**EECS560 Data Structures Kong**

**Topic 7:** Advanced Search Trees

**Read:** Chpt. 4 and Chpt. 12, Weiss

Recall that in implementing the ADT: Search tree based on BST, we have

**"Good" characteristics of BST:**

- Simplicity.
- Support general search/delete as well as special searchMin(Max)/deleteMin(Max) operations.
- Can easily be sorted (inorder traversal).
- Can easily be stored (preorder traversal).
- Good average performance, $T_a(n) = O(\lg n)$.

**"Bad" characteristics of BST:**

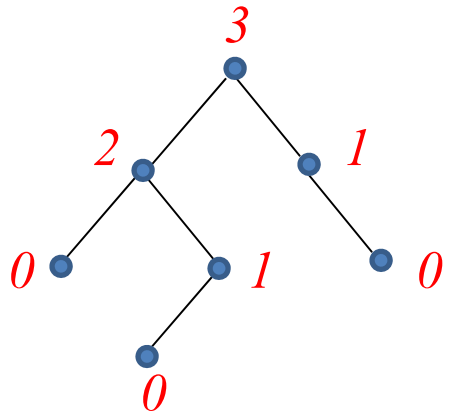
- Inefficient when many items are having identical keys since height may increase.
- Worst-case complexity depends on height of tree. Hence, $T_w(n) = O(h) = O(n)$.

**Q:** Can we design a balanced BST structure such that $T_w(n) = T_a(n) = O(\lg n)$?

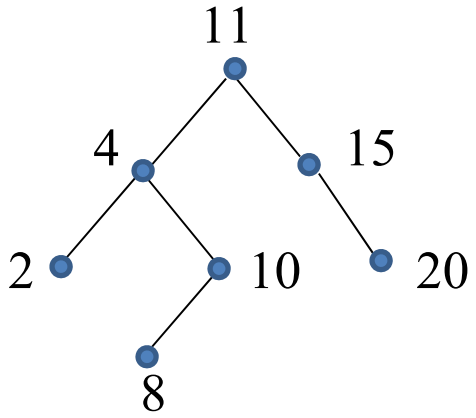
**Balanced Binary Tree:**

A balanced binary tree T is a binary tree such that for any node x in the tree, the height of its left subtree differs by no more than one from the height of its right subtree. Hence, $|h(T_L(x)) - h(T_R(x))| \leq 1$, where $h(T_L(x))$, and $h(T_L(x))$, are the height of the left, and right, subtree rooted at x, respectively. (The height of an empty tree is defined to be -1.)

**Example:** A balanced binary tree showing height of nodes.




**Example:** A balanced BST.



## I. AVL ( Adelson-Velski & Landis) Trees

An AVL tree is a heighted balanced BST using self-balancing operations during dynamic operations. Hence, for each node x in an AVL tree,

$$|h(T_L(x)) - h(T_R(x))| \leq 1.$$

**Dfe:** The ***balanced factor*** of x in an AVL tree is defined as $BF(x) = h(T_L(x)) - h(T_R(x))$.

For each node x in an AVL tree, BF(x) = 0, +1, or -1.
And if BF(x) = 0, x is balanced,
= +1, x is left-heavy,
= -1, x is right-heavy.

**Theorem:** The height of an AVL tree T with n nodes is O(lgn).

**AVL Tree Operations:**

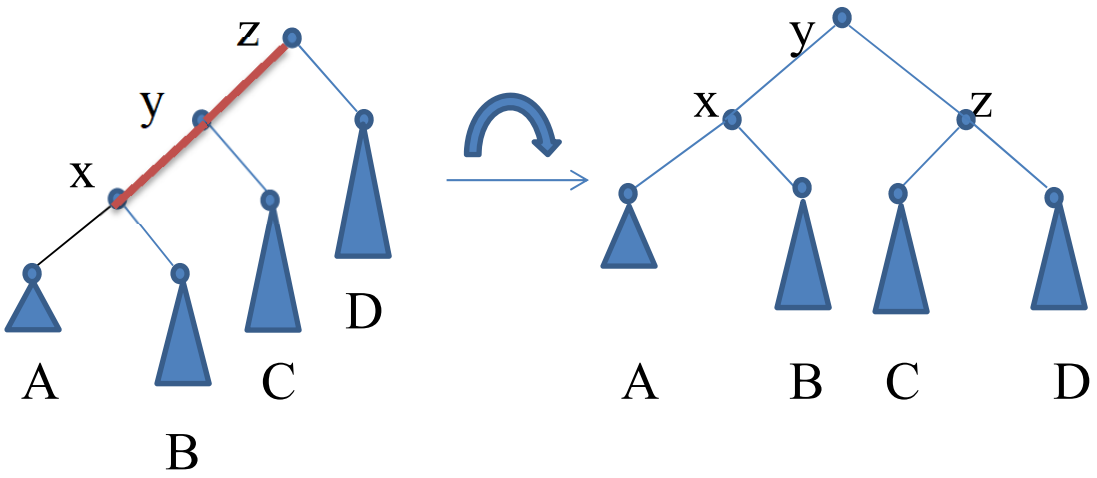
1. ***Static operations***: Find, findMin, findMax.
   Perform operations as in regular BST.

2. ***Dynamic operations***: Insert, Delete.
   Step 1: Perform operations as in regular BST.
   Step 2: If resulting BST tree becomes unbalanced, perform rotation operation(s) to get back a balanced AVL tree.

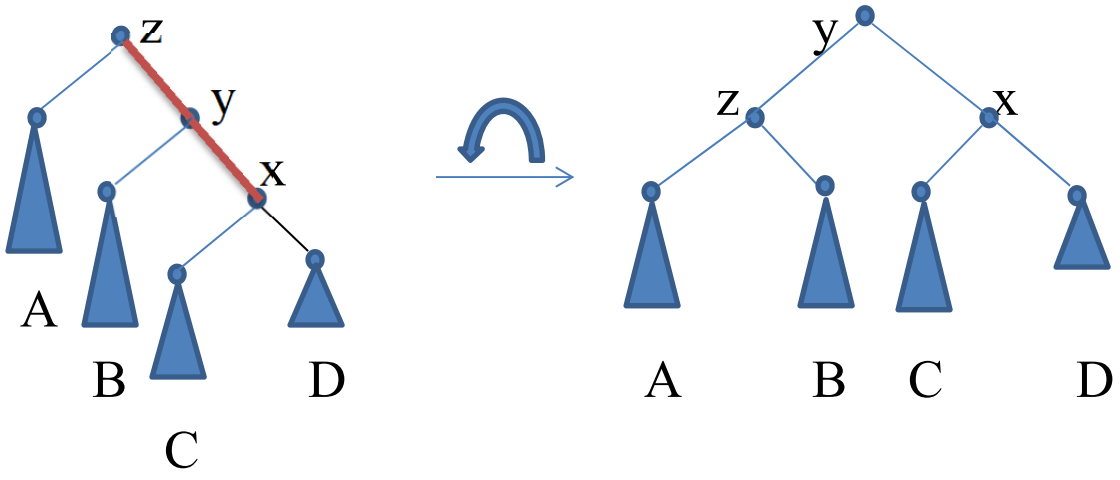
**Inserting into an AVL tree T:**

If an insertion causes T to be unbalanced, we will have to travel a path P from the point of insertion to the root of T so as to find three nodes x, y, and z on P, where z is the first unbalanced node encountered, y is the child of z, and x is the child of y. Based on the parent-child relationship among x, y, and z, we will perform one of the following four rotations exactly once.

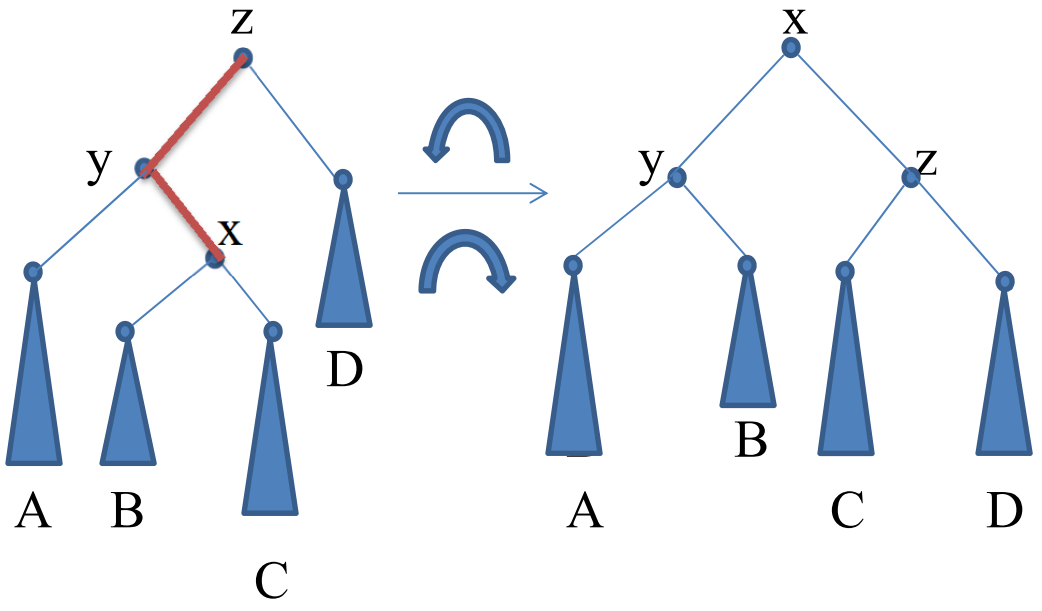
1. **LL-Rotation**: An insertion onto the *Left subtree of the Left child of z*.



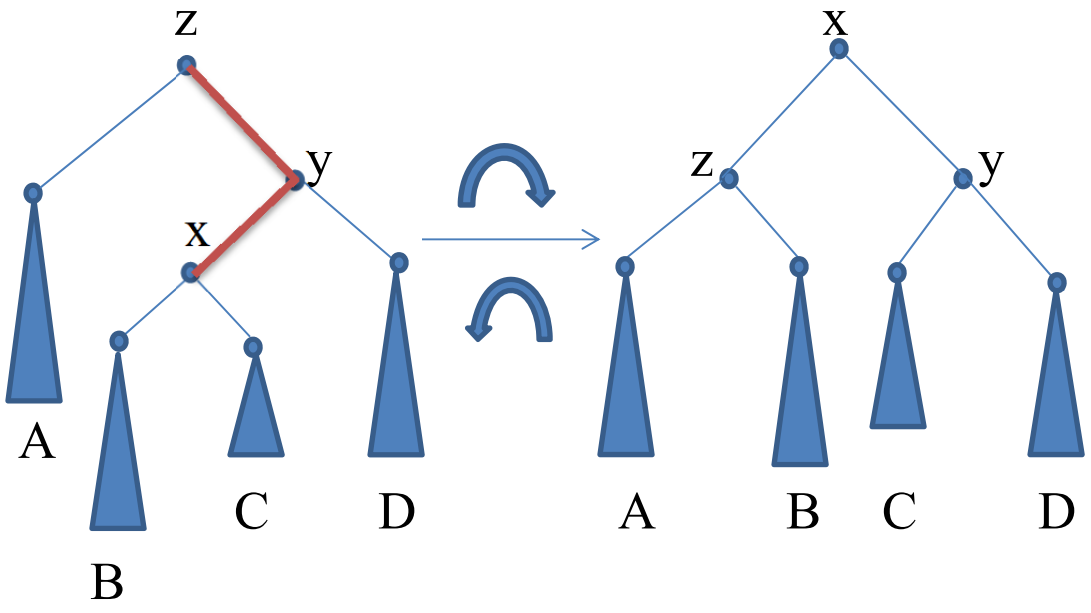
2. **RR-Rotation:** An insertion onto the *Right subtree of the Right child of z*.




3. **LR-Rotation:** An insertion onto the *Right subtree of the Left child of z*.



4. **RL-Rotation:** An insertion onto the *Left subtree of the Right child of z*.




**Remarks:**

(1) In traversing a path P from the point of insertion to the root of T to find the first unbalanced node z on P, the height information of each node encountered along P will be updated.

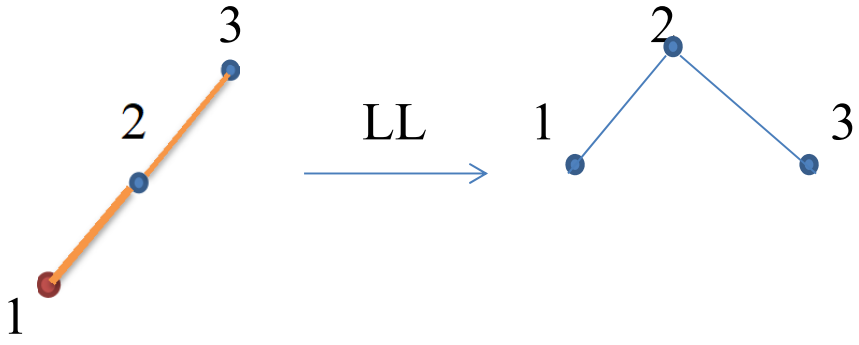
(2) After insertion, if the resulting BST becomes unbalanced, then exactly one of the above four rotations will be executed once.

