VE281

Data Structures and Algorithms

2-3 Trees and Graphs

Review

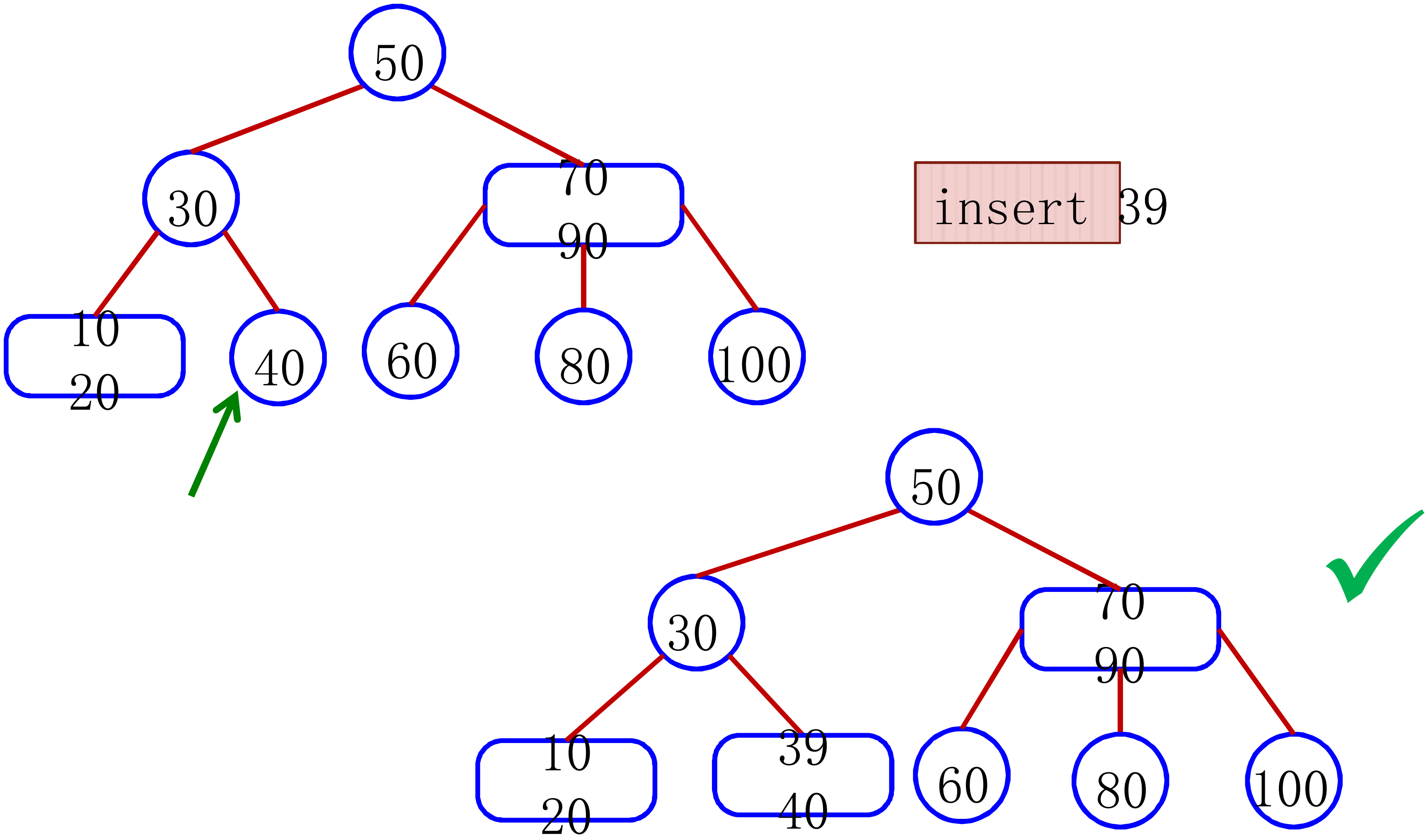
- Trie: Insertion, removal, and time-complexity
- M-way search tree: A generalization of binary search tree
- 2-3 Tree
  - A balanced 3-way search tree with all leaves at the same level.
  - To store N keys, height $h = \Theta(\log N)$
  - In-order traversal and search
- 2-3 Tree: Insertion
  - Sometimes we will make an internal node have three keys. Then we **split** the node and move the middle key up to its parent.

