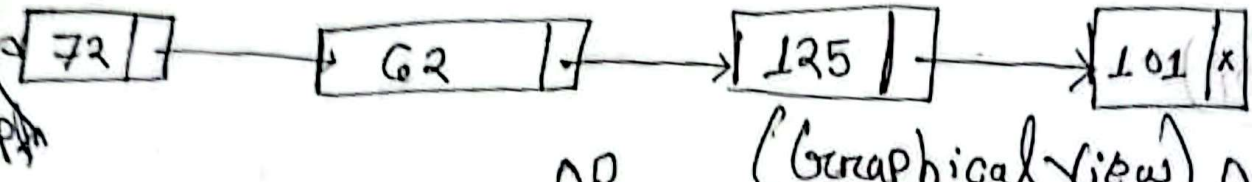
DS (Theory):

Lec-01:

Linear (Singly) Linked list:

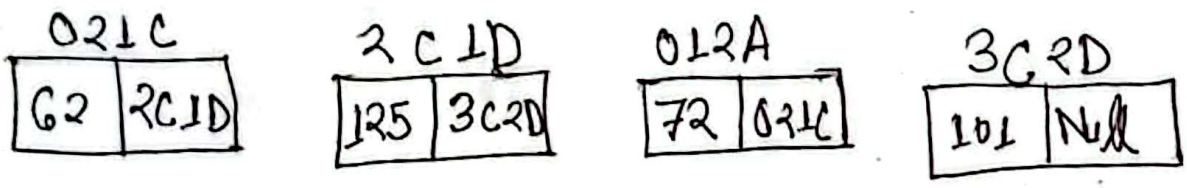


OR (Graphical View) of a Linked list

- Null mane Last Node
- → first Node

} I. For MCQ

OR



- Any element of Linked list ~~ie~~ is called a node.
- A node has two parts. One is data (value) part and another is pointer (address) part.

- A set (collection or list) of some nodes is called linked list.

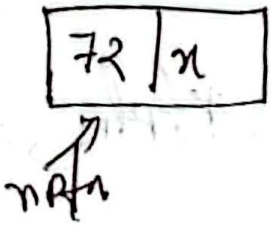
Linked List using structure:

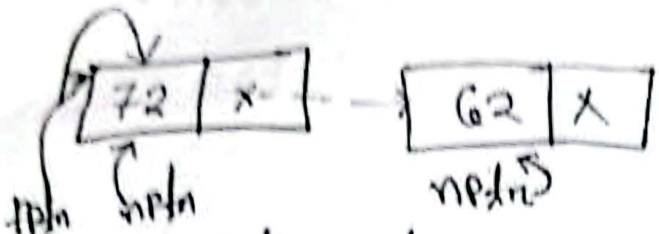
```
struct node{
    int data; // for value
    node *next ; // pointer
};
```

- This type of structure is called self reference structure.

Storing data to a node:

```
node *nptr;
nptr = new (node);
nptr -> data = 72
nptr -> next = null;
```






```
fptr = nptr;
fptr -> next = nptr;
```

- Simulation on Graphical view to Create a Linked list → (see book)

Pseudo-code → (see book)

Code:

```
struct node{
    int data;
    node *next;
};
node *nptr, *tptr, *fptr;
int value;
nptr = new (node);
nptr -> data = value;
nptr -> next = NULL;
fptr = nptr;
```

outside the loop

within the loop

Operations on linked list:

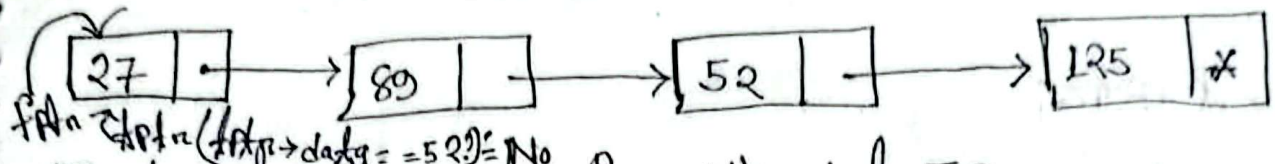
(1) Searching or locating
(2) Insertion (Addition)
(3) Deletion
(4) Traversal

~~3. Rm:~~

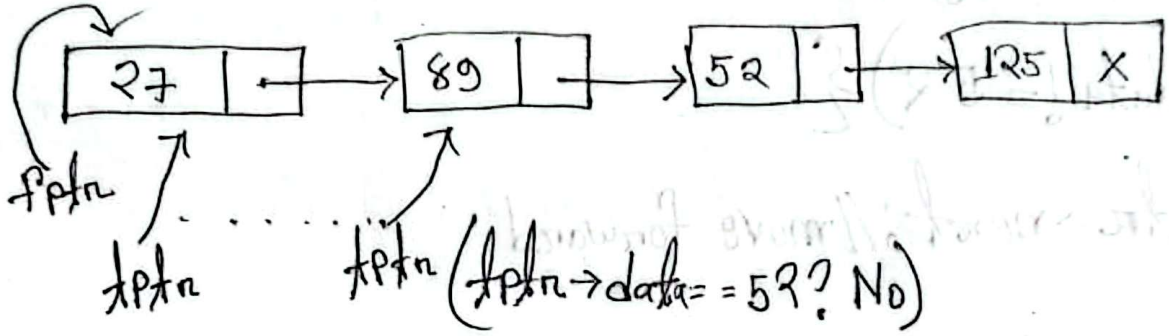
DS (Theory):

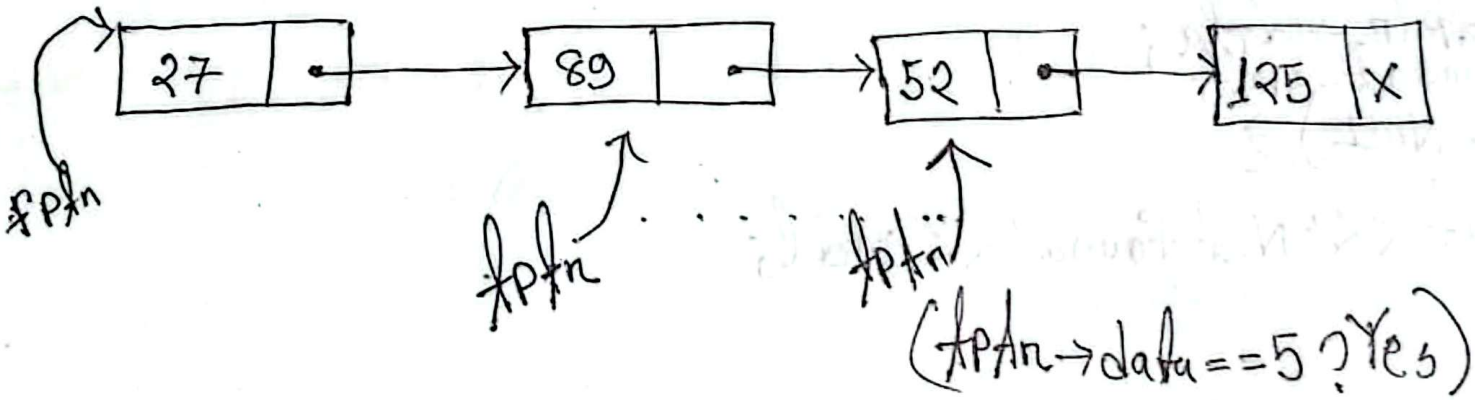
Lec-02:

Searching (Locating):



Find out the node with data 52.

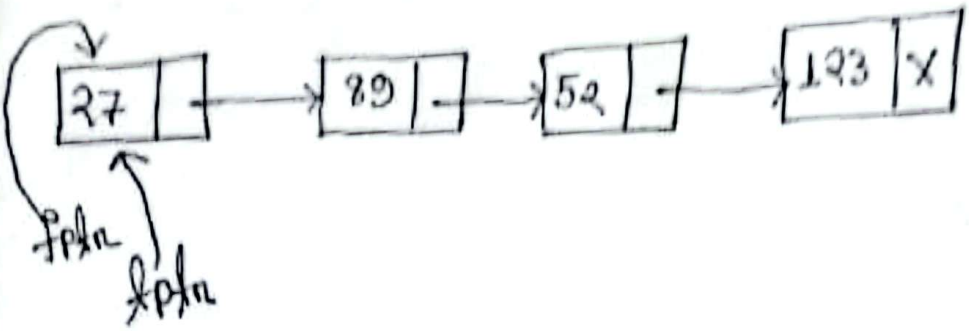




Pseudo-code:

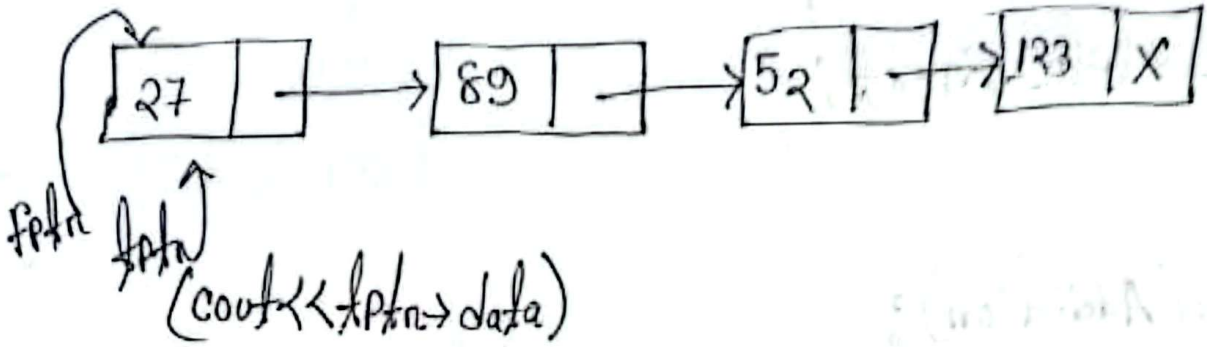
```
struct node {
    int data;
    node* next;
};
node *fptr, *tptr;
tptr = fptr; //Starting
int Pos=0;
while (tptr->data != 52){
    tptr = tptr->next;//move forward
    Pos++;
}
cout<<tptr->data;
cout<<"found at"<<Pos;
if (tptr == NULL) {
    cout<<"Not found"<<endl;
}
```

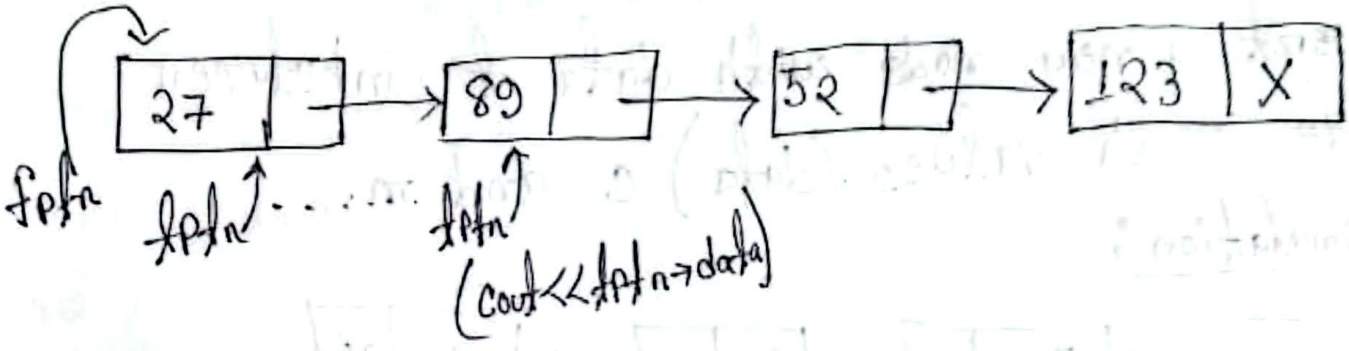
[4] Traversal:

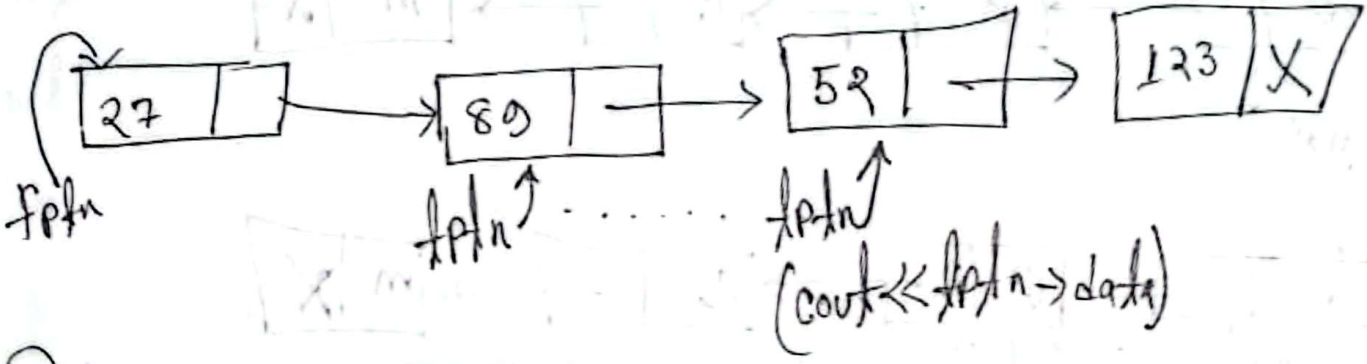


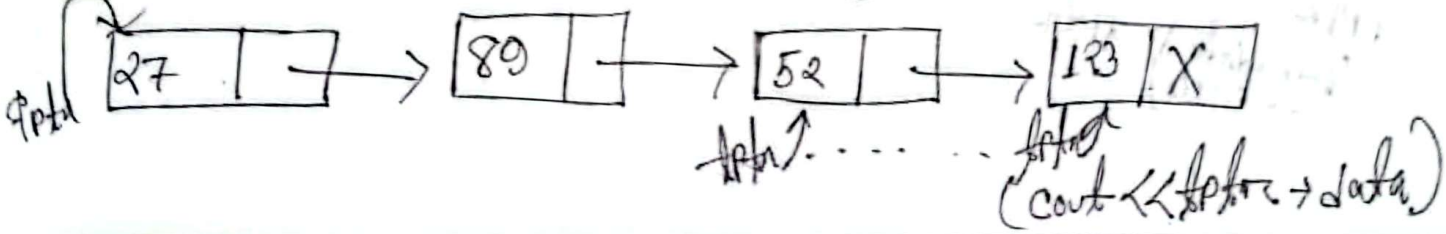
Q: Print all the data of LL.

Simulation:









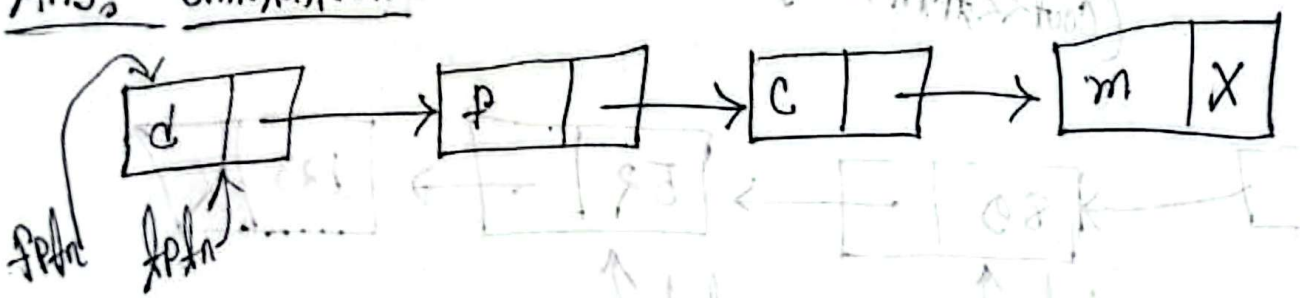
Pseudo-Code:

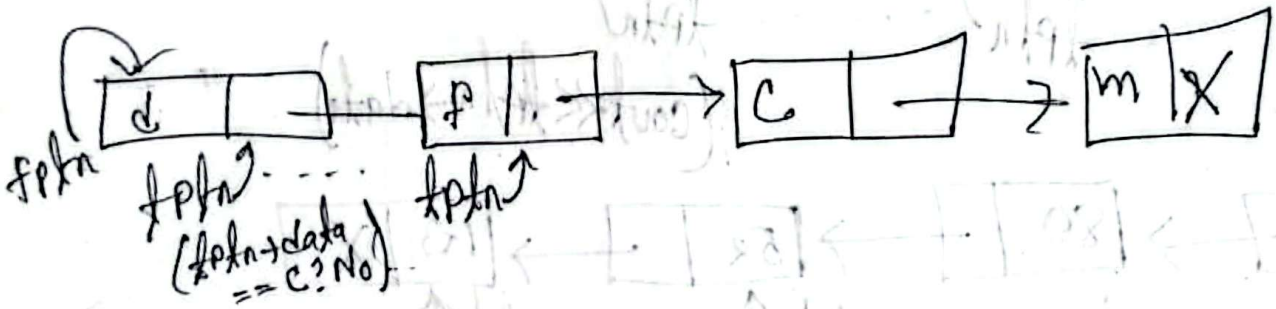
```
struct node{
        int data;
        node* next;
};
node *fptr, *lptr;
  while (fptr != NULL){
      cout << fptr -> data;
      fptr = fptr -> next;
  }
```

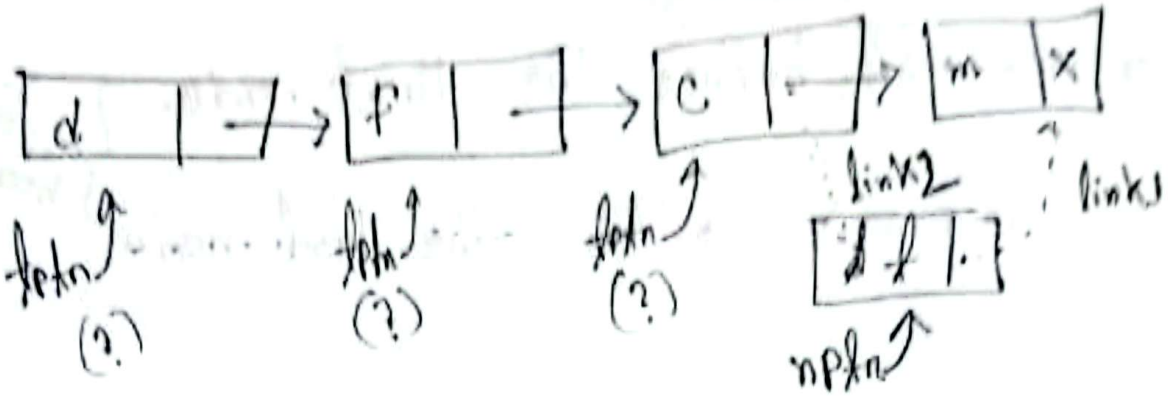
Insertion (Addition):

Q: Insert a new node with data f inbetween the node with values (data) c and m.

Ans: Simulation:







Pseudo-Code:

```
struct node {
    char data;
    node * next;
};
node *fptr, *tptr, *nptr;
nptr = new(node);
nptr -> data = 't';
nptr -> next = NULL;
while (tptr -> data != c){
        tptr = tptr -> next
}
nptr -> next = tptr -> next; //li
tptr -> next = nptr; //link2
```

Home Work

Q: Add a new node before the first node.

Q: Add a new node after the last node.

} see book

Deletion:

Step:

(1) locate the node to be deleted

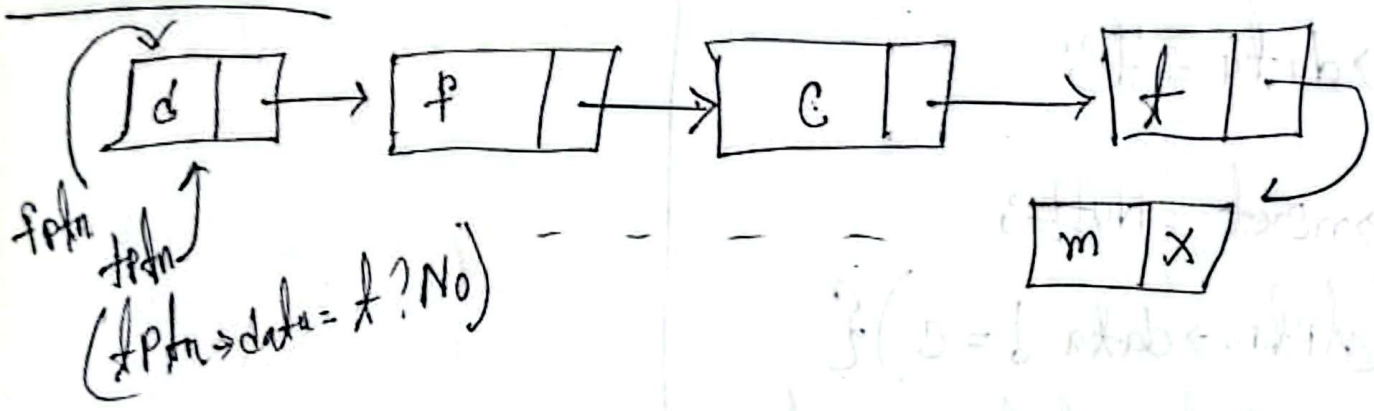
(2) Establish necessary links.

