Session 06 - Tree Structure

Instructor:

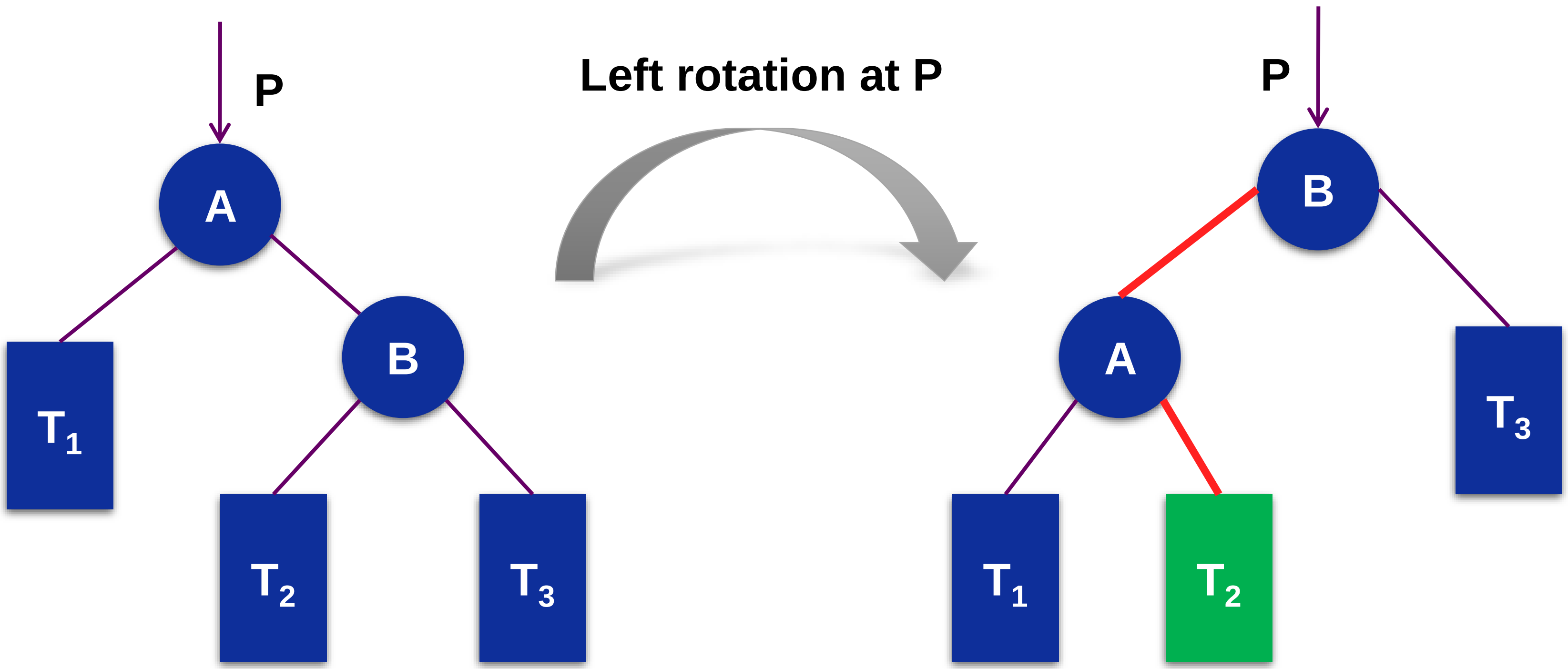
Dr. LE Thanh Tung

Content

1. Tree Rotation
2. AVL Tree
3. Red-Black Tree
4. 2-3, 2-3-4 Tree

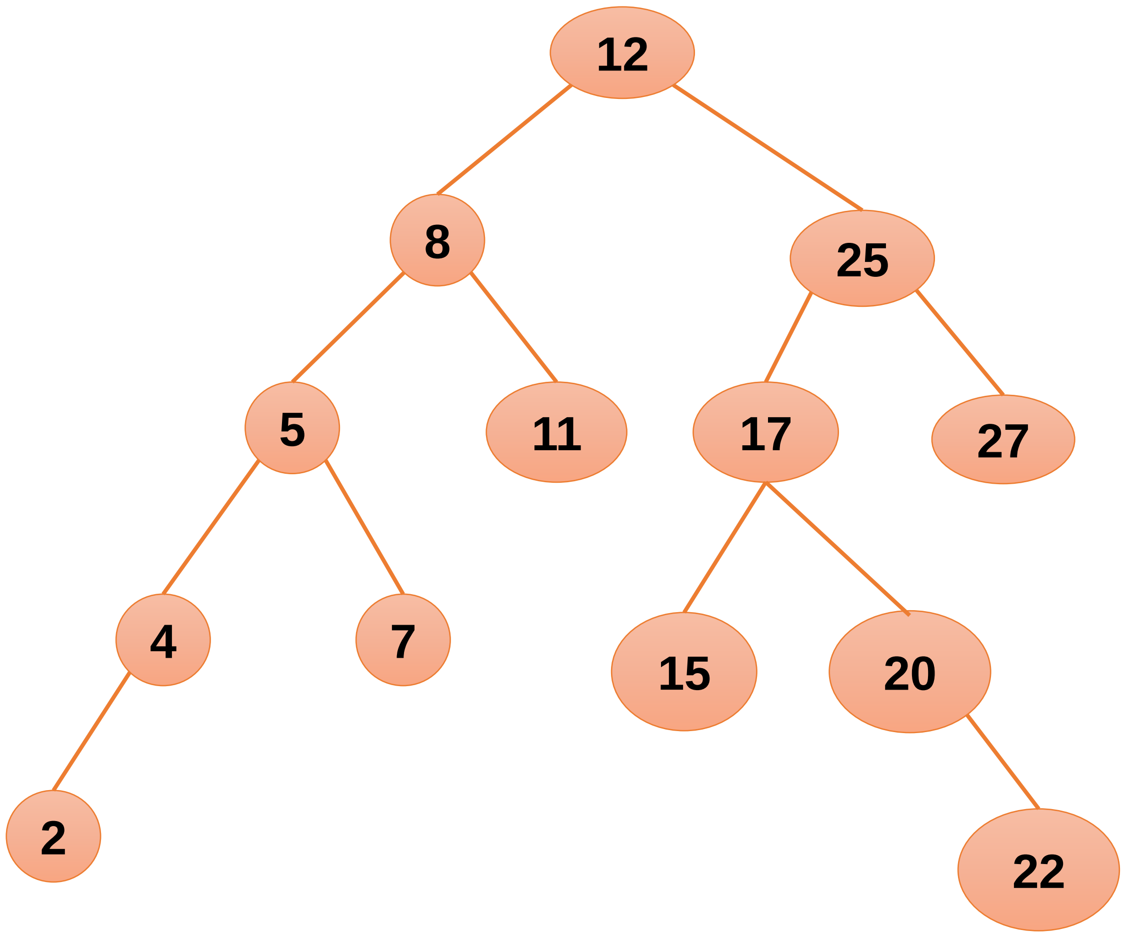
Tree Rotation

Left Rotation



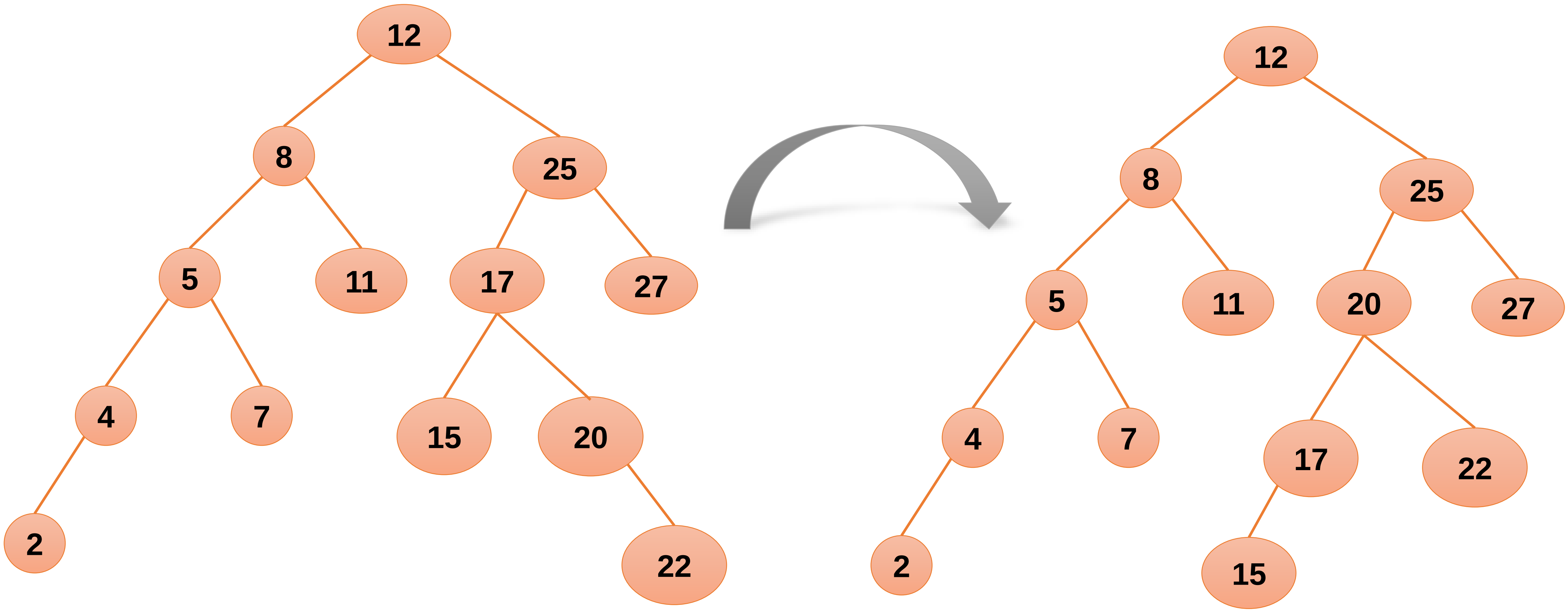
Left Rotation

- Left Rotate the following tree at Node 17

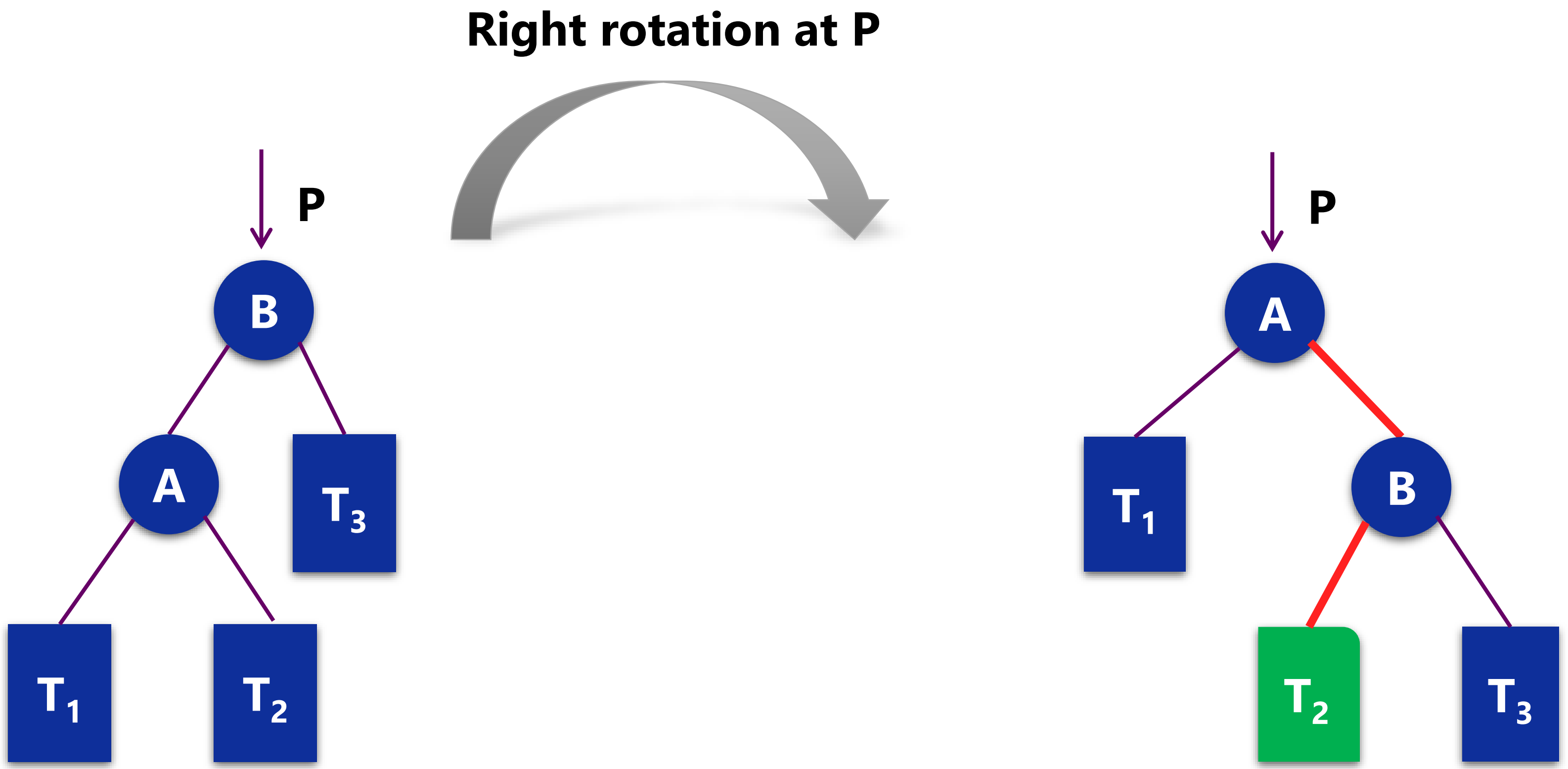


Left Rotation

- Left Rotate the following tree at Node 17

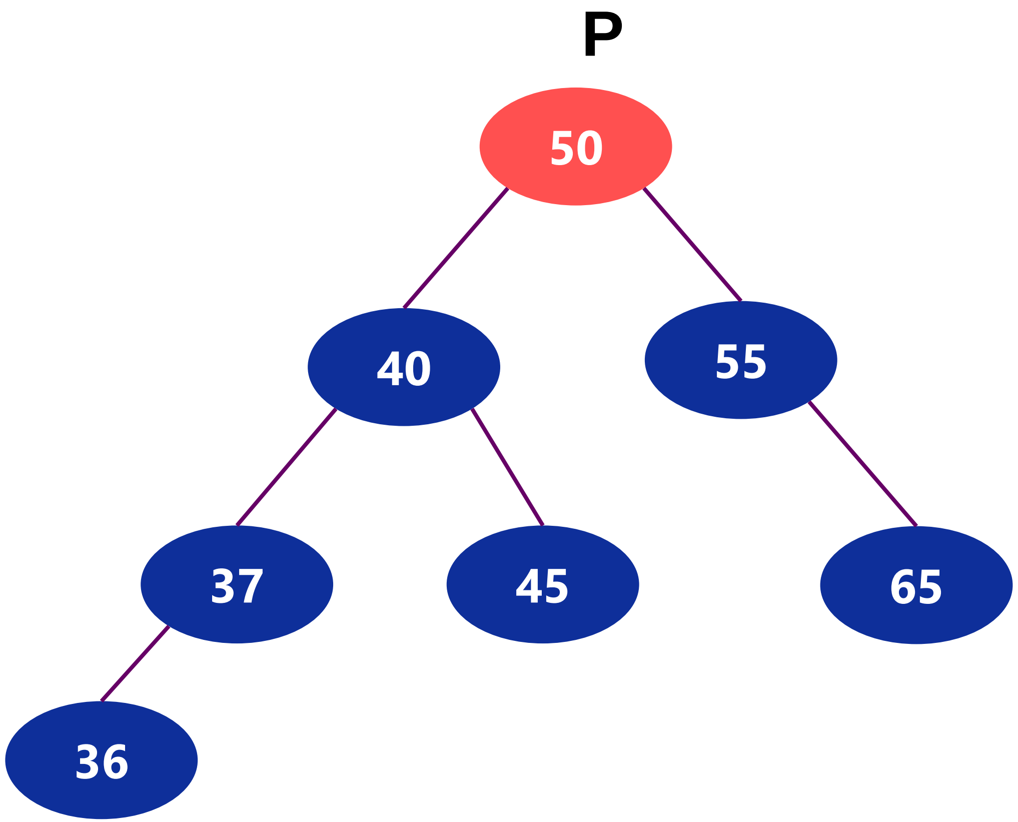


Right Rotation



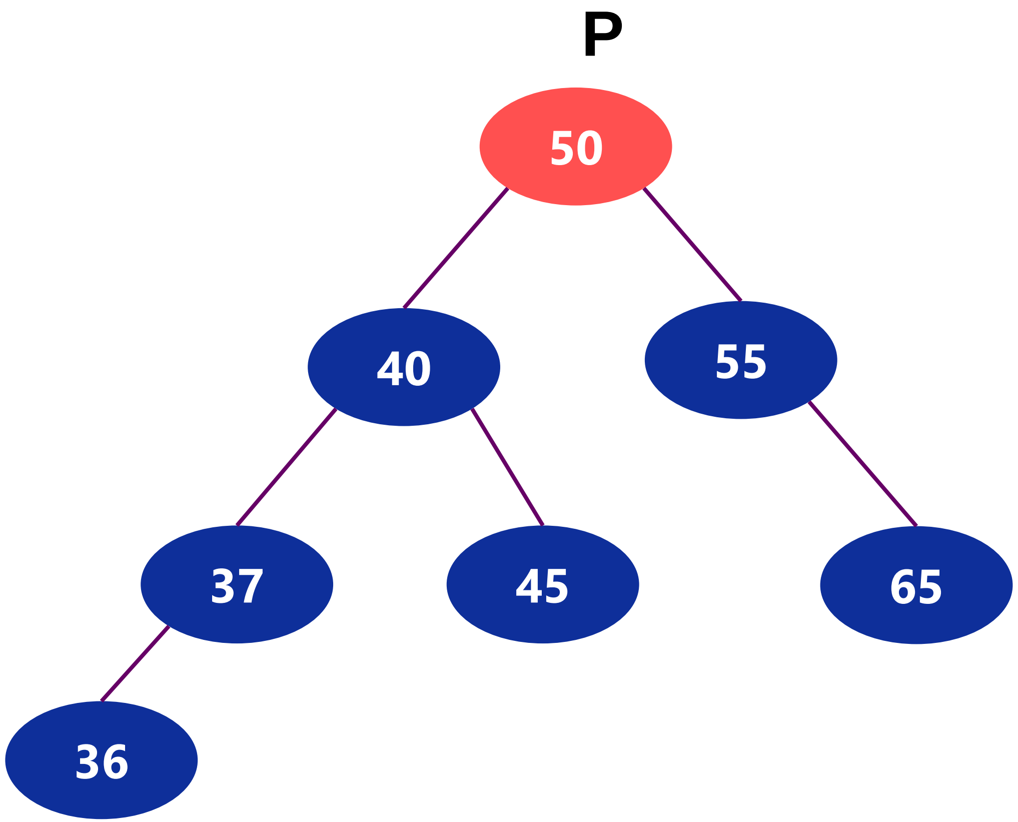
Right Rotation

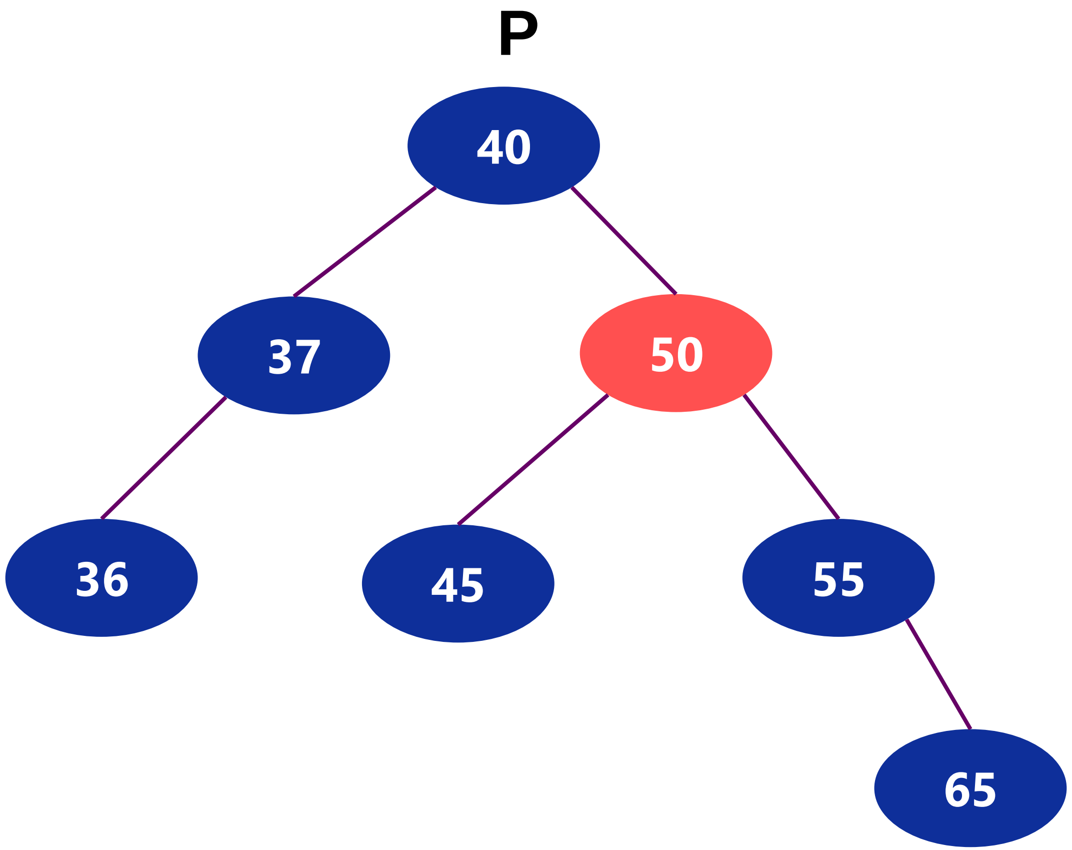
- Right Rotate the following tree at Node P



Right Rotation

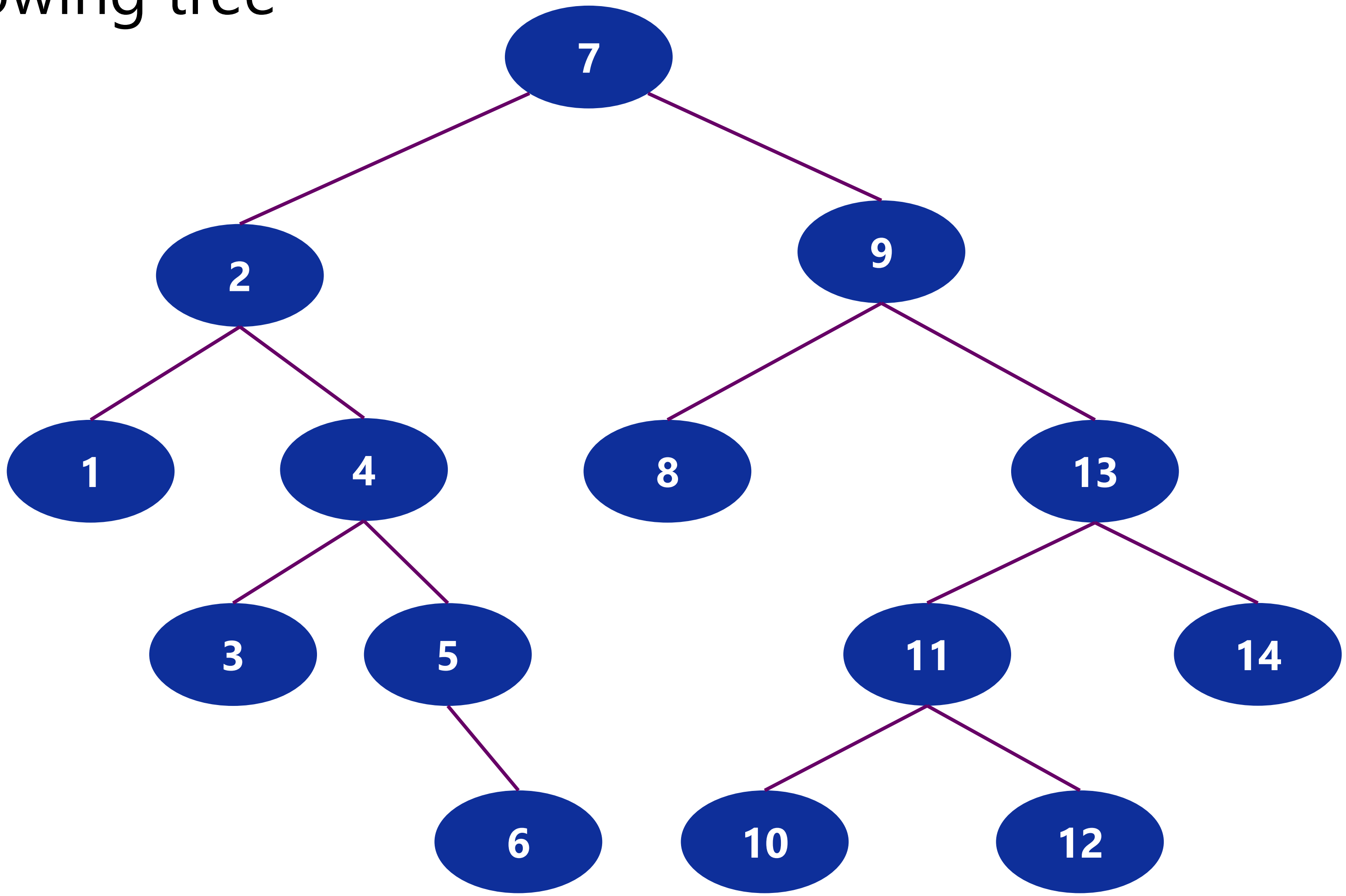
- Right Rotate the following tree at Node P





Exercise

- Give the following tree

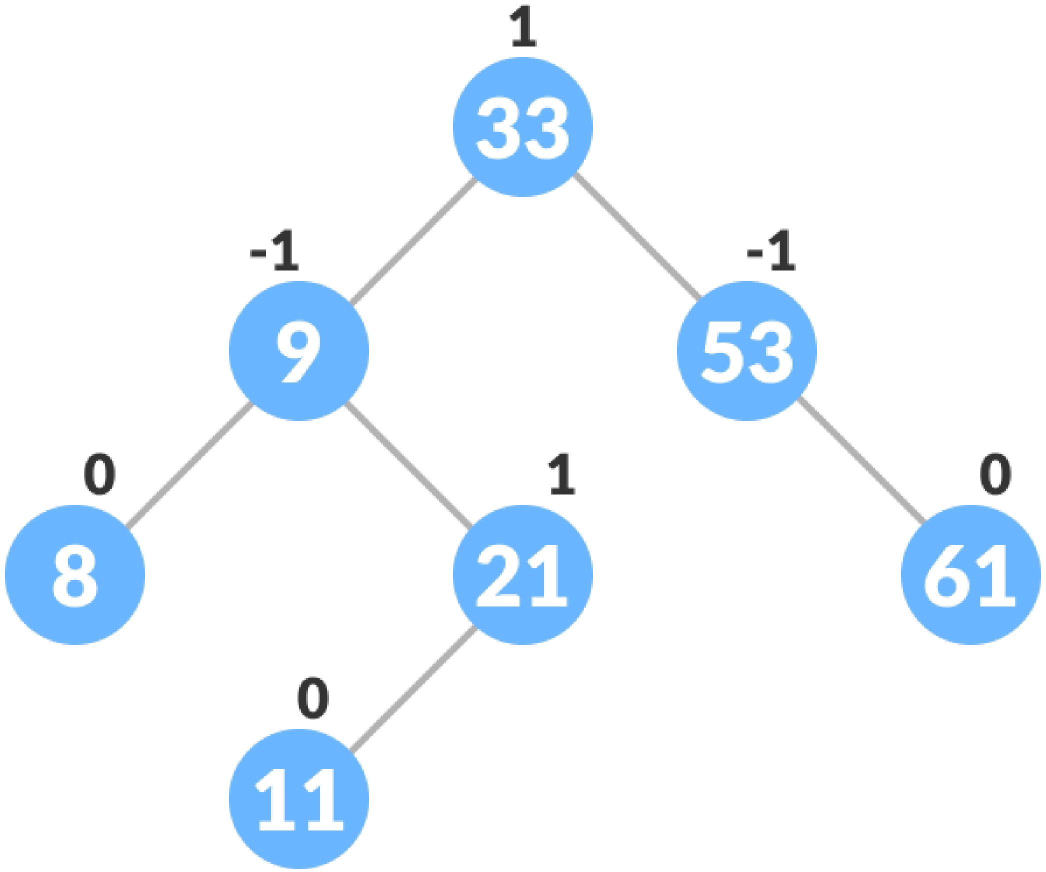


AVL Tree

- Named for inventors, (Georgii) Adelson-Velsky and (Evgenii) Landis
- Invented in 1962 (paper "*An algorithm for organization of information*").
- AVL Tree is a self-balancing binary search tree where
  - for ALL nodes, the difference between height of the left subtrees and the right subtrees **cannot be more than one**. (height invariant, or balance invariant).

- Balance Factor = (Height of Left Subtree - Height of Right Subtree)

  or = (Height of Right Subtree - Height of Left Subtree)



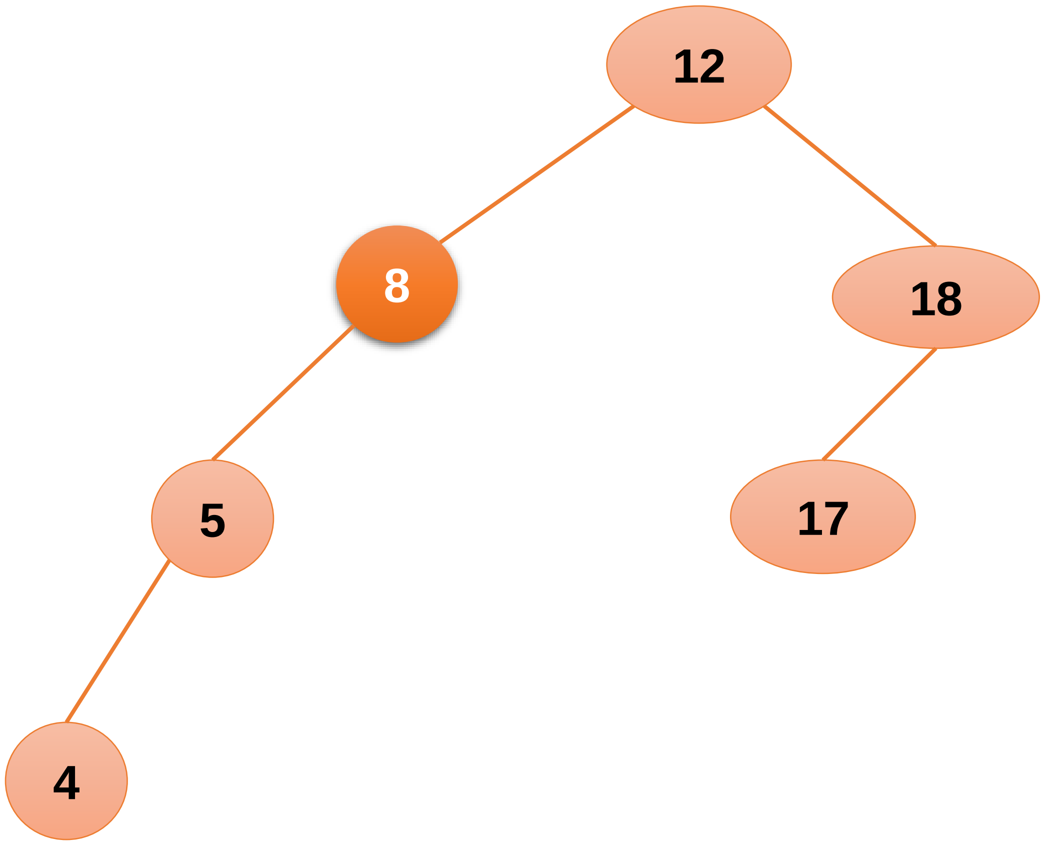
AVL Tree

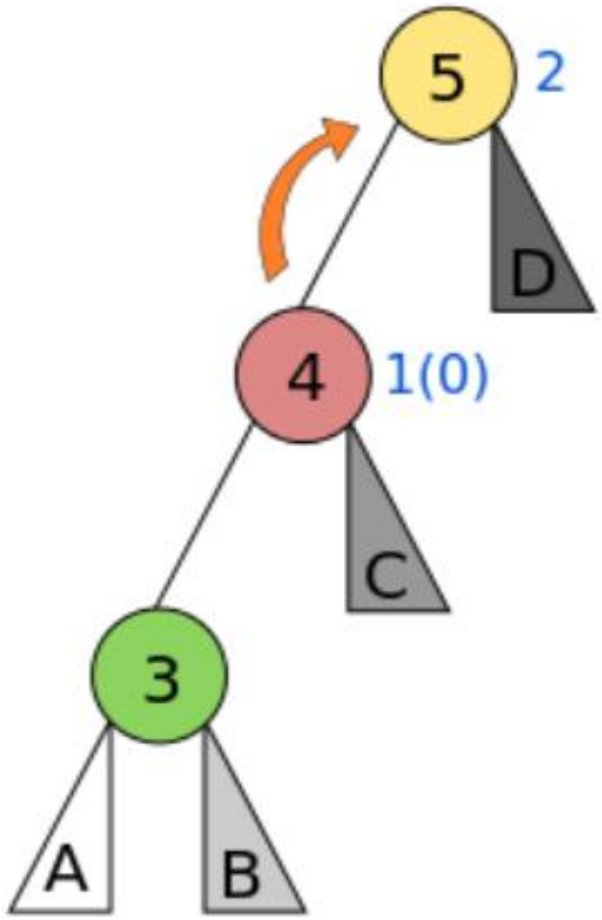
- A balanced binary search tree
  - Maintains height close to the minimum
  - After insertion or deletion, check the tree is still AVL tree – determine whether any node in tree has left and right subtrees whose heights differ by more than 1
- Can search AVL tree almost as efficiently as minimum-height binary search tree.

