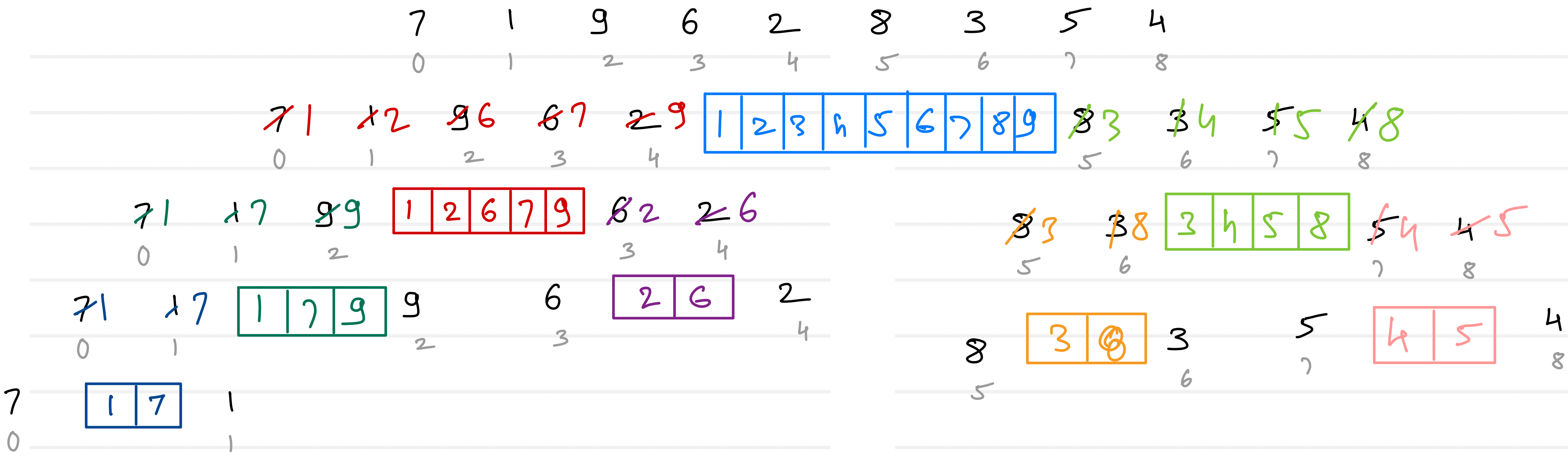
Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com



Quick sort

1. Select pivot/axis/reference element from array
2. Arrange lesser elements on left side of pivot
3. Arrange greater elements on right side of pivot
4. Sort left and right side of pivot again (by quick sort)

11 22 33 44 55
11 22 44 55
22 55

No. of levels = $\log n$
No. of comps/level = n
Total comps = $n \log n$
Time $\propto n \log n$

Best
Avg
$\boxed{T(n) = O(n \log n)}$

11 22 33 44 55
22 33 44 55
33 44 55
44 55
55

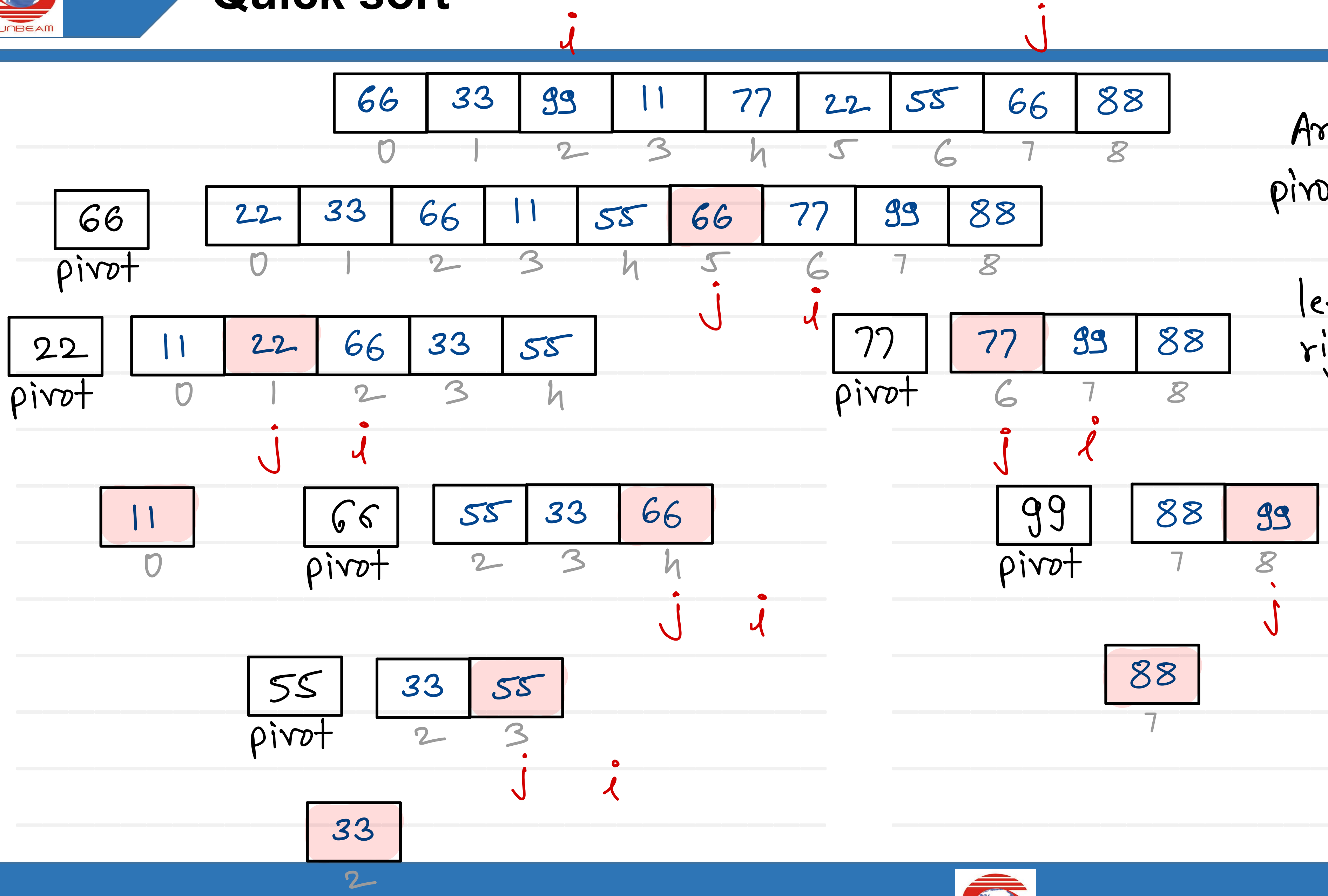
11 22 33 44 55
11 22 33 44
11 22 33
11 22
11