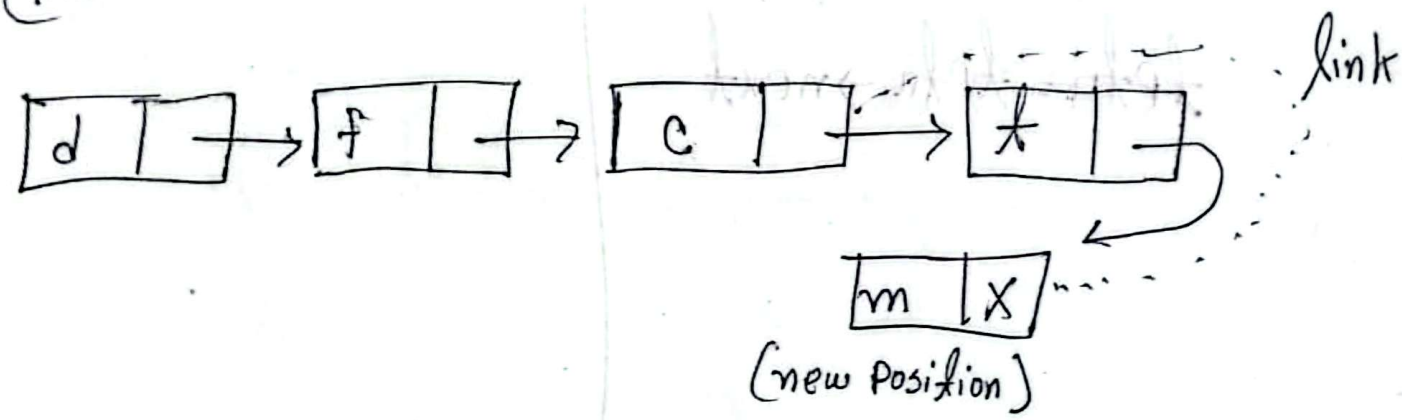
(3) Delete the node

Q: delete the node with data f.

Ans:

Simulation:





DS (Theory):

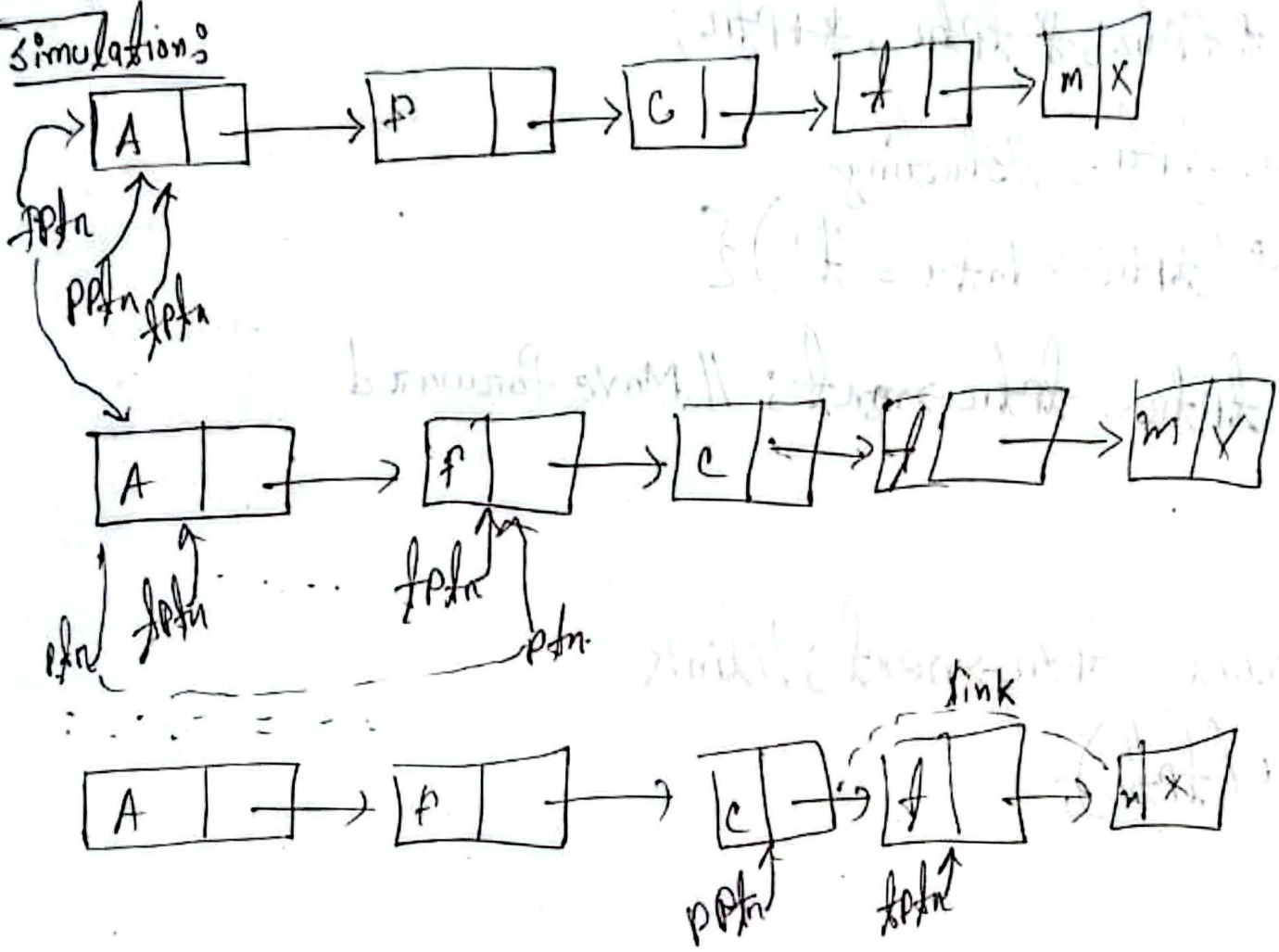
Lec-03:

Deletion:

Q: Deleteion the node with data 'f'

Ans:

Simulation:



link: pptr → next = fptr → next

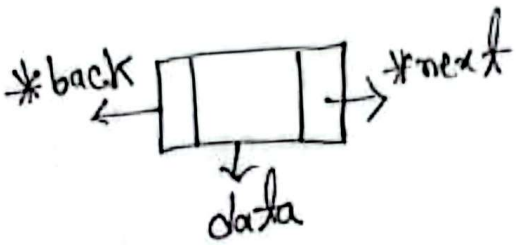
Pseudo-Code:

```
struct node {
    Char data;
    nod* next;
};

node *fptr, *tptr, *pptr;

tptr = fptr; //starting
while (tptr -> data = 't') {
    tptr = tptr -> next; // Move forward
}

pptr -> next = tptr -> next; //link
delete (tptr);
```

Q: Delect the First node of a LL.

Q: Delect the last node of LL.

Doubly linked list (two-way linked list):



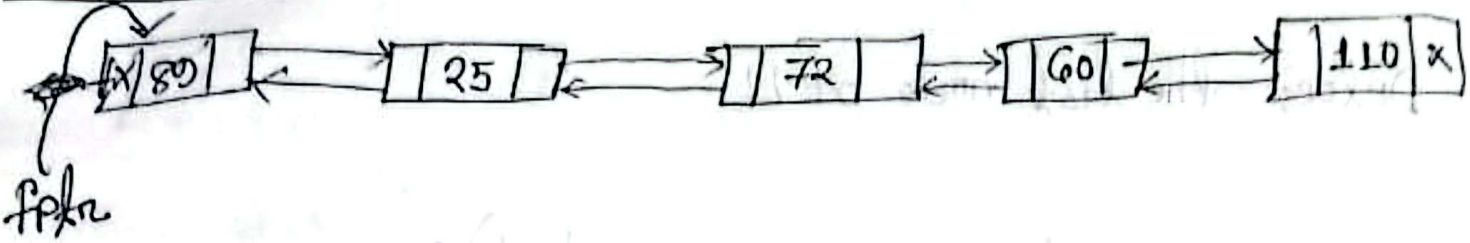
Node:

(1) data part
(2) Pointer to the next node
(3) Pointer to the Previous node

Node declaration:

```
struct node{
    int data;
    node *next;
    node * ~~Prev~~ back;
};
```

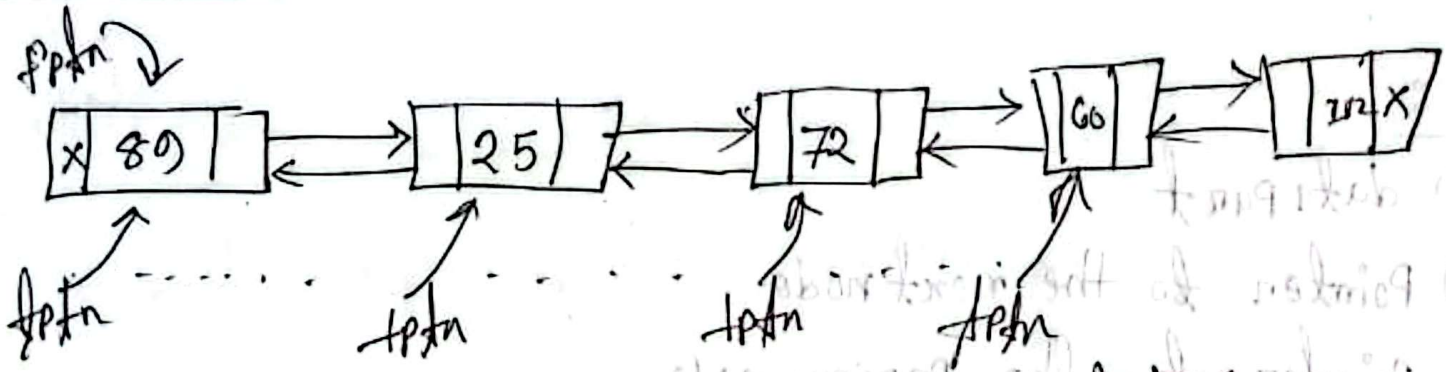
Deleteion :

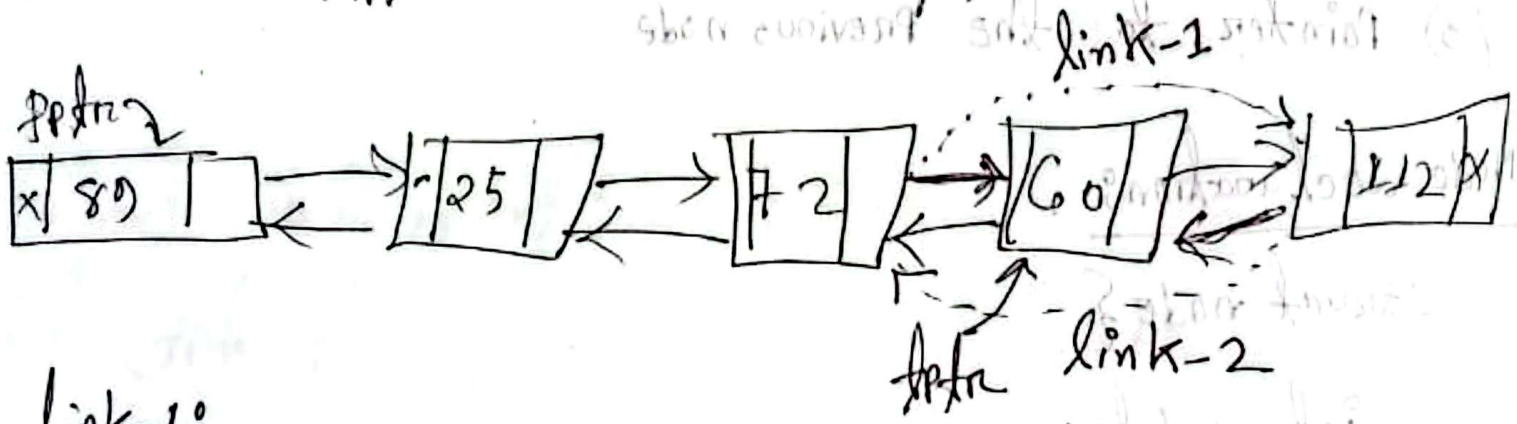


Q: Delect the node with Value 60.

Ans:

Simulation:





link-1:

```
fptr->back->next = fptr->next;
```

link-2:

```
fptr->next->back = fptr->back;
```

Pesudo-Code:

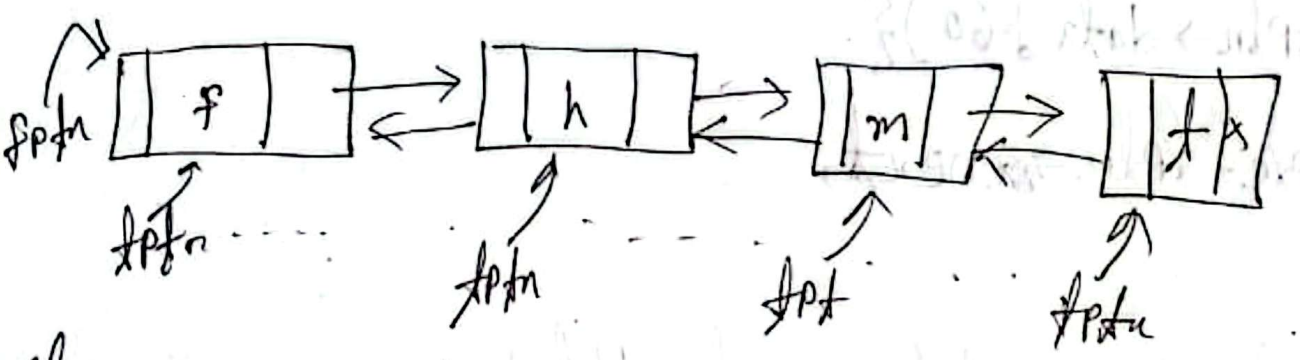
```
struct node{
    int data;
    node *next;
    node* back;
};
node * fptr, tptr;
tptr = fptr // starting
while(tptr->data != 60){
        tptr = tptr->next;
    }
tptr->next->back = tptr->next; //link-1
tptr->back->next = tptr->back; //link-2
delect(tptr);
tptr->back->next = tptr->next;
tptr->next->back = tptr->back;
delect(tptr);
[Searching and Travarsal same as LL]
```

Traversal:

*back → NULL (1st node)
*next → null (last node)

Q: Print all the data of DLL in reverse order. Suppose that there is no pointer at last node.

Ans:



Step-d:

(1) Locate Last node

(2) Print the data one by one and move backward.

Pseudo-code:

```
struct node {
        char data;
        node* next;
        node* back;
};
node* fptr, *lptr;
  lptr = fptr;
while(lptr->next != NULL){
        lptr = lptr->next;
  }
while (lptr != fptr){
        cout << lptr->data;
        lptr = lptr->back;
  }
```

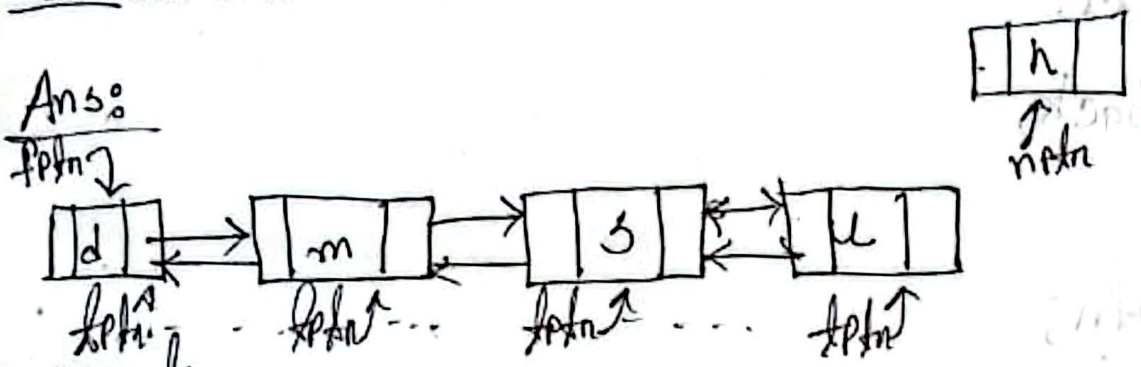
Q: reverse ~~the~~ singly LL. (Homework)

Addition:

Insert a new node:

Q: Insert a new node with data 'h' at the last node.

Ans:

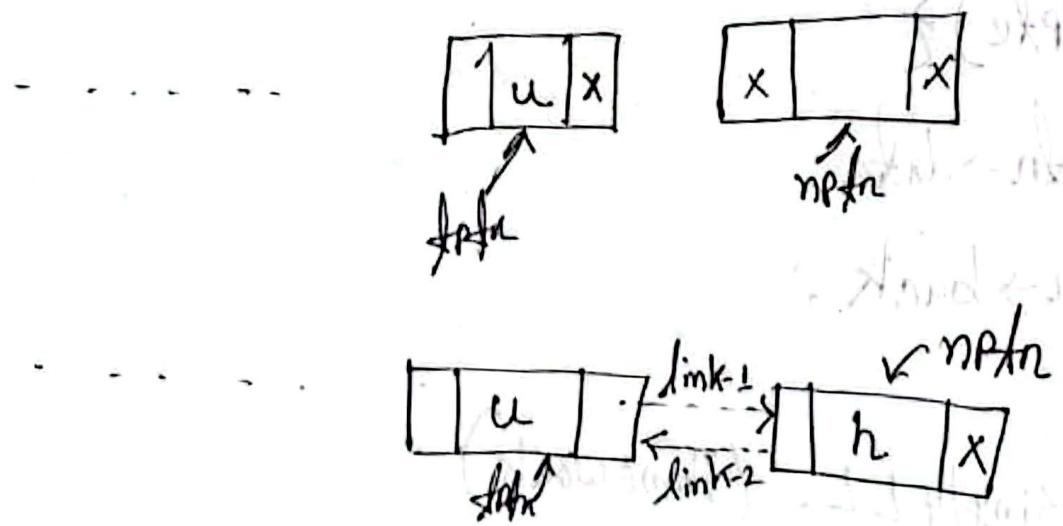


Operations:

(1) Create a new node with data h.

(2) locate the last node.

(3) Make links



Pseudo-Code:

```
struct node{
    char data;
    node * next;
    node * back;
};
nod * fptr, * tptr, * nptr;
tptr = fptr; // starting
while (tptr->next != NULL){
        tptr = tptr->next;
}
nptr = new (node);
nptr -> data = 'h';
nptr -> next = NULL;
nptr -> back = NULL;
tptr -> next = nptr;
nptr -> back = tptr;
```

Q: Add a new node before the first node

Q: Add a new node in between two existing nodes.
(See book)

DS (Theory):

Lec-04:

Q: Print all the data of a DLL in reverse order.

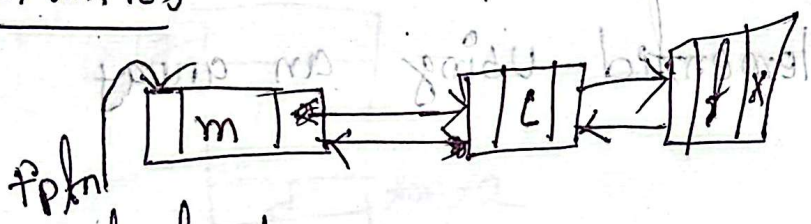
Ans:

Operations:

(1) Locate the data node.

(2) Print and move backward.

