BITS Pilani
Hyderabad Campus

Data Structures and Algorithms Design

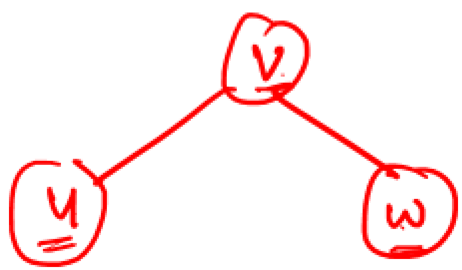
Febin.A.Vahab

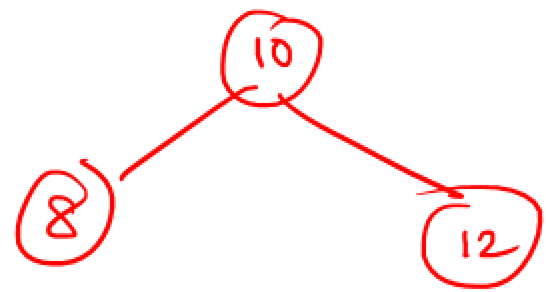
SESSION 6 -PLAN

Online Sessions(#)	List of Topic Title	Text/Ref Book/external resource
6	Binary Search Tree - Motivation with the task of Searching and Binary Search Algorithm, Properties of BST, Searching an element in BST, Insertion and Removal of Elements,	T1: 3.1

Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property
  - Let ***u***, ***v***, and ***w*** be three nodes such that ***u*** is in the left subtree of ***v*** and ***w*** is in the right subtree of ***v***. We have $\boldsymbol{key}(\boldsymbol{u}) \le \boldsymbol{key}(\boldsymbol{v}) \le \boldsymbol{key}(\boldsymbol{w})$





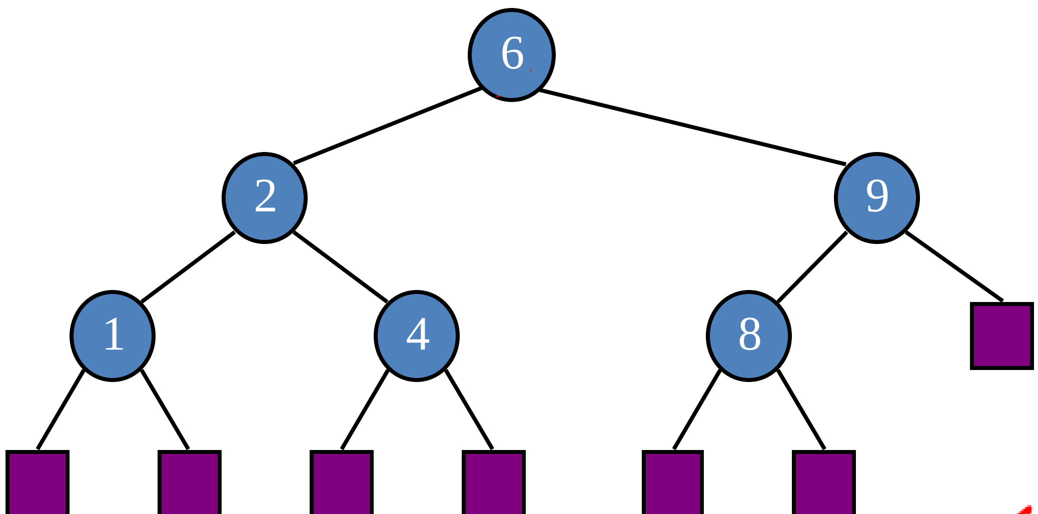
8 10 12

Binary Search Tree

- An inorder traversal of a binary search trees visits the keys in increasing order

LNR

4



0	1	2	3	4	5
	6	2	9	1	4

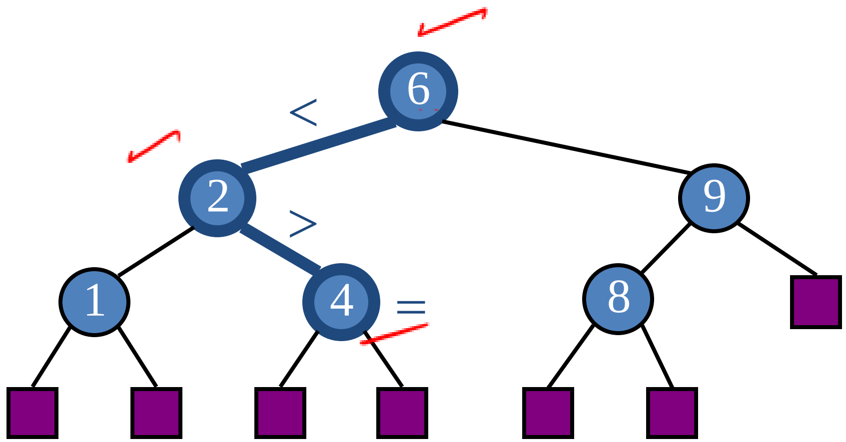
1 2 4 6 8 9

Binary Search Tree- Search

- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY
- Example: findElement(4)

- External nodes do not store items

Binary Search Tree- Search



Binary Search Tree- Search

Algorithm ***findElement(k, v)***

- Input: A search key k, and a node v of a binary search tree T
- Output: A node w of the subtree T (v) of T rooted at v, such that either w is an internal node storing key k or w is the external node where an item with key k would belong if it existed

```
if T.isExternal (v)
    return NO_SUCH_KEY
if k < key(v)
    return findElement(k, T.leftChild(v))
else if k = key(v)
    return element(v)
else { k > key(v) }
    return findElement(k, T.rightChild(v))
```

