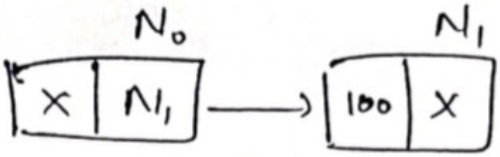
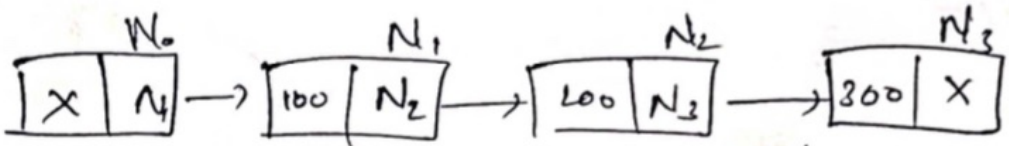
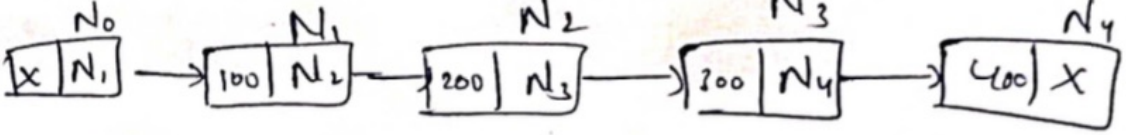
iii) 

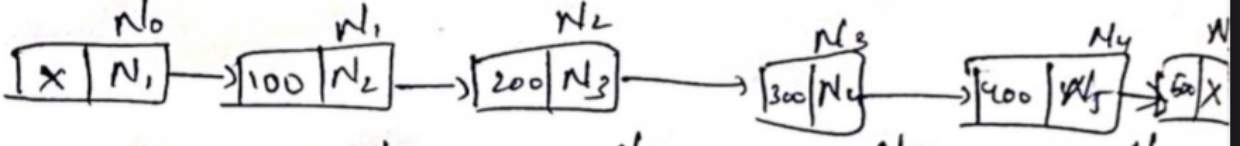
UNIT - 3

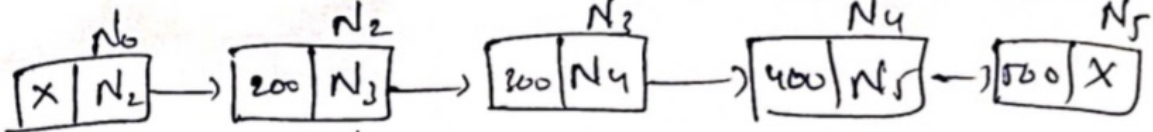
1) i) 

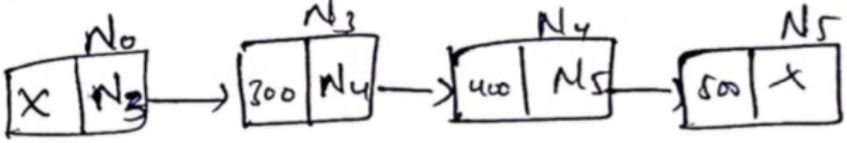
ii) 

iii) 

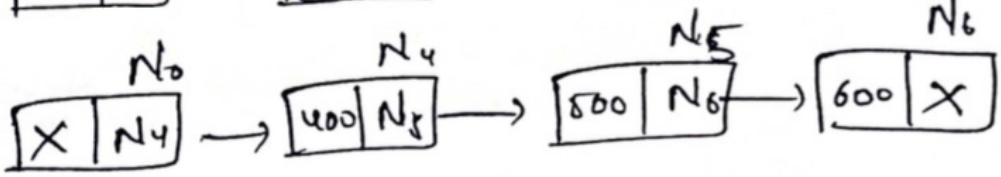
iv) 

v) 

vi) 

vii) 

viii) 

ix) 

The elements of the queue are 400, 500, 600

Unit - III

Stacks and Queues

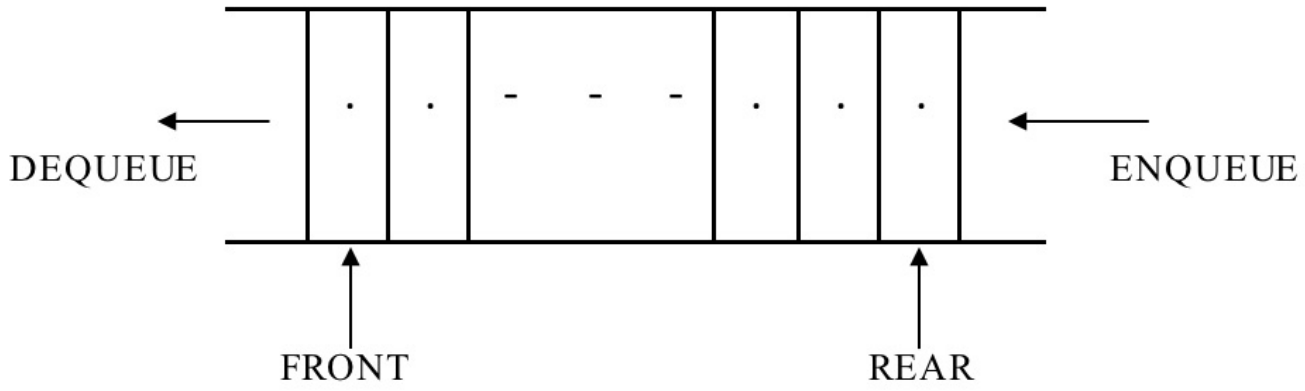
Learning Material

QUEUES

Queue is a linear Data structure.

Definition: Queue is a collection of homogeneous data elements, where insertion and deletion operations are performed at *two extreme ends*.

- The insertion operation in Queue is termed as *ENQUEUE*.
- The deletion operation in Queue is termed as *DEQUEUE*.
- An element present in queue is termed as *ITEM*.
- The number of elements that a queue can accommodate is termed as *LENGTH* of the Queue.
- In the Queue the *ENQUEUE* (insertion) operation is performed at ***REAR*** end and *DEQUEUE* (deletion) operation is performed at ***FRONT*** end.
- Queue follows ***FIFO*** principle. i.e. First In First Out principle. i.e. a item First inserted into Queue, that item only First deleted from Queue, so queue follows FIFO principle.



Schematic Representation of Queue

Algorithm Enqueue(item)

Input: *item* is new item insert in to queue at rear end.

Output: Insertion of new item queue at rear end if queue is not full.

1. if(rear = = N-1)
    a) print "queue is full, not possible for enqueue operation"
2. else
    a) if(front = = -1 && rear = = -1) /* Queue is Empty */
        i) rear = rear+1
        ii) Q[rear] = item
        iii) front = 0
    b) else
        i) rear = rear+1
        ii) Q[rear] = item

    c) end if
3. end if

End Enqueue

While performing ENQUEUE operation two situations are occur.

1. if queue is empty, then newly inserting element becomes first element and last element in the queue. So Front and Rear points to first element in the list.
2. If Queue is not empty, then newly inserting element is inserted at Rear end.

Algorithm Dequeue( )

Input: Queue with some elements.

Output: Element is deleted from queue at front end if queue is not empty.

1. if(front = = -1 && rear = = -1)
    a) print "Q is empty, not possible for dequeue operation"
2. else
    a) if(front = = rear) /* Q has only one element */
        i) item = Q[front]
        ii) front = -1
        iii) rear = -1
    b) else
        i) item = Q[front]
        ii) front = front+1
    c) end if
    d) print "deleted item is" item
3. end if

End Dequeue

Algorithm Enqueue _LL(item)

Input: item is new item to be insert.

Output: new item i.e new node is inserted at rear end.

1. new = getnewnode()
2. if(new = = NULL)
    a) print "required node is not available in memory"
3. else
    a) if(front = = NULL && rear = = NULL) /* Q is EMPTY */
        i) header→link = new
        ii) new→link = NULL
        iii) front = new
        iv) rear = new
        v) new→data = item
    b) else /* Q is not EMPTY */
        i) rear→link = new /* 1 */
        ii) new→link = NULL /* 2 */
        iii) rear = new /* 3 */
        iv) new→data = item
    c) end if
4. end if

End_Enqueue_LL

Algorithm Dequeue_LL( )

Input: Queue with some elements

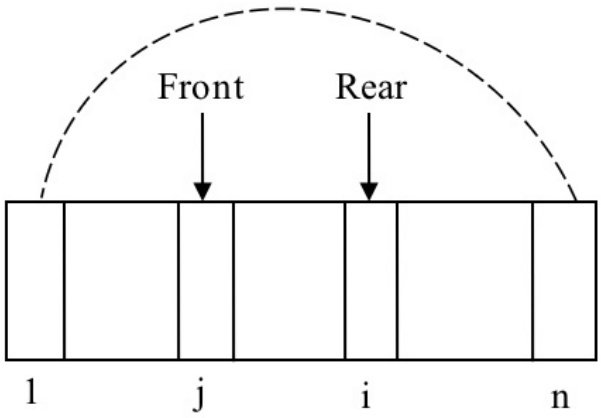
Output: Element is deleted at front end, if queue is not empty.

