Tree

BEFORE START

- Although stacks and queues reflect some hierarchy
- They are limited to only one dimension.

- Motivation and Terminology
- Representations
- Traversals
- Three Problems

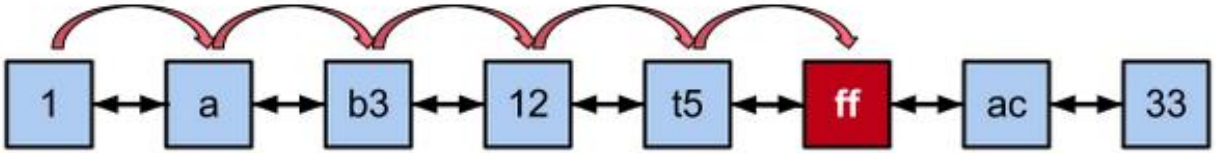
-Operations like insert, delete, etc. over linked lists are performed in a linear time.

-small data sets this works fine, but as the data grows these operations, especially the search operation becomes too slow.

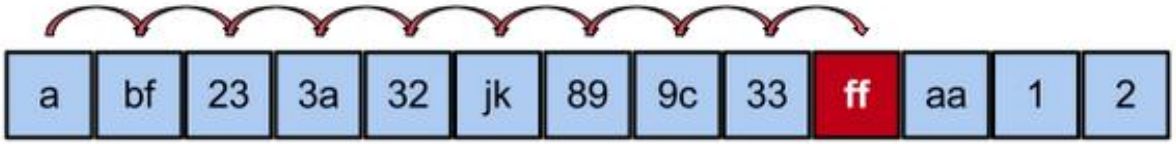
- we must go through the entire list in order to find the desired element.
- -The worst case is when the item doesn’t belong to the list and we must check every single item of the list even the last one without success.

