Data Structures:
Height-Balanced Search Trees:
2-3 Tree, T Tree

Won Kim
(Lecture by Youngmin Oh)
Spring 2022

2-3-Tree

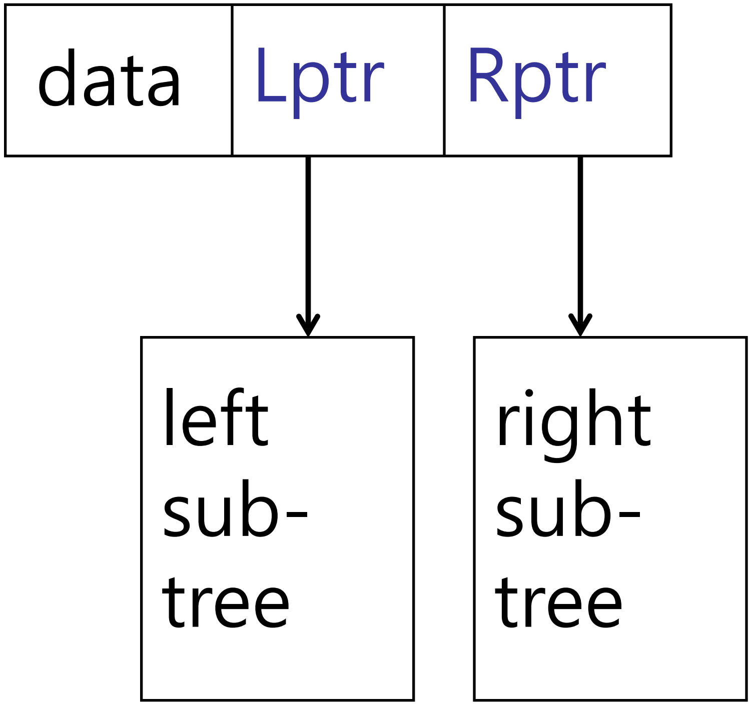
2-3 Tree

- A "Perfectly Balanced Tree"
  - All leaf nodes are on the same level
- Invented by J.E. Hopcroft in 1970.
- Not used much
- But, a special case of B Tree/B+ Tree, and base of T Tree
  - B Tree/B+ Tree is very important
  - T Tree is important

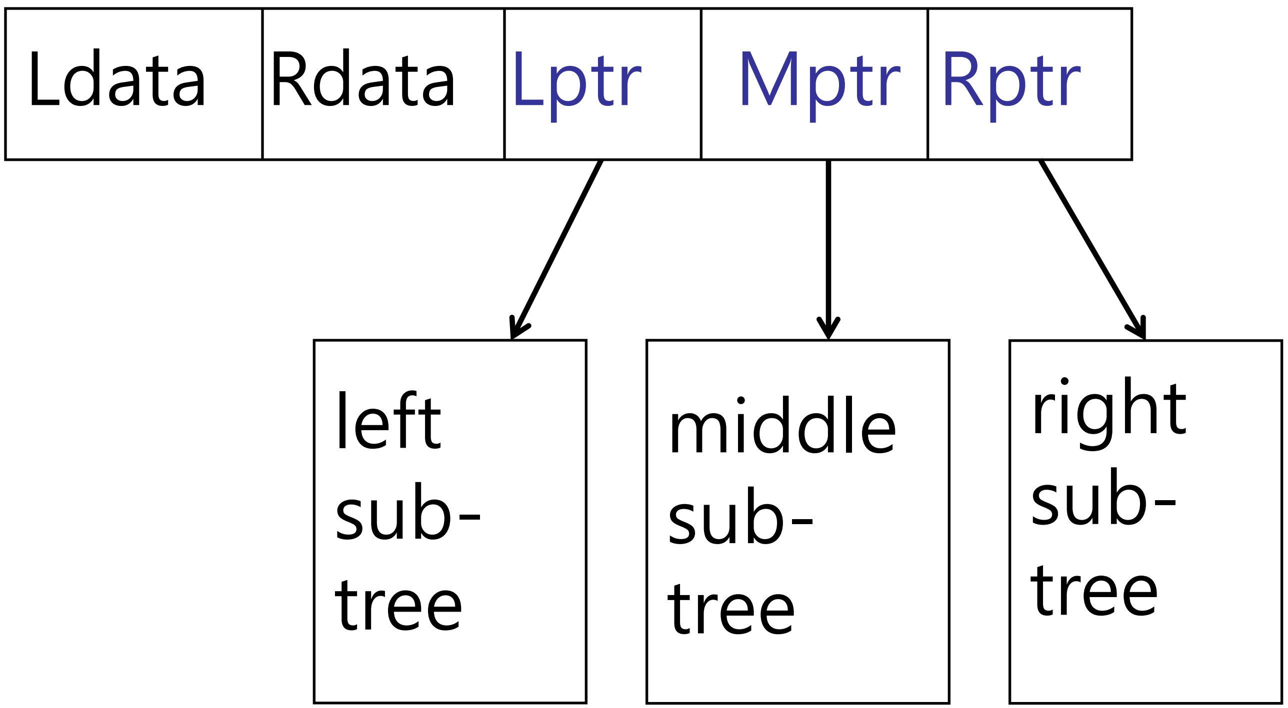
2-3 Tree

- Has Only 2-Nodes and 3-Nodes.
- smaller key to the left subtree, and larger key to the right subtree
- 2-node
  - with one key, and two child nodes (left, right)
  - root key of the left subtree < key
  - root key of the right subtree > key
- 3-node
  - with two keys (left, right), and three child nodes (left, middle, right)
  - root key of the left subtree < left key
  - root key of the middle subtree > left key AND < right key
  - root key of the right subtree > right key

2 Node (Implementation)



3 Node (Implementation)

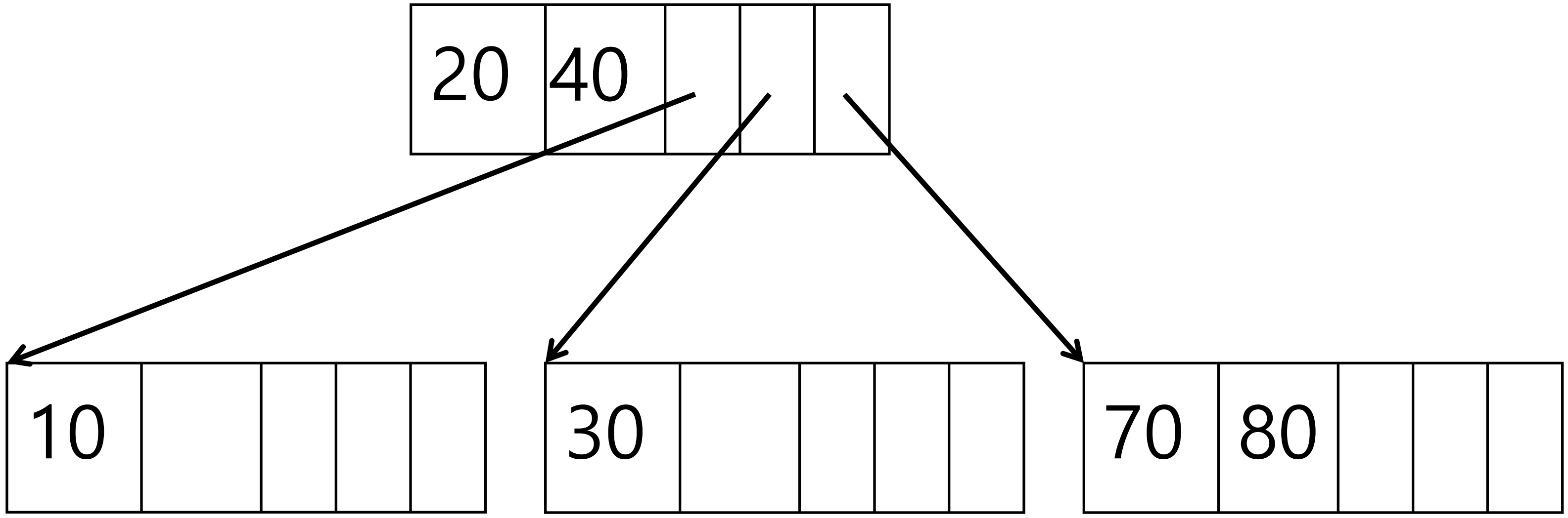


Searching a 2-3 Tree

- Search key X
- In a 2-Node
  - If X = the key of the node, search ends.
  - If X < the key of the node, search the left subtree.
  - If X > the key of the node, search the right subtree.
- In a 3-Node
  - If X = the left data or right data, search ends.
  - If X < the left data, search the left subtree.
  - If X > the left data and < the right data, search the middle subtree
  - If X > the right data, search the right subtree.
- If X is not found, search fails.

Searching a 2-3 Tree

Search for 80, 10, 25, 60



Height Balancing a 2-3 Tree

- Node Promotion and Node Demotion
  - node promotion: a 2-node becomes a 3-node
  - node demotion: a 3-node becomes a 2-node
- Data Re-Distribution
  - node split and node merge

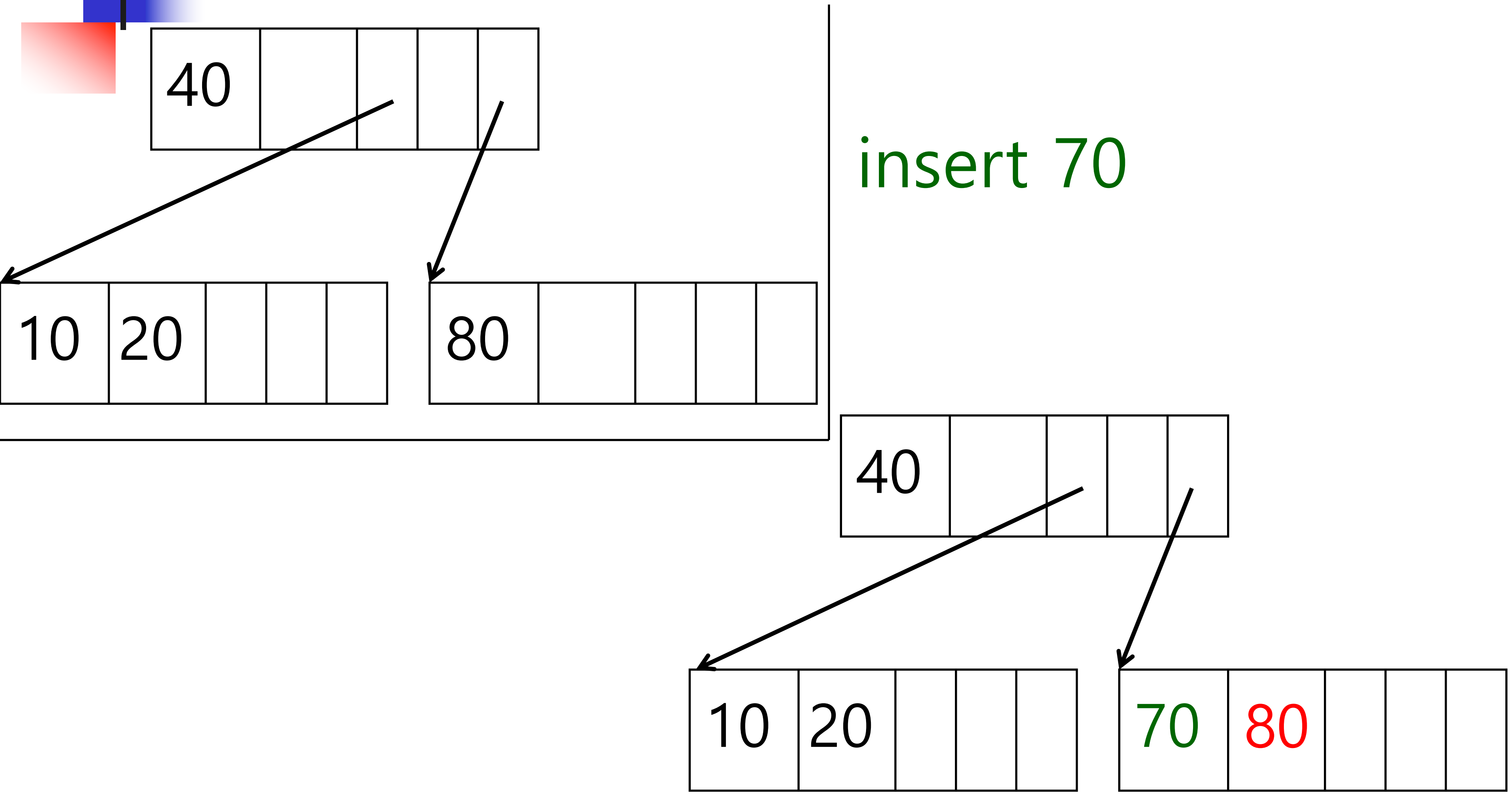
Insight on a 2-3 Tree

- A node has a minimum 1 data, and maximum 2 data.
  - maximum # of data = 2 x minimum # of data
  - overflow: 3rd data
  - underflow: 0 data
- Overflow and underflow require tree restructuring.
- Tree height increases by 1, only when all nodes are 3-nodes.

Inserting Data Into a 2-Node

- A 2-node becomes a 3-node.
- The smaller data becomes the "left" data.
- The larger data becomes the "right" data.
- Pointers (to the child nodes) in the node are adjusted.

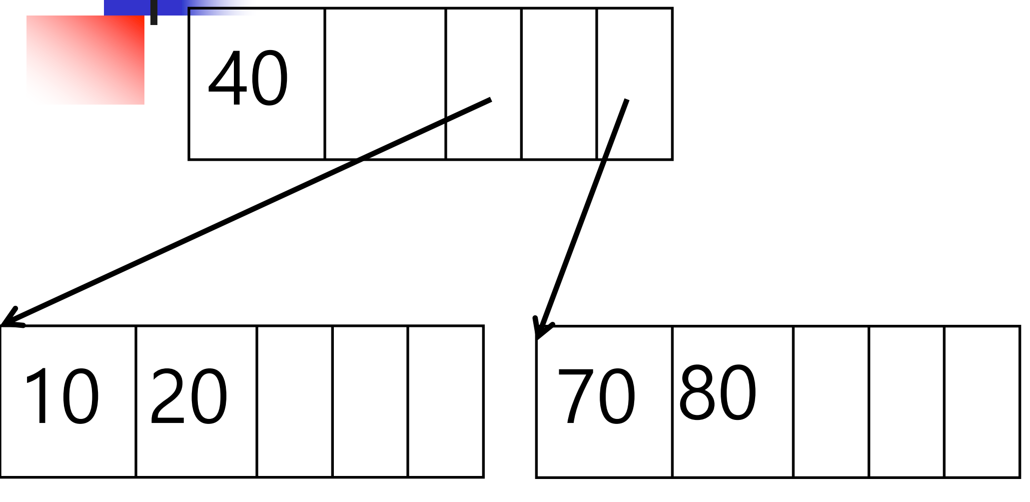
Example



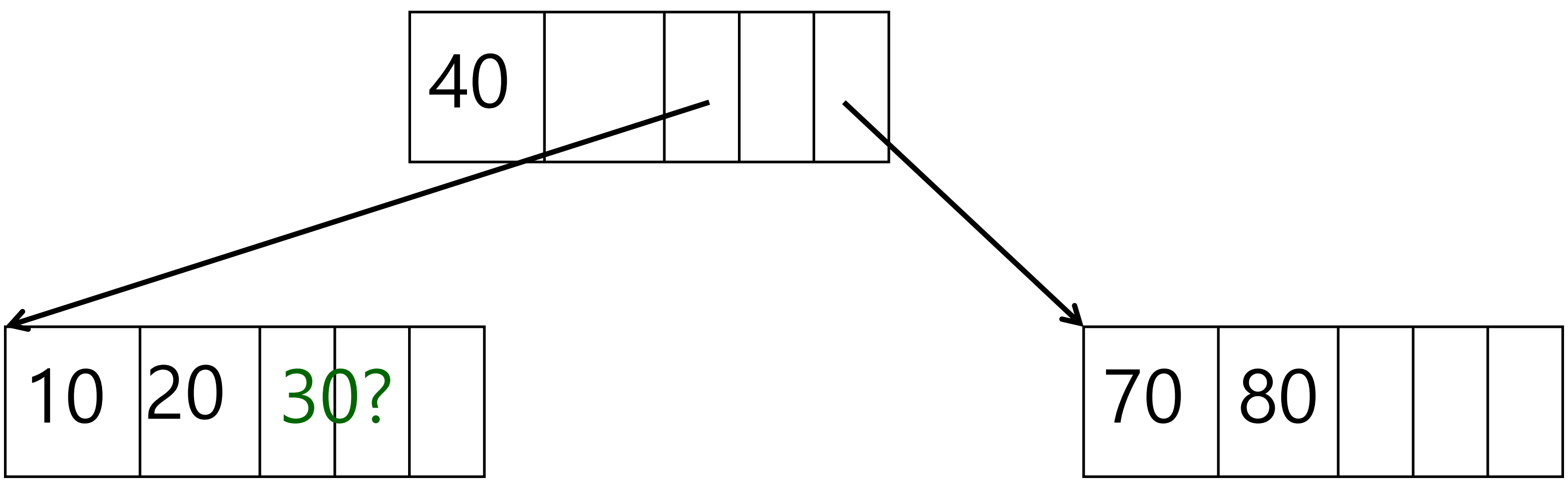
Inserting Data Into a 3-Node

- The 3-node splits into 2 separate 2-nodes (to reserve space for future inserts)
  - The “smallest” data goes to the left 2-node.
  - The “largest” data goes to the right 2-node.
  - The “middle” data goes to the parent node.
- The “middle” pointer in the parent node points to one of the two new 2-nodes.
- If the parent node is a 3-node, it is split, too, recursively.