**Example:** Insert <3, 2, 1, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10, 8, 9>, in the given order, into an initially empty AVL tree.

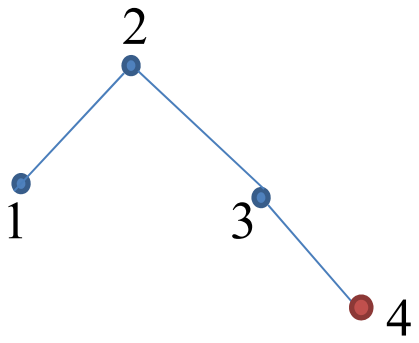
**Insert(3):**


**Insert(2):**

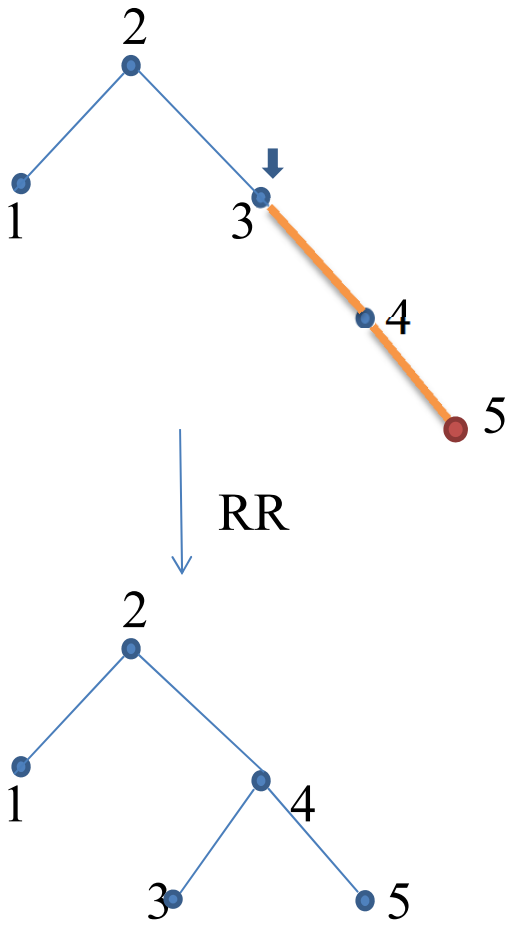

**Insert(1):**



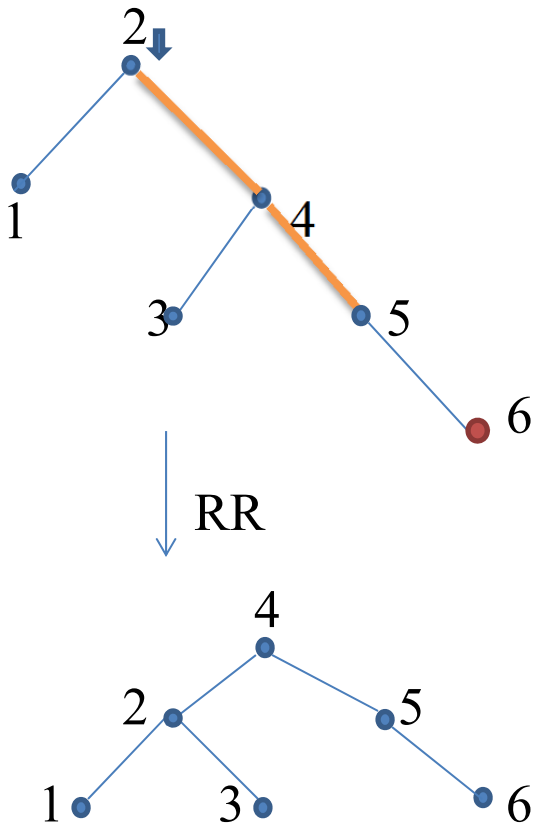
**Insert(4):**



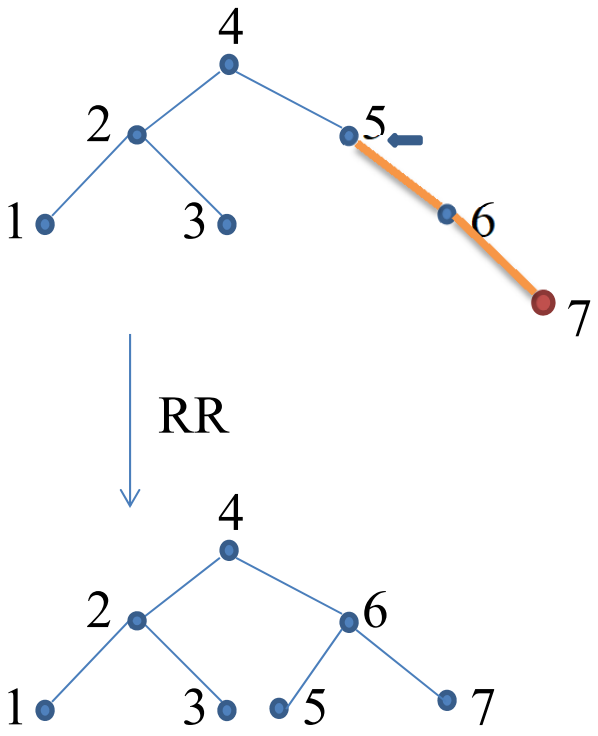

**Insert(5):**



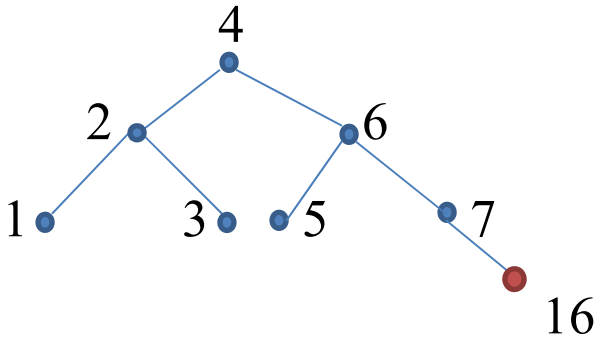
**Insert(6):**



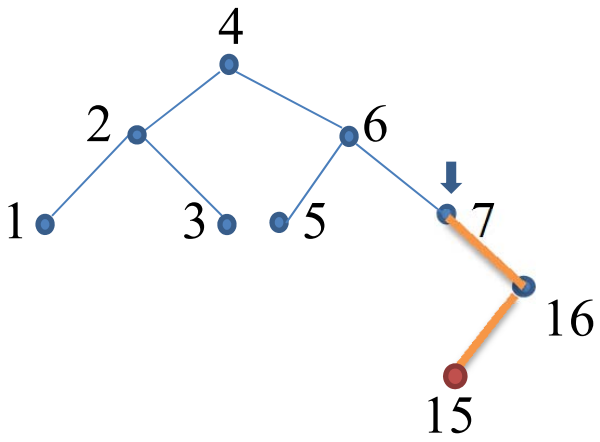
**Insert(7):**

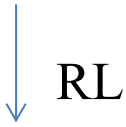


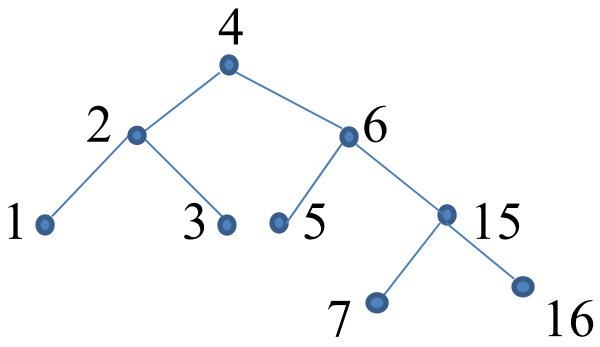

**Insert(16):**



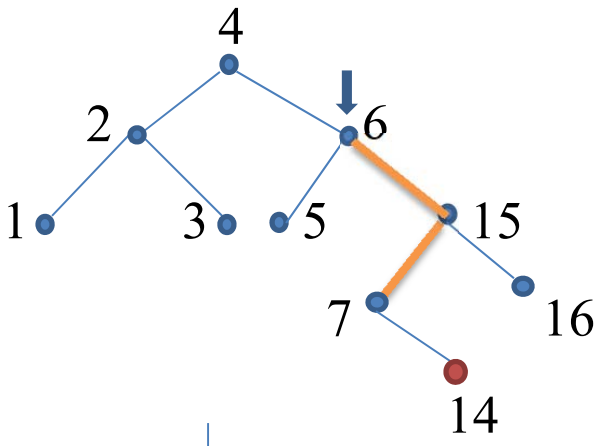
**Insert(15):**



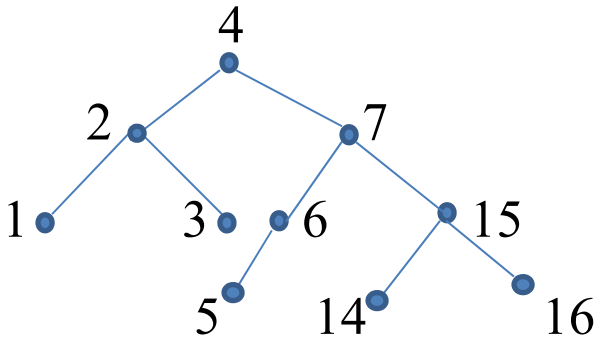



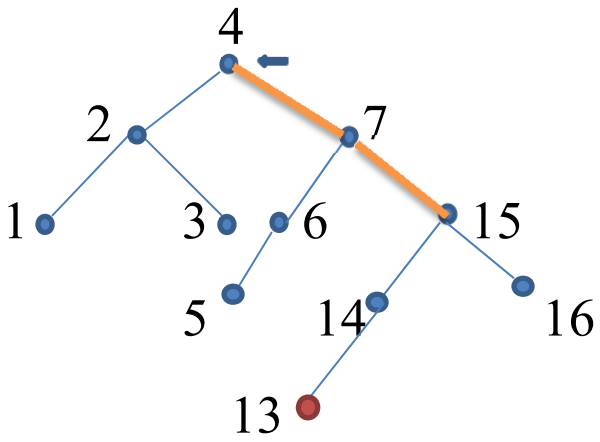
**Insert(14):**



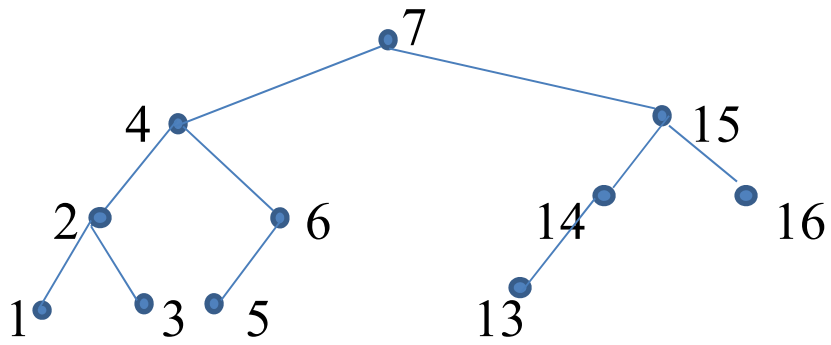




