Data Structure
Unit 3
Trees

Presented by
Dr. Mamata P. Wagh
CGU, Odisha

COURSE OUTCOMES:

Upon successful completion of this course, students will be able to:

- **CO1:** Analyze and design of linear data structure such as Queue and Stack for solving problems.
- C**O2:** Analyze and design of linear data structure such as Linked List for solving problems.
- **CO3:** Analyze and design of non-linear data structure such as Tree for solving problems.
- **CO4:** Analyze and design of non-linear data structure such as Graph for solving problems.
- **CO5:** Use different sorting and searching mechanisms by analyzing suitability.

Basic Terminology

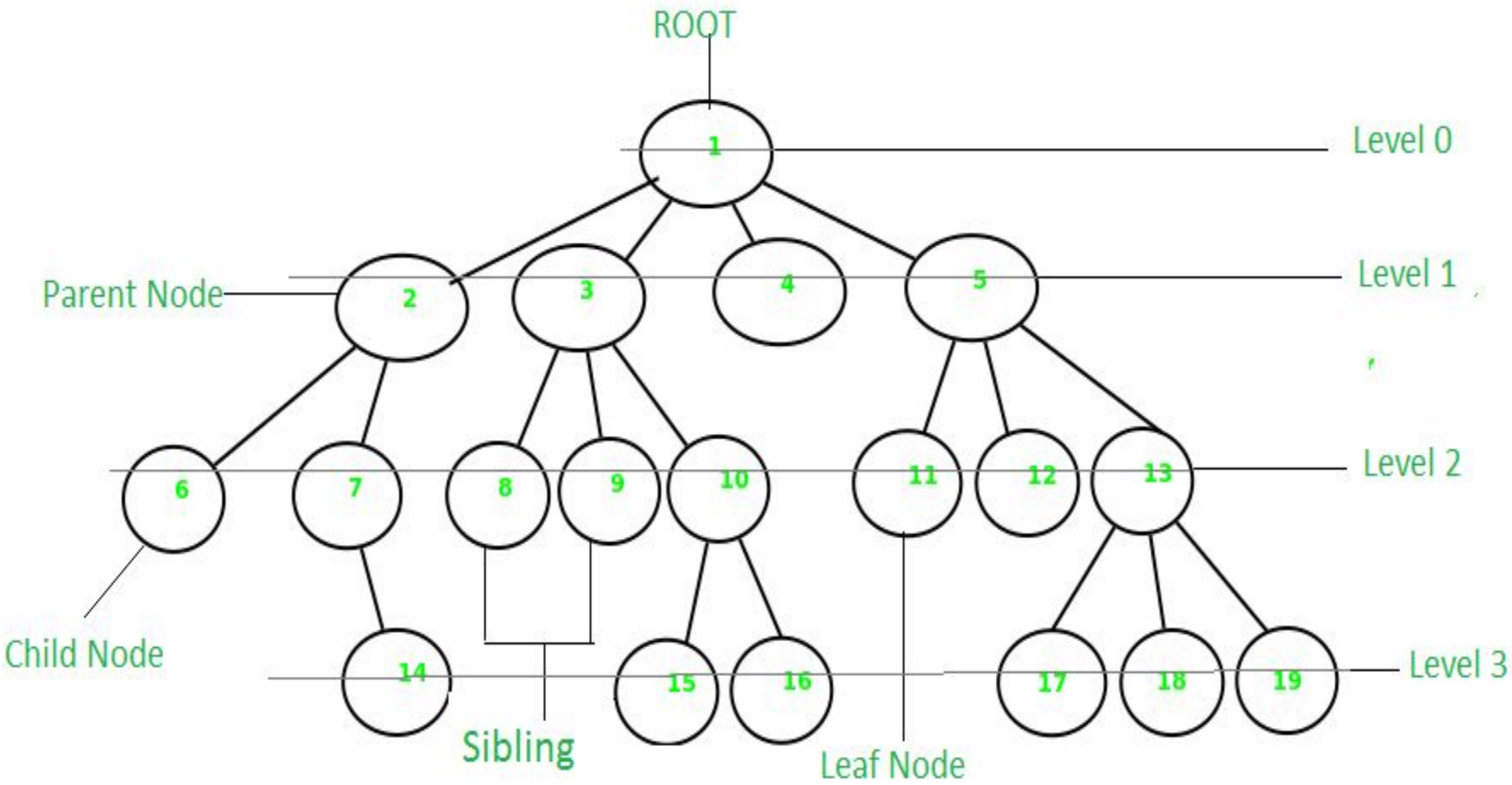
Tree is a non-linear data structure which presents a hierarchical structure of elements.

A tree is a graph with one and only path existing between every pair of vertices.

The terminologies include:

- Root
- Edge
- Parent
- Child
- Siblings
- Degree
- Internal Node
- Leaf Node
- Level
- Depth
- Sub-Tree
- Forest

Basic Terminology

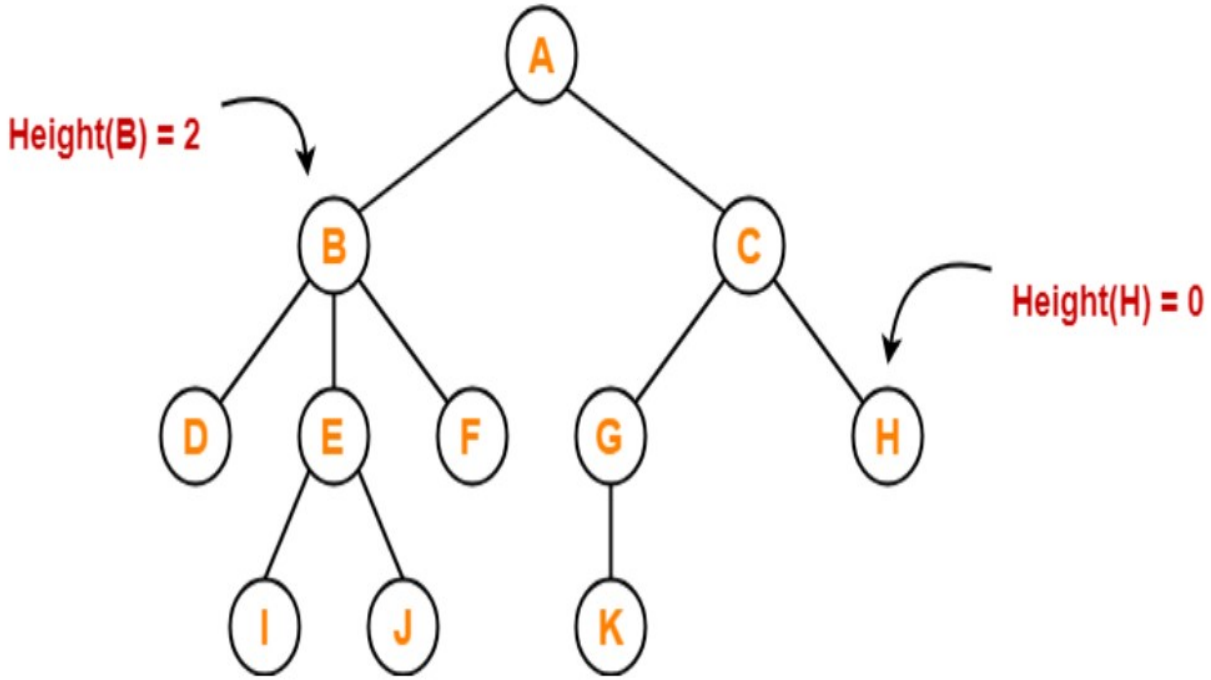


Height of a Tree

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



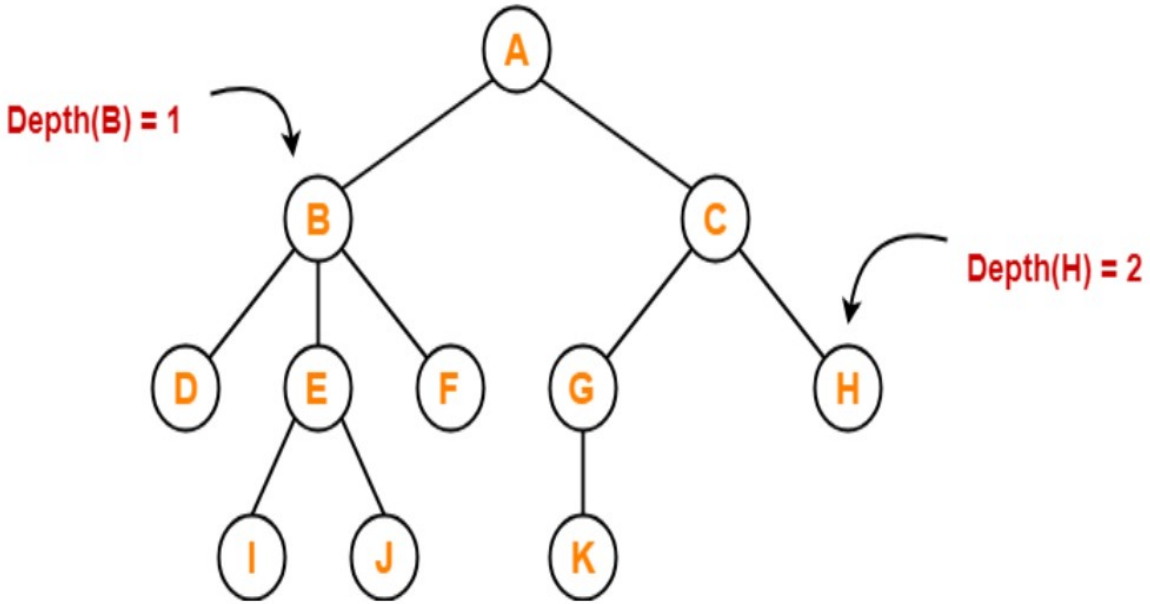
Depth

The no. of edges existing from the root node to a particular node is called as depth of that node.

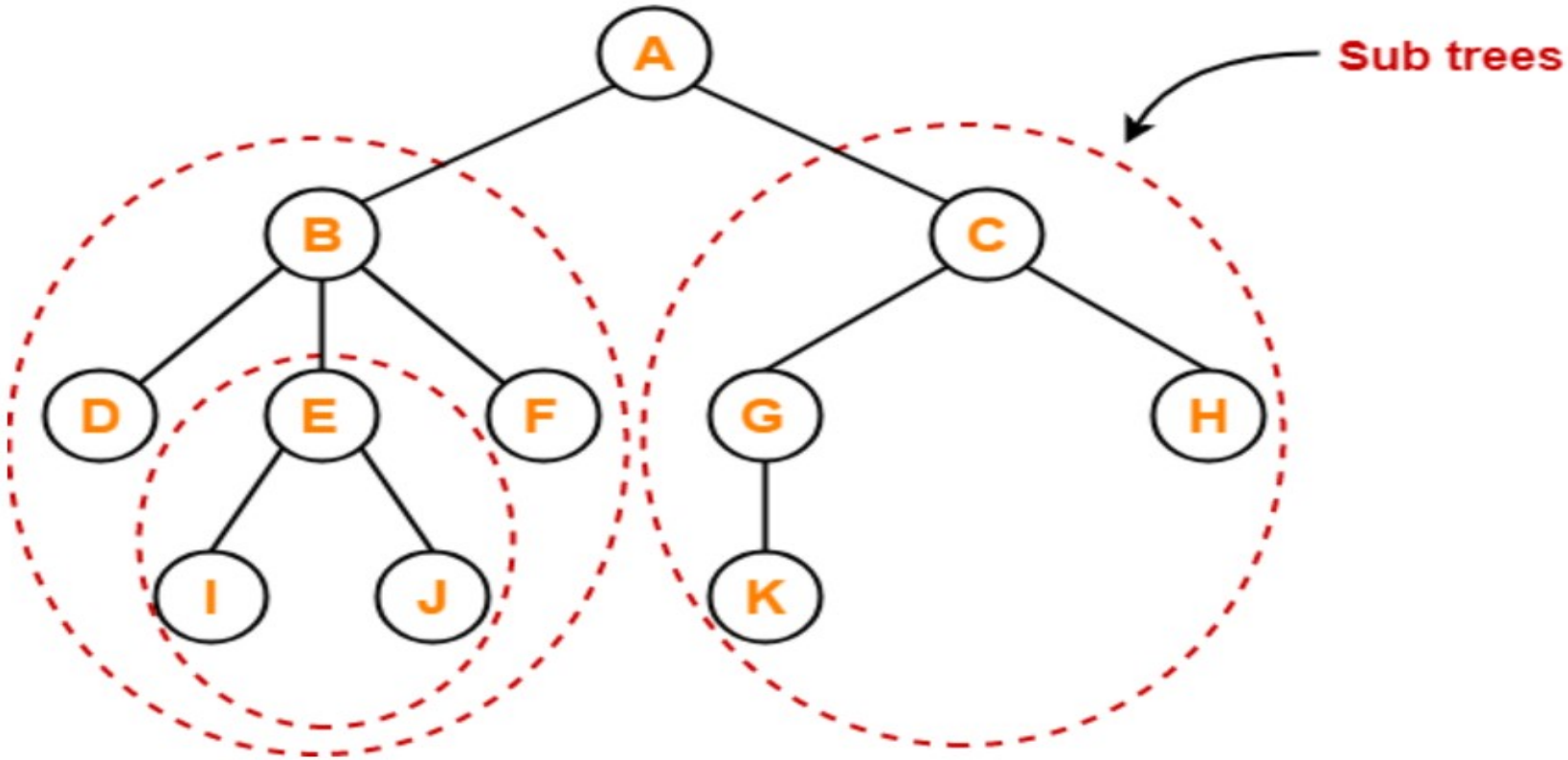
Depth of a tree is defined as the total number of edges existing from root node to a leaf node in the longest path covered.

Depth of the root node = 0

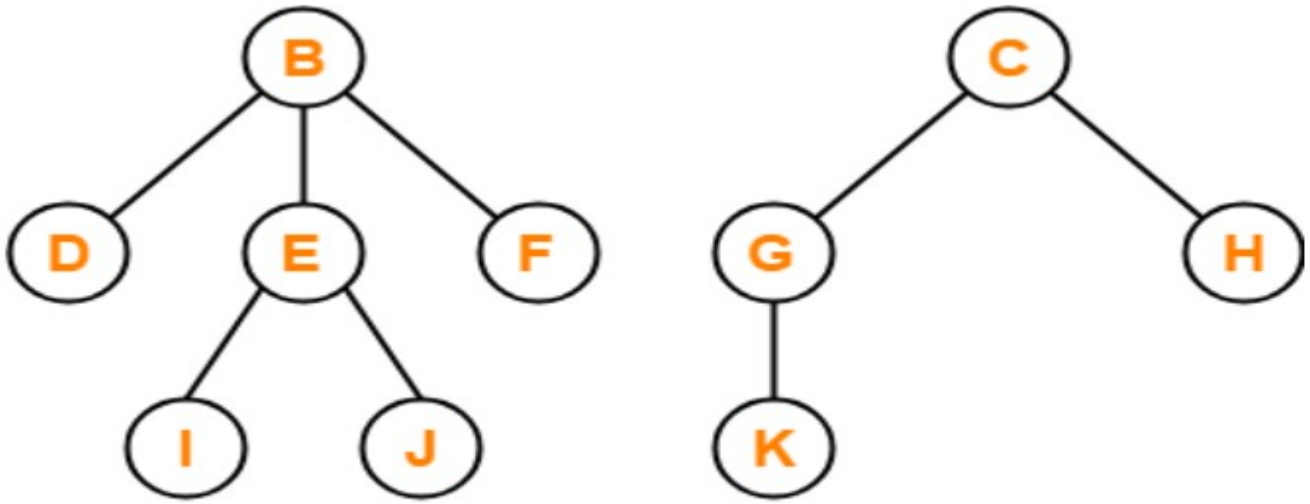
The terms “level” and “depth” are used interchangeably.



Subtrees



Forest (Collection of disjoint trees)