Outline

- 2-3 Tree: Insertion
- 2-3 Tree: Removal
- Graphs

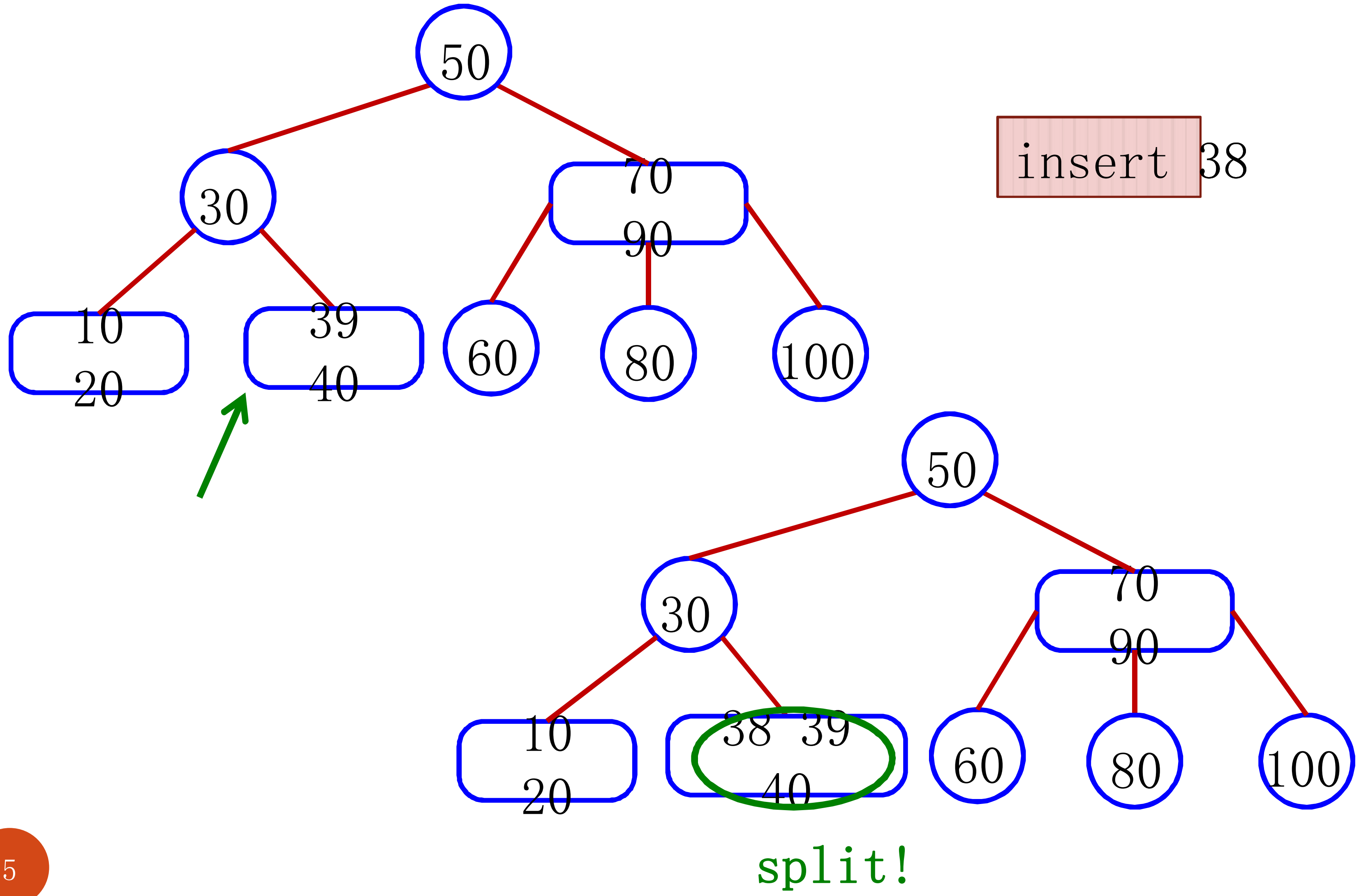
2-3 Trees Insertion

Example



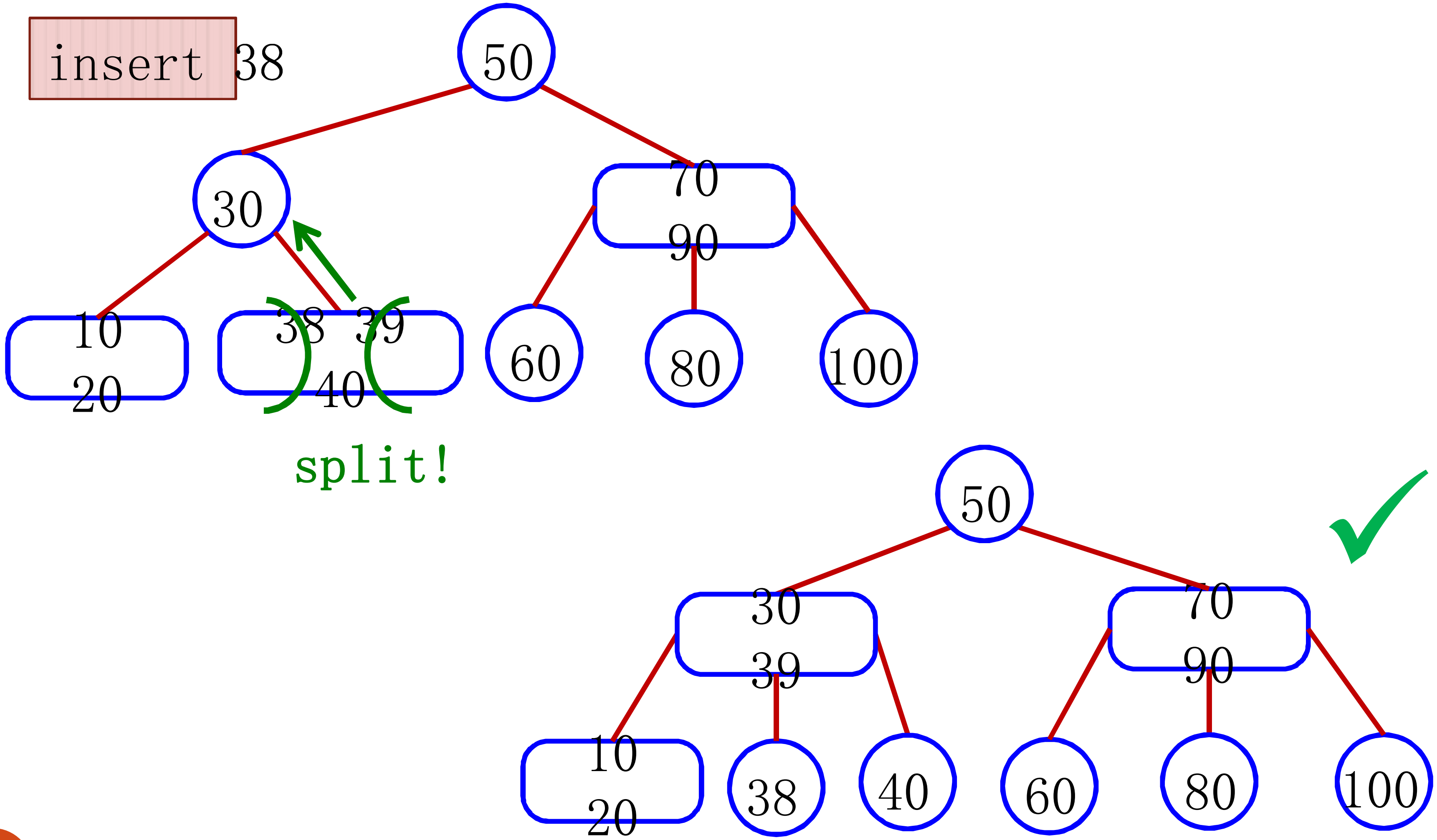
2-3 Trees Insertion

Example



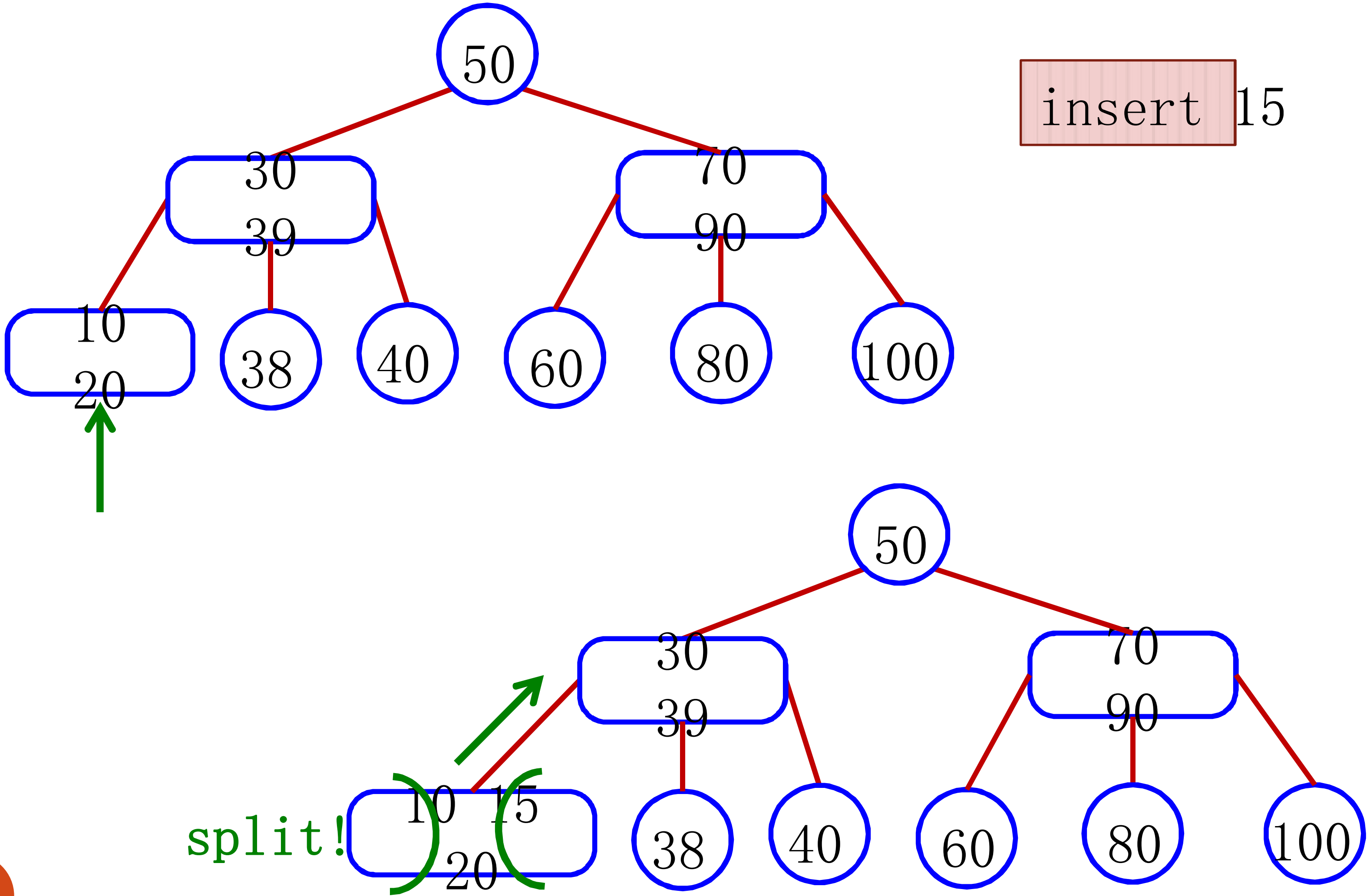
2-3 Trees Insertion

Example



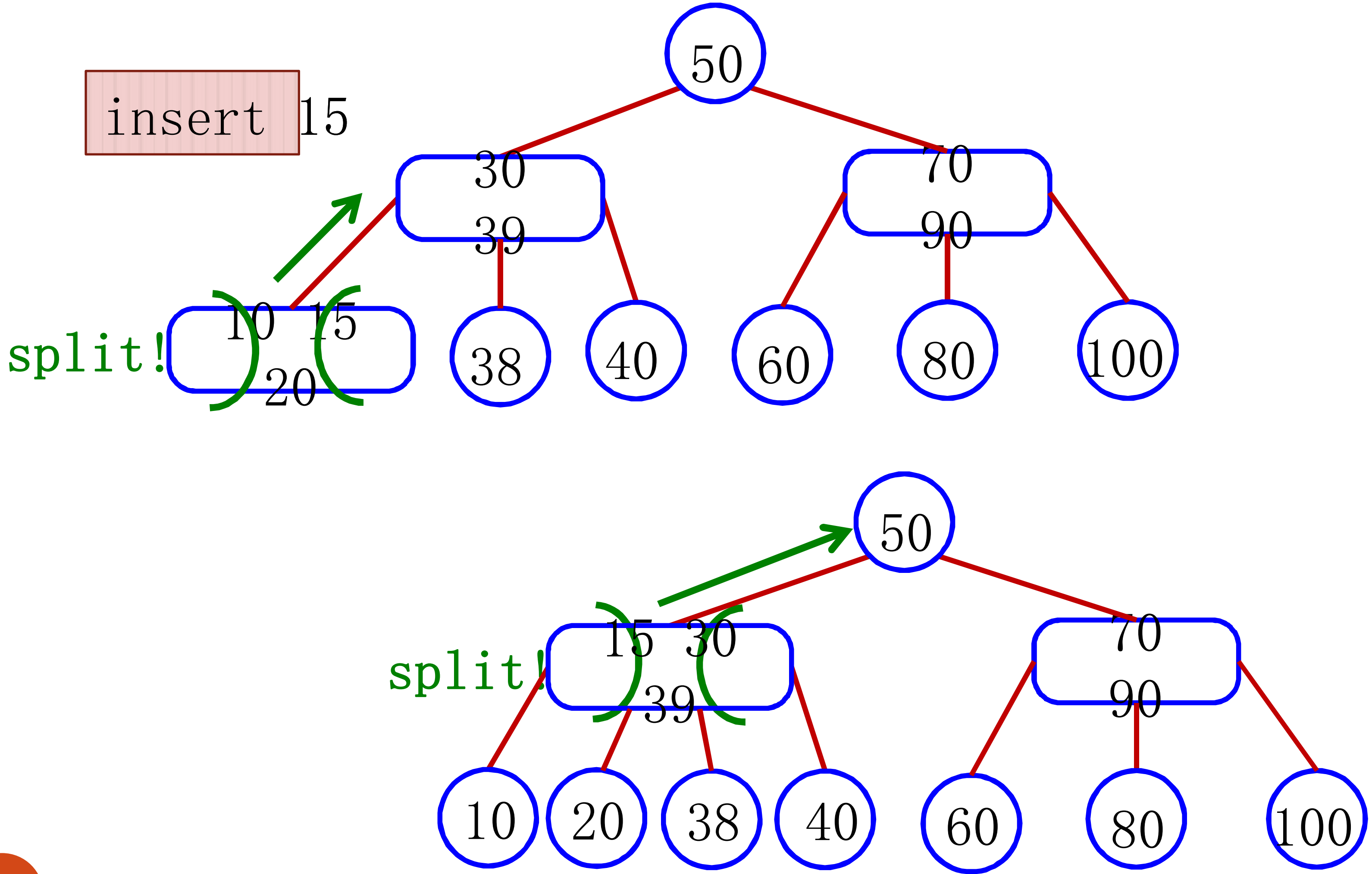
2-3 Trees Insertion

Example



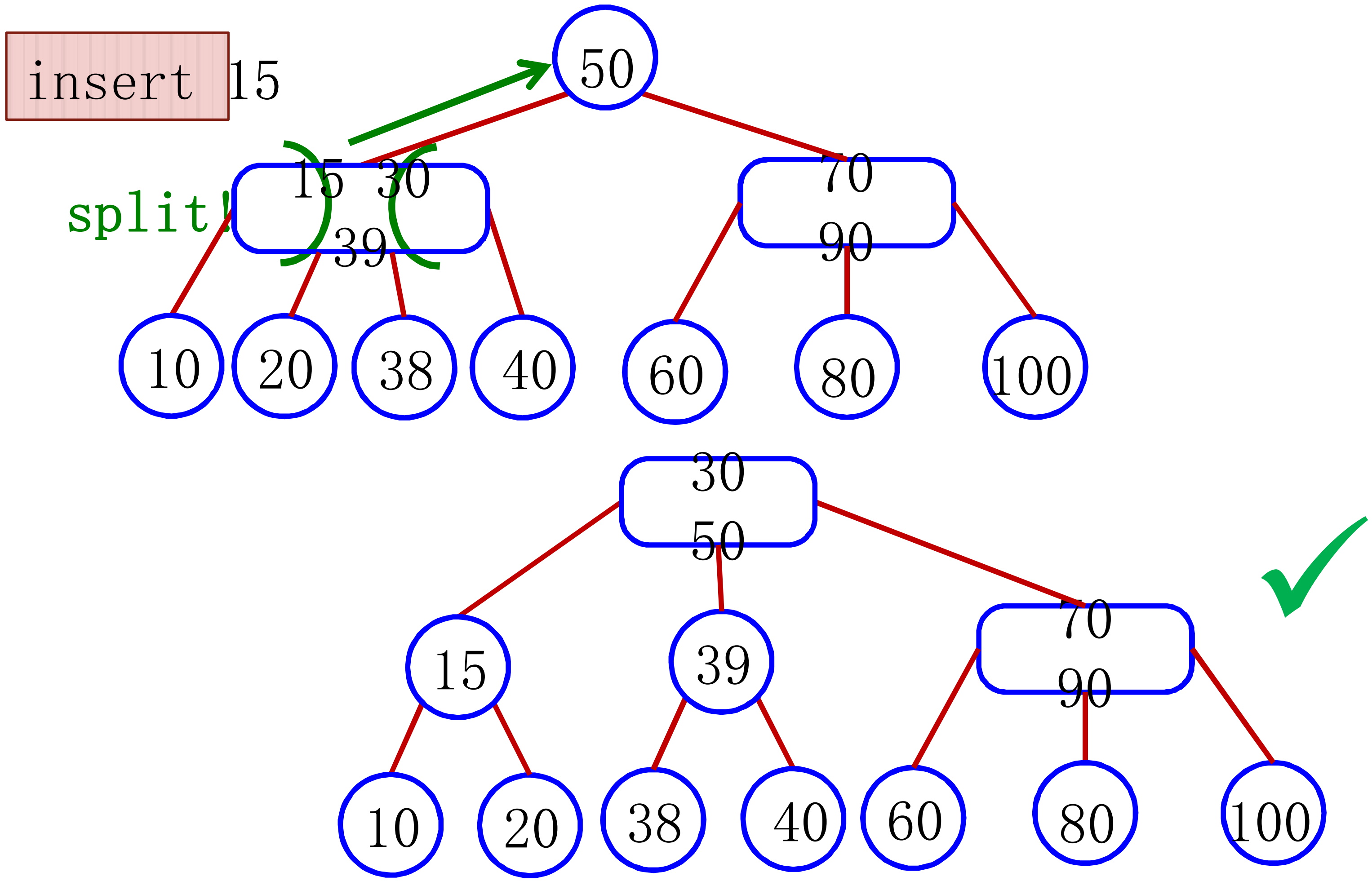
2-3 Trees Insertion

Example



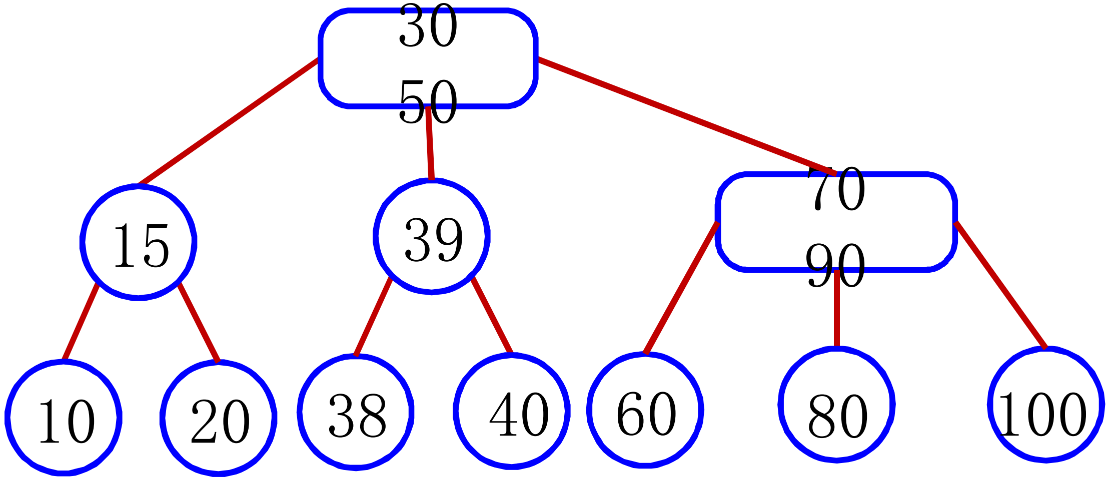
2-3 Trees Insertion

Example

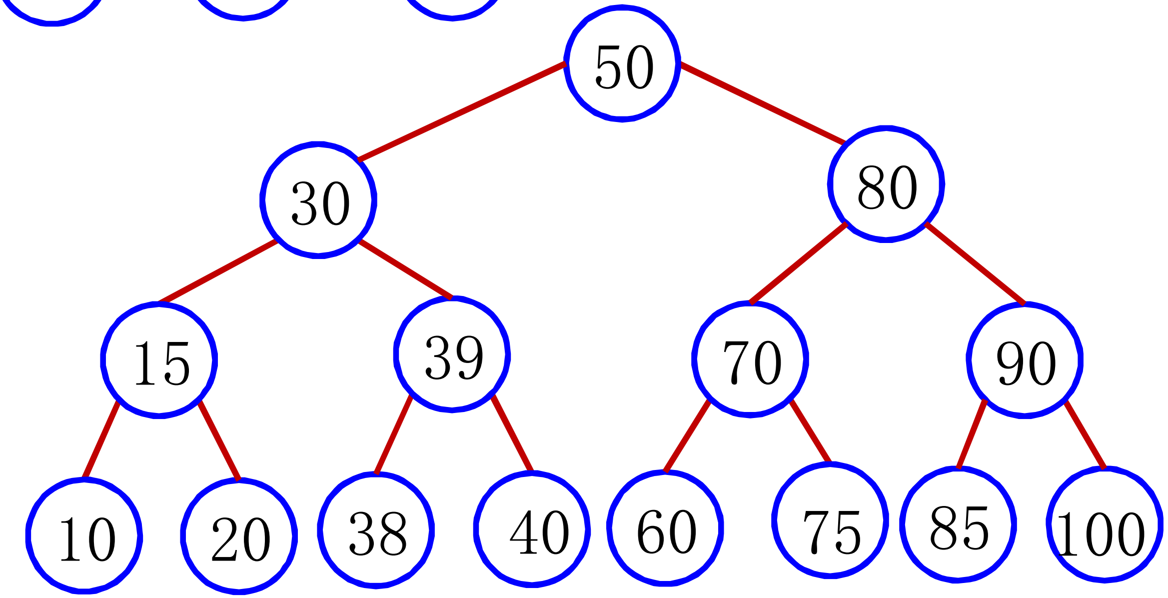


2-3 Trees Insertion

Exercise



insert 75, 85



2-3 Tree Insertion

Procedure

1. Search for leaf where key belongs.
2. If leaf is a 2-node, add key to leaf.
3. If leaf is a 3-node, adding the new key makes it an invalid node with 3 keys. Split the invalid node into two 2-nodes with the smallest and largest keys and pass the middle key up to parent.
4. If parent is a 2-node, add the child's middle key with the two new children; else split parent by Step 3 above, move the middle key to its parent, and repeat this step.
5. If there's no parent, create a new root (increase tree height by 1).

2-3 Tree Insertion

Summary