No. of levels = n
No. of comps/level = n
Total comps = n^2
$\boxed{T(n) = O(n^2)}$
worst

$\boxed{S(n) = O(1)}$

- time complexity is dependent on selection of pivot
- pivot is selected by any one of the below method
  1. extreme left
  2. extreme right
  3. mid
  3. median of three
  4. median of five
  5. dual pivot

Quick sort



Array : left → right

pivot is alway placed on j^{th} index

arr[j] = pivot

left partition : left → j-1

right partition : j+1 → right

i = left : i is valid till right

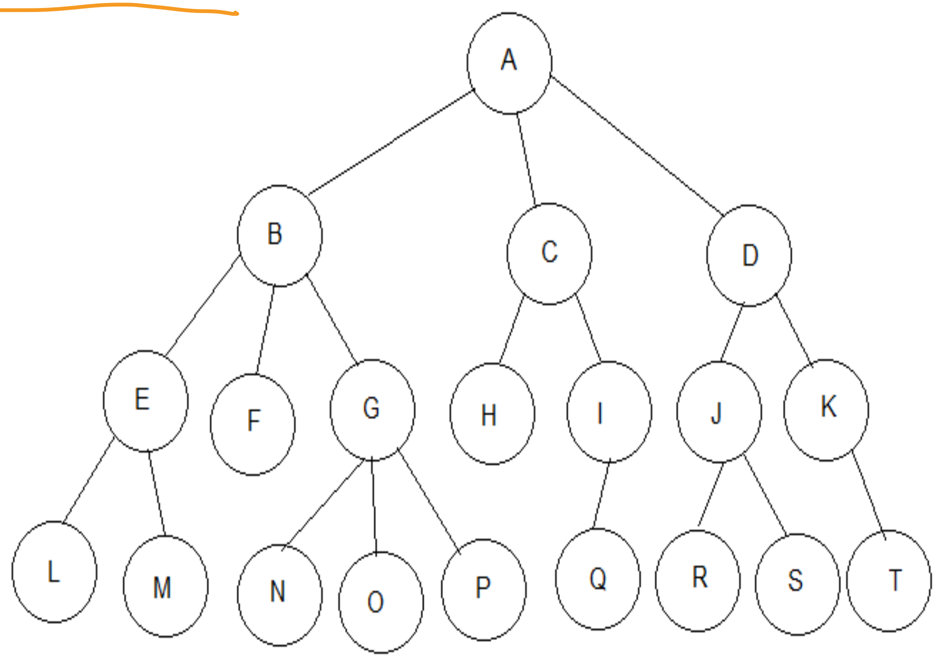
j = right : j is valid till left

	Space	Time Best	Time Avg	Time Worst
Selection sort	$O(1)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubble sort	$O(1)$	$O(n)$	$O(n^2)$	$O(n^2)$
insertion sort	$O(1)$	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	$O(1)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(1)$	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

inplace sorting algorithm

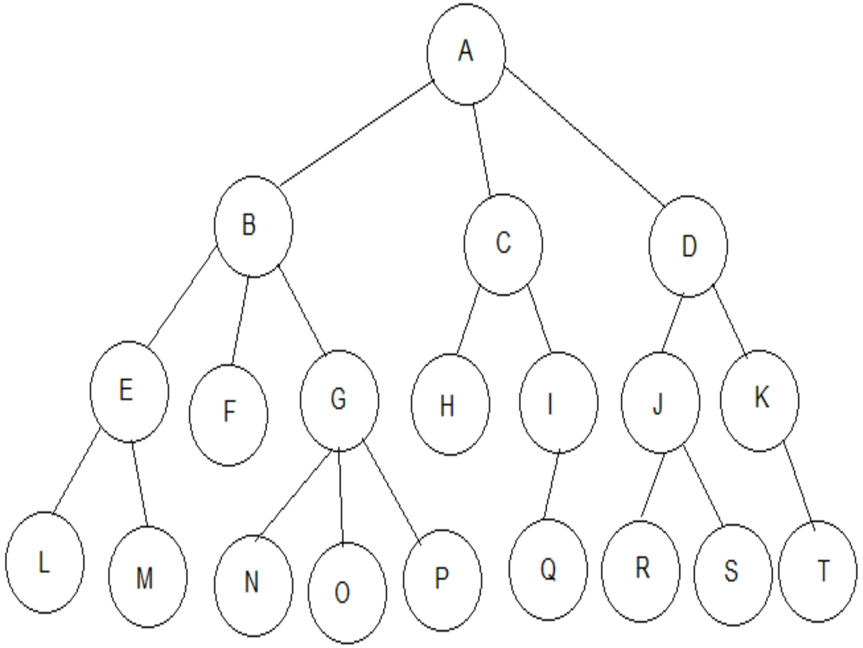
Tree - Terminologies

- **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

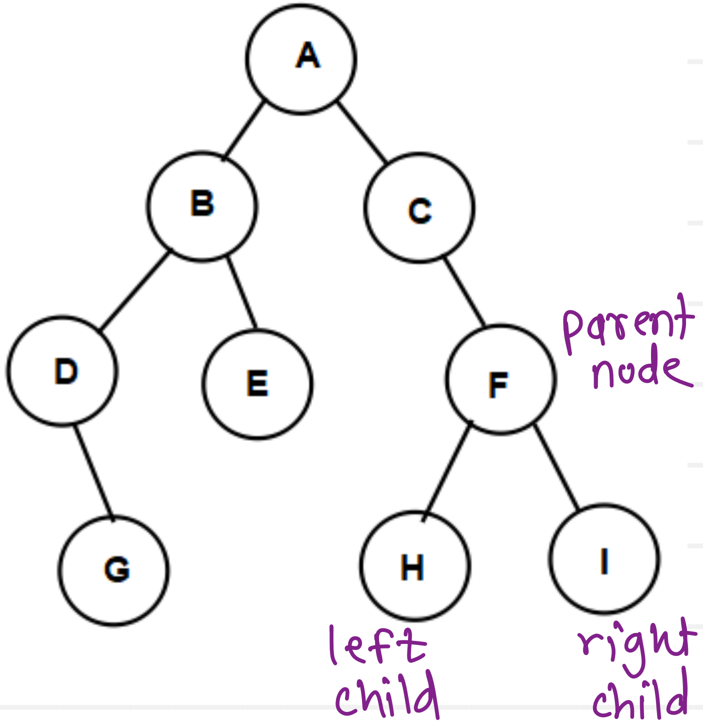
- **Degree of a node :-** number of child nodes for any given node.
- **Degree of a tree :-** Maximum degree of any node in tree.
- **Level of a node :-** indicates position of the node in tree hierarchy
  - Level of child = Level of parent + 1
  - Level of root = 0
- **Height of node :-** number of 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



