## Chapter 7

# Trees

Data Structure & Algorithm

**Compiled by :**

**Ankit Bhattarai**
Email: ankit.bca@kathford.edu.np

## Syllabus

| Unit | Contents | Hours | Remarks |
|---|---|---|---|
| 1. | Introduction to Data Structure | 2 | |
| 2. | The Stack | 3 | |
| 3. | Queue | 3 | |
| 4. | List | 2 | |
| 5. | Linked Lists | 5 | |
| 6. | Recursion | 4 | |
| 7. | Trees | 5 | |
| 8. | Sorting | 5 | |
| 9. | Searching | 5 | |
| 10. | Graphs | 5 | |
| 11. | Algorithms | 5 | |

Credit : 3

## Outlines

## Introduction: Trees

- Until now we have studied linear data structures. But there are many applications where only the linear data structures are not sufficient.
- For such situations, we use non-linear data structures like trees and graphs.
- A tree is a hierarchical structure in which hierarchical relationship exist among several data items.
- A tree consists of nodes with parent child relationship.

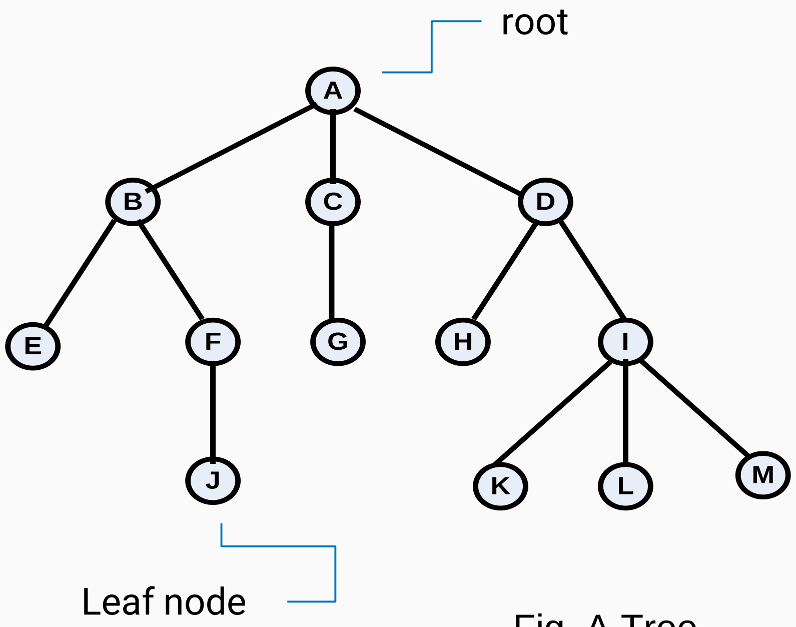


Fig. A Tree

## Introduction: Basic terminologies in a tree

- **Root**: Root is a special data item, which is the first in the hierarchy. In the above tree,' A' is the root item.
- **Node**: Each item in a tree is called a node. A, B, F, G etc. are examples of node.
- **Parent**: A node that points to (one or more) other nodes is the parent of those nodes while the nodes pointed to are the **children**. B is parent of E, F and E,F are children of B. Similarly, I has three children K, L & M.

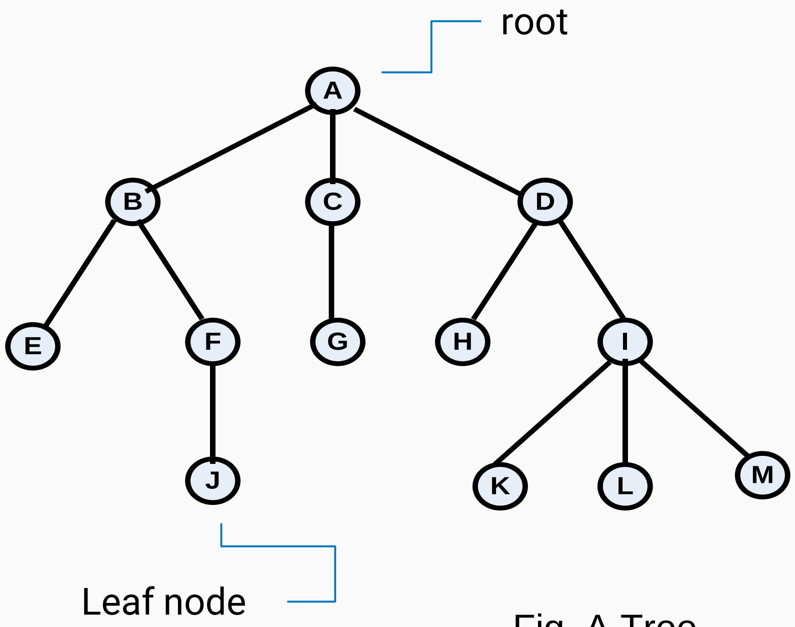


Fig. A Tree

## Introduction: Basic terminologies in a tree

- **Leaf (External Nodes):** Nodes with no children are leaf nodes or external nodes or terminal nodes. In the above tree E, J, G, H, K, L, and M are terminal nodes.
- **Internal Nodes:** Nodes with children are internal nodes or non-terminal nodes. Note that the root node is also an internal node. Node such as B, C, D are non-terminal nodes.
- **Siblings (Brother):** Nodes that have the same parent are siblings. E.g. E and F are siblings of parent node B.

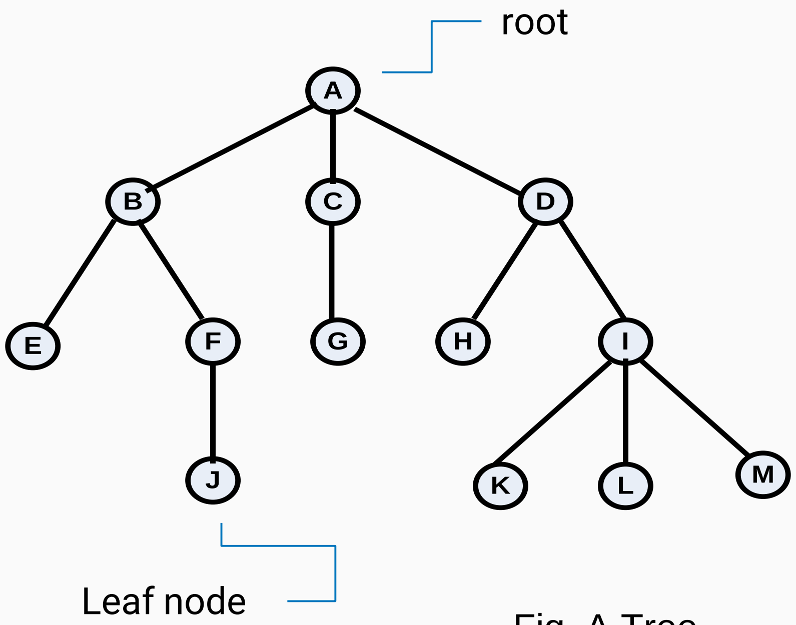


Fig. A Tree

## Introduction: Basic terminologies in a tree

- **Order (Degree):** Order (degree) of a **node** is given by the number of its children. If a node has two children, its degree is two. In the above tree:

✓ degree of node A is 3.

✓ degree of node B is 2.

✓ degree of node C is 1.

- Similarly, **order (degree) of a tree** is given by the maximum of the degrees of its nodes. So the degree of the above tree is 3.

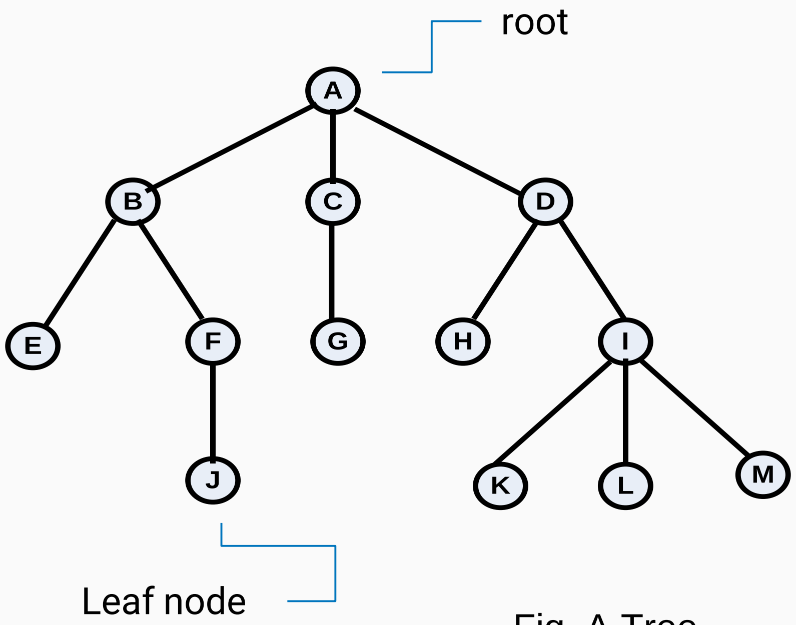


Fig. A Tree

## Introduction: Basic terminologies in a tree

- **Edge:** The link between any two nodes is called an edge. It is directed from the parent towards its child.

- **Path:** There is a single, unique path from the root to any node. For e.g. the path from node A to J is A, B, F, and J.

- **Height:** A height of a node is equal to the maximum path length from that node to a leaf node. A leaf node has a height of 0. The height of a tree is equal to the height of the root. For e.g. the height of node B from leaf node J is 2.

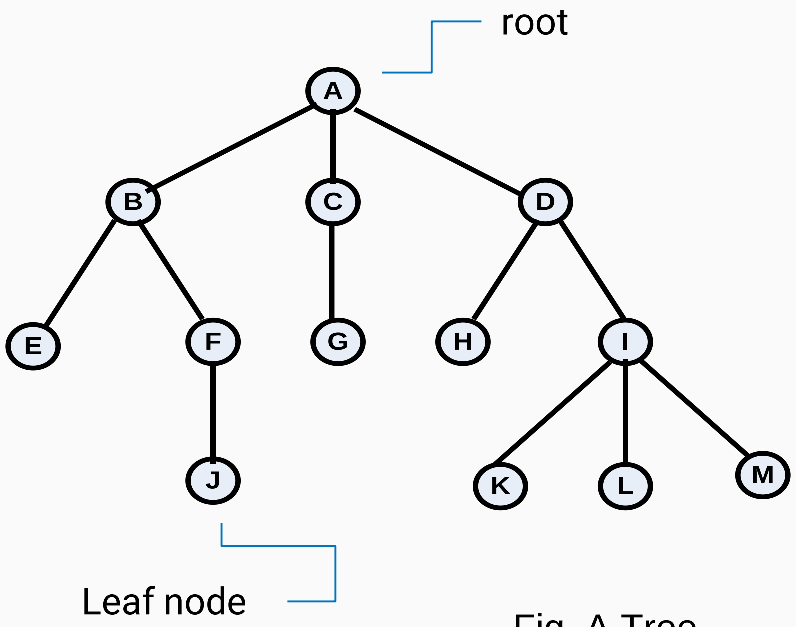


Fig. A Tree

## Introduction: Basic terminologies in a tree

- **Depth:** A depth of a node is equal to the maximum path length from root to that node. A root has a depth of 0. The depth of a tree is equal to the depth of the deepest node in the tree. It, of course, is equal the tree height. For e.g. the depth of node J is 3.
- **Level**: The level of a node is 0, if it is root. Otherwise, it is one more than its parent. For e.g.

✓ Node C is at level 1.

✓ Node K is at level 3.

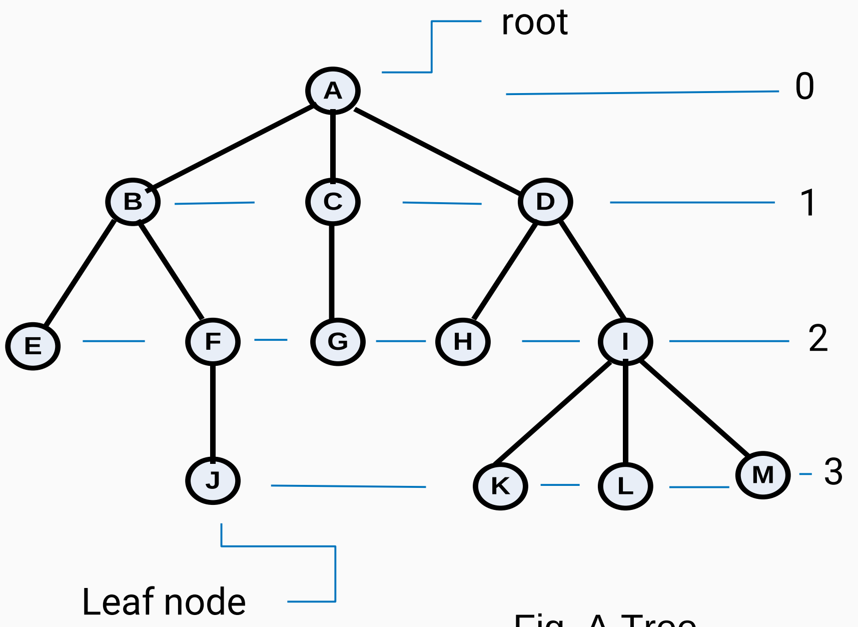


Fig. A Tree

## Binary Tree

## Binary Tree: Introduction

- A binary tree is a finite set of data items which is either empty or consists of a single item called the root and the two disjoint binary trees called the **left subtree** and **right subtree.**
- In a binary tree the maximum degree of any node is **at most two.** That means, there may be a zero degree node (usually an empty tree) or one degree node or two degree node.
- Fig below shows a binary tree consists of 9 elements.