Example:



Output: f, c, m

Q: Print all the data of a singly LL in reverse order

Forward movement: fptr = fptr → next

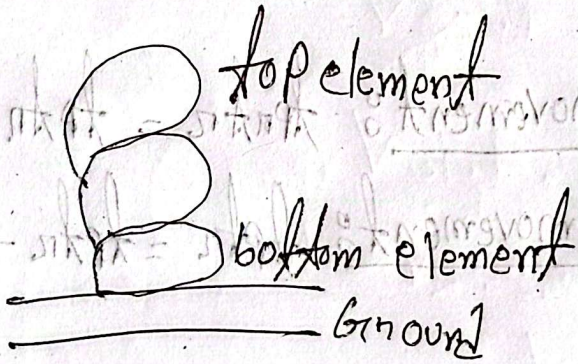
Backward movement: fptr = fptr → back

Stack: (LIFO)

- It is an distract data structure.
- It has a special indicator (index) or pointer ~~or~~ called top.
- Any element is added at top of the stack.
- Any element is deleted (accessed) from the top of the stack.
- A stack can be implemented using an array or a LL.

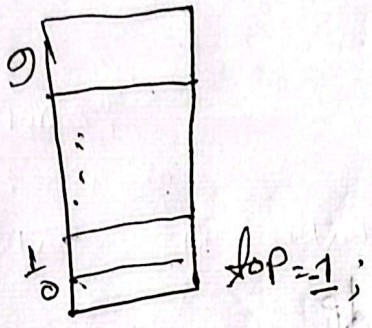
Simple Concept:

- Stack of books
- Stack of Chairs
- Stack of Plates

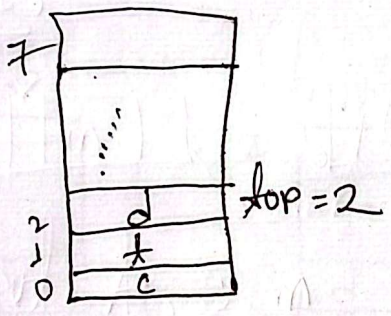


Elements: int, char, float, etc.

Array based stack:



- above stack is empty.



- There are three elements in the stack.

Operations or ~~Functions~~ Functions:

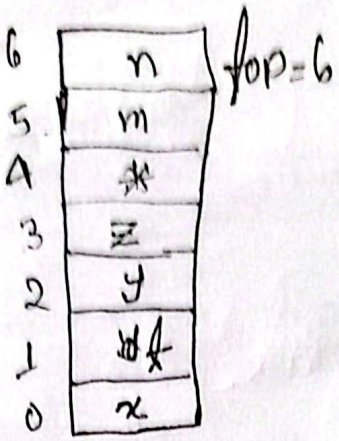
(1) Push or addition

(2) Pop or Remove

(3) Empty stack or underflow

(4) Full stack or overflow

(5) Show stack or Printall



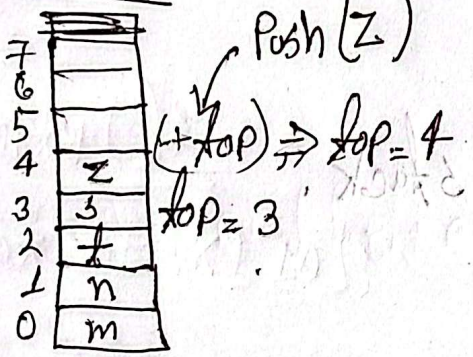
Overflow

- above stack is Full

```
Char stack[7];
int size=7; int top;
if(top == size-1){
    cout << "Full";
}
```

Pseudo-code

Push( ):

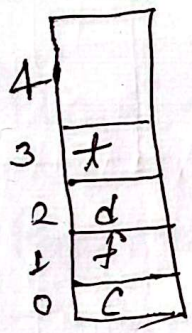


Pseudo-Code:

```
Char st[8];
int size=8, top;
Char element;
```

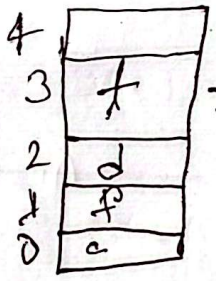
```
Push() {
if(top==size-1){
    cout<<"stack is full";

else{

    ++top;
    st[top]=element; // for the stack

}

}
```

Pop:



s[ ];
Before Pop()

⇒



top=2 (--top)

after Pop( )

Pseudo-Code:

```
Char s[5];
int size=5, top;
Char element;
```

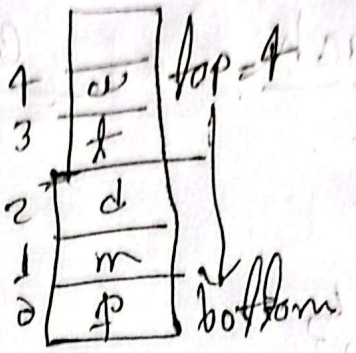
```
Pop(t)
{
if (top == -1){
    cout << "stack is empty";

else{
    element = s[top]; // From the stack
  --top;
}

}
```

Empty(){

```
  int stack[10];
int top;
  if (top == -1){
      cout << "stack is empty";
  }
}
```

Showstack:

```
Showstack(){

  Char st[8];
  int size=8;
  int top;
  int i;
  for (i=top i>=0; i--){
      cout << st[i];
  }
}
```



st[ ]

output: w, t, d, m, p

Applications of stack:

(1) Parenthesis checking (validation)

(2) Infix to Postfix Conversion

(3) Arithmatic expression evaluation

Parenthesis checking:

Q: [{(a+b)-d}-e]

Ans:

Symbol	stack	Comments/operations
[	[	Push([)
{	[, {	Push({)
(	[, {, (	Push(()
a	[, {, (	~~Push~~ none
+	[, {, (	none
b	[, {, (	none
)	[, {,	Pop(() & matching (ok)
-	[, {	none
d	[, {	none
}	[	Pop({) & matching (ok)
+	[	none
e	[	none
]		Pop([) & match(not matching)

Result: Stack is empty but matching is not okay so the expression is not valid.

Note:
- Push () to the stack.
- Pop ( ) from the stack.
- Push () bottom to top.
- Pop () top to bottom.

Symbol("(,{,[")	Stack	Operation
[	[	Push ([)
{	[, {	Push ({)
)	[, {, (	Push (()

Note:
- Stack must be empty and all matching must be okay, then expression is valid. otherwise invalid.

{(a+b)+c

Stack

{, (

{ (stack is not ~~not~~ empty)

Infix to Postfix:

(1) a+b infix

(2) +ab Prefix

(3) ab+ Postfix

Input: a+b

Output: ab+

$\underbrace{+, -}_{3rd}$ $\underbrace{*, /}_{2nd}$, $\underset{1st}{exp(Power)}$

DS (Theory):

Lec-05:

Q-1: Monday (19-05-25)

Infix to Postfix Conversition:

Exp: 2*5 + (8-3)

Ans:

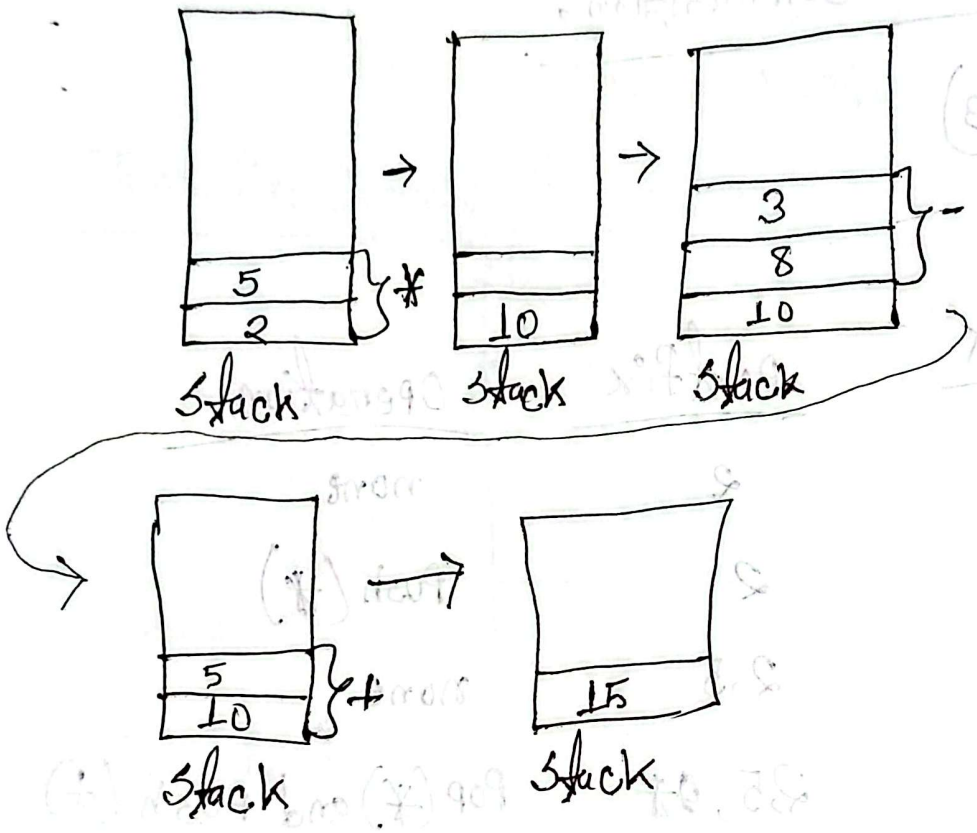
Symbol	Stack	Postfix	Operation
2		2	none
*	*	2	Push (*)
5	*	2,5	none
+	+	2,5,*	Pop (*) and Push (+)
(	+,(	2,5,*	Push (( )
8	+,(	2,5,*,8	~~Push~~ none
-	+,(,-	2,5,*,8	Push (-)
3	+,(,-	2,5,*,8,3	none
)	+	2,5,*,8,3,-	Pop (-) and Remove '('
		2,5,*,8,3,-,+	Pop (+)

Infix: 2*5+(8-3)

Postfix: 2, 5, *, 8, 3, -, +

Postfix evaluation:

Postfix = 2, 5, * 8, 3, -, +



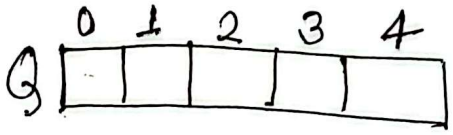
Pseudo-Code:

(1) Parenthesis checking
(2) Infix to postfix
(3) Evaluation

} See Book

Queue: ~~(LIFO)~~ (FIFO)

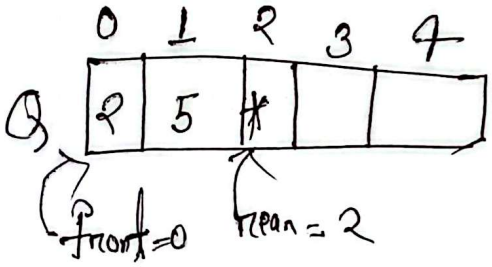
- It has two indicator or index, One is front and another is rear.
- A queue can be implemented using an array or a LL (Not needed)
- front is used to delete (Remove) the data and rear is used to add data.

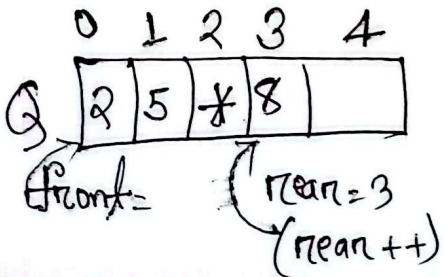


front = -1;

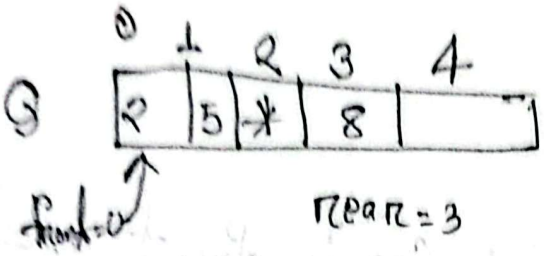
rear = -1;

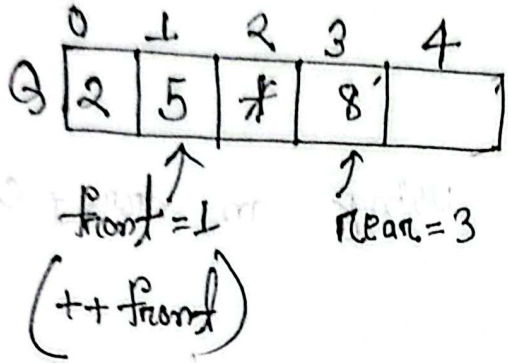
This Queue is empty



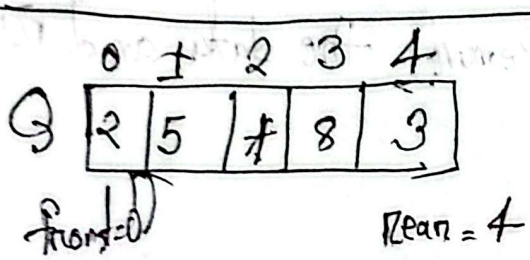


(Enqueque or addition)





(Dequeue or Deletion)



- The above Queue is full

Operations (Function):

(1) Empty Q( )

(2) Full Q( )

(3) En Q( )

(4) De Q( )

(5) Show Q( )

Pseudo Code:

Empty Q:

~~Pseudo-Code:~~

```
Char Q[5];
front = -1 , rear = -1
EmptyQ ( ){
    if (rear = = -1){
        cout << " Q is empty";
    }
}
```

Full Q:

~~Pse~~

```
Char Q[5];
int front , rear ; int size;
size = 5;
FullQ ( ){
    if (rear = = size -1) {
        cout << " Q is Full";
    }
}
```

```
if (rear = = -1){
    ++rear;
    Q[rear] = element;
    front = rear;
}
```

EnQ:

~~EnQ~~

```
Char Q[5];          | Char element;
int front, rear;    |
int size = 5;       |
EnQ(){
if (rear < size-1){
    ++rear;
    Q[rear]=element;
}
else{
cout << "Q is Full";
}
```

DeQ :

```
Char Q[10];
int front, rear;
int size=10;
```

```
Char x;
```

```
DeQ ( ){
 if (rear = front){
      ~~rear = -1~~;
      ~~front = -1~~;
      x = Q[front];
      rear = -1;
      front = -1;
 }
 if (rear > front){
      x = Q[front];
      ++front;
 }
 if (front == -1){
      cout << "Q is empty";
}
```

Show Q:

0	1	2	3	4	5	6
c	d	~~f~~	e	f	~~f~~	~~X~~

front = 0 rear = 5

```
Char Q[6];
int size = 6
int front, rear;
int i;

Show Q(){
~~for~~ for(i = front; i ≤ rear; i++){
        Cout << Q[i];
```

Ex: Print Reverse. [HW]

```
}
}
```

Ex:

	0	1	2	3	4	5
Q	g		c	d	e	f

front=0 rear=2

Print Now?

DS (Theory):

Lec-06:

Circular Queue:

Operations:

(1) EmptyQ()

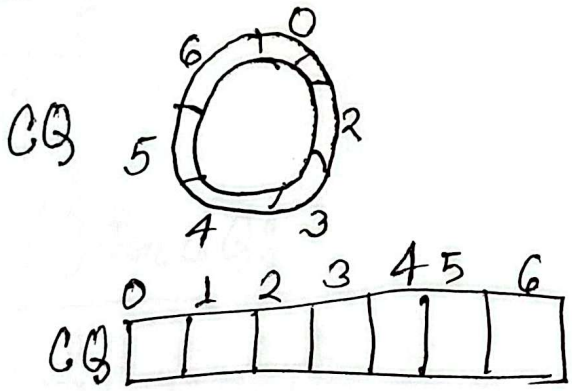
(2) FullQ()

(3) EnQ( )

(4) DeQ()

(5) showQ()

(1) EmptyQ:



rear = -1;
front = -1;
} This is the condition to check whether the Q empty.

Pseudo-code: Same as Linear queue.

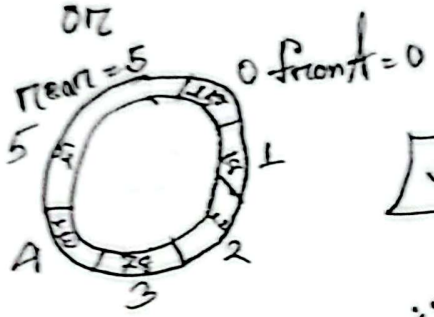
FullQ():

0	1	2	3	4	5	
15	18	22	78	110	52	X

rear = 5

front = 0

or



size = 6

```
if ((rear+1) % size == front) {
        cout << "CQ is full";
}
```

For Full Pesudo-Code is linear-queue and use above Condition.

EnQ() for CQ:

(1) CQ

0	1	2	3	4
b	a	f	h	

front = 0 rear = 3

(2) CQ

0	1	2	3	4
		p	r	q

front = 2 rear = 4

EnQ() ⇔ rear
DeQ() ⇔ front

Pseudo-Code:

```
Char CQ[5];
int rear, front, size;
size=5; Char ch;
EnQ(){
if((rear+1)%size == front){
        cout << "CQ is full";
}
else{
  rear=(rear+1)%size;
  CQ[rear]=ch;
}
}
```

DeQ() for CQ:

0	1	2	3	4	5
t	u	r			z

rear=2 front=5

Pseudo-code:

```
Char CQ[6];
int front, rear;
int size=6;
Char xa;
```

```
DeQ(){
if (rear == -1 && front == -1){
  cout<<" Queue is empty";
}

~~else~~{
  else if (front == rear){
    ~~cout<<~~
      xa = CQ[front];
      front = rear = -1;
  }

  else{
    ~~front = (front+1)%size~~
    xa = CQ[front];
    front = (front+1)%size;
  }

}
```

Show Q() §:

	0	1	2	3	4	5	6
CQ	125	77	88			75	98

rear=2 front=5

Print all data of the above queue.

Ans: Try yourself

Q: Showfront(), what does it mean?

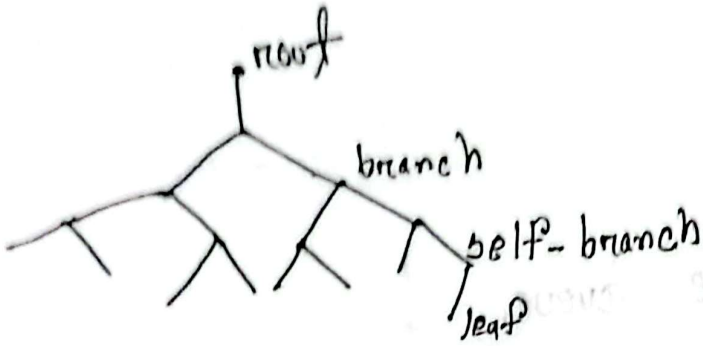
Ans:

```
cout << CQ[front];
```

or

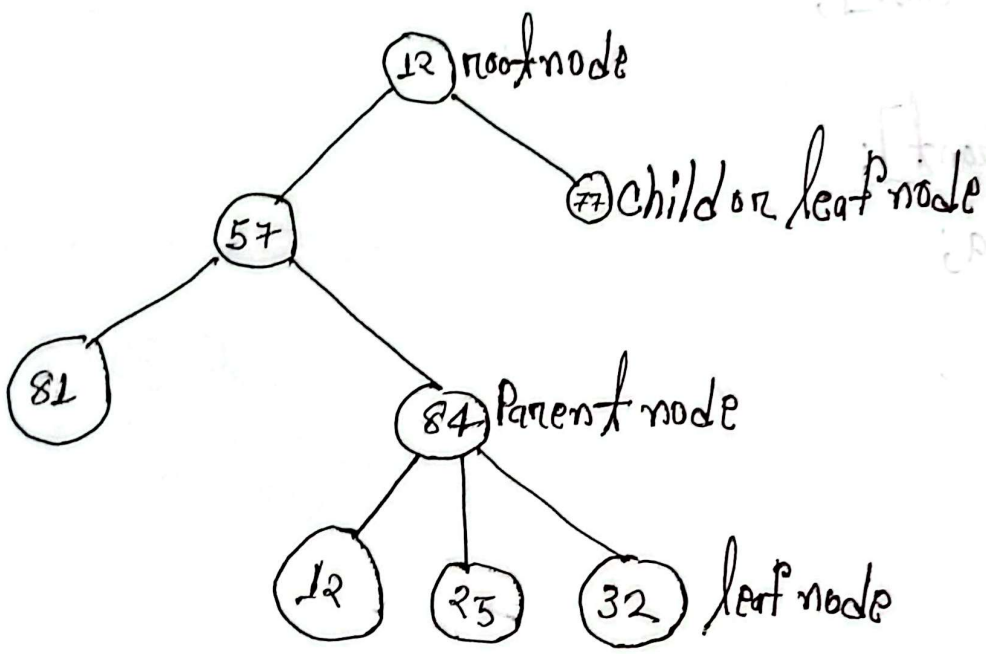
```
xa = CQ[front];
cout << xa;
```

Tree :



top-down figure

Element : integer, char, Float;



A tree of integer data

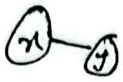
leaf node → no child
Child node → has Parent
root node → has no Parent

(1) (x) Tree

(2)

Tree

all are tree

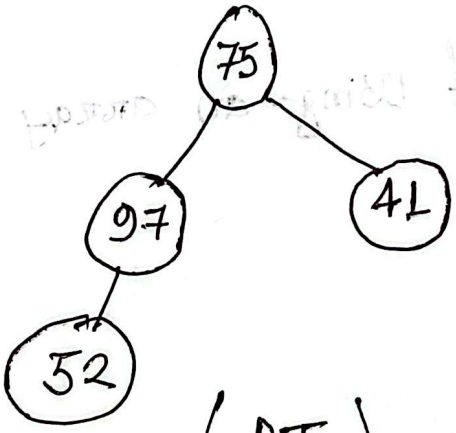
(3) 

Tree

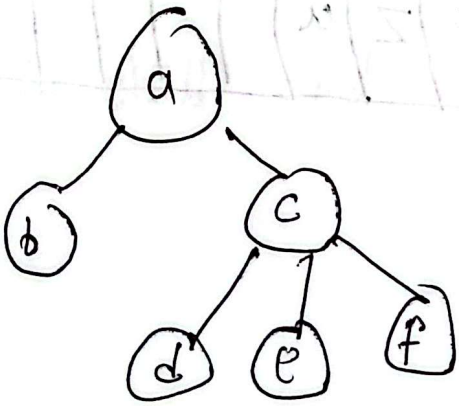
Binary Tree: Any node can have almost 2 children.

