I111E Algorithms & Data Structures
7. Data structure (3)
Binary Search Tree and its balancing

School of Information Science

Ryuhei Uehara & Giovanni Viglietta

uehara@jaist.ac.jp & johnny@jaist.ac.jp

2019-11-11

All materials are available at

http://www.jaist.ac.jp/~uehara/couse/2019/i111e

Announcement

- 1^{st} report: deadline is November 11, 10:50am.
- Mid term examination (30pts):
  - November 11, 13:30-15:10
  - Range: up to November 6
    - Bring copy of slides, and hand written notes

Reports so far…

- On 8:25am, November 11:

s1810235, s1810446, s1910192, s1910402, s1910416, s1910438, s1810403, s1910069, s1910231, s1910412, s1910421, s1910440, s1810433, s1910158, s1910401, s1910414, s1910436,

- Quick comments
  - We have sent confirmation email
  - Send your report to both of Giovanni and Ryuhei
  - File name should be, e.g., s1234567.pdf of one PDF file.

Review:

- We have three combinations of “data structure”, “what to do” and “algorithm”.
- “What to do”: E.g., i-th data, search, add/insert/remove.

- Array: access in $O(1)$, search in $O(n)$
- Array in order: search in $O(\log n)$, but add/remove in $O(n)$
- Linked list: access in $O(n)$, but add/remove in $O(1)$
- Hash: easy to add and search
- Binary search tree: dynamic search

Dynamic search and data structure

- Sometimes, we would like to search in dynamic data, i.e., we add/remove data in the data set.
- Example: Document management in university
  - New students: add to list
  - Alumni: remove from list
  - When you get credit: search the list

Q. Good data structure?

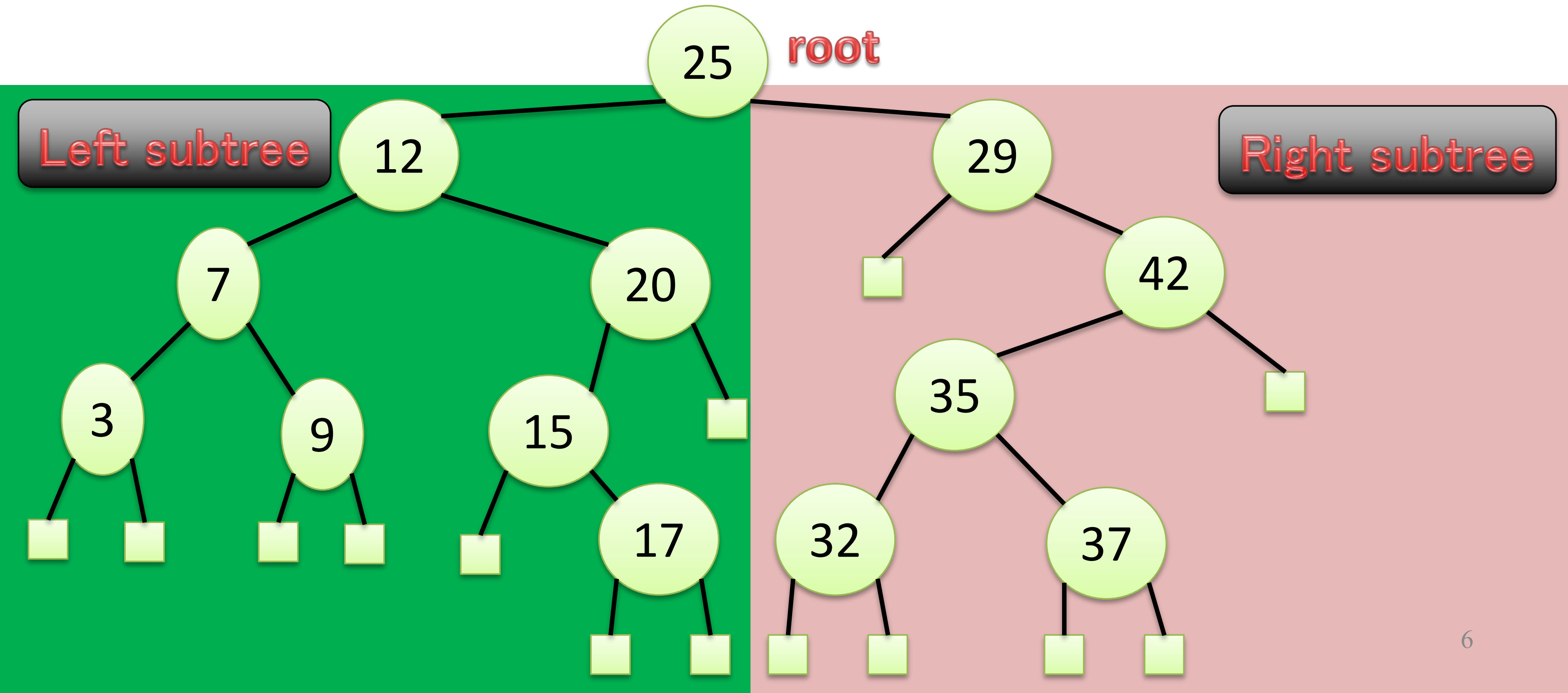
Naïve idea: array or linked list?

- Data in order:
  - Search: binary search in O(log n) time
  - Add and remove: O(n) time per data
- Data not in order:
  - Search and remove: O(n) time per data
  - Add: in O(1) time

Imagine: you have 10000 students, and you have 300 new students!

Better idea: binary search tree

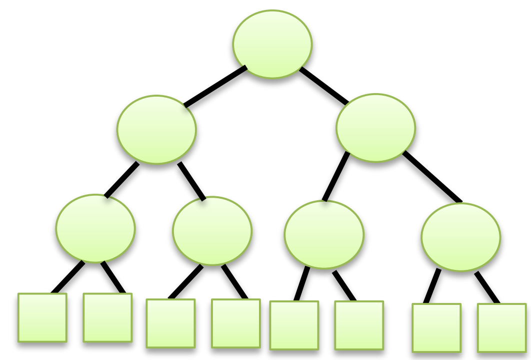
- For every vertex v, we have the following;
  - Data in v ≧ any data in a vertex in left subtree
  - Data in v ≦ any data in a vertex in right subtree



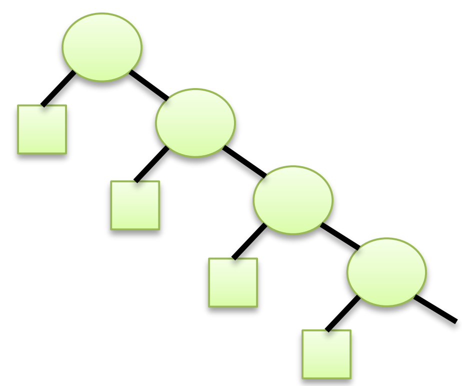
Better idea: binary search tree

- We construct binary search tree for a given data set; we learnt it can be updated in O(L) time, where L is the length of the route from a leaf to the root.

- When data is random:
  - Depth of the tree: O(log n)
  - Search, add, remove: O(log n) time.



- In the worst case:
  - Depth of the tree: n
  - When data is given in order, we have the worst case.
  - Search, add, remove: O(n) time…



Today: More binary search tree (BST)

1. Get maximum/minimum data (⇔ heap)
2. Enumerate all data in the tree（⇔ array）
3. “Good” and “bad” structure?
4. How can we fix bad to good?

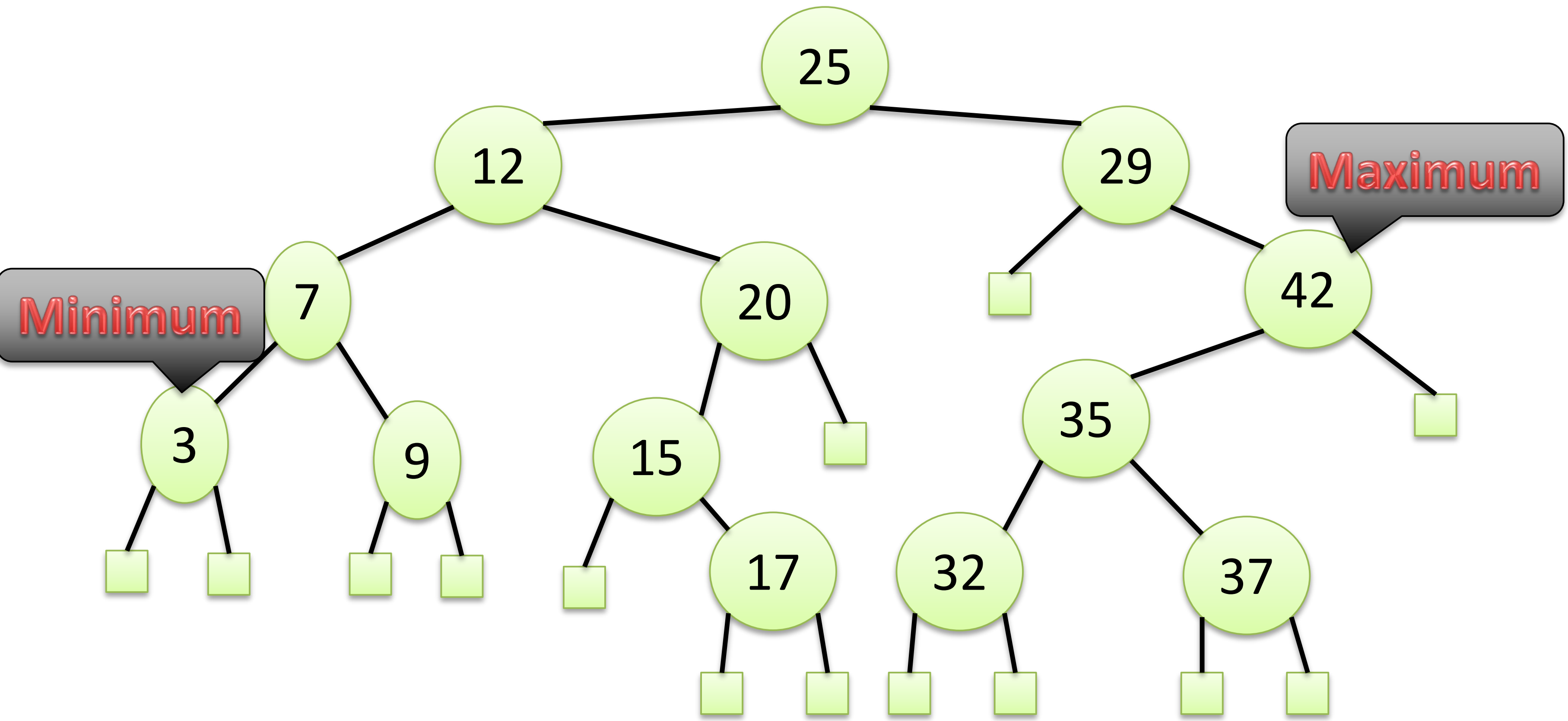
1. Max/min data in BST

- Properties of a BST
  - All left descendants have smaller values
  - All right descendants have larger values
- Using the properties…
  - Minimum: the leftmost lowest descendant from the root
  - Maximum: the rightmost lowest descendant from the root

- Tips: It is easy to remove the minimum/maximum node (since it has at most one child)

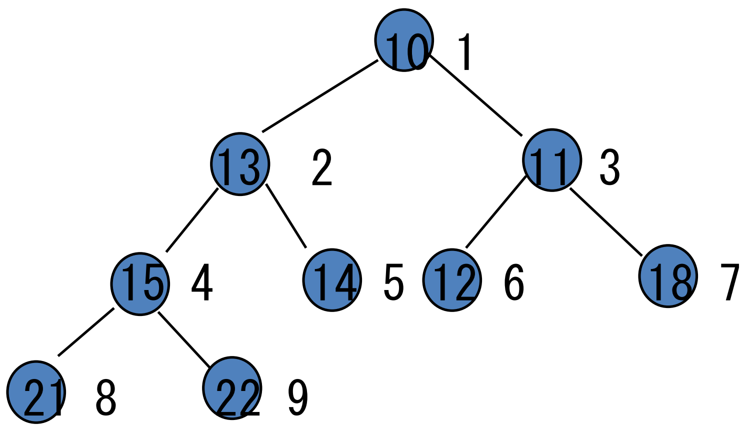
1. Max/min data in BST (Example)

(consider remove them also)



[Review]

How about heap?