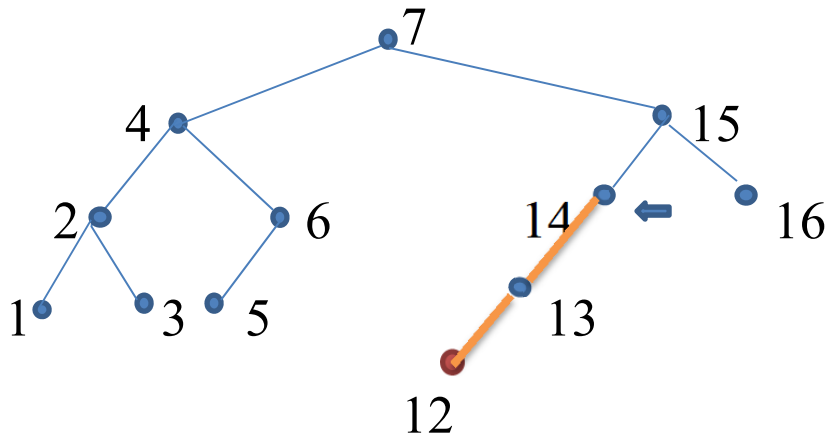
**Insert(13):**



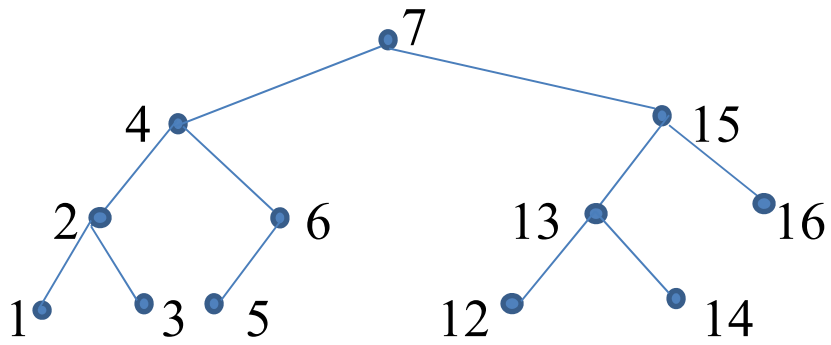




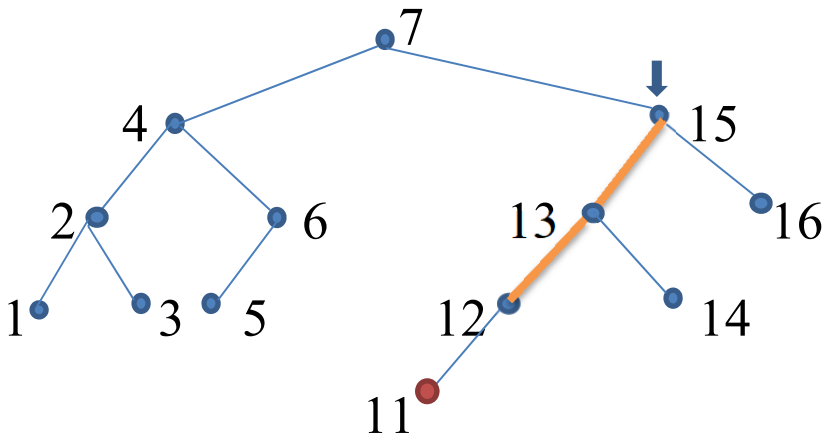
**Insert(12):**

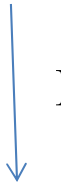




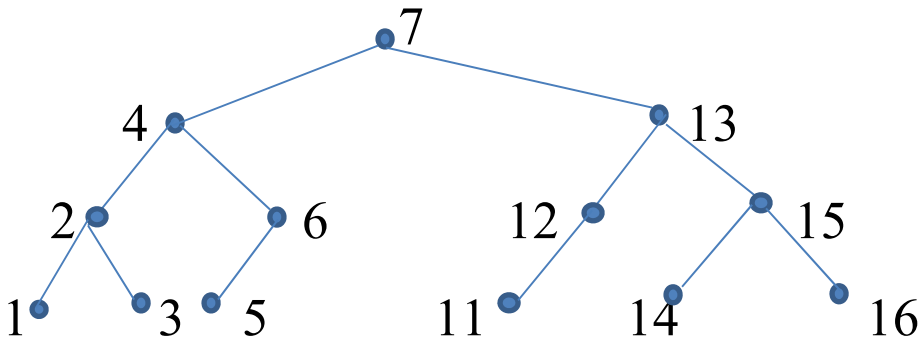


**Insert(11):**

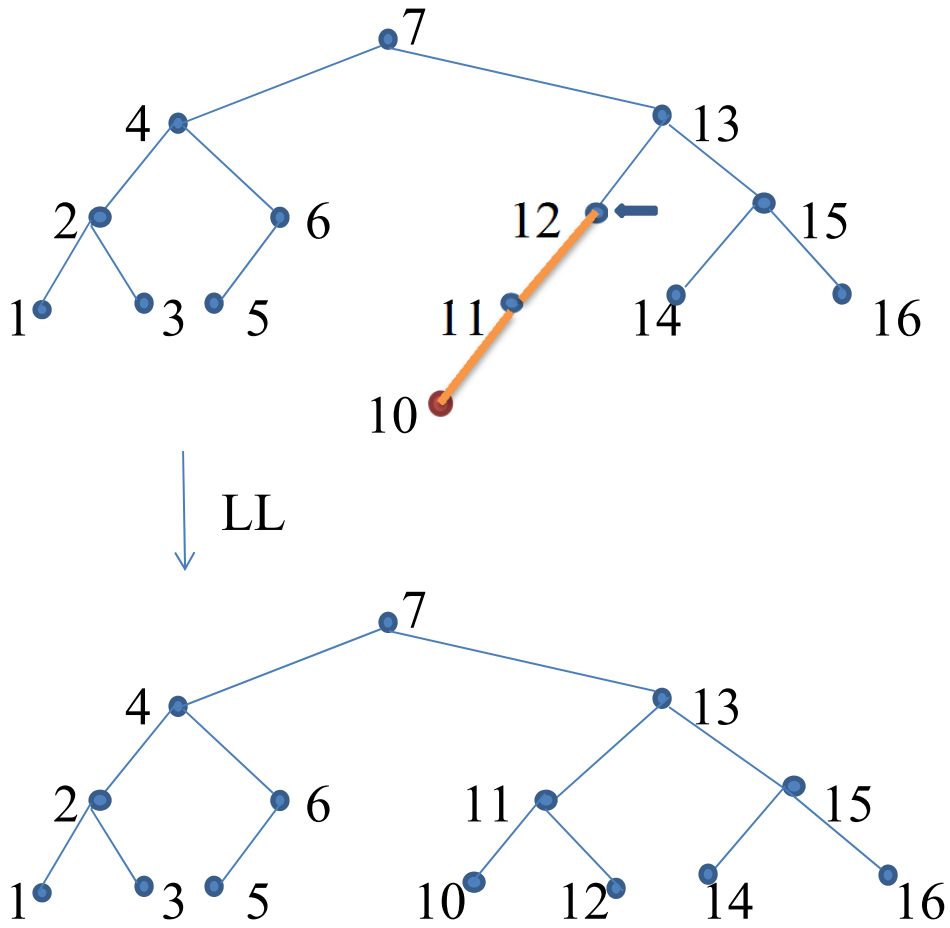


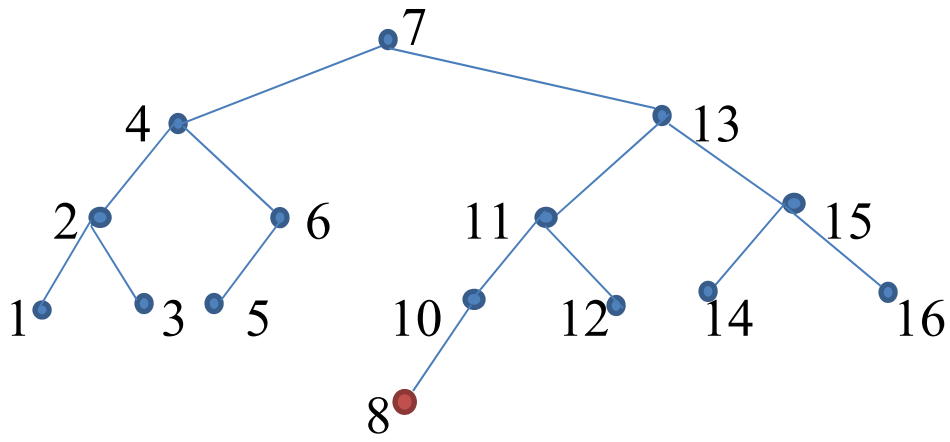
LL



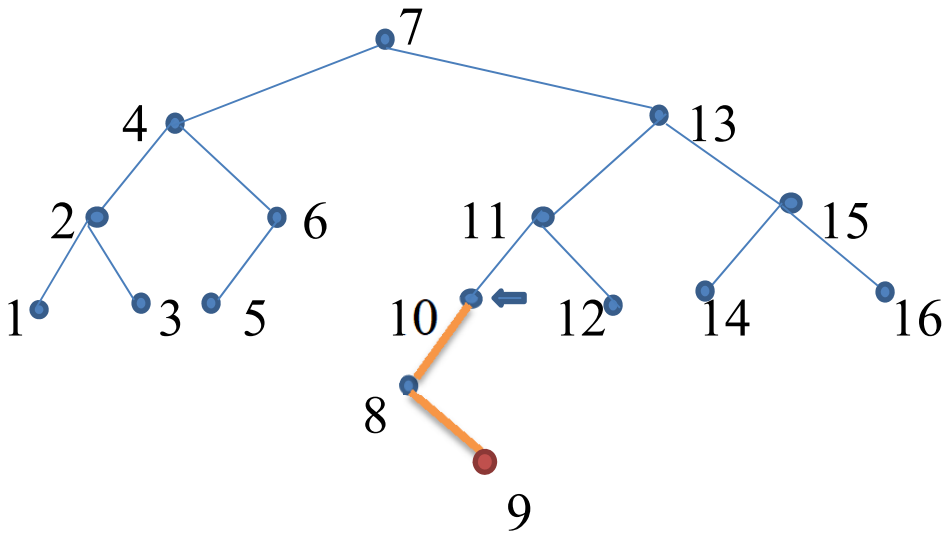
**Insert(10):**

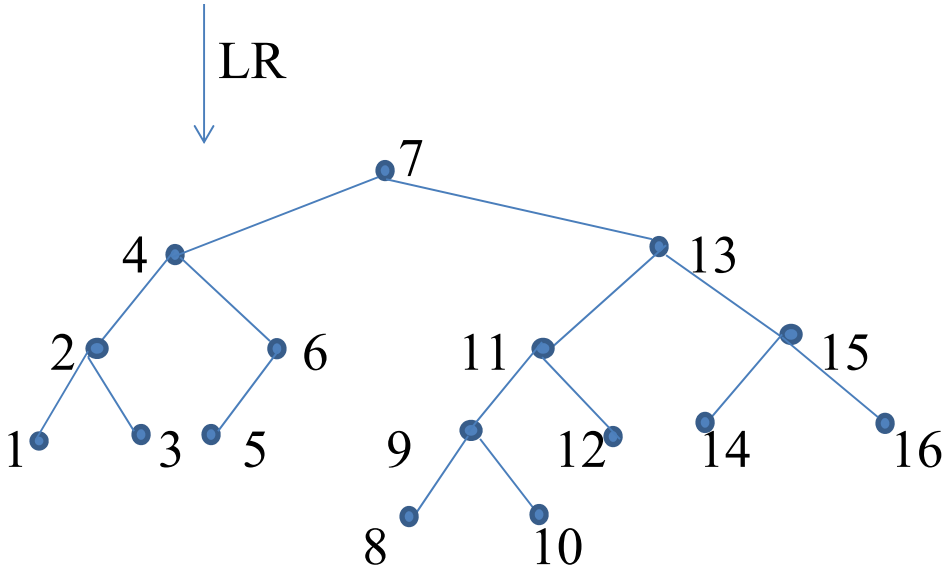



**Insert(8):**



**Insert(9):**



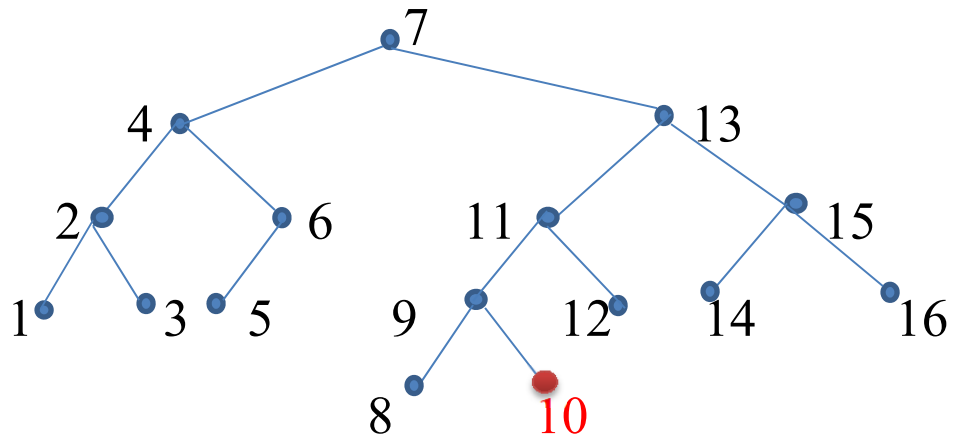



**Deleting a node from an AVL Tree T:**

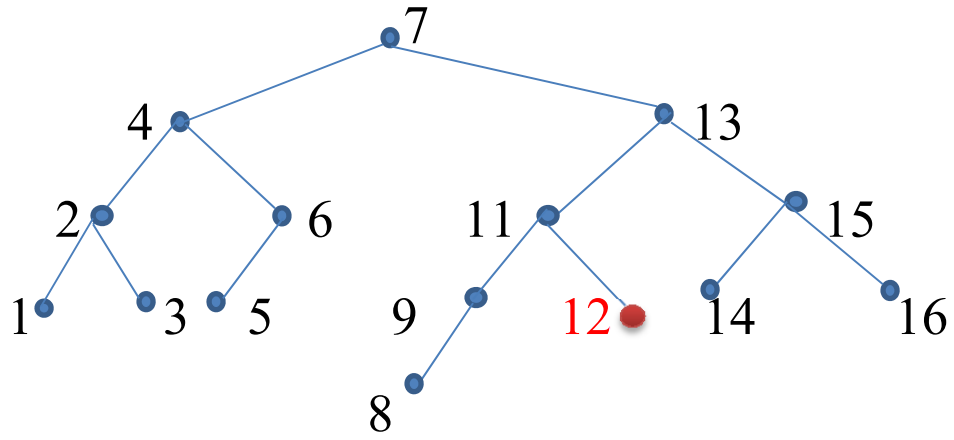
**Step 1:** First, perform delete() operation as in regular BST. If the node to be deleted is a leaf node in T, just delete it. Else, if the node to be deleted is not a leaf node, then we must replace it by using another node in T.

