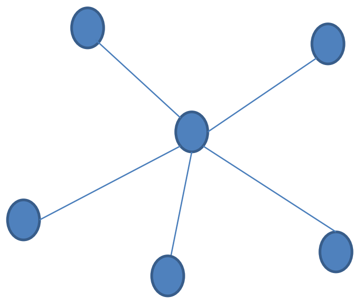
TREE

INTRODUCTION

❑ A tree is non-linear data structure used to store information.

❑ It is a hierarchical data structure which differentiates from stacks & queues (linear data structures).

❑ A tree is connected, acyclic graph.



A connected / acyclic graph [TREE]

TREE

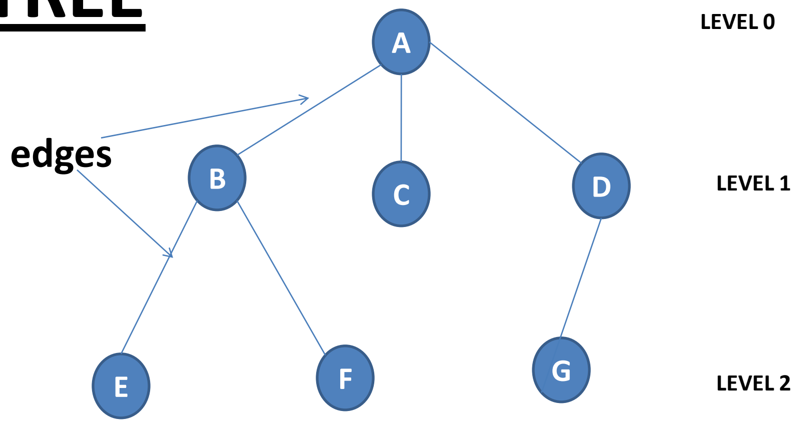
INTRODUCTION

❑ A tree is so connected that any node in the graph can be reached from any other node by exactly one path.

❑ A tree does not contain any closed path.

❑ Consider the following ROOTED TREE (i.e. A tree which has only single root node which has no parents).

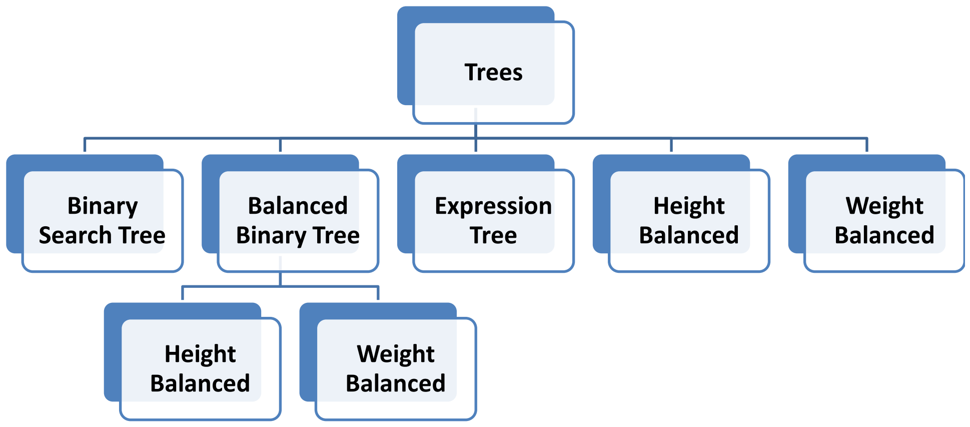
TREE



HERE, A is Root Node [Which has no Parent Node]. B, D are known as Interior Nodes and C, E, F, G are known as Leaf nodes

TREE

DIFFERENT CATEGORIES OF TREE



<u>TREE</u>

<u>BASIC TERMINOLOGIES OF TREE:</u>

- ❑ **ROOT :** This is the one of the main component of tree. A node without parent is known as root. In diagram, A is root node.
- ❑ **BRANCH :** This is also one of the main component of tree. Branching factor defines the maximum number of children to any node. So, if a branching factor of 2 means Binary Tree.
- ❑ **Edge :** A finite collection of directed arcs that connect pair of nodes. In short, link between two nodes.
- ❑ The direction from root to leaves is '**DOWN**' and the opposite direction is '**UP**'.
- ❑ **CLIMBING TREE** means going from leaves to the root.
- ❑ **DESCENDING TREE** means going from root to the leaves.
- ❑ **NULL TREE** means tree with no nodes.

<u>TREE</u>

<u>BASIC TERMINOLOGIES OF TREE:</u>

❑ **LEAF NODE:** A node that has no children is called a leaf node. It is also known as <u>terminal nodes</u> and rest of the nodes are known as <u>non-terminal nodes</u>.

❑**INTERIOR NODE:** Nodes which possess children are called the interior nodes.

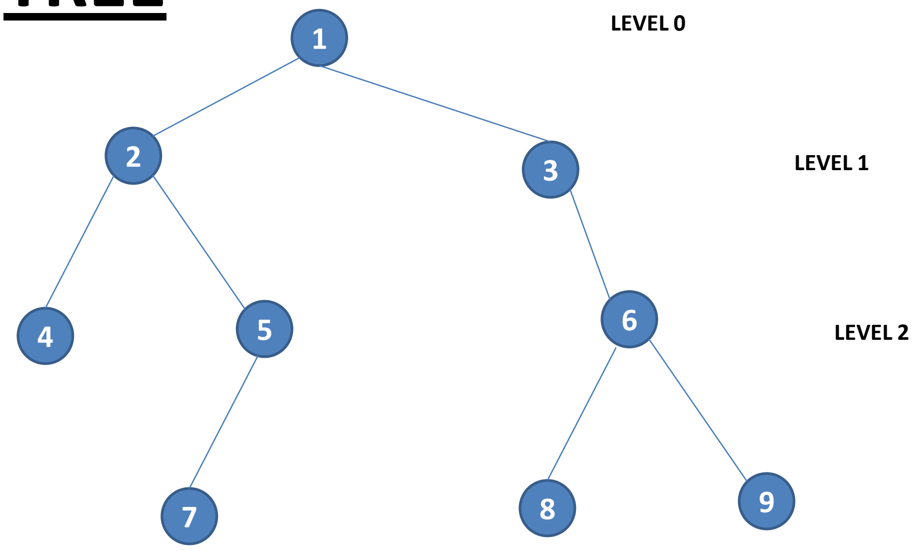
❑**SIBLINGS :** All the nodes of the same parent node are called siblings (Brother).

❑**DEGREE OF NODE :** The number of the sub-trees of a node is called the degree of the node. A node with degree 0 is called leaf node.