(1)

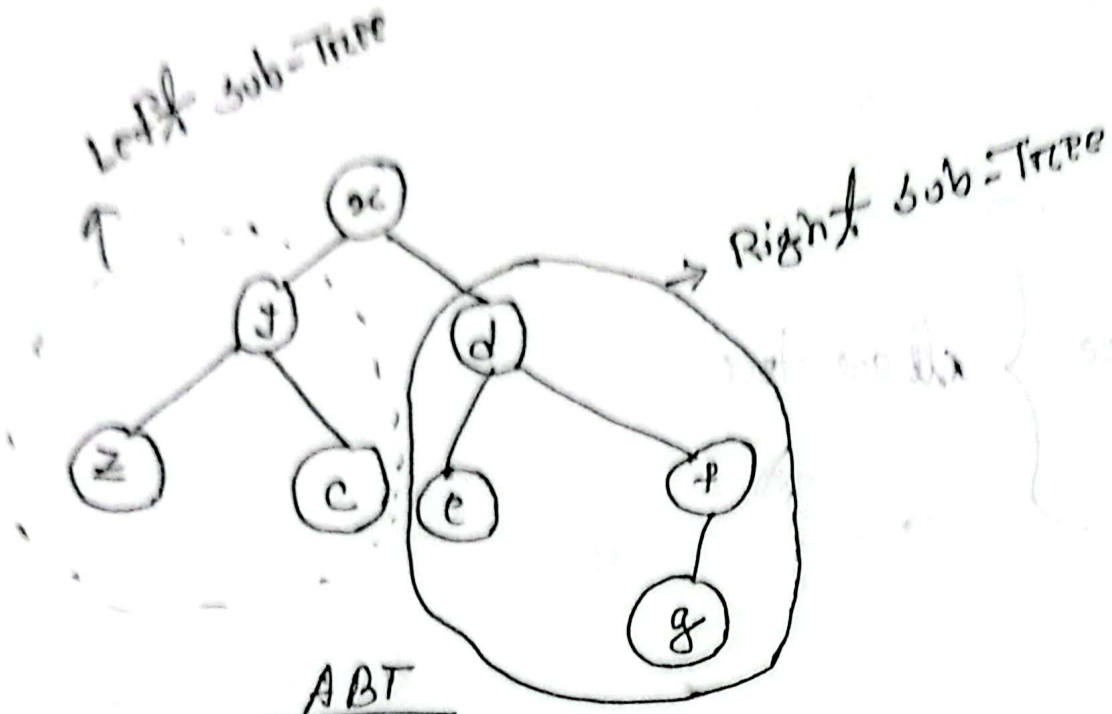
(Binary Tree (BT))

(2) 

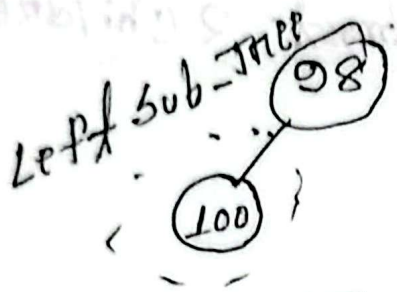
(BT)

(3) 

(Not BT)



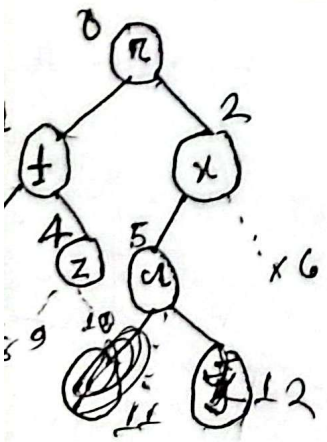
ABT



ABT

- Tree or BT can be implement using an array or a LL.

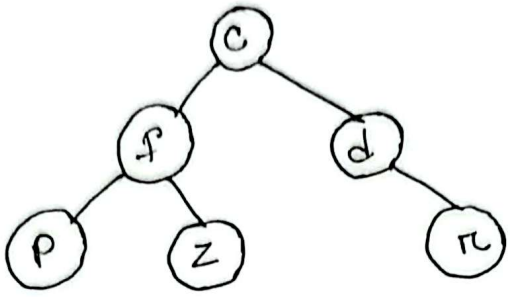
Array representation of BT:



⇒

0	1	2	3	4	5	6	7	8	9	10	11	12
r	f	x	y	z	u		-					f

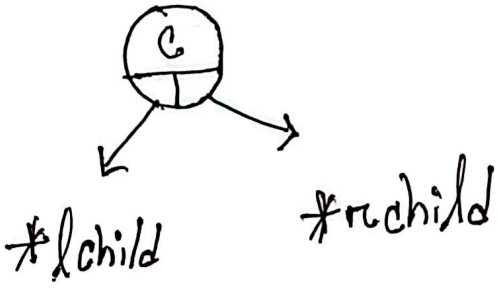
LL representation:



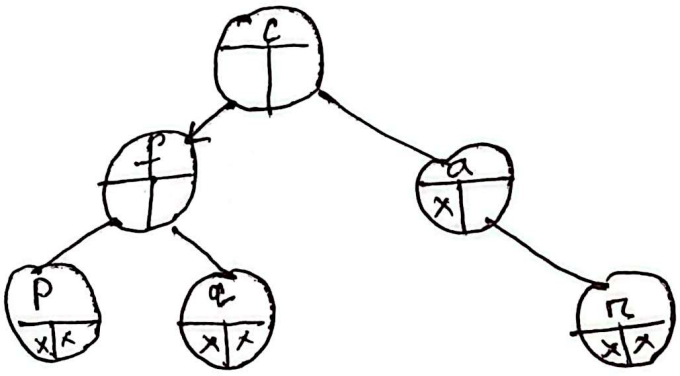
A BT

Any node have two pointers ~~on~~. One is to the left side child. Another is to the right child.

(1)



(2)



- A leaf node's left child and right child NULL.

Code:

```
struct node {
    char data;
    node *lchild;
    node *rchild;
};
```

DS (Theory):

Lec-07:

Parent-Child Relationship:

If Parent's index = i

left Child's index = $2i+1$

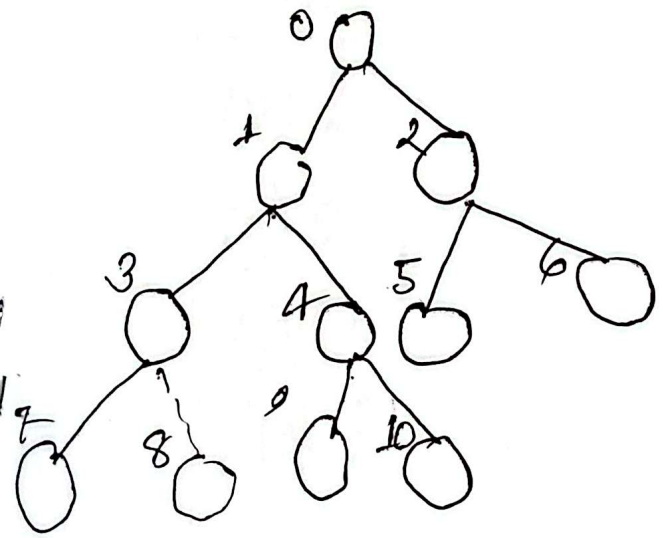
Right " " = $2i+2$

Example:

$i = 4$ (Parent's index)

$2i+1 = 2 \times 4 + 1 = 9$ (Left Chil's index)

$2i+2 = 2 \times 4 + 2 = 10$ (Right " ")



Child-Parent relationship:

If Child index = k

Parent's " = (k-1)/2

Examples:

Child's index = k = 5

Parent's index = (k-1)/2

= (5-1)/2

= 2

Tree-Traversal Method:

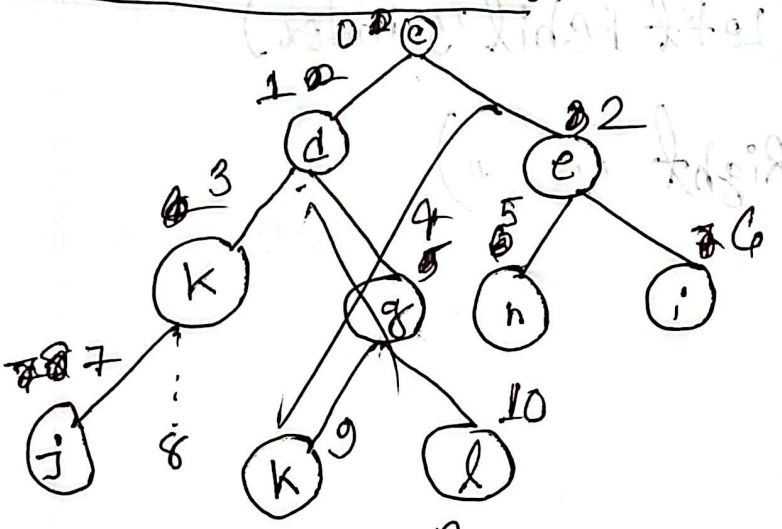


figure-1

Three-Methods:

(1) Pre order TTM

(2) In-order TTM

(3) Post " "

(1) Pre order TTM:

(1) Visit the root

(2) Traverse the left ~~side~~ sub-tree (In preorder)

(3) " " right " " (In ")

Printing (visiting) sequence: For fig-1

(1) c

(2) d

(3) f

(4) j

(5) g

(6) k

(7) l

(8) e

(9) h

(10) i

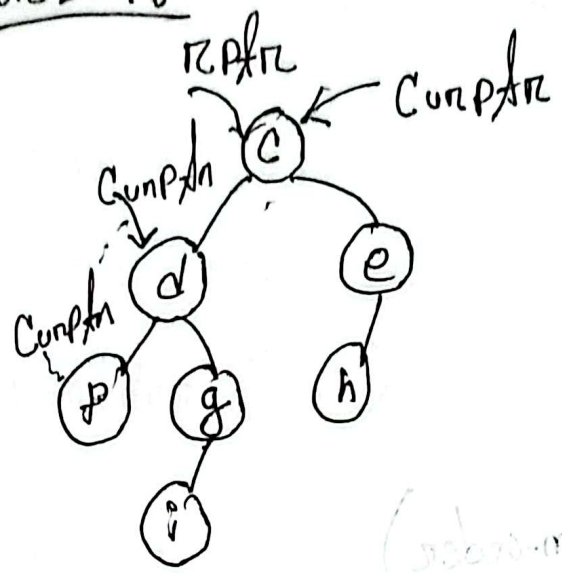
Pseudo-Code for Pre-order: Ref: fig-1

(If the tree is an array)

```
Char T[11];
int i, n=11;
Preorder (i, tree){
if (i<n) {
cout<int << Tree[i];
Preorder (2i+1, Tree);
Preorder (2i+2, Tree);
}
}
```

Note: If a function calls itself within it's body it is called a recursive ~~call~~ function

For LL :



Pseudo-Code of Preorder TTM in case of LL:

```
struct node{
    Char data;
    node *lChild;
    node* rChild;
};
node * rptr ,* curptr;
Curptr = rptr; // Starting
Preorder (Curptr)
{
 if (Curptr != NULL) {
        cout << Curptr->data
        Preorder (Curptr->lchild);
```

```
Preorder (Curptr->rchild);
}
}// end of Preorder ()
```

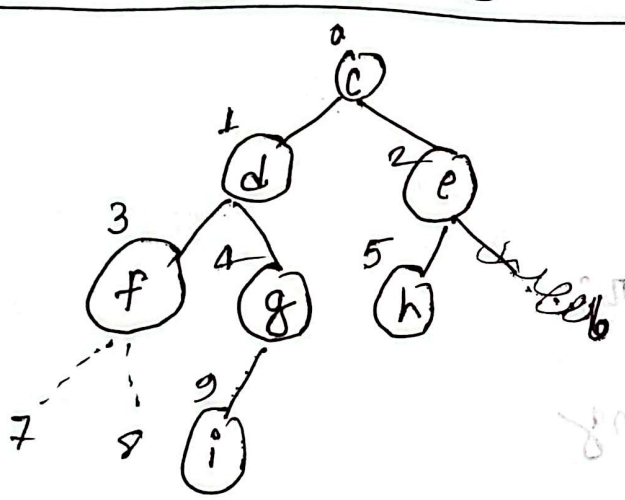
Inorder TTM:

(1) Traverse the left subtree (in-order)

(2) Visit the root

(3) Traverse the right subtree (Inorder)

Visiting (Printing) order (Sequence):



(1) f
(2) d
(3) i
(4) g
(5) c
(6) h
(7) e

Pseud-Code for Inorder TTM:

```
Char T[10];
int i , n=10;
Inorder (i,T)
{
 if (i<n){
    Inorder (2i+1 ,T)
     Cout << T[i];
    Inorder (2i+2 ,T)
 }
} //end of InOrder
```

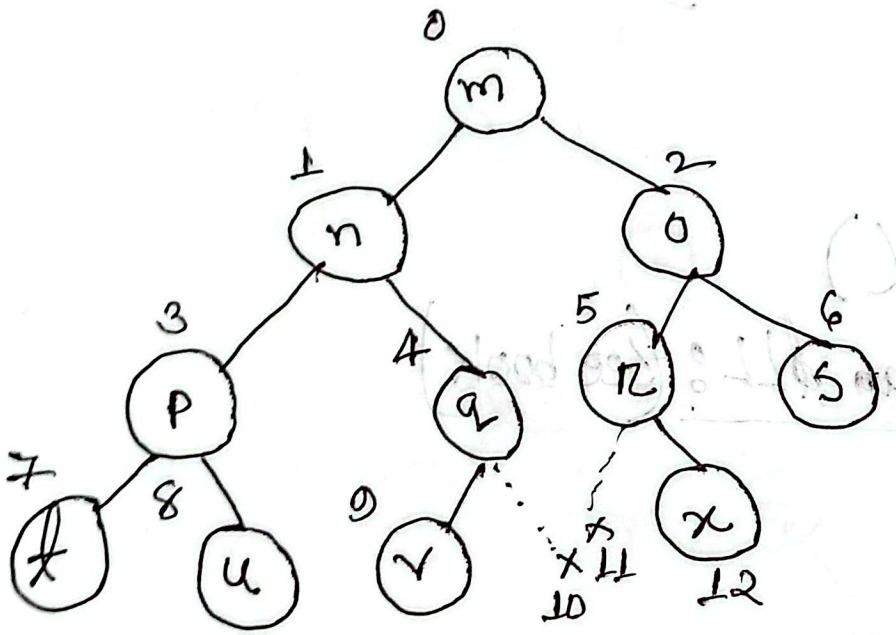
For LL based Tree see Pseudo-code from book

DS (Theory):

Lec-08:

Post order TTM:

(1) Traverse the left sub tree (In postorder)
(2) " " right " " (In postorder).
(3) Visit the root.



A BT

Visiting sequence (order):

(1) t
(2) u
(3) p
(4) u
(5) q
(6) n (root of the left subtree)
(7) x (8) r (9) s (10) o (11) m

Pseudo-Code:

When the tree is an array:

```
Char T[13];
int i;
Postorder (i, Tree){
if(i<n){
  Postorder (2i+1 , Tree);
  Post order (2i+2 ; Tree);
  Cout << T[i];
}
} //end of Postorder()
```

when the tree is an LL: (see book)

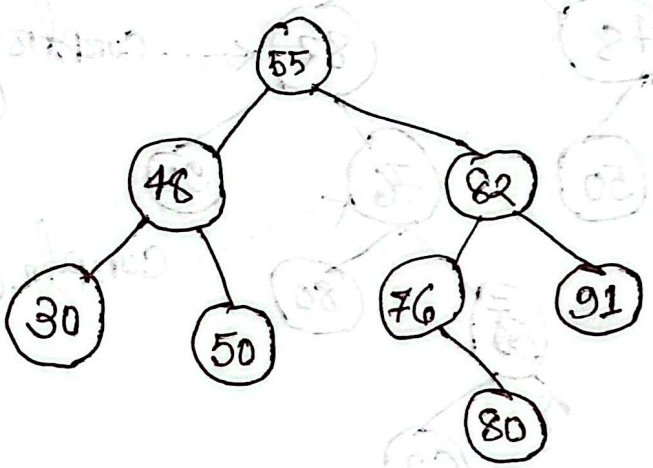
Binary Search Tree (BST):

Left Child's value is less than root's value.

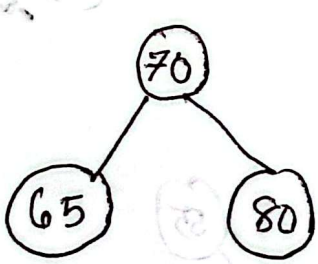
Right Child's " is greater than root's value.

Any value of the left sub-tree will be less than root's value.

Any value of the right sub-tree will be greater than the root's value.