Search

1. Linked Lists `search for "ff"`



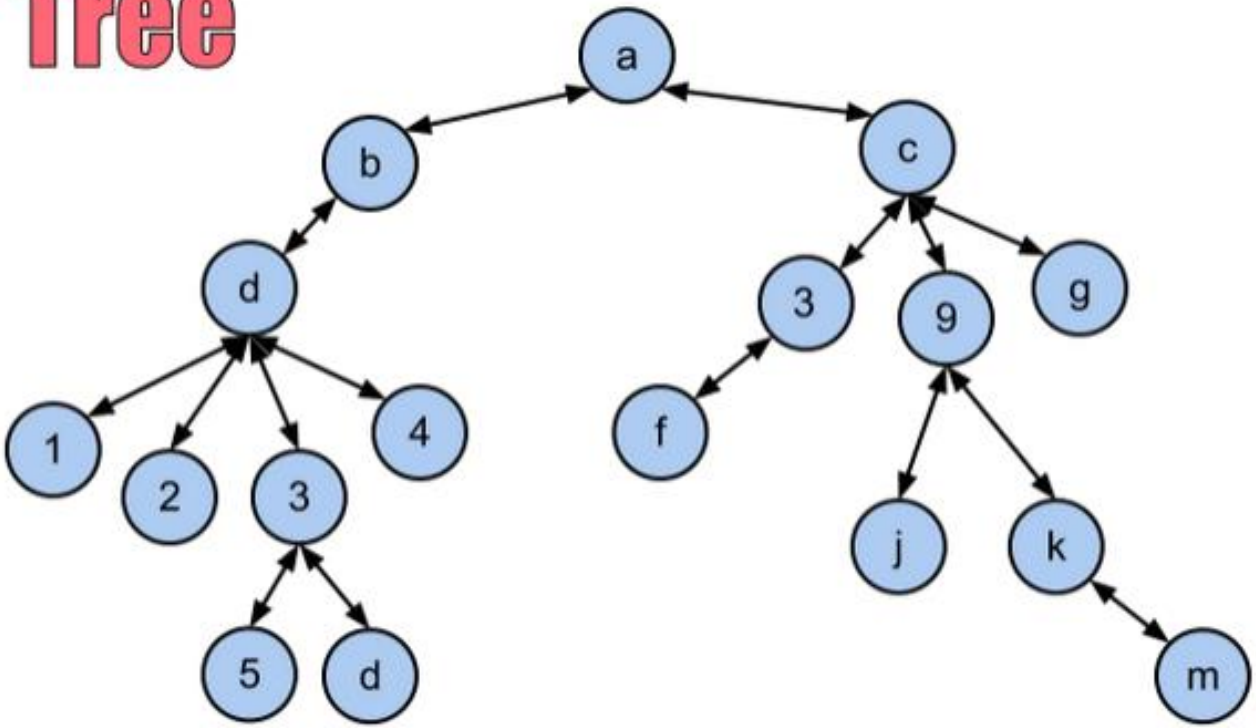
2. Arrays `sequential search`

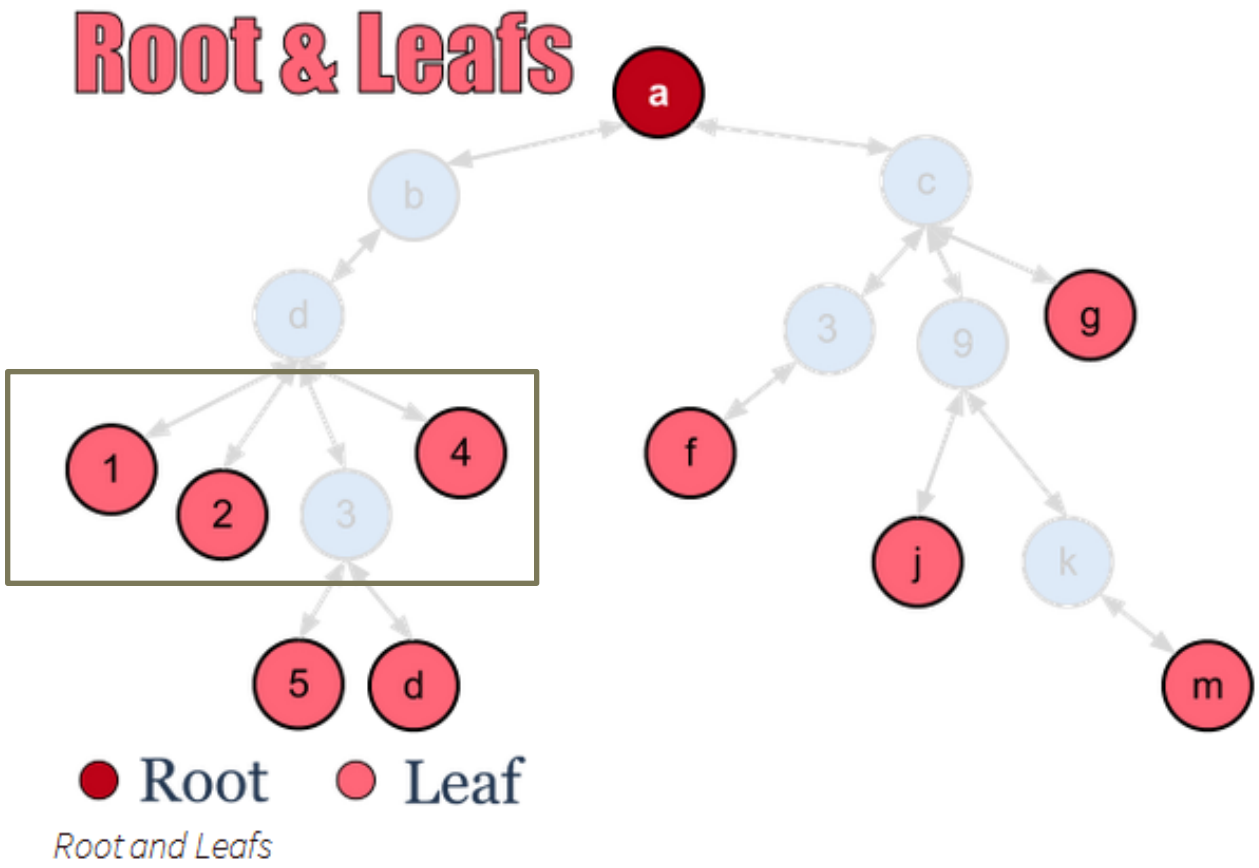


!Ineffective operation

We need another effective data structure

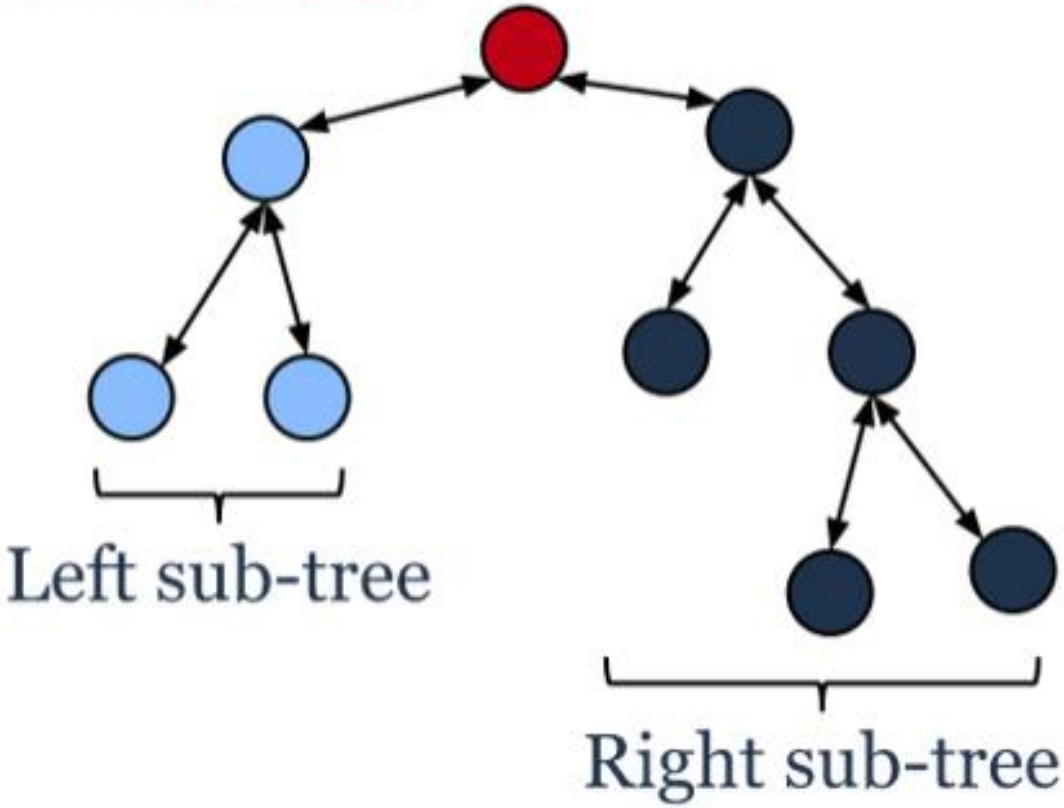
Tree





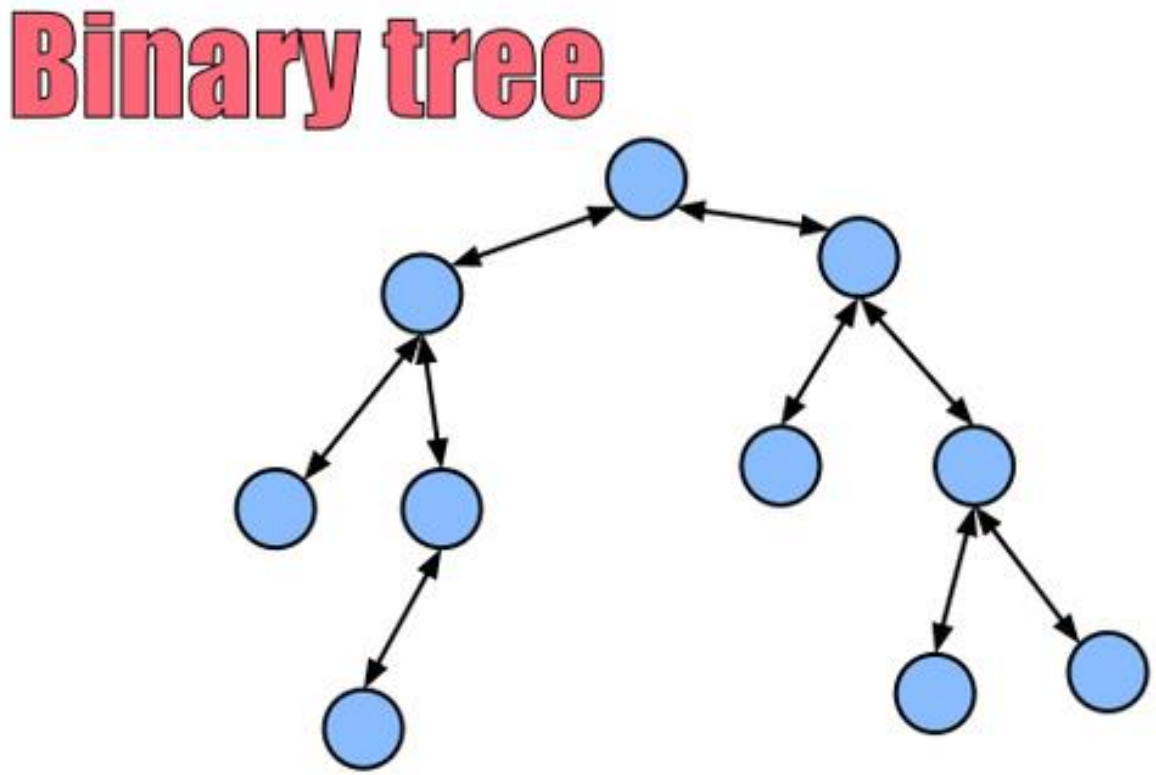
Root and Leafs

Sub-trees



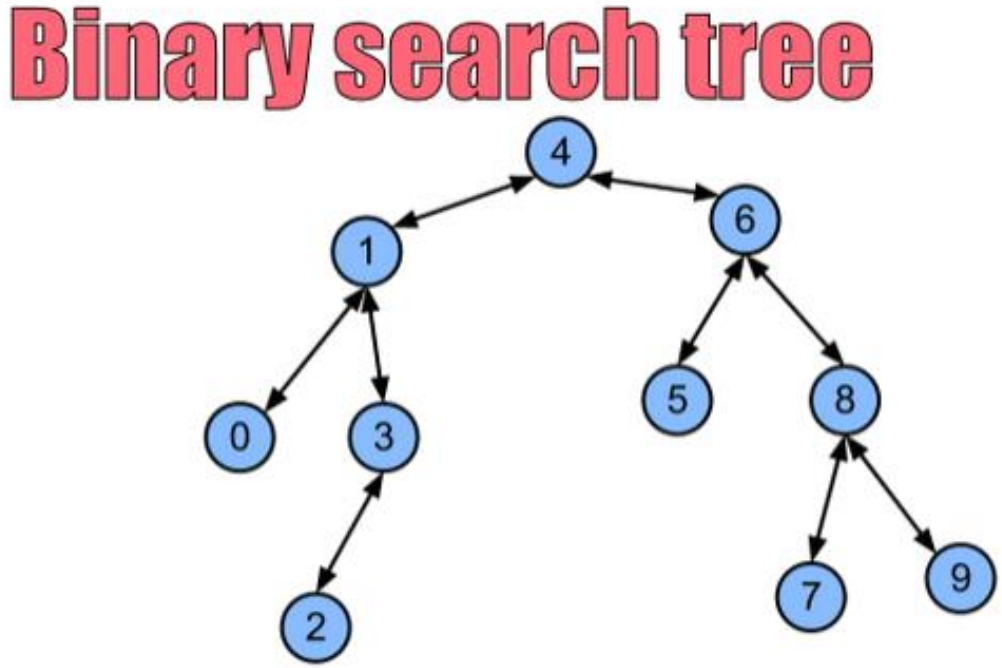
Binary Tree

A binary tree is a tree where each item can have at most two children.



a binary tree isn’t more successful in searching than any other tree or data structure. If the items aren’t placed in a specific order we must go through the entire tree in order to find the searched item. This isn’t a great optimization, so we must put an order in it to improve the searching process

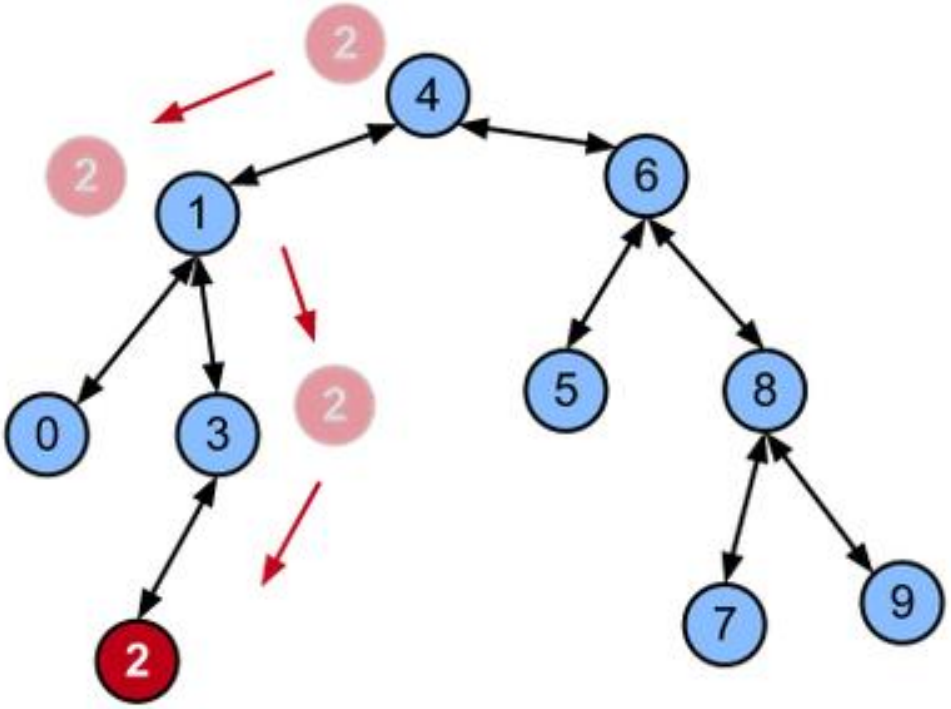
The binary search tree is a specific kind of binary tree, where the each item keeps greater elements on the right, while the smaller items are on the left.



There are four basic BST operations: traversal, search, insert, and delete.

Traversals
Searches
Insertion
Deletion

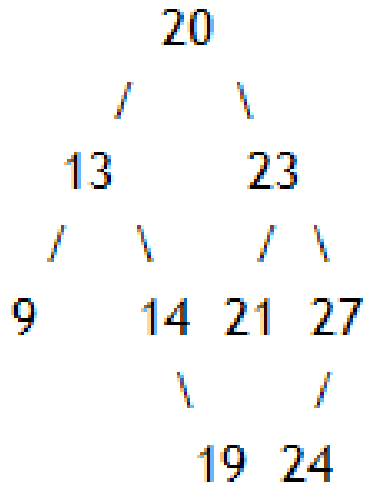
Insert in BST



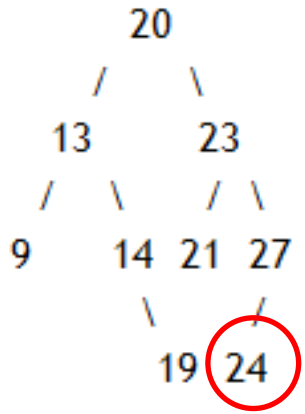
Deletion in Binary Search Tree:

case 1: Node with no children (or) leaf node
case 2: Node with one child
case 3: Node with two children.