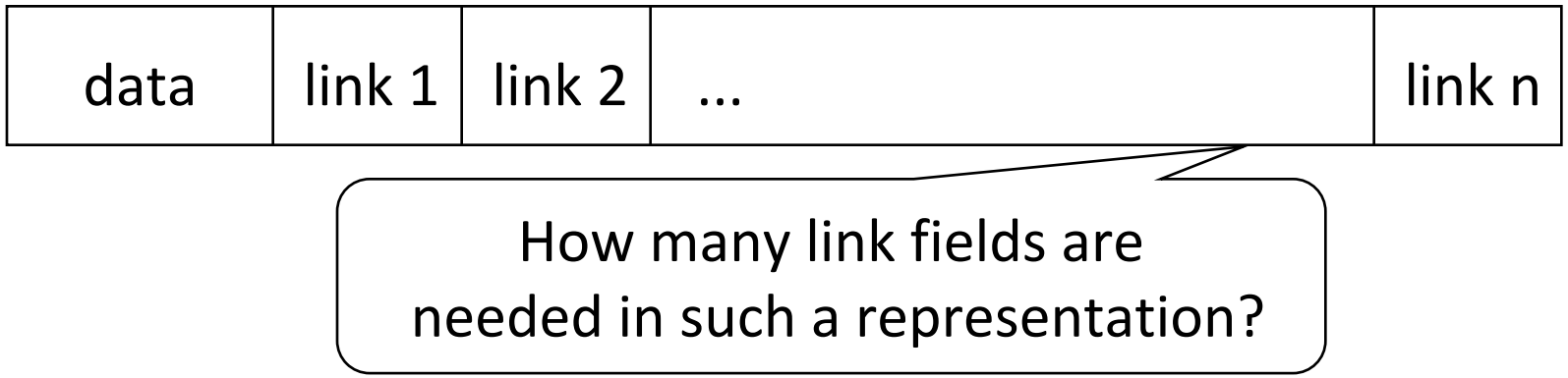
Forest

Representation using Array and Linked List

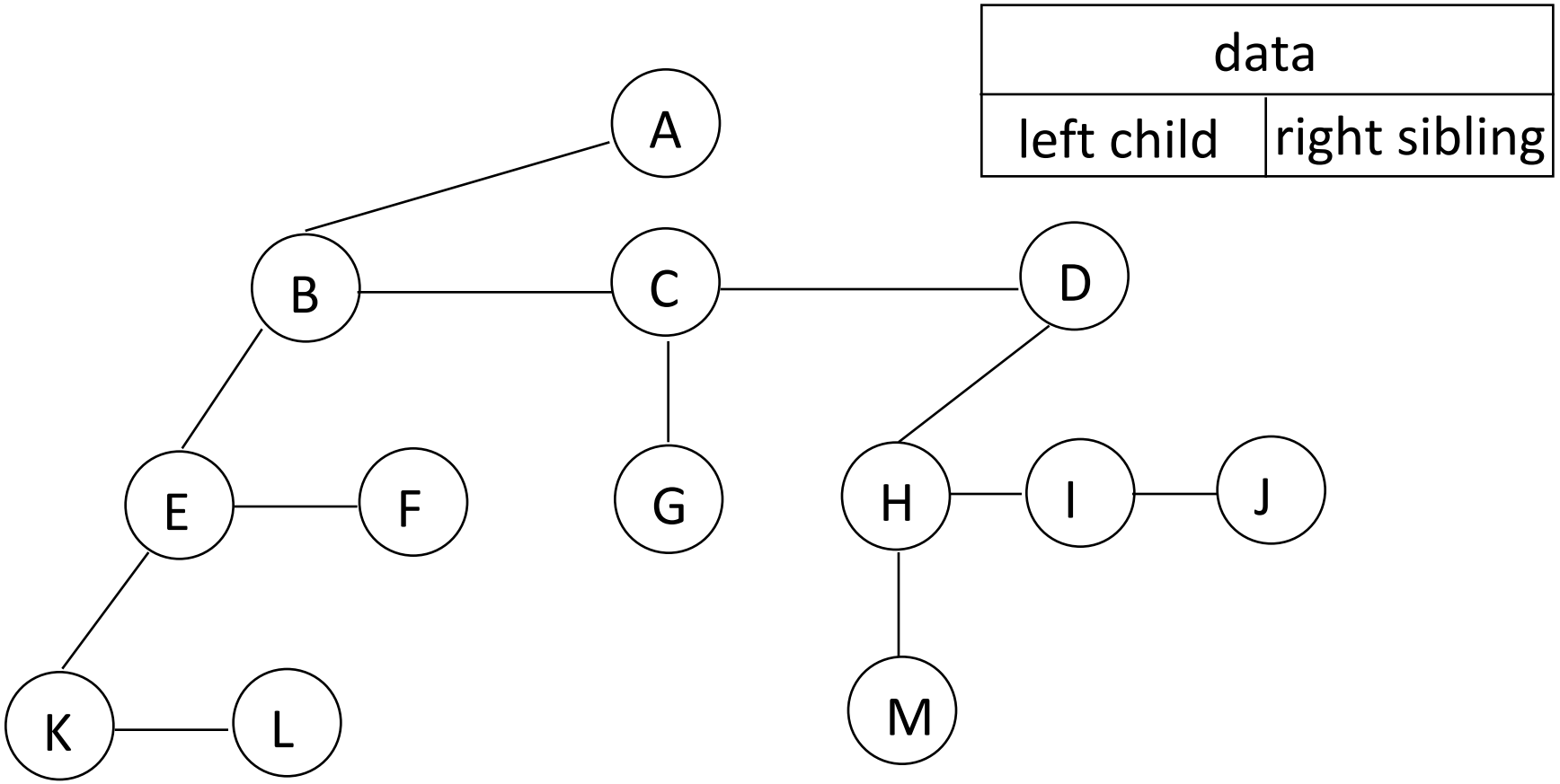
- There must a question arising that if a tree can be represented using array and linked list (both array and linked list are linear DS), then why tree is a non-linear DS ???
- The answer is a DS is called linear not only if its sequential but also if for every node, there exists at least one predecessor or a successor or both but in case of non-linear its not the case.
- Further, trees are non-linear since it has multiple levels whereas linear DS are uni-level.
- Lets see how to represent a tree using array and linked list.

Representation of Trees

- List Representation
  - (A (B (E (K, L), F), C (G), D (H (M), I, J)))
  - The root comes first, followed by a list of sub-trees



Left Child - Right Sibling



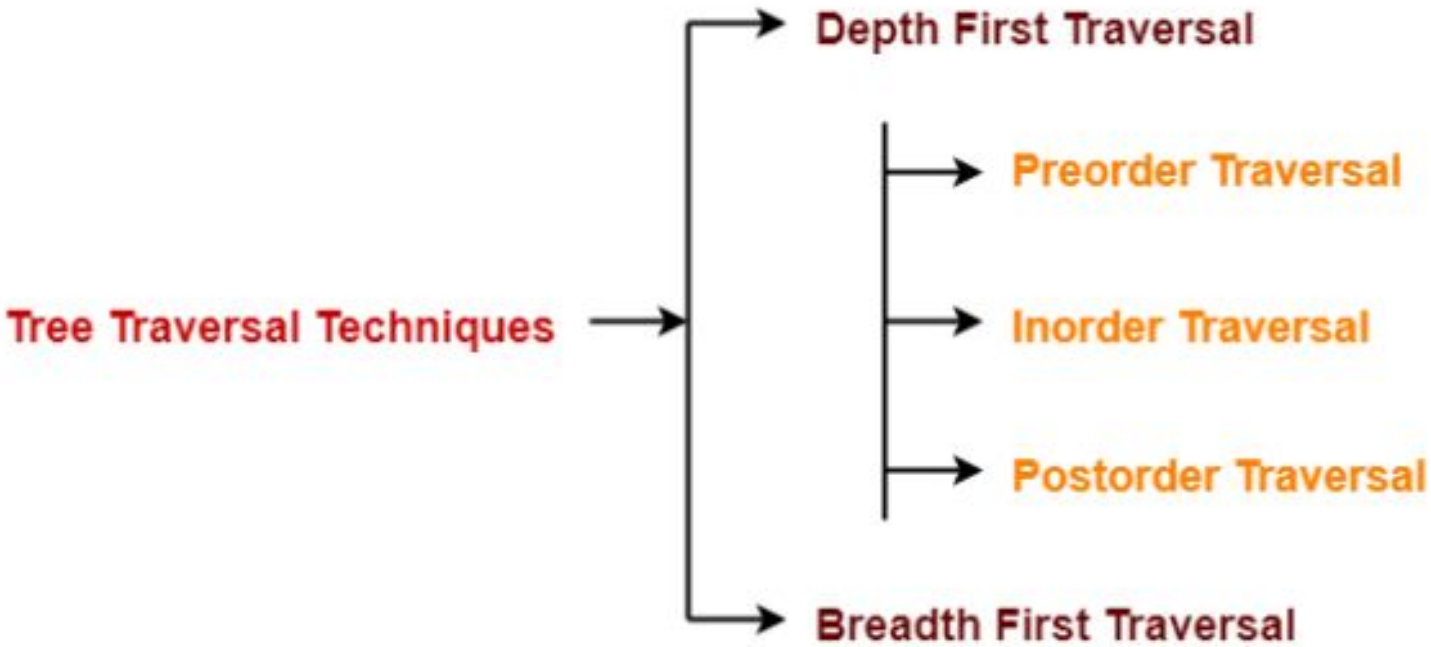
Tree Traversal

- Traversal is the process of visiting every node once
- Visiting a node entails doing some processing at that node, but when describing a traversal strategy, we need not concern ourselves with what that processing is

Tree Traversal Techniques

- Three recursive techniques for tree traversal.
- In each technique, the left subtree is traversed recursively, the right subtree is traversed recursively, and the root is visited.
- What distinguishes the techniques from one another is the order of those 3 tasks.

Tree Traversal



Preoder, Inorder, Postorder

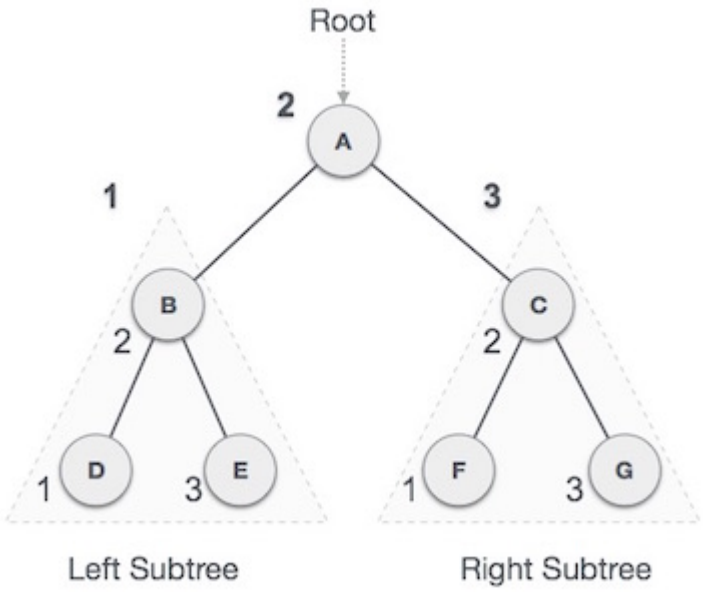
- In Preorder, the root is visited before (pre) the subtrees traversals
- In Inorder, the root is visited in-between left and right subtree traversal
- In Preorder, the root is visited after (pre) the subtrees traversals

Preorder Traversal:
1. Visit the root
2. Traverse left subtree
3. Traverse right subtree

Inorder Traversal:
1. Traverse left subtree
2. Visit the root
3. Traverse right subtree

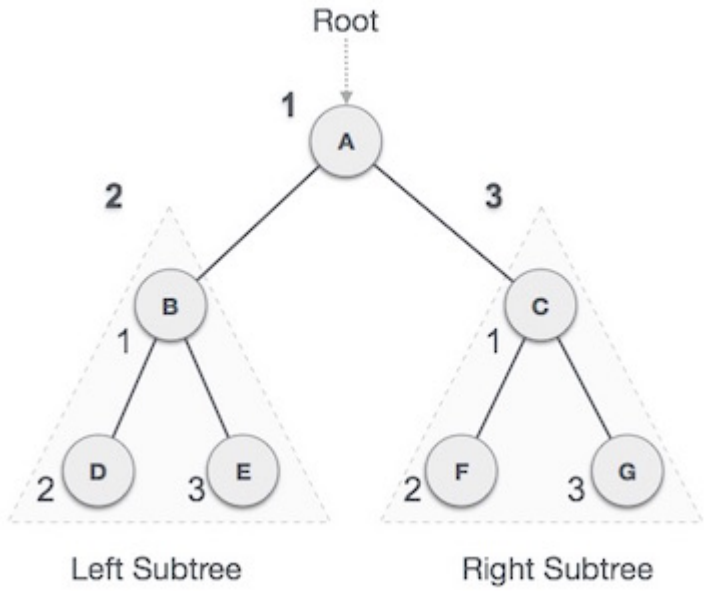
Postorder Traversal:
1. Traverse left subtree
2. Traverse right subtree
3. Visit the root

In Order Traversal



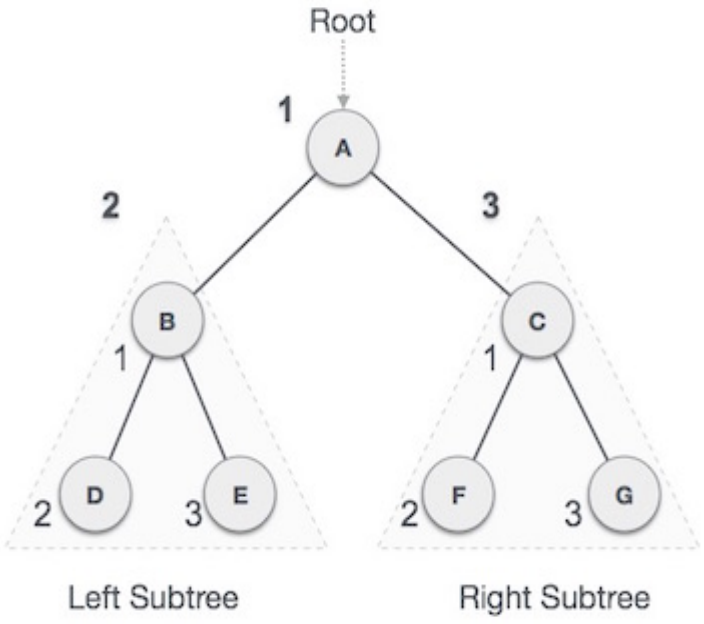
Left → Root → Right

Pre-order Traversal



Root → Left → Right

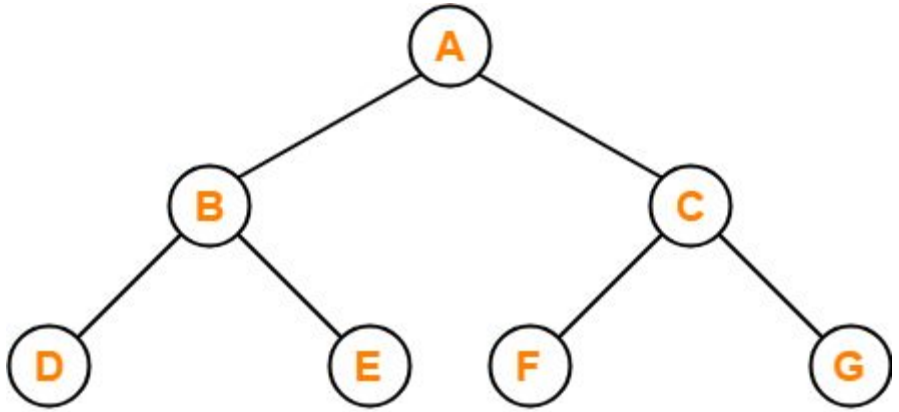
Post-order Traversal



Left → Right → Root

Breadth First Traversal-

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.



Level Order Traversal : A , B , C , D , E , F , G

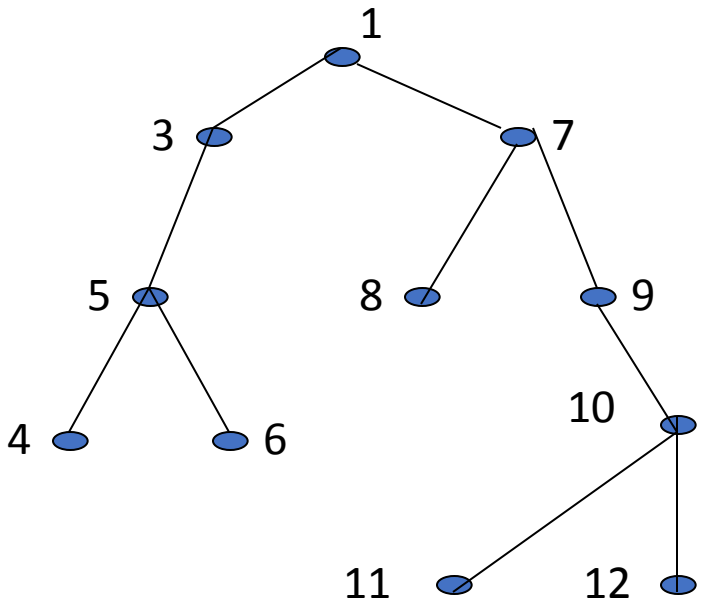
Illustrations for Traversals

- Assume: visiting a node is printing its label
- Preorder:

  1 3 5 4 6 7 8 9 10 11 12
- Inorder:

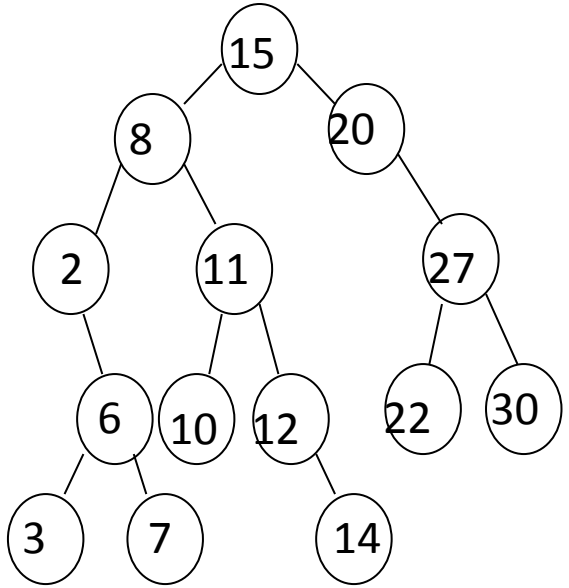
  4 5 6 3 1 8 7 9 11 10 12
- Postorder:

  4 6 5 3 8 11 12 10 9 7 1



Illustrations for Traversals (Contd.)

- Assume: visiting a node is printing its data
- Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30
- Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30
- Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15



Code for the Traversal Techniques

- The code for visit is up to you to provide, depending on the application
- A typical example for visit(…) is to print out the data part of its input node

```
void preOrder(Tree *tree){
    if (tree->isEmpty( ))      return;
    visit(tree->getRoot( ));
    preOrder(tree->getLeftSubtree());

preOrder(tree->getRightSubtree());
}
void inOrder(Tree *tree){
    if (tree->isEmpty( ))      return;
    inOrder(tree->getLeftSubtree( ));
    visit(tree->getRoot( ));
    inOrder(tree->getRightSubtree( ));
}
void postOrder(Tree *tree){
    if (tree->isEmpty( ))      return;
       postOrder(tree->getLeftSubtree(
));
     postOrder(tree->getRightSubtree(
));
    visit(tree->getRoot( ));
}
```

Non-Recursive Tree Traversals: In-order

IN ORDER TRAVERSAL WITHOUT RECURSSION ALGORITHM

(I) Create an empty stack

(II) Initialize current node as root

(III) Do

{

Push the current node in stack;

set current = current->left ;

} until current is NULL

(IV) While (current is NULL and stack is not empty)

{

(a) Pop the top item from stack;

(b) Print the popped item and set current = popped item->right;

}

(V) If (current is NULL and stack is empty)

then end

else

Go to step 3.

Non-Recursive Tree Traversals: Pre-order

Non-Recursive Pre-order Traversal

(I) Create a empty stack

(II) Push root on to the stack

(III) Do

{

(a) Pop node from stack and display it

(b) Push right child of popped node (if present) on to the stack

(c) Push left child of popped node (if present) on to the stack

} Until (Stack is not empty)

Non-Recursive Tree Traversals: Post-order

Non-Recursive Post-order Traversal

(I) Create a two empty stack S1 and S2

(II) Push root on to the stack S1

(III) While (stack S1 in not empty)

{

(a) Pop top node of S1 and push to S2

(b) Push left child of node (if present) on to S1

(c) Push right child of node (if present) on to S1

}

(IV) Pop and Print content of S2

Operations On Binary Tree

- Finding Height of Binary Tree
- Leaf nodes of Binary Tree
- Counting no of nodes in Binary Tree

Operations On Binary Tree

- Finding Height of Binary Tree
- Leaf nodes of Binary Tree
- Counting no of nodes in Binary Tree

Finding Height of Binary Tree

Algorithm 1: Algorithm for calculating the height of a binary tree

Data: *root*: root node of the binary tree
Result: Height of the binary tree
Procedure BTHeight(*root*)

```
if root == NULL then
    return 0;
end
leftTHeight = BTHeight(root → left);
rightTHeight = BTHeight(root → right);
returnMax(leftTHeight, rightTHeight) + 1;
```

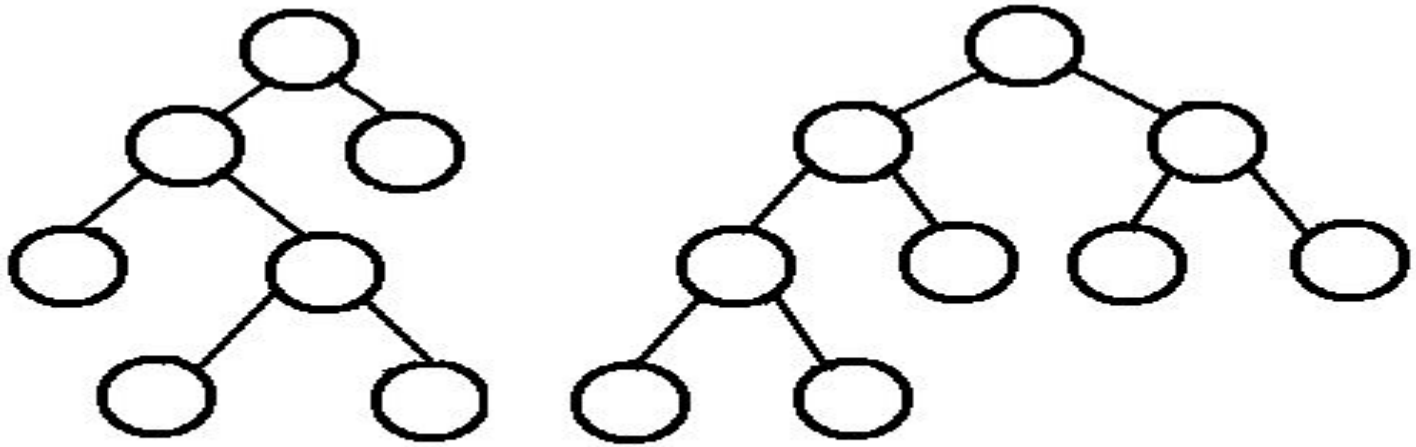
Counting Number of Leaf Nodes

Algorithm to count leaf nodes in a binary tree
Let "root" be the root pointer of a binary tree. If root is NULL, return zero.

