# Trees: part 1

Petr Kurapov

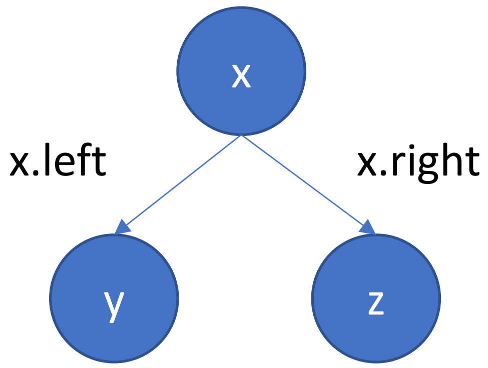
Fall 2024

# Binary trees

- BST: maps & sets
- Binary tries (digital/prefix trees): high-bandwidth routing, dictionary for autocompletion
- Heaps: priority queues
- Huffman coding tree: compression (i.e. jpeg)
- B+/w trees: database indexing

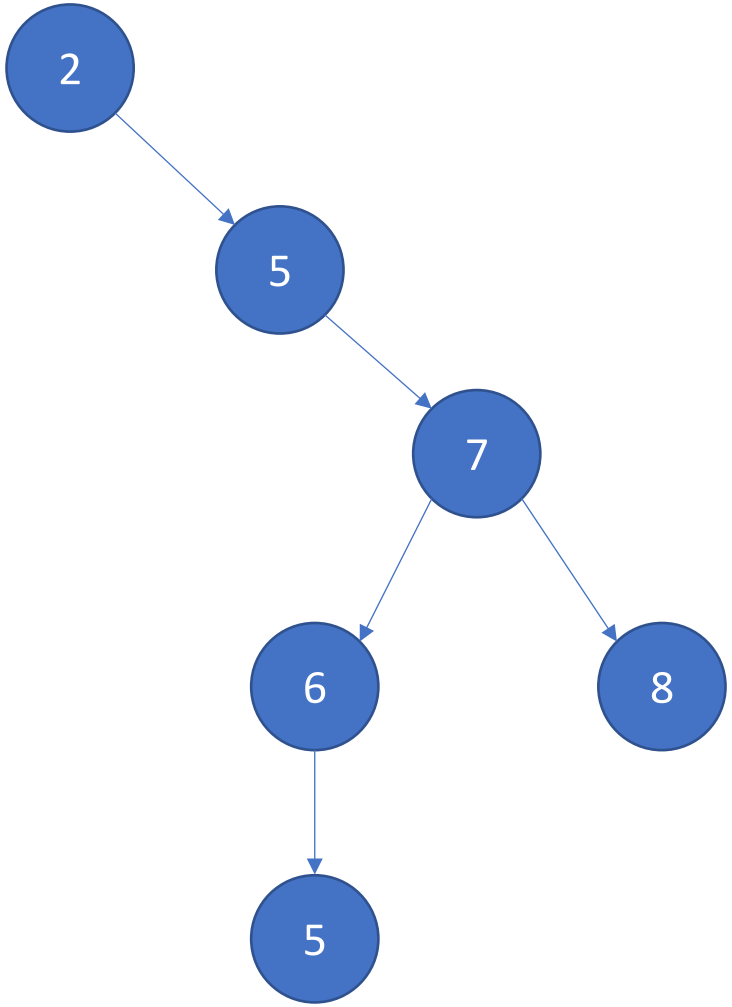
# Binary Search Tree (BST)

- Binary tree
  - $\{y.key \leq x.key, z.key \geq x.key\}$
  - Operator < defined on keys
- Most operations done in $O(T.hight) \sim O(\log n)$ mean for a random tree
- Dictionary/priority queue



# Binary Search Tree (BST)

- Binary tree
  - $\{y.key \leq x.key, z.key \geq x.key\}$
  - Operator < defined on keys
- Most operations done in $O(T.hight) \sim O(\log n)$ mean for a random tree
- Dictionary/priority queue



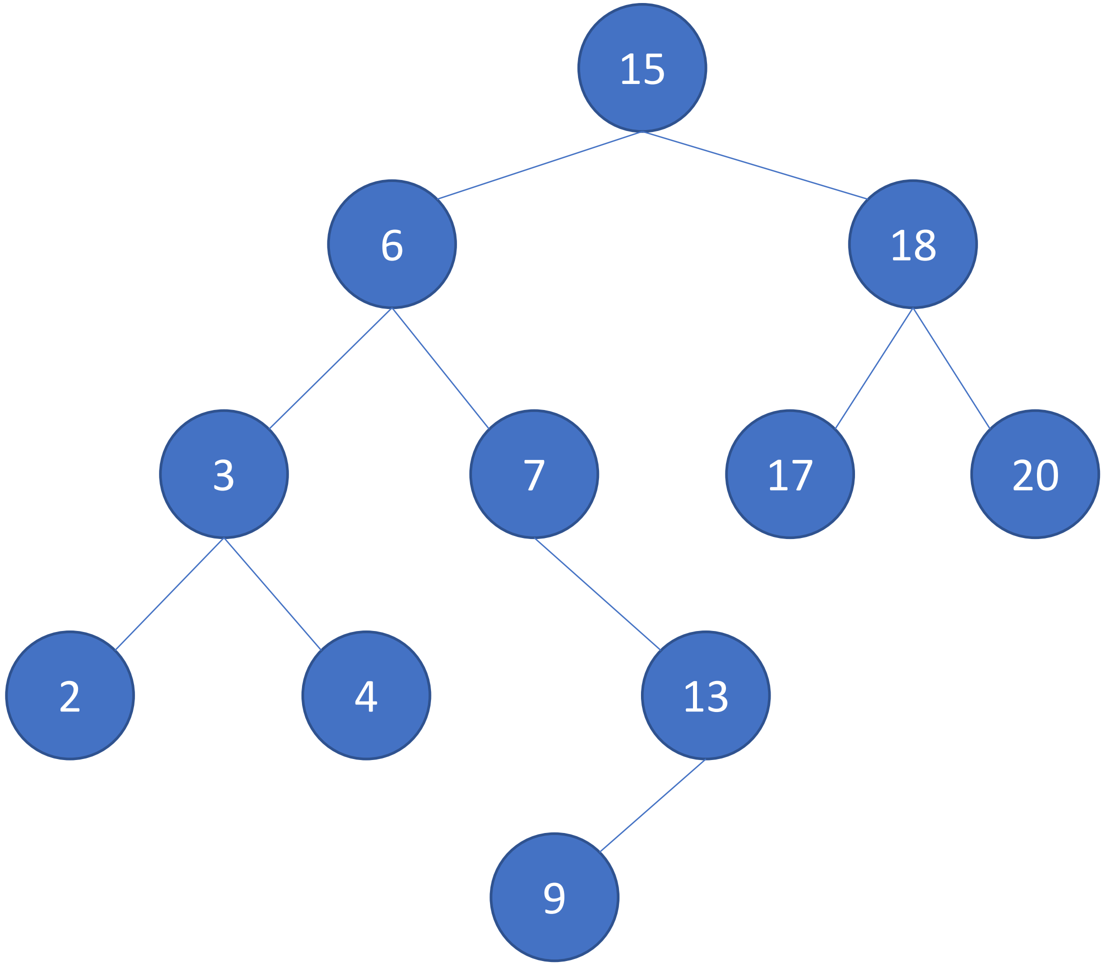
# Binary Search Tree (BST)

- Binary tree
  - $\{y.key \leq x.key, z.key \geq x.key\}$
  - Operator < defined on keys
- Most operations done in $O(T.hight) \sim O(\log n)$ mean for a random tree
- Dictionary/priority queue

Base operations:

- Search
- Insertion
- Min/max
- Deletion
- Traverse sorted

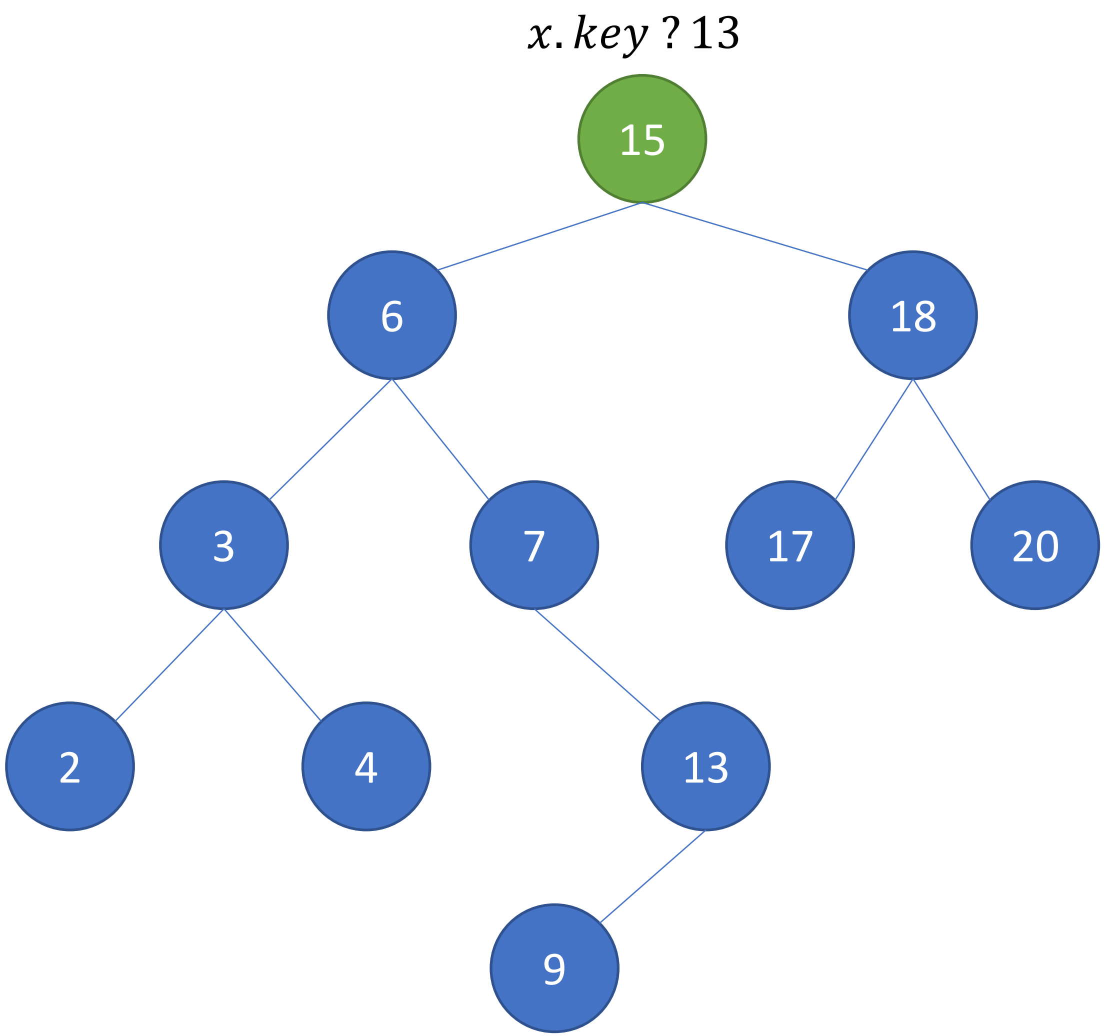
# Binary Search Tree (BST)




Base operations:

- **Search**
- Insertion
- Min/max
- Deletion
- Traverse sorted

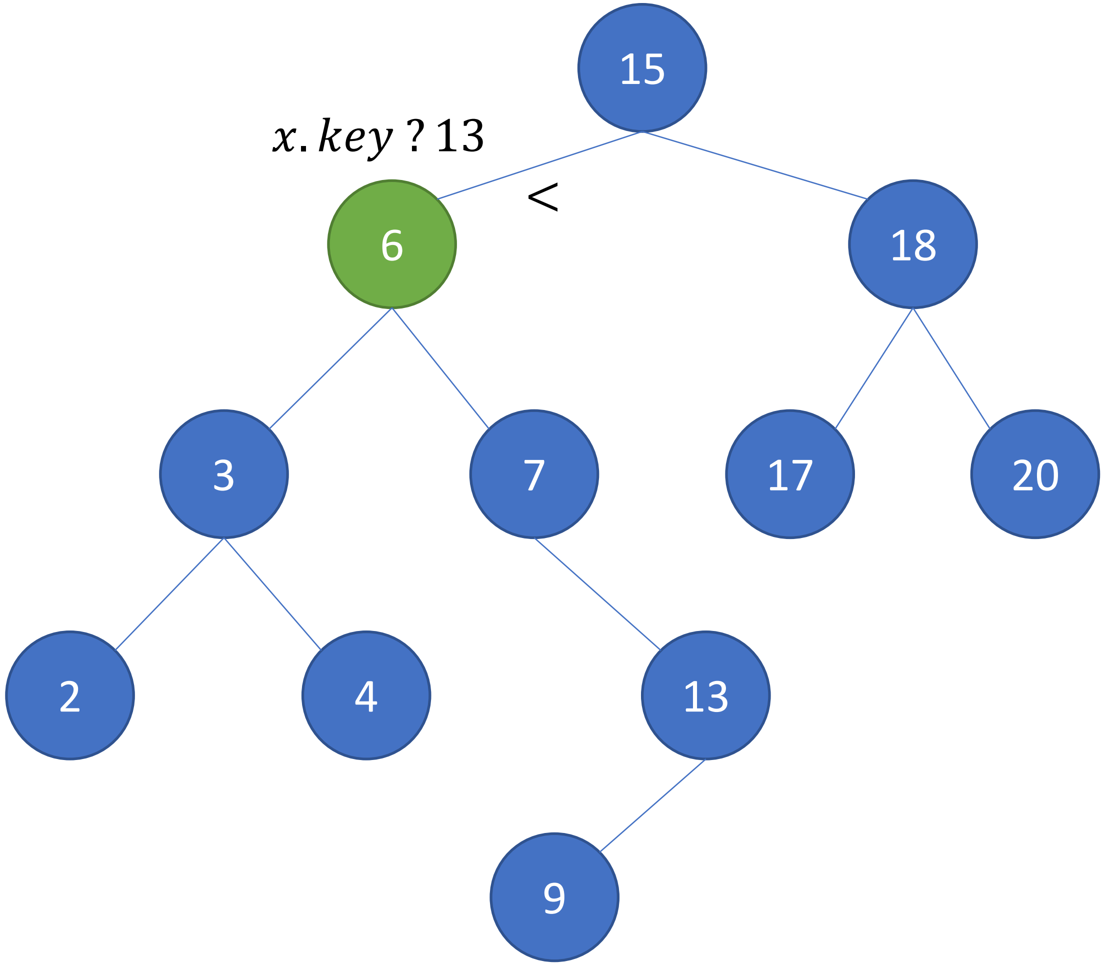
# Binary Search Tree (BST)




Base operations:

- **Search**
- Insertion
- Min/max
- Deletion
- Traverse sorted

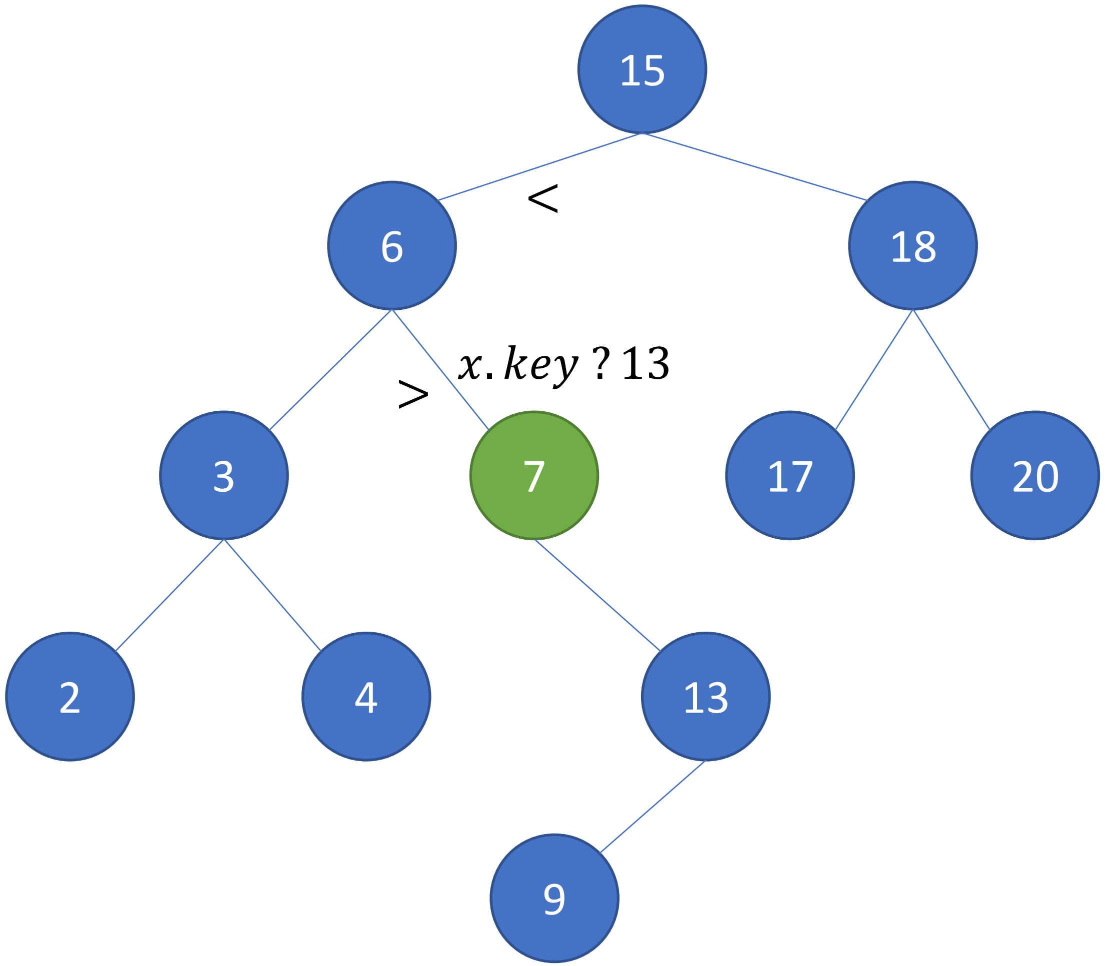
# Binary Search Tree (BST)




Base operations:

- **Search**
- Insertion
- Min/max
- Deletion
- Traverse sorted

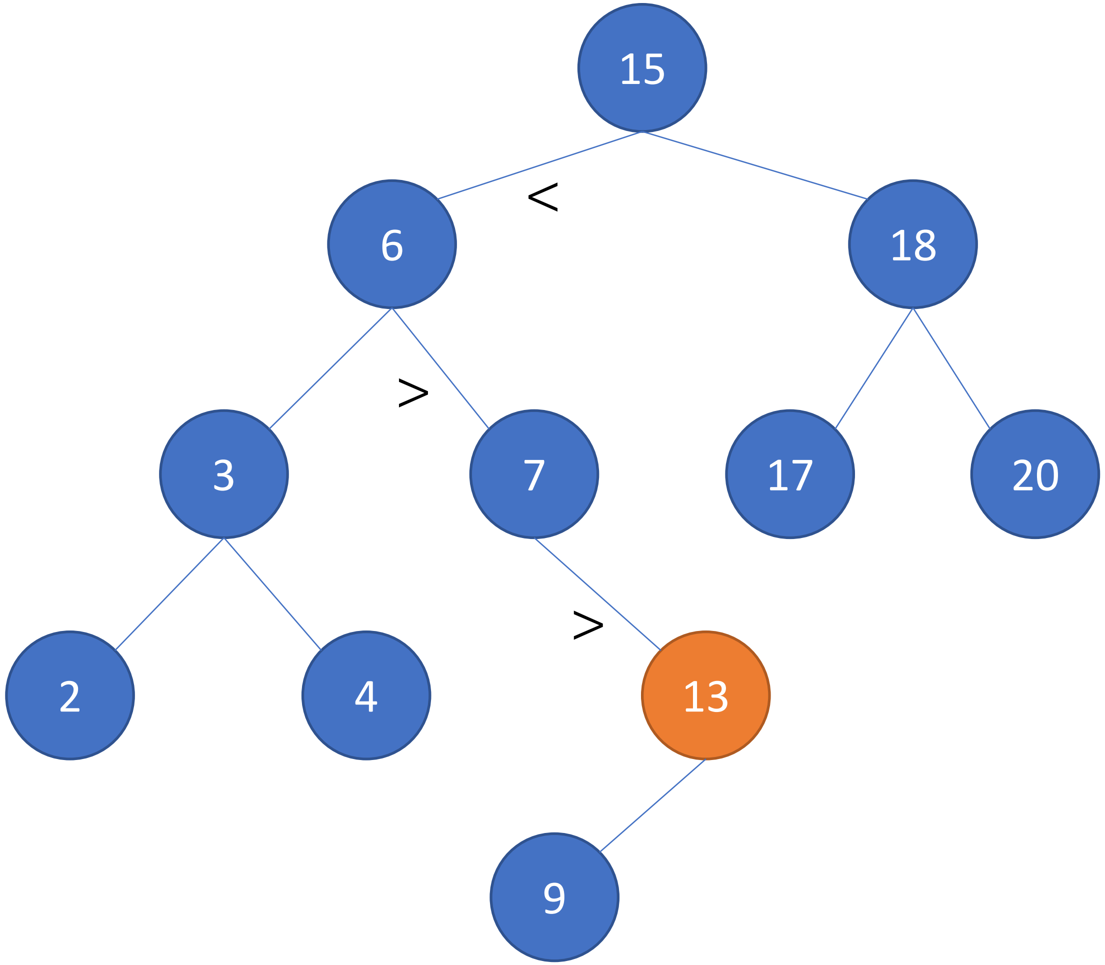
# Binary Search Tree (BST)




