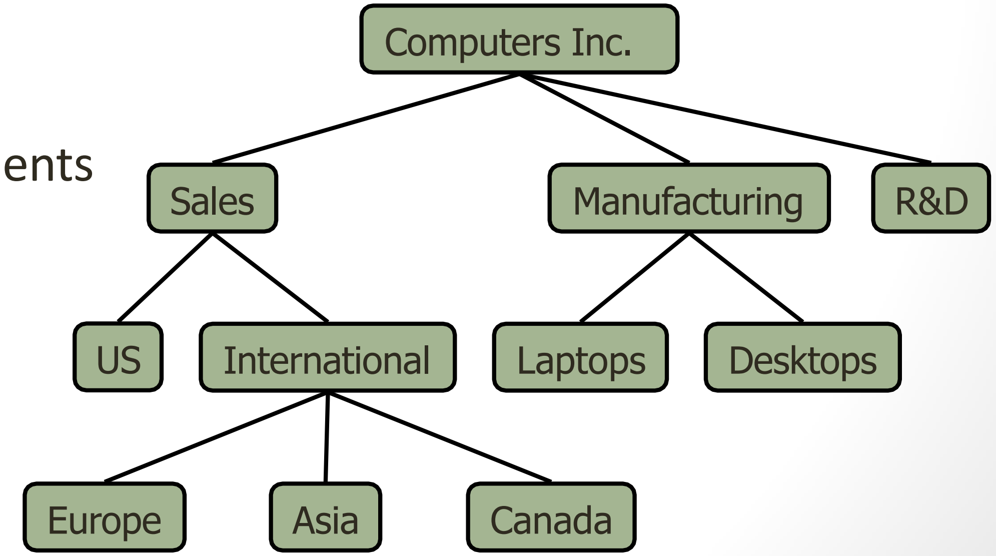
CHAPTER 6: TREES

BIBHA STHAPIT
ASST. PROFESSOR
IoE, PULCHOWK CAMPUS

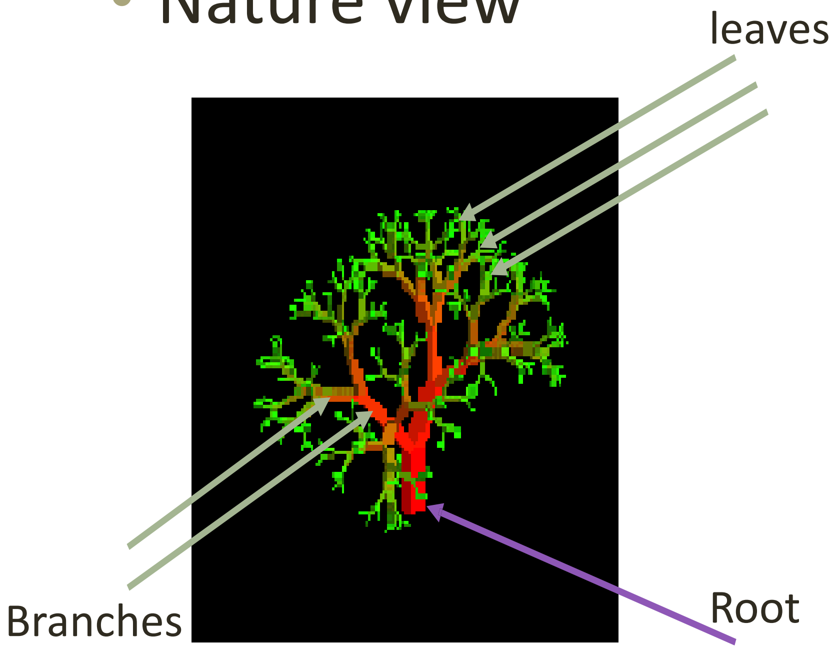
Trees

- In computer science, a tree is a widely-used data structure that emulates a hierarchical tree structure with a set of linked nodes.
- A tree is a finite nonempty set of elements.
- It is an abstract model of a hierarchical structure.
- consists of nodes with a parent-child relation.
- Applications:
  - Organization charts
  - File systems
  - Programming environments

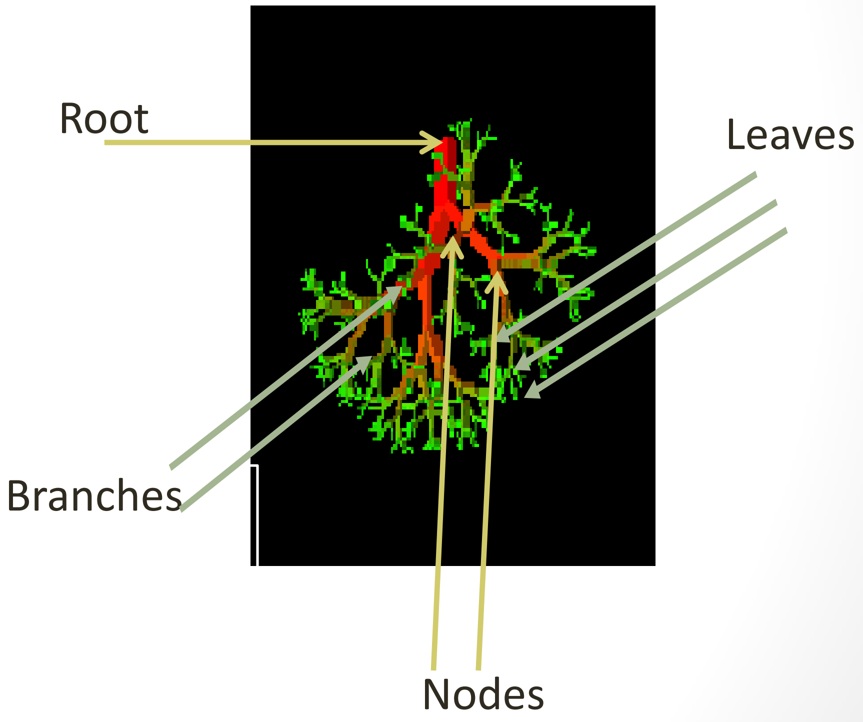


Trees

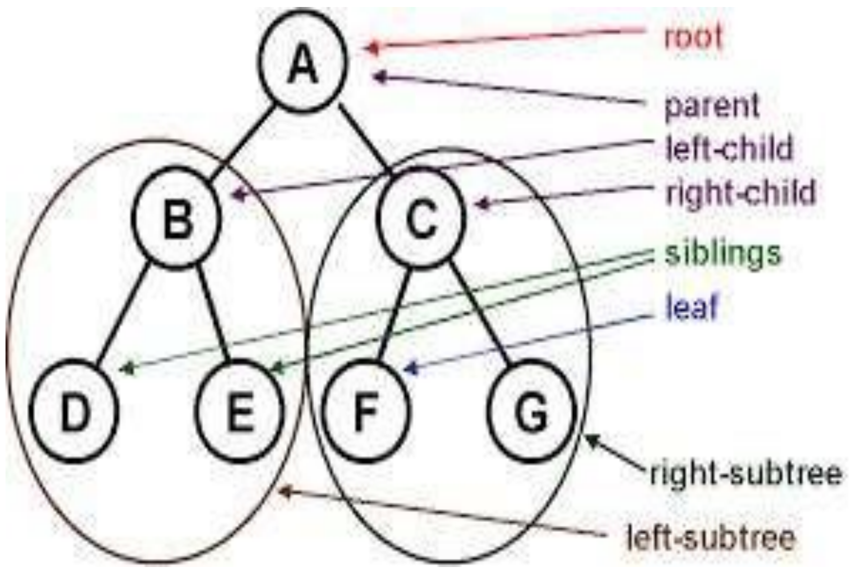
- Nature view

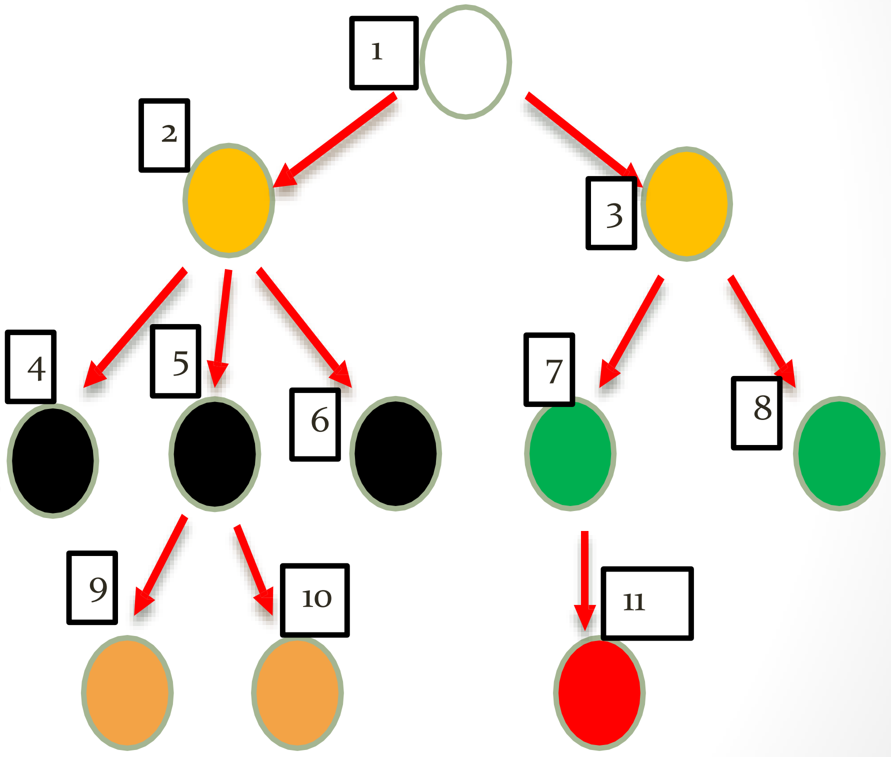


- Computer view



Tree terminologies



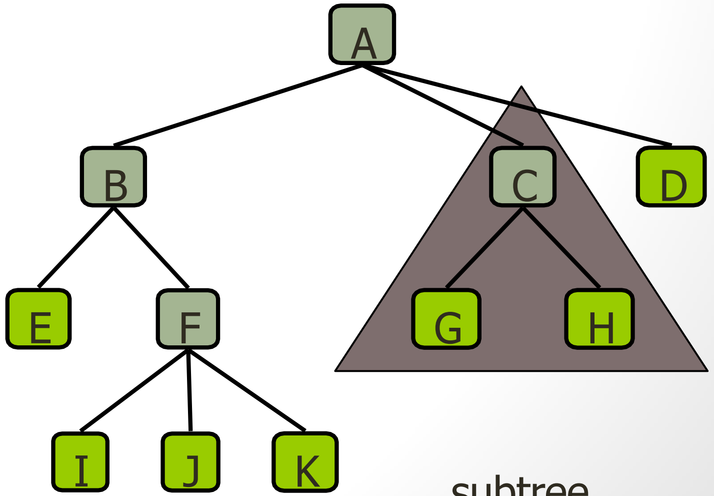


Tree terminologies

- **Path**: Traversal from node to node along the edges results in a sequence calledpath.
- **Root**: Node at the top of thetree.
- **Parent**: Any node, except root has exactly one edge running upward to another node. The node above it is called parent.
- **Child**: Any node may have one or more lines running downward to other nodes. Nodes below arechildren.
- **Leaf**: A node that has nochildren.
- **Subtree**: Any node can be considered to be the root of a subtree, which consists of its children and its children's children and soon.

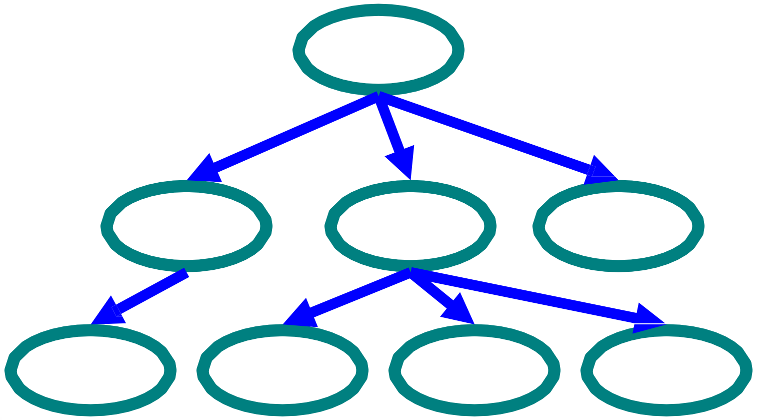
Tree terminologies

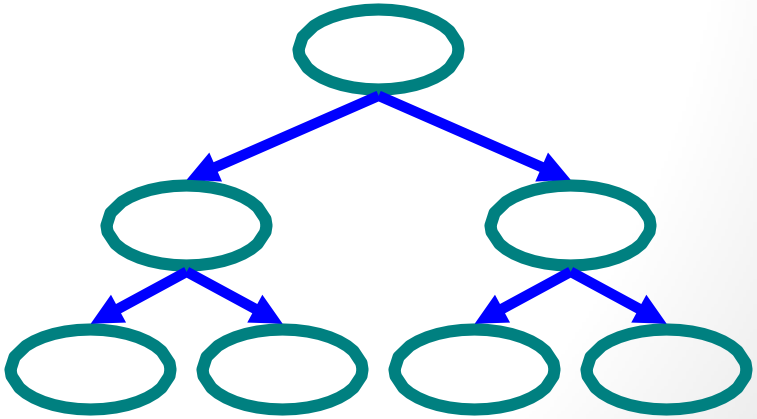
- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children(E,I,J,K,G,H,D)
- **Ancestors** of a node: parent, grandparent, great-grandparent,etc.
- **Descendant** of a node: child, grandchild, great-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum number of its node.
- **Subtree**: tree consisting of a node and its descendants



General trees and binary trees

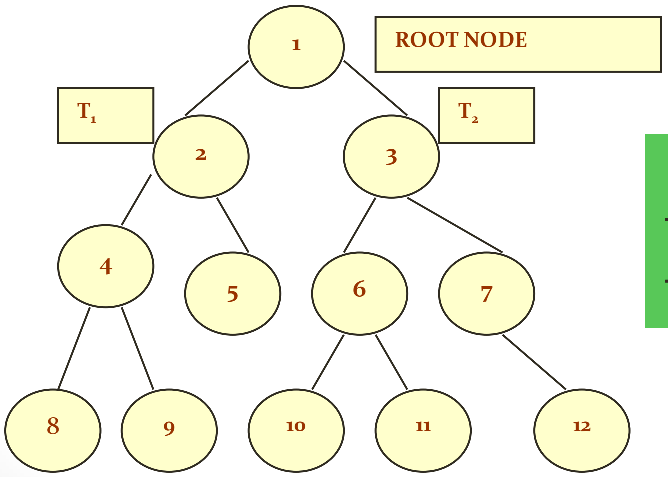
- Tree
  - Nodes
  - Each node can have 0 or more children
  - A node can have at most one parent
- Binary tree
  - Tree with 0–2 children per node


Tree


Binary Tree

Binary trees

- A binary tree is a data structure which is defined as a collection of elements called nodes.
- In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.
- Every node contains a data element, a "left" pointer which points to the left child, and a "right" pointer which points to the right child.
- The root element is pointed by a "root" pointer.
- If root = NULL, then it means the tree is empty.
- A binary tree of height h has at least h nodes and at most $2^h - 1$ nodes. This is because every level will have at least one node and can have at most 2 nodes.
- The height of a binary tree with n nodes is at least $\log_2(n+1)$ and at most n.



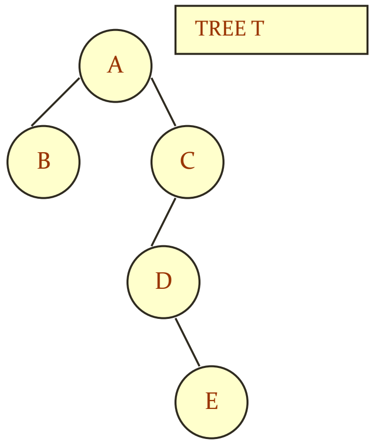
R – Root node (node 1)

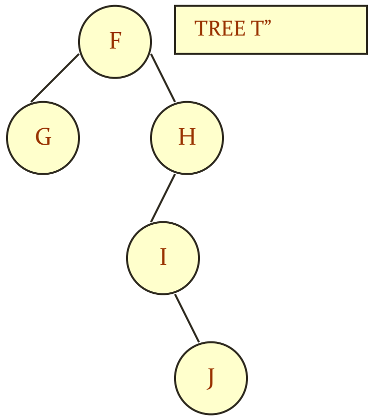
T_1- left sub-tree (nodes 2, 4, 5, 8, 9)

T_2- right sub-tree (nodes 3, 6, 7, 10, 11, 12)

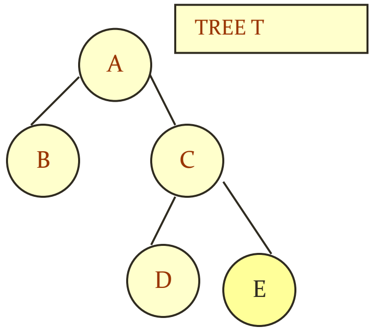
Binary trees

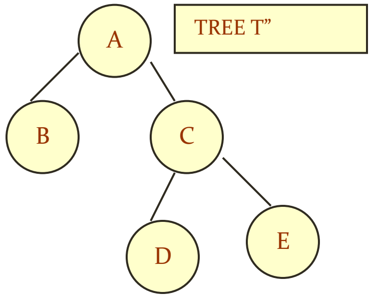
•*Similar binary trees:* Given two binary trees T and T' are said to be similar if both these trees have the same structure.





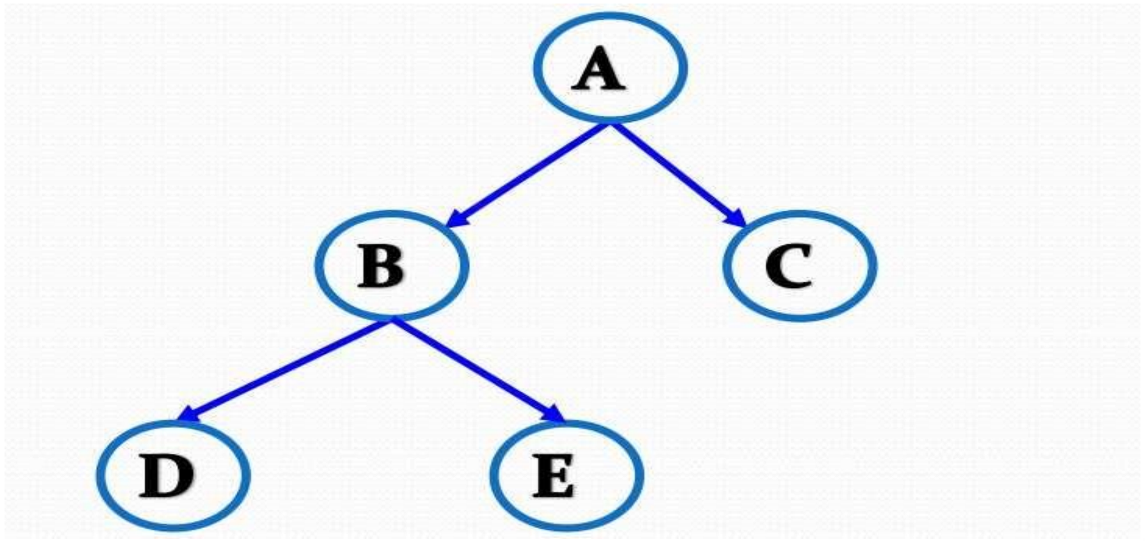
Copies of binary trees: Two binary trees T and T' are said to be *copies* if they have similar structure and same content at the corresponding nodes.





Strict binary tree

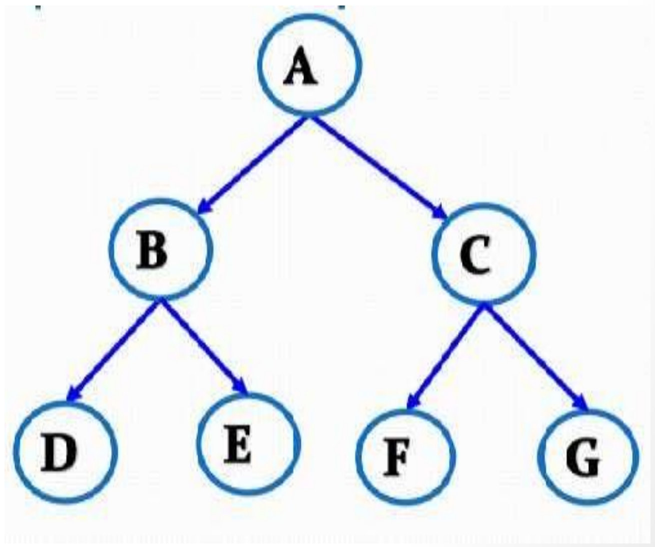
- Strict Binary Tree is a Binary tree in which root /node can have exactly two children or no children at all.
- That means, there can be 0 or 2 children of any node.
- Strict binary tree with ‘n’ leaves contains 2n-1 nodes.



Complete binary tree

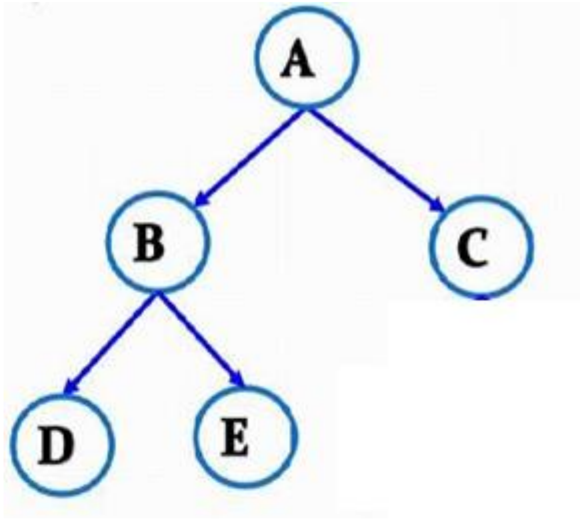
- Complete Binary Tree is a Strict Binary tree in which every leaf node is at same level/ depth ‘d’.
- That means, there are equal number of children in right and left subtree for every node.

- Leaf nodes= 2^d
- Nodes=$2^{(d+1)}-1$
- Non leaf nodes =2^d-1
- No. of nodes at each depth=2^d
- Total nodes= $\sum_{j=0}^{d} 2^j$



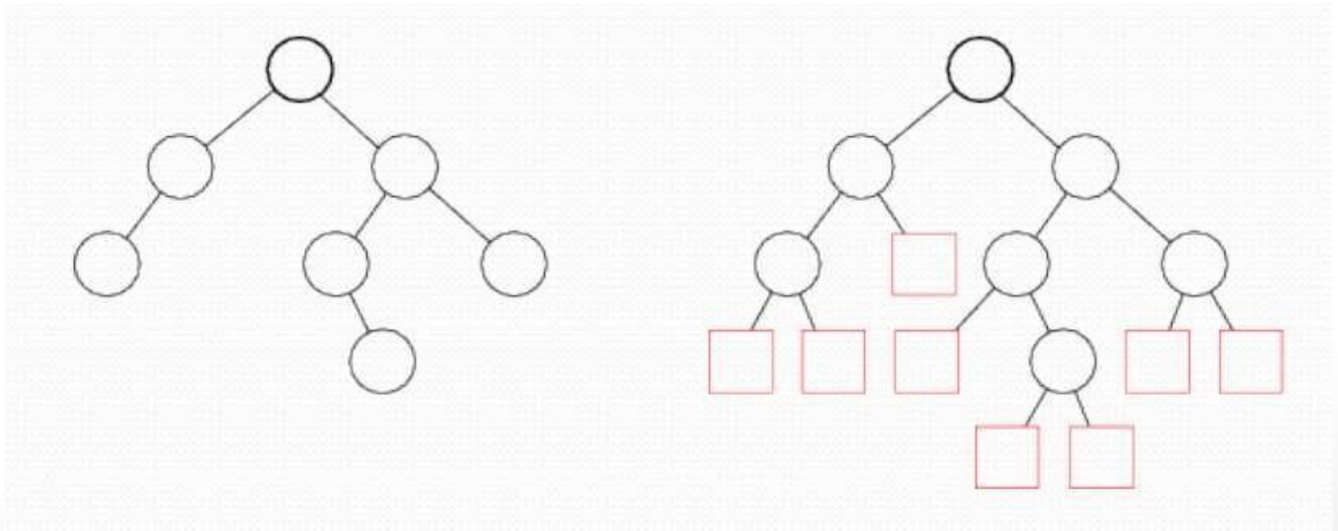
Almost complete binary tree

- A binary tree is said to be a almost complete binary tree, if all its level, except the last level, have maximum number of possible nodes, and all the nodes of the last level appear as far left as possible
- In almost complete binary tree, all leaf nodes are at last and second last level and levels are filled from left to right.



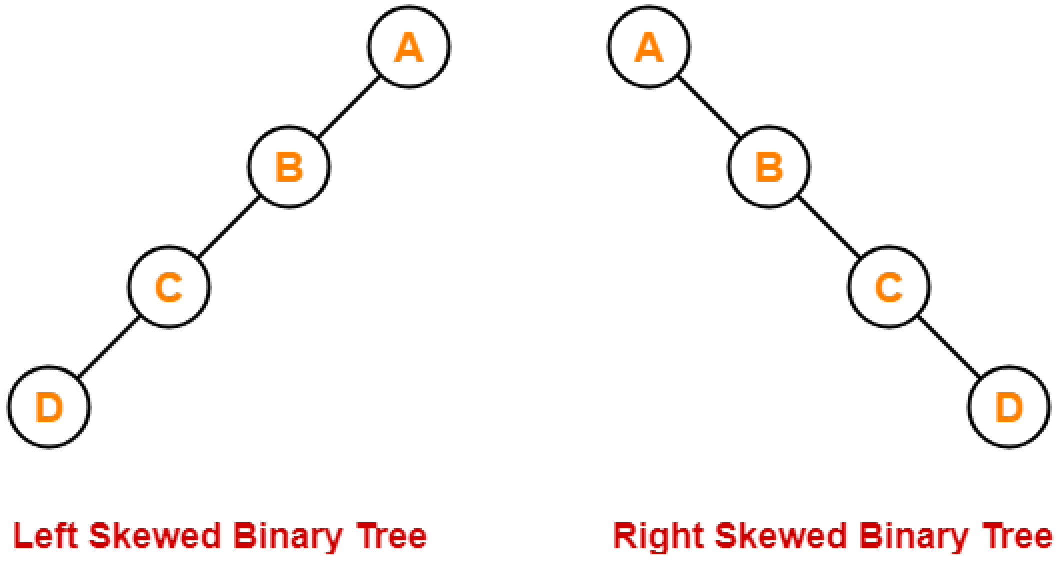
Extended binary tree

- An extended binary tree is a transformation of any binary tree into a complete binary tree.
- This transformation consists of replacing everynull subtree of the original tree with“special nodes.”
- The nodes from the original tree are then called as internal nodes, while the “special nodes” are calledas external nodes.



