log n? How about n log log n?

Proving upper bound for n log log n :

$$
\begin{aligned}
T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\
&\le \sqrt{n}.c. \sqrt{n}\sqrt{n}\ \log\log\sqrt{n} + \sqrt{n} + n \\
&= n.\ c.\log\log n - c.\ n + n \\
&\le c\ n \log\log n,\ \text{if } c \ge 1
\end{aligned}
$$

Proving lower bound for n log log n:

$$
\begin{aligned}
T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\
&\ge \sqrt{n}.\ k.\ \log\log\sqrt{n} + n \\
&= n.\ k.\log\log n - k.\ n + n \\
&\ge k\ n \log\log n,\ \text{if } k \le 1
\end{aligned}
$$

From the above proofs, we can see that $T(n) \le c\ n \log\log n$, if $c \ge 1$ and $T(n) \ge k\ n \log\log n$, if $k \le 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $T(n) = \Theta(n \log\log n)$.

Review Questions

Q. 1 What is algorithm ? **(Refer section 1.1.1)**

Q. 2 List and explain characteristics of an algorithm. **(Refer section 1.1.2)**

Q. 3 List and explain various asymptotic notation **(Refer section 1.8).**

Q. 4 Explain different performance characteristics of algorithm. **(Refer sections 1.12, 1.12.1 and 1.12.2)**

□□□

UNIT II

CHAPTER 2 Tree Algorithms

Syllabus

What is a Tree? Glossary, Binary Trees, Types of Binary Trees, Properties of Binary Trees, Binary Tree Traversals, Generic Trees (N-ary Trees), Threaded Binary Tree Traversals, Expression Trees, Binary Search Trees (BSTs), Balanced Binary Search Trees, AVL (Adelson-Velskii and Landis) Trees.

Syllabus Topic : What is a Tree ?

2.1 Tree

- In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order.
- Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows.
- "Tree is a non-linear data structure which organizes data in hierarchical structure."
- Tree represents the nodes connected by edges.

Syllabus Topic : Glossary

2.1.1 Glossary of Tree

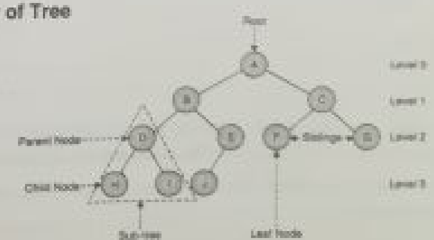


Fig. 2.1.1

Root

- In a tree data structure, the first node is called as **Root Node**. Every tree must have root node.
- Root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

Edge

- An edge is another fundamental part of a tree. In a tree data structure, the connecting link between any two nodes is called as **EDGE**.
- In a tree with 'N' number of nodes there will be a maximum of **'N-1'** number of edges.

Node

Each element in the hierarchical representation is called a node. In the above tree A, B, C, D, E, F, G, H, I, J, K all are nodes.

Path

- Path between any two nodes in a tree is a sequence of distinct nodes in which successive nodes are connected by edges.
- **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.

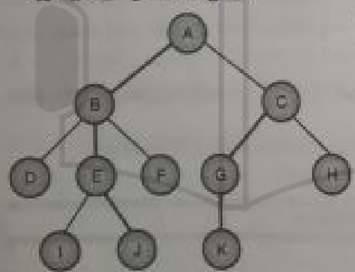


Fig. 2.1.2

- In any tree, 'Path' is a sequence of nodes and edges between to nodes.
- Here, 'path' between A and J is A – B – E– J.
- Here, 'path' between C and K is C – G – K.

Parent

- In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. Parent node can also be defined as **"The node which has child / children"**.
- The root node is the only node which does not have a parent. In the above tree, nodes A, B, C, D, E are parent nodes.

Child

- The immediate successor of a node is called as **CHILD Node**.
- In a tree, any parent node can have any number of child nodes. In the above tree, D and E are childrens of node B.

Siblings

- Children of the same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.
- In the above tree, F and G are siblings, also D and E are siblings.

Subtree

A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.

Leaf node

- The nodes which does not have any child is called as LEAF Node.
- Leaf nodes are also called as terminal nodes.
- In the above tree, F, G, H, I, J are the leaf nodes.

Internal nodes

- A node of a tree that has one or more child is called as **INTERNAL Node**.
- **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as **'Non-Terminal'** nodes.

External nodes

The leaf nodes are also called as **External Nodes**. In the above tree, F, G, H, I, J are the external nodes.

Degree of a node

- In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has.
  - In above tree degree of a node A is 2.
  - In above tree degree of a node B is 2.
  - In above tree degree of a node C is 2.
  - In above tree degree of a node E is 1.

Degree of a tree

The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'. The degree of above tree is 2.

Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on.

Height or Depth of a node

- The height of any node is the length of longest path from that node to a terminal node is called as **HEIGHT** of that Node.
- In a tree, height of the root node is said to be **height of the tree.** In a tree, height of all **leaf nodes is '0'.**

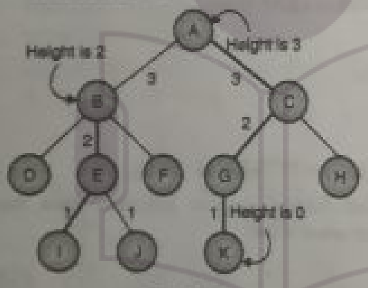


Fig. 2.1.3

- Here, Height of tree is 3
- In any tree, 'Height of Node' is total number of edges from leaf to that node in longest path.
- In any tree, 'Height of tree' is the height of the root node.

Depth of a tree

- The total number of edges from root node to a leaf node at the last level is said to be Depth of the tree.
- Depth or Height is same for the tree but the difference between these is height is always measured from leaf node to root node while depth is measured from root node to leaf node.

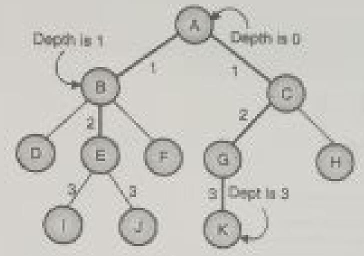


Fig. 2.1.4

- Here, Depth of tree is 3
- In any tree, 'Depth of Node' is total number of edges from edges from root to that node.
- In any tree, 'depth of tree' is total number of edges from root to leaf in the longest path.

