Trees

Introduction

A tree is an ideal data structure for representing hierarchical data. A tree can be theoretically defined as a finite set of one or more data items (or nodes) such that:

1. There is a special node called the **root** of the tree.

2. Removing nodes (or data item) are partitioned into number of mutually exclusive (i.e., disjoined) subsets each of which is itself a tree, are called sub tree.

Basic Tree Structure

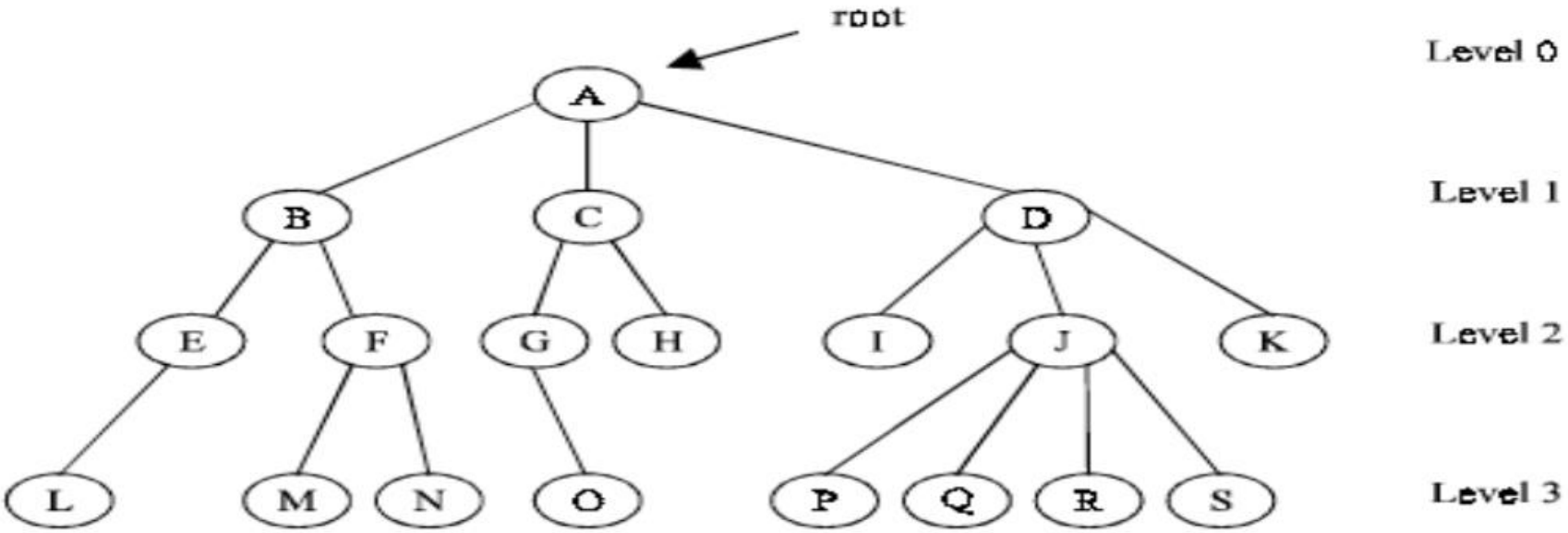


Fig 1: A Tree

Basic Terminology

Root: Root is the first node in the hierarchical arrangement of the data items. ‘A’ is a root node in the Fig1. Each data item in a tree is called a **node**. It specifies the data information and links (branches) to other data items.

Degree of a node is the number of sub trees of a node in a given tree. In an above figure, The degree of node A, B, C and D are 3,2,2 and 3 respectively.

The **degree of a tree** is the maximum degree of node in a given tree. In above tree, degree of a node J is 4. All the other nodes have less or equal degree, so, the degree of the above tree is 4.

A node with degree 0 is called a **terminal node** or a **leaf**. In above tree M, N, I, O etc. are leaf node (terminal node).

Any Node whose degree is not zero is called **non-terminal** node. There are intermediate nodes while traversing the given tree from its root node to the terminal nodes.

A tree is structured in different **levels**. The entire tree is leveled in such a way that the root node is always of level 0. Then, its immediate children are level 1 and their immediate children are at level 2 and so on up to the terminal nodes. That is, if a node is at level n, then its children will be at level n+1.

Depth of a tree is the maximum level of any node in a given tree. The term height is also used to denote the depth.

Binary tree

A binary tree is a tree in which no node can have more than two children. Typically these children are described as “left child” and “right child” of the parent’s node.

A binary tree T is defined as a finite set of elements, called nodes, such that:

- T is empty (i.e. if T has no node called null tree or empty tree).
- T contains a special node R, called root node of T, and the remaining nodes of T from an ordered pair of disjoined binary trees T1 and T2, and they are called left and right sub tree of R. If T1 is non empty then its root is called the left successor of R, Similarly if T2 is non empty then its root is called the right successor of R.

Consider a binary tree T in Fig 2. Here ‘A’ is the root node of the binary tree T. Then ‘B’ is the left child of ‘A’ and ‘C’ is the right child of ‘A’ i.e., ‘A’ is a father of ‘B’ and ‘C’. The node ‘B’ and ‘C’ are called brothers. If a node has no child then it is called a leaf node. Nodes H, I, F, J are leaf node in Fig 2.

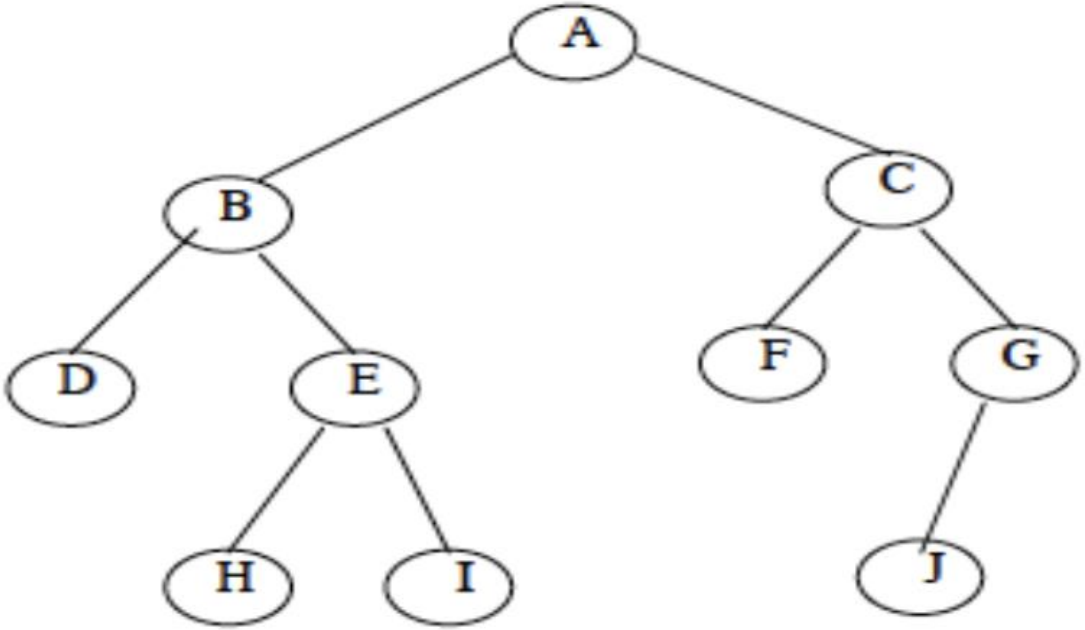


Fig 2: Binary Tree

Strictly Binary tree

The tree is said to be **strictly binary tree**, if every non-leaf node in a binary tree has non-empty left and right sub trees. A strictly binary tree with *n* leafs always contains *2n–1* node. The tree in Fig 3 is strictly binary tree; where as the tree in Fig 2 is not. That is every node in the strictly binary tree can have either no children or two children. They are also called *2-tree or extended binary tree*.

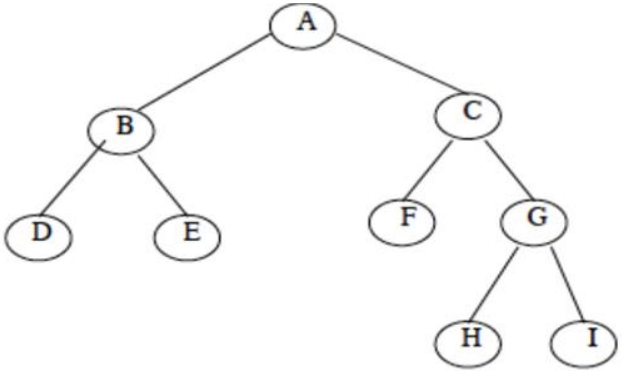


Fig 3: Strictly Binary Tree

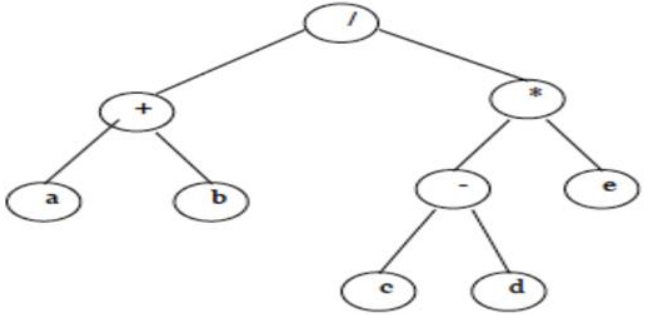


Fig 4: Expression Tree

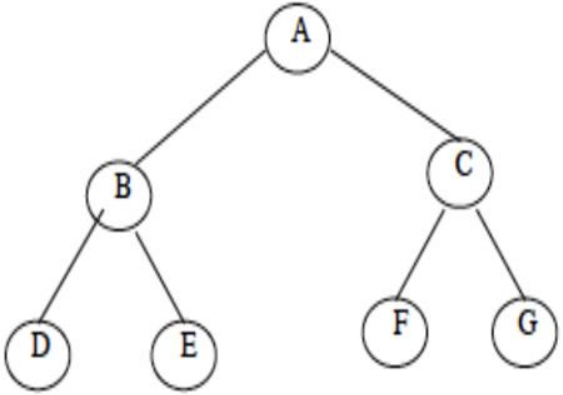


Fig5: Complete Binary Tree

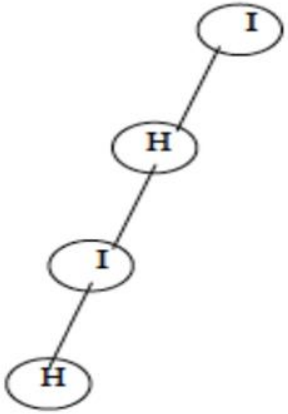


Fig6: Left Skewed

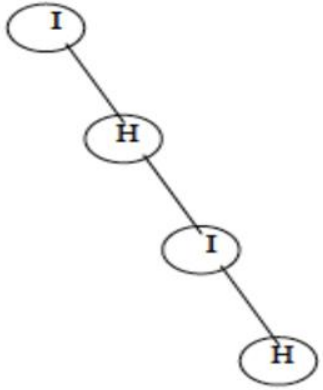


Fig7: Right Skewed

Complete Binary tree

A **complete binary tree** at depth '*d*' is the strictly binary tree, where all the leafs are at level *d*. Fig 5 illustres the complete binary tree of depth 2.

A binary tree with n nodes, n > 0, has exactly n – 1 edges. If a binary tree contains *n* nodes at level *I*, then it contains at most 2n nodes at level *I* + 1. A complete binary tree of depth *d* is the binary tree of depth *d* contains exactly 2^I nodes at each level *I* between 0 and *d*.

The main difference between a binary tree and ordinary tree is:

1. A binary tree can be empty where as a tree cannot.

2. Each element in binary tree has exactly two sub trees (one or both of these sub trees may be empty). Each element in a tree can have any number of sub trees.

3. The sub tree of each element in a binary tree is ordered, left and right sub trees. The sub trees in a tree are unordered.

If a binary tree has only left sub trees, then it is called **left skewed** binary tree. Fig 6 is a left skewed binary tree. If a binary tree has only right sub trees, then it is called **right skewed**

Binary tree representation

ARRAY REPRESENTATION

An array can be used to store the nodes of a binary tree. The nodes stored in an array of memory can be accessed sequentially. Suppose a binary tree T of depth d. Then at most 2^d-1 nodes can be there in T. (i.e.,SIZE = 2^d– 1).Consider a binary tree in Fig8 of depth 3. Then SIZE = 2^3 – 1 = 7. Then the array A [7] is declared to hold the nodes.