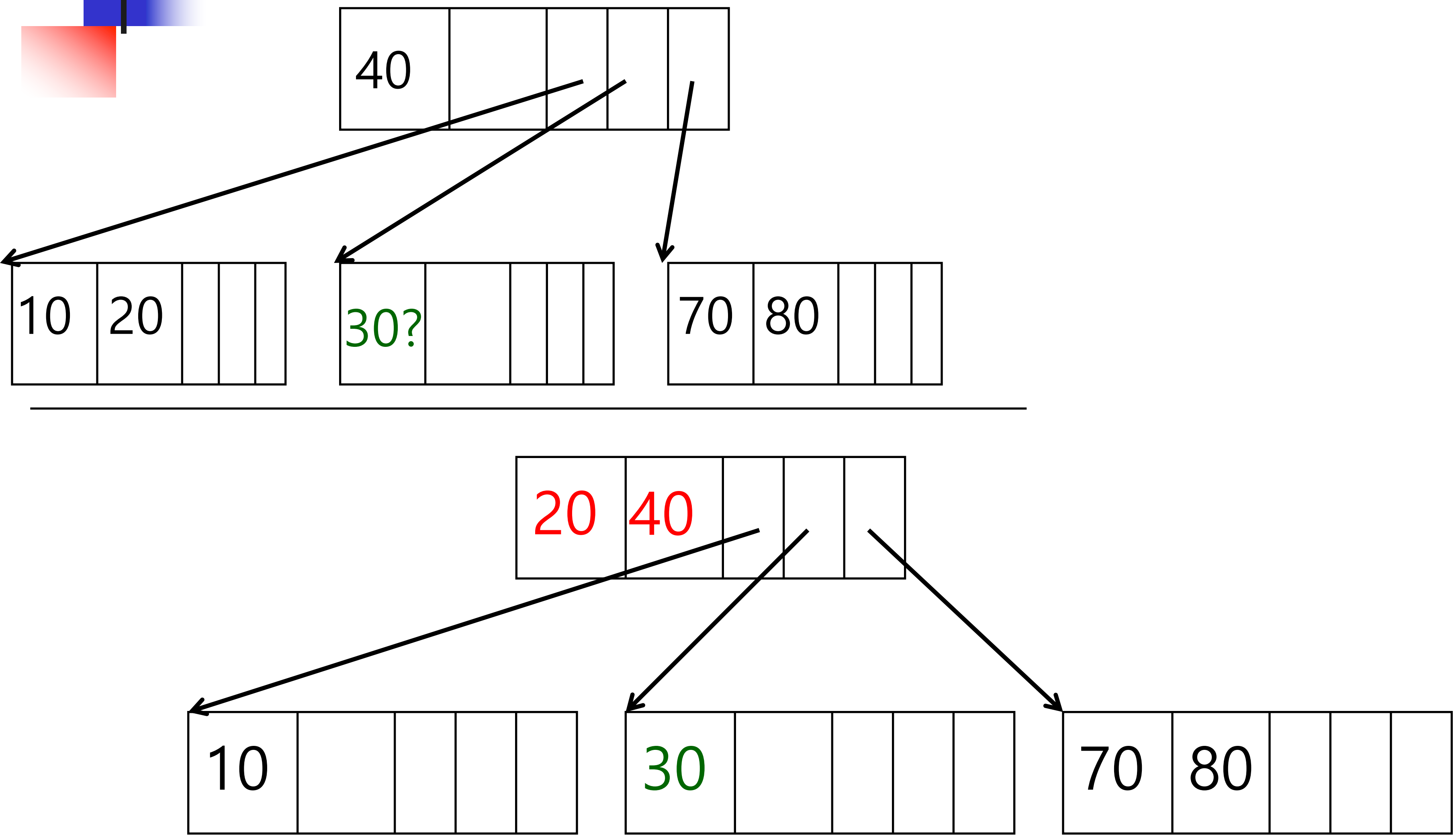
Example 1: (1/2)



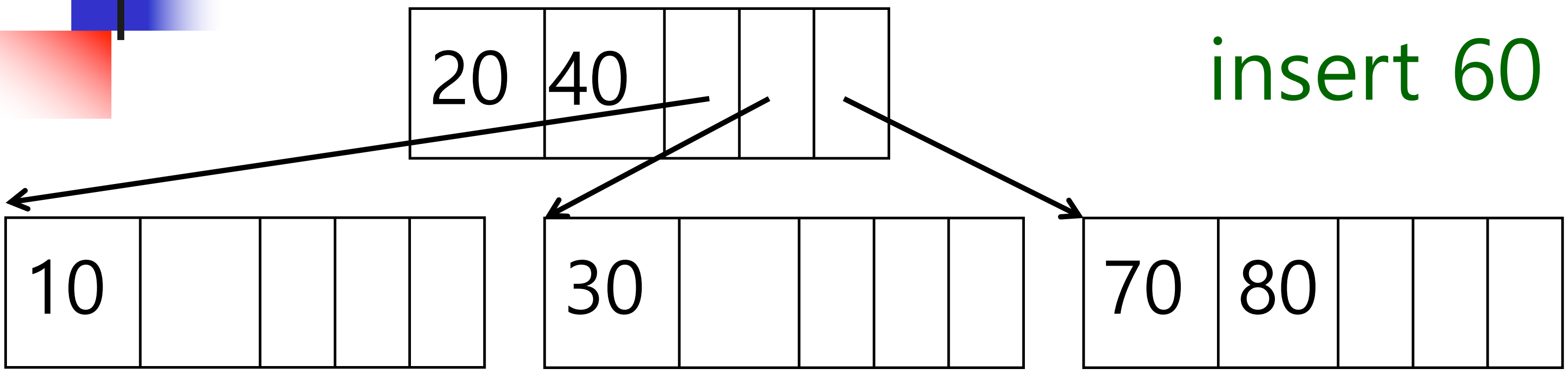
insert 30



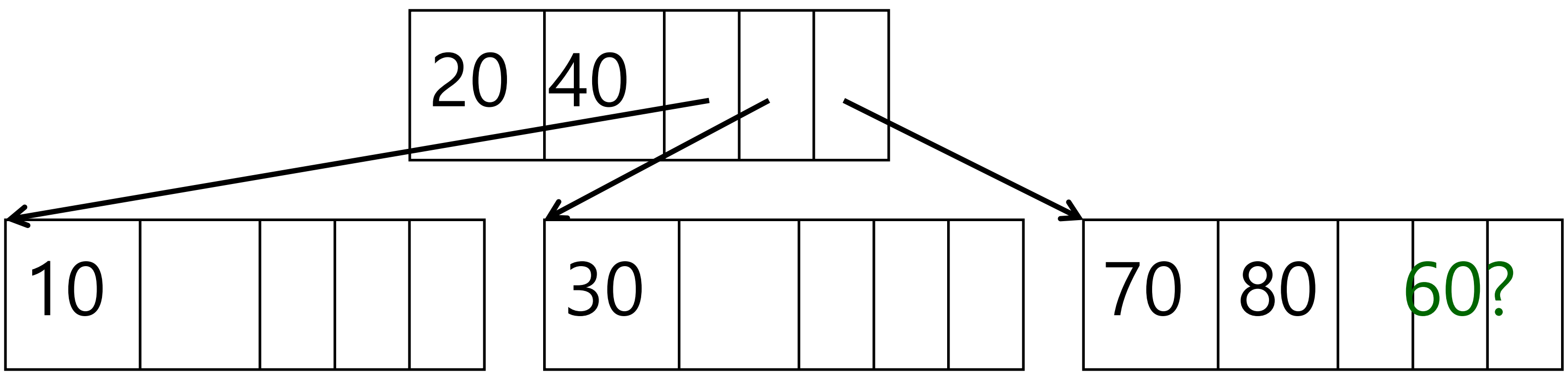
Example 1: (2/2)



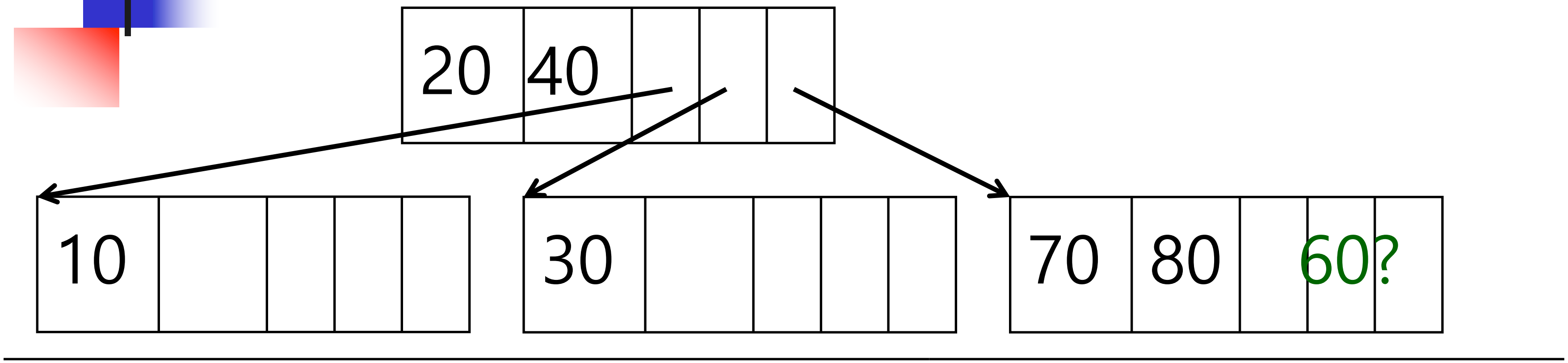
Example 2: (1/3)

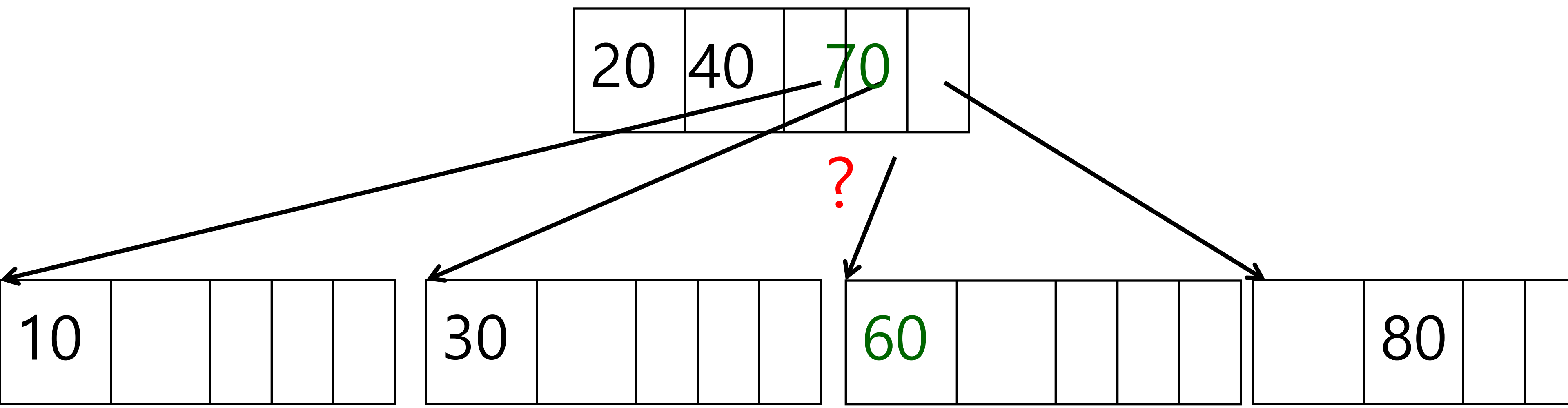


insert 60

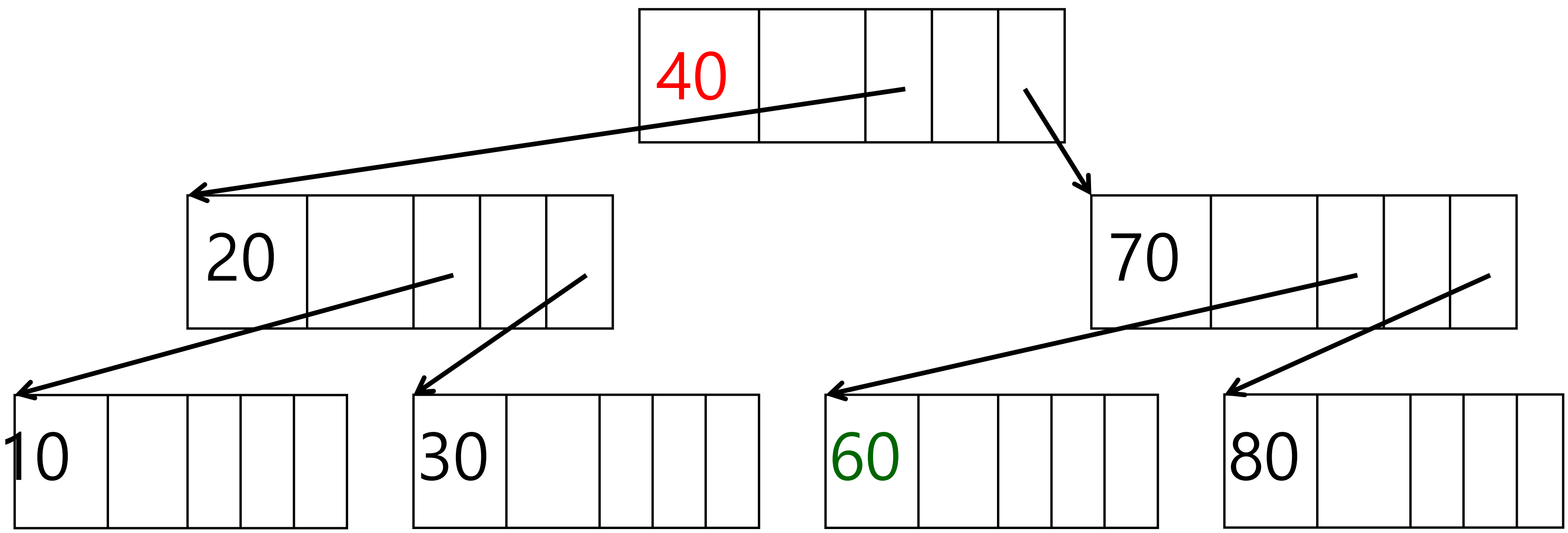


Example 2: (2/3)





Example 2: (3/3)



Example 3: (1/2)

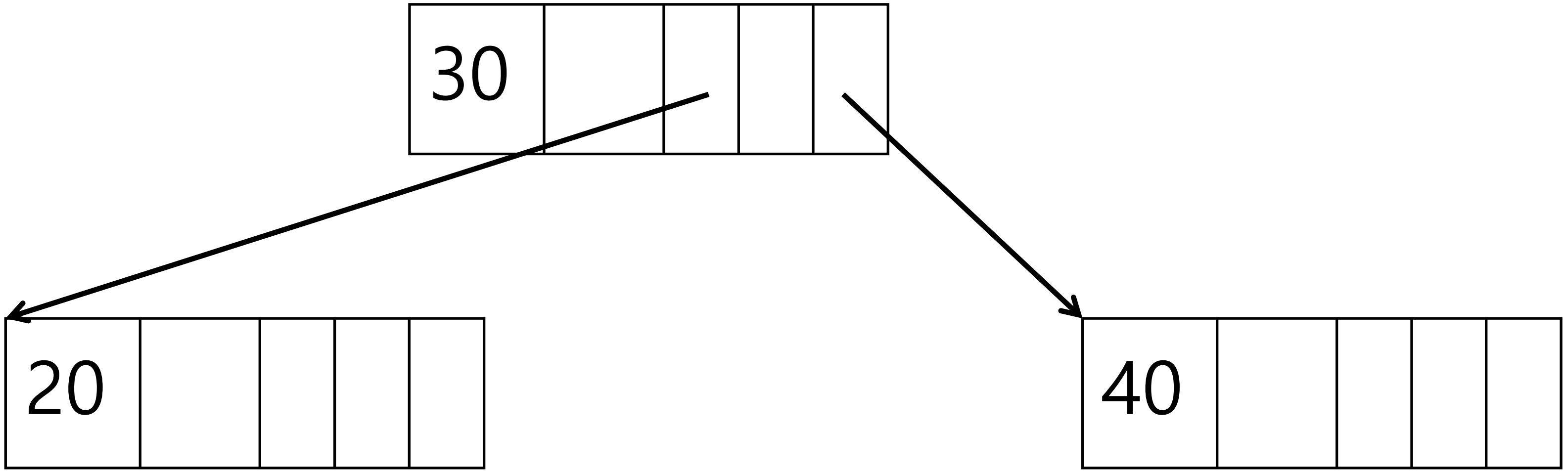
insert 30

20	40			

30?

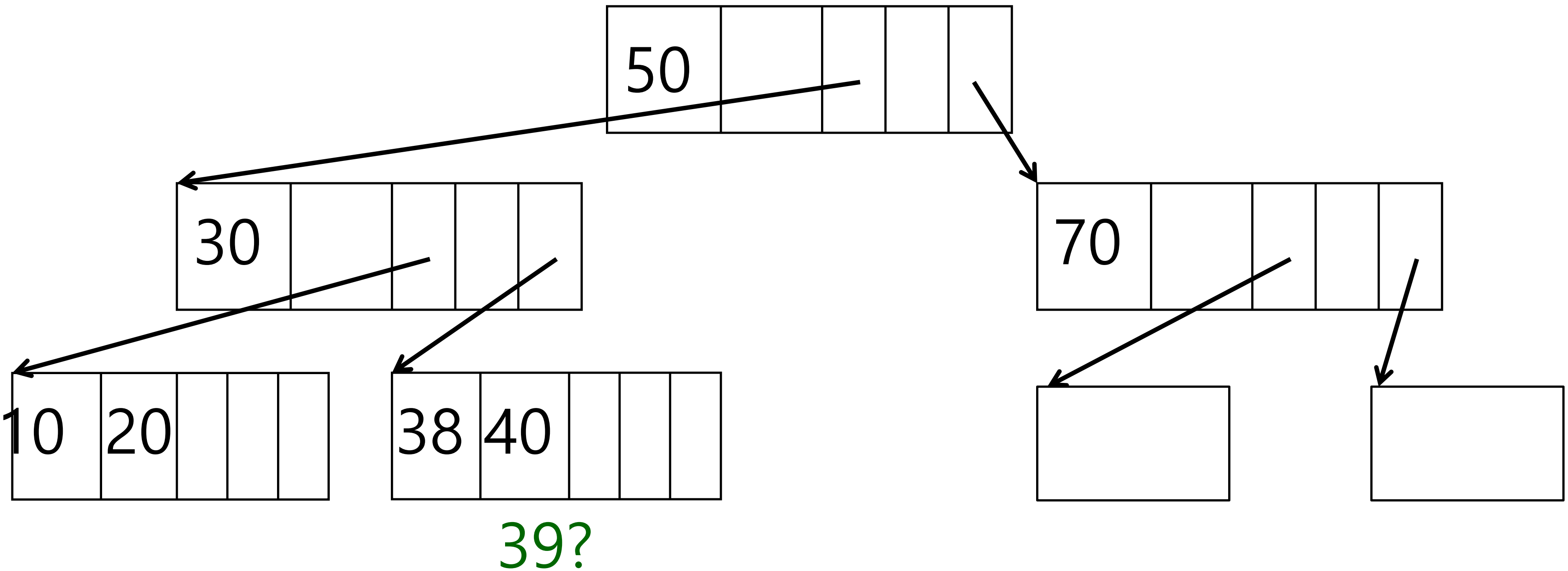
Example 3: (2/2)

