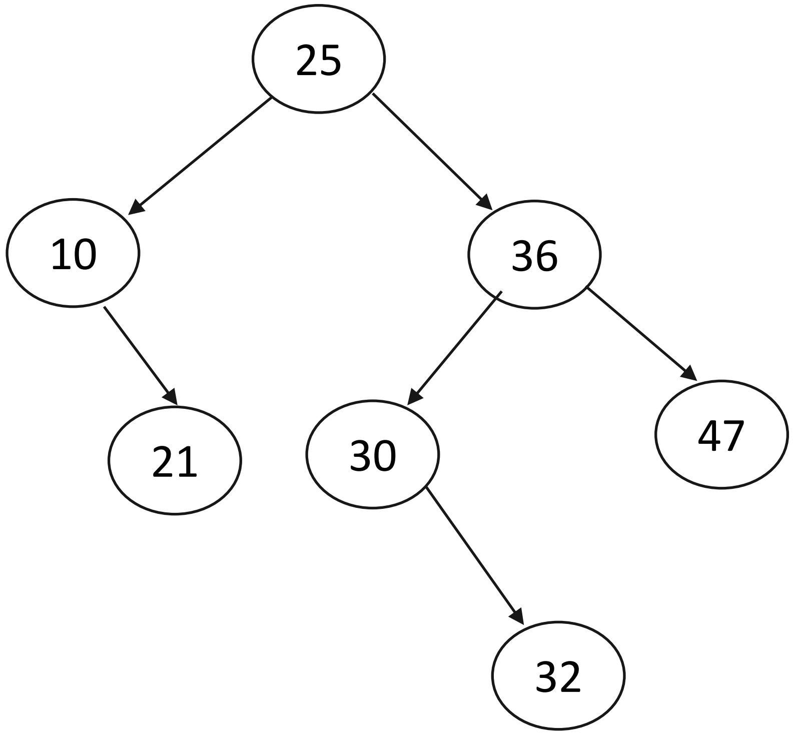
Balanced trees in distributed environments

Balanced trees hate many threads

- We have observed that the process of inserting or deleting nodes into those trees can sometimes change the structure of neighboring nodes, or even the entire tree, ***dramatically***.
- This means that if we have multiple threads, looking at different positions in the tree, then what those threads "see" can change dramatically when those threads resume control of the CPU!
- Let's look at some examples.

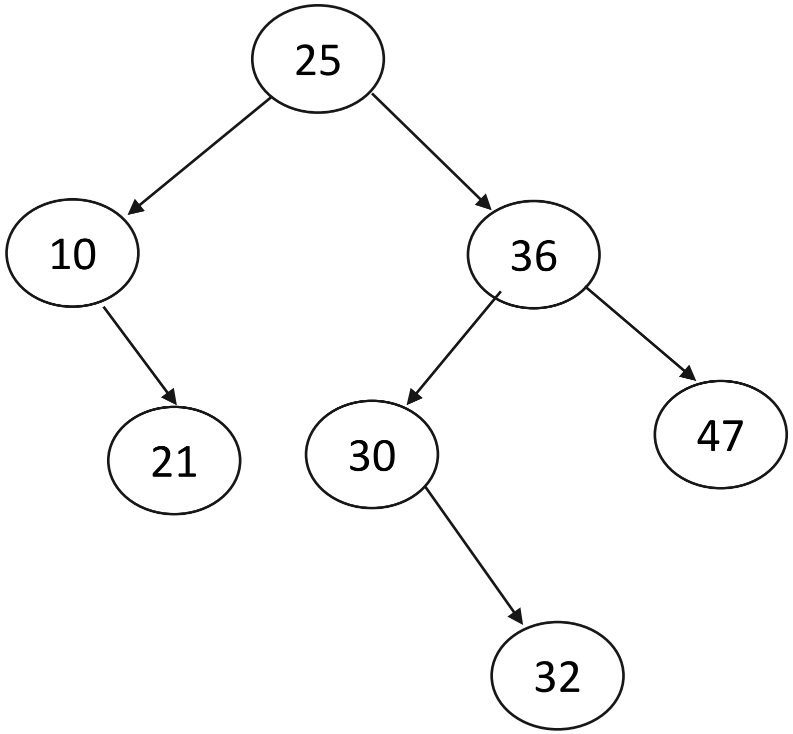
AVL Tree Example: Search and Insert



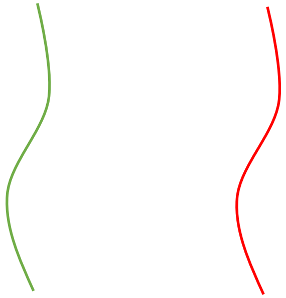
T_1: `search(32)` T_2: `insert(31)`

Both threads should succeed in their goal!

AVL Tree Example: Search and Insert

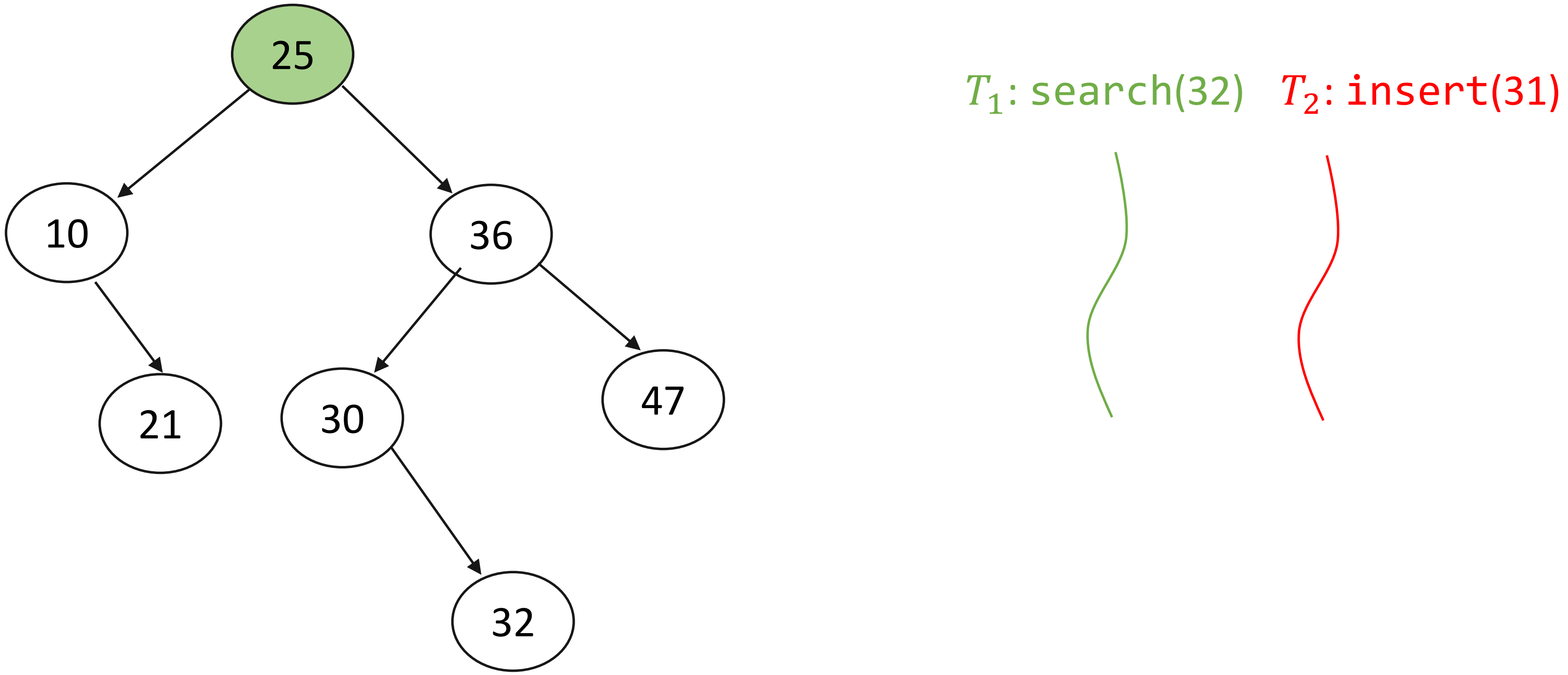


T_1: `search(32)` T_2: `insert(31)`

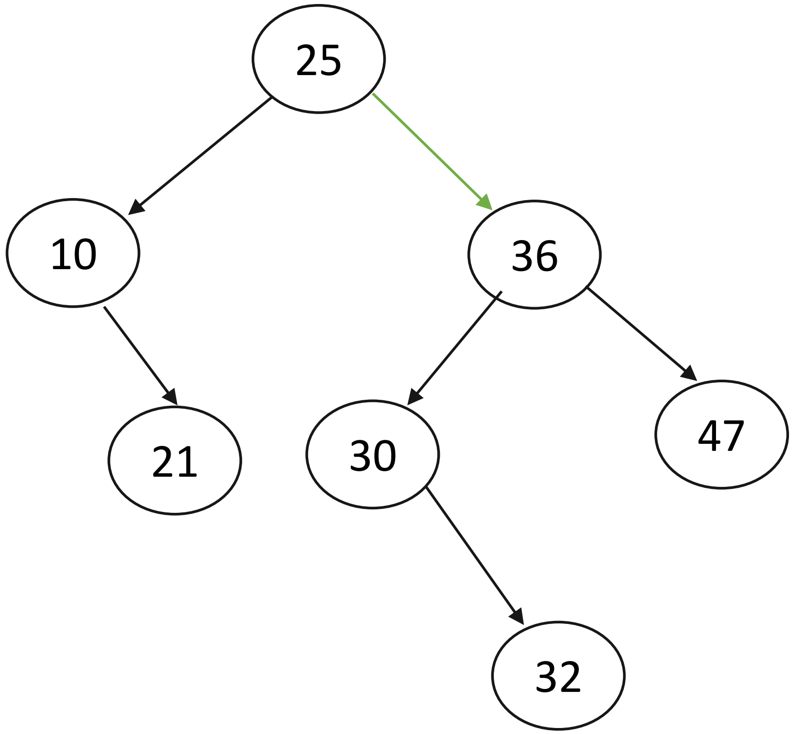


Suppose T_1 is the first thread to claim the CPU.