Base operations:

- **Search**
- Insertion
- Min/max
- Deletion
- Traverse sorted

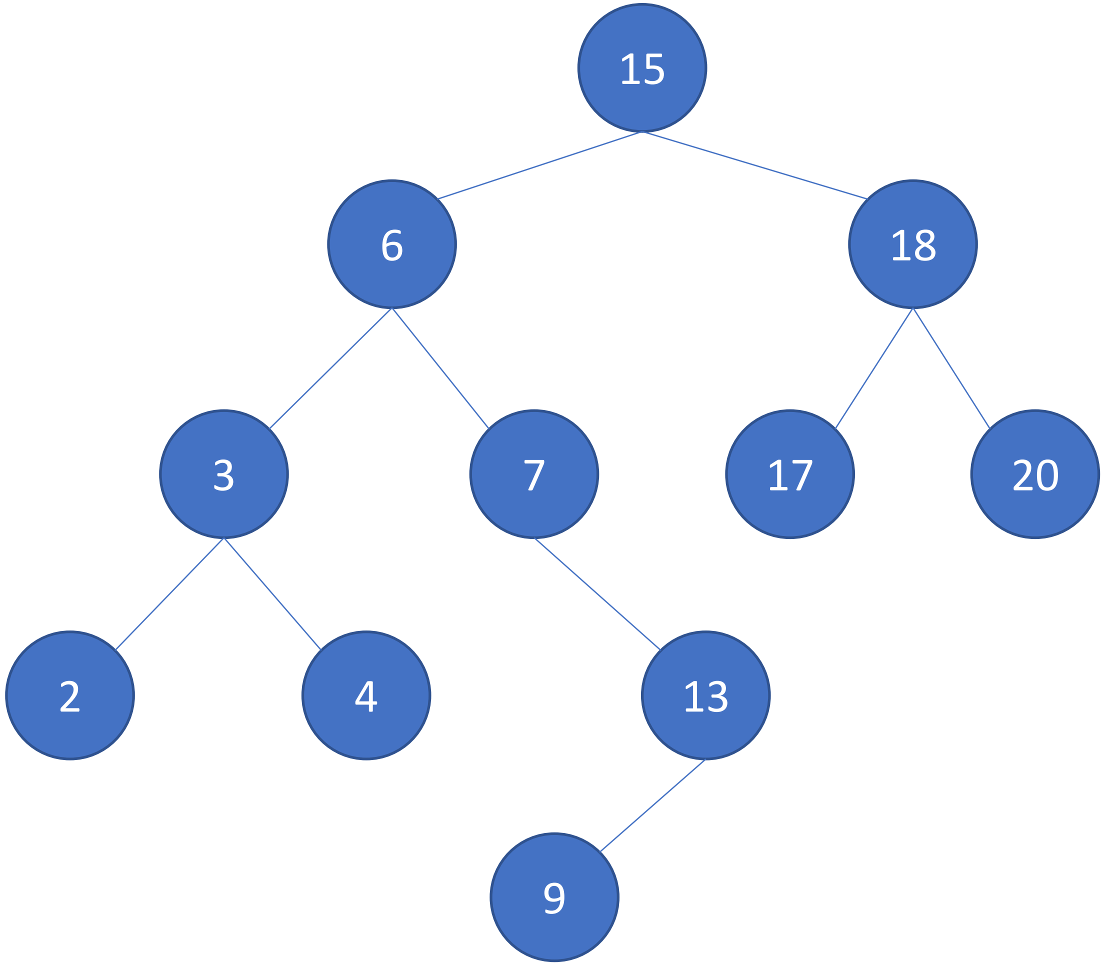
# Binary Search Tree (BST)




Base operations:

- **Search**
- Insertion
- Min/max
- Deletion
- Traverse sorted

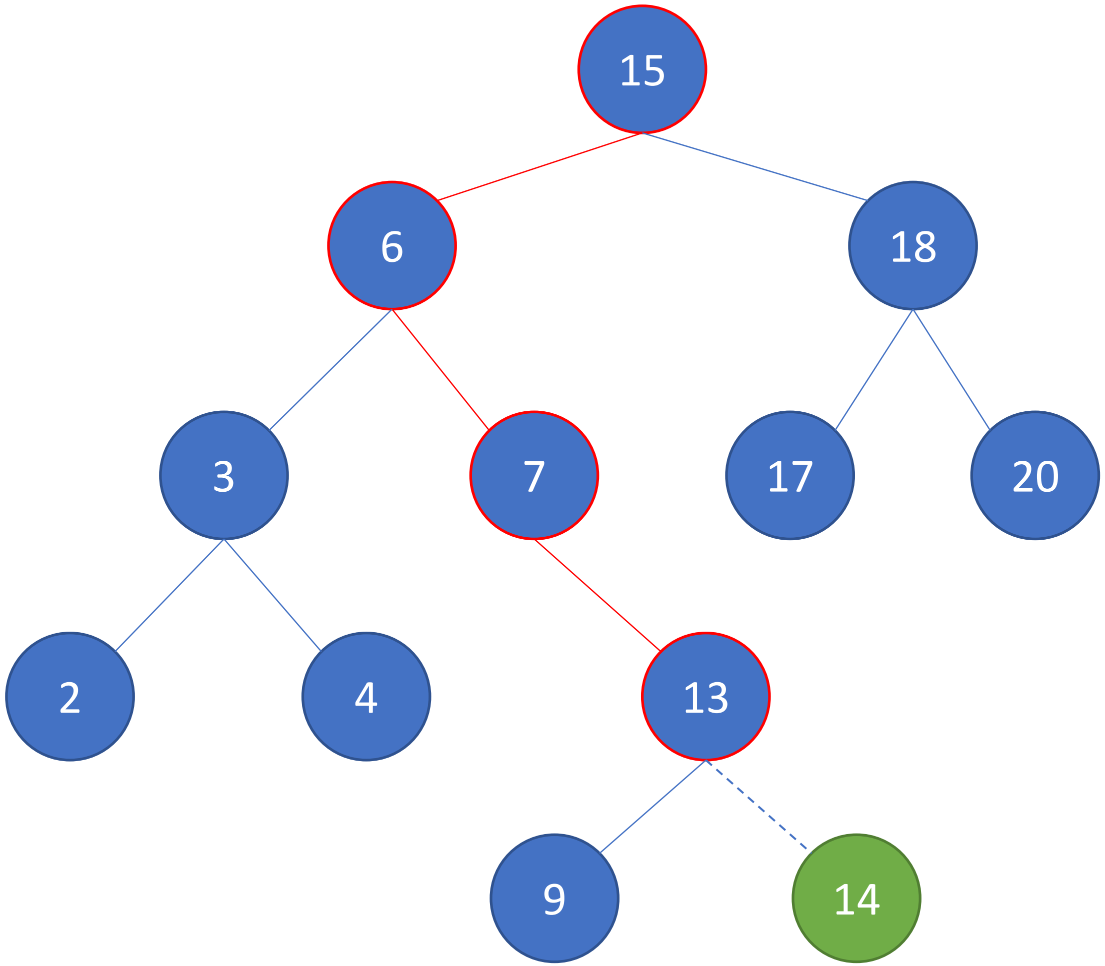
# Binary Search Tree (BST)




Base operations:

- Search
- **Insertion**
- Min/max
- Deletion
- Traverse sorted

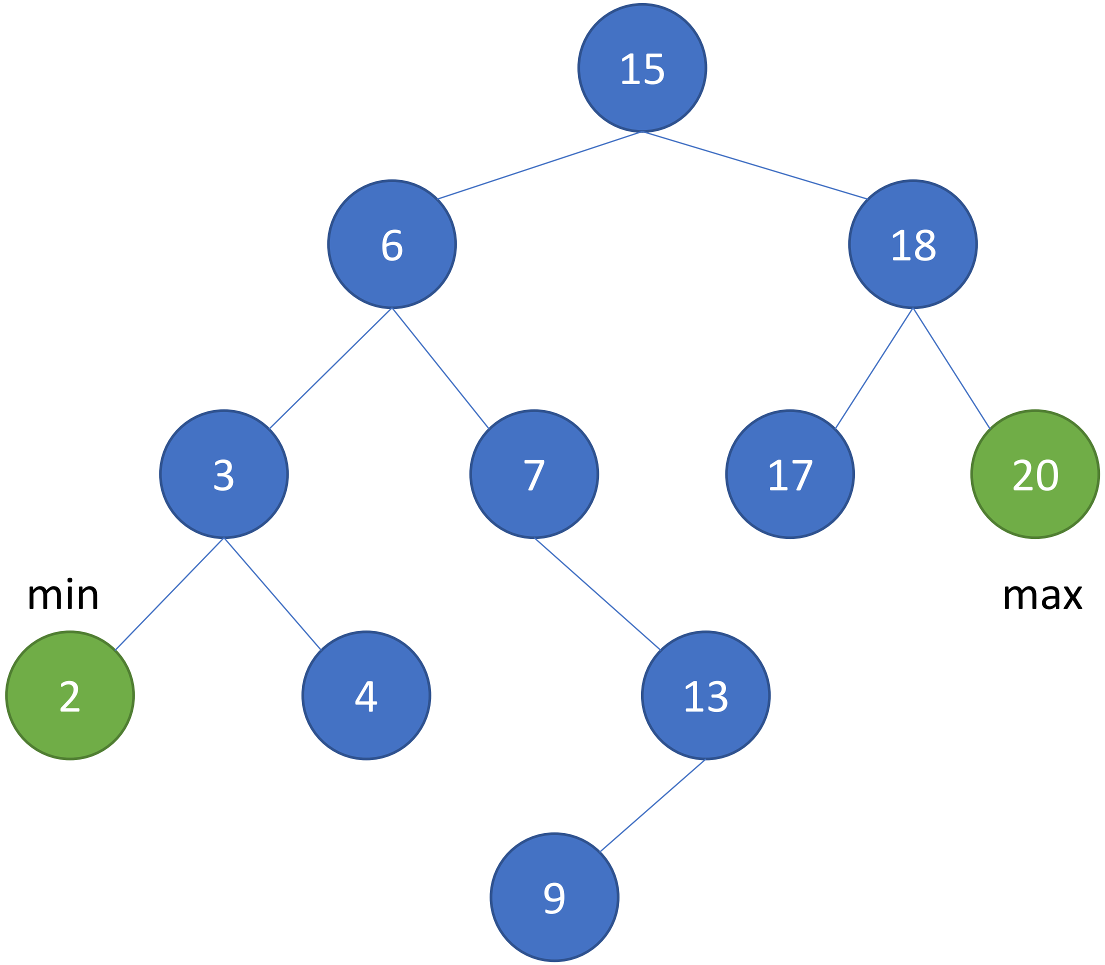
# Binary Search Tree (BST)




Base operations:

- Search
- **Insertion**
- Min/max
- Deletion
- Traverse sorted

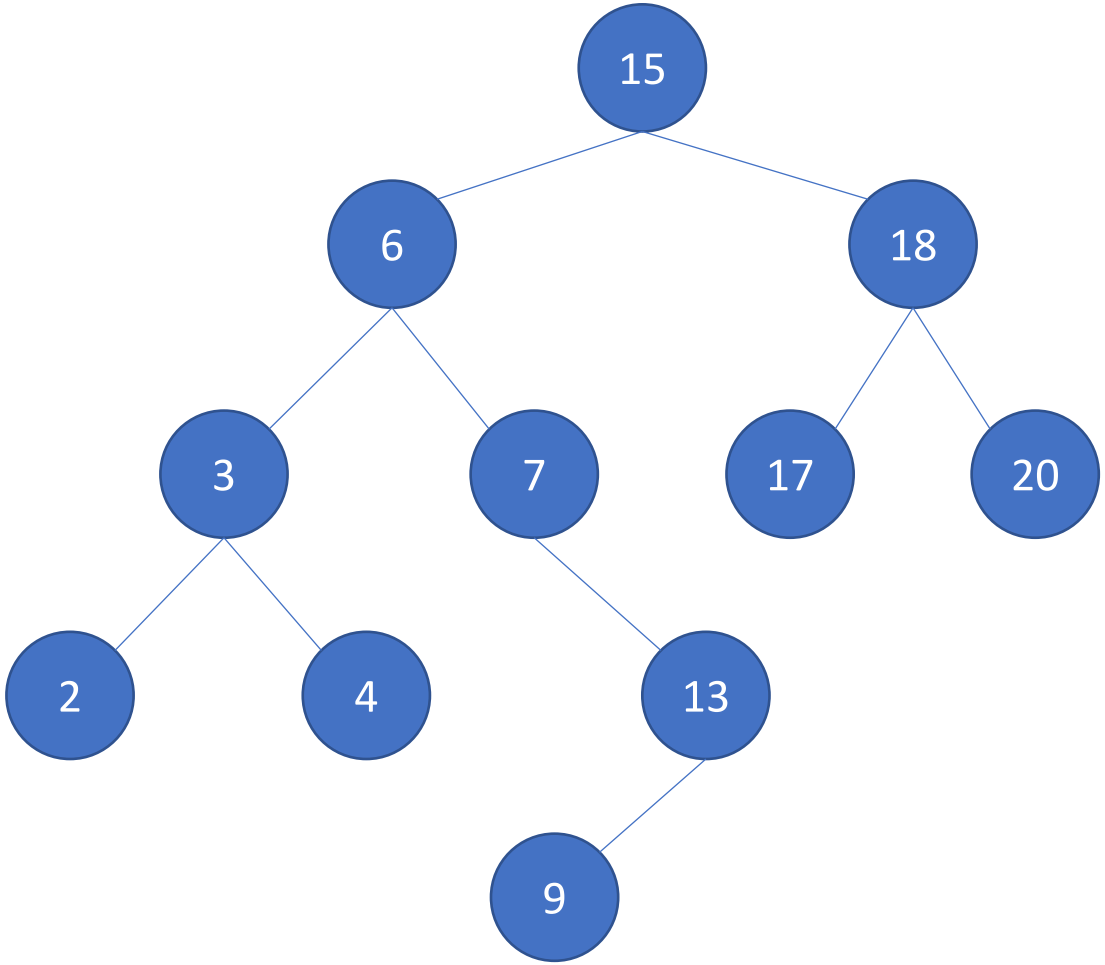
# Binary Search Tree (BST)




Base operations:

- Search
- Insertion
- **Min/max**
- Deletion
- Traverse sorted

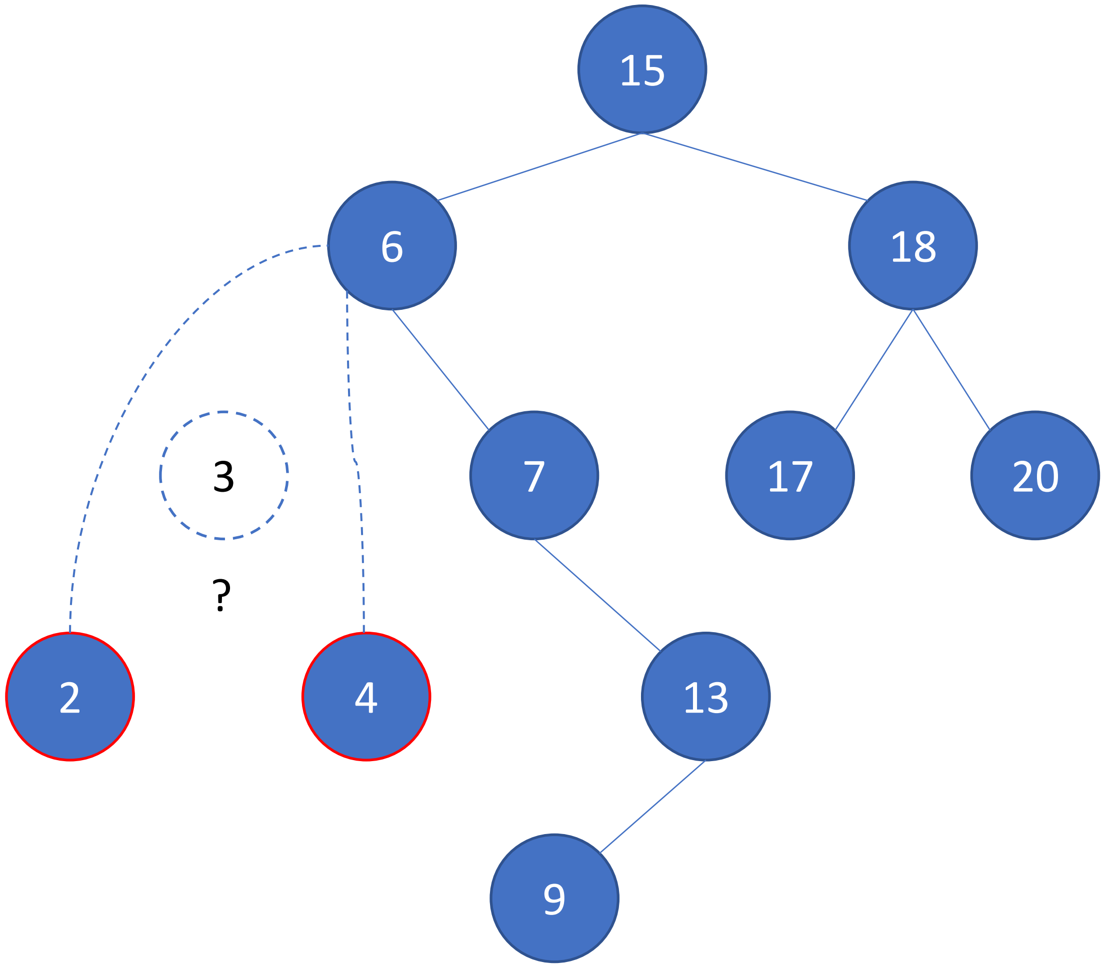
# Binary Search Tree (BST)




Base operations:

- Search
- Insertion
- Min/max
- **Deletion**
- Traverse sorted

# Binary Search Tree (BST)




Base operations:

- Search
- Insertion
- Min/max
- **Deletion**
- Traverse sorted

# Binary Search Tree (BST)

Three possibilities:

1. No child nodes
   - Just remove it - trivial
2. Single child node
   - Remove and relink to parent - simple
3. Both children present
   - Replace with next
   - Same left subtree
   - Reminder of right subtree becomes new right subtree

Base operations:

- Search
- Insertion
- Min/max
- **Deletion**
- Traverse sorted

# Binary Search Tree (BST)

Three possibilities:

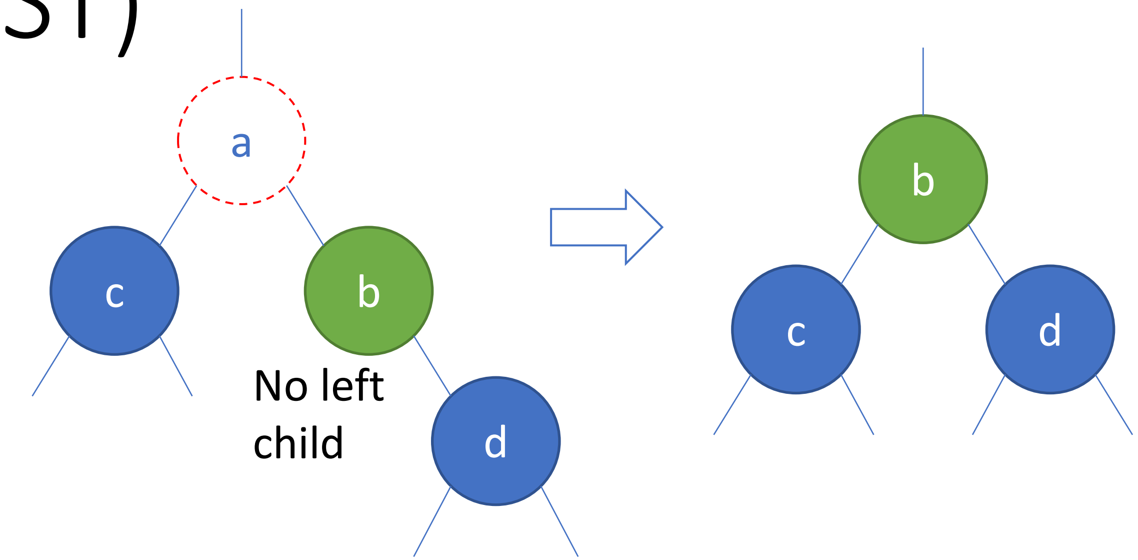
1. No child nodes
    - Just remove it - trivial
2. Single child node
    - Remove and relink to parent - simple
3. **Both children present**
    - Replace with *next*
    - Same left subtree
    - Reminder of right subtree becomes new right subtree

How do we get next?