insert 30



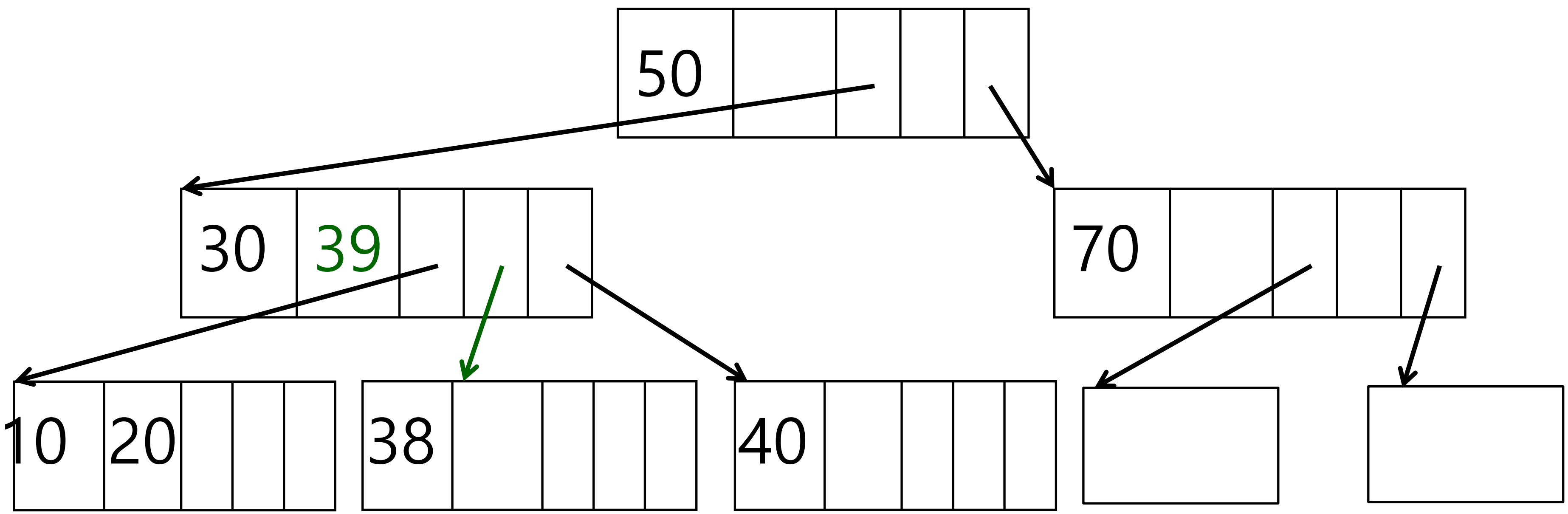
Example 4: (1/2)

insert 39



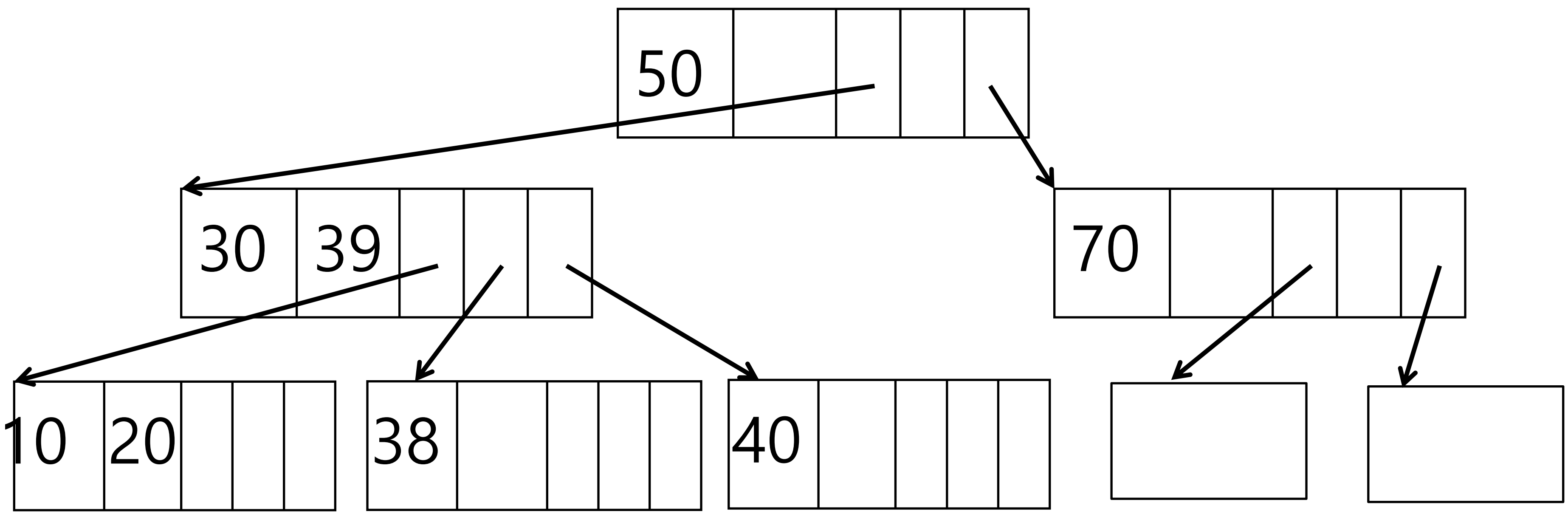
Example 4: (2/2)

insert 39



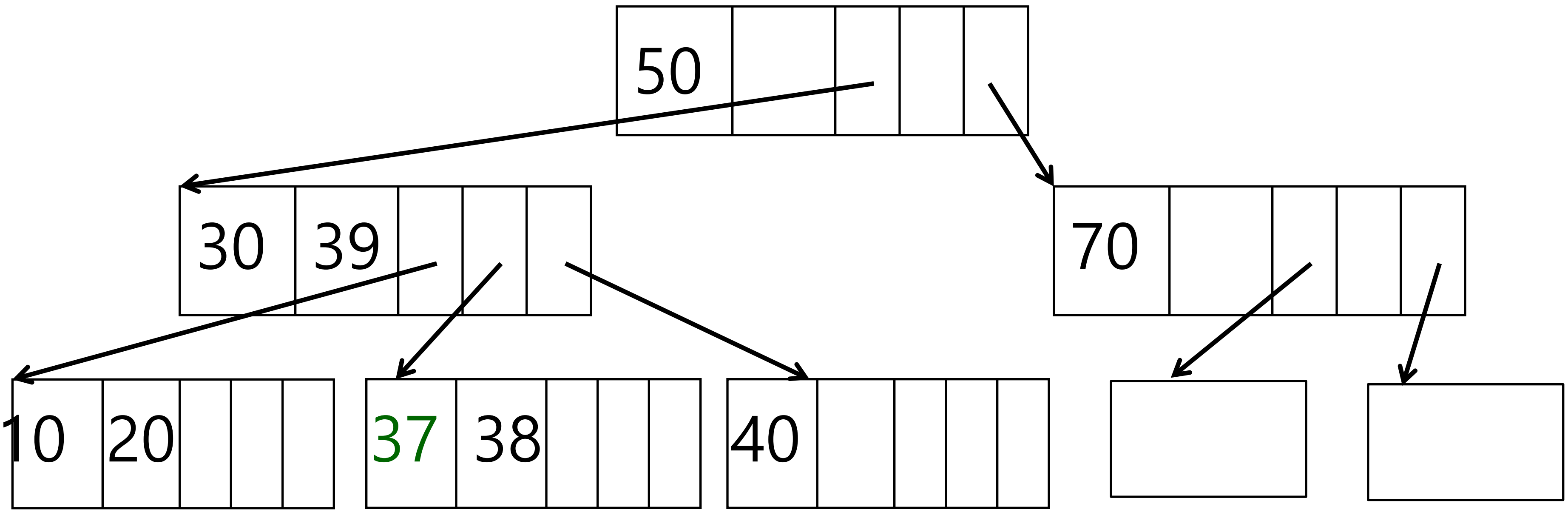
Example 5: (1/2)

insert 37



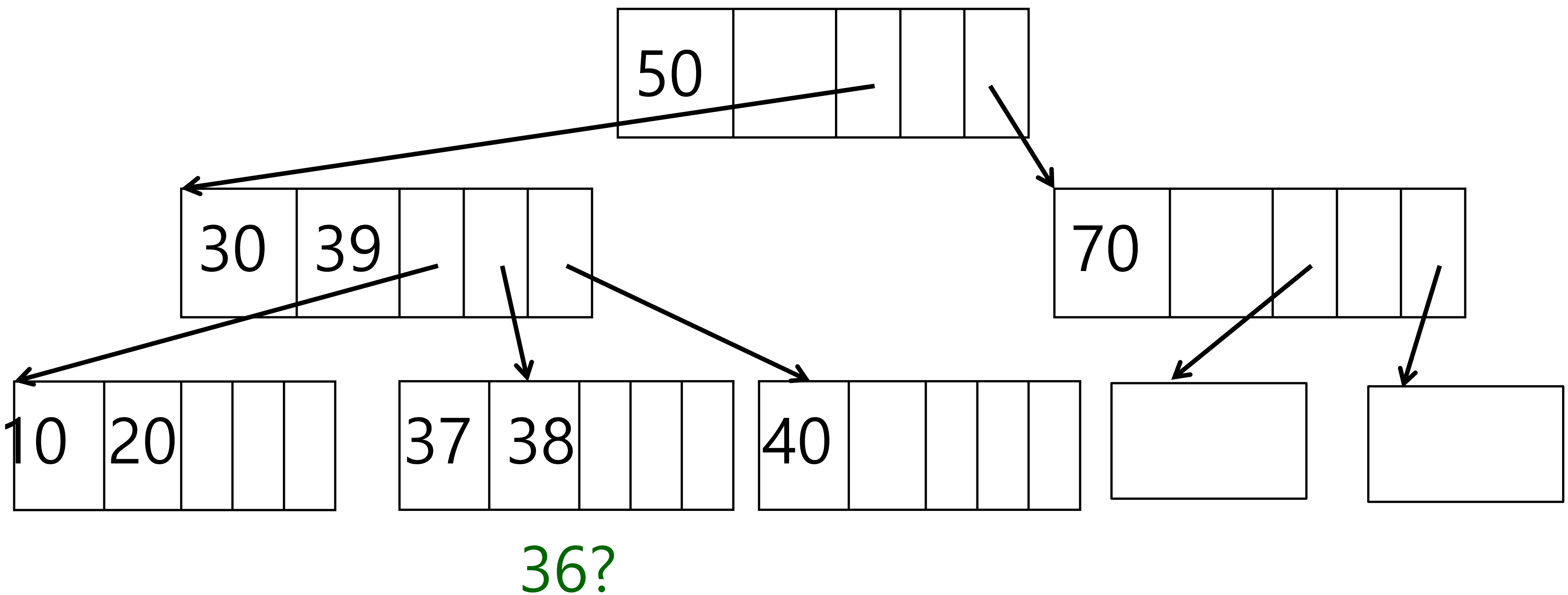
Example 5: (2/2)

insert 37



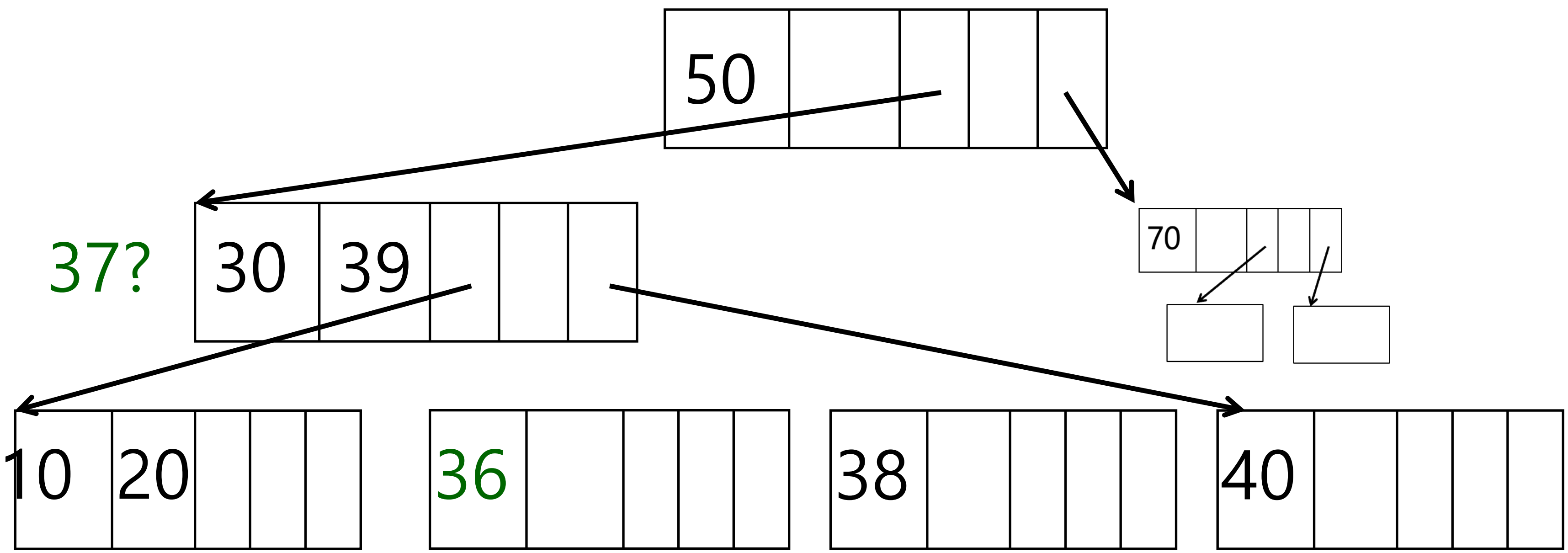
Example 6: (1/3)

insert 36



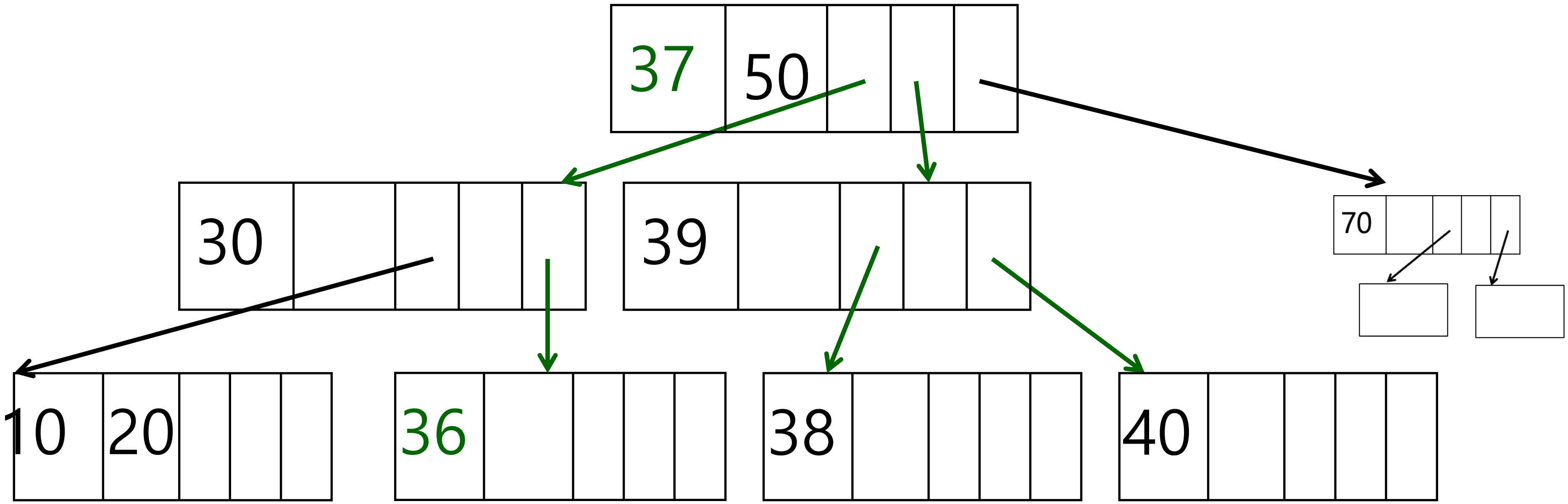
Example 6: (2/3)

insert 36



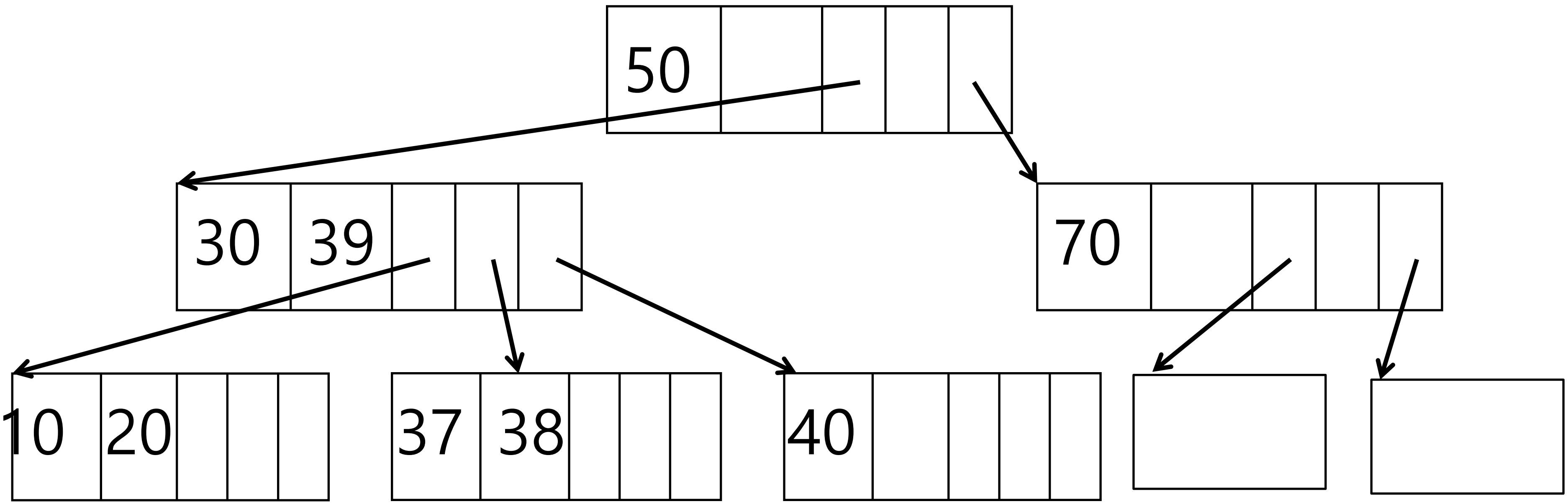
Example 6: (3/3)

insert 36



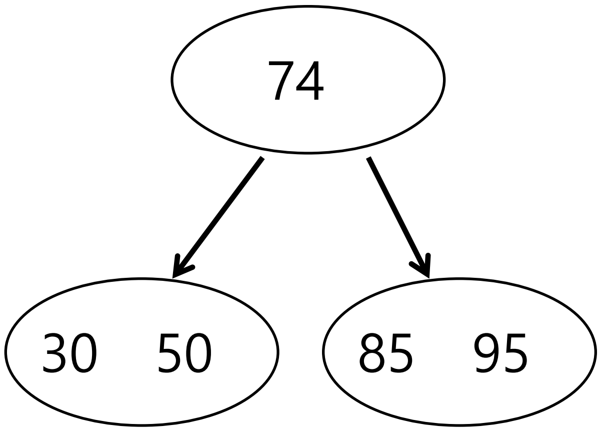
Exercise

insert 6

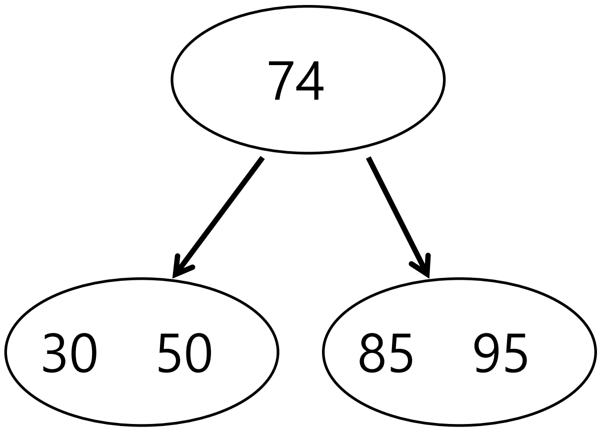


Exercise : Insert "60" Into the Following 2-3 Tree.

