CSC230

Intro to C++ Lecture 21

Outline

- Binary Search Tree
- AVL Tree

Printing Contents of BST

Because of ordering rules for a BST, it's easy to print the items in alphabetical order

- Recursively print left subtree
- Print the node
- Recursively print right subtree

```
/** Print BST root in alphabetic order */
template <class T>
void show(TreeNode<T>* root){
 if(root == nullptr) return;
 show(root->getLeft());
 cout << root->getDatum() << endl;
 show(root->getRight());
}
```

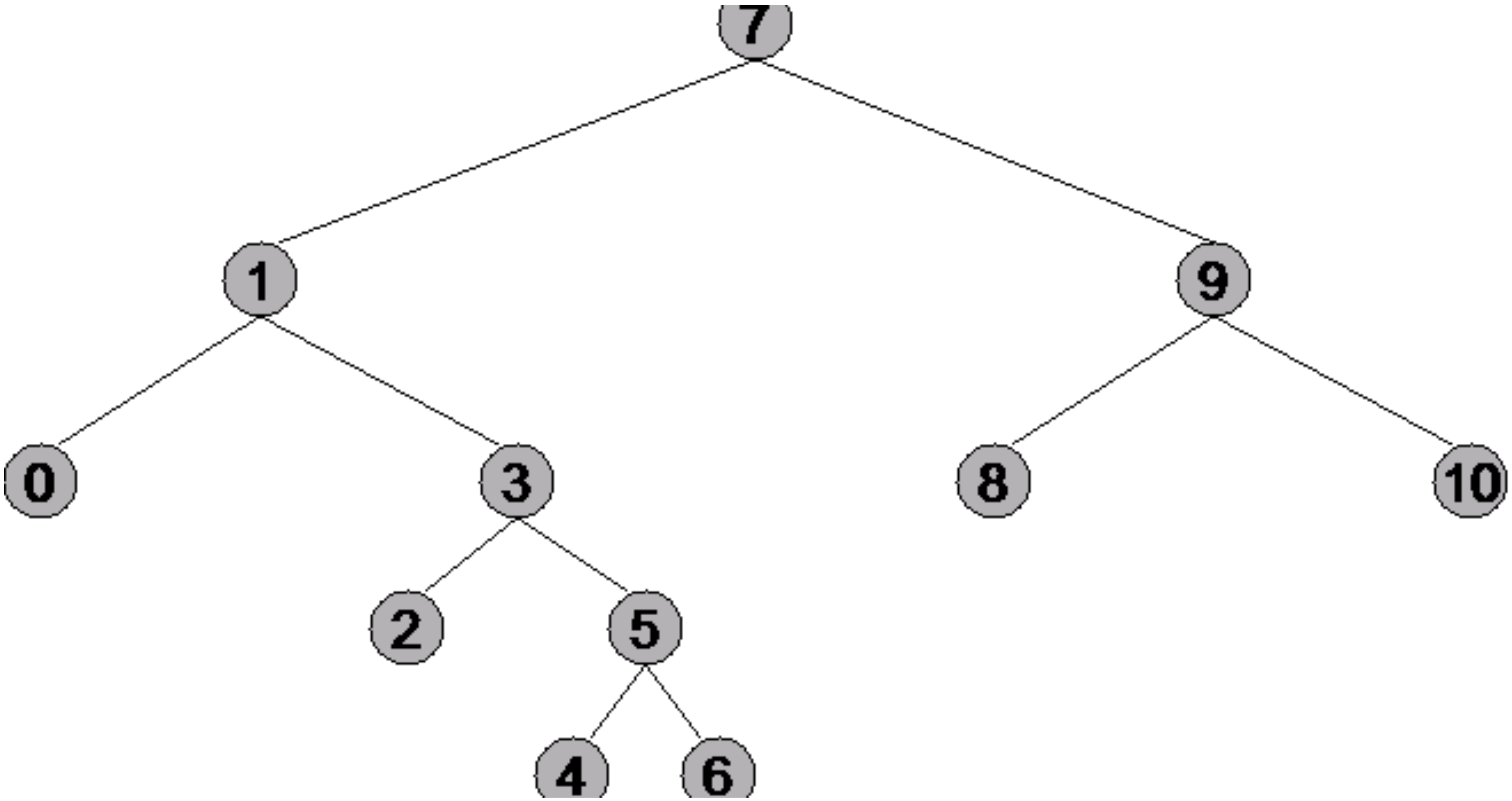
Tree Traversals

- "Walking" over whole tree is a tree **traversal**
  - Done often enough that there are standard names
  - Previous example: inorder traversal
    - Process left subtree
    - Process node
    - Process right subtree
- Note: Can do other processing besides printing

Other standard kinds of traversals

- Preorder traversal
  - Process node
  - Process left subtree
  - Process right subtree
- Postorder traversal
  - Process left subtree
  - Process right subtree
  - Process node
- Level-order traversal
  - Not recursive uses a queue

Tree Traversals



Pre-order: 7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10

Post-order: 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

Some Useful Methods

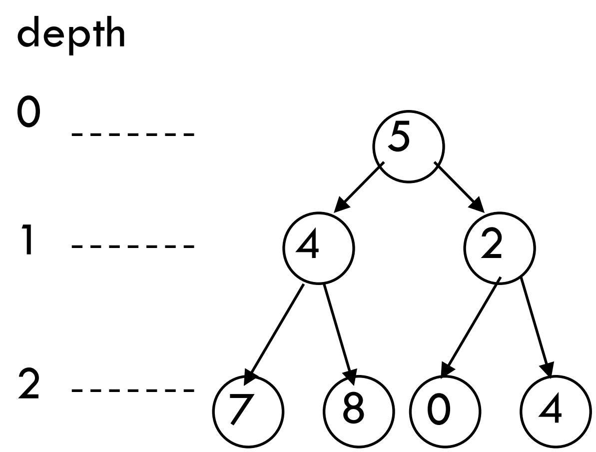

```
/* Return true iff node t is a leaf */
template <class T>
bool isLeaf(TreeNode<T>* t){
  return t!= nullptr && t->getLeft() == nullptr && t->getRight() == nullptr;
}

/* Return height of node t using postorder traversal
template <class T>
int height(TreeNode<T>* t){
  if(t == nullptr) return -1; // empty tree
  if(isLeaf(t)) return 0;
  return 1 + std::max(height(t->getLeft()),height(t->getRight()));
}

/* Return number of nodes  in t using postorder traversal */
template <class T>
int nNodes(TreeNode<T>* t){
  if(t == nullptr)return 0;
  return 1 + nNodes(t->getLeft()) + nNodes(t->getRight());
}
```

Useful Facts about Binary Trees

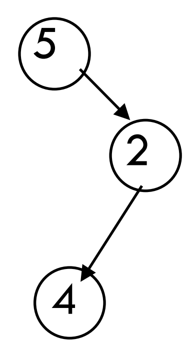
Max number of nodes at depth d: 2^d

If height of tree is h
- min number of nodes in tree: $h + 1$
- Max number of nodes in tree:
- $2^0 + \ldots + 2^h = 2^{h+1} - 1$

Complete binary tree
- All levels of tree down to a certain depth are completely filled



Height 2,
maximum number of nodes

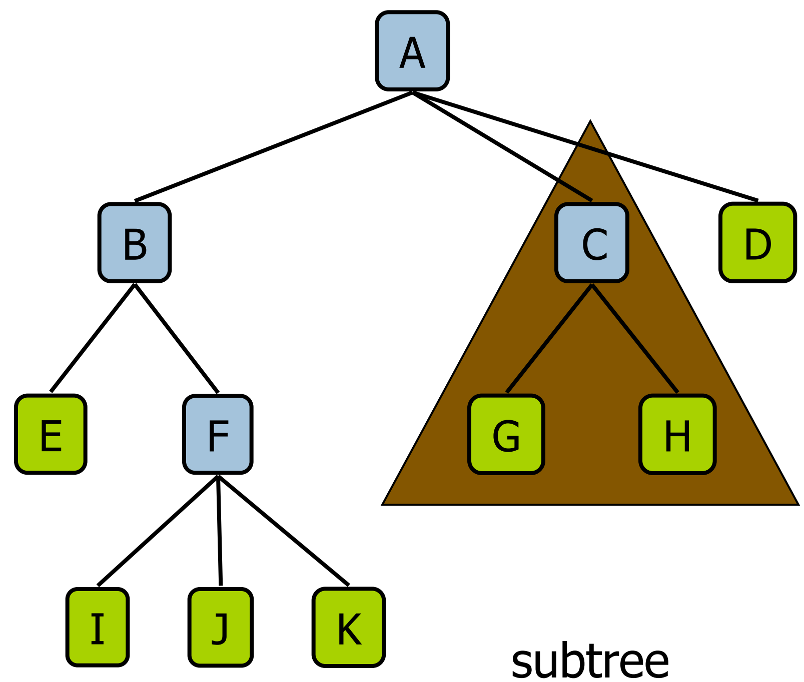


Height 2,
minimum number of nodes

Tree Terminology

- **Root**: node without parent (A)
- **Siblings**: nodes share the same parent
- **Internal node**: node with at least one child (A, B, C, F)
- **External node** (leaf): node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node: parent, grandparent, grand-grandparent, etc.
- **Descendant** of a node: child, grandchild, grand-grandchild, etc.
- **Depth** of a node: number of ancestors
- **Height** of a tree: maximum depth of any node (3)
- **Degree** of a node: the number of its children
- **Degree** of a tree: the maximum degree of its node.

- **Subtree**: tree consisting of a node and its descendants



Balanced Tree

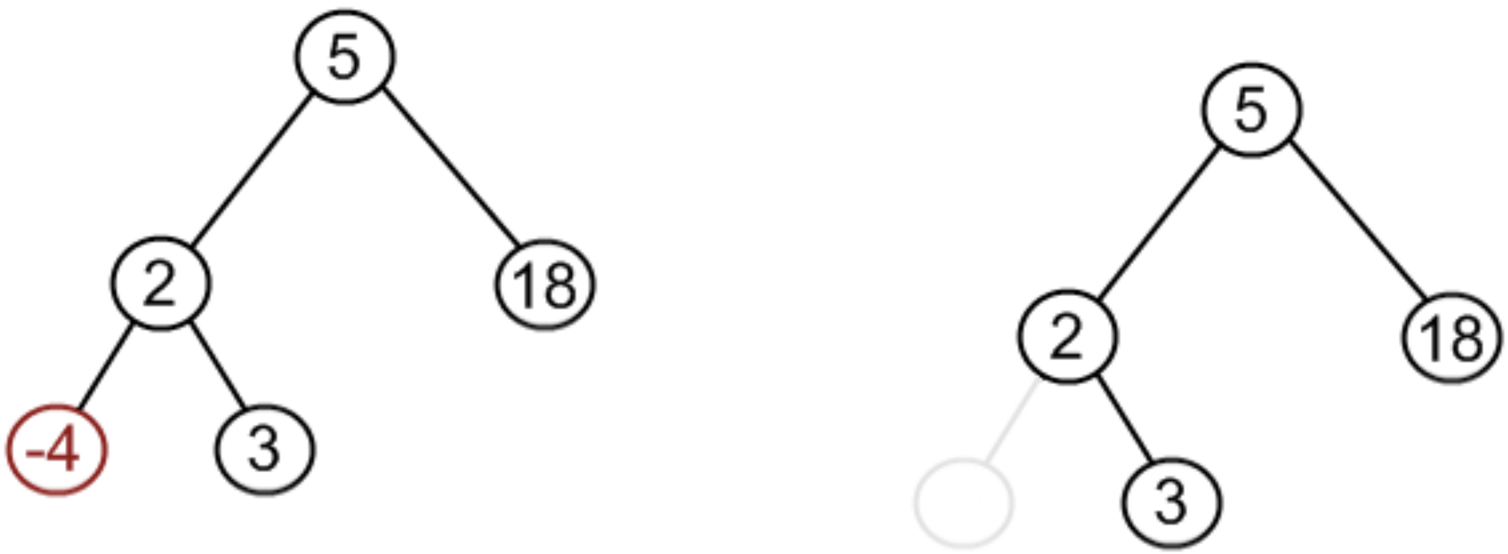
- A binary tree is balanced if for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ by at most 1 (Height-balancedness)
- A binary tree is balanced if for any two leaves the difference of the depth is at most 1 (Weight -balancedness)
  - Self-balancing BST
    - AVL Tree
    - B+ Tree

Things to Think About

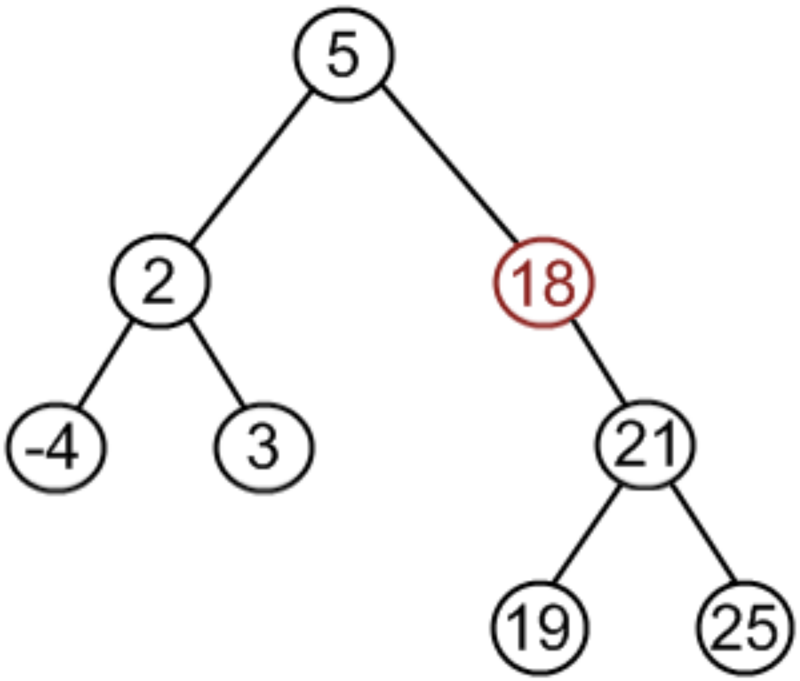
A BST works great as long as it's *balanced*

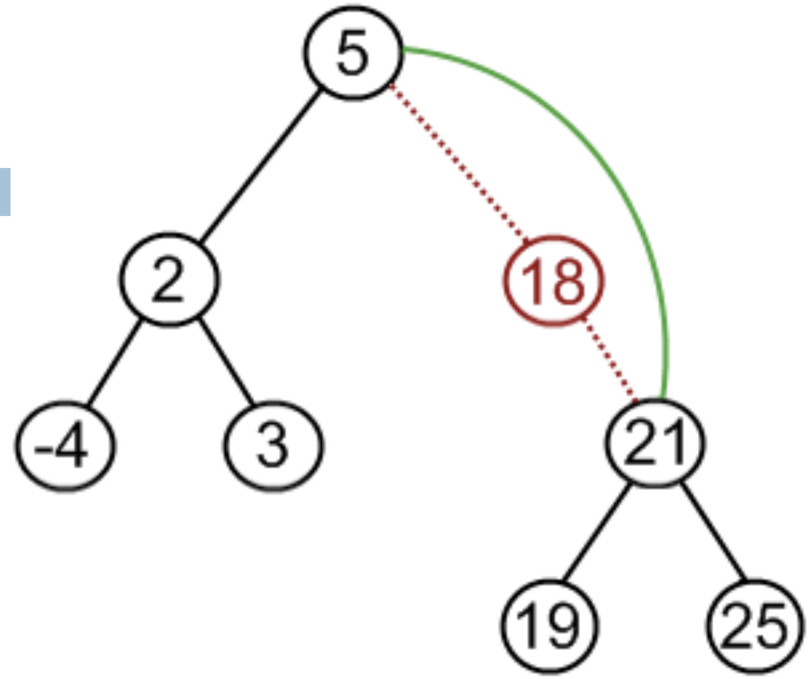
How can we keep it balanced? *This turns out to be hard enough to motivate us to create other kinds of trees*

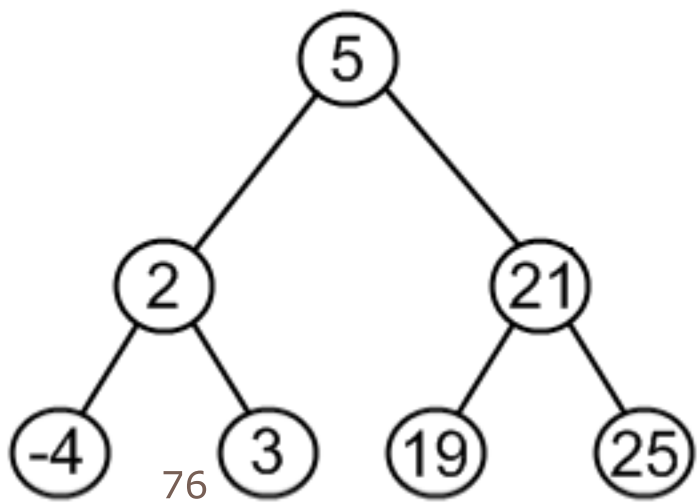
Remove a node



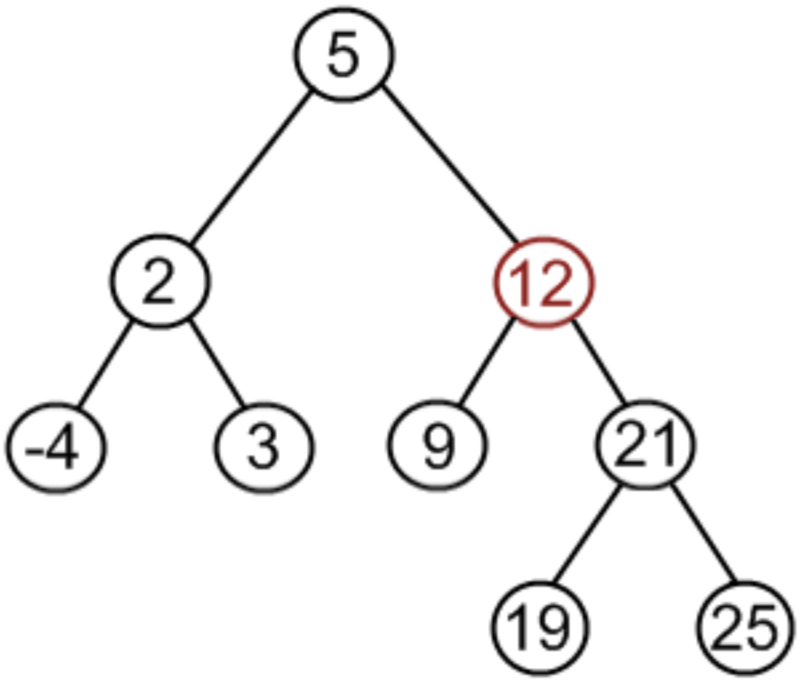
Remove a node





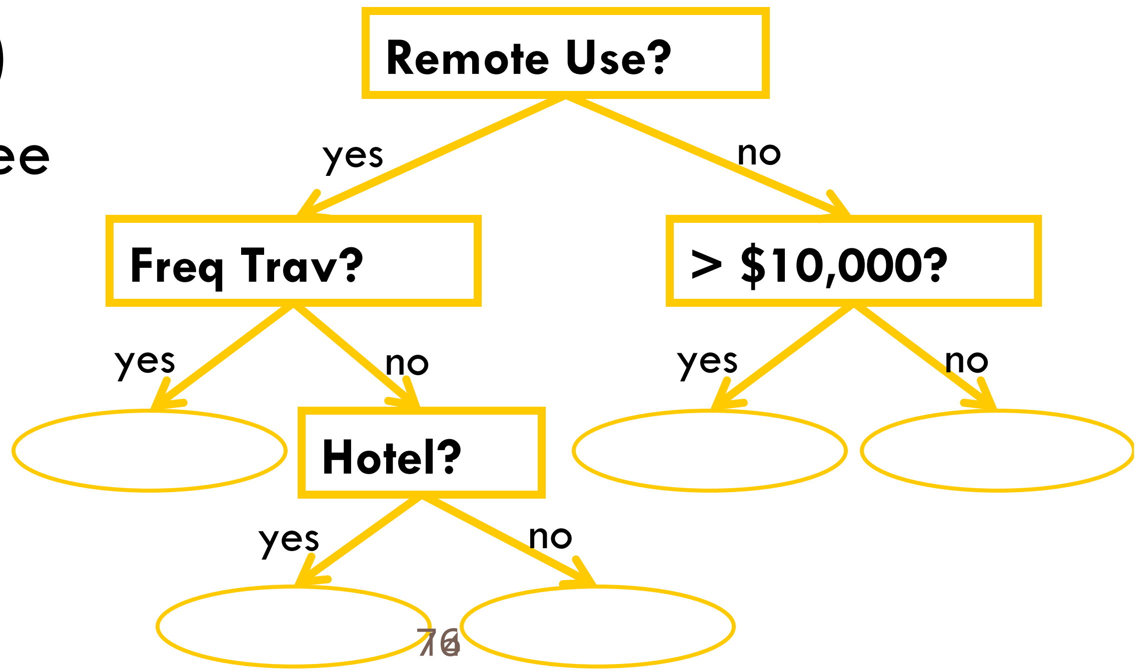


Remove a node



Decision Trees

- Classification:
  - Attributes (e.g. is CC used more than 200 miles from home?)
  - Values (e.g. yes/no)
  - Follow branch of tree based on value of attribute.
  - Leaves provide decision.
- Example:
  - Should credit card transaction be denied?