1. Assign 1 to the root.
2. For a node of number i, assign 2 × i to the left child and assign 2 × i+1 to the right child.
3. No nodes assigned by the number greater than n.
4. For each edge, parent stores data smaller than one in child.

We can use <u>an array</u>, instead of linked list!

1	2	3	4	5	6	7	8	9
10	13	11	15	14	12	18	21	22

- It is easy to obtain the minimum one (at root)
- However, maximum one is not easy in the tree/array

Today: More binary search tree (BST)

1. Get maximum/minimum data (⇔ heap)
2. Enumerate all data in the tree（⇔ array）
3. “Good” and “bad” structure?
4. How can we fix bad to good?

We have three ways of enumeration

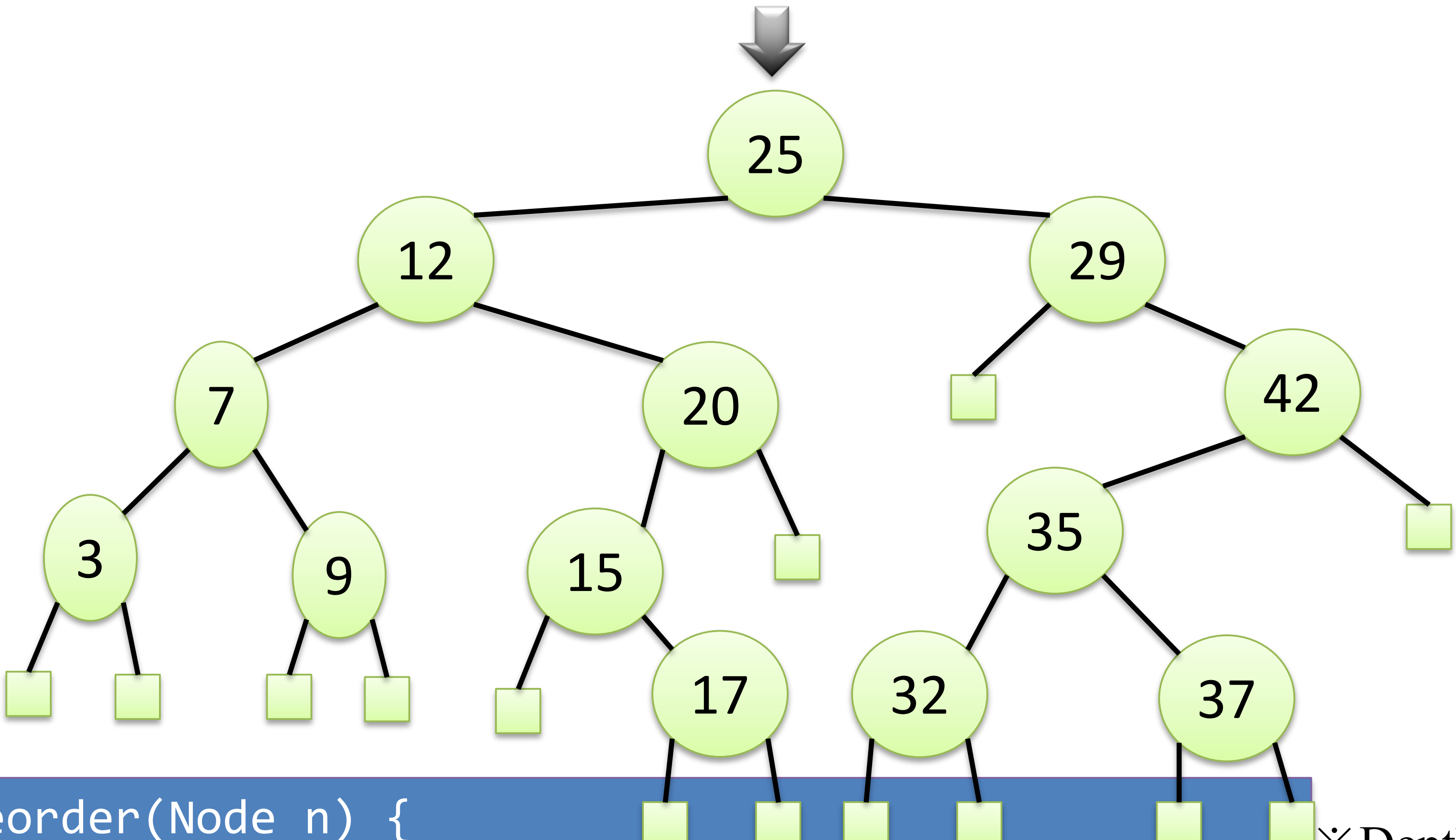
(general traverse ways of a binary tree)

- Preorder:
  Data in the current node → left subtree → right subtree
- Inorder:
  left subtree → Data in the current node → right subtree
- Postorder:
  left subtree → right subtree → Data in the current node

※It is easy to enumerate all data in array or linked list

How to traverse binary tree: preorder

Data in node → left subtree → right subtree



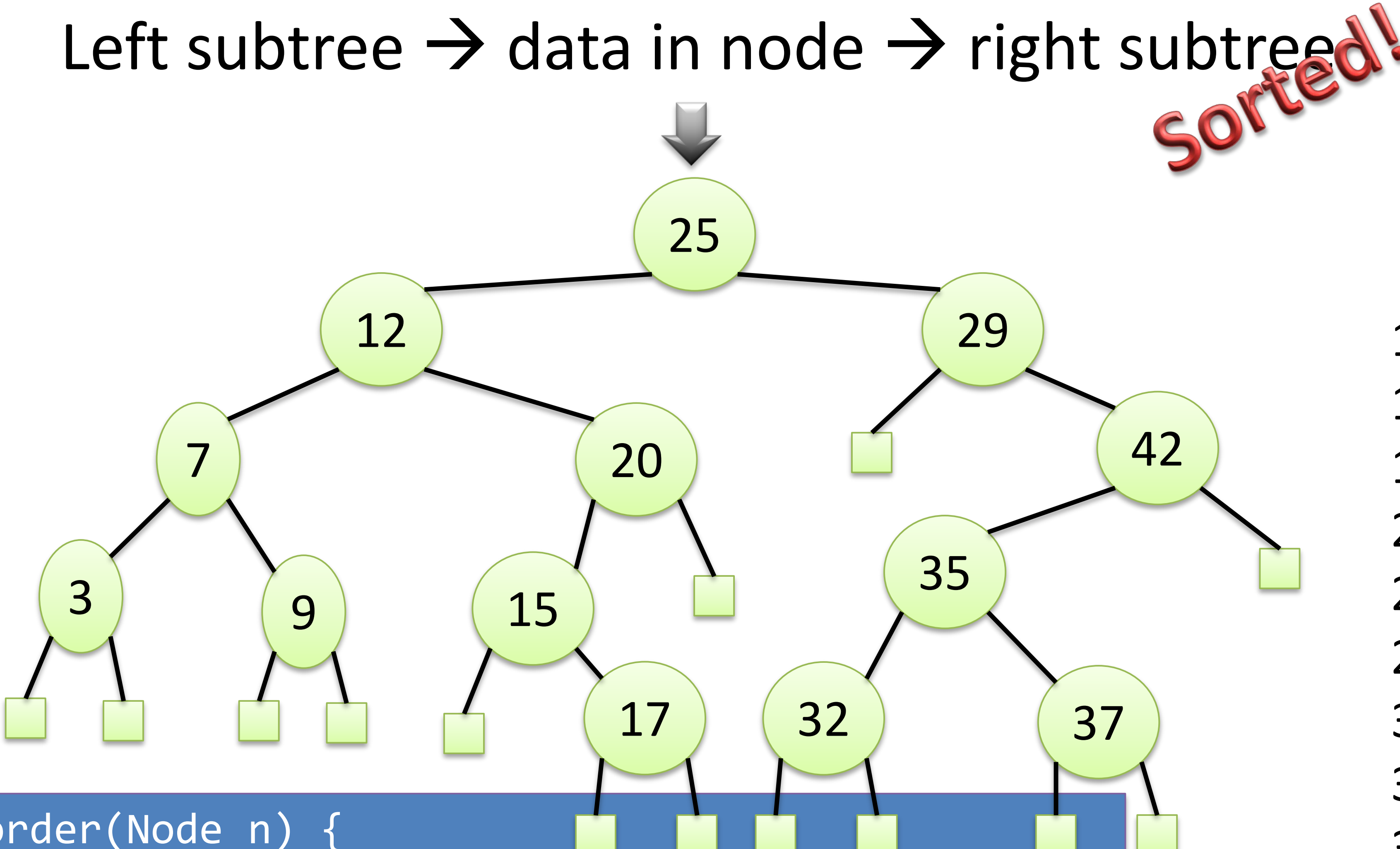
25
12
7
3
9
20
15
17
29
42
35
32
37

```
preorder(Node n) {
    if (n==null) return;
    visit(n); preorder(n.lson); preorder(n.rson);
}
```

※Depth first manner

How to traverse binary tree: inorder

Left subtree → data in node → right subtree

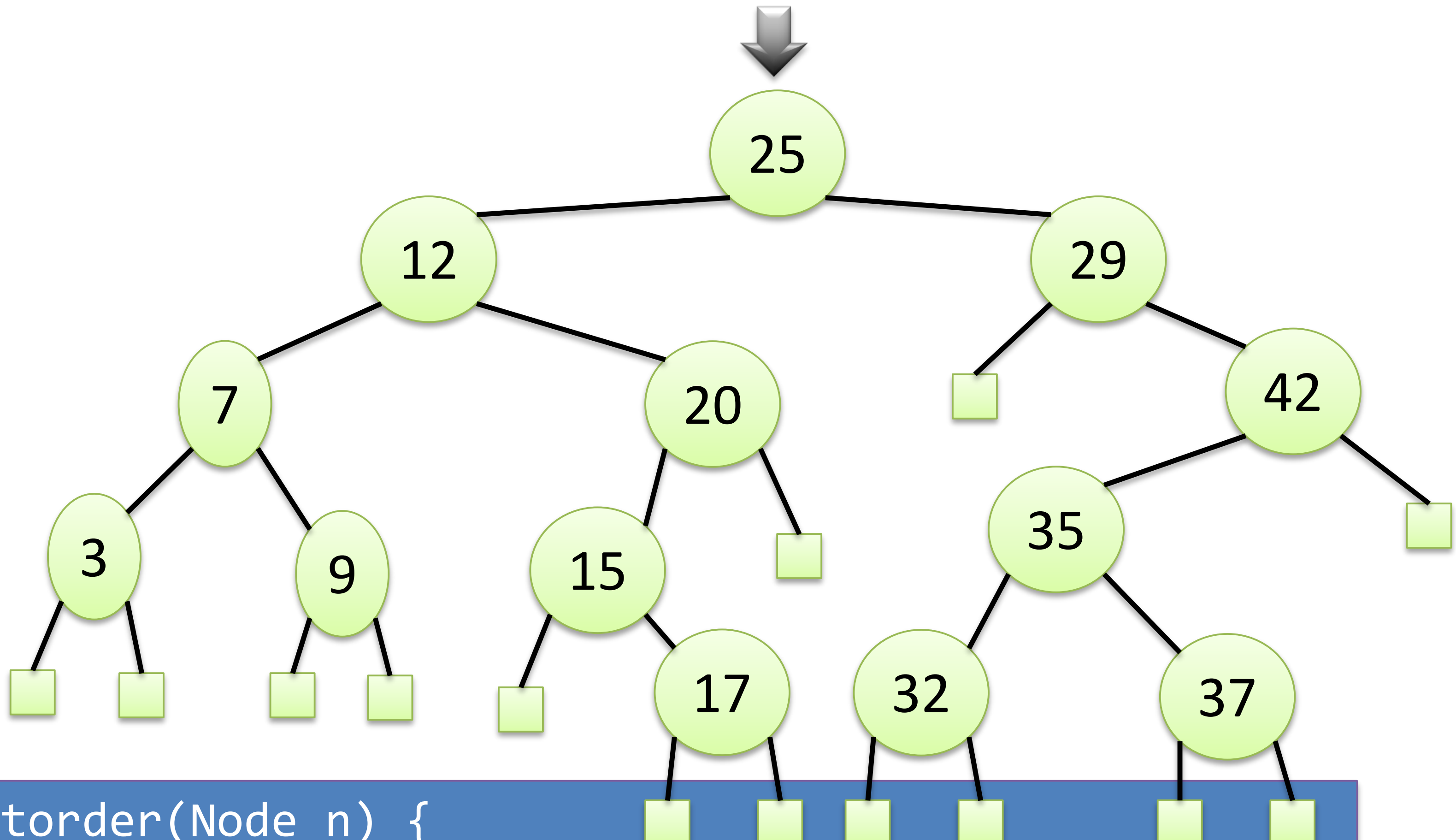


3
7
9
12
15
17
20
25
29
32
35
37
42

```
inorder(Node n) {
    if (n==null) return;
    inorder(n.lson); visit(n); inorder(n.rson);
}
```