❑ **DEPTH OR HEIGHT OF TREE :** Length of the longest path from root to any node is known as depth or height of the tree.

❑ The nodes which are at the same level are said to be of the **same generation**.

TREE



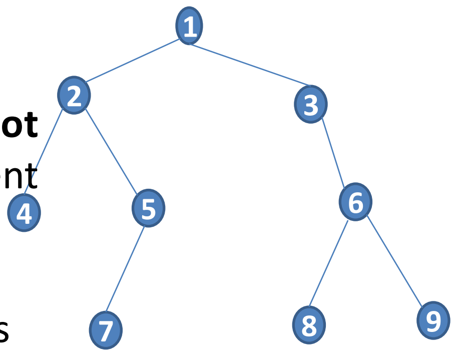
TREE

✓ In this diagram, 1 is **root node** as it has no parent node.

✓ 4, 7, 8, 9 have no branches which is known as **Leaf Nodes.**

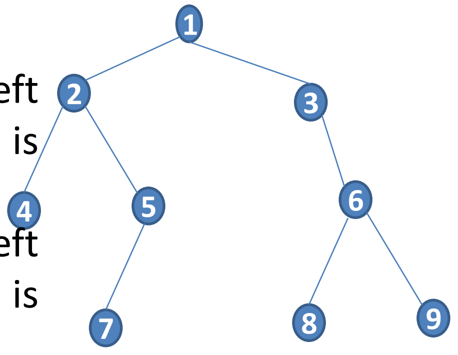
✓ 4 and 5 are children of 2 while 2 is parent of 4 and 5.

✓ As 4 and 5 has the common parent 2, they are known as **Siblings.**



✓ The nodes 1, 2, 3, 5, 6 have a branch to other nodes, so they are known as **Non-Leaf Nodes.**

TREE



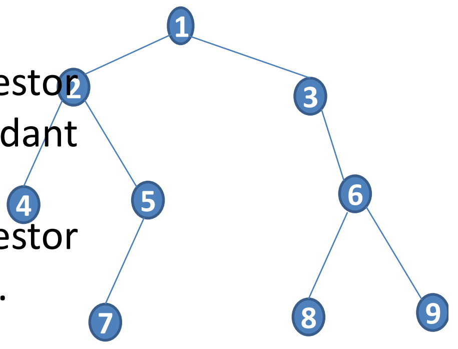
✓ In this diagram, 2 is left sub-tree of root 1 and 3 is right sub-tree of root 1.

✓ Similarly, node 4 is left sub-tree of node 2 and 5 is right sub-tree of node 2.

✓ Every node which is parent to leaf and non-leaf node is known as **ANCESTOR.**

✓ All the nodes you can reach from a given node are known as **DESCENDANT.**

TREE



✓ In this diagram, 1 is ancestor of node 5 and 8 is descendant of node 3.

✓ Node 5 is neither an ancestor nor a descendant of node 3.

✓ In this diagram, 1 is @ depth / height 0, 2 and 3 is @ depth / height 1.

✓ Similarly, 4, 5, 6 are @ depth / height 2 and so on.

BINARY TREE - DEFINITION

There are different ways to define binary tree. Among these some of as are under:

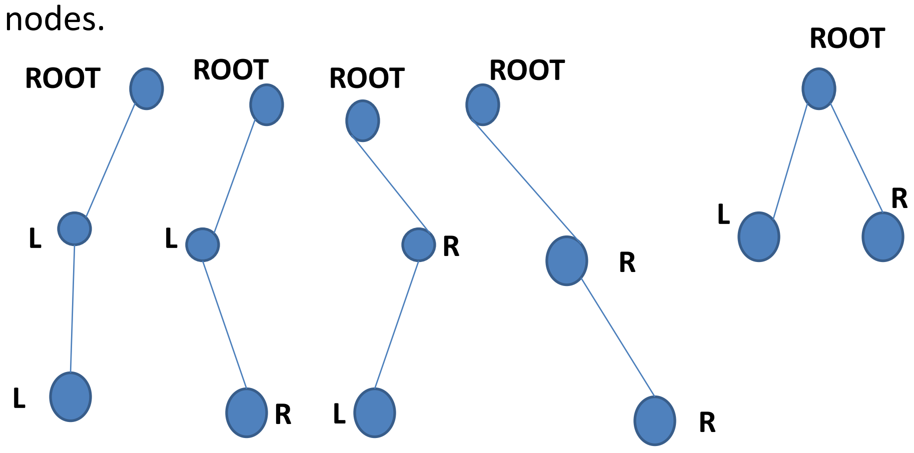
❑ Every nodes in a tree has a maximum of two children are known as Binary Tree. OR We can say that every node of a tree can have at most degree 2 is known as binary tree.

❑ A binary tree is a finite set of elements that is either empty or divides into three disjoint subsets. The first subset contains a single element called a root, and the other two subsets are left and right sub-trees.

❑ A tree in which every root has either one or two nodes is called binary tree. The Left part of root is known as left sub-tree and right part of root is known as right sub-tree.

BINARY TREE - DEFINITION

Following figure shows possible binary trees with three nodes.



Generate Binary Tree for
10,3,20,2,7,5,15,31

The binary tree creation follows a simple principle to add new node to the tree.

➔ It compares the new value with the current element of the tree.

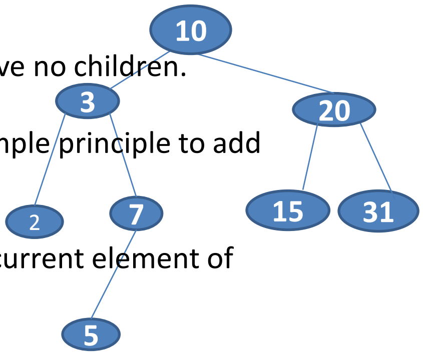
➔ if its value is less than the current element in the tree then it will move towards the left side of the tree.

➔ And if its value is greater than the current element in the tree then it will move towards the right side of the tree.

BINARY TREE - CONCEPT

❑ In this diagram, 10 is root node as it does not have any parent node.

❑ 2, 5, 15, 31 are leaf nodes as they have no children.