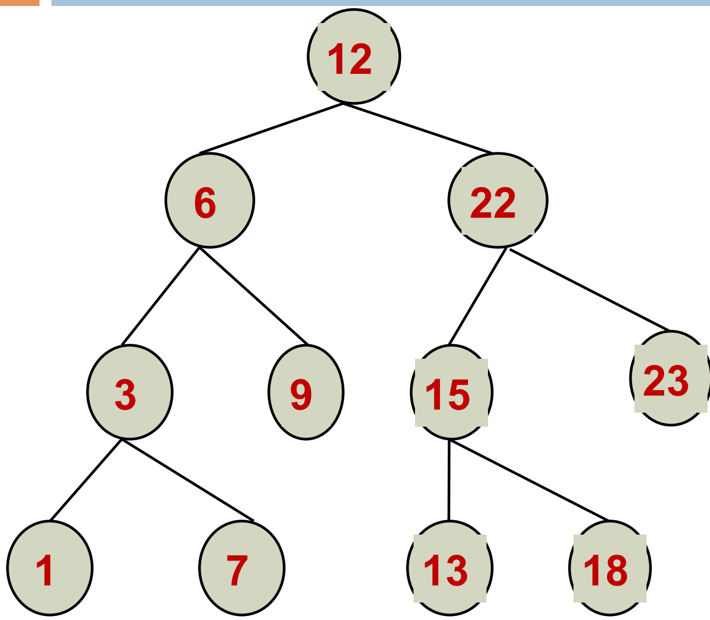
Tree Summary

- A *tree* is a recursive data structure
  - Each cell has 0 or more successors (*children*)
  - Each cell except the *root* has at exactly one predecessor (*parent*)
  - All cells are reachable from the *root*
  - A cell with no children is called a *leaf*
- Special case: *binary tree*
  - Binary tree cells have a left and a right child
  - Either or both children can be null
- Trees are useful for exposing the recursive structure of natural language and computer programs

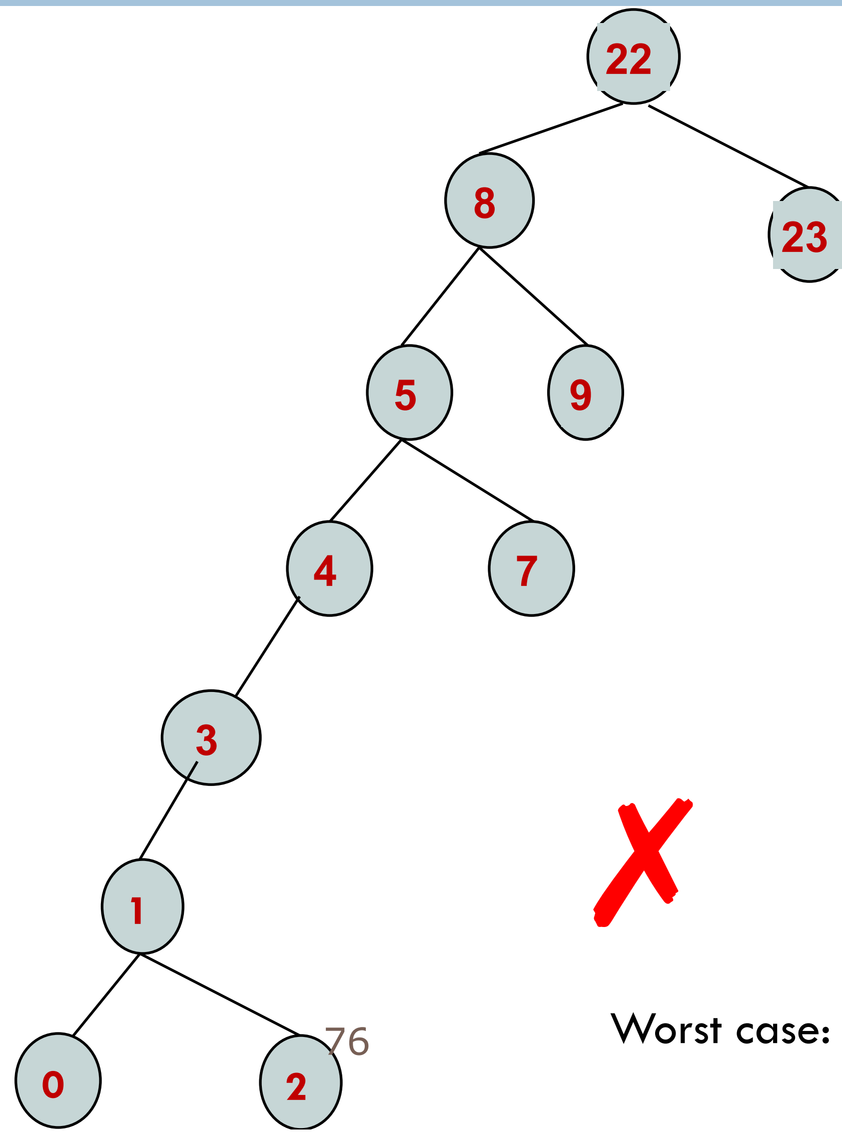
Outline

- Binary Search Tree
- AVL Tree

Tree has a problem



Worst case: O(log n)



Worst case: O(n)

What do we want?

	Lookup	Insertion	Deletion
Worst case	O(log n)	O(log n)	O(log n)
Average case	O(log n)	O(log n)	O(log n)

AVL tree

- An AVL tree is a binary search tree in which
  - for *every* node in the tree, the height of the left and right subtrees differ by at most 1.

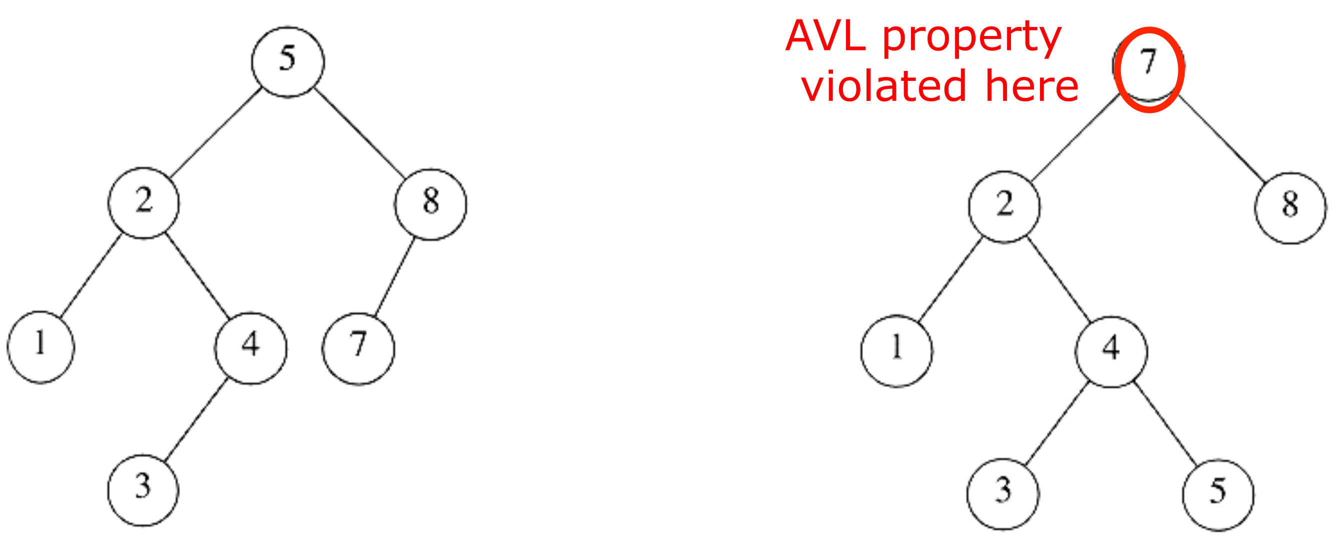
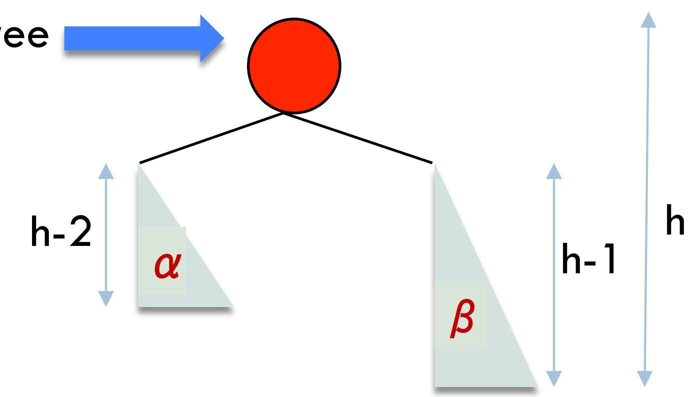


Figure 4.32 Two binary search trees. Only the left tree is AVL.

AVL tree

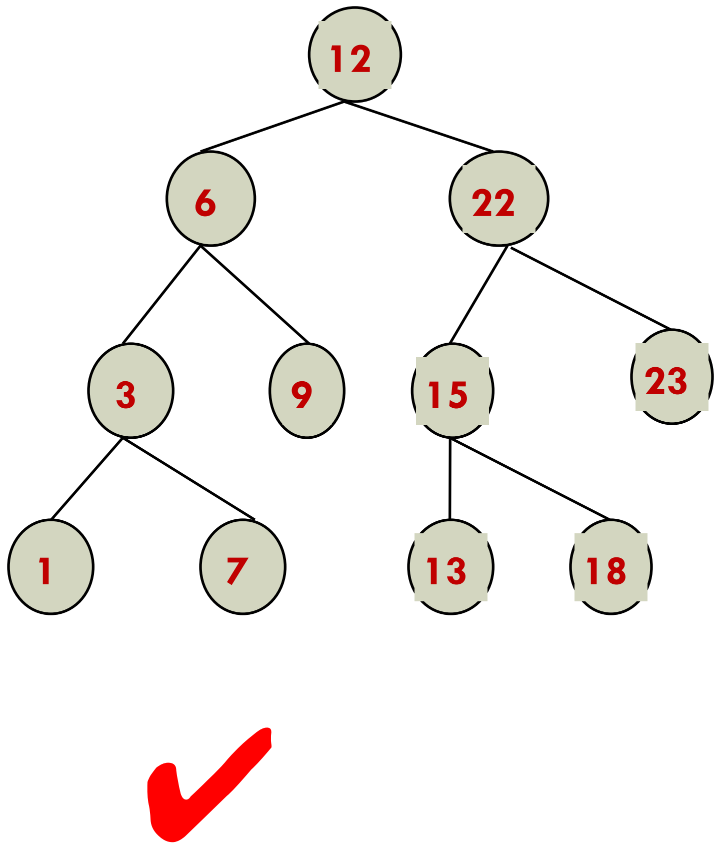
- Invented by ***Georgy Adelson-Velsky*** and ***Evgenii Landis*** (**AVL**) in 1962
- It is a self-balancing binary search tree
- Lookup, insertion, and deletion can be done in **O(log n)** under both average and worst cases. n is the number of nodes in the tree
- The heights of two child subtrees of ***any*** given node diff by at most one

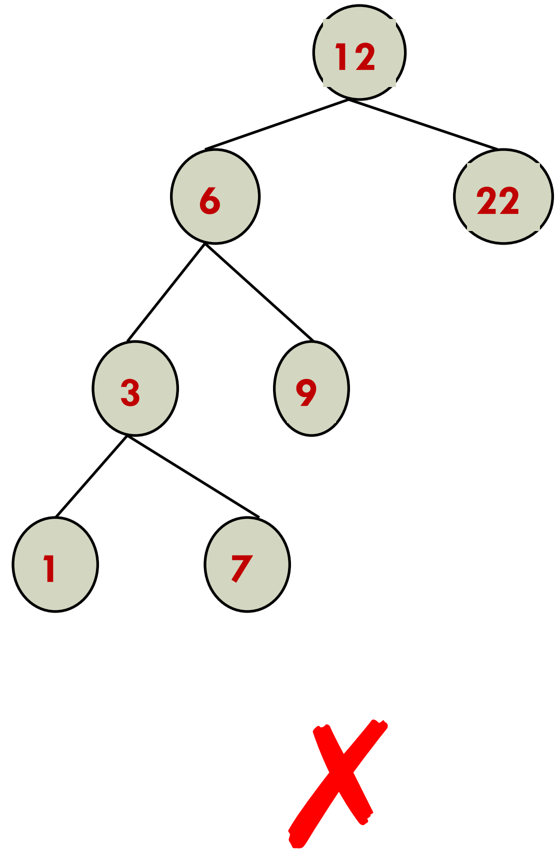
Any node of AVL tree



AVL tree

- The heights of two child subtrees of **any** given node diff by at most one





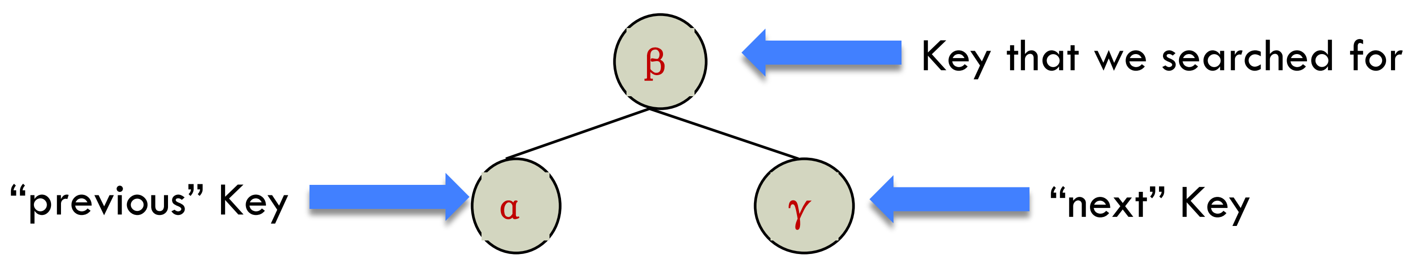
Searching/Lookup is trivial

- AVL is a special binary search tree

- Searching a key in an AVL tree is the same way as that of a normal binary search tree

Traversal

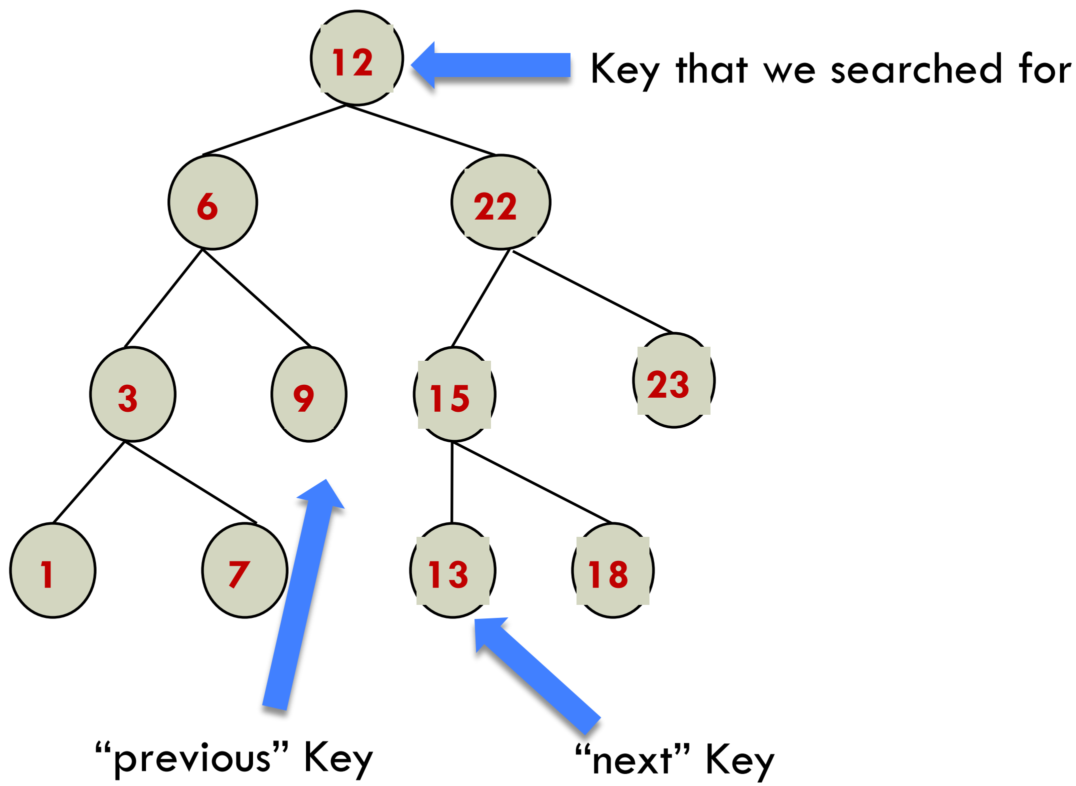
- Suppose that you already found the key in the AVL tree
- You want to find the “previous” or “next” key value in the AVL tree



O(1)

Traversal

- Suppose that you already found the key in the AVL tree
- You want to find the “previous” or “next” key value in the AVL tree



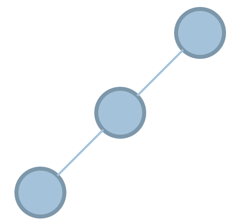
worst-case time complexity: O(log n)

Insert and Rotation in AVL Trees

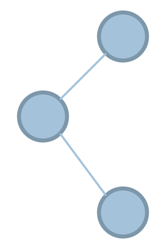
- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Insertion in an AVL tree

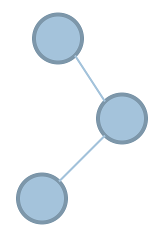
- Let us call the node that must be rebalanced α
  - Since any node has at most 2 children, and a height imbalance requires that α's 2 subtrees' height differ by 2, there are 4 violation cases:



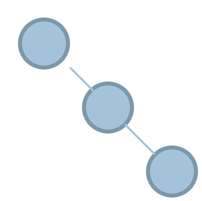
1. An insertion into the left subtree of the left child of α.