How to traverse binary tree: postorder

Left subtree → right subtree → data in node



3
9
7
17
15
20
12
32
37
35
42
29
25

```
postorder(Node n) {
    if (n==null) return;
    postorder(n.lson); postorder(n.rson); visit(n);
}
```

Example of code

```
public class I111_08_p22{
    public static void Main(){
        Node n3 = new Node (3, null, null);
        Node n9 = new Node (9, null, null);
        Node n7 = new Node (7, n3, n9);
        Node n17 = new Node (17, null, null);
        Node n15 = new Node (15, null, n17);
        Node n20 = new Node (20, n15, null);
        Node n12 = new Node (12, n7, n20);
        Node n32 = new Node (32, null, null);
        Node n37 = new Node (37, null, null);
        Node n35 = new Node (35, n32, n37);
        Node n42 = new Node (42, n35, null);
        Node n29 = new Node (29, null, n42);
        Node n25 = new Node (25, n12, n29);

        inorder(n25);
    }

    static void inorder(Node n) {
        if (n==null) return;
        inorder(n.lson);
        visit(n);
        inorder(n.rson);
    }

    static void visit(Node n) {
        System.Console.Write(n.data+" ");
    }
}
```

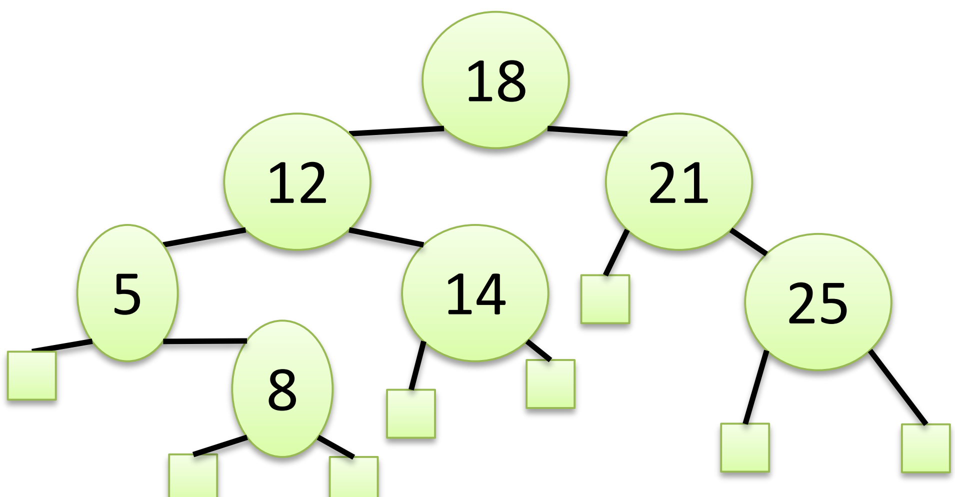
Easy to modify to pre, post

output

```
public class Node {
    public int data;
    public Node lson;
    public Node rson;
    public Node (int i, Node ls, Node rs) {
        data = i;
        lson = ls;
        rson = rs;
    }
}
```

Small exercise

- Make a small binary search tree (around 10 nodes)
- Find the maximum and minimum data
- Remove the root node
- Enumerate data in preorder, inorder, and postorder

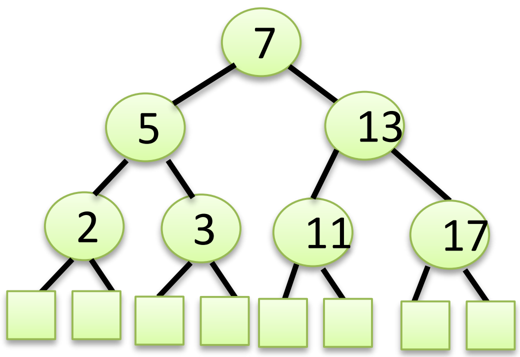


Today: More binary search tree (BST)

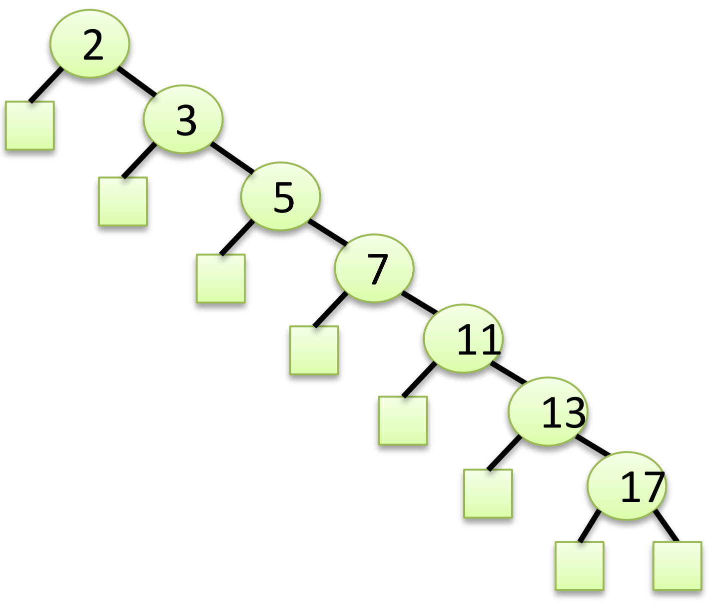
1. Get maximum/minimum data (⇔ heap)
2. Enumerate all data in the tree（⇔ array）
3. “Good” and “bad” structure?
4. How can we fix bad to good?

Efficiency of BST

- Best case: O(log n)
  - Each of n data is kept in BST of depth $\log_2 n$



- Worst case: O(n)
  - If we put in increasing order→ we have depth n



- "Random order" is also interesting topic, but we make it of depth O(log n) in any case.

Nice idea: (Self-)Balanced Binary Search Tree

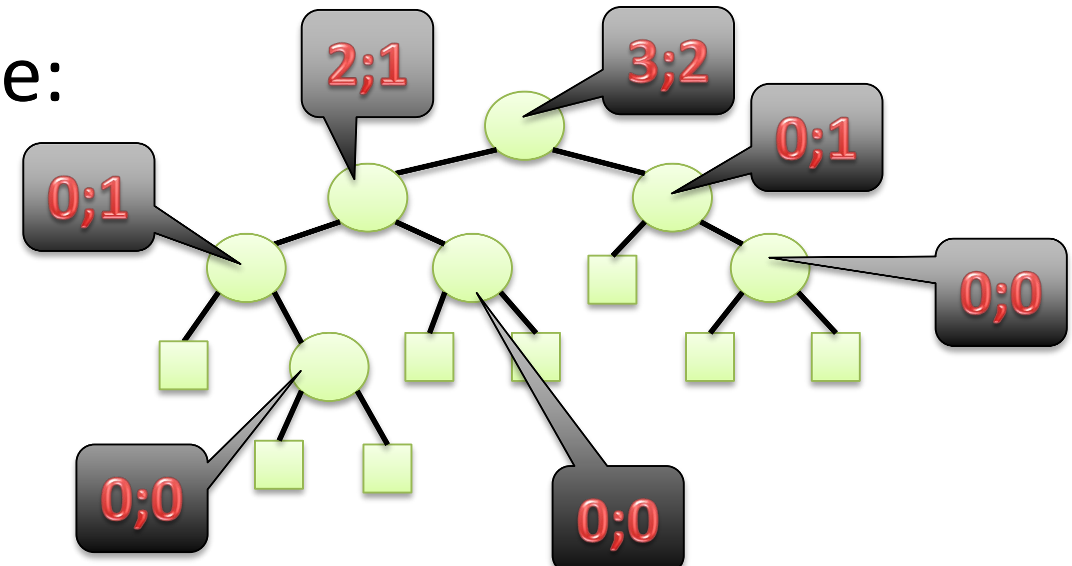
- There are some algorithms that maintain to take balance of tree in depth $O(\log n)$.
  - e.g., AVL tree, 2-3 tree, 2-color tree (red-black tree)

Georgy M. Adelson-Velsky
(1922−2014)

Evgenii M. Landis
(1921−1997)

AVL tree [G.M. Adelson-Velskii and E.M. Landis '62]

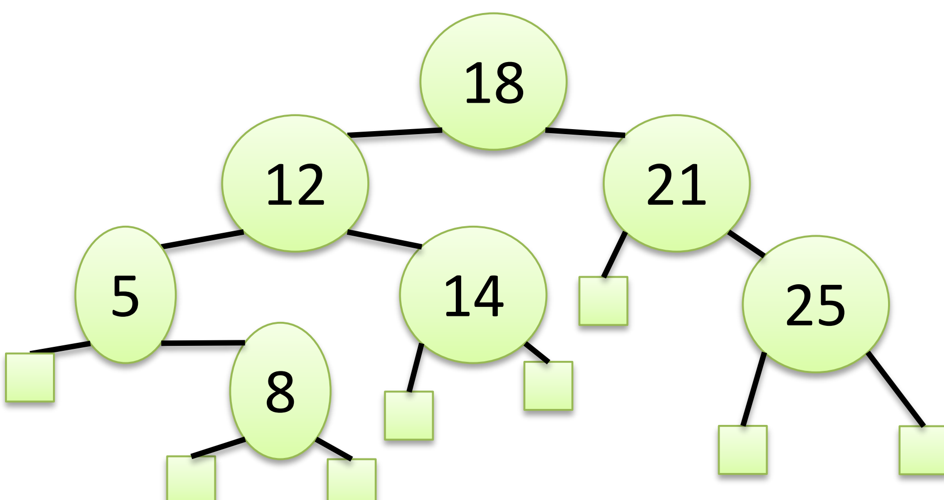
- Property (or assertion): at each vertex, the depth of left subtree and right subtree differs at most 1.
- Example:



AVL tree: Insertion of data

- Find a leaf v for a new data x
- Store data x into v (v is not a leaf any more)
- Check the change of balance by insertion of x
- From v to the root, check the balance at each vertex, and rebalance (rotation) if necessary.