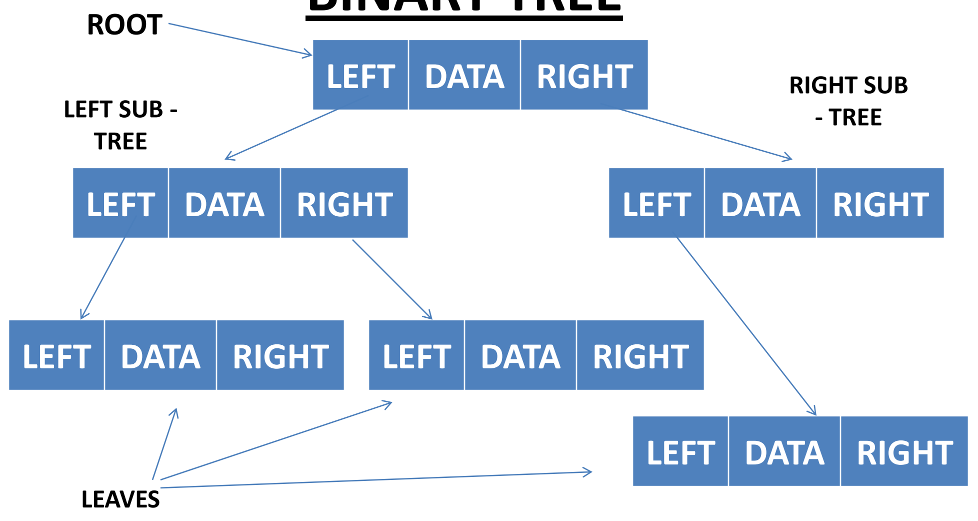
❑ The binary tree creation follows a simple principle to add new node to the tree.

❑ It compares the new value with the current element of the tree.

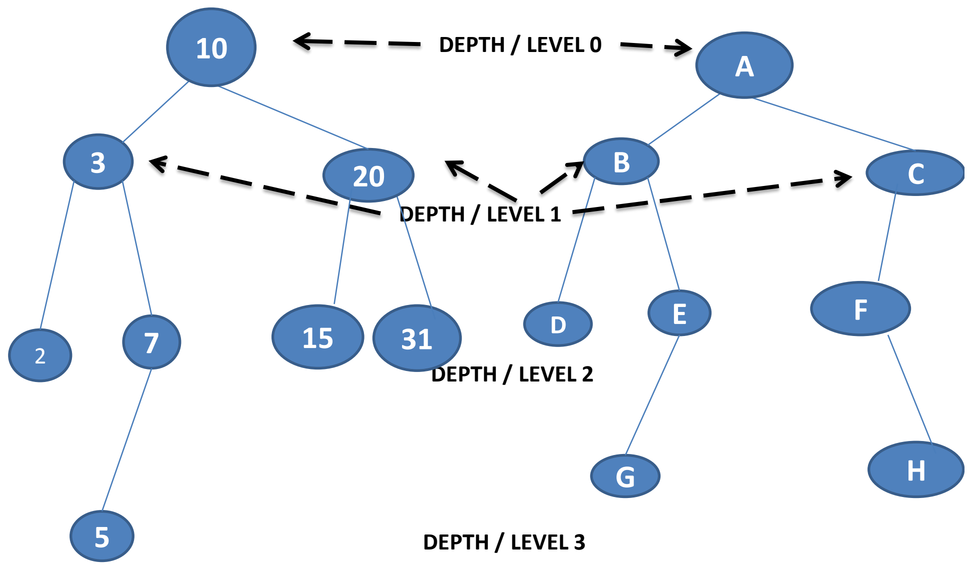
❑ if its value is less than the current element in the tree then it will move towards the left side of the tree.

❑ And if its value is greater than the current element in the tree then it will move towards the right side of the tree.

LINKED LIST REPRESENTATION OF BINARY TREE



DEPTH / LEVEL OF BINARY TREE



DEPENDING ON THE DEPTH / LEVEL OF BINARY TREE, IT CAN BE CLASSIFIED AS:

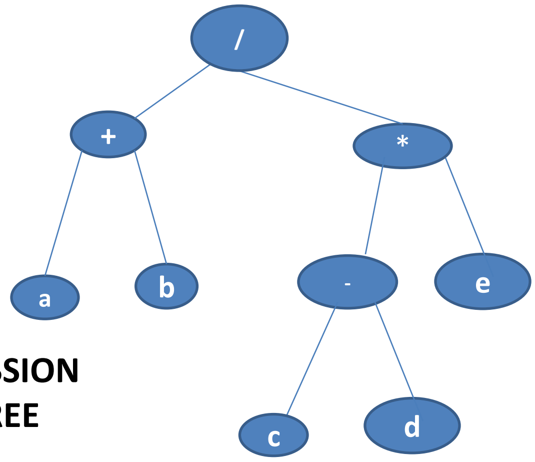
1. STRICTLY BINARY TREE:

- If every non-leaf node in a binary tree has exactly two non-empty children (left and right sub tree) then it is known as Strictly Binary Tree.
- That means, strictly binary tree with n leaves will always have 2n-1 nodes.
- Every node in this type of tree can have either no child or two children.
- Strictly Binary Tree also known as **2-Tree or Extended Binary Tree**.
- These types of trees are generally used to represent any expression with binary operation. [And so it is also said to be **EXPRESSION TREE**]

DEPENDING ON THE DEPTH / LEVEL OF BINARY TREE, IT CAN BE CLASSIFIED AS:

Consider Expression

E = (a + b) / ((c – d) * e)

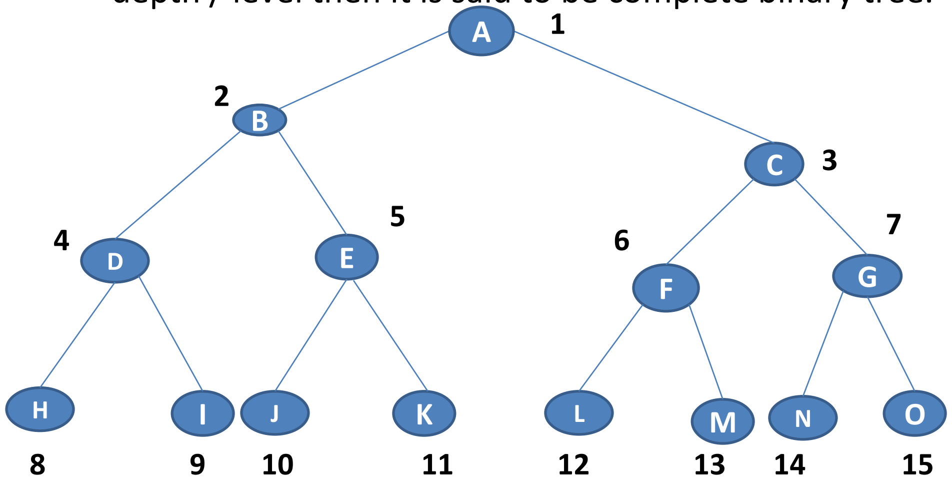


EXPRESSION TREE

DEPENDING ON THE DEPTH / LEVEL OF BINARY TREE, IT CAN BE CLASSIFIED AS:

2. COMPLETE BINARY TREE:

- If all the leaves of strictly binary tree are at the same depth / level then it is said to be complete binary tree.