A BST



A BST

Note: BST can be array or LL

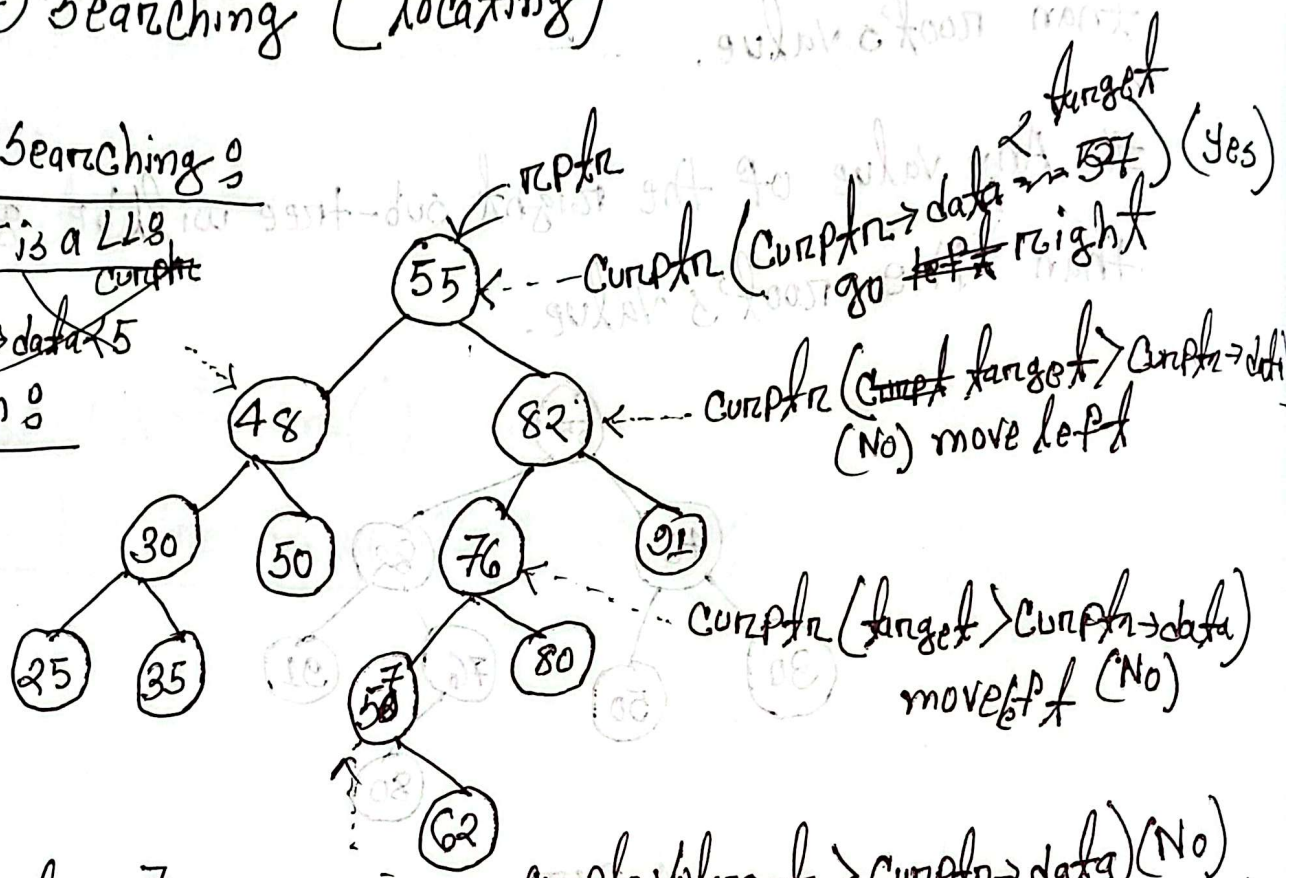
#Operations:

(1) Creation (Construction)
(2) Add (Insert)
(3) Delete (Removal)
(4) Searching (locating)

#Searching:

when BST is a LL:

Simulation:

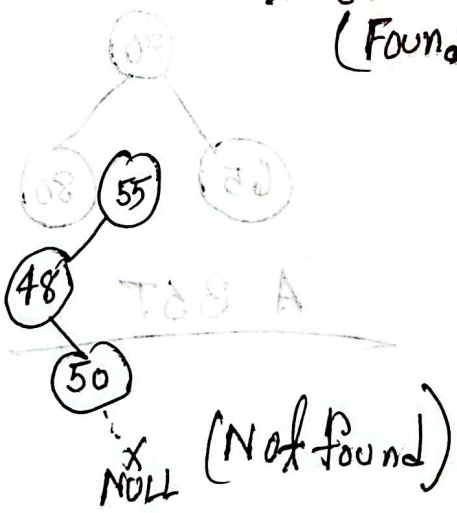


Target = 57;

~~when BST is a LL:~~

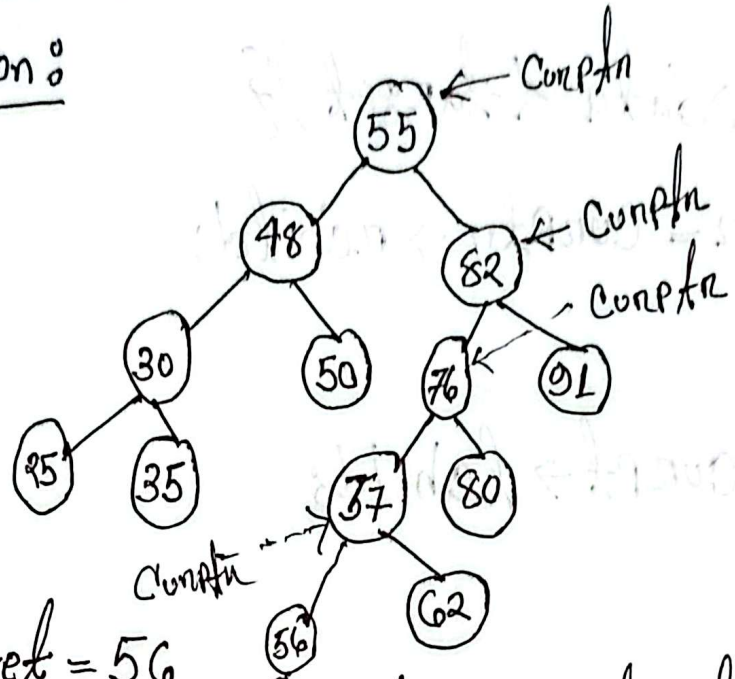
For Target = 52:

Track:



Add (Insert):

Simulation:



Target = 56

nptr link: curptr→lChild=nptr;

(1) locate the parent of the node to be inserted

(2) Establish necessary link

Pseudo-Code:

```
struct node {
    int data;
    node * lchild;
    node * rchild;
};
node *nptr, *curptr;
```

```
while(                    ){
    if (curptr->data < target){
        curptr = curptr -> rchild;
    else{
        curptr = curpt -> lchild;
    }
}
node *nptr;
nptr = new (node);
nptr->lchild = NULL;
nptr->rchild = NULL;
nptr->data = 56;
curptr->lchild = nptr;
```

Deletion:

Three Cases:

(1) If the node is leaf node

(2) If the node has only one child.

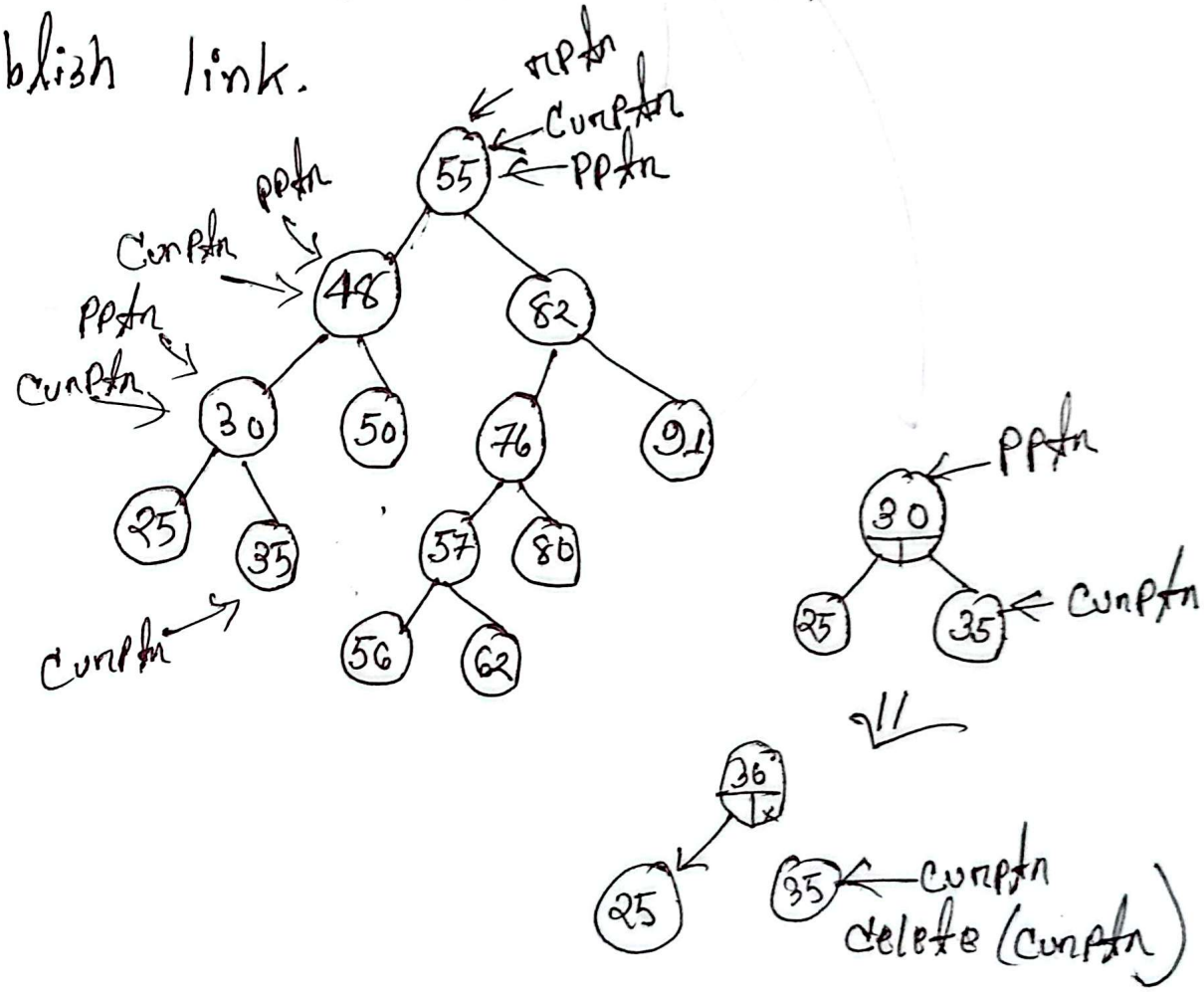
(3) If the node has ~~two~~ two children and grand child on children

Case-1: From Insert fig.

target = 35;

(1) locate the node and it's parent.

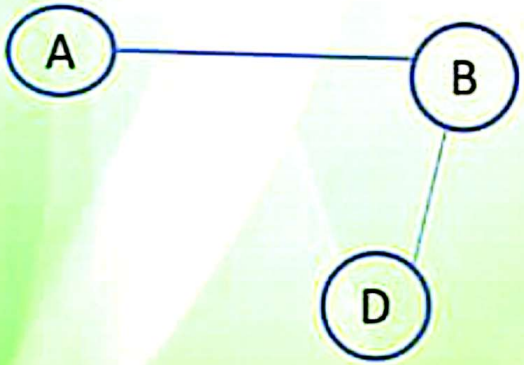
(2) Establish link.



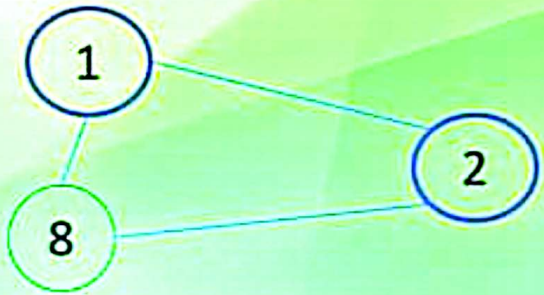
Case-2 → (~~Try~~ see book)

Basics of Graph [contd.]

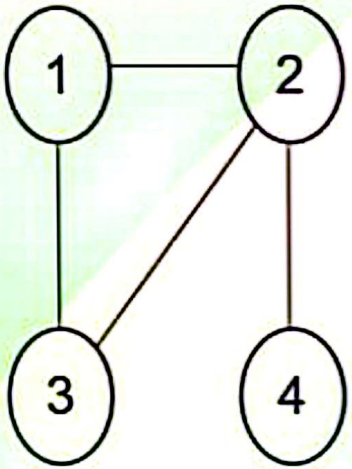
Path: A path is a sequence of vertices where each pair of successive vertices is connected by an edge.



Cycle: A cycle is a path where first and last vertices are the same.

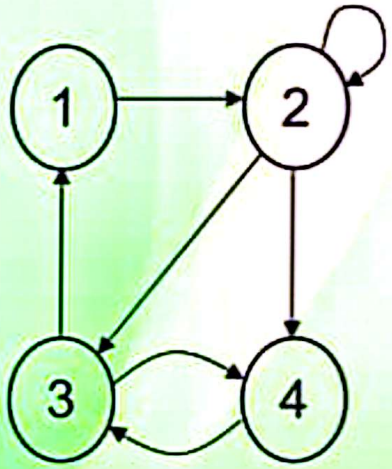


Graphs

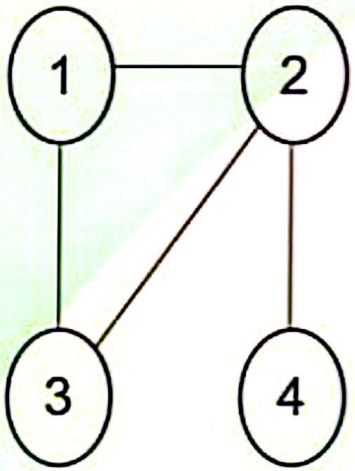


Undirected graph

Graphs



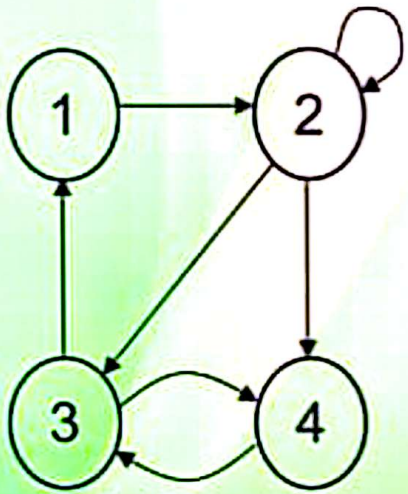
Directed graph



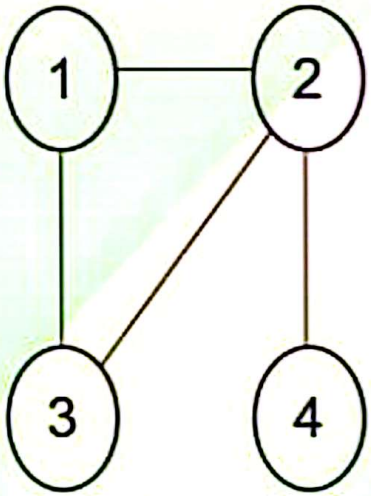
Undirected graph

Graphs

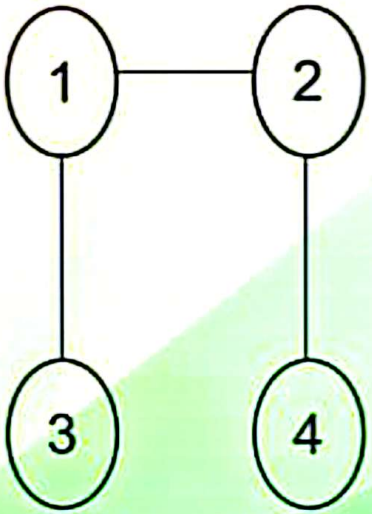
Acyclic Graph: A graph with no path that starts and ends at the same vertex.



Directed graph



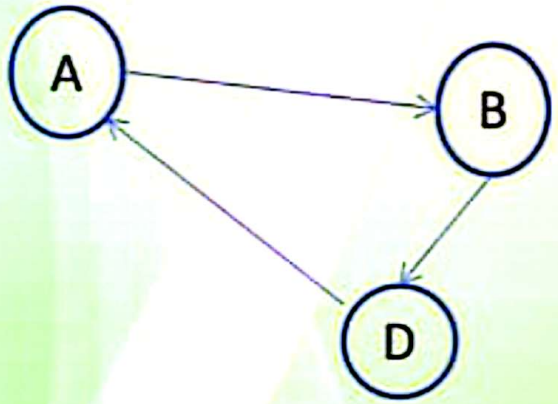
Undirected graph



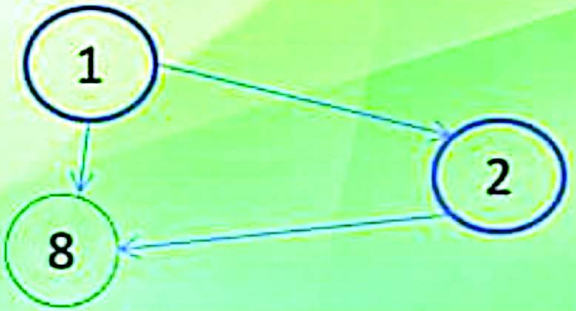
Acyclic graph

Basics of Graph [contd.]

The A, B, C is a cycle.



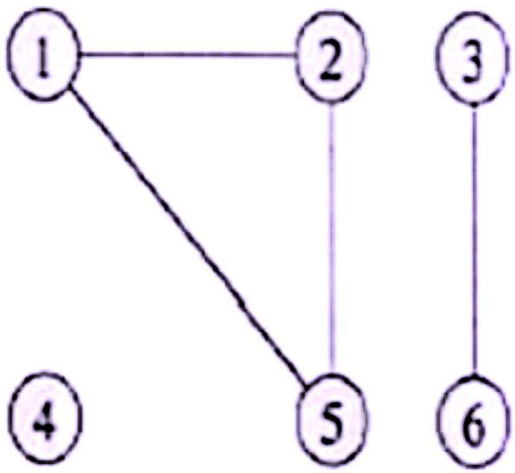
In the following graph there is no cycle.



Graph Example

An **undirected** graph G = (V, E), where
V = {1, 2, 3, 4, 5, 6} and
E = { {1,2}, {1,5}, {2,5}, {3,6}}.
The vertex 4 is **isolated**.
Vertex 3 is **adjacent** to vertex 6 and vice versa; {1, 5} is **adjacent** to 2;
4 is not **adjacent** to any other vertex.

Vertex 1, 2, 5 has **degree** 2;
vertex 3, 6 has **degree** 1;
vertex 4 has **degree** 0.

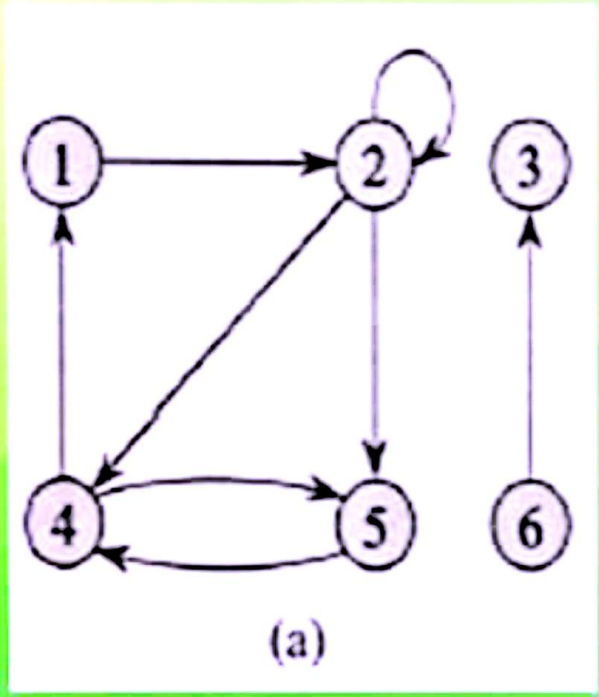


(b)

Graph Example

A **directed** graph G= (V, E), where
V = {1, 2, 3, 4, 5, 6} and
E={ (1,2), (2,2), (2,4), (2,5), (4,1), (4,5), (5,4), (6,3) }.
The edge (2,2) is **self loop**.
Vertex 4 is **adjacent** to vertex 5; {1, 5} is **adjacent** to 4;
3 is not **adjacent** to any other vertex except 6.

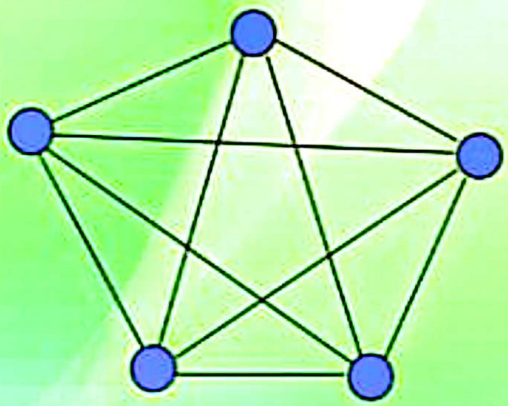
Vertex 5 has **in-degree** 2 and **out-degree 1**.



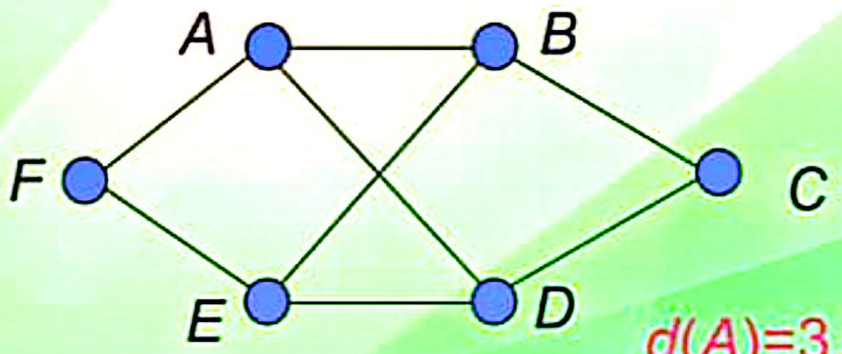
Graph Introduction

Complete graph: When every vertex is strictly connected to each other. (The number of edges in the graph is maximum).