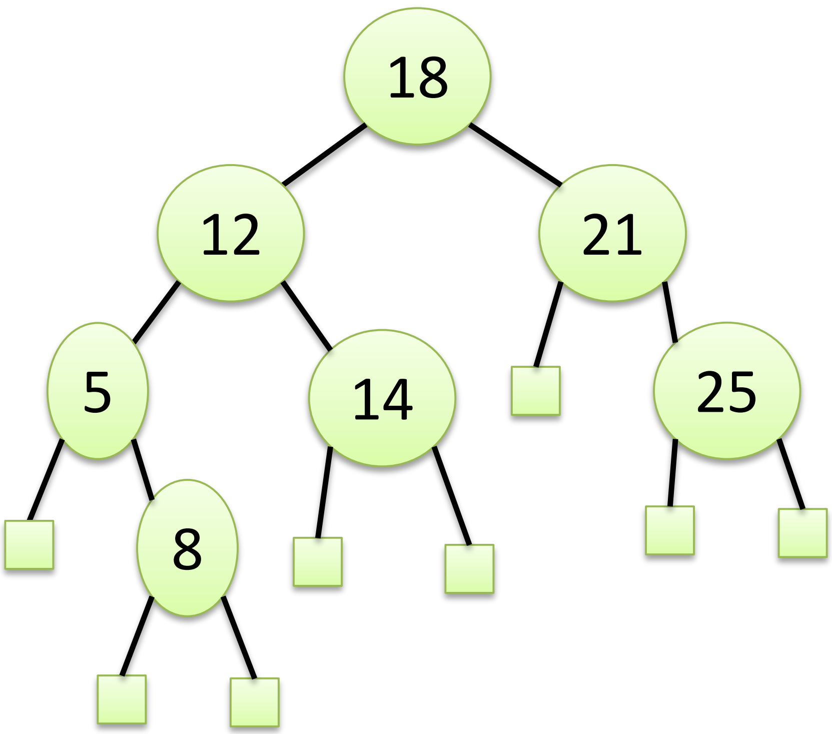
We have nothing to do up to here



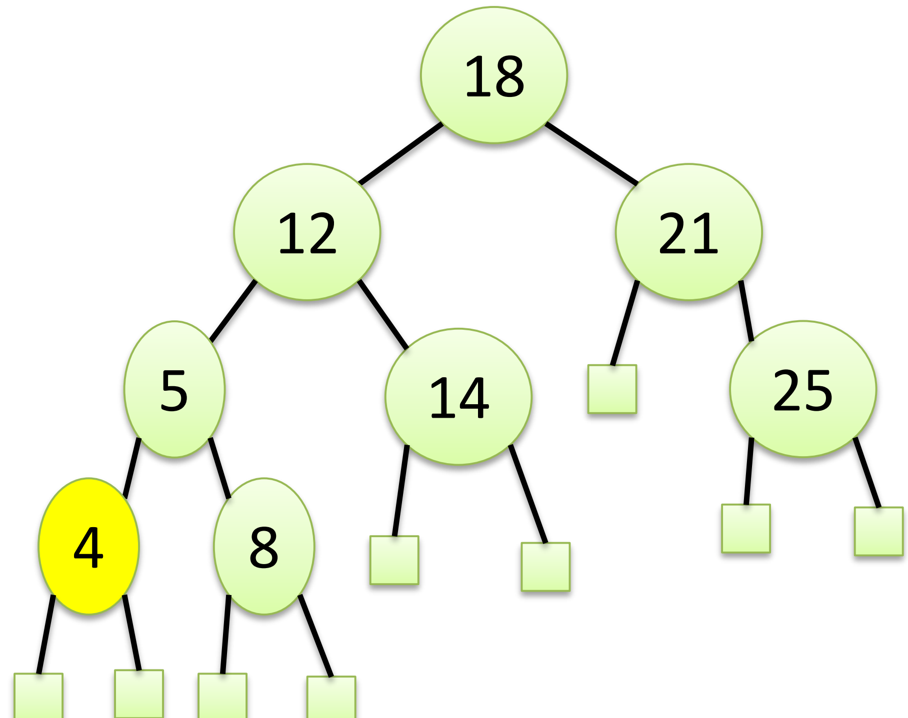
What happens if you insert x=4? How about x=10, x=20, x=23?

AVL tree: Insertion of data
Insert x=4

before



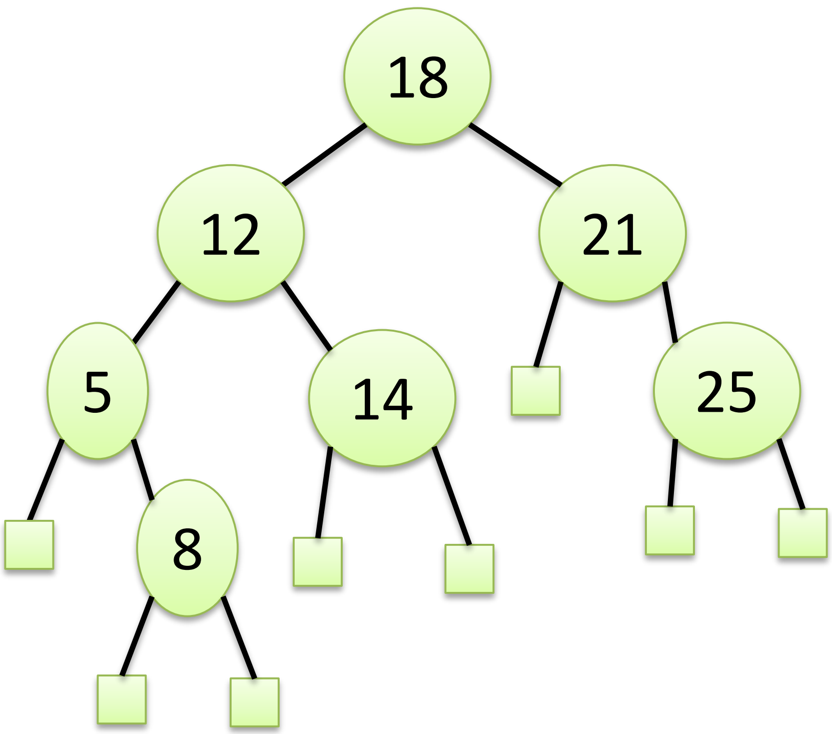
after



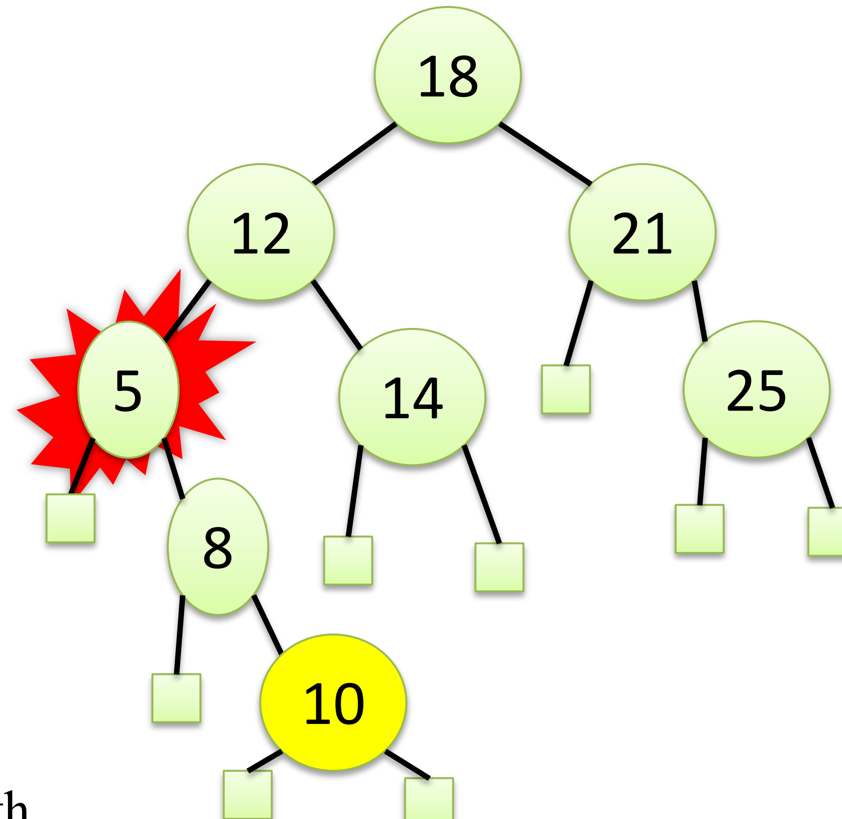
Balance: OK

AVL tree: Insertion of data
Insert x=10

before



after

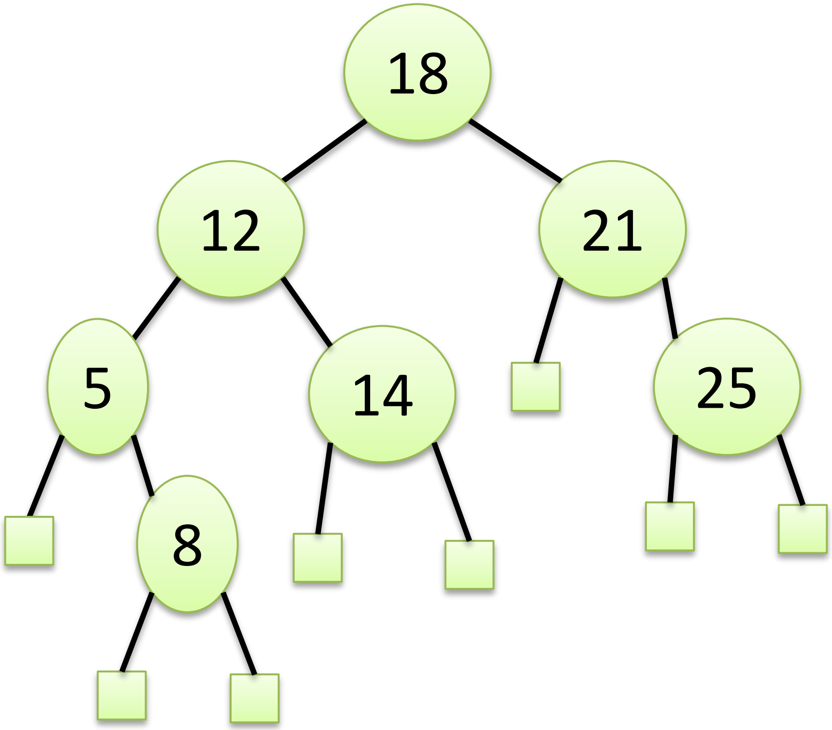


※We also have unbalanced at 12 and 18 with 1:3 and 2:4, resp, but we first handle the deepest point