# Binary Search Tree (BST)

Three possibilities:

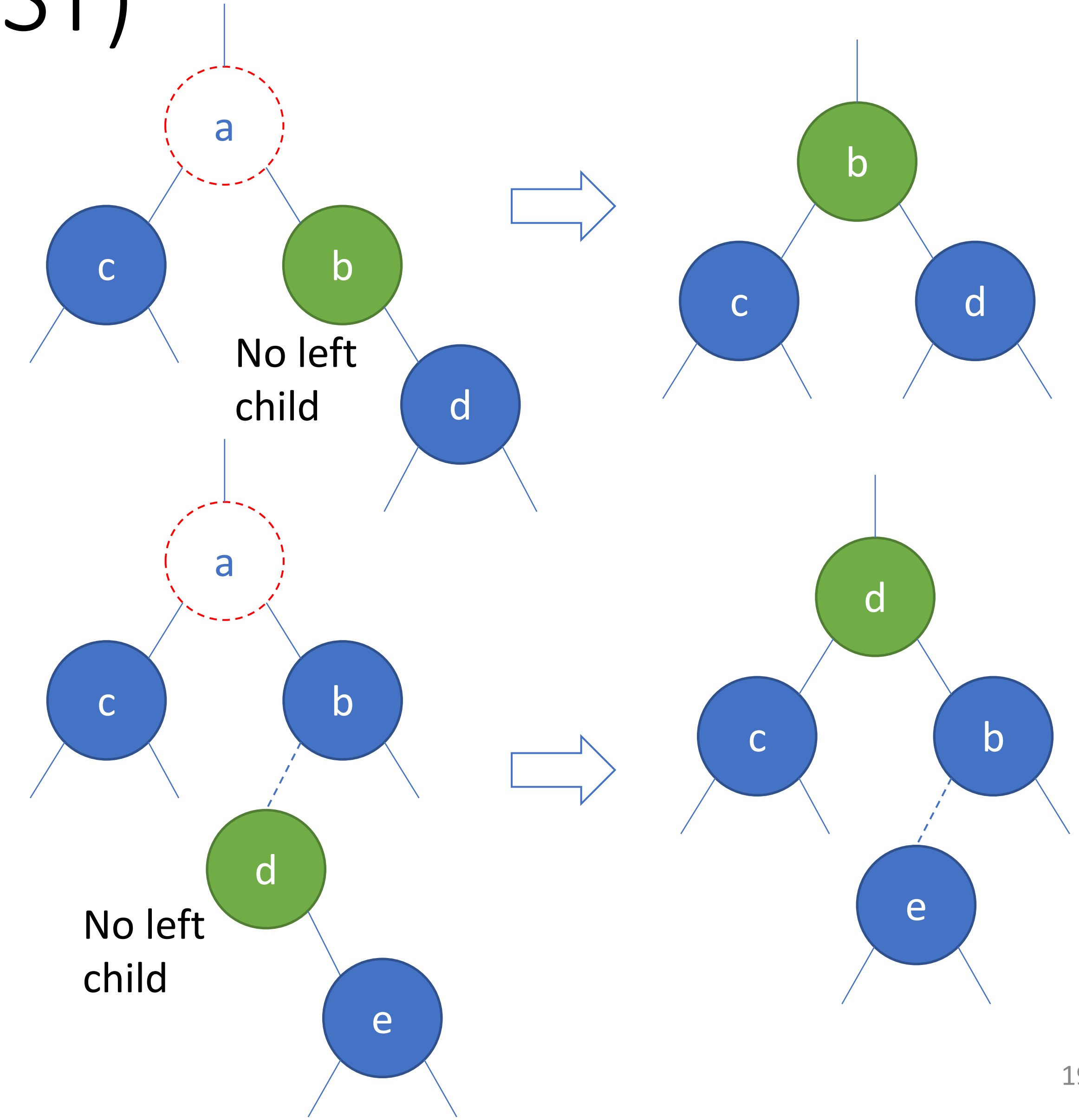
1. No child nodes
   - Just remove it - trivial
2. Single child node
   - Remove and relink to parent - simple
3. **Both children present**
   - Replace with *next*
   - Same left subtree
   - Reminder of right subtree becomes new right subtree



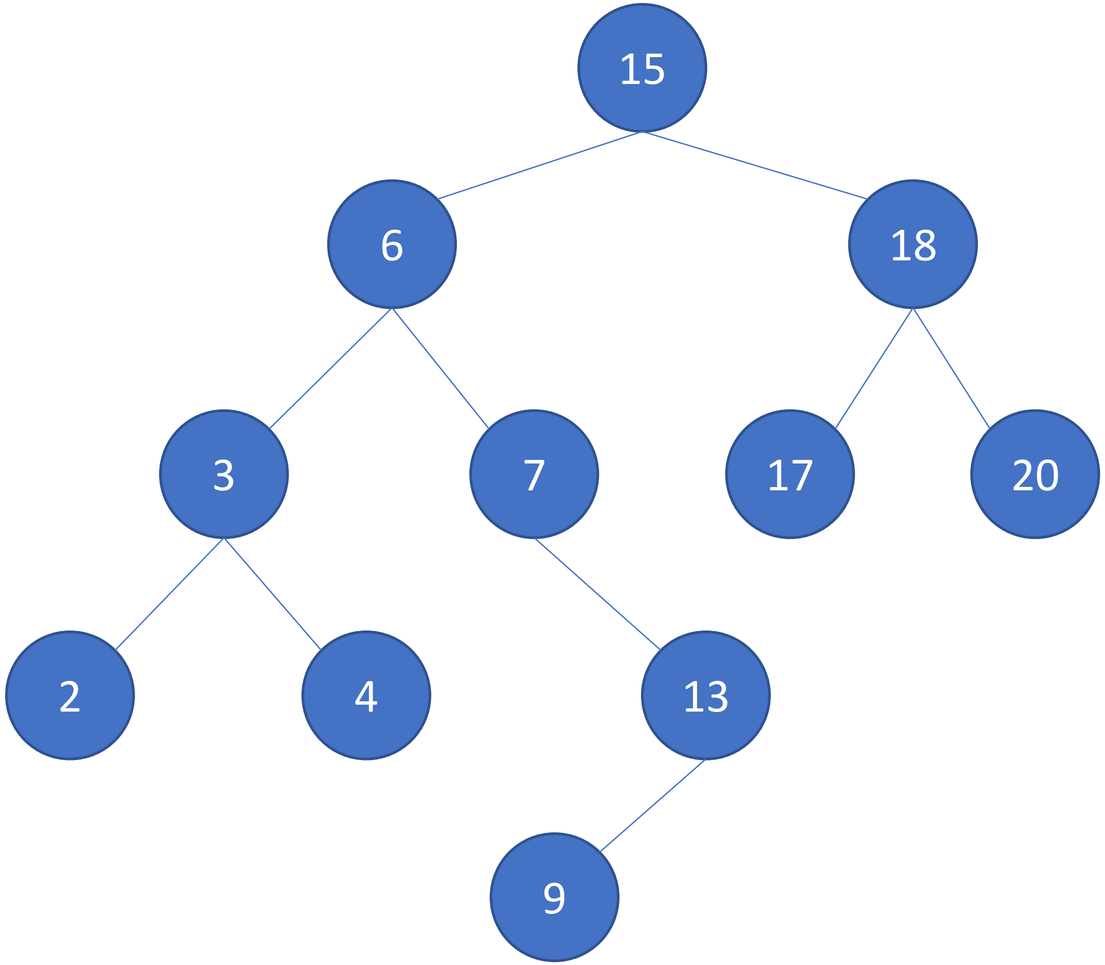
# Binary Search Tree (BST)

Three possibilities:

1. No child nodes
   - Just remove it - trivial
2. Single child node
   - Remove and relink to parent - simple
3. **Both children present**
   - Replace with *next*
   - Same left subtree
   - Reminder of right subtree becomes new right subtree



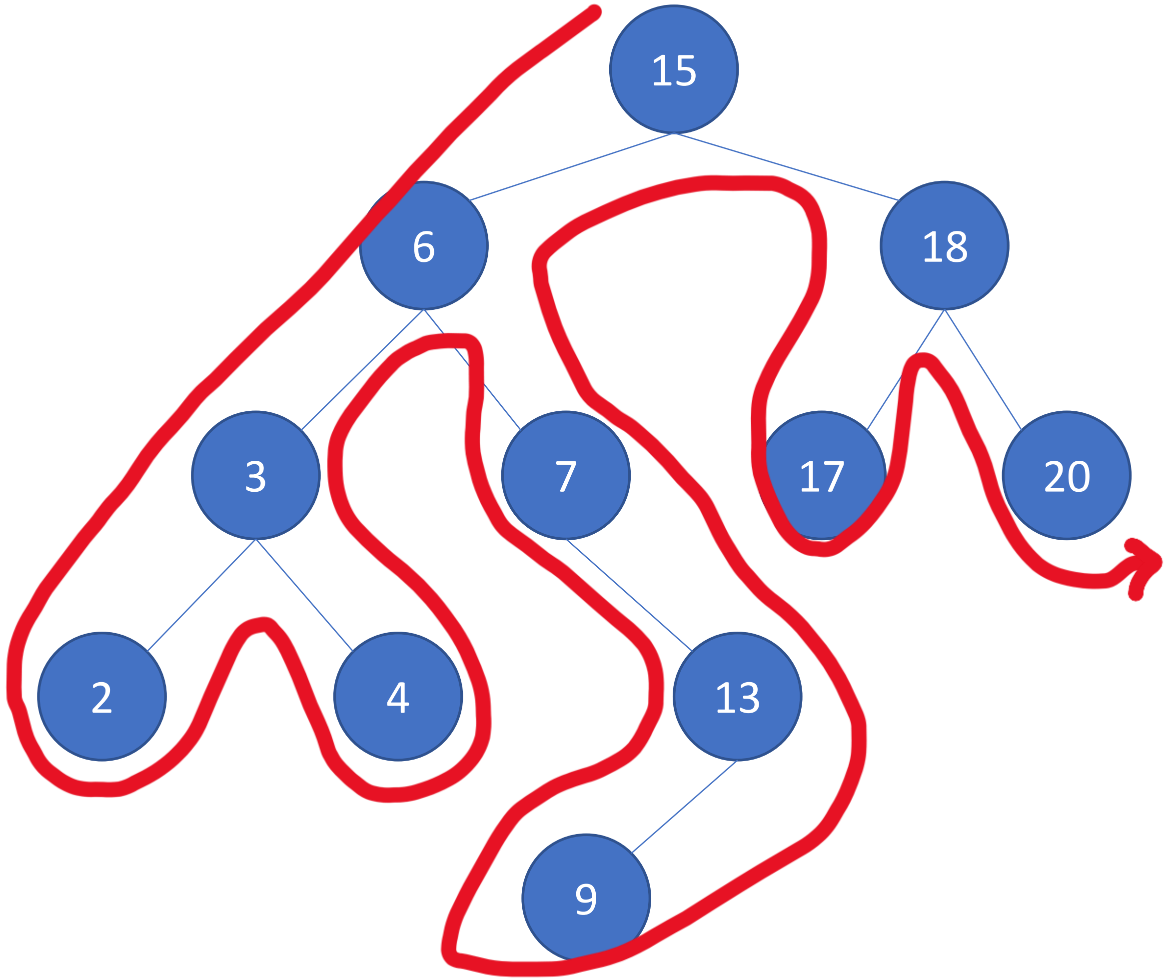
# Binary Search Tree (BST)




Base operations:

- Search
- Insertion
- Min/max
- Deletion
- **Traverse sorted**

# Binary Search Tree (BST)




Base operations:

- Search
- Insertion
- Min/max
- Deletion
- **Traverse sorted**

# Binary Search Tree (BST): summary

- Binary tree
  - $\{y.key \leq x.key, z.key \geq x.key\}$
  - Operator < defined on keys
- Most operations done in $O(T.hight) \sim O(\log n)$ mean for a *random tree* – **worst case is still $\boldsymbol{O(n)}$**
- Dictionary/priority queue

Base operations:

- Search
- Insertion
- Min/max
- Deletion
- Traverse sorted

**Idea: guarantee "dense" tree structure**

# Self-balancing trees

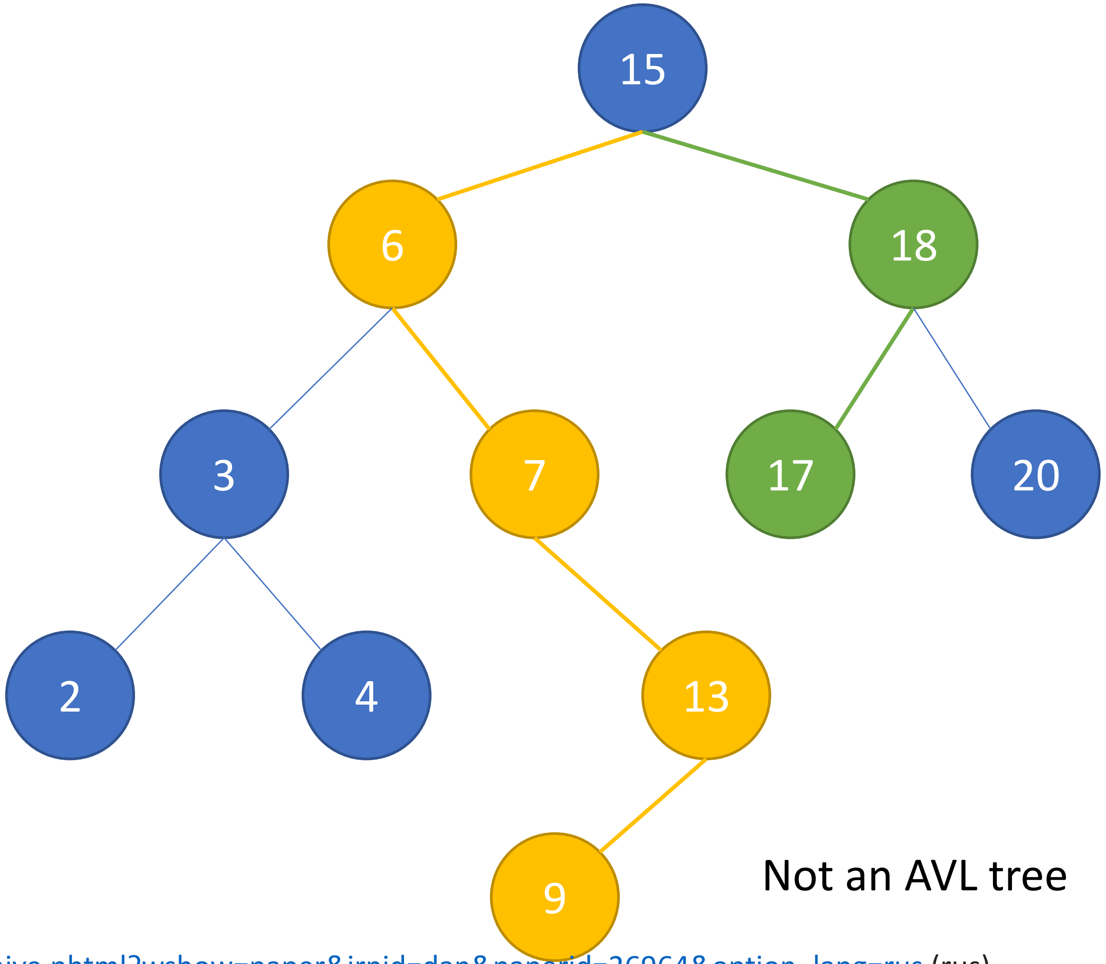
- Ensure tree height to be logarithmic – any $O(T.height)$ operation becomes $O(\log n)$

- AVL, WAVL tree
- Red–black tree
- Splay tree
- B-tree
- T-tree
- …

# AVL tree

- **A**delson-**V**elsky and **L**andis (1962)
- Add balance factor to a node – difference in height between right and left subtrees
- A tree is AVL whenever every node's balance factor $\in \{1, 0, -1\}$
- Changing tree structure (introducing new or removing elements) may disturb AVL invariant, so a *rebalancing* is required

$balance\ factor = 4 - 2 = -2$



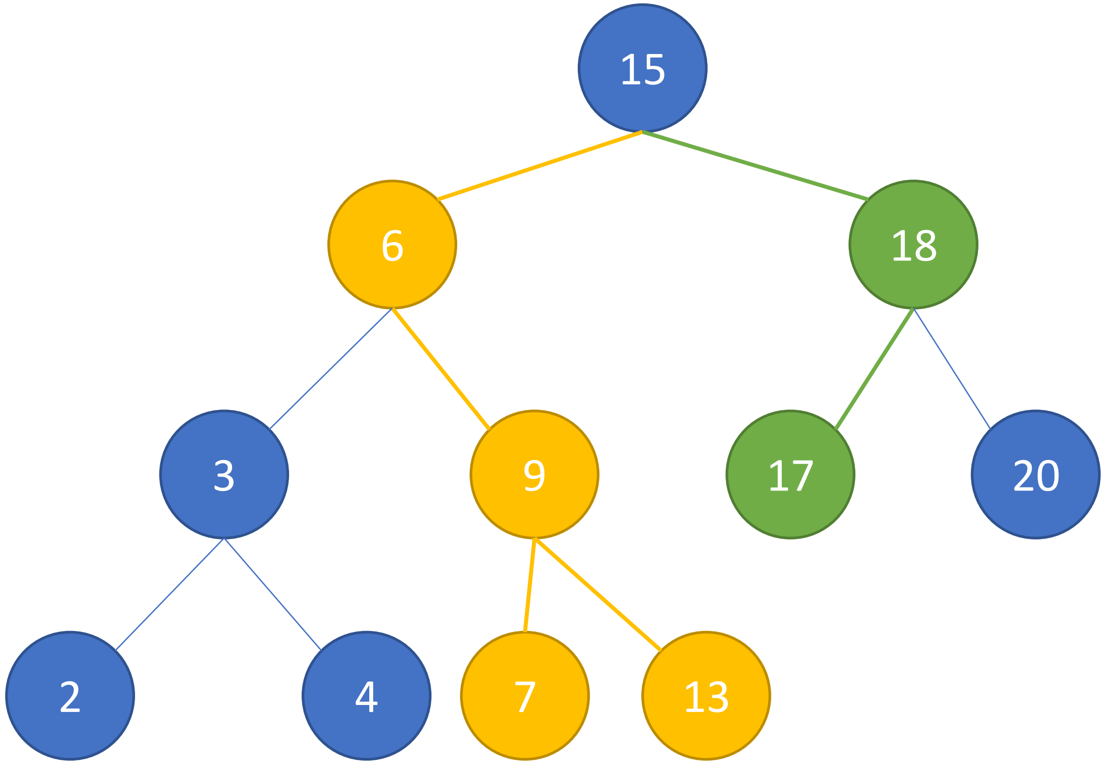

Not an AVL tree

Adelson-Velsky, Georgy; Landis, Evgenii. http://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=dan&paperid=26964&option_lang=rus (rus)