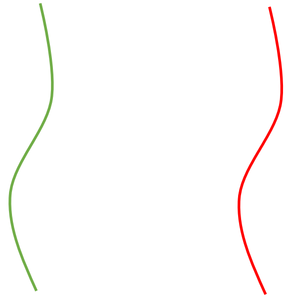
AVL Tree Example: Search and Insert



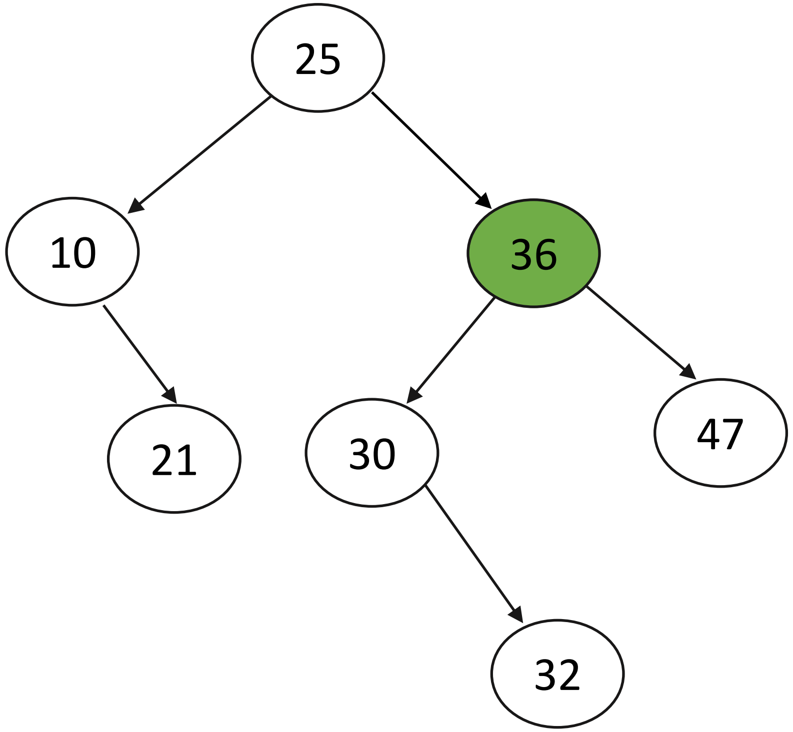
AVL Tree Example: Search and Insert



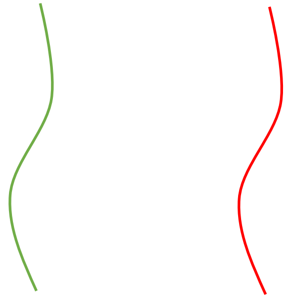
T_1: `search(32)` T_2: `insert(31)`



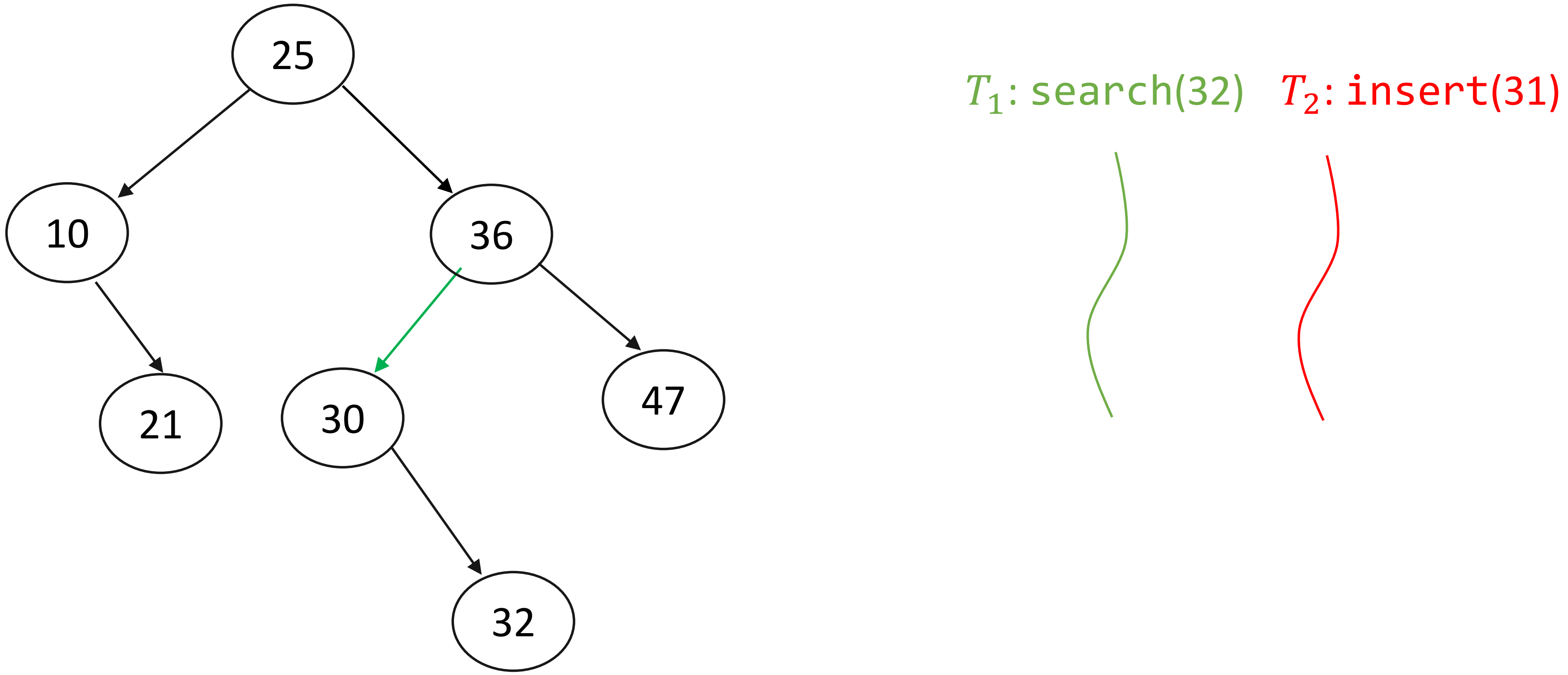
AVL Tree Example: Search and Insert



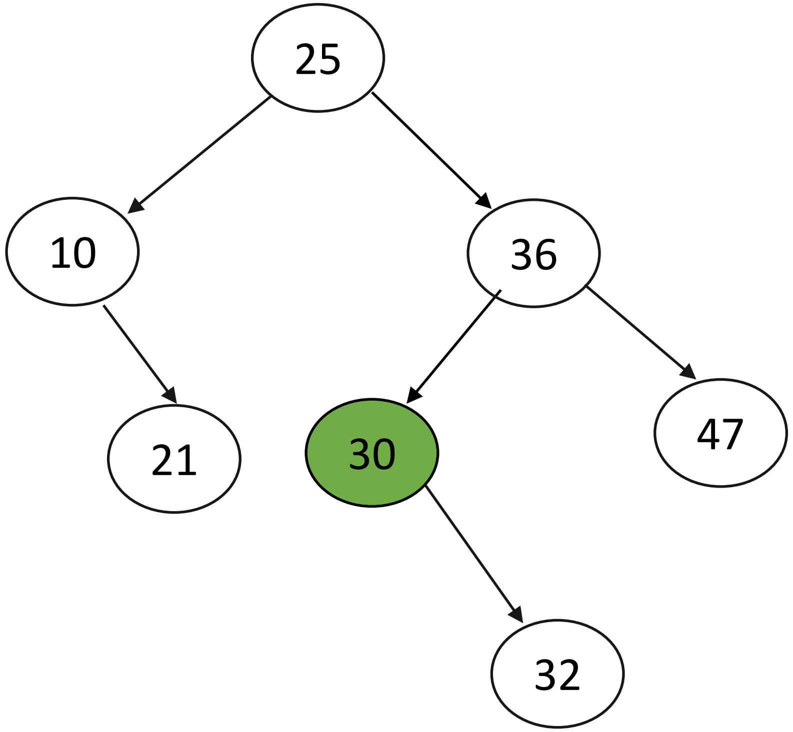
T_1: search(32) T_2: insert(31)



AVL Tree Example: Search and Insert

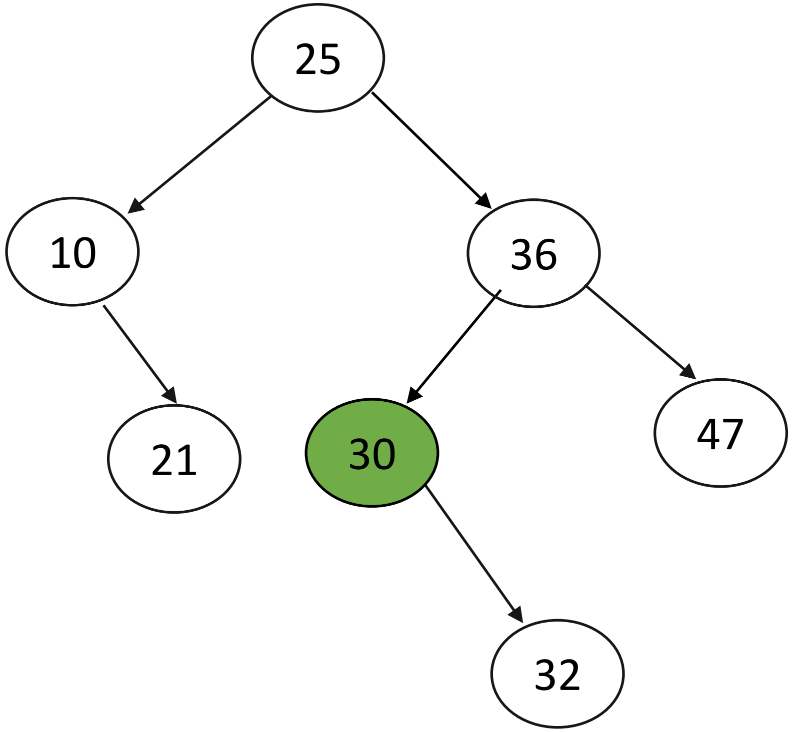


AVL Tree Example: Search and Insert



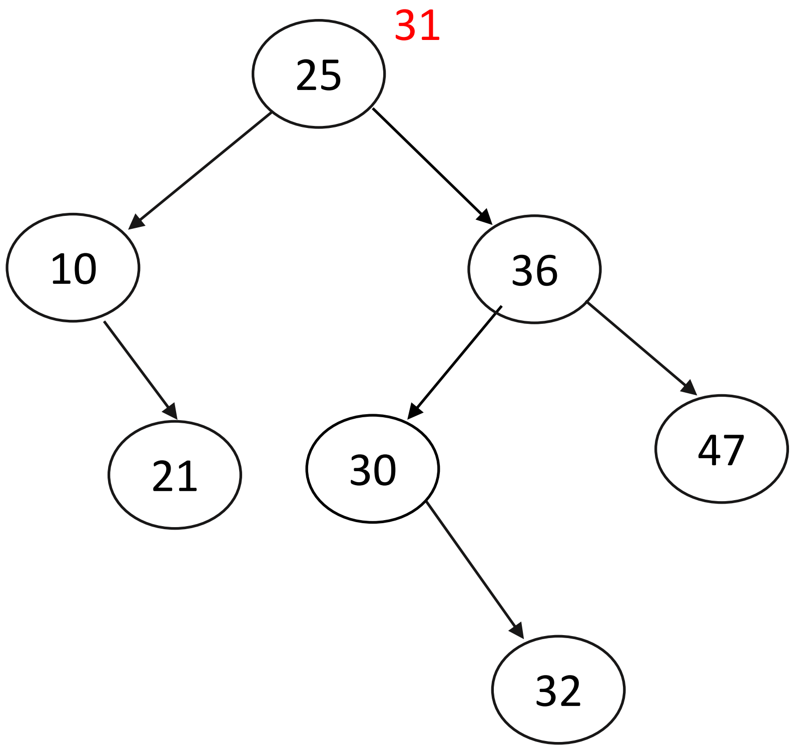
T_1: `search(32)` T_2: `insert(31)`

AVL Tree Example: Search and Insert

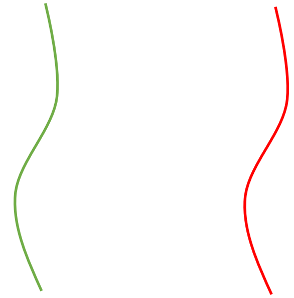


Now suppose that T_2 takes the CPU.

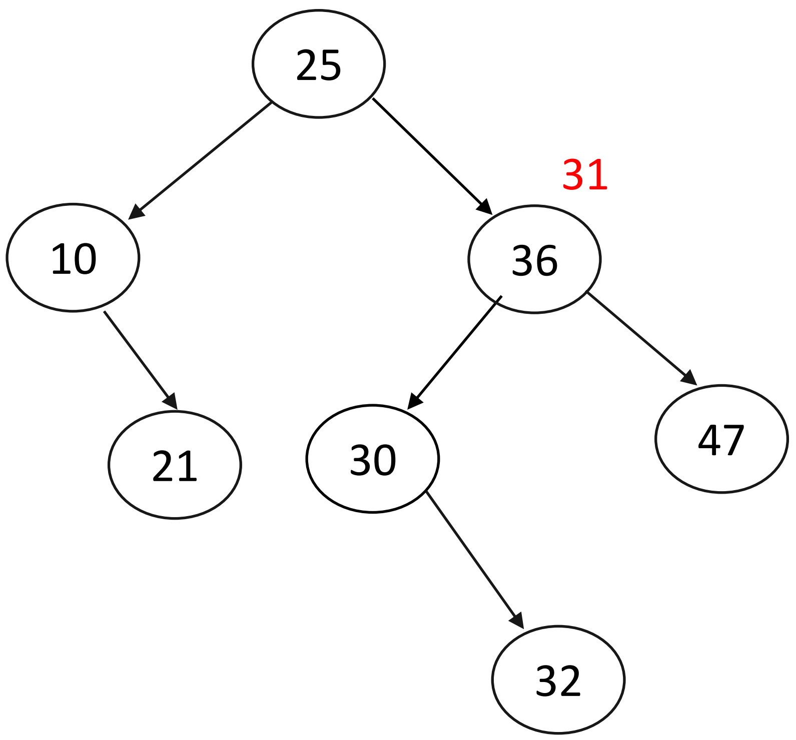
AVL Tree Example: Search and Insert



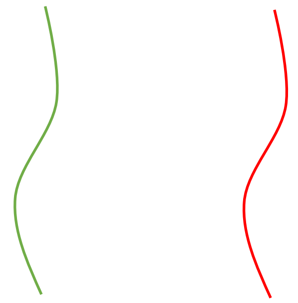
T_1: search(32) T_2: insert(31)



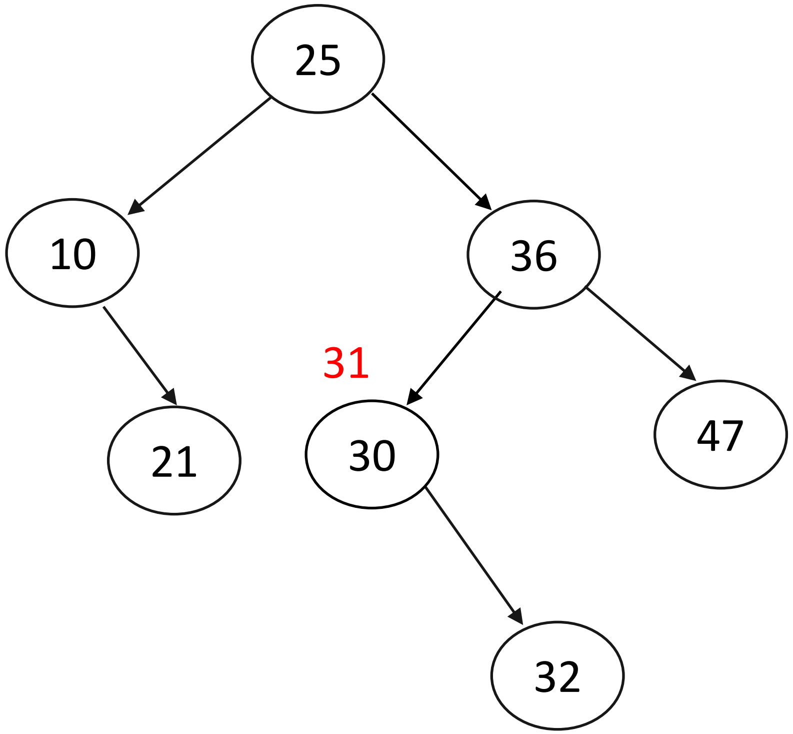
AVL Tree Example: Search and Insert



T_1: search(32) T_2: insert(31)