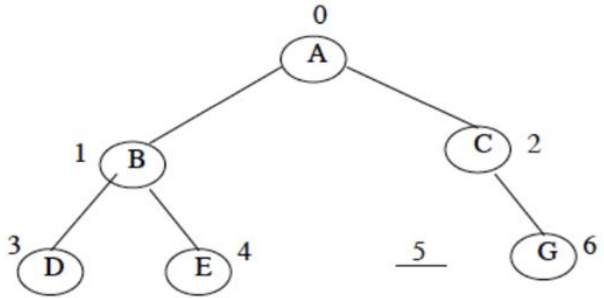


Fig8: Binary Tree of Depth 3

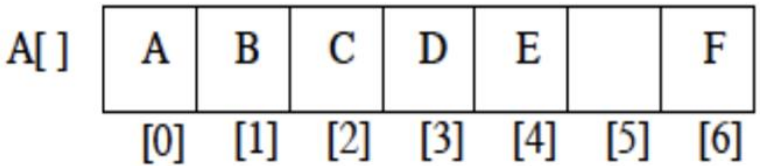


Fig9: Array Representation of Binary Tree of Fig8

LINKED LIST REPRESENTATION

The most popular and practical way of representing a binary tree is using linked list (or pointers). In linked list, every element is represented as nodes. A node consists of three fields such as:

(a) Left Child (LChild)

(b) Information of the Node (Info)

(c) Right Child (RChild)

The LChild links to the left child of the parent's node, info holds the information of every node and RChild holds the address of right child node of the parent node. The following figures shows that the Linked List representation of the binary tree.

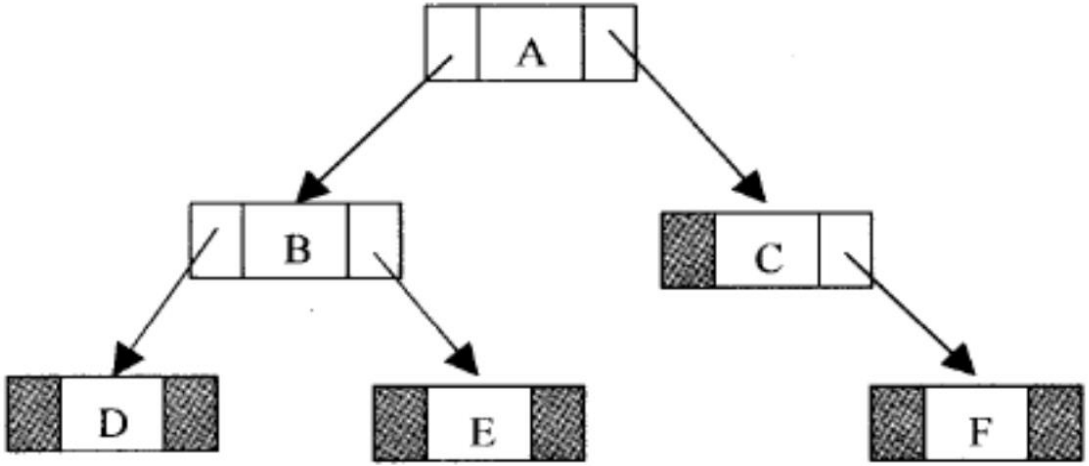


Fig9: Linked List representations of Binary Tree

Basic Operation in Binary Tree

- Traversing a binary tree
- Inserting a new node
- Deleting a Node
- Searching for a Node
- Copying the mirror image of a tree
- Determine the total no: of Nodes
- Determine the total no: leaf Nodes
- Determine the total no: non-leaf Nodes

Traversing Binary Tree

Tree traversal is one of the most common operations performed on tree data structures. It is a way in which each node in the tree is visited exactly once in a systematic manner. There are three standard ways of traversing a binary tree. They are:

- Pre Order Traversal (Node-left-right)
- In order Traversal (Left-node-right)
- Post Order Traversal (Left-right-node)

Pre Order (Root-Left-Right)

A systematic way of visiting all nodes in a binary tress that visits a node, then visits the nodes in the left sub tree of the node, and then visits the nodes in the right sub tree of the node.

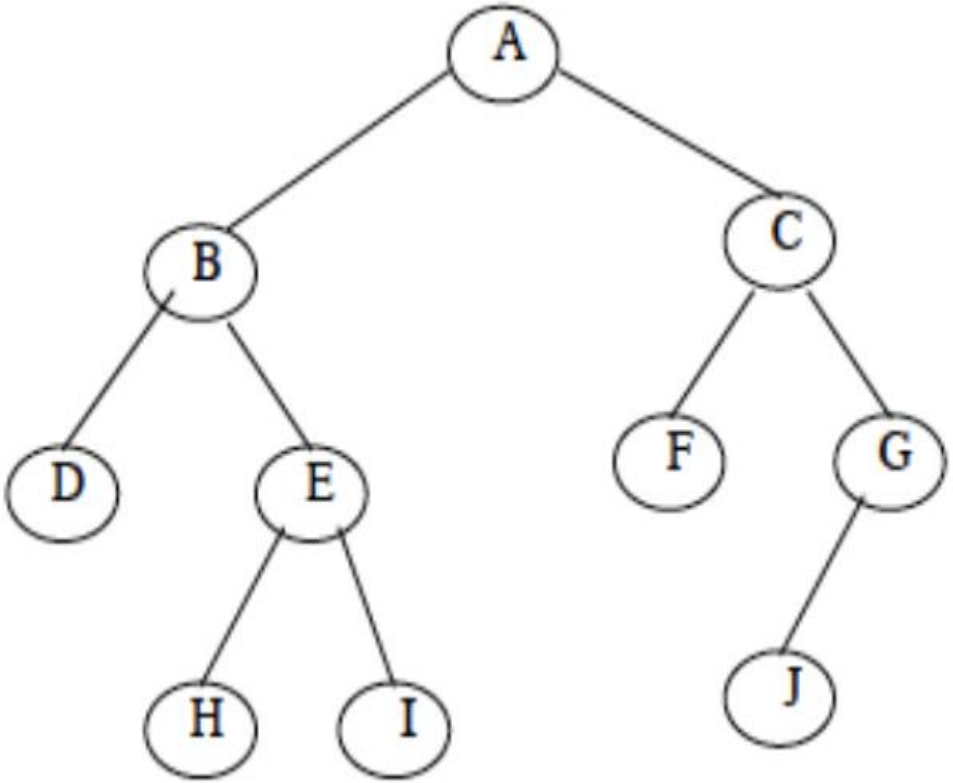
- Visit the root node

- Traverse the left sub tree in preorder
- Traverse the right sub tree in preorder

That is in preorder traversal, the root node is visited (or processed) first, before traveling through left and right sub trees recursively. C/C++ implementation of preorder traversal technique is as follows:

```
void preorder (Node * Root) {

    If (Root != NULL) {

printf ("%d\n",Root → Info); preorder(Root → Lchild); preorder(Root → Rchild);

    } }
```

Pre order traversal



The preorder traversal of the following Binary tree is:
A,B,D,E,H,I,C,F,G,J

Inorder traversal (left - root - right)

A systematic way of visiting all nodes in a binary tress that visits the nodes in the left sub tree of a node, then visits the node, and then visits the nodes in the right sub tree of the node.

- Traverse the left sub tree in order
- Visit the root node
- Traverse the right sub tree in order

In order traversal, the left sub tree is traversed recursively, before visiting the root. After visiting the root the right sub tree is traversed recursively. C/C++ implementation of inorder traversal technique is as follows:

*void inorder (NODE *Root) {*

If (Root != NULL){

The in order traversal of a binary tree in Fig10 is: D, B, H, E, I, A, F, C, J, G.

Post order traversal (left-right-root)

A systematic way of visiting all nodes in a binary tress that visits the node in the left sub tree of the node, then visits the node in the right sub tree in the node, and then visits the node.

- Traverse the left sub tree in post order
- Traverse the right sub tree in post order

- Visit the root node

In Post Order traversal, the left and right sub tree(s) are recursively processed before visiting the root. C/C++ implementation of post order traversal technique is as follows:

```
void postorder (NODE *Root) {

If (Root != NULL) {

postorder(Root →Lchild);

postorder(Root → Rchild);

printf ("%d\n", Root → info);        }}
```

The post order traversal of a binary tree in Fig10 is: D, H, I, E, B, F, J, G, C, A.

Expression tree

An expression tree is used to represent arithmetic and logical expressions.

It is built up from the simple operands and operators of an expression by placing the simple operands as the leaves of a binary tree, and the operators as the interior nodes.

For each binary operator, the left subtree contains all the simple operands and operators in the left operand of the given operator and the right subtree contains everything in the right operand

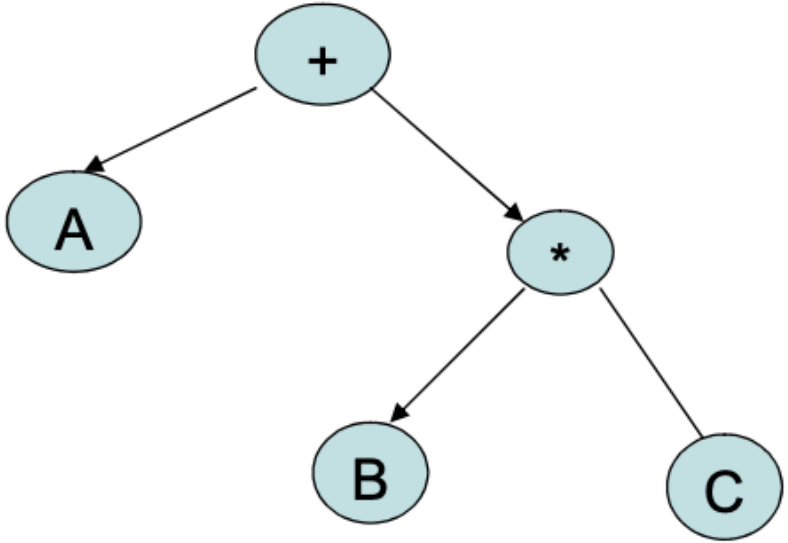
For a unary operator, one subtree will be empty.

Example of an expression tree from a given expression 1. A + B * C

When an expression tree is traversed in inorder, we get an infix expression. When it is traversed in preorder, we get the equivalent prefix expression. Similarly, when it is traversed in postorder, we get the equivalent postfix expression.So, now let us check:

The preorder traversal is: + A * B C (equivalent to prefix expression)

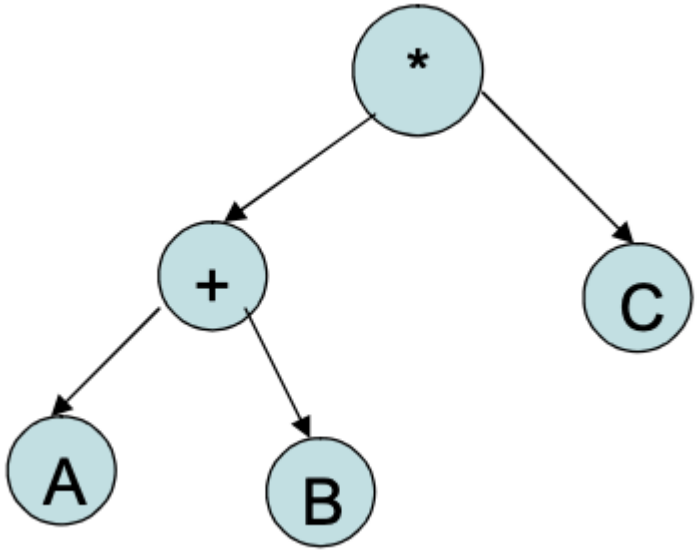
The postorder traversal is: A B C * + (equivalent to postfix expression)



(A + B) * C

The preorder traversal is: * + A B C
(equivalent to prefix expression)

The postorder traversal is: A B + C *
(equivalent to postfix expression)



Construct a Binary Tree from Preorder and Inorder

```
Inorder Traversal : { 4, 2, 1, 7, 5, 8, 3, 6 }
Preorder Traversal: { 1, 2, 4, 3, 5, 7, 8, 6 }
```

- Inorder sequence: D B E A F C
- Preorder sequence: A B D E C F

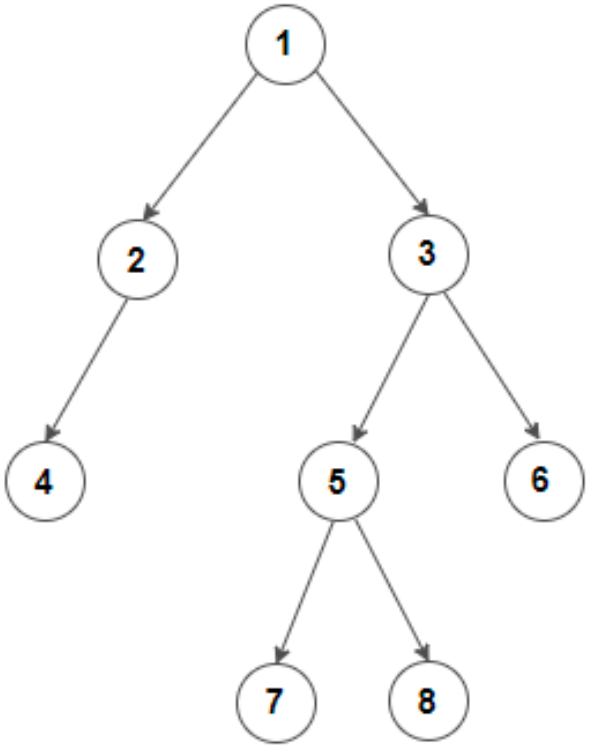
Construct a Binary Tree from Preorder and Inorder

```
Inorder Traversal :
```

```
{ 4, 2, 1, 7, 5, 8, 3, 6 }
```

```
Preorder Traversal:
```

```
{ 1, 2, 4, 3, 5, 7, 8, 6 }
```