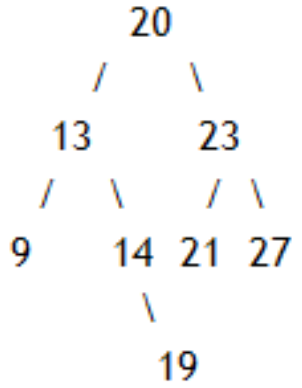
Original Tree



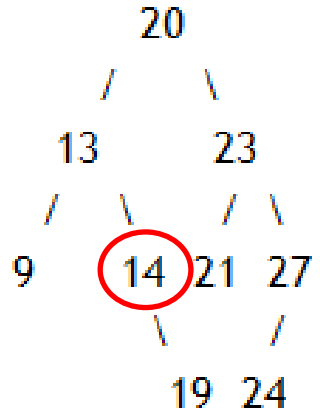
Case 1: Delete a leaf node/ node with no children.



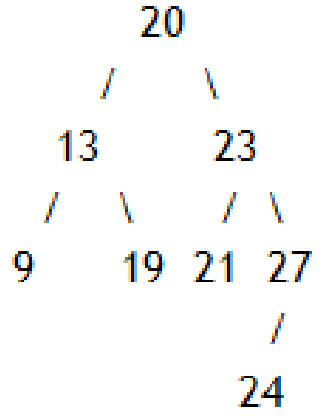
Delete 24 from the above binary search tree.



Case 2: Delete a node with one child.

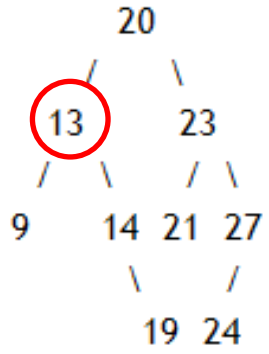


Delete 14 from above binary search tree.

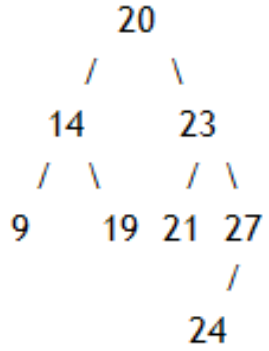


Case 3: Delete a node with two children.

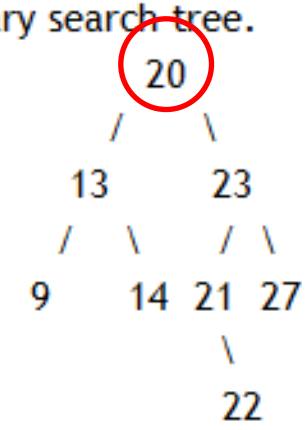
Delete a node whose right child is the smallest node in the right sub-tree. (14 is the smallest node present in the right sub-tree of 13).



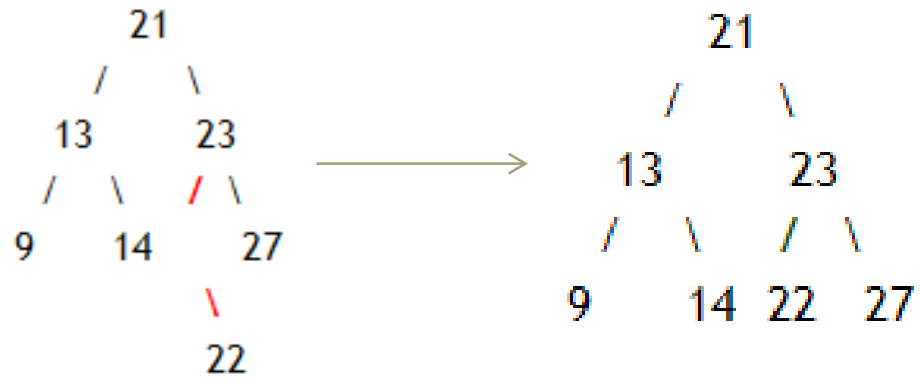
Delete 13 from the above binary tree. Find the smallest in the left subtree of 13. So, replace 13 with 14.



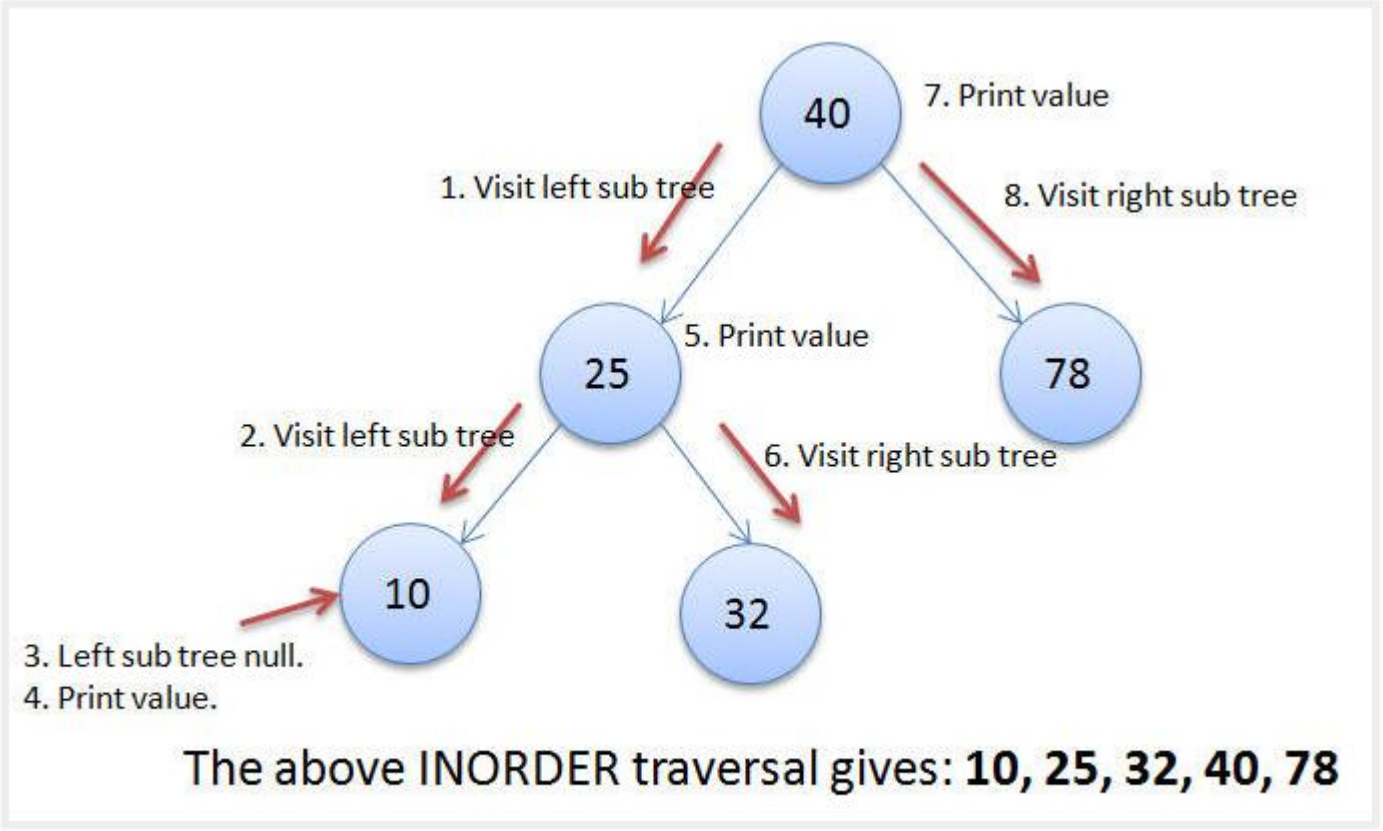
Delete 20 from the below binary search tree.



