SOMAIYA VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Trees

swatimali@somaiya.edu

Somaiya TRUST

Outline

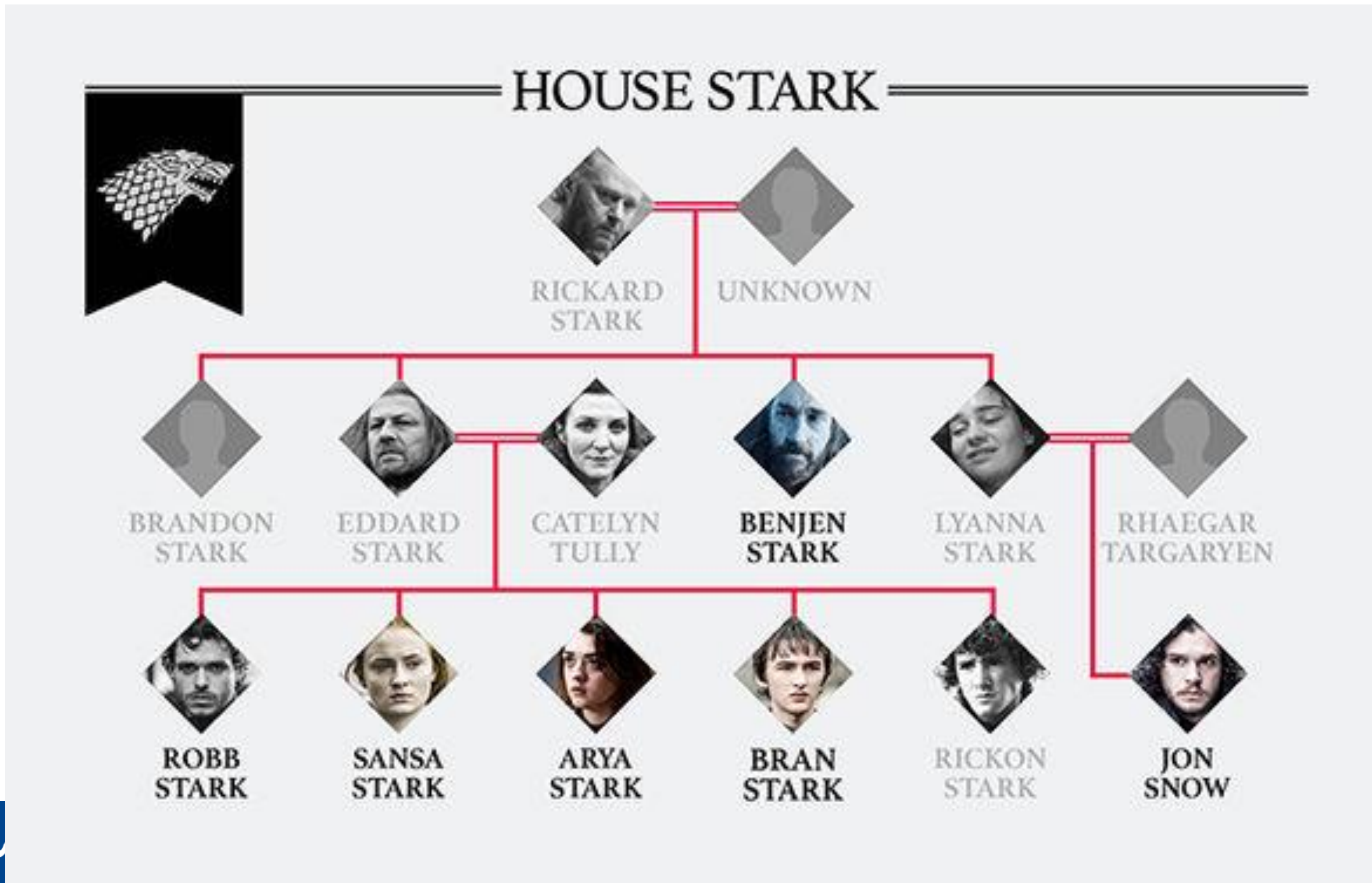
- Tree – concept
- General tree
- Types of trees
- Binary tree: representation, operation
- Binary tree traversal
- Binary search tree
- BST- The data structure and implementation
- Threaded binary trees.
- Search Trees –
  - AVL tree, Multiway Search Tree, B Tree, B+ Tree, and Trie,
- Applications/Case study of trees.
- Summary

Queries?

Tree

- linear data structures – strings, arrays, lists, stacks and queues
- Non-linear data structure - tree.
- Mainly used to represent data containing a hierarchical relationship between elements, for example, records, family trees and table of contents.
  - E.g. a parent-child relationship

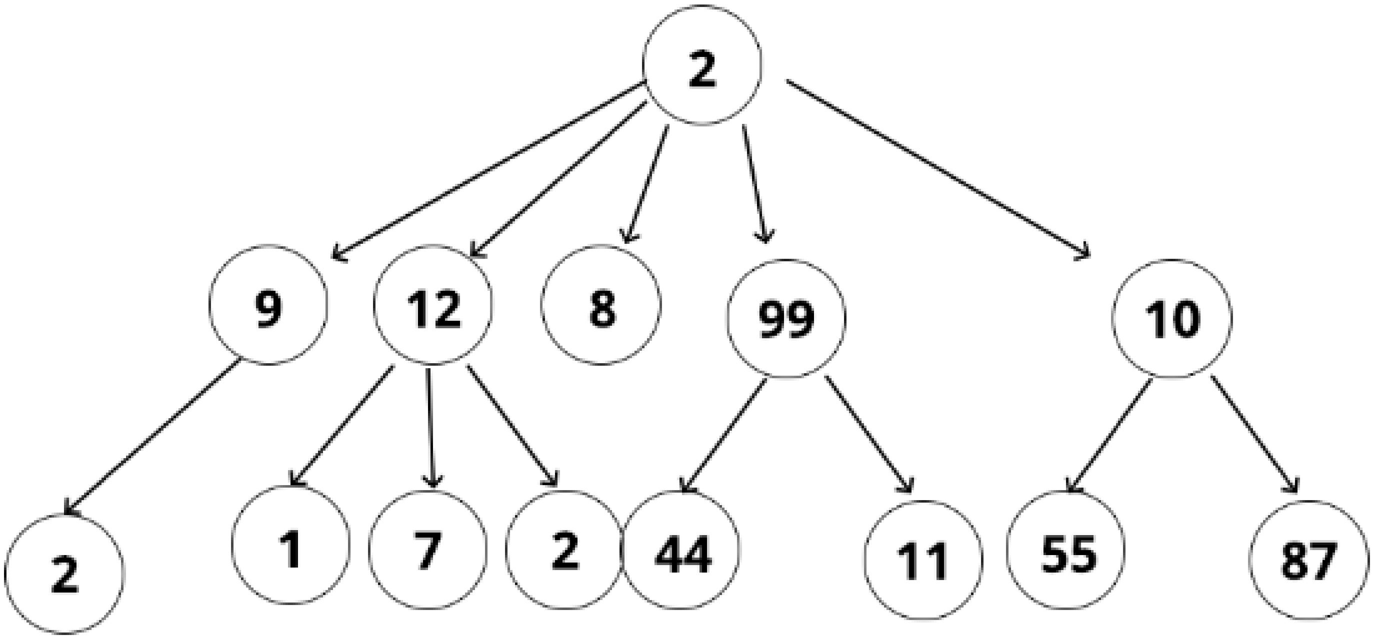
A family tree



SOMAIYA VIDYAVIHAR UNIVERSITY

K J So

TREES

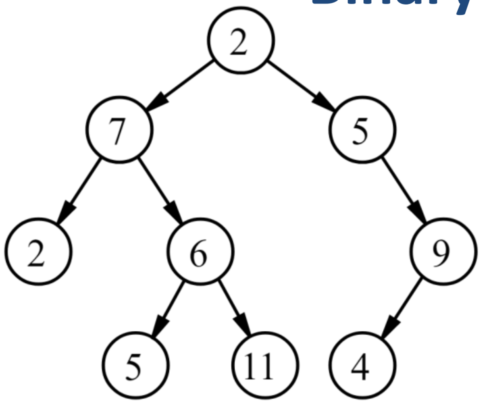


Somaiya TRUST

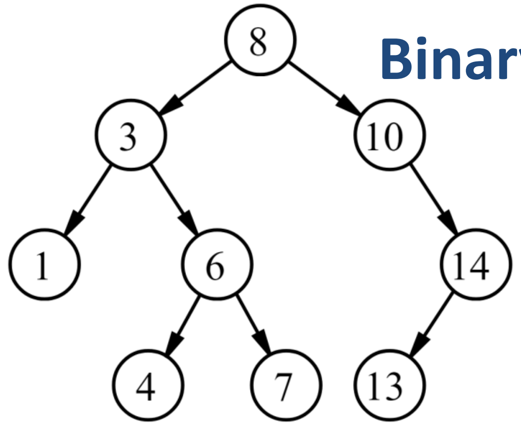
Types of trees

- General tree
- Binary tree
- Binary search tree
- Threaded binary tree
- AVL Tree
- B tree
- B+ Tree
- Trie
- Heap
- Red black tree
- Splay tree

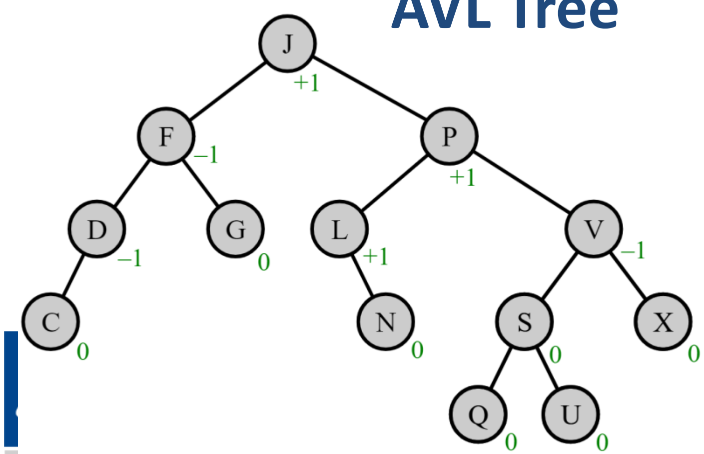
Binary Tree



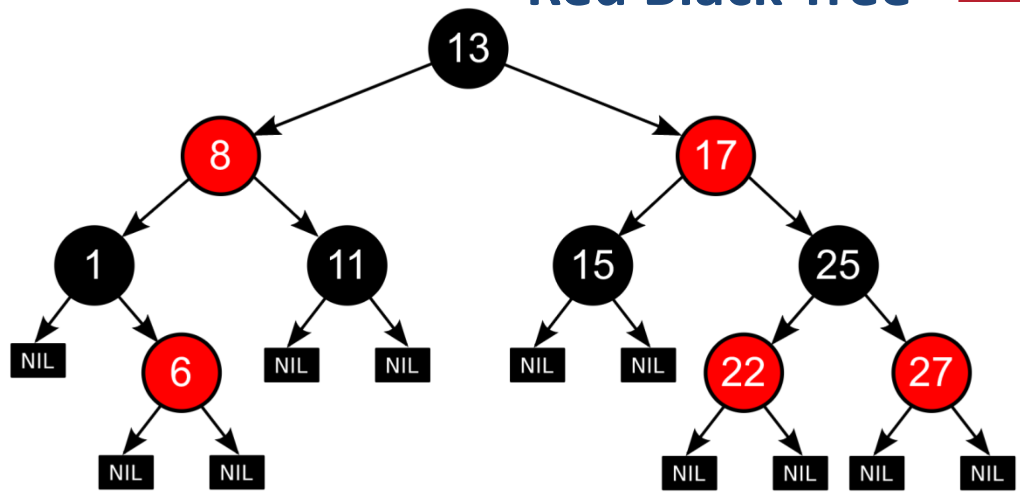
Binary search Tree



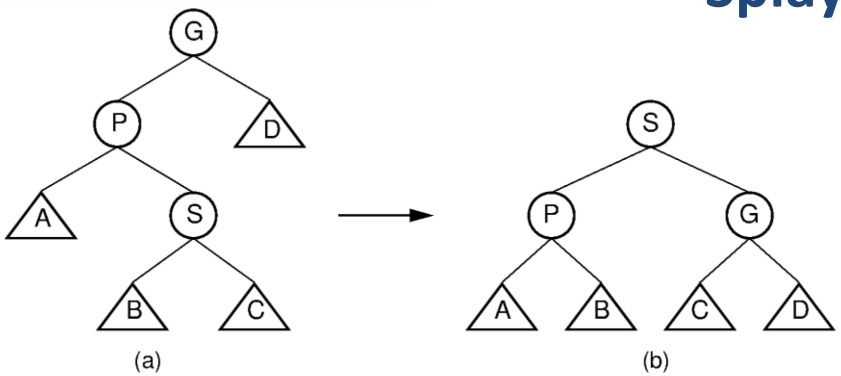
AVL Tree

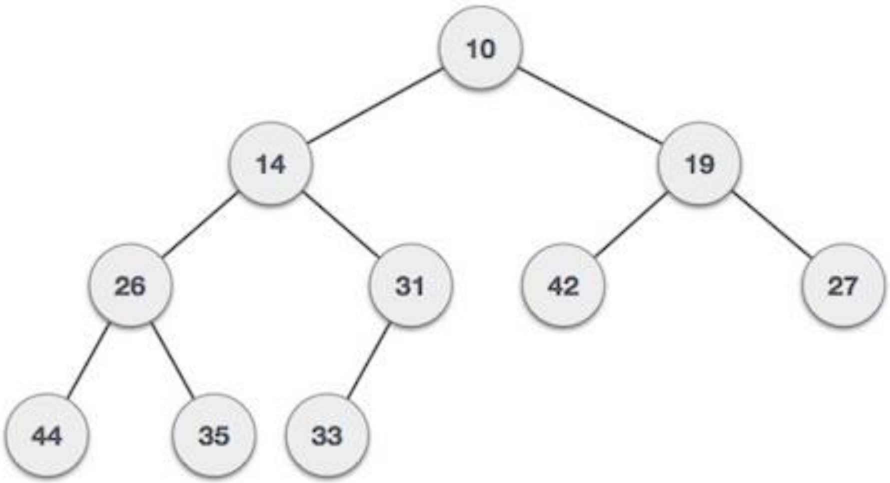


Red Black Tree

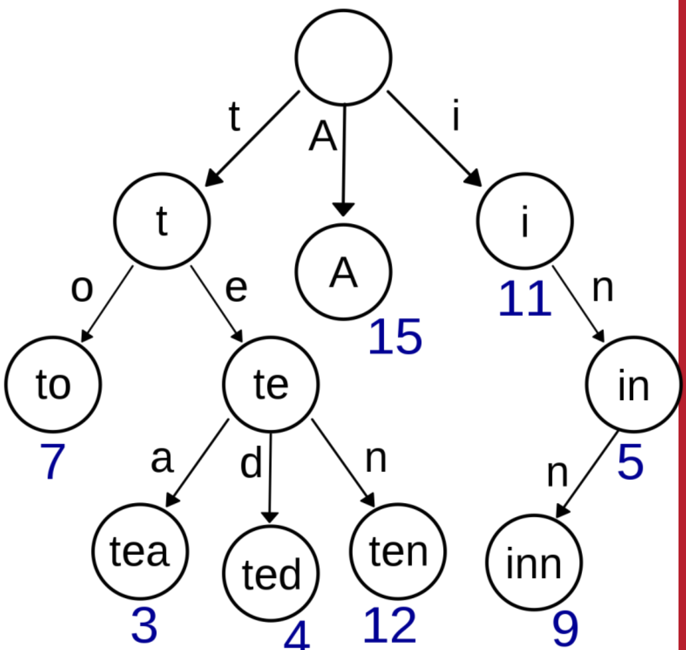


Splay Tree

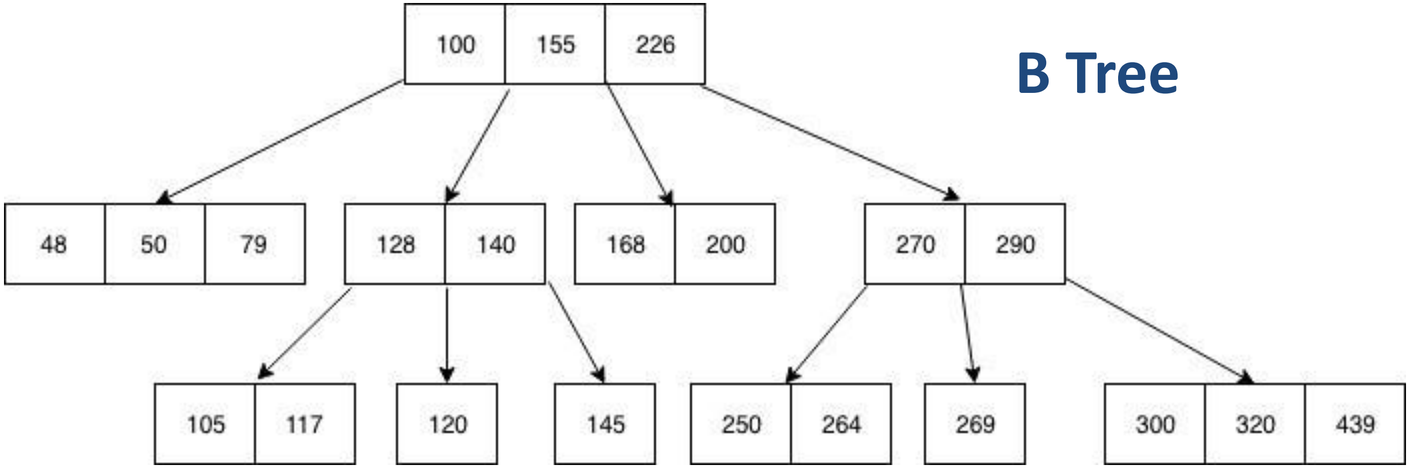




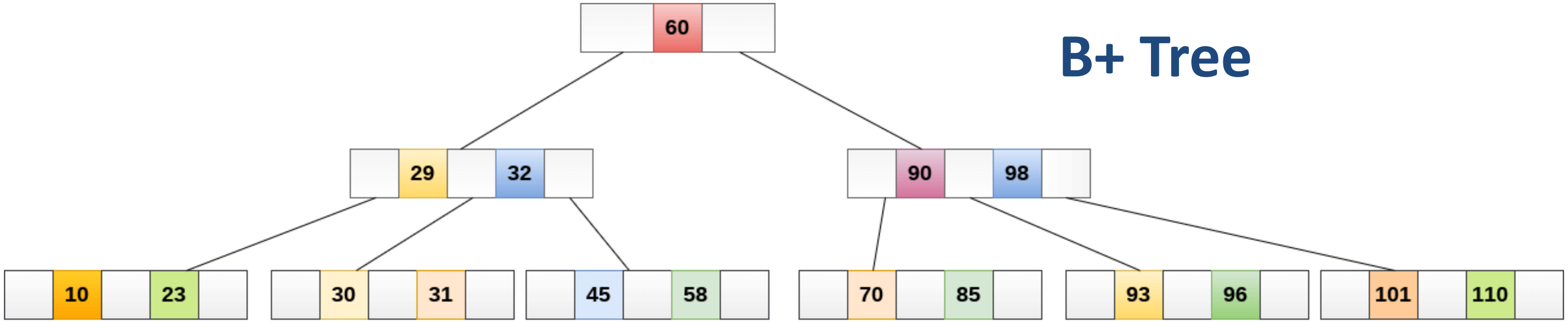
Heap (MinHeap)



Trie


100
155
226
48
50
79
128
140
168
200
270
290
105
117
120
145
250
264
269
300
320
439

B Tree


60
29
32
90
98
10
23
30
31
45
58
70
85
93
96
101
110

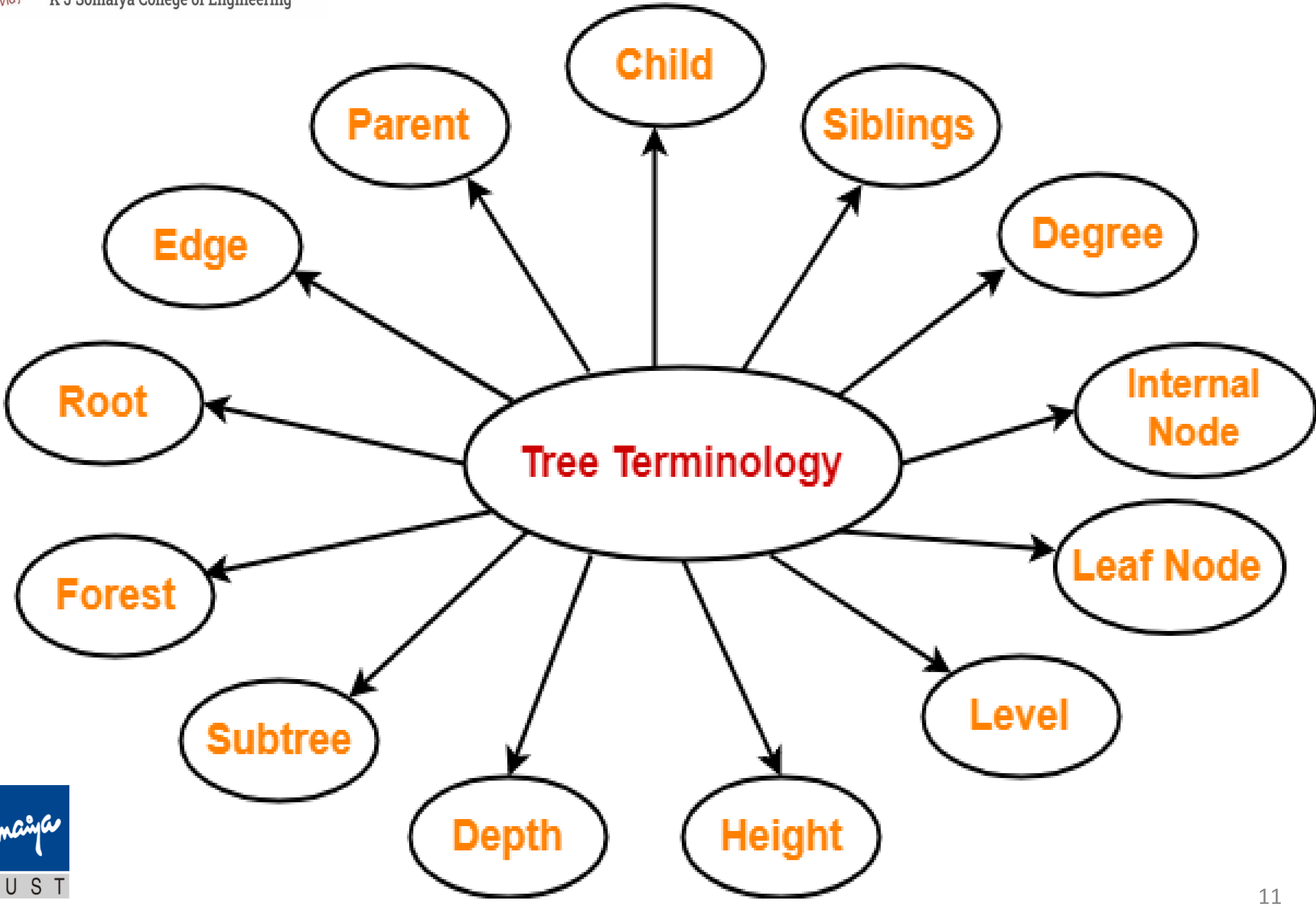
B+ Tree

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Tree Data structure

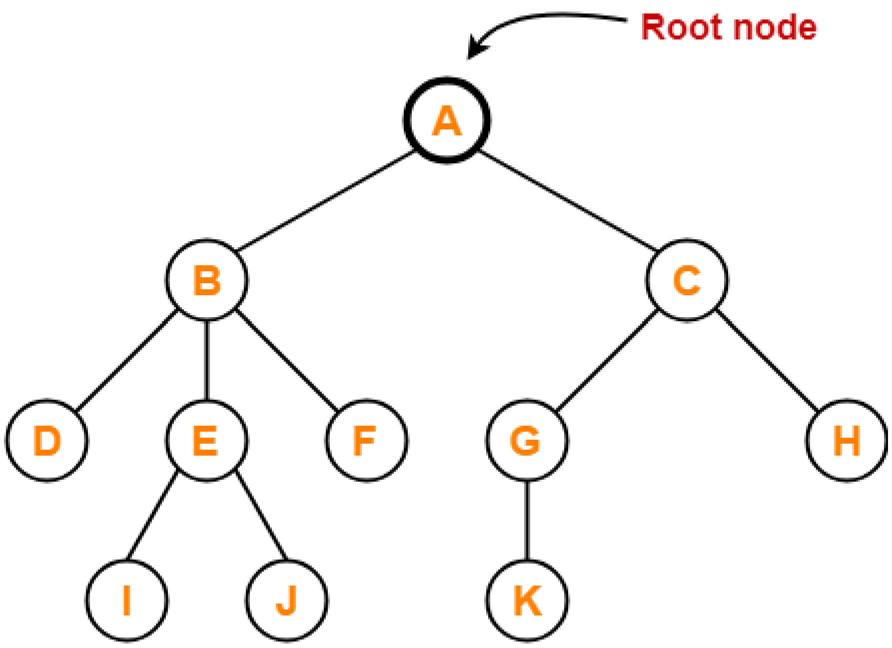
- A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.
- Tree is a sequence of nodes
- There is a starting node known as a root node
- Every node other than the root has a parent node.
- Nodes may have any number of children

Somaiya TRUST


Tree Terminology
Child
Parent
Siblings
Edge
Degree
Root
Internal Node
Forest
Leaf Node
Subtree
Level
Depth
Height

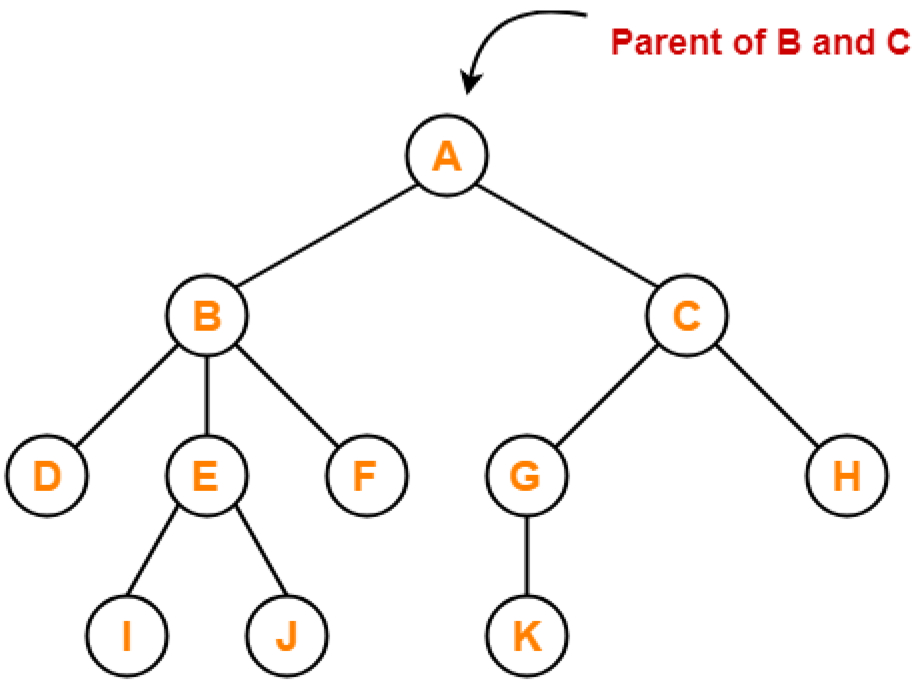
1. Root

- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



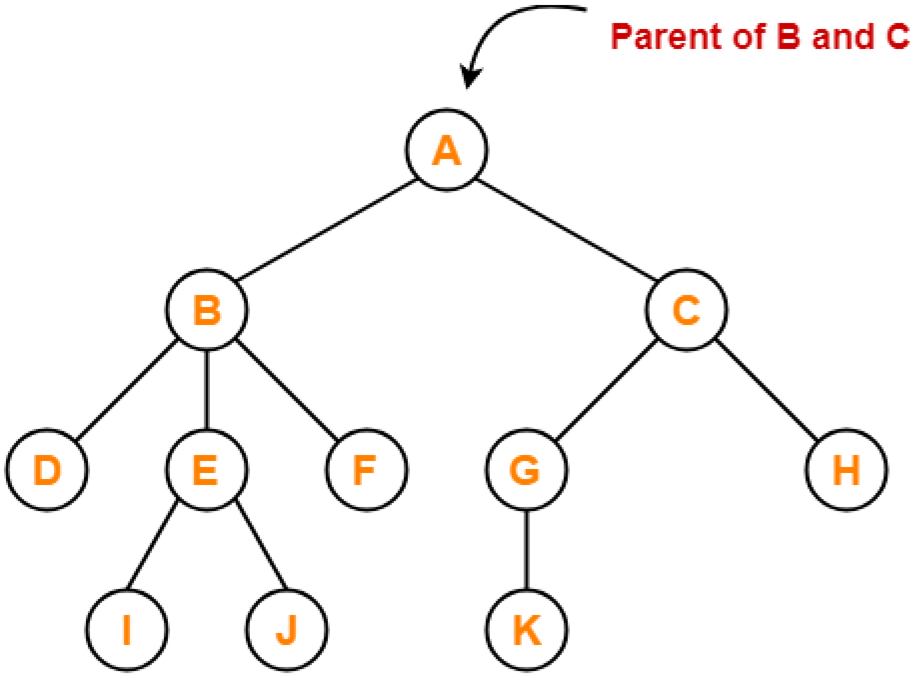
2. Edge

- The connecting link between any two nodes is called as an edge.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.



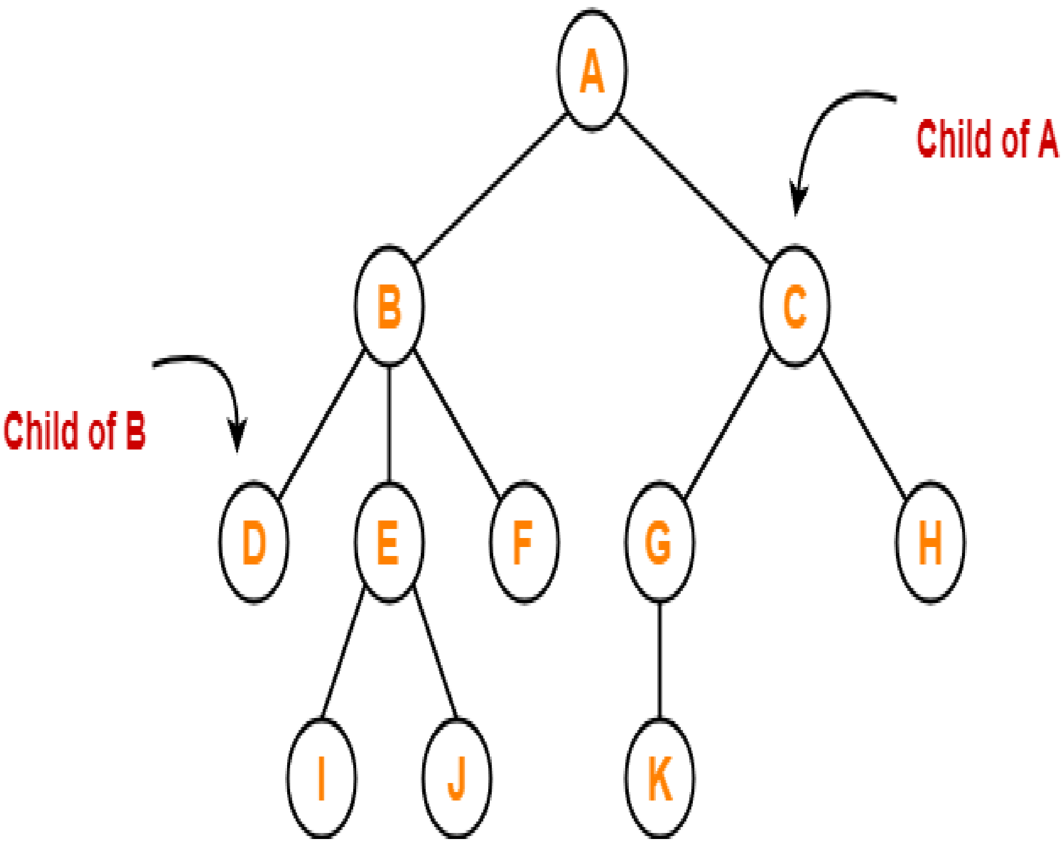
3. Parent

- The node which has a branch from it to any other node is called as a parent node.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



4. Child

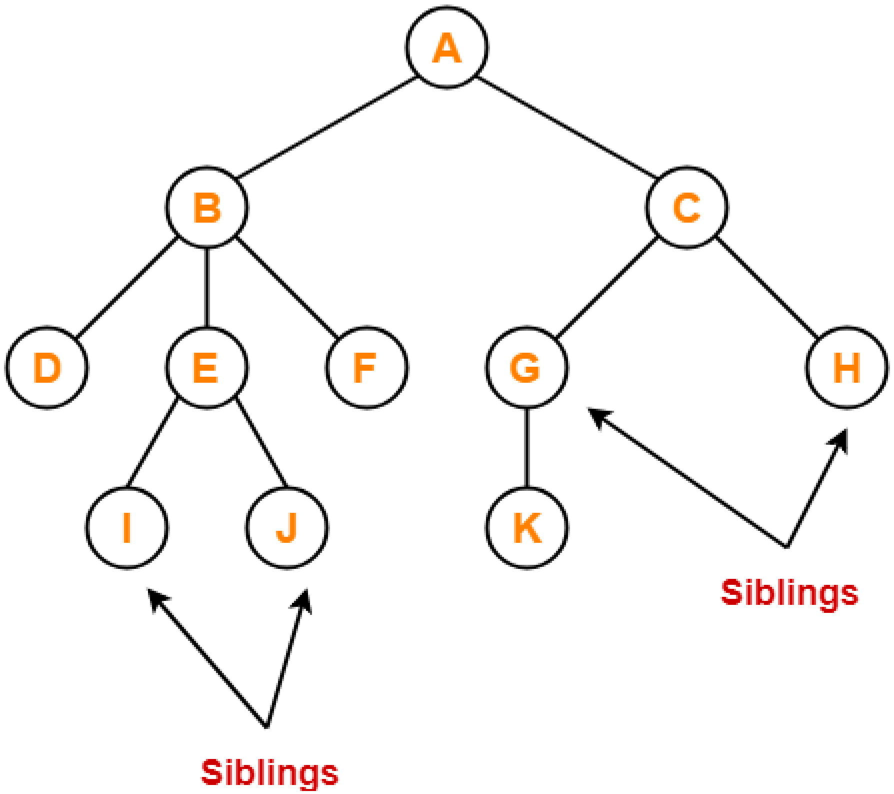
- The node which is a descendant of some node is called as a child node.
- All the nodes except root node are child nodes.