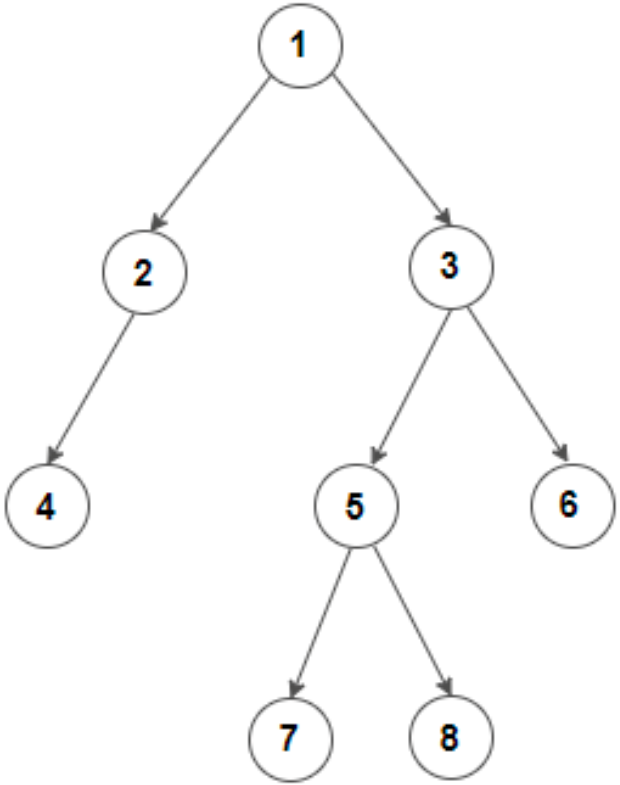
Construct a Binary Tree from Postorder and Inorder

```
Inorder Traversal   : { 4, 2, 1, 7, 5, 8, 3, 6 }
Postorder Traversal : { 4, 2, 7, 8, 5, 6, 3, 1 }
```

```
Inorder Traversal: D B E A F C
Postorder Traversal: D E B F C A
```

Construct a Binary Tree from Postorder and Inorder

```
Inorder Traversal   :

{ 4, 2, 1, 7, 5, 8, 3, 6 }

Postorder Traversal :

{ 4, 2, 7, 8, 5, 6, 3, 1 }
```



Binary Search Tree (BST)

A Binary Search Tree is a binary tree, which is either empty or satisfies the following properties:

1. Every node has a value and no two nodes have the same value (i.e., all the values are unique).

2. If there exists a left child or left sub tree then its value is less than the value of the root.

3. The value(s) in the right child or right sub tree is larger than the value of the root node.

The Fig11 shows a typical binary search tree. Here the root node information is 50. Note that the right sub tree node's value is greater than 50, and the left sub tree nodes value is less than 50. Again right child node of 25 has large values than 25 and left child node has small values than 25. Similarly right child node of 75 has large values than 75 and left child node has small values that 75 and so on.

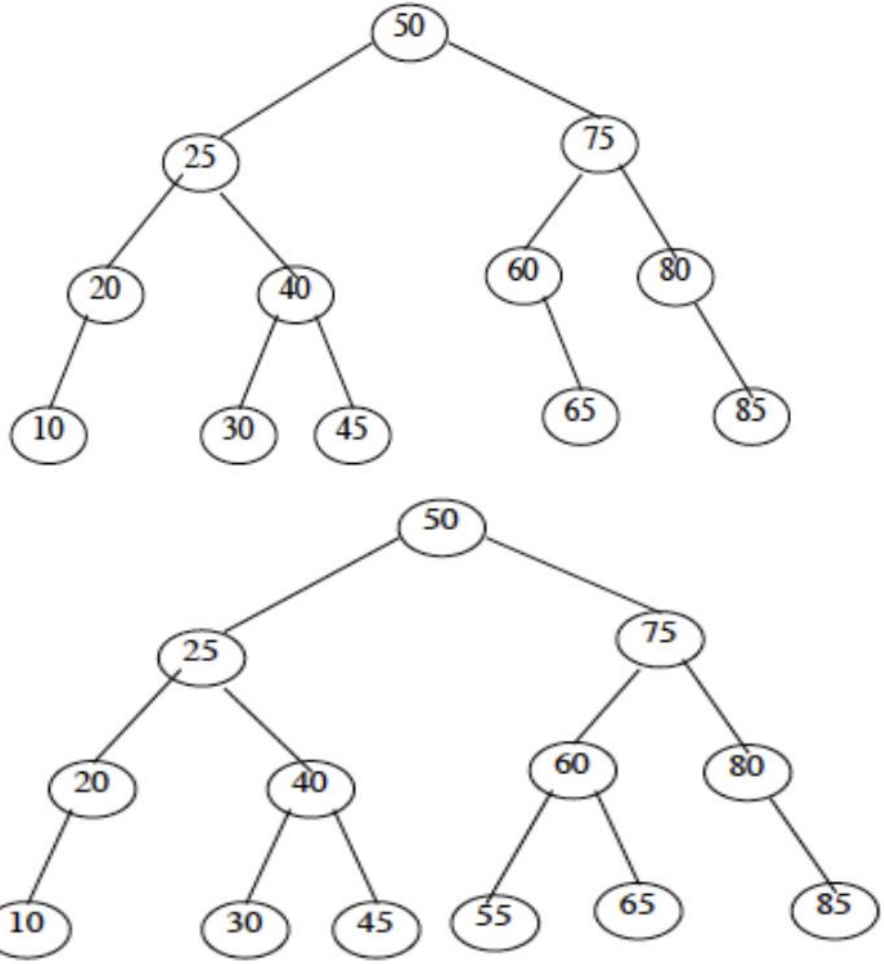


Fig11: Binary Search Tree

Inserting a node in BST

A BST is constructed by the repeated insertion of new nodes to the tree structure. Inserting a node in to a tree is achieved by performing two separate operations.

1. The tree must be searched to determine where the node is to be inserted.
2. Then the node is inserted into the tree.

Suppose a “DATA” is the information to be inserted in a BST.

Step 1: Compare DATA with root node information of the tree

(i) If (DATA < ROOT → Info)

Proceed to the left child of ROOT

(ii) If (DATA > ROOT → Info)

Proceed to the right child of ROOT

Step 2: Repeat the Step 1 until we meet an empty sub tree, where we can insert the DATA in place of the empty sub tree by creating a new node.

Step 3: Exit

For example, consider a binary search tree in Fig11. Suppose we want to insert a DATA = 55 in to the tree, then following steps one obtained:

1. Compare 55 with root node info (i.e., 50) since 55 > 50 proceed to the right sub tree of 50.
2. The root node of the right sub tree contains 75. Compare 55 with 75. Since 55 < 75 proceed to the left sub tree of 75.
3. The root node of the left sub tree contains 60. Compare 55 with 60. Since 55 < 60 proceed to the right sub tree of 60.
4. Since left sub tree is NULL place 55 as the left child of 60 as shown in above Fig12.

Algorithm for insertion in BST

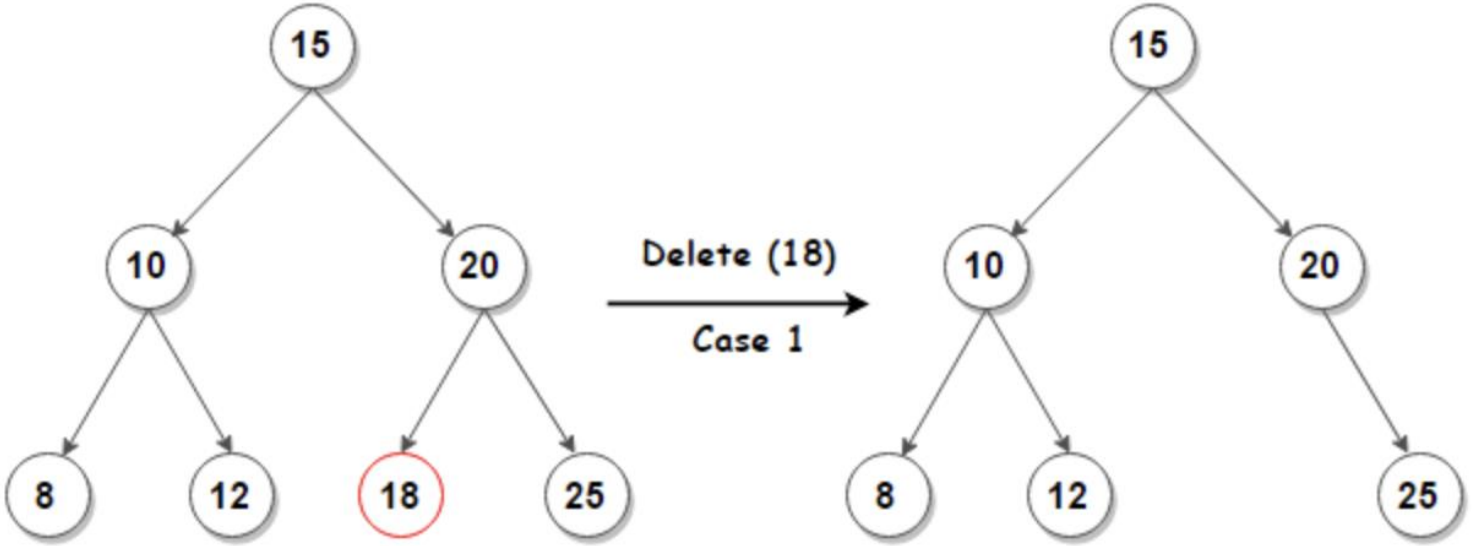
```
void insert(int Data, NODE Root) {

NODE NewNode = (NODE)malloc(sizeof(NODE));

NODE temp;

NewNode->info = Data;

if(Root == NULL)

        Root = NewNode;

else            {

        temp = Root;


        while(1) {

                if(Data < temp->info) {


                if(temp->LChild != NULL)

else {

                        temp->LChild = NewNode;

return; }

}

else if(Data > temp->RChild) {


if(temp->RChild != NULL)

temp = temp->RChild;


else {


temp->RChild = NewNode;

return; }

} else {

printf("Duplicate node \n");
```