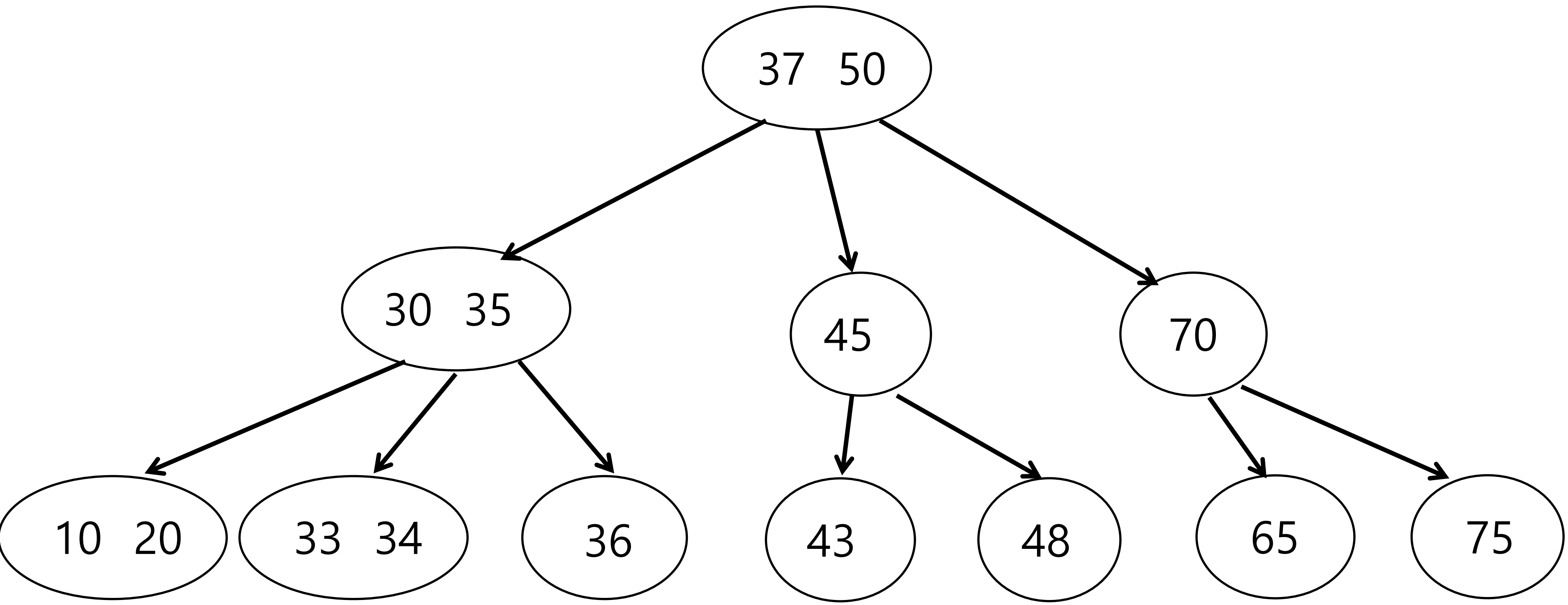
simplified notation



Exercise : Insert "40" Into the Following 2-3 Tree.



Exercise: Insert “32” into the Following 2-3 Tree.



Deletion

- Node merge is the reverse of node split

Deleting Data from a 3-Node (1/2)

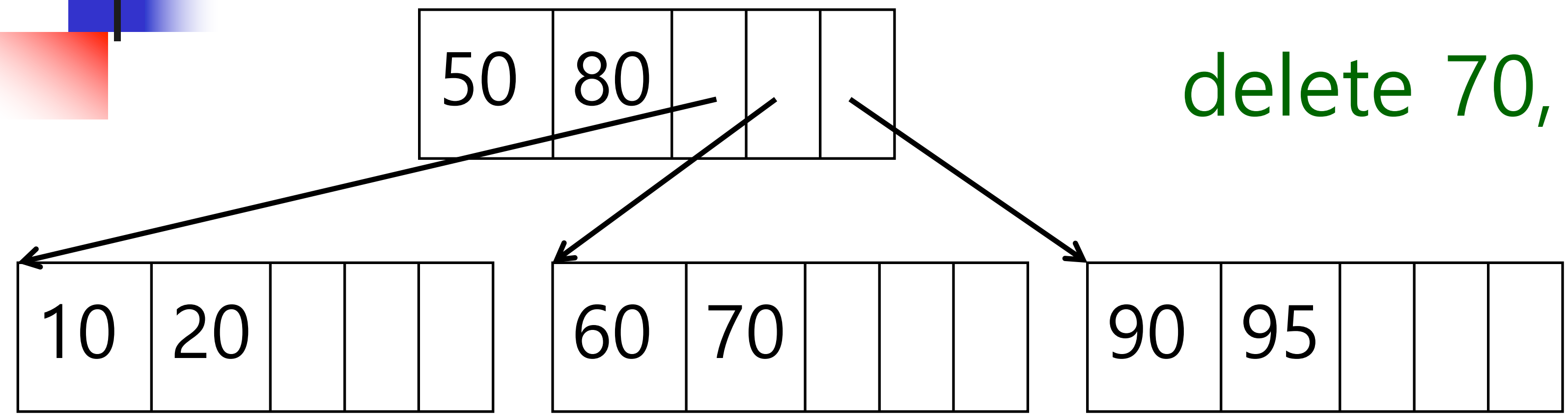
- If the 3-node is a leaf node
  - Just delete the data.
  - The node is now a 2-Node.

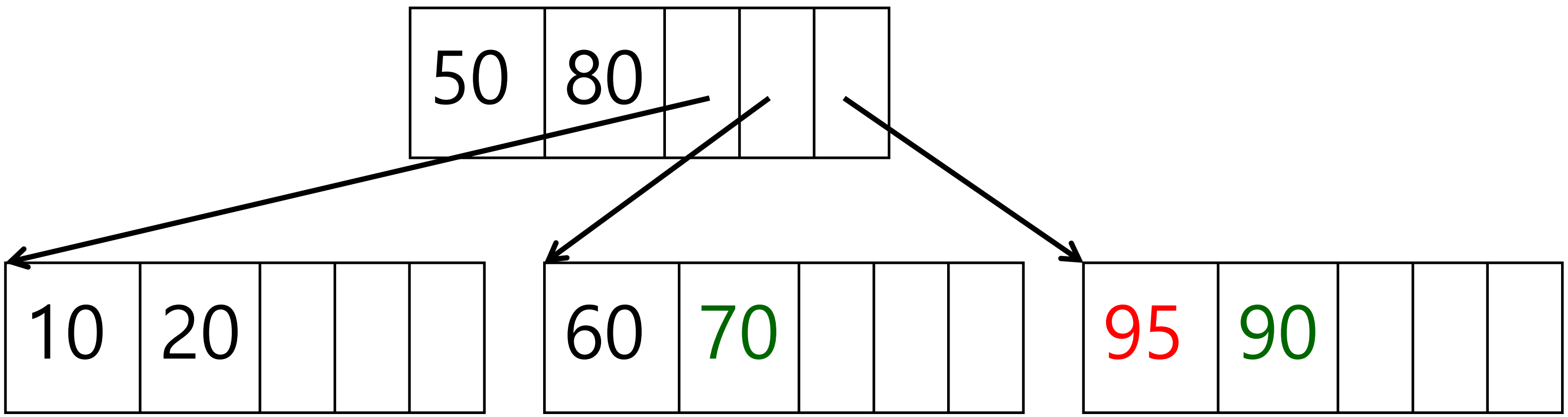
Deleting Data from a 3-Node (2/2)

- If the 3-node is a non-leaf node
- (with respect to the key to be deleted)
  - ** If both the left and right child nodes are 2-nodes
    - Merge the child nodes, and delete the key in the 3-node
  - *** If one of the left and right child nodes is a 3-node
    - If left data is to be deleted, swap the left data with the greatest key on the left subtree, or the smallest key on the middle subtree.
    - If right data is to be deleted, swap the right data with the greatest key on the middle subtree, or the smallest key on the right subtree.
    - Delete the data after the swap.
    - If the node underflows, solve the problem recursively.

Example 1 (cf. page 33)

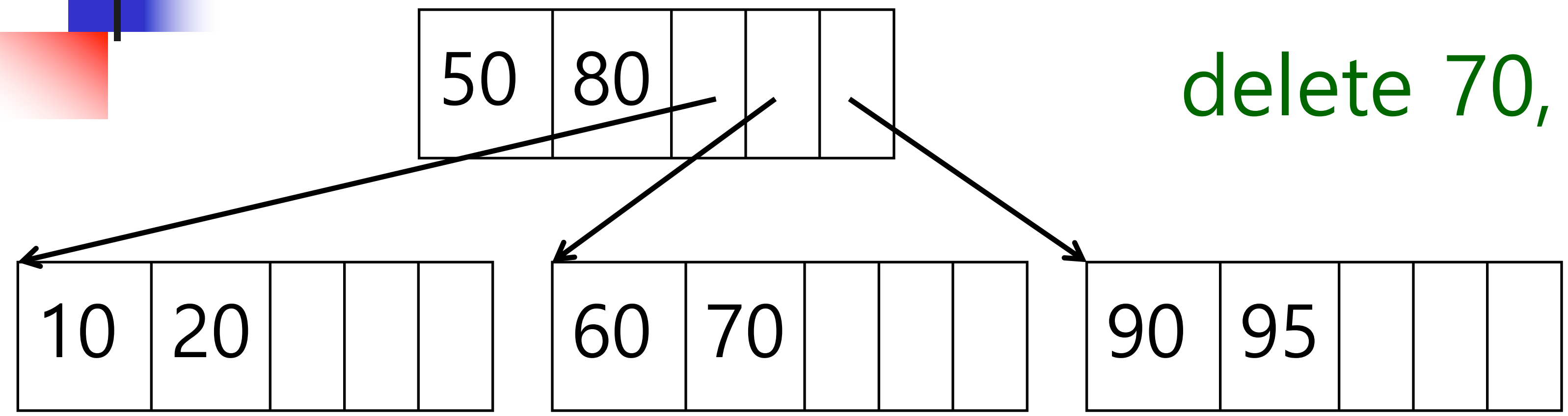
delete 70, 90

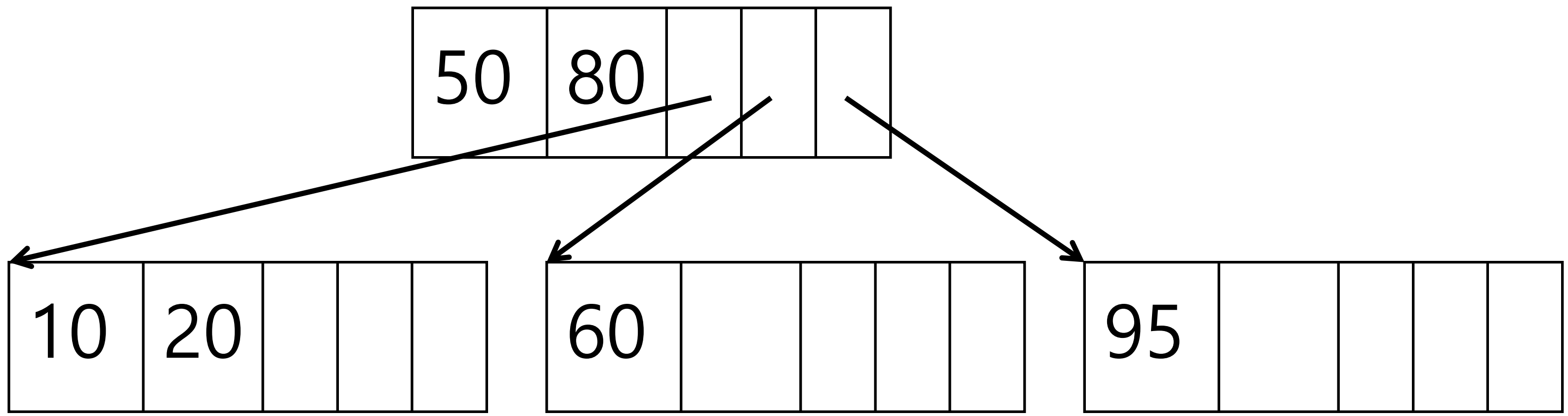




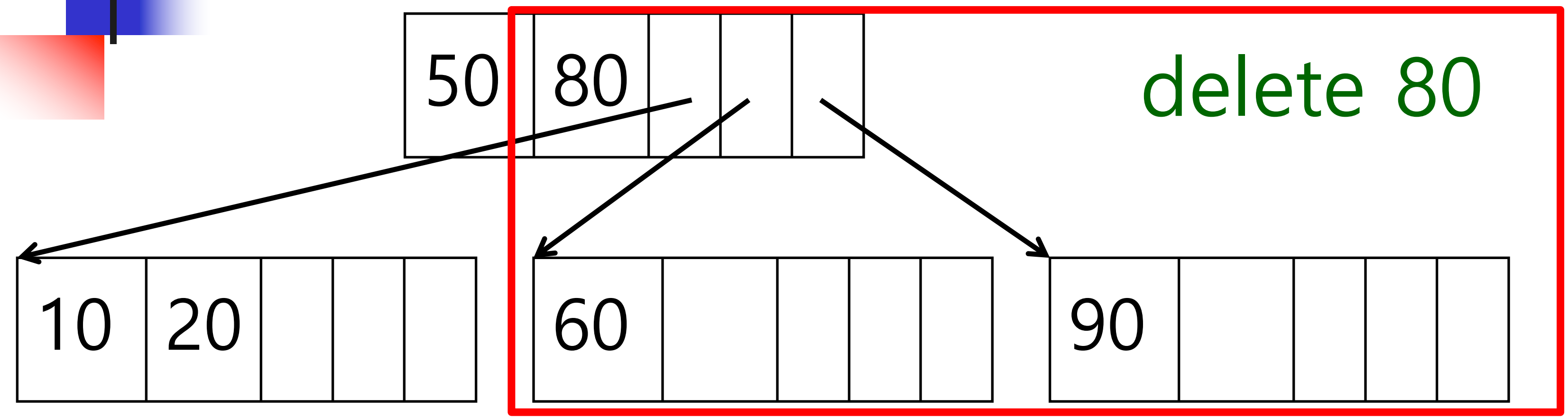
Example 1 (cont'd)