2. An insertion into the right subtree of the left child of α.



3. An insertion into the left subtree of the right child of α.



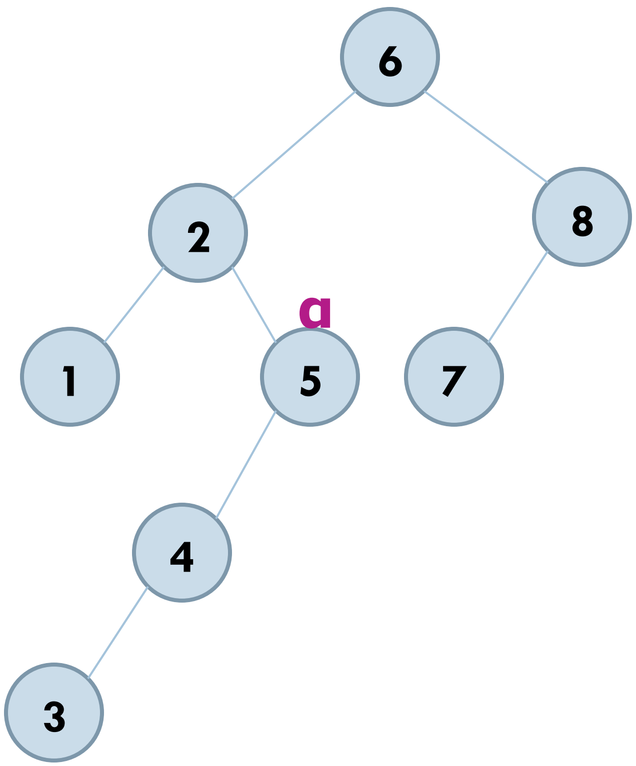
4. An insertion into the right subtree of the right child of α.

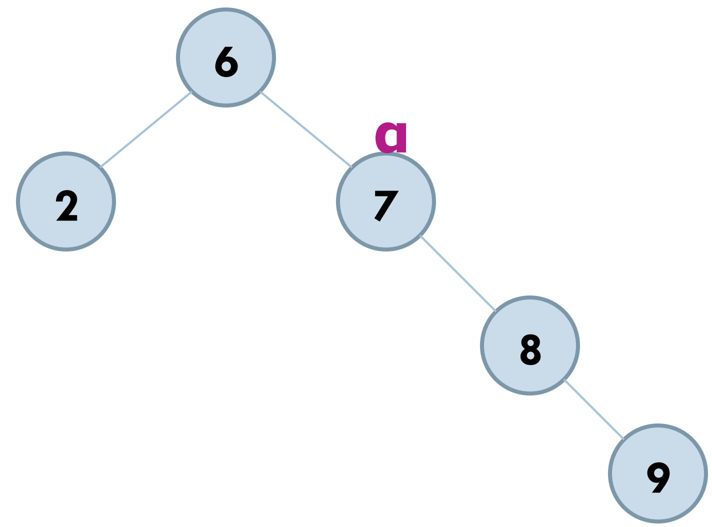
Insertion in an AVL tree

- Outside cases (left-left or right-right), fixed by a **single rotation**:

  (1) An insertion into the left subtree of the left child of α.

  (4) An insertion into the right subtree of the right child of α.



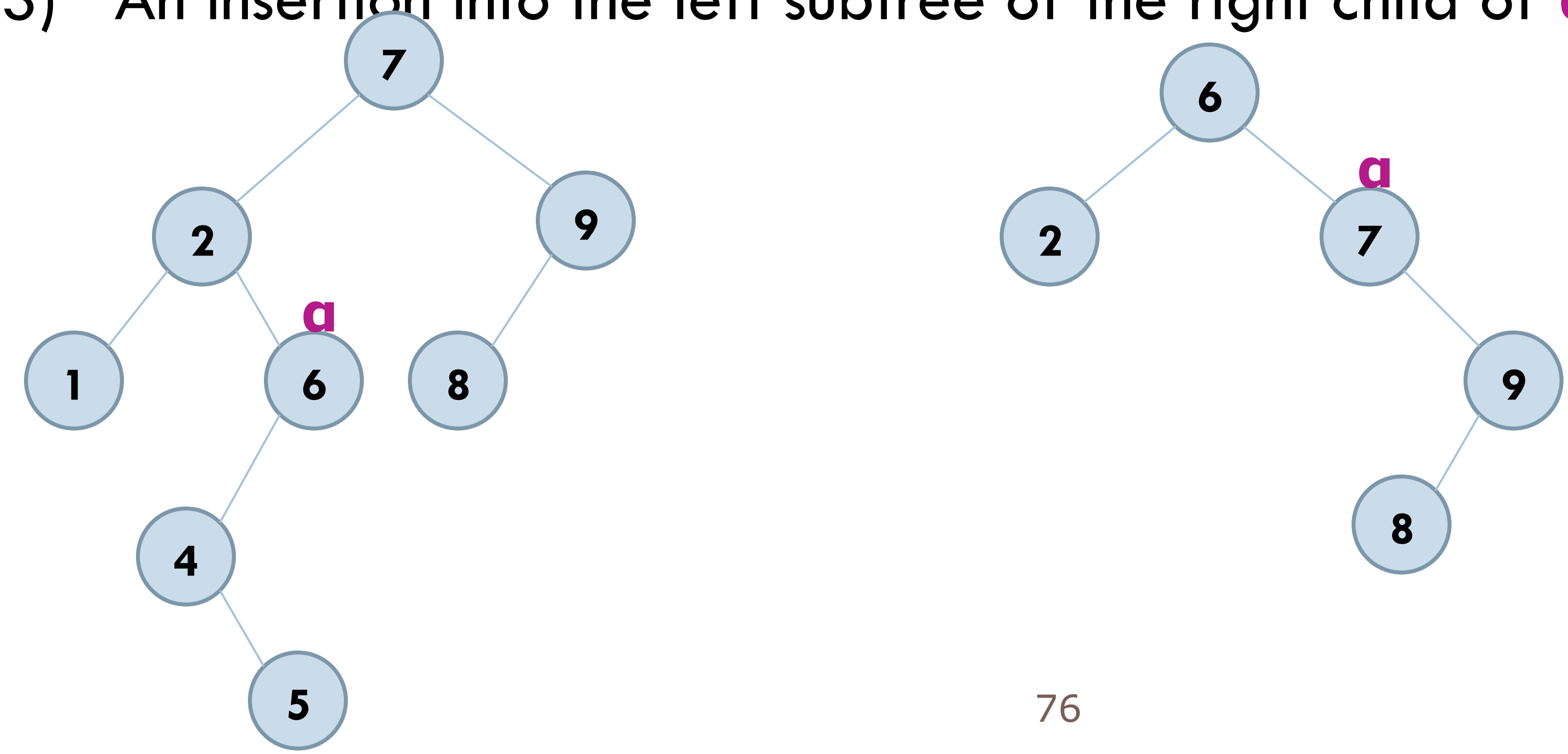


Insertion in an AVL tree

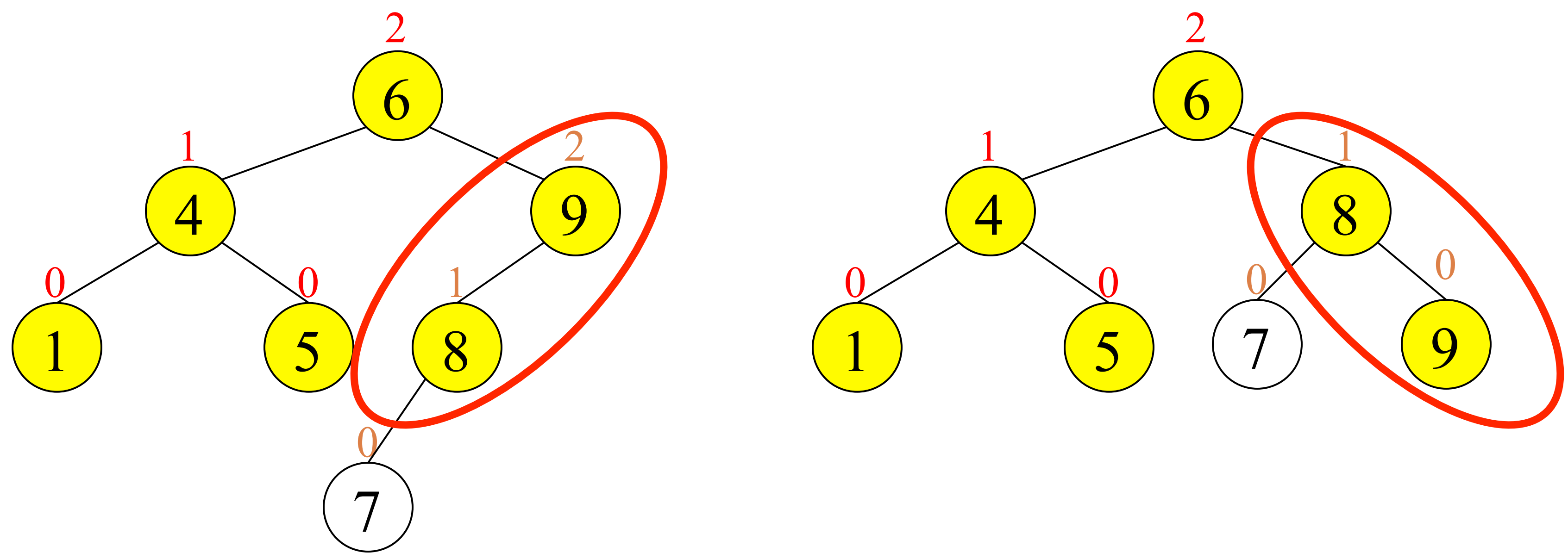
- Inside cases (right-left or left-right), fixed by a **double rotation**:

(2) An insertion into the right subtree of the left child of α.

(3) An insertion into the left subtree of the right child of α.



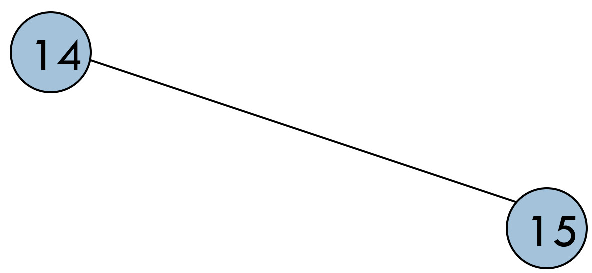
Single Rotation in an AVL Tree



AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- First insert 14 and 15:

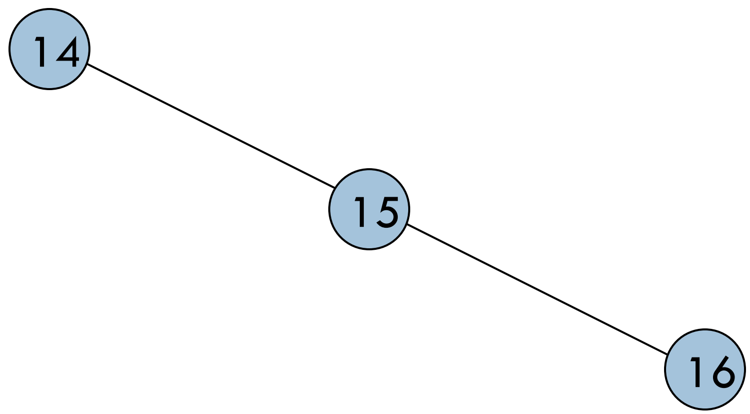


- Now insert 16.

AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- Inserting 16 causes AVL violation:

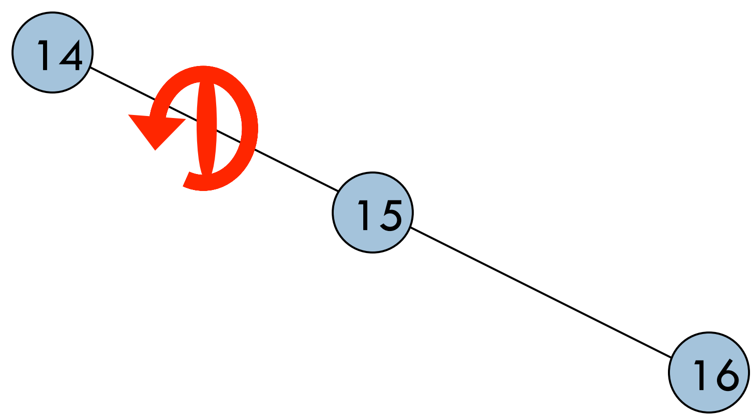


- Need to rotate.

AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

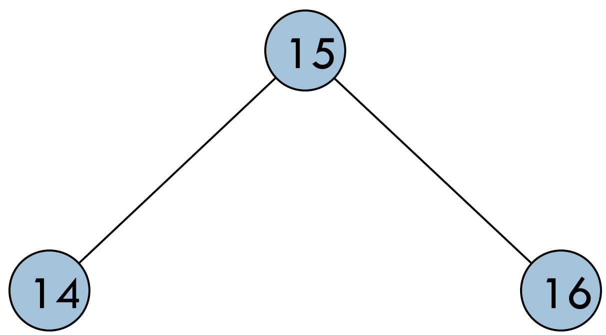
- Rotation type:



AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

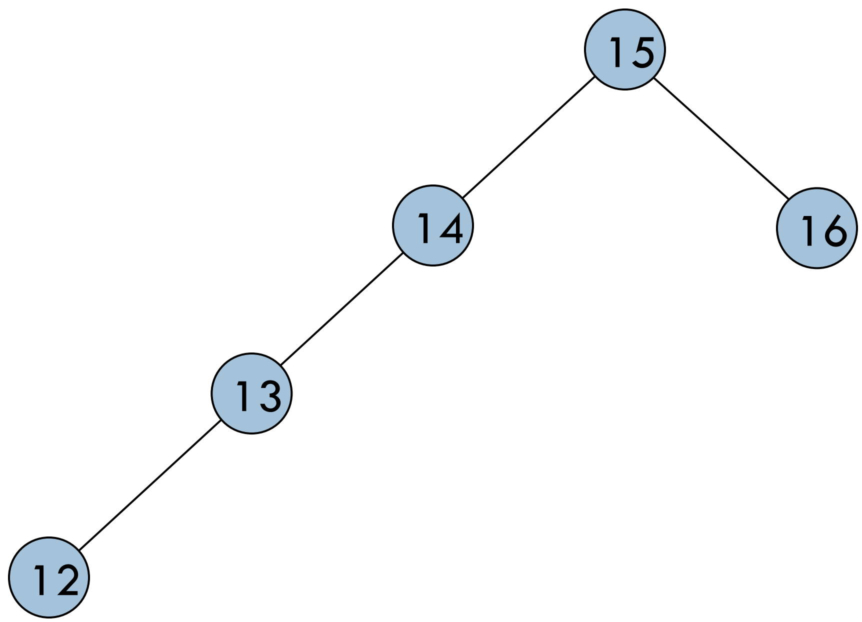
- Rotation restores AVL balance:



AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- Now insert 13 and 12:

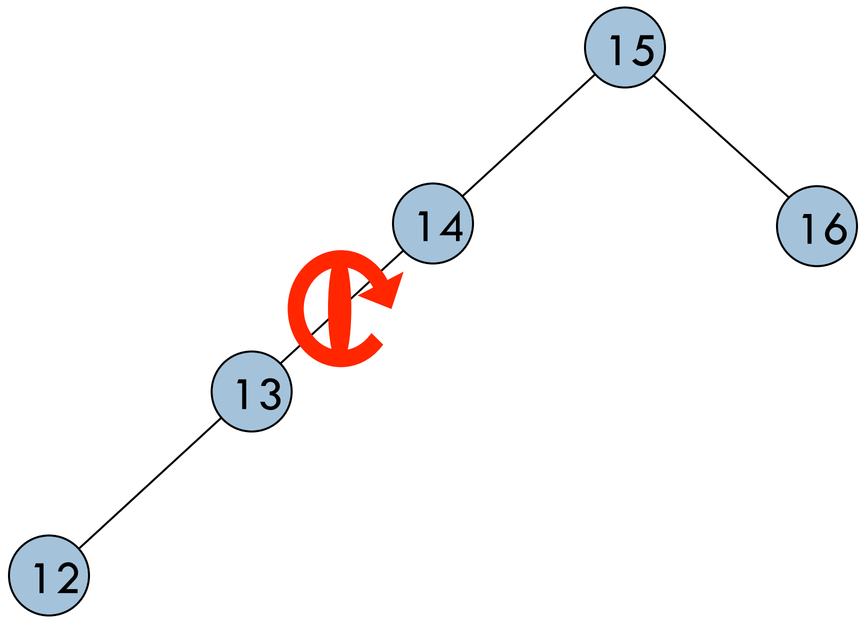


- AVL violation - need to rotate.