Algorithm for Searching A Node from BST

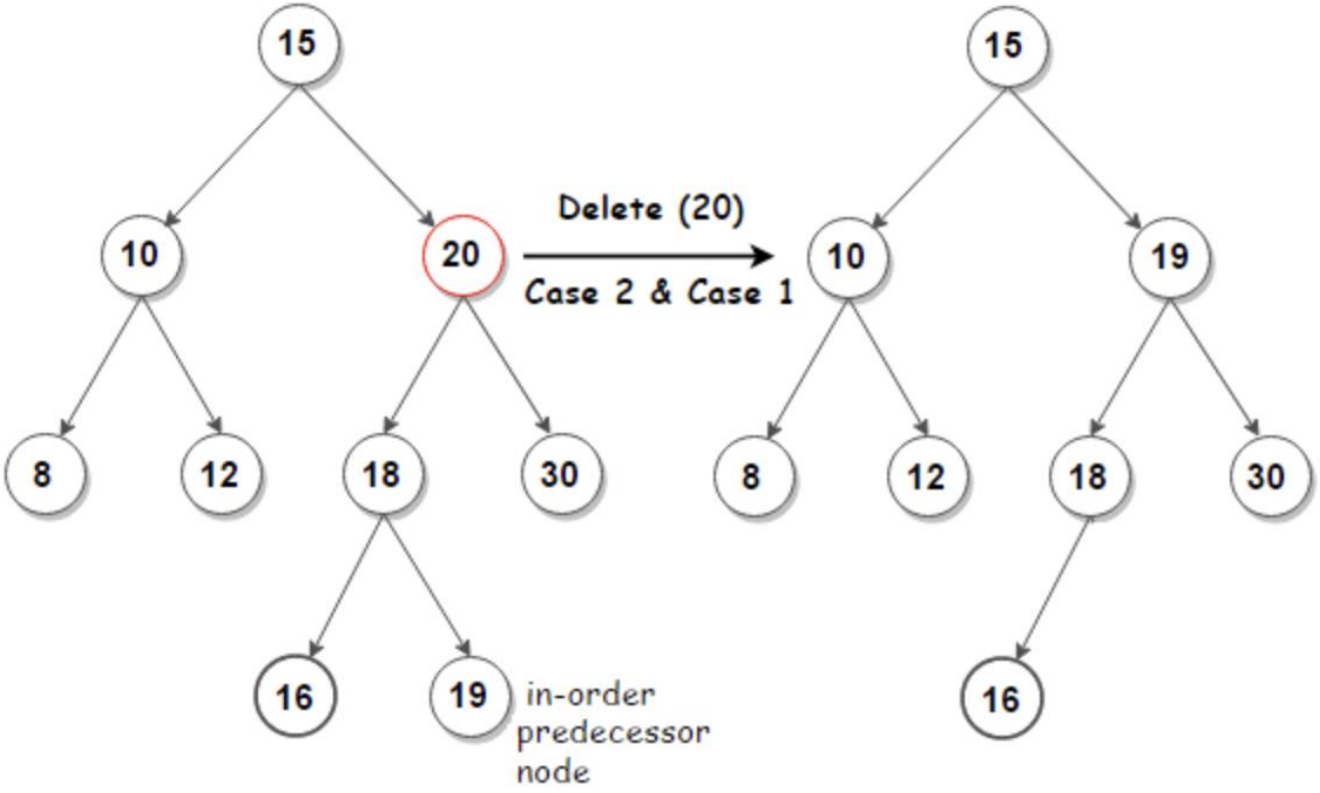
```
void findBST(int data, NODE Root, NODE *node) {

    NODE TEMP = Root;
    while(data != TEMP->info) {
        if(data == TEMP->info) {
            printf("Data Found in tree \n");
            *node = TEMP;
            return;

        }

    if(TEMP == NULL) {

        printf("Data Not found in tree");
        return;
    }
    if(data > TEMP->info)
        TEMP = TEMP->RChild;

    else if(data < TEMP->info)
        TEMP = TEMP->LChild;



} }
```

Deletion in binary search tree

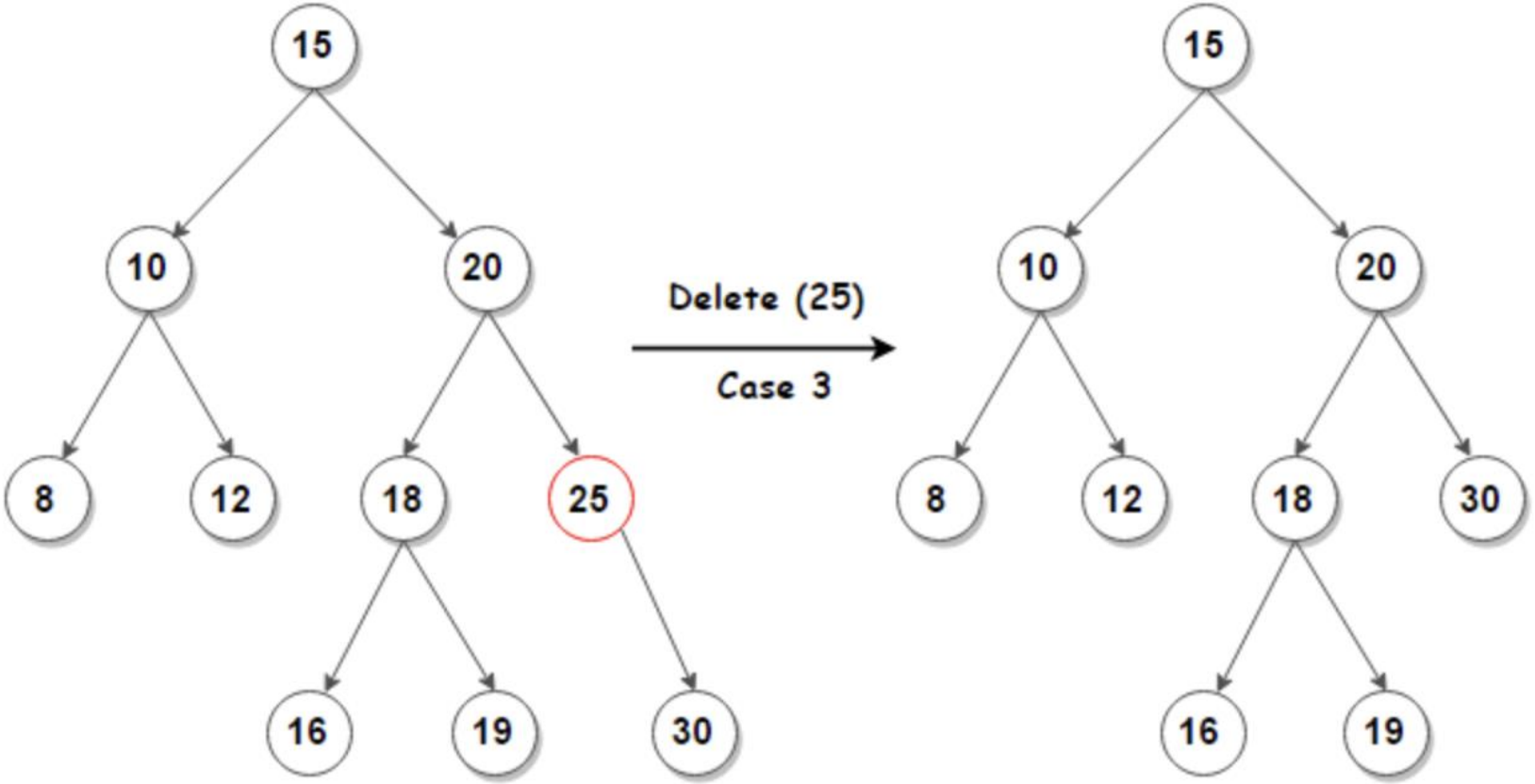
Case 1: Deleting a node with no children: remove the node from the tree.



Case 2: Deleting a node with 2 children



Case 3: Deleting a node with one child: remove the node and replace it with its child.



Algorithm for deleting a node in BST

NODE is the current position of the tree, which is in under consideration. LOC is the place where node is to be replaced and TEMP is the temporary node. DATA is the information of node to be deleted.

1. Find the location NODE of the DATA to be deleted.

2. If (NODE = NULL)

   (a) Display “DATA is not in tree”

   (b) Exit

3. If(NODE → LChild == NULL) (Node has no left child)
   (a) NODE = NODE → RChild

4. ELSE If(NODE → RChild == NULL) (Node has no right child)
   (a) NODE = NODE → LChild

5. Else If((NODE → LChild not equal to NULL) && (NODE → RChild not equal to NULL)
   (a) TEMP = NODE →LChild
   (b) LOC = NODE
   (c) While(TEMP → RChild not equal to NULL)

(d) NODE→info = TEMP→info

(e) IF(LOC == NODE)

i. LOC→LChild =TEMP→LChild

(f) ELSE

i. LOC → RChild = TEMP→LChild

6. Free TEMP

7. Exit

Balanced Binary Tree

A balanced binary tree is one in which the **largest path** through the left sub tree is the same length as the **largest path** of the right sub tree, i.e., from root to leaf.

Searching time is very less in balanced binary trees compared to unbalanced binary tree. i.e., balanced trees are used to maximize the efficiency of the operations on the tree.

AVL Trees

Two Russian Mathematicians, G.M. Adel'son Vel'sky and E.M. Landis developed algorithm in 1962; here the tree is called AVL Tree.

An AVL tree is a binary tree in which the left and right sub tree of any node may differ in height by at most 1, and in which both the sub trees are themselves AVL Trees.

Each node in the AVL Tree possesses any one of the following properties:

- A node is called **left heavy**, if the largest path in its left sub tree is one level larger than the largest path of its right sub tree

- A node is called **right heavy**, if the largest path in its right sub tree is one level larger than the largest path of its left sub tree.
- c. The node is called **balanced**, if the largest paths in both the right and left sub trees are equal. Fig17 shows some example for AVL trees.

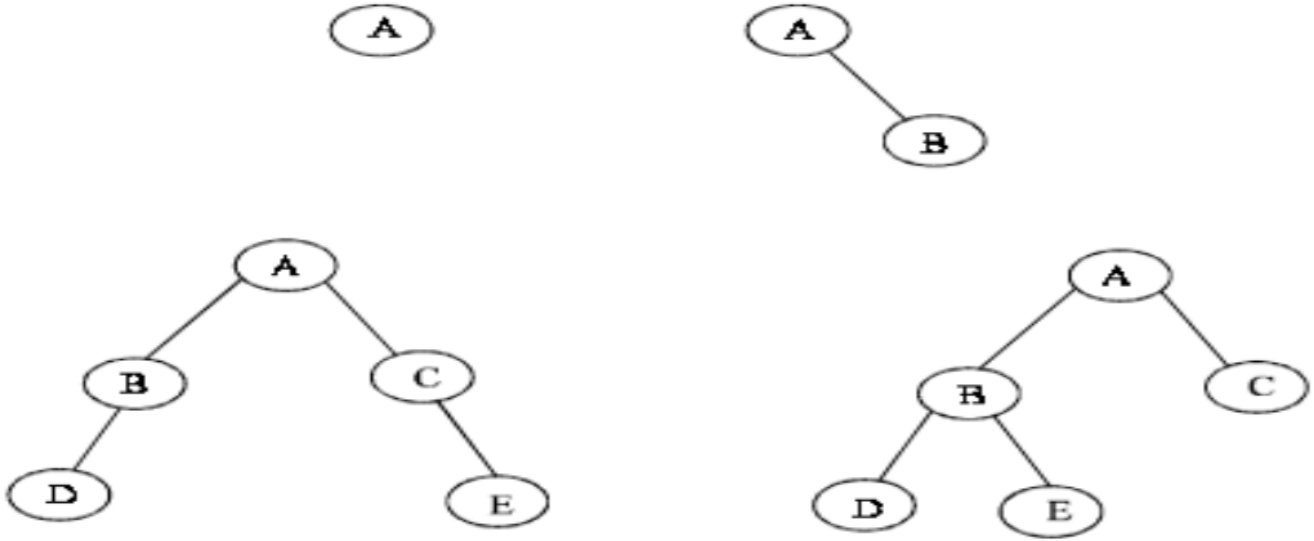


Fig17: AVL Trees

The construction of an AVL Tree is same as that of an ordinary binary tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated.

If the new node causes an imbalance in the tree, some rearrangement of the tree's nodes must be done.

Inserting a node in an AVL tree

Main point: balanced factor = Height of left subtree – height of right sub tree.

1. Insert the node in the same way as in an ordinary binary tree.
2. Trace a path from the new nodes, back towards the root for checking the height difference of the two sub trees of each node along the way.
3. Consider the node with the imbalance and the two nodes on the layers immediately below.
4. If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.
5. If these three nodes lie in a dogleg pattern (i.e., there is a bend in the path) apply a double rotation to correct the imbalance.
6. Exit.

The above algorithm will be illustrated with an example shown in Fig18, which is an unbalance tree. We have to apply the rotation to the nodes 40, 50 and 60 so that a balance tree is generated. Since the three nodes are lying in a straight line, single rotation is applied to restore the balance.

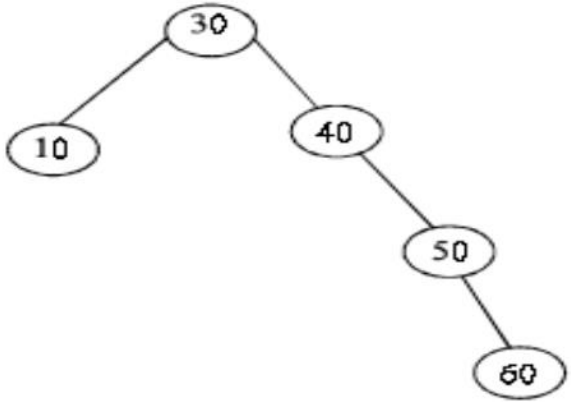


Fig 18

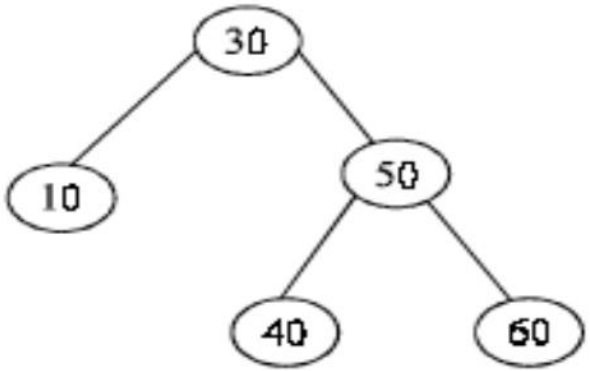


Fig19

Fig19 is a balance tree of the unbalanced tree in Fig18.