Syllabus Topic : Binary Trees

2.2 Binary Trees

- Binary tree is defined on a finite set of nodes that either:

  a. Contains no nodes.

  b. Composed of three disjoint set of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.

- "A tree in which every node can have a maximum of two children is called as Binary Tree."
- In a binary tree have the property that it can have either 0, 1 or 2 children but not more than 2 children. A binary tree may be empty known as **Null Tree.**

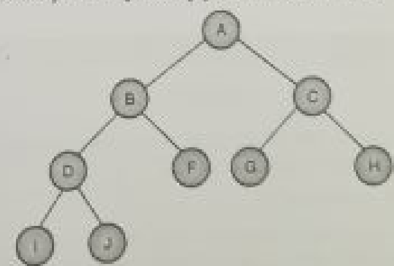


Fig. 2.2.1

How to represent a binary tree ?

A binary tree data structure is represented using two methods. Those methods are as follows.

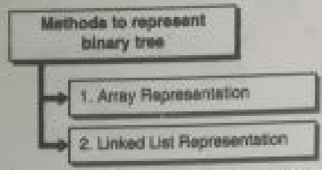


Fig. C2.1 : Methods of representation of binary tree

Consider the following binary tree.

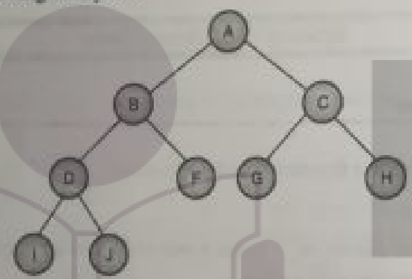


Fig. 2.2.2

1. Array Representation

- In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows.

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

Fig. 2.2.3

- To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

2. Linked List Representation

- We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

- In this linked list representation, a node has the following structure.

Left Child Address	Data	Right Child Address

Fig. 2.2.4

- The above example of binary tree represented using Linked list representation is shown as follows.

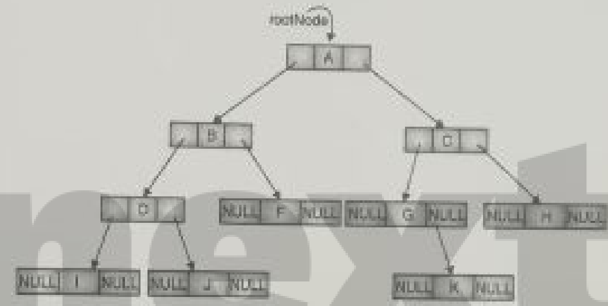


Fig. 2.2.5

Code to define node of Binary tree

- We need to represent a tree node. To do that, we create a new class named Node with 3 attributes.
  - Left node
  - Right node
  - Node's data

```
class Node(object)
#Tree node: left and right child + data which can be any object.
        def __init__(self, data):
        #Node Constructor
        #@param data node data object
        self.left = None
        self.right = None
        self.data = data
```

2.2.1 Implementation of Binary Tree Class and its Methods

```
class BinaryTree():
def __init__(self,root):
self.left = None
self.right = None
self.root = rootid
def getLeftChild(self):
returnself.left
def getRightChild(self):
returnself.right
defsetNodeValue(self,value):
self. rootid = value
def getNodeValue(self):
returnself.rootid
def insertRight(self,newNode):
ifself.right == None:
self.right = BinaryTree(newNode)
else:
tree = BinaryTree(newNode)
tree.right = self.right
self.right = tree
def insertLeft(self,newNode):
ifself.left == None:
self.left = BinaryTree(newNode)
else:
tree = BinaryTree(newNode)
tree.left = self.left
self.left = tree
def printTree(tree):
if tree != None:
printTree(tree.getLeftChild())
print(tree.getNodeValue())
printTree(tree.getRightChild())
```



```
# test tree
deftestTree():
myTree = BinaryTree("Maud")
myTree.insertLeft("Bob")
myTree.insertRight("Tony")
myTree.insertRight("Steven")
printTree(myTree)
```

2.2.2 Applications of Binary Trees

- Expression trees are used in compilers.
- Huffman coding tree that are used in data compression algorithms.
- Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in O(log n) (average).
- Priority Queue (PQ), which support, search and deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

Syllabus Topic : Types of Binary Trees

2.3 Types of Binary Trees

There are different types of binary trees :

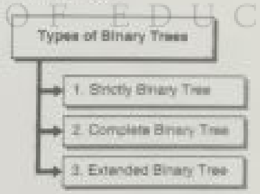


Fig. C2.2 : Types of Binary Trees

1. Strictly Binary Tree

- A binary tree is a strictly binary tree in which every internal node should have exactly two children or none.
- Strictly binary tree is used to represent algebraic expressions where non-leaf nodes represent operators and leaf nodes represent operands.
- Strictly binary tree data structure is used to represent mathematical expressions.

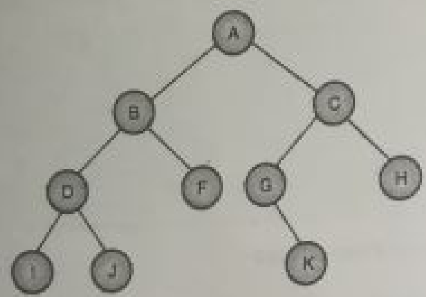


Fig. 2.3.1

➜ 2. Complete Binary Tree

- A binary tree in which every internal node has exactly two children that means all internal nodes have degree 2 and all leaf nodes are at same level is called Complete Binary Tree.
- It is also termed as Perfect Binary Tree.
- In complete binary tree, if there are n nodes at level l then at level i+1, there are 2^i nodes.