- If root is a leaf node, return 1. To determine a leaf node check if both left and right children's are NULL.
- Recursively, calculate the count of leaf nodes in left and right sub tree.
- Return the sum of leaf node count of left and right sub tree.

The following are common types of Binary Trees

- **Full Binary Tree** A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.



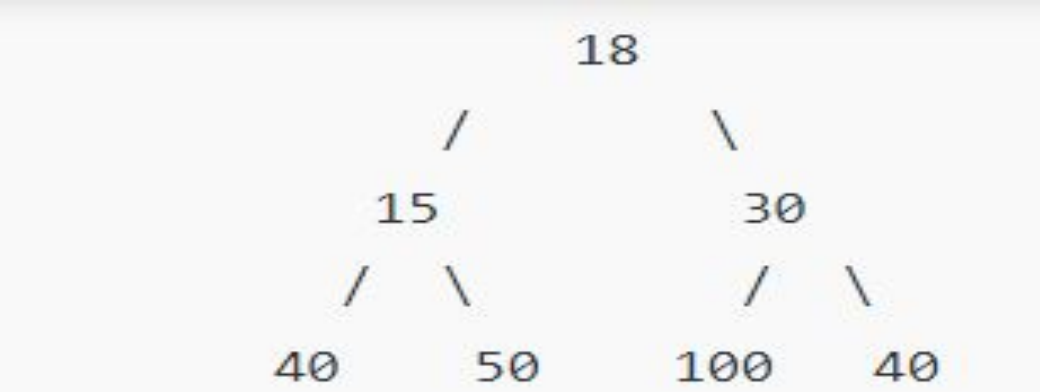
full tree

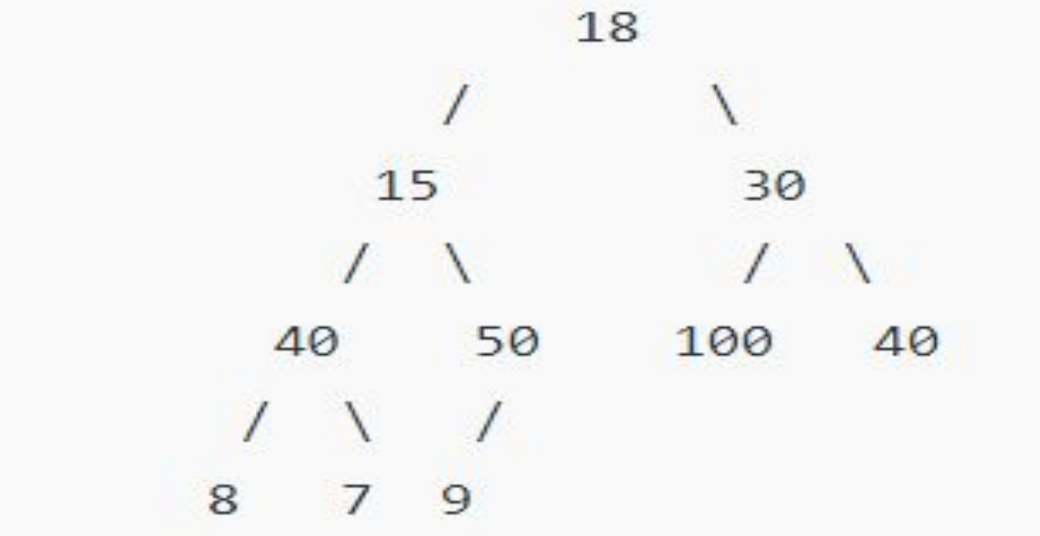
complete tree

Complete Tree

- A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

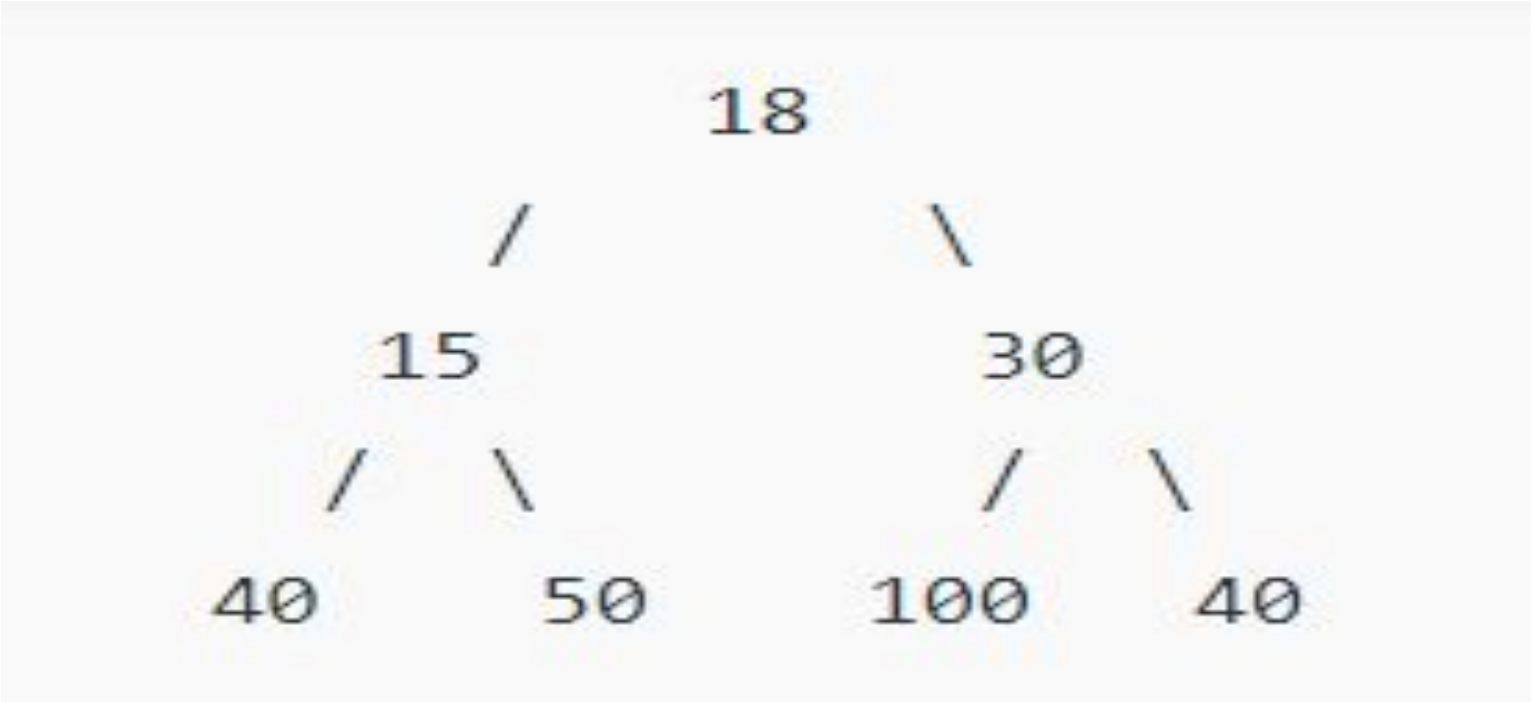
Complete Tree





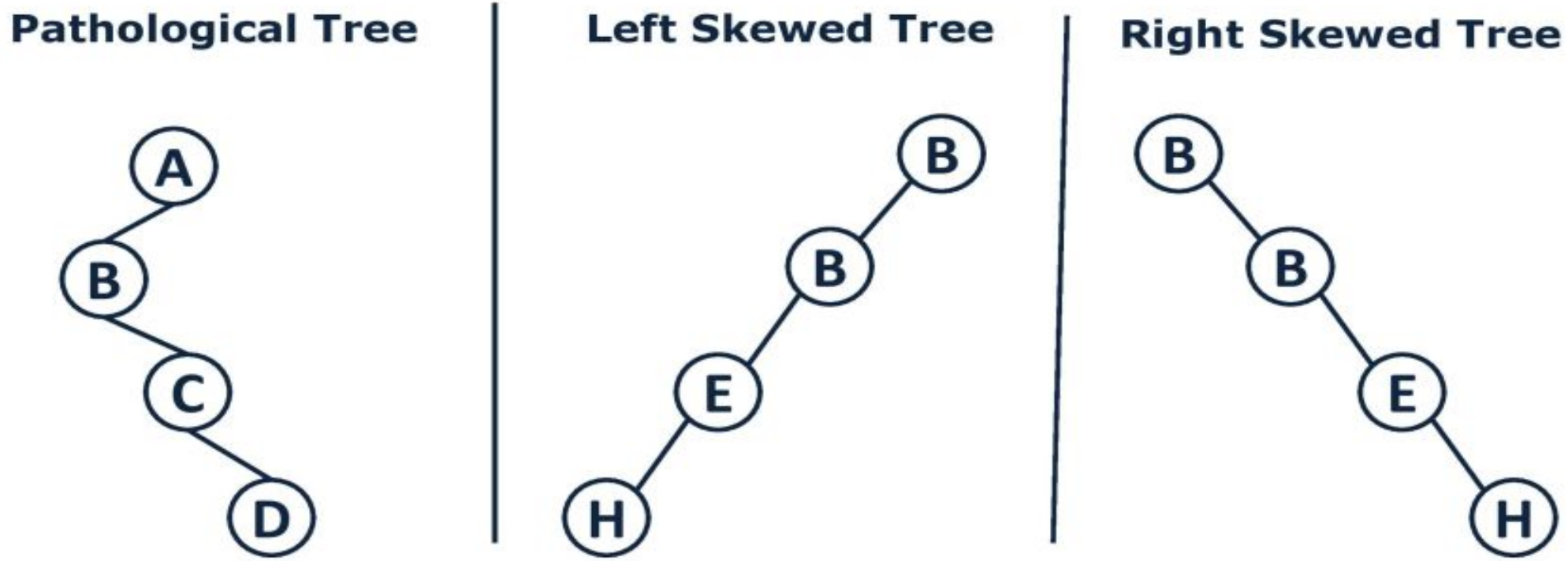
Perfect Binary Tree

- A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.
  The following are the examples of Perfect Binary Trees.



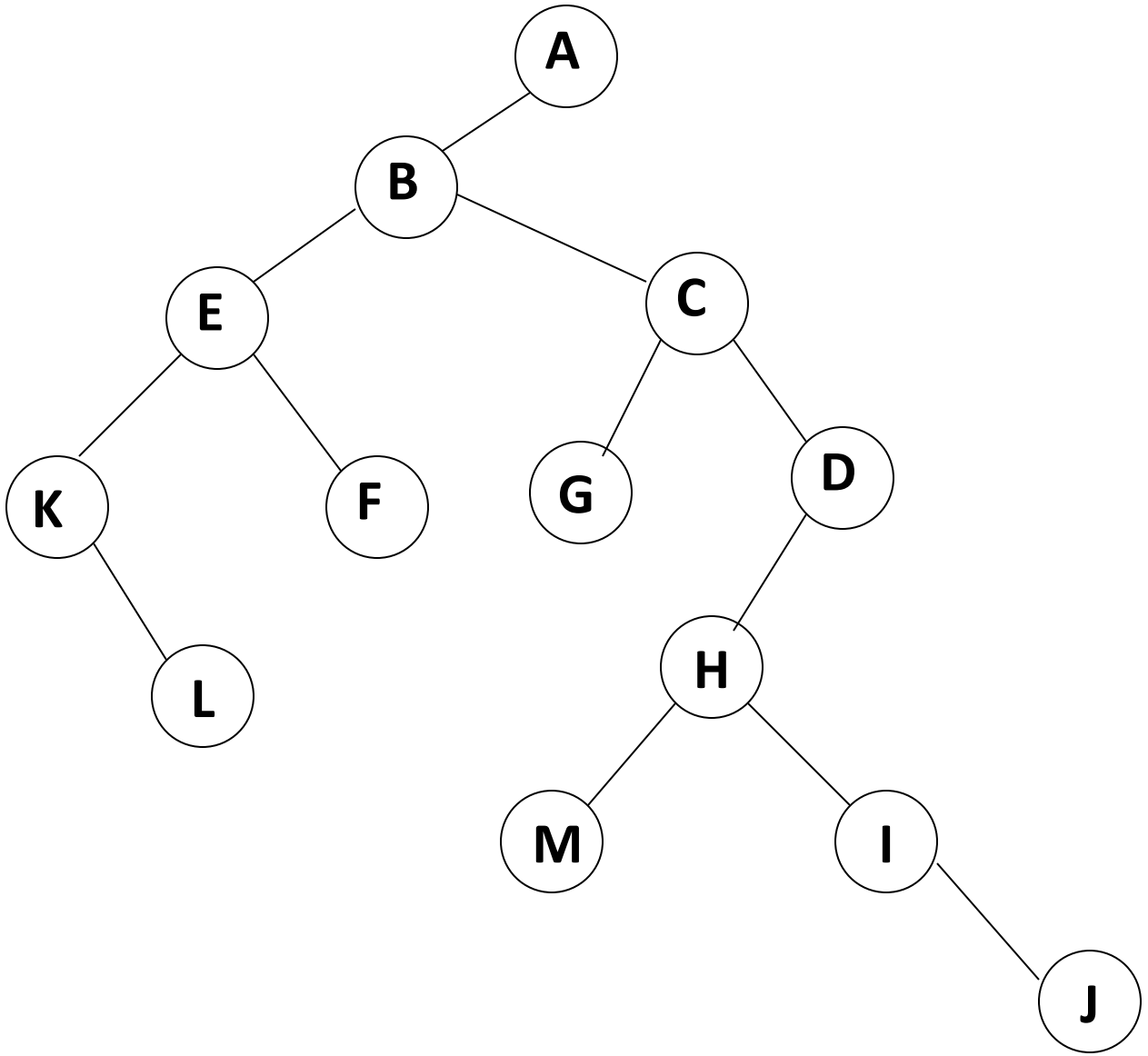
A degenerate (or pathological) tree

- A Tree where every internal node has one child. Such trees are performance-wise same as linked list. Further it is divided into 3 parts. Pathological, left-skewed and right-skewed tree.

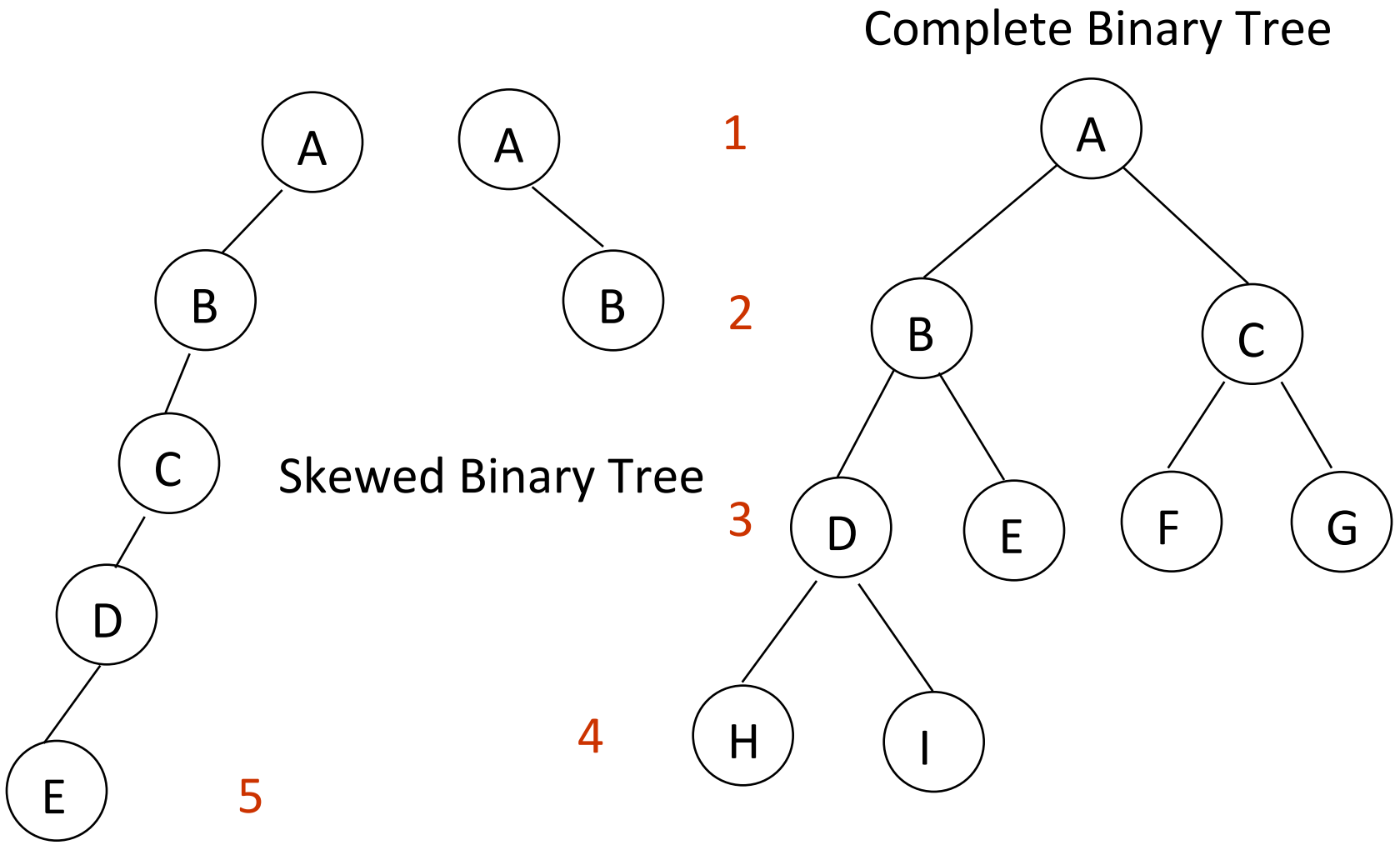


Binary Trees

- A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation
- The left subtree and the right subtree are distinguished.