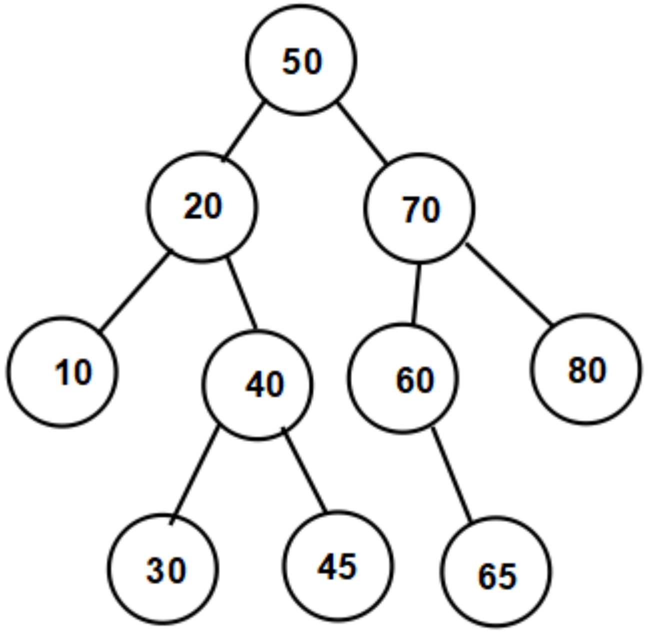
- Tree with zero nodes (ie empty tree) is called as “**Null tree**”. Height of Null tree is -1.
  - Tree can grow up to any level and any node can have any number of Childs.
  - That’s why operations on tree becomes un efficient.
  - Restrictions can be applied on it to achieve efficiency and hence there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

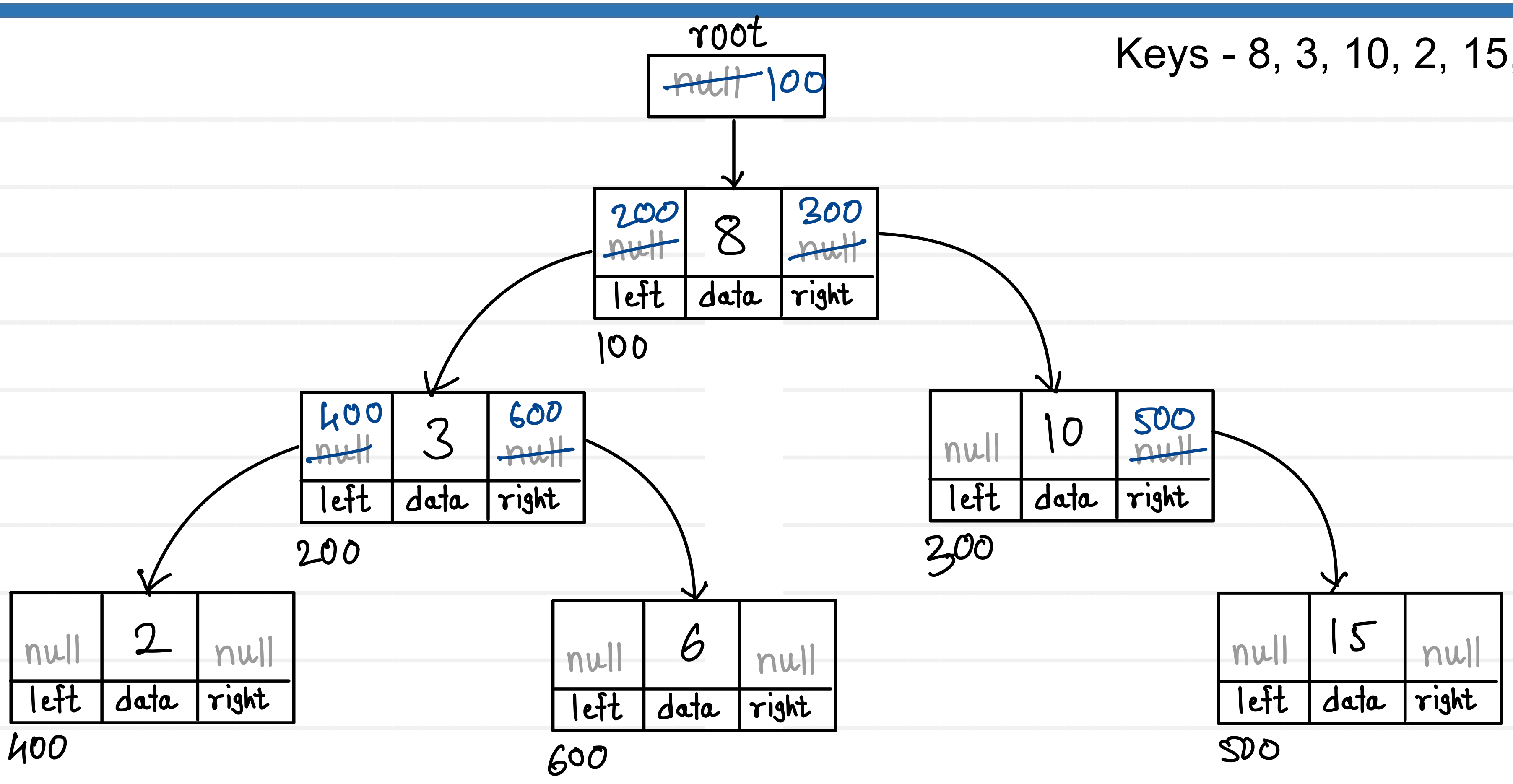
Node :
data - actual data
left - reference of left child
right - reference of right child