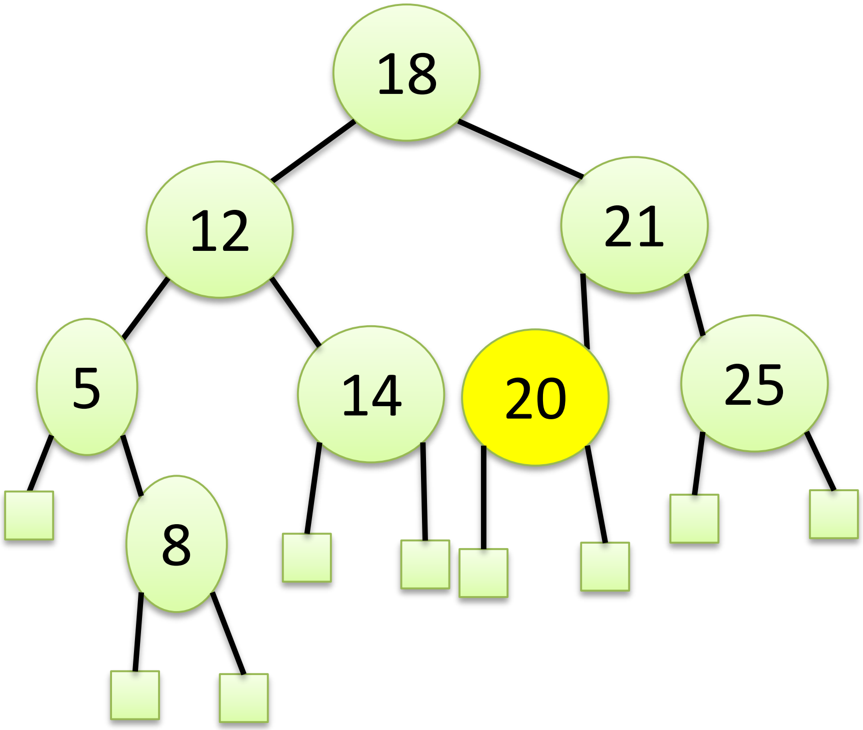
0;2@vertex 5

AVL tree: Insertion of data
Insert x=20

before



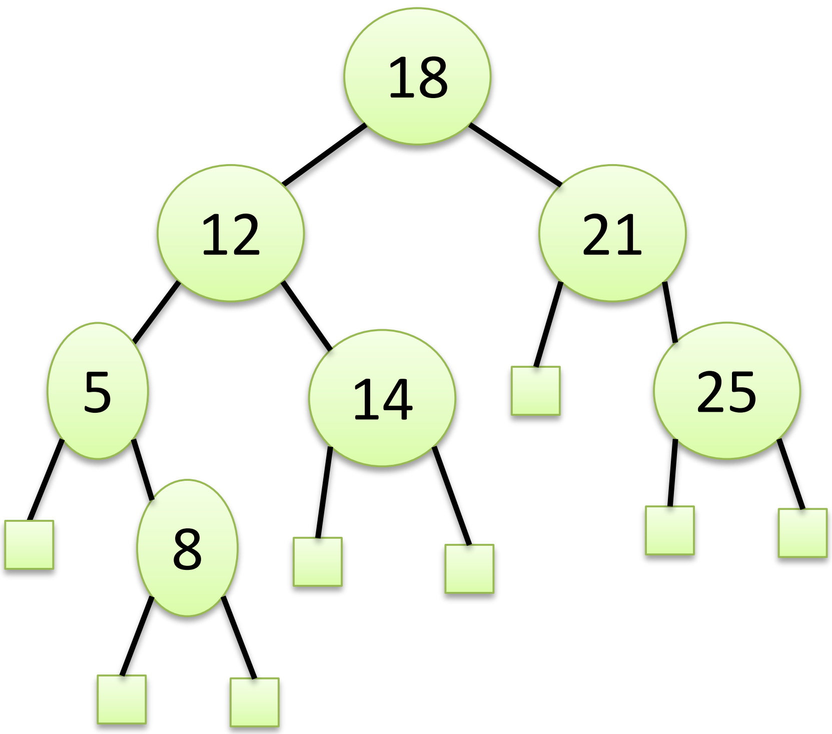
after



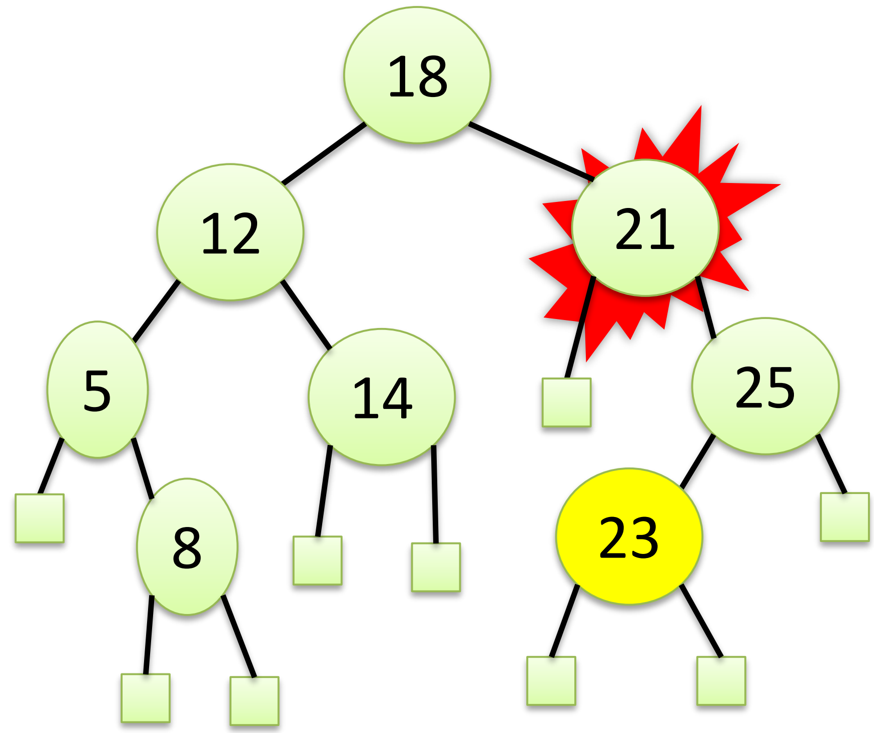
Balance: OK

AVL tree: Insertion of data
Insert x=23

before



after



0;2@vertex 21

Today: More binary search tree (BST)

1. Get maximum/minimum data (⇔ heap)
2. Enumerate all data in the tree（⇔ array）
3. “Good” and “bad” structure?
4. How can we fix bad to good?

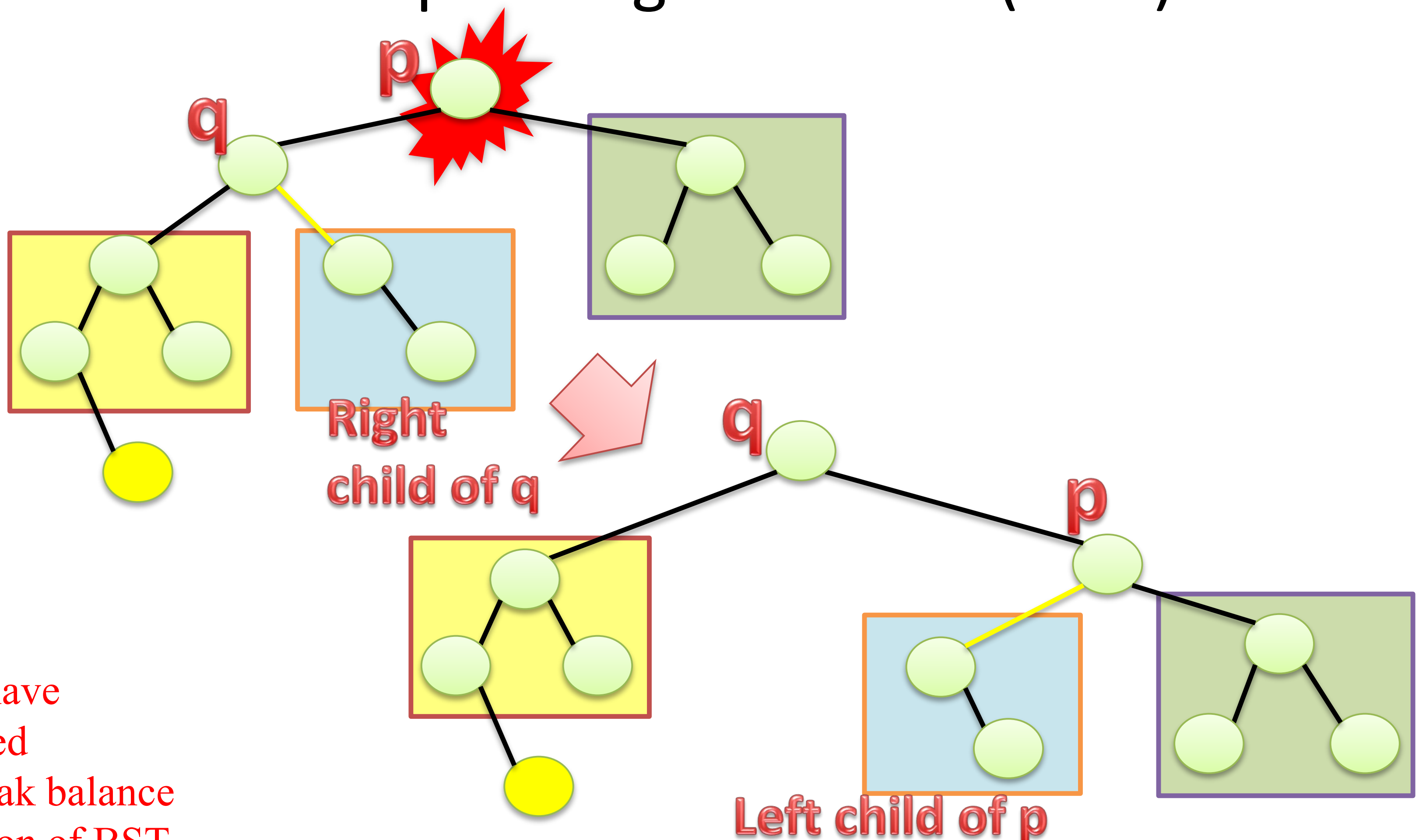
AVL tree: Rebalance by rotations

- If you insert/remove data, the BST can get unbalanced.
- “Rotate” tree vertices to make the difference up to 1:
  - Rotation LL
  - Rotation RR
  - Double rotation LR
  - Double rotation RL

Rebalance of AVL-tree by rotation: Rotation LL

- Lift up left subtree (yellow) if too deep

  we have to transplant right subtree (blue)

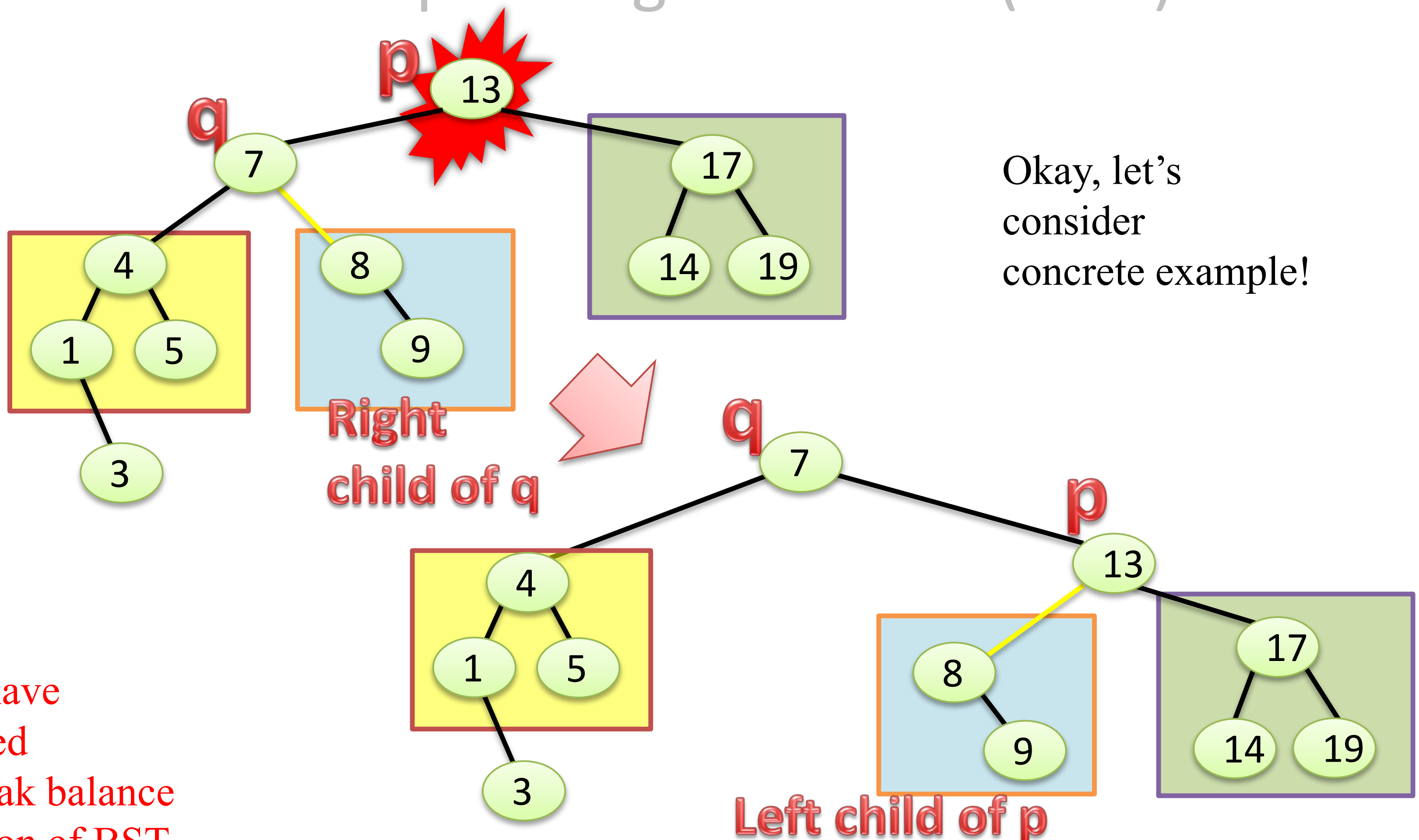


Now we have
- balanced
- not break balance
- condition of BST

Rebalance of AVL-tree by rotation: Rotation LL

- Lift up left subtree (yellow) if too deep

  we have to transplant right subtree (blue)