# AVL tree

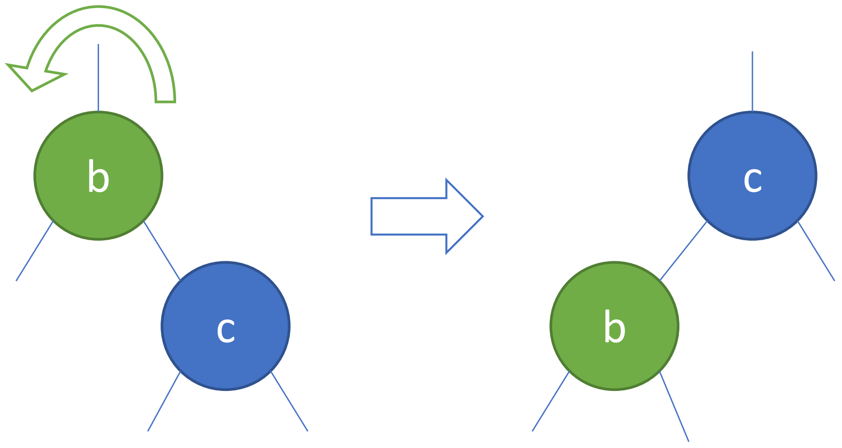
- **A**delson-**V**elsky and **L**andis
- Add balance factor to a node – difference in height between right and left subtrees
- A tree is AVL whenever every node's balance factor $\in \{1, 0, -1\}$
- Changing tree structure (introducing new or removing elements) may disturb AVL invariant, so a *rebalancing* is required

