Binary Search Tree

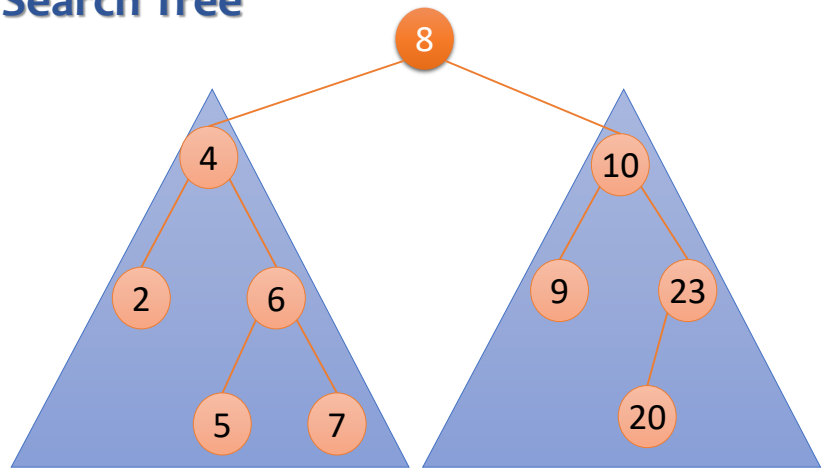
47

Binary Search Tree

- A binary search tree is a binary tree in which each node *n* has properties:
  - *n* 's value greater than all values in left subtree T_L
  - *n* 's value less than all values in right subtree T_R
  - Both T_R and T_L are binary search trees.

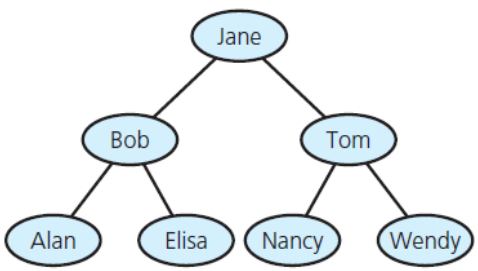
48

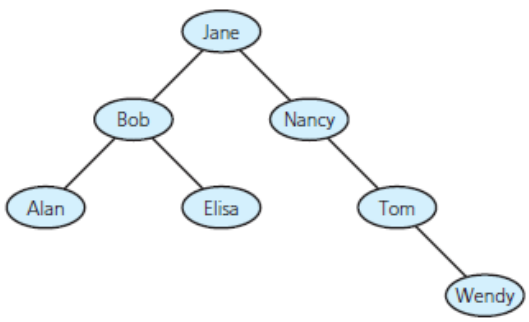
Binary Search Tree



49

Binary Search Tree





50

Binary Search Tree | Operations

- Insert (a key)
- Search (a key)
- Remove (a key)
- Traverse
- Sort (based on key value)
- Rotate (Left rotation, Right rotation)

51

Binary Search Tree | Insert

```
Insert (root, Data)
{
   if (root is NULL) {
         Create a new_Node containing Data
         This new_Node is root of the tree
   }
   //Compare root's key with Key
   if root's Key is less than Data's Key
         Insert Key to the root's RIGHT subtree
   else if root's Key is greater than Data's Key
         Insert Key to the root's LEFT subtree
   else
         Do nothing //Explain why?
}
```


52

Binary Search Tree | Insert

- Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?

15, 5, 12, 8, 23, 1, 17, 21

53

Binary Search Tree | Insert

- Beginning with an empty binary search tree, what binary search tree is formed when you insert the following values in the order given?
  - W, T, N, J, E, B, A
  - W, T, N, A, B, E, J
  - A, B, W, J, N, T, E
  - B, T, E, A, N, W, J

54

4.0 Binary Search Tree | Search

```
Search (root, Data)
{
    if (root is NULL) {
          return NOT_FOUND;
    }
    //Compare root's key with Key
    if root's Key is less than Data's Key
          Search Data in the root's RIGHT subtree
    else if root's Key is greater than Data's Key
          Search Data in the root's LEFT subtree
    else
          return FOUND //Explain why?
}
```


55

4.0 Binary Search Tree | Remove

- When we delete a node, three possibilities arise.
- Node to be deleted:
  - **is leaf:**
    - Simply remove from the tree.
  - has **only one child:**
    - Copy the child to the node and delete the child
  - has **two children:**
    - Find in-order successor (predecessor) ***S_Node*** of the node.
    - Copy contents of ***S_Node*** to the node and delete the ***S_Node***.

