Sunbeam Institute of Information Technology
Pune and Karad

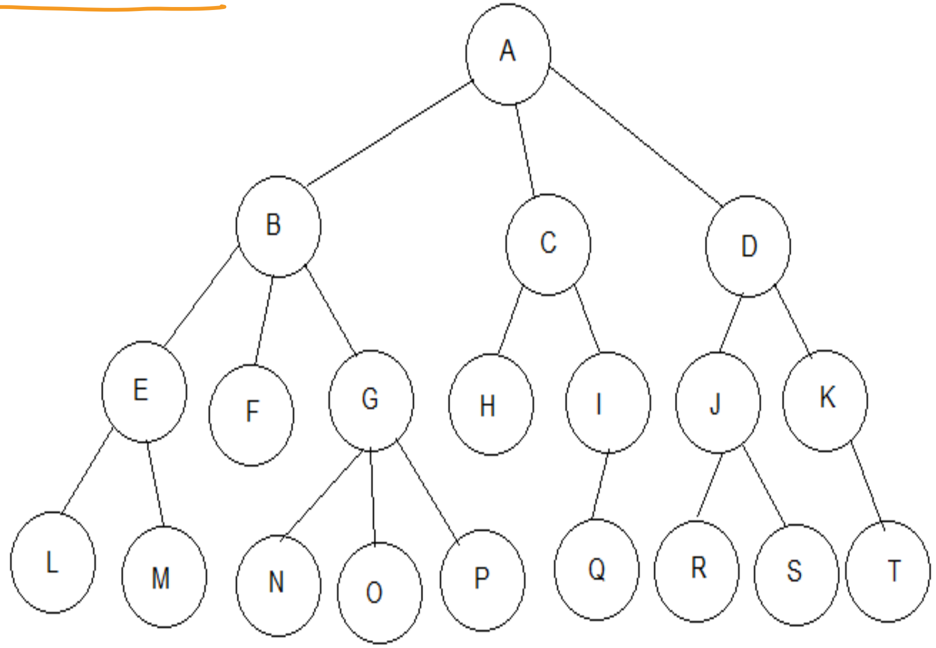
Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

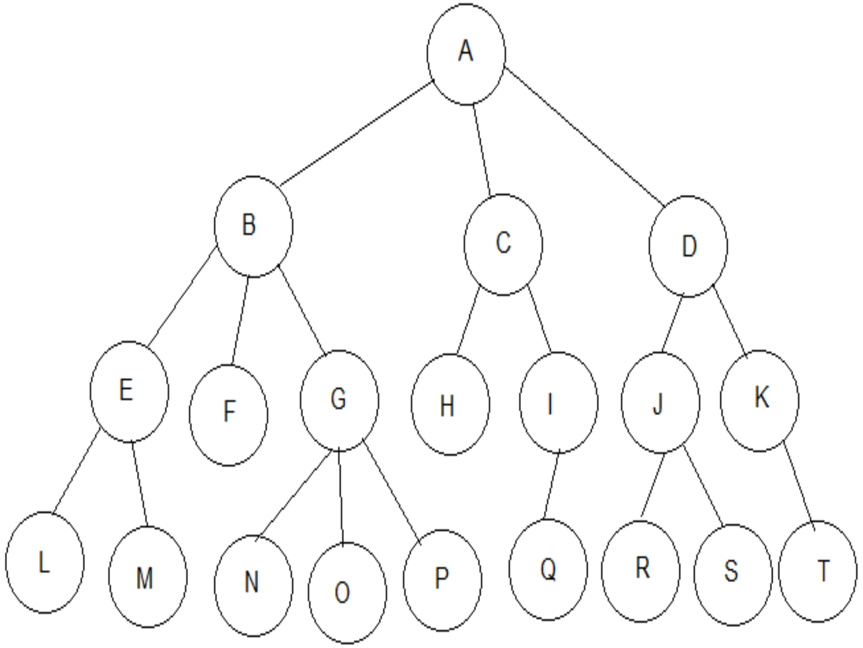
Tree - Terminologies

- **Tree** is a **non linear** data structure which is a finite set of nodes with one specially designated node is called as "**root**" and remaining nodes are partitioned into m disjoint subsets where each of subset is a tree..
- **Root** is a **starting point** of the tree.
- All nodes are connected in **Hierarchical manner (multiple levels)**.
- **Parent node:-** having other child nodes connected
- **Child node:-** immediate descendant of a node
- **Leaf node:-**
  - Terminal node of the tree.
  - Leaf node does not have child nodes.
- **Ancestors:-** all nodes in the path from root to that node.
- **Descendants:-** all nodes accessible from the given node
- **Siblings:-** child nodes of the same parent



Tree - Terminologies

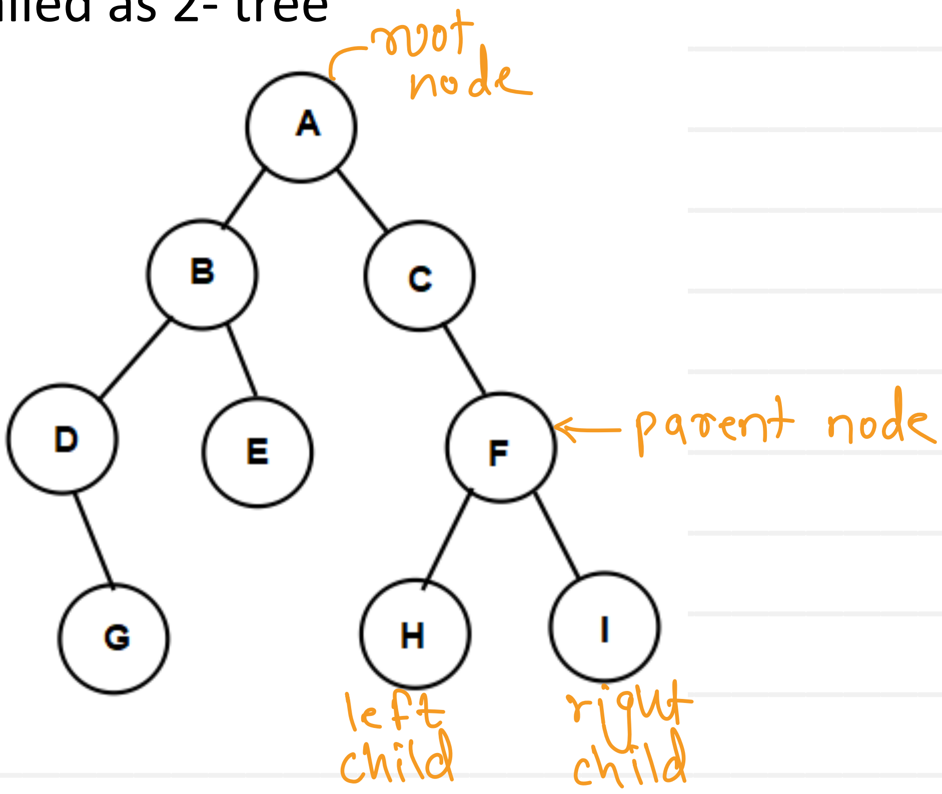
- **Degree of a node :-** number of child nodes for any given node.
- **Degree of a tree :-** Maximum degree of any node in tree.
- **Level of a node :-** indicates position of the node in tree hierarchy
  - Level of child = Level of parent + 1
  - Level of root = 0 / 1
- **Height of node :-** number of links from node to longest leaf.
- **Depth of node :-** number of links from root to that node
- **Height of a tree :-** Maximum height of a node
- **Depth of a tree :-** Maximum depth of a node