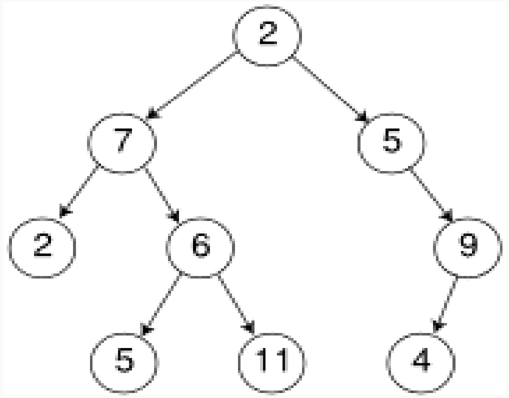
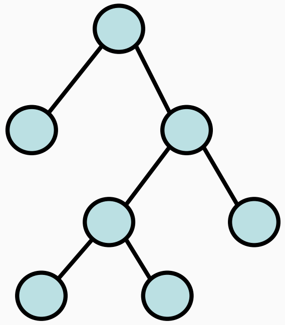


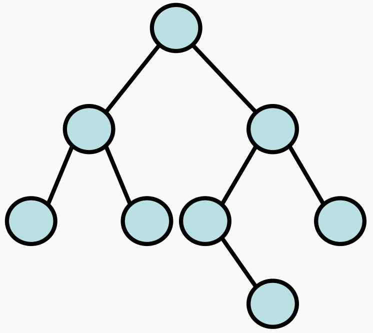
Fig. Binary Tree

## Binary Tree: Types

i. Strictly Binary Tree

ii. Complete Binary Tree


(i)


(ii)

**i. Strictly Binary Tree**

- A **Strictly binary tree** is a tree in which every node has zero or two children. i.e. Every internal node has non empty left and right subtrees.
- A strictly binary tree with n leaves always contains 2n -1 nodes.
- Here the first tree from the figure (i) is a Strictly binary tree , while the second one (ii) is not.

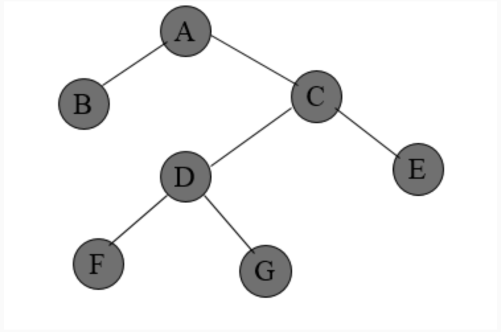
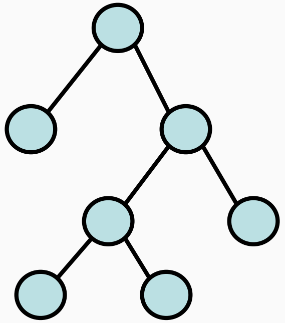


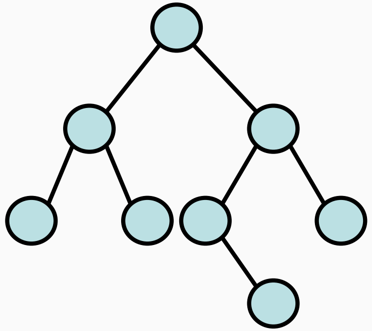
Fig. Strictly Binary tree

## Binary Tree: Types

ii. **Complete Binary Tree**

- A Complete binary tree is a strictly binary tree in which all leaves are at the same depth.
- In other words, a binary tree with n nodes and of depth d is a strictly binary tree all of whose terminal nodes are at level d.
- In a complete binary tree, there is exactly one node at level 0, two nodes at level 1, and four nodes at level 2 and so on.


(i)


(ii)

## Binary Tree: Operations

1. **Create :** It creates an empty binary tree
2. **MakeBT** : It creates a new binary tree having a single node with data field set to some values.
3. **EmptyBT** : It returns true if the binary tree is empty otherwise it returns false.
4. **Lchild** : It returns pointer to the left child of the node. If the node has no left child it returns the null pointer.
5. **Rchild** : It returns pointer to the right child of the node. If the node has no right child it returns the null pointer.
6. **Father** : It returns a pointer to the father of the node. Otherwise it returns the null pointer.
7. **Sibling** : It returns a pointer to the brother of the node. Otherwise returns the null pointer.
8. **Data** : It returns the contents of the node.

## Binary Tree: Other Operations

**Others Operations** that can be applied to the binary tree are:

- ✓ Tree traversal
- ✓ Insertion of nodes
- ✓ Deletion of nodes
- ✓ Searching for the node
- ✓ Copying the binary node

## Traversing in Binary Tree

- Visiting the nodes in certain order is called **traversing**.

- Tree, being a non-linear data structure, there are three different ways for traversal.

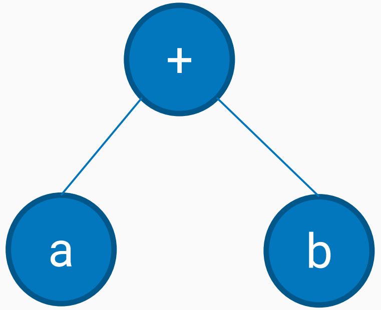
1. **Preorder** Traversal (Root, Left, Right)
2. **In order** Traversal (Left, Root, Right)
3. **Post order** Traversal (Left, Right, Root)

## 1. Preorder Traversal

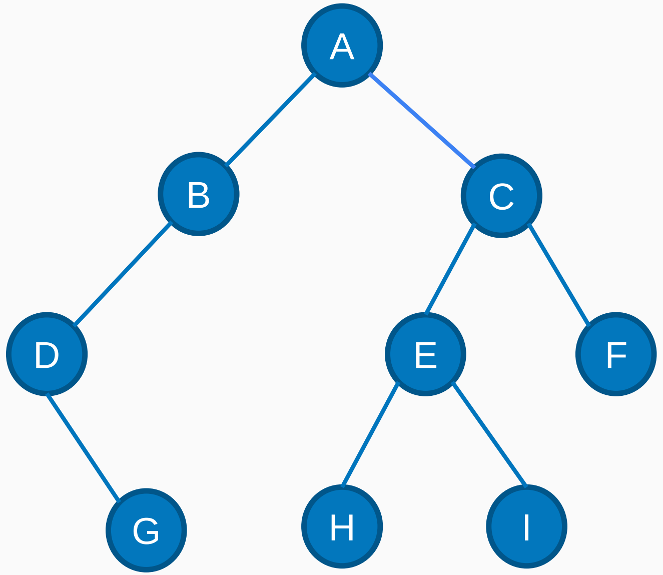
(Root, Left, Right)

1. Visit the root node.
2. then, visit the nodes in the left subtree in preorder.
3. finally, visit the nodes in the right subtree in preorder.

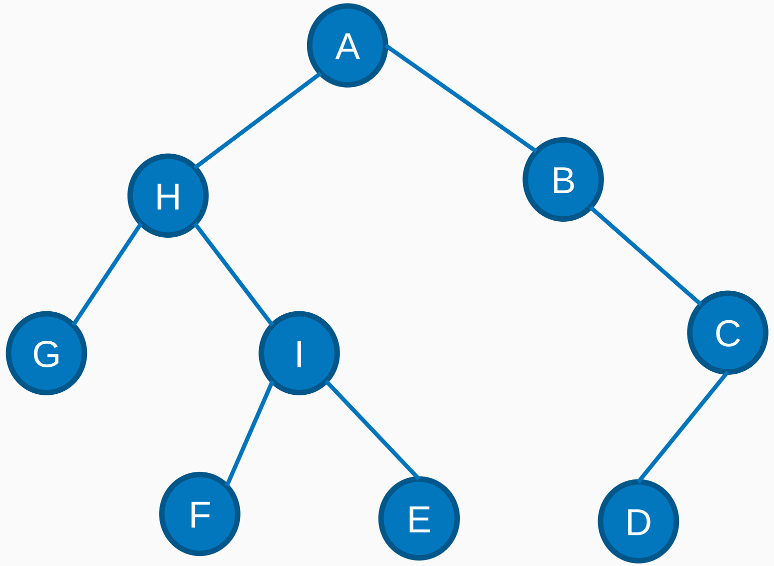
**Preorder**: +ab



## 1. **Preorder Traversal** (Root, Left, Right)



**Preorder**: ABDGCEHIF

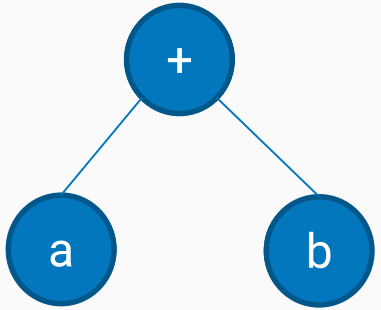


**Preorder**: AHGIFEBCD

## 2. In order Traversal

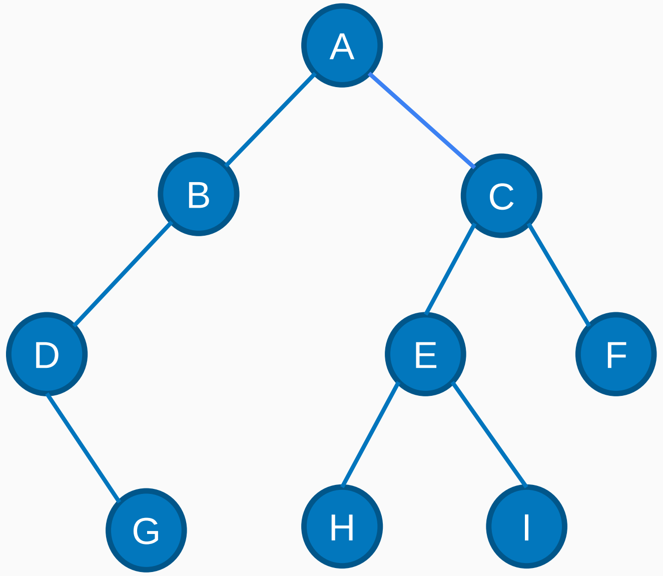
(Left, Root, Right)

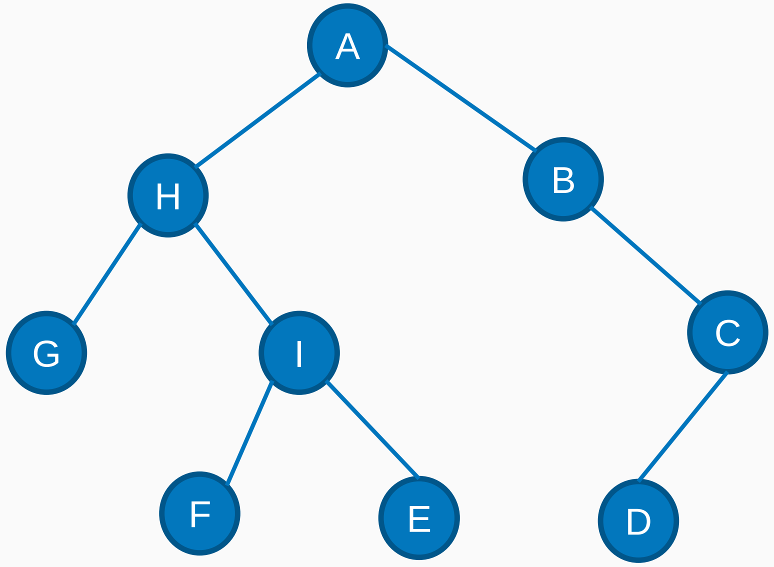
1. Visit the nodes in the left subtree in in order .
2. Visit the root node.
3. Finally, visit the nodes in the right subtree in in order.



**In order**: a + b

## **2. In order Traversal** (Left, Root, Right)





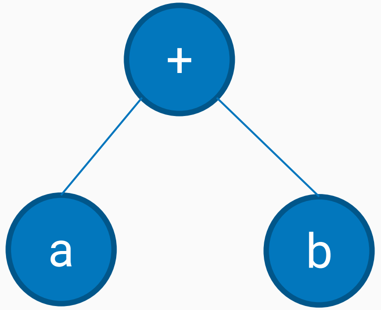
**In order**: DGBAHEICF

**In order**: GHFIEABDC

## 3. Post order Traversal

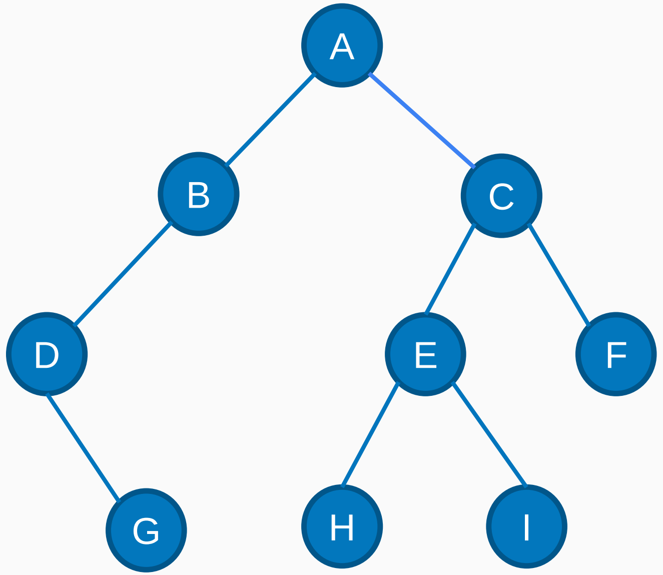
(Left, Right, Root)

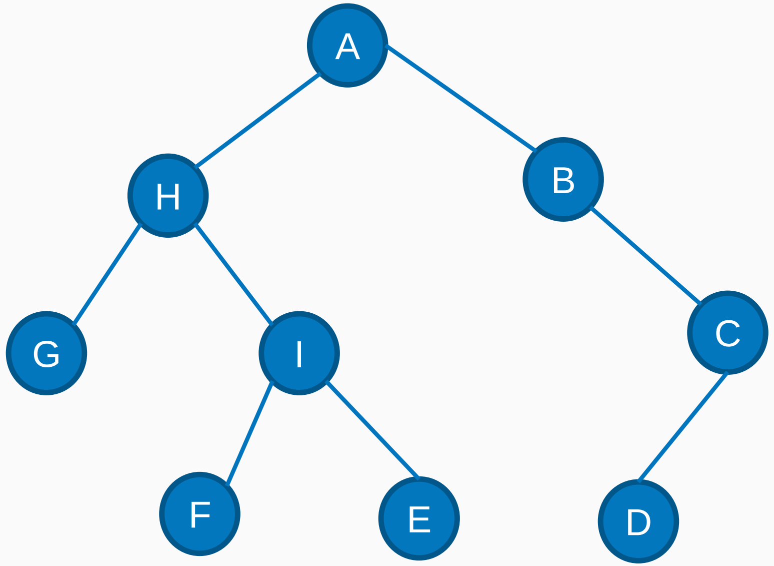
1. Visit the nodes in the left subtree in post order.
2. then, visit the nodes in the right subtree in post order.
3. Finally, visit the root node .