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

5. Siblings

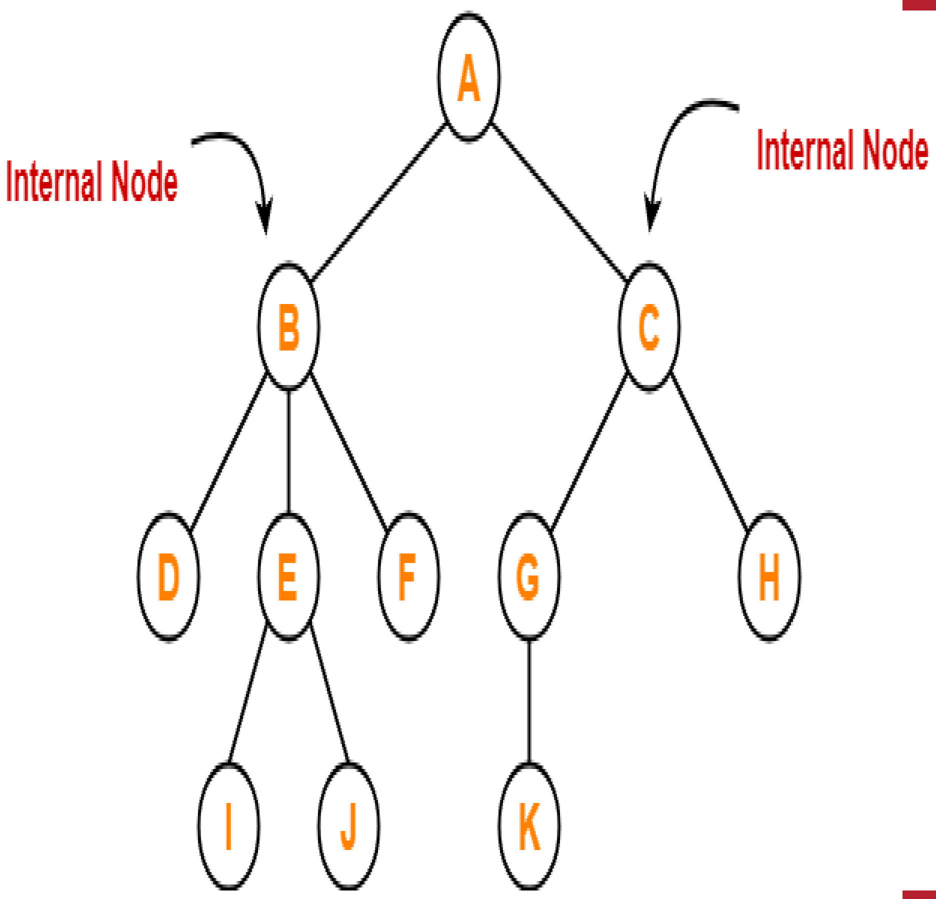
- Nodes which belong to the same parent are called as siblings.
- In other words, nodes with the same parent are sibling nodes.



Somaiya TRUST

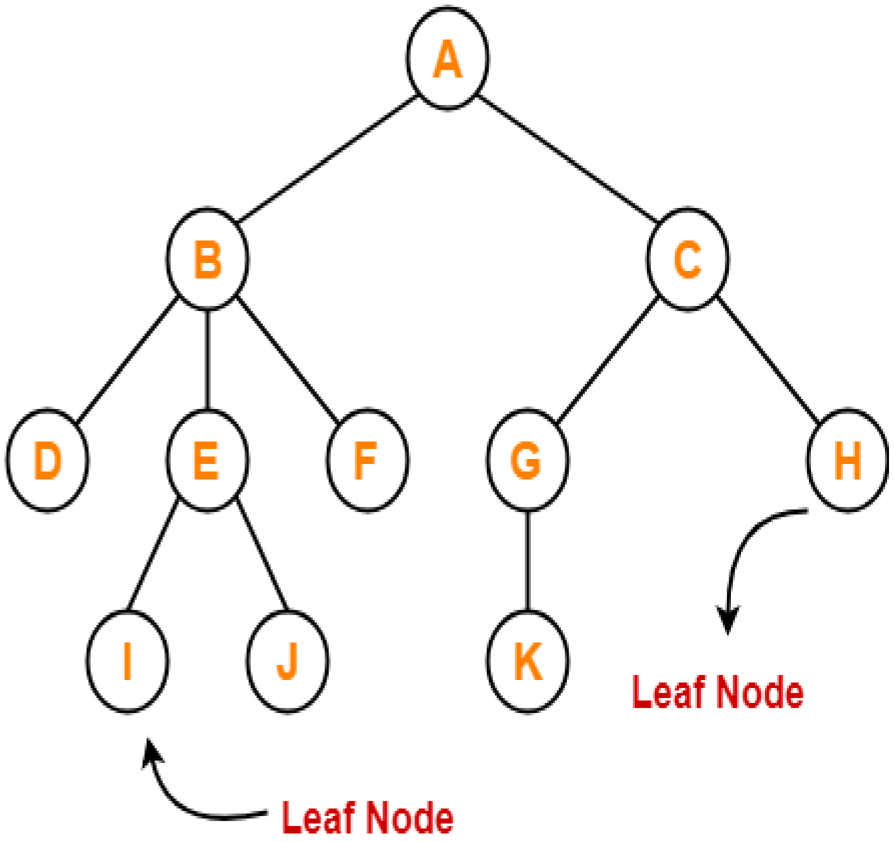
7. Internal Node

- The node which has at least one child is called as an internal node.
- Internal nodes are also called as non-terminal nodes.
- Every non-leaf node is an internal node.



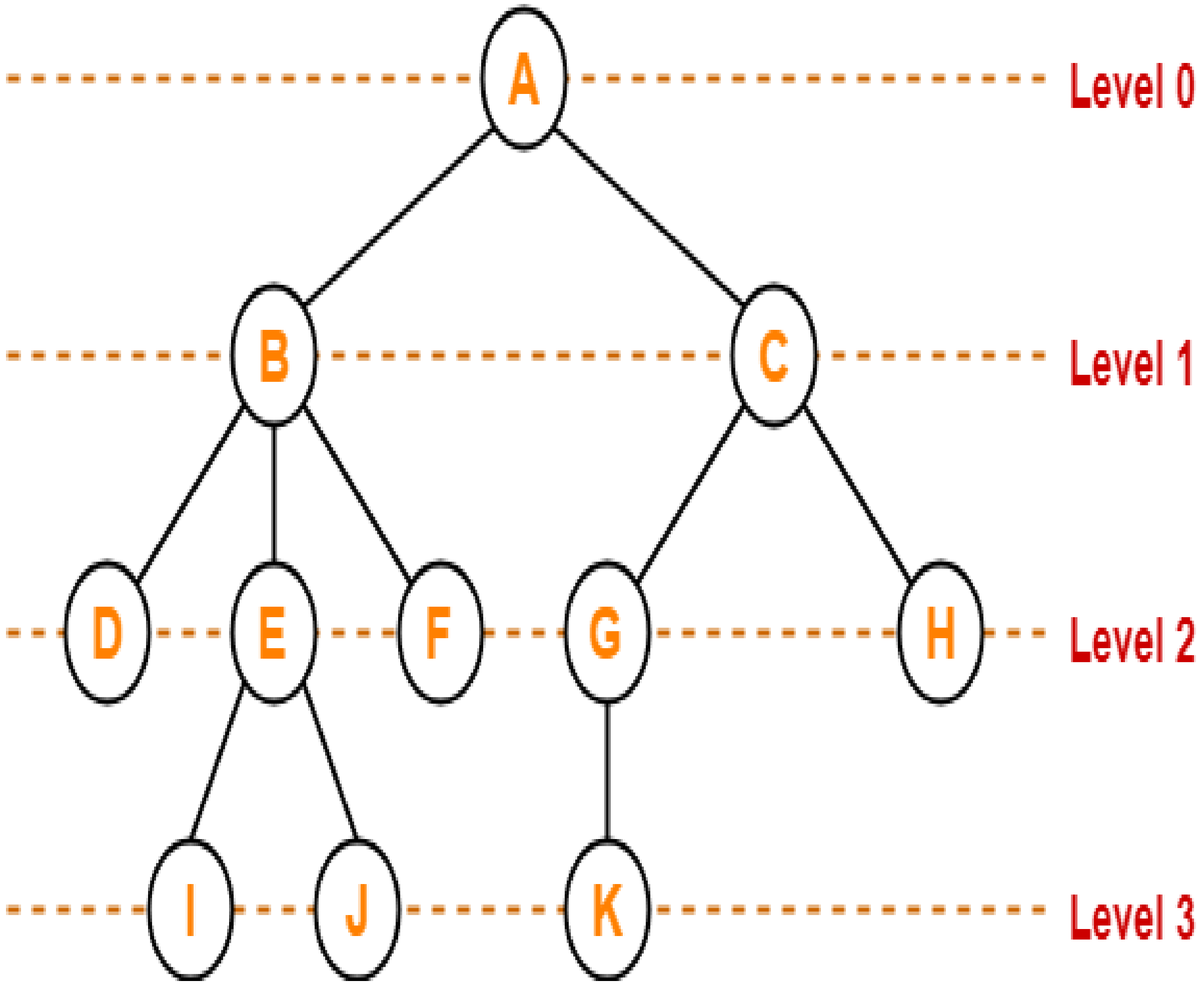
8. Leaf Node

- The node which does not have any child is called as a leaf node.
- Leaf nodes are also called as external nodes or terminal nodes.



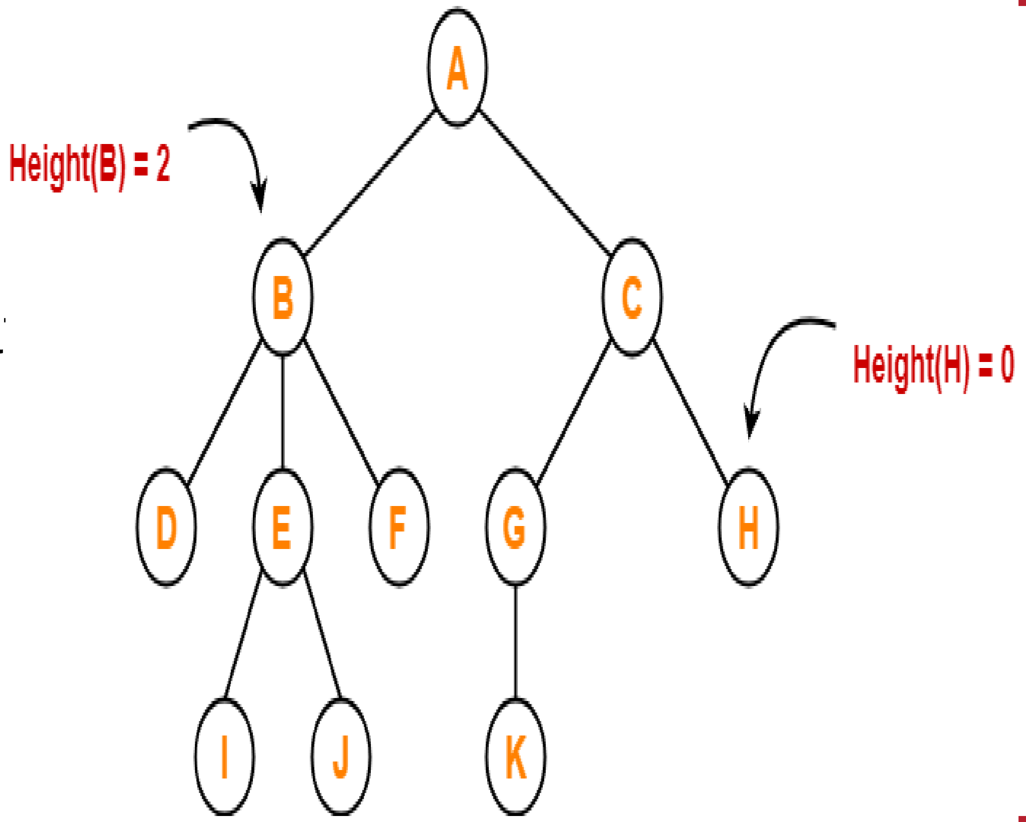
9. Level

- In a tree, each step from top to bottom is called as level of a tree.
- The level count starts with 0 and increments by 1 at each level or step.



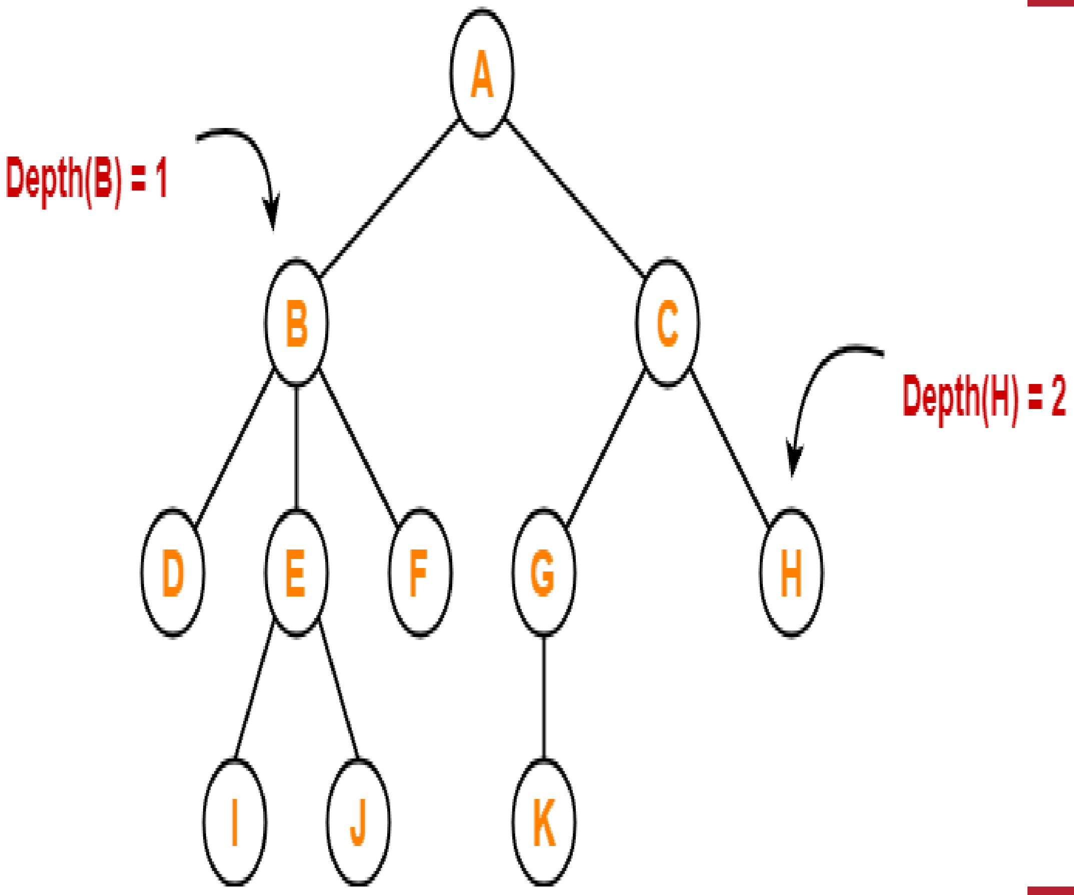
10. Height

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
- Height of a tree is the height of root node.
- Height of all leaf nodes = 0
- <u>Computed from bottom to top</u>



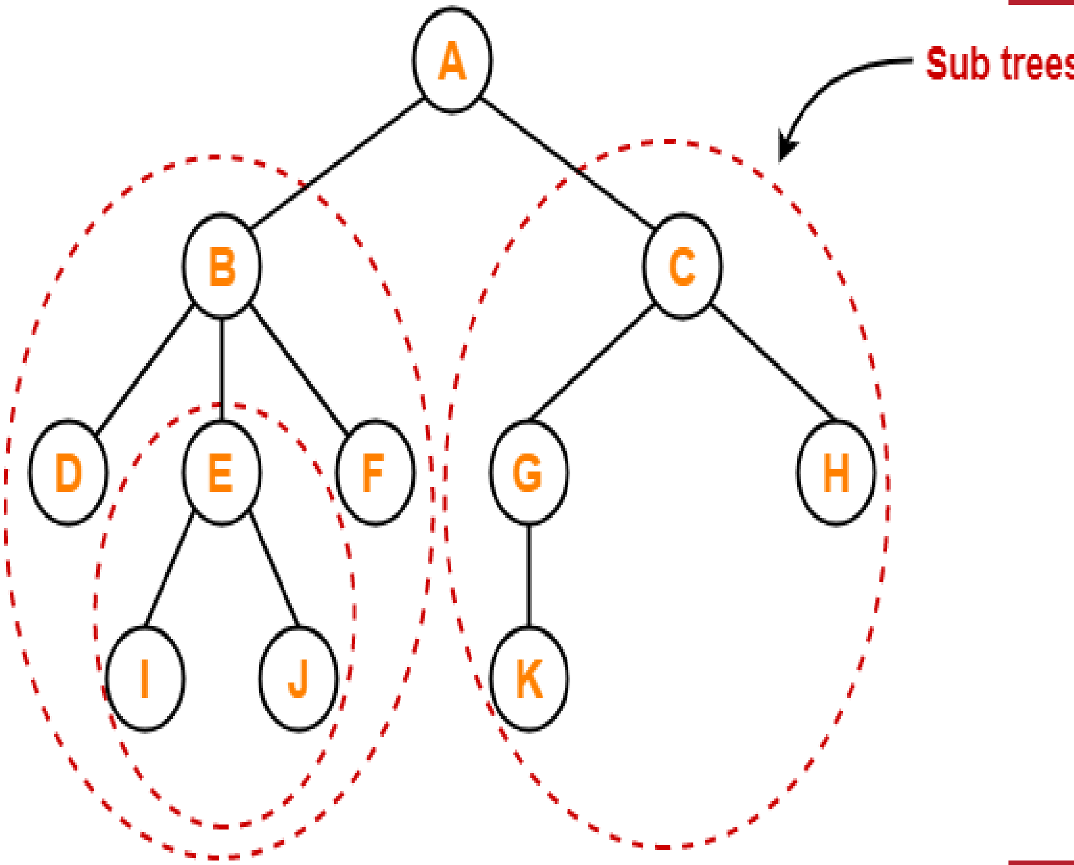
11. Depth

- Total number of edges from root node to a particular node is called as depth of that node.
- Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms "level" and "depth" are used interchangeably.
- <u>Computed from top to bottom</u>



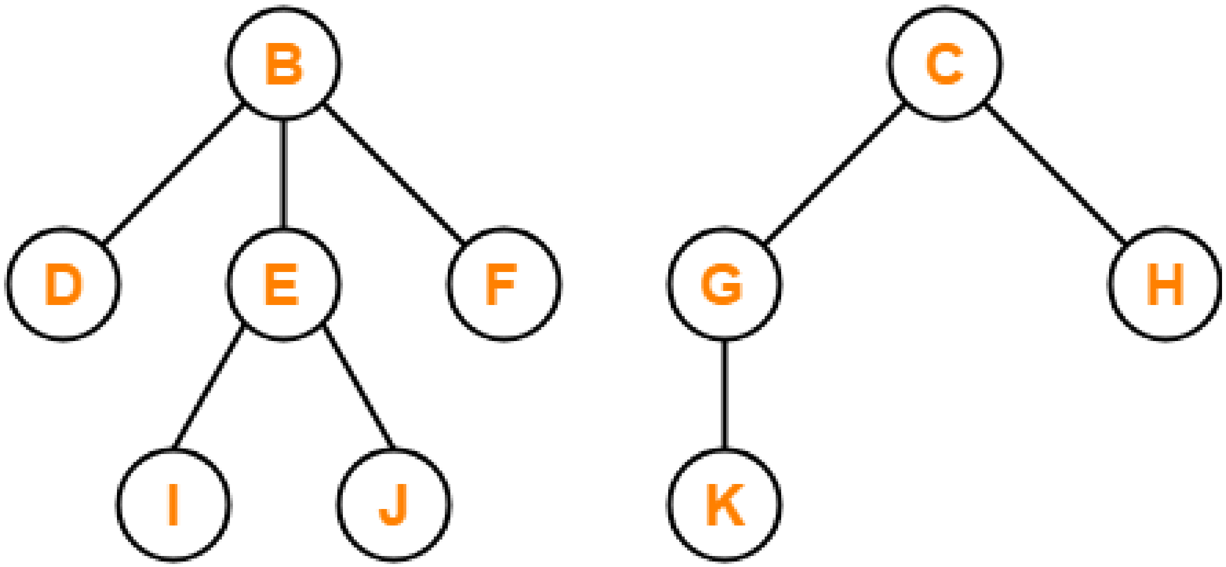
12. Subtree

- In a tree, each child from a node forms a subtree recursively.
- Every child node forms a subtree on its parent node.



13. Forest

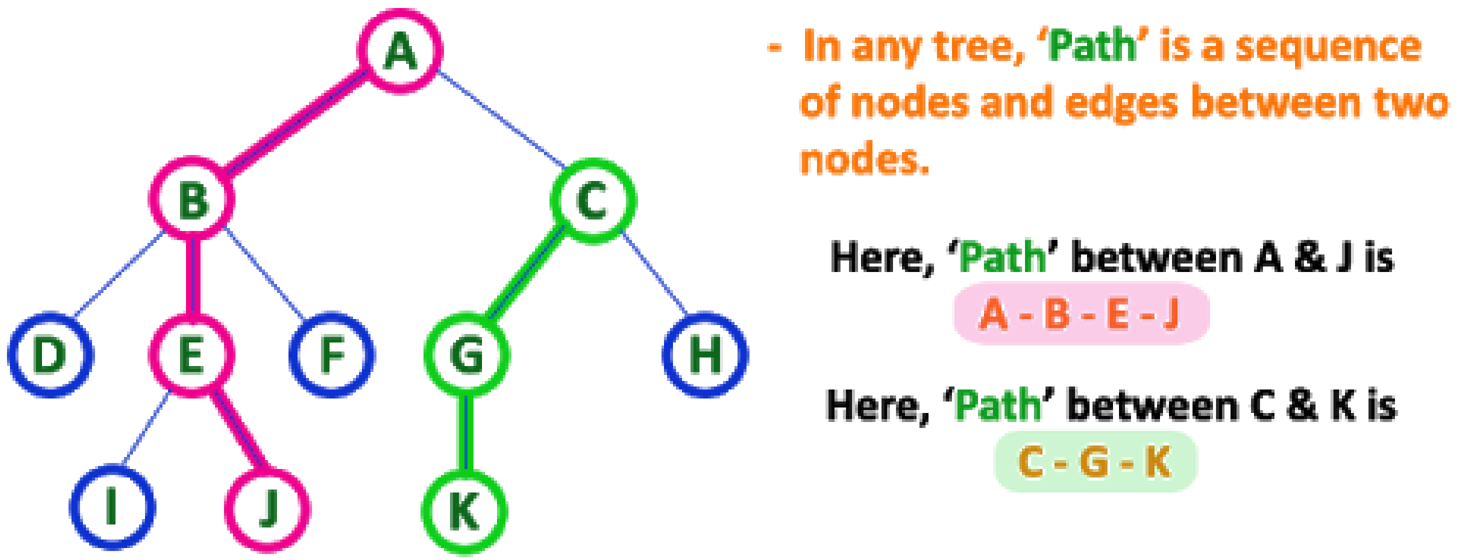
- A forest is a set of disjoint trees.



Forest

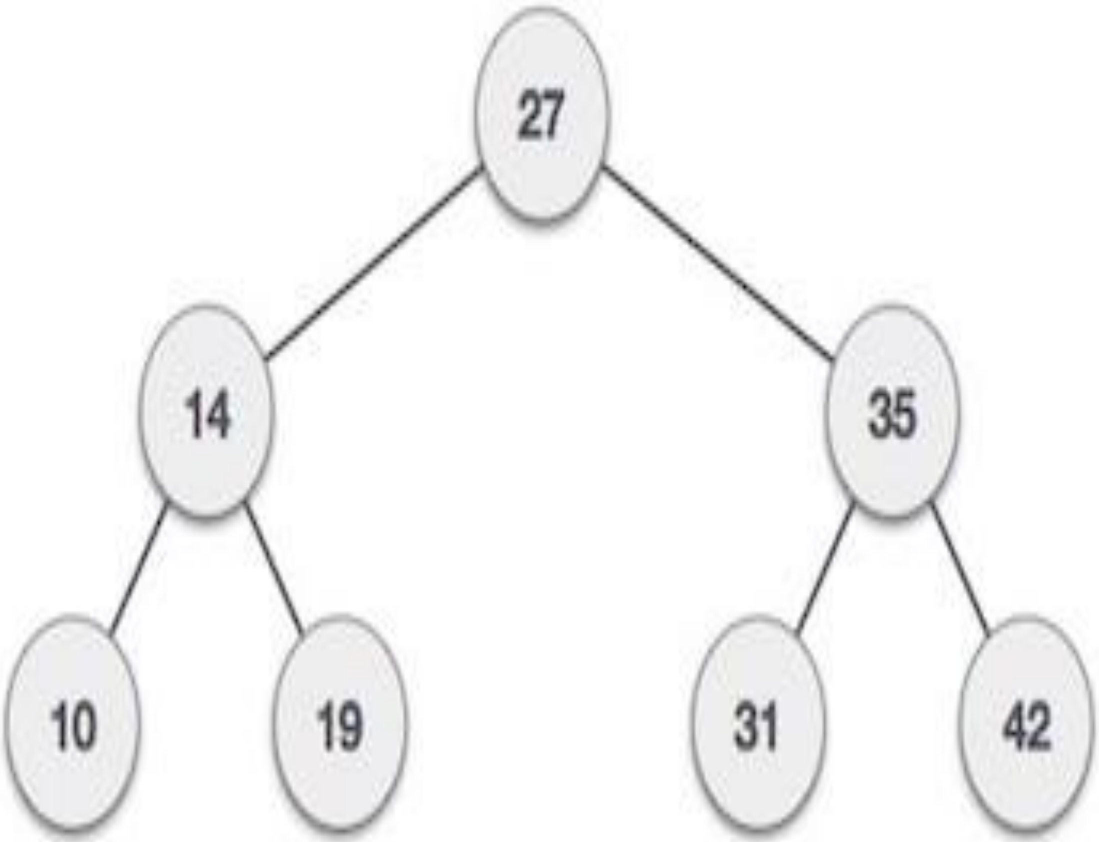
14. Path

- The sequence of consecutive edges from source node to destination node.



15. Keys

- Key represents a value of a node based on which a search operation is to be carried out for a node.



SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Characteristics of trees

- Non-linear data structure
- Combines advantages of an ordered array and linked list
- Searching as fast as in ordered array
- Insertion and deletion as fast as in linked list
- Simple and fast

Somaiya TRUST

Application

- Directory structure of a file storage
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- used in compression algorithms, such as those used by the .jpeg and .mp3 file formats
- Game trees

Directory structure of a file storage

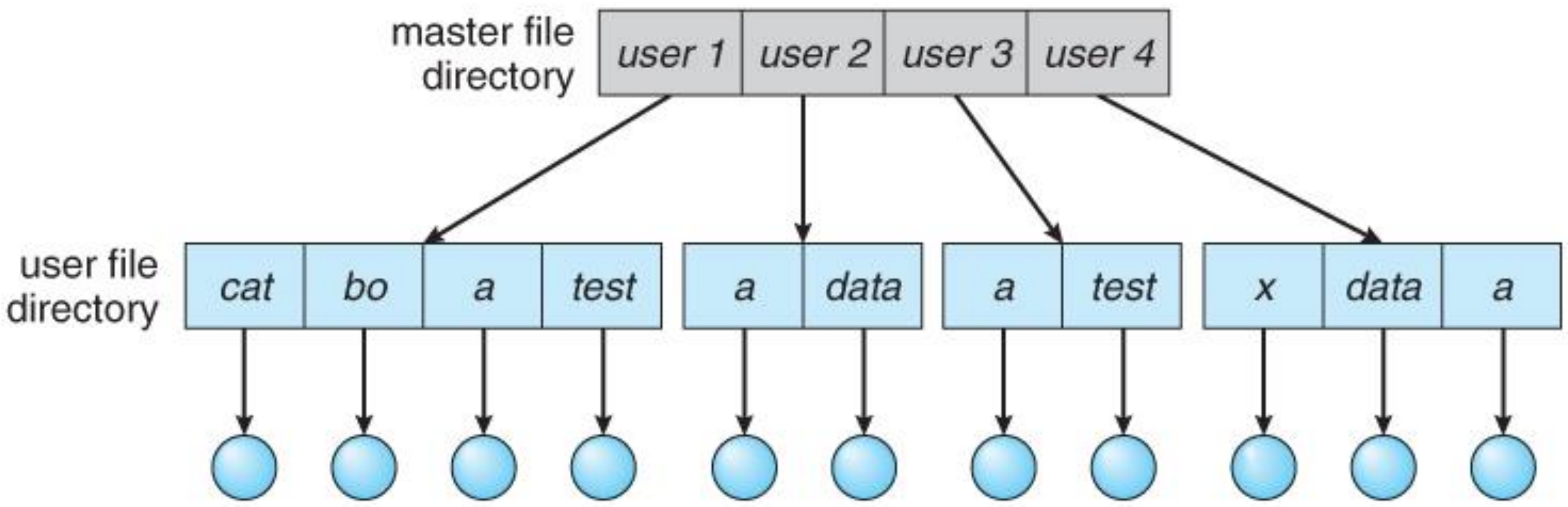


Image courtesy: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/images/Chapter11/11_10_TwoLevelStructure.jpg

Structure of an arithmetic expressions

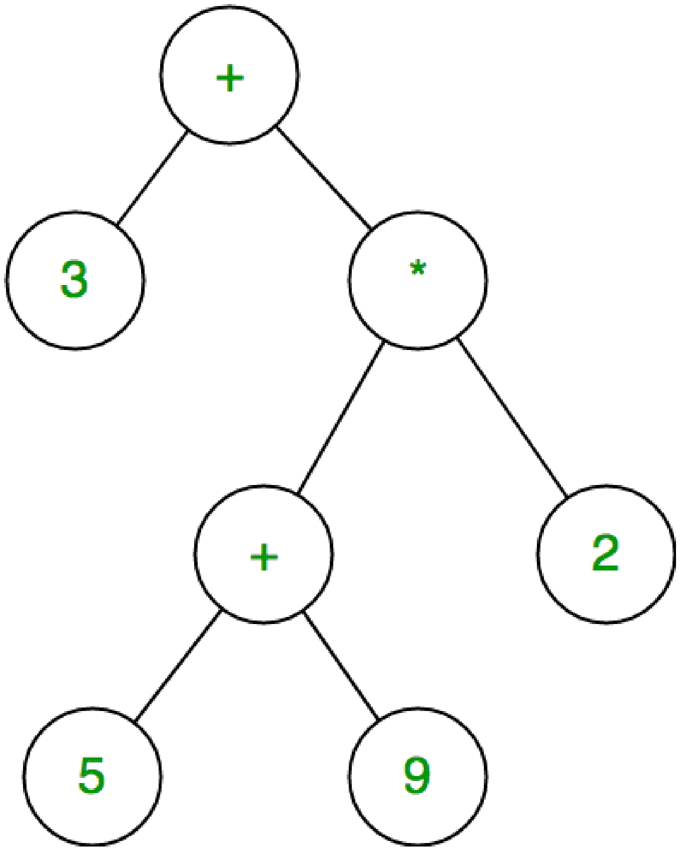


Image courtesy: https://media.geeksforgeeks.org/wp-content/uploads/expression-tree.png