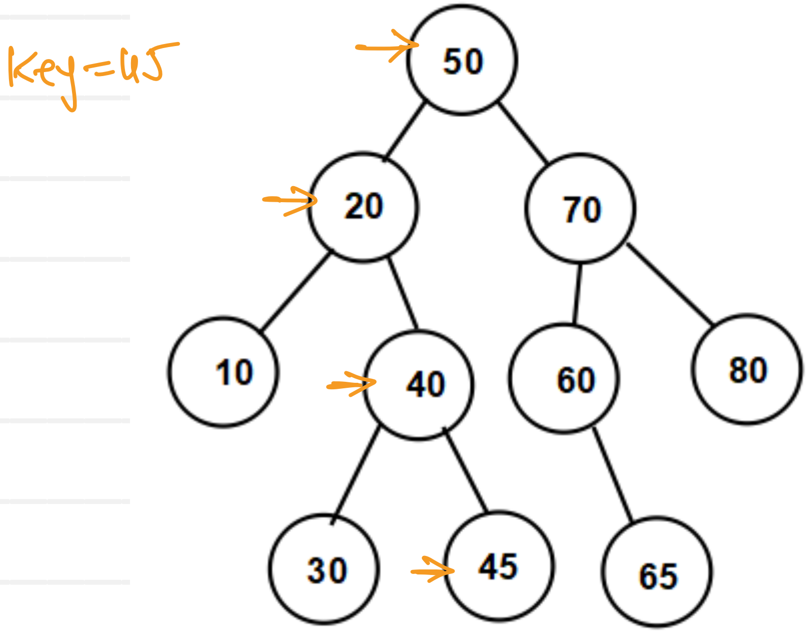
- Tree with zero nodes (ie empty tree) is called as “**Null tree**”. Height of Null tree is -1.
  - Tree can grow up to any level and any node can have any number of Childs.
  - That’s why operations on tree becomes un efficient.
  - Restrictions can be applied on it to achieve efficiency and hence there are different types of trees.

Tree - Terminologies

- **Binary Tree**
  - Tree in which each node has maximum two child nodes
  - Binary tree has degree 2. Hence it is also called as 2- tree



- **Binary Search Tree**
  - Binary tree in which left child node is always smaller and right child node is always greater or equal to the parent node.
  - Searching is faster
  - Time complexity : O(h) h – height of tree



Binary Search Tree - Implementation

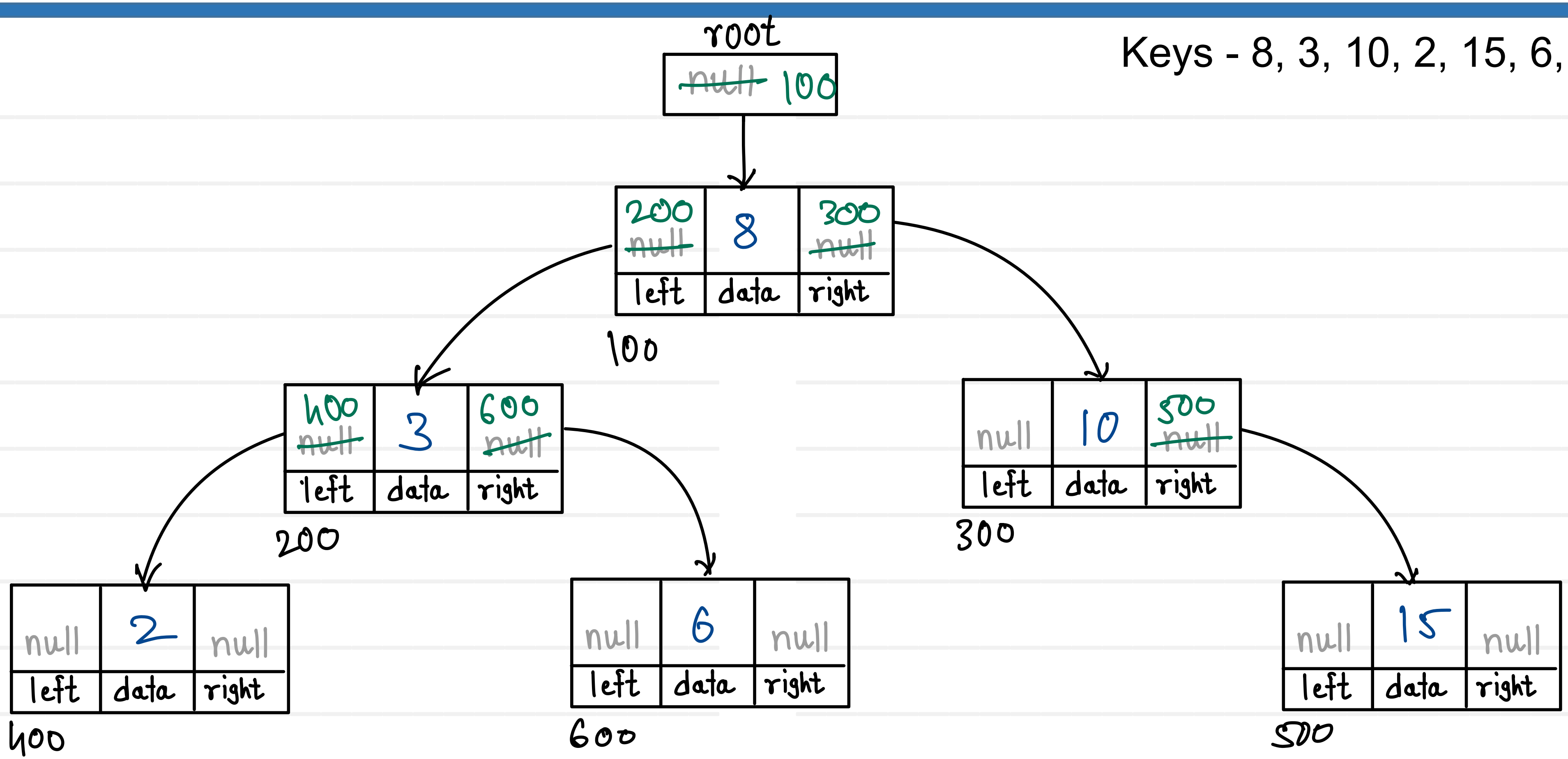
Node :
data : any type
left : reference of left child
right : reference of right child

```
Class Node {
    int data;
    Node left;
    Node right;
}
```

```
class BST {
    static class Node {
        int data;
        Node left;
        Node right;
    }
    Node root;
    BST( ) { . . . }
    void AddNode(int value) { . . . }
    void DeleteNode(int key) { . . . }
    Node searchNode(int key) { . . . }
    void display( ) { . . . }
}
```

