M8. Dictionary ADT – Binary Search Tree (BST) ADT/DS

Instructor: Manikandan Narayanan

Weeks 9-10

CS2700 (PDS) Moodle: https://courses.iitm.ac.in/course/view.php?id=4892

Acknowledgment of Sources

- Slides based on content from related
  - Courses:
    - IITM – Profs. **Rupesh**/Krishna(S)/Prashanth/Kartik's PDS (Thy/Lab) offerings (slides, quizzes, notes, lab assignments, etc. for instance from Rupesh's Jul 2019 offering - www.cse.iitm.ac.in/~rupesh/teaching/pds/jul19/)
    - *Most slides are based on Rupesh Nasre's slides – we thank him and acknowledge by marking* ***[RN]*** *in the bottom right of these slides.*
  - Books:
    - **Main textbook:** *"Data Structures and Algorithm Analysis in C++"* by ***Weiss*** (content, figures, slides, exercises/questions, etc.). – cited as [WeissBook]
    - Additional/optional book: *"Practice of Programming"* by *Kernighan and Pike* (style of programming, programming exercises/questions, etc.) – cited as [KPBook]

Dictionary Instances

- Binary Search Trees
  - Balanced BST
- Hash Tables

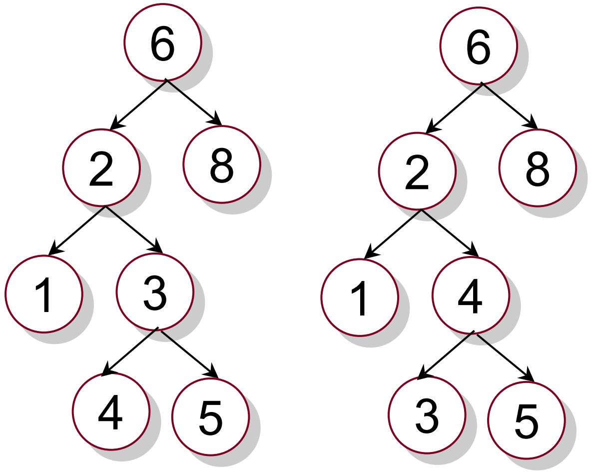
Outline for Module M8

- **M8 Binary Search Tree (BST) ADT**
  - **M8.1 BST DS**
    - **Definition and Basic Operations**
    - **Insertion and Deletion Operations**
  - M8.2 AVL Tree DS
    - Definition and Basic Properties
    - Insertion and Deletion Operations

      (with single/double rotations)

Definition

- A BST is a binary tree.
- If it is non-empty, the value at the root is larger than any value in the left-subtree, and
- the value at the root is smaller than any value in the right-subtree.
- The left and the right subtrees are BSTs.
- Assumption: All values are unique.



Not a BST **BST**

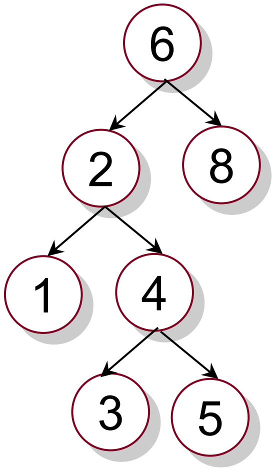
[RN]

BST ADT

```
class BST {
        ...
public:
        ...
        PtrToNode insert(DataType element);
        bool delete(DataType element);
        PtrToNode search(DataType element);
        PtrToNode findMin();
        PtrToNode findMax();
};
```

Search

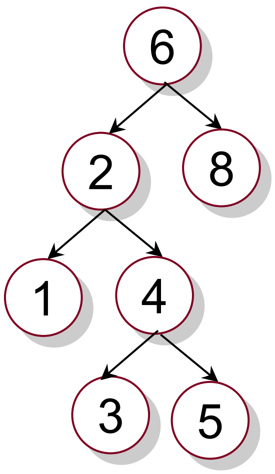
```
bool Tree::search(DataType data, PtrToNode rr) {
	if (rr == NULL) return false;
	if (data == rr->data) return true;
	if (data < rr->data) return search(data, rr->left);
	return search(data, rr->right);
}
bool Tree::search(DataType data) {
	bool present = search(data, root);
	if (present)
		std::cout << data << " present." << std::endl;
	else
		std::cout << data << " NOT present." << std::endl;
	return present;
}
```



Switch to bst.cpp.

FindMin

```
DataType Tree::findmin(PtrToNode rr) {
	if (rr) {
		if (rr->left) return findmin(rr->left);
		return rr->data;
	}
	return -1;
}
DataType Tree::findmin() {
	return findmin(root);
}
```

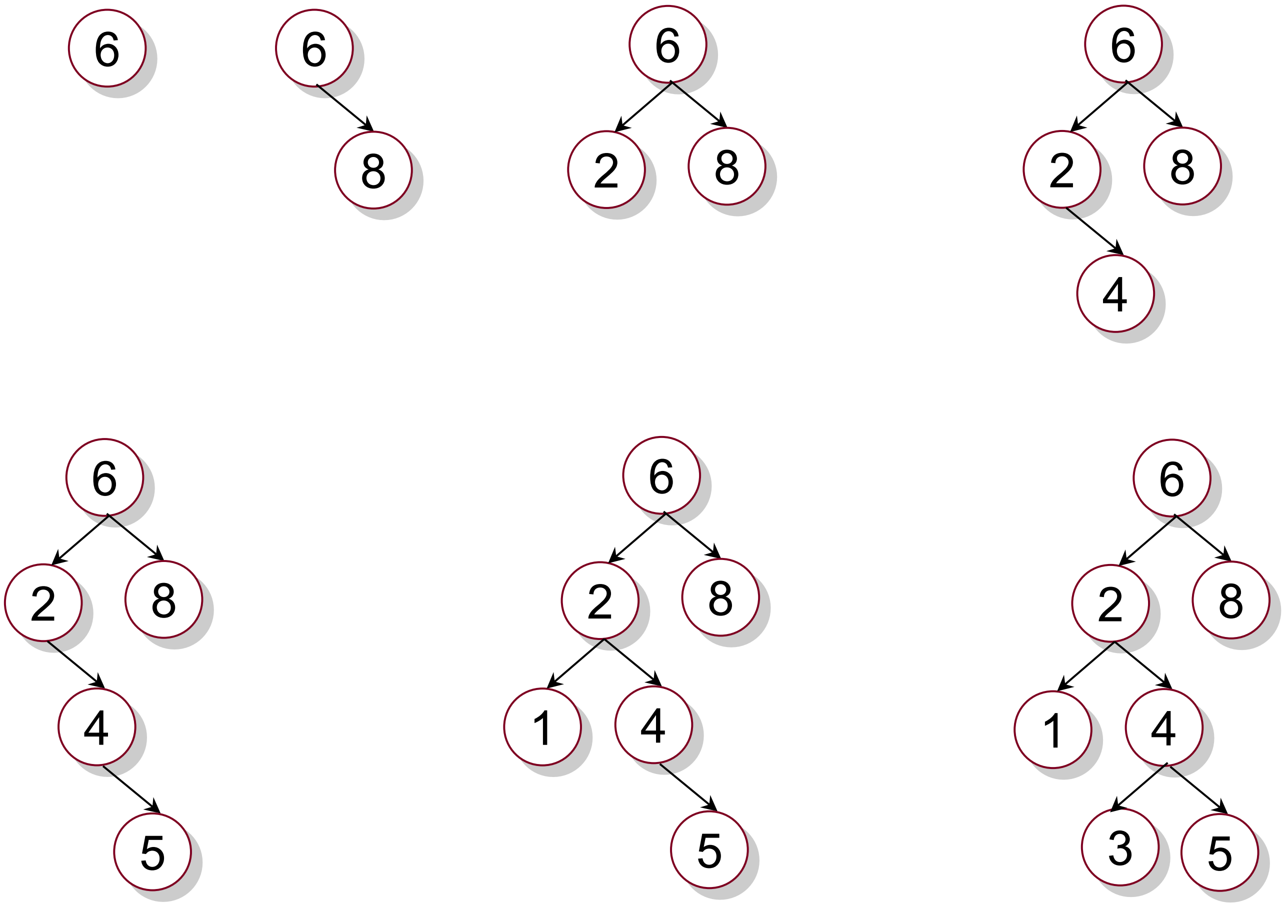


Write iterative findMax.

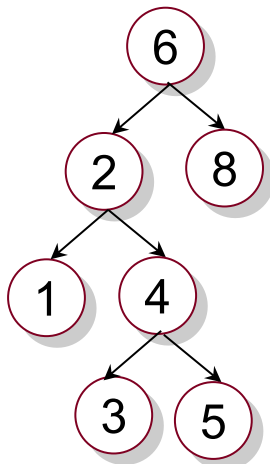
```
DataType Tree::findminiterative() {
	PtrToNode ptr = root;
	if (ptr) {
		while (ptr->left) ptr = ptr->left;
		return ptr->data;
	}
	return -1;
}
```

Insert

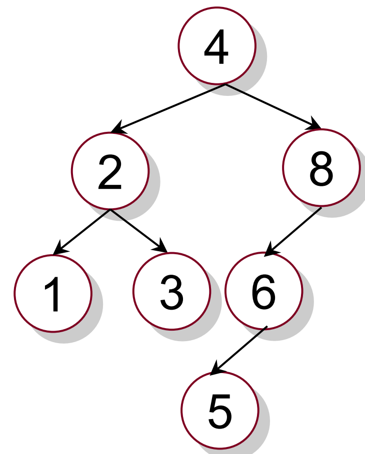
Insert 6, 8, 2, 4, 5, 1, 3



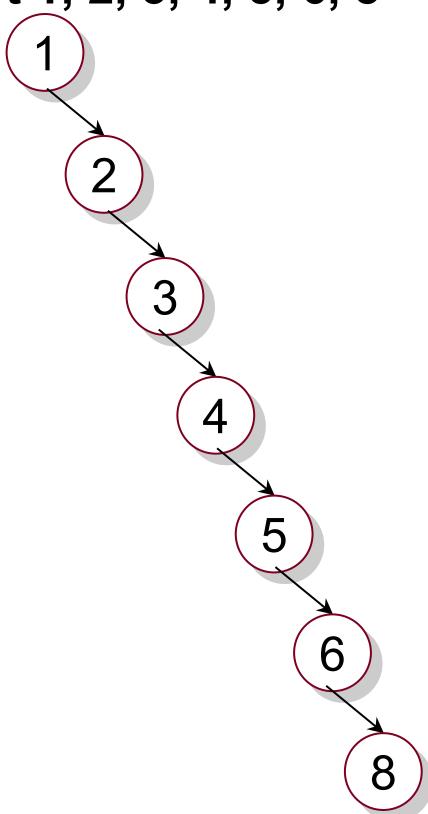
Insert 6, 8, 2, 4, 5, 1, 3



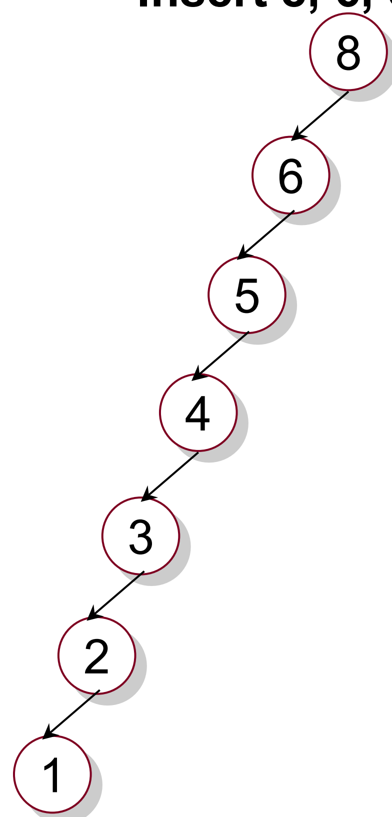
Insert 4, 8, 2, 6, 5, 1, 3



Insert 1, 2, 3, 4, 5, 6, 8



Insert 8, 6, 5, 4, 3, 2, 1



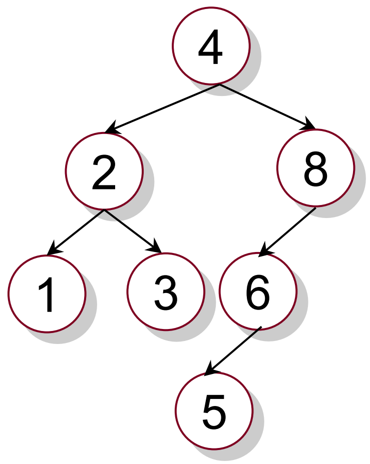
Perform inorder traversal on the last BST.