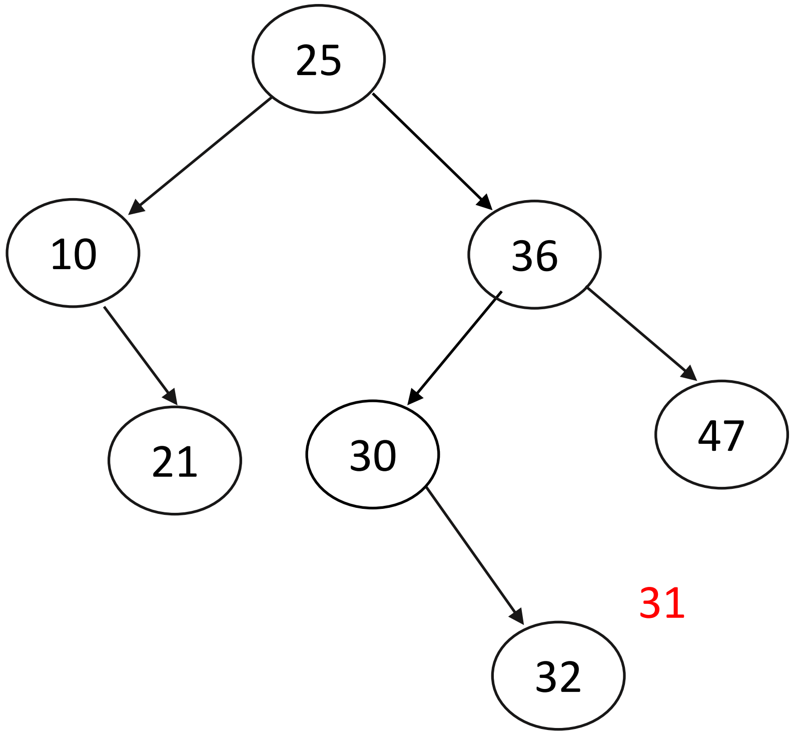


AVL Tree Example: Search and Insert

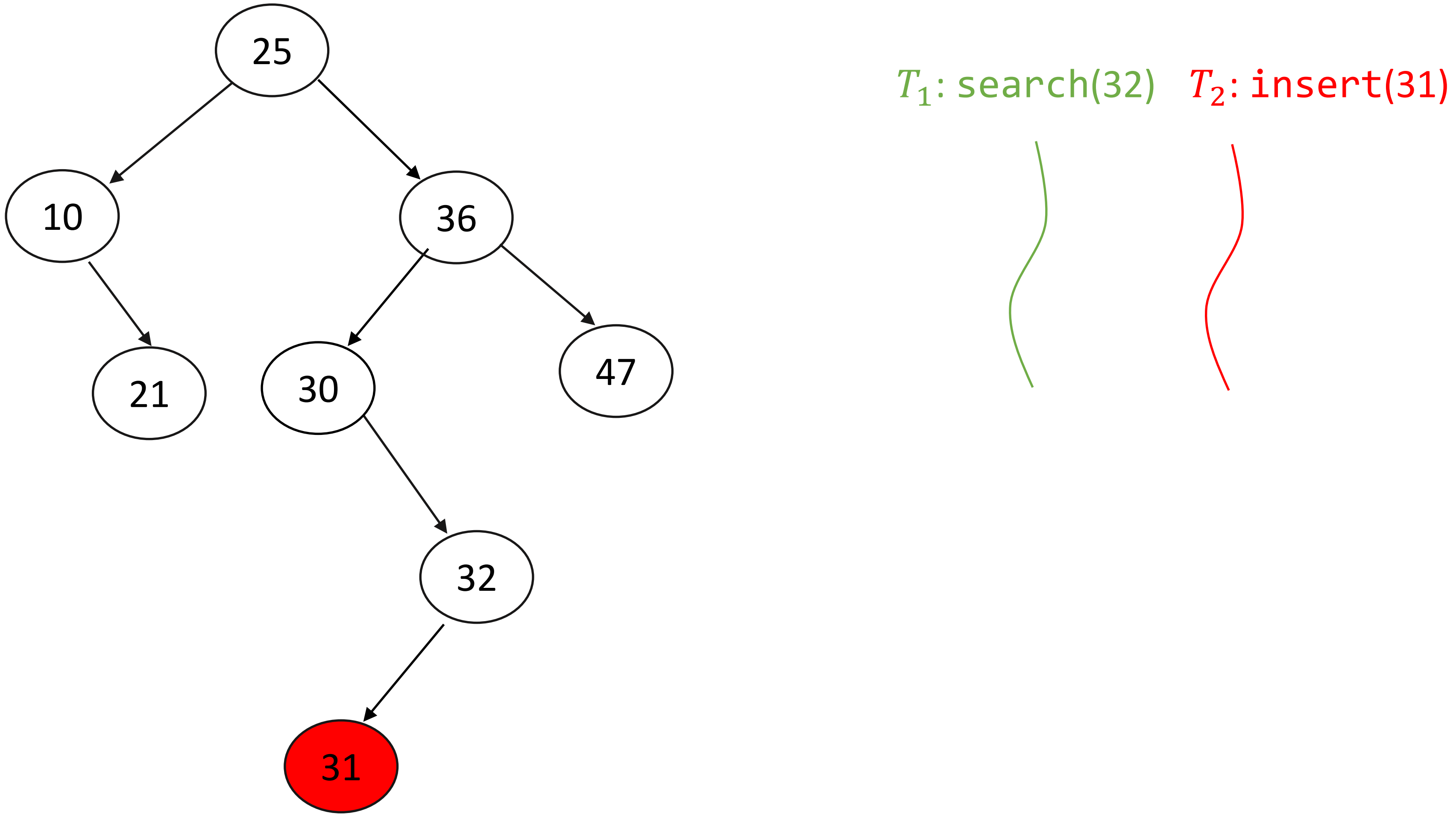


T_1: search(32) T_2: insert(31)

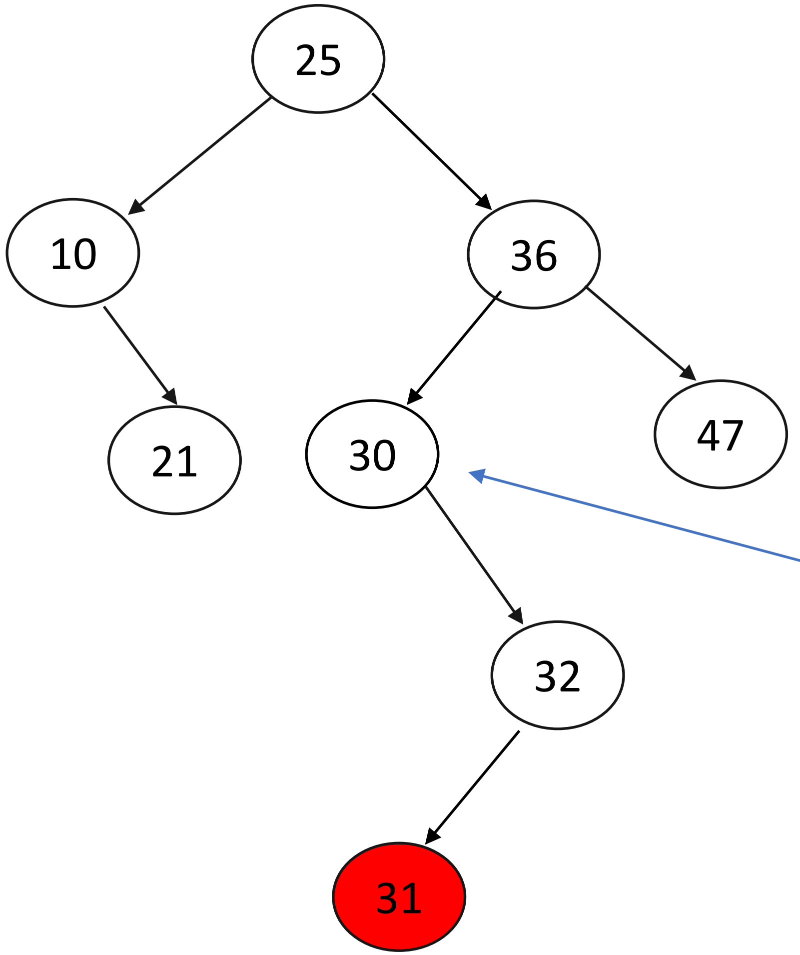
AVL Tree Example: Search and Insert



AVL Tree Example: Search and Insert

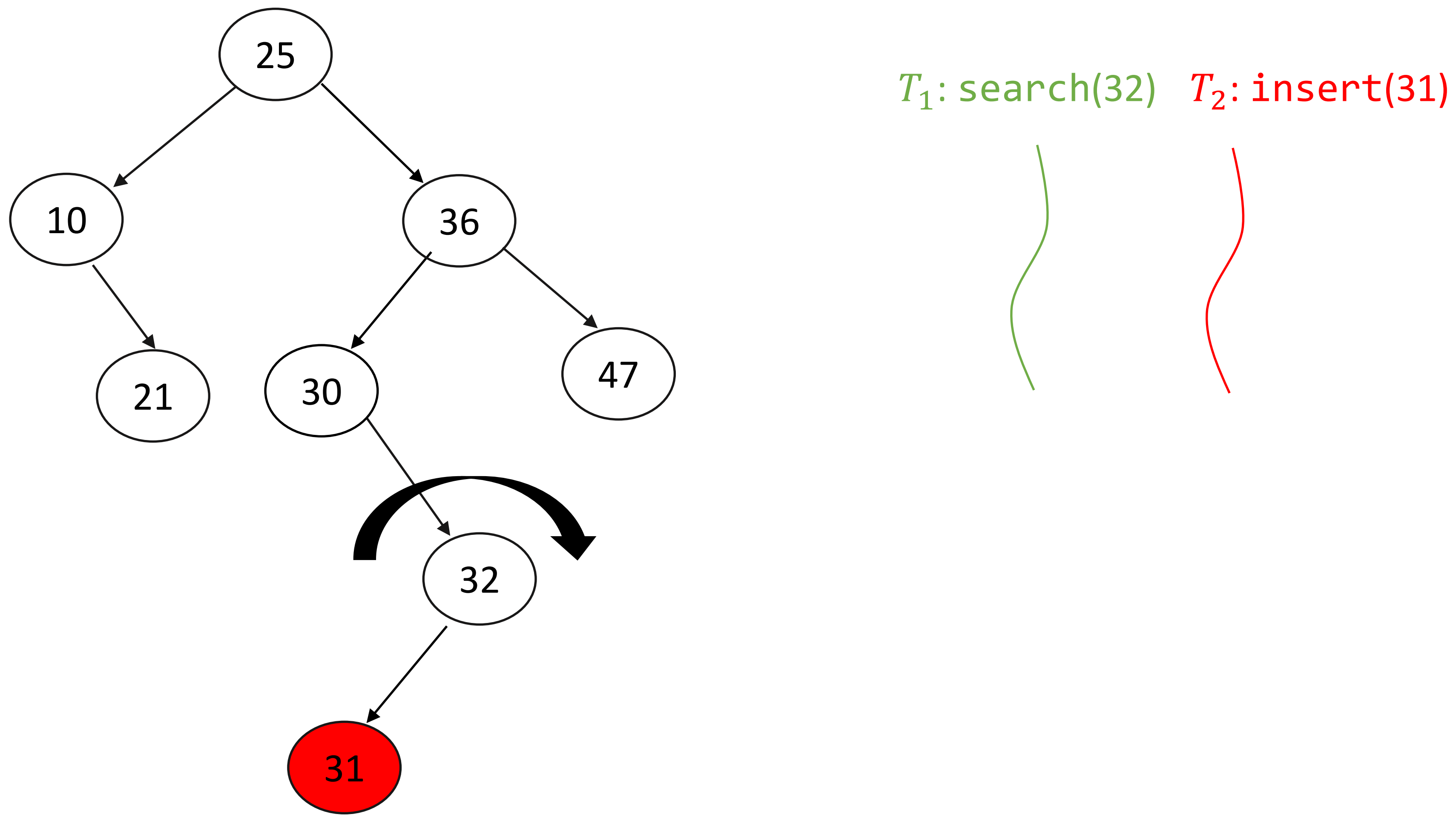


AVL Tree Example: Search and Insert

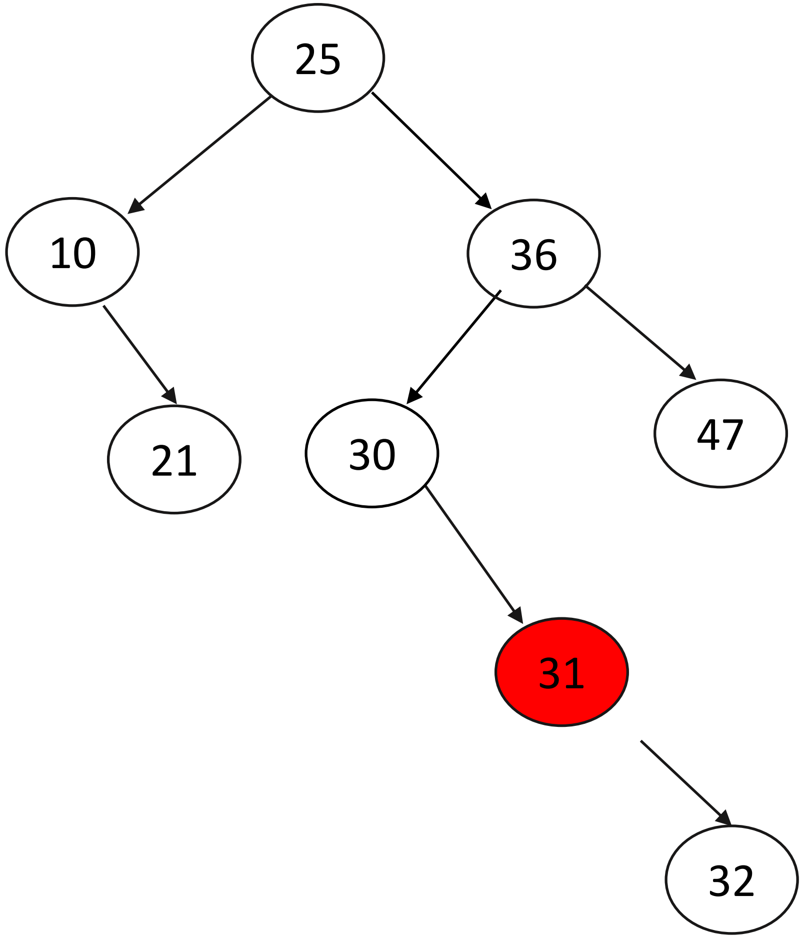


We need to re-balance the subtrees rooted at this node to satisfy the AVL condition!