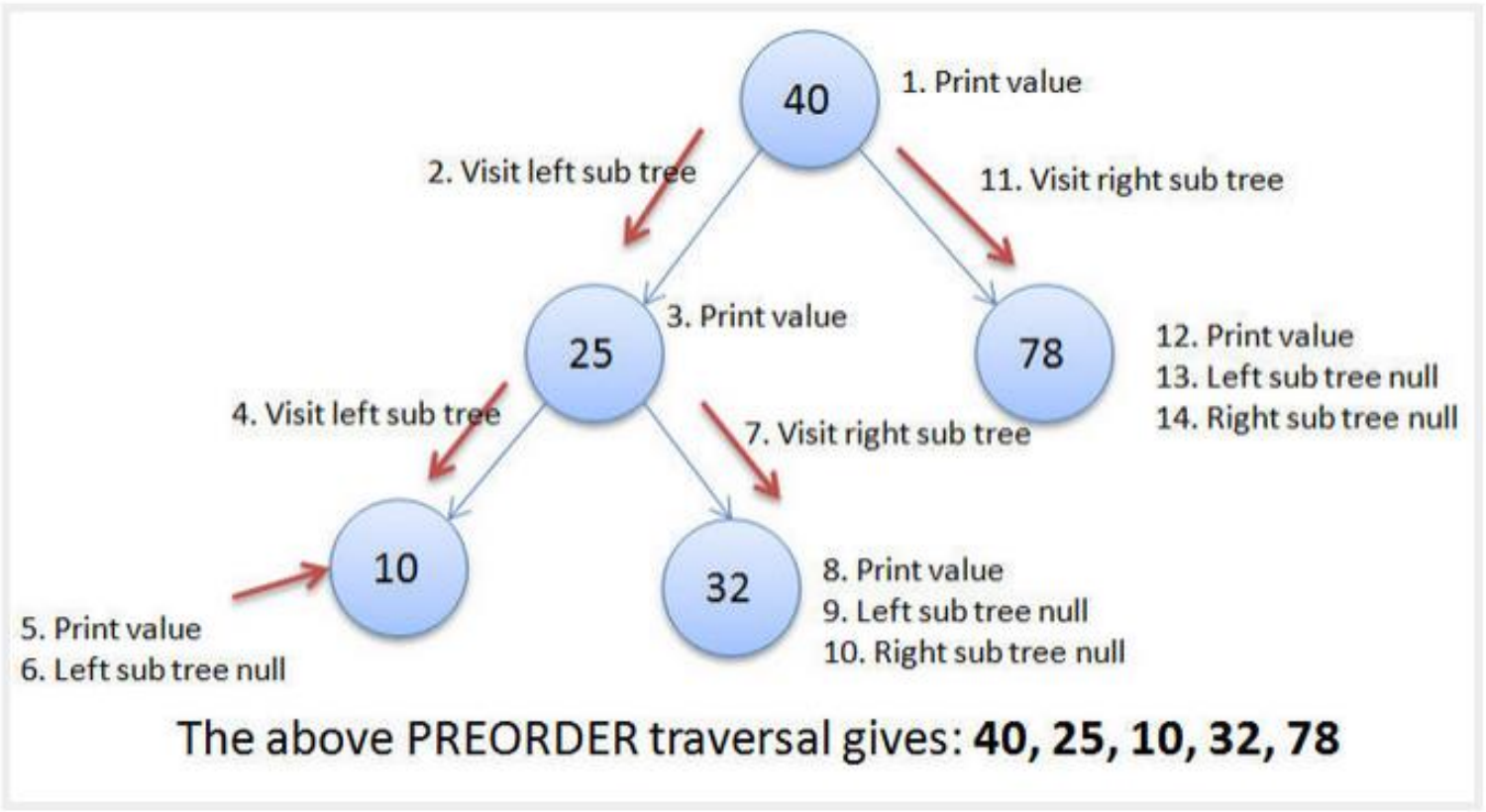
Find the smallest node in the right sub-tree of 20. And that smallest node is 21. So, replace 20 with 21. Since 21 has only one child(22), the pointer currently pointing to 21 is made to point to 22. So, the resultant binary tree would be the below.



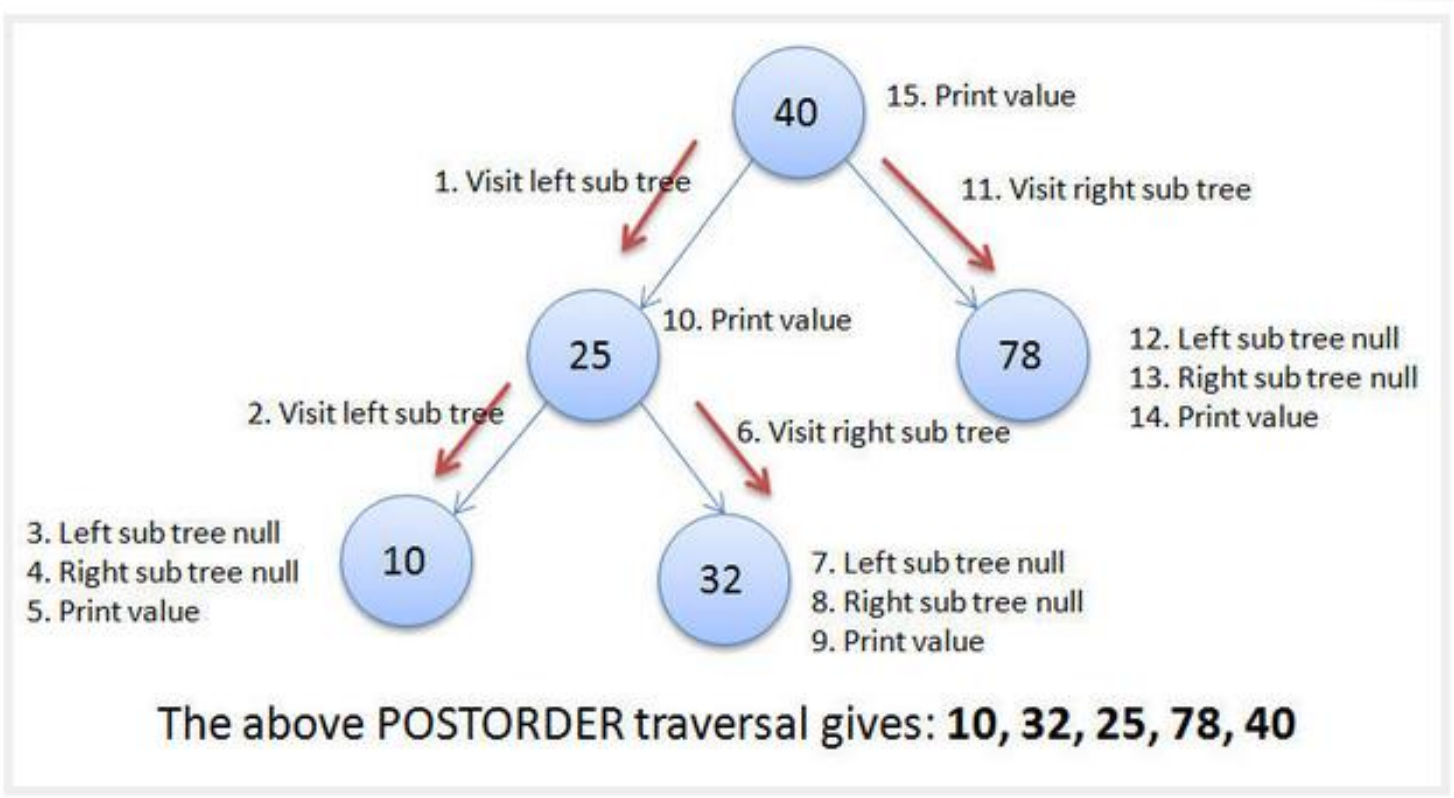
BST Travesal



Applying the Inorder traversal for the give example we get: 3, 10, 17, 25, 30, 32, 38, 40, 50, 78, 78, 93.



Applying the Preorder traversal for the give example we get: 40, 25, 10, 3, 17, 32, 30, 38, 78, 50, 78, 93.



Applying the Postorder traversal for the give example we get: 3, 17, 10, 30, 38, 32, 25, 50, 93, 78, 78, 40.

Insert node

```
11 void insert(node ** tree, int val) {
12 node *temp = NULL;
13 if(!(*tree)) {
14   temp = (node *)malloc(sizeof(node));
15   temp->left = temp->right = NULL;
16   temp->data = val;
17   *tree = temp;
18   return;
19 }
20
21 if(val < (*tree)->data) {
22      insert(&(*tree)->left, val);
23   } else if(val > (*tree)->data) {
24     insert(&(*tree)->right, val);
25   }
26 }
```

This function would determine the position as per value of node to be added and new node would be added into binary tree. Function is explained in steps below and code snippet lines are mapped to explanation steps given below.

[Lines 13-19] Check first if tree is empty, then insert node as root.

[Line 21] Check if node value to be inserted is lesser than root node value, then

a. [Line 22] Call insert() function recursively while there is non-NULL left node

b. [Lines 13-19] When reached to leftmost node as NULL, insert new node.

[Line 23] Check if node value to be inserted is greater than root node value, then

a. [Line 24] Call insert() function recursively while there is non-NULL right node

b. [Lines 13-19] When reached to rightmost node as NULL, insert new node.

Searching into binary tree

Searching is done as per value of node to be searched whether it is root node or it lies in left or right sub-tree. Below is the code snippet for search function. It will search node into binary tree.

```
46 node* search(node ** tree, int val) {
47 if(!(*tree)) {
48   return NULL;
49  }
50 if(val == (*tree)->data) {
51   return *tree;
52  } else if(val < (*tree)->data) {
53    search(&((*tree)->left), val);
54  } else if(val > (*tree)->data){
55    search(&((*tree)->right), val);
56  }
57 }
```

This search function would search for value of node whether node of same value already exists in binary tree or not. If it is found, then searched node is returned otherwise NULL (i.e. no node) is returned. Function is explained in steps below and code snippet lines are mapped to explanation steps given below.

[Lines 47-49] Check first if tree is empty, then return NULL.

[Lines 50-51] Check if node value to be searched is equal to root node value, then return node

[Lines 52-53] Check if node value to be searched is lesser than root node value, then call search() function recursively with left node

[Lines 54-55] Check if node value to be searched is greater than root node value, then call search() function recursively with right node

Repeat step 2, 3, 4 for each recursion call of this search function until node to be searched is found.

Deletion of binary tree

Binary tree is deleted by removing its child nodes and root node. Below is the code snippet for deletion of binary tree.

```
38 void deltree(node * tree) {
39 if (tree) {
40   deltree(tree->left);
41   deltree(tree->right);
42   free(tree);
43  }
44 }
```

This function would delete all nodes of binary tree in the manner – left node, right node and root node. Function is explained in steps below and code snippet lines are mapped to explanation steps given below.

[Line 39] Check first if root node is non-NULL, then
a. [Line 40] Call deltree() function recursively while there is non-NULL left node
b. [Line 41] Call deltree() function recursively while there is non-NULL right node
c. [Line 42] Delete the node.