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

```
class BST {
    static class Node {
        int data;
        Node left;
        Node right;
    }
    Node root;
    public BST() {...}
    public addNode(value) {...}
    public deleteNode(value) {...}
    public searchNode(key) {...}
    public traverse() {...}
    public deleteAll() {...}
}
```

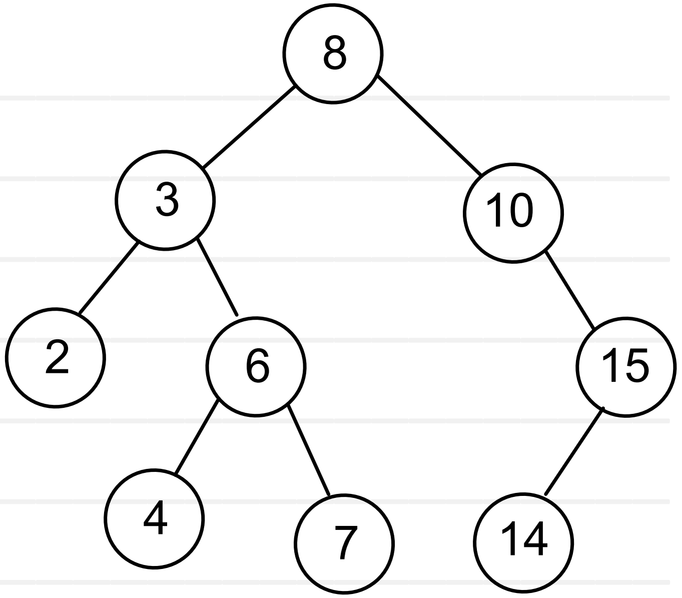
Binary Search Tree - Add Node

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



Binary Search Tree – Add Node

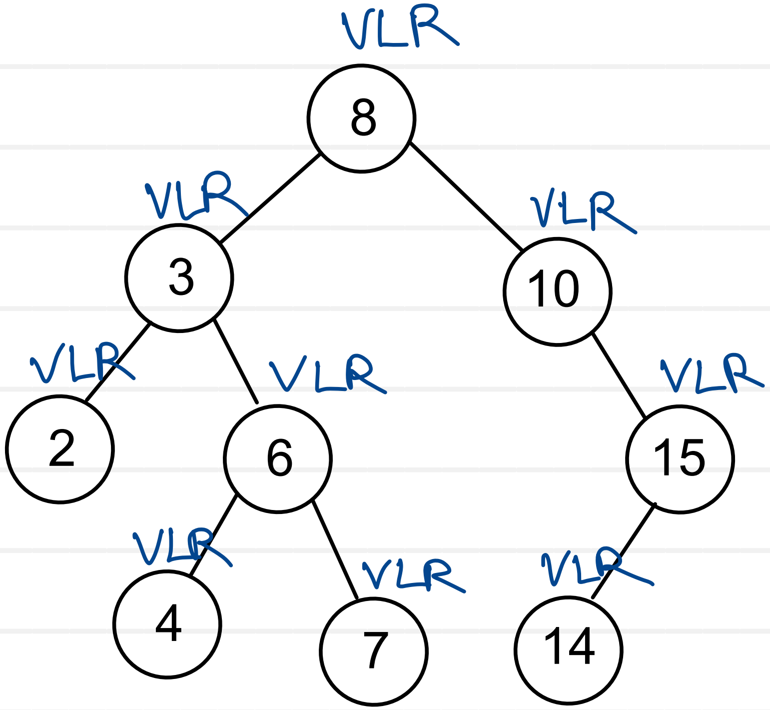
//1. create node for given value
//2. if BSTree is empty
    // add newnode into root itself
//3. if BSTree is not empty
    //3.1 create trav reference and start at root node
    //3.2 if value is less than current node data (trav.data)
        //3.2.1 if left of current node is empty
            // 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.

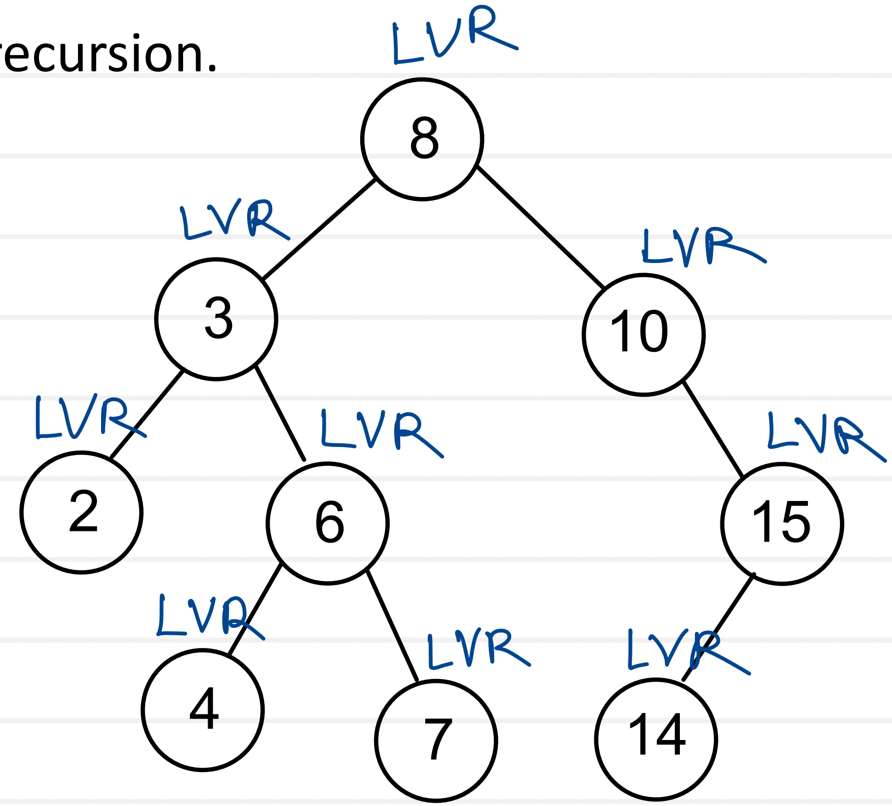


- **Pre-Order** :- 8, 3, 2, 6, 4, 7, 10, 15, 14

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.