Object rendering in 3-D game

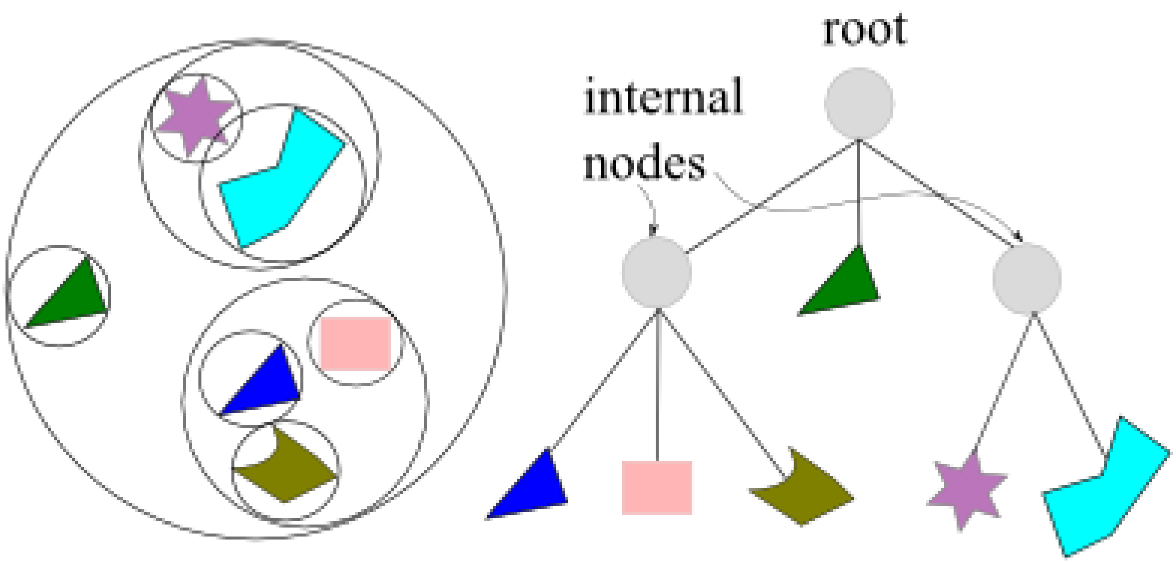


Image Courtesy: https://www.bogotobogo.com/Games/images/BVH.png

DNS Server entries

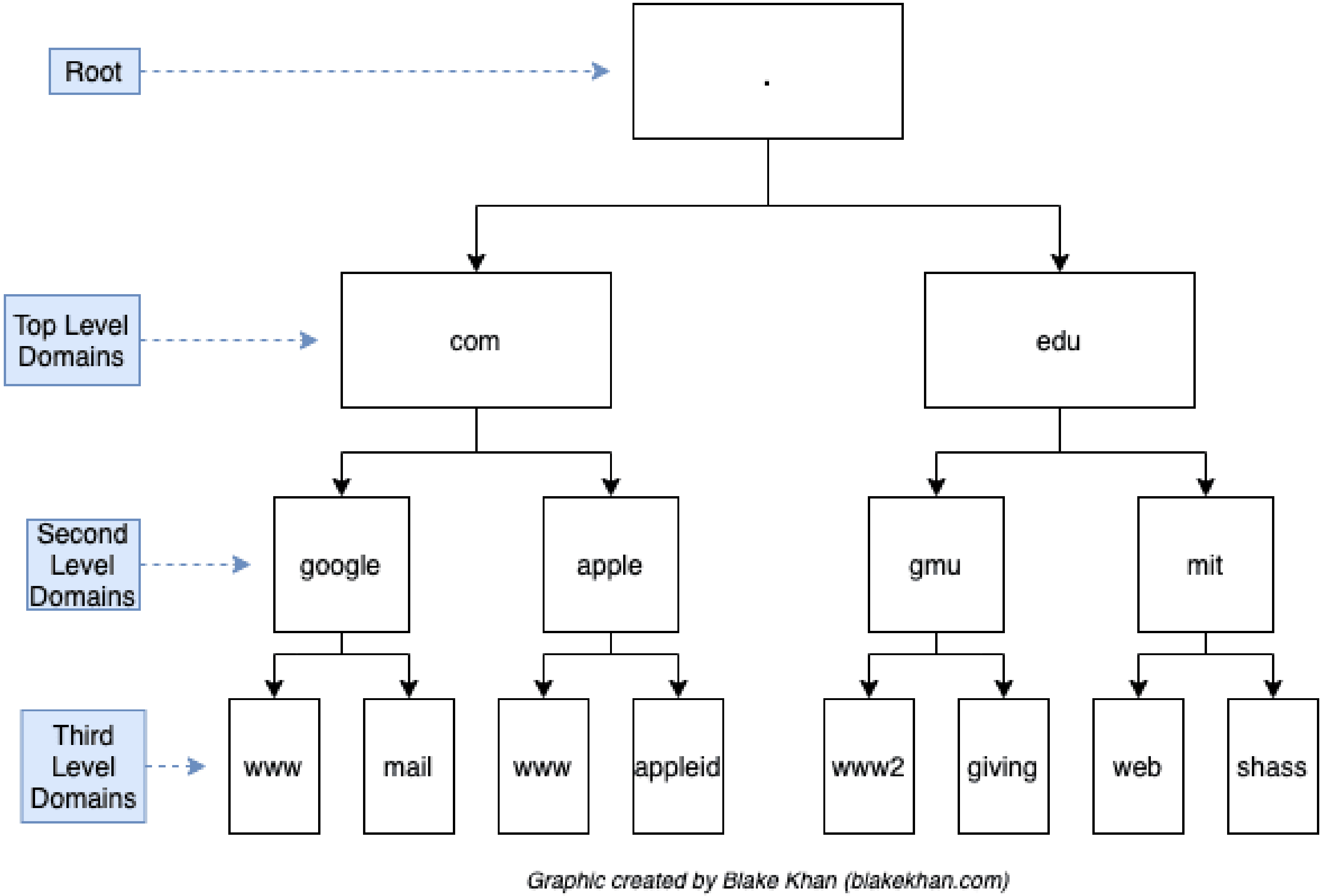
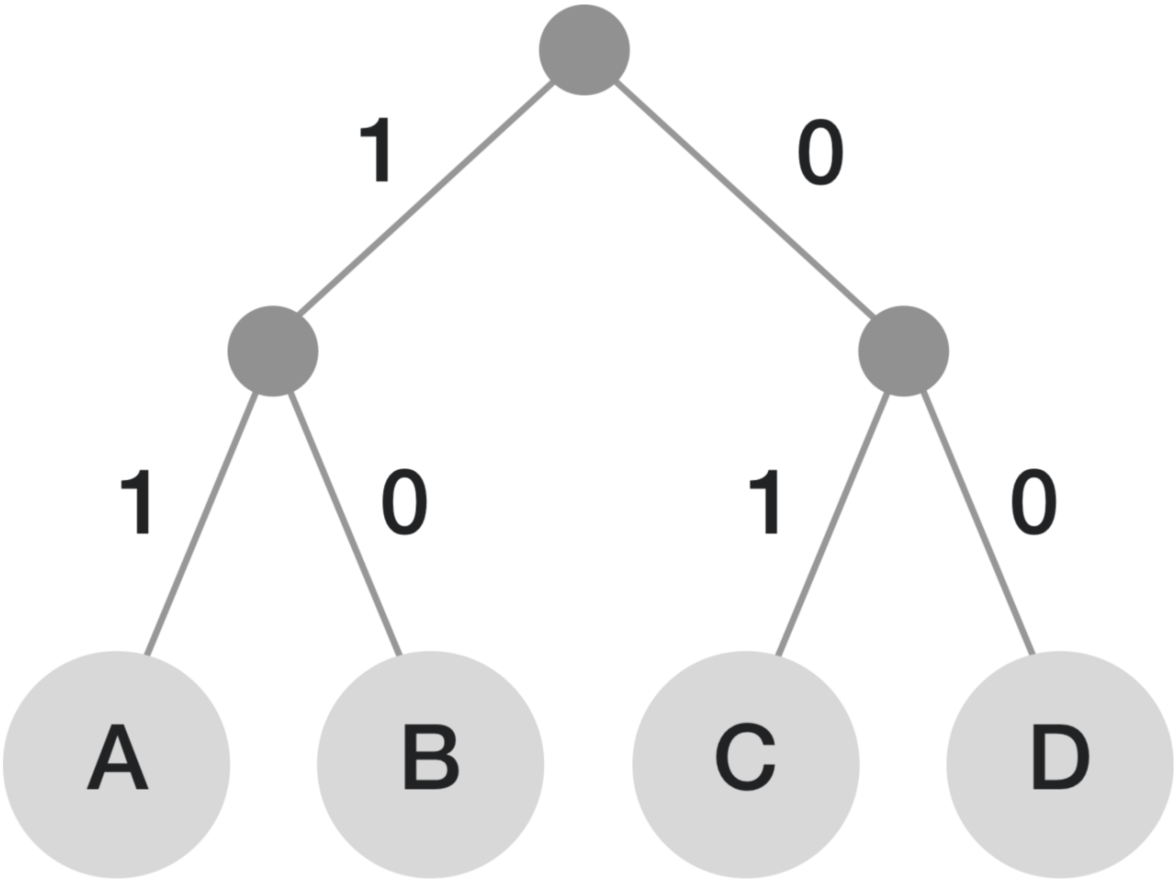


Image Courtesy: https://res.cloudinary.com/practicaldev/image/fetch/s--b9G6DenD--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https://i.imgur.com/xOdVIPZ.png

Compression Algorithm



https://brilliant-staff-media.s3-us-west-2.amazonaws.com/tiffany-wang/VEIWKBhSSc.png

Game Tree

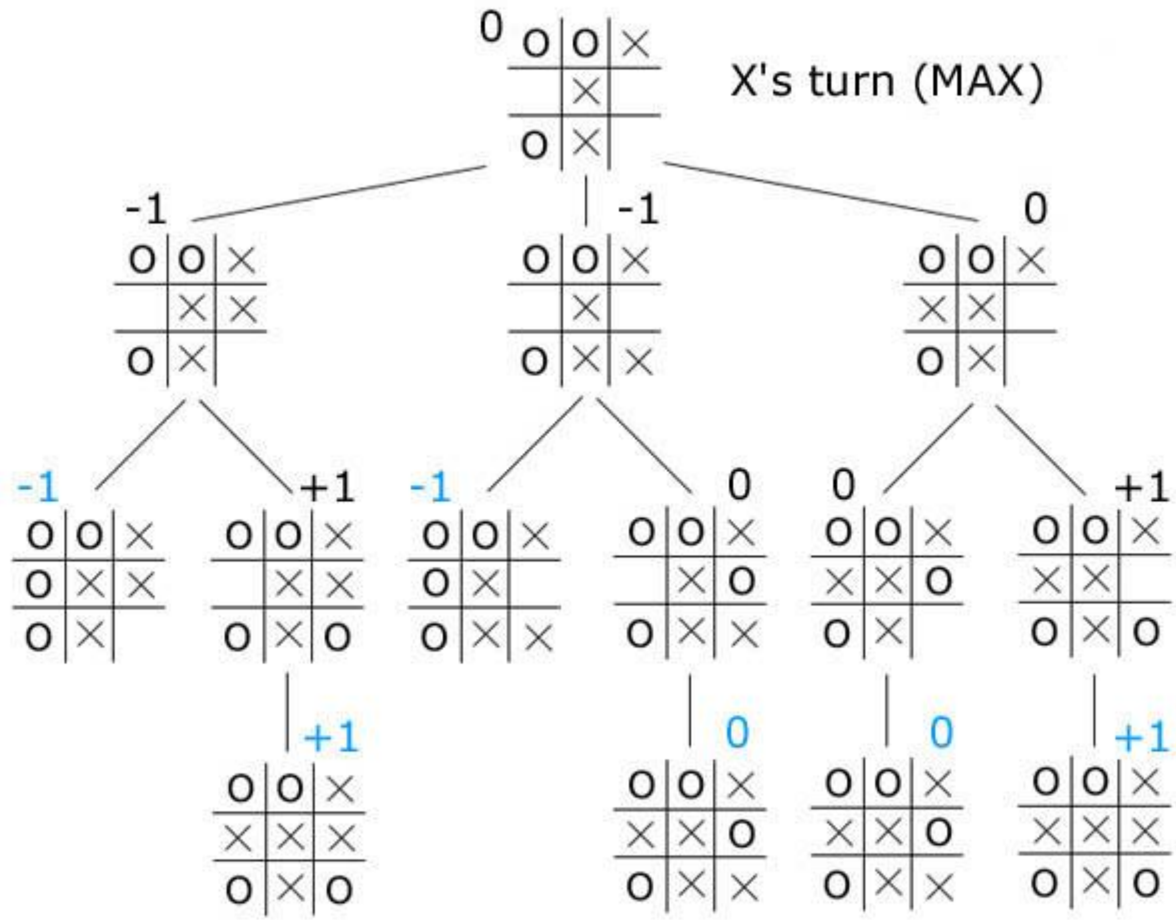
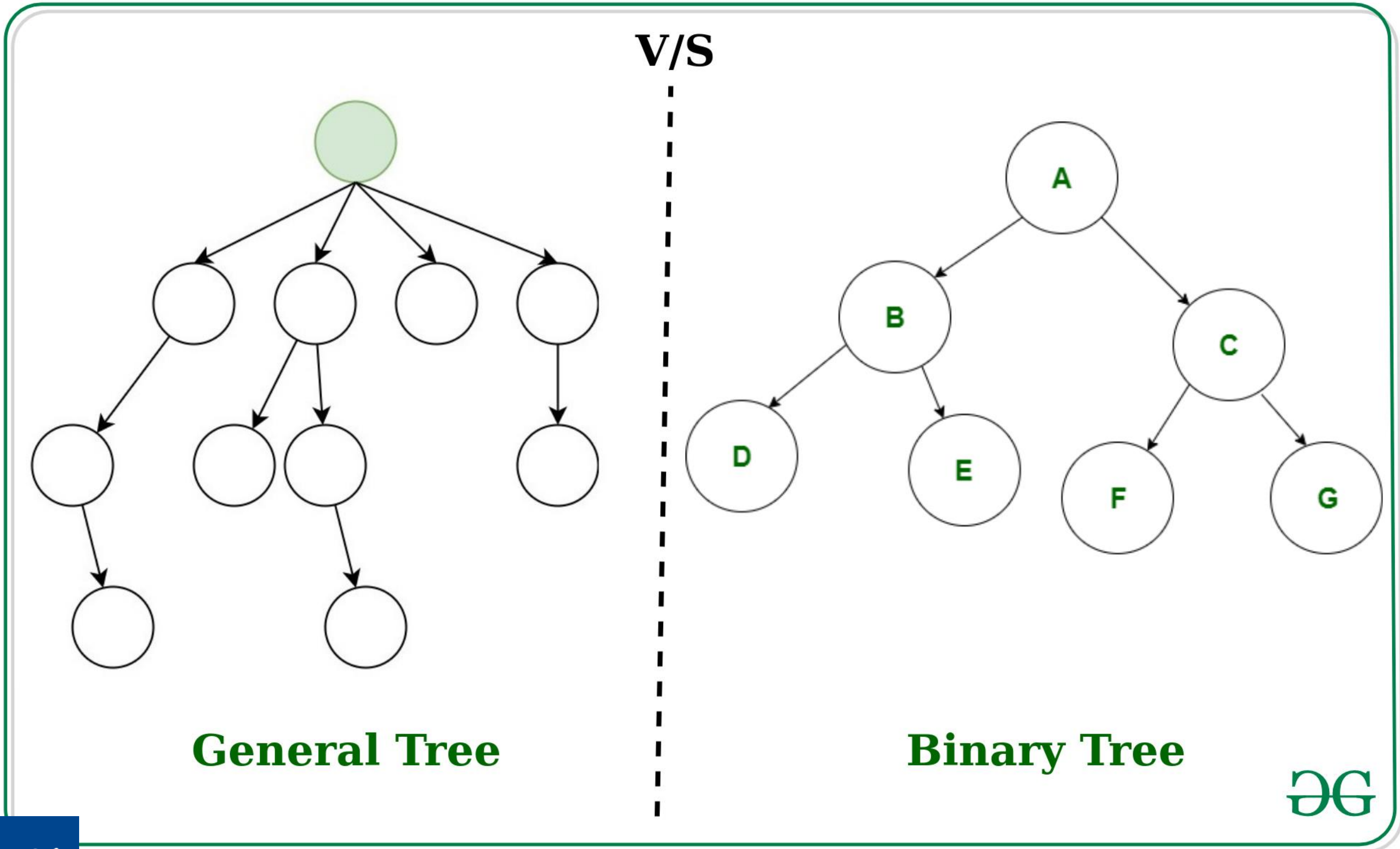


Image Courtesy: https://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.jpg

Binary Trees

A binary tree, T, is either empty or such that

1. T has a special node called the root node
2. T has two sets of nodes L_T and R_T , called the left subtree and right subtree of T, respectively.
3. L_T and R_T are binary trees.


V/S
A
B
C
D
E
F
G
General Tree
Binary Tree

Binary Tree

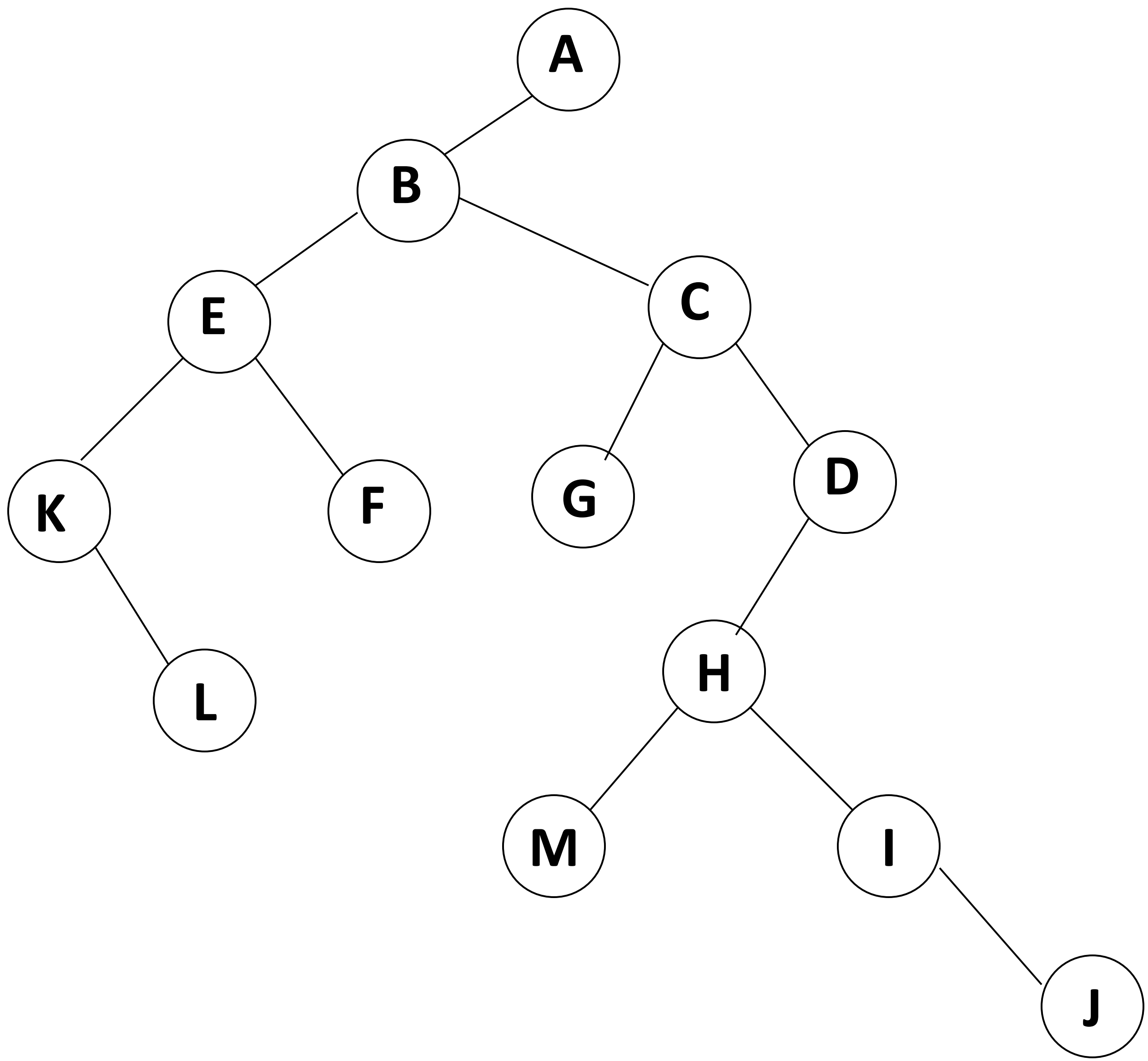
- A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets.
- The first subset contains a single element called the root of the tree.
- The other two subsets are themselves binary trees called the left and right sub-trees of the original tree.
- A left or right sub-tree can be empty.
- Each element of a binary tree is called a node of the tree

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

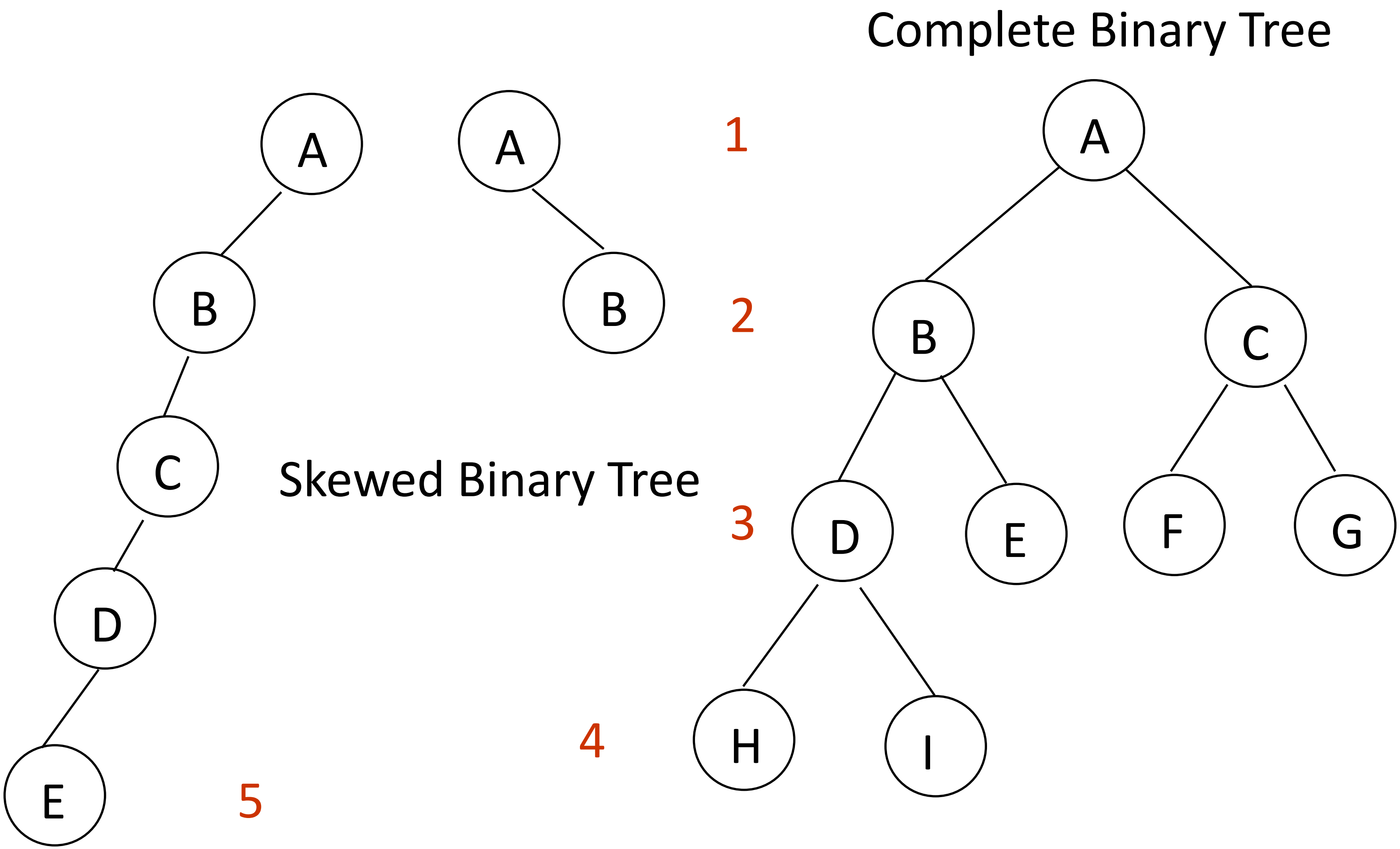
Binary Tree Properties

- If a binary tree contains m nodes at level L, it contains at most 2^m nodes at level L+1
- Since a binary tree can contain at most 1 node at level 0 (the root), it contains at most 2^L nodes at level L.

Somaiya TRUST


A
B
E
C
K
F
G
D
L
H
M
I
J

Samples of Trees

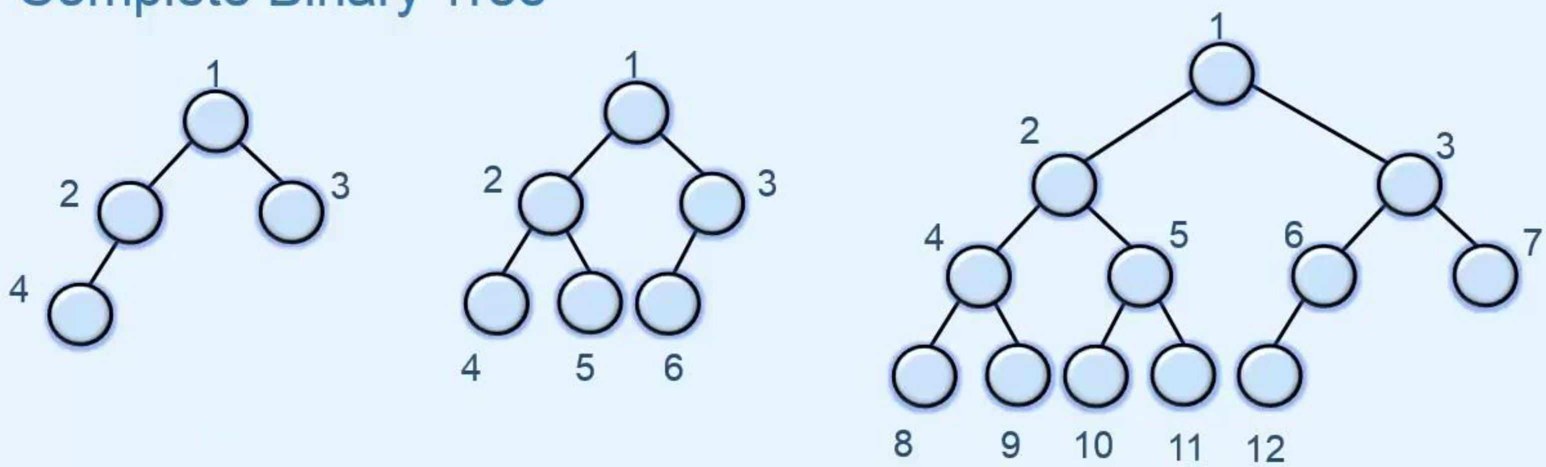


Types of Binary Tree

- Complete binary tree
- Strictly binary tree
- Almost complete binary tree

A complete binary tree

Complete Binary Tree



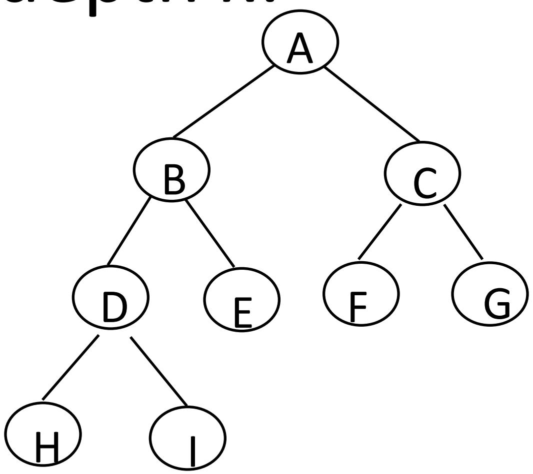
If a node in a complete binary tree is assigned a number k, where 1$\leq$ k $\leq$ n, then

Binary tree types

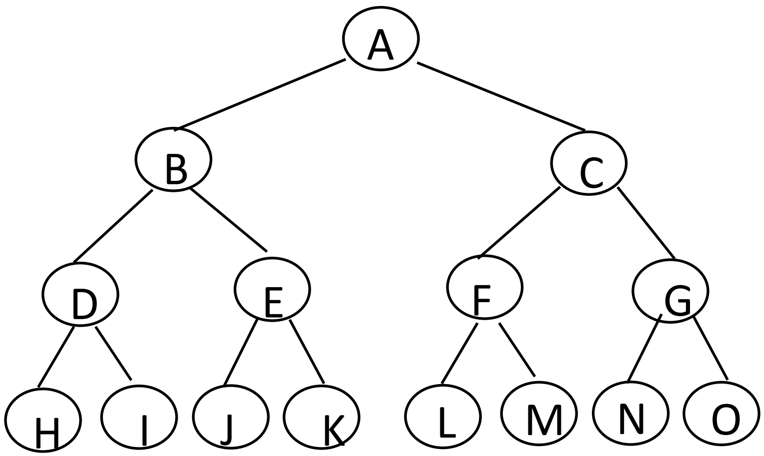
- **Strictly binary** trees are binary trees where every node either has two children or is a leaf (has no children).
- **Complete** binary trees are strictly binary trees where every leaf is on the same "maximum" level.
- **Almost complete** binary trees are not necessarily strictly binary (although they can be), and are *not* complete.

Full BT VS Complete BT

- A full binary tree of depth *k* is a binary tree of depth *k* having 2^k-1 nodes, *k*>=0.
- A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.