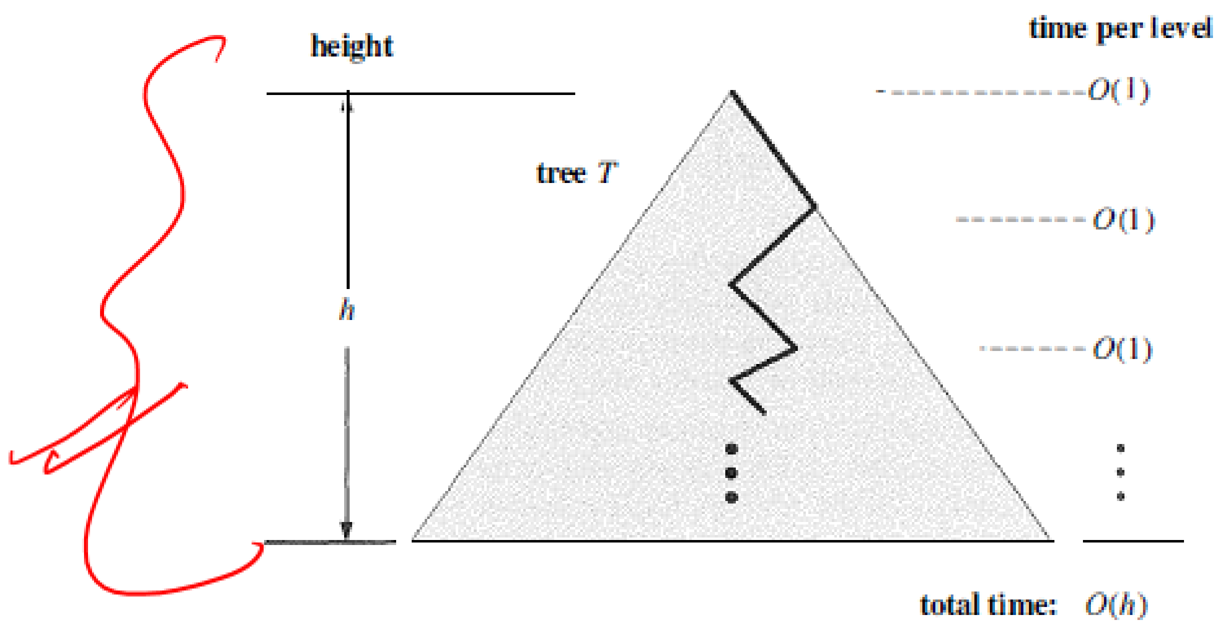
Analysis of Binary Tree Searching

- The binary tree search algorithm executes a constant number of primitive operations for each node it traverses in the tree.
- Each new step in the traversal is made on a child of the previous node.
- That is, the binary tree search algorithm is performed on the nodes of a path of T that starts from the root and goes down one level at a time.
- Thus, the number of such nodes is bounded by h + 1, where h is the height of T.

Analysis of Binary Tree Searching

- In other words, since we spend O(1) time per node encountered in the search, method findElement (or any other standard search operation) runs in O(h) time, where h is the height of the binary search tree T used to implement the dictionary D.
- ie. The running time of searching in a binary search tree T is proportional to the height of T . The height of a tree with n nodes can be **O(log n)**