Displaying binary tree

Binary tree can be displayed in three forms – pre-order, in-order and post-order.

Pre-order displays root node, left node and then right node.
In-order displays left node, root node and then right node.
Post-order displays left node, right node and then root node.

```
28 void print_preorder(node * tree) {
29 if (tree) {
30 printf("%d\n",tree->data);
31 print_preorder(tree->left);
32 print_preorder(tree->right);
33 }
34 }
35 void print_inorder(node * tree) {
36 if (tree) {
37 print_inorder(tree->left);
38 printf("%d\n",tree->data);
39 print_inorder(tree->right);
40 }
41 }
42 void print_postorder(node * tree) {
43 if (tree) {
44 print_postorder(tree->left);
45 print_postorder(tree->right);
46 printf("%d\n",tree->data);
47 }
48 }
```

These functions would display binary tree in pre-order, in-order and post-order respectively. Function is explained in steps below and code snippet lines are mapped to explanation steps given below.

Pre-order display

a. [Line 30] Display value of root node.
b. [Line 31] Call print_preorder() function recursively while there is non-NULL left node
c. [Line 32] Call print_preorder() function recursively while there is non-NULL right node

In-order display

a. [Line 37]Call print_inorder() function recursively while there is non-NULL left node
b. [Line38] Display value of root node.
c. [Line 39] Call print_inorder() function recursively while there is non-NULL right node

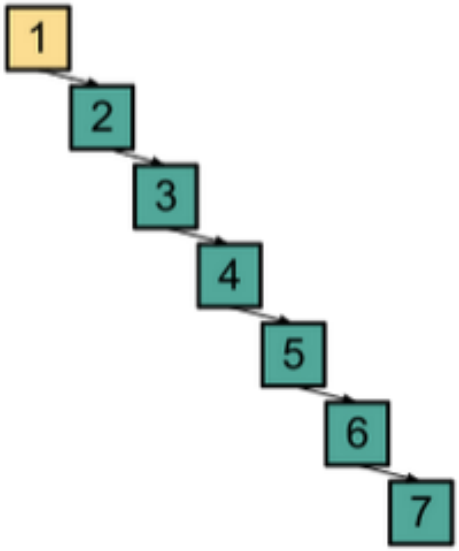
Post-order display

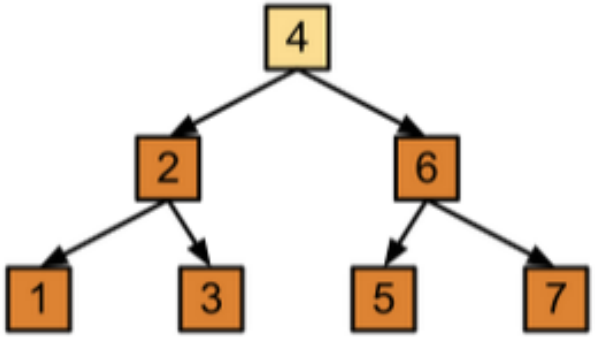
a. [Line 44] Call print_postorder() function recursively while there is non-NULL left node
b. [Line 45] Call print_postorder() function recursively while there is non-NULL right node
c. [Line46] Display value of root node.

Balanced Binary Search Tree

BALANCED vs. NON-BALANCED

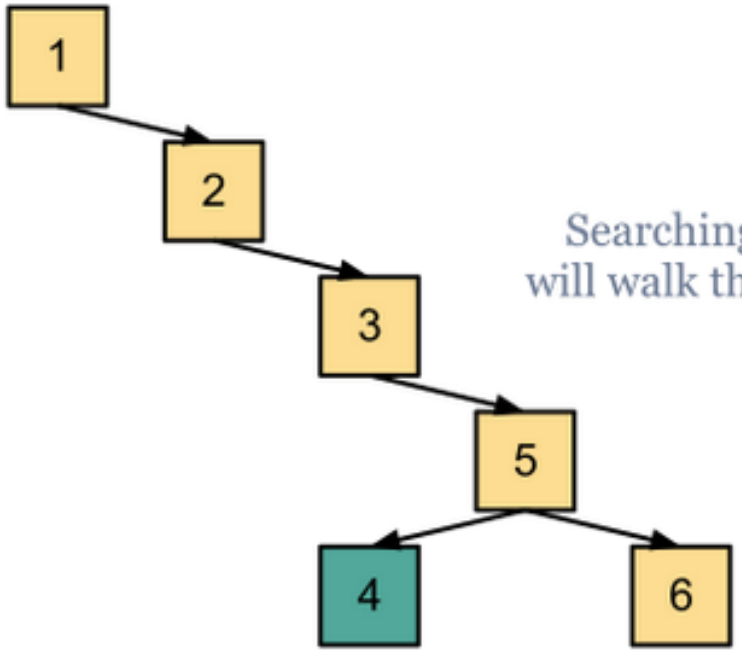
Built from the sequence [1,2,3,4,5,6,7]





INSERT

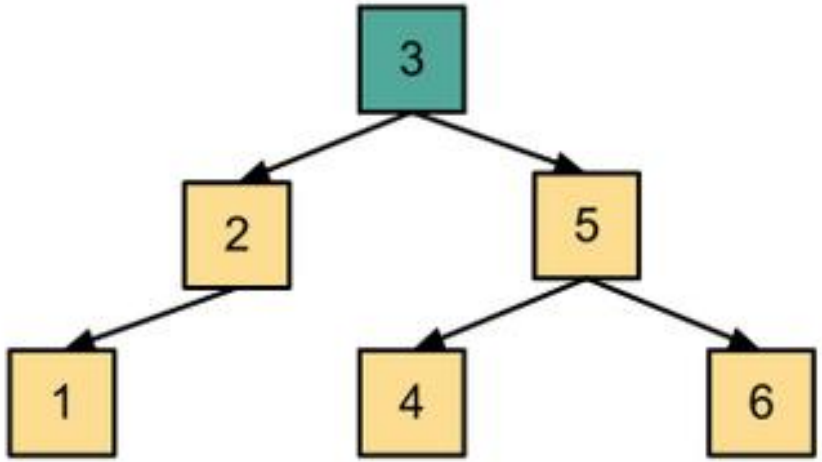
Insert the sequence [1,2,3,5,4,6]



Searching for key = 4
will walk through 5 items

BALANCED TREE

Insert the sequence [1,2,3,5,4,6] into a balanced tree

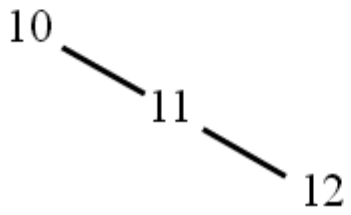


AVL Tree

(Adelson-Velsky and Landis' tree, named after the inventors)

Balanced Trees

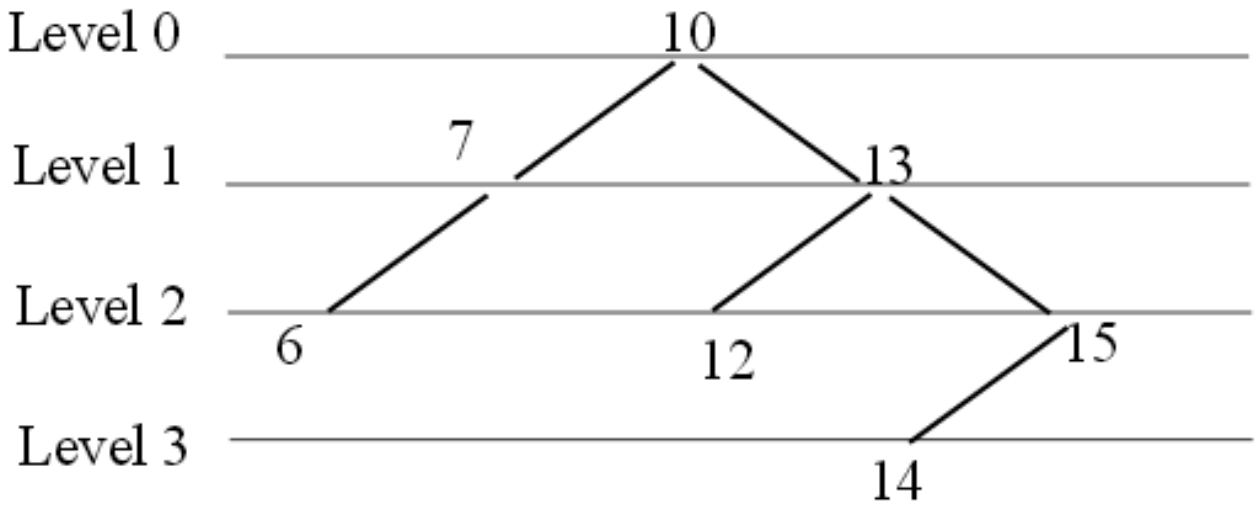
- Binary search trees: if all levels filled, then search, insertion and deletion are O(log N).
- However, performance may deteriorate to linear if nodes are inserted in order:

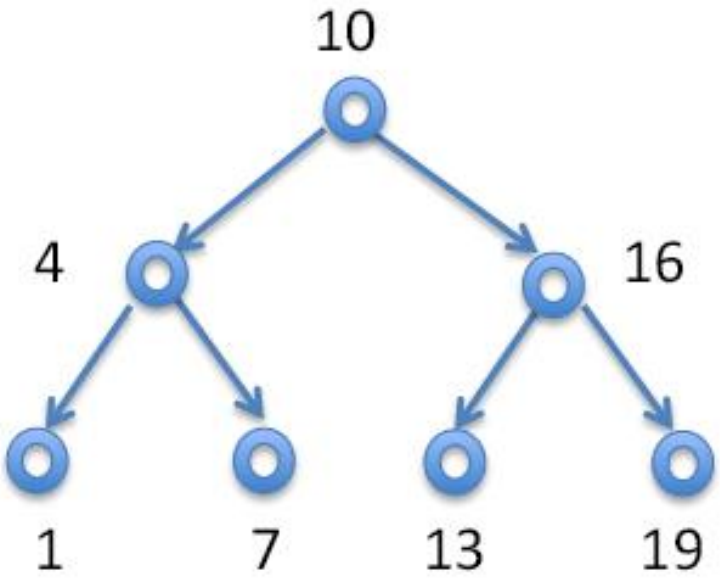


Solution

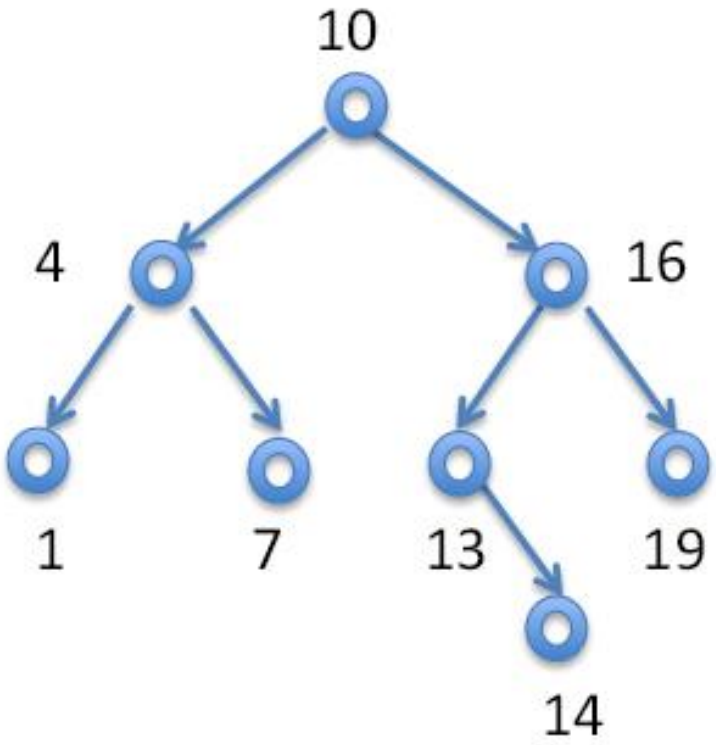
- Keep the trees height balanced (for every node, the difference in height between left and right subtrees at most 1)
- Performance always logarithmic.

Example

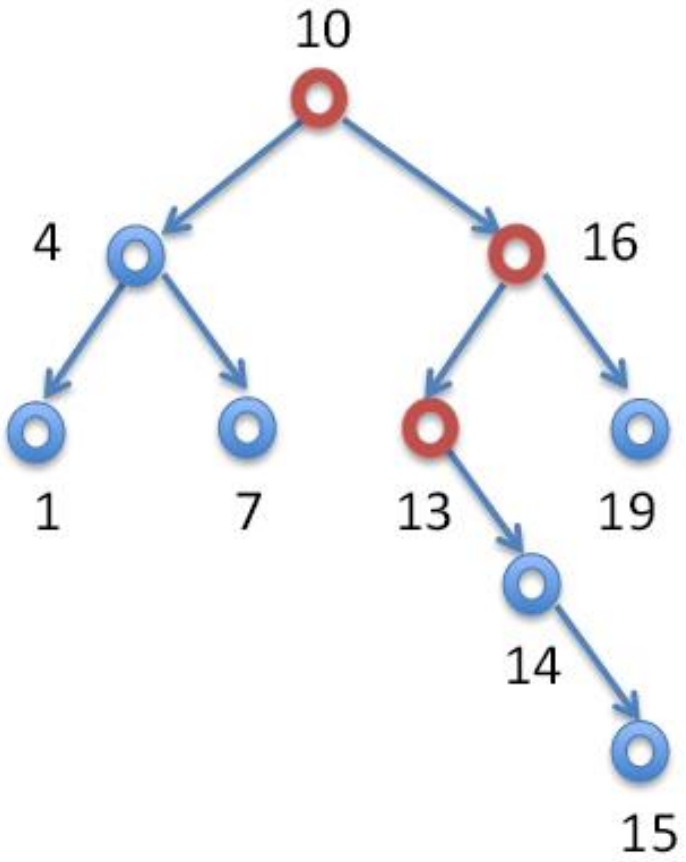



10
4
16
1
7
13
19

Add key 14



Add key 15

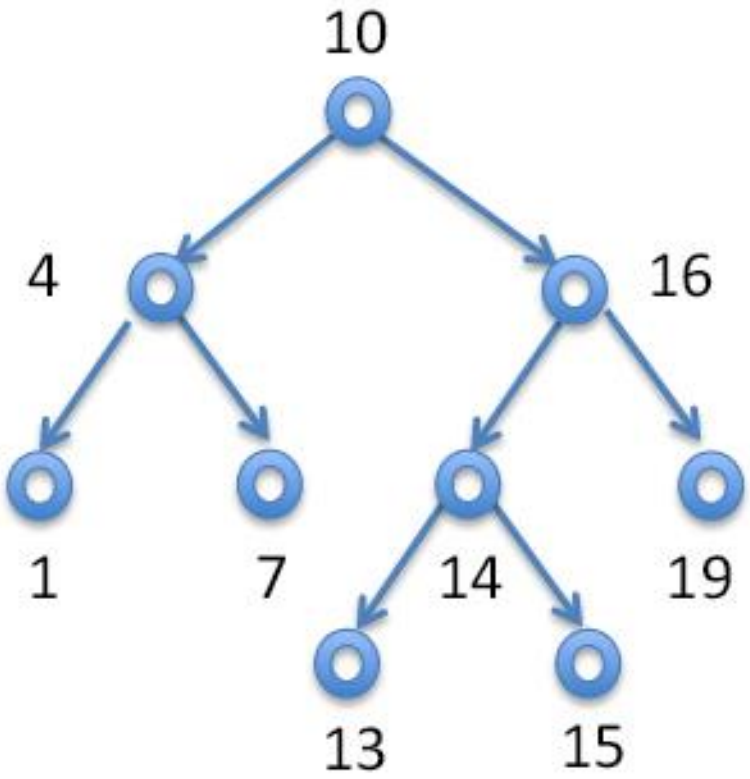


Height at key 13,16 and 10

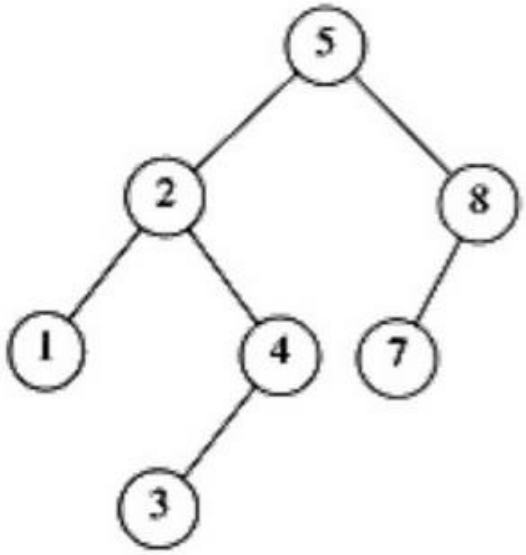
at the node labeled 13, the left subtree has height 0, while the right subtree has height 2

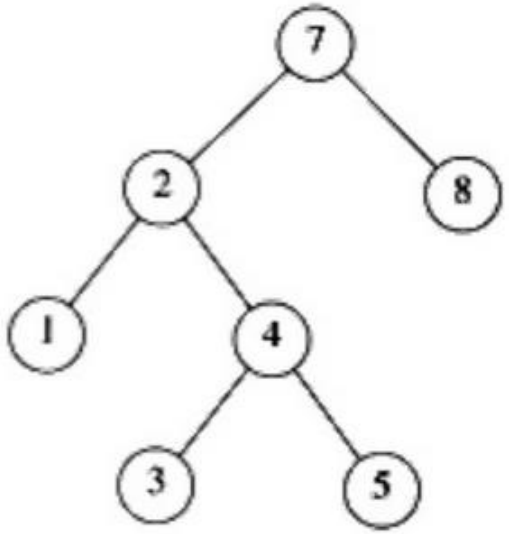
key 16, the left subtree has height 3 while the right subtree has height 1

After AVL operation



AVL tree?