**Step 2:** After a node has been deleted from T, we will have to travel a path from the point of deletion to the root to find the first unbalanced node z. Once found, select a child y of z with the larger height and a grandchild x of z with the larger height. When selecting x, if both subtrees of y are of the same height, we will select x in such a way that both z and y, and y and x, will have the same parent-child relation. Now, based on the parent-child relationship among x, y, and z, we will perform one of the previous four rotations. When done, repeat this process by following the path to the root to find the next unbalanced node z.

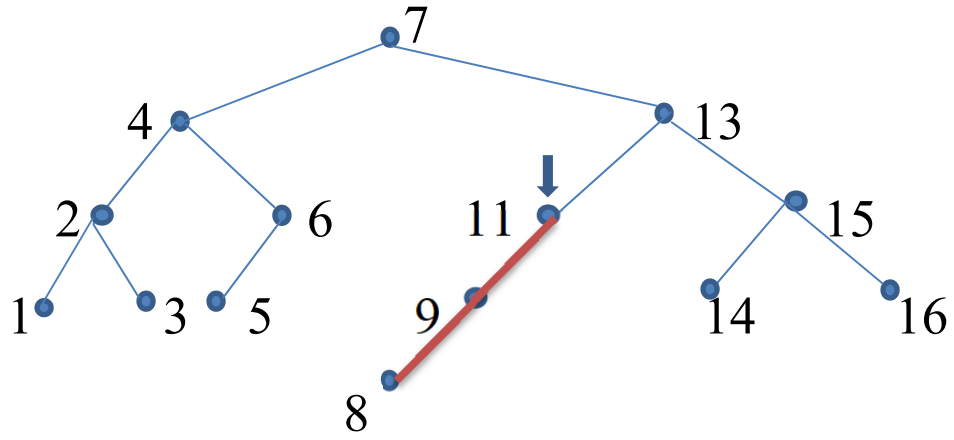
**Example:** Delete 10 and then 12 from the last AVL tree T.

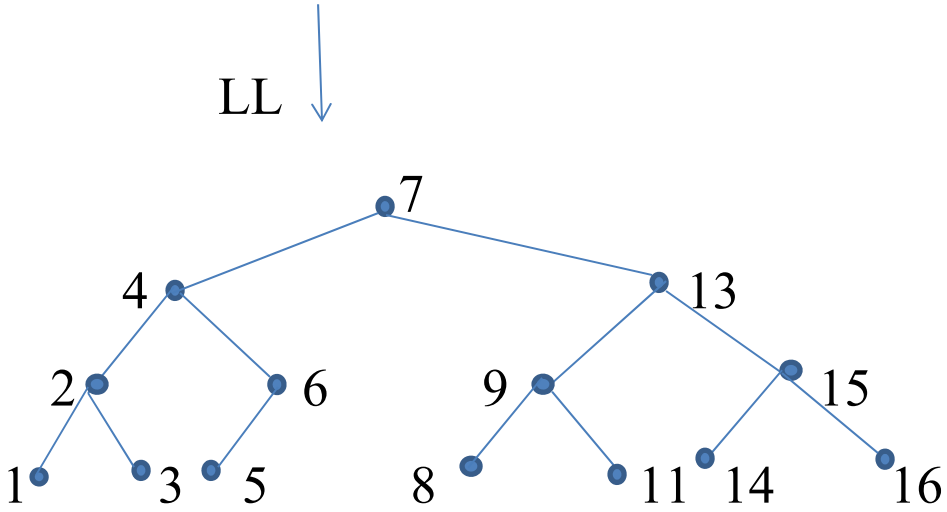



**Delete(10):** No re-balancing required.

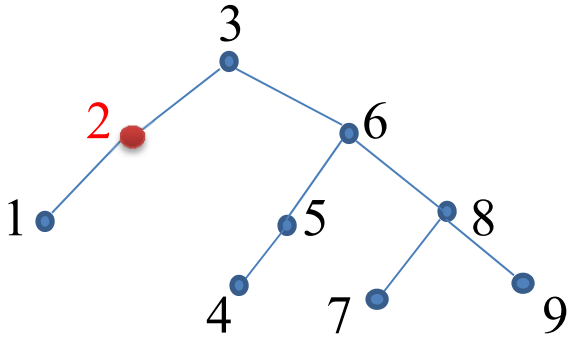



**Delete(12):**

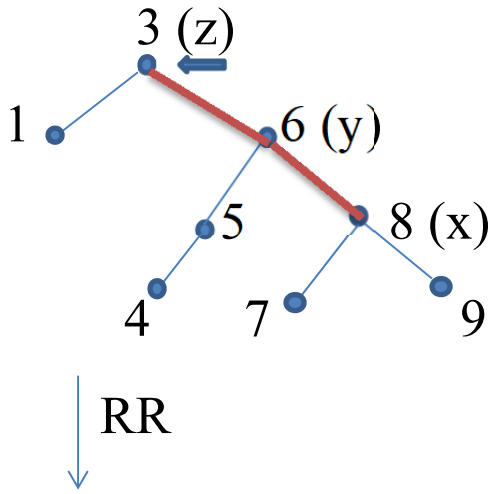


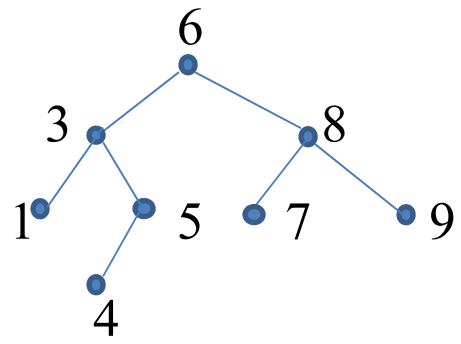



**Example:** Delete 2 from the given AVL tree.



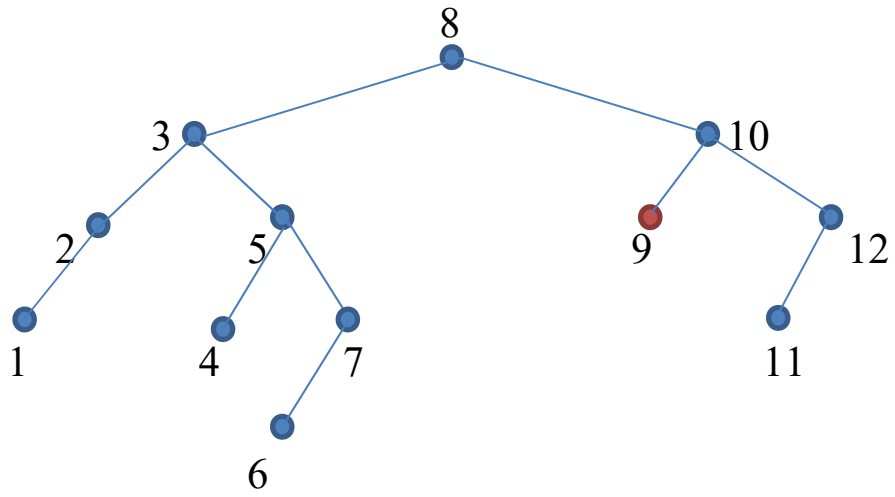

**Delete(2):**



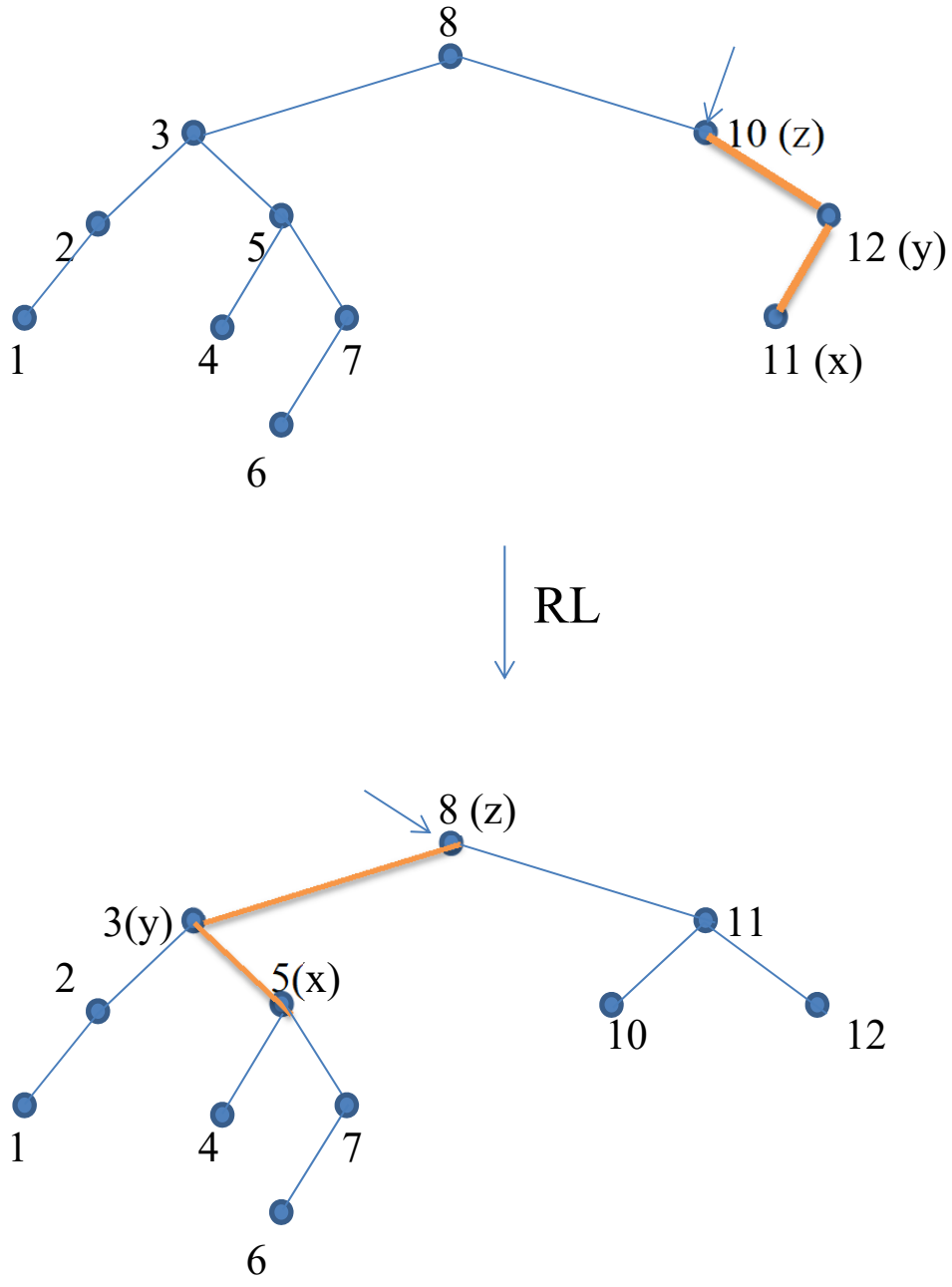



**Remark:** Observe that x is chosen in such a way that x is the right child of y since y is the right child of z.

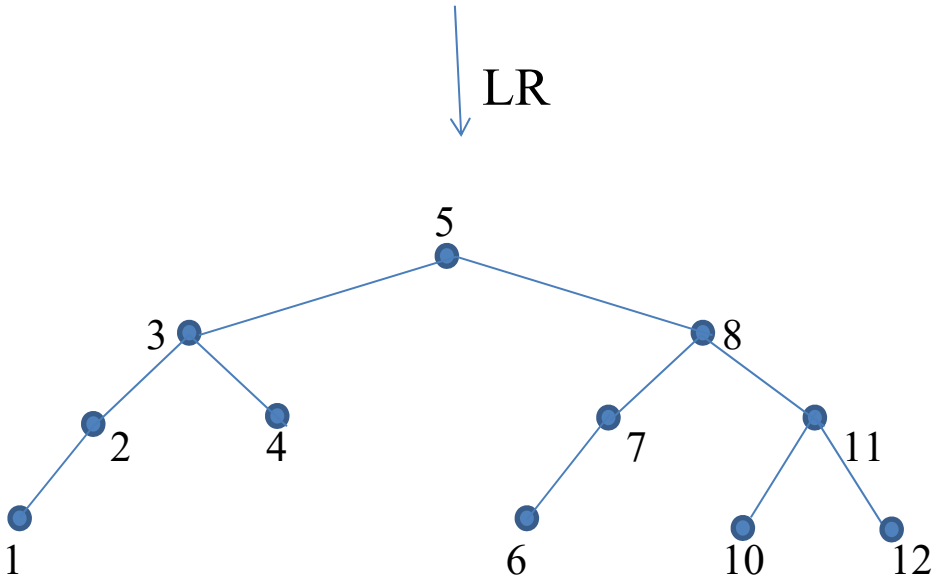
**Example:** Deletion in AVL tree may require more than one rotation.