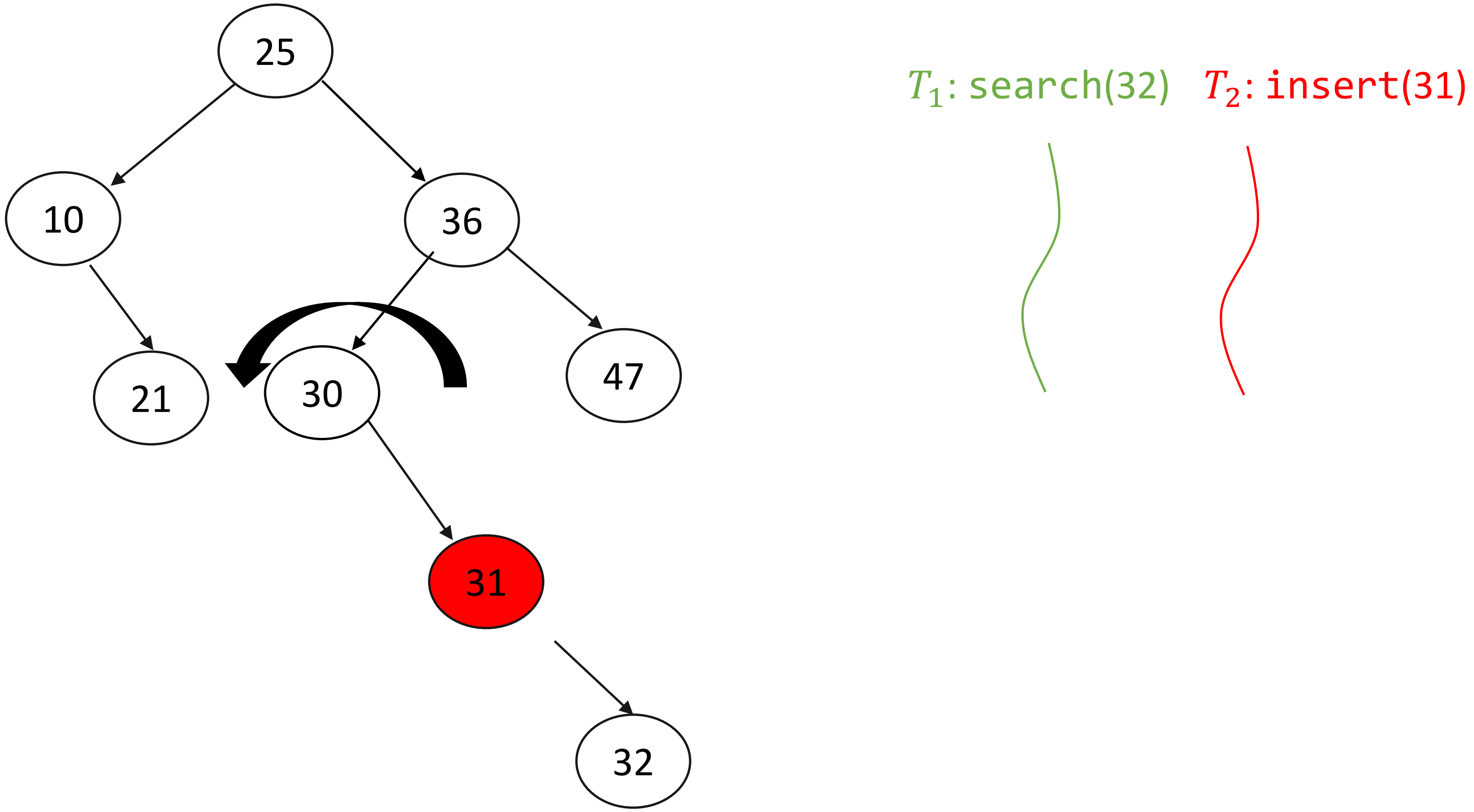
AVL Tree Example: Search and Insert



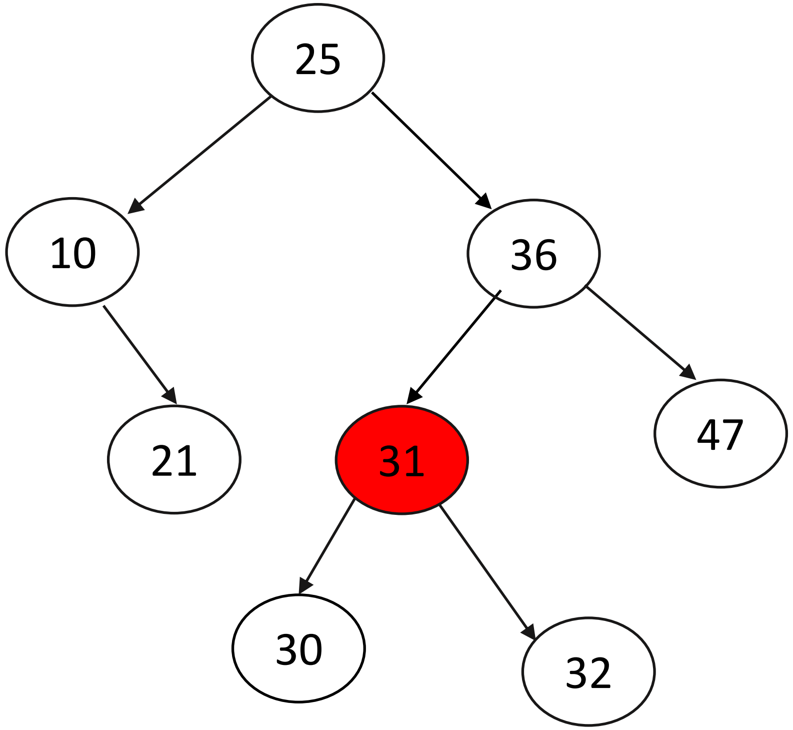
AVL Tree Example: Search and Insert



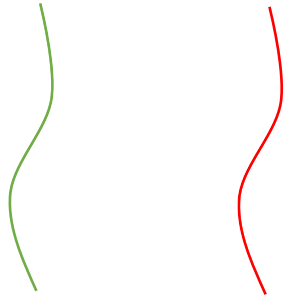
AVL Tree Example: Search and Insert



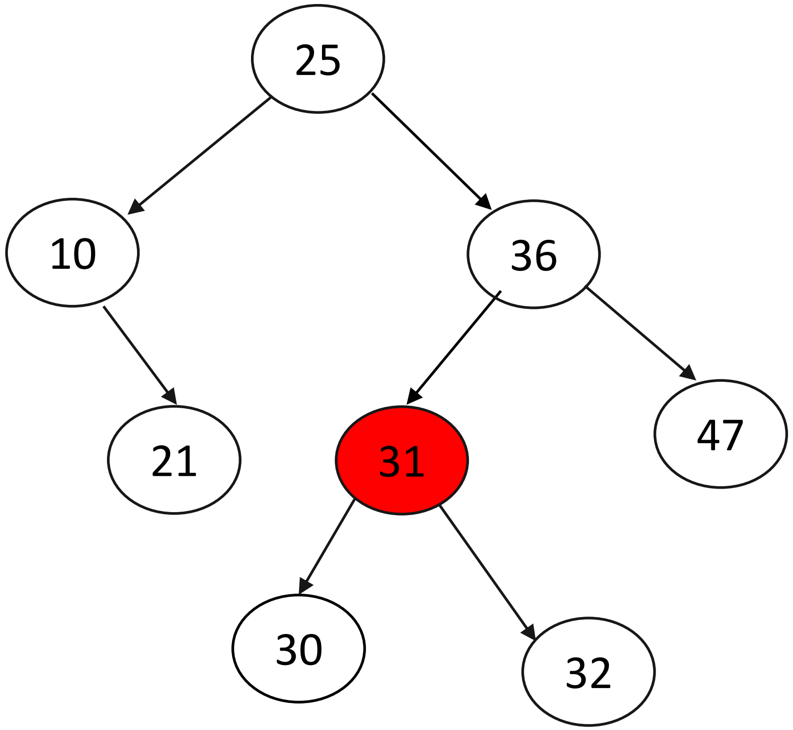
AVL Tree Example: Search and Insert



T_1: search(32) T_2: insert(31)