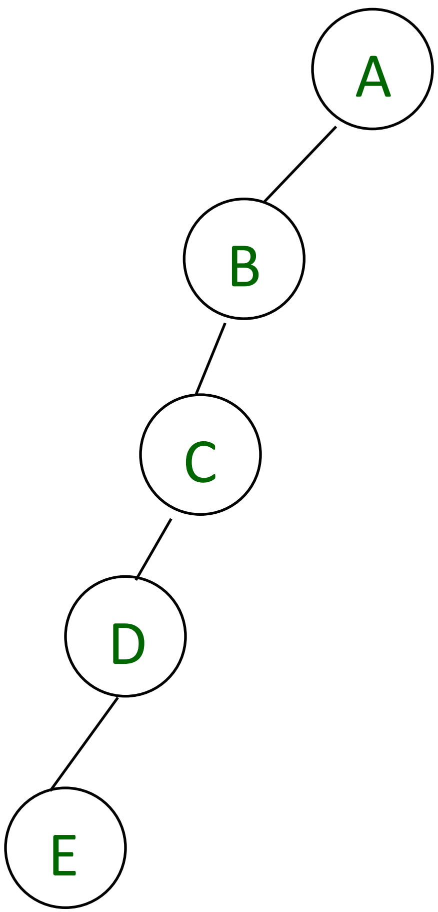


Complete binary tree



Full binary tree of depth 4

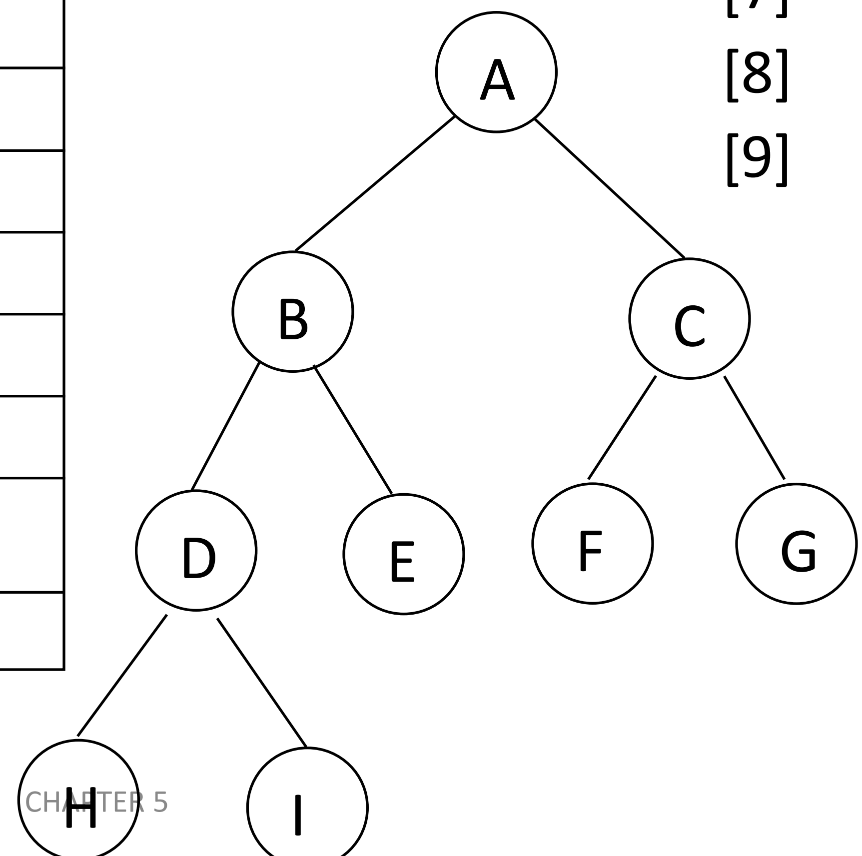
Sequential Representation



[1]	A
[2]	B
[3]	--
[4]	C
[5]	--
[6]	--
[7]	--
[8]	D
[9]	--
.	.
[16]	E

(1) waste space
(2) insertion/deletion problem

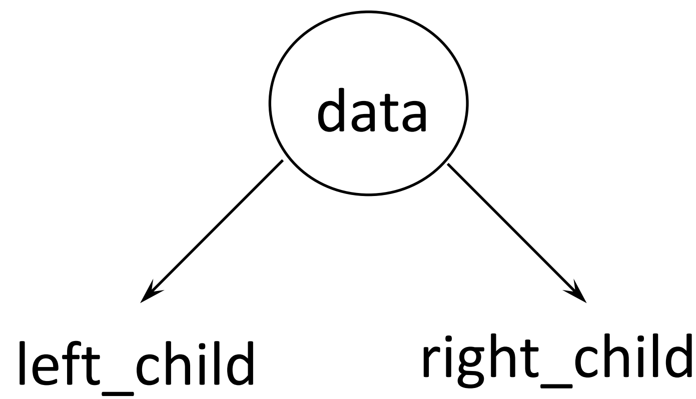
[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I



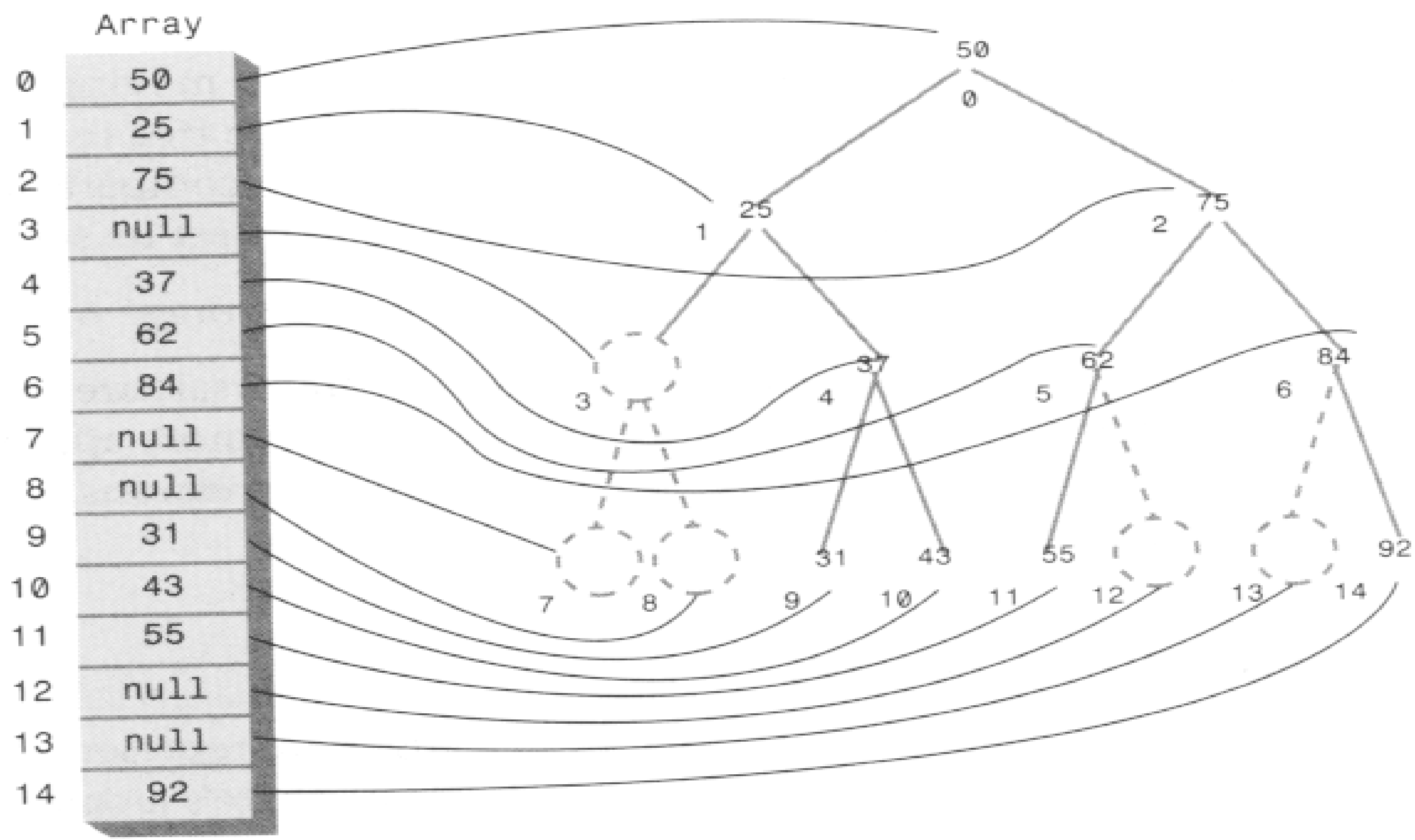
Linked Representation

```c
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```

left_child	data	right_child



Array representaion of tree



Somaiya
TRUST

Binary tree traversal

Traversal: visiting each node only once

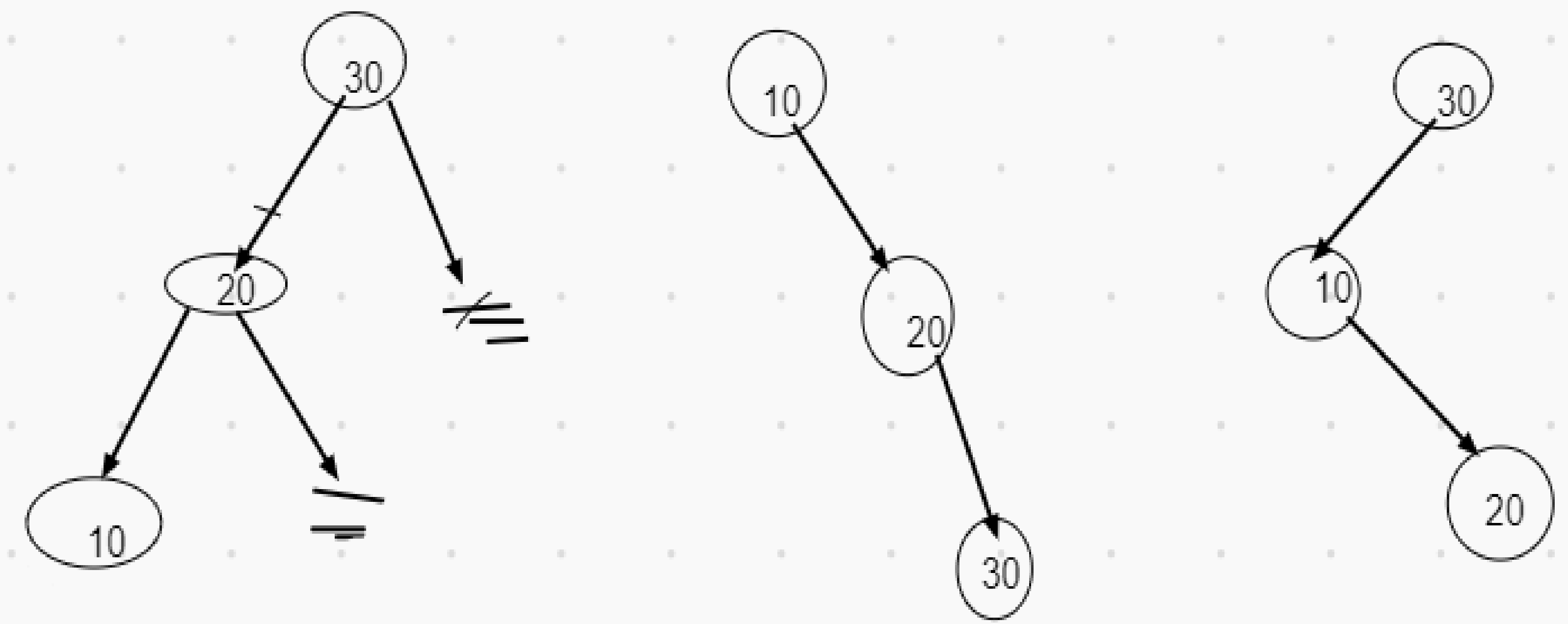
Traversal methods:

- Inorder : Left-Root-Right
- Preorder : Root-Left-Right
- Postorder : Left-Right-Root

Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of travers
  - LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain
  - LVR, LRV, VLR
  - inorder, postorder, preorder

Binary Tree Traversals

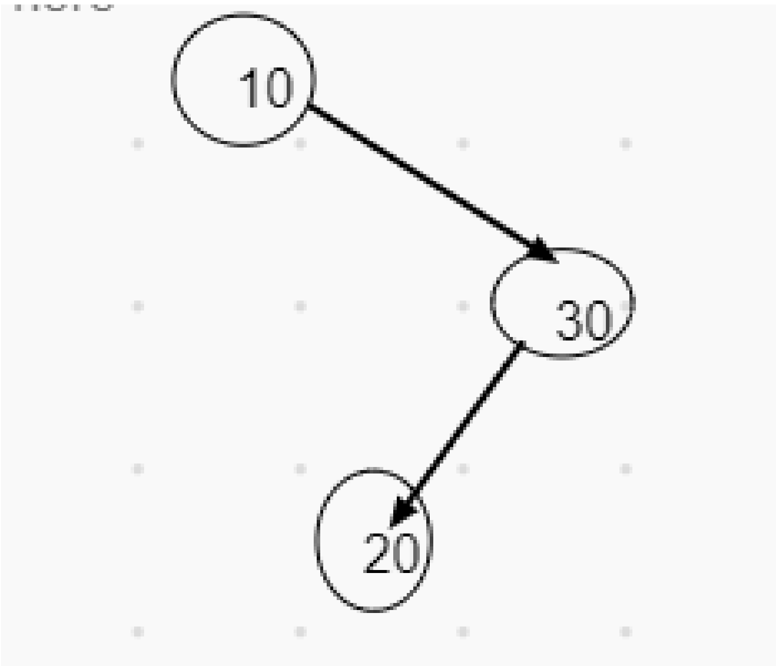


inorder -10, 20,30,
Preorder - 30, 20, 10
Postorder - 10, 20,30

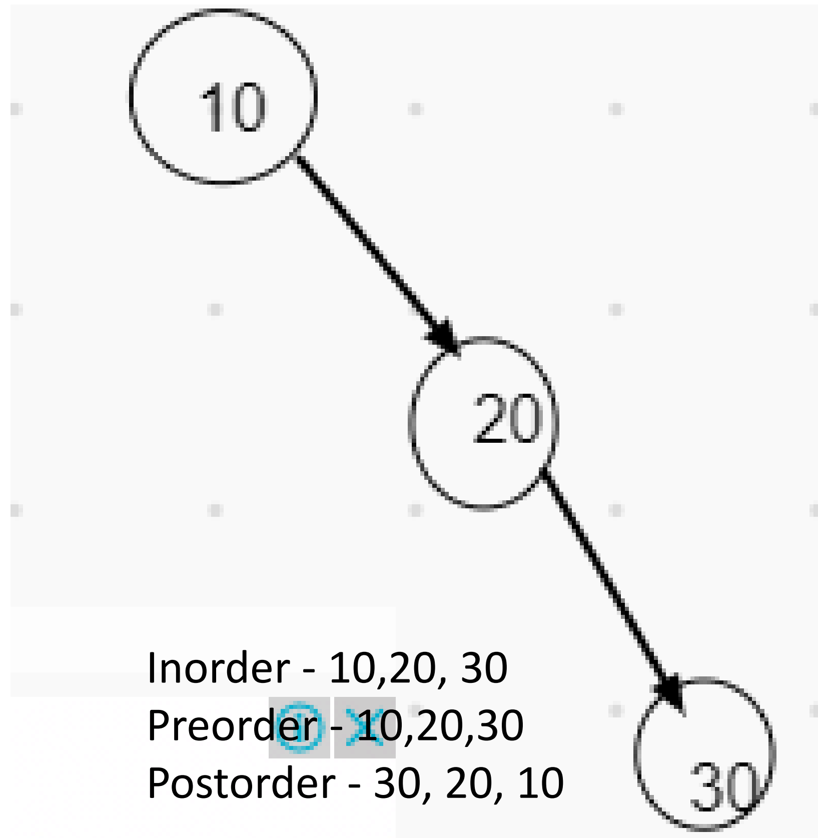
inorder - 10, 20,30
Preorder - 10, 20,30
Postorder - 30, 20,10

inorder - 10, 20,30
Preorder - 30, 10,20,
postorder - 20, 10,30

Binary Tree Traversals



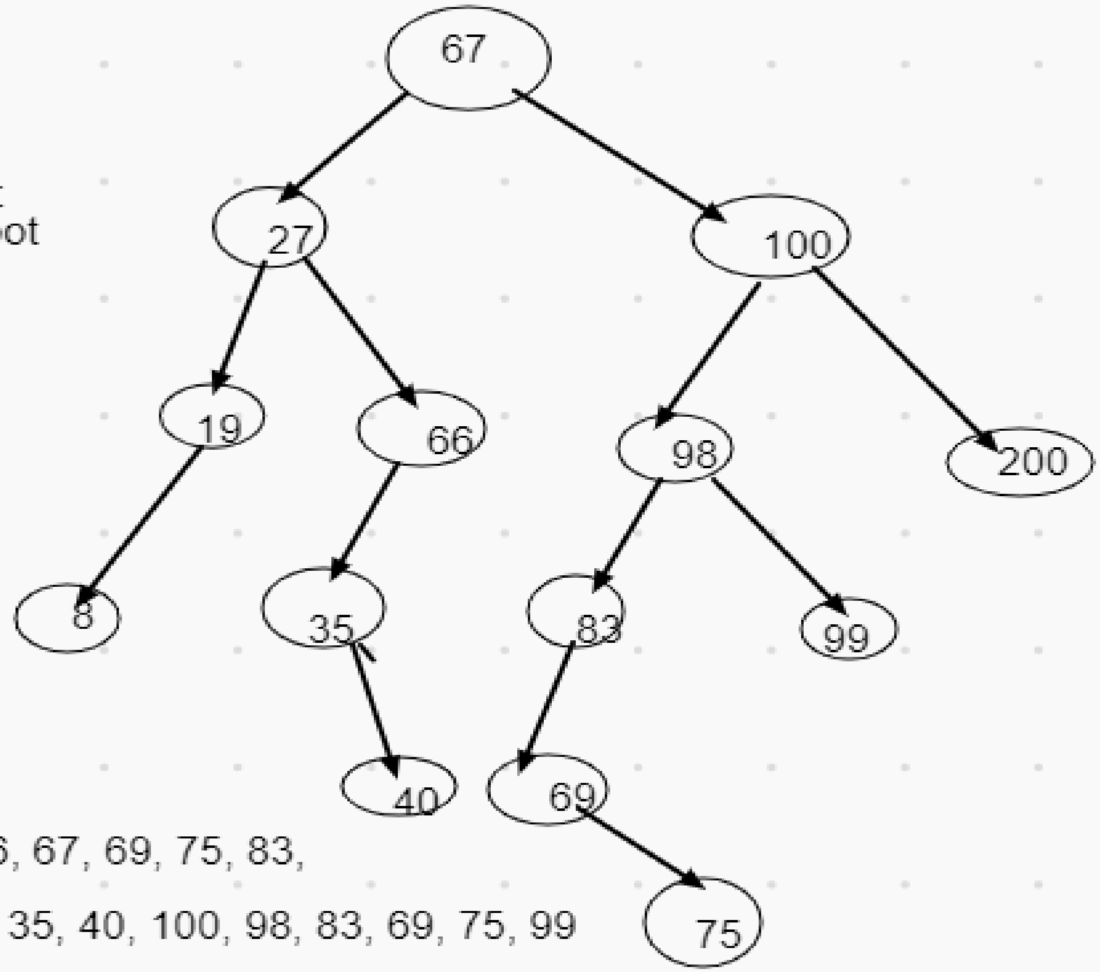
inorder - 10, 20,30
Preorder - 10,30, 20
Postorder - 20, 30,10



Inorder - 10,20, 30
Preorder - 10,20,30
Postorder - 30, 20, 10

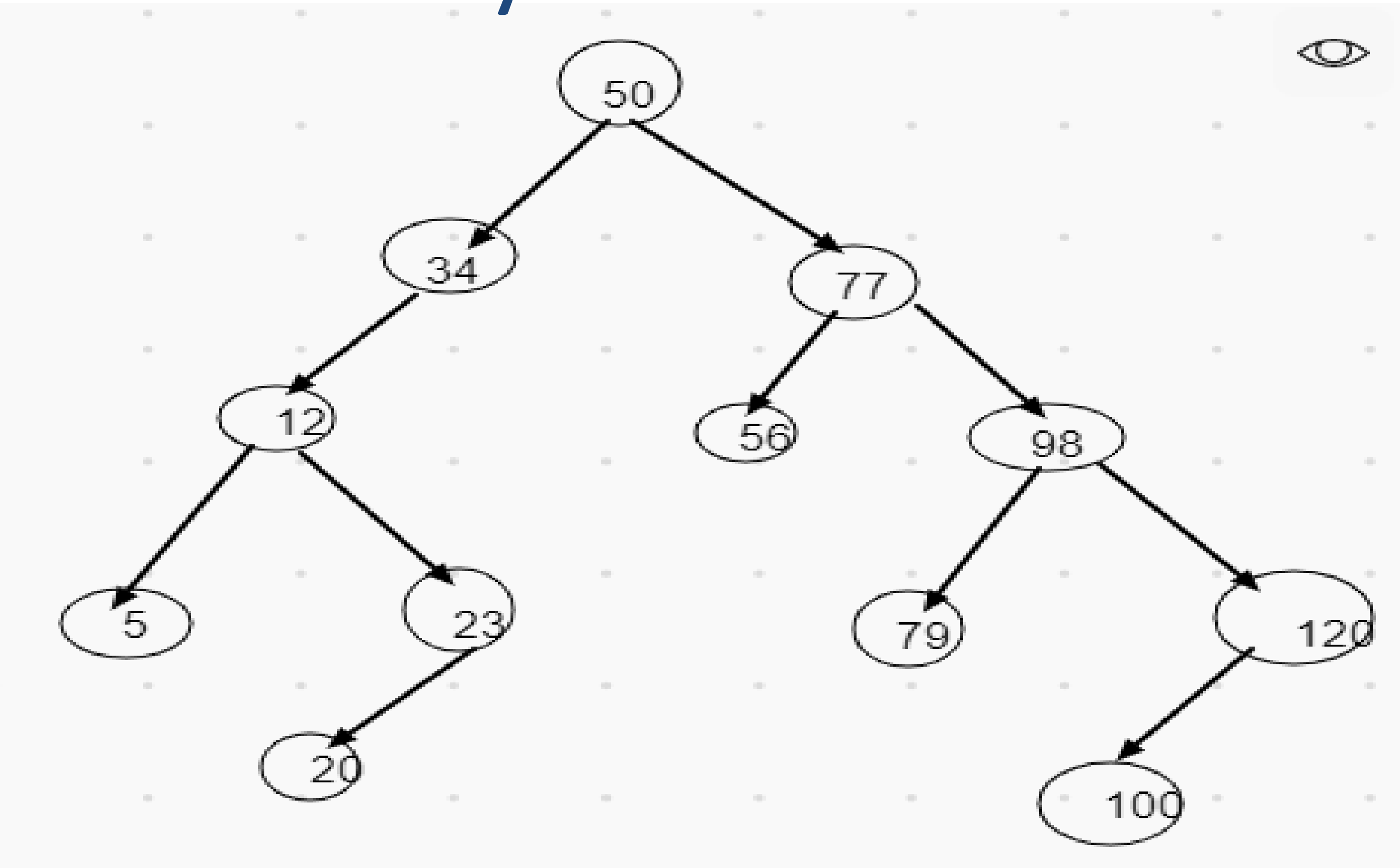
Binary Tree Traversals

Traversal techs
1. inorder - Left-Root-Right
2. Preorder- Root-Left-right
3. Postorder- Left-Right-Root



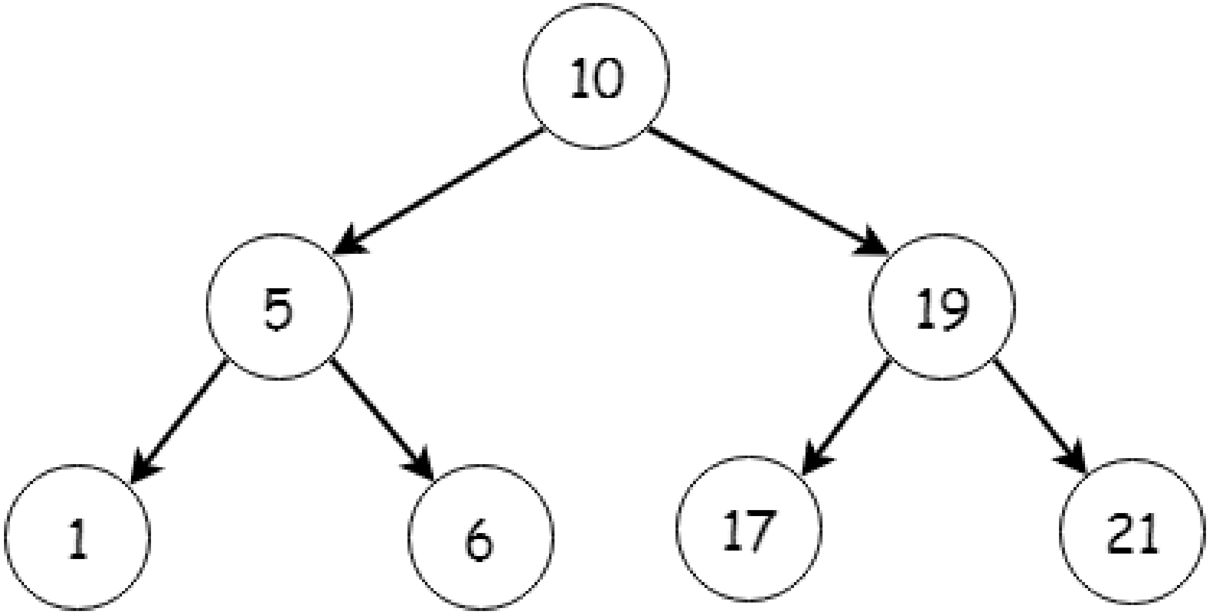
inorder-8, 19, 27, 35, 40, 66, 67, 69, 75, 83,
98, 99, 100, 200
Preorder- 67, 27, 19, 8, 66, 35, 40, 100, 98, 83, 69, 75, 99
200
Postorder- 8, 19, 40, 35, 66, 27, 75, 69,83, 99,98, 200, 100, 67

Binary Tree Traversals



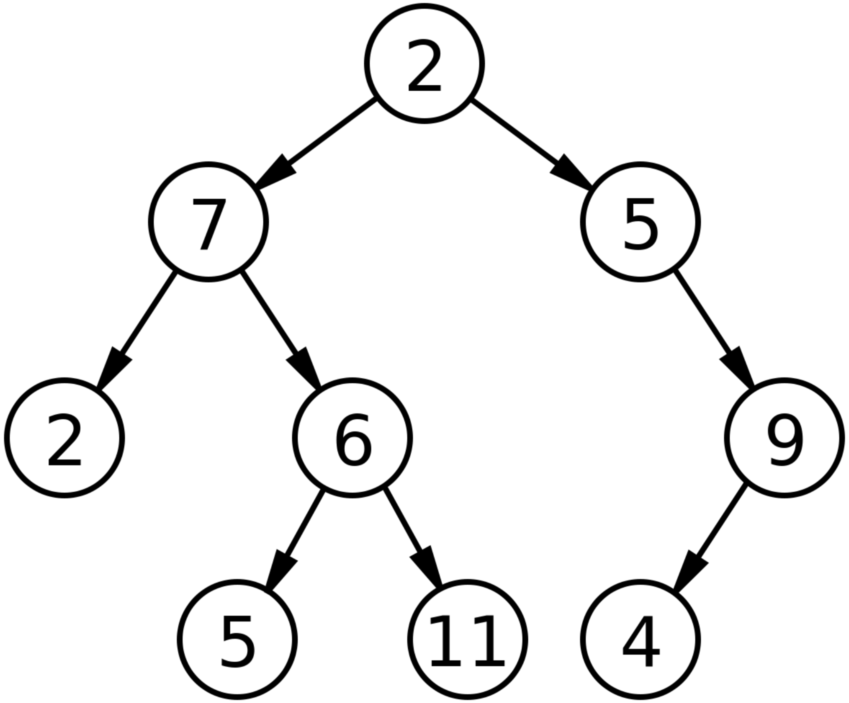
Preorder- 50, 34, 12,5, 23, 20,77, 56,98 79,120, 100,
Postorder- 5, 20, 23, 12, 34, 56, 79, 100, 120, 98, 77, 50
Inorder- 5, 12, 20, 23, 34, 50, 56, 77,79, 98, 100, 120

Binary tree Traversal



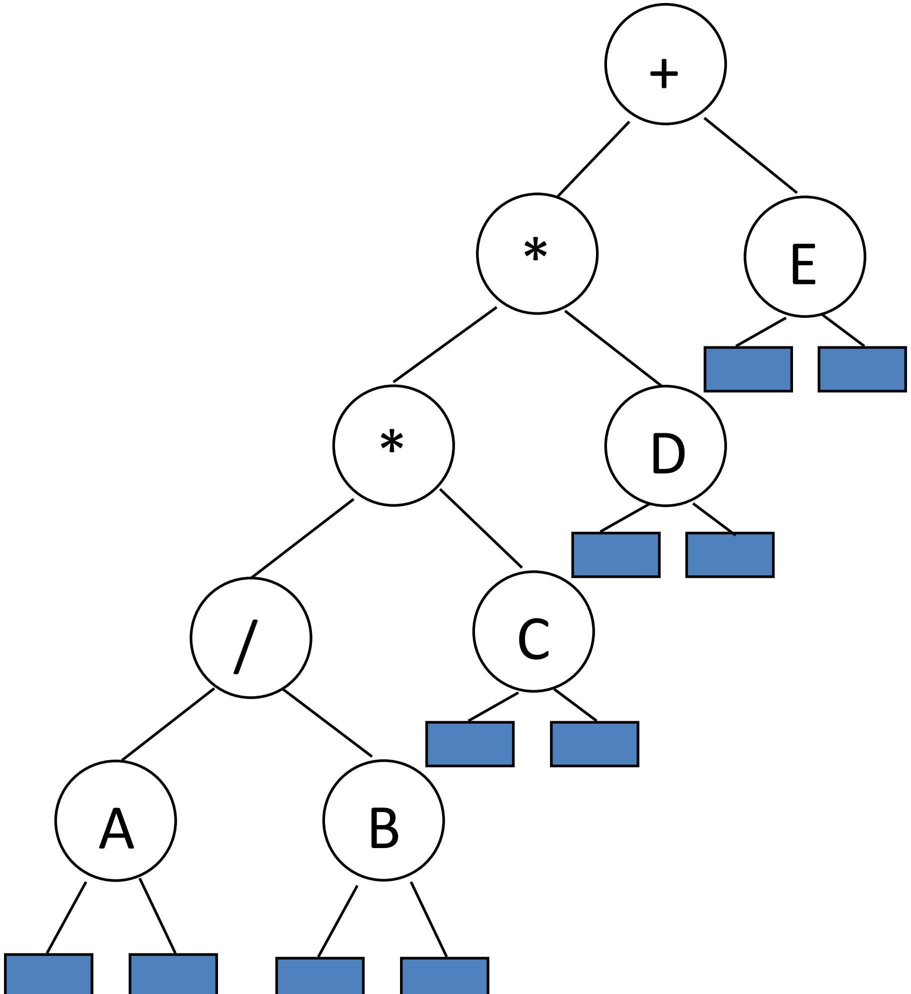
- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21
- Postorder: 1-6-5-17-21-19-10

Binary tree Traversal



- Inorder: ?
- Preorder: ?
- Postorder: ?

Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

level order traversal

+ * E * D / C A B

Inorder Traversal (recursive version)

```
void inorder(tree_pointer ptr)
/* inorder tree traversal */
{
    if (ptr) {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        indorder(ptr->right_child);
    }
}
```

A / B * C * D + E

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Preorder Traversal (recursive version)

```
void preorder(tree_pointer ptr)
/* preorder tree traversal */
{
    if (ptr) {
        printf("%d", ptr->data);
        preorder(ptr->left_child);
        predorder(ptr->right_child);
    }
}
```

+ * * / A B C D E

Postorder Traversal (recursive version)