$$balance\ factor = 3 - 2 = -1$$



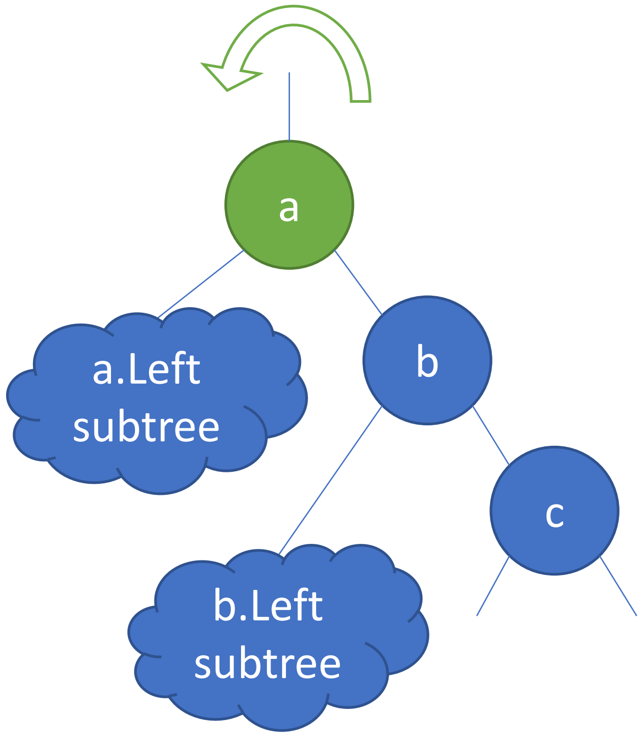
# Tree rotation

- Change tree structure without changing elements order



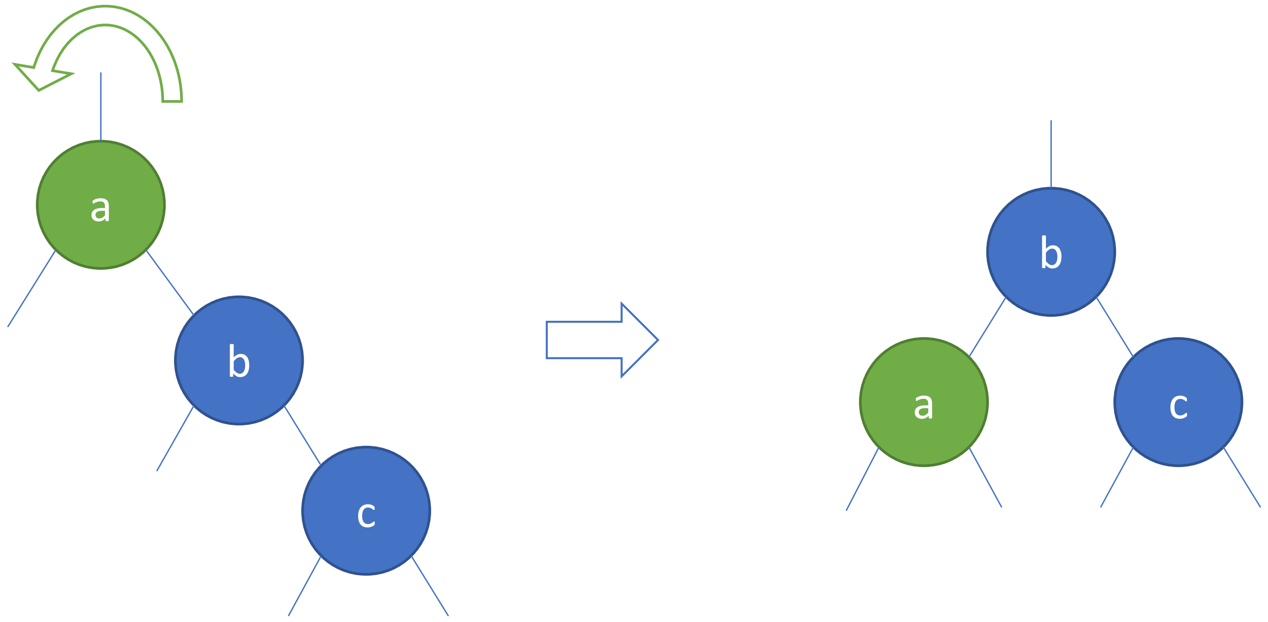
# Tree rotation

- Change tree structure without changing elements order



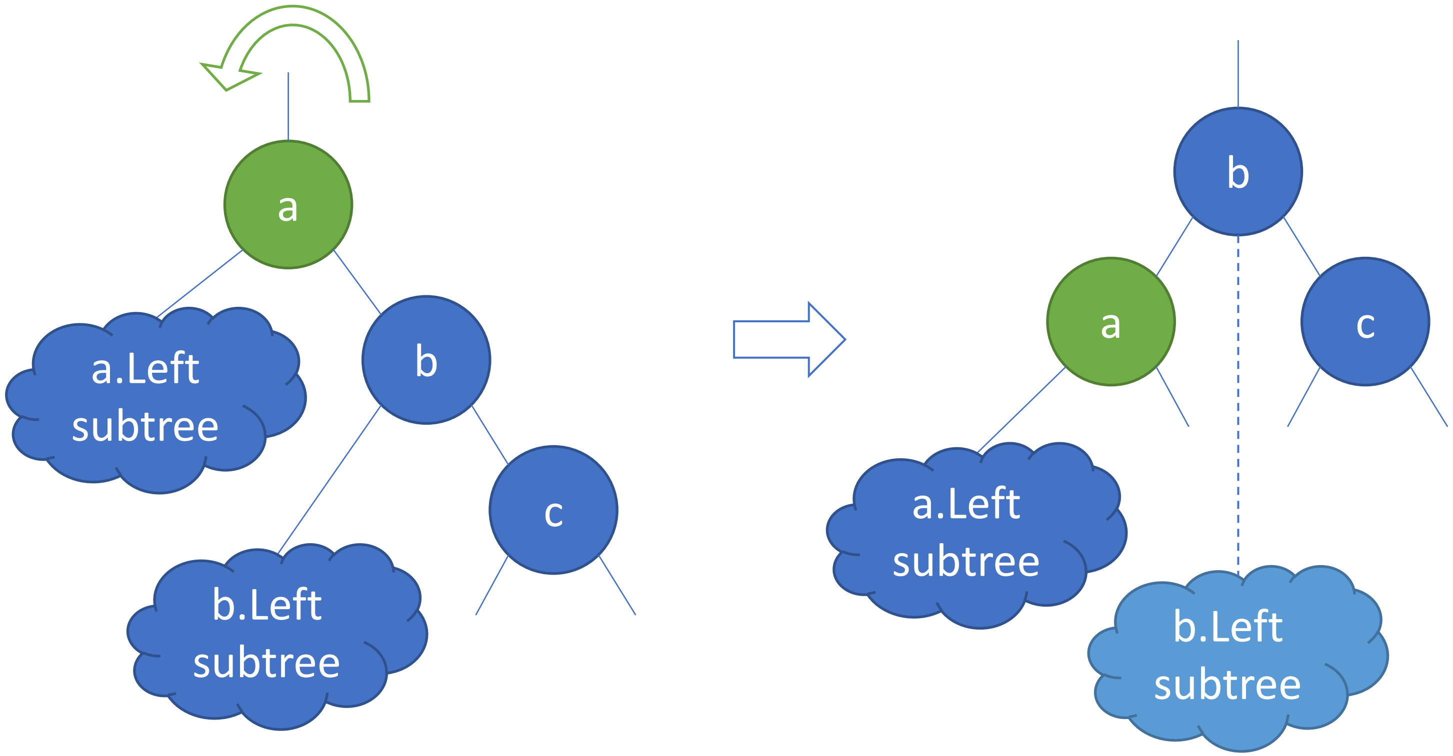
# Tree rotation

- Change tree structure without changing elements order



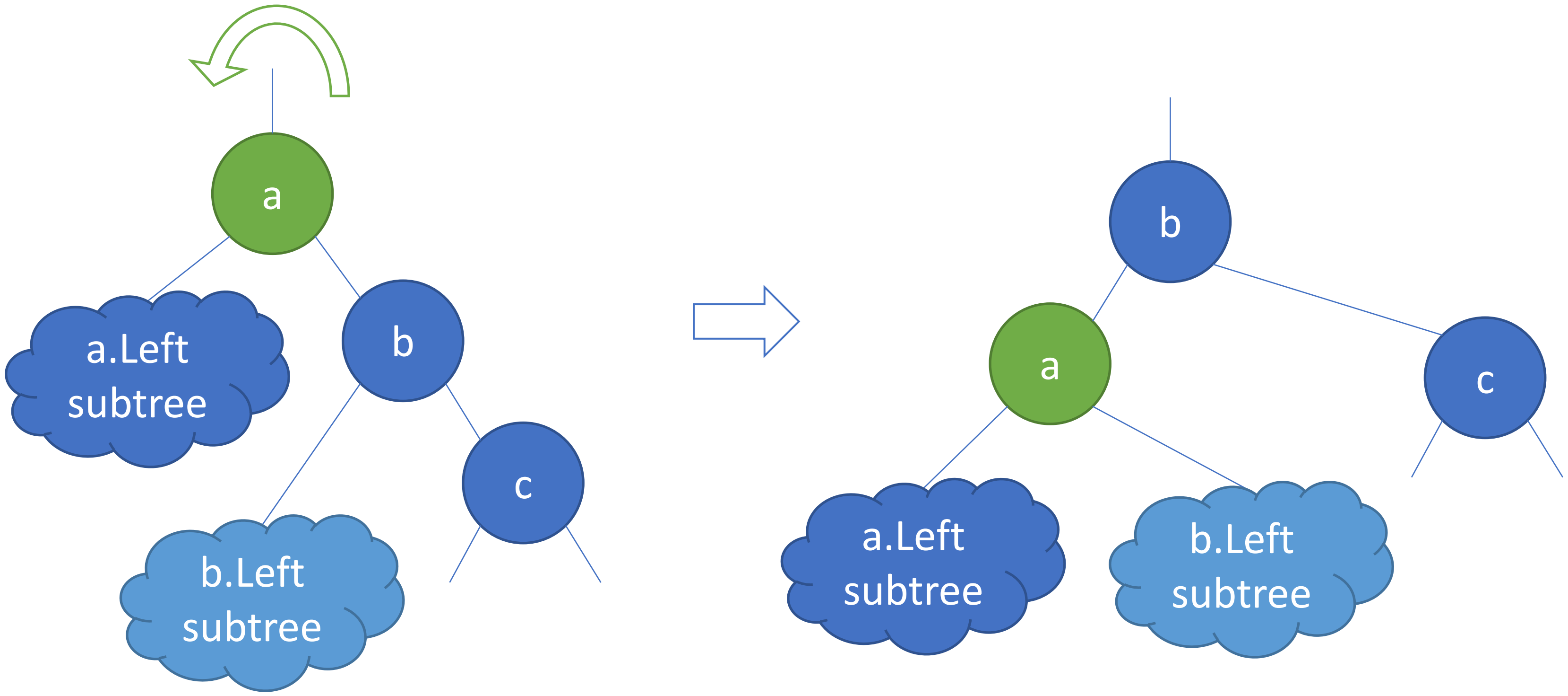
# Tree rotation

- Change tree structure without changing elements order



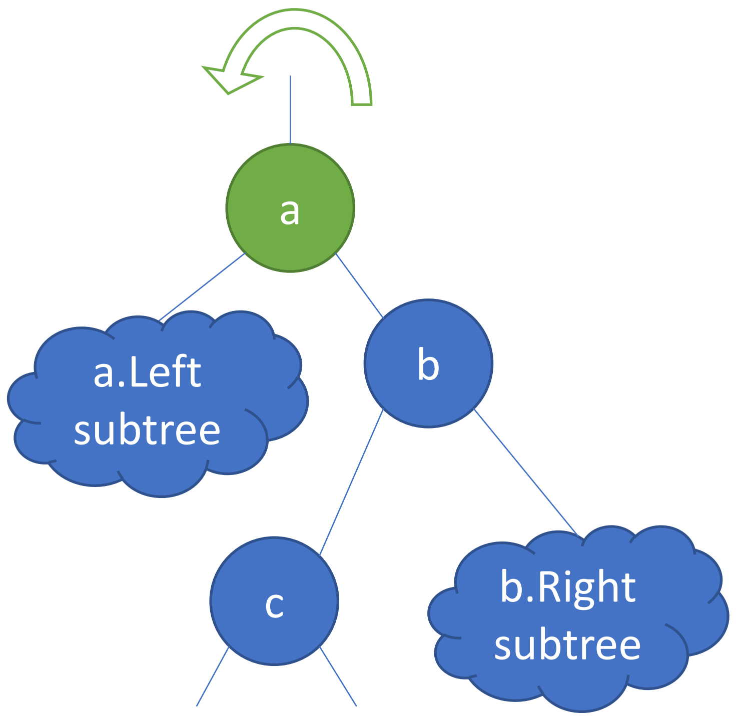
# Tree rotation

- Change tree structure without changing elements order



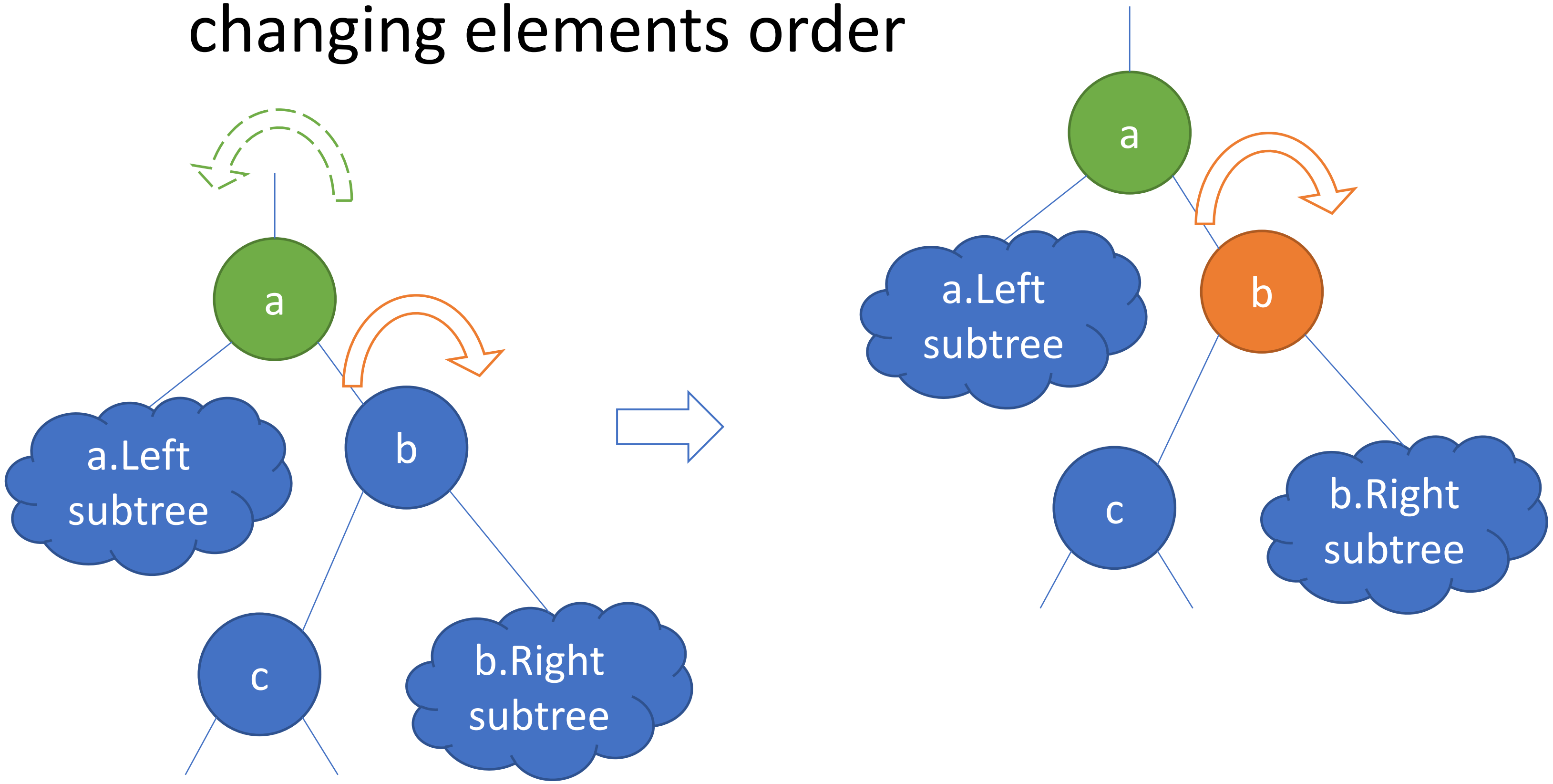
# Tree rotation

- Change tree structure without changing elements order



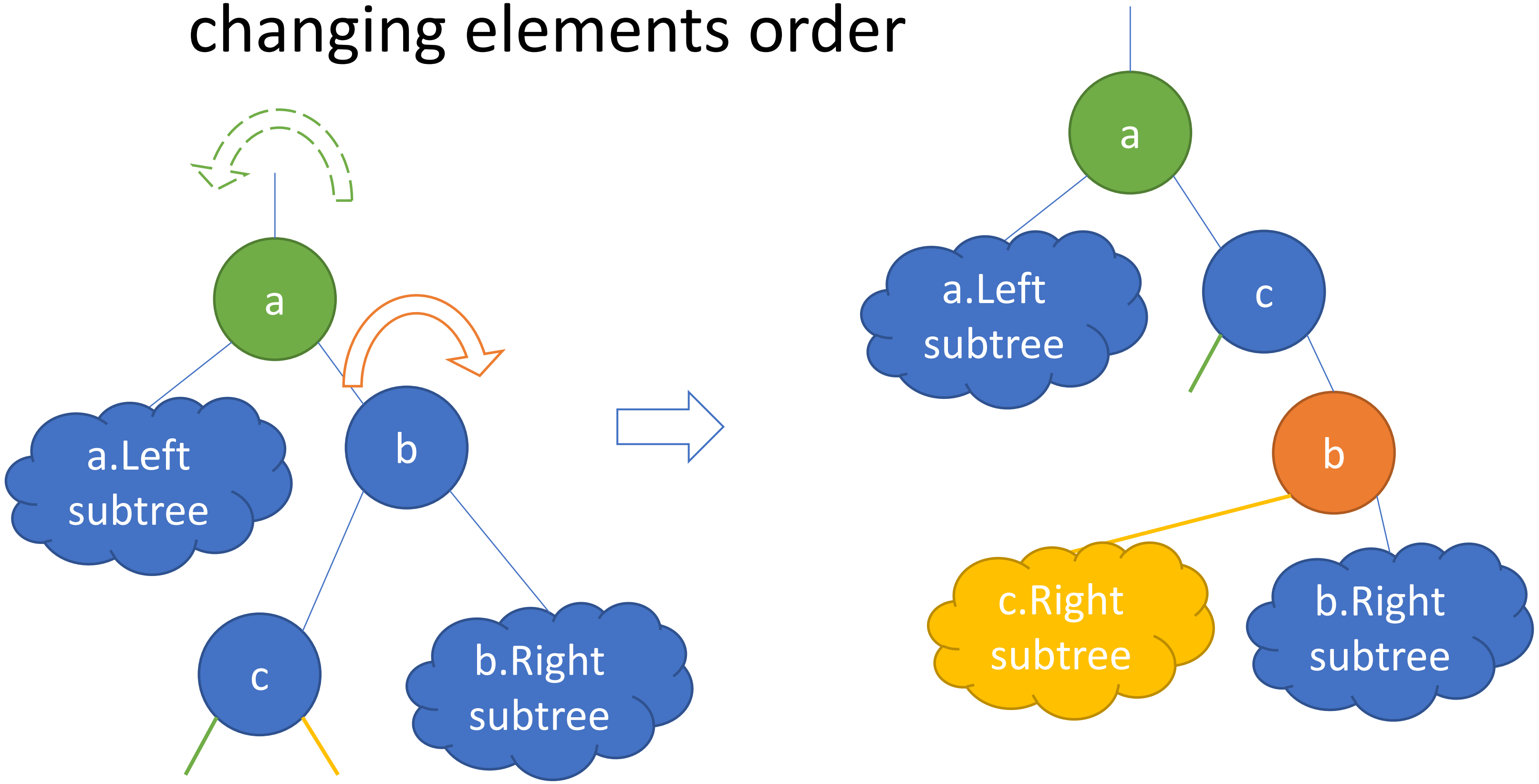
# Tree rotation

- Change tree structure without changing elements order



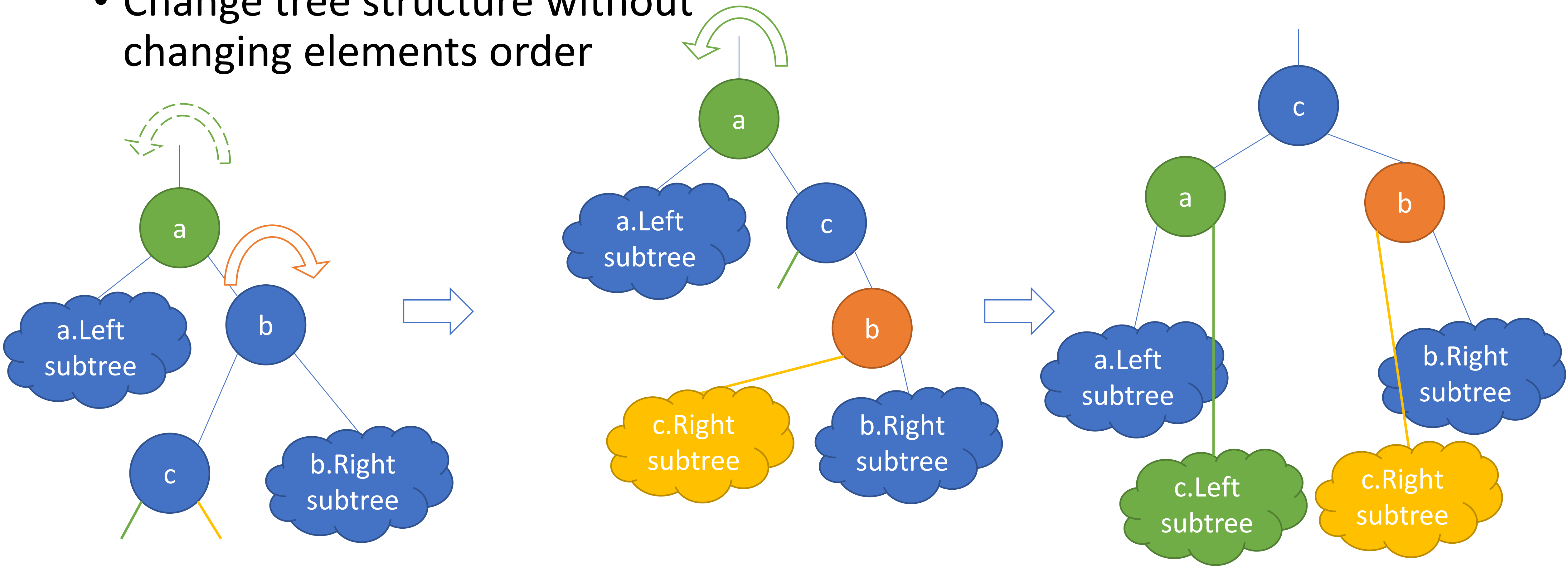
# Tree rotation

- Change tree structure without changing elements order



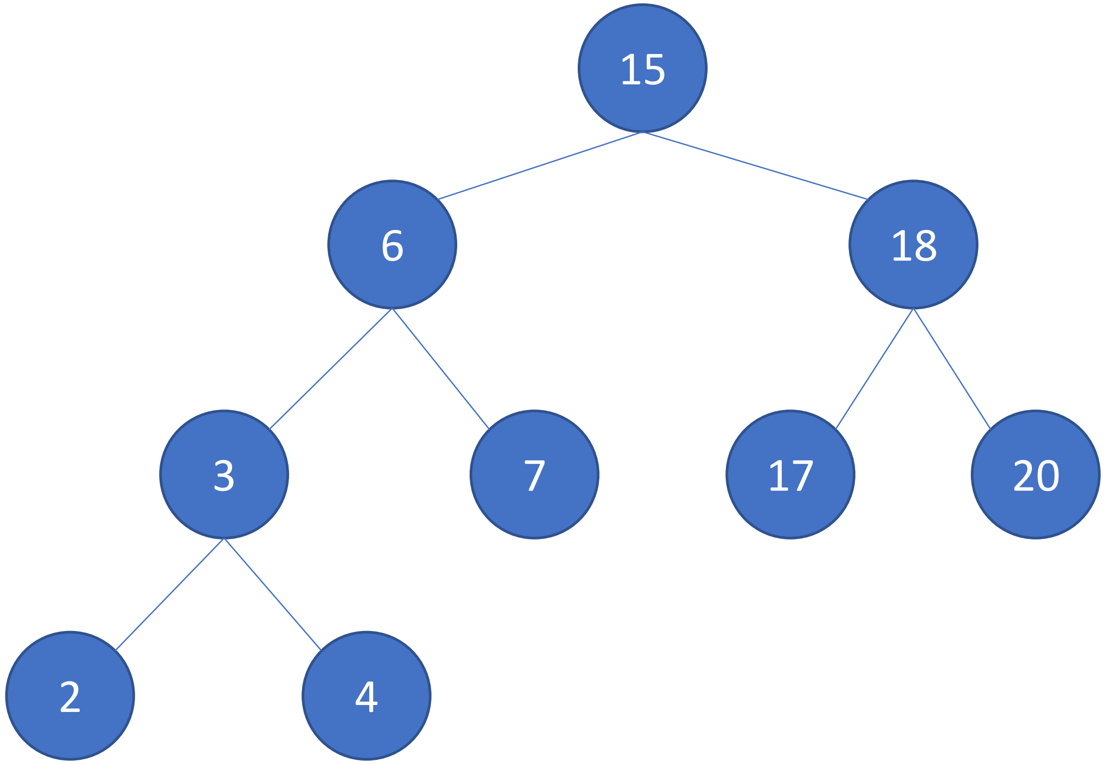
# Tree rotation

- Change tree structure without changing elements order



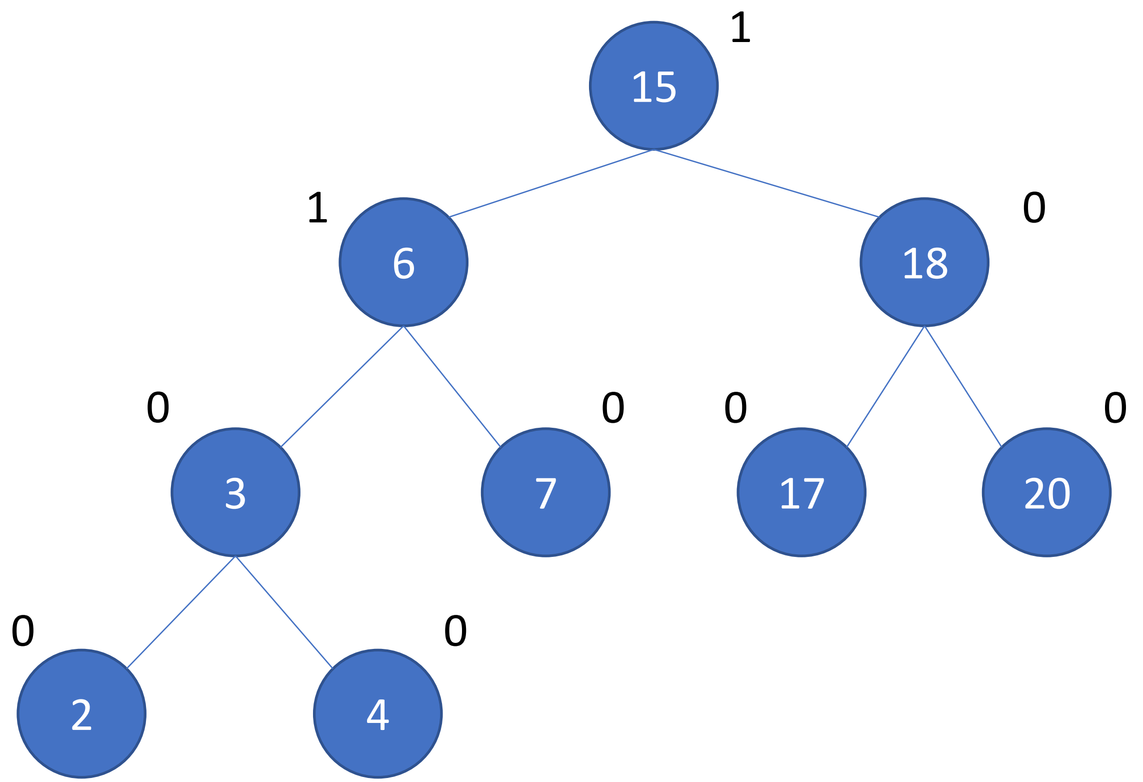
# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed



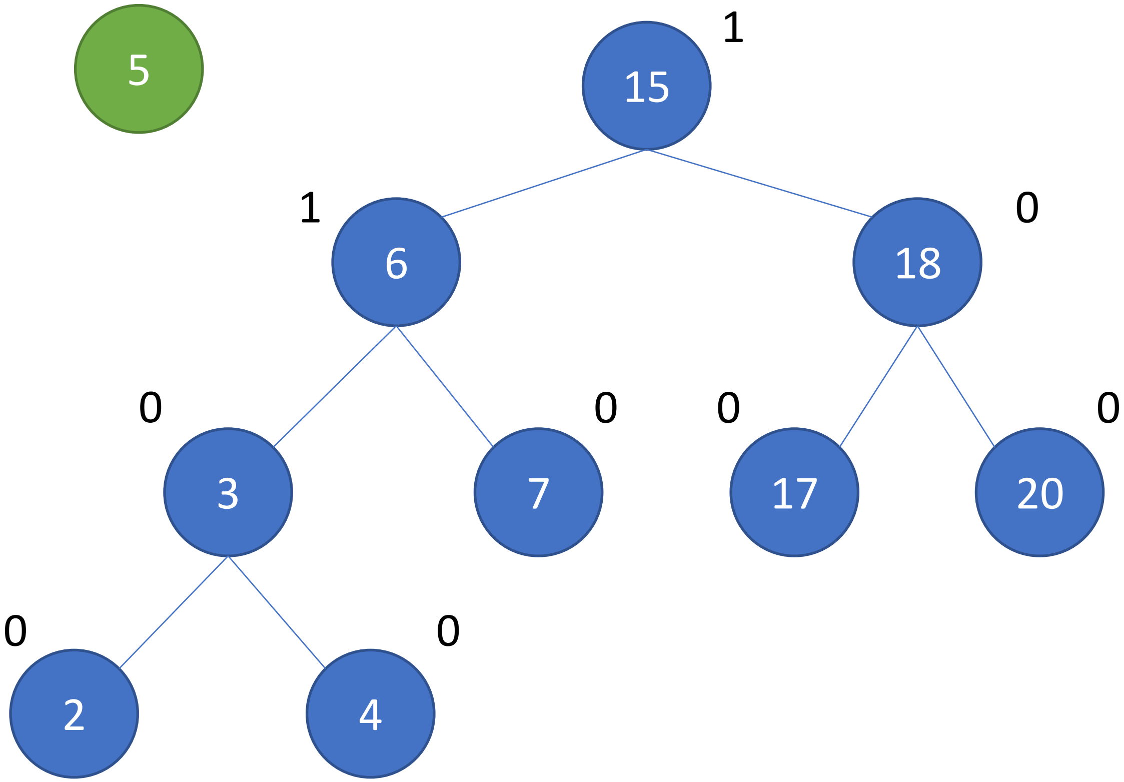
# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed



# AVL tree: Insertion

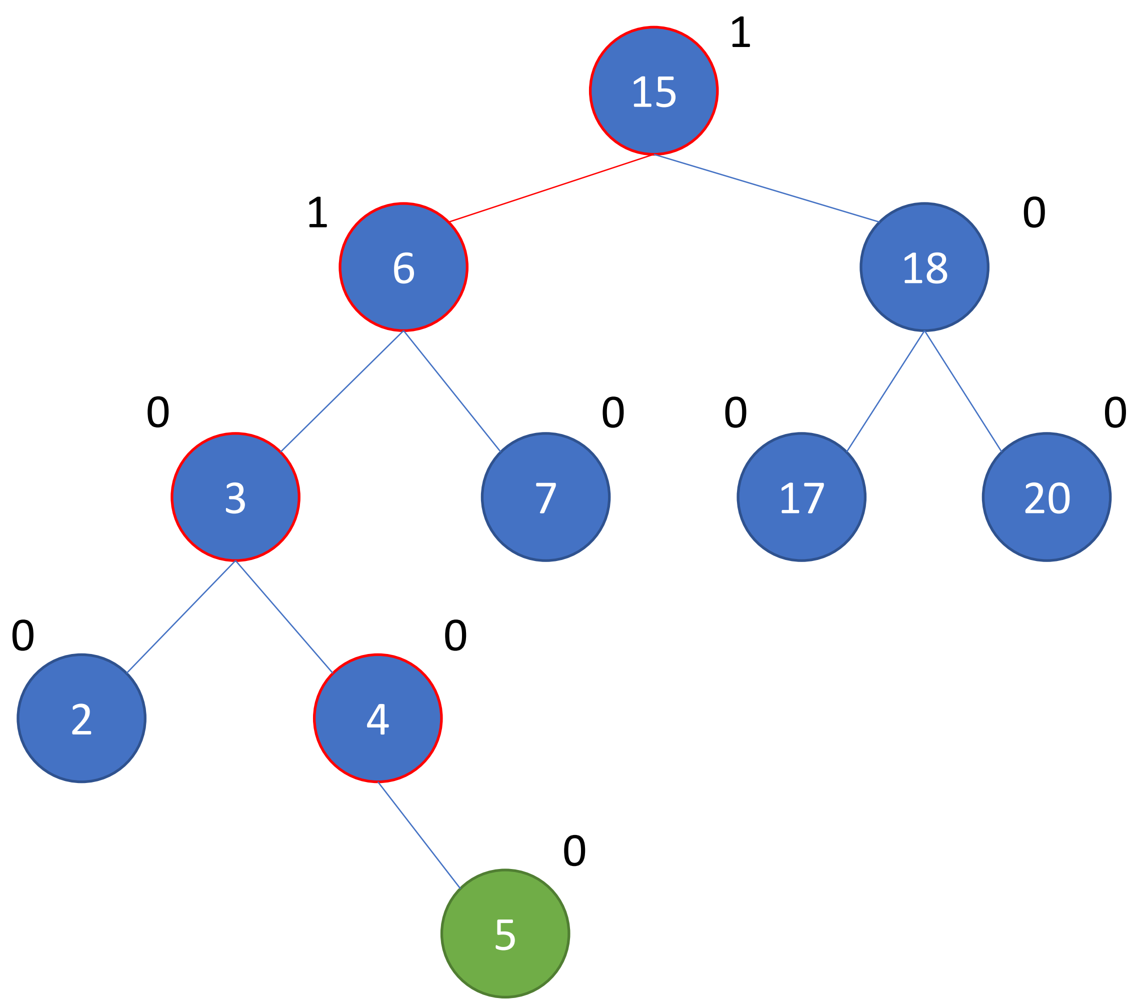
Idea: use BST algorithm, but rebalance the tree with rotations whenever needed



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

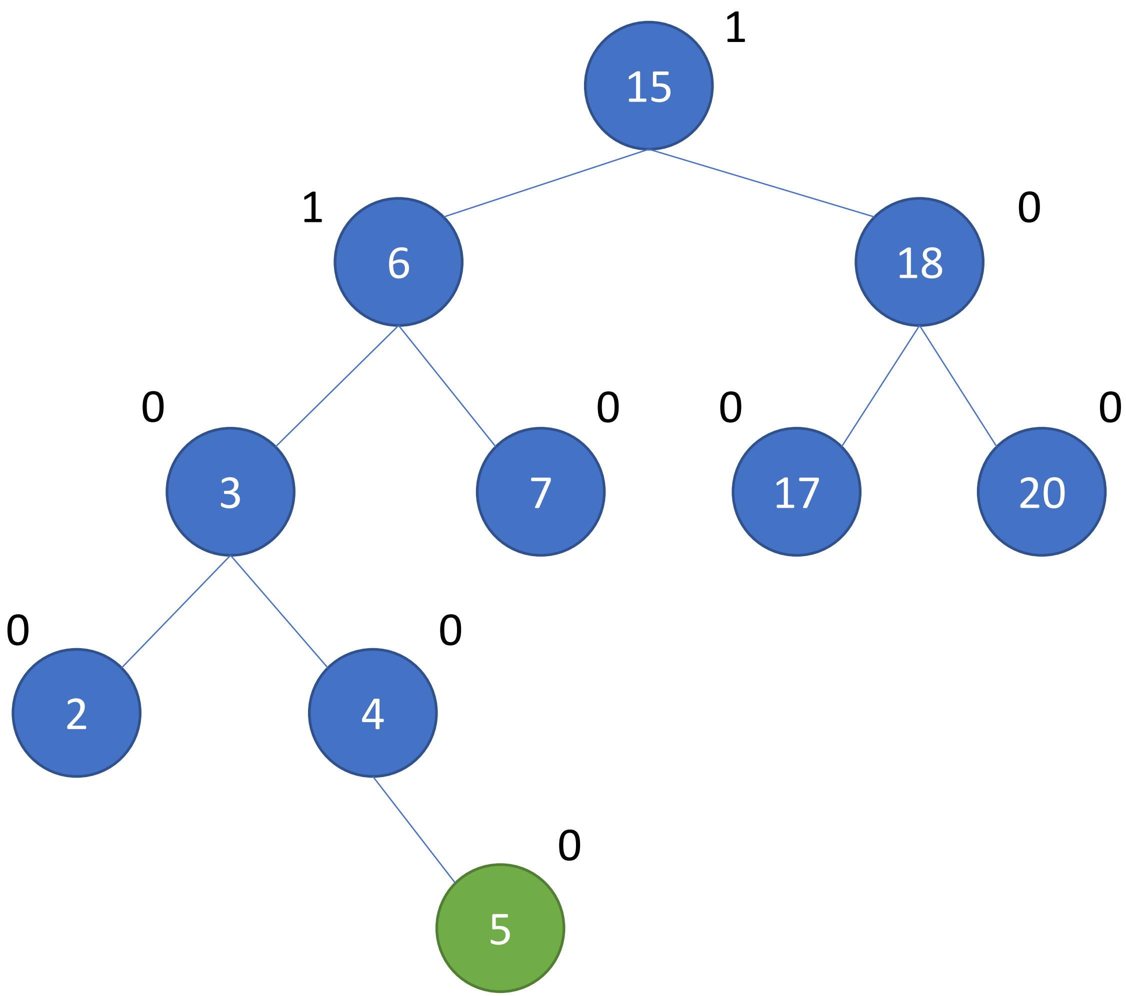
- **Insert as in BST**
- Go up, updating weights and restoring AVL invariant



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

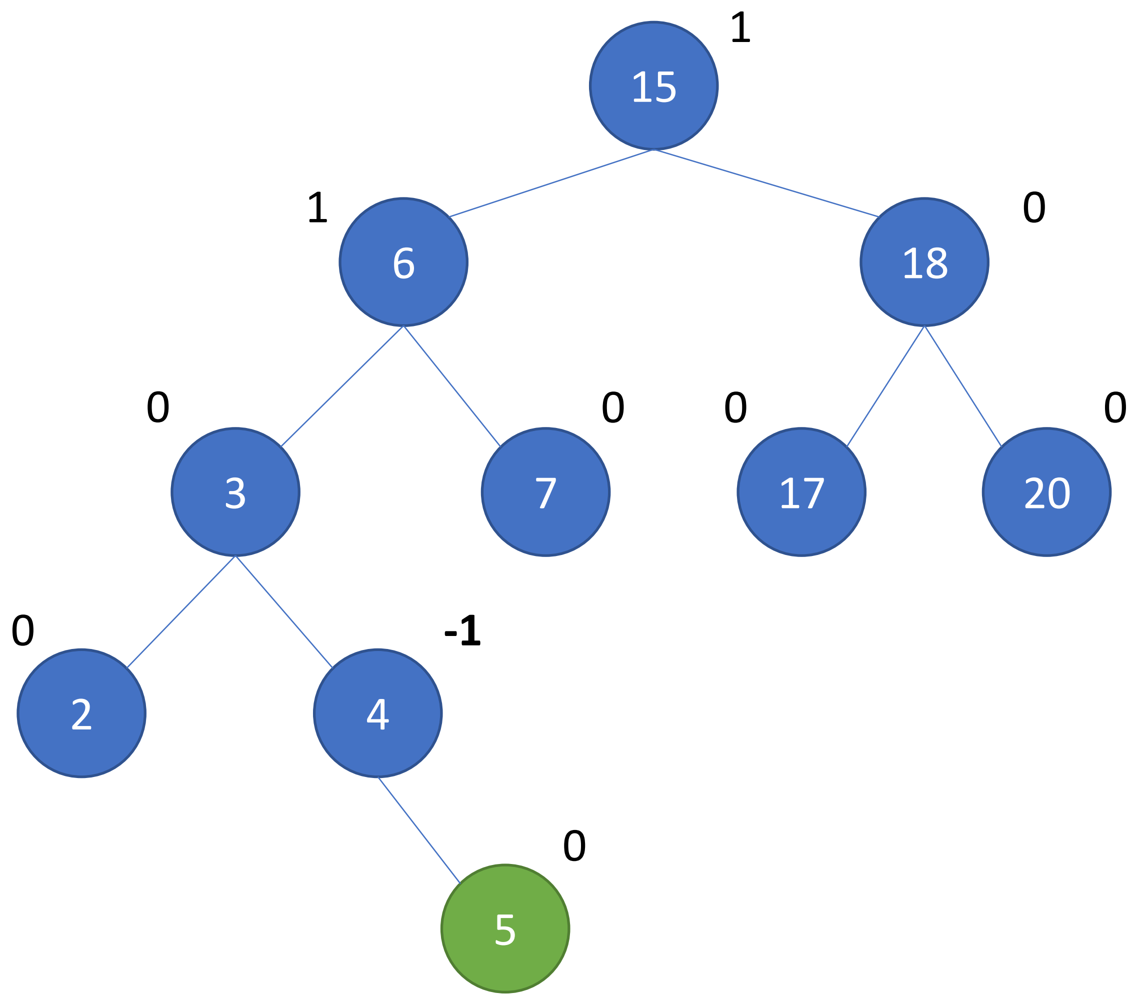
- **Insert as in BST**
- Go up, updating weights and restoring AVL invariant
    - Rotate lowest unbalanced node