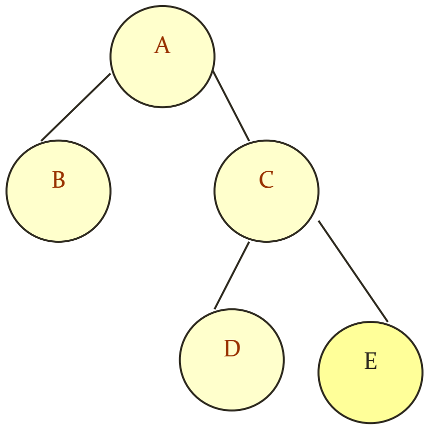
Skewed binary tree

- A **skewed binary tree** is a binary tree that satisfies the following 2 properties-
  - All the nodes except one node has one and only one child.
  - The remaining node has no child.
- A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).



Array implementation of binary tree

- If i=0 to n-1, then,
  - For PARENT with index (i)
  - LCH ILD (i) = 2i + 1
  - RCHILD (i) = 2i + 2



0	1	2	3	4	5	6	7	8	9
A	B	C	-	-	D	E	-	-	-

Array implementation of binary tree

- **Advantages:**

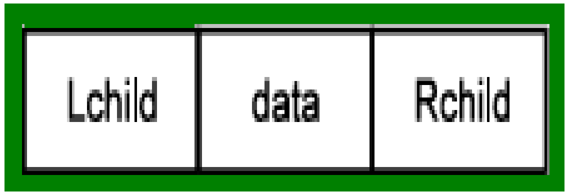
- Any node can be accessed from any other node by calculating the index
- Here, data are stored without any pointers to their successor or ancestor
- Programming languages, where dynamic memory allocation is not possible (such as BASIC, FORTRAN), array representation is the only mean to store a tree

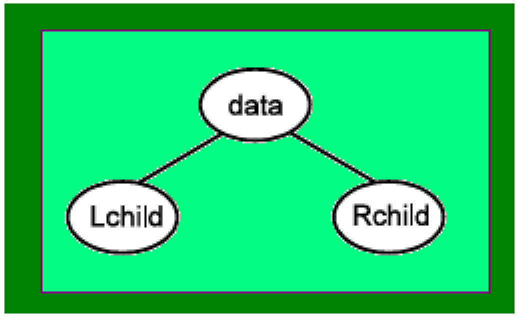
- **Disadvantages:**

- Other than full binary trees, majority of the array entries may be empty
- A new node to it or deleting a node from it are inefficient with this representation, because these require considerable data, movement up and down the array which demand excessive amount of processing time

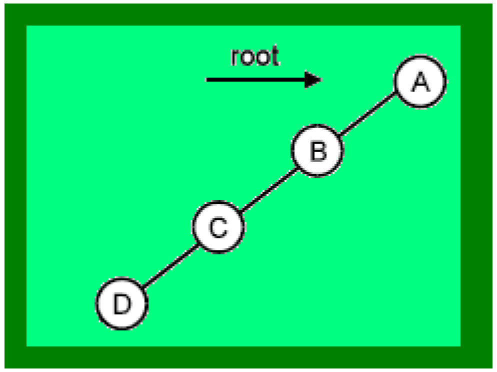
Linked implementation of binary tree

- Binary tree has natural implementation in linked storage. In linked organization we wish that all nodes should be allocated dynamically
- Hence we need each node with data and link fields
- Each node of a binary tree has both a left and a right subtree
- Each node will have three fields Lchild, Data, and Rchild. Pictorially this node is shown in Fig.





Linked implementation of binary tree



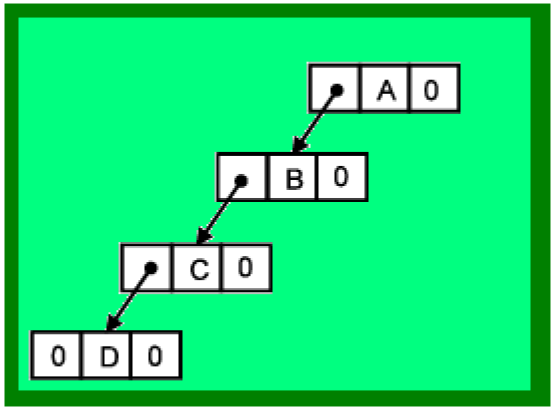


Figure 'a'

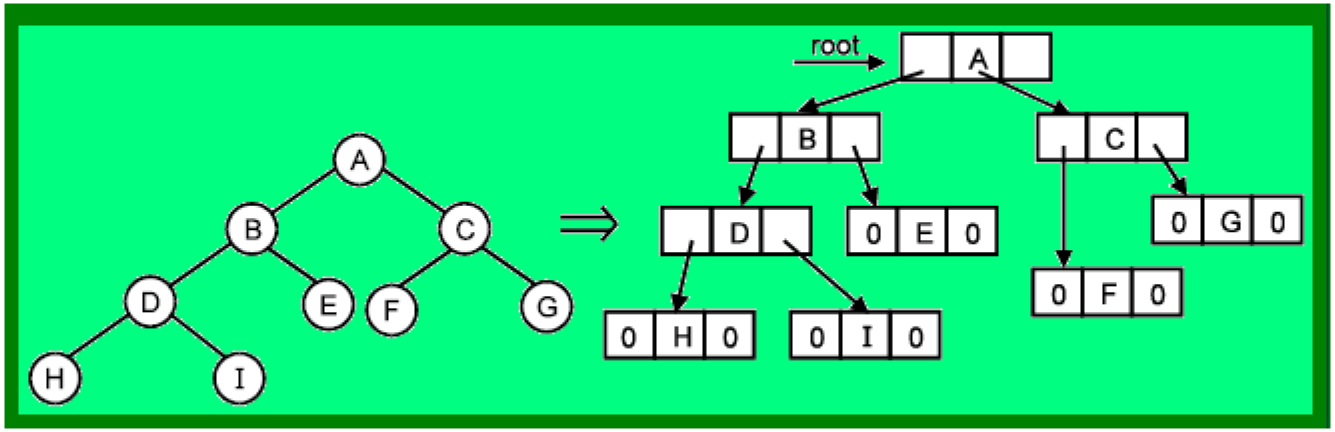


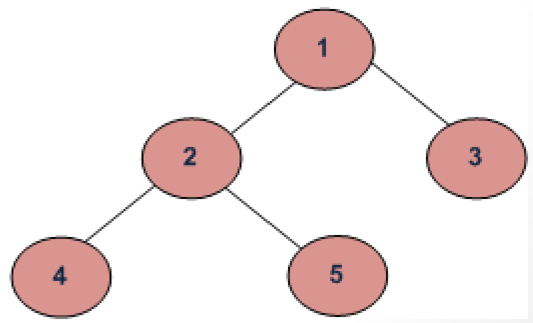
Figure 'b'

Linked implementation of binary tree

- **Advantages:**
- The drawback of sequential representation are overcome in this representation
- We may or may not know the tree depth in advance. Also for unbalanced tree, memory is not wasted
- Insertion and deletion operations are more efficient in this representation.
- **Disadvantages:**
- In this representation, there is no direct access to any node
- It has to be traversed from root to reach to a particular node
- As compared to sequential representation memory needed per node is more
- This is due to two link fields (left child and right child for binary trees) in node

Tree traversal

- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.
- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node.
- That is, we cannot randomly access a node in a tree.
- There are three ways which we use to traverse a tree
  - Inorder (Left, Root, Right) : 4 2 5 1 3
  - Preorder (Root, Left, Right) : 1 2 4 5 3
  - Postorder (Left, Right, Root) : 4 5 2 3 1



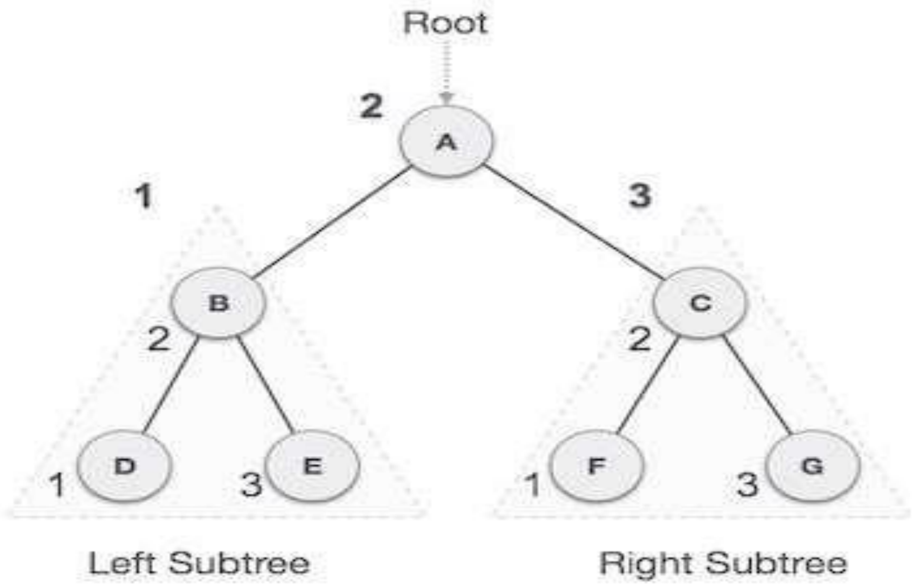
In-order traversal(L, V, R)

- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.

- Until all nodes are traversed –
- **Step 1** – Recursively traverse left subtree.
- **Step 2** – Visit root node.
- **Step 3** – Recursively traverse right subtree.

In-order traversal(L, V, R)

- We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –
- $D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$



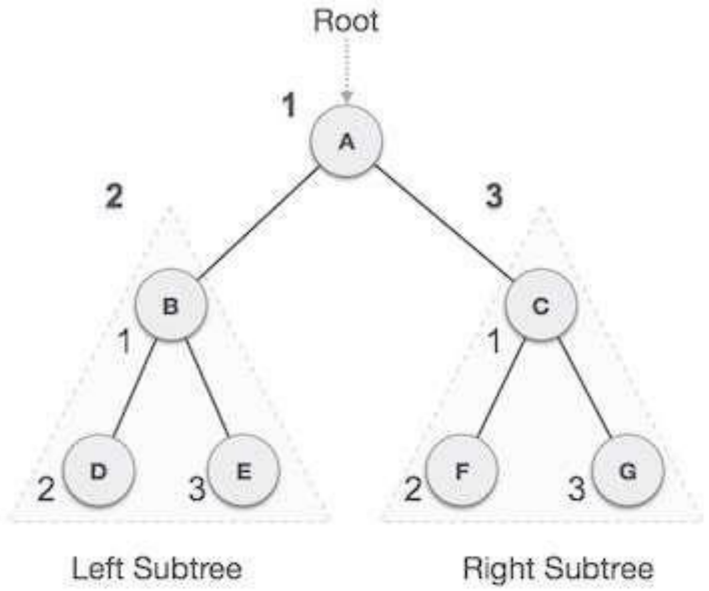
Pre-order traversal (V, L, R)

- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

- Until all nodes are traversed –
- **Step 1** – Visit root node.
- **Step 2** – Recursively traverse left subtree.
- **Step 3** – Recursively traverse right subtree.

Pre-order traversal (V, L, R)

- We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be.
- $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$



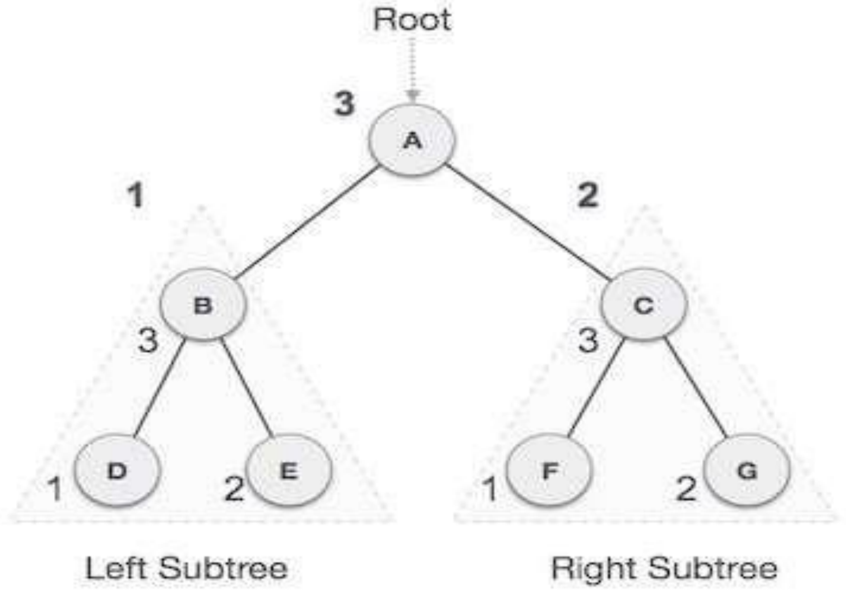
Post-order traversal (L, R, V)

- In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

- Until all nodes are traversed
- **Step 1** – Recursively traverse left subtree.
- **Step 2** – Recursively traverse right subtree.
- **Step 3** – Visit root node.

Post-order traversal (L, R, V)

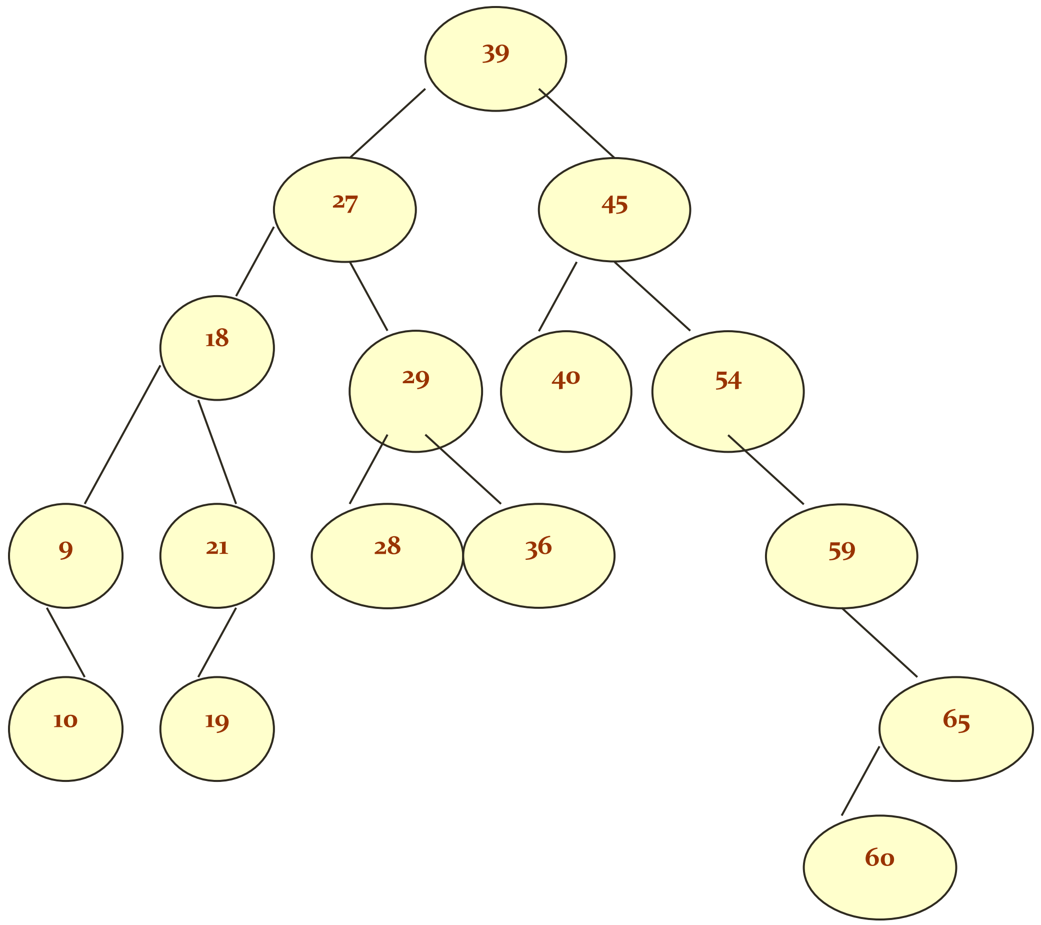
- We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be.
- ***D → E → B → F → G → C → A***



Binary search tree (BST)

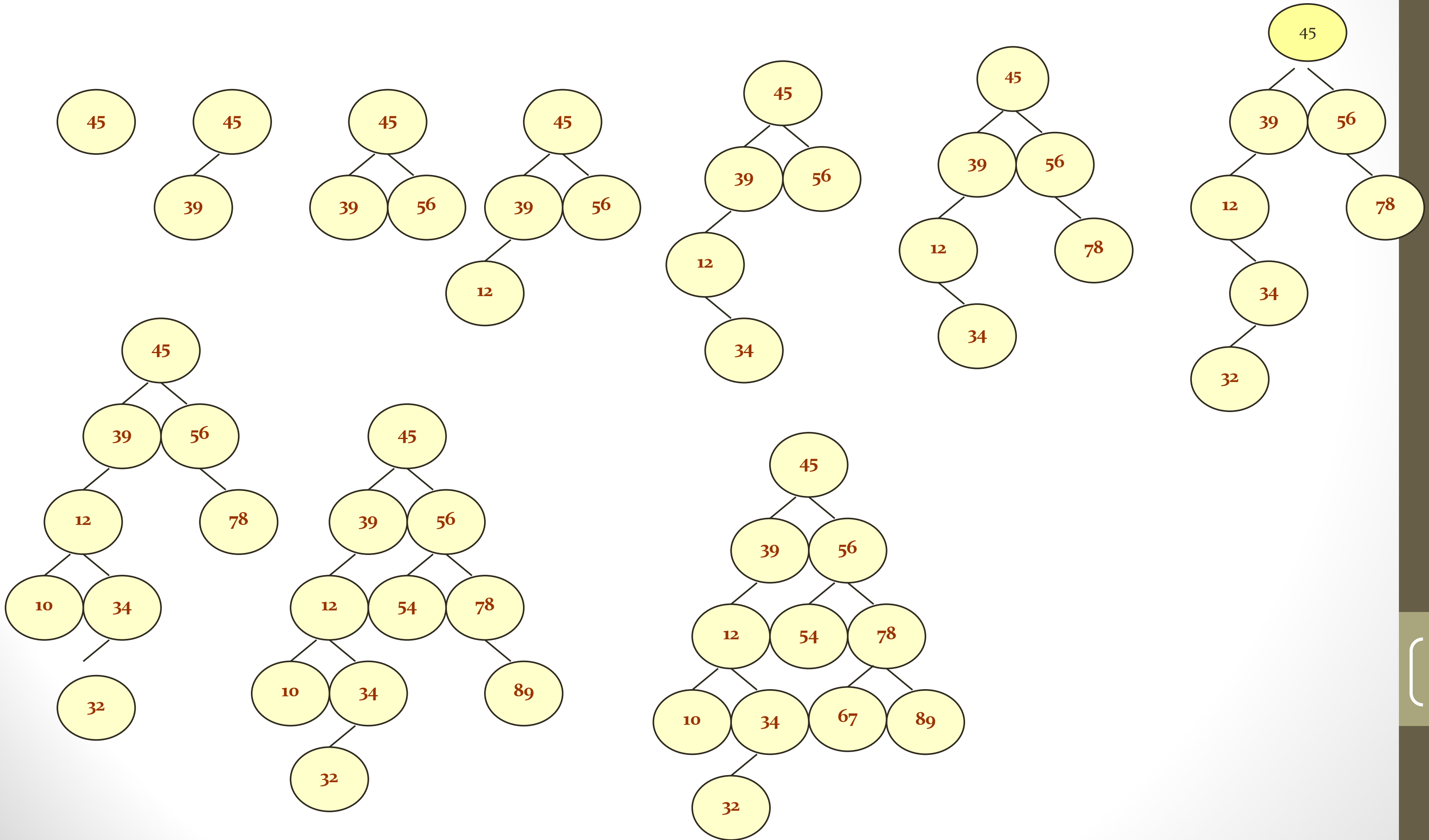
- A binary search tree (BST), also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in order.
- In a BST, all nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
- Due to its efficiency in searching elements, BSTs are widely used in dictionary problems where the code always inserts and searches the elements that are indexed by some key value.

Binary search tree (BST)



Creating BST from given value

45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67



Algorithm to insert value in BST

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL, then
        Allocate memory for TREE
           SET TREE->DATA = VAL
        SET TREE->LEFT = TREE ->RIGHT = NULL
         ELSE
        IF VAL < TREE->DATA
        Insert(TREE->LEFT, VAL)
        ELSE
        Insert(TREE->RIGHT, VAL)
           [END OF IF]
         [END OF IF]

Step 2: End
```

Searching a value in BST

- The search function is used to find whether a given value is present in the tree or not.
- The function first checks if the BST is empty. If it is, then the value we are searching for is not present in the tree, and the search algorithm terminates by displaying an appropriate message.
- However, if there are nodes in the tree then the search function checks to see if the key value of the current node is equal to the value to be searched.
- If not, it checks if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node.
- In case the value is greater than the value of the node, it should be recursively called on the right child node.

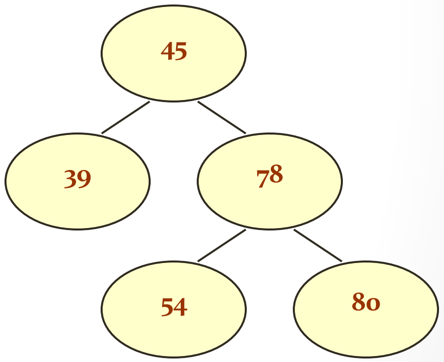
Algorithm to search a value in BST

```
searchElement (TREE, VAL)

Step 1: IF TREE->DATA = VAL OR TREE = NULL,
  then
       Return TREE
          ELSE
        IF VAL < TREE->DATA
       Return searchElement(TREE->LEFT, VAL)
        ELSE
       Return searchElement(TREE->RIGHT, VAL)
        [END OF IF]
          [END OF IF]

Step 2: End
```

Determining height of BST

- In order to determine the height of a BST, we will calculate the height of the left and right sub-trees. Whichever height is greater, 1 is added to it.
- Since height of right sub-tree is greater than the height of the left sub-tree, the height of the tree = height (right sub-tree) + 1= 2 + 1 = 3

```
Height (TREE)

Step 1: IF TREE = NULL, then
      Return 0
        ELSE
      SET LeftHeight = Height(TREE->LEFT)
      SET RightHeight = Height(TREE->RIGHT)
      IF LeftHeight > RightHeight
             Return LeftHeight + 1
      ELSE
             Return RightHeight + 1
      [END OF IF]
        [END OF IF]
Step 2: End
```



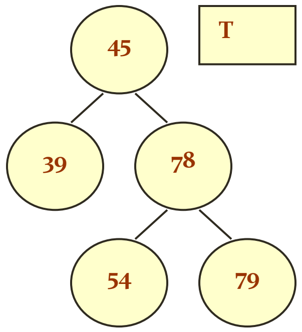
Determining number of nodes in BST

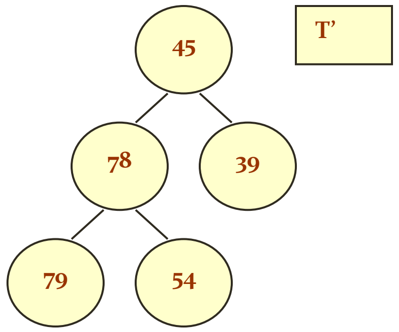
- To calculate the total number of elements/nodes in a BST, we will count the number of nodes in the left sub-tree and the right sub-tree.
- *Number of nodes = totalNodes(left sub-tree) + total Nodes(right sub-tree) + 1*

```
totalNodes (TREE)

Step 1: IF TREE = NULL, then
     Return 0
       ELSE
     Return totalNodes(TREE->LEFT)  +
                    totalNodes(TREE->RIGHT) + 1
       [END OF IF]
Step 2: End
```

Finding mirror image of BST

- Mirror image of a binary search tree is obtained by interchanging the left sub-tree with the right sub-tree at every node of the tree. For example, given the tree T, the mirror image of T can be obtained as T'.





```
MirrorImage (TREE)
Step 1: IF TREE != NULL , then
      MirrorImage(TREE->LEFT)
      MirrorImage(TREE->RIGHT)
      SET TEMP = TREE->LEFT
      SET TREE->LEFT = TREE->RIGHT
      SET TREE_>RIGHT = TEMP
        [END OF IF]
Step 2: End
```

Deleting a BST

- To delete/remove the entire binary search tree from the memory, we will first delete the elements/nodes in the left sub-tree and then delete the right sub-tree.

```
deleteTree (TREE)
Step 1: IF TREE != NULL , then
     deleteTree (TREE->LEFT)
     deleteTree (TREE->RIGHT)
     Free (TREE)
        [END OF IF]
Step 2: End
```

Finding smallest node in BST

- The basic property of a BST states that the smaller value will occur in the left sub-tree.
- If the left sub-tree is NULL, then the value of root node will be smallest as compared with nodes in the right sub-tree.
- So, to find the node with the smallest value, we will find the value of the leftmost node of the left sub-tree.
- However, if the left sub-tree is empty then we will find the value of the root node.

