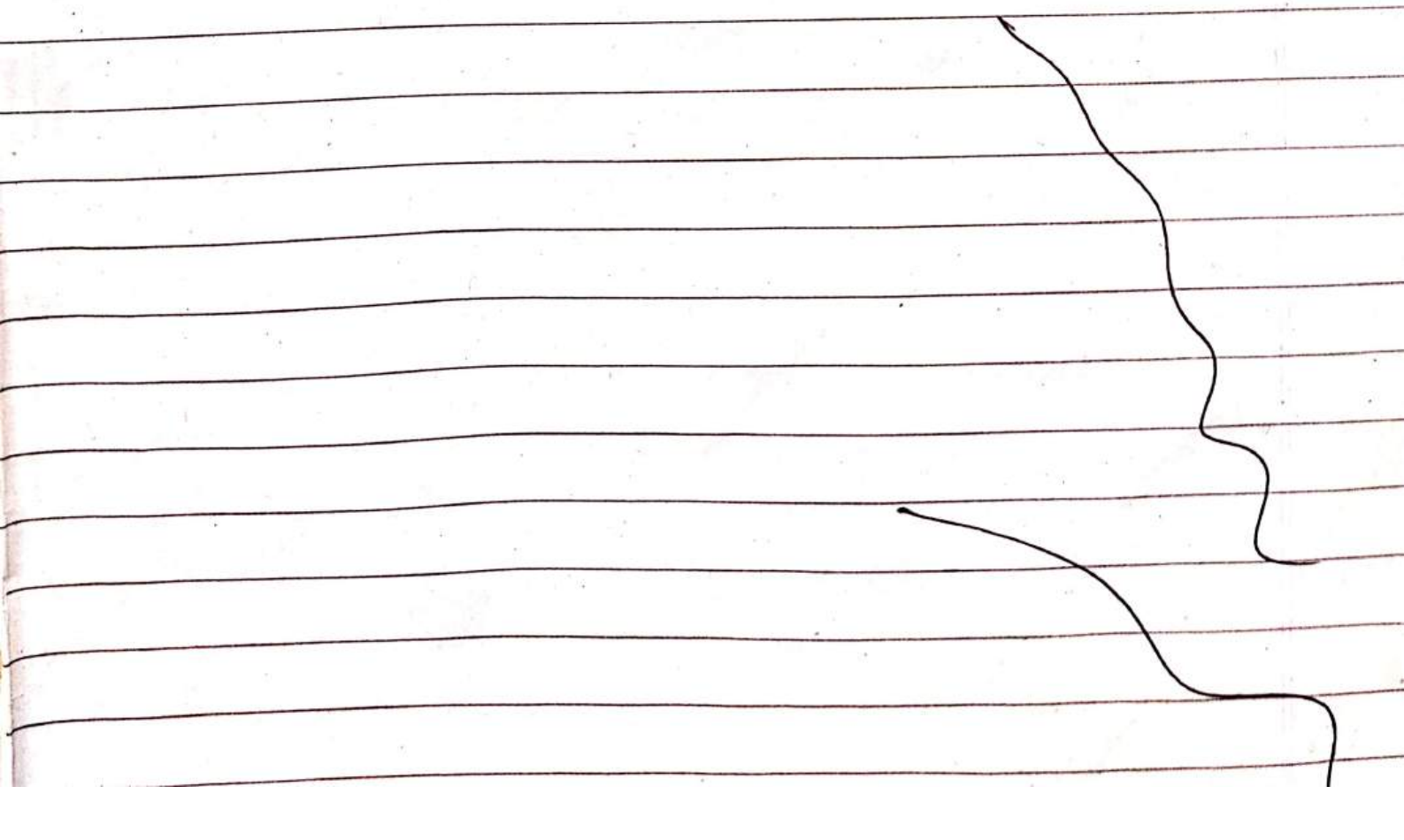
Date
Page

Santosh Bhandari

OOP in Java

Data Structures & Algorithms



Data Structures and Algorithms
CACS 201

Unit 1: Introduction to Data Structure
Unit 2: The Stack
Unit 3: Queue
Unit 4: List
Unit 5: Linked lists
Unit 6: Recursion
Unit 7: Trees
Unit 8: Sorting
Unit 9: Searching
Unit 10: Graphs
Unit 11: Algorithms

Unit :- 1 Introduction to Data Structure

1. Algorithm

An algorithm is the step by step description of a program. There is no any exact rules for developing an algorithm, However :-

i) It should be simple to understand and implement
ii) It Should produce desire output.
iii) It should have finite number of steps
iv) It should have definite starting and ending steps.

Example 1 :-

An algorithm to Input radius of a circle and outputs its area and circumference.

Step 1 : START
Step 2 : Input a Radius (r)
Step 3 : Calculate
area = πr^2, where $\pi = 3.14$
circumference = $2\pi r$
Step 4 : Output area and circumference
Step 5 : END

Example 2 :-

An algorithm to input a number and display if it is positive or negative.

Step 1 : START
Step 2 : Input a Number (n)

Step 3 : if $n > 0$, then display n is positive
Step 4 : if $n < 0$, then display n is negative
Step 5 : if $n = 0$, then display n is neither positive nor negative
Step 6 : END

Another Way

① START
② Input a Number (n)
③ If $n > 0$
then display n is positive
else if $n < 0$
then display n is negative
else
Display n is neither Positive nor Negative
④ END

Another Way

① START
② Input a Number (n)
③ if $n > 0$, then go to step 6
④ if $n < 0$, then go to step 7
⑤ if $n = 0$, then go to step 8
⑥ Display n is positive and END

⑦ Display n is Negative and END
⑧ Display n is Neither Positive Nor Negative and END

Example : 3
An algorithm to calculate sum of all natural Numbers up to 100.

① START
② Set i = 1, sum = 0
③ Repeat step 4 and 5 when i <= 100
④ sum = sum + i
⑤ i = i + 1
⑥ Output sum
⑦ END

2. Data Structures
A data structures is the logical collection of data. It defines the structures use for storing data, manipulating it and deleating the data.
Some of the common Data Structures used are:
i) Stack
ii) Queue
iii) List
iv) Linked list
v) Tree
vi) Graph

ADT (Abstract Data Type)

An abstract data type (ADT) consists of data type together with a set of operations, which define how the type may be manipulated. ADT exists conceptually and concentrate on the mathematical properties of the data type ignoring implementation constraints and details.

Advantages of ADT

- Modularity
- Precise Specifications
- Information Hiding
- Simplicity
- Integrity
- Implementation Independence

Stack as ADT

Stack can be defined as ADT:-

- Finite Sequence of Elements
- Operations on the Elements like:-
  - → CreateEmptyStack (s)
  - → Push (s, x)
  - → Pop (s)
  - → Top (s) / Peek (s)
  - → IsFull (s)
  - → IsEmpty (s)

Queue as ADT

Queue can be defined as ADT:-

- Finite sequence of elements
- Operations on the elements like:-
  - → MakeEmpty(q)
  - → IsEmpty(q)
  - → IsFull(q)
  - → Enqueue(q, x)
  - → Dequeue(q)
  - → Traverse(q)

Linked list as an ADT

Linked list can be defined as ADT:

- Finite sequence of Elements
- Operations on the elements like:
  - Create()
  - Insert(x)
  - Delete()
  - Traverse()
  - IsEmpty()
  - Find()
  - Count()
  - Free

Graph as an ADT

Graph can be defined as ADT:

- Finite Sequence of elements
- Operations on the elements like
  - → Graph()
  - → addVertex(vert)
  - → addEdge(fromVert, toVert)
  - → addEdge(fromVert, toVert, weight)
  - → getVertex(vertKey)
  - → getVertices()

Unit: 2 The Stack

1. Stack

A Stack is a linear collection of a data in which data are inserted and deleted from only one end called as top of Stack. It follows LIFO (Last In First Out) order i.e. that data inserted as the last will be the first one to be removed.

2. Some term Related to stack

- The process of inserting data in the stack is called as push.
- The process of deleting the data from the stack is called as pop.
- The process of identifying the data at the top of the stack is called as peek.
- The stack is called fullstack if all the location of the stack contains data.
- The stacks is called as emptystack if the stack doesn't contains any data.
- An attempt to push data in the fullstack result in overflow contains.
- An attempt to pop data from an empty stack results in underflow.

3. Application of Stack

- Recursion
- Arithmetic Calculations (Mathematical Calculations)
- Solving Graphs problems (traversing in Graph)
- Solving problem using backtracking techniques

4. Algorithm

Algorithm for Push Operation

(1) [Check if the stack is full or not]
    if TOS = Max - 1
        then print Overflow and END.

(2) [Increase the value of TOS by 1]
    TOS = TOS + 1

(3) [Insert the New data in TOS of the Stack]
    STACK [TOS] = newdata

(4) END

Algorithm for Pop Operation

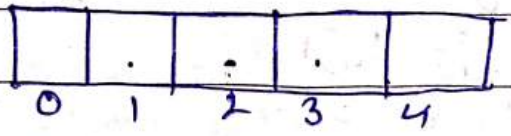
(1) [Check if the Stack is empty or not]
    if TOS = -1
        then print UNDERFLOW and EXIT.

(2) [Delete the data at the TOS]
    Delete STACK [TOS]

(3) [Decrease the value of TOS by 1]
    TOS = TOS - 1

(4) EXIT

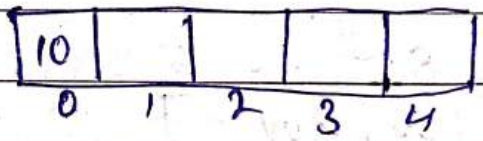
Example :-



TOS = -1

Max = 5

① PUSH data 10

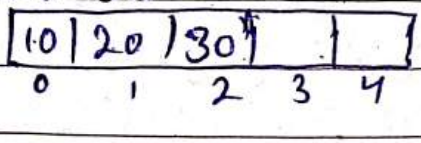


TOS = -1+1 = 0

② PUSH data 20

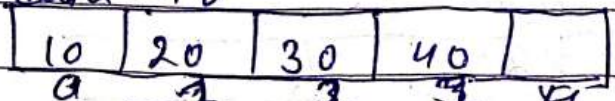
TOS = 0+1 = 1

③ PUSH data 30



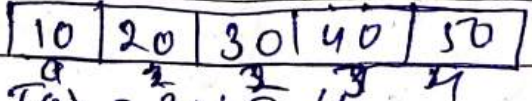
TOS = 1+1 = 2

④ PUSH data 40



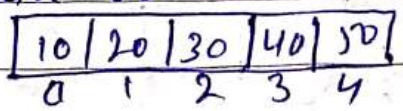
TOS = 2+1 = 3

⑤ PUSH data 50



TOS = 3+1 = 4

⑥ PUSH data 60

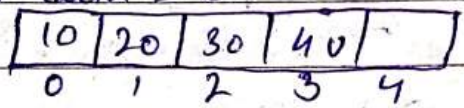


TOS = 4

OVERFLOW

⑦ POP data

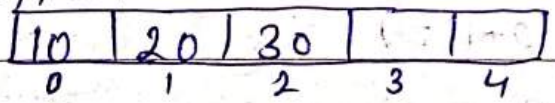
Popped data = 50



TOS = 4-1 = 3

⑧ Pop data

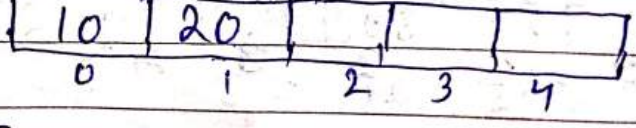
Popped data = 40



TOS = 3-1 = 2

⑨ POP Data

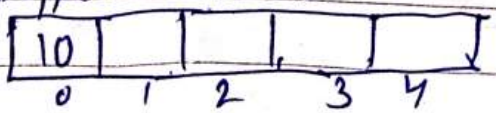
Popped data = 30



TOS = 2-1 = 1

⑩ Pop Data

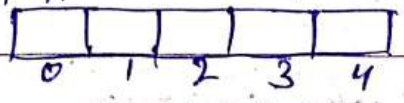
Popped data = 20



TOS = 1-1 = 0

(11) Pop Data

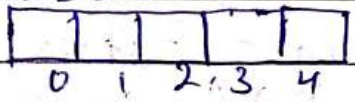
Popped data = 10



TOS = 0 - 1 = -1

(12) Pop Data

UNDERFLOW



TOS = -1

5. Mathematical Expression / Notation

A mathematical expression includes operands and the operator.

Types of Expression

i) Infix Expression
ii) Postfix Expression
iii) Prefix Expression

In infix expression, the operator is placed in between the operands. It is the most common mathematical expression used.
Ex:- a+b

In postfix expression, the operator is placed after the operands.
Ex:- ab+

In prefix expression, the operator is placed before the operands.
Ex:- +ab

6. Algorithm to evaluate a Postfix expression
The following algorithm calculates the result of a mathematical expressi
(P written in Postfix notation)

① START
② Add a right parenthesis ")" at the end of P.
③ Scan all the elements of P from left to right individually
and repeat steps 4 and 5 until ")" is encountered.
④ If an operand is encountered, PUSH it to Stack.
⑤ If an operator (X) is encountered, then
a) pop the top two elements of Stack, where A is the top element and B is the next to top element.
b) Evaluate B (X) A
c) PUSH the result back to Stack.
⑥ Set the result as the element at the top of the Stack.
⑦ EXIT

Example:-
① P = 5,6,2,+,*,12,4,/,-,)

Scanned Element	STACK
5	5
6	5, 6
2	5, 6, 2

+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
−	37
)	

A = 2, B = 6
6 + 2 = 8
(B ⊗ A)
(B − A)

∴ Result = 37

(b) P: 12, 7, 3, −, /, 2, 1, 5, +, *, +

(c) P: 3, 1, +, 2, ^, 7, 4, −, 2, *, +, 5, −

(d) P: A B C D * − E / + F + G H / −, where
A=4, B=8, C=2, D=5, E=6, F=9, G=1, H=3

[Exponent Symbol :- ^, $, ↑]

(b) P: 12, 7, 3, −, /, 2, 1, 5, +, *, +,)

Scanned Element	STACK
12	12
7	12, 7
3	12, 7, 3
−	12, 4
/	3
2	3, 2
1	3, 2, 1

5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15
)	

∴ Result = 15

(c) P: 3, 1, +, 2, ^, 7, 4, -, 2, *, +, 5, -,)

Scanned Element	STACK
3	3
1	3, 1
+	4
2	4, 2
^	16
7	16, 7
4	16, 7, 4
-	16, 3
2	16, 3, 2
*	16, 6
+	22
5	22, 5
-	17
)	

∴ Result = 17

Date
Page

(d) P: ABCD * − E / + F − G H / −, where A = 4, B = 8, C = 2, D = 5, E = 6, F = 9, G = 1, H = 3

Now,

P: 4, 8, 2, 5, *, −, 6, /, +, 9, +, 1, 3, /, −,)

Scanned Elements	STACK
4	4
8	4, 8
2	4, 8, 2
5	4, 8, 2, 5
*	4, 8, 10
−	4, −2
6	4, −2, 6
/	4, −1/3
+	11/3
9	11/3, 9
+	38/3
1	38/3, 1
3	38/3, 1, 3
/	38/3, 1/3
−	37/3
)	

∴ Result :- 37/3

presendend order [i) Exponent ($\wedge$, \$, $\uparrow$)
ii) Multiplication (*) and Division (/)
iii) Addition (+) and Subtraction (−)]

Date
Page

7. Algorithm to convert infix expression into postfix
The following algorithm converts the infix(Q)expression into postfix (P):

① START
② Push left parenthesis "(" onto stack and add right parenthesis ")" at the end of Q.
③ Scan each element of Q from left to right and repeat steps 4 to 7 until the stack is empty.
④ If an operand is encountered, then add it to P.
⑤ If a left parenthesis is encountered, then push it to Stack.
⑥ If an operator (X) is encountered, then
  a) pop all the operators at the TOP of the stack which has higher or same precedence than (X) and add into P.
  b) Push the Operator (X) onto stack.
⑦ If a right parenthesis ")" is encountered, then
  a) Repeatedly pop all the operators at the top of the stack until left parenthesis "(" is encountered and add in P.
  b) Remove the left parenthesis "(" from stack.
⑧ EXIT

Example:-
i) Q: A+(N*C−(D/E^F)*G)*H)

Scanned elements	STACK	Postfix (P)
	C	
A	(	A
+	C, +	A
(	C, +, C	A
N	C, + , C	A, N
*	C, + , C, *	A, N,
C	C, + , C, *	A, N, C
−	C + C −	A, N, C, *
(	C + C − C	ANC *
D	C + C − C	ANC * D
/	C + C − C /	ANC * D
E	C + C − C /	ANC * DE
^	C + C − C / ^	ANC * DE
F	C + C − C / ^	ANC * DEF
)	C + C −	ANC * DEF ^ /
*	C + C − *	ANC * DEF ^ /
G	C + C − *	ANC * DEF ^ / G
)	C +	ANC * DEF ^ / G * −
*	C + *	ANC * DEF ^ / G * −
H	C + *	ANC * DEF ^ / G * − H
)	−	ANC * DEF ^ / G * − H * +

∴ Result :- ANC * DEF ^ / G * − H * +

Q ii) $P+Q-(R*S/T+U)-V*W$

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

iv) $(A+B)/D\$(E-F)*G$

ii) $P+Q-(R*S/T+U)-V*W)$

Scanned Elements	STACK	Postfix
	(	
P	(	P
+	(+	P
Q	(+	PQ
−	(−	PQ+
(	(−(	PQ+
R	(−(	PQ+R
*	(−(*	PQ+R
S	(−(*	PQ+RS
/	(−(/	PQ+RS*
T	(−(/	PQ+RS*T
+	(−(+	PQ+RS*T/
U	(−(+	PQ+RS*T/U
)	(−	PQ+RS*T/U+
−	(−	PQ+RS*T/U+−
V	(−	PQ+RS*T/U+−V
*	(−*	PQ+RS*T/U+−V
W	(−*	PQ+RS*T/U+−VW
)	−	PQ+RS*T/U+−VW*−

∴ Results :- $PQ+RS*T/U+-VW*-$

iv) (A+B)/D$(E-F)*G)

Scanned Elements	STACK	POSTFIX
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
/	(/	AB+
D	(/	AB+D
$	(/$	AB+D
(	(/$(	AB+D
E	(/$(	AB+DE
-	(/$(-	AB+DE
F	(/$(-	AB+DEF
)	(/$	AB+DEF-
*	(*	AB+DEF-$/
G	(*	AB+DEF-$/G
)		AB+DEF-$/G*

∴ Result :- AB+DEF-$/G*

iii) A*B/C + (D+E-(F*(G/H)))

Scanned Elements	STACK	Post Fix
(	(	
A	(	A
*	(*	A
B	(*	AB
/	(/	AB*
C	(/	AB*C
+	(+	AB*C/
(	(+(	AB*C/
D	(+(	AB*C/D
+	(+(+	AB*C/D
E	(+(+	AB*C/DE
-	(+(-	AB*C/DE+
(	(+(-(	AB*C/DE+
F	(+(-(	AB*C/DE+
*	(+(-(*	AB*C/DE+
(	(+(-(*(	AB*C/DE+
G	(+(-(*(	AB*C/DE+G
/	(+(-(*/	AB*C/DE+G
H	(+(-(*/	AB*C/DE+GH
)	(+(-(*	AB*C/DE+GH/
)	(+(-	AB*C/DE+GH/*
)	(+	AB*C/DE+GH/*-
)	-	AB*C/DE+GH/*-+

∴ Result :- AB*C/DE+GH/*-+

8. Mathematically convert infix to postfix.

i) a+b
= ab+

ii) A*B-C
= AB*-C
= AB*C-

iii) A*B+C/D
= AB*+C/D
= AB*+CD/
= AB*CD/+

iv) A*C+(D-E/F)-G
= A*C+(D-EF/)-G
= A*C+DEF/--G
= AC*+DEF/--G
= AC*DEF/-+-G
= AC*DEF/-+G-

v) (A+B)/D$(E-F)*G
= AB+/D$EF-*G
= AB+/DEF-$*G
= AB+DEF-$/*G
= AB+DEF-$/G*

vi) A*B/C+(D+E-(F*(G/H)))
= A*B/C+(D+E-(F*GH/))
= A*B/C+(D+E-FGH/*)
= A*B/C+(DE+-FGH/*)
= A*B/C+DE+FGH/*-
= AB*/C+DE+FGH/*-
= AB*C/+DE+FGH/*-+

For the given infix expression

Q: (A+B-C)*D+E-(F/G), where

A=1, B=4, C=2, D=8, E=3, F=9, G=3. Convert the expression into postfix and evaluate.

Converting into postfix:-

Q: (A+B-C)*D+E-(F/G),

Scanned Elements	STACK	Postfix
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
-	((-	AB+
C	((-	AB+C
)	(	AB+C-
*	(*	AB+C-
D	(*	AB+C-D
+	(+	AB+C-D*
E	(+	AB+C-D*E
-	(-	AB+C-D*E+
(	(-(	AB+C-D*E+
F	(-(	AB+C-D*E+F
/	(-(/	AB+C-D*E+F
G	(-(/	AB+C-D*E+FG
)	(-	AB+C-D*E+FG/
)	-	AB+C-D*E+FG/-

evaluating the equation

P: AB+C-D*E+FG/-)

P: 1,4,+,2,-,8,*,3,+,9,3,/,-,)

Scanned Elements	STACK
1	1
4	1,4
+	5
2	5,2
-	3
8	3,8
*	24
3	24,3
+	27
9	27,9
3	27,9,3
/	27,3
-	24
)	24

∴ Result = 24

9. Algorithm to evaluate a prefix expression

The following algorithm calculates the result of a prefix expression:

① START
② Add left parenthesis "(" at the start (front) of P.
③ Scanned all the elements of P from right to left and repeat steps 4 and 5 until "(" is encountered.
④ If an operand is encountered, PUSH it onto Stack.
⑤ If an Operator (X) is encountered, then
   a) Pop the top two element of stack where A is the top element and B is the next to top element.
   b) Evaluate A (X) B.
   c) PUSH the result of (b) onto the Stack.
⑥ Set the result equal to the element at the top of stack.
⑦ EXIT.

Example:

i) Evaluate -+AB*C-+DE/FG, where

A=1, B=4, C=2, D=8, E=3, F=9, G=3

Here:

P: (, -, +, 1, 4, *, 2, -, +, 8, 3, /, 9, 3

Scanned element	STACK
3	3
9	3, 9
/	3
3	3, 3
8	3, 3, 8,
+	3, 11

-	8
2	8, 2
*	16
4	16, 4
1	16, 4, 1
+	16, 5
÷	-11
(	

∴ Result :- -11

2i) P:- +A-/B$CD*E+F-/GHI, where
A=1, B=2, C=3, D=2, E=1, F=5, G=9, H=3, I=2

Here:-

P: (, +, 1, -, /, 2, $, 3, 2, *, 1, +, 5, -, /, 9, 3, 2

Scanned Elements	STACK
2	2
3	2, 3
9	2, 3, 9
/	2, 3
-	1
5	1, 5
+	6
1	6, 1
*	6
2	6, 2
3	6, 2, 3

-43/9

9	6, 9
2	6, 9, 2
/	6, 9/2
−	−9/2
1	−9/2, 1
+	−43/9
(	

∴ Result :- −43/9

10. Algorithm to convert infix expression to prefix

The following algorithm converts the mathematical expression written in infix notation (Q) into prefix (P).

1. START
2. Push the right parenthesis ")" onto stack and add left parenthesis "(" at the front of Q.
3. Scan each element Q from right to left and repeat steps 4 to 7 until the stack is empty.
4. If an operand is encountered, add it to P.
5. If a right parenthesis ")" is encountered, push it onto STACK.
6. If an operator (X) is encountered, then
   a) pop the operator on the top of the stack which has higher precedence than (X) and add it to P.
   b) PUSH the operator (X) to STACK.
7. If a left parenthesis "(" is encountered, then
   a) Repeatedly pop the operator on the top of STACK and add it to P until right parenthesis ")" is encountered.

b) Remove the right parantheses ")" from stack.

⑧ Reverse the P.

⑨ EXIT

Example :-

:) Q : ((A+B*C$D)/(E+F-G*H)$I/J

Scanned Elements	STACK	Prefix (P)
	)	
J	)	J
/	)/	J
I	)/	JI
$	)/$	JI
)	)/$)	JI
H	)/$)	JIH
*	)/$)*	JIH
G	)/$)*	JIHG
−	)/$)−	JIHG*
F	)/$)−	JIHG*F
+	)/$)−+	JIHG*F
E	)/$)−+	JIHG*FE
(	)/$	JIHG*FE+−
/	)//	JIHG*FE+−$
I	)//)	JIHG*FE+−$
D	)//)	JIHG*FE+−$D
$	)//)$	JIHG*FE+−$D

C	)))$	JIHG*FE+-$DC
*	)))*	JIHG*FE+-DC
B	)))*	JIHG*FE+-DCB
+	)))+	JIHG*FE+-DCB*
A	)))+	JIHG*FE+-DCB*A
(	))	JIHG*FE+-DCB*A+
[illegible]	-	JIHG*FE+-DCB*A+//
[illegible]		

∴ prefix Expression :- //+A*BCD-+EF*GHIJ

ii) Q: ((A-(B+C))*D)$(E+F)

iii) Convert the infix expression A+B-C*(D+E-F/G) into prefix and evaluate using A=1, B=4, C=2, D=8, E=3, F=9, G=3

Mathematically convert infix to prefix

a) A+B
= +AB

b) A*B-C
= *AB-C
= -*ABC

c) A*C+(D-E/F)-G
= A*C+(D-/EF)-G
= A*C+-D/EF-G
= *AC+-D/EF-G
= +*AC-D/EF-G
= -+*AC-D/EFG

d) A+B-C*(D+E-F/G)
= A+B-C*(D+E-/FG)
= A+B-C*(+DE-/FG)
= A+B-C*-+DE/FG
= A+B-*C-+DE/FG
= +AB-*C-+DE/FG
= -+AB*C-+DE/FG

e) $((A-(B+C))*D)\$(E+F)$

$= ((A-+BC)*D)\$+EF$

$= (-A+BC*D)\$+EF$

$= *-A+BCD\$+EF$

$= \$*-A+BCD+EF$

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

$= (A+B*\$CD)/(E+F-*GH)\$I/J$

$= (A+*B\$CD)/(+EF-*GH)\$I/J$

$= +A*B\$CD/-+EF*GH\$I/J$

$= +A*B\$CD/\$-+EF*GHI/J$

$= /+A*B\$CD\$-+EF*GHI/J$

$= //+A*B\$CD\$-+EF*GHIJ$

i) Q: $(((A-(B+C))*D)\$(E+F)$

Scanned Element	STACK	prefix
	)	
)	))	
F	))	F
+	))+	F
E	))+	FE
(	)	FE+
$	)$	FE+
)	)$)	FE+
D	)$)	FE+D

*	)$)*	FE+D
)	)$)*)	FE+D
)	)$)*))	FE+D
C	)$)*))	FE+DC
+	)$)*))+	FE+DC
B	)$)*))+	FE+DCB
(	)$)*)	FE+DCB+
-	)$)+)-	FE+DCB+
A	)$)*)-	FE+DCB+A
(	)$)*	FE+DCB+A-
(	)$	FE+DCB+A-*
(	-	FE+DCB+A-*$

∴ Result :- $*-A+BCD+EF

iii) Convert the infix expression A+B-C*(D+E-F/G) into prefix and evaluate using A=1, B=4, C=2, D=8, E=3, F=9, G=3.

Q: (A+B-C*(D+E-F/G)

Scanned Elements	STACK	Prefix (P)
	)	
)	))	
G	))	G
/	))/	G
F	))/	GF
-	))-	GF/
E	E))-	GF/E

+	)) −+	GF/E
D	)) −+	GF/ED
(	)	GF/ED+−
*	)*	GF/ED+−
C	)*	GF/ED+−C
−	)−	GF/ED+−C*
B	)−	GF/ED+−C*B
+	)−+	GF/ED+−C*B
A	)−+	GF/ED+−C*BA
(	−	GF/ED+−C*BA+−

∴ Result :- −+AB*C−+DE/FG

Evaluating

P: (−, +, 1, 4, *, 2, −, +, 8, B, /, 9, 3

Scanned Elements	STACK	
3	3,	
9	3, 9	
/	3	
3	3, 3	
8	3, 3, 8	
+	3, 11	
−	8	
2	8, 2	
*	16	∴ Result = −11
4	16, 4	
1	16, 4, 1	
+	16, 5	
−	−11	

Unit 3 Queue

1. Queue

Queue is a linear collection of data in which data are inserted from one end called as 'rear' and deleted from another end called as 'Front'.

In this data structures, the first element inserted will be the first one to be deleted. It follows FIFO.

Applications of Queue

i) Print Queue
ii) Playlist in media player
iii) Reservation or Booking Systems
iv) CPU Scheduling
v) Traversal in Graph

Some terms related to Queue

i) Enqueue
The process of inserting data in the Queue.

ii) Dequeue
The process of deleting data from the queue.

iii) Peek
The process of identifying the data to be deleted next.

iv) Full queue
The condition in which all the location of the queue contains data.

v) Empty Queue
The condition in which Queue doesn't contain any data.

vi) Overflow
The result of inserting data in a full queue.

vii) Underflow
The result of deleting data from an empty queue.

2. Implementation of Queue
Queue can be implemented by using:-
i) Array
ii) Linked list

3. Array Implementation of Queue
It is a static implementation of queue i.e. the size of the queue has to be defined in advance.
There are two types of Queue
i) Linear Queue
ii) Circular Queue

4. Algorithm to Insert data (enqueue) in a linear Queue:

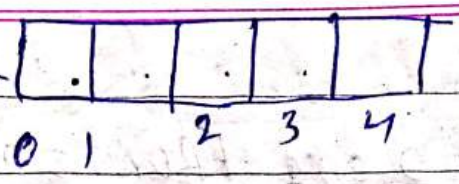
Step 1: START
Step 2: [check if the queue is full or not]
If REAR = MAX - 1
print OVERFLOW and EXIT
Step 3: [Update the value of FRONT and REAR]

If REAR = -1
    then Set REAR = 0 and FRONT = 0
else
    REAR = REAR + 1
Step 4: [Insert data at the REAR of Queue]
    QUEUE [REAR] = newdata
Step 5: EXIT

5. Algorithm to Delete data (Dequeue) from a linear Queue
Step 1 : START
Step 2 : [Check if the queue is empty or not]
    If REAR = -1
    then print UNDERFLOW and EXIT.
Step 3 : [Delete the Data]
    Delete QUEUE [FRONT]
Step 4 : [Update the value of FRONT and REAR]
    If REAR = FRONT
        then set REAR = -1
            FRONT = -1
    else
        FRONT = FRONT + 1
Step 5 : EXIT

In the above assumption Queue is an array that stores maximum of MAX numbers of elements and the initial value of FRONT & REAR is -1.

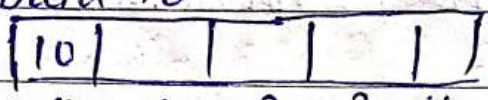
Example:-



MAX = 5

REAR = -1 FRONT = -1

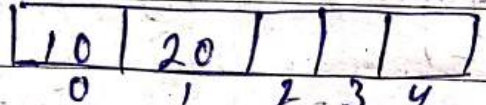
i) Insert Data 10



REAR = 0

FRONT = 0

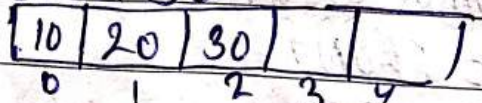
ii) Insert Data 20



REAR = 1

FRONT = 0

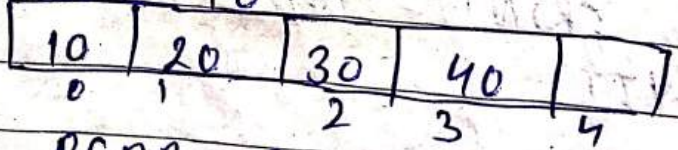
iii) Insert Data 30



REAR = 2

FRONT = 0

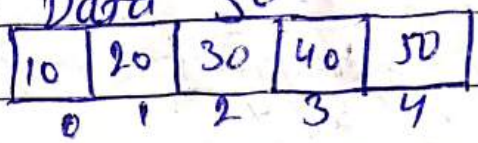
iv) Insert Data 40



REAR = 3

FRONT = 0

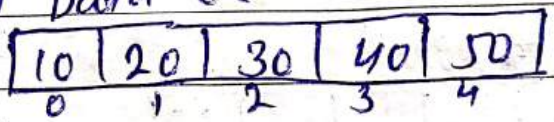
v) Insert Data 50



REAR = 4

FRONT = 0

vi) Insert Data 60

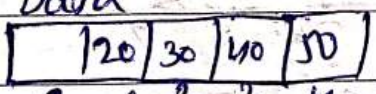


REAR = 4

FRONT = 0

OVERFLOW

vii) Delete Data

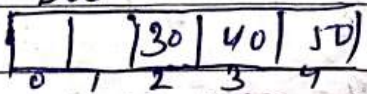


REAR = 4

FRONT = 1

Deleted data = 10

viii) Delete Data

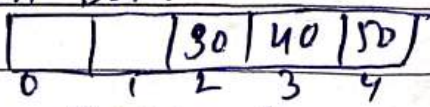


REAR = 4

FRONT = 2

Deleted Data = 20

ix) Insert Data 60

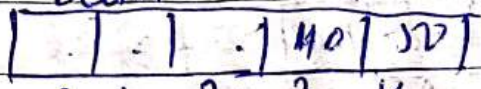


REAR = 4

FRONT = 2

OVERFLOW

x) Delete Data



REAR = 4

FRONT = 3

Deleted Data = 30

xi) Delete Data

REAR = 4

FRONT = 4

Deleted Data = 40

xii) Delete Data

REAR = -1

FRONT = -1

Deleted Data = 50

xiii) Delete Data

REAR = -1

FRONT = -1

UNDERFLOW

6. Algorithm of a Circular Queue

Algorithm to insert data (enqueue) in a circular Queue

① START

② [check if the Queue is full or Not]

if REAR = MAX-1 and FRONT = 0

Then print OVERFLOW and EXIT

OR

FRONT = REAR + 1

Then print OVERFLOW and EXIT

③ [Update the value of FRONT and REAR]

if REAR = -1

then set REAR = 0 and FRONT = 0

else if REAR = MAX-1

then set REAR = 0

else

REAR = REAR + 1

④ [Insert Data]

QUEUE [REAR] = new data

⑤ EXIT

Algorithm to Delete Data (Dequeue) from a Circular Queue

① START

② [Check if the Queue is empty or Not]

If REAR = -1

then print UNDERFLOW and EXIT

③ [Delete Data]

Delete Queue [FRONT]

④ [Update the Value of FRONT and REAR]

if REAR = FRONT

then set REAR = -1 and FRONT = -1

else if FRONT = MAX - 1

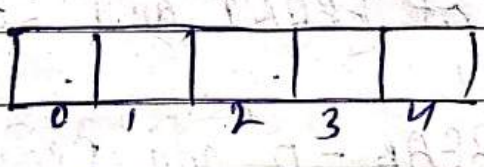
then Set FRONT = 0

else

FRONT = FRONT + 1

⑤ EXIT

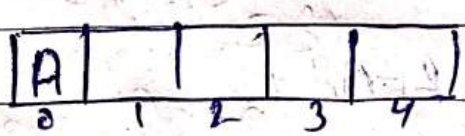
Example :-



Max = 5

REAR = -1 , FRONT = -1

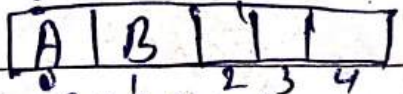
i) Insert Data A



REAR = 0

FRONT = 0

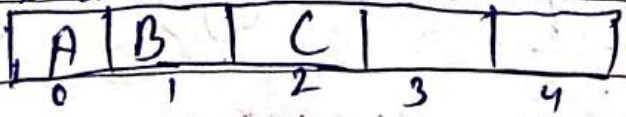
ii) Insert Data B



REAR = 1

FRONT = 0

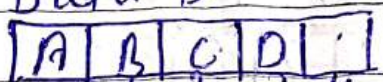
iii) Insert Data C



REAR = 2

FRONT = 0

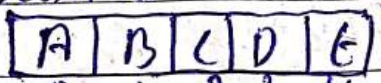
iv) Insert Data D



REAR = 3

FRONT = 0

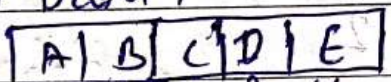
v) Insert Data E



REAR = 4

FRONT = 0

vi) Insert Data F

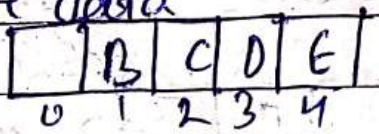


REAR = 4

FRONT = 0

OVERFLOW

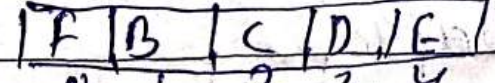
vii) Delete data



REAR = 4

FRONT = 1

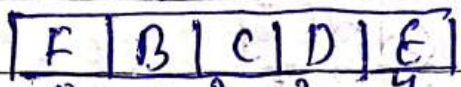
viii) Insert Data F



REAR = 0

FRONT = 1

ix) Insert Data G

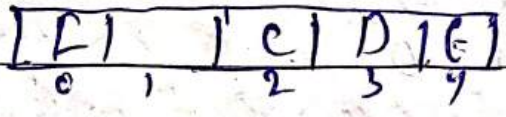


REAR = 0

FRONT = 1

OVERFLOW

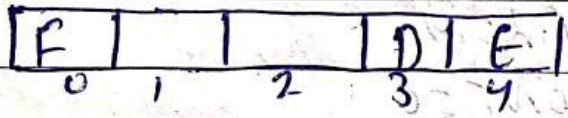
x) Delete Data



REAR = 0

FRONT = 2

xi) Delete Data



REAR = 0

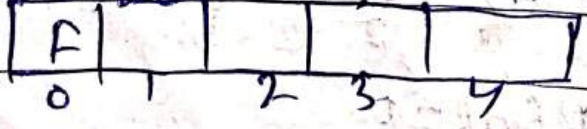
FRONT = 3

xii) Delete Data

REAR = 0

FRONT = 4

xiii) Delete Data



REAR = 0

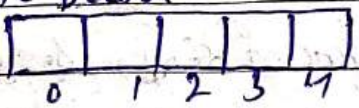
FRONT = 0

xiv) Delete Data

REAR = -1

FRONT = -1

xv) Delete Data



REAR = -1

FRONT = -1

UNDERFLOW

7. Differentiate between linear Queue and Circular Queue.

S.N	Linear Queue	S.N	Circular Queue
1.	linear Queue is a type of linear data structure that contains the elements in a sequential manner.	1.	Circular Queue is also a linear data structure where last element of the Queue is connected to the first element, thus creating a circle.
2.	It requires more memory than Circular Queue.	2.	It requires less memory as compare to linear queue.
3.	The usage of memory is inefficient	3.	The memory can be more efficient utilized.
4.	The insertation and deletation operation are fixed i.e. done at the rear and front end respectively.	4.	Insertation and deletation are not fixed and it can be done in any position.
5.	It is easy to implement as compared to Circular Queue.	5.	It is complex to implement as compared to linear Queue.

6. We cannot insert data in the initial locations even it is empty if there are data in the final locations.	6. We can insert data in the initial locations if it is empty if there are data in the final locations.
7. The overflow condition exists when REAR = MAX.	7. The overflow condition exists when REAR = MAX and FRONT = 0 OR FRONT = REAR + 1
8. While insertion the update of REAR and FRONT is done as if REAR = -1 [When the Queue is empty], then Set REAR = 0, and FRONT = 0 else Set REAR = REAR + 1	8. While insertion the update of REAR and FRONT is done as: if REAR = -1 [When the Queue is empty], Then Set REAR = 0 and FRONT = 0 else if REAR = MAX, then Set REAR = 0 else Set REAR = REAR + 1

8. Double ended Queue (Deque)

It is a variation of Queue in which data can be inserted or deleted from any end i.e. either rear or front.

i) Input restricted double ended queue

ii) Output restricted double ended queue

i) Input restricted double ended queue

In input restricted double ended queue, insertion can be done only from the rear. However deletion can be done from both rear and front.

ii) Output restricted Double ended Queue
In output restricted double ended Queue, deletion can be done only from the front whereas insertion can be done from both rear or front.

9. Priority Queue
It is a type of Queue in which the data are processed/deleted on the basis of priority assigned to the data using following rules:-
i) The data with higher priority is processed before the data with lower priority.
ii) The two or more data with same priority are processed in the order in which they are inserted in the queue (FIFO).
The priority queue can be used in application like process Scheduling, Booking System, Appointment System etc.

Types of priority Queue
i) Ascending priority Queue
ii) Descending priority Queue
In Ascending priority Queue, the smaller value of priority is processed before the larger value of priority whereas in descending priority Queue, the larger value of priority is processed before the smaller value of priority.

Unit: 4 List

1. List
A list is a linear collection data in which the data can be inserted or deleted from any position.
It is used to store the multiple collection of data.
List can be implemented by using:-
i) Array (Static list)
ii) DMA (Dynamic list)

2. Difference between Static list and Dynamic list.

S.N	Static list	S.N	Dynamic list
1.	In static list, the size of memory list is fixed.	1.	In dynamic list, the size of list is define during operation.
2.	The size of allocated memory cannot be changed during run time.	2.	The size of memory can be changed during run time.
3.	The size of data in static list should be equal to or less than predefined size.	3.	Any size of data can be inputted because the user can define how much memory required during execution.
4.	It is less efficient in memory uses.	4.	It is more efficient in memory uses.
5.	It is easy to implement.	5.	It is complex to implement.
6.	Elements are stored in continuous memory location.	6.	Elements can be stored anywhere in the memory.
7.	Example:- Array	7.	Example:- Linked list

3. Array implementation of List

(1) Creating a list
   i) Initilize the array
   ii) Insert the data in the array.

