22/07

Queue

PAGE NO. DATE 22/07

1. Queue is a collection of objects that are inserted and used according to the First In First out (FIFO) principle.
2. The elements enter a queue at the back and removed from the front.
3. Insertion point is also known as 'rear'.
4. It is a linear data structure.
5. Example: queue of consumers for a resource where the consumer that came first is served first.
6. Applications of queue →

a] Simulations and Operating system:
Os often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occurs.

b] Buffering →
Computer system must often provide a holding area for messages between two process, two program or even two systems.

Another application of queue is to help us to simulate and analyze real world.

7. In queue ~~dast~~ data structure, items are maintained in the order in which they are added to the structure.
8. Queue operations:

a] Queue() → It creates a new empty queue with containing no items.

b] ~~isEmpty~~ length() → It returns the length of the queue (total no. of items in queue).

Teacher's Signature:.....

c] IsEmpty() → It returns a boolean value whether the queue is empty.

d] enqueue() → It insert the given 'item' at the rear point of the queue.

e] dequeue() → It removes and return the front item of the queue.
An item can't be dequeue when the queue is a empty queue.

9. Queue implementation using list.

```
class Queue:
  def __init__(self):
    self._element = list()
  def isEmpty(self):
    return len(self) == 0
  def __len__(self):
    return len(self._element)
  def enqueue(self, item):
    self._element.append(item)
  def dequeue(self):
    assert not self.isEmpty(), "Empty Queue"
    return self._element.pop(0)
```

Description:

1. `def __init__(self):`
   `self.element = list()`

→ It creates an empty queue.

Teacher's Signature:

2. def IsEmpt(self):
    return len(self)==0

→ It returns true if the queue is empty.

3. def __len__(self):
    return len(self._element)

→ Returns the number items in queue.

4. def enqueue(self, item):
    self._element.append(item)

→ Adds the item at the end (rear) of the queue.

5. def dequeue(self):
    assert not self.IsEmpty(), "Empty Queue"
    return self._element.pop(0)

→ Remove and return the first element.

Data structure

Unit I →

Queue:

* Queue ADT →

1. Operation in Queue ADT →

a. enqueue(): Insert an element at the end of the queue.

b. dequeue(): Remove and return the first element of the queue, if the queue is not empty.

c. peek(): Return the element of the queue without ~~rem~~ removing it, if the queue is not empty.

d. size(): Returns the number of element in the queue.

e. isEmpty(): Return true if the queue is empty, otherwise return false.

f. isFull(): Return true if the queue is full, otherwise return false.

* Advantages of queue:

1. Large amount of data can be managed efficiently.
2. Insertion and deletion operations can be performed with ease as it follows FIFO.
3. Queue are useful when a particular service is used by multiple consumer.
4. Queue are fast in speed for inter-process communication.
5. Queue can be used in implementation of other data structure.

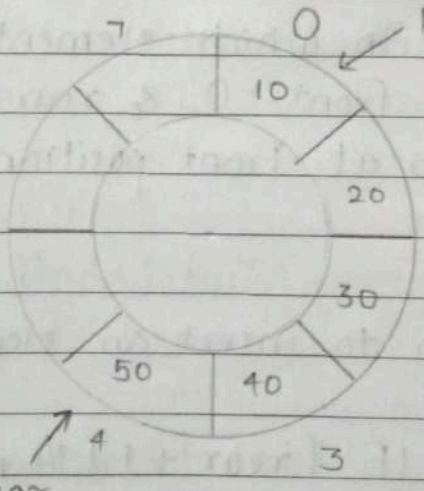
Teacher's Signature:.....

* Disadvantage of queue:

1. The Operation such as insertion and deletions of elements from the middle are time consuming.
2. Limited space.
3. A new element can only be inserted when the existing elements are deleted from the queue.
4. Searching an element O(N) time.
5. Maximum size of a queue must be defined prior.

* Circular queue →

1. A circular queue is the extented version of regular queue. where the last element is connected to the first element. Thus forming a ~~circul~~ circle-like structure.
2. The operations are performed based on FIFO principle. It is also called 'Ring Buffer'.
3. In a normal queue, we can insert elements until queue become full.



But once queue becomes full, we can not insert next elements even if there is a space in front of queue.

4. Operations in circular queue:

a. Front: Get the front item from queue.
b. Rear: Get the last item from queue.
c. enqueue(): Used to insert an element ~~at~~ into the circular queue. the element is always deleted from ~~from front~~ Rear position.

Teacher's Signature:

i) enqueue(): The queue is full or not, initially the front and rare position are said to -1. Then we insert a new element ~~the rare gets~~ in the queue the front and rare, both are said to zero.
When we insert the new element, the rare gets incremented (rare = rare + 1).

A] Cases in which queue is not full:
i) If rear != MaxSize - 1 then rear will be incremented to mod (MaxSize) and the new value will be inserted at the rare end of the queue.
ii) If front != 0 ~~and~~ && rear == MaxSize - 1.

B] Two case in which element cannot be inserted:
i) When front == 0 & rear = MaxSize - 1, this means front is at front position and rear is at last position.

C] Algorithm to insert an element in a circular queue:

Step 1 → · If (rear + 1) % max = front
write "Overflow".
· Go to Step 4
· End of if

Step 2 → If front = -1 && rear = -1
Set front = rear = 0
else if rear = MaxSize - 1 && front != 0
Set rear = 0.
else
Set rear = (rear + 1) % MaxSize
End of if

Teacher's Signature:

Step 3 → Set queue [rear = val]
Step 4 → Exit

ii) dequeue ():
The ~~first~~ steps of dequeue operation ~~is~~ are:
a. → First we check whether the queue is empty or not (If queue is empty we cannot apply the dequeue operation).
b. → When the ~~value~~ element is deleted, the value of queue is incremented by 1.
c. → If there is only one element left which is to be deleted, then the front and rear are set to -1.

A] Algorithm to delete an element from the circular queue.

Step 1 → If front = -1
write "Underflow"
Go to Step 4.

Step 2 → Set val = queue [front]

Step 3 → If front == rear
Set front = rear - 1
else if front = ~~=~~ MaxSize - 1
~~if~~ Set front = 0
else
Set front = front + 1
end if

Step 4 → Exit

Teacher's Signature:

30/07
Fri

*

Queue implementation using array:

```
from array import Array
class Queue:
    def __init__(self, maxSz):
        self._count = 0
        self._front = 0
        self._back = maxSz - 1
        self._qArr = Array(maxSz)
```

Create an empty queue.

```
    def isEmpty(self):
        return self._count == 0
```

Returns True if the queue is empty.

```
    def isFull(self):
        return self._count == len(self._qArr)
```

Return True if the queue is full

```
    def __len__(self):
        return self._count
```

Returns the number of item in the queue.

```
    def enqueue(self, item):
        assert not self.isFull(), "Queue is Full."
        maxSize = len(self._qArr)
        self._back == (self._back + 1) % maxSz
        self._qArr[self._back] = item
        self._count += 1
```

Adds the given item to the queue.

```
    def dequeue(self):
        assert not self.isEmpty(), "Empty queue!!"
        item = self._qArr[self._front]
        maxSz = len(self._qArr)
        self._front = (self._front + 1) % maxSz
        self._count -= 1
        return item
```

Removes and returns the first item in the queue

Teacher's Signature:

* Priority Queue:

1. This is a queue in which items are assigned a priority and the item with a higher priority are dequeued first.
2. There are two basic types of priority ~~queue~~ queues:

a. → Bounded

b. → Unbounded

The bounded priority queue assumes a small limited range of 'P' priorities over the interval of integers. [0, P)

The unbounded places not limit on the range of integer value that can be used as priorities.

Teacher's Signature:.....

03/08
Wed

PAGE No.
DATE / /

* Abstract data-type →

1. An ADT is a programmer defined datatype that specifies a set of data values & collection of well-define operations that can be performed on those values.
2. ADT's are defined independently of their implementation allowing us to focus on new data type instead of how it is implemented.
3. Other user programs interact with instances of ADT by invoking one of the several operations defines by its interface.
4. This set of operations are grouped into following categories →

a. Constructors: Create and initialize new instances of ADT.

b. Assessors: Return data content in an instance without modifing it.

c. Mutators: Modify the contents of ADT and return in instance.

d. ~~Itet~~ Iterators: Process individual data components sequencially.

Teacher's Signature:.....

* The date() as ADT →

Operations:

a. Date(month, day, year) – It creates a new date instance initialize to a given gregorian date, which must be valid.
b. day() - Returns the day number of ~~this~~ this date.
c. month() - Returns the month number of ~~this~~. this date.
d. year() - Returns the year number of this date.
e. monthName() - Returns the month name.
f. dayofWeek() - Returns the day of the week as a number between 0-6 with 0 representing monday and 6 representing sunday.
g. numDays(OtherDate) - Returns the number of days as positive integer between this date and the other date. (~~Fi~~ Returns duration)
h. isLeapYear() – Determines if this date falls in a leap year and returns the appropriate boolean value.
i. advanceBy(days) - It advances the date by even number of days. The date is incremented if days is positive & decrement when days is negative.
j. comparable(otherdate) - Compares these date to the other date to determine their logical ordering and comparison can be done using any logical operator.
k. tostring() - Returns a string representing the representing the gregorian date in a format mm/dd/yyyy.

Teacher's Signature:

08/08
Mon

PAGE NO.
DATE / /

* Priority Queue →

A bounded priority queue restricts the priorities to the ~~integral~~ integer value between 0 and predefine upper limit. ⟹

The Operations are :

a. PriorityQueue() → It creates a new empty unbounded Priority Queue.

b. BPriorityQueue(numlevels) → It creates a new empty bounded priority queue with priority level in the range from 0 to (numlevel-1).

c. IsEmpty() → It Returns boolean value, whether queue is empty or not.

d. length() → It Returns the total number of items in a priority queue.

e. enqueue(item, priority) → Add the given item to the queue, by inserting it in the proper position, based on the given priority. The priority value must be within the proper range for bounded priority queue.

F. dequeue() →

* Tree data structure:

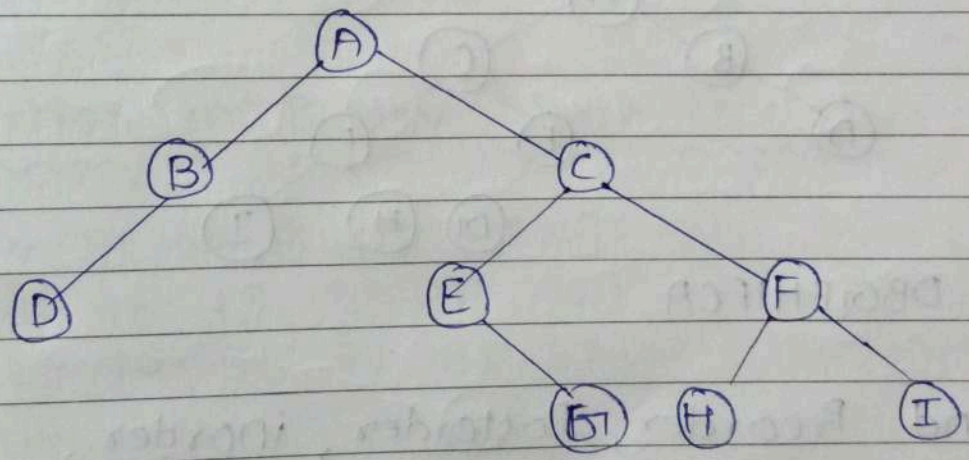
- A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such ~~as~~ that each node of the tree stores a value and a list of references to other nodes ("The children").

1. Traversal in tree data structure:

* Binary Search tree →

a. Preorder Traversal →

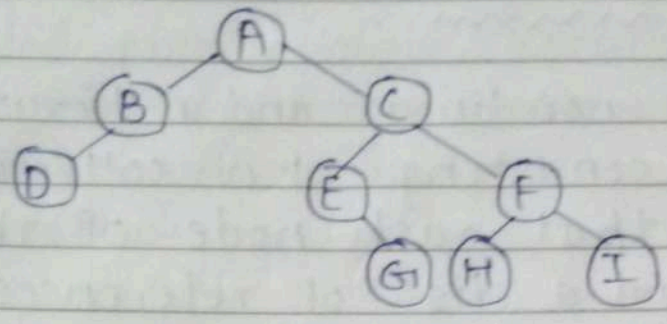
1) Visit root
2) Traverse left subtree
3) Traverse Right subtree.



→ ABDCEGFHI

b. InOrder Traversing: (LNR)

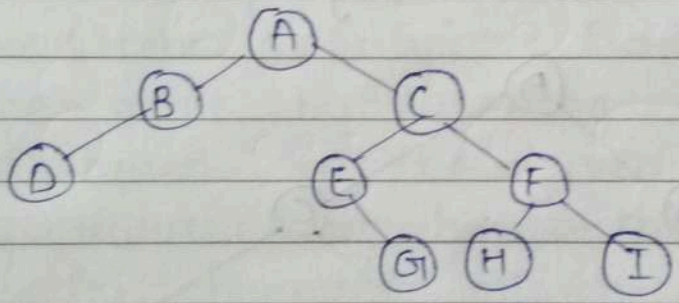
1) Traverse left subtree
2) Visit tree
3) Traverse right subtree.



⇒ DBAEGCHFI

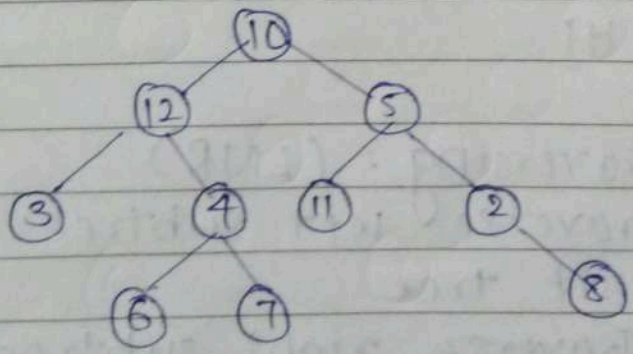
C. PostOrder Traversal : (LRN)

1) Traversal left subtree
2) Traverse right subtree
3) Visit root.



⇒ DBGEHIFCA

Q. Find Preorder, Postorder, inorder.



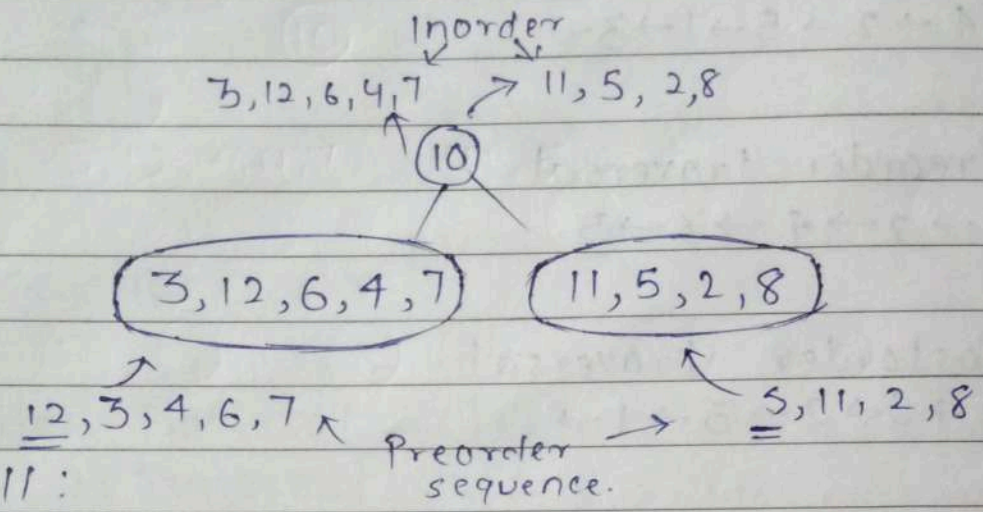
Preorder: 10, 12, 3, 4, 6, 7, 5, 11, 2, 8.
Inorder: 3, 12, 6, 4, 10, 11, 5, 2, 8.
Postorder: 3, 6, 7, 4, 12, 11, 8, 2, 5, 10.

Teacher's Signature:

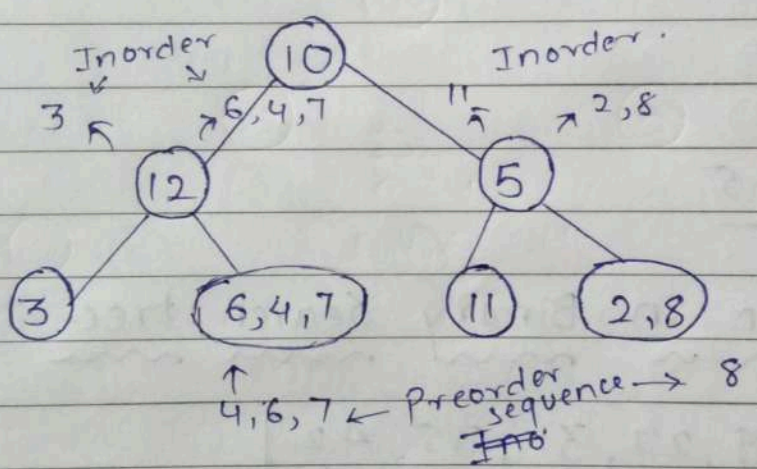
* In preorder root is in first position.

Preorder: 10, 12, 3, 4, 6, 7, 5, 11, 2, 8.
Inorder: 3, 12, 6, 4, 7, 10, 11, 5, 2, 8.

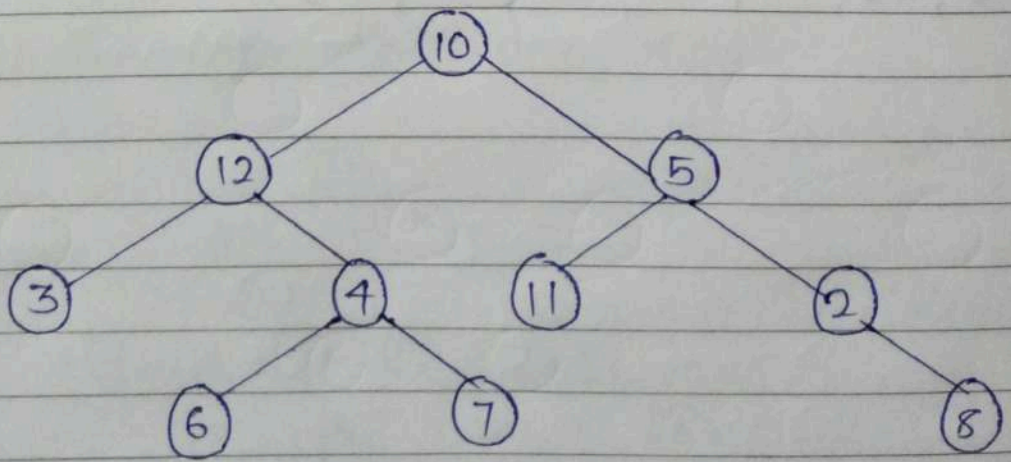
Step I:



Step II:



Step III:



*In Postorder root is in last ~~posti~~position.