DEPENDING ON THE DEPTH / LEVEL OF BINARY TREE, IT CAN BE CLASSIFIED AS:

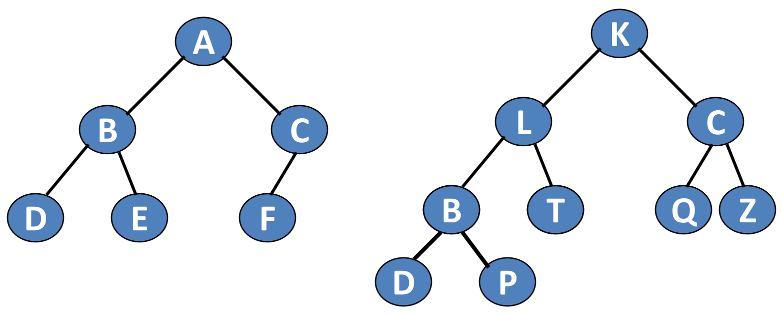
2. COMPLETE BINARY TREE:

- The main advantage of this type of tree is we can easily search the left and right sub tree of any node.
- In this type, if we want to find out the parent of particular child then any node N will be at floor(N/2).
- For example, if want to find parent of node O then floor(15/2) = 7, i.e. G.
- Similarly, if we want to find out left and right sub-tree of any Node then evaluate 2N and 2N + 1 respectively.
- For Example, Left sub-tree of Node G is N [2N = 2*7 = 14) and Right sub-tree of Node G is O [2N + 1= 2*7 + 1= 15]

DEPENDING ON THE DEPTH / LEVEL OF BINARY TREE, IT CAN BE CLASSIFIED AS:

3. ALMOST COMPLETE BINARY TREE:

- A binary tree of depth / level 'd' is known as almost complete binary tree if each node in the tree is either at level d or d – 1.



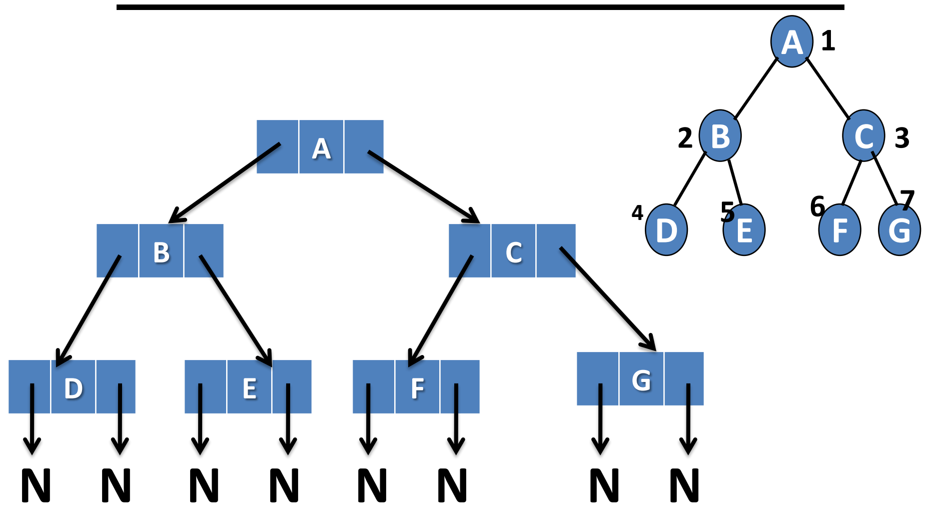
LINKED LIST REPRESENTATION

POINTER TO LEFT SUB-TREE (LEFT CHILD NODE)	INFORMATION / DATA PART	POINTER TO RIGHT SUB-TREE (RIGHT CHILD NODE)

As above mentioned in diagram we have three members for structure while working with linked list representation of trees. [Left Pointer – DATA – Right Pointer]

```
struct node
{
        int data;
        struct node *left_tree;
        struct node *right_tree;
};
```

LINKED LIST REPRESENTATION



TREE TRAVERSAL TECHNIQUES

- ❑ Traversal methods are used to display data from the tree.
- ❑ In this techniques, each node of tree is visited only once .
- ❑ Followings are different traversal techniques:

1. Preorder Traversal (ROOT – LEFT – RIGHT)
2. Inorder Traversal (LEFT – ROOT – RIGHT)
3. Postorder Traversal (LEFT – RIGHT – ROOT)

TREE TRAVERSAL TECHNIQUES

- Preorder Traversal:
  1. Visit the root node.
  2. Traverse the left sub-tree – Recursively.
  3. Traverse the right sub-tree – Recursively.

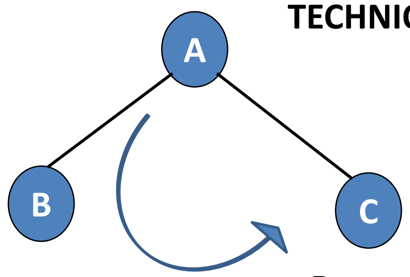
In the preorder traversal (D) is used for root node, (L) for left sub-tree and (R) is used for right sub-tree.

So, the preorder traversal is D – L – R.

TREE TRAVERSAL TECHNIQUES

D – ROOT

PRE ORDER TRAVERSAL TECHNIQUE [D – L – R]



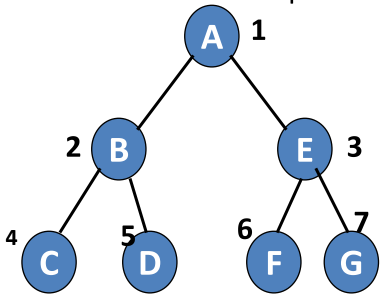
L – LEFT SUB -TREE

R – RIGHT SUB -TREE

TREE TRAVERSAL TECHNIQUES

Consider following tree. The sequence for preorder traversal is:

1) Root A is processed.
2) After that the left sub-tree of root is processed.
    2.1) To process the left sub-tree, first process the root of left sub-tree i.e. B.
    2.2) Then its left sub-tree (i.e. C) and its right sub-tree (D).
3) Now, start the processing for right sub-tree of root (i.e. E). And do the recursive process as we have done for left sub-tree.