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

Samples of Trees



Maximum Number of Nodes in BT

- The maximum number of nodes on level i of a binary tree is 2^{i-1}, i>=1.
- The maximum nubmer of nodes in a binary tree of depth k is $2^{k}-1$, k>=1.

i-1

Prove by induction.

$$\sum_{i=1}^{k} 2^{i-1} = 2^{k} - 1$$

Relations between Number of Leaf Nodes and Nodes of Degree 2

For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0=n_2+1$

proof:

Let n and B denote the total number of nodes & branches in T.

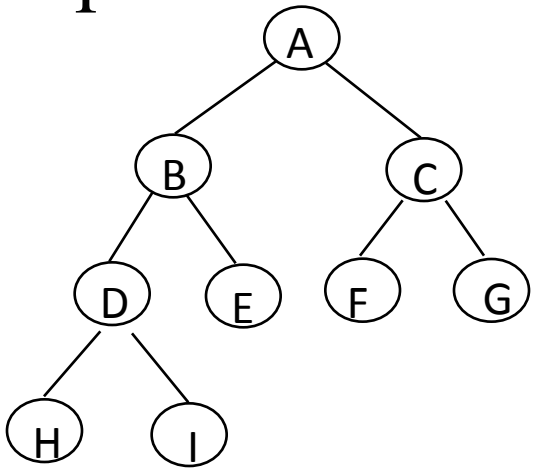
Let n_0, n_1, n_2 represent the nodes with no children, single child, and two children respectively.

$n= n_0+n_1+n_2$, $B+1=n$, $B=n_1+2n_2$ ==> $n_1+2n_2+1= n$,

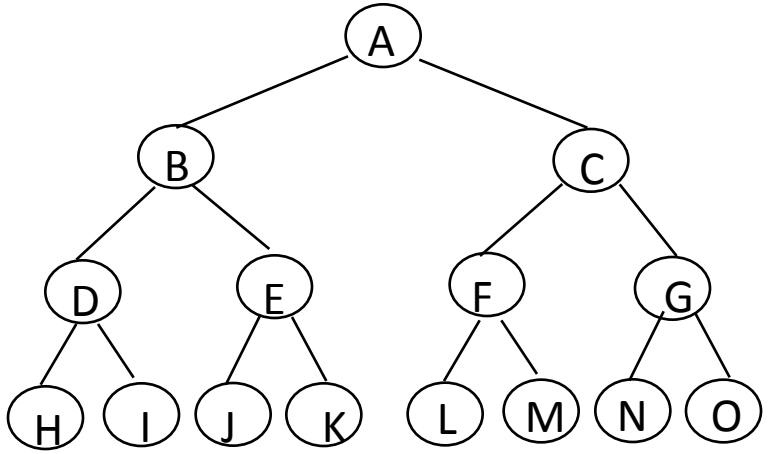
$n_1+2n_2+1= n_0+n_1+n_2$ ==> $n_0=n_2+1$

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

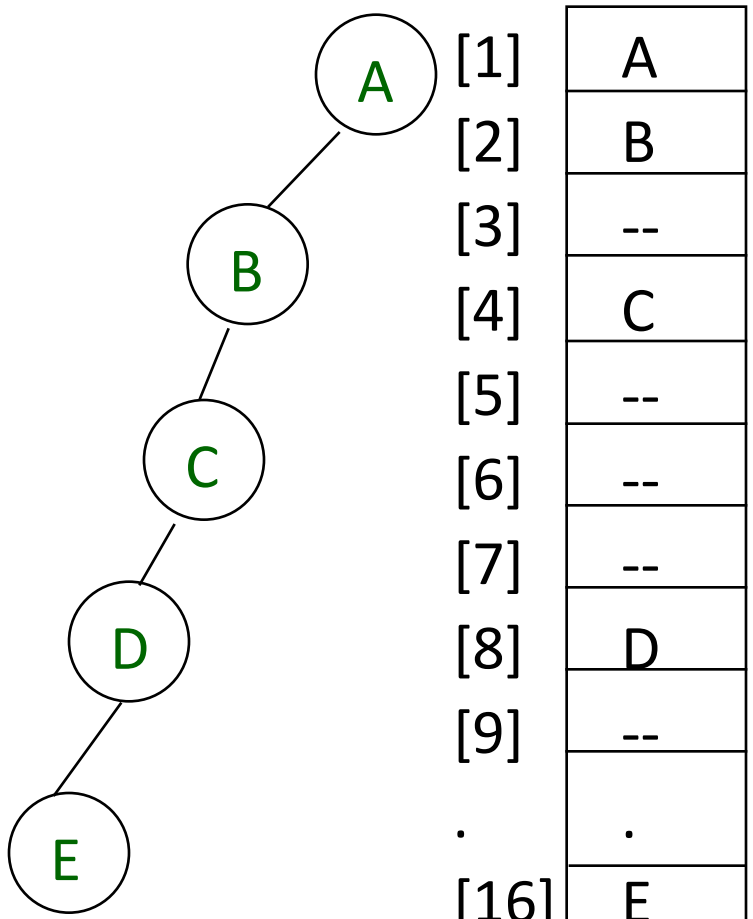


Full binary tree of depth 4

Binary Tree Representations

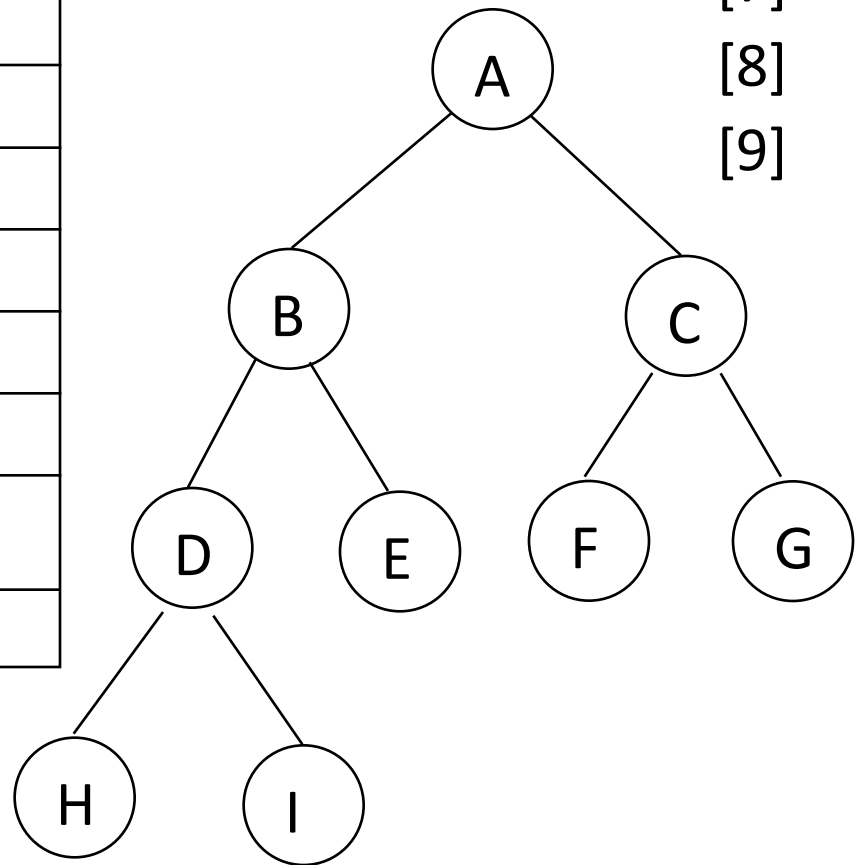
- If a complete binary tree with n nodes (depth = $\log n + 1$) is represented sequentially, then for any node with index i, $1 <= i <= n$, we have:
  - *parent*(i) is at $i/2$ if i!=1. If i=1, i is at the root and has no parent.
  - *left_child*(i) ia at $2i$ if $2i <= n$. If $2i$>n, then i has no left child.
  - *right_child*(i) ia at $2i+1$ if $2i + 1 <= n$. If $2i + 1 >$n, then i has no right child.

Sequential Representation



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

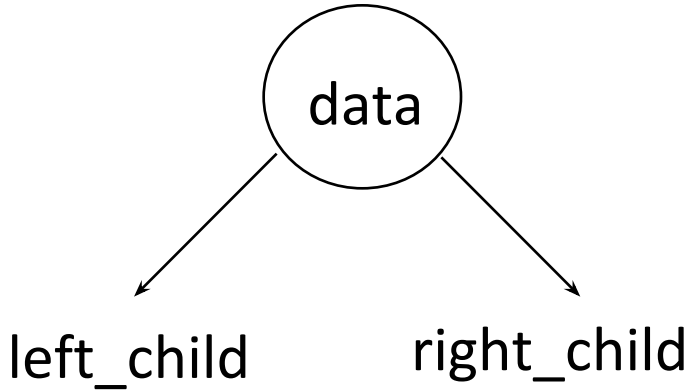
(1) waste space
(2) insertion/deletion problem



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

Linked Representation

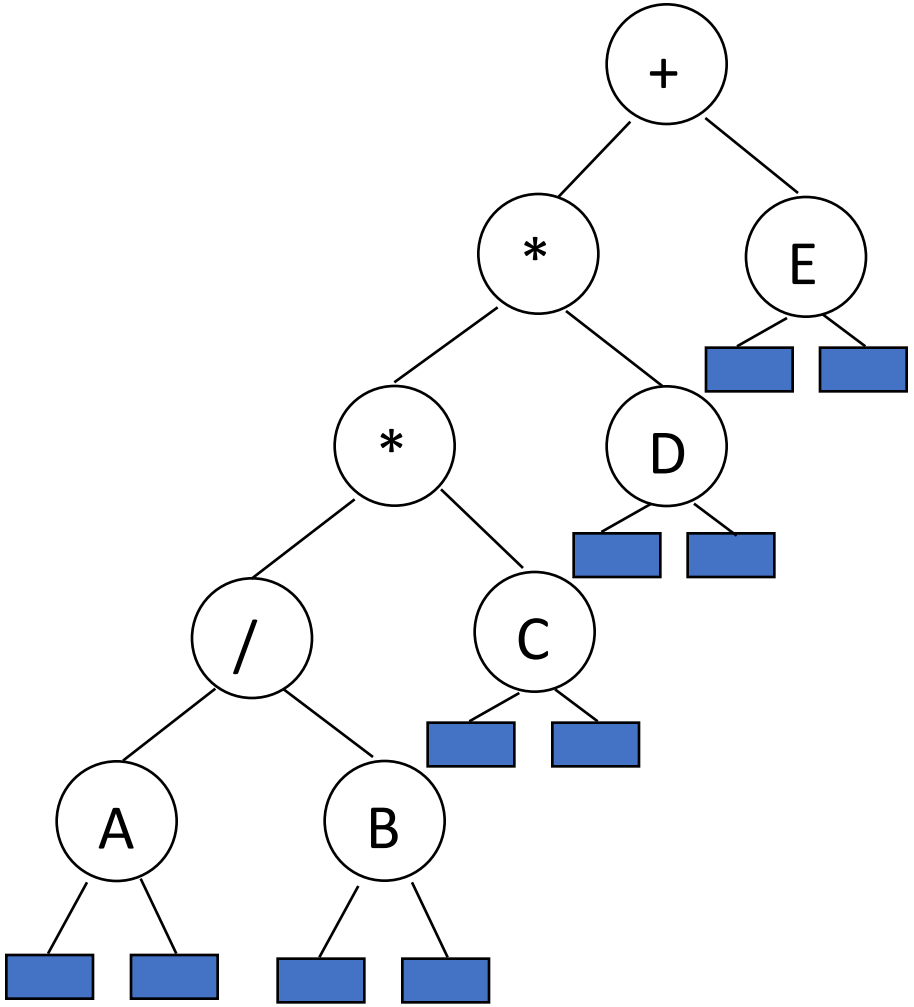
```
typedef struct node *tree_pointer;
typedef struct node {
 int data;
 tree_pointer left_child, right_child;
};
```



Binary Tree Traversals

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

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

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 +

Iterative Inorder Traversal

(using stack)

```
void iter_inorder(tree_pointer node)
{
  int top= -1; /* initialize stack */
  tree_pointer stack[MAX_STACK_SIZE];
  for (;;) {
   for (; node; node=node->left_child)
     add(&top, node);/* add to stack */
   node= delete(&top);
                    /* delete from stack */
   if (!node) break; /* empty stack */
   printf("%D", node->data);
   node = node->right_child;
  }
}
```

O(n)

Level Order Traversal

(using queue)

```
void level_order(tree_pointer ptr)
/* level order tree traversal */
{
   int front = rear = 0;
   tree_pointer queue[MAX_QUEUE_SIZE];
   if (!ptr) return; /* empty queue */
   addq(front, &rear, ptr);
   for (;;) {
      ptr = deleteq(&front, rear);
```

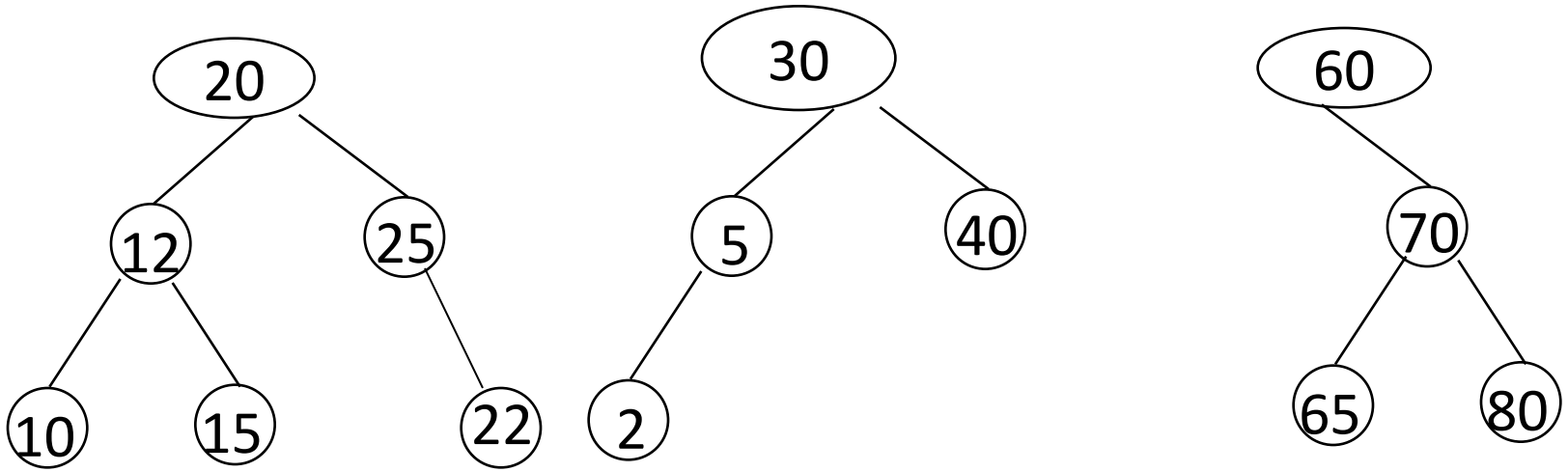
```
      if (ptr) {
        printf("%d", ptr->data);
        if (ptr->left_child)
          addq(front, &rear,
                       ptr->left_child);
        if (ptr->right_child)
          addq(front, &rear,
                       ptr->right_child);
      }
      else break;
    }
}
```

+ * E * D / C A B

Binary Search Tree

- Heap
  - a min (max) element is deleted. $O(\log_2 n)$
  - deletion of an arbitrary element $O(n)$
  - search for an arbitrary element $O(n)$
- Binary search tree
  - Every element has a unique key.
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
  - The left and right subtrees are also binary search trees.

Examples of Binary Search Trees



Searching a Binary Search Tree

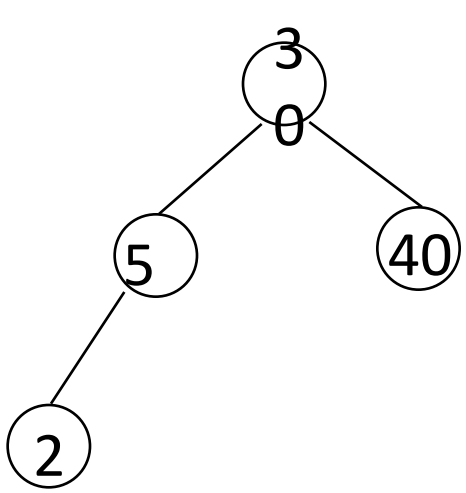
```
tree_pointer search(tree_pointer root,
                    int key)
{
/* return a pointer to the node that
   contains key. If there is no such
   node, return NULL */

   if (!root) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
        return search(root->left_child,
                      key);
   return search(root->right_child,key);
}
```

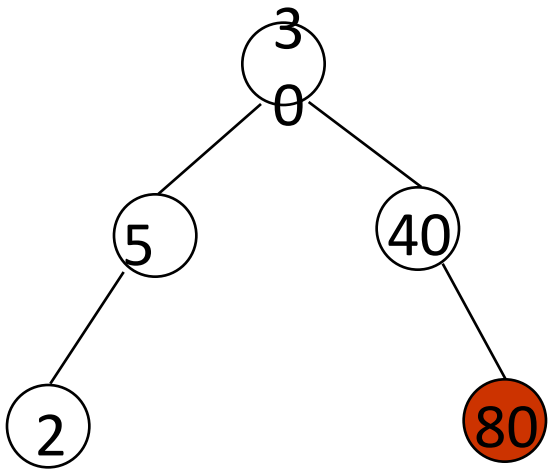
Another Searching Algorithm

```
tree_pointer search2(tree_pointer tree,
  int key)
{
  while (tree) {
    if (key == tree->data) return tree;
    if (key < tree->data)
        tree = tree->left_child;
    else tree = tree->right_child;
  }
  return NULL;
}
```

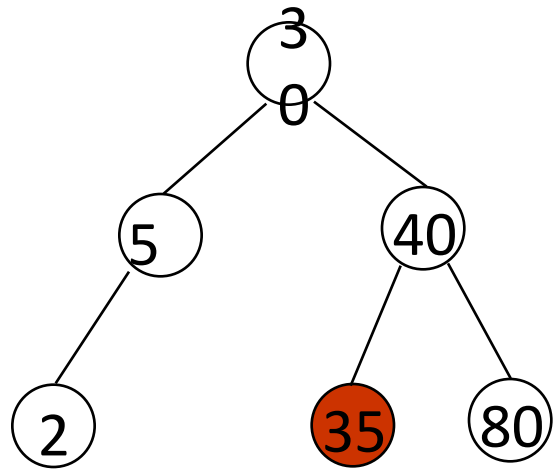
O(h)