AVL Tree Rotations

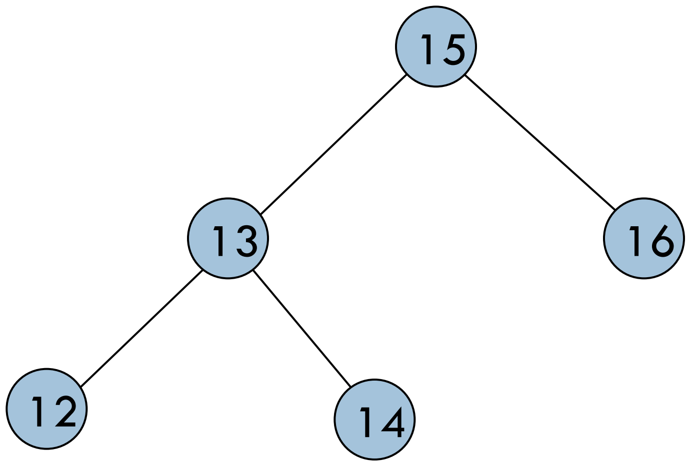
Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- Rotation type:



AVL Tree Rotations

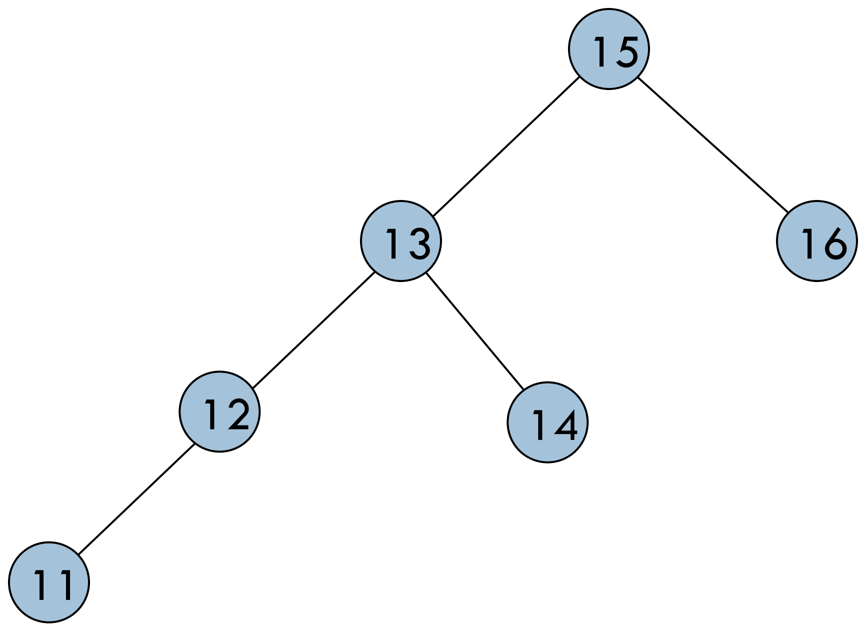
Single rotations: insert 14, 15, 16, 13, 12, 11, 10



- Now insert 11.

AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

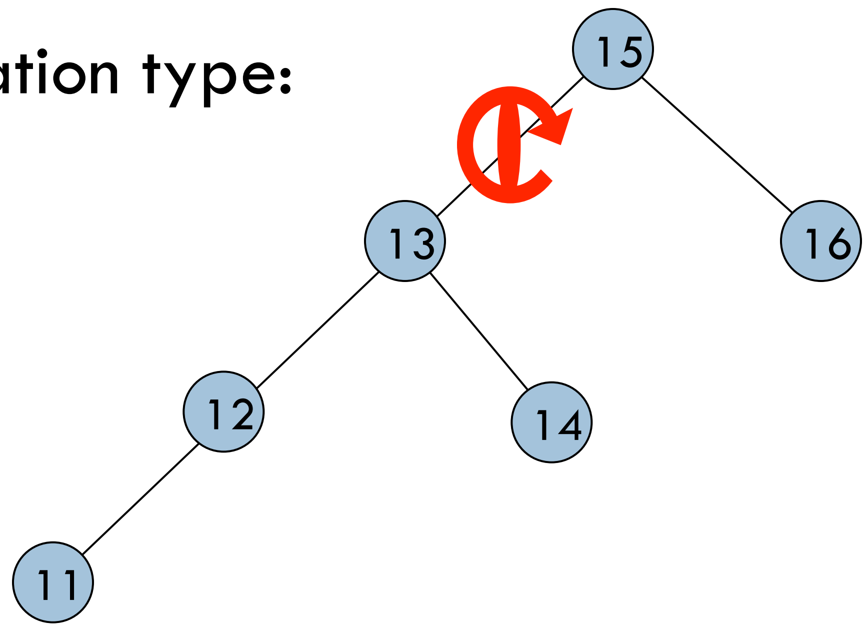


- AVL violation – need to rotate

AVL Tree Rotations

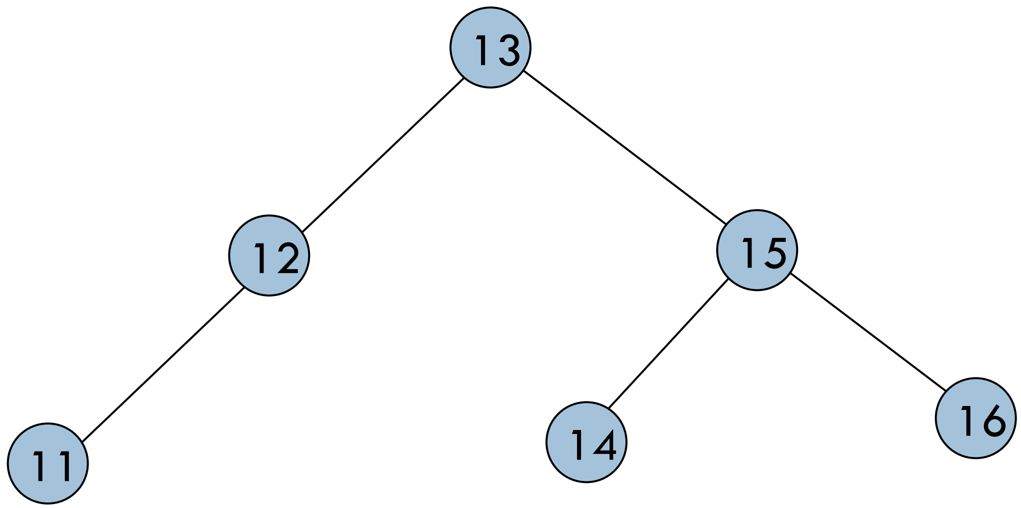
Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- Rotation type:



AVL Tree Rotations

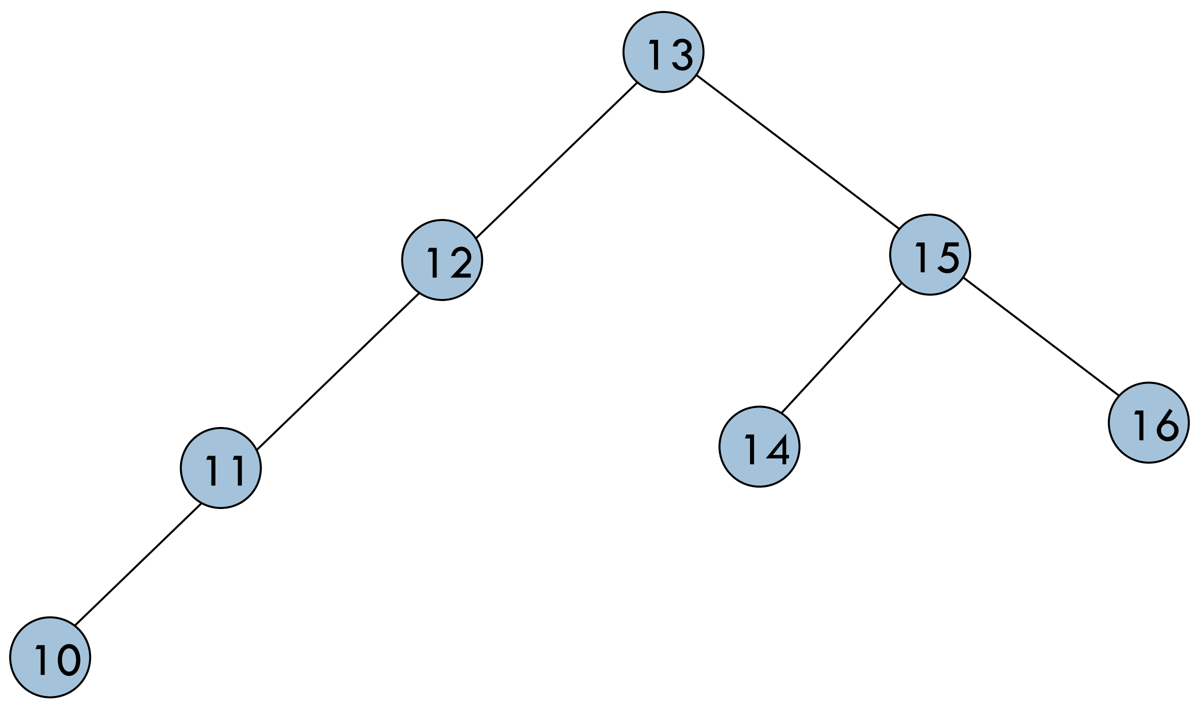
Single rotations: insert 14, 15, 16, 13, 12, 11, 10



- Now insert 10.

AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

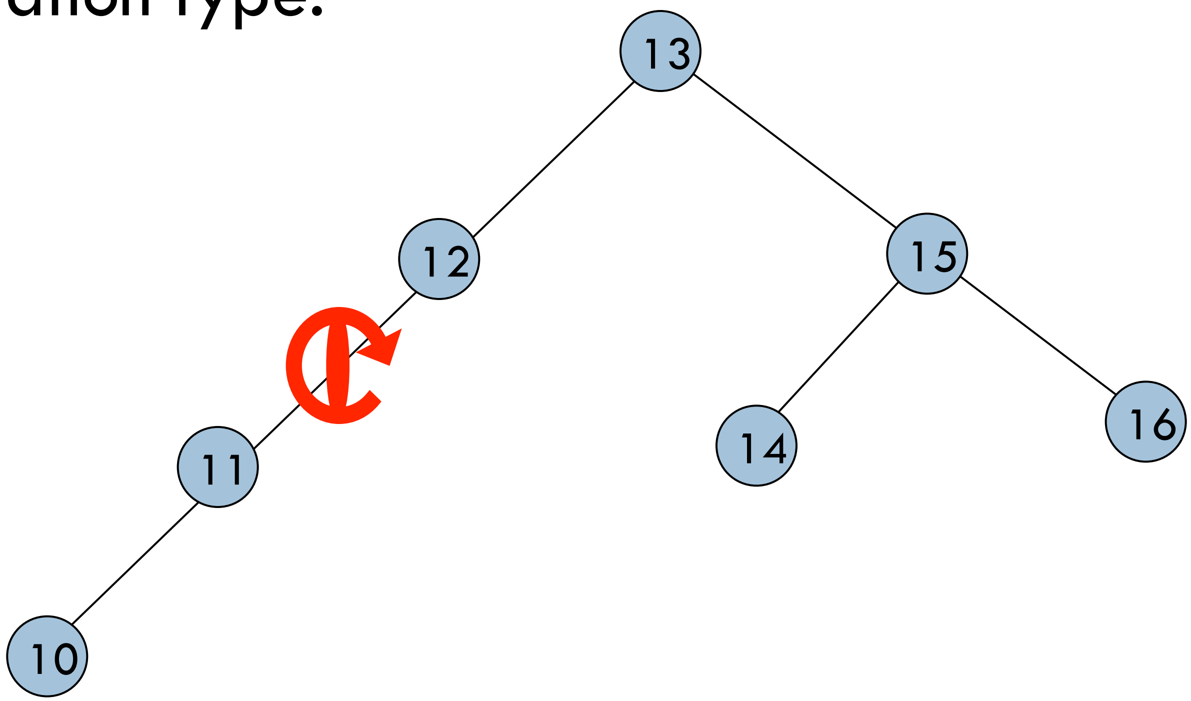


- AVL violation – need to rotate

AVL Tree Rotations

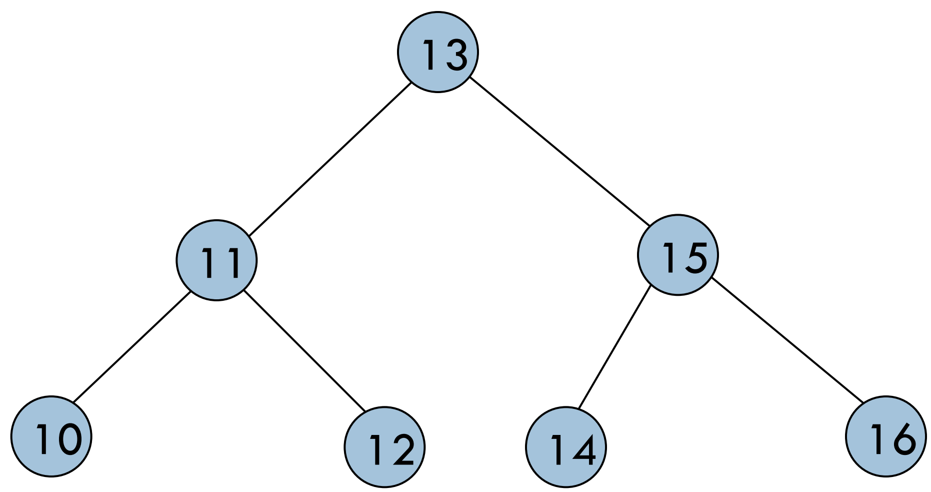
Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- Rotation type:



AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10



- AVL balance restored.

Overview of Insertion and Deletion

- It is performed as in binary search tree
- If the balance is destroyed, rotation(s) is performed to correct balance
- One rotation is sufficient for one insertion
- For one deletion, O(log n) rotations at most

AVL Insertion

- When insert a new node to the AVL, pretend that you are searching the node in the tree. Insert the new node to the place where searching falls off the tree

- After an insertion, only the nodes that on the path from the inserted node to the root can have altered balance

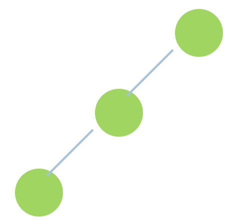
- If the inserted AVL tree is out of balance (given a node, the heights of its two subtrees differ by more than 1), rebalance the AVL (by rotation).

Rebalancing for insertion

Suppose the node to be rebalanced is **x**, there are four possible scenarios (two of them are symmetric to the other two)

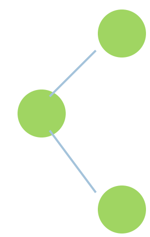
1. An insertion in the left subtree of the left child of X,

   Left left case



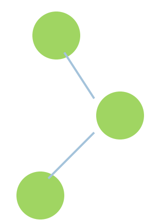
2. An insertion in the right subtree of the left child of X,

   Left right case



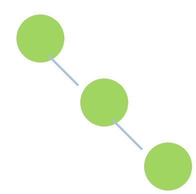
3. An insertion in the left subtree of the right child of X, or

   Right left case



4. An insertion in the right subtree of the right child of X.

   Right right case



How to rebalance? Rotation

- Case 1 (**left left case**) and case 4 (**right right case**) are symmetric and requires the same operation for balance.
  - Cases 1,4 are handled by *single rotation.*

- Case 2 (**left right case**) and case 3 (**right left case**) are symmetric and requires the same operation for balance.
  - Cases 2,3 are handled by *double rotation.*

Analysis

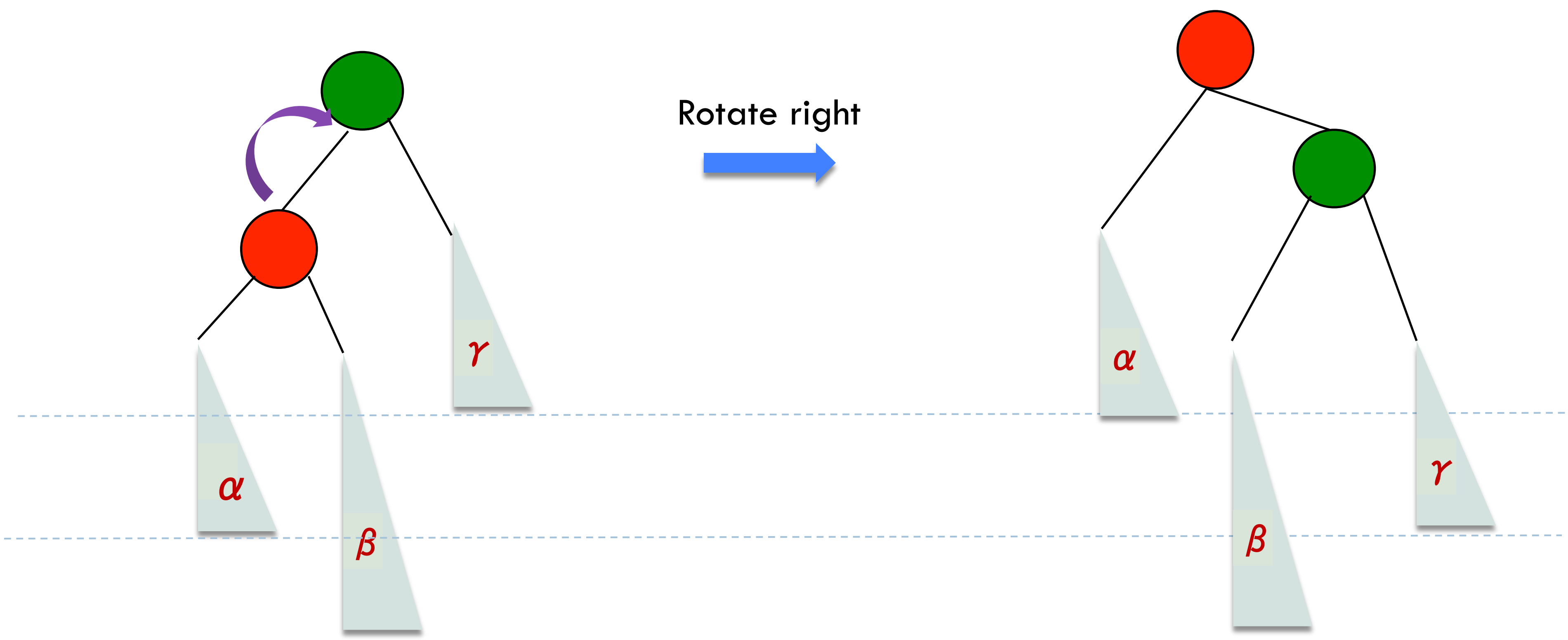
After one node insertion, if we have **case 1 or 4**.