A	B	C	D	E	F	G
D	L			R		

TREE TRAVERSAL TECHNIQUES

ALGORITHM FOR Pre-Order Binary Tree:

Temp = pointer variable initialized with root.
Data = data member of structure node
Left = pointer to left child node and member pointer of structure node.
Right = pointer to right child node and member pointer of structure node.

Step – 1: Repeat step 2,3,4 and check temp is not null.

Step – 2: Print data of node

Step – 3: Call function itself as a left most node.

Step -4: Call function itself as a right most node.

TREE TRAVERSAL TECHNIQUES

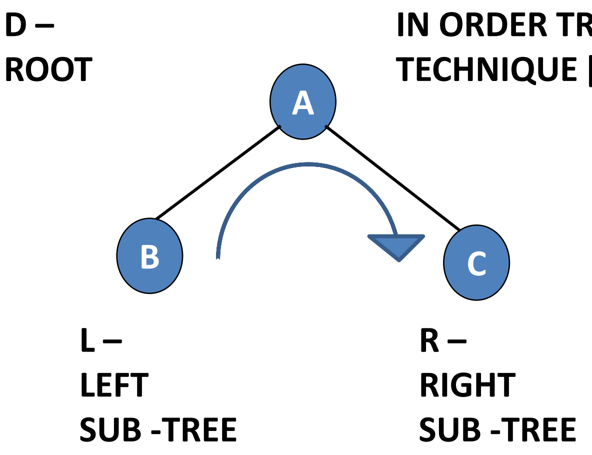
- Inorder Traversal:
  1. Traverse the left sub-tree – Recursively.
  2. Visit the root node.
  3. Traverse the right sub-tree – Recursively.

In the inorder traversal (D) is used for root node, (L) for left sub-tree and (R) is used for right sub-tree.

So, the inorder traversal is L – D – R.

TREE TRAVERSAL TECHNIQUES

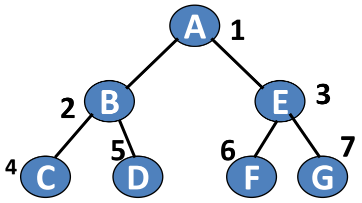
IN ORDER TRAVERSAL TECHNIQUE [L – D – R]



TREE TRAVERSAL TECHNIQUES

Consider following tree. The sequence for inorder traversal is:

1) Left Sub-tree processed first.
    1.1) For that process upto the left most leaf node of the tree (i.e. C).
    1.2) Then process to the root of left most leaf node of the tree (i.e. B).
    1.3) And then process to the root of right most leaf node of the tree (i.e. D).
2) Now, process the root node (i.e. A).
3) After processing of root node, now process its right sub-tree as per the rule of inorder traversing (i.e. L-D-R).



C	B	D	A	F	E	G
L			D	R		

TREE TRAVERSAL TECHNIQUES

ALGORITHM FOR InOrder Binary Tree:

Temp = pointer variable initialized with root.
Data = data member of structure node
Left = pointer to left child node and member pointer of structure node.
Right = pointer to right child node and member pointer of structure node.

Step – 1: Repeat step 2,3,4 and check temp is not null.

Step – 2: Call function itself as a left most node.

Step – 3: Print data of node.

Step -4: Call function itself as a right most node.

TREE TRAVERSAL TECHNIQUES

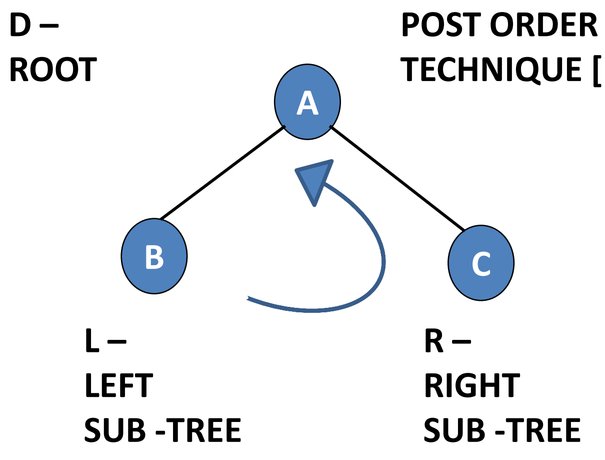
- Postorder Traversal:
  1. Traverse the left sub-tree – Recursively.
  2. Traverse the right sub-tree – Recursively.
  3. Visit the root node.

  In the postorder traversal (D) is used for root node, (L) for left sub-tree and (R) is used for right sub-tree.

  So, the postorder traversal is L – R – D.

TREE TRAVERSAL TECHNIQUES

POST ORDER TRAVERSAL TECHNIQUE [L – R – D]

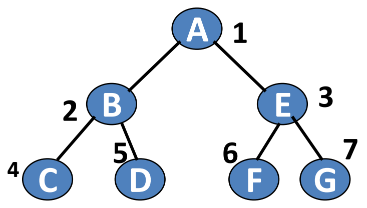


TREE TRAVERSAL TECHNIQUES

Consider following tree. The sequence for postorder traversal is:

1) Left Sub-tree processed first.
   1.1) For that process upto the left most leaf node of the tree (i.e. C).
   1.2) Then process to the right sub-tree (i.e. D) and then the root node (i.e. B).
2) Now, follow the rule L-R-D and process the right sub-tree.
   2.1) That is, as per the rule of post order (L-R-D), first process the left sub-tree (i.e. F).
   2.2) Then process right sub-tree (i.e. G) and then its parent (i.e. E).
3) At last process the root A.

C	D	B	F	G	E	A
L			R			D



TREE TRAVERSAL TECHNIQUES

ALGORITHM FOR PostOrder Binary Tree:

Temp = pointer variable initialized with root.
Data = data member of structure node
Left = pointer to left child node and member pointer of structure node.
Right = pointer to right child node and member pointer of structure node.

Step – 1: Repeat step 2,3,4 and check temp is not null.

Step – 2: Call function itself as a left most node.

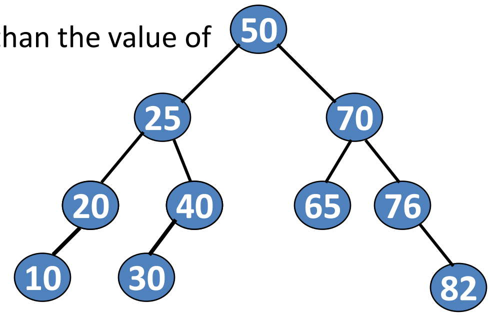
Step – 3: Call function itself as a right most node.