Insert Node in Binary Search Tree





Insert 80



Insert 35

Insert Node in Binary Search Tree

Algorithm

1. Create a new BST node and assign values to it.

2. insert(node, key)
   i) If root == NULL,
      return the new node to the calling function.
   ii) if root=>data < key
      call the insert function with root=>right and assign the return value in root=>right.
      root->right = insert(root=>right,key)
   iii) if root=>data > key
      call the insert function with root->left and assign the return value in root=>left.
      root=>left = insert(root=>left,key)

3. Finally, return the original root pointer to the calling function.

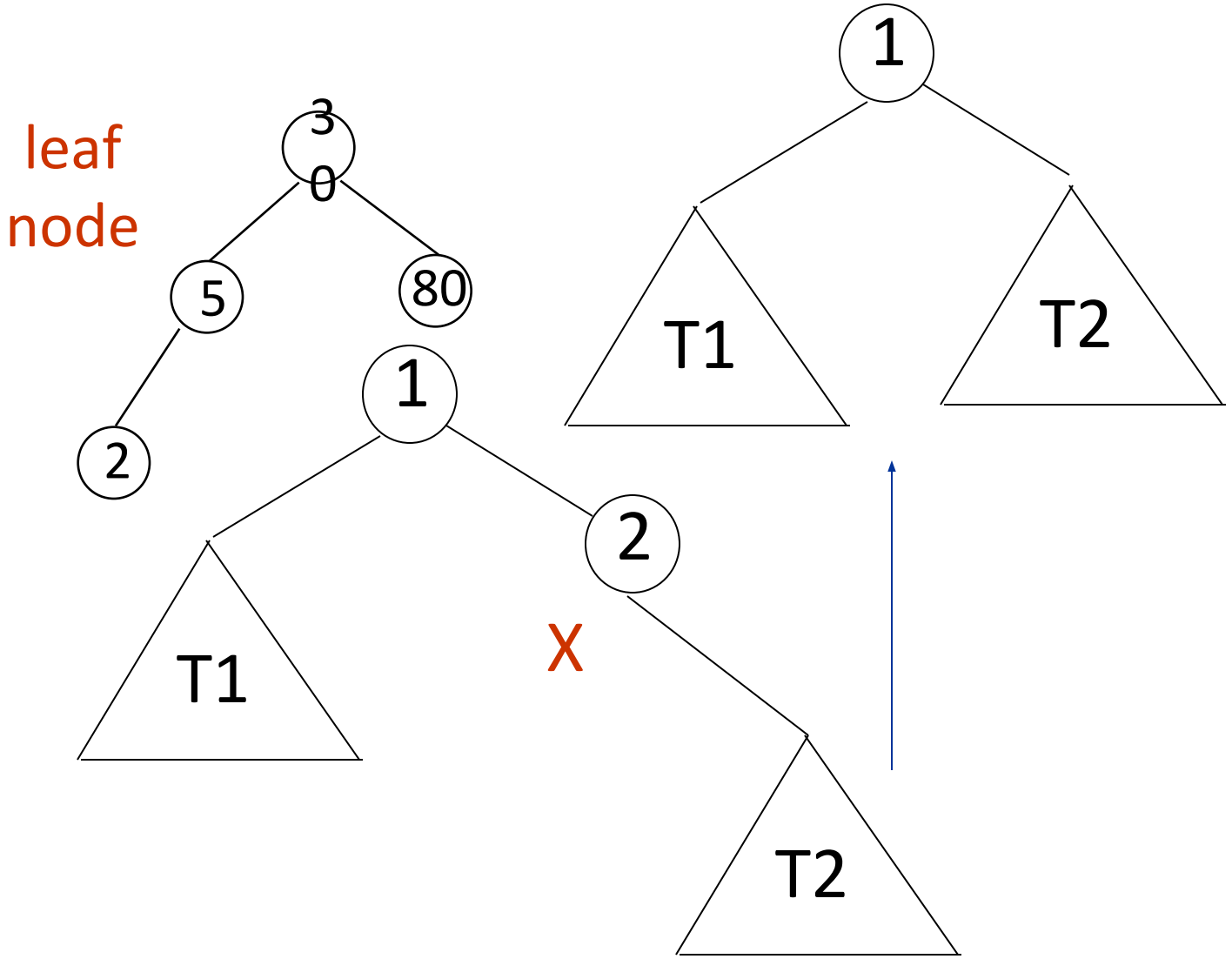
Algorithm

1. Create a new BST node and assign values to it.
2. insert(node, key)
    i) If root == NULL,
        return the new node to the calling function.
    ii) if root=>data < key
        call the insert function with root=>right and assign the return value in root=>right.
        root->right = insert(root=>right,key)
    iii) if root=>data > key
        call the insert function with root->left and assign the return value in root=>left.
        root=>left = insert(root=>left,key)
3. Finally, return the original root pointer to the calling function.

Insertion into A Binary Search Tree

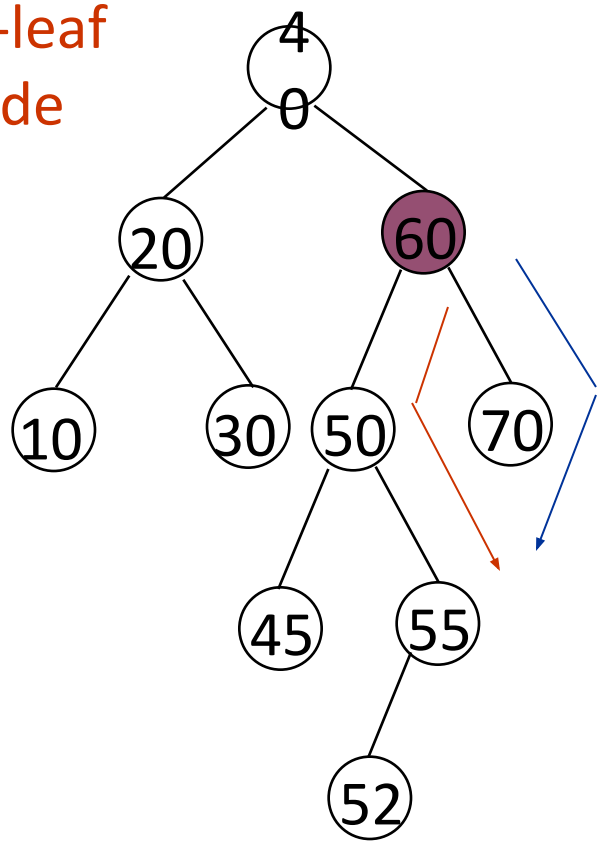
```
void insert_node(tree_pointer *node, int num)
{tree_pointer ptr,
      temp = modified_search(*node, num);
  if (temp || !(*node)) {
   ptr = (tree_pointer) malloc(sizeof(node));
   if (IS_FULL(ptr)) {
     fprintf(stderr, "The memory is full\n");
     exit(1);
   }
   ptr->data = num;
   ptr->left_child = ptr->right_child = NULL;
   if (*node)
     if (num<temp->data) temp->left_child=ptr;
        else temp->right_child = ptr;
   else *node = ptr;
  }
}
```

Deletion for A Binary Search Tree

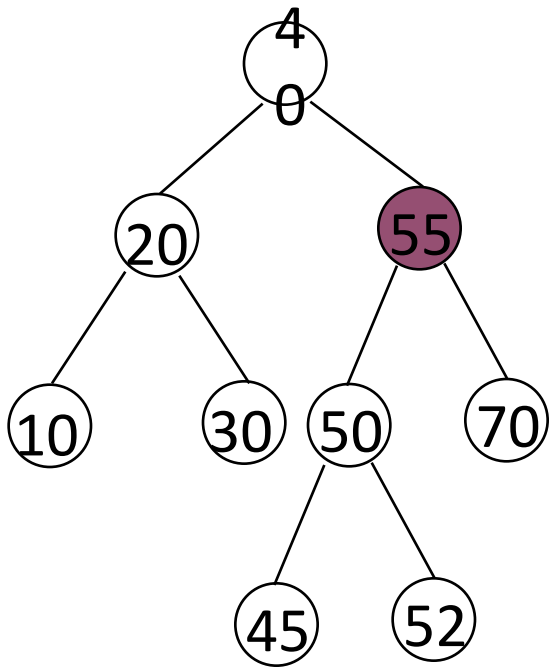


Deletion for A Binary Search Tree

non-leaf node



Before deleting 60



After deleting 60

Deletion for A Binary Search Tree

Algorithm

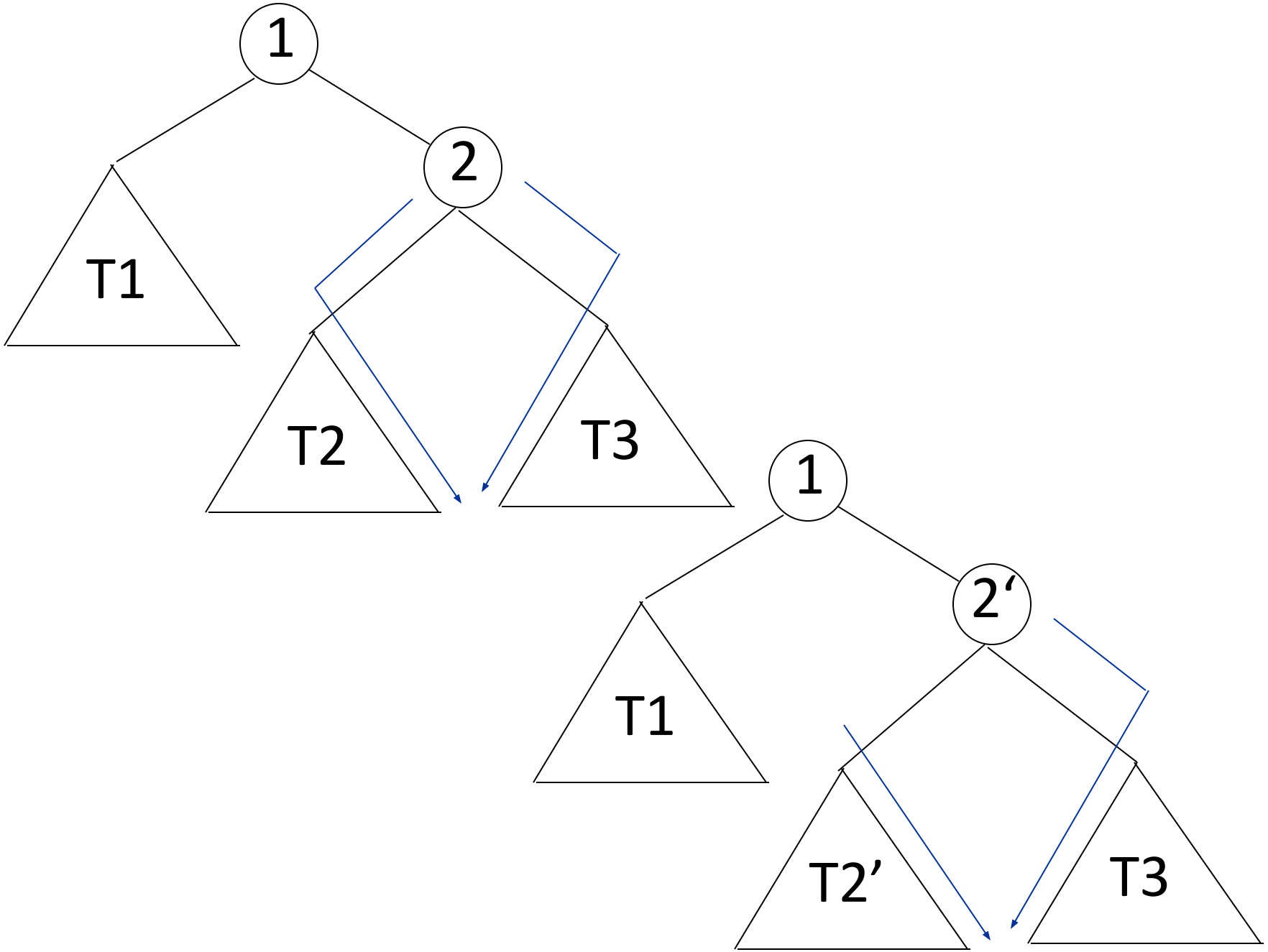
•Compare the value of **key** with the value of root. If the key > root -> value, recursively traverse the right sub-tree.

•If key < root -> value, recursively traverse the left sub-tree.

•While traversing if key == root->value, we need to delete this node:

- If the node is a leaf, make **root = NULL.**
- If the node is not a leaf and has the right child, recursively replace its value with the successor node and delete its successor from its original position.
- If the node is not a leaf and has left child, but not right child, recursively replace its value by predecessor node and delete its predecessor from its original position.

•Return root


1
2
T1
T2
T3
1
2’
T1
T2’
T3

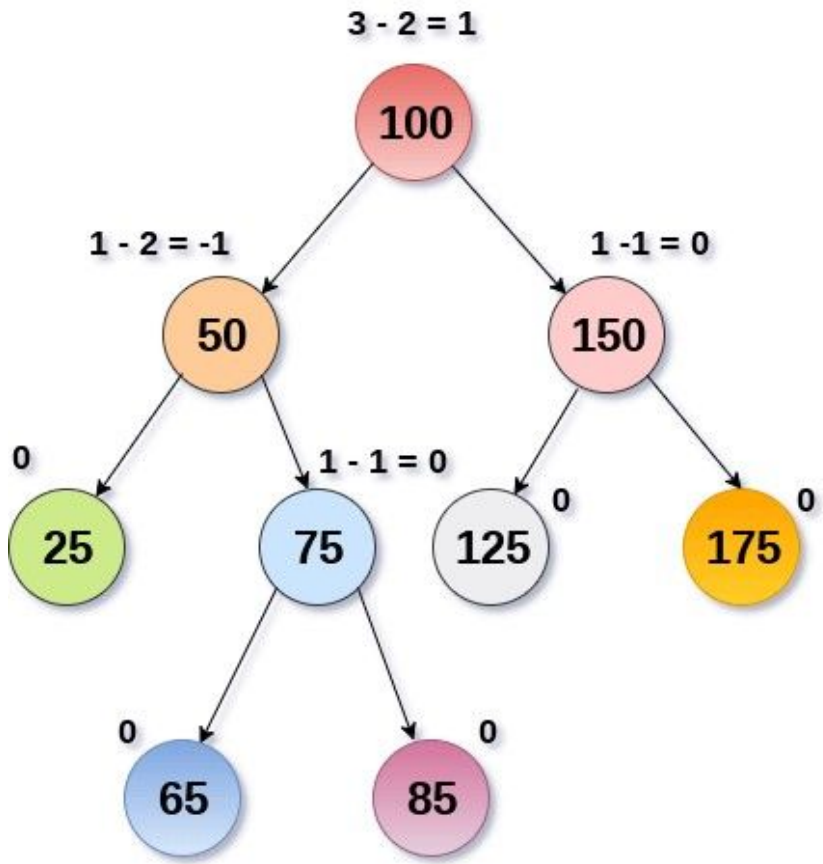
AVL tree

- AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Or

- AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right sub-trees cannot be more than one for all nodes.

AVL Tree



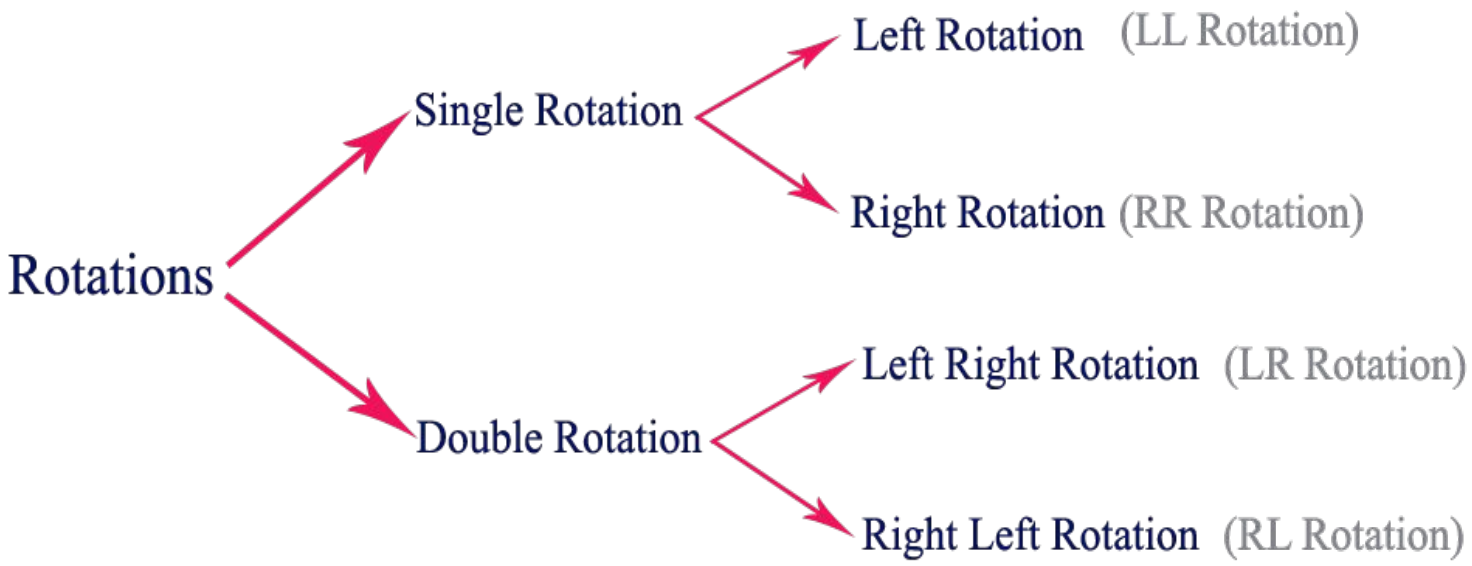
Balance Factor

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.
- An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.

Why AVL Trees?