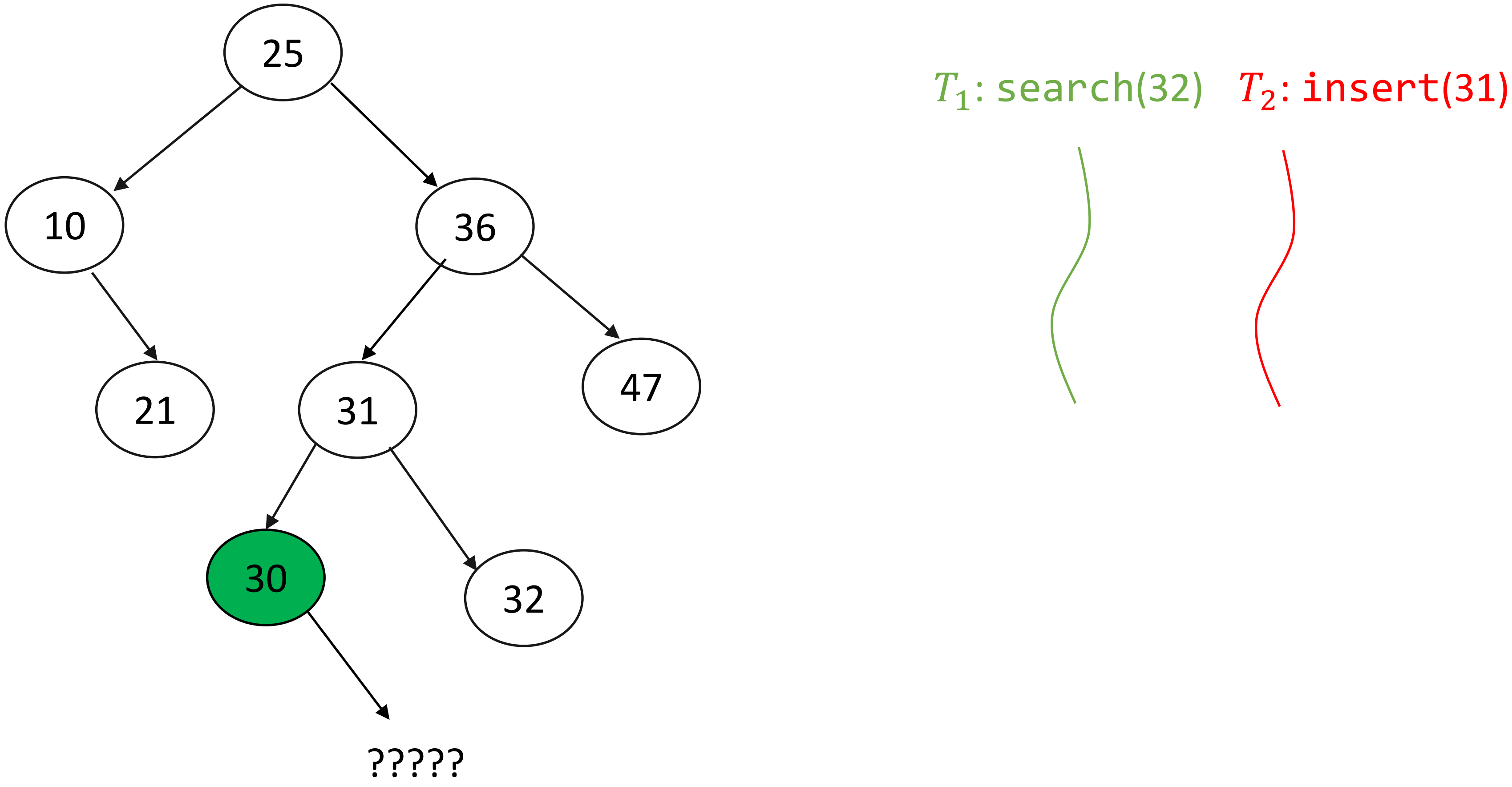


AVL Tree Example: Search and Insert

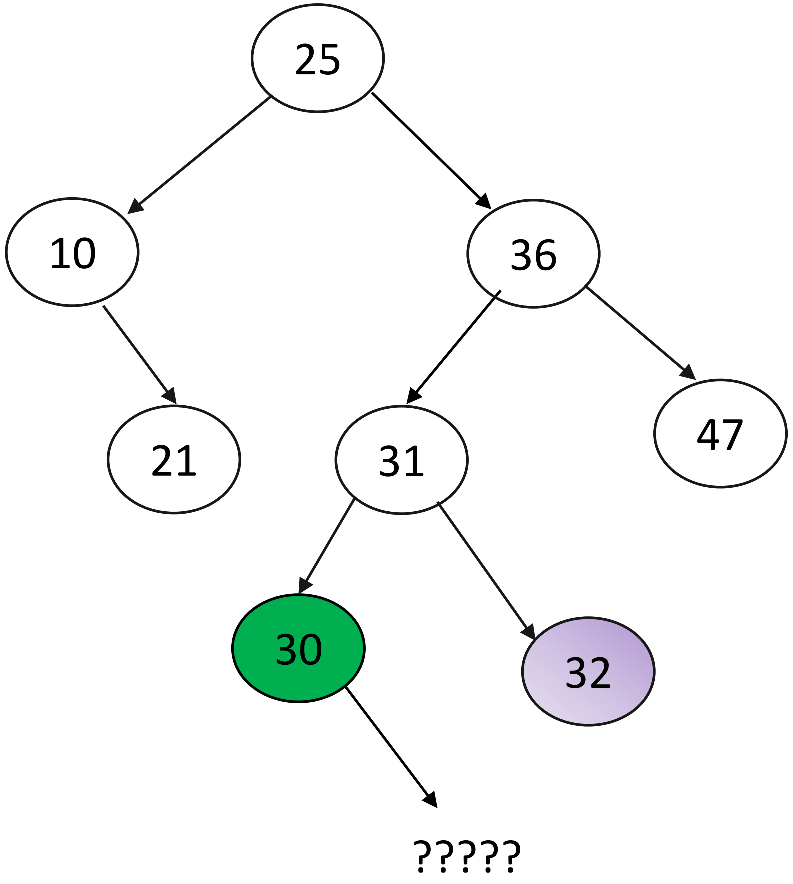


Now suppose that T_1 gets control of the CPU again. As a reminder, T_1 had stopped at node 30, and was just about to dereference the right link to succeed in its search of 32!

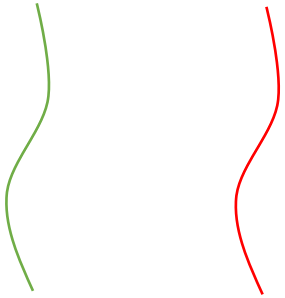
AVL Tree Example: Search and Insert



AVL Tree Example: Search and Insert

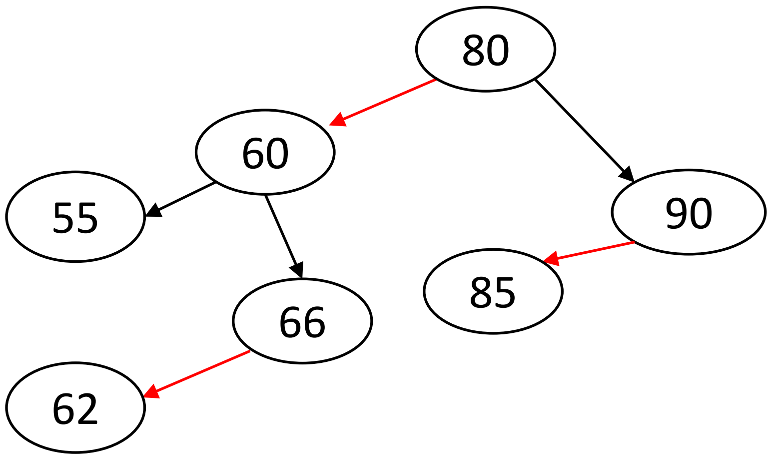


T_1: search(32) T_2: insert(31)



32 is **still** in the tree, but T_1 can't find it!!!

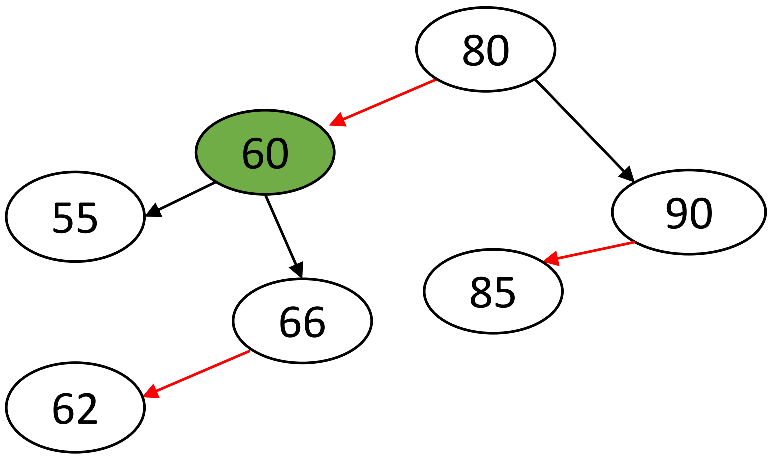
Red-Black Tree example: Search and Insert



T_1: `search(66)` T_2: `insert(70)`

Again, suppose T_1 takes control of the CPU first.

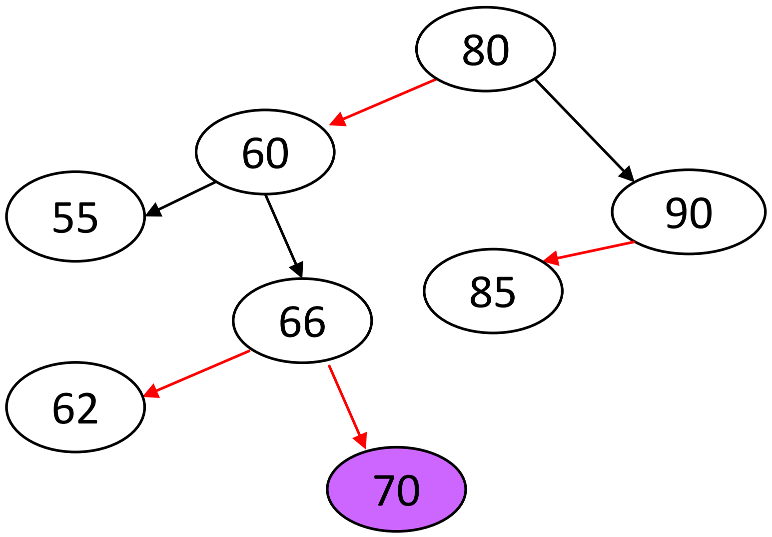
Red-Black Tree example: Search and Insert



T_1: `search(66)` T_2: `insert(70)`

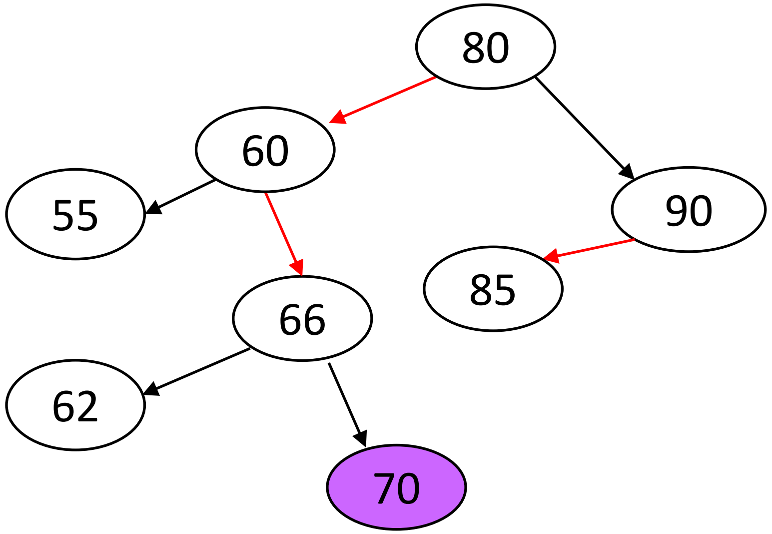
Suppose T_1 stops its search at node 60, and then yields the CPU to T_2.

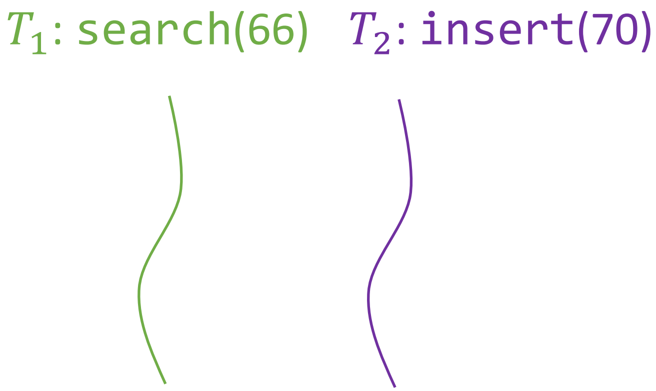
Red-Black Tree example: Search and Insert