Cases of Height Invariant Violation

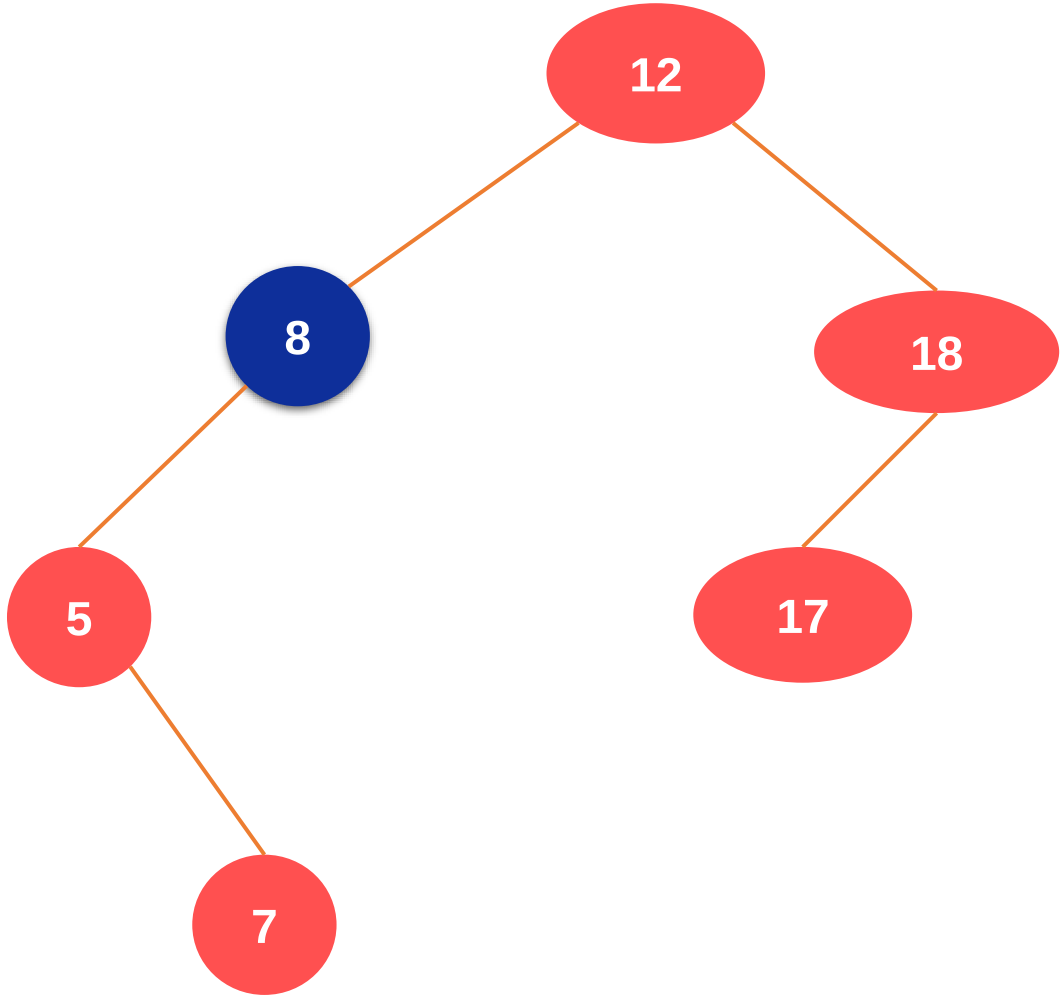
- Left-Left case
- Left-Right case
- Right-Right case
- Right-Left case

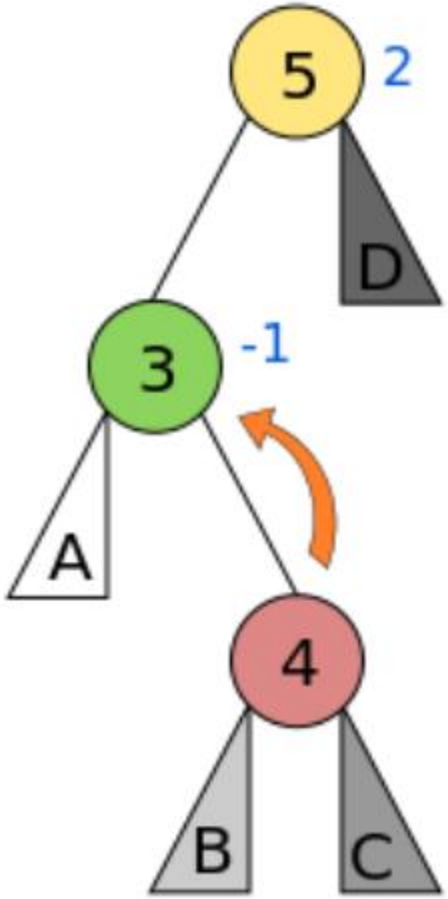
- Left-Left case





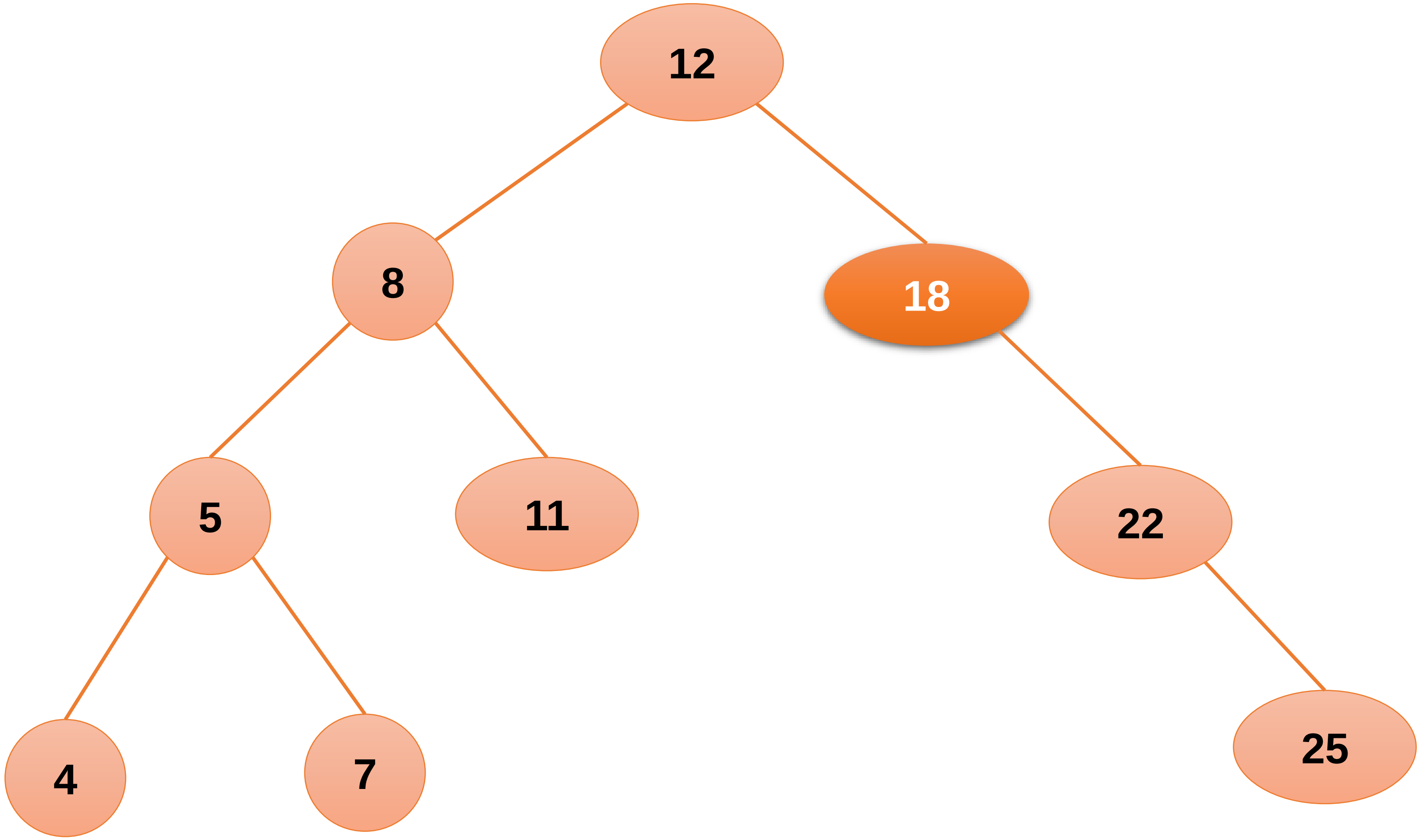
- Left-Right case

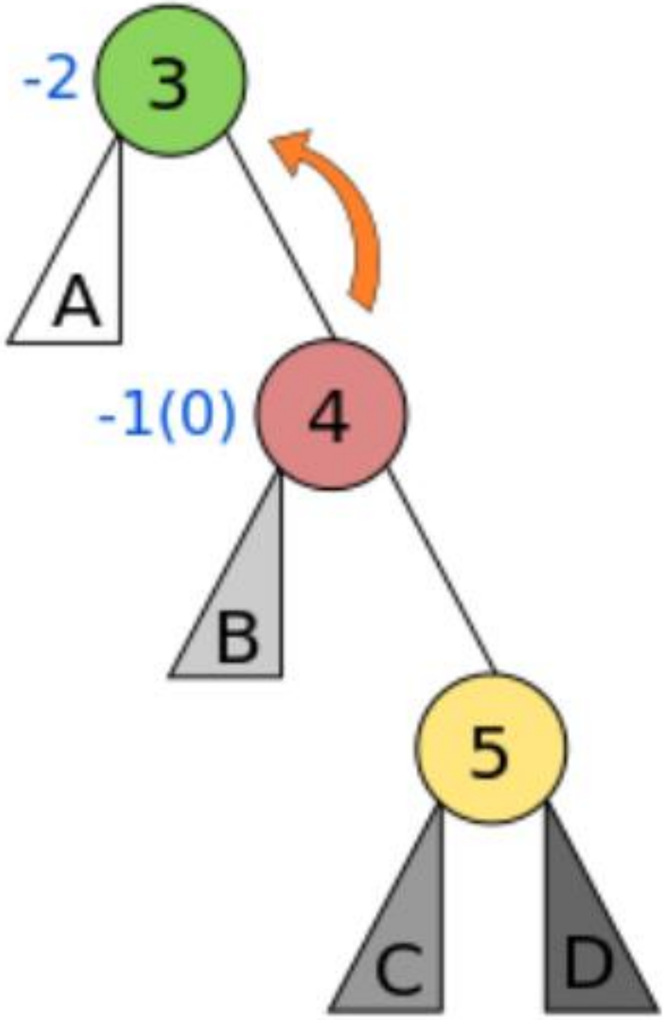




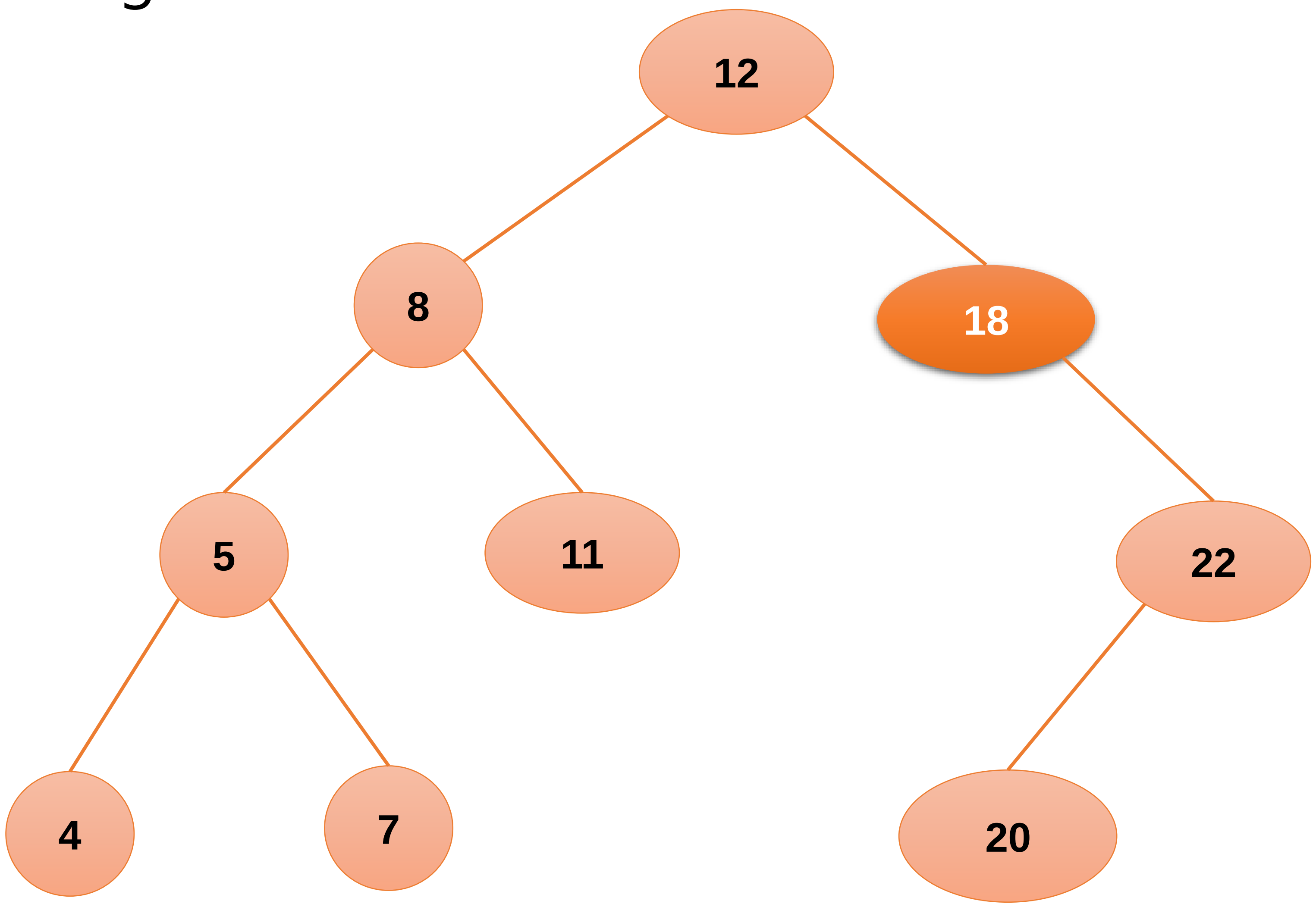
Cases of Height Invariant Violation

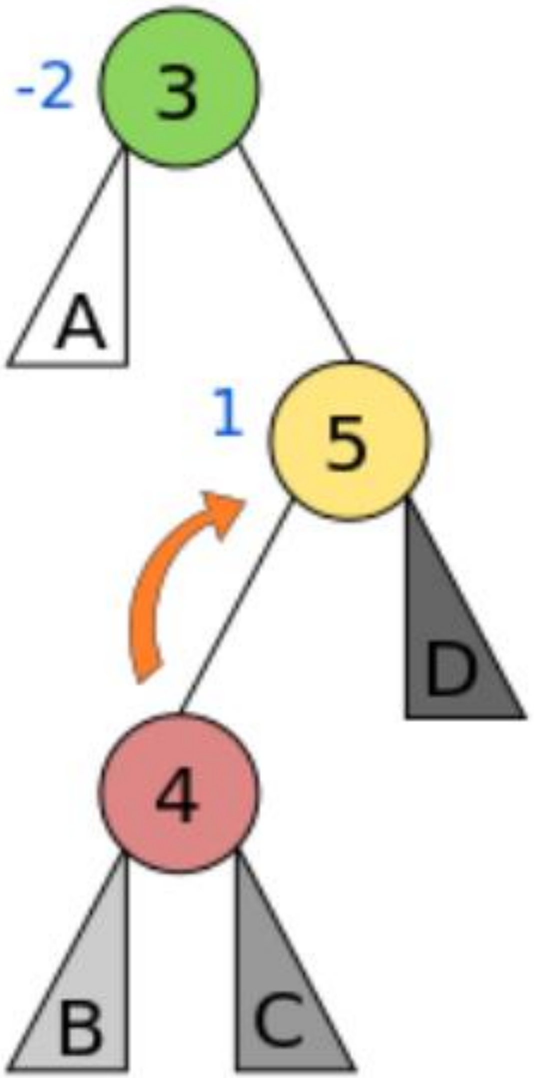
- Right-Right case





- Right-Left case





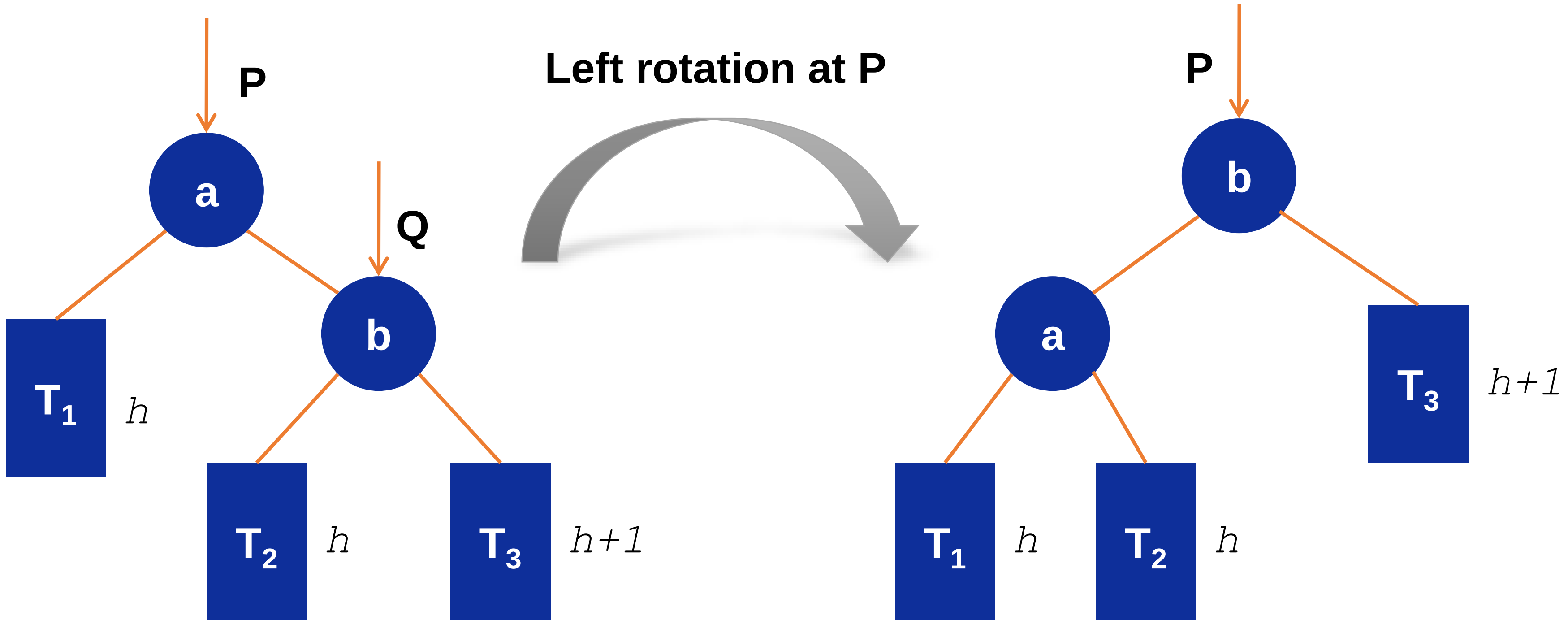
Violation Resolving

- **Right-Right case:**
  - Left rotation at un-balanced node.

- **Right-Left case:**
  - Right rotation at un-balanced node's right child
  - Left rotation at un-balanced node.

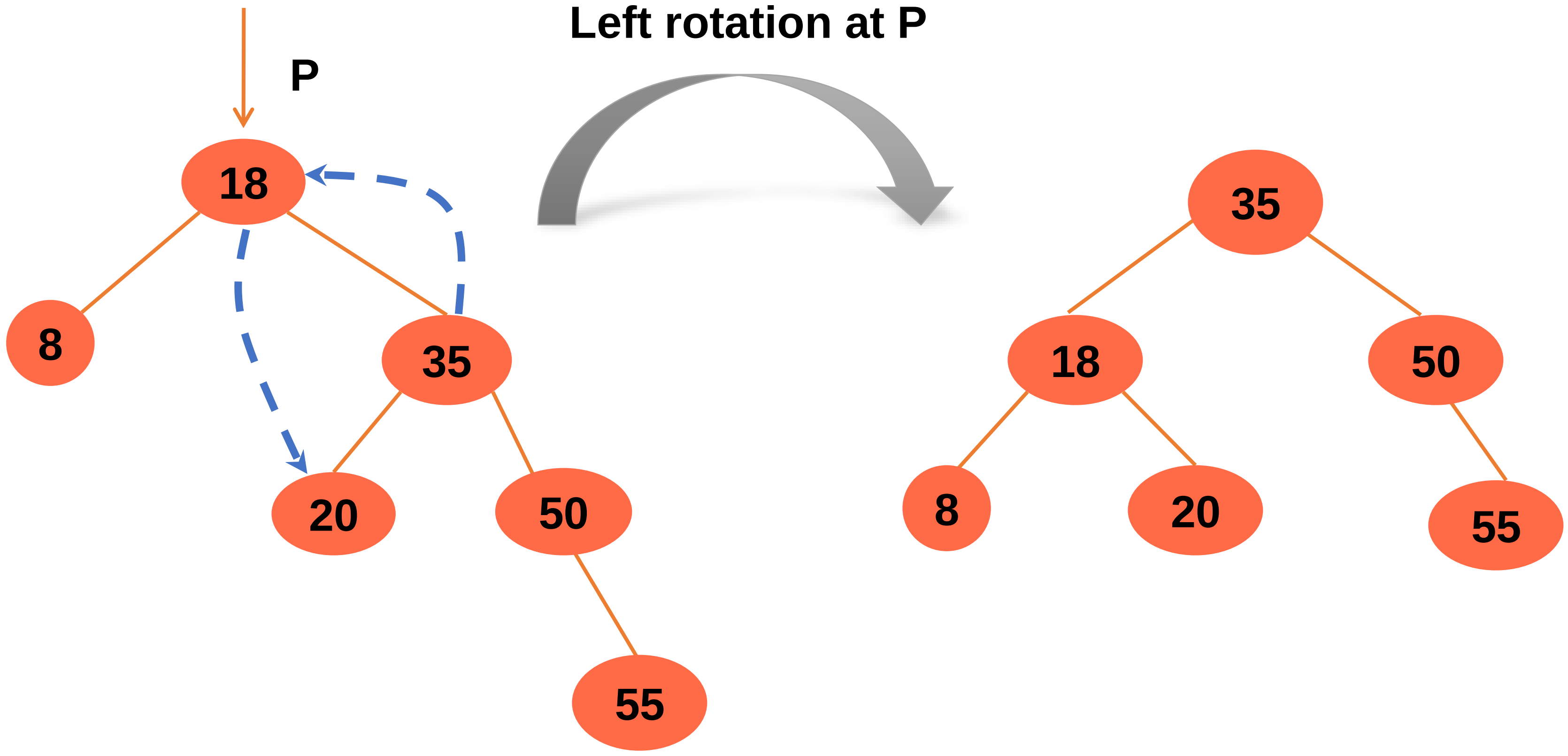
Violation Resolving

- **Right-Right case:**
    - Unbalance at P



Violation Resolving

- **Right-Right case:** example

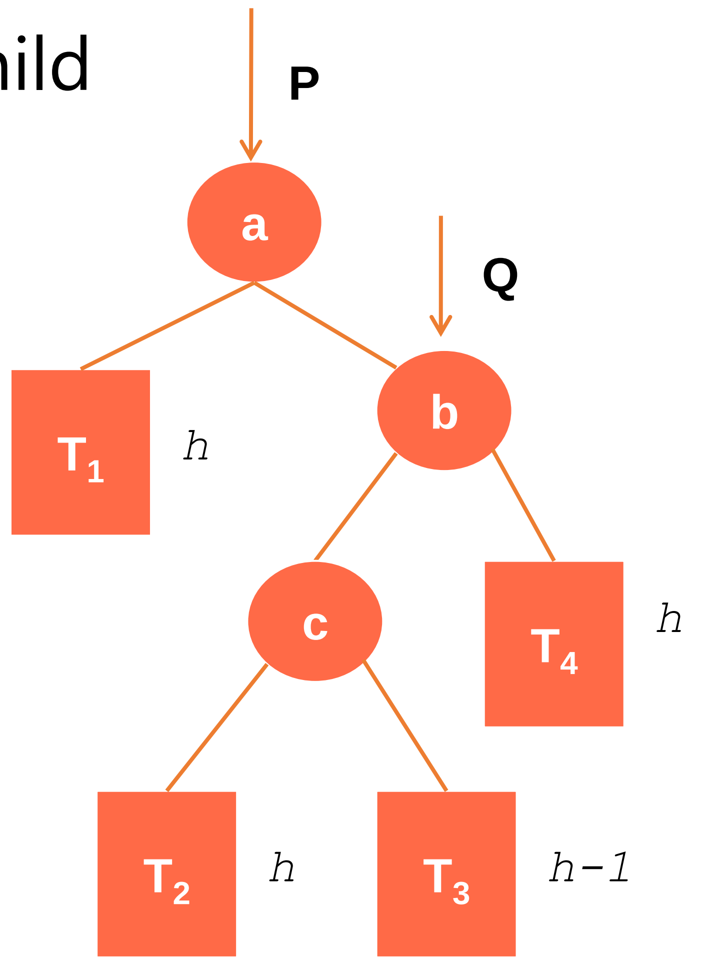


Violation Resolving

- **Right-Left case:**
  - Right rotation at un-balanced node's right child
  - Left rotation at un-balanced node

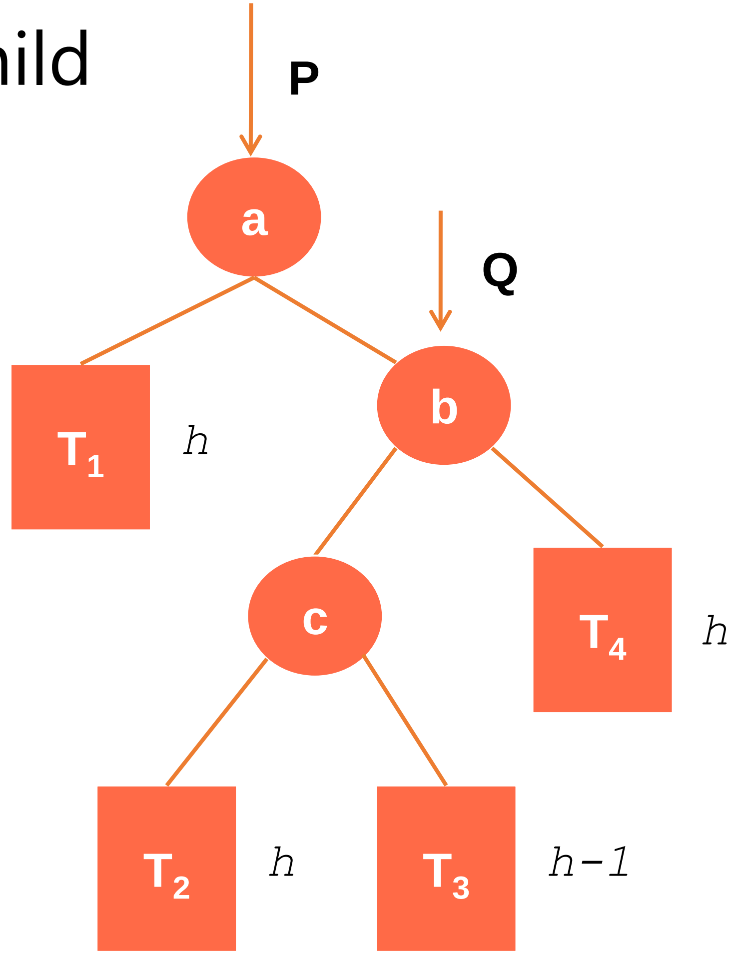
- In the following tree,

what is the unbalanced node ?



Violation Resolving

- **Right-Left case:**
  - Right rotation at un-balanced node's right child
  - Left rotation at un-balanced node



- In the following tree,

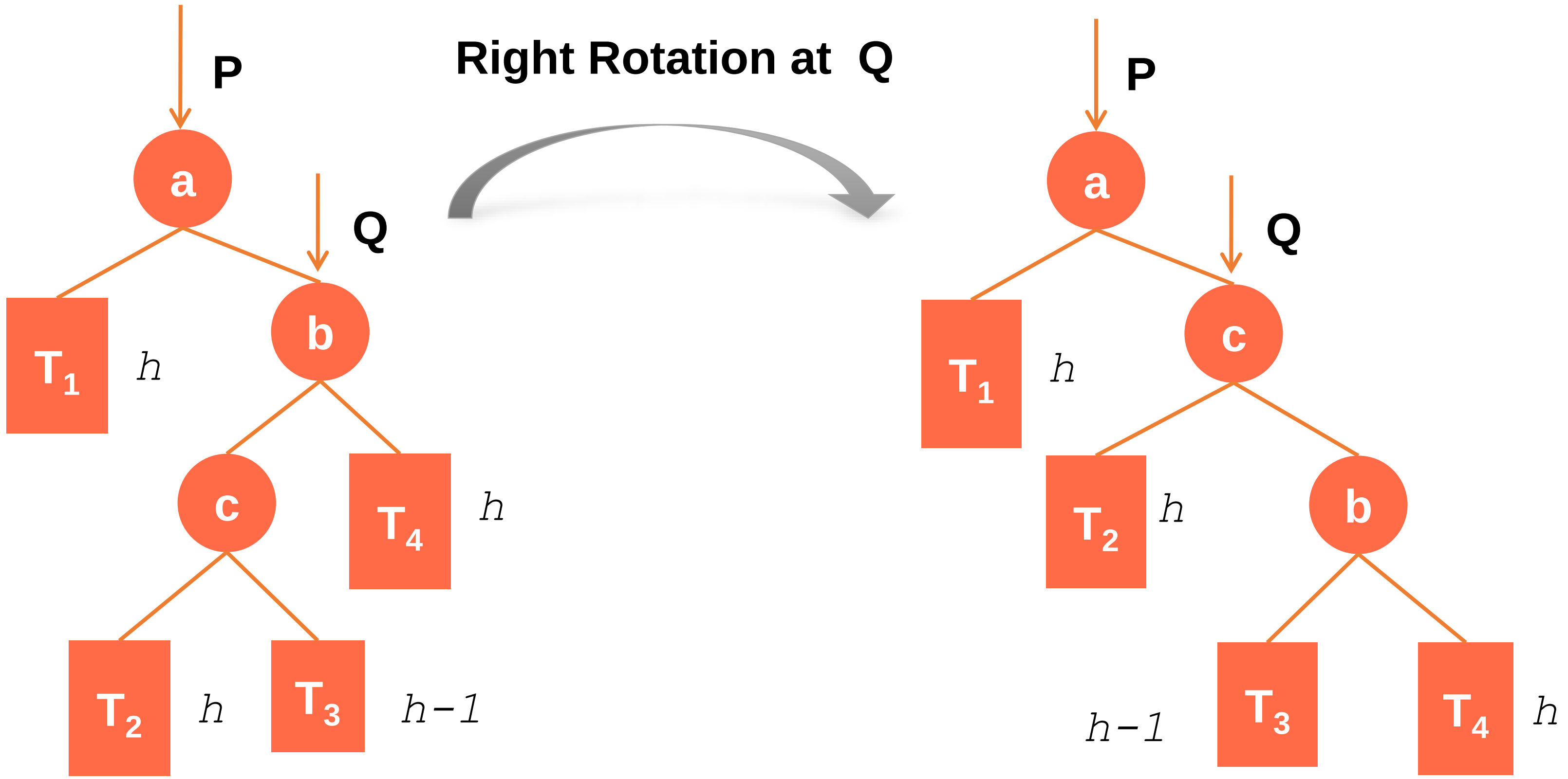
what is the unbalanced node ?