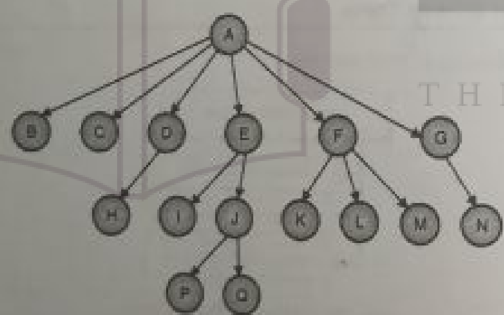


Fig. 2.3.2

➜ 3. Extended Binary Tree

- An extended binary tree is a transformation of any binary tree into a complete binary tree. This transformation consists of replacing every null subtree of the original tree with "special nodes". These special nodes are called as dummy nodes.
- The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

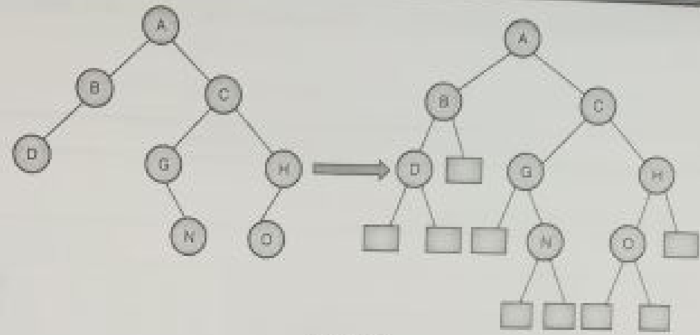


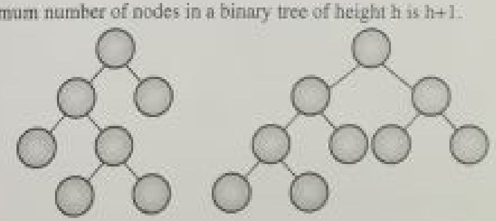
Fig. 2.3.3

- In above tree the nodes which are represented by rectangle are the dummy nodes.

Syllabus Topic : Properties of Binary Trees

2.4 Properties of Binary Trees

- A binary tree with n nodes has exactly n – 1 nodes.
- In a binary tree every node except the root node has exactly one parent.
- In a binary tree, there is exactly one path connecting any two nodes in the tree.
- The maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$. For example, number of nodes in a binary tree of height 4 will be $2^{4+1} - 1 = 31$.
- The minimum number of nodes in a binary tree of height h is h+1.



(a) Full tree (b) Complete tree

Fig. 2.4.1

- Number of leaf nodes in a complete binary tree is (n + 1)/2.
- In a complete binary tree, Number of external nodes = Number of internal nodes + 1.

- In a complete binary tree, if there are n nodes at level 'l' then at level 'l+1', there are 2n nodes.
- A full binary tree is a binary tree in which each node has exactly zero or two children.

Syllabus Topic : Binary Trees Traversals

2.5 Binary Trees Traversals

Traversal is a process to visit all the nodes of a tree. There are three types of binary tree traversals.

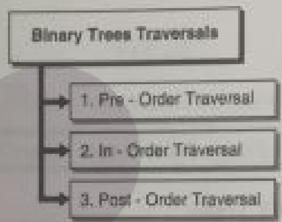


Fig. C2.3 : Types of binary tree traversals

Consider the following binary tree.

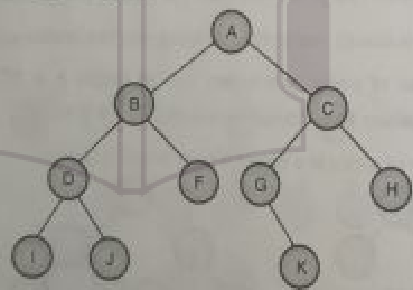


Fig. 2.5.1

Let's explore each tree traversal method one by one.

2.5.1 Pre - Order Traversal

- This traversal is also known as depth-first order.
- In this traversal technique the traversal order is root-left-right i.e.
  - Process data of root node

  - First, traverse left subtree completely
  - Then, traverse right subtree.
- In the above example if we perform the preorder traversal then first we start from its root which is A. Next we move to the left subtree of A, so we visit left child B. Again B has D and F as its left and right child reply, so first we visit left child D.
- Again D has I and J as its left and right child reply, so we visit its left child I which is the leftmost child and then we move to the right child J because I does not have any child. Next we visit the B's right child which is F. With this we have completed root and left parts of node A.
- Now we go for A's right child which is C so visit it. Again C has lest and right child, so move to the left child of C which is G. But G have only right child K, it does not have any left child. So we visit G's right child K. Next move to the right child of C because we have done with G. Visit C's right child H which is the right most child in the tree. So we stop the process.
- Pre-Order Traversal for above example :

  A - B - D - I - J - F - C - G - K – H

Code to implement pre-order traversal

(i) Recursive Preorder Traversal

```
#Pre-order recursive traversal. The nodes' values are appended to the result #list in traversal order
defpreorderRecursive(root, result):
if not root:
return
result.append(root.data)
preorderRecursive(root. left, result)
preorderRecursive(root.right, result)
```

Time Complexity: O(n). Space Complexity: O(n)

(ii) Non-Recursive Preorder Traversal

```
#Pre-order iterative traversal. The nodes' values are appended to the result list in traversal order
def preorder_iterative(root, result):
if not root:
return
stack= []
stack.append(root)
```

```
while stack:
node =stack.pop()
result.append(node.data)
ifnode.right: stack.append(node.right)
ifnode.left: stack.append(node.left)
```

Time Complexity: O(n). Space Complexity: O(n).

2.5.2 In - Order Traversal

- This traversal is also known as depth-first order.
- In this traversal technique the traversal order is left-root-right i.e.
  - First process left subtree (before processing root node)
  - Then, process current root node
  - Process right subtree.
- In the above example of binary tree, first we start from the left child of root node 'A' which is B but again B has left child D, so we move to D. Again D has left child I so we visit I because this is the leftmost child in a tree (it does not have any child).
- Then visit its root node D and then visit its right child J. With this we have completed the left part of node B. Then Visit B. Next move to B's right child which is F, visit F. With this we have done with A's left part.
- Now visit A and move to its right part. A has C as its right child. Again C have left child G and G have only right child K.G does not have any left child, so visit G. then visit K. with this we have done with left part of C.
- Then visit C and move to the right part of C. C has right child H which is the rightmost child so visit H and stop the process.
- In-Order Traversal for above example is :

  I - D - J - B - F - A - G - K - C – H

Code to Implement In-order traversal

(I) Recursive inorder Traversal

```
# In-order recursive traversal. The nodes' values are appended to the result #list in traversal order
def inorderRecursive(root, result):
if not root:
return
inorderRecursive(root. left, result)
result.append(root.data)
inorderRecursive(root.right, result)
```


Time Complexity : O(n). Space Complexity: O(n).