```
void postorder(tree_pointer ptr)
/* postorder tree traversal */
{
    if (ptr) {
        postorder(ptr->left_child);
        postdorder(ptr->right_child)
        printf("%d", ptr->data);
    }
}
```

A B / C * D * E +

Construction of binary tree from traversals

- Can be done with two pairs of information:
  - Inorder & Preorder
  - Inorder & Postorder

- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21

Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

Creation of Binary Search Tree from traversals

- For inorder & Preorder traversal pairs:

Creation of Binary Search Tree from traversals

- For inorder & Preorder traversal pairs:
  - Read the preorder input from left to right
  - Mark the corresponding nodes in inorder traversal to mark and left & right subtrees recursively
- For inorder & Postorder traversal pairs:
  - Read the postorder input from right to left
  - Mark the corresponding nodes in inorder traversal to mark and right & left subtrees recursively

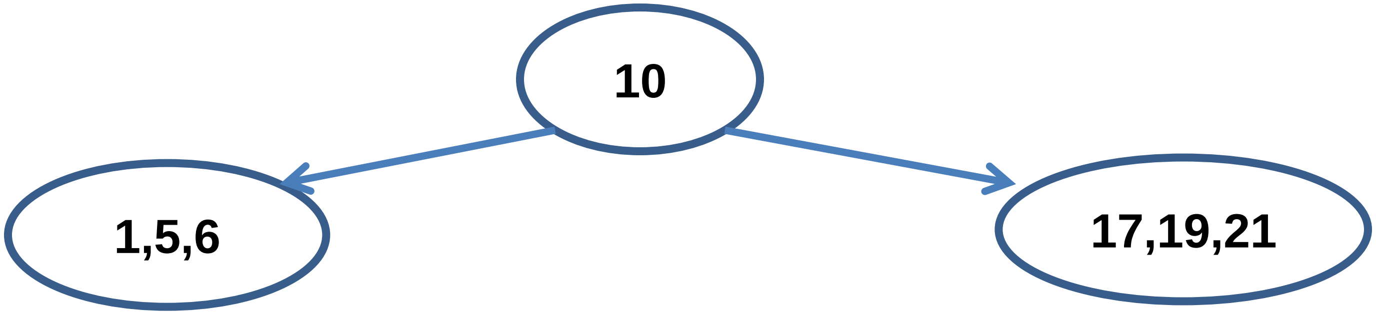
Creation of Binary Search Tree from traversals

- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21

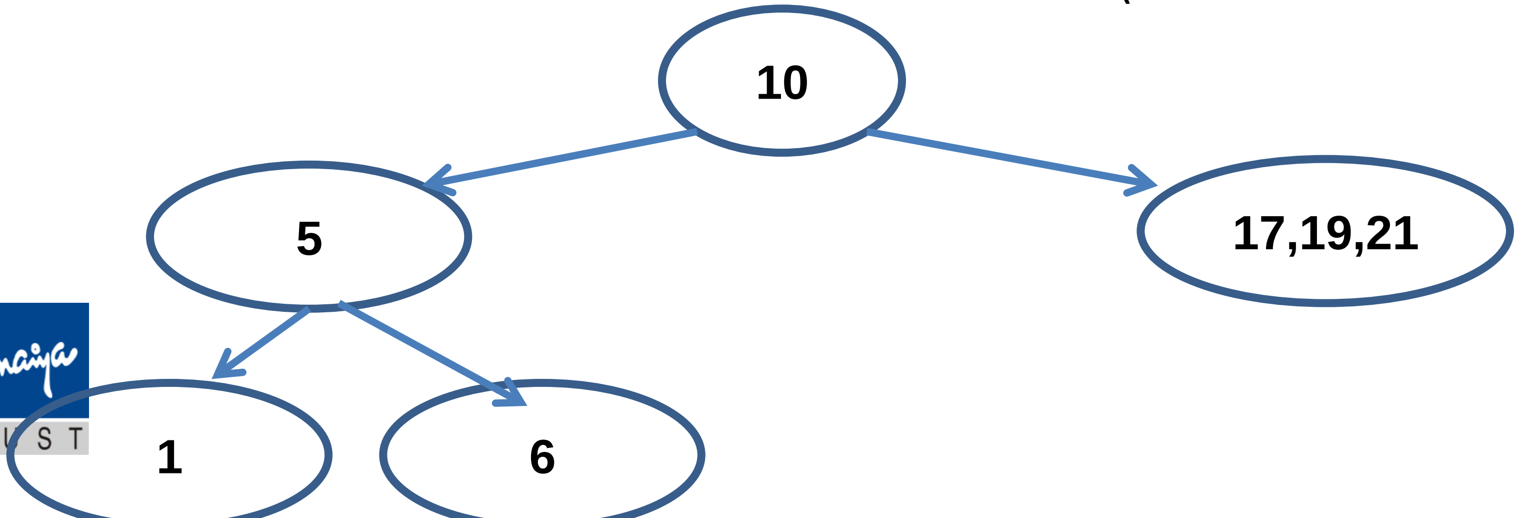
Step 1: read preorder sequence from left to right, mark the corresponding node in inorder.

The preorder sequence gives roots and subroots while inorder sequence divides them in left and right part.

Creation of Binary Search Tree from traversals

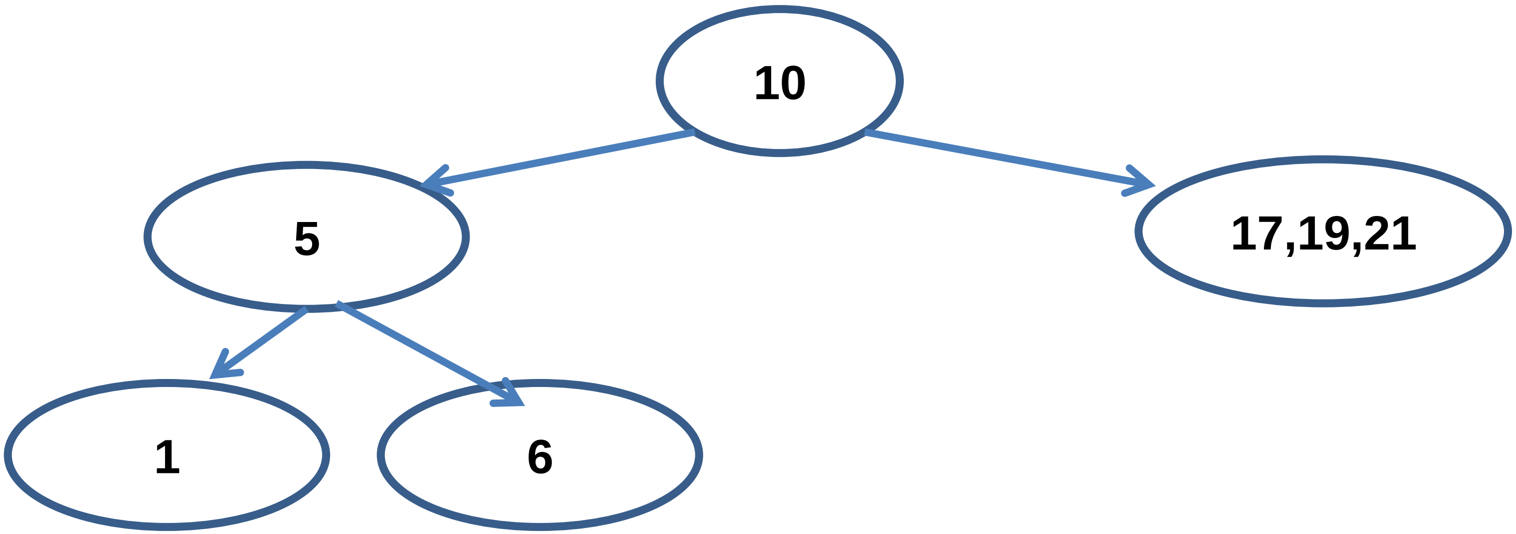


- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21 (mark 5 as next root)



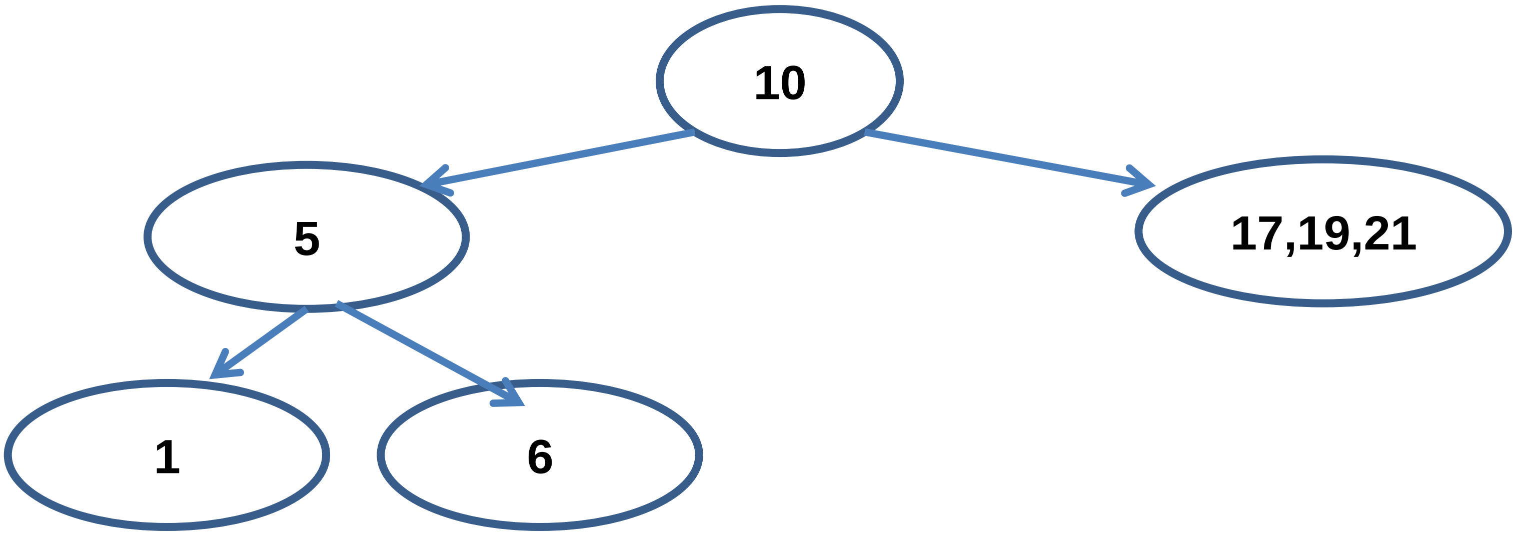
Creation of Binary Search Tree from traversals

- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21 (Mark 1 as next root, but it has no unmarked elements in inorder sequence i.e. it's a leaf node)



Creation of Binary Search Tree from traversals

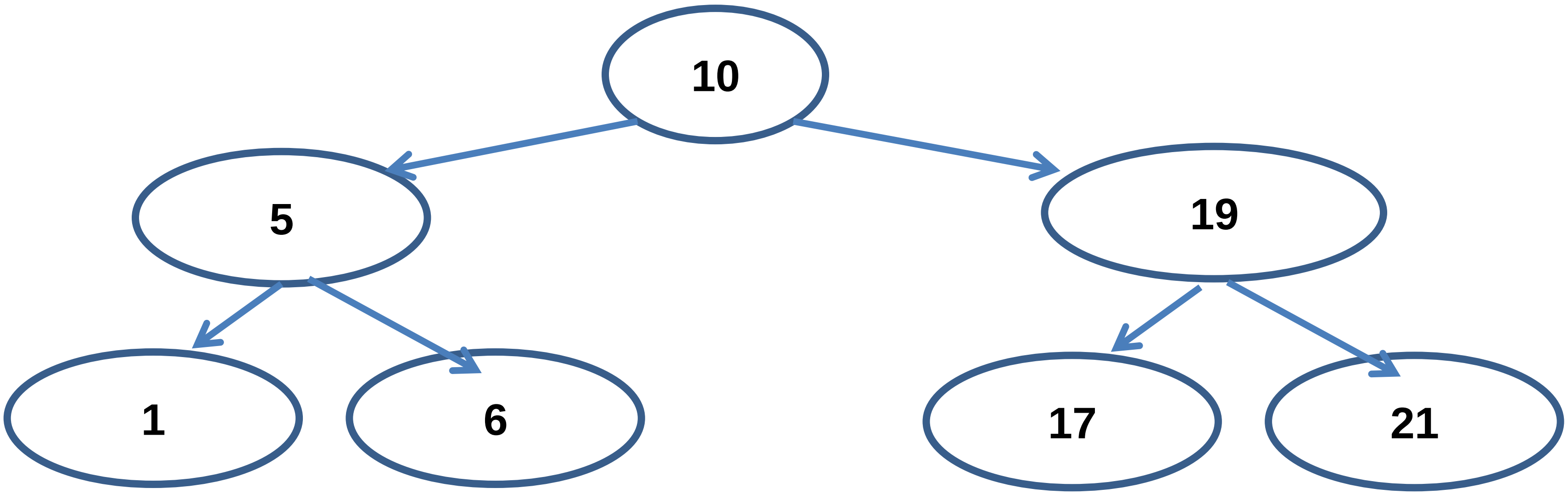
- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21 (Mark 6 as next root, but it has no unmarked elements in inorder sequence i.e. it's a leaf node,too)



SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Creation of Binary Search Tree from traversals

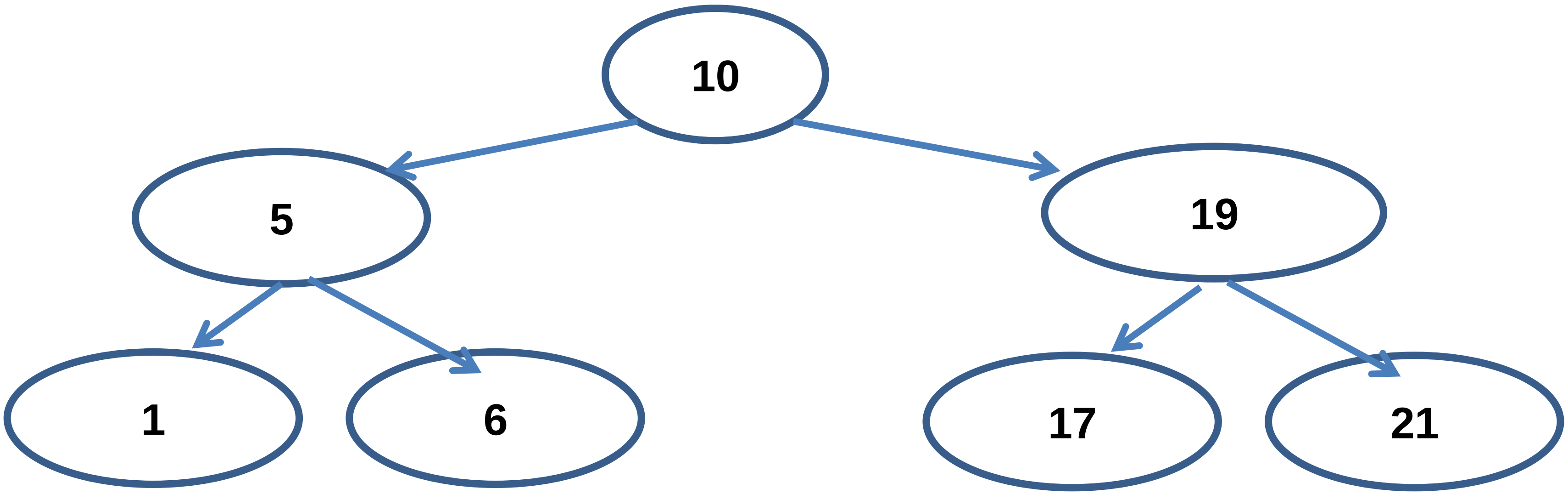
- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21 (Mark 19 as next root)



Somaiya TRUST

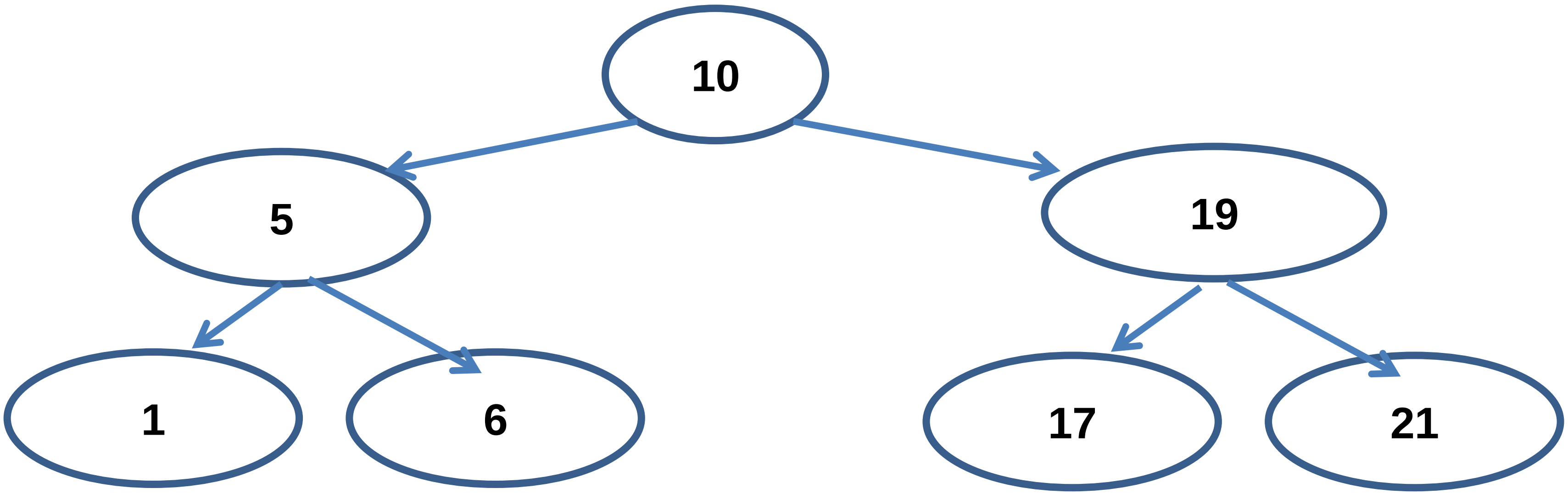
Creation of Binary Search Tree from traversals

- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21 (Mark 17 as next root, it turns out to be a leaf node)



Creation of Binary Search Tree from traversals

- Inorder: 1-5-6-10-17-19-21
- Preorder: 10-5-1-6-19-17-21 (Mark 21 as next root, it turns out to be a leaf node,too)



Creation of Binary Search Tree from traversals

- Follow same process for postorder and inorder sequence, but read the postorder from right to left.

Construction of Binary Search Tree

Construct a BST for:

- 10,45,23,90,21,65,100,4,78,50
- 50,78,4,100,65,21,90,23,45,10
- 65,4,50,10,78,45,100,23,90,21

Binary Search implementation

```
Struct tree{
        int data;
        struct tree *left;
        struct tree *right;
        }
Struct tree *t;
```

Insertion in BST

```
Treetype  insert(TreeType *root, int key)
{       CreateNode(NewNode)
    // find the position to insert the new node
   Treetype *temp  = root;
    // Pointer parent maintains the trailing  pointer of temp
    Treetype *parent = NULL;

   while (temp != NULL) {
      parent = temp;
      if (key < temp->data)           temp = temp->left;
      else                temp = temp->right;
   }
    // If the root is NULL i.e the tree is empty  The new node is the root node
   if (parent == NULL)         root = newnode;
    // If the new key is less then the leaf node key Assign the new node to be its left child
   else if (key < parent->data)             parent->left = newnode;
    // else assign the new node its right child
   else                                 parent->right = newnode;
       return root;
}
```

Count nodes

```
int countNodes(TreeType t)
{
If (t==Null)
Print "tree is empty"
Else if (Left(t)==Null AND Right(t)==Null)
        return 1
Else
return(CountNodes(Left(t)+CountNodes(Right(t))+1
}
```

Binary Search tree deletion

Cases:

1. Deletion from empty tree
2. The key to be deleted doesn’t exist in tree
3. The node to be deleted is the only node in tree
4. The node to be deleted is root
5. The node to be deleted has
   1. No child
   2. Exactly one child
   3. Two children

Deletion of a node in BST

```
//Deletion from empty tree
If (root==null)
{ print "Error"
  exit
}

//Deletion of only node
If(root->data ==key && root->left==Null && root->right==Null)
{
    temp=root
    Root=null
    Return(temp)
}
```

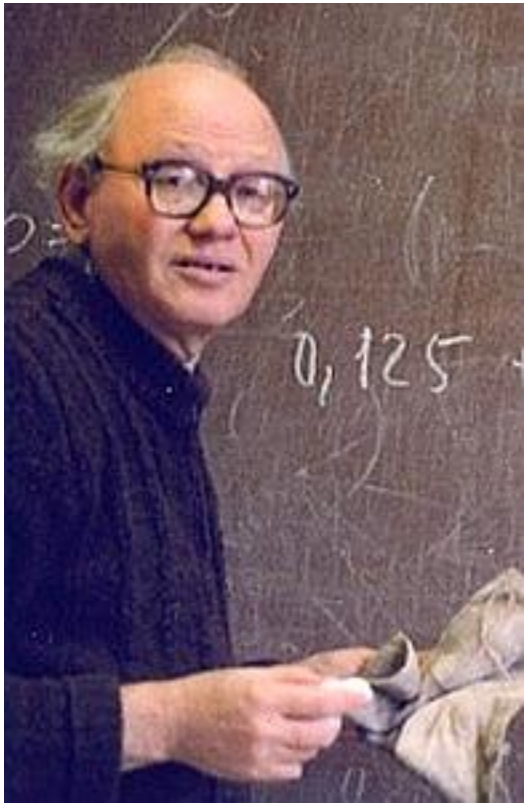
```
Parent = null
Temp =root
While(temp!=null && temp->data !=key)
{ if (key< temp->data)
      parent = temp; temp=temp->left
  else
      parent = temp; temp=temp->right
}
If (temp==null)
Print "Error, element not found" Exit
Elseif (temp->left== null && temp->right==Null)
//node with no children
    { if (temp==parent->left)
          parent->left= Null
        Else
          parent->right=Null
     }
```

