RAMCO INSTITUTE OF TECHNOLOGY
Approved by AICTE, New Delhi & Affiliated to Anna University
Accredited by NAAC & An ISO 9001:2015 Certified Institution
NBA Accredited UG Programs: CSE, EEE, ECE and MECH

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND DATA SCIENCE

Academic Year: 2021-2022(Even Semester)

Regulation 2021

Subject code & Title: AD3251 - Data Structures Design

Semester/Year : II/ I

Notes

14-06-2022

Prepared by: Dr.M.Kaliappan, Professor/AD
Dr.S.Vimal, Associate Professor/AD

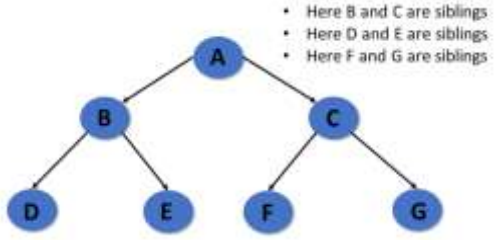
Tree Data Structure

1. What is tree?

A tree is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements.

2. What is sibling?

Two nodes that are children of the same parent are siblings



3. Define Binary Tree?

Binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets- The root, left sub-tree and right sub-tree.

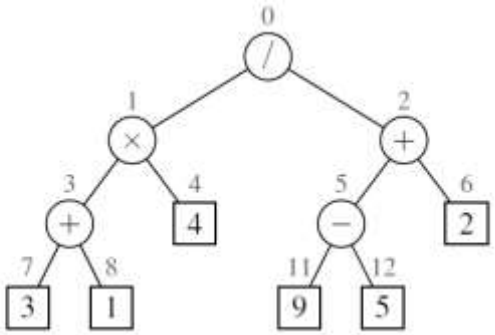
4. What is a leaf node?

The nodes that do not have any sons is called leaf node.

5. Define the term ancestor?

Node n1 is said to be the ancestor of node n2, if n1 is either the father of n2 or father of some ancestor of n2.

6. Give the array representation of the given binary tree ?



Answer:

/	×	+	+	4	−	2	3	1			9	5		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

7. What are the Applications of Tree Traversals?

- Table of Contents
- Parenthetic Representations of a Tree
- Computing Disk Space

8. Define the term descendent?

Node n2 is said to be the descendant of node n1, if n2 is either the left or right son of node n1 or son of some descendant of n1.

9. Define the term left descendant?

A node n2 is the left descendant of node n1, if n2 is either the left son of n1 or a descendant of the left son of n1.

10. Define the term right descendant?

A node n2 is the right descendant of node n1, if n2 is either the right son of n1 or a descendant of the right son of n1.

11. Define expression tree

Expression tree is a binary tree to represent the structure of an arithmetic expression

12. What is a strictly binary tree?

The binary tree, in which every non-leaf node has nonempty left and right sub-trees, is called a strictly binary tree.

13. Define level of a binary tree?

The level of a binary tree is defined as, the root of the tree has level 0, and the level of any other node in the tree is one more than the level of its father.

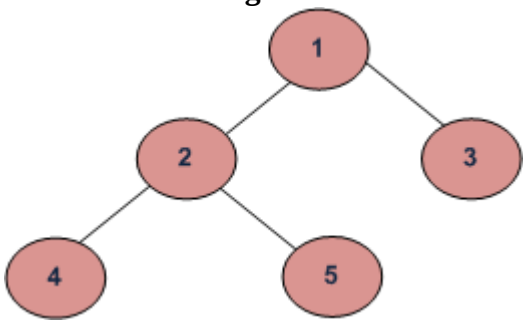
14. Define Depth of a binary tree?

The depth of a binary tree is the maximum level of any leaf in the tree. This is same as the length of the longest path from the root to any leaf.

15. What is a complete binary tree?

A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d.

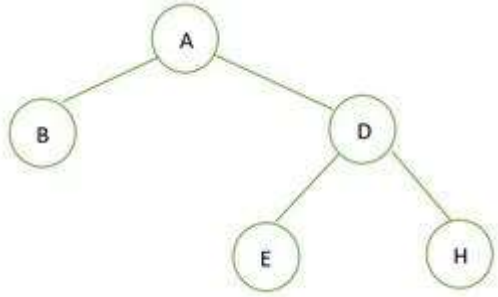
16. Write the in-order, pre-order, post-order and Breadth-First or Level Order Traversal for the given tree.



Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1
Breadth-First or Level Order Traversal: 1 2 3 4 5

17. What is Full Binary Tree?

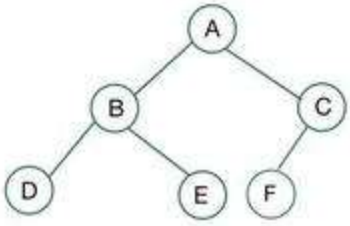
For a full binary tree, every node has either 2 children or 0 children.



18. What is sibling?

Nodes having the same parent are sibling. Example- J, K are siblings as they have the same parent E.

19. What is Degree of a node in a tree? What is the Degree of and H for the given tree?

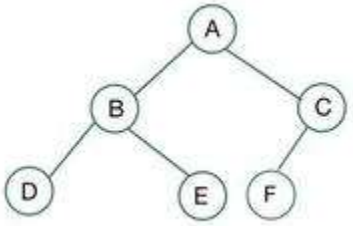


Degree of a node is a number of children of a particular parent. Example- Degree of A is 2 and Degree of H is 1. Degree of L is 0.

20. What is an Internal/External node?

Leaf nodes are external nodes and non-leaf nodes are internal nodes.

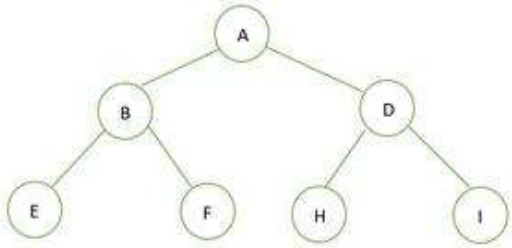
21. What is Height of a node in a tree? What is the height of E in the given node?



Number of edges to reach the destination node, Root is at height 0. Example – Height of node E is 2 as it has two edges from the root

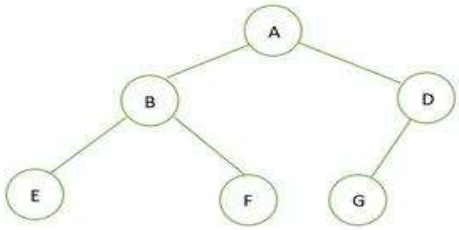
22. What is Perfect Binary Tree?

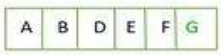
A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.



23. Draw a tree for the given data in the array ?

Answer:





24. What are the application of the complete binary tree?

- Heap Sort
- Heap sort based data structure

25. What are the properties of complete binary tree?

- In a complete binary tree number of nodes at depth **d** is 2^d.
- In a complete binary tree with **n** nodes height of the tree is **log (n+1)**.
- All the levels **except the last level** are completely full.

26. What are the basic operations on a binary tree?

Let p be a pointer to a node and x be the information. Now, the basic operations are:

i) info(p)
ii) father(p)
iii) left(p)
iv) right(p)
v) brother(p)
vi) isleft(p)
vii) isright(p)

27. What is the length of the path in a tree?

The length of the path is the number of edges on the path. In a tree there is exactly one path form the root to each node.

28. Define expression trees?

The leaves of an expression tree are operands such as constants or variable names and the other nodes contain operators.

29. Define strictly binary tree?

If every non-leaf node in a binary tree has nonempty left and right sub-trees, the tree is termed as a strictly binary tree.

30. Define complete binary tree?

A complete binary tree of depth d is the strictly binary tree all of whose are at level d.