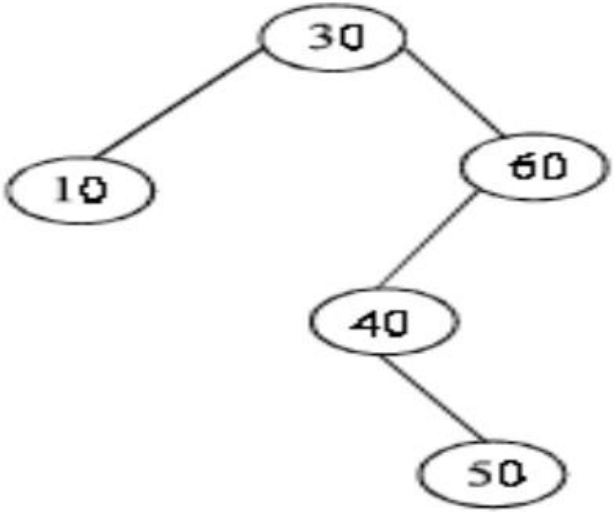


Fig20

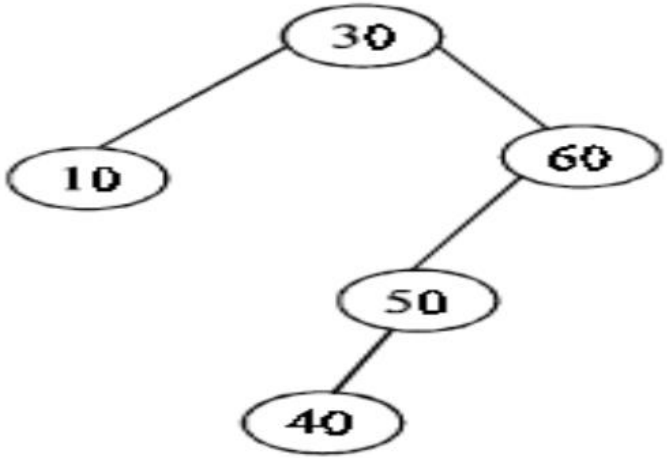


Fig21

Consider a tree in Fig20 to explain the double rotation.

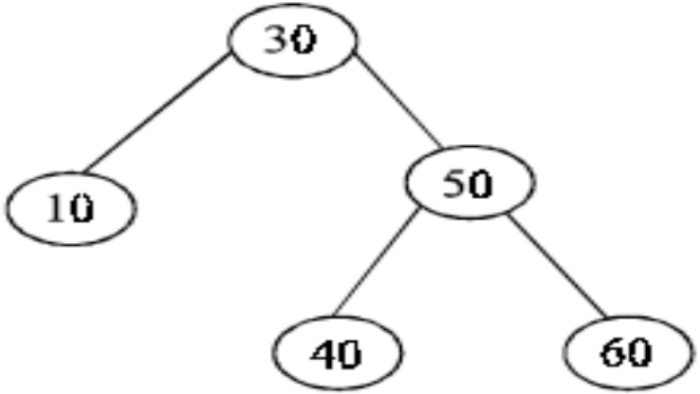
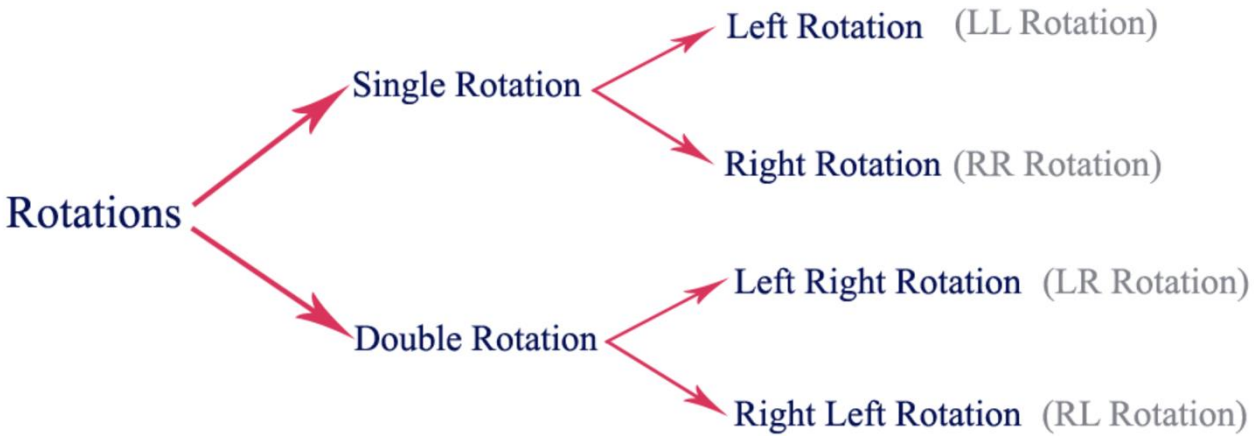


Fig22

AVL Tree Rotation

In AVL tree, after performing operations 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. Whenever the tree becomes imbalanced due to any operation we use **rotation** operations to make the tree balanced.

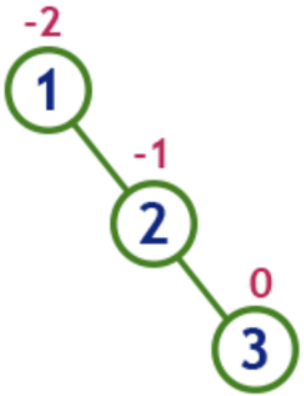
Rotation operations are used to make the tree balanced.



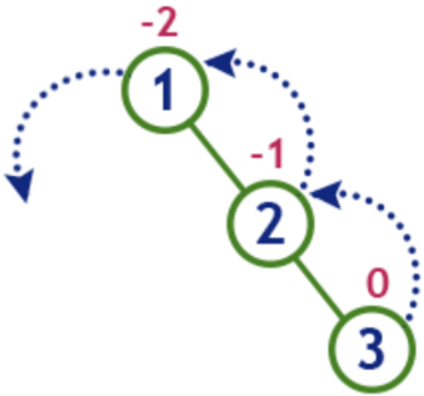
Single left rotation

In LL Rotation, every node moves one position to left from the current position. To understand LL Rotation, let us consider the following insertion operation in AVL Tree...

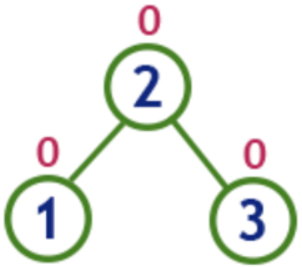
insert 1, 2 and 3



Tree is imbalanced



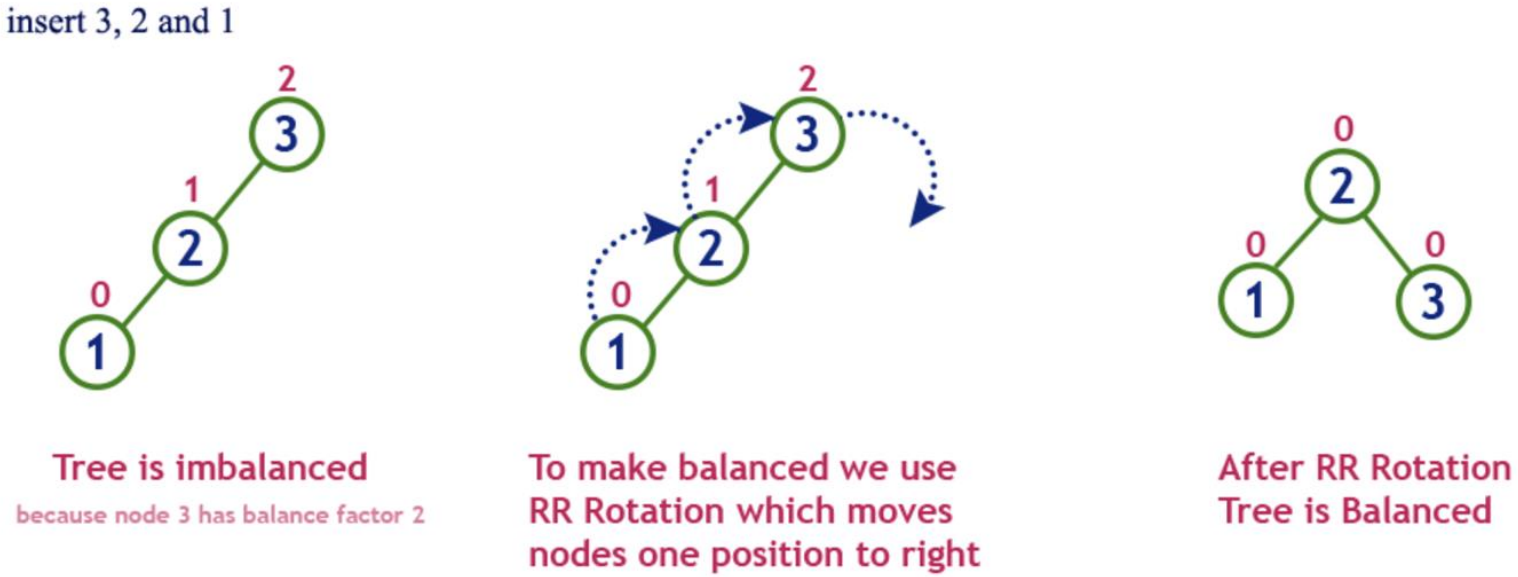
To make balanced we use LL Rotation which moves nodes one position to left



After LL Rotation Tree is Balanced

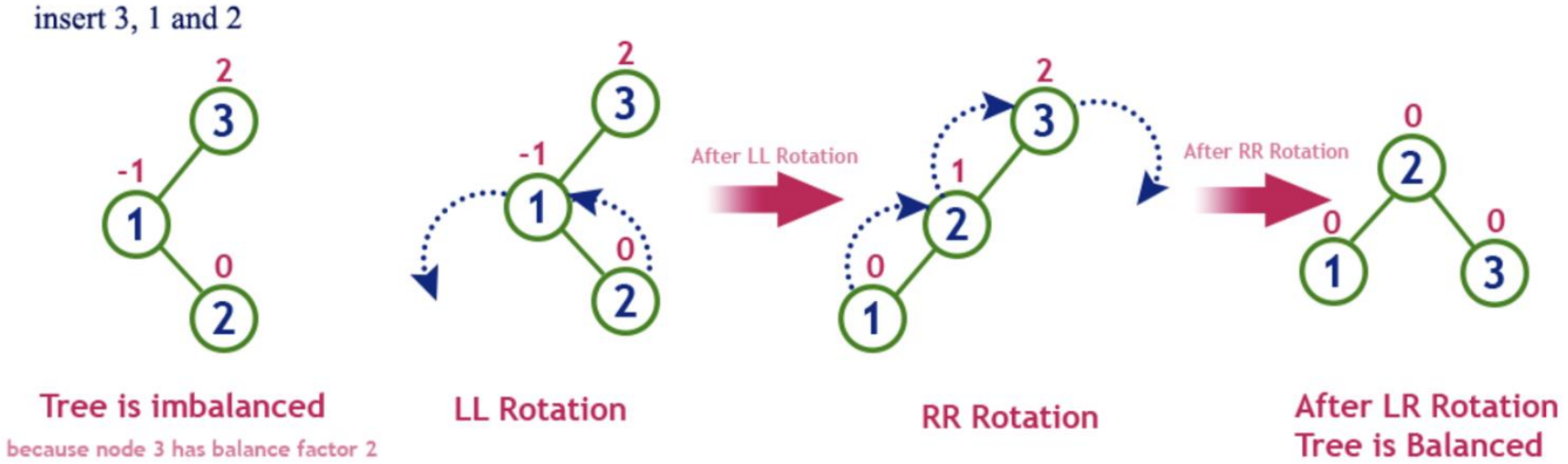
Single right rotation

In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...