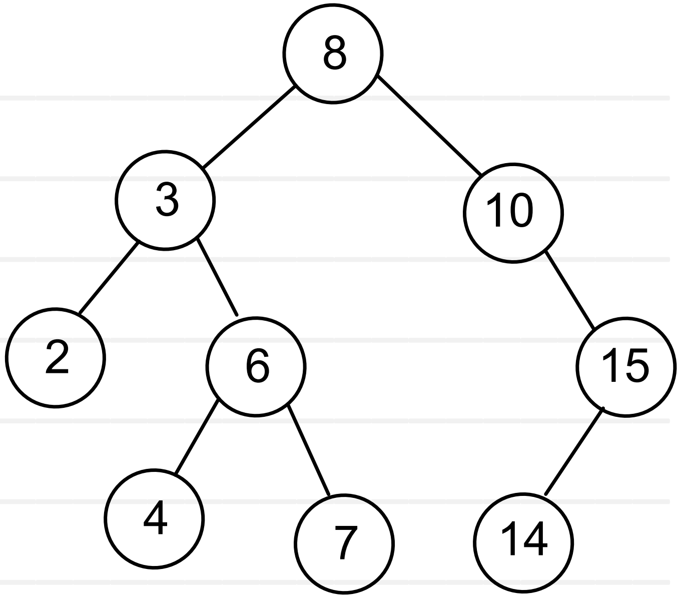
Binary Search Tree - Add Node

Keys - 8, 3, 10, 2, 15, 6, 14, 4, 7, 9

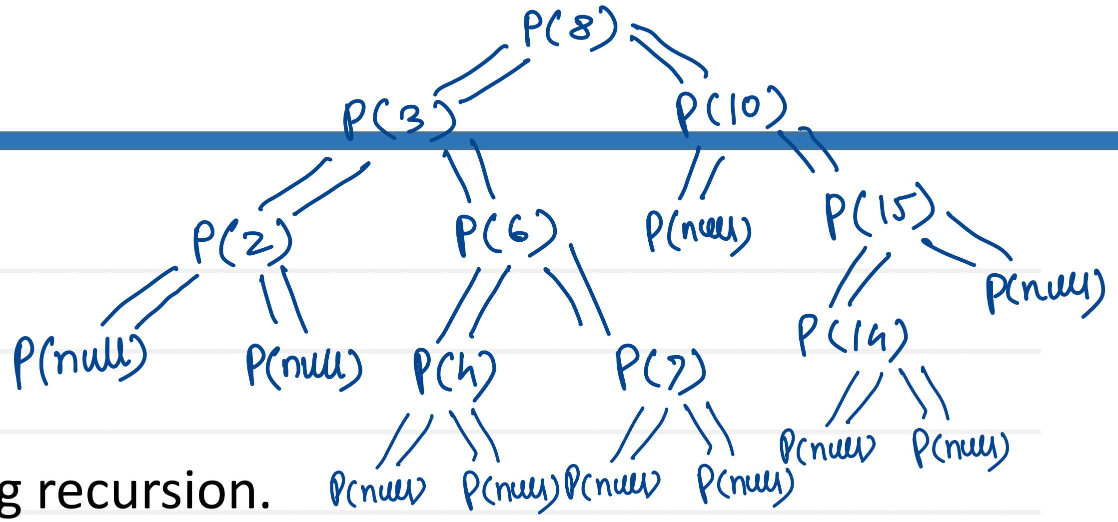


Binary Search Tree – Add Node

//1. create node for given value
//2. if BSTree is empty
    // add newnode into root itself
//3. if BSTree is not empty
    //3.1 create trav reference and start at root node
    //3.2 if value is less than current node data (trav.data)
        //3.2.1 if left of current node is empty
            // add newnode into left of current node
        //3.2.2 if left of current node is not empty
            // go into left of current node
    //3.3 if value is greater or equal than current node data (trav.data)
        //3.3.1 if right of current node is empty
            // add newnode into right of current node
        //3.3.2 if right of current node is not empty
            // go into right of current node
    //3.4 repeat step 3.2 and 3.3 till node is not getting added into BSTree

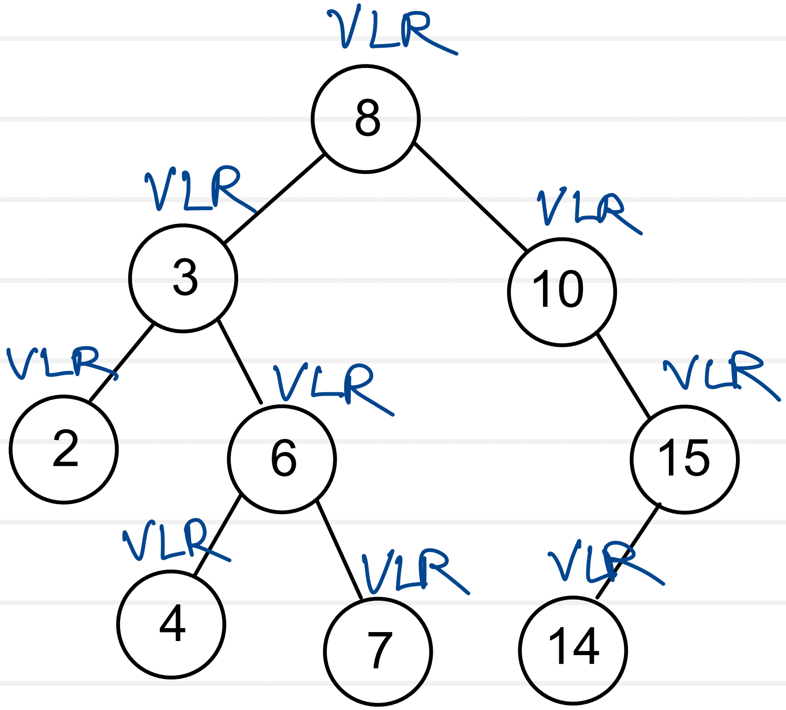


Tree Traversal Techniques



- **Pre-Order:-** V L R
- **In-order:-** L V R
- **Post-Order:-** L R V
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal

needs stack to store node pointers.