31. What are the applications of binary tree?

Binary tree is used in data processing.

a. File index schemes
b. Hierarchical database management system

32. What is meant by traversing?

Traversing a tree means processing it in such a way, that each node is visited only once.

33. What are the different types of traversing?

a. Pre-order traversal.
b. In-order traversal
c. Post-order traversal

34. What are the two methods of binary tree implementation?

a. Linear representation.
b. Linked representation

35. Define pre-order traversal?

a. Visit the root node

b. Traverse the left sub-tree
c. Traverse the right sub-tree

36. Define post-order traversal?

a. Traverse the left sub-tree
b. Traverse the right sub-tree
c. Visit the root node

37. Define in -order traversal?

a. Traverse the left sub-tree
b. Visit the root node
c. Traverse the right sub-tree

38. Explain the steps for Depth-first order traversal?

- Visit the root
- Traverse the left sub-tree
- Traverse the right sub-tree.

39. Explain the steps for Symmetric order traversal?

- Traverse the left sub-tree
- Visit the root
- Traverse the right sub-tree.

40. Explain the steps for post-order traversal?

- Traverse the left sub-tree
- Traverse the right sub-tree.
- Visit the root

41. What is a binary search tree?

A binary tree in which all the elements in the left sub-tree of a node n are less than the contents of n, and all the elements in the right sub-tree of n are greater than or equal to the contents of n is called a binary search tree.

42. What are the methods to indicate the non-existent nodes in a binary tree, when a sequential representation is made?

One way is to use negative numbers as null indicators in a positive binary tree. Another way is to use a used field to indicate whether the node is in use or not.

43. How can you say recursive procedure is efficient than non-recursive?

There is no extra recursion. The automatic stacking and unstacking make it more efficient. There are no extraneous parameters and local variables used.

44. Construction of expression trees?

1. Covert the given infix expression into postfix notation
2. Create a stack and read each character of the expression and push into the stack, if operands are encountered.
3. When an operator is encountered pop 2 values from the stack. From a tree using the operator.

45. Define lazy deletion?

When an element is to be deleted it is left in the tree itself and marked a s being deleted. This is called as lazy deletion and is an efficient procedure if duplicate keys are present in the element can be decremented of the element can be decremented.

46. Define AVL tree?

AVL tree also called as height balanced tree. It is a height balanced tree in which every node will have a balancing factor of –1,0,1 Balancing factor Balancing factor of a node is given by the difference between the height of the left sub tree and the height of the right sub tree.

47 .what are the various operation performed in the binary search tree ?

1.insertion
2.deletion
3.find
4.find min
5.find max

48. What are the various transformation performed in AVL tree?

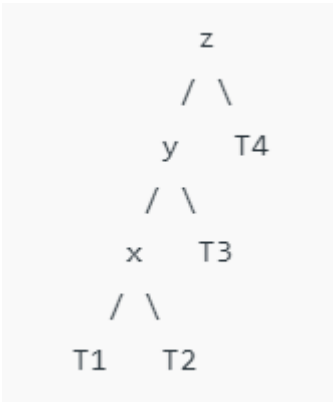
1.single rotation
- single L rotation
- single R rotation

2.double rotation
-LR rotation
-RL rotation

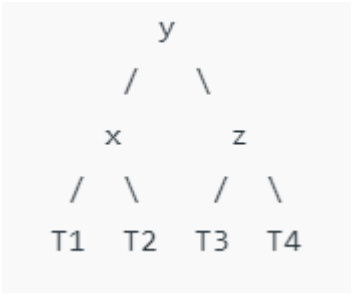
49. Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree.

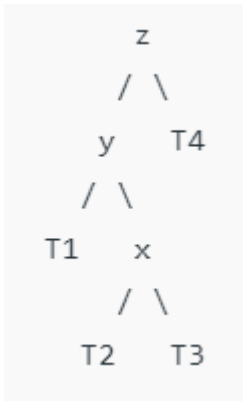
50. What is the right rotation of the given tree?



Answer

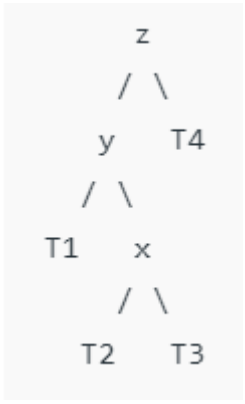


51. What is the left rotation of the given tree?

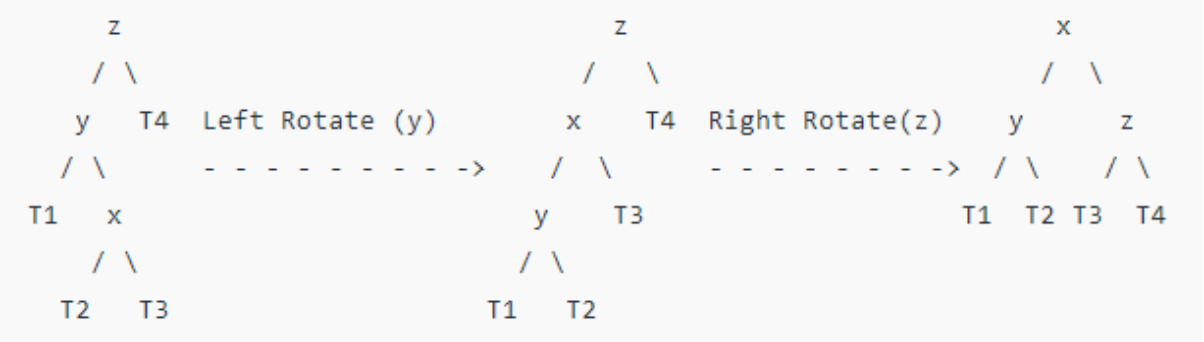


Answer

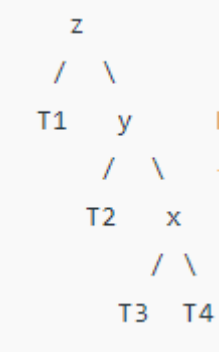
52. What is the left-right rotation of the given tree?



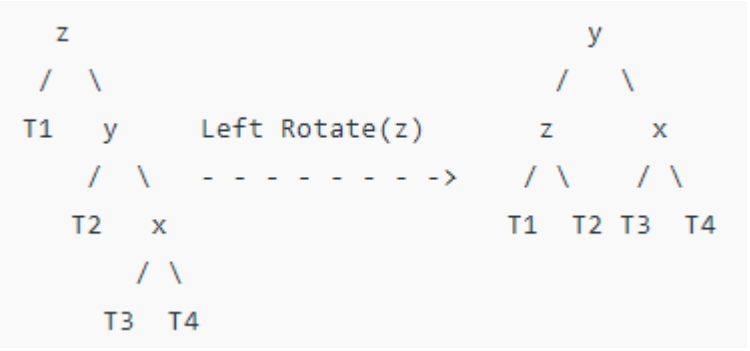
Answer :



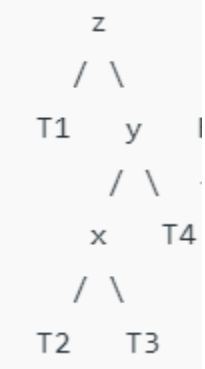
53. What is the right-right rotation of the given tree?