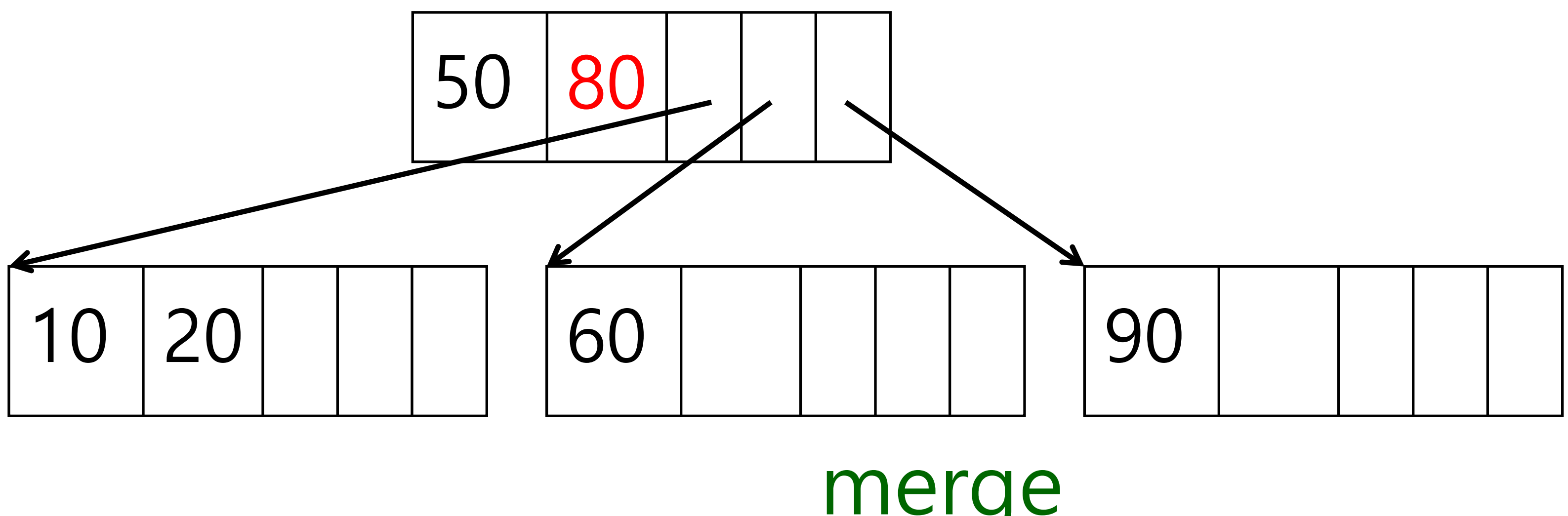
delete 70, 90



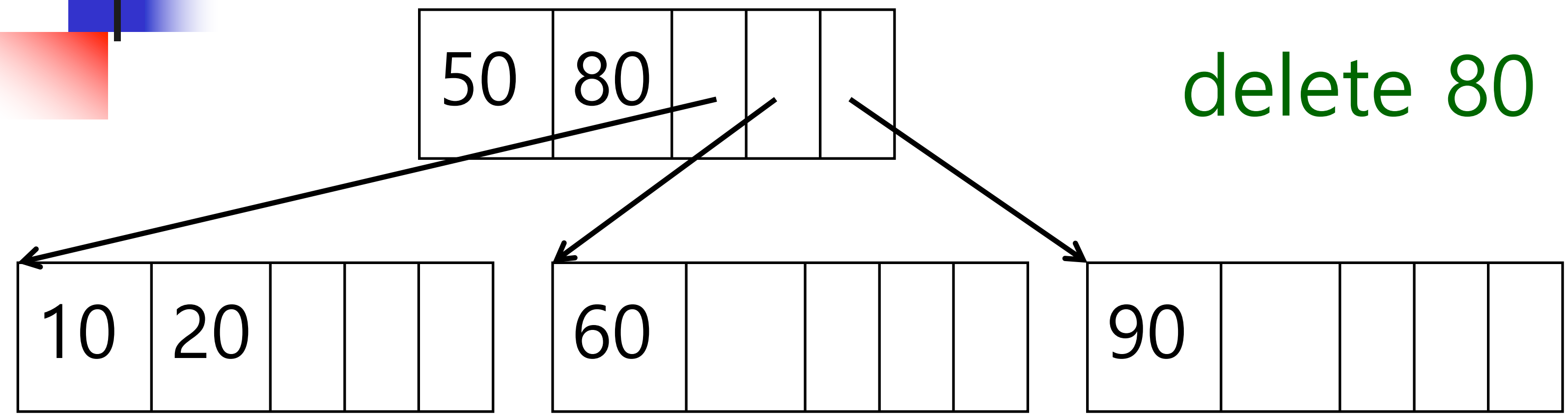


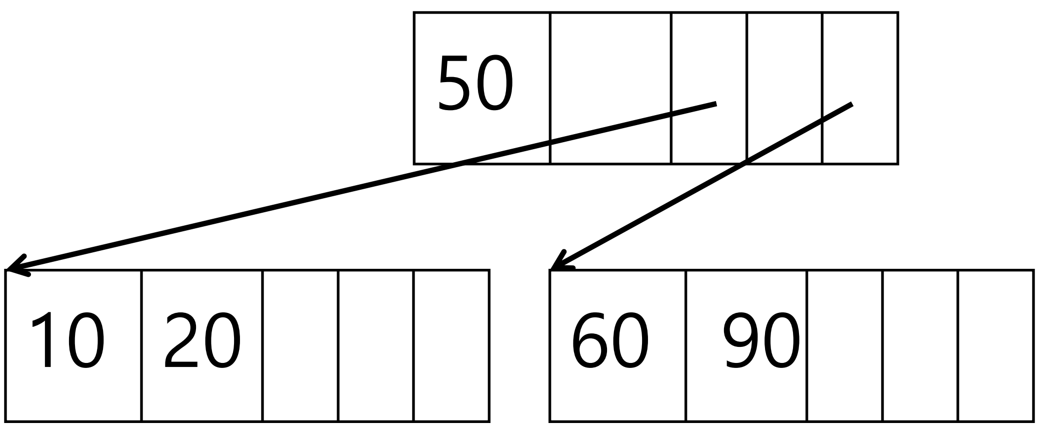
Example 2 (cf. page 34 **)



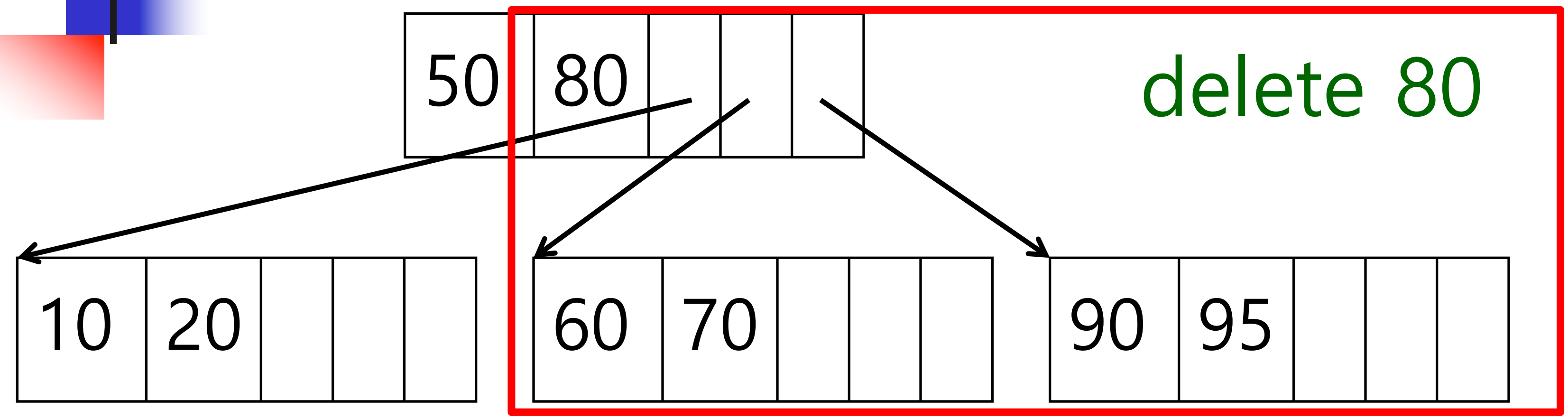


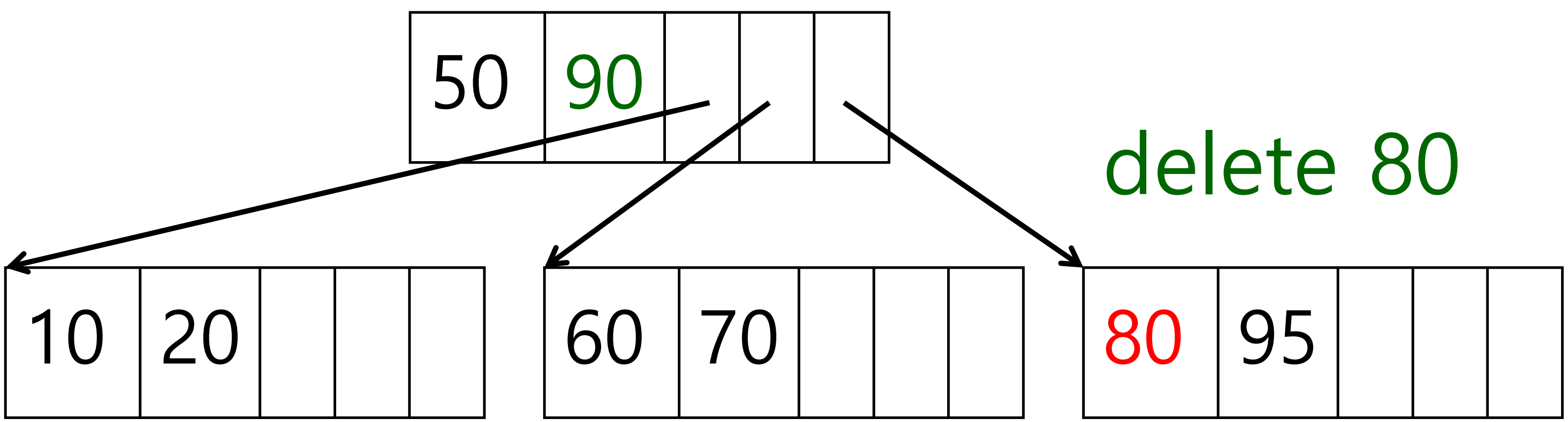
Example 2 (cont'd)





Example 3 (cf. page 34 ***)





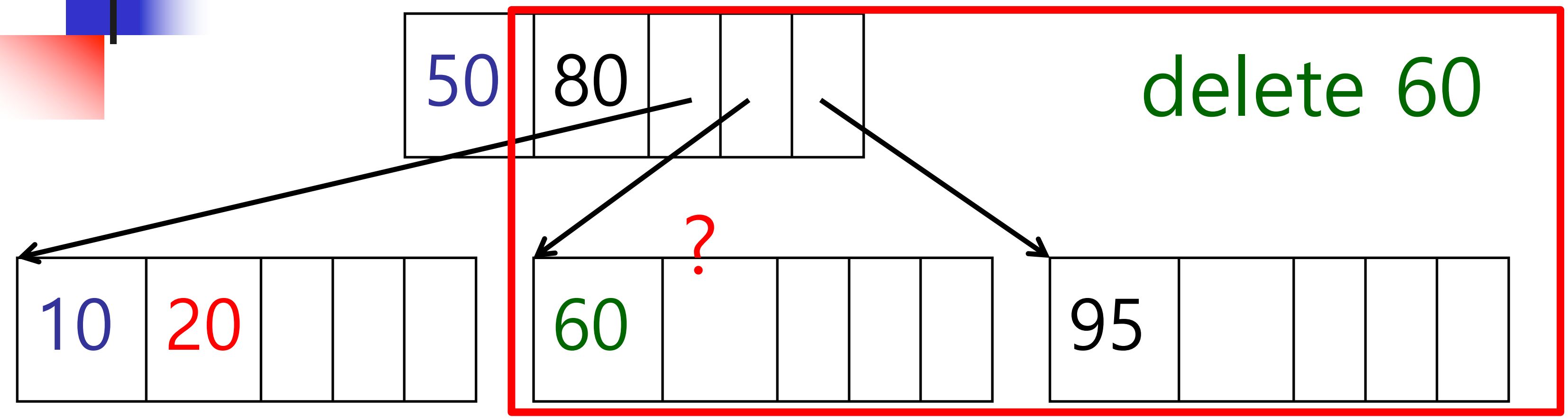
Deleting Data from a 2-Node (1/2)

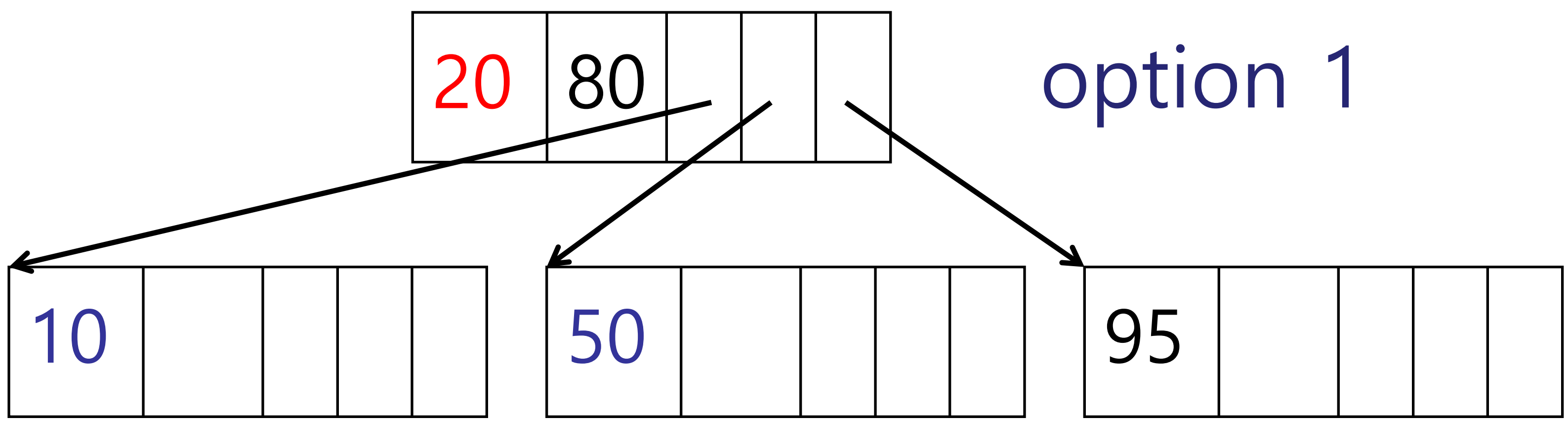
- If there is a sibling 3-node, delete the data in the 2-node (let's call it 2N), and
  - If 2N is the leftmost sibling, and
    - if the middle sibling node is a 3-Node (3N), move the smaller of the parent's data into 2N, and move the smaller of 3N's data into the parent node.
    - if the middle sibling node is a 2-Node, move the smaller of the parent's data into the middle sibling node, and delete 2N.
  - If 2N is the rightmost sibling, and
    - if the middle sibling node is a 3-Node (3N), move the larger of the parent's data into 2N, and move the larger of 3N's data into the parent node.
    - if the middle sibling node is a 2-Node, move the larger of the parent's data into the middle sibling node, and delete 2N.
  - ** If 2N is the middle sibling,
    - ** If the leftmost node is the sibling 3-Node (3N), move the smaller of the parent's data into 2N, and move the larger of 3N's data into the parent node.
    - If the rightmost node is the sibling 3-Node (3N), move the larger of the parent's data into 2N, and move the smaller of 3N's data into the parent node.
  - Adjust the pointers in the sibling node and/or the parent node.

Deleting Data from a 2-Node (2/2)

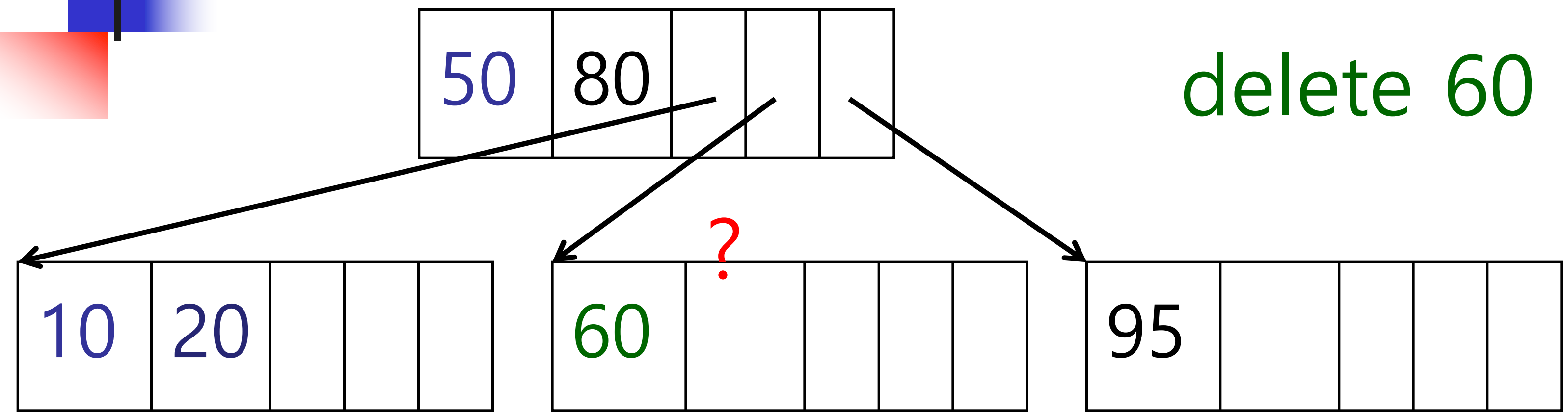
- If there is no sibling 3-node,
  - Move parent's data to the left or right sibling node of the 2-Node (2N), and delete 2N. (The parent node and the sibling node are merged.)
  - If the parent node underflows as a result, take care of the parent node deletion.
  - Adjust the pointers in the sibling node and/or the parent node.

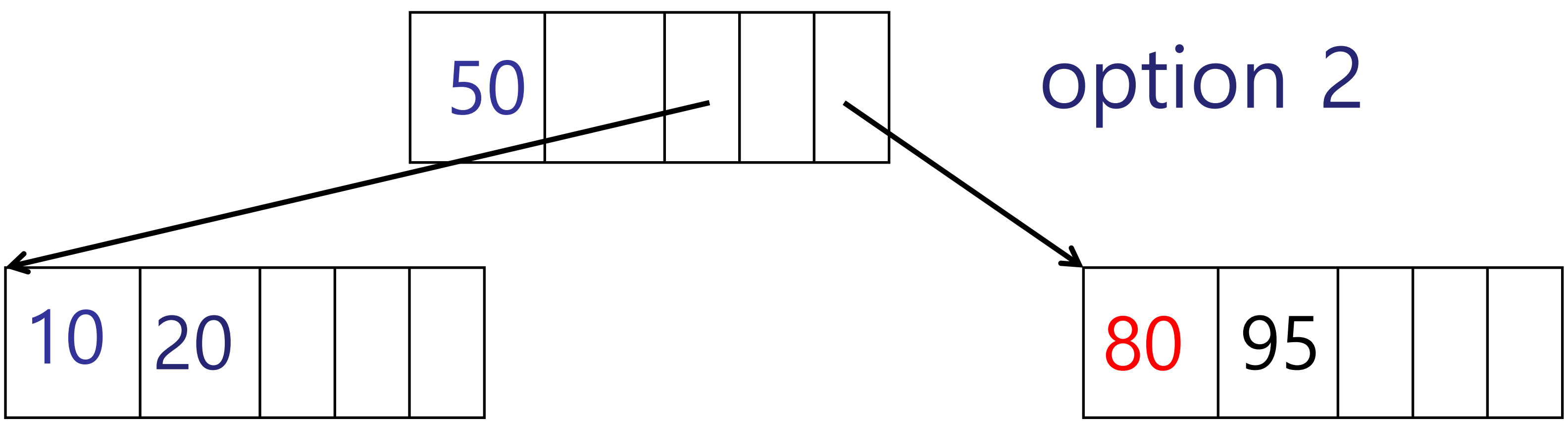
Example 4 (cf. page 40**)