```
findSmallestElement (TREE)
Step 1: IF TREE = NULL OR TREE->LEFT = NULL, then
           Return TREE
  ELSE
           Return findSmallestElement(TREE->LEFT)
  [END OF IF]
Step 2: End
```

Deleting a value/node from a BST

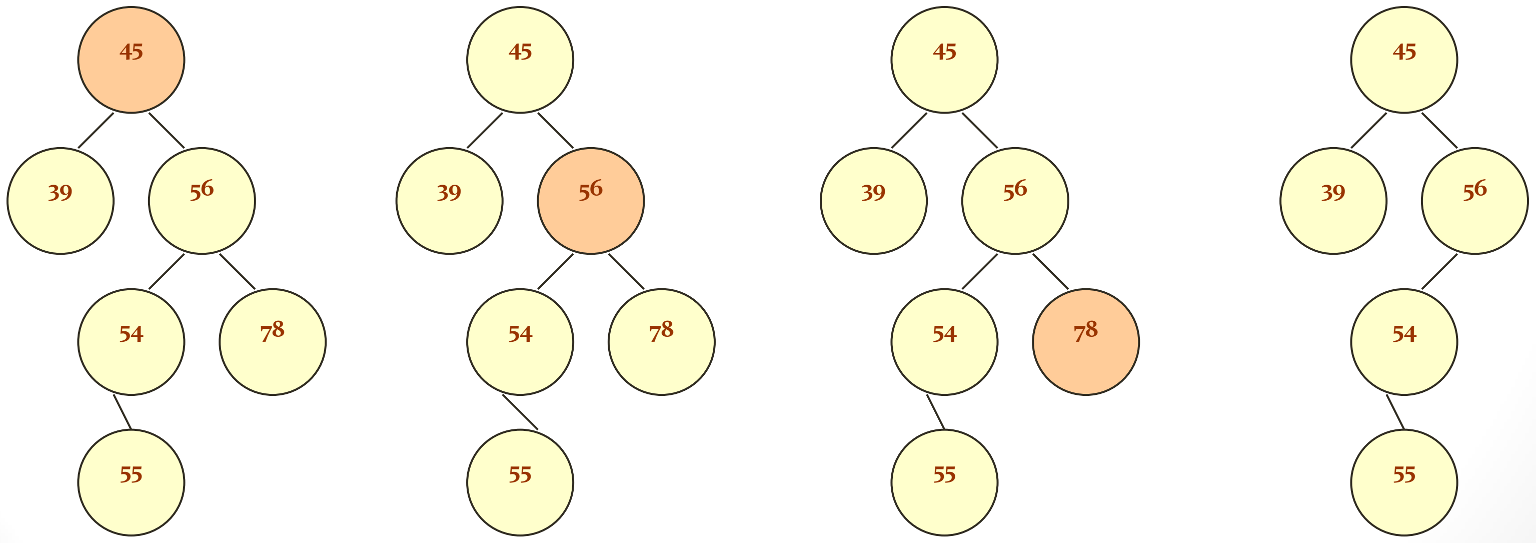
- The delete function deletes a node from the binary search tree.
- However, care should be taken that the properties of the BSTs do not get violated and nodes are not lost in the process.
- The deletion of a node involves any of the three cases.
  - Deleting a node that has no children.
  - Deleting a node that has one child.
  - Deleting a node that has both children.

Deleting a value/node from a BST

***Case 1:* Deleting a node that has no children.**

- Just remove / disconnect the leaf node that is to deleted from the tree.

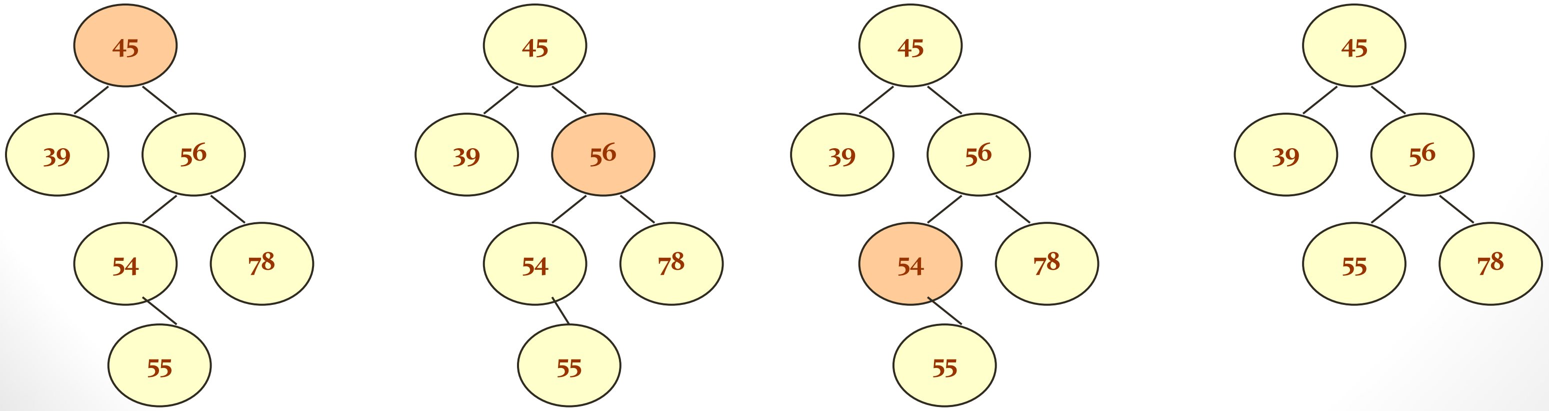
For example, deleting node 78 in the tree below.



Deleting a value/node from a BST

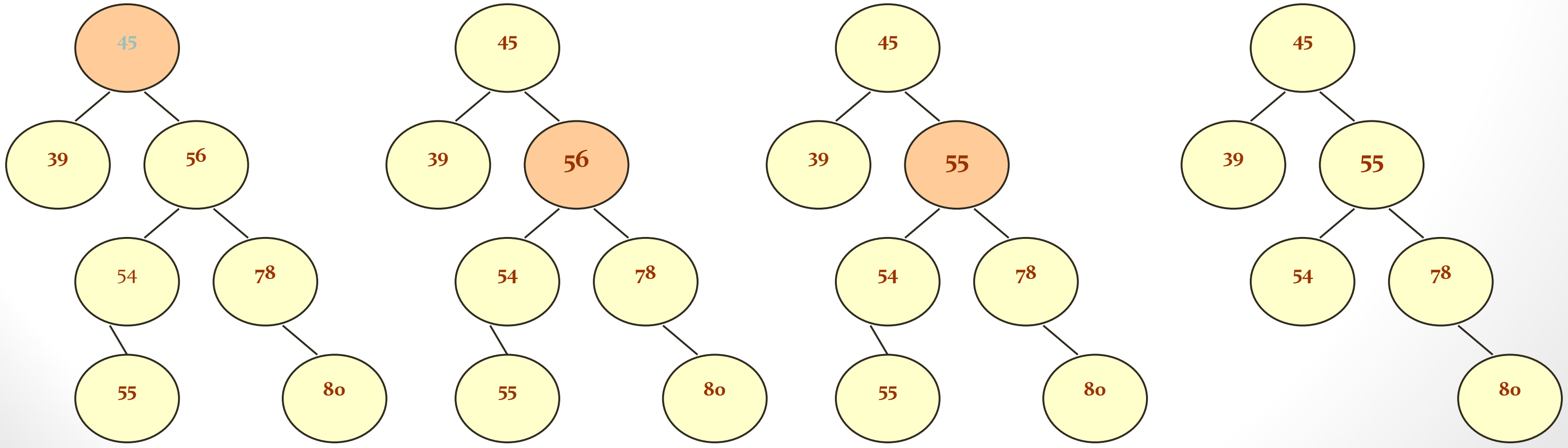
- ***Case 2:* Deleting a node with one child (either left or right).**
- To handle the deletion, the node's child is set to be the child of the node's parent. (Just make the child of the deleting node, the child of its grandparent.)
- Now, if the node was the left child of its parent, the node's child becomes the left child of the node's parent.
- Correspondingly, if the node was the right child of its parent, the node's child becomes the right child of the node's parent.

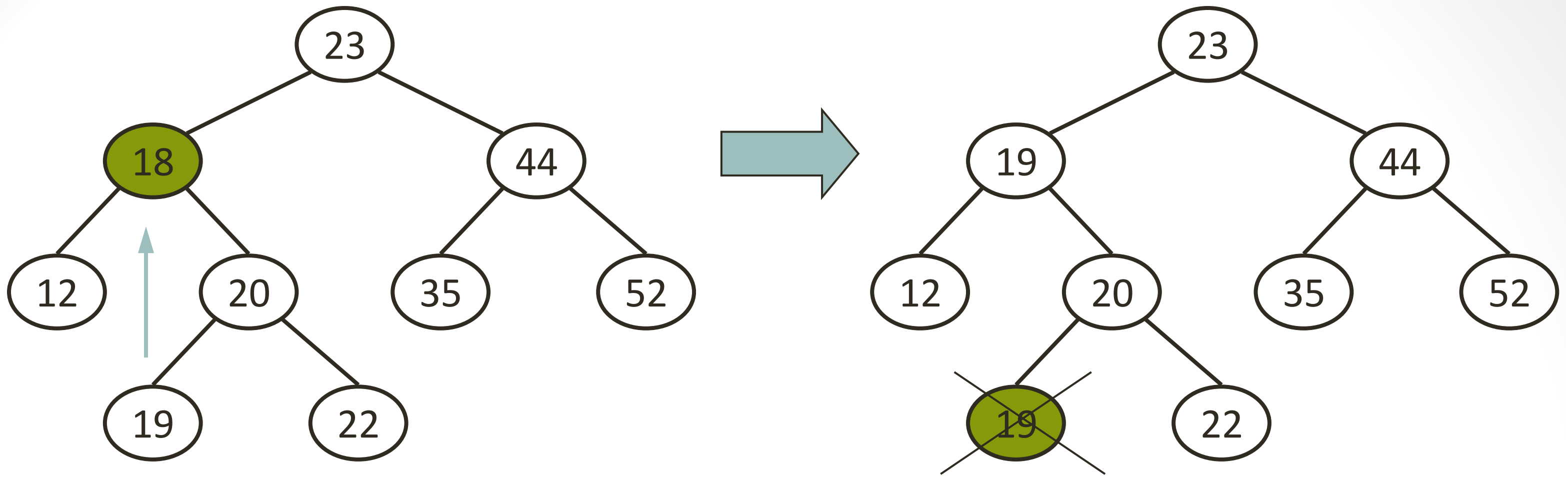
- For example, deleting node 54 in the tree below.



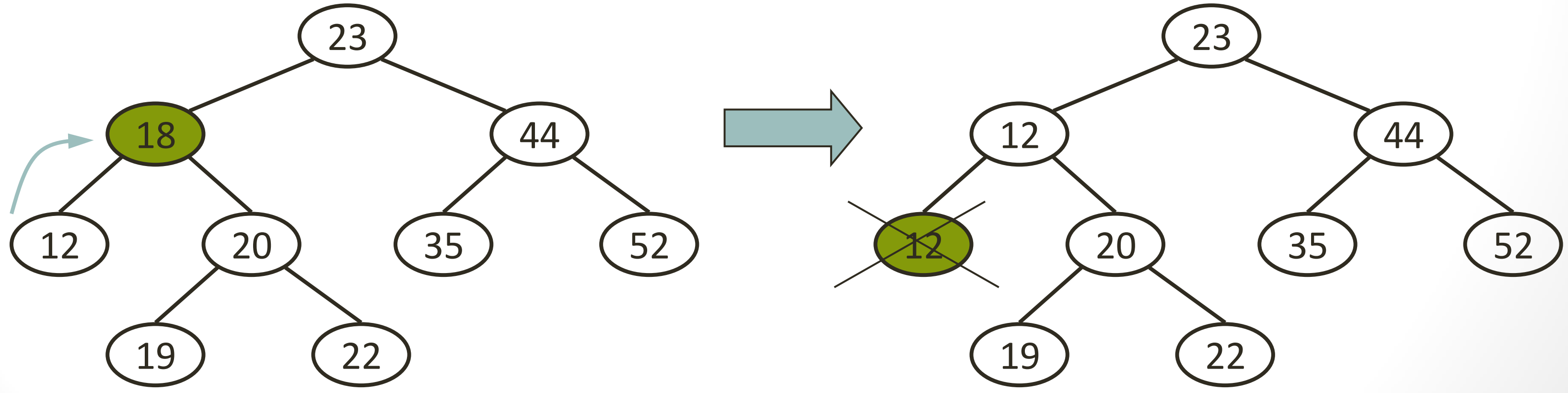
Deleting a value/node from a BST

- *Case 3:* Deleting a node with two children.
- To handle this case of deletion, replace the node's value with its in-order predecessor (largest value in the left sub-tree) or in-order successor (smallest value in the right sub-tree).
- The in-order predecessor or the successor can then be deleted using any of the above cases.
- For example, deleting node 56 in the tree below.





Using smallest node in the right subtree



Using largest node in the left subtree

Algorithm to delete from BST

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL, then
           Write "VAL not found in the tree"
           ELSE IF VAL < TREE->DATA
             Delete(TREE->LEFT, VAL)
           ELSE IF VAL > TREE->DATA
           Delete(TREE->RIGHT, VAL)
           ELSE IF TREE->LEFT AND TREE->RIGHT
       SET TEMP = findLargestNode(TREE->LEFT)
       SET TREE->DATA = TEMP->DATA
       Delete(TREE->LEFT, TEMP->DATA)
           ELSE
      SET TEMP = TREE
      IF TREE->LEFT = NULL AND TREE ->RIGHT = NULL
         SET TREE = NULL
      ELSE IF TREE->LEFT != NULL
         SET TREE = TREE->LEFT
      ELSE
         SET TREE = TREE->RIGHT
      [END OF IF]
                FREE TEMP
           [END OF IF]
Step 2: End
```

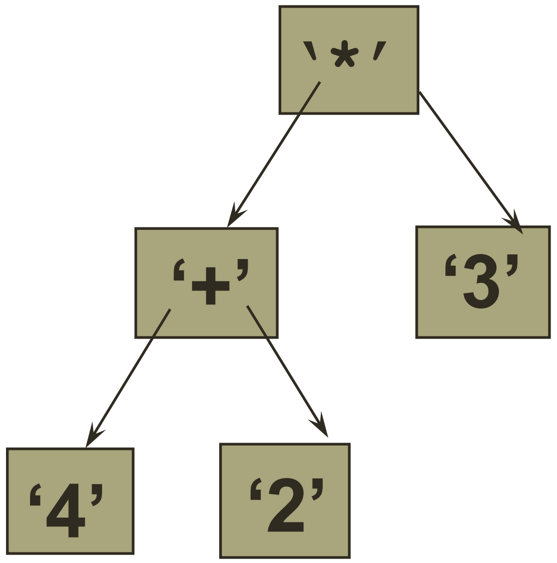
Expression trees

- A special kind of binary tree in which:

1. Each leaf node contains a single operand
2. Each non-leaf node contains a single binary operator
3. The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

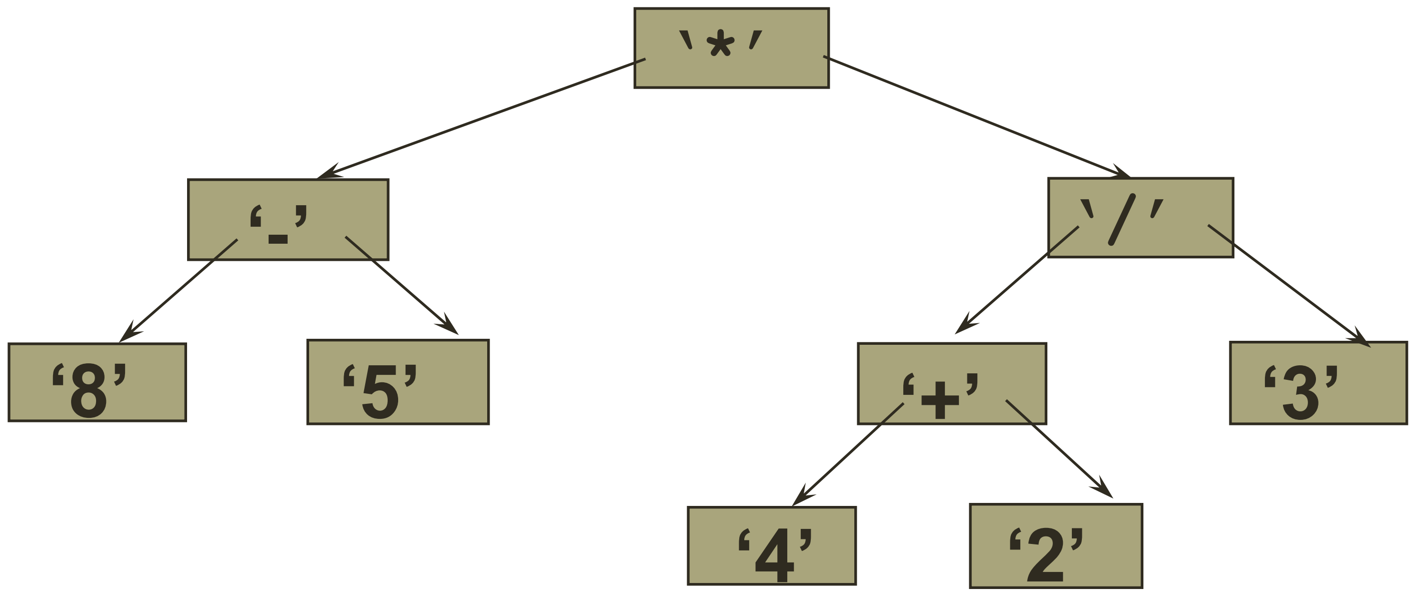
- The levels of the nodes in the tree indicate their relative precedence of evaluation (we do not need parentheses to indicate precedence).
- Operations at higher levels of the tree are evaluated later than those below them.
- The operation at the root is always the last operation performed.

Expression trees



- What value does it have?
- (4 + 2) * 3 = 18

Expression trees

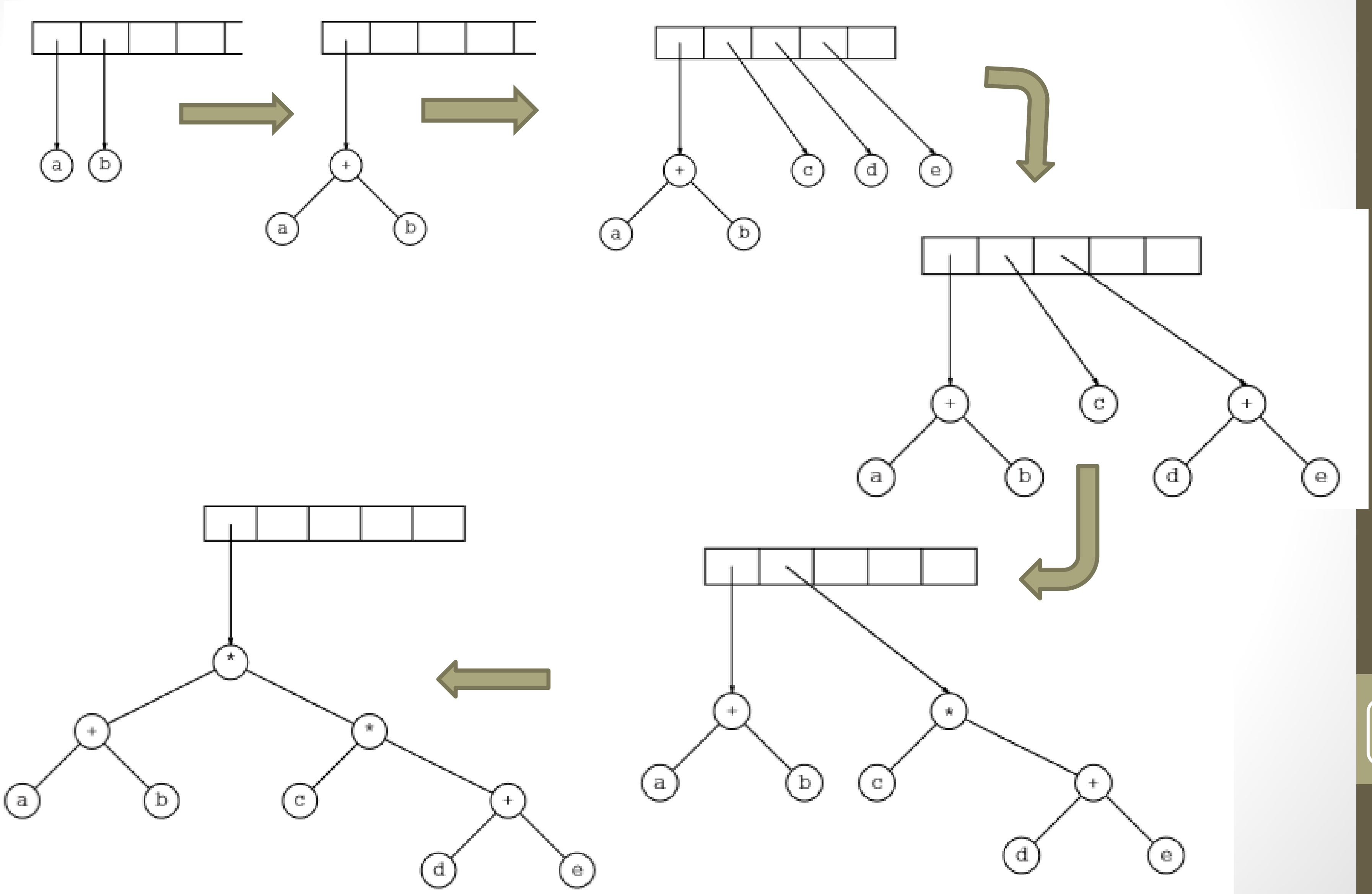


- Infix: ((8 - 5) * ((4 + 2) / 3))
- Prefix: * - 8 5 / + 4 2 3
- Postfix: 8 5 - 4 2 + 3 / *

Creating expression tree from postfix expression

- Read the expression one symbol at a time.
- If the symbol is an operand,
  - we create a one-node tree and push a pointer to it onto a stack.
- If the symbol is an operator,
  - we pop (pointers) to two trees T1 and T2 from the stack (T1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T2 and T1, respectively.
- A pointer to this new tree is then pushed onto the stack.

Construct expression tree for ab+cde+**



Creation of binary tree using traversal

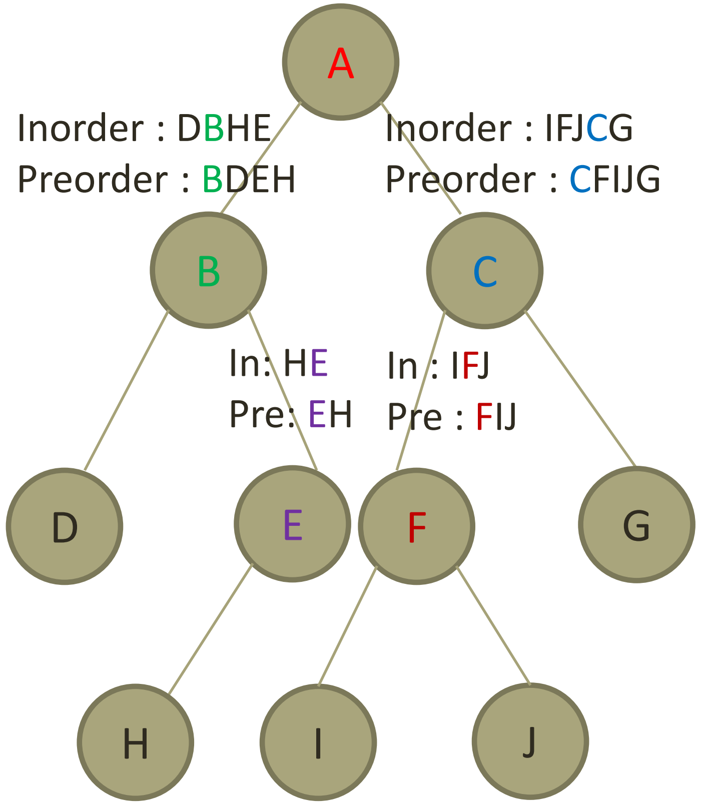
- Binary tree can be constructed from combination of-
  - Preorder and Inorder traversal
  - Postorder and Inorder traversal

- **For given Preorder and Inorder traversal :**

1. Scan Preorder traversal from L -> R
2. For each scanned node ‘X’, locate its position in Inorder traversal
3. The node preceding node ‘X’ in Inorder traversal forms the left subtree and node succeeding ‘X’ forms the right subtree
4. Repeat step 1 for each symbol in the preorder.

- For Postorder and Inorder, the the last node will be the root node. (i.e., scan the Postorder traversal from R->L)

- Preorder : ABDEHCFIJG
- Inorder : DBHEAIFJCG



Applications of trees

- Trees are used to store simple as well as complex data. Here simple means an int value, char value and complex data (structure).
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- A self-balancing tree, Red-black tree is used in kernel scheduling to preempt massively multi-processor computer operating system use.
- Another variation of tree, B-trees are used to store tree structures on disc. They are used to index a large number of records.
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
- Trees are used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables.

Huffman tree and its application

- Developed by David A. Huffman in 1952 A.D.
- Huffman tree is a strict binary tree with minimum weighted path length.
- One of the most important applications of binary tree is in Communication .
- Used in lossless data compression that uses variable length encoding for a character based on its estimated probability of occurrence.

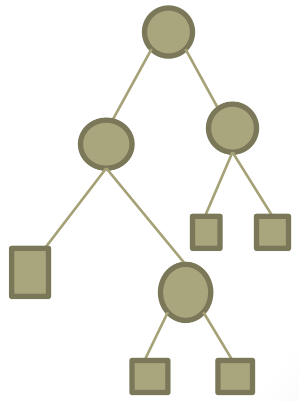
Huffman tree

- For a strictly binary tree, for 'n' leaves/external nodes, there will be total 2n-1 nodes.

- External nodes = Internal nodes +1
- External path length (L_E) = sum of all path to leaves

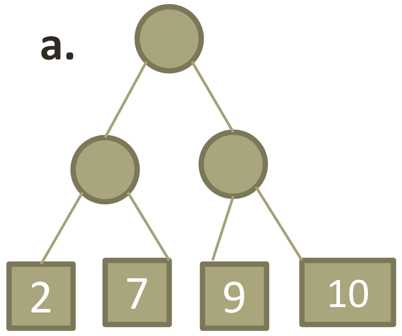
  = 2+2+2+3+3 = 12
- Internal path length (L_I) = sum of all path to int. nodes

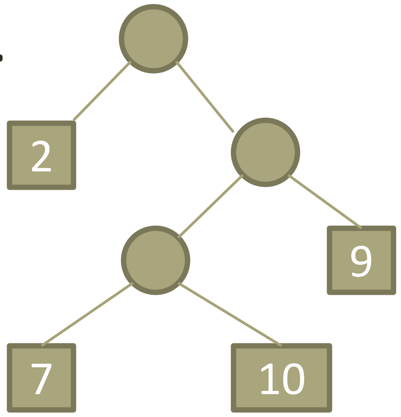
  =0+1+1+2 = 4
- Hence, $L_E = L_I + 2n_i$

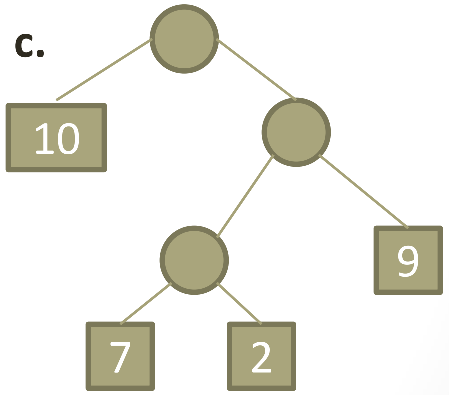


Huffman tree

- If external nodes have weight 'W', then weighted path length will be,
- $\sum_{i=1}^{n} W_i L_i$
- Since, $P = W_1L_1 + W_2L_2 + \ldots W_nL_n$
- Lets assume weight be 2, 7, 9, 10 . Then tree can be,

a.


b.


c.


- $P_a = 2*2 + 7*2 + 9*2 + 10*2 = 56$
- $P_b = 2*1 + 7*3 + 9*2 + 10*3 = 71$
- $P_c = 2*3 + 7*3 + 9*2 + 10*1 = 55$

Huffman tree

- That means, we can create different forms of tree for same given weights.
- Path length of those trees will be different .
- Although the tree structures are same, path length can differ (as in b. and c.)
- Solution for this problem is creating a **Huffman tree** to find **minimum path length**.

Huffman tree (Algorithm)

- For the ‘n’ external nodes with W_n weights , Huffman algorithm is,
  - 1. Suppose there are ‘n’ weights W_1, W_2, ……. , W_n
  - 2. Create a priority queue for it In which highest priority is given to lowest weight (optional).
  - 3. Take two minimum weighted nodes to form a new node(for e.g., if W_1 and W_2 are minimum weighted nodes, they will form a new node $W_1 + W_2$)
  - 4. Now , add new node , say $W_1 + W_{2\,,}$ to the queue so that remaining nodes will be $W_1 + W_2$, W_3, ….. , W_n
  - 5. Create all subtree at last weight.