Delete(9):




**Remark:** Observe that both y and x are chosen in such a way that the subtrees rooted at them have larger height than their siblings.




**HW:** Practice AVL operations.

**Implementation:**

Using children and parent pointers; must also store balancing/height information.

**Complexity:**

Find, findMin, findMax, delete, deleteMin, deleteMax:

$T_a(n) = T_w(n) = O(\lg n)$.

**Limitations of AVL trees:**

- Much more complicated than a regular BST due to rebalancing operations.
- Extra space needed to store balancing information for each node.
- Difficult to implement; insertion and deletion are much more expensive and error-prone.
- Independent of input data, most operations have complexity $T_a(n) = T_w(n) = O(\lg n)$.
  Hence, an AVL tree cannot take advantage of "locality" of data.

**Observation:**

**The 90-10 rule:** In practice, 90% of the accesses are to the 10% of the data.

To improve efficiency in subsequent searching, a recently accessed item should be moved closer to the root of the search tree.

**Self-Adjusted Binary Search Tree:**

A BST T in which every operation on an element x in T will re-arrange the elements in T so that the node x will be placed at the root of the resulting BST.

**Splay Tree (Sleator & Tarjan):** Not included in final

A self-adjusted BST in which every operation on an element x in T will use a technique of ***splaying***, splay(x,T), to relocate a node x to the root of the resulting BST.

**Remarks:**

1. The basic operation in a splay tree is the splay function splay(x,T), which will splay the object x to the root of the resulting tree.
2. Every BST operation on an element x must use the splay function splay(x,T). Hence, it will always operate on the root of the tree.
3. Splay tree is a BST that may not be balanced at time.
4. As nodes are accessed over time, the splay tree will be re-organized and become more balanced.

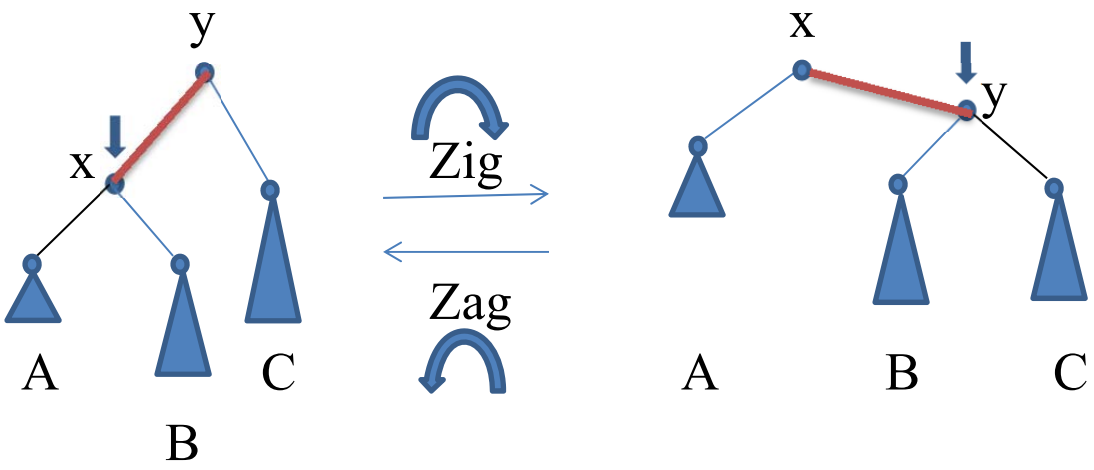
**Q:** How do we execute the splay(x,T) function?

Use a sequence of rotation operations similar to AVL tree.

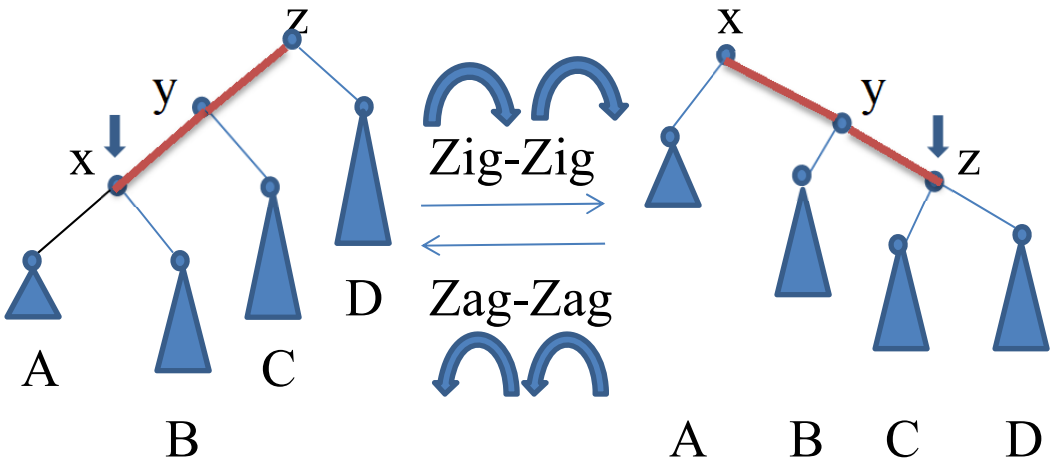
**Executing splay(x,T):**

Repeatedly rotate x up T using the following operations until x becomes the root of the tree.

1. Zig and Zag rotations:
   Assume that x is a node in T with parent y and no grandparent.

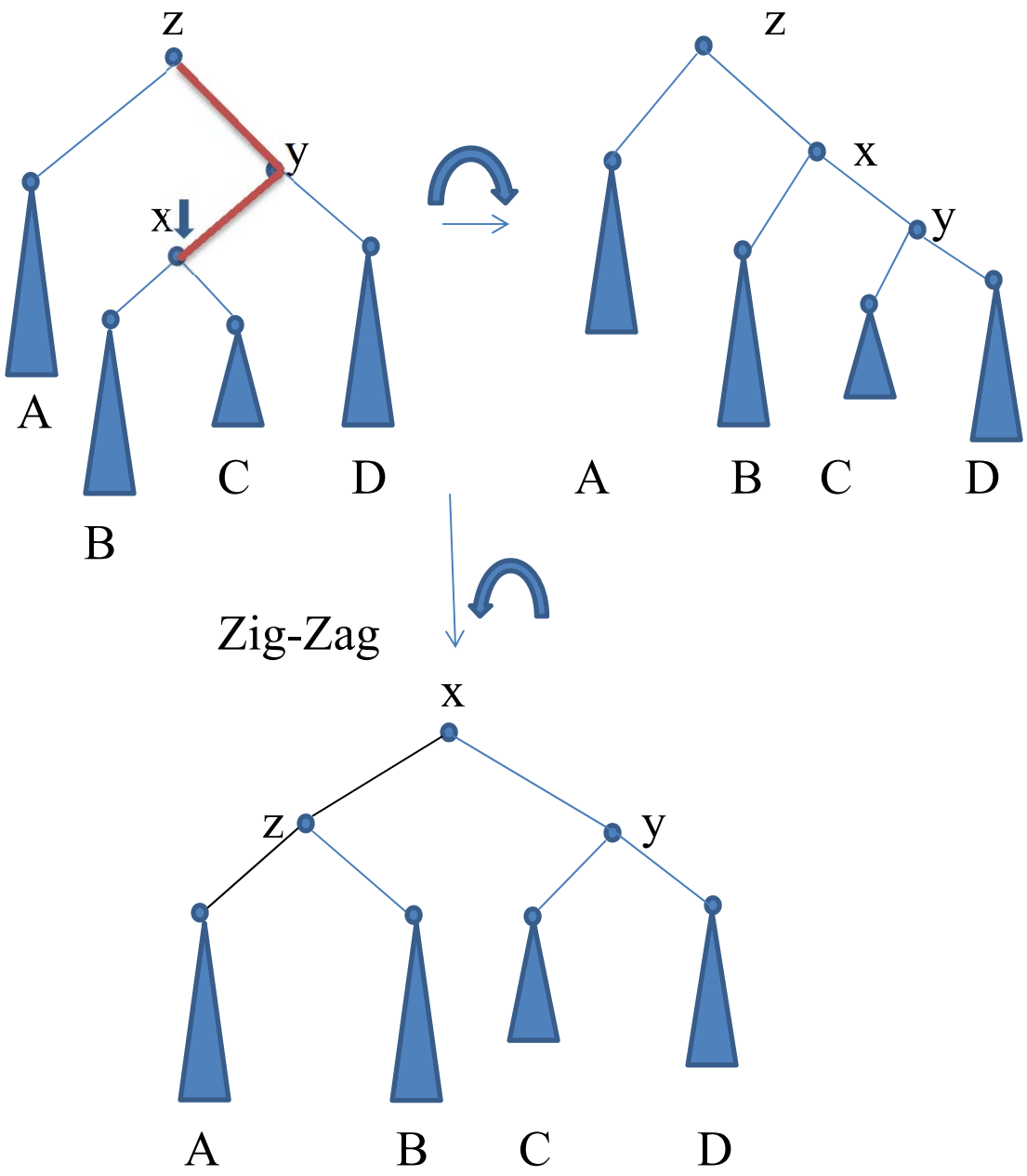



2. Zig-Zig and Zag-Zag rotations:
   Assume that x is a node in T with parent y and grandparent z.

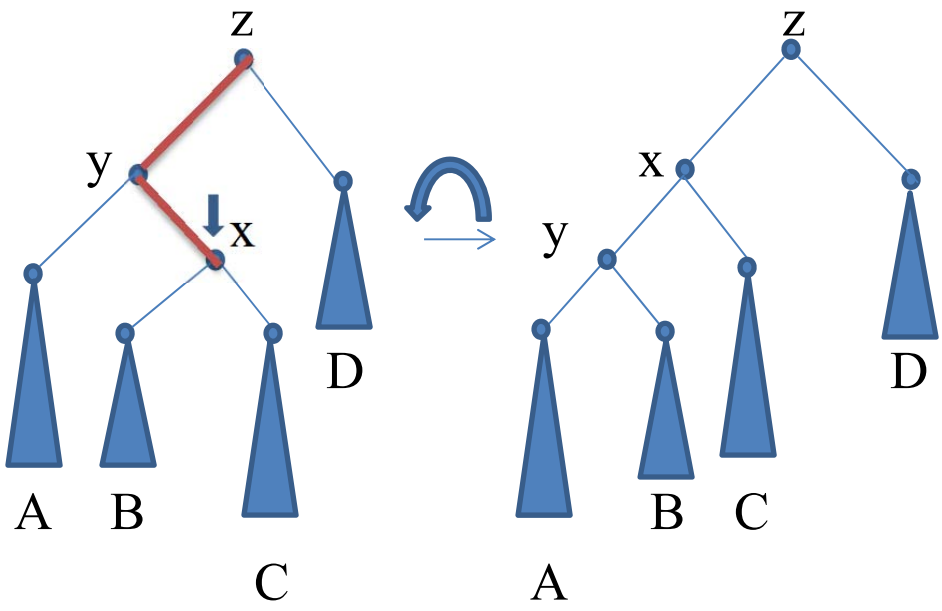


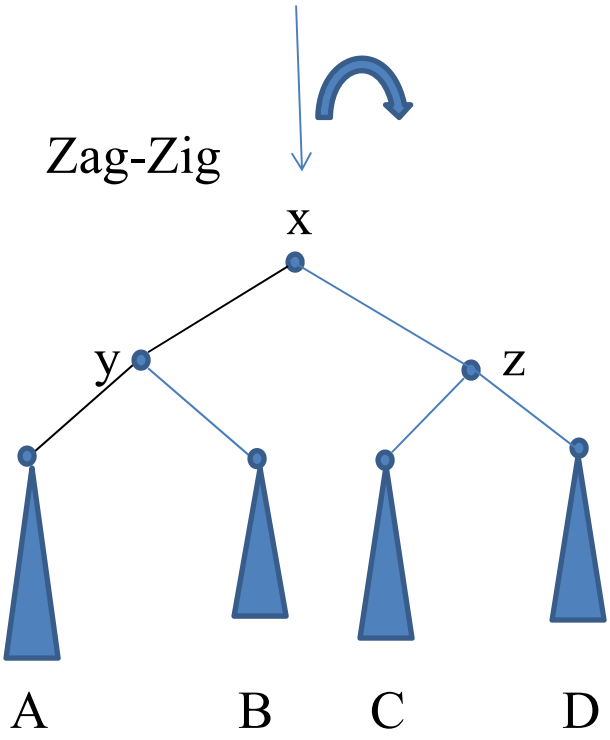
3. Zig-Zag and Zag-Zig rotations:
   Assume that x is a node in T with parent y and grandparent z.

   Zig-Zag Rotation:



Zag-Zig Rotation:



**Splay Tree Operations:**

1. Find(x,T): Search for x. If found, call splay(x,T).
   Remark: If not found, call splay(y,T), where y is the element accessed before reaching the NULL pointer.

2. Insert(x,T): Serach for x. If not found, create new node containing x and insert x as a leaf node into T. Call splay(x,T).
   Remark: If duplicate element is not allowed and x is found in T, call splay(x,T).