Step -4: Print data of node.

BINARY TREE PROGRAM

BINARY SEARCH TREE (BST)

- ❑ A Binary Search Tree is a binary tree, which is either empty or satisfies:
  - ❑ Every node has unique value.
  - ❑ If there is an existence of left child (left sub-tree) then its value is less than the value of the root.
  - ❑ If there is an existence of right child (right sub-tree) then its value is greater (larger) than the value of the root.



BINARY SEARCH TREE (BST)

- The primitive operations performed on BST are:
  - Inserting a node
  - Searching a node
  - Deleting a node

BINARY SEARCH TREE PROGRAM

BALANCED BINARY TREES

- A largest path through the left sub-tree is the same length as the largest path through the right sub-tree is called a BALANCED BINARY TREE.
- In this, the searching time is very less as compared to unbalanced binary tree.
- There are two types of Balanced Trees:
  - Height Balanced Trees
  - Weight Balanced Trees

BALANCED BINARY TREES

- ❑ In Height Balanced Trees, balancing the height is the important factor.
- ❑ Followings approaches may be used to balance the height /depth of a binary tree:
  - ➢ Insert a number of elements in binary tree (Like BST). After that copy these elements into another binary tree in such a way that the tree is balanced.
  - ➢ Use AVL Tree Algorithm.

BALANCED BINARY TREES

- ❑ AVL TREES:
  - ❖ The AVL Tree algorithm was developed by Two Russian Mathematicians, G. M. Adel'son Vel'sky and E. M. Landis in 1962 to save binary search trees.
  - ❖ An AVL Tree is a binary tree in which the left and the right sub – trees of any node may differ in height at most 1 and in which both the sub-trees are AVL Trees.
  - ❖ The height of an empty tree (NULL TREE) is defined to be -1.
  - ❖ The development of an AVL Tree is same as that of an ordinary binary tree except that after the addition of each new node, a check must be made to ensure that the AVL balance conditions have not been violated.

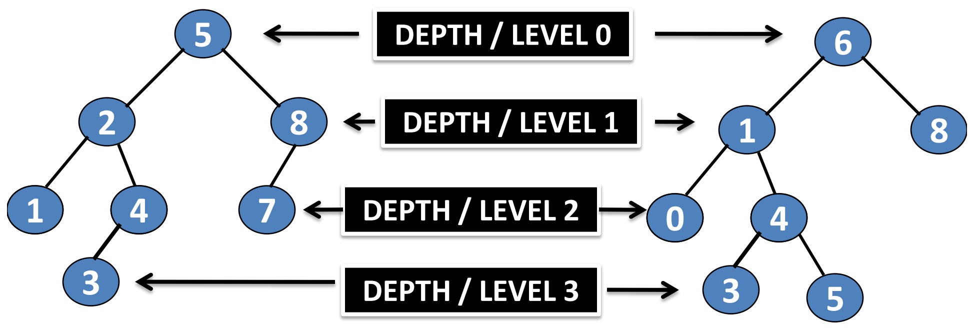
BALANCED BINARY TREES

❑ AVL TREES:

For any AVL Trees, one of the following properties must be true:

- ❖ If the largest path in its left sub-tree is one level larger than the largest path in its right sub-tree, then a node is called *LEFT HEAVY NODE*.
- ❖ If the largest path in its right sub-tree is one level larger than the largest path in its left sub-tree, then a node is called *RIGHT HEAVY NODE*.
- ❖ If both left and right sub-trees are equal then a node is called *BALANCED*.

BALANCED BINARY TREES



BALANCED NODES IN AVL TREE

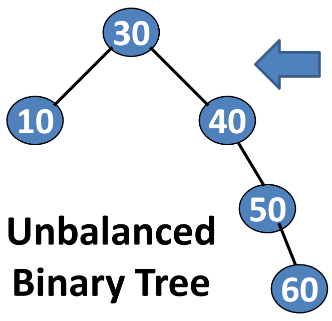
NOT AN AVL TREE

BALANCED BINARY TREES

❑ AVL TREE ALGORITHM:

- Step – 1: Insert new node similarly like Ordinary Binary Tree.
- Step – 2: Trace the path from new nodes to the root for checking height difference of two sub-trees.
- Step – 3: If these nodes lie in a straight line, then apply a *Single Rotation* to correct the imbalance and make Balanced Binary Tree (AVL Tree).
- Step – 4: If these nodes lie in a dogleg pattern (i.e. bend in a path) , then apply *Double Rotation* to correct the imbalance and make Balanced Tree.
- Step – 5: Exit

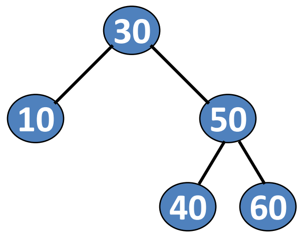
BALANCED BINARY TREES



Unbalanced Binary Tree

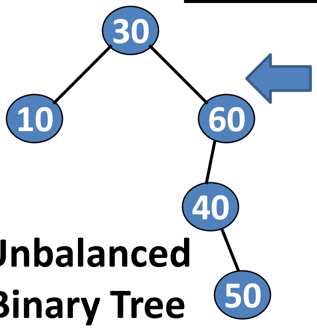
Now, As per the AVL Tree Algorithm, here the nodes lie in straight line and imbalance. So, applying single rotation to make this tree balanced tree.

The single rotation is to be applied to nodes 40, 50, 60 to generate the balanced tree.



Balanced Binary Tree

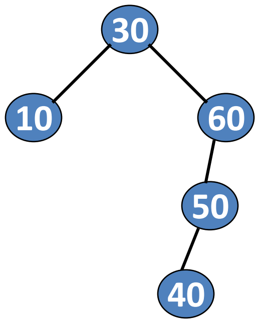
BALANCED BINARY TREES



Unbalanced Binary Tree

Now, As per the AVL Tree Algorithm, here the nodes lie in dogleg pattern and imbalance. So, applying double rotation to make this tree balanced tree.

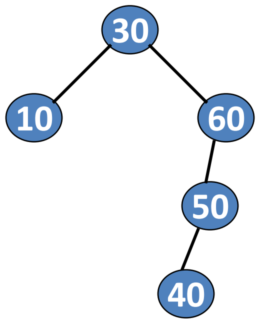
The double rotation is nothing but two single rotations.



1st Single Rotation

The 1st rotation performs on two levels (nodes 40 and 50). For that rotating the node 50 replacing 40 and so now, 50 becomes parent of 40.