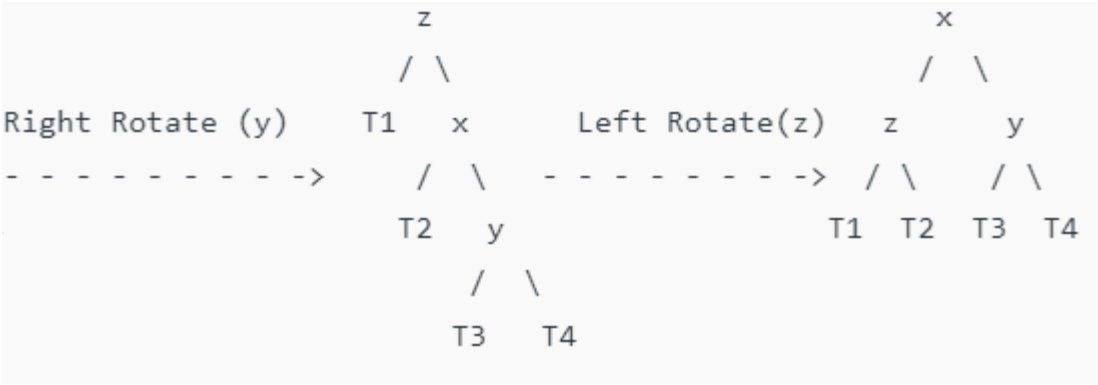
Answer:



54. What is the Right Left Case rotation of the given tree?



Answer:

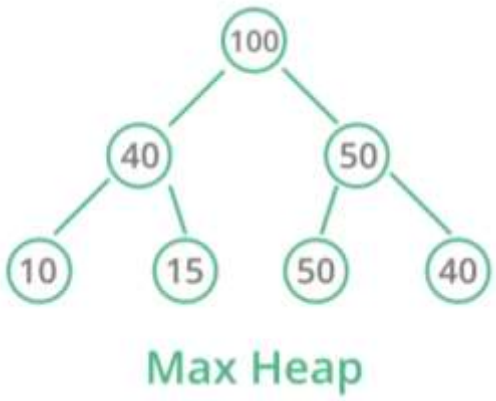


55. What is heap?

A heap is **a tree-based data structure in which all the nodes of the tree are in a specific order**

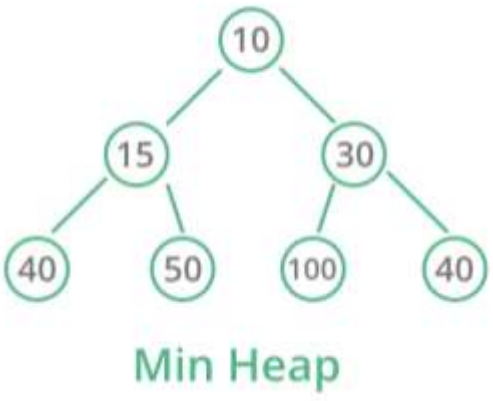
56. What is Max heap ?

In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



57. What is Min-Heap?

In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



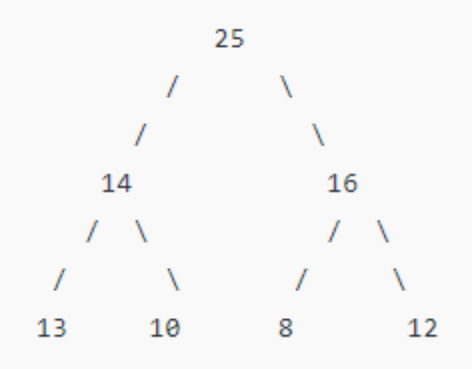
58. What are the applications of Heap?

- Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm.
- **Order statistics:** The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.
- **Priority Queues**: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time.

59. In a binary max heap containing n numbers, the smallest element can be found in time (GATE CS 2006)

Time complexity : O(n)
In a max heap, the smallest element is always present at a leaf node. So we need to check for all leaf nodes for the minimum value. Worst case complexity will be O(n)

60. Consider the following number 25, 14,16,13,10,8,12 to build a binary max-heap implemented using an array. (GATE CS 2009)



61. What are the two relational property of heap?

***Heap-Order Property*:** In a heap *T*, for every position *p* other than the root, the key stored at *p* is greater than or equal to the key stored at *p*'s parent.
Complete Binary Tree Property: A heap *T* with height *h* is a ***complete*** binary tree if levels 0,1,2, . . . ,*h*−1 of *T* have the maximum number of nodes possible (namely, level *i* has 2*i* nodes, for $0 \leq i \leq h-1$) and the remaining nodes at level *h* reside in the leftmost possible positions at that level

62. What is multi -way search tree ?

Multi-way trees or m-Way tree are generalised versions of binary trees where each node contains multiple elements. In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children.

63. What are the operations of Heap ?

heapify(iterable) :- This function is used to convert the iterable into a heap data structure. i.e. in heap order.
heappush(heap, ele) :- This function is used to insert the element mentioned in its arguments into heap. The order is adjusted, so as heap structure is maintained.

heappop(heap) :- This function is used to remove and return the smallest element from heap. The order is adjusted, so as heap structure is maintain.

64. What is heapify ?

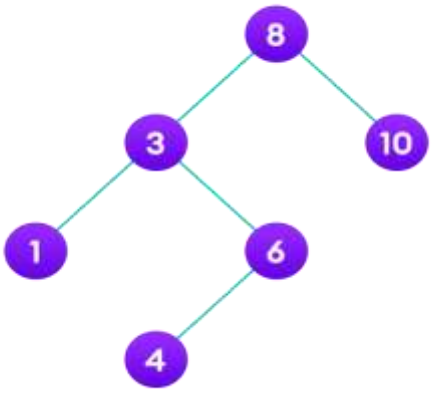
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap

---

16 marks Questions and Answer

1. Discuss about binary tree. Write a python program to construct a binary search tree?

- Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.
- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.




```
#Binary tree insertion and searching
class Node:

    def __init__(self, data):

        self.left = None
        self.right = None
        self.data = data

# Insert method to create nodes
    def insert(self, data):

        if self.data:
            if data < self.data:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.data:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.data = data
# findval method to compare the value with nodes
```

```
    def findval(self, val):
        if val < self.data:
            if self.left is None:
                return str(val)+" is not Found"
            return self.left.findval(val)
        elif val > self.data:
            if self.right is None:
                return str(val)+" is not Found"
            return self.right.findval(val)
        else:
            return str(self.data) + " is found"
# Print the tree

    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print(self.data),
        if self.right:
            self.right.PrintTree()

root = Node(27)
root.insert(14)
root.insert(35)
root.insert(31)
root.insert(10)
root.insert(19)
print(root.findval(7))
print(root.findval(14))
```

---

2. How will you delete a node from a binary search tree. Write a python program to delete a node.

Deletion Operation

There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

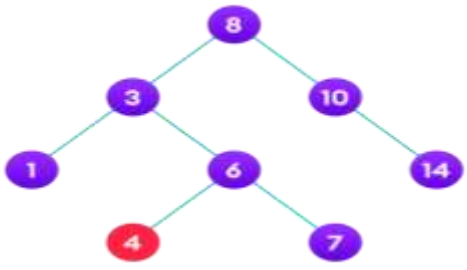


Figure: 4 is to be deleted

Case II
In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

- Replace that node with its child node.
- Remove the child node from its original position

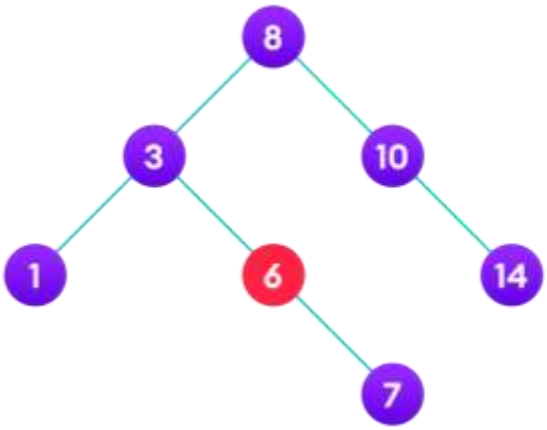
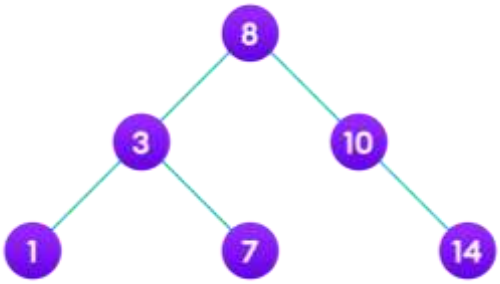
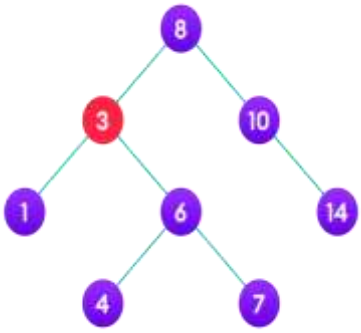