Resolve:

  - Right rotation at Q
  - Left rotation at P

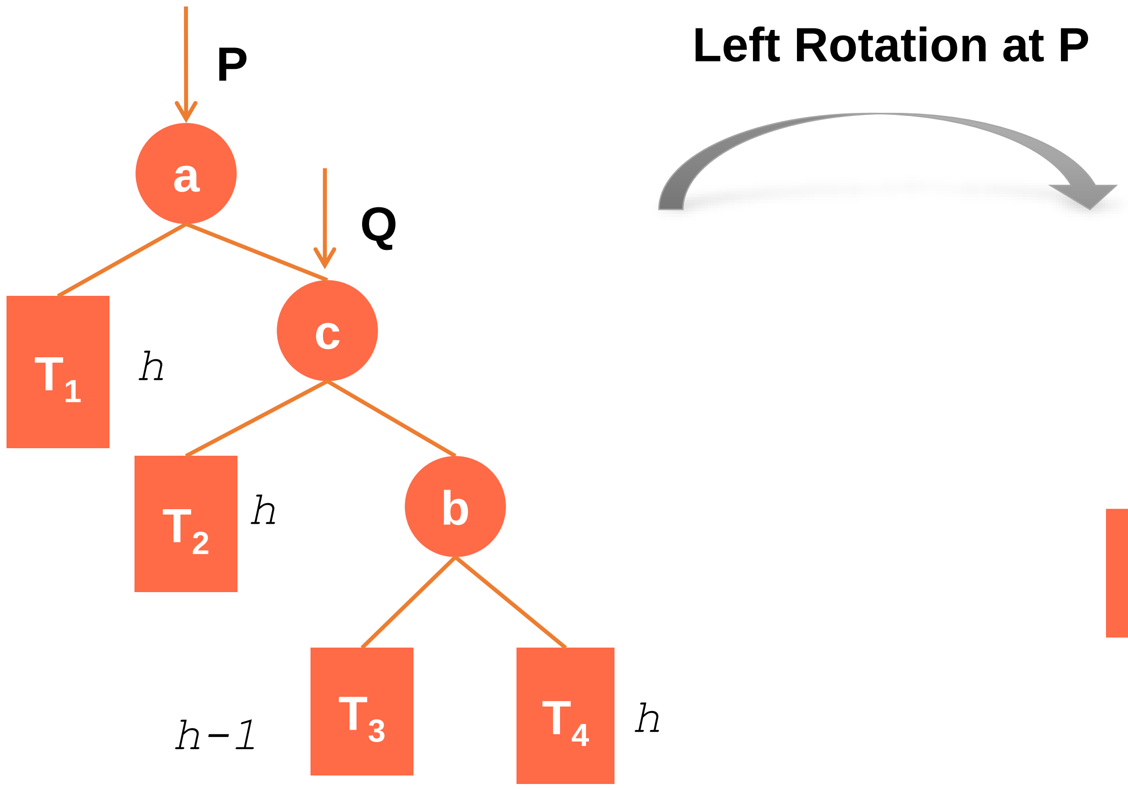
Violation Resolving

- **Right-Left case:**
  - Right rotation at Q

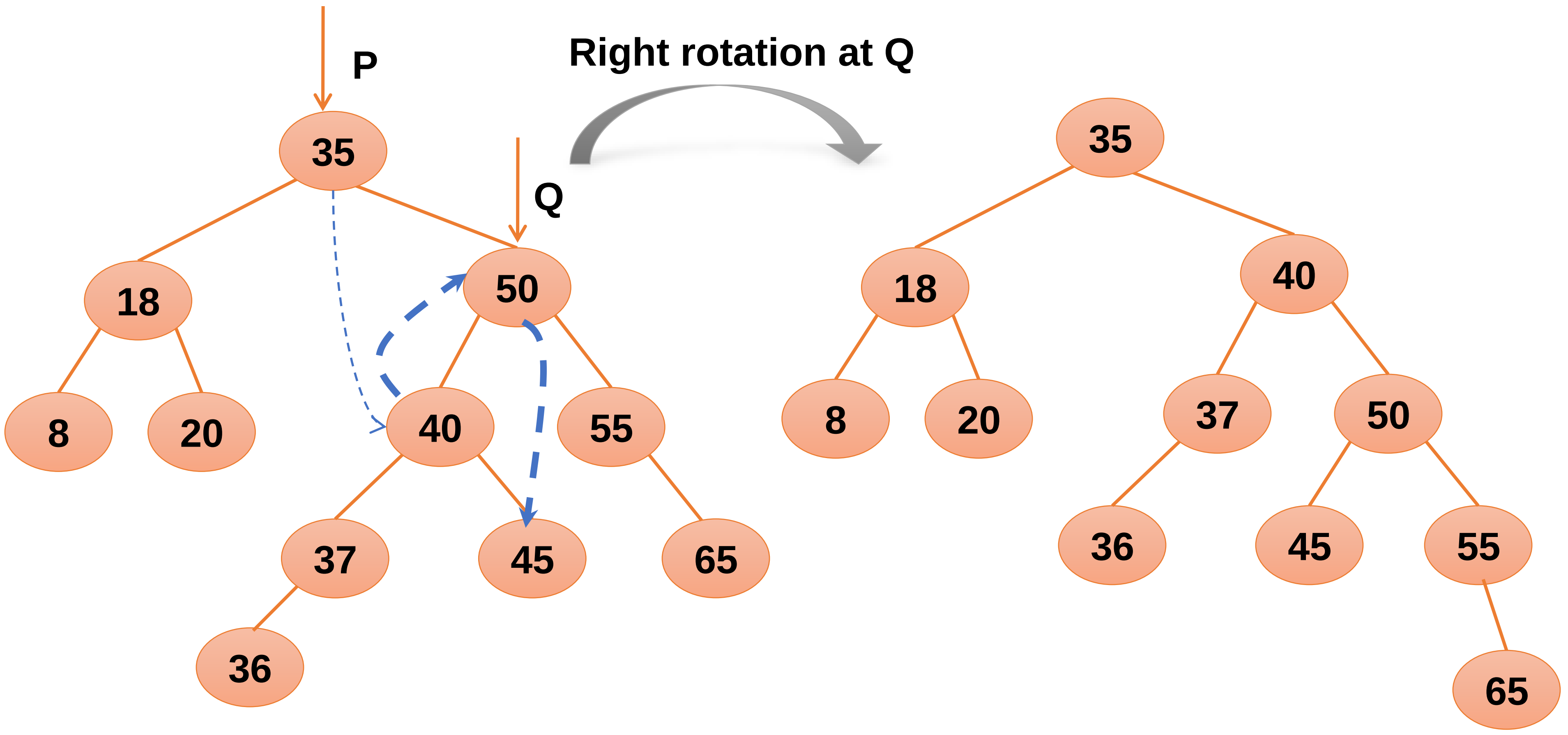


Violation Resolving

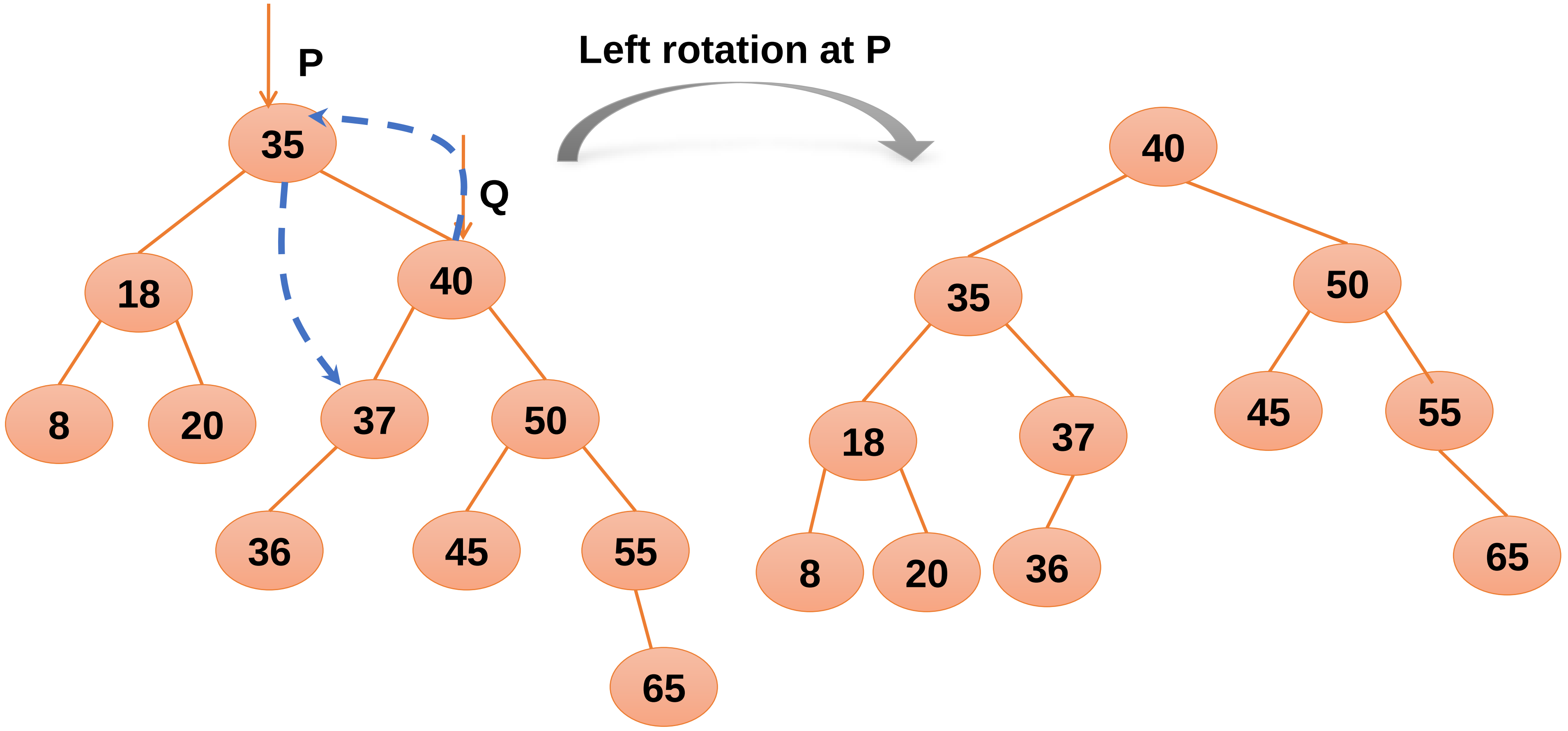
- **Right-Left case:**
  - Left rotation at P



- **Right-Left case: example**



- **Right-Left case: example**

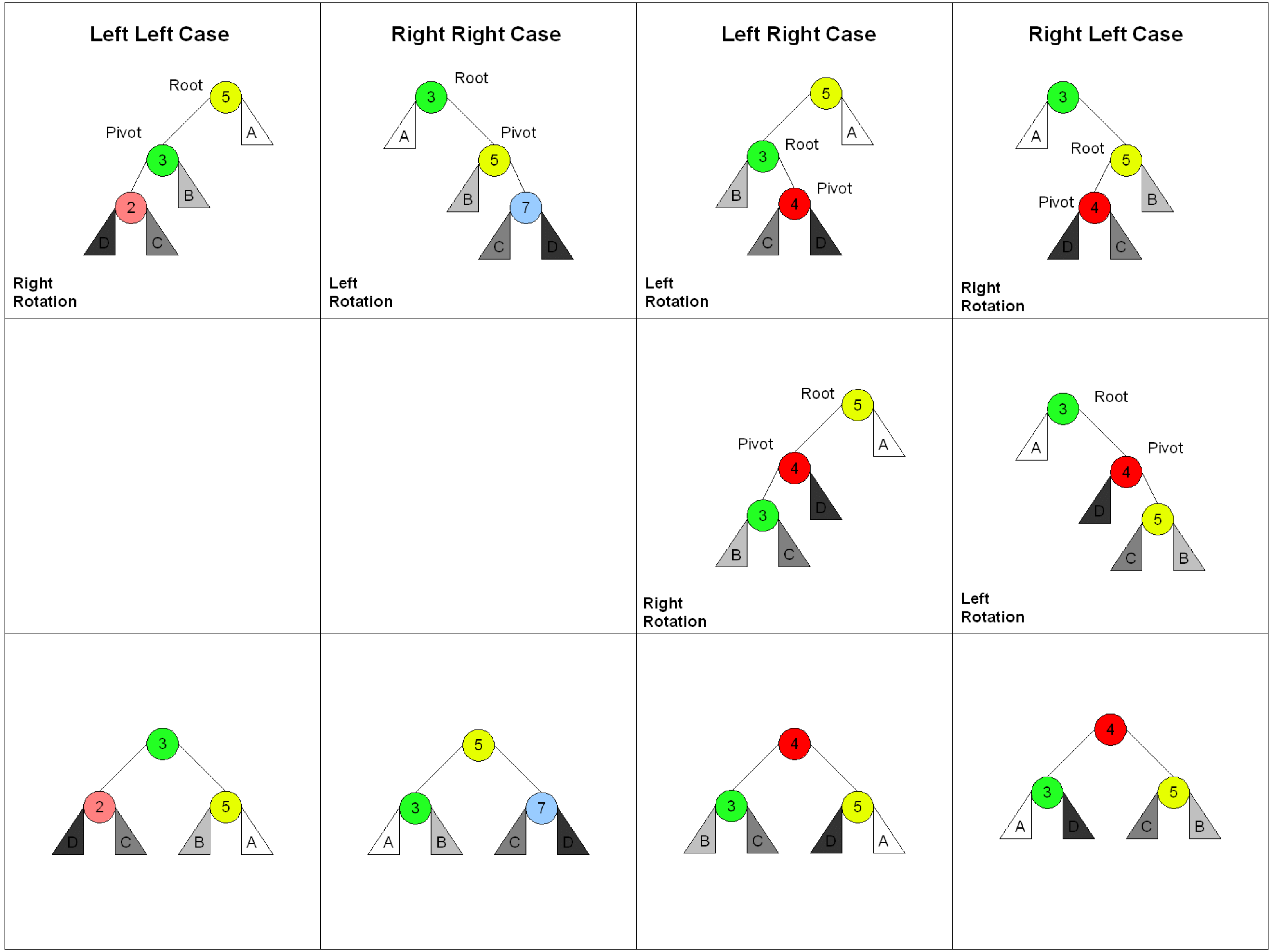


- **Left-Left case:**
  - Right rotation at un-balanced node.
- **Left-Right case:**
  - Left rotation at un-balanced node's left child
  - Right rotation at un-balanced node.

Violation Resolving

There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.



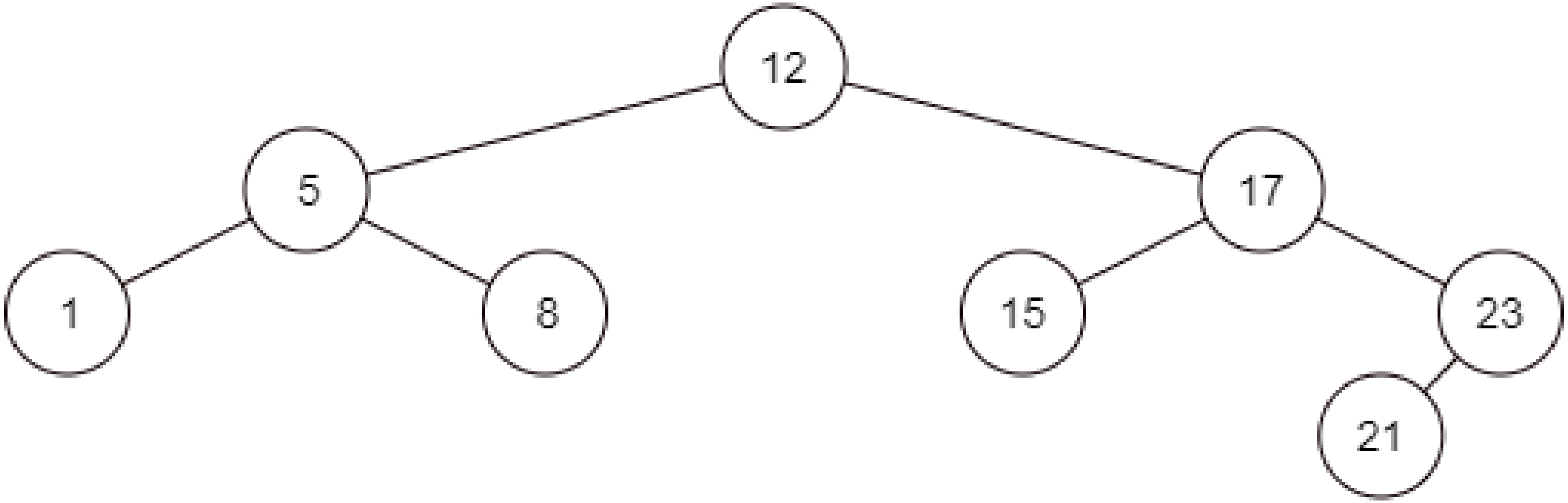
Insertion

- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

15, 5, 12, 8, 23, 1, 17, 21

- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

15, 5, 12, 8, 23, 1, 17, 21



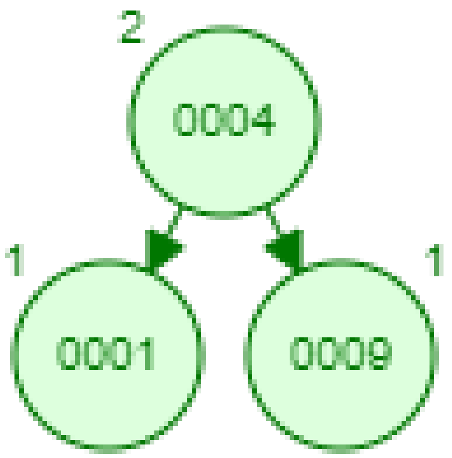
Insertion

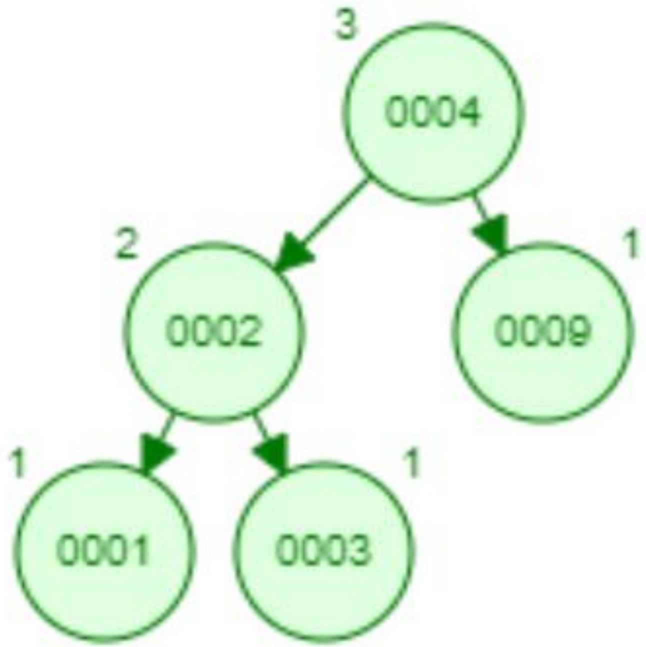
- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

9, 1, 4, 2, 3, 9, 5, 8, 6, 7, 4

- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

9, 1, 4, 2, 3, 9, 5, 8, 6, 7, 4

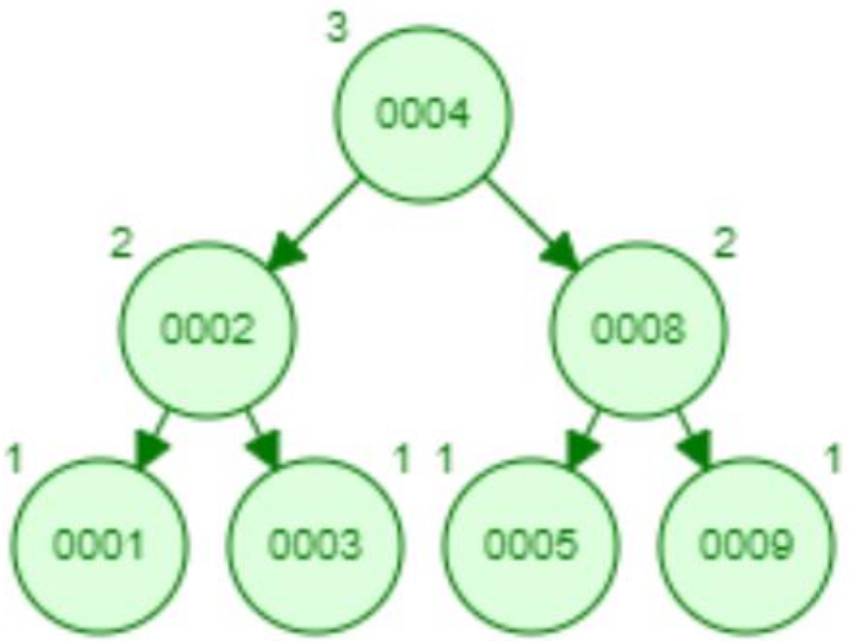


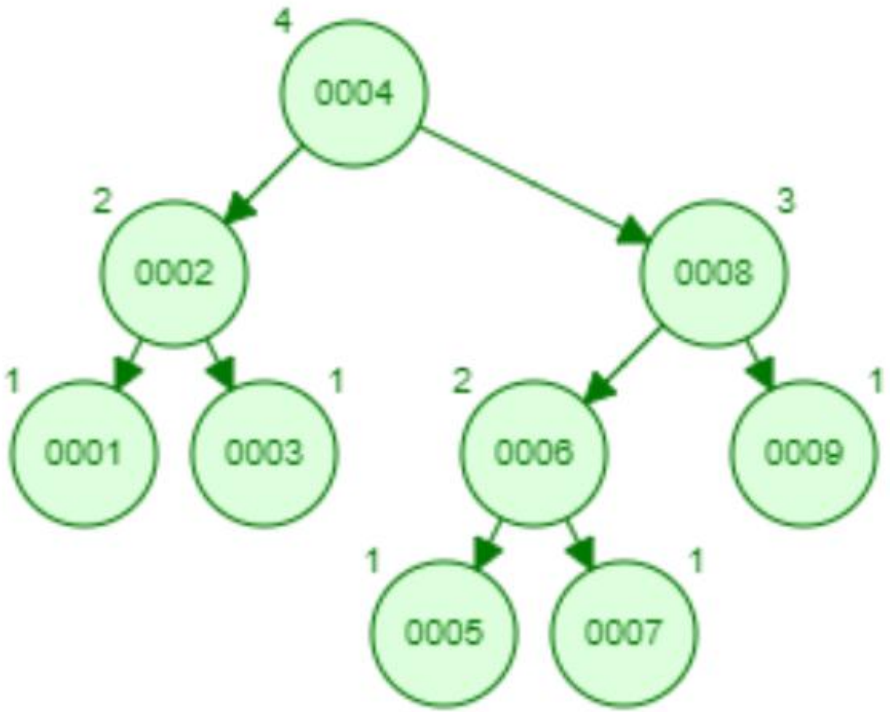


Insertion

- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

9, 1, 4, 2, 3, 9, 5, 8, 6, 7, 4

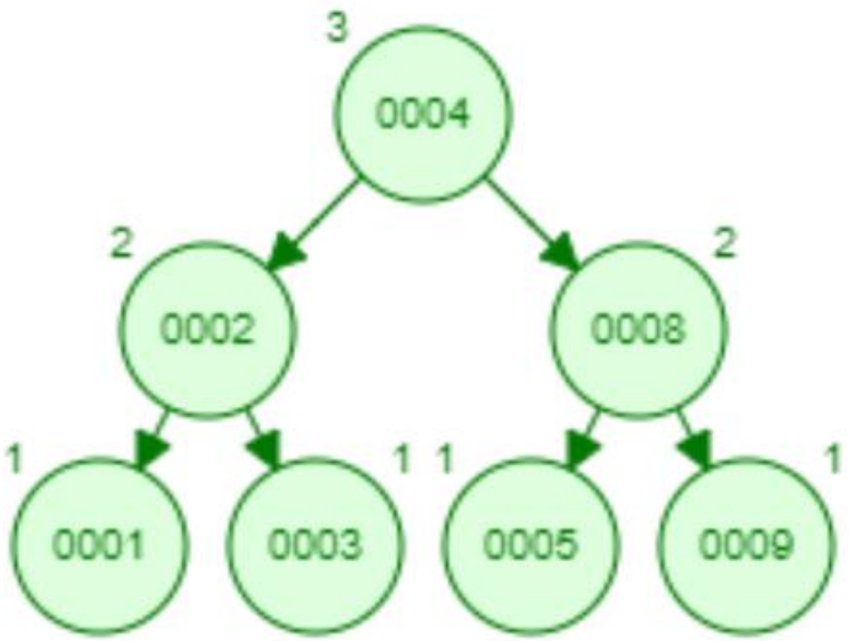


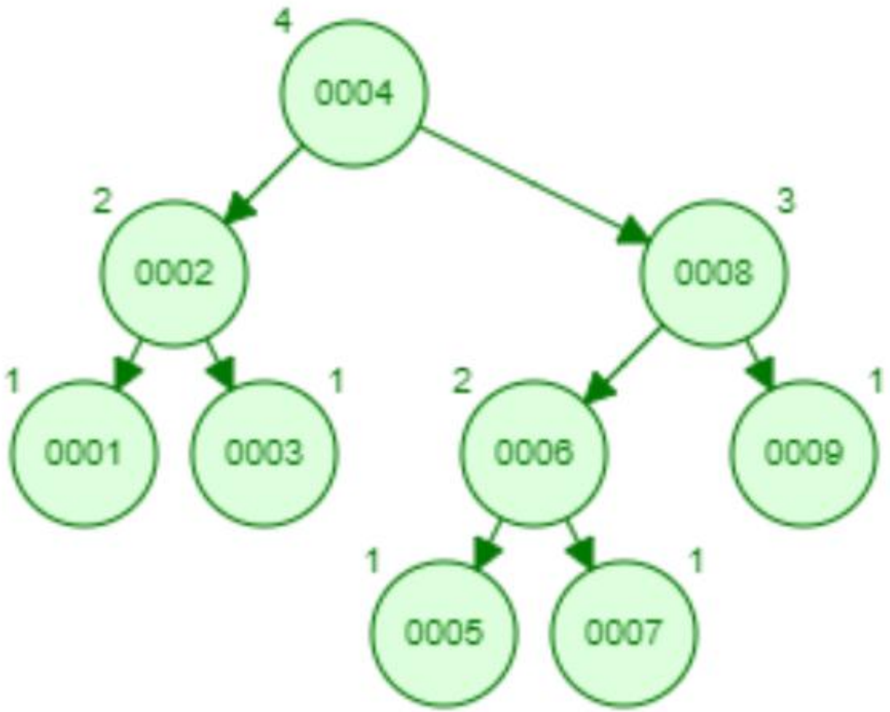


Insertion

- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

9, 1, 4, 2, 3, 9, 5, 8, 6, 7, 4

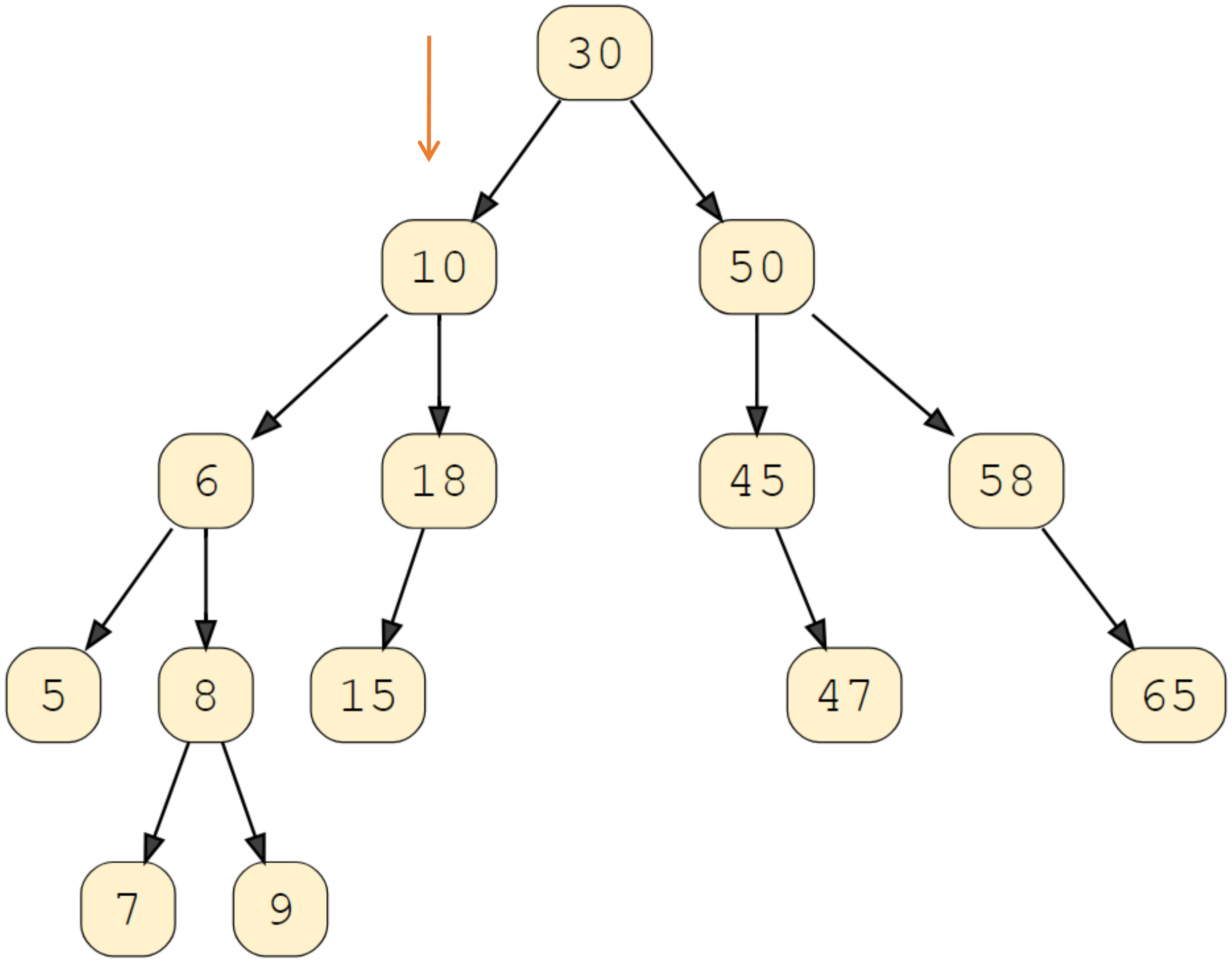




Deletion

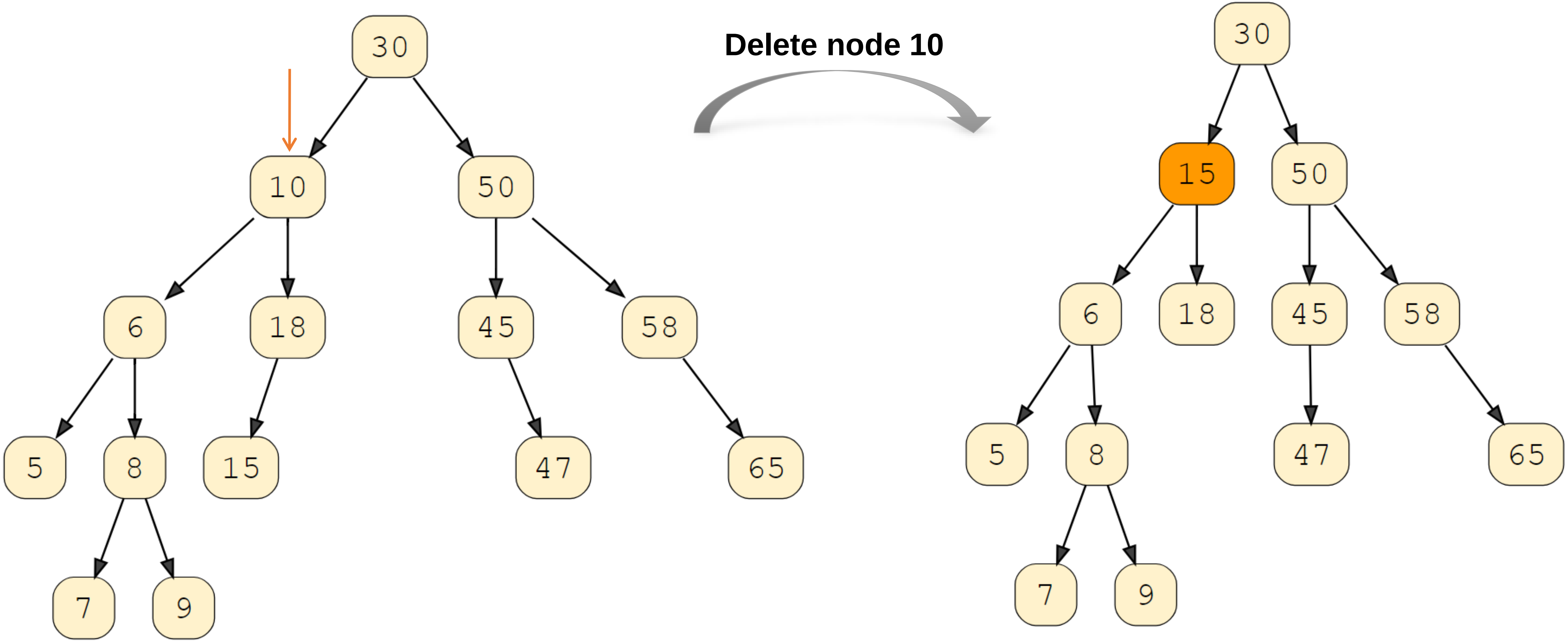
- Deleting a node from an AVL tree is similar to that in a binary search tree
  - Delete the node (in 3 case of BST)
  - Rebalance the tree once the node is deleted