Degree of a vertex v: The degree of vertex v in a graph G, written d (v), is the number of edges incident to v, except that each loop at v counts twice (*in-degree* and *out-degree* for directed graphs)



Complete Graph



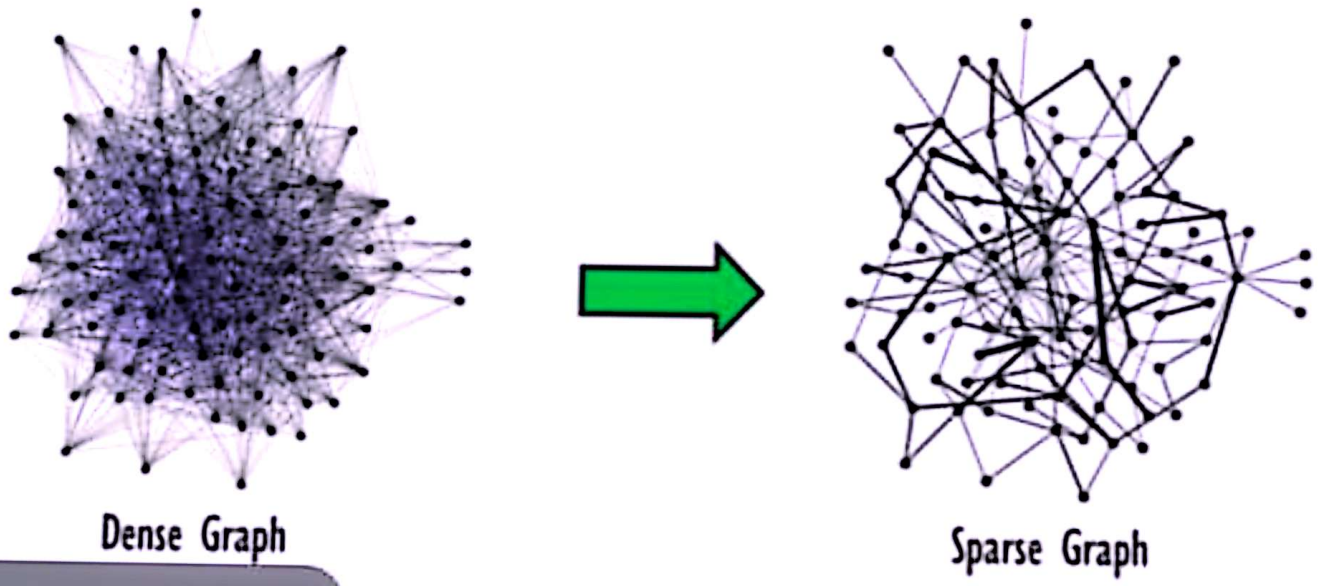
$d(A)=3$, $d(B)=3$, $d(C)=2$

$d(D)=3$, $d(E)=3$, $d(F)=2$

Graph Introduction

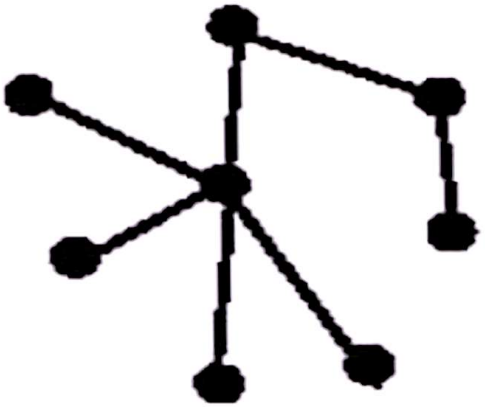
Dense graph: When the number of edges in the graph is close to maximum. (*adjacency matrix* is used to store info for this)

Sparse graph: When number of edges in the graph is very few. (*adjacency list* is used to store info for this)

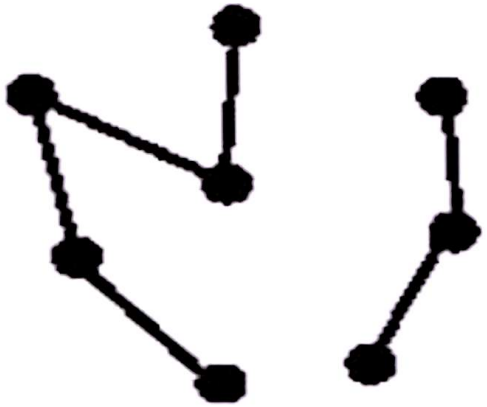


Md. Rafiqul Islam

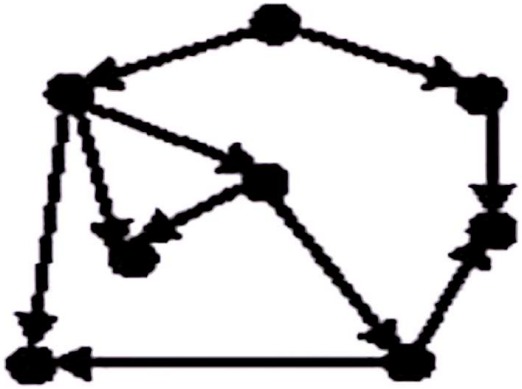
Forests, DAG, Components



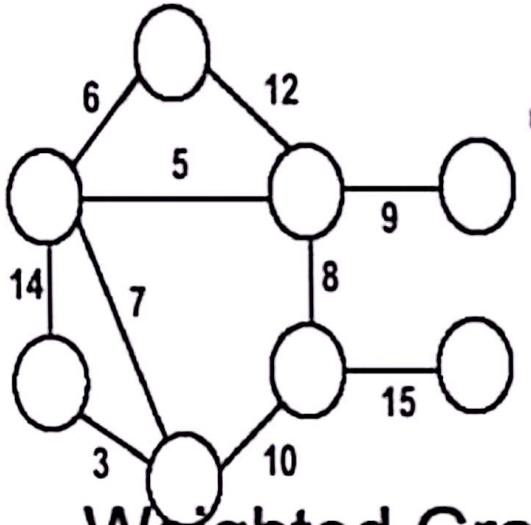
Tree



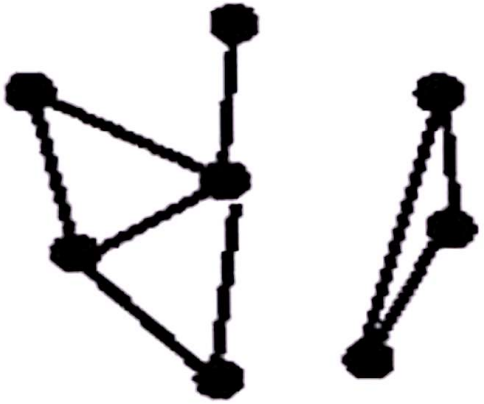
Forest



DAG



Weighted Graph

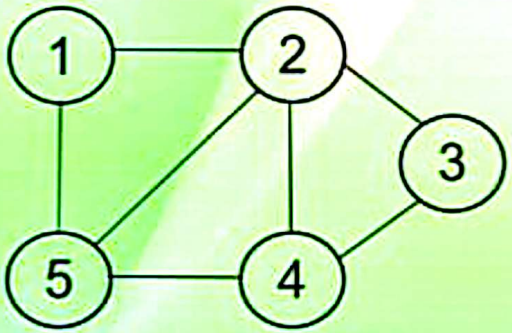


2 Components

Strongly-connected Component

Graph Representation

- Adjacency matrix representation of G = (V, E)
  - Assume vertices are numbered 1, 2, … |V|
  - The representation consists of a matrix $A_{|V| \times |V|}$:
  - $a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$



Undirected graph

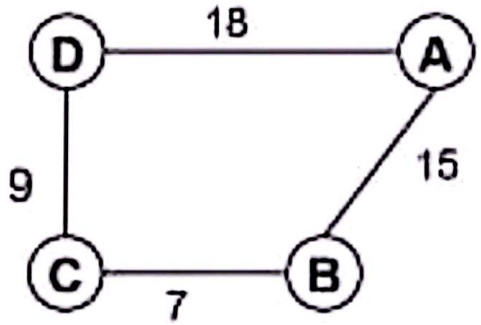
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

For undirected graphs matrix A is symmetric:

$a_{ij} = a_{ji}$

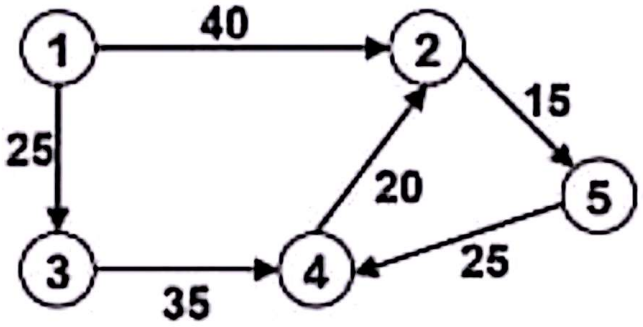
$A = A^T$

Graph Representation [contd.]



	A	B	C	D
A	0	15	0	18
B	15	0	7	0
C	0	7	0	9
D	18	0	9	0

(a): Adjacency matrix of an undirected graph



	1	2	3	4	5
1	∞	40	25	∞	∞
2	∞	∞	∞	∞	15
3	∞	∞	∞	35	∞
4	∞	20	∞	∞	∞
5	∞	∞	∞	25	∞

(b): Adjacency matrix of a directed and weighted graph

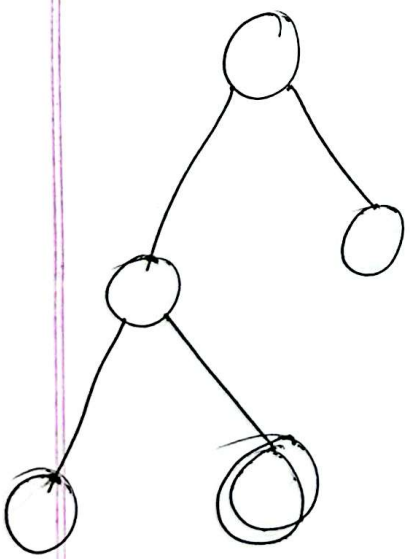
Heap (Binary Tree)

Shape will be complete or full BT

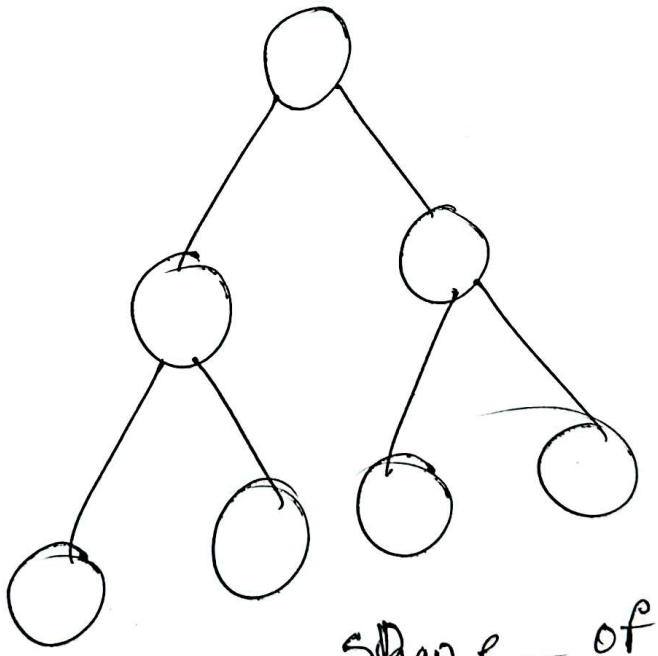
Parent's value greater than any child's value (Max-heap)

Parent's value smaller than any child's value (Min-heap)

Shape

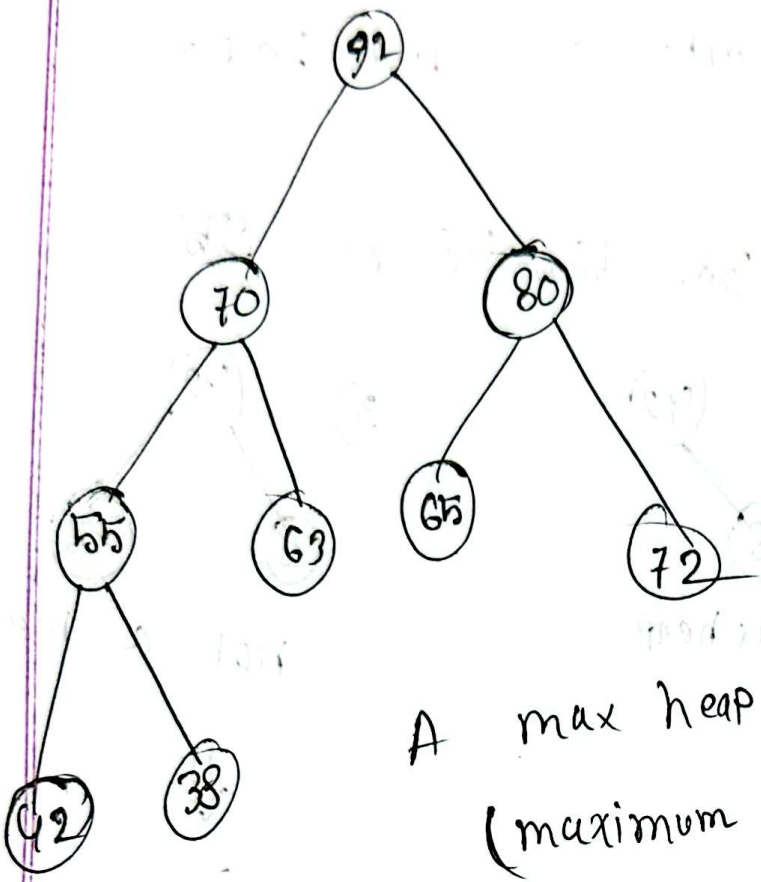


Shape of a complete BT



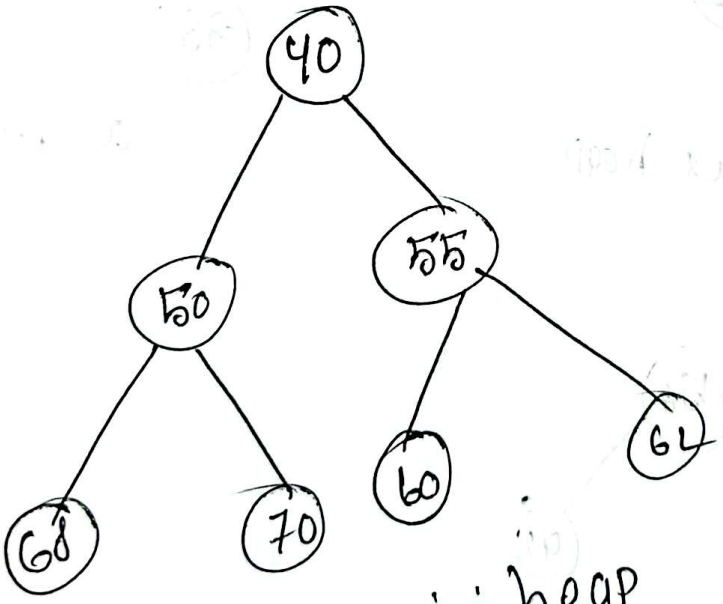
Shape of a full BT

example of max heap:



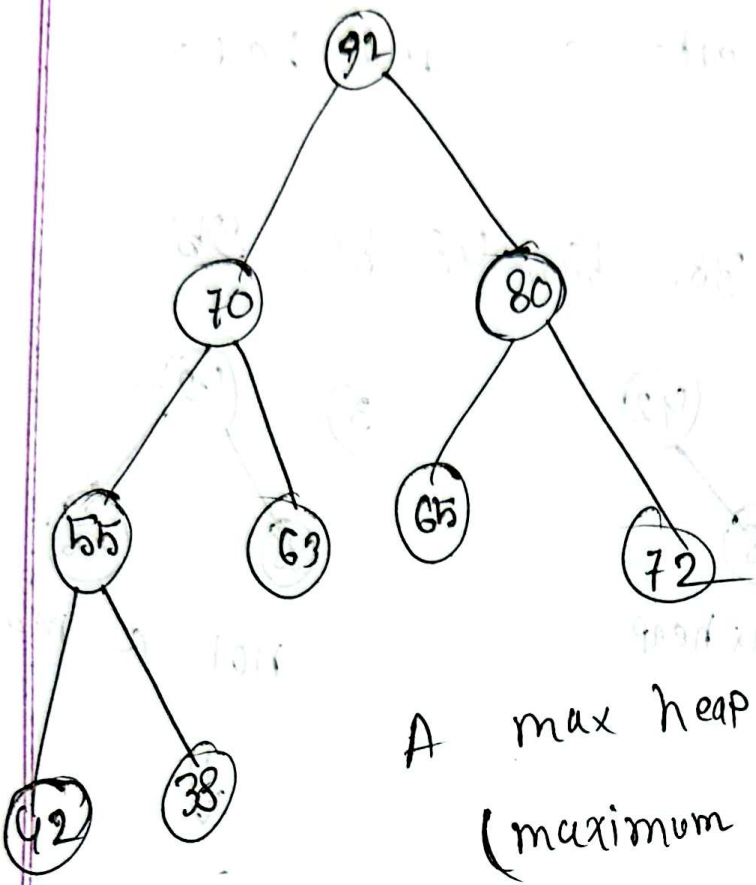
A max heap
(maximum value at the root)

Example of a min heap:



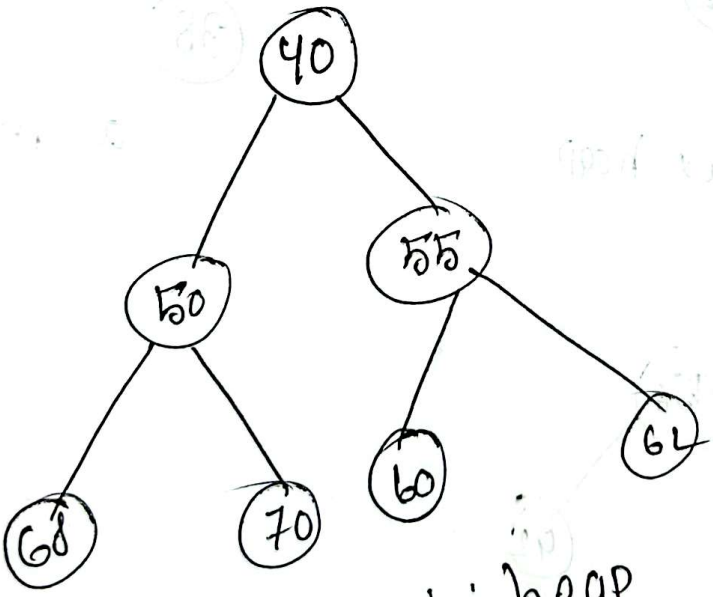
A mini heap
(Minimum value at the root)

example of max heap:



A max heap
(maximum value at the root)

Example of a min heap:



A mini heap

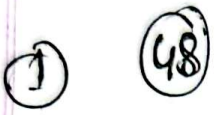
(Minimum value at the root)

~~left → big value~~
~~Right → Small~~

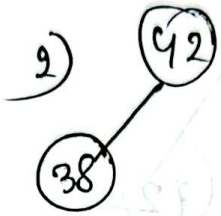
1) Creation or Construction (Heapify)

Target is to create a max heap:

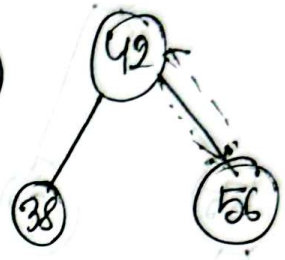
List of Data: 42, 38, 56, 48, 80, 95

1) 

A max heap

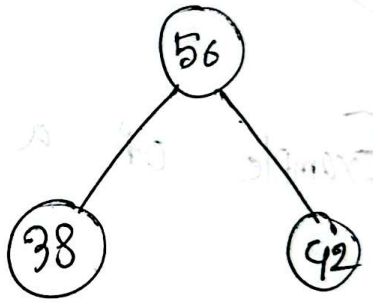
2) 

a max heap

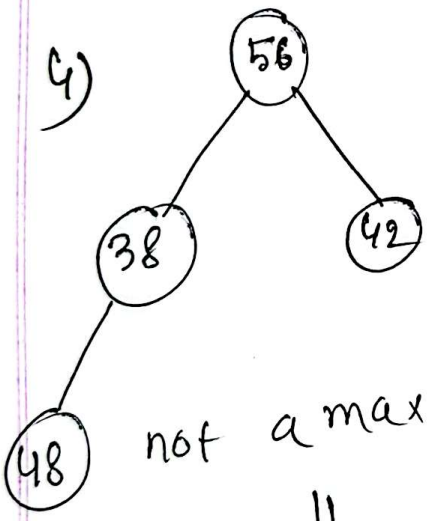
3) 

not a max heap

⇓

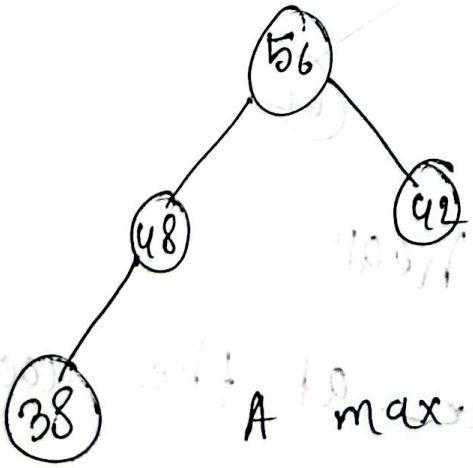


a max heap

4) 