1. if(front = = NULL && rear = = NULL)
    a) print "queue is empty, not possible to perform dequeue operation"
2. else
    a) if(front = = rear) /* Q has only one element */
        i) header→link = NULL
        ii) item = front→data
        iii) front = NULL
        iv) rear = NULL
    b) else /* Q has more than one element */
        i) header→link = front→link /* 1 */
        ii) item = front→data
        iii) free(front)
        iv) front = header→link /* 2 */
    c) end if
    d) print "deleted element is item"
3. end if

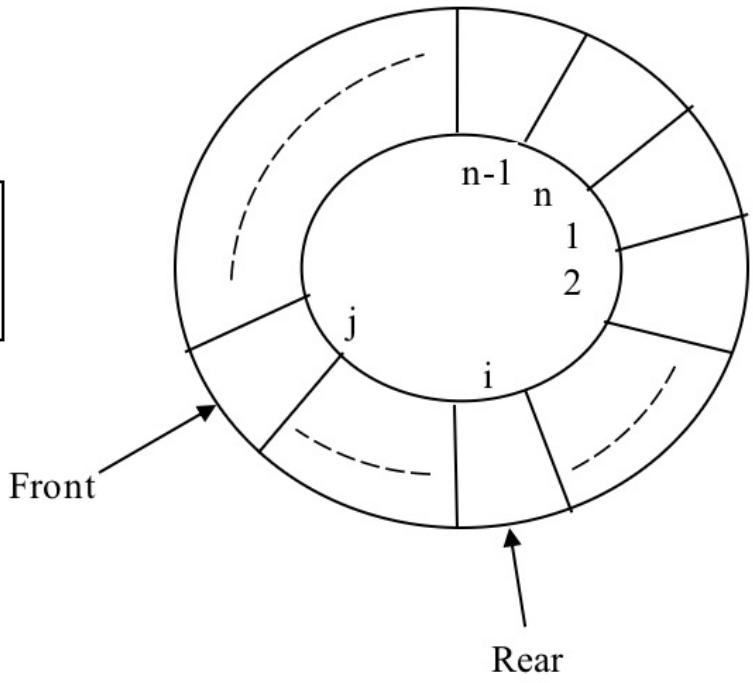
End_Dequeue_LL

1. Circular Queues

- Physically a circular array is same as ordinary array, say a[1-N], but logically it implements that a[1] comes after a[N] or a[N] comes after a[1].
- The following figure shows the physical and logical representation for circular array.



Circular Array (Physical)



Circular Queue (Logical)

Logical and physical view of a Circular Queue

- Here both *Front* and *Rear* pointers are move in clockwise direction. This is controlled by the ***MOD*** operation.
- For e.g. if the current pointer is at i, then shift next location will be

  (i mod LENTH) +1, 1<= i <= Length

Circular Queue empty condition is

Front = = 0 && Rear = = 0

Circular Queue is full

Front = = (Rear % Length) + 1

Algorithm CQ_Enqueue(item)

Input: item is new element insert in to Circular queue at rear end.

Output: Insertion of new item in Circular queue at rear end if queue is not full.

1. next = (REAR % N) + 1
2. if(FRONT == next)
    a) print("Circular queue is full, enqueue operation is not possible")
3. else
    i) if(FRONT==0 and REAR==0) /* CQ is Empty */
        a) REAR=(REAR % N) +1
        b) CQ[REAR]=item
        c) FRONT=1
    ii) else
        a) REAR=(REAR% N) + 1
        b) CQ[REAR]=item
    iii) end if
4. endif

End CQ_Enqueue

Algorithm CQ_Dequeue( )

Input: Circular Queue with some elements.

Output: Element is deleted from circular queue at front end if circular queue is not empty.

1. if(FRONT= =0 and REAR= =0)
    a) print("Circular Queue is empty, dequeue operation is not possible")
2. else
    i) if(FRONT = =REAR) /* Q has only one element */
        a) item=CQ[FRONT]
        b) FRONT=0
        c) REAR=0
    ii) else
        a) item=CQ[FRONT]
        b) FRONT=(FRONT % N)+1
    iii) End if
    iv) print("deleted item is 'item' ")
3. end if

End CQ_Dequeue

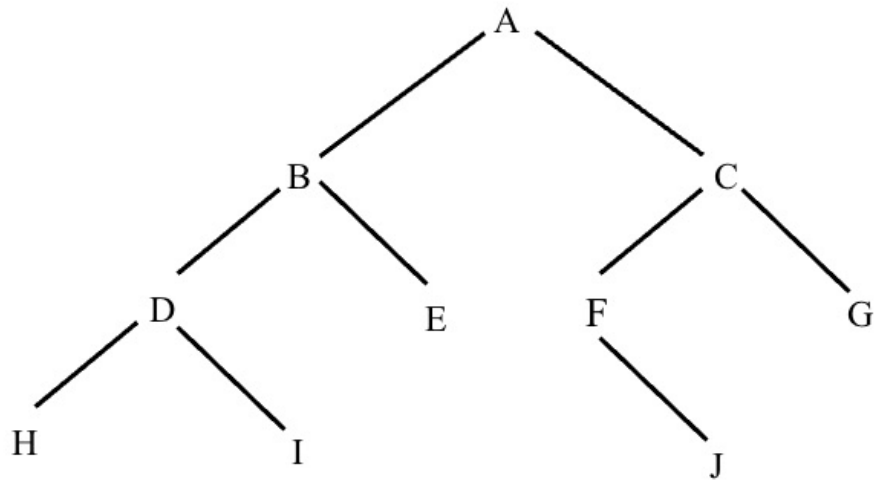
BINARY TREE: Is a special form of a tree.

Definition:

A binary tree is T is a finite set of nodes such that,

(i) T is empty called as empty binary tree.

(ii) T has specially designed node called as Root Node and remaining node of binary tree are partitioned into 2 disjoint sets. One is Left sub tree and another one is Right sub tree.



A sample Binary Tree

1. Linear (or) Sequential representation using arrays

- In this representation, a block of memory for an array is to be allocated before going to store the actual tree in it.
- Once the memory is allocated, the size of the tree will be restricted to memory allocated.
- In this representation, the nodes are stored level by level starting from zero level, where only *ROOT* node is present.
- The *ROOT* node is stored in the first memory location. i.e. first element in the array.

The following rules are used to decide the location on any node of tree in the array. (Assume the array index start from 1)

1. The ROOT node is at index**1**.
2. For any node with index i, $1 \leq i \leq n$.

   (a) **Parent (i)** = $\left\lfloor \frac{i}{2} \right\rfloor$

   For the node when i = 1, there is ***no parent node***.

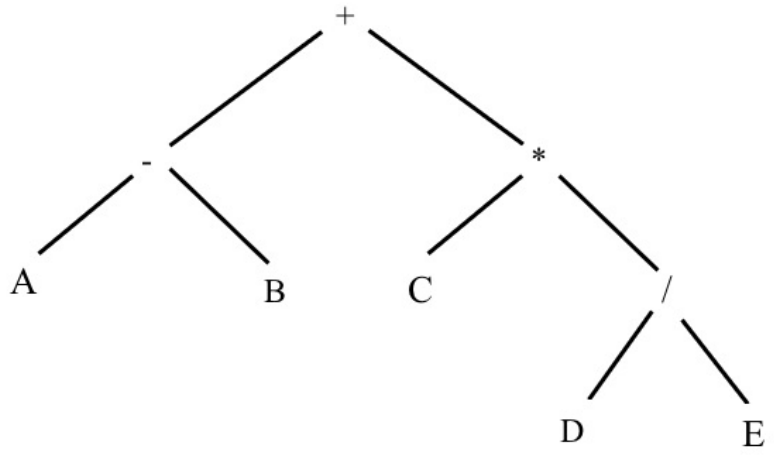
   (b) **LCHILD (i)** = 2 * i

   If 2 * i > n then **i** has ***no left child***.

   (c) **RCHILD (i)** = 2*i +1

   If 2*i + 1 > n then **i** has ***no right child***.

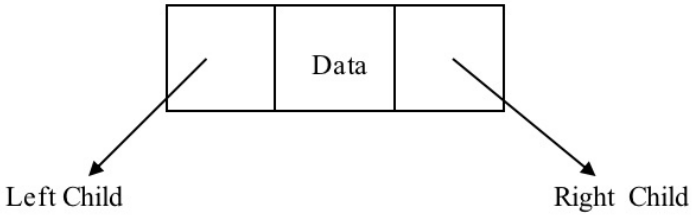
Eg. (A-B) + C * (D / E)



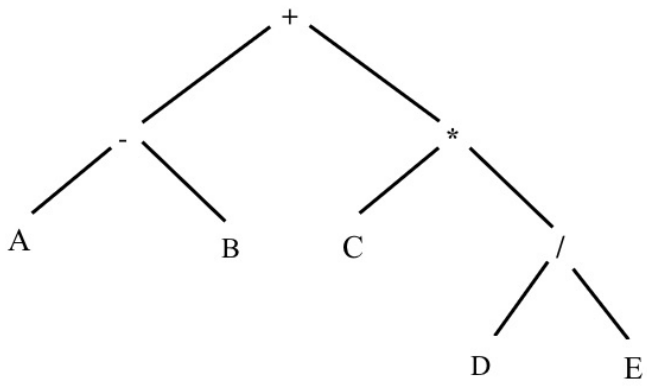
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	-	*	A	B	C	/							D	E

2. Linked List representation of Binary Tree using pointers

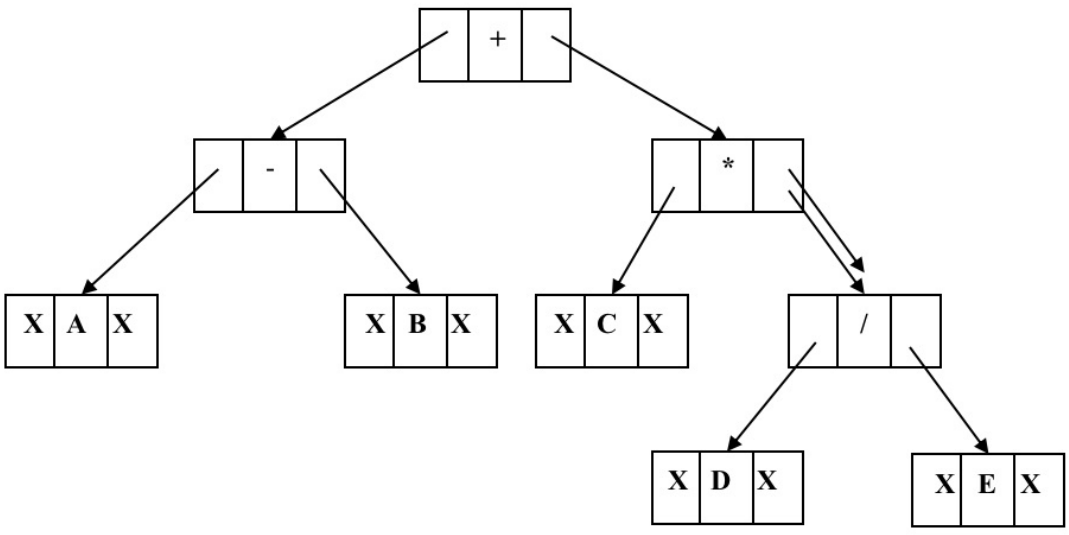
- Linked list representation of assumes structure of a node as shown in the following figure.