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

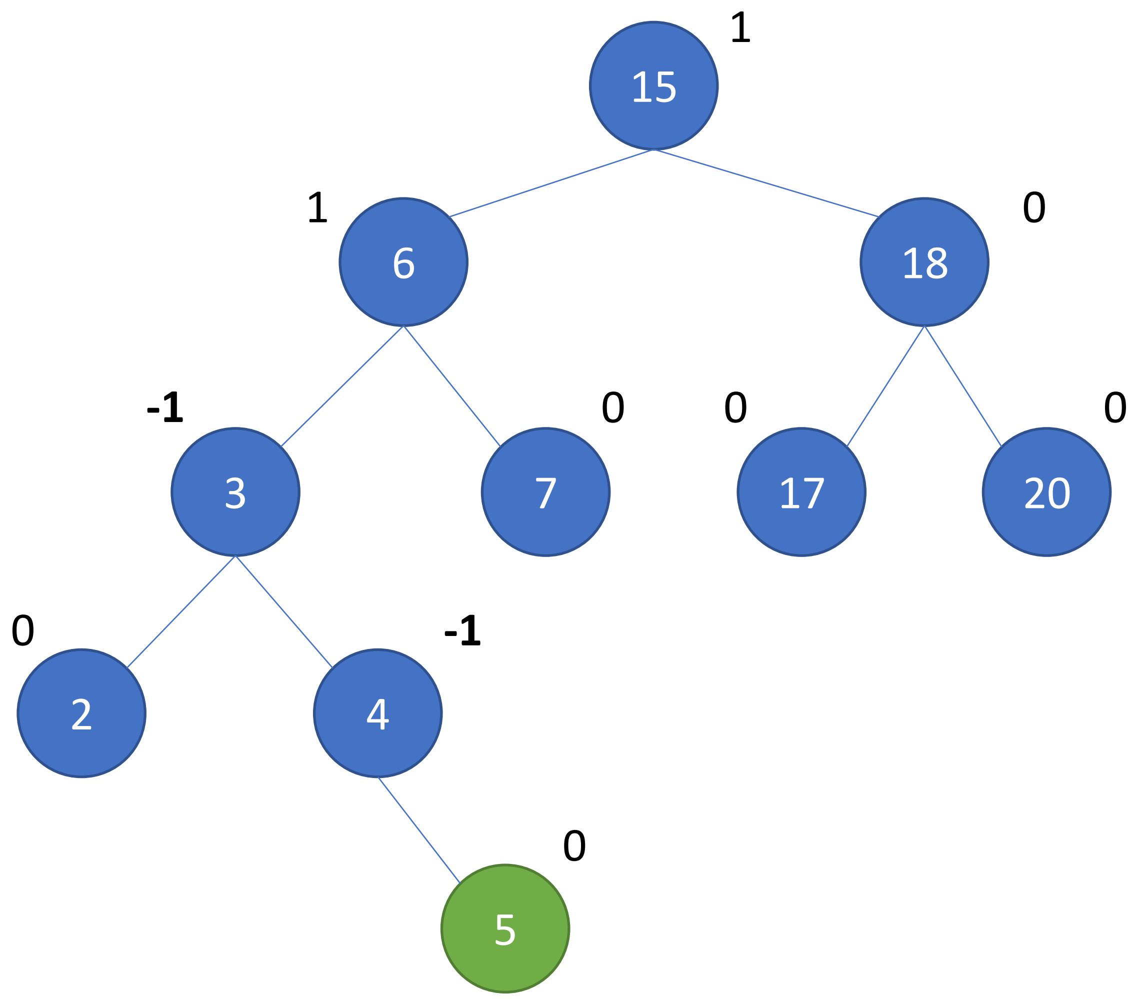
- Insert as in BST
- **Go up, updating weights and restoring AVL invariant**
    - Rotate lowest unbalanced node



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

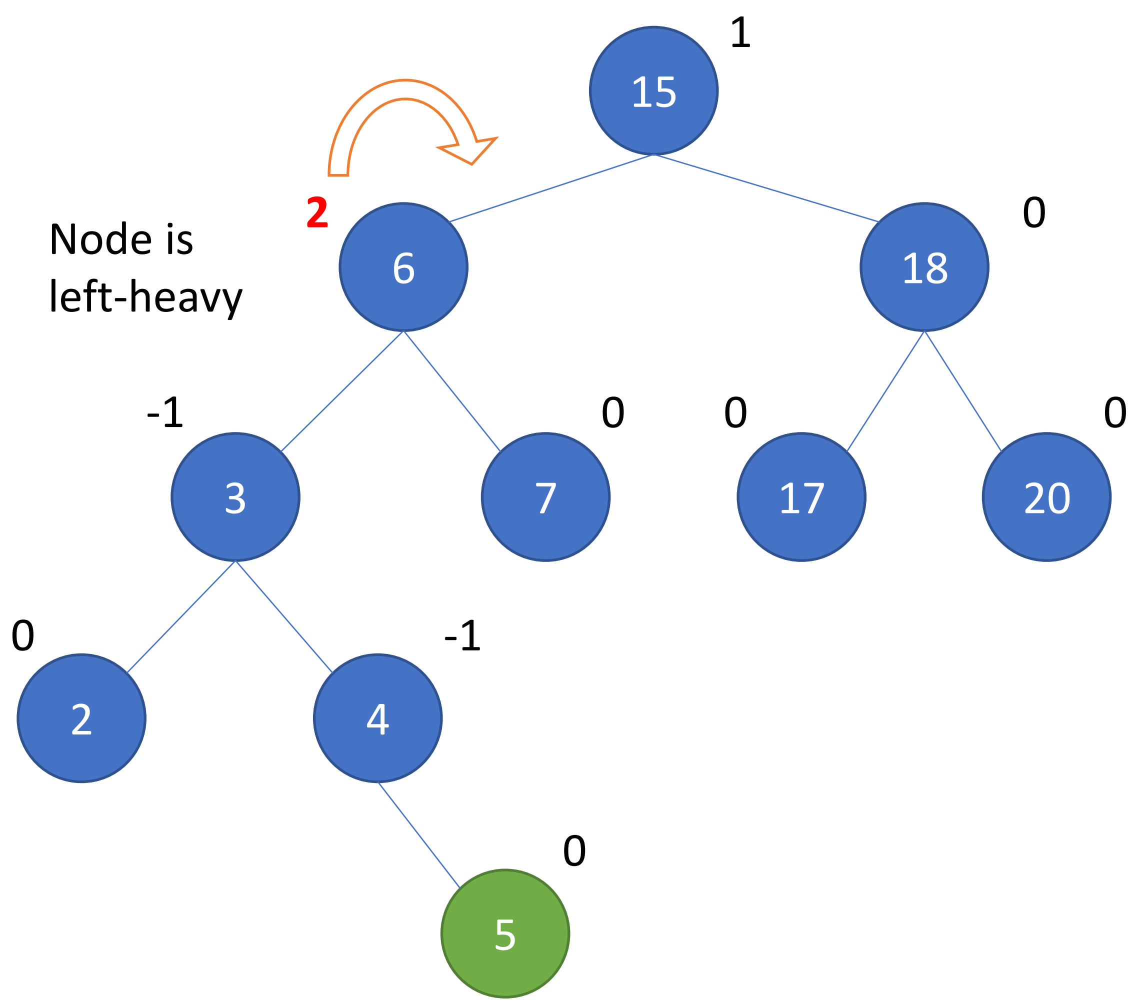
- Insert as in BST
- **Go up, updating weights and restoring AVL invariant**
  - Rotate lowest unbalanced node



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

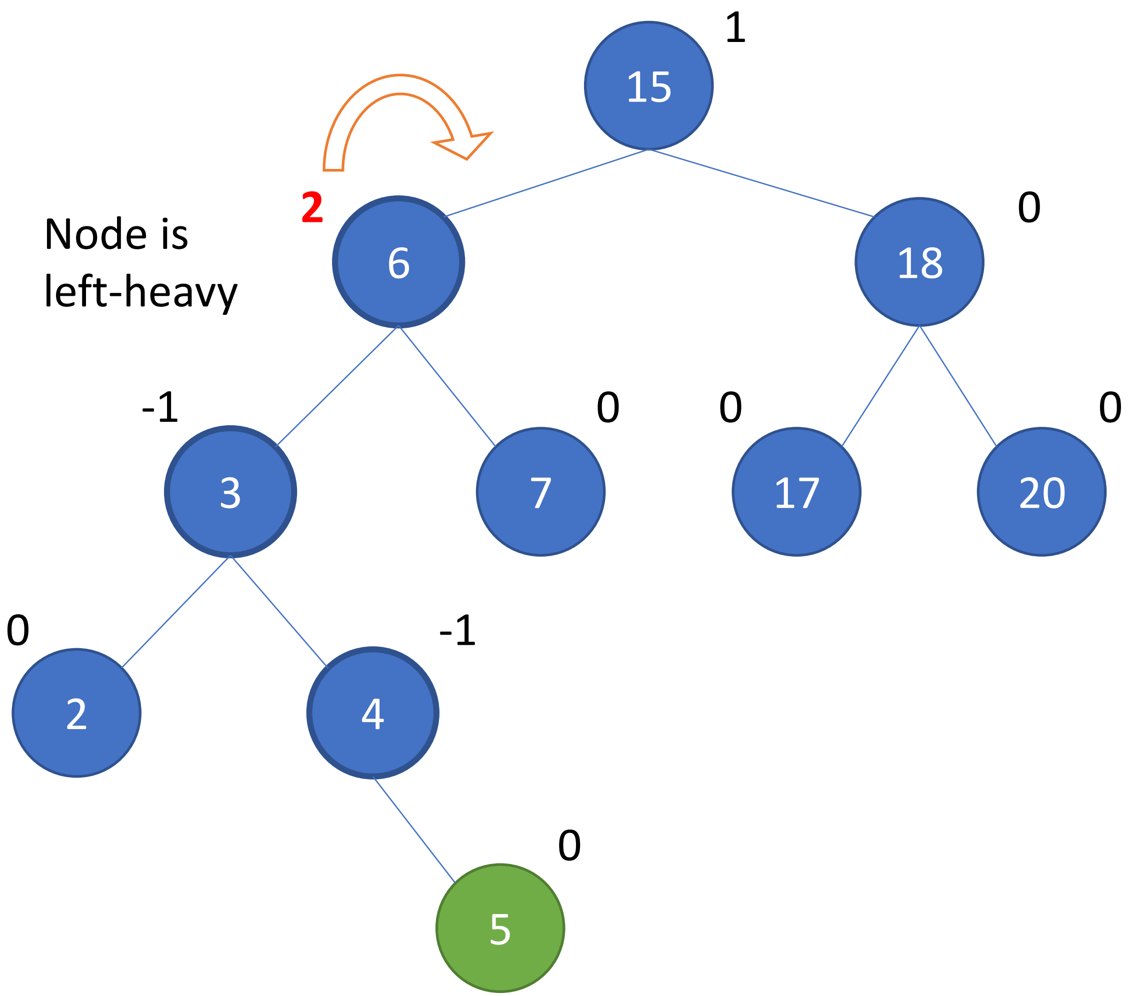
- Insert as in BST
- Go up, updating weights and restoring AVL invariant
  - **Rotate lowest unbalanced node**



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

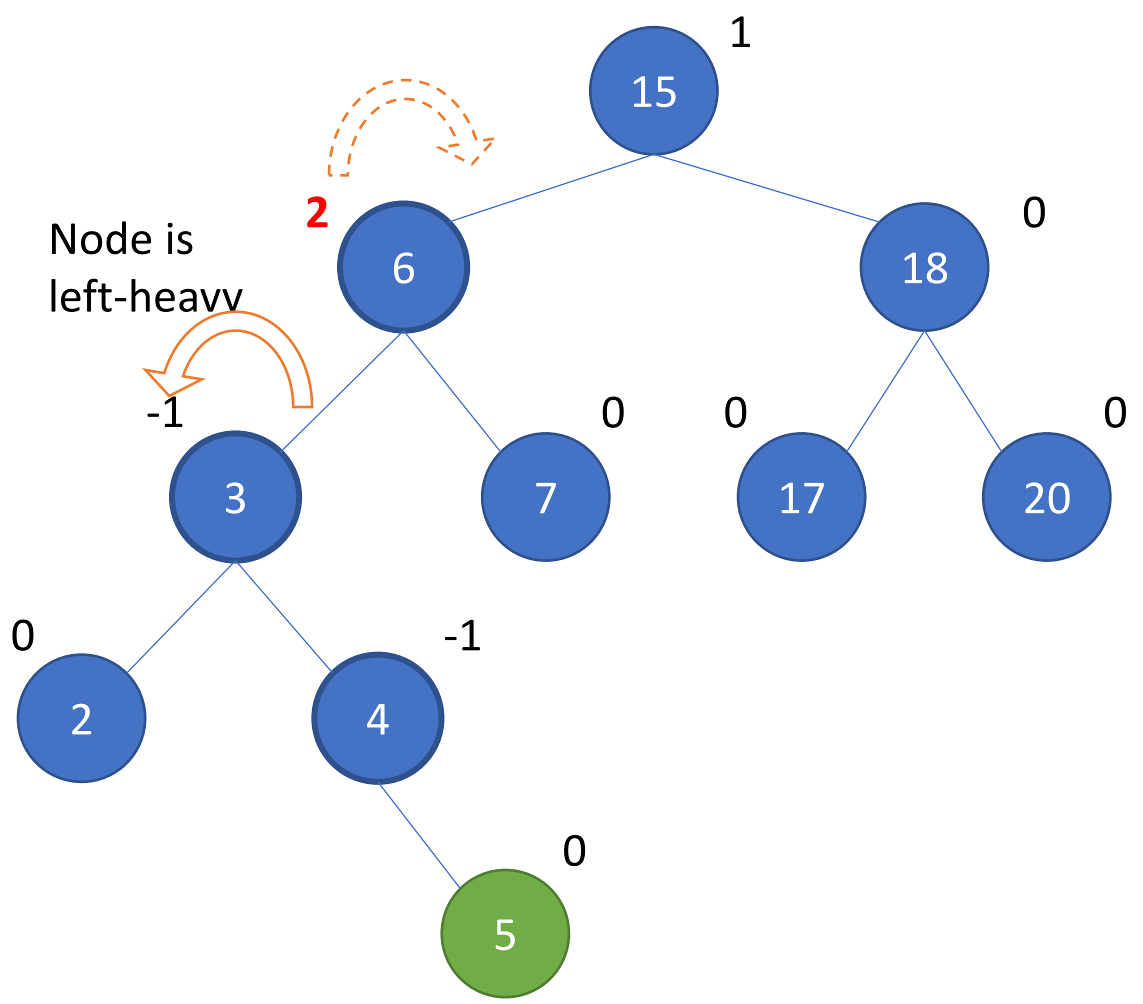
- Insert as in BST
- Go up, updating weights and restoring AVL invariant
  - **Rotate lowest unbalanced node**



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

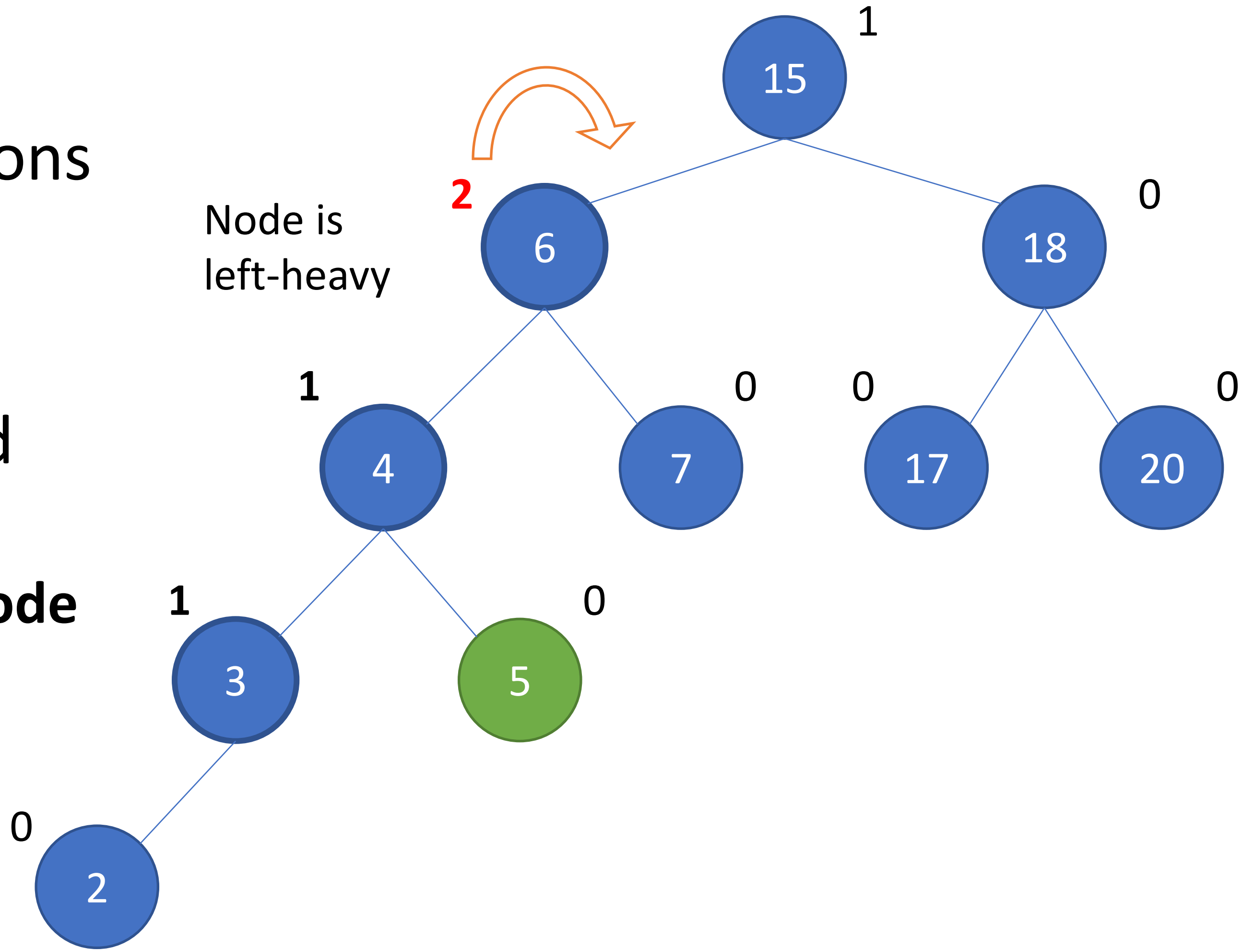
- Insert as in BST
- Go up, updating weights and restoring AVL invariant
  - **Rotate lowest unbalanced node**



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

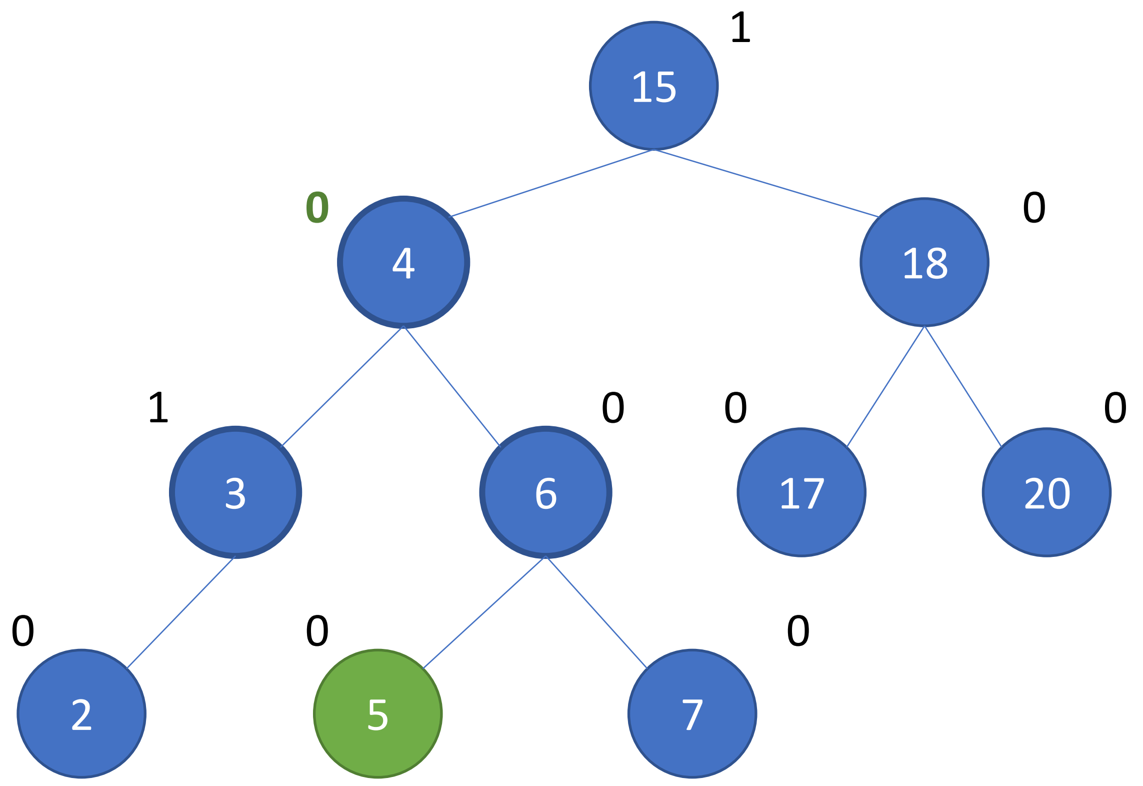
- Insert as in BST
- Go up, updating weights and restoring AVL invariant
  - **Rotate lowest unbalanced node**



# AVL tree: Insertion

Idea: use BST algorithm, but rebalance the tree with rotations whenever needed

- Insert as in BST
- Go up, updating weights and restoring AVL invariant
  - Rotate lowest unbalanced node
  - May require multiple rotations (how many?)