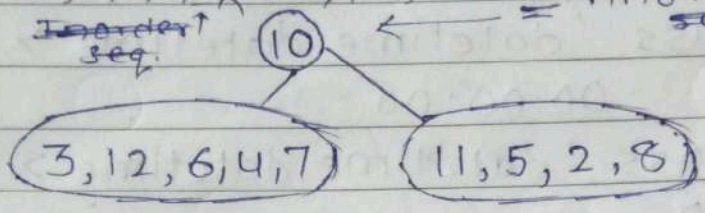
Postorder: 3, 6, 7, 4, 12, 11, 8, 2, 5, 10

Inorder: 3 12 6 4 7 10 11 5 2 8.

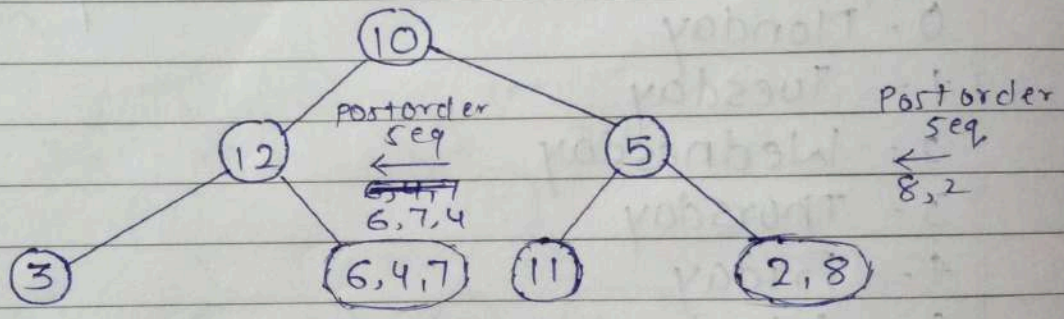
postorder seq: 3,6,7,4,12 ~~Postorder~~ 11,8,2,5 postorder seq

~~Inorder seq.~~ ~~Inorder seq.~~

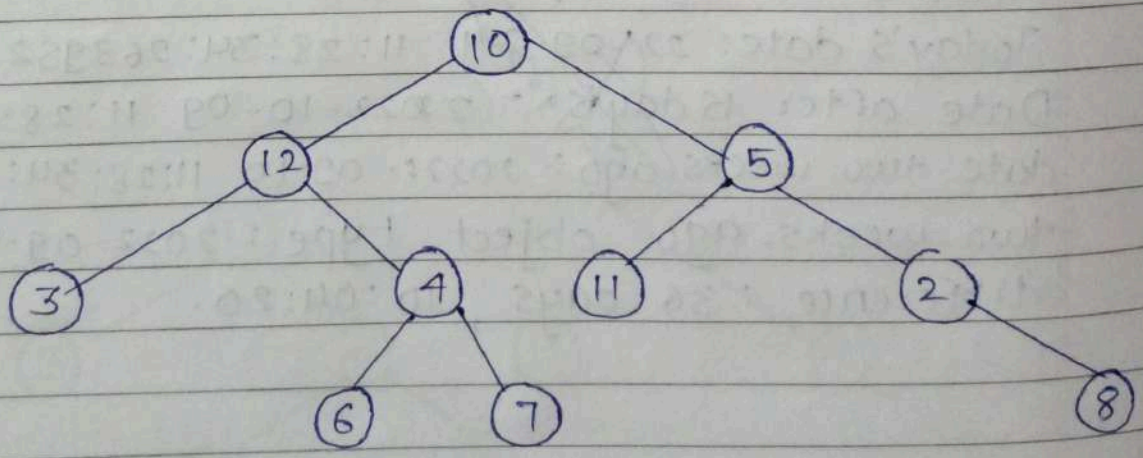
Step I:



Step II:



Step III:



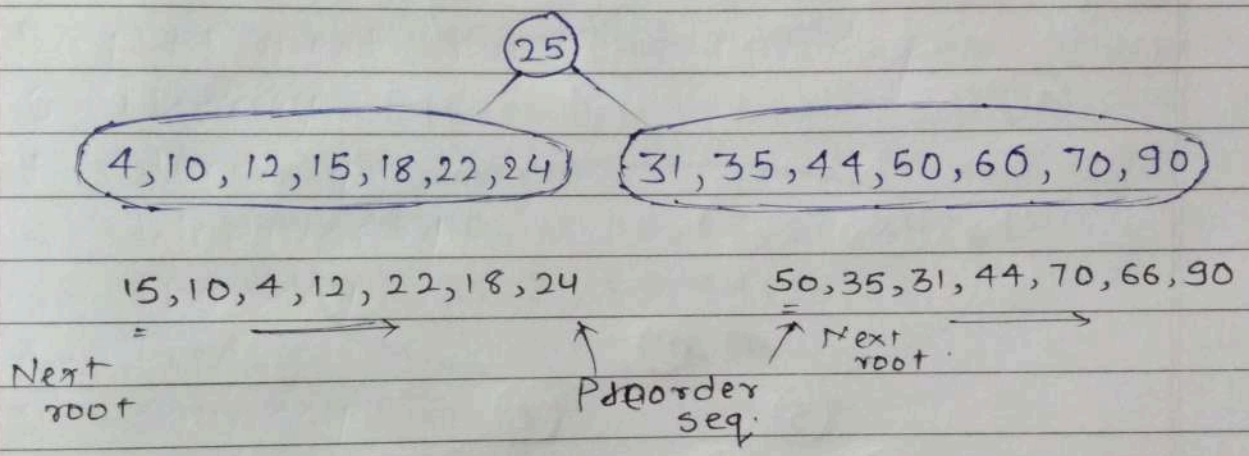
Teacher's Signature:

Q. H.W Draw a tree for given orders.

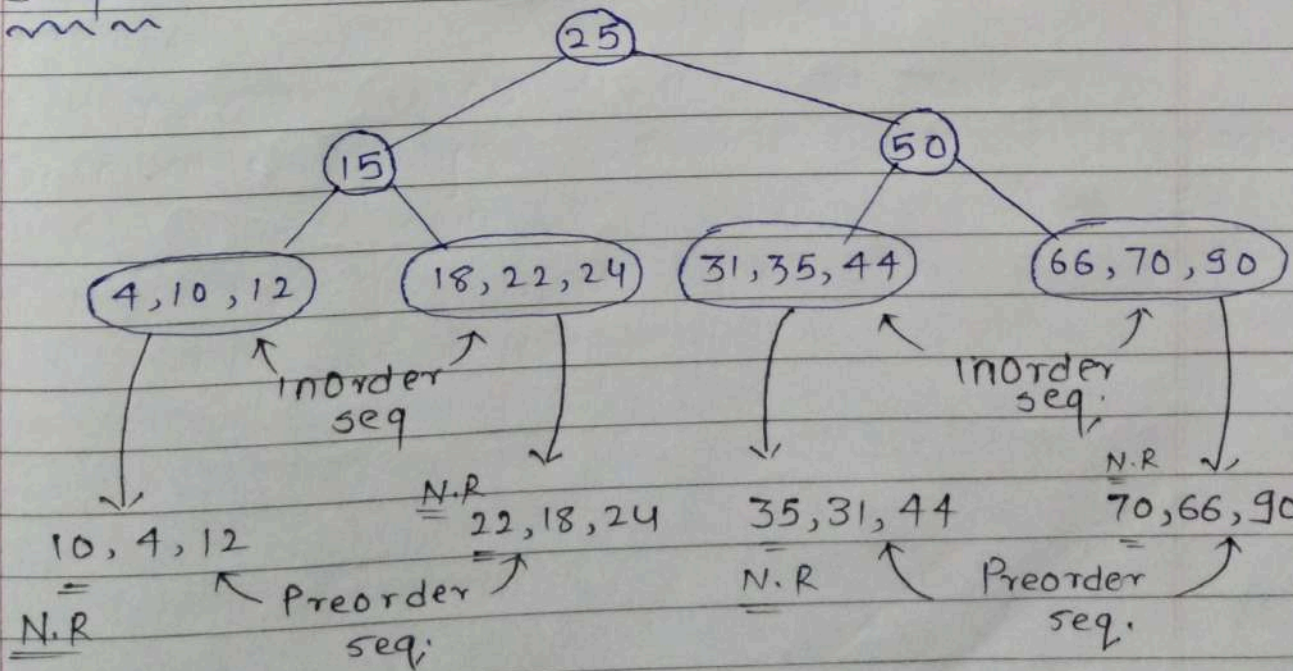
seq→ Preorder: 25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, ~~30~~ 70, 66, 90.

Inorder: 4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90.

Step I: i. In preorder, root is in first position.



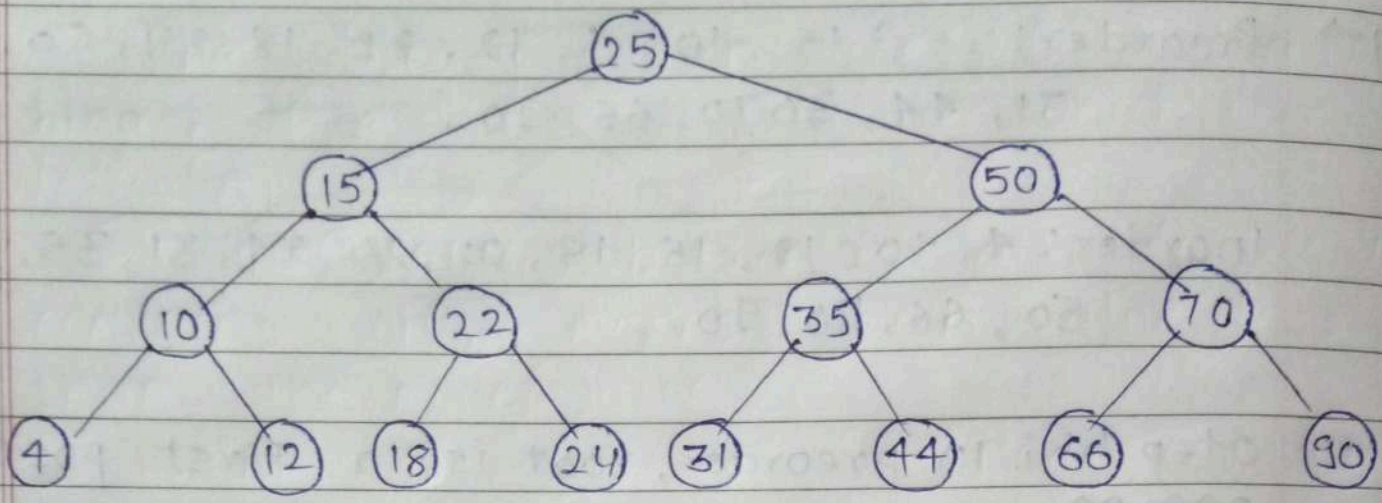
Step II:



Teacher's Signature:.....

PAGE No.
DATE 11

Step III :



DATE / /

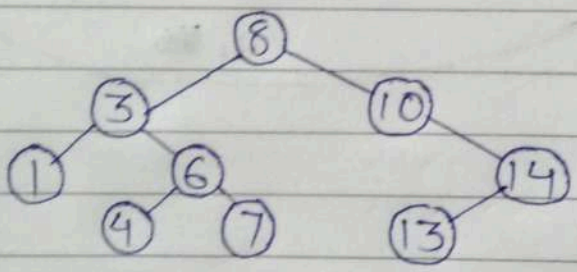
* Binary Search tree:

The binary search tree (BST) is a binary tree data structure which has following properties →

1. The left sub-tree of a node (root) contains only nodes which value less than the node value.

2. The right sub-tree of a node contain only nodes with value greater than the node value.

3. The left and right subtree must also be binary search tree. (no duplication of the nodes.)

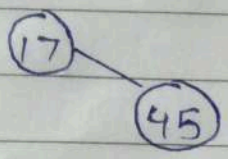


Teacher's Signature:

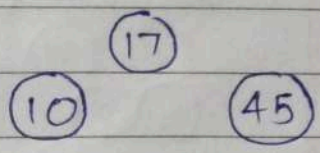
Q. Draw a binary search tree for the following sequence.

1. 17, 45, 10, 38, 61, 5, 53, 70, 65, 13.

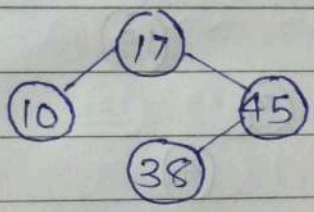
Step I:



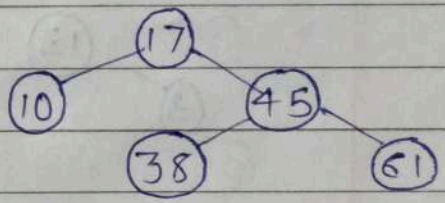
Step II:



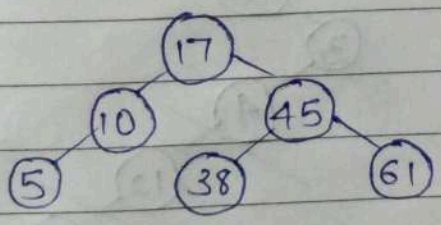
Step III:



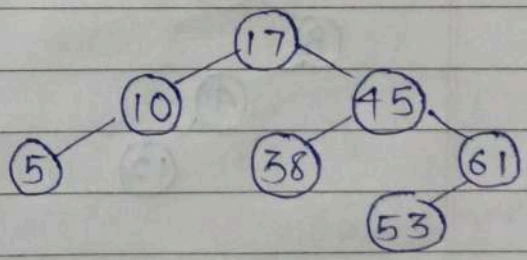
Step IV:



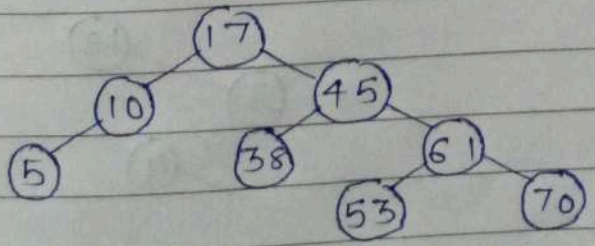
Step V:



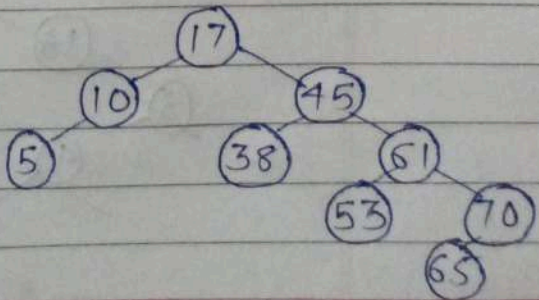
Step VI:



Step VII:

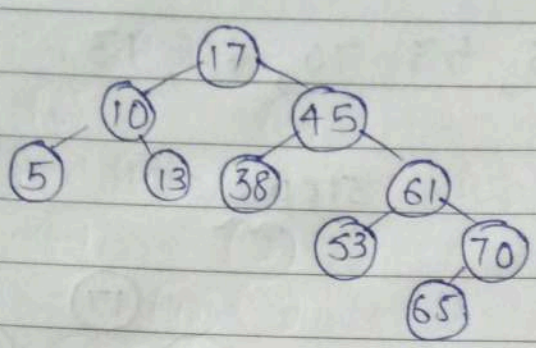


Step VIII:



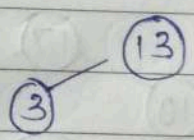
Teacher's Signature:

Step IX :

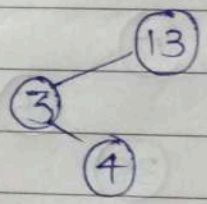


Q. 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18.

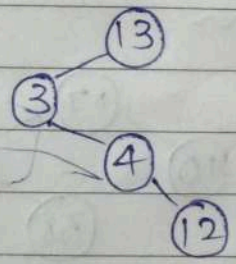
Step I :



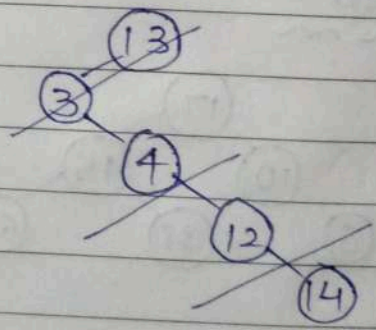
Step II :



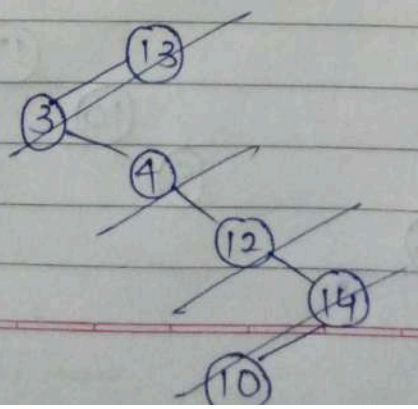
Step III :



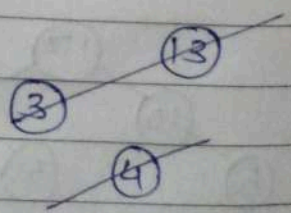
step IV :



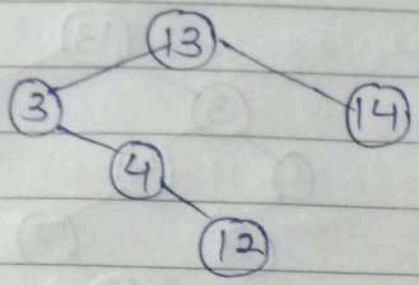
Step V :



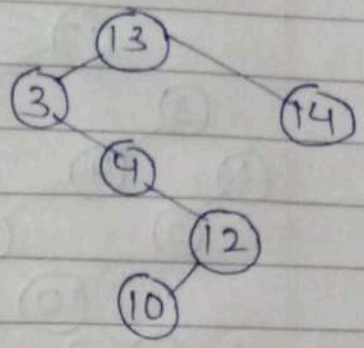
Step VI :



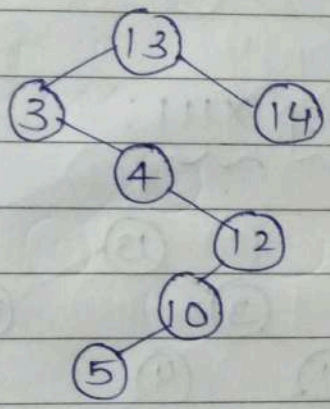
Step IV :