```
Elseif(temp->left!= null && temp->right==Null)
//node with only left child
{ if (temp==parent->left)
            parent->left= temp->left
        Else
          parent->right=temp->left
      }
Elseif(temp->left== null && temp->right!=Null)
//node with only right child
{ if (temp==parent->left)
            parent->left= temp->right
        Else
          parent->right=temp->right
      }
```

//Deletion of node with two children

AVL tree

Named after their inventors **Adelson, Velski & Landis**, AVL trees are height balancing binary search tree.



AVL tree

Let's consider creation of a BST i.e. insert values starting from an empty tree

Insert values 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

- If inserted in given order, what is the tree?
- Is inserting in the reverse order any better?

BST: Efficiency of Operations?

Problem:

Worst-case running time:

- `find`, `insert`, `delete`

- `buildTree`

How can we make a BST efficient?

Observation

Solution: Require a **Balance Condition** that

- When we build the tree, make sure it's balanced.
- BUT…Balancing a tree only at build time is insufficient.
- We also need to also keep the tree balanced as we perform operations.

Potential Balance Conditions

- Left and right subtrees

- Left and right subtrees

The AVL Tree Data Structure

An AVL tree is a self-balancing binary search tree.

Structural properties

Binary tree property (same as BST)

Order property (same as for BST)

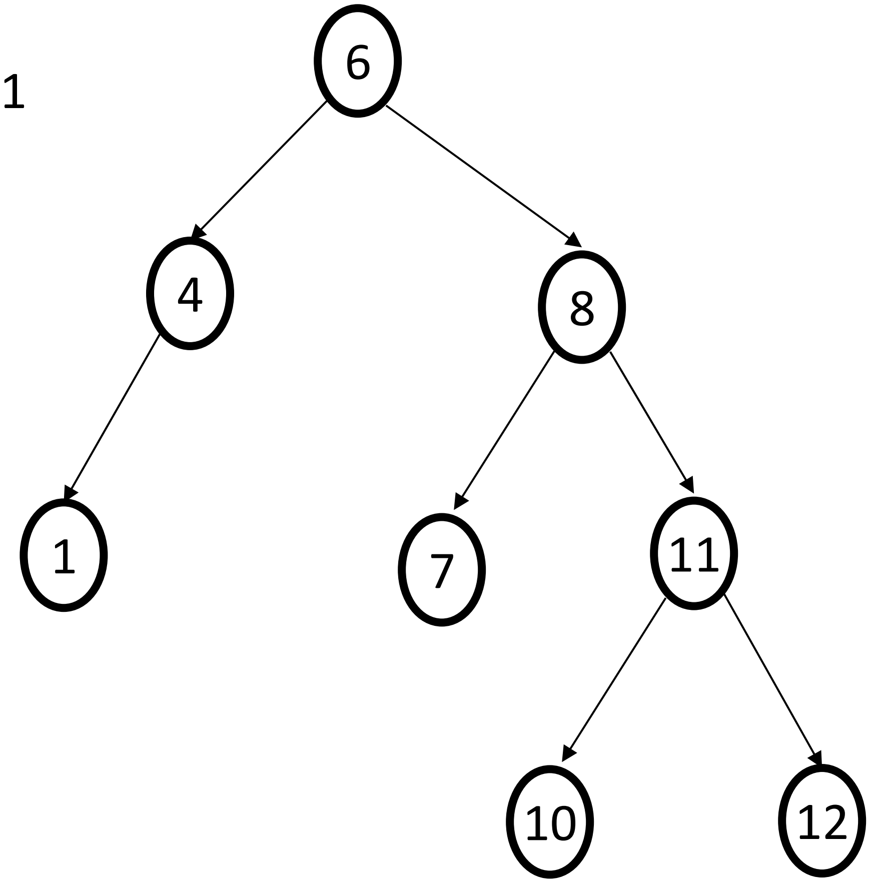
Balance condition:
balance of every node is between -1 and 1

where balance(node) = height(node.left) – height(node.right)

Example #1: Is this an AVL Tree?

Balance Condition:
balance of every node is between -1 and 1

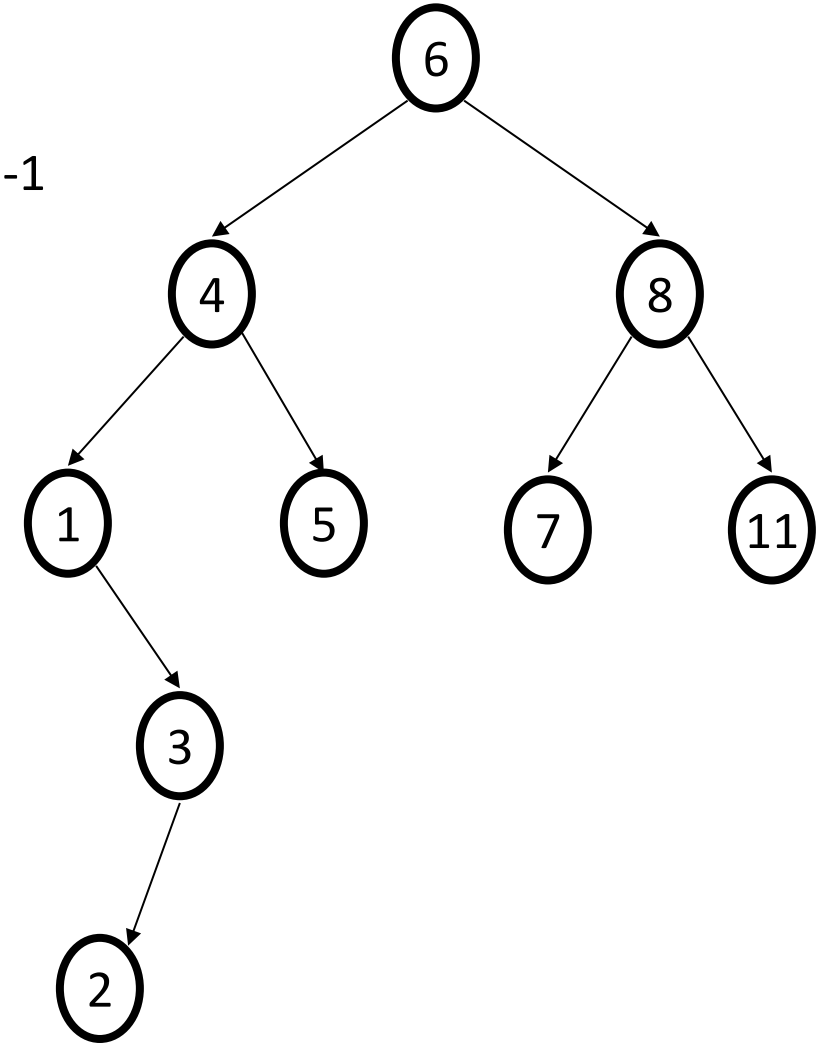
where balance(*node*) =
height(*node*.left) – height(*node*.right)



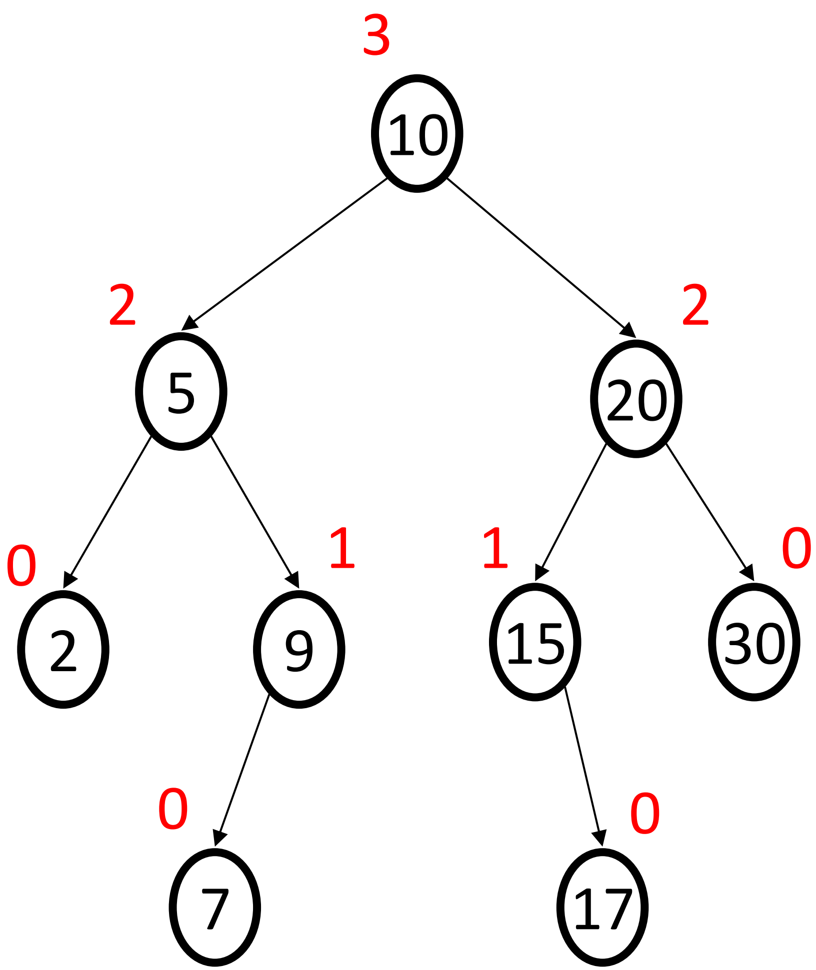
Example #2: Is this an AVL Tree?

Balance Condition:
balance of every node is between -1 and 1

where balance(*node*) =
height(*node*.left) – height(*node*.right)



AVL Trees



First `insert` example

Insert(6)

Insert(3)

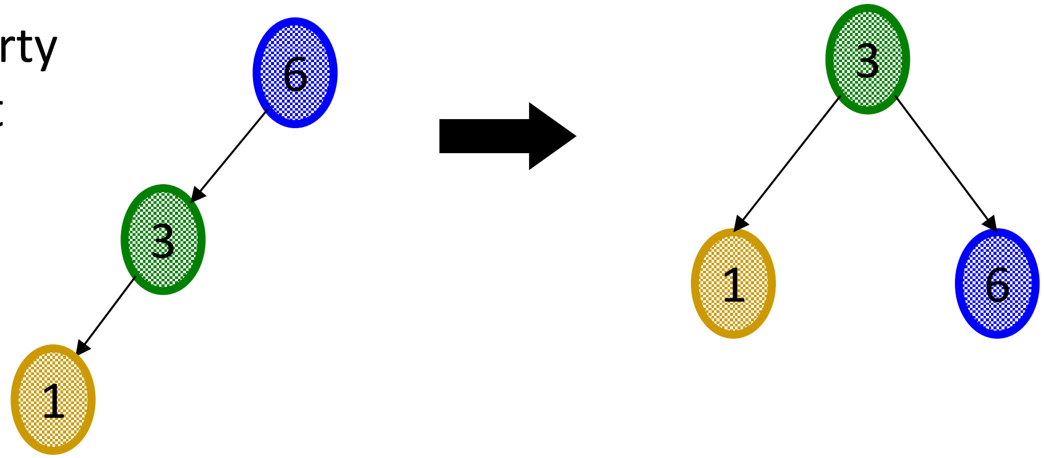
Insert(1)

Third insertion

What's the only way to fix it?

Fix: Apply “Single Rotation”

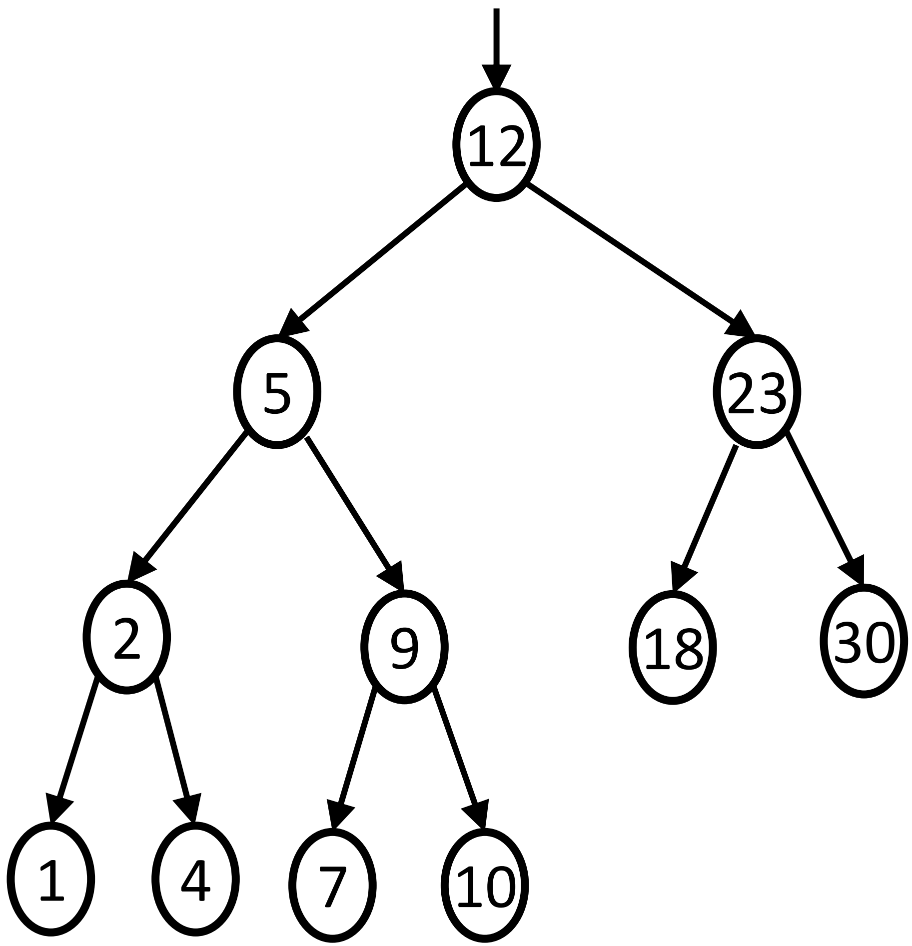
AVL Property violated at node 6



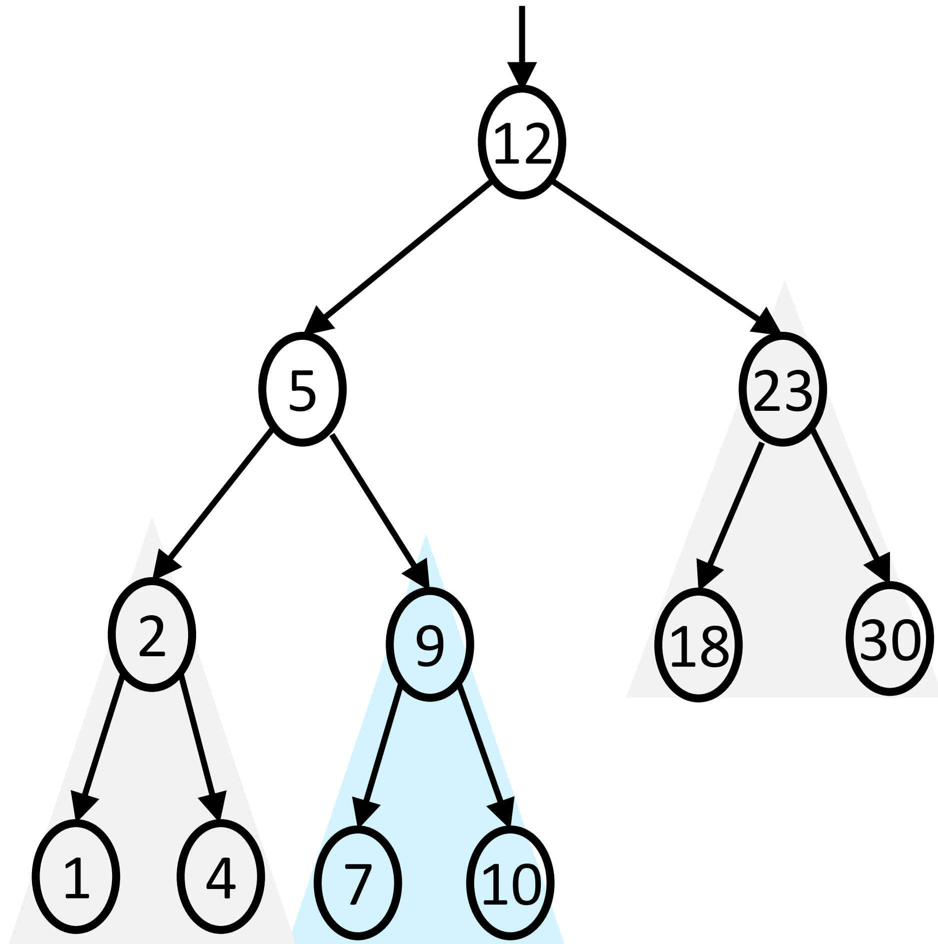
- ***Single rotation:*** The basic operation we'll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the "other" child (always okay in a BST!)
  - Other subtrees move in only way BST allows (we'll see in generalized example)

TREE ROTATIONS: GENERALIZED

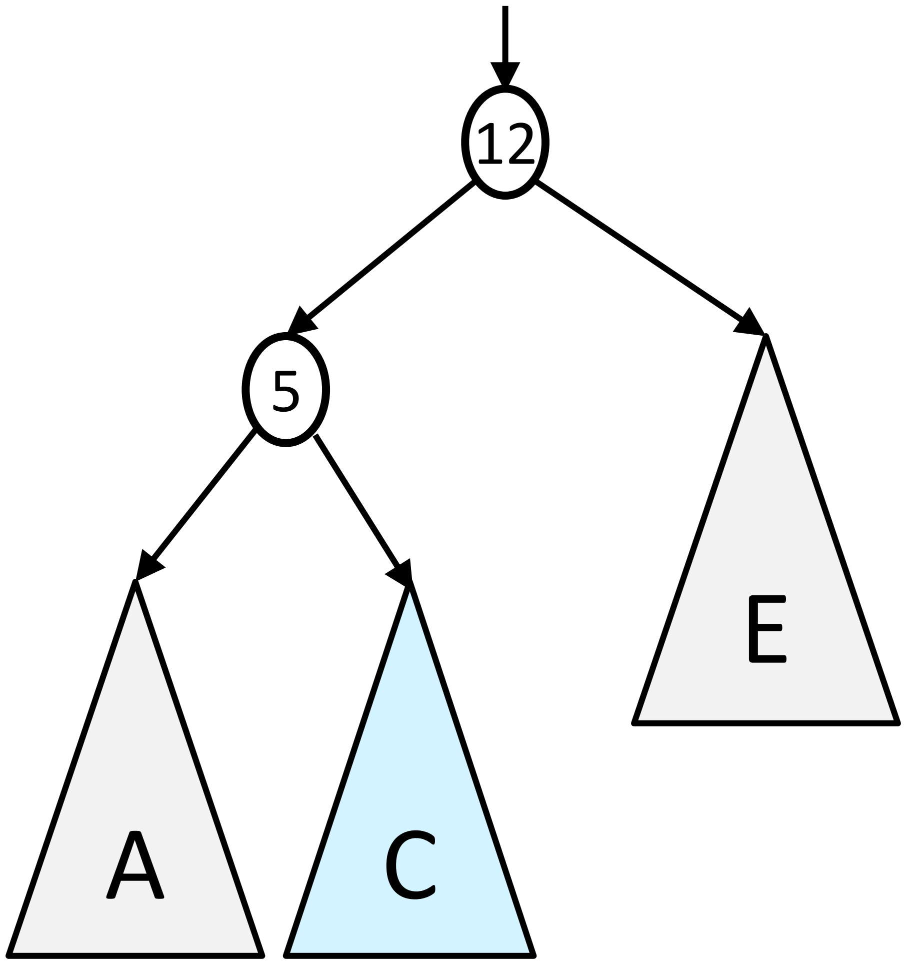
Generalizing our examples…



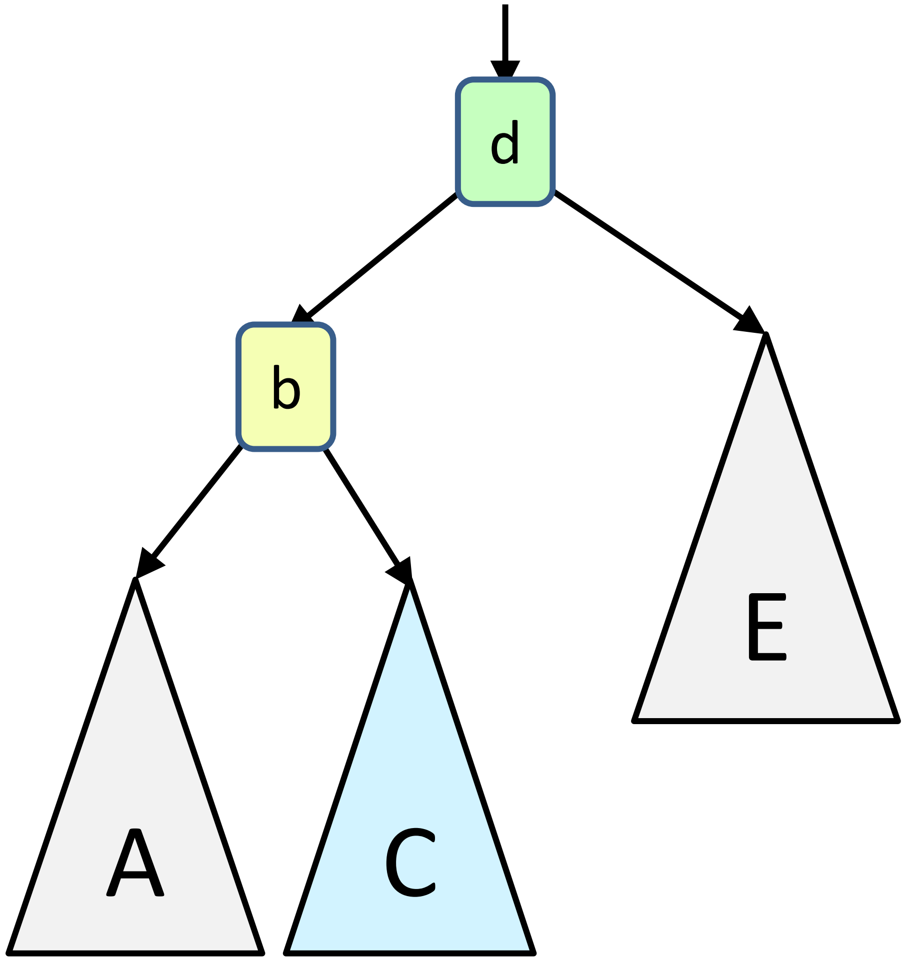
Generalizing our examples…



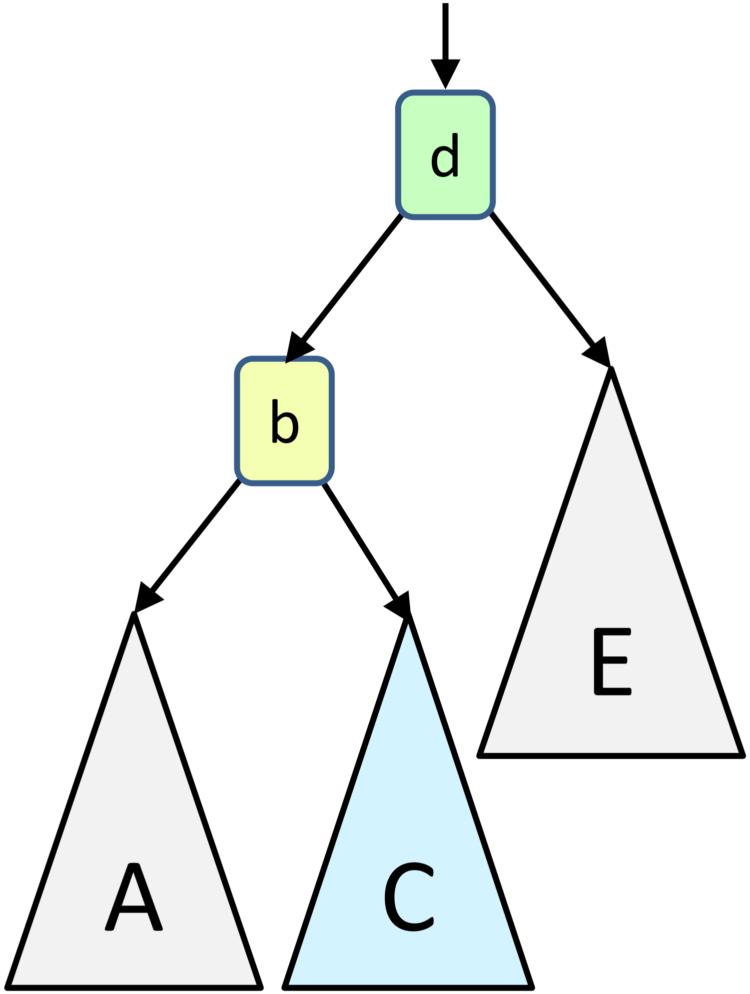
Generalizing our examples…



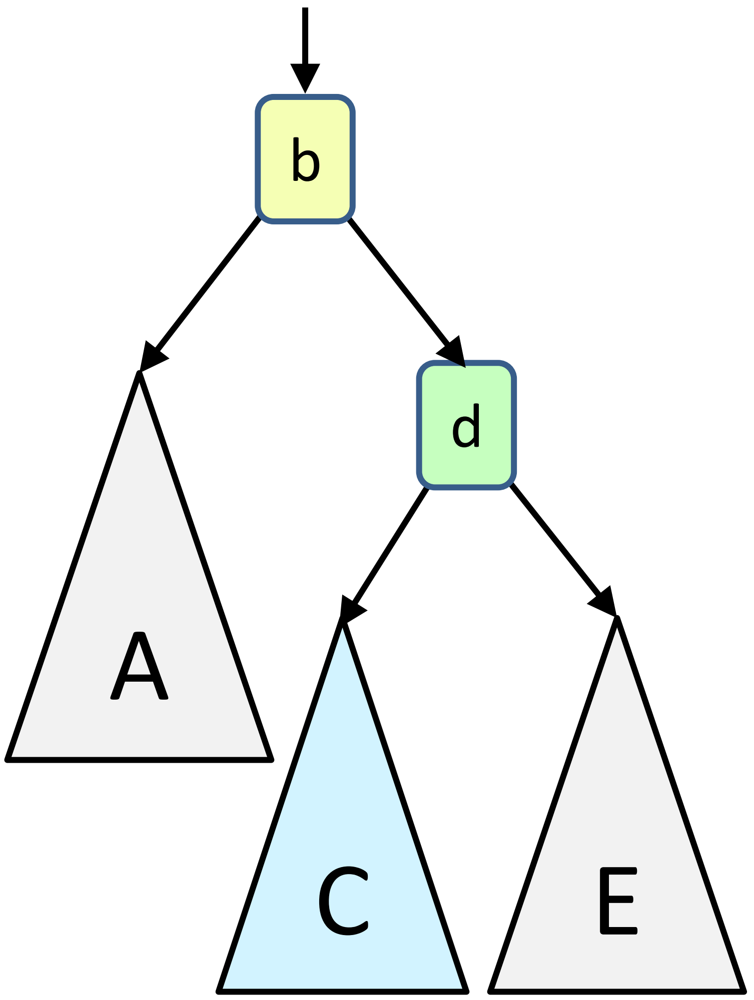
Generalizing our examples...



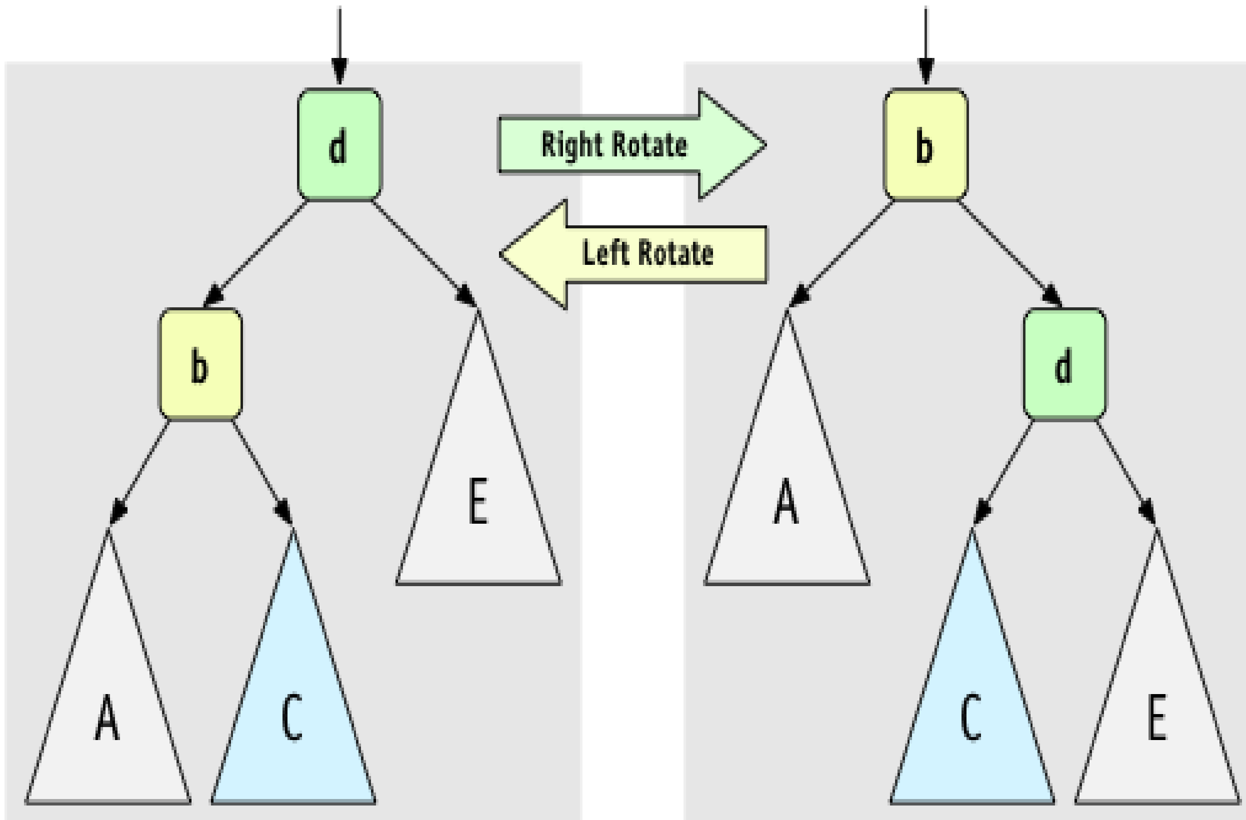
Generalized Single Rotation