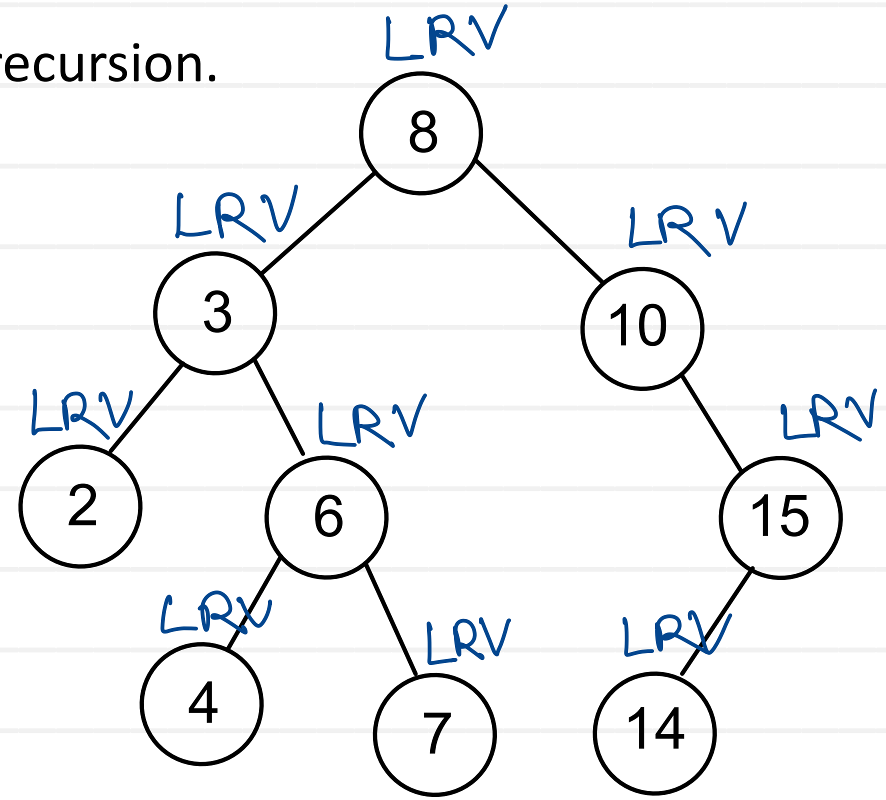


- **In-Order :-** 2 , 3 , 4 , 6 , 7 , 8 , 10 , 14 , 15

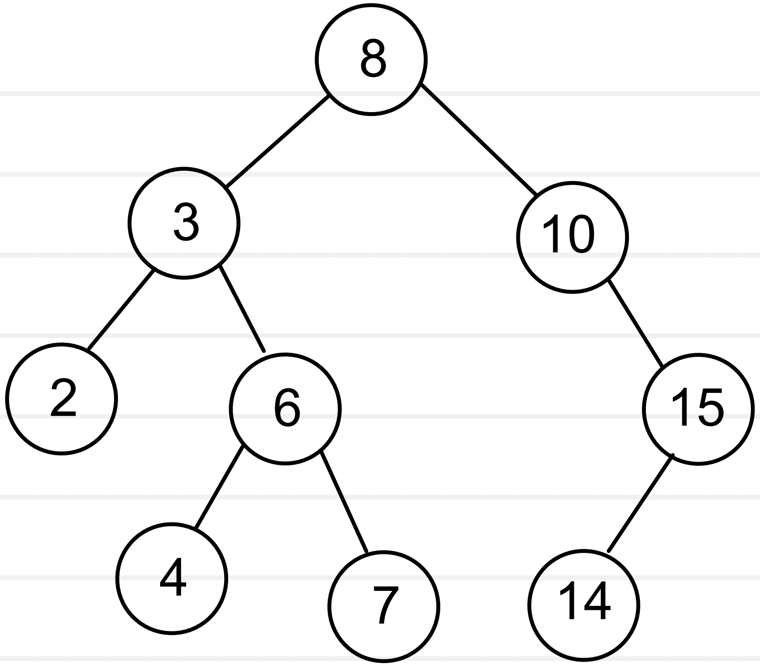
Tree Traversal Techniques

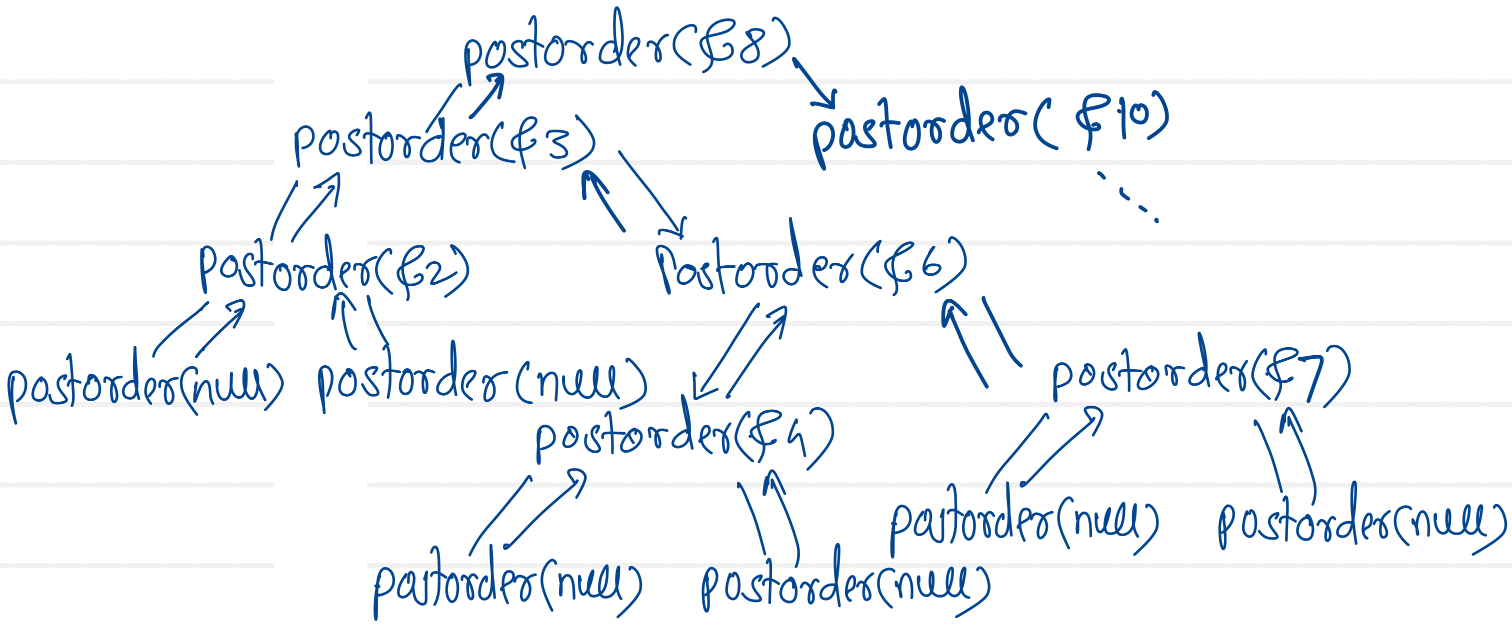
- **Pre-Order:-** V L R
- **In-order:-** L V R
- **Post-Order:-** L R V
- The traversal algorithms can be implemented easily using recursion.
- Non-recursive algorithms for implementing traversal

needs stack to store node pointers.