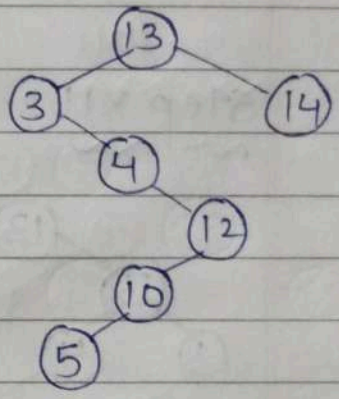
Step V :



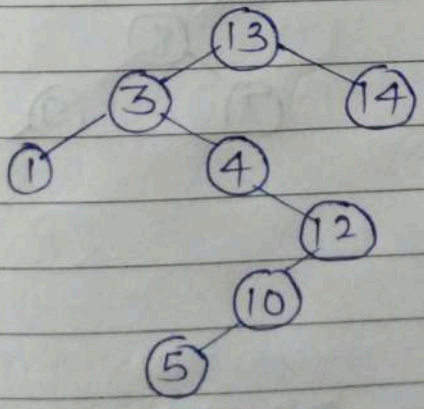
Step VI :



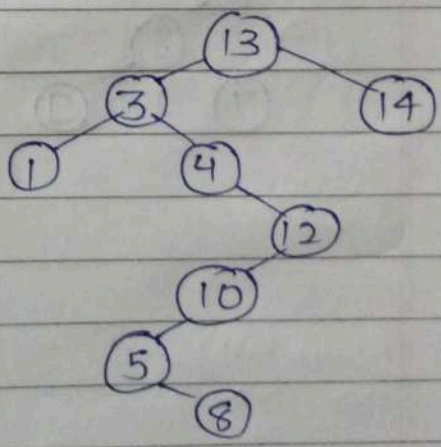
Step VII :



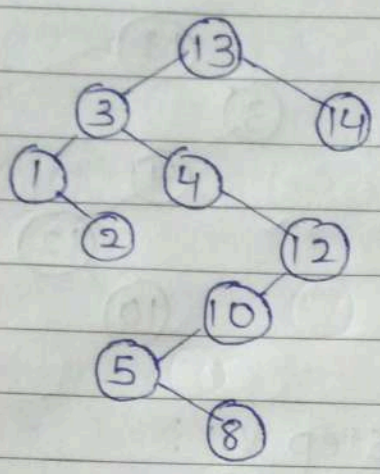
Step VIII :



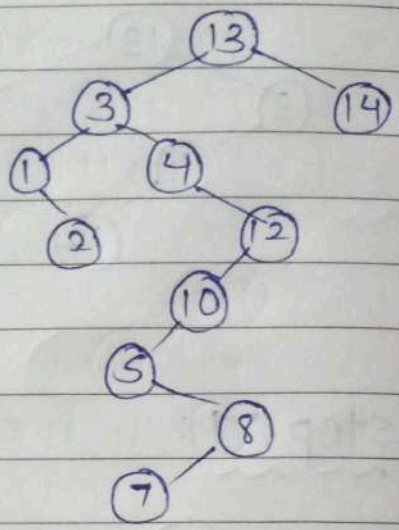
Step IX :



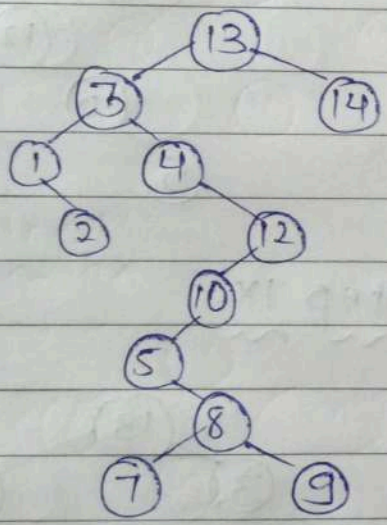
step X:



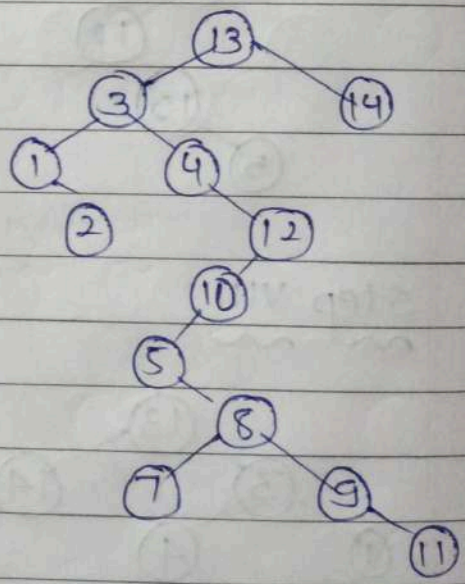
step XI:



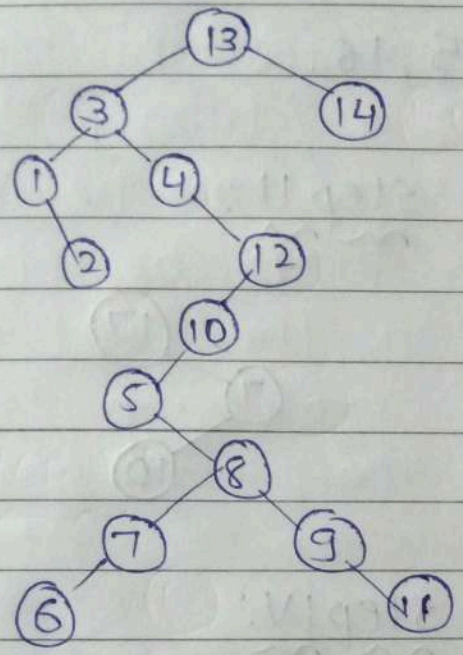
step XII:



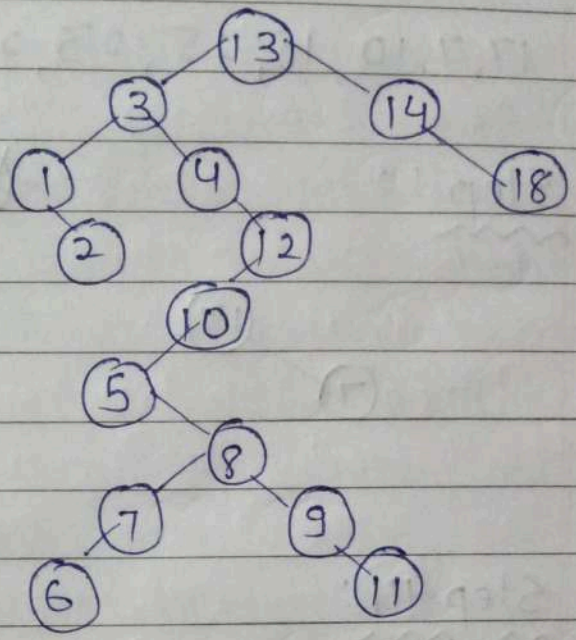
step XIII:



step XIV :



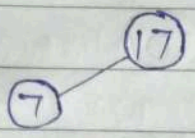
stepXV:



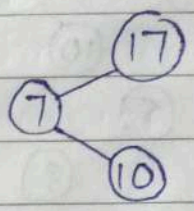
Q. Draw a binary tree using following sequence.

Seq ⇒ 17, 7, 10, 18, 25, 23, 20, 5, 15, 16.

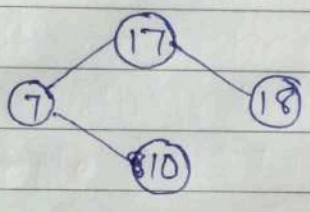
Step I:



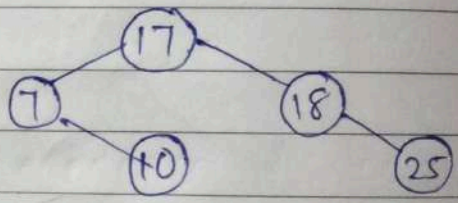
Step II:



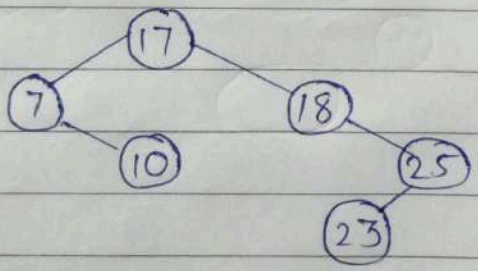
Step III:



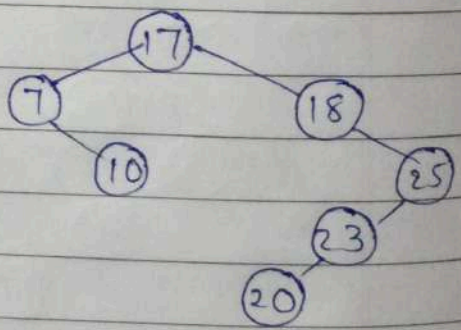
Step IV:



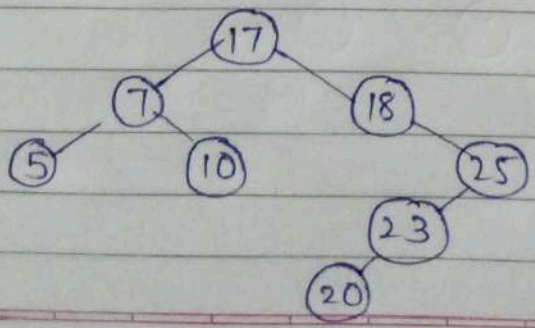
Step V:



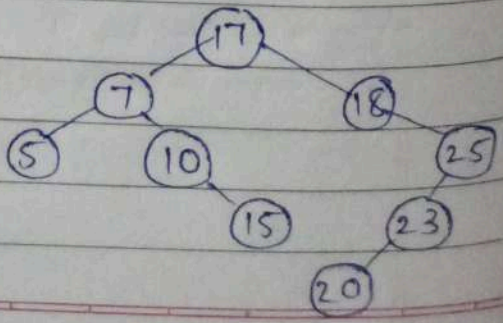
Step VI:



Step VII:

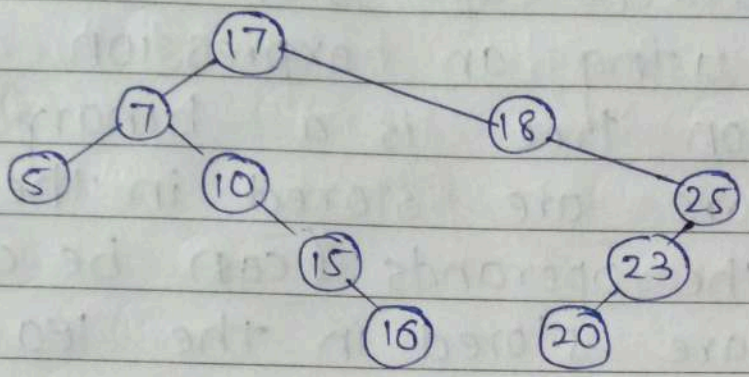


Step VIII:



Teacher's Signature:

Step IX:



* Expression tree is an representation of an expression in a tree like data structure.
* Leaves = Operands (external nodes),
  Internal node = operators.

PAGE No.
DATE / /

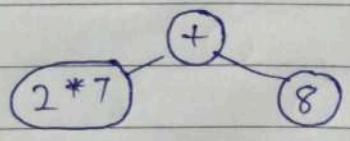
* Expression tree:

A mathematical expression can be represented using an expression tree.
An expression tree is a binary tree in which operators are stored in the internal node, and the operands (can be constant or variable) are stored in the leaf nodes.
Once constructed an expression tree can be used to evaluate the expression or for converting an expression to either infix or postfix notation.
The structored of the expression tree are based on the order in which the operators are evaluated.

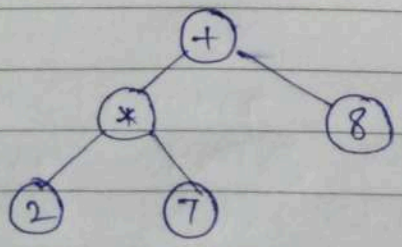
Example: i) 2*7+8.

Precedence	Associativity
^	Right → Left
* /	Left → Right
- +	Left → Right

step I:



step II:



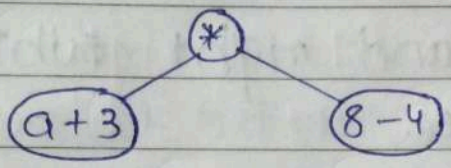
Prefix: + * 278.
Postfix: 27 * 8 +

Inorder: Left Root Right
Preorder: Root Left Right (for ~~###~~ Prefix)
Postorder : Left Right Order .(For Postfix)

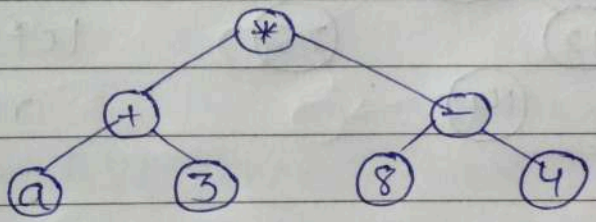
PAGE No.
DATE / /

2) (a+3) * (8-4)

Step I :



Step II :



Prefix: *+93-84
Postfix: 93+84-*

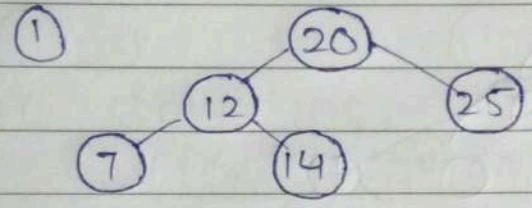
Height of left subtree - Height of Right subtree ≤ 1

PAGE No.
DATE / /

* **Balanced Binary Search tree:**

A Balanced Binary Search tree is also know as height balanced tree. The difference between the height of left subtree and right subtree is not more than one.

Example:

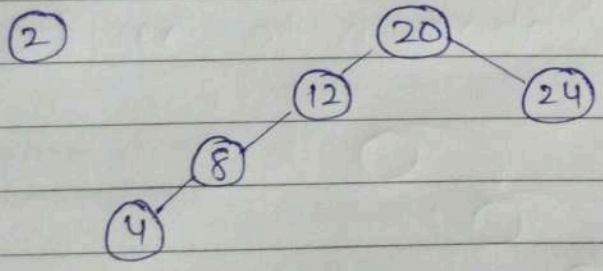


Height = ~~tel~~ Height of left subtree - Height of right subtree.

= 2 - 1

= 1

=



Height = Height of left subtree - Height of right * subtree

= 3 - 1.

= 2

=

* Following are the condition for the balanced BST.

1. Difference between left subtree & right subtree should not greater than 1.
2. The left subtree should also be balanced.
3. The right subtree should also be balanced.

20/09

PAGE No.
DATE / /

* AVL Tree : (Adelson Velsky & Landis)

- It can be defined as the height balanced tree. in which each node is accociated with a balanced factor is calculated by subracting the height of right subtree - height of left subtree. The balanced factor of the nodes should be between -1 to 1 otherwise, the tree is said to be unbalanced which need to be balanced.

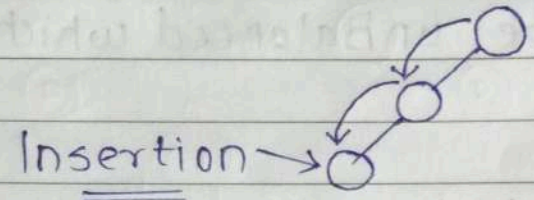
1) Insertion in AVL tree:
The operation can be performed same as that of BST however it may lead to violation in the AVL tree defination, tree need to be balanced, by applying the rotations.

2) Deletion in AVL tree:
The deletion operation can be performed same as that of BST however it may lead to voialation in the AVL tree definition, tree need to be balanced, by applying the rotation.

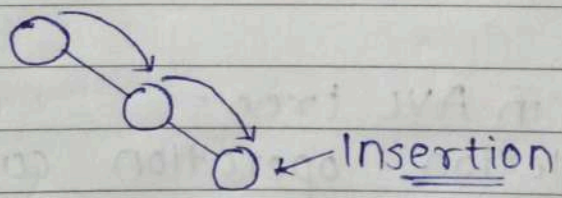
* AVL Rotations:

The rotation will be performed on AVL tree only if case of balanced factor value is no other than -1, 0, -1.

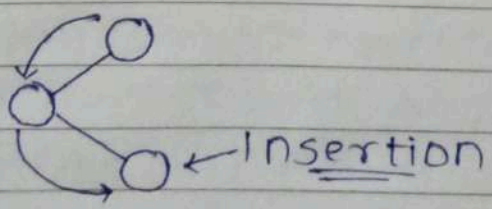
1. LL Rotation (Left right rotation): Inserted node is in the left subtree of the left subtree of a node.



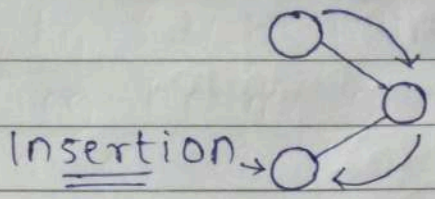
2. RR Rotation (Right Right rotation): Inserted node is in the right subtree of the right subtree of a node.



