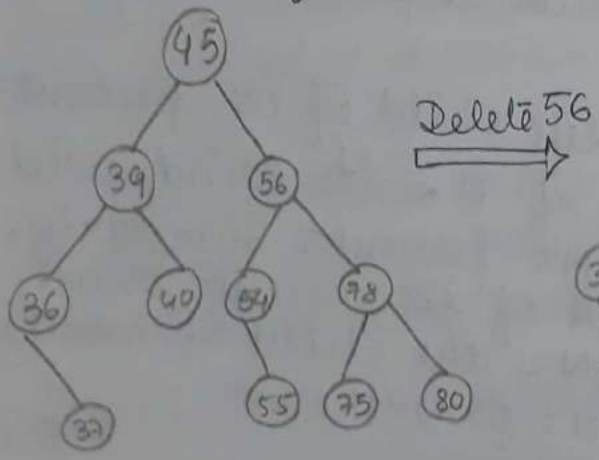


- Case 3:-

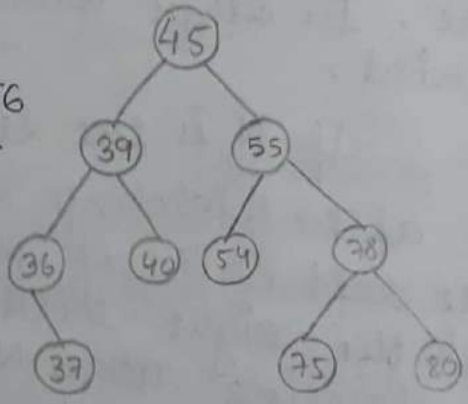
Delete a node with 2 children.

In this case after deleting a node (N) replace the N's value with its inorder predecessor (right most child of the left sub tree) or inorder successor (left most child of the right sub tree).

Consider the following example:-

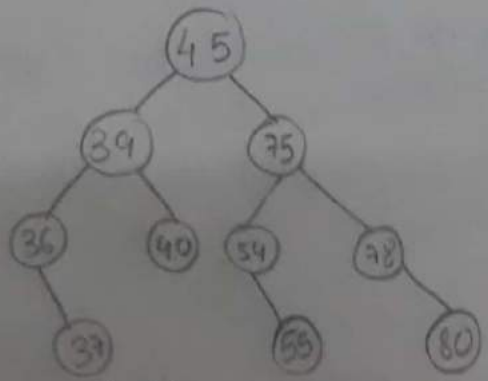


Delete 56



55 is the predecessor of 56.

Delete 56



Here 75 is the successor of 56

(iii) Balanced Binary Tree :-

In tree data structure the balance factor is very important to make a balanced tree. This balance factor helps to form proper binary ~~sear~~ tree. As a ~~tree~~ balanced ~~tag t~~ binary tree is much efficient than non-balanced binary tree because many operations such as insertion, deletion, searching etc. can be implemented on balanced binary tree very easily and efficiently.

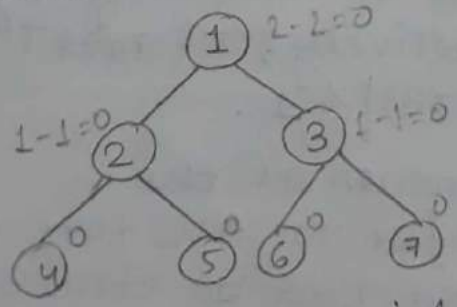
There are 2 methods to check a binary tree whether it is balanced or not. These are :-

(1) Height balance
(2) Weight balance.

(ii) Height Balance Tree :-

If T is a height balanced binary tree then T has left and right sub trees which differ in height by not more than 1.

consider the following eg :-



In this tree height balance factor is 0.

* Complete binary tree is always height balanced tree.

(IIII) Weight balanced Tree :

Weight balanced trees are a type of balanced binary trees. Here trees are balanced in order to keep the sizes of the sub trees of all the nodes within a constant factor of all the present nodes. As a result, there is a logarithm for every single operation. In these types of tree, the weight is proportional to the no. of associations present in that tree.

(III) AVL Search Tree :-

AVL Search tree is a self balancing binary search tree invented by Adelson-Velskii and Landis in 1962. In honour of its inventors this balanced BST is known as AVL tree.

Definition :-

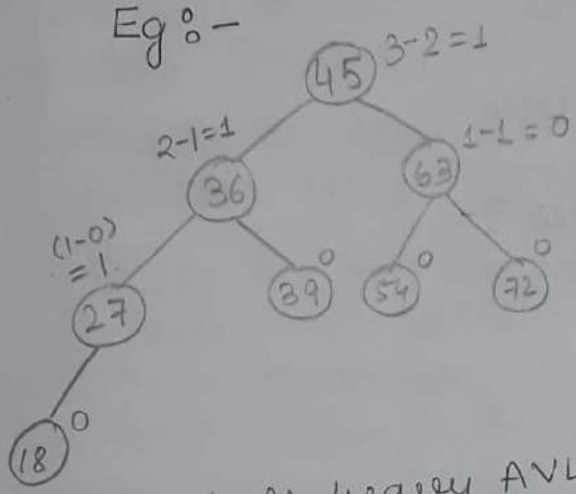
An empty binary tree is an AVL tree. A non-empty binary tree T is an AVL tree if given T^L and T^R to be the left and right sub trees of T and $h(T^L)$ and $h(T^R)$ to be the heights of sub trees T^L and T^R respectively, T^L and T^R are AVL trees and $|h(T^L) - h(T^R)| \leq 1$.

$h(T^L) - h(T^R)$ is known as the Balanced Factor (BF) and for an AVL tree the balanced factor of a node can be either 0, 1, or -1.

Now, consider the following conclusion :-

(1) If the balanced factor of a node N is 1 :
(i) It means that the left sub tree the left sub tree of N is 1 level higher than of the right sub tree of N. Such a tree is called as a left heavy tree.

Eg :-

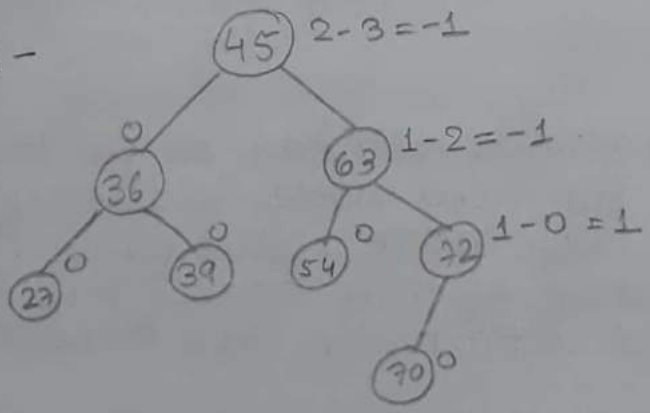


(Longest tree of a particular node should be taken)

Left heavy AVL Tree.

(2) If the balance factor of a node N is -1 :-
It means that the left sub tree of node N is one level lower than of the right sub tree of N. Such a tree is called right heavy tree.

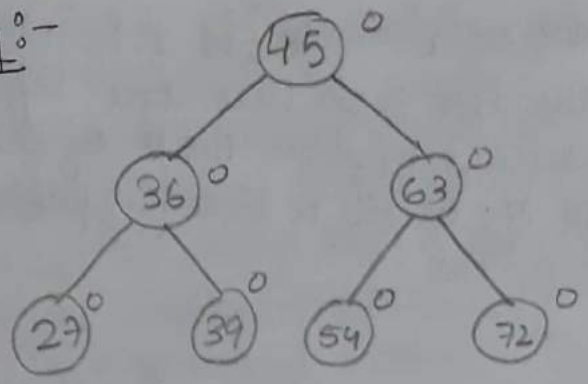
Eg :-



Right heavy AVL Tree.

(3) If the balance factor of a node N is 0 :-
It means that the height of the left sub tree of that node is equal to the height of the right sub tree of N.

Eg:-



Fully balanced AVL tree.

(III) Operations on AVL tree :-

(1) Searching
(2) Insertion
(3) Deletion

(I) Searching :-

In an AVL tree the searching method is exactly similar to the method used in a binary search tree. Due to the balancing factor the time complexity for searching a node in a AVL tree is O(Log n) [Big O Log n]

(II) Insertion :-

This operation is also as similar as in BST. In the AVL tree the new node is always inserted as the leaf node. But after insertion the balance factor of any node in the tree may be affected so that the tree becomes unbalanced.

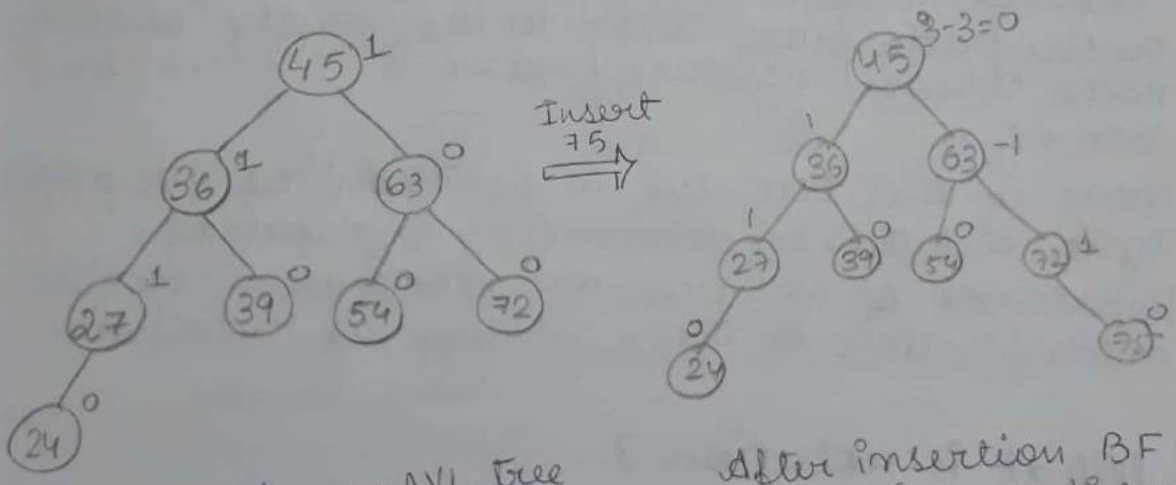
In this case the technique - rotation is applied to restore the balance of the tree. However if insertion of the new node does not disturb the balance factor of the entire tree then any type of rotation is not required

Newly inserted node will be leaf node. So its BF is 0 always. But the other nodes from root to that newly ~~root~~ leaf node the balance factor will be changed. So due to the insertion the possible changes may occur in the tree as follows:

(1) Initially the tree was either left or right heavy. So after insertion it may become balanced.

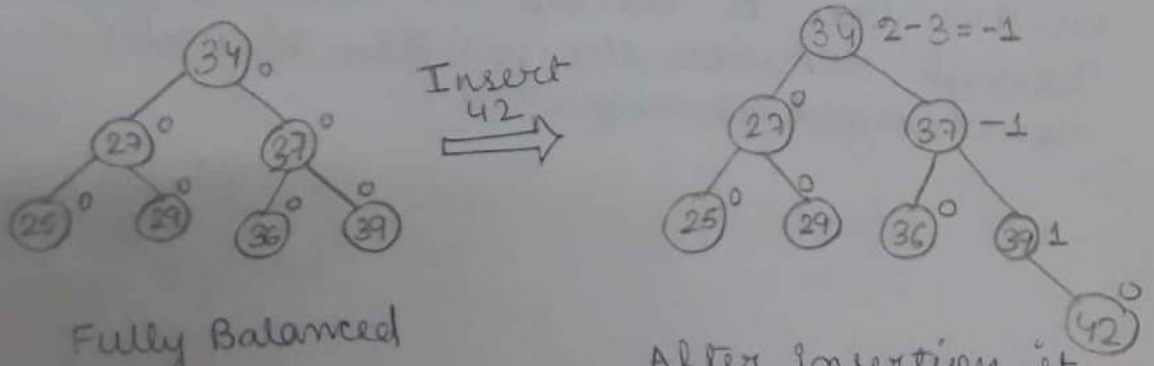
(2) Initially the tree was fully balanced. So after insertion it becomes either left or right heavy.

(3) Initially the tree was heavy (either left or right) & the new node has been inserted in the heavy sub tree. That's why it creates an unbalanced sub tree. Now in an unbalanced tree the node(s) whose balance factor is not as AVL tree that means 0, 1 or -1 then this node(s) is called <u>critical node</u>.



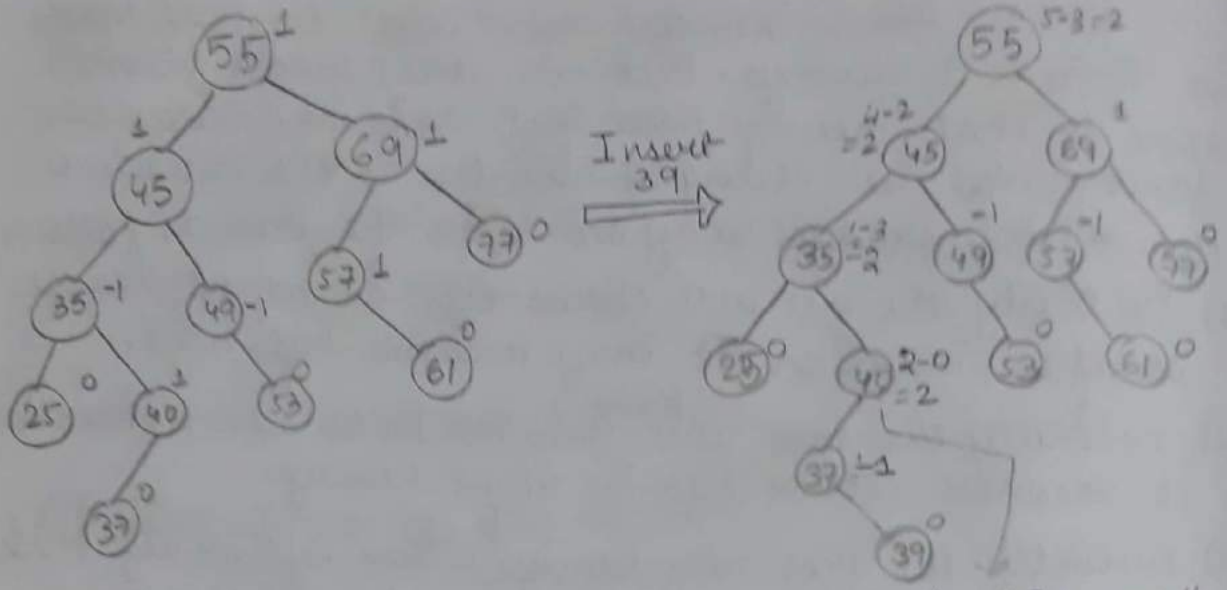
Left heavy AVL tree

After insertion BF of the tree is not disturbed



Fully Balanced AVL tree

After insertion it becomes right heavy AVL tree



Q) What is critical node?

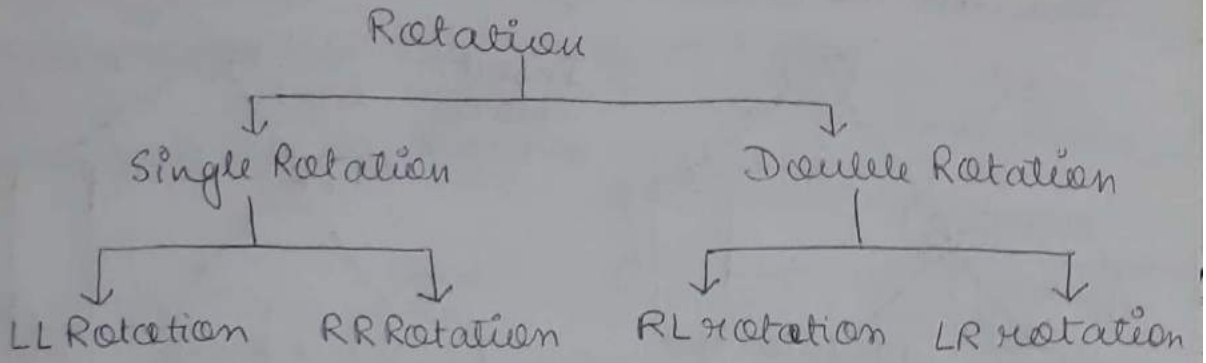
⇒ Critical node is the nearest ancestors node on the path from root to the newly inserted node whose balance factor is neither -1, 0 nor +1.

Now in this tree due to insertion the AVL property is violated. So rebalancing is required. Rebalance of AVL tree are done with simple modification to tree, known as rotation.

Q) What is rotation?

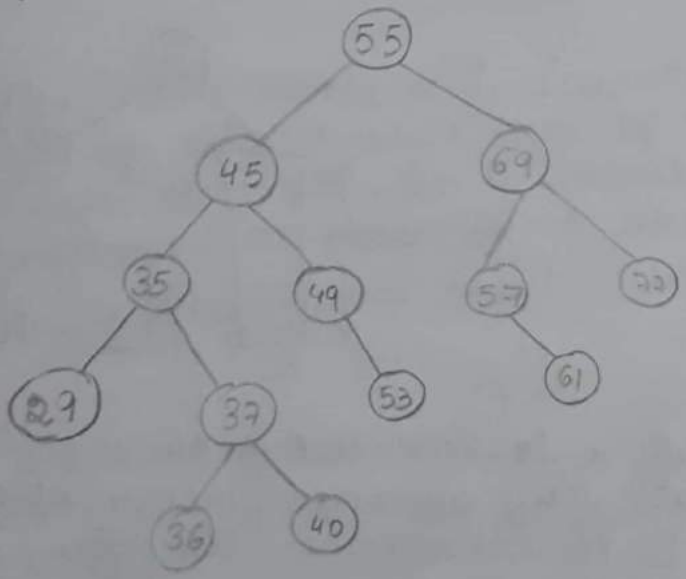
⇒ Rotation is a process which is implemented on AVL tree to restore the balance factor. Through rotation the position of nodes are rearranged as required.

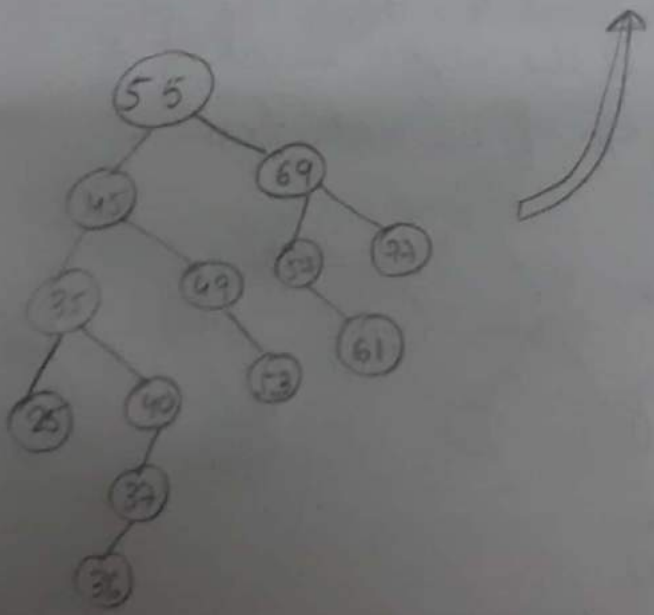
(III) Classification of rotation :-



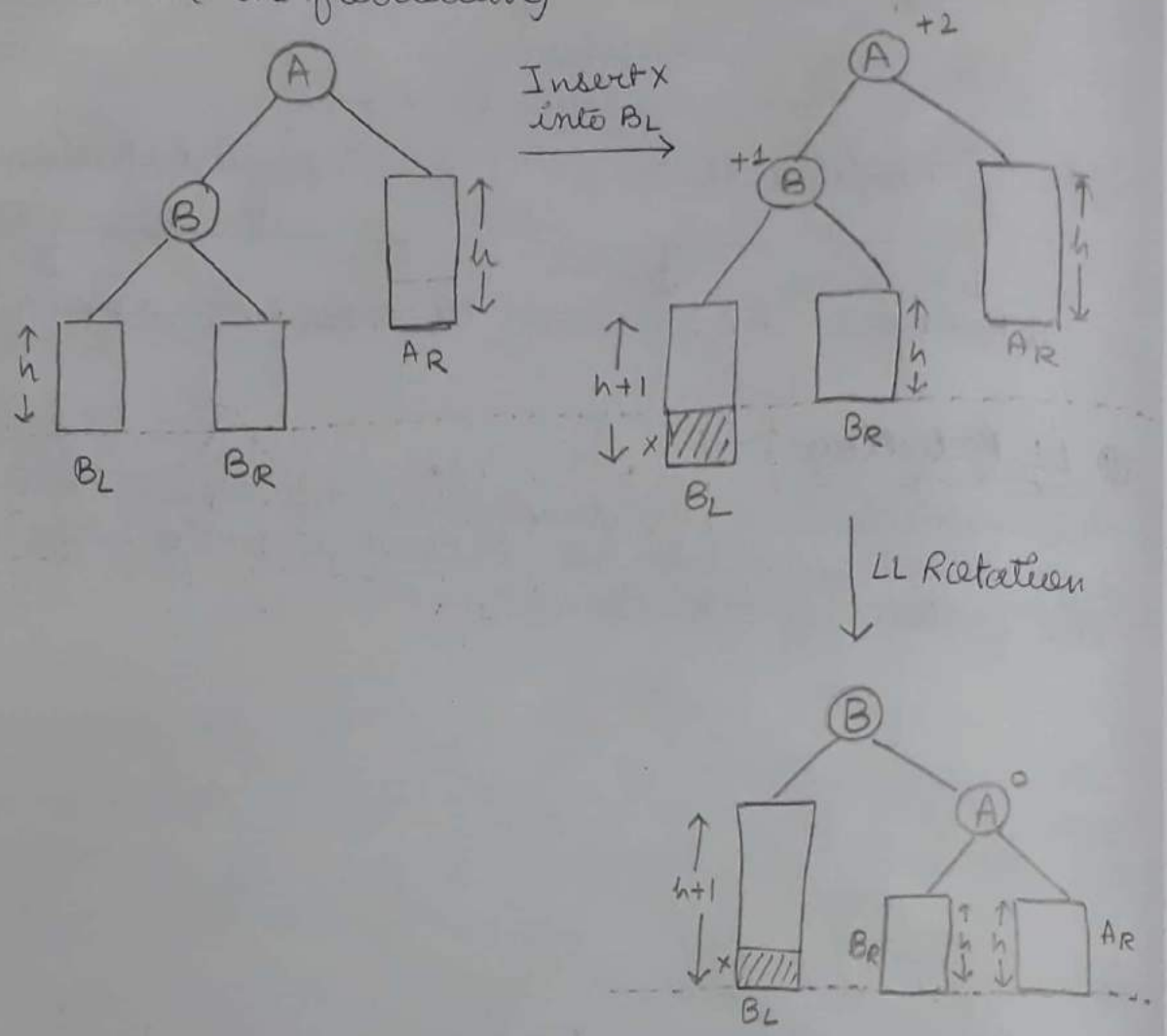
(1) LL Rotation :-

This rotation is used after an insertion into the left sub tree of left sub tree of a node. Consider the following situation :-



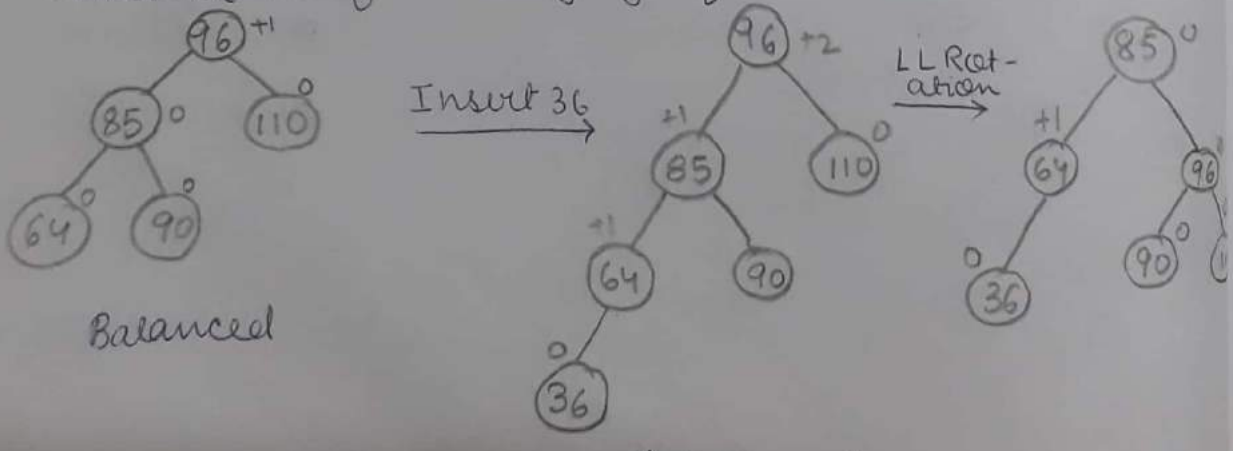


Consider the following :-



The new element- x is inserted in the left sub tree of A and is that's why balance factor of A (near ancestor) is +2. So to rebalance the tree, ~~AVL~~ LL rotation is applied. After the rotation B becomes the root of B_L and A. B_R is the left child of A and A_R is the right child of A.