- With linked list representation, if one knows the address of *ROOT* node, then any other node can be accessed.



Binary Tree



Linked List representation of Binary Tree

Recursive implementation of Binary Tree Traversals:

Algorithm preorder(ptr)

Input: Binary Tree with some nodes.

Output: *preorder* traversal of given Binary Tree.

1. if(ptr != NULL)
    a) print(ptr.data)
    b) preorder(ptr.lchild)
    c) preorder(ptr.rchild)
2. end if

End preorder

Algorithm inorder(ptr)

Input: Binary Tree with some nodes.

Output: *inorder* traversal of given Binary Tree.

1. if(ptr != NULL)
    a) inorder(ptr.lchild)
    b) print(ptr.data)
    c) inorder(ptr.rchild)
2. end if

End inorder

Algorithm postorder(ptr)

Input: Binary Tree with some nodes.

Output: *postorder* traversal of given Binary Tree.

1. if(ptr != NULL)
    a) postorder(ptr.lchild)
    b) postorder(ptr.rchild)
    c) print(ptr.data)
2. end if

End postorder

Non recursive implementation of Binary Tree Traversals

To implement non recursive implementation of binary tree traversals, here we are using single array for Inorder and Preordere traversals and two stacks for Postorder traversals.

Algorithm Inorder_Stack(i)

Input: *i* is root node **index** and a stack is used.

Output: Inorder traversal of binary tree.

1. while(stack is not empty || a[i] != NULL)
    a) if(a[i] !=NULL)
        i) push(i)
        ii) I = 2*i
    b) else
        i) i = pop()
        ii) print(a[i])
        iii) I = 2*i+1
    c) end if
2. end loop

End Inorder_Stack

The following operations are performed to traverse a binary tree in in-order using a stack:

1. Start from the root, call it PTR.
2. Push PTR onto stack if PTR is not NULL.
3. Move to left of PTR and repeat step 2.
4. If PTR is NULL and stack is not empty, then Pop element from stack and set as PTR.
5. Process PTR and move to right of PTR , go to step 2.

Algorithm Preorder_Stack(i)

Input: *i* is root node **index** and a stack used.

Output: Preorder traversal of binary tree

1. push(i)

---

2. while(stack is not empty)
    a) i = pop()
    b) if(a[i] != NULL)
        i) print(a[i])
        ii) push(2*i+1)
        iii) push(2*i)
    c) end if
3. end loop

End Preorder Stack

Algorithm Postorder_Stack(ROOT)

Input: *ROOT* is root node and two stacks named as stack1 and stack2 are used.

Output: Postorder traversal of binary tree.

1. PUSH the ROOT node into stack1.
2. While the stack1 is not empty
    a) POP a node from stack1 and PUSH it into the stack2.
    b) PUSH the LEFT and RIGHT Childs of poped nodes into stack1.
3. End loop
4. Now print the content of stack2. i.e. POP every node from stack2 and print them.

End Postorder_Stack

1. Searching for data a Binary Search Tree

- Searching data in a binary search tree is much faster than searching data in arrays or linked lists.
- To find an element ITEM from the BST first it should be compared with the root node R, and then one of the following conditions may be true:
    a) ITEM == R
    b) ITEM < R
    c) ITEM > R
- If ITEM == R, the required element is found at Root, and the search is successful.
- If ITEM < R means that the required element is less than Root , so ITEM may reside in the left sub tree of the root, therefore we continue the search in the left sub tree of the root.
- If ITEM > R means that the required element is greater than Root, so ITEM may reside in the right sub tree of the root, therefore we continue the search in the right sub tree of the root.

Algorithm BST_Search(item)

Input: item is data part of new node to be search in BST.

Output: if found then pointer to node containing data part as item. Otherwise error message.

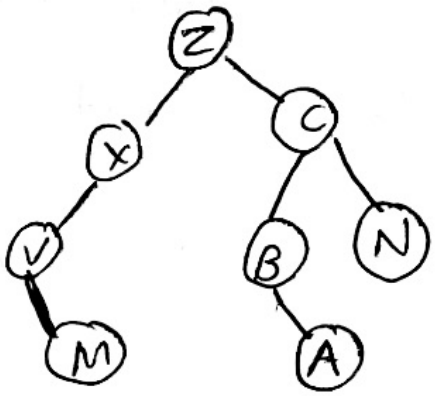
1. ptr = Root
2. flag = 0

---

3. while(ptr != NULL and flag == 0)
    a) if(item == ptr.data)
        i) flag = 1
        ii) print “item found at ptr”
    b) else if(item <ptr.data)
        i) ptr = ptr.LCHILD goto step 3
    c) else if(item >ptr.data)
        i) ptr = ptr.RCHILD goto step 3
    d) end if
4. end loop
5. if(flag == 0)
    a) print “item not found”
6. end if

End BST_Search

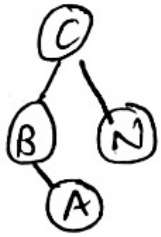
3. b)



Inorder:

LS R RS

Z



V M X Z B A C N

VMX Z BACN

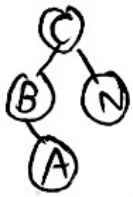
Preorder:

R LS RS

Z X V M C B A N

Postorder:

LS RS R



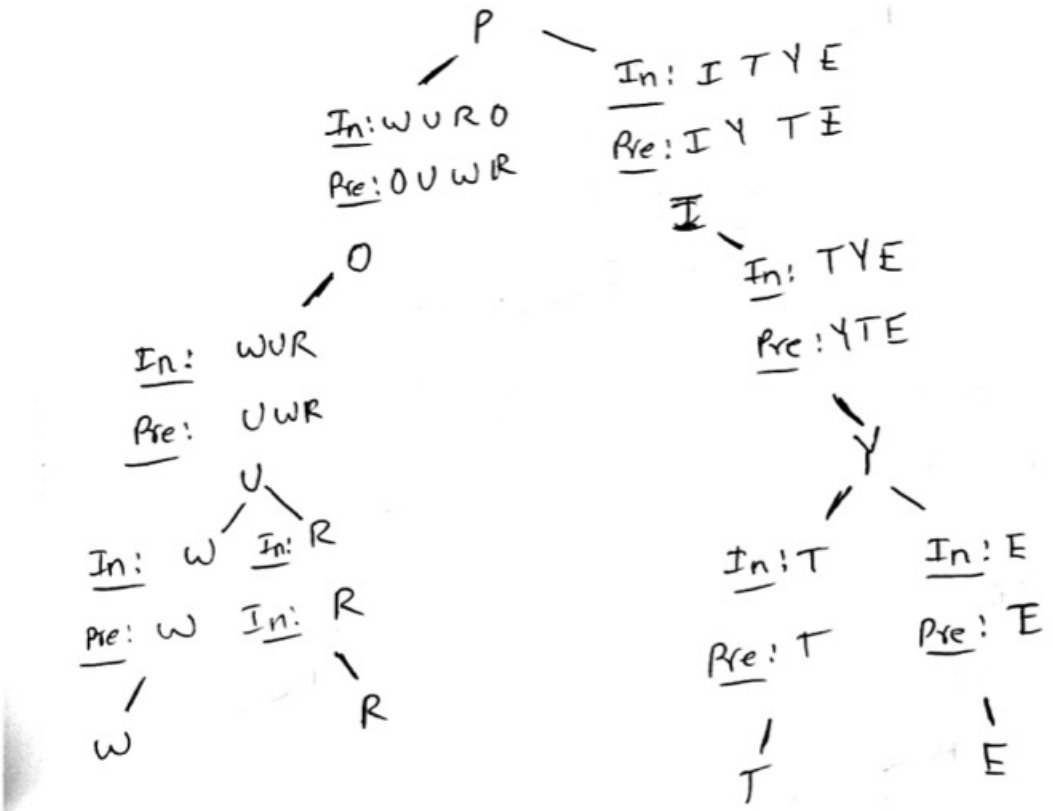
Z

V M X B A N C Z

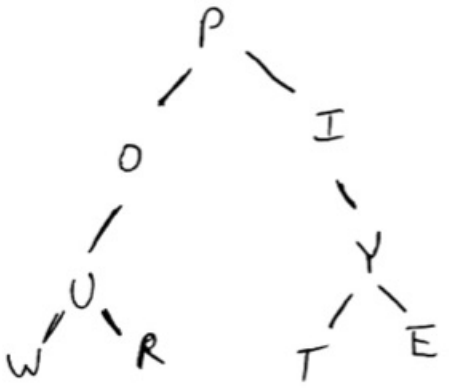
MVX ABNC Z

4: In: W U R O P I T Y E

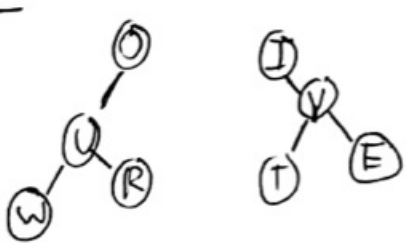
Pre: P O U W R I Y T E



Final Binary Tree



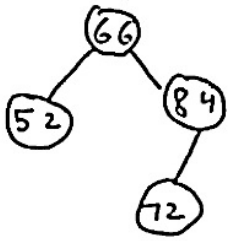
Postorder: LS RS R



P

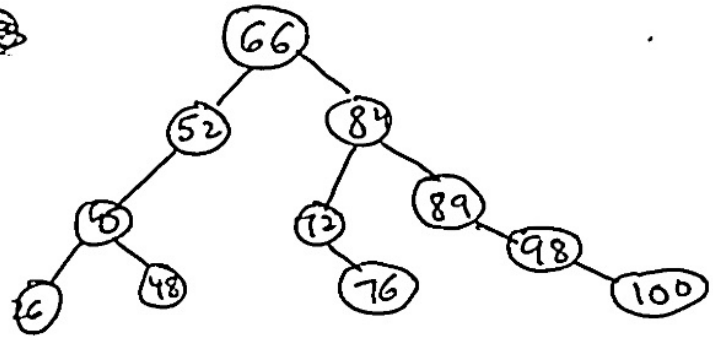
WRUOTEYIP

5.