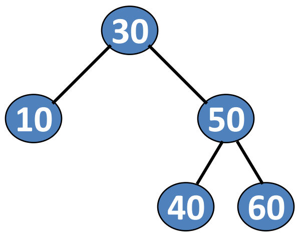
BALANCED BINARY TREES



2nd Single Rotation

Now, we can see that in the beside tree, we have three nodes 60, 50 and 40 which lie in straight line.

So, applying single rotation by replacing 60 by 50 and placing 60 as the right child of 50.



Balanced Binary Tree

B-TREES

- A B-tree is one type of data structure which generally found in databases and file systems.
- A B-tree generally grow from the bottom up as elements are inserted whereas most binary tree grow down.
- By maximizing the number of child nodes within each internal node, the height of the tree decreases, balancing occurs less often and efficiency increases.
- The idea behind B-trees is that inner nodes can have a variable number of child nodes within some pre-defined range.

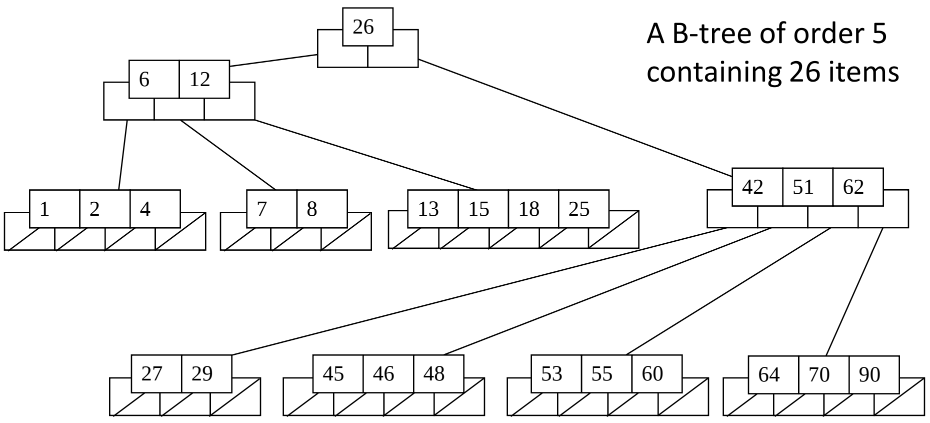
B-TREES

- 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 (*m*/2) children
  4. The root is either a leaf node, or it has from two to *m* children
  5. A non-leaf node contains no more than *m* – 1 keys
- The number *m* should always be odd.

B-TREES

- The creator of B-Trees' Rudolf Bayer has not explained the meaning of B. The most common belief is that B stands for Balanced, as all the leaf nodes are at the same level in the tree.
- B may also stand for Bayer or for Boeing because he was working for Boeing Scientific Research Labs.
- A node is considered to be in an illegal state if it has an invalid number of child nodes.

An example B-Tree

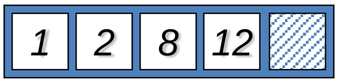


A B-tree of order 5 containing 26 items

Note that all the leaves are at the same level

Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 6 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:

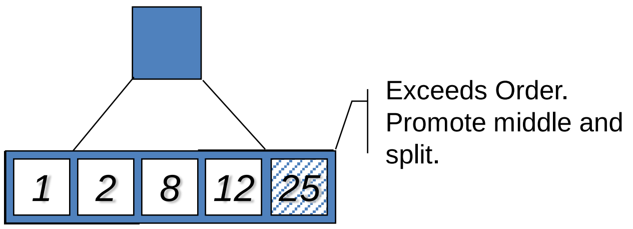


- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

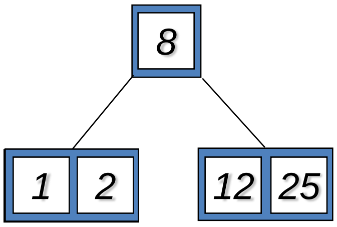
Constructing a B-tree

Add 25 to the tree

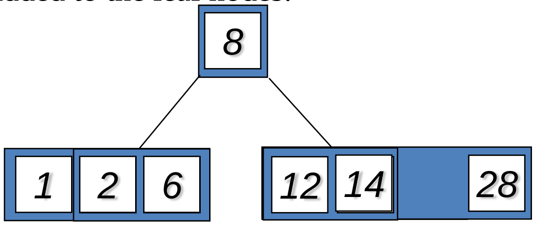


1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)



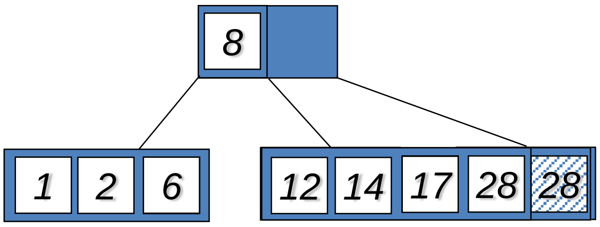
6, 14, 28 get added to the leaf nodes:



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

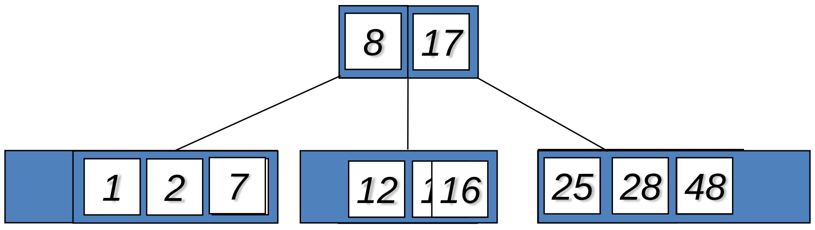
Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

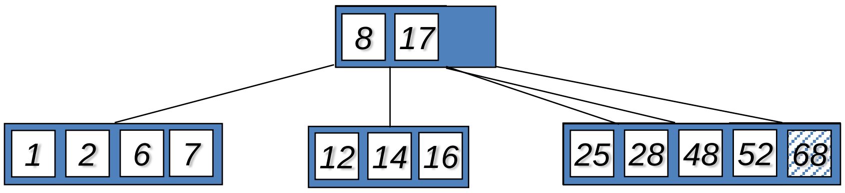
7, 52, 16, 48 get added to the leaf nodes



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

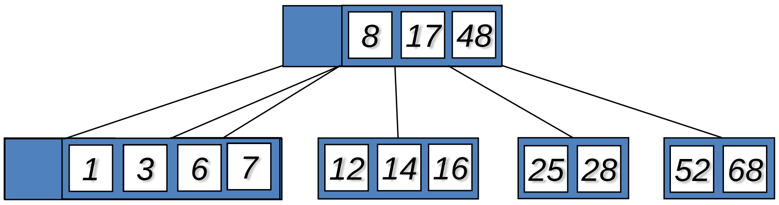
Adding 68 causes us to split the right most leaf, promoting 48 to the root



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

Constructing a B-tree (contd.)