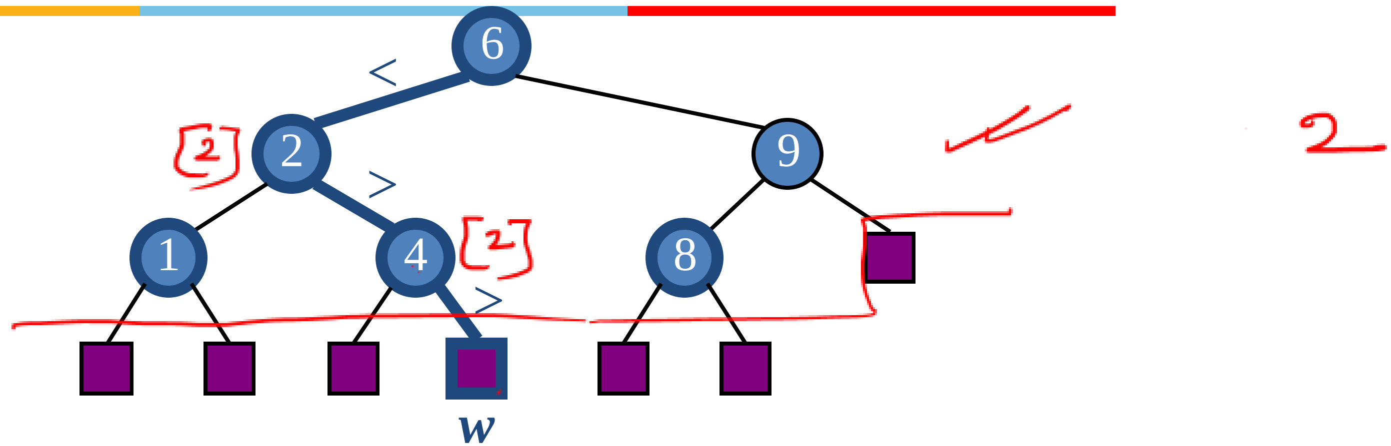
Analysis of Binary Tree Searching

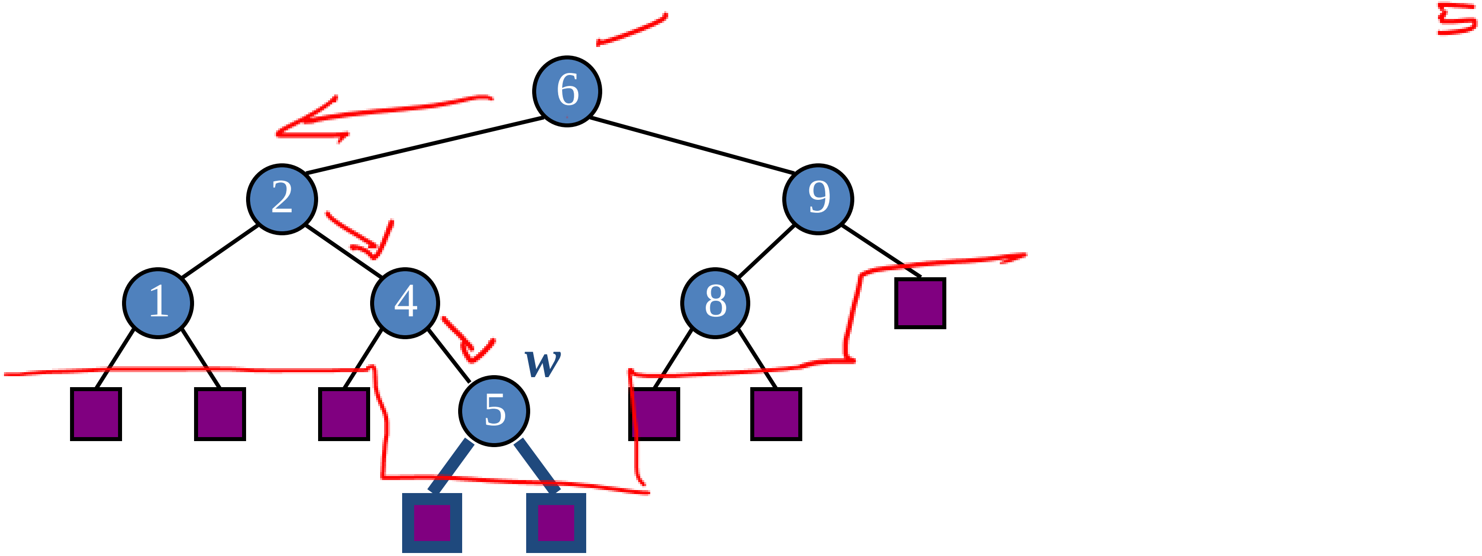


Binary Search Tree- Insertion

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5

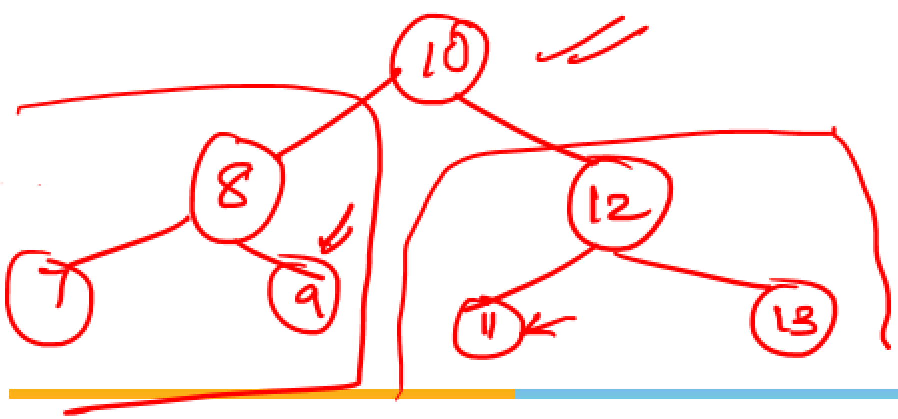
Binary Search Tree- Insertion





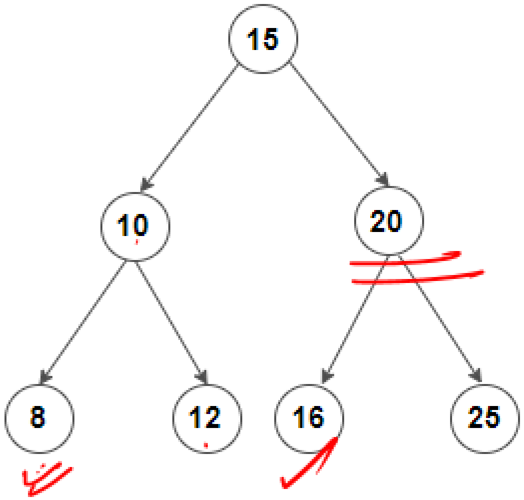
In-order Successor and Predecessor

- In a Binary Search Tree, the successor of a given key is the smallest number which is larger than the key.
- In the same way, a predecessor is the largest number which is smaller than the key.
- If X has two children then its in-order predecessor is the maximum value in its left subtree and its in-order successor the minimum value in its right subtree.



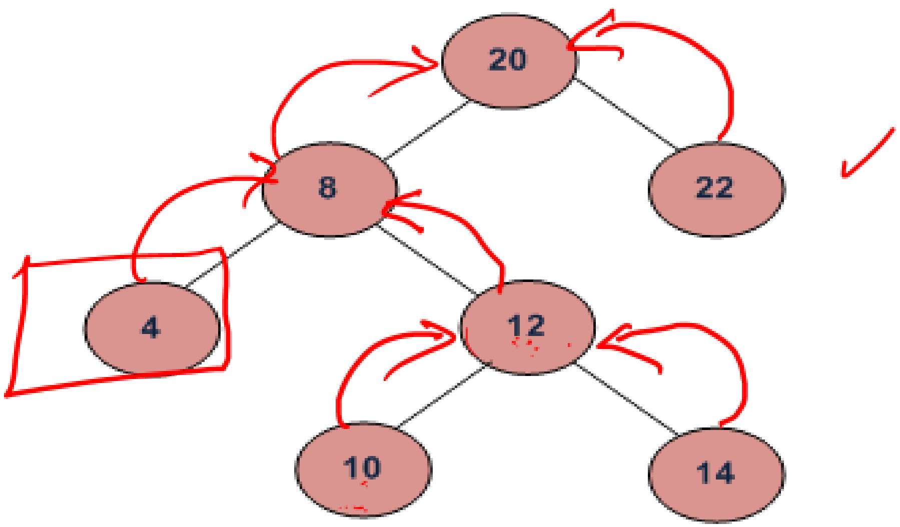
Predecessor

- Inorder predecessor of 8 doesn't exist
- Inorder predecessor of 20 is 16
- Inorder predecessor of 12 is 10



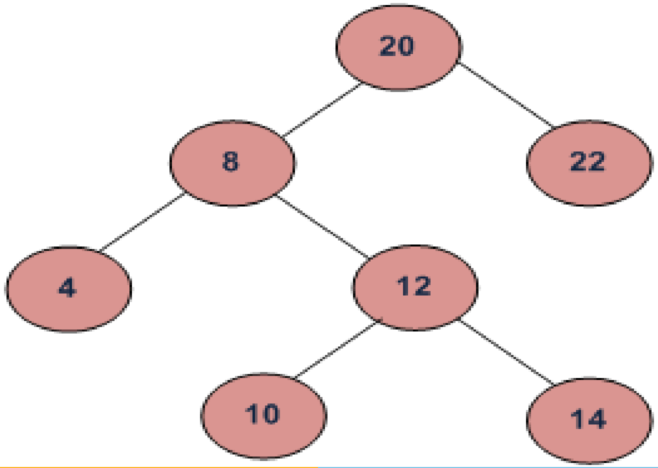
Predecessor

- Inorder predecessor of a node is a node with maximum value in its left subtree. i.e left subtree's right most child.
- If left subtree doesn't exists, then predecessor is one of the ancestors. Travel up using the parent pointer until you see a node which is right child of it's parent. The parent of such a node is the predecessor.



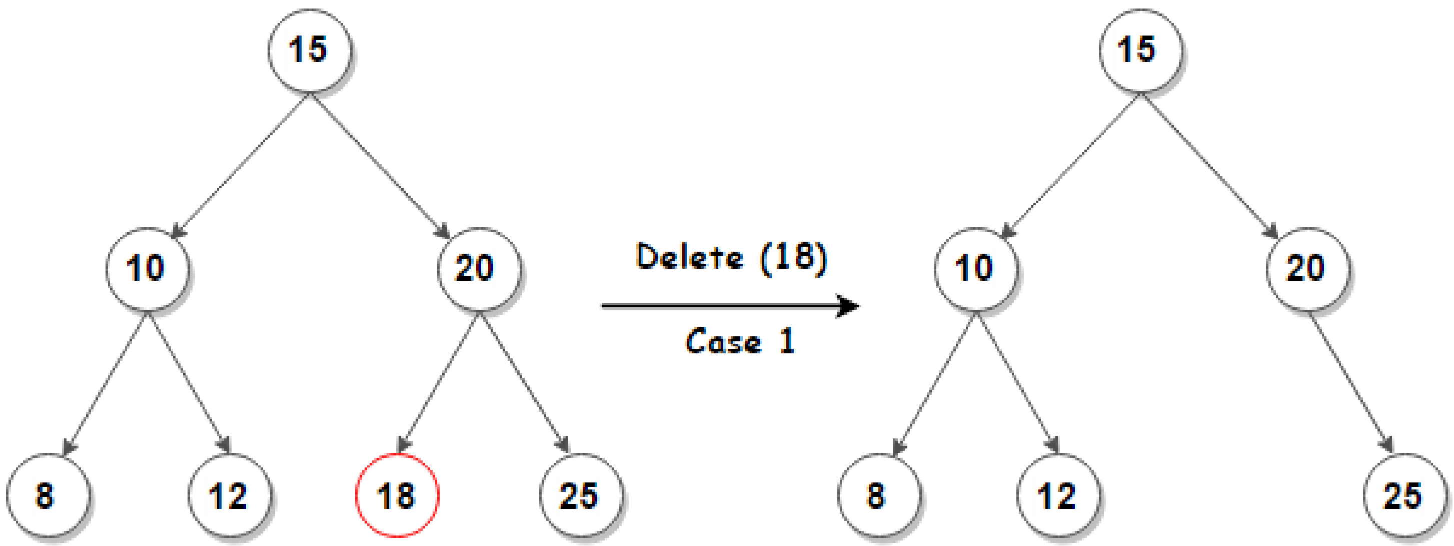
Successor

- If right subtree of node is not NULL, then succ lies in right subtree.Go to right subtree and return the node with minimum key value in right subtree.i.e **right subtree's left most child**.
- If right subtree of node is NULL, then succ is one of the ancestors. Travel up using the parent pointer until you see a node which is left child of it's parent. The parent of such a node is the succ.