YES

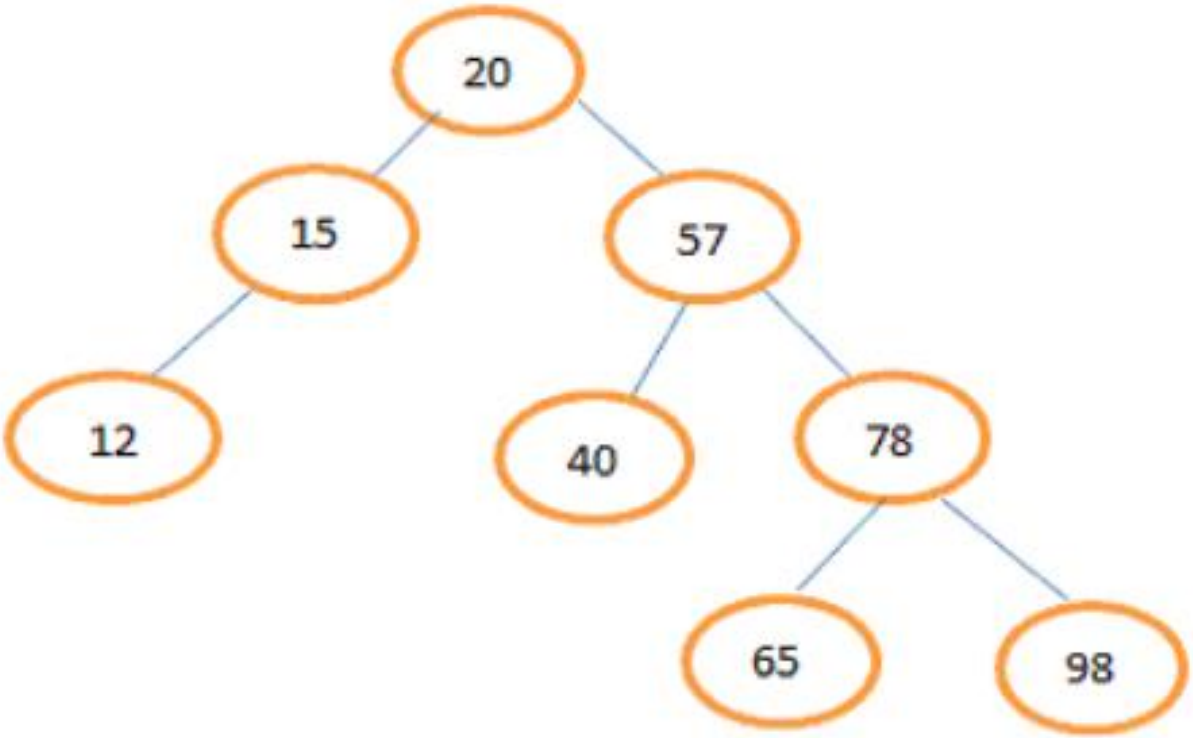
Each left sub-tree has height 1 greater than each right sub-tree

NO

Left sub-tree has height 3, but right sub-tree has height 1

Is the following tree AVL balanced?

Example:



YES! Why?

For the leaf node 12, 40, 65 and 98 left and right subtrees are empty so difference of heights of their subtrees is zero.

For node 20 height of left subtree is 2 and height of right subtree is 3 so difference is 1.

For node 15 height of left subtree is 1 and height of right subtree is 0 so difference is 1.

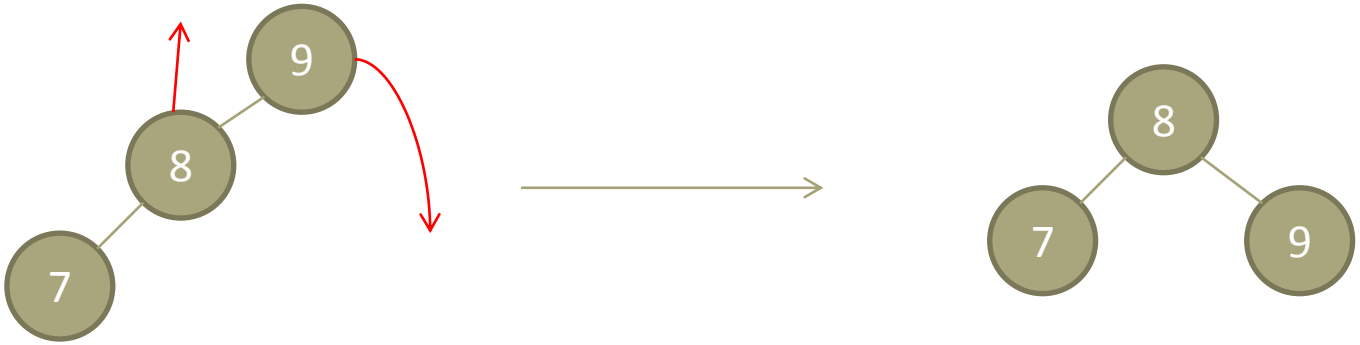
For node 57 height of left subtree is 1 and height of right subtree is 2 so difference is 1.

For node 78 height of left subtree is 1 and height of right subtree is 1 so difference is 0.

Each node of an AVL tree has a balance factor, which is defined as the difference between the heights of left subtree and right subtree of a node.

4 type operations

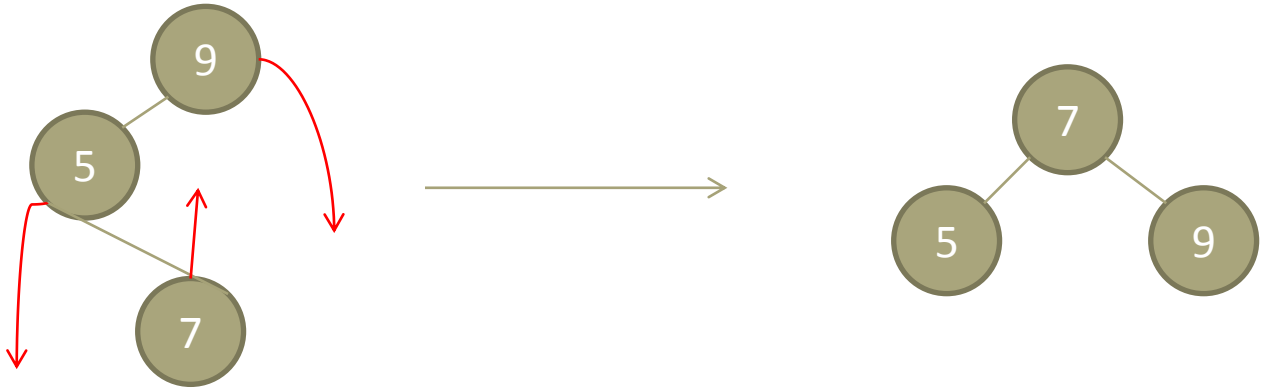
1. LL (Left-Left)



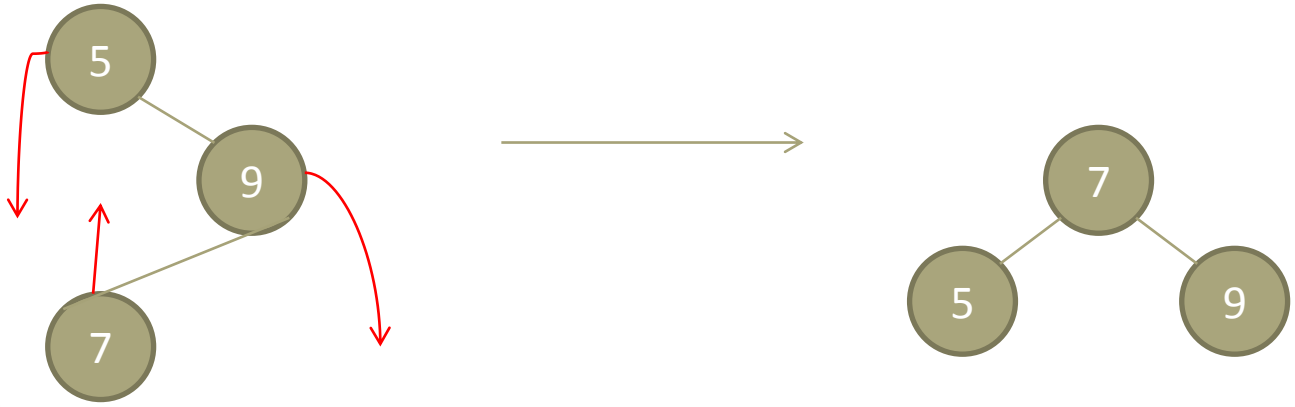
2. RR (Right-Right)

4 type operations

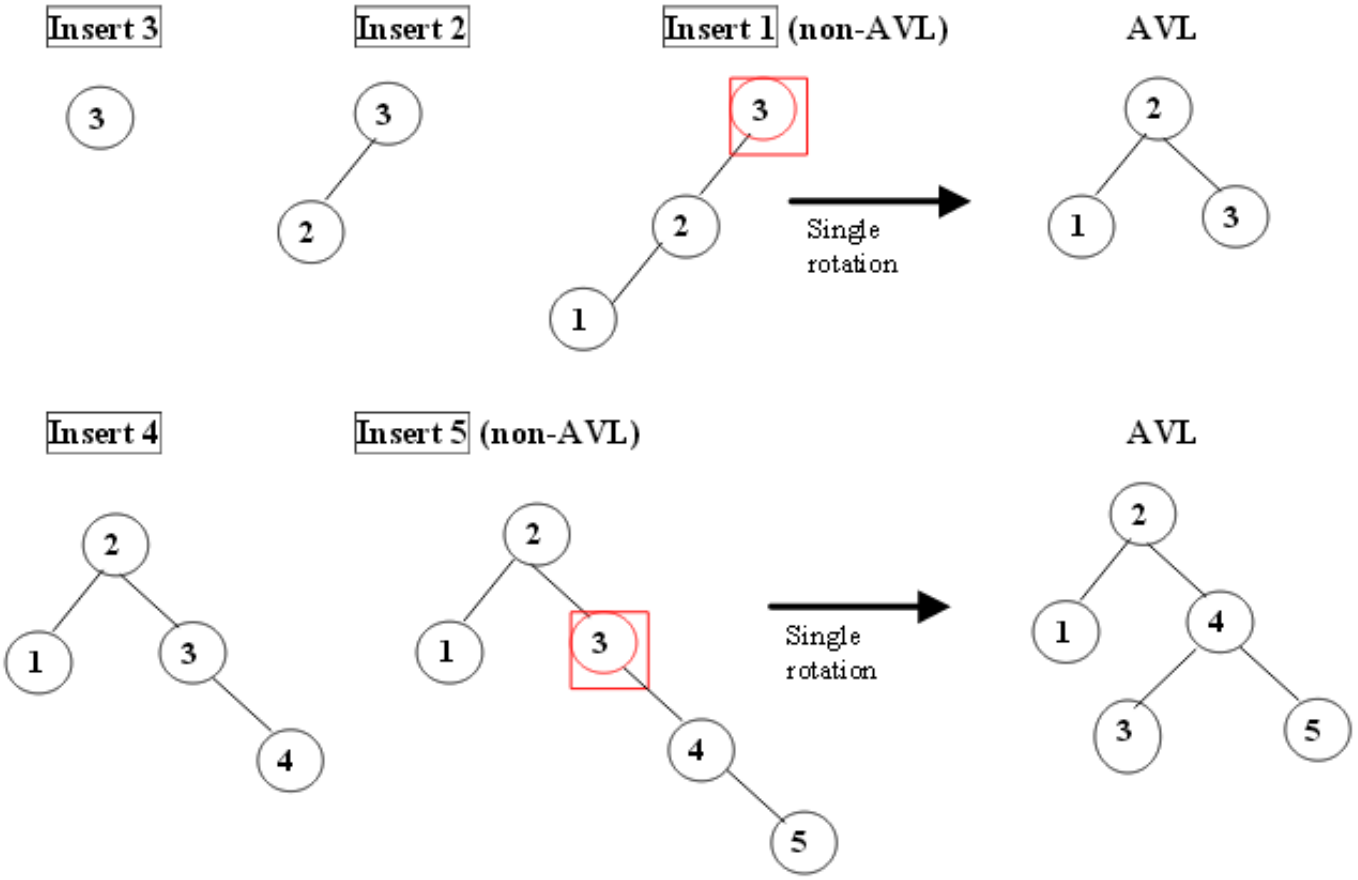
3. LR (Left-Right)