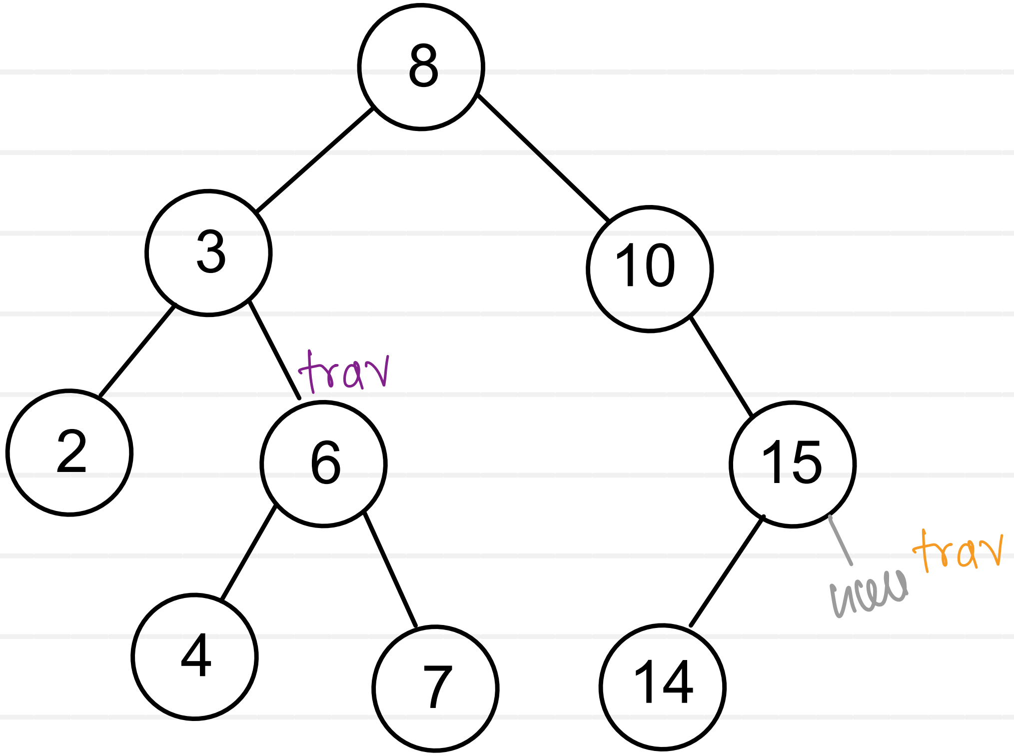
- **Post-Order** :- 2, 4, 7, 6, 3, 14, 15, 10, 8






```
postorder( trav) {
    if( trav == null)
        return;
postorder( trav. left);
postorder( trav. right);
 sysout ( trav. data);
}
```