Generalized Single Rotation



Single Rotations



(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Rotations

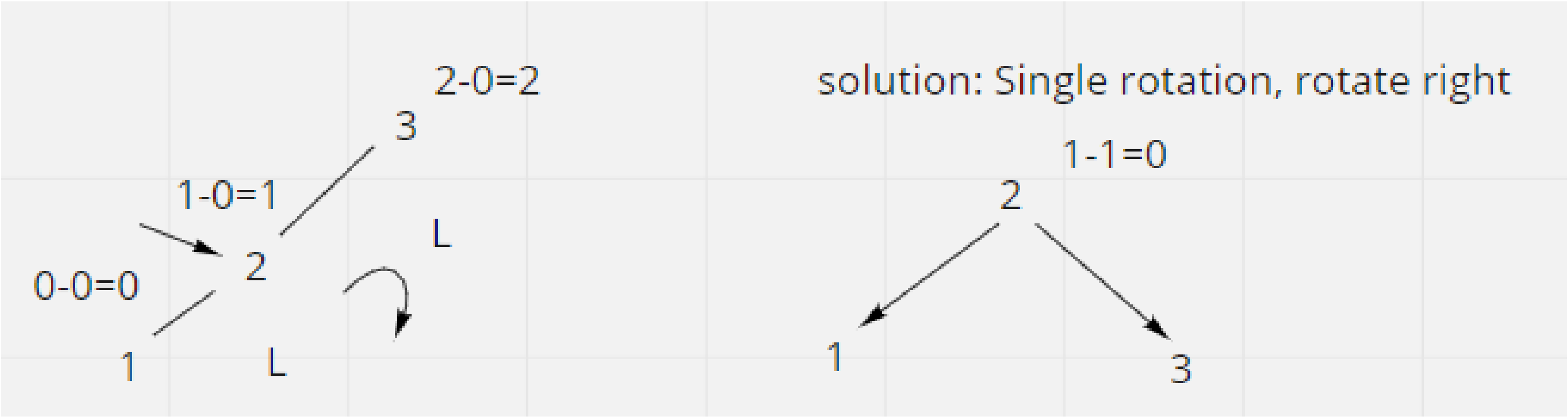
- Insert 1,2,3
- Right-Right or R-R case
- Solution: rotate left

Insertion

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of left(x) and right(x) differ by at most 1.
- If yes, proceed to parent(x). If not, restructure by doing either a single rotation or a double rotation
- For insertion, once we perform a rotation at a node x, we won’t need to perform any rotation at any ancestor of x.

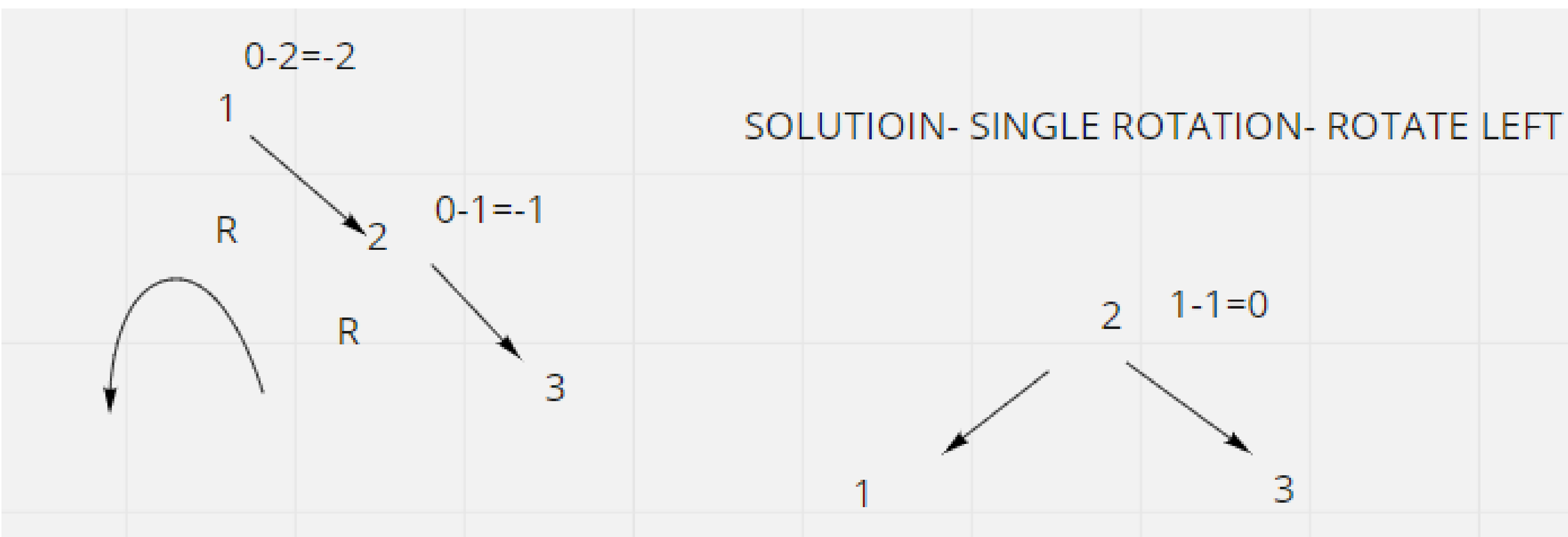
Rotations

- Insert 3,2,1
- Left-Left or L-L situation
- Solution: Rotate right



Rotations

- Insert 1,2,3
- Right-Right or R-Right situation
- Solution: one rotation



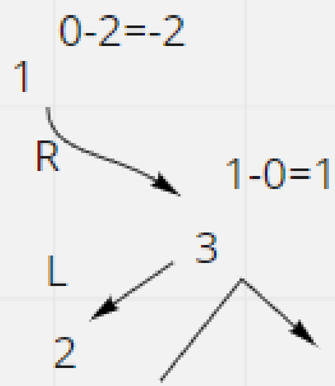
Rotations

- Insert 1,3,2
- Right-Left or R-L situation
- Solution: Two rotations
  - Rotate right ➔ R-R
  - Rotate left

Rotations

- Insert 1,3,2
- Right-Left or R-L situation
- Solution: Two rotations
  - Rotate right ➔ R-R
  - Rotate left

INSERT 1,3,2



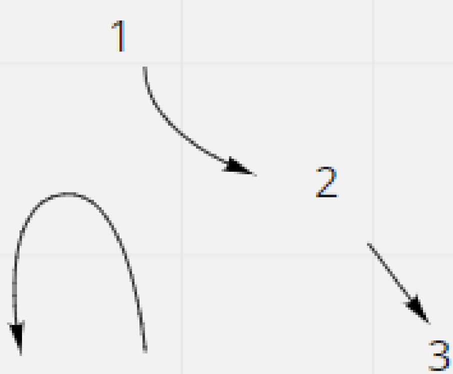
R-L

SOLUTION: 2 ROTATIONS

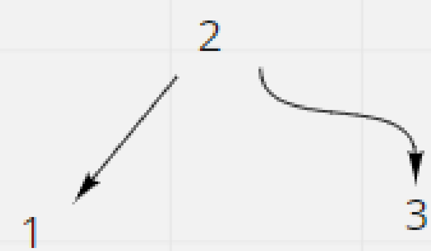
RORATE RIGHT-- R-R

ROTATE LEFT - BALANCED TREE

rotate right



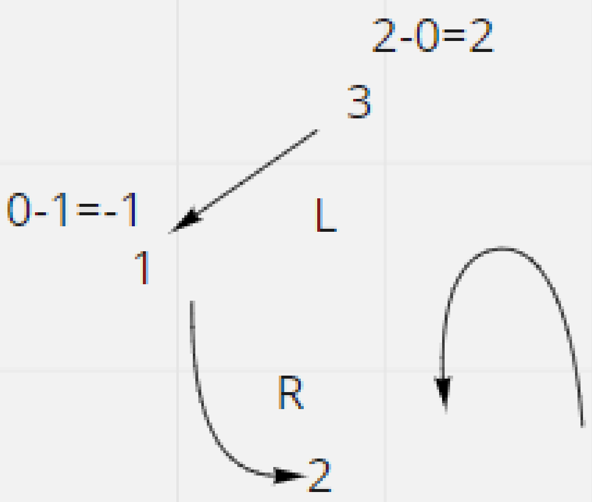
rotate left



Rotations

- Insert 3,1, 2
- Left-Right or L-R situation
- Solution: Two rotations
  - Rotate left➔ L-L
  - Rotate right

insert 3,1,2

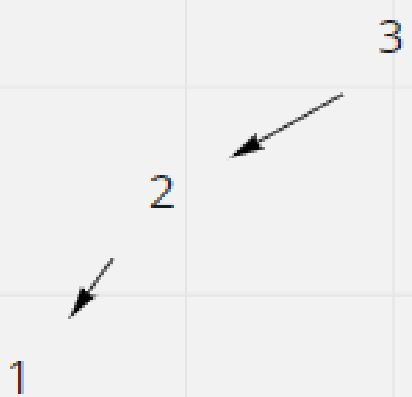


solution: two rotations

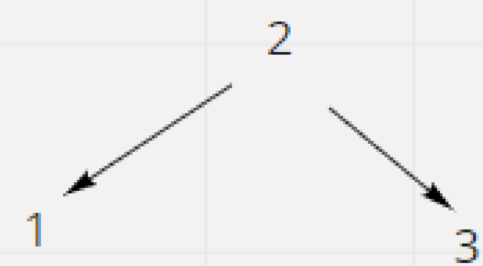
L-R --> rotate to left to make L-L

rotate right to balance tree

rotate left



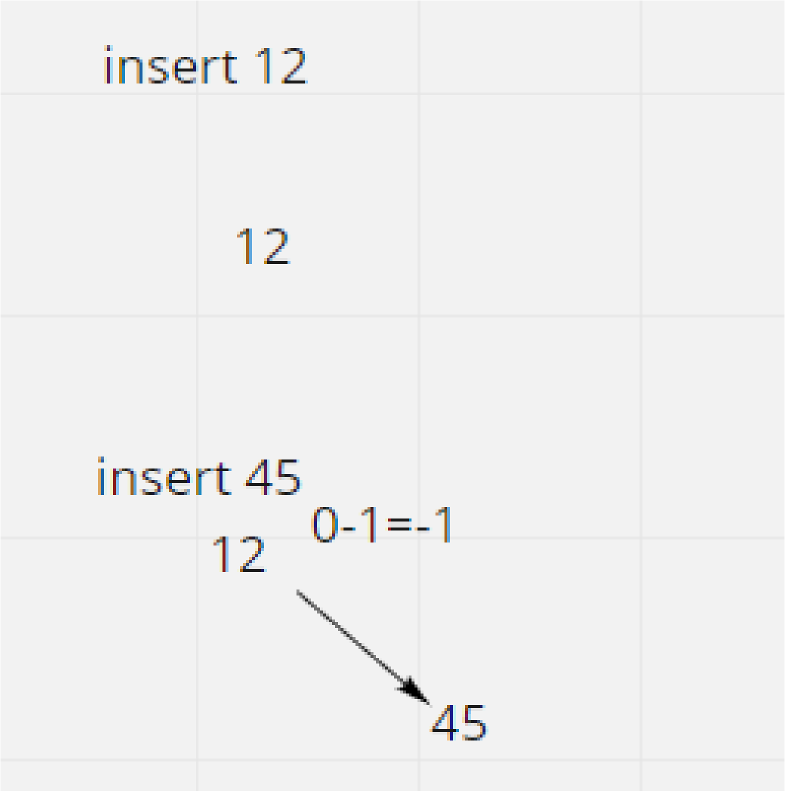
rotate right

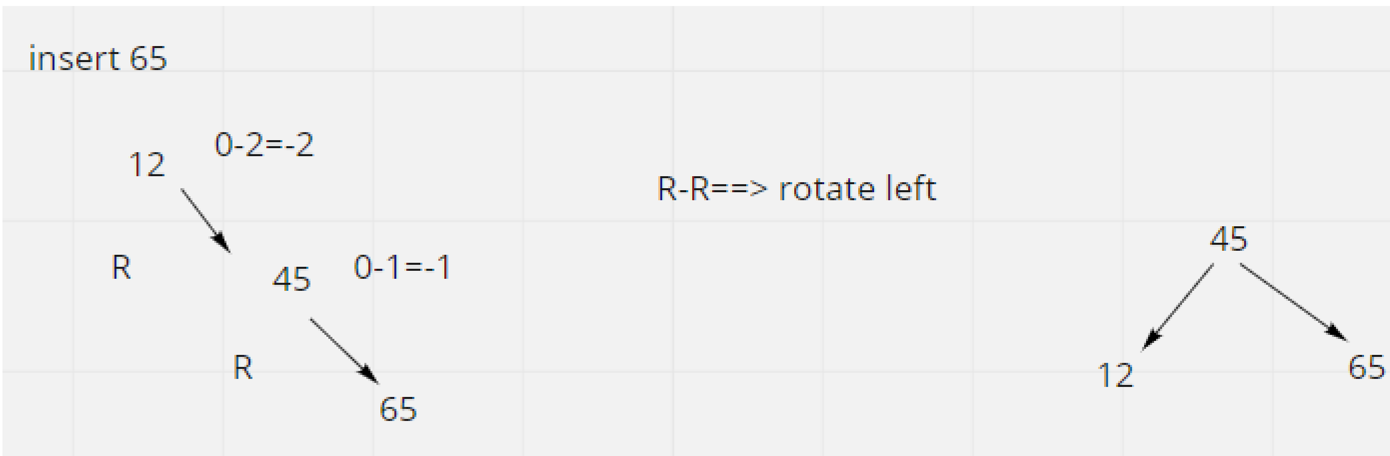


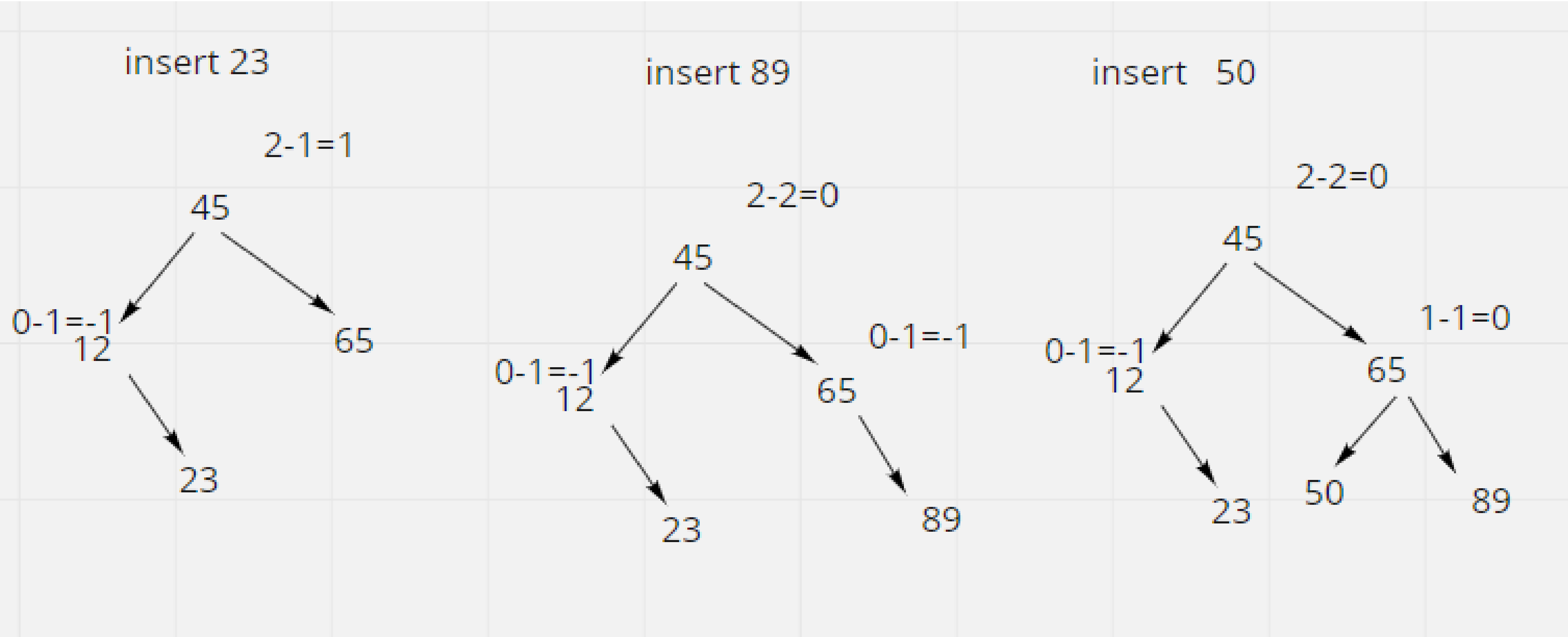
Rotation summary

- L-L then single rotation --> rotate right
- R-R then single rotation--> rotate left
- L-R then double rotation --> rotate left to get L-L then rotate right
- R-L then double rotation --> rotate right to get R-R then rotate left

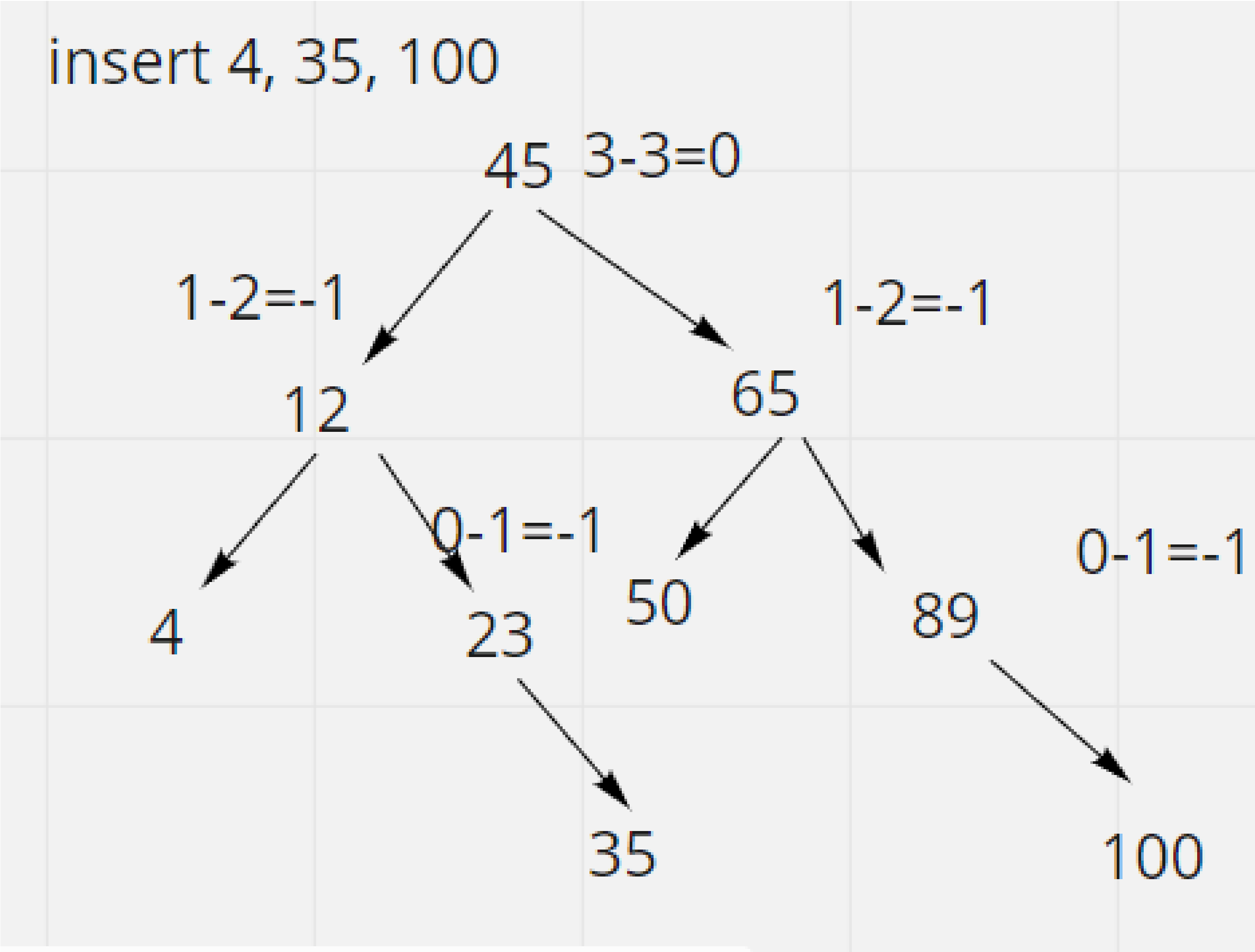
Example- 12, 45, 65, 23, 89, 50, 4, 35,100



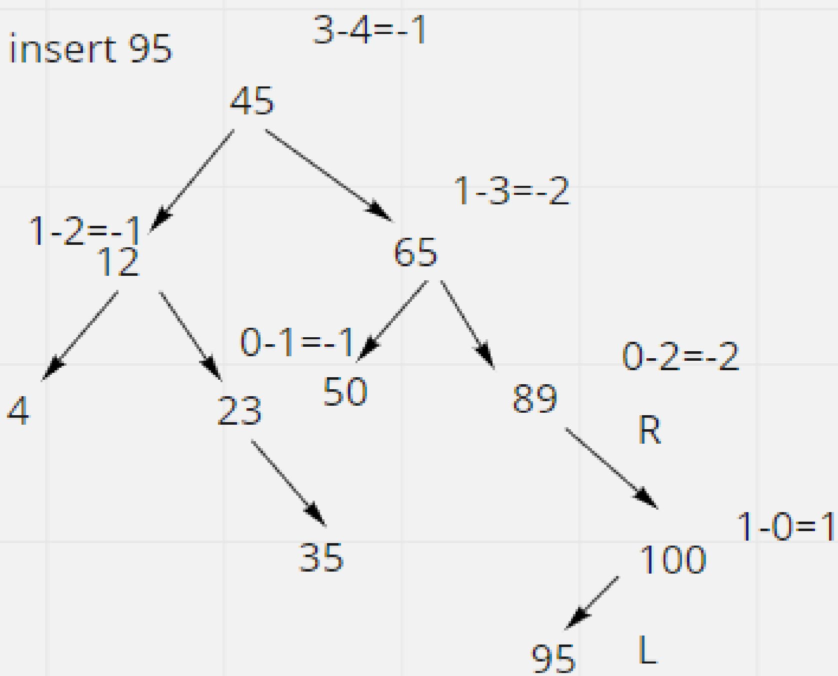

insert 65
12
0-2=-2
R
45
0-1=-1
R
65
R-R==> rotate left
45
12
65



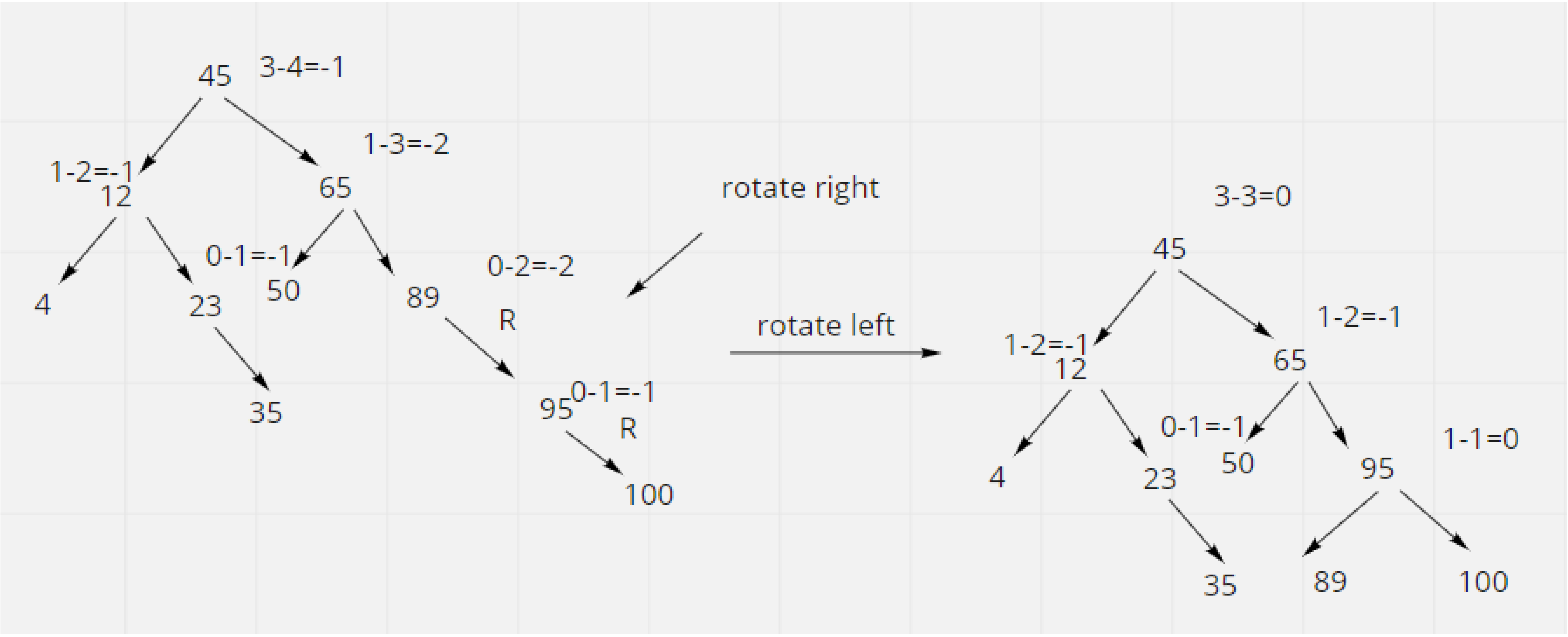
insert 23
2-1=1
45
0-1=-1
12
65
23
insert 89
2-2=0
45
0-1=-1
12
65
0-1=-1
23
89
insert 50
2-2=0
45
0-1=-1
12
1-1=0
65
23
50
89



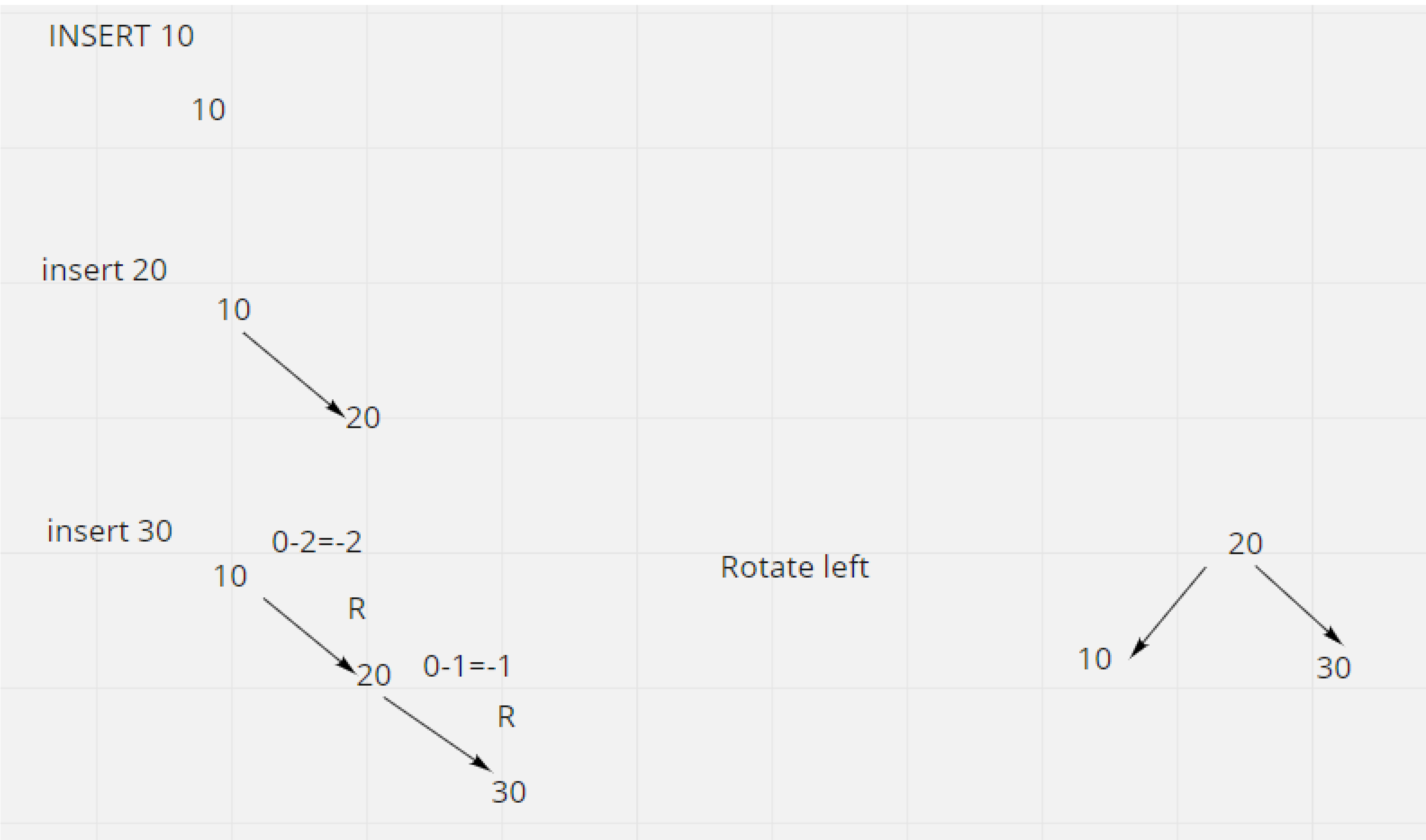
insert 4, 35, 100
45
3-3=0
1-2=-1
12
1-2=-1
65
0-1=-1
4
23
50
89
0-1=-1
35
100



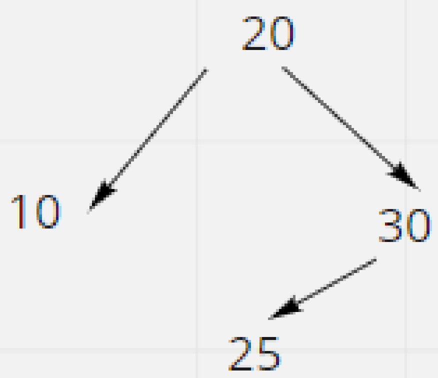
R-L ==> convert to R-R with rotate right
then
rotate left


45
3-4=-1
1-3=-2
1-2=-1
12
65
4
23
0-1=-1
50
89
0-2=-2
R
35
95
0-1=-1
R
100
rotate right
rotate left
3-3=0
45
1-2=-1
1-2=-1
12
65
0-1=-1
4
23
50
95
1-1=0
35
89
100

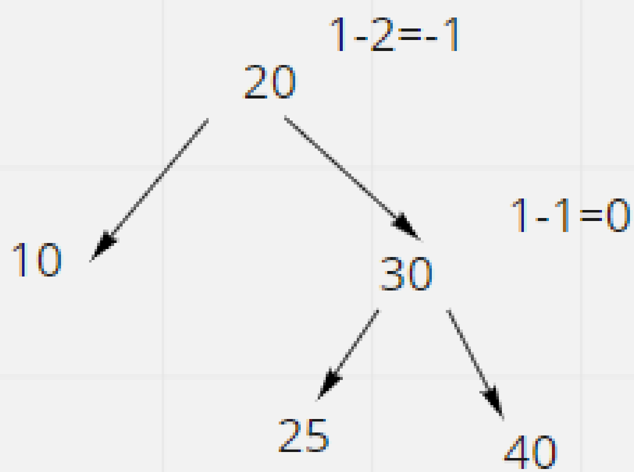
Create AVL tree:10, 20, 30, 25, 40, 50, 35, 33, 37, 60,38



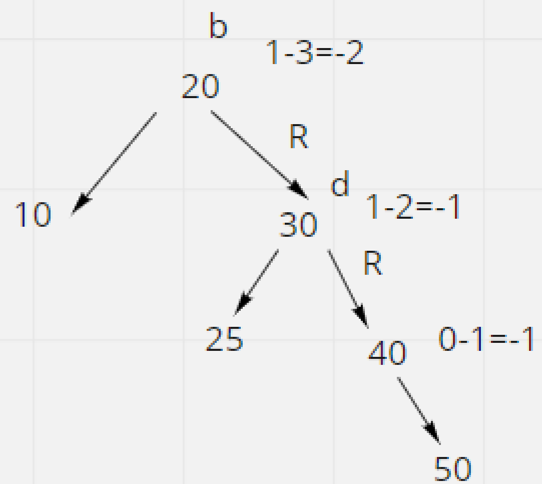
insert 25



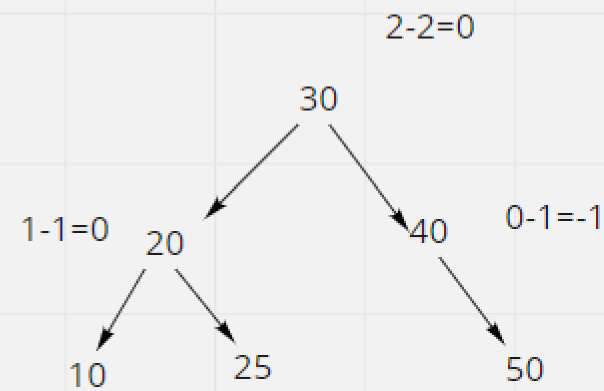