- One **single rotation** can fix it
  - The new height of the rotated subtree is the same as that of the subtree before insertion
- When imbalance exists, find the first node from the inserted node to the root that it is out of balance
- Single rotation take time O(1)
- Insertation takes time O(log n)

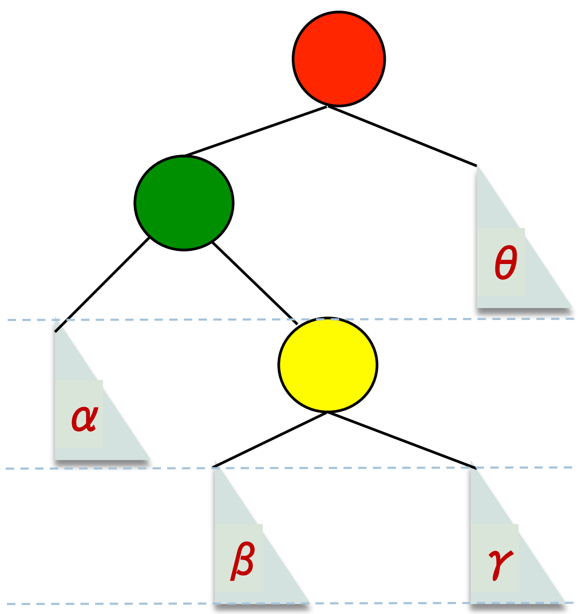
Case 2 and Case 3

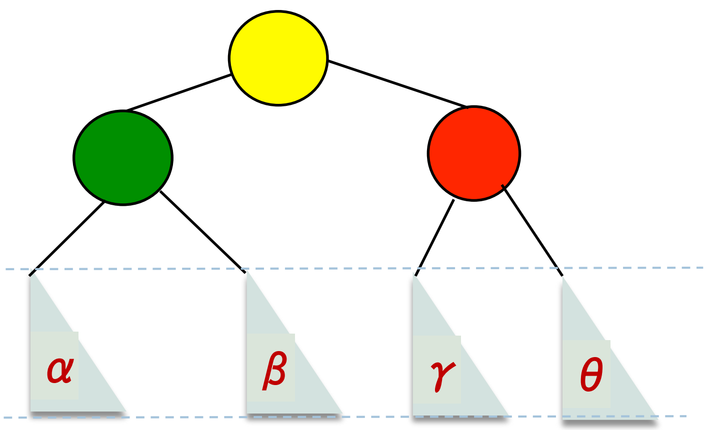
- Single rotation does not fix case 2 and case 3.
- They can be fixed by double rotation

Single Rotation does not work for case 2 and case 3



Double Rotation



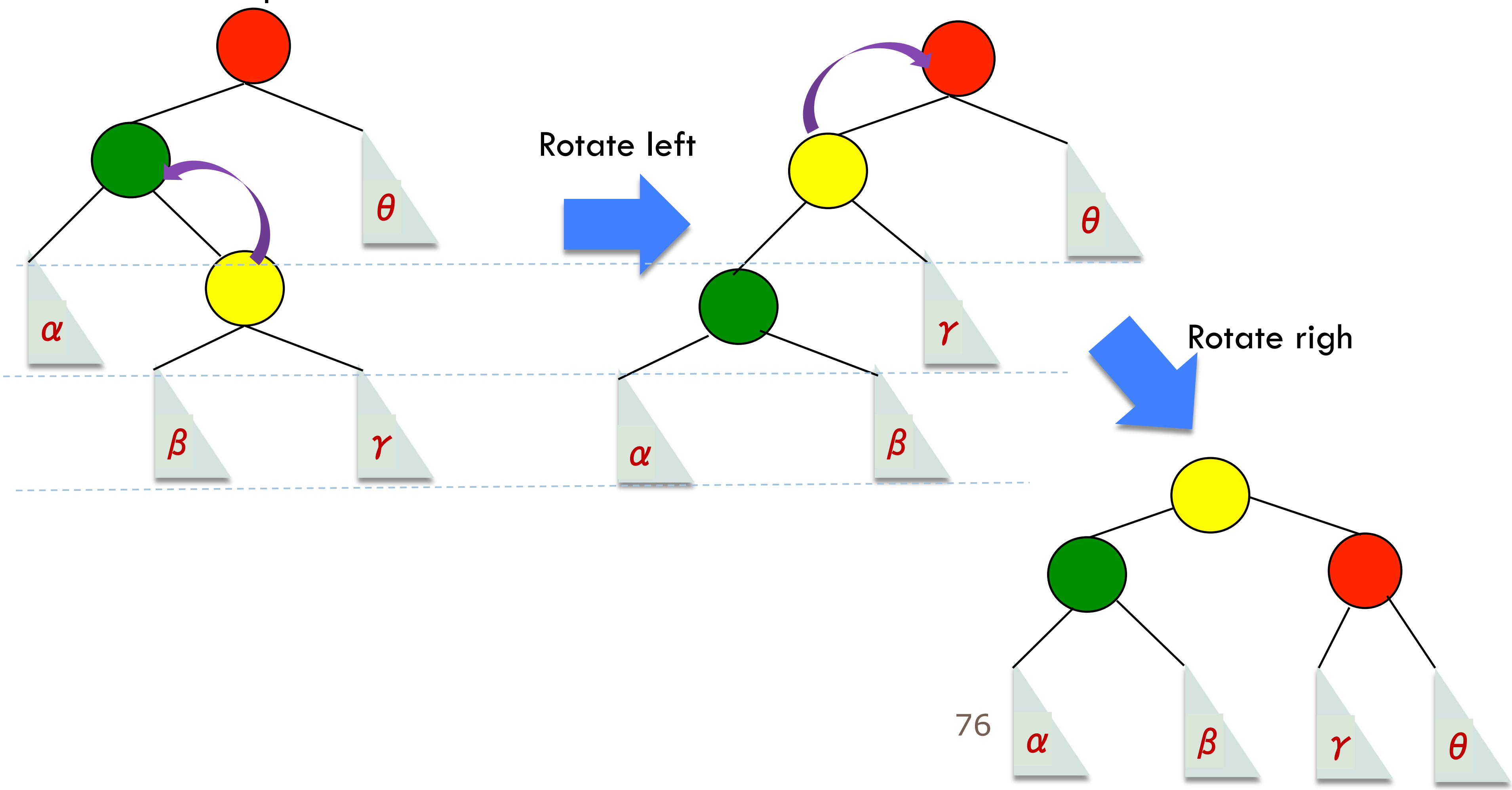


Left sub-tree is two levels deeper than the right sub-tree

Move yellow node up two levels and red node one level down

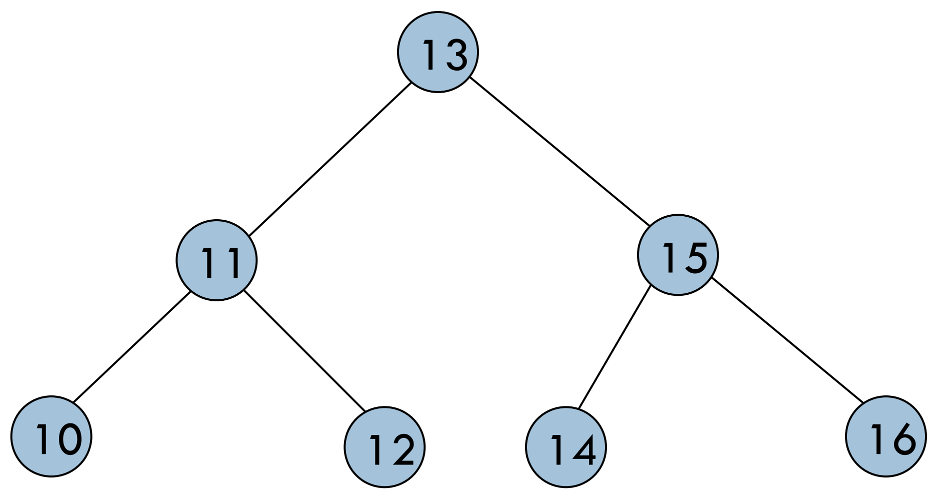
Double Rotation – in slow motion

- double rotation can be viewed as two single rotations
- Example: case 2



AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

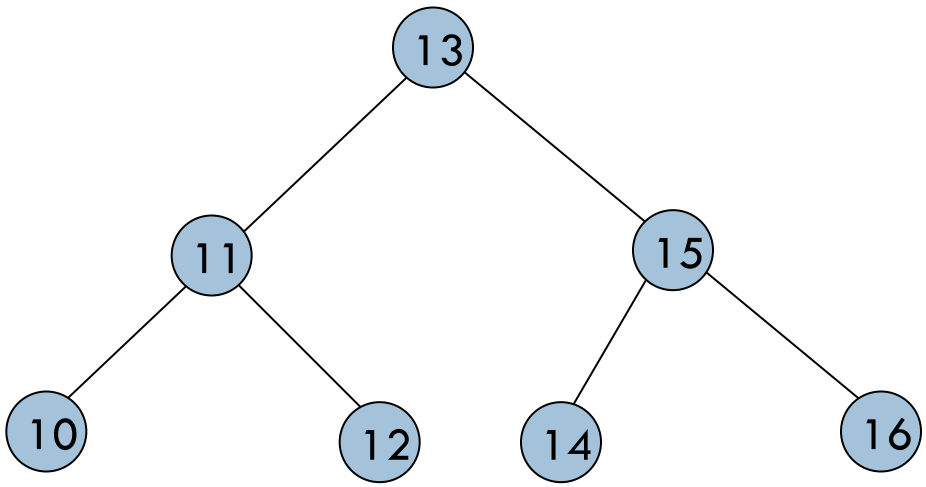


- AVL balance restored.

AVL Tree Rotations

Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

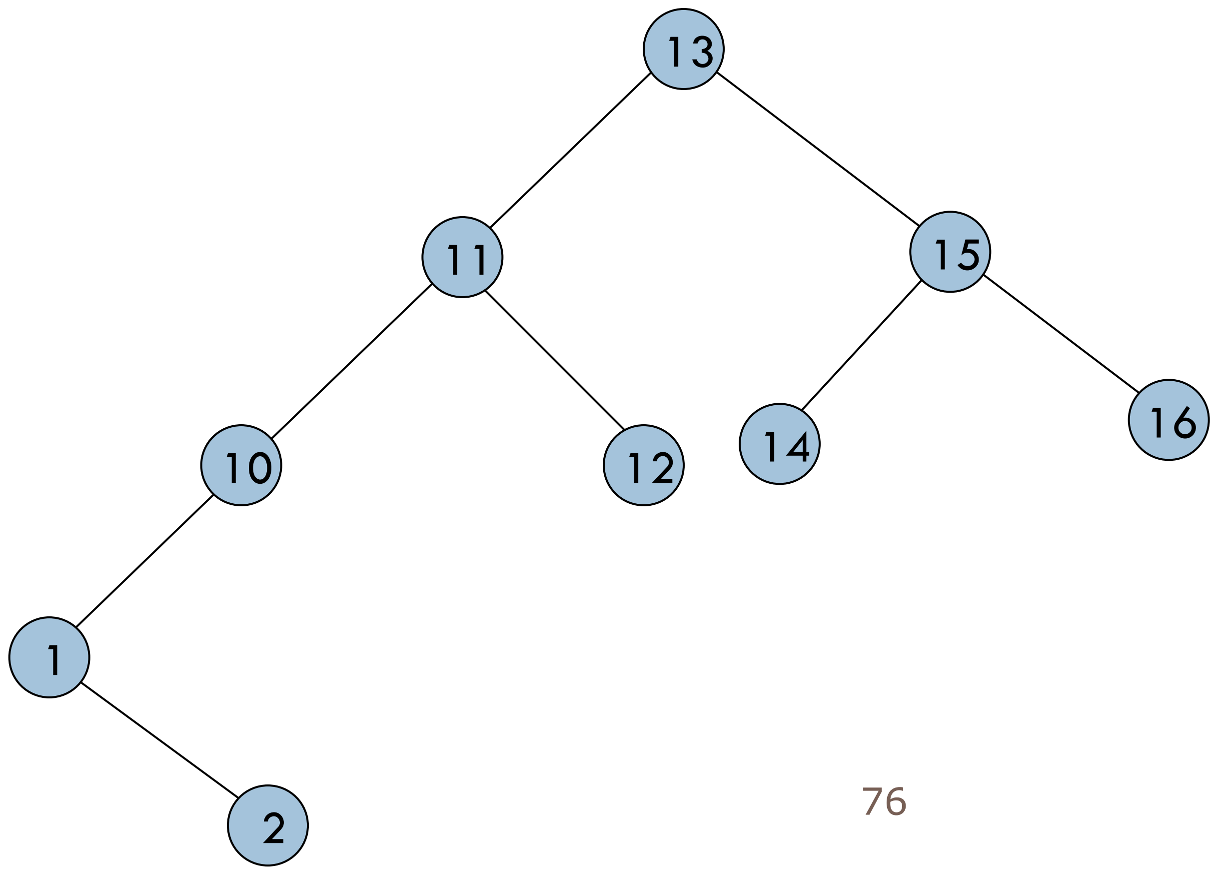
- First insert 1 and 2:



AVL Tree Rotations

Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

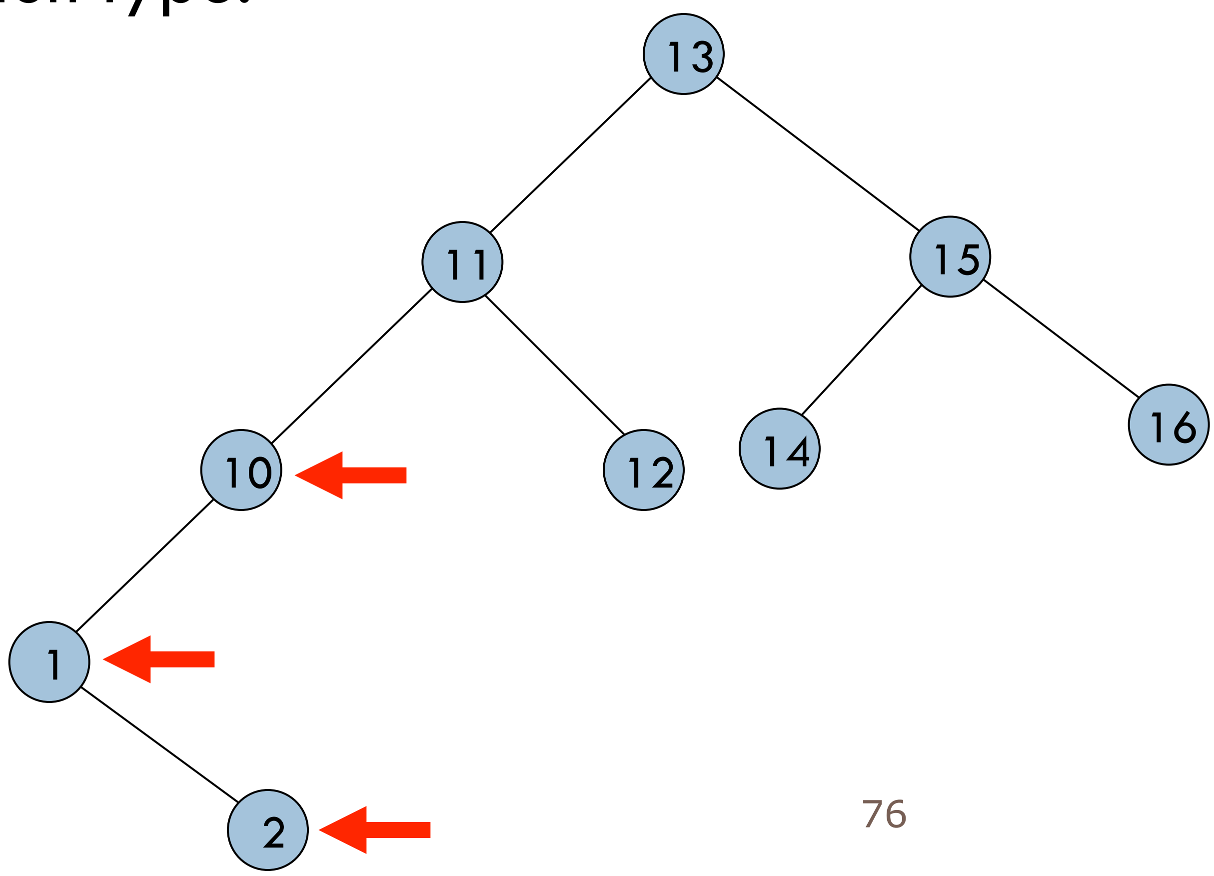
- AVL violation - rotate



AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

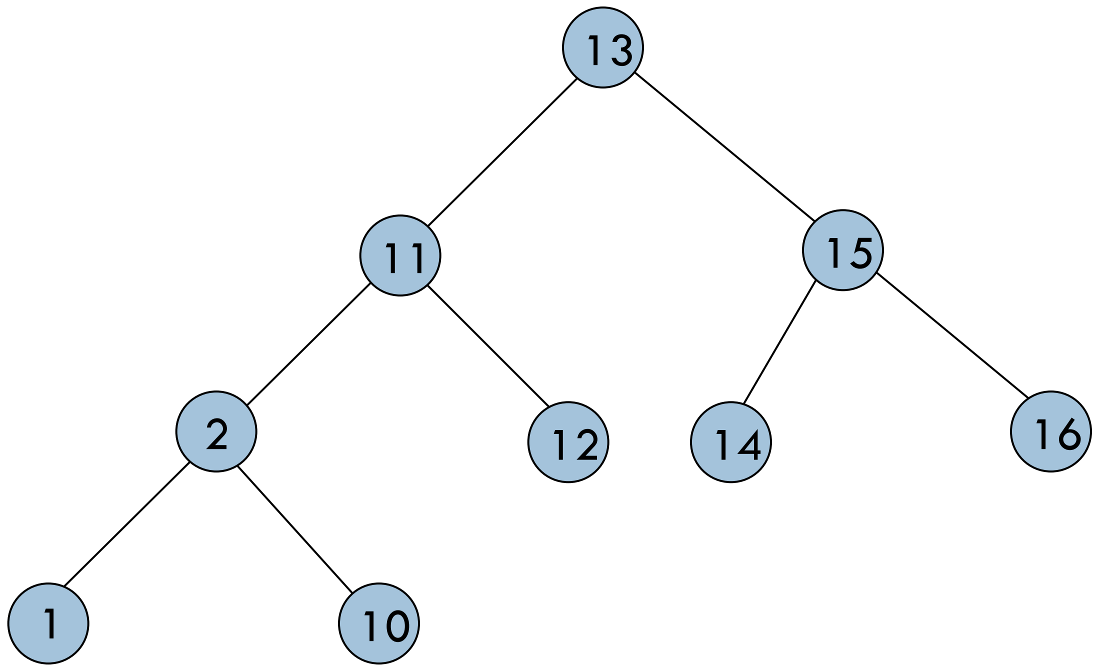
- Rotation type:



AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

- AVL balance restored:

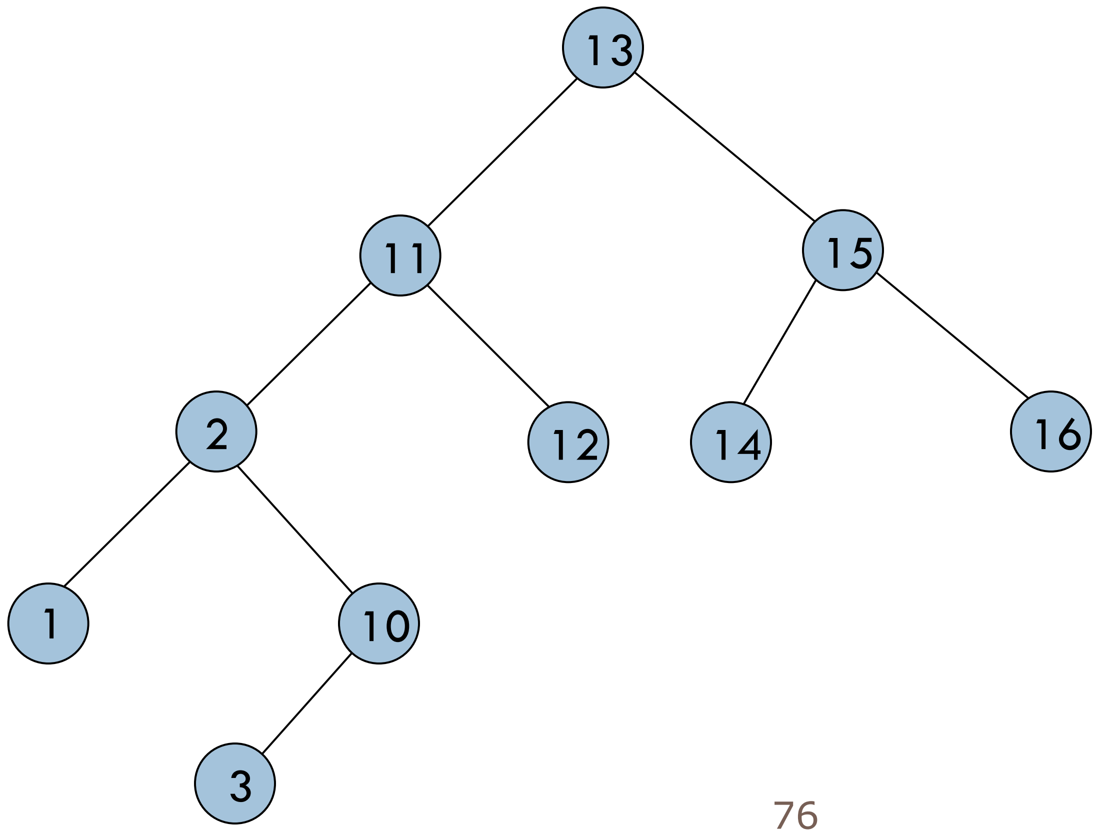


- Now insert 3.

AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

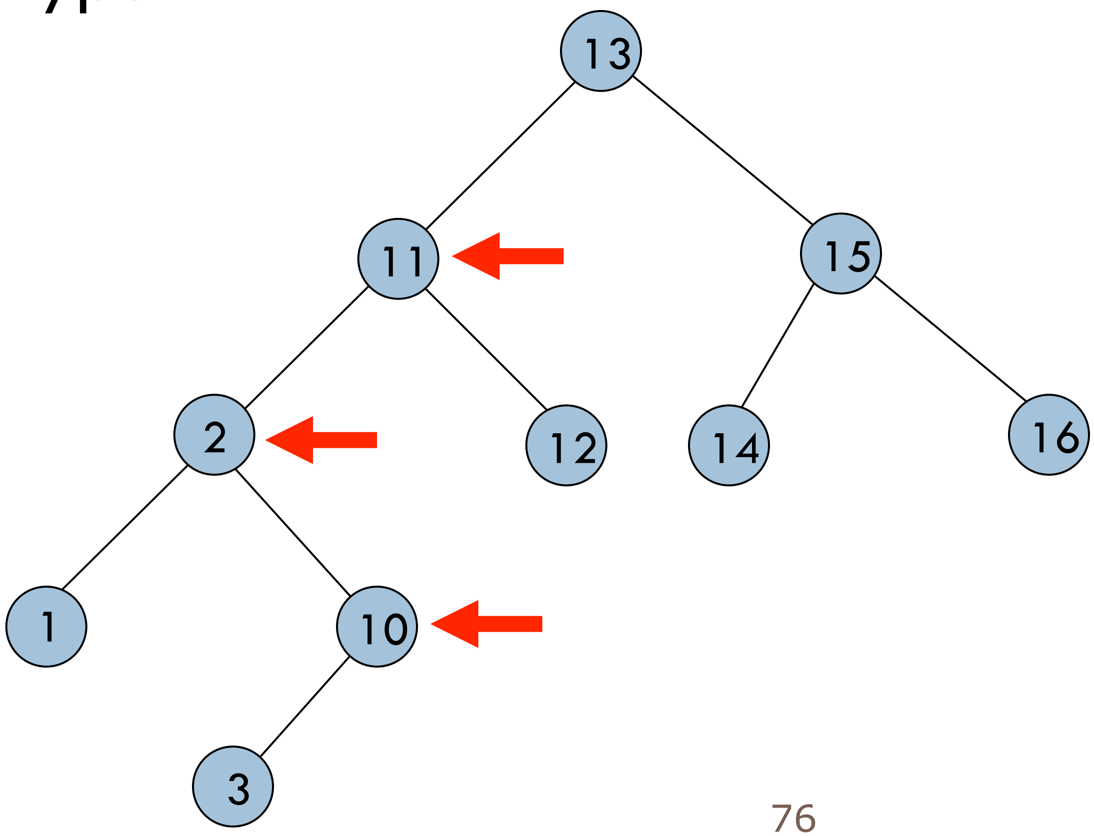
- AVL violation – rotate:



AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

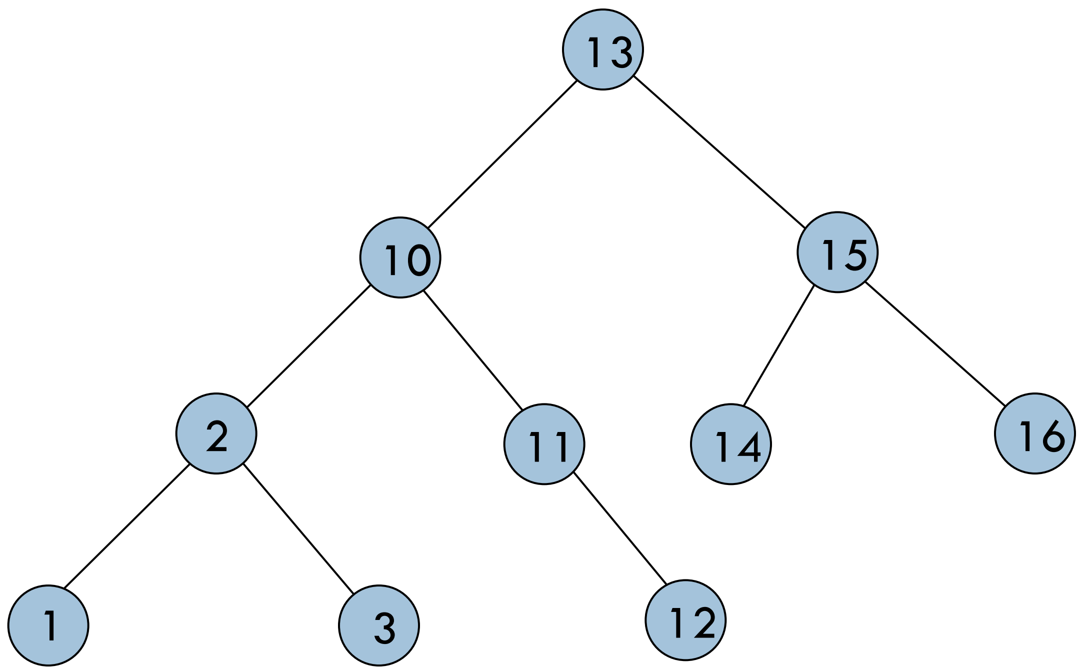
- Rotation type:



AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

• AVL balance restored:

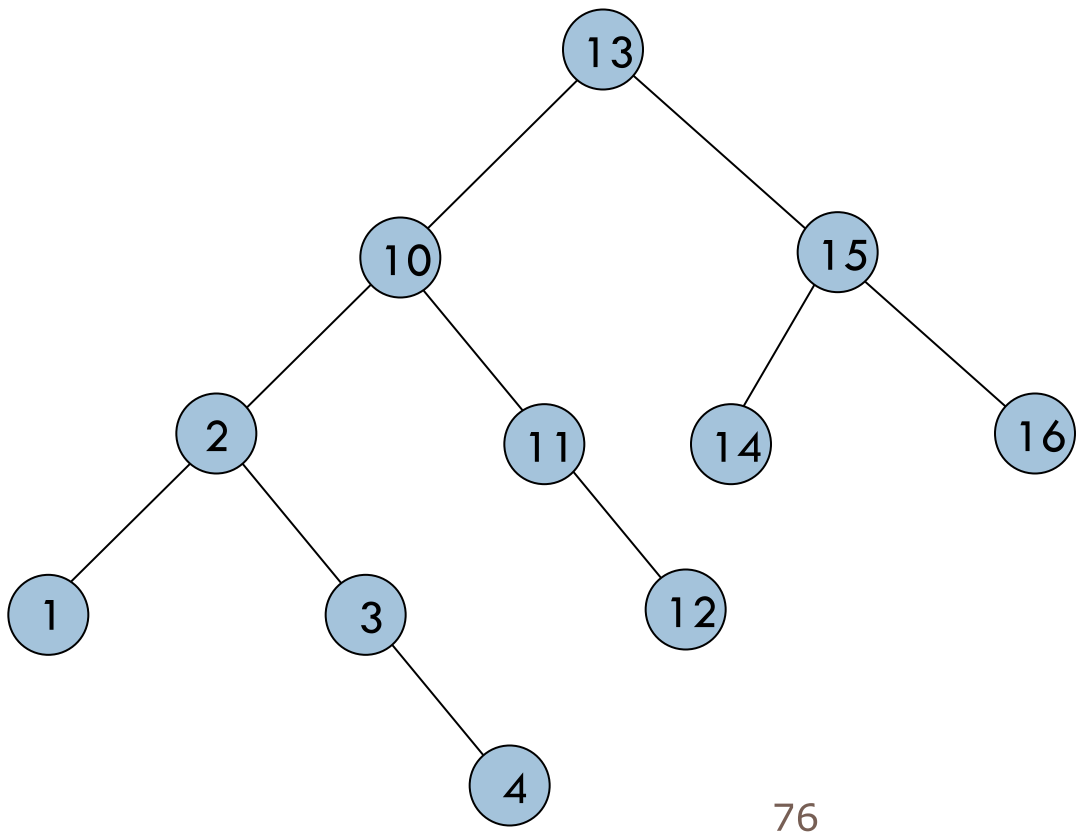


• Now insert 4.

AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

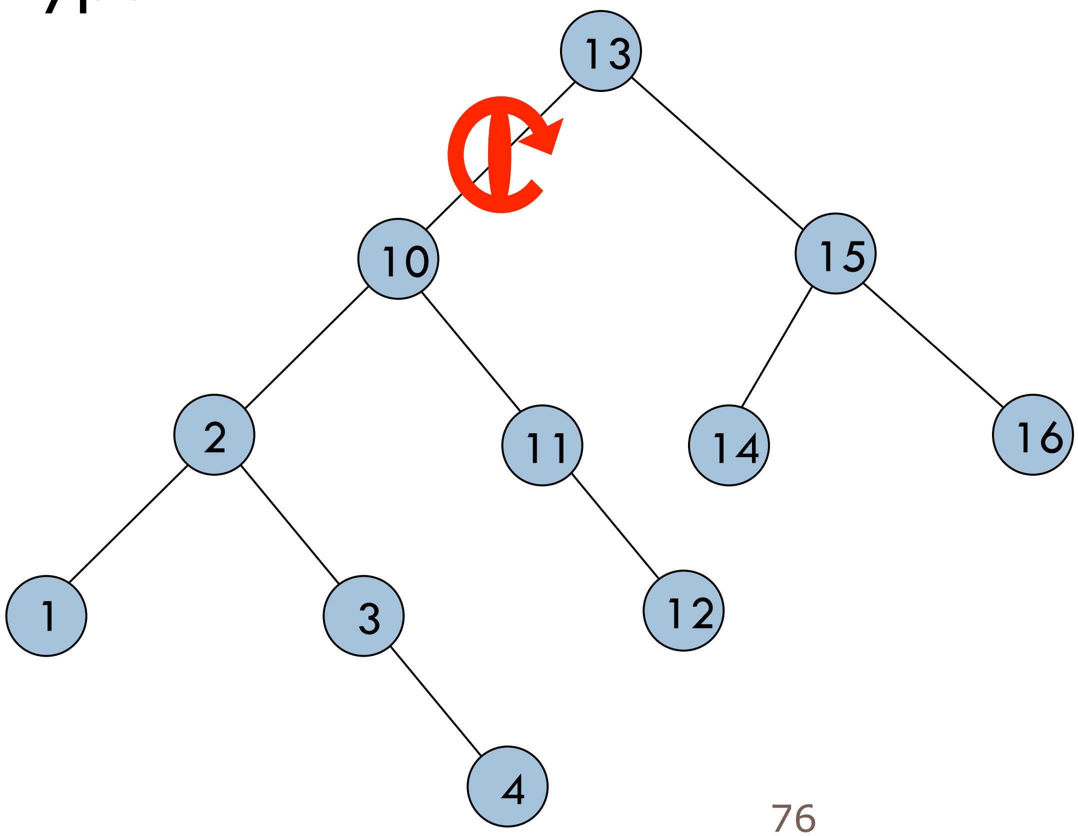
• AVL violation - rotate



AVL Tree Rotations

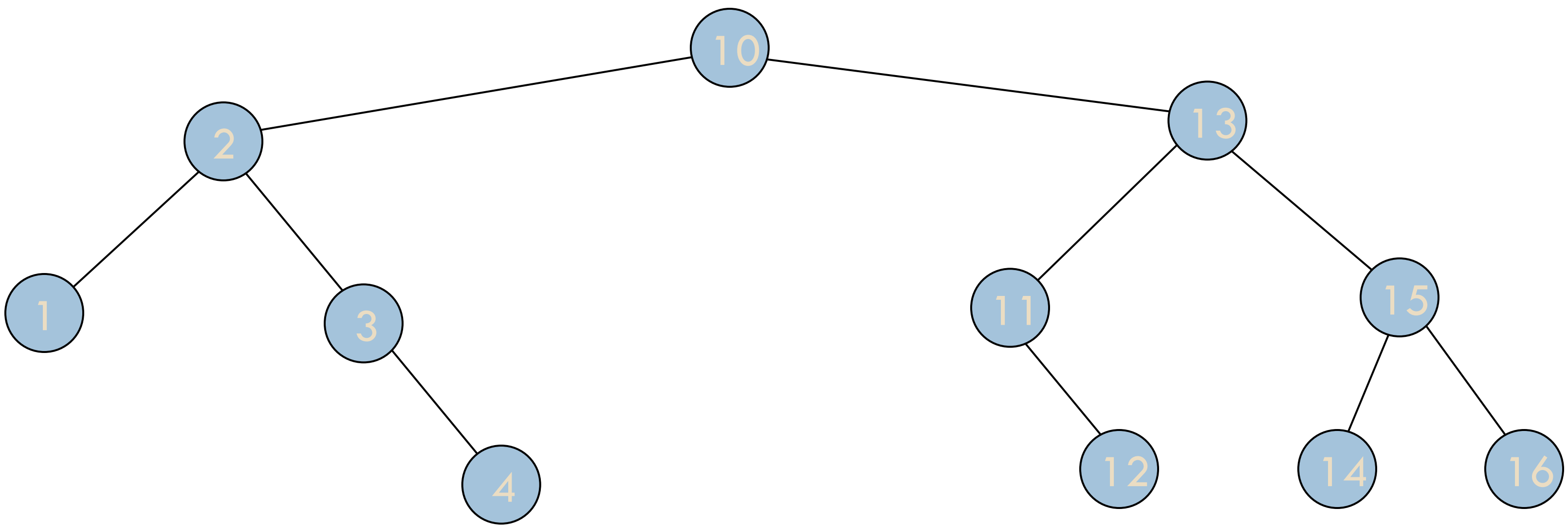
- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

• Rotation type:



AVL Tree Rotations

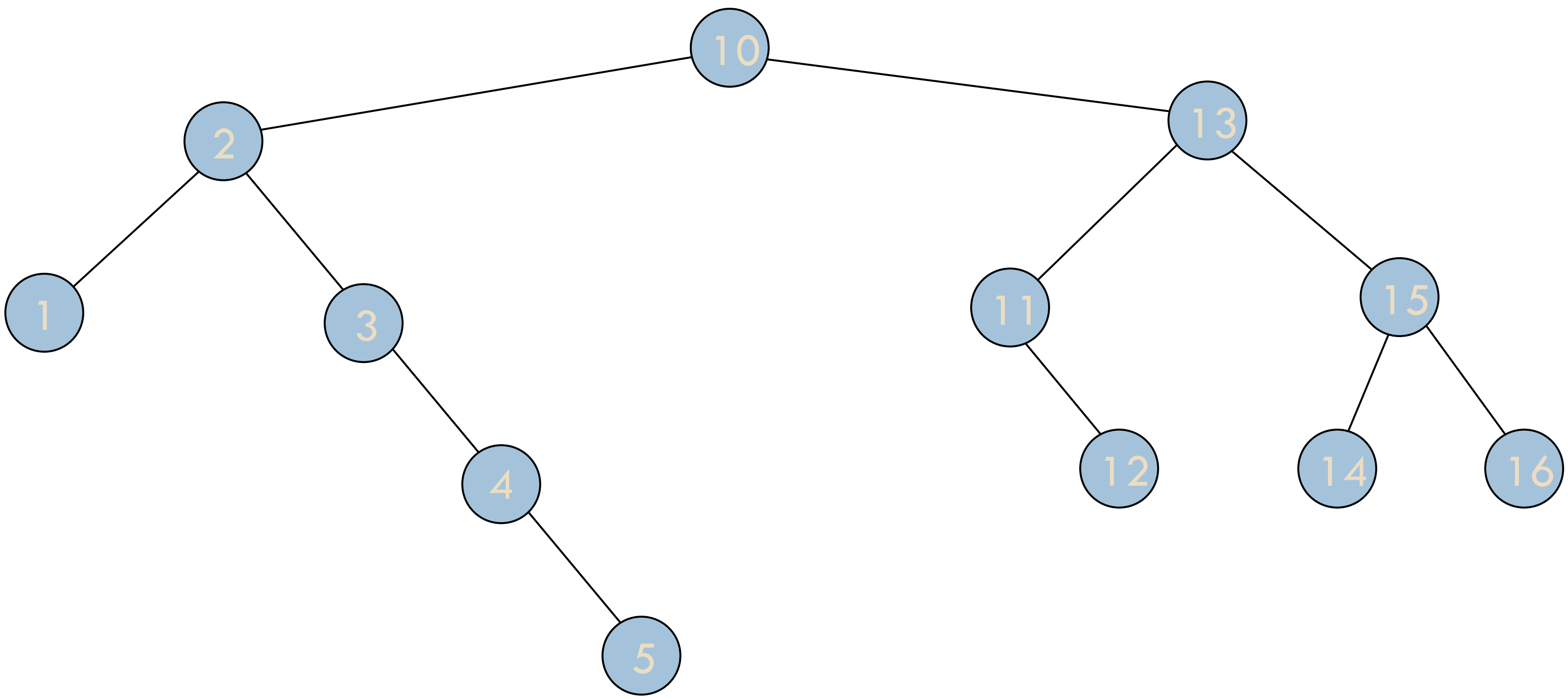
- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8



- Now insert 5.

AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

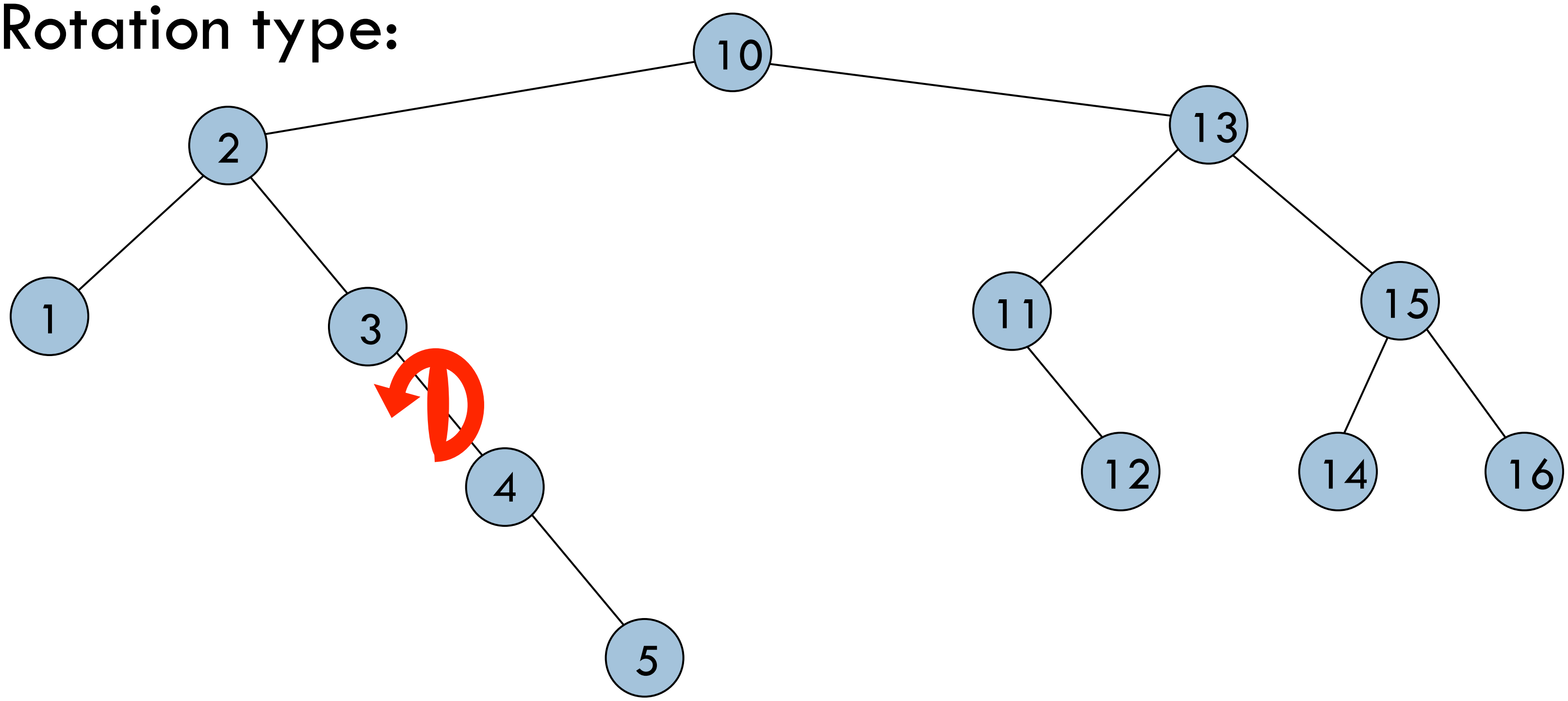


- AVL violation – rotate.

AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

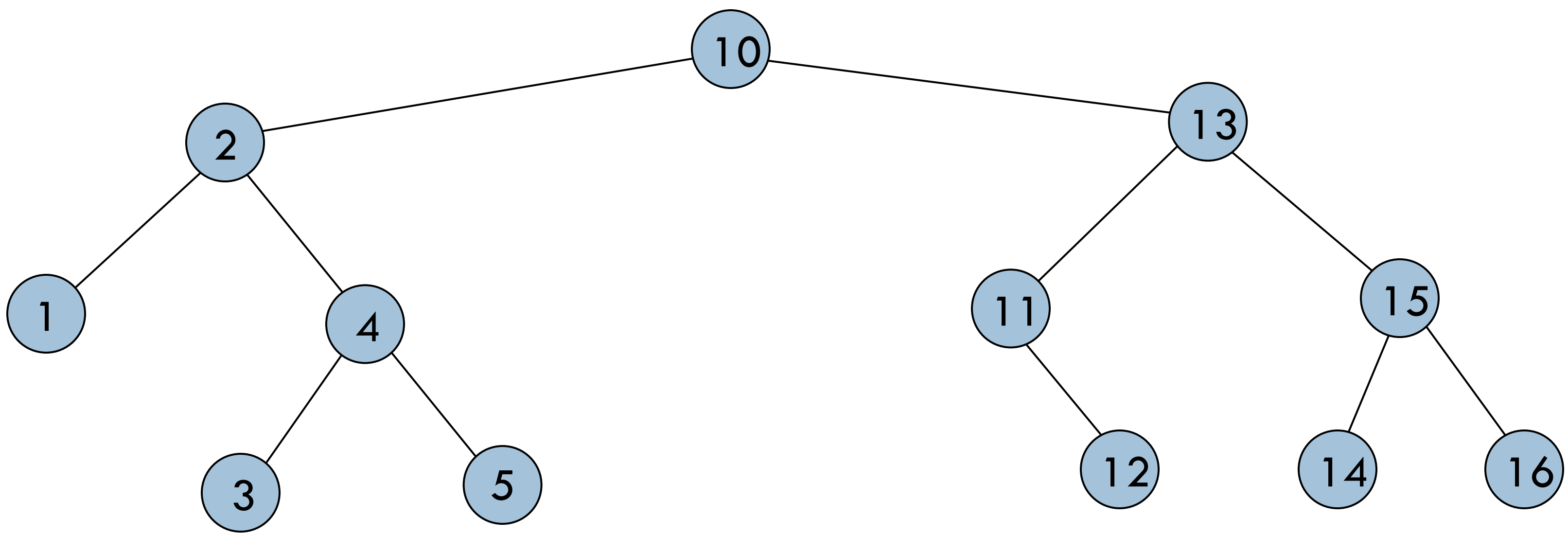
- Rotation type:



AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

- AVL balance restored:

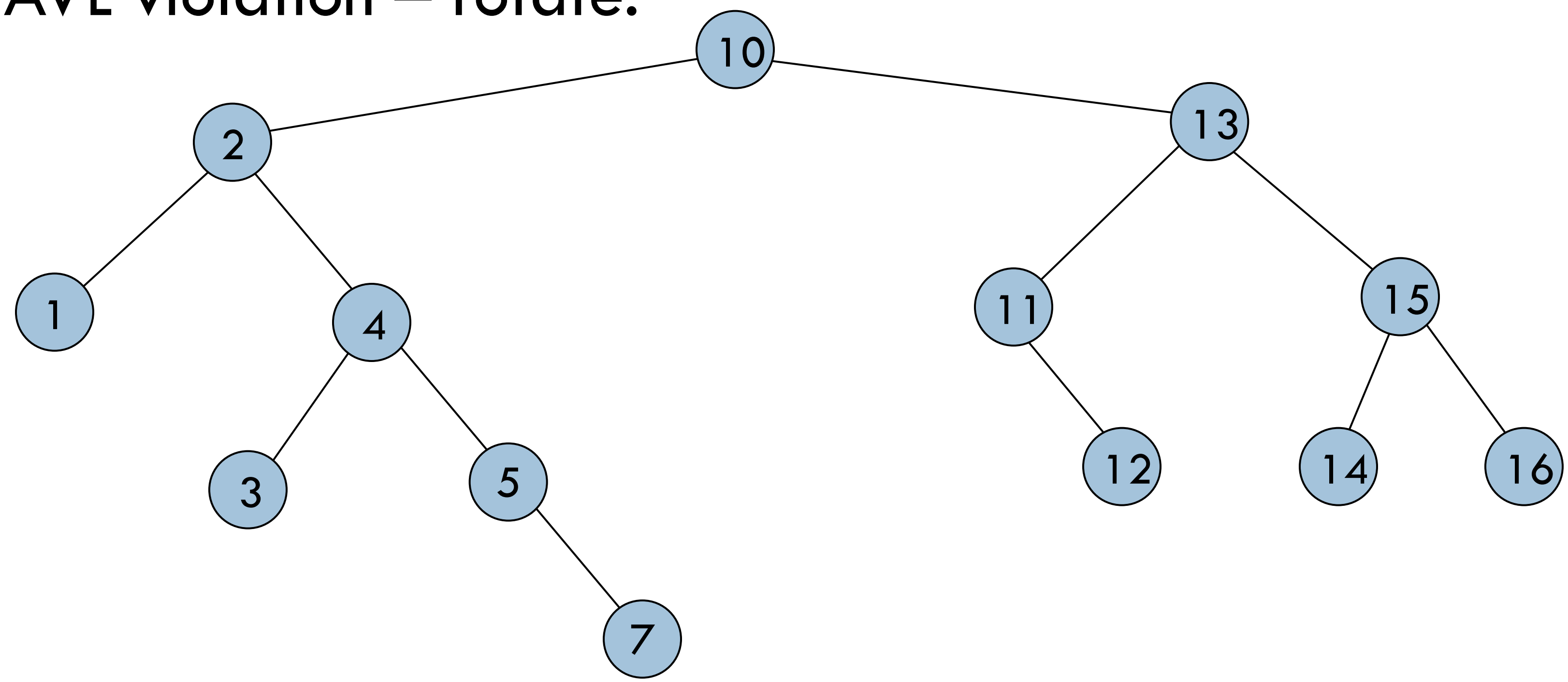


- Now insert 7.

AVL Tree Rotations

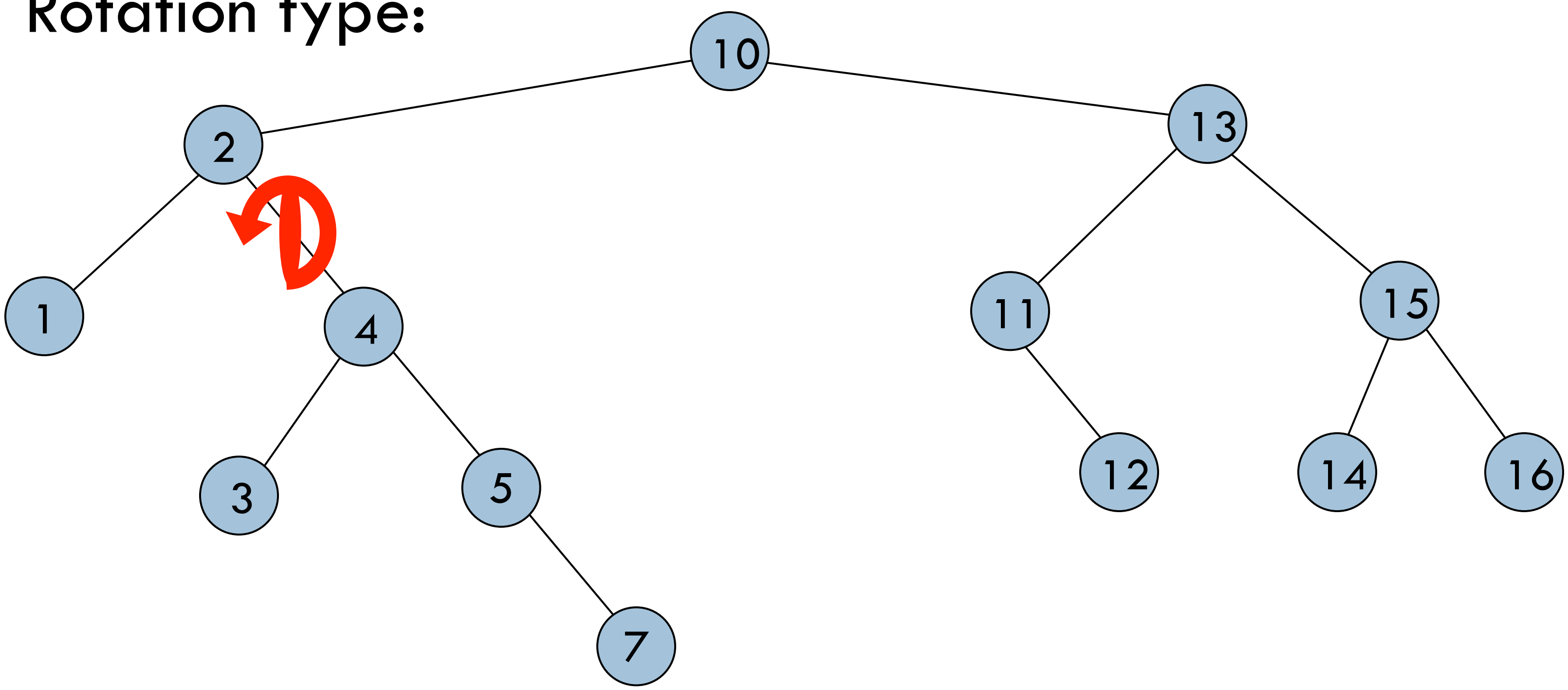
- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

- AVL violation – rotate.



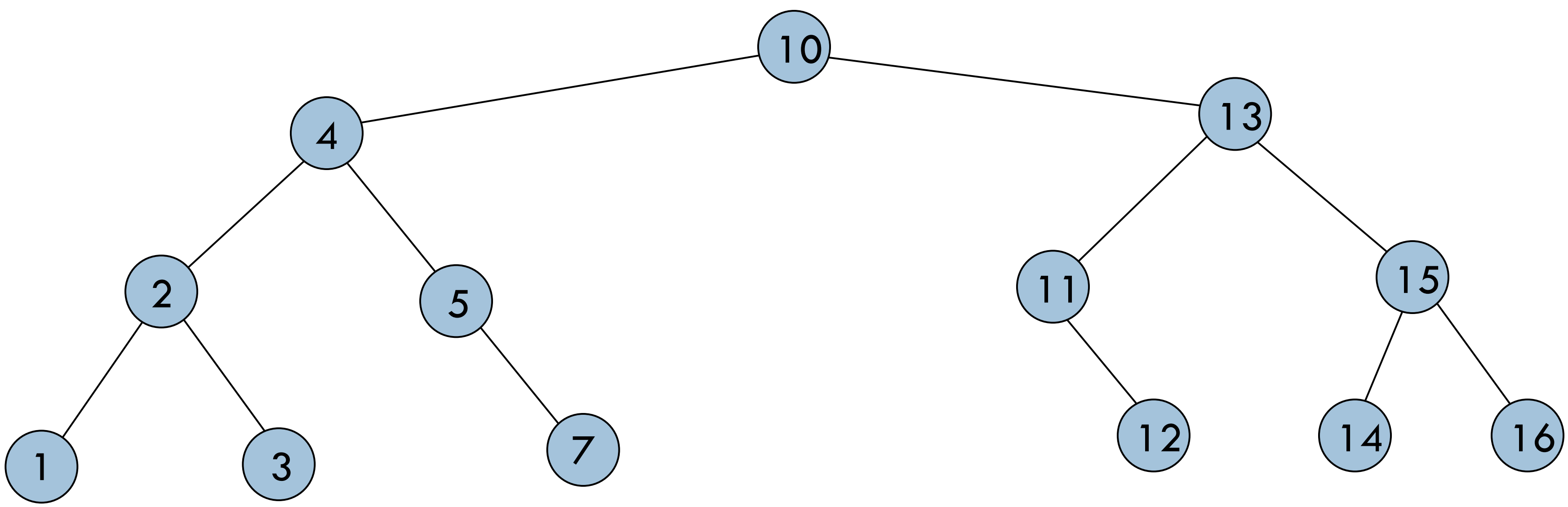
AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8
  - Rotation type:



AVL Tree Rotations

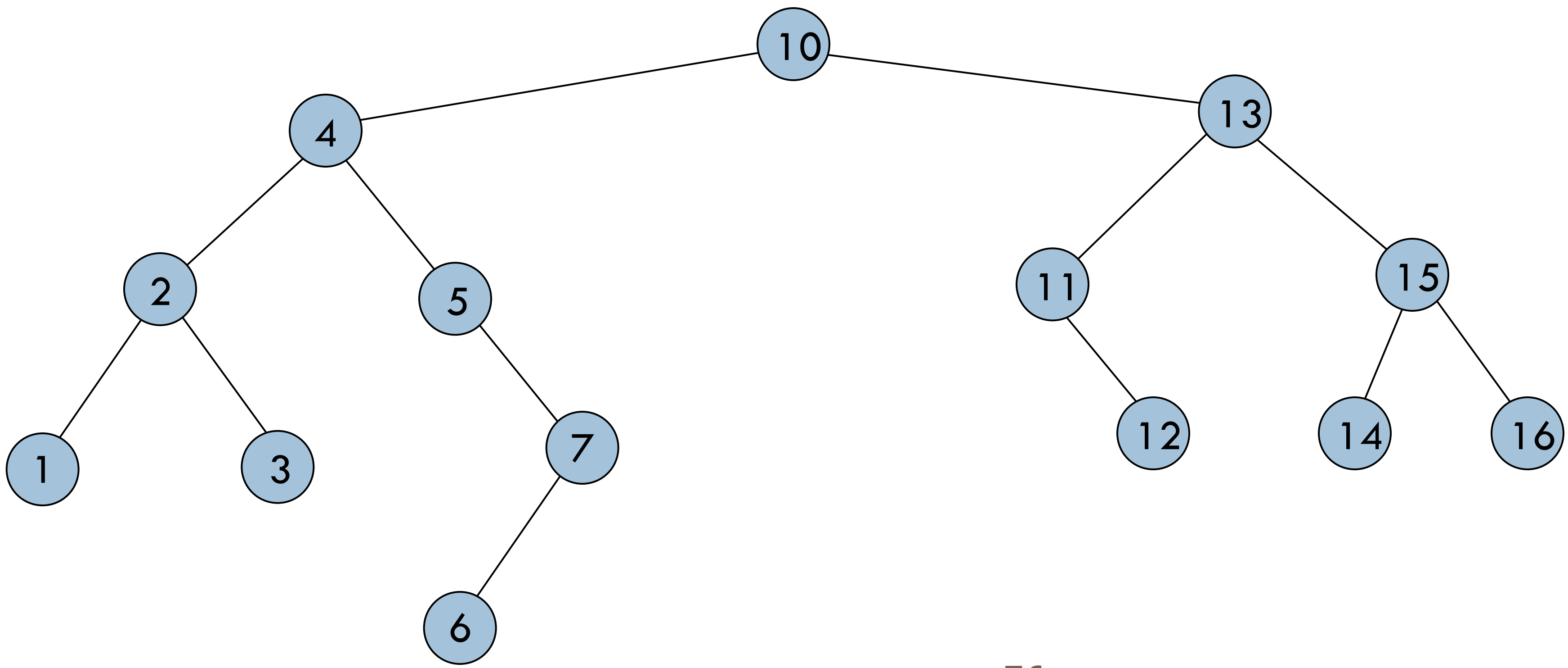
- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8
  - AVL balance restored.



- Now insert 6.