# AVL tree: Deletion

~Same idea

- Delete a node as if it was a BST
- Go up, updating and rebalancing

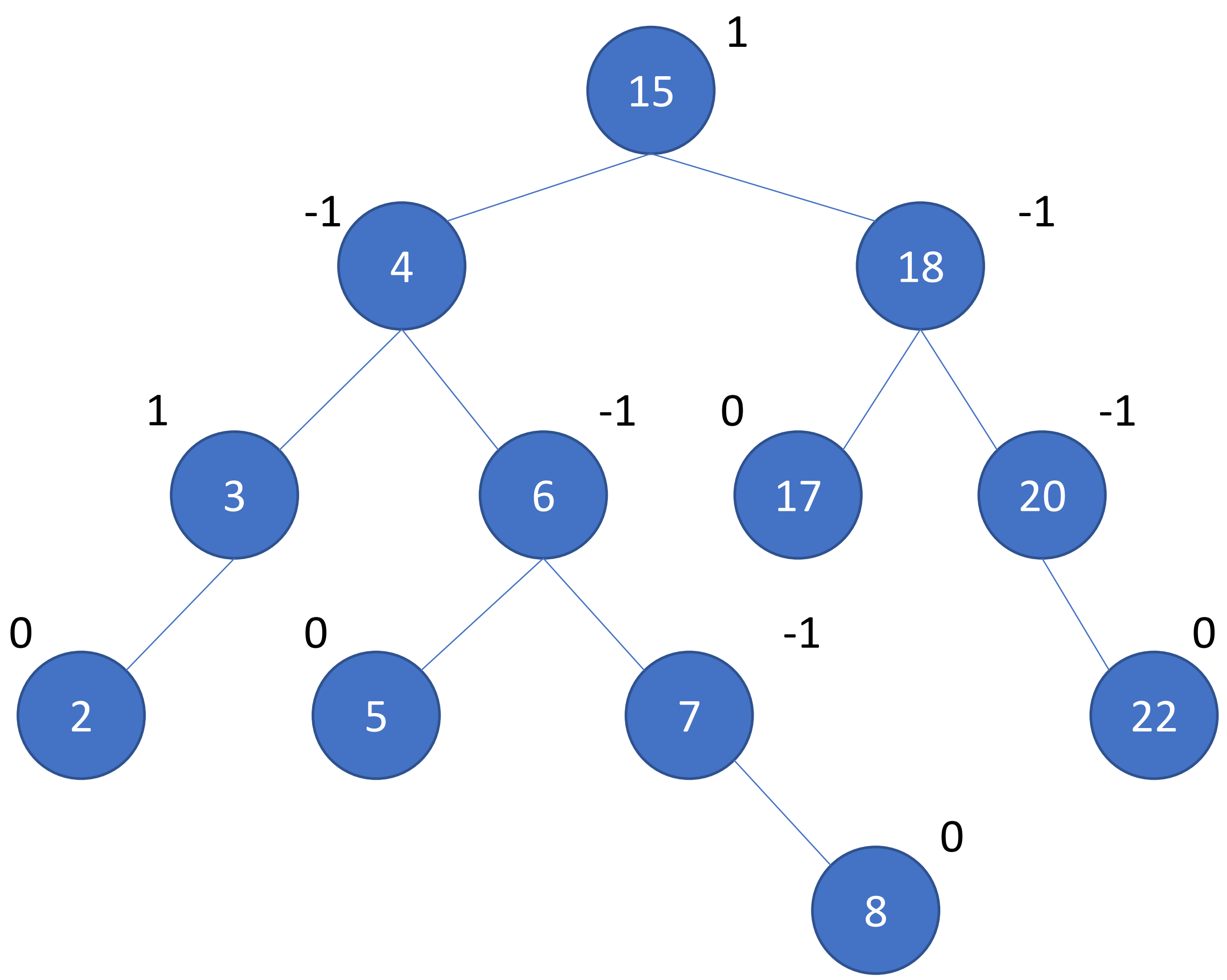
# AVL tree: Height

How tall is an AVL tree?

# AVL tree: Height

How tall is an AVL tree?

- Worst case: all (internal) nodes have balance factor of $\pm 1$

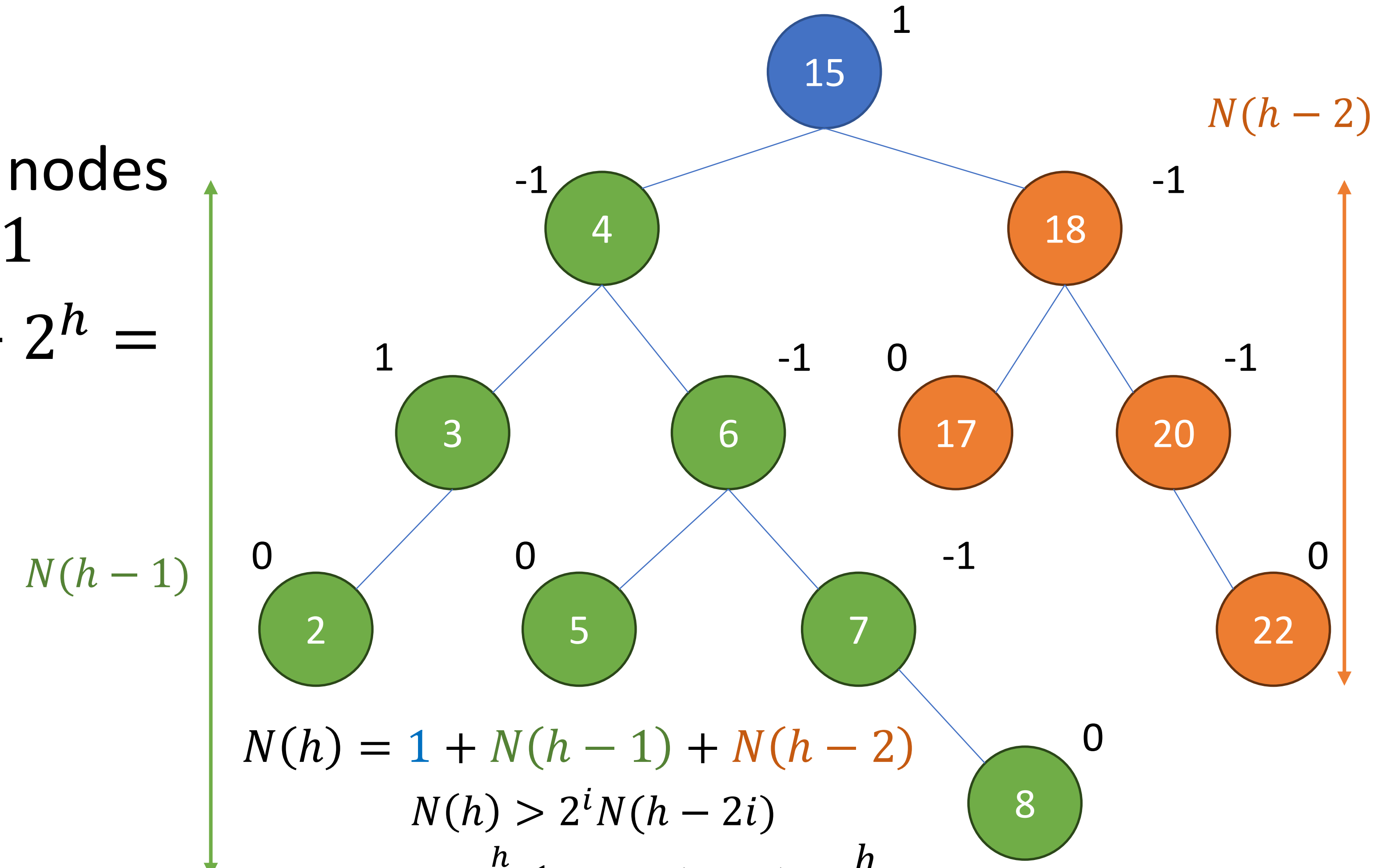


# AVL tree: Height

$N(1) = 1, N(2) = 2$

How tall is an AVL tree?

- Worst case: all (internal) nodes have balance factor of $\pm 1$
- Height h: $2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1 (\geq n)$
- $h \geq \log n$




$$N(h) = 1 + N(h-1) + N(h-2)$$

$$N(h) > 2^i N(h-2i)$$

$$N(h) > 2^{\frac{h}{2}-1} \rightarrow \log\big(N(h)\big) > \frac{h}{2} - 1$$

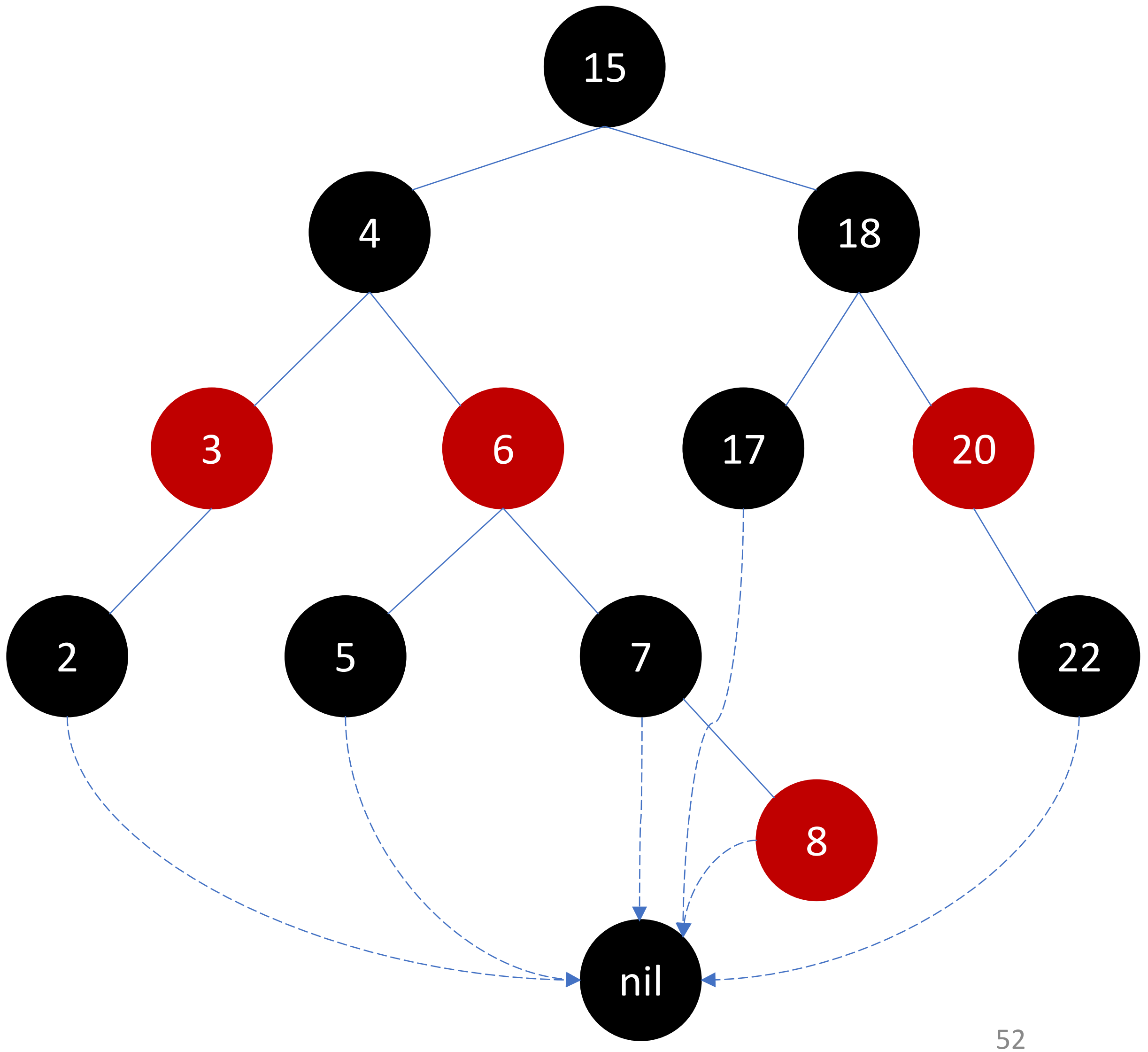
# AVL tree: Summary

- Guarantees logarithmic complexity
- Easy to implement

Just join for Parallel Ordered Sets https://dl.acm.org/doi/10.1145/2935764.2935768

# Red-black tree

- Additional bit for color: red or black
- Root & all leaf nodes are black
- Red node has only black children
- For each node, all simple paths to leaves have the same number of black nodes (bh – black height)
    - Height difference 2x max



# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

- Root is black
- Leaves are black
- Red’s children are black
- BH(left) == BH(right)

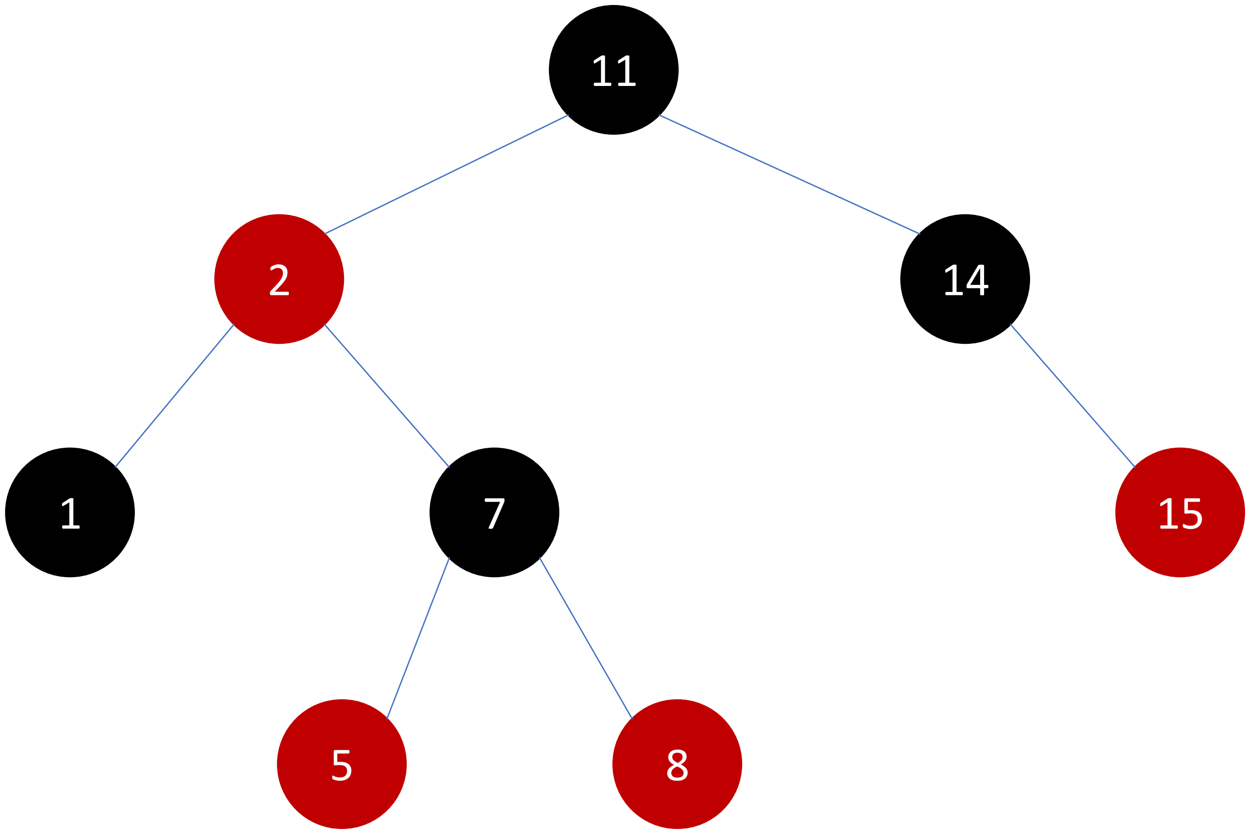
# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

- **Root is black**
- Leaves are black
- **Red’s children are black**
- BH(left) == BH(right)

# Red-black tree: insertion

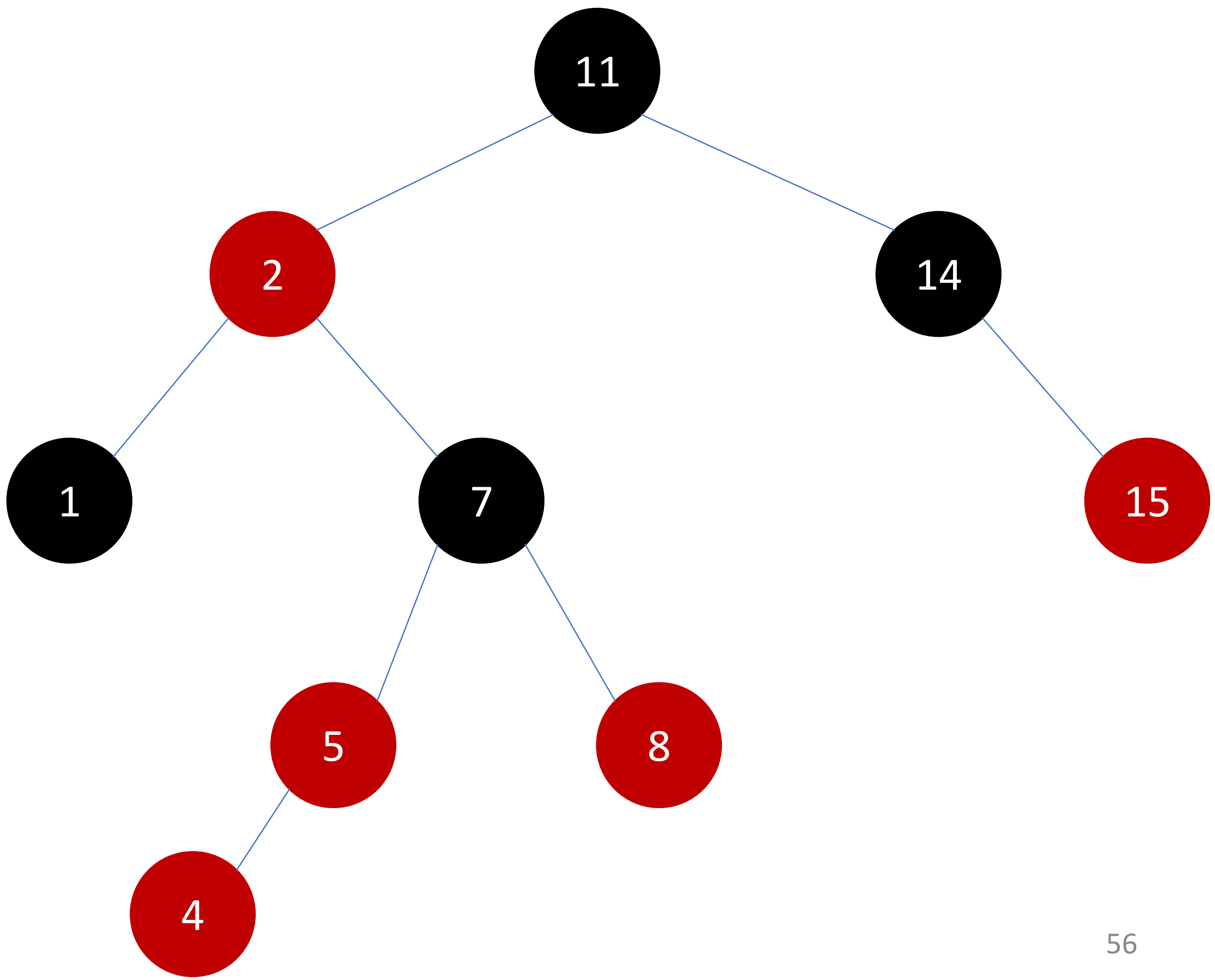
- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

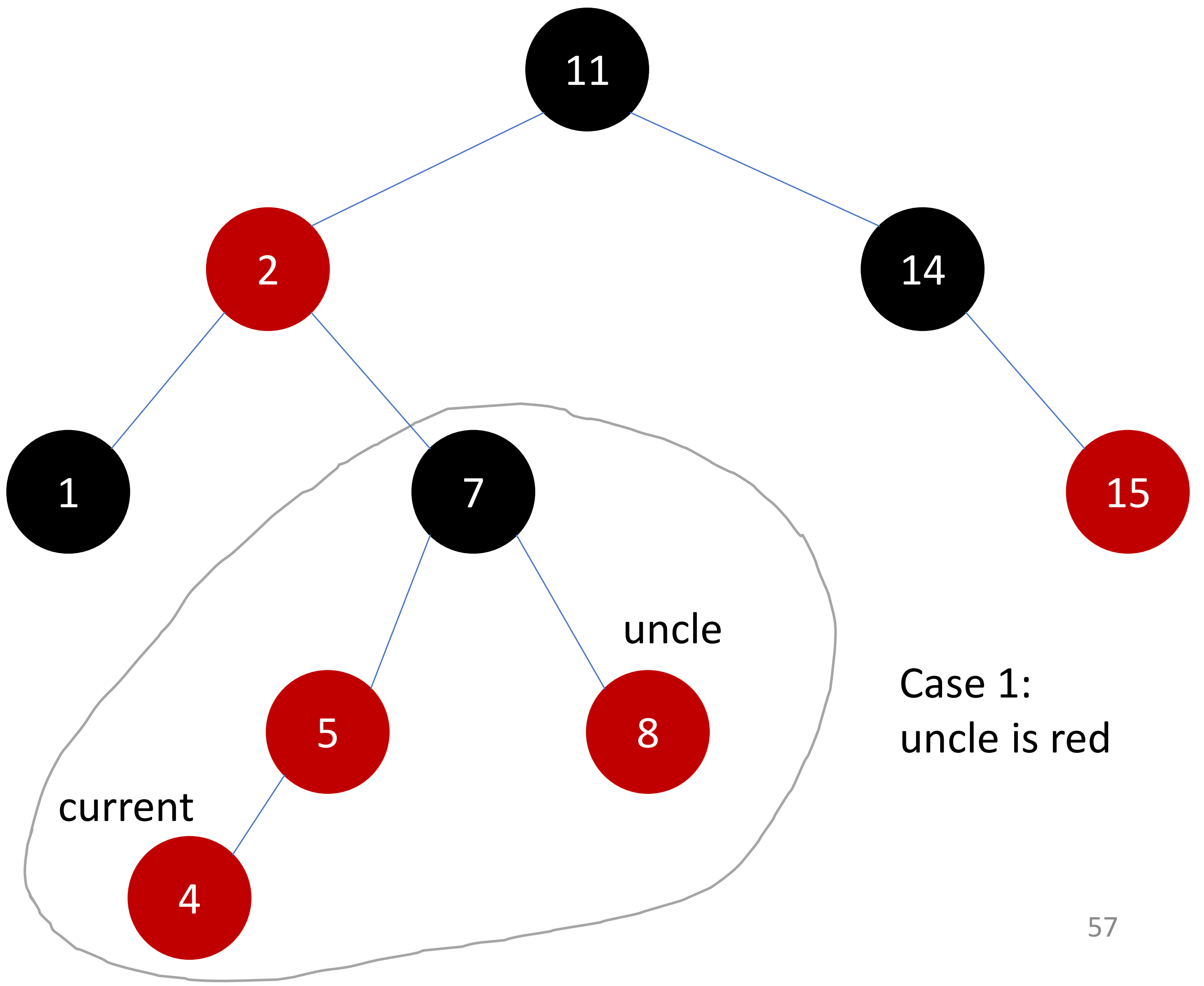
- Root & leaves are black
- **Red’s children are black**
- BH(left) == BH(right)



# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

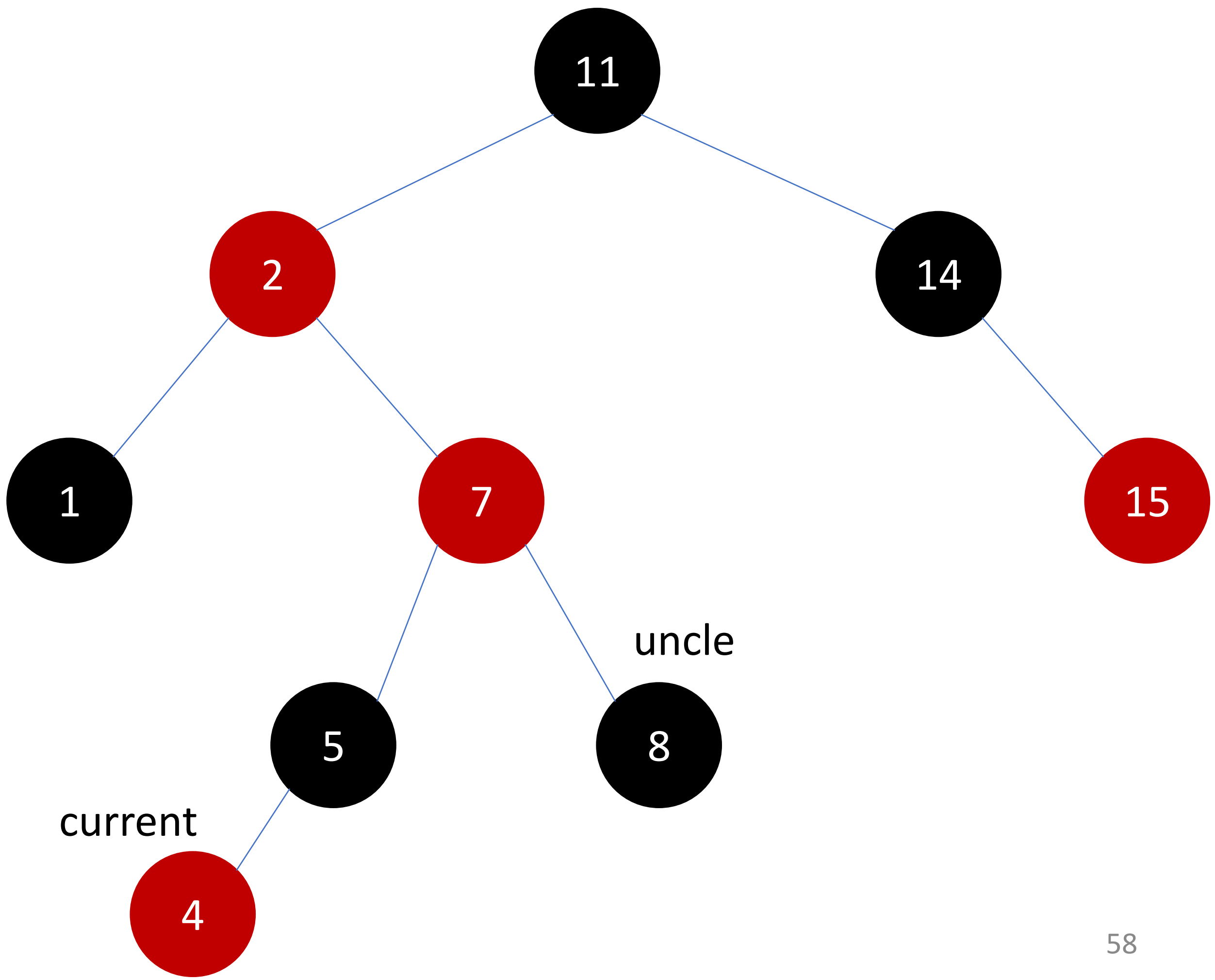
- Root & leaves are black
- **Red's children are black**
- BH(left) == BH(right)



# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

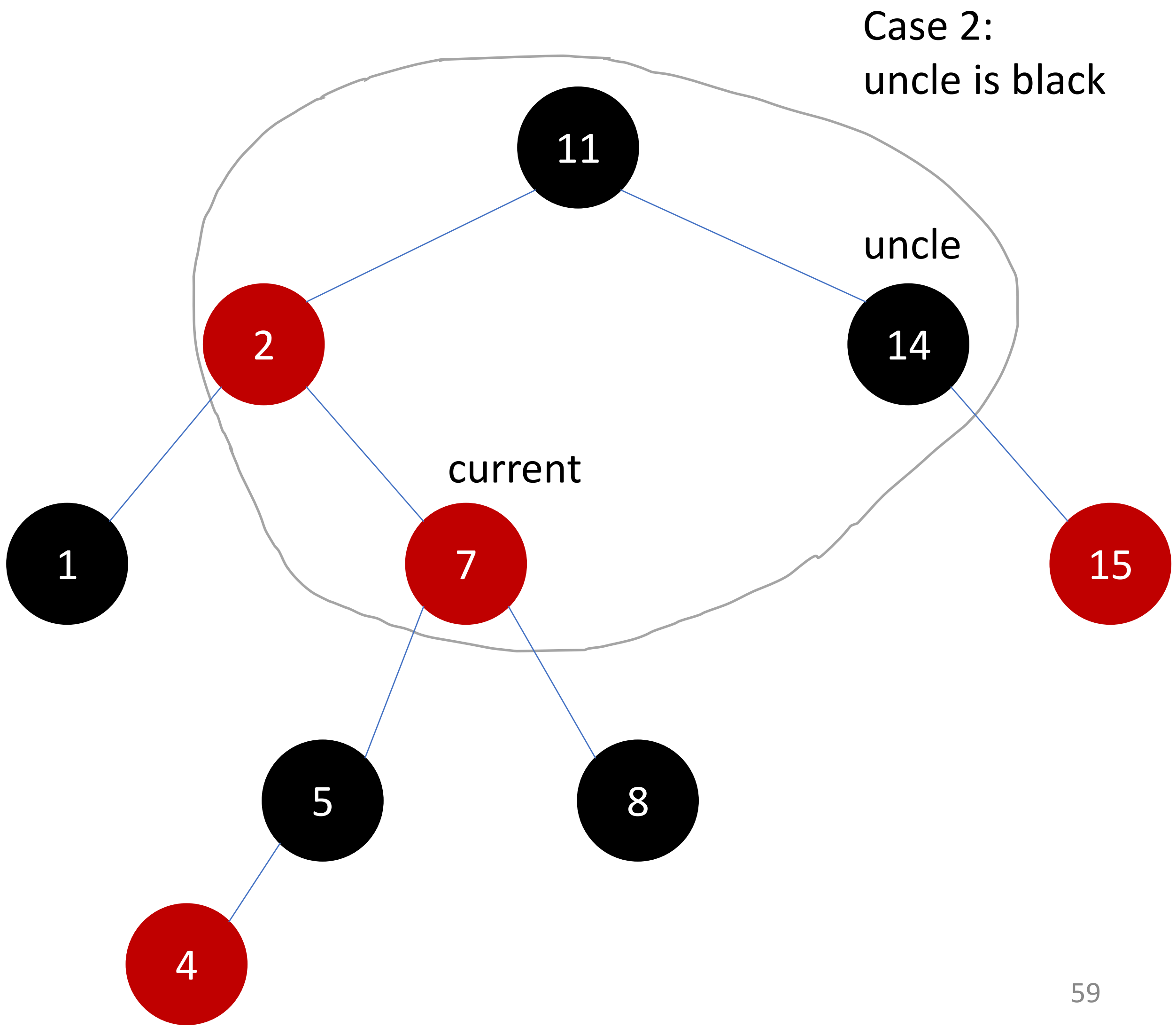
- Root & leaves are black
- **Red’s children are black**
- BH(left) == BH(right)



# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

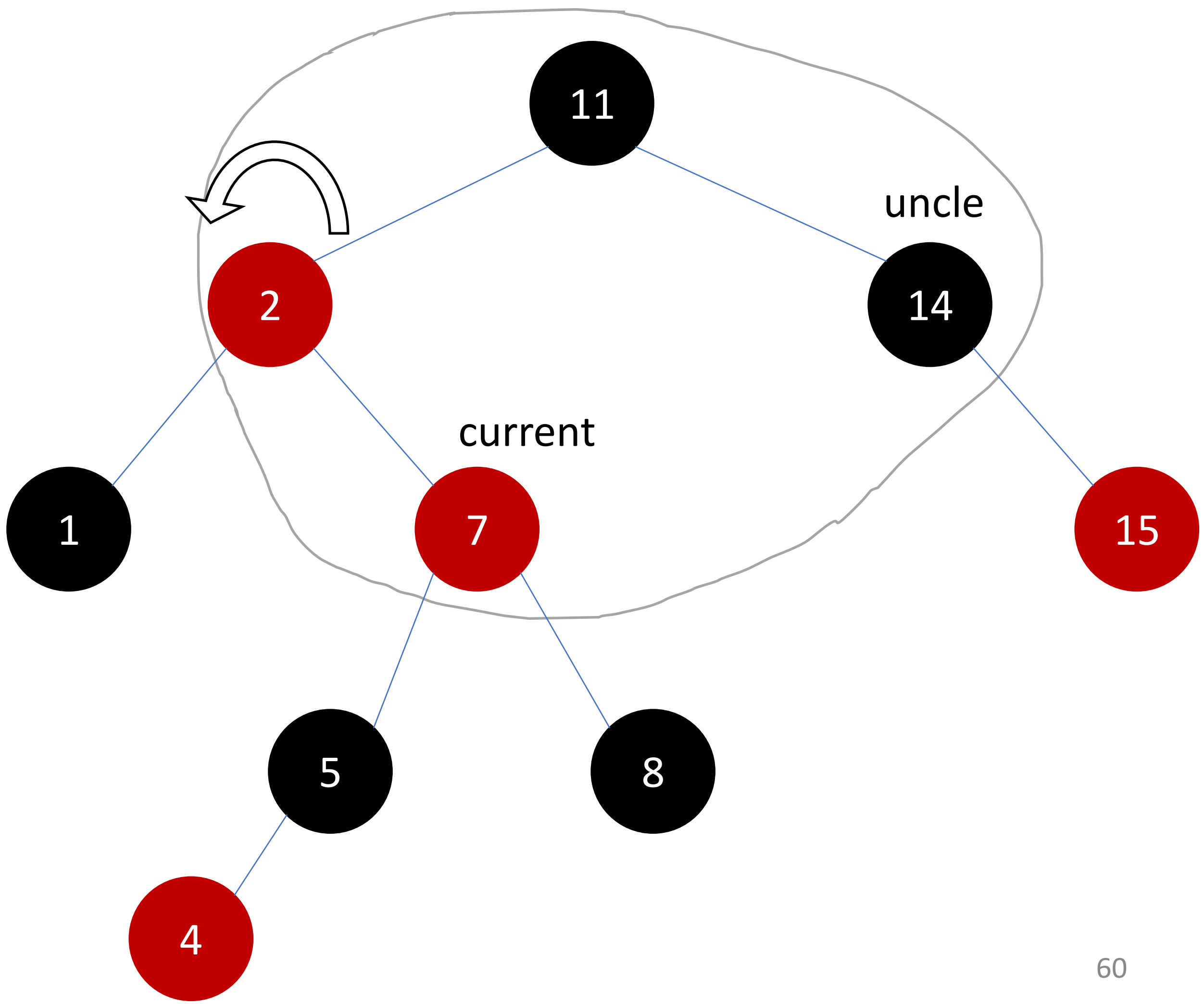
- Root & leaves are black
- **Red's children are black**
- BH(left) == BH(right)



# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

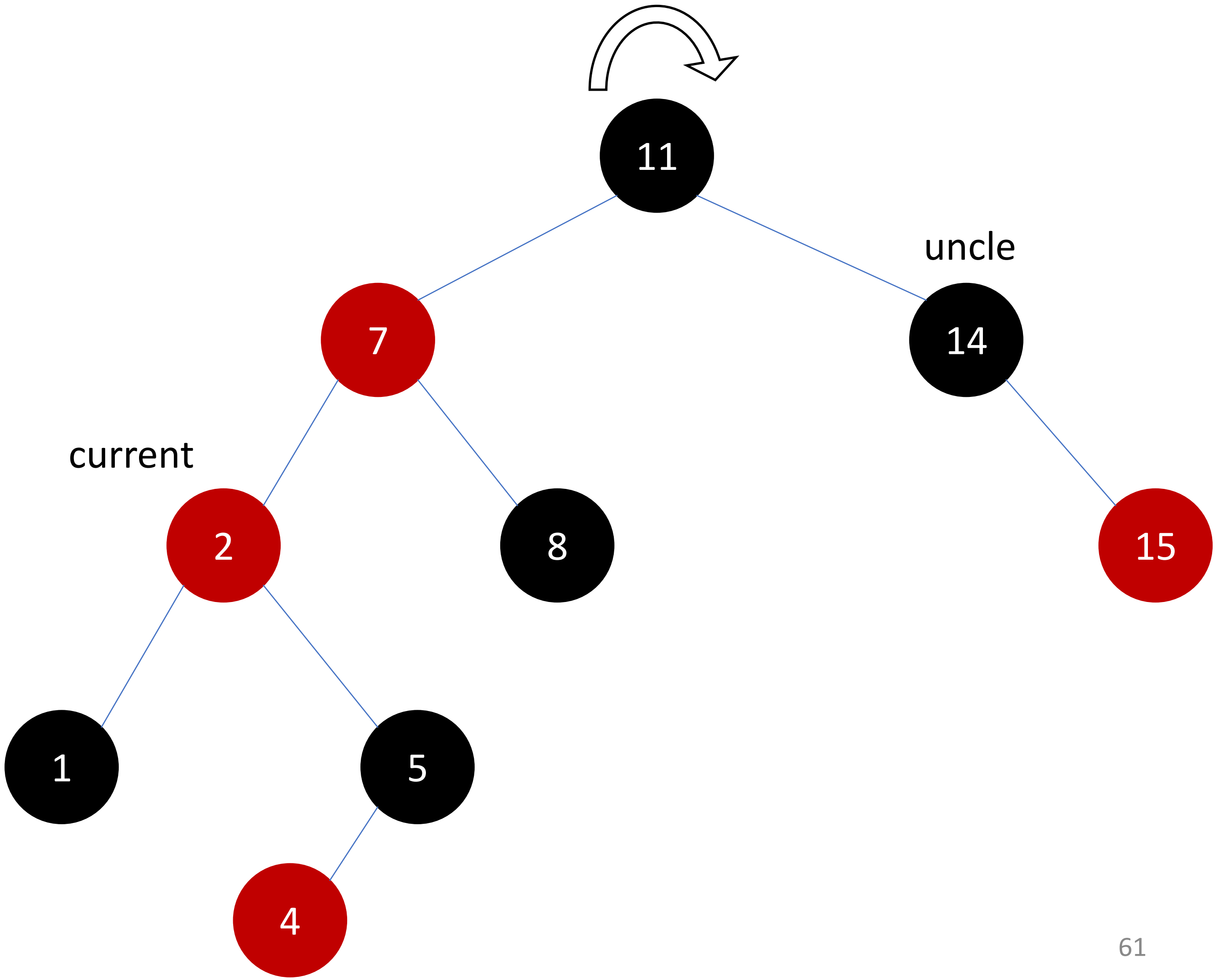
- Root & leaves are black
- **Red's children are black**
- BH(left) == BH(right)



# Red-black tree: insertion

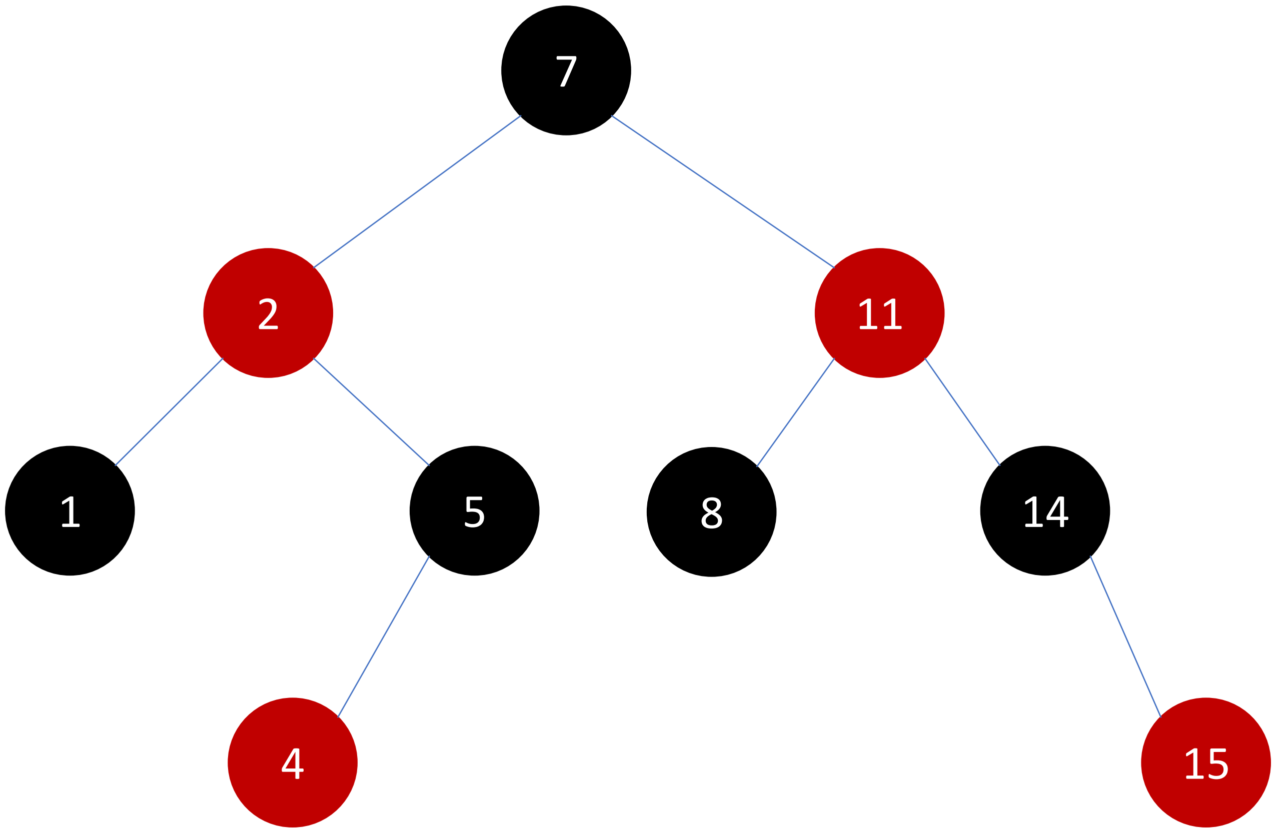
- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations

- Root & leaves are black
- **Red's children are black**
- BH(left) == BH(right)



# Red-black tree: insertion

- Insert new node as in BST
- Paint it red
- Restore RB properties with rotations




- No more than 2 adjustments

# Red-black tree: deletion

- Based on BST's deletion procedure: replace with next node in the tree & restore RB properties (see [1])

- Root is black
- Leaves are black
- Red's children are black
- BH(left) == BH(right)

# Red-black tree: Summary

- Guarantees log complexity despite relaxed height constraint
- Faster in practice* & requires much less memory (sometimes, no additional mem required)

*Ben Pfaff. Performance Analysis of BSTs in System Software https://benpfaff.org/papers/libavl.pdf

# Resources


- [1] Introduction to Algorithms, Thomas H. Cormen, chapters 12,13.
- [2] AVL tree visualization
- [3] Red-black tree visualization
- [4] Real systems balanced trees implementation comparison study by Ben Pfaff

# BACKUP