**Post order**: ab +

## 3. Post order Traversal (Left, Right, Root)

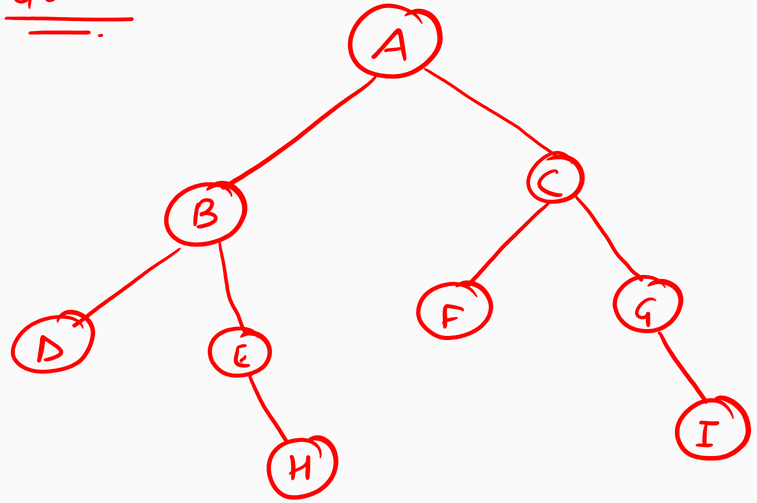




**Post order**: GDBHIEFCA

**Post order**: GFEIHDCBA

Question :



Preorder :

Inorder :

Post order :

## Binary Search Tree

- A Binary Search Tree is a binary tree which is either empty, or in which every node contains a key and satisfies the following criteria:

- All keys (if any) in the left –subtree is smaller than the key in the root.
- All keys (if any) in the right subtree is larger than the key in the root.
- The left and right subtrees are again BST.

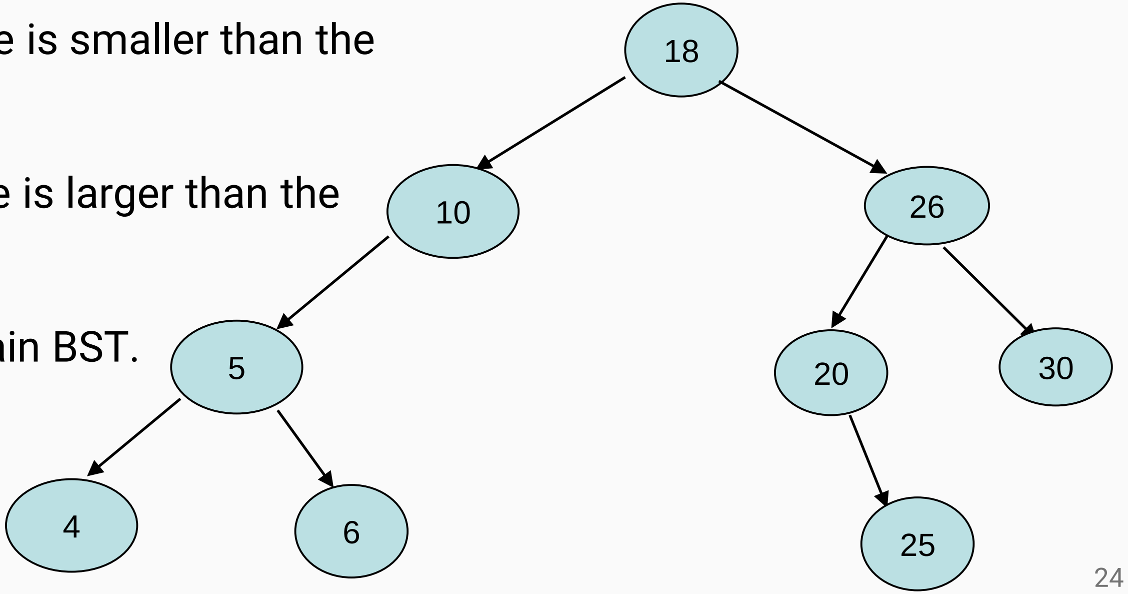


Fig. Binary Search Tree

## Operations on a Binary Search Tree

1. **Search(x, T):** Search for key x in the BST T. if x is found in some node n of T return a pointer to node n or NULL otherwise.
2. **Insert (x, T):** Insert a new node with x in the info part in the tree T such that the property of BST is maintained.
3. **Delete(x, T):** Delete a node x in the info part from the tree T such that the property of BST is maintained.
4. **FindMin(T):** Returns pointer to a node having minimum key in tree T, NULL if T is empty.
5. **FindMax(T):** Returns pointer to a node with maximum key in T, NULL if T is empty.

## Searching on a Binary Search Tree

In BST, searching of the desired data item can be performed by branching into left or right subtree until the desired data item is found. We proceed from the root node.

- ✓ If the tree is empty, then we failed to find x, so return NULL.

Else, compare x with the key stored in the root of T.

- ✓ If x is same as the key in root, return address of the root.
- ✓ If x is smaller than the key in root, search for x in the left subtree of T recursively.
- ✓ If x is greater than the key in root, search for x in right subtree recursively.

## Insert Operation on a Binary Search Tree

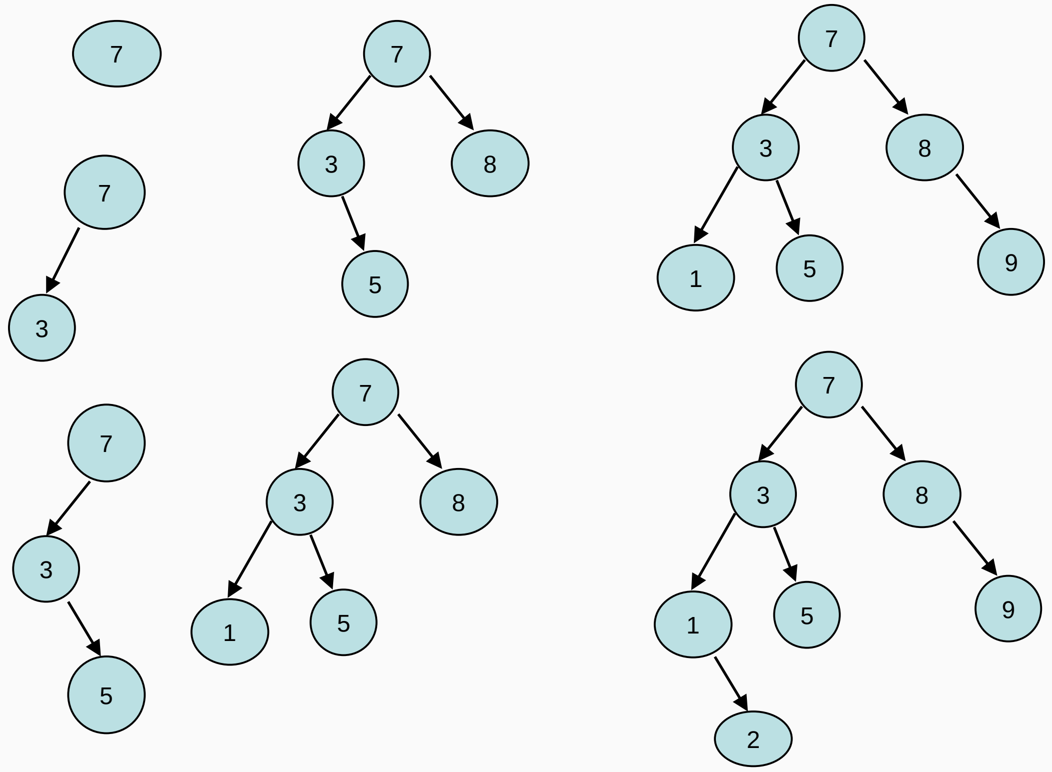
**Algorithm**

Given a new item with key x, then following algorithm will insert it in BST T.

- If the tree is empty, create a new node n storing the key x and make this new node n the root of the tree. Otherwise, compare x to the key stored at root R.
- If x is identical to key in R, don’t change the tree.
- If x is smaller, insert new item into left subtree of the root recursively.
- If x is larger, insert new item into right subtree of the root recursively.

Construct a BST using the following data:

7, 3, 5, 8, 1, 9, 2



Construct a BST using the following data inserted in an order:

39, 45, 30, 60, 42, 35, 25, 32, 44

Also, perform different tree traversals.

Construct a BST using the following data inserted in an order:

D, A, T, A, S, T, R, U, C, T, U, R, E

Also, perform different tree traversals.

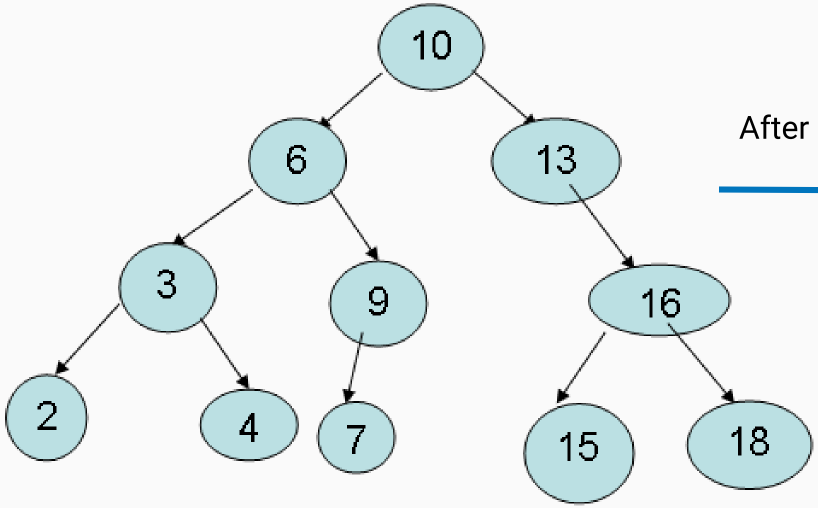
## Delete Operation on a Binary Search Tree

**Algorithm**

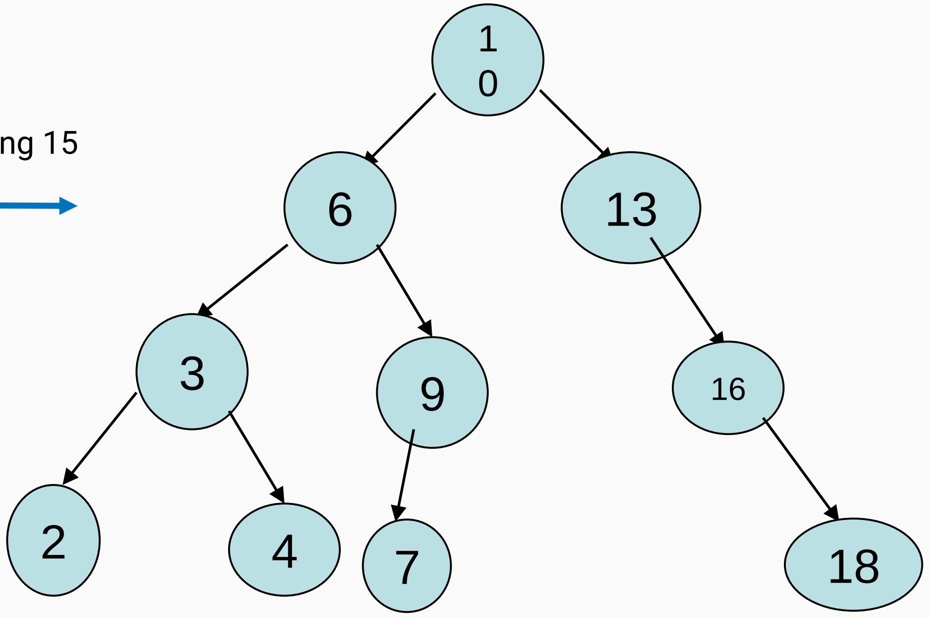
To delete a given node (key), it is searched first. If search is unsuccessful, the algorithm terminates. Otherwise, the following possibilities arise:

- ✓ If node is a leaf, it can be deleted immediately.
- ✓ If the node has one subtree, the node can be deleted by replacing it with the root of the non-empty subtree.
- ✓ If the node has two subtrees, we replace the node with the maximum node of the left subtree T or the minimum node of the right subtree T.