56

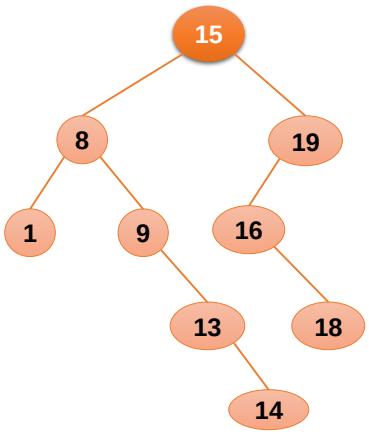
Binary Search Tree | Traversal

- Pre-Order:
  - Node - Left - Right
- In-Order:
  - Left - Node - Right
- Post-Order:
  - Left - Right - Node

57

Binary Search Tree | Traversal

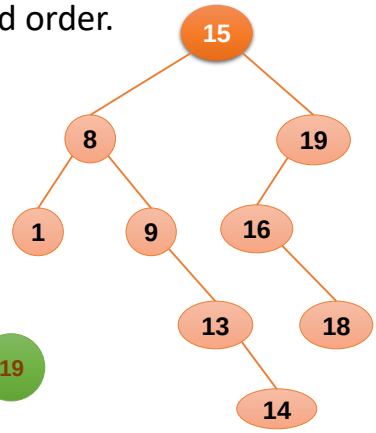
- What are the pre-order, in-order and post-order traversals of this binary search tree?



58

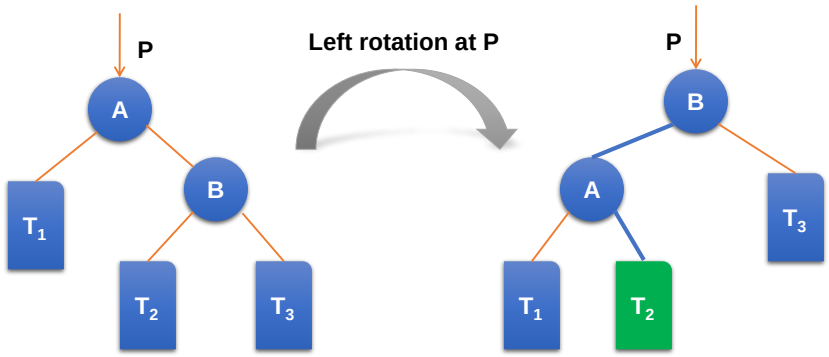
Binary Search Tree | Sort

- In-order traversal gives the key values in sorted order.