- This is bottom up approach to create a tree
- Here , the leaves contains actual characters and internal nodes stores the weight and link to the child nodes.

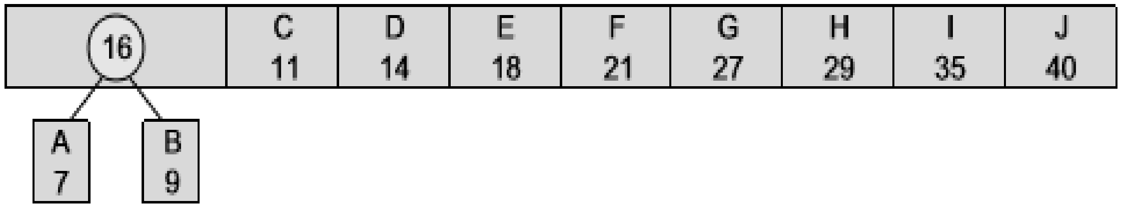
Huffman tree (Example)

Create a Huffman tree with the following nodes.

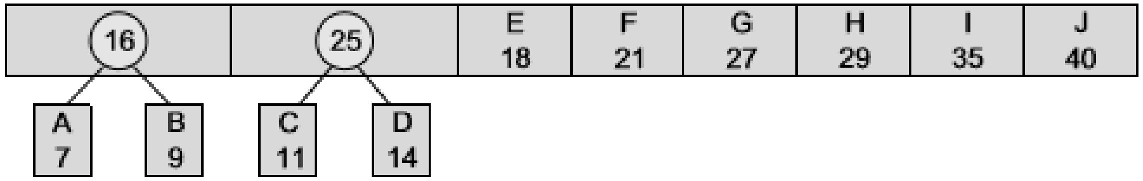
A	B	C	D	E	F	G	H	I	J
7	9	11	14	18	21	27	29	35	40

Solution:

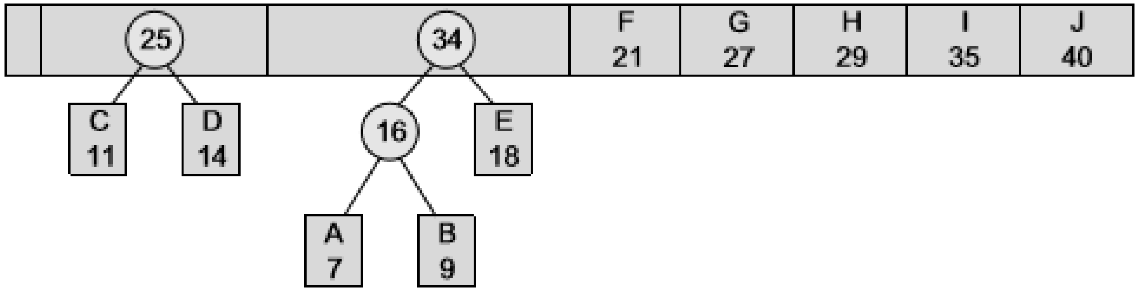
1.



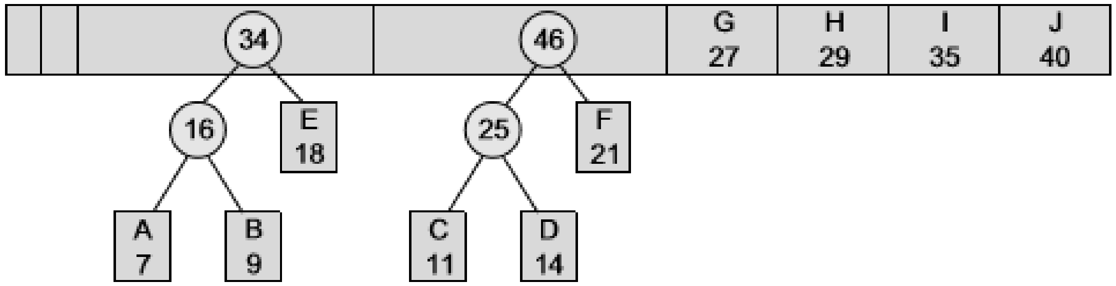
2.



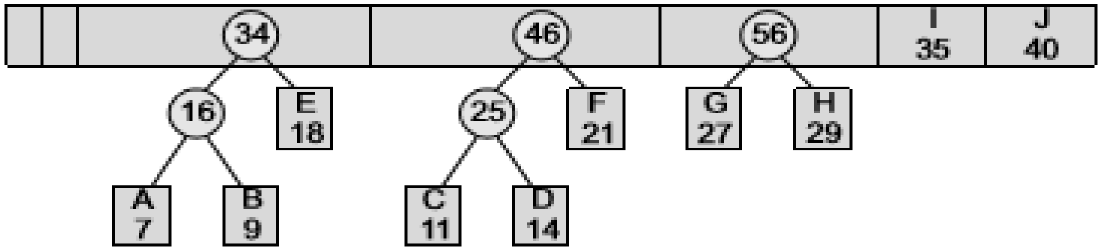
3.



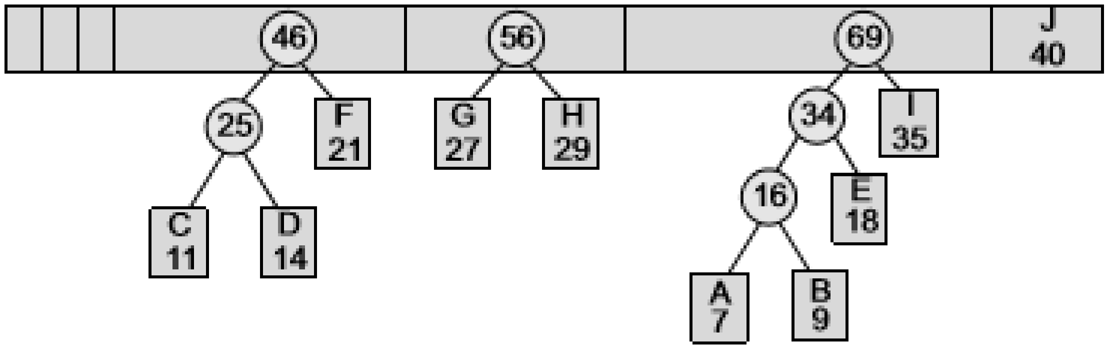
4.


34
46
G 27
H 29
I 35
J 40
16
E 18
25
F 21
A 7
B 9
C 11
D 14

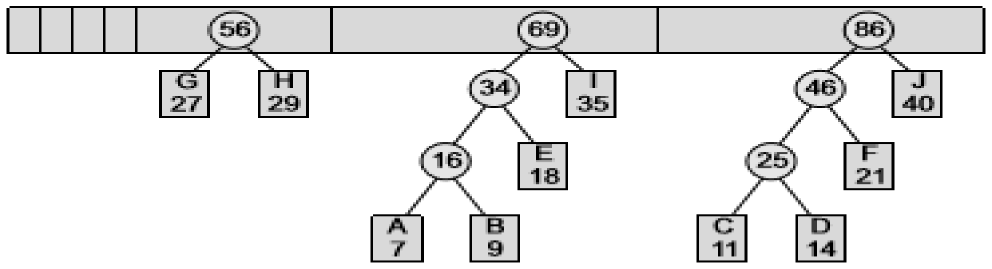
5.


34
46
56
I 35
J 40
16
E 18
25
F 21
G 27
H 29
A 7
B 9
C 11
D 14

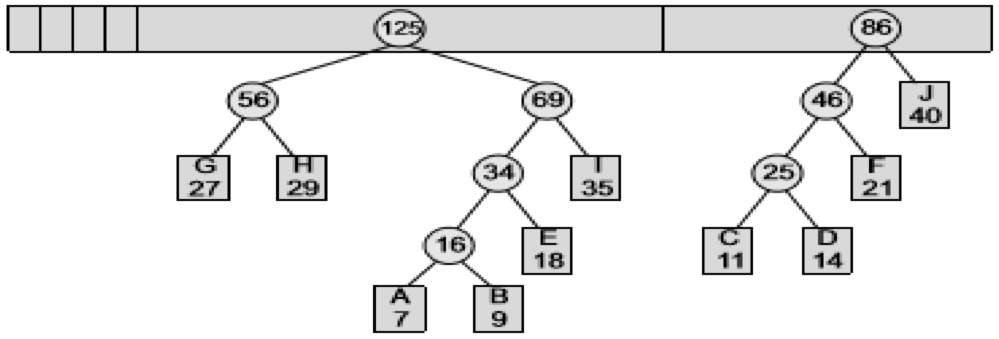
6.


46
56
69
J 40
25
F 21
G 27
H 29
34
I 35
C 11
D 14
16
E 18
A 7
B 9

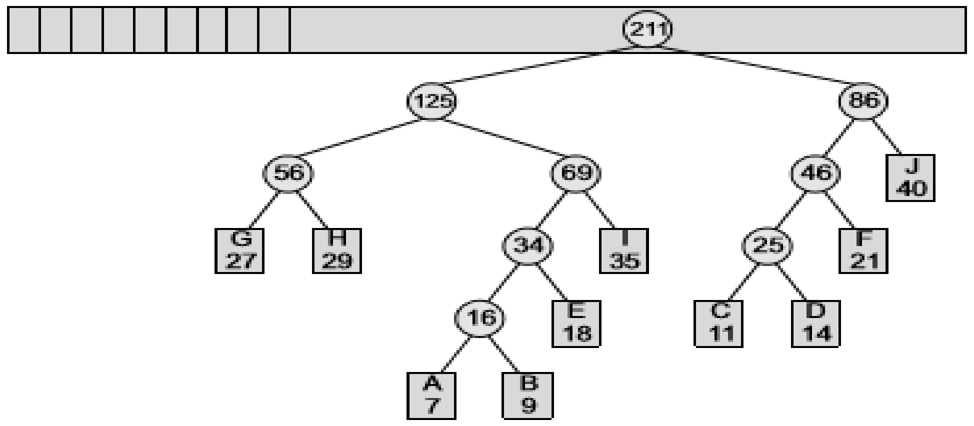
7.


56
69
86
G 27
H 29
34
I 35
46
J 40
16
E 18
25
F 21
A 7
B 9
C 11
D 14

8.


125
86
56
69
46
J 40
G 27
H 29
34
I 35
25
F 21
16
E 18
C 11
D 14
A 7
B 9

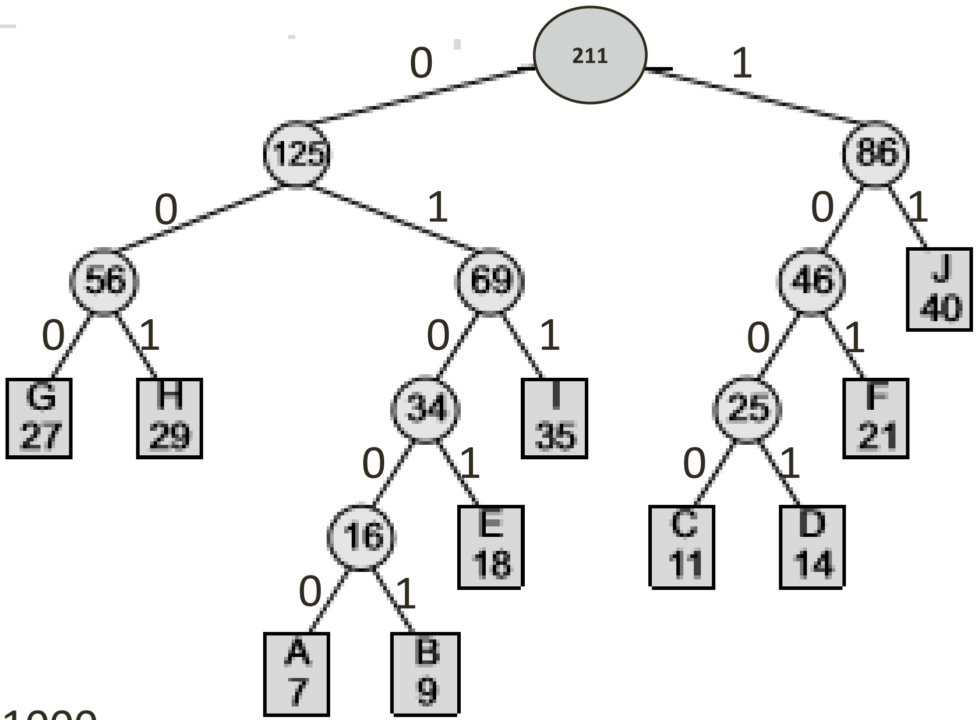
9.


211
125
86
56
69
46
J 40
G 27
H 29
34
I 35
25
F 21
16
E 18
C 11
D 14
A 7
B 9

Huffman coding

- It is a variable length coding based on estimated probability of occurrence of each possible value of source character
- If 26 character set (of alphabets) is to be represented in binary code, then , we need $2^5<26<2^4$ i.e., 5 bits
- It will be inefficient if all character uses 5-bit representation (fixed length coding)
- So, the character with highest probability of occurrence is assigned less number of bits in variable length coding

- **For Huffman coding:**
- 1. Generate a Huffman tree.
- 2. Every left branch is coded with ‘0’ and right branch with ‘1’
- 3. To generate code, visit root to leaf through coded branch.



A(7) = 01000
B(9) = 01001
C(11) = 1000
D(14) = 1001
E(18) = 0101
F(21) = 101
G(27) = 000
H(29) = 001
I(35) = 011
J(40) = 11

Balanced binary trees

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least Θ(log N)
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

AVL Trees

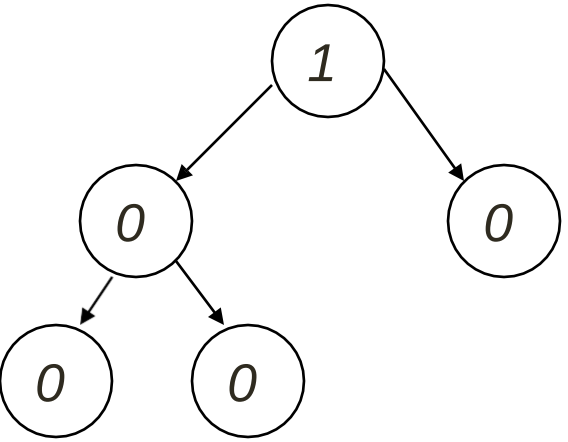
- AVL tree is a self-balancing binary search tree in which the heights of the two sub-trees of a node may differ by at most one. Because of this property, AVL tree is also known as a height-balanced tree.
- The key advantage of using an AVL tree is that it takes O(logn) time to perform search, insertion and deletion operations in average case as well as worst case (because the height of the tree is limited to O(logn)).

AVL Trees

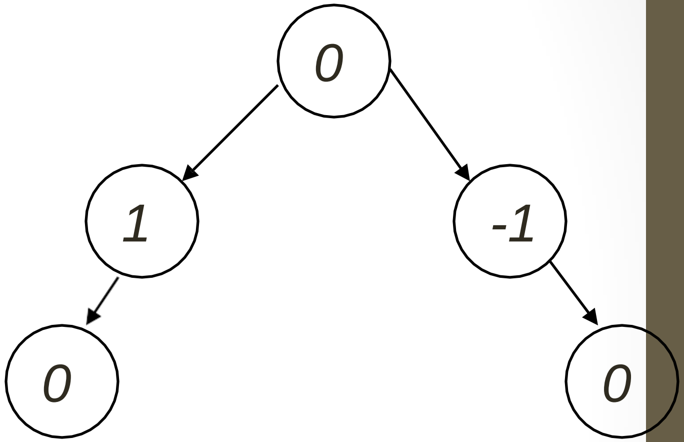
- The structure of an AVL tree is same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the *BalanceFactor*.
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

  Balance factor = Height (left sub-tree) – Height (right sub-tree)
- A binary search tree in which every node has a balance factor of -1, 0 or 1 is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing.

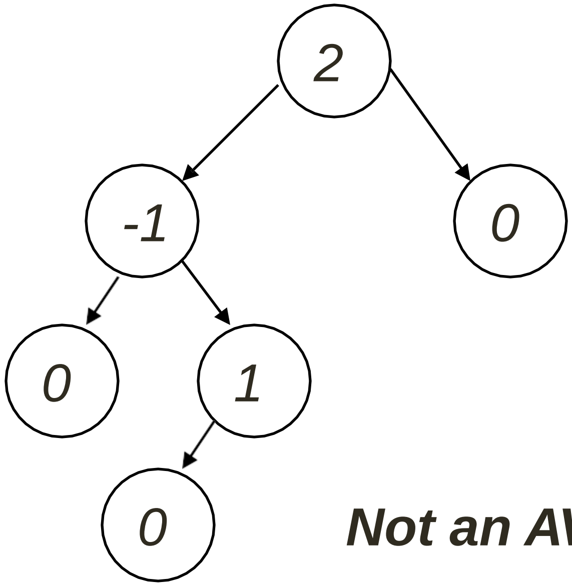
AVL Trees



AVL Tree



AVL Tree



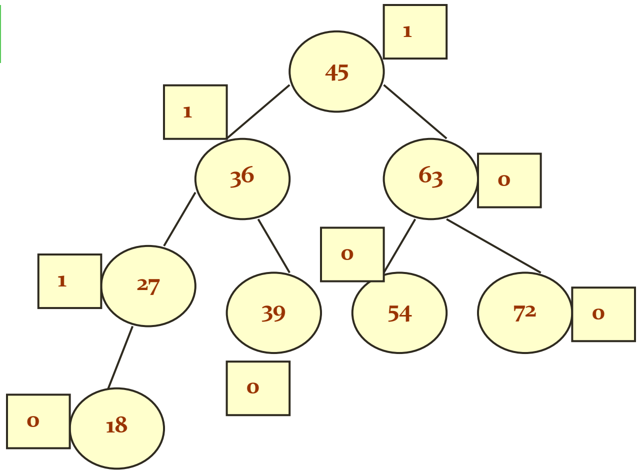
Not an AVL Tree

AVL Trees

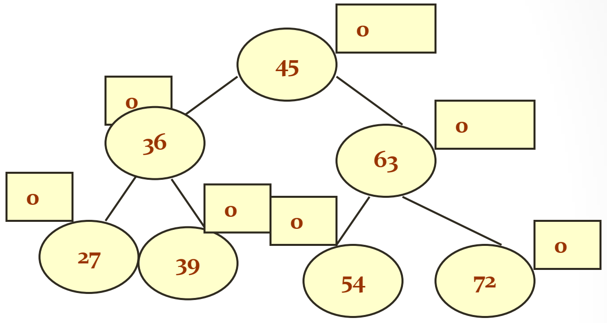
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called *Left-heavy tree*.
- If the balance factor of a node is 0, then it means that the height of the left sub-tree is equal to the height of its right sub-tree.
- If the balance factor of a node is -1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called Right-*heavy tree*.

AVL Trees

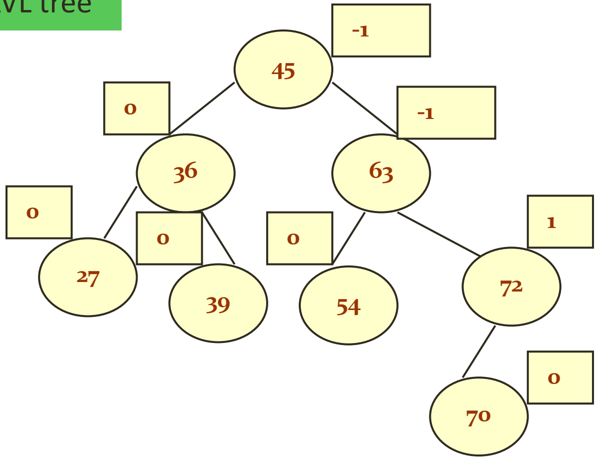
Left heavy AVL tree



Balanced AVL tree



Right heavy AVL tree



Searching in AVL Tree

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes O(log *n*) time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken.

Insertion in AVL Tree

- Since an AVL tree is also a variant of binary search tree(BST), insertion is also done in the same way as it is done in case of a BST.
- Like in BST, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation.
- Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still -1, 0 or 1, then rotations are not needed.

Insertion in AVL Tree

- During insertion, the new node is inserted as the leaf node, so it will always have balance factor equal to zero.
- The nodes whose balance factors will change are those which lie on the path between the root of the tree and the newly inserted node.
- The possible changes which may take place in any node on the path are as follows:
  - Initially the node was either left or right heavy and after insertion has become balanced.
  - Initially the node was balanced and after insertion has become either left or right heavy.
  - Initially the node was heavy (either left or right) and the new node has been inserted in the heavy sub-tree thereby creating an unbalanced sub-tree. Such a node is said to be a critical node.

Rotations to balance AVL Tree

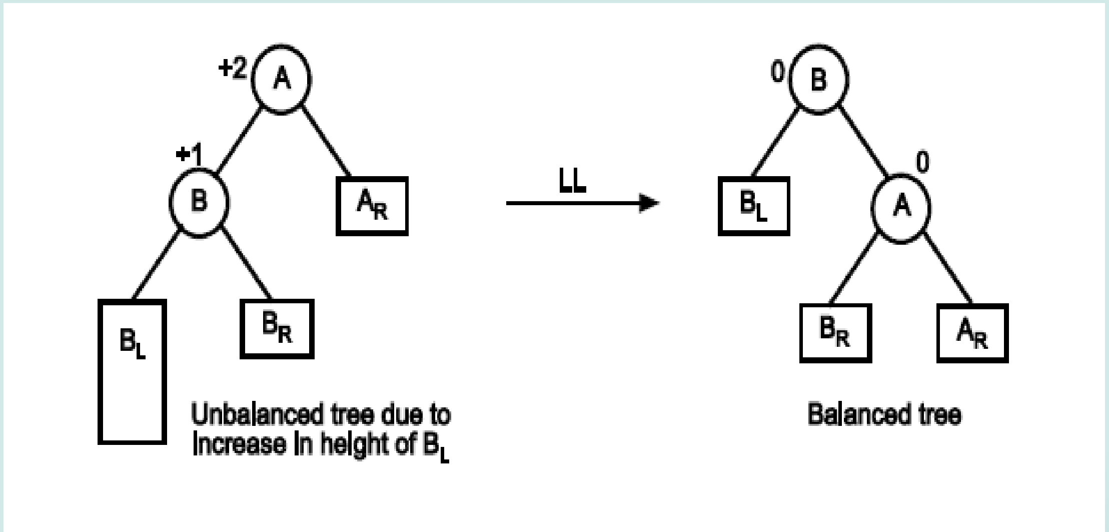
- To perform rotation, our first work is to find the *critical node*. Critical node is the nearest ancestor node on the path from the root to the inserted node whose balance factor is neither -1, 0 nor 1.
- The second task is to determine which type of rotation has to be done.
- There are four types of rebalancing rotations and their application depends on the position of the inserted node with reference to the critical node.

Rotations to balance AVL Tree

- *LL rotation:* the new node is inserted in the left sub-tree of the left sub-tree of the critical node
- *RR rotation:* the new node is inserted in the right sub-tree of the right sub-tree of the critical node
- *LR rotation:* the new node is inserted in the right sub-tree of the left sub-tree of the critical node
- *RL rotation:* the new node is inserted in the left sub-tree of the right sub-tree of the critical node

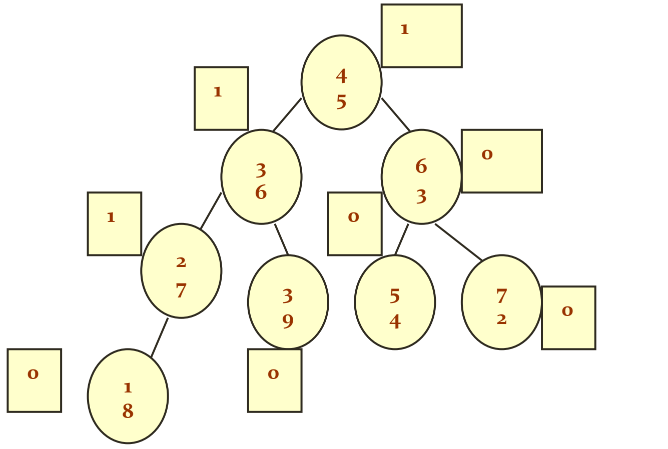
LL Rotation

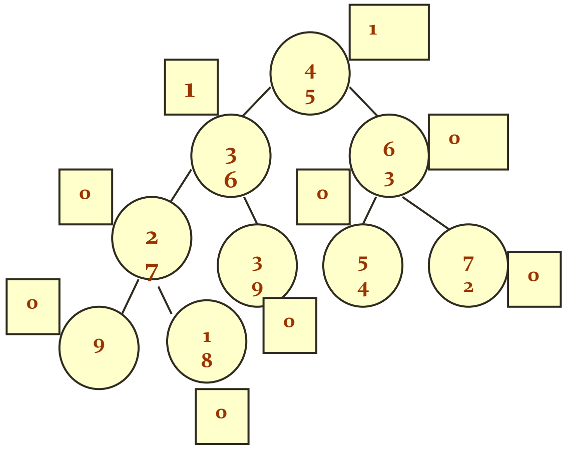
- Imbalance occurs when new node is inserted **at left of left subtree** of critical node.
- Perform single clockwise(right) rotation.
- During rotation :
  - Replace A by B by rotating clockwise rotation along B.[A <- B]
  - Attach A as right subtree of B. [right(B) <- A]
  - Re-attach right-subtree B_R of B as left-subtree of A. [left(A) <- B_R]

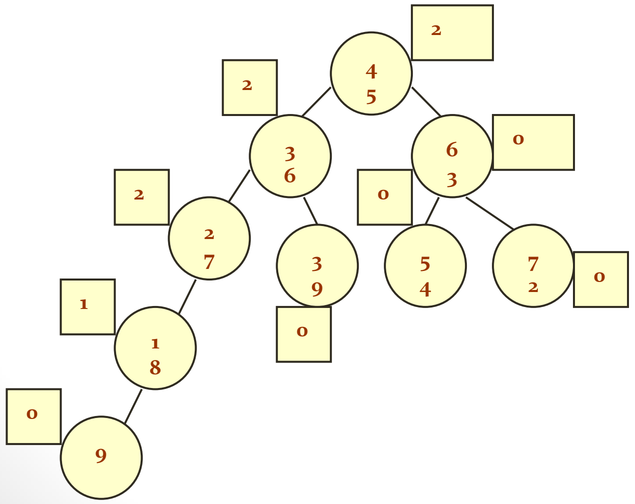


LL Rotation

Example: Consider the AVL tree given below and insert 9 into it.