(2) Inserting data at the specified position in the list.
   i) Check for overflow i.e. check if the list is full or not.
   ii) Read the data and the position to insert the data.
   iii) Shift the data from the location in which the data has to be inserted till the end by one location forward.
   iv) Insert the new data at the specified location.

(3) Deleting data from the specified position in the list.
   i) Check for Underflow i.e. check if the list is empty or not.
   ii) Read the position of data to be deleted.
   iii) Display the data to be deleted.
   iv) Shift till the data after specified location till the end by one position backward.

(4) Update the data in the list
   i) Read the New data and the location of the data to be updated.
   ii) Replace the data at the specified position by the new data.

(5) Sorting the data in the list
   i) Arrange data in some logical order such as ascending or descending

(6) Traversing the list
   i) Traversing can be done by simply using a loop from the first location of the list till the last.

(7) Merging two lists

i) Merging a list includes appending data of one list to the end of another list.

4. Queue as a list

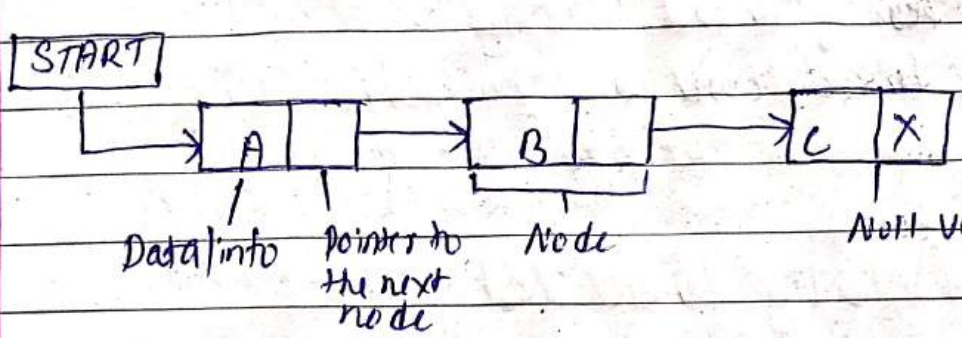
Since, Queue follows FIFO order, It can be implemented by using list with some restriction. Insertion can be done from only one end and deletion from another end.

Similarly, list can be implemented from Queue by allowing insertion and deletion from any position without only depending on the front and the rear.

Unit:- 5 Linked lists

1. Linked list

A linked list is a dynamic list. It is implemented by using concept of pointer and DMA (Dynamic Memory Allocation).



∴ Fig:- Linked list

In a linked list, data are stored in a nodes. Each node is divided into 2 parts. The first contains the data and the second part contains address of the next node (pointer to the next node). The pointer filled up the last node contains NULL which is any invalid memory space.

A linked list also contains a pointer start which points the first node.

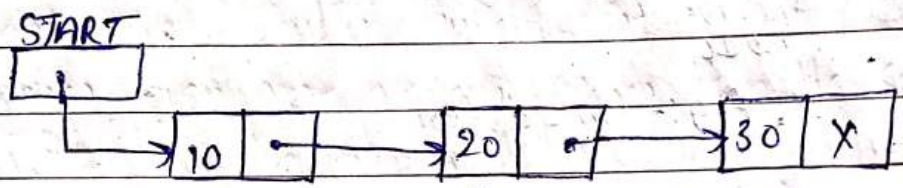
Advantages of using linked list

i) Since, it is a dynamic list, it can grow or shrink as per our requirement during run time.
ii) It is efficient in terms of memory utilization.
iii) It is faster for execution due to the use of pointer.
iv) It can be used to implement data structures like stack, queue, list, tree, Graph etc.

Types of linked list

a. Singly linked list
b. Circular linked list
c. Doubly linked list
d. Circular doubly linked list

a. Singly linked list



It is a most common type of linked list. The node of a singly linked list is divided into two parts, the first part contains the data and the second part contains the pointer to the next node. The pointer field of the last node contains NULL. It contains an additional pointer, START which points the first node.

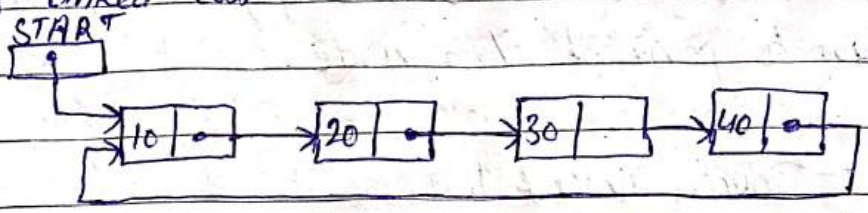
Advantages of Singly linked list

i) It can grow or shrink as per requirement.
ii) Insertion and deletion can be done at any position.
iii) The execution ~~will faster~~ Speed will be faster.

Disadvantages of Singly linked list

i) Traversal can be done only in one direction. Backward traversal is not possible.
ii) It requires more memory space than the Static list as both data and pointer has to be stored.

b. Circular linked list



It is the modification of singly linked list so that the last node points the first node. The pointer field of the last node contains the address of the first node.

Advantages of Circular linked list

i) The traversal from last node to the first node is possible. Thes makes traversal easier and faster.

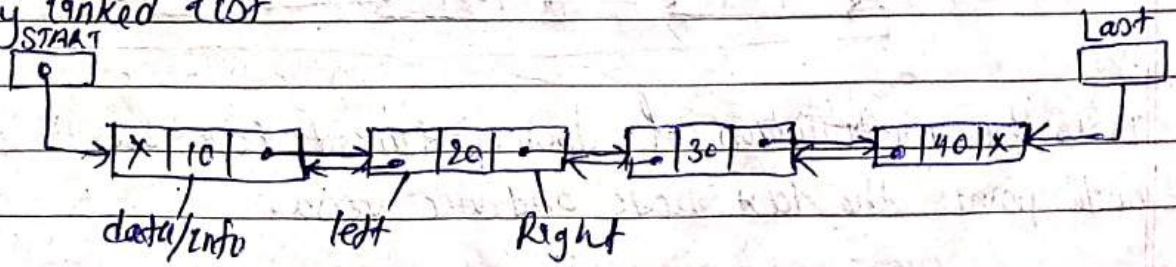
- [It covers the advantages of singly linked list too.]

Disadvantages of Circular linked list

i) Backward or Reverse traversal is not possible.

[It covers the disadvantages of singly linked list too.]

c. Doubly linked list



In a doubly linked list, the node is divided into three parts:

- The central part contains the data
- The left part points the previous node
- The right part points the next node
- The left of the first Node and right of the last node contains NULL

—It contains two additional pointers Start to point the first node and last to point the last node.

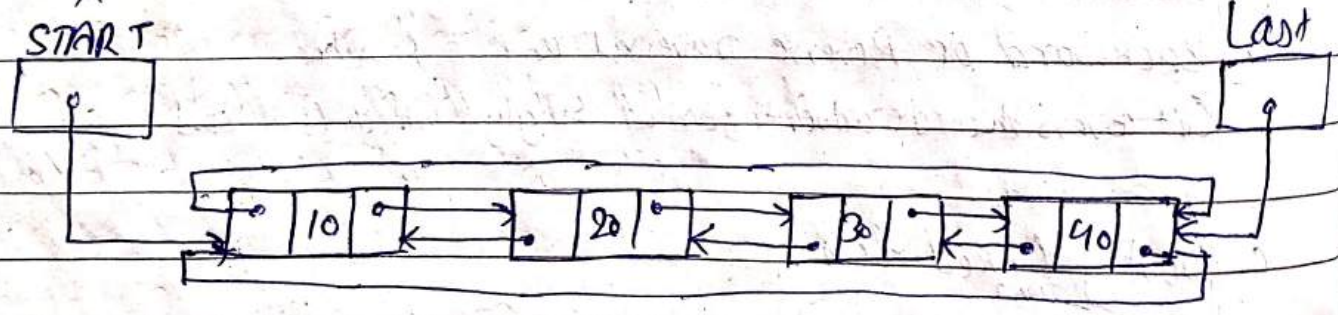
Advantages of Doubly linked list

i) It supports both forward and backward traversal.
[Contains the advantages of other list too.]

Disadvantages of Doubly linked list

i) It requires even more memory spaces than the singly link list due to the use of double pointer.
ii) Insertion and deletion is complex.
iii) Traversal from last to first node and vice versa is not possible.

d. Circular doubly linked list



It is the modification of doubly linked list such that the first node points the last node and vice versa.

Advantages of Circular doubly linked list

i) It supports forward, backward and circular traversal
[Contains the advantages of other list too.]

Disadvantages of Circular doubly linked list

i) [Contains the disadvantages of other list too.]

2. Implementation of Singly linked list

The implementation of ~~Singly~~ linked list includes the operation like

i) Inserting a node/data

ii) Deleting a node / data

iii) Traversing the linked list

iv) Searching in the linked list

Q. Inserting a node/Data in Singly linked list

Steps

i) Allocate a new node

ii) Assign data to the new node

iii) Adjust the pointer

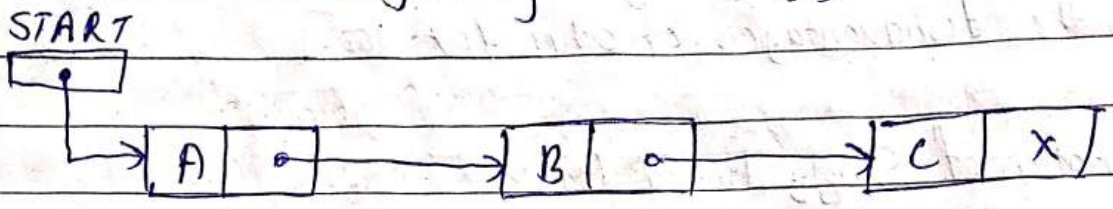
The insertion can have three cases:-

i) Insertion at the Beginning of a Singly linked list (Inserting as the first node)

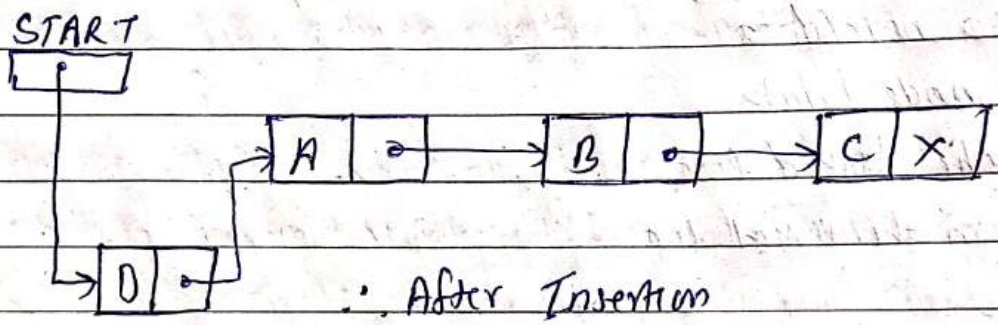
ii) Insertion at the End of a Singly linked list (Inserting node as the last node)

iii) Insertion at the Specified position of a Singly linked list (Inserting in Between Nodes)

Insertion at the Beginning of a Singly Linked List



∴ Before Insertion



∴ After Insertion

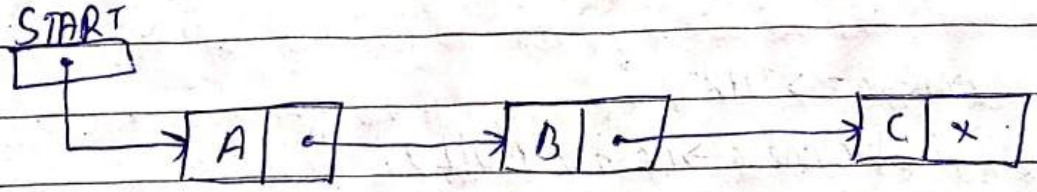
Algorithm 1

1. Allocate a New Node
2. Assign Data to the New Node
3. Adjust pointer so that:-
   a) New Node points the first Node
   b) Start points the New Node

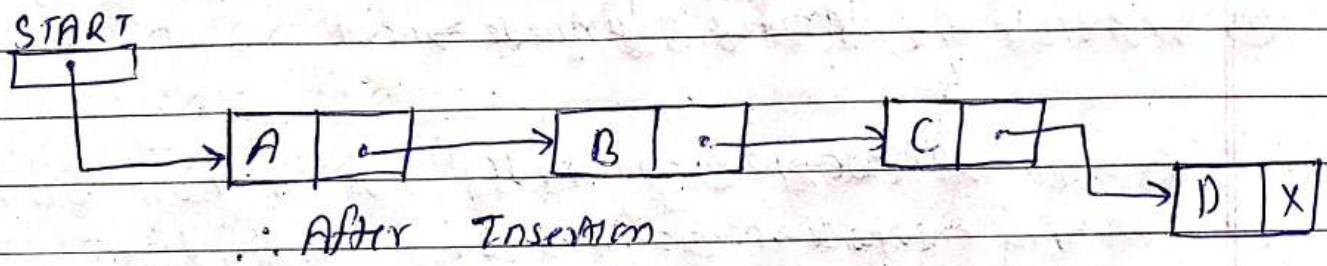
Algorithm 2

1. START
2. [Allocate a new Node]
   newnode = (nodeType *) malloc (sizeof (nodeType))
3. Read the data (el) to insert.
4. Assign newnode → info = el
5. newnode → next = START
6. START = newnode
7. END

Insertion at the End of a Singly Linked list



∴ Before Insertion



∴ After Insertion

Algorithm 1

① Allocate a new node
② Assign data to the new node
③ Adjust the pointer, so that:
   a) Last node points the new node
   b) New node contains NULL in its pointer field

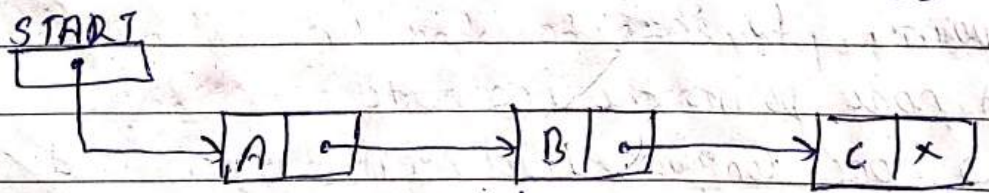
Algorithm 2

① START
② [Allocate a new node]
   newnode = (nodeType *) malloc (sizeof (nodeType))
③ Read the data (el) to insert
④ Assign newnode → info = el
⑤ if START = NULL [when there are no any nodes]
   {
      START = new node
      newnode → next = NULL
   }

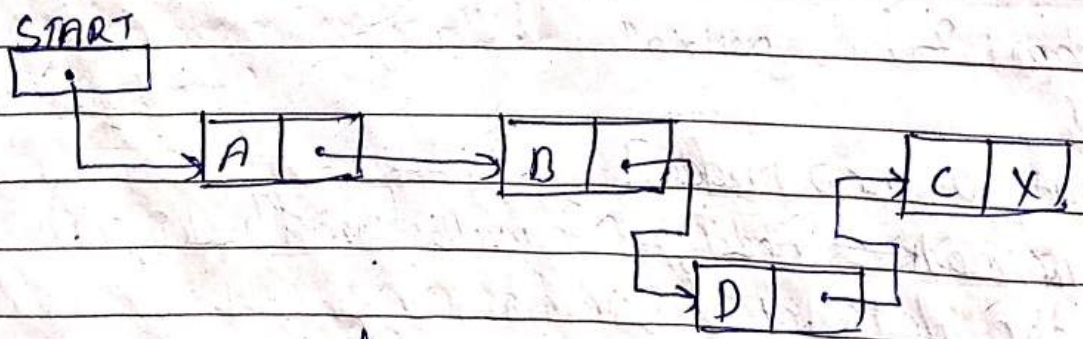
```
else
{
        lastnode = START
        While (lastnode -> next ≠ NULL)
        {
            lastnode = lastnode -> next
        }
        lastnode -> next = newnode
        newnode -> next = NULL
    }
```

(6) END

Insertion at the specified position of a Singly linked list



∴ Before Insertion



∴ After Insertion

Algorithm 1

1. Allocate a newnode
2. Assign data to the newnode
3. Adjust the pointer, so that:
   a) newnode points the nextnode of the specified position
   b) previous node of the specified position points the new node.

Algorithm 2

1. START
2. [Allocate a newnode]
   newnode = (nodeType *) malloc (sizeof(nodeType))
3. Read the data (el) to insert
4. Assign newnode → info = el
5. Read the position (pos) to insert
6. Set previousnode = START, i = 1
7. While (i < pos - 1)
   {
   previousnode = previousnode → next
   i = i + 1
   }
8. nextnode = previousnode → next
9. newnode → next = nextnode
10. previousnode → next = newnode
11. END

b. Deletion of nodes in a Singly linked list

The deletion of the node requires following steps

i) Check if the linked list is empty or not. If it is empty, print underflow and exit.

ii) Adjust pointer so that the node to be deleted is not linked

iii) Deallocate the deleted node.

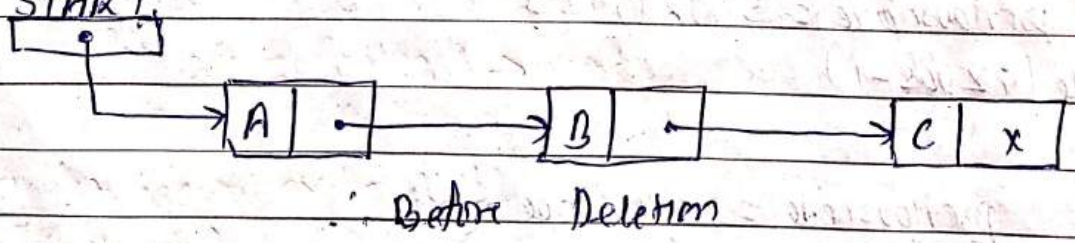
The deletion in the singly linked list can have following cases:-

i) Deleting the first node of a singly linked list

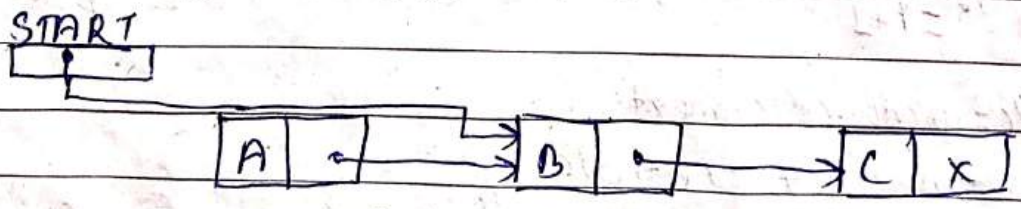
ii) Deleting the last node of a singly linked list

iii) Deleting the node at the specified position of a singly linked list.

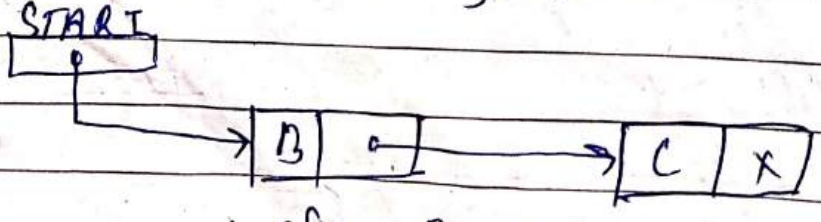
Deleting the first node of a Singly linked list



∴ Before Deletion



∴ During Deletion



∴ After Deletion

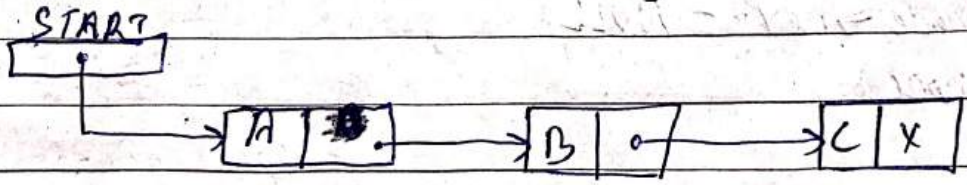
Algorithm 1

① Check if the linked list is empty or not. If it is empty, print UNDERFLOW and exit
② Adjust pointer so that START points the second node.
③ Deallocate the first node

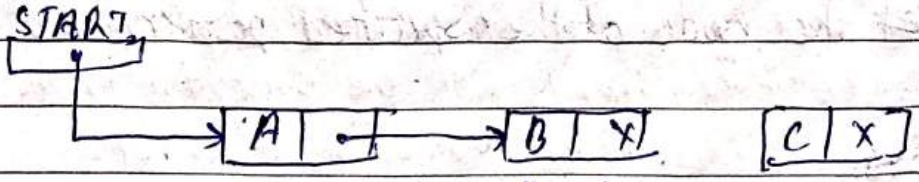
Algorithm 2

① START
② if START = NULL
then print UNDERFLOW and exit
③ firstnode = START
④ Secondnode = firstnode → next
⑤ START = Secondnode
⑥ free (firstnode)
⑦ END

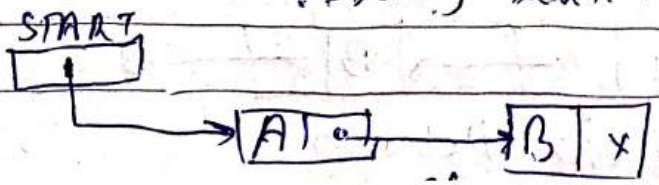
Deleting the last node of a Singly linked list



∴ Before Deletion



∴ During Deletion



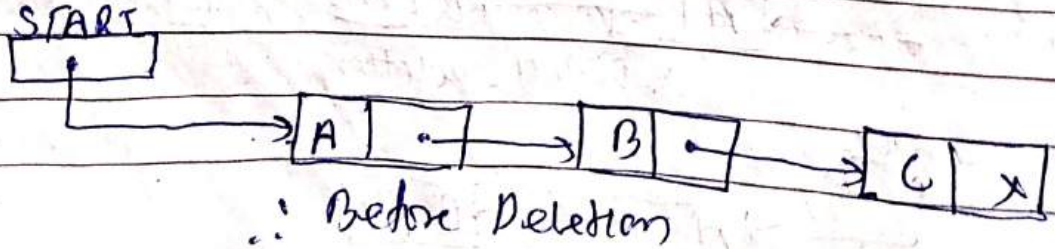
Algorithm 2

(1) check if the linked list is empty or not. If it is empty, print UNDERFLOW and exit.
(2) Adjust pointer so that the second last node contains null in its pointer field.
(3) Deallocates the lastnode

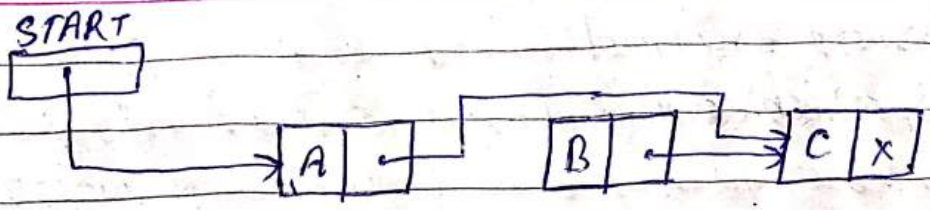
Algorithm 2

(1) START
(2) If START = NULL,
    then print UNDERFLOW and Exit
(3) Set lastnode = START
(4) While (lastnode → next ≠ NULL)
    {
        Secondlastnode = lastnode
        lastnode = lastnode → next
    }
(5) Secondlastnode → next = NULL
(6) free(lastnode)
(7) END

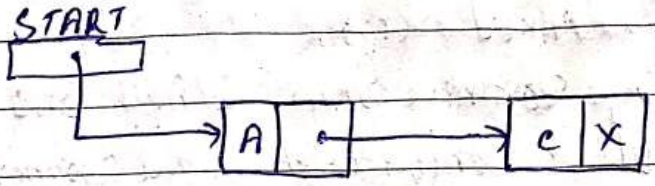
Deleting the node at the Specified position of a Singly linked list



∴ Before Deletion



∴ During Deletion



∴ After Deletion

Algorithm 1

① Check if the linked list is empty or not. If it is empty, print UNDERFLOW and exit.

② Adjust pointer so that previous node of the specified position points the next node.

③ Deallocate the node at the specified position.

Algorithm 2

① START

② If START = NULL,
then print UNDERFLOW and exit

③ Read the position of the node (POS) to be deleted

④ Set specificnode = START and i = 1

⑤ While (i < POS)

{

previousnode = specificnode

specificnode = specificnode → next

nextnode = specificnode → next

i = i + 1

}

(6) prevnode → next = nextnode
(7) Deallocate (Specificnode)
(8) END

c. Traversing in a Singly linked list
Traversing can be done by starting from START and visiting all the nodes once until NULL is encountered in the pointer field of the node.

Algorithm
(1) START
(2) Check if START = NULL
then Print the linked list is empty and exit
(3) Set node = START
(4) While(node → next ≠ NULL)
{
node = node → next
}
(5) EXIT

d. Searching an element in a Singly linked list
Searching is done to check whether the required data is in the linked list or not. The search will be successful if the required data is found in the linked list. otherwise, it is unsuccessful.

Algorithm

(1) START

(2) Check if START = NULL
    then print the linked list is empty and exit.

(3) Read the data (el) to search.

(4) Set node = START and Search = 0

(5) While (node ≠ NULL)
{
    if (node → info = el)
        Set Search = 1
    node = node → next
}

(6) if search = 1
        print Search is Successful.
    else
        print Search is Unsuccessful.

(7) END

3. Implementation of Doubly linked list

The implementation of doubly linked list includes the operation like -

a) Inserting a node / data

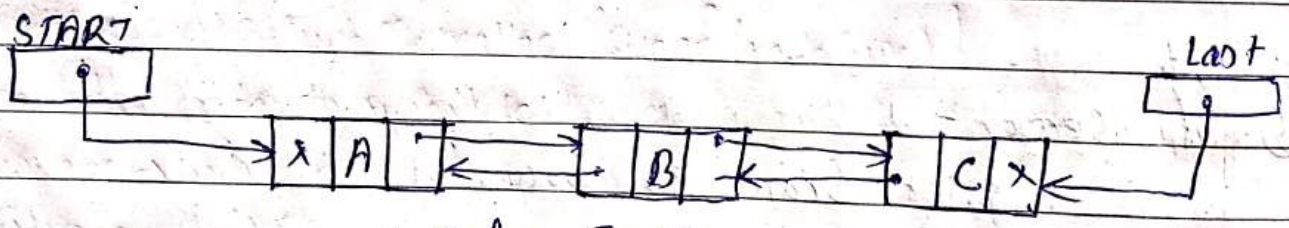
b) Deleting a node / data

c) Traversing the doubly linked list

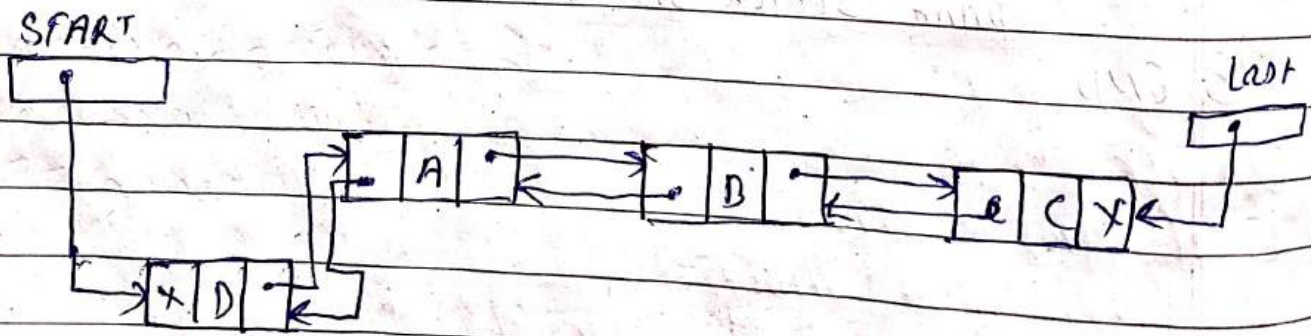
d) Searching the doubly linked list

a. Insertion in a doubly linked list

Insertion at the beginning of a doubly linked list



∴ Before Insertion



∴ After Insertion

Algorithm 1

(1) Allocate a new node

(2) Assign data to the new node

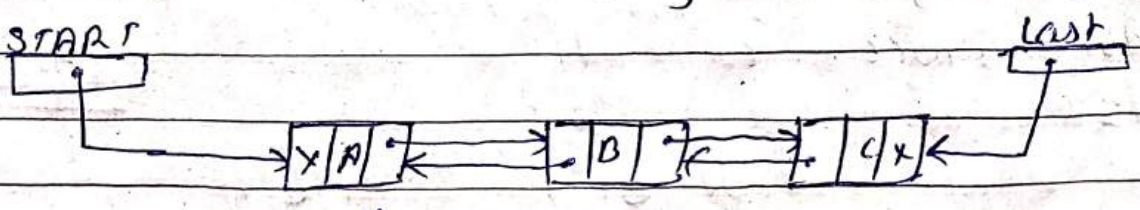
(3) Adjust pointers, so that:

a) Right of the newnode points the first node
b) left of the firstnode points the new node
c) left of the ~~first~~ new node contains NULL
d) START points the new node.

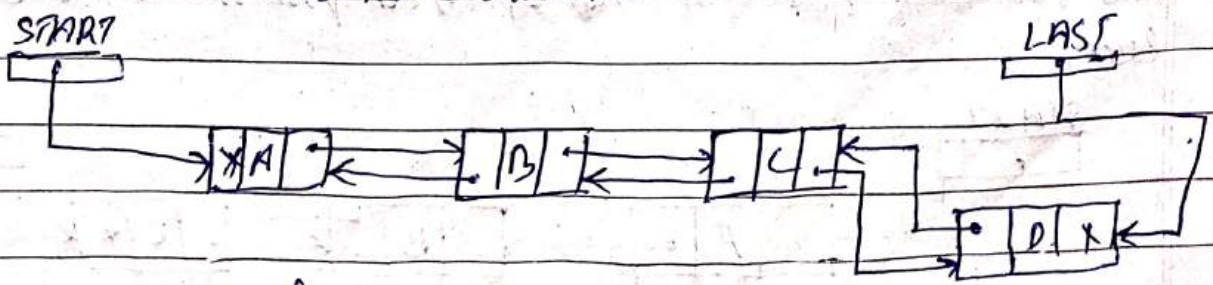
Algorithm 2

1. START
2. newnode = (nodeType *) malloc (sizeof (nodeType))
3. Read the data to insert (el)
4. newnode → info = el
5. firstnode = START
6. newnode → right = firstnode
7. firstnode → left = newnode
8. newnode → left = NULL
9. START = newnode
10. END

Insertion at the End of a doubly linked list



∴ Before Insertion



∴ After Insertion

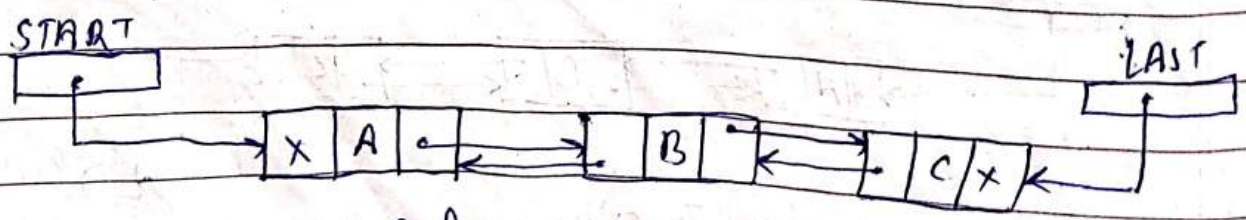
Algorithm 1

(1) Allocate a newnode
(2) Assign data to the new node
(3) Adjust pointer, so that :-
    a) Left of the newnode points the last node
    b) Right of the lastnode points the newnode
    c) right of the newnode contains NULL
    d) LAST points the newnode

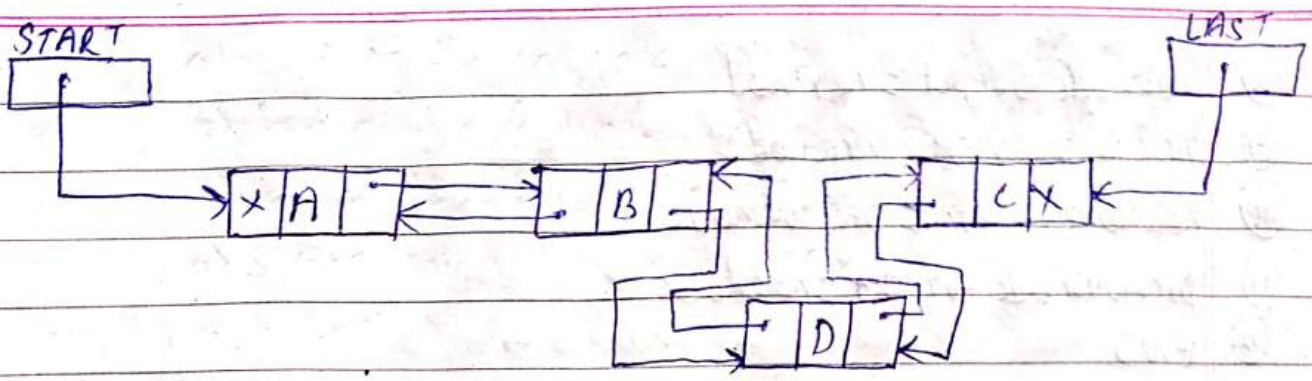
Algorithm 2

(1) START
(2) newnode = (nodeType *) malloc (sizeof (nodeType))
(3) Read the data to insert (el)
(4) newnode → info = el
(5) lastnode = LAST
(6) newnode → left = lastnode
(7) lastnode → right = newnode
(8) newnode → right = NULL
(9) LAST = newnode
(10) EXIT

Insertion the node at the specified position of a doubly linked list



∴ Before Insertion



∴ After Insertion

Algorithm 1

① Allocate a new node
② Assign data to the new node
③ Adjust pointer, so that :-
   a) Right of the new node points the next node
   b) left of the next node points the new node
   c) Left of the new node points the previous node
   d) Right of the previous node points the new node

Algorithm 2

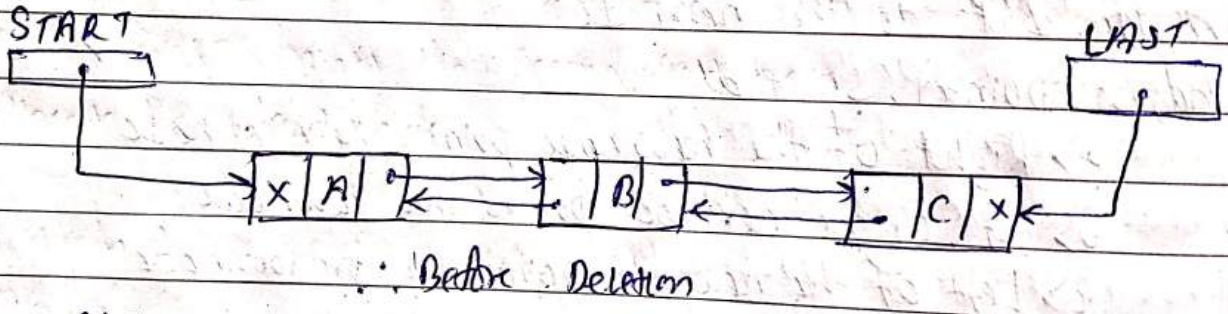
① START
② newnode = (nodeType *) malloc (Sizeof (nodeType))
③ Read the data to insert (el)
④ newnode → info = el
⑤ Read the position of the specified node (pos)
⑥ Set nextnode = START , i = 1
⑦ While (i < pos)
{

previousnode = nextnode
nextnode = nextnode → right
i = i + 1

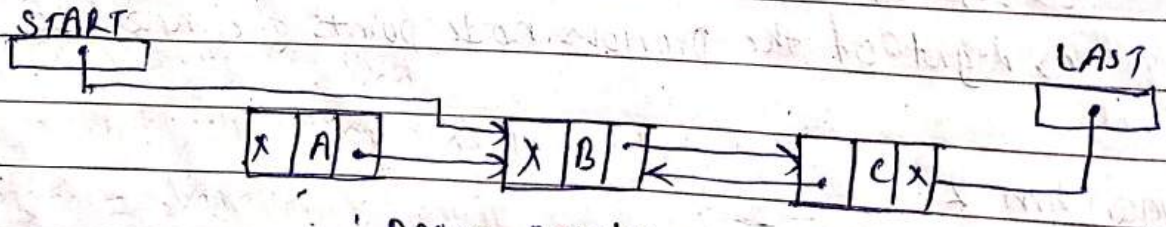
(8) newnode → right = nextnode
(9) nextnode → left = newnode
(10) newnode → left = previousnode
(11) previousnode → right = newnode
(12) END

b. Deletion of nodes in a doubly linked list

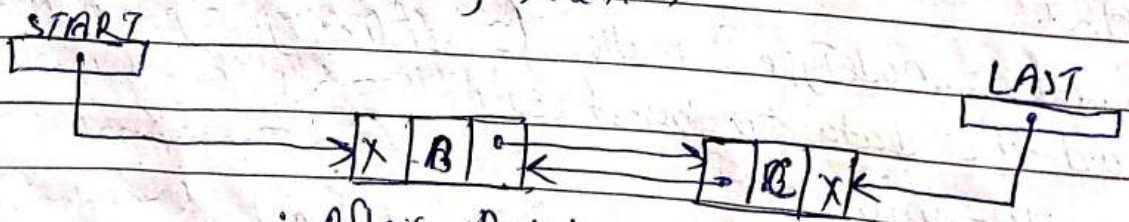
Deleting the firstnode of a doubly linked list



∴ Before Deletion



∴ During Deletion



∴ After Deletion

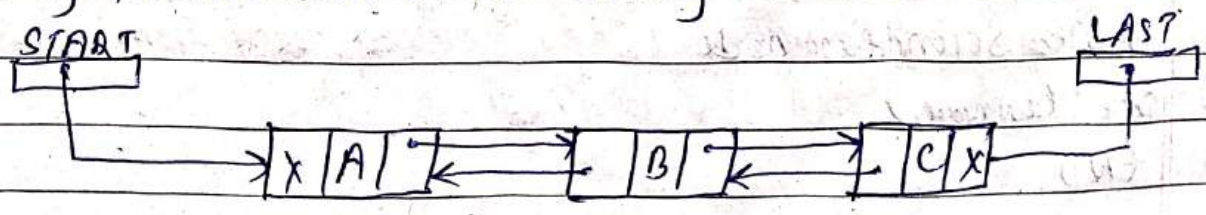
Algorithm 1

(1) Check if the linked list is empty or Not. If it is empty, print UNDERFLOW and exit
(2) Adjust pointer, so that :-
    a) left of the second Node Contains NULL
    b) START points the second node
(3) Deallocate the first node

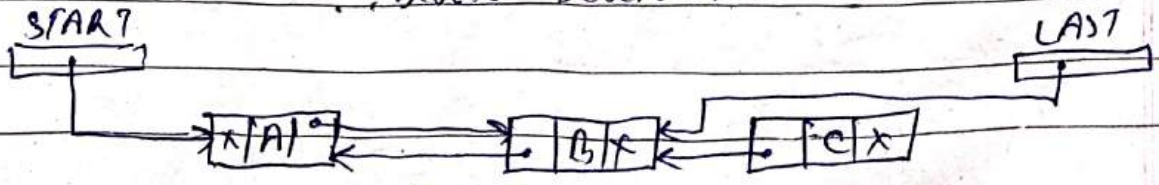
Algorithm 2

(1) START
(2) If START = NULL, then
        Print UNDERFLOW and exit
(3) firstnode = START
(4) secondnode = firstnode → right
(5) secondnode → left = NULL
(6) START = secondnode
(7) free (firstnode)
(8) END

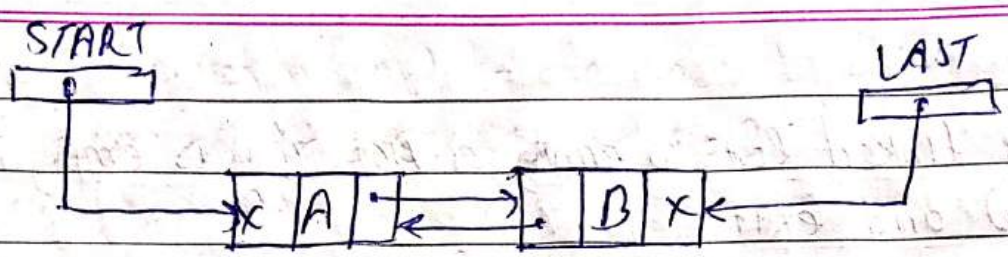
Deleting the lastnode of a doubly linked list



∴ Before Deletion



∴ During Deletion



∴ After Deletion

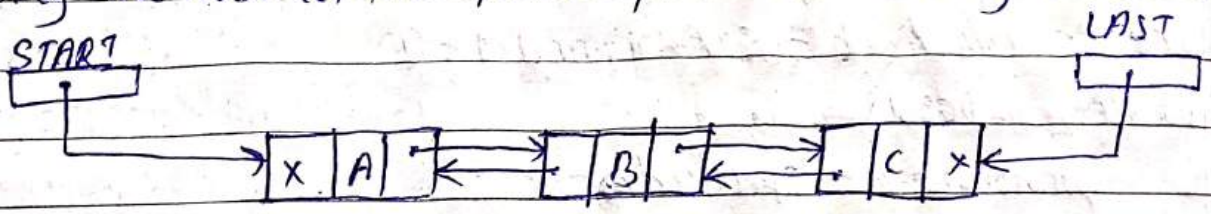
Algorithm 1

1. Check if the linked list is empty or not. If it is empty, print UNDERFLOW and Exit
2. Adjust the pointer, so that :
   a) Right of the second last node contains NULL
   b) LAST points the second last node
3. Deallocate the last node

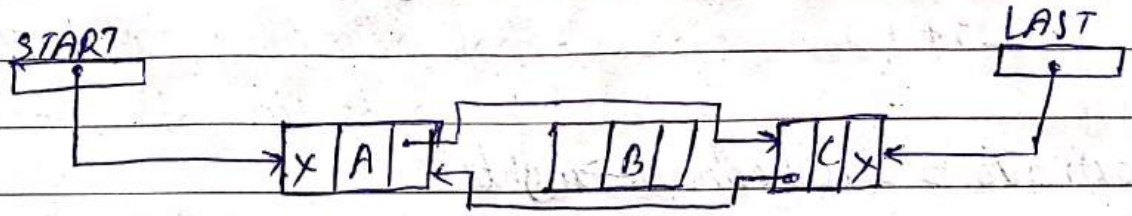
Algorithm 2

1. START
2. If START = NULL, then
   print UNDERFLOW and exit
3. lastnode = LAST
4. Secondlastnode = lastnode → left
5. Secondlastnode → right = NULL
6. LAST = Secondlast node
7. free (lastnode)
8. END

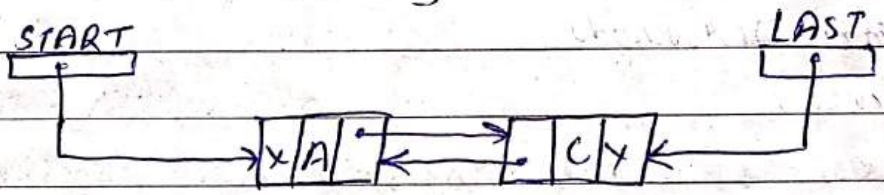
Deleting the node at the Specified position of a doubly linked list



∴ Before Deletion



∴ During Deletion



∴ After Deletion

Algorithm 1

(1) Check if the linked list is empty or not. If it is empty, print UNDERFLOW and exit

(2) Adjust the pointers, So that :-
   a) Right of the previous node points the next node.
   b) left of the next node points the previous node.