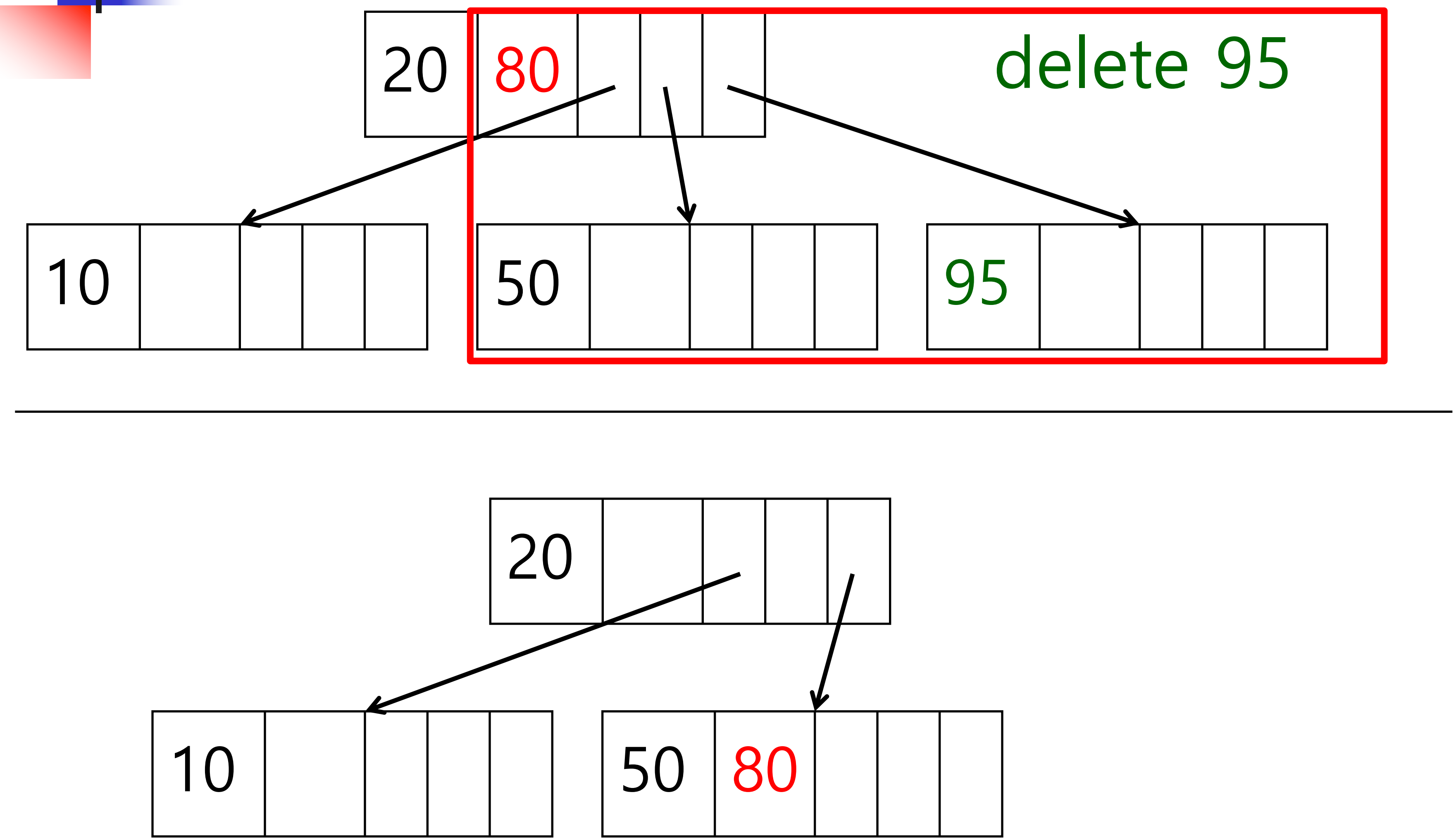


Example 4 (cont'd)

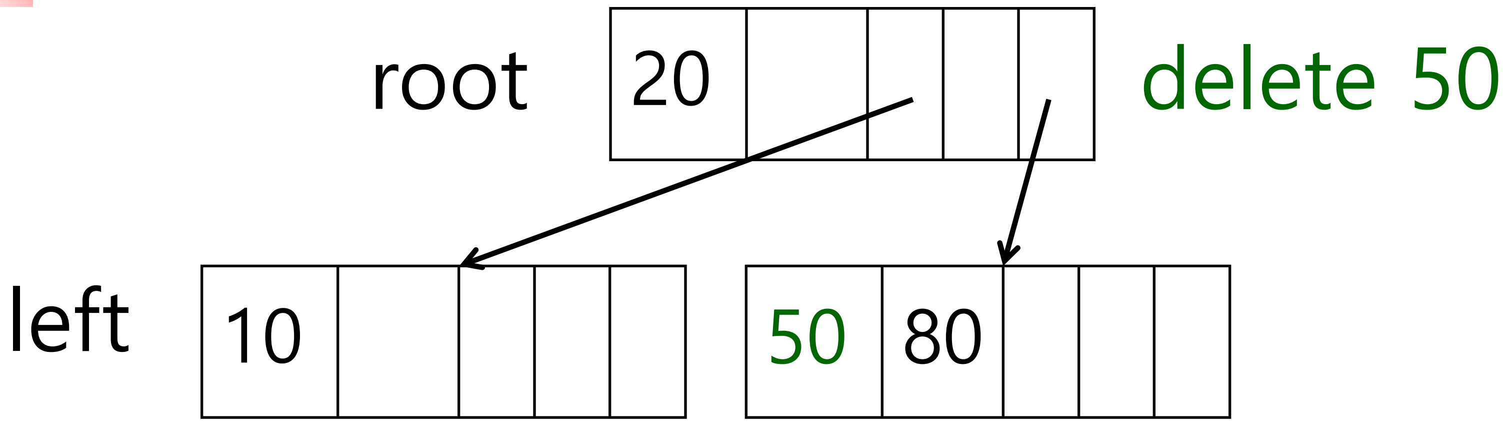




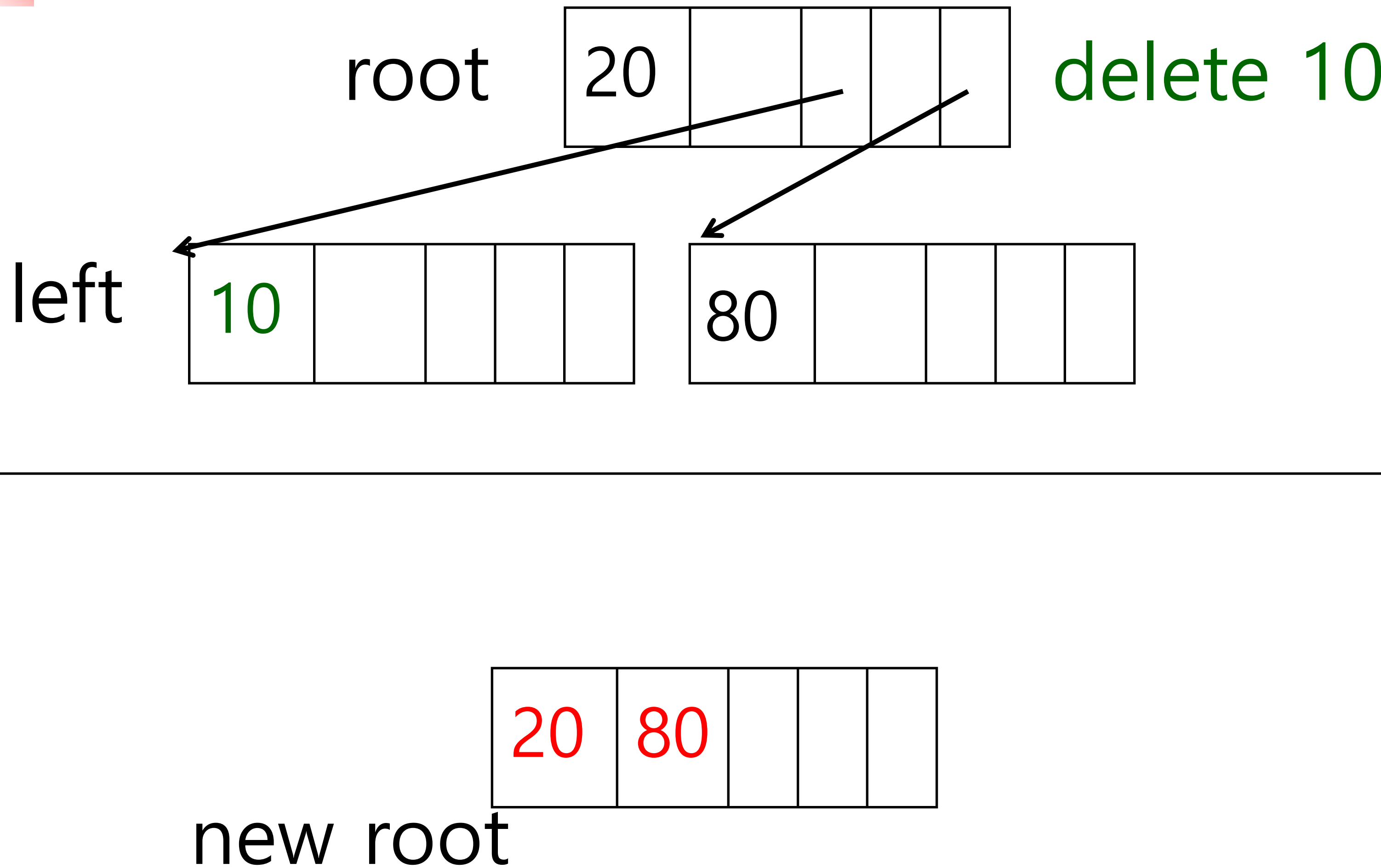
Example 5 (cf. page 41)



Example 6

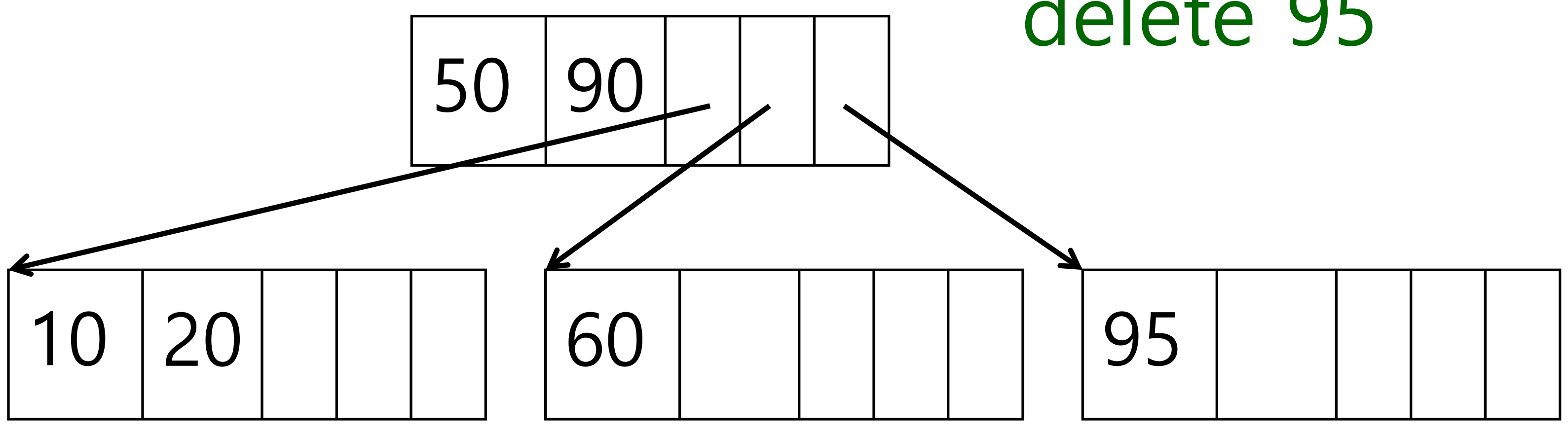


Example 7

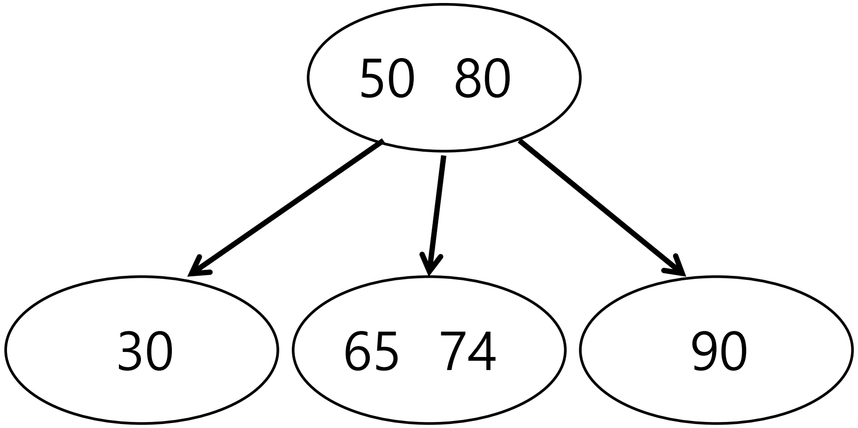


Exercise

delete 90
delete 50
delete 20
delete 95



Exercise : Delete "30" From the Following 2-3 Tree.



Performance of a 2-3 Tree

- Average Case and Worst-Case
  - Between $O(\log_3 n)$ and $O(\log_2 n)$
  - $O(\log_2 n)$: if all nodes are 2-Nodes
  - $O(\log_3 n)$: if all nodes are 3-Nodes

T Tree

T Tree

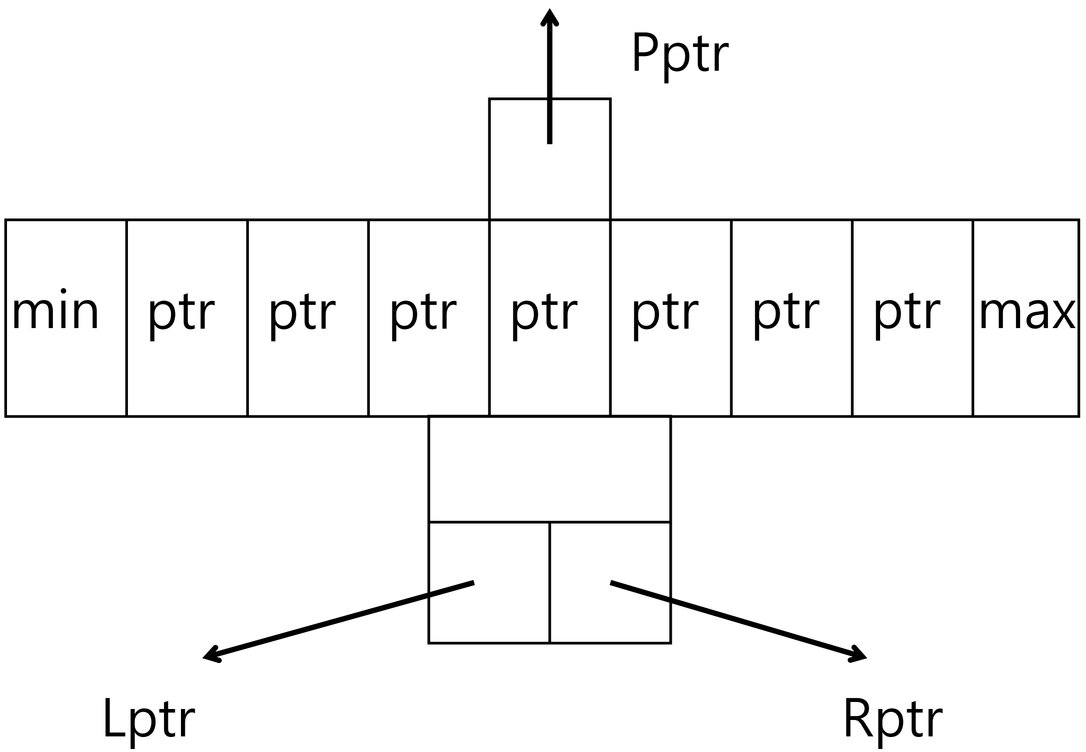
- Combination of AVL Tree and B Tree
  - (borrows from) AVL tree
    - tree rotations for height balancing
    - not perfectly balanced
  - (borrows from) B tree
    - N to 2N data in each node
- Important in main-memory database systems
  - Oracle, MySQL,…
- Reading
  - http://en.wikipedia.org/wiki/T Tree

Each T Tree Node (Implementation)

- An array of N to 2N data
  - (pointers to data in main memory)
  - "2N" is fixed at tree-creation time.
  - Underflow: < N (root node is an exception)
  - Overflow: > 2N
- Pointers to left subtree and right subtree
- Pointer to the parent
- Some control data

Visualization of Each T Tree Node

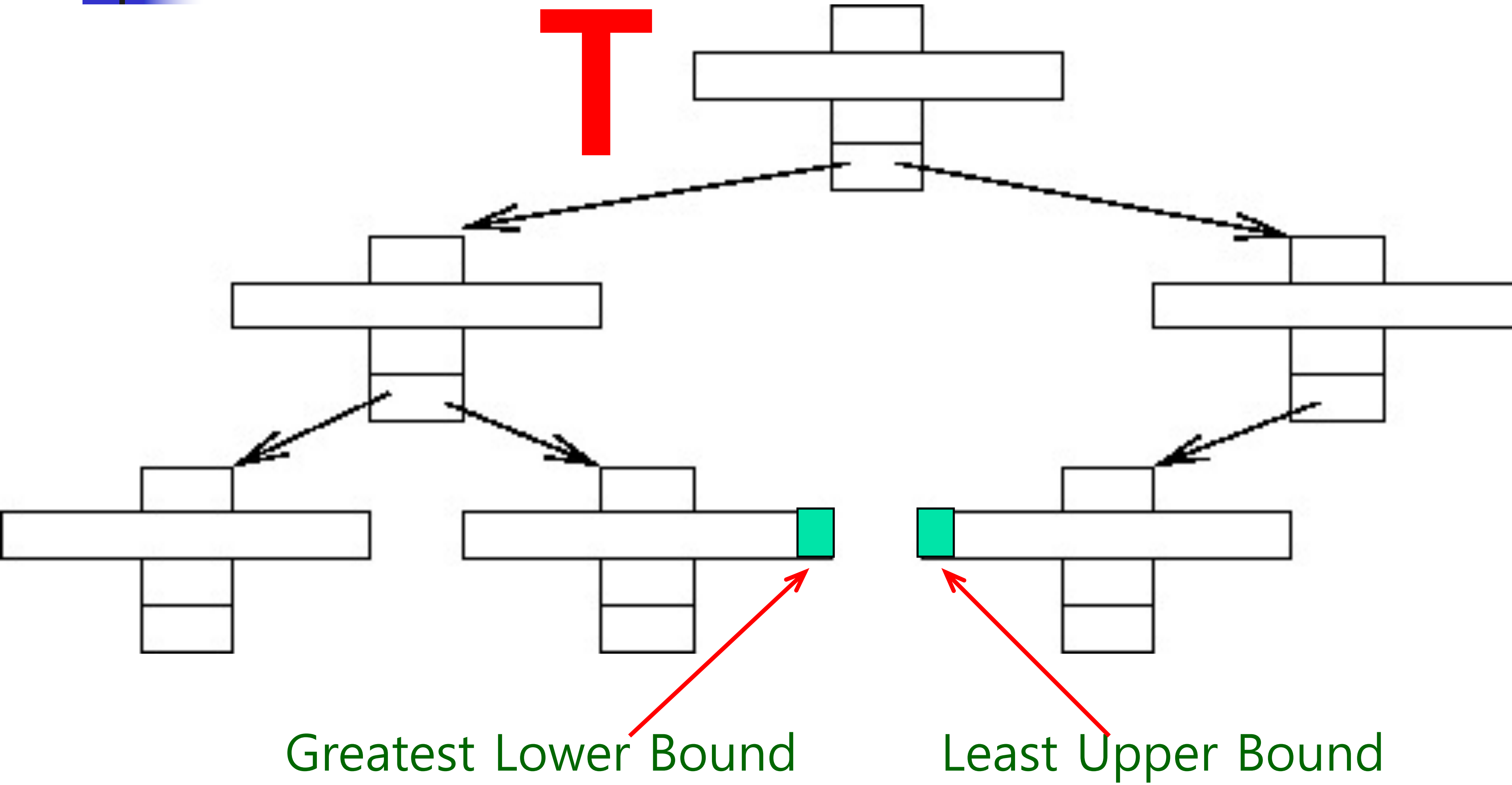
Node



T Tree Organization

- Root Node
- Interior Nodes
  - 2 subtrees
- Half-Leaf Nodes
  - 1 subtree
- Leaf Nodes
  - 0 subtree