(ii) Non-Recursive inorder Traversal

```
# In-order iterative traversal. The nodes' values are appended to the result list in traversal order
def inorderiterative(root, result):
if not root:
return
stack= []
node= root
while stack or node:
if node:
stack.append(node)
node = node.left
else:
node= stack.pop()
result.append(node.data)
node = node.right
```

Time Complexity: O(n). Space Complexity: O(n).

2.5.3 Post - Order Traversal

- In this traversal technique the traversal order is left-right-root.
  - Process data of left subtree
  - First, traverse right subtree
  - Then, traverse root node.
- In post-order traversal, the left subtree is traversed first in post-order traversal then the right subtree is traversed in post-order traversal and then root is traversed.
- In the above example, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. So we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J.
- So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for itsright child 'J'and later we visit root node 'D'. With this we have completed the left part of node B.
- Then visit B's right child 'F' and then visit 'B'. With this we have completed left part of node A. With this we have completed left parts of root node A. Then we go for right part of the node A. In right of A again there is a subtree with root C.

- So go for left child of C and again it is a subtree with root G. But G does not have left part but it has right child so first visit that right child 'K' and then visit node 'G'. With this we have completed the left part of node C.
- Then visit C's right child 'H' which is the right most child in the tree and then visit node 'C'. And finally visit the root node 'A' so we stop the process.
- Post-Order Traversal for above example binary tree is :

  I - J - D - F - B - K - G - H - C - A

Code to implement post-order traversal

(I) Recursive postorder Traversal

```
# Post-order recursive traversal. The nodes' values are appended to the result #list in traversal order
def postorderRecursive(root, result):
if not root:
return
postorderRecursive(root.left, result)
postorderRecursive(root. right, result)
result.append(root.data)
```

Time Complexity: O(n). Space Complexity: O(n).

(II) Non-Recursive postorder Traversal

```
# Post-order iterative traversal. The nodes' values are appended to the result #list in traversal order
def postorderIterative(root, result):
if not root:
return
visited = set()
stack = []
node= root
while stack or node:
if node:
stack.append( node)
node = node.left
else:
node = stack. pop()
if node.right and not node.right in visited:
```



```
stack.append(node)
node = node.right
else:
visited.add(node)
result.append(node.data)
node= None
```

Time Complexity: O(n). Space Complexity: O(n).

Program 2.5.1

Write a program Python program for tree traversals.

Solution :

Python program for tree traversals

```
# A class that represents an individual node in a Binary Tree
class Node:
  def _init_(self,key):
      self.left = None
      self.right = None
      self.val = key
# A function to do inorder tree traversal
def printInorder(root):
 if root:
      # First recur on left child
      printInorder(root.left)
      # then print the data of node
      print(root.val),
      # now recur on right child
      printInorder(root.right)
# A function to do postorder tree traversal
def printPostorder(root):
 if root:
      # First recur on left child
      printPostorder(root.left)
      # the recur on right child
      printPostorder(root.right)
```

```
        # now print the data of node
        print(root.val),
# A function to do postorder tree traversal
def printPreorder(root):
    if root:
        # First print the data of node
        print(root.val),
        # Then recur on left child
        printPreorder(root.left)
        # Finally recur on right child
        printPreorder(root.right)
# Driver code
root = Node(1)
root.left   = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print "Preorder traversal of binary tree is"
printPreorder(root)
print "\nInorder traversal of binary tree is"
printInorder(root)
print "\nPostorder traversal of binary tree is"
printPostorder(root)
```

Syllabus Topic : Generic Trees(N-ary Trees)

2.6 Generic Trees (N-ary Trees)

In the previous section we discussed binary trees where each node can have a maximum of two children and these are represented easily with two pointers. But suppose if we have a tree with many children at every node and also if we do not know how many children a node can have, how do we represent them?

For example, consider the tree shown in Fig. 2.6.1

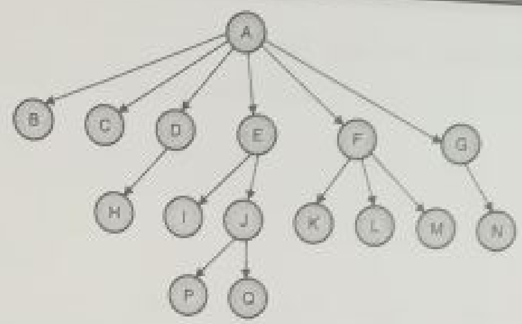


Fig. 2.6.1

Representation of tree

- In the above tree, there are nodes with 6 children, with 3 children, with 2 children, with 1 child, and with zero children (leaves).
- To present this tree we have to consider the worst case (6 children) and allocate that many child pointers for each node.

Based on this, the node representation can be given as follows.

```
#Node of a Generic Tree
class TreeNode:
#constructor
def _init_(self, data=None, next=None):
self.data= data
self.firstChild None
self.secondChild = None
self.thirdChild None
self.fourthChild = None
self.fifthChild = None
self.sixthChild = None
```

- Since we are not using all the pointers in all the cases, there is a lot of memory wastage.
- Another problem is that we do not know the number of children for each node in advance.
- In order to solve this problem we need a representation that minimizes the wastage and also accepts nodes with any number of children.

2.6.1 Representation of Generic Trees

- Since our objective is to reach all nodes of the tree, a possible solution to this is as follows :
  - At each node link children of same parent (siblings) from left to right.
  - Remove the links from parent to all children except the first child.