(3) Deallocate the Specified node

Algorithm 2

(1) START

(2) If START = NULL, then
   print UNDERFLOW and exit

(3) Read the Position (pos) of the node to delete.
(4) Set specifiednode = START, and i = 2
(5) While (i<pos){
    previousnode = specifiednode
    specifiednode = specifiednode → right
    i = i + 1
}
(6) nextnode = specifiednode → right
(7) previousnode → right = nextnode
(8) nextnode → left = previousnode
(9) free(specifiednode)
(10) Exit

c. Traversal in doubly linked list
Traversal can be :-
    i) forward Traversal
    ii) Backward Traversal

In forward traversal, the traversal starts from START and traverses all the nodes till the last node.
In Backward Traversal, the traversal starts from LAST and traverses all the nodes till the first node.

Algorithm for Forward Traversal

(1) START
(2) Set temp = START
(3) While (temp → right ≠ NULL) {
        temp = temp → right
    }
(4) Exit

Algorithm for Backward Traversal

(1) START
(2) Set temp = LAST
(3) While (temp → left ≠ NULL) {
        temp → temp → left
    }
(4) Exit

d. Searching elements in doubly linked list

Algorithm

(1) START
(2) if START = NULL, then
        print the linked list is empty and Exit
(3) Read the data(el) to search
(4) Set temp = START, Search = 0
(5) While (temp → right ≠ NULL) OR temp → info ≠ el)
    {

```
    if (temp->info = el)
        Set Search = 1
    temp = temp->right
}
```

(6) if Search = 1 then,
    print Search is Successful.
else
    print Search is Unsuccessful.

(7) Exit

→ Implementation of Circular Linked List
→ Implementation of doubly Circular Linked List

4. Linked list implementation of STACK

It is the dynamic implementation of stack. Stack can be implemented using linked list with certain restriction:-

i) Inserting node and deleting node only from the beginning of the linked list.
ii) Inserting and deleting node only from the end of the linked list.

5. Linked list implementation of Queue

It is the dynamic implementation of queue. It can be implemented by using certain restrictions such as:-

i) Inserting node only from the beginning and deleting the node only from the end.

ii) Inserting node only from the end of the linked list and deleting node only from the beginning of the linked list.

Unit :- 6 Recursion

1. Recursion

Recursion is a concept of repeating the execution of statements for multiple times by calling the function or procedure itself until some condition (stopping condition) is satisfied.

To be a recursive function, it should

i) Call itself.
ii) Have a stopping condition.
iii) Every time the function is called, it should be closer to the stopping condition.

Recursion is used to repeat the execution of statement for multiple times similar to loop however it is not as efficient as loop. It requires larger memory space and more processing steps than loop however it is one of the most natural way to solve the problem.

2. Differentiate between recursion and iteration.

S.N	Recursion	S.N	Iteration
1.	Function calls itself.	1.	A set of instruction repeatedly executed.
2.	It contains set of instructions, statement calling itself and a termination condition.	2.	It contains initialization, increment, condition, set of instruction within a loop and a control variable.
3.	Recursion terminates when a base case is recognized.	3.	An iteration terminates when the loop condition fails.

4. It is usually slower than iteration due to the overhead of maintaining the stack.	4. An iteration doesnot use the stack so it's faster than recursion.
5. Recursion uses more memory compare to iteration.	5. Iteration uses less memory compare to recursion.

6. Example:-
#include<stdio.h>
int factorial (int n){
if(n==0)
return 1;
return (n* factorial(n-1));
}
void main() {
factorial(5);
}
	6. Example:-
	#include<stdio.h>
	void main() {
	int i, fact=1, n=5;
	for(i=1; i<=n; ++i)
	fact = fact * i;
	}

3. Types of Recursion

i) Direct recursion

ii) Indirect Recursion

i) Direct Recursion

In a direct recursion, the function calls itself directly.

ii) Indirect Recursion

In Indirect recursion, the function calls itself through the use of another function.

```
function 1()
{
    function 1();
}
```

∴ Direct recursion

```
function 1()
{
    function 2();
}
function 2()
{
    function 1();
}
```

∴ Indirect recursion

4. Application of Recursion
i) Factorial Calculation
ii) Fibonacci Series Generation
iii) TOH (Tower of Hanoi)
iv) Quick Sort and merge Sort
v) Binary Search
vi) Binary Search Tree (BST)

5. Algorithm to calculate the factorial of a Number.
① START
② Input a number (n)
③ r = Factorial (n)
if n = 1
then return 1
else
return (n * factorial (n-1))
④ Output r
⑤ EXIT

6. Algorithm to calculate the sum of all natural Numbers upto n.
① START
② Input a number (n)
③ r = Sum (n)
if n = 1
return 1
else
return (n + Sum (n-1))

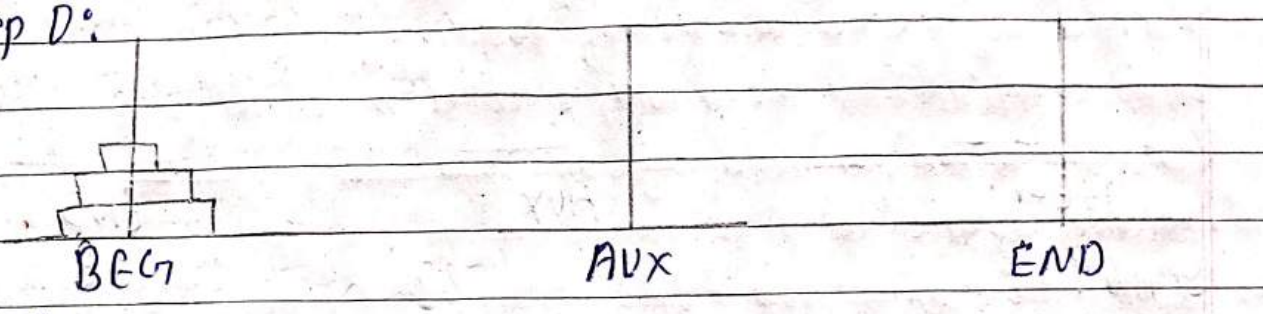
(4) Output s
(5) EXIT

7. Algorithm to calculate the term of a fibonacci sequence
(1) START
(2) Input a number (n)
(3) term = fibo(n)
if n=1
return 0
else if n=2
return 1
else
return (fibo(n-1) + fibo(n-2))
(4) Output term
(5) EXIT

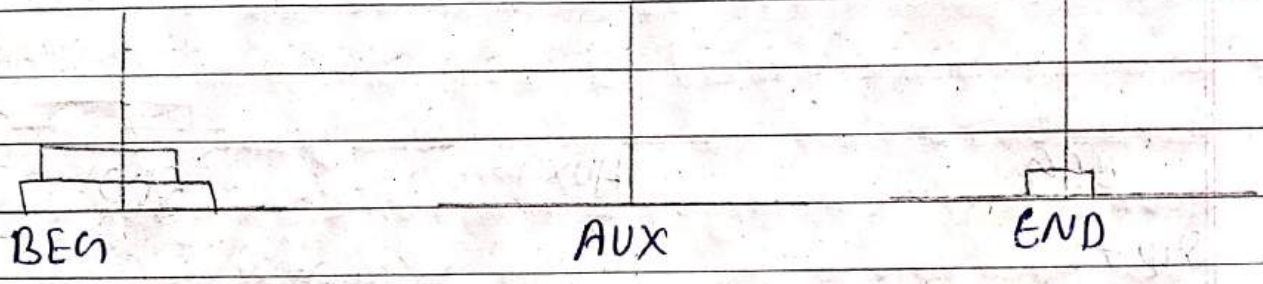
8. Tower of Hanoi (TOH)
TOH is a classical problem that can be solved by using recursion. It is a game that contains three stands/pegs (Beg, Aux, End). BEG contains 'n' number of disk with decreasing size on bottom to top. The target or goal of the game is to move all the disk from Beg to End using Aux. The rule of the game are:-
i) Only one disk can be moved at a time. Only the top most disk can be moved.
ii) A longer disk cannot be placed above the smaller disk at any point of time.
Note:- Total steps required $(2^n - 1)$.

Tracing (Tracing for n=3)

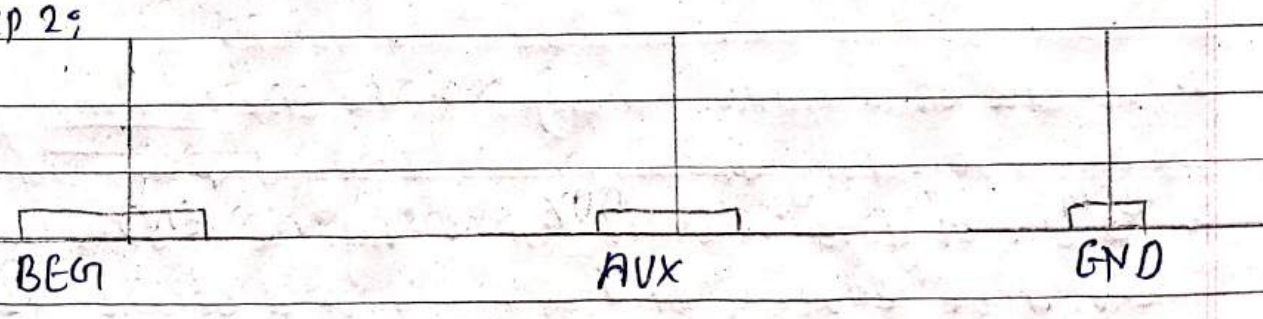
Step 0:



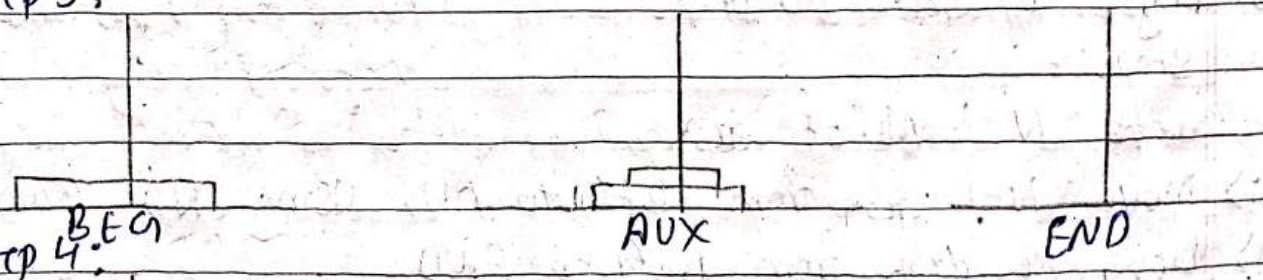
Step 1:



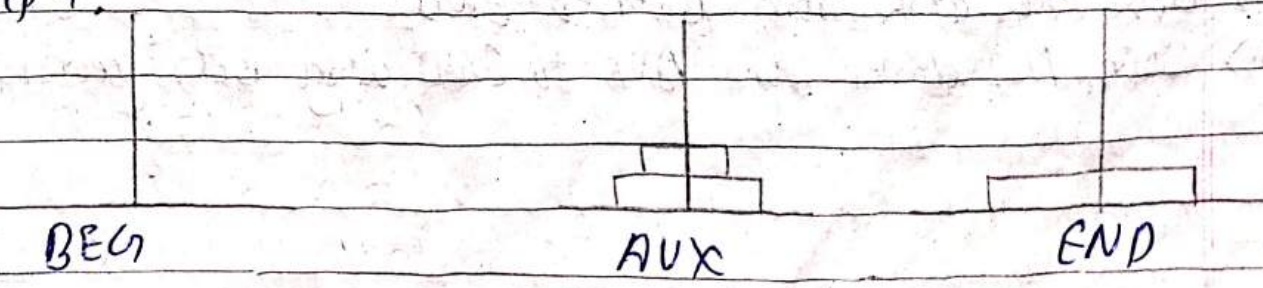
Step 2:



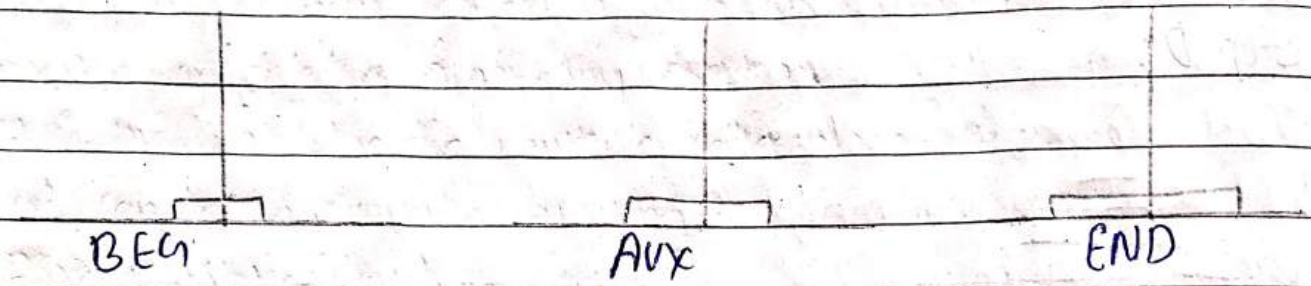
Step 3:



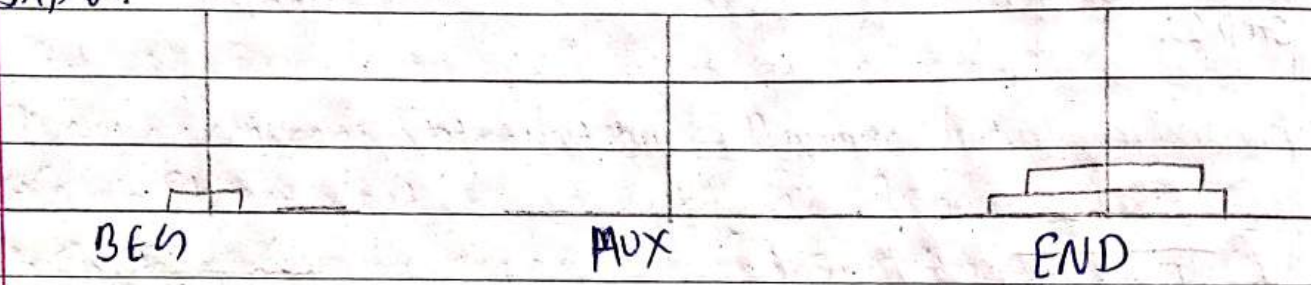
Step 4:



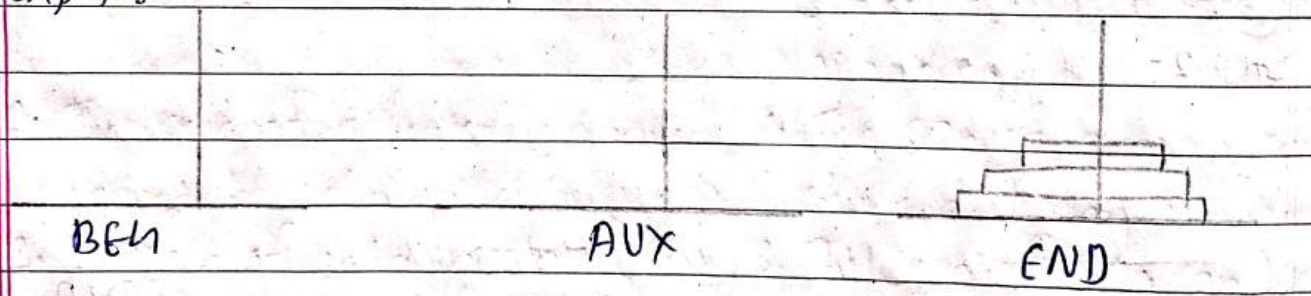
Step 5:



Step 6:



Step 7:



9. Logic to Solve TOH

Steps:-

Lets N = No. of disk

i) Move N-1 disk from BEG to AUX using END recursively

ii) Move the disk from BEG to END

iii) Move N-1 disk from AUX to END using BEG recursively.

Algorithm to Solve TOH

1. ① START
   ② Input the Number of desk (N)
   ③ TOH (N, BEG, AUX, END)
      a) if N=1
         Write BEG → END and Return
      b) TOH (N-1, BEG, END, AUX)
      c) Write BEG → END
      d) TOH (N-1, AUX, BEG, END)
      e) Return
   ④ EXIT

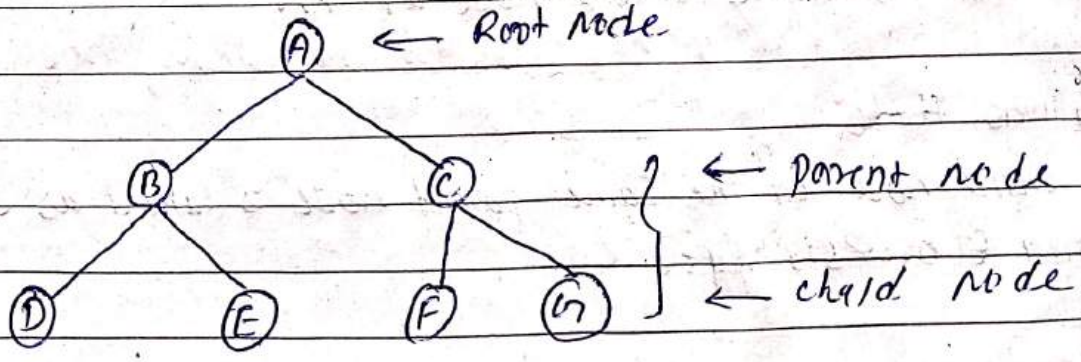
Unit :- 7 Trees

I. Tree

A tree is a non-linear data structure that models the hierarchical organization of data. It is used to store data that are represented in the hierarchical form such as folder structure, family structure, table of contains etc.

Application of Tree

- Representing the files system of a Computer
- Router Algorithm
- Syntax tree used in compilers
- Heap sort
- B Tree and B+ tree are used for indexing in database
- It is used to organize points in multi dimensional space
- Suffix tree is used for quick pattern searching



∴ Fig :- Tree

1) Root Node

It is the top most node of the tree. A tree contains a single root node.

ii) Terminal / Leaf Node
The node which has no child nodes is called the terminal node. Here, D, E, F & G are terminal nodes.

iii) Non-Terminal Node
The node that contains the child node is called non-terminal node. Here, A, B, C are non-terminal nodes.

iv) Parent Node
All the non-terminal nodes are the parents nodes. Here, B is the parent node of 'D' and 'E'.

v) Child Node
The node below the parent node connected by it's edge is called child node. B is the child node of A.

vi) Siblings ~~Node~~
The child node of the same parent node is called as sibling. Here D and E are sibilings.

vii) Ancestors
The ancestor of a node are all the nodes along the path from the node to the root. Here, C and A are ancestors of G.

viii) Level of the tree

The level of the tree represents the generation of nodes of the tree. the root is at level 0, the child of root are at level 1 and the Grand child of root at level 2 and so on.

ix) Depth of the tree

The depth of the tree is the longest path from root to the leaf node. The depth of the above tree is 2.

x) Depth of a node

The length of the path from the root to a particular node. In the above example, the depth of node E is 2 and B is 1.

xi) Height of a Node

It is the length of the path from the node to the deepest leaf. In the above example, the height of B is 1.

2. Binary Tree

A tree is called as binary tree if all the nodes contains at most two child node. A node in a binary tree can have 0, 1 or 2 child nodes.

Example:-

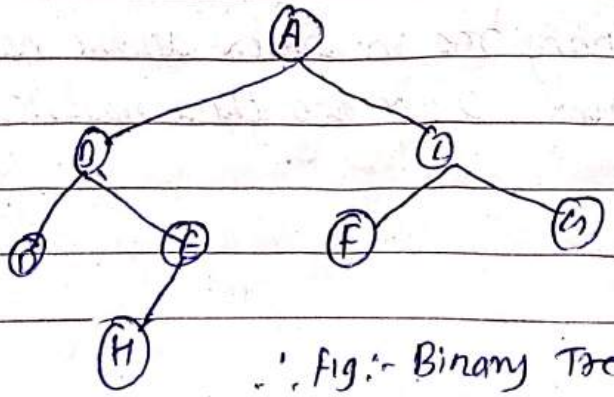


Fig:- Binary Tree

Representation of Binary Tree

A binary tree can be represented in the memory by using the linked list. The list contains the nodes. Each node is divided into three parts: The central part contains data, the left part points the left child node and the right part points the right child node. The left and the right part contains NULL if the node doesn't have child.

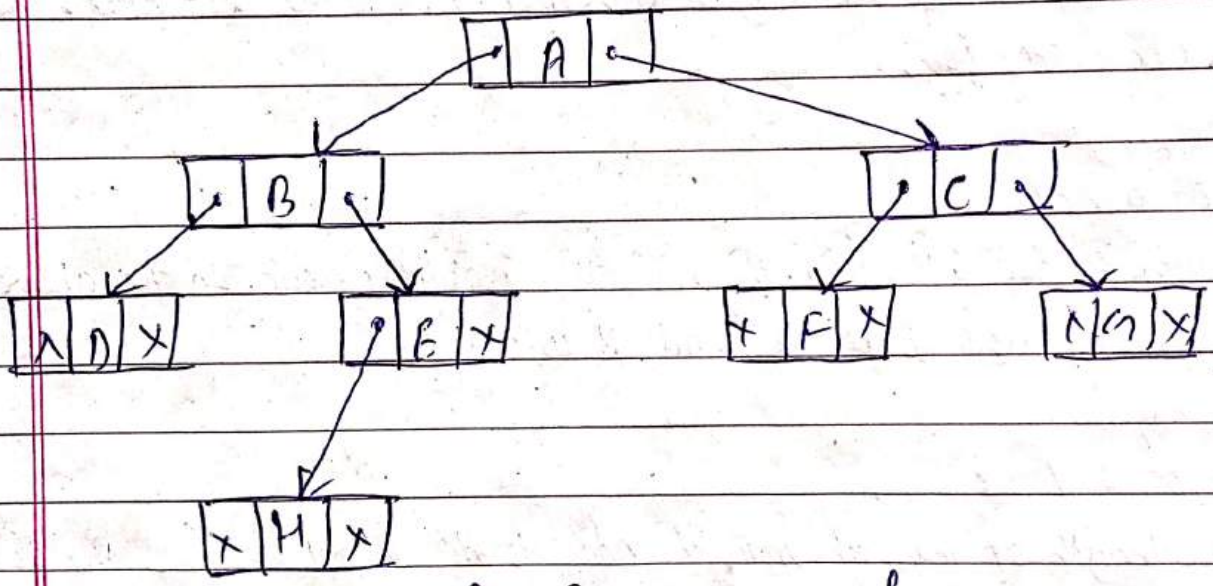


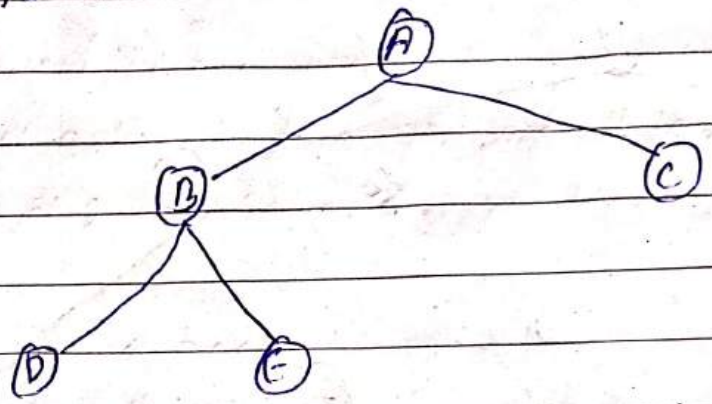
Fig:- Representation of Binary Tree

Types of Binary Tree

a) Strictly Binary Tree

It is a type of Binary tree in which all the nodes ~~contain at most 2 child node~~ can have 0 or two child nodes. It can never have one child.

Example:-

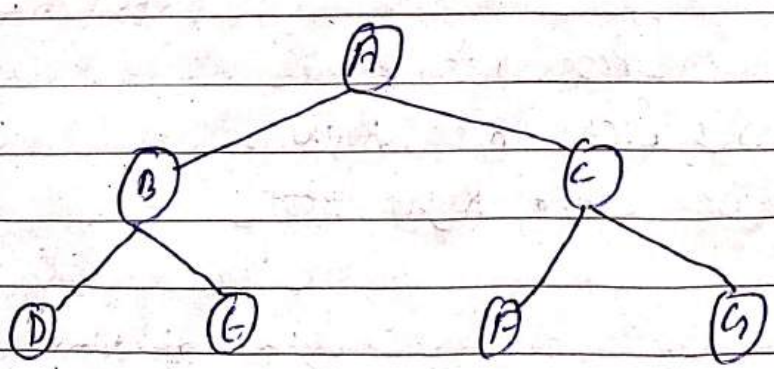


∴ Fig:- Strictly Binary Tree

b) Complete Binary Tree

A binary tree is said to be complete if all the nodes till the second last level from the root contains two child nodes and the nodes at the last level doesn't contain any child node.

Example:



∴ Fig:- Complete Binary Tree

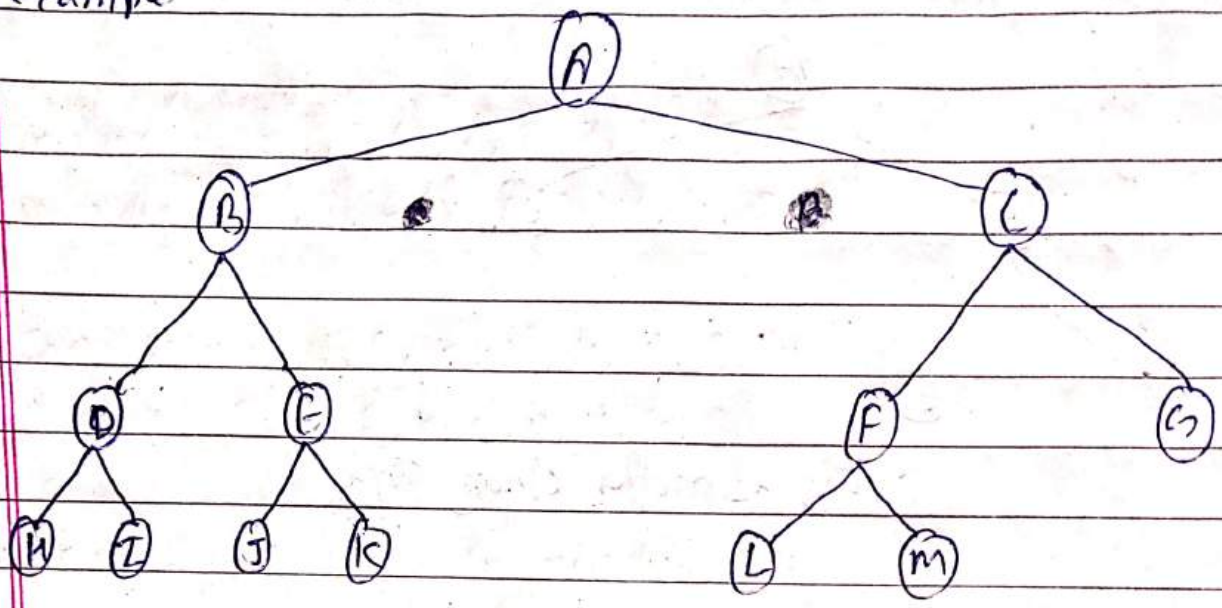
c) Almost Complete Binary Tree

A binary tree is said to be almost complete if all the nodes till ~~second last node~~ the third last level from the root contains two child nodes and the nodes at the second last level can have either 0 or 1 or 2 child nodes however the nodes in the last level must be as far left as possible. There shouldn't be any missing nodes in between them.

[extended binary tree]

Date
Page

Example

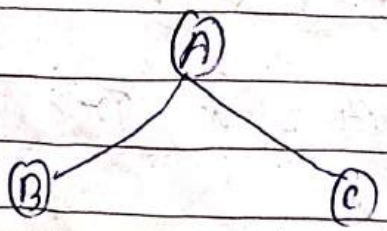


∴ Fig :- Almost Complete Binary tree

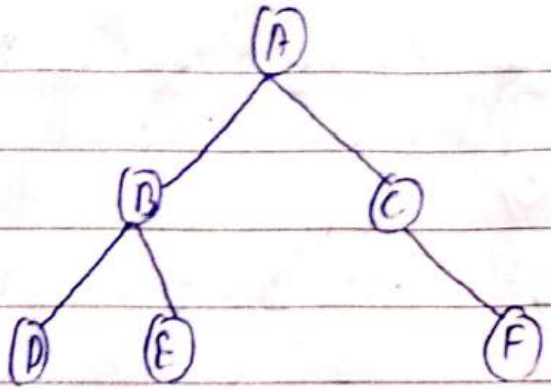
Traversal in Binary Tree

a) preorder Traversal [Root, left, Right]
b) Inorder Traversal [left, Root, Right]
c) postorder Traversal [left, Right, Root]

Example



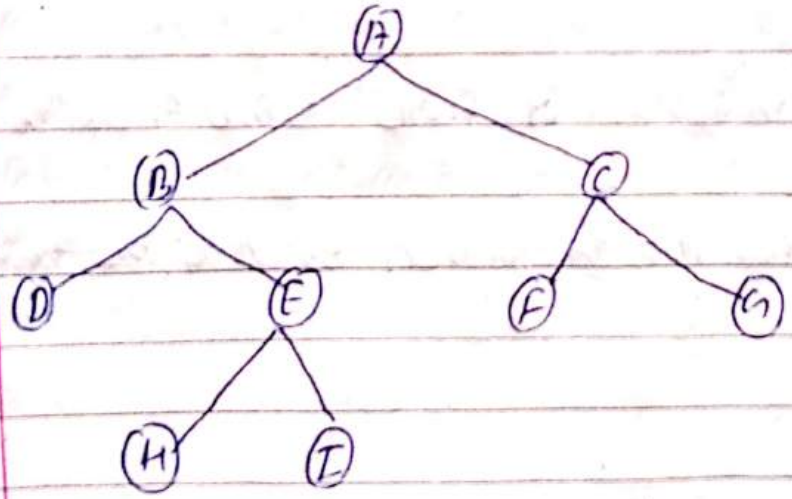
∴ preorder Traversal :- A, B, C
∴ Inorder Traversal :- B, A, C
∴ postorder Traversal :- B, C, A



∴ preorder Traversal :- A, B, D, E, C, F

∴ Inorder Traversal :- D, B, E, A, C, F

∴ postorder Traversal :- D, E, B, F, C, A



∴ preorder Traversal :- A, B, D, E, H, I, C, F, G

∴ Inorder Traversal :- D, B, H, E, I, A, F, C, G

∴ postorder Traversal :- D, H, I, E, B, F, G, C, A

Construct the Binary tree with 9 nodes having following traversal:

Inorder :- E, A, C, K, F, H, D, B, G
preorder :- F, A, E, K, C, D, H, G, B

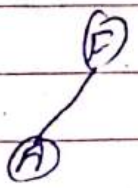
Inorder :- E, A, C, K, F, H, D, B, G
preorder :- F, A, E, K, C, D, H, G, B

Soln

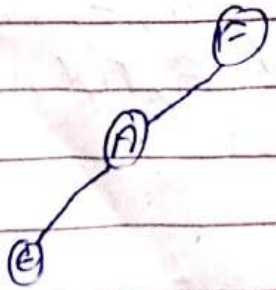
i) Here, F comes first in preorder, so, it is the root of the tree

ii) From Inorder, E, A, C, K are in the left of F and H, D, B, G are in the right of F.

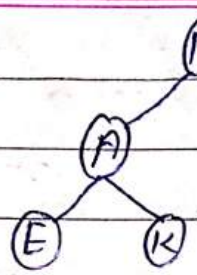
iii) Among E, A, C, K, A comes first in preorder so A is the immediate left child of F.



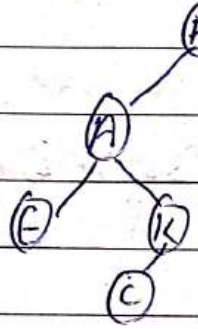
iv) From Inorder, E is the left child of A and C, K are in the right of A.



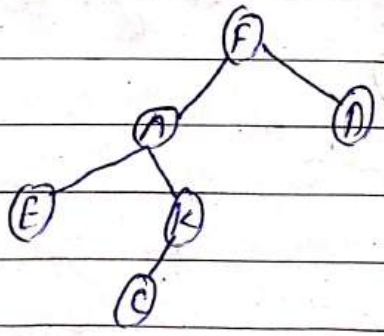
v) Among, C, K, K comes first in pre-order So, K is the immediate right child of A



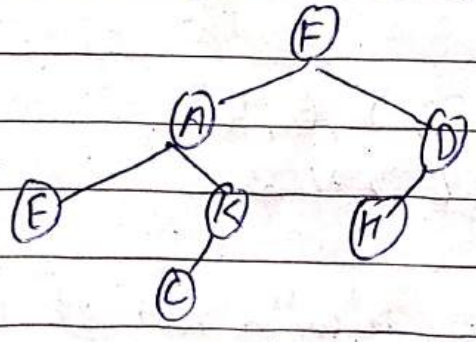
vi) From inorder, C is the left child of K.



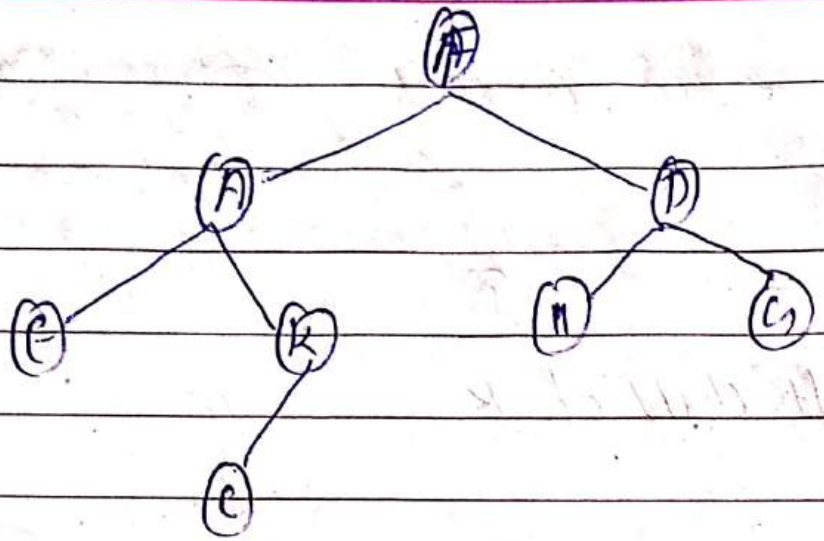
vii) Among, H, D, B, G, D comes first in preorder. So, D is the immediate right child of F.



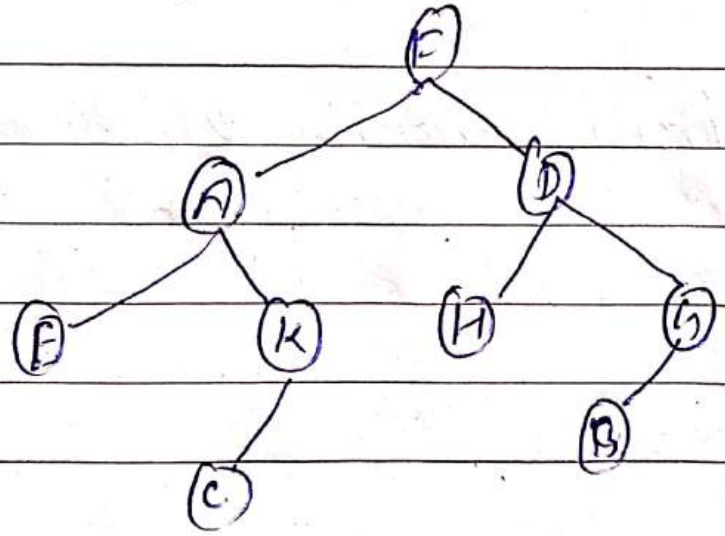
viii) From Inorder, H is in the left of D and B, G are in the right of D. So, H is immediate left child of D



ix) Among B and G, G comes first in preorder. So, it is the immediate right child of D.



*) From in order, B is in the left of G. So, it is the immediate left child of G.



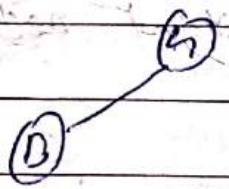
Construct the binary tree with 12 nodes having following traversals.

preorder :- G, B, Q, A, C, K, F, P, D, E, R, H

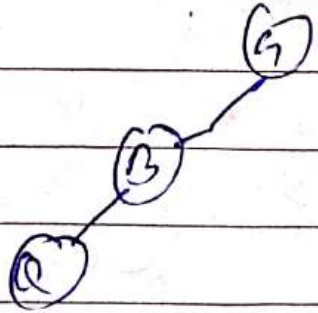
Inorder :- Q, B, K, C, F, A, G, P, E, D, H, R

i) Here, G comes first in preorder. So, G is the root node of tree

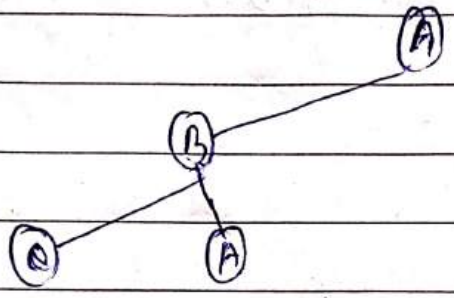
ii) From In order, Q, B, K, C, F are the left node of G and P, E, D, H, R are the right node. Among, Q, B, K, C, F, B comes first in preorder. So, B is the left immediate child of G.



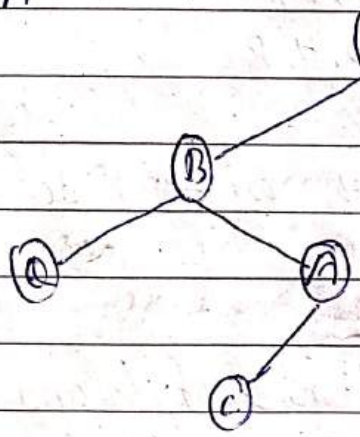
iii) From Inorder, Q is the left child of B and K, C, F, A are right child of B.



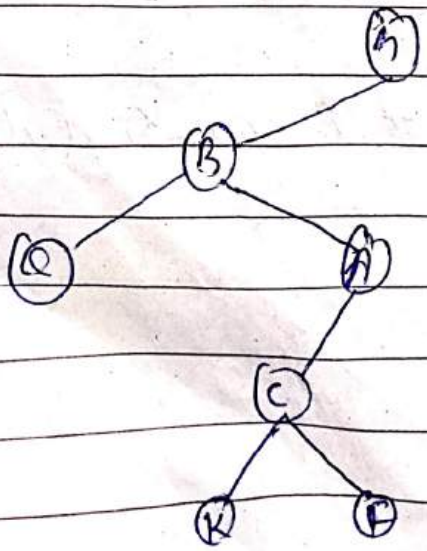
iv) Among K, C, F, A, A comes first in preorder, so, A is the ~~left~~ right immediate node of B.



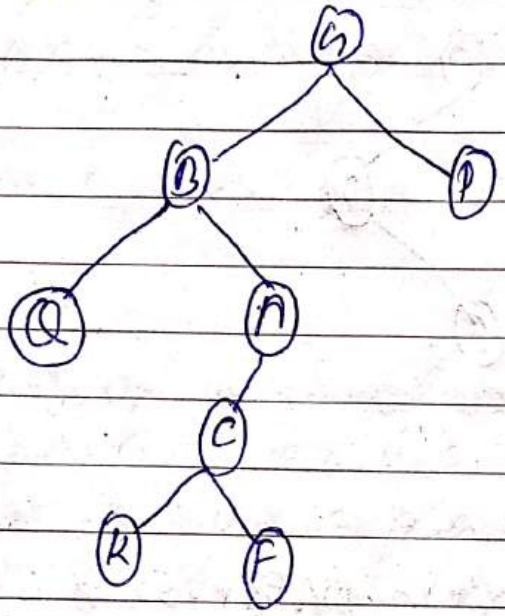
v) Among K, C, F, C comes first in preorder. So, C is the left element of A



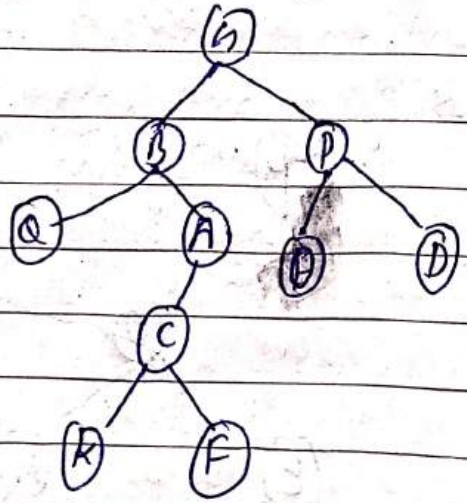
vi) Among K and F, K comes first in Inorder and F comes after K. So, K is the left node and F is the right node of C.