Insert

- Insertion order may change the tree structure.
  - Different insertion orders may form the same tree.
- Inorder traversal prints the values in exactly the same order, irrespective of the BST structure.
  - This is also the sorted order.
- The first insertion forms the root.
- Insertions always happen at leaves.
  - New node cannot be added as an intermediate node.
- Insertion order decides the tree height.
  - Tree height affects efficiency/complexity of operations.

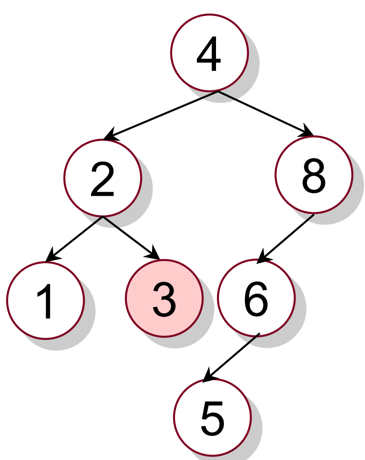
Insertion Orders

- For this BST, find three different insertion orders.
  - 4, 2, 8, 1, 3, 6, 5
  - 4, 8, 2, 6, 1, 3, 5
  - 4, 2, 1, 3, 8, 6, 5
  - ...

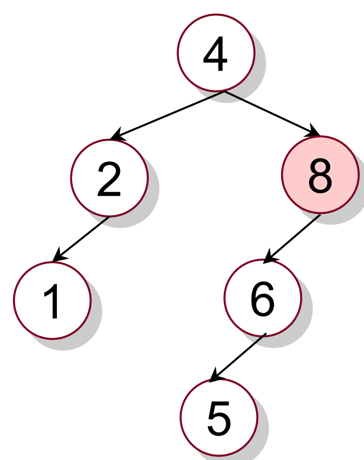


Remove(value)

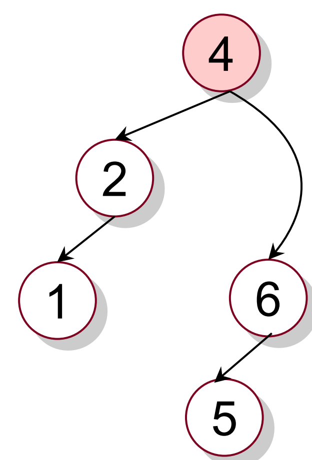
- Search for the node **n** to be removed.
- If **n** is a **leaf**, remove **n** from its parent.
- If **n** has **one** child **c**, make **c** the child of **n**'s parent.
- If **n** has **two** children, scratch your head.



Remove(3)



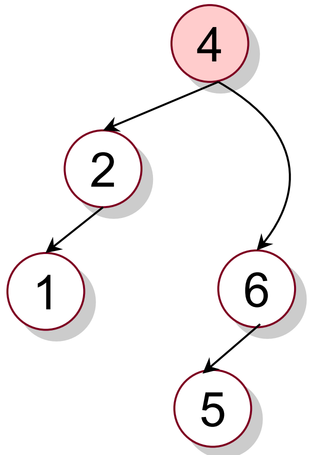
Remove(8)



Remove(4)

Remove(value)

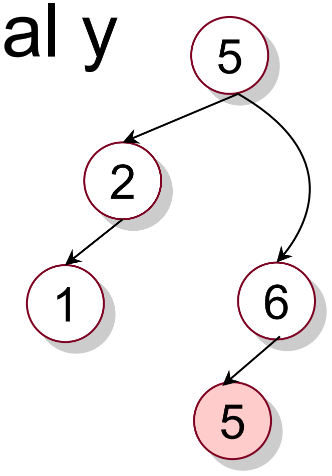
- We would like to convert this complicated remove into another simpler remove.
  - That is, convert this case of two children into a case of one child or zero children.
  - For instance, remove(4) can be converted to remove(5) or remove(6) or remove(2) or remove(1).
  - Which one would be the best, in general?
- **General strategy:**
  - Copy the smallest value from the
  - right subtree here.
  - Recursively delete that smallest value.



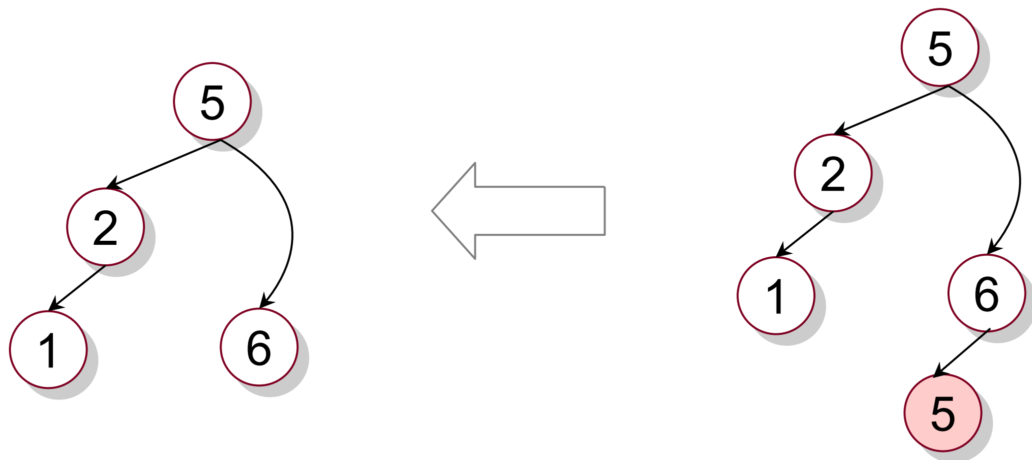
Remove(4)

Guarantees on the smallest value

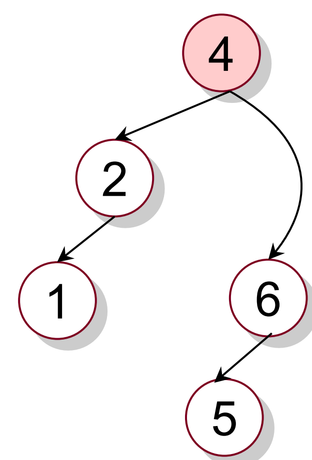
- If x is the value to be deleted, and y is the smallest value in its right subtree,
  - There are no values in the BST between x and y.
  - The node with y value cannot have two children.
  - In fact, y cannot have a left child.
  - When y replaces x, after removing original y node, the BST structure is not affected.
  - Removal of a node with two children does not result in further removal of another node with two children.



Remove(value)



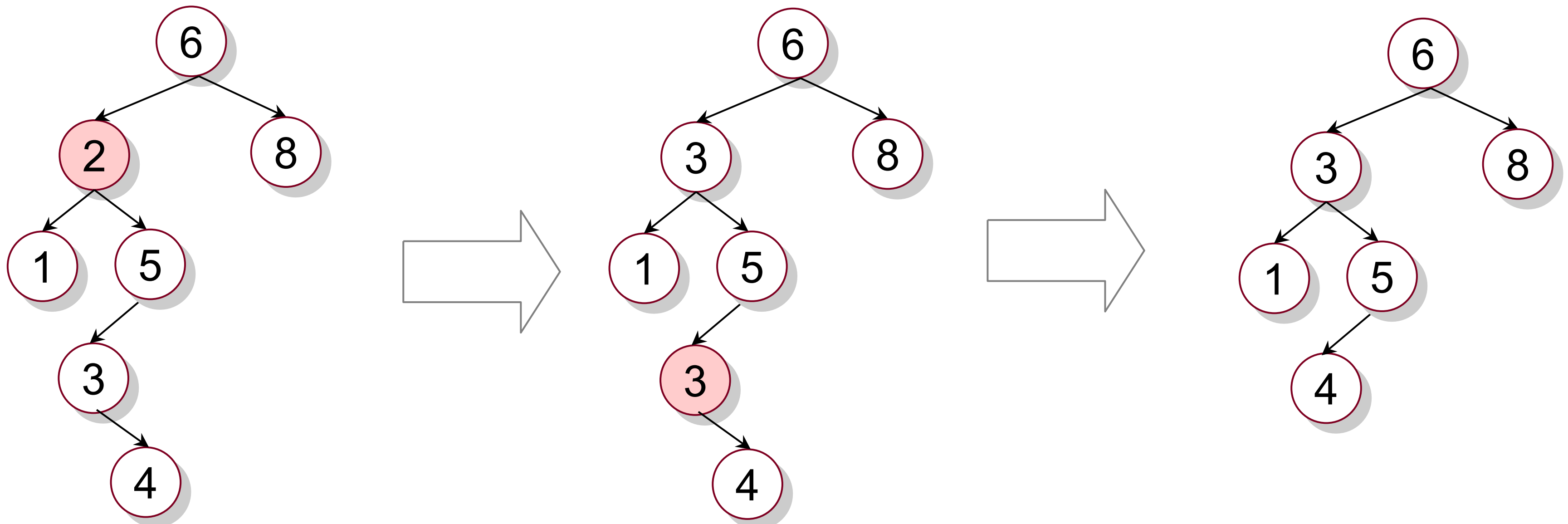
Remove(5)