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

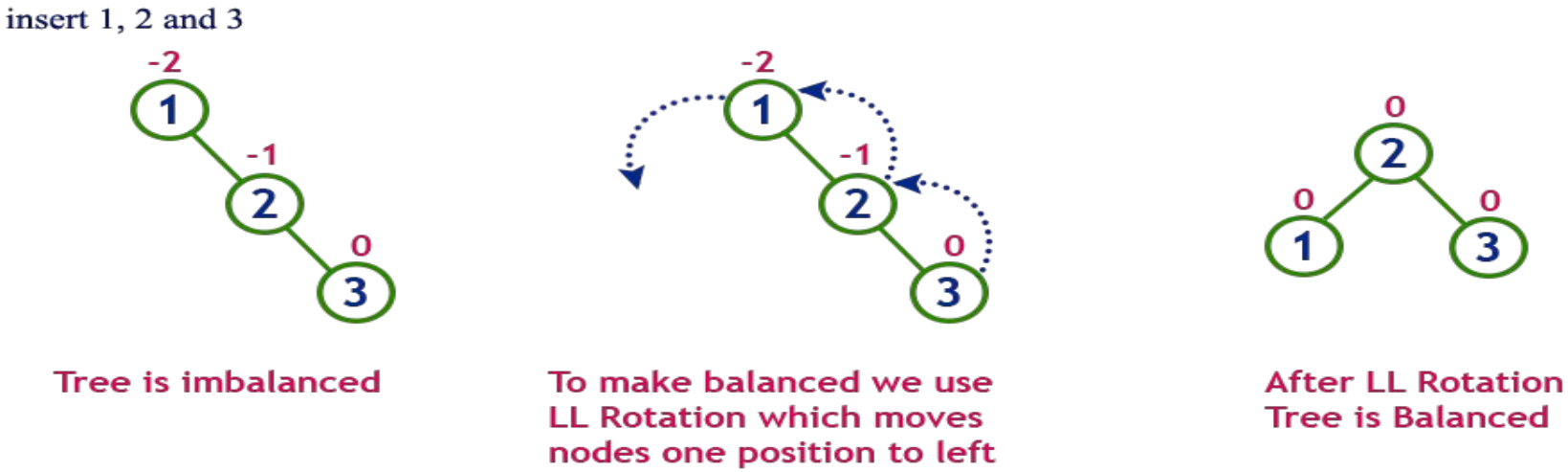
Tree Rotations



Tree Rotations

Single Left Rotation (LL 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…

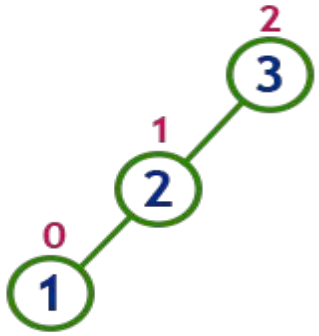


Tree Rotations

Single Right Rotation (RR Rotation)

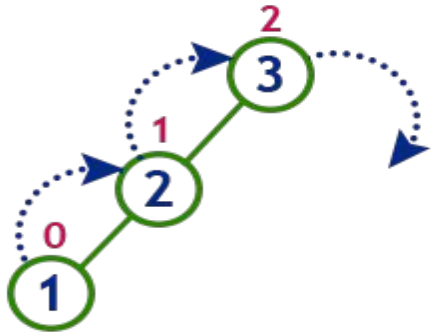
In RR Rotation, every node moves one position to right from the current position. To understand RR Rotation, let us consider the following insertion operation in AVL Tree...

insert 3, 2 and 1

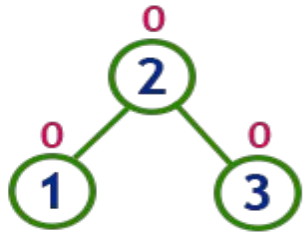


Tree is imbalanced

because node 3 has balance factor 2



To make balanced we use RR Rotation which moves nodes one position to right

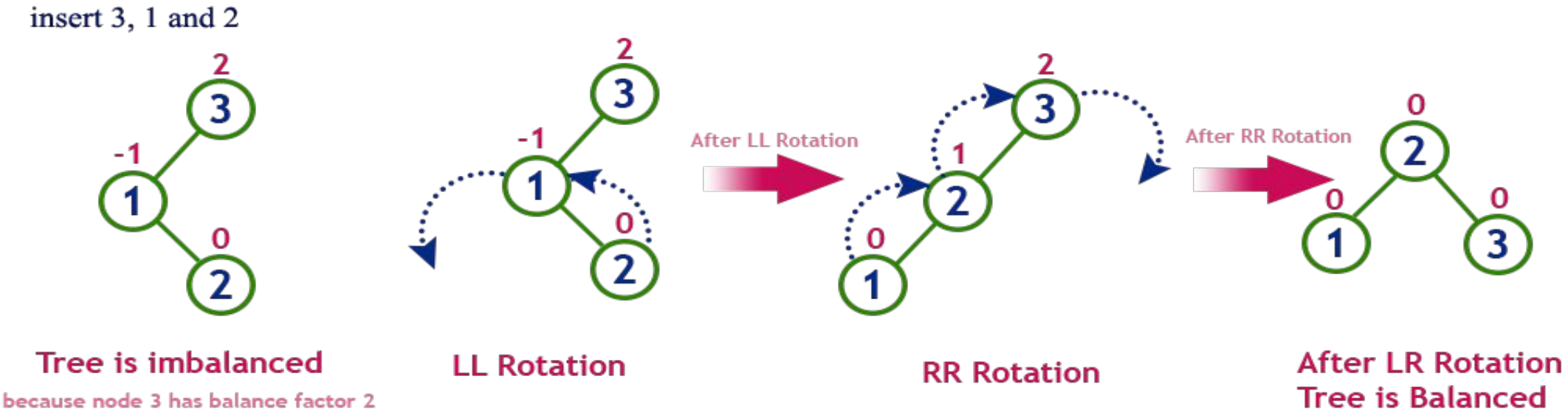


After RR Rotation Tree is Balanced

Tree Rotations

Left Right Rotation (LR Rotation)

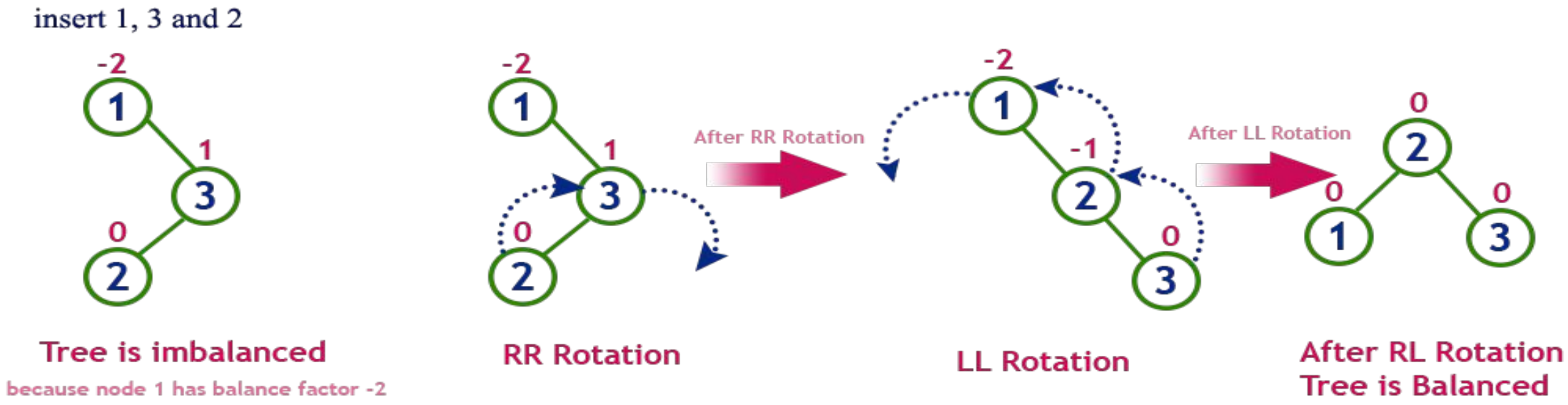
The LR Rotation is a sequence of single left rotation followed by a single right rotation. In LR Rotation, at first, every node moves one position to the left and one position to right from the current position. To understand LR Rotation, let us consider the following insertion operation in AVL Tree...



Tree Rotations

Right Left Rotation (RL Rotation)

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...



Operations on an AVL Tree

The following operations are performed on AVL tree...

- **Search**
- **Insertion**
- **Deletion**

AVL Tree Rebalancing

- We only discuss inserts; deletes similar though a bit more messy.
- How can an insert violate AVL condition?
- Before the insert, at some node x the height difference between children was 1, but becomes 2 after the insert.
- We will fix this violation from bottom (leaf) up.

AVL Tree Rebalancing

- Suppose A is the first node from bottom that needs rebalancing.
- Then, A must have two subtrees whose heights differ by 2.
- This imbalance must have been caused by one of two cases:

(1) The new insertion occurred into
the left subtree of the left child of A; or
the right subtree of the right child of A.

(2) The new insertion occurred into
the right subtree of the left child of A; or
the left subtree of the right child of A.

(1) and (2) are different: former is an outside (left-left, or right-right) violation, while the latter is an inside (left-right) or (right-left) violation.

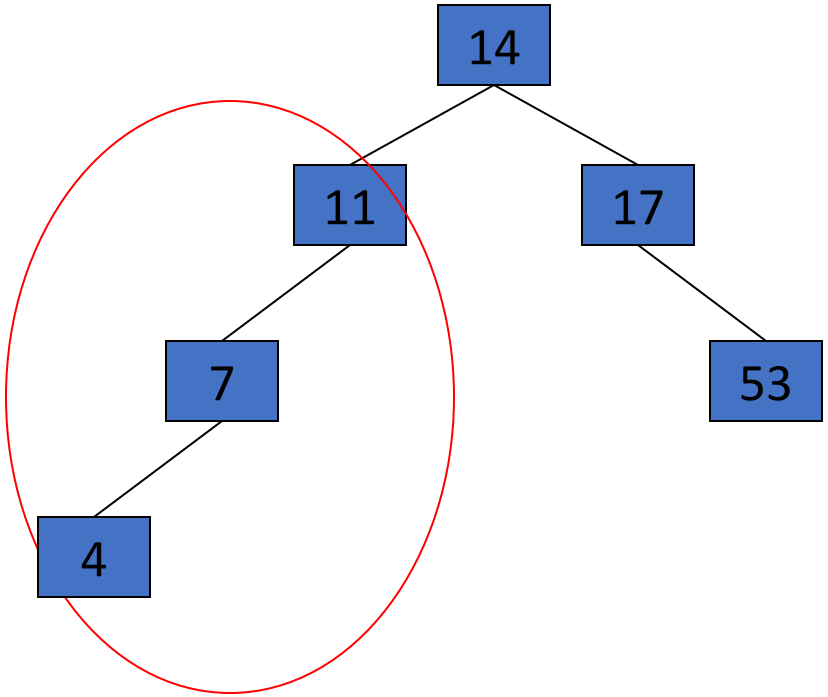
We fix (1) with a SINGLE ROTATION; and (2) with a DOUBLE ROTATION.

AVL Tree Complexity Bounds

- An AVL Tree can perform the following operations in worst-case time O(log n) each:
  - Insert
  - Delete
  - Find
  - Find Min, Find Max
  - Find Successor, Find Predecessor
  - It can report all keys in range [Low, High] in time O(log n + OutputSize)

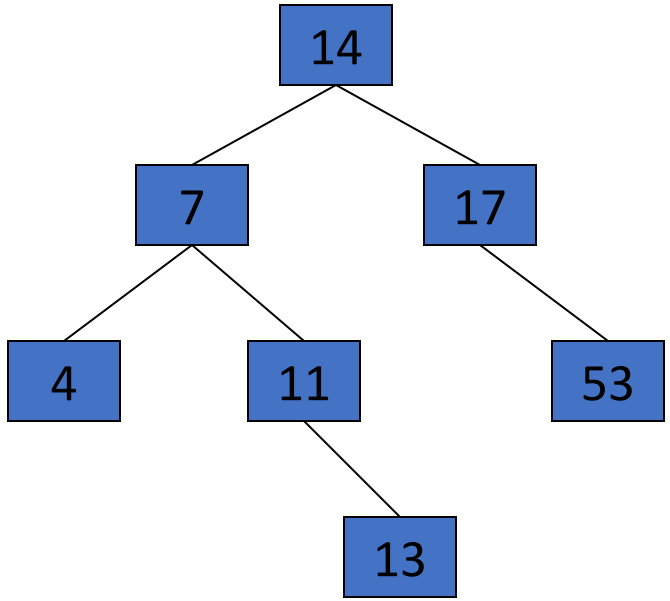
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



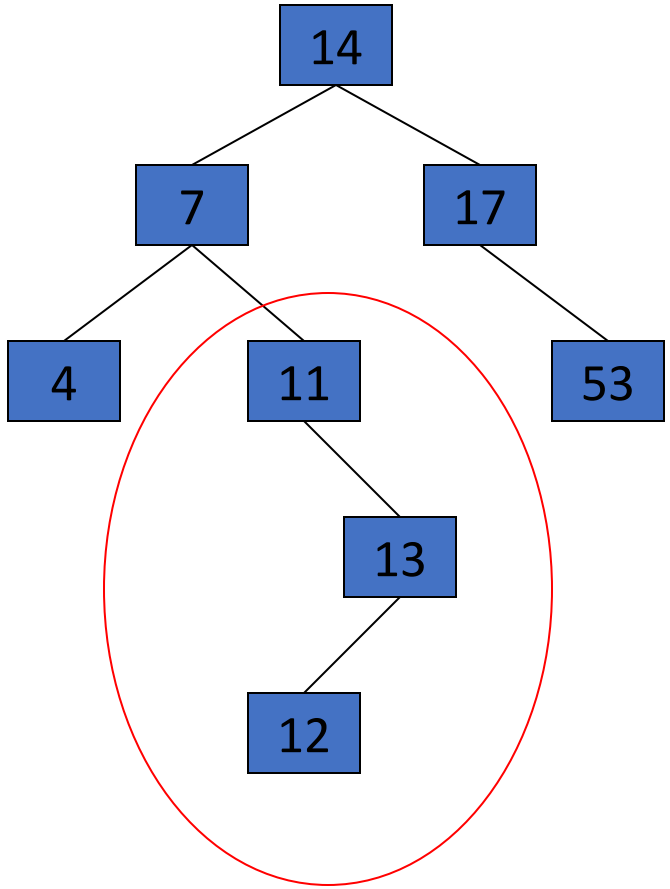
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



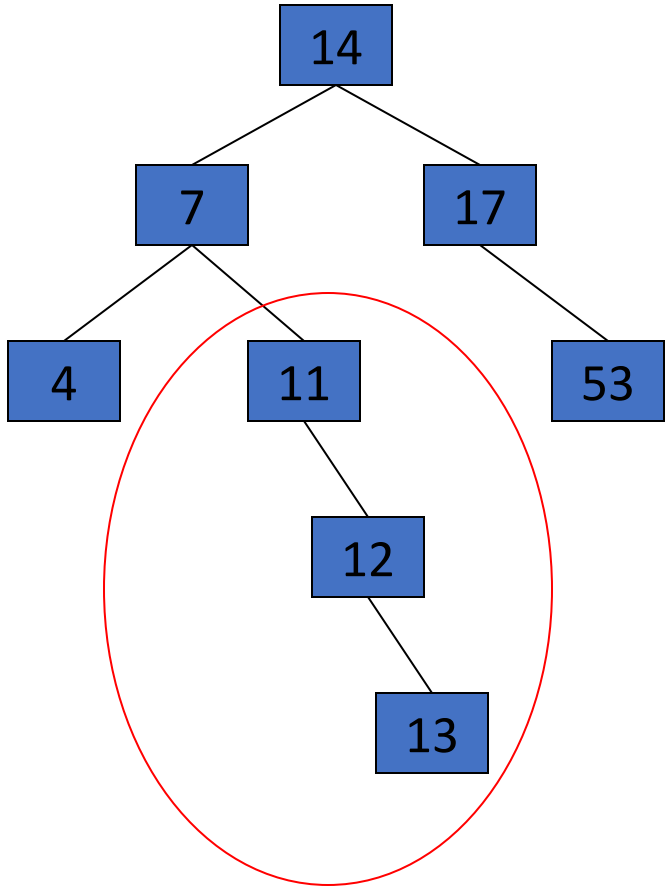
AVL Tree Example:

- **Now insert 12**



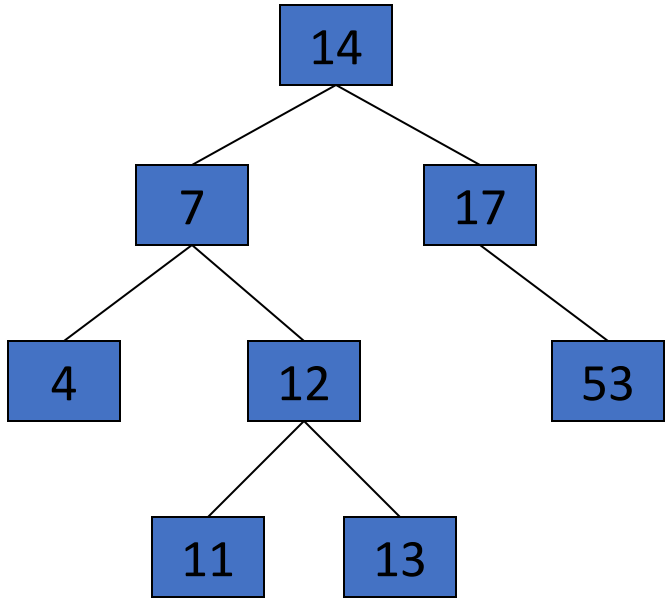
AVL Tree Example:

- **Now insert 12**



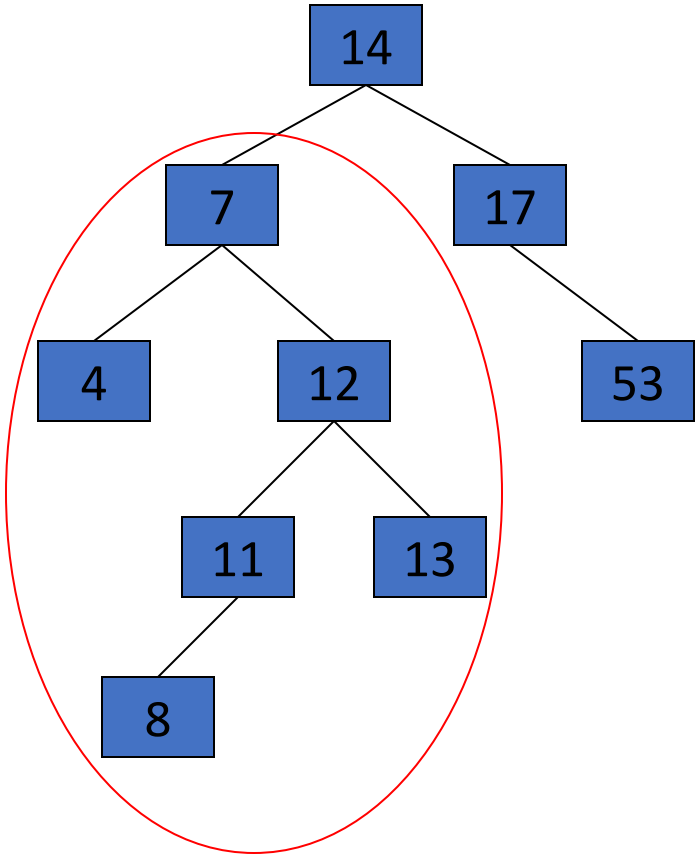
AVL Tree Example:

- **Now the AVL tree is balanced.**



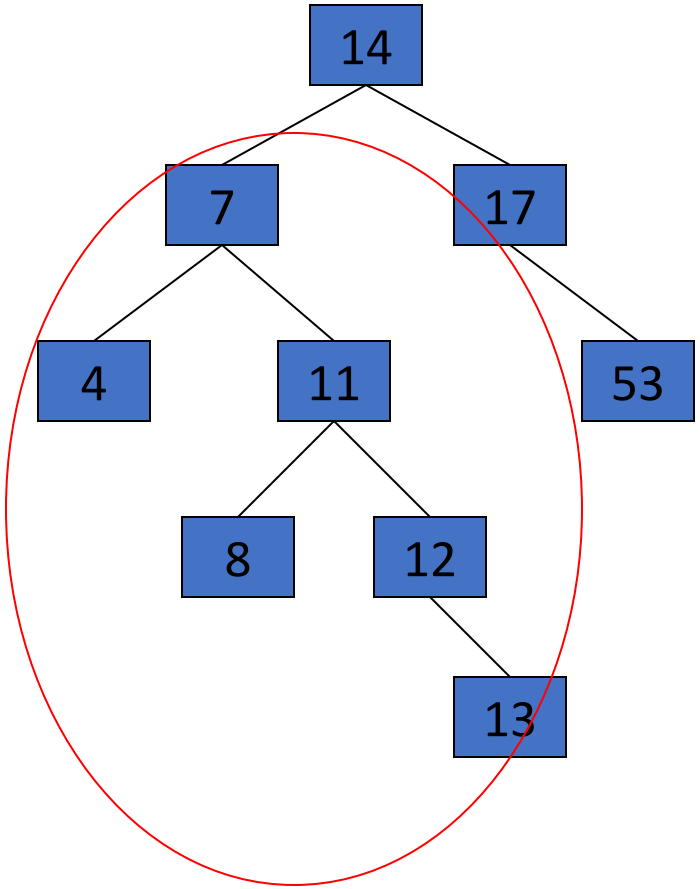
AVL Tree Example:

- **Now insert 8**



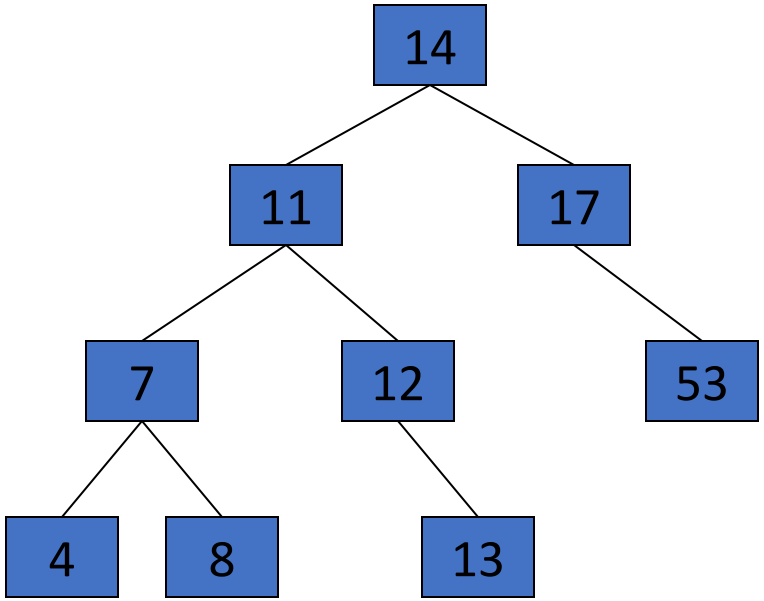
AVL Tree Example:

- **Now insert 8**



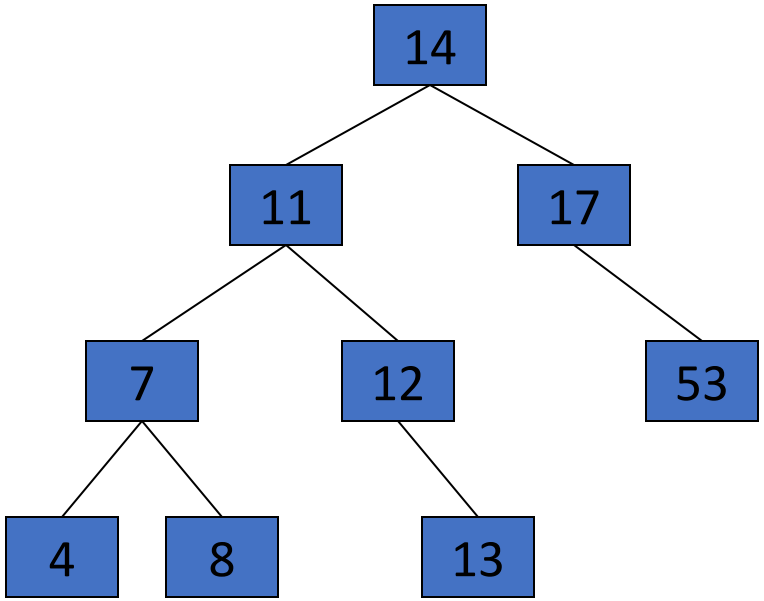
AVL Tree Example:

- **Now the AVL tree is balanced.**



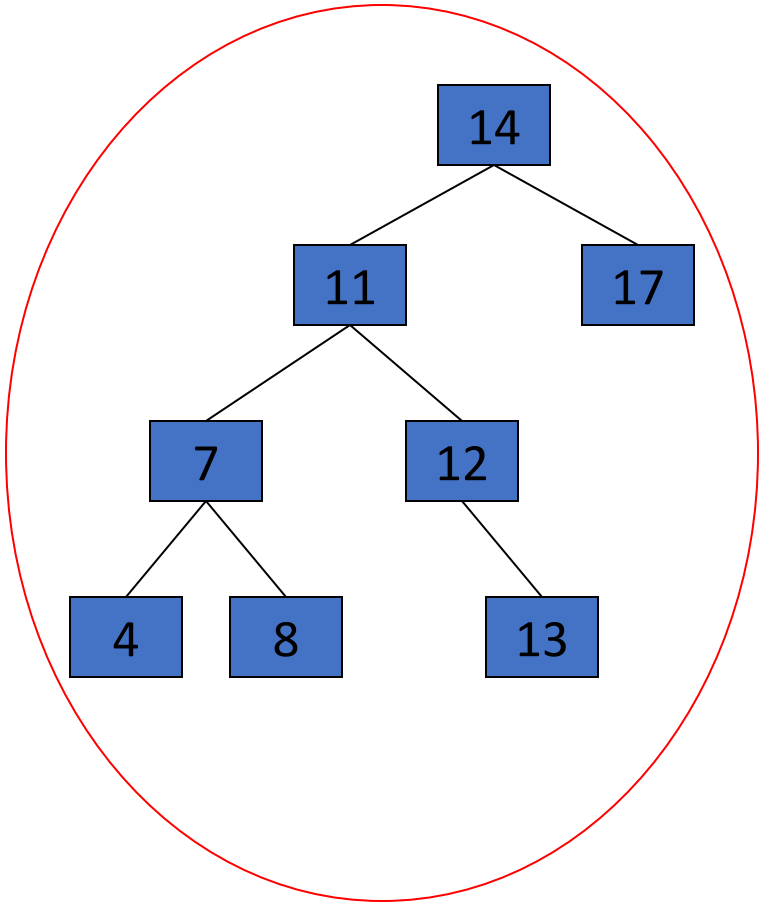
AVL Tree Example:

- **Now remove 53**



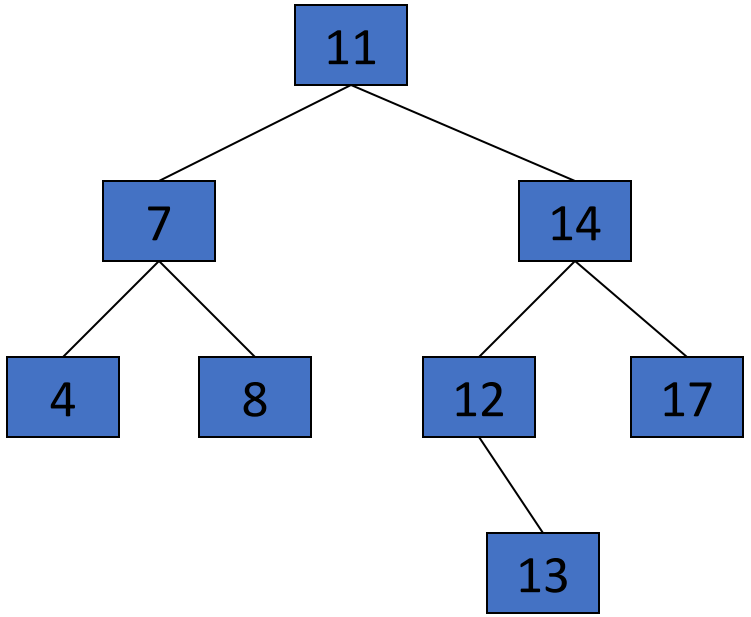
AVL Tree Example:

- **Now remove 53, unbalanced**



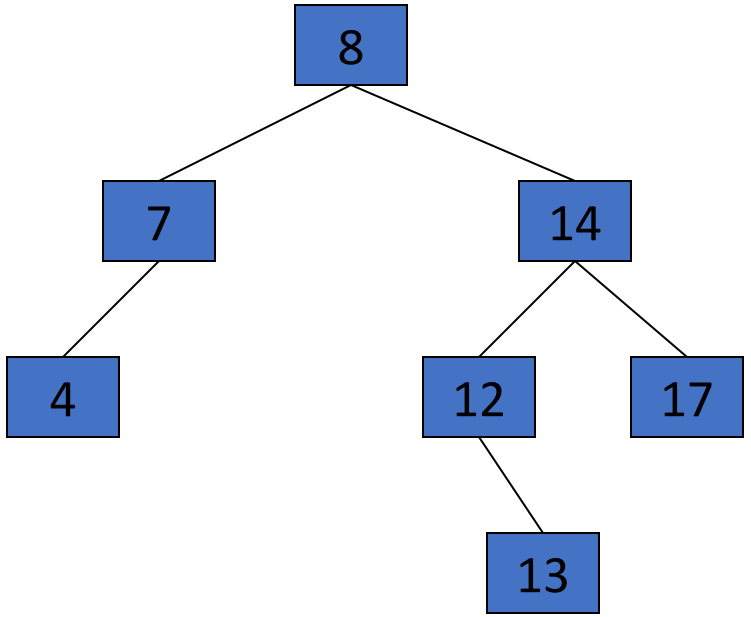
AVL Tree Example:

- **Balanced! Remove 11**



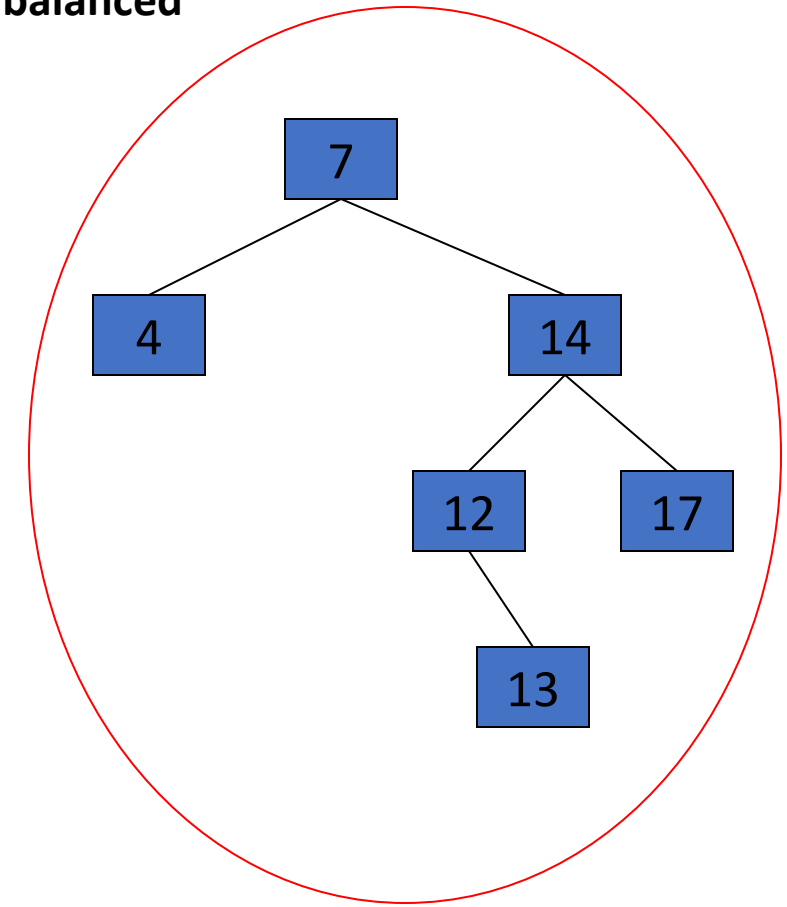
AVL Tree Example:

- **Remove 11, replace it with the largest in its left branch**



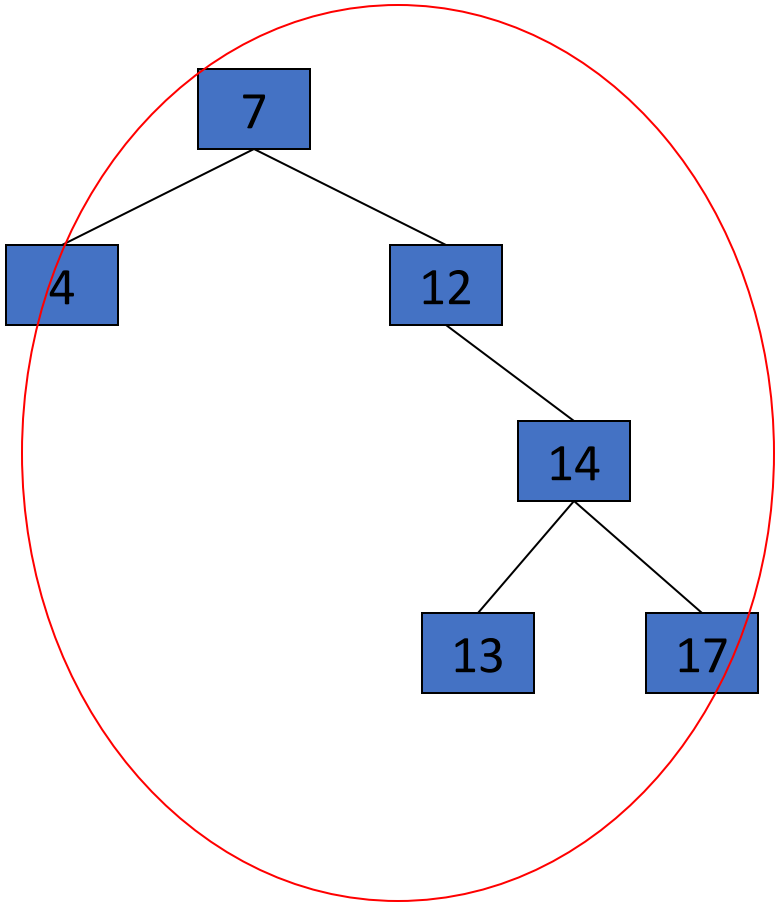
AVL Tree Example:

- **Remove 8, unbalanced**



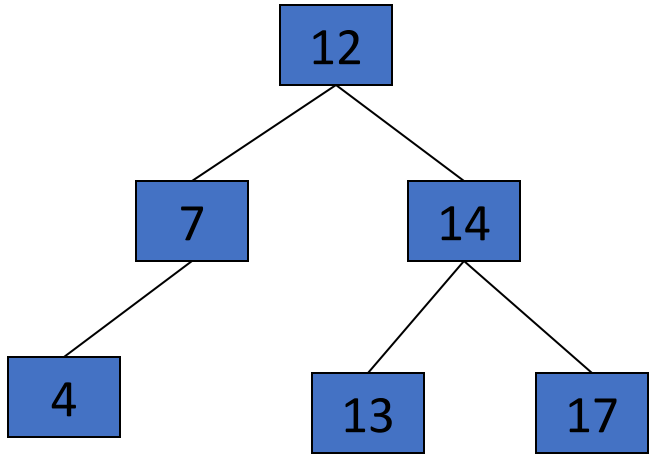
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

- **Balanced!!**

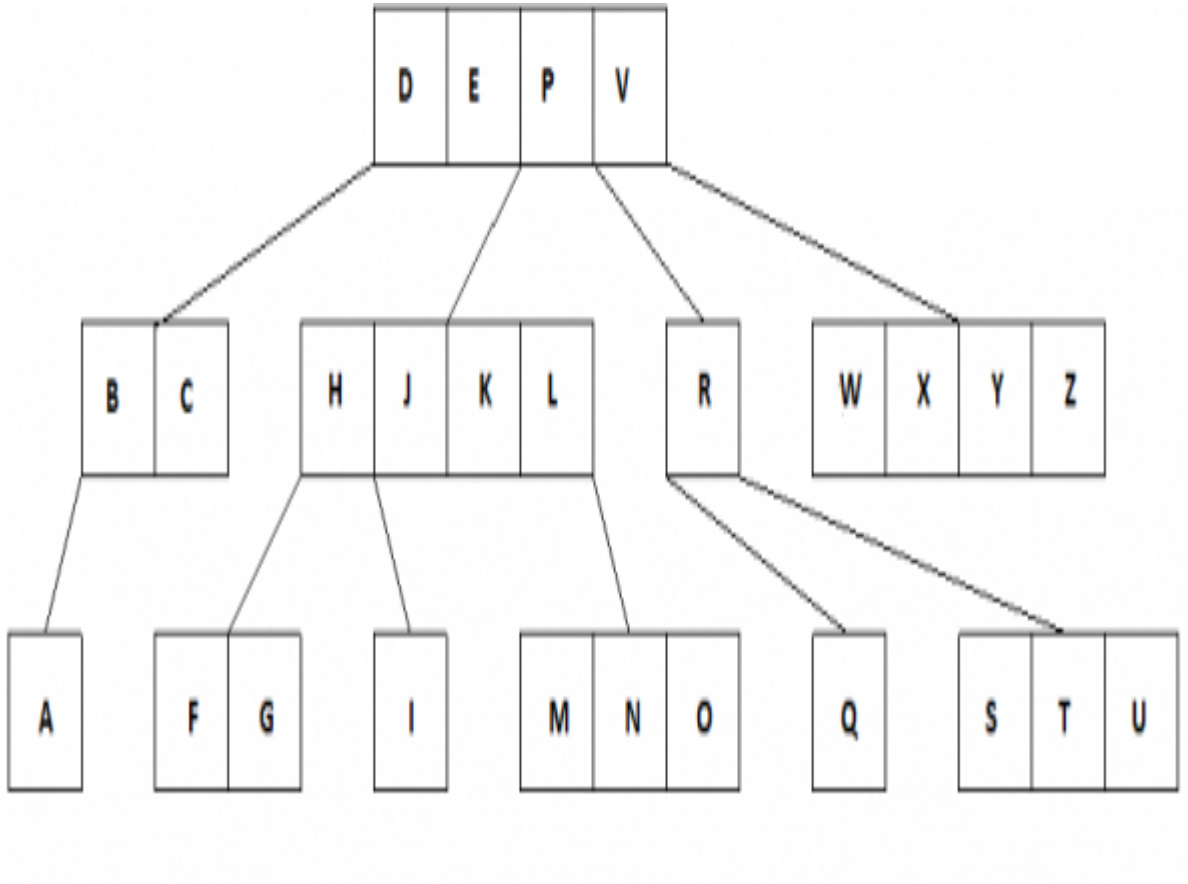


M-way Search Tree

A multi-way tree is defined as a tree that can have more than two children. If a multi-way tree can have maximum m children, then this tree is called as multi-way tree of order m (or an m-way tree).