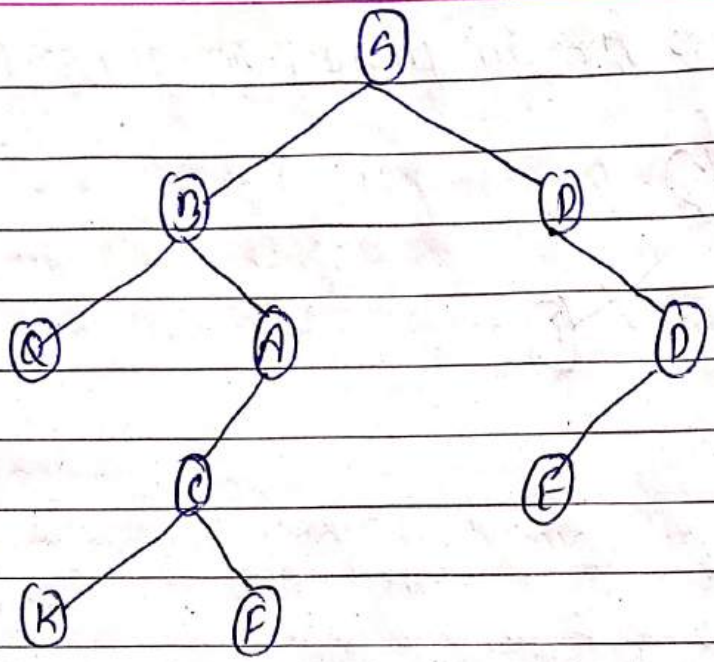
vii) Among D, C, D, H, R, P comes first in preorder. So, P is the right immediate node of root G.



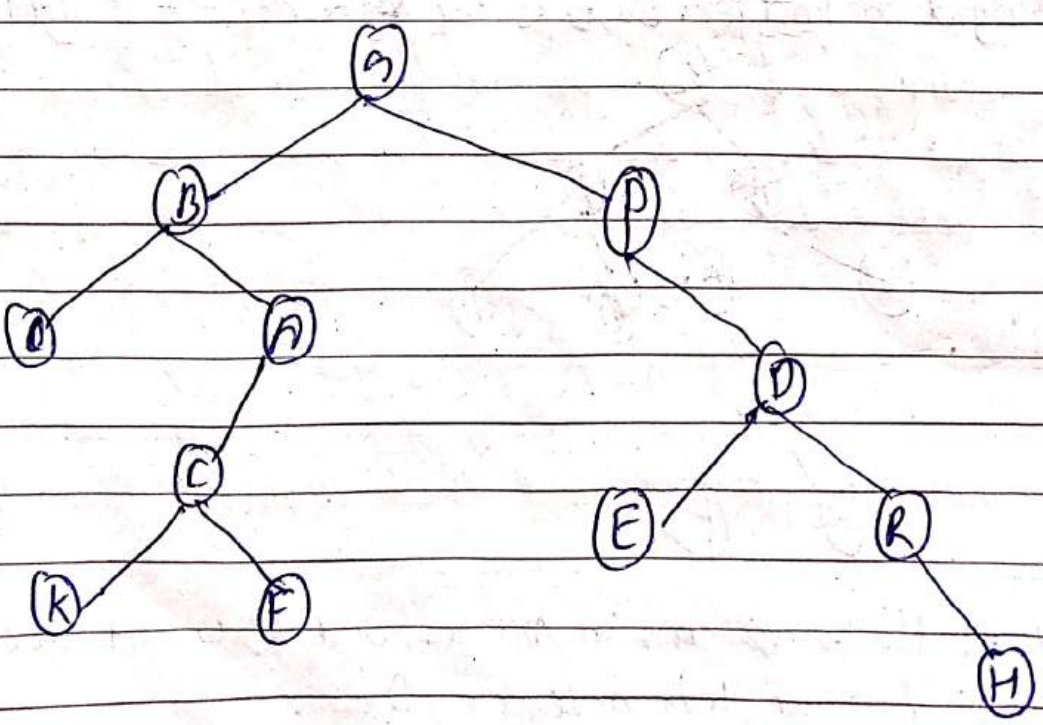
viii) Among E, D, H, R, D comes first in preorder. D is ~~left~~ right in Inorder ~~and H, R are right in inorder~~. So, D is the ~~right node of D~~ left. right node of P.



ix) Among E, H, R, E comes first in preorder and E is left side of D in inorder. So, E is the left node of D.



x) Among H and R, R comes first in preorder. So, R is the right node of D and H comes right part in Inorder of R. So, H is the right node of R.



a) Pre-order Traversal

Steps

1. process the root (R)
2. Traverse the left subtree of R in preorder.
3. Traverse the right subtree of R in preorder.

Algorithm

The following algorithm traverses the binary tree (T) in preorder using STACK

1. [Initially PUSH NULL onto STACK and initialize PTR]
   Set TOS = 0, STACK[0] = NULL, PTR = root
2. Repeat steps 3 to 5 while PTR ≠ NULL
3. Traverse PTR
4. [Right child ?]
   if PTR[RIGHT] ≠ NULL, then PUSH it onto STACK
   Set TOS = TOS + 1 and STACK[TOS] = PTR[RIGHT]
5. [left child ?]
   if PTR[LEFT] ≠ NULL, then
   set PTR = PTR[LEFT]
   else [pop from STACK]
   Set PTR = STACK[TOS] and TOS = TOS - 1
6. Exit

b) Inorder Traversal

Steps

① Traverse the left subtree of R in Inorder
② Process the root (R).
③ Traverse the right subtree of R in Inorder.

Algorithm

The following algorithm traverses the binary tree (T) in inorder using stack.

① [Initially PUSH NULL onto STACK and initialize PTR]
Set TOS = 0, STACK[0] = NULL, PTR = root
② Repeat while PTR ≠ NULL [pushes the leftmost path onto STACK]
a) Set TOS = TOS + 1 and STACK[TOS] = PTR
b) Set PTR = PTR[LEFT]
③ Set PTR = STACK[TOS] and TOS = TOS - 1
④ Repeat steps 5 to 7 while PTR ≠ NULL
⑤ Traverse PTR
⑥ [Right child ?]
if PTR[RIGHT] ≠ NULL, then
a) Set PTR = PTR[RIGHT]
b) Goto Step 4
⑦ Set PTR = STACK[TOS] and TOS = TOS - 1
⑧ Exit

c) Postorder Traversal

Steps

1. Traverse the leftsubtree of R in postorder
2. Traverse the rightsubtree of R in postorder
3. process the root (R)

Algorithm

The following algorithm traverse the binary tree (T) in postorder using STACK

1. [Initially PUSH NULL onto STACK and initialize PTR]
   Set TOS = 0, STACK[0] = NULL and PTR = root
2. [PUSH leftmost path onto stack]
   Repeat steps 3 to 5 while PTR ≠ NULL
3. Set TOS = TOS + 1 and STACK[TOS] = PTR
4. if RIGHT[PTR] ≠ NULL then PUSH onto STACK
   Set TOS = TOS + 1 and STACK[TOS] = −RIGHT[PTR]
5. Set PTR = LEFT[PTR]
6. Set PTR = STACK[TOS] and TOS = TOS − 1
7. Repeat While PTR > 0
   a) Traverse PTR
   b) Set PTR = STACK[TOS] and TOS = TOS − 1
8. if PTR < 0
   a) PTR = −PTR
   b) Goto step 2
9. Exit

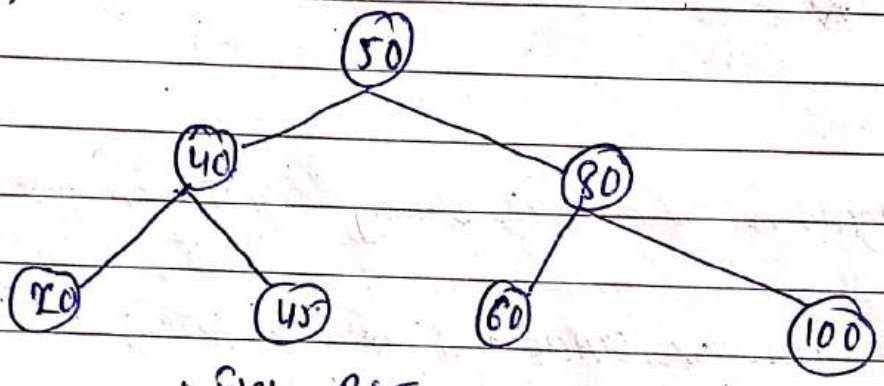
3. Binary Search Tree (BST)

A BST is a binary tree in which:-

a) The left child is always smaller than the root node.

b) The right child is always greater than the root node.

Example:-



∴ Fig:- BST

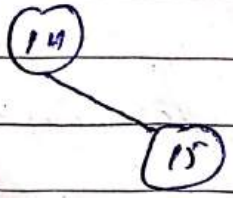
Construct BST using following data.

14, 15, 1, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9

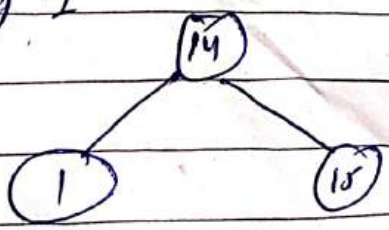
Soln

(1) Inserting 14

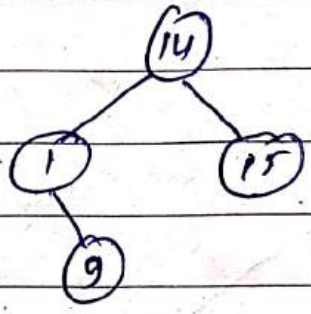
(2) Inserting 15



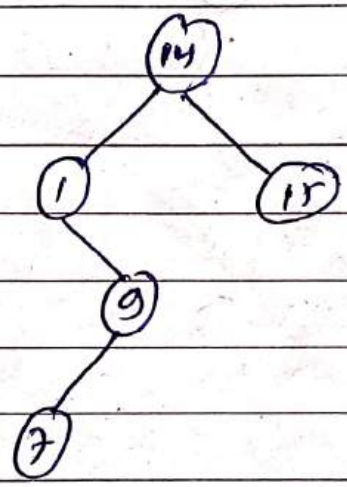
(3) Inserting 1



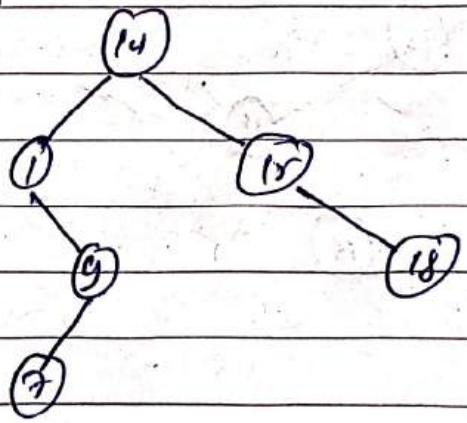
(4) Inserting 9



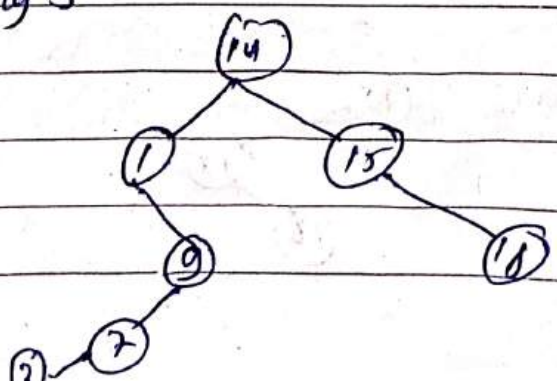
(5) Inserting 7



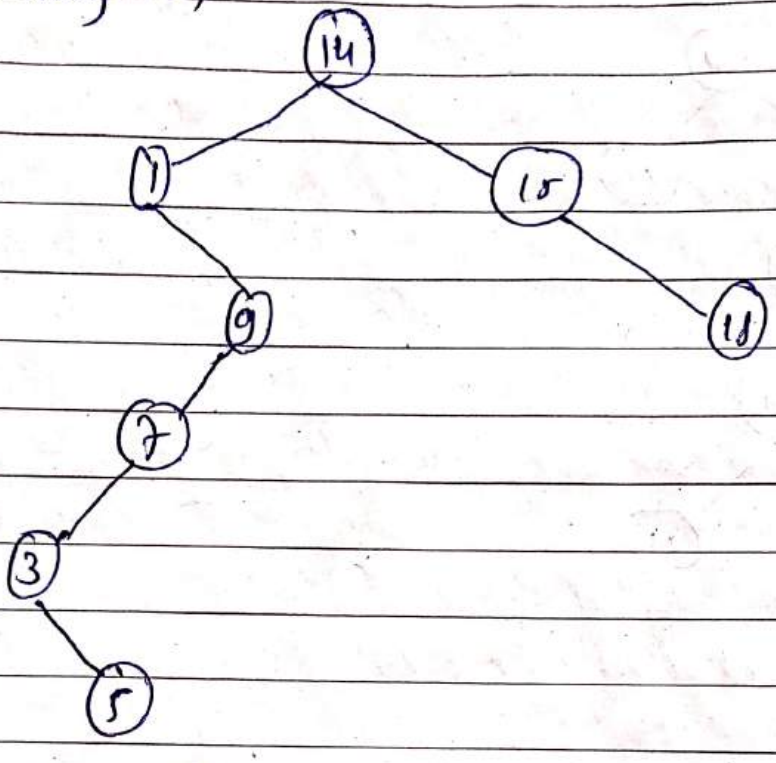
(6) Inserting 18



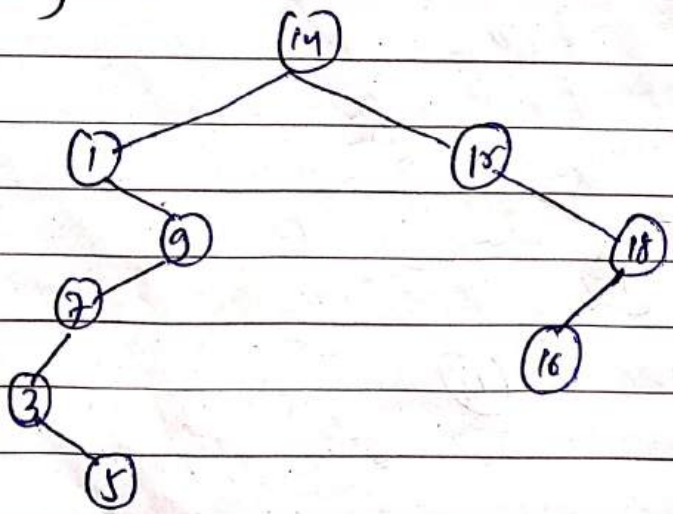
(7) Inserting 3



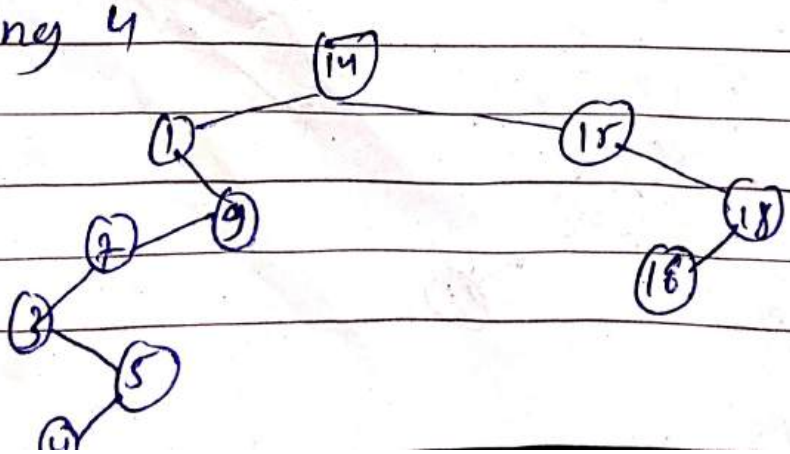
(8) Inserting 5,



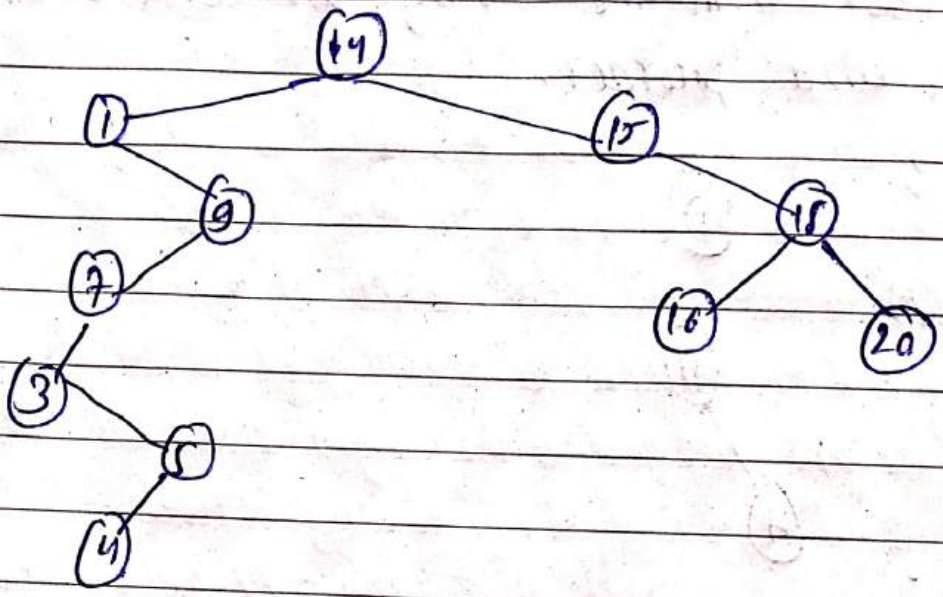
(9) Inserting 16



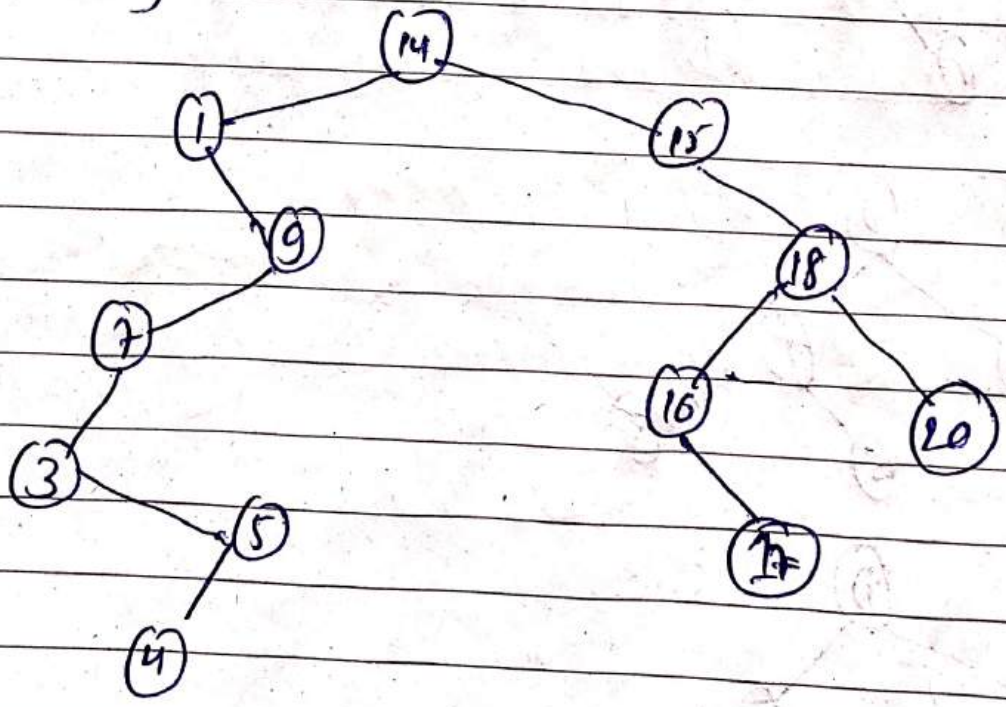
(10) Inserting 4



(11) Inserting 20



(12) Inserting 17

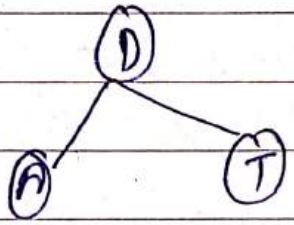


Draw a BST from the String DATASTRUCTURE and traverse in postorder and preorder.

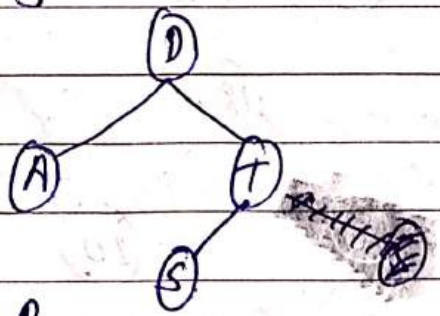
i) Inserting D

ii) Inserting A

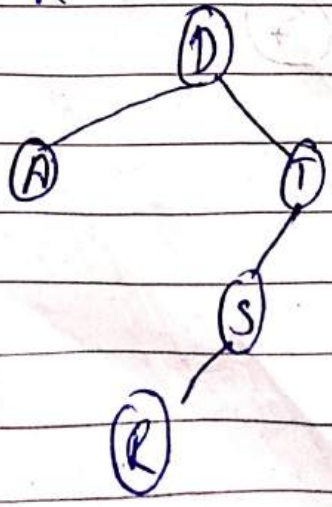
iii) Inserting T



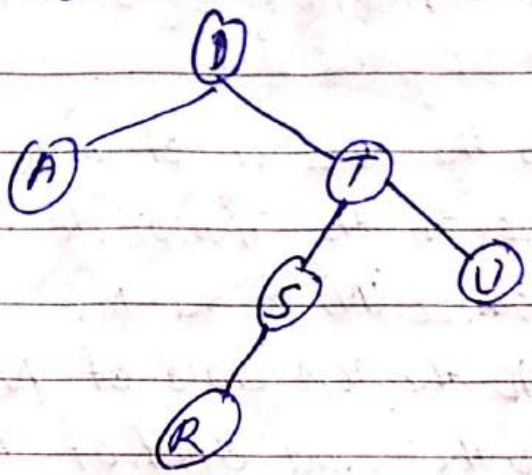
iv) Inserting S



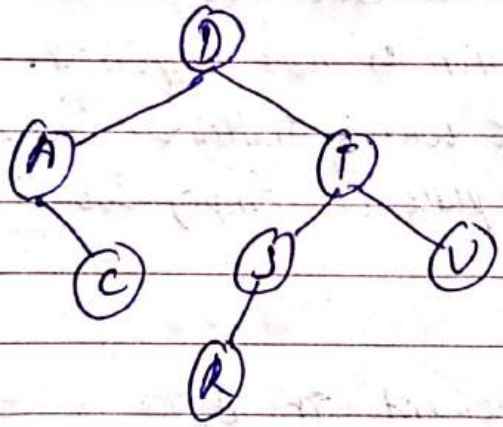
v) Inserting R



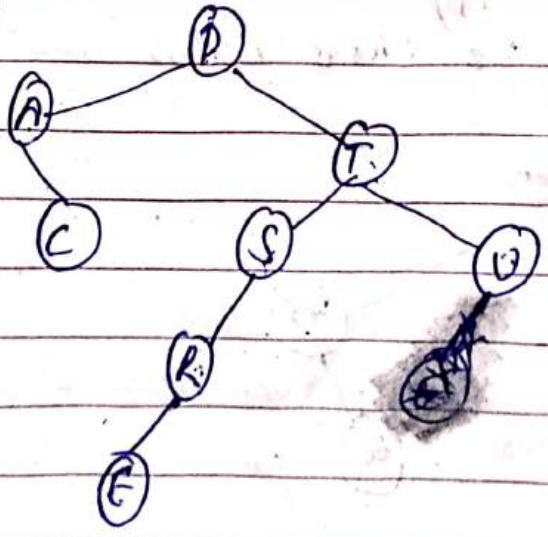
vi) Inserting U



vii) Inserting C



viii) Inserting E



∴ postORDER = C, A, E, R, S, U, T, D

∴ Preorder :- D, A, C, T, S, R, E, U

Algorithm for Searching and Inserting element in BST

The following algorithm finds the location of data in the BST or inserts data as a new node in its appropriate place in the tree.

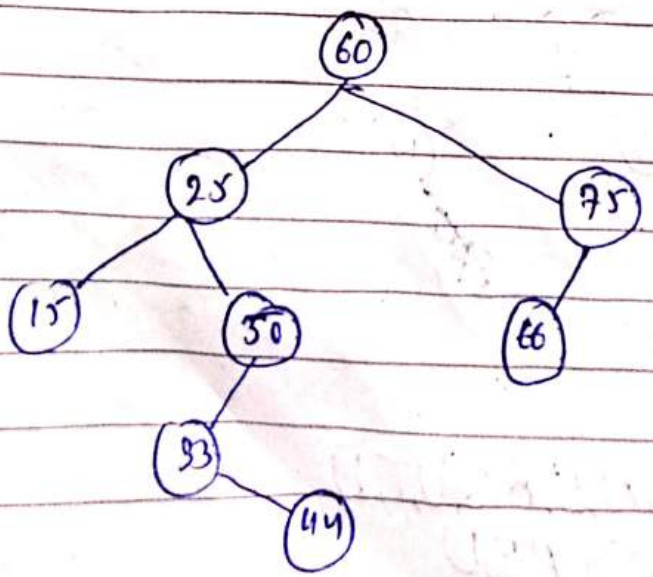
1) Compare the data with the root node (N) of the tree
   a) if data < N, proceed to the left child of N.
   b) if data > N, proceed to the right child of N.
2) Repeat Step 1 until one of the following occurs
   a) A node N is met such that data = N, in this case the search is successful and insertion is not required.
   b) An empty subtree is met which indicates search is unsuccessful and insert data in the place of empty subtree.
3) Exit

Algorithm for Deleting Element in BST

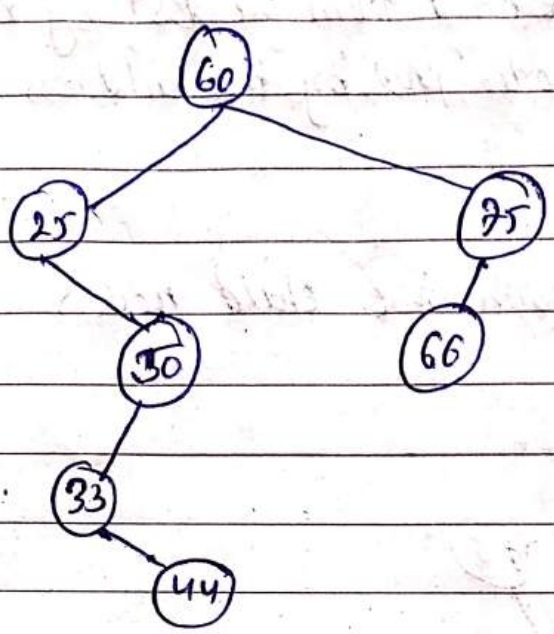
Deletion in BST can have three cases:-

a) Deleting the node with no child nodes

Example:-



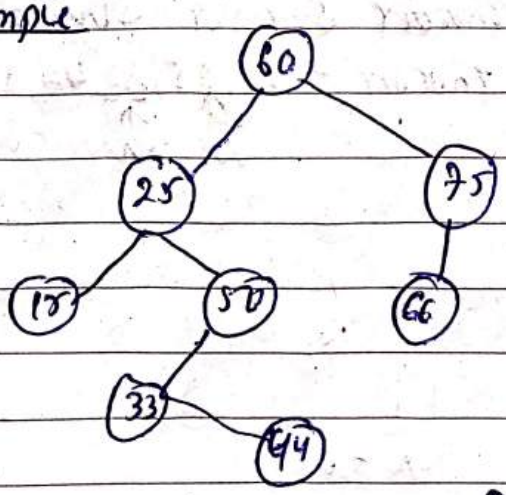
Deleting 15.



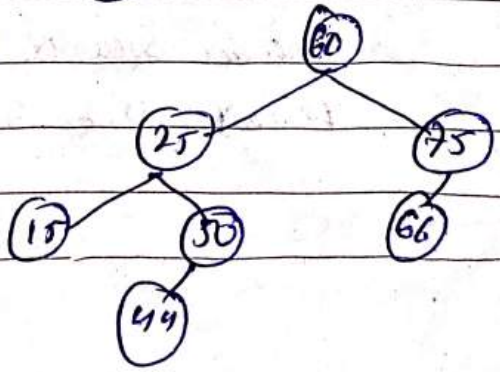
The node (N) can be deleted from BST by replacing the address of N in its parent node (PN) by NULL value.

b) Deleting the node with Single Child node

Example



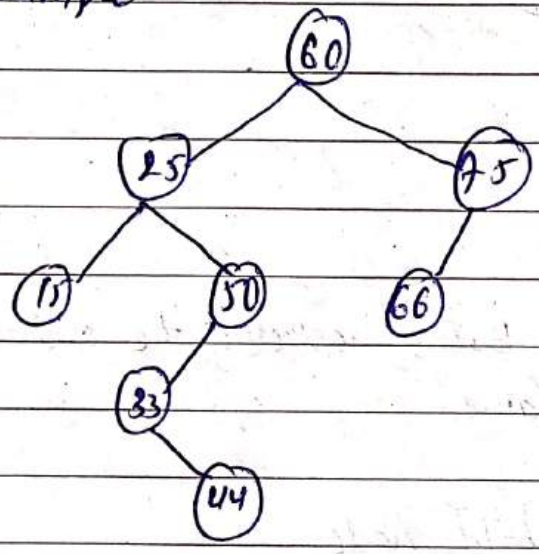
Deleting 33



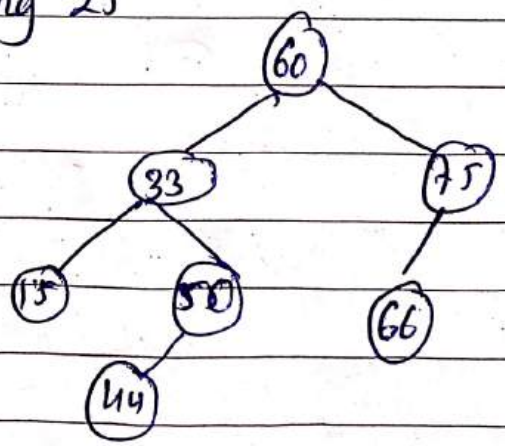
The node (N) can be deleted from the BST by replacing the location of N in its parent Node (PN) by the address of its only child node.

c) Deleting the node with two child nodes

Example



Deleting 25



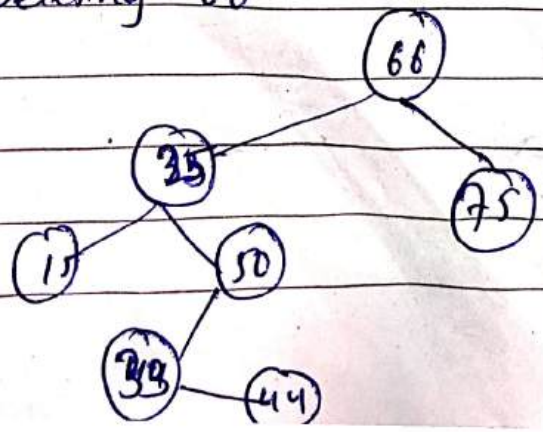
N = 25

Inorder Successor S(N) = 33

Inorder = 15, 25, 33, 44, 50, 60, 66, 75

N S(N)

Deleting 60



N = 60

Inorder Successor S(N) = 66

Inorder = 15, 25, 33, 44, 50, 60, 66, 75

N SN

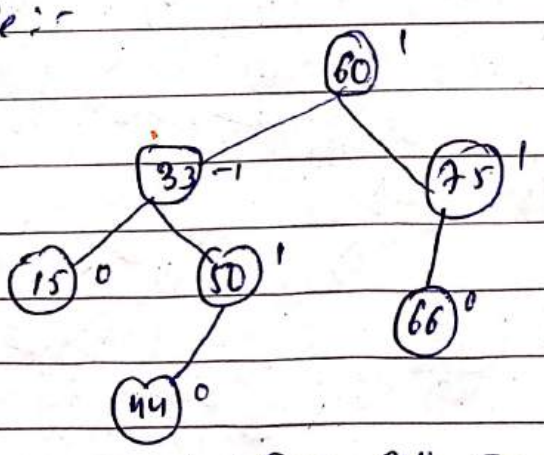
Steps:-

a) Find the inorder successor (SN) of the node to be deleted (N).
b) Delete SN from the BST using either case I or case II.
c) Replace the node N by SN

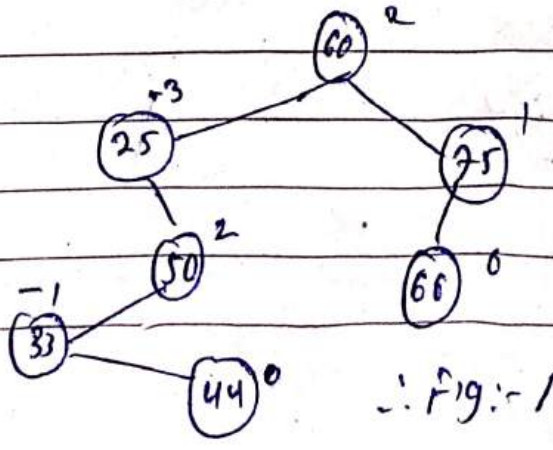
4. AVL Tree

It is an almost height balanced tree. It may or maynot be perfectly height balanced. It has the property that the height of left subtree is either equal, or one more, or one less than the height of right subtree. The balanced factor (BF) can be used to determine for each node using Balance Factor (BF) = Height of left subtree − Height of right subtree. To be an AVL Tree, the balance factor of each node must be either 0, 1, or −1.

Example:-



∴ Fig:- AVL Tree



∴ Fig:- Not AVL Tree

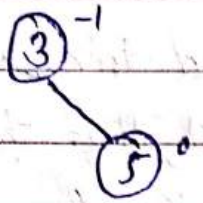
Create an AVL tree by using following data.