- Whereas a BST increases height by extending a single path, a 2-3 tree increases height **globally** by **raising** the root.

- Therefore, all of the leaves of a 2-3 tree are at the same level.
  - The 2-3 tree is always balanced.

- What is the worst case time complexity?
  - $O(\log N)$

Outline

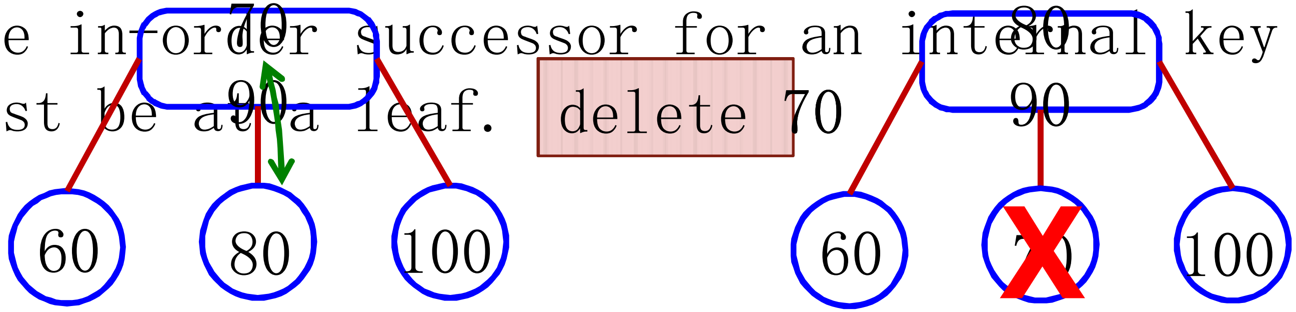
- 2-3 Tree: Insertion
- 2-3 Tree: Removal
- Graphs

2-3 Trees

Removal

- Removal process usually begins with a leaf, but the key to be deleted may not be at a leaf.

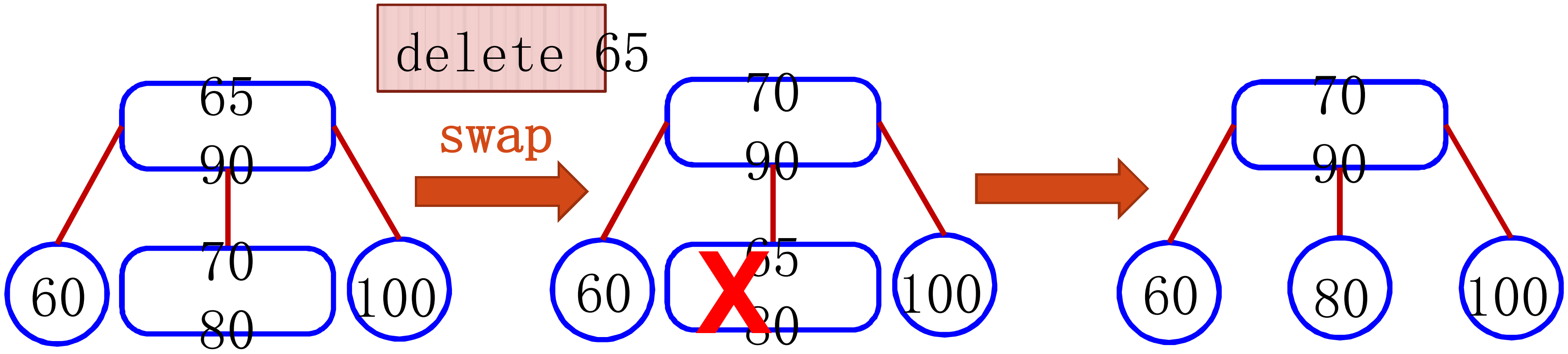
- Swap the key at an internal node with its **in-order successor**, i.e., the smallest key in the subtree on the right of the key.
  - The in-order successor for an internal key must be at a leaf.



2-3 Trees

Removal

- If after swapping, the key to be deleted is in a 3-node, remove that key. We are done.

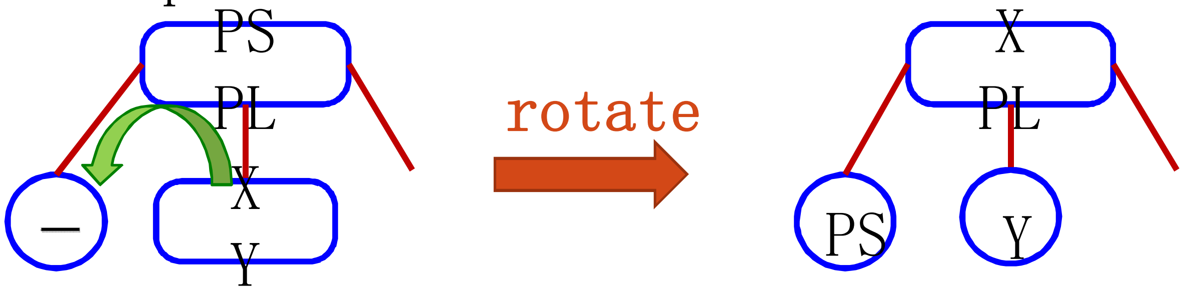


- If the key is in a 2-node, removing the key violates the 2-3 tree property.
- We need to restore the property by either **rotating** keys or **merging** nodes.

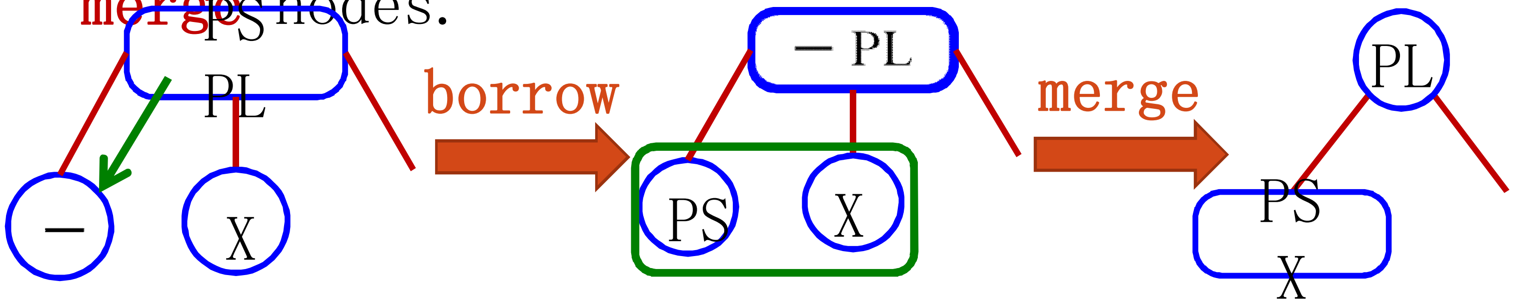
2-3 Trees

Removal

- Main idea:
  - Rotate the keys in the adjacent sibling node and the parent node.