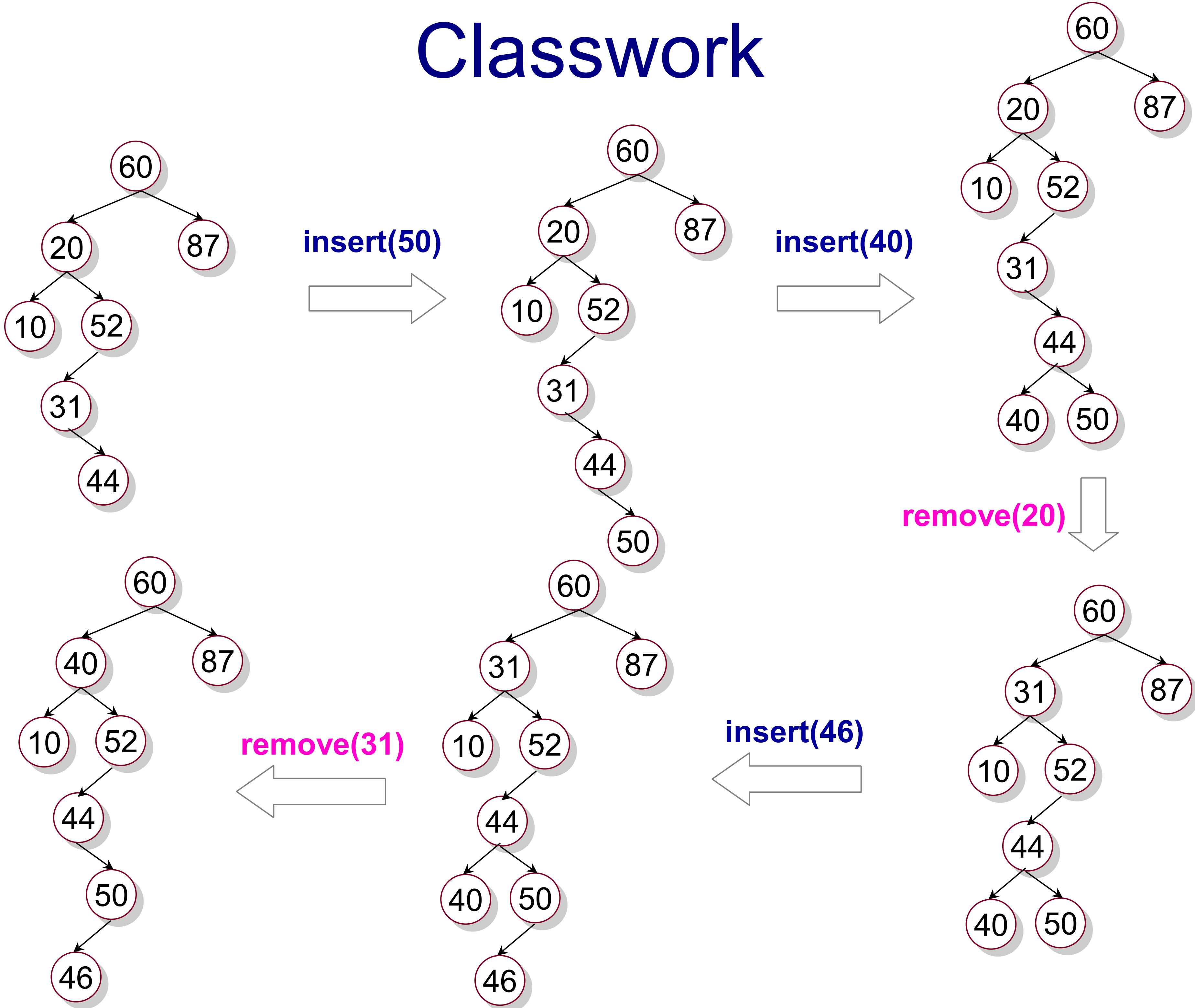
- **General strategy:**
  - Copy the smallest value in the
  - right subtree here.
  - Recursively delete that smallest value

Remove(4)

Remove(value)



Classwork



60
20
87
10
52
31
44
insert(50)
60
20
87
10
52
31
44
50
insert(40)
60
20
87
10
52
31
44
40
50
remove(20)
60
31
87
10
52
44
40
50
insert(46)
60
31
87
10
52
44
40
50
46
remove(31)
60
40
87
10
52
44
50
46

Time Complexity

- **Search**
  - One may get tempted to conclude it to be O(log n).
  - But it is O(tree height), which could be O(n).
- **Insert**
  - Same as that of search.
- **Remove**
  - There may be two remove calls.
  - Still the complexity does not change. It is same as that of search.
- All the complexities improve if the BST is height-balanced (AVL trees, Splay Trees, red-black trees, B trees, …).

Some Questions?

- What if a BST has duplicates?
- Can a BST node contain strings? Other types?
- Can I store more pointers in a node?

Exercises

- Given a binary tree, find out if it is BST.
- Given an insertion sequence, how would you permute it to achieve the minimum height of the resultant BST?
  - See if your answer has a resemblance with binary search in a sorted array.
- Count the number of leaves in a BST.
- Print a BST in a level-order (breadth-first) manner.
- Write a program to print values in a BST in reverse-sorted order.

Outline for Module M8

- **M8 Binary Search Tree (BST) ADT**
  - M8.1 BST DS
    - Definition and Basic Operations
    - Insertion and Deletion Operations
  - **M8.2 AVL Tree DS**
    - **Definition and Basic Properties**
    - **Insertion and Deletion Operations**

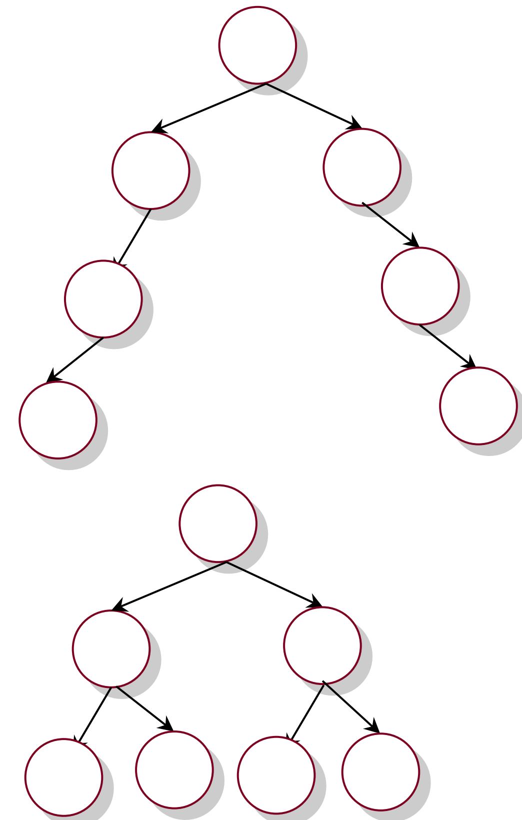
      (with single/double rotations)

AVL Trees

- Normal BSTs may have height O(N).
- As long as BST property is satisfied, the BST can be restructured to maintain O(log N) height.
- Invented by **two** researchers Georgy Adelson-Velsky and Evgenii Landis from Russia in 1962.
- Often called height-balanced trees or self-balancing BSTs.

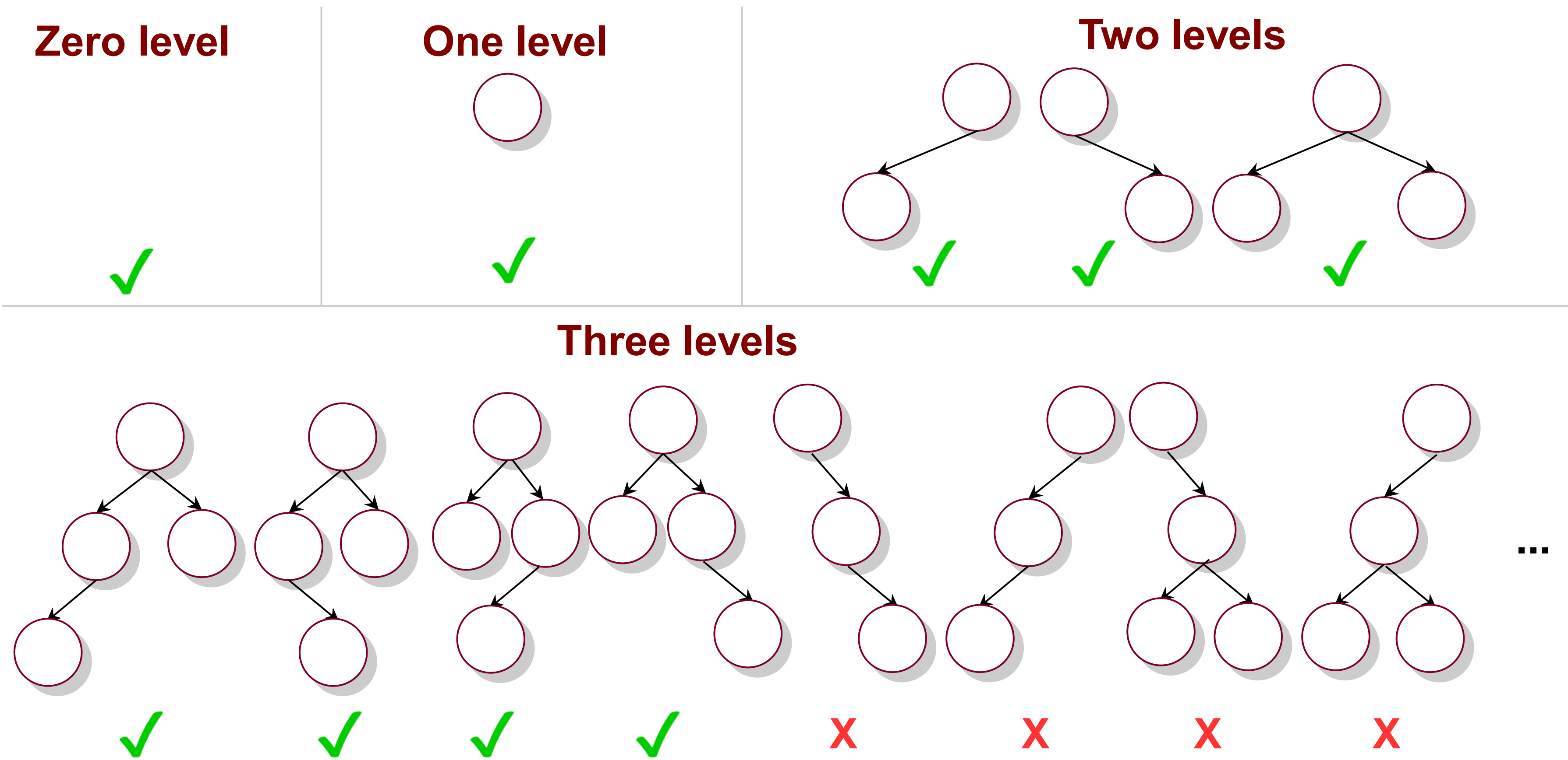
What Doesn't Work

- Ensure that at the root, the left and the right subtrees have the same heights.
  - Doesn't guarantee height balance.
- Ensure the above at every node in the BST.
  - Allows only a few BSTs (number of nodes $2^K - 1$)