3. LR Rotation (Left-Right Rotation): Inserted node is in the left subtree of the right subtree of a node.

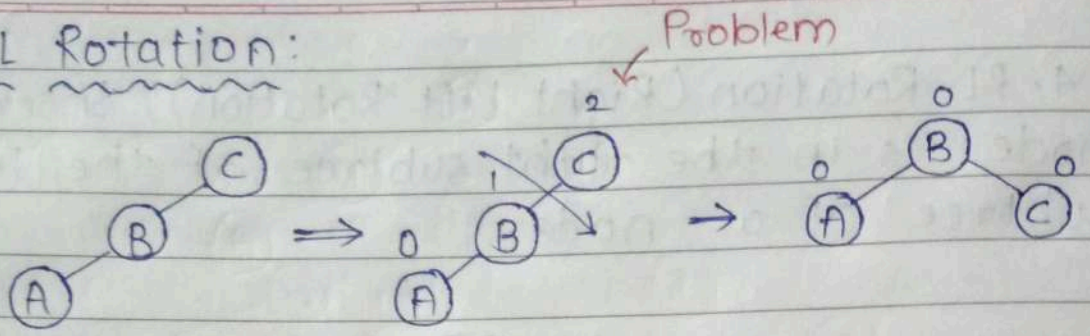


4. RL Rotation (Right Left Rotation): Inserted node is in the right subtree of the left subtree of a node.

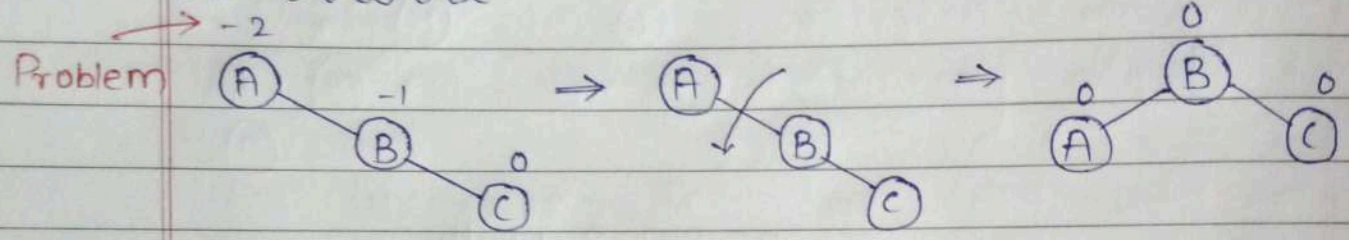


26/09

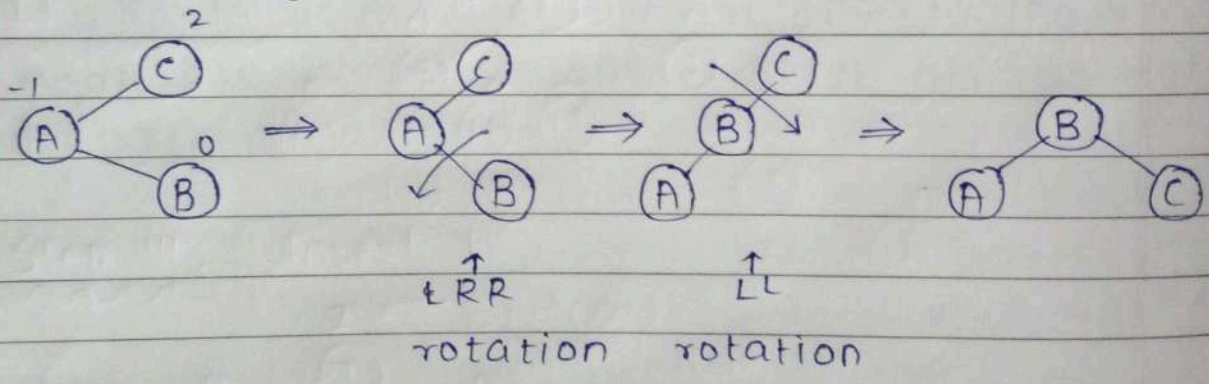
1. LL Rotation:



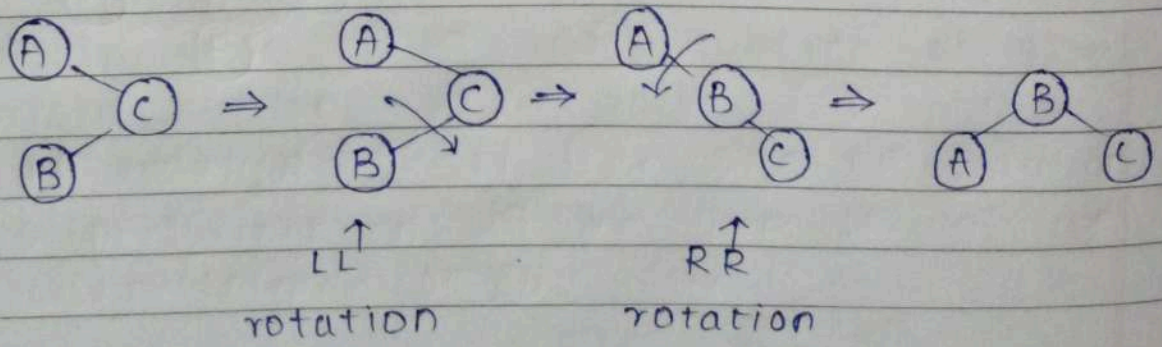
2. RR Rotation:



3. LR Rotation: (LR = RR + LL)



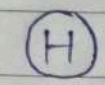
4. RL Rotation: (RL = LL + RR)



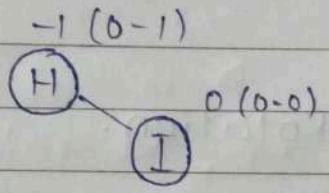
Q. Construct an AVL tree for following element:

Sequence → H, I, J, B, A, E, C, F, D.

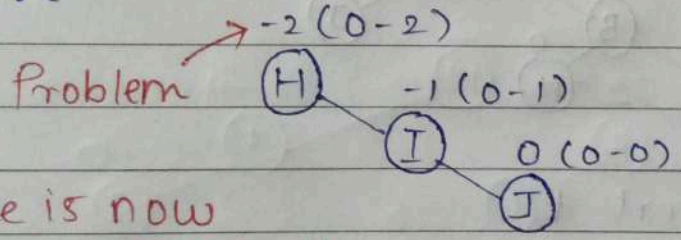
Step I: Insert H (root node)



Step II: Insert I.



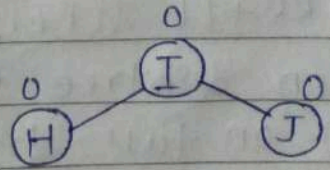
Step III: Insert J.



Tree is now unbalanced.

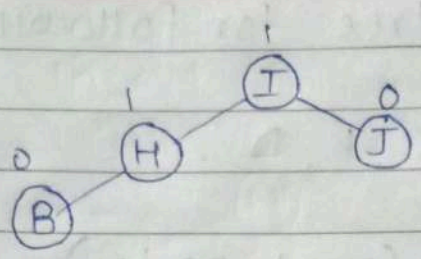
∴ Balancing factor = RR

Since J is inserted in the right subtree of right subtree of H, i.e. it'll do the RR rotation.

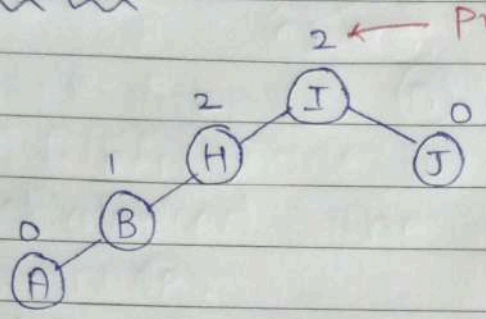


Step IV: Insert B.

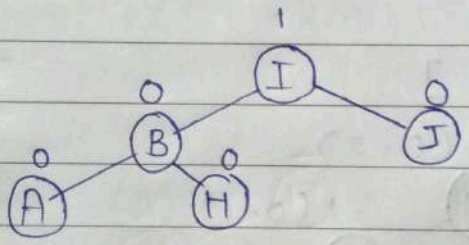
Teacher's Signature:.....



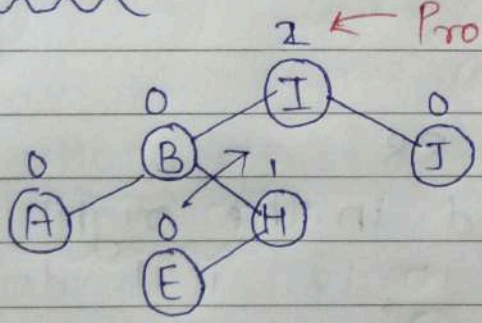
Step V: Insert A



⇊ LL Rotation.

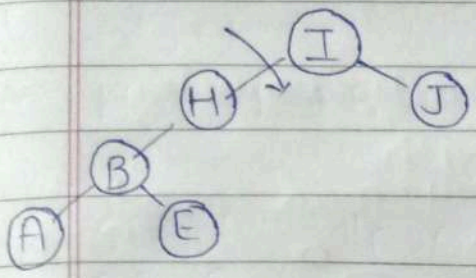


Step VI: Insert E

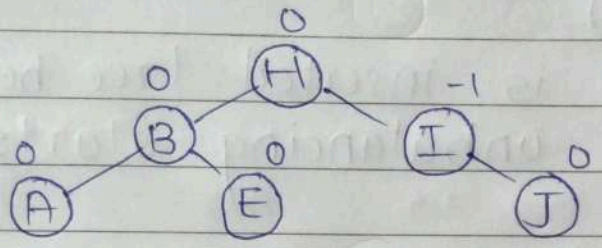


LR rotation : RR+LL ... i.e., one RR rotation is performed on subtree then LL rotation is performed on full tree.

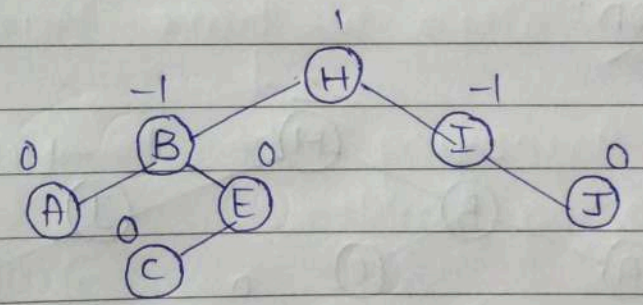
RR rotation:



LL rotation:

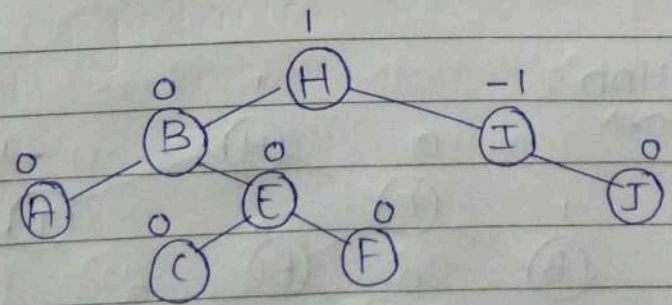


Step VII: Insert C



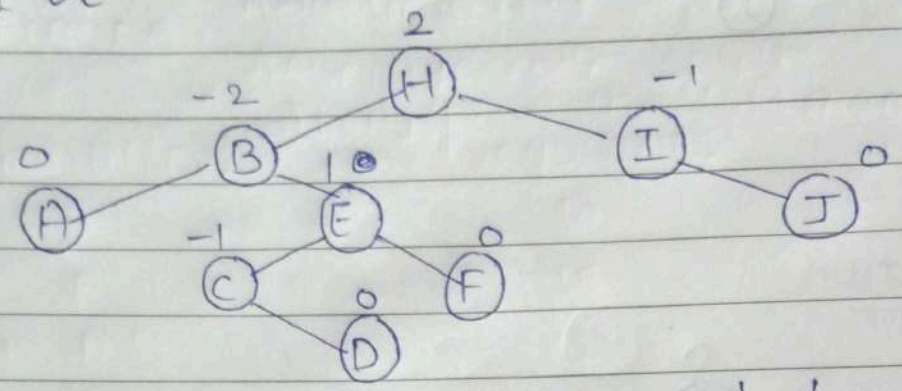
(Still balanced)

Step VIII: Insert F.



(Still balanced)

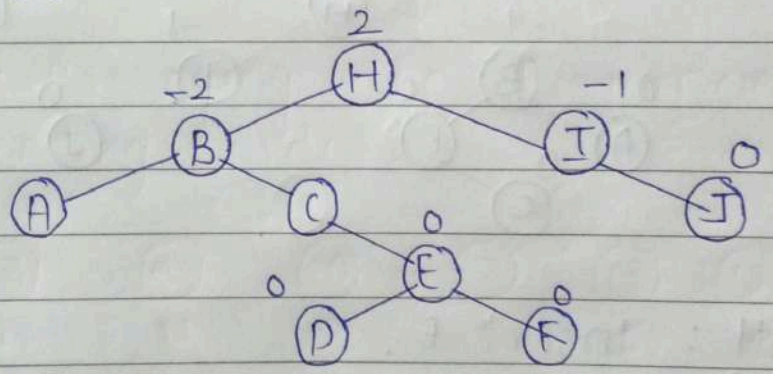
Step IX: Insert D



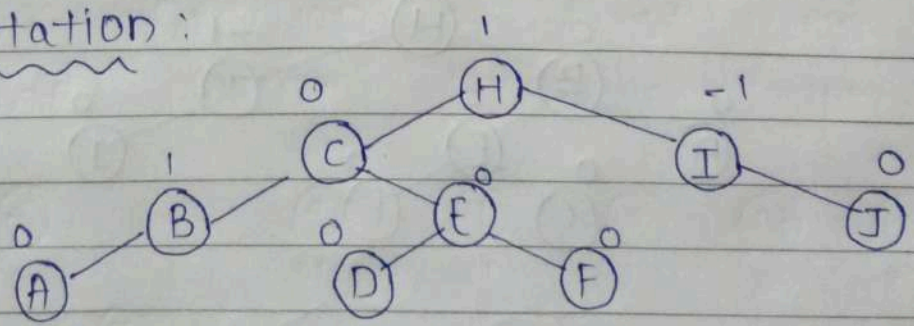
When node D is inserted tree become unbalanced, the unbalancing starts from B node.

For B node, D is at left subtree of right subtree, i.e., RL rotation will be done.

LL rotation:



RR rotation:



PAGE No.
DATE / /

* Practice questions:

Q. Construct a Binary search tree for the given order.

Preorder: 1, 2, 4, 5, 7, 3, 6, 8.
Inorder: 4, 2, 7, 5, 1, 8, 6, 3

Q. Construct the AVL tree for the given sequence.
1, 2, 3, 4, 5, 6, 7, 8.

DATE / /

* Huffman coding:

Q. Determine: 1) Huffman code for each character
2) Average code length
3) Length of the Huffman encoded mssg (in bit

Characters	Frequencies
A	10
E	15
I	12
O	3
U	4
S	13
T	1

→ First let us construct the Huffman tree using the steps we have learnt above:

Step-01: Arranging the frequencies in ascending order.

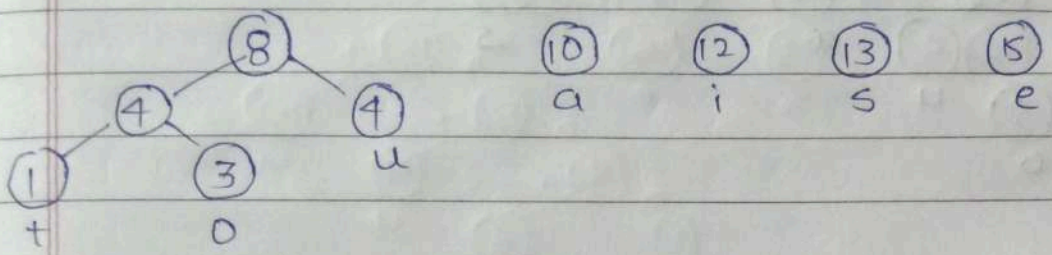
(1)	(3)	(4)	(10)	(12)	(13)	(15)
t	o	u	a	i	s	e

Step-02: Addition of first two frequencies and placing them in proper order.

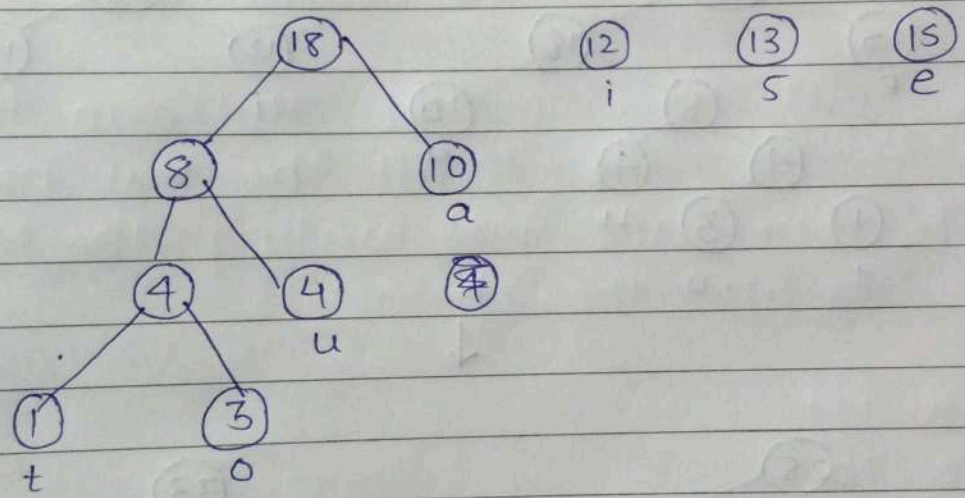
(4)		(4)	(10)	(12)	(13)	(15)
		u	a	i	s	e
(1)	(3)					
t	o					

Teacher's Signature:.....