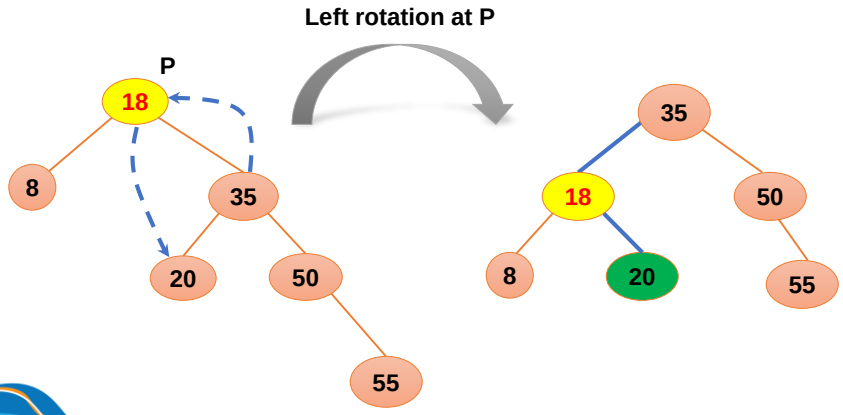
59

Binary Search Tree | Left Rotation



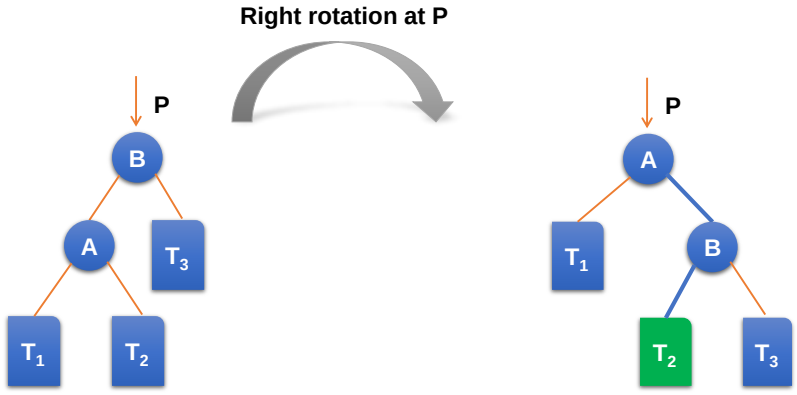
60

Binary Search Tree | Left Rotation



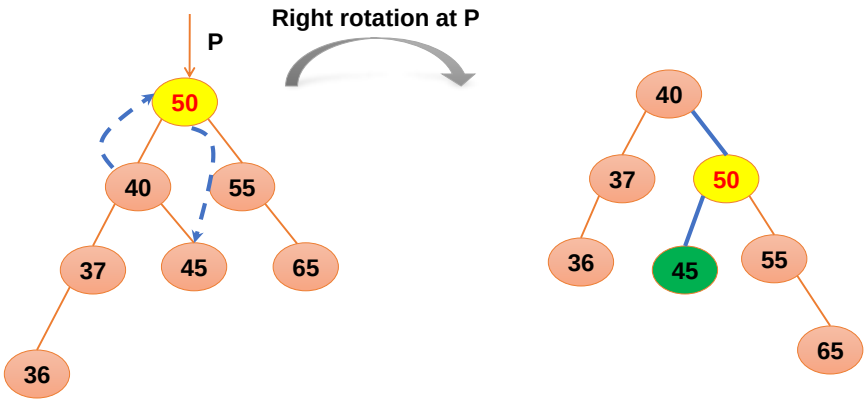
61

Binary Search Tree | Right Rotation



62

Binary Search Tree | Right Rotation



63

Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	O(log *n*)	O(*n*)
Insertion	O(log *n*)	O(*n*)
Removal	O(log *n*)	O(*n*)
Traversal	O(*n*)	O(*n*)

64

Very Bad Binary Search Tree

- Beginning with an empty binary search tree, what binary search tree is formed when inserting the following values in the order given?

2, 4, 6, 8, 10, 12, 14, 18, 20

65

AVL Tree

66

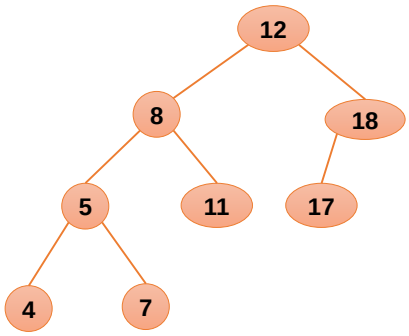
4.0 AVL Tree

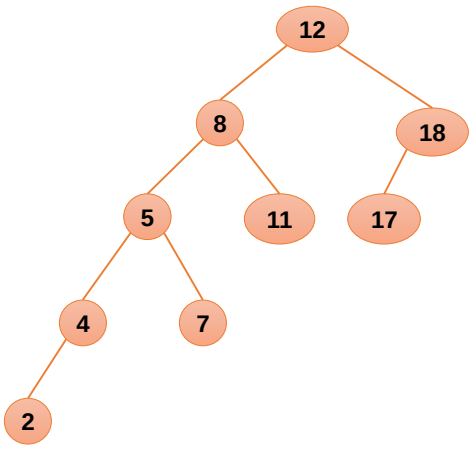
- Named for inventors, **A**del'son-**V**el'skii and **L**andis (1962).
- AVL Tree is a **self-balancing** binary search tree where
  - for ALL nodes, the difference between height of the left subtrees and the right subtrees **cannot be more than one**.