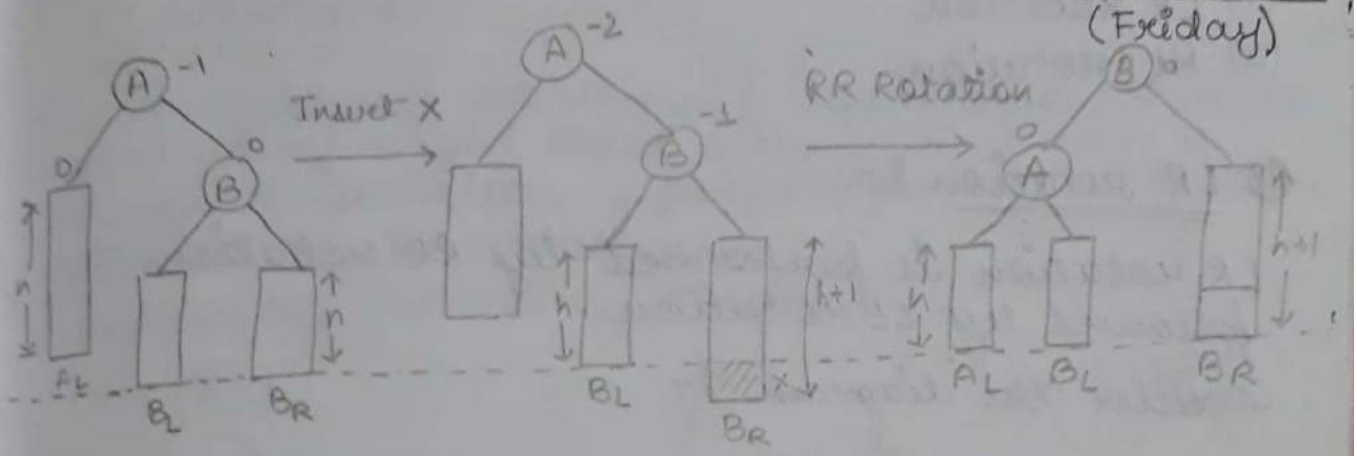
Consider the following eg. of LL rotation :-



(iii) RR rotation :

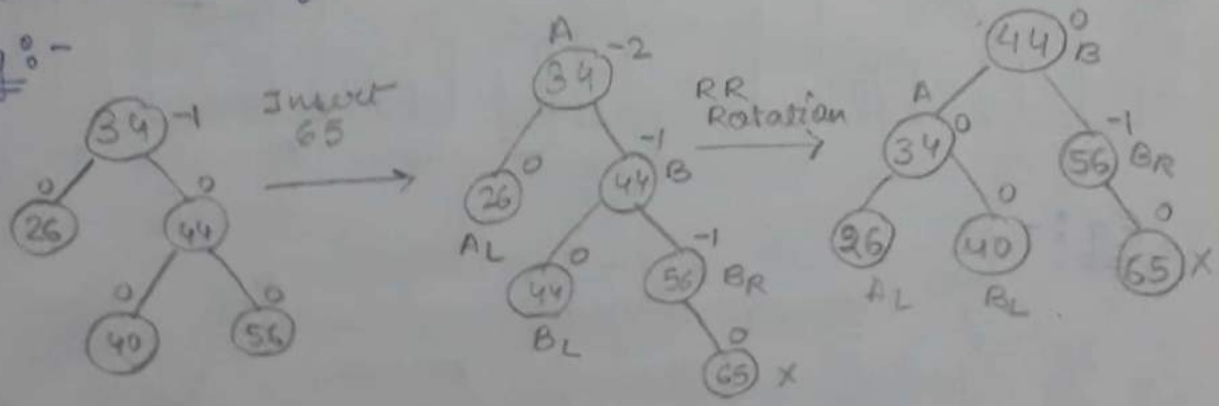
RR rotation is the mirror of LL rotation. RR rotation is used after an insertion into the right sub tree of right sub tree of a node.

5/10/18
(Friday)



Here the new node X is inserted into right sub tree of right sub tree of A. Then BF of A is changed to -2. So to releolance the tree B is changed as root of A and B_R. Then A_L and B_L are the left and right sub tree of A.

Eg :-



(iii) Double Rotation :-

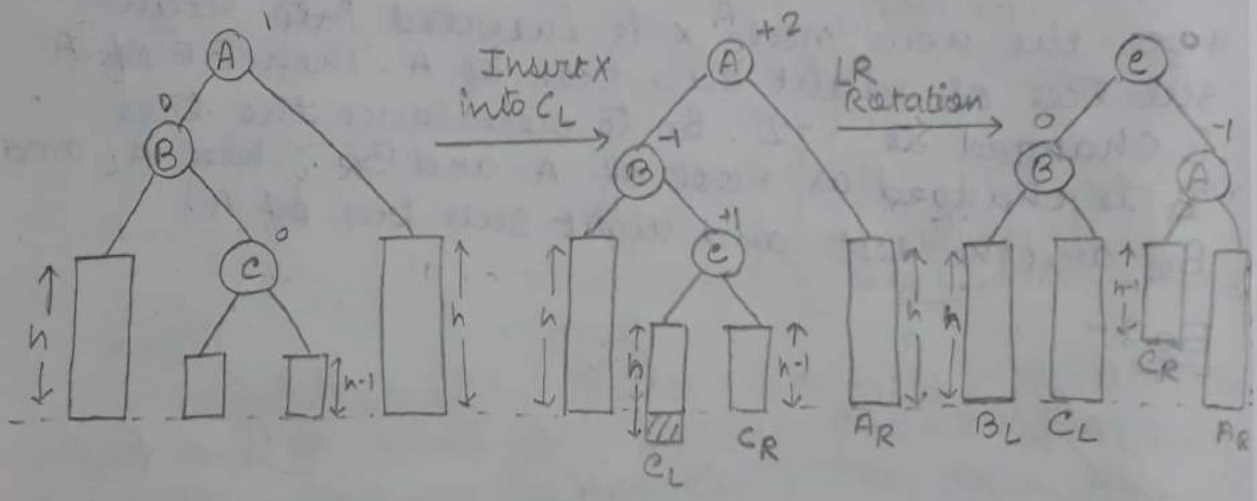
Sometimes a single rotation (LL or RR) is not sufficient sufficient to balance an unbalanced tree. In this case double rotation is essential. It is 2 types :-

(1) LR rotation
(2) RL rotation

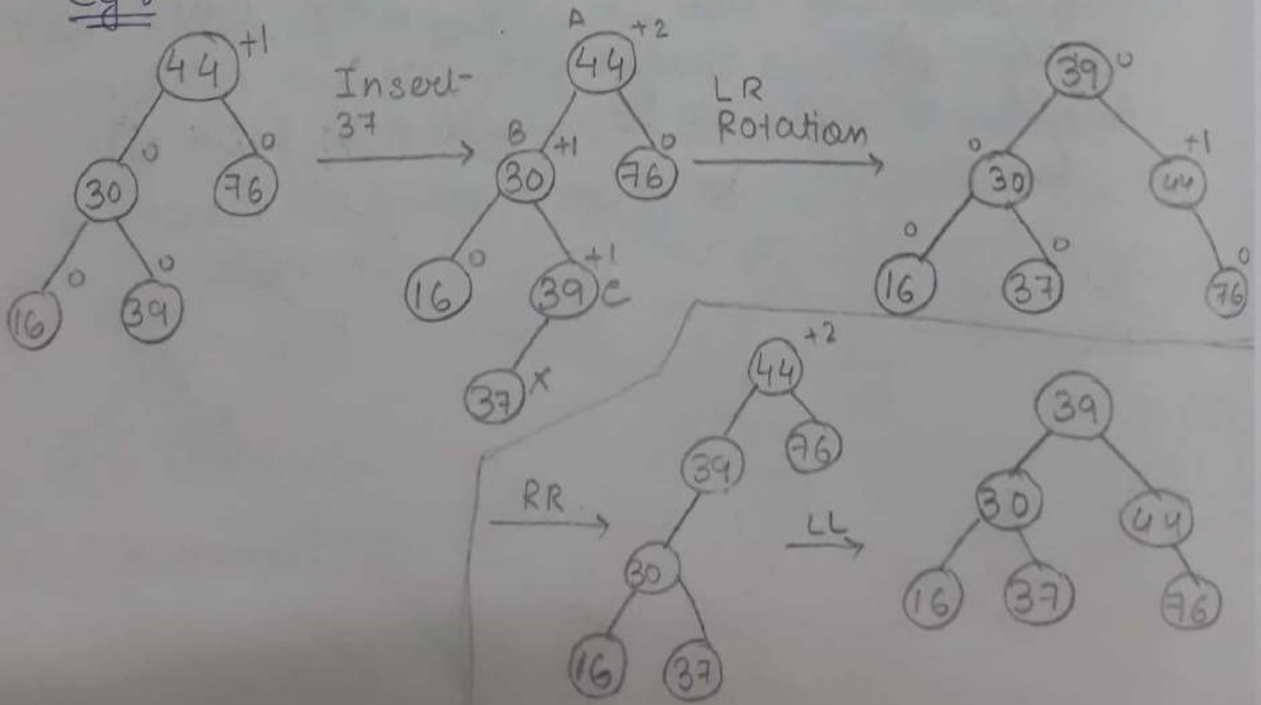
(iii) LR rotation :-

LR rotation is performed by RR rotation followed by LL rotation.

Consider the diagram :-



Eg :-

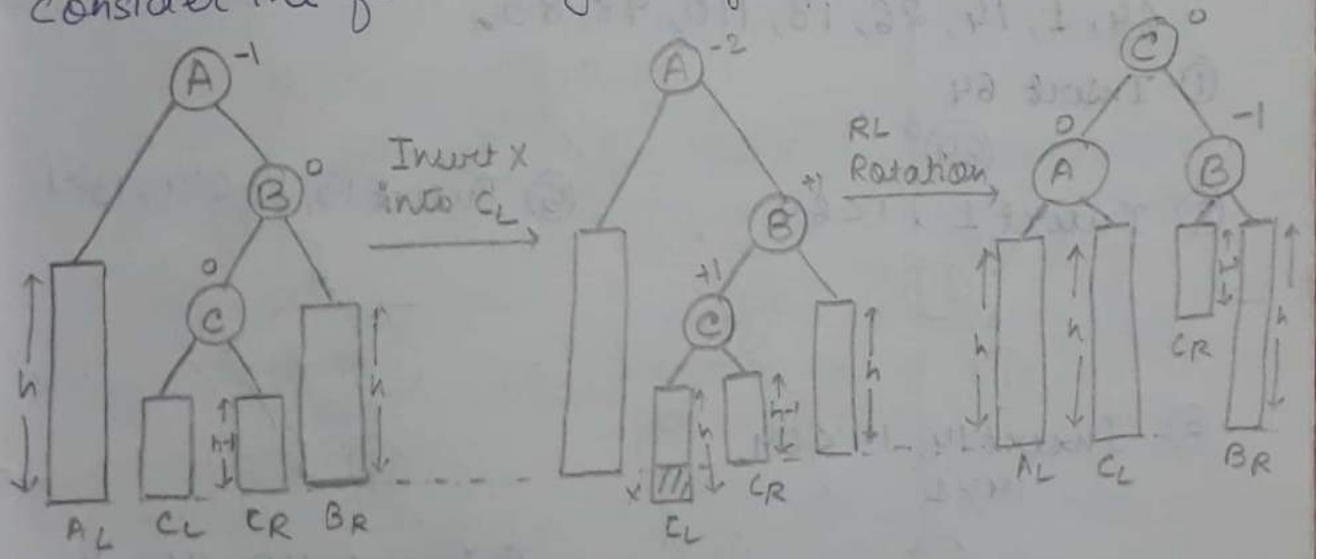


(II) RL rotation :-

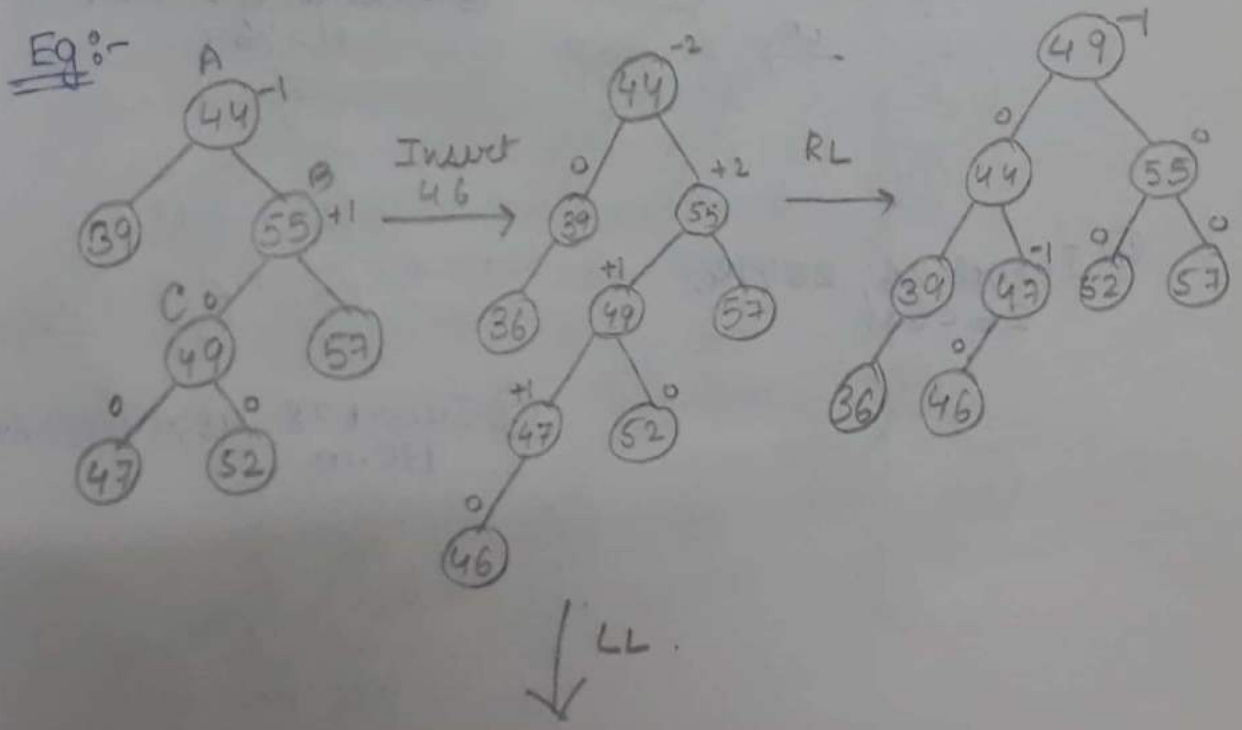
LR rotation or Left Right rotation is applicable when right child of left child of the critical node is heavy.

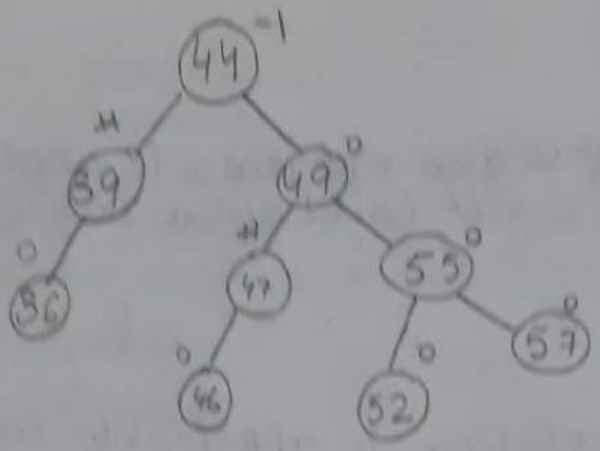
(III) RL rotation :-

RL or right left rotation is applicable when the left child of right child of the critical node is heavy. RL rotation is performed by LL rotation followed by RR rotation.
consider the following diagram :-



Eg :-



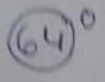


↷ LL (if left imbalance)
↶ RR (if right imbalance)
↶↑ LR (<)
↷↑ RL (>)

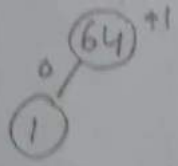
8) Construct an AVL Search tree using the sequence of nodes:-

64, 1, 14, 26, 13, 110, 98, 85.

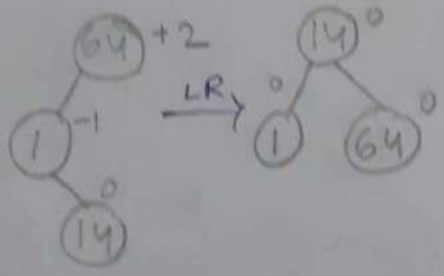
① Insert 64



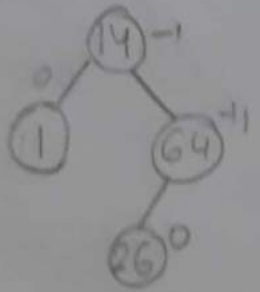
② Insert 1, 1<64



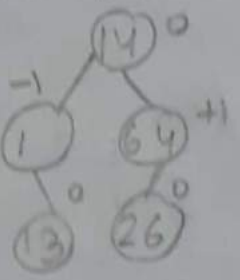
③ Insert 14, 14<64, 14>1



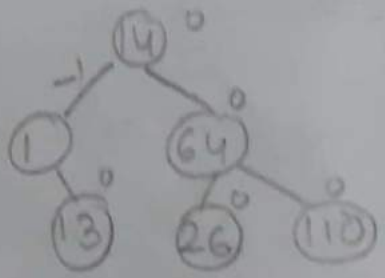
④ Insert 26, 26>14, 26<64



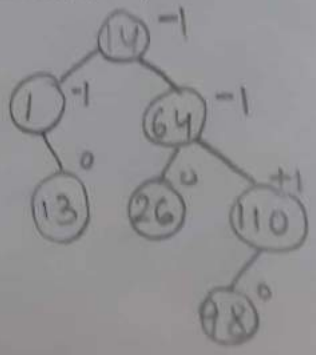
⑤ Insert 13, 13<14; 13>1



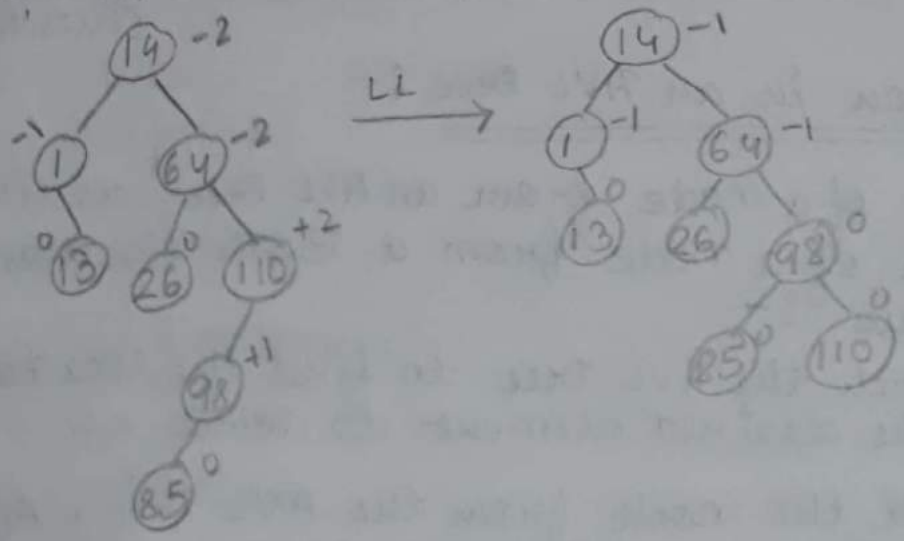
⑥ Insert 110, 110>14, 110>64

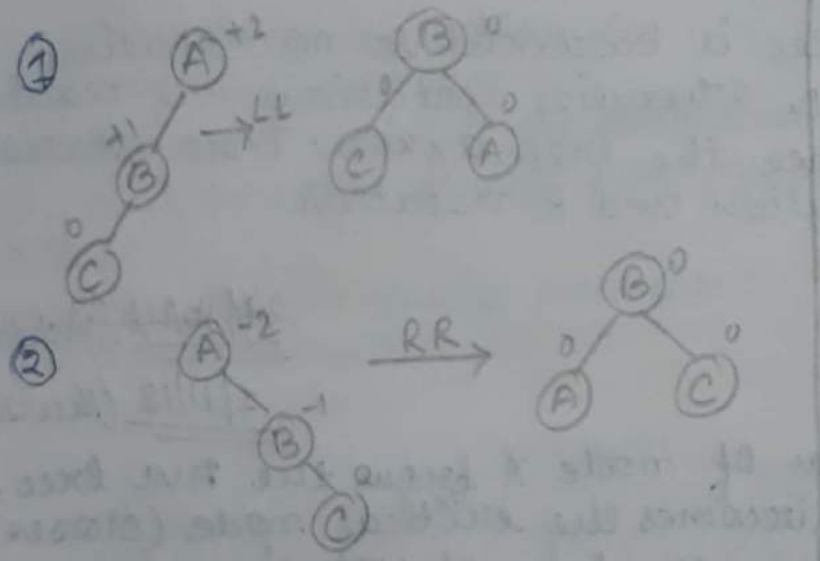


⑦ Insert 98, 98>14, 98>64, 98<110



⑧ Insert 85, 85>14, 85>64, 85<110, 85<98.





9/10/18
(Tuesday)

(III) Deletion in an AVL tree :-

Deletion of a node from an AVL tree is as like deletion of a node from a BST. The steps are as follows :-

1) Search the AVL tree to find the location of the desired element to delete.
2) Delete the node from the AVL tree. After that check the balance factor of each node of the tree.
3) If the tree is balanced so no rotation is essential. Otherwise rotations are required to rebalance the tree again. These rotations are L rotation and R rotation.

~~29/10/18 (Monday)~~
1/11/18 (Thursday)

On deletion of node X from the AVL tree, if node A becomes the critical node (closest ancestor) then the type of rotation depends on whether X is on the left sub tree of A or its right sub tree.

If X is on the left sub tree then L rotation is performed/applied.

If X is on the right sub tree of A then R rotation is applied.

(ii) Types of L rotation :-

It is 3 types :-

(a) L_0 Rotation

(b) L_1 rotation

(c) L_{-1} rotation

(ii) Types of R rotation :-

It is 3 types :-

(a) R_0 rotation

(b) R_1 "

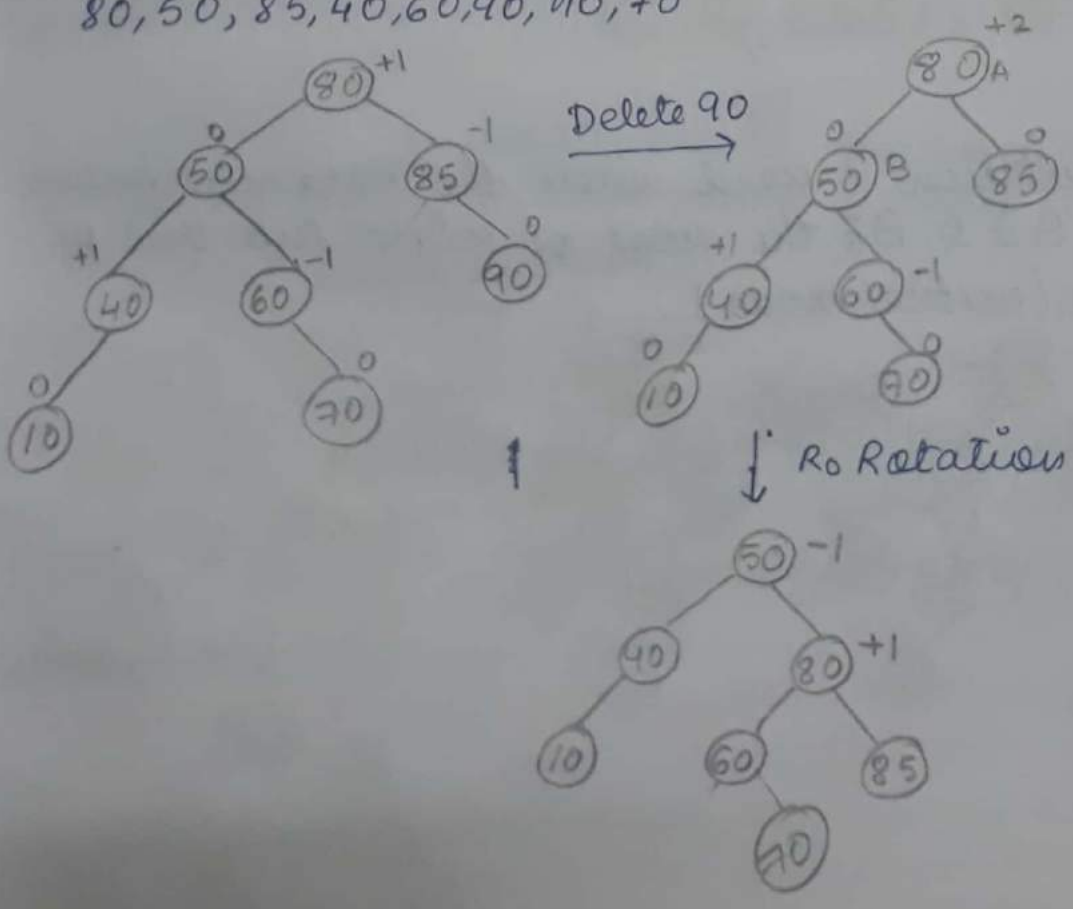
(c) R_{-1} "

(iii) L_0 and R_0 rotation :-

R_0 rotation is used if the balanced factor of B is 0. B is the root of left subtree of A (critical node).