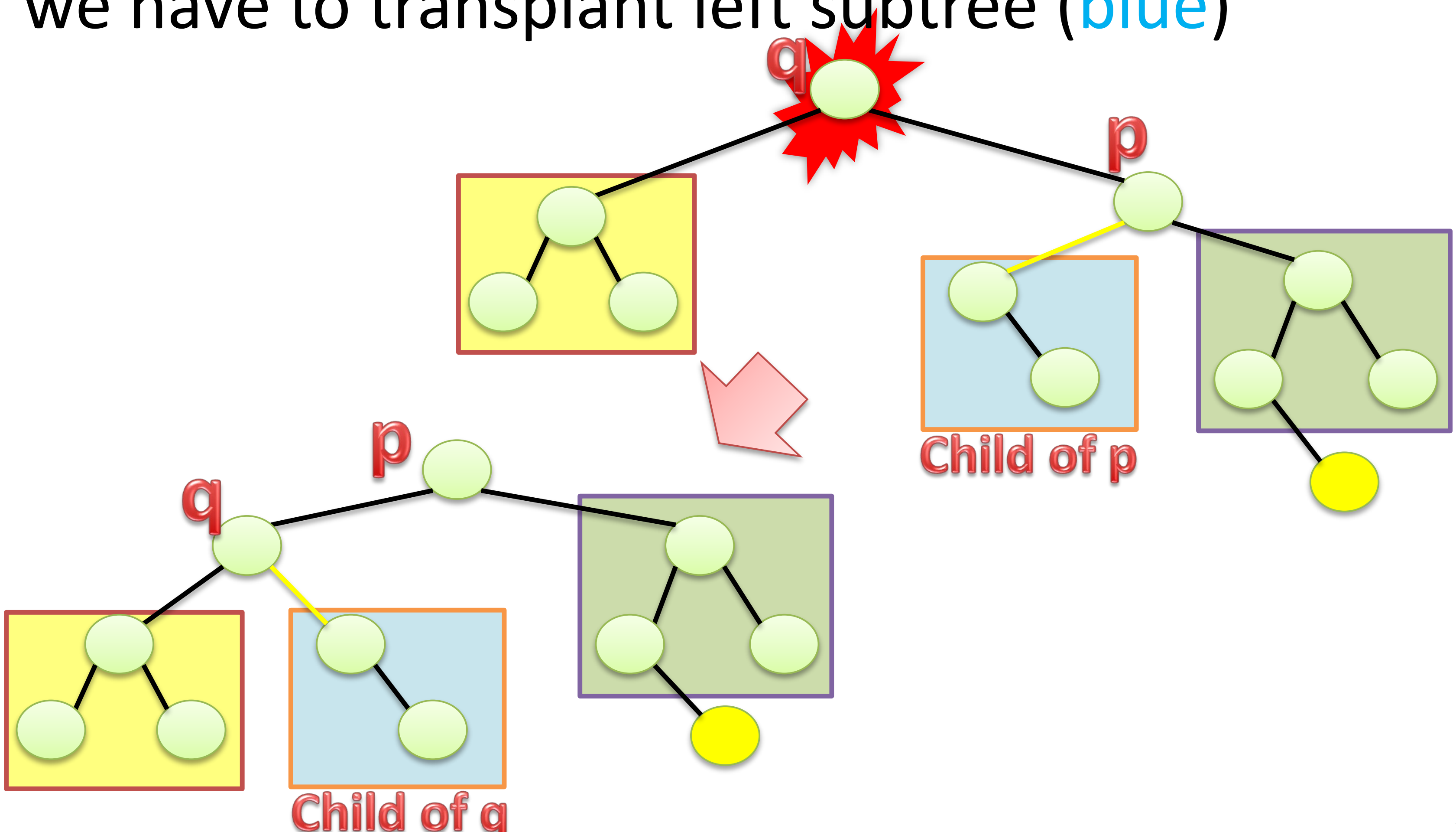
Okay, let's consider concrete example!

Now we have
- balanced
- not break balance
- condition of BST

Rebalance of AVL-tree by rotation: Rotation RR (just mirror image of LL)

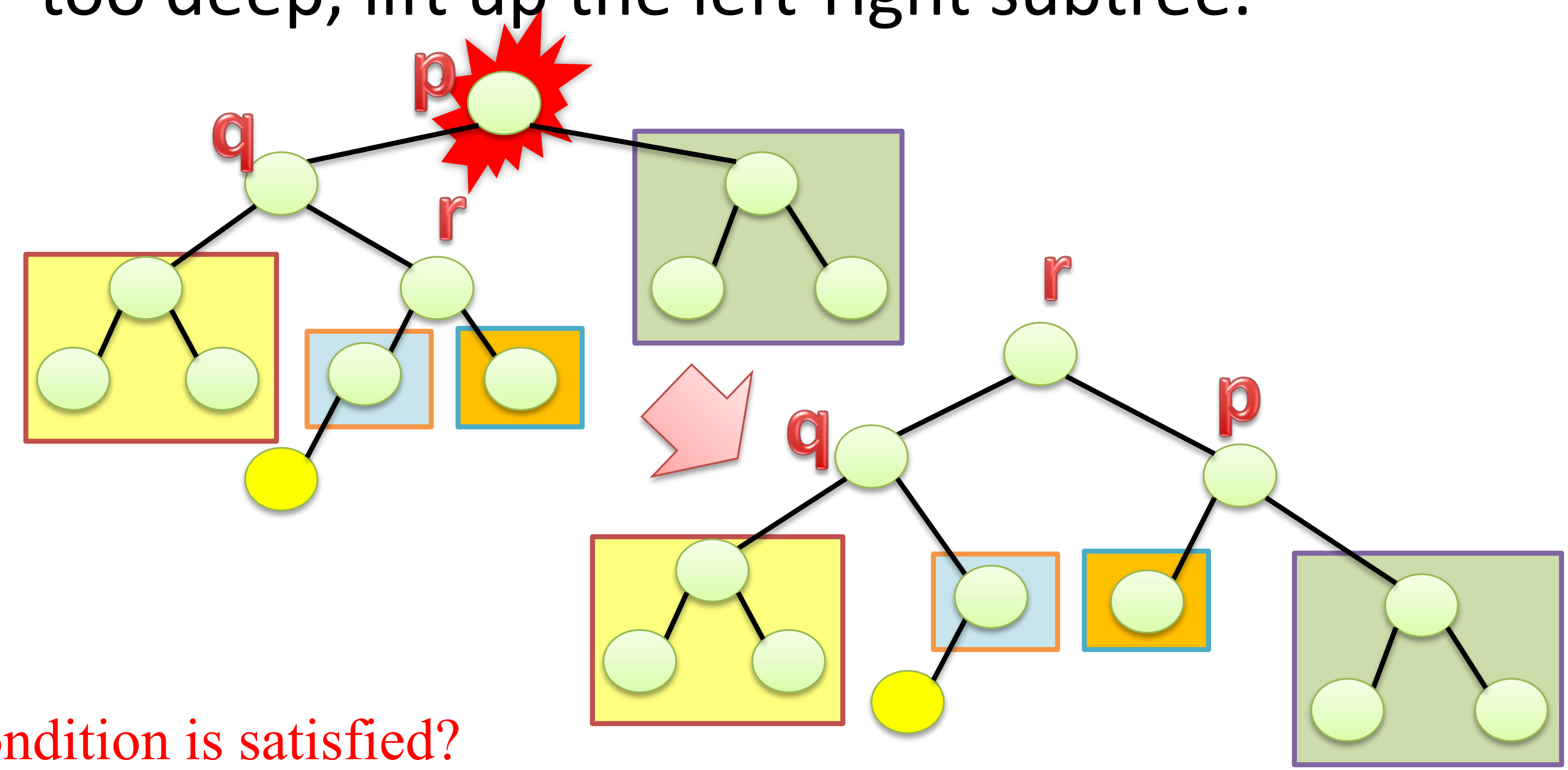
- Lift up right subtree (green) if too deep

  we have to transplant left subtree (blue)



AVL tree: Rebalance by rotation: Double rotation LR

- When right subtree of left subtree becomes too deep, lift up the left-right subtree.

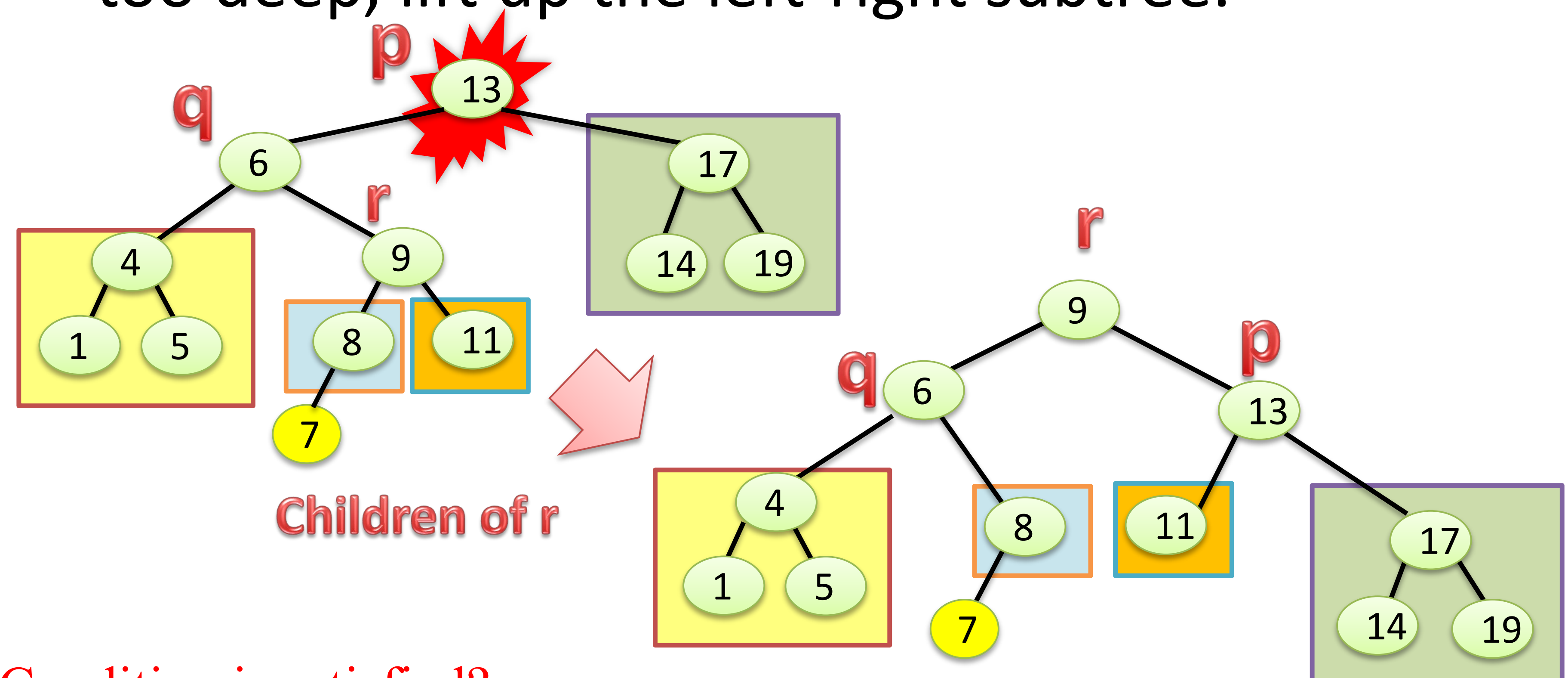


※Condition is satisfied?