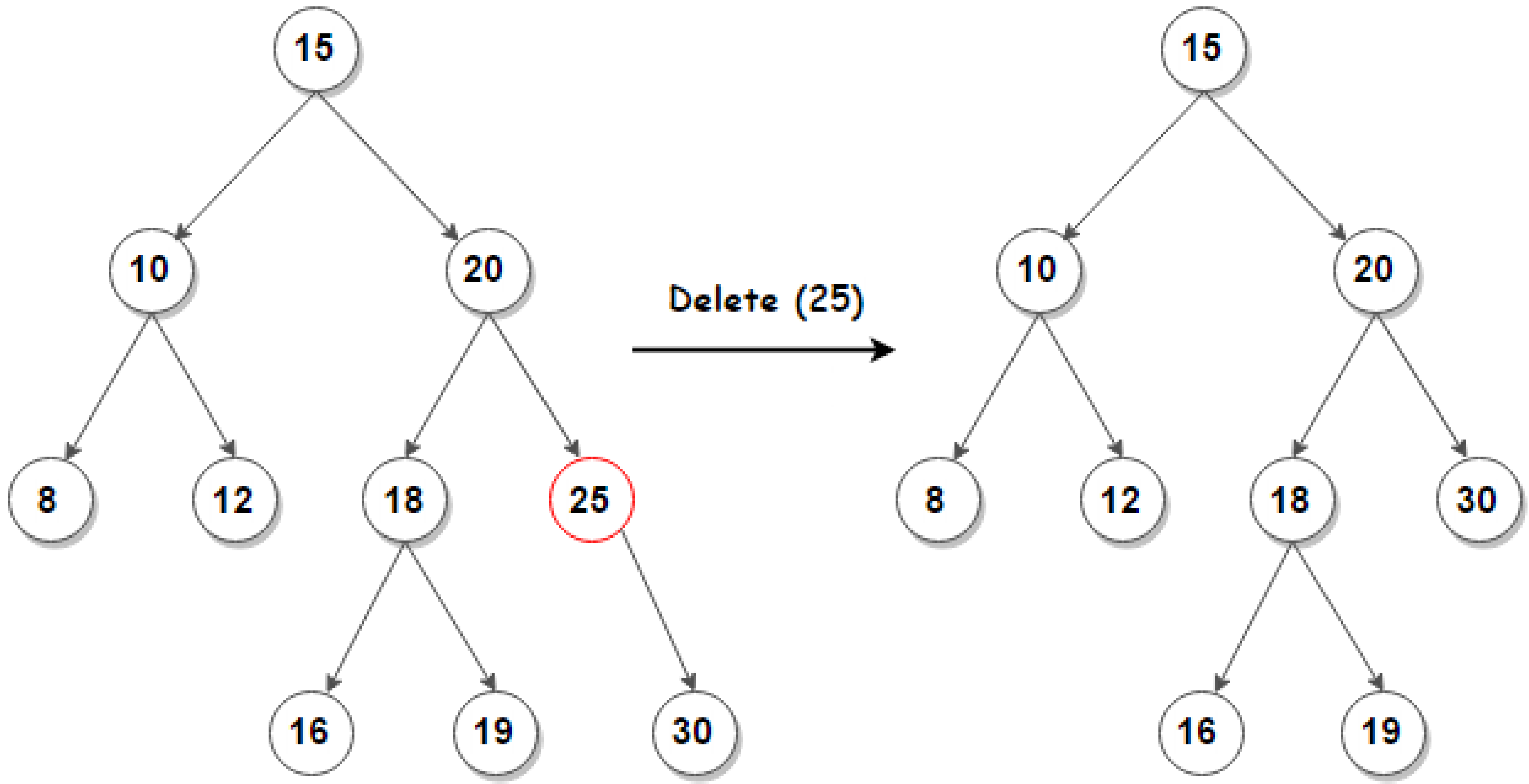
Binary Search Tree-Deletion

- Deleting a node with no Children

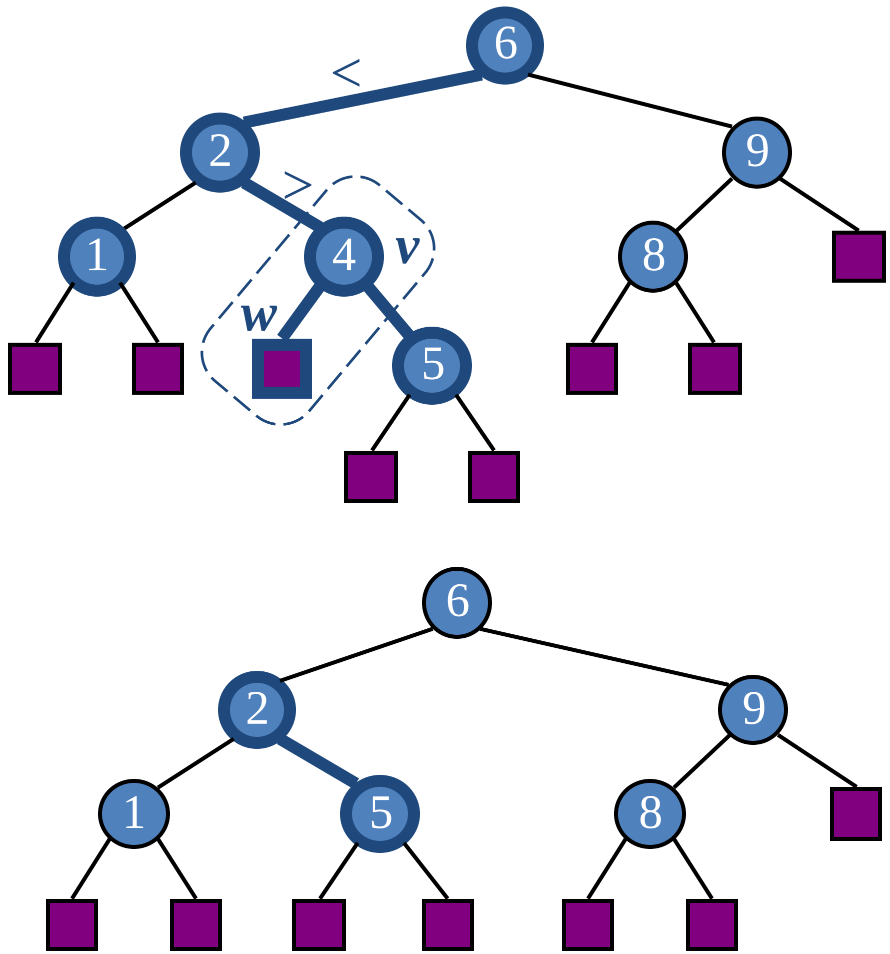


Binary Search Tree-Deletion

- **Deleting a node with 1 child**: Remove the node and replace it with its child

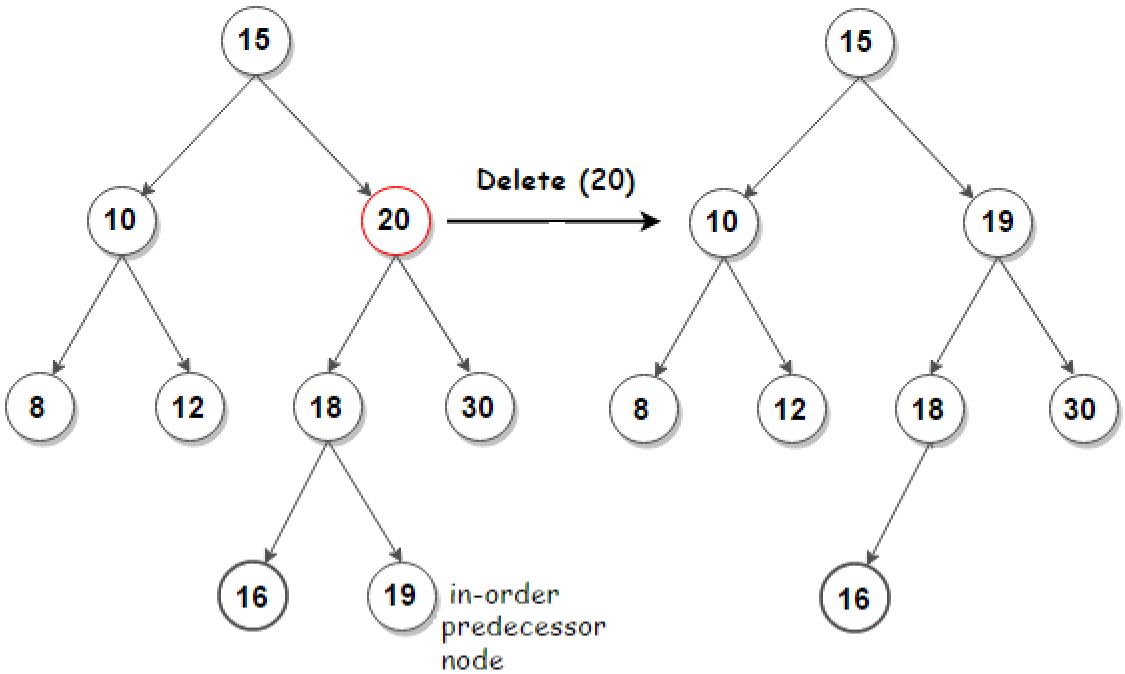


Binary Search Tree-Deletion



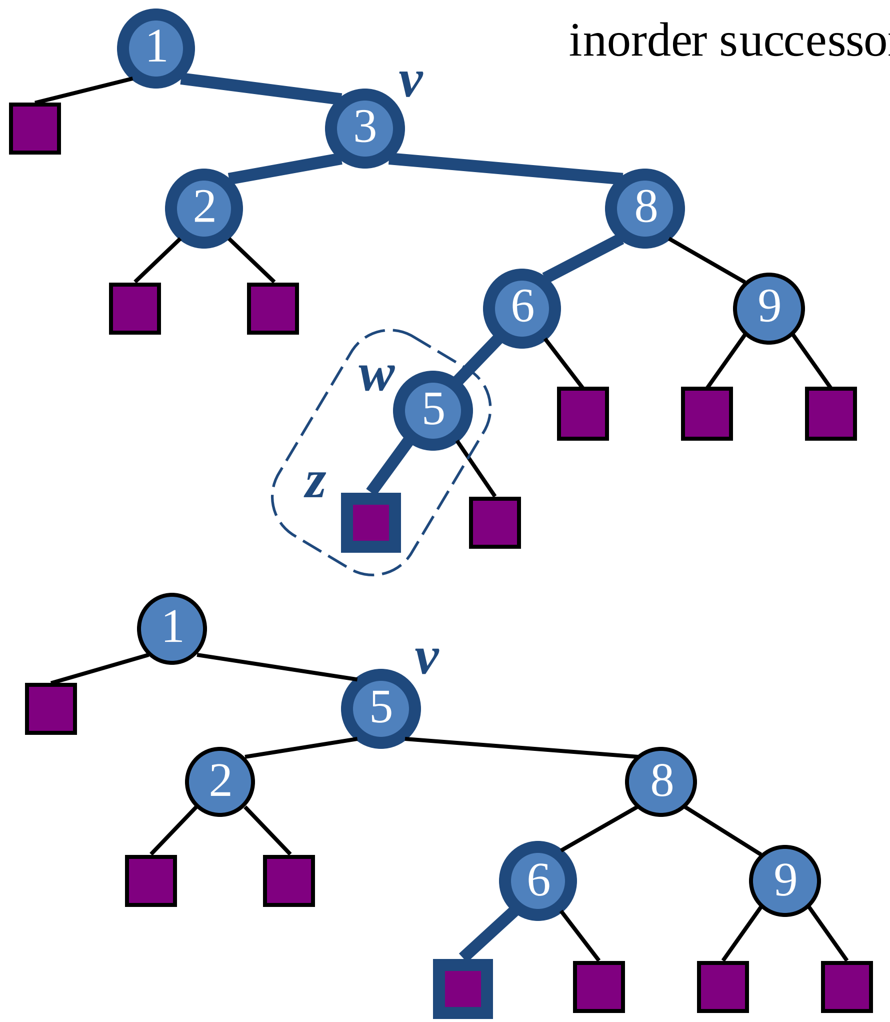
Binary Search Tree-Deletion

- Deleting a node with 2 children: Replace the node with its inorder successor(Predecessor)



Binary Search Tree-Deletion

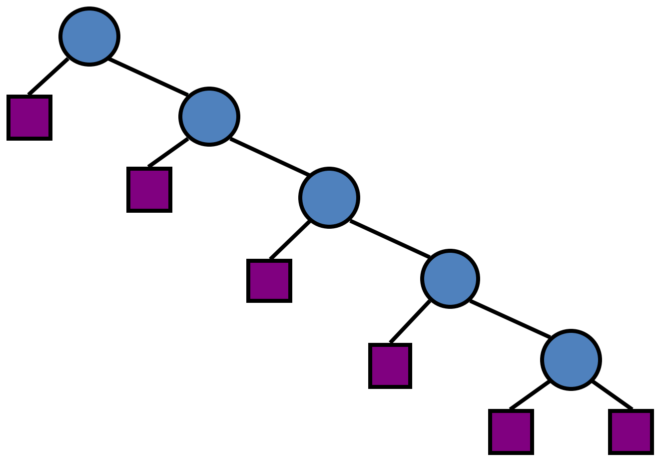
Delete node 3: Replace the node with its inorder successor ,node 5

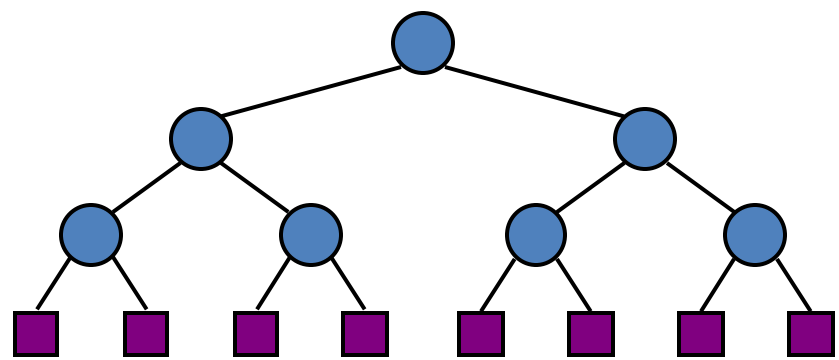


Performance

- Consider a BST with n items and height h
- The space used is O(n)
- Methods findElement, insertItem and removeElement take O(h) time
- The height h is O(n) in the worst case and O(log n) in the best case

Binary Search Tree





Balanced tree

- The worst-case performance, a BST achieves for various operations is linear time, which is no better than the performance of sequence-based dictionary implementations (such as log files and lookup tables).
- A simple way of correcting this problem is balanced binary search tree.

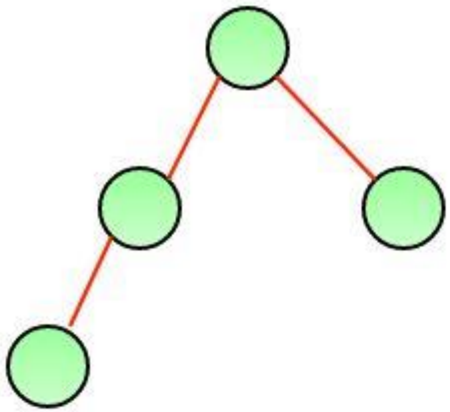
Balanced tree

- A balanced tree is a tree where every leaf is “not more than a certain distance” away from the root than any other leaf.
- Add a rule to the binary search tree definition that will maintain a logarithmic height for the tree
- Height-balance property

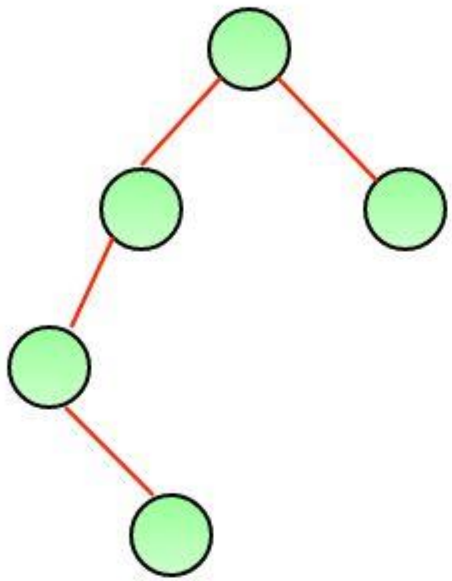
Balanced tree

- Height-Balance Property:

For every internal node v of T, the heights of the children of v can differ by at most 1 .



A height balanced tree

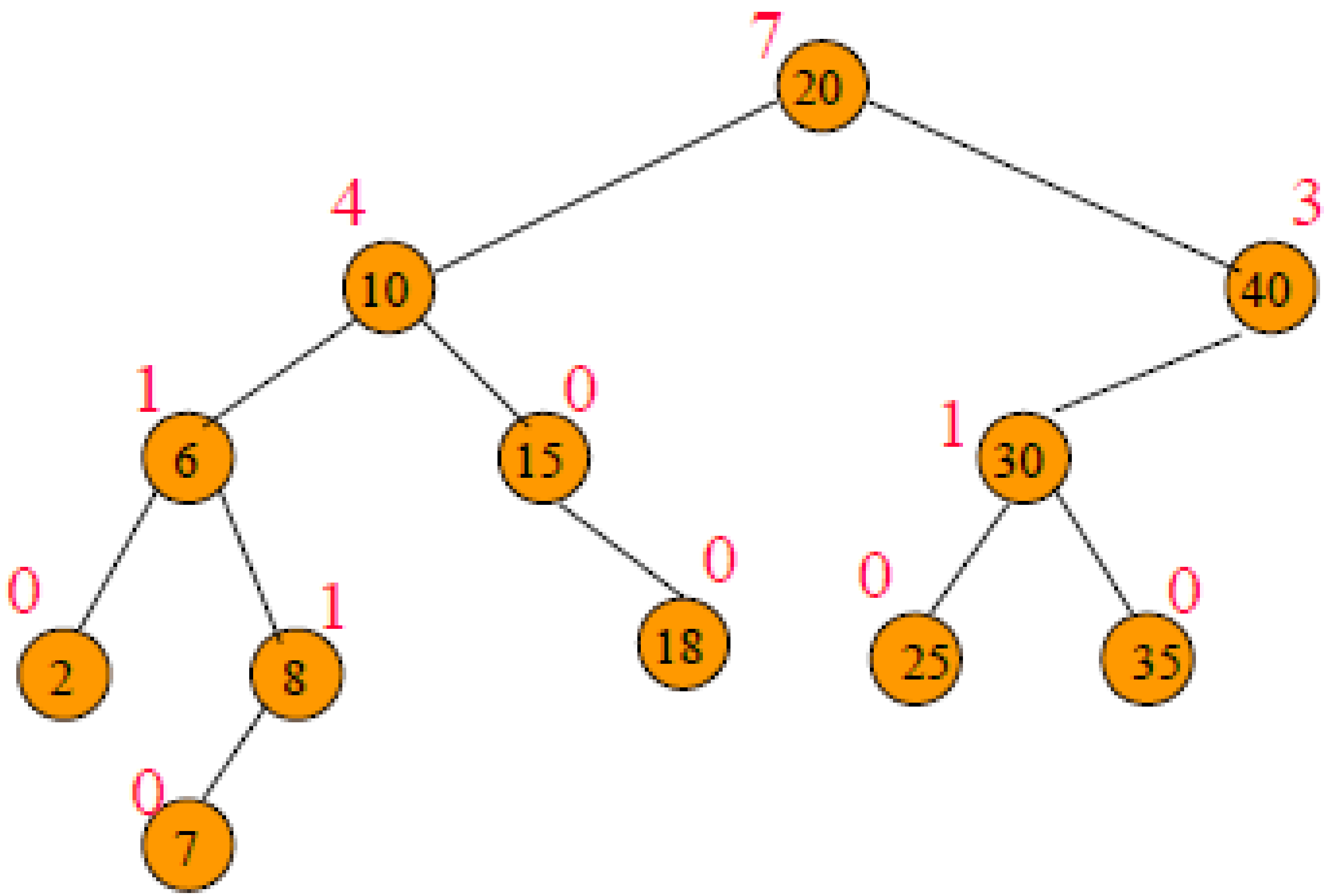


Not a height balanced tree

Leftsize

- Binary search tree.
- Each node has an additional field.
- leftSize = number of nodes in its left subtree

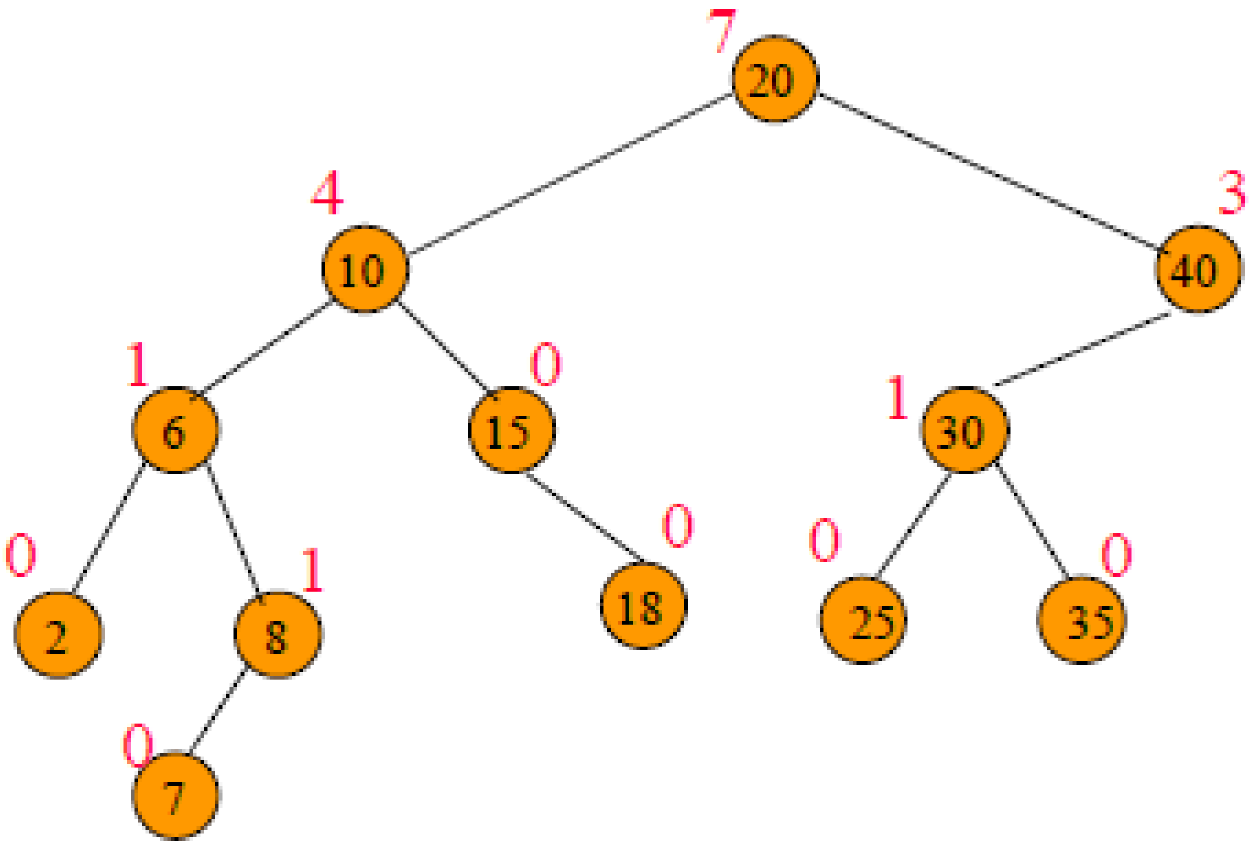
Leftsize



Rank

- Rank of an element is its position in inorder traversal (inorder = ascending key order).
- [2,6,7,8,10,15,18,20,25,30,35,40]
- rank(2) = 0
- rank(15) = 5
- rank(20) = 7
- **leftSize(x) = rank(x) with respect to elements in subtree rooted at x**

Rank



sorted list = [2,6,7,8,10,15,18,20,25,30,35,40]

Find k-th smallest element in BST

- The idea is to maintain rank of each node.
- We can keep track of elements in a subtree of any node while building the tree.
- Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Find k-th smallest element in BST

- Assume that the root is having N nodes in its left subtree.
  - If K = N + 1, root is K-th node.
  - If K > N, we continue our search in the right subtree for the (K – (N + 1))-th smallest element.
  - Else we will continue our search (recursion) for the Kth smallest element in the left subtree of root.
  - Note that we need the count of elements in left subtree only.
- Time complexity: O(h) where h is height of tree.

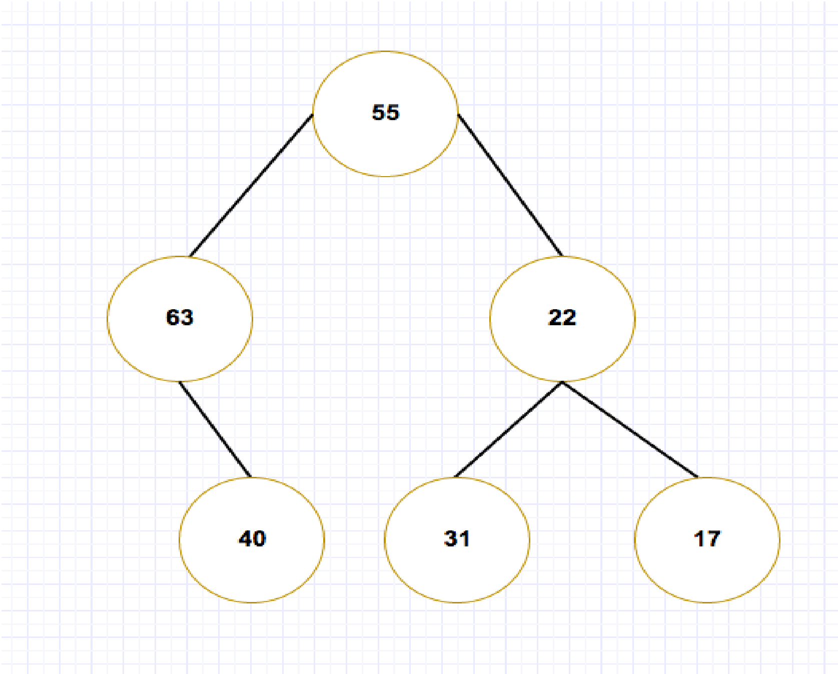
Find k-th smallest element in BST

1. *start*
2. ***if K = root.leftElements + 1***
   1. *root node is the K th node.*
   2. *goto stop*
3. ***else if K > root.leftElements***
   1. *K = K - (root.leftElements + 1)*
   2. *root = root.right*
   3. *goto start*
4. ***else***
   1. *root = root.left*
   2. *goto start*
5. *stop*

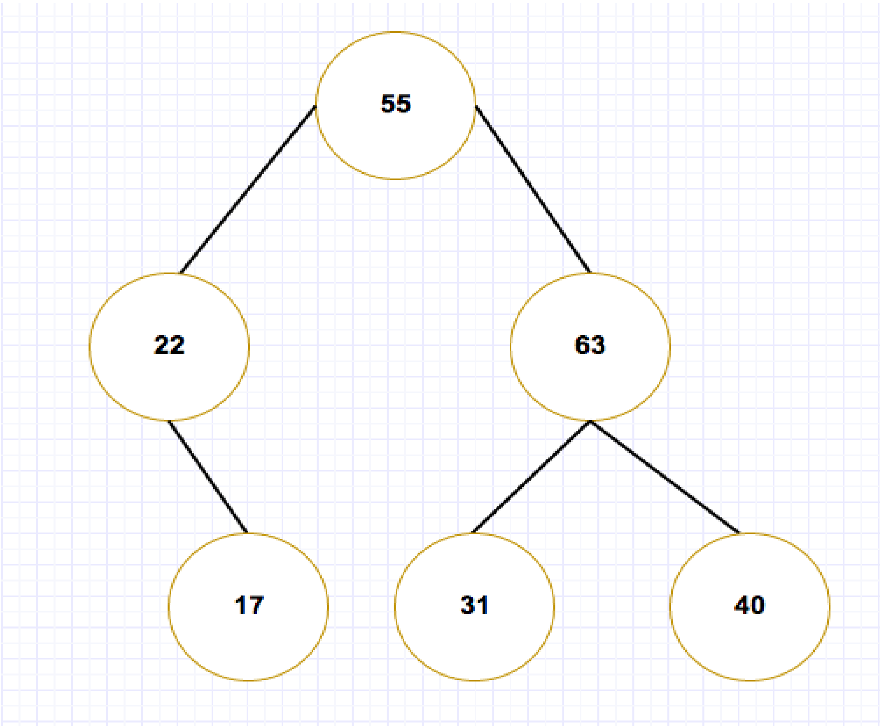
innovate achieve lead

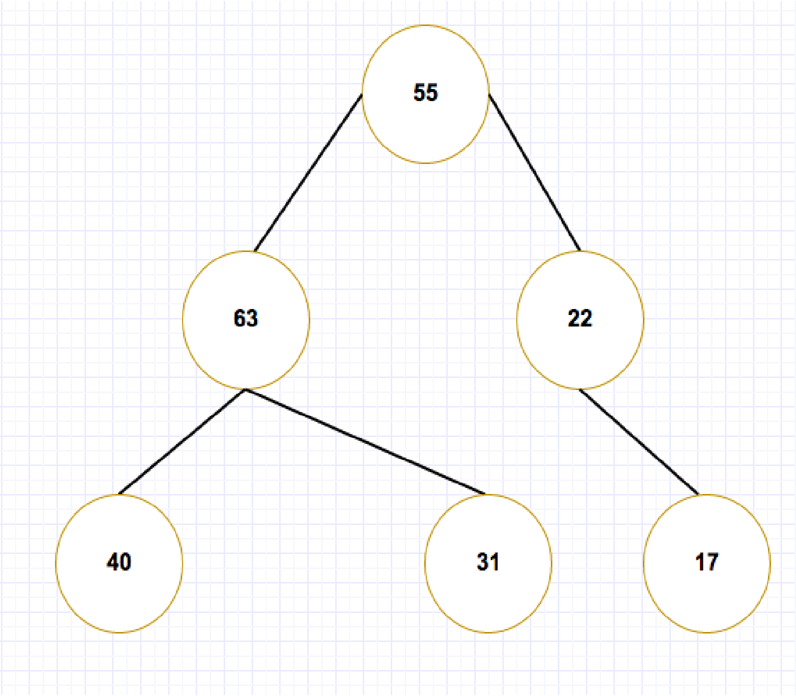
Qn:

- Suppose the keys 55,63,31,17,22,40 are inserted into a binary tree in that order. Which of the following is the BST that is formed?

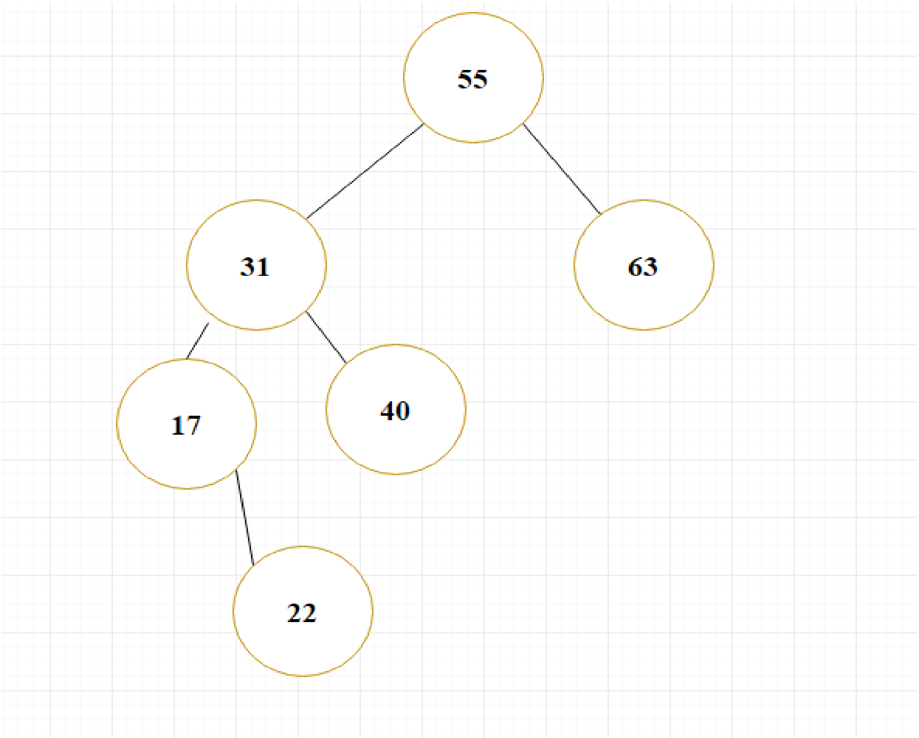


Qn :





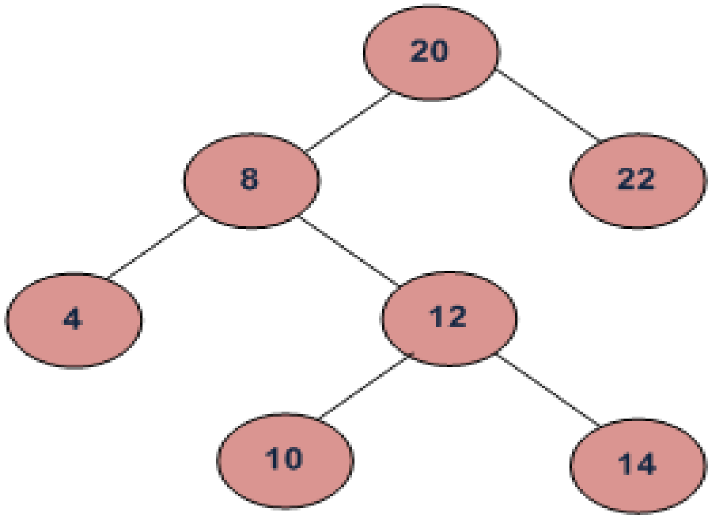
Qn :



Qn:LCA

- Given a binary search tree and two values say n1 and n2, write a program to find the least common ancestor. You may assume that both the values exist in the tree and n1<n2.

Qn:LCA-ANSWER



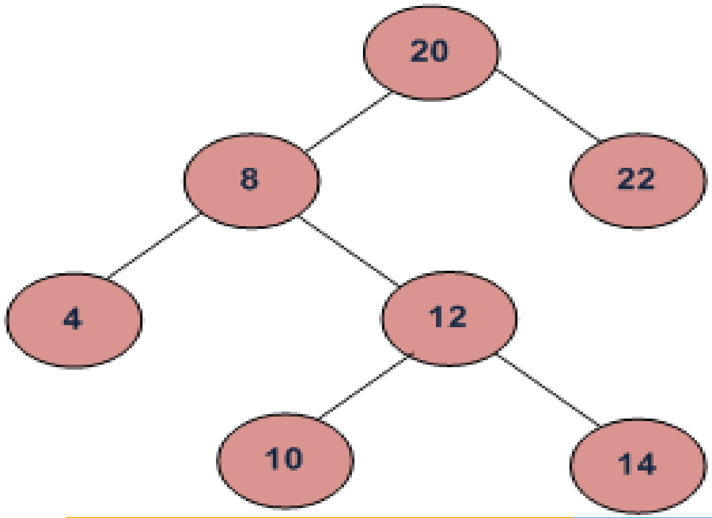
LCA of 10 and 14 is 12

LCA of 14 and 8 is 8

LCA of 10 and 22 is 20

Qn:LCA-ANSWER

- Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).
- The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root.



Qn:LCA-ANSWER

- We can solve this problem using BST properties. We can **recursively traverse** the BST from root.
- The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that n1 < n2).
- So just recursively traverse the BST, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

Qn:LCA-ANSWER

- Time complexity of above solution is O(h) where h is height of tree.

BITS Pilani
Hyderabad Campus

THANK YOU!!!