3. FindMin(T) or FindMax(T): Find min, or max, element m as in BST. Call splay(m,T).

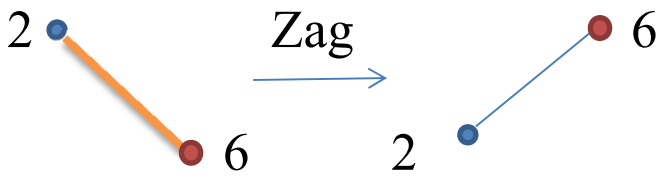
4. DeleteMin(T) or DeleteMax(T): Call FindMin, or FindMax(T), and then delete the root of the tree.

5. Delete(x,T): Search for x. If found, call splay(x,T) to move x to the root. Delete root to decompose T into its left and right subtrees $T_L$ and $T_R$. Call FindMax($T_L$) to transform $T_L$ into a BST with root m such that m has no right child. Now, attach $T_R$ as the new right child to $T_L$.
   Remark: If not found, call splay(y,T), where y is the element accessed before reaching the NULL pointer.)

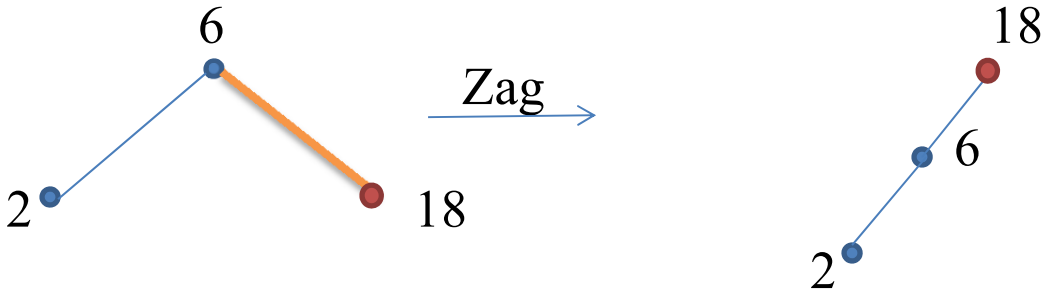
**Example:** Insert 2, 6, 18, 8, 12, 7, 15 into an empty splay tree.