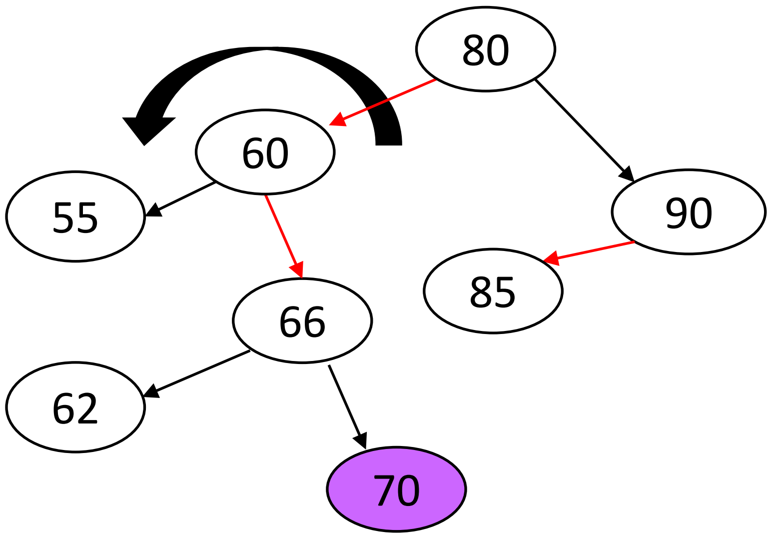
T_1: `search(66)` T_2: `insert(70)`

Red-Black Tree example: Search and Insert



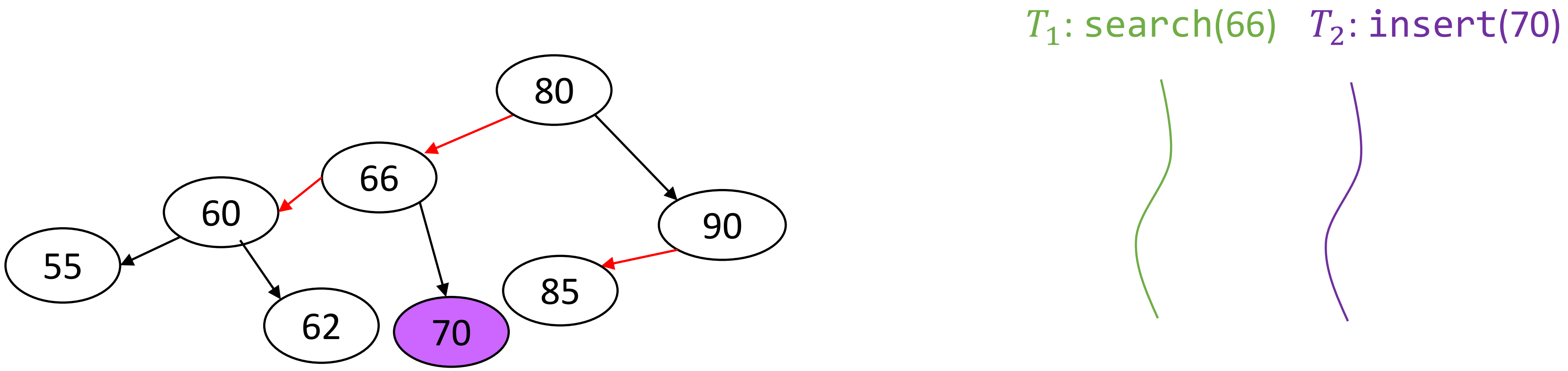


Red-Black Tree example: Search and Insert

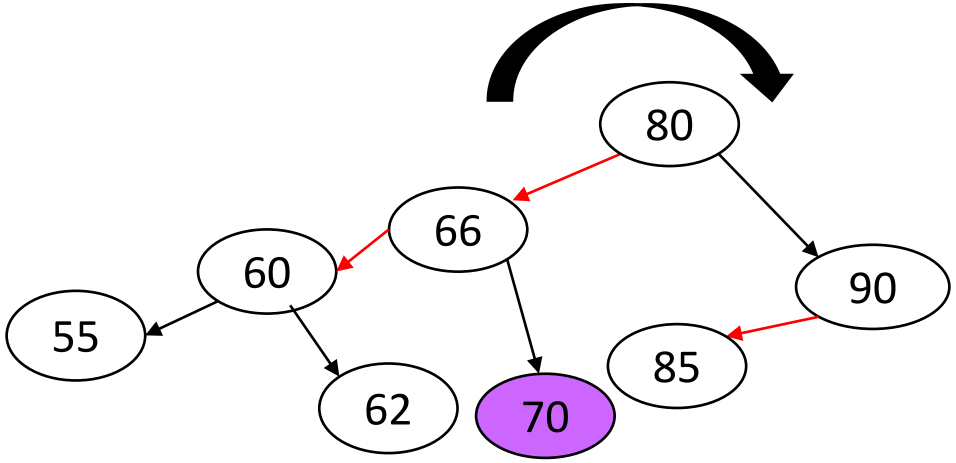


T_1: `search(66)` T_2: `insert(70)`

Red-Black Tree example: Search and Insert

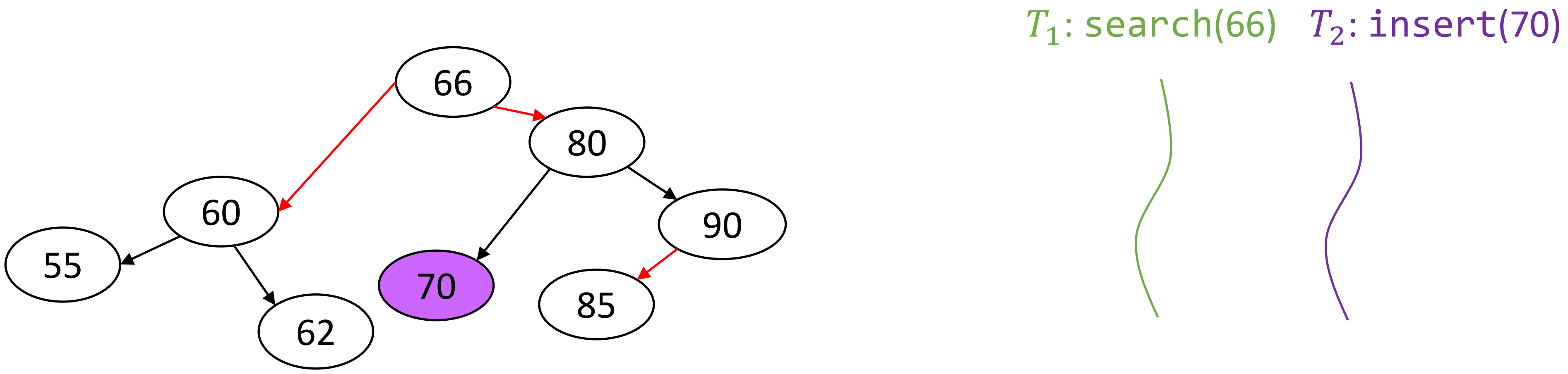


Red-Black Tree example: Search and Insert

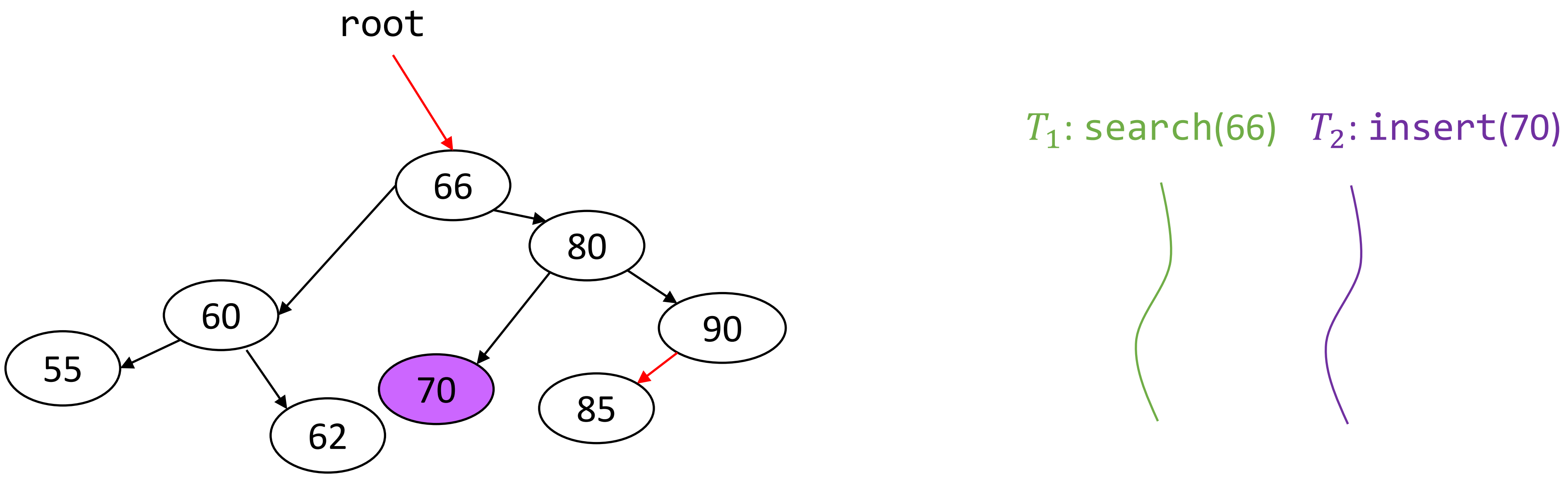


T_1: `search(66)` T_2: `insert(70)`

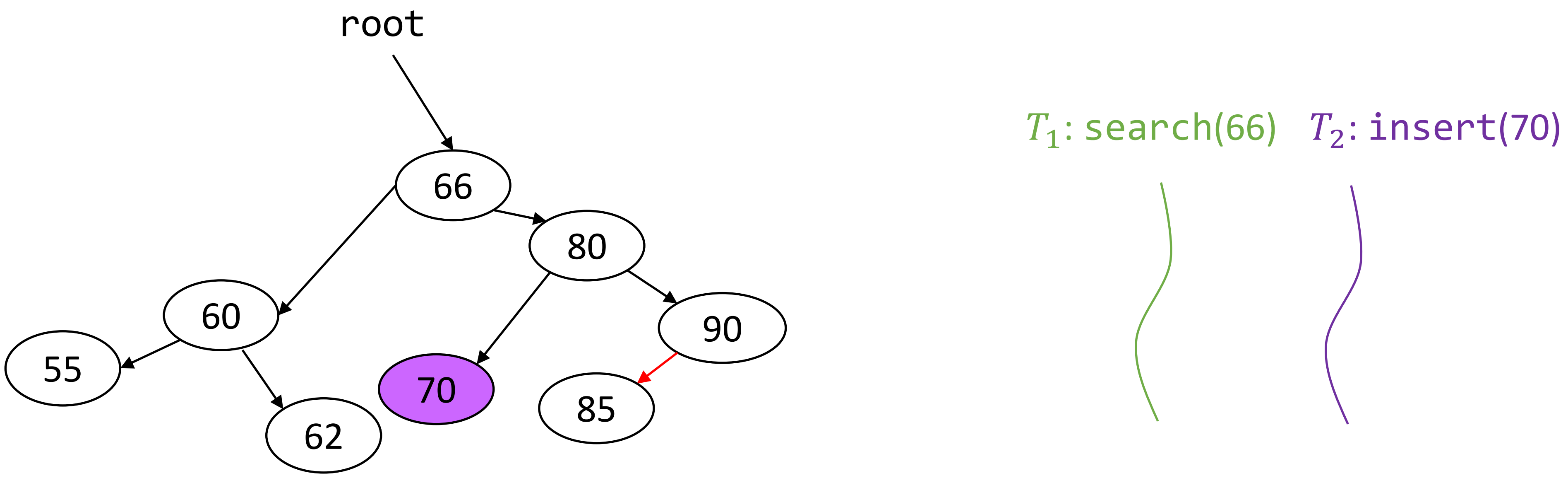
Red-Black Tree example: Search and Insert



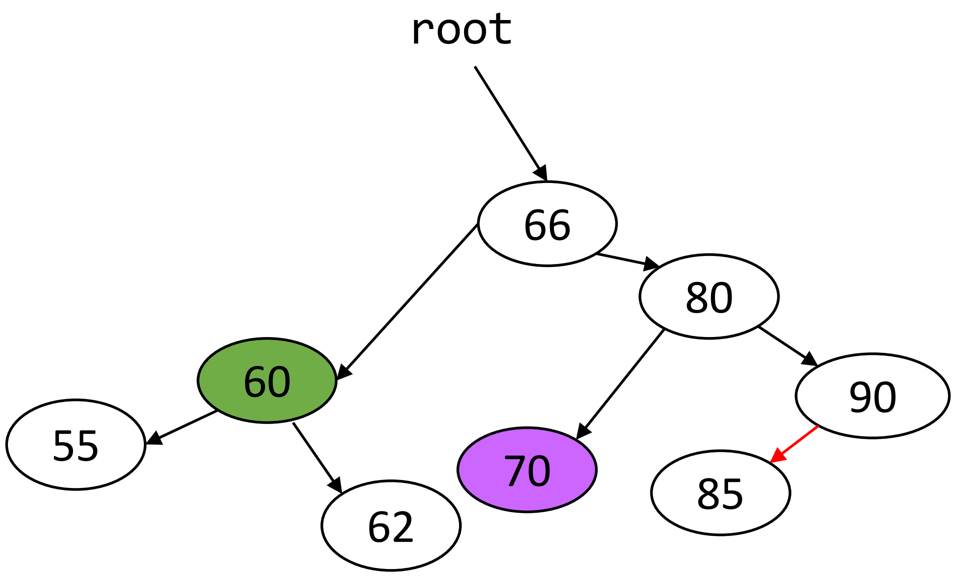
Red-Black Tree example: Search and Insert



Red-Black Tree example: Search and Insert



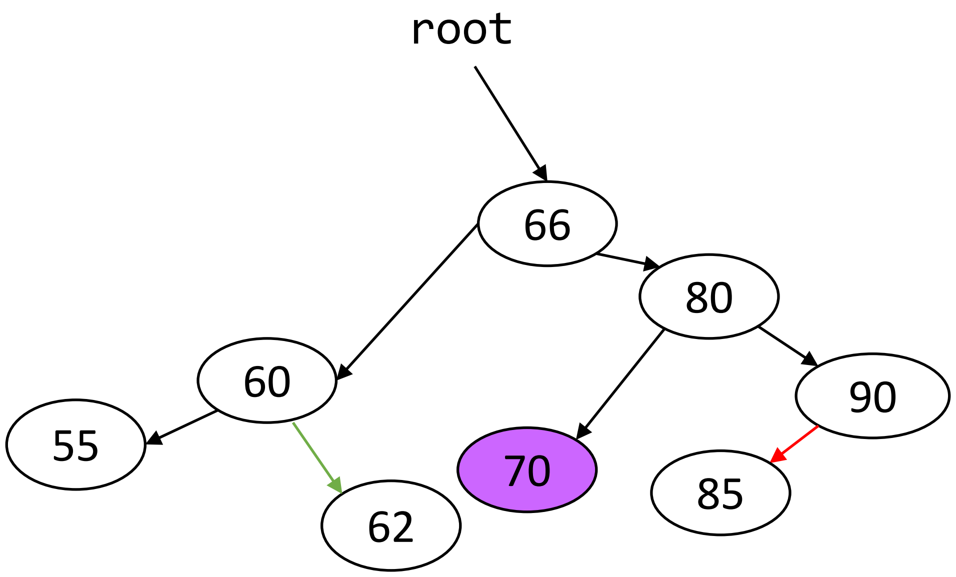
Red-Black Tree example: Search and Insert

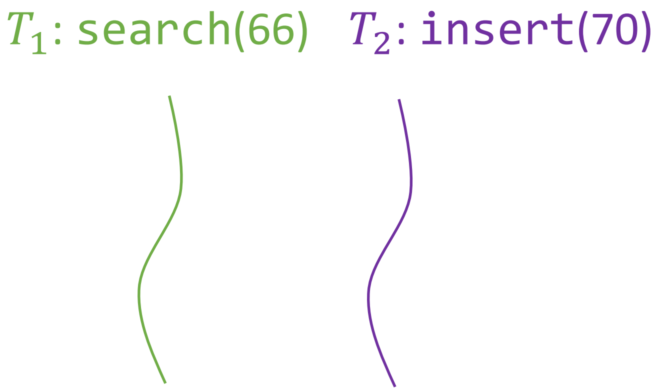


T_1: `search(66)` T_2: `insert(70)`

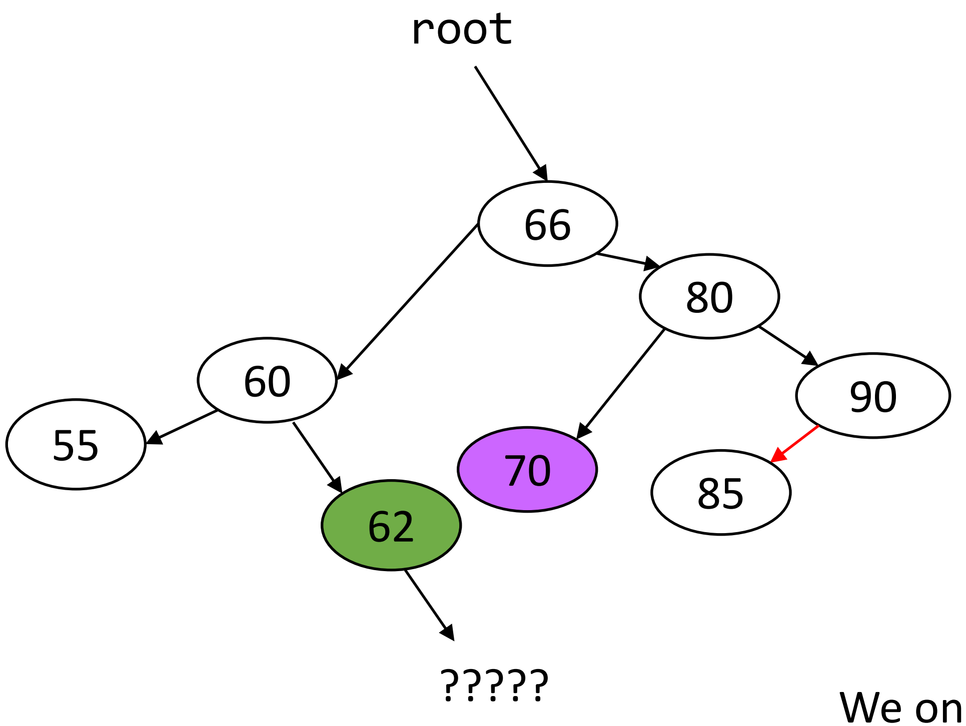
Now T_1 takes the CPU again. Remember; it was stopped when visiting node **60**.