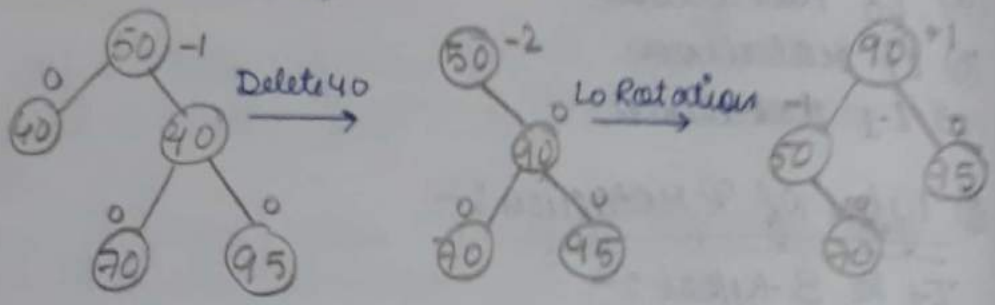
Consider the following example :-

80, 50, 85, 40, 60, 90, 10, 70



Lo rotation is used if the balanced factor of B is 0. B is the root of left right sub tree of A (critical node)

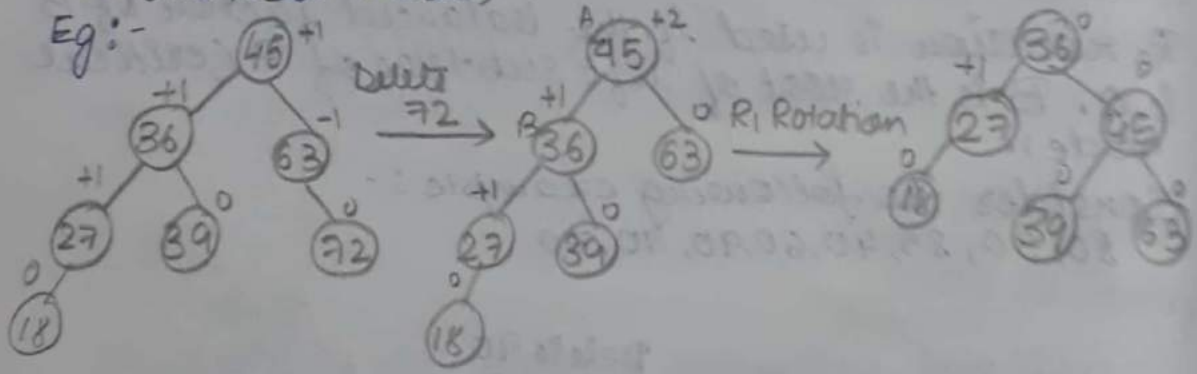
Eg :-



① R_1 and L_1 Rotation :-

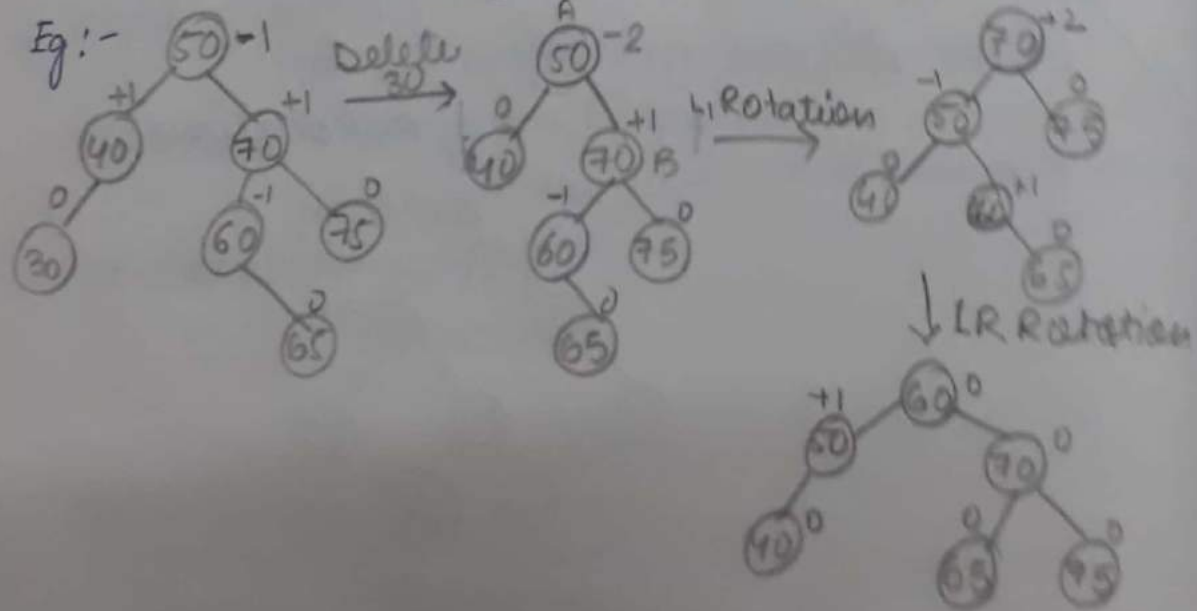
R_1 Rotation is used when the balanced factor of B is 1. B is the root of left sub tree of A (critical node).

Eg :-



L_1 rotation is used when the balanced factor of B is 1. B is the root of right sub tree of A (critical node).

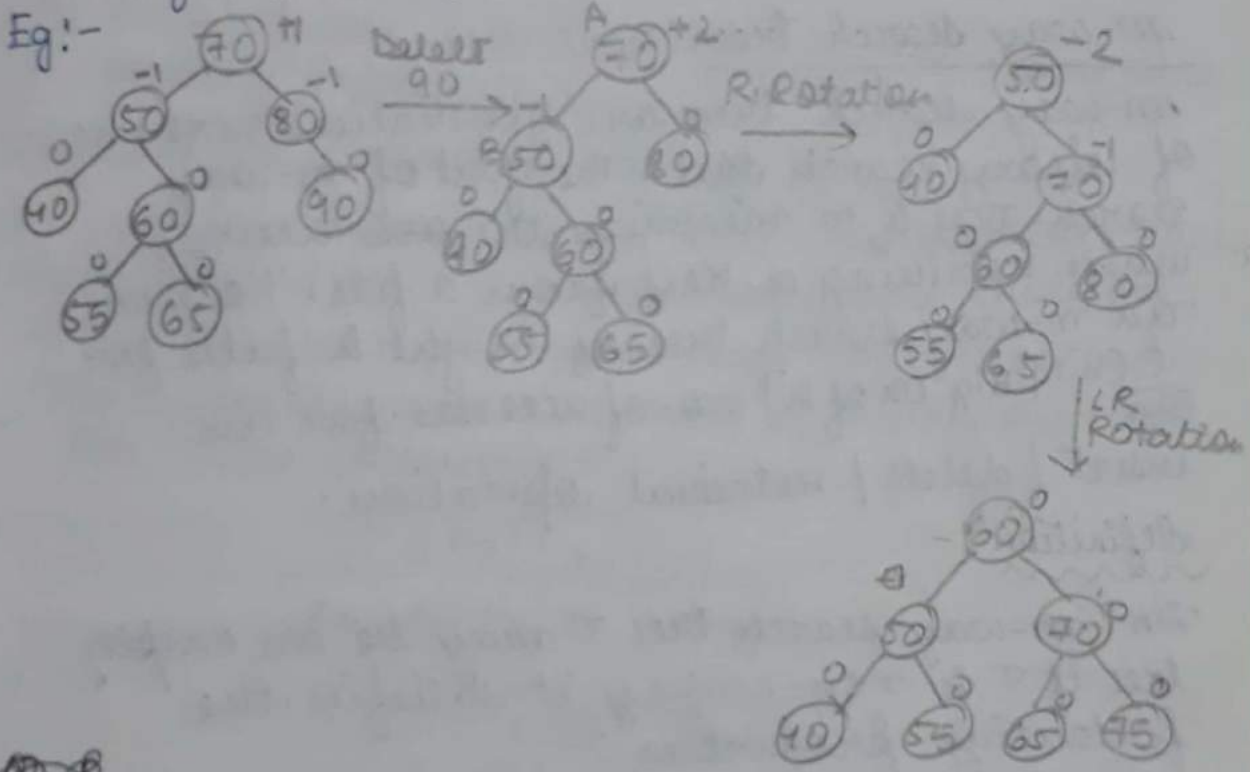
Eg :-



(11) R_{-1} and L_{-1} Rotation :-

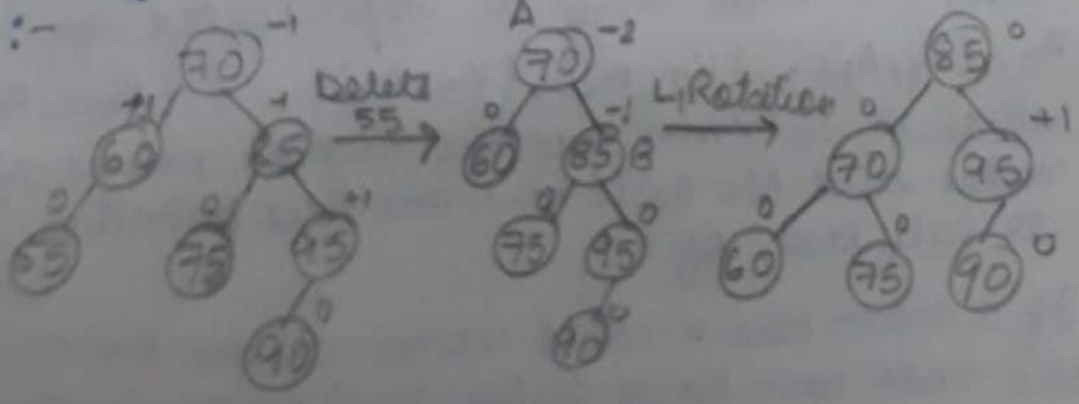
The R_{-1} rotation is applicable if the balanced factor of B is -1. B is the root of left sub tree of A (critical node)

Eg:-



L_{-1} rotation is applicable if the balanced factor of B is -1. B is the root of right sub tree of A (critical node)

Eg:-



(iii) Structure of a BST :-

Pointer to left subtree	Value of Key of the node	Pointer to right subtree

m-way search tree :-

m-way search trees are generalised versions of binary search tree. The goal of m-way search tree is to minimize the ~~ass~~ accesses while retriving a key from a file. However, an m-way search tree of height h falls for O(h) [Big Oh of h] no. of accesses for an insert / delete / retrieval operation.

Bigo

Definition :-

An m-way search tree T may be an empty tree if T is non-empty it satisfies the following properties :-

1) For some integer m, known as order of the tree, each node is of degree which can reach a maximum of m. In other words each node has atmost m child nodes. A node may be represented as $A_0, (K_1, A_1), (K_2, A_2), (K_3, A_3), \ldots\ldots, (K_{m-1}, A_{m-1})$ where K_i, $1 \le i \le m-1$ are the keys or values of nodes and A_i, $0 \le i \le m-1$ are the pointers to sub trees of T.

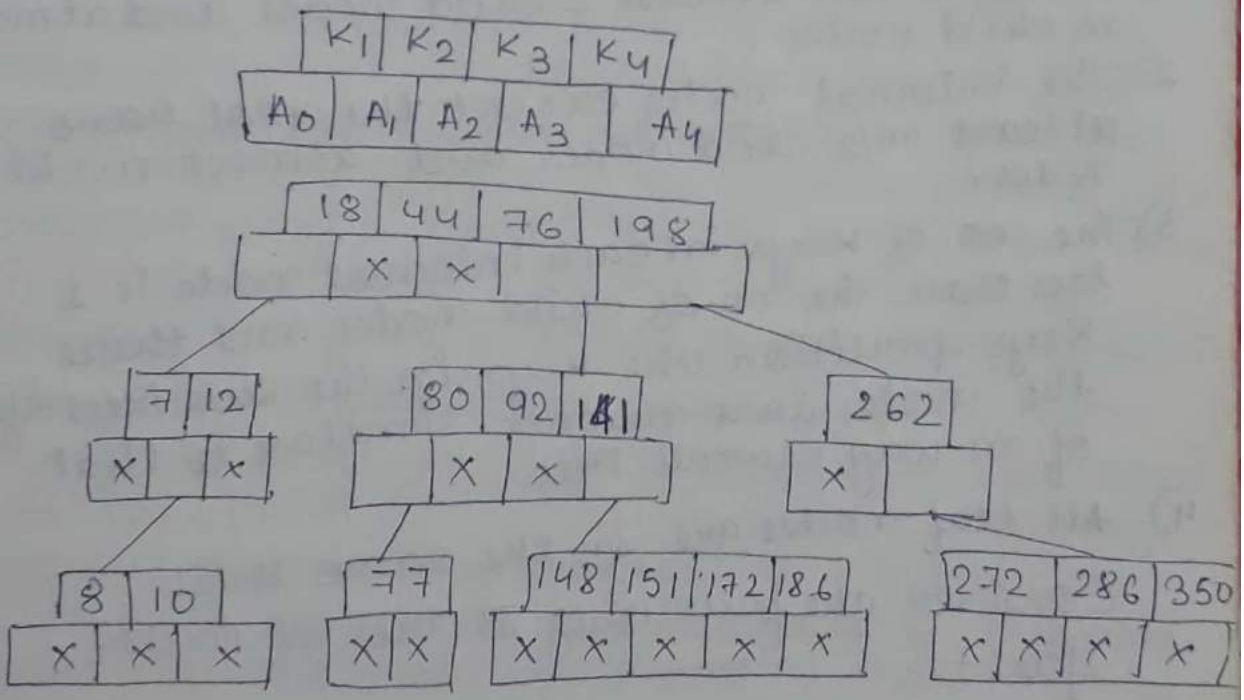
2) If a node has K child nodes where $K \le m$ then the node can have only (K-1) keys - $K_1, K_2, \ldots K_{K-1}$ contained in the node such that $K_i \le K_{i+1}$ and each of the keys partitions all the keys in the sub trees into K subsets.

3) For a node $A_0, (K_1, A_1), (K_2, A_2) \ldots (K_{n-1}, A_{n-1})$ all key values in the sub tree pointed to by A_i are less than the key K_{i+1}, $0 \le i \le m-2$ and all key value in the sub tree pointed to by A_{m-1} are greater than K_{m-1}.

4) Each of the sub trees A_i, $0 \le i \le m-1$ are also m-way search tree.

Consider the following example :-

It is a 5 way search tree. In this tree each node has atmost 5 child nodes and therefore, each node has atmost 4 keys contained in it. The general node structure of 5 way search tree is as follows :-



(11) B-Tree :-

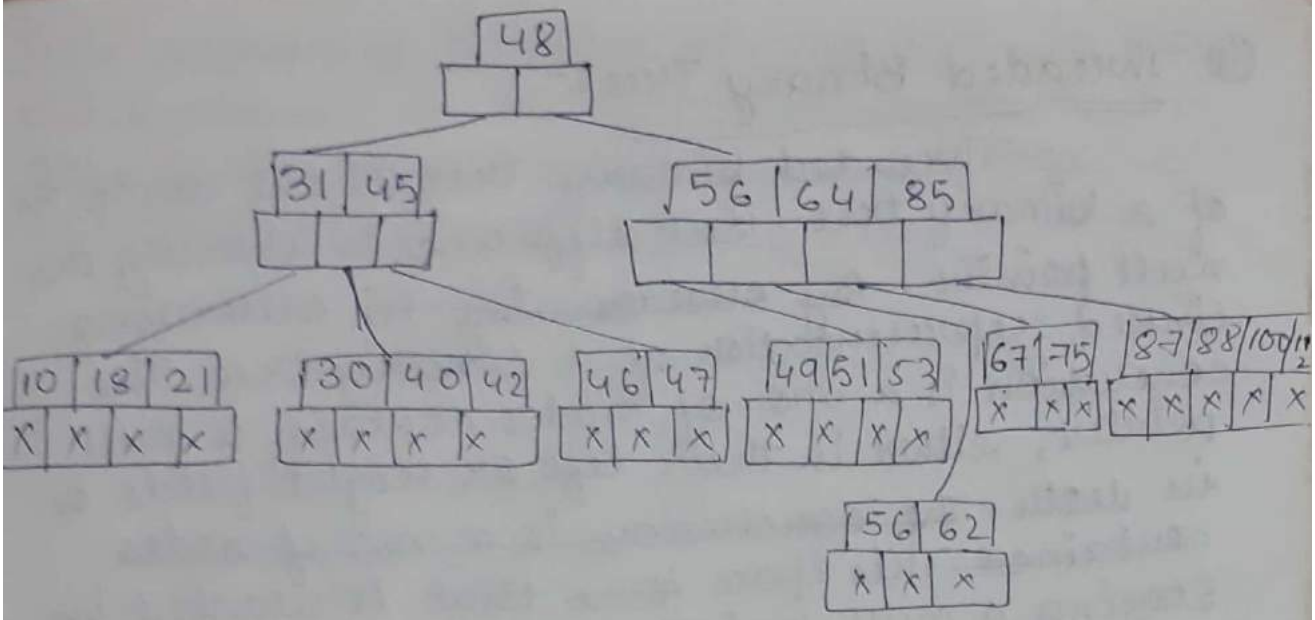
A B-tree is a specialized m-way search tree ie; used for disk access. A B-tree may contain a large no. of key values and pointers to sub trees. Storing a large no. of keys to a single node the height of the tree relatively small. B-tree is designed to store a sorted data and allows search, insert and delete operation to be performed in logarithmic, amortized time.

Definition :-

A B-tree of order m and it is non empty then it is an m-way search tree in which :-

1) The root has atleast 2 child nodes and atmost m child nodes.
2) The internal nodes except the root have atleast m/2 child nodes and atmost m child nodes.
3) The no. of keys in each internal node is 1 less than the no. of child nodes and these keys partition the keys in the sub trees of the node in a manner similar to that of m-way search tree.
4) All leaf nodes are on the same level.

Consider the following B-tree of order 5 :-



(iii) B+ Tree :-

B+ tree is a variant of B-tree which stores sorted data in a way that allows for efficient insertion, retrival, deletion of records where each of which identified by a key. B+ tree stores all the records at leaf level of the tree and keys are stored in the interior nodes. The leaf nodes of a B+ tree are linked to one another in a linked ~~to~~ list. It gives advantage of making the queries simpler and more efficient. Basically B+ trees are ~~needed~~ used to store large amount of data that cannot be stored in the main memory. So with B+ trees the secondary ~~memory~~ storage (CD, Hard disk) is used to store the leaf nodes of the tree and the internal nodes are stored in main memory.

B+ trees store data only in the leaf nodes. All other nodes or internal nodes called index nodes or I-nodes and store index values which allow to traverse the tree from the root down to leaf node that stores the desired data item.

(11) Threaded Binary Tree:-

Threaded binary tree is the same as of a binary tree. But difference is storing the null pointer. By ~~observing to~~ observing linked representation of a binary tree. The conclusion is a no. of nodes contain a null pointer, either in their left or right fields or in both. ~~The conclusion is a no. of nodes contained~~ This space ~~is, a~~ that is wasted in storing a null pointer can be efficiently used to store some other useful other information. For eg:- the null entries can be replaced to store a pointer to the inorder predecessor or inorder successor of the node. This special pointers are called threads and binary tree containing threads are called threaded binary tree.

Types:-

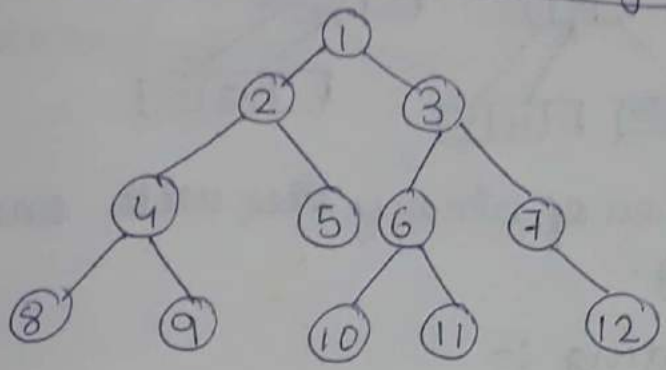
According to traverse threaded binary tree is of two types:-

(1) One way threading
(2) Two way threading

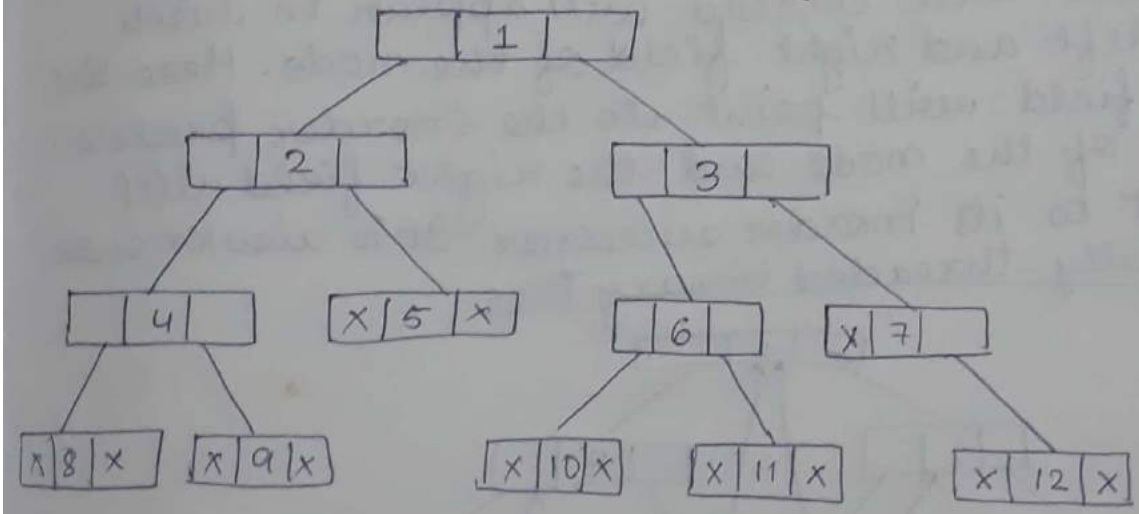
(11) One way threading:-

In one way threading a thread will appear either in the right field or the left field of the node. It is also called single threaded tree. If the threades appear in the left field then the left field will be made to point to the inorder predecessor of the node

It is called left threaded binary tree. If the thread appears in the right field then it will point to the inorder successor of the node. It is called right threaded binary tree.