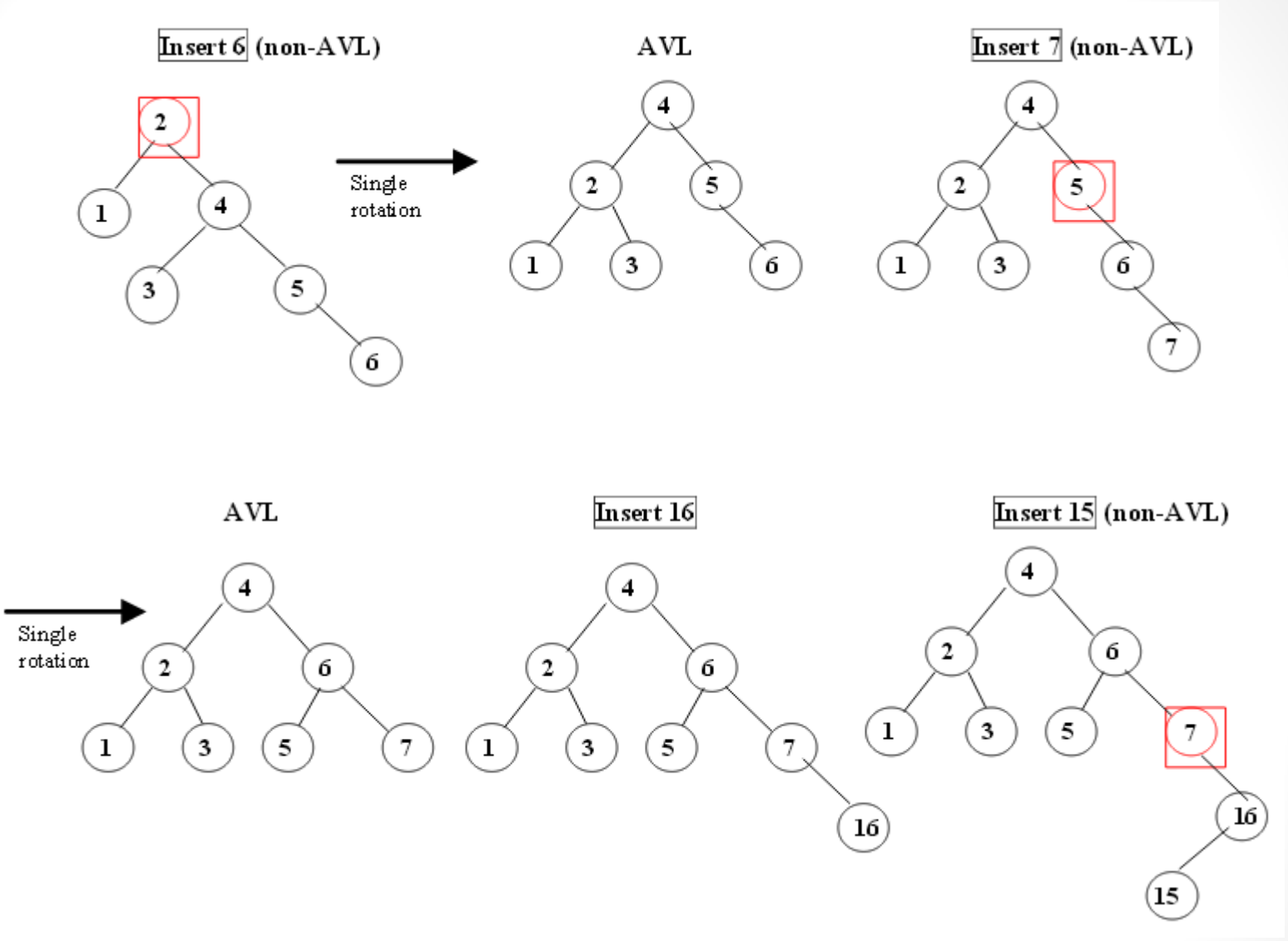


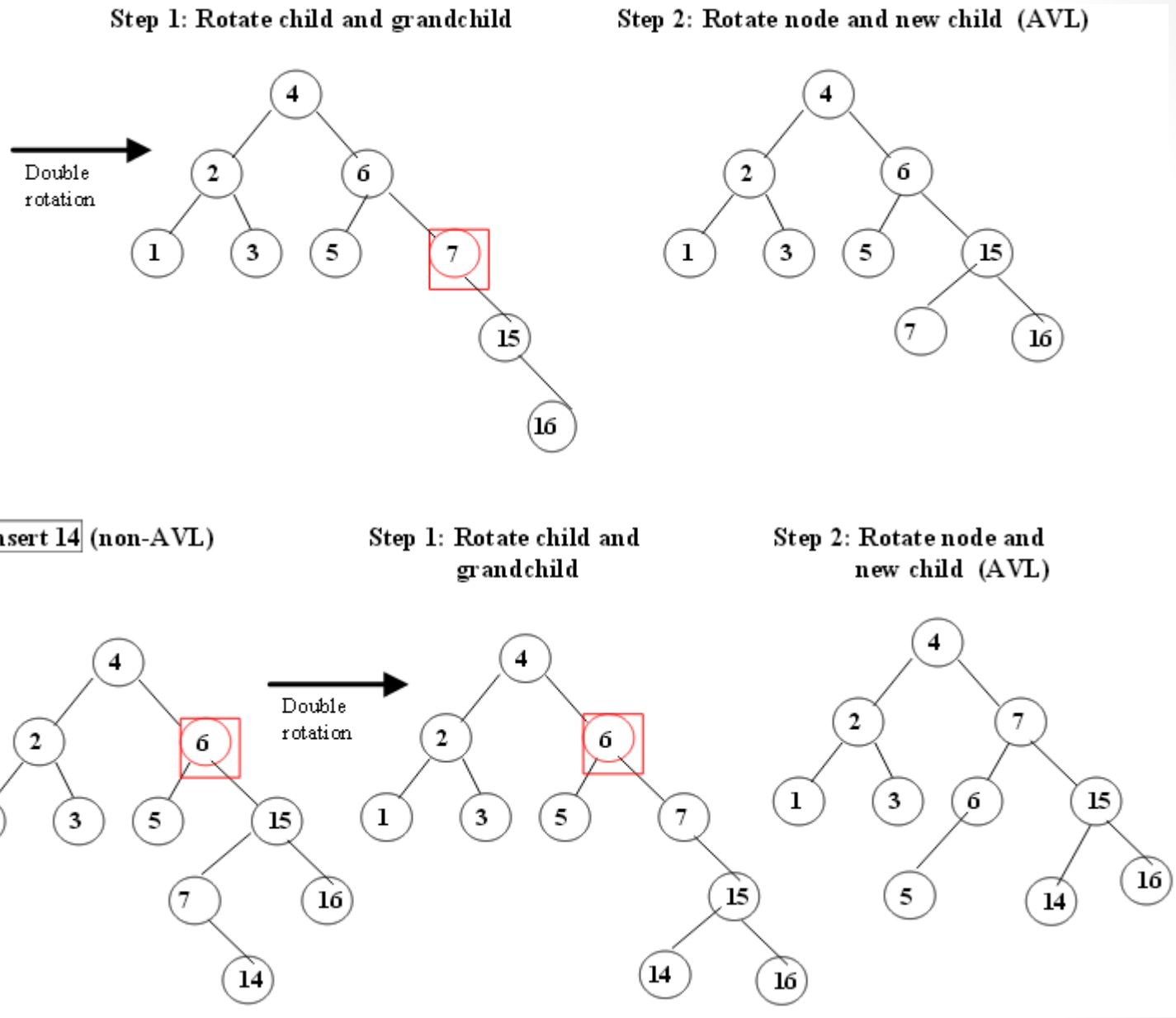
2. RL (Right-Left)



AVL Trees Example




Insert 6 (non-AVL)
AVL
Insert 7 (non-AVL)
Single rotation
AVL
Insert 16
Insert 15 (non-AVL)
Single rotation


Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
Double rotation
4
2
6
1
3
5
7
15
16
Insert 14 (non-AVL)
Step 1: Rotate child and grandchild
Step 2: Rotate node and new child (AVL)
Double rotation
14