As with the other trees that have been studied, the nodes in an m-way tree will be made up of m-1 key fields and pointers to children.

- Each node is associated with m children and m-1 key fields
- The keys in each node are arranged in ascending order.
- The keys in the first j children are less than the j-th key.
- The keys in the last m-j children are higher than the j-th key.



B-Tree

B-Tree is known as a self-balancing tree as its nodes are sorted in the inorder traversal.

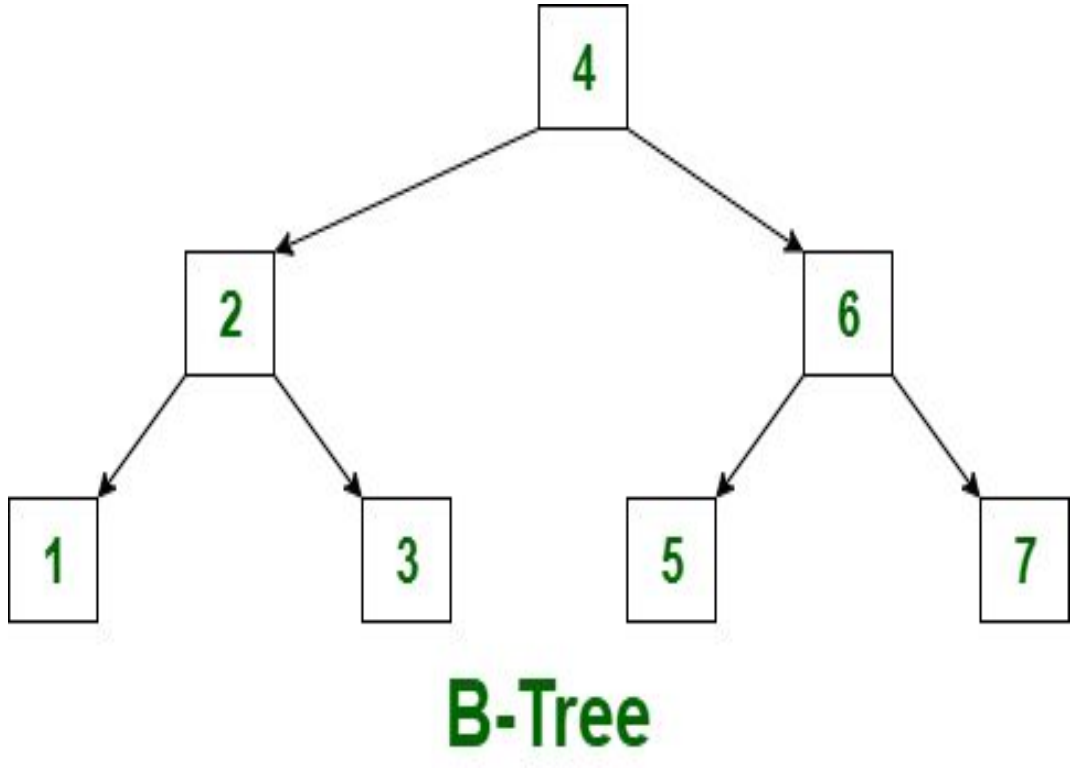
In B-tree, a node can have more than two children. B-tree has a height of logM N (Where ‘M’ is the order of tree and N is the number of nodes).

And the height is adjusted automatically at each update. In the B-tree data is sorted in a specific order, with the lowest value on the left and the highest value on the right.

To insert the data or key in B-tree is more complicated than a binary tree.

Some conditions must be hold by the B-Tree:

- All the leaf nodes of the B-tree must be at the same level.
- Above the leaf nodes of the B-tree, there should be no empty sub-trees.
- B- tree’s height should lie as low as possible.



B+ Tree

B+ tree eliminates the drawback B-tree used for indexing by storing data pointers only at the leaf nodes of the tree.

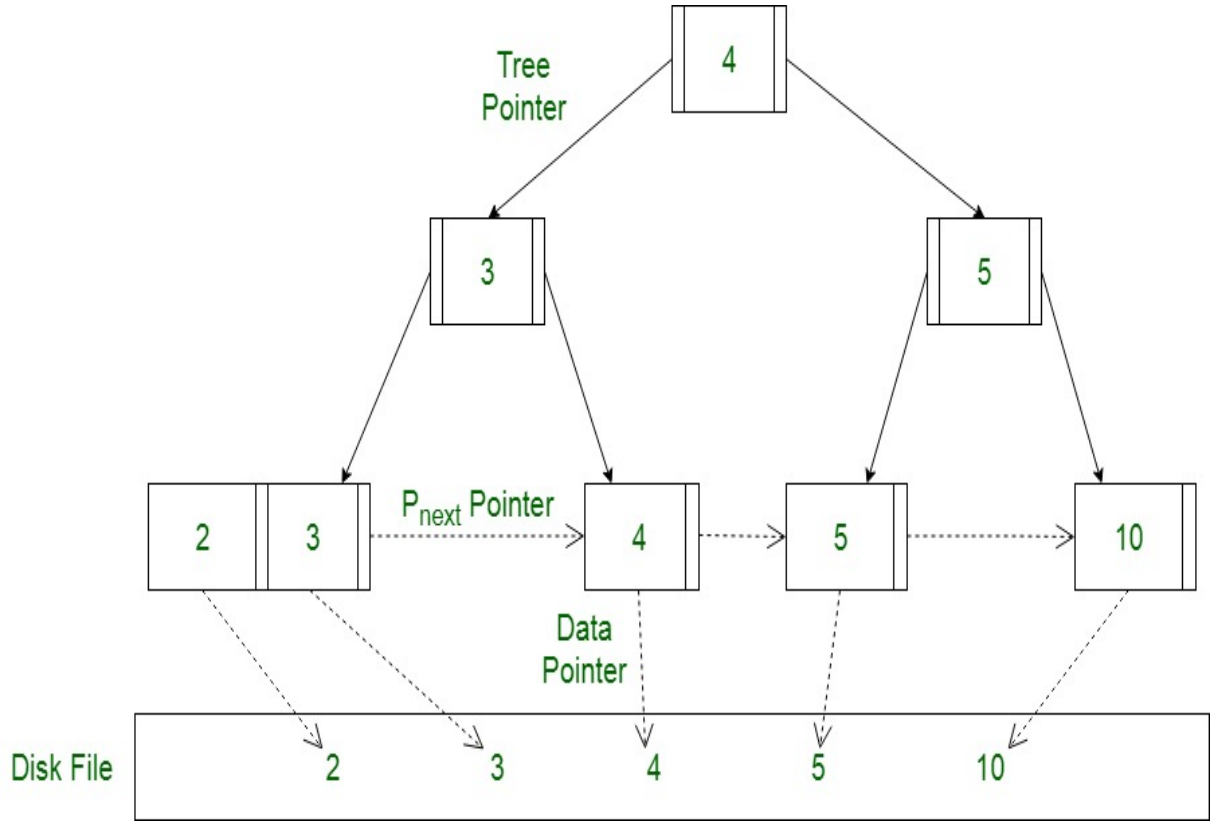
Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree.

It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, to access them.

Moreover, the leaf nodes are linked to providing ordered access to the records.

The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index.

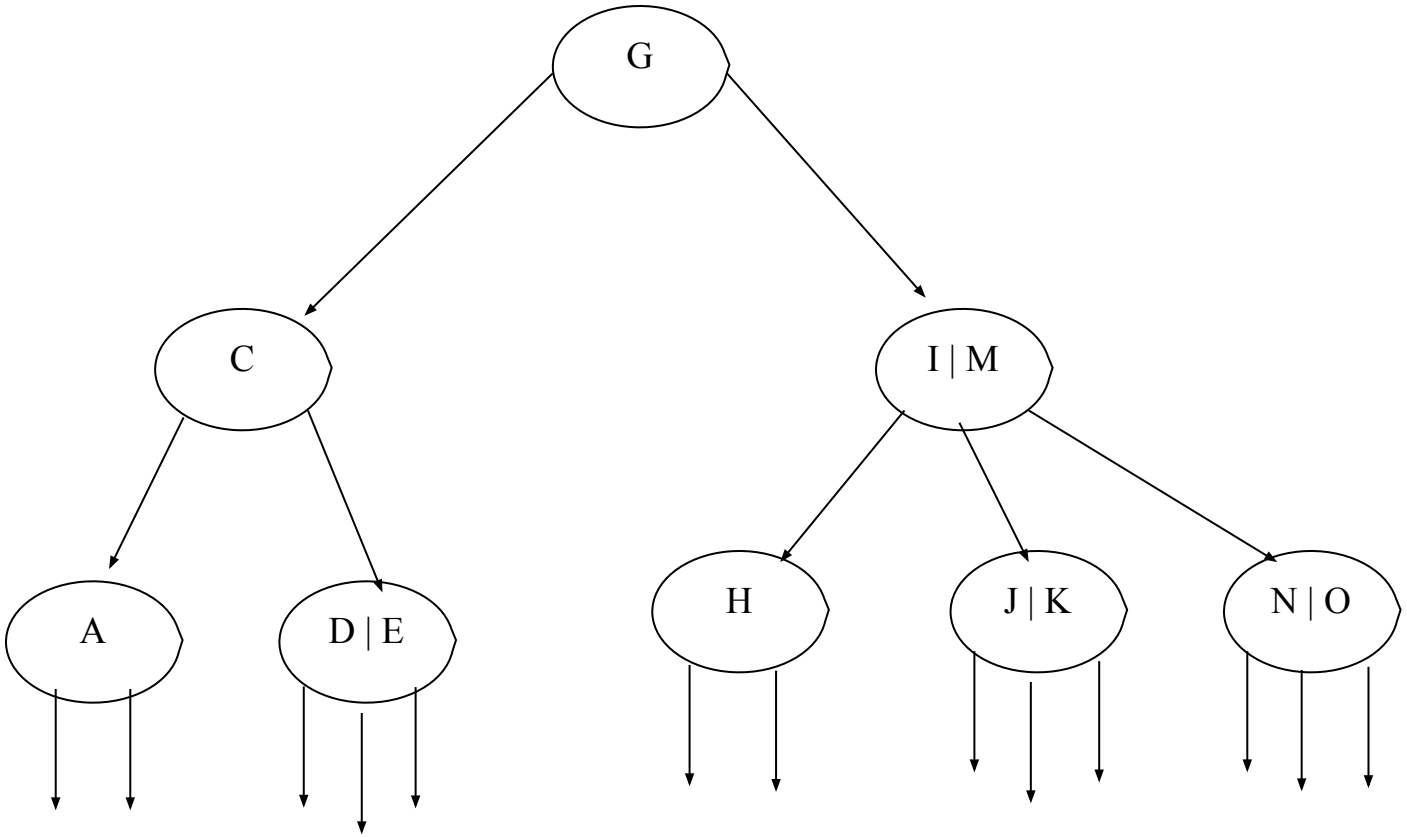
Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the searching of a record.



Definition of a B-Tree

- Def: B-tree of order m is a tree with the following properties:
  - The root has at least 2 children, unless it is a leaf.
  - No node in the tree has more then m children.
  - Every node except for the root and the leaves have at least $\lceil m/2 \rceil$ children.
  - All leaves appear at the same level.
  - An internal node with k children contains exactly k-1 keys.

2-3 Trees



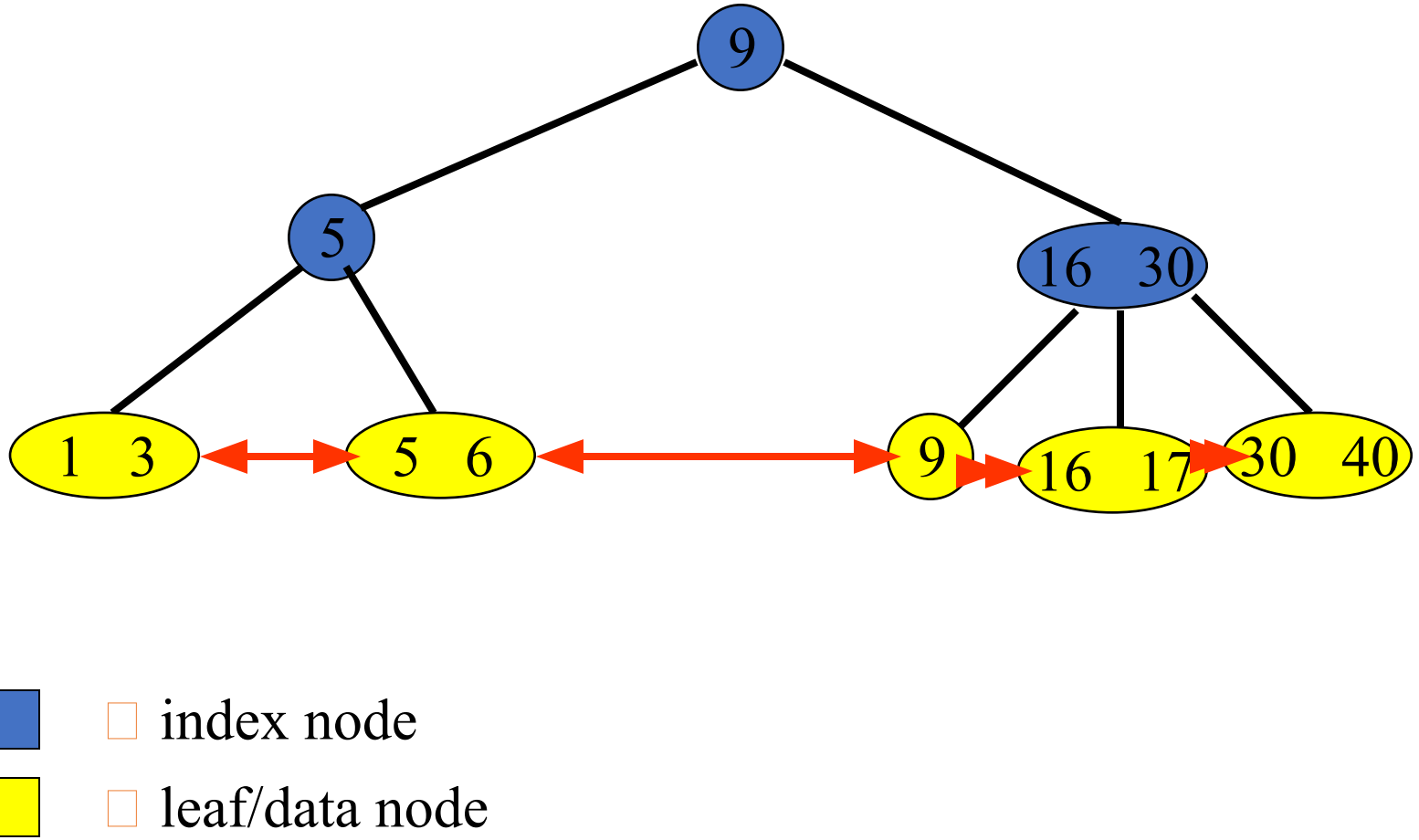
B^+-Trees

- Same structure as B-trees.
- Dictionary pairs are in leaves only. Leaves form a doubly-linked list.
- Remaining nodes have following structure:

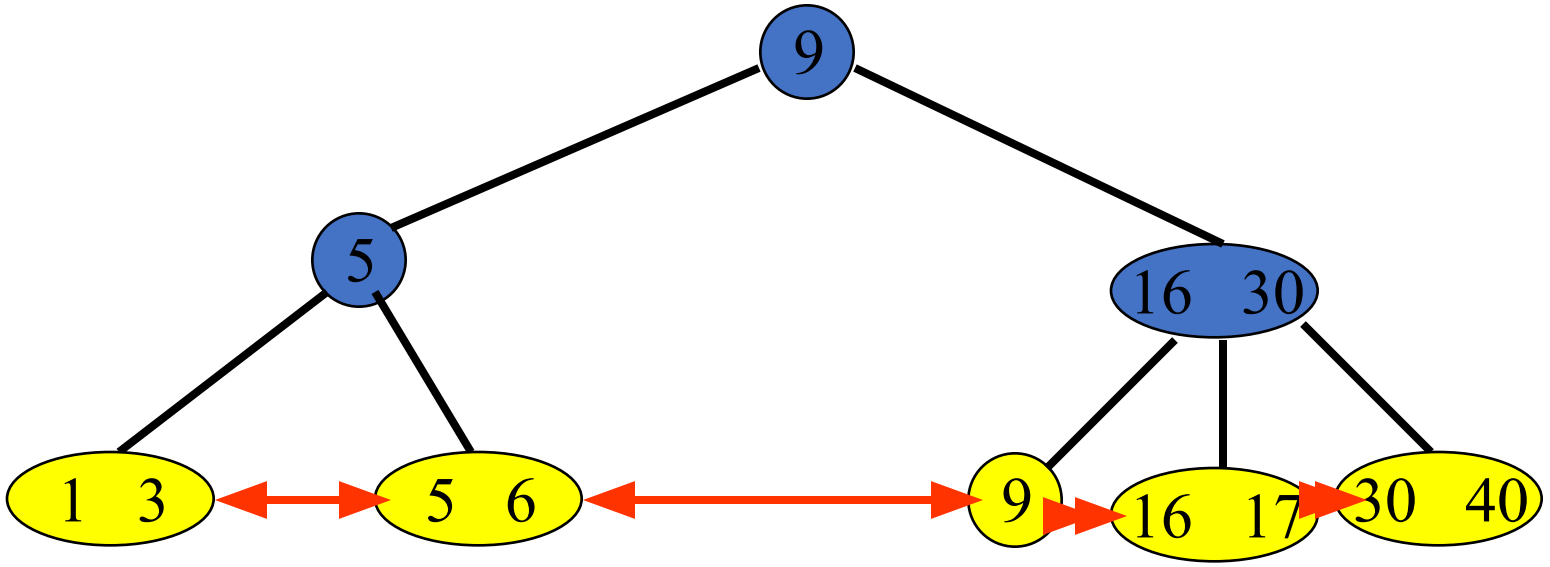
$$j\ a_0\ k_1\ a_1\ k_2\ a_2\ \ldots\ k_j\ a_j$$

- j = number of keys in node.
- a_i is a pointer to a subtree.
- k_i <= smallest key in subtree a_i and > largest in a_{i-1}.

Example B+-tree



B+-tree—Search



key = 5

6 <= key <= 20