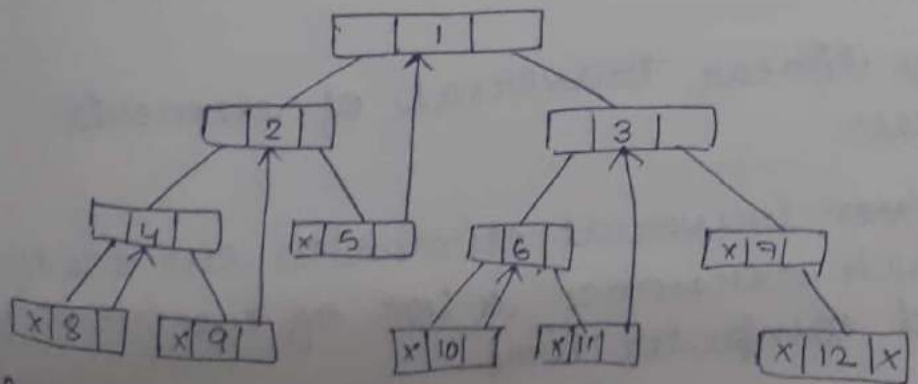
Binary Tree without threading:



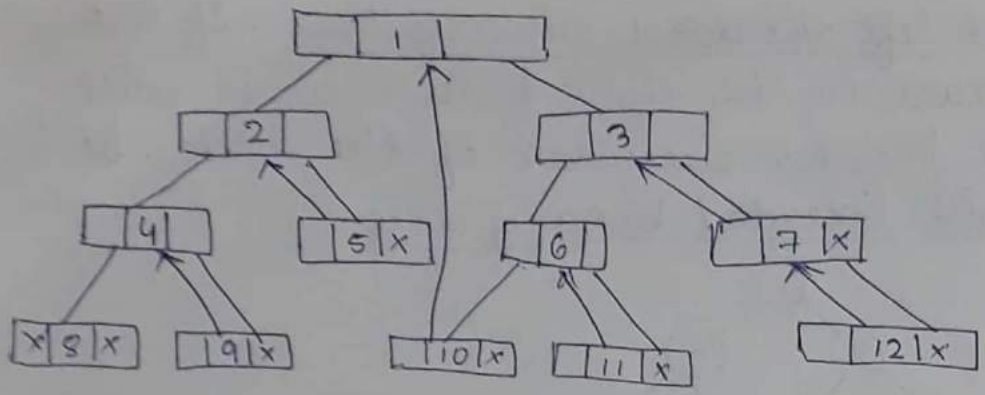
① Link representation of binary tree without threading :-

Inorder traversal :-

8 4 9 2 5 1 10 6 11 3 7 12



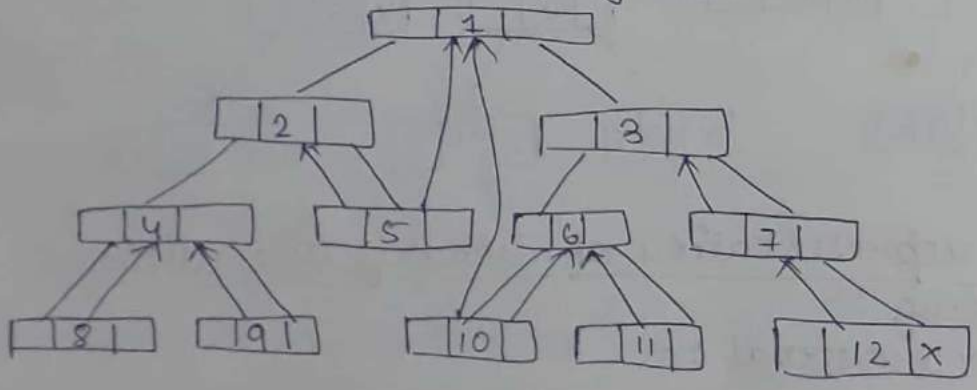
Link representation of binary tree with one way right threading



Link representation of binary tree with one way left threading.

(iii) Two way threading :-

It is also called double threaded binary tree. In this tree thread will appear in both the left and right field of the node. Here the left field will point to the inorder predecessor of the node and the right field will point to its inorder successor. It is also known as fully threaded binary tree.



Advantages :-

1. It enables linear traversal of elements in the tree.
2. Linear ~~trave~~ traversal eliminates the use of stack which consumes a lot of ~~me~~ memory space and computer time.

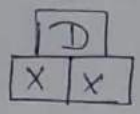
③ In It enables to find the parent of a given element without explicit use of parent-pointer.

④ Threaded binary tree enable forward and backward traversal of the nodes.

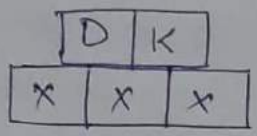
12/11/18 (Monday)

9) A 3 way search tree constructed out of an empty search tree with the following keys in the order as :- D, K, P, V, A, G.

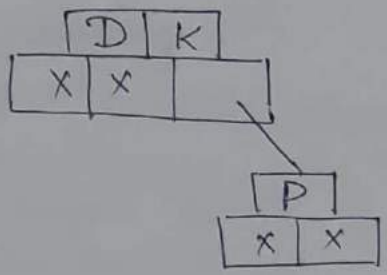
⇒ ① Insert : D



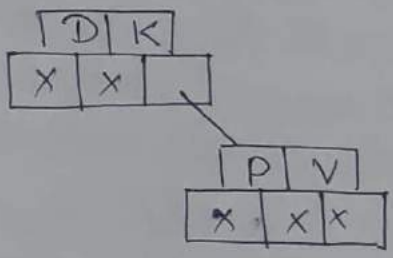
② Insert : K



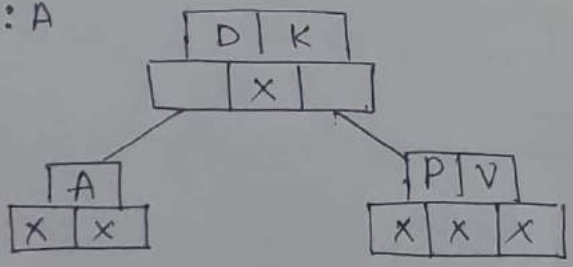
③ Insert : P



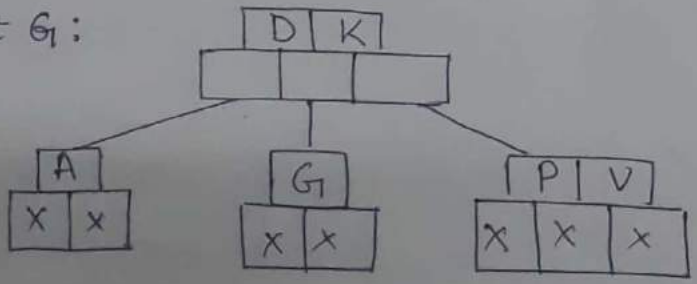
④ Insert : V



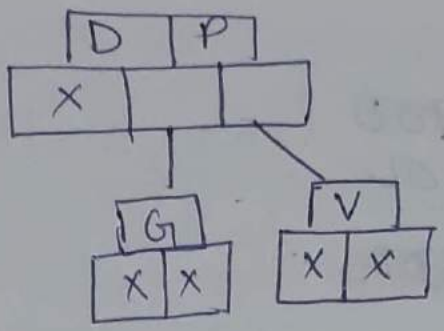
⑤ Insert : A



⑥ Insert G :



Delete A and K :



25/7/18 (Wednesda

9) Write a C program to implement the following operation into the array data structure:

(i) Create an array
(ii) Insert data elements into the array using function.
(iii) Traverse the array using function
(iv) Delete an element from the array using function

(v)

=>

```
#include<stdio.h>
void Insert();
void Traverse();
void Delete();
int A[10], i, n;
void main()
{
    Insert();
    Traverse();
    Delete();
}
void Insert()
{
    printf("\n Enter the limit :");
    scanf("%d", &n);
```

```
void Insert ();
void Traverse ();
void Delete ();
int A[10], i = -1;
void main()
{
    int n;
    do
    {
        Printf("*** Your choice ***");
        Printf("1. Insert");
        Printf("2. Delete");
        Printf("3. Traverse");
        Printf("Enter Your Choice: \n");
        Scanf("%d", &n);
        Switch(n)
        {
        Case 1:
            Insert ();
            break;
        Case 2:
            Delete ();
            break;
        Case 3:
            Traverse();
            break;
        Default:
            Printf("\n Your choice is invalid");
            break;
        }
    }
    while(n != 4);
    getch();
}
```

30/7/18 -
(Monday)

Insert

~~for(i=0; i<=max; i++)~~ ~~if(i<=max)~~
~~{~~ ~~pri~~
~~scanf("%d", &a[i]);~~
~~}~~

```
if (i<=max)
   printf("Enter the element");
while(i<=max)
{
   scanf("%d", &a[i]);
   i++;
}
else
  printf("\n The array is full");
```

Traverse

```
printf("The array elements are:");
{
```

1/08/18
(Wednesday)

```
void Delete ()
{
  int item, flag = 0, Pos ;
  Printf (" ..... ");
  Scanf ("%d", &item) ;
  for (j = 0 ; j <= i ; j ++)
  {
    if (a[j] == item)
    {
      flag = 1
      Pos = j ;
    }}
  {if (flag)
    for (j = pos ; j <= i ; j++)
    {
      a[j] = a[j+1]
    }
  i--;
  }
  else
  {Printf ("In The no. is not in the list-");
  }
  max = max - 1 ;
}
```

Merge:

```
void merged()
{
    int array1[100], array[100], array3[200];
    p=5;
    q=5;                    } for input of elements.
  [ Enter value to array 1:]
  [ Enter  "   "   "   2:]

    c=0;
    m=0;
    n=0;
  while((m<p)&&(n<q))
  {
      if(array1[m]<=array2[n])
            array3[c]=array1[m++];
      else
            array3[c]=array2[n++];
          c++;
      }
  while(m<p)
  {
      array3[c]=array1[m];
       c++;
       m++;
      }
{ while(n<q)
   array3[c]=array2[n];
  c++;
  n++;
}
```

[Display] ⇒ m = 0 → C

6/8/18
(Monday)

Stack

#define MAXSTK 10

```
void PUSH();
void POP();
void Traverse();
int STACK[MAXSTK];
int TOP=0;

void PUSH()
{
   int item;
   if(TOP==MAXSTK)
     {
        printf("\nThe stack is full");
     }
   else
   {
      printf("\n Enter the elements to be
                          inserted....");
      scanf("%d", &item);
      TOP = TOP+1;
      STACK[TOP] = item;
   }
}
```

```
void POP()
{
    int item;
    if(TOP==0)
    {
      printf("The Stack is empty");
    }
    else
    {
      item = STACK[TOP];
      TOP = TOP-1;
      printf("\n Pop %d", item);
    }
}

void Traverse()
{
    int i;
    if(TOP==0)
    {
      printf("\n The stack is empty...");
    }
    else
    {
      printf("\n The elements of stack are :");
      for(i=TOP; i>0; i--)
          printf("\n %d", STACK[i]);
      }
    }
```

Case 4:-
Exit.

10/9/18.

Sorting

```
void Create_list()
void Display_list()
Void Bubble Sort()

void main()
{
    Create_list();
    Bubble_sort();
    Displaylist();
```

10/10/18
(Wednesday)

(11) Bubble Sort Algorithm

Bubble_Sort (A, N)

This algorithm sorts the list A with N elements.

1. [Initialization]
   Set I = 0
2. Repeat Steps 3 to 5 until I < N.
3. Set J = 0
4. Repeat step 5 until J < (N-I-1)
5. If A[J] > A[J+1], then
   Set TEMP = A[J]
   Set A[J] = A[J+1]
   Set A[J+1] = TEMP
   End If
   [End of loop structure]
6. Exit

(11) Insertion Sort Algorithm:

This sorting is done by using very simple & easy technique. Here sorted list is build using one element at a time. In this technique programmers use a data element as key for comparisons. Suppose X[0] is the key element and X[1] is inserted and compared. If X[0] > X[1] then sorted list is X[1], X[0] otherwise X[0], X[1]. Then X[2] is inserted into the sorted list — suppose X[1], X[0] and then X[1], X[0], X[2] are compared to each other and then create a sorted list — Suppose X[2], X[1], X[0]. The remaining part of list X is inserted.

Using this order the unsorted data elements are iteratively inserted into the sorted list one by one and create sorted list at the K^{th} iteration, the X[K] is inserted into sorted X[0] to X[K-1] and then after comparison X[0] to X[K] are sorted.

The important idea behind the insertion sort is that at every iteration the newly inserted item is placed at its actual position in the final sorted list by ~~using~~ performing the comparison ~~to each element~~ in between the elements of the sorted list. However, at any intermediate

time we can observe that in the primary list there are 2 parts :-

Left side of the key element is sorted and right side of key element is unsorted.

Algorithm :-

Insertion_Sort (A, N)

This algorithm sorts the list A with N elements.

1. Set K = 1
2. Repeat Step 3 to 6 for K = 1 to n.
3. Set TEMP = A[K]
4. Set J = K - 1
5. Repeat while TEMP < A[J]

   Set A[J+1] = A[J]

   Set J = J + 1

   [End of loop structure]
6. Set A[J+1] = TEMP

   [End of step 2 loop]
7. Exit

29/10/18
(Monday)

GRAPH THEORY & ITS APPLICATION

A graph is an abstract data structure that is used to implement the graph concept from mathematics. It is basically a collection of vertices and edges that connect these vertices. A graph is viewed as a generalization of the tree structure where instead of having a purely parent-to-child relationship between the tree nodes, any kind of complex relationship can be presented.

(II) Why are graphs useful?

⇒ Graphs are widely used to model any situation where entities or things are related to each other in pairs.

For Eg:- The following information can be represented by graphs:

(1) Family trees in which the member nodes have an edge from parent to each of their children.

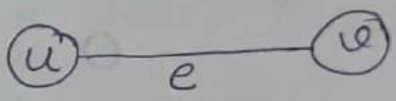
(2) Transportation networks in which nodes are airports, intersections, ports, rail stations, rail junction etc. The edges can be airline flights, bus ~~root~~ routes, shipping routes, rail lines etc.

(II) Definition of Graph :-

A graph G consists of 2 things :-

(1) A set V of elements called nodes or points or vertices.

(2) A set E of edges such that each edge e in E is identified with a unique (unordered) pair $[u, v]$ of nodes in V, denoted by $e = [u, v]$.



Sometimes, we indicate the parts of a graph by writting $G = (V, E)$. However $e = [u, v]$ means that u and v are the end point of e. u and v are also called adjacent nodes or neighbours.

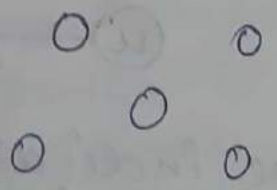
(III) Graph Terminology :-

(1) Adjacent nodes / neighbours - For every edge $e = [u, v]$ that connects nodes u and v. u and v are the end points of e and u and v are called adjacent nodes or neighbours.

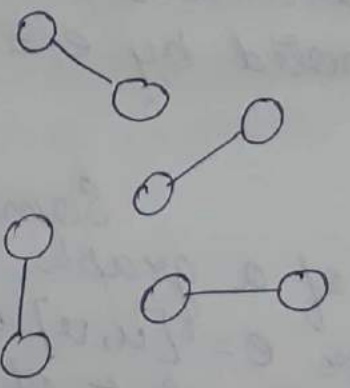
(2) Degree of a node - Degree of a node u, deg(u), is the total no. of edges containing the node u. If deg(u) = 0, it means that u does not belong to any edge and such a node is known as an Isolated node.

(3) Regular Graph - It is a graph where each vertex has the same no. of neighbours. That is, every node has the same degree.

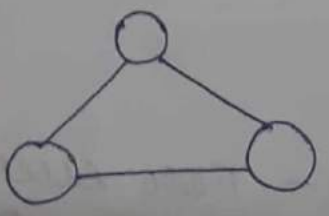
Eg:-

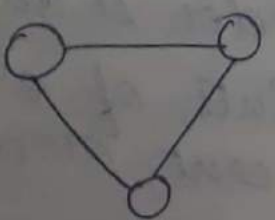


0 Regular graph



1 regular graph





2 Regular Graph

(4) Path – A path P is written as :-

$P = \{v_0, v_1, v_2, v_3, \ldots, v_n\}$ of length n from a node u to v is defined as a sequence of $(n+1)$ nodes. Here $u = v_0$; v_{i-1} is adjacent to v_i for $i = 1, 2, 3, \ldots n$ and $v_n = v$.

(5) Closed Path – A path P is known as closed path if the edge has the same end point ie; $v_0 = v_n$.

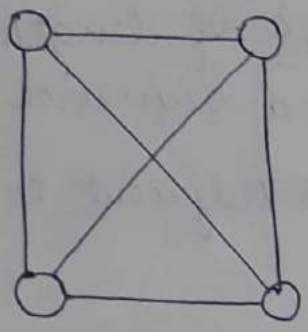
(6) Simple path – The path P is said to be simple path if all nodes are distinct with an exception that v_0 may be equal to v_n. If $v_0 = v_n$ then the path is called a closed simple path.

(7) Cycle – A closed simple path with length 3 or more is known as a cycle. A cycle of length K is called K cycle.

(8) Connected Graph – A graph G is said to be connected graph if there is a path between any 2 of its nodes. In a connected graph there is no isolated node.

** A connected graph that does not have any cycle is called a tree.

(9) Complete Graph – A graph G is said to be complete if all its nodes are fully connected. A complete graph has $\frac{n(n-1)}{2}$ edges where n is the no. of nodes in the graph.



$n = 4$

$\frac{4(4-1)}{2} = \frac{4 \times 3}{2} = 6$

(10)

Size of a graph – It is the total no. of edges in the graph.

(11) Labeled Graph – A graph G is said to be labeled if its edges are assigned data.

(12) Weighted graph – A graph G is said to be weighted if each edge in G is assigned a non-negative numerical value.

(13) Loop – An edge that has identical end points is called a loop ie; $e = (u, u)$.

~~(14) Multiple~~

(14) Multiple Edge – e and e' are called multiple edges if they connect the same end point.

(IIII) Directed Graph / Di-Graph :

A directed graph G is also called a di-graph, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G.

For an edge $e = (u, v)$:—

(1) e begins at u and ends at v.

(2) u is the origin of e and v is the destination of e.

(3) u is the predecessor of v and v is the successor of u.

(4) u is the adjacent of v and v is the adjacent of u.

(III) Terminology of a directed graph :-

(1) Out degree of a node - The out degree of a node u, outdeg(u), is the no. of edges that originate at u.

(2) In degree of a node - The in degree of a node u, indeg(u), is the no. of edges that terminate at u.

(3) Degree of a node - It is equal to the sum of indegree and out degree of a node. It is written as deg(u).

(4) Source - A node u is known as source if it has a positive out degree but a zero in degree.

(5) Sink - A node u is known as sink if it has a positive in degree but a zero outdegree.

(6) Reachability - A node v is said to be reachable from u, if and only if there exist a directed path from u to v.

(11) Transitive closure of a directed graph:

A transitive closure of a graph is constructed to ~~answer ensure~~ answer reachability questions. ie; is there a path from a node A to node E in one or more ?

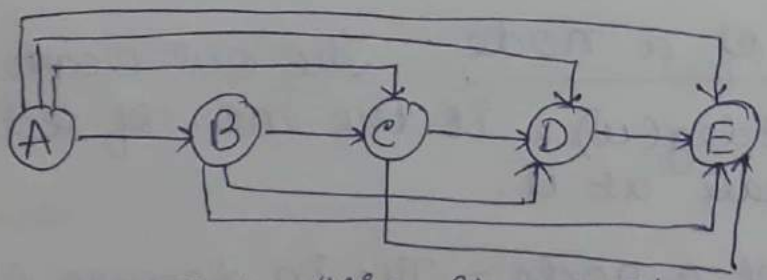
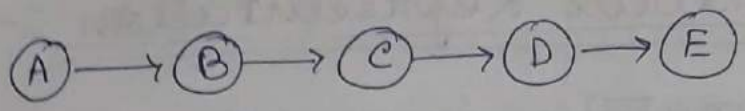


Fig:- Transitive closure of graph G

A binary relation indicates only whether the node A is connected to node B and B is connected to C and so on. But once the transitive closure is constructed then we can easily determine whether node E is reachable from node A in O(1) time or not.



Simple Graph

~~definition~~ :-

(III) Application :

Finding the transitive closure of a directed graph is an important problem in the following computational task :-

(1) Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallal systems.

(2) It is used in the construction of parsing automata in compiler construction.

(3) Recently transitive closure computation is being used to evaluate recursive database queries.

(III) Representation of Graphs :-

There are 2 common ways of storing graphs in the computer's memory. They are :-

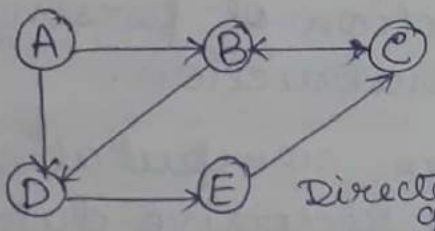
(1) Sequential representation by using an adjacency matrix.

(2) Linked representation by using an adjacency list that stores the neighbours of a node using a linked list.

(iii) Adjacency Matrix Representation :-

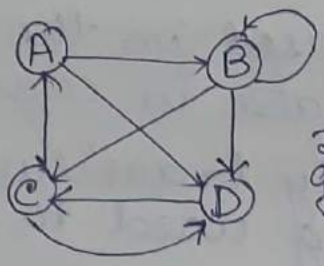
A adjacency matrix is used to represent which nodes are adjacent to one another. By definition, 2 nodes are said to be adjacent if there is an edge ~~connectio~~ connecting them. In an adjacency matrix, the rows and columns are labeled by graph vertices. The variable ~~a_{ij} will be~~ $a_{ij} = 1$ if vertices v_i and v_j are adjacent to each other, otherwise $a_{ij} = 0$.

Now consider some adjacency matrix for different graph :-



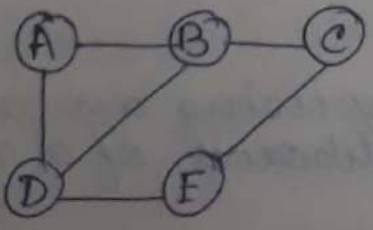
Directed graph

	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	1	0
C	0	1	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0



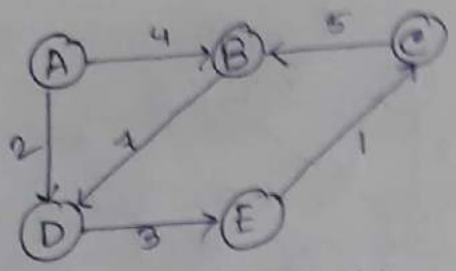
Directed graph with loop.

	A	B	C	D
A	0	1	0	1
B	0	1	1	1
C	1	0	0	1
D	0	0	1	0



Undirected

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0



weighted graph

	A	B	C	D	E
A	0	4	0	2	0
B	0	0	0	7	0
C	0	5	0	0	D
D	0	0	0	0	9
E	0	0	1	0	0

30/10/18

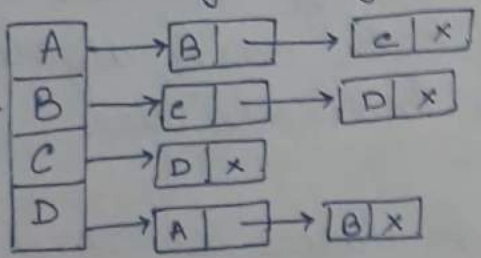
(Tuesday)

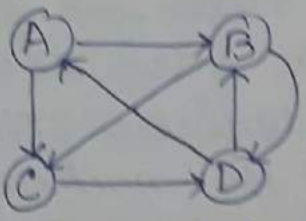
(iii) Adjacency List:

It is another way to represent a graph in the computer memory. This structure consists of a list of all nodes in graph G. Here every nodes is linked to its own list that contains the names of all other nodes that are adjacent to it.

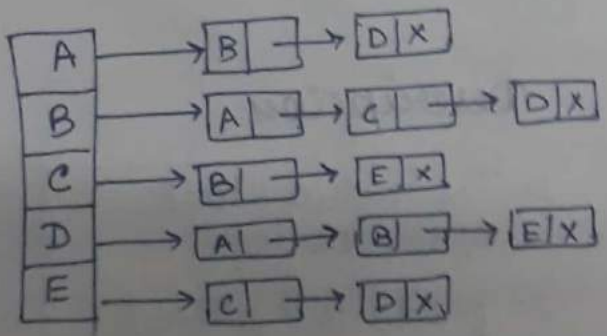
Consider the following eg:-

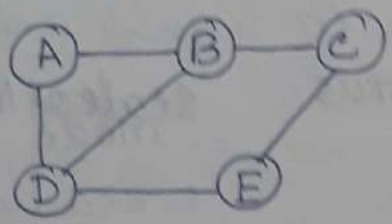
Adjacency list



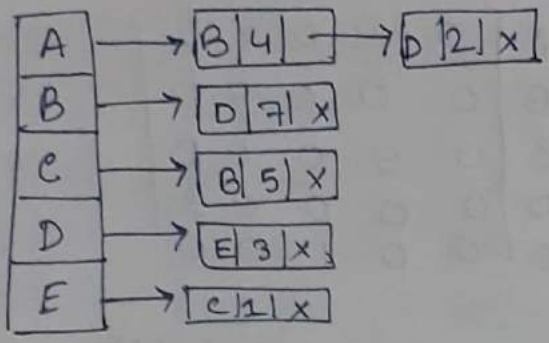


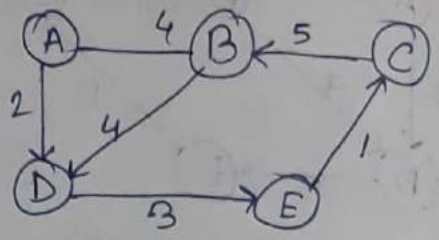
Directed Graph





Undirected Graph





weighted graph

(III) Graph Traversal :-

In this section, we will discuss how to traverse graph. By traversing a graph we mean the method examining the nodes and edges of the graph. There are 2 standard methods of graph traversal :-

(1) Breadth first Search (BFS)

(2) Depth first Search (DFS)

DFS uses a queue as an auxilliary data structure to store nodes for further processing. DFS uses a stack to store nodes but in both cases, algorithm uses a variable STATUS which indicates the state of the nodes.