67

4.0 AVL Tree

- Which is the AVL Tree?





68

4.0 AVL Tree

- A balanced binary search tree
  - Maintains height close to the minimum
  - After insertion or deletion, check the tree is still AVL tree – determine whether any node in tree has left and right subtrees whose heights differ by more than 1
- Can search AVL tree almost as efficiently as minimum-height binary search tree.

69

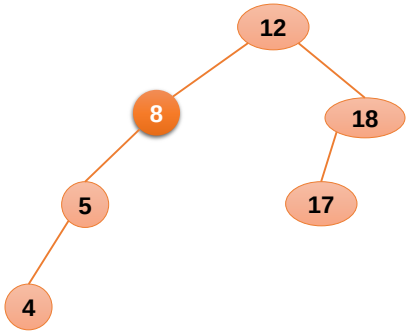
4.0 Cases of Un-Balanced Nodes

- Left-Left case
- Left-Right case
- Right-Right case
- Right-Left case

70

Cases of Un-Balanced Nodes

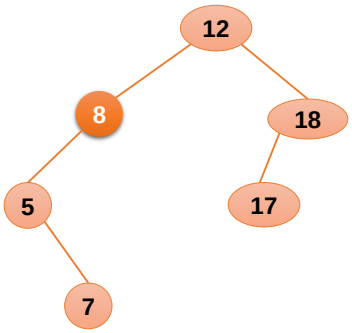
- Left-Left case



71

Cases of Un-Balanced Nodes

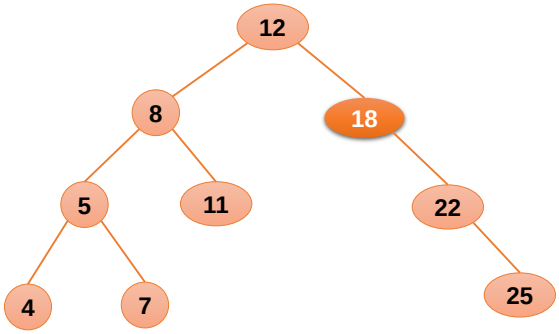
- Left-Right case



72

Cases of Un-Balanced Nodes