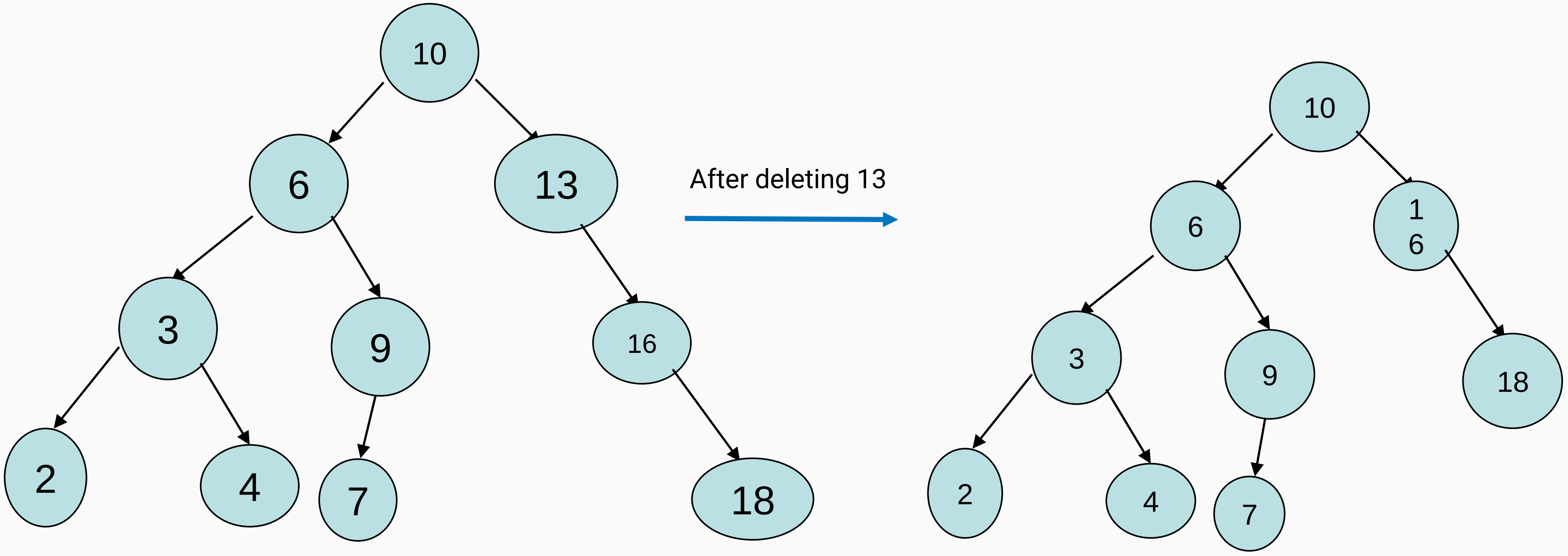
## Delete a leaf node in BST



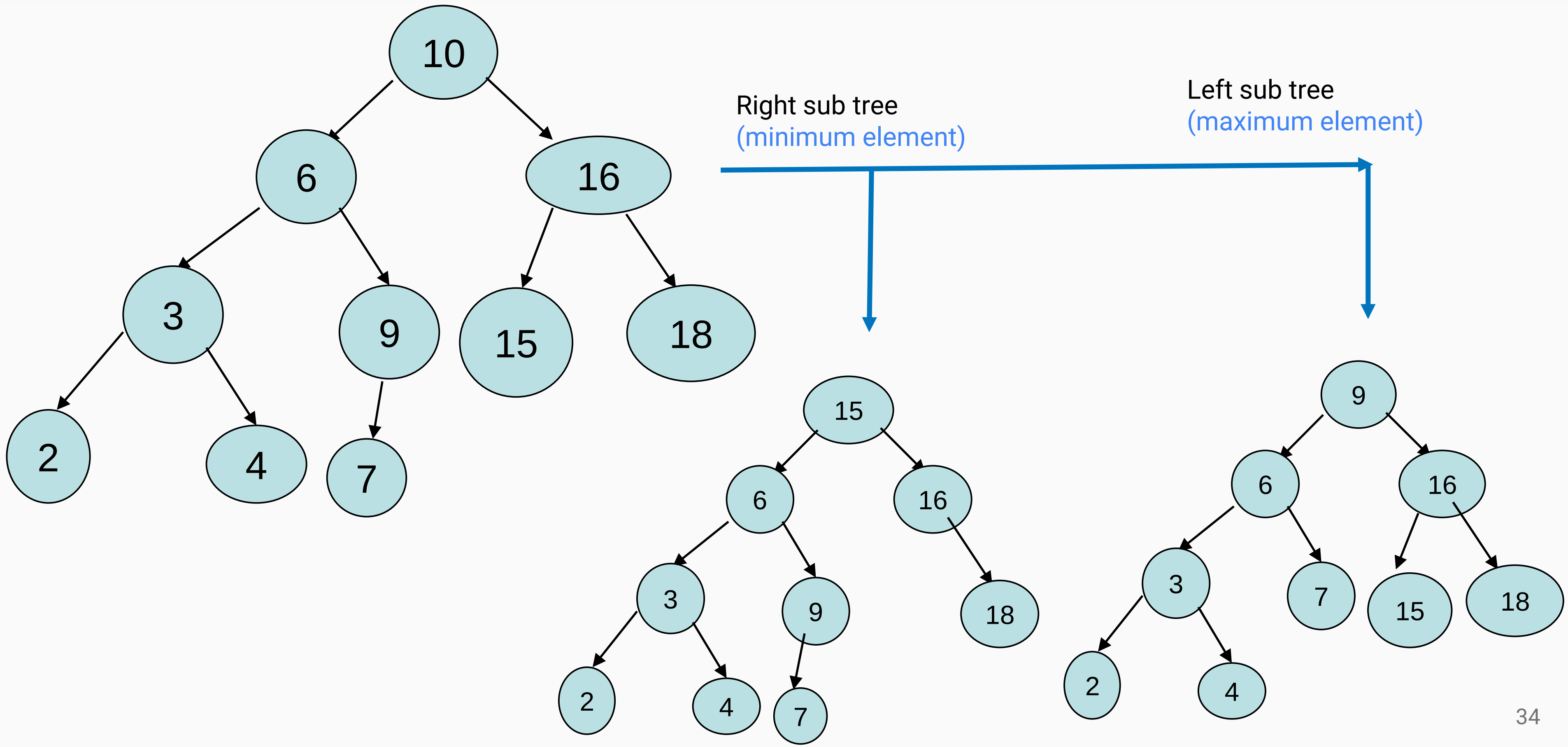
After deleting 15



## Delete a internal node in BST : Node of 13



## Delete a root node in BST : Node of 10



## Efficiency of a Binary Search Tree (BST)

1. Complexity of BST operations is proportional to the length of the path from the root to the node being manipulated, i.e., height of the tree

2.In a well balanced tree, the length of the longest path is roughly log n, where n is the no of nodes in the tree

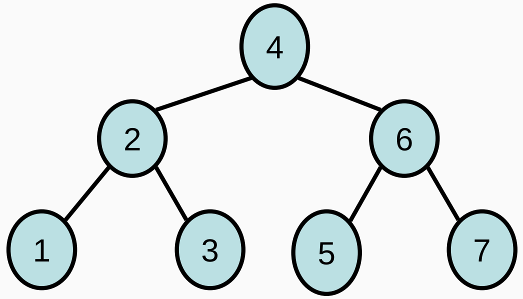
✓ E.g., 1 million entries → longest path is log21,048,576 = 20

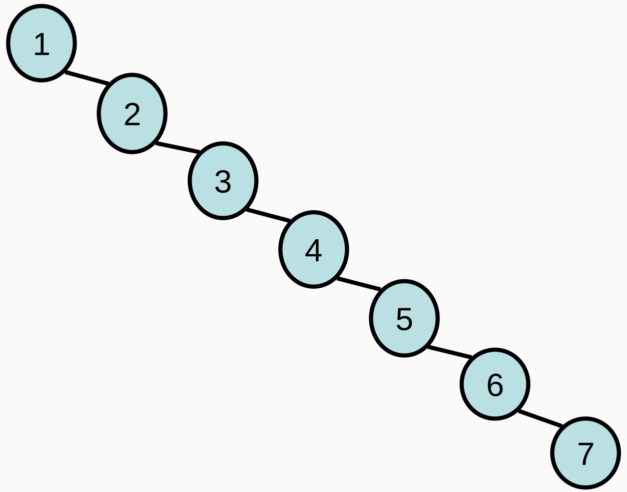
3. For a thin, unbalanced tree, operations become O(n):

✓ E.g., elements are added to tree in sorted order

4. So, Balancing is important to decrease the computational complexity

## Efficiency of a Binary Search Tree (BST)





In the left tree, the Search operation requires 3 comparisons to find 7.

In the right tree, the Search operation requires 7 comparisons to find 7

## Balanced Binary Tree

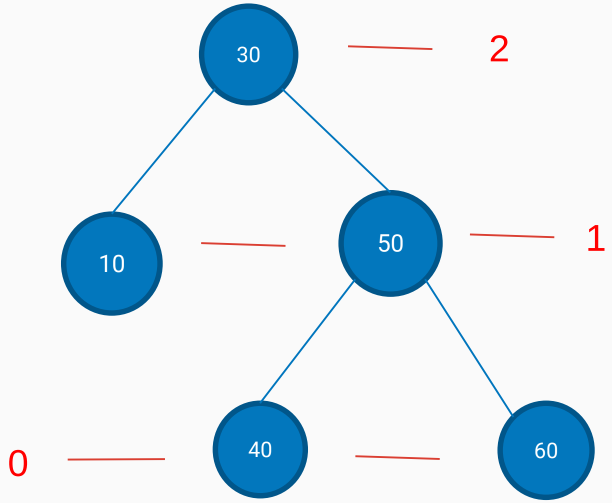
- A balanced binary tree is the one in which the largest path through the left sub tree is the same length as the the largest path of the right sub tree i.e. from root to leaf.
- Searching time is very less in balanced binary tree compared to unbalanced binary tree i.e. balanced trees are used to maximize the efficiency of the operations on the tree.
- **Height balanced tree**: In height balanced tree, balancing the height is the important factor.

## AVL Tree

## AVL Tree

- An AVL tree is a binary search tree in which the heights of the left and right subtree of the root differ by, at most, 1 and in which the left and right subtree are again AVL trees. It is named after their inventors G.M. Adelson, Velskii and E.M. Landis.

**Balance Factor = height(left subtree) – height (right subtree)**



## Insertion in AVL Tree

1. Start
2. Insert the node in the same way as in ordinary binary tree.
3. Trace a path from the new nodes back towards the root for checking the height difference of the sub trees of each node along the way.
4. Consider the node with the imbalance and the two nodes on the layers immediately below.
5. If these three nodes lies in a straight line, apply a single rotation to correct the imbalance.
6. 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.
7. Stop

## Insertion in AVL Tree

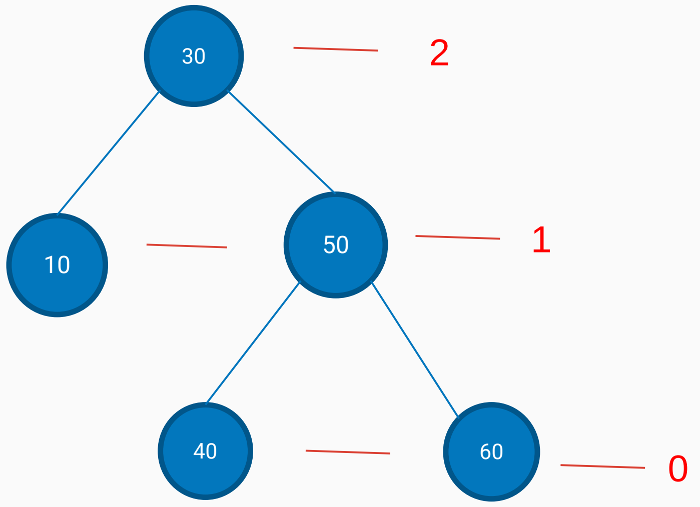


Fig. Balanced AVL Tree

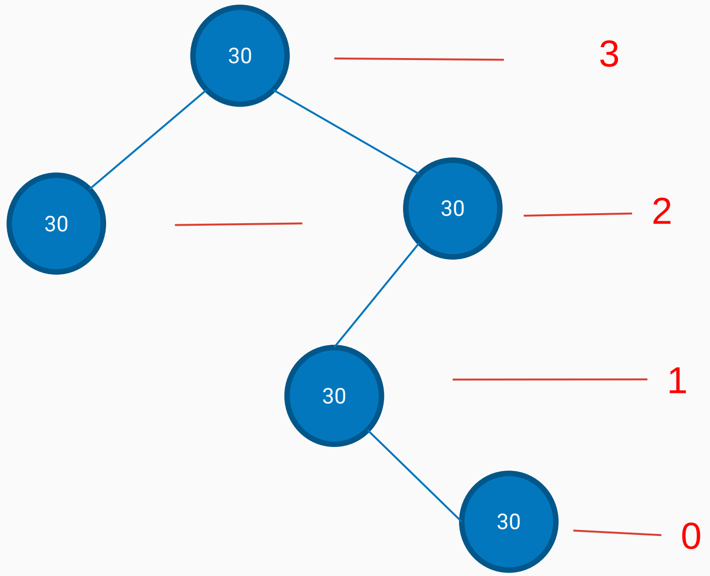
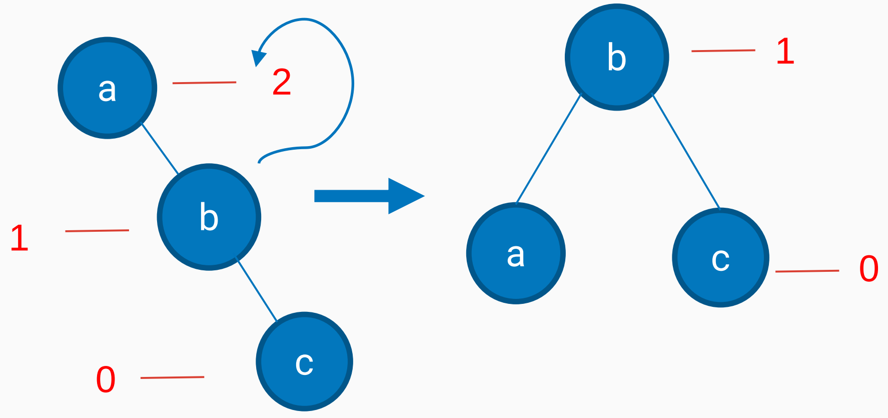