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

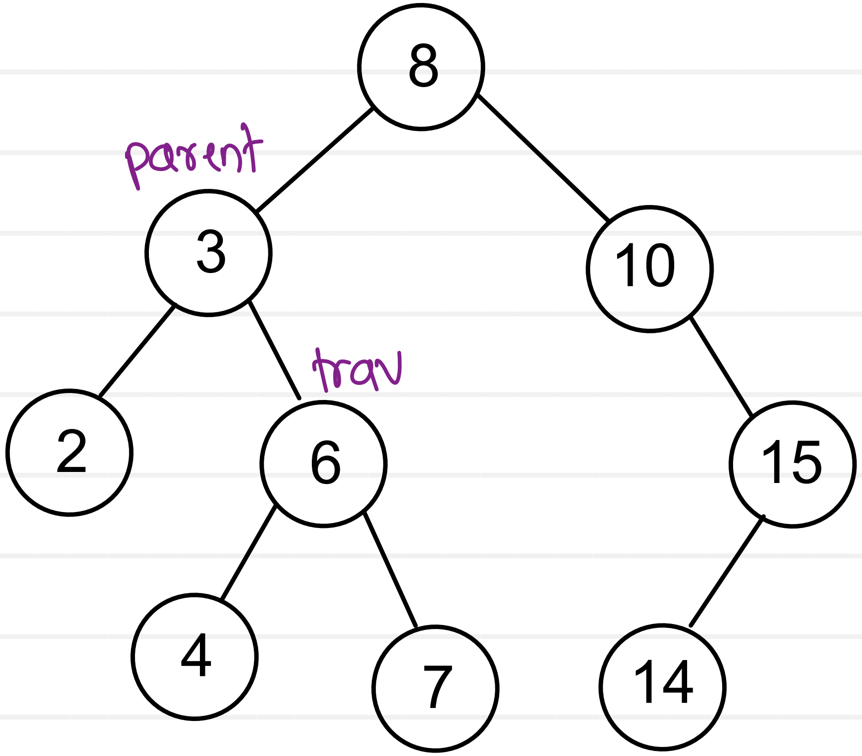
Key = 6

trav	key
&8	<
&3	>
&6	=

Key = 18

trav	key
&8	>
&10	>
&15	>
null	

Binary Search Tree - Binary Search with Parent



key = 8

trav	parent	key
&8	null	==

key = 6

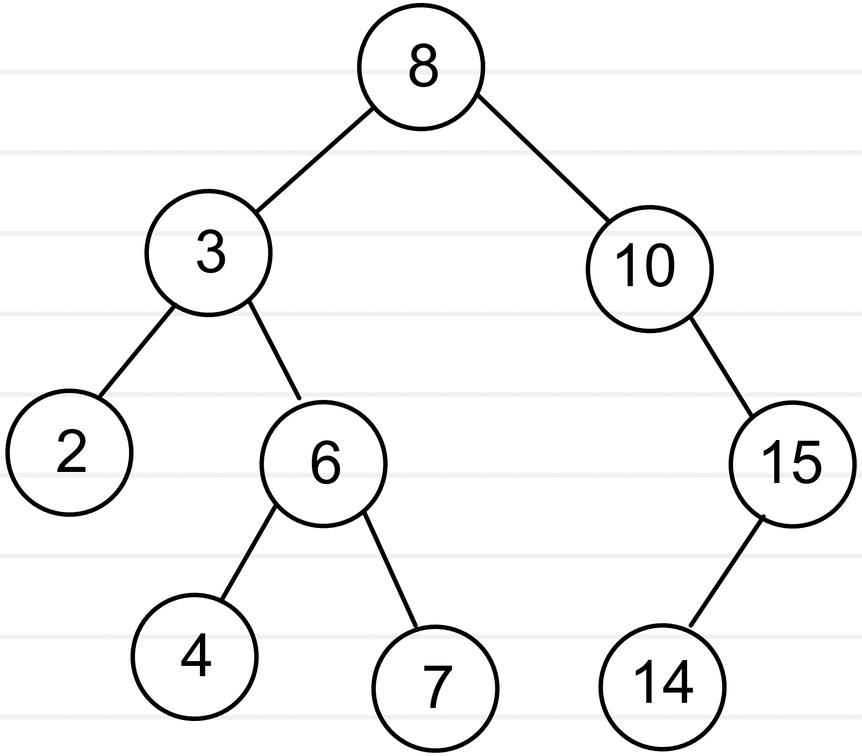
trav	parent	key
&8	null	<
&3	&8	>
&6	&3	==

key = 18

trav	parent	key
&8	null	>
&10	&8	>
&15	&10	>
null	&15	

key not found

Binary Search Tree - Delete Node



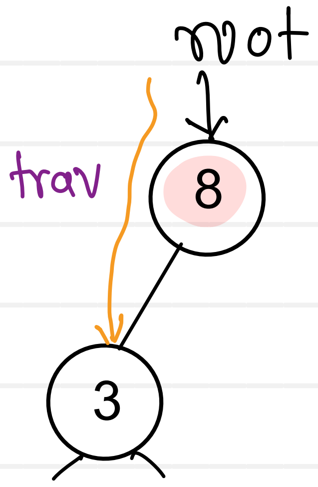
- parent's left child
- root node
- parent's right child

Node (to be deleted)

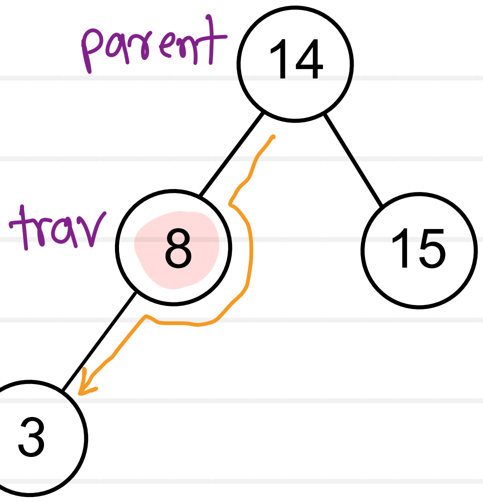
- leaf node
- non leaf node
  - single child
    - left child
    - right child
  - 2 child

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

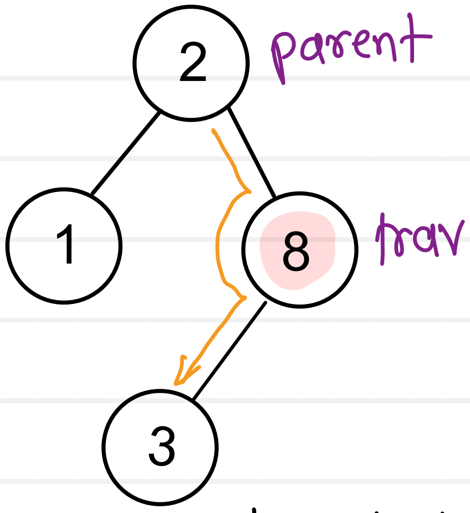
(a) root node



(b) Parent's left



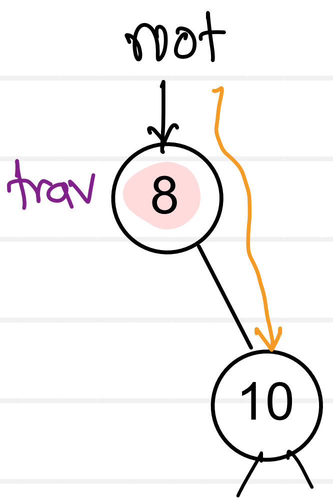
(c) Parent's right



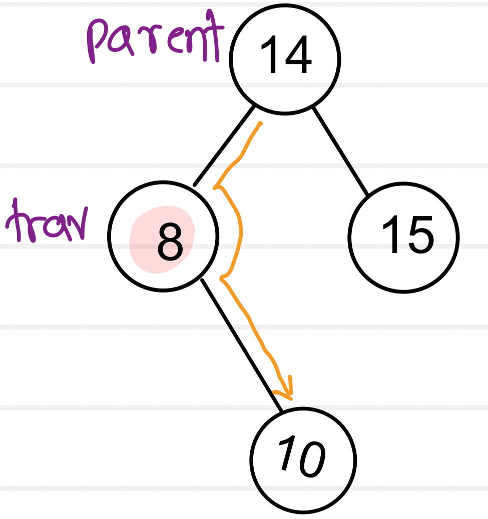
1. check if has only left child
2. if it is root node
   then update root by its left
3. if it is parent's left child
   then update parent's left by left child
4. if it is parent's right child
   then update parent's righ by left child