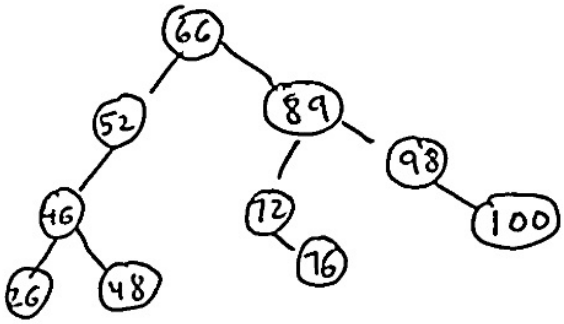


Insertion:

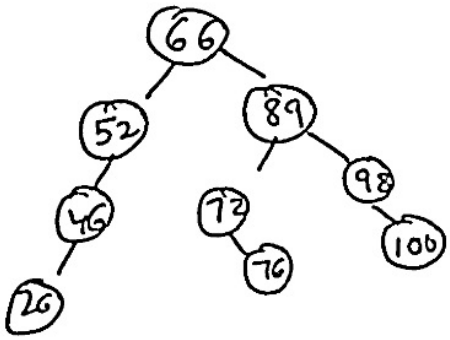
89, 46, 48, 26, 76, 98, 100



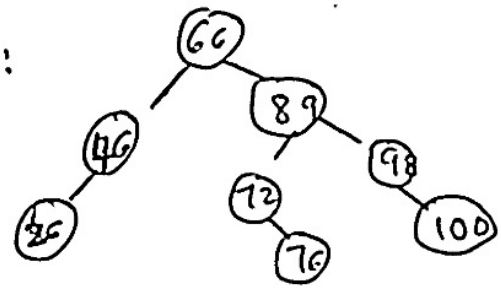
Deletion of 84:



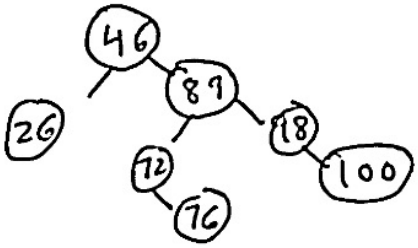
Delete 48:



Delete 52:



Delete 66:



Algorithm BST_Insert(item)

Input: *item* is data part of new node to be insert into BST.

Output: BST with new node has data part *item*.

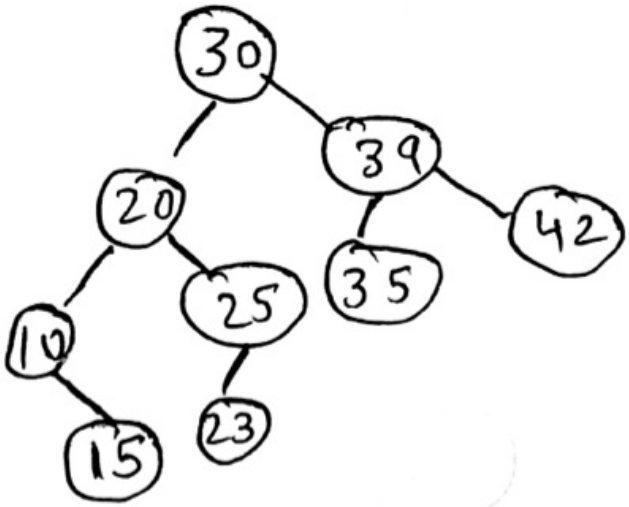
1. ptr = Root
2. flag = 0
3. while(ptr != NULL && flag == 0)
    a) if(item == ptr.data)
        i) flag = 1
        ii) print "item already exist"
    b) else if(item < ptr.data)
        i) ptr1 = ptr
        ii) ptr = ptr.LCHILD
    c) else if(item > ptr.data)
        i) ptr1 = ptr
        ii) ptr = ptr.RCHILD
    d) end if
4. end loop
5. if(ptr == NULL)
    a) new = getnewnode()
    b) new.data = item
    c) new.lchild = NULL
    d) new.rchild = NULL
    e) if(root.data == NULL)
        i) root = new
            A) print "New node inserted successfully as ROOT Node"

---

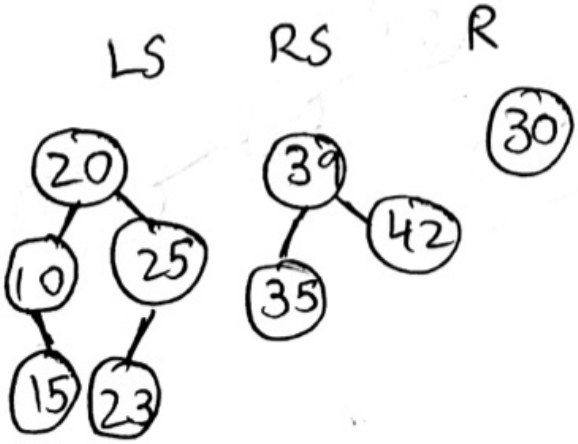
    f) else if(item < ptr1.data) /* inserting new node as left child to its parent*/
        i) ptr1.lchild = new
        ii) print "New Node is inserted successfully as LEFT child"
    g) else /* inserting new node as right child to its parent*/
        i) ptr1.rchild = new;
        ii) print "New Node is inserted successfully as Right Child"
    h) end if
6. end if

End BST_Insert

6.



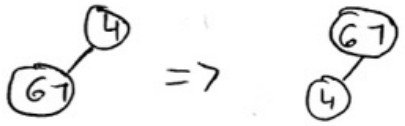
Postorder:



15 10 23 25 20 35 42 39 30

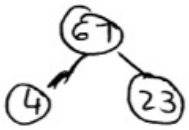
1. Max Heap : 4, 67, 23, 89, 12, 8, 7, 44, 78, 64, 70

Step 1 :



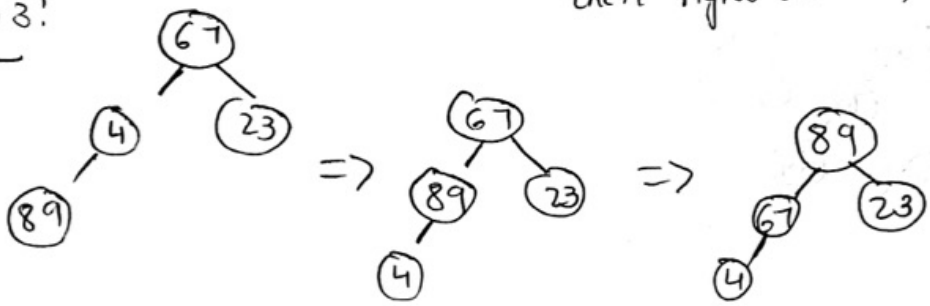
(Larger number should be swapped After comparing to root node)

Step 2 :

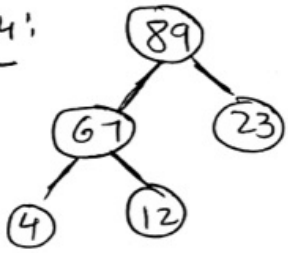


(First left side must be filled then right side of tree node)

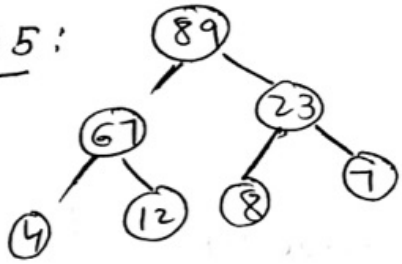
Step 3 :



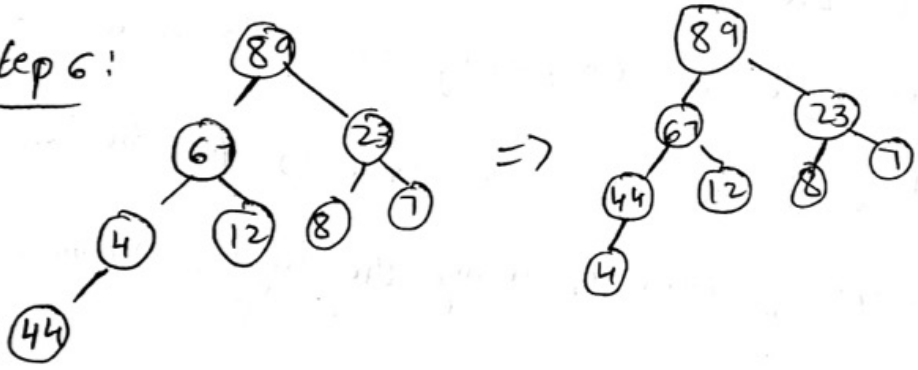
Step 4 :