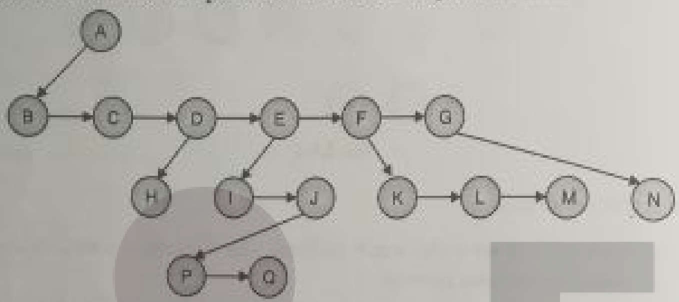


Fig. 2.6.2

- What these above statements say is if we have a link between children then we do n[ot] need extra links from parent to all children.
- This is because we can traverse all the elements by starting at the first child of the paren[t]. So if we have a link between parent and first child and also links between all children o[f] same parent then it solves our problem.
- This representation is sometimes called first child/next sibling representation. Fi[rst] child/next sibling representation of the generic tree is shown above. The actu[al] representation for this tree is shown in Fig. 2.6.3.

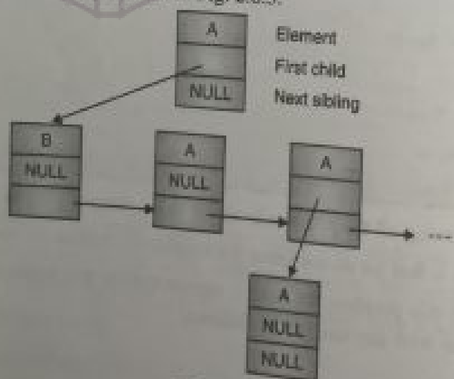


Fig. 2.6.3

- Based on this discussion, the tree node declaration for general tree can be given as :

```
#Node of a Generic Tree
classTreeNode:
#constructor
definit_ (self, data=None, next .. None):
self.data= data
sdf.firstChild = None
self.nextSibling .. None
```

Syllabus Topic : Threaded Binary Tree Traversals

2.7 Threaded Binary Tree Traversals

- The basic difference between a Binary tree and The Threaded Binary tree is that in Binary trees the nodes are null if there is no left or right child associated with it.
- That means in the linked list representation of binary tree, each node contains two pointers.
- The first pointer points to the left child and second pointer points to the right child but most of pointers are NULL because of the absence of left and right child and so there is no way to traverse back.
- So binary trees have a lot of wasted space. But in threaded binary tree we have threads associated with the nodes.
- That means they either are linked to the predecessor or successor in the inorder traversal of the nodes.
- This helps us to traverse further or backward in the inorder traversal fashion.
- "Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor."

Note : If there is no in-order predecessor or in-order successor, then it point to root node.

Why do we need Threaded Binary Tree ?

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.
- Consider the following binary tree (Fig. 2.7.1).

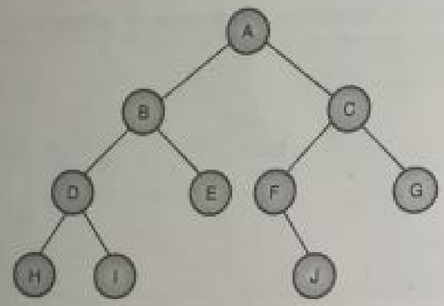


Fig. 2.7.1

- To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree.
- In-order traversal of above binary tree.

  H - D - I - B - E - A - F - J - C - G
- Above example binary tree become as follows after converting into threaded binary tree.

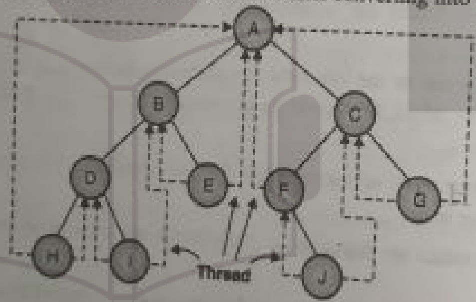


Fig. 2.7.2

In Fig. 2.7.2, threads are indicated with dotted links.

2.7.1 Types of Threaded Binary Trees

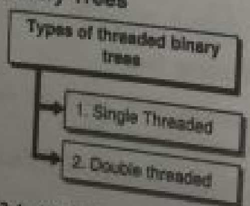


Fig. C2.4 : Types of threaded binary trees

➜ 1. Single Threaded

Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

➜ 2. Double threaded

Each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor AND all left null pointers will point to inorder predecessor.

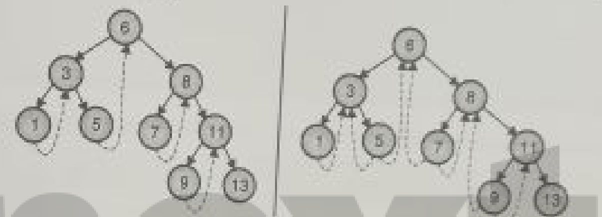


Fig. 2.7.3

2.7.2 Predecessor and Successor

When you do the inorder traversal of a binary tree, the neighbors of given node are called **Predecessor** (the node lies behind of given node) and **Successor** (the node lies ahead of given node).

☞ **Example**

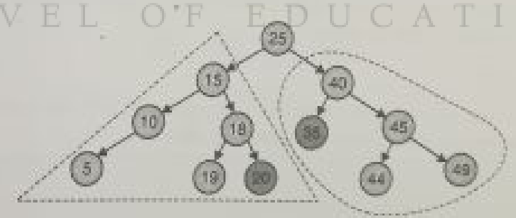


Fig. 2.7.4

2.7.3 Threaded Binary Tree Structure

- Any program examining the tree must be able to differentiate between a regular left/right pointer and a thread.
- To do this, we use two additional fields in each node, giving us, for threaded trees, nodes of the following form :

Left	LTag	Data	RTag	Right

Fig. 2.7.5

```
#Threaded Binary Tree Class and its methods
class ThreadedBinaryTree:
def _init_(self, data):
self.data = data                #data
self.left= None                 #left child
self.LTag = None
self.right= None                #right child
self.RTag =None
```

2.7.4 Difference between Binary Tree and Threaded Binary Tree Structures

	Regular Binary Tree	Threaded Binary Tree
If LTag = 0	NULL	Left points to the inorder predecessor
If LTag = 1	Left points to the left child	Left points to the left child
If RTag = 0	NULL	Right points to the inorder successor
If RTag = 1	Right points to the right child	Right points to the right child

2.7.5 Finding Inorder Successor in Inorder Threaded Binary Tree

To find inorder successor of a given node without using a stack, assume that the node for which we want to find the inorder successor is P.