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

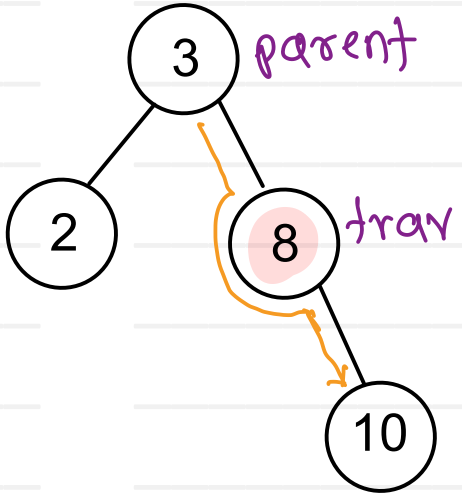
(a) root node



(b) Parent's left



(c) Parent's right



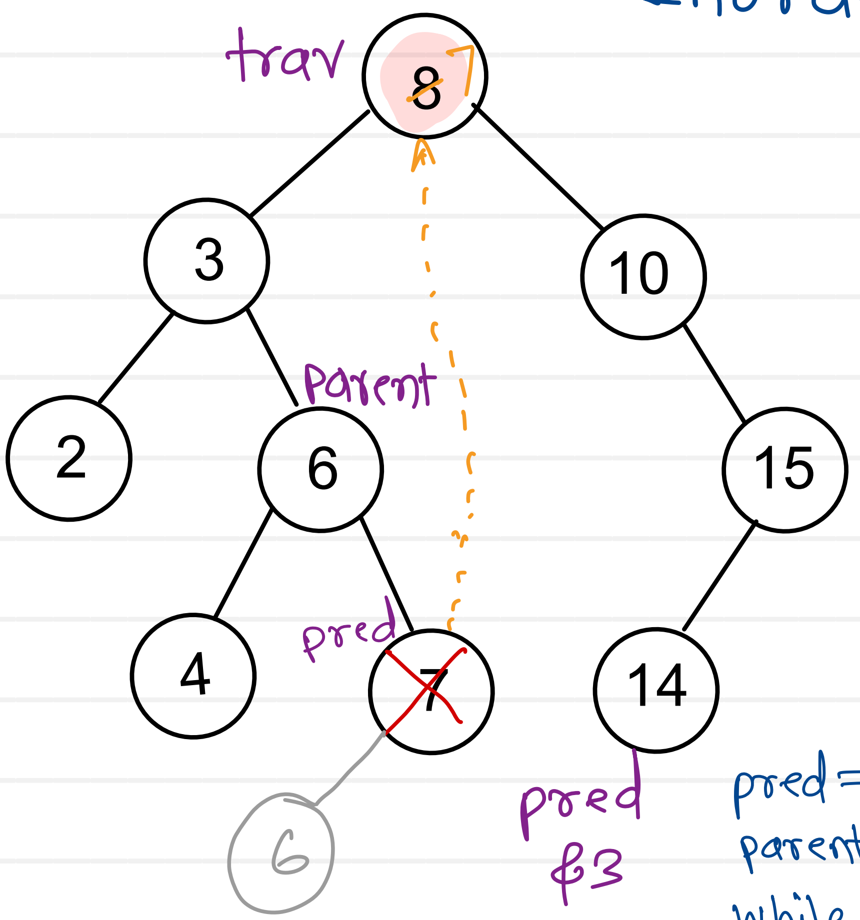
1. check if has only right child
2. if it is root node
   then update root by its right child
3. if it is parent's left child
   then update parent's left by right child
4. if it is parent's right child
   then update parent's right by right child

BST - Delete Node with Two child node

Inorder : 2 3 4 6 7 8 10 14 15

extreme right of left sub tree ← inorder predecessor (7)

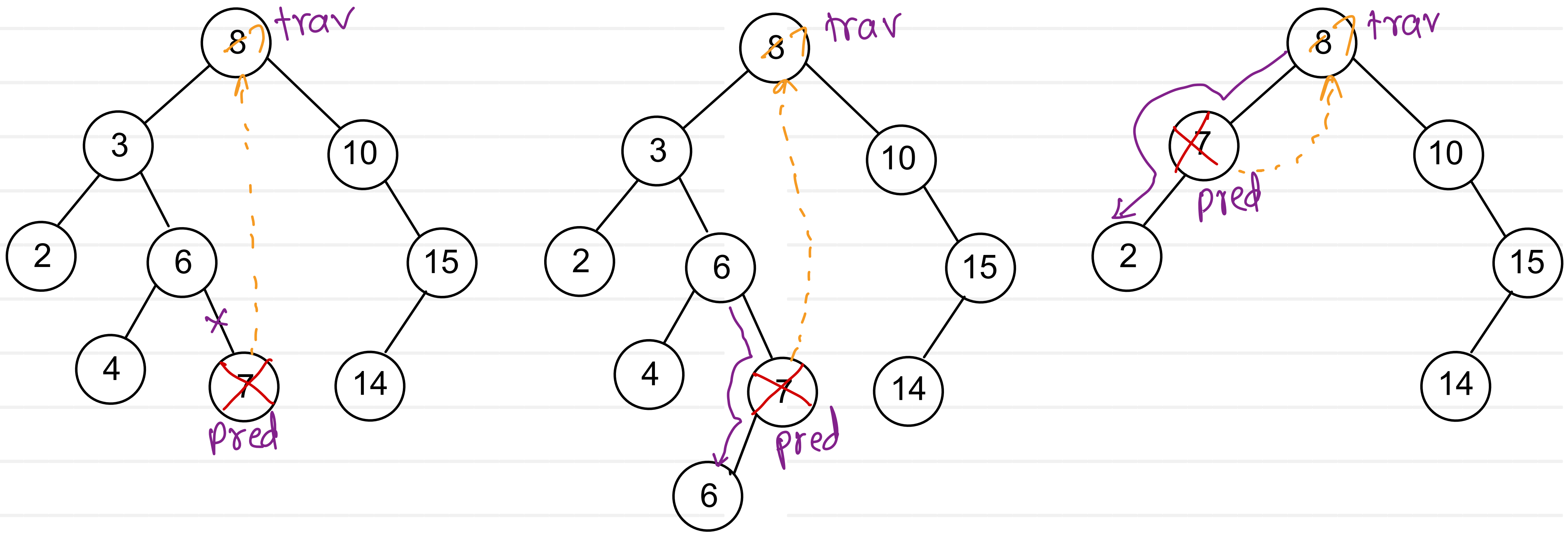
inorder successor (10) → extreme left of right sub tree



1. check if has both childs
2. find predecessor of node
3. replace value by predecessor value
4. delete predecessor

pred
3
6
7

```
pred = trav.left;
parent = trav;
while(pred.right != null) {
    parent = pred;
    pred = pred.right;
}
```



8
trav
3
10
2
6
15
4
7
14
pred
8
trav
3
10
2
6
15
4
7
14
pred
6
8
trav
7
10
pred
2
15
14

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com