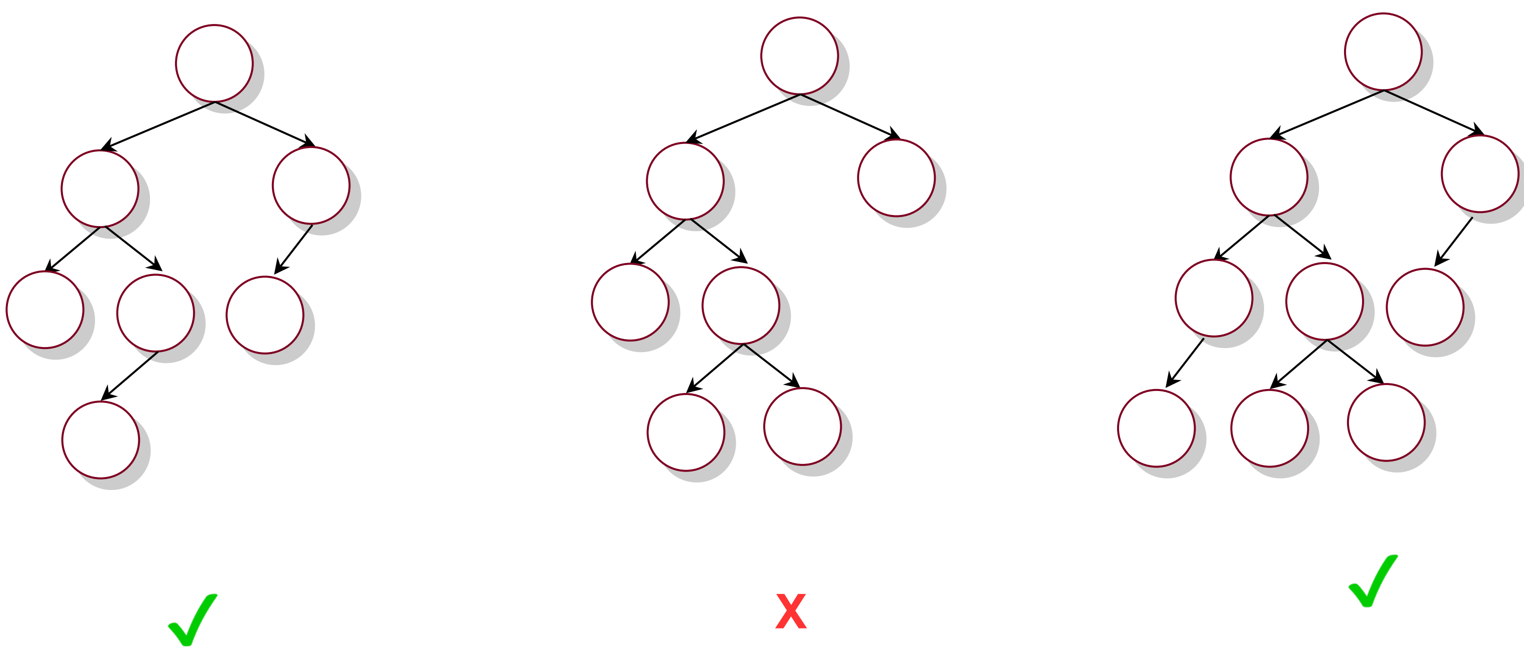
AVL Property

- At every node, the height-difference must not exceed 1.



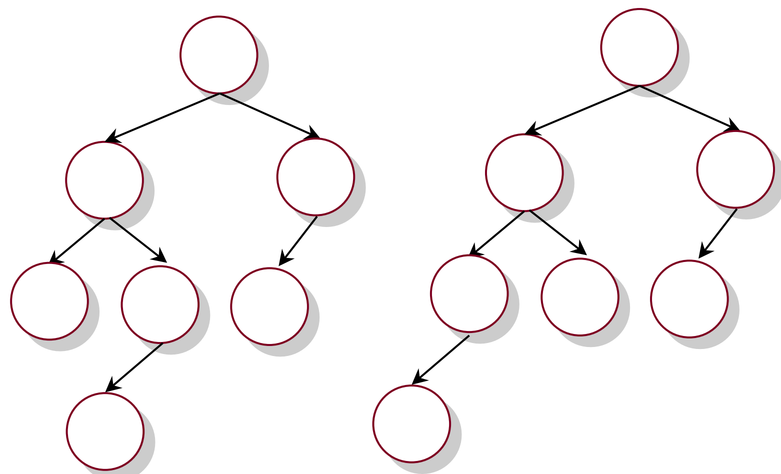
AVL Property

- At every node, the height-difference must not exceed 1.
- **Classwork**: Which of these have AVL structure?



AVL Property

- **Classwork**: Find ~~the~~ an AVL tree having four levels and fewest number of nodes.



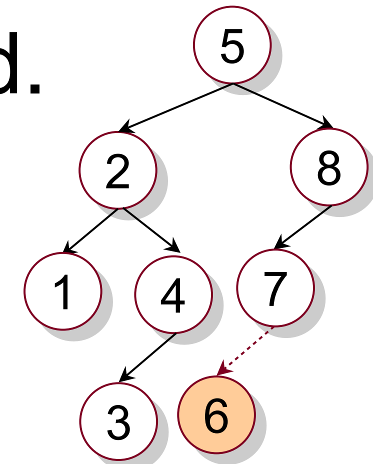
...

Minimum number of nodes in an h-height AVL tree:

S(h) = S(h-1) + S(h-2) + 1

- **Classwork**: Find the maximal AVL tree having four levels (such that addition of any edge makes it a height-imbalanced BST).

Insertion may violate AVL property



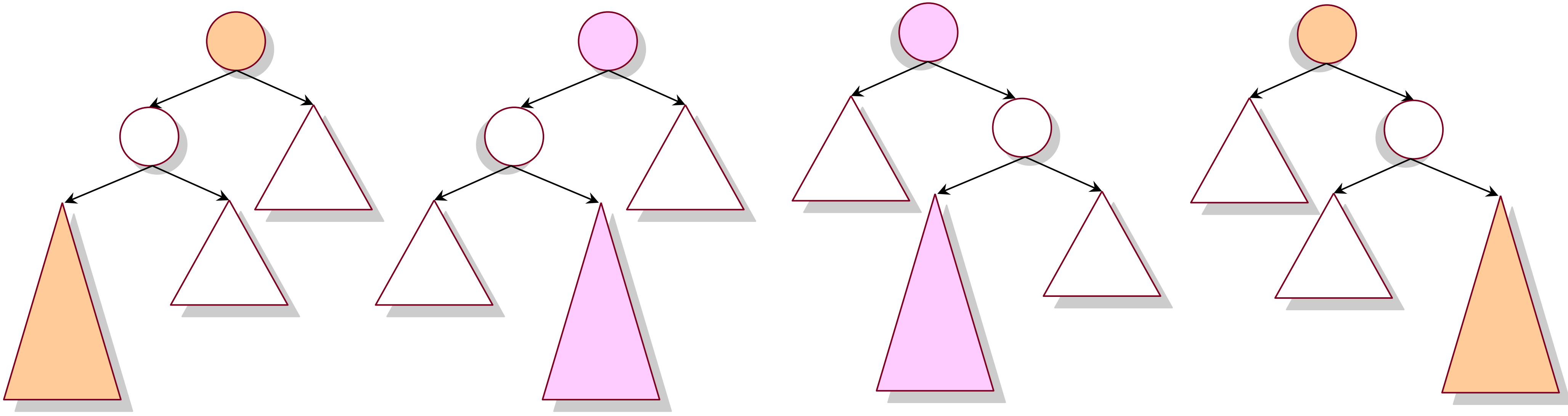
- Originally, the BST is height-balanced.
- Insert(6) violates AVL property
  - at node 8
- This is handled using *rotations*.
  - Exploits BST property which allows multiple structures for the same set of keys.
- **Observation**: Only nodes along the path from root to the new node have their subtrees altered.
  - Only these nodes may be checked for imbalance.
  - This means O(log N) rotations may be required.
  - We will show that only 2 rotations are sufficient.

AVL insert

- **Four cases:**
  1. insert into left subtree of left child
  2. Insert into right subtree of left child
  3. Insert into left subtree of right child
  4. Insert into right subtree of right child