Strategy

If P has a no right subtree, then return the light child of P. If P has right subtree, then return the left of the nearest node whose left subtree contains P.

```
def InorderSuccessor(P):
    if(P.RTag == 0):
    else:
    return P.right
    Position = P.right
    while(Position.LTag == 1):
    Position = Position.left
    return Position
```


2.7.6 Inorder Traversal in Inorder Threaded Binary Tree

We can start with dummy node and call inorderSuccessor() to visit each node until we reach dummy node.

```
def InorderTraversal(root):
P = InorderSuccessor(root)
while(P != root):
P = InorderSuccessor(P)
printP.data
```

Syllabus Topic : Expression Trees

2.8 Expression Tree

- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.
- Consider the expression : $((3+5)\times(5+9))$
- It can be represented as a binary tree.

Fig. 2.8.1

- To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps :
  1. Read all the symbols one by one from left to right in the given Infix Expression.
  2. If the reading symbol is operand, then directly print it to the result (Output).
  3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
  4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
  5. If the reading symbol is operator (+ , - , * , / etc.,), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

☞ **Example**

A + (B * C – (D / E – F) * G) * H

Stack	Input	Output
Empty	A+(B*C-(D/E-F)*G)*H	-
Empty	+(B*C-(D/E-F)*G)*H	A
+	(B*C-(D/E-F)*G)*H	A
+(	B*C-(D/E-F)*G)*H	A
+(	*C-(D/E-F)*G)*H	AB
+(*	C-(D/E-F)*G)*H	AB
+(*	-(D/E-F)*G)*H	ABC
+(-	(D/E-F)*G)*H	ABC*
+(-(	D/E-F)*G)*H	ABC*
+(-(	/E-F)*G)*H	ABC*D
+(-(/	E-F)*G)*H	ABC*D
+(-(/	-F)*G)*H	ABC*DE
+(-(-	F)*G)*H	ABC*DE/
+(-(-	F)*G)*H	ABC*DE/
+(-(-	)*G)*H	ABC*DE/F
+(-	*G)*H	ABC*DE/F-
+(-*	G)*H	ABC*DE/F-
+(-*	)*H	ABC*DE/F-G
+	*H	ABC*DE/F-G*-
+*	H	ABC*DE/F-G*-
+*	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*+

Program 2.8.1

Write a program for Building Expression Tree from Postfix Expression.

Solution :

```
# An expression tree node
class Et:
    # Constructor to create a node
```



```
    def _init_(self , value):
        self.value = value
        self.left = None
        self.right = None
# A utility function to check if 'c' is an operator
def isOperator(c):
    if (c == '+' or c == '-' or c == '*'
            or c == '/' or c == '^'):
            return True
    else:
            return False
# A utility function to do inorder traversal
def inorder(t):
    if t is not None:
        inorder(t.left)
        print.value,
        inorder(t.right)
# Returns root of constructed tree for given postfix expression
def constructTree(postfix):
    stack = []
    # Traverse through every character of input expression
    for char in postfix :
        # if operand, simply push into stack
        if not isOperator(char):
            t = Et(char)
            stack.append(t)
        # Operator
        else:
            # Pop two top nodes
            t = Et(char)
            t1 = stack.pop()
            t2 = stack.pop()
            # make them children
            t.right = t1
```

```
            t.left = t2
            # Add this subexpression to stack
            stack.append(t)
    # Only element will be the root of expression tree
    t = stack.pop()
    return t
# Driver program to test above
postfix = "ab+ef*g*-"
r = constructTree(postfix)
print "Infix expression is"
inorder(r)
```

Syllabus Topic : Binary Search Trees (BSTs)

2.9 Binary Search Trees (BSTs)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties :

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.
- Both the left and right subtrees must also be binary search trees.
- "Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree."
- Fig. 2.9.1 shows a pictorial representation of BST.

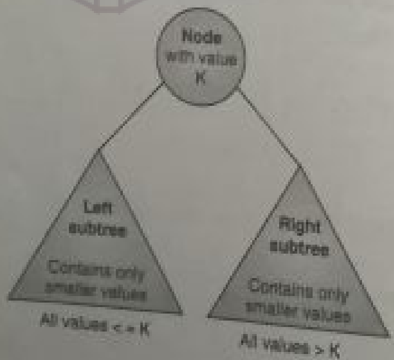


Fig. 2.9.1

Example

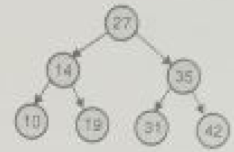


Fig. 2.9.2

We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Binary Search Tree Declaration

```
#Binary Search Tree Class and its methods
class BSTNode:
def init. (self, data):
self.data = data
self.left= None
self. right = None
#set data
def setdata(self, data):
self.data = data
#get data
def getData(self):
return self.data
#get left child of a node
def getLeft(self):
returnself.left
#get right child of a node
def getRight(self):
return self.right
```

2.9.1 Operations on Binary Search Trees

Operations on Binary Search Trees

- 1. Finding an Element in Binary Search Trees
- 2. Finding Minimum Element in Binary Search Trees
- 3. Finding Maximum Element in Binary Search Trees
- 4. Inserting an Element from Binary Search Tree
- 5. Deleting an Element from Binary Search Tree

Fig. C2.5 : Operations on Binary Search tree

→ 1. Finding an Element in Binary Search Trees

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

```
def find( root, data ):
currentNode = root
whilec urrentNode is not None and data != currentNode.getData():
if data>currentNode.getData():
currentNode = currentNode.getRight()
else:
currentNode = currentNode.getLeft()
return currentNode
```

→ 2. Finding Minimum Element in Binary Search Trees

In BSTs, the minimum element is the left-most node, which does not have left child.

```
deffindMin(root):
currentNode = root
if currentNode.getLeft() == None:
return currentNode
else:
return findMin(currentNode.getLeft())
```