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

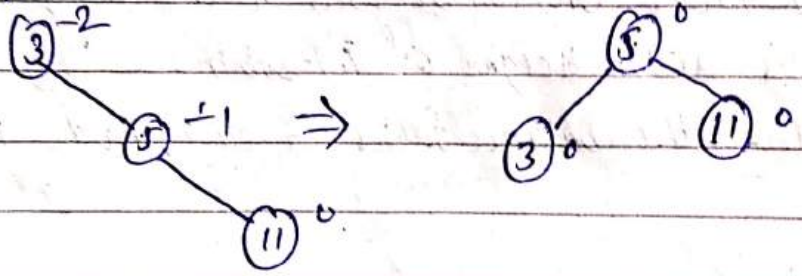
Soln

i) Inserting 3

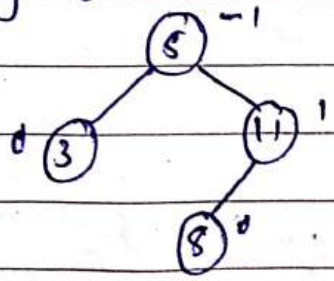
ii) Inserting 5



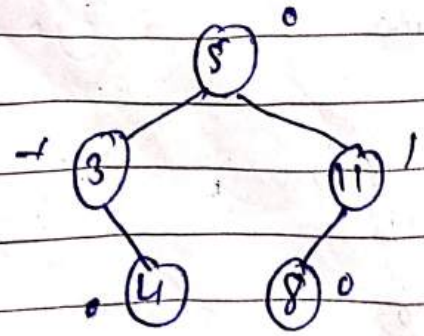
iii) Inserting 11



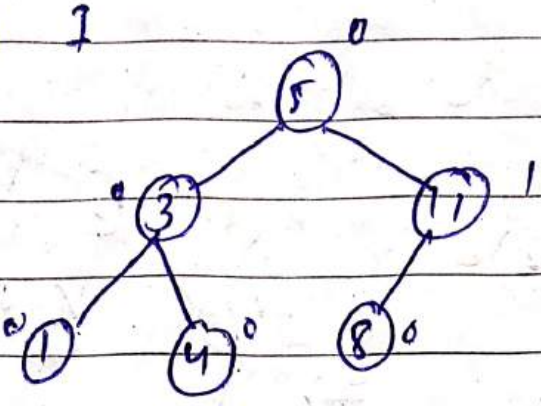
iv) Inserting 8



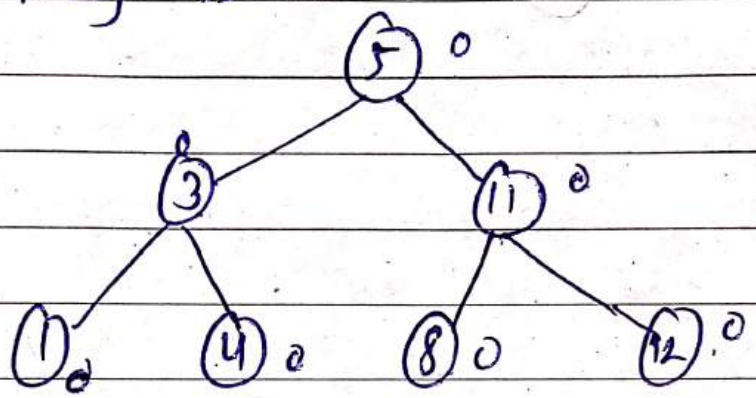
v) Inserting 4



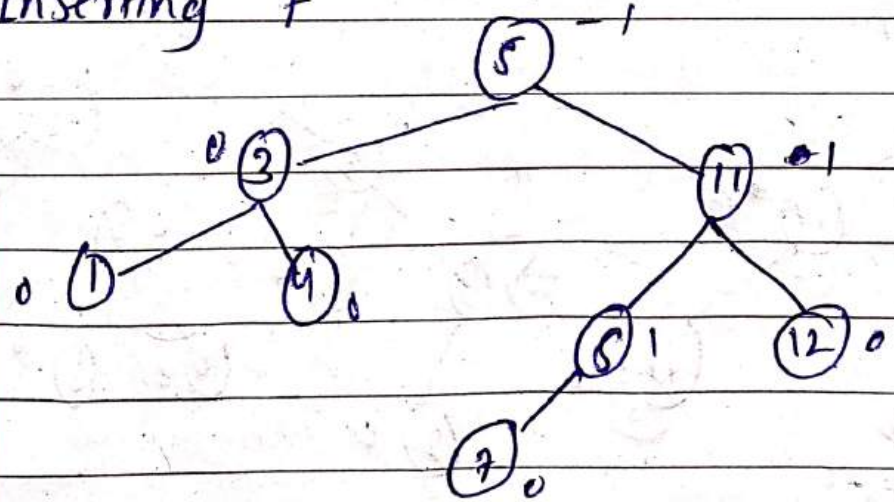
vi) Inserting 1



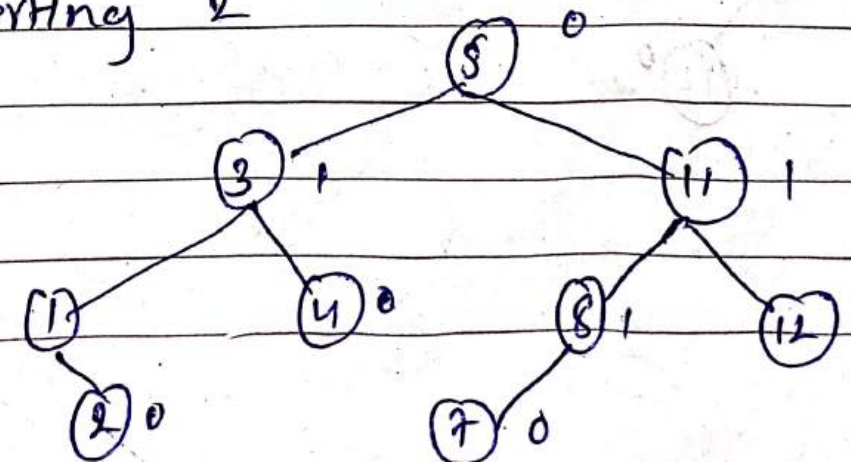
vii) Inserting 12



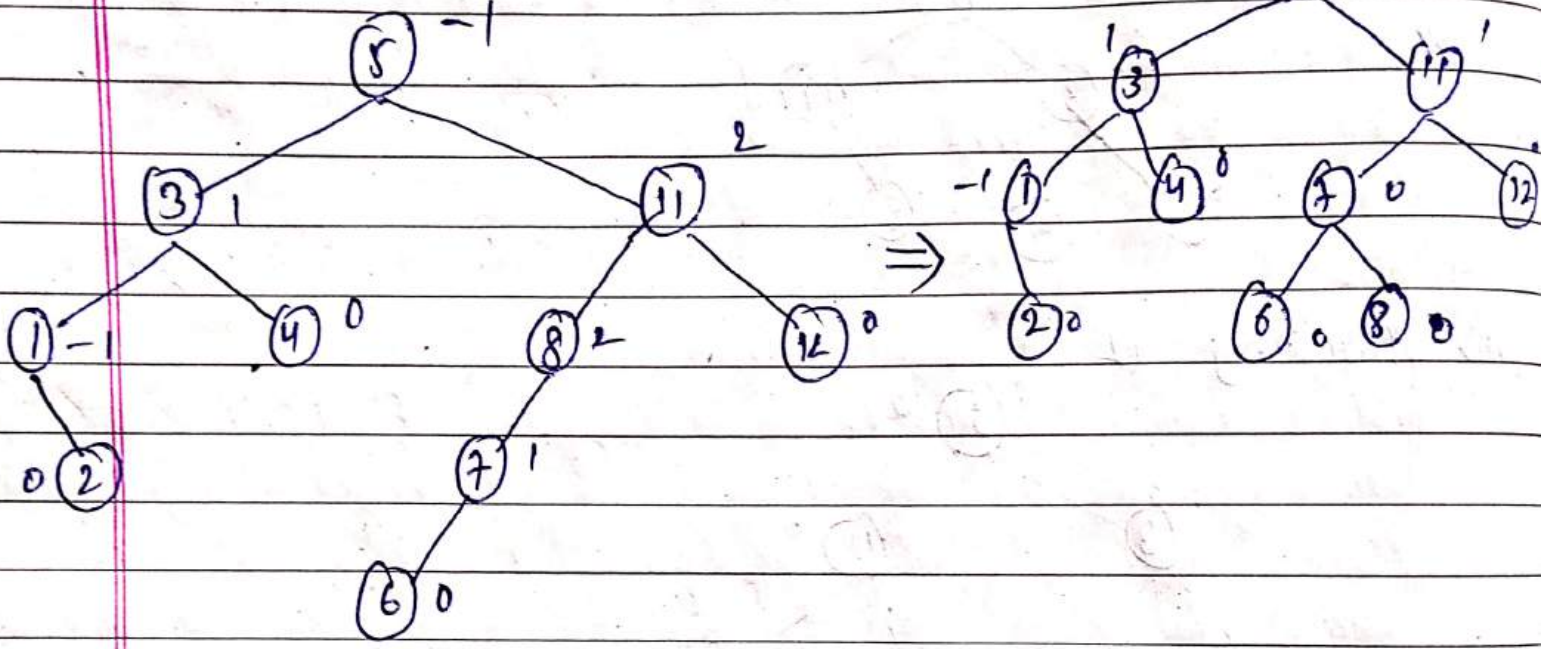
viii) Inserting 7



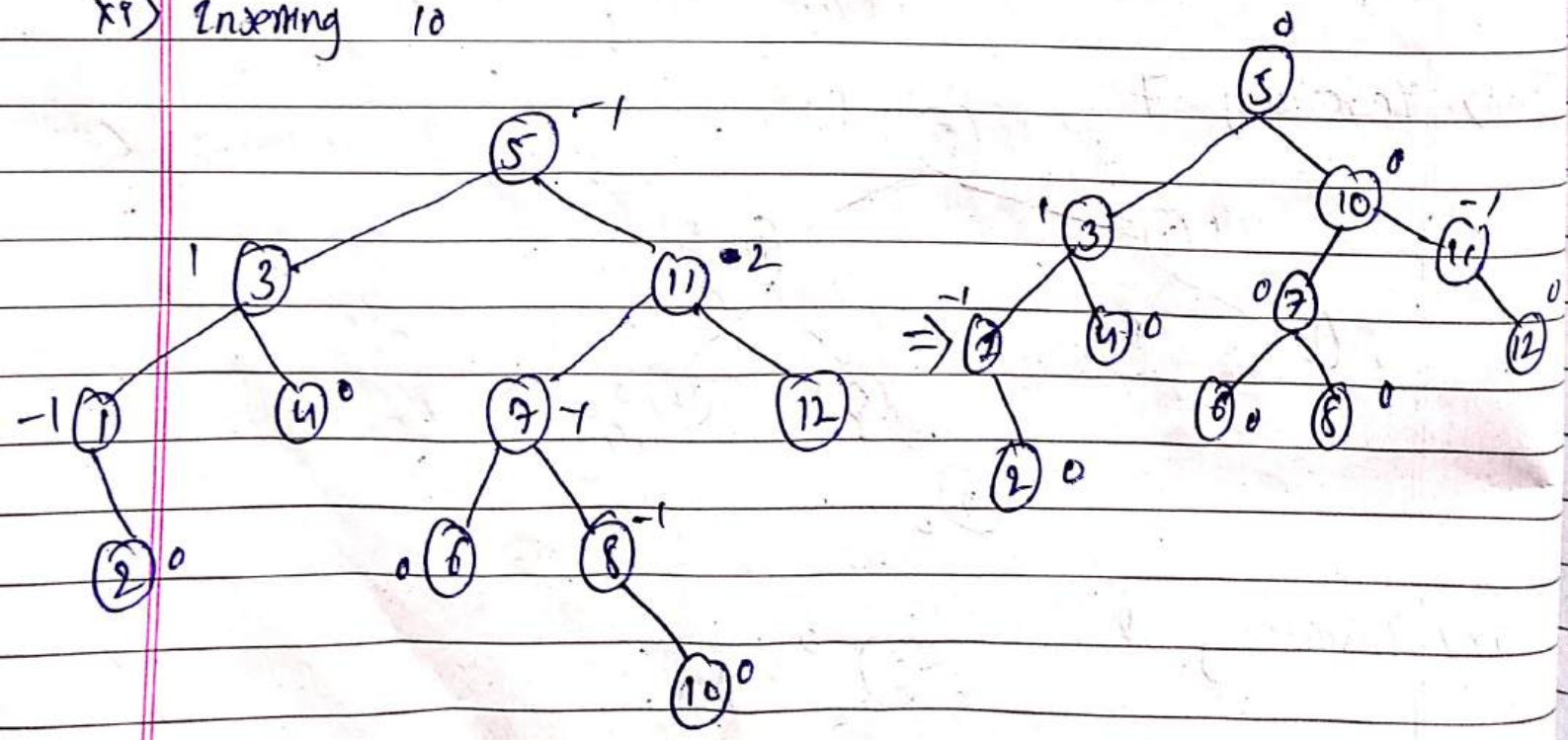
ix) Inserting 2



x) Inserting 6



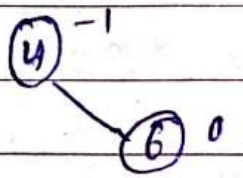
xi) Inserting 10



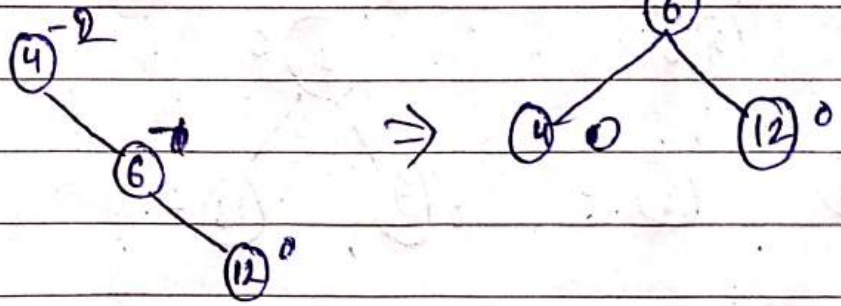
Construct AVL tree using
4, 6, 12, 9, 5, 2, 13, 8, 3, 7, 11

i) Inserting 4

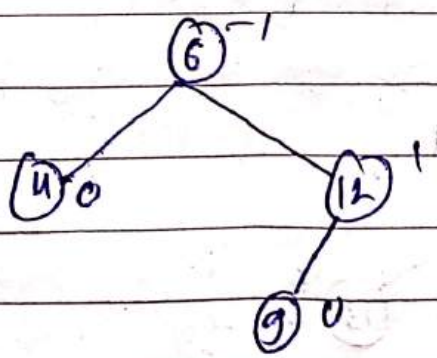
ii) Inserting 6



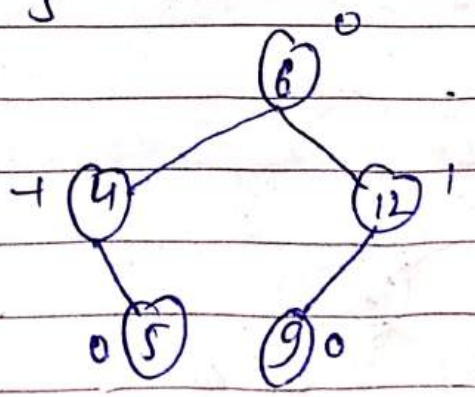
iii) Inserting 12



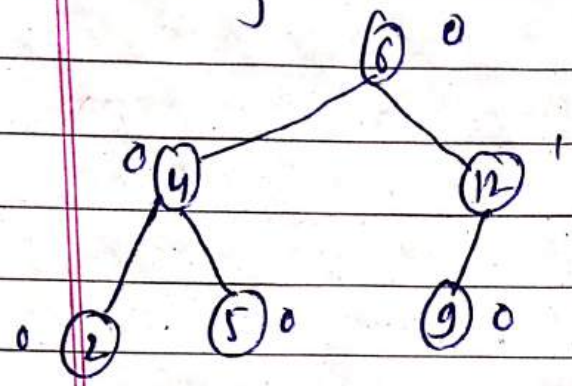
iv) Inserting 9



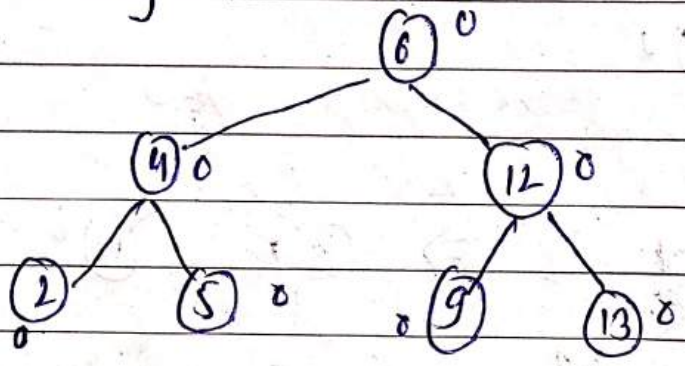
v) Inserting 5



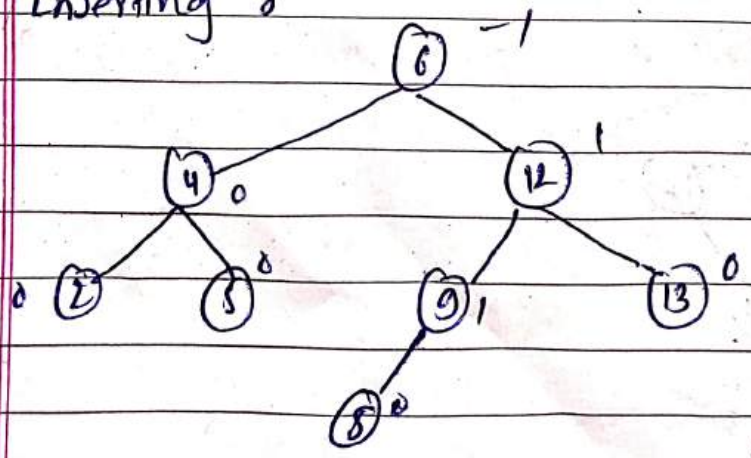
vi) Inserting 2



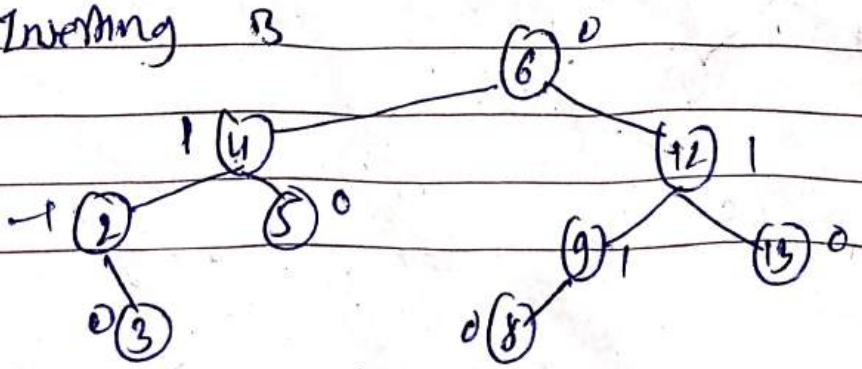
vii) Inserting 13



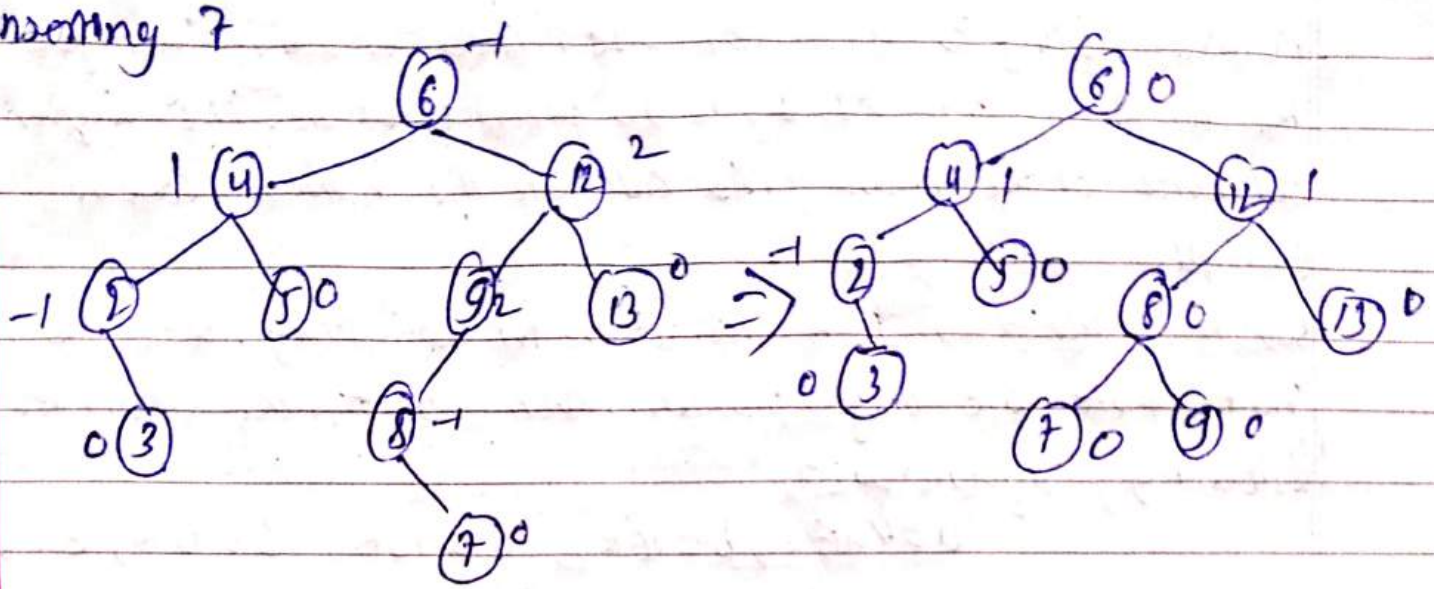
viii) Inserting 8



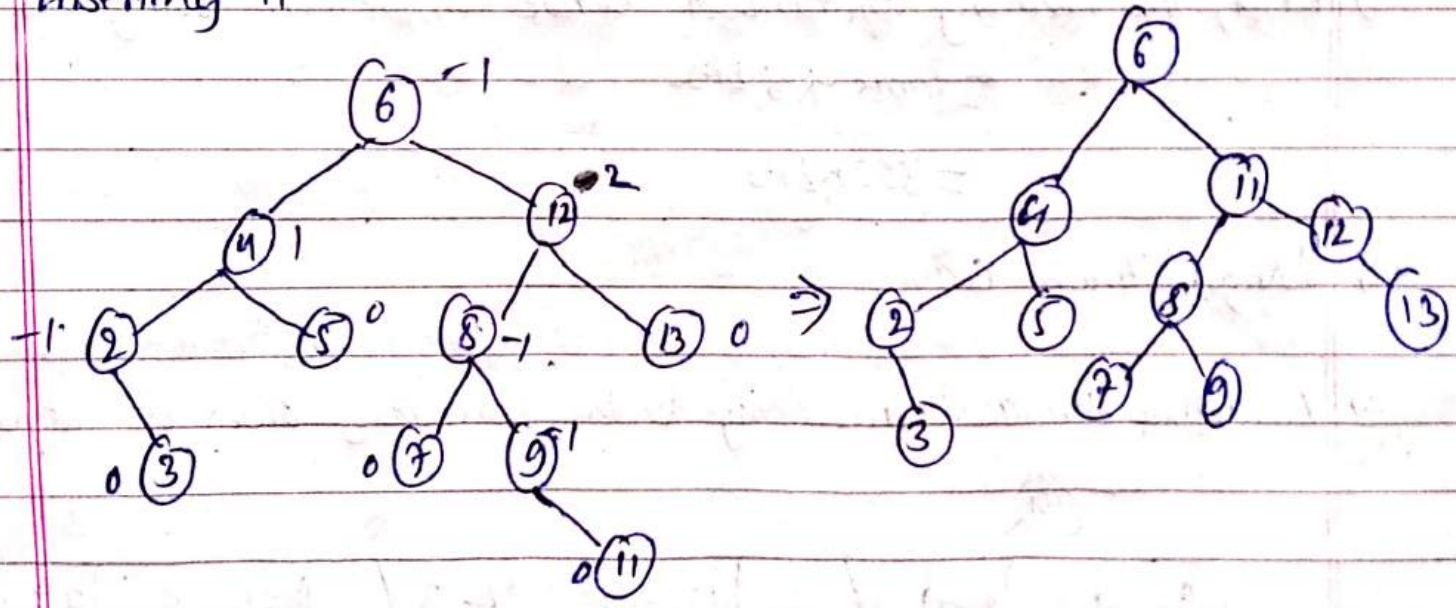
ix) Inserting 3



x) Inserting 7



xi) Inserting 11



5. Huffman coding Algorithm

Huffman coding is a lossless data compression technique. Compression is required to :-

i) Minimize the memory requirement to store data.

ii) Faster the data transmission speed through the network.

Huffman coding is a variable sized data encoding technique. It has the compression rate of 20 to 90 percent. It uses the frequency of occurance of the characters to calculate the huffman codes.

Example:-

Suppose, A file has 100k characters. For simplicity, let us assumed that there are only charactors from a-f. The characters have following frequencies:-

a=45K , b=16K , c=13K, D=12K, e=9k, f=5

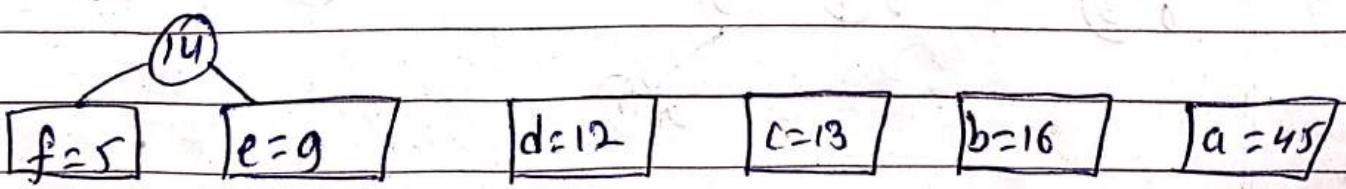
i) First, the memory requirement before compression (using ASCII)

= 100k * 8 bits

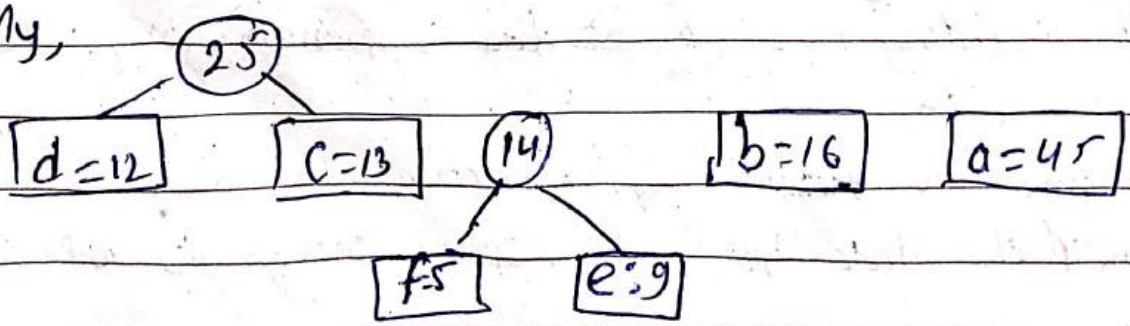
= 800 Kbits

ii) Using Huffman coding

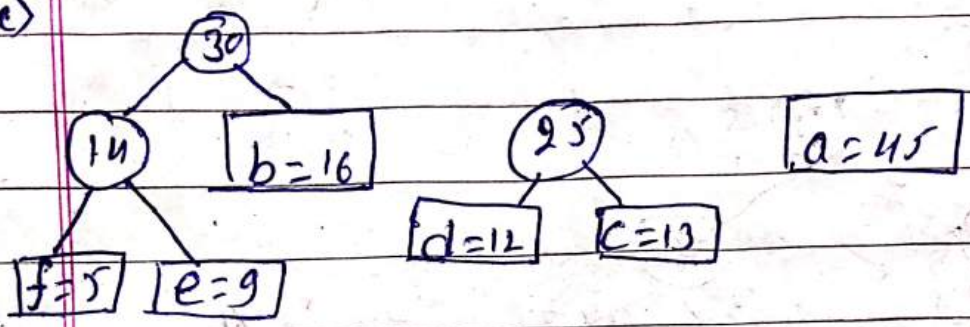
a) Arranging the data according to the ascending order of frequencies



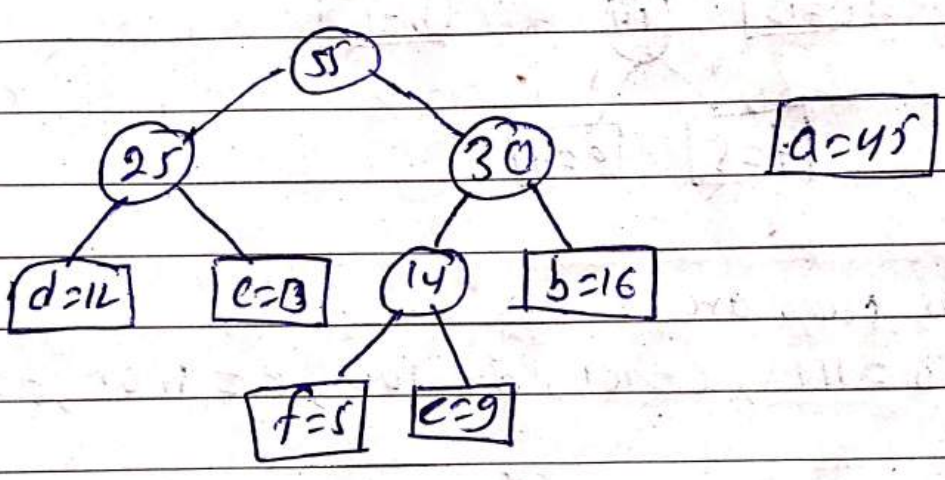
b) Similarly,



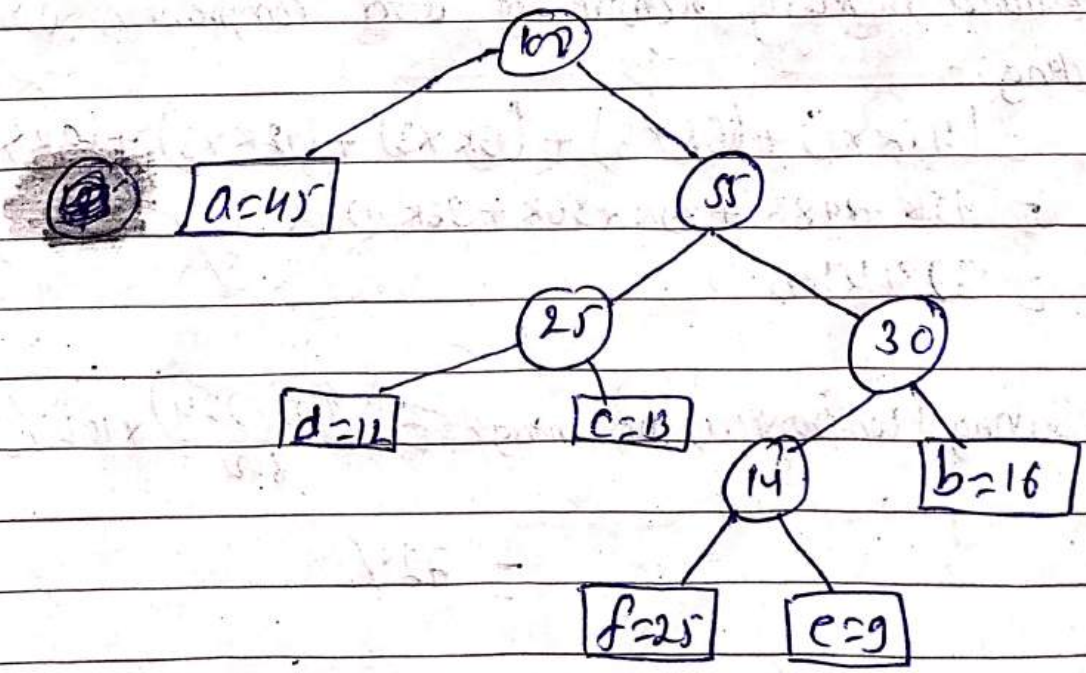
c)



d)

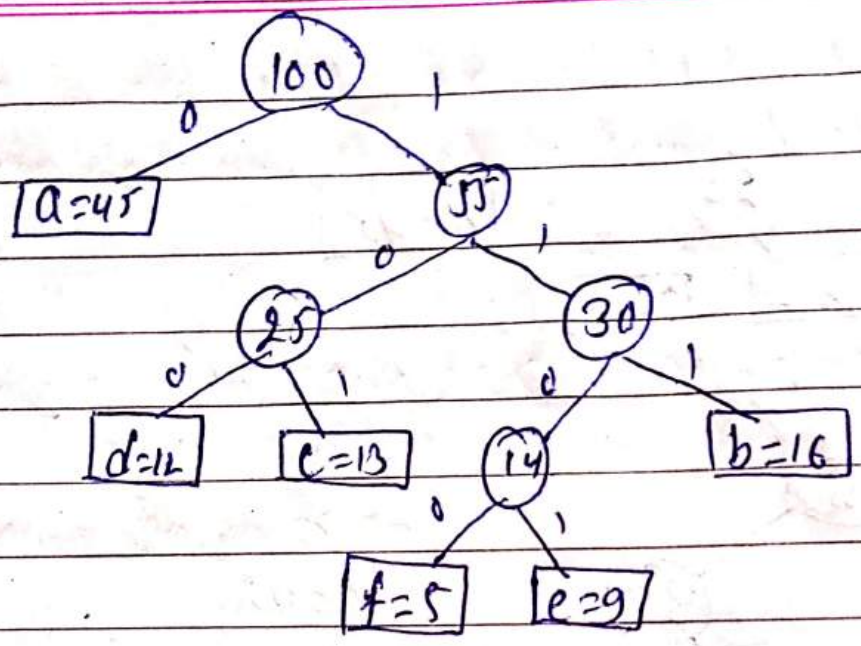


e)



Now, inserting 0 at the left and 1 at the right edge of the tree.

i.e.



∴ The hoffman codes are :-

$a = 0$, $b = 111$, $c = 101$, $d = 100$, $e = 1101$, $f = 1100$

The total memory requirement after compression using Huffman coding :-

$= (45k \times 1) + (16k \times 3) + (13k \times 3) + (12k \times 3) + (9k \times 4) + (5k \times 4)$

$= 45k + 48k + 39k + 36k + 36k + 20k$

$= 224$ kbit

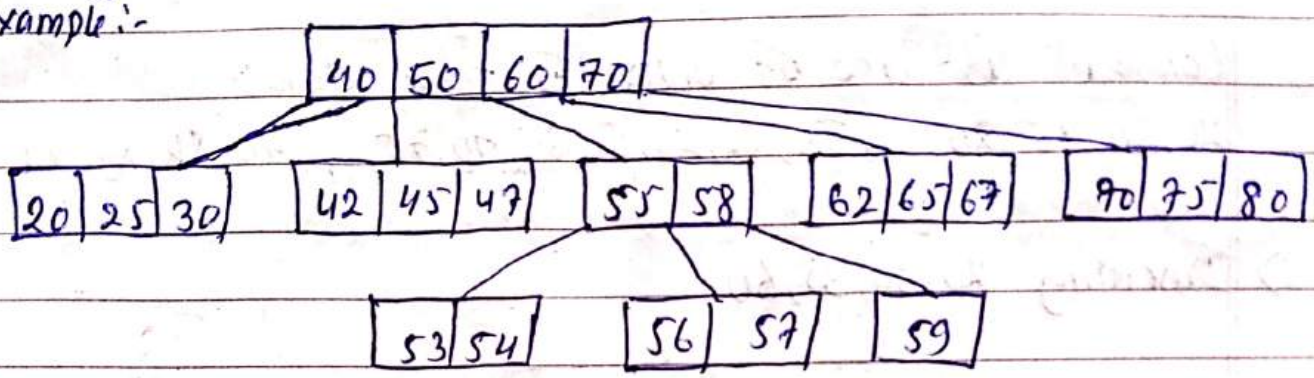
∴ Saving (compression) percentage $= \frac{(800 - 224)}{800} \times 100\%$

$= 72\%$

6. M-Tree

Multiway Tree

Example:-



∴ Fig:- Multiway tree of order 5

A M-Tree is a search tree that can have more than two child nodes. A M-Tree of order 5 can have five child nodes. The elements of each node are arranged in ascending order.

7. B-Tree

A B-Tree is a self balancing tree that maintains sorted data and allows searching, insertion and deletion. It is a specialized M-Tree. It is suitable for storage system that reads and writes large block of data. It is commonly used in database and file system.

properties of B-Tree

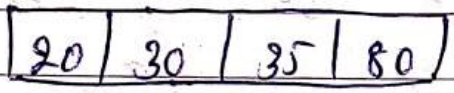
i) Every node in B-Tree contains at most M child nodes, where M is the order of the B-Tree

ii) Every node in the B-Tree except the root node and the leaf node contains at least $\frac{M}{2}$ child nodes.

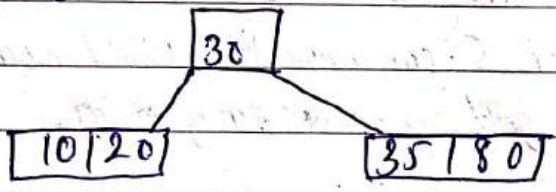
iii) The root node must have at least two child nodes.
iv) The leaf node must be at the same level.

Construct B-Tree of order 5 using data:-
20, 30, 35, 80, 10, 55, 60, 25, 5, 65, 70, 75, 15, 40, 50, 85, 45

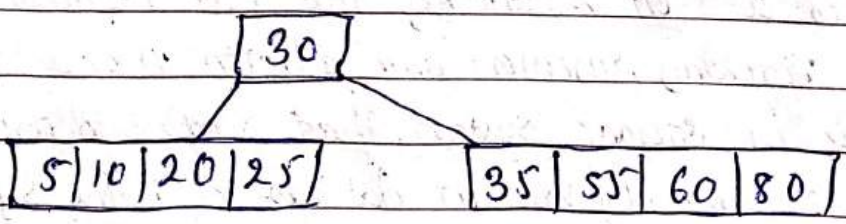
i) Inserting 20, 30, 35, 80



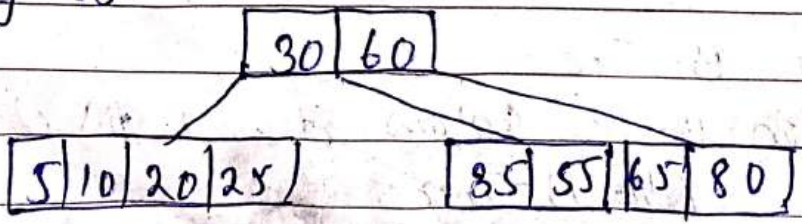
ii) Inserting 10



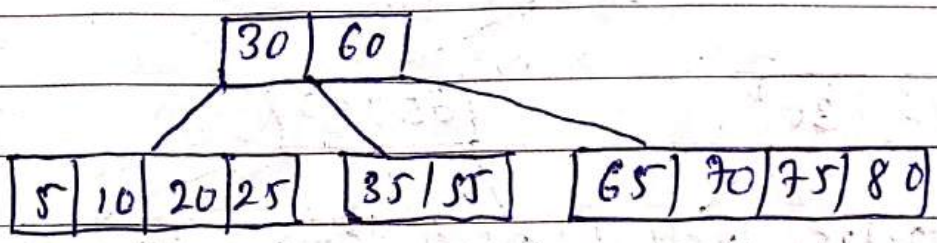
iii) Inserting 55, 60, 25, 5



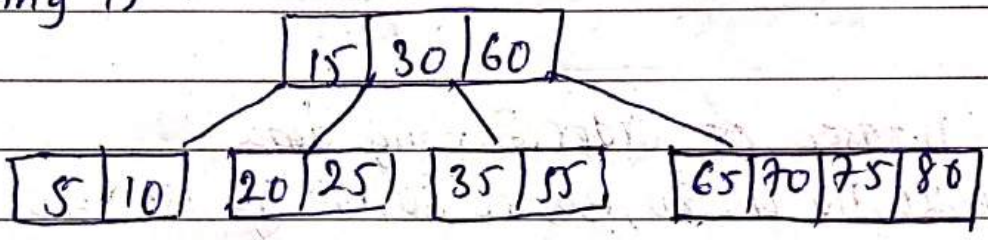
iv) Inserting 65



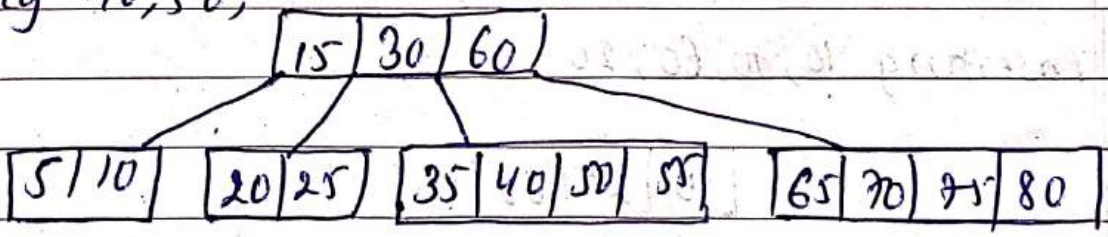
v) Inserting 70, 75,



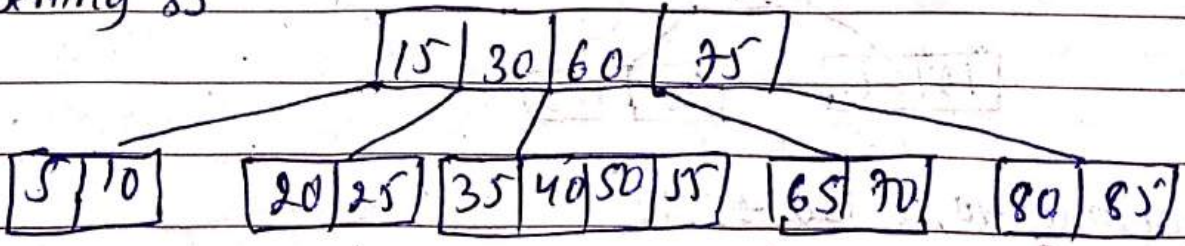
vi) Inserting 15



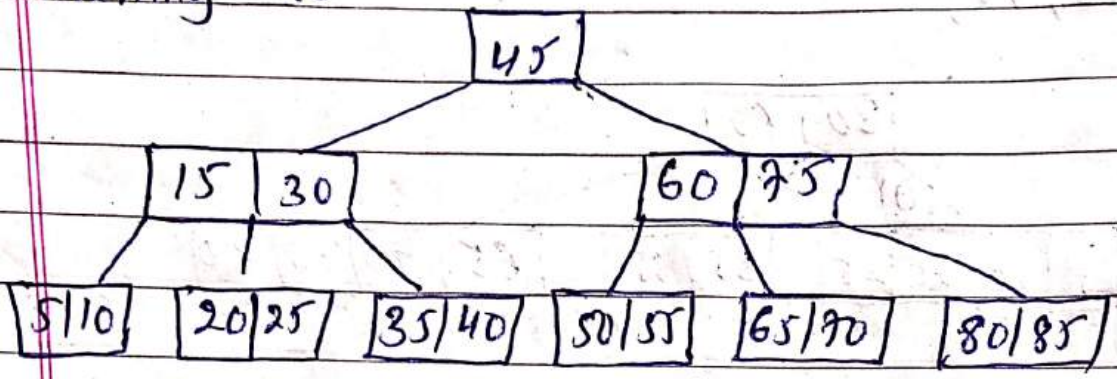
vii) Inserting 40, 50,



viii) Inserting 85

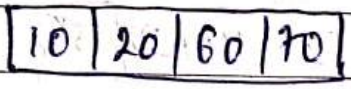


ix) Inserting 45

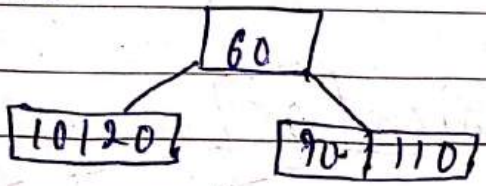


Construct B-tree of Order 5 using data
10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 240, 30, 120, 140, 200, 210, 160

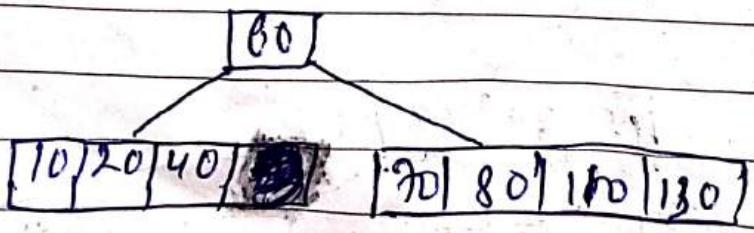
i) Inserting 10, 70, 60, 20



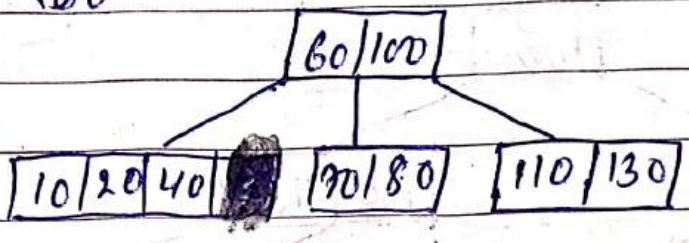
ii) Inserting 110



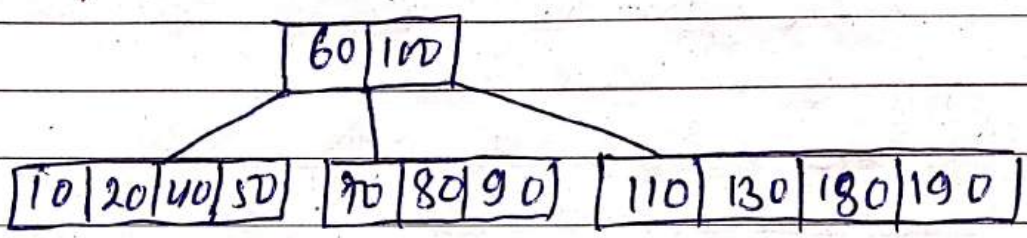
iii) Inserting 40, 80, 130, ~~100, 50~~



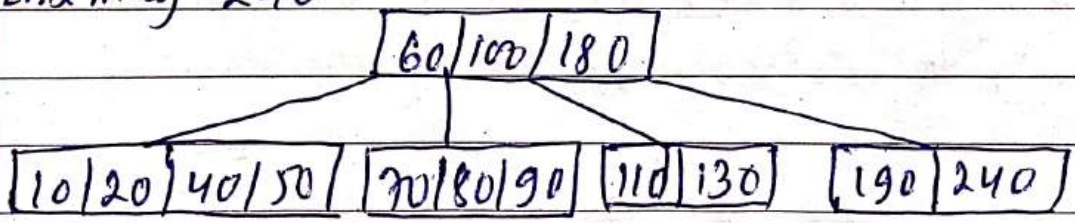
iv) Inserting 180



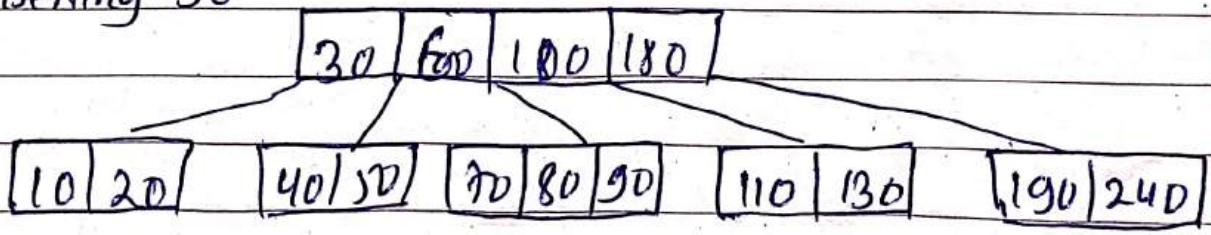
v) Inserting 50, 190, 90, 180



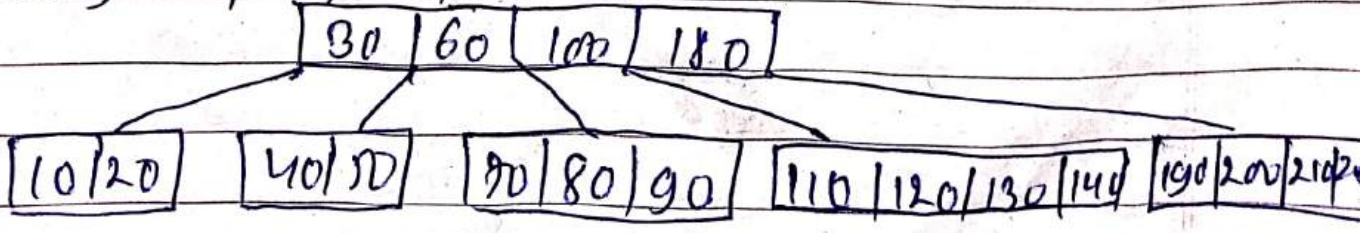
vi) Inserting 240



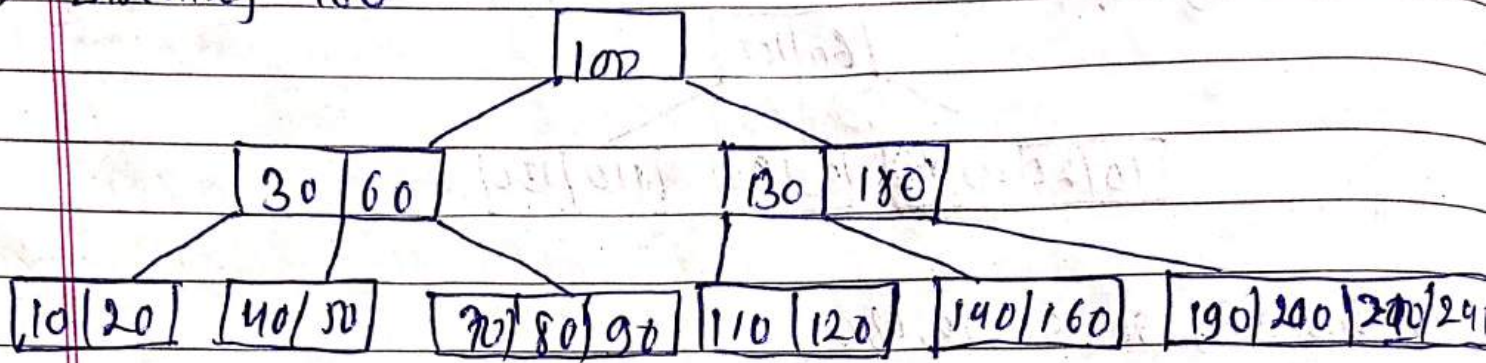
vii) Inserting 30



viii) Inserting 120, 140, 200, 210



ix) Inserting 160.