  - Or, borrow a key from the parent and then merge nodes.



2-3 Trees

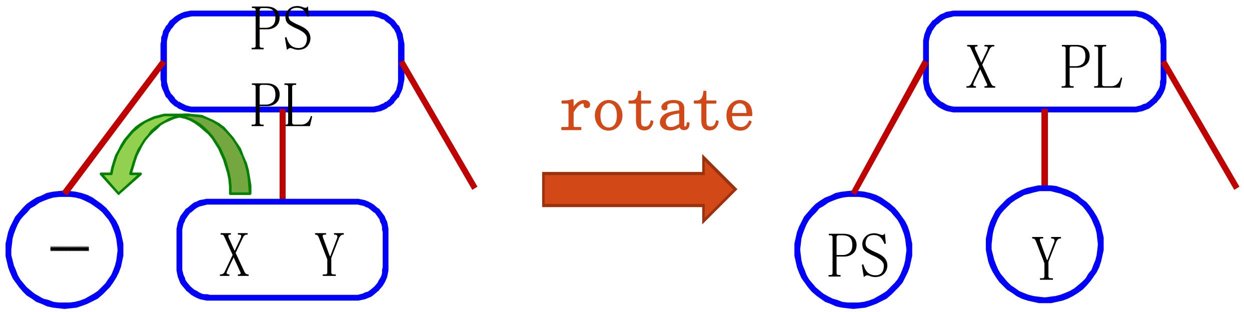
Removal

- Distinguish cases based on whether the **parent** of the empty leaf is a 2-node or a 3-node.
- When parent is a 3-node, we have 3 subcases:
  - (1) left leaf empty, (2) center leaf empty, and (3) right leaf empty.
- When parent is a 2-node, we have 2 subcases:
  - (1) left leaf empty and (2) right leaf empty.

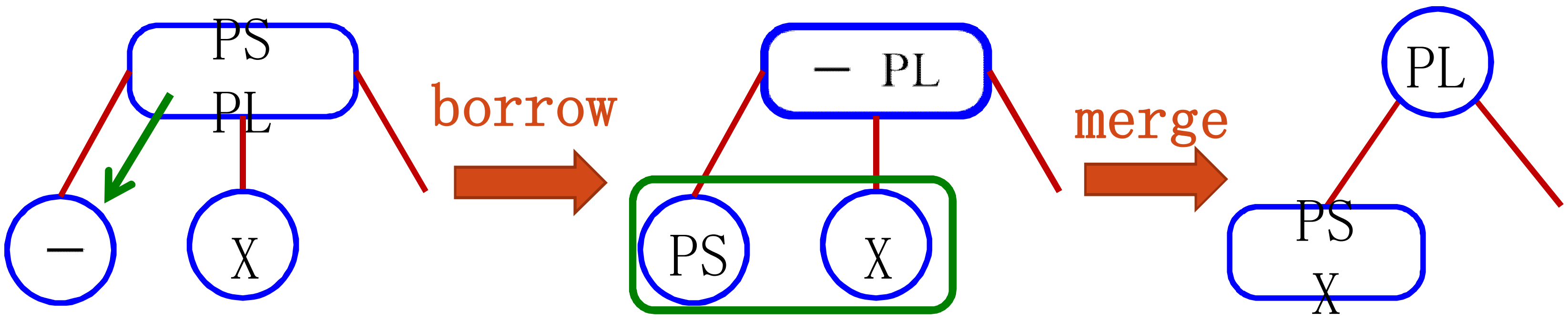
- For each of the above subcases, further distinguish based on whether the **sibling** leaf is a 2-node or a 3-node.

3-Node Parent Case 1: Left Leaf Empty

- Subcase 1: Center sibling is a 3-node.

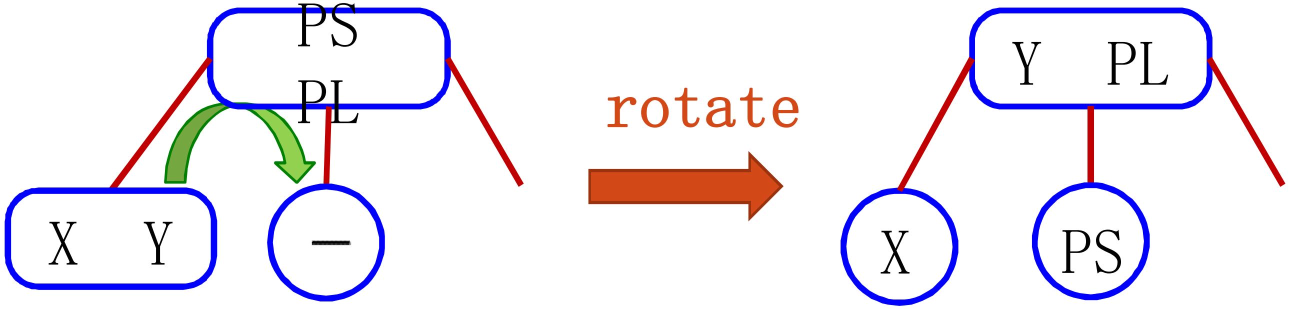


- Subcase 2: Center sibling is a 2-node.

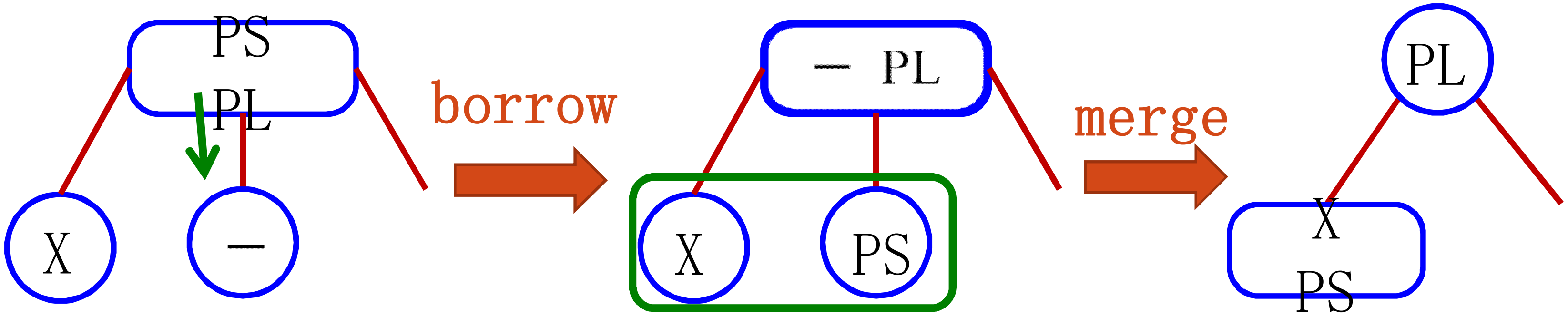


3-Node Parent Case 2: Center Leaf Empty

- Subcase 1: Left sibling is a 3-node.

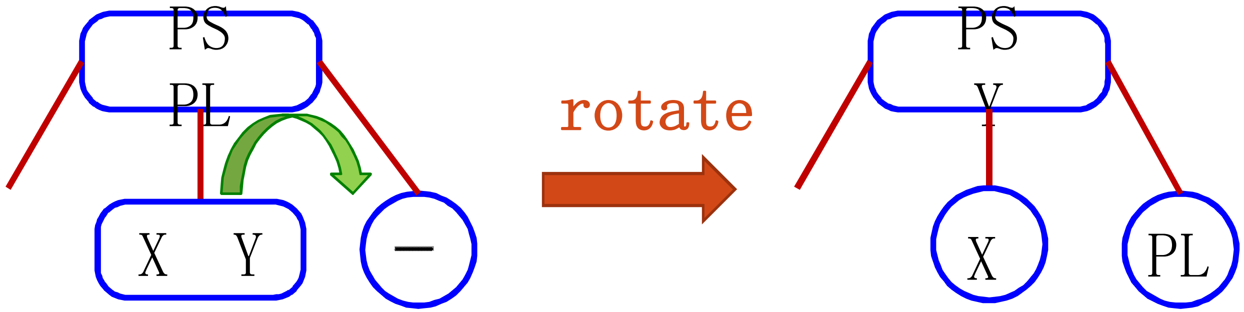


- Subcase 2: Left sibling is a 2-node.

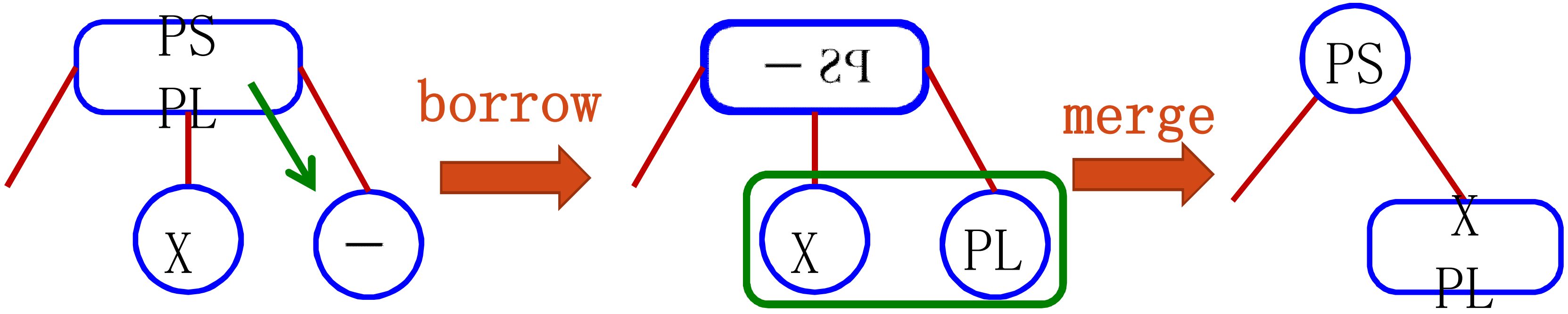


3-Node Parent Case 3: Right Leaf Empty

- Subcase 1: Center sibling is a 3-node.

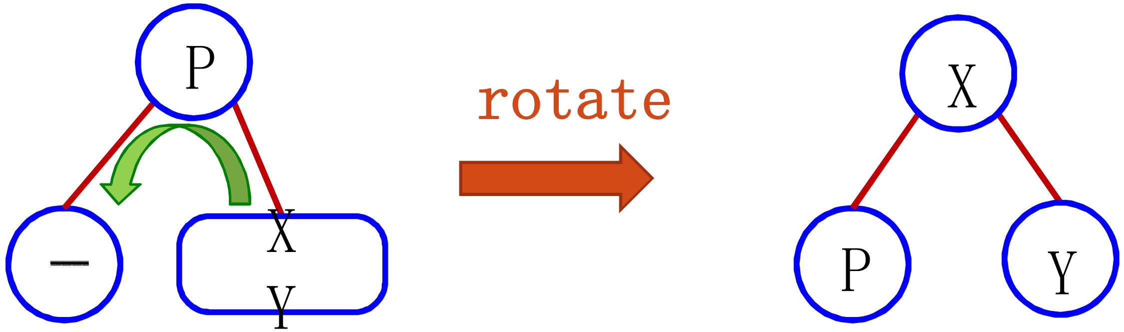


- Subcase 2: Center sibling is a 2-node.