Figure: 6 to be deleted

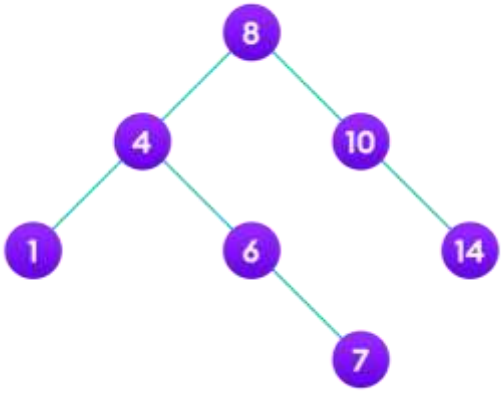


Case III
In the third case, the node to be deleted has two children. In such a case follow the steps below:

- Get the inorder successor of that node.
- Replace the node with the inorder successor.
- Remove the inorder successor from its original position



3 to be deleted. Copy the value of the inorder successor (4) to the node



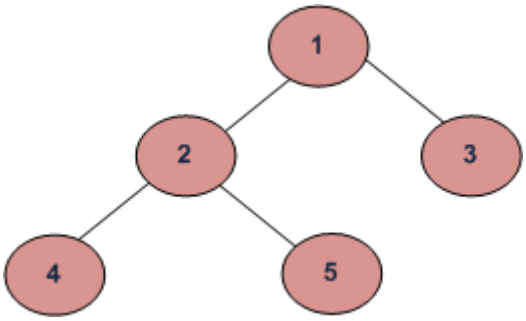

```
# A class to store a BST node
class Node:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right

# Function to find the maximum value node in the subtree rooted at `ptr`
def findMaximumKey(ptr):
    while ptr.right:
        ptr = ptr.right
    return ptr

# Function to delete a node from a BST
def deleteNode(root, key):

    # base case: the key is not found in the tree
    if root is None:
        return root
    # To find key
    # if the given key is less than the root node, recur for the left subtree
    if key < root.data:
        root.left = deleteNode(root.left, key)

    # if the given key is more than the root node, recur for the right subtree
    elif key > root.data:
        root.right = deleteNode(root.right, key)

    # key found
    else:

        # Case 1: node to be deleted has no children (it is a leaf node)
        if root.left is None and root.right is None:
            # update root to None
            return None

        # Case 2: node to be deleted has two children
```

```
        elif root.left and root.right:
            # find its inorder predecessor node
            predecessor = findMaximumKey(root.left)

            # copy value of the predecessor to the current node
            root.data = predecessor.data

            # recursively delete the predecessor. Note that the
            # predecessor will have at most one child (left child)
            root.left = deleteNode(root.left, predecessor.data)

        # Case 3: node to be deleted has only one child
        else:
            # choose a child node
            child = root.left if root.left else root.right
            root = child

    return root

 # Function to perform inorder traversal on the BST
def inorder(root):
    if root is None:
        return
    inorder(root.left)
    print(root.data, end=' ')
    inorder(root.right)


if __name__ == '__main__':

    keys = [15, 10, 23, 8, 12,18,28,16,19, 22,30,20,23]

    root = None
    for key in keys:
        root = insert(root, key)

    root = deleteNode(root, 23)
    inorder(root)
```

---

3. Discuss about tree traversal and write a python program for tree traversal or depth first traversal.

Traversal is a process to visit all the nodes of a tree.



Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1
Breadth-First or Level Order Traversal: 1 2 3 4 5

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

```
class Node:
    def __init__(self, item):
        self.left = None
        self.right = None
        self.val = item
def inorder(root):

    if root:
        inorder(root.left) # Traverse left
        print(str(root.val) + "->", end='') # Traverse root
        inorder(root.right)          # Traverse right
```

```
def postorder(root):

    if root:
        postorder(root.left)        # Traverse left
        postorder(root.right)       # Traverse right
        print(str(root.val) + "->", end='') # Traverse root


def preorder(root):

    if root:
        print(str(root.val) + "->", end='')  # Traverse root
        preorder(root.left)                  # Traverse left
        preorder(root.right)                 # Traverse right


root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("Inorder traversal ")
inorder(root)

print("\nPreorder traversal ")
preorder(root)

print("\nPostorder traversal ")
postorder(root)
```

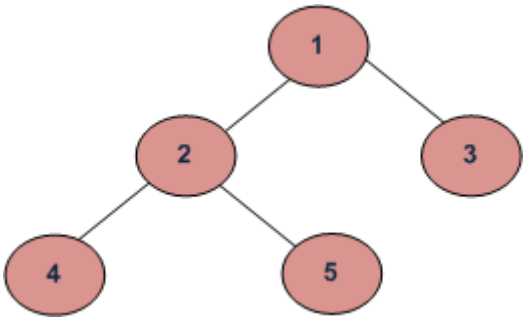
Output

```
Inorder traversal
4->2->5->1->3->
Preorder traversal
1->2->4->5->3->
Postorder traversal
4->5->2->3->1->
```

---

4. Discuss about Breadth-First or Level Order Traversal and write a python program for Breadth-First or Level Order Traversal.

Traversal is a process to visit all the nodes of a tree.



Breadth-First or Level Order Traversal: 1 2 3 4 5

```
class Node:

  def __init__(self, key):        # A utility function to create a new node
    self.data = key
    self.left = None
    self.right = None

def printLevelOrder(root):  # Function to print level order traversal of tree
  h=height(root)
  print(h)
  for i in range(1,h+1):
    printCurrentLevel(root,i)

def printCurrentLevel(root, level):  # Print nodes at a current level
  if root is None:
    return
  if level == 1:
    print(root.data, end=" ")
  elif level > 1:
    printCurrentLevel(root.left, level-1)
    printCurrentLevel(root.right, level-1)

def height(node):       #Compute the height of a tree
  if node is None:
    return 0
  else:
    lheight=height(node.left)
    rheight=height(node.right)

  if lheight > rheight:   # Use the larger one
    return lheight+1
  else:
    return rheight+1
```

```
# Driver program to test above function
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

print("Level order traversal of binary tree is -")
printLevelOrder(root)
```