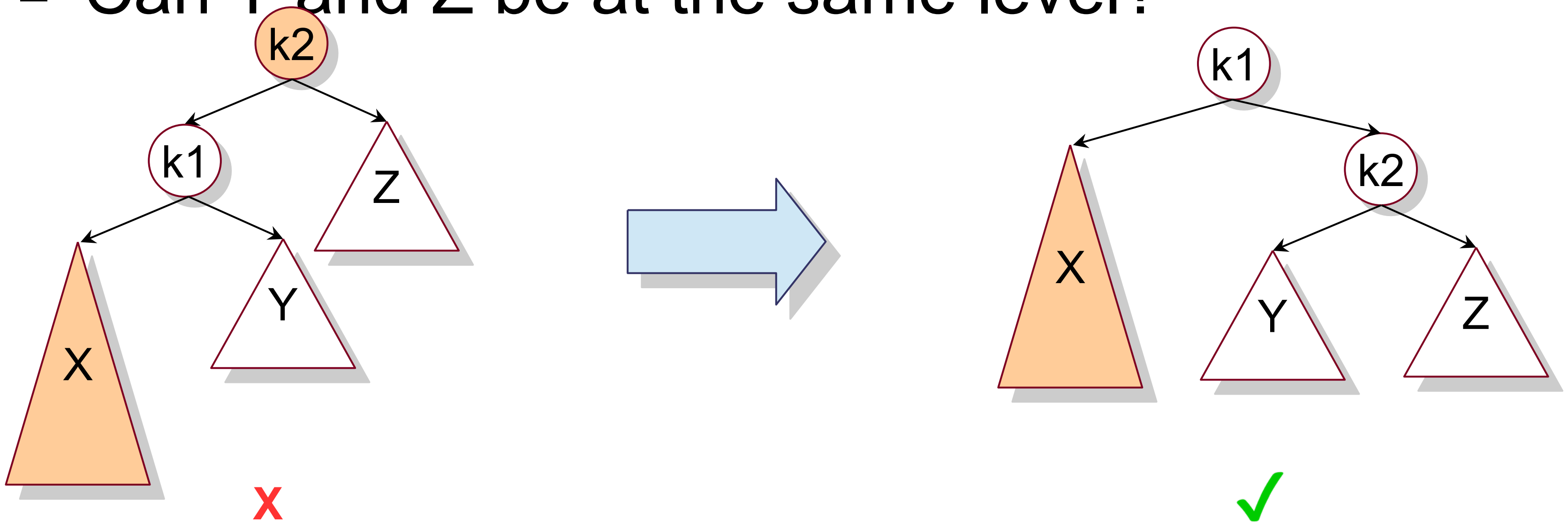
Double rotation

Single rotation



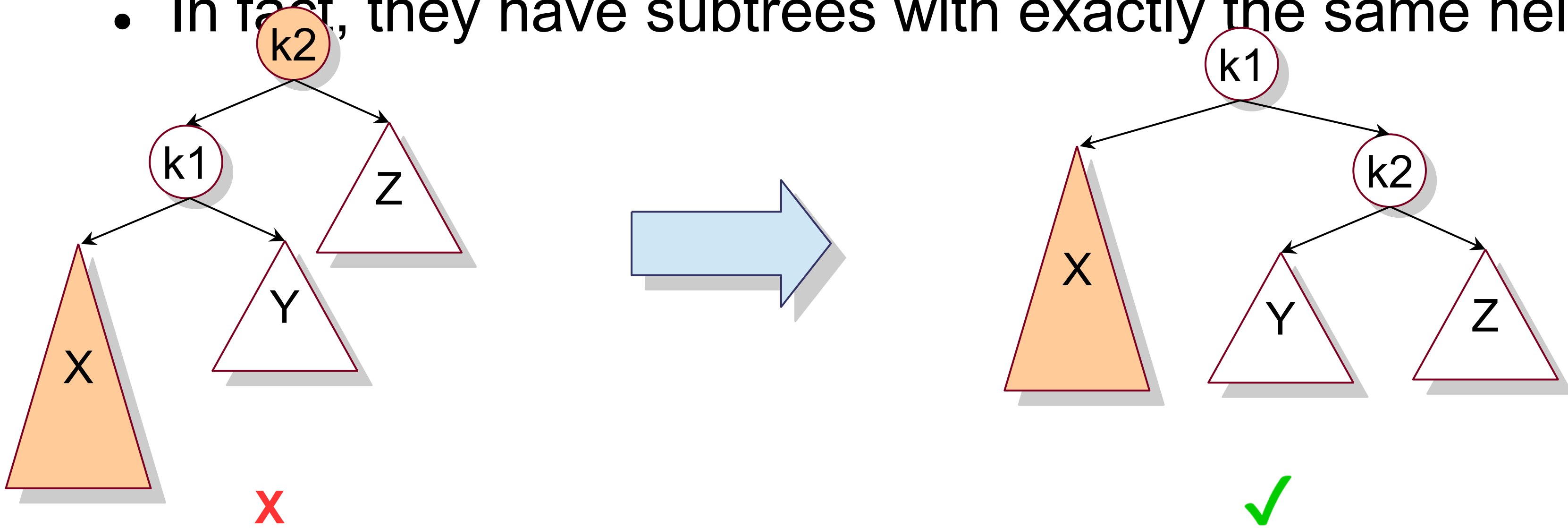
AVL insert: Case 1

- Insert into left subtree of left child
  - Before insertion into X, AVL property was satisfied.
  - Let k2 be the first node upward with imbalance.
  - Height(k2→left) – Height(k2→right) > 1
  - k1 continues to be height-balanced.
  - Can Y and Z be at the same level?

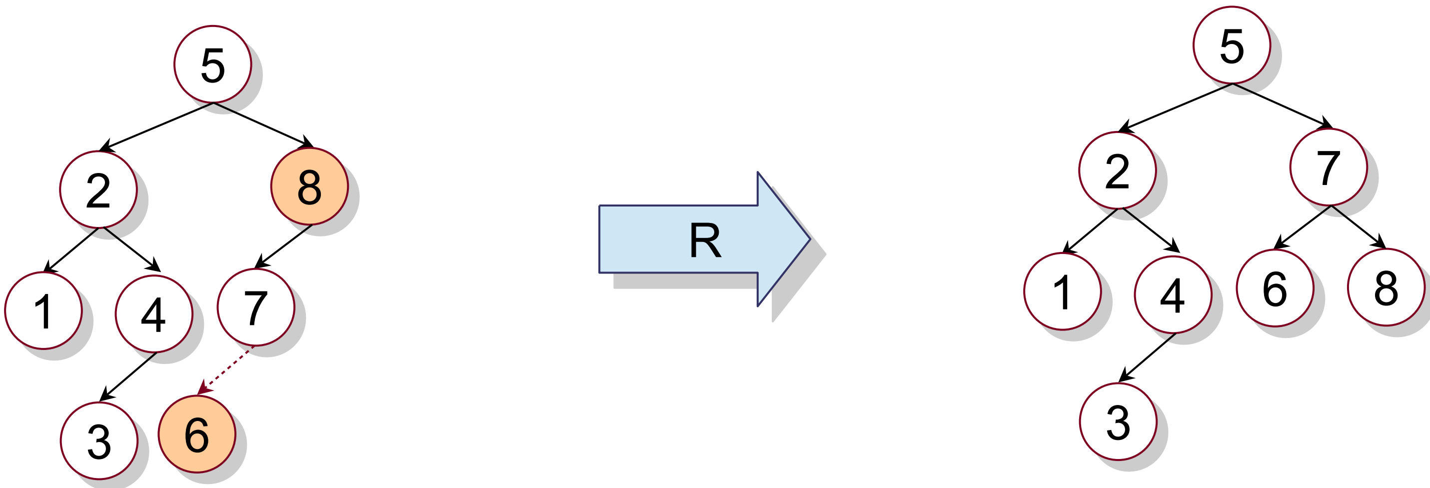


AVL insert: Case 1

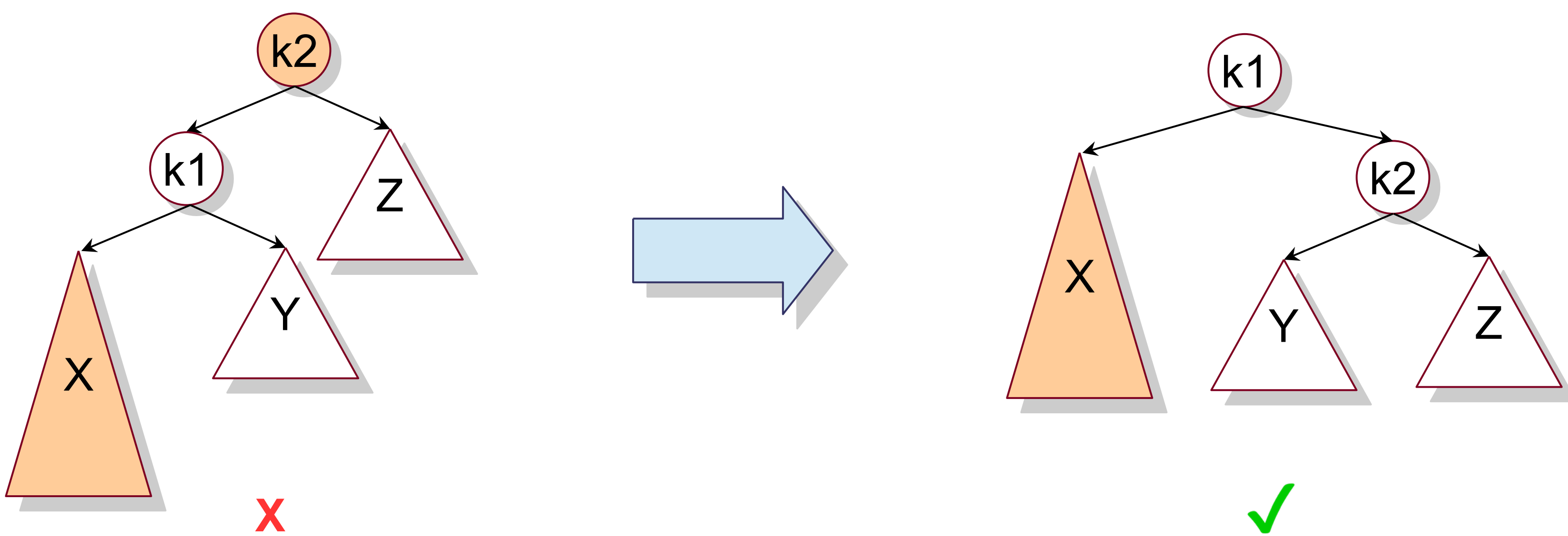
- Insert into left subtree of left child
  - Original order: X k1 Y k2 Z
  - New order: X k1 Y k2 Z
  - X moves up one level, Y stays at the same level, and Z moves down one level.
  - k1 and k2 satisfy AVL property.
    - In fact, they have subtrees with exactly the same height.



AVL insert: Case 1

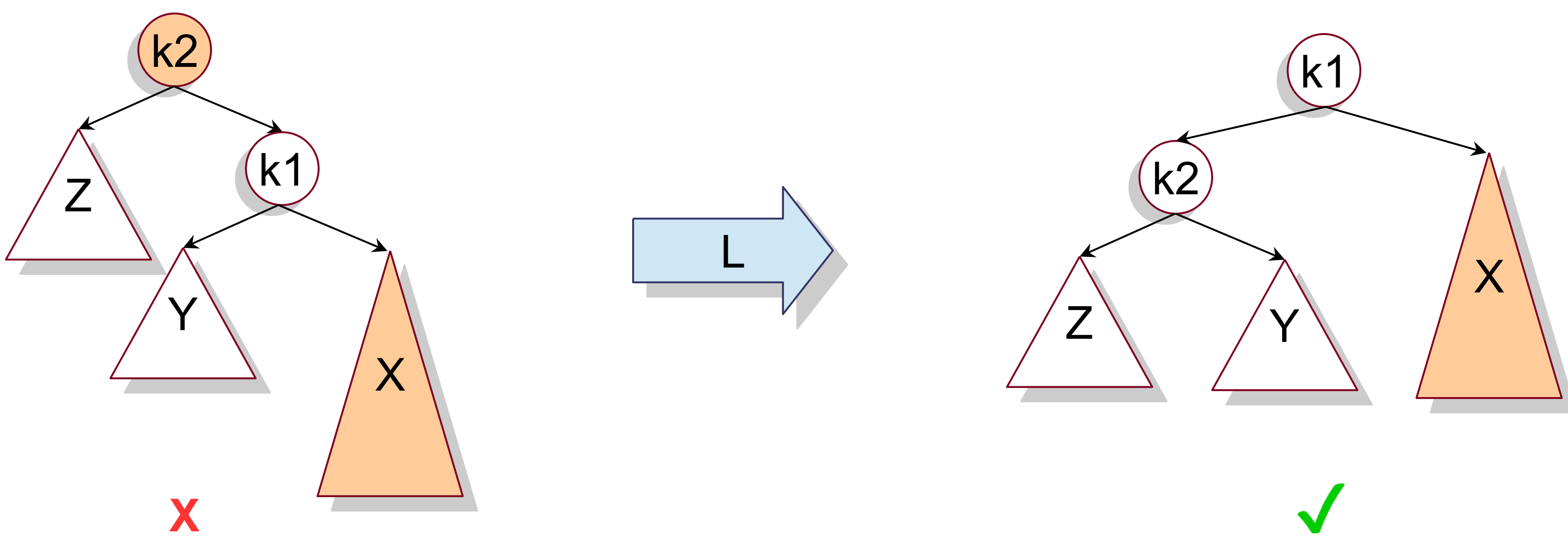


K1 == 7, k2 == 8, X is subtree rooted at 6, Y is empty, Z is empty.