※Why rotation LL does not work?

AVL tree: Rebalance by rotation: Double rotation LR

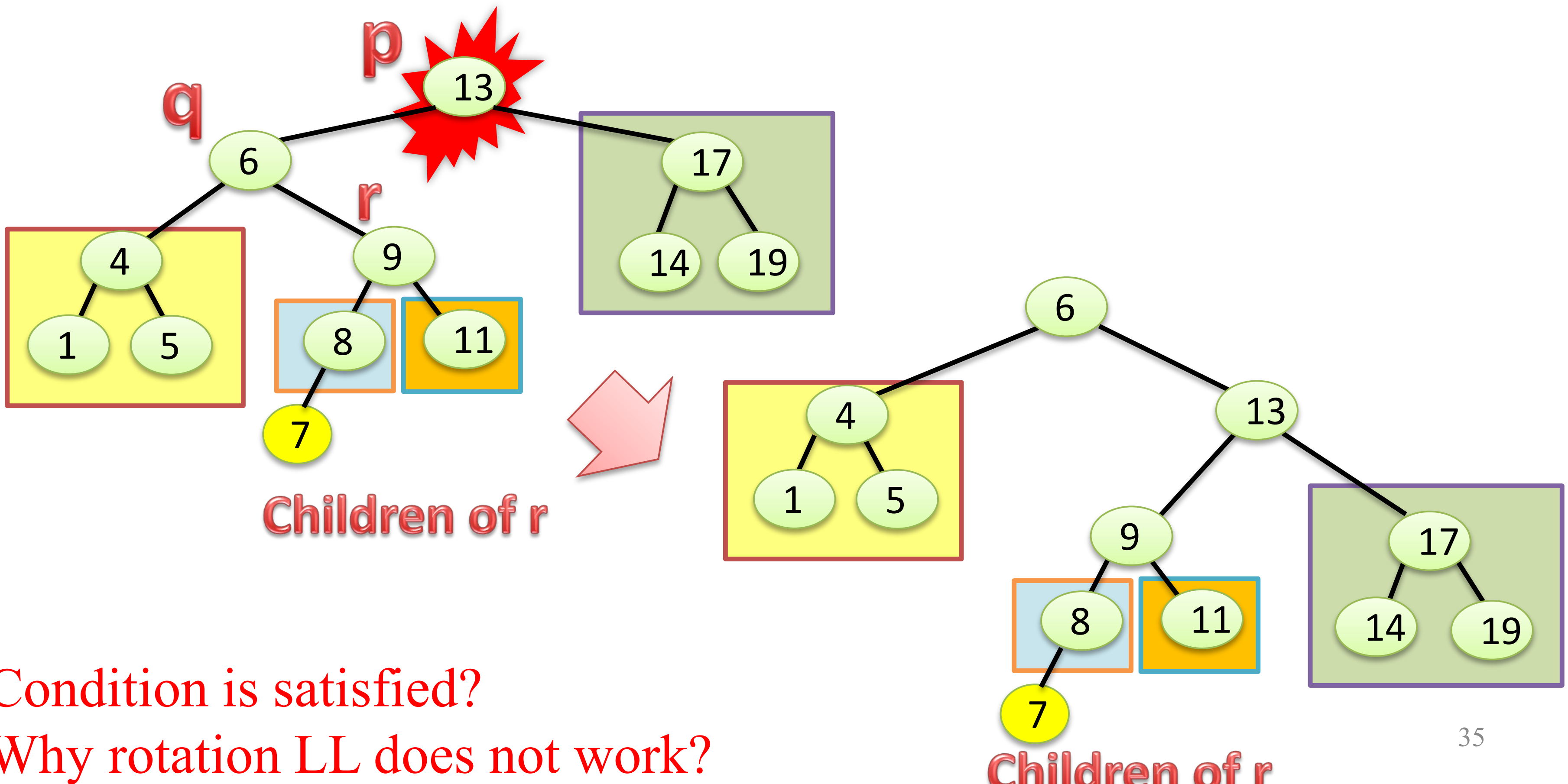
- When right subtree of left subtree becomes too deep, lift up the left-right subtree.



※Condition is satisfied?

※Why rotation LL does not work?

(If you apply rotation LL)

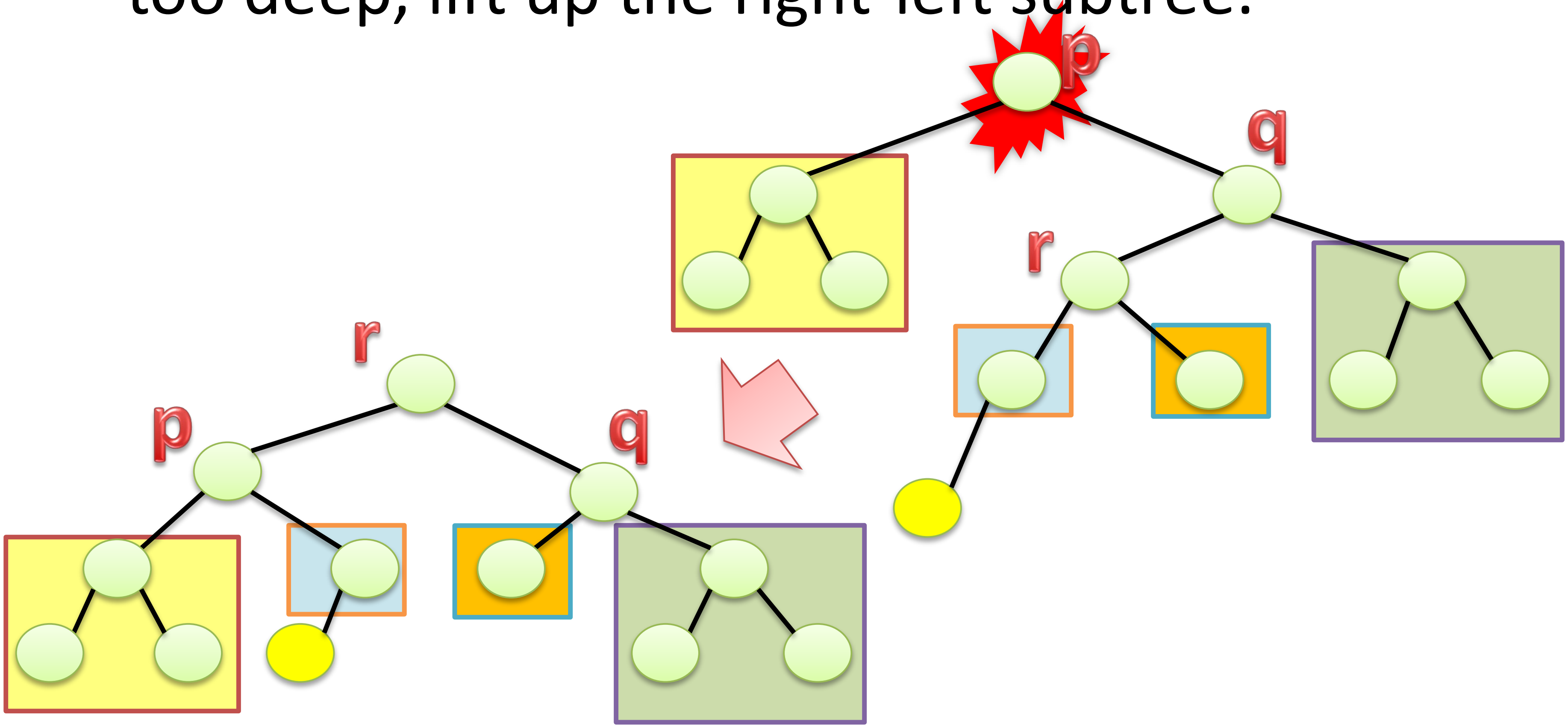


※Condition is satisfied?

※Why rotation LL does not work?

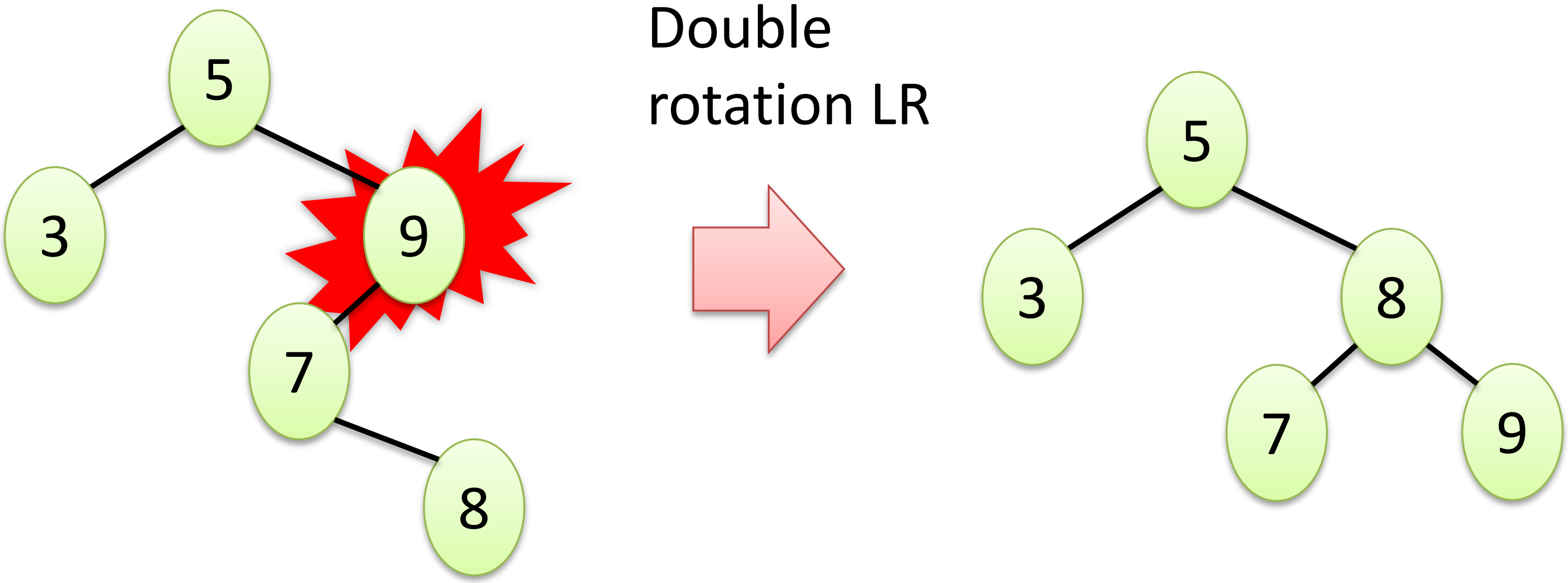
AVL tree: Rebalance by rotation: Double rotation RL (just mirror image of LR)

- When left subtree of right subtree becomes too deep, lift up the right-left subtree.