→ 3. Finding Maximum Element in Binary Search Trees

In BSTs, the maximum element is the right-most node, which does not have right child.

```
#Search the key from node, iteratively
deffindMax(root):
currentNode = root
if currentNode.getRight() == None:
return currenLNode
else:
return findMax(currentNode.getRight())
```

→ 4. Inserting an Element from Binary Search Tree

- When looking for a place to insert a new key, we traverse the tree from root to leaf, making comparisons to key stored in the nodes of the tree and deciding, based on the comparisons, to continue searching in the left or right subtree.
- In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of root, or the right subtree if its key is greater than or equal to that of root.
- Code to insert a node in BST

```
def insertNode(root, node):
if root is None:
root= node
else:
if root.data>node.data:
if root.left == None:
rool.left = node
else:
insertNode(root.left, node)
else:
if root.right == None:
root.right = node
else:
insertNode(rooLright, node)
```

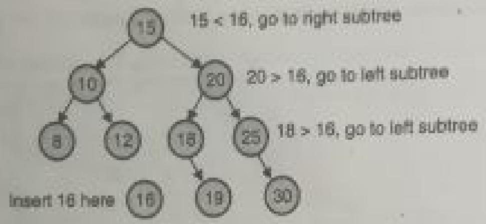


Fig. 2.9.3

➔ 5. Deleting an Element from Binary Search Tree

Deleting a node from Binary search tree has following three cases.

- **Case 1 :** Deleting a Leaf node (A node with no children)
- **Case 2 :** Deleting a node with one child
- **Case 3 :** Deleting a node with two children

1. Node to be deleted is leaf: Simply remove from the tree.

Fig. 2.9.4

2. Node to be deleted has only one child: Copy the child to the node and delete the child.

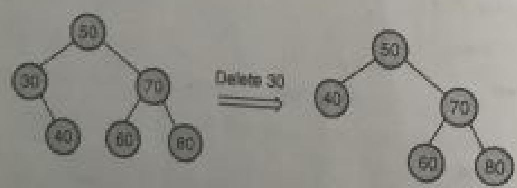


Fig. 2.9.5

3. Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

Fig. 2.9.6

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

☞ Code to delete a node in BST

```
def deleteNode(root, data):
"""delete the node with the given data and return the root node of the tree """
if root.data == data:
        # found the node we need to delete
ifroot.right and root.left:
#get the successor node and its parent
[psucc, succ] = findMin(root.right, root)
#splice out the successor
# (we need the parent to do this)
ifpsucc.left ==succ:
psucc.left ==succ.right
else:
psucc.right ==succ.right
# reset the left and right children of the successor
succ.left = root.left
succ.right = root.right
return succ
        else:
# "easier" case
ifroot.left:
return root.left
# promote the left subtree
else:
```

```
return root.right
# promote the right subtree
else:
if root == data > data:
if root.left:
root. left " deleteNode(root.left, data)
# else the data is not in the tree
else:
#data should be in the right subtree
ifroot.right:
root.right = deleteNode(root.right, data)
return root
def findMin(root, parent):
""" return the minimum node in the current tree and its parent """
# we use an ugly trick: the parent node is passed in as an argument
#so that eventually when the leftmost child is reached, the
# call can return both the parent to the successor and the successor
if root.left:
return findMin(root.data,root)
else:
return [parent, root]
```

Syllabus Topic : Balanced Binary Search Trees

2.10 Balanced Binary Search Trees

Binary search trees are a nice idea, but they fail to accomplish our goal of doing lookup, insertion and deletion each in time $O(\log_2(n))$, when there are n items in the tree. Imagine starting with an empty tree and inserting 1, 2, 3 and 4, in that order.

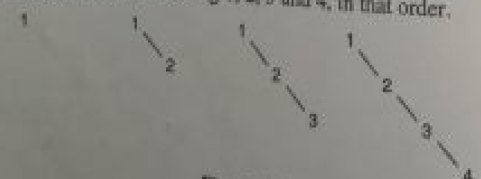


Fig. 2.10.1

- You do not get a branching tree, but a linear tree. All of the left subtrees are empty. Because of this behavior, in the worst case each of the operations (lookup, insertion and deletion) takes time $\Theta(n)$. From the perspective of the worst case, we might as well be using a linked list and linear search.
- That bad worst case behavior can be avoided by using an idea called height balancing, sometimes called AVL trees.

2.10.1 Height Balanced Trees

- The height of a node in a tree is the length of the longest path from that node downward to a leaf, counting both the start and end vertices of the path.
- The height of a leaf is 1. The height of a nonempty tree is the height of its root.
- For example, tree has height 3. (There are 3 equally long paths from the root to a leaf. One of them is (30 18 24).) The height of an empty tree is defined to be 0.

Fig. 2.10.2

- We define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

balanceFactor=height(leftSubTree)−height(rightSubTree)

- We will define a tree to be balance if the balance factor is -1, 0, or 1.

Example

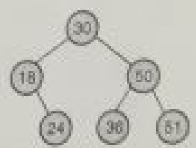


Fig. 2.10.3 : Tree

- In the above tree, balanced factor of each node is as follows :

$$\text{balanceFactor}(24) = \text{height(leftSubTree)} - \text{height(rightSubTree)} = 0 - 0 = 0$$

balanceFactor(36) = 0 – 0 = 0

balanceFactor(51) = 0 – 0 = 0

balanceFactor(18) = 0 – -1 = -1

balanceFactor(50) = 1 – 1 = 0

balanceFactor(30) = 2 – 2 = 0

– So the above tree is balanced binary search tree because the balanced factor of each node is – 1, 0, or 1.

Syllabus Topic : AVL (Adelson and Velski and Landis) Trees

2.11 AVL (Adelson, Velski and Landis) Trees

– AVL tree is also a binary search tree but it is a balanced tree. The technique of balancing the height of binary trees was developed by Adelson, Velskii and Landis and hence given the short form as AVL tree or Balanced Binary Tree.

– A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.

– An AVL tree is defined as follows :

"An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1."