**Insert(2):**

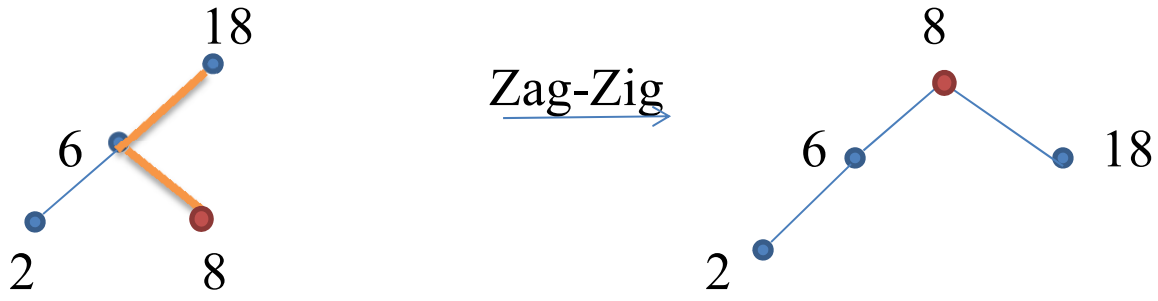

**Insert(6):**



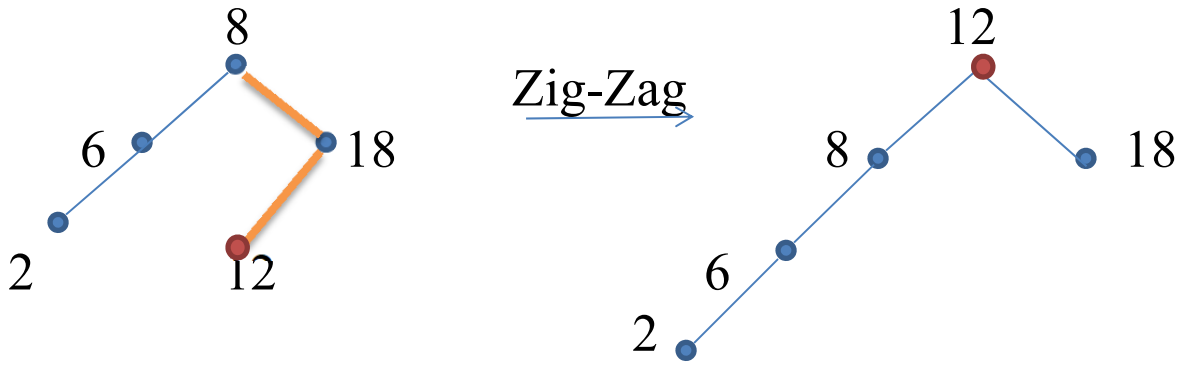

**Insert(18):**



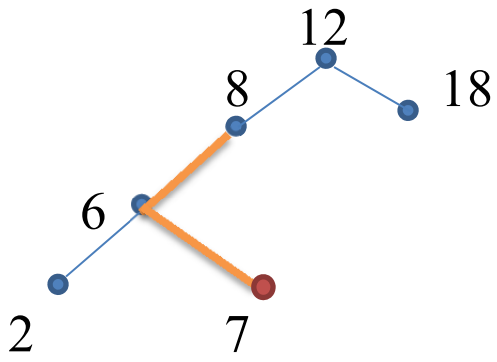

**Insert(8):**

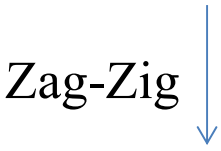


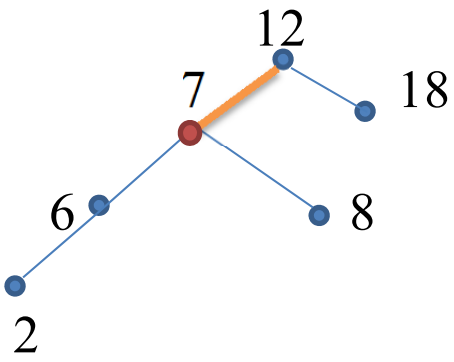
**Insert(12):**

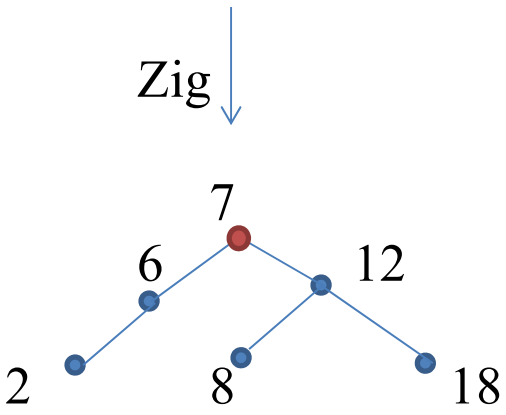



**Insert(7):**

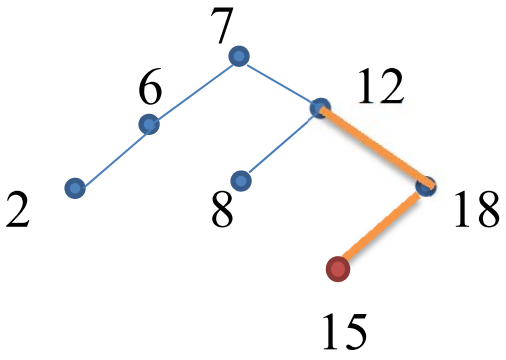


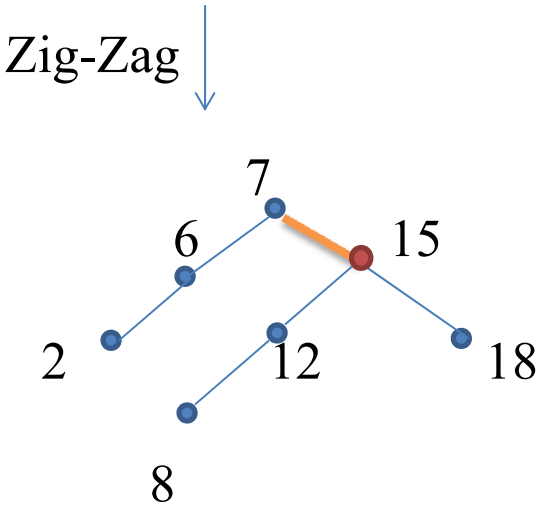



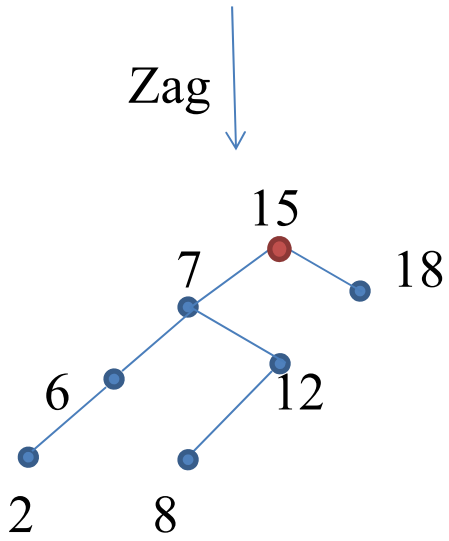



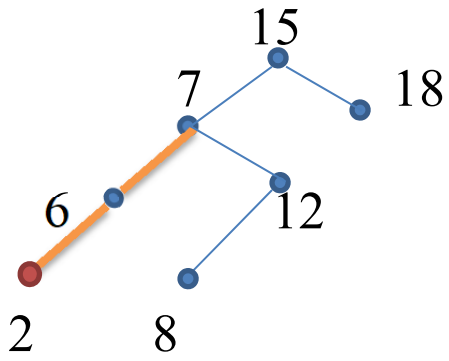

**Insert(15):**

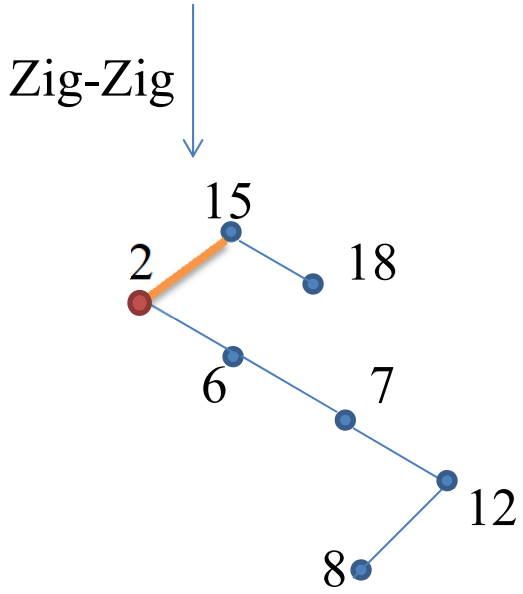


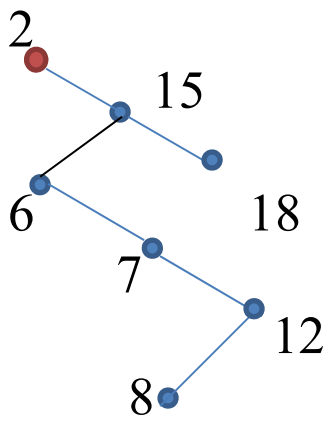



**DeleteMin(T):**

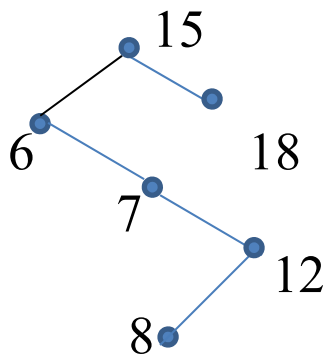


Zig

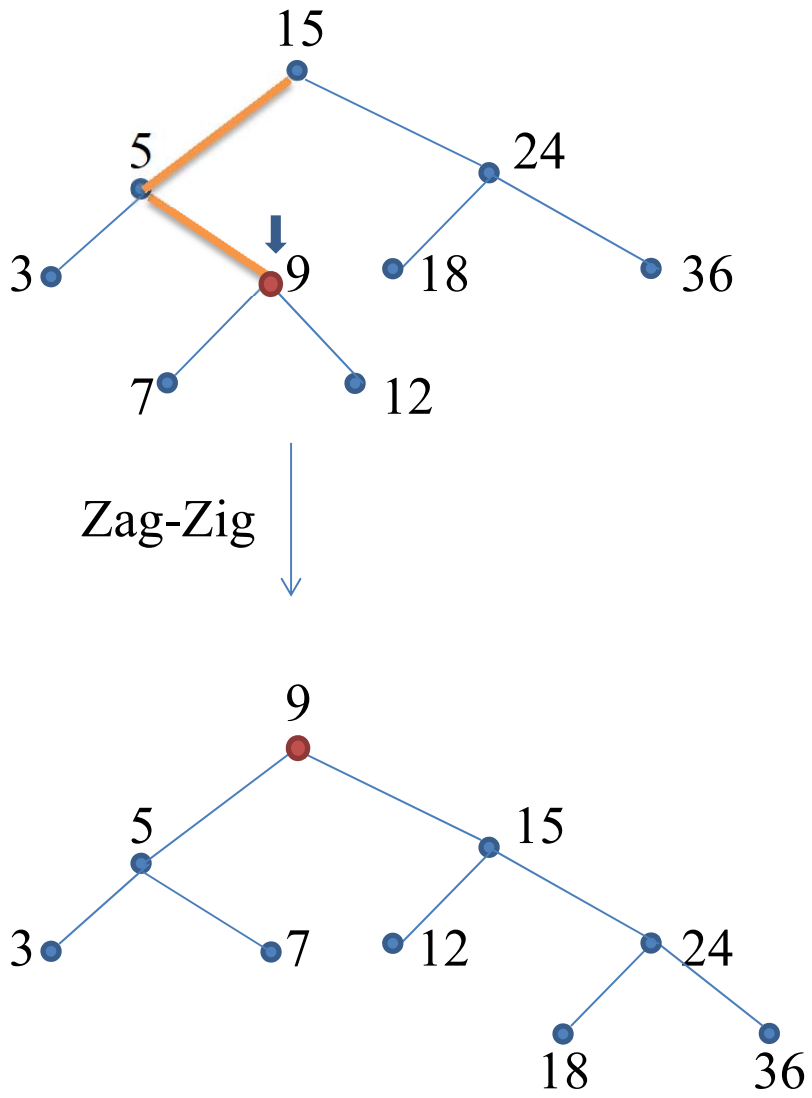


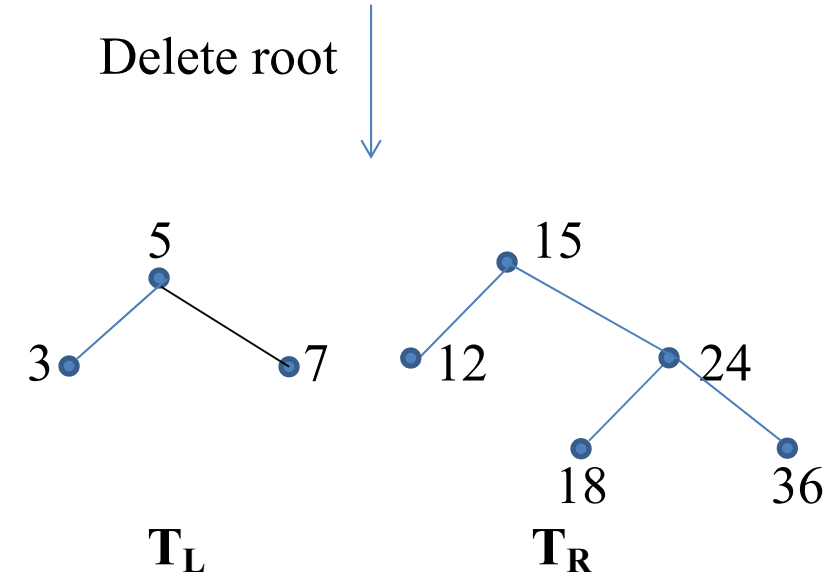

delete root

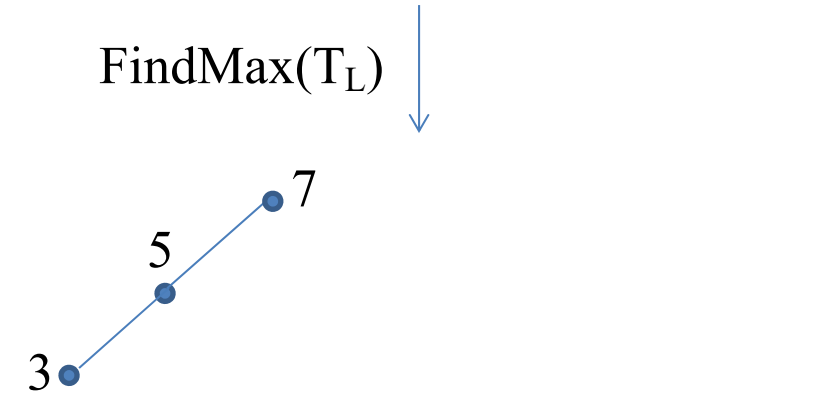


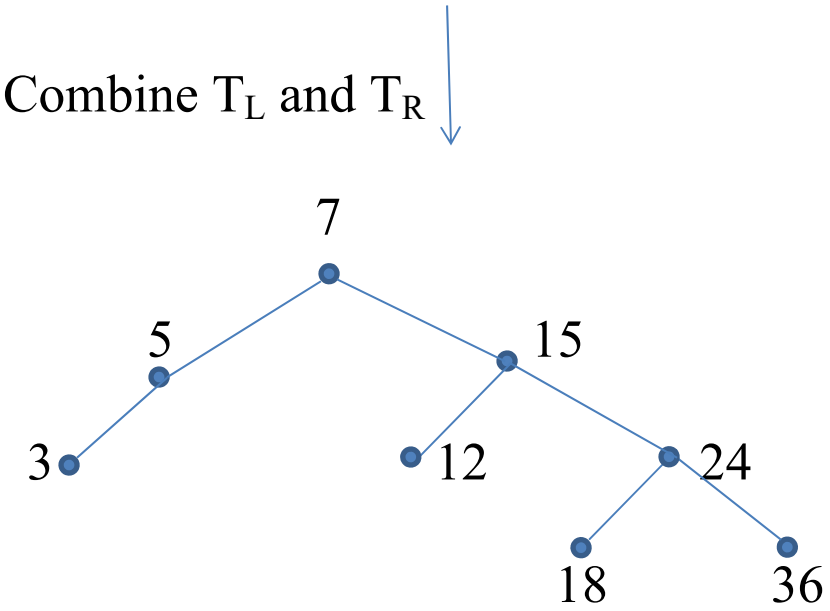
**Example:** General delete(x,T) in splay tree.

**Delete(9):**




Delete root
5
3
7
15
12
24
18
36
$T_L$
$T_R$


FindMax($T_L$)
7
5
3


Combine $T_L$ and $T_R$
7
5
15
3
12
24
18
36

**Implementation:**

Using children and parent pointers; no balancing information needs to be stored (required for AVL trees).

**Complexity:**

Find, findMin, findMax, delete, deleteMin, deleteMax:

$T_w(n) = O(n)$,

$T_a(n) = O(\lg n)$.

**Amortized complexity:**

For a collection of m search tree operations on a tree with n objects, $T(m,n) = O(m \lg n)$ with amortized cost of $O(\lg n)$ per operation.

**Remarks:**

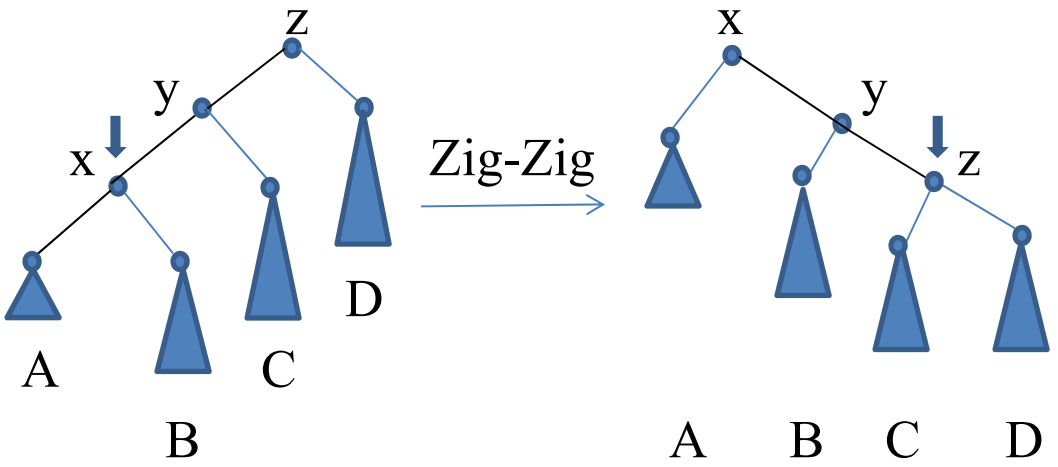
1. Splay trees are just regular BST with splay operations and without balancing requirement.
2. Splay tree may not be balanced.
3. Over time, splay trees tend to be balanced as they re-organize their tree structures.
4. Splay trees have very good locality property with recently accessed items always near the root of the trees.

**HW:** Practice splay tree operations.

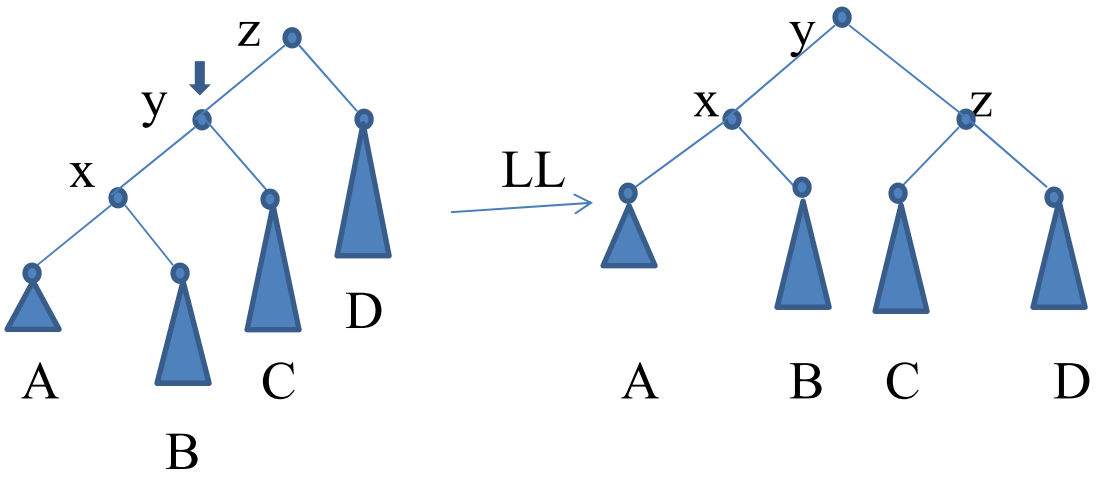
**Comparing Rotation Operations between AVL and Splay Trees:**

1. Zig or Zag rotation in splay trees:
   Not exist in AVL trees.

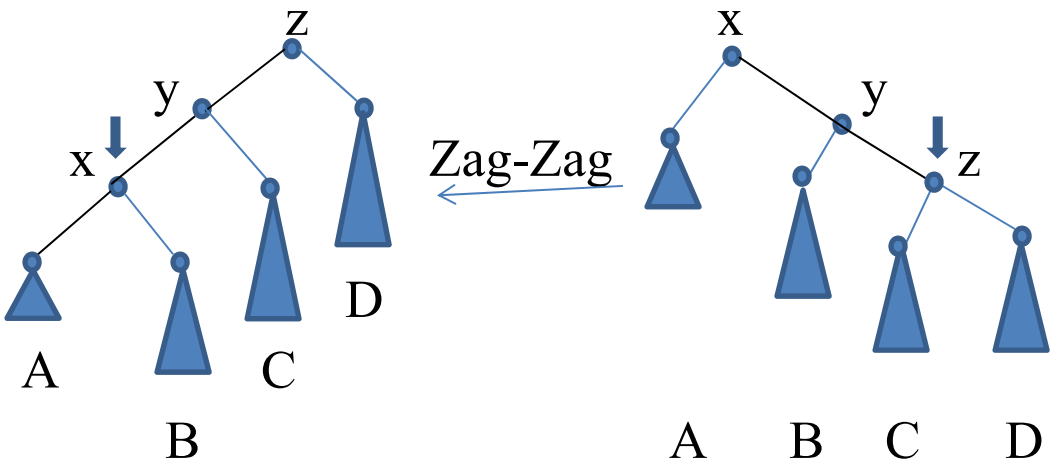
2. Zig-Zig rotation in splay trees:
   Splay Tree:



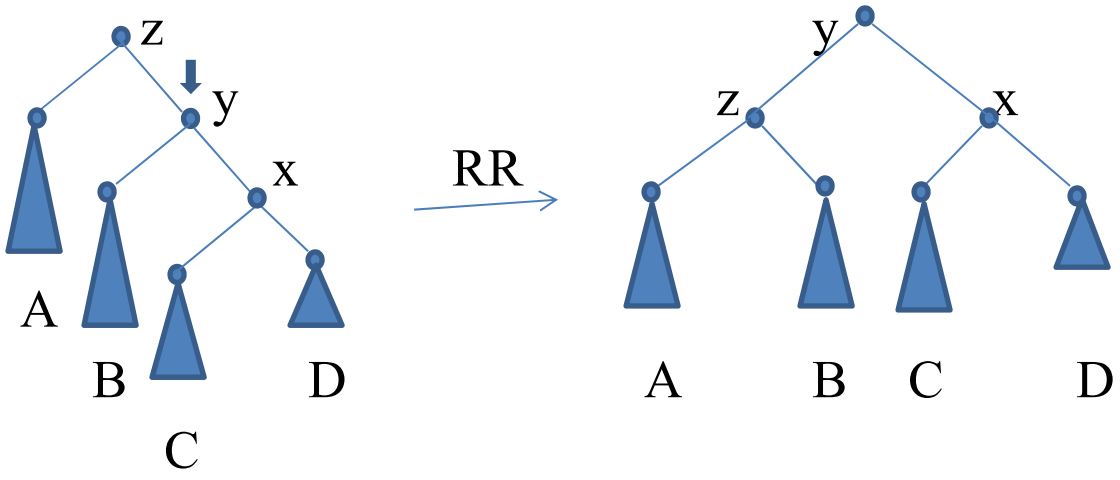

AVL Tree:



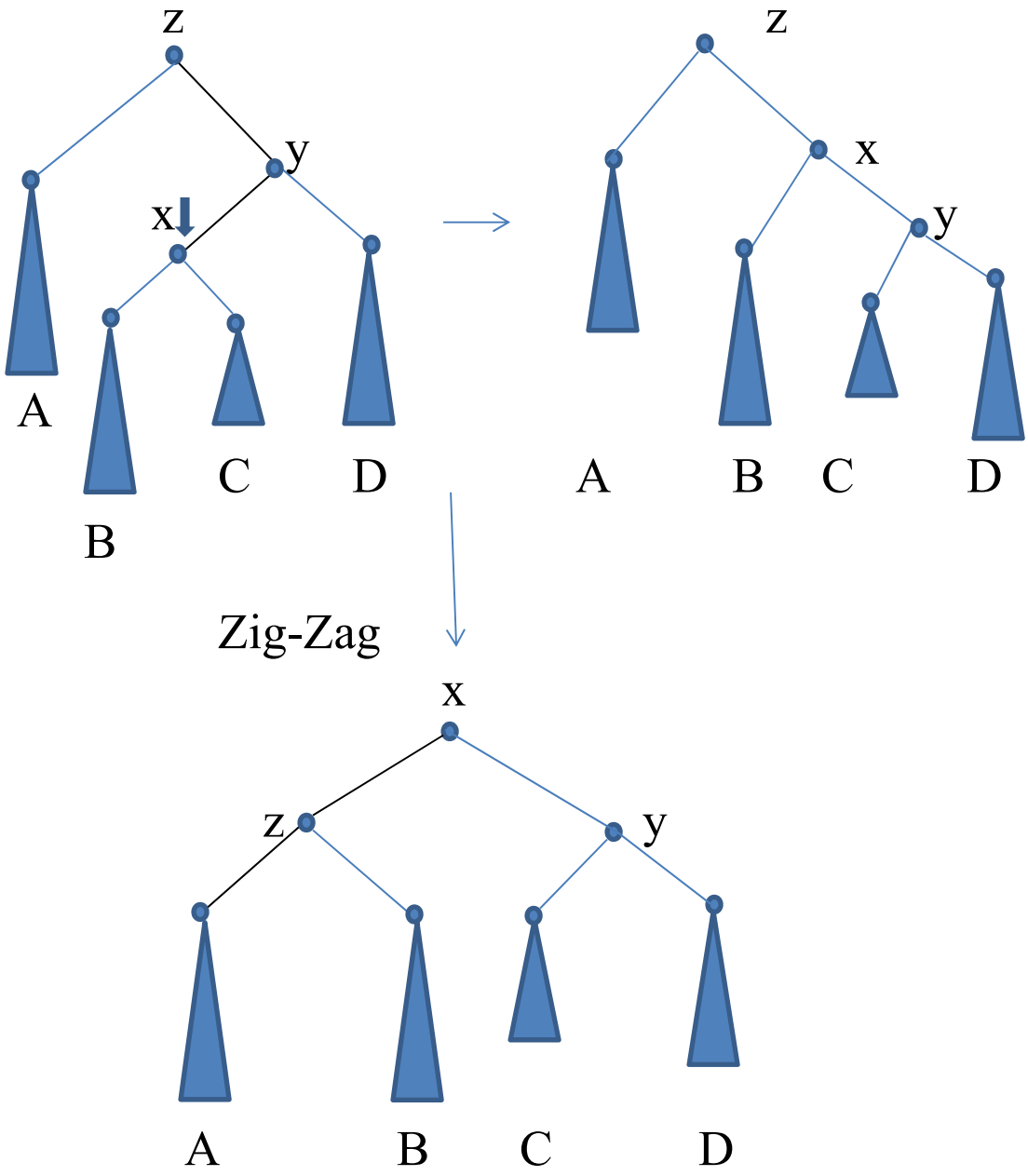
3. Zag-Zag rotation in splay trees:
   Splay Tree:



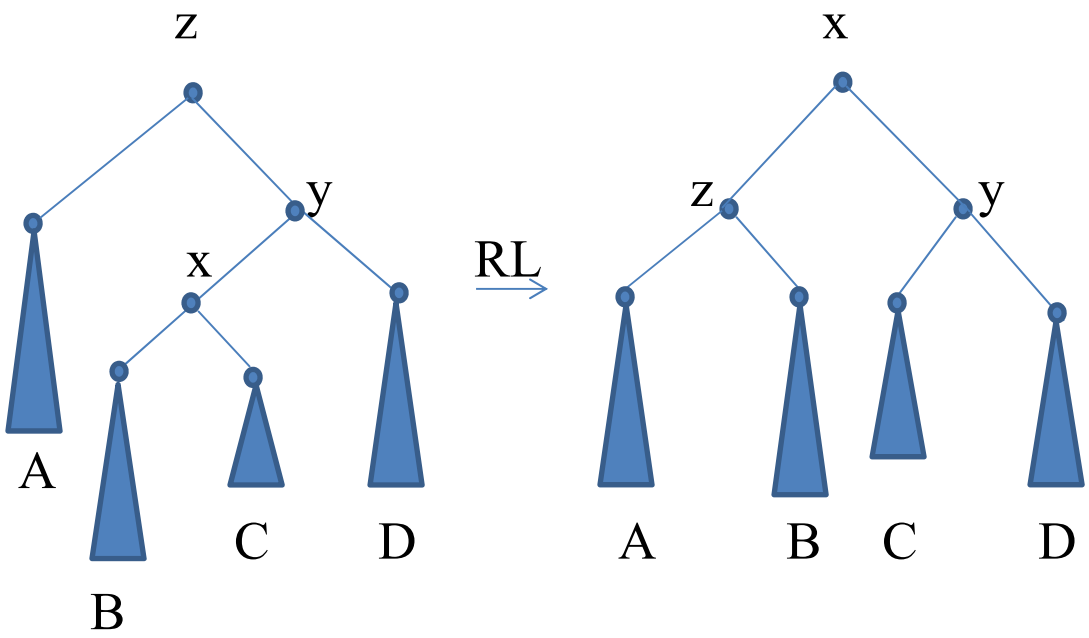

AVL Tree:



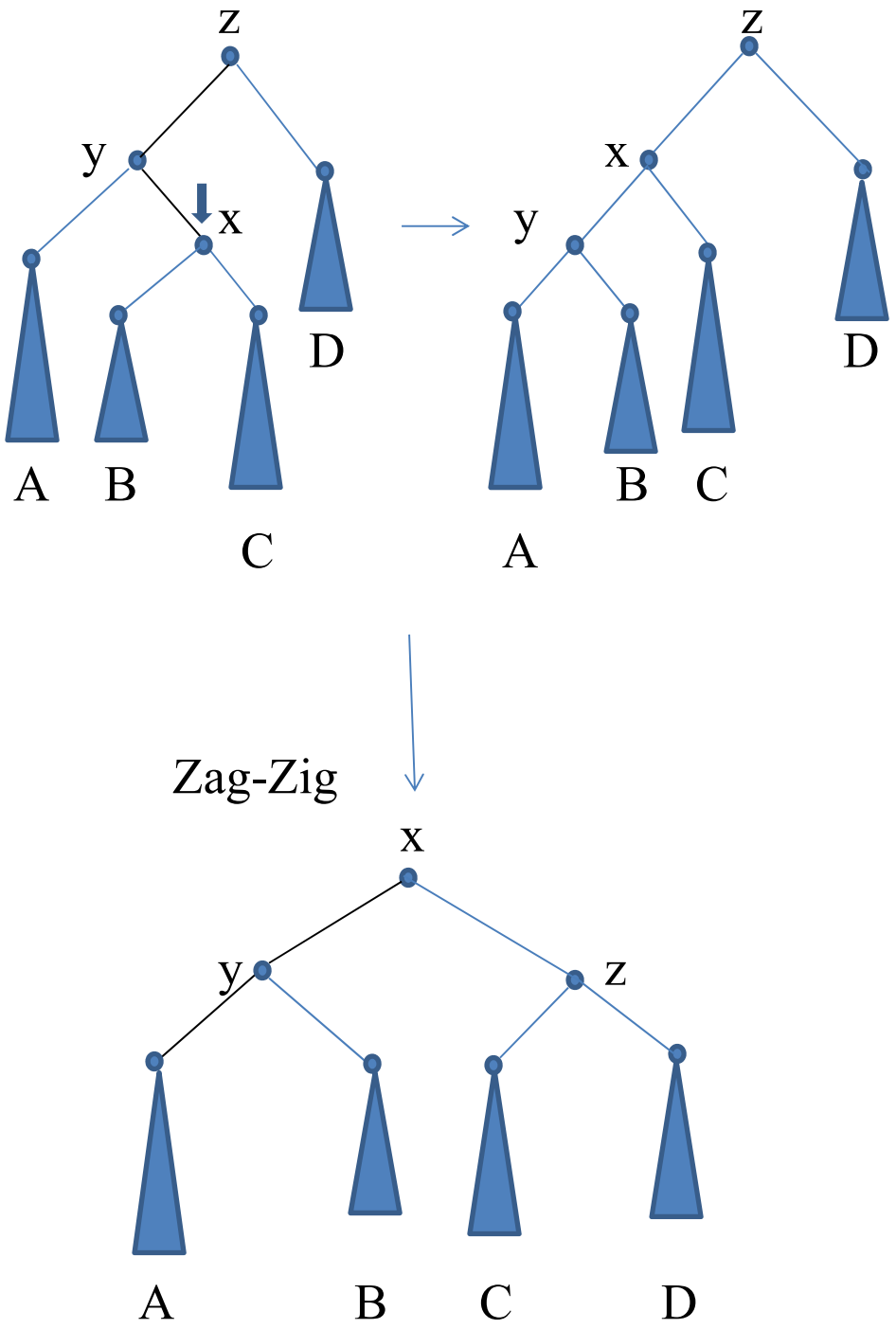
4. Zig-Zag rotation in splay trees:
   Splay Tree:



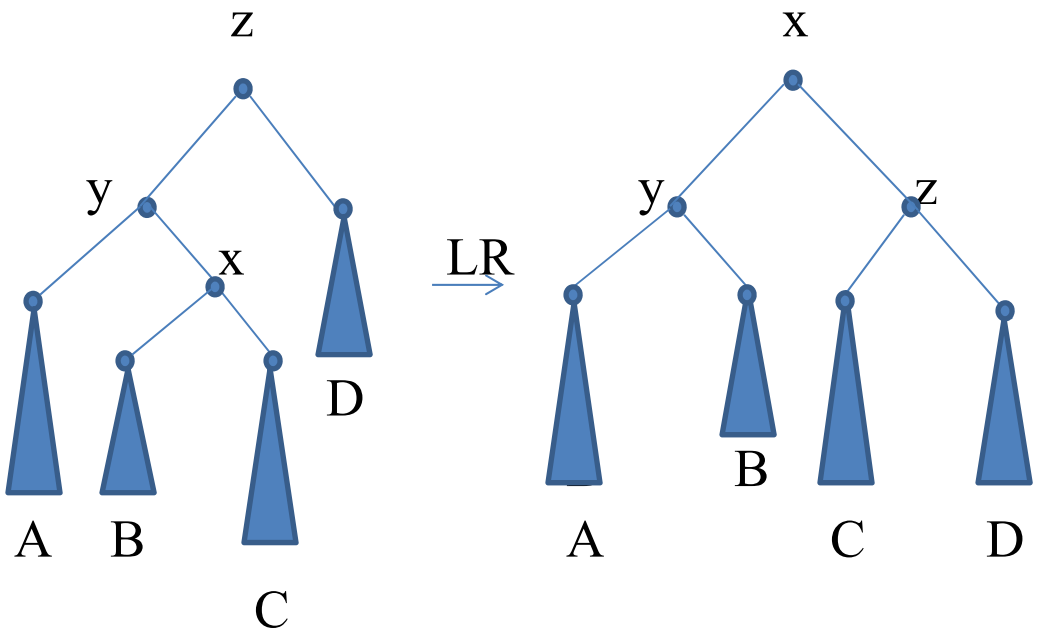
AVL Tree:



5. Zag-Zig rotation in splay trees:
   Splay Tree:



AVL Tree:




*11/27/18*