- Delete Node 10:

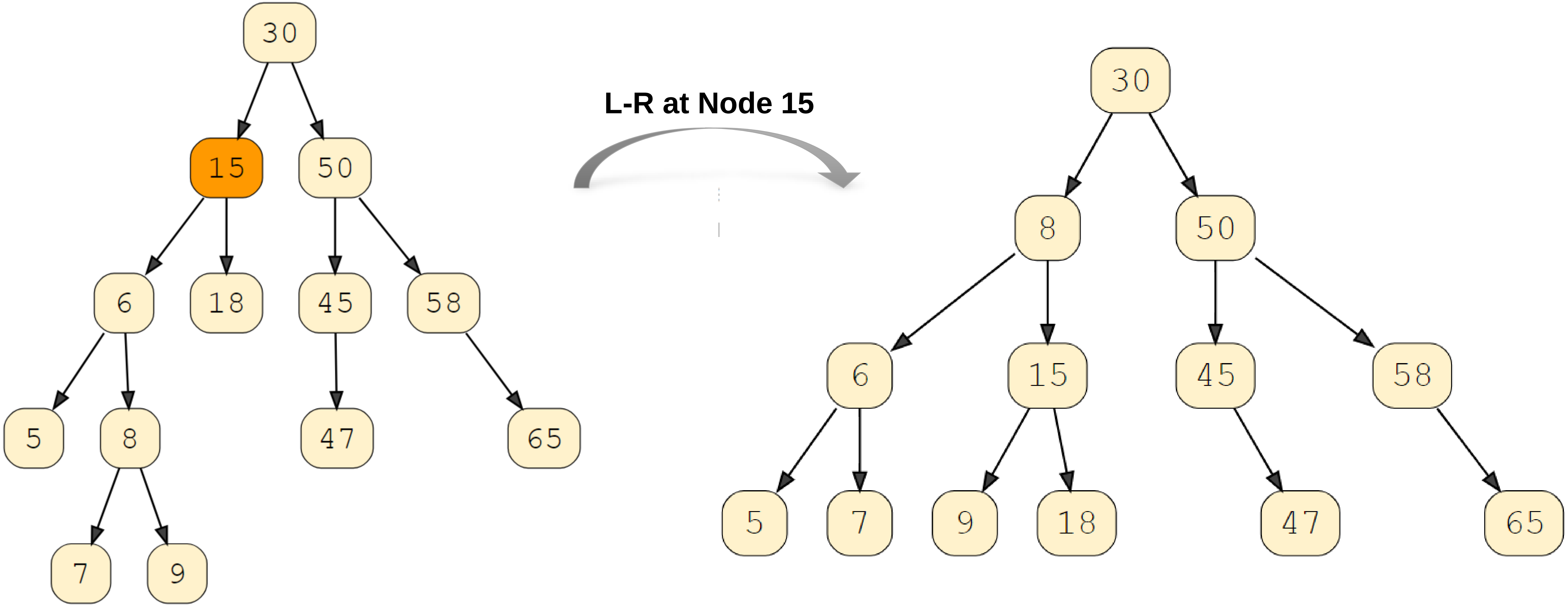


Deletion

- Delete Node 10:



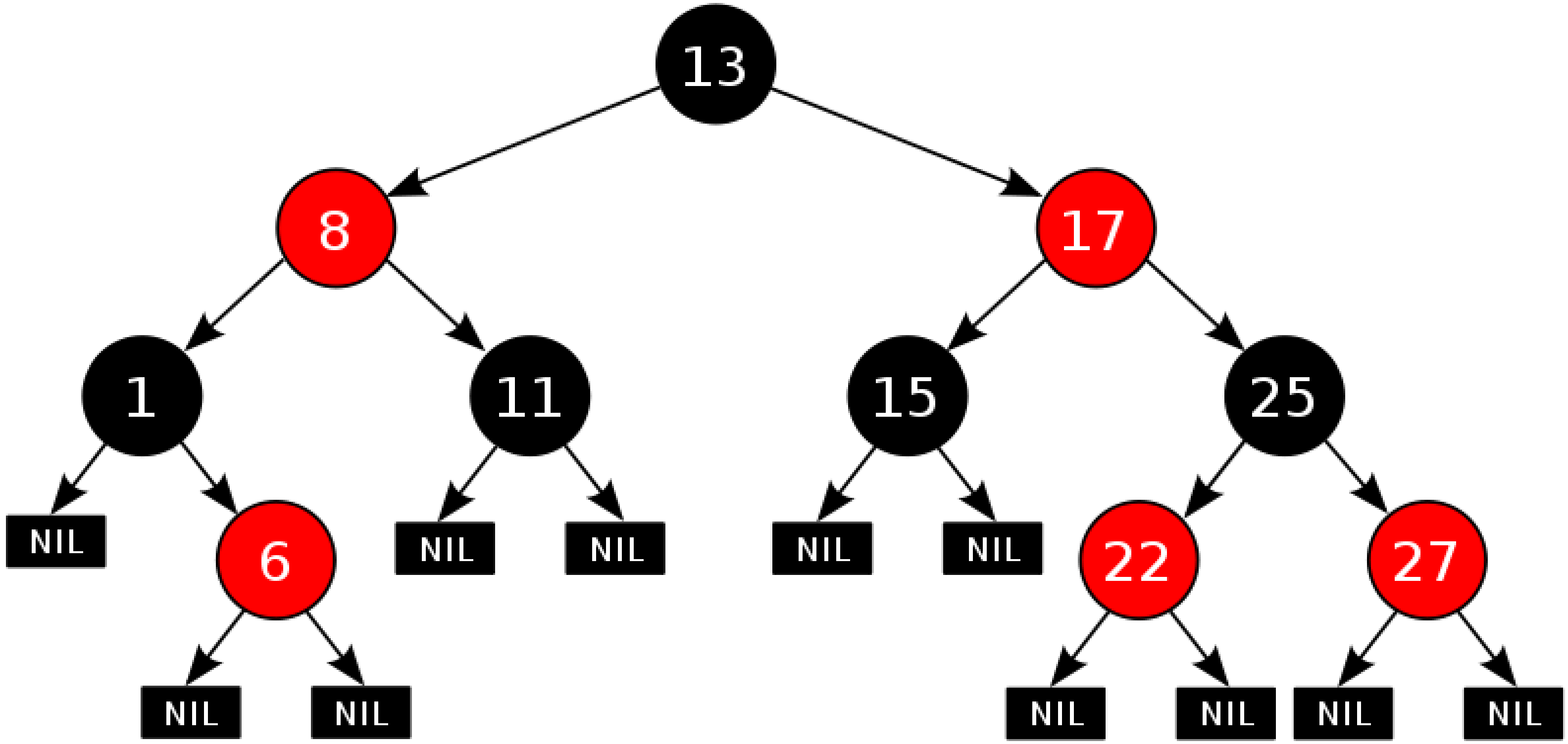
- Rebalance the tree



Red-Black Tree

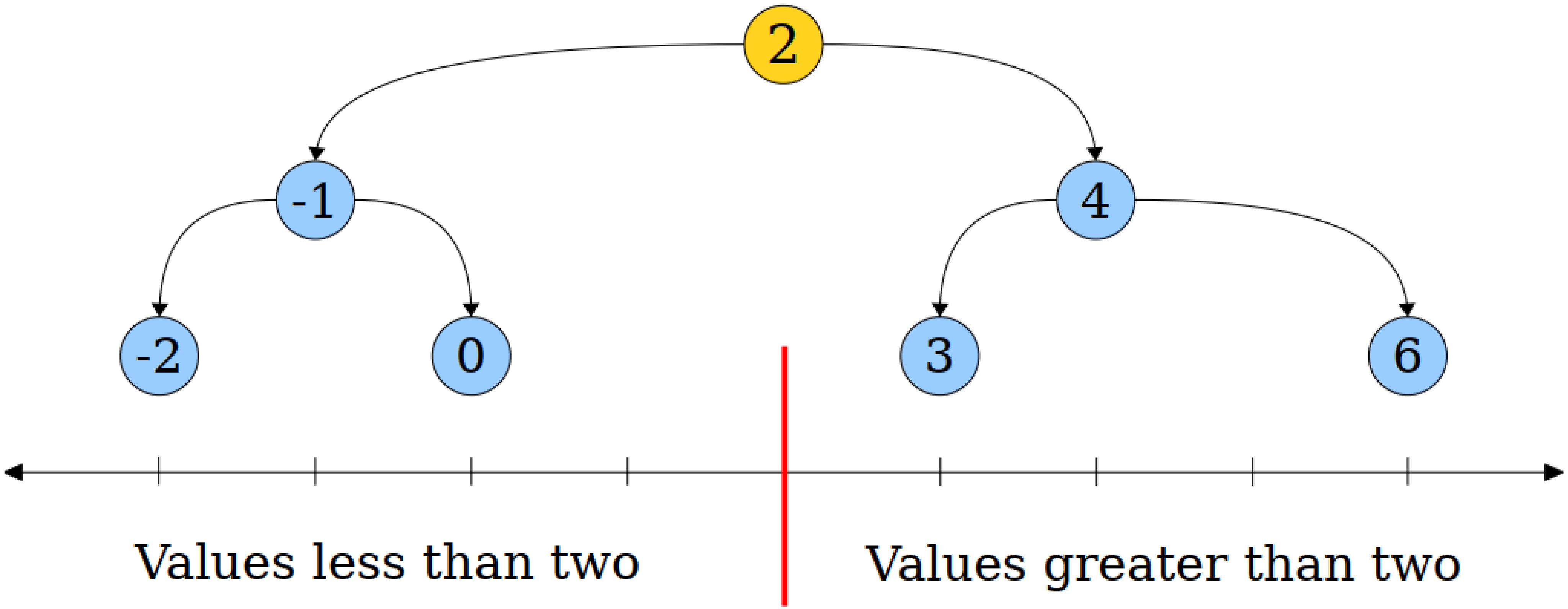
- Invented in 1972 by Rudolf Bayer.
- Red-Black tree is a binary search tree with the following rules:
  - Every node has a color either **red** or **black**.
  - The root of the tree is always **black**.
  - There are no two adjacent red nodes (A red node cannot have a red parent or red child).
  - Every path from a node (including root) to any of its descendants NULL nodes has the same number of **black** nodes.
  - All leaf nodes are **black** nodes.

- Red-Black Tree is one type of **self-balancing** tree where each node has one extra bit that is often interpreted as color of the node
- This bit (the color) is used to ensure that the tree remains balanced

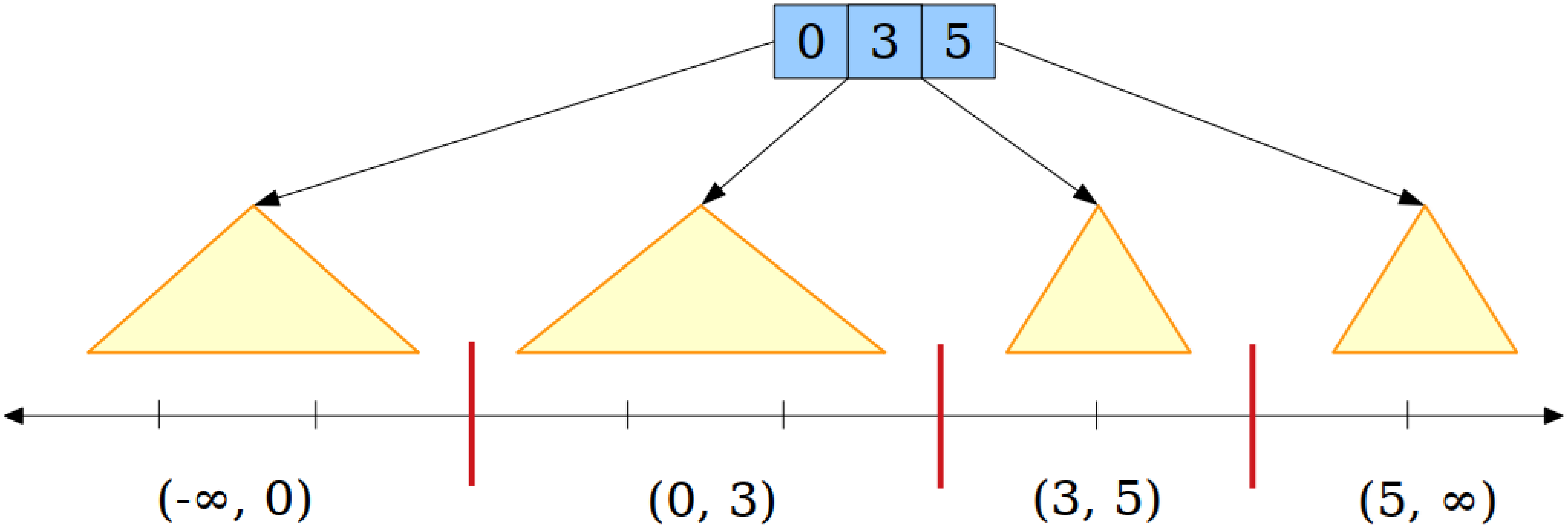


B Tree
2-3, 2-3-4 Tree

- In a binary search tree, each node stores a single key.
- That key splits the "key space" into two pieces, and each subtree stores the keys in those halves.

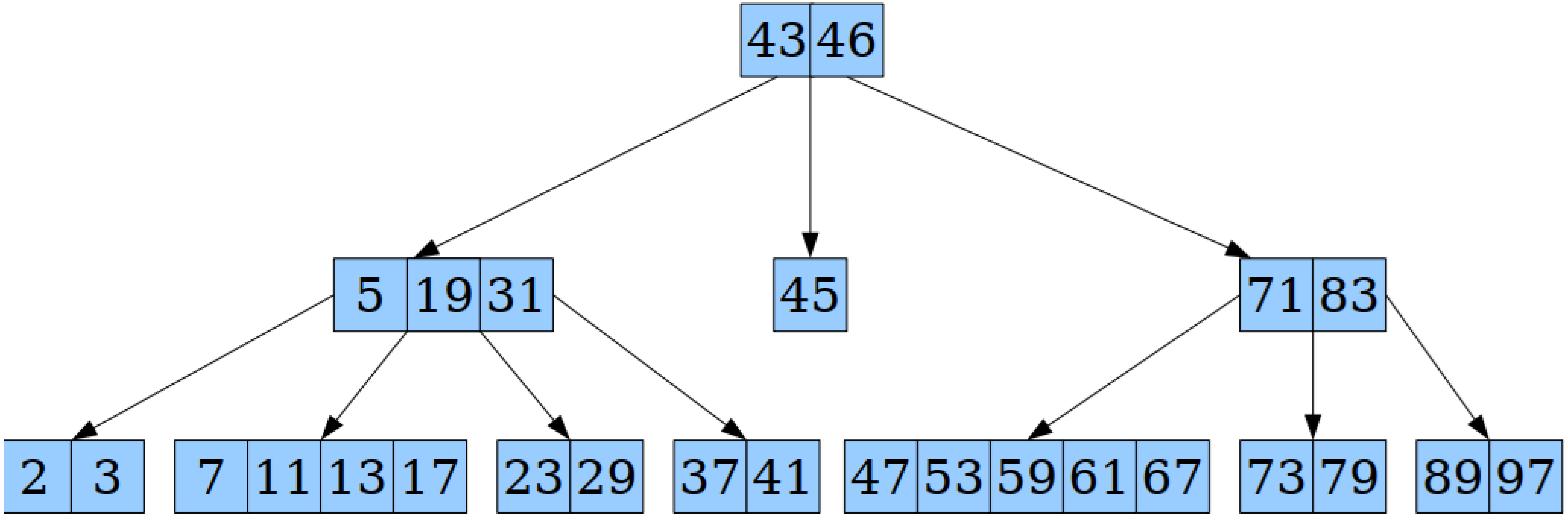


- In a **multiway search tree**, each node stores an arbitrary number of keys in sorted order.
- A node with k keys splits the key space into k+1 regions, with subtrees for keys in each region



B-Tree

- In a **multiway search tree**, each node stores an arbitrary number of keys in sorted order



- Surprisingly, it's a bit easier to build a balanced multiway tree than it is to build a balanced BST

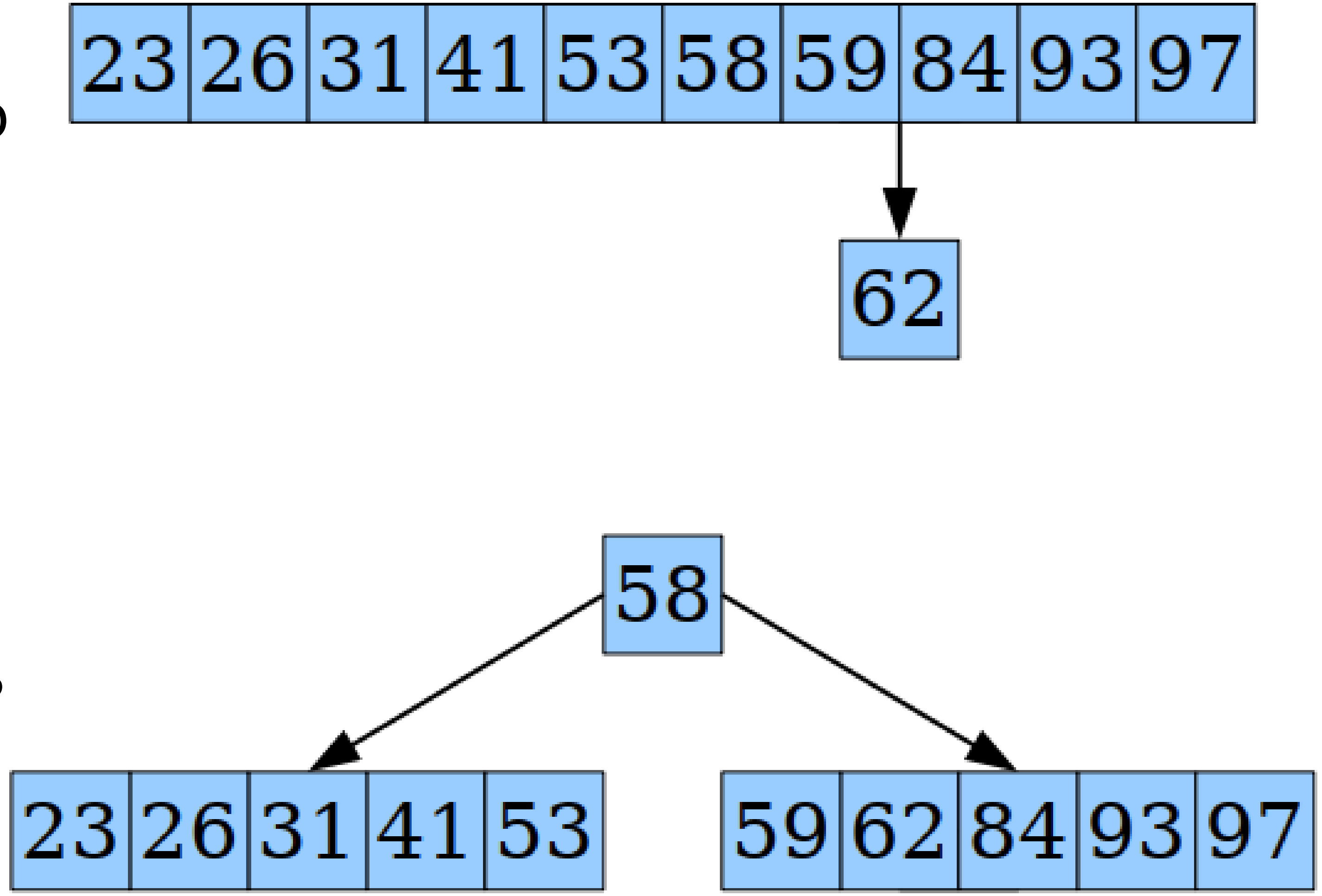
B-Tree

- In some sense, building a balanced multiway tree isn't all that hard.
- We can always just cram more keys into a single node!

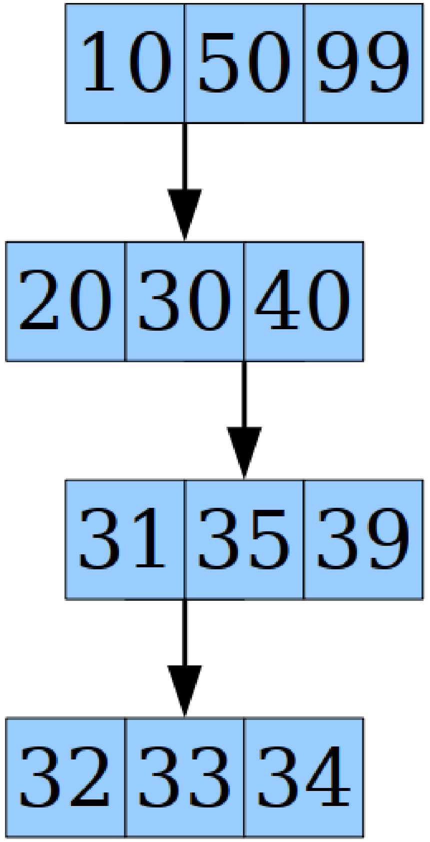
23	26	31	41	53	58	59	62	84	93	97

- At a certain point, this stops being a good idea – it's basically just a sorted array

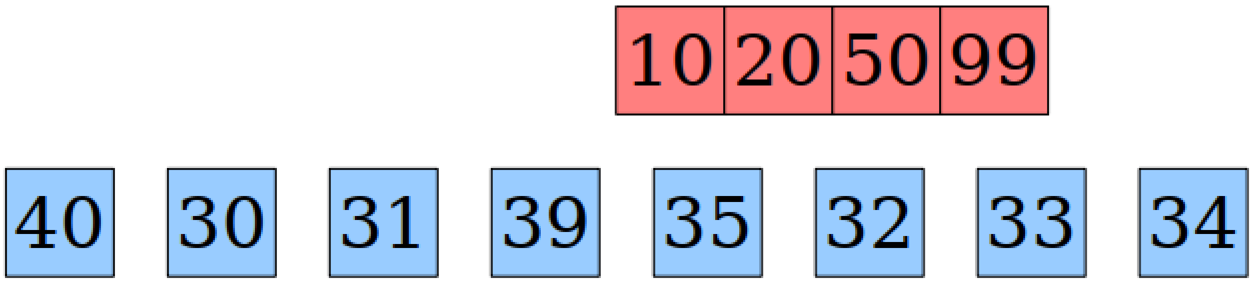
- What could we do if our nodes get too big?
- **Option 1:** Push keys down into new nodes.
- **Option 2:** Split big nodes, kicking keys higher up.
- What are some advantages of each approach?
- What are some disadvantages?



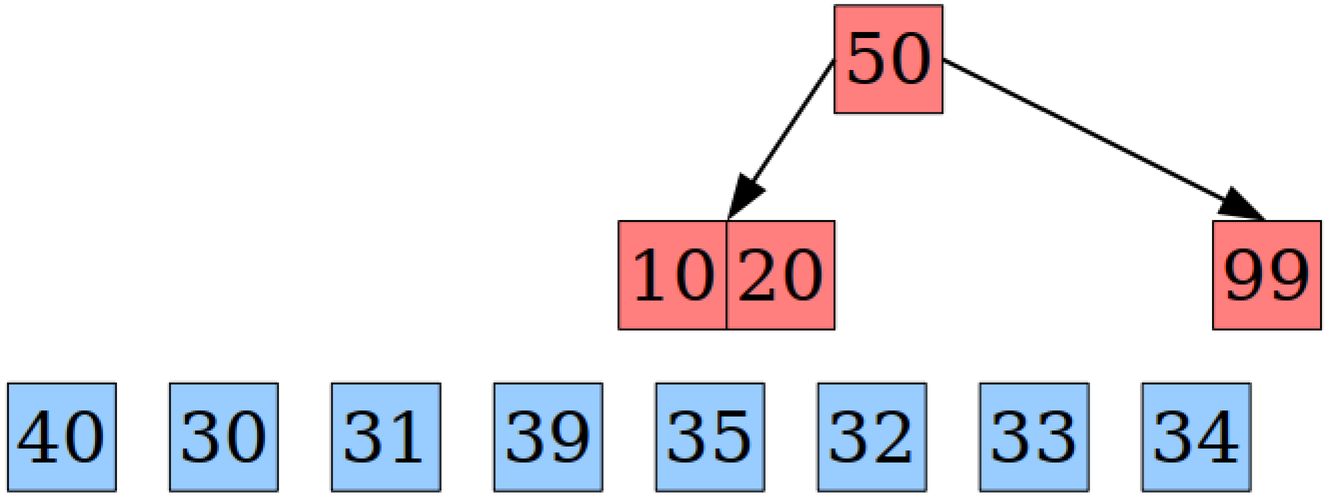
- **Option 1:** Push keys down into new nodes
  - Simple to implement
  - Can lead to tree imbalances



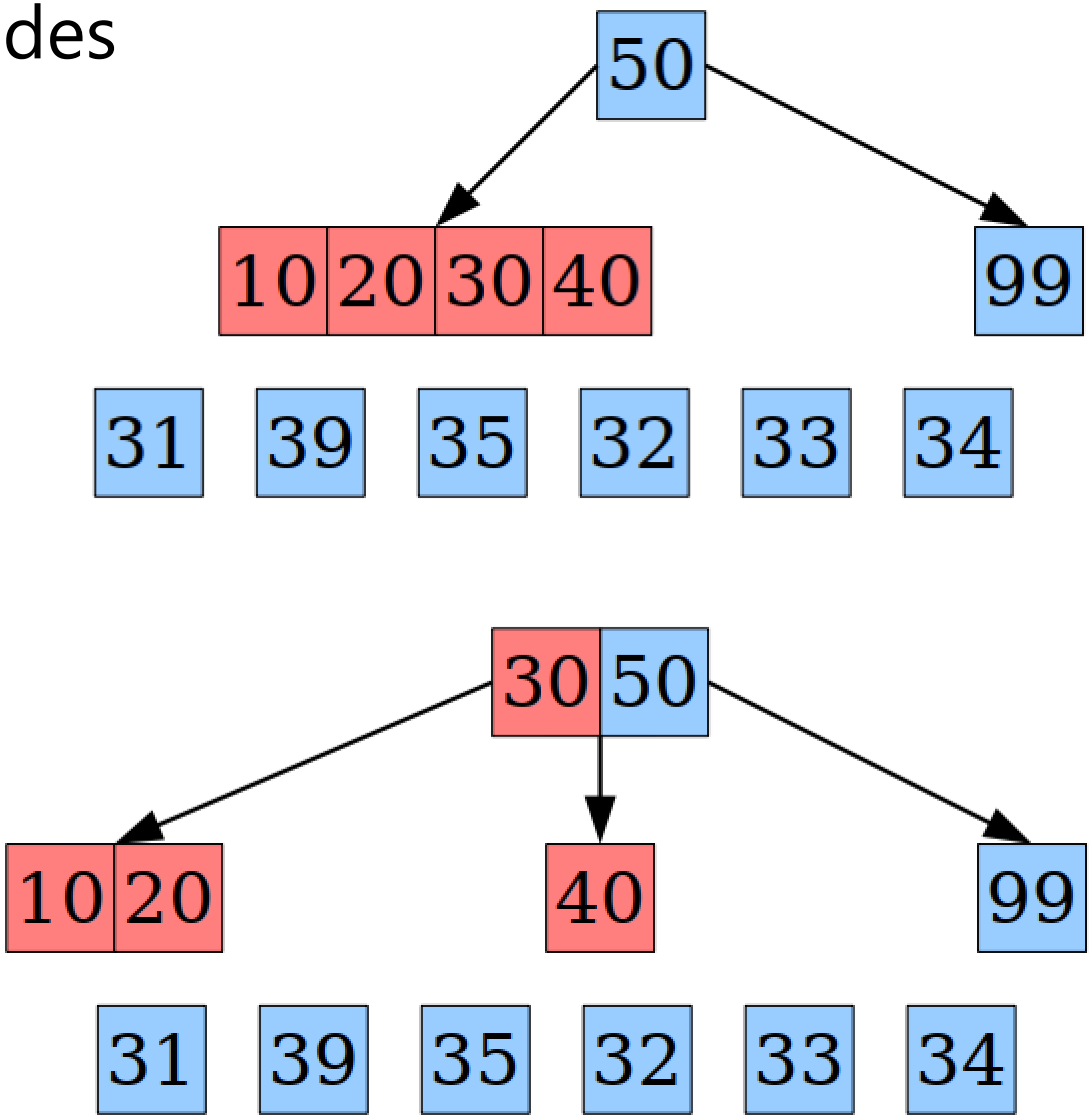
- Option 1: Push keys down into new nodes
  - Simple to implement
  - Can lead to tree imbalances
- Option 2: Split big nodes, kicking keys higher up
  - Keeps the tree balanced
  - Slightly trickier to implement



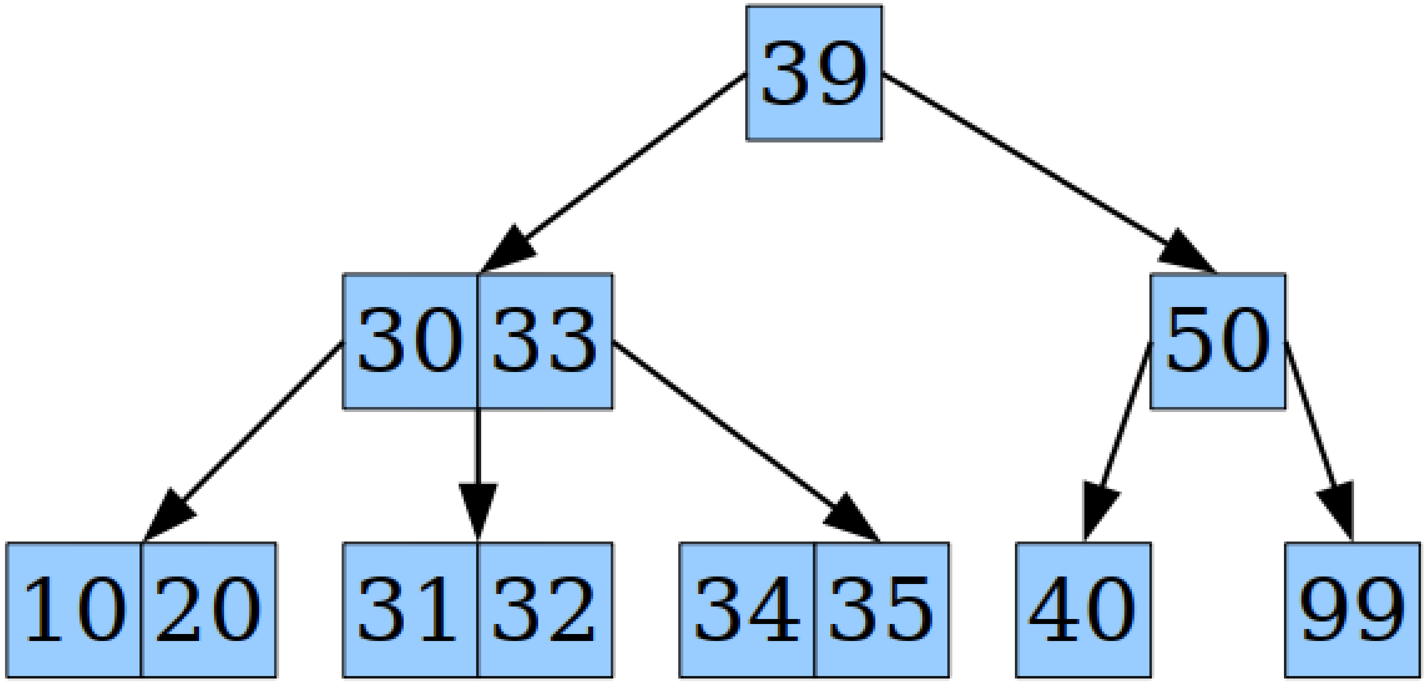
- Option 1: Push keys down into new nodes
  - Simple to implement
  - Can lead to tree imbalances
- Option 2: Split big nodes, kicking keys higher up
  - Keeps the tree balanced
  - Slightly trickier to implement