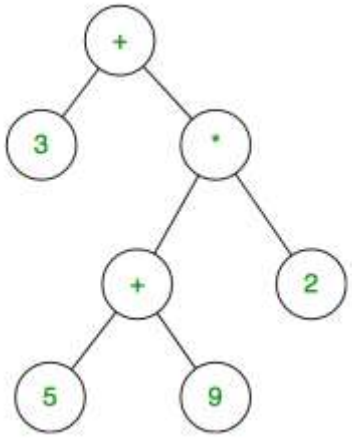
Output

1 2 3 4 5

---

5. Python program to evaluate expression tree

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be:




```
class node:
  def __init__(self, value):
    self.left = None
    self.data = value
    self.right = None

def evaluateExpressionTree(root):

  if root is None:        # empty tree
    return 0

  if root.left is None and root.right is None:  # leaf node
    return int(root.data)

  left_sum = evaluateExpressionTree(root.left) # evaluate left tree
```

```
  right_sum = evaluateExpressionTree(root.right) # evaluate right tree

  # check which operation to apply
  if root.data == '+':
    return left_sum + right_sum

  elif root.data == '-':
    return left_sum - right_sum

  elif root.data == '*':
    return left_sum * right_sum

  else:
    return left_sum // right_sum

# Driver function to test above problem
if __name__ == '__main__':

  # creating a sample tree
  root = node('+')
  root.left = node('*')
  root.left.left = node('5')
  root.left.right = node('-4')
  root.right = node('-')
  root.right.left = node('100')
  root.right.right = node('20')
  print (evaluateExpressionTree(root))

  root = None

  # creating a sample tree
  root = node('+')
  root.left = node('*')
  root.left.left = node('5')
  root.left.right = node('4')
  root.right = node('-')
  root.right.left = node('100')
  root.right.right = node('/')
  root.right.right.left = node('20')
  root.right.right.right = node('2')
  print (evaluateExpressionTree(root))
```

Output

60
110

---

6. How is Min Heap represented? Write a python code to build a min heap data structure?

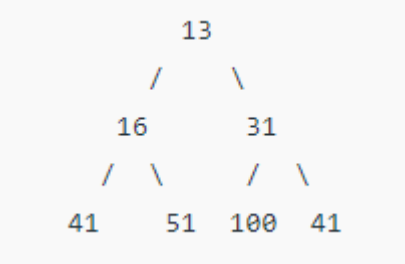
A Min Heap is a Complete Binary Tree. A Min heap is typically represented as an array. The root element will be at **Arr[0]**. For any ith node, i.e., **Arr[i]**:

- **Arr[(i -1) / 2]** returns its parent node.
- **Arr[(2 * i) + 1]** returns its left child node.
- **Arr[(2 * i) + 2]** returns its right child node.

MinHeap operations

1. Insertion O(log n): Finding the exact position of the new element is performed in log n*logn* since it is only compared with the position of the parent nodes.
2. Delete Min O(log n): After the minimum element is removed, the heap has to put the new root in place.
3. Find Min O(1): This is possible because the heap data structure always has the minimum element on the root node.
4. Heapify O(n): This operation rearranges all the nodes after deletion or insertion operation. The cost of this operation is n since all the elements have to be moved to keep the heap properties.
5. Delete O(log n): A specific element from the heap can be removed in log n*logn* time.

Example:




```
class MinHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def sift_up(self, i): # Moves the value up in the tree to maintain the heap property

        while i // 2 > 0: # While the element is not the root or the left element
            # If the element is less than its parent swap the elements
            if self.heap_list[i] < self.heap_list[i // 2]:
                self.heap_list[i], self.heap_list[i // 2] = self.heap_list[i // 2], self.heap_list[i]
            i = i // 2 # Move the index to the parent to keep the properties

    def insert(self, k):
        self.heap_list.append(k) # Append the element to the heap.
        self.current_size += 1  # Increase the size of the heap
        self.sift_up(self.current_size) # Move the element to its position from bottom to the top
```

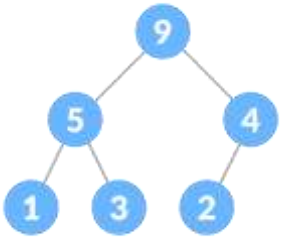
```
    def sift_down(self, i):
        while (i * 2) <= self.current_size:  # if the current node has at least one child
            mc = self.min_child(i) # Get the index of the min child of the current node

            # Swap the values of the current element is greater than its min child
            if self.heap_list[i] > self.heap_list[mc]:
                self.heap_list[i], self.heap_list[mc] = self.heap_list[mc], self.heap_list[i]
            i = mc

    def min_child(self, i):
        # If the current node has only one child, return the index of the unique child
        if (i * 2)+1 > self.current_size:
            return i * 2
        else:
            # Herein the current node has two children
Return the index of the min child according to their values
            if self.heap_list[i*2] < self.heap_list[(i*2)+1]:
                return i * 2
            else:
                return (i * 2) + 1

    def delete_min(self):
        if len(self.heap_list) == 1: # Equal to 1 since the heap list was initialized with a value
            return 'Empty heap'
        root = self.heap_list[1]  # Get root of the heap (The min value of the heap)
        # Move the last value of the heap to the root
        self.heap_list[1] = self.heap_list[self.current_size]

        *self.heap_list, _ = self.heap_list # Pop the last value since a copy was set on the root
        self.current_size -= 1  # Decrease the size of the heap
        self.sift_down(1) # Move down the root (value at index 1) to keep the heap property

        return root

my_heap = MinHeap()
my_heap.insert(5)
my_heap.insert(6)
my_heap.insert(7)
my_heap.insert(9)
my_heap.insert(13)
my_heap.insert(11)
my_heap.insert(10)

print(my_heap.delete_min()) # removing min node i.e 5
```

---

7. Discuss about Heap Data structure or Max-heap. Write a python program to build a Max-Heap data structure.

Max-heap

Always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property**.**



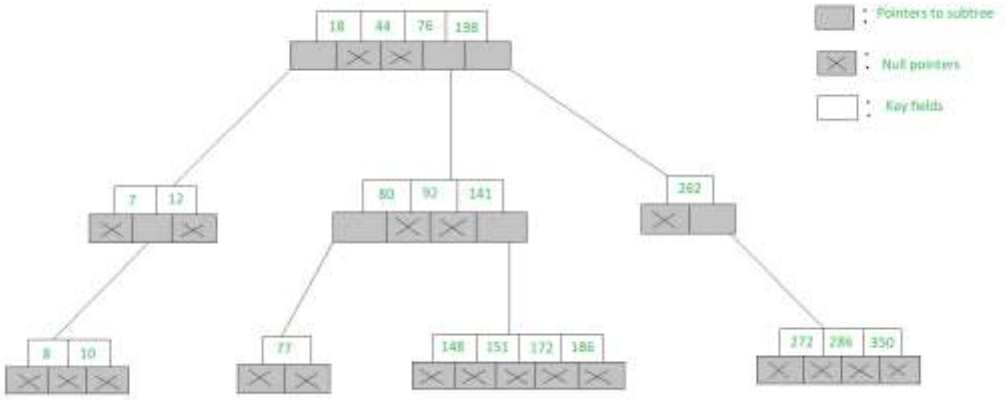

```
# Max-Heap data structure in Python

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]
        heapify(arr, n, largest)

def insert(array, newNum):
    size = len(array)
    if size == 0:
        array.append(newNum)
    else:
        array.append(newNum);
        for i in range((size//2)-1, -1, -1):
            heapify(array, size, i)

def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]:
            break
```

```
    array[i], array[size-1] = array[size-1], array[i]

    array.remove(num)

    for i in range((len(array)//2)-1, -1, -1):
        heapify(array, len(array), i)

arr = []

insert(arr, 3)
insert(arr, 4)
insert(arr, 9)
insert(arr, 5)
insert(arr, 2)

print ("Max-Heap array: " + str(arr))

deleteNode(arr, 4)
print("After deleting an element: " + str(arr))
```

---

8. Discuss about multi -way search tree. Write a python program for Searching for a key in an m-Way search tree

Multi-way trees or m-Way tree are generalised versions of binary trees where each node contains multiple elements. In an m-Way tree of order m, each node contains a maximum of m – 1 elements and m children



- To search for 77 in the 5-Way search tree, shown in the figure, we begin at the root & as 77> 76> 44> 18, move to the fourth sub-tree