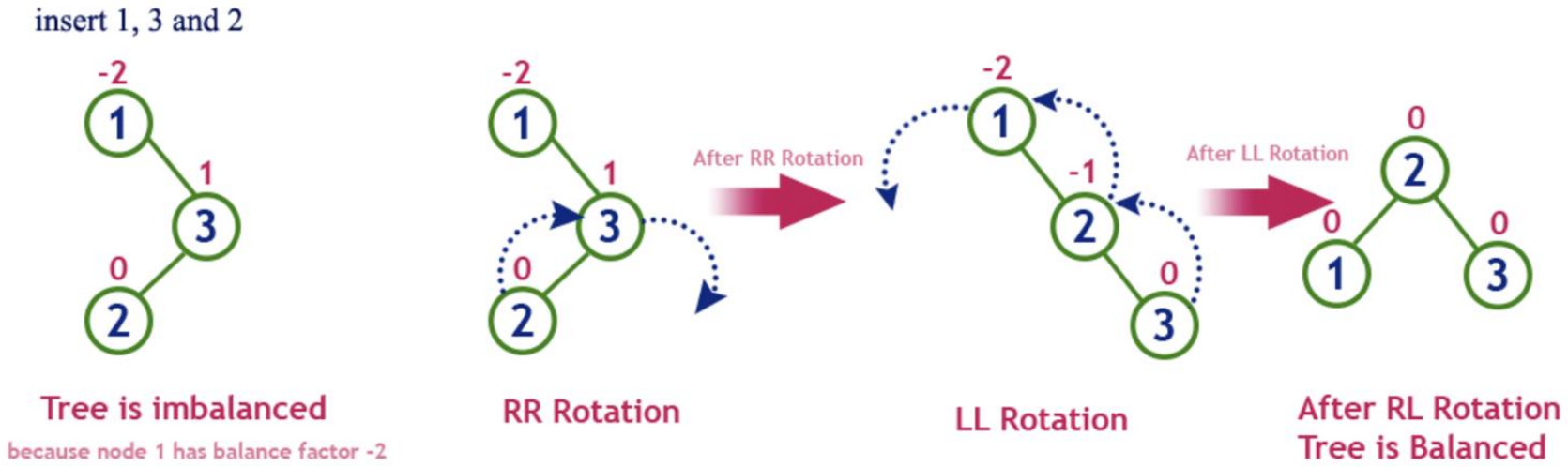
Left-Right Rotation (LR)

The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Right Left Rotation (RL)

The RL Rotation is sequence of single right rotation followed by single left rotation. In RL Rotation, at first every node moves one position to right and one position to left from the current position. To understand RL Rotation, let us consider the following insertion operation in AVL Tree...



Insertion in AVL

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, a new node is always inserted as a leaf node. The insertion operation is performed as follows...

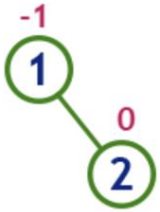
- Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2 - After insertion, check the **Balance Factor** of every node.
- Step 3 - If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
- Step 4 - If the **Balance Factor** of any node is other than **0 or 1 or -1** then that tree is said to be imbalanced. In this case, perform suitable **Rotation** to make it balanced and go for next operation.

Construct an AVL tree by inserting number from 1 to 8.

insert 1

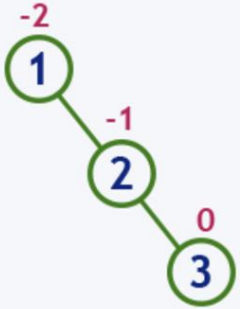
Tree is balanced

insert 2

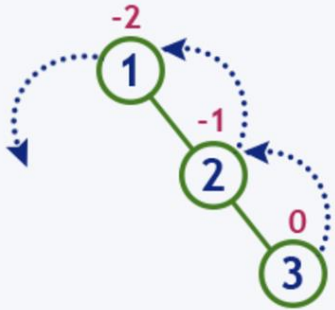


Tree is balanced

insert 3

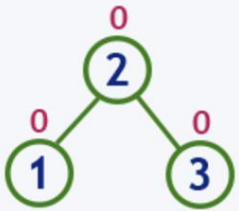


Tree is imbalanced



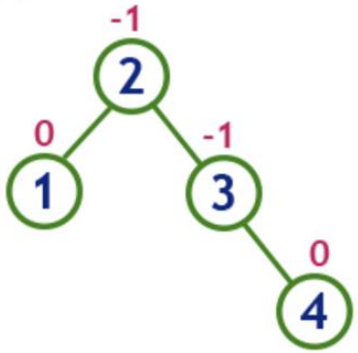
LL Rotation

After LL Rotation



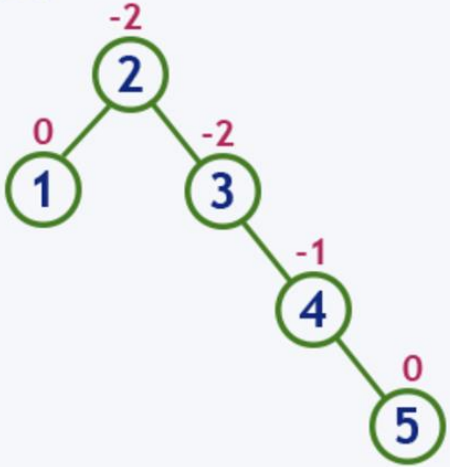
Tree is balanced

insert 4

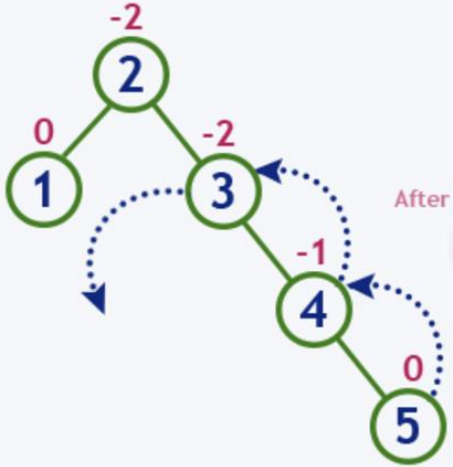


Tree is balanced

insert 5

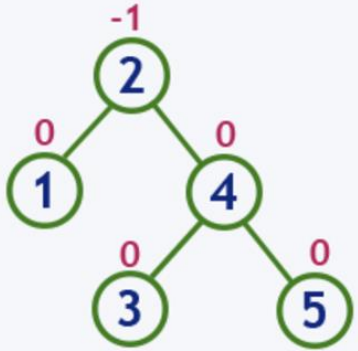


Tree is imbalanced

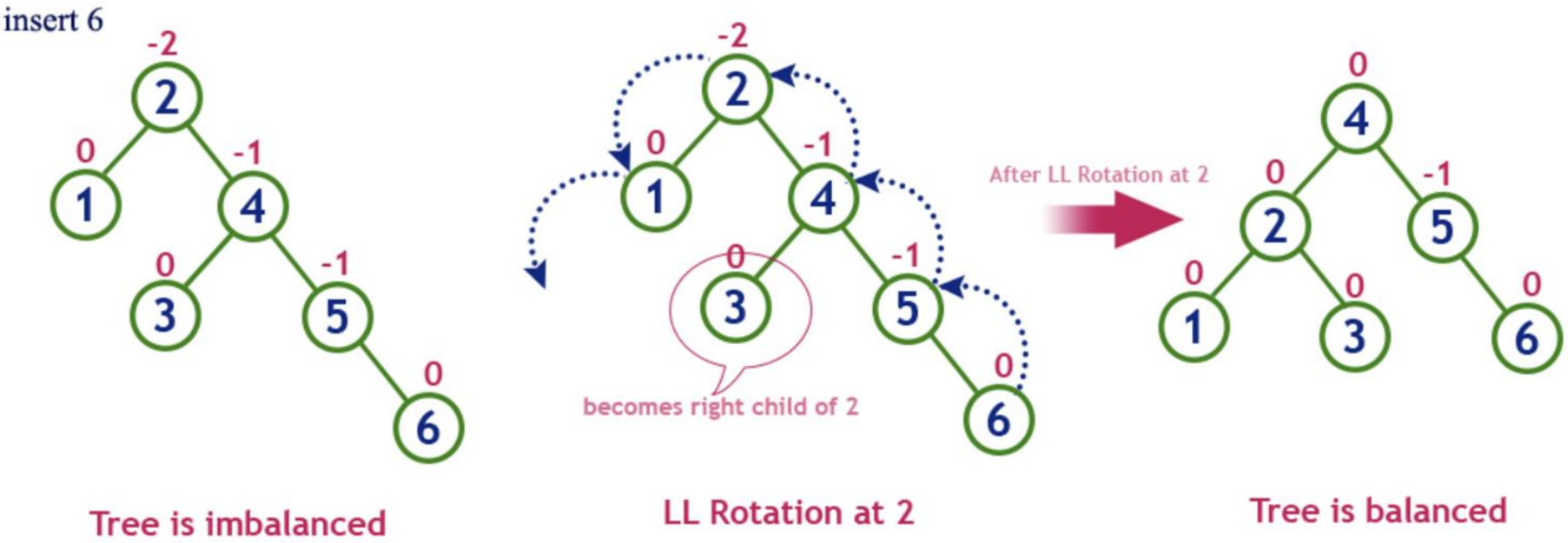


LL Rotation at 3

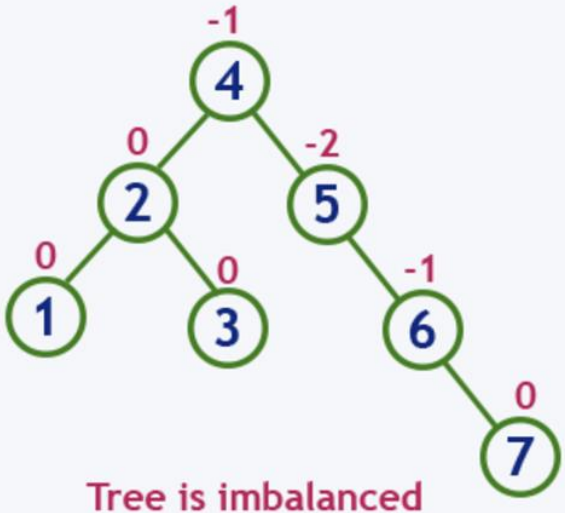
After LL Rotation at 3



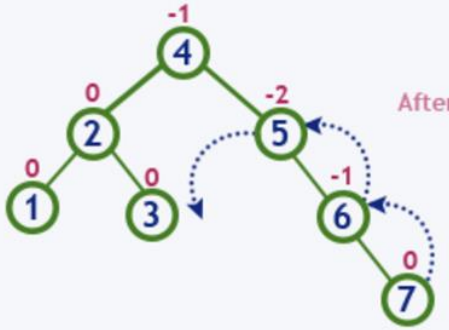
Tree is balanced


insert 6
-2
2
0
1
-1
4
0
3
-1
5
0
6
becomes right child of 2
After LL Rotation at 2
Tree is imbalanced
LL Rotation at 2
Tree is balanced

insert 7

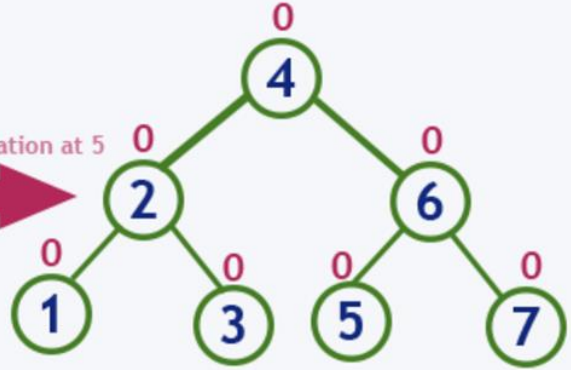


Tree is imbalanced



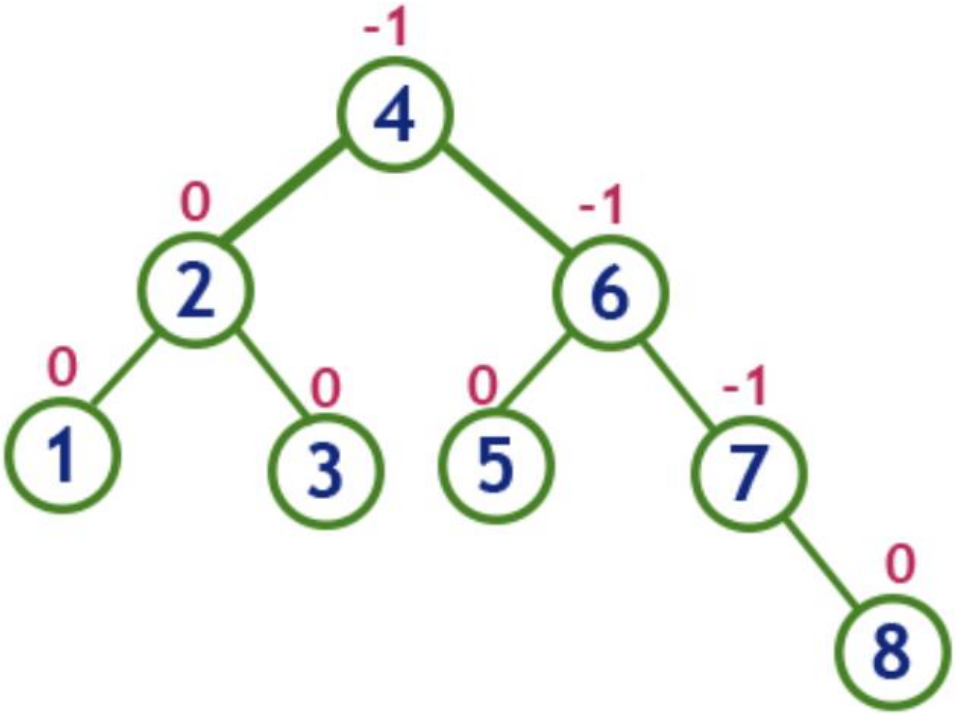
LL Rotation at 5

After LL Rotation at 5



Tree is balanced

insert 8



Tree is balanced

Deletion in AVL trees

The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition.