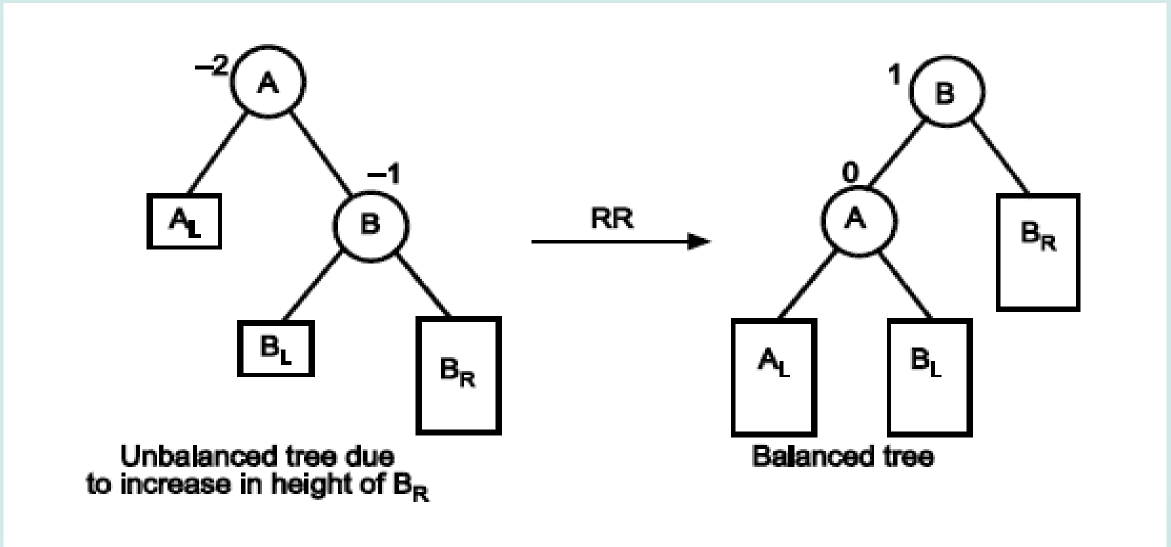




The tree is balanced using LL rotation

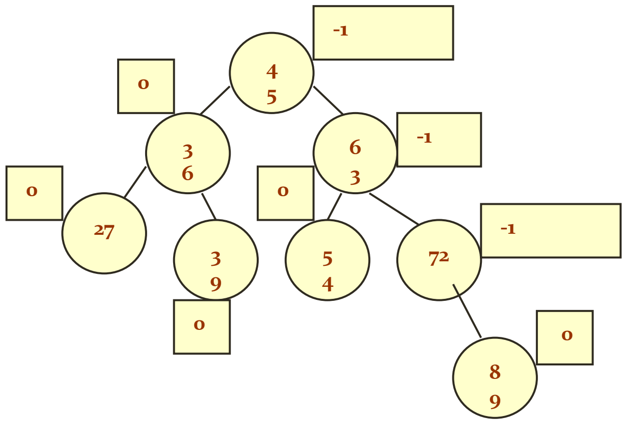
RR Rotation

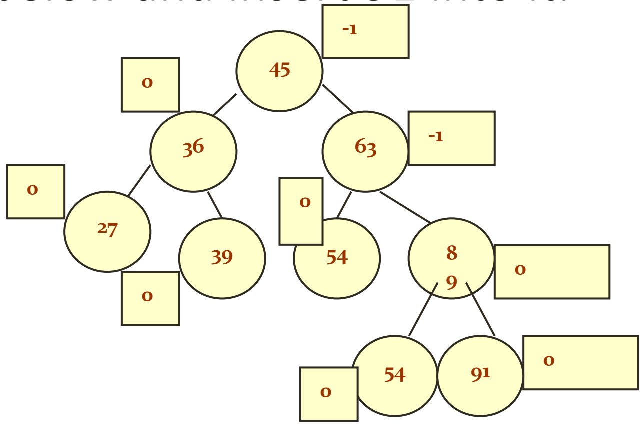
- Imbalance due to insertion of new **node at right of right subtree** of critical node.
- Perform single counter-clockwise (left) rotation
- During rotation :
  - Replace A by B by rotating counter-clockwise rotation along B.[A <- B]
  - Attach A as left subtree of B. [left(B) <- A]
  - Re-attach left-subtree B_L of B as right-subtree of A. [right(A) <- B_L]

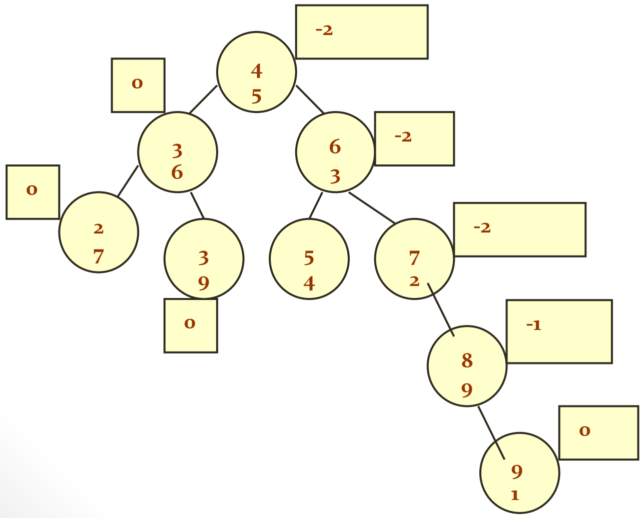


RR Rotation

- Example: Consider the AVL tree given below and insert 91 into it.



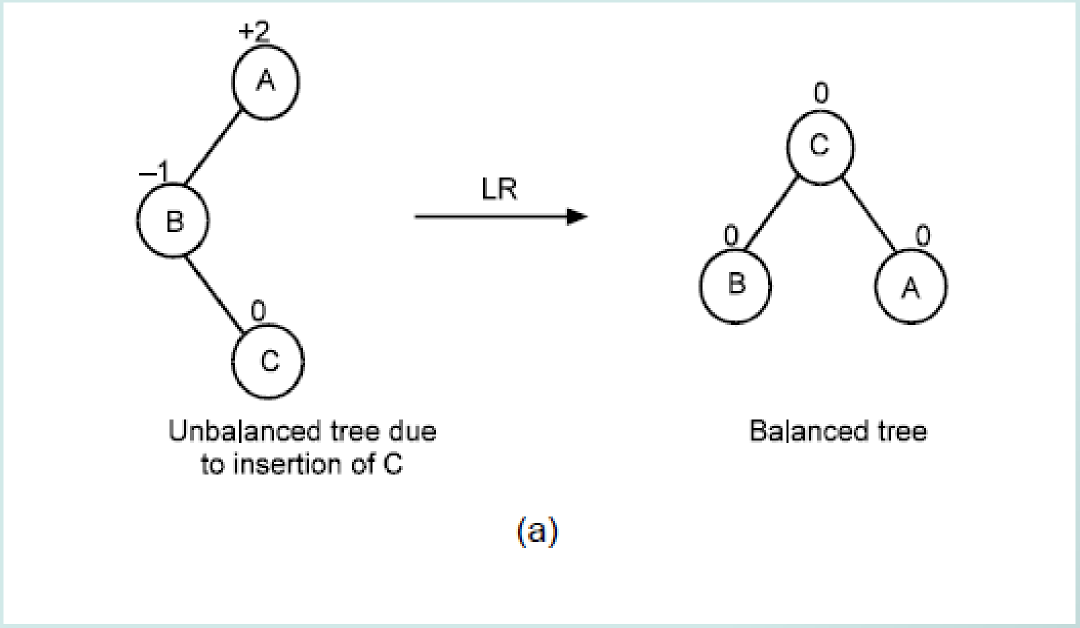


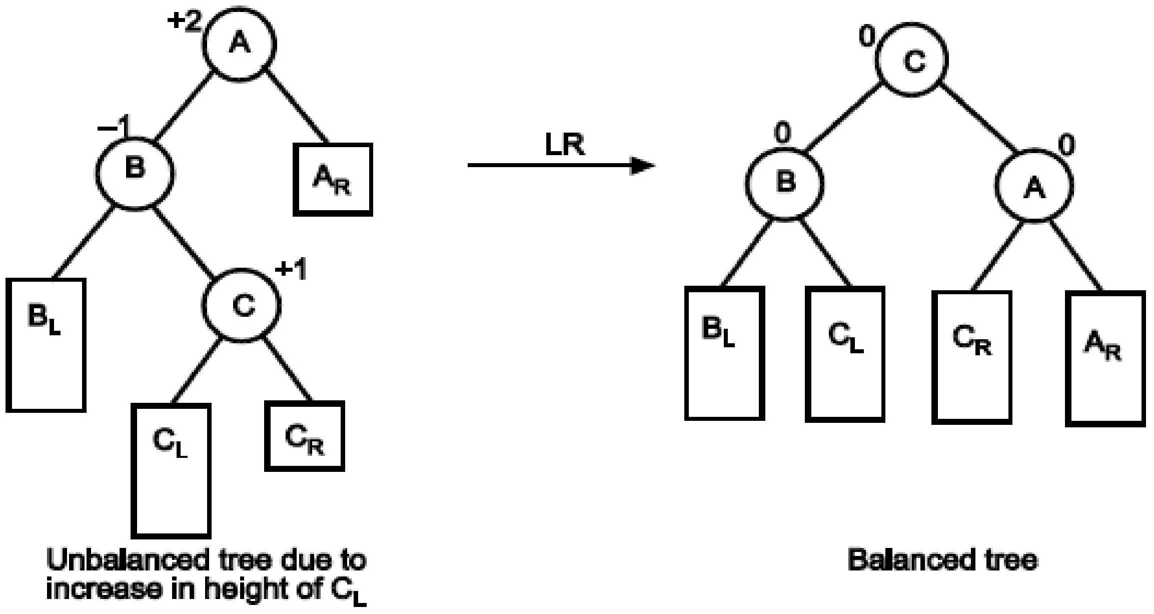


The tree is balanced using RR rotation

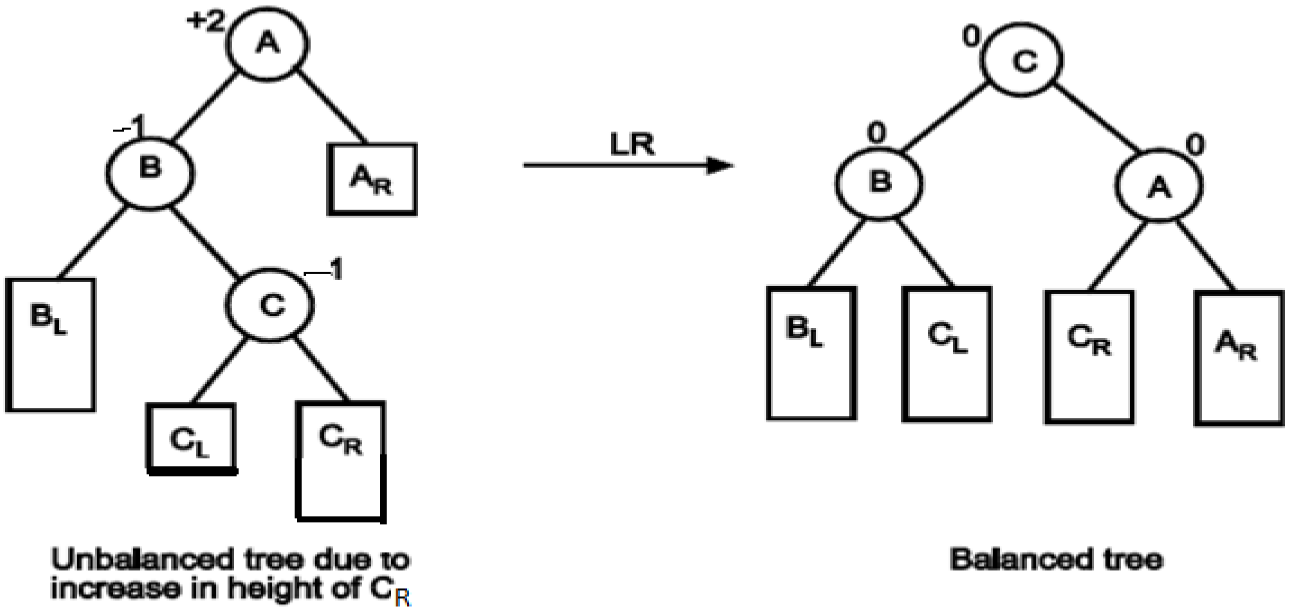
LR Rotation

- Imbalance due to insertion of new node at right sub-tree of left child of critical node.
- Perform double rotation - RR (along B) followed by LL (along A) rotation.
- During rotation :
  - Right (B) <- left (C)
  - left(A) <- right (C)
  - Left (C) <- B
  - Right (C) <-A




+2
A
–1
B
A_R
+1
C
B_L
C_L
C_R
LR
0
C
0
B
0
A
B_L
C_L
C_R
A_R
Unbalanced tree due to increase in height of C_L
Balanced tree

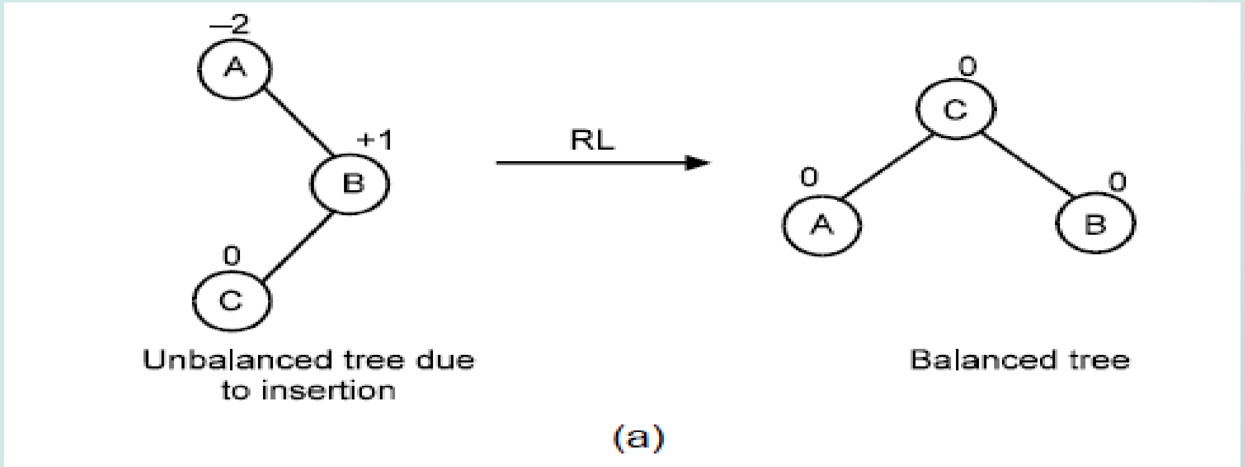
(b)

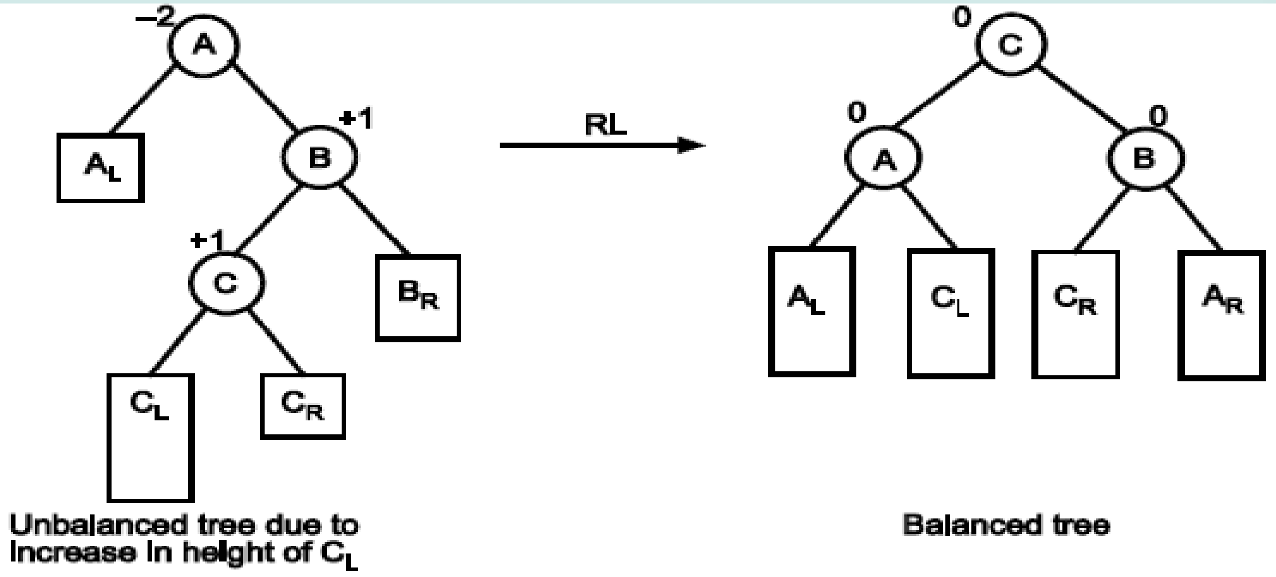

+2
A
–1
B
A_R
–1
C
B_L
C_L
C_R
LR
0
C
0
B
0
A
B_L
C_L
C_R
A_R
Unbalanced tree due to increase in height of C_R
Balanced tree

(c)

RL Rotation

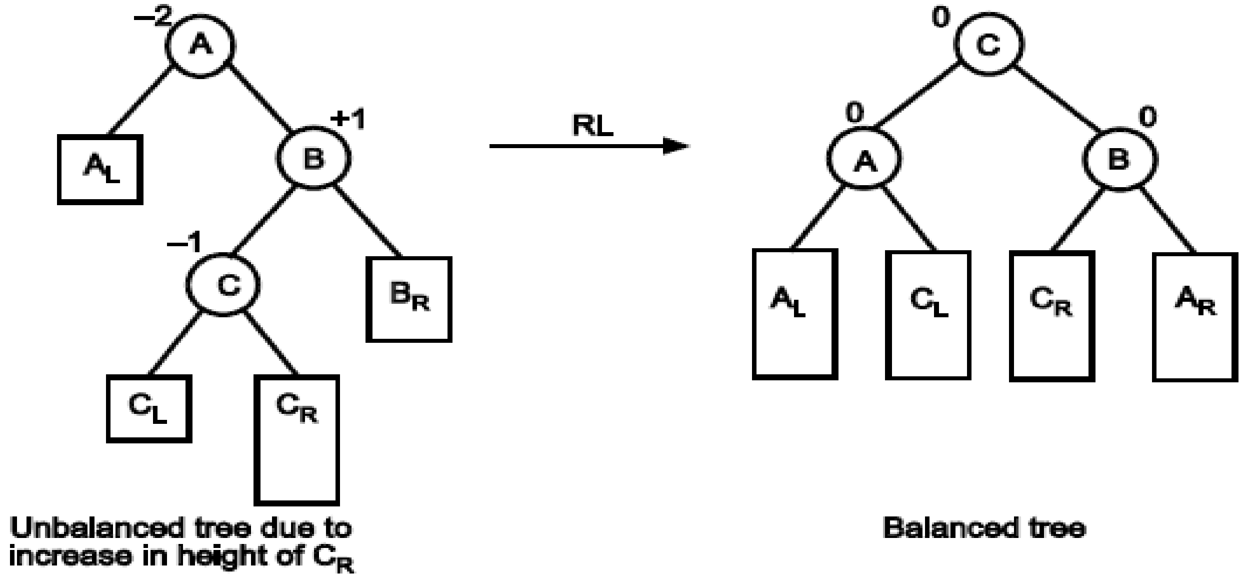
- Imbalance due to insertion of new node at left sub-tree of right child of critical node.
- Perform double rotation - LL (along B) followed by RR (along A) rotation.
- During rotation :
  - Right (A) <- left (C)
  - left(B) <- right (C)
  - Left (C) <- A
  - Right (C) <- B





−2
A
A_L
+1
B
+1
C
B_R
C_L
C_R
RL
0
C
0
A
0
B
A_L
C_L
C_R
A_R
Unbalanced tree due to increase in height of C_L
Balanced tree

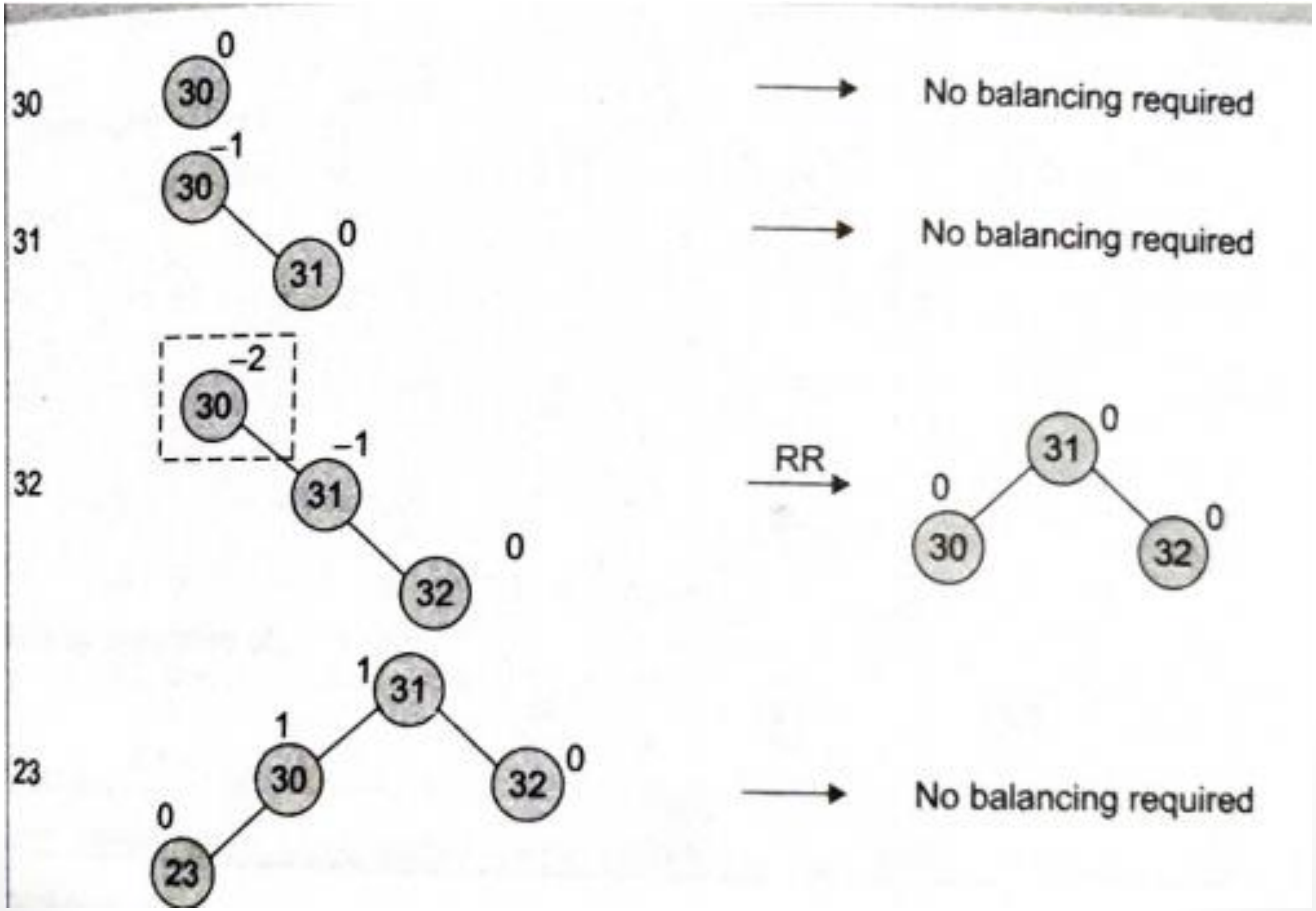
(b)



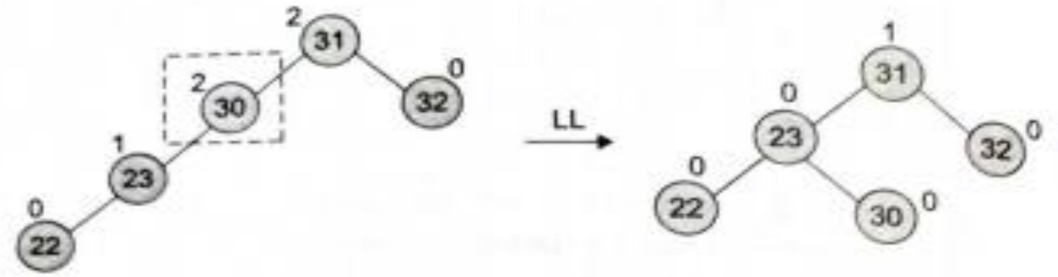
−2
A
A_L
+1
B
−1
C
B_R
C_L
C_R
RL
0
C
0
A
0
B
A_L
C_L
C_R
A_R
Unbalanced tree due to increase in height of C_R
Balanced tree

(c)

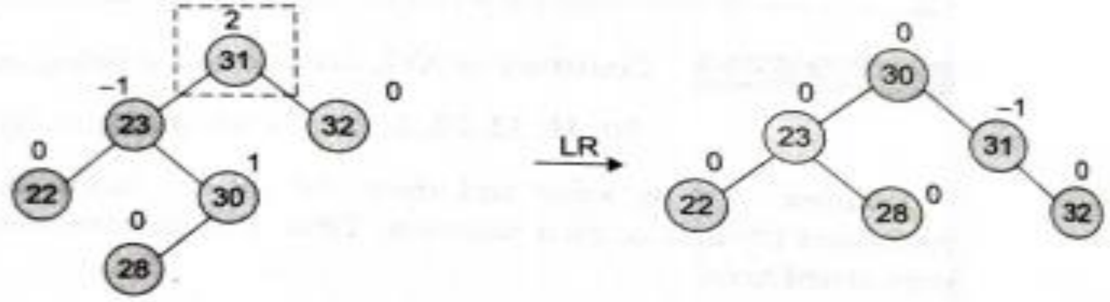
Construct an AVL tree for following data: 30, 31, 32, 23, 22, 28, 24, 29, 26, 27, 34, 36.