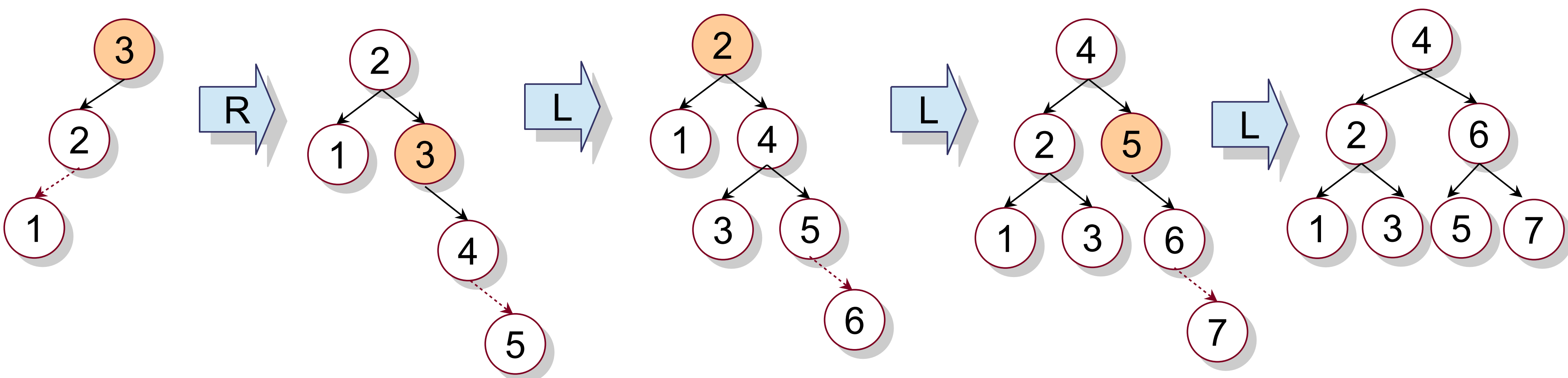
AVL insert: Case 4

- Scenario is symmetric to Case 1.
- Case 1 had a right rotation.
- Case 4 needs a left rotation.



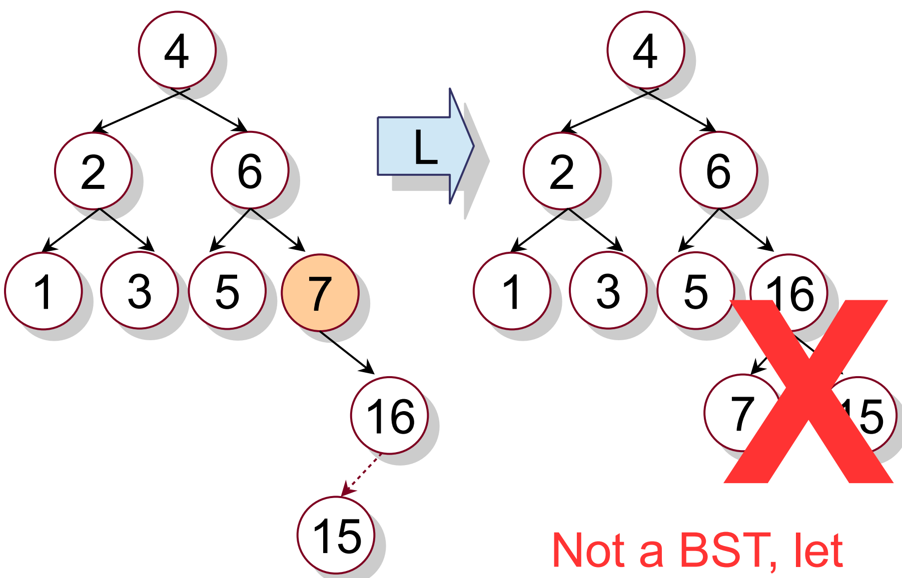
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



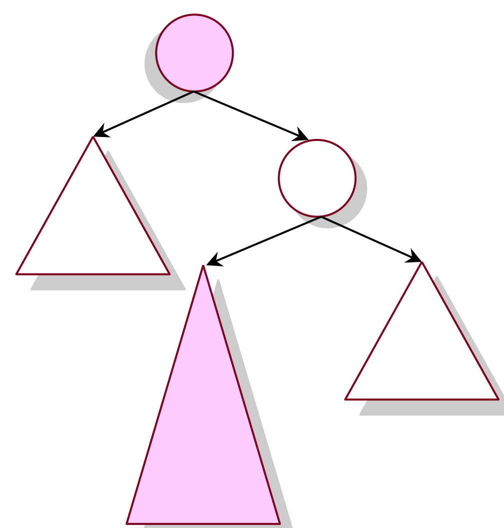
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



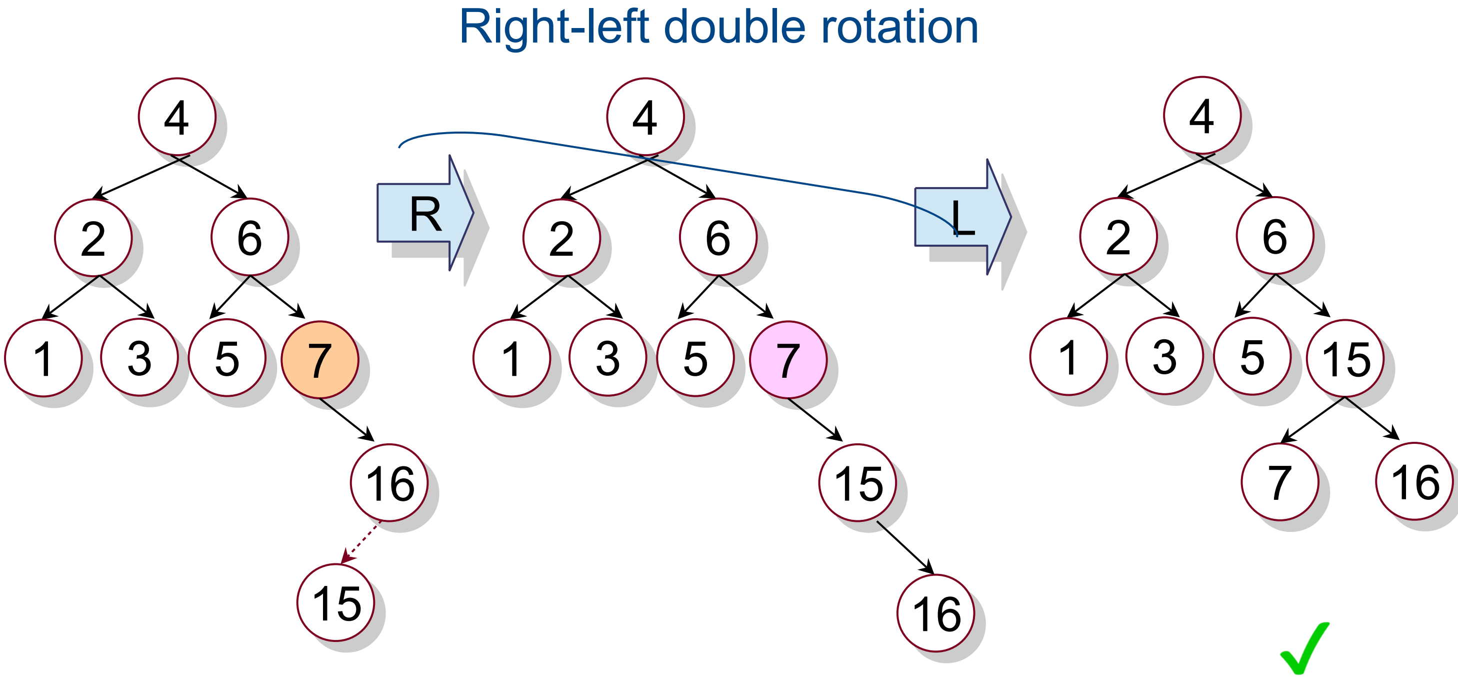
Not a BST, let
alone being AVL

The case is neither Case 1 (left-left)
nor Case 4 (right-right).
It is Case 3 (right-left).



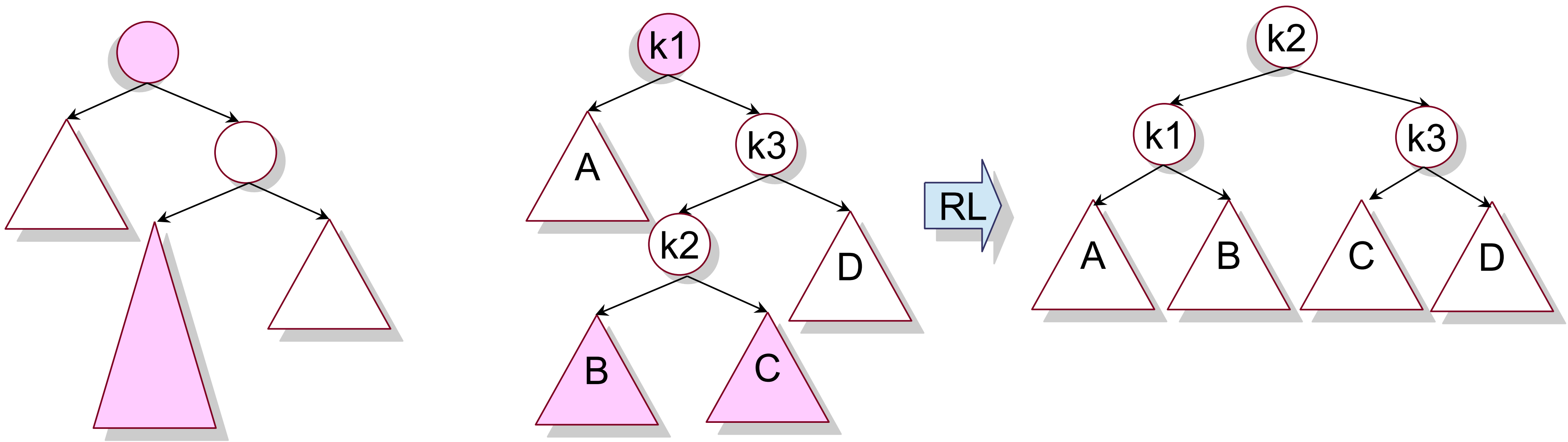
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



Example

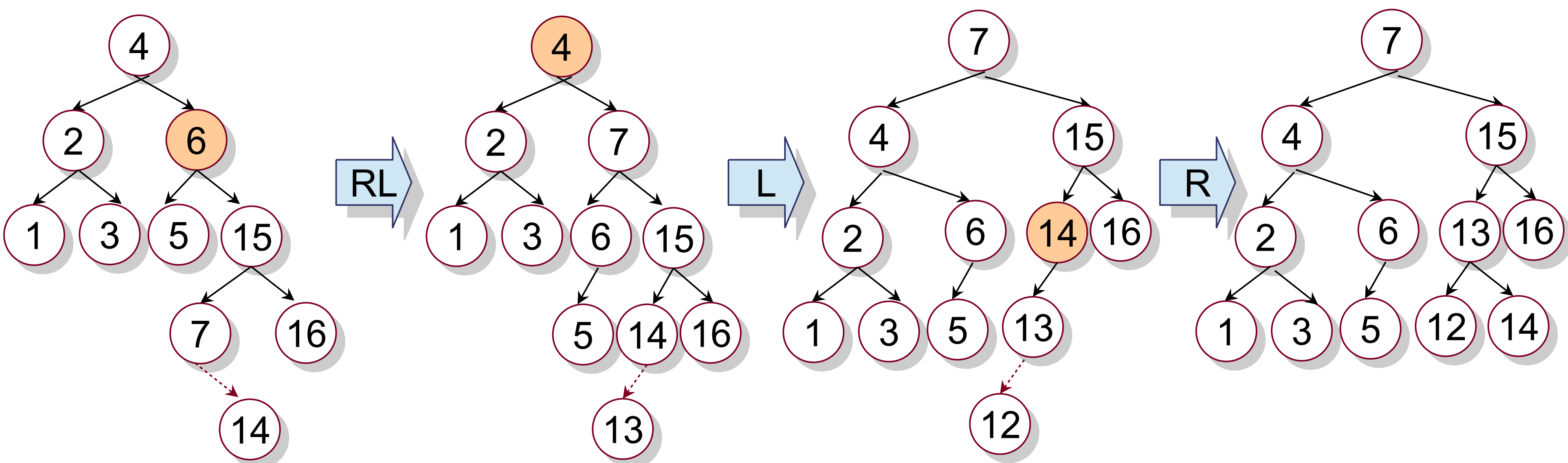
- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