insert 40

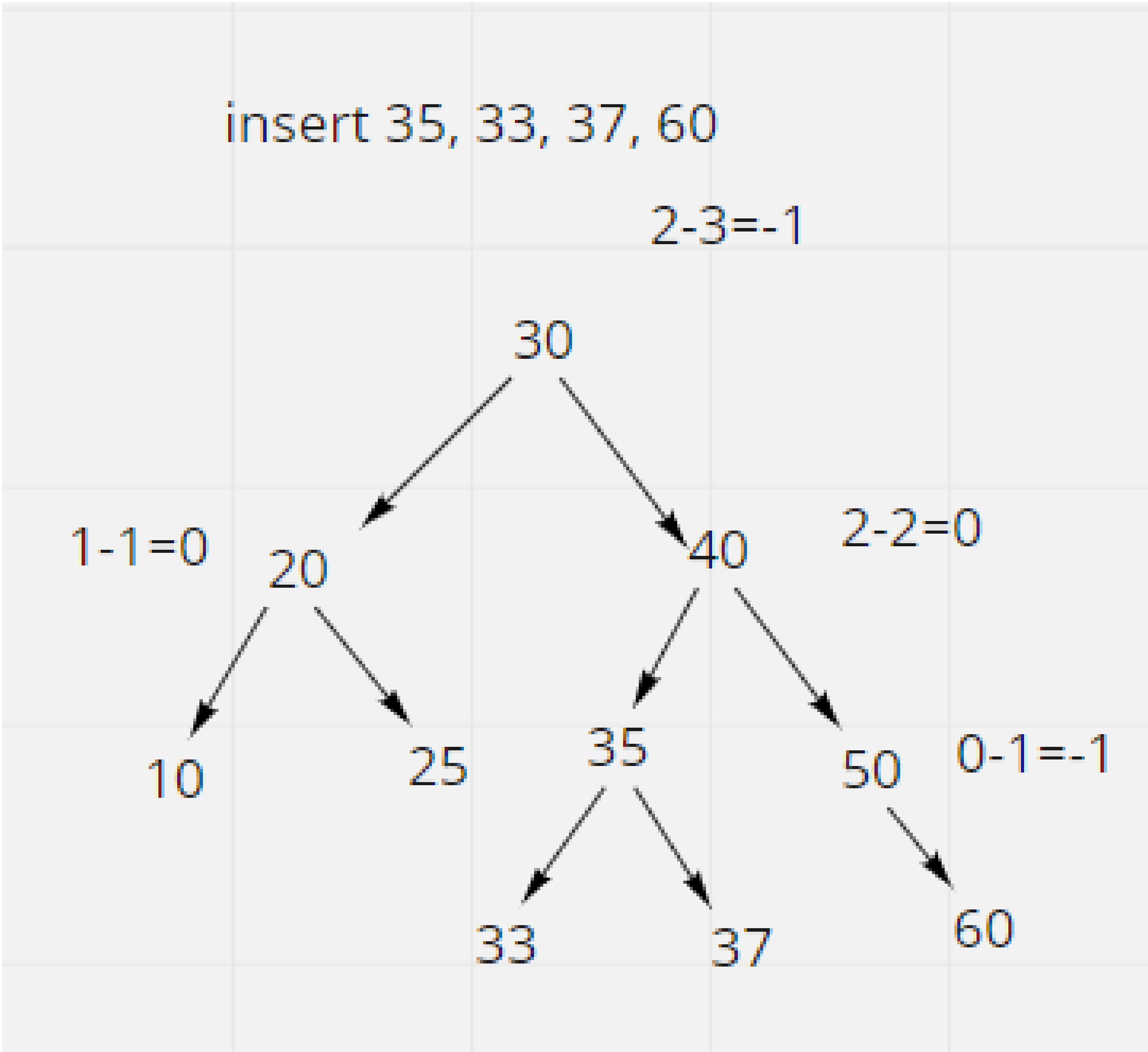


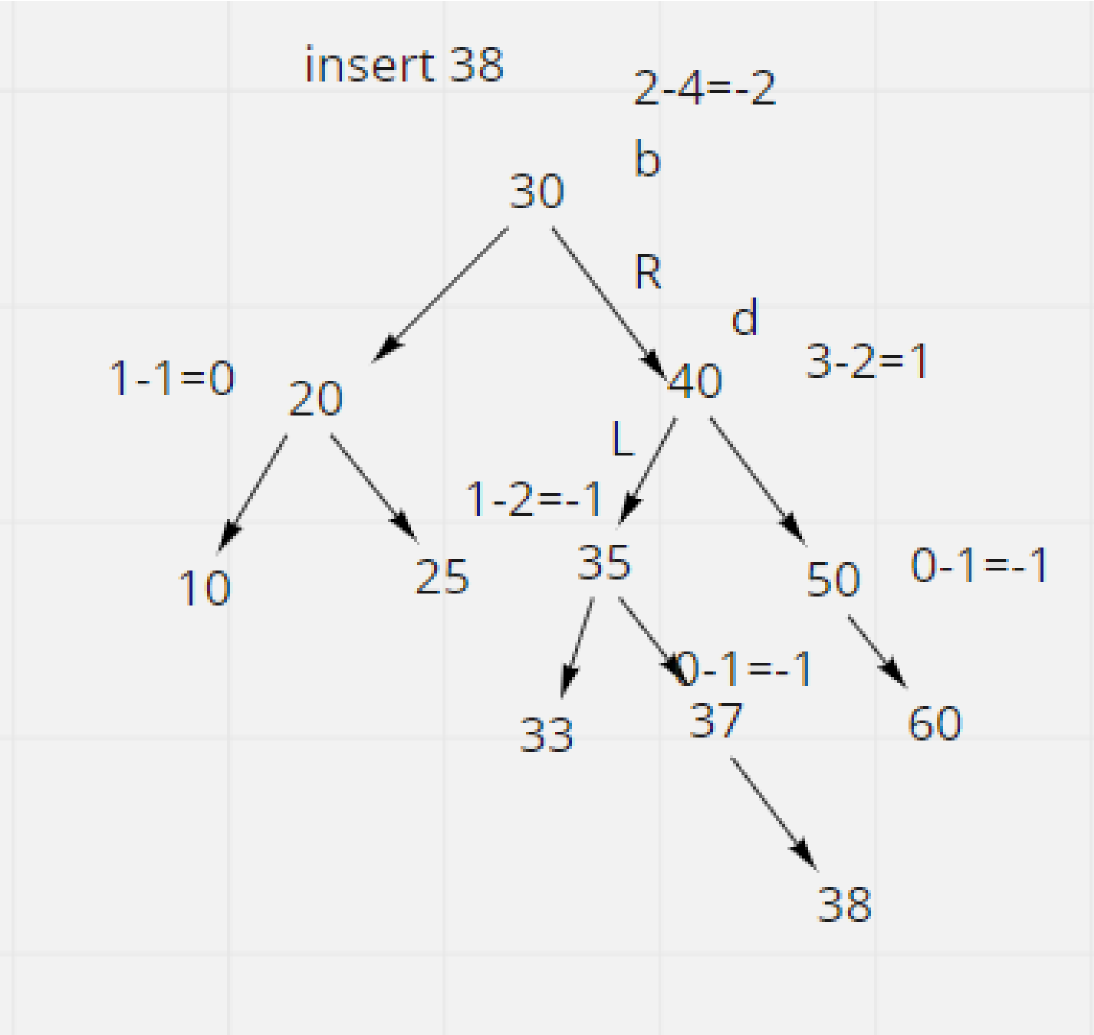
insert 50



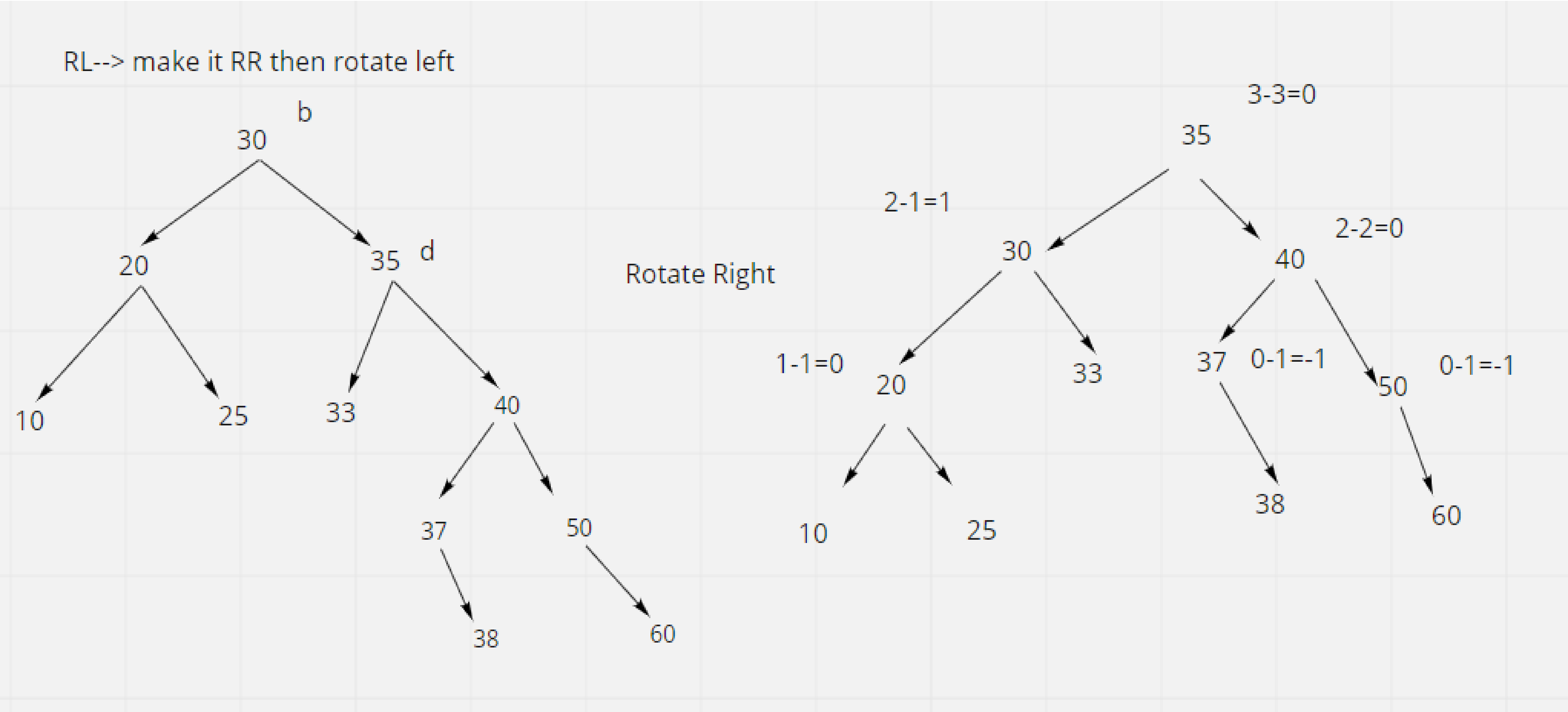
R-R so rotate left



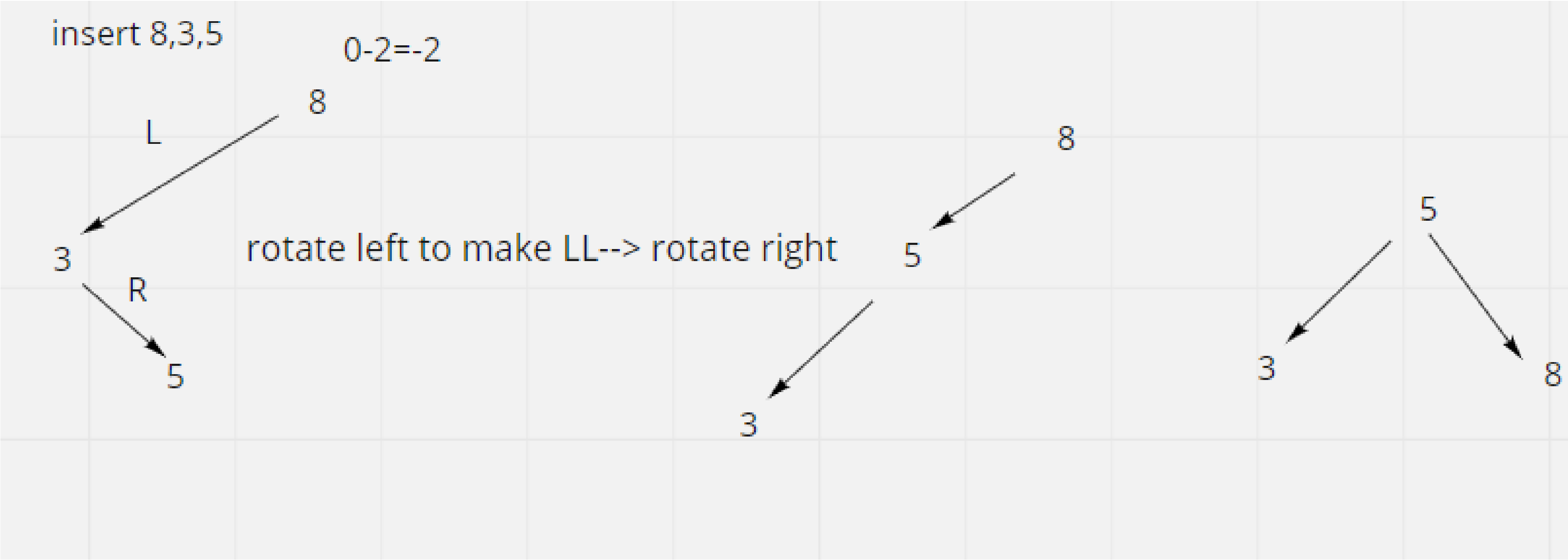

insert 35, 33, 37, 60
2-3=-1
30
1-1=0
20
40
2-2=0
10
25
35
50
0-1=-1
33
37
60

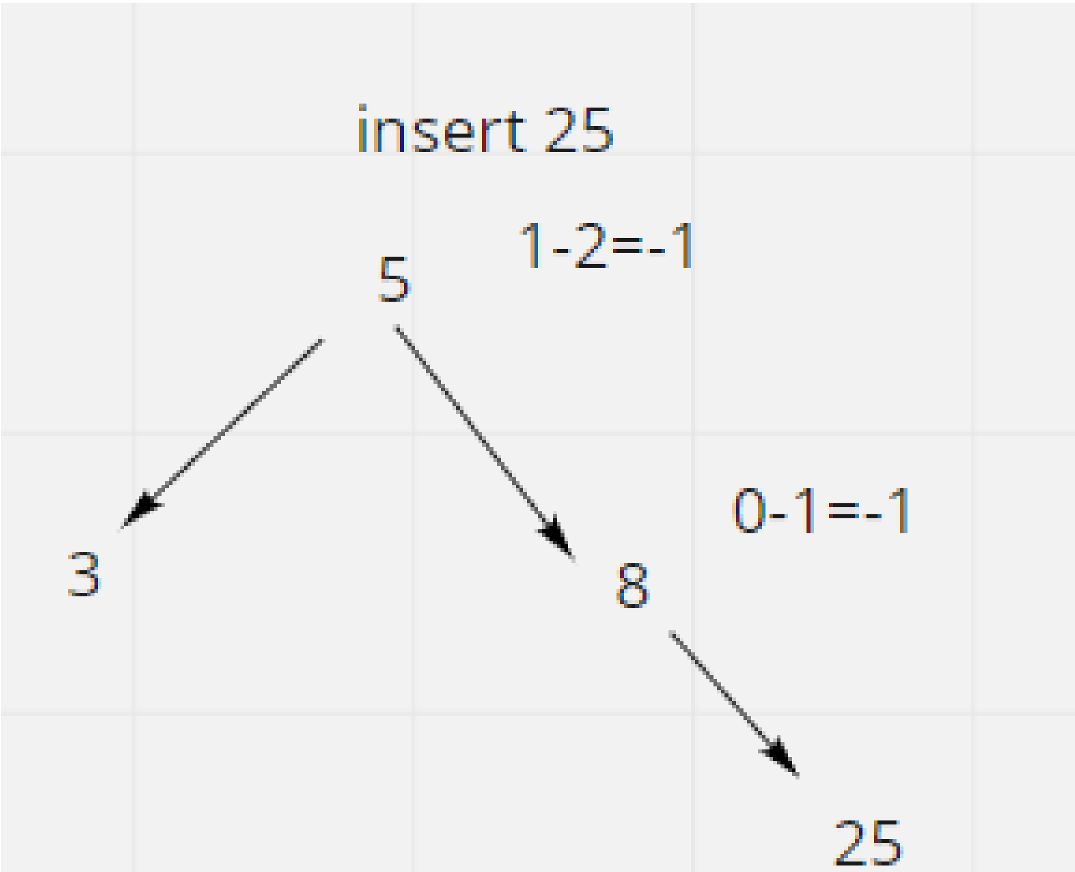

insert 38
2-4=-2
b
30
R
d
3-2=1
1-1=0
20
40
L
1-2=-1
10
25
35
50
0-1=-1
0-1=-1
33
37
60
38

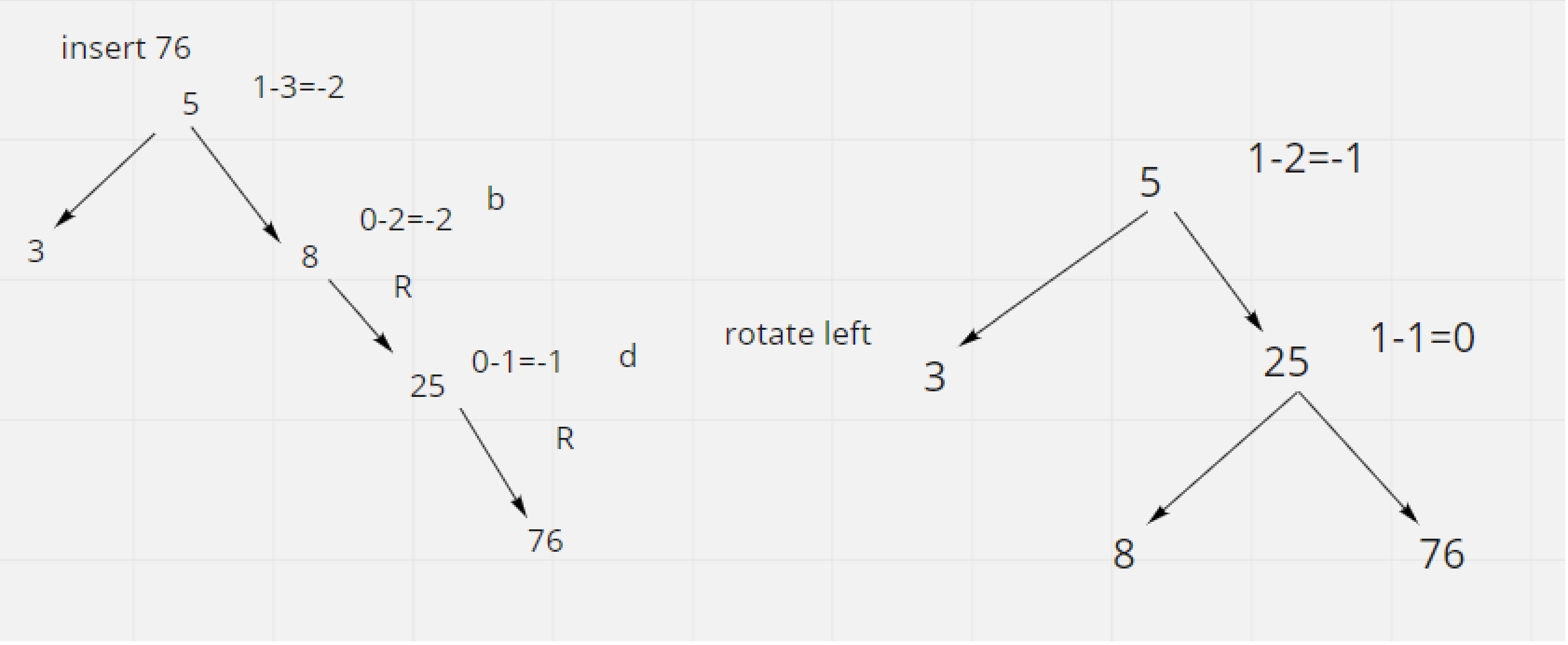
Balance the tree



Example 3: 8,3,5,25,76, 45, 30,26,28,27

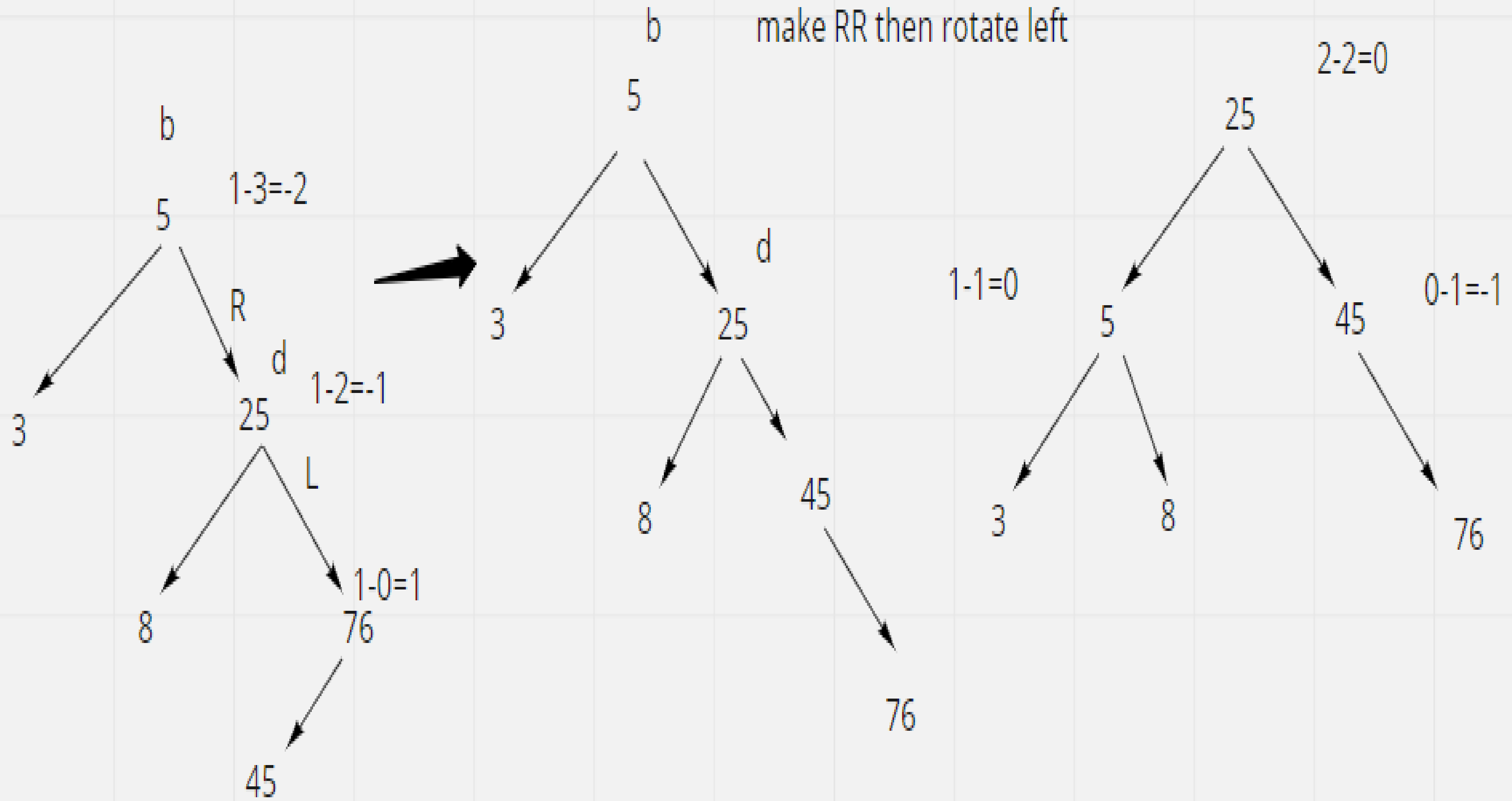



insert 25
1-2=-1
5
0-1=-1
3
8
25

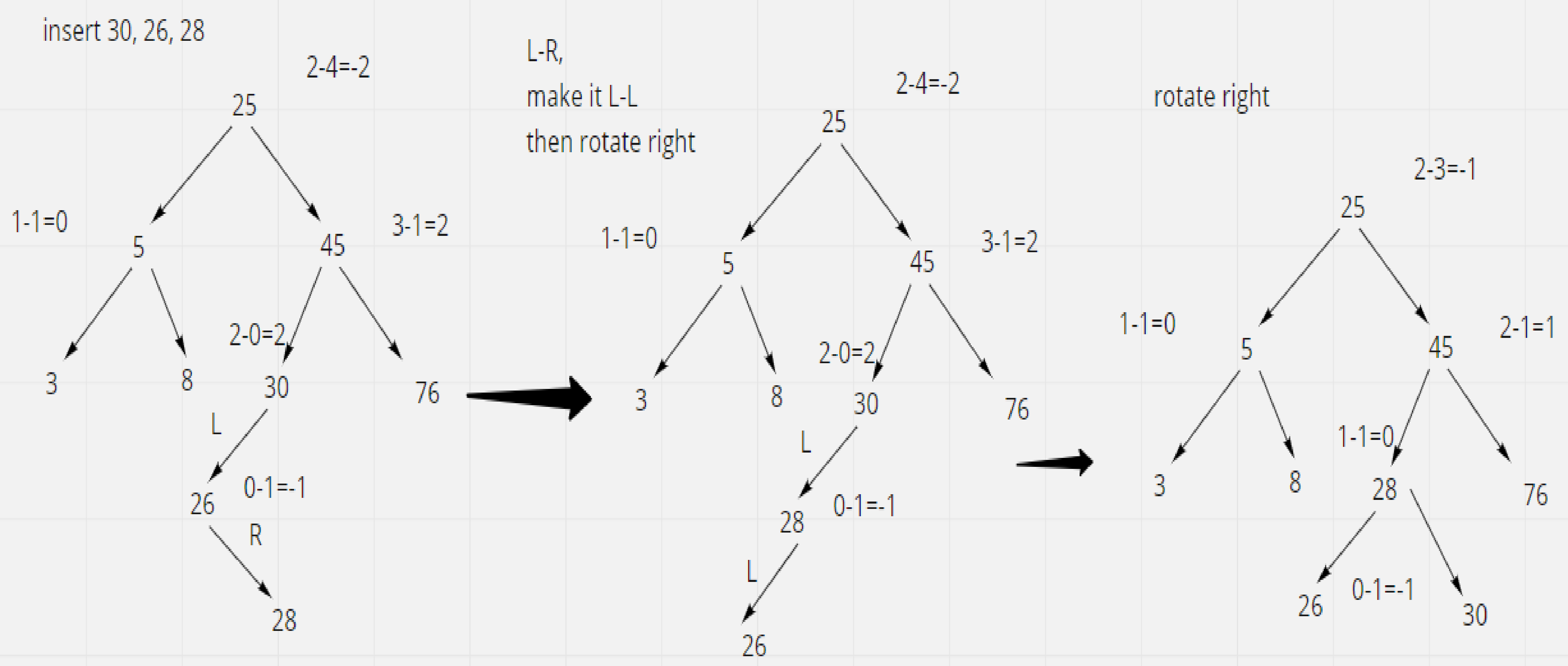


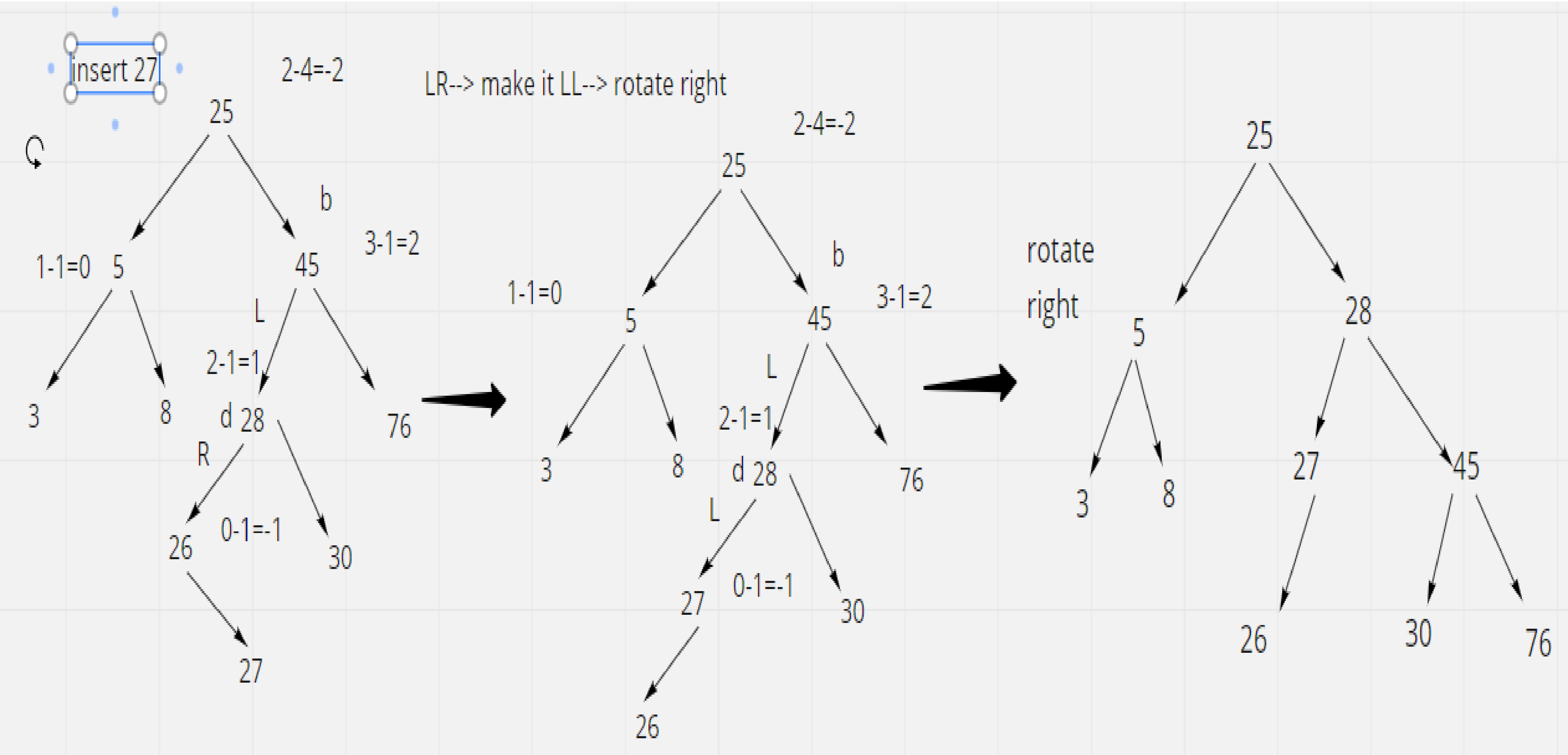
insert 76
5
1-3=-2
3
8
0-2=-2
b
R
25
0-1=-1
d
R
76
rotate left
5
1-2=-1
3
25
1-1=0
8
76

insert 45

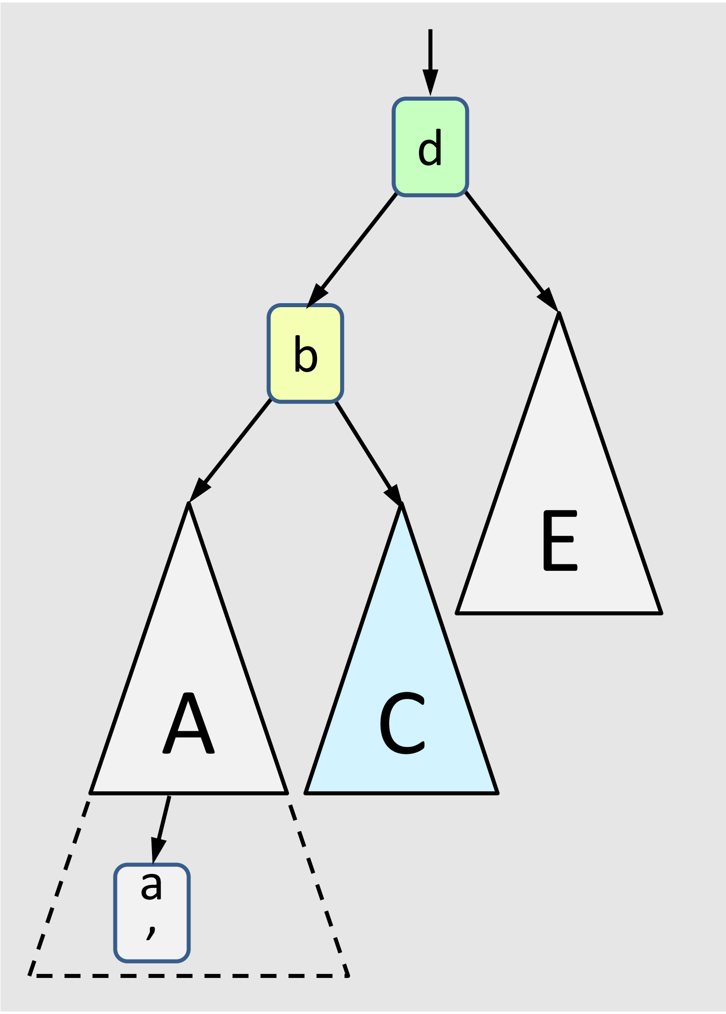


b
5
1-3=-2
R
3
d
25
1-2=-1
L
8
1-0=1
76
45
b
make RR then rotate left
5
d
3
25
8
45
76
2-2=0
25
1-1=0
5
45
0-1=-1
3
8
76


insert 30, 26, 28
2-4=-2
25
1-1=0
5
45
3-1=2
3
8
2-0=2
30
76
L
26
0-1=-1
R
28
L-R,
make it L-L
then rotate right
2-4=-2
25
1-1=0
5
45
3-1=2
3
8
2-0=2
30
76
L
28
0-1=-1
L
26
rotate right
2-3=-1
25
1-1=0
5
45
2-1=1
3
8
1-1=0
28
76
26
0-1=-1
30


insert 27
2-4=-2
LR--> make it LL--> rotate right
25
b
3-1=2
1-1=0
5
45
L
2-1=1
3
8
d 28
76
R
26
0-1=-1
30
27
2-4=-2
25
b
3-1=2
1-1=0
5
45
L
2-1=1
3
8
d 28
76
L
27
0-1=-1
30
26
rotate
right
25
5
28
3
8
27
45
26
30
76

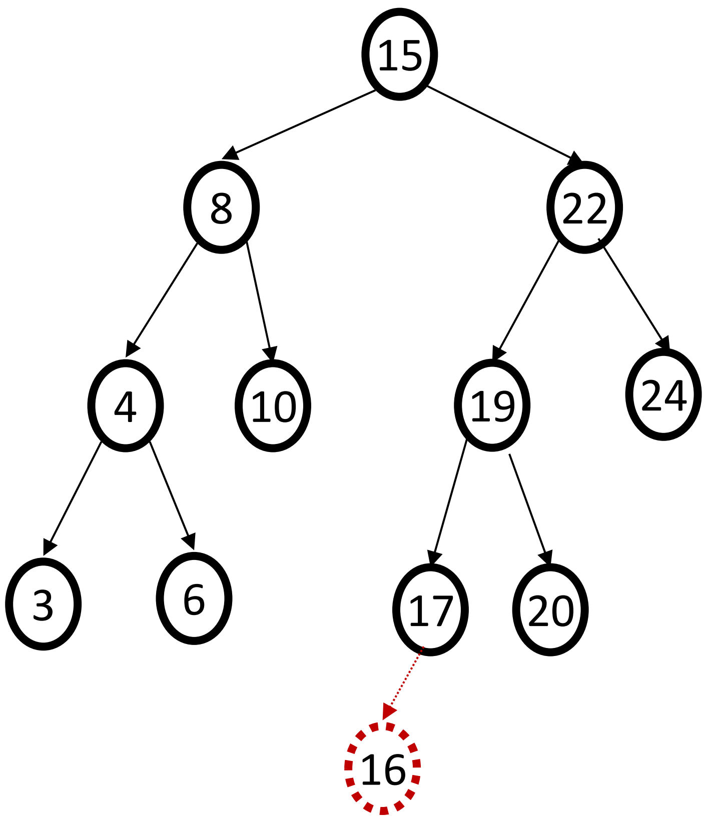
Case #1:



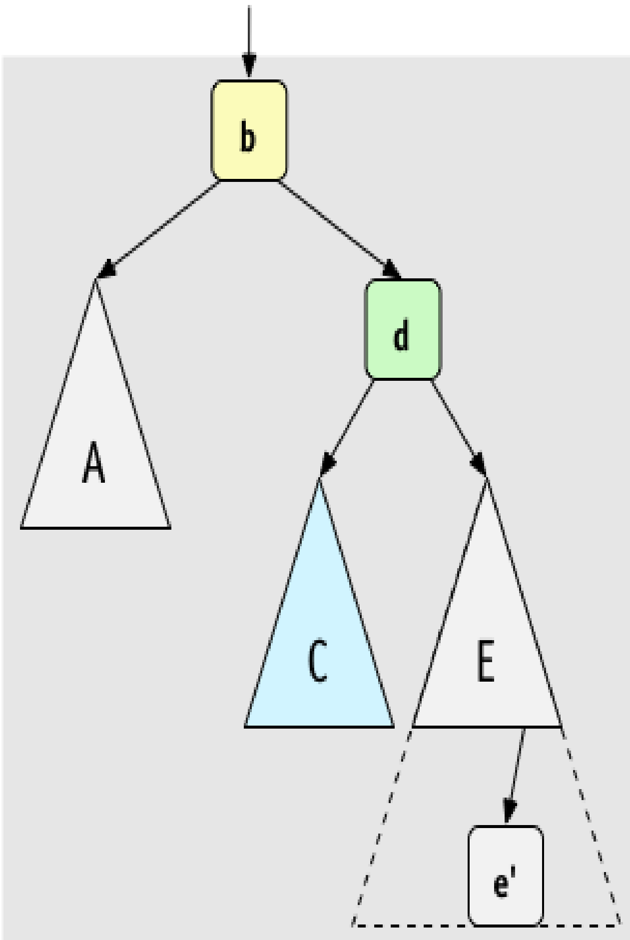
(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Example #2 for left-left case:

`insert(16)`



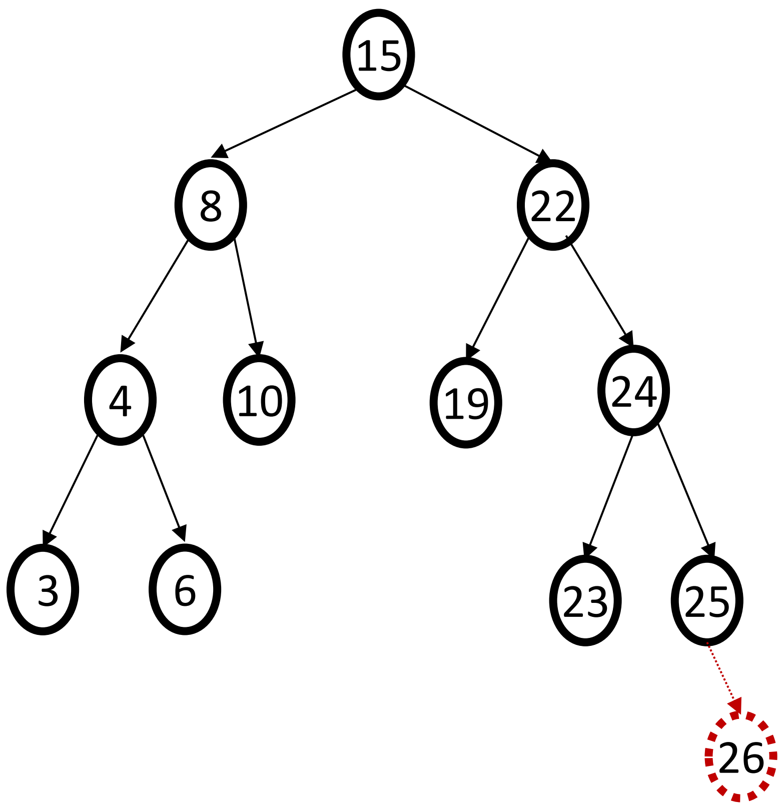
Case #2:

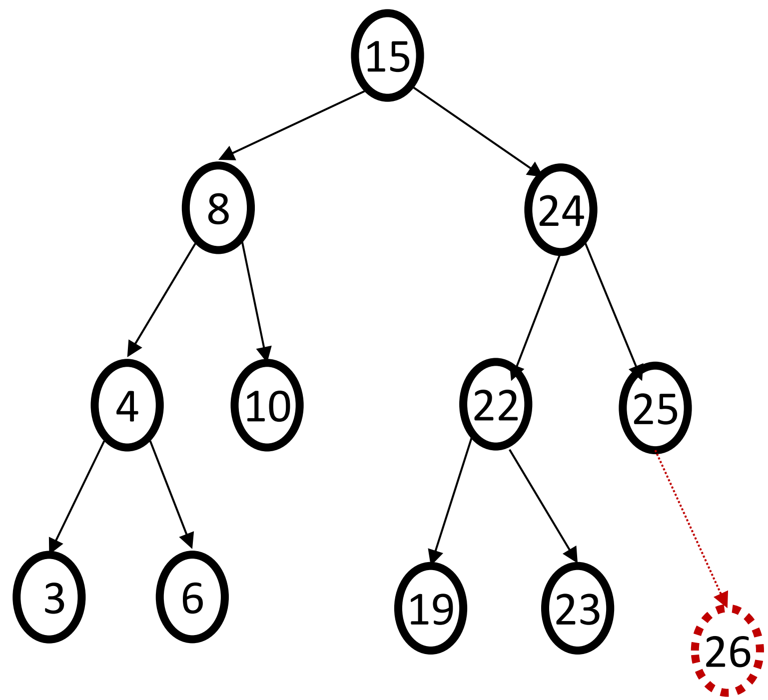


(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

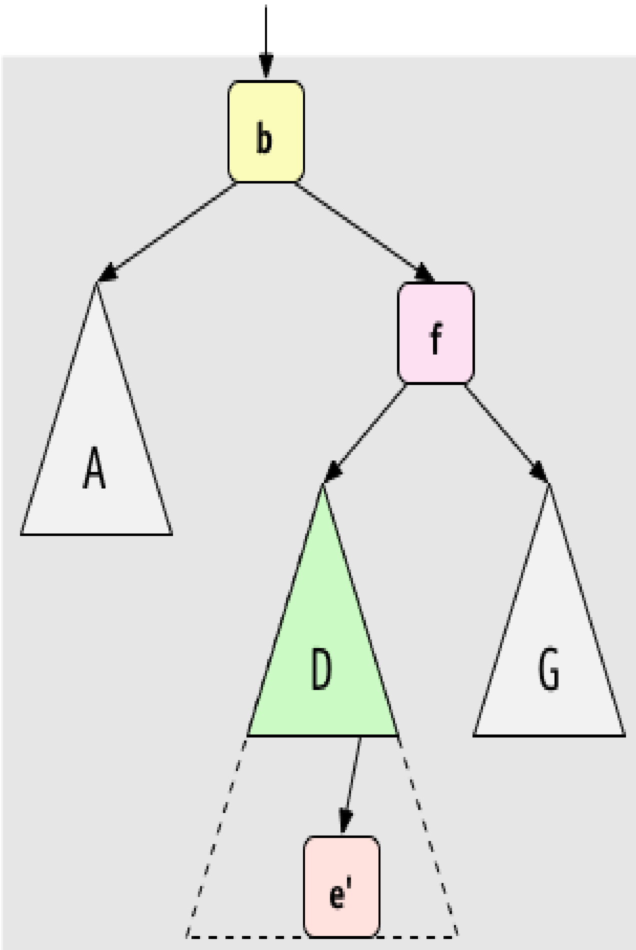
Example for right-right case:

`insert(26)`



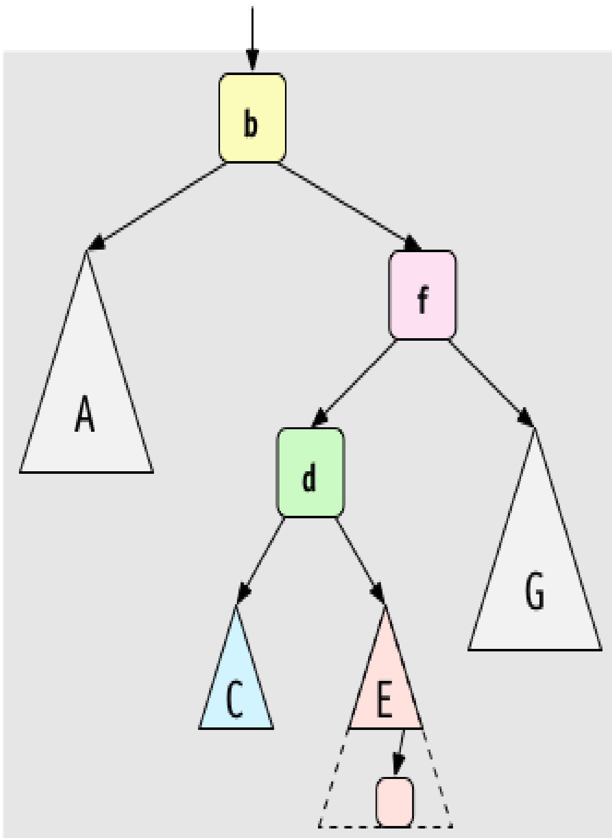


Case #3:



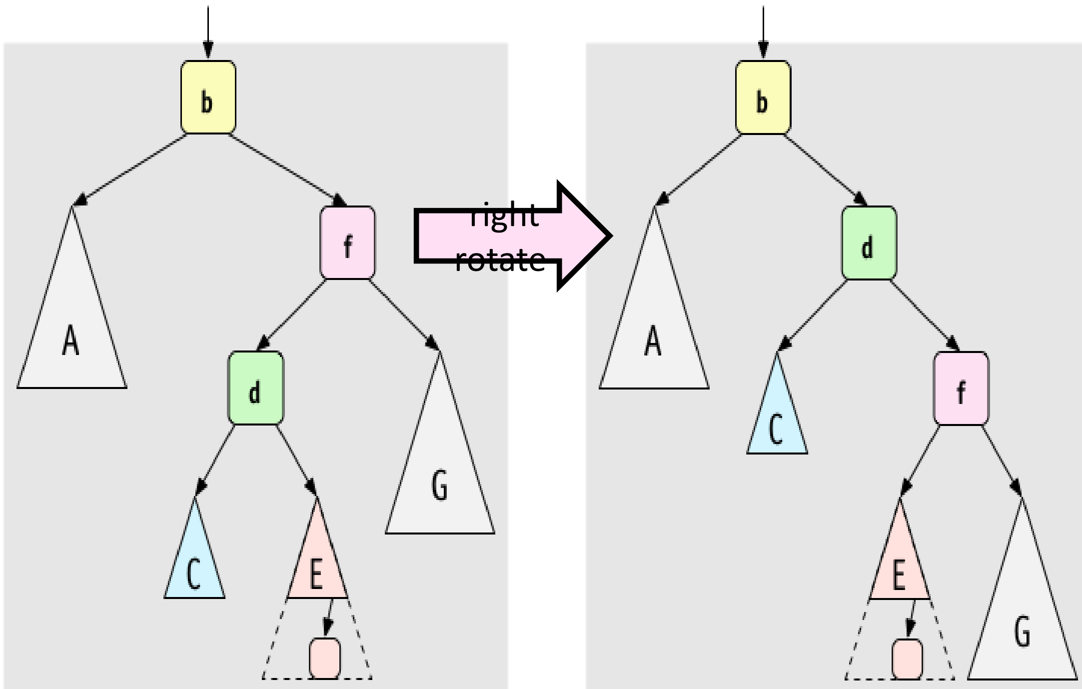
(Figures by Melissa O’Neill, reprinted with her permission to Lilian)

A Better Look at Case #3:



(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

Case #3: Right-Left Case (after one rotation)

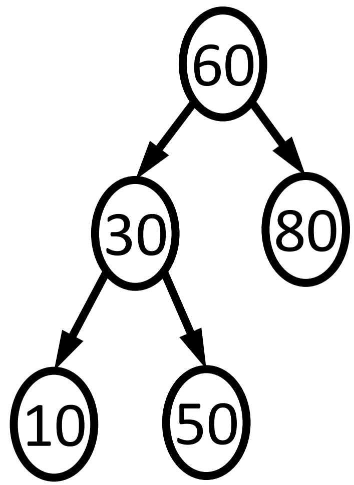


A way to remember it:
Move d to grandparent's position. Put everything else in their only legal positions for a BST.

(Figures by Melissa O'Neill, reprinted with her permission to Lilian)

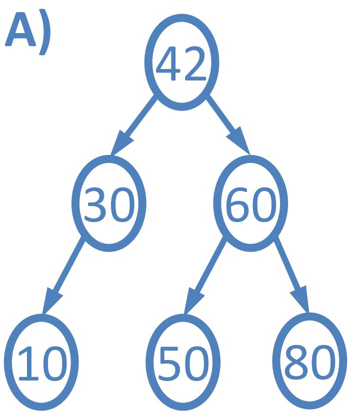
Practice time! Example of Case #4

Starting with this AVL tree:

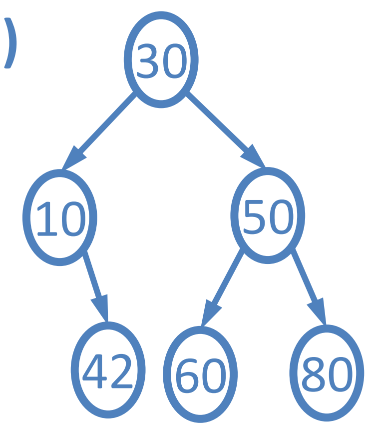


Which of the following is the updated AVL tree after inserting 42?

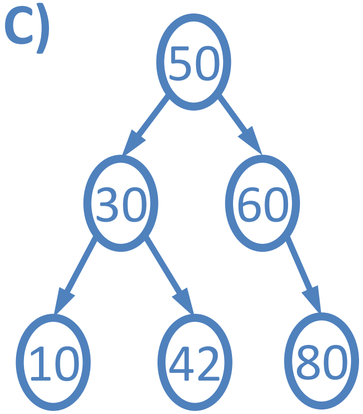
A)



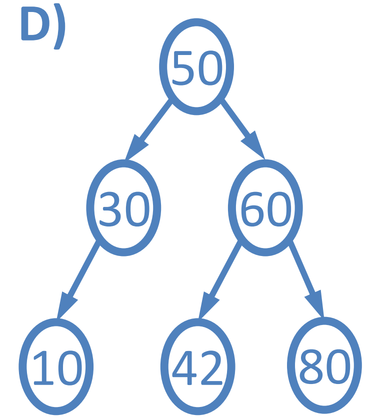
B)



C)



D)



Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of `insert` and `delete`

Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. If *amortized* logarithmic time is enough, use splay trees (also in the text, not covered in this class)

Queries?

Thank you!