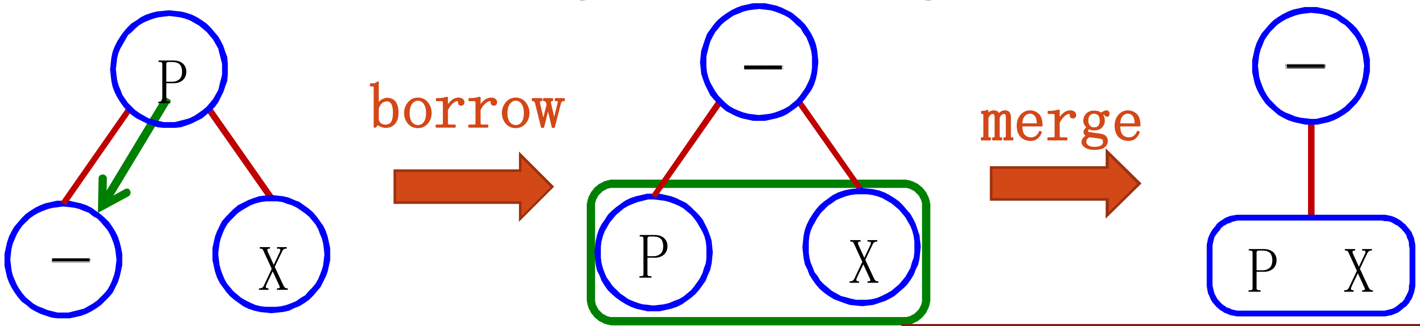


2-Node Parent Case 1: Left Leaf Empty

- Subcase 1: Right sibling is a 3-node.



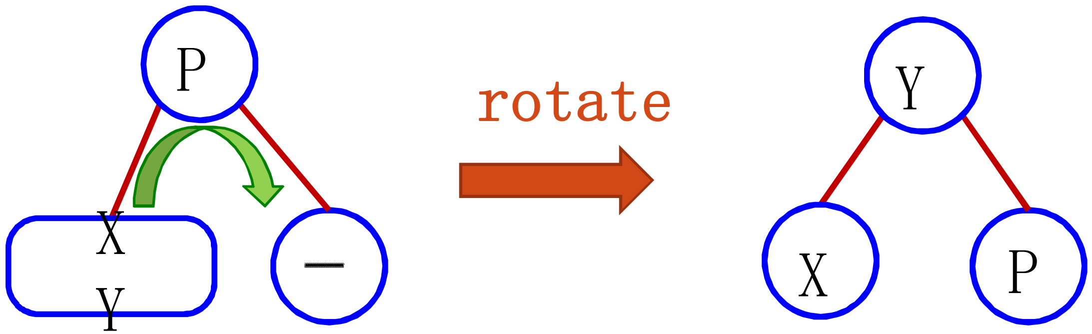
- Subcase 2: Right sibling is a 2-node.



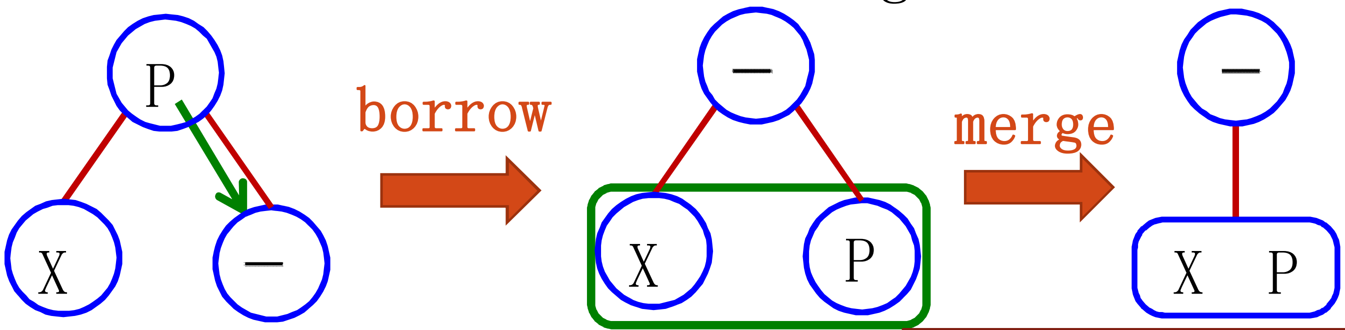
Now the parent becomes empty.
Further need to handle

2-Node Parent Case 2: Right Leaf Empty

- Subcase 1: Left sibling is a 3-node.



- Subcase 2: Left sibling is a 2-node.



Now the parent becomes empty.
Further need to handle

Remove Leaf

Summary

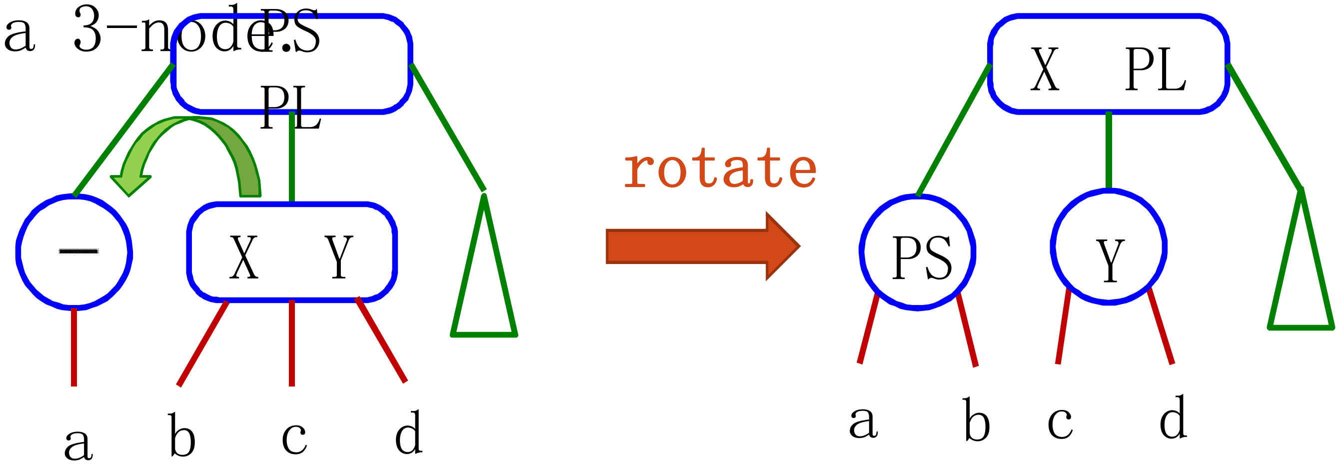
- When the **adjacent** sibling is a 3-node, rotate keys in the sibling node and the parent node. We are Done.

- When the **adjacent** sibling is a 2-node, borrow a key from the parent and then **merge** nodes.
  - If the parent is a 3-node, we are done.
  - … becomes … in a similar way.

Note: when the parent is a 3-node, we only take the left child as the adjacent sibling of the center child.

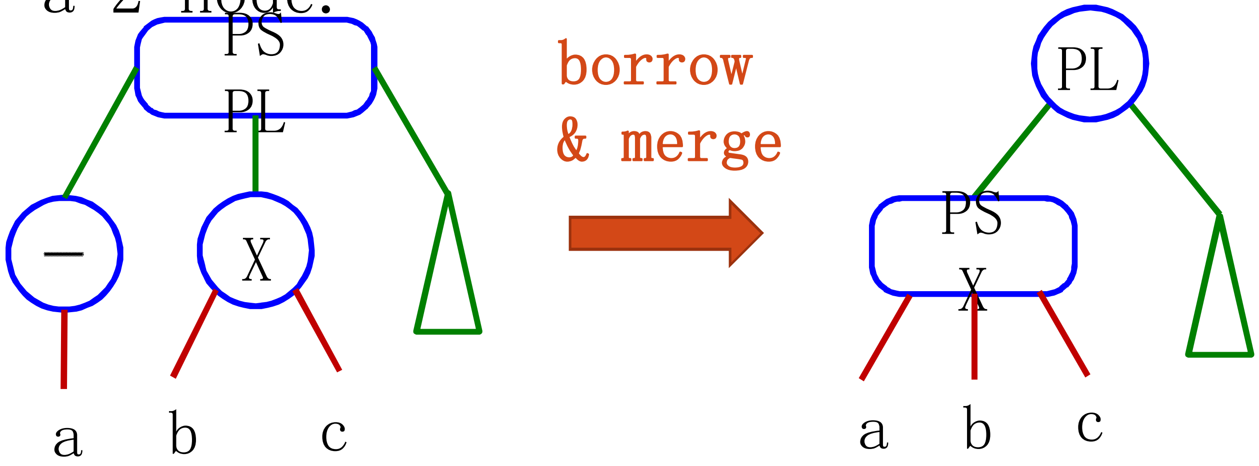
Handling Internal Empty Node

- Parent is a 3-node and the adjacent sibling a 3-node.



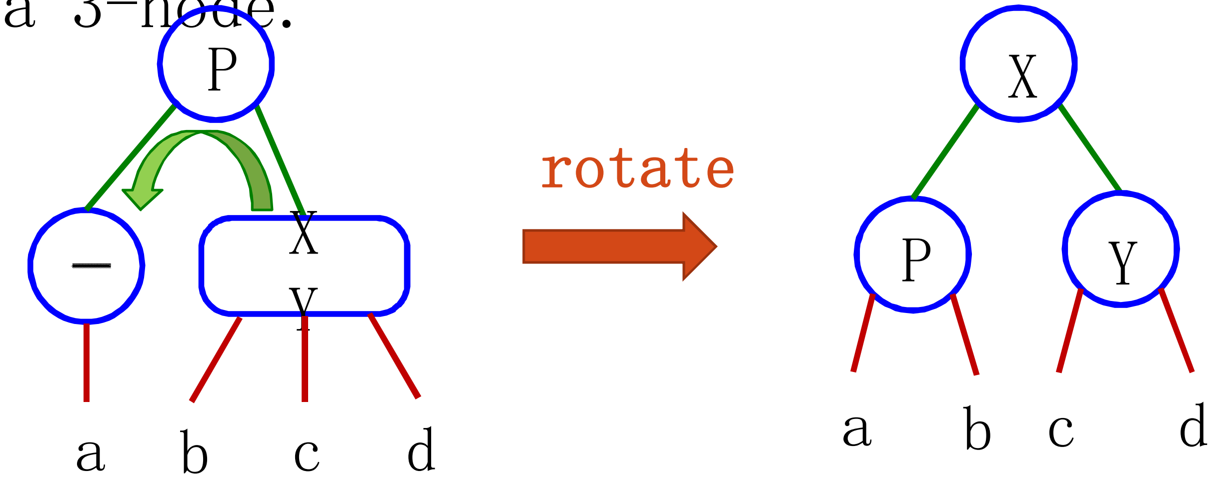
Handling Internal Empty Node

- Parent is a 3-node and the adjacent sibling a 2-node.