BST Sequence: A k1 B k2 C k3 D

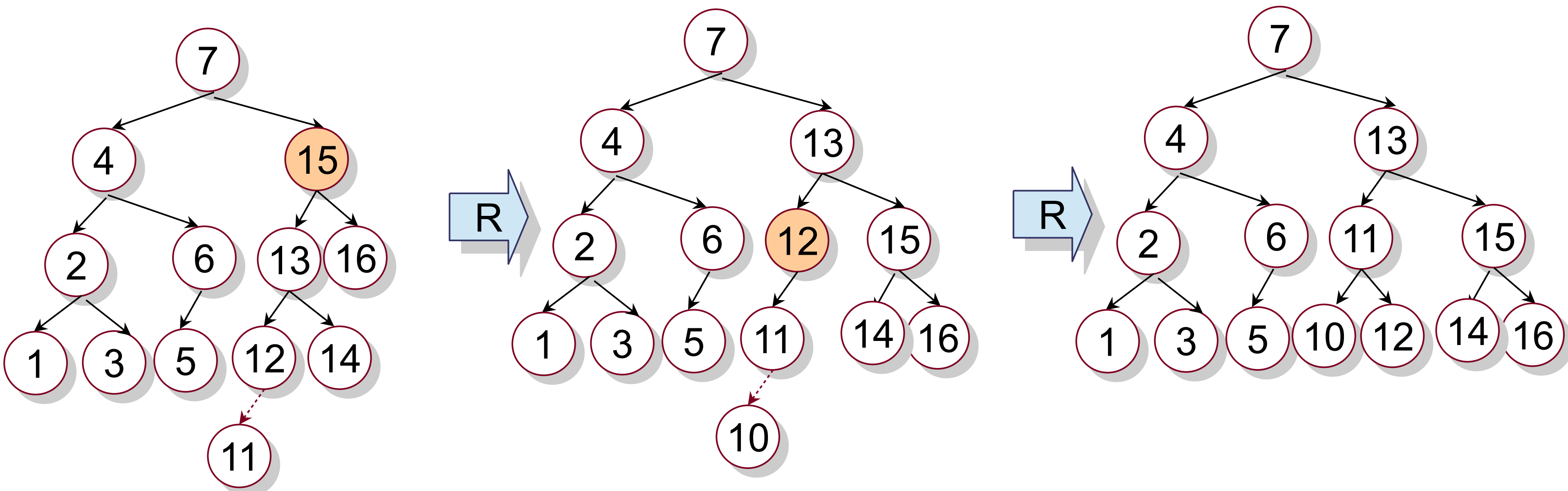
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



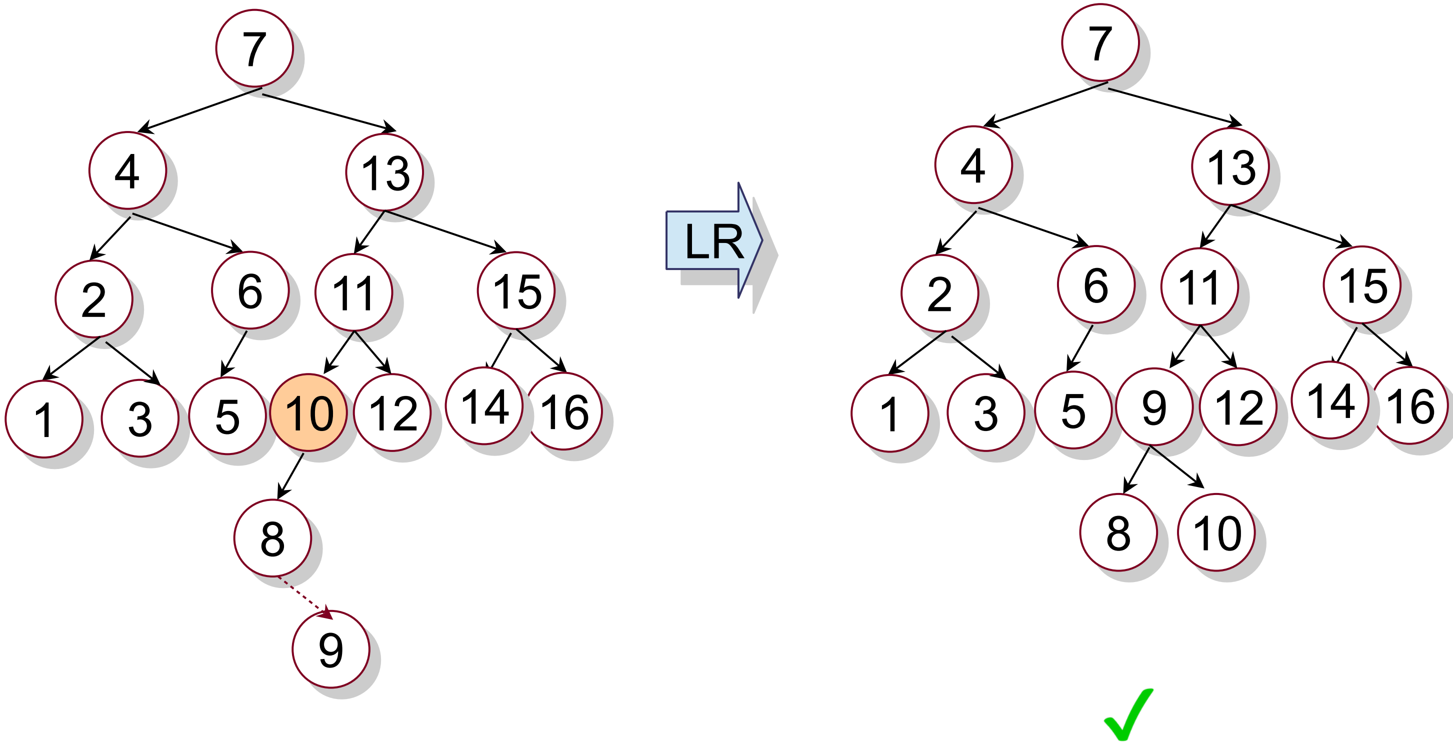
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



Source: avl.cpp

Classwork

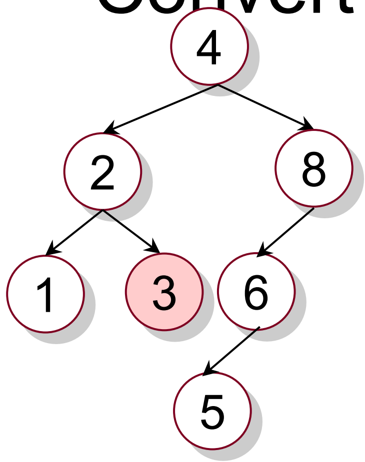
- In an empty AVL tree,
- insert 4, 10, 1, 2, 5, 7, 3, 6, 8, 9.

Deletion in AVL

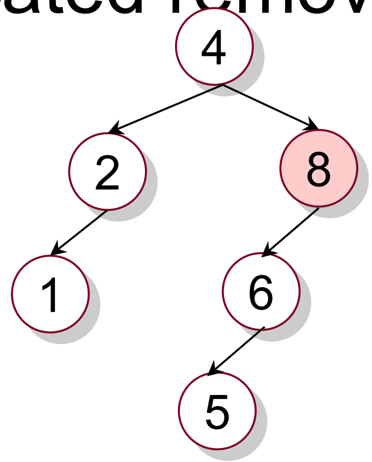
- Can be implemented lazily (marking).
  - Does not maintain AVL property.
- Inverse of insertion, so need to think backwards.
- But rotations would help us rebalance.
  - The four rotations can be called as primitives.
  - Deletion at internal node starts similar to as in BST. Gets converted to deletion at the leaf.
- Need to find taller of the two subtrees (left or right).
  - We can then rotate that subtree to rebalance.
- Unlike insertion, deletion may need to be repeated for all the ancestors.

Recall BST Remove(value)

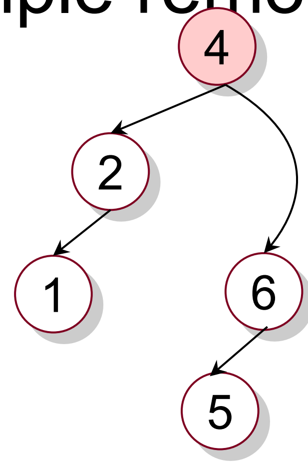
- Search for the node **n** to be removed.
- If **n** is a **leaf**, remove **n** from its parent.
- If **n** has **one** child **c**, make **c** the child of **n**'s parent.
- If **n** has **two** children, scratch your head.
  - Convert complicated remove to simple remove.