- In the root node of the fourth sub-tree, 77< 80 & therefore we move to the first sub-tree of the node. Since 77 is available in the only node of this sub-tree, we claim 77 was successfully searched

```
# Searches value in the node
def search(val, root, pos):

        # if root is None then return
        if (root == None):
                return None
        else :
                # if node is found
                if (searchnode(val, root, pos)):
                        return root

                # if not then search in child nodes
                else:
                        return search(val, root.child[pos], pos)

# Searches the node
def searchnode(val, n, pos):
        # if val is less than node.value[1]
        if (val < n.value[1]):
                pos = 0
                return 0


        # if the val is greater
        else :
                pos = n.count

                # check in the child array for correct position
                while ((val < n.value[pos]) and pos > 1):
                        pos-=1
                if (val == n.value[pos]):
                        return 1
                else:
                        return 0
```

search():

- The function **search()** receives three parameters
- The first parameter is the value to be searched, second is the address of the node from where the search is to be performed and third is the address of a variable that is used to store the position of the value once found
- Initially a condition is checked whether the address of the node being searched is NULL
- If it is, then simply a NULL value is returned
- Otherwise, a function **searchnode()** is called which actually searches the given value
- If the search is successful the address of the node in which the value is found is returned

- If the search is unsuccessful then a recursive call is made to the **search()** function for the child of the current node

searchnode():

- The function **searchnode()** receives three parameters
- The first parameter is the value that is to be searched
- The second parameter is the address of the node in which the search is to be performed and third is a pointer **pos** that holds the address of a variable in which the position of the value that once found is stored
- This function returns a value 0 if the search is unsuccessful and 1 if it is successful
- In this function initially it is checked whether the value that is to be searched is less than the very first value of the node
- If it is then it indicates that the value is not present in the current node. Hence, a value 0 is assigned in the variable that is pointed to by **pos** and 0 is returned, as the search is unsuccessful

---

9. Write a python program to implement min-heap using heapq class.

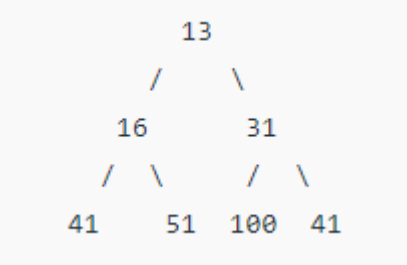
A Min Heap is a Complete Binary Tree. A Min heap is typically represented as an array. The root element will be at **Arr[0]**. For any ith node, i.e., **Arr[i]**:

- **Arr[(i -1) / 2]** returns its parent node.
- **Arr[(2 * i) + 1]** returns its left child node.
- **Arr[(2 * i) + 2]** returns its right child node.

MinHeap operations

1. heappush: Finding the exact position of the new element and insert the new element.
2. heappop: After the minimum element is removed, the heap has to put the new root in place..

Example:




```
from heapq import heapify, heappush, heappop

heap = []   # Creating empty heap
heapify(heap)

heappush(heap, 10)  # Adding items to the heap using heappush function
heappush(heap, 30)
```

```
heappush(heap, 20)
heappush(heap, 400)

print("Head value of heap : "+str(heap[0]))  # printing the value of minimum element
print("The heap elements : ") # printing the elements of the heap
for i in heap:
  print(i, end = ' ')
print("\n")

element = heappop(heap)

print("The heap elements : ")  # printing the elements of the heap
for i in heap:
  print(i, end = ' ')
```

---

10. Write an efficient program for printing k largest elements in an array. Elements in an array can be in any order. For example, if the given array is [1, 23, 12, 9, 30, 2, 50] and you are asked for the largest 3 elements i.e., k = 3 then your program should print 50, 30, and 23.

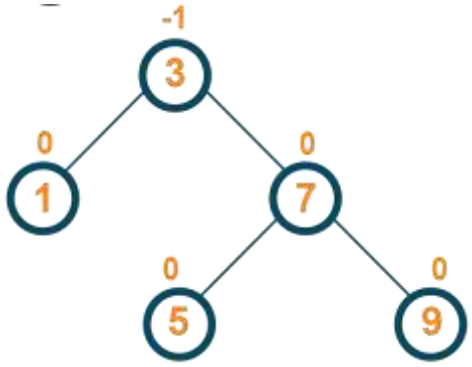
```
''' Python3 code for k largest elements in an array'''


def kLargest(arr, k):

        # Sort the given array arr in reverse  order.

        arr.sort(reverse = True)

        # Print the first kth largest elements

        for i in range(k):

                print (arr[i], end =" ")


# Driver program

arr = [1, 23, 12, 9, 30, 2, 50]

# n = len(arr)

k = 3

kLargest(arr, k)
```

---

11. Discuss about AVL tree and its type of rotation?

AVL tree is a height-balanced tree where the difference between the heights of the right subtree and left subtree of every node is either -1, 0 or 1

Balance Factor = Height Of Left Subtree - Height Of Right Subtree



AVL Rotation

Rotation is the method of moving the nodes of trees either to left or to right to make the tree heighted balance tree.

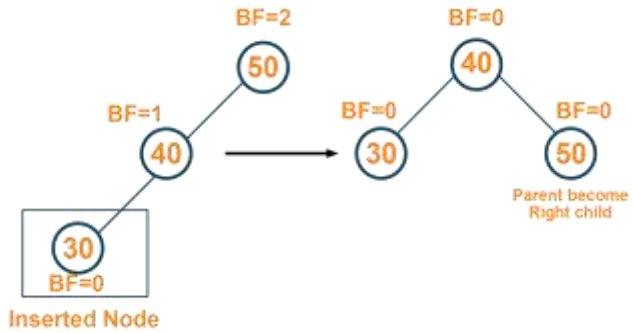
There are total two categories of rotation which is further divided into two further parts:

1) Single Rotation

Single rotation switches the roles of the parent and child while maintaining the search order. We rotate the node and its child, the child becomes a parent.

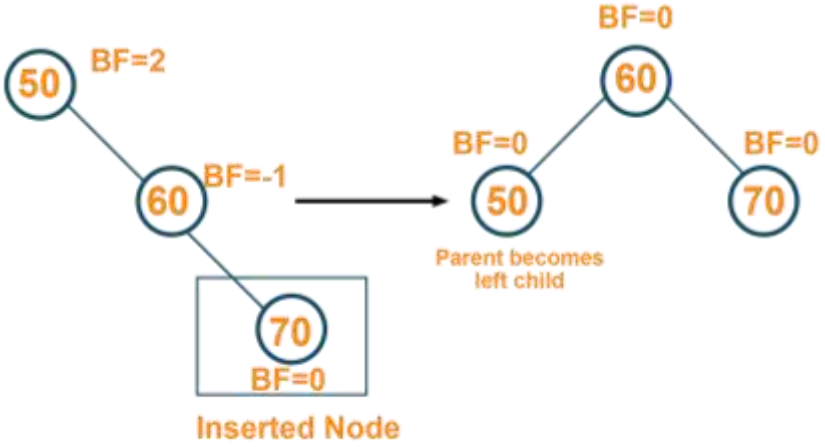
Single LL(Left Left) Rotation

Here, every node of the tree moves towards the left from its current position. Therefore, a parent becomes the right child in LL rotation. Let us see the below examples



Single RR(Right Right) Rotation

Here, every node of the tree moves towards the right from the current position. Therefore, the parent becomes a left child in RR rotation. Let us see the below example

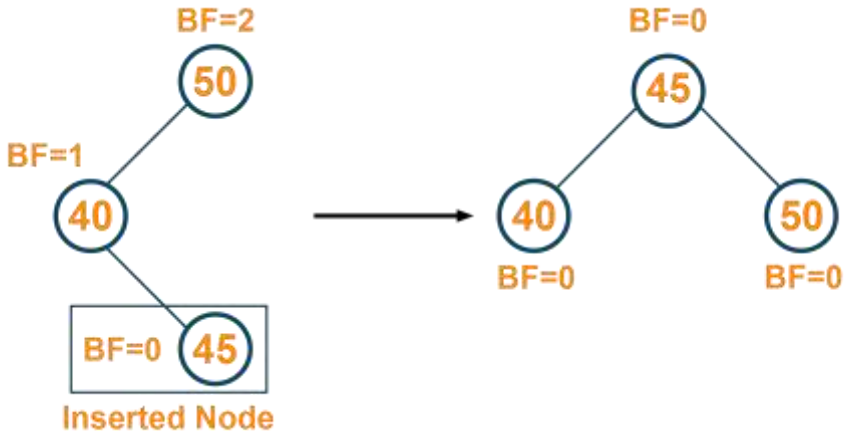


2) Double Rotation

Single rotation does not fix the LR rotation and RL rotation. For this, we require double rotation involving three nodes. Therefore, double rotation is equivalent to the sequence of two single rotations.