Consider the following table of value of STATUS and its significant :-

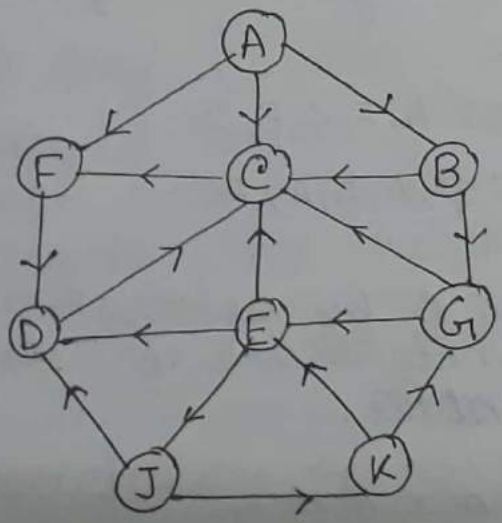
STATUS	State of the node	Description
1	1. Ready	1. The initial state of the node N
2	2. Waiting	2. Node N is placed in the queue or stack and waiting to be processed.
3	3. Processed	3. Node N has been completely processed

① BFS Algorithm :-

It is a graph search algorithm that begins at the root node and explore all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes and so on until it finds its goal.

31/10/18
(Wednesday)

8) Consider the graph G and its adjacency list. Assume that G represents the daily flights b/w different cities and suppose we want to fly from city A to city J with minimum no. of stops. In other words we want minimum path P from A to J.



Adjacency List

A: F, C, B
B: G, C
C: F
D: C
E: D, C, J
F: D
G: C, E
J: D, K
K: E, G

Ans :-

The minimum path P can be found by using BFS beginning at city A and ending when J is encountered. During the execution of the search, we will also keep track of the origin of each edge by using an array ORIG together with the array QUEUE.

The steps are as follows :-

a) Initially add A to QUEUE and add null to ORIG as follows :-

FRONT = 1 QUEUE : A
REAR = 1 ORIG : ϕ

b) Remove the FRONT element A by setting FRONT = FRONT + 1 and add to QUEUE the neighbours of A as follows :-

FRONT = 2 QUEUE : A, F, C, B
REAR = 4 ORIG : ϕ, A, A, A

c) Remove FRONT element F by setting FRONT = FRONT + 1 and add to QUEUE the neighbours of F as follows :-

FRONT = 3 QUEUE : A, F, C, B, D
REAR = 5 ORIG : ϕ, A, A, A, F

d) Remove FRONT element C by setting FRONT = FRONT + 1 and add to QUEUE the neighbours of C. ~~as follows:~~ F already in ready state.

FRONT = 4 QUEUE : A, F, C, B, D
REAR = 5 ORIG : ϕ, A, A, A, F

e) Remove FRONT element B by setting FRONT = FRONT + 1 and add to QUEUE the neighbours of B. C already in ready state.

FRONT = 5 QUEUE : A, F, C, B, D, G
REAR = 6 ORIG : ϕ, A, A, A, F, B

f) Remove FRONT element D by setting FRONT = FRONT+1 and add to QUEUE the neighbours of D.

FRONT = 6 QUEUE: A, F, C, B, D, G
REAR = 6 ORIG: Φ, A, A, A, F, B

g) Remove FRONT element G by setting FRONT = FRONT + 1 and add to QUEUE the neighbours of G.

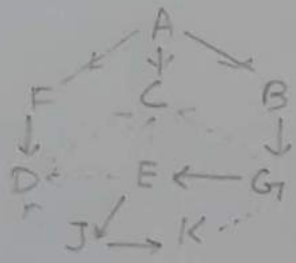
FRONT = 7 QUEUE: A, F, C, B, D, G, E
REAR = 7 ORIG: Φ, A, A, A, F, B, G

h) Remove FRONT element E by setting FRONT = FRONT+1 and add to QUEUE by neighbours of E.

FRONT = 8 QUEUE: A, F, C, B, D, G, E, J
REAR = 8 ORIG: Φ, A, A, A, F, B, G, E

i) We stop as soon as J is added to QUEUE, Since J is our final destination. We now back track from J using the array ORIG to find the path P.

P = J ← E ← G ← B ← A



(III) Application of BFS :-

(1) BFS can be used to solve many problems such as finding all connected components in a graph G.

(2) Finding shortest path b/w 2 nodes of an unweighted graph.

(3) Finding the shortest path b/w 2 nodes of a weighted graph.

~~(4) FF~~

(4) Finding all nodes within an individual connected component.

~~(III) Depth First Search~~

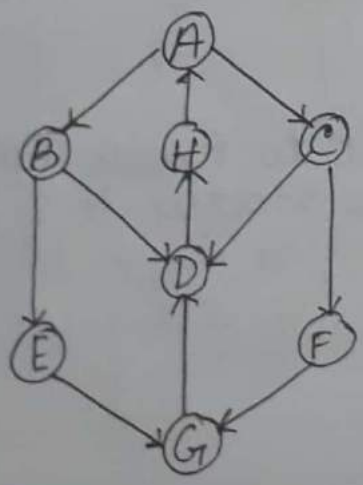
(II) Algorithm for BFS:

(III) Depth First Search :-

The DFS algorithm progresses by expanding the starting node of graph G and going deeper and deeper until a goal node is found or until a node that has no children is encountered.

In other words, DFS begins at a starting node A which becomes the current node, then we process the neighbour of A, then a neighbour of neighbour of A and so on. During the execution of the algorithm, if we reach a path that has a node which has been already processed then we back track to the current node. otherwise the unvisited node becomes the current node.

Q) Consider the following graph G :



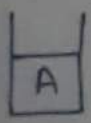
Adjacency List :

A : B, C
B : D, E
C : D, F
D : H
E : G
F : G
G : D
H : A

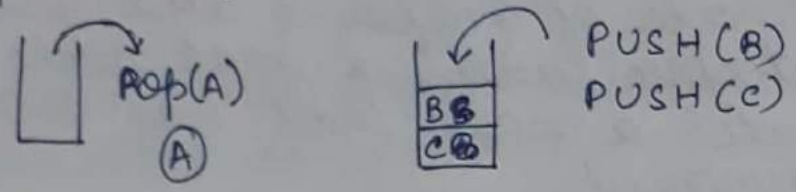
Ans :-

The steps are as follows :-

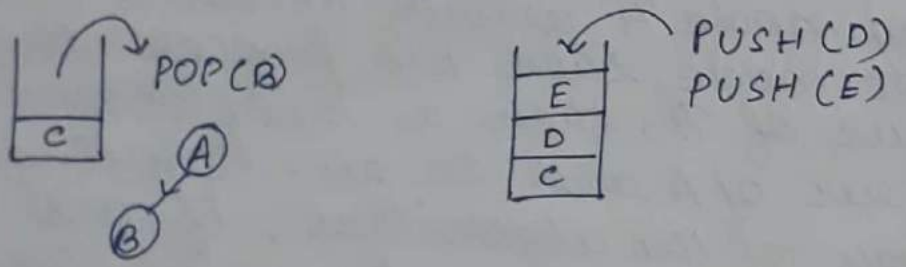
① Push A into STACK



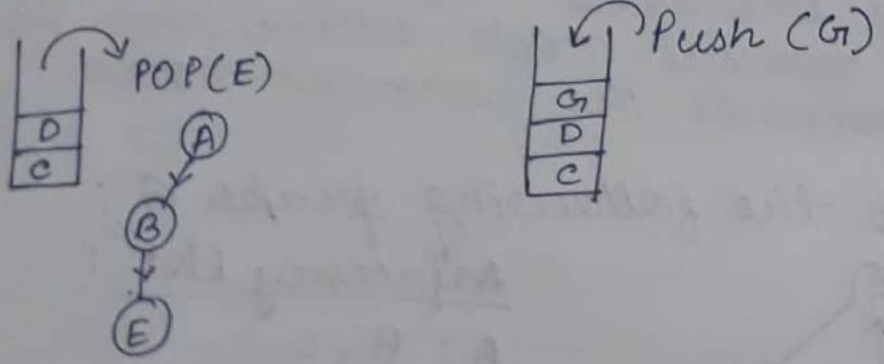
2) Pop and Print A and Push B and C (adjacence of A)



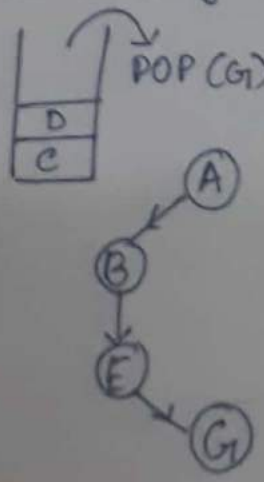
3) POP B and Print- B . Then Push D and E.



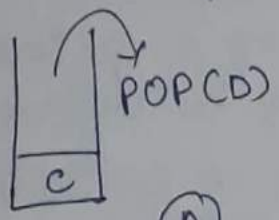
4) POP E and Print E . Then Push G.

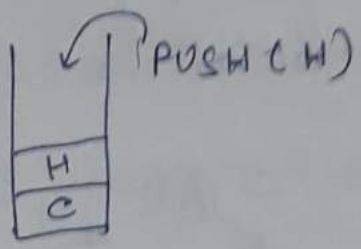


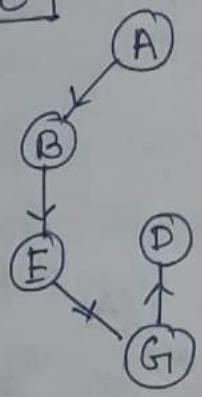
5) POP G and Print . No element to Push (as D is already inserted whose status is 2)



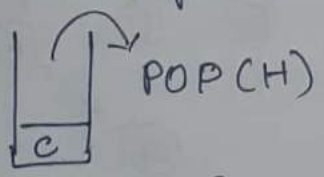
6) POP and print D. and Push H.

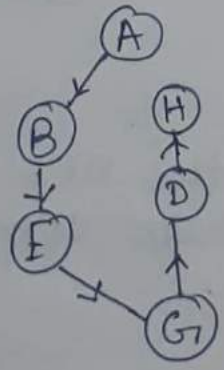




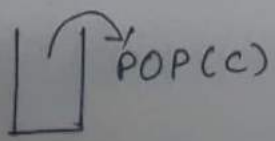


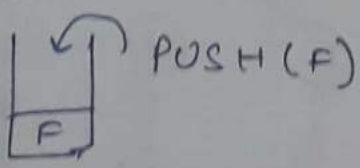
7) POP and print H. No element to push.

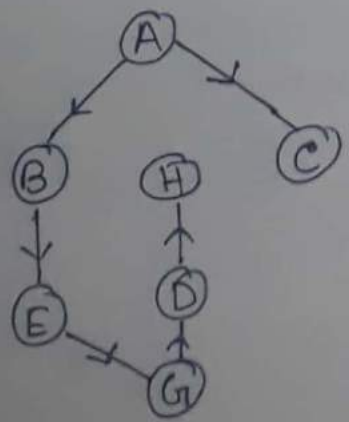




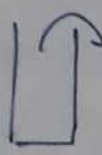
8) POP and print C and ~~Print~~ Push F

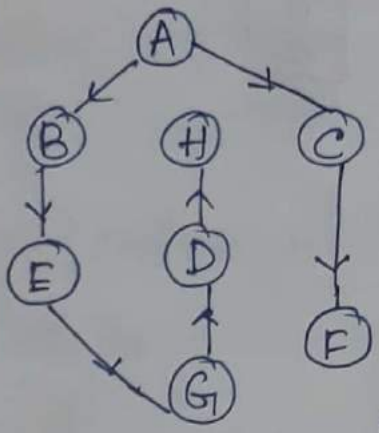




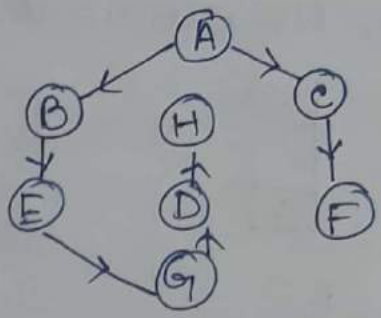


9) POP F and Print. NO element to Push.

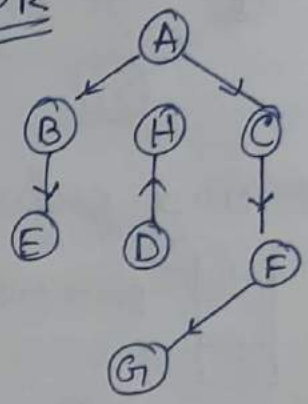




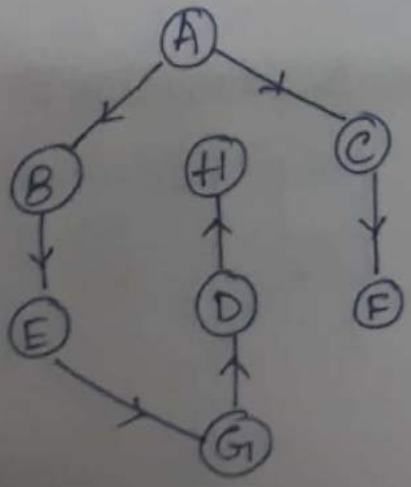
OR



OR



Since the ~~sta~~ STACK is empty the DFS of the graph G ~~is~~ starting at A is now complete.



A, B, E, G, D, H, C, F

(II) Applications of DFS:

1) Finding a path b/w 2 specified nodes of an unweighted graph.
2) Finding a path b/w 2 specified nodes of a weighted graph
3) Finding whether a graph is connected or not.
4) Computing the spanning tree of a connected graph.

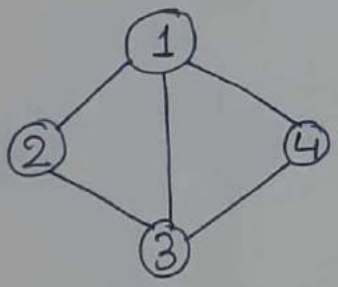
(III) Algorithm for DFS :-

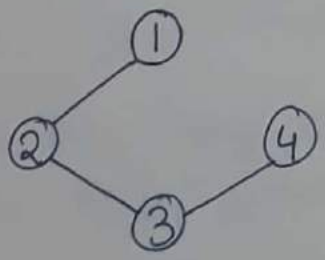
(IIII) Spanning Trees :-

DEFINATION :

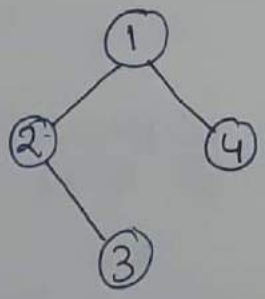
Let $G_1 = (V, E)$ be a connected undirected graph. A spanning of G_1 is a sub graph of G_1 ie; a tree containing every vertex of G_1. A spanning tree of $G_1 = (V, E)$ is a connected graph on V with a minimum no. of edges $|V| - 1$.

Eg :-

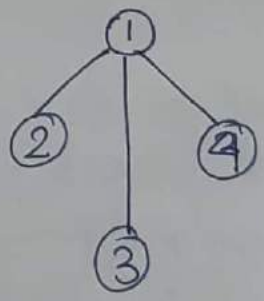




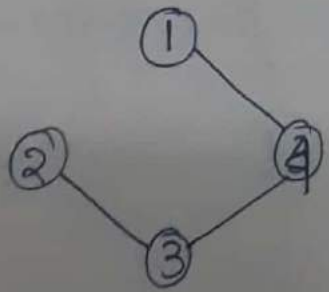
(1)



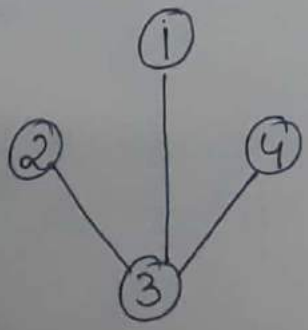
(2)



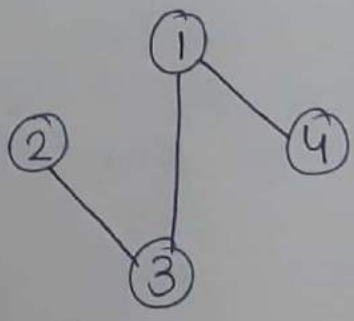
(3)



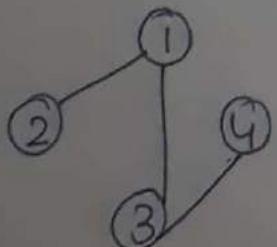
(4)



(5)



(6)



(7)

(III) Shortest path algorithm :-

There are different algorithms to calculate the shortest path blw the vertices in a graph G. These are :-

(1) Minimum Cost Spanning Tree

(2) ~~Disaster~~ Dijkstra's Algorithm

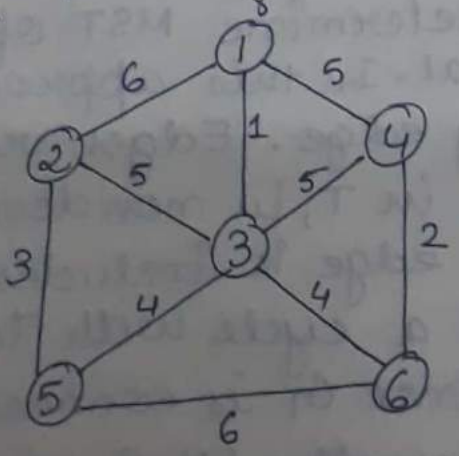
(3) Warshal's Algorithm

N.B

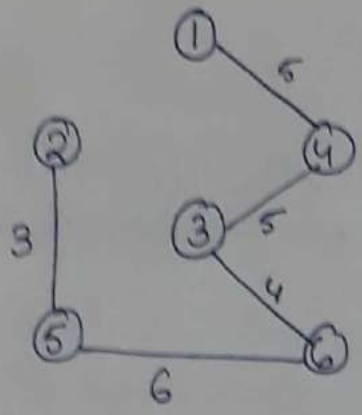
MST and Dijkstra use the adjacency list to find the shortest path whereas Warshal uses the adjacency matrix to find the shortest path.

(II) Minimum Cost Spanning Tree :

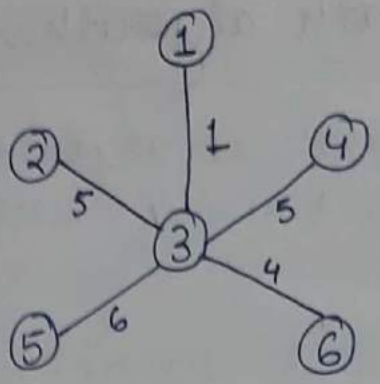
Let $G=(V,E)$ be a connected undirected graph with weights of its edges and (V,T) is a spanning tree of G. If G is weighted and the sum of the weights of the edges in the T is minimum, then (V,T) is called minimum cost spanning tree or minimum spanning tree.



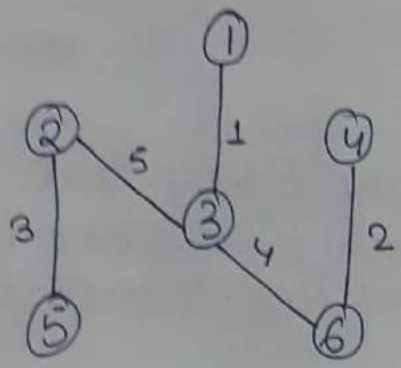
Total weight = 43



(a) Total Cost = 23



(b) Total Cost = 21



(c) Total Cost = 15

Therefore, Graph (c) is the MST ~~becoo~~ because its cost is minimum than (a) and (b)

~~1/11/2018 (Thursday)~~

~~(II) Dijkstra's Algorithm :-~~

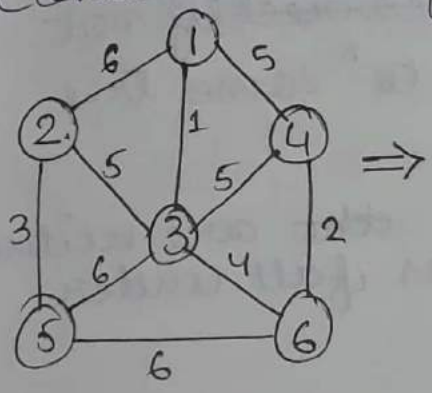
(III) Kruskal's Algorithm :-

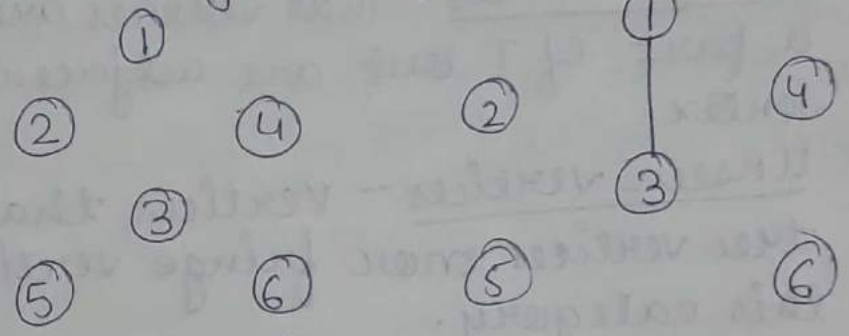
Another approach to determine MST of a graph has been given by Kruskal. In this approach a MST T is built edge by edge. Edges are considered for inclusion in T, in non decreasing order of their cost. An edge is included in T if it does not formed a cycle with the edges already in T. Since G is connected and has N>0 vertices, exactly (N-1) edge will be selected for inclusion in T.

The outline of the algorithm is given below:-

(1) Sort the edges by weight in ascending order.
(2) Select the lowest cost edge from the list. Remove this edge from list and add this edge to the list. If addition results in a cycle discard this edge.
(3) Stop when N-1 edges added to the tree.

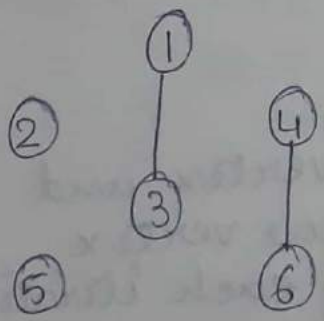
Consider the following example:-



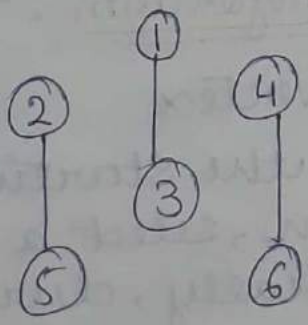


Initial configuration

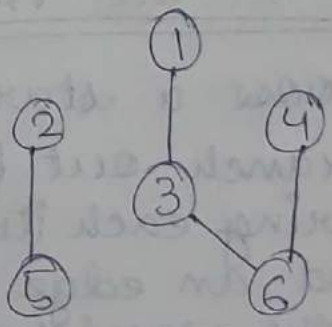
Step 1: Choose (1,3)



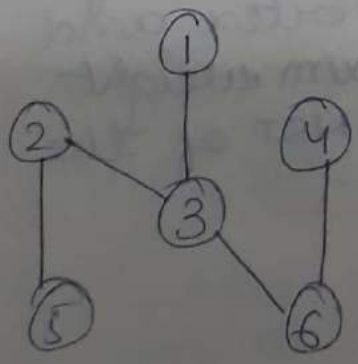
Step 2: Choose (4,6)



Step 3: Choose (2,5)



Step 4: Choose (3,6)



Step 5: Choose (2,3)

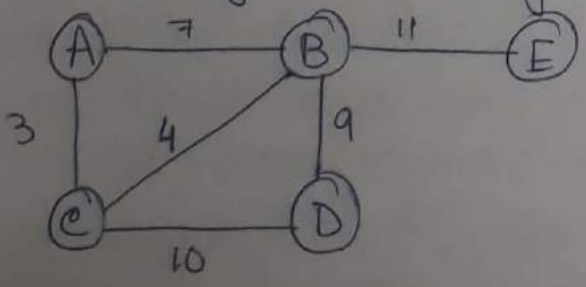
(11) Prim's Algorithm :-

There is another approach to find out the MST ie; Prim's algorithm. At first there are three kinds of vertices that will be used during the execution of Prim's algorithm.

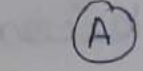
1. Tree vertices :- That vertices are a part of minimum spanning tree T.
2. Fringe vertices - That vertices are ~~correctly~~ currently not a part of T but are adjacent to some tree vertex.
3. Unseen vertices - Vertices that ~~the~~ are neither tree vertices nor fringe vertices fall under this category.