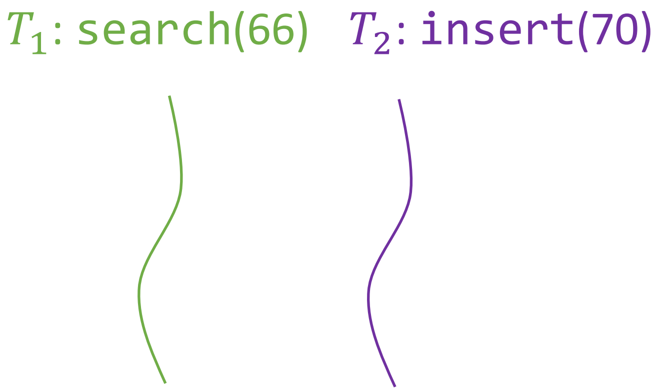
Red-Black Tree example: Search and Insert





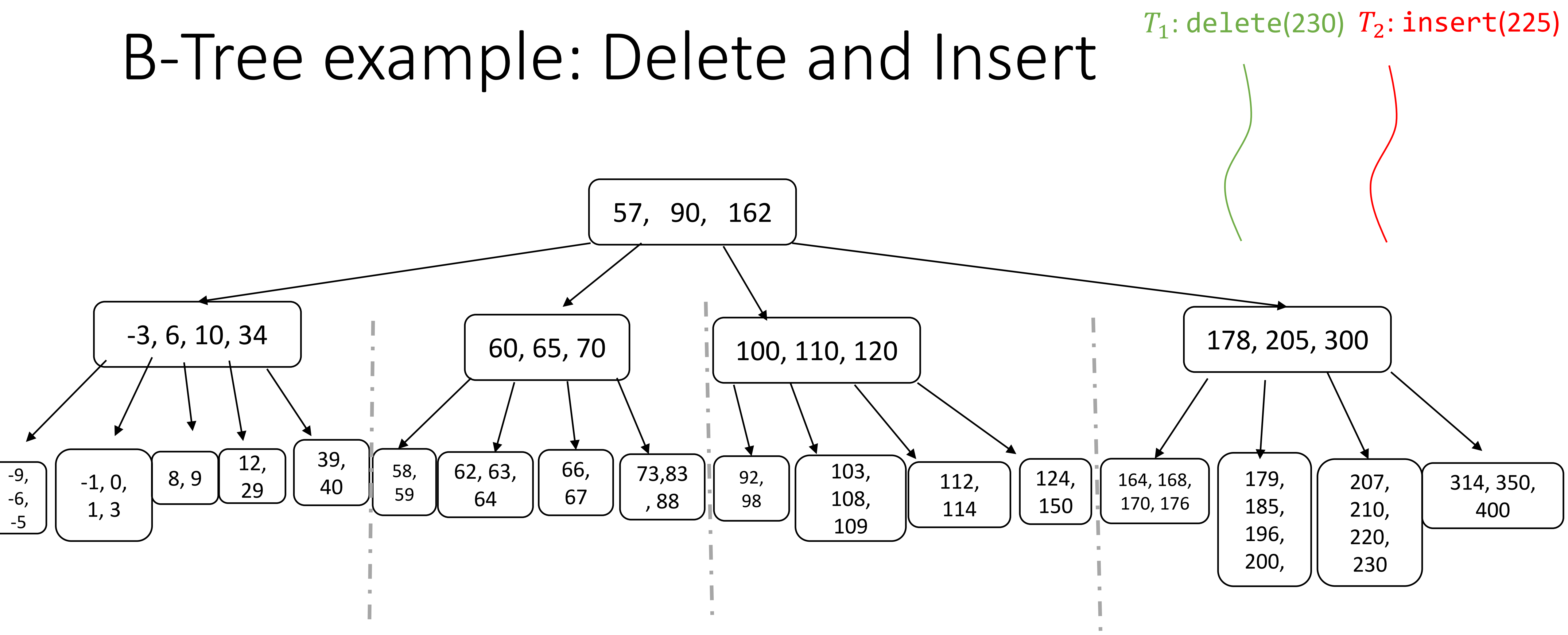
Red-Black Tree example: Search and Insert



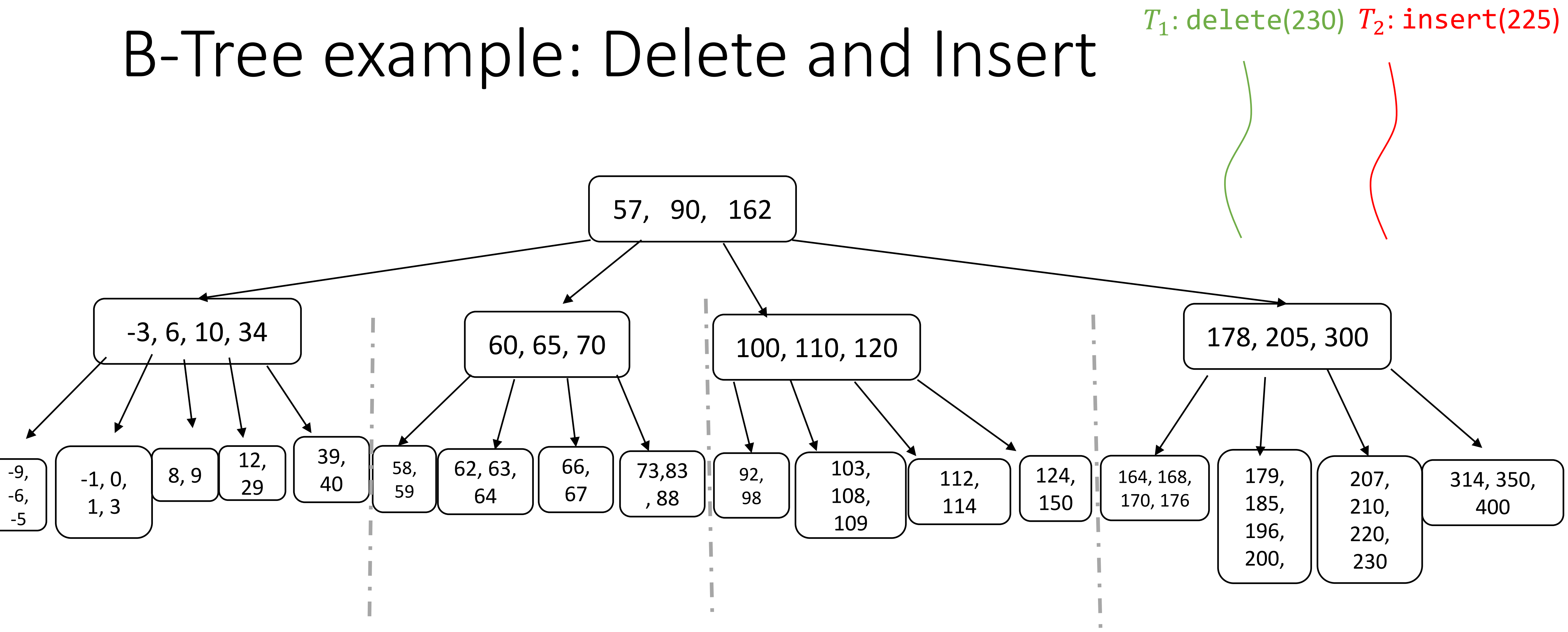


We once again produce a **false negative:** we report that 66 is not in the tree, when, in fact, it is its very **root!**

B-Tree example: Delete and Insert



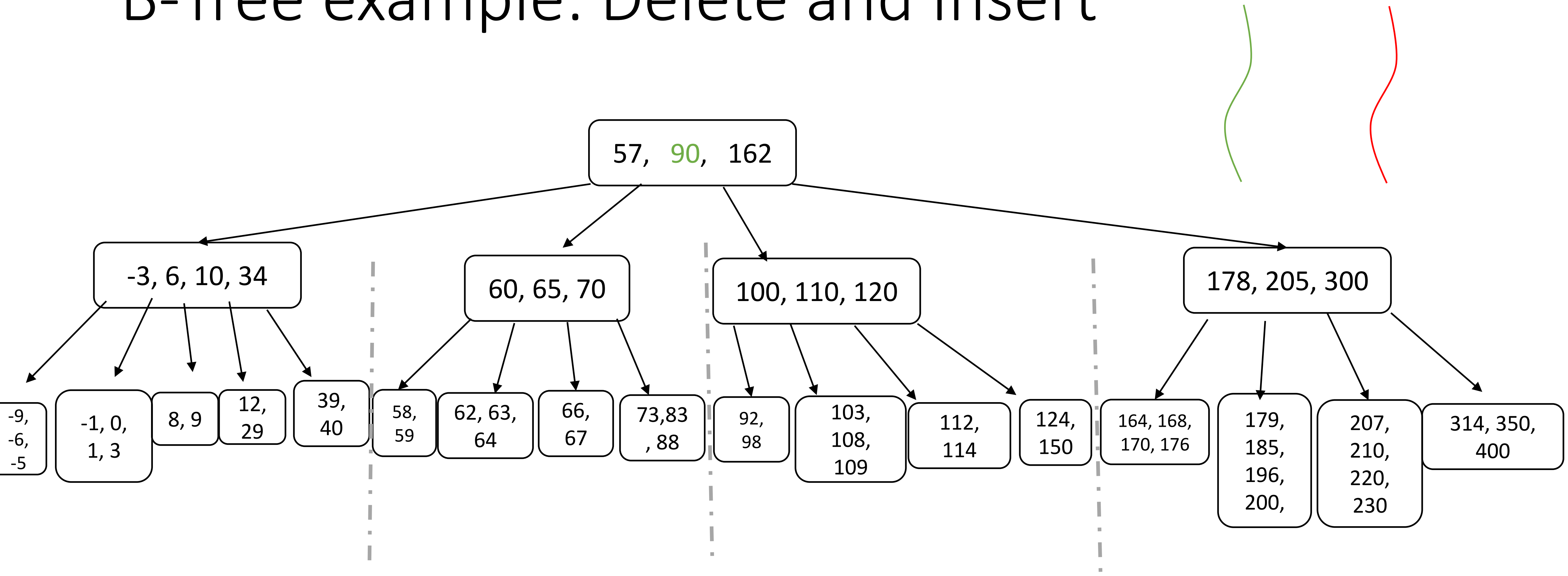
B-Tree example: Delete and Insert



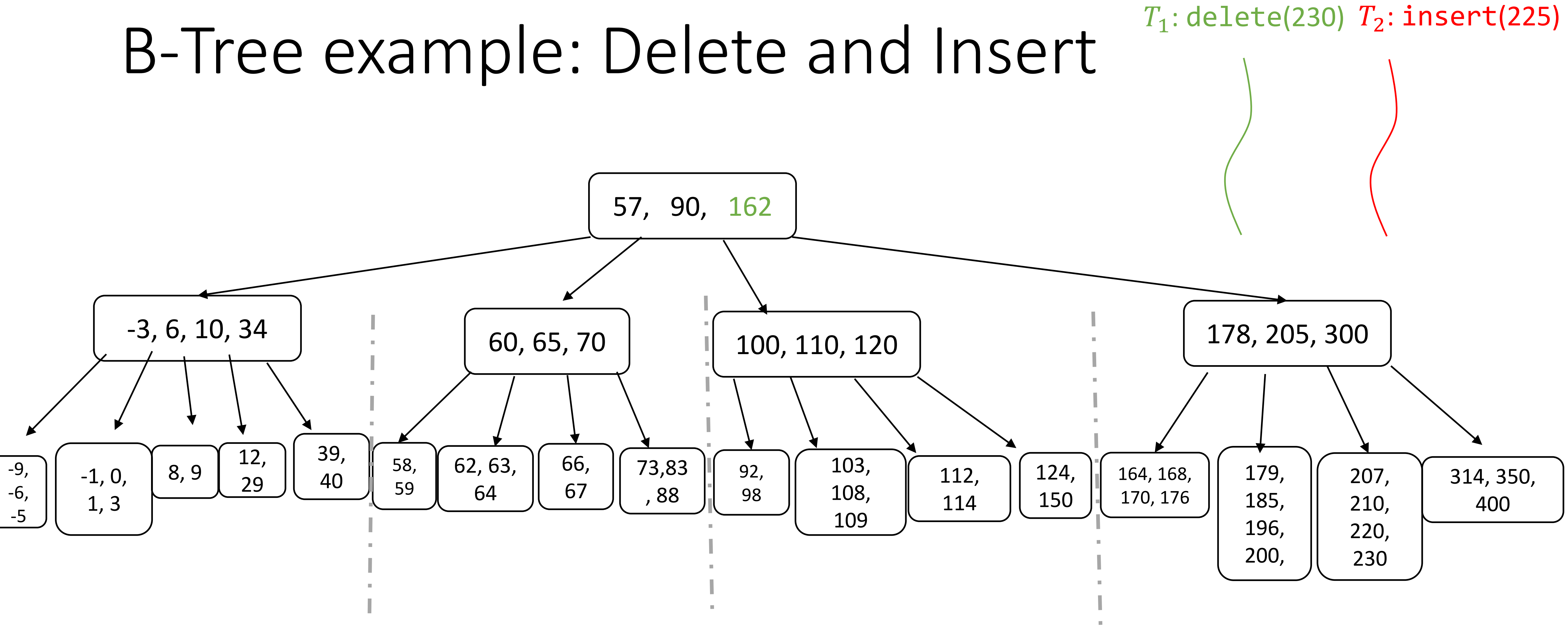
T_1 takes CPU first and starts searching for 230 to delete it.