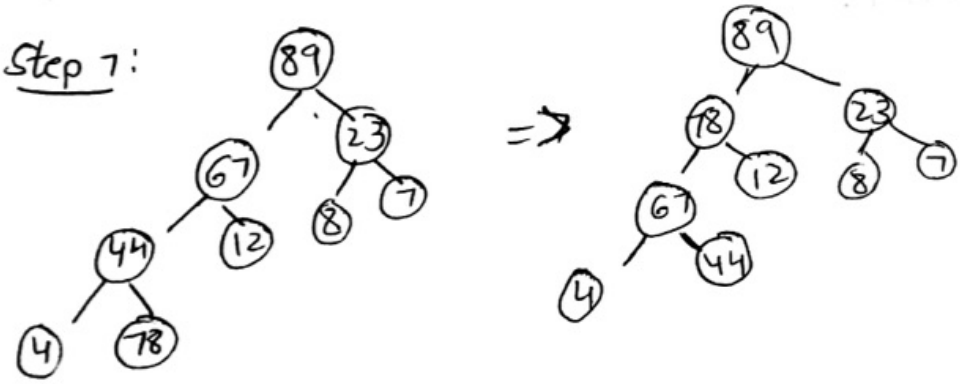
Step 5 :



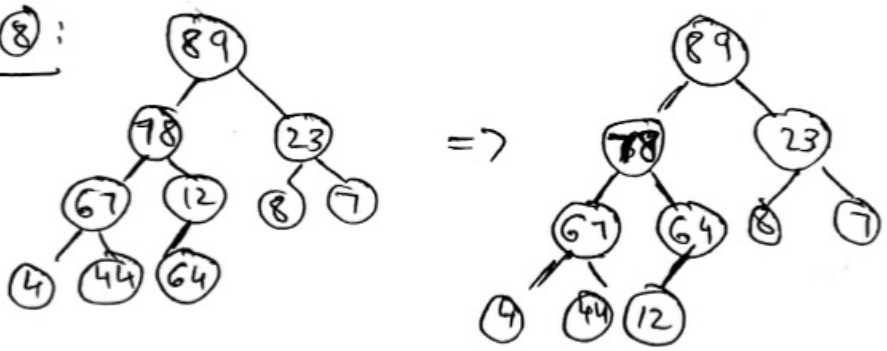
Step 6 :



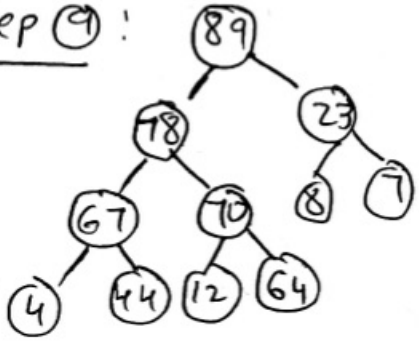
Step 7 :



Step (8):

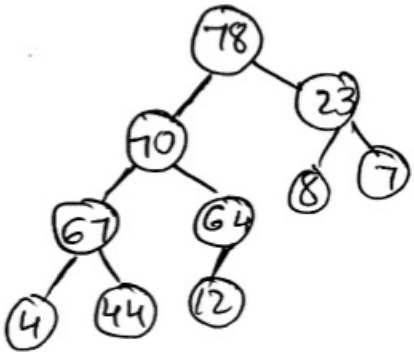


Step (9):



Applying deleteMax operation:

89 is deleted, then 78 is root node.



As we remove the '89' node, we must replace it with the next higher node by comparing the child nodes of '89' nod And we should continue the same process for below child nodes by comparing them & using the higher number for replacing the number.

Algorithm Insert_Maxheap(A[1…N], N, X)

```
{
    N=N+1; A[N]=X;
    Reheap_up(N); /* rebuild heap tree if the heap ordering property is violated */
}
Algorithm Reheap_up(node M)
{
while(M>1)
{
parent=M/2;
if(A[parent] < A[M])          /* if(A[parent] > A[M]) in case of Min-heap */
{

            temp=A[M];

            A[M]=A[parent];

            A[parent]=temp;
```

```
             M=parent;
}
}
```

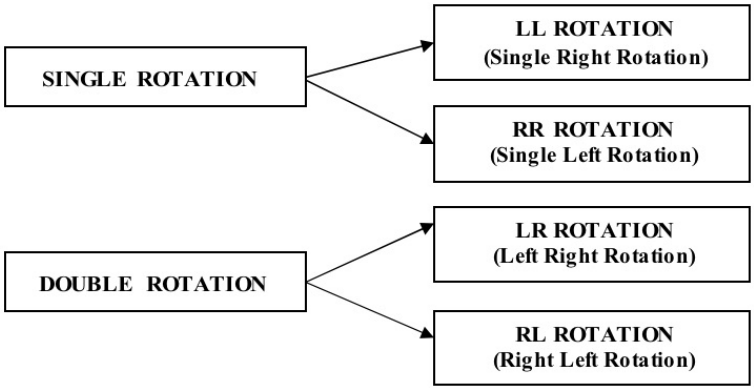
AVL TREES

- An AVL tree is a binary search tree in which difference between the heights of the left and right sub trees will be either -1, 0 or 1.
- Therefore an AVL tree is a balanced/height balanced binary search tree.
- ***Definition:*** An empty binary tree is an AVL tree. If T is a non empty binary tree with TL and TR are its left and right sub trees, then T is an AVL tree if and only if
  1) TL and TR are AVL trees and
  2) |HL-HR|<=1 where HL and HR are the height of left sub trees(TL) and right sub tree (TR) respectively of T.

  Balance factor (bf) is associated with every node is an AVL tree which may be either 0 or +1 or -1.
- ***Balance factor:*** The balance factor bf(u) of a node u is defined as the height of the left sub tree of u minus the height of the right sub tree of u.
- **bf(u)=($h(u_L)$-$h(u_R)$)** where $h(u_L)$ and $h(u_R)$ are the height of the left and right sub trees of the node u respectively.

INSERTION OPERATION ON AVL TREES

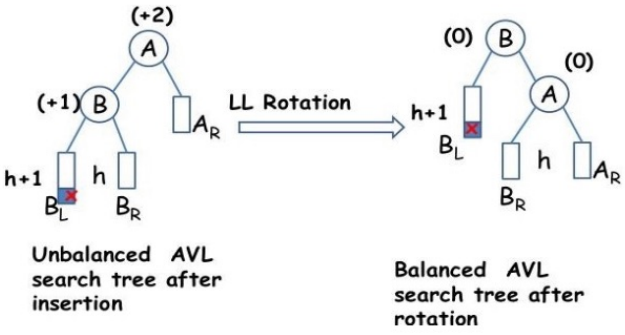
- In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows....
  - **Step 1:** Insert the new element into the tree using Binary Search Tree insertion logic.
  - **Step 2:** After insertion, check the **Balance Factor** of every node.
  - **Step 3:** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
  - **Step 4:** If after insertion the balance factor of any of the nodes turns out to be anything other than 0, +1 or -1, then the tree is said to be unbalanced.
- To balance the tree we perform rotations
- ***Rotations:*** Rotations are mechanisms which shift some of the sub trees of the unbalanced tree either to left or right to obtain a balanced tree.
- There are four rotations and they are classified into two types.



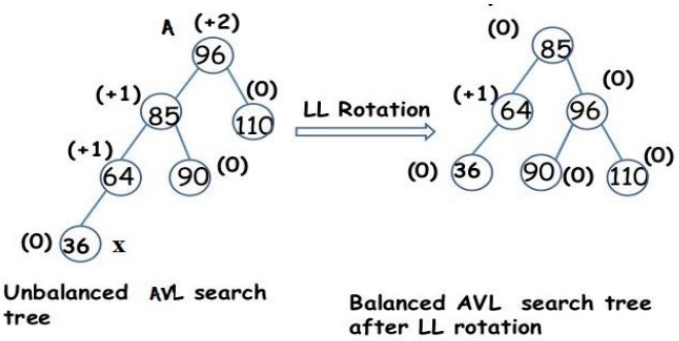
1. LL Rotation (Single Right Rotation)

- The new node 'x' is inserted in the Left sub tree of left child of node A. As a result, the balance of A becomes +2.
- To restore balance at A Single right rotation need to be applied at A.

General Notation:



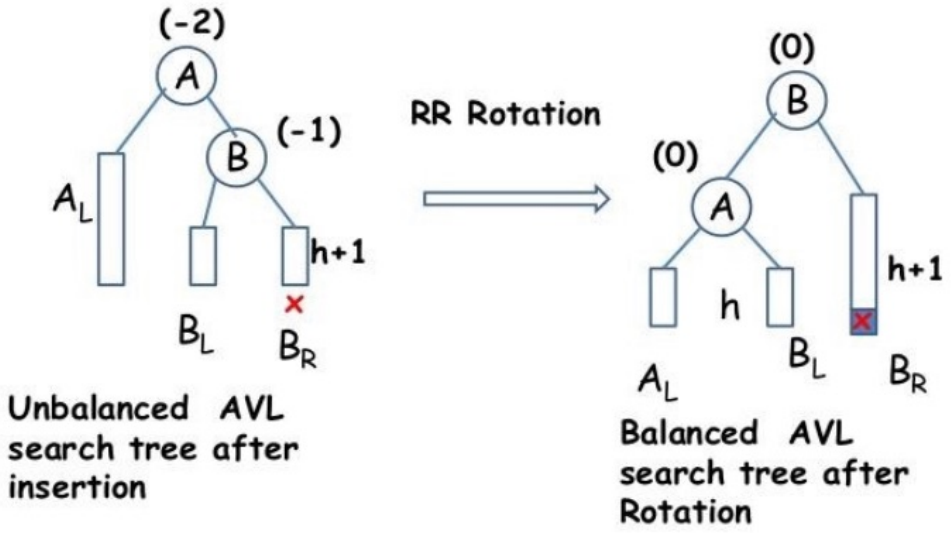
Example:



2. RR Rotation (Single Left Rotation)

- The new node 'x' is inserted in the Right sub tree of right child of node A. As a result, the balance of A becomes -2.
- To restore balance at A Single left rotation need to be applied at A.