AVL Tree Rotations

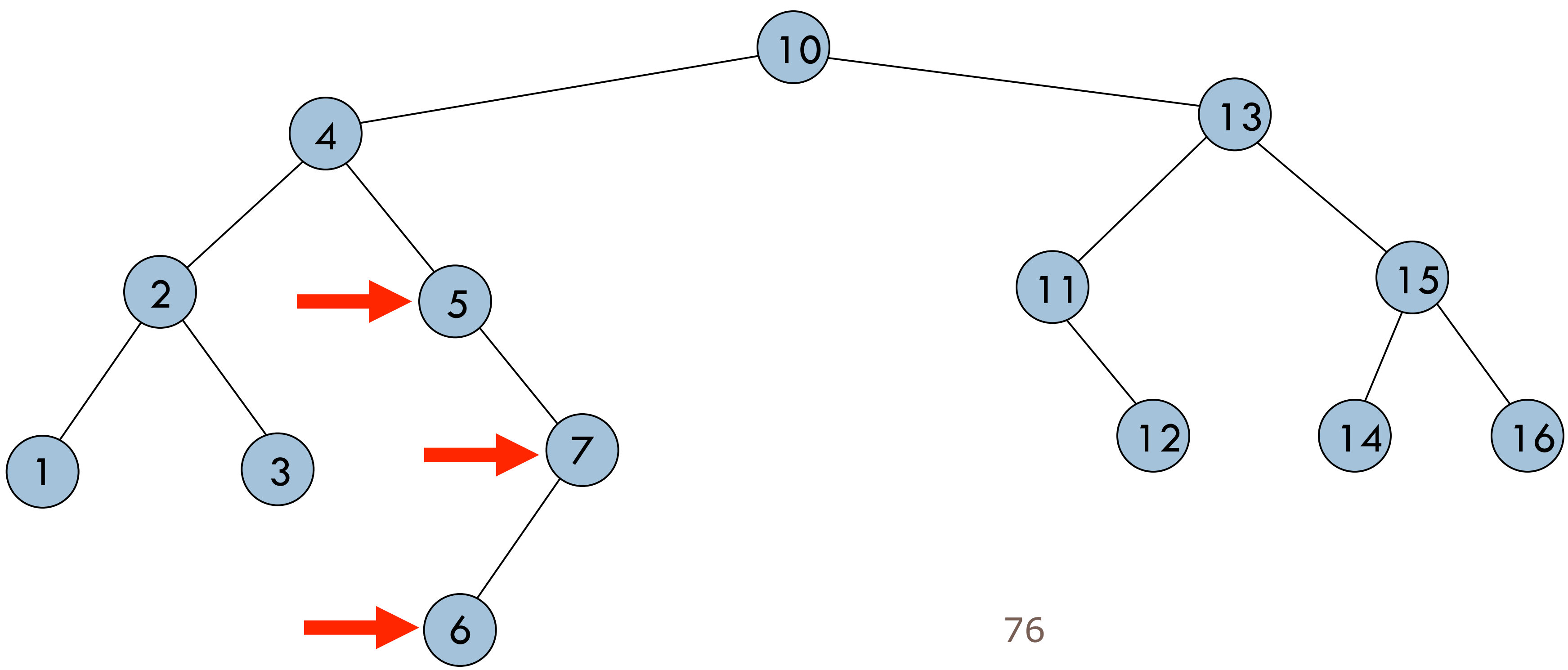
- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8
  - AVL violation - rotate.



AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

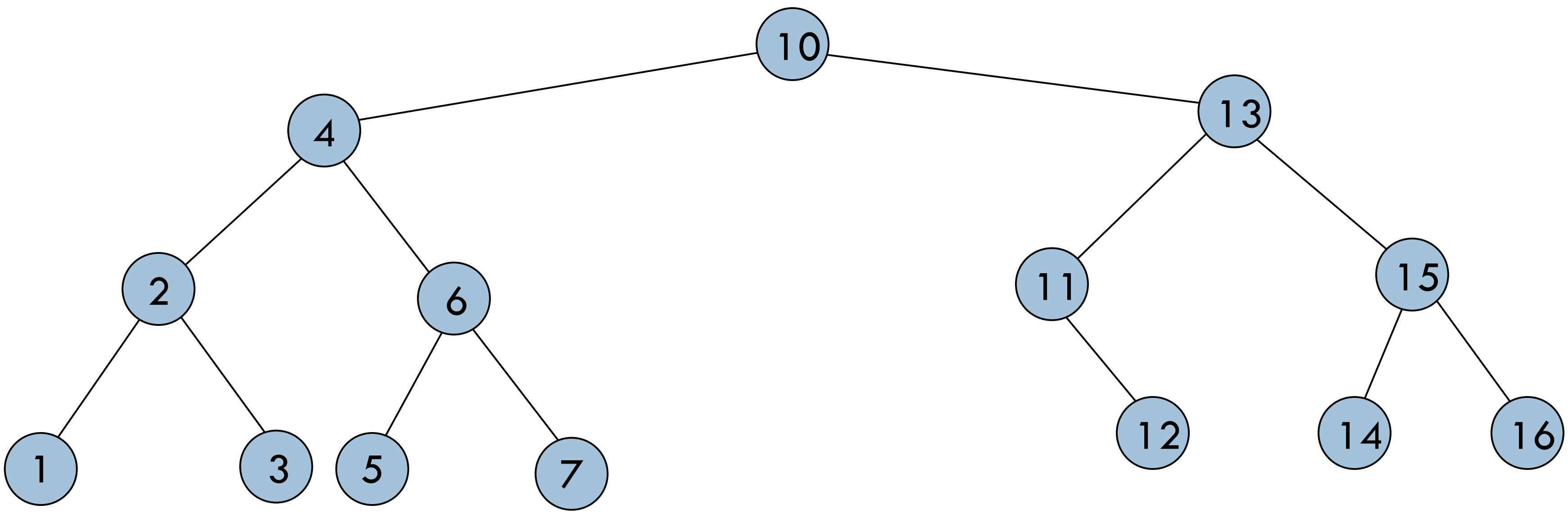
  - Rotation type:



AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

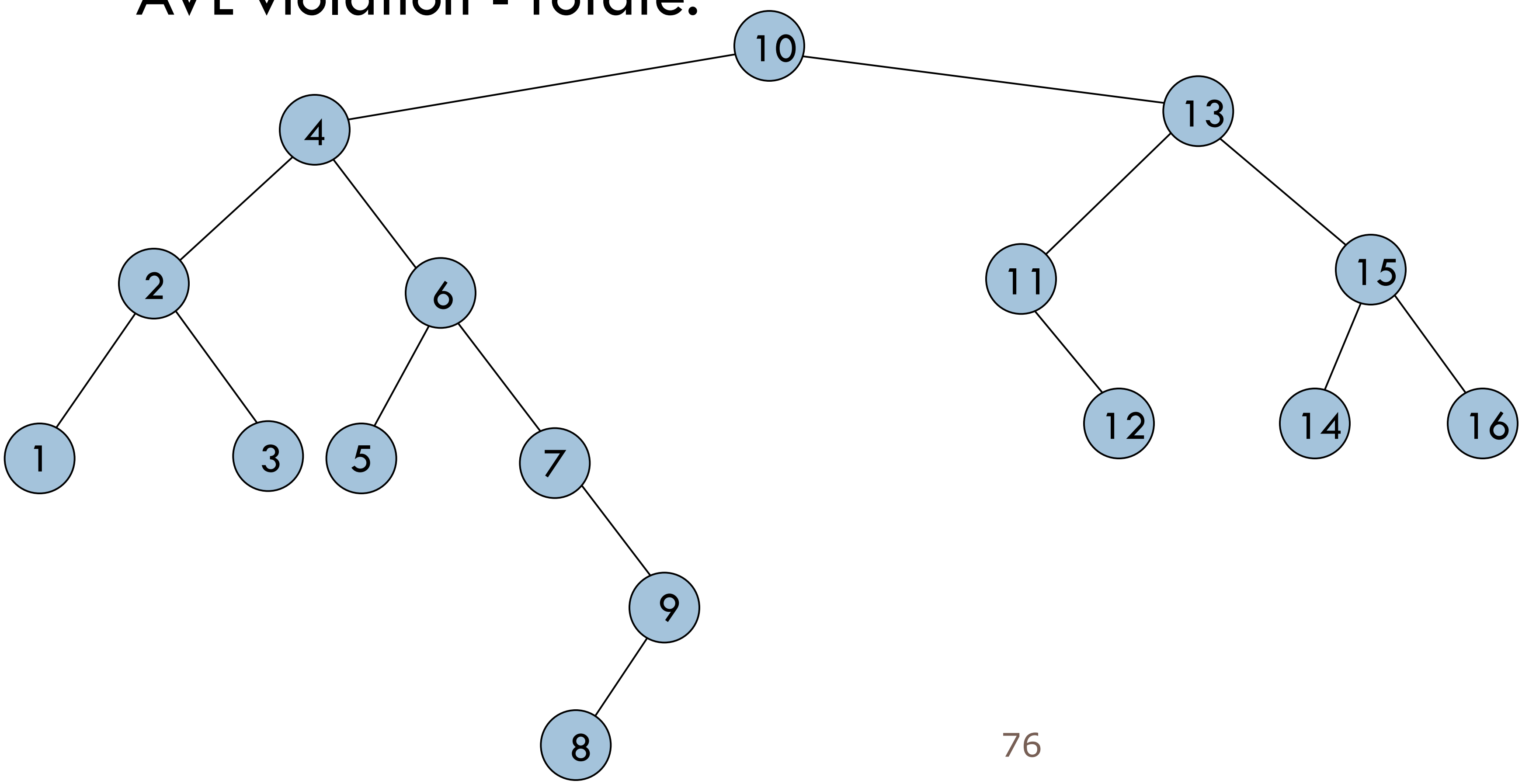
• AVL balance restored.



• Now insert 9 and 8.

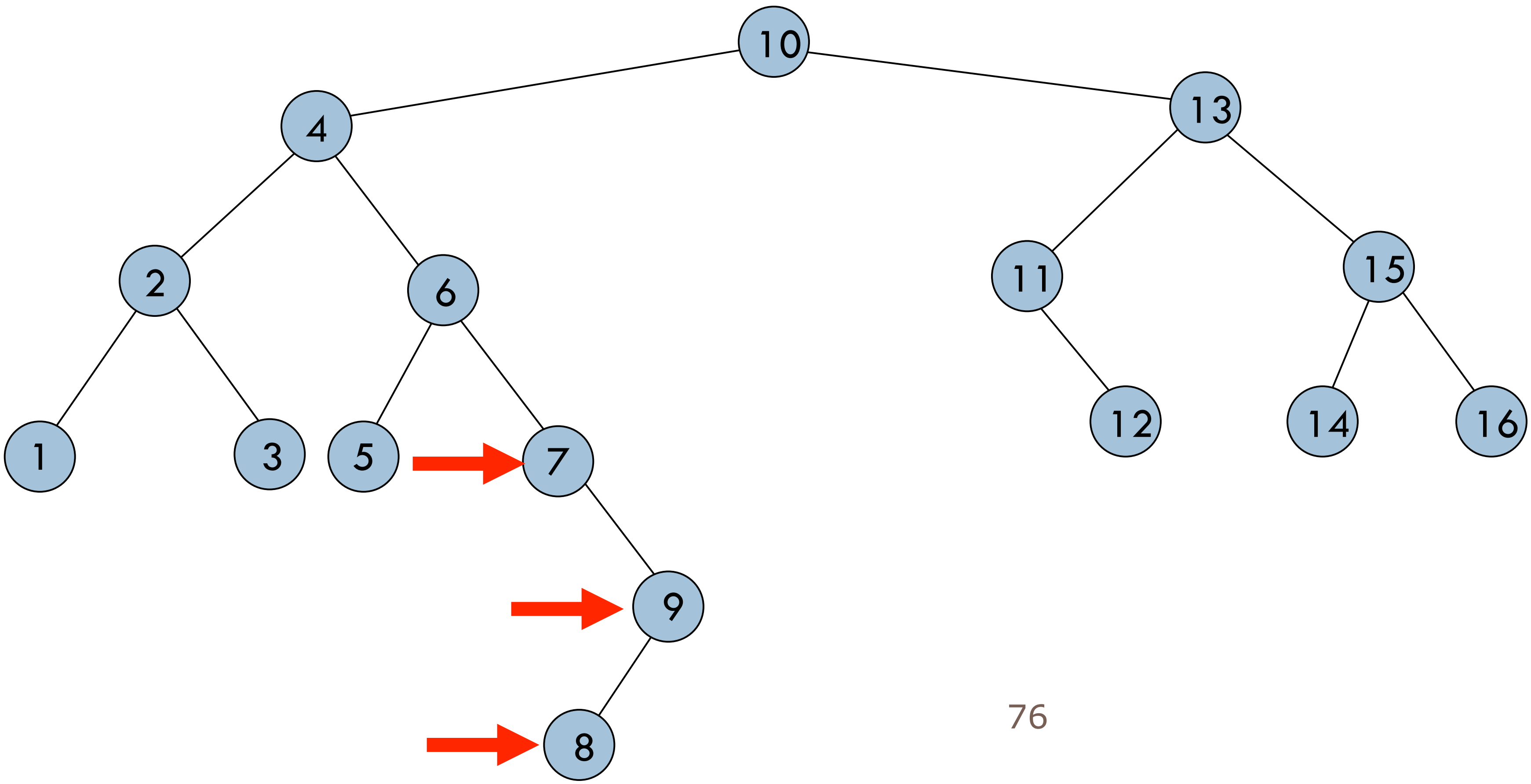
AVL Tree Rotations

- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8
  - AVL violation - rotate.



AVL Tree Rotations

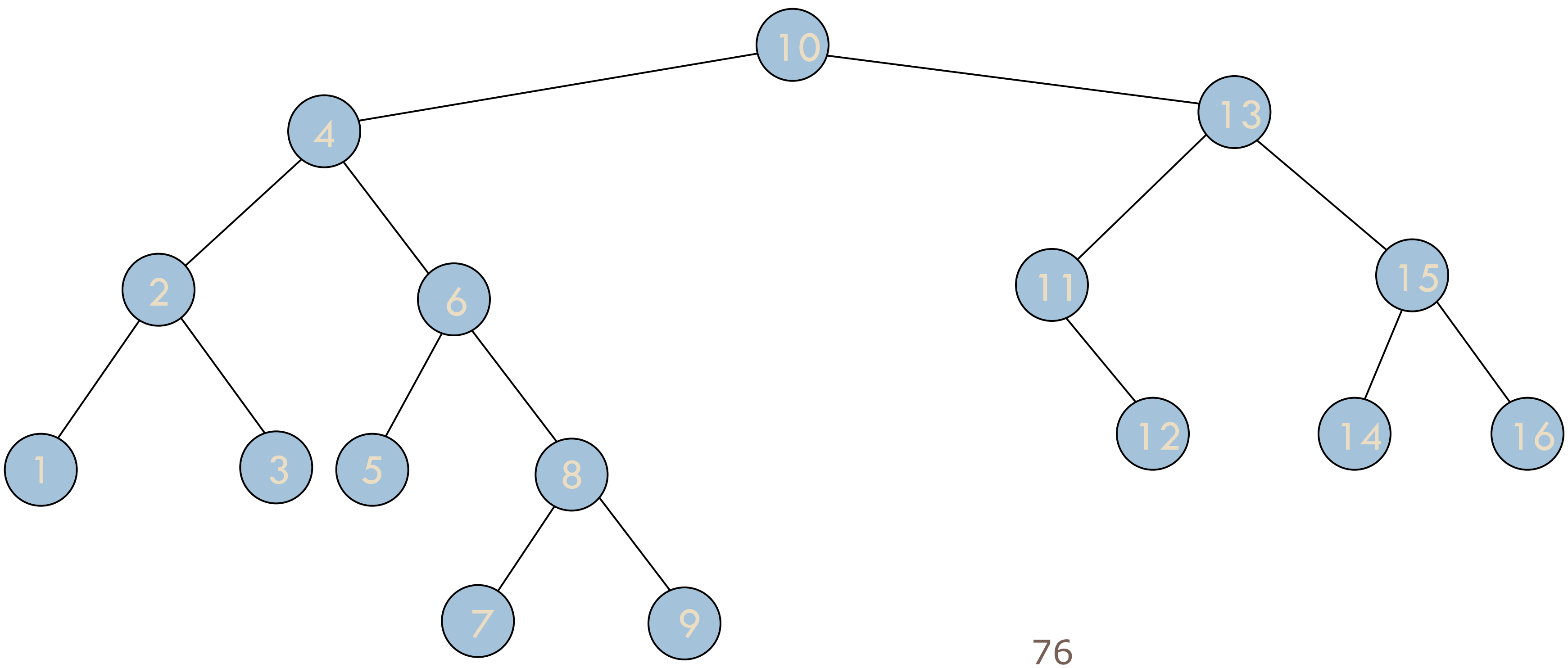
- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8
- Rotation type:



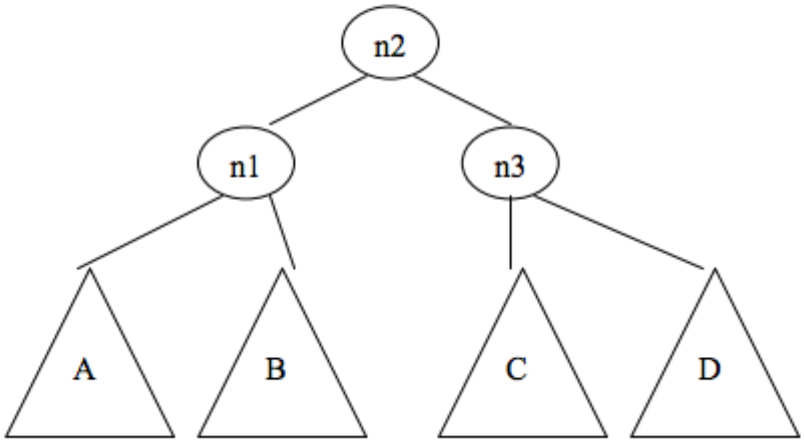
AVL Tree Rotations

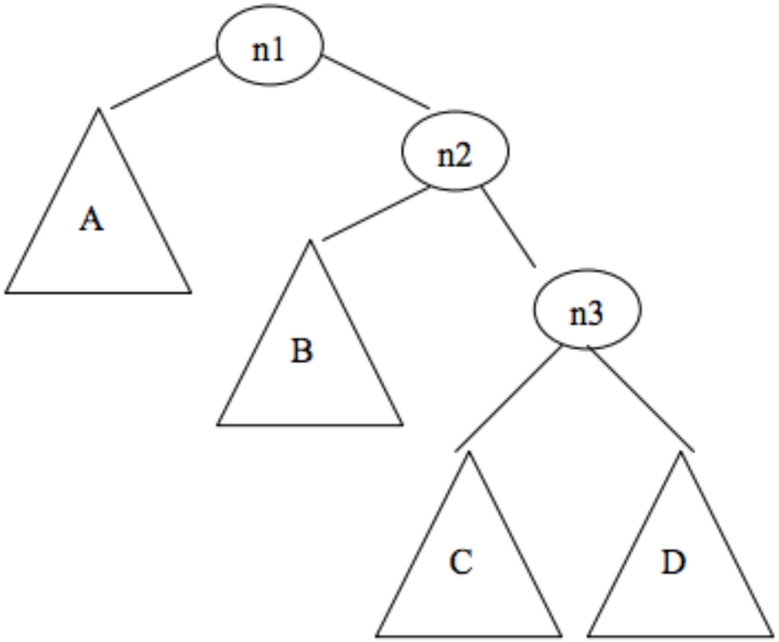
- Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

- Tree is almost perfectly balanced



Exercises





Exercises


```
// rotate n2 with (left child) n1, i.e., clockwise
Node rotateRight (Node n2) {
   Node k = n2.left;
   n2.left = k.right;
   k.right = n2;

   // update heights here if needed
   // ...

   return k;
}


// rotate n2 with (right child) n3, i.e., counter clockwise
Node rotateLeft (Node n2) {
   Node k = n2.right;
   n2.right = k.left;
   k.left = n2;
   // heights ...
   return k;
}
```