Unit :- 8 Sorting

1. Sorting

Sorting is the process of arranging data in some logical order such as ascending and descending. Sorting is important as it support for faster searching of data.

Sorting can be :- Internal and External sorting

Internal Sorting :-

It refers to the process of sorting data from the internal memory (primary memory). It is faster but applicable for limited number of data.

External Sorting :

It refers to the process of sorting data from the external file by reading from the external memory. It is slower than the internal sorting but is applicable for large number of data.

Sorting Algorithms

Some of the common sorting algorithms are :-

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap sort
- Radix Sort
- Shell Sort
- Exchange Sort
- Binary Sort

Bubble Sort :-

Example :- 30, 10, 50, 40, 20

Pass - I :- (30), (10), 30, 40, 20
swap
10, (30), (50), 40, 20
10, 30, (50), (40), 20
swap
10, 30, 40, (50), (20)
swap
10, 30, 40, 20, 50

pass - II :- (10), (30), 40, 20, 50
10, (30), (40), 20, 50
10, 30, (40), (20), 50
swap
10, 30, 20, 40, 50

pass - III :- (10), (30), 20, 40, 50
10, (30), (20), 40, 50
swap
10, 20, 30, 40, 50

pass - IV :- (10), (20), 30, 40, 50

∴ The Sorted data : 10, 20, 30, 40, 50

Concept :

In bubble Sort, each element of the array is compared with its adjac element and swapped (if required) to arrange them in proper order. If the elements are in proper order, the elements are left as it is. It requires N-1 pass, where N is the Number of elements to be sorted.

Suppose an array A[], where N elements, this algorithm works as following:-

i) In first pass, A[0] and A[1] are compared, A[1] and A[2] are compared, A[2] and A[3] are compared and so on until N elements. While comparing, the elements are swapped if required. After the completation of first pass, the largest element will be at its position.

ii) In second pass, A[0] and A[1], A[1] and A[2] and so on are compared and swapped (if required) until N-1 elements.

N-1. In pass N-1, A[0] and A[1] are compared and swapped (if required. After pass N-1 all the elements are in sorted order.

Algorithm

1. START
2. For the first iteration, compare all the elements (N). For the subsequent runs, compare (N-1), (N-2) and so on.
3. Compare each elements with its right neighbour
4. Swap the smallest element to the left.
5. Keep repeating steps 2 to 4 until the whole list is covered.
6. END

Selection Sort :-

Example :-

30, 20, 40, 50, 10

pass - I :-

Smallest among all = 10

Swap 30 and 10

Therefore, 10, 20, 40, 50, 30

pass - II :-

Smallest among remaining = 20

Therefor, 10, 20, 40, 50, 30

pass - III

Smallest among remaining = 30

Swap 40 and 30

Therefore, 10, 20, 30, 50, 40

pass - IV

Smallest among remaining = 40

Swap 50 and 40

Therefore 10, 20, 30, 40, 50

∴ The Sorted Data :- 10, 20, 30, 40, 50

Concept

In selection Sort, Sorting is done by selecting the smallest element from the list and placing in its appropriate location.

Suppose, an array A[] contains N elements, selection sort algorithm works as following :-

Pass-II :- Find the location of the smallest element in the list and interchange it with the first element A[0].
Pass-II :- find the location of the smallest element from N-1 elements (except the first element) and interchange with second element A[1].
Pass-III :- find the location of the smallest element from N-2 elements (except the first and second) and interchange with third element A[2].
- - - - -
Pass N-1 :- Find the location of the smallest element from the last two remaining elements and interchange it with the second last element A[N-2].

Algorithm
The following algorithm sorts the data in the array A[N]

1. Repeat step 2 to 5 for i=0, 1, 2 ... N-2
2. Min = A[i], loc = i
3. Repeat step 4 for j=i+1, i+2, ... N-1
4. If A[j] < Min, then :
   Min = A[j] and loc = j
5. Interchange A[i] and A[loc]
   Temp = A[i]
   A[i] = A[loc]
   A[loc] = Temp
6. EXIT

Insertion Sort :-

Example :- 77, 33, 44, 11, 88, 22, 66, 55

Pass-I :- 77, 33, 44, 11, 88, 22, 66, 55

Pass-II :- 77, 33, 44, 11, 88, 22, 66, 55
33, 77, 44, 11, 88, 22, 66, 55

Pass-III :- 33, 77, 44, 11, 88, 22, 66, 55
33, 44, 77, 11, 88, 22, 66, 55

Pass-IV :- 33, 44, 77, 11, 88, 22, 66, 55
11, 33, 44, 77, 88, 22, 66, 55

Pass-V :- 11, 33, 44, 77, 88, 22, 66, 55

Pass-VI :- 11, 33, 44, 77, 88, 22, 66, 55
11, 22, 33, 44, 77, 88, 66, 55

Pass-VII :- 11, 22, 33, 44, 77, 88, 66, 55
11, 22, 33, 44, 66, 77, 88, 55

Pass-VIII :- 11, 22, 33, 44, 66, 77, 88, 55
11, 22, 33, 44, 55, 66, 77, 88

Concept :-

In Insertion sort, data are sorted by inserting them in appropriate position in the list. It works as following :-

Suppose an array A[N] contains N elements

Pass-I : A[0] is itself sorted

Pass-II :- A[1] is inserted so that A[0] and A[1] are in sorted order.

Pass-III :- A[2] is inserted so that A[0], A[1] and A[2] are in sorted or

---

Pass-N :- A[N-1] is inserted so that all the data in the list are in sorted order.

Algorithm

The following algorithm sorts N data in the array

(1) Repeat steps 2 to 4 for $i = 0, 1, 2, \ldots N-1$

(2) Temp = A[i] and PTR = i-1

(3) Repeat While temp < A[PTR]

a) Set A[PTR+1] = A[PTR] [Moves element forward]

b) Set PTR = PTR-1

(4) Set A[PTR+1] = temp [Insertion in proper position]

(5) Exit

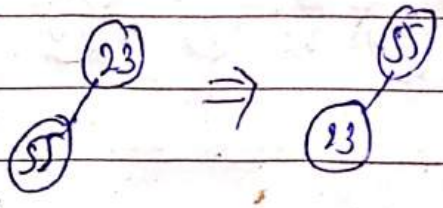
Heap Sort :-

Example - 23, 55, 46, 35, 10, 90, 84, 31

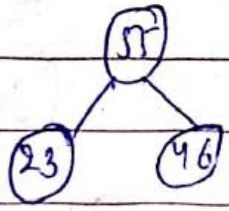
(A) Building a Heap

i) Inserting 23

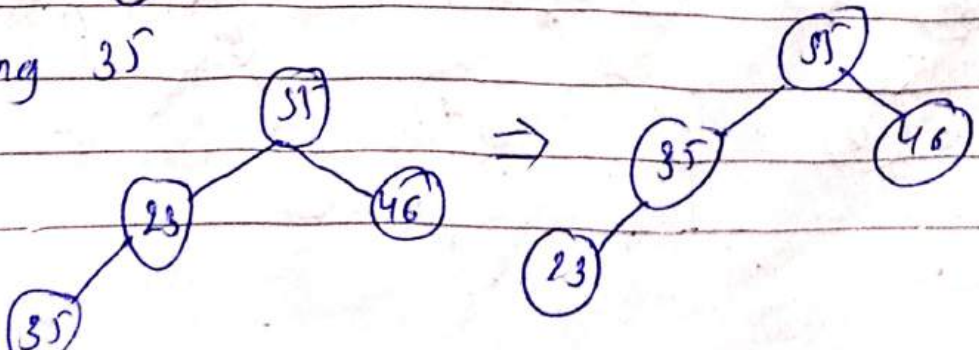
ii) Inserting 55



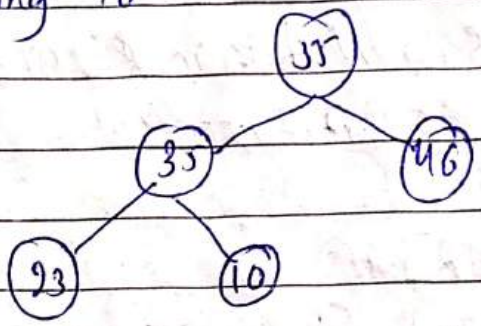
iii) Inserting 46



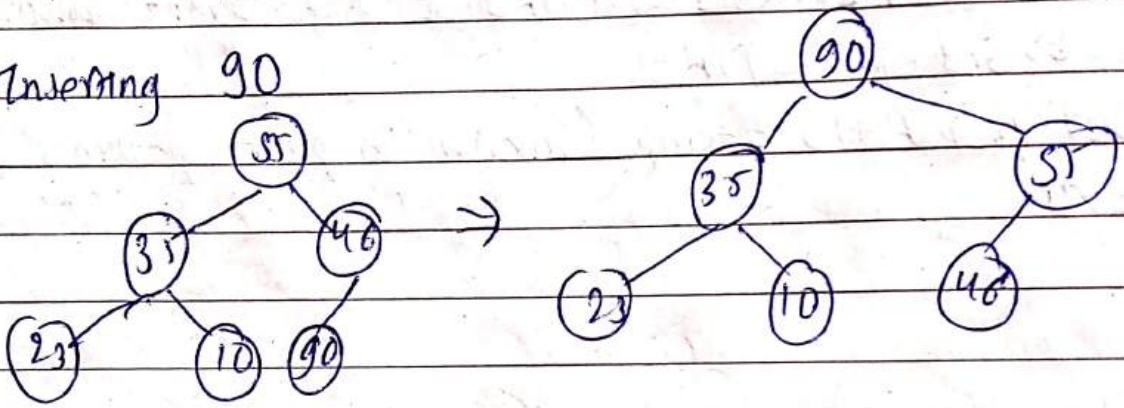
iv) Inserting 35



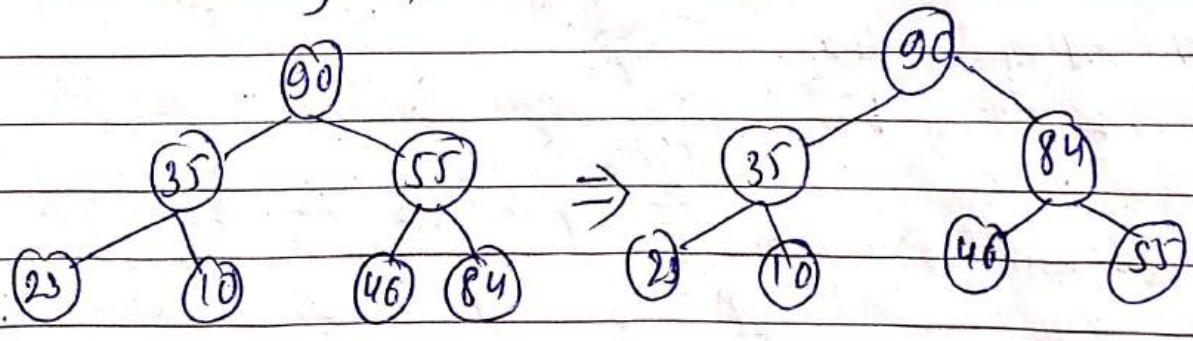
v) Inserting 10



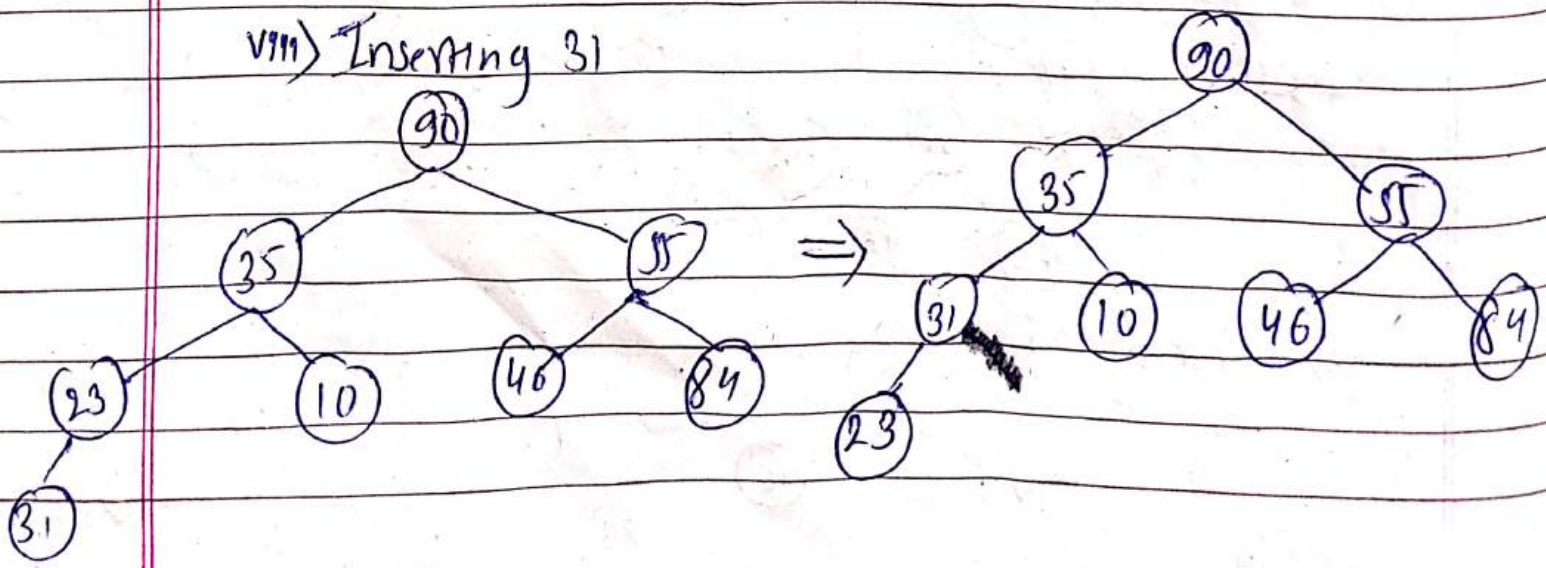
vi) Inserting 90



vii) Inserting 84

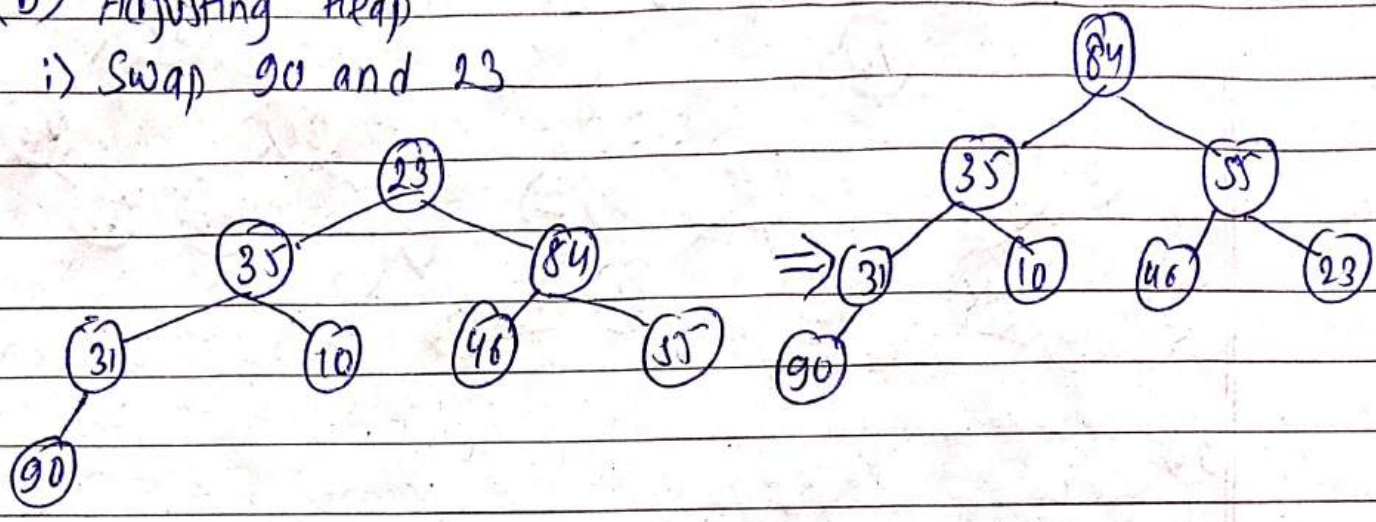


viii) Inserting 31

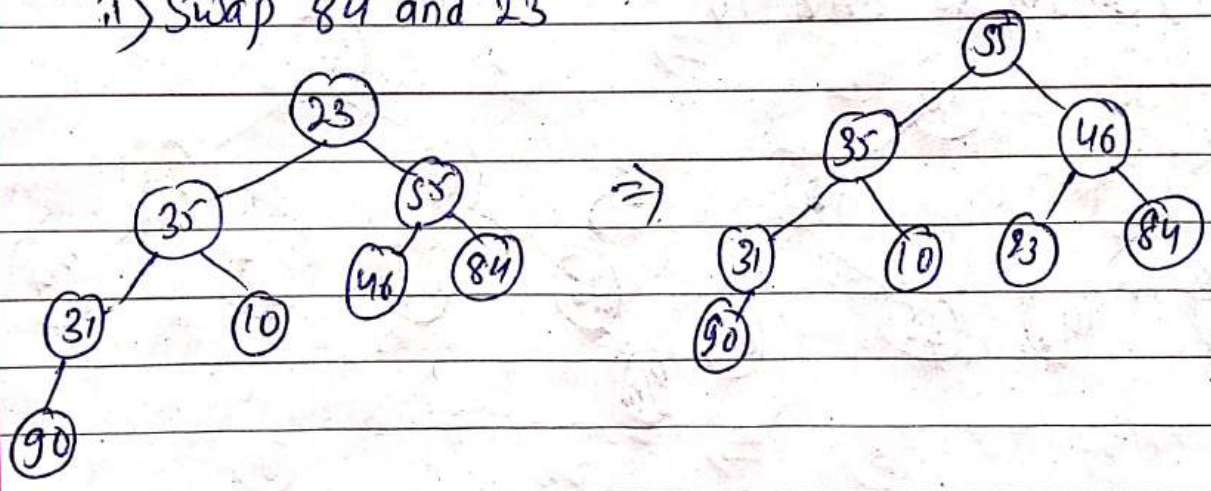


(B) Adjusting Heap

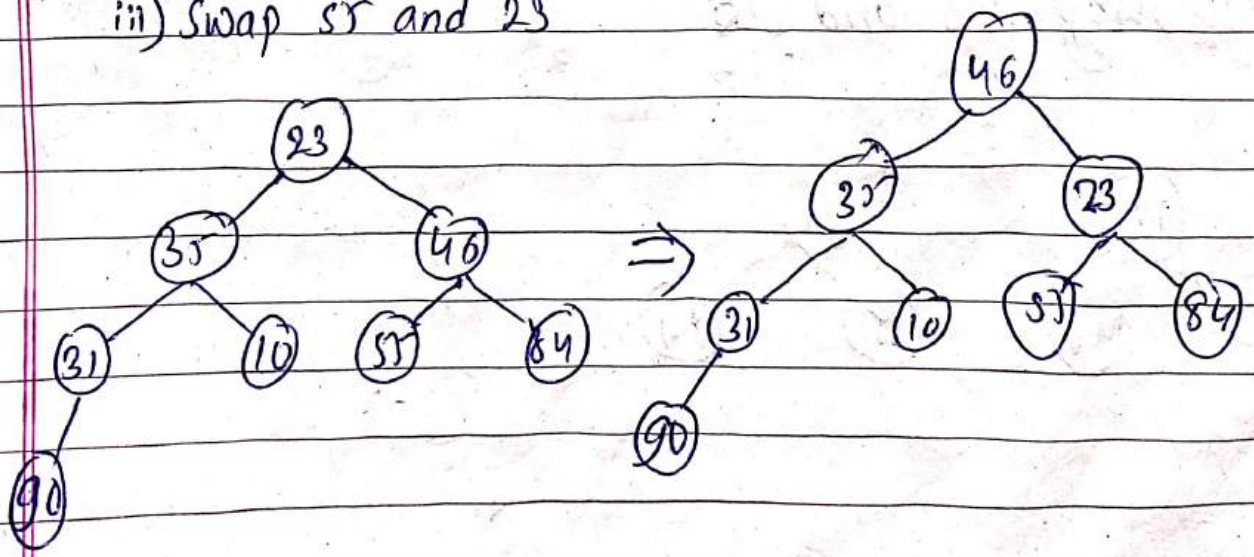
i) Swap 90 and 23



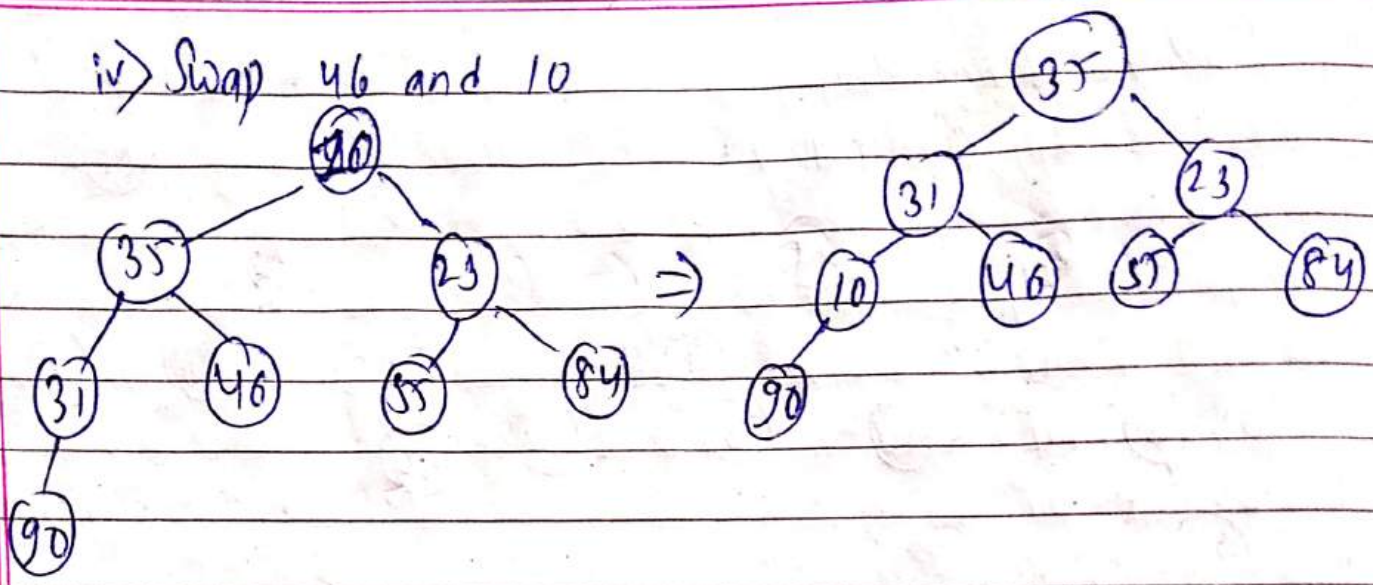
ii) Swap 84 and 23



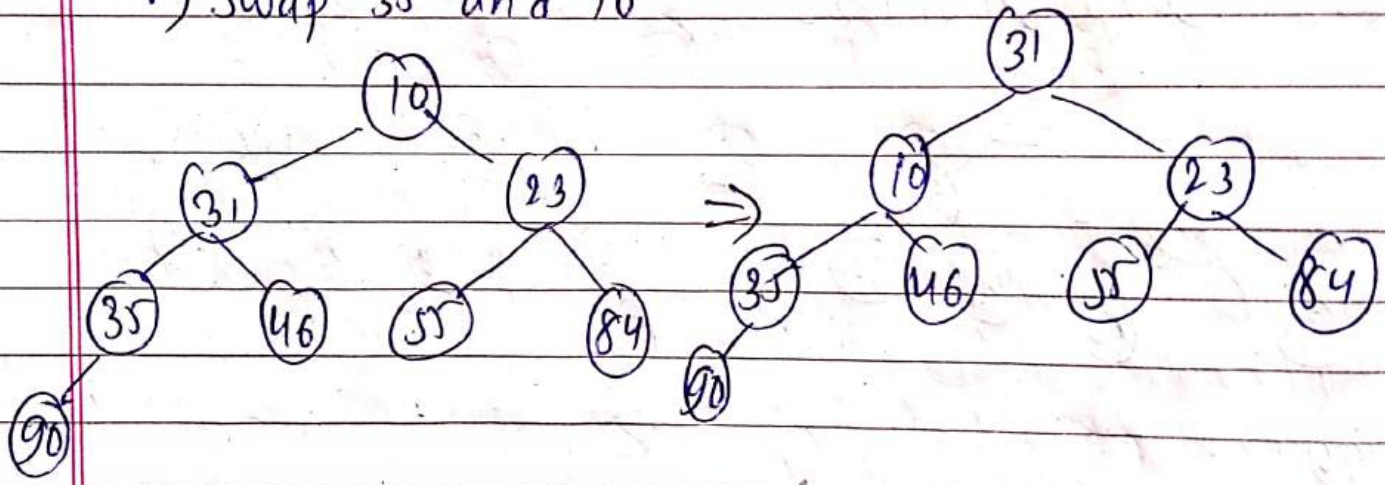
iii) Swap 55 and 23



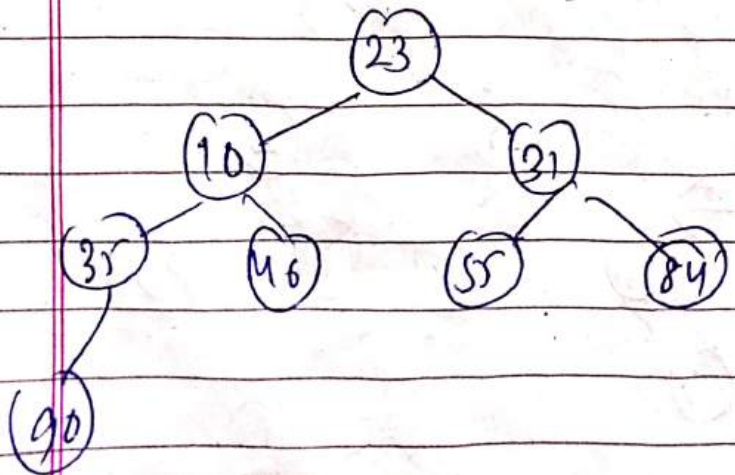
iv) Swap 46 and 10



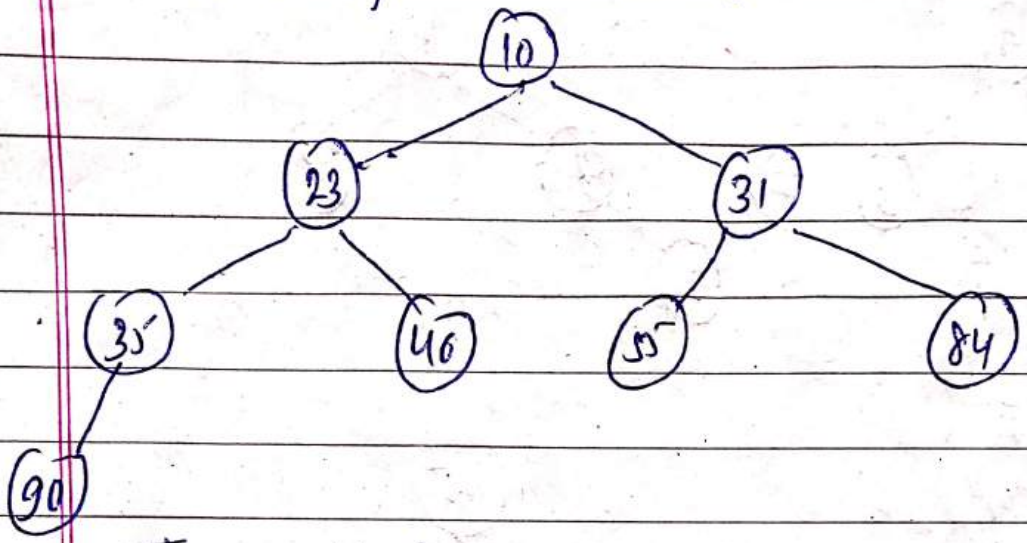
v) Swap 35 and 10



vi) Swap 23 and 31



vii) Swap 23 and 10



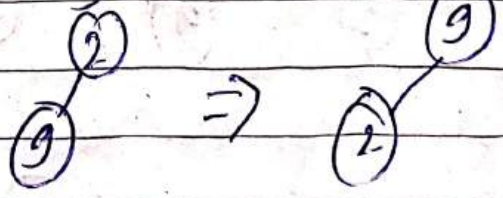
∴ The sorted data are :- 10, 23, 31, 35, 46, 55, 84 and 90

Sort the following data in ascending order by Heap Sort Method :- 2, 9, 3, 12, 15, 8, 11

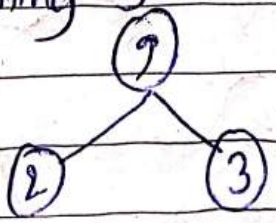
(A) Building a Heap

i) Inserting 2

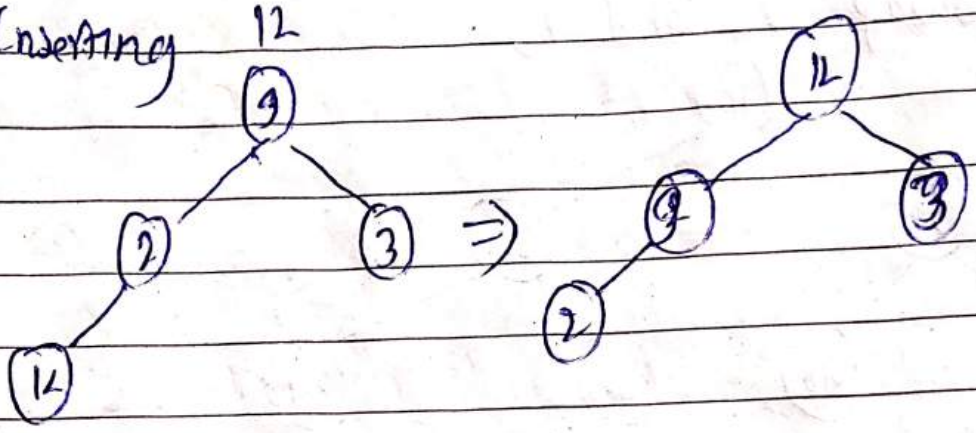
ii) Inserting 9



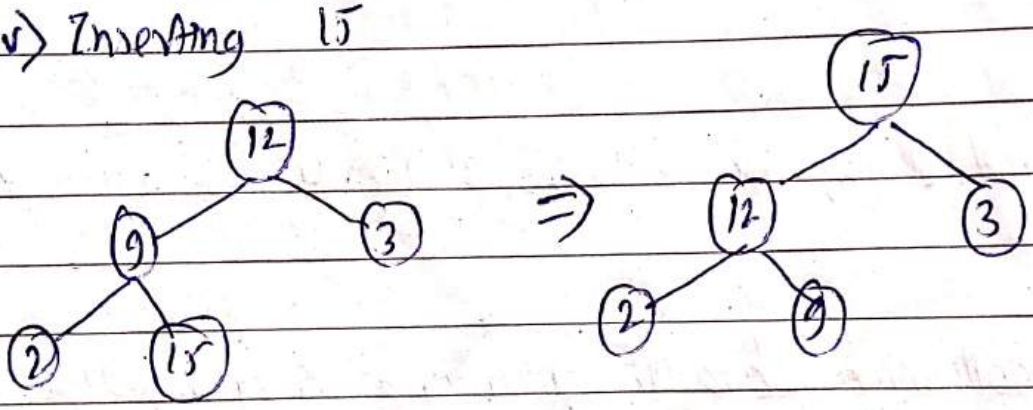
iii) Inserting 3



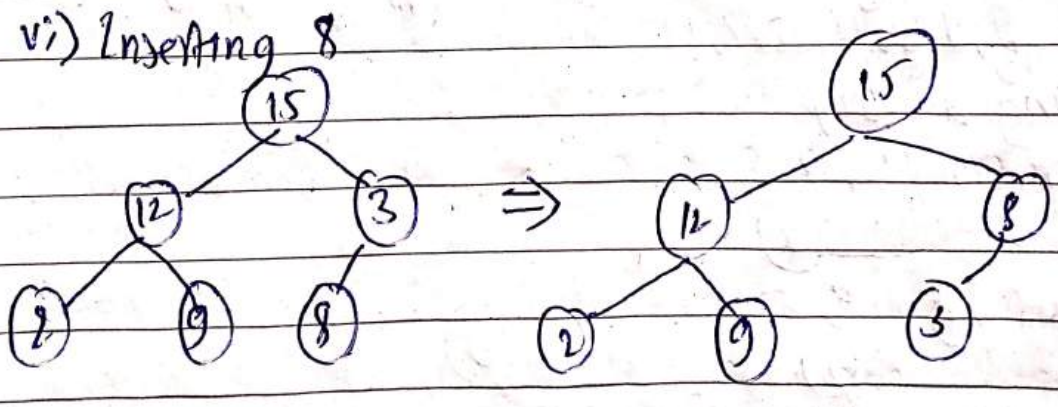
iv) Inserting 12



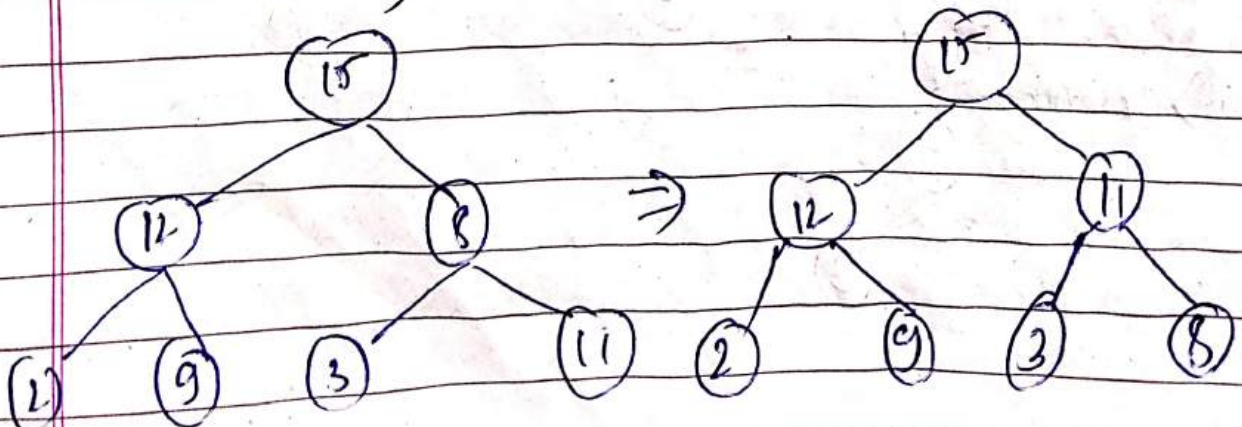
v) Inserting 15



vi) Inserting 8

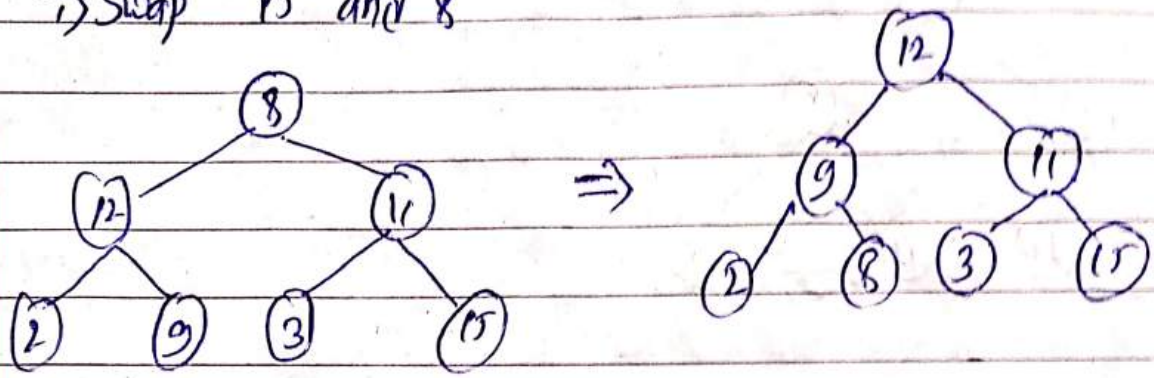


vii) Inserting 11

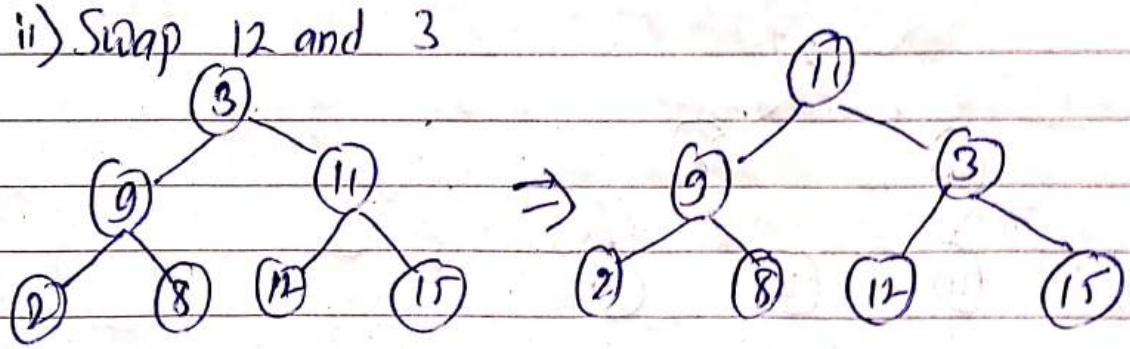


(B) Adjusting Heap

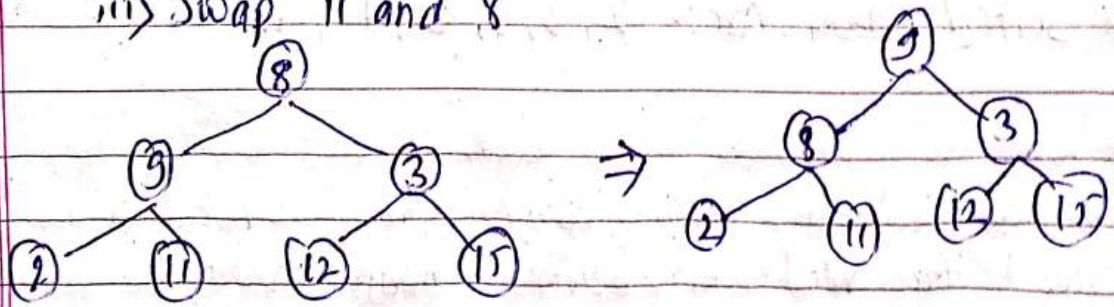
i) Swap 15 and 8



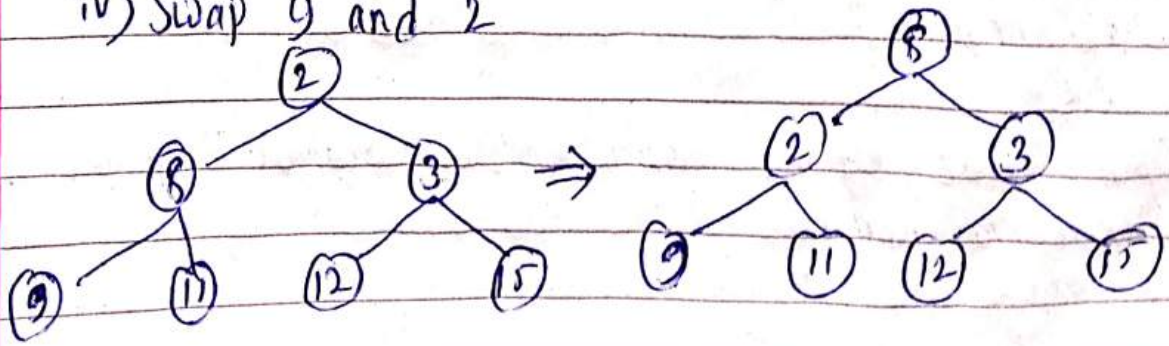
ii) Swap 12 and 3



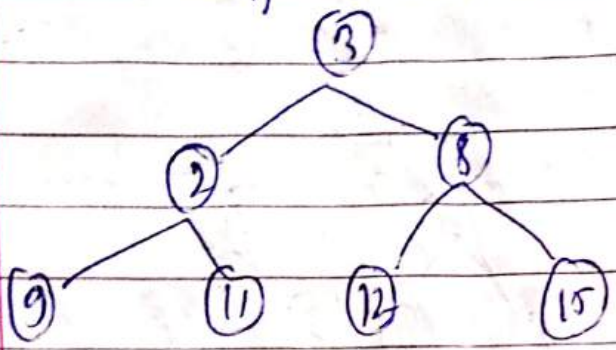
iii) Swap 11 and 8



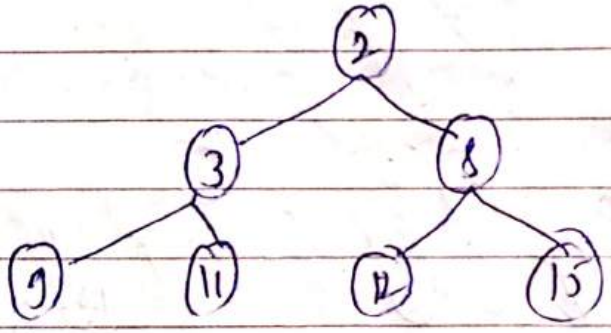
iv) Swap 9 and 2



v) Swap 8 and 3



vi) Swap 3 and 2



∴ The sorted data are:- 2, 3, 8, 9, 11, 12, 15

Concept

A heap is an almost complete binary tree whose elements have keys (value) that satisfy the following heap property:-

— The value of the parent node is larger than the value of the child nodes

OR

— The value of the parent node is smaller than the value of the child nodes.