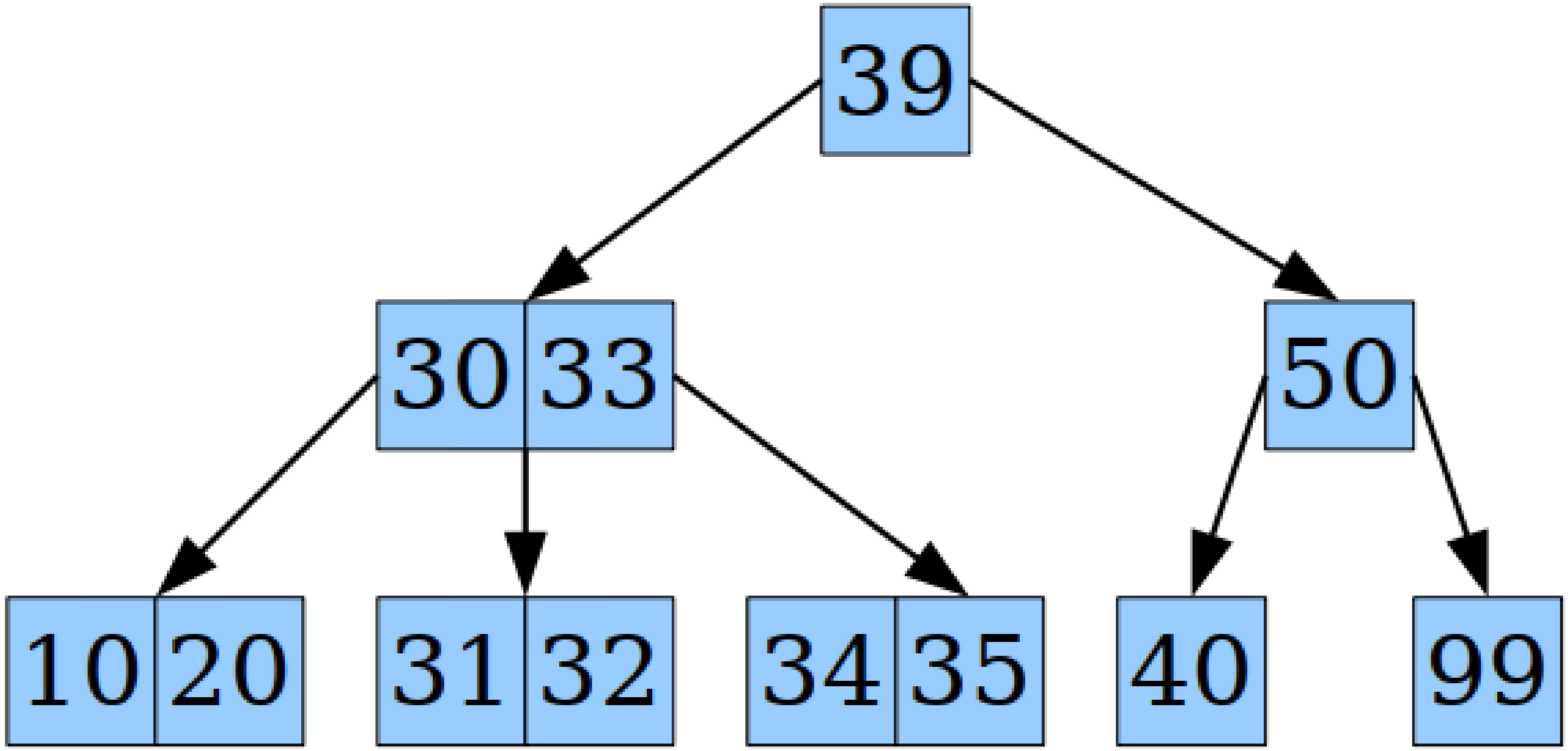
- Option 1: Push keys down into new nodes
  - Simple to implement
  - Can lead to tree imbalances
- Option 2: Split big nodes, kicking keys higher up
  - Keeps the tree balanced
  - Slightly trickier to implement



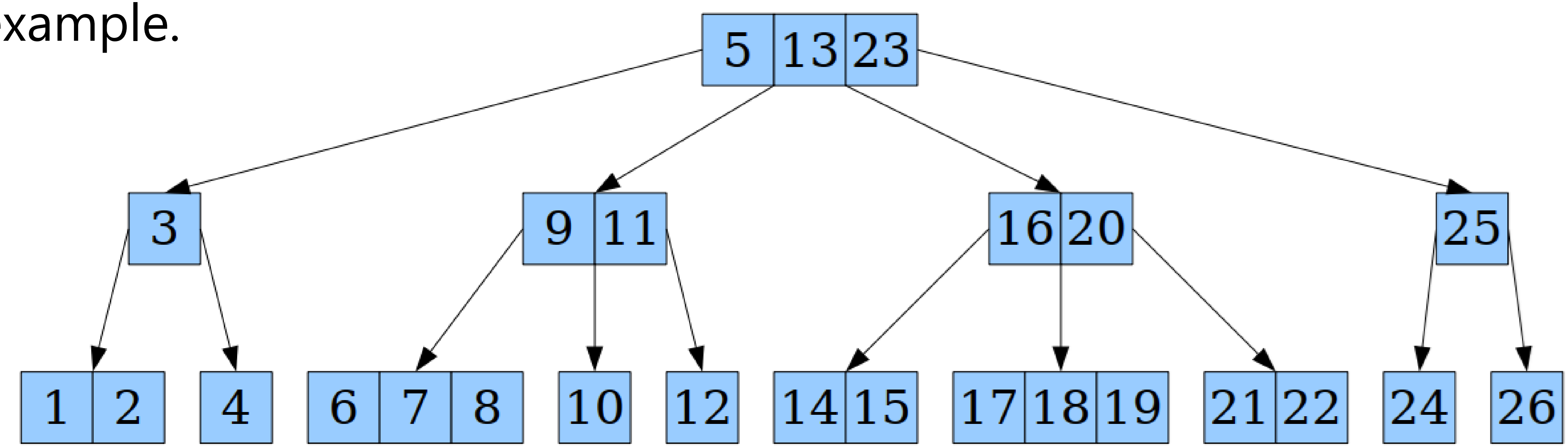
- General idea: Keep nodes holding roughly between b and 2b keys, for some parameter b
  - Exception: the root node can have fewer keys (1 to b keys)
- If a node gets too big, split it and kick a key higher up



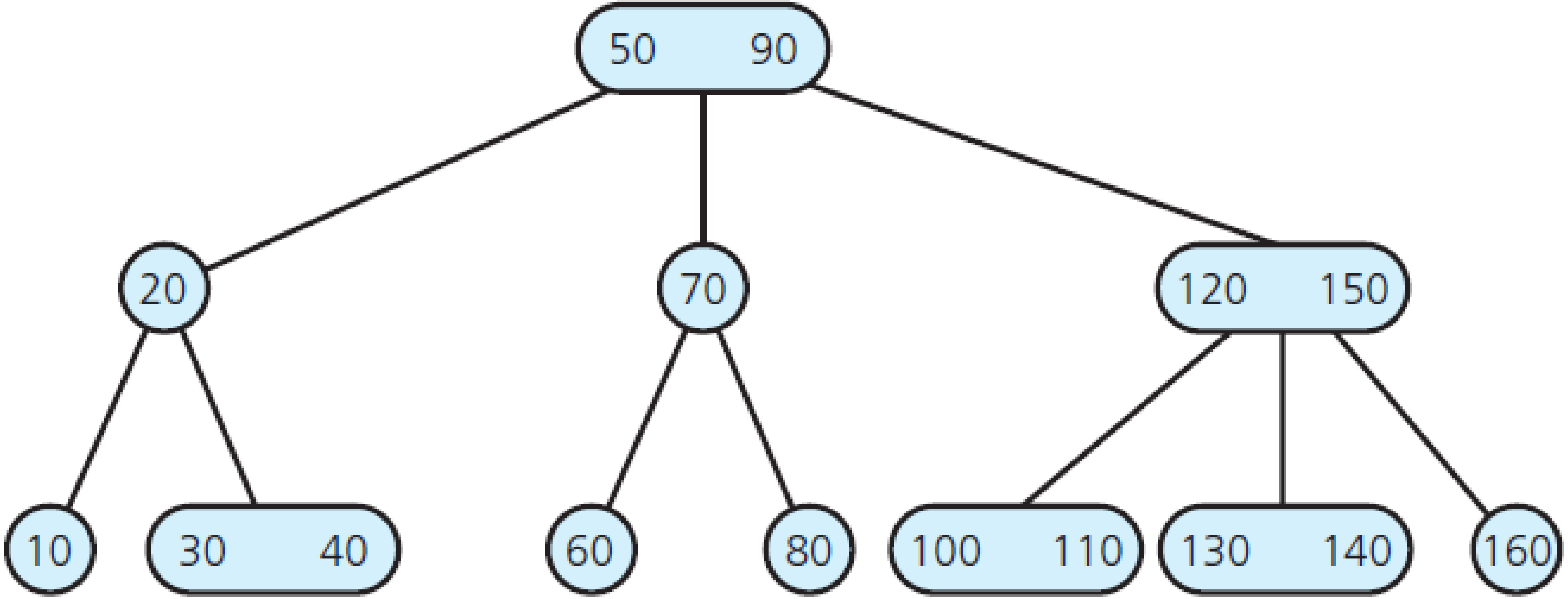
- Advantage 1: The tree is always balanced.
- Advantage 2: Insertions and lookups are pretty fast



- A 2-3-4 tree is a B-tree of order 2. Specifically:
  - each node has between 1 and 3 keys;
  - each node is either a leaf or has one more child than key; and
  - all leaves are at the same depth.
- You actually saw this B-tree earlier! It's the type of tree from our insertion example.



2-3 Tree



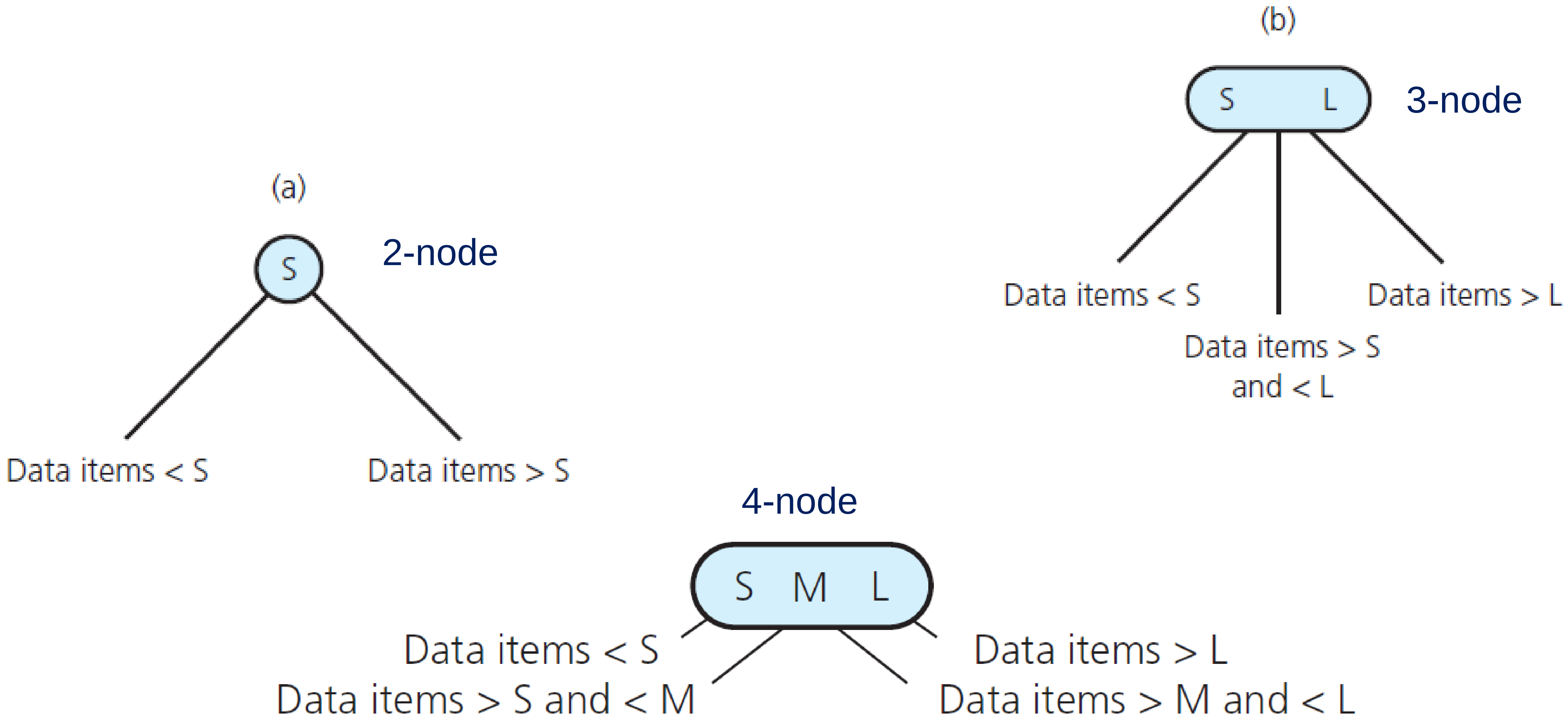
2-node, 3-node, 4-node

- A 2-node (has two children) must contain **single** data item greater than left child's item(s) and less than right child's item(s).

- A 3-node (has three children) must contain **two** data items, S and L, such that
  - S is greater than left child's item(s) and less than middle child's item(s);
  - L is greater than middle child's item(s) and less than right child's item(s).

- A 4-node (has our children) must contain **three** data items S, M, and L that satisfy:
  - S is greater than left child's item(s) and less than middle-left child's item(s)
  - M is greater than middle-left child's item(s) and less than middle-right child's item(s);
  - L is greater than middle-right child's item(s) and less than right child's item(s).

2-node, 3-node, 4-node



2-3 Tree

- Invented by John Hopcroft in 1970.
- 2-3 tree is a tree in which
  - Every internal node is either a 2-node or a 3-node.
  - Leaves have no children and may contain either one or two data items.

2-3-4 Tree

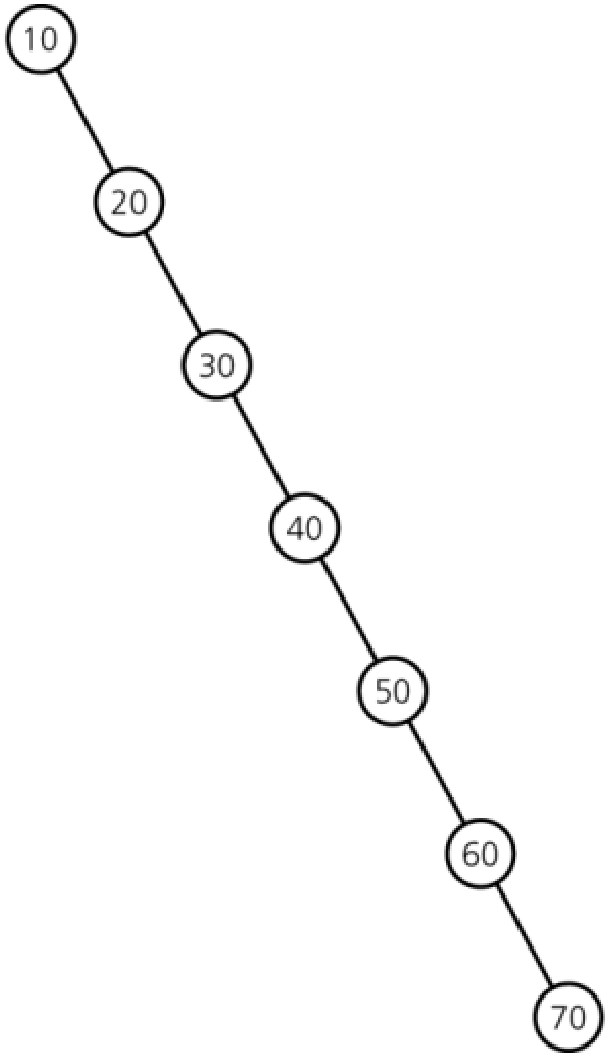
- 2-3-4 tree is a tree in which
  - Every internal node is a 2-node, a 3-node or a 4-node.
  - Leaves have no children and may contain either one, two or three data items.

Properties

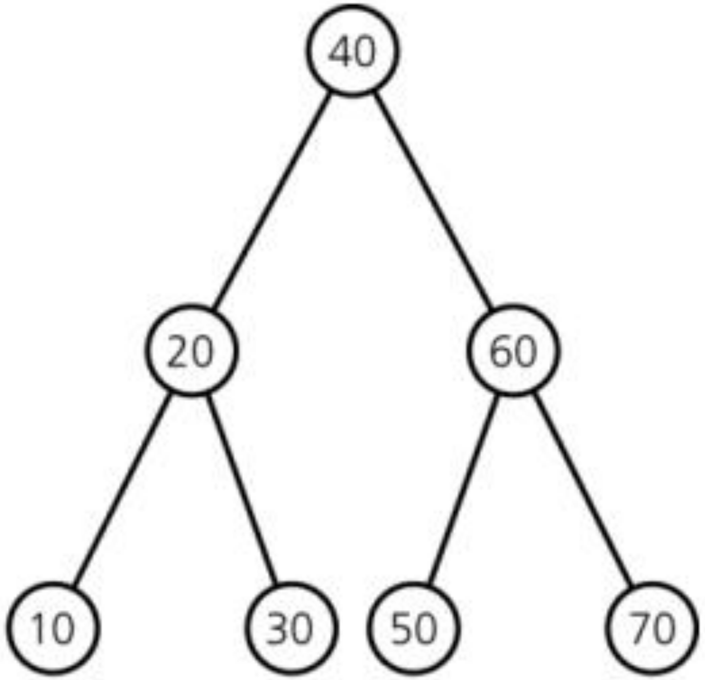
- All leaves are at the same level
- 2-3, 2-3-4 is self-balancing tree
- The elements in each node should be sorted from smallest to greatest

Properties

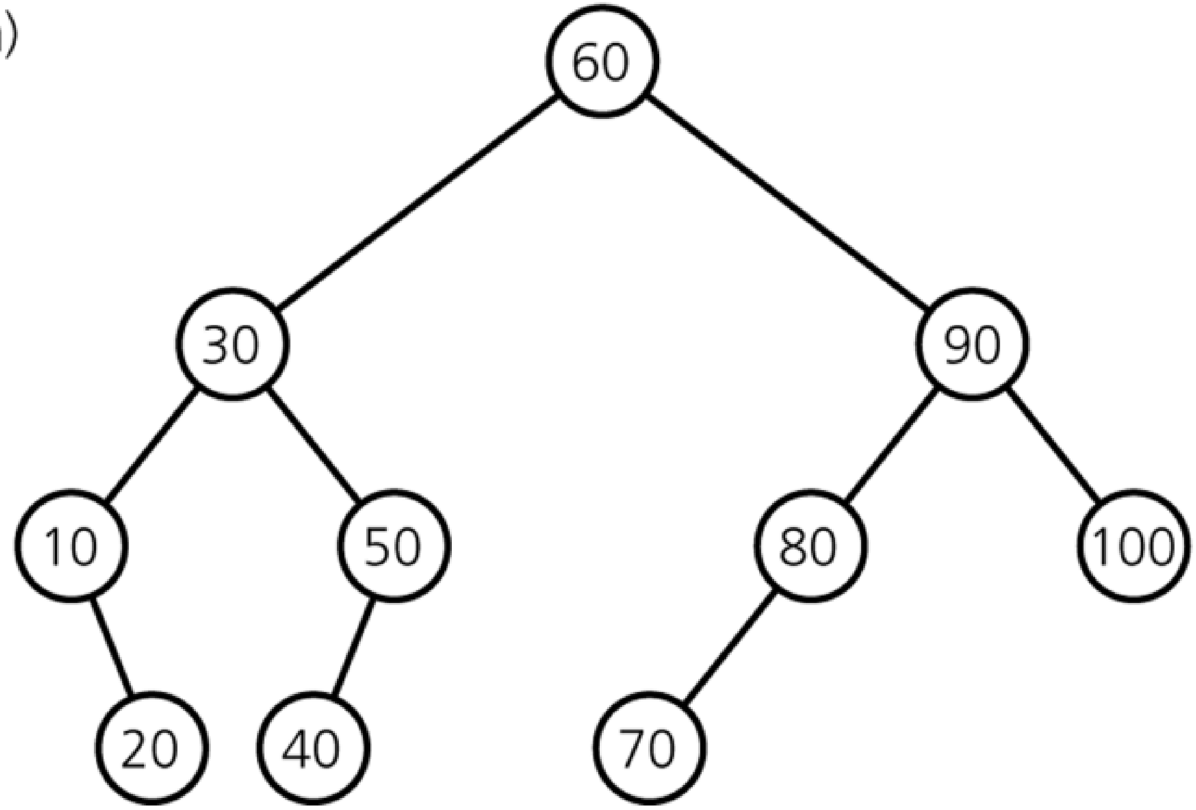
(a)

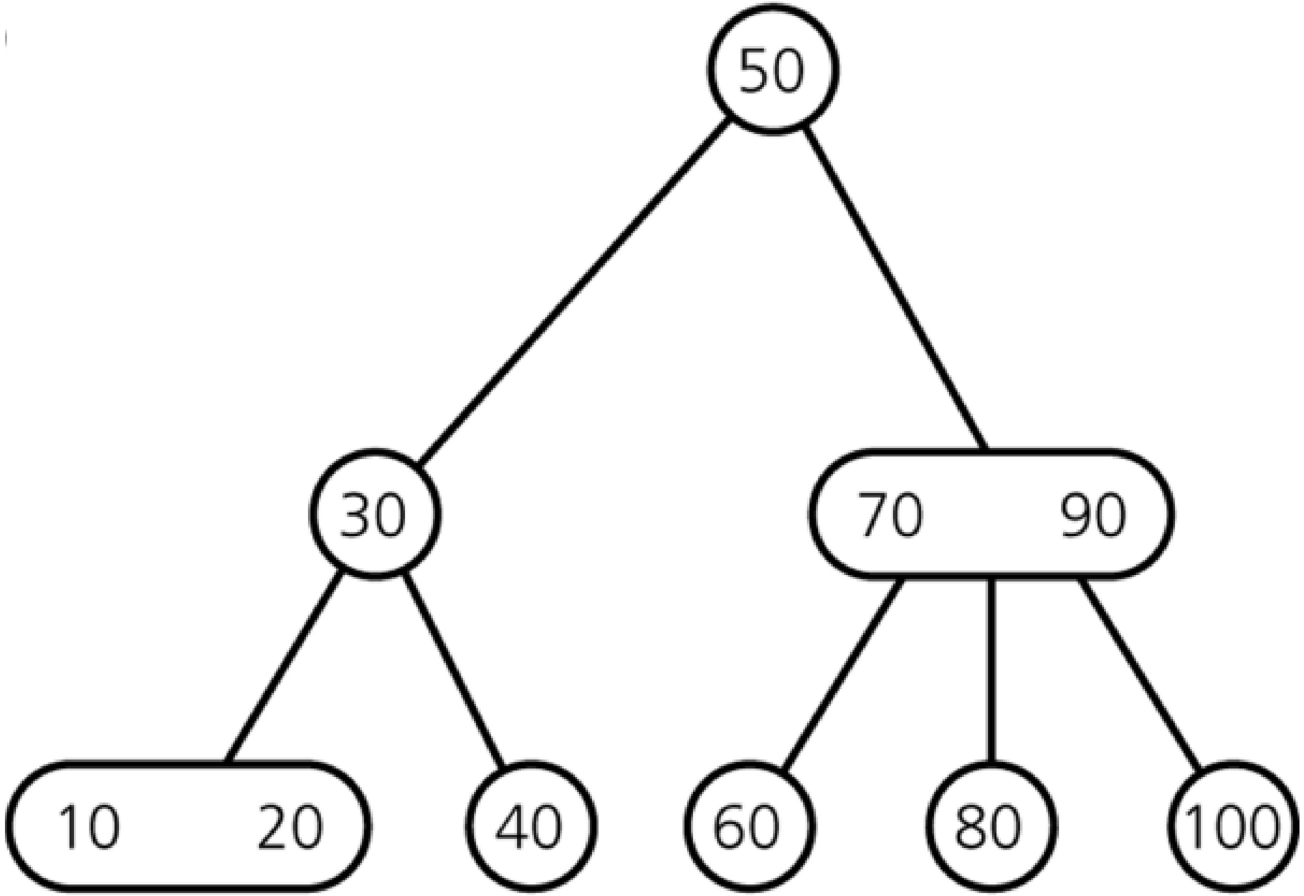


(b)

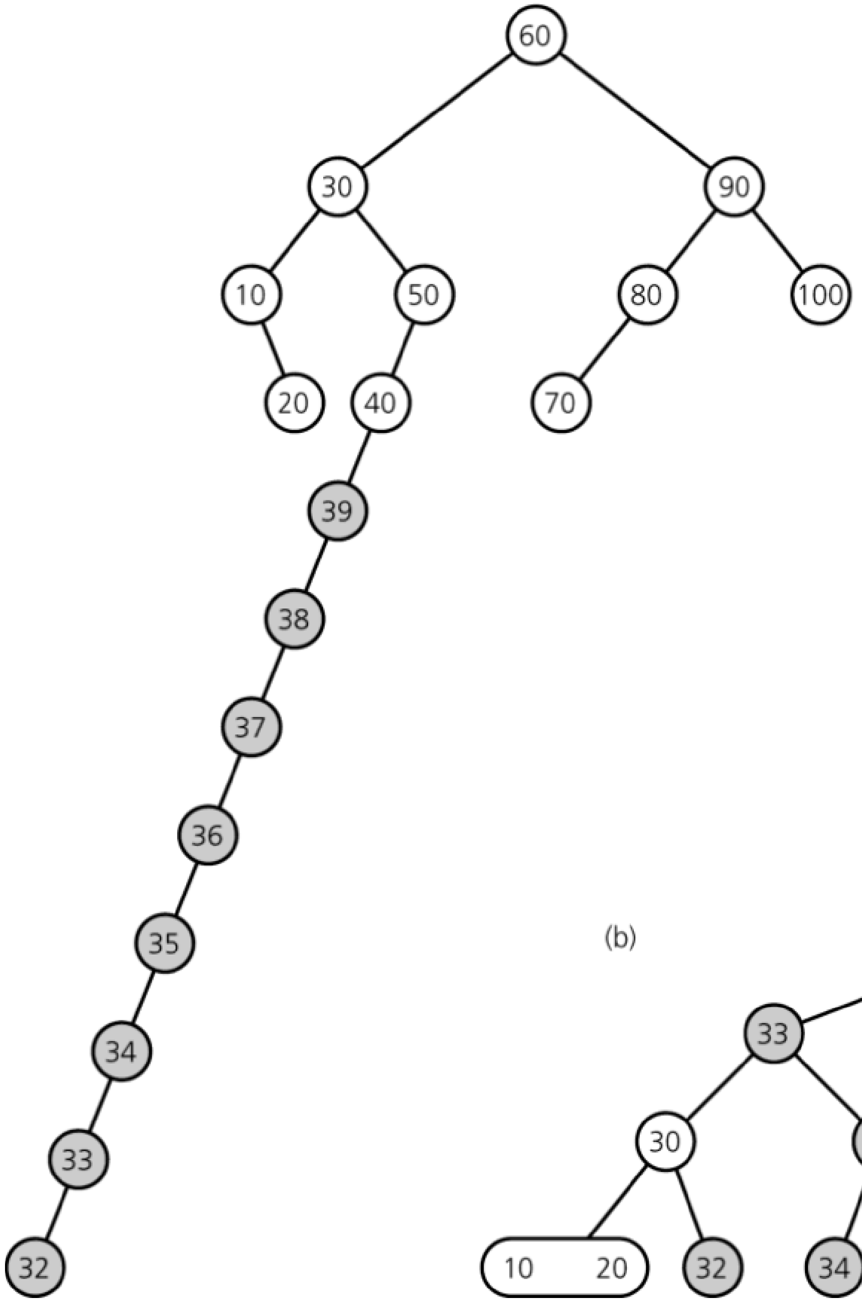


(a)



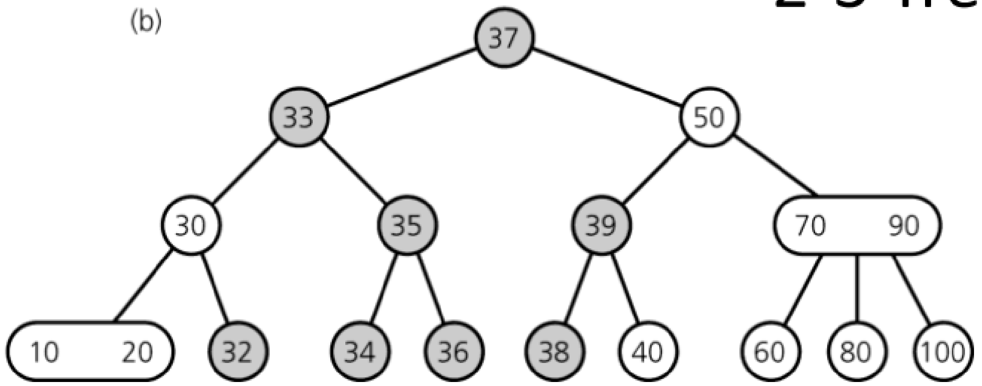


Properties



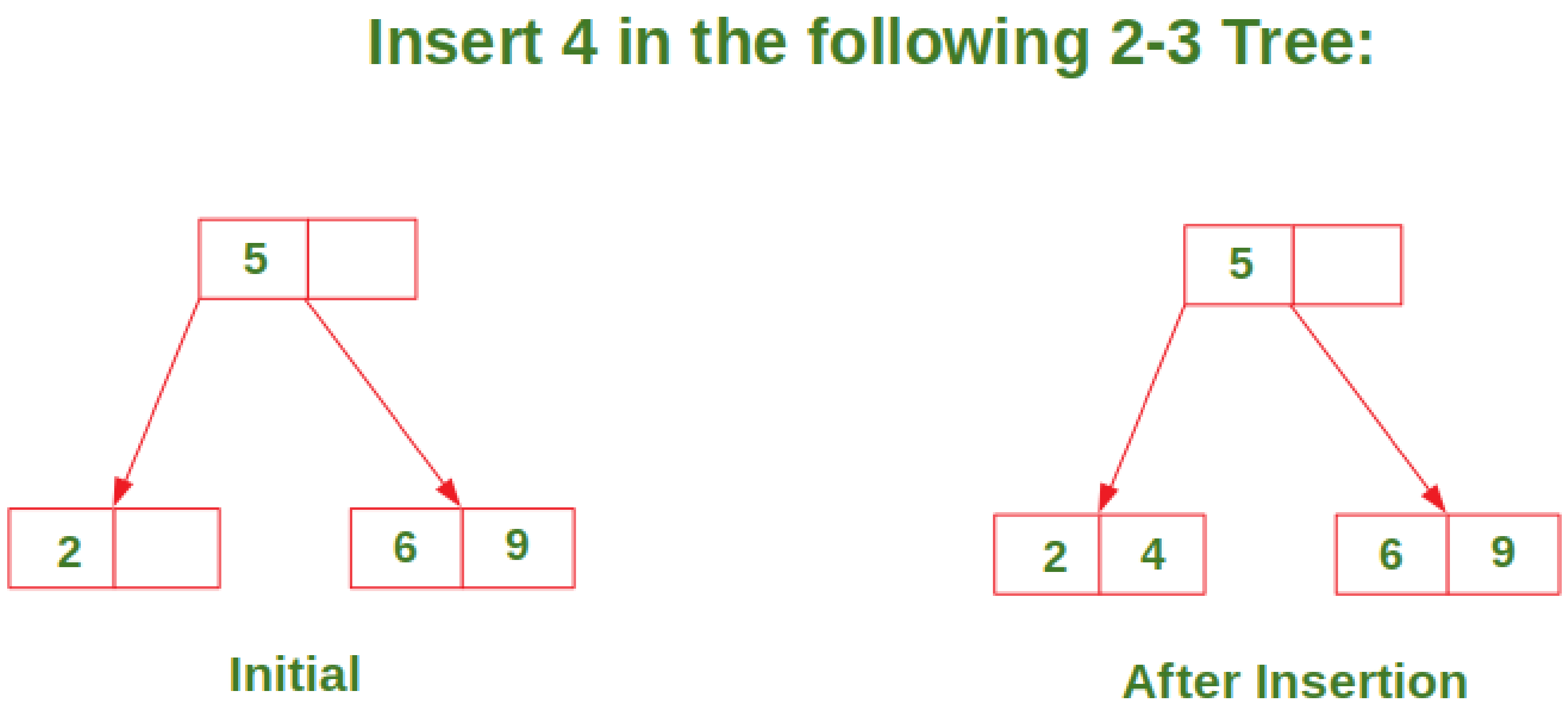
both trees after inserting items 39, 38, ... 32