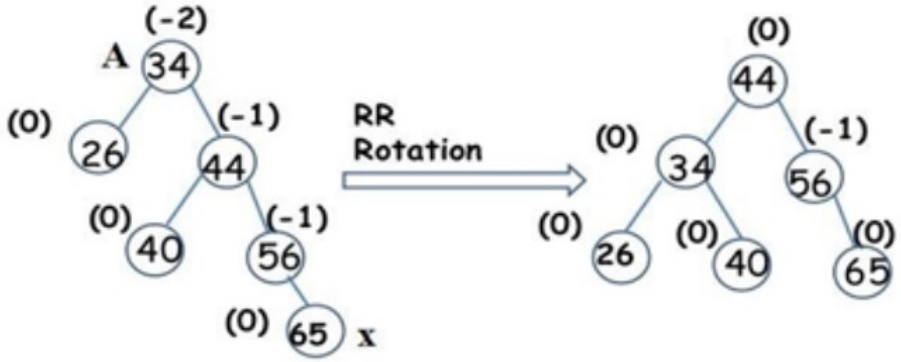
General Notation:



Unbalanced AVL search tree after insertion

Balanced AVL search tree after Rotation

Example:



Balanced AVL search tree after RR rotation

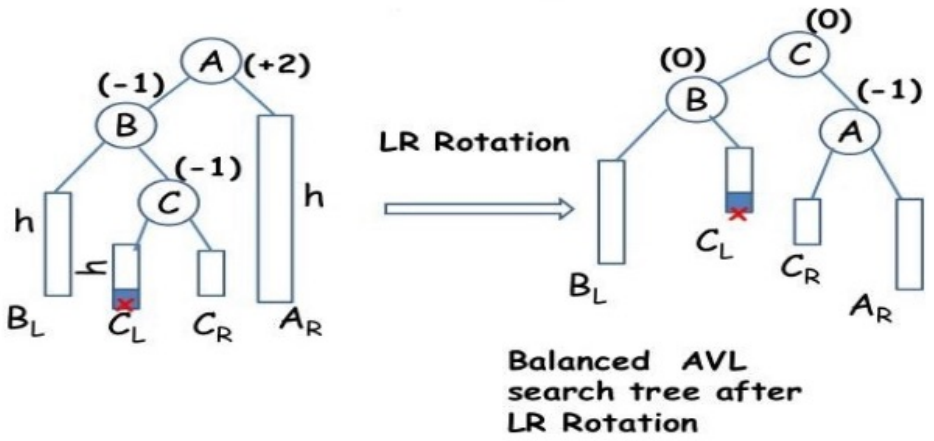
3. LR Rotation (Left Right Rotation)

- Imbalance occurred at A due to the insertion of node x in the right sub tree of left child of node A.
- LR rotation involves sequence of two rotations

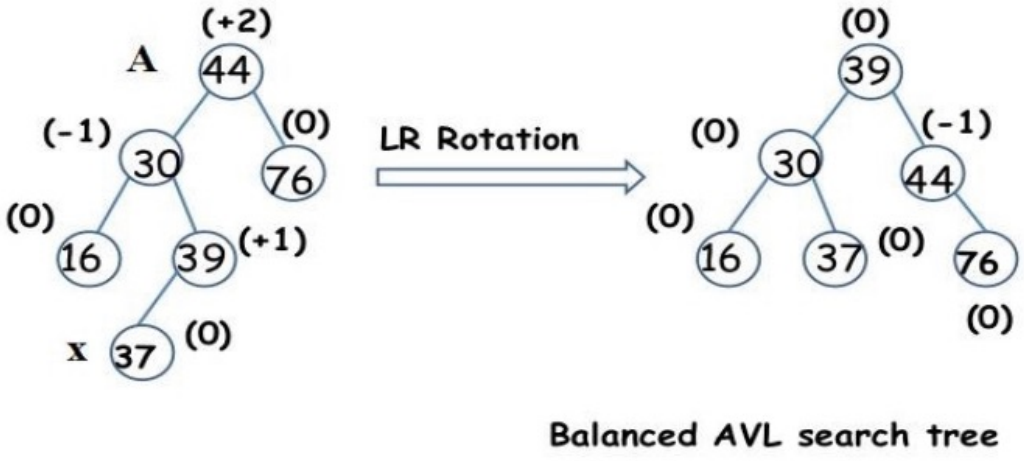
  (i) Single Left rotation/RR rotation

  (ii) Single Right rotation/LL rotation

General Notation:



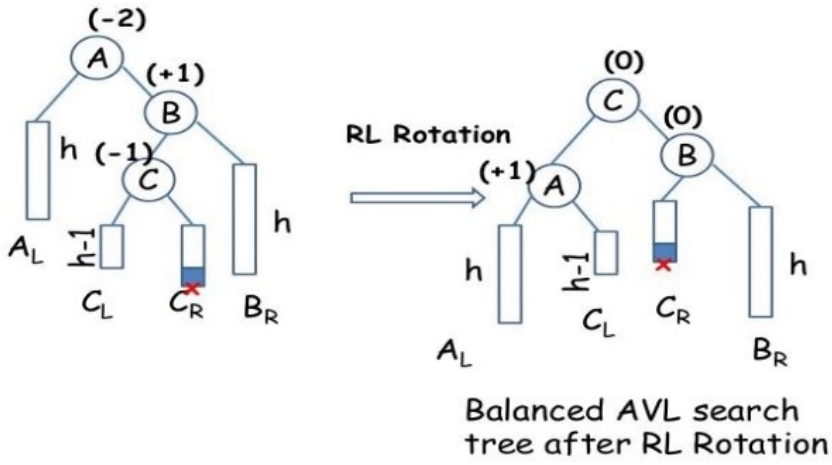
Example:



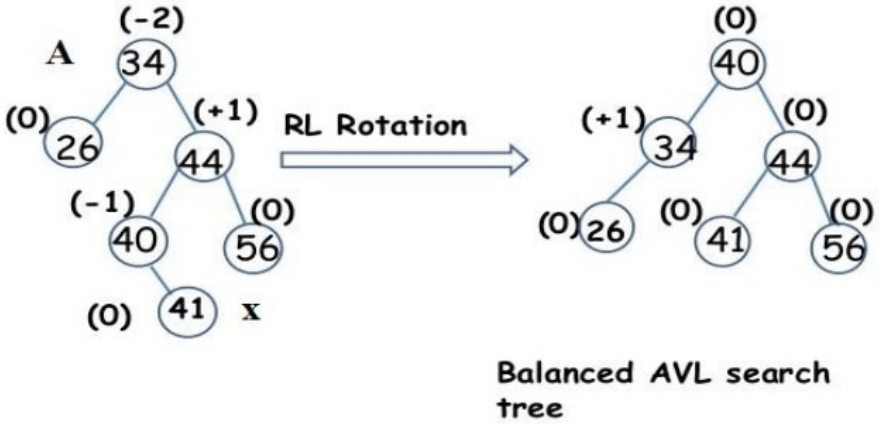
4. RL Rotation (Right Left Rotation)

- Imbalance occurred at A due to the insertion of node x in the Left sub tree of right child of node A.
- RL rotation involves sequence of two rotations
  - (i) Single Right rotation/LL rotation
  - (ii) Single Left rotation/RR rotation

General Notation:



Example:



DELETION OPERATION ON AVL TREES

The sequence of steps to be followed in deletion are:

1. Initially, the AVL tree is searched to find the node to be deleted
2. If the search is successful, delete the node. The following are the 3 possibilities for the node 'x' that is to be deleted

   (i) x is a leaf – In this case the leaf node is discarded

   (ii) x has exactly one non empty sub tree – If x has no parent, the root of its sub tree becomes the new search tree root. If x has a parent P, then we change the pointer from parent p so that it points to x's only child

---

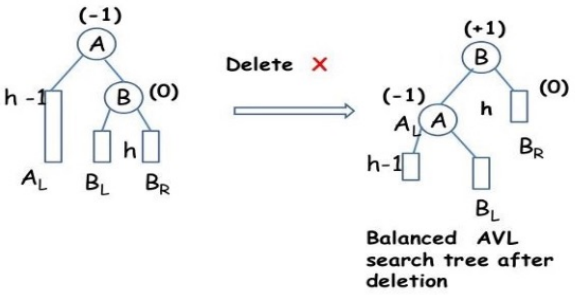
   (iii) x has two non empty sub trees – x is replaced with either the largest element in the left sub tree or the smallest element in its right sub tree.
3. After deletion of node, check the balance factor of each node
4. Rebalance the tree if the tree is unbalanced. For this AVL tree deletion rotations are used.

- On deletion of a node x from the AVL tree. Let A be the closest ancestor node on the path from x to the root node, with a balance factor of +2 or -2. To restores balance at node A, we classify the type of imbalance as follows:
- **L – type imbalance:**
  - The imbalance is of type L if the deletion took place from A's left sub tree.
  - The balance factor of A = **-2**
  - A has a right sub tree with root B
  - L-type imbalance is sub classified into types L0, L1 and L-1 depending on the balance factor of B

1. L0 ROTATION

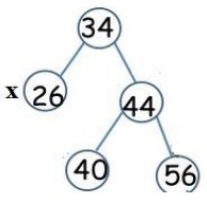
- L0 imbalance occurs if the deletion takes place from the left sub tree of A and balance factor of B is **0**
- L0 rotation is Single Left rotation, that is applied at node A

General Notation:

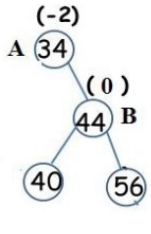


Example:

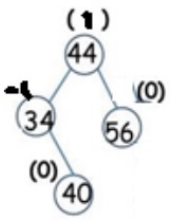
Before deletion



After deleting 26



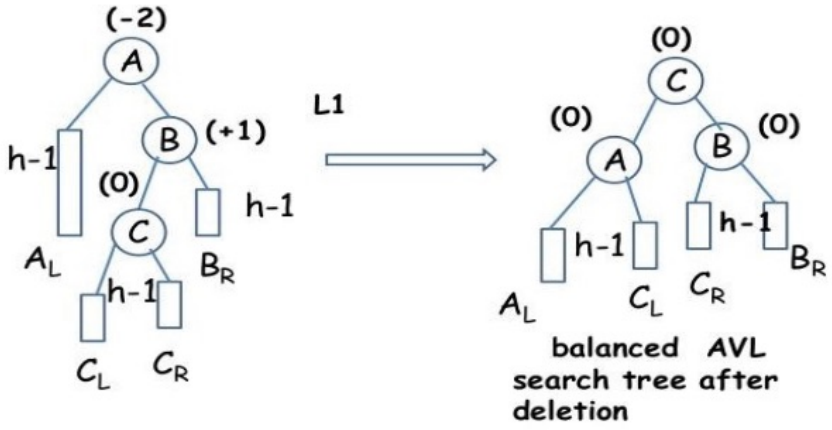
After L0 rotation



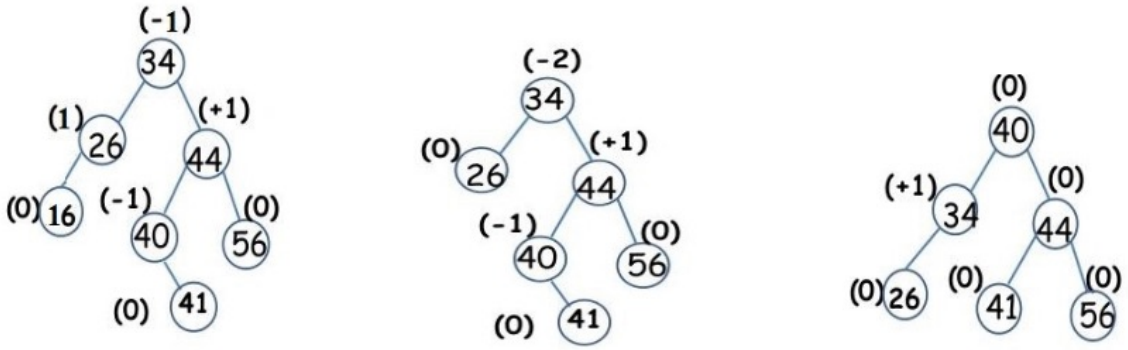
2. L1 ROTATION

- L1 imbalance occurs if the deletion takes place from the left sub tree of A and balance factor of B is **1**
- L1 rotation is RL rotation, which involves 2 rotations:
  i. Single Right
  ii. Single Left

General Notation:



Example:



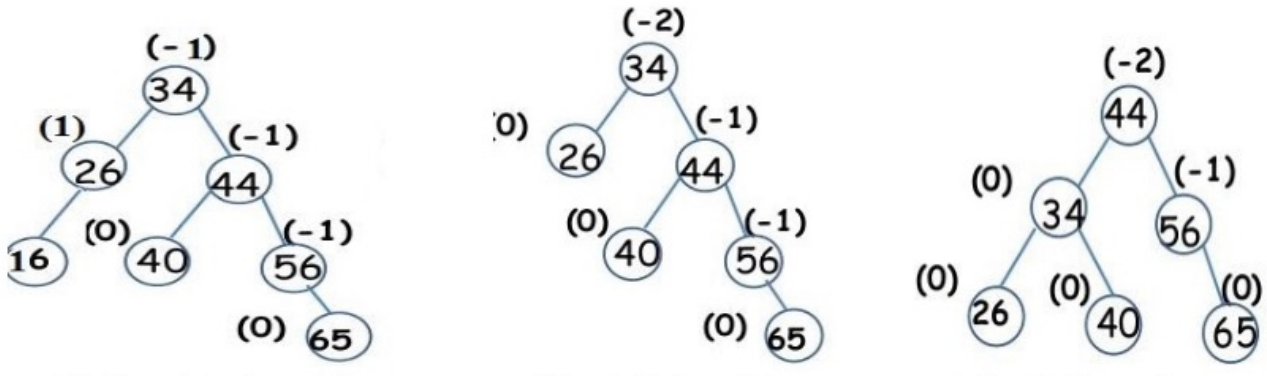
Before deletion | After deleting 16 | After L1 rotation

3. L-1 ROTATION

- L-1 imbalance occurs if the deletion takes place from the left sub tree of A and balance factor of B is **-1**
- L-1 rotation is Single Left rotation, that is applied at node A

General Notation:

Example:



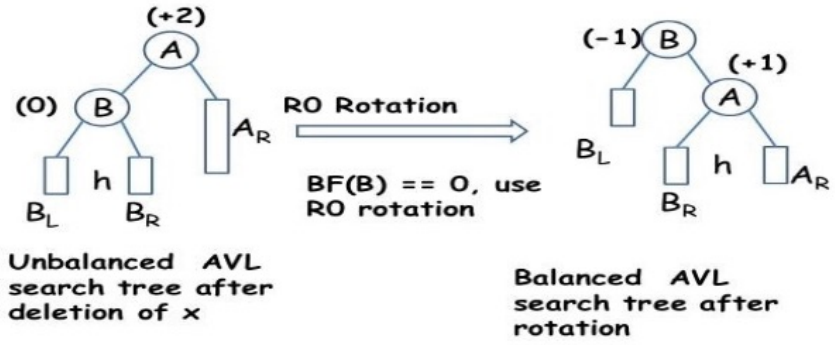
Before deletion | After deleting 16 | After L-1 rotation

- **R – type Imbalance:**
  - The imbalance is of type R if the deletion took place from A's Right sub tree.
  - The balance factor of A = **2**
  - A has a left sub tree with root B
  - R-type imbalance is sub classified into types R0, R1 and R-1 depending on the balance factor of B

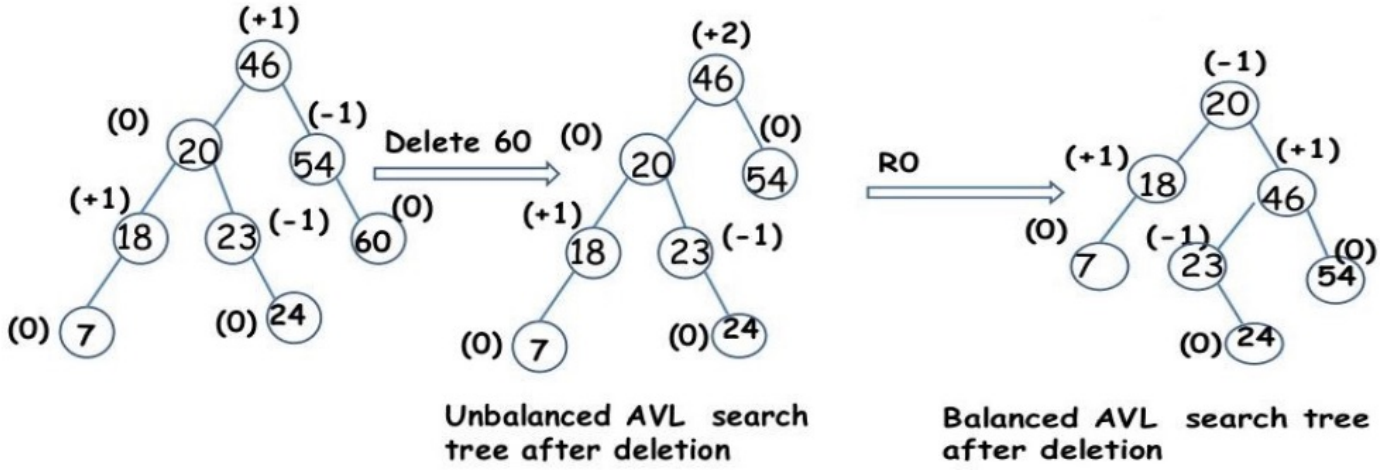
1. R0 ROTATION

- R0 imbalance occurs if the deletion takes place from the Right sub tree of A and balance factor of B is **0**
- R0 rotation is Single Right rotation, that is applied at node A

General Notation:



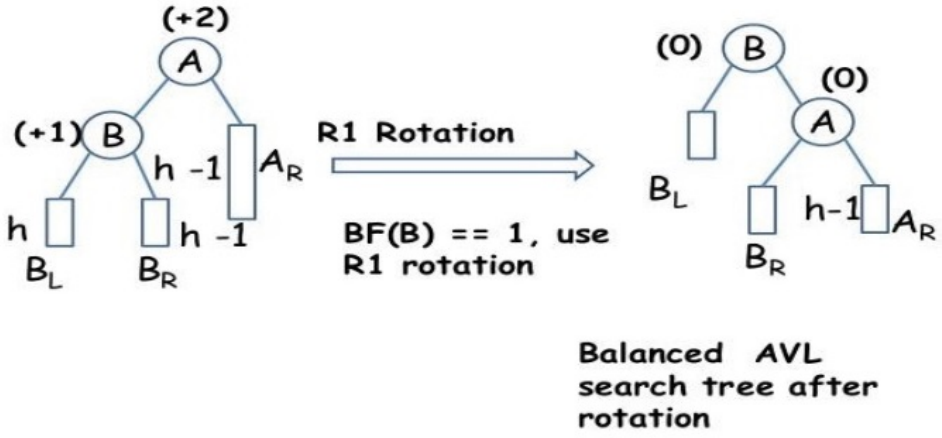
Example:



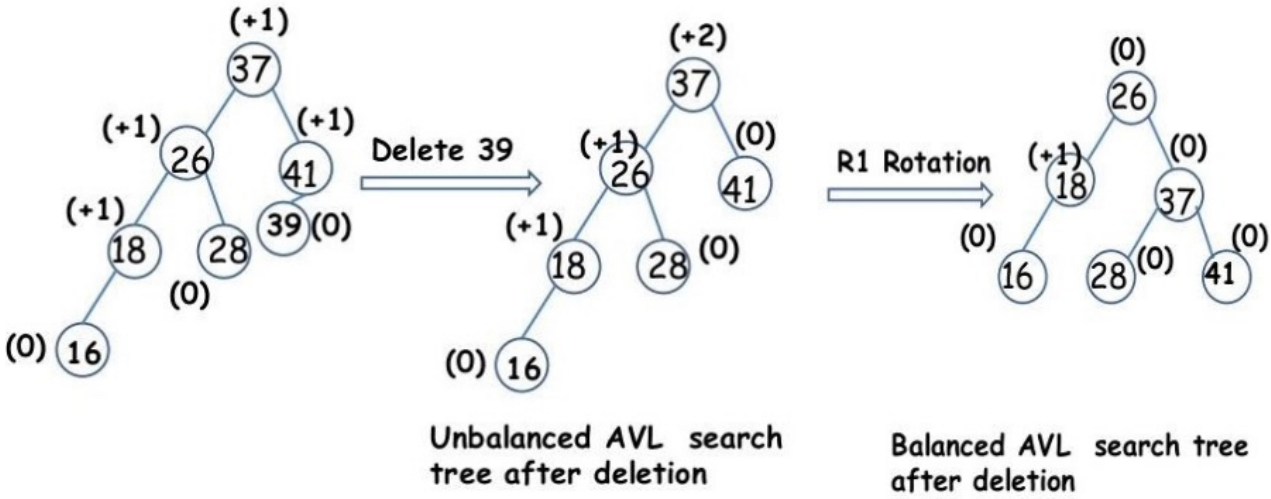
2. R1 ROTATION

- R1 imbalance occurs if the deletion takes place from the Right sub tree of A and balance factor of B is **1**
- R1 rotation is Single Right rotation, that is applied at node A

General Notation:



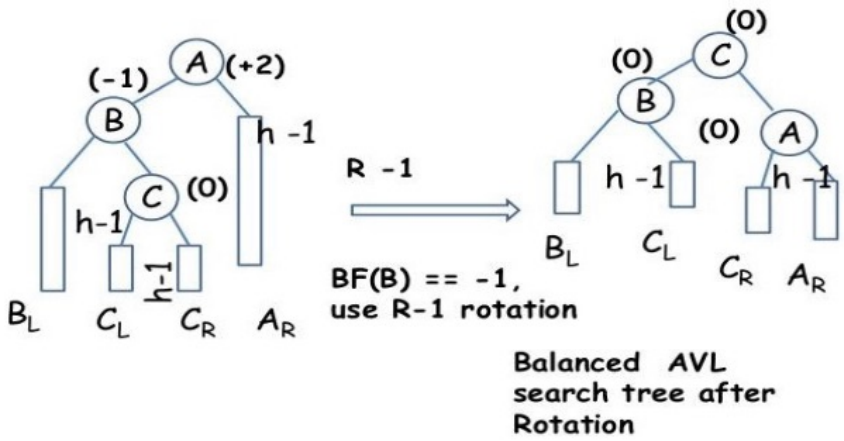
Example:



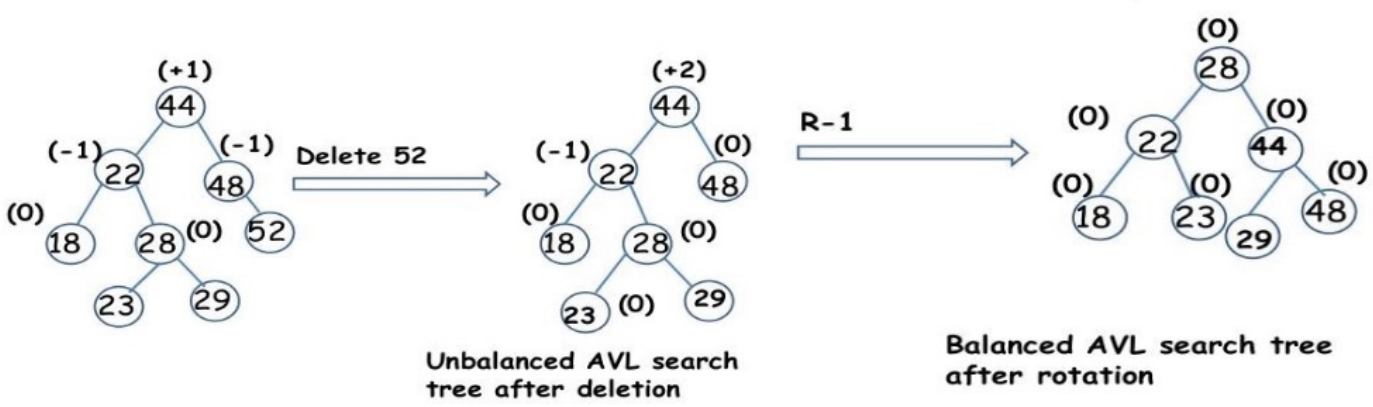
3. R-1 ROTATION

- R-1 imbalance occurs if the deletion takes place from the Right sub tree of A and balance factor of B is **-1**
- R-1 rotation is LR rotation, which involves 2 rotations:
  i. Single Left
  ii. Single Right

General Notation:



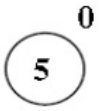
Example:



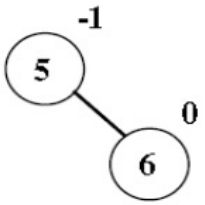
CONSTRUCTION OF AN AVL TREE

Construct AVL tree for the list by successive insertion: 5, 6, 8, 3, 2, 4, 7

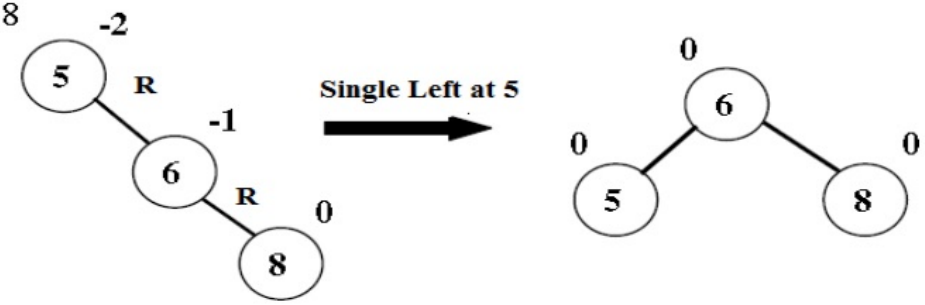
i. Insert 5 into empty tree



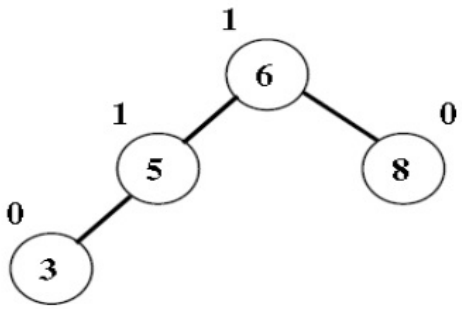
ii. Insert 6



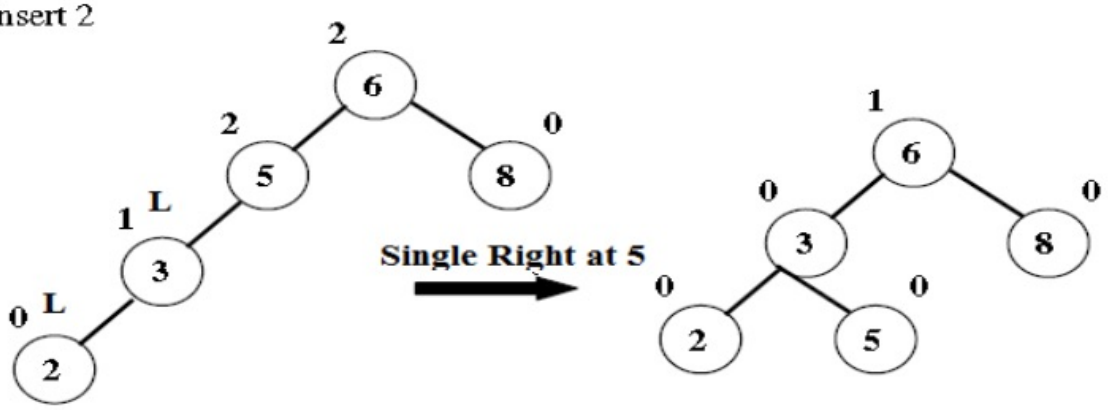
iii. Insert 8



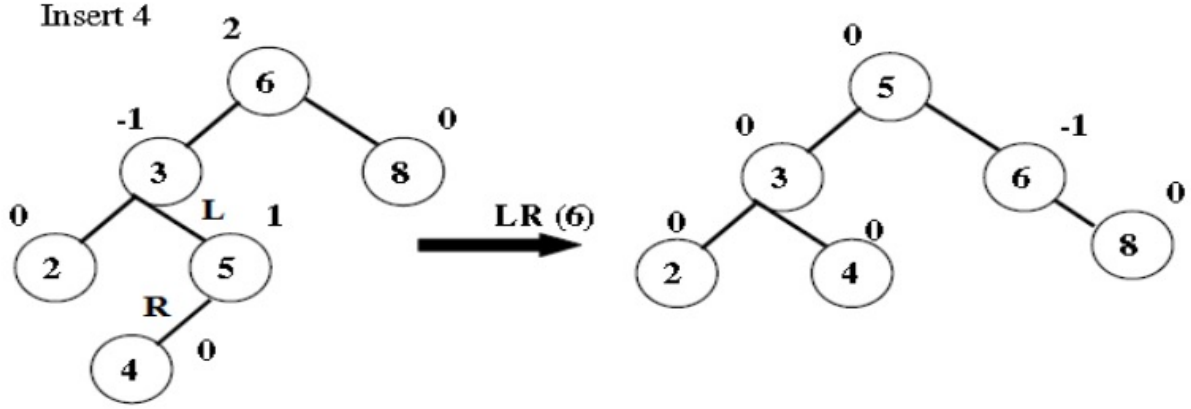
iv. Insert 3



v. Insert 2



vi. Insert 4



vii. Insert 7