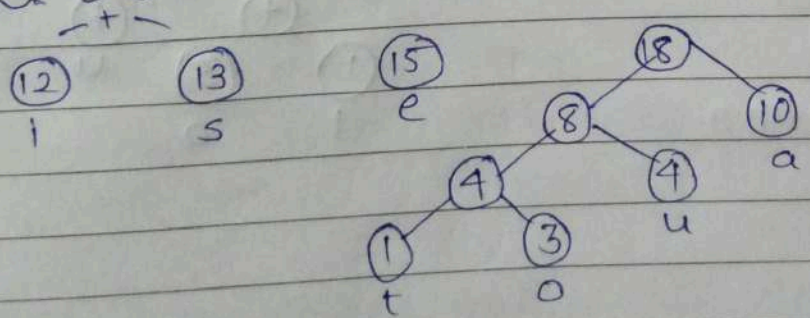
Step - 03:



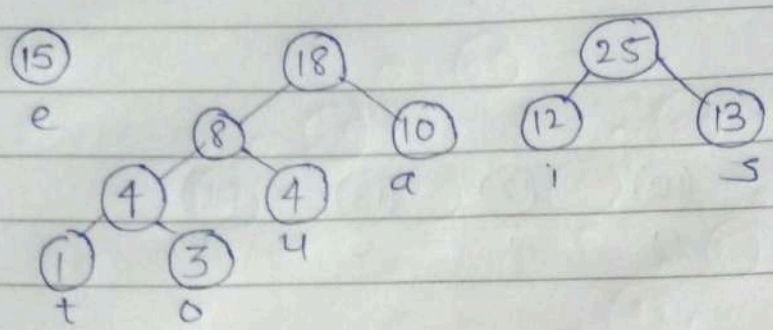
Step - 04:



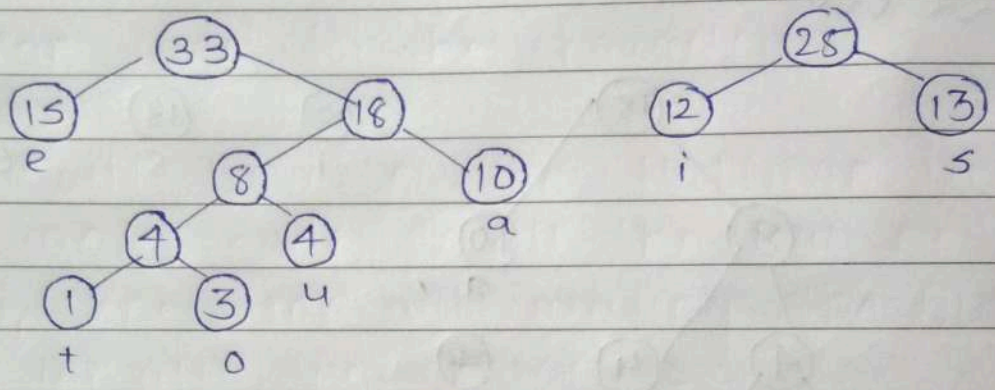
Step - 05:

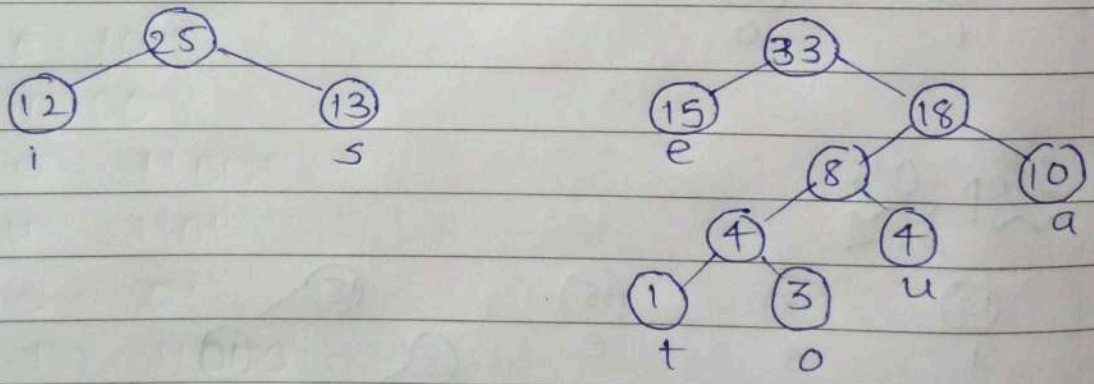


Step - 06:

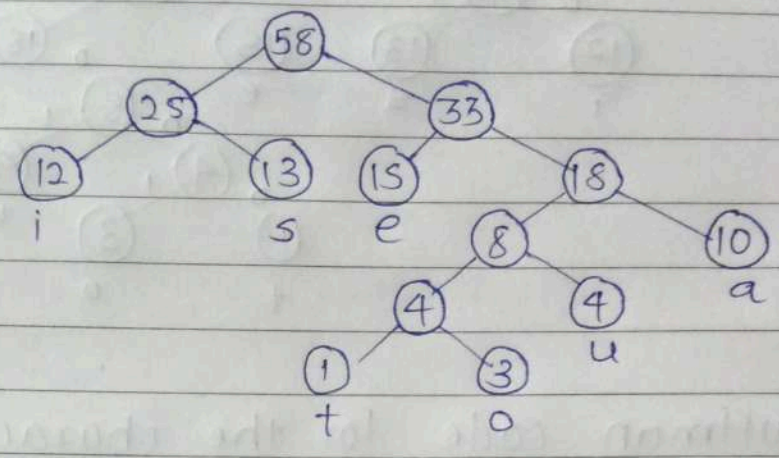

15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
i
13
s

Step 07:


33
15
e
18
8
10
a
4
4
u
1
3
t
o
25
12
i
13
s


25
12
i
13
s
33
15
e
18
8
10
a
4
4
u
1
3
t
o

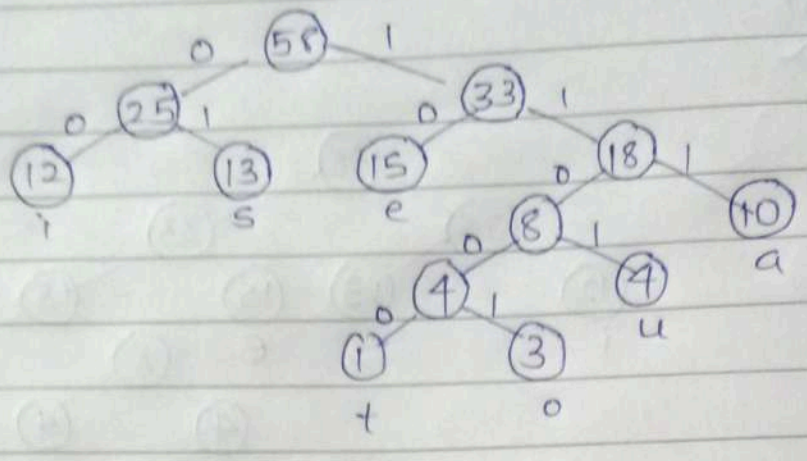
Step 08:



After we have constructed the Huffman tree, we will assign weights to all the edges. let us assign weight '0' to the left edge and weight '1' to the right edges.

* Note:
- We can also assign weight '1' to the left edges and weight '0' to the right edges.
- The only thing to keep in mind is that we must follow the same convention at the time of decoding which we adopted at the time of encoding.

After assigning weight '0' to the left edges and weight '1' to the right edges, we get-



* Huffman code for the characters:

We will traverse the Huffman tree from the root node to all the leaf nodes one by one and will writing the Huffman code for all the characters.

a = 111
e = 10
i = 00
o = 11001
u = 1101
s = 01
t = 11000

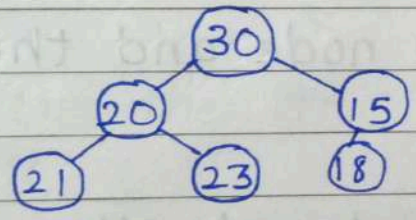
1/10

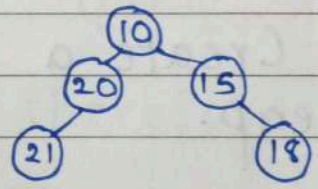
Heap tree:

1. It complete binary tree that satisfy the heap property.
2. According to heap property the root node is compare with his children and arrange accordingly.
3. As per this arrangement there are two type of heap tree:
   a) Min Heap → Value of root node is less than of its children.
   b) Max Heap → Value of root node is greater than its children.

Min Heap →

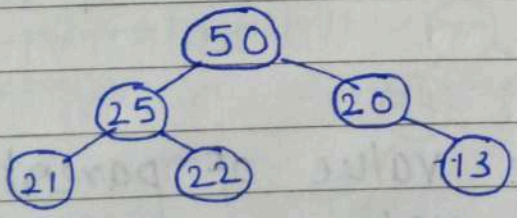
satisfy the property of BST. →

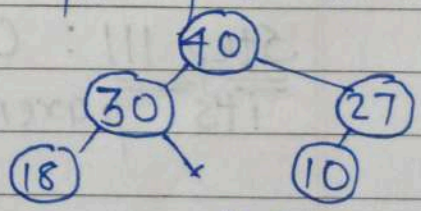




Not satisfy property of binary tree. because the children always start filling from left.

Max Heap →





not binary tree because Right side is empty.

* Operation of heap tree:

1. Heapify → A process of creating a heap from an array.
2. Insertion → Process of Inserting an element in an existing tree.
3. Deletion → deleting the top element from an existing heap. (Highest priority element) then organising the heap (Log in time).

* Insertion in a heap tree → (Max heap)

Step I: Create a new node and the end of the heap.

Step II: Assign the value to the new node

Step III: Compare the value of node with it's parent.

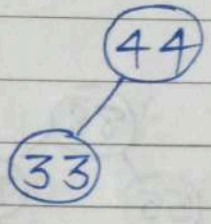
Step IV: If the value of parent is less than the child value swap them.

Step V: Repeat step (III) and (IV).

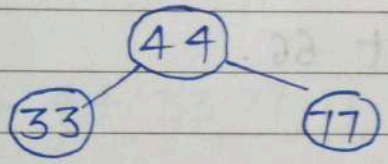
Example → 44, 33, 77, 11, 55, 88, 66.

Step I → Insert 44.

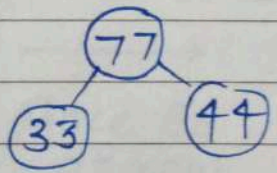
Step II → Insert 33.



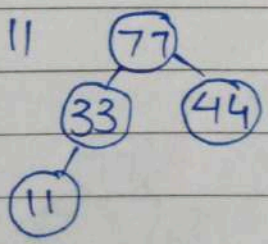
Step III → Insert 77.



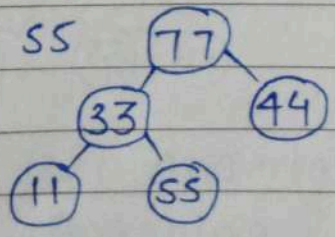
Here the 44 < 77, therefore, heapify (swap).



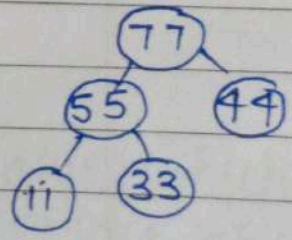
Step IV → Insert 11



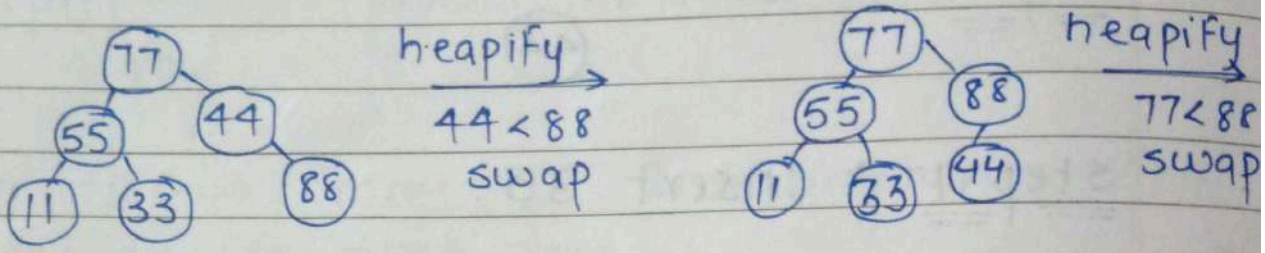
Step V → Insert 55

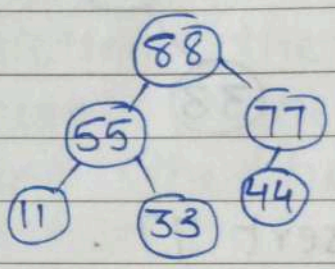


heapify → 33 < 55 swap

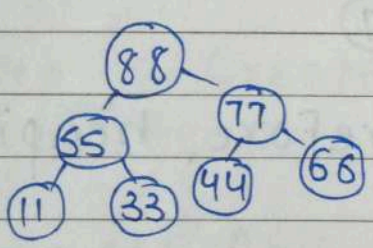


Step VI : Insert 88.





Step VII : Insert 66.



* Deletion operation. (Always start from root node)

Step I : Move the last element of the last lvl to the root.

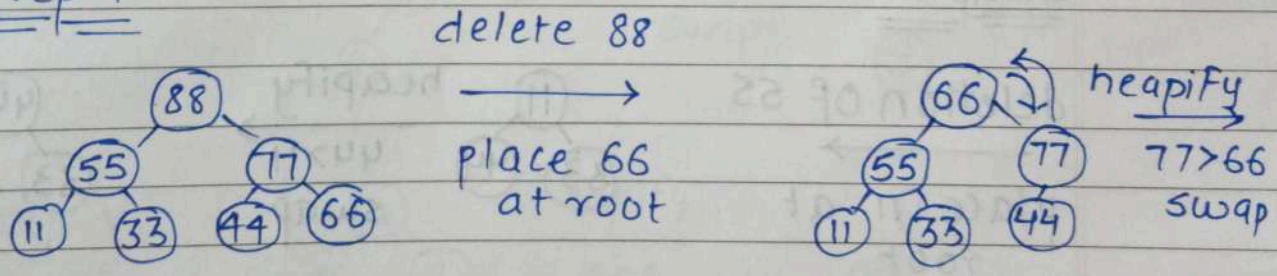
Step II: Compare the values of this child node with its parents.

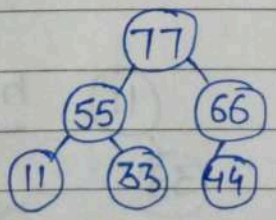
Step III : If value of the parent is less than the child swap them.

Step IV: Repeat step (III) and (IV) untill heap property hold.

* Example ⟶ 44, 33, 77, 11, 55, 88, 66.

Step I :





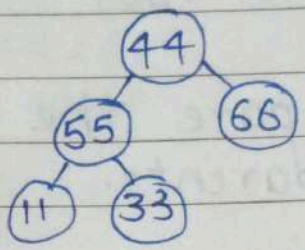
Compare the left and right child value, which ever is greater swapping will take place at that place.

Teacher's Signature

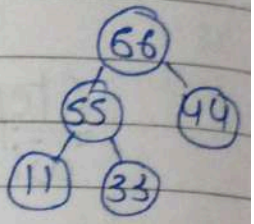
Do the swapping untill heap property hold.

Step II:

deletion of 77 → Place 44 at root

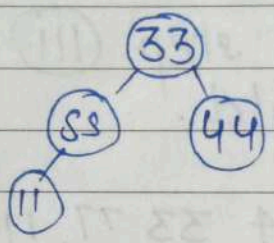


heapify → 66>44 swap

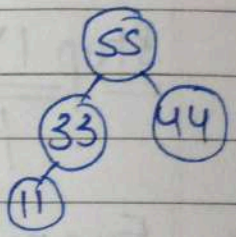


step ~~Swap~~ III:

deletion of 66 → Place 33 at root

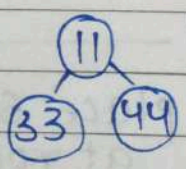


heapify → 55>33 swap

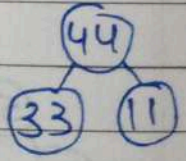


step ~~Swap~~ IV:

deletion of 55 → place 11 at root

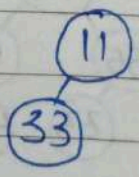


heapify → 44>11 swap

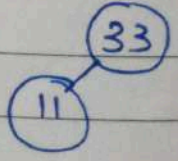


step ~~Swap~~ V:

deletion of 44 → Place 11 at root

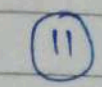


heapify → 33>11 swap



Step VI:

deletion of 33 → Place 11 at root.



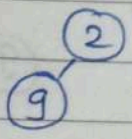
Teacher's Signature:

* Insertion in heap tree: (Min heap).

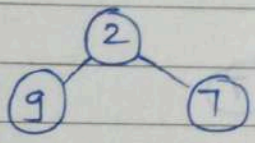
Q. 2, 9, 7, 6, 5, 8.

→ Step I :

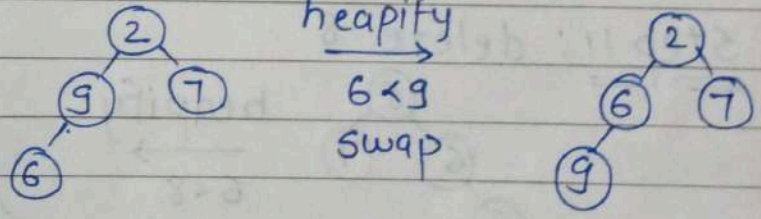
Step II :



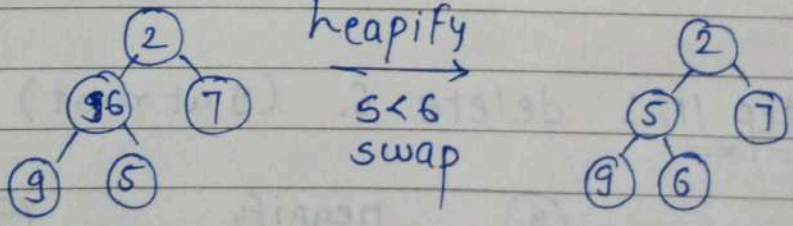
Step III :



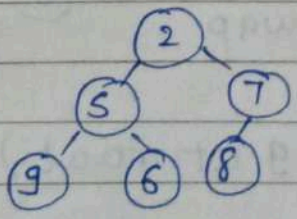
Step IV :



Step V :



Step VI :

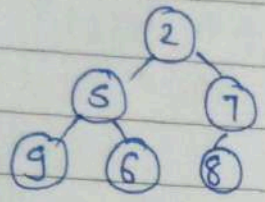


Teacher's Signature:

* Deletion in heap tree: (Min heap).

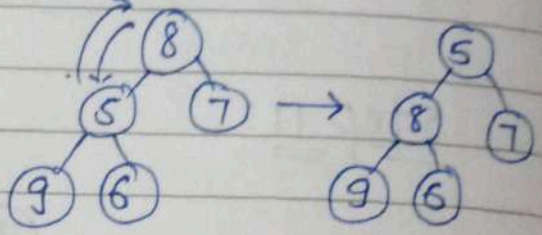
Q. 2, 5, 6, 7, 8, 9.

Step I : delete 2

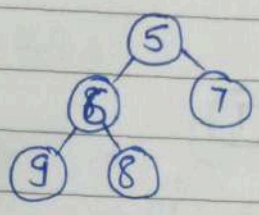


delete 2 → place 8 at root

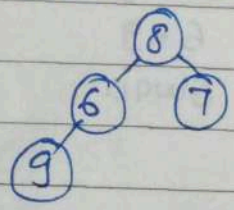
swap leastsmall.



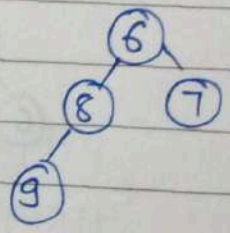
heapify →



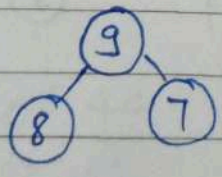
Step II : delete 8



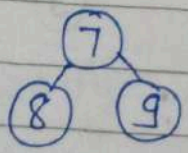
heapify → 6<8 swap



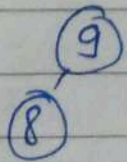
Step III : delete 6. (9 at root)



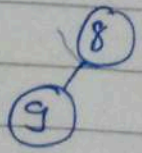
heapify → 7<9 swap



Step IV: delete 7 (9 at root)



heapify → 8<9 swap



Step V: delete 8 (9 at root)

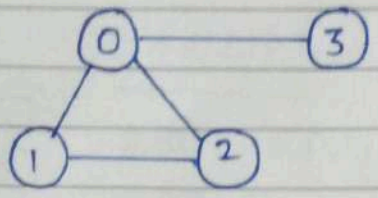
(9)

Teacher's Signature:

Graph →

It is a collection of nodes that have data and are connected to other nodes.

It is a (V, E) that consist of connection of vertices 'V' and collection of edges 'E' represented as order pairs of vertices.