2-3 Tree

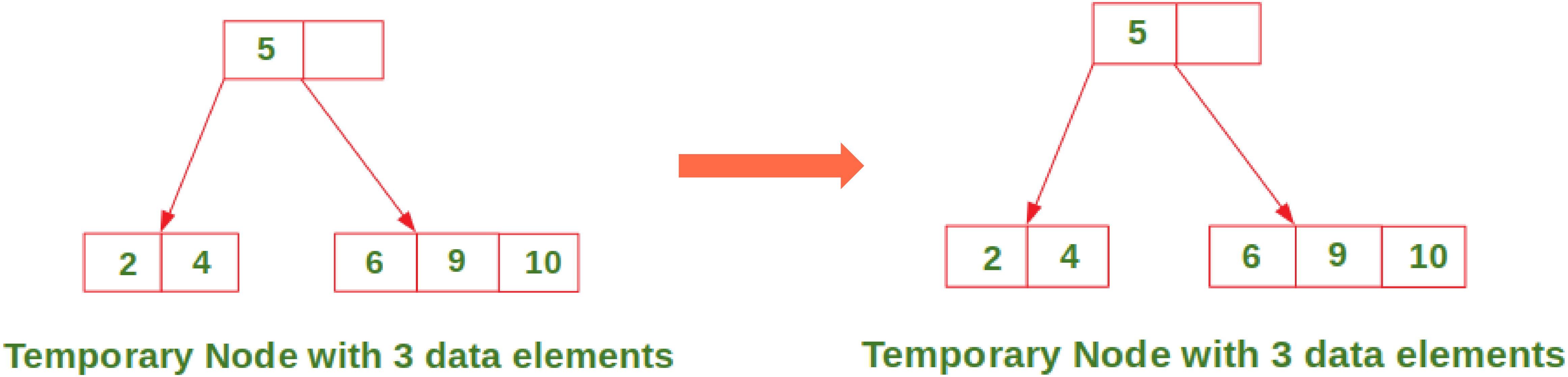


Insertion: 2-3 Tree

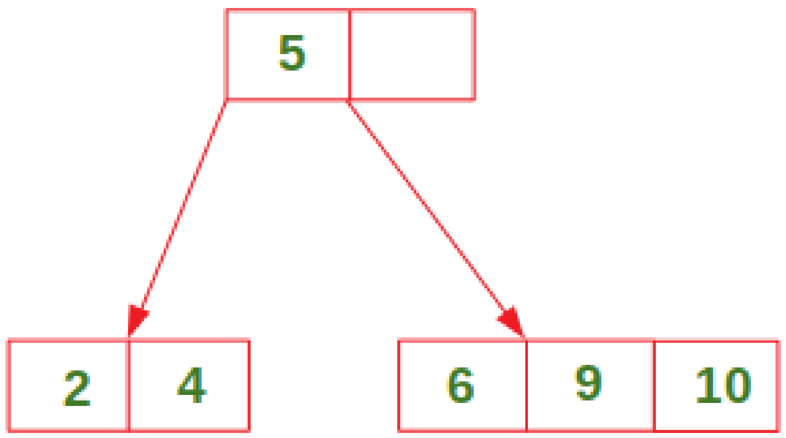
- **Case 1:** Insert in a node with only one data element



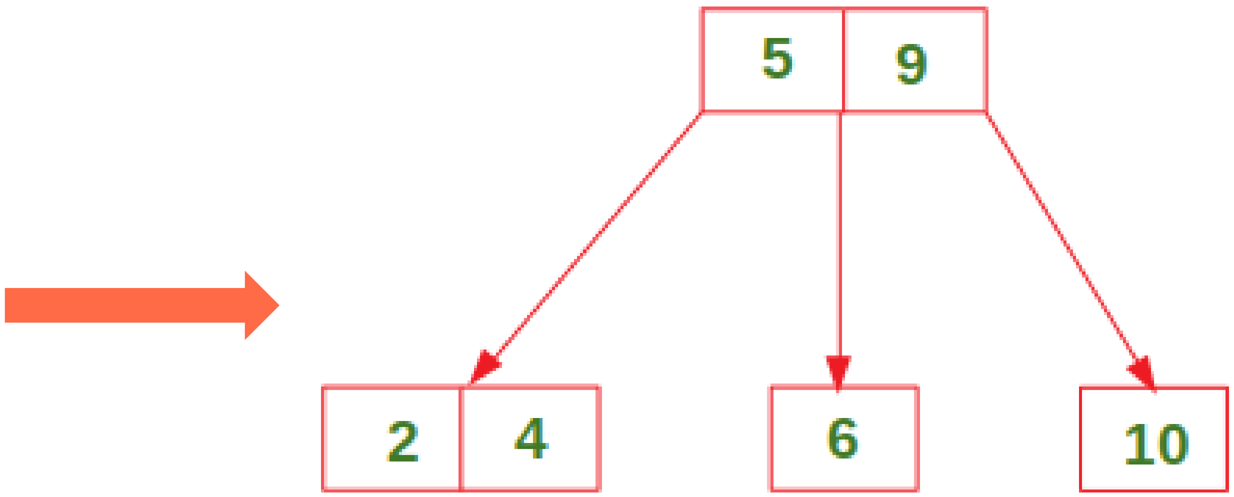
- **Case 2:** Insert in a node with two data elements whose parent contains only one data element



- **Case 2:** Insert in a node with two data elements whose parent contains only one data element



Temporary Node with 3 data elements



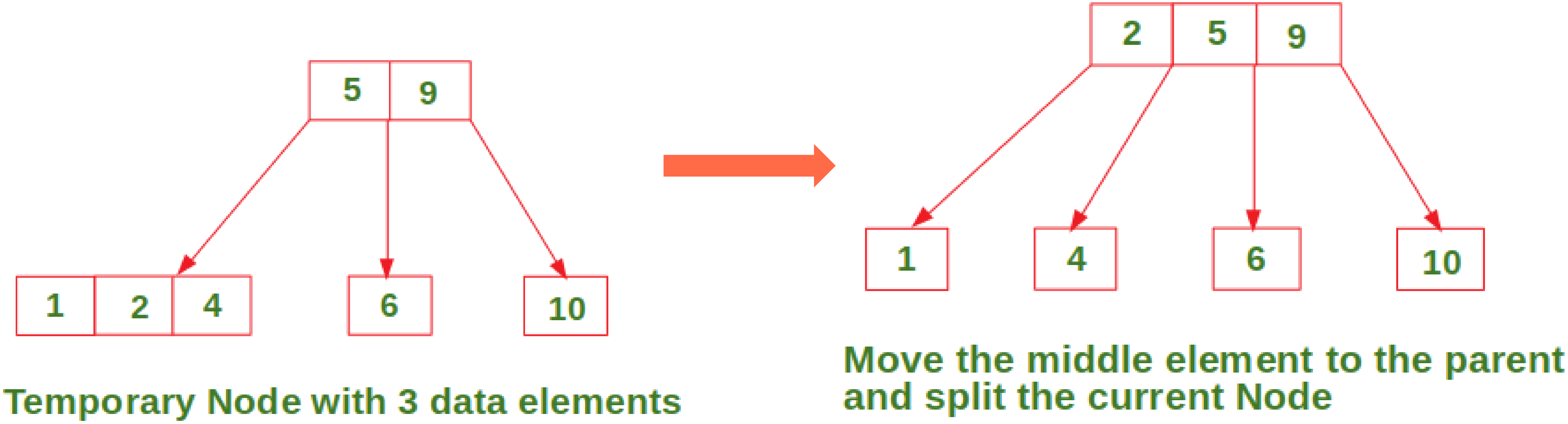
Move the middle element to parent and split the current Node

- **Case 3:** Insert in a node with two data elements whose parent also contains two data elements

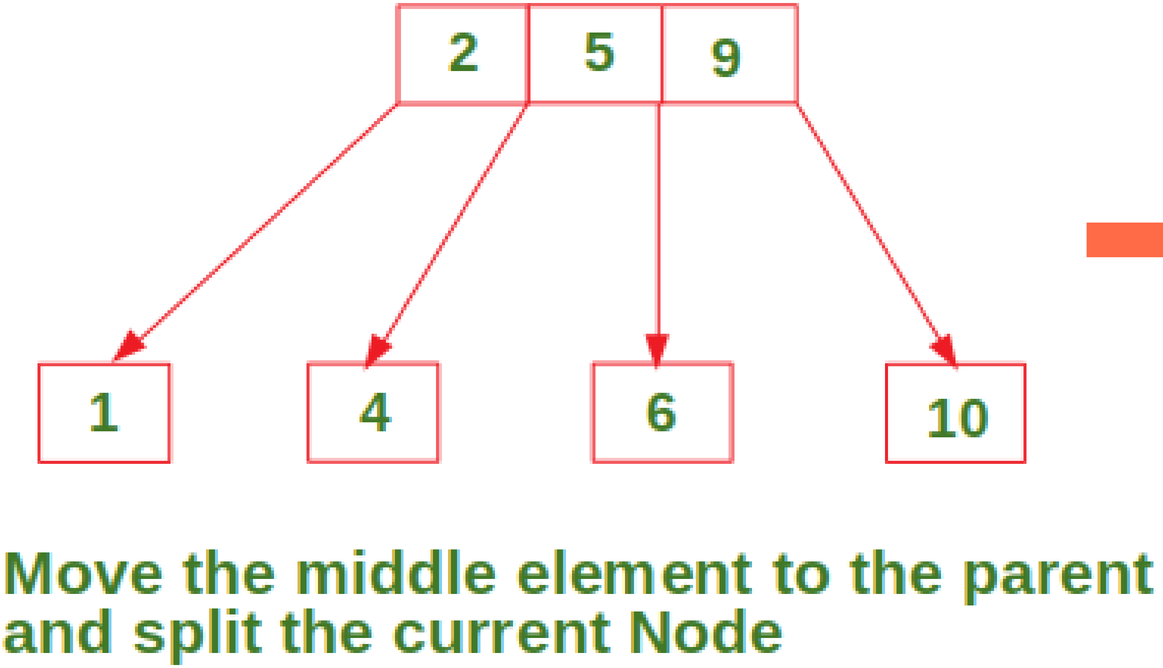
Insert 1 in the following 2-3 Tree:

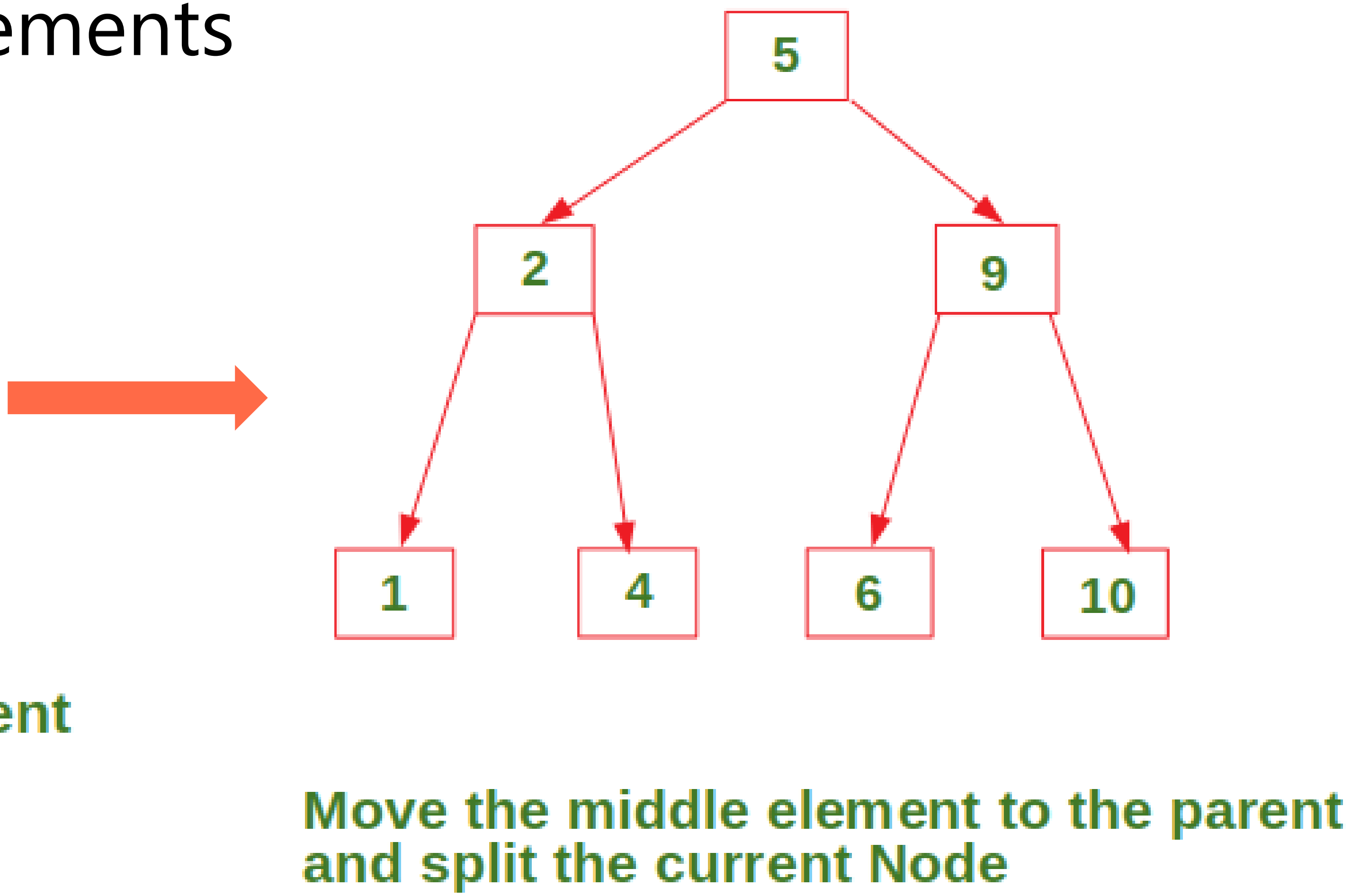
Insertion: 2-3 Tree

- **Case 3:** Insert in a node with two data elements whose parent also contains two data elements



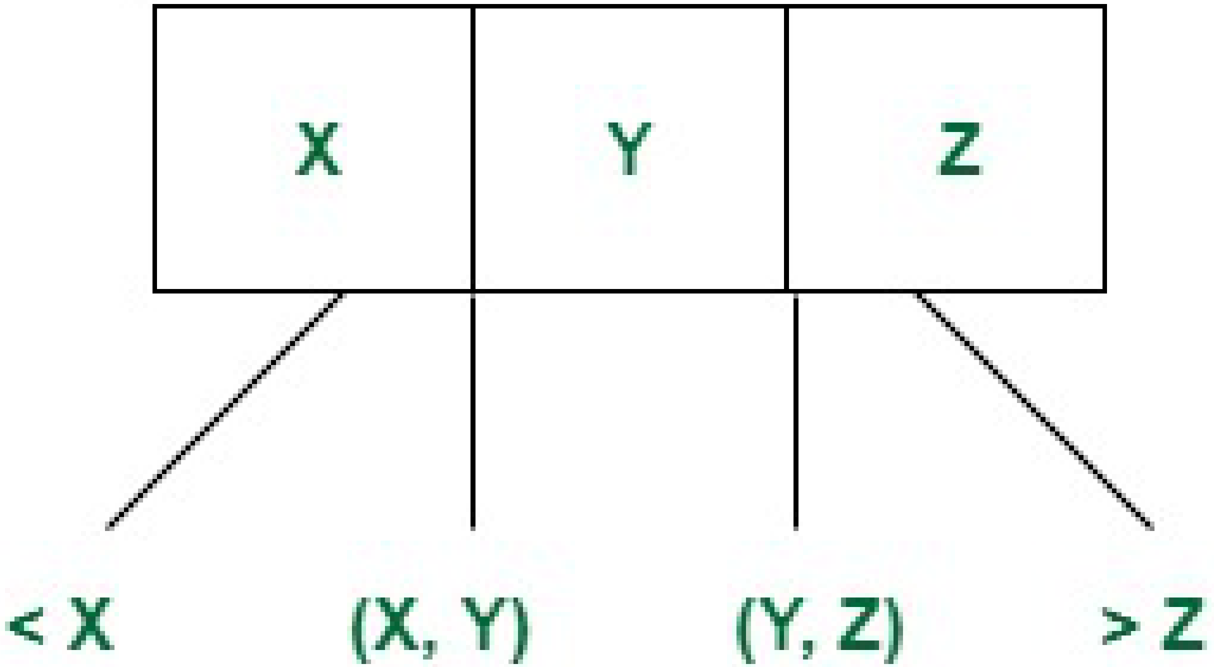
- **Case 3:** Insert in a node with two data elements whose parent also contains two data elements





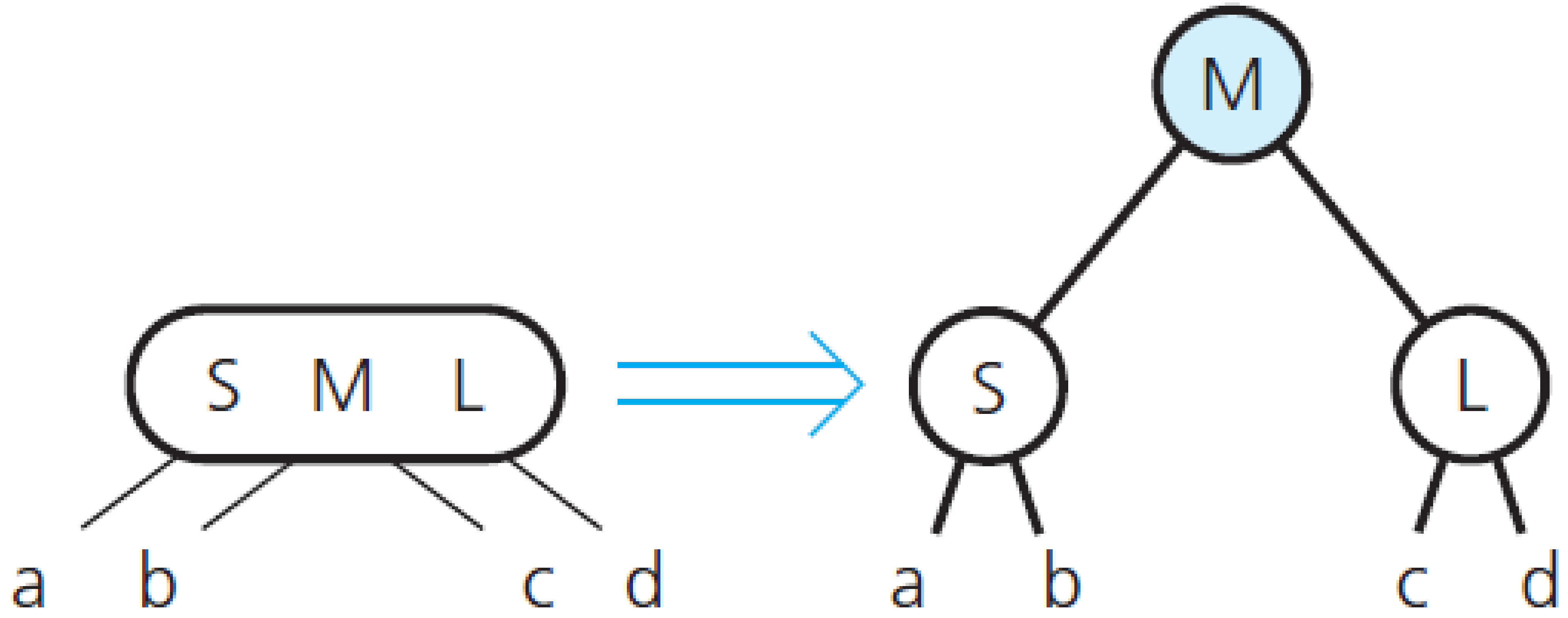
Insertion: 2-3-4 Tree

- Insertion algorithm splits a node by moving one of its items up to its parent node



Insertion: 2-3-4 Tree

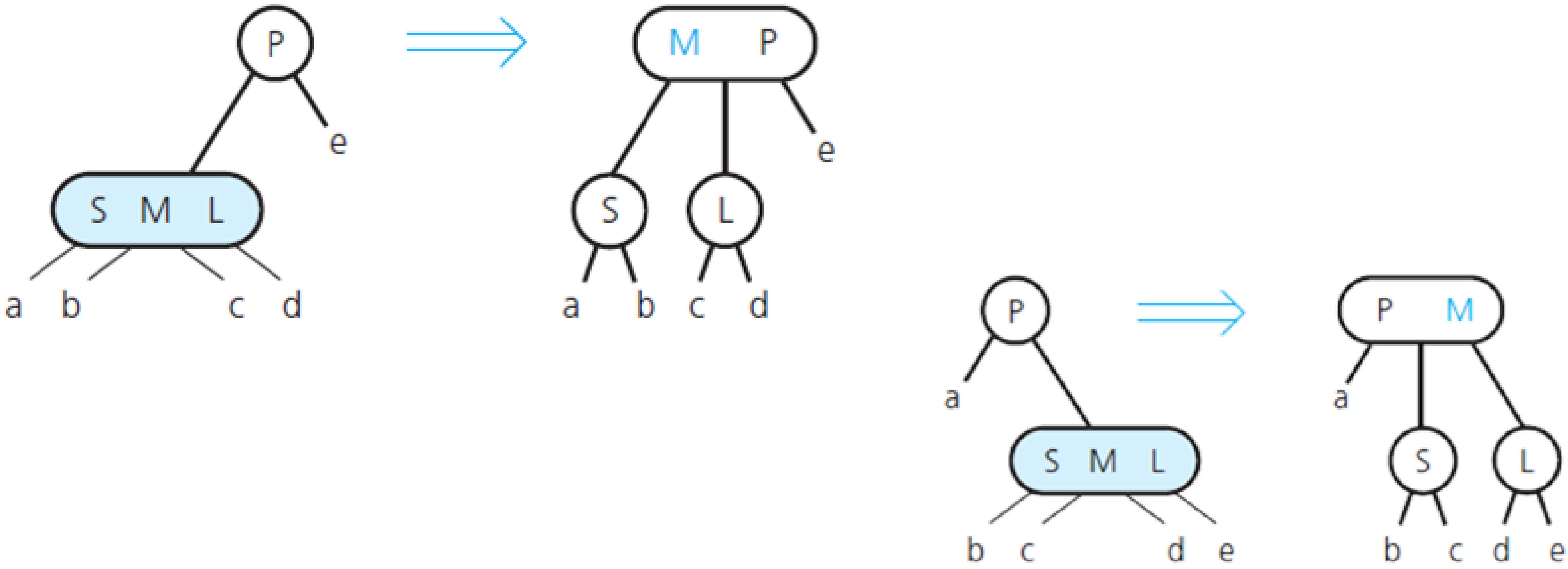
- Splitting a 4-node root during insertion into a 2-3-4 tree



Insertion: 2-3-4 Tree

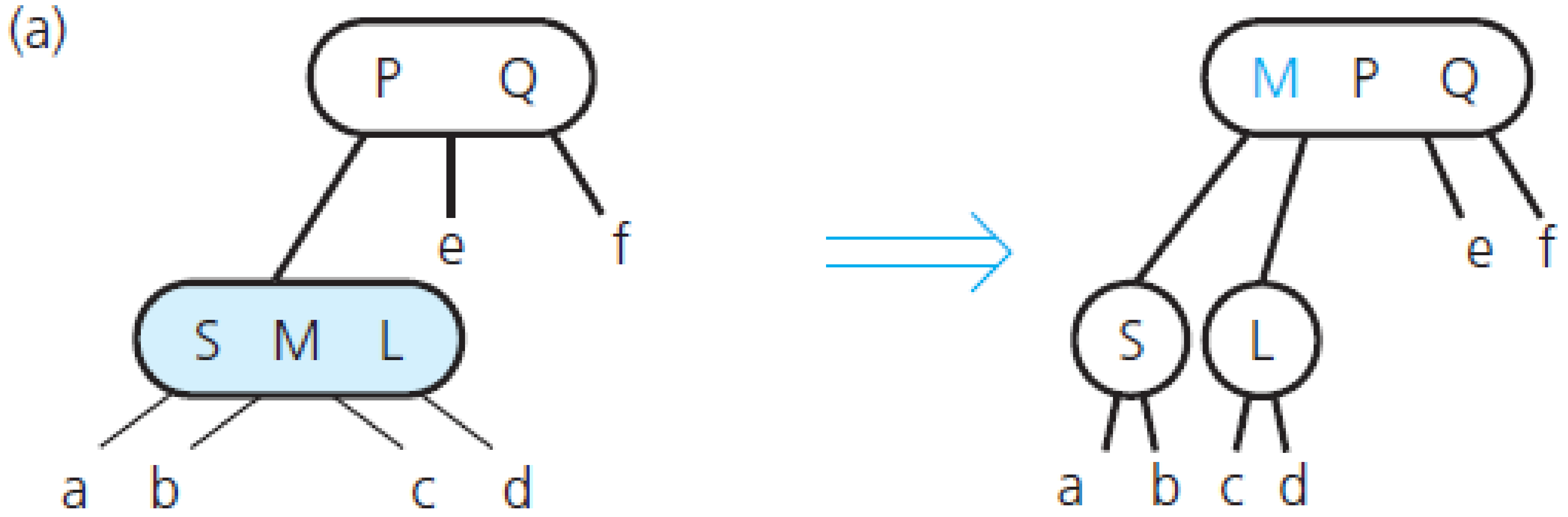
- Splitting a 4-node whose parent is a 2-node during insertion into a 2-3-4 tree, when the 4-node is a (a) left child; (b) right child

(a)



Insertion: 2-3-4 Tree

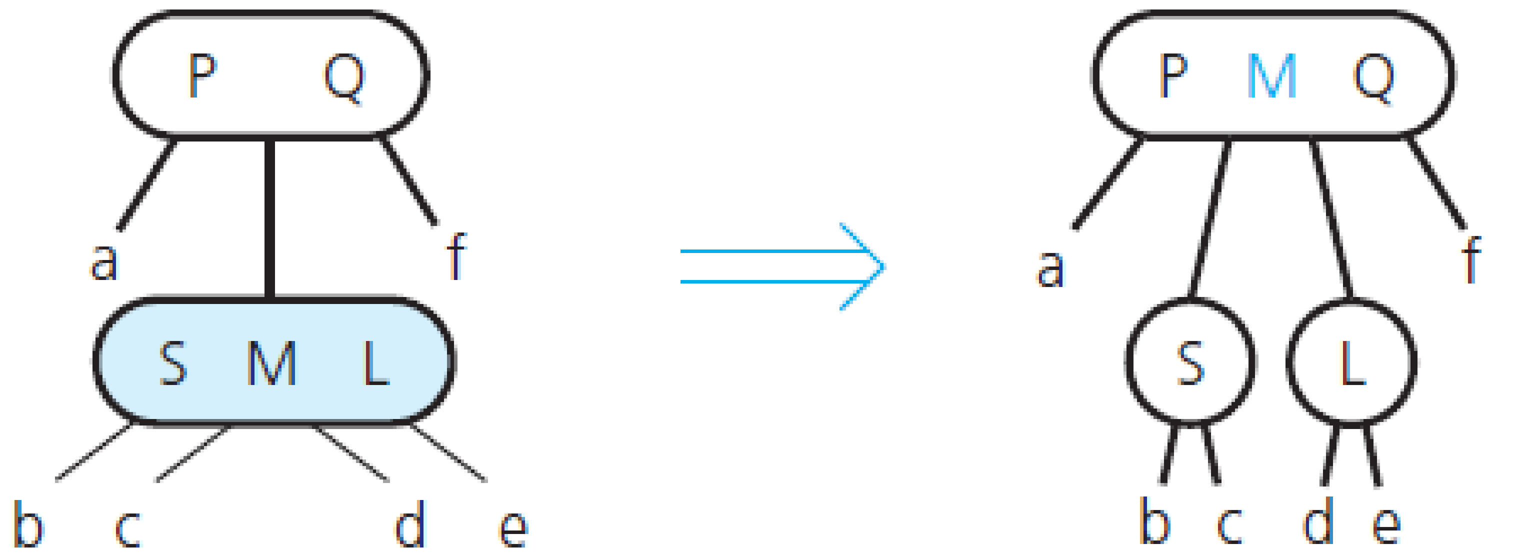
- Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (a) left child



Insertion: 2-3-4 Tree

- Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (b) middle child

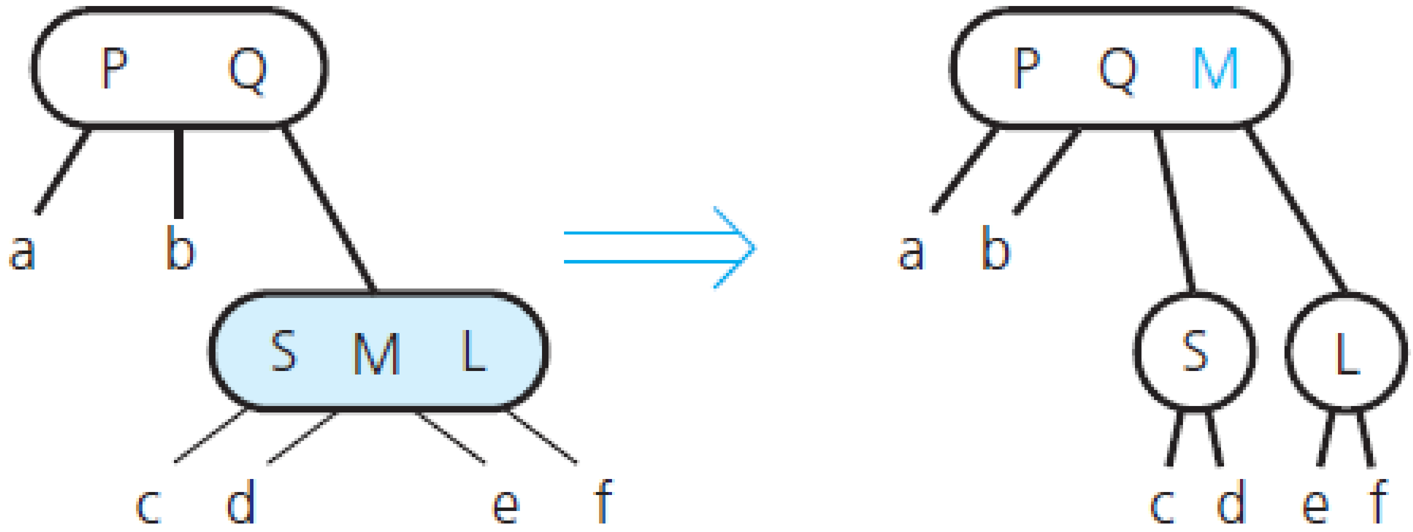
(b)



Insertion: 2-3-4 Tree

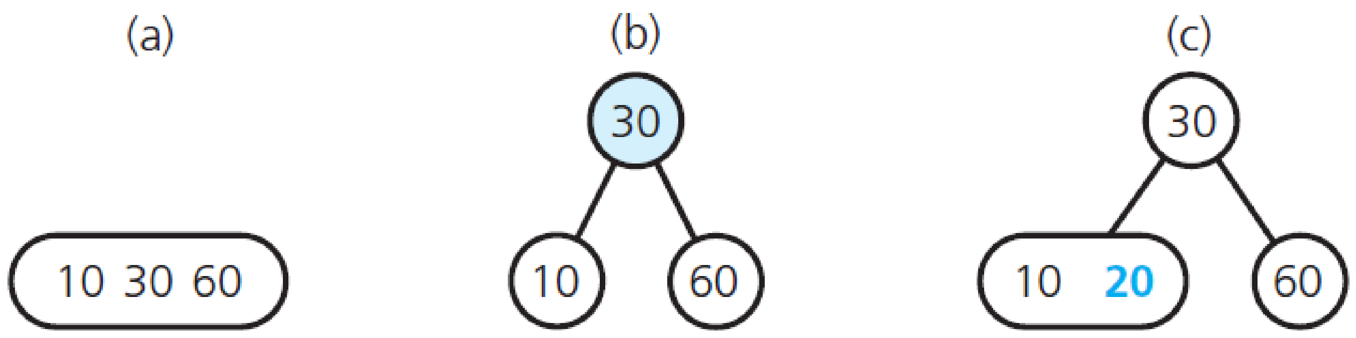
- Splitting a 4-node whose parent is a 3-node during insertion into a 2-3-4 tree, when the 4-node is a (c) right child

(c)



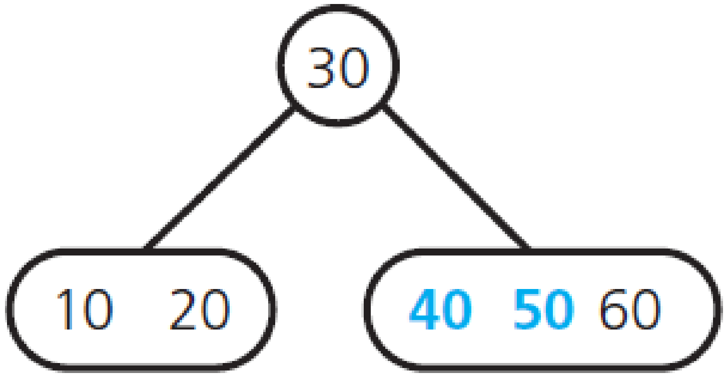
Example

- Inserting 20 into a 2-3-4 tree
  - (a) the original tree;
  - (b) after splitting the node;
  - (c) after inserting 20