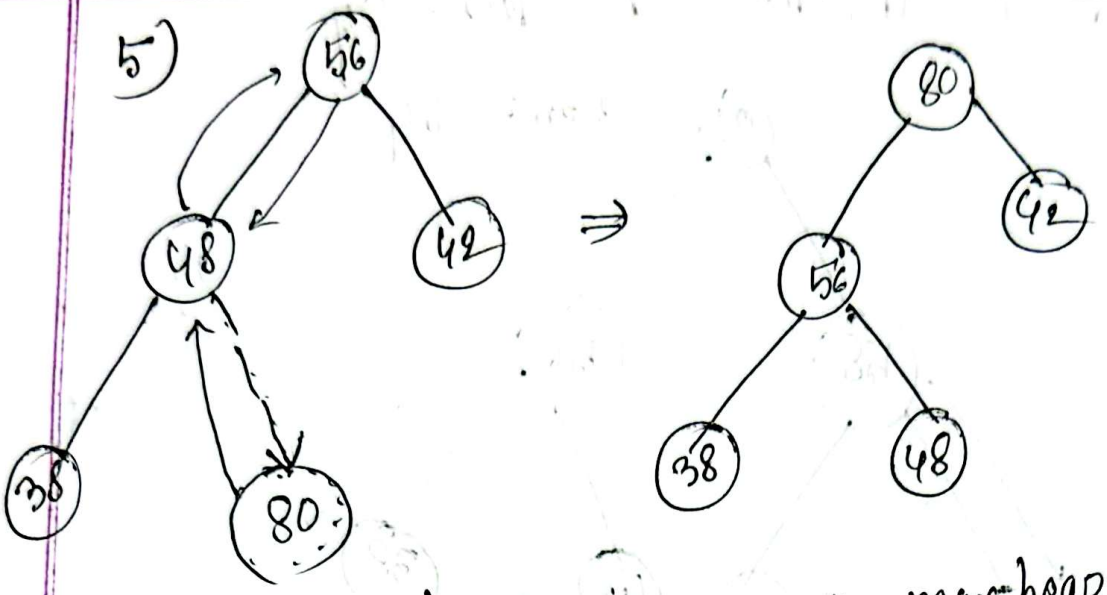
not a max heap

⇓



A max heap

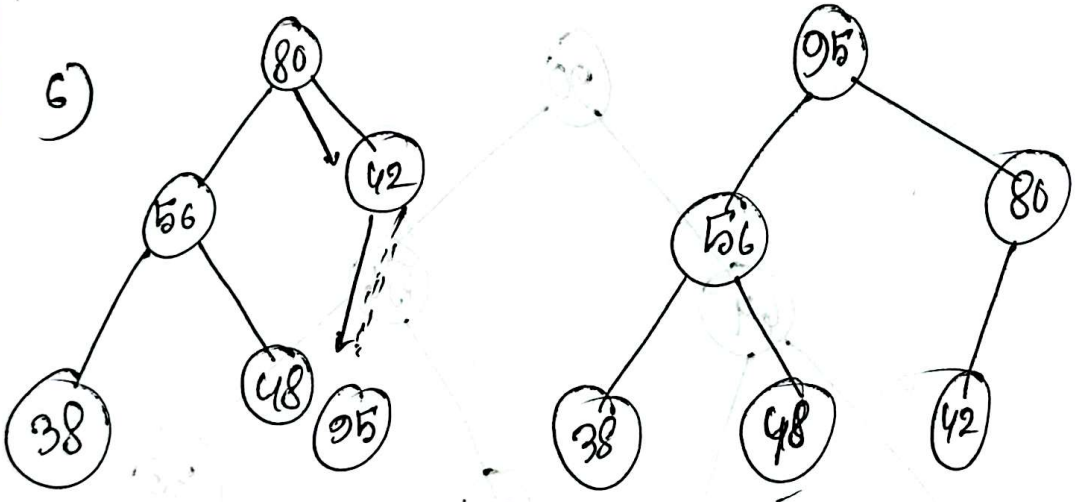
5)



Not a max heap a max heap

root এর
পরে right-
always left
থেকে বড় হবে।

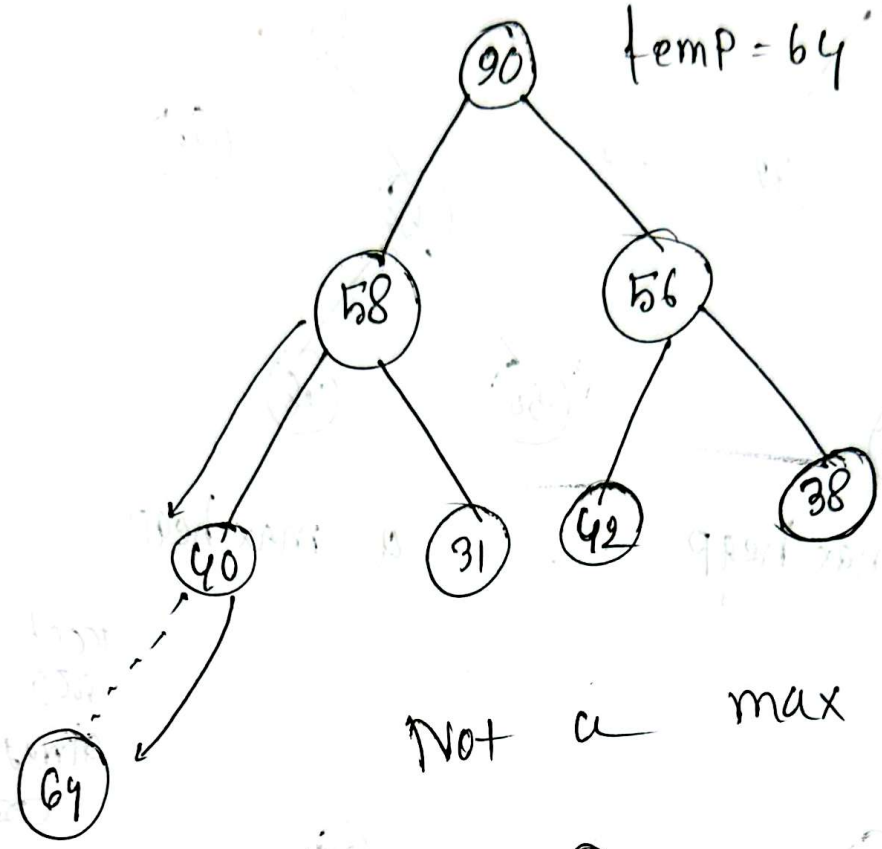
6)



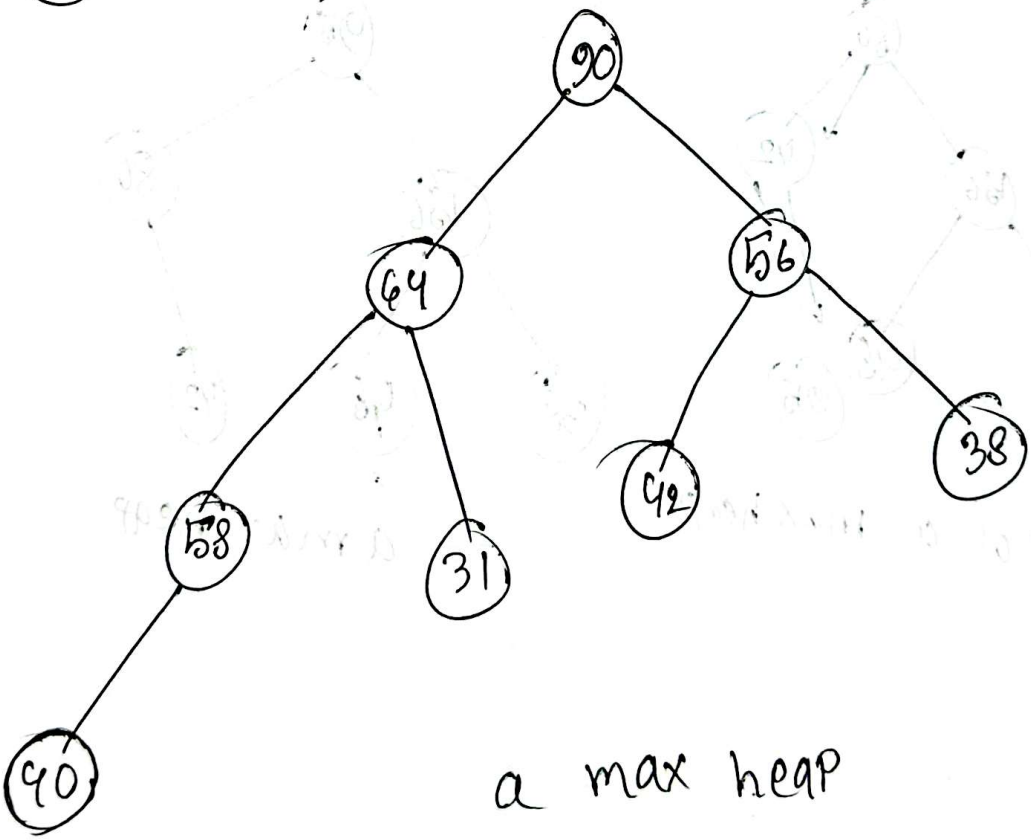
not a max heap a max heap

2) Add or insert a data = 64

temp = 64



Not a max heap



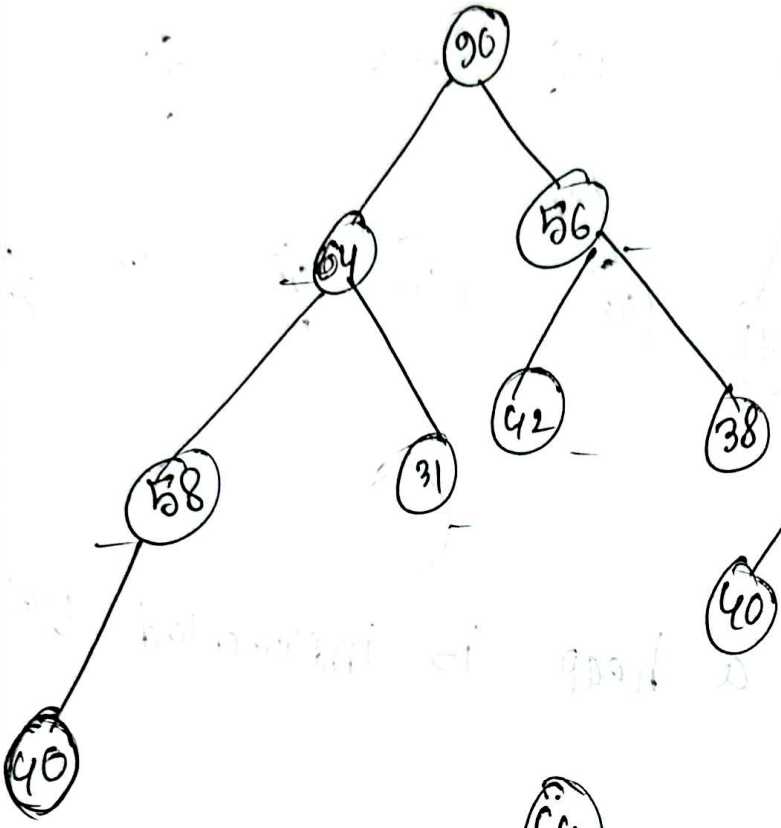
a max heap

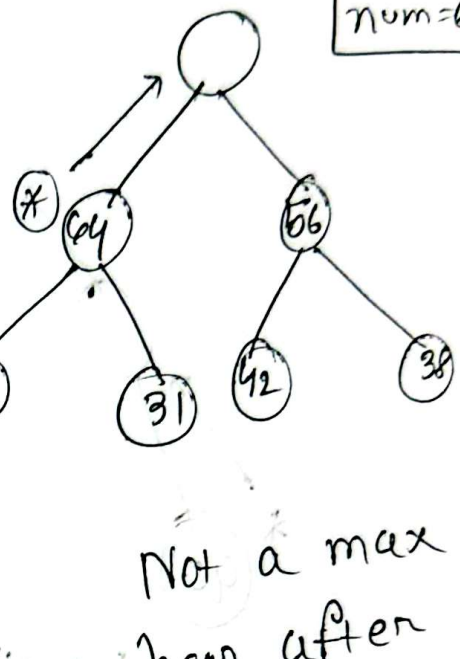
(root always ৫ থাকবে)

Deletion (Removal)

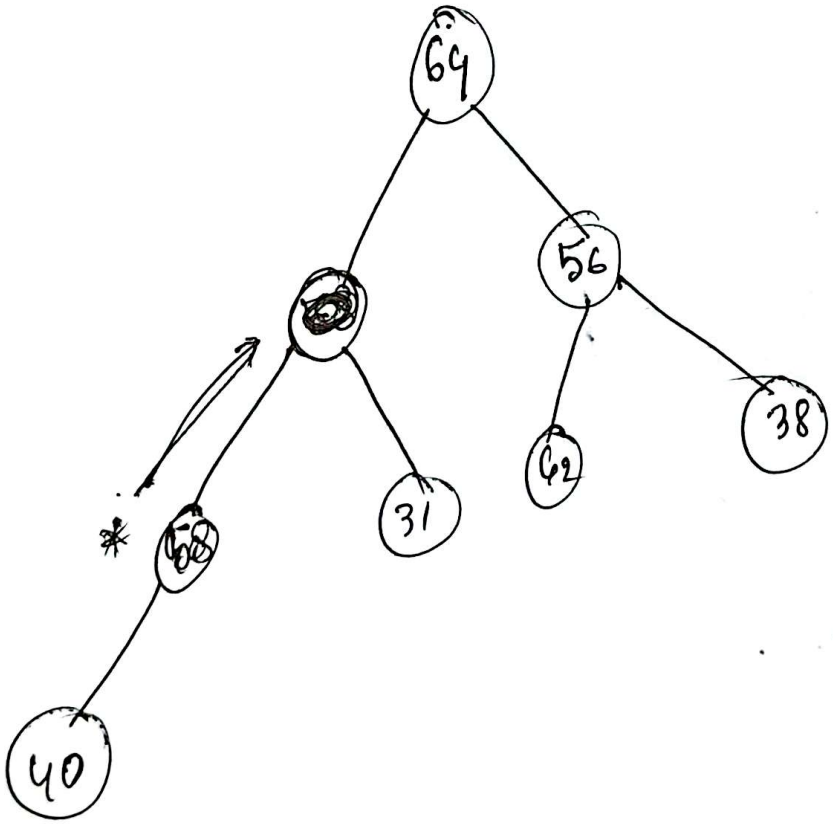
target is to delete 90

max num = 64





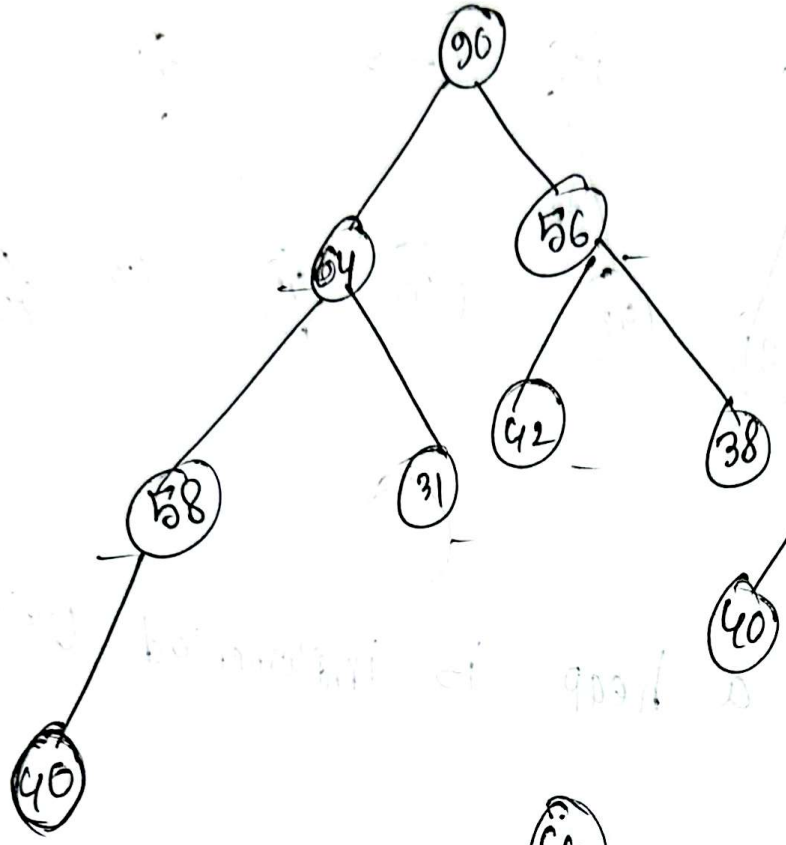
Not a max heap after deletion

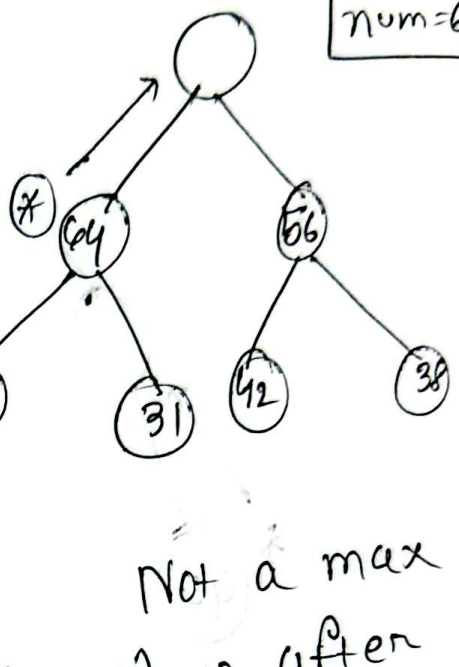


Deletion (Removal)