B-Tree example: Delete and Insert

T_1: delete(230) T_2: insert(225)

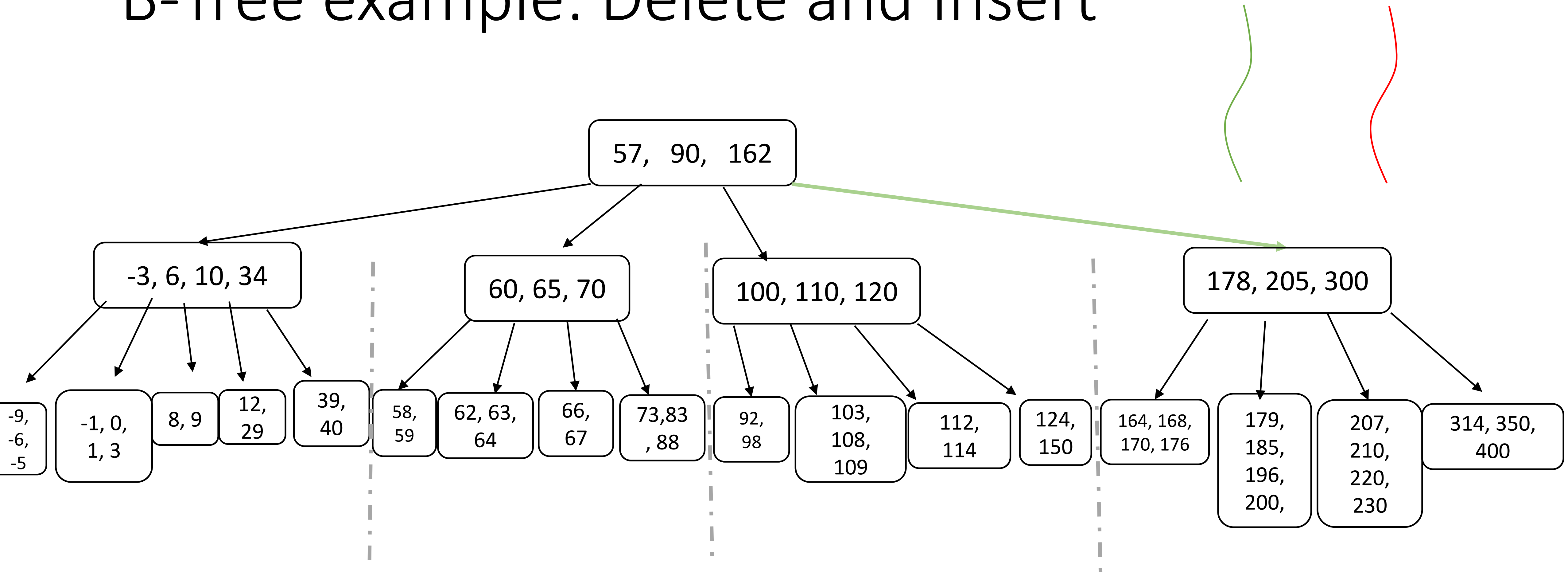


B-Tree example: Delete and Insert

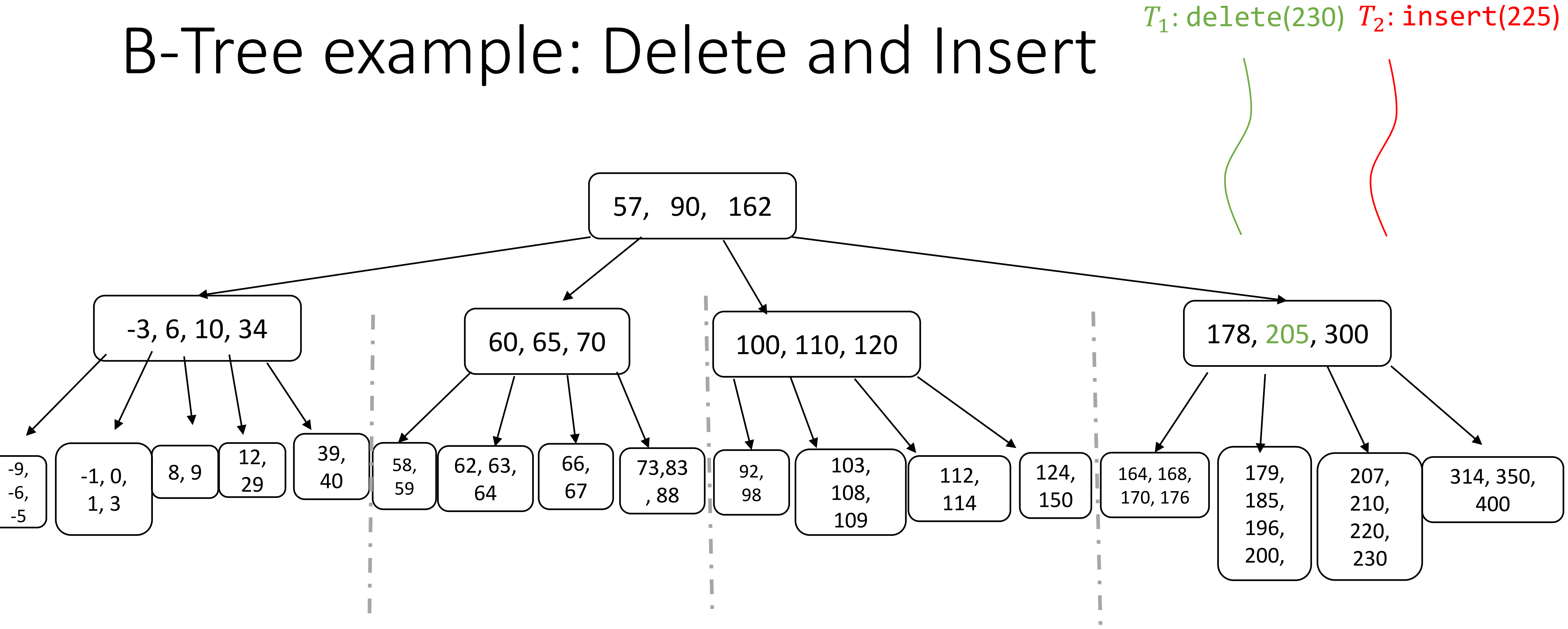


B-Tree example: Delete and Insert

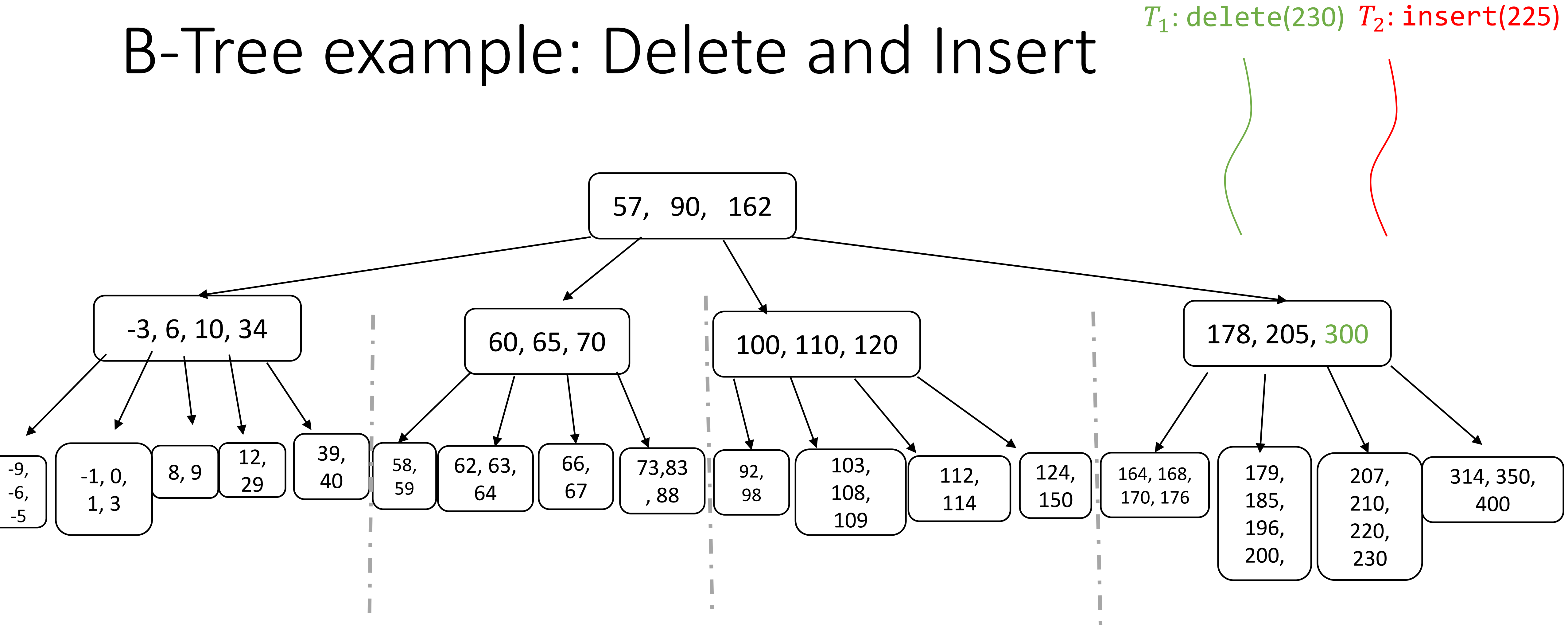
T_1: delete(230) T_2: insert(225)



B-Tree example: Delete and Insert

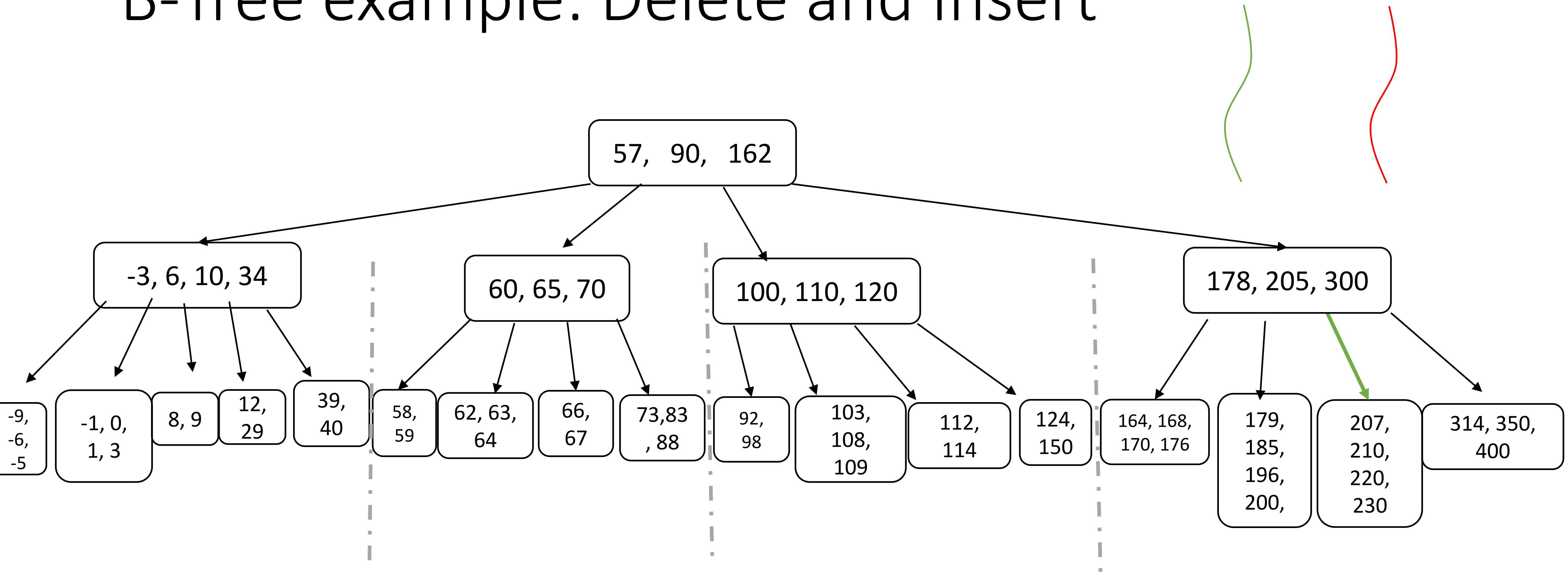


B-Tree example: Delete and Insert