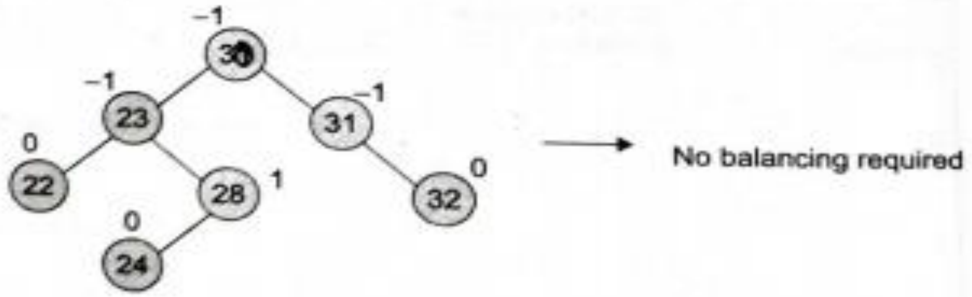
22



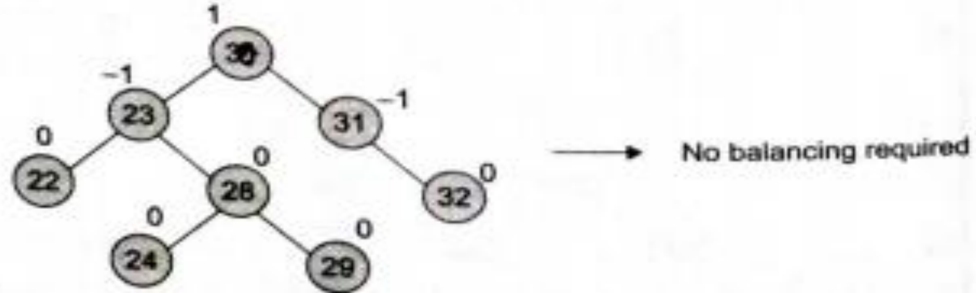
28

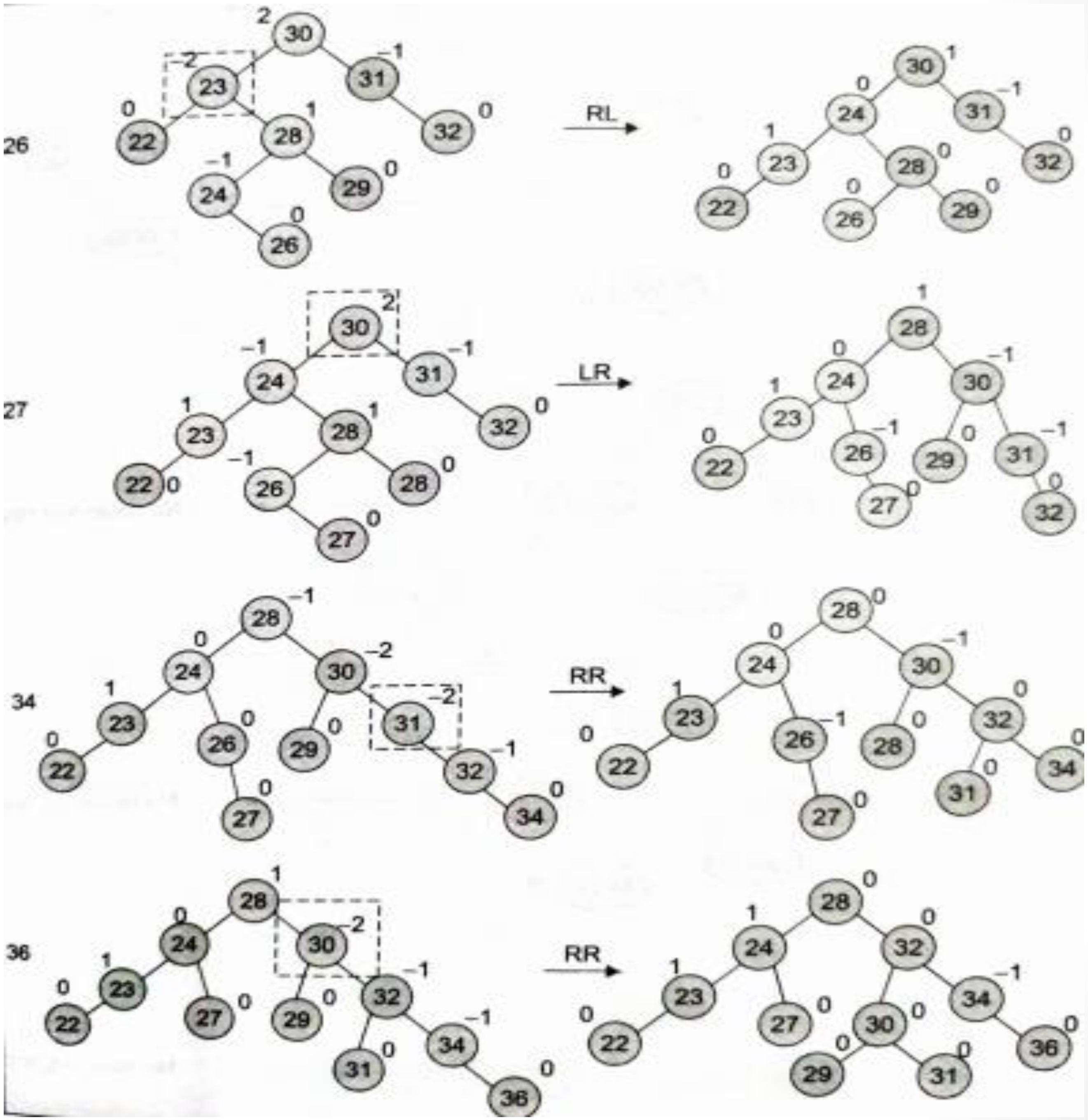


24



29




26
RL
27
LR
34
RR
36
RR

Deletion in AVL tree

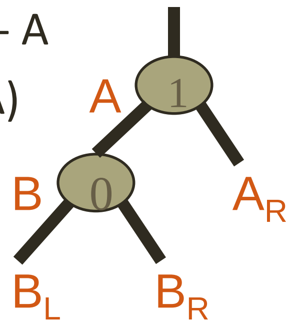
- Deletion of a node in an AVL tree is similar to that of BST.
- But deletion may disturb the AVLness of the tree, so to re-balance the AVL tree we need to perform rotations.
- There are two classes of rotation that can be performed on an AVL tree after deleting a given node: R rotation and L rotation.
- If the node to be deleted is present in the left sub-tree of the critical node, then L rotation is applied else if node is in the right sub-tree, R rotation is performed.
- Further there are three categories of L and R rotations. The variations of L rotation are: L-1, L0 and L1 rotation. Correspondingly for R rotation, there are R0, R-1 and R1 rotations.

R0 Rotation

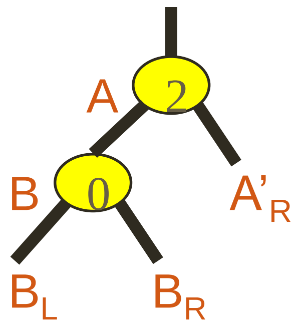
- Used when BF of left subtree of critical node is 0.
- Node to be deleted is at the right subtree of critical node.

- Subtree height is unchanged.
- No further adjustments to be done.
- Similar to LL rotation.

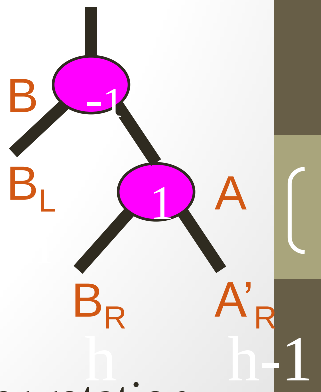
- During rotation :
  - Critical node(A) <- B
  - Right (B) <- A
  - B_R <- left(A)



Before deletion.

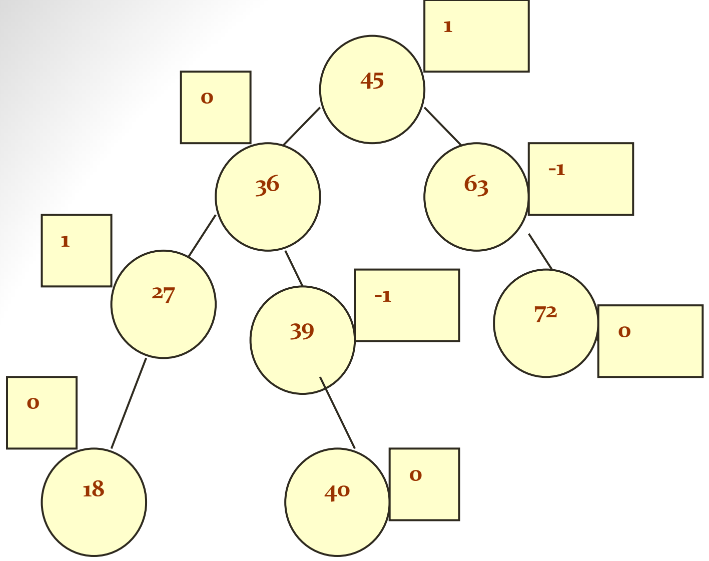


After deletion.

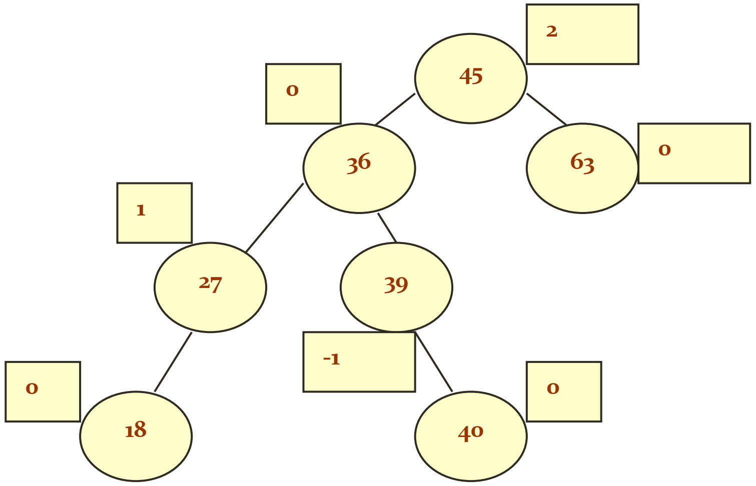


After rotation.

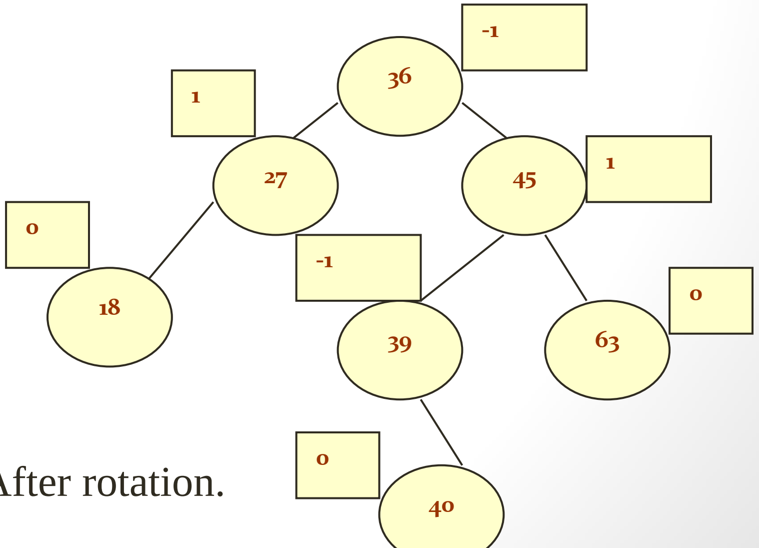
Consider the given AVL tree and delete 72 from it.



Before deletion.



After deletion.



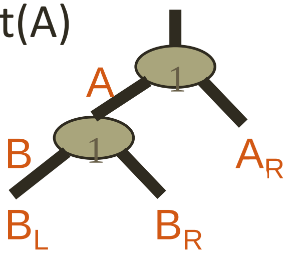
After rotation.

R1 Rotation

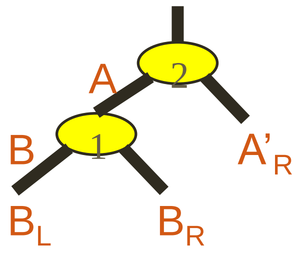
- Used when BF of left subtree of critical node is 1.
- Node to be deleted is at the right subtree of critical node.

- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to LL and R0 rotations.

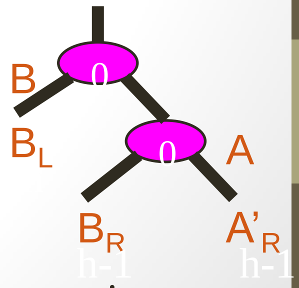
- During rotation :
  - Critical node(A) <- B
  - Right (B) <- A
  - B_R <- left(A)



Before deletion.

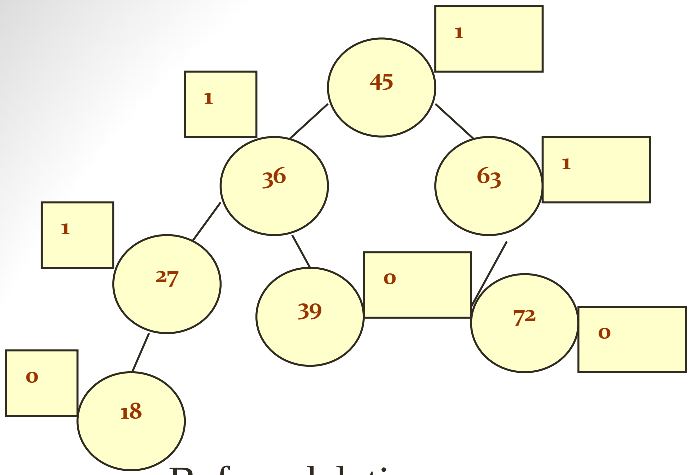


After deletion.

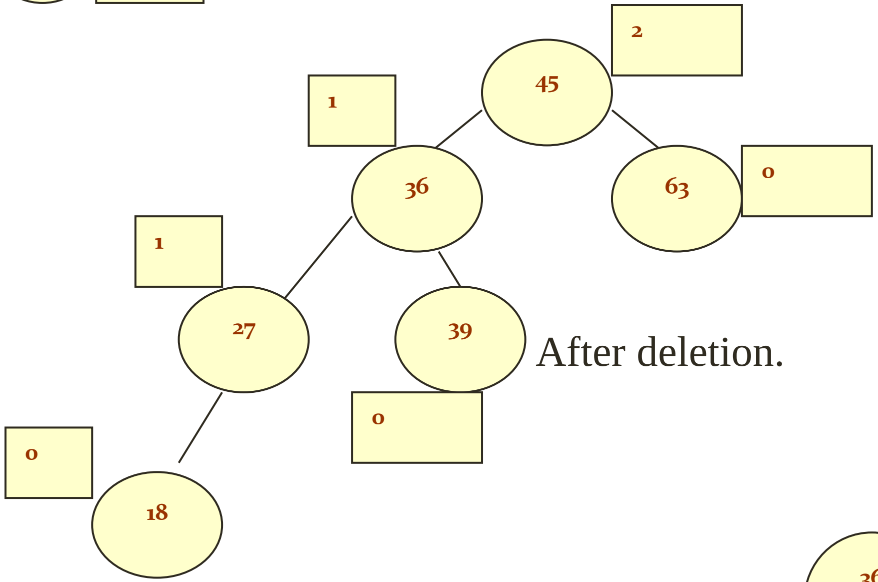


After rotation.

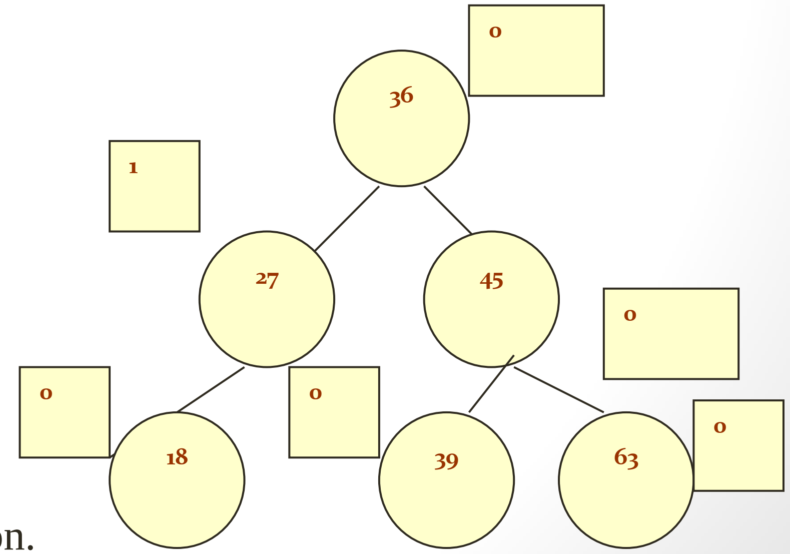
Consider the given AVL tree and delete 72 from it



Before deletion.



After deletion.



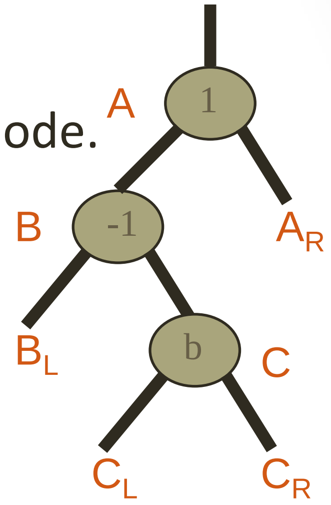
After rotation.

R-1 Rotation

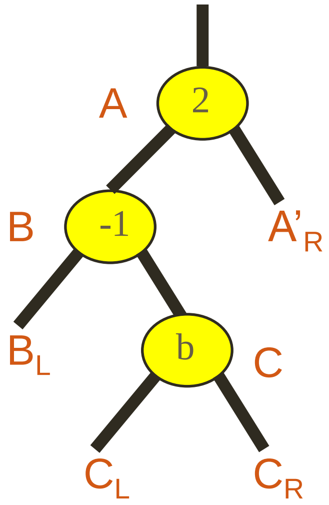
- Used when BF of left subtree of critical node is -1.
- Node to be deleted is at the right subtree of critical node.

- Subtree height is reduced by 1.
- Must continue on path to root.
- Similar to LR.

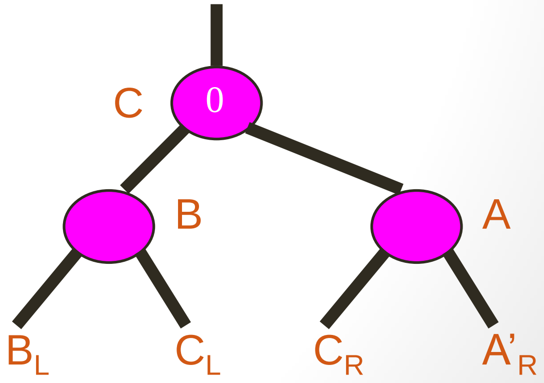
- During rotation :
  - Critical node(A) <- C
  - Right(C)<- A
  - Left (C)<- B
  - Right(B)<-C_L
  - Left(A) <- C_R



Before deletion.

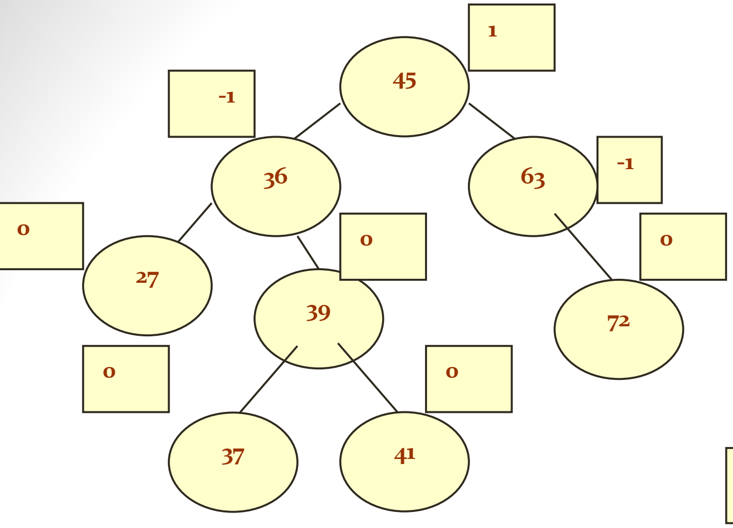


After deletion.

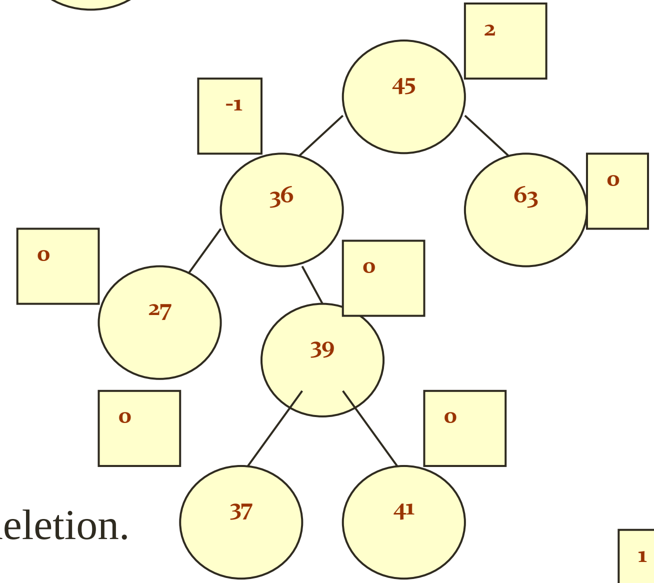


After rotation.

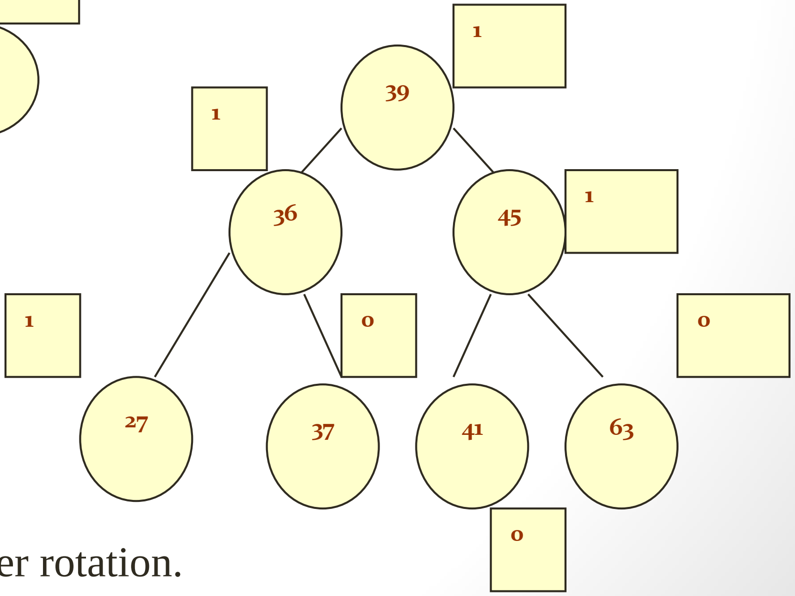
Consider the given AVL tree and delete 72 from it



Before deletion.



After deletion.



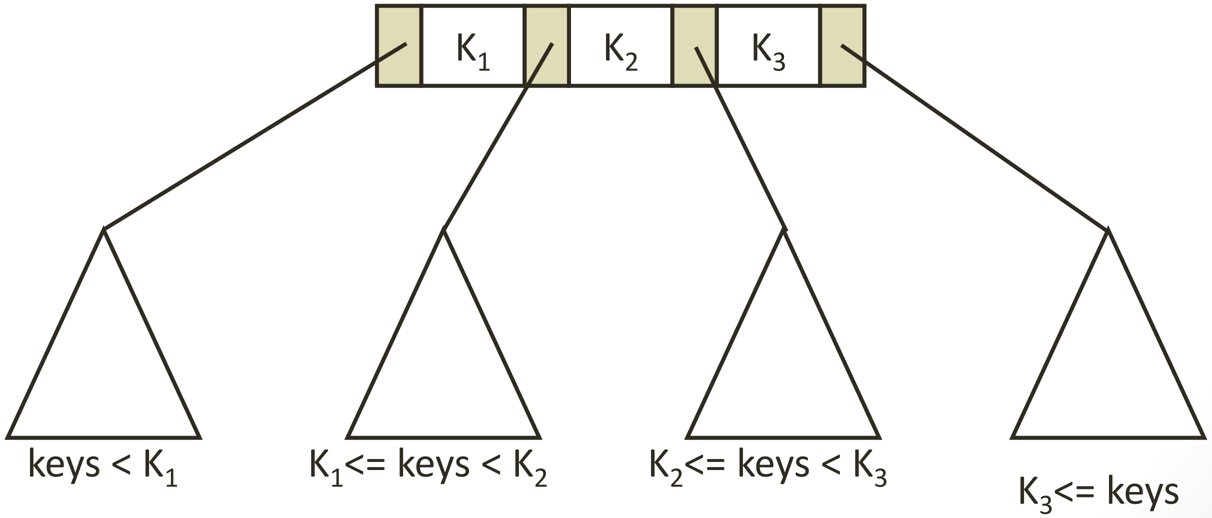
After rotation.

L0, L1 and L-1 Rotations

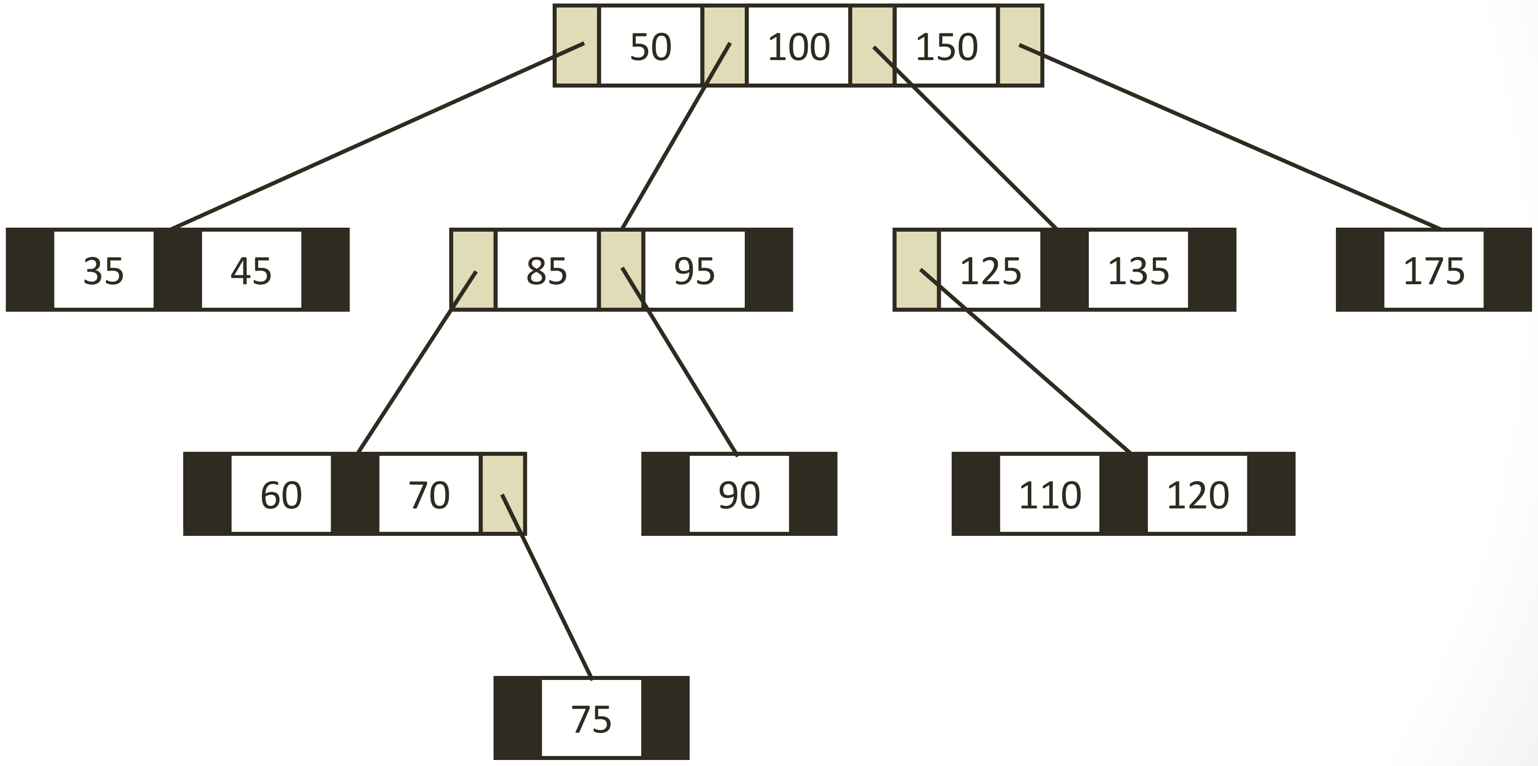
- Used when BF of right subtree of critical node is either 0, 1 or -1 and node to be deleted will be at left subtree of critical node.
- LO ≡ RR rotation
- L1 ≡ RR rotation
- L-1 ≡ RL rotation

M-Way Search Trees

- Tree whose out degree is not restricted to 2 while retaining the general properties of binary search trees.
- Each node has m - 1 data entries and m subtree pointers.
- The key values in a subtree
  - >= the key of the left data entry
  - <   the key of the right data entry.