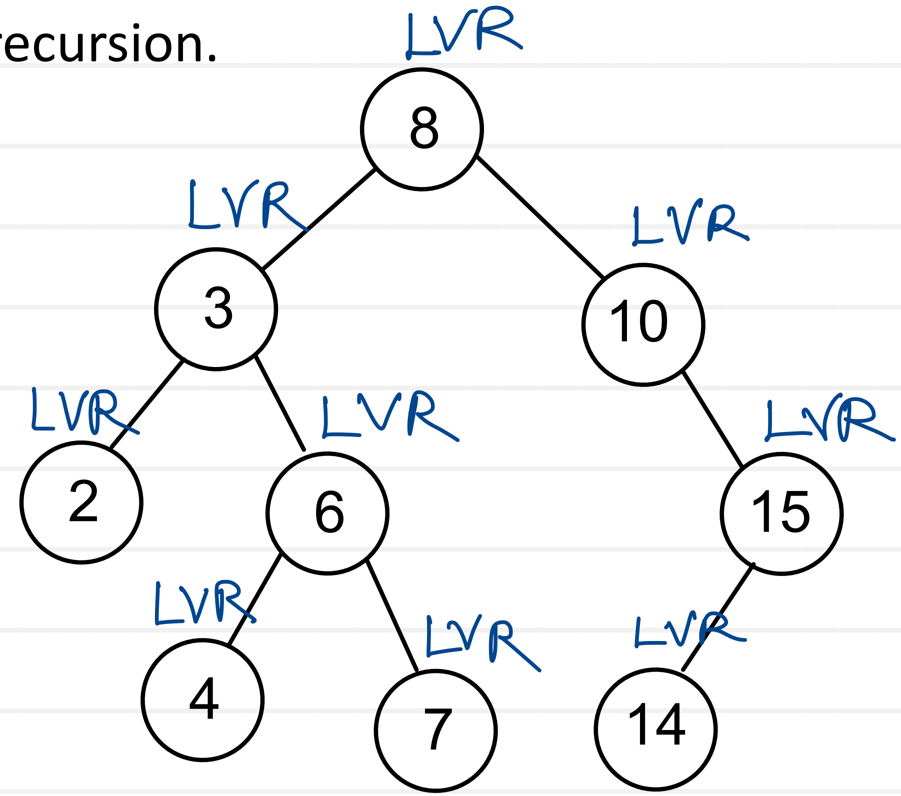
- **Pre-Order :-** 8, 3, 2, 6, 4, 7, 10, 15, 14

Tree Traversal Techniques

- **Pre-Order:-** V L R
- **In-order:-** L V R
- **Post-Order:-** L R V
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal

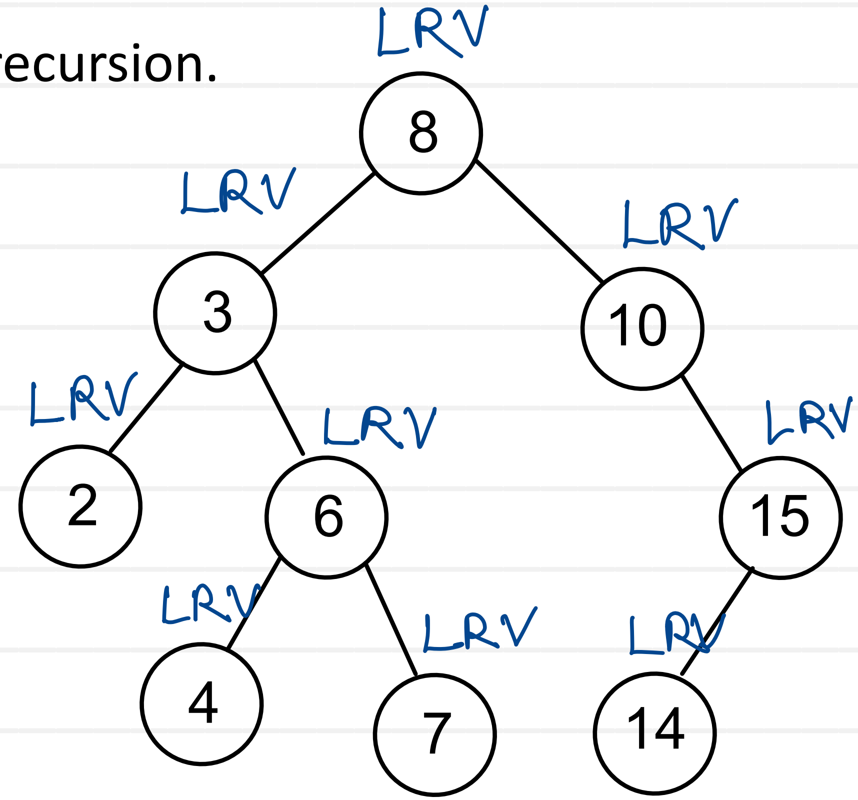
needs stack to store node pointers.