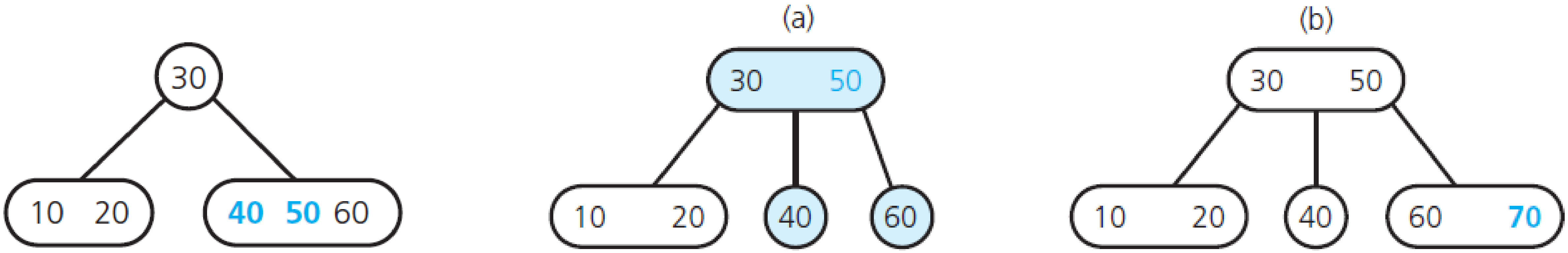
Example

- After inserting 50 and 40 into the tree



Example

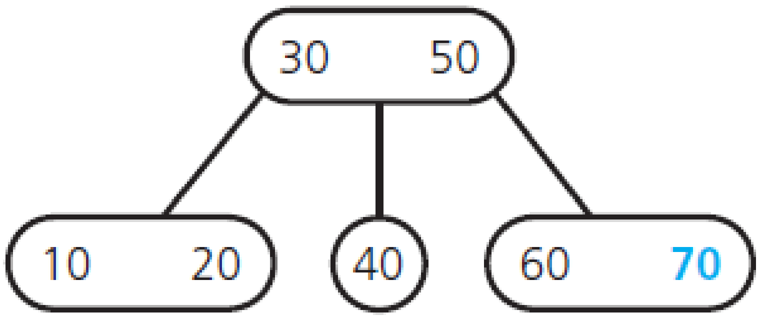
- The steps for inserting 70 into the tree
  - (a) after splitting the 4-node;
  - (b) after inserting 70

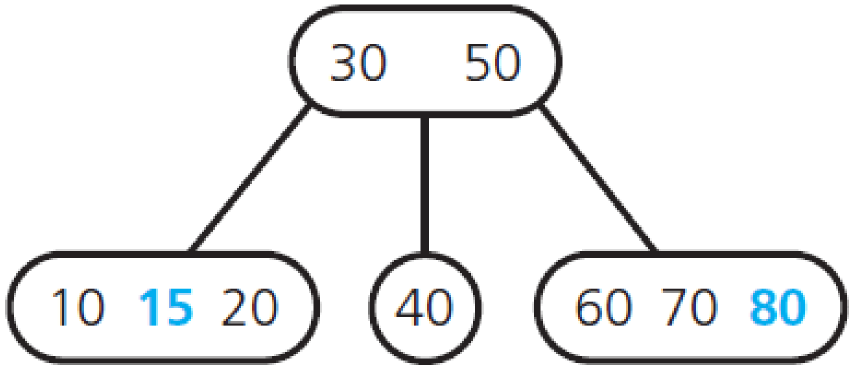


Example

- After inserting 80 and 15 into the tree

(b)

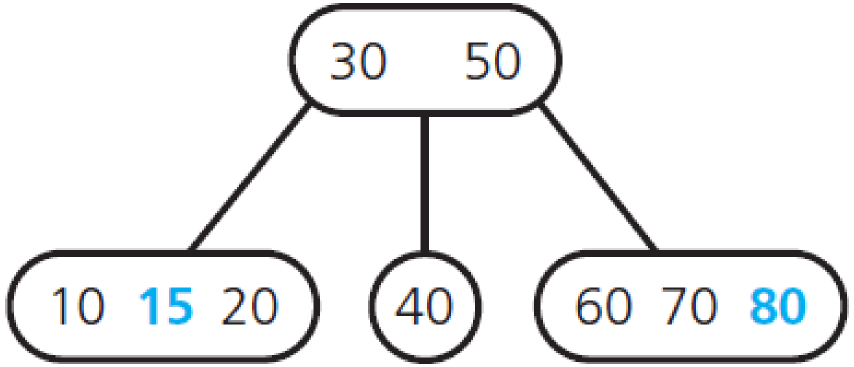




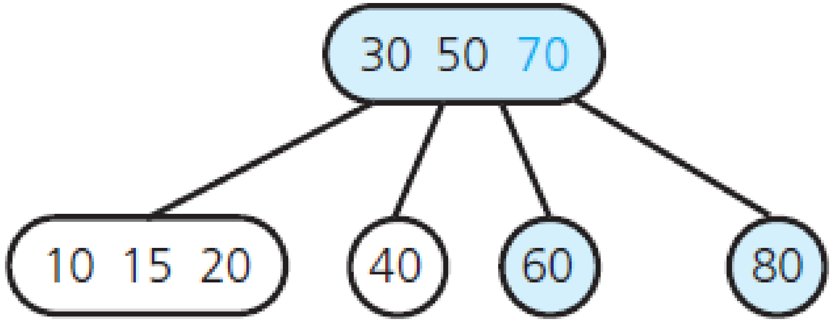
Example

fit@hcmus

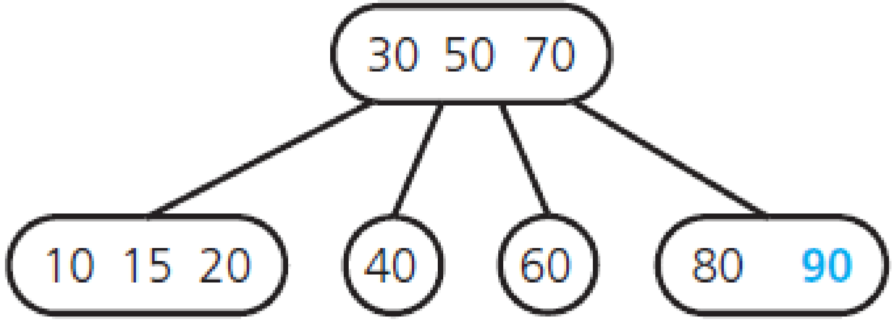
- The steps for inserting 90 into the tree



(a)

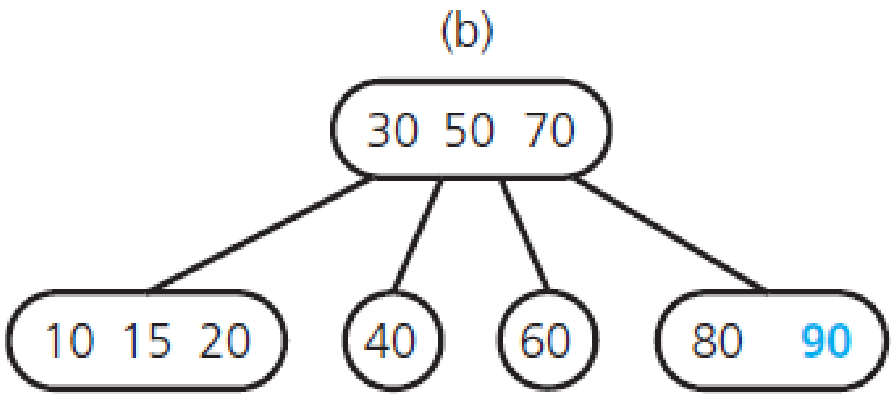


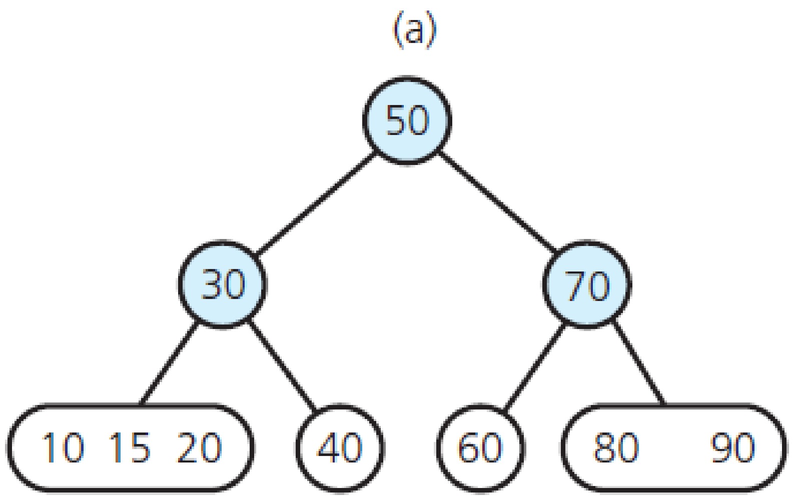
(b)

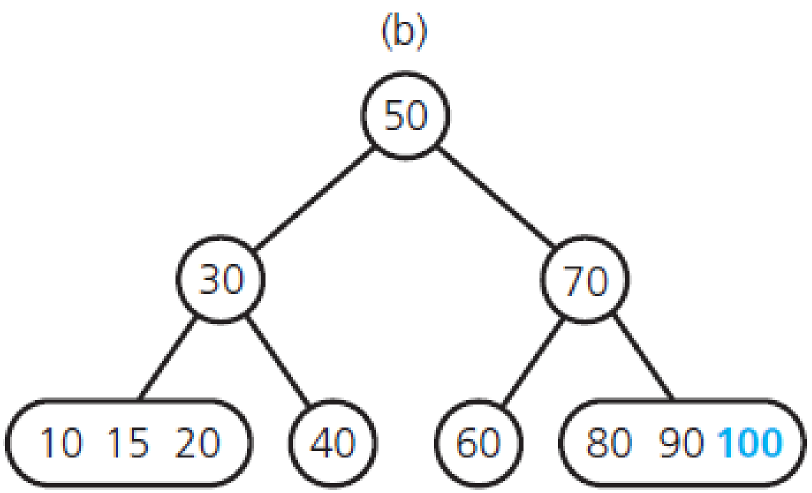


Example

- The steps for inserting 100 into the tree







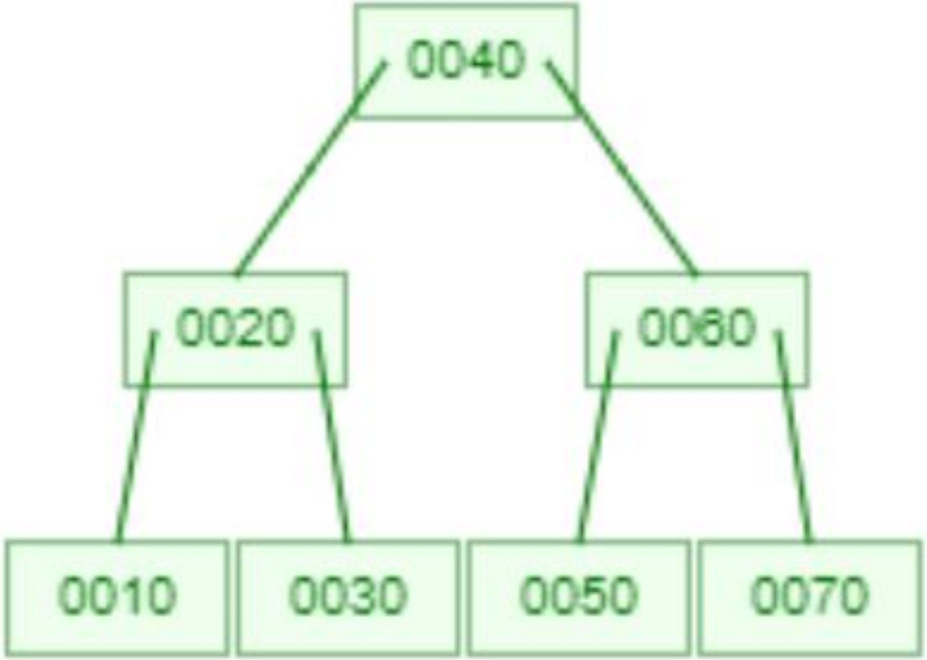
Exercise

- Insert the following numbers in the empty 2-3 tree

10, 20, 30, 40, 50, 60, 70

- Insert the following numbers in the empty 2-3 tree

  10, 20, 30, 40, 50, 60, 70



Exercise

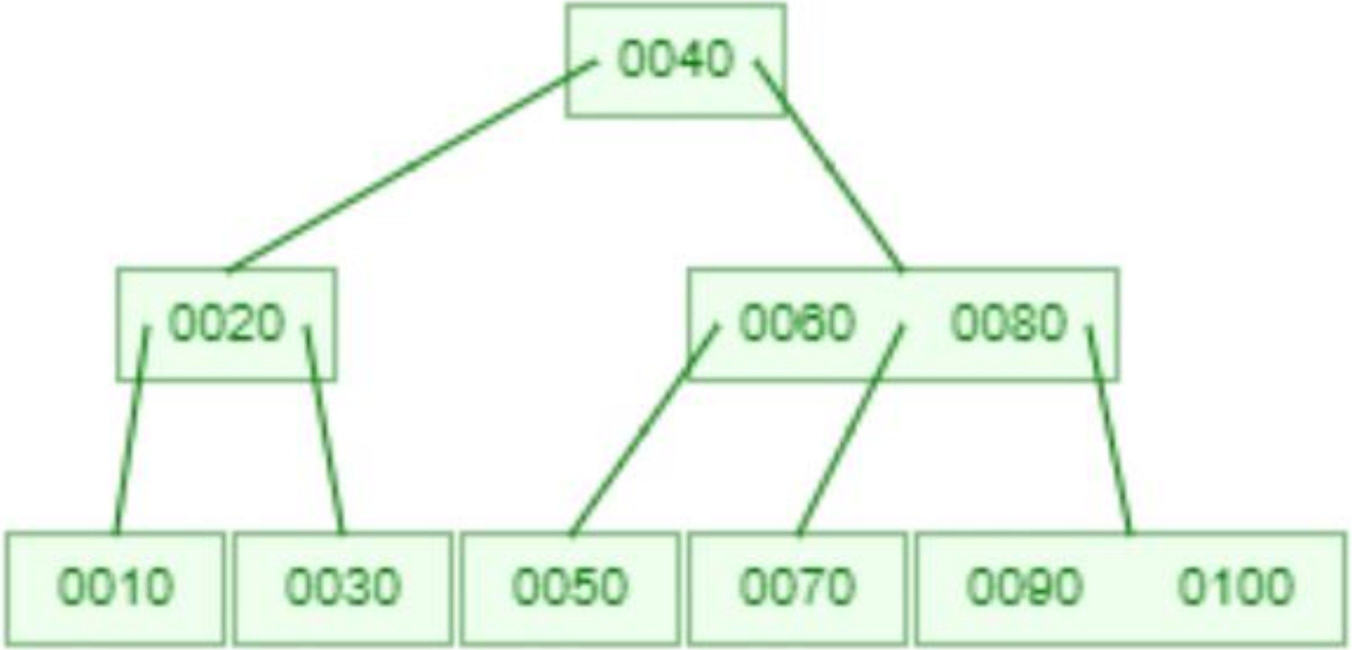
- Insert the following numbers in the empty 2-3-4 tree:

10, 20, 30, 40, 50, 60, 70, 80, 90, 100

Exercise

- Insert the following numbers in the empty 2-3-4 tree:

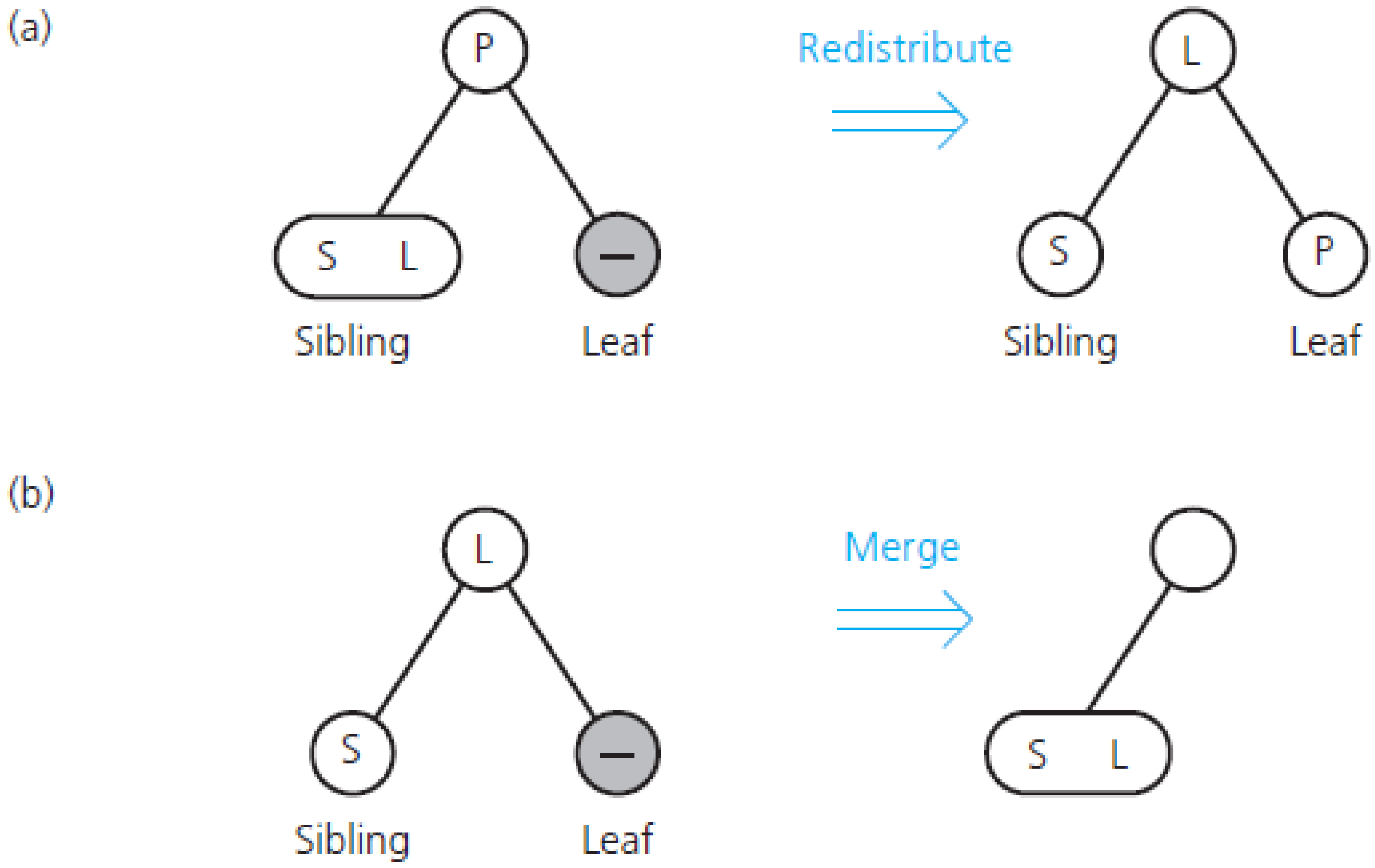
  10, 20, 30, 40, 50, 60, 70, 80, 90, 100



Deletion: 2-3 Tree

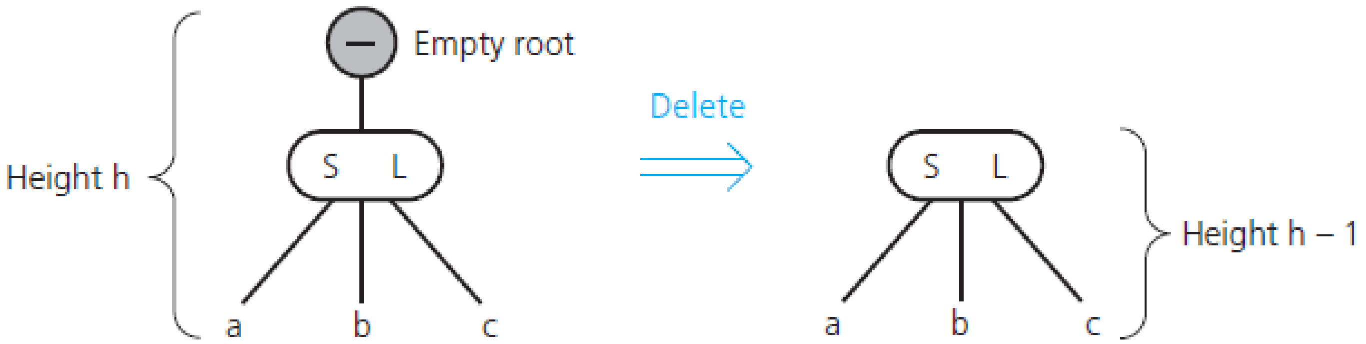
- To delete a value, it is replaced by its in-order successor and then removed.
- If a node is left with less than one data value, then two nodes must be merged together.
- If a node becomes empty after deleting a value, it is then merged with another node.

- (a) Redistributing values;
- (b) merging a leaf;



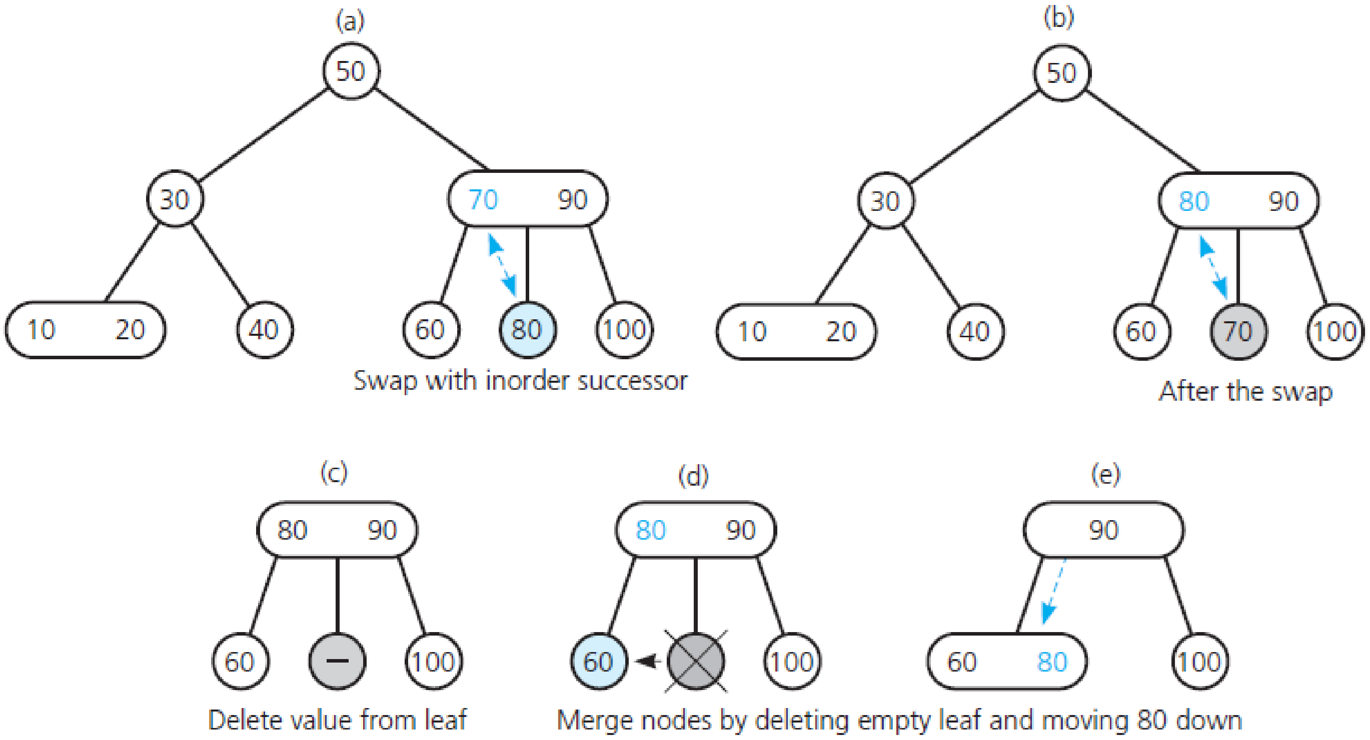
- (e) deleting the root

(e)



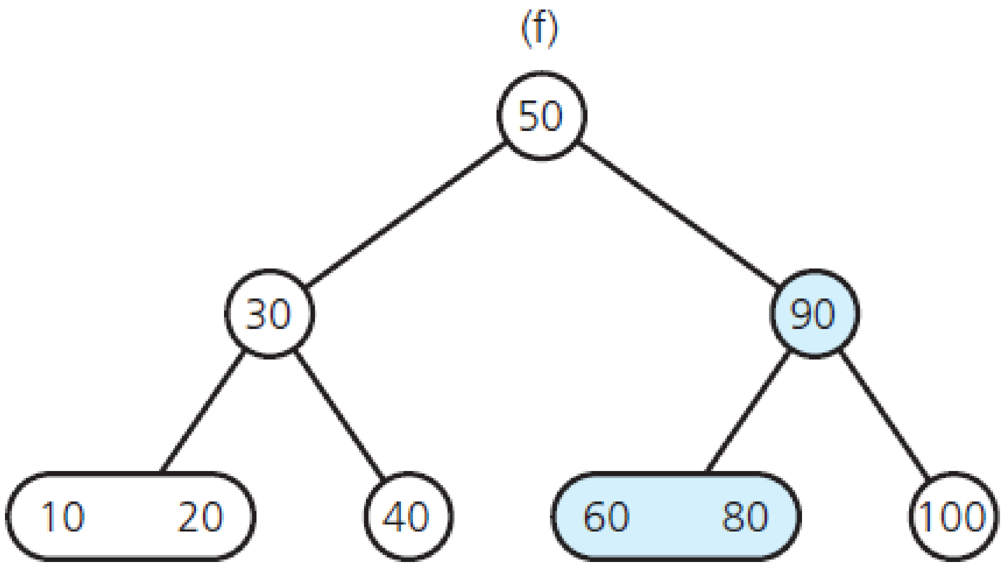
Example

- (a) A 2-3 tree;
- (b), (c), (d), (e) the steps for removing 70;



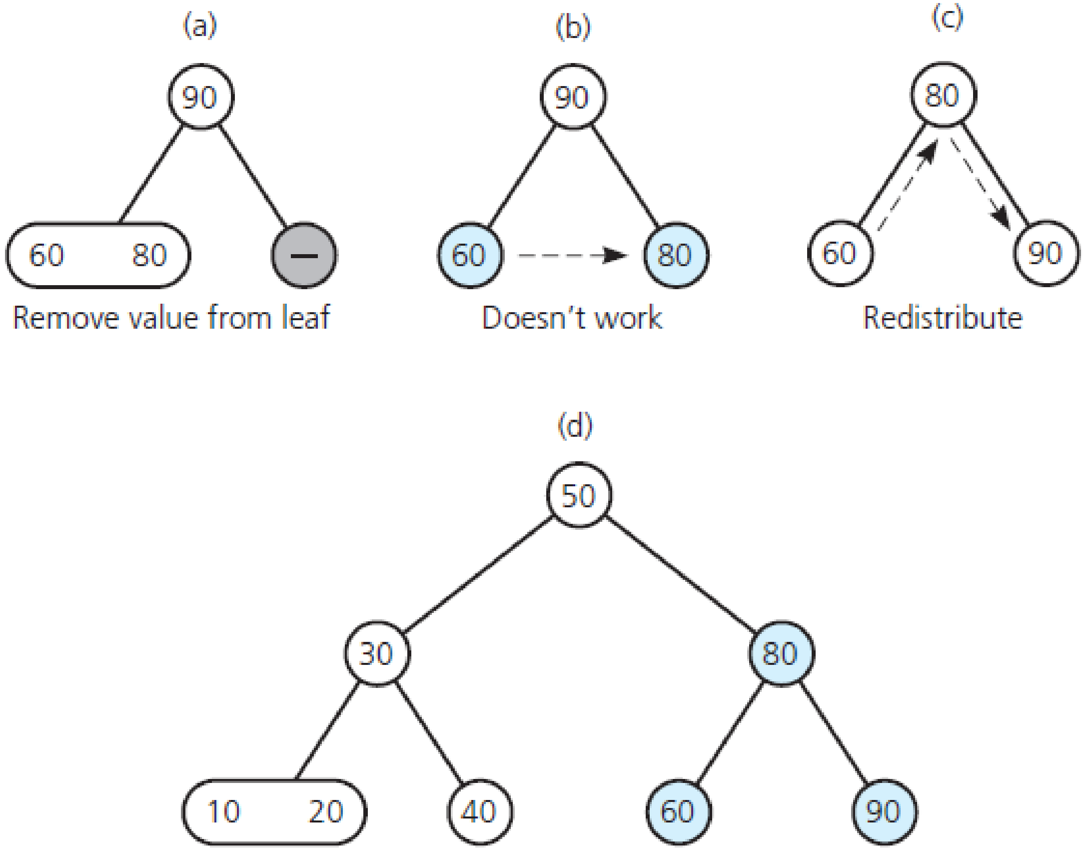
Example

- The resulting tree:



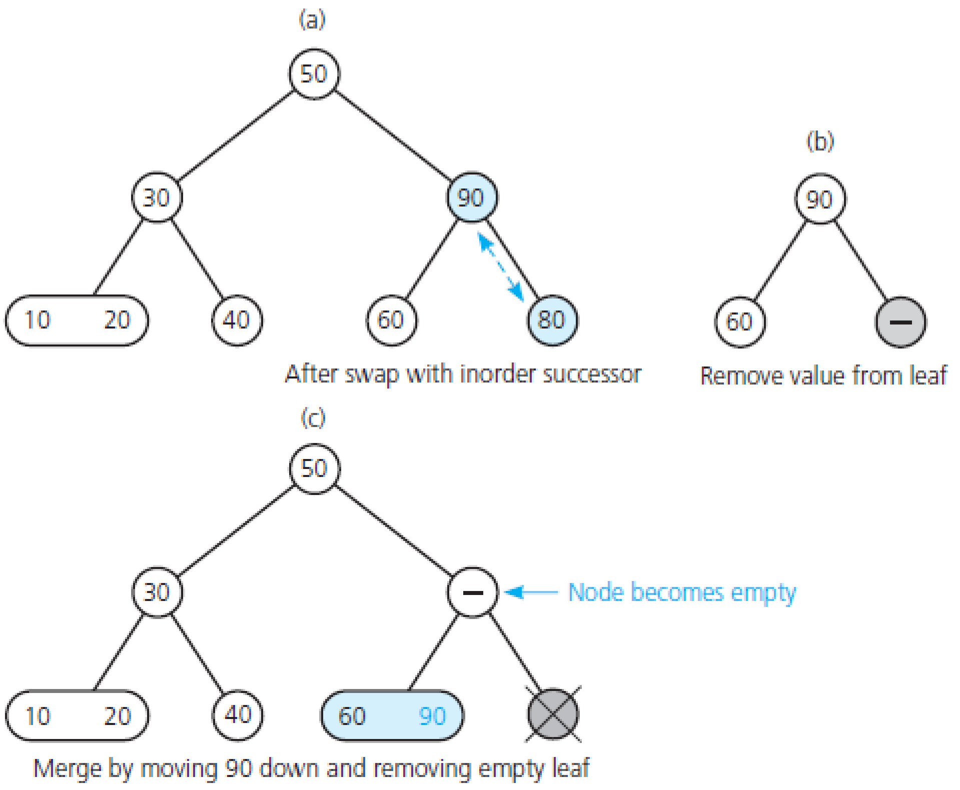
Example

- (a), (b), (c) The steps for removing 100 from the tree;
- (d) the resulting tree



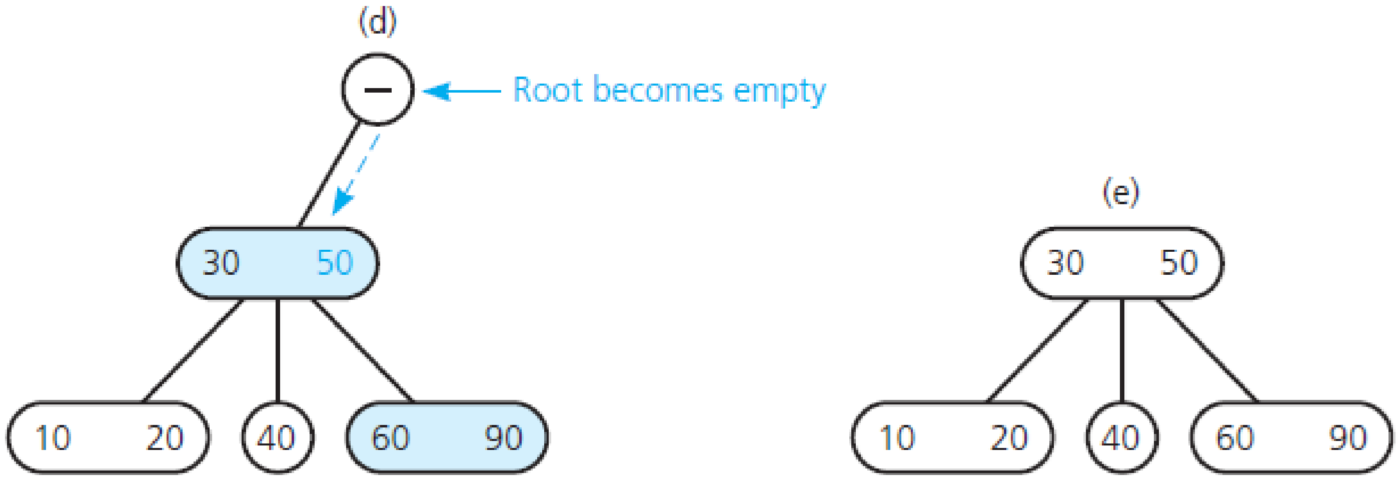
Example

- The steps for removing 80 from the tree



Example

- The steps for removing 80 from the tree



Merge: move 50 down, adopt empty leaf's child, delete empty node

Delete empty root

Deletion: 2-3-4 Tree

- Removal algorithm has same beginning as removal algorithm for a 2-3 tree
- Locate the node n that contains the item I you want to remove.
- Find I 's in-order successor and swap it with I so that the removal will always be at a leaf.
- If leaf is either a 3-node or a 4-node, remove I .