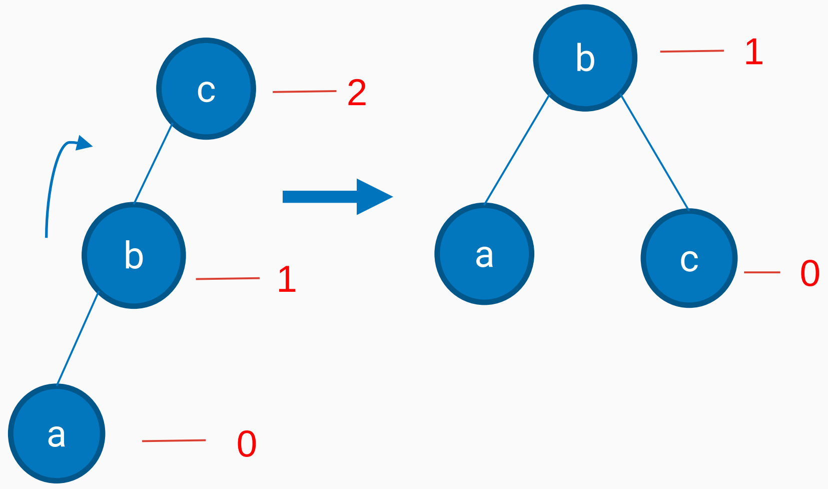
Fig. Unbalanced AVL Tree

## Method to balance AVL Tree

Single Left Rotation

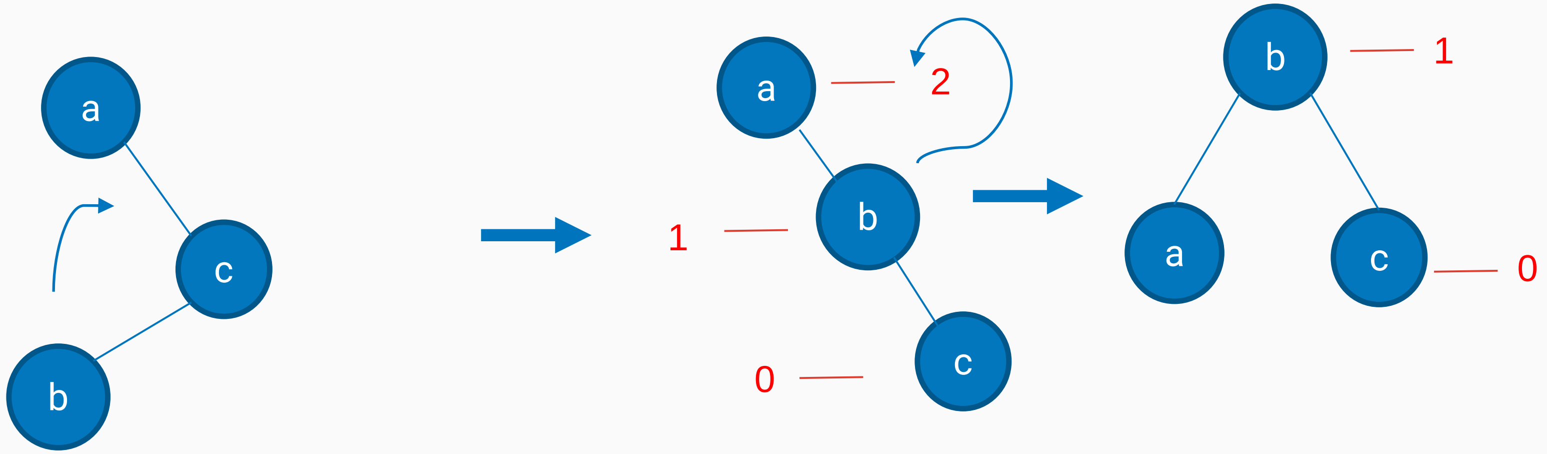


Single Right Rotation



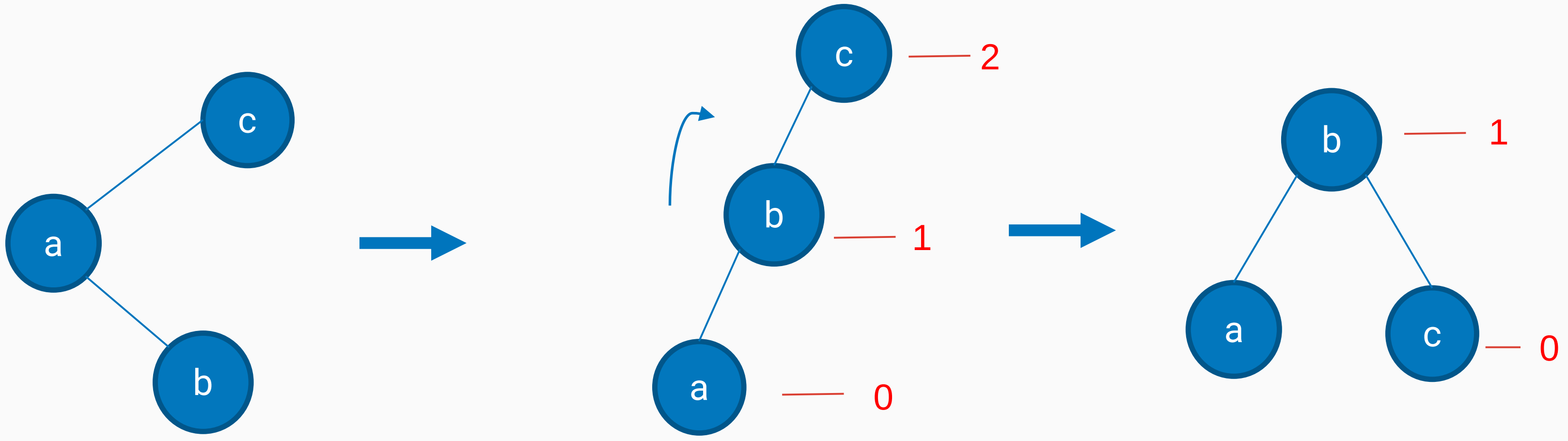
## Method to balance AVL Tree

Right Left Rotation (Double Rotation)



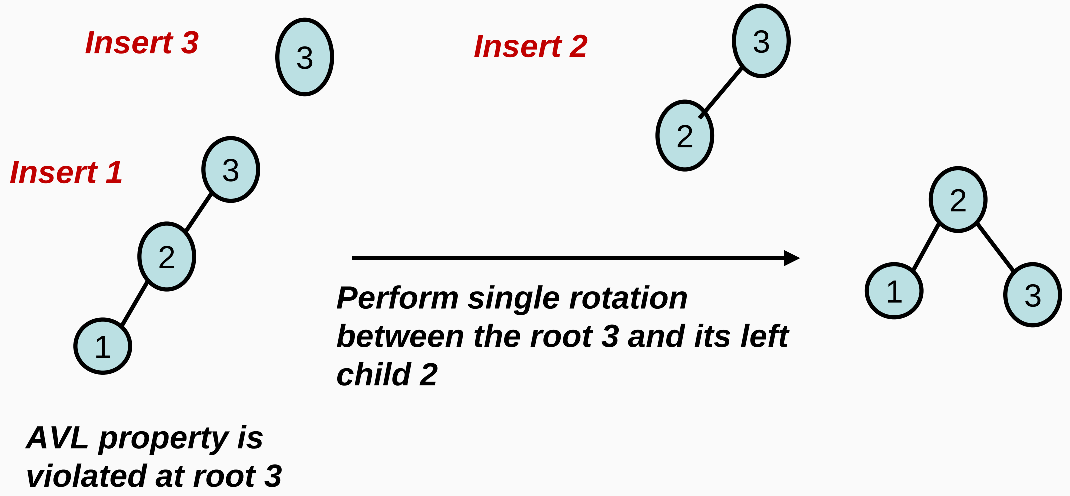
## Method to balance AVL Tree

Left Right Rotation (Double Rotation)



# Insertion in AVL Tree

**Tracing of Single Rotation:**
Insert 3, 2, 1, 4, 5, 6, and 7 in an empty AVL tree.

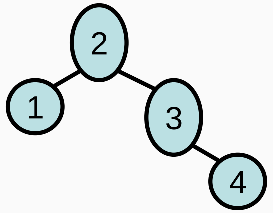


*Insert 3*
3
*Insert 2*
3
2
*Insert 1*
3
2
1
*Perform single rotation between the root 3 and its left child 2*
2
1
3
*AVL property is violated at root 3*

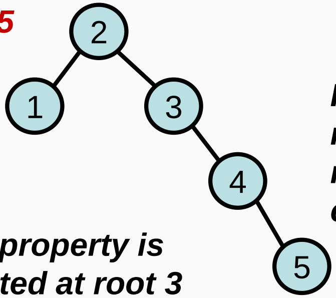
## Insertion in AVL Tree

**Tracing of Single Rotation:**
Insert 3, 2, 1, 4, 5, 6, and 7 in an empty AVL tree.

*Insert 4*

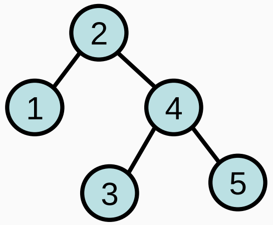


*Insert 5*



***AVL property is violated at root 3***

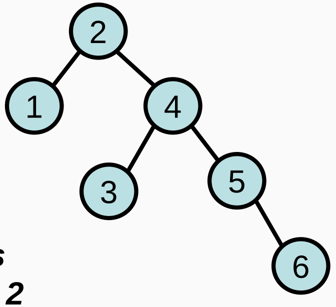
***Perform single rotation between the root 3 and its right child 4***



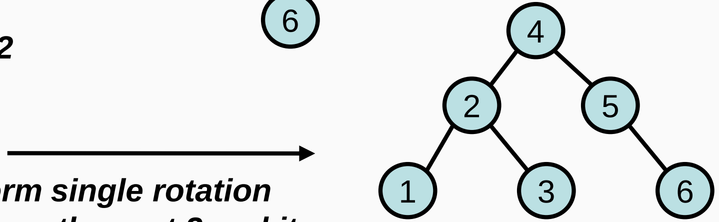
## Insertion in AVL Tree

**Tracing of Single Rotation:**
Insert 3, 2, 1, 4, 5, 6, and 7 in an empty AVL tree.

*Insert 6*



*AVL property is violated at root 2*

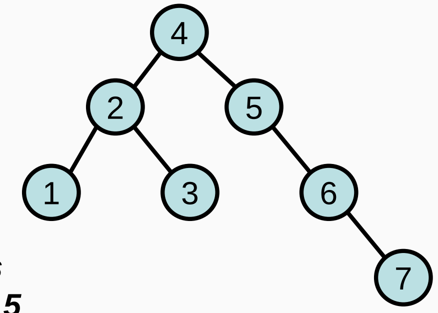


*Perform single rotation between the root 2 and its right child 4*

## Insertion in AVL Tree

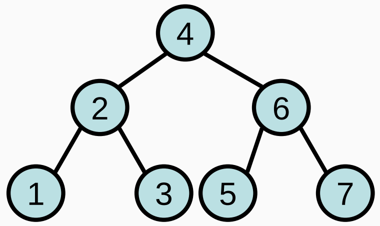
**Tracing of Single Rotation:**
Insert 3, 2, 1, 4, 5, 6, and 7 in an empty AVL tree.