M-Way Search Trees



B-Trees

- M-way trees are unbalanced.
- Bayer, R. & McCreight, E. (1970) created B-Trees.

- A B-tree is a specialized m- way tree that is widely used for disk access. A B tree of order m can have maximum m-1 keys and m pointers to its sub-trees. A B-tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

P_0	K_0	P_1	K_1	P_2	K_2	………	P_{n-1}	K_{n-1}	P_n

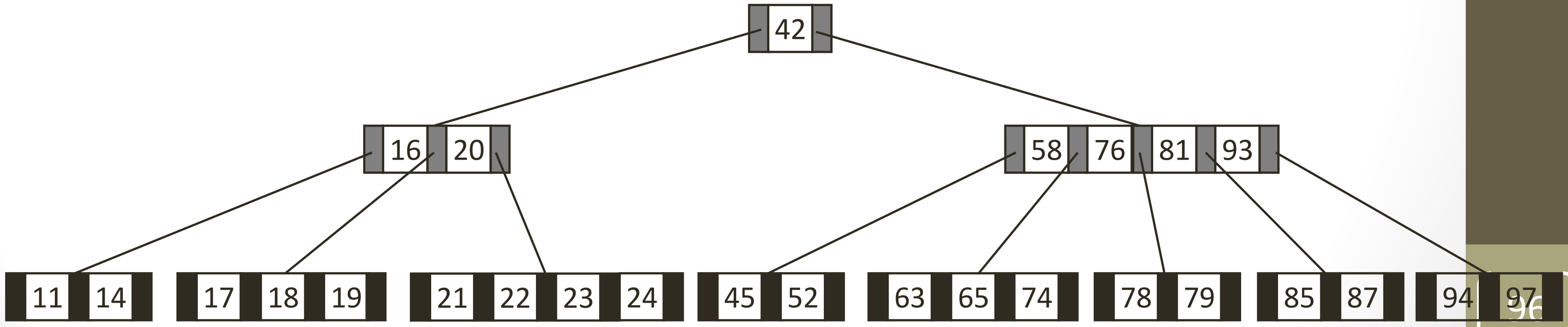
- In the structure, P0, P1, P2,…, Pn are pointers to the node's sub-trees and K0, K1, K2,…, Kn-1 are the key values of the node. All the key values are stored in ascending order. That is, Ki < Ki+1 for 0 ≤ i ≤ n-2.

B-Trees

- A B-tree is designed to store sorted data and allows search, insert, and delete operations to be performed in logarithmic amortized time. A B-tree of order *m* (the maximum number of children that each node can have) is a tree with all the properties of an m-way search tree and in addition has the following properties:
  - Every node in the B-tree has at most (maximum) *m* children.
  - Every node in the B-tree except the root node and leaf nodes have at least (minimum) *m/*2 children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short even in a tree that stores a lot of data.
  - The root node has at least two children if it is not a terminal (leaf) node.
  - All leaf nodes are at the same level.
  - An internal node in the B tree can have n number of children, where 0 ≤n ≤ m. it is not necessary that every node has the same number of children, but the only restriction is that the node should have at least m/2 children.

B-Trees

- In summary, a B-tree is an m-way tree has following properties:
  - The root is either a leaf or has at least 2 and at most m subtrees.
  - All internal nodes have at least [m/2] and at most m subtrees.
  - A leaf node has at least [m/2] - 1 and at most m - 1 entries.
  - All leaf nodes are at the same level.

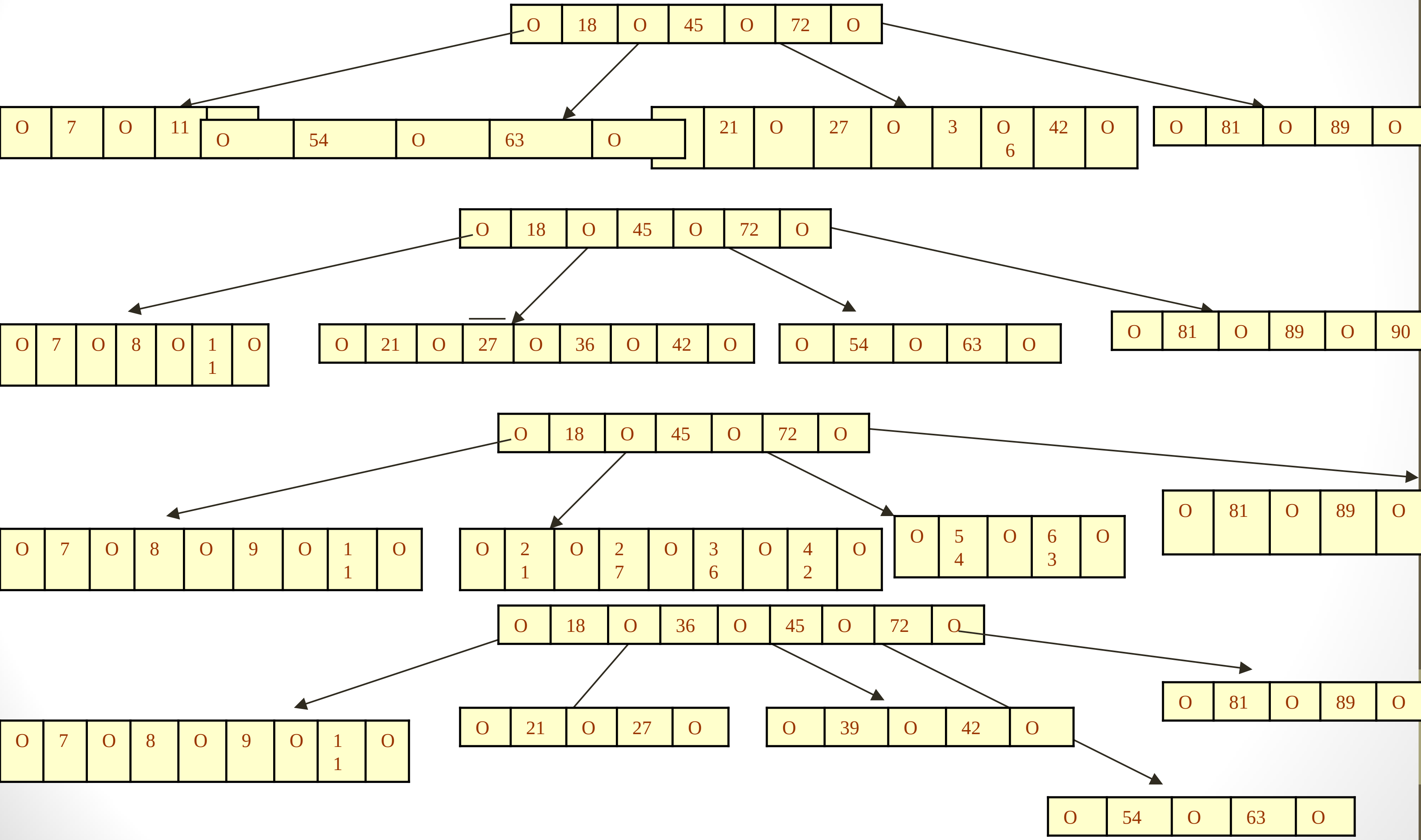


M=5

Insertion in B-tree

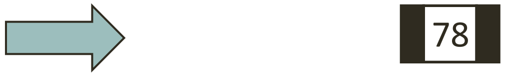
- In a B tree all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.
- 1. Search the B tree to find the leaf node where the new key value should be inserted.
- 2. If the leaf node is not full, that is it contains less than m-1 key values then insert the new element in the node, keeping the node's elements ordered.
- 3. If the leaf node is full, that is the leaf node already contain m-1 key values , then,
  - 3.a. Insert the new value in order into the existing set of keys
  - 3.b.Split the node at its median into two nodes. Note that the split nodes are half full.
  - 3.c.Push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Example: Look at the B tree of order 5 given below and insert 8, 9, 39 into it.

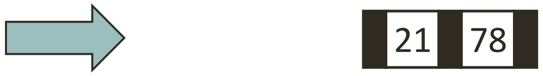


B-Tree Insertion (example)

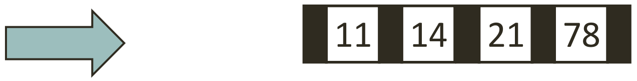
Insert 78



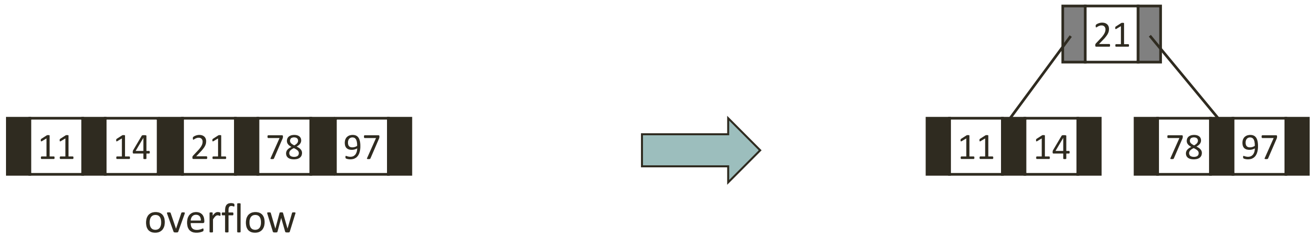
Insert 21



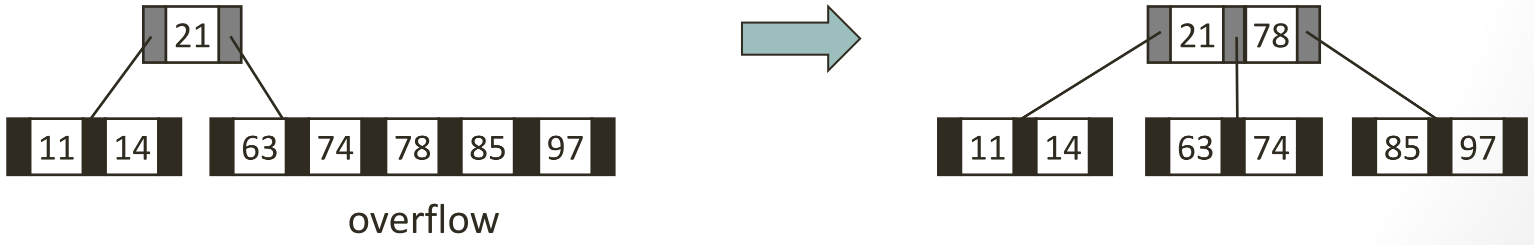
Insert 14, 11



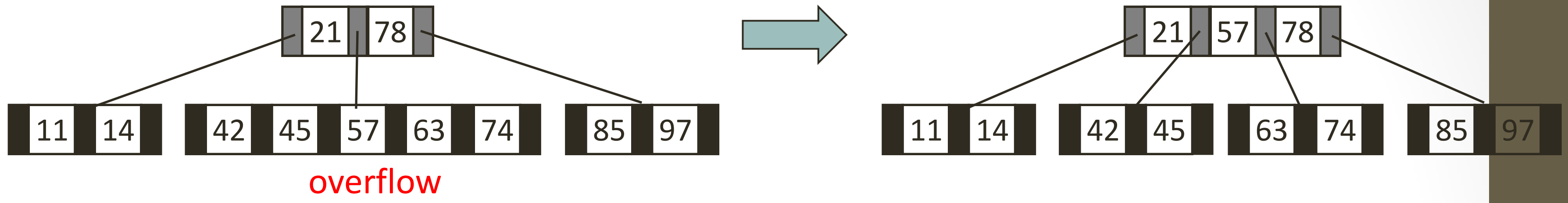
Insert 97



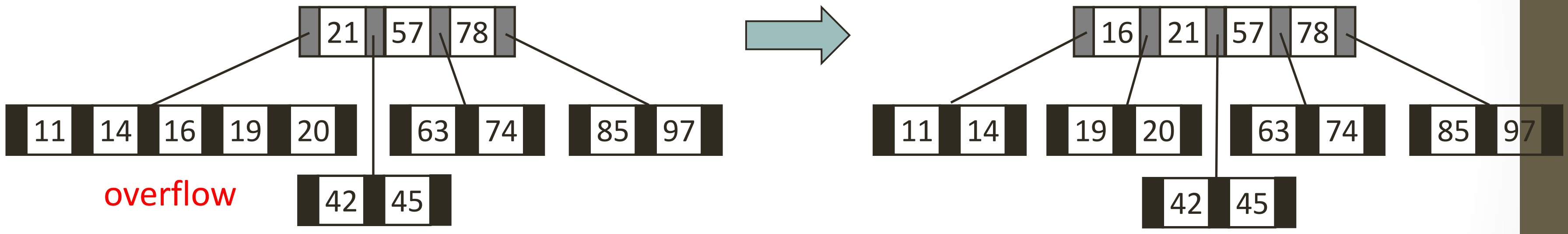
Insert 85, 74, 63



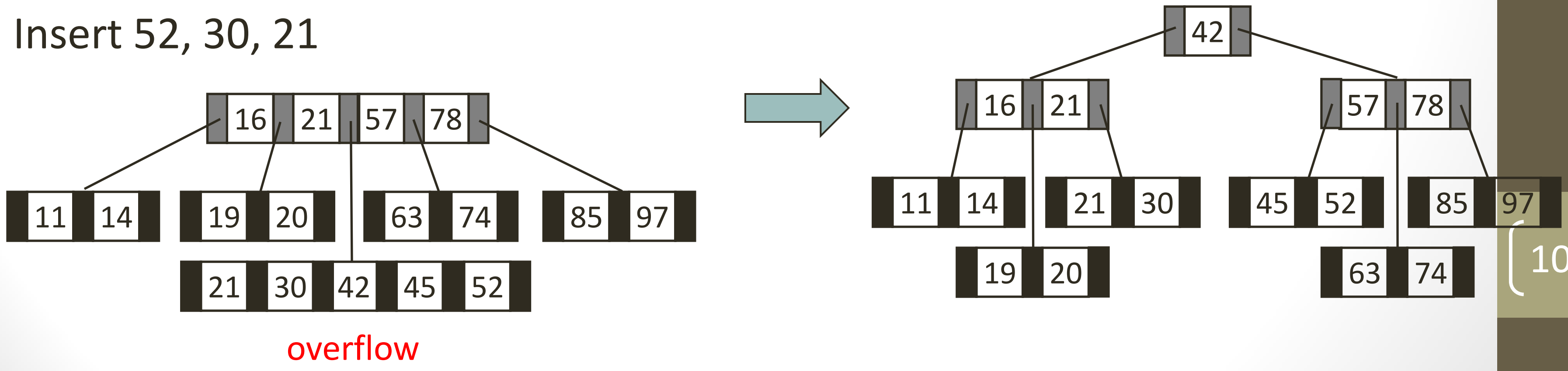
Insert 45, 42, 57



Insert 20, 16, 19



Insert 52, 30, 21



Deletion in B-Tree

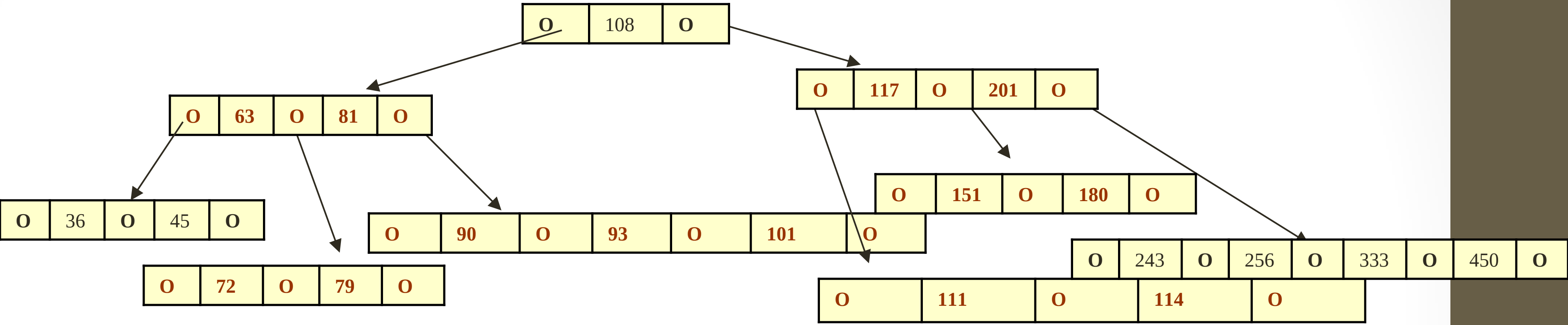
- Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. First, a leaf node has to be deleted. Second, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.
- 1. Locate the leaf node which has to be deleted
- 2. If the leaf node contains more than minimum number of key values (more than m/2 elements), then delete the value.
- 3. Else, if the leaf node does not contain even m/2 elements ,then, fill the node by taking an element either from the left or from the right sibling
  - 3.a. If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.
  - 3.b. Else if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

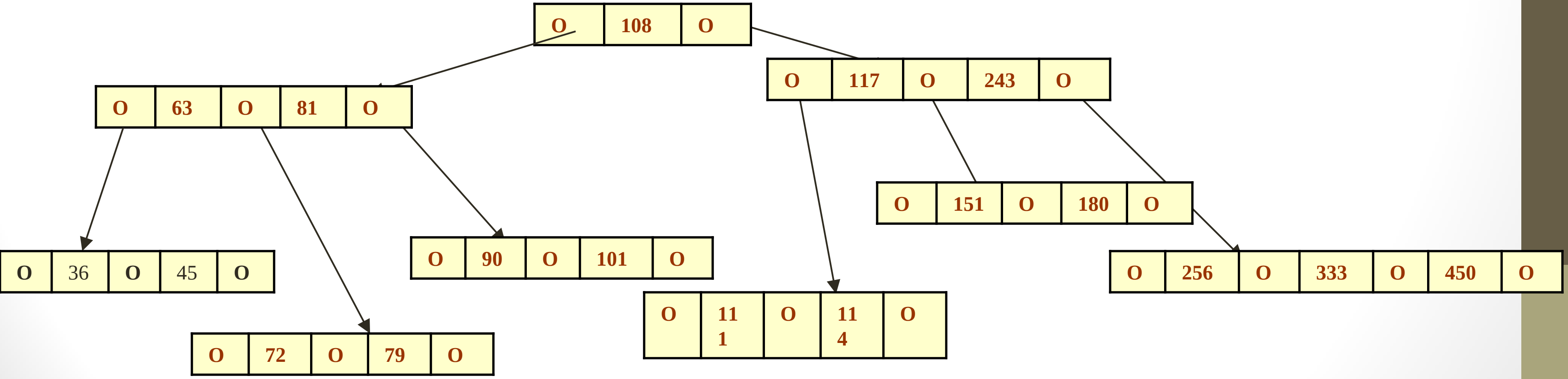
Deletion in B-Tree

- 4. Else, if both left and right siblings contain only minimum number of elements, then ,
  - 4.a. Create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements do not exceed the maximum number of elements a node can have, that is, m).
  - 4.b. If pulling the intervening element from the parent node leaves it with less than minimum number of keys in the node, then propagate the process upwards thereby reducing the height of the B tree.

- 5. To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So further the processing will be done as if a value from the leaf node has been deleted.

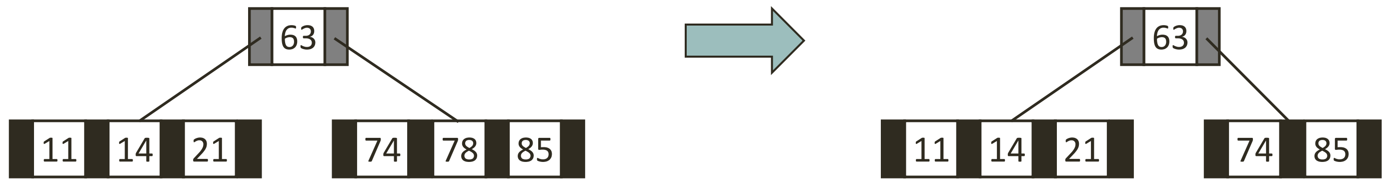
Consider the B tree of order 5 given below and delete the values- 93, 201 from it.



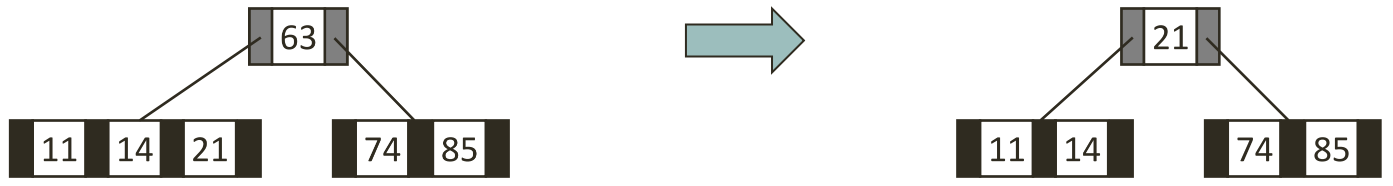


B-Tree Deletion

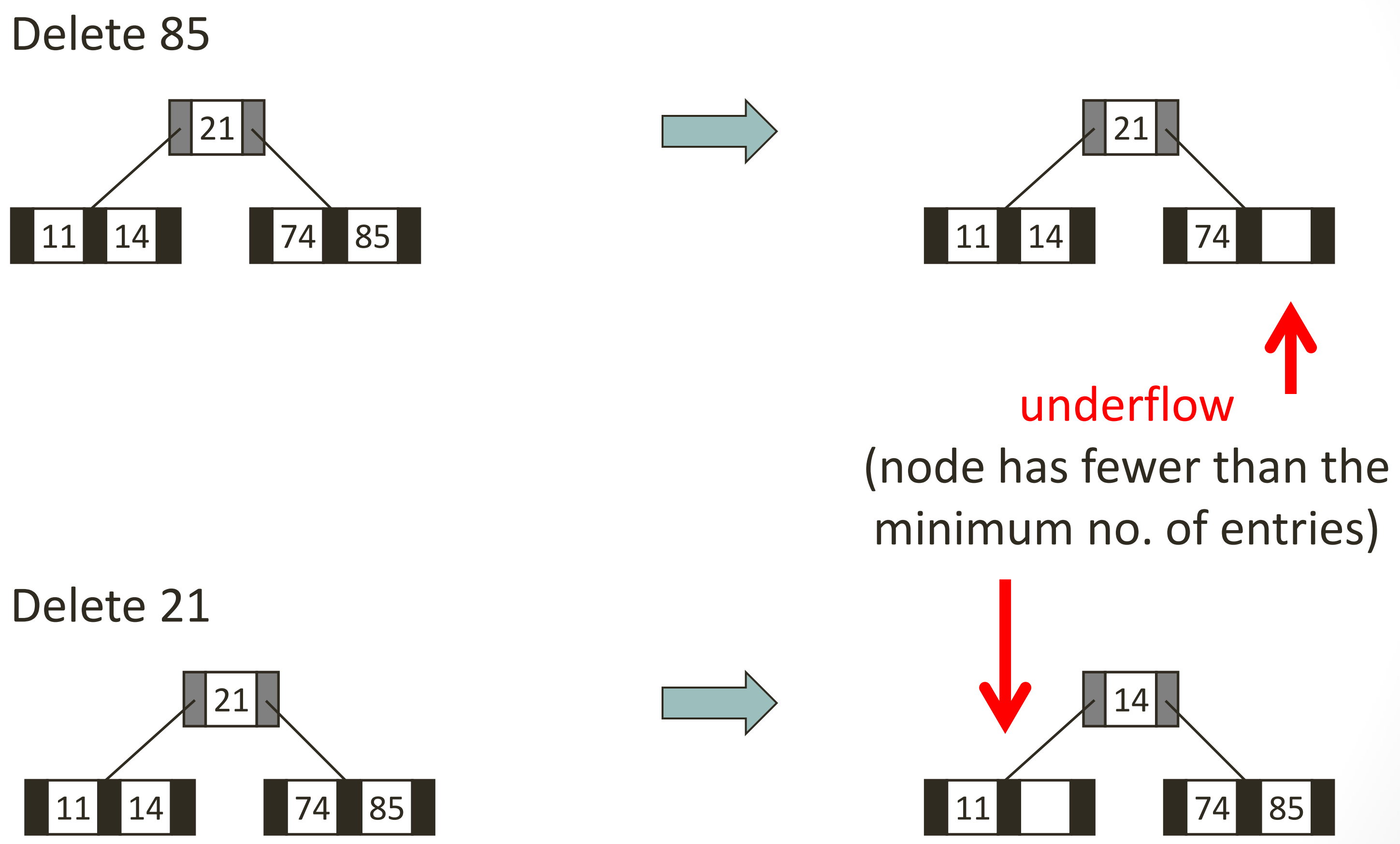
Delete 78



Delete 63



B-Tree Deletion



- For each node to have sufficient number of entries:
  - Balance: shift data among nodes.
  - Combine: join data from nodes.