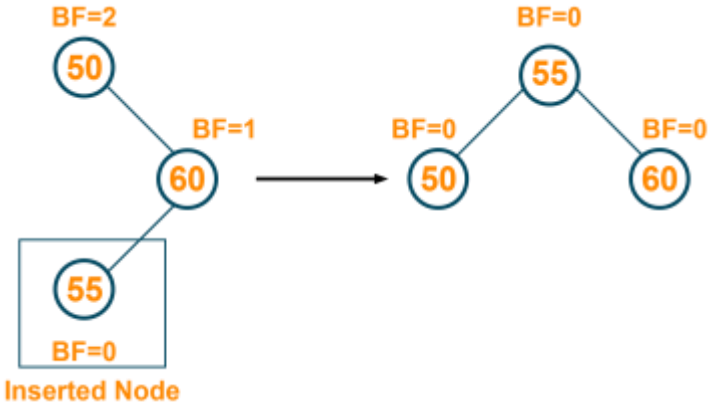
LR(Left-Right) Rotation

The LR rotation is the process where we perform a single left rotation followed by a single right rotation. Therefore, first, every node moves towards the left and then the node of this new tree moves one position towards the right. Let us see the below example



RL (Right-Left) Rotation

The RL rotation is the process where we perform a single right rotation followed by a single left rotation. Therefore, first, every node moves towards the right and then the node of this new tree moves one position towards the left. Let us see the below example



---

12. Discuss about Operations In AVL Tree.

There are 2 major operations performed on the AVL tree

1. Insertion Operation
2. Deletion Operation

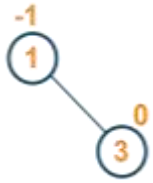
Insertion Operation in AVL Tree

In the AVL tree, the new node is always added as a leaf node. After the insertion of the new node, it is necessary to modify the balance factor of each node in the AVL tree using the rotation operations. The algorithm steps of insertion operation in an AVL tree are:

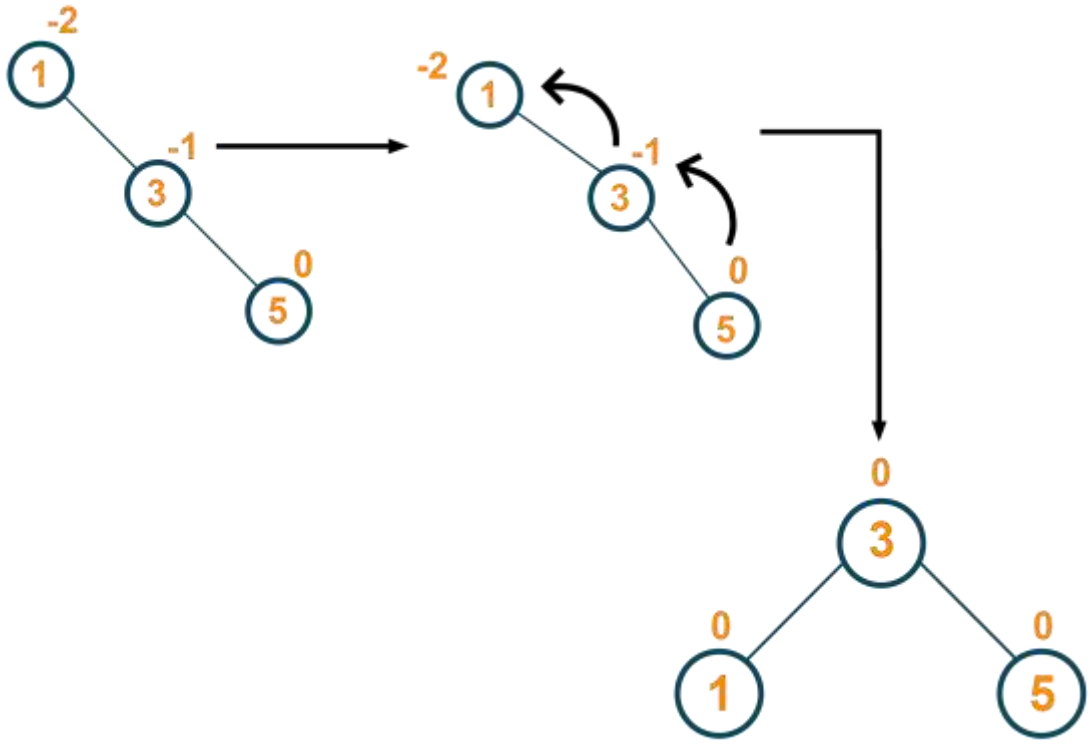
1. Find the appropriate empty subtree where the new value should be added by comparing the values in the tree
2. Create a new node at the empty subtree
3. The new node is a leaf ad thus will have a balance factor of zero
4. Return to the parent node and adjust the balance factor of each node through the rotation process and continue it until we are back at the root. Remember that the modification of the balance factor must happen in a bottom-up fashion

The root node is added as shown in the below figure

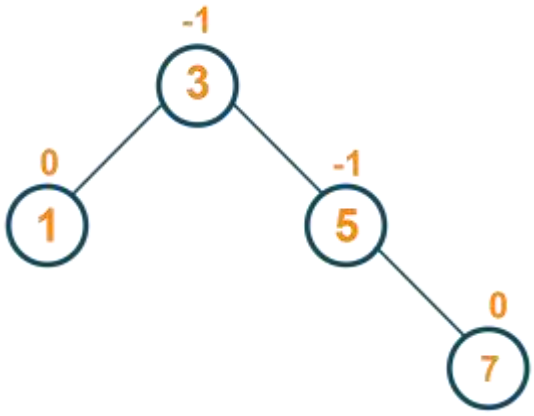
The node to the root node is added as shown below. Here the tree is balanced



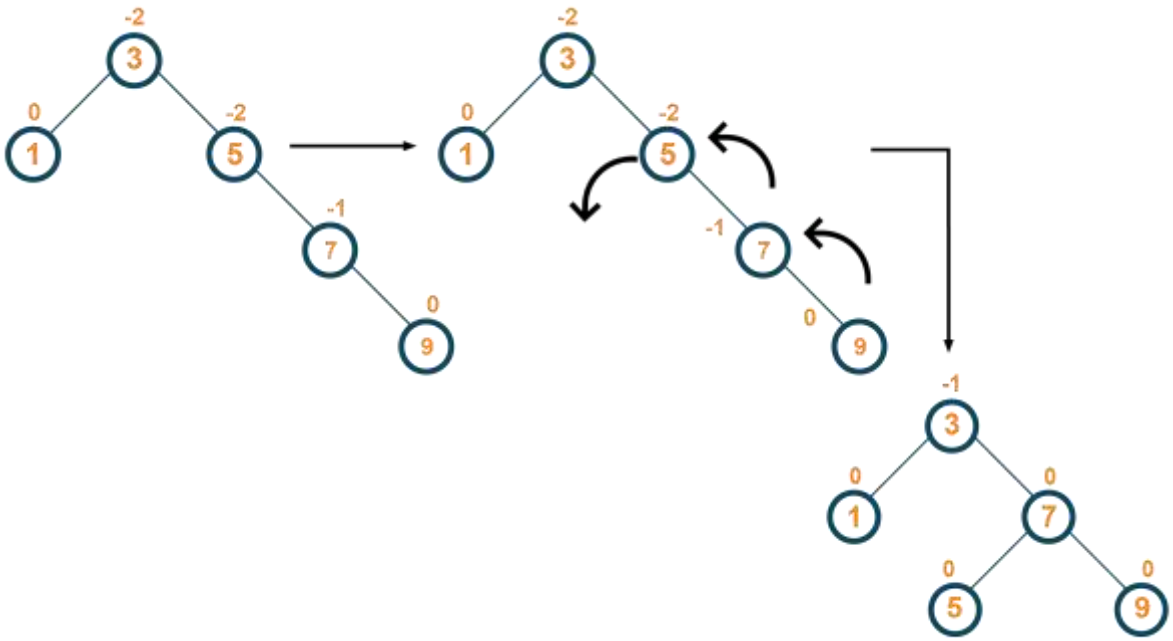
Then, The right child is added to the parent node. Here, the balance factor of the tree is changed, therefore, the LL rotation is performed and the tree becomes a balanced tree