- **In-Order :-** 2, 3, 4, 6, 7, 8, 10, 14, 15



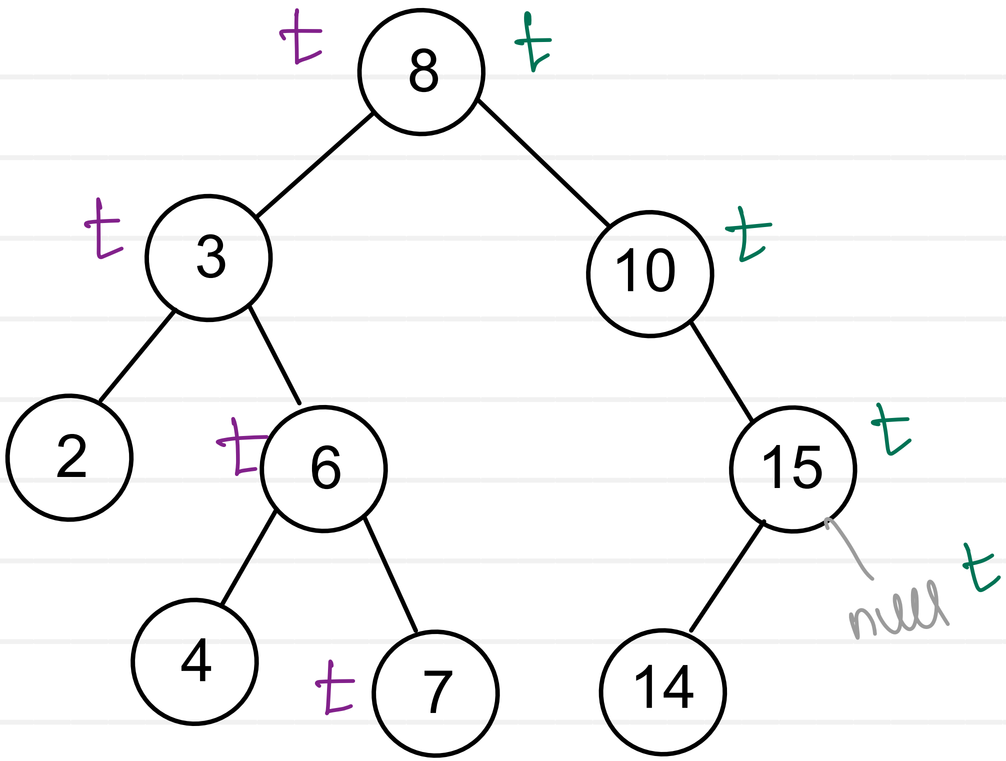
Tree Traversal Techniques

- **Pre-Order:-** V L R
- **In-order:-** L V R
- **Post-Order:-** L R V
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal needs stack to store node pointers.



- **Post-Order :-** 2, 4, 7, 6, 3, 14, 15, 10, 8

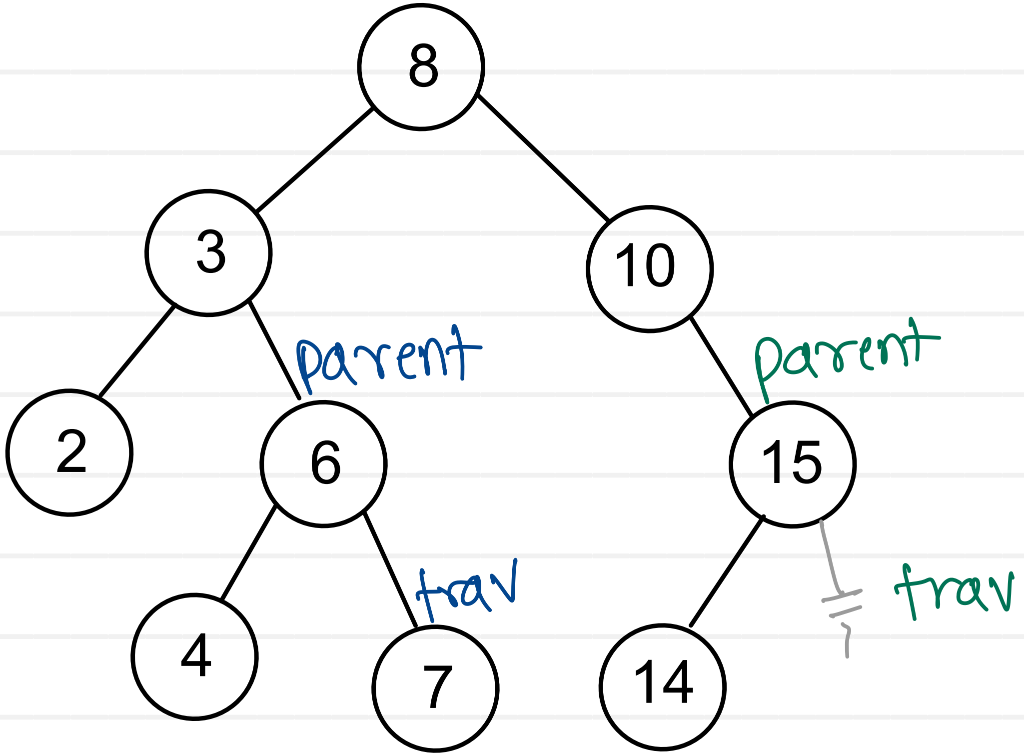
Binary Search Tree - Binary Search



1. Start from root
2. If key is equal to current node data return current node
3. If key is less than current node data search key into left sub tree of current node
4. If key is greater than current node data search key into right sub tree of current node
5. Repeat step 2 to 4 till leaf node