Balance

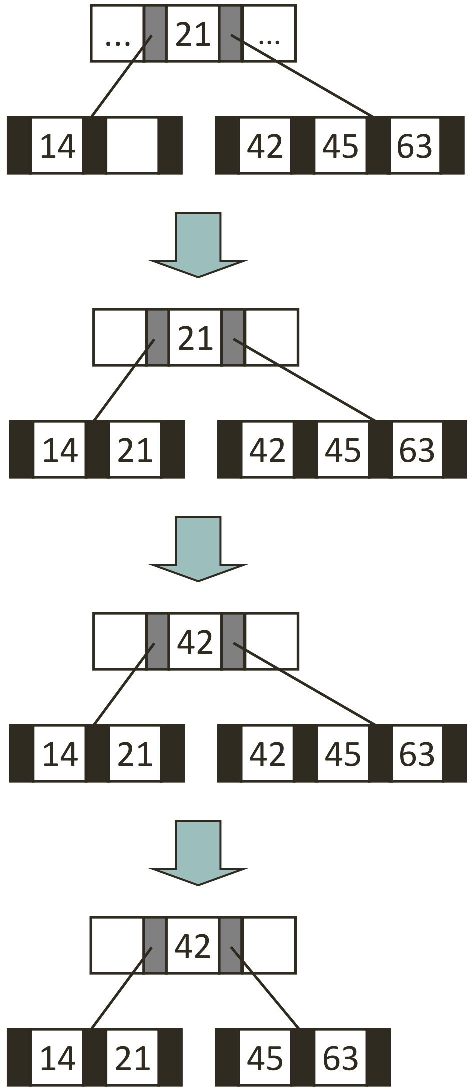
Borrow from right

Original node

Rotate parent data down

Rotate data to parent

Shift entries left



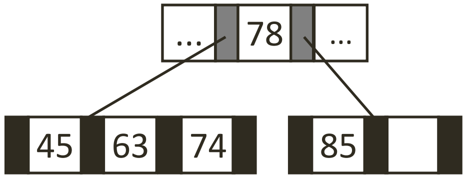
when the right sibling of the underflow node has more than minimum no. of entries

Balance

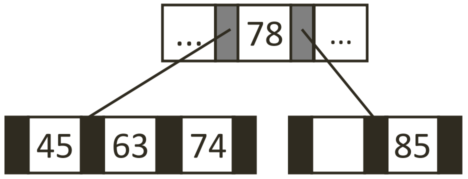
Borrow from left

when the left sibling of the underflow node has more than minimum no. of entries

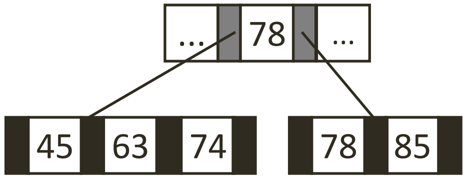
Original node



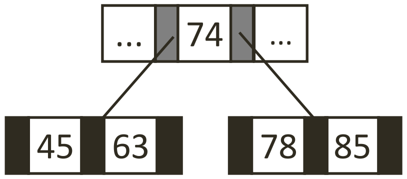
Shift entries right

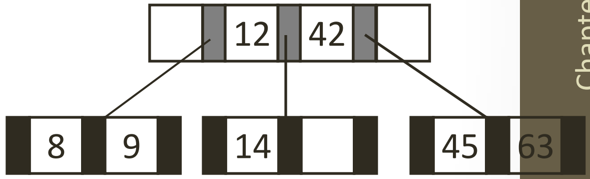


Rotate parent data down



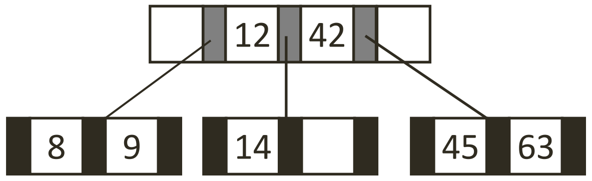
Rotate data up

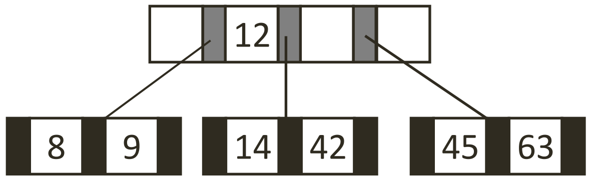




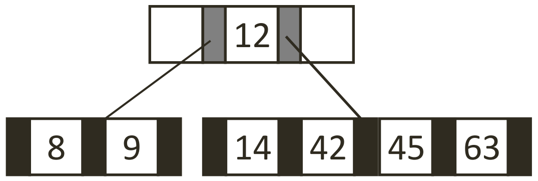
Combine

- when both left and right sibling nodes of the underflow nodes have minimum no. of entries





- choose one of its sibling
- move the separator down to the underflow node



- combine the underflow node with the chosen sibling
- if the parent node is underflow, repeat this combination until the root.

Red Black Tree

- A red-black tree is a self-balancing binary search tree.
- Although a red black tree is complex, it has good worst-case running time for its operations and is efficient to use as searching, insertion, and deletion can all be done in O(log *n*) time, where *n* is the number of nodes in the tree.
- A red-black tree is a binary search tree in which every node has a *color* which is either *red* or *black*. Apart from the other restrictions of binary search trees, red black trees have some additional requirements that can be given as follows:

✓ The color of a node is either red or black.

✓ The color of the root node is always black.

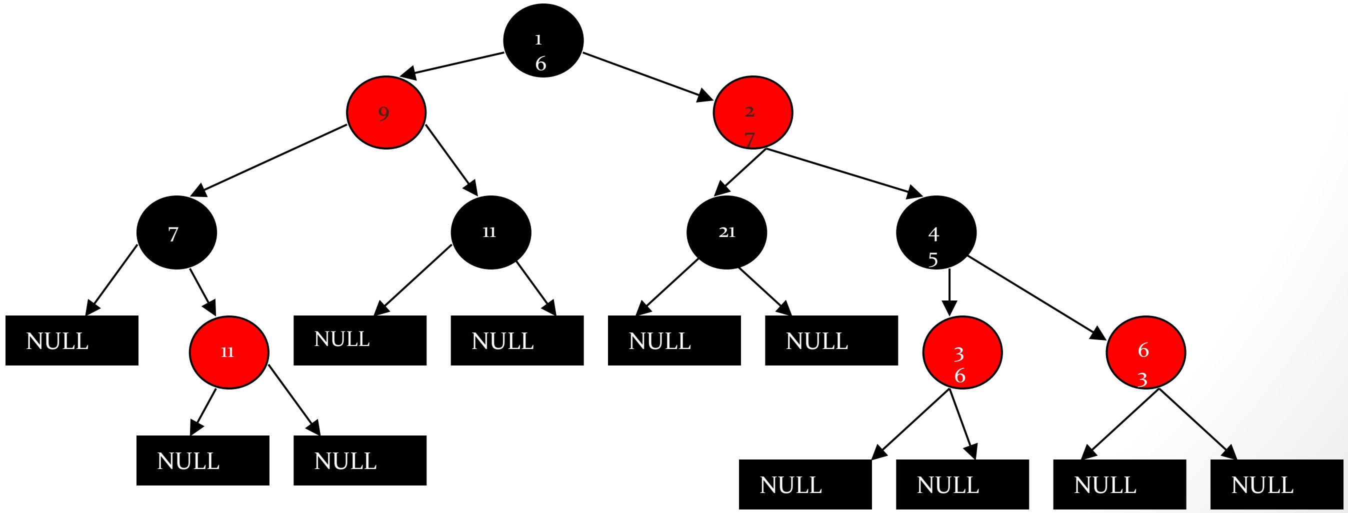
✓ All leaf nodes are black.

✓ Every red node has both the children colored in black.

✓ Every simple path from a given node to any of its leaf nodes has equal number of black nodes.

Red Black Tree

- These constraints enforce a critical property of red-black trees, that is, *the longest path from the root node to any leaf node is no more than twice as long as the shortest path from the root to any other leaf in that tree.*
- This results in a roughly balanced tree. Since operations such as insertion, deletion, and searching require worst-case time proportional to the height of the tree, this theoretical upper bound on the height allows red-black trees to be efficient in the worst-case, unlike ordinary binary search trees.



Insertion in Red Black tree

- Once the new node is added, one or the other property of the red-black tree may get violated.
- So in order to restore their property, we will check for certain cases and will do restoration of the property depending on the case that turns up after insertion.

- ***Case 1: The new node N is added as the root of the tree.***
- In this case, N is repainted black as the root of the tree is always black (property 2). Since N adds one black node to every path at once, Property 5 that says all paths from any given node to its leaf nodes has equal number of black nodes is not violated.

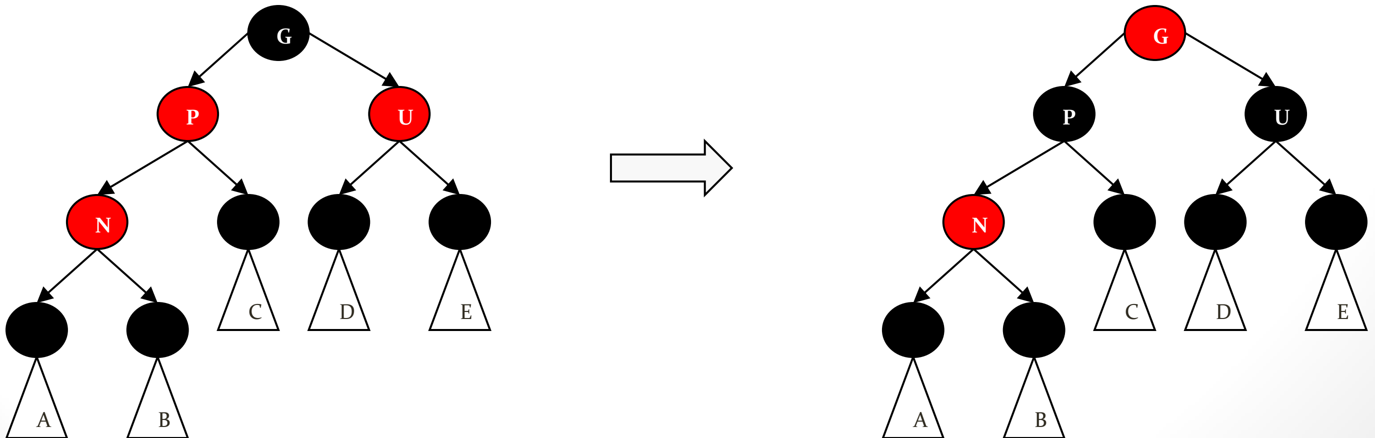
Insertion in Red Black tree

- ***Case 2: The new node's parent P is black***
- In this case, both children of every red node are black, so property 4 is not invalidated. Property 5 which says that all paths from any given node to its leaf nodes have equal number of black nodes is also not threatened. This is because the new node N has two black leaf children, but because N is red, the paths through each of its children have the same number of black nodes.

- ***Case 3: If both the parent (P) and the uncle (U) are red***
- In this case property 5 which says all paths from any given node to its leaf nodes have equal number of black nodes is violated.

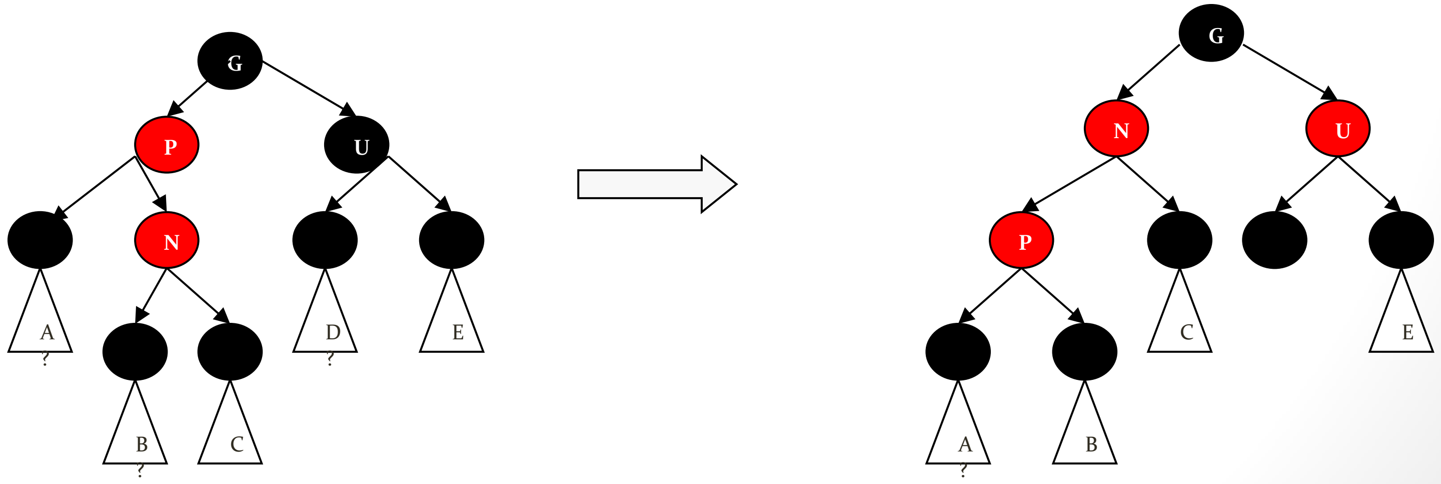
Insertion in Red Black tree

- So in order to restore property 5, both nodes (P and U) are repainted black and the grandparent G is repainted red. Now, the new red node N has a black parent. Since any path through the parent or uncle must pass through the grandparent, the number of black nodes on these paths has not changed.
- However, the grandparent G may now violate property 2 which says that the root node is always black or property 4 which states that both children of every red node are black. Property 4 will be violated when G has a red parent. So in order to fix this problem, this entire procedure is recursively performed on G from case 1.



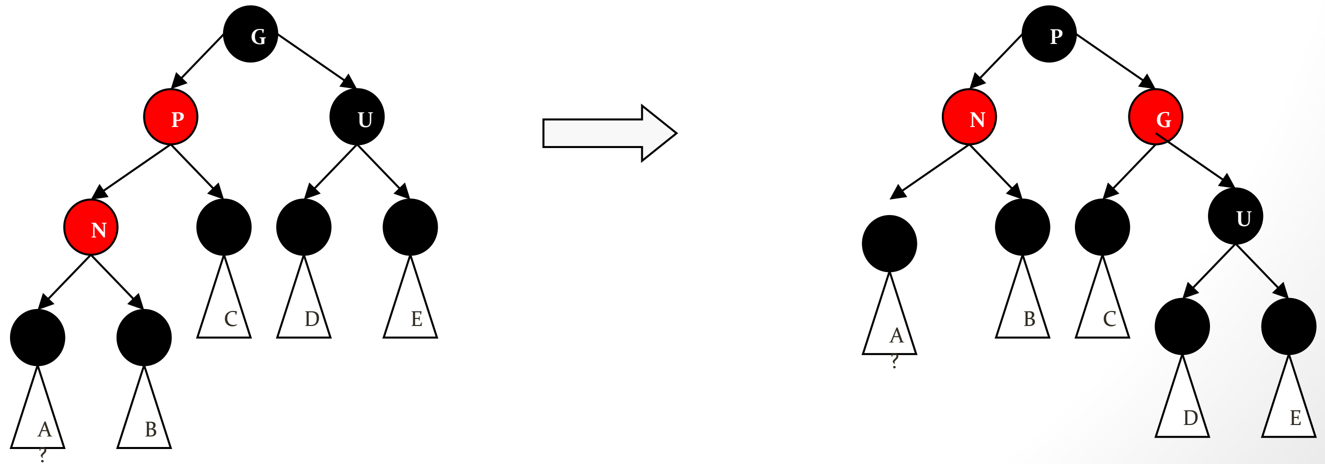
Insertion in Red Black tree

- ***Case 4: The parent P is red but the uncle U is black and N is the right child of P and P is the left child of G***
- In order to fix this problem, a left rotation is done to switch the roles of the new node N and its parent P. After rotation, note that in the C code we have re-labeled N and P and then case 5 is called to deal with the new node's parent. This is done because property 4 which says both children of every red node should be black is still violated.



Insertion in Red Black tree

- ***Case 5: The parent P is red but the uncle U is black and the new node N is the left child of P, and P is the left child of its parent G.***
- In order to fix this problem, a right rotation on G (the parent of parent of N) is performed. After this rotation, the former parent P is now the parent of both the new node N and the former grandparent G.
- We know that the color of G is black (because otherwise its former child P could not have been red) so now switch the colors of P and G so that the resulting tree satisfies Property 4 which states that both children of a red node are black.



Deletion in Red Black tree

- While deleting a node, if its color is red then we can simply replace it with its child, which must be black. All paths through the deleted node will simply pass through one less red node, and both the deleted nodes’ parent and child must be black, so none of the properties will be violated.
- Another simple case is when we delete a black node that has a red child. In this case property 4 (both children of every red node are black) and property 5 (all paths from any given node to its leaf nodes has equal number of black nodes) could be violated, so to restore them just repaint the deleted node’s child with black.

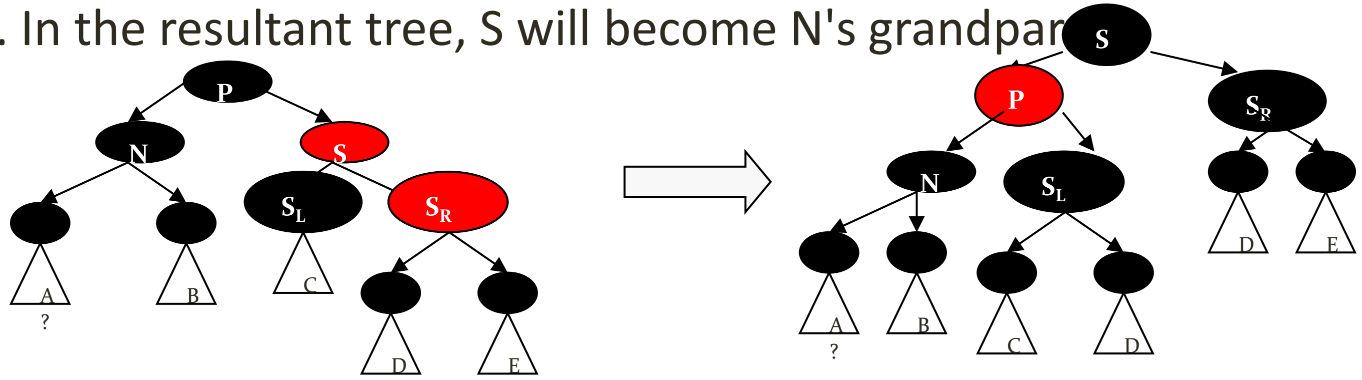
Deletion in Red Black tree

- However, complex situation arises when both the nodes to be deleted as well as its child are black. In this case, we begin by replacing the node to be deleted with its child. This will violate property 5 which states that all paths from any given node to its leaf nodes have equal number of black nodes. Therefore, the tree needs to be rebalanced. There are several cases to consider.

- ***Case 1: N is the new root.***
- In this case, we have removed one black node from every path, and the new root is black, so none of the properties are violated.

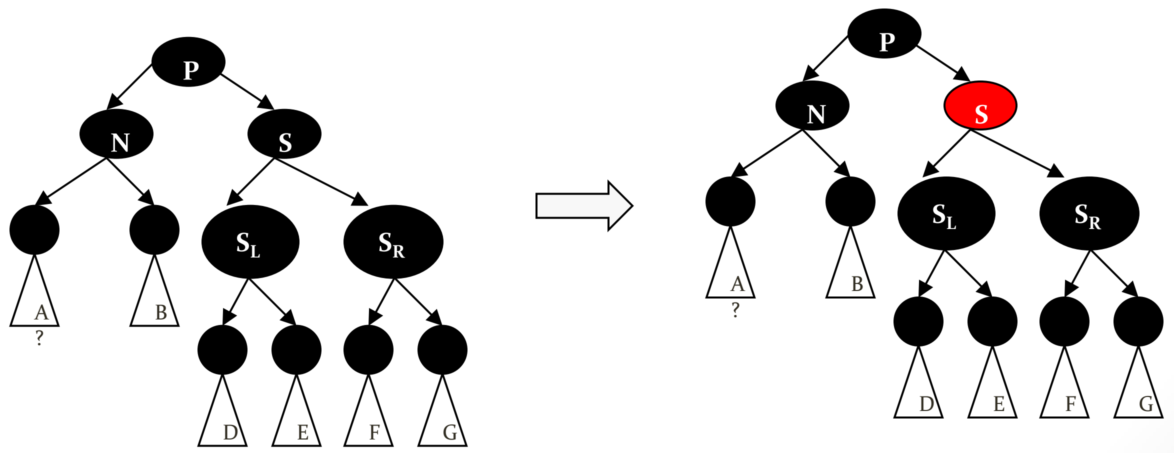
Deletion in Red Black tree

- ***Case 2: Sibling S is red.***
- In this case, interchange the colors of P and S, and then rotate left at P. In the resultant tree, S will become N's grandpar



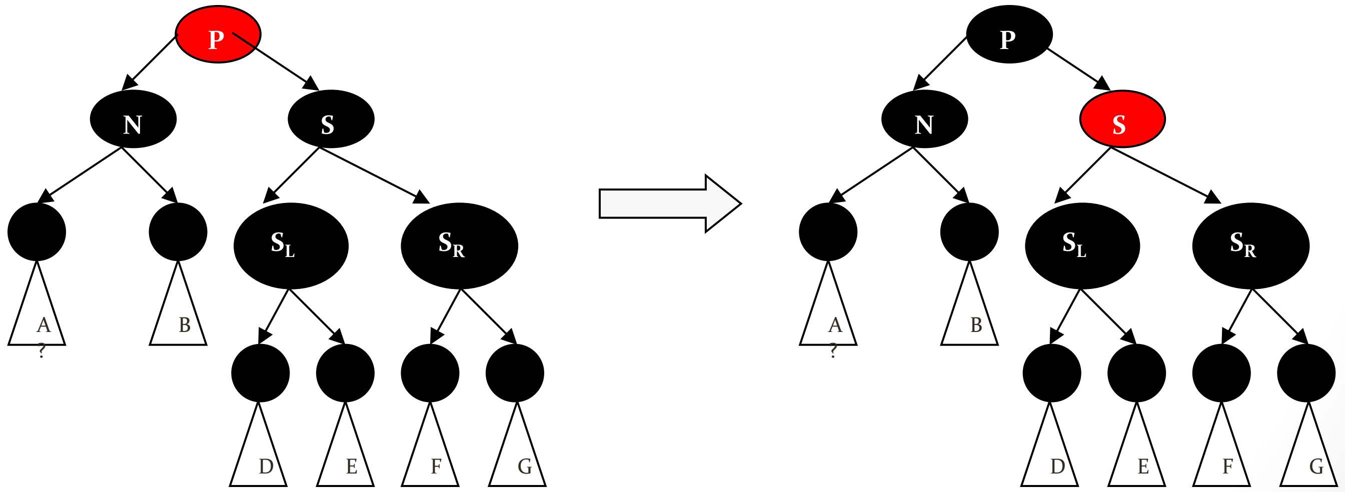
Deletion in Red Black tree

- ***Case 3: P, S, and S's children are black.***
- In this case simply repaint S with red color. The resultant tree will have all paths passing through S, have one less black node. Therefore, all paths that pass through P now have one fewer black node than paths that do not pass through P, so Property 5 which states that all paths from any given node to its leaf nodes have equal number of black nodes is still violated. To fix this problem, we perform the rebalancing procedure on P, starting at case 1.



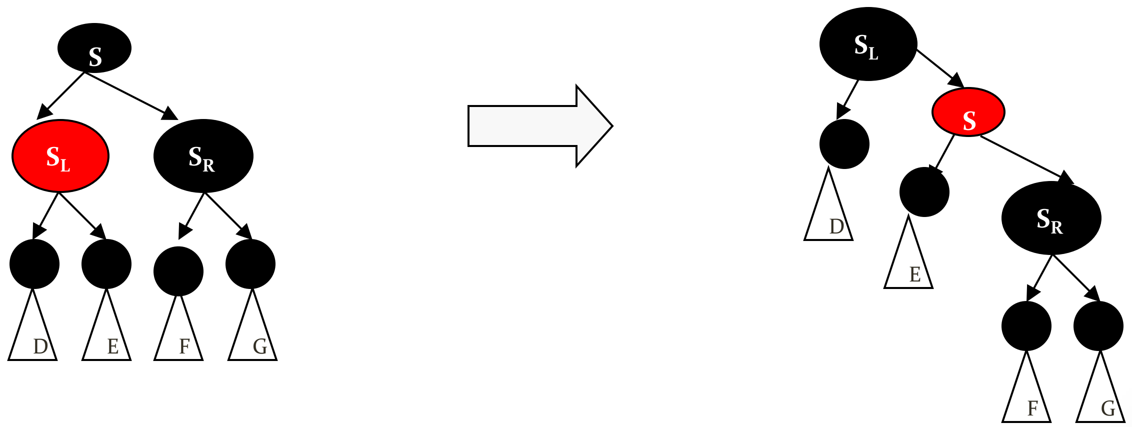
Deletion in Red Black tree

- ***Case 4: If S and S's children are black, but P is red.***
- In this case, we interchange the colors of S and P. Although this will not affect the number of black nodes on paths going through S, but it will add one black node to the paths going through N, making up for the deleted black node on those paths.



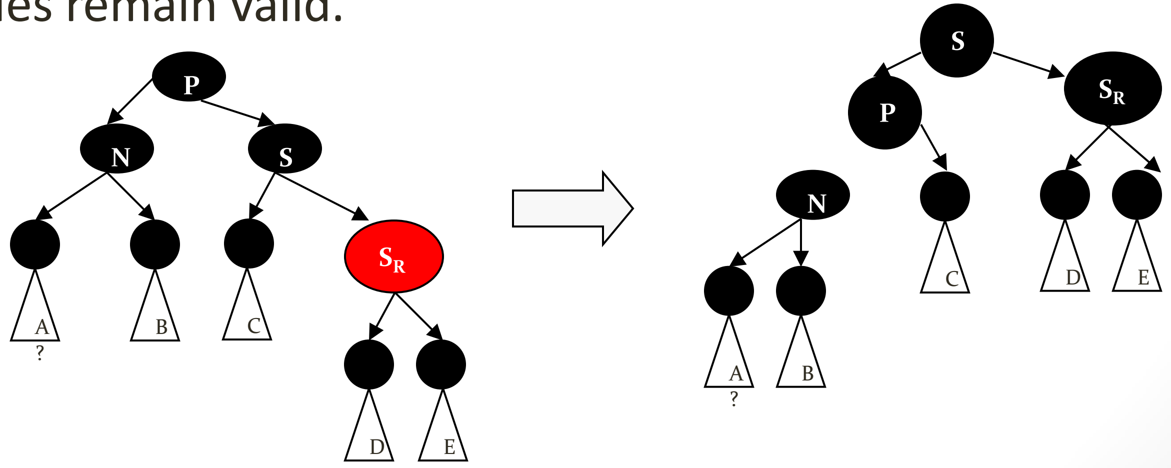
Deletion in Red Black tree

- ***Case 5: If N is the left child of P and S is black, S's left child is red, S's right child is black.***
- In this case, perform a right rotation at S. After the rotation, S's left child becomes S's parent and N's new sibling. Also interchange the colors of S and its new parent.
- Note that now all paths still have equal number of black nodes, but N has a black sibling whose right child is red, so we fall into case 6.



Deletion in Red Black tree

- ***Case 6: S is black, S's right child is red, and N is the left child of its parent P.***
- In this case, a left rotation is done at P to make S the parent of P and S's right child. After the rotation, the colors of P and S are interchanged and S’s right child is colored black. Once these steps are followed, you will observe that property 4 which says both children of every red node are black and property 5 that states all paths from any given node to its leaf nodes have equal number of black nodes remain valid.



Thank you!