Steps in the Prim's Algorithm :-

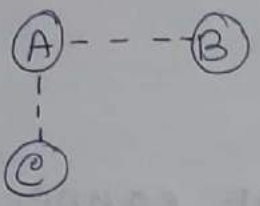
1) Choose a starting vertex
2) Branch out from the starting vertex and during each iteration, select a new vertex and an edge. Basically, during each iteration of the algorithm, we have to select a vertex from the fringe vertices in such a way that the edge connecting the tree vertex and the new vertex has the minimum weight assigned to it. Construct a ~~MST~~ MST of the graph using Prim's algorithm.



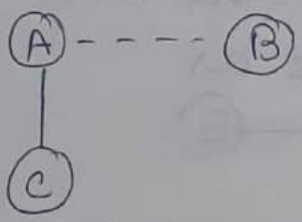
Step 1: Choose a starting vertex A



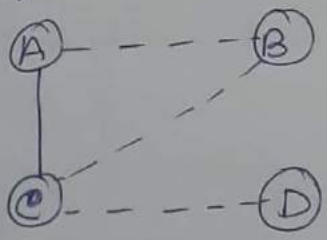
Step 2: Add the fringe vertices that are adjacent to A with dotted lines.



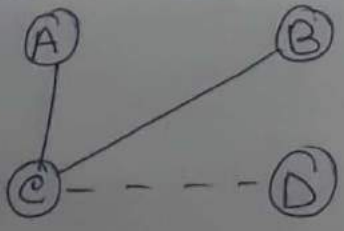
Step 3: Now among the fringe vertices select a vertex that has the minimum weight and then ~~at it~~ add it to the MST.T. Now C is the tree vertex as AC has less weight.



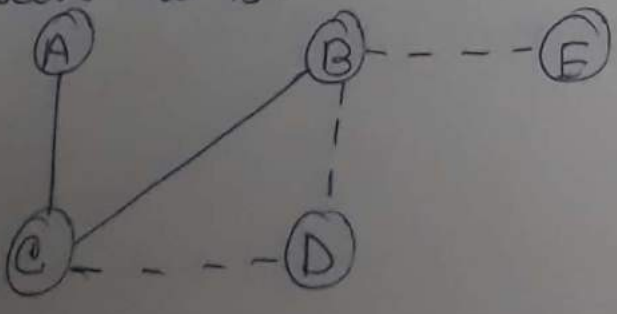
Step 4: Add the fringe vertices that are adjacent to C.



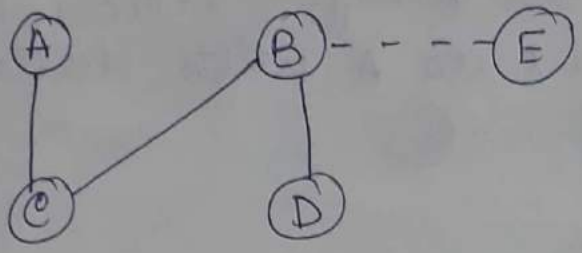
Step 5: Now as BC has less weight, B is added to tree. B becomes a tree vertex.



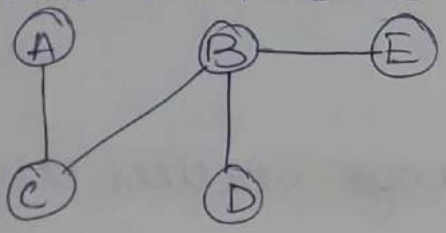
Step 6: Add the fringe vertices that are adjacent to B.



Step 7 : New BD has less weight and D is added to the tree. D becomes a tree vertex.



Step 8 : Note that E is not connected to T. So, it will be added in the tree because a minimum spanning tree is 1 in which all the N nodes are connected with N-1 edges that has minimum weight. So the MST is as



1/11/18 (Thursday)

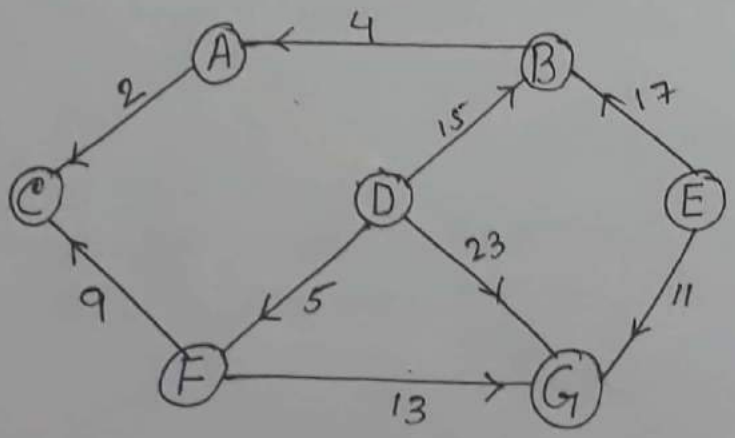
(III) Dijkstra's Algorithm :-

This algorithm was given by a Dutch scientist Edsger Dijkstra in 1959. It is used to find the shortest path tree. This algorithm is widely used in network routing protocols.

Suppose G is a graph and A is a source node. This algorithm is used to find the shortest path b/w A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths from a source node to a destination node.

Ex :-

Consider the graph G. Taking D as the initial node, execute the Dijkstra algorithm on it.



⇒

Step 1 : Set the label of D = 0 and N = {D}

Step 2 : Label of D = 0, B = 15, G = 23 and F = 5
Therefore N = {D, F}

Step 3: ~~Set~~ label of D = 0, B = 15, G has been relabeled 18 because minimum (5+13, 23) = 18, C has been relabeled 14 (5+9).
Therefore N = {D, F, C}

Step 4: Label of D = 0, B = 15, G = 18.
Therefore N = {D, F, C, B}

Step 5: Label of D = 0, G = 18, A has been relabeled 19 (15+4).
Therefore N = {D, F, C, B, G}

Step 6: Label of D = 0, and A = 19
Therefore N = {D, F, C, B, G, A}

Note that we have no labels for node E. This means that E is not reachable from D.
The reachable nodes from D are D, F, C, B, G, A

(III) Algorithm for Dijkstra's

(II) Warshal Algorithm :-

If a graph G is given as $G = (V, E)$, where V is the set of vertices and E is the set of edges. The path matrix of G can be found as

$$P = A + A^2 + A^3 + \ldots + A^n$$

This is a lengthy process. So, Warshal has given a very efficient algorithm to calculate the shortest path b/w 2 vertices.

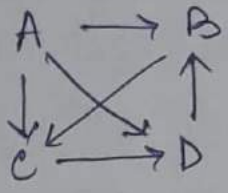
Warshal's algorithm defines matrices $P_0, P_1, P_2, \ldots$ as given as

$P_K[i][j]$ → 1 [If there is a path from v_i to v_j. The path should not use any other nodes except $v_1, v_2, v_3, \ldots, v_K$]

→ 0 [otherwise]

This means that if $P_0[i][j] = 1$ then there exist an edge from v_i to v_j.

If $P_1[i][j] = 1$ then there exist an edge from v_i to v_j that does not use any other vertex except v_1.



	A	B	C	D
A	0	1	1	0
B	0	0	1	0
C	0	0	0	1
D	0	1	0	0

2/11/18 (Friday)

Insertion Sort :-

85, 31, 24, 55, 26, 11, 20, 45

0	1	2	3	4	5	6	7
85	31	24	55	26	11	20	45

Trace of Algorithm :-

Iteration 1:

31 85 (24) 55 26 11 20 45

Iteration 2:

24 31 85 (55) 26 11 20 45

Iteration 3:

24 31 55 85 (26) 11 20 45

Iteration 4:

24 26 31 55 85 (11) 20 45

Iteration 5:

11 24 26 31 55 85 (20) 45

Iteration 6:

11 20 24 26 31 55 85 (45)

Iteration 7:

11 20 24 26 31 45 55 85

Then no element left for sorting. The final sorted list is :-

11 20 24 26 31 45 55 85

(11) Selection Sort :-

A[10] = {77, 33, 44, 11, 88, 22, 66, 55}

Trace of Algorithm :-

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
K=0 LOC=3	77	33	44	11	88	22	66	55
K=1 LOC=5	11	33	44	77	88	22	66	55
K=2 LOC=5	11	22	44	77	88	33	66	55
K=3 LOC=5	11	22	33	77	88	44	66	55
K=4 LOC=7	11	22	33	44	88	77	66	55
K=5 LOC=6	11	22	33	44	55	77	66	88
K=6 LOC=6	11	22	33	44	55	66	77	88
K=7 LOC=7	11	22	33	44	55	66	77	88
	11	22	33	44	55	66	77	88

Quick Sort :-

A[10] = {50, 30, 10, 90, 80, 20, 40, 60}

0	1	2	3	4	5	6	7
50	30	10	90	80	20	40	60

Choose Pivot = 50 End = 60
Start = 50

A[start] > Pivot ? No → 50 30 10 90 80 20 40 60 (Start →, End)

A[start] > Pivot ? No → 50 30 10 90 80 20 40 60 (Start →, End)

A[start] > Pivot ? No → 50 30 10 90 80 20 40 60 (Start →, End)

A[start] > Pivot ? Yes → 50 30 10 90 80 20 40 60 (Start, ← End) ← No A[End] ≤ Pivot ?

50 30 10 90 80 20 40 60 (Start, End) ← Yes A[End] ≤ Pivot ?

Swap(40, 90)

A[start] > Pivot ? No → 50 30 10 40 80 20 90 60 (Start →, End) ← No A[End] ≤ Pivot ?

A[start] > Pivot ? Yes → 50 30 10 40 80 20 90 60 (Start, ← End) ← Yes A[End] ≤ Pivot ?

50 30 10 40 80 20 90 60 (Start, End)

Swap(20, 80)

A[start] > Pivot ? No → 50 30 10 40 20 80 90 60 (Start, End)

A[start] > Pivot ? Yes → 50 30 10 40 20 80 90 60 (Start →, End)

50 30 10 40 20 80 90 60 (Start, End) ← No A[End] ≤ Pivot ?

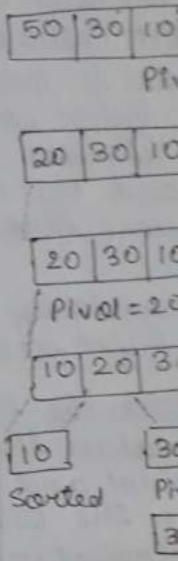
50 30 10 40 20 80 90 60 (Start, ← End) ← Yes A[End] ≤ Pivot ?

50 30 10 40 20 80 90 60 (End, Start) ← Yes A[End] > Start

20 30 10 40 50 80 90 60

B ≤ Pivot C

From the above example it is clear that 50 is placed in its proper position (4) where every element of part B is less than 50 and every element of part C is greater than 50. Now the 2 sub array



6 7
10 60

=60

End
40 60

End
40 60

End
0 40 60

← End
0 40 60 ← No A[End] ≤ Pivot ?

End
0 40 60 ← Yes A[End] ≤ Pivot ?

End
0 90 60

← End
20 90 60 ← No A[End] ≤ Pivot ?

End
20 90 60 ← Yes A[End] ≤ Pivot ?

End
80 90 60

End
80 90 60

tart
nd
80 90 60 ← No A[End] ≤ Pivot ?

art
nd
80 90 60 ← Yes A[End] ≤ Pivot ?

Start
80 90 60 ← Yes ~~A[End]~~ > Start

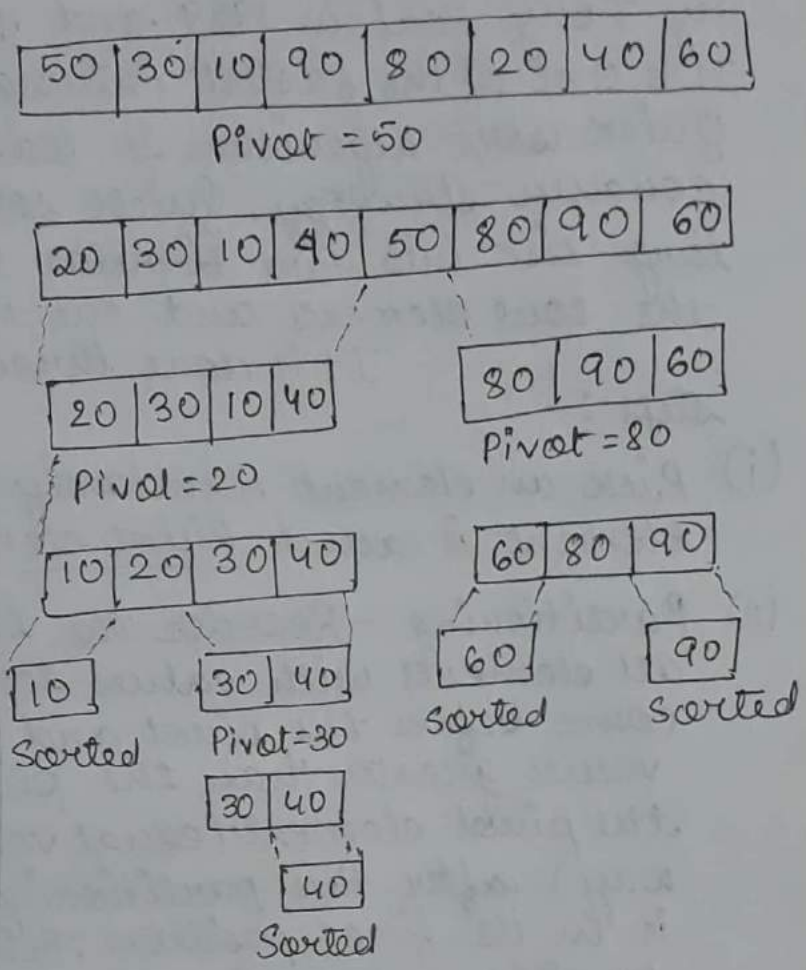
80 90 60)

C

0 is

y
element
array

B and C can be sorted by applying the same method



Quick sort is an efficient sorting algorithm which is used as a systematic method for placing the elements of a list in order. It was developed by Tony Hoar in 1959 and published in 1961. It is one of the fastest internal sorting technique. Quick sort algorithm is based on divided and conquer strategy. Quick sort first divides a large list into two ~~similar~~ smaller sub lists - the low elements and the high elements.

It is done through the following steps :-

(i) Pick an element randomly from the list. The element is called Pivot element.

(ii) Partitioning - Reorder the list ~~show~~ so that all elements with values less than the pivot come before the pivot and all elements with values greater than the pivot, come after the pivot element (equal values can go either way) after this partitioning the pivot element is in its final position. This is called the partition operation.

(iii) Recursively apply the above steps to the sub list of elements with smaller values and separately to the sub list of elements with greater values.

(III) Merge Sort :-

Merge Sort is a sorting algorithm that uses the divide, ~~compare~~ conquer and combine methods to sort a list of unsorted elements.

Divide :-

It means partitioning the n element array to be sorted into 2 sub arrays of n/2 elements in each sub array. If A is an array containing 0 or 1 element, then it is already sorted.

However, if there are more elements in the array, divide A into 2 sub arrays A_1 and A_2 - each containing about half of the elements of A.

Conquer :-

It means sorting the 2 sub arrays recursively using merge sort.

Combine :-

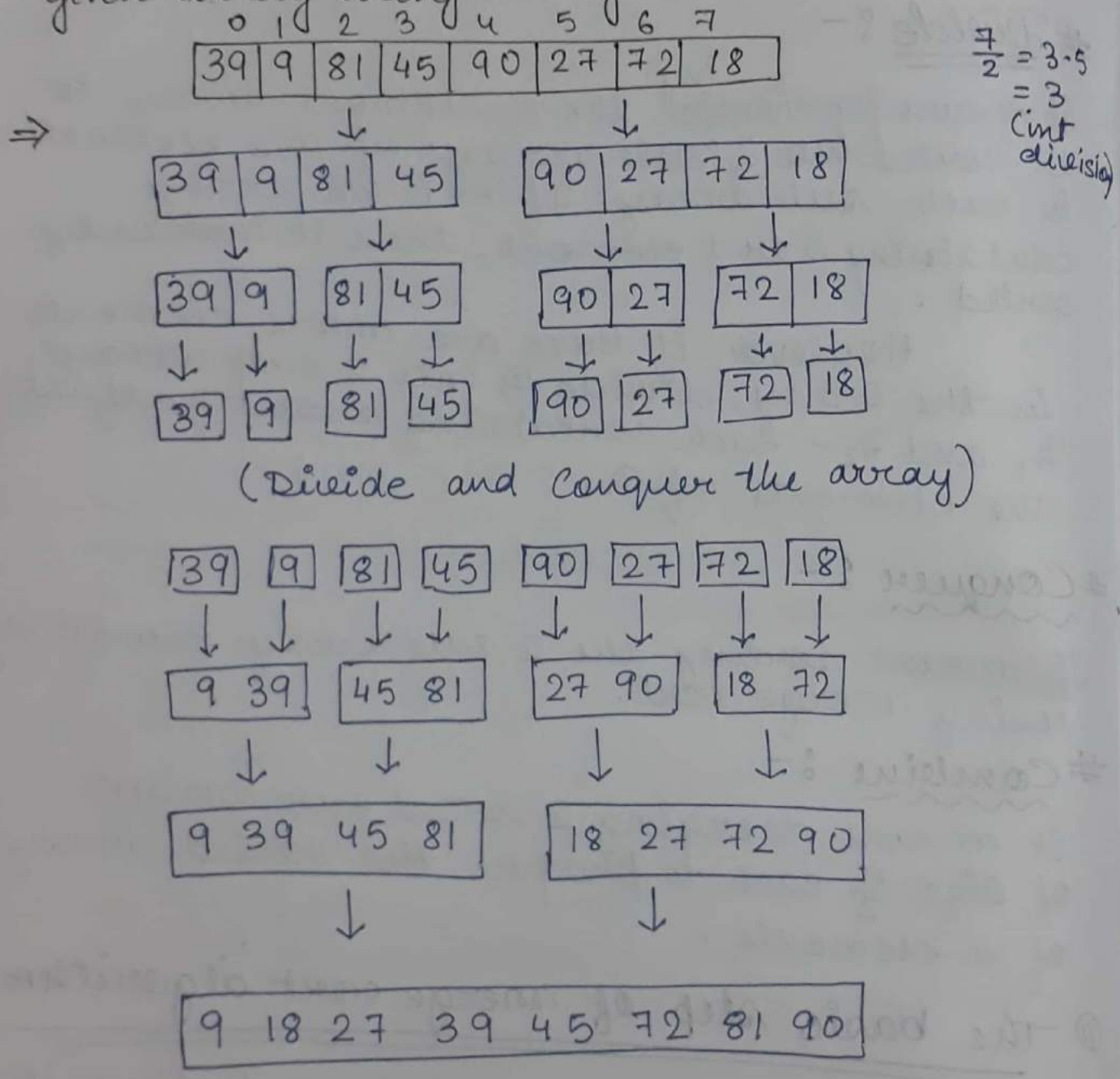
It means merging 2 sorted sub arrays of size $\frac{n}{2}$ each to produce the sorted array of n elements.

(IV) The basic step of merge sort algorithm :

1. If the array is of length 0 or 1 then it is already sorted.
2. Otherwise divide the unsorted array into 2 sub arrays ~~of~~ of about half the size.
3. Use merge sort algorithm recursively to sort each sub array.

4. Merge the 2 sub arrays to create a single sorted list.

- Consider the following example. Sort the given array using merge sort.



(Divide and Conquer the array)

(combine the elements to form a sorted array)

(II) Space Complexity :-

Space complexity is defined as the process of determining a formula for prediction of how much memory space is generally required to execute an algorithm. This space is generally the space of primary memory. The space needed by a program depends on the following 2 parts :-

(1) Fixed part → It varies from problem to problem. It includes the space needed for storing instructions, constants, variables and structure variable.

(2) Variable part → That varies from program to program. It includes the space needed for recursion stack and for structured variables that are allocated space dynamically during the run time of a program.

(III) Time Complexity :-

It is defined as the process of determining a formula for total time required to execute that algorithm. In the other words, no. of machine instructions which is executed during the program execution is called time complexity. This no. is primarily depended on the size of programs input and the algorithm used.

⑩ Types of Time complexity :-

Generally time complexity is of 3-types :-
(a) Worst case running time
(b) Average case running time
(c) Best case running time

⑪ Worst Case :-

The worst case running time of an algorithm is an upper bound on the running time or any input. That means the execution time of an algo will never go beyond this time limit.

⑫ Average Case :-

It is an estimate of the running time for an average input.

⑬ Best Case :-

The best case running time of an algo. is the minimum required time for any input under optimal condition.

⑭ Expressing Time And Space Complexity :-

This can be expressed using a function $f(n)$ where n is the input size for a given problem. It is essential when :-

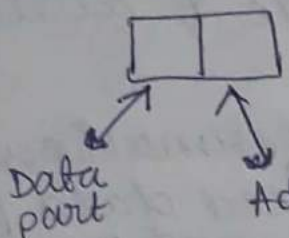
(1) we want to predict the rate of growth of complexity as the size of the problem increases.

(2) There are multiple algorithms that find a solution to a given problem and we need to find the algorithm that is most efficient.

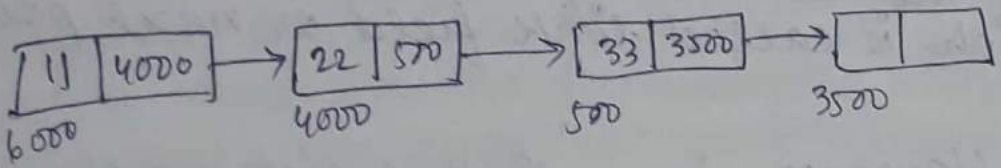
The most widely used notation to express this function $f(n)$ is the Big-Oh notation.

3/11/18 (Saturday)

LINKED LIST



→ nodes are created dynamically



→ CPU allocates memory as required dynamically.

Array	Linked List
(1) Static memory allocation	(1) Dynamic.
(2) Contiguous memory allocation.	(2) May not be contiguous.
(3) Less Efficient	(3) Efficient memory management.
(4) Searched using Index.	(4) Searched using address.
(5) Searching & sorting is ~~easier~~ less easier.	(5) Easier.
(6) Time complexity more.	(6) Time complexity less

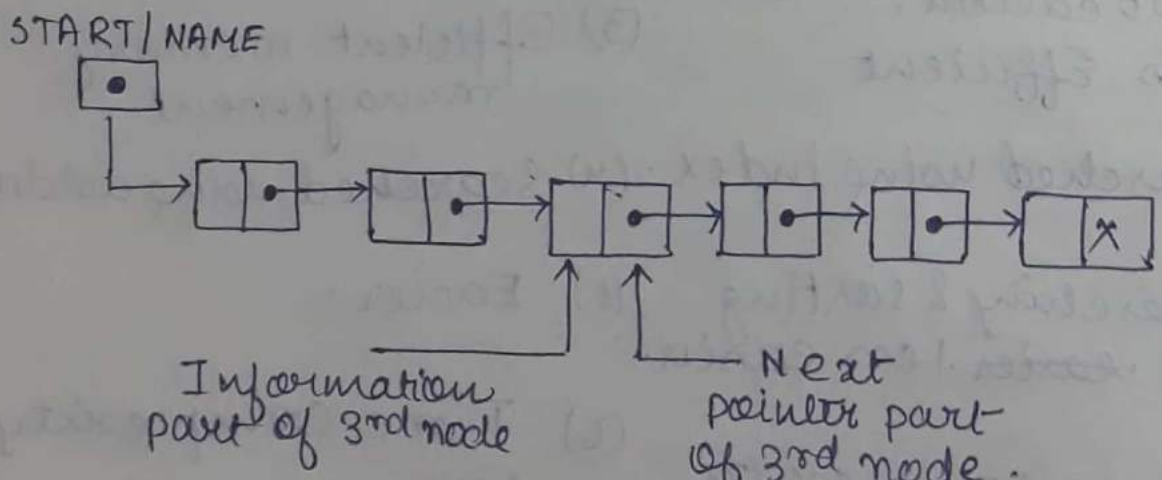
(III) Linked List :-

A linked list is a linear collection of data elements. These data elements are called nodes, where the linear order is given by means of pointers. That is each node is divided into 2 parts:-

The first part contains the information of the element, and the 2nd part called data part or information part and the 2nd part contains the address of the next node in the list which is called link field or next pointer field.

However, in a linked list nodes are logically interconnected or in ordered with the help of pointers in the memory but physically the nodes may not be sequential.