*Insert 7*



***AVL property is violated at root 5***

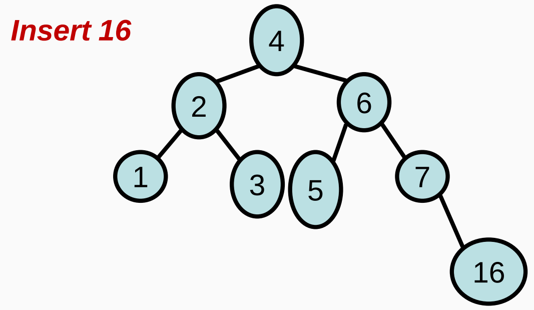
***Perform single rotation between the root 5 and its right child 6***

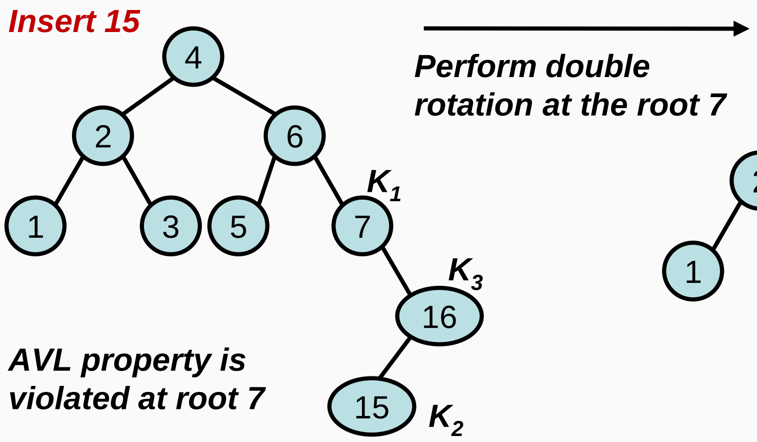


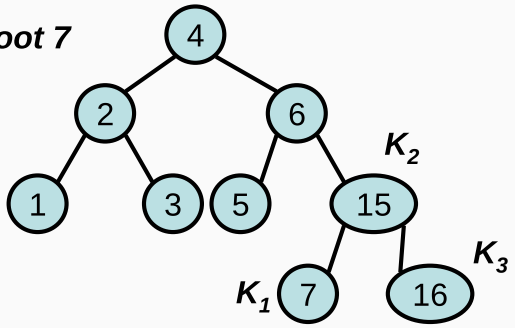
## Insertion in AVL Tree

**Tracing of Double Rotation:**

Continue inserting 16,15,14,13,12 in the previous tree



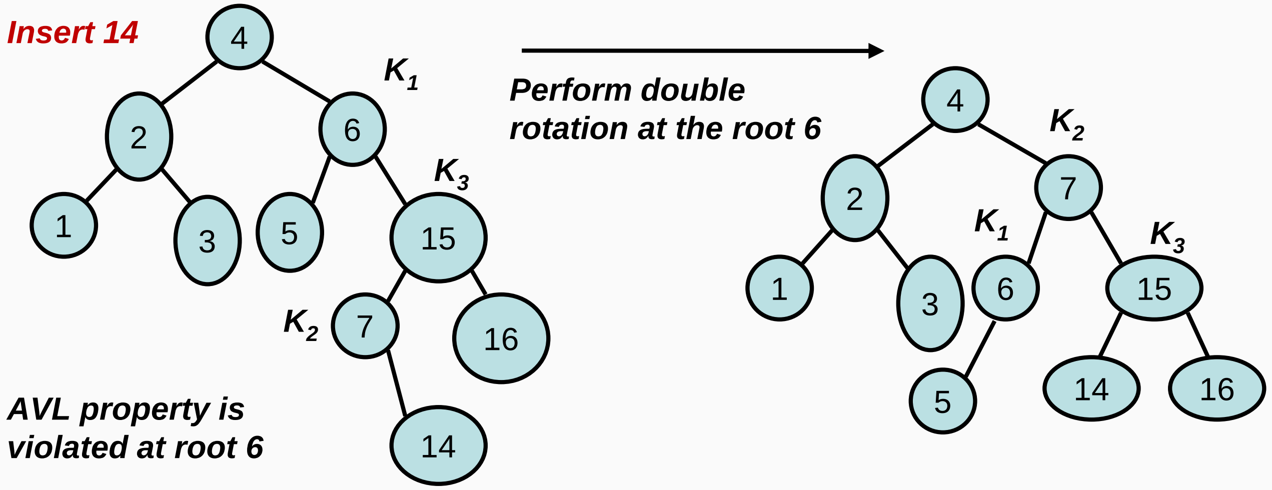

Insert 15
4
2
6
1
3
5
7
16
15
K1
K3
K2
AVL property is violated at root 7
Perform double rotation at the root 7



## Insertion in AVL Tree

**Tracing of Double Rotation:**

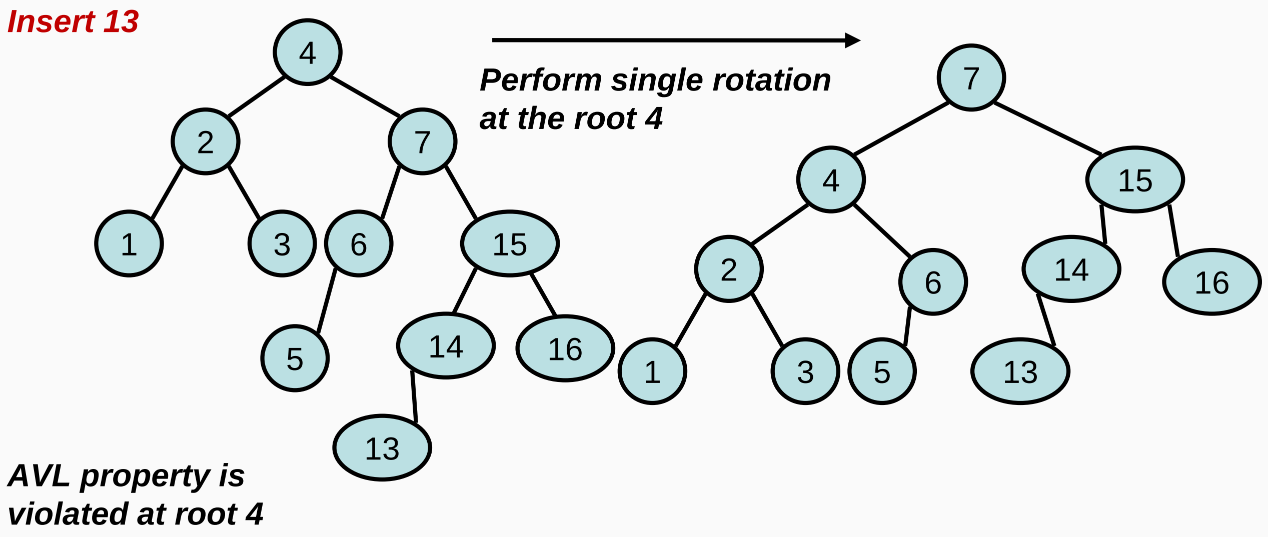
Continue inserting 16,15,14,13,12 in the previous tree



Insert 14
4
2
6
1
3
5
15
7
16
14
K1
K3
K2
AVL property is violated at root 6
Perform double rotation at the root 6
4
2
7
1
3
6
15
5
14
16
K2
K1
K3

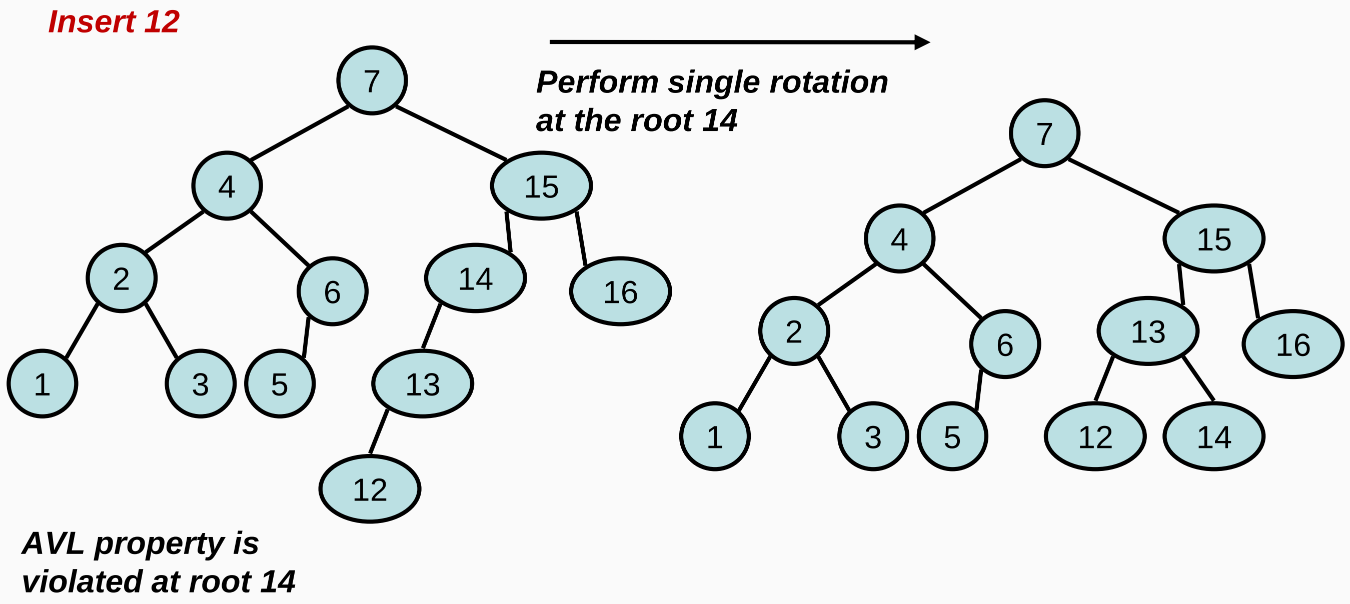
## Insertion in AVL Tree

**Tracing of Double Rotation:**
Continue inserting 16,15,14,13,12 in the previous tree


*Insert 13*
4
2
7
1
3
6
15
5
14
16
13
*AVL property is violated at root 4*
*Perform single rotation at the root 4*
7
4
15
2
6
14
16
1
3
5
13

## Insertion in AVL Tree

**Tracing of Double Rotation:**
Continue inserting 16,15,14,13,12 in the previous tree



*Insert 12*
*Perform single rotation at the root 14*
*AVL property is violated at root 14*
7
4
15
2
6
14
16
1
3
5
13
12
7
4
15
2
6
13
16
1
3
5
12
14

Construct an AVL tree from the given data

35, 56, 64, 68, 65, 44, 31, 49, 45, 20, 25.

Also, explain the methods of balancing the AVL tree.

## Deletion in AVL Tree

- Like insertions, deletion of nodes in AVL Tree is also similar to deletion in BST. After deletion of nodes, we have to check for imbalance and correct it to rebalance the tree.

**Efficiency of AVL Tree:**

- The maximum height of an AVL tree is 1.44 * log n, which is an O(log n) function

✓ This means that in the worst possible case, a lookup in a large AVL tree needs no more than 44% more comparisons than a lookup in a completely balanced tree

- Even in the worst case, then, AVL trees are efficient; they still have O(log n) lookup times
- On average, for large n, AVL trees have lookup times of (log n) + 0.25, which is even better than the above worst case figure, though still O(log n)
- On average, a rotation (single or double) is required in 46.5% of insertions. Only one (single or double) rotation is needed to readjust an AVL tree after an insertion throws it out of balance

## Game Tree

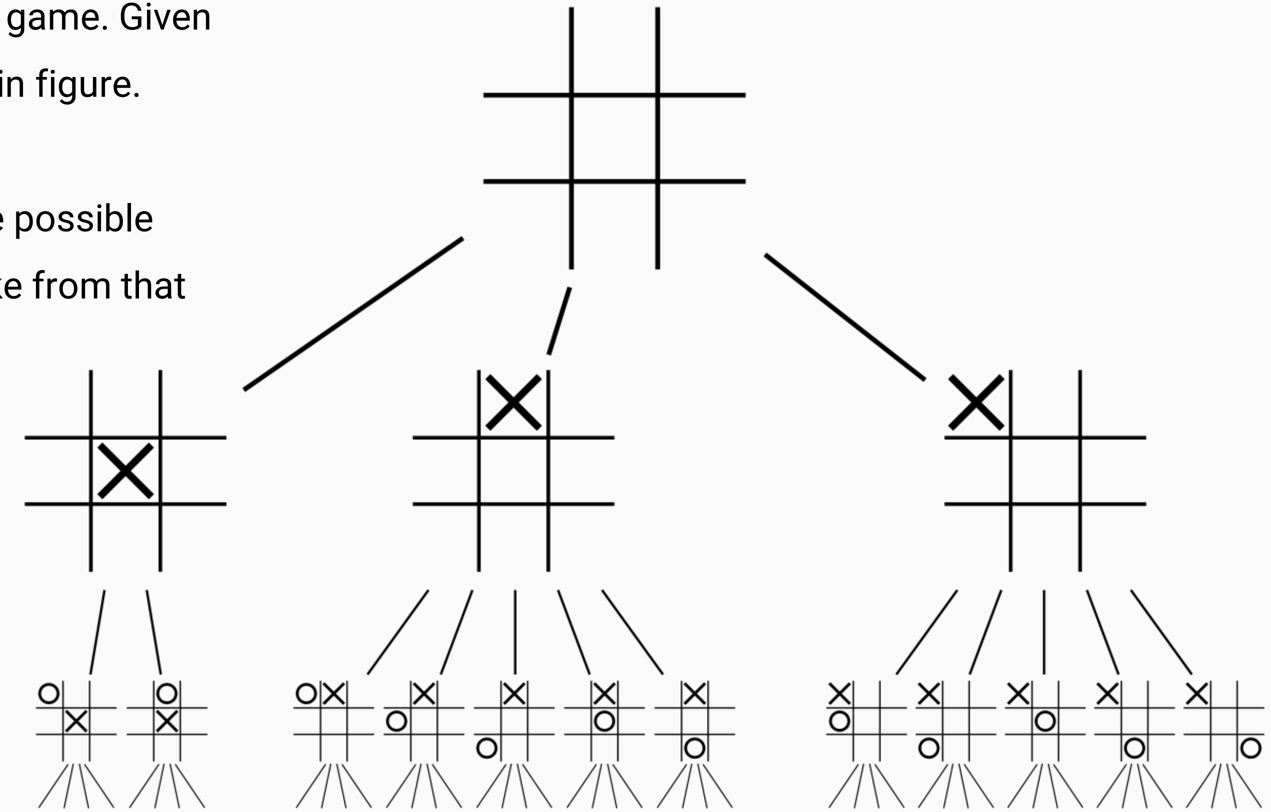
## Game Tree

- A game tree is a type of recursive search function that examine all the possible moves of a strategy game, and their results in an attempt to ascertain the optimal move.
- Such games include well-known ones such as chess, checkers, Go, and tic-tac-toe. This can be used to measure the complexity of a game, as it represents all the possible ways a game can pan out.
- It is possible to look ahead several moves, we define a look ahead level as number of future moves to be considered.
- Game tree is popular for determining the best move for a player during a game.

## Game Tree

- Let us consider a Tic-Tac-Toe game. Given a board position at as shown in figure.

- The figure illustrates the three possible moves, that cross(X) can make from that position.