$V = \{0, 1, 2, 3\}$

$E = \{(0,1), (0,2), (0,3), (1,0), (1,2), (2,0), (2,1), (3,0)\}$

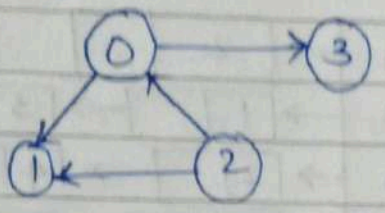
* Graph terminologies →

1. Adjacency: A vertex is said to be adjacent to another vertex if there is an edge connecting them.
   Example: Node '2' and '3' are not adjacent. Since, there is not edge between them.

2. Path: A sequence of edges that allows you to go from one vertices to another.

Teacher's Signature:

* Directed graph → A graph in which an edge (u,v) doesn't necessary, means that their is an edge (v,u).
The edges edge in such a graph are represented by arrow to show the direction of edges.



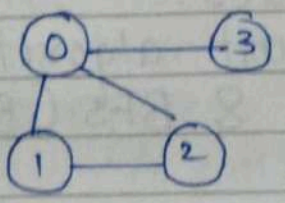
$V = \{0, 1, 2, 3\}$

$E = \{(0,1), (0,3), (2,0), (2,1)\}$

* Graph representation → Graph are commonly represented in two ways →

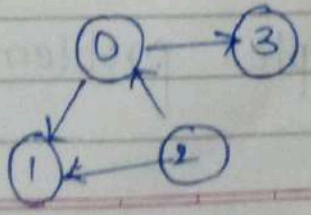
A. Adjacency matrix: A adjacency matrix is a 2D array of V x V vertices. Each row and columns represents a vertex.
If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex 'i' and vertex 'j'.

(1)



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

(2)

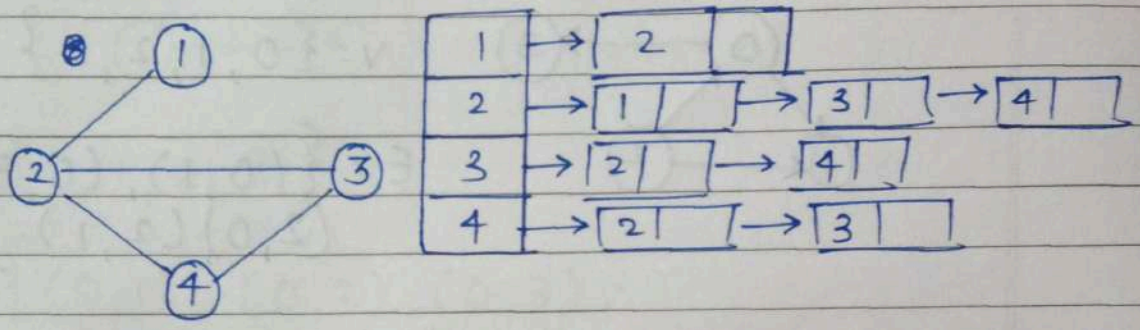


	0	1	2	3
0	0	1	0	1
1	0	0	0	0
2	1	1	0	0
3	0	0	0	0

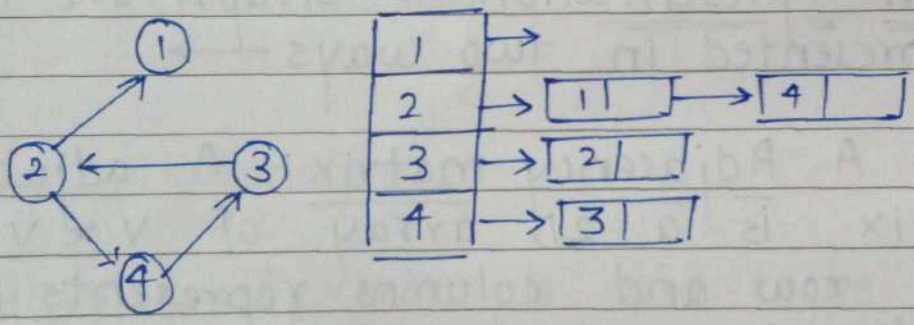
Teacher's Signature:

B. Adjacency List: An adjacency list represent a graph as an array of linked list. The index of the array represented a vertex and each element in it linked list. represent the other vertices that form an edge with a vertex.

E.g. →



E.g. →



* Advantages of graph →
- By using graph we can easily find shortest path, neighboors of the nodes.
- Graphs are used to implement algorithm like DFS (Depth ~~for~~ first Search) & BFS (Breadth first ~~for~~ search)
- Used to find minimum spanning tree.
- Helps in organizing data.
- Helps in understanding complex problems and their visualizations.

Teacher's Signature:

PAGE No.
DATE / /

* Disadvantage of Graph →

- Graph use lots of pointers, thus complex to handle.
- Can require large memory (pointers).
- If it is represented in matrix it doesn't allows parallel edges.

* Graph traversal →

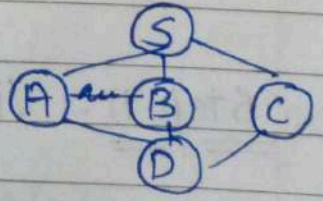
There are two types of graph traversal algorithm:

A. BFS → Breadth first Search.
B. DFS → Depth First Search.

A. BFS (Breadth first Search):

- It's a traversing algorithm that starts traversing the graph from a start node & explores all the neighbour node.
- Then it select nearest node & explores all unexplores nodes.
- It's a recursive one, it make use of queue data structure.
- The graph may contain cycle so we may visit same node again to avoid processing a node more than onee divide the vertices into two categories:

A. Visited
B. Not Visited.



Teacher's Signature:

* Implementation:

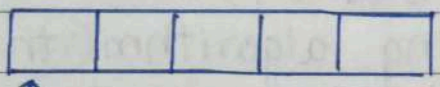
Step I: Start with a node.
Visit the adjacent unvisited vertices
Marked them as visited.
Insert it in the queue.

Step II: If no adjacent vertex is found removed the first vertex from the queue.

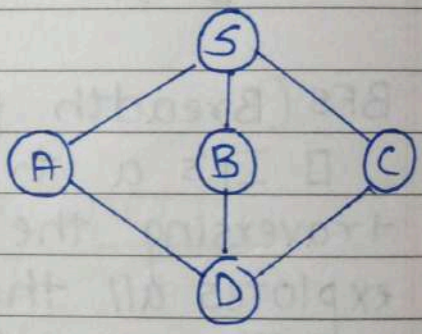
Step III: Repeat step (I) and (II), until the queue is empty.

e.g. →

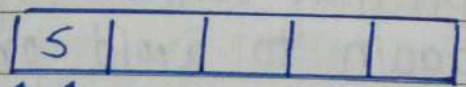
~~Empty~~ Queue.



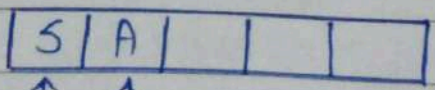
initialize the queue



Step I: Visit S and mark it as visited.

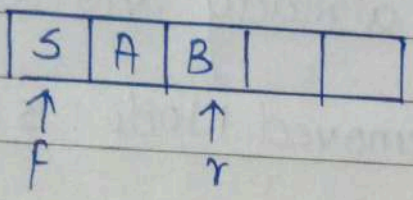


Step II: Visit the adjacent of A and marked it visited.

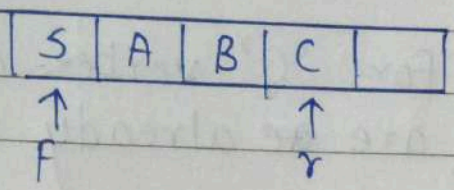


Teacher's Signature:

→ Insert B & marked it as visited.

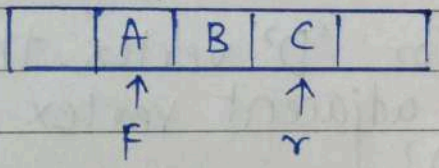


→ Insert C & marked it as visited.

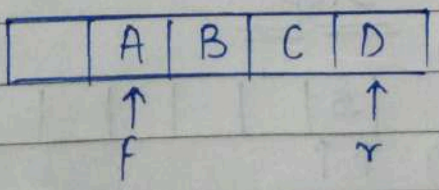


Step III: Since all the adjacents of S are visited, remove S.

Removed Node: S

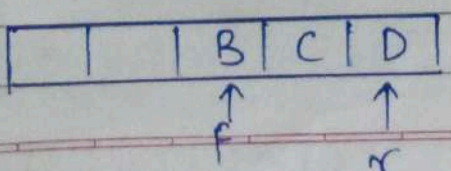


Step IV: For A, search for all adjacents, S & D D are adjacents to A (S is already visited, do not insert S into the queue, and since D is unvisited, then insert it into queue and marked it as visited).



Since all the adjacent vertices are visited, remove A.

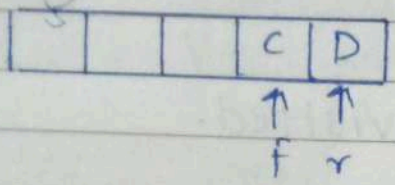
Removed Node: SA



Teacher's Signature:

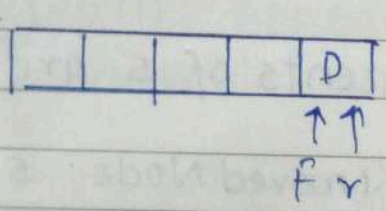
Step V: Now, for 'B' vertex all the adjacent vertex ('S' & 'D') are already visited, therefore remove B.

Removed Node: S A B

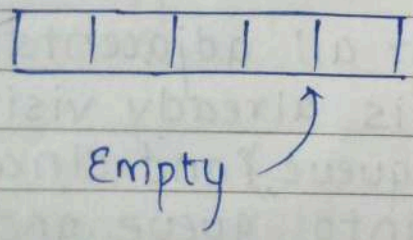


Step VI: ~~Sim~~ Now, for 'C' vertex all adjacent vertex ('S' and 'D' are ~~de~~ already visited. Therefore removed 'C'.

Removed Node: S A B C

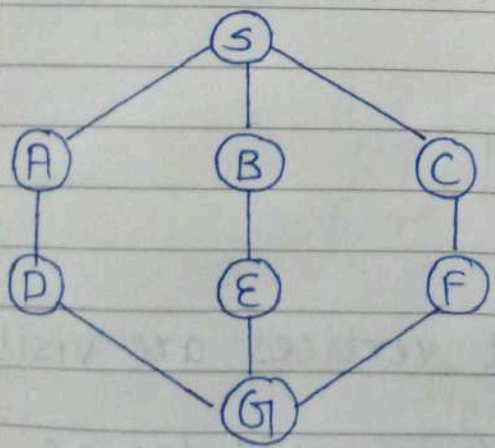


Step VII: Similarly, for 'D' vertex there is no unvisited/pending adjacent vertex is left. Therefore, remove 'D'.

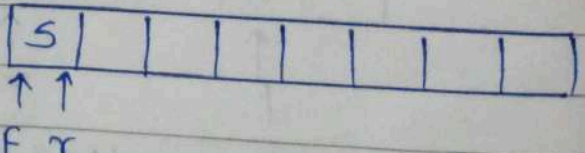


Removed Node: S A B C D.

Q.

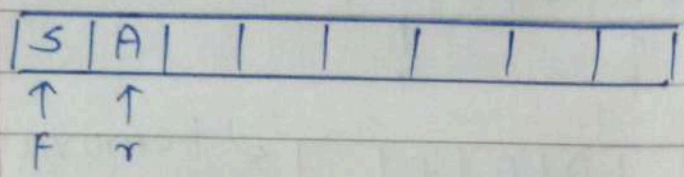


Step 1:

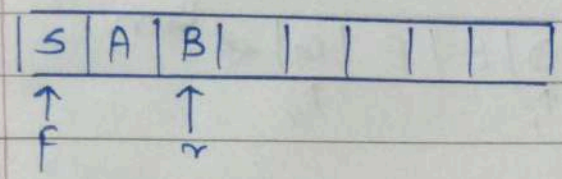


Teacher's Signature:

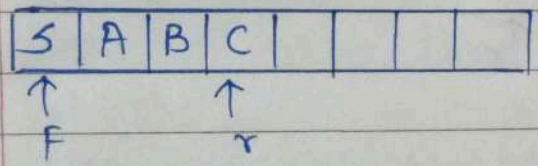
Step II:



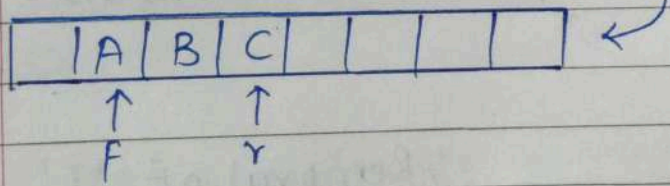
Step III:



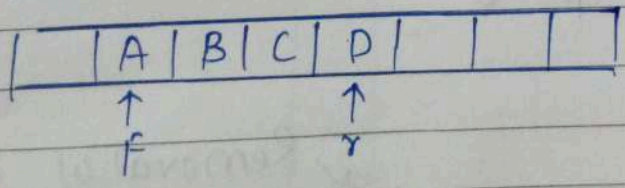
Step IV:



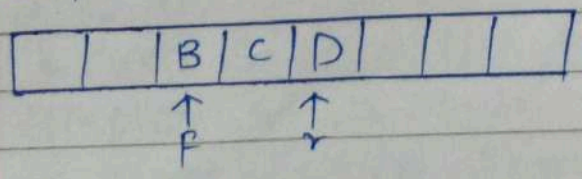
Step V:



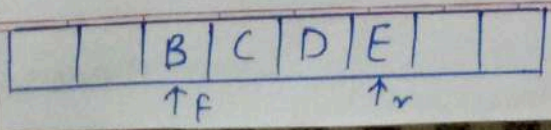
Step VI:



Step VII:

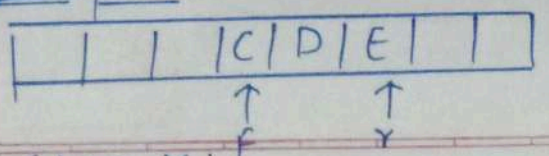


Step VIII:



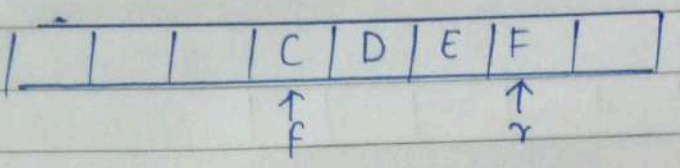
Teacher's Signature:.....

Step IX:

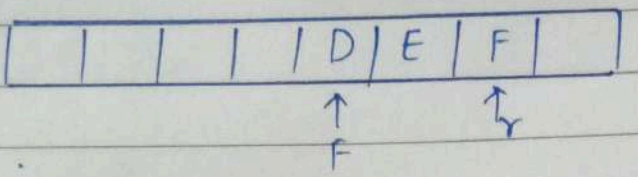


Removal of 'B'
R.N → SAB.

Step X:

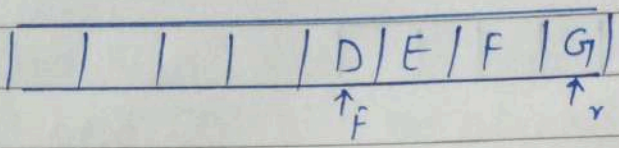


Step XI:



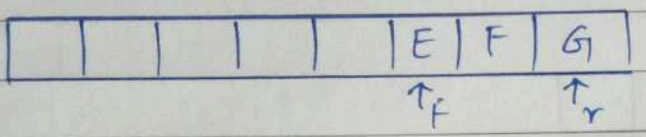
Removal of 'C'
R.N → SABC

Step XII:



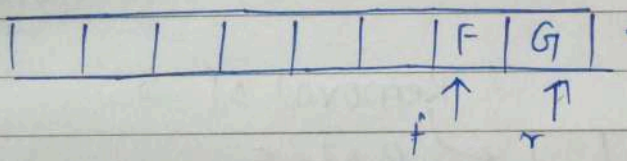
~~Rear~~

Step XIII:



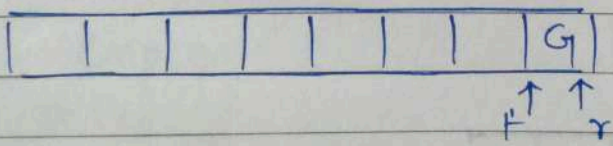
Removal of 'D'
R.N → SABCD.

Step XIV:



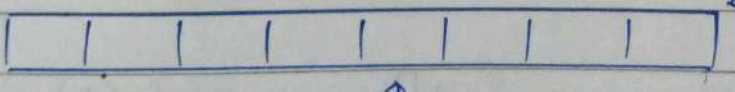
Removal of 'E'
R.N → SABCDE

Step XV:



Removal of 'F'
R.N → SABCDEF

Step XVI:



Removal of 'G'
R.N → SABCDEFG.

Empty

Teacher's Signature:.....

* DFS (Depth First Search):

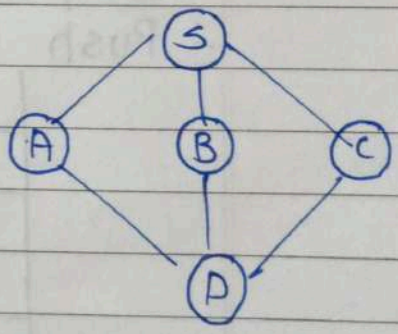
- It is the algorithm that traverses a graph in a depth-ward motion.
- It uses a stack to remember to get the next vertex to start a search when a deadend occurs in any iteration. (Back-tracking)

Step I: Visit the adjacent unvisited vertex, marked it visited, display it push it into the stack.

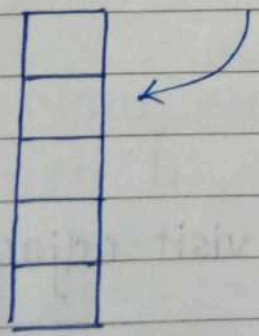
Step II: If no adjacent vertex is found popup a vertex from the stack. (Popup all the vertices ~~Step III~~ from the stack which do not adjacent vertices.

Step III: Repeat step (I) and (II) until the stack is empty.

Example:

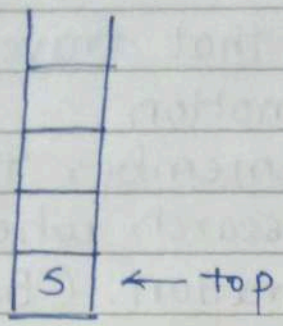


Step I: Initialize a stack.

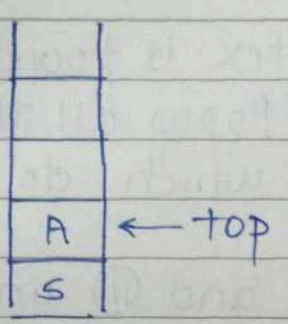


Step II:

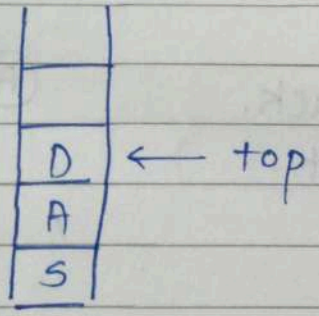
Push S into the stack, marked it visited, Display S. (S Node has 3 adjacent vertices, thus ~~it~~ visiting them in ~~of~~ alphabetical order.)