Types of Heap

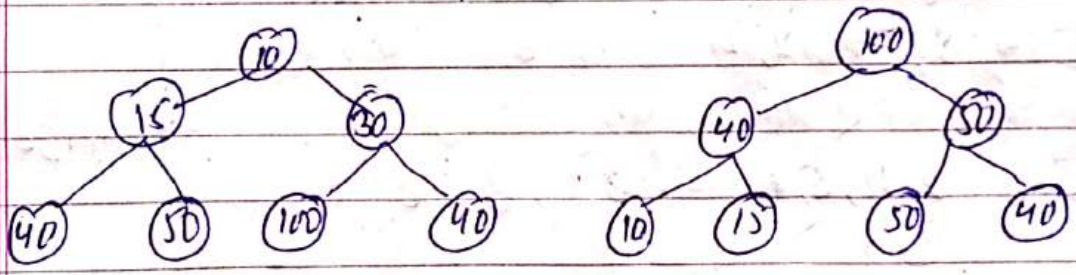
i) Max Heap

ii) Min Heap

In max heap the value of the parent node is always larger than or equal to the value of the child nodes.

In min heap the value of the parent node is always smaller than or equals to the value of child nodes.

Heap Data Structure



∴ Fig:- Min Heap

∴ Fig:- Max Heap

Heap Sort

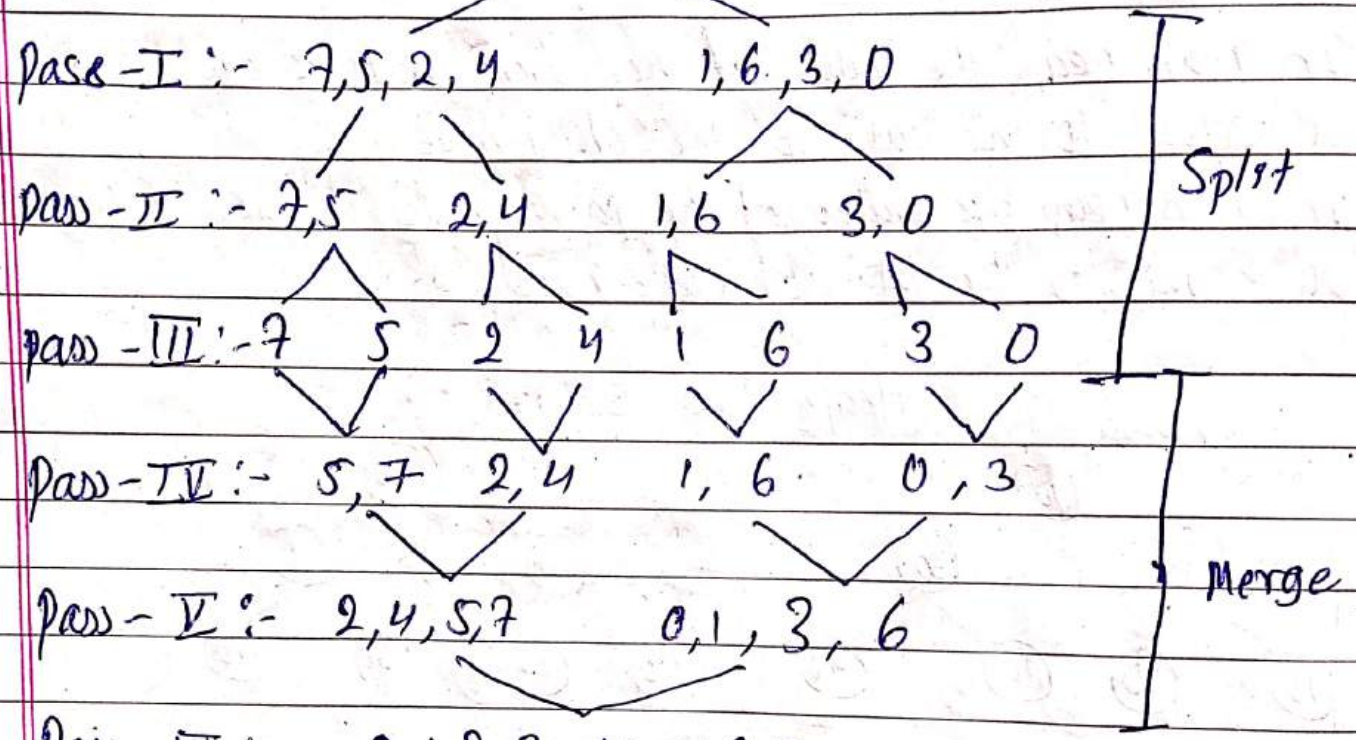
Heap sort is a comparison based sorting algorithm. It is similar to the selection sort where we find the maximum element and place it at its proper place. We repeat the same process for remaining elements.

It works as following:-

1. Build the max heap from the array.
2. Swap the root (largest number) with the last element of the array.
3. Discard this last node to form the max heap of the remaining elements.
4. Repeat step 2 and 3 until one node remains.

Merge Sort :-

Example :- 7, 5, 2, 4, 1, 6, 3, 0



Concept :-

Merge sort is an efficient sorting algorithm. It uses divide and conquer concept. It involves merging two or more sorted files into third sorted file. The process of merge sort includes three basic operation :-

i) Divide the array into two sub arrays of roughly equal size

ii) Recursively sort the two sub arrays

iii) Merge the two sorted sub arrays into a single sorted array.

Algorithm

```
Mergesort(array A, int p, int r)
{
    If(p<r){
        q=(p+r)/2
        Mergesort(A,p,q)
        Mergesort(A,q+1,r)
        Merge(A,p,q,r)
    }
}
Merge(array A, int p, int q, int r)
{
    array B[p...r]
    i=k=p
    j=q+1
    While(i<=q and j<=r){
        If(A[i]<=A[j]){
            B[k]=A[i]
            k=k+1
            i=i+1
        }
        Else{
            B[k]=A[j]
            k=k+1
            j=j+1
        }
    }
```

```
While (i<=q) {
        D[k] = A[i]
        k = k+1
        i = i+1
}
While (j<=r) {
        B[k] = A[j]
        k = k+1
        j = j+1
}
For i = p to r do
        A[i] = B[i]
}
```

Radix Sort :-

Example:- 348, 143, 361, 423, 538, 128, 321, 543, 366

Pass-I:-

Data	0	1	2	3	4	5	6	7	8
348									348
143				143					
361		361							
423				423					538
538									128
128									
321		321							
543				543					
366							366		

Therefore:- Data after pass I:- 361, 321, 143, 423, 543, 366, 348, 538, 128

Pass - II :-

Data	0	1	2	3	4	5	6	7	8	9
361							361			
321			321							
143					143					
423			423							
543					543					
366							366			
348					348					
538				538						
128		128								

∴ Therefore Data after pass II :- 361, 423, 128, 538, 143, 543, 348, 361, 366

Pass - III :-

Data	0	1	2	3	4	5	6	7	8	9
321				321						
423					423					
128		128								
538						538				
143		143								
543						543				
348				348						
361				361						
366				366						

∴ The Sorted Data :- 128, 143, 321, 348, 361, 366, 423, 538, 543,

Concept :-
Radix Sort is a common method used for arranging a large list of data, mainly the list of names.
The radix sort works as following
i) Sort all the number on the basis of last digit and combine them in an array.
ii) Sort all the number on the basis of second last digit and combine them in an array.
- - - - - -
n) Sort all the numbers on the basis of first digit and combine them in an array.

Quick Sort :-
Example :- 40, 20, 10, 80, 60, 50, 7, 30, 100
Pass-I :- pivot = 40
20, 10, 7, 30, 40, 80, 60, 50, 100
Pass-II :- pivot = 20
10, 7, 20, 30, 40, 80, 60, 50, 100
Pass-III :- pivot = 10
7, 10, 20, 30, 40, 80, 60, 50, 100
Pass-IV :- pivot = 80
7, 10, 20, 30, 40, 60, 50, 80, 100
Pass V :- pivot = 60
7, 10, 20, 30, 40, 50, 60, 80, 100

Concept:-

Quick sort is one of the efficient sorting algorithm. It sorts data by selecting a pivot element and arranging data in such a way that all the elements less than pivot are in left of pivot and the elements larger than pivot are on the right of pivot.
The selection of pivot is important as the selection of middle element as a pivot gives the best case whose as selection of the largest or the smallest number gives the worst case.

Details of Pass-I:-

Pass-I :- Pivot = 40

i) 40, 20, 10, 80, 60, 50, 7, 30, 100
(up: 40, down: 100)

ii) 40, 20, 10, 80, 60, 50, 7, 30, 100
(up: 20, down: 100)

iii) 40, 20, 10, 80, 60, 50, 7, 30, 100
(up: 10, down: 100)

iv) 40, 20, 10, [80], 60, 50, 7, 30, 100
(up: 80, down: 100)

v) 40, 20, 10, [80], 60, 50, 7, [30], 100
(up: 80, down: 30)

vi) Swap 80 and 30
40, 20, 10, 30, 60, 50, 7, 80, 100
(up: 30, down: 80)

vii) 40, 20, 10, 30, [60], 50, 7, 80, 100
(up: 60, down: 80)

viii) 40, 20, 10, 30, [60], 50, [7], 80, 100
(up: 60, down: 7)

ix) Swap 7 and 60

40, 20, 10, 30, 7, 50, 60, 80, 100

up down

x) 40, 20, 10, 30, 7, 50, 60, 80, 100

up down

xi) 40, 20, 10, 30, 7, 50, 60, 80, 100

down

xii) 40, 20, 10, 30, 7, 50, 60, 80, 100

down up

xiii) Swap 40 and 7

7, 20, 10, 30, 40, 50, 60, 80, 100

<pv pv >pv

Algorithm:-

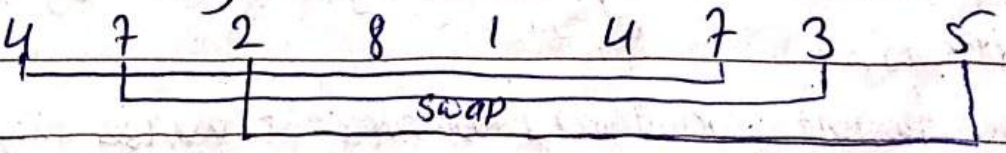
```
QuickSort(array A, int p, int r){
    if p<r {
        q = Partition(A,p,r)
        QuickSort(A,p,q-1)
        QuickSort(A,q+1,r)
    }
}
Partition(A,p,r){
    pv = A[p], up = p, down = r
    Repeat {
        Repeat {
            up = up+1
        }
```

```
Until A[up] >= Pv
Repeat {
    down = down - 1
}
Until A[down] <= Pv
Temp = A[up]
A[up] = A[down]
A[down] = Temp
}
Until up > down
A[p] = A[down]
A[down] = Pv
}
```

Shell-Sort :-

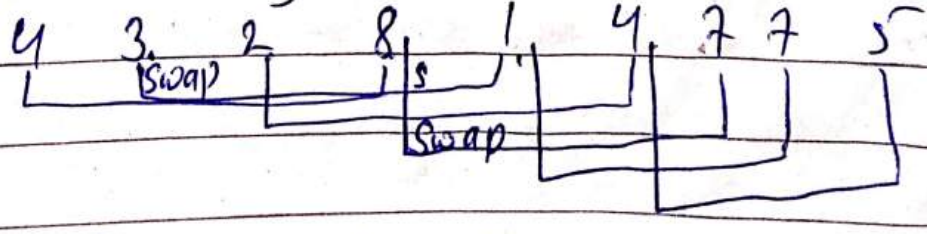
Example :- 4, 7, 2, 8, 1, 4, 7, 3, 5

Pass-I :- Using Increment = 6



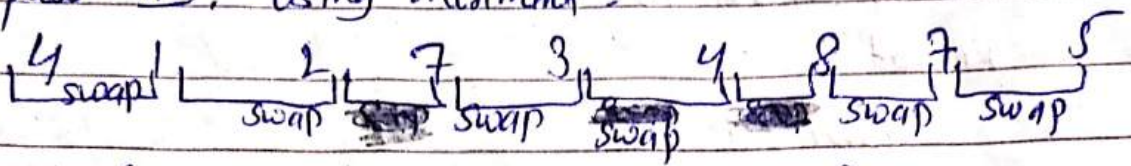
∴ Therefore Data :- 4, 3, 2, 8, 1, 4, 7, 7, 5

Pass-II :- Using Increment = 3



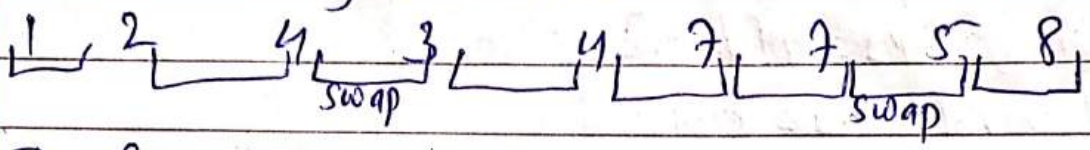
∴ Therefore Data :- 4, 1, 2, 3, 4, 7, 5

Pass - III :- Using Increment = 1



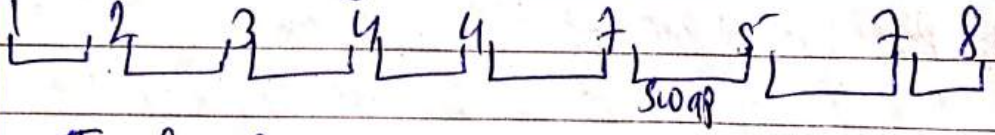
Therefore Data :- 1, 2, 4, 3, 4, 7, 7, 5, 8.

Pass - IV :- Using Increment = 1



Therefore Data :- 1, 2, 3, 4, 4, 7, 5, 7, 8

Pass - V :- Using Increment = 1



∴ Therefore Data :- 1, 2, 3, 4, 4, 5, 7, 7, 8

Concept :-

Shell sort is a generalized form of Insertion sort. It overcomes the drawback of insertion sort by comparing elements by a gap of several positions.

It is not the fastest algorithm, however it avoids large amounts data movement by comparing the elements that are far apart and then by comparing the elements that are less far apart and so on.

2. Analysis of Algorithm

The analysis of Algorithm refers to the investigation of an algorithm efficiency with respect to:-

i) Processing Time / Running Time

ii) Memory Requirement

So, The following complexities are compared:-

i) Time Complexity

ii) Space Complexity

i) Time Complexity

Time Complexity is the function that describes the amount of time an algorithm in terms of input to the algorithm. The time here means

- The number of times the loop is repeated
- The number of comparision between data
- The number of memory accessed perform

The time doesn't represent the natural unit of time such as hour, minutes, seconds etc. The algorithm with lower time complexity is treated as the better one.

ii) Space Complexity

It is a function that describes the amount of memory space required by the algorithm for processing. It doesn't represent the memory required to store the data itself. But the additional memory required for processing.

The algorithm with lower space complexity is treated as the better one. The space complexity is sometime ignored as space required is minimum and the present computers have unlimited issues of space.

3. Worst Case, Best Case and Average Case

Worst Case
Worst case complexity gives the upper bound (upper limit) on the running time and memory requirement of the algorithm for all instances of input. This ensures that no input can overcome the running time and memory requirement provided by the worst case complexity.

Best Case
Best case complexity gives the lower bound (lower limit) on the running time and memory requirement of the algorithm for all the instances of input. This indicates that no input can have lower running time or memory requirement than defined by the best case complexity.

Average Case
It gives the average number of running time and memory requirement of an algorithm for all the instances of input.

4. Asymptotic Notations
It is a mathematical function used to describe the time and space complexity of an algorithm. The common notations are:-
a) Big Oh Notation (O)
b) Big Omega Notation (Ω)
c) Big Theta Notation (Θ)

a) Big Oh Notation (O)

It is used to represent the asymptotic upper bound (upper limit) of the algorithm. It defines that an algorithm can never have complexity more than that define by the Big Oh Notation (O). It is used to define the worst case analysis of an algorithm.

b) Big Omega Notation (Ω)

It is used to represent the asymptotic lower bound (Lower limit). It is used to define the best case analysis of the algorithm.

c) Big Theta Notation (Θ)

It is used to represent the asymptotic tight bound (both upper and lower limit). It is used to define the average case analysis of the algorithm.

5. Efficiency of Sorting Algorithm

Algorithm	Worst Case	Best Case	Average Case
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Shell Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3)$

Unit :- 9 Searching

1. Introduction to Searching

Searching is the process of finding out whether the required data is available in the given list or not. If the searched data is available in the list, it will return the location of the data, otherwise the search will be unsuccessful.

Different Searching Techniques

a) Sequential Search / linear Search
b) Binary Search
c) Hashing

a) Sequential Search / linear Search

Example:- a[] : 40, 30, 50, 10, 60

Search Data 50

i) Compare 50 and a[0], Not found
ii) Compare 50 and a[1], Not Found
iii) Compare 50 and a[2], Found, Search is Successful at location = 2

Search Data 70

i) Compare 70 and a[0], Not Found
ii) Compare 70 and a[1], Not found
iii) Compare 70 and a[2], Not found
iv) Compare 70 and a[3], Not Found
v) Compare 70 and a[4], Not found
Search is Unsuccessful

Concept:
In a linear search, the required data is searched in the list one after another from the beginning until either the required data is found or all the data in the list are compared till the end to determine the search is unsuccessful.

It works as following :-

i) Search the required data with the first element in the list, if it is the required data return the location and exit

ii) Search the required data with the second element in the list, if it is the required data return the location and EXIT.

---

n) Search the required data with the last element in the list, if it is the required data return the location and EXIT otherwise return search is unsuccessful.

Algorithm:
The following algorithm search the Data in the linear list A[] with N element. It will determine the location (loc) of the data, if the search is successful.

Step 1 : [Initialize]
Set i=0 and loc=-1
Step 2 : Repeat steps 3 and 4 while loc=-1 and i<=N-1
Step 3 : If Data = A[i], then set loc = i
Step 4 : i=i+1
Step 5 : If loc=-1 then
print Search is Unsuccessful

Else

Print the required data is at Location = loc

Step 6 : Exit

Complexity Analysis of Linear Search:

Worst Case: O(n)

Best Case : O(1)

Average case: O(n)

b) Binary Search

Example:- 9, 12, 24, 30, 36, 45, 70

i) Search data 24

a) BEG = 0, END = 6, MID = $\frac{0+6}{2}$ = 3

Now, A[MID] = A[3] = 30

Since, 30 ≠ search data (24)

∴ END = MID - 1 = 3 - 1 = 2

b) BEG = 0, END = 2, MID = $\frac{0+2}{2}$ = 1

Now, A[MID] = A[1] = 12

Since, 12 < search data (24)

∴ BEG = MID + 1 = 1 + 1 = 2

c) BEG = 2, END = 2, MID = $\frac{2+2}{2}$ = 2

Now, A[MID] = A[2] = 24

Since 24 = search data

∴ Search is successful at location MID = 2

ii) Search data 45

a) BEG = 0, END = 6, MID = $\frac{0+6}{2} = 3$

Now, A[MID] = A[3] = 30

Since, 30 < search data (45)

∴ BEG = MID + 1 = 3 + 1 = 4

b) BEG = 4, END = 6, MID = $\frac{4+6}{2} = 5$

Now, A[MID] = A[5] = 45

Since 45 = Search data

∴ Search is successful at location MID = 5

ii) Search Data 25

a) BEG = 0, END = 6, MID = $\frac{0+6}{2} = 3$

Now, A[MID] = A[3] = 30

Since, 30 > search data (25)

∴ END = MID − 1 = 3 − 1 = 2

b) BEG = 0, END = 2, MID = $\frac{0+2}{2} = 1$

Now, A[MID] = A[1] = 12

Since 12 < Search data (25)

∴ BEG = MID + 1 = 1 + 1 = 2

c) BEG = 2, END = 2, MID = $\frac{2+2}{2} = 2$

Now, A[MID] = A[2] = 24

Since 24 < search data (25)

∴ BEG = MID + 1 = 2 + 1 = 3

d) BEG = 3, END = 2

Here BEG > END

∴ The Search data is not in the list and the search is unsuccessful.

Concept:-

Binary search is one of the extremely efficient algorithm. However, it required the data to be in sorted order.

The logic behind this technique is as following:-

i) Find the middle element of the array.

ii) Compare the middle element with the search data.

iii) There can be three cases:-

CASE-I:- If the middle element is the search data, the search is successful and return the location of the middle element.

CASE-II:- If the middle element is less than the search data, then search only the second half of the array. [Update the BEG]

CASE-III:- If the middle element is greater than the search data, then search only the first half of the array. [Update the END]

iv) Repeat the above steps until either the search data is found or exit if (BEG > END)

Algorithm

The following algorithm search DATA in the array a[P-Q] with N elements and return location (loc) of the search data of search = successful.

(1) Set:- BEG = P, END = Q and MID = int((BEG + END)/2)

(2) Repeat steps 3 and 4 while BEG <= END and a[MID] != DATA

(3) If DATA < a[MID], then:

set END = MID - 1

else

set BEG = MID + 1

(4) Set MID = int((BEG + END)/2)

(5) If a[MID] = DATA, then:

Set loc = MID and print Search is Successful at location = MID

else

Set loc = 1 and print Search is Unsuccessful.

(6) EXIT

Complexity Analysis of Binary Search

Worst and Average Case = $O(\log n)$

Best Case = $O(1)$

c) Hashing

Hashing is one of the efficient searchnique technique.

It uses hash value of the data to search the required data.

The hash value of the data is calculated using a Specific hash function.

The hash value calculated from the hash function is stored in a table called a hash table.

During Searching, hash value of the required data is calculated using hash function, the hash value is searched in the hash table to locate the original data.

This Searching technique is appropriate for searching large number of records in a file or the database.

Hash function

It is a mathematical function used for calculating the hash value.

Some of the common hash function are:-

i) MID Square method

ii) Folding Method

iii) Division Method

i) Mid Square Method

In this method, the number (K) is squared and the middle digits are used as the hash value.

Example K = 3205

$K^2 = 10,272,025$

H(K) = 72

K = 7148

$K^2 = 51,093,904$

H(K) = 93

ii) Folding Method

In this method, the number (K) is divided into two halves and added to generate the hash value.

Example:-

K = 3205

H(K) = 32 + 05 = 37

K = 7148

H(K) = 71 + 48 = 119

iii) Division Method

In this method, the hash value is generated by dividing the original number by a selected prime number.

$H(k) = K \bmod m$ OR

$H(k) = (K \bmod m) + 1$

Where, $H(k)$ = Hash Value, K = number, M = Divisor (prime Number)

Example;

$K = 3205$

$M = 7$

$H(k) = 3205 \bmod 7 = 6$

$K = 7148$

$H(k) = 7148 \bmod 5 = 3$

Hash Collision

It is the condition in which two or more data have the same hash value when a specific hash function is applied. Hash collisions are unavoidable. However, it can be managed.

Collision resolution methods can be applied to manage the hash collision.

Some popular methods used for collision resolution are:

1) Open Addressing
   a) Linear probing
   b) Quadratic probing
   c) Double [redacted] Hashing

ii) Rehashing
iii) Chaining
iv) Hashing using buckets

i) Open Addressing
In this technique, when a data cannot be placed at its position calculated by the hash function (hash collision occurs), another location in the hash table is searched. There are three methods in open addressing:- linear probing, quadratic probing and double probing

a) Linear probing
In this technique, hash collision is resolved by placing the data in the next empty place in the hash table. It starts with the location where the hash collision occurs and insert in the next empty location.

Example: Consider a hash table of size 10 and Insert the keys: 62, 37, 36, 44, 67, 91, and 107 using linear probing.

Soln

Hash Table

0	1	2	3	4	5	6	7	8	9
	91	62		44		36	37	67	107

1) Inserting 62
K=62
using division method
H(k) = 62 mod 10 = 2, Insert 62 at location = 2

ii) Inserting 37

K = 37

H(K) = 37 mod 10 = 7, Insert 37 at location 7

iii) Insert 36

K = 36

H(K) = 36 Mod 10 = 6, Insert 36 at location 6

iv) Insert 44

K = 44

H(K) = 44 mod 10 = 4, Insert 44 at location 4

v) Insert 67

K = 67

H(K) = 67 mod 10 = 7, Hash collision occurs

So, Insert 67 in the next possible location i.e. 8

vi) Inserting 91

K = 91

H(K) = 91 mod 10 = 1, Insert 91 at location 1

vii) Inserting 107

H(K) = 107 mod 10 = 7, Hash collision occurs

So, Insert 107 in the next possible location i.e. 9

b) Quadrating probing

In this technique, if the hash collision occurs, the new hash value is generated as following:

(Hash value + 1^2) mod hash-table-size

If the collision occurs again, the new hash value is calculated as:

(Hash value + 2^2) mod hash-table-size

Again if the collision occurs again, the new hash value is calculated as:
(Hash value $+3^2$) mod hash_table_size
And so on...
i.e. (Hash value $+i^2$) mod hash_table_size, where i is the number of hash collision occured.

Example:-
Insert the keys 89, 18, 49, 58 and 69 with the hash table size 10 using quadratic probing
Soln

Hash Table

0	1	2	3	4	5	6	7	8	9
49		58	69					18	89

i) Inserting 89
$K = 89$
$H(K) = 89 \bmod 10 = 9$, Insert 89 at location 9
ii) Inserting 18
$K = 18$
$H(K) = 18 \bmod 10 = 8$, Insert 18 at location 8
iii) Inserting 49
$K = 49$
$H(K) = 49 \bmod 10 = 9$, Hash collision occurs
New hash value $= (9+1^2) \bmod 10 = 0$
Now, Insert 49 at location 0
iv) Inserting 58
$K = 58$

H(k) = 58 mod 10 = 8, Hash collision occurs
New Hash value = (8+1²) mod 10 = 9 mod 10 = 9, Hash collision occurs
New Hash value = (8+2²) mod 10 = 12 mod 10 = 2
Insert 58 at location 2

v) Inserting 69
K = 69
H(k) = 69 mod 10 = 9 Hash collision occurs
New hash value = (9+1²) mod 10 = 10 Mod 10 = 0, Hash collision occurs
New Hash value = (9+2²) mod 10 = 13 mod 10 = 3
Insert 69 at location 3

c) Double Hashing
In this technique, if the hash collision occurs, the collision resolution is done by using new hash function:

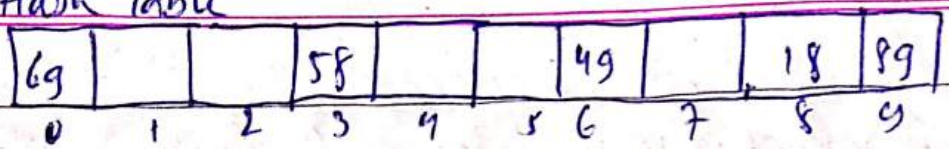
$H_1(k) = R - (k \bmod R)$

where, k is the key (data), R is the prime number smaller than the size of the hash table.
The new element can be inserted in the position calculated by using the following function;

position = (original value(k) + i * $H_1(k)$) mod hash_table_size

Example: Insert the keys 89, 18, 49, 58 and 69 with the hash table size 10 using double hashing:-

Hash Table



i) Inserting 89

K = 89

H(K) = 89 mod 10 = 9, Insert 89 at location 9

ii) Inserting 18

K = 18

H(K) = 18 mod 10 = 8, Insert 18 at location 18

iii) Inserting 49

K = 49

H(K) = 49 mod 10 = 9, Hash Collision Occurs

$H_1(K) = R - (K \bmod R) = 7 - (49 \bmod 7) = 7 - 0 = 7$

Position = (49 + 1 * 7) mod 10 = 56 mod 10 = 6

Insert 49 at location 6

iv) Inserting 58

K = 58

H(K) = 58 mod 10 = 8, Hash Collision Occurs

$H_1(K) = R - (K \bmod R) = 7 - (58 \bmod 7) = 7 - 2 = 5$

Position = (58 + 1 × 5) mod 10 = 63 mod 10 = 3

Insert 58 at location 3

v) Inserting 69

K = 69

H(K) = 69 mod 10 = 9, Hash collision occurs

$H_1(K) = R - (K \bmod R) = 7 - (69 \bmod 7) = 7 - 6 = 1$

Position = (69 + 1 × 1) mod 10 = 70 mod 10 = 0

Insert 69 at location 0

ii) Rehasing

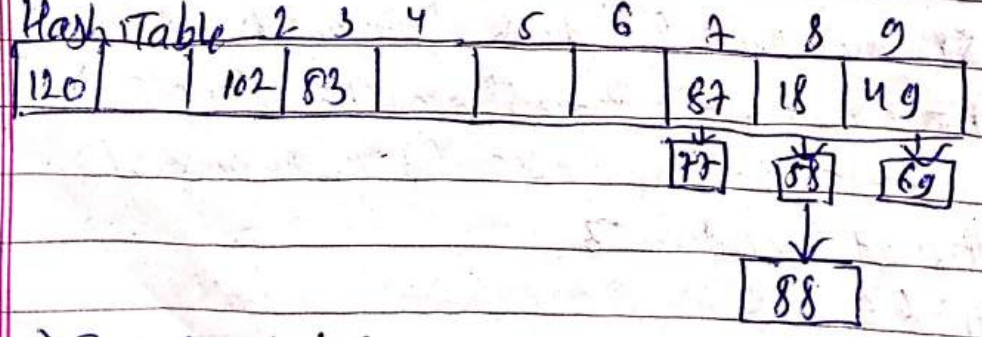
In this collision resolution method, once the collision occurs, same hash function is applied to the hash value to generate the new hash value. If collision occurs further, the same process is repeated until the unique hash value is generated.

Double hashing is also one of the concept of rehashing.

iii) Chaining

In this technique, multiple data with same hash value can be managed in the same hash table. When the hash collision occurs, another slot in the hash table is created and added for the same hash value

Example:- Insert key 102, 18, 49, 58, 69, 87, 88, 77, 83 and 20 with the hash table of size 10 Using chaining Method



i) Inserting 102

K = 102

H(K) = 102 mod 10 = 2, Insert 102 at location 2.

ii) Inserting 18

K = 18

H(K) = 18 mod 10 = 8, Insert 18 at location 2.

iii) Inserting 49

K = 49

H(K) = 49 mod 10 = 9, Insert 49 at location 9.

iv) Inserting 58

K = 58

H(K) = 58 mod 10 = 8, Hash Collision occurs, Insert 58 in new slot at location 8.

v) Inserting 69

K = 69

H(K) = 69 mod 10 = 9, Hash Collision occurs, Insert 69 in new slot at location 9.

vi) Inserting 87

K = 87.

H(K) = 87 mod 10 = 7, Insert 87 at location 7.

vii) Inserting 77

K = 77

H(K) = 77 mod 10 = 7, Hash Collision occurs, Insert 77 in new slot at location 7.

viii) Inserting 83

K = 83

H(K) = 83 mod 10 = 3. Insert 83 at location 3.

ix) Inserting 120

K = 120

H(K) = 120 mod 10 = 0, Insert 120 at location 0.

iv) Hashing Using Buckets (Bucket Addressing)

In this technique, the multiple data having same hash value are inserted in the same position in the table. The hash table use buckets. A bucket is a block of large memory space that can store multiple data.

Example:- Insert Keys 102, 18, 49, 58, 69, 87, 88, 77, 83, and 20 with the hash table of the size 10 using Hashing using buckets.

Hash Table

0	1	2	3	4	5	6	7	8	9
20		102	83				87, 77	18, 58, 88	49, 69

i) Inserting 102

K = 102

H(K) = 102 mod 10 = 2, Insert 102 at location 2

ii) Inserting 18

K = 18

H(K) = 18 mod 10 = 8, Insert 18 at location 8

iii) Inserting 49

K = 49

H(K) = 49 mod 10 = 9, Insert 49 at location 9

iv) Inserting 58

K = 58

H(K) = 58 mod 10 = 8, Hash collision, Insert 58 at location

v) Inserting 69

K = 69

H(K) = 69 mod 10 = 9, Hash Collision, Insert 69 at location 9

vi) Inserting 87

K = 87

H(K) = 87 mod 10 = 7, [redacted] Insert 87 at location 7

vii) Inserting 88

K = 88 Hash Collision

H(K) = 88 mod 10 = 8, Inserting 88 at location 8

viii) Inserting 77

K = 77

H(K) = 77 mod 10 = 7, Hash collision, Inserting 77 at location 7

ix) Inserting 83

K = 83

H(K) = 83 mod 10 = 3, Inserting 83 at location 3

x) Inserting 120

K = 120

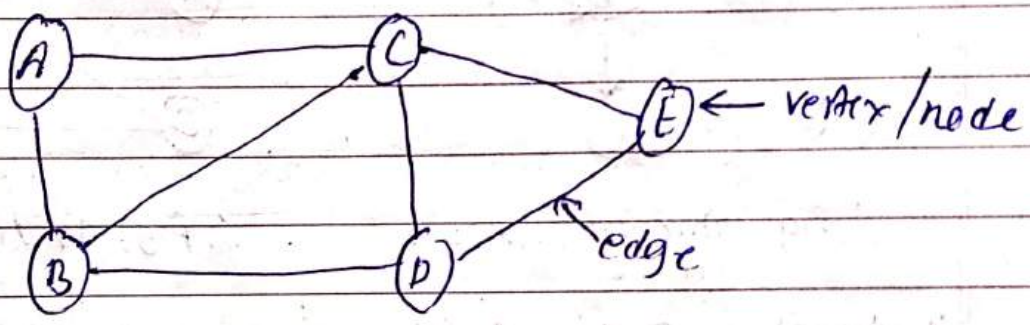
H(K) = 120 mod 10 = 0, [redacted] Inserting 120 at location 0

Unit:- 10 Graphs

1. Graph

A Graph is a non-linear data structure that includes the vertices (node) and the edge. A vertex can represent any component like the computer, networking device, electronic component, place etc. and the edge represent the wire, path, line between the vertices.

Example:-



∴ Fig:- graph

Graph is used in Applications:-

i) Electronic circuit design
ii) Computer Network design
iii) Telecommunication Network
iv) Electrical Network
v) Designing of Roads, paths, city

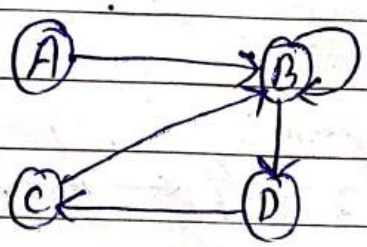
Graph theory can be used to model variety of application like:-

i) Social Network
ii) Communication Network
iii) Information Network
iv) Software Application Design
v) Transportation Network

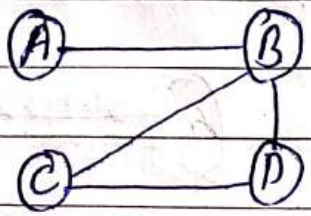
vi) Biological Networks
vii) Tournaments

Types of Graph

a) Directed and Undirected Graph



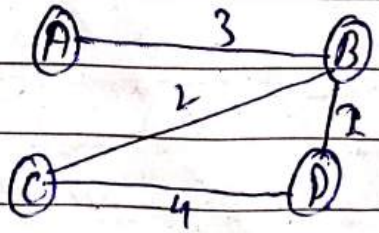
∴ Fig:- Directed Graph



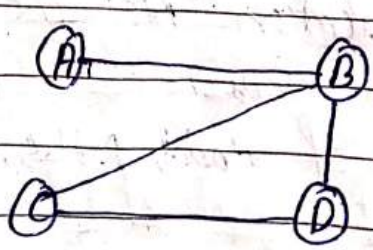
∴ Fig:- Undirected Graph

A directed graph contains direction in its edge whereas an undirected graph contains edges without direction. It represents bi-direction.

b) Weighted and Unweighted Graph



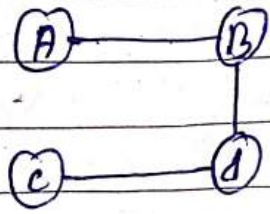
∴ Fig:- Weighted Graph



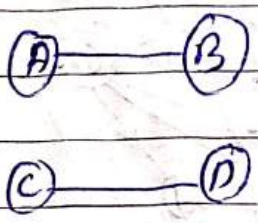
∴ Fig:- Unweighted Graph

A weighted graph contains some value as weight in its edge. The weights represents values like:- distance, time, cost, etc. An Unweighted graph doesn't contain any weight in its edge.

c) Connected and Unconnected Graph



∴ Fig:- Connected Graph



∴ Fig:- Unconnected Graph

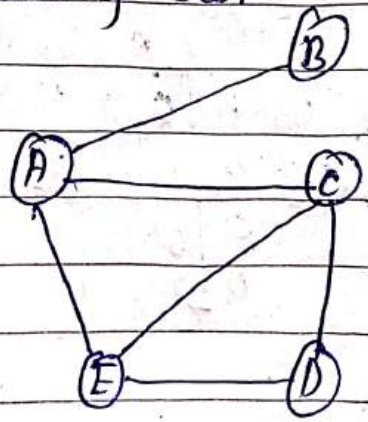
In a connected graph, all the nodes are connected to form a single structure where as In an Unconnected Graph, all the nodes are not connected.