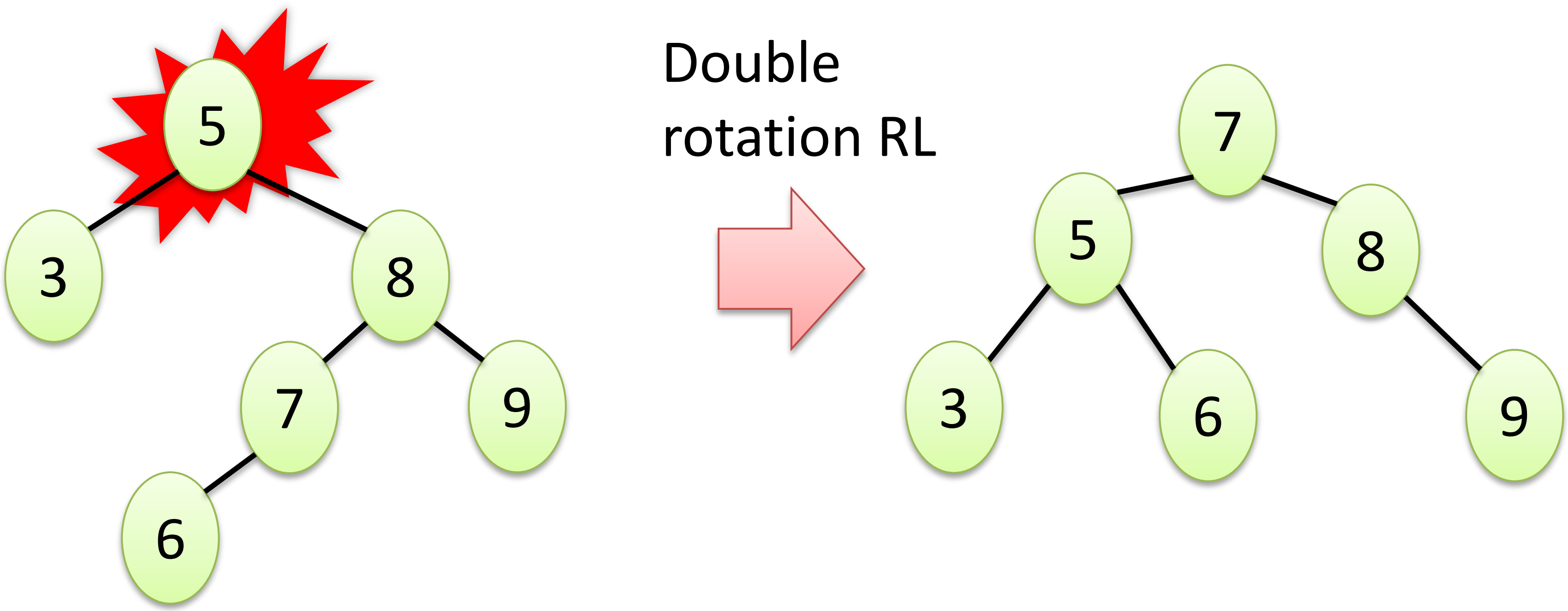
AVL tree: Example

- Insertion of 8



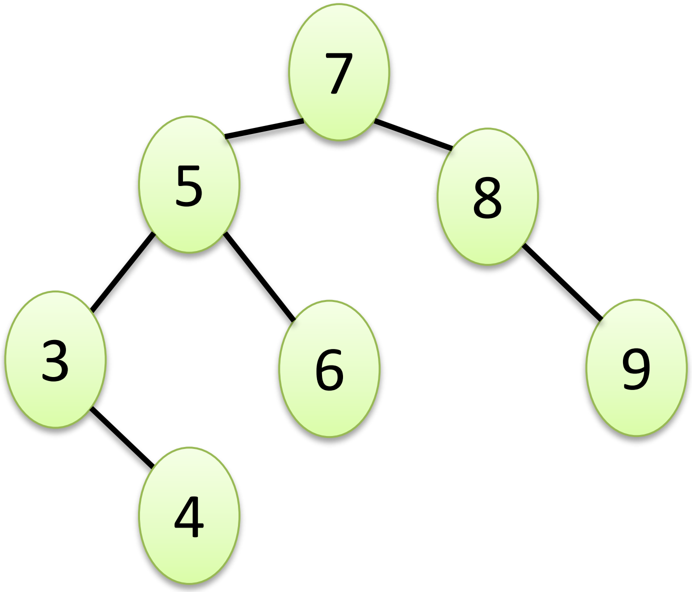
AVL tree: Example

- Insertion of 6



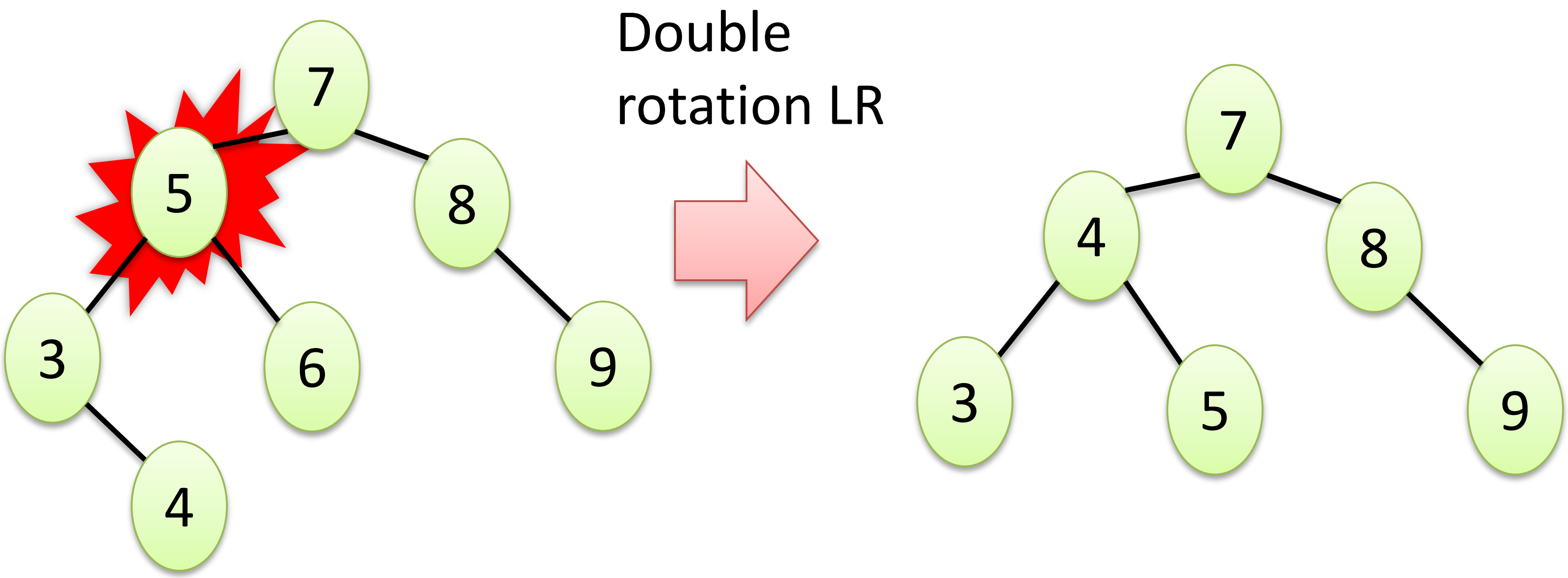
AVL tree: Example

- Insertion of 4 (balance is okay)



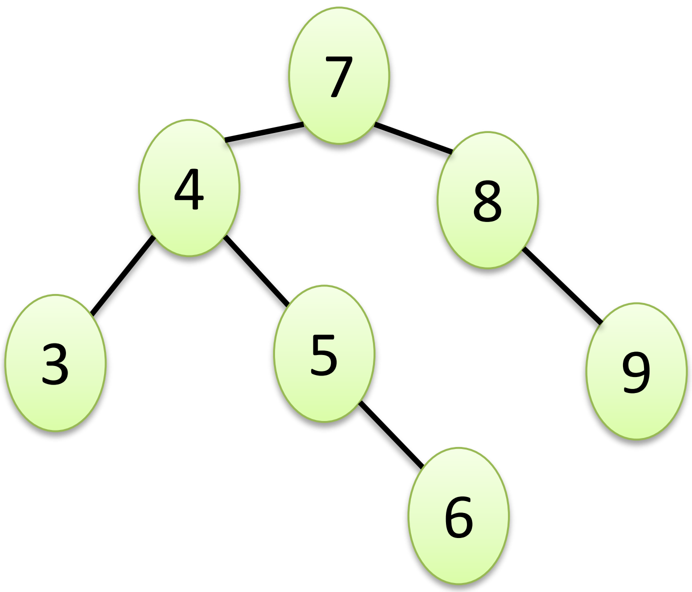
AVL tree: Example

- Deletion of 6



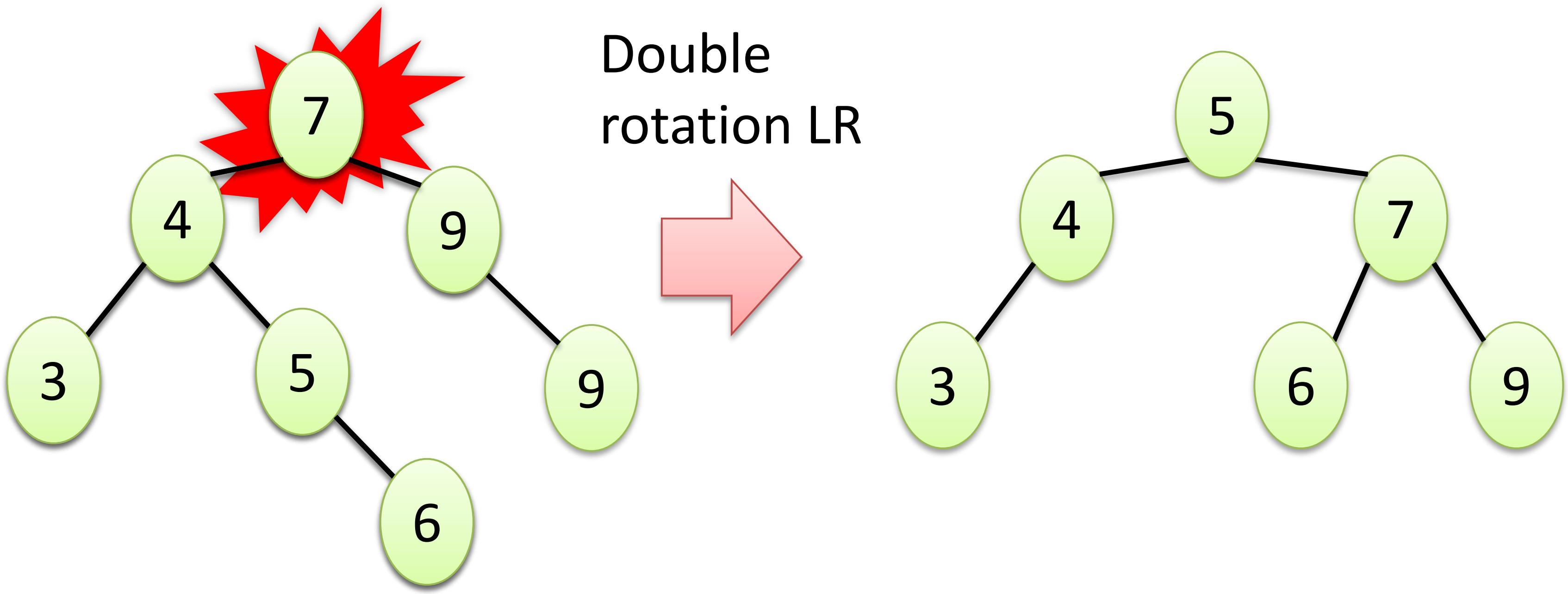
AVL tree: Example

- Insertion of 6 (balance is okay)



AVL tree: Example

- Deletion of 8



Time complexity of balanced binary search tree

- Search: $O(\log n)$ time
- Insertion/Deletion: $O(\log n)$ time
  - $O(\log n)$ rotations
  - Each rotation takes constant time

- In total, on a balanced binary search tree, every operation can be done in $O(\log n)$ time.

  (n is the number of data in the tree)

Announcement

- 1st report: deadline is November 11, 10:50am.
- Mid term examination (30pts):
  - November 11, 13:30-15:10
  - Range: up to November 6
    - Bring copy of slides, and hand written notes

Reports so far

- On 8:25am, November 11:

s1810235, s1810446, s1910192, s1910402, s1910416, s1910438, s1810403, s1910069, s1910231, s1910412, s1910421, s1910440, s1810433, s1910158, s1910401, s1910414, s1910436,

- Quick comments
  - We have sent confirmation email
  - Send your report to both of Giovanni and Ryuhei
  - File name should be, e.g., s1234567.pdf of one PDF file.