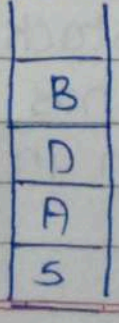
Step III: Push A into the stack, marked it visited, Display A.



Step IV:
Now, we will visit all adjacents of A.
∴ Push D as S is all ready visited.

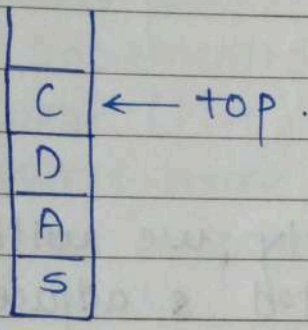


Step V: Now, we will visit adjacents of D as it is on the top.
∴ Push B (alphabetical order), marked it visited.
Display B.

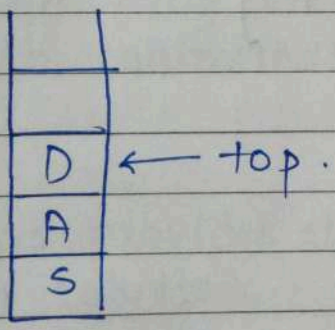


Teacher's Signature:.....

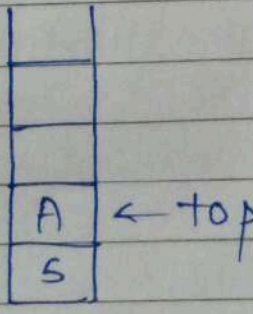
Step VI: Now, since for 'B', 's' and 'D' nodes are already visited, thus pop-out 'B'.
Now, 'D' will be on top and 'C' is unvisited (adjacent of D).
∴ Push C onto the stack, marked it visited, Display C.



Step VII: Now, there is no unvisited adjacents For 'C', thus pop-out 'C'.



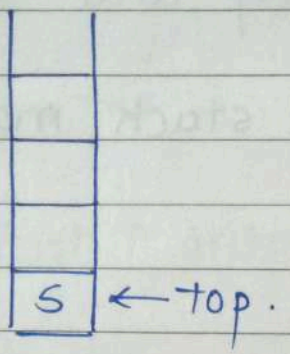
Step VIII: Now, since there is no unvisited adjacents for 'D', therefore we will pop the 'D'.



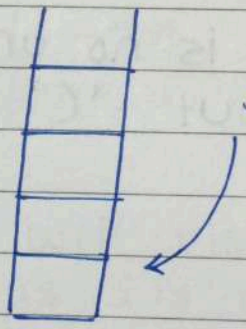
Teacher's Signature

for 'A'

Step IX: Now, since every adjacent node is visited, therefore pop 'A'.



Step X: Similarly, we will pop S for not having any unvisited adjacents remaining.



* Difference between DFS & BFS.

BFS	DFS
i. BFS stands for Breadth first search.	i. DFS stands for depth first search.
ii. It traverses the tree / graph levelwise.	ii. It traverses the graph depthwise.
iii. BFS is implemented using queue (FIFO)	iii. DFS is implemented using stack (LIFO)
iv. BFS requires more memory as compare to DFS.	iv. DFS requires less memory as compare to BFS.
v. Applications: a. To find shortest path b. Spanning tree. c. In connectivity.	v. Applications: a. Useful in cycle detection. b. In connectivity testing. c. Finding spanning tree and forest.
vi. Always provides shallowest path solution.	vi. Doesn't guarantee the shallowest path solution.
vii. No back-tracking required.	vii. Back-tracking is required.
viii. Can never get trap into infinite loop.	viii. Generally gets traps into infinite loops.

17/10

PAGE No.
DATE / /

* Shortest path Algorithm:

□. In a weighted (edges with weighted) to find the shortest path between nodes. there are 3 Algorithm:

1. Dijkstra's shortest Path.
2. Minimum spanning tree Algorithm:
   a. Prim's
   b. kruskal

1. Dijstra's shortest path: shortest path

□. It's the single source algorithm for directed graph.

□. It finds the shortest path between a particular node called 'source node' (initial node). and any other node in connected graph.

□. To select the source node, search for the node whose incoming edges are less and outgoing ~~and~~ edges are maximum.

→ Steps →

step 1: Select the source node.
step 2: Define an empty set 'N'. that will be use to hold nodes to which a shortest path has been foond
Step 3: Lebel the initial node with value 0 and insert it into 'N'.
step 4: Repeat step 5 to 7 until the destination node is in an or there are no more labelled node in
step 5: Consider each node i.e., not in an and it is connected by an edge for a newly inserted node.

Teacher's Signature:.....

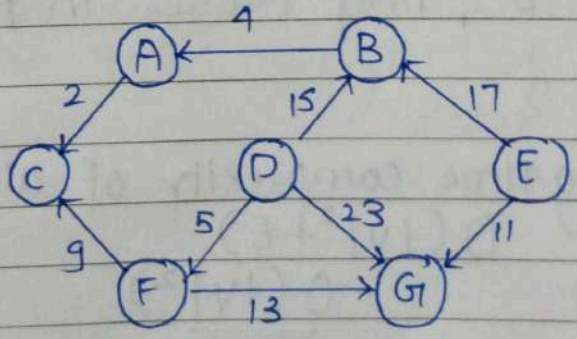
Step 6: a) If the node, i.e., not in 'N' and has no lebel, then set the lebel of the node is equal to lable of newly inserted node + the length of the edge.

b) Else if the node that's not in 'N' was already labelled then set its new label = minimom of (label of newly inserted vertex + legth of edge, old label)

Step 7: Pick a node, not in 'N' that has smallest label assign to it and add it to 'N'.

The execution of this algorithm produce either of the following 2 results:

1. If the destination is labelled then the label inturn represents the distance from source node to destination node.
2. If destination node is not labelled then there is no path from destination.



Step 1: Select 'D' as source node, since it has maximum outgoing edges.
Set the lebel of "D" = 0. & N = {D}

Teacher's Signature:

minimum.

Step 2: Label of D=0, B=15, G=23 & F=5.
∴ N = {D, F}.

D→F F→G

Step 3: Label of D=0, B=15, & G = minimum (5+13, 23) = 18 & C = 5+9 = 14, 1

D→F F→C

∴ N = {D, F, C}

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

D→B B→A

Step 5: D=0, G=18, A = 15+4 = 19
∴ N = {D, F, C, B, G}

Step 6: D=0, A=19,

∴ N = {D, F, C, B, G, A}

We've no label for E i.e., 'E' is not reachable from 'D', only Nodes in 'A' are reachable from 'D'.

Running time complexity of algorithm is:
$O(|V|^2 + E)$
$= O(|V|^2)$

Teacher's Signature:

18/10

PAGE No.
DATE / /

* Spanning tree:

□. A Spanning tree of a connected undirected graph 'G' is a subgraph of 'G' which is a tree, that connects all the vertices together.

□. A graph 'G' can have many different spanning tree.

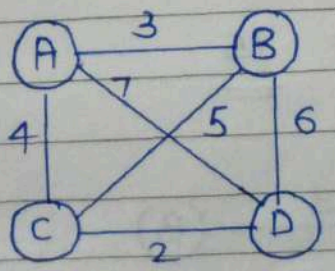
□. We can assign weights to each edge and use it to assign a weight to a spanning tree by calculating the sum of the edges in that tree.

□. A spanning tree will never have a cycle.

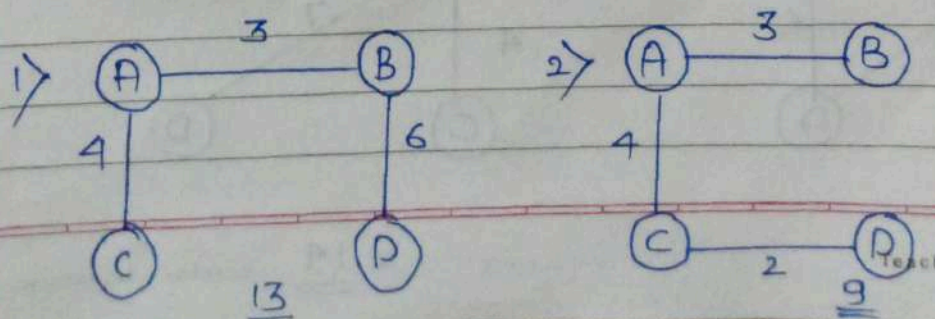
* Minimum Spanning tree:

□. It is define as a spanning tree with weight less than or equal to the weight of every other spanning tree. i.e., it is the tree that has weights associated with it's edges and the total weight of the tree is at minimum.

E.g →

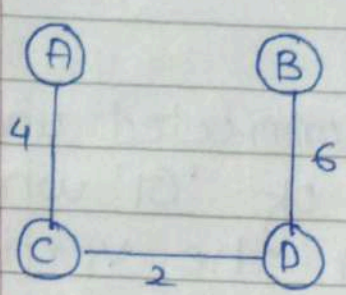


→



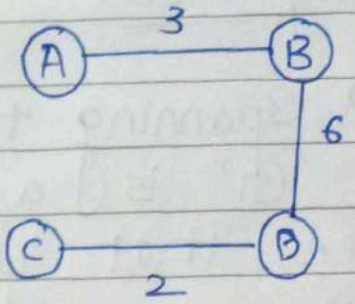
Teacher's Signature:

3)



= 12

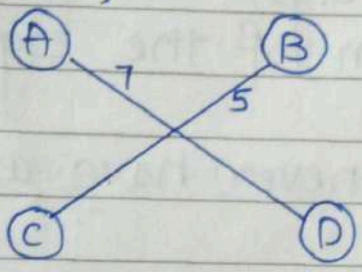
4)



= 11

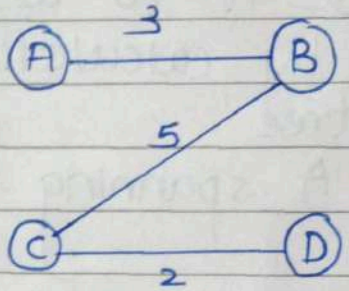
wrong

5)



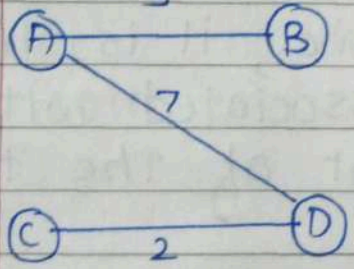
= 12

6)



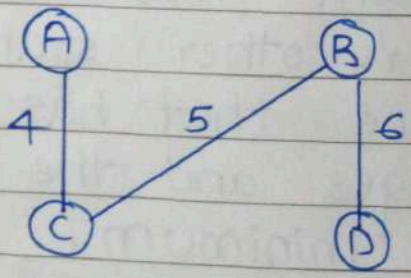
= 10

7)



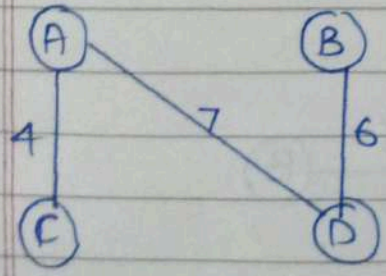
= 12

8)



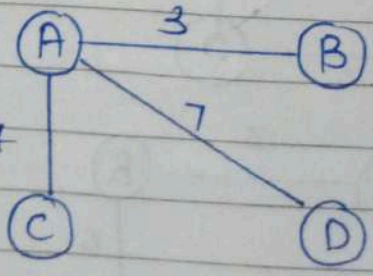
= 15

9)



= 17

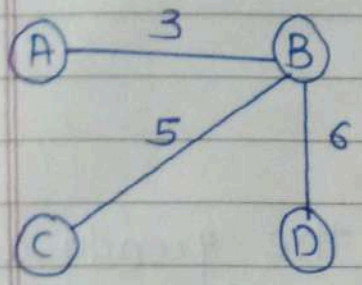
10)



= 14

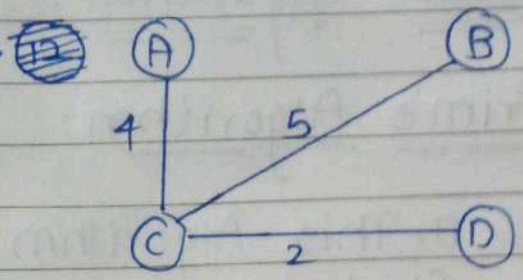
Teacher's Signature:

11)



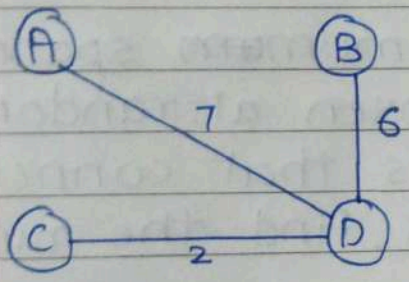
= 14

12)



= 11

13)



= 15

* ~~Prince Algorithm~~ :

* Prim's Algorithm :

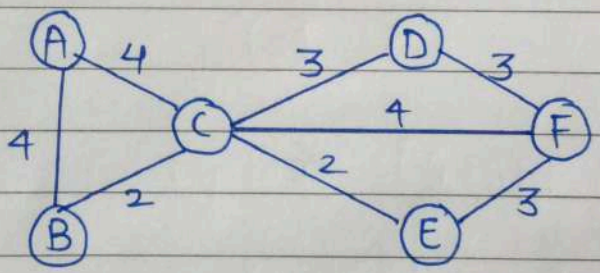
□. This Algorithm follows the gready approach to find the minimum spanning ~~tree~~ tree, i.e. it starts with find current & best solution.

→ Steps →

Step1: Initialize the minimum~~um~~ spanning tree with the vertex choosen at random.

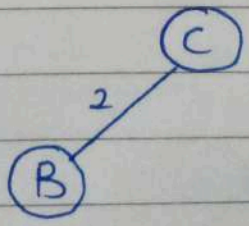
Step2: Find all the edges that connect the tree to new vertices, Find the minimum and add it to the tree.

Step 3: Keep repeating step (2) until a minimum spanning tree is formed.



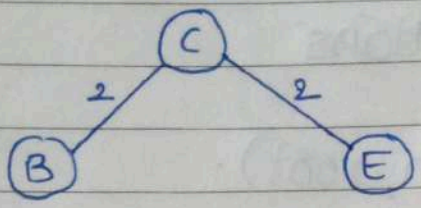
Step 1: Select 'C' vertex ~~ran~~ randomly to form the spanning tree.

Step II:

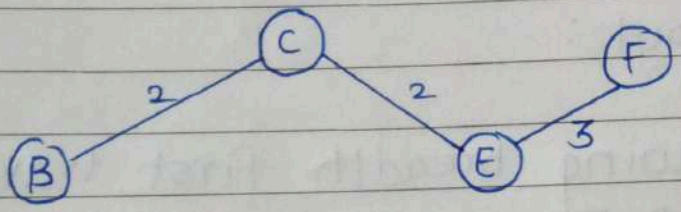


Teacher's Signature:

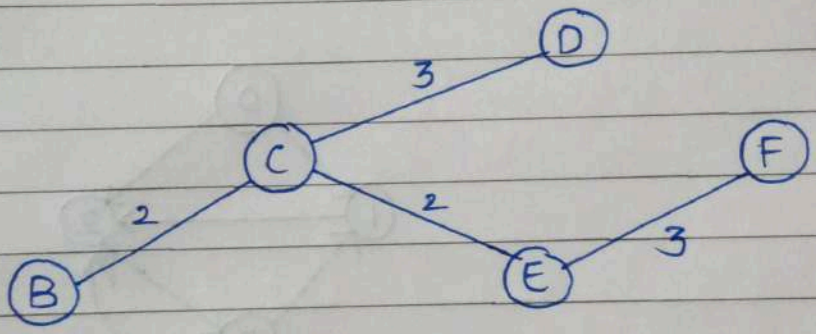
Step III:



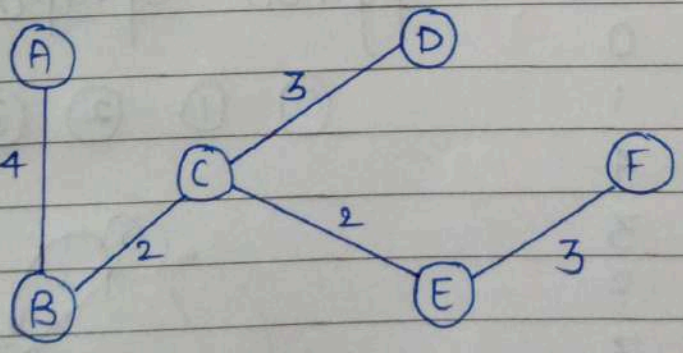
Step IV:



Step V:



Step VI:



Total weight = 14

19/10

PAGE NO.
DATE / /

* Kruskal Algorithm

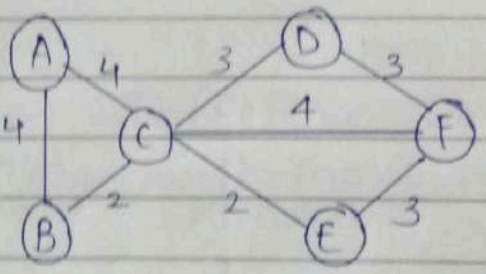
1. This is a greedy approach to find the minimum spanning tree.
2. It starts from the edges with lowest weight and keep added edges until minimum spanning tree form.

→ Steps →

Step I: ~~Ser~~ Sort all the edges from low weight to height high.

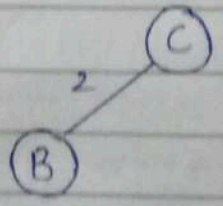
Step II: Take the edge, with lowest weight and add it to minimum spanning tree.
If the adding edge creating a cycle, the reject that edge.

Step III: keep adding edges untill a minimum spanning tree is form.



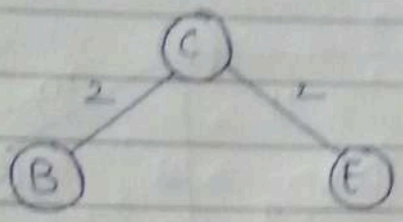
Step I: BC → 2 ; CE → 2 ; CD → 3
DF → 3 ; EF → 3 ; AB → 4
AC → 4 ; CF → 4 .