Representation of Graph

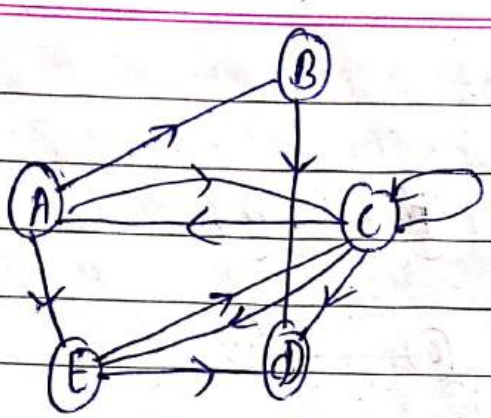
The graph can be represented in the memory by using following techniques:-

a) Adjacency list
b) Adjacency Matrix
c) Incidence Matrix

a) Adjacency List



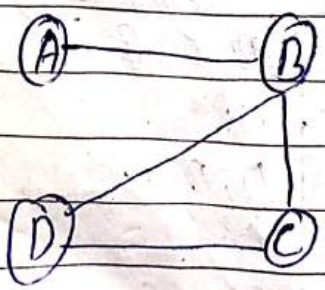
vertex	Adjacency list
A	B, C, E
B	A
C	A, D, E
D	C, E
E	A, C, D



Vertex	Adjacency list
A	B, C, E
B	D
C	C, E, A, D
D	—
E	D, C

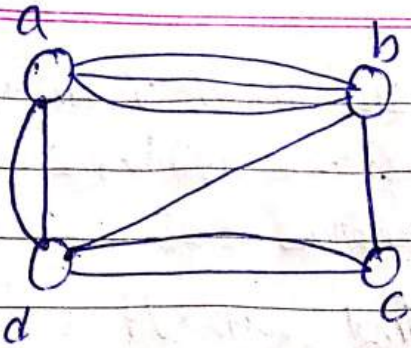
The graph with no multiple edges can be represented using adjacency list. It specifies the vertices that are adjacent to each vertex of the graph.

b) Adjacency Matrix



Adjacency Matrix =

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



∴ Adjacency matrix =

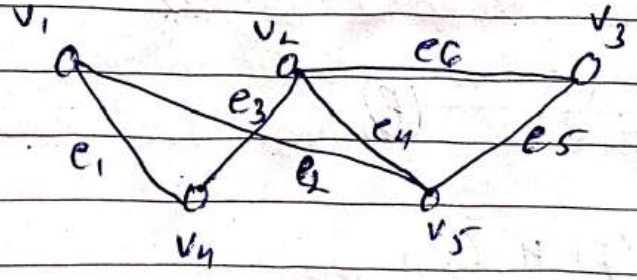
	a	b	c	d
a	0	3	0	2
b	3	0	1	1
c	0	1	0	2
d	2	1	2	0

In this representation, An adjacency matrix of size 'n×n' is formed, where 'n' is the number of vertices. The element of the ~~vertices with~~ matrix will be ~~as~~ as following:-

$$A_{ij} = \begin{cases} 1, & \text{if there is an edge between the vertices} \\ 0, & \text{if there is no edge between the vertices} \\ k, & \text{if there are } \geq 2 \text{ edges between vertices.} \end{cases}$$

Here k, is the number of edges between the vertices

c) Incidence Matrix



Incidence Matrix =

	e1	e2	e3	e4	e5	e6
v1	1	1	0	0	0	0
v2	0	0	1	1	0	1
v3	0	0	0	0	1	1
v4	1	0	1	0	0	0
v5	0	1	0	1	1	0

In this representation, A matrix of size n x m is formed where n is the number of vertices and m is the number of edges. The elements of the matrix will be as follows:-

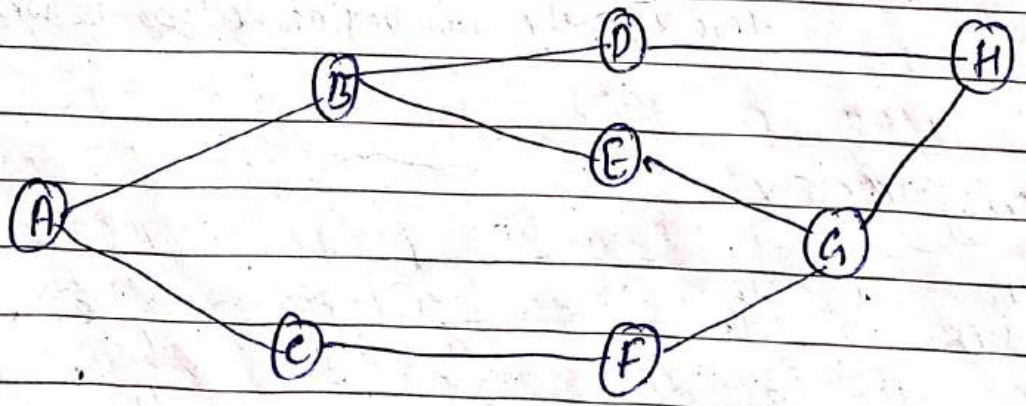
$$A_{ij} = \begin{cases} 1, \text{ if } e_j \text{ is incident with } V_i \\ 0 \text{ if } e_j \text{ is not incident with } V_i \\ 2, \text{ if } V \text{ is the end of the loop } e_j \end{cases}$$

2. Graph Traversal

Graph traversal is the process of visiting all the nodes once. Different graph traversal techniques are:-

a) BFS (Breadth First Search)

b) DFS (Depth First Search)



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

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

a) BFS (Breadth First Search)
The general idea behind BFS is:-
i) Traverse the starting node (A)
ii) Traverse all the neighbours of A.
iii) Traverse all the neighbours of neighbours of A and so on.
[It uses queue for its operation.]

b) DFS (Depth First Search)
The general idea behind DFS is:-
i) Traverse the starting node (A).
ii) Traverse a neighbour of A.
iii) Traverse the neighbour of neighbour of A and so on until the dead end is reached, Backtrack and continue the above operation for another part.

[It uses stack for its operation.]

Algorithm for BFS and DFS
The following use three states for traversal:-
i) STATUS = 1 (Readystate) = The initial state of the node.
ii) STATUS = 2 (Waitingstate) = The node is either in Queue or STACK.
iii) STATUS = 3 (Traversedstate) = The node has been traversed.

Algorithm for BFS

(1) Initialize all the nodes to the ready state (STATUS = 1)
(2) Insert the starting node (A) in the QUEUE and change its status to waiting state (STATUS = 2)
(3) Repeat step 4 and 5 until the QUEUE is empty.
(4) Remove the frontnode (N) from the QUEUE, traverse it and change its status to traversed state (STATUS = 3)
(5) Insert all the neighbours of N that are in readystate (STATUS = 1) to the rear of the QUEUE and change their status to waiting state (STATUS = 2)
(6) END

Algorithm FOR DFS

(1) Initialize all the nodes to the ready state (STATUS = 1)
(2) PUSH the starting node (A) in the STACK and change its status to waiting state (STATUS = 2)
(3) Repeat step 4 and 5 until STACK is empty
(4) Pop the TOPnode (N) from the STACK, traverse it and change its status to traversed state (STATUS = 3)
(5) PUSH all the neighbours of N that are in readystate (STATUS = 1) to the TOP of the STACK and change their status to waiting state (STATUS = 2)
(6) END

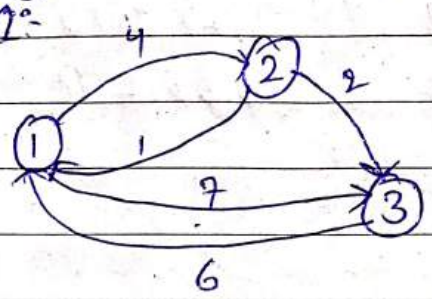
3. Floyd - Warshall Algorithm

It is a graph analysis algorithm used for finding the shortest path between nodes in a weighted graph. It finds the length of the shortest path between all pair of vertices.

It is used in application like:-

- Path computations in path finder application
- Finding optimal routing in Network, telecommunication, Internet etc

Example:-



(1) The adjacency matrix is

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & \infty & 0 \end{bmatrix} \end{array}$$

(2) Considering Vertex 1

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 7 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix} \end{array}$$

(3) Considering Vertex 2

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix} \end{array}$$

④ Considering Vertex 3

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 4 & 6 \\ 1 & 0 & 2 \\ 6 & 10 & 0 \end{bmatrix} \end{array}$$

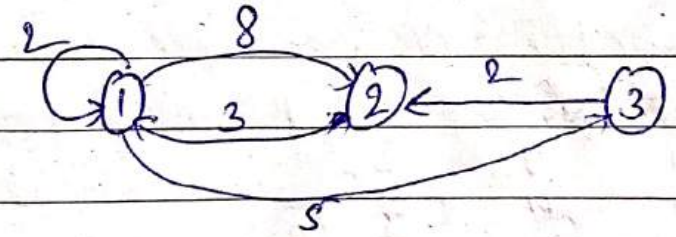
∴ Therefor, the Shortest path

$(1,1) = 0$ $(2,1) = 1$ $(3,1) = 6$

$(1,2) = 4$ $(2,2) = 0$ $(3,2) = 10$

$(1,3) = 6$ $(2,3) = 2$ $(3,3) = 0$

Example 2:-



① The adjacency Matrix

$$D_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 2 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{bmatrix} \end{array}$$

② Considering vertex 1

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3. \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{bmatrix} \end{array}$$

③ Considering vertex 2

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \end{array}$$

④ Considering vertex 3

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 7 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{bmatrix} \end{array}$$

∴ Therefore, the shortest path

(1,1) = 0	(2,1) = 3	(3,1) = 5
(1,2) = 7	(2,2) = 0	(3,2) = 2
(1,3) = 5	(2,3) = 8	(3,3) = 0

4. Spanning Tree, Minimum (minimal) Spanning tree, Spanning Forest

Spanning Tree

Spanning Tree of a graph is formed by including all the vertices of the graph without forming the cycle. So, a spanning tree of a graph will include all the vertices but may not include all the edges.

Spanning Forest

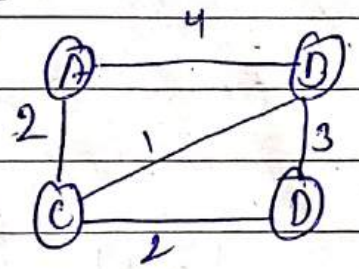
A graph can have multiple spanning trees. The spanning forest is the collection of multiple spanning trees of a graph.

Minimum (Minimal) Spanning tree

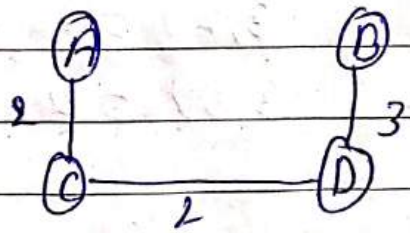
A spanning tree in a spanning forest with least total weight is called as minimum spanning tree (MST). The minimum spanning tree of the graph can be obtained by using :-

- Kruskal's Algorithm
- Prim's Algorithm

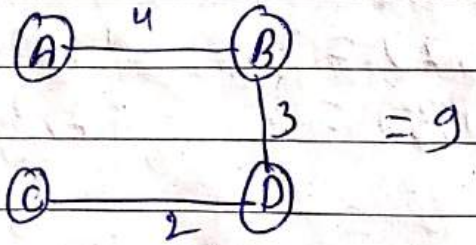
Example :-



∴ Graph

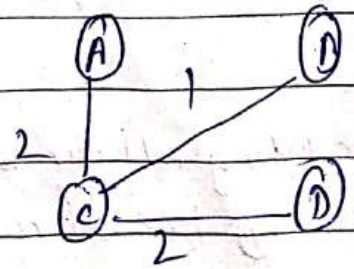


= 7



= 9

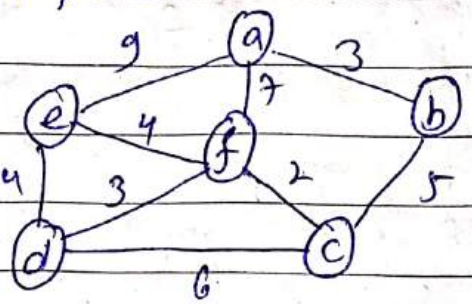
∴ All are Spanning tree



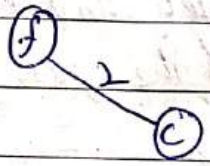
= 5 → It is minimum Spanning tree

So Kruskal's Algorithm

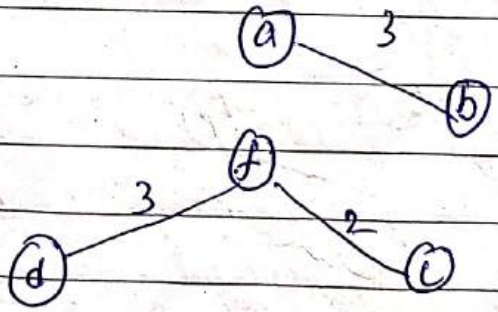
Example 1 :-



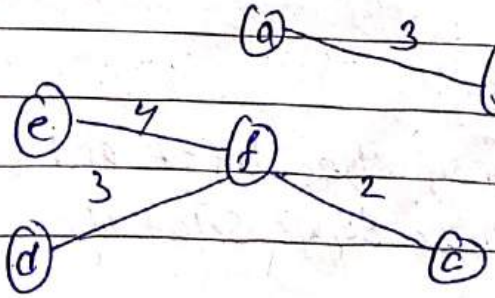
① Among all the edges, fc = 2 is the shortest. So, we select it.



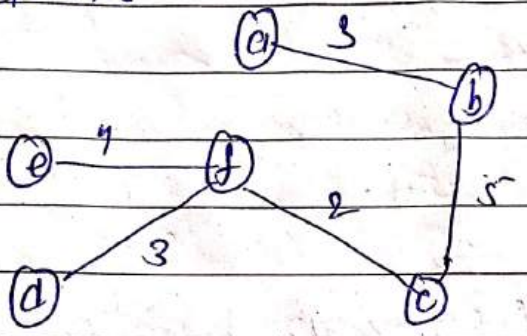
② Among all the remaining edges, ab = 3 and fd = 3 are the shortest and they will not form cycle. So, we select it.



③ Among all the remaining edges, ef = 4 and ed = 4 are the shortest. We will select only one of them (ef) as the resultant tree will include the cycle if we also select the second one (ed).

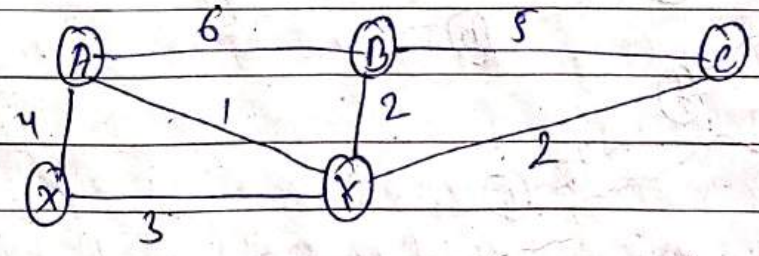


(4) Among all the remaining edges, bc = 5 is the shortest. and it will not form cycle if we include it in the tree. So, Select it.

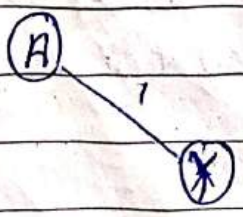


∴ Since, all the vertices are included and connected. So, the tree above is the minimum spanning tree of the given graph and its total weight is 17.

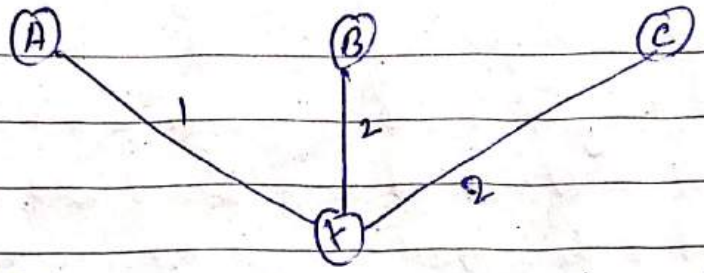
Example 2 :-



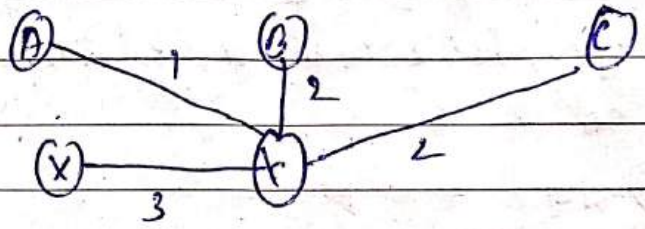
(1) Among all the edges, AY = 1 is the shortest. So, we select it.



(2) Among all the remaining edges, BY = 2 and CY = 2 are the shortest they will not form cycle. So, we select it.



(3) All among the remaining edges, XY is the shortest and it will not form cycle if we include it in the tree. So, we select it.



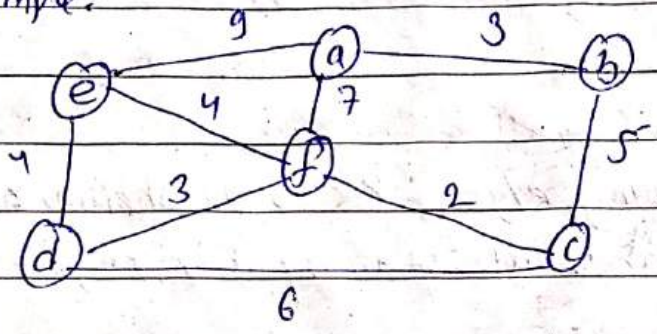
∴ Since, all the vertices are included and connected. So, the tree above is the minimum spanning tree of the given graph and it's total weight is 8.

Algorithm

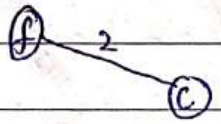
(1) Select an edge with minimum weight and include it to the spanning tree.
(2) Among all the remaining edges, select the edge with minimum weight and the tree formed doesn't contain any cycle.
(3) Repeat step 2 until all N vertices and N-1 edges are included, where N is the No. of vertices in the graph.

6. Prim's Algorithm

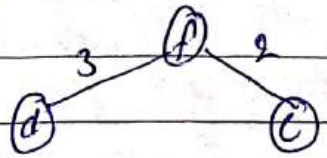
Example:-



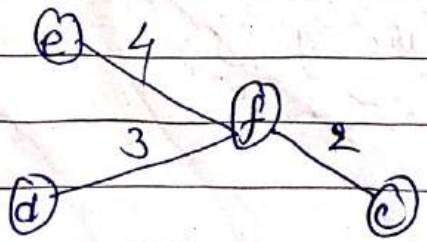
① Among all the edges, fc=2 is the shortest. So, we select it



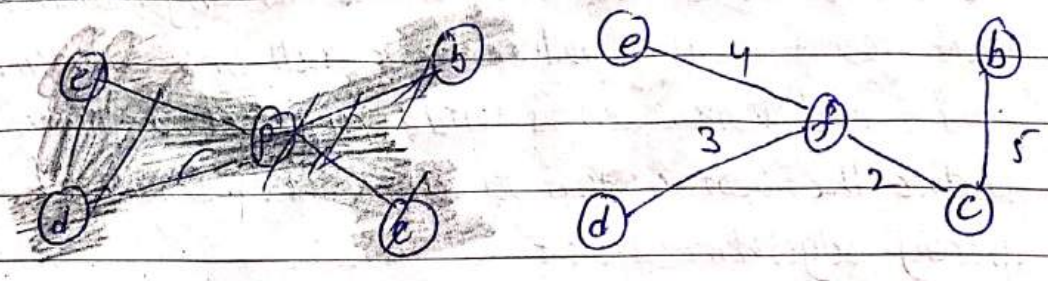
② Among all the remaining edges connected with vertices f and c, fd=3 is the shortest and it will not form cycle if it is included. So, we select fd.



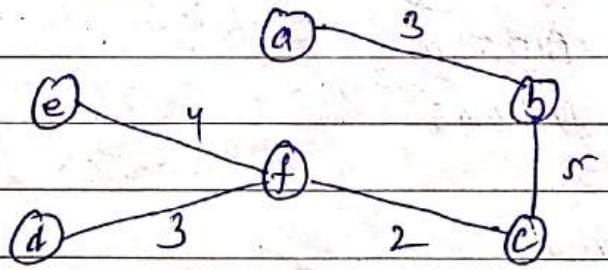
③ Among all the remaining edges connected with vertices f, c and d, ef=4 and ed=4 are the shortest. If we select ef and include it, it will not form cycle. So, we select ef. However we cannot select ed as it will form cycle when included.



④ Among all the remaining edges connected with vertices e, f, c and d, bc = 5 is the shortest and it will not form cycle if it is included. So, we select bc.



⑤ Among all the remaing edges connected with vertices e, f, d, c and b, ab = 3 is the shortest and it will not form cycle if it is included. So, we select ab



∴ Since, all the vertices are included and connected i.e. 6 vertices and 5 edges are included. So, the above tree is the minimum spanning tree and its total weight is 17.

Algorithm

① Select an edge with minimum weight and include it to the spanning tree.
② Among all the remaining edges which are adjacent with the selected edges, Select the edge with minimum weight and the tree formed doesn't form any cycle.
③ Repeat step 2 until all N vertices and N-1 edges are included where N is the No. of vertices in the graph.

7. Greedy Algorithm

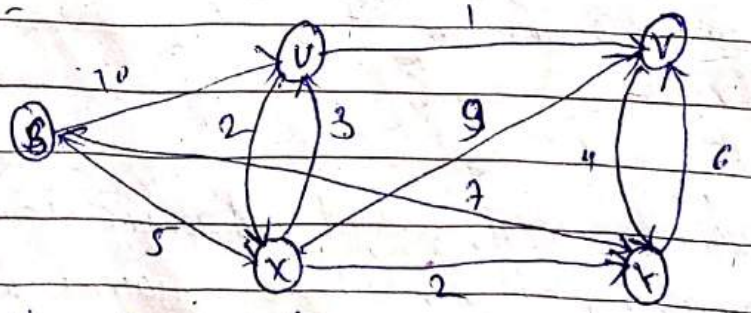
A greedy algorithm always makes the choice that looks best at the moment. It makes the best choice at any moment with the expectation that result will produce the overall best solution. It is the powerful concept and works well for wide range of application however it will not produce the best solution for all the cases. Some of the examples of greedy algorithm are:-

i) Kruskal's Algorithm
ii) Prim's Algorithm
iii) Huffman Coding Algorithm
iv) Dijkstra's Algorithm
v) Network Routing Algorithm
vi) Graph Coloring problems

8. Dijkstra's Algorithm

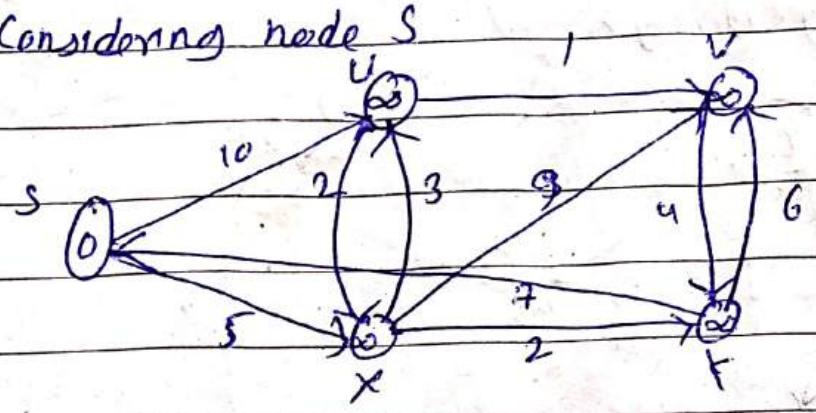
Dijkstra's Algorithm is a single source shortest path algorithm. It is used to find the shortest path on a weighted graph from a single source to all the vertices.

Example:-

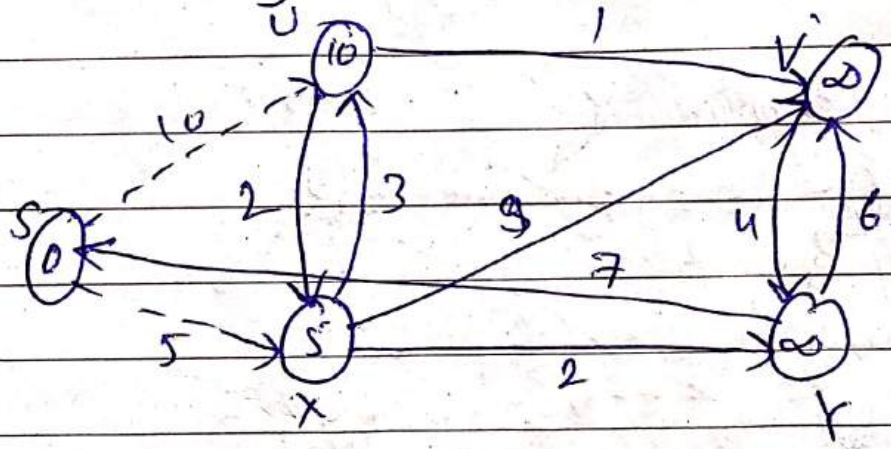


Find the shortest path from s to all other nodes.

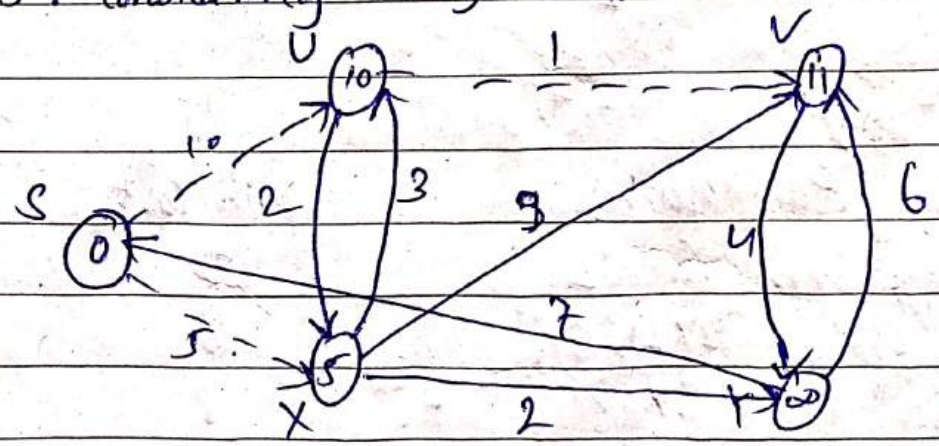
Step 1 :- Considering node S



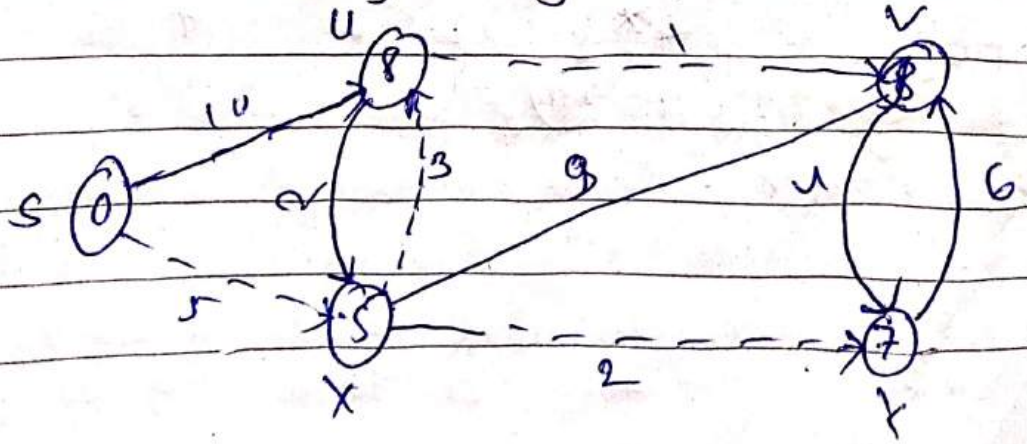
step 2 :- Considering the vertex connected with S (U and X)



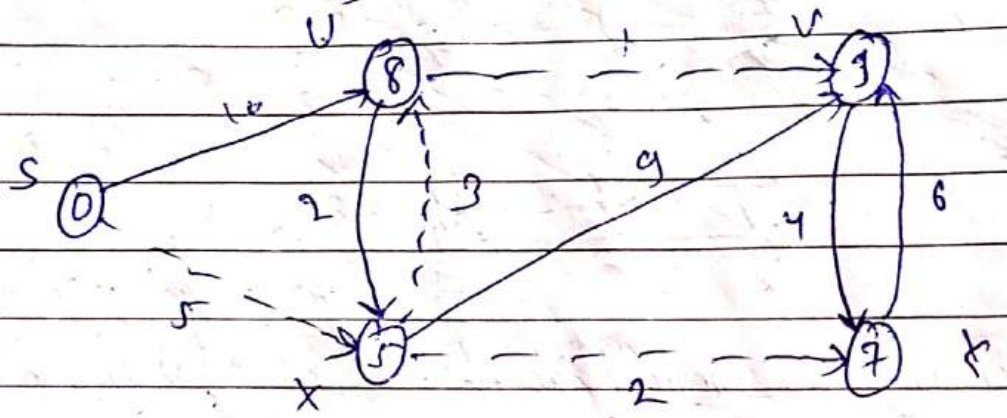
Step 3 :- Considering through U



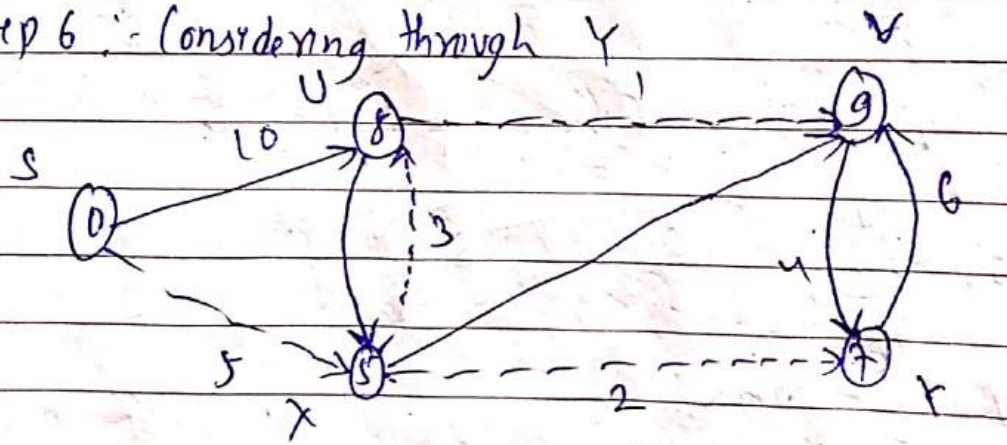
Step 4 :- Considering through X



Step 5 :- Considering through V



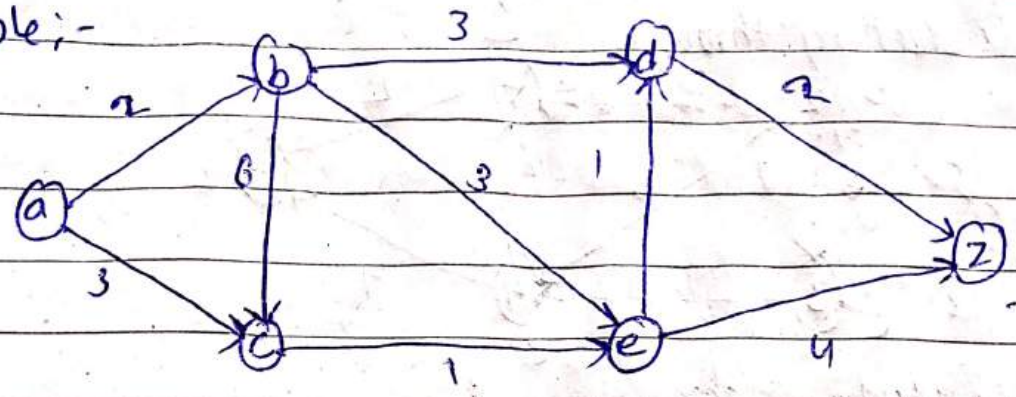
Step 6 :- Considering through Y



Therefore, Shortest path

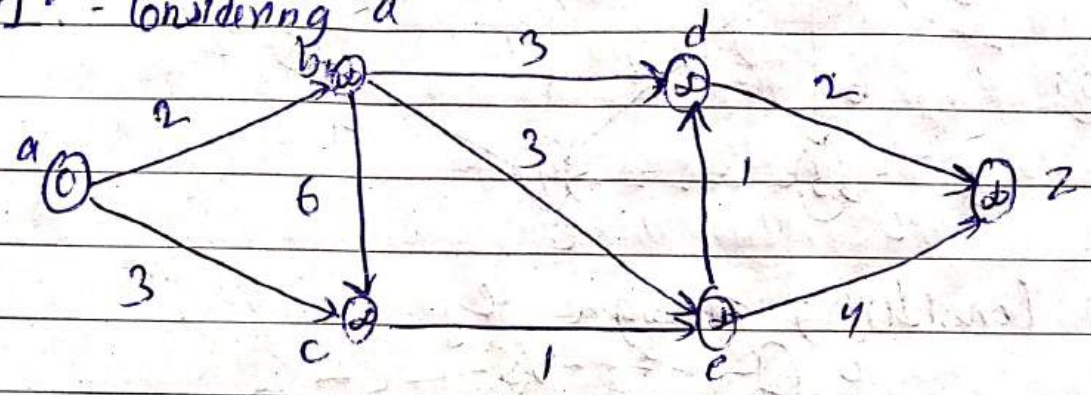
node	path	path cost
S to S	S	0
S to U	SXU	8
S to X	SX	5
S to V	SXUV	9
S to Y	SXY	7

Example:-

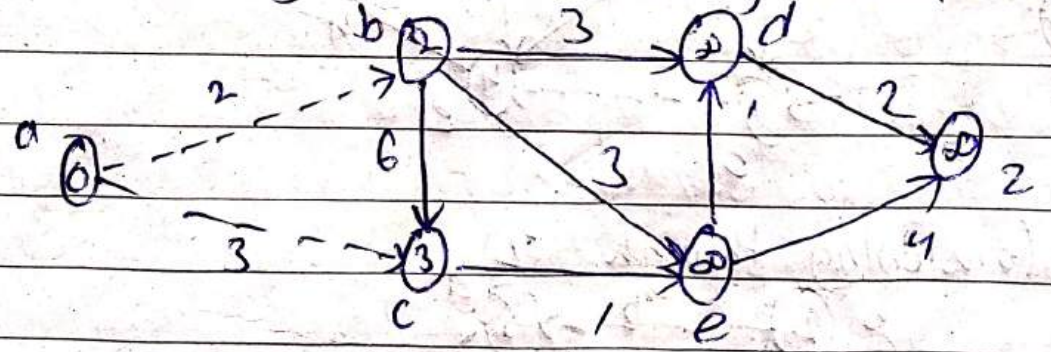


Find the shortest path from a to all other nodes:-

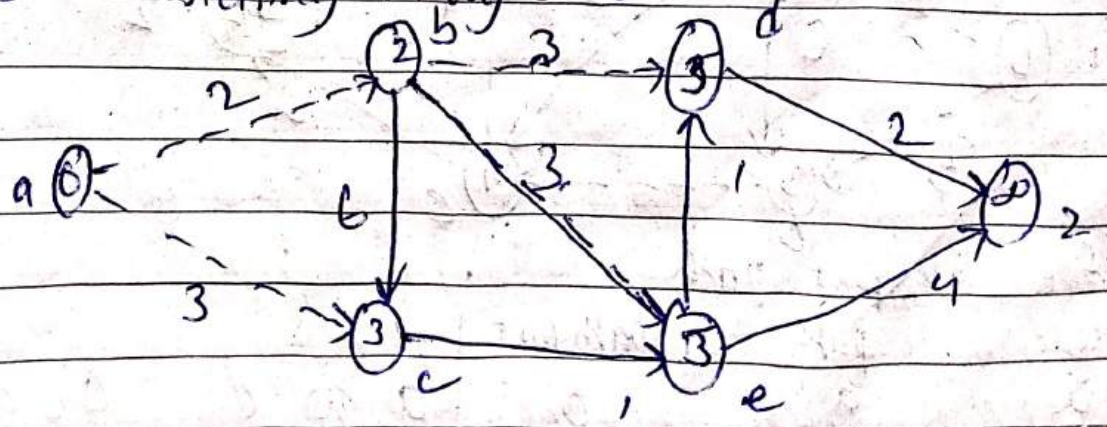
Step 1:- Considering a



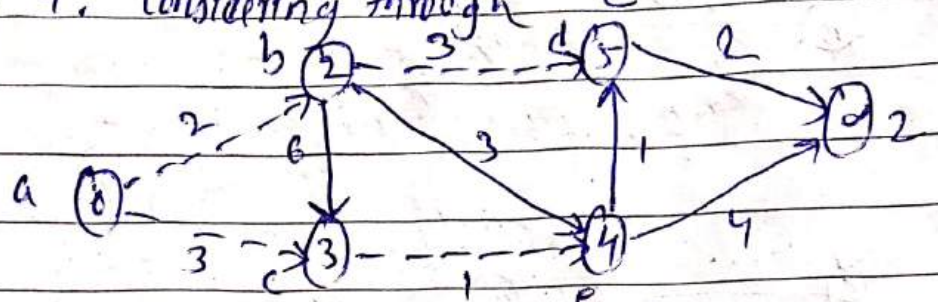
Step 2:- Considering the vertex connecting to a (b and c)



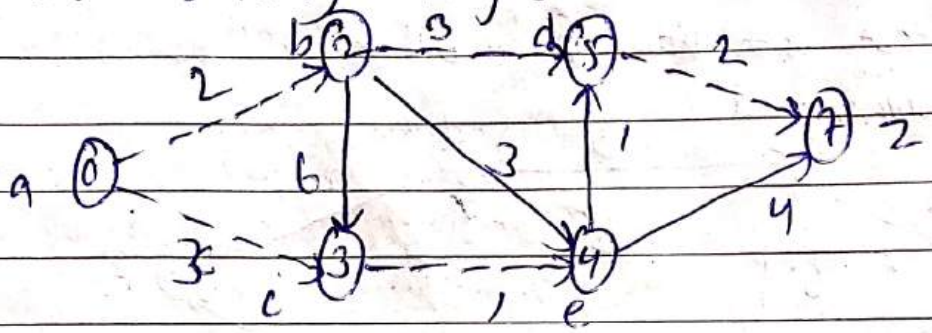
Step 3:- Considering through b



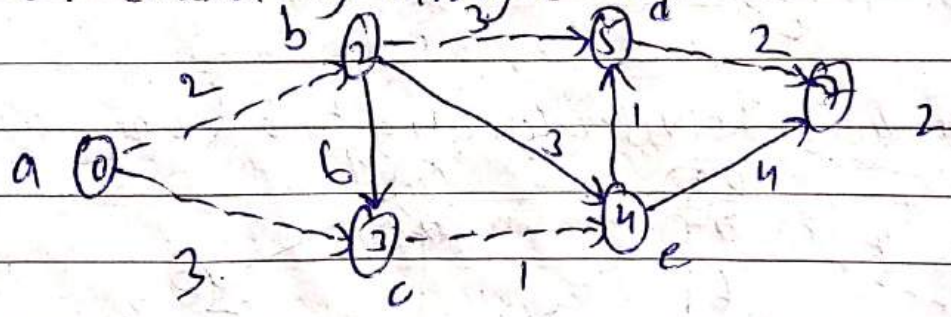
Step 4: Considering through c



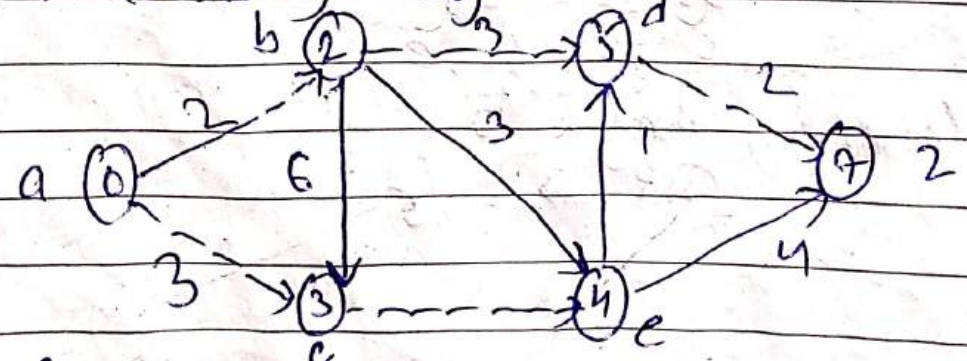
Step 5: Considering through d



Step 6: Considering through e



Step 7: Considering through 2



Therefore shortest path

Node	path	path cost			
a to a	a	0	a to d	abd	5
a to b	ab	2	a to e	ace	4
a to c	ac	3	a to z	abdz	7

Unit :- 11 Algorithms

1. Algorithm

An algorithm is a procedure or steps for solving a problem based on conducting a sequence of specified actions. It is the step by step description of a program.

Example:-

An algorithm to find the Sum of Two Numbers:-

Step 1:- START
Step 2: Accept the First Number (a)
Step 3: Accept the Second Number (b)
Step 4: Add these two Numbers (a+b)
Step 5: Display Result (sum)
Step 6: Stop

Types of Algorithms

a) Deterministic and Non-Deterministic Algorithm
b) Divide and Conquer Algorithm
c) Series and Parallel Algorithm
d) Heuristic and Approximate Algorithm

a) Deterministic and Non-Deterministic Algorithm

Deterministic Algorithm:-

- It is an algorithm that is purely determined by its inputs.
- Will always produce same output for particular given inputs.
- Can Determine the next step of execution
- Backtracking is allowed.

Example: for i from 1 to 9
print 9
Will always print 123456789

Non-Deterministic Algorithm:

- It is an algorithm that provides different outputs for the same inputs on different executions.
- Cannot determine the next step of execution.
- Backtracking is not always allowed.
- It is usually has two phases and output steps:-
  - → First phase is guessing phase, which makes use of Imaginary character to run the problem and
  - → Second phase is verifying phase, which returns True or False for chosen string.

S.N	Deterministic Algorithm	S.N	Non-Deterministic Algorithm
1.	For a particular input the computer will always produce the same output.	1.	For a particular input the computer will produce different output on different execution.
2.	It can solve the problem in polynomial time.	2.	It can't solve the problem in polynomial time.
3.	It can determine the next step of execution.	3.	It cannot determine the next step of execution due to more than one path the algorithm can take.
4.	Backtracking is allowed.	5.	Backtracking is not always allowed.

b) Divide and Conquer Algorithm

- In Divide and Conquer approach, the problem is divided into smaller sub-problems and then each problem is solved independently.
- When we keep on dividing the sub problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. These smallest possible sub-problems are solved and combined.
- Divide and Conquer algorithm solves a problem using three steps:-
  - i) Divide / Break :- Break the given problem into sub problems of same type.
  - ii) Conquer/Solve :- Recursively solve these sub problems
  - iii) Merge/Combine :- Appropriately combine the result.

Example :- Merge Sort, Quick Sort, Binary Search etc.

c) Serial and Parallel Algorithm

- Algorithms in which operations must be executed step by step are called Serial or Sequential algorithms.
- Algorithms in which several operations may be executed simultaneously are referred to as parallel algorithms.
- A parallel algorithm can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.
- In parallel algorithm, the problem is divided into sub-problems and are executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output.

- It is not easy to divide a large problem into sub-problems. Sub-problems may have data dependency among them. Therefore the processors have to communicate with each other to solve the problem.
- It has been found that the time needed by the processors in communicating with each other is more than the actual processing time. So, while designing a parallel algorithm, proper CPU utilization should be considered to get an efficient algorithm.
- To design an algorithm properly, we must have a clear idea of the basic model of computation in a parallel computer.

d) Heuristic and Approximate Algorithm

- A Heuristic technique, is an approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect or rational, but which is nevertheless sufficient for reaching an immediate, short-term goal.
- It is a technique designed for solving a problem more quickly when classic method are too slow or for finding an approximate solution when classic methods fails to find any exact solution.
- Example : Travelling Salesman Problem, Searching Algorithms

[Note:- Meaning of heuristics - enabling someone to discover or learn something for themselves]

— Approximation algorithms are efficient algorithms that find approximate solutions to the problems with provable guarantees on the distance of the returned solution to the optimal one.
— The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time.
— Example: Minimum Vertex Cover problem, Knapsack problem
— A heuristic is typically a bunch of intuitive steps that may or may not lead you an optimal solution. An approximation algorithms, on the other hand, is equipped with a formal promise of being reasonably close to an optimal solution.