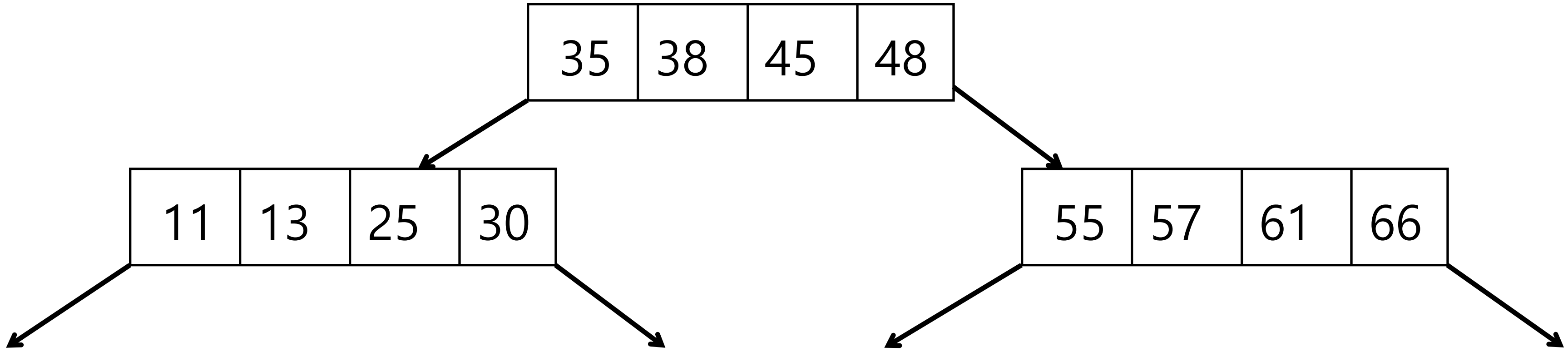
Visualization of a T Tree



Searching

- Search key X, starting at the root node.
- If X < the MIN of the node, search the left subtree.
- If X > the MAX of the node, search the right subtree.
- Otherwise, search the data array on the node.

Example



Leaf Node Overflow

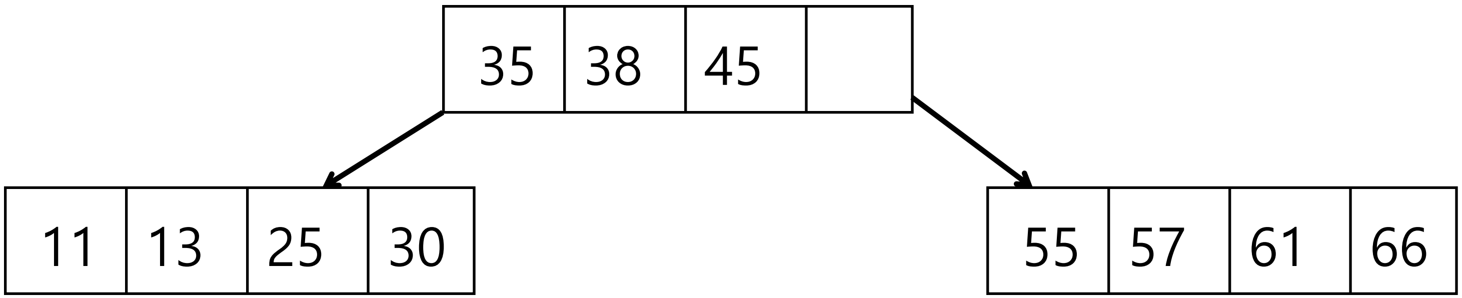
- If a leaf node has 2N + 1 data, split the node.
- Move Left
  - keep the largest N+1 data in the current node, and
  - move the smallest N data to a new left child leaf node
- Move Right
  - keep the smallest N+1 data in the current node, and
  - move the largest N data to a new right child leaf node

Inserting

- Search key X, starting at the root node.
- If X is found, finish.
- Insert in a node where search fails.
- If there is room, insert X. Finish.
- If there is no room,
- Either (move left)
  - if the current node is a leaf node, split the node. Finish.
  - else (remove the smallest data, and insert x in the node.
  - insert the removed data into the “greatest lower bound”
  - leaf node. If the leaf node overflows, split the leaf node. Finish.)
- Or (move right)
  - if the current node is a leaf node, split the node. Finish.
  - else (remove the largest data, and insert x in the node.
  - insert the removed data into the “least upper bound”
  - leaf node. If the leaf node overflows, split the leaf node. Finish.)
- If the tree is out of balance, perform tree rotations. Finish.

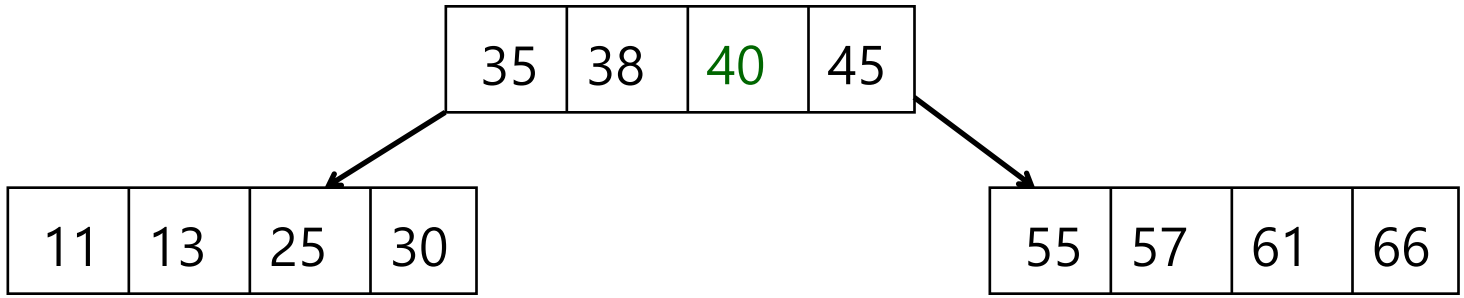
Example 1

insert 40



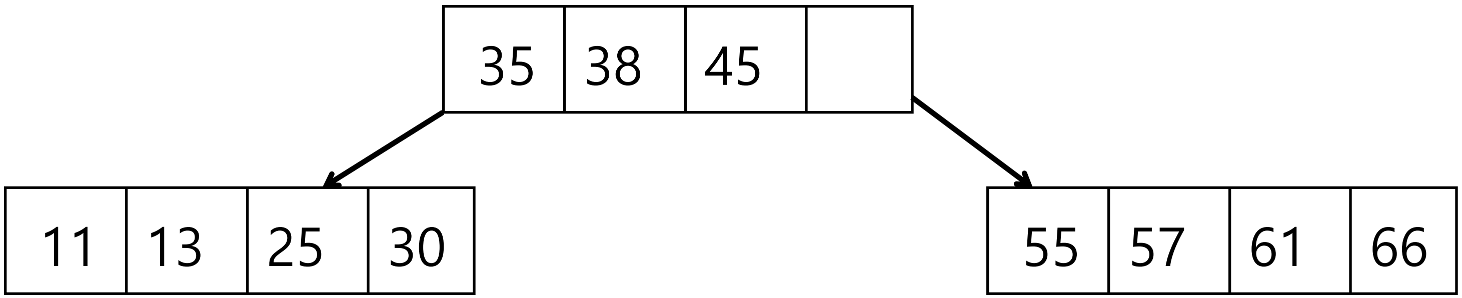
Example 1 (cont'd)

insert 40



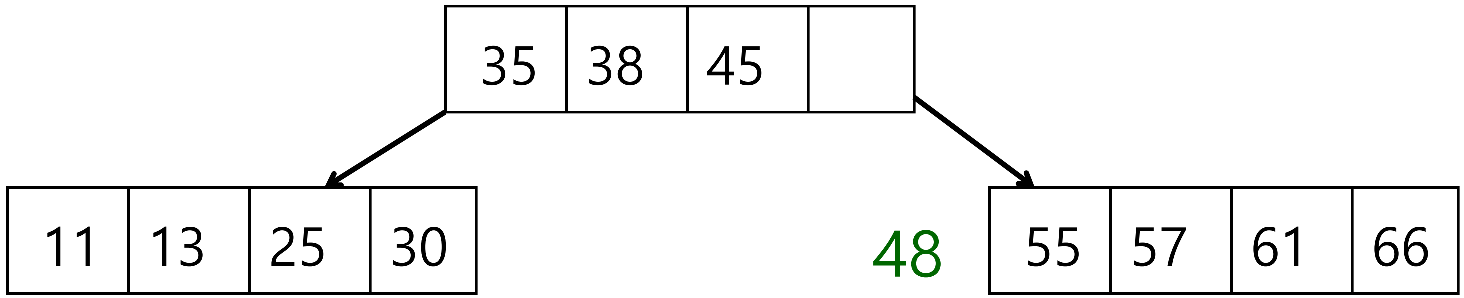
Example 2 (leaf node overflow)

insert 48



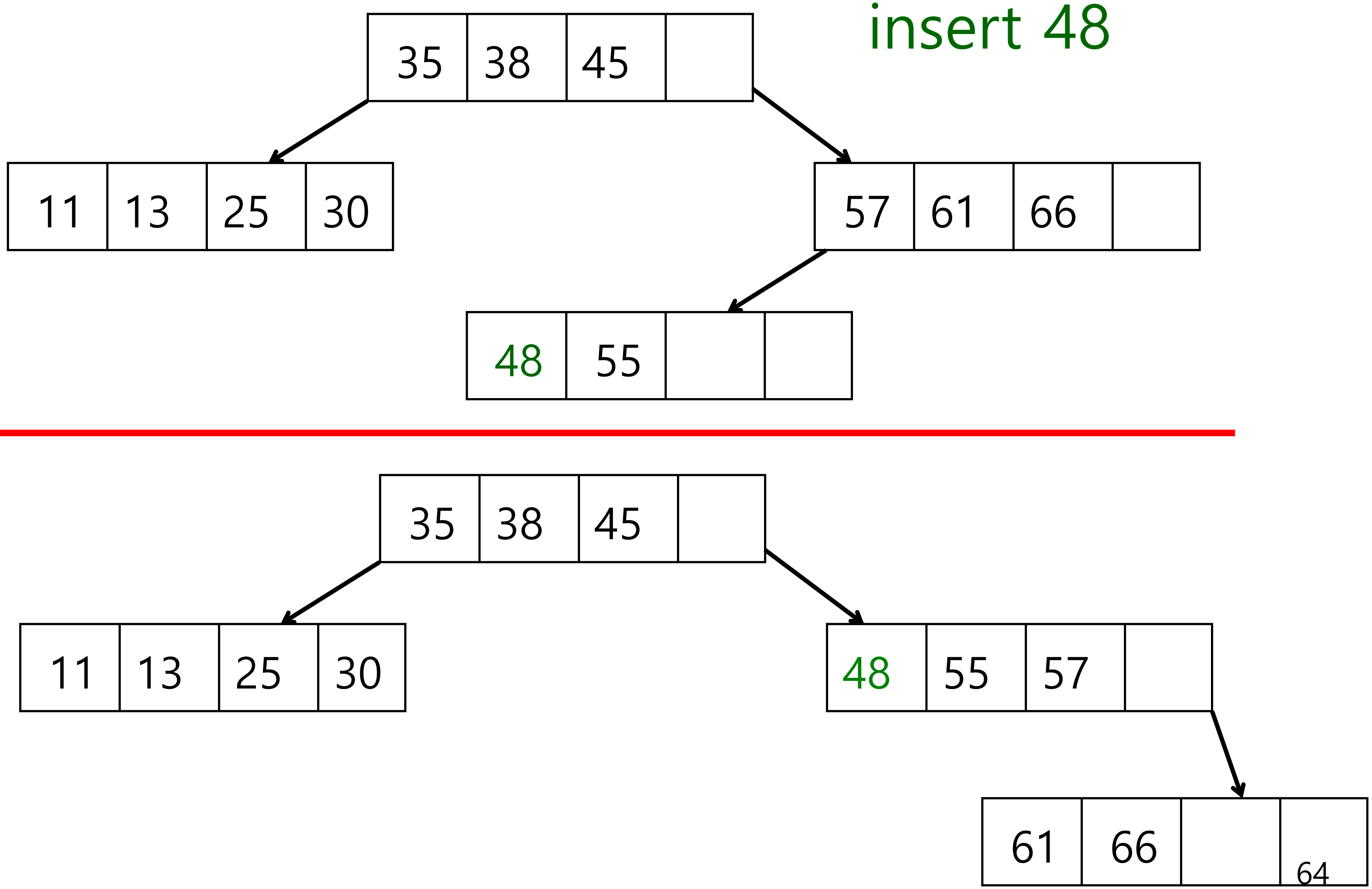
Example 2 (cont'd)

insert 48



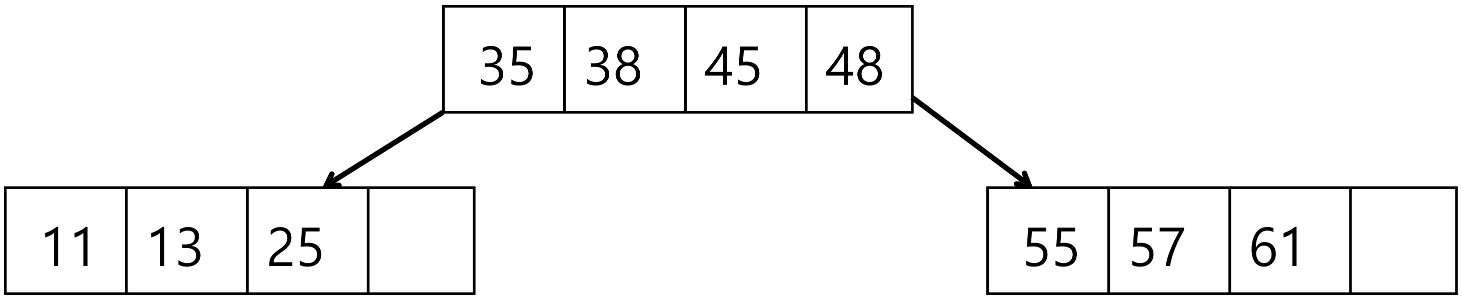
Example 2 (cont'd)

insert 48



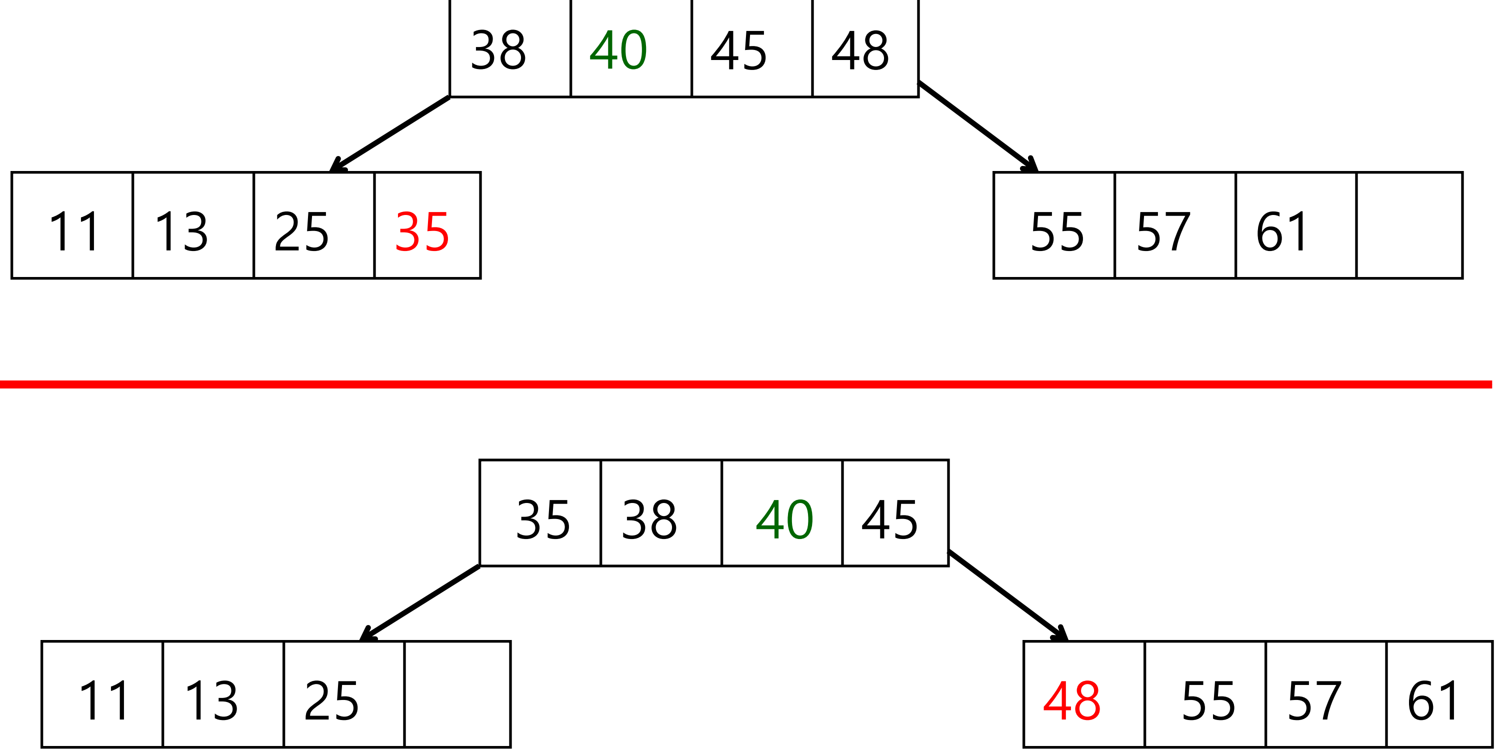
Example 3 (non-leaf node overflow)

insert 40



Example 3 (cont'd)

insert 40



Leaf Node Underflow

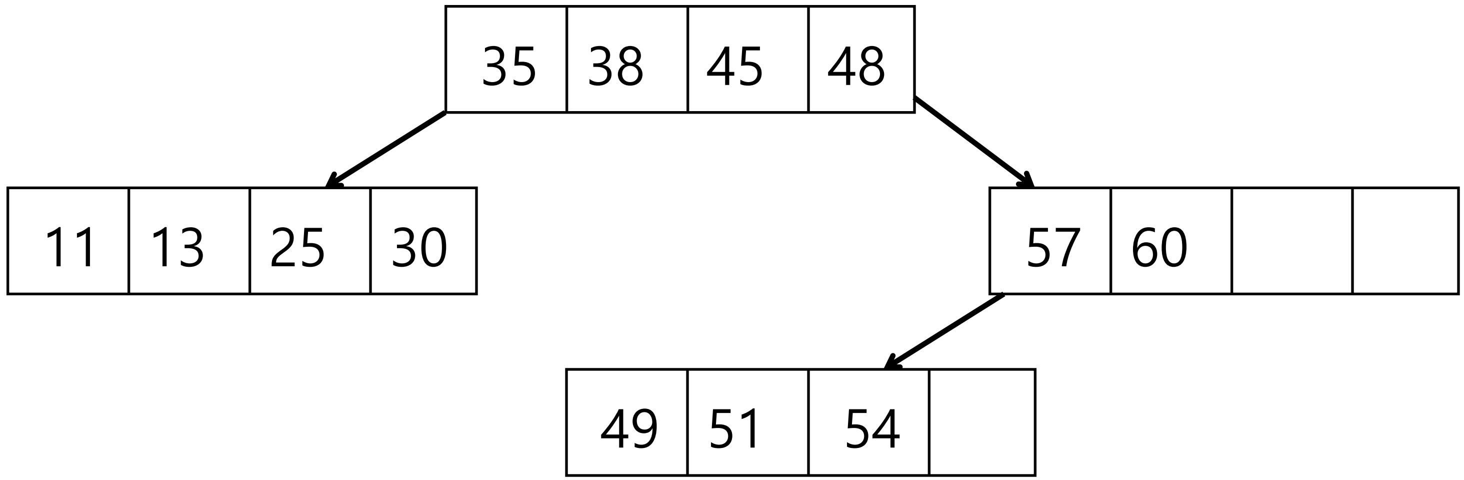
- If a leaf node has N-1 data, merge the node with its parent node.
- If the parent node overflows as a result, split the parent node.