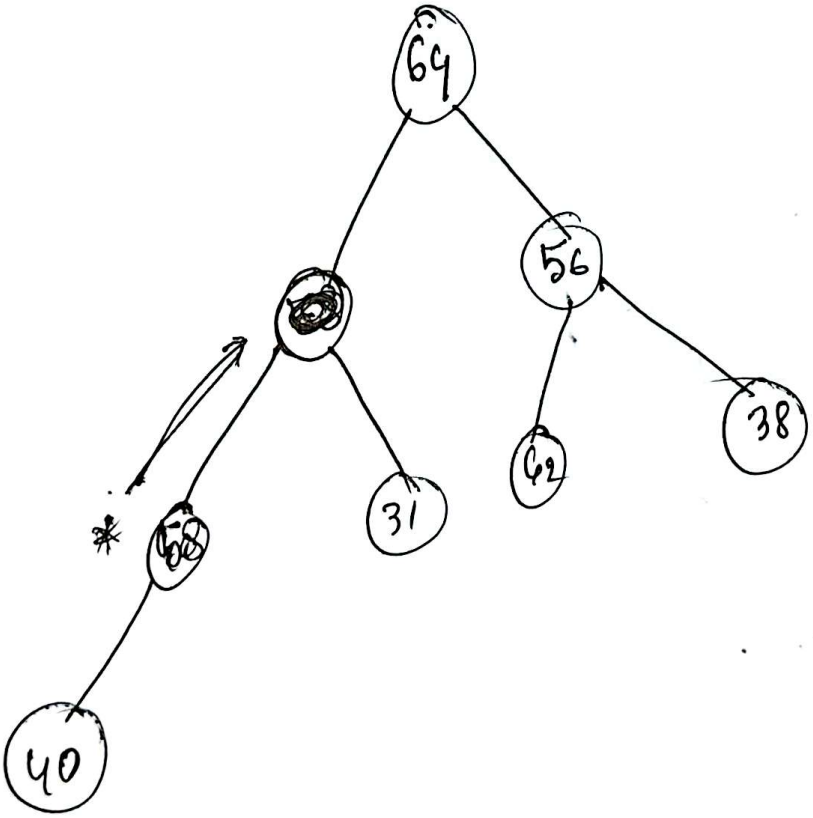
target is to delete 90

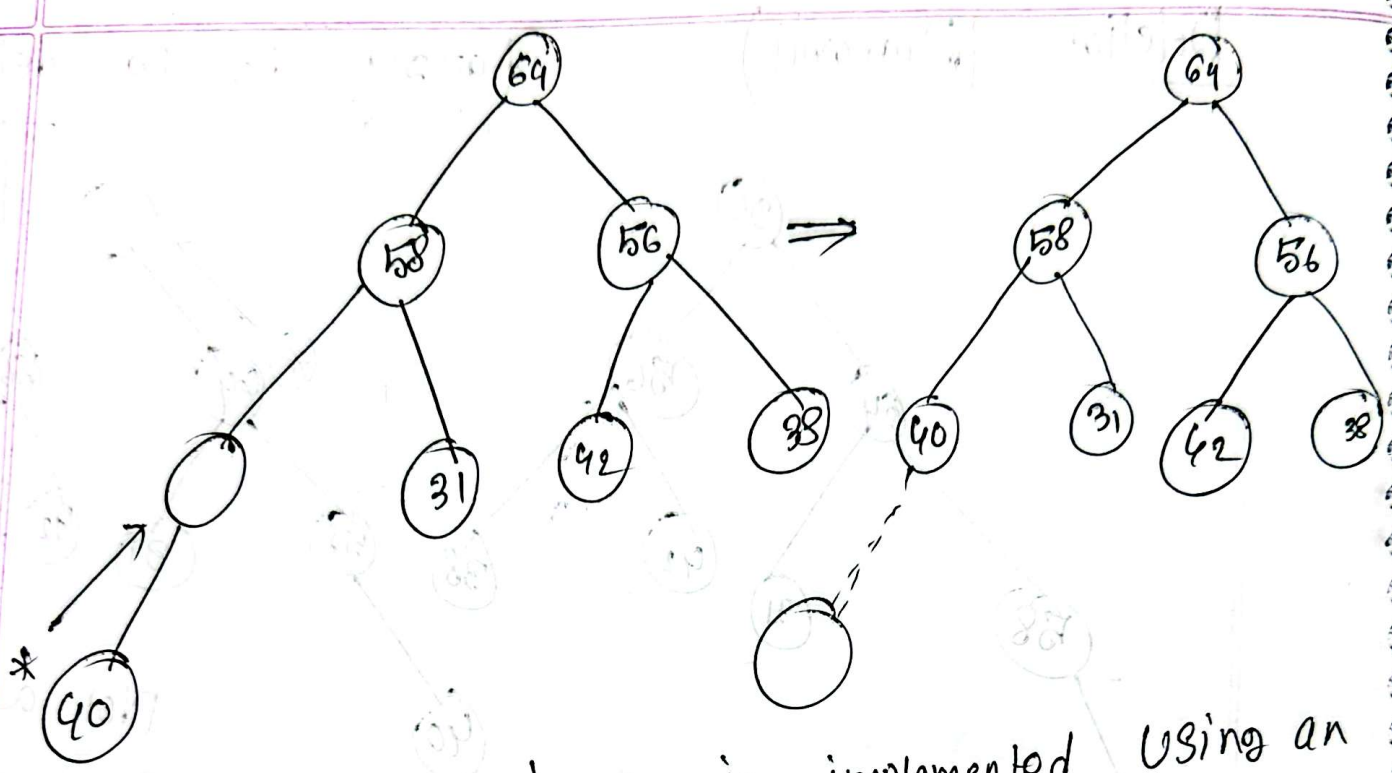
max num = 64





Not a max heap after deletion





Usually a heap is implemented using an array.

American International University-Bangladesh (AIUB)

Data Structure (FT_Lab Performance 2) **[Answer Any 1]** **Marks: 20**

********** Task 1 **********

In a hospital registration system, patients arrive one by one and are added to a waiting list based on the order of arrival. Each patient has a name, a unique ID, and a disease name. The system must allow adding new patients to the queue, but if a patient arrives who has a **Name like yours**, the system must **Reject** him/her and not add them to the list. After adding patients, the system should allow searching for patients based on a given disease name and display all patients who have that disease. This will help the medical staff quickly identify and group patients with similar illnesses. The system should also allow viewing all patients currently waiting in the queue with their full details. Implement this logic using a queue data structure and a class in C++ to manage the patient registration and search functionality.

********** Task 2 **********

In a small clinic's medicine storage system, new medicine boxes are stacked one over another as they arrive, with the most recently placed box at the top. Each medicine box has a medicine name, a batch number, and an expiry date. User should input all the details. The system allows staff to add new boxes, but if the medicine's **Expiry Date is today's date, it must not be added**. The staff can also search for medicine boxes based on a specific medicine name and view all boxes currently in the storage stack. Use proper data structure concept in C++ to implement this system for managing the medicine storage.

******* If the code is found to be copied from ChatGPT or any other sources, no marks will be given *******

Write a C++ program to implement a stack using an array without using classes. The program should include basic stack operations such as **push()** to insert an element, **pop()** to remove the top element, **peek()** to view the top element without removing it, and **display()** to show all elements currently in the stack. Use a fixed-size array (size 5) and ensure that the program handles stack overflow and underflow conditions appropriately. Demonstrate each of these operations clearly in the **main()** function. Do not use object-oriented concepts like classes or structures.

Quiz-2, Section: C

Set: C

Marks: 15x2, Time: 20 minutes

Sl#: ID: Name:

Answer two questions

1. The figure shown below is a **circular queue**. Write **pseudo-code** to print the data from the **front to the rear positions.** Show the figure after enqueue h to the queue.

b	d			c	f	g
	rear = 1			front = 4		

2. **Draw a BST** with the data: 69, 80, 76, 60, 90, 33, 45, 55, 74, 98. **Delete a node with only one child** and show the updated BST.

Write answers below

Quiz-2, Section: C

Set A

Marks: 15x2, Time: 20 minutes

Sit: ID: Name:

Answer two questions

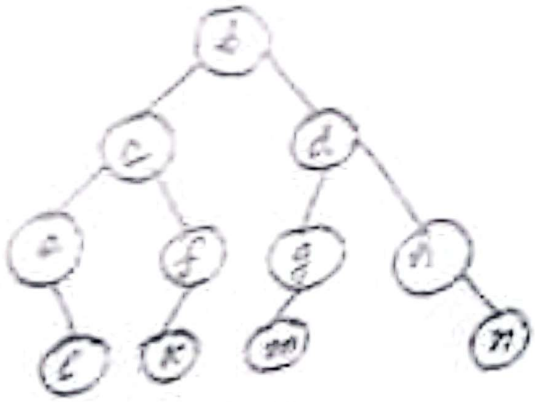
1. The figure shown below is a circular queue. Now enqueue t and b. Next deque p. Write **pseudo-code** to perform operations and show figures after enqueue and dequeue separately.

	p	q	r	s	

Front = 1 rear = 4

2. See the figure below and **print the visiting sequence** for the tree using the **post-order method**. Consider the tree as an array and write **pseudo-code** for the **post-order method**.

Write answers below



A BT

Set: D

Quiz-2, Section: C

Marks: 15x2, Time: 20 minutes

Sl#: ID: Name:

Answer two questions

1. The figure shown below is a circular queue. Now enqueue 92 and 51. Show the updated queue. Write pseudo-code to print all the data from the front to the rear positions of the updated queue.

			71	62	83	44

front = 3 rear = 6

2. Draw a BST with the data: 29, 80, 76, 10, 90, 23, 45, 55, 14, 98. Delete a node with two children and show the updated BST.

Write answers below

ID:	Marks:	Sec:

Part: A	OBE Evaluation	[1x15 = 15 Marks]

The answer of this part will be evaluated for the following OBE Course Outcome:

CO4: Use determined approaches in solving complex real-life problems with specific requirements.

Knowledge and proper use of Data Structure (5)	Problem Analysis (5)	Solution (5)	Total (15)

You are simulating a delivery route for a courier service where each delivery location is represented as a node. The initial route includes delivery points at house numbers **10, 20, 35, 50, 65, and 80, connected such that each point links to both the next and previous locations**. A new delivery stops at 90 needs to be added at the first of the route, and another stop at 30 should be inserted at the 2nd last end position. After the updates, the last stop should be removed. After each operation, the updated route should be displayed. Finally find out the length of the routes and find out the mid-point **(if possible)**.

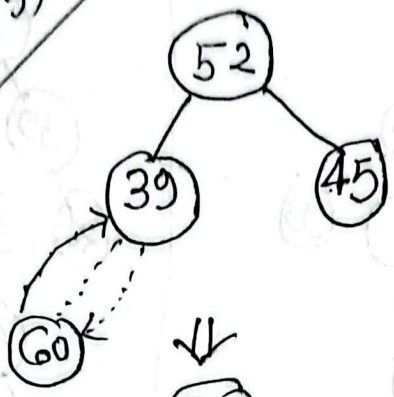
Once all list operations are completed, use the remaining delivery points to construct a Binary Search Tree (BST). Finally, display the delivery points using post-order traversal of the BST to show the locations.

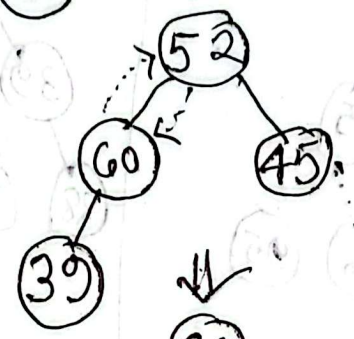
CQ:

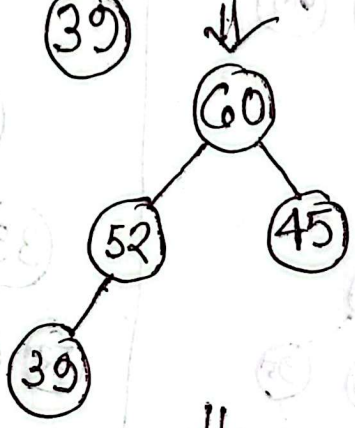
```
showQ( ) {
    if(rear == -1 & front == -1) {
            cout << "Q is empty";
    }
else if( front <= rear){
    for (int i= front ; i<= rear ; i++){
            cout << Q[i];
    }
}
else{
    for (int i= front ; i<=(maxSize-1) ; i++){
            cout << Q[i];
    }
    for(int i=0 ; i<=rear ; i++){
            cout << Q[i];
    }
}
}
```

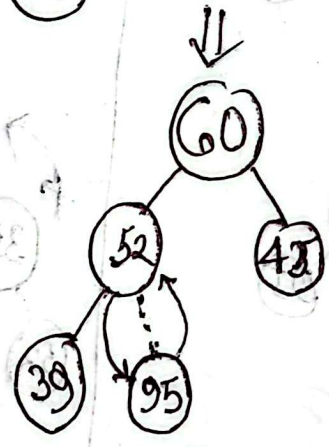
Heap

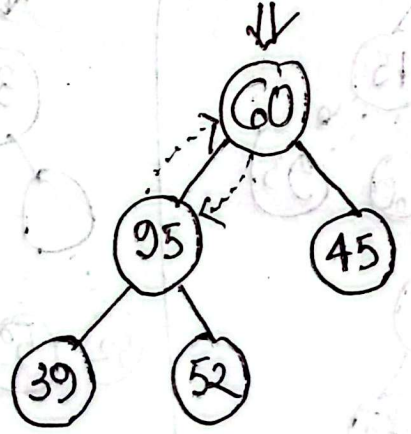
52, 39, 45, 60, 95, 28, 59

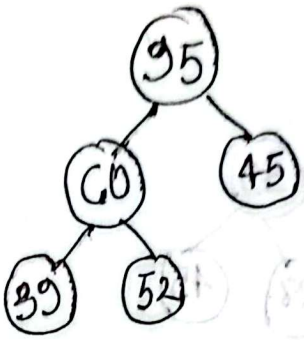


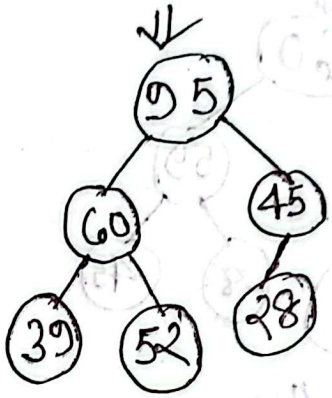


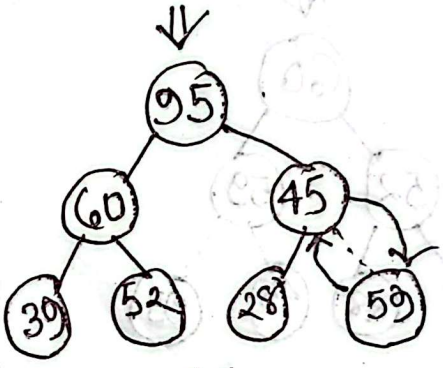


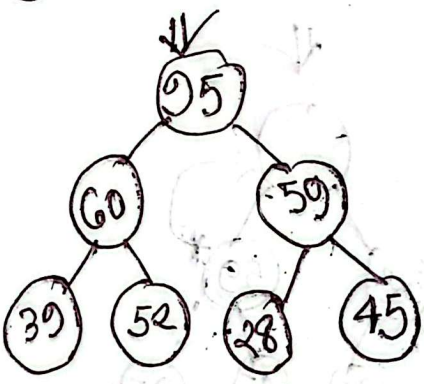




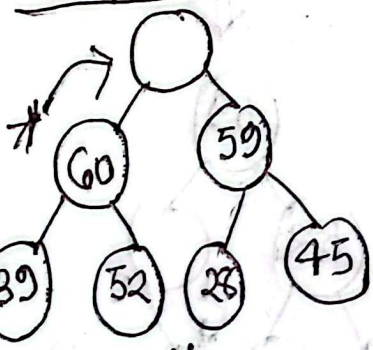


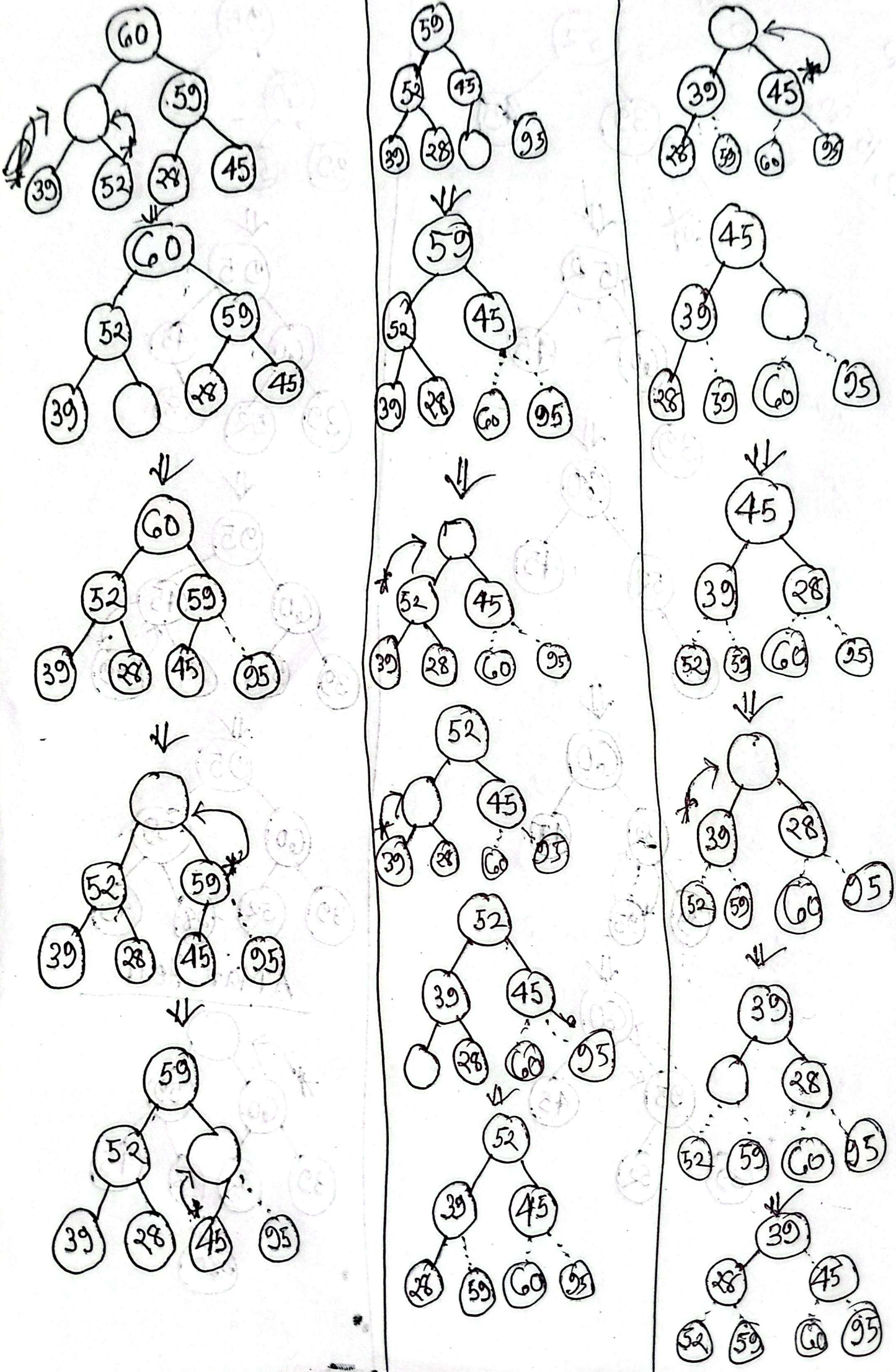


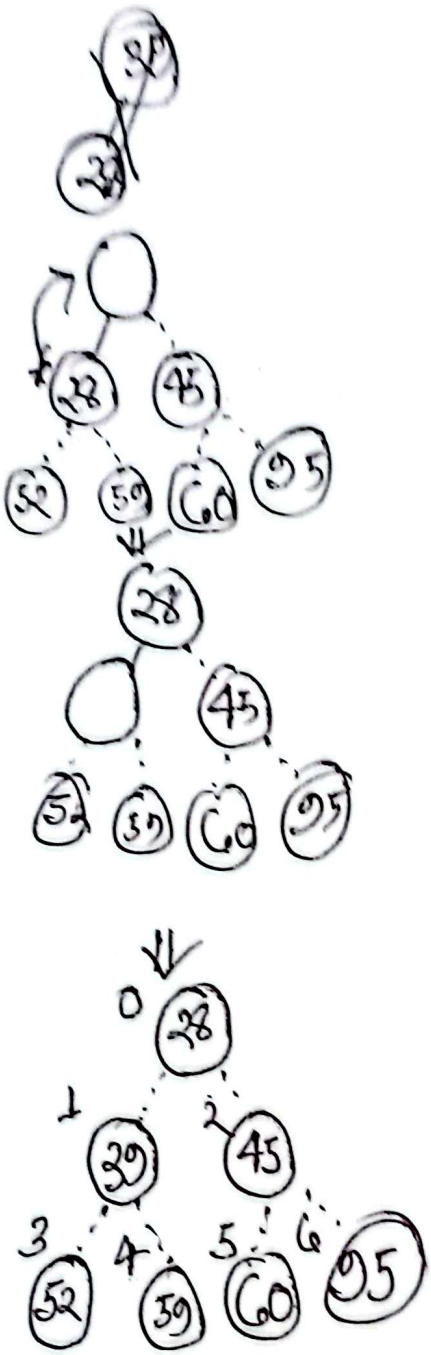




A Max Heap




60
59
39
52
28
45
60
52
59
39
28
45
60
52
59
39
28
45
95
52
59
39
28
45
95
59
52
39
28
45
95
59
52
45
39
28
95
59
52
45
39
28
60
95
52
45
39
28
60
95
52
39
45
28
60
95
52
39
45
28
59
60
95
39
45
28
52
59
60
95
45
39
28
52
59
60
95
45
39
28
52
59
60
95
39
28
52
59
60
95
39
28
52
59
60
95
39
28
52
59
60
95
39
28
52
59
60
95



28	39	45	52	59	60	95

(Ans)