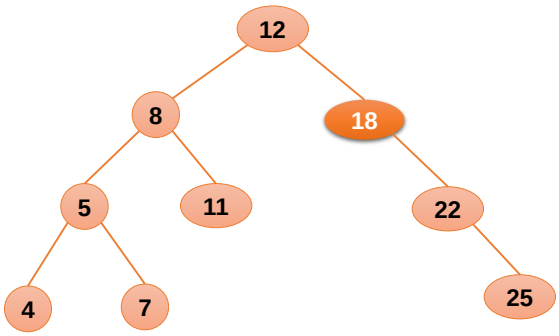
- Right-Right case



73

Cases of Un-Balanced Nodes

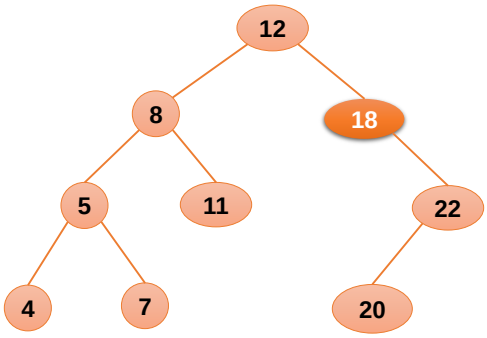
- Right-Right case



74

4.0 Cases of Un-Balanced Nodes

- Right-Left case



75

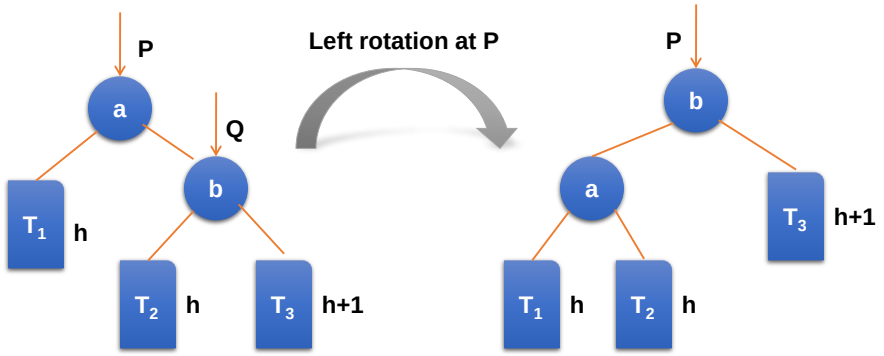
4.0 Un-balanced Resolving

- Right-Right case:
  - Left rotation at un-balanced node.
- Right-Left case:
  - Right rotation at un-balanced node's right child.
  - Left rotation at un-balanced node.

76

4.0 Un-balanced Resolving

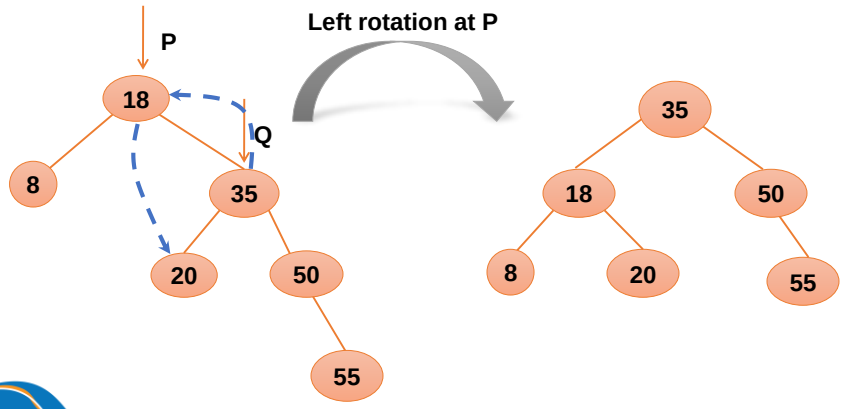
- Right-Right case:



77

4.0 Un-balanced Resolving

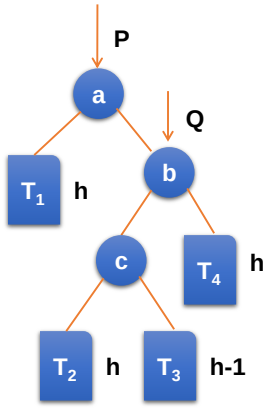
- Right-Right case:



78

4.0 Un-balanced Resolving

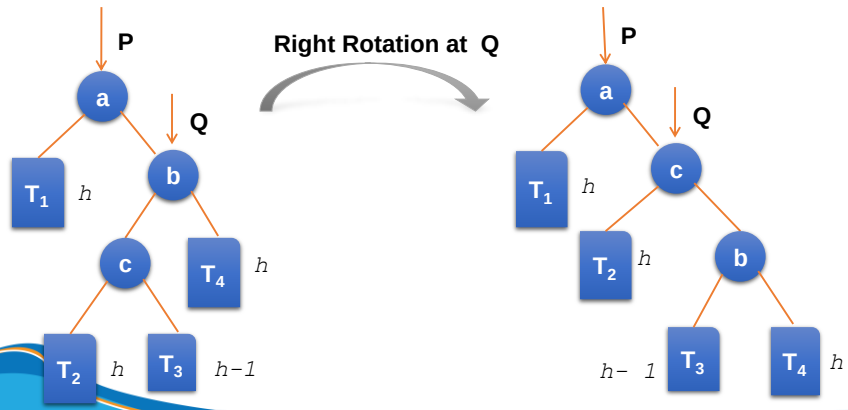
- Right-Left case:
  - Right rotation at Q
  - Left rotation at P