key = 7 - key is found
key = 16 - key is not found

Binary Search Tree - Binary Search with Parent



key = 7

trav	parent
&8	null
&3	&8
&6	&3
&7	&6

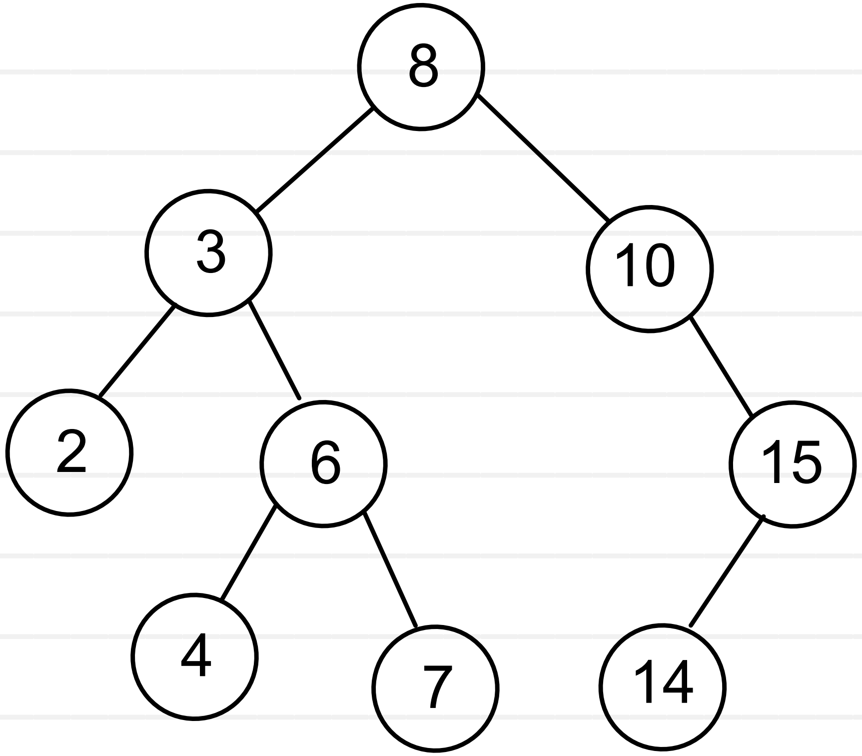
key = 8

trav	parent
&8	null

key = 16

trav	parent
&8	null
&10	&8
&15	&10
null	&15

Binary Search Tree - Delete Node

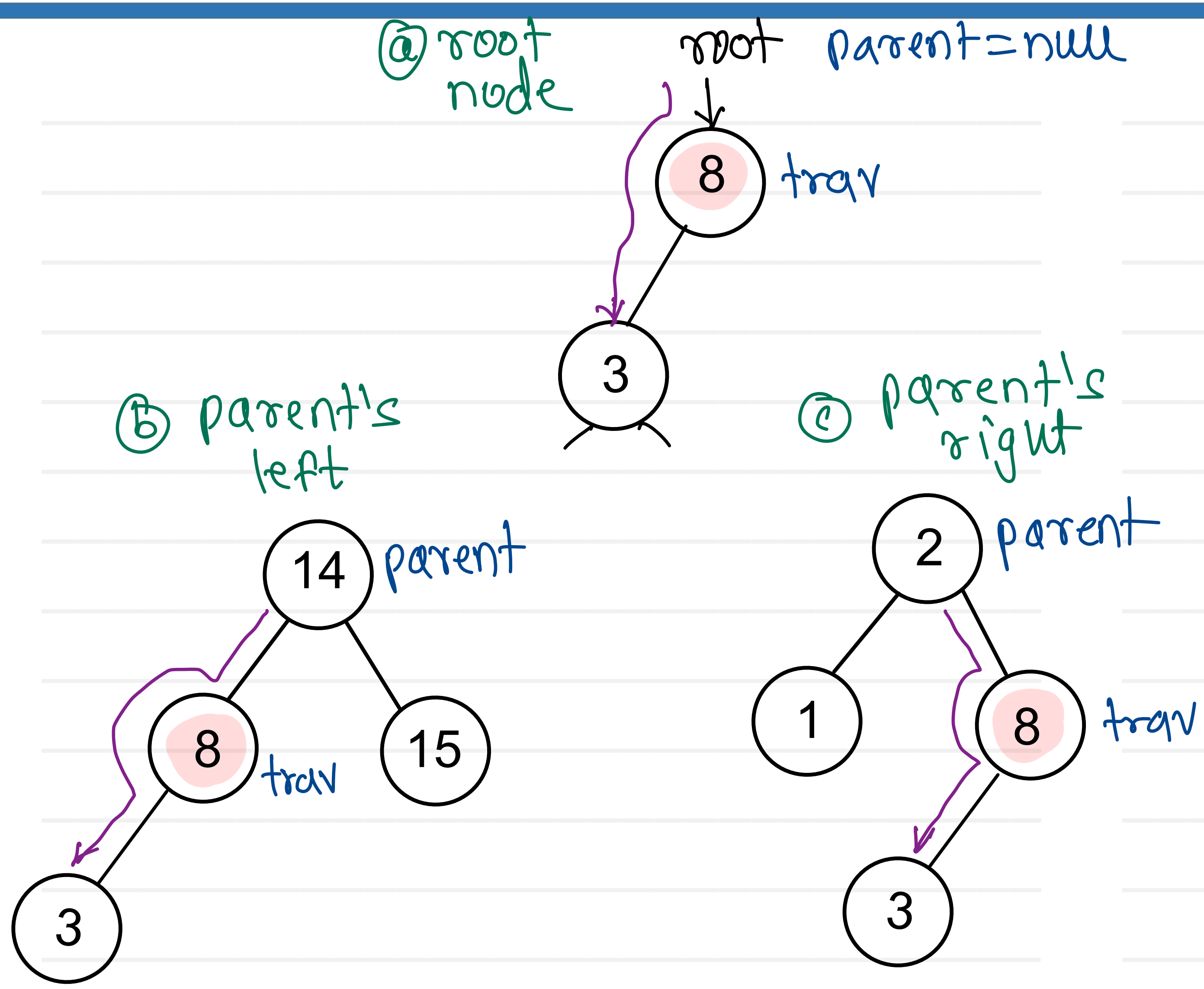


parent's left | root | Parent's right

Node (to be deleted)

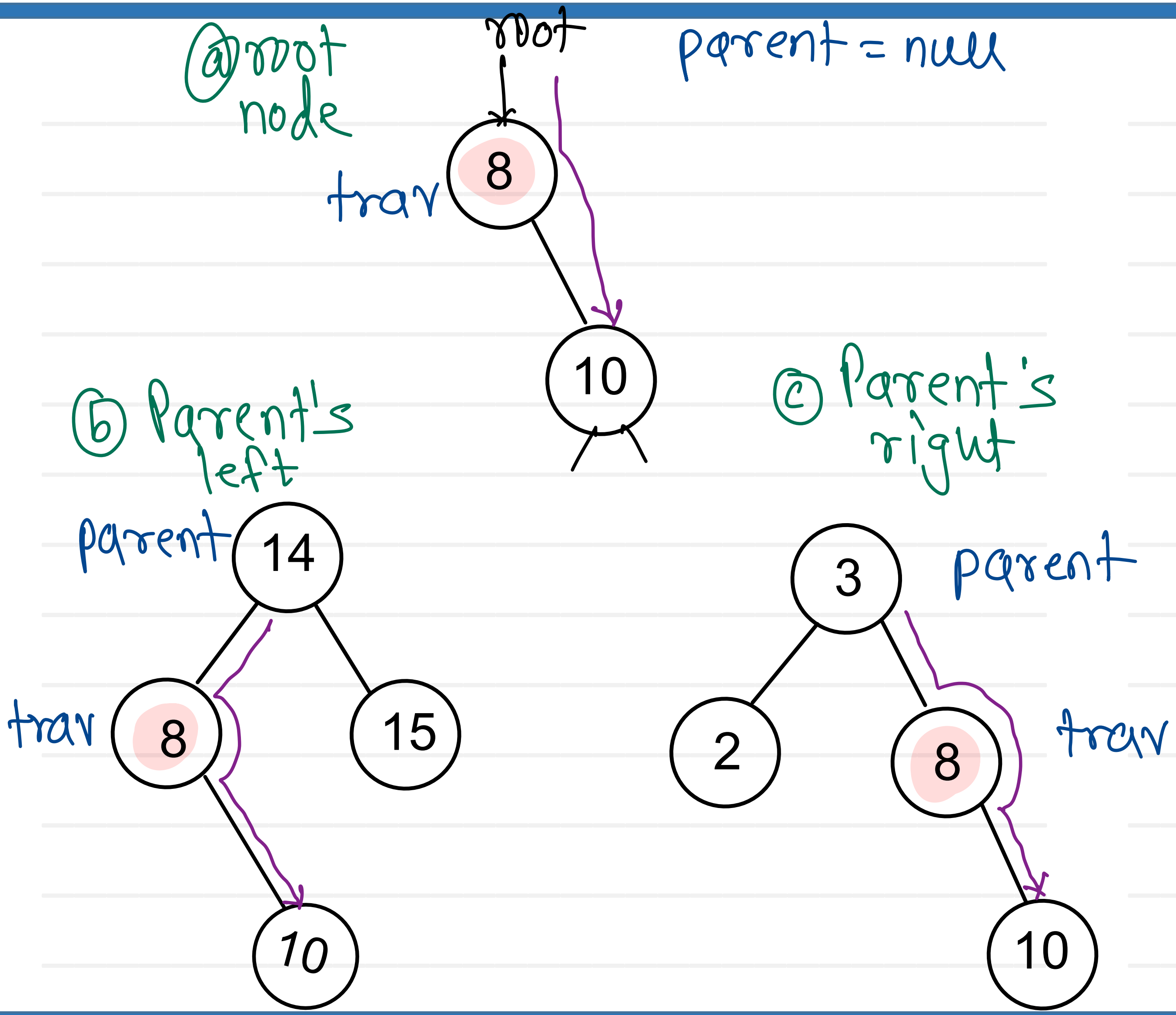
- Non leaf
  - 2 childs
  - 1 child
    - left child
    - right child
- Leaf