Step II:

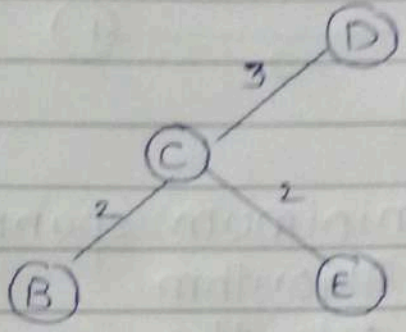


Teacher's Signature:

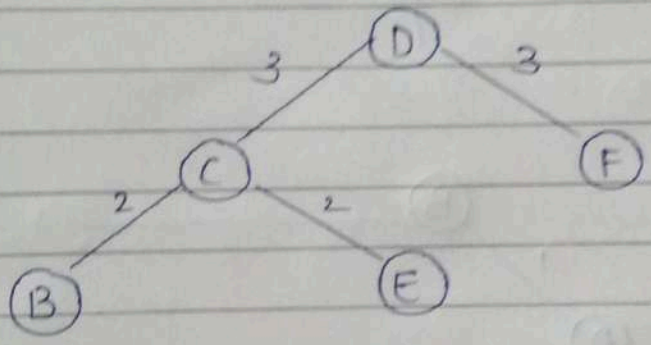
Step III:



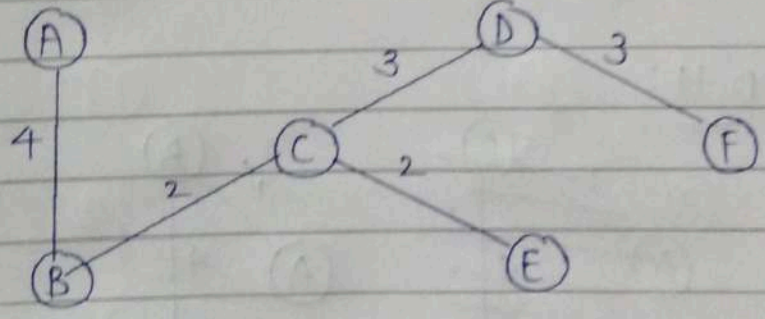
Step IV:



Step V:

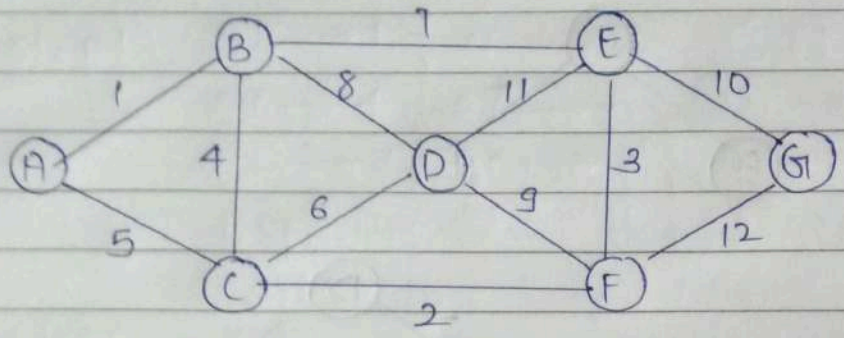


Step VI:



total weight: 14

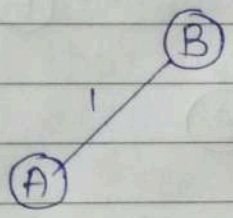
Teacher's Signature:



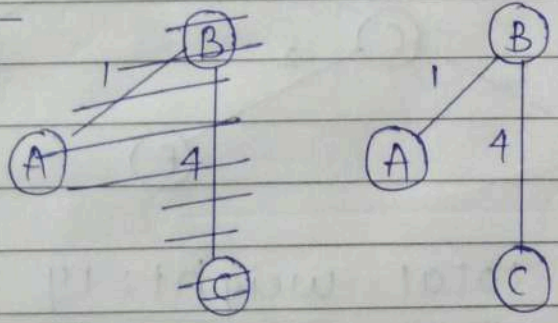
8. Find the minimum spanning tree using
A) Prim's Algorithm.
B) kruskal Algorithm.

Prim's Algo

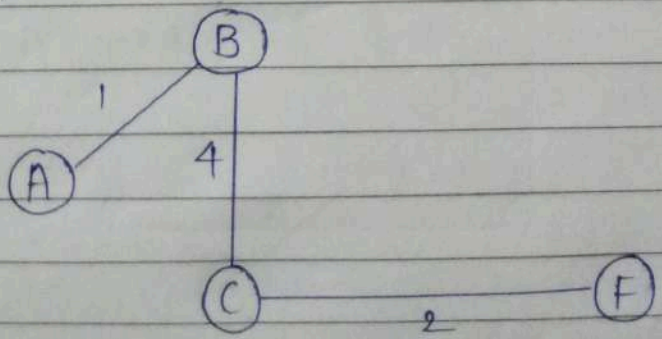
A) → step I :



step II :

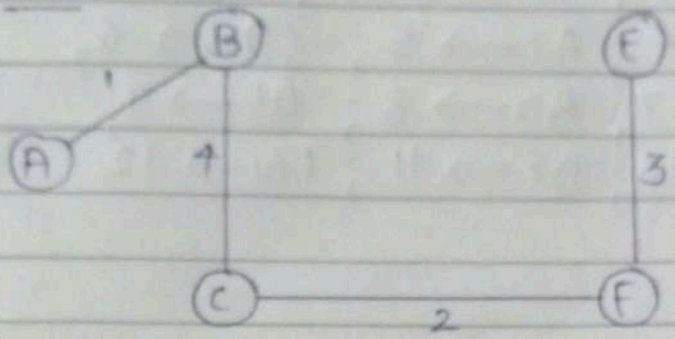


step III :

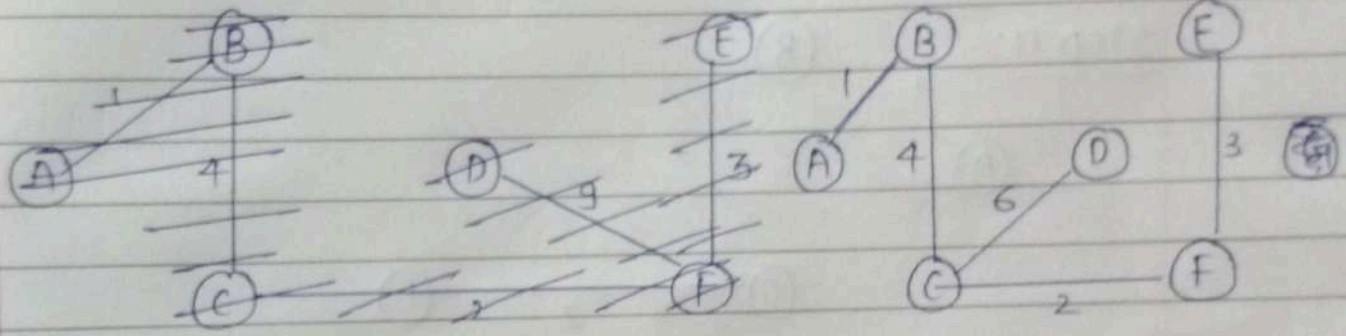


Teacher's Signature:

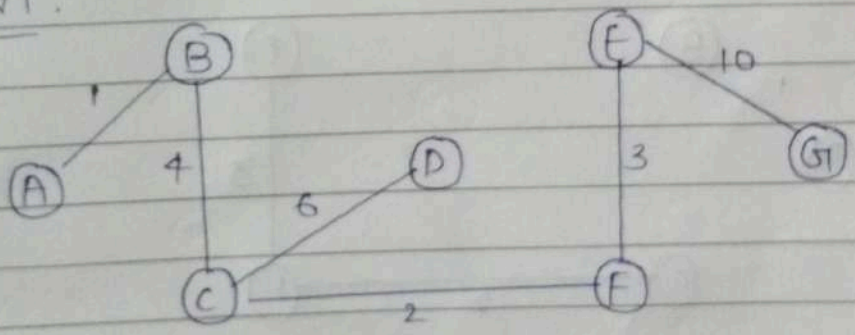
Step IV:



Step V:



Step VI:



total weight: 1+4+6+2+3+10

= 26

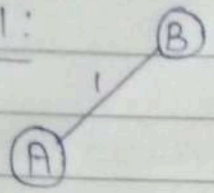
Teacher's Signature:.....

Kruskal Algo:

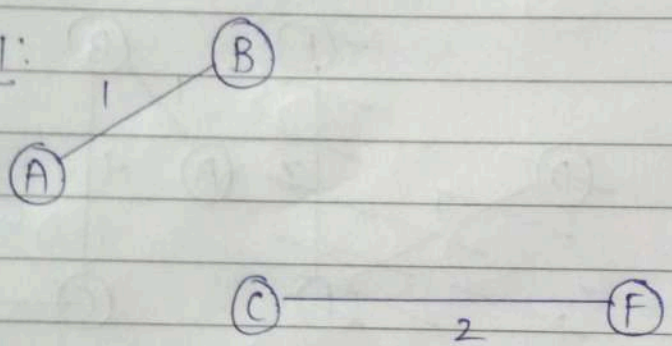
B)

Step I: AB → 1 ; CF → 2 ; FF → 3 ;
BC → 4 ; AC → 5 ; CD → 6 ;
BE → 7 ; BD → 8 ; DF → 9 ;
EG → 10 ; DE → 11 ; FG → 12

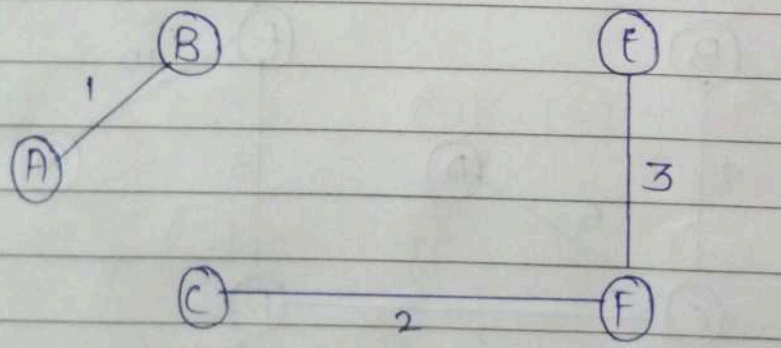
Step I:



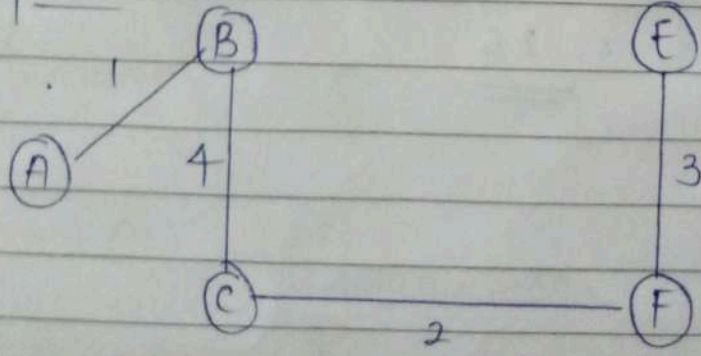
Step II:



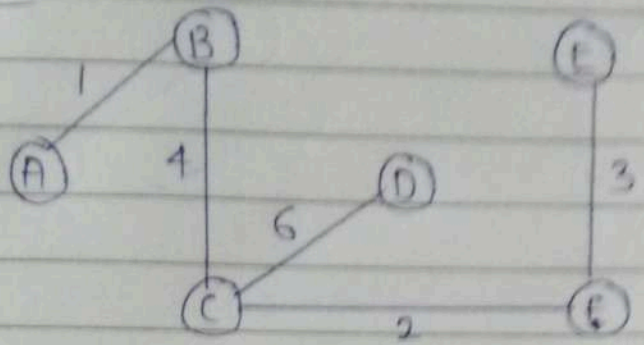
Step III:



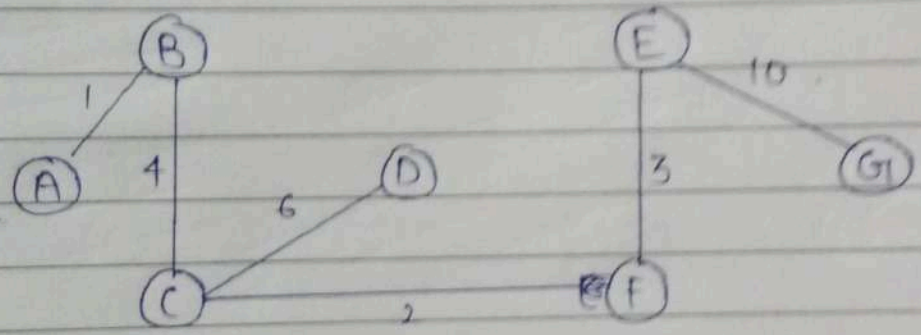
Step IV:



Step V:



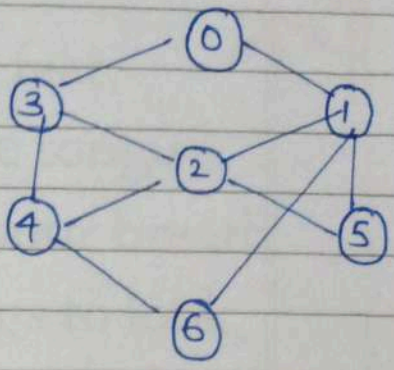
Step VI:



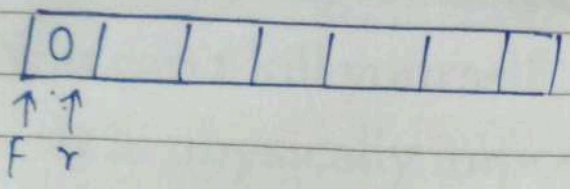
total weight: 1 + 4 + 6 + 2 + 3 + 10.

= 26.

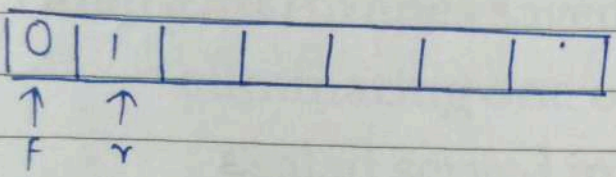
Q. H.W.



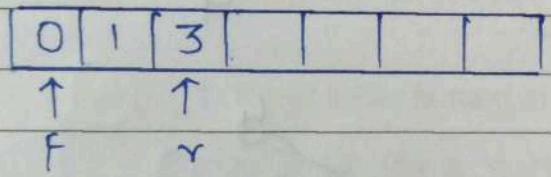
→ Step I:



Step II:

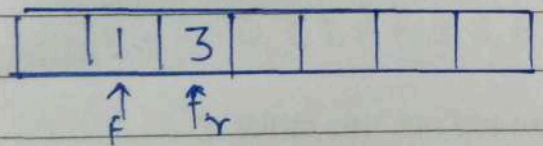


Step III:

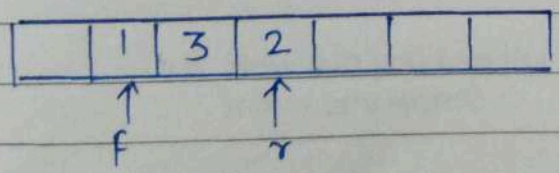


Step IV: Remove 0.

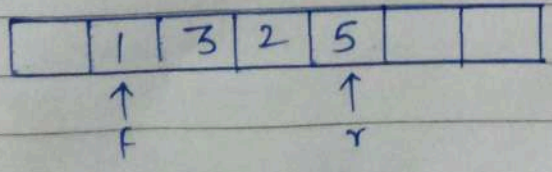
Removed Node: 0



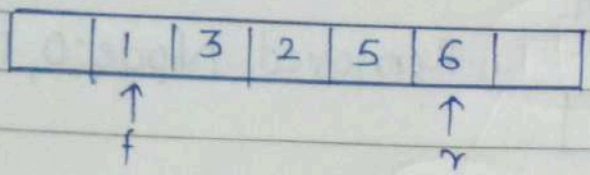
Step V:



Step VI:

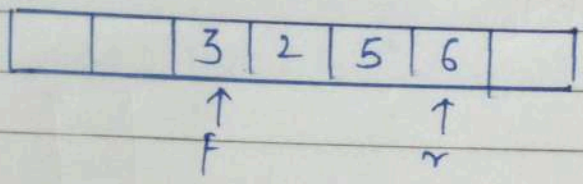


Step VII :

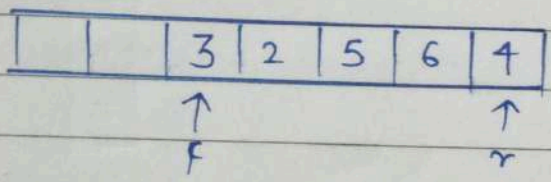


Step VIII : Remove 1.

Removed Node : 0, 1.

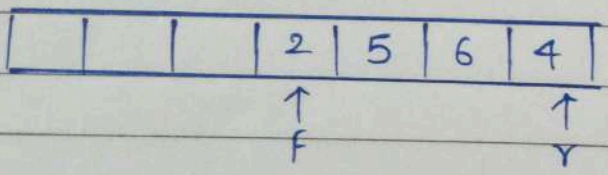


Step IX :



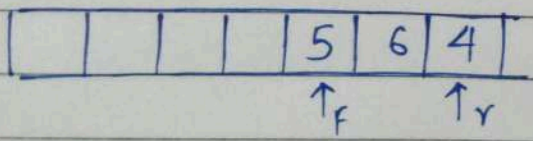
Step X : Remove 3.

Removed Node: 0, 1, 3.



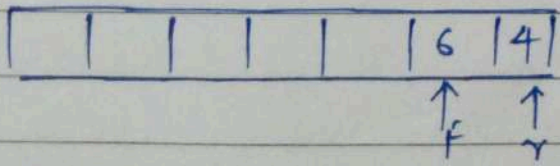
Step XI : Remove 2

Removed Node : 0, 1, 3, 2



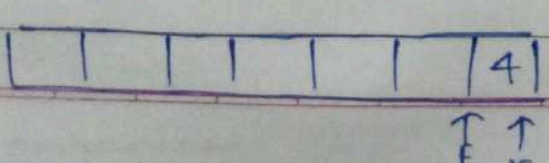
Step XII : Remove 5

Removed Node : 0, 1, 3, 2, 5



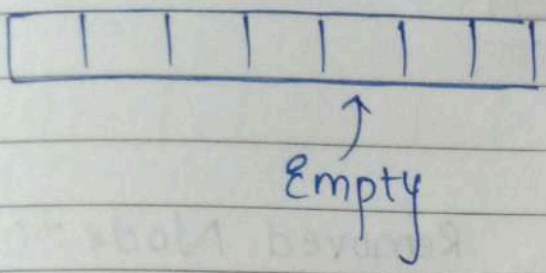
Step XIII : Remove 6

Removed Node : 0, 1, 3, 2, 5, 6.



Teacher's Signature:.....

Step XIV: Remove 4



Removed Node: 0, 1, 3, 2, 5, 6, 4.