79

4.0 Un-balanced Resolving

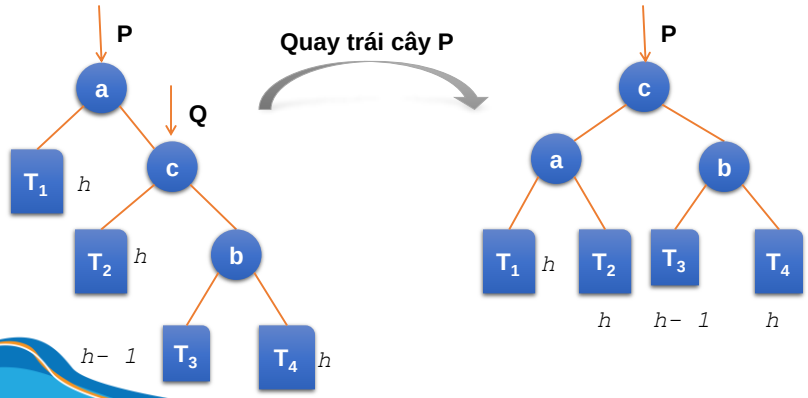
- Right-Left case:
  - Right rotation at Q



80

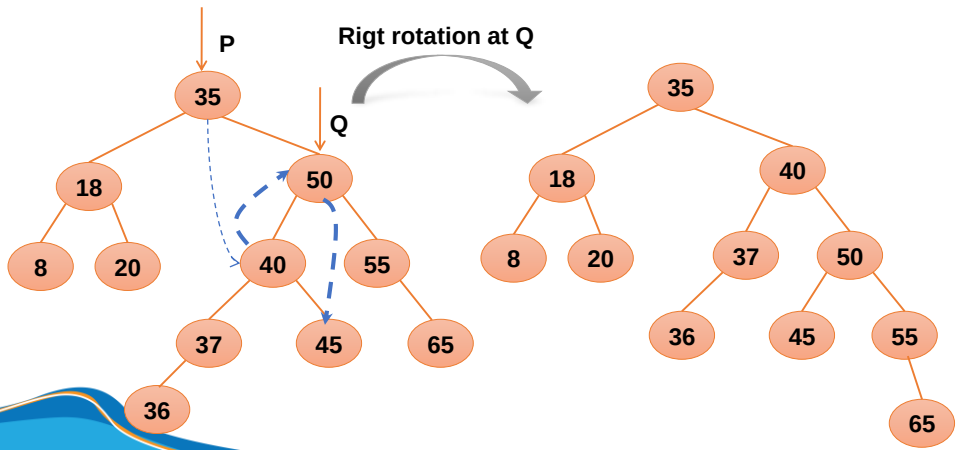
4.0 Un-balanced Resolving

- Right-Left case:
  - Right rotation at P



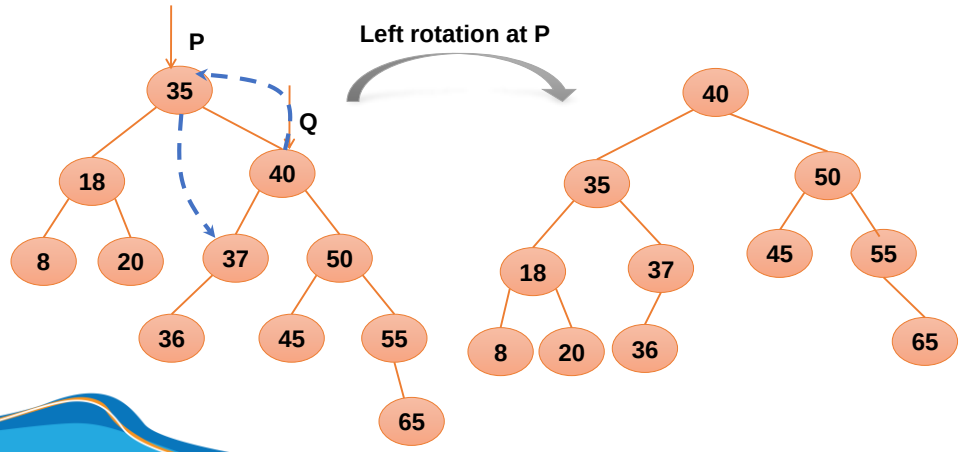
81

4.0 Un-balanced Resolving



82

4.0 Un-balanced Resolving



83

4.0 Un-balanced Resolving

- Left-Left case:
  - Right rotation at un-balanced node.
- Left-Right case:
  - Left rotation at un-balanced node's left child.
  - Right rotation at un-balanced node.

84

4.0 Un-balanced Resolving

Source: Wikipedia