Later, one more right child is added to the new tree as shown below



Again further, one more right child is added and the balance factor of the tree is changed. Therefore, again LL rotation is performed on the tree and the balance factor of the tree is restored as shown in the below figure



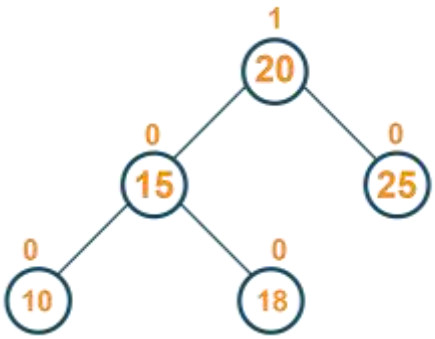
Deletion Operation in AVL

The deletion operation in the AVL tree is the same as the deletion operation in BST. In the AVL tree, the node is always deleted as a leaf node and after the deletion of the node, the balance factor of each node is modified accordingly. Rotation operations are used to modify the balance factor of each node. The algorithm steps of deletion operation in an AVL tree are:

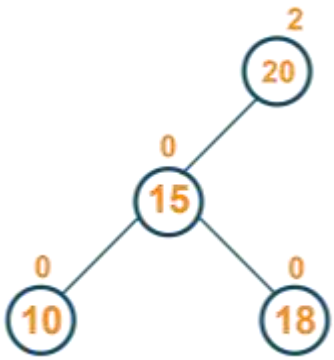
1. Locate the node to be deleted
2. If the node does not have any child, then remove the node
3. If the node has one child node, replace the content of the deletion node with the child node and remove the node
4. If the node has two children nodes, find the inorder successor node ‘k' which has no child node and replace the contents of the deletion node with the ‘k’ followed by removing the node.
5. Update the balance factor of the AVL tree

Example:

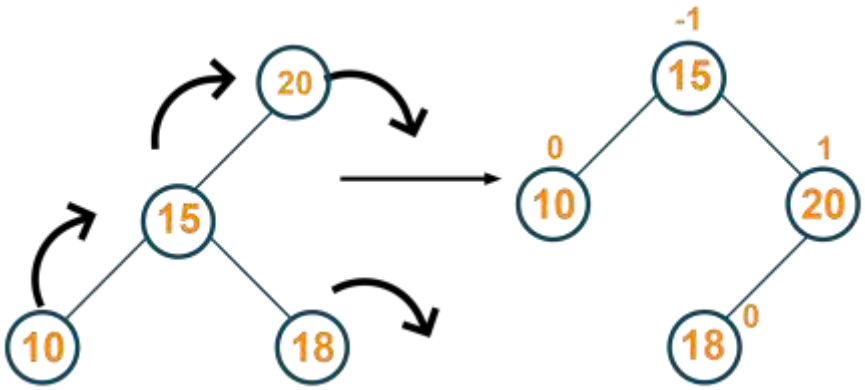
Let us consider the below AVL tree with the given balance factor as shown in the figure below



Here, we have to delete the node '25' from the tree. As the node to be deleted does not have any child node, we will simply remove the node from the tree



After removal of the tree, the balance factor of the tree is changed and therefore, the rotation is performed to restore the balance factor of the tree and create the perfectly balanced tree

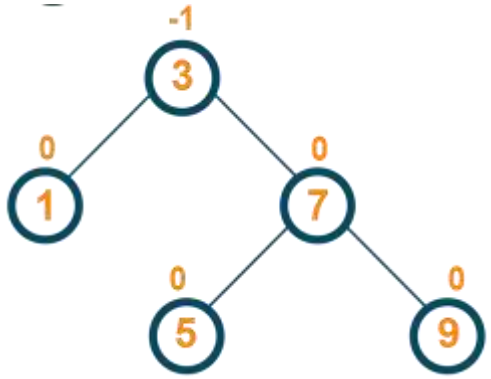


---

13. Write a python for AVL tree.

AVL tree is a height-balanced tree where the difference between the heights of the right subtree and left subtree of every node is either -1, 0 or 1

Balance Factor = Height Of Left Subtree - Height Of Right Subtree




```
class treeNode(object):
        def __init__(self, value):
                self.value = value
                self.l = None
                self.r = None
                self.h = 1

class AVLTree(object):

        def insert(self, root, key):

                if not root:
                        return treeNode(key)
                elif key < root.value:
                        root.l = self.insert(root.l, key)
                else:
                        root.r = self.insert(root.r, key)

                root.h = 1 + max(self.getHeight(root.l),
                                                self.getHeight(root.r))

                b = self.getBal(root)

                if b > 1 and key < root.l.value:
                        return self.rRotate(root)

                if b < -1 and key > root.r.value:
```

```
                return self.lRotate(root)

        if b > 1 and key > root.l.value:
                root.l = self.lRotate(root.l)
                return self.rRotate(root)

        if b < -1 and key < root.r.value:
                root.r = self.rRotate(root.r)
                return self.lRotate(root)

        return root

def lRotate(self, z):

        y = z.r
        T2 = y.l

        y.l = z
        z.r = T2

        z.h = 1 + max(self.getHeight(z.l),
                                        self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),
                                        self.getHeight(y.r))

        return y

def rRotate(self, z):

        y = z.l
        T3 = y.r

        y.r = z
        z.l = T3

        z.h = 1 + max(self.getHeight(z.l),
                                        self.getHeight(z.r))
        y.h = 1 + max(self.getHeight(y.l),
                                        self.getHeight(y.r))

        return y

def getHeight(self, root):
        if not root:
                return 0

        return root.h

def getBal(self, root):
        if not root:
```

```
                return 0

            return self.getHeight(root.l) - self.getHeight(root.r)

        def preOrder(self, root):

            if not root:
                return

            print("{0} ".format(root.value), end="")
            self.preOrder(root.l)
            self.preOrder(root.r)

Tree = AVLTree()
root = None

root = Tree.insert(root, 1)
root = Tree.insert(root, 2)
root = Tree.insert(root, 3)
root = Tree.insert(root, 4)
root = Tree.insert(root, 5)
root = Tree.insert(root, 6)

# Preorder Traversal
print("Preorder traversal of the",
        "constructed AVL tree is")
Tree.preOrder(root)
print()
```

References:

1. Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, “Data Structures and Algorithms in Python” (An Indian Adaptation), Wiley, 2021
2. https://www.educative.io/answers/heap-implementation-in-python
3. https://www.geeksforgeeks.org/min-heap-in-python/
4. https://favtutor.com/blogs/avl-tree-python