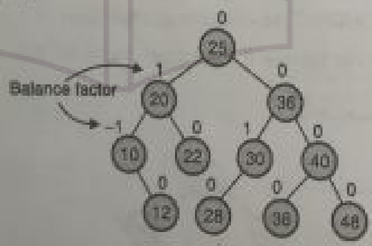


Fig. 2.11.1

– The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

AVL Tree Declaration

```
class AVLNode:
def _init_ (self,data, balanceFactor, left,right):
        self.data= data
```



```
self. balanceFactor = 0
self.left = left
self.right= right
```

Finding the Height of an AVL tree

```
def height(self):
return self.recHeight(self.root)
def recHeight(self,root):
if root == None:
return 0
else:
leftH = self.recHeight(r.left)
rightH = self. recHeight(r.right)
if leftH>rightH:
return 1 +leftH
else:
return 1 +rightH
```

2.11.1 AVL Tree Rotations

– In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree.

– If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.

– Rotation operations are used to make a tree balanced.

– There are **four** rotations and they are classified into **two** types :

1. Single rotations
   1. Left roatation (LL rotation)
   2. Right roatation (RR rotation)
2. Double rotations
   1. Left Right rotation (LR rotation)
   2. Right Left rotation (RL rotation)

2.11.1(A) Single Rotations

(a) Single Left Rotation (LL Rotation)

- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right child, then we perform a single left rotation .

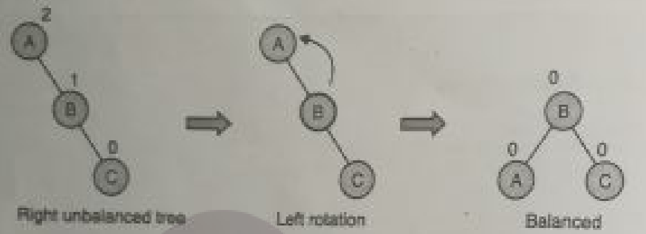


Fig. 2.11.2

- In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

(b) Single Right rotation(RR Rotation)

- AVL tree may become unbalanced, if a node is inserted in the left subtree of the left child. The tree then needs a right rotation.

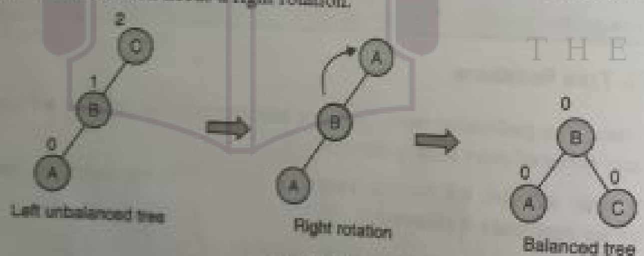


Fig. 2.11.3

- The unbalanced node becomes the right child of its left child by performing a right rotation.

2.11.1(B) Double Rotations

(a) Left-Right rotation(LR rotation)

- The LR Rotation is combination of single left rotation followed by single right rotation.

- In LR Rotation, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree.

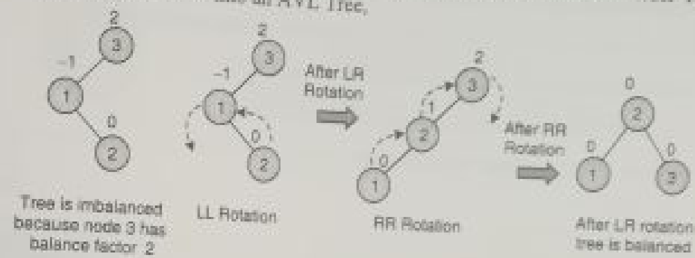


Fig. 2.11.4

(b) Right Left Rotation (RL Rotation)

- The RL Rotation is combination of single right rotation followed by single left rotation.
- In RL Rotation, first every node moves one position to right then one position to left from the current position.
- To understand RL Rotation, let us consider following insertion operations into an AVL Tree :

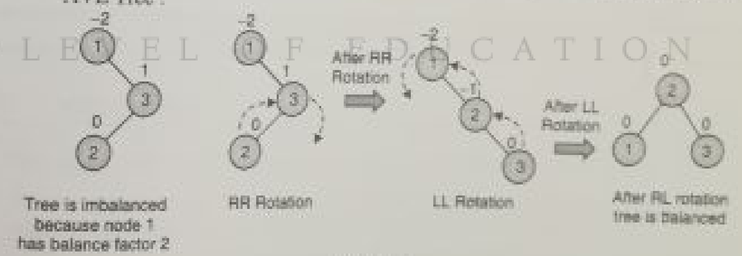


Fig. 2.11.5

Review Questions

Q. 1 What is tree ? **(Refer section 2.1)**

Q. 2 How represent binary tree? **(Refer section 2.2)**

Q. 3 Explain different types of binary tree. **(Refer section 2.3)**

Q. 4 Explain pre-order, in-order and post-order traversals with example. **(Refer sections 2.5.1, 2.5.2 and 2.5.3)**

Q. 5 Write short note on threaded binary tree traversals. **(Refer section 2.7)**

Q. 6 Define predecessor and successor. **(Refer section 2.7.2)**

Q. 7 Write short note on binary reach trees. **(Refer section 2.9)**

Q. 8 Describe AVL tree. **(Refer section 2.11)**

❑❑❑

CHAPTER 3 UNIT II

Graph and Selection Algorithms

Syllabus

Graph Algorithms : Introduction, Glossary, Applications of Graphs, Graph Representation, Graph Traversals, Topological Sort, Shortest Path Algorithms, Minimal Spanning Tree.

Selection Algorithms : What are Selection Algorithms? Selection by Sorting, Partition-based Selection Algorithm, Linear Selection Algorithm - Median of Medians Algorithm, Finding the K Smallest Elements in Sorted Order.

Syllabus Topic : Graph Algorithm : Introduction

3.1 Introduction

- This chapter introduces an important non-linear data structure called Graph.
- It has applications in various fields like electrical and electronics engineering, computer science, games and puzzles, Geographical Information System etc.

Syllabus Topic : Glossary

3.1.1 Glossary of Graph

- Graph is a non linear data structure. A Graph G consists of finite set of vertices V and finite set of edges E which can be denoted by G = (V, E).
- Where, V represent the entities which has names and other attributes.
- Edges E represent the links that connect the vertices.