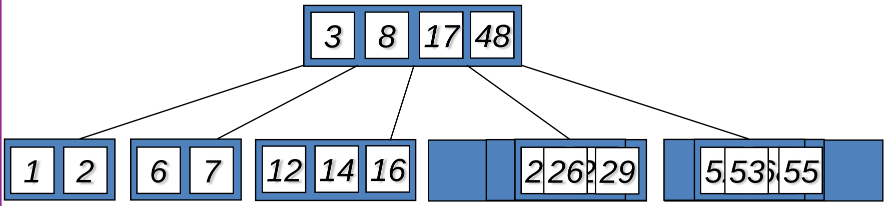
Adding 3 causes us to split the left most leaf



1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45

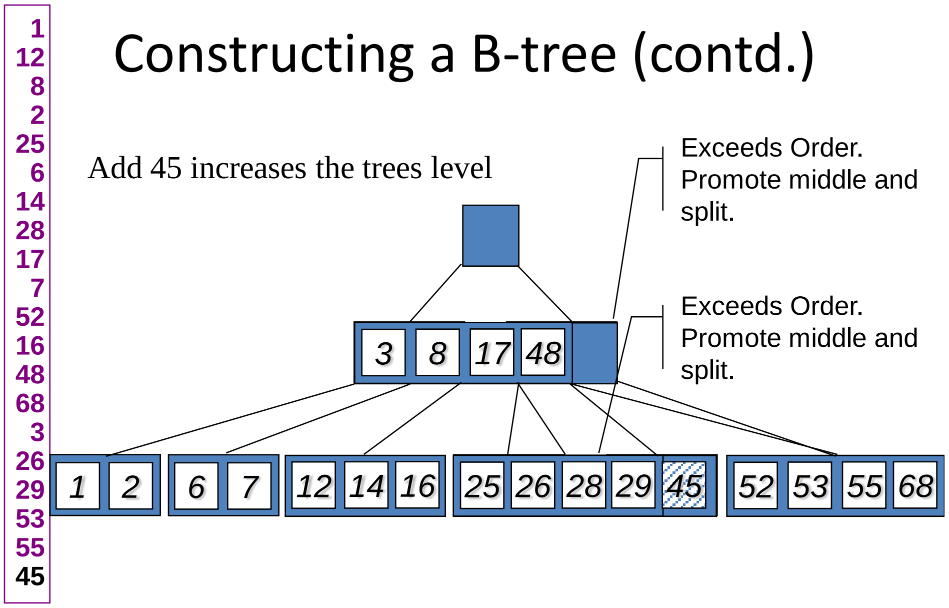
Constructing a B-tree (contd.)

Add 26, 29, 53, 55 then go into the leaves



Constructing a B-tree (contd.)

Add 45 increases the trees level

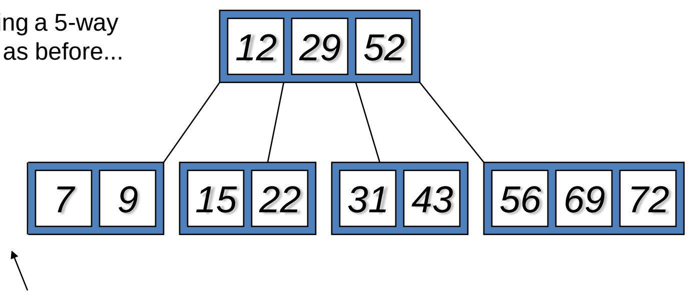

1
12
8
2
25
6
14
28
17
7
52
16
48
68
3
26
29
53
55
45
Exceeds Order. Promote middle and split.
Exceeds Order. Promote middle and split.
3
8
17
48
1
2
6
7
12
14
16
25
26
28
29
45
52
53
55
68

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

Type #1: Simple leaf deletion

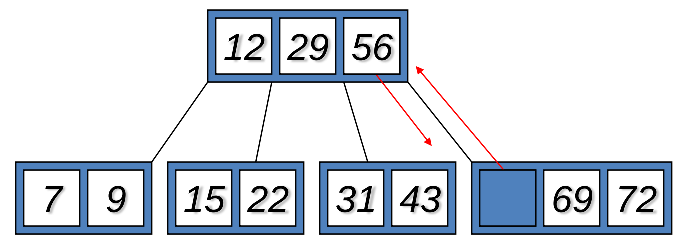
Assuming a 5-way B-Tree, as before...



Delete 2: Since there are enough keys in the node, just delete it

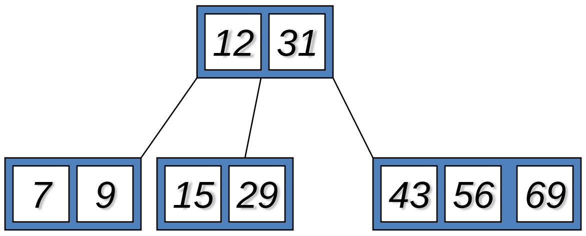
Note when printed: this slide is animated

Type #2: Simple non-leaf deletion



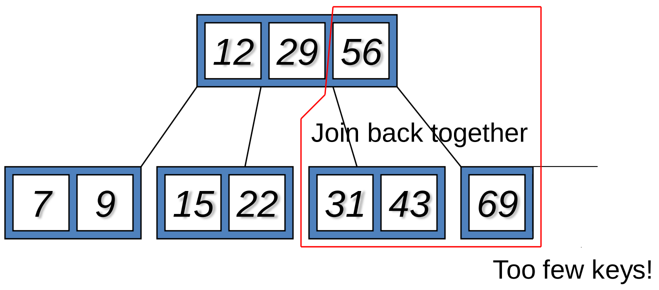
Note when printed: this slide is animated

Type #3: Enough siblings



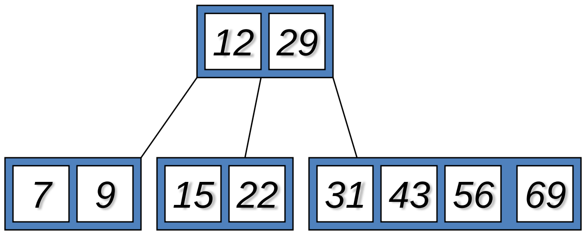
Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings



Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings



Note when printed: this slide is animated

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

- 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 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.