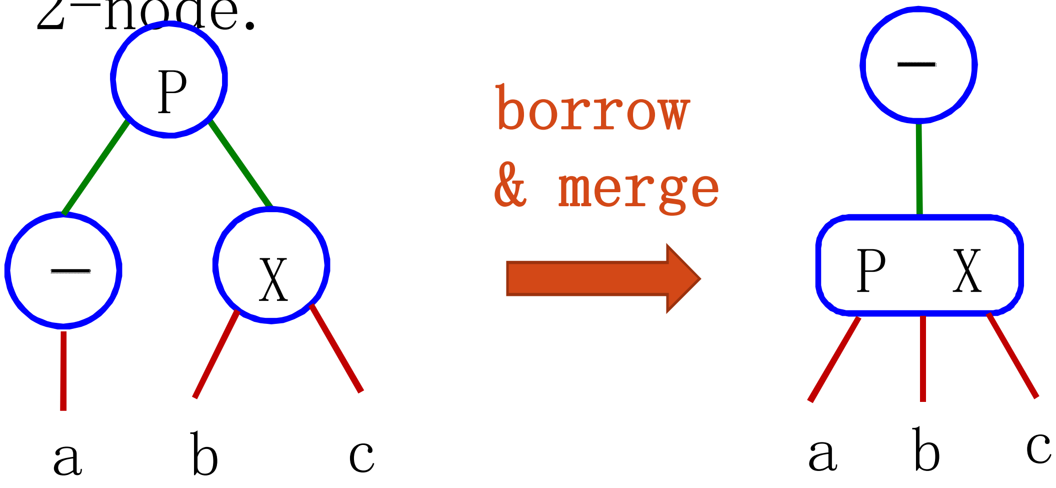
Handling Internal Empty Node

- Parent is a 2-node and the adjacent sibling a 3-node.



Handling Internal Empty Node

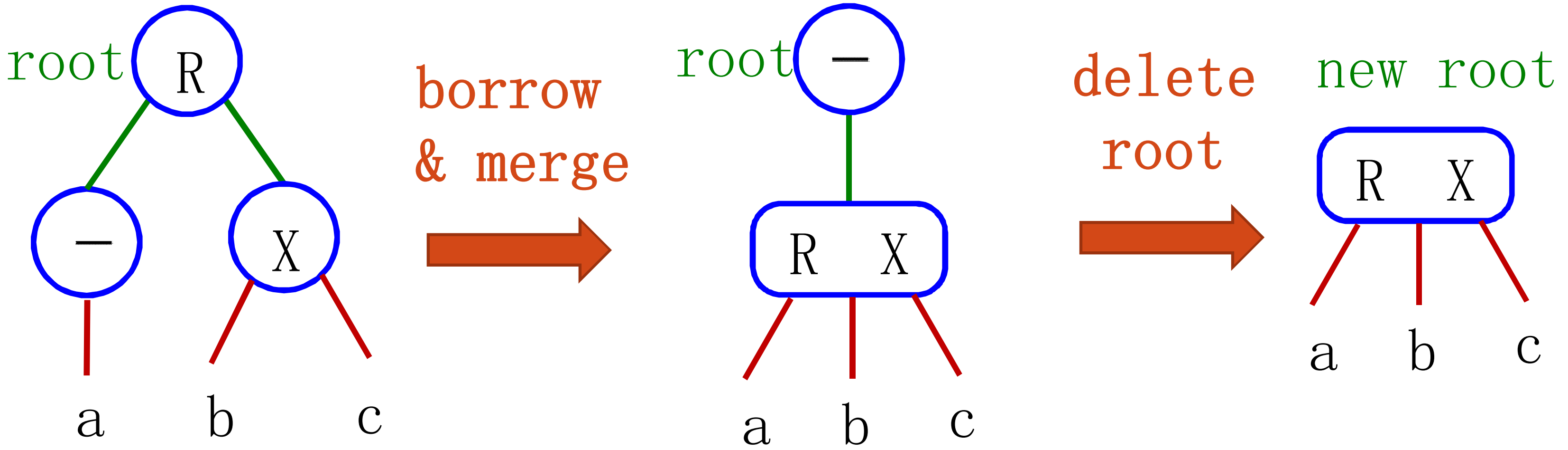
- Parent is a 2-node and the adjacent sibling a 2-node.



Now the parent becomes empty.
Need to handle internal empty node again (recursively).

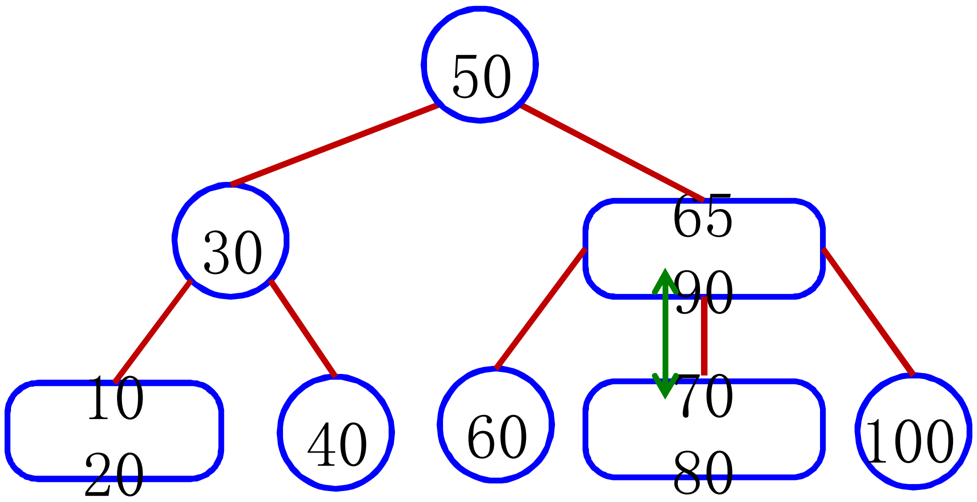
Deleting the Root

- We may **repeatedly** make the parent node empty.
- In the extreme case, the root becomes empty. Then, we delete the root.