If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

B-trees

In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children.

But there is a special type of search tree called B-Tree in which a node contains more than one value (key) and more than two children.

B-Tree was developed in the year 1972 by **Bayer and McCreight** with the name ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

Here, the number of keys in a node and number of children for a node depends on the order of B-Tree. Every B-Tree has an order.

B-Tree of Order m has the following properties...

- **Property #1** - All **leaf nodes** must be **at same level**.
- **Property #2** - All nodes except root must have at least **[m/2]-1** keys and maximum of **m-1** keys.
- **Property #3** - All non leaf nodes except root (i.e. all internal nodes) must have at least **m/2** children.
- **Property #4** - If the root node is a non leaf node, then it must have **atleast 2** children.
- **Property #5** - A non leaf node with **n-1** keys must have **n** number of children.
- **Property #6** - All the **key values in a node** must be in **Ascending Order**.

Insertion in B-tree

In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only. The insertion operation is performed as follows...

- Step 1 - Check whether tree is Empty.
- Step 2 - If tree is **Empty**, then create a new node with new key value and insert it into the tree as a root node.
- Step 3 - If tree is **Not Empty**, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
- Step 4 - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
- Step 5 - If that leaf node is already full, **split** that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
- Step 6 - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example

Construct a B-Tree of Order 3 by inserting numbers from 1 to 10.

Construct a B-Tree of order 3 by inserting numbers from 1 to 10.

insert(1)

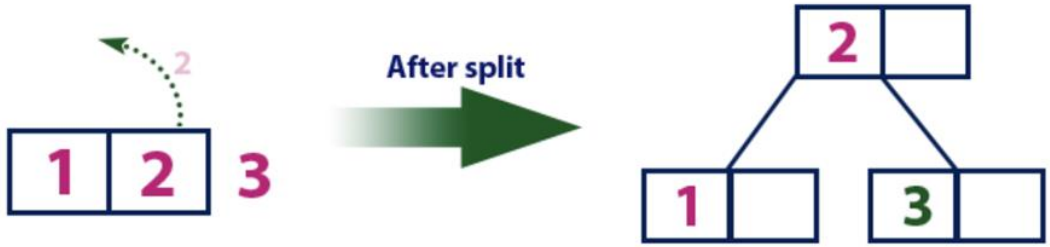
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

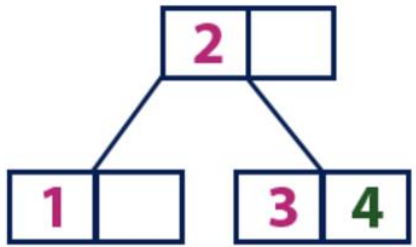
insert(3)

Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.

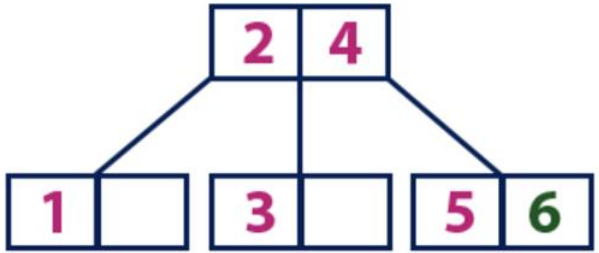


insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

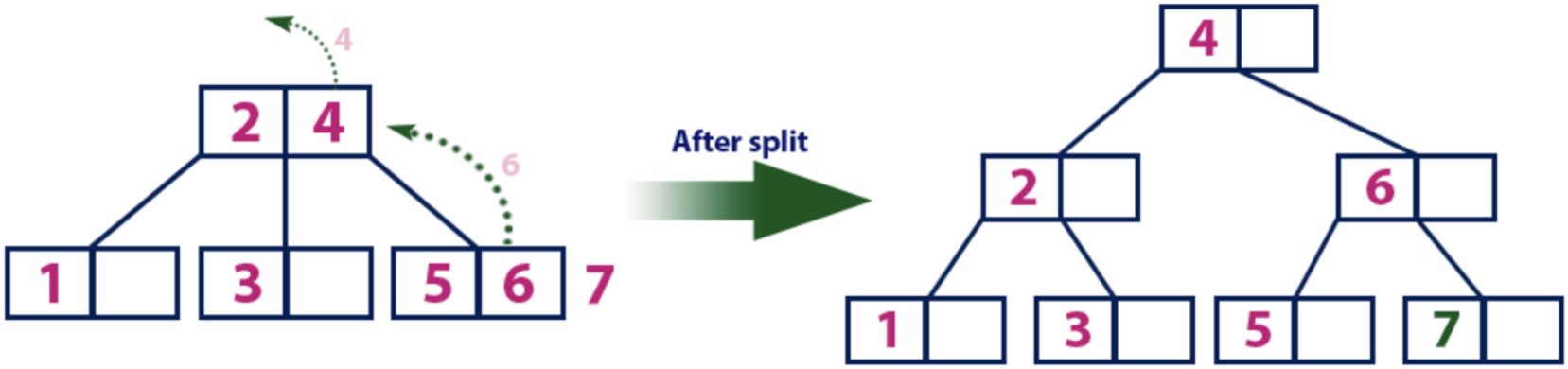
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



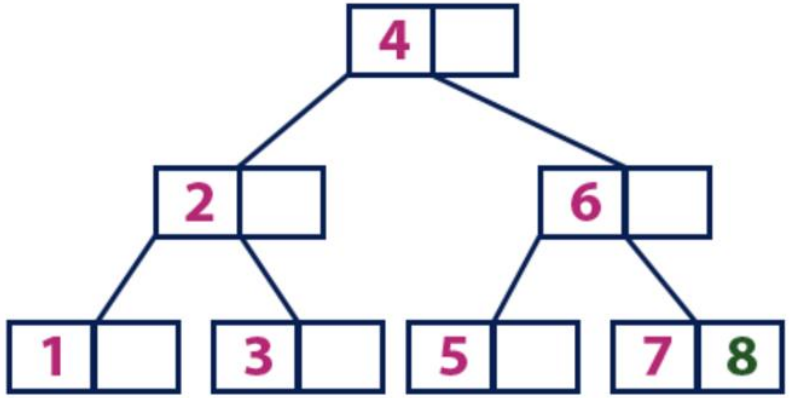
insert(7)

Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



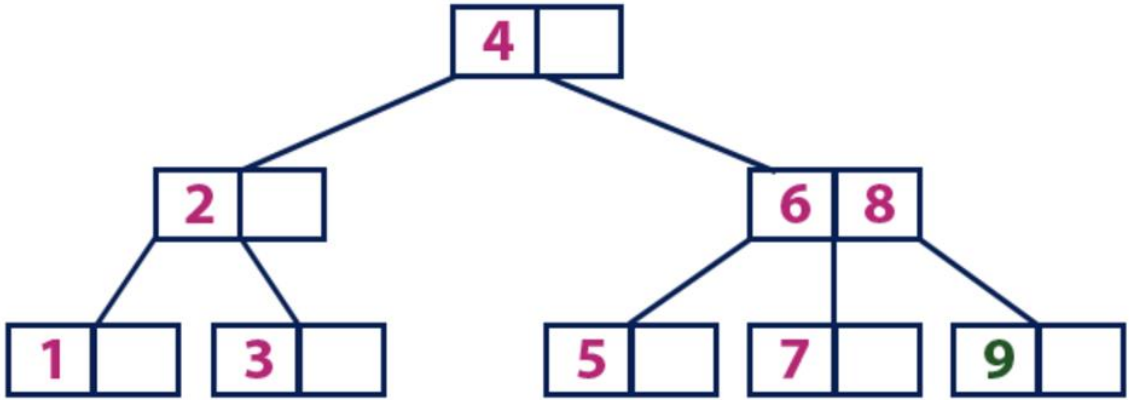
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



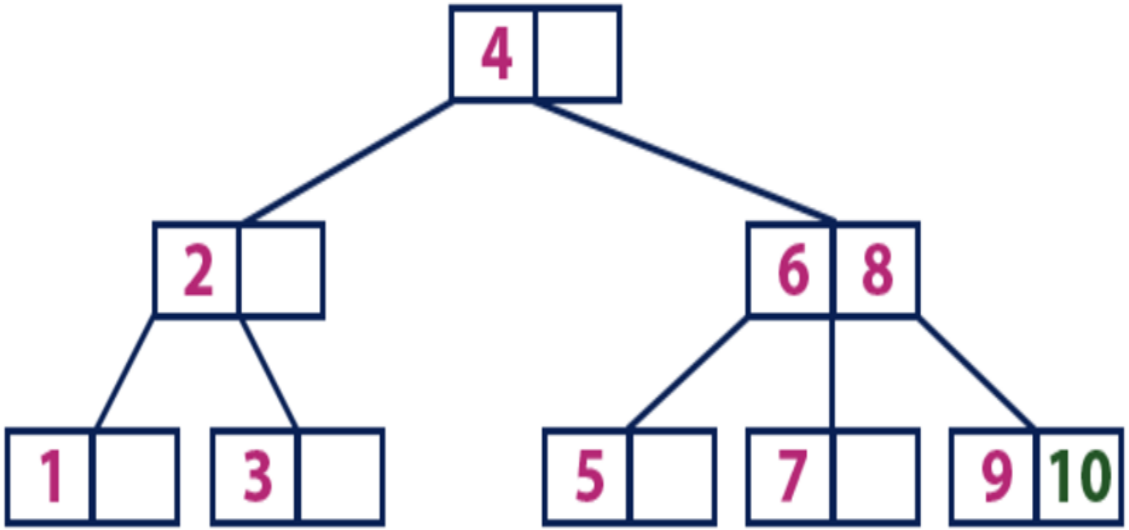
insert(9)

Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



Huffman Algorithm

Huffman code is a technique for compressing data. Huffman's greedy algorithm looks at the occurrence of each character and it as a binary string in an optimal way.

Huffman coding

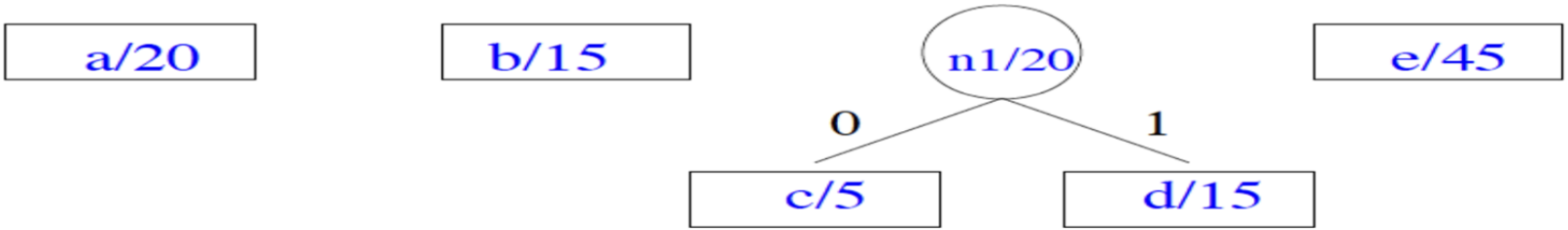
Step 1: Pick two letters x and *y* from Alphabet A with the smallest frequencies and create a subtree that has these two characters as leaves. (greedy idea) Label the root of this subtree as *z.*

Step 2: Set frequency *f(z) = f(x) + f(y) :*Remove *x,y* and add *z* creating new alphabet . $A'' = A \cup \{Z\} - \{x,y\}$

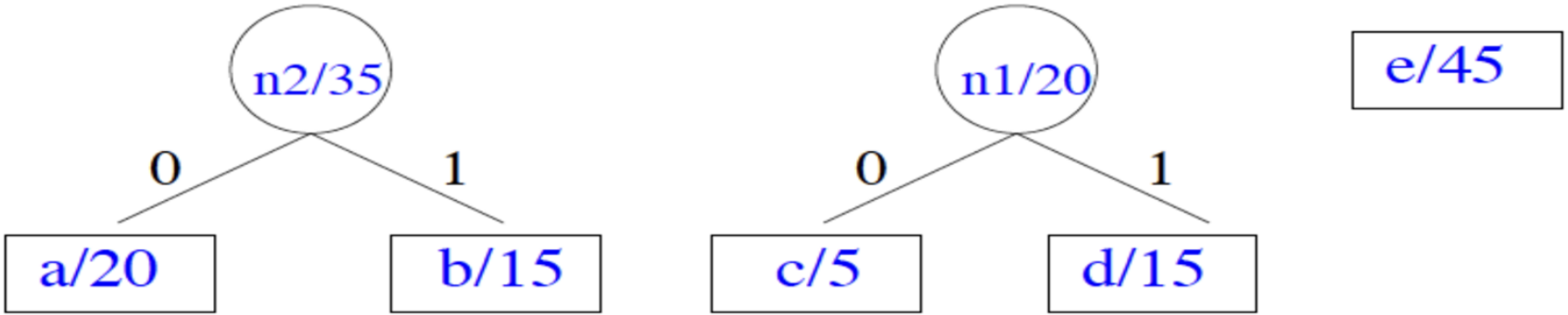
Note that mod$|A''| = |A| - 1$. Repeat this procedure, called merge, with new alphabet A″ until an alphabet with only one symbol is left. Then the resulting tree is the Huffman.