The systamatic diagram of a linked list with 6 nodes are as follows :-



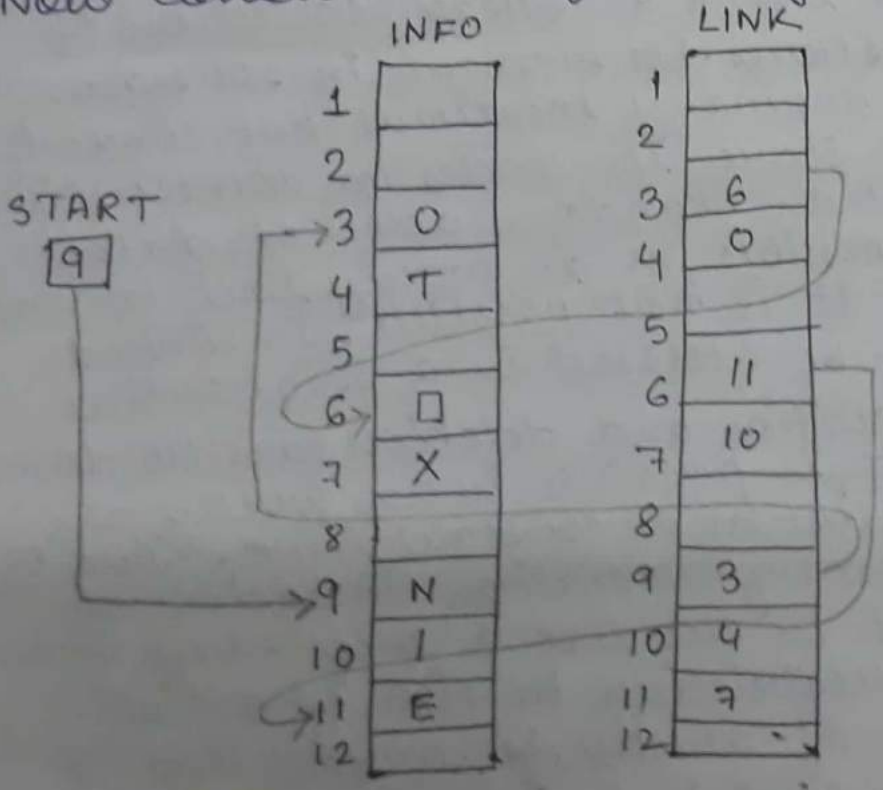
In this linked list the last node's next pointer field contains a special value which is called null pointer, which is any invalid address. In actual practice 0 or a negative

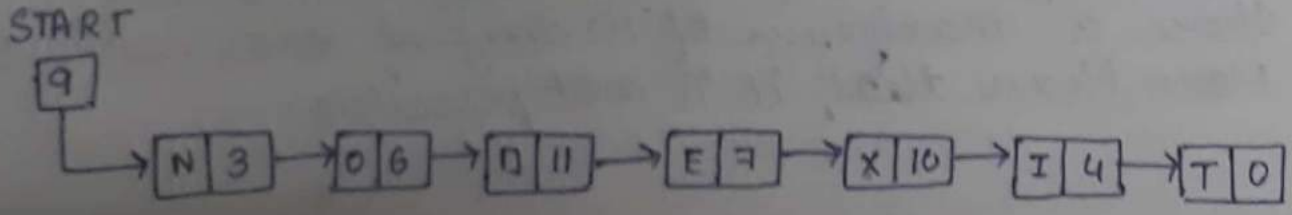
no. is used for a null pointer. The null pointer, denoted by X in the diagram, signals the end of the list.

The linked list also contains a list pointer variable which is called START/NAME - contains the address of the first node in the list. So we need only this address in START to trace through the list.

(11) Memory Representation of linked list :-

Suppose LIST is a linked list. In memory LIST is maintained by using 2 linear arrays - INFO and LINK such that INFO[K] and LINK[K]. Now consider the following eg :-





START = 9, So INFO[9] = N is the first element.
LINK[9] = 3, So INFO[3] = O is the second element
LINK[3] = 6, So INFO[6] = □ (blank) is the third element
LINK[6] = 11, So INFO[11] = E is the 4th element.
LINK[11] = 7, So INFO[7] = X is the 5th element
LINK[7] = 10, So INFO[10] = I is the 6th element
LINK[10] = 4, So INFO[4] = T is the 7th element

LINK[4] = 0, the NULL value, so the linked list has ended.

(iii) Linked list vs Array :-

Array is a linear collection of data elements and a linked list is a linear collection of nodes. (i) Array stores its element in consecutive memory locations but linked list does not store its nodes in consecutive memory location. (ii) Random access of data elements is possible in an array but in case of linked list it is not possible. (iii) In linked list nodes can be accessed in a sequential manner. Insertion and deletion can be done at any point is possible in an array as well as in a list in a constant time. Due to dynamic memory allocation insertion of new elements in the list is very easy because there is no restriction in size of a list. But in case of array, ~~its~~ array has its size. Suppose A[10] is an array and it can store a maximum of 10 ~~data~~ el data elements. More than that it is not possible.

(III) Types of Linked list :-

Depending on the number and nature of links used to connect each element in the list, linked list can be classified into 4 major groups :-

(i) Single linked list or singly linked list or one way linked way.

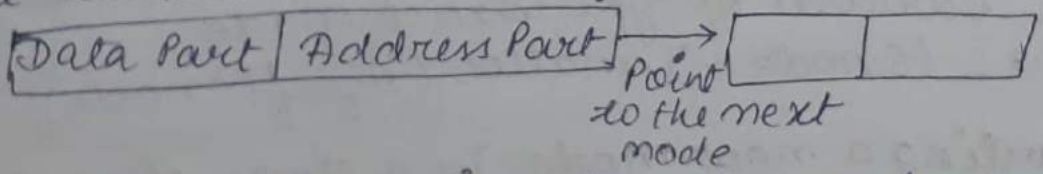
(II) Double linked list or doubly linked list or two way linked list.

(III) Single circular linked list (circular list using single list)

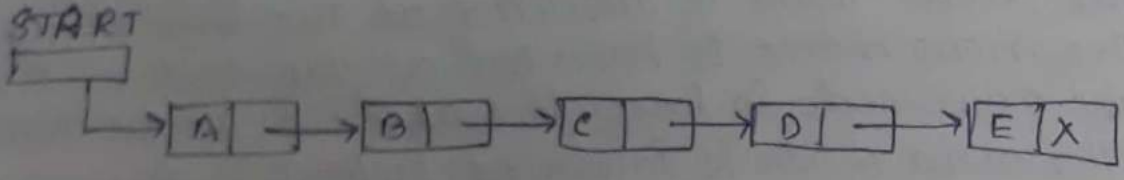
(IV) Double circular linked list (circular list using double list)

(I) Single linked list :-

The single linked list is the simpliest type of linked list in which every node contains some data and a pointer to the next node of the same data type. The basic structure of node in a single linked list is as follows :-



The data part contains the data element and the address part stores the address of the next node in sequence.



Operations on a single linked list :-

Some possible operation on a single linked list are :-

(i) Creation of a new node.
(ii) Deletion of a node (iii) Insertion of a node into linked list
(iv) Searching of a node into linked list
(v) Reverses the direction of all links of a single linked list.
(vi) Counts the no. of nodes in a linked list.
(vii) Concatenation of 2 linked list.
(viii) Print a linked list in reverse order.

(i) Creation of a new node :-

Node creation means allocating space of suitable size for the node in memory. In C the space allocation is done by malloc (sizeof datatype). This when called ~~new~~ reserves the space of size - size of (data type) and returns the starting address of the allocated space if the memory space is allocated. This address should be stored in a variable of type struct node.

```
nw = (snode *) malloc (sizeof(snode))
```

(ii) Inserting a new node in a single linked list :-

There are 5 cases about how insertion is done into a single linked list.

Case 1 : The new node is inserted at the beginning
Case 2 : The new node is inserted at the end.
Case 3 : The new node is inserted after a given node.
Case 4 : The new node is inserted before a given node.
Case 5 : The new node inserted in a sorted linked list.

Before we start with the algorithm to do the insertion in all these 5 cases, let us first discuss an important term called OVERFLOW. Overflow

is a condition that occurs when AVAIL = NULL or no free memory ~~cell~~ cell is present in the system.

The insertion algorithms will use a node in the AVAIL list. The steps are as follows:-

① Checking to see if space is available in the AVAIL list. If AVAIL = NULL the algorithm will print the message overflow.

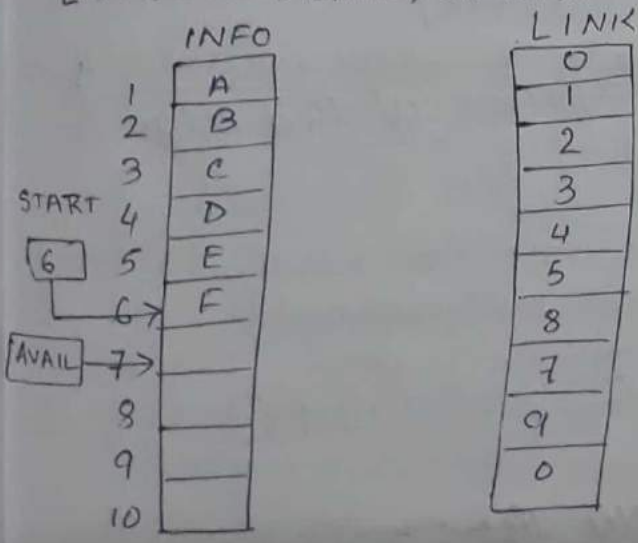
② Removing the first node from the avail list, using the variable NEW to keep track of the location of new node, this step can be implemented by the pair of assignments.

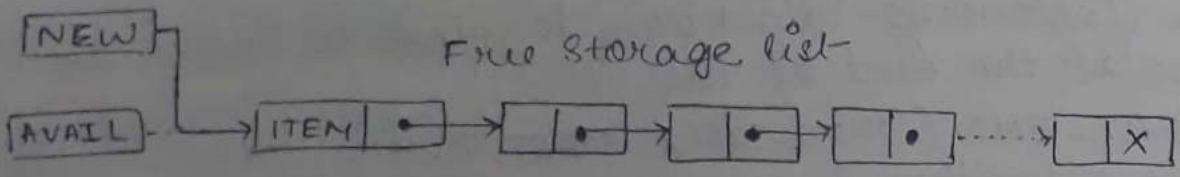
NEW = AVAIL , AVAIL = LINK [AVAIL]

③ Copy new information in to the NEW node.

INFO [NEW] = ITEM

Assume a linked list in the memory in the form LIST (INFO, LINK, START, AVAIL)





Free storage list

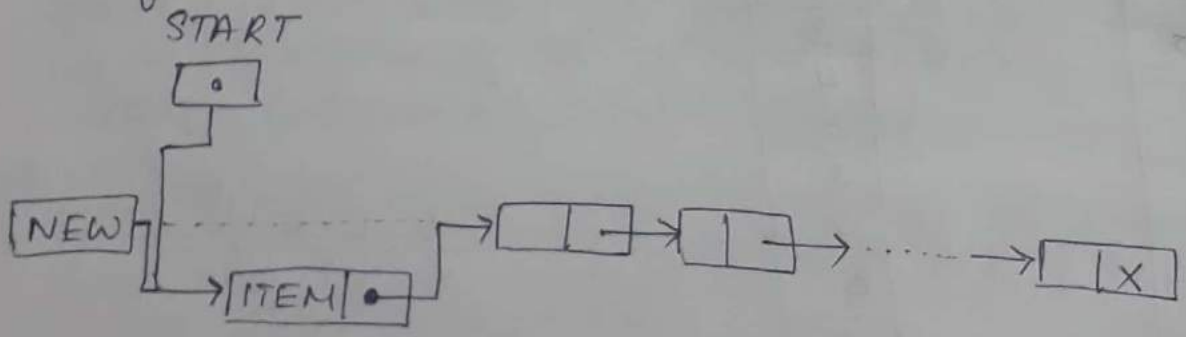
① Insertion at the beginning of a list :-

The following algorithm is used to insert an item at the beginning of a list.

INSERT_FIRST (INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM as the 1st node in the list.

① [OVERFLOW?] if AVAIL = NULL then
write : OVERFLOW and Exit
② [Remove first node from AVAIL list]
Set NEW = AVAIL and AVAIL = LINK[AVAIL]
③ Set INFO[NEW] = ITEM [copies the new data into the new node]
④ Set LINK[NEW] = START [new node now points to original first node]
⑤ Set START = NEW [Changes start so it points to the new node]
⑥ Exit

The systematic diagram of this algo is as follows :-



② INSERTION at the end of the list :-

The following algorithm is used to insert an item at the End of the list.

INSERT_END (INFO, LINK, START, AVAIL , ITEM)
This algorithm inserts ITEM at the last node in the list.

① [OVERFLOW ?] if AVAIL != NULL then
NEW = AVAIL and AVAIL = LINK[AVAIL]

② Set INFO[NEW] = ITEM

③ Set LINK[NEW] = NULL

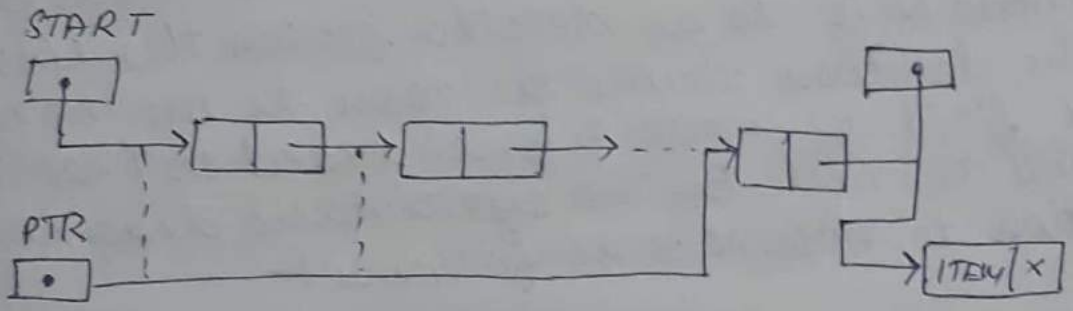
④ Set PTR = START

⑤ Repeat Step 7 while LINK[PTR] != NULL.

⑥ Set PTR = LINK[PTR]
[End of loop]

⑦ Set LINK[PTR] = NEW

⑧ Exit



③ Insertion after a given node :-

The following algorithm is used to insert an item into the list after a given node.

INSERT_AFTER (~~IFF~~ INFO, LINK, START, AVAIL, ITEM, NUM)
This algorithm insert item after that node whose value is NUM.

① [OVERFLOW ?] if AVAIL = NULL, then
Write : OVERFLOW and Exit.

② [Remove first node from AVAIL list]
Set NEW : AVAIL and AVAIL = LINK[AVAIL]

③ Set INFO[NEW] = ITEM

④ Set PTR = START

⑤ Set PREPTR = PTR

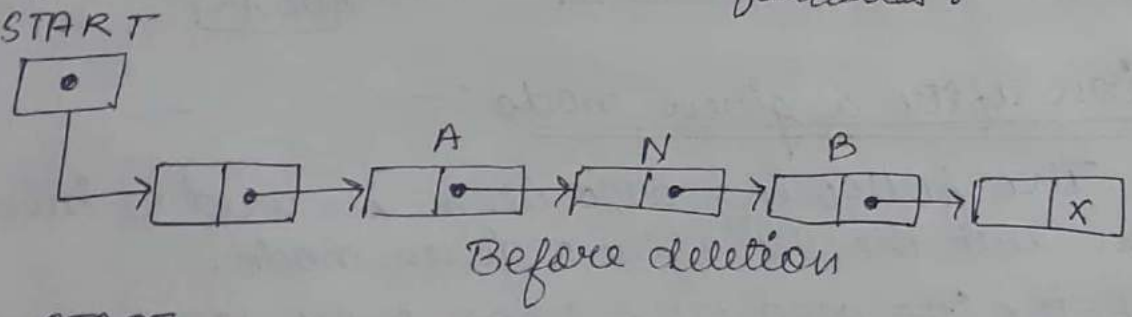
⑥ Repeat 7 and 8 while ~~PTF~~ PREPTR != NUM

⑦ Set PREPTR = PTR

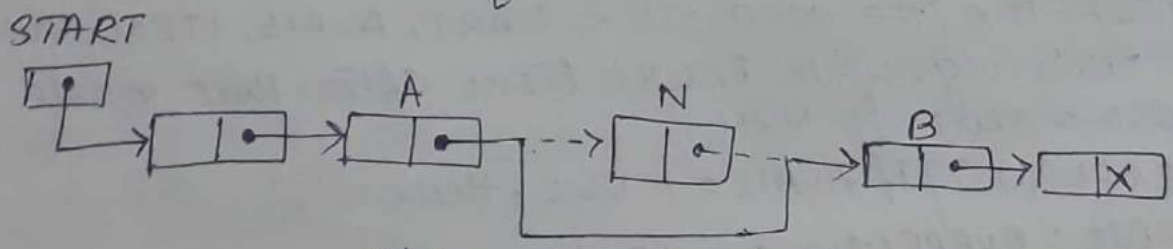
(8) Set PTR = LINK [PTR]
[End of Loop]
(9) LINK [PREPTR] = NEW
(10) Set LINK [NEW] = PTR
(11) Exit.

(III) Deletion from a single linked list :-

Through the deletion process a node can be removed from a single linked list. Suppose, LIST is a single linked list. Here N is a node lying between two nodes ; node A and node B. Now node N is to be deleted from the linked list. The deletion occurs as soon as the next pointer field of node A is changed so that it points to node B. The systamatic diagram of deletion to node N is as follows :-



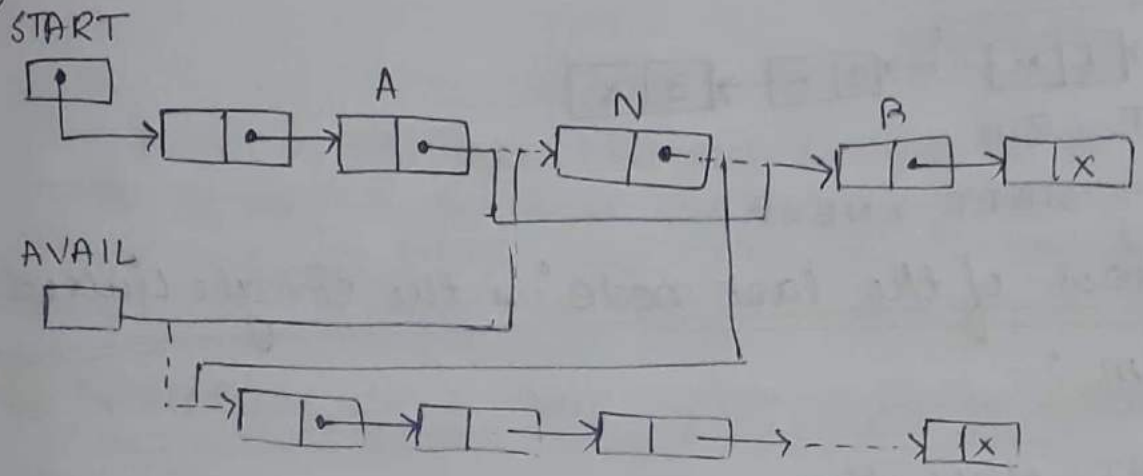
Before deletion



After deletion Node N

Suppose linked list is maintained in the memory in the form :
LIST (INFO, LINK, START, AVAIL)
When a node N is deleted from the list, its memory space is returned to AVAIL list.

specially, for easier processing, the free memory space will be returned to the beginning of the avail list. So the one more diagram is as follows :-



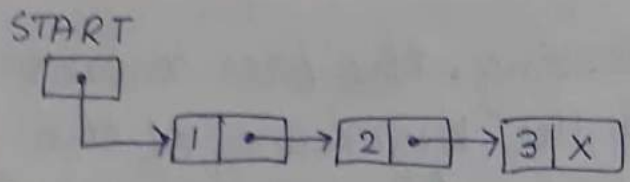
(II) Types of deletion from a single linked list :-

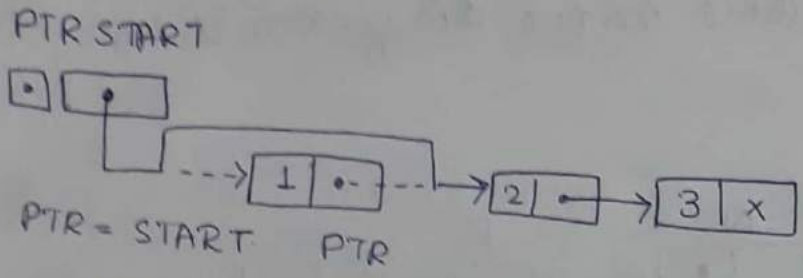
Case 1 : Deletion of 1st node in the single linked list.
Case 2 : Deletion of last node in the single linked list.
Case 3 : Deletion of a node after a given node.

(III) Deletion of the 1st node in the single linked list:

Algorithm :-

1. If START = NULL then
   write UNDERFLOW and exit
   [End of if]
2. Set PTR = START
3. Set START = START → NEXT
4. FREE PTR
5. Exit.



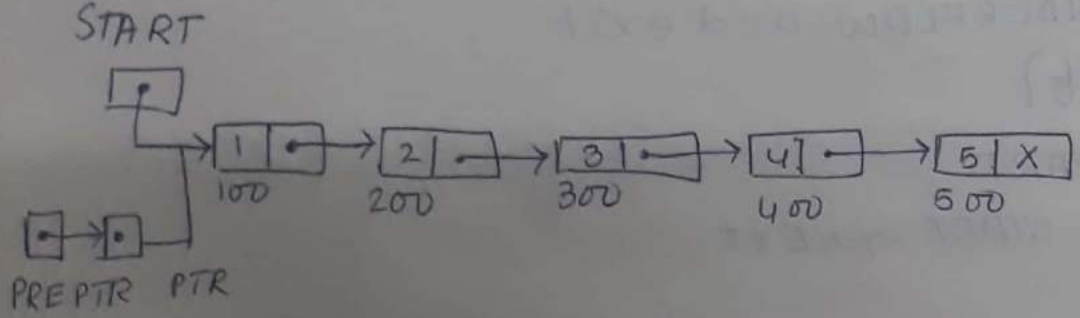


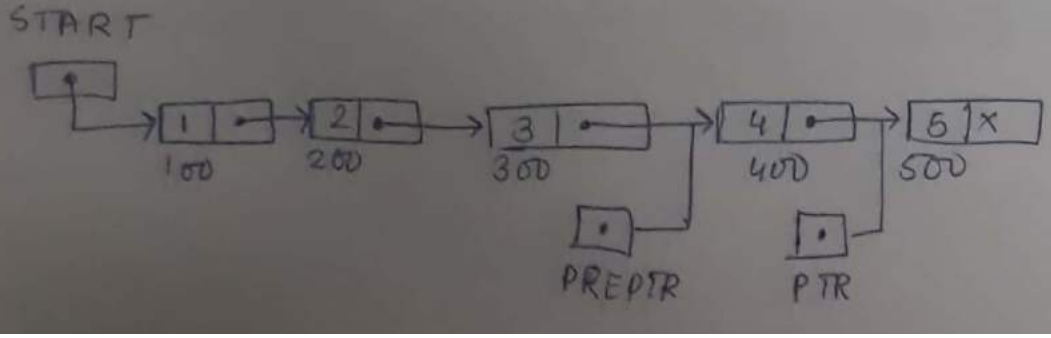
START = START → NEXT

(A1) Deletion of the last node in the single linked list:

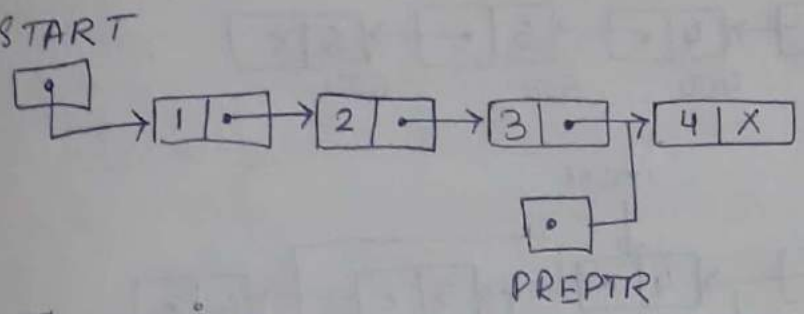
Algorithm :-

1. If START = NULL then
   Write UNDERFLOW and Exit
   [End of if structure]
2. Set PTR = START
3. Repeat Step 4 and 5 while PTR → NEXT != NULL
4. Set ~~PTR~~ PREPTR = PTR
5. Set PTR = PTR → NEXT
   [End of ~~if~~ loop structure]
6. PREPTR → NEXT = NULL
7. FREE PTR
8. Exit





~~The pointer variable~~



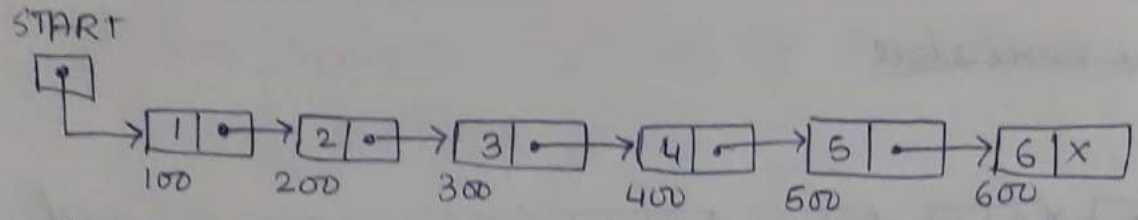
(This node is the 2nd last node of the list initially.