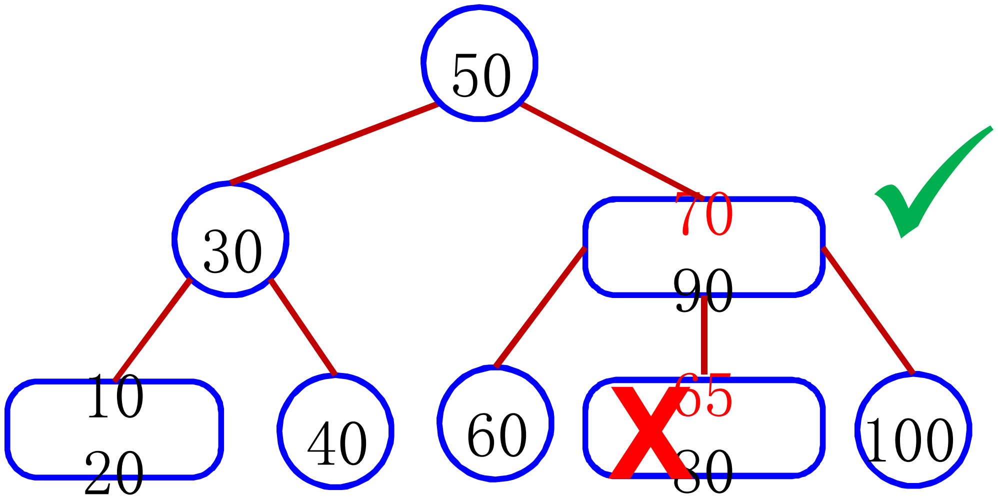


2-3 Tree Removal

Example

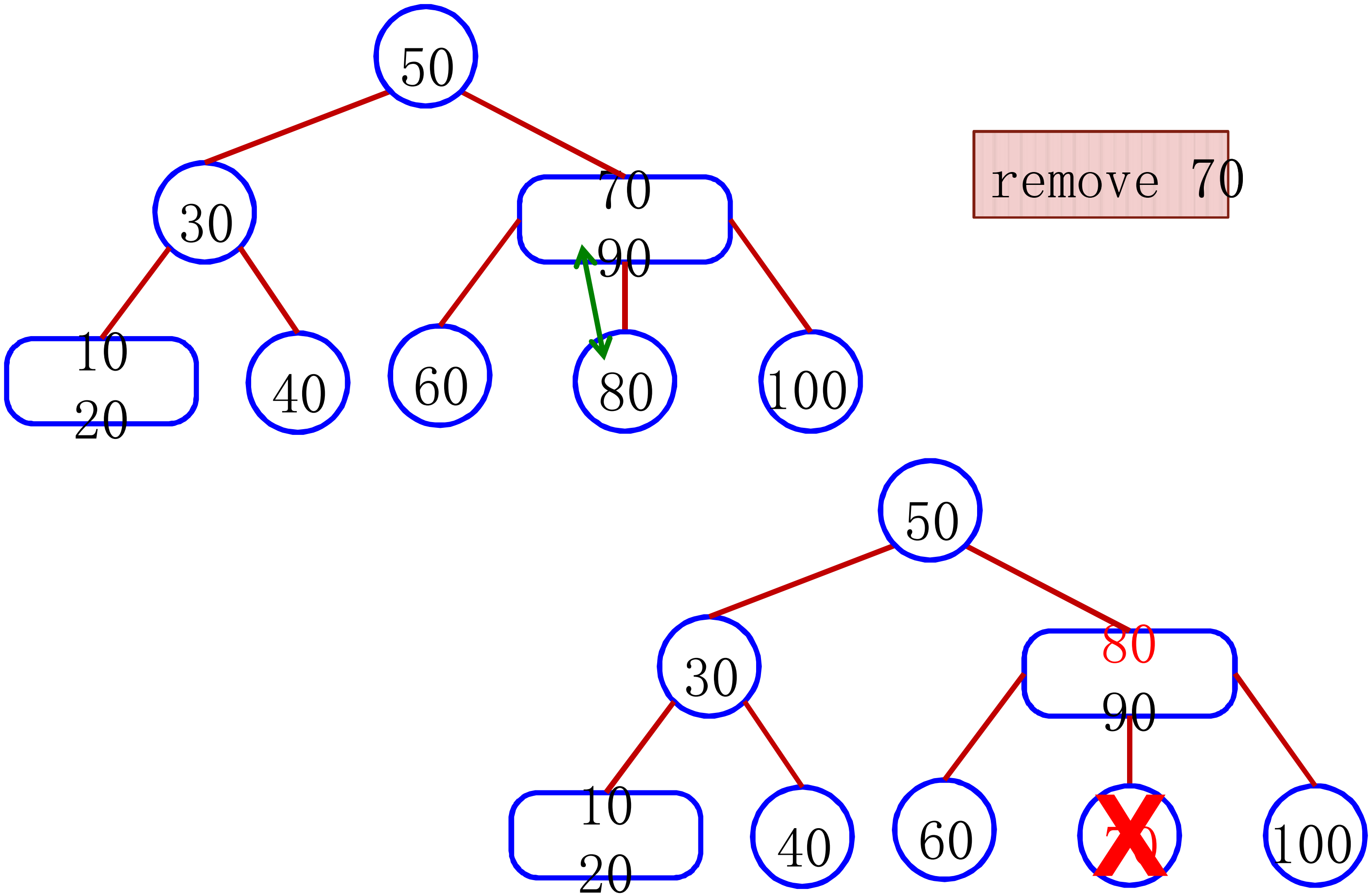


remove 65



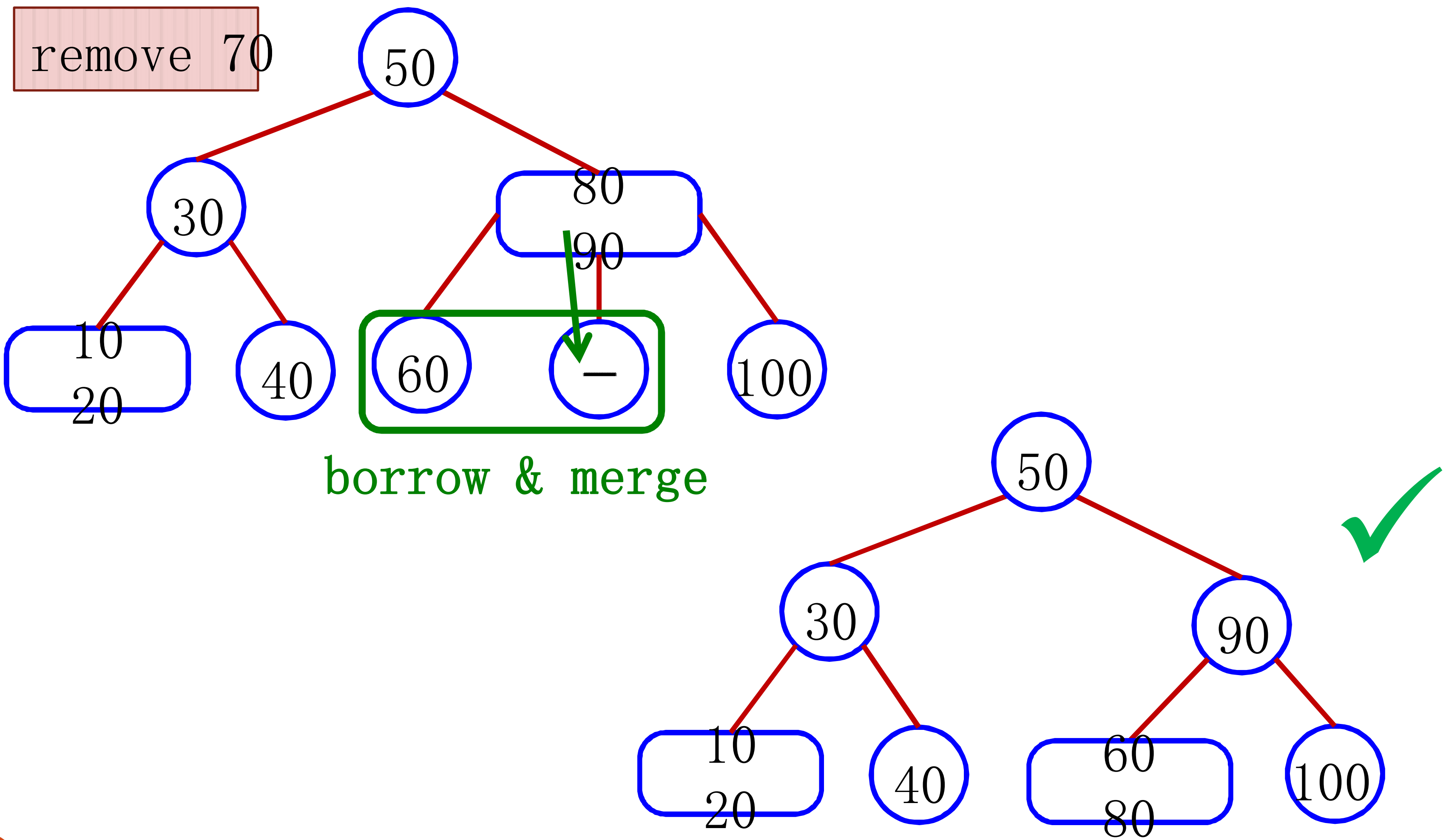
2-3 Tree Removal

Example



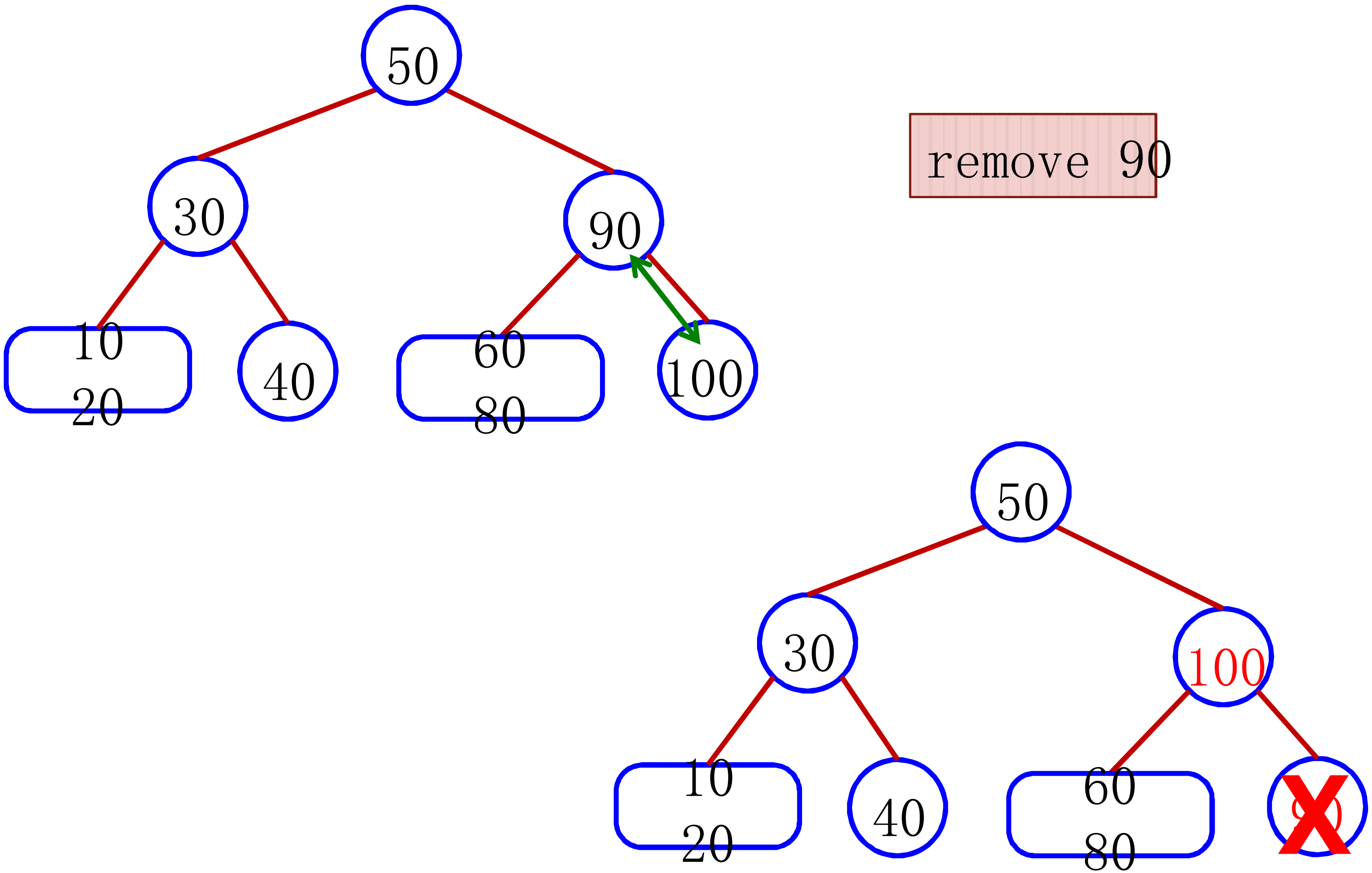
2-3 Tree Removal

Example



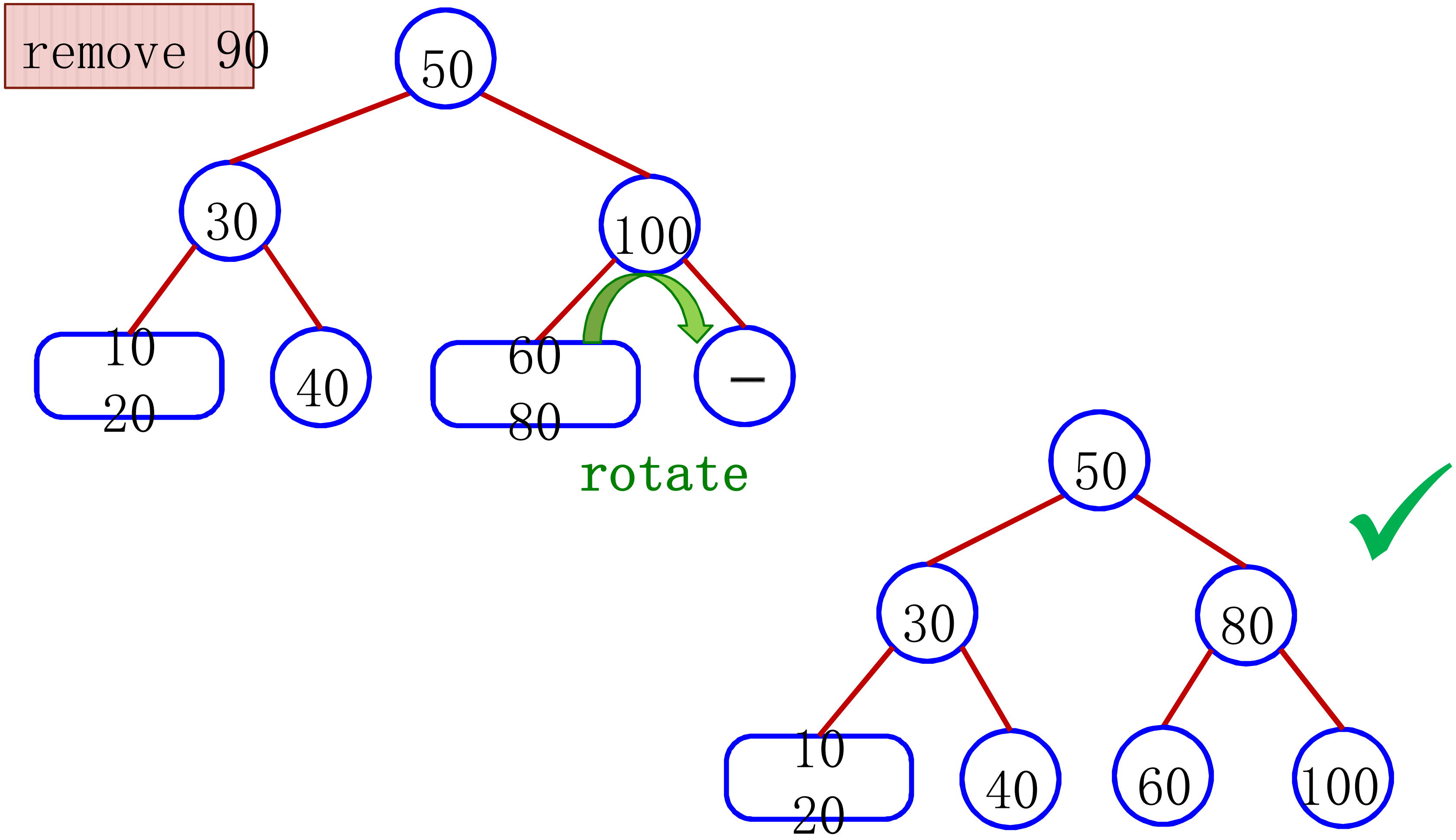
2-3 Tree Removal

Example



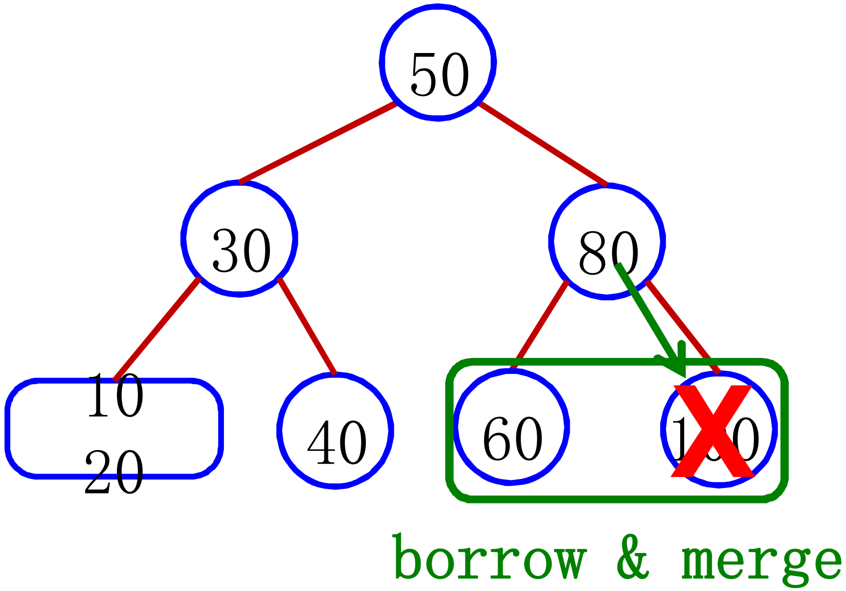
2-3 Tree Removal

Example

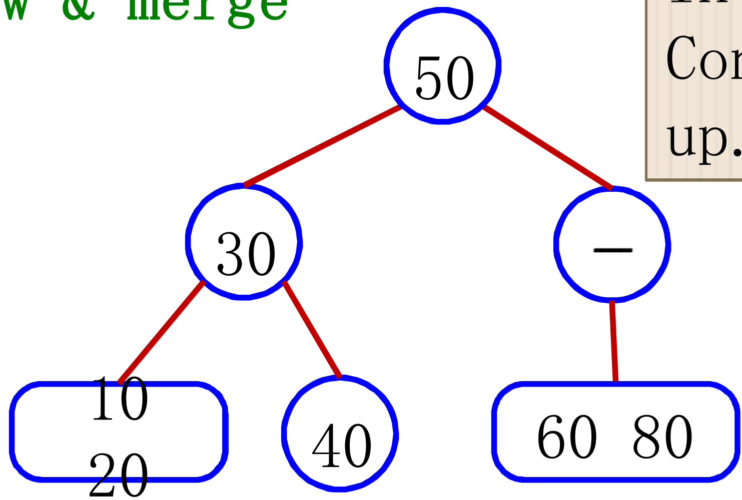


2-3 Tree Removal

Example



remove 100

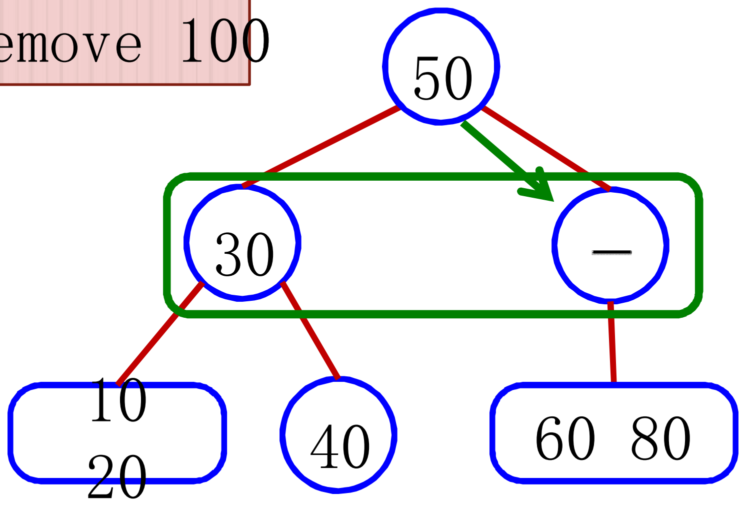


Internal node empt
Continue recursive
up.

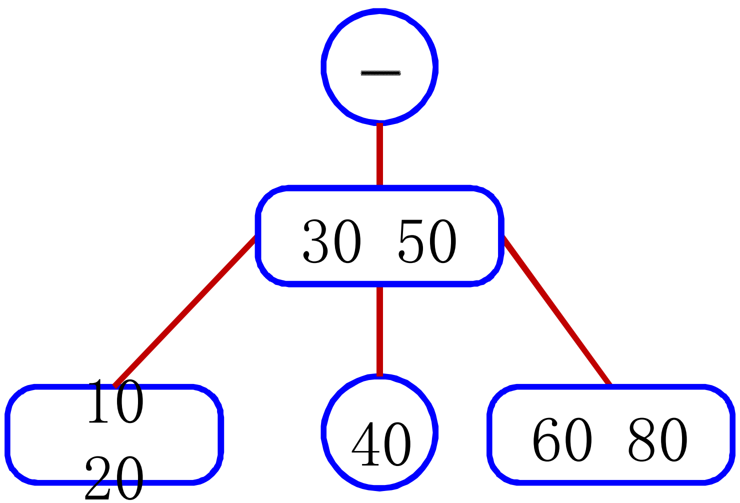
2-3 Tree Removal

Example

remove 100



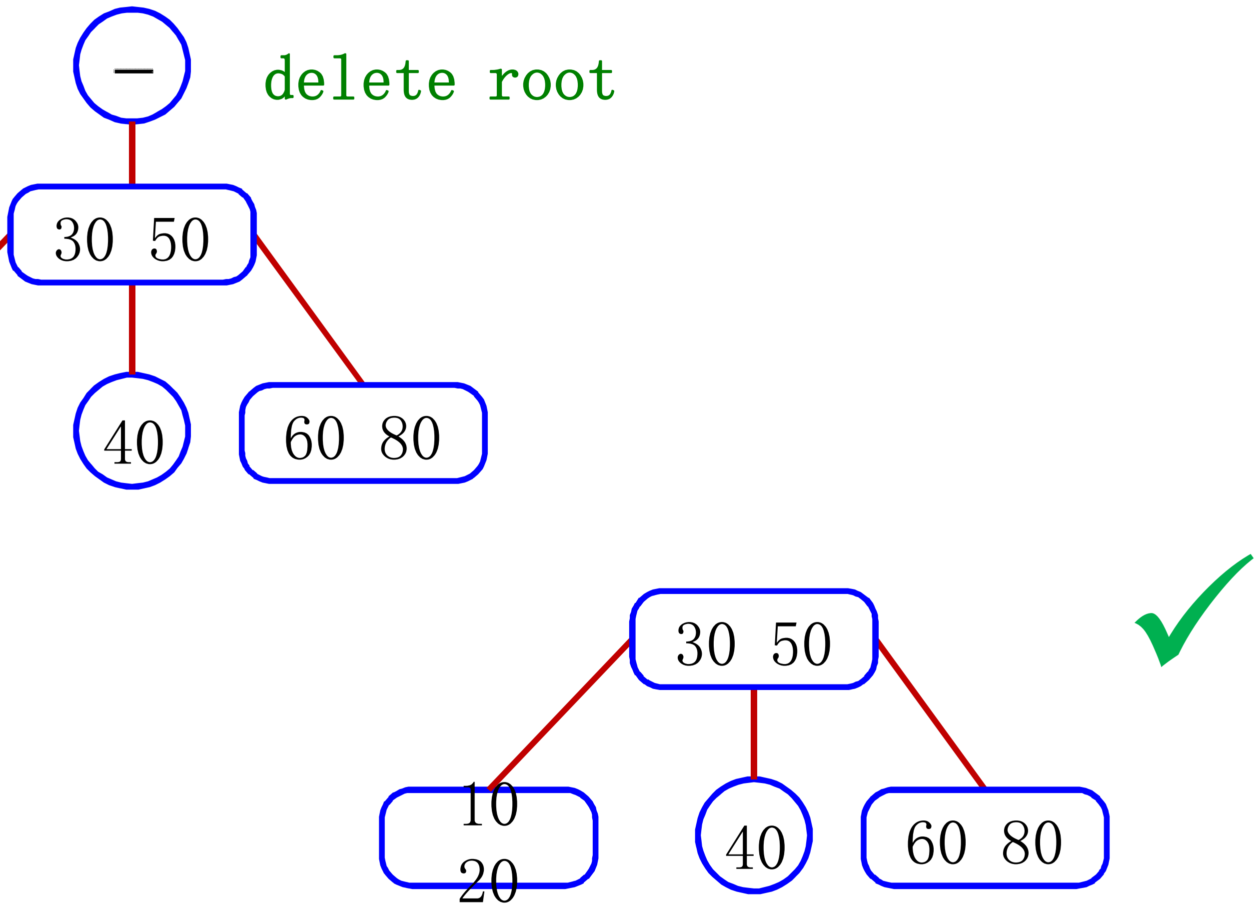
borrow & merge



2-3 Tree Removal

Example

remove 100



2-3 Tree Removal

Summary

- Whereas a BST pushes empty nodes down to the leaves, a 2-3 tree percolates empty nodes up and decreases height globally by lowering the root.

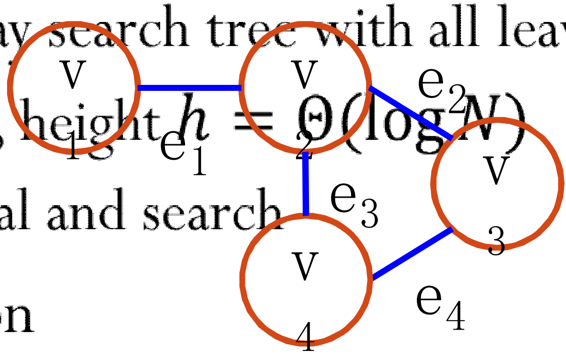
- What is the worst case time complexity?
  - $O(\log N)$

Outline

- 2-3 Tree: Insertion
- 2-3 Tree: Removal
- Graphs