## The Huffman Algorithm

## The Huffman Algorithm

- Lots of compression systems use Huffman Algorithm for the compression of text, images, audio, data etc. The general idea behind the Huffman algorithm is that, use less number of bits to represent mostly used key.
- The output from Huffman's algorithm can be viewed as a variable-length code table for encoding a source symbol (such as a character in a file).
- The algorithm derives this table from the estimated probability or frequency of occurrence (weight) for each possible value of the source symbol.
- It is commonly used for lossless data compression.

## The Huffman Algorithm

1. Find frequencies of each symbol
2. Begin with a single node trees

✓ each contain symbol and its frequency

3. Do recursively

✓ select two trees with smallest frequency at the root

✓ produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root

4. Recursion ends when there is one tree

✓ this is the Huffman coding tree

## Example

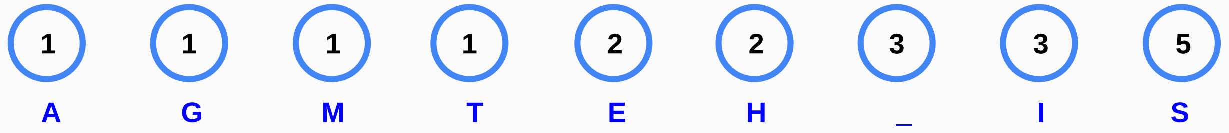
- Build the Huffman coding tree for the message

  ***This is his message***

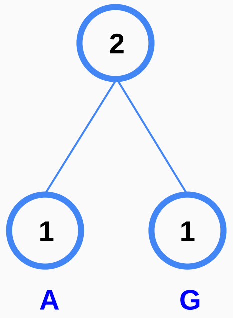
- Character frequencies

| A | G | M | T | E | H | _ | I | S |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 2 | 2 | 3 | 3 | 5 |