Remove(3)

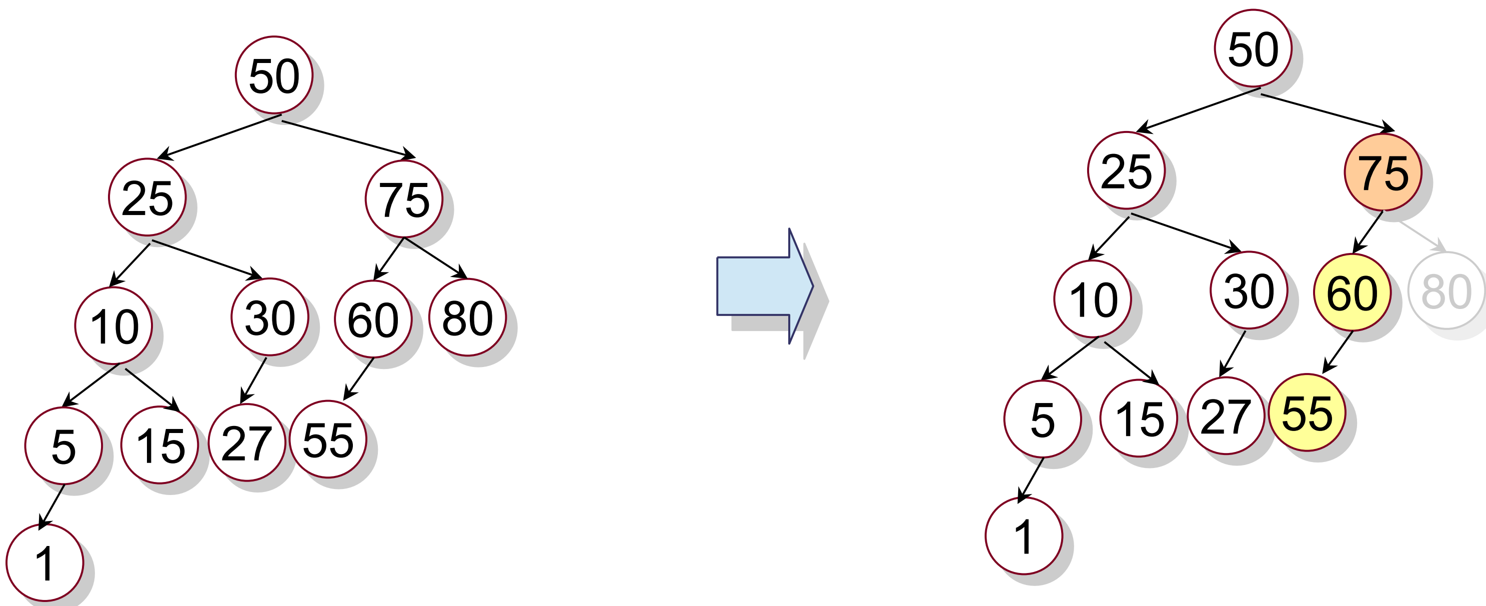


Remove(8)



Remove(4)

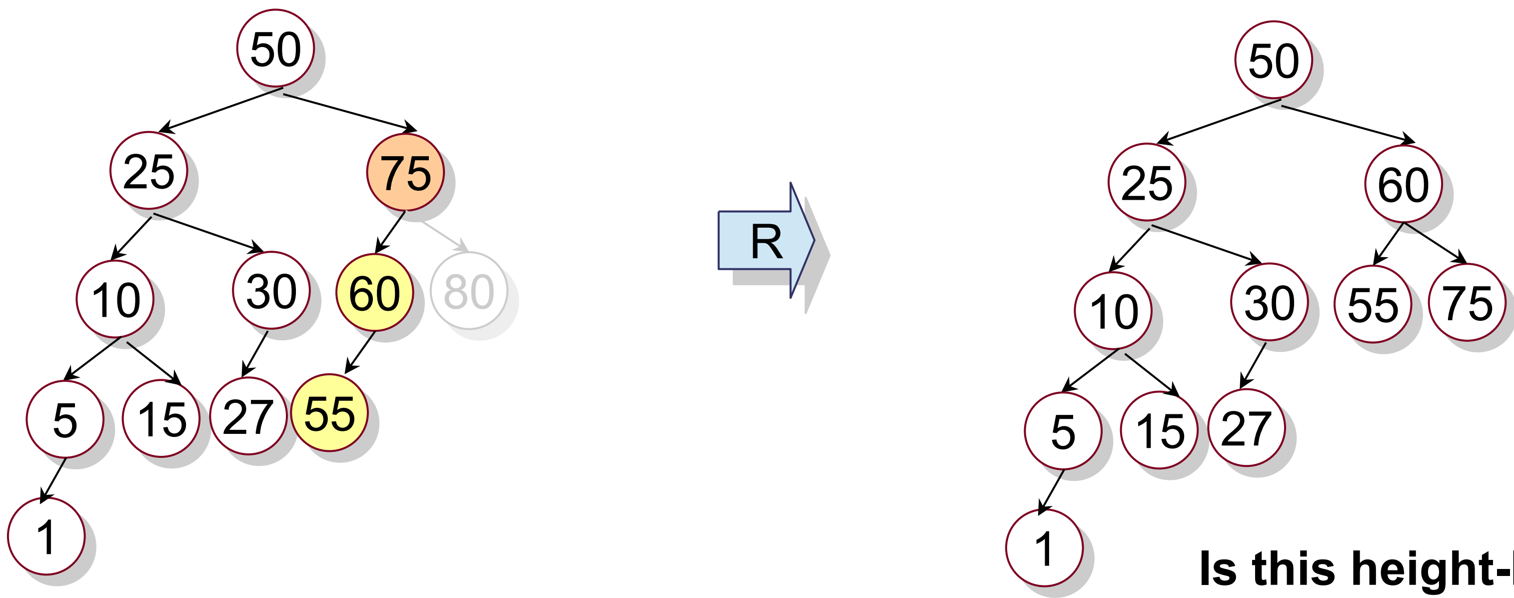
AVL Deletion Example



Delete 80

- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (75).
- Find x's tallest child y (60). Find y's tallest child z (55).
- Rotate x, y, z.

Deletion Example



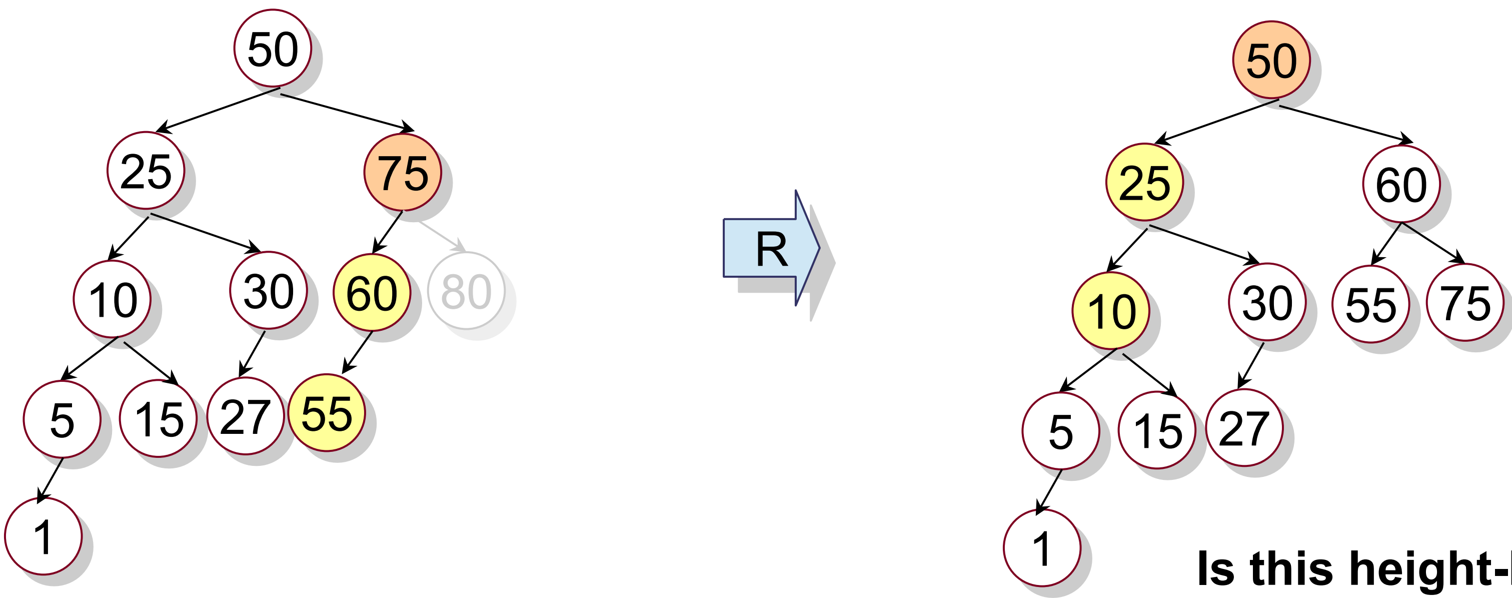
Is this height-balanced?
- Yes at 60, No at 50.

Delete 80

- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (75).
- Find x's tallest child y (60). Find y's tallest child z (55).
- Rotate x, y, z.

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html

Deletion Example



Is this height-balanced?
- Yes at 60, No at 50.

Delete 80

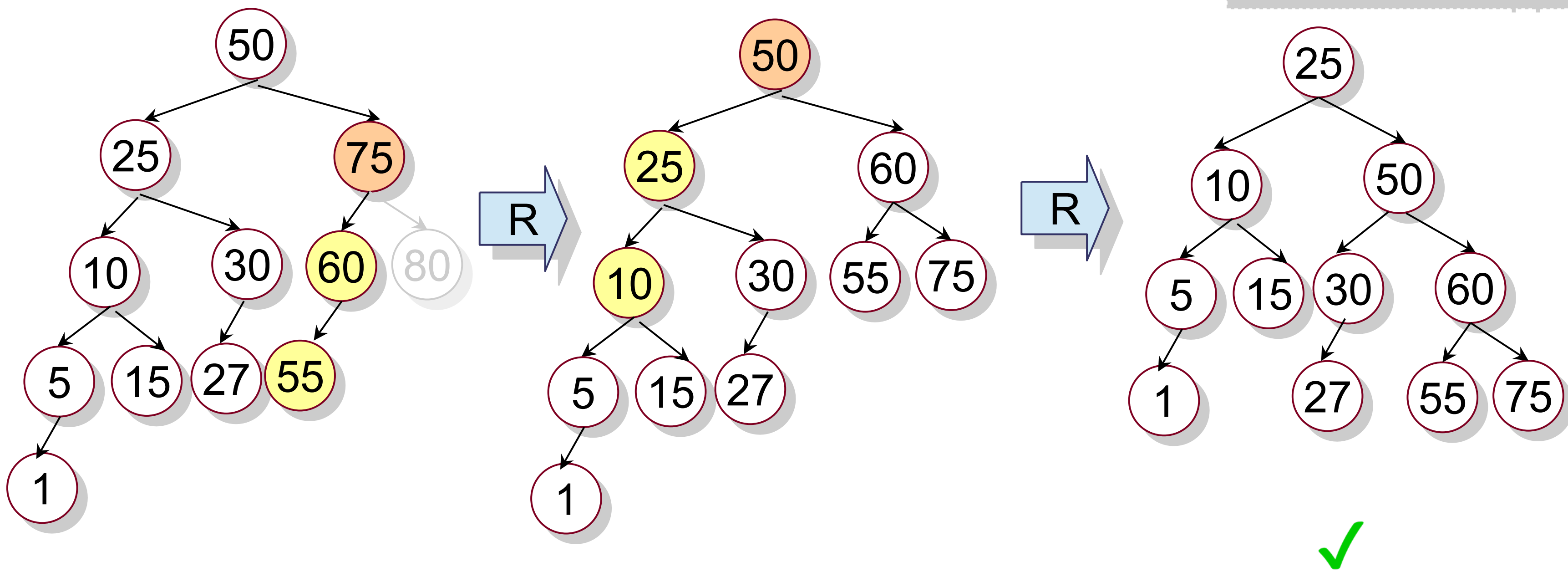
- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (50).
- Find x's tallest child y (25). Find y's tallest child z (10).
- Rotate x, y, z.

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html

Deletion Example

Classwork: Delete 75, 55, 60, 50, 27.

Source: avl.cpp



Delete 80

- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (50).
- Find x's tallest child y (25). Find y's tallest child z (10).
- Rotate x, y, z.

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html