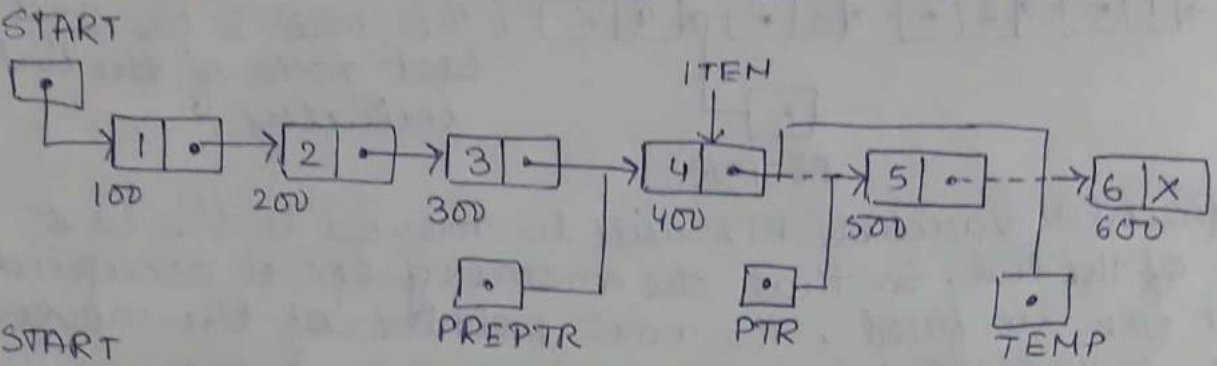
The pointer variable PTR will be moved to the last node of the list. So that the memory space occupied by it can be freed. The next pointer of the node which is indicated by PREPTR is assigned to NULL.

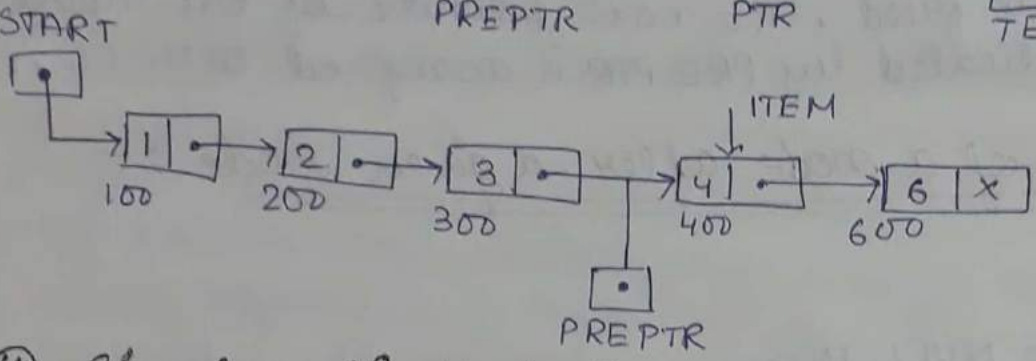
(III) Deletion of a node after a given node :-

Algorithm

1. If START = NULL then
   write UNDERFLOW and Exit
   [End of if Structure]
2. Set PTR = START
3. Set PREPTR = PTR
4. Repeat step 5 and 6 while PREPTR → DATA ! = ITEM
5. Set PREPTR = PTR
6. Set PTR = PTR → NEXT
   [End of loop structure]
7. Set TEMP = PTR → NEXT
8. PREPTR → NEXT = TEMP
9. FREE PTR
10. Exit

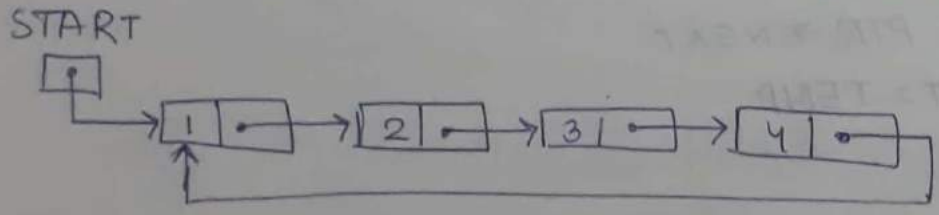






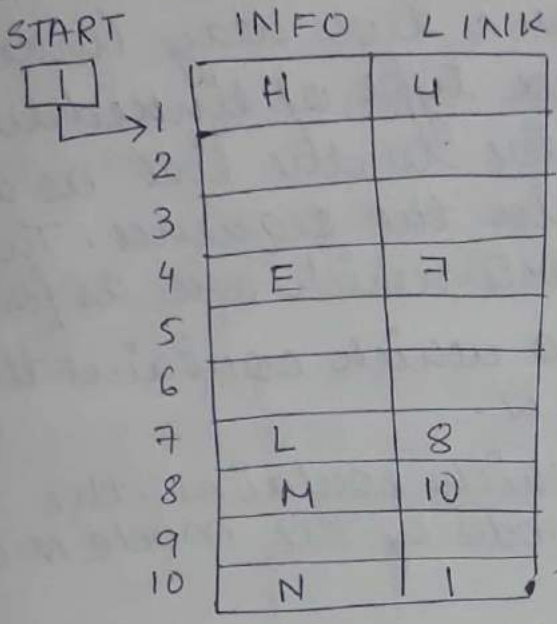
(III) Circular Linked List :-

Circularly linked list is a special type of linked list. In this linked list the ~~list of~~ link field of the last node points to the first node of the list. It is mainly used in the list that allow to access two nodes in the middle of the list without starting at the beginning. The systamatic diagram of a circularly linked list with 4 nodes are as follows :-



Notice that the last node, does not contain a null pointer. The link field of the last node is connected to the information part of the first node.

The memory representation of circular linked list is as follows:-



Insertion and deletion from a circular linked list follows the same pattern used in the single ~~list~~ linked list. However, in this case the last node to the first node. Therefore, when inserting or deleting the last node, besides updating the REAR pointer in the header, it is essential to point the link field to the first node.

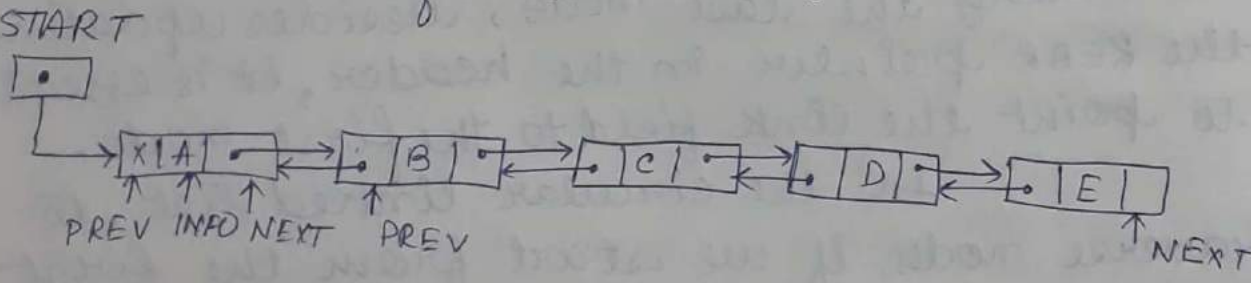
In the circular linked list, to traverse nodes, if we start from the first node to the last node then the traverse will be completed when the link field of a node contains the address of the first node because that node is the last node of the circular linked list.

(iii) Two way linked list or doubly linked list :-

A doubly linked list or a two way linked list is a more complex type of linked list which contains a pointer to the list as well as the previous node in the sequence. Therefore, it consists of three parts which are as follows:

1. An information field INFO which contains the data of a node suppose N.
2. A pointer field NEXT which contains the location of the next node of the node N of the list.
3. A pointer field PREV which contains the location of the preceding node of node N in the list.

The systematic diagram of a doubly linked list as follows:-



Observe that, the PREV field of each node is used to store the address of the preceding node as well as the NEXT field of each node is used to store the address of the next node.

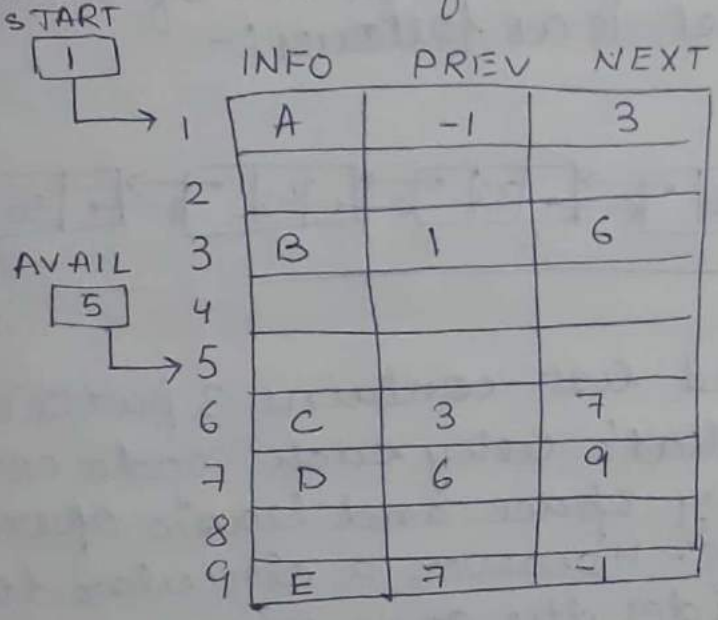
Remember that the ~~previous~~ PREV field of first node and the NEXT field of the last node will contain null value!

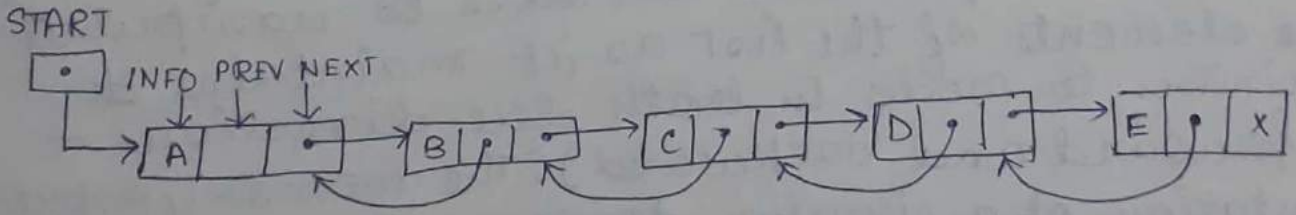
In doubly linked list each node assumes more memory space and basic operations are more expensive than

other linked list. However, doubly linked list provides the easy to manipulate the elements of the list as it maintains pointers to nodes in the both directions (forward and backward)

The main advantage of a doubly linked list is that it makes searching twice as efficient.

The memory representation of a doubly linked list is as follows :-



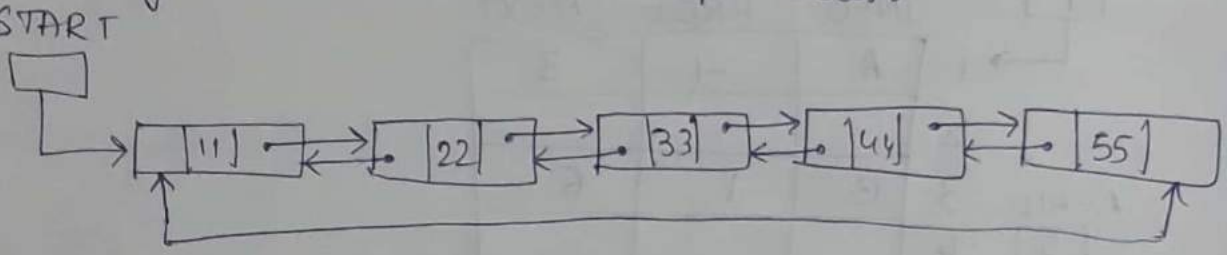


(II) Circular doubly linked list :-

A circular doubly linked list or a circular two way linked list is a more complex type of linked list which ~~contents~~ contains a pointer the next as well as the previous node in the sequence. The difference b/w a doubly linked list and a circular doubly linked list is same as that exist b/w a single linked list and a circular linked list.

The circular doubly linked list does not contain null in the previous field of the first node and the next field of the last node. In this case the next field of the last node stores the address of the first node of the list. Similarly, the previous field of the first node stores the address of the last node.

The systamatic diagram of a circular doubly linked list is as follows:-

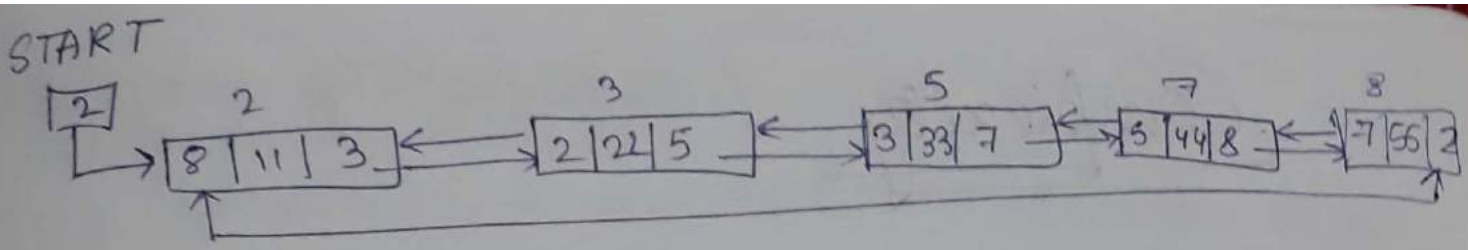


This type of linked list contains 3 parts in its nodes structure. That's why each node consumed more memory space and basic operations are more expensive. However a circular doubly linked list provides the ease to manipulate the elements of the list as it maintains ~~th~~ pointers to node in both the directions — (forward and backward). The memory representation of a circular doubly linked list is as follows:-

START [2]

AVAIL [6] →

	ITEM	PREV	NEXT
1			
2	11	8	3
3	22	2	5
4			
5	33	3	7
6			
7	44	5	8
8	55	7	2
9			



Complexity

(iii) Big-Oh notation (O):-

Suppose, M is an algorithm and suppose n is the size of input data. Now the complexity f(n) increases as n increases. Big-Oh examines the rate of increases of f(n). Big-Oh factor is expressed as O(n) where the O stands for the "order of" which concerned with what happens for very large values of n.

For eg :- If a sorting algorithm performs n^2 operations to sort just n elements then that algorithm will be described as an $O(n^2)$ algorithm. Suppose, f(n) and g(n) are the functions defined on a positive integer no. n then $f(n) = O(g(n))$.

That means, it is true if and only if positive constant c and n exist such that $f(n) \leq c\, g(n) \leq n$. It means that for large amounts of data f(n) will grow not more than a constant factor than g(n). Here g provides an upper bound and c is a constant which depends on various factors like -

1. The programming languaged used.
2. The quality of compiler or interpreter.
3. The CPU speed.
4. The size of main memory & the access time
5. The Knowledge of programmer.

(iii) Catagory of Algorithms :-

According to the Big-Oh notation there are five different categories of algorithm :

1. Constant time algorithm :- Running time complexity given as $O(1)$.
2. Linear time algorithm : Running time complexity given as $O(n)$.
3. Logarithmic time algorithm :- Running time complexity given as $O(\log n)$.
4. Polynomial time algorithm :- Running time given as $O(n^k)$ where $k > 1$
5. Expanential time algorithm :- Running time given as $O(2^n)$.

The Big-Oh notation is derived from function $f(n)$ using the following steps :-

1. Set the coefficient of each term to 1.
2. Rank each term starting from the lowest to the highest.
3. Then keep the largest term in the function and discard the rest.

Consider the following eg :-

Calculate the Big-Oh notation for the function $f(n) = \frac{n(n+1)}{2}$.

⇒ The function can be expanded as $\frac{1}{2}n^2 + \frac{1}{2}n$.

Step 1: Set the co-efficient of each term to 1. So now we have $n^2 + n$.

Step 2: Keep the largest term and discard the rest. So discard n and the Big-Oh notation can be given as - $O(f(n)) = O(n^2)$

(II) Complexity of Bubble Sort :-

The complexity of any sort algorithm depends upon no. of in bubble sort there are $(n-1)$ passes in total. In the first pass $(n-1)$ comparison to made to place the highest element in its correct position. Then in pass ~~to~~ 2 there are $(n-2)$ comparisons to place the 2nd highest element to its position and so on.

So, $f(n) = (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1$

$$= \frac{n(n-1)}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2} \quad [n^2]$$

So, Big-O $(f(n)) = O(n^2)$

Worst Case :- $\frac{n(n-1)}{2} = n^2$

Average Case :- $\frac{n(n-1)}{2} = n^2$

(III) Complexity of Insertion Sort :-

For insertion sort the best case occurs when the array is already sorted. In this case the time complexity is $O(n)$.

In Insertion sort the worst case occurs when the array is reversed order and the

inner loop must use the maximum no.(K-1) of comparisons.

So, $f(n) = 1+2+3+\ldots\ldots+(n-1)$

$= \frac{n(n-1)}{2}$

$= O(n^2)$

On the average case this comparison will be $\frac{K-1}{2}$ in the inner loop.

$\therefore f(n) = \frac{1}{2} + \frac{2}{2} + \frac{3}{2} + \ldots\ldots + \frac{n-1}{2}$

$= \frac{n(n-1)}{4}$

$= O(n^2)$

(iii) Complexity of Selection Sort :-

In case of selection sort algorithm. The comparison is independent of the original order of the linked list. In pass 1, selecting the smallest element $(n-1)$ comparisons is required. In pass 2, $(n-2)$ comparisons is essential to select the 2nd smallest element, so on :-

So, $f(n) = (n-1) + (n-2) + (n-3) + \ldots\ldots + 3+2+1$

$= \frac{n(n-1)}{2}$

$= O(n^2)$

Worst Case :- $O(n^2)$

Average Case :- $O(n^2)$

(II) Complexity of Merge Sort :-

In case of merge sort algorithm. The running time in the average case and the worst case can be given as $O(n \log(n))$ because the function $f(n)$ is recalled at most $\log n$ passes. Each pass merges a total of n elements. So call n elements ~~to~~ + the total comparisons are :-

$$f(n) < n \log(n)$$

(III) Complexity of Quick Sort :-

In the average case the running time of quick sort can be given as $O(n \log(n))$. In this case the comparison of function n is $1.4(n \log(n))$. In case of worst case the comparison of

$$f(n) = \frac{n(n+3)}{2}$$

So, Big O is $O(n^2)$

(II) Time complexity of binary search $O(\log(n))$

(IV) " " " linear search $O(n)$

(III) Hashing :-

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some example of how hashing is used in our lives includes -

(i) In universities, each student is ~~eg~~ assigned a unique roll no. that can be used to retrieve information about them.

(ii) In libes this is book is assigned a unique no. that can be used to determine information about the book.

In both these examples the students and books were hashed to a ~~int~~ unique no. Assume that we want to assign unique no. or key each object to made searching easy to store the key we can use a simple array like - data structure. Then at one time the key becomes large and then it cannot be used directly as an index. That time hashing can help to handle this situation. In hashing large keys are converted into small keys by using hash function. The values are then stored in a data structure called Hash table.

(iii) Hash Table :-

Hash table is a data structure in which keys are mapped to array positions by a hash function basically a hash table is near round in array ie; address by a ~~hash~~ hash function. Each element of the array is called a bucked. So a hash table ht is divided into n buckedes - ht[0], ht[1] .... ht[n-1] and each bucked has the capability to holding a record. For eg:- In a hash table an element with key K is stored in as index h[K] that means the hash function h the use to

calculate the index at which the element with key K will be stored. This process of mapping the key to appropriate location in a hash table is called hashing.

(iv) Hash Function :

A hash function can be defined as function that takes as a node and maps it into an index in hash table. A hash function can be defined as function

Recursion

(I) Recursive Function :-

A function that ~~calles~~ calls itself directly or indirectly and approaches towards the terminating condition or table condition is called recursive function and this techniane is called as recursion.

(II) Basic requirements for recursion :-

For implementing and ~~di~~ designing the good recursive program, the following ~~criterias~~ ~~shown~~ should be followed :-

(i) Base Case → Base case is the terminating condition for the problem. While designing any recursive algorithm. We must choose a proper terminating condition for a problem.

(ii) If condition :- In the ~~new~~ recursive algorithm if condition defines the terminating condition.

(iii) Everytime a new recursive called is made a new memory space which is allocated to each automatic variable used by the ~~recursive~~ recursive routine.

(iv) Each time recursive call is there the values of the local variables are pushed on to the stack. This values are poped when respective functions are called.

(v) Recursive case - Else part of the recursive definition calls the function recursively.

Ex: Consider the following eg. to ~~exam~~ explain the recursive function

```
#include <stdio.h>
int sum(int n);
void main()
{
    int number, result;
    printf("Enter a positive no");
    scanf("%d", &number);
    result = sum(number);
    printf("Sum = %d", result);
}
int sum(int x)
{
    if(x!=0)
    return (x + sum(x-1));
    else
    return(x);
}
```

Output: Enter a positive no : 3.
Sum = 6

(iii) ~~No~~ Trace of this program is as follows:

```
void main()
{
..... [3]
  result = sum(number);
}.....
int sum(int x)
{
   if(x!=0)
   return (x + sum(x-1));   [3] [2]
   else
   return x;
}
int sum(int x)
{
```

3+3 = 6 is returned

```
if (x!=0)
return (x + Sum(x-1));
else
return x;
}
int sum(int x)
{
  if (x!=0)
  return (x, sum(x-1));
  else
  return x;
}
int sum(int x)
{
  if (x!=0)
  return(x, sum(x-1));
  else
  return x;
}
```

2+1=3 is returned

0+1 is returned

0 is returned

(III) Types of recursion :-

Generally recursion technique is 3 types :-

1. Direct recursion
2. Indirect
3. Tail

(III) Direct recursion :-

A function is said to be directly recursive if it explicitly calls itself.

For eg :- Consider the following function :-

```
int function1(int n)
{
  if(n==0)
  return n;
  else
    return (function1(n-1));
}
```

Here, function1() calls itself for all positive values of n. It is said to be a directly recursive function.

(iii) **Indirect recursion** :-

A function is ~~set~~ said to be indirectly recursive if it contains a call to a recursive function which ultimately calls it.

Consider following eg :-

```
int function1(int n)
{
  if(n==0)
  return n;
  else
  return (function2(n));
}
int function2(int x)
{
  return (function1(x-1));
}
```

These 2 function function1 and function2 are indirectly recursive as they both call each other.

(iv) **Tail recursion** :-

A ~~tail~~ recursive function is ~~set~~ said to be tail recursive. If no operations are pending to ~~the~~ be performed when the recursive function return to its caller ie; when the called function returns, the return value is immidiately

return from the calling function.

Consider the following example :-

```
int fact(int n)
{
    return fact1(n,1);
}
int fact(int n, int result)
{
    if (n==1)
        return result;
    else
        return fact1(n-1, n*result);
}
```

Advantages :-

The advantages of using a recursive program are :-

1. Recursive solutions are comparitively shorter & simpler than non-recursive function
2. Code is clear and easier to use.
3. Recursion represents the original formula to solve a problem.
4. It follows a divide and conquer technique to solve problems.
5. In some cases recursion may be more efficient.

Disadvantages :-

The drawbacks of using a recursive function are :-

1. It consumes more storage place because the recursive function calls along with automatic variables and stores onto the stack.

(2) It is not more efficient in terms of speed & execution time.

(3) If proper precautions are not taken, recursion may result in non terminating iterations.

(4) It is difficult to find errors when using global variables.

(III) Recursion Vs Iteration :-

(1) Both iteration and recursion are based on a control structure:
Looping structure is used in iteration whereas recursion uses a selection or branching structure.

(2) Both recursion and iteration is repeatation:
In iteration, repeatation is made by loop structure. On the other hand in recursion repeatation is achieved through repeated function calls.

(3) Iteration and recursion each involve a checking of terminating condition:
Iteration terminates when the loop continuation condition fails. On the other hand recursion terminates when a base case is recognized.

(4) Both iteration and recursion can occur infinitely:
An infinite loop occurs in iteration if the testing of the loop continuation becomes always true. On the other hand

infinite sequence of function calls occurs if the recursion step doesn't involve any further.

(5) Recursion can be expensive in both processor type and memory space:

Recursive is slower than iterative method because overhead for loop repeatition in iteration is smaller than overhead for repeatidly function called and return in recursive.

(6) The recursive version is shorter and simpler than iterative solution:

Recursive code simpler than iterative because recursive code is easier to write, read and debug.