Deleting

- Search key X, starting at the root node.
- If X is not found, finish.
- If X is found, delete it.
  - If X was in a leaf node, and the node underflows, merge the node with the parent node.
  - If X was in an interior node and the node underflows, replace X with the largest data from the left subtree, or the smallest data from the right subtree.
  - If the tree becomes out of balance (the balance factor of any node becomes +2 or -2), perform tree rotations.

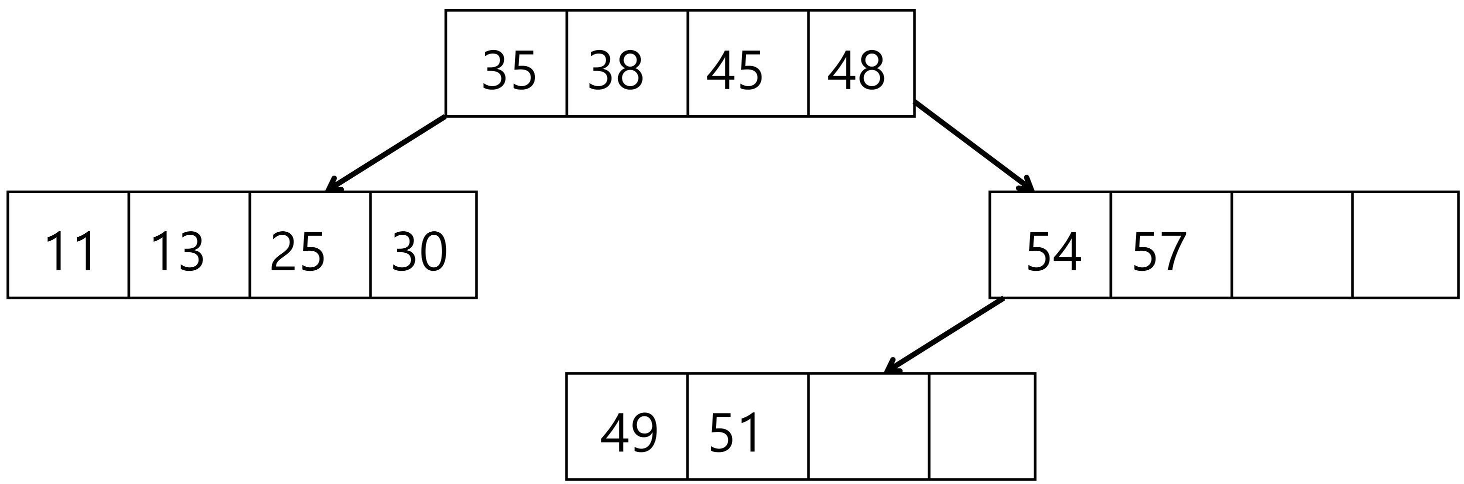
Example 1 (interior node underflow)

delete 60



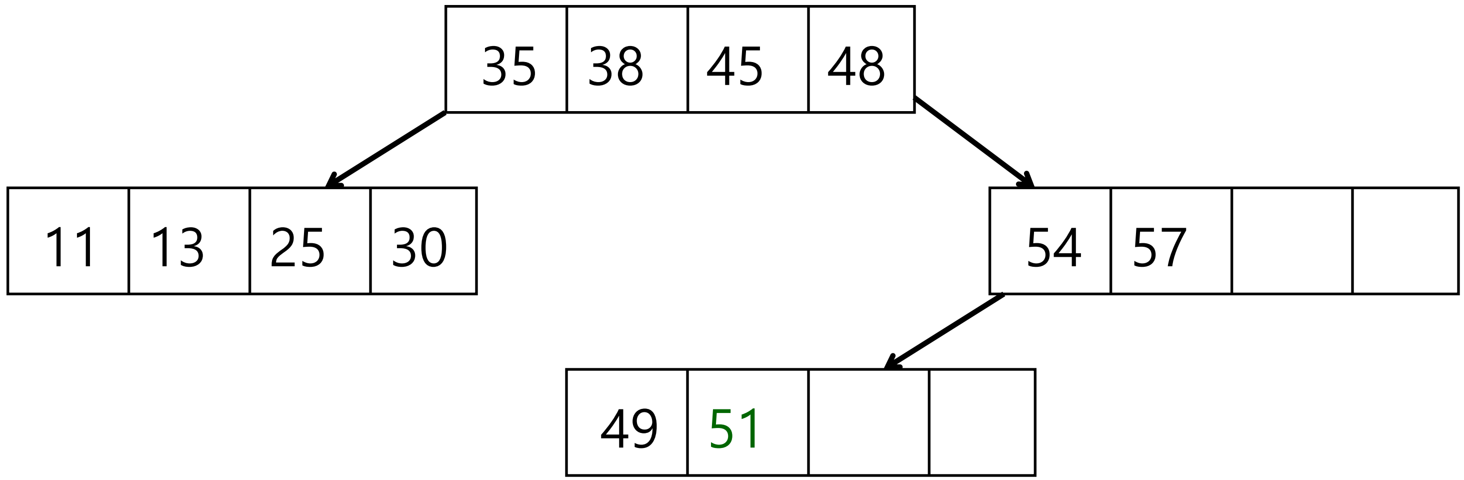
Example 1 (cont'd)

delete 60



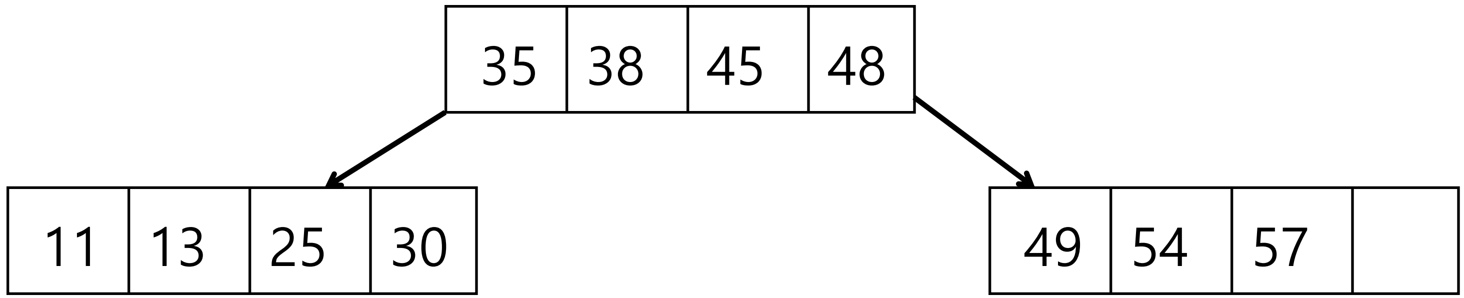
Example 2 (leaf node underflow)

delete 51



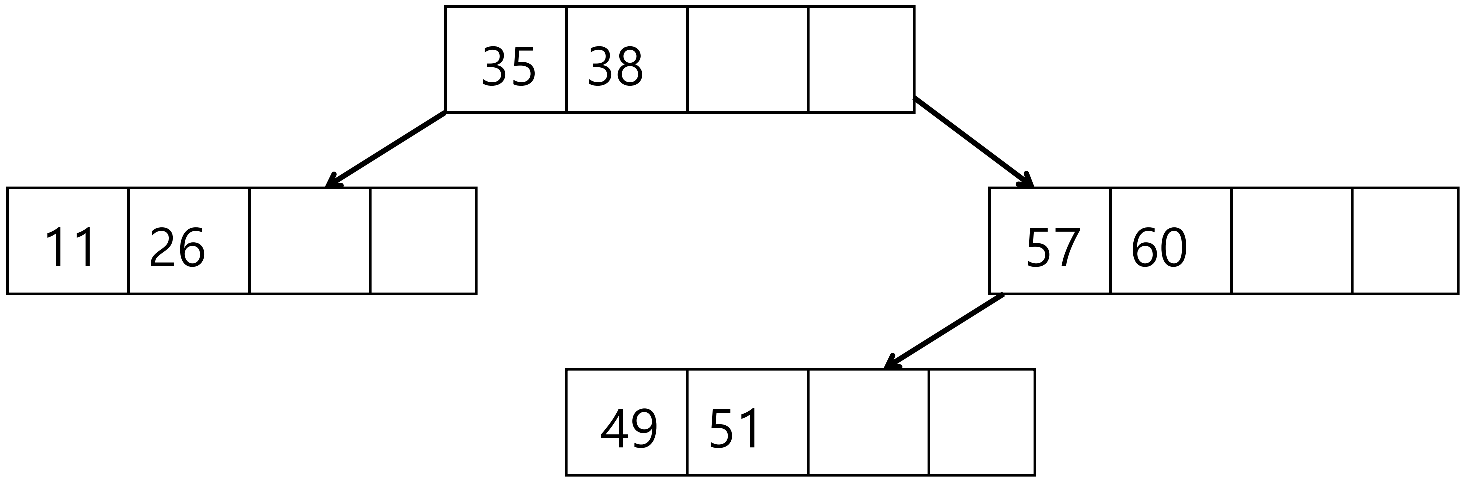
Example 2 (cont'd)

delete 51



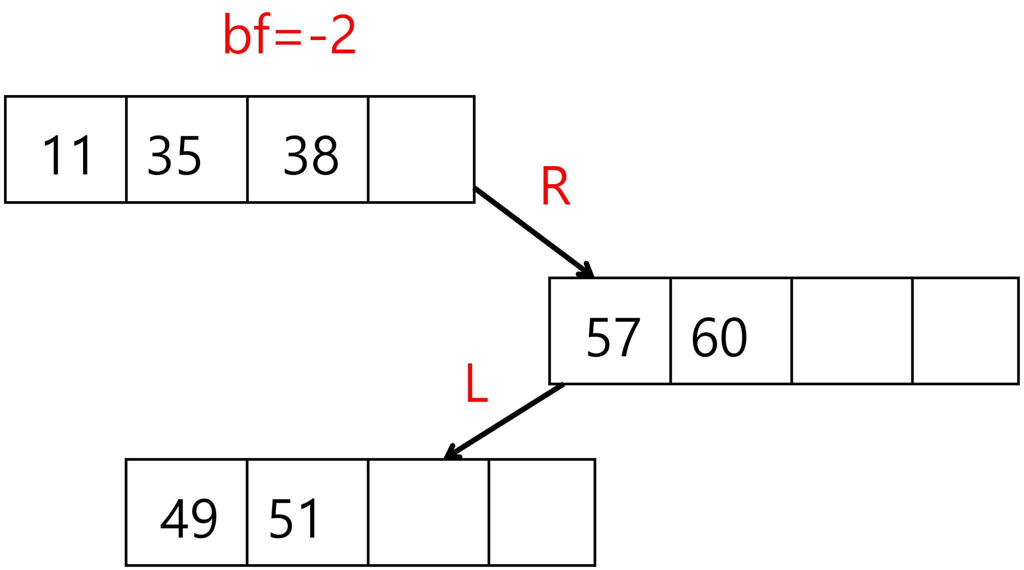
Example 3: Tree Rotation

delete 26

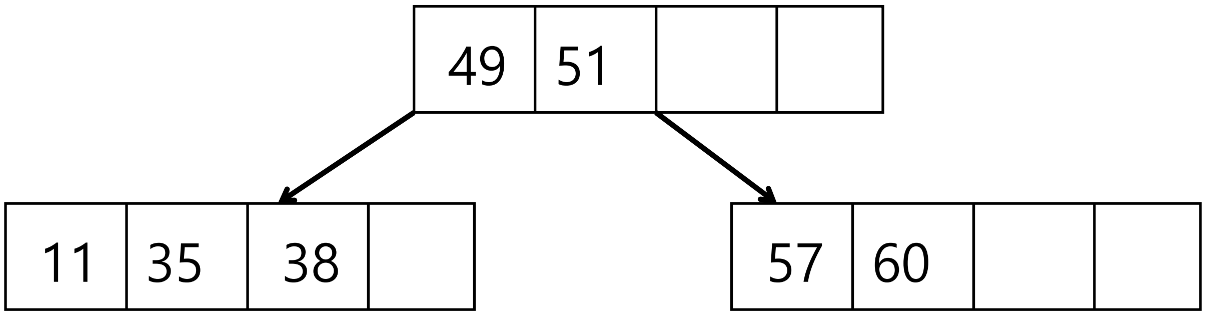


Example 3: (cont'd)

delete 26

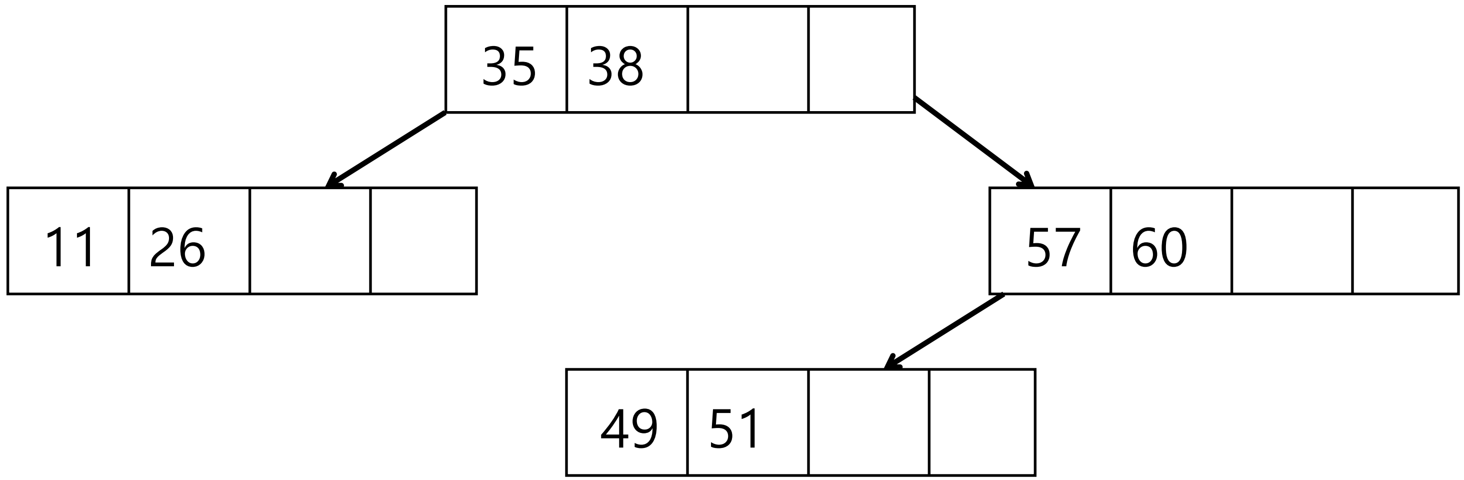


Example 3: (cont'd) RL Rotation



Exercise: Delete a Key From a T Tree

delete 38



Performance Properties of a T Tree

- Reduced tree height
  - $\log_2 [N/M]$
  - N = total number of keys, M = number of keys per node
- Node split and merge
- The usual problems of the array for the keys in each node
- Maintaining Min, Max key values in each node

WHW 3-1 (20 points)

- Construct a 2-3 Tree with keys D, A1, T1, A2, S, T2, R1, U1, C, T3, U2, R2, E
  - in the given order, starting from an empty tree.
  - (you must show each insert and each node split)
- From the constructed 2-3 Tree, delete the nodes with keys A1, T1, T2, T3
  - in the given order.
  - (you must show each delete and each node merge)

WHW 3-2 (14 points)

- Construct a T-Tree (where M=2) with keys 20, 80, 60, 40, 15, 25, 30, 35
  - in the given order, starting from an empty tree.
  - (you must show each insert; node split and tree rotation)
- From the constructed T-Tree, delete the nodes with keys 20, 35, 60, 80
  - in the given order.
  - (you must show each delete; node merge and tree rotation)

WHW 3-3 (6 points)

- Discuss the tradeoffs between an AVL tree and a T tree. (Need trend and summary comparison)
- (hint: Compare the two in terms of performance, memory requirements, and insert/delete processing overhead; work with N = 100, 1000, 100000, for example)

End of Lecture