Example of Huffman Coding

Let A = {a/20, b/15, c/5, d/15, e/45} be the alphabet and its frequency distribution. In the first step Huffman coding **merge c and d.**

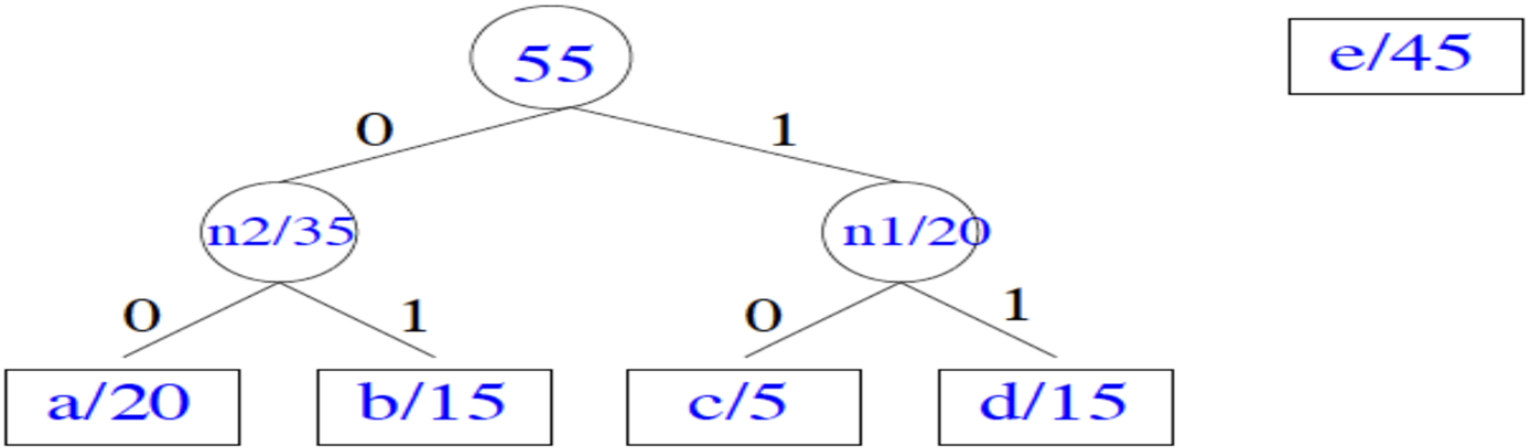


Alphabet is now A1 = {a/20, b/15, n1/20,e/45}. Algorithm **merges a and b**

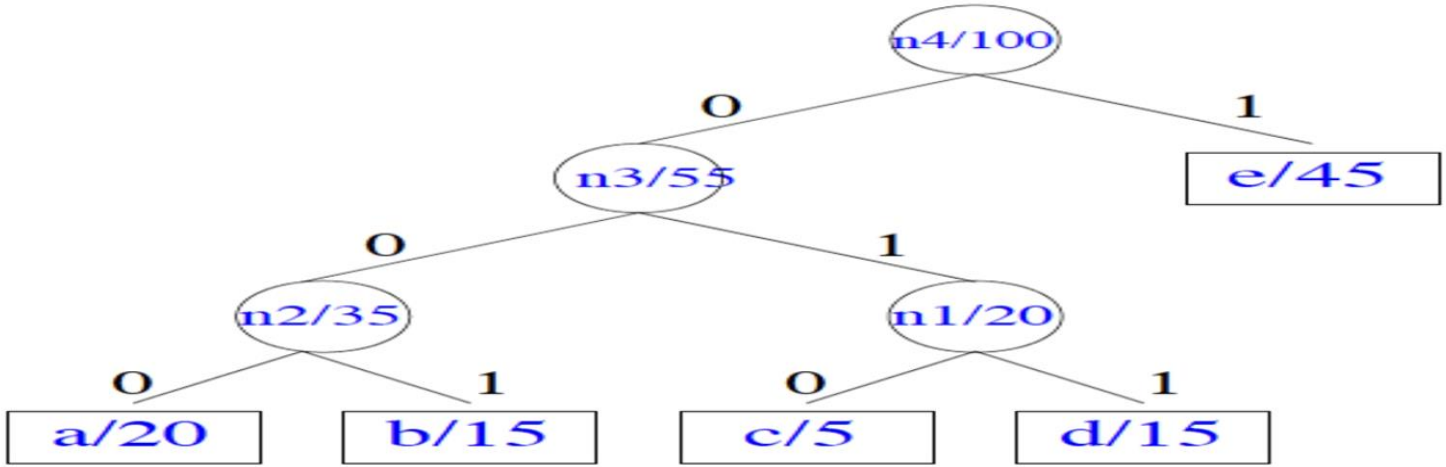


Alphabet is now A2 = { n2/35,n1/20, e/45}. Algorithm **merge n1 and n2**.

Alphabet is now A2 = { n2/35,n1/20, e/45}. Algorithm **merge n1 and n2**.



Alphabet is now A3 = {n3/55, e/45}. Algorithm merges n3 and e and finished.

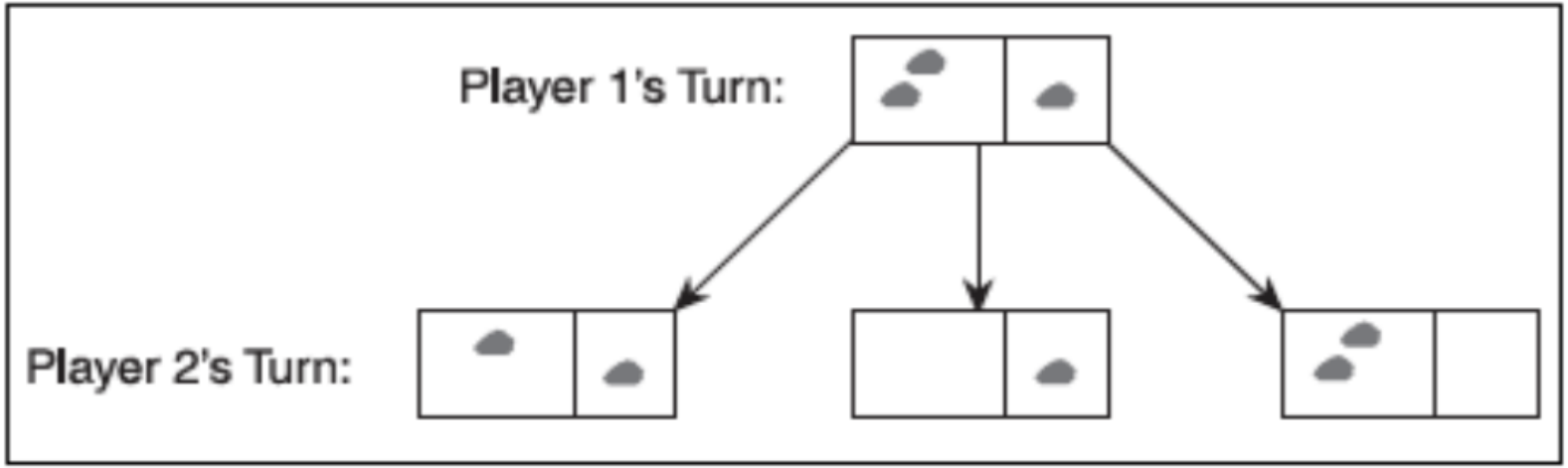


Huffman code a=000, b=001, c = 010, d= 011, e=1

Game Tree

A game tree isn’t a special new data structure—it’s a name for any regular tree that maps how a discrete game is played.

Lets take an example of game called Rocks. In this game, we have different piles of rocks, with one or more rocks in each pile. The game has two players, who take turns taking one or more rocks from a single pile until one pile is left. When one pile is left, our goal is to force our opponent to remove the last rock. The person who removes the last rock loses.


Player 1's Turn:
Player 2's Turn:

In the figure, there are two piles. The first pile has two rocks, and the second pile has only one rock. The first player has three choices:

- Remove one rock from pile 1.
- Remove two rocks from pile 1.
- Remove one rock from pile 2.

We can start the game off with one of those three moves. We can create a simple game tree to represent these moves, as shown in above fig.

After Player 1 has moved, it is now Player 2's turn. Player2's choice of a move is limited to the current state of the game, however.

In the leftmost state of above figure, Player 2 has two choices:

He can remove one rock from pile 1 or one rock from pile 2.

His choice for the middle state is even less useful He can only remove one rock from pile 2.

Of course, because this is the last rock, Player 2 has lost the game.

On the right state, Player 2 has two options again: He can remove one or two rocks from pile 2.

Below figure shows the game tree for all five of these moves and goes down one more level to show the complete game tree.

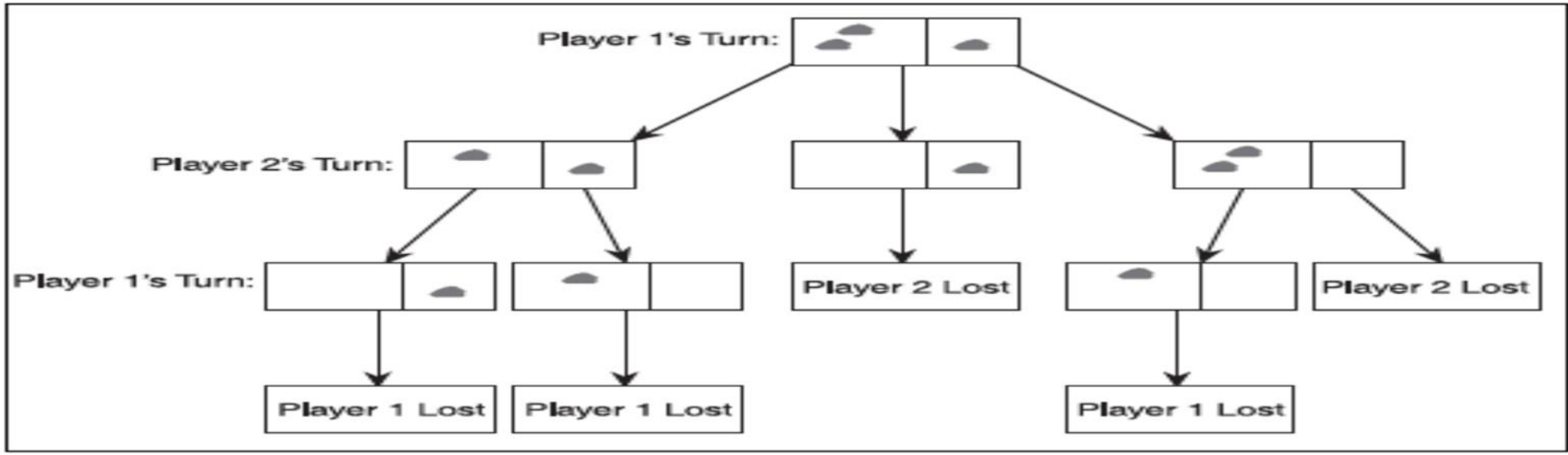


Fig26: A Complete Game tree

In Short, following are the common uses of tree data structure.

1. Manipulate hierarchical data.
2. Make information easy to search.
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects. 5. Router algorithms

Other Applications

1. *Binary Search Tree*: Used in many search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.

2. *Hash Trees* - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.

3. *Heaps* - Used in heap-sort; fast implementations of Dijkstra's algorithm; and in implementing efficient priority-queues, which are used in scheduling processes in many operating systems, Quality-of-Service in routers, and A* (path-finding algorithm used in AI applications, including video games).

4. *Huffman Coding Tree* - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

5. Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.

6. Treap - Randomized data structure used in wireless networking and memory allocation.

7. T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so