Practice

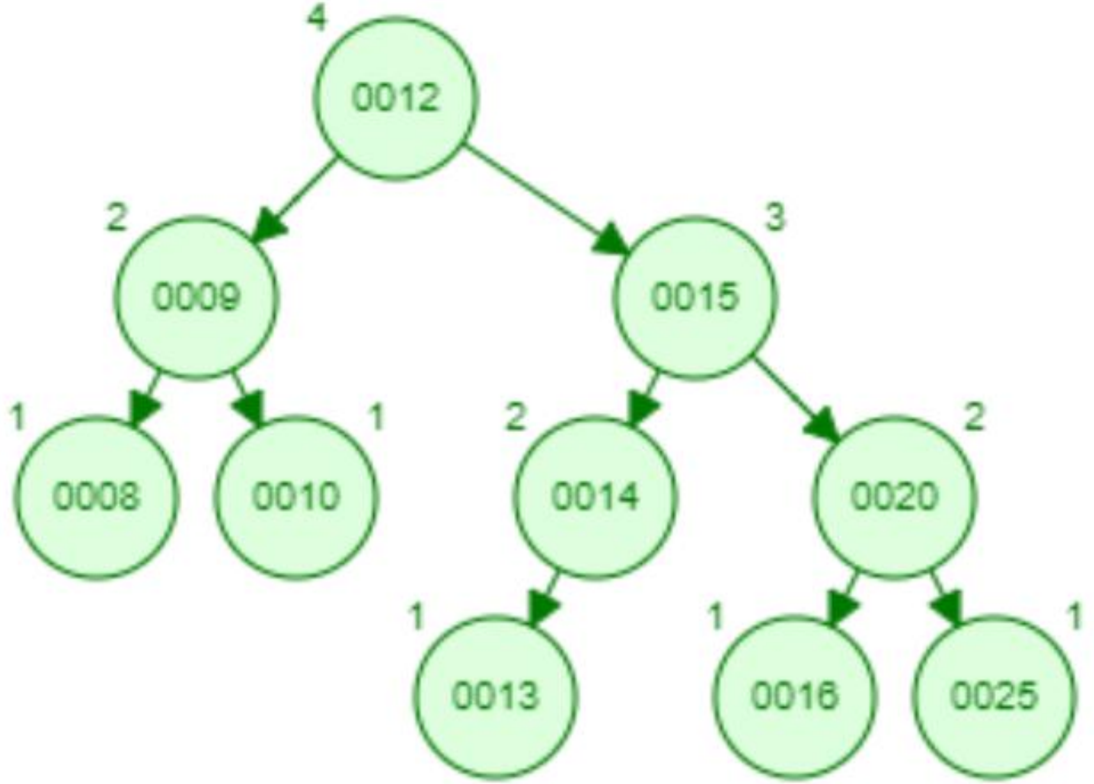
- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

15 10 12 8 9 14 13 20 25 16

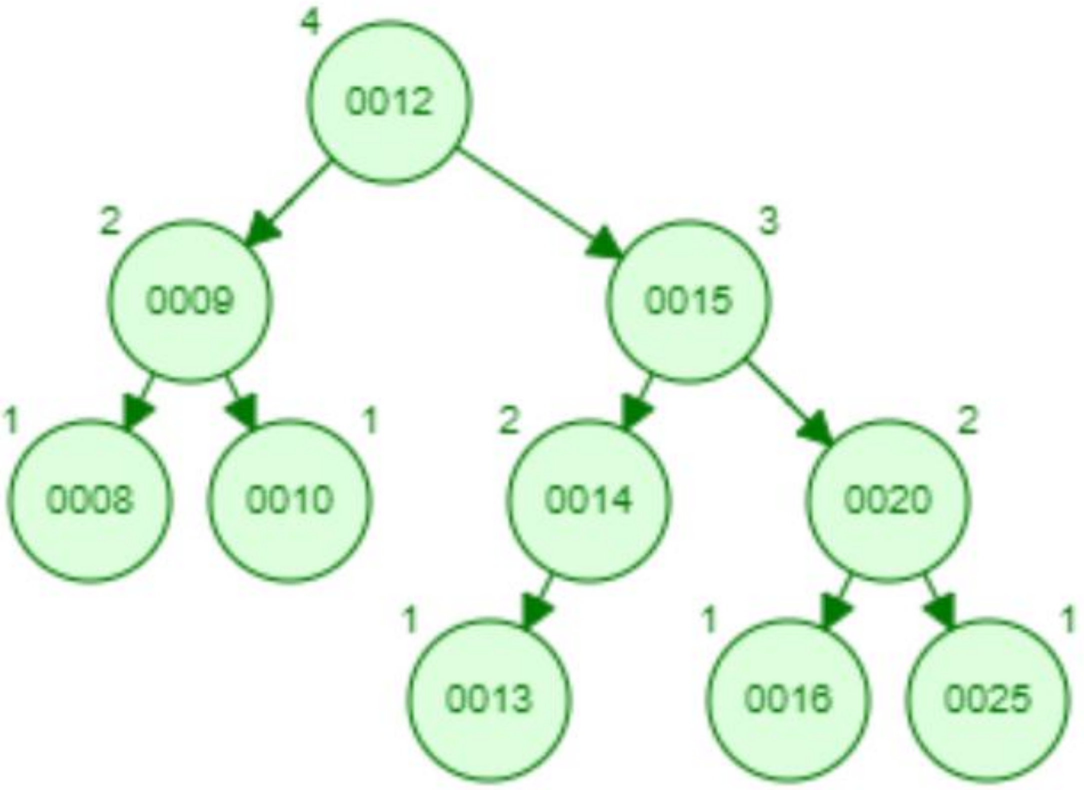
Practice

- Beginning with an empty AVL tree, perform step-by-step the insertion of the following values in the order given

15 10 12 8 9 14 13 20 25 16

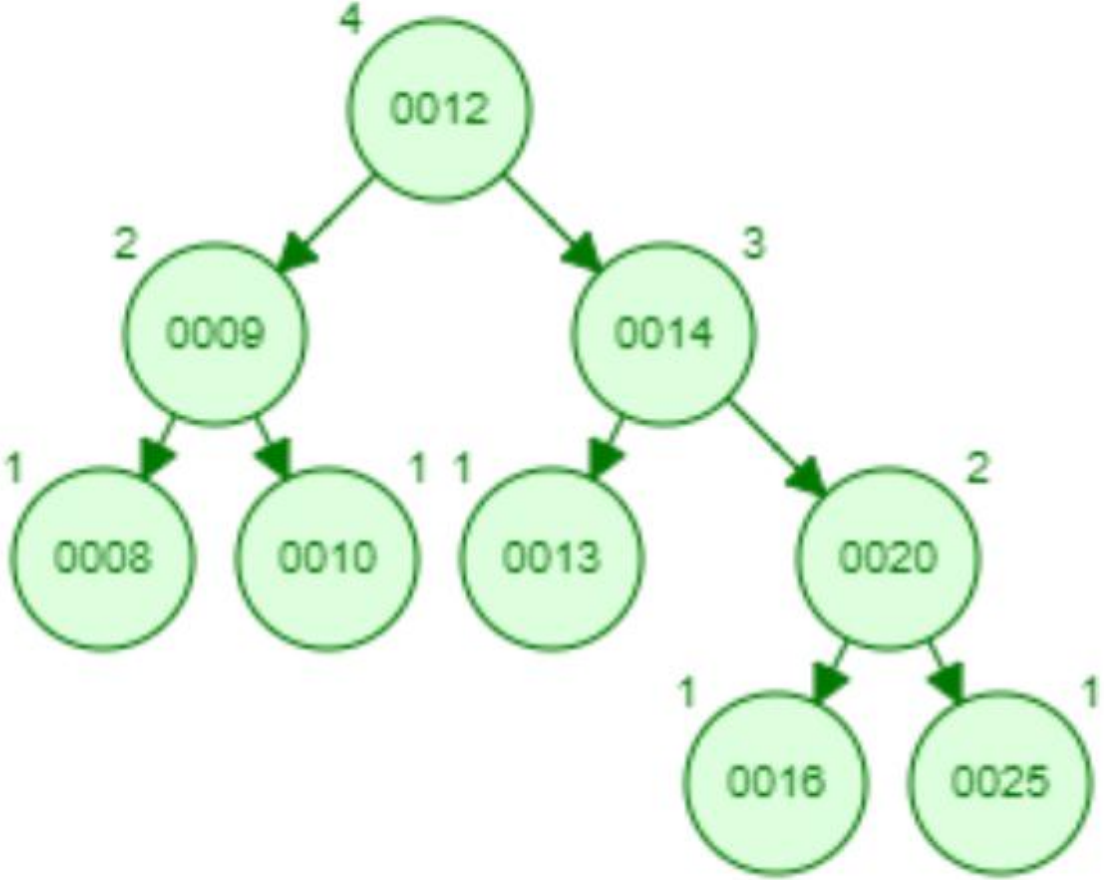


- From the following tree, delete Node 15

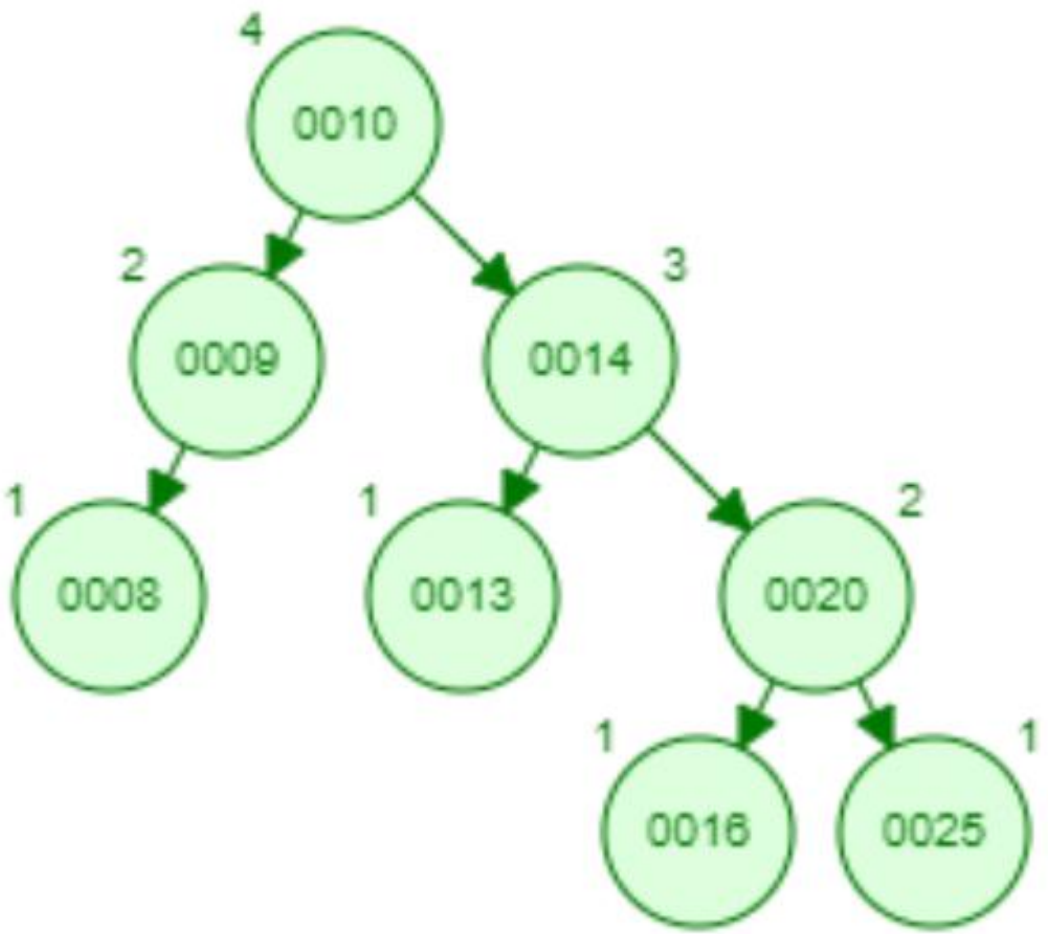


Practice

- From the following tree, delete Node 12

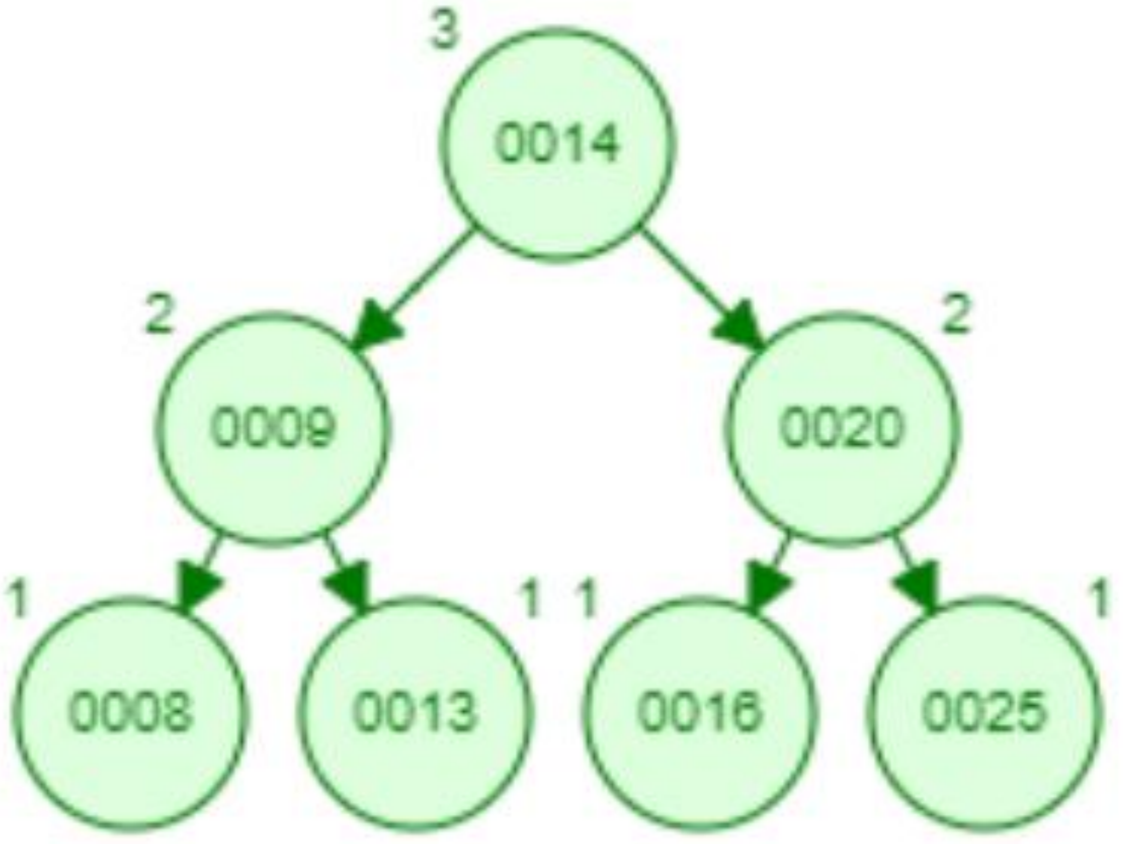


- From the following tree, delete Node 10



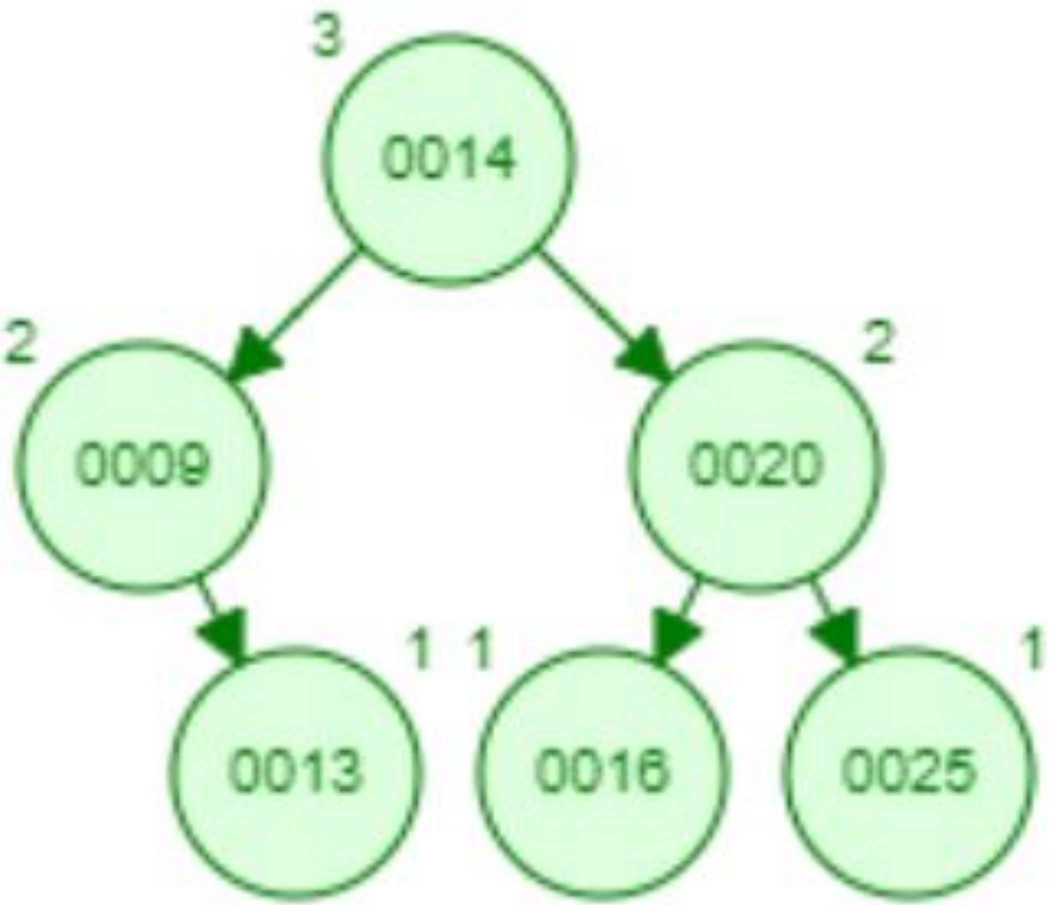
Practice

- From the following tree, delete Node 8



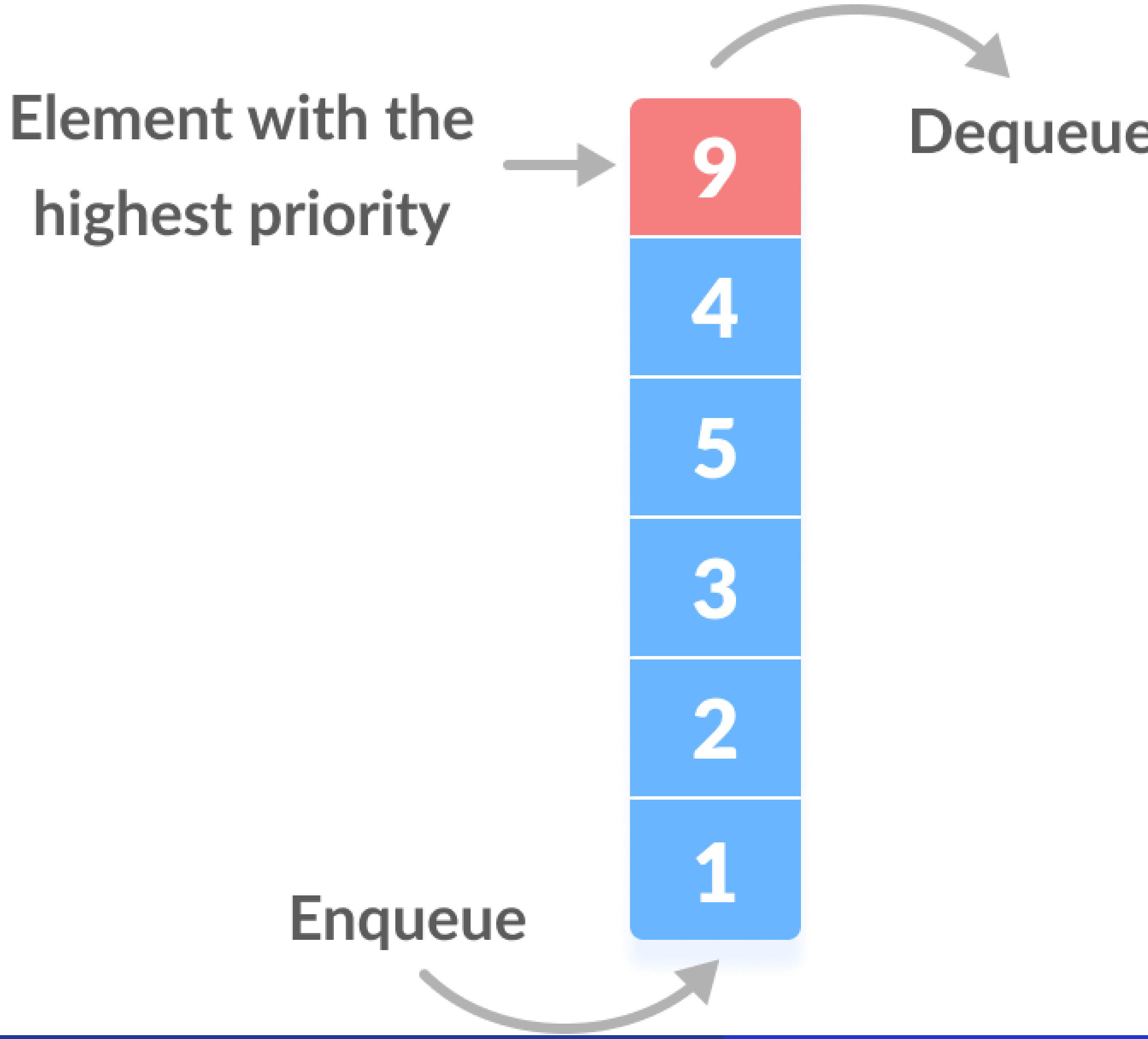
Practice

fit@hcmus

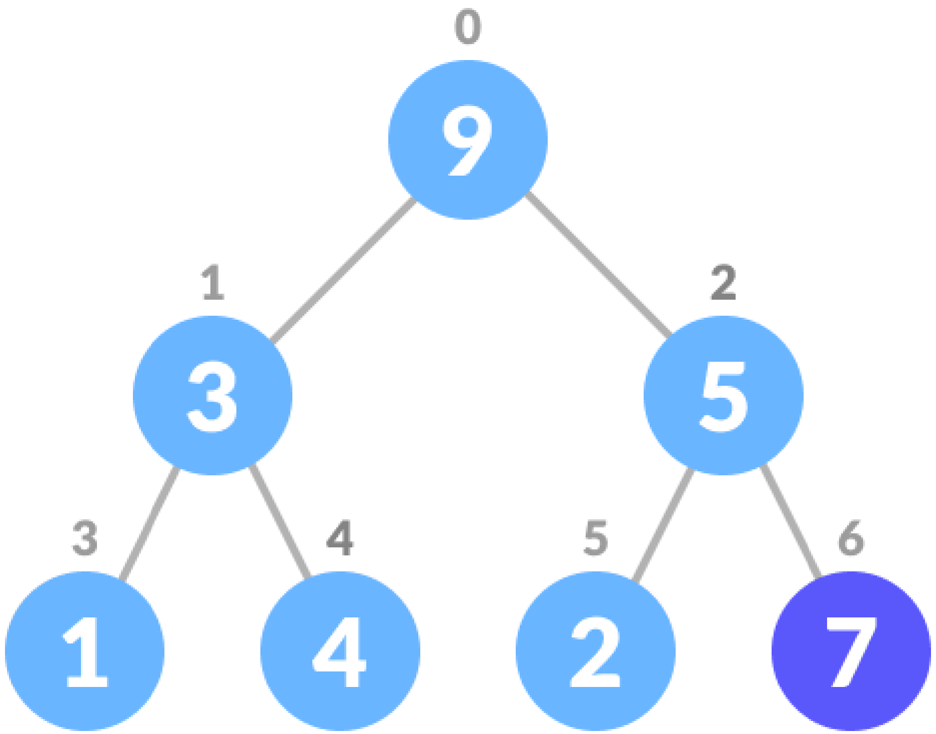


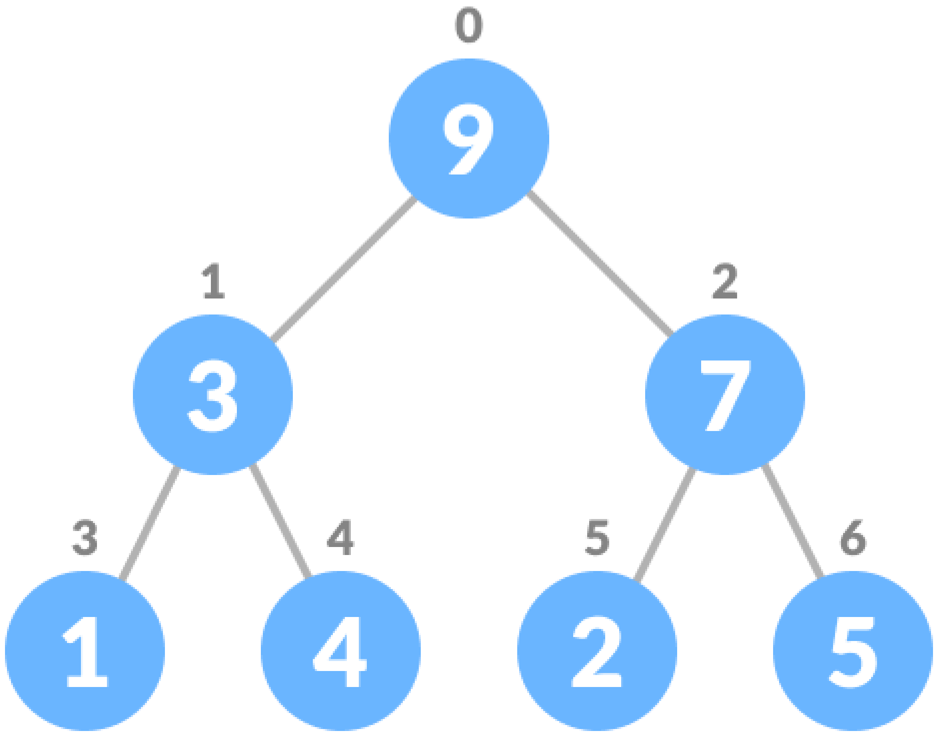
Priority Queue

- A priority queue is a special type of queue in which each element is associated with a priority value
- That is, higher priority elements are served first
- Priority queue can be implemented using an array, a linked list, a heap data structure, or a **binary search tree**

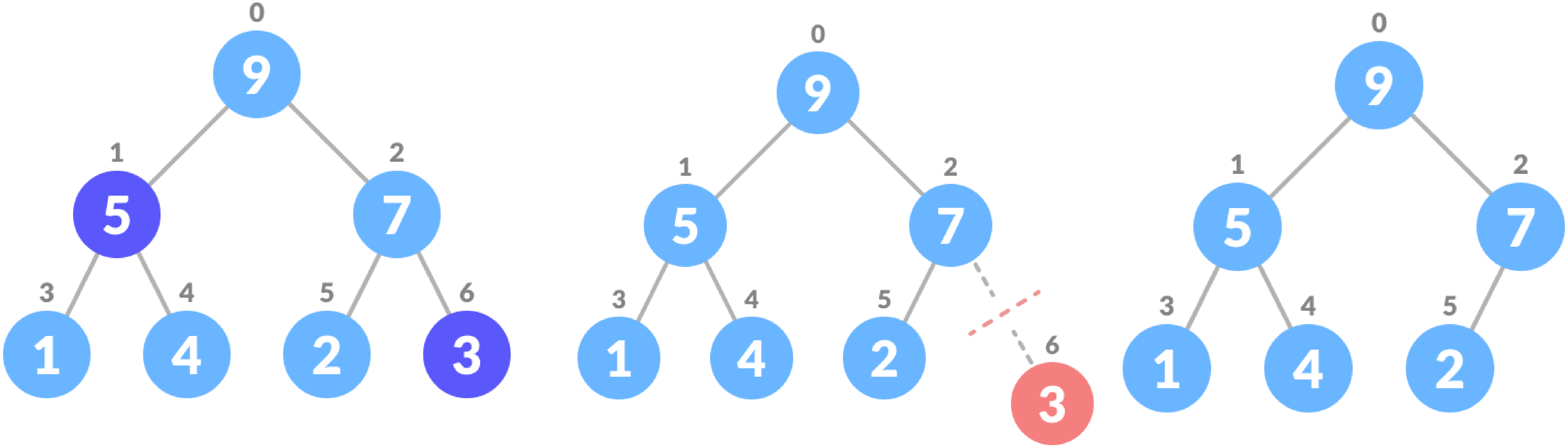


- Inserting an Element into the Priority Queue
  - Insert the new element at the end of the tree
  - Heapify the tree

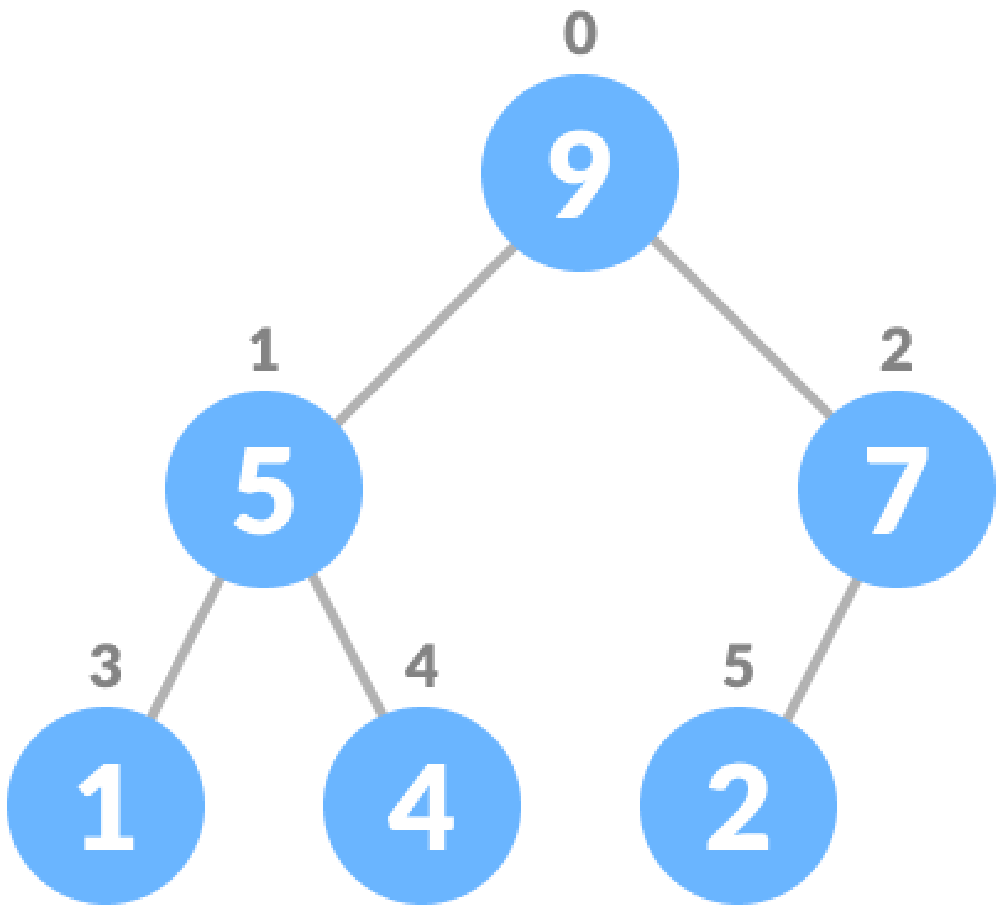




- Deleting an Element from the Priority Queue
  - Swap element to be deleted with the last element
  - Remove the last element
  - Heapify the tree



- Peeking from the Priority Queue
  - returns the maximum element from Max Heap



THANK YOU

for YOUR ATTENTION