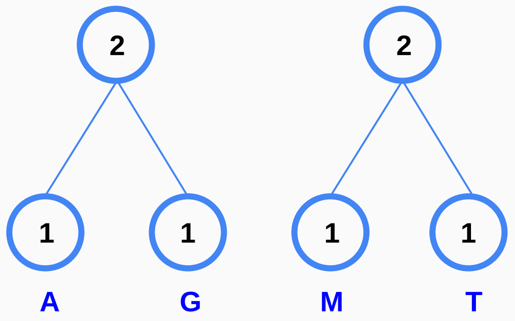
- Begin with single nodes trees



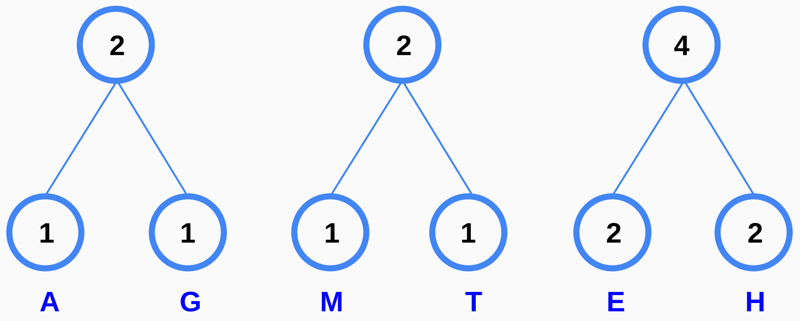
## Step 1



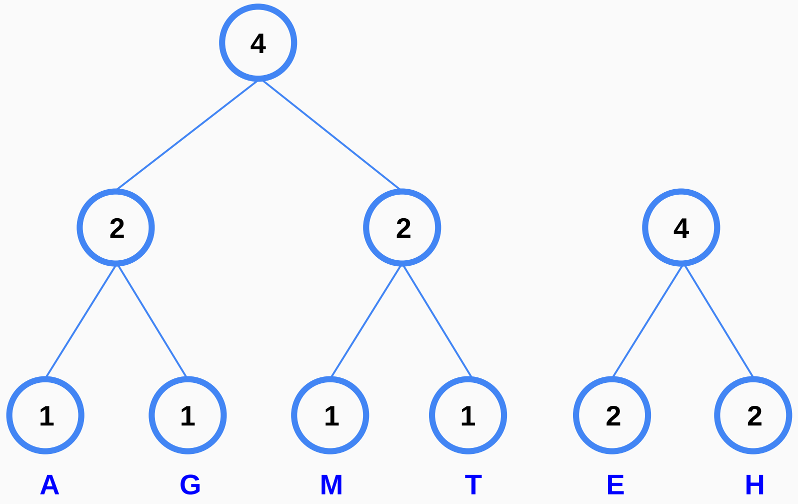
## Step 2



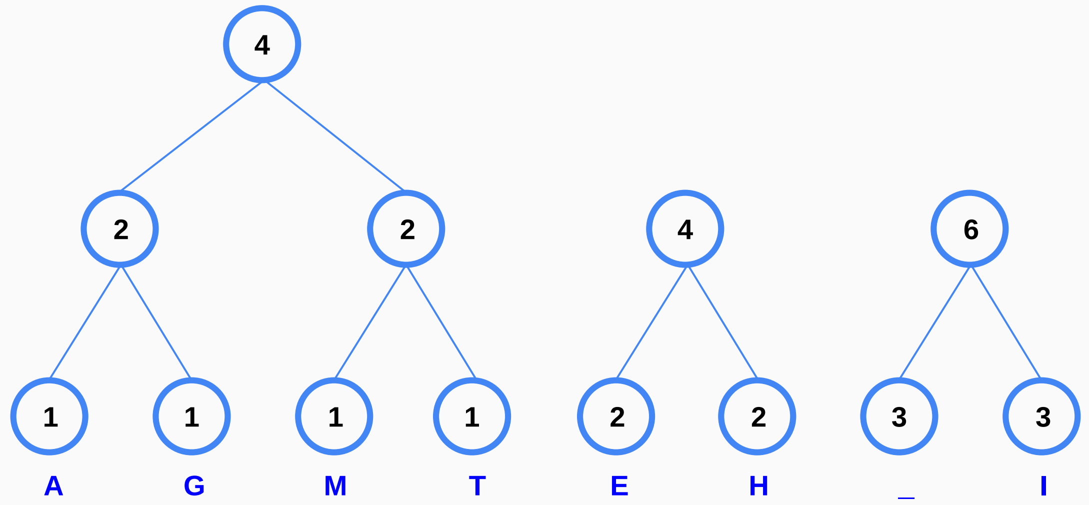
## Step 3



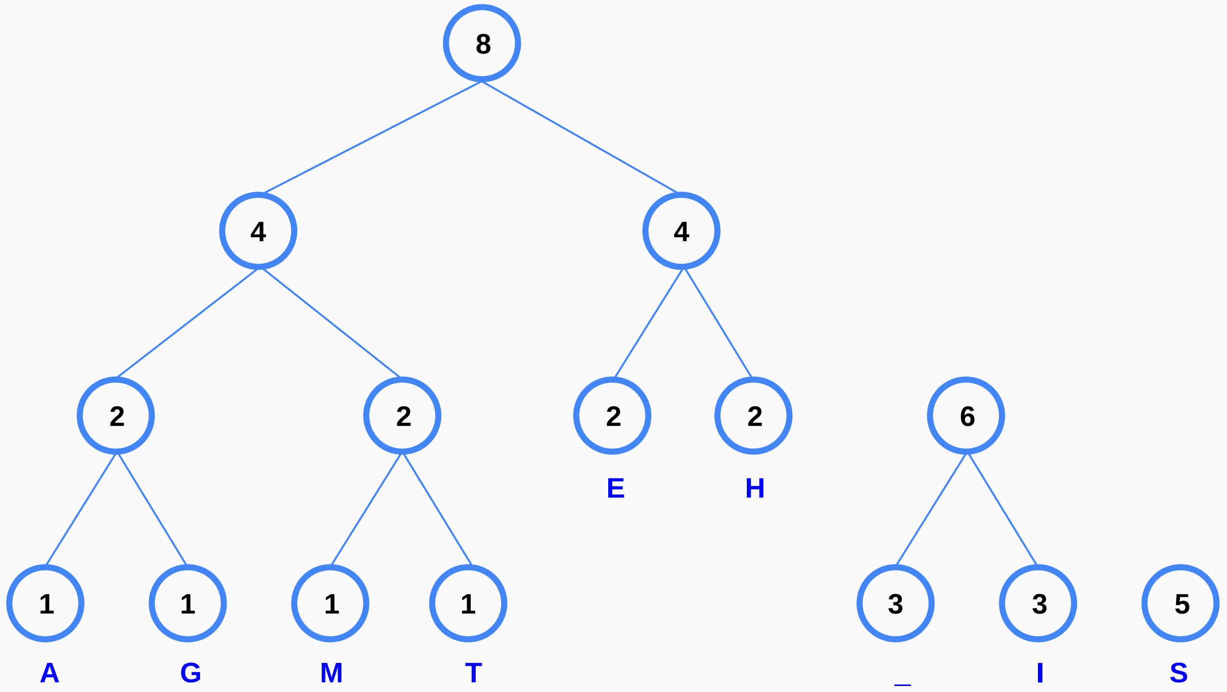
## Step 4



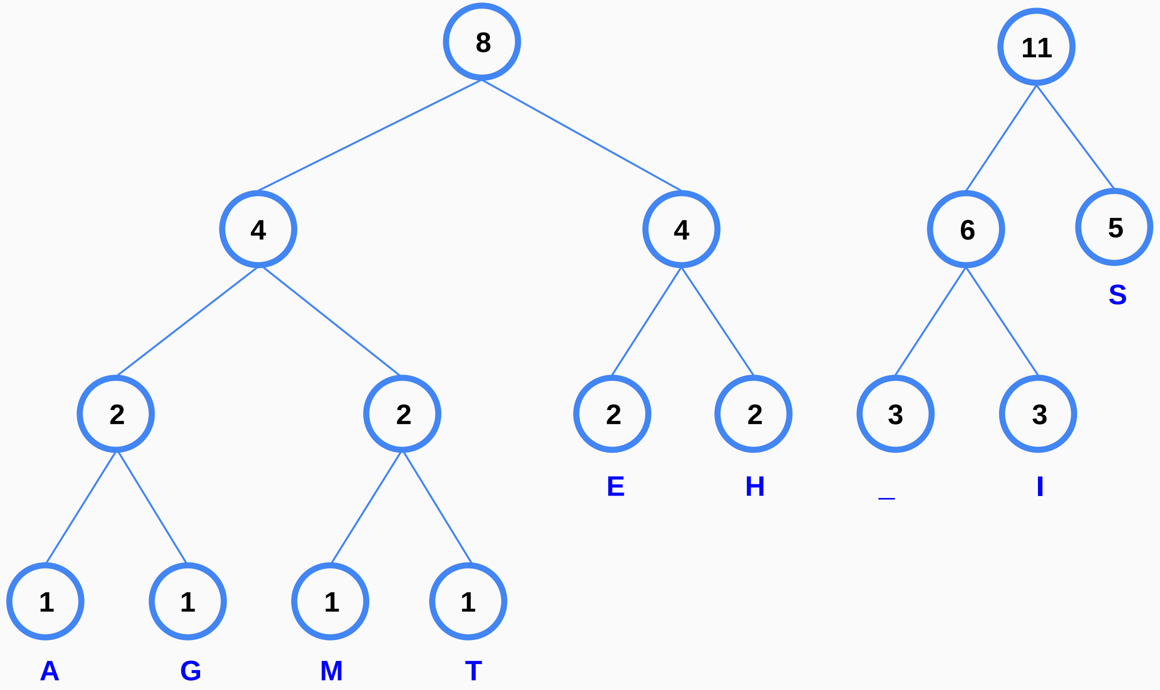
## Step 5



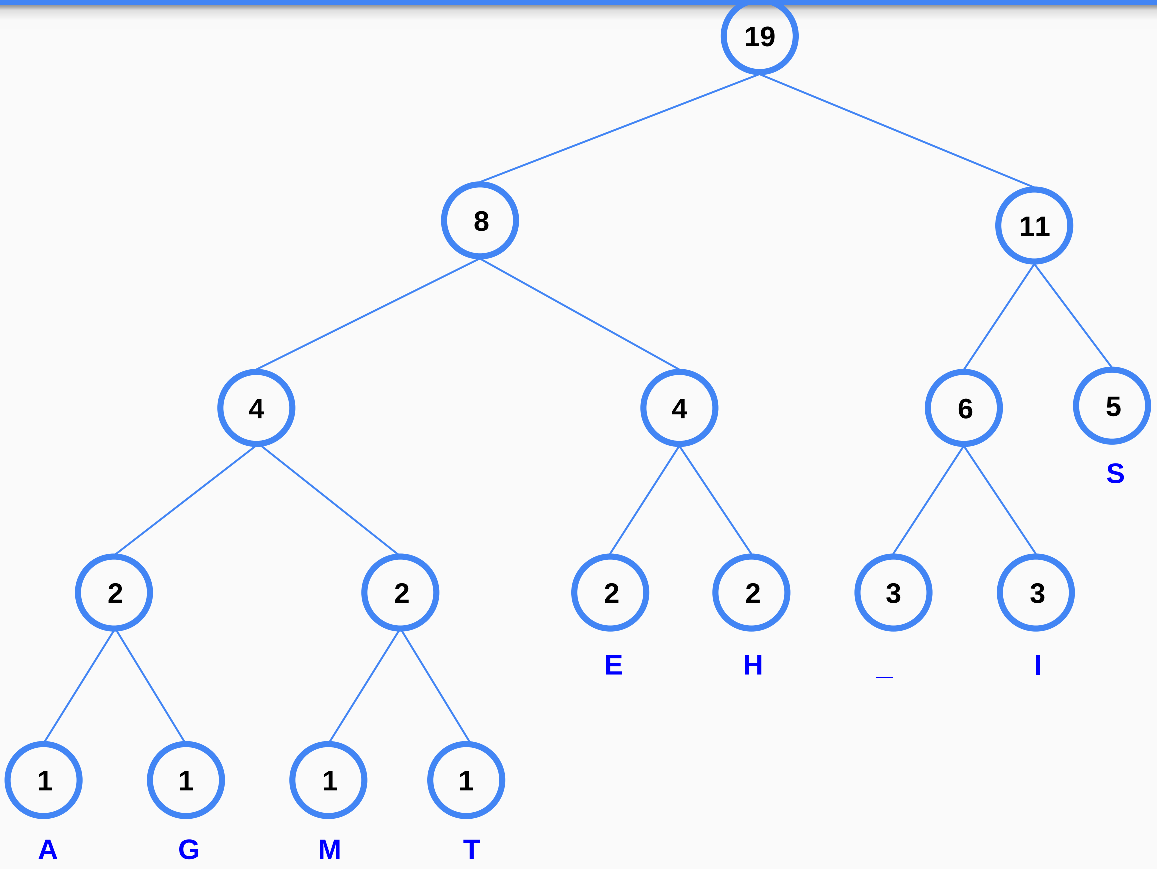
## Step 6



## Step 7

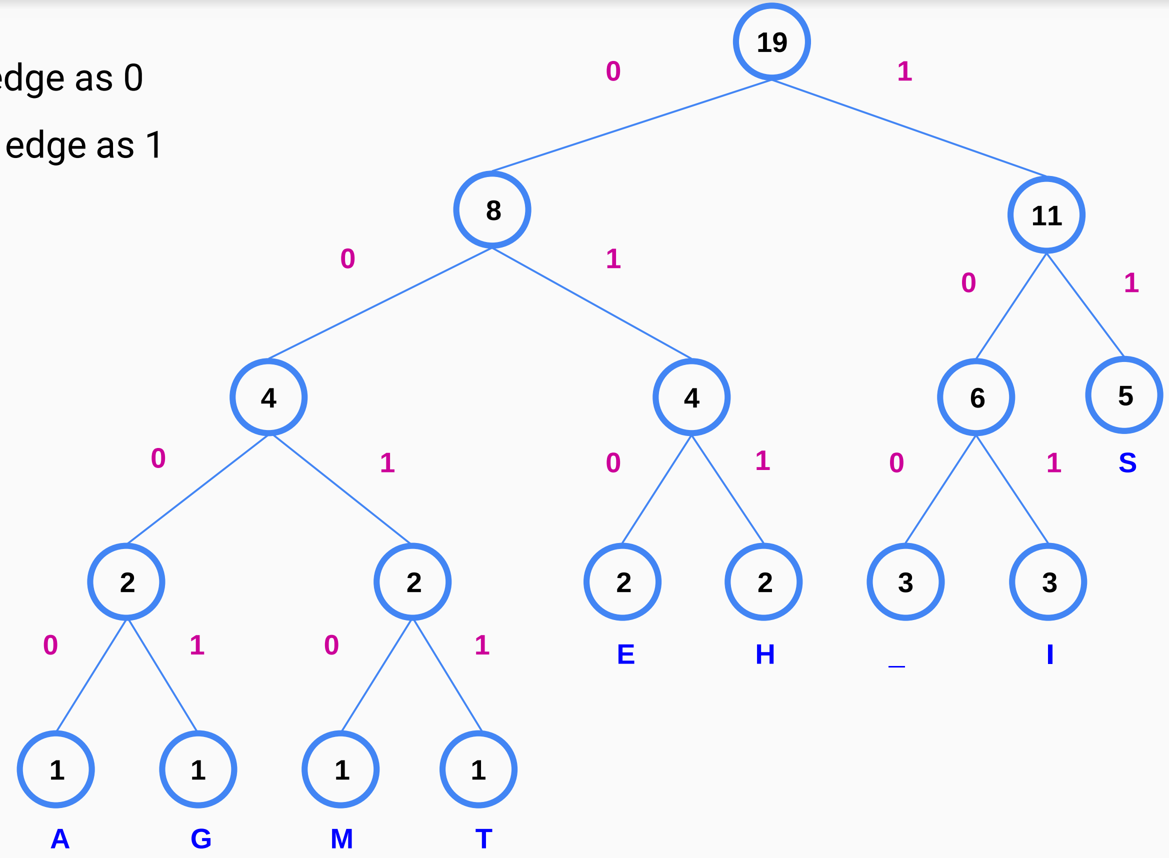


## Step 8



## Label edges

- Assign left edge as 0
- Assign right edge as 1



# Huffman code & encoded message

*This is his message*

| | |
|---|---|
| S | 11 |
| E | 010 |
| H | 011 |
| _ | 100 |
| I | 101 |
| A | 0000 |
| G | 0001 |
| M | 0010 |
| T | 0011 |

**00110111011110010111100011101111000010010111100000001010**

Construct Huffman tree and find Huffman code for the symbol a, b, c, d, e with probabilities 0.12, 0.40, 0.15, 0.08 and 0.25 respectively.

Construct Huffman tree and find Huffman code for the symbol

A, B, C, D, E with frequencies 24, 12, 10, 8 and 8 respectively.

## B-Tree

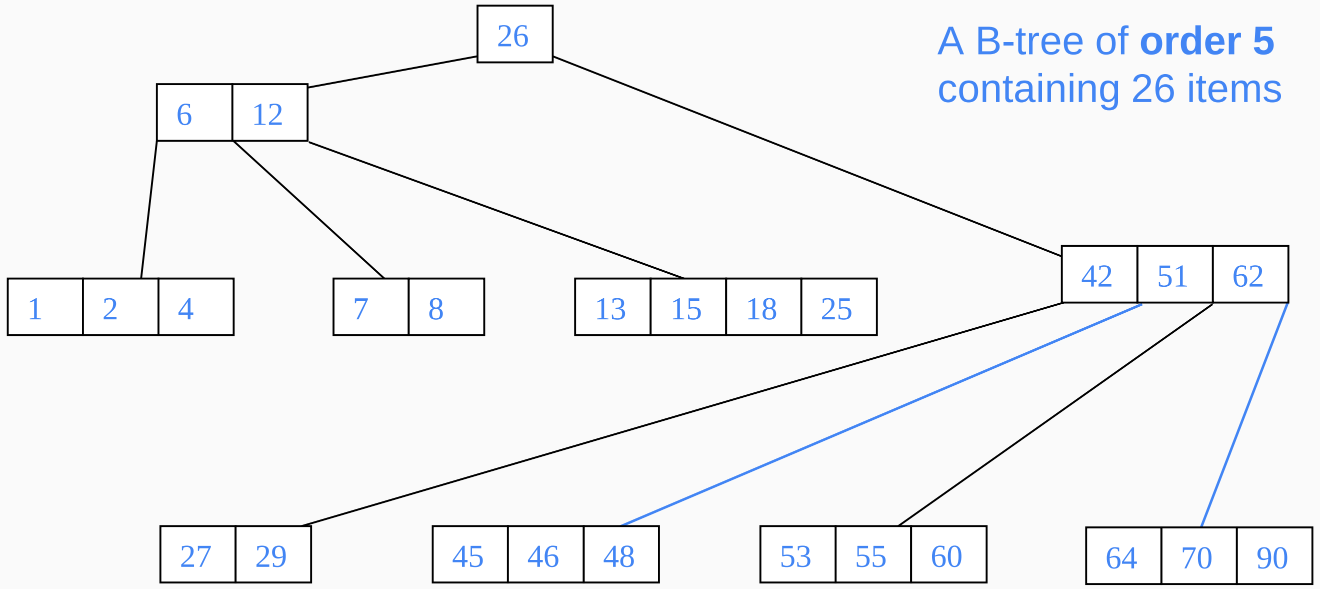
## B-Tree

A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:

1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
2. all leaves are on the same level
3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
4. the root is either a leaf node, or it has from two to *m* children
5. a leaf node contains no more than *m* – 1 keys

   The number *m* should always be odd

## An example B-Tree



A B-tree of **order 5**
containing 26 items

*Note that all the leaves are at the same level*

## Inserting into a B-Tree

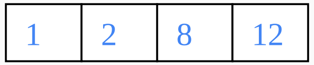
- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

## Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:

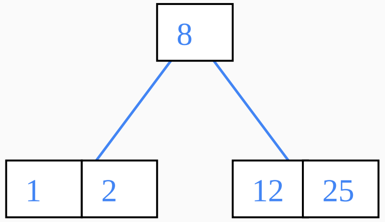
  **1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45**

- We want to construct a B-tree of order 5.
- The first four items go into the root:

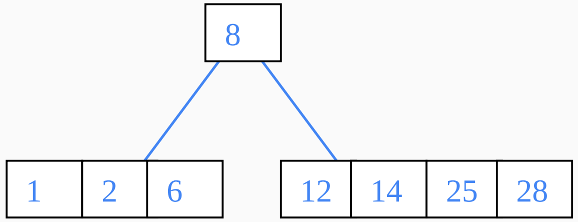


- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

## Constructing a B-tree (contd.)

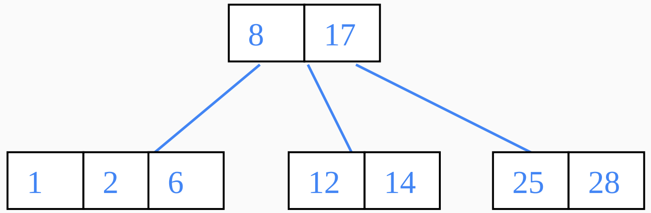


6, 14, 28 get added to the leaf nodes:

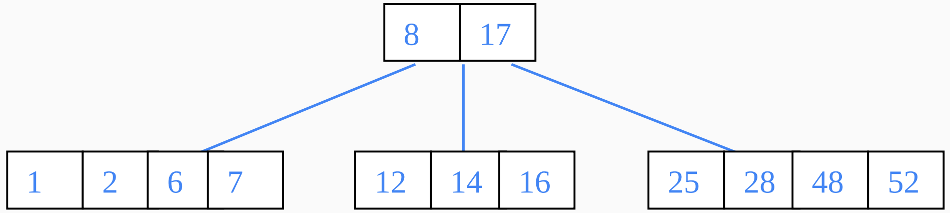


## Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

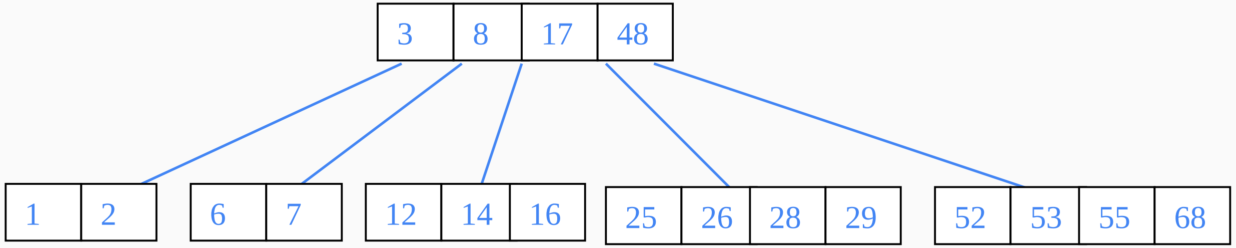


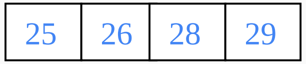
7, 52, 16, 48 get added to the leaf nodes



## Constructing a B-tree (contd.)

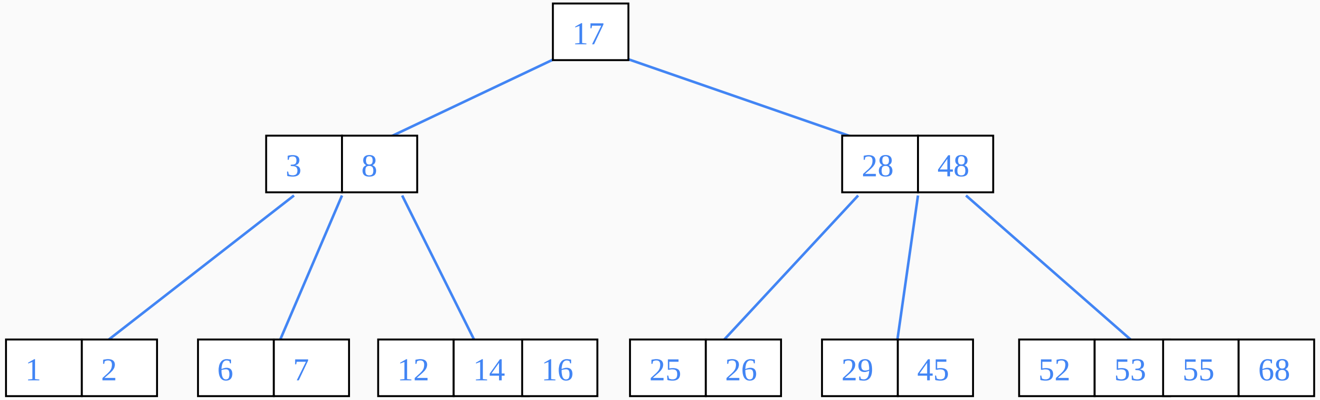
Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves



Adding 45 causes a split of 

and promoting 28 to the root then causes the root to split

# Constructing a B-tree (contd.)



Insert the following data in B-tree of order 5.

10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 240

Insert the following data from 1 to 10 in B-tree of order 3.

## Removal from a B-tree

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:

1. If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
2. If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

## Removal from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - 4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

## Applications of Tree Data Structure

- To manipulate hierarchical data.
- Used in many search applications where data is entering/leaving such as map and set objects in many languages libraries.
- Huffman coding tree is used in compression algorithms such as those used by the .jpeg and .mp3 file formats.
- To manipulate sorted lists of data.

THANK YOU

# Any Queries ?

ankit.bca@kathford.edu.np