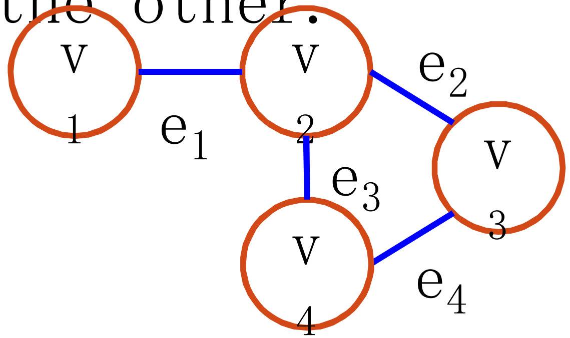
Graphs

- Trie: Insertion, removal, and time-complexity
- M-way search tree: A generalization of binary search tree
- 2-3 Tree
  - A balanced 3-way search tree with all leaves at the same level.
  - To store N keys, height $h = \Theta(\log N)$
  - In-order traversal and search
- 2-3 Tree: Insertion
  - Sometimes we will make an internal node have three keys. Then we **split** the node and move the middle key up to its parent.



Graphs

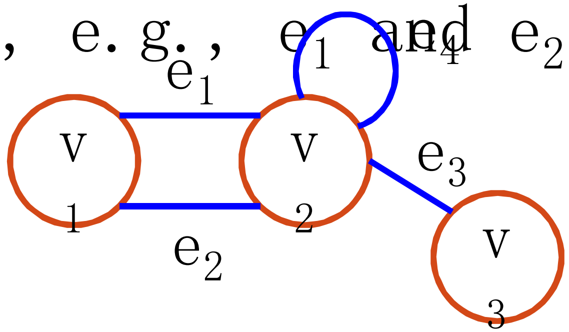
- Directly connected nodes are **adjacent** to each other (e.g., v_1 and v_2), and one is the **neighbor** of the other.



- The edge directly connecting two nodes are **incident** to the nodes, and the nodes **incident** to the edge.

Simple Graphs

- Two nodes may be directly connected by more than one **parallel edges**, e.g., e_1 and e_2.



- An edge connecting a node to itself is called a **self-loop**, e.g., e_4.
- A **simple graph** is a graph without parallel edges and self-loops.
  - Unless otherwise specified, we will work only with simple graphs in this course.