BST- Delete Node with Single child node (Left child)



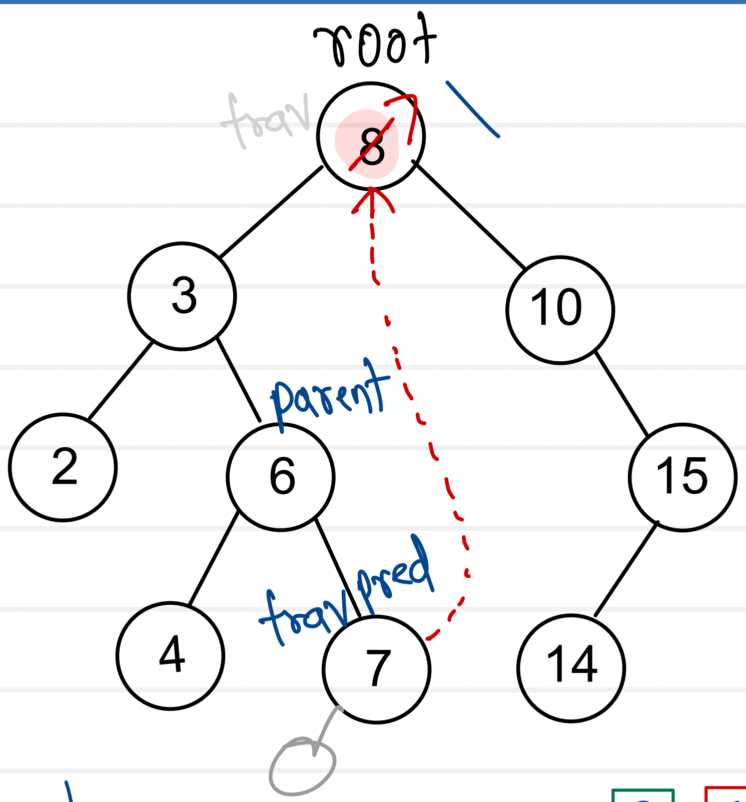

```
if ( trav.left != null){
    if( trav == root)
ⓐ       root = trav.left;
    else if( trav == parent.left)
ⓑ       parent.left = trav.left;
    else if( trav == parent.right)
ⓒ       parent.right = trav.left;
}
```

BST - Delete Node with Single child node (Right child)




```
if ( trav.right != null) {
    if( trav == root)
ⓐ       root = trav.right;
    else if( trav == parent.left)
ⓑ       parent.left = trav.right;
    else if( trav == parent.right)
ⓒ       parent.right = trav.right;
}
```

BST - Delete Node with Two child node



pred	parent
~~3~~	~~8~~
~~6~~	~~3~~
~~7~~	~~6~~

```
if(trav.left != null && trav.right != null) {
    //1. find predecessor of trav
    Node pred = trav.left; parent = trav;
    while(pred.right != null) {
        parent = pred;
        pred = pred.right;
    }
    //2. replace data by predecessor data
    trav.data = pred.data;
    //3. delete predecessor
    trav = pred;
}
```

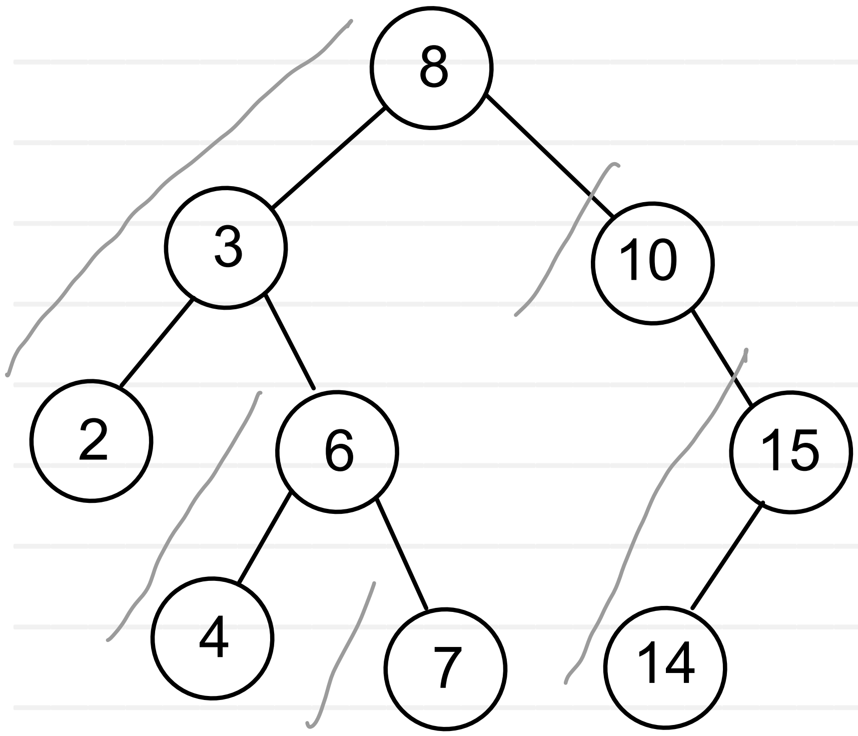
inorder : 2 3 4 6 $\boxed{7}$ $\boxed{8}$ $\boxed{10}$ 14 15

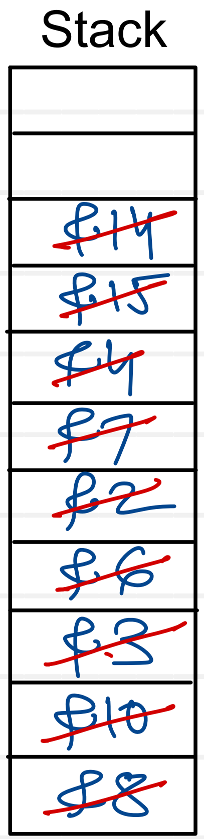
left extreme right ← inorder predecessor

inorder successor → right extreme left

Binary Search Tree - DFS Traversal

(Depth First Search)



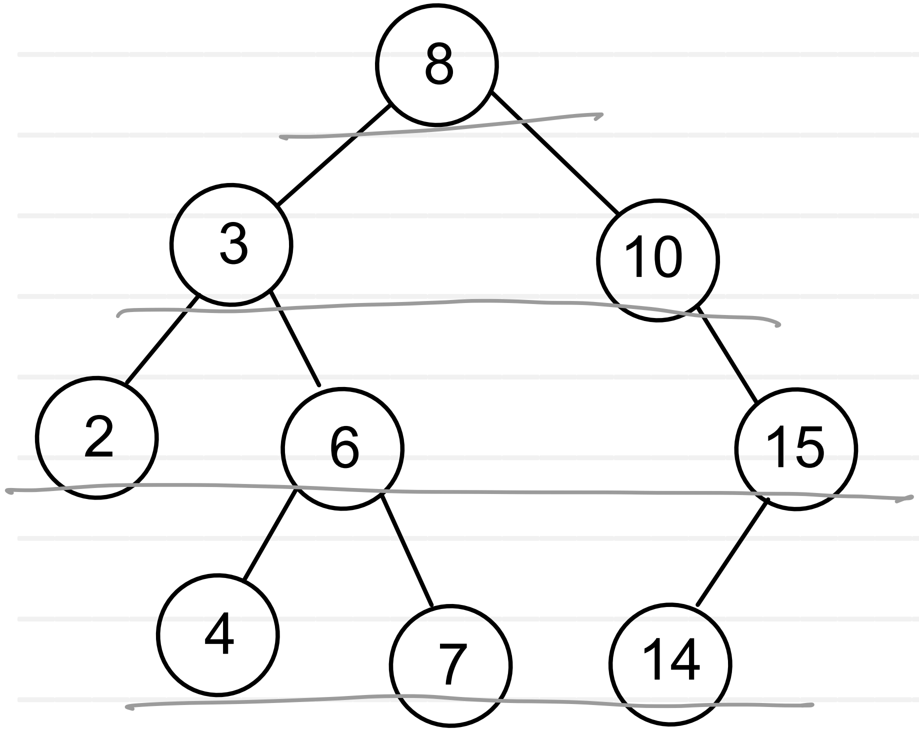


1. Push root node on stack
2. Pop one node from stack
3. Visit (print) popped node
4. If right exists, push it on stack
5. If left exists, push it on stack
6. While stack is not empty, repeat step 2 to 5

8, 3, 2, 6, 4, 7, 10, 15, 14

Binary Search Tree - BFS Traversal

(Bredth First Search)



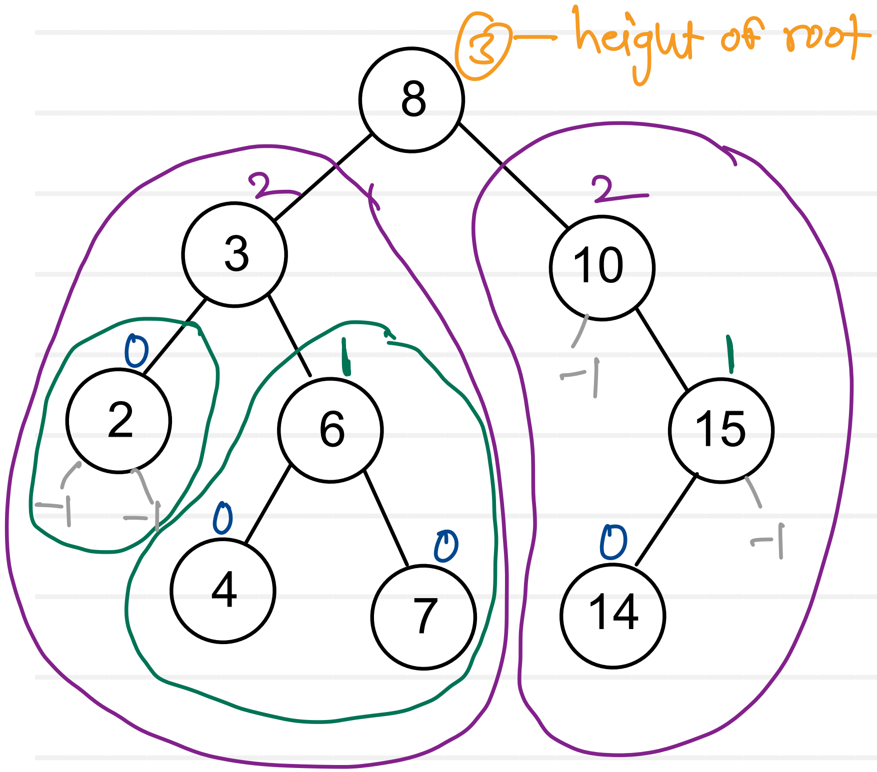
1. Push root node on queue
2. Pop one node from queue
3. Visit (print) popped node
4. If left exists, push it on queue
5. If right exists, push it on queue
6. While queue is not empty, repeat step 2 to 5

Level order traversal

8, 3, 10, 2, 6, 15, 4, 7, 14

Binary Search Tree - Height

Height of root = MAX (height (left sub tree), height (right sub tree)) + 1



1. If left or right sub tree is absent then return -1
2. Find height of left sub tree
3. Find height of right sub tree
4. Find max height
5. Add one to max height and return

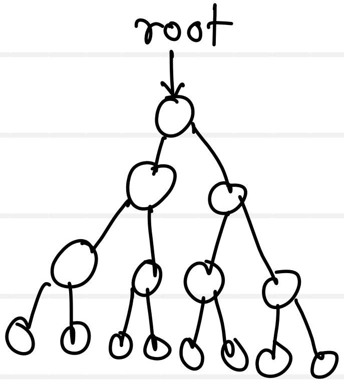
BST - Time complexity of operations

capacity : max number of nodes for given height.

h - height of tree

n - no. of nodes in tree

h	n
-1	0
0	1
1	3
2	7
3	15



$$\boxed{n = 2^{h+1} - 1}$$

$$n \approx 2^h$$

$$\log 2^h = \log n$$

$$h \log 2 = \log n$$

$$h = \frac{\log n}{\log 2}$$

$$\text{Time} \propto \text{height}$$

$$\text{Time} \propto \frac{\log n}{\log 2}$$

Add : $O(h)$ or $O(\log n)$

Delete : $O(h)$ or $O(\log n)$

Search : $O(h)$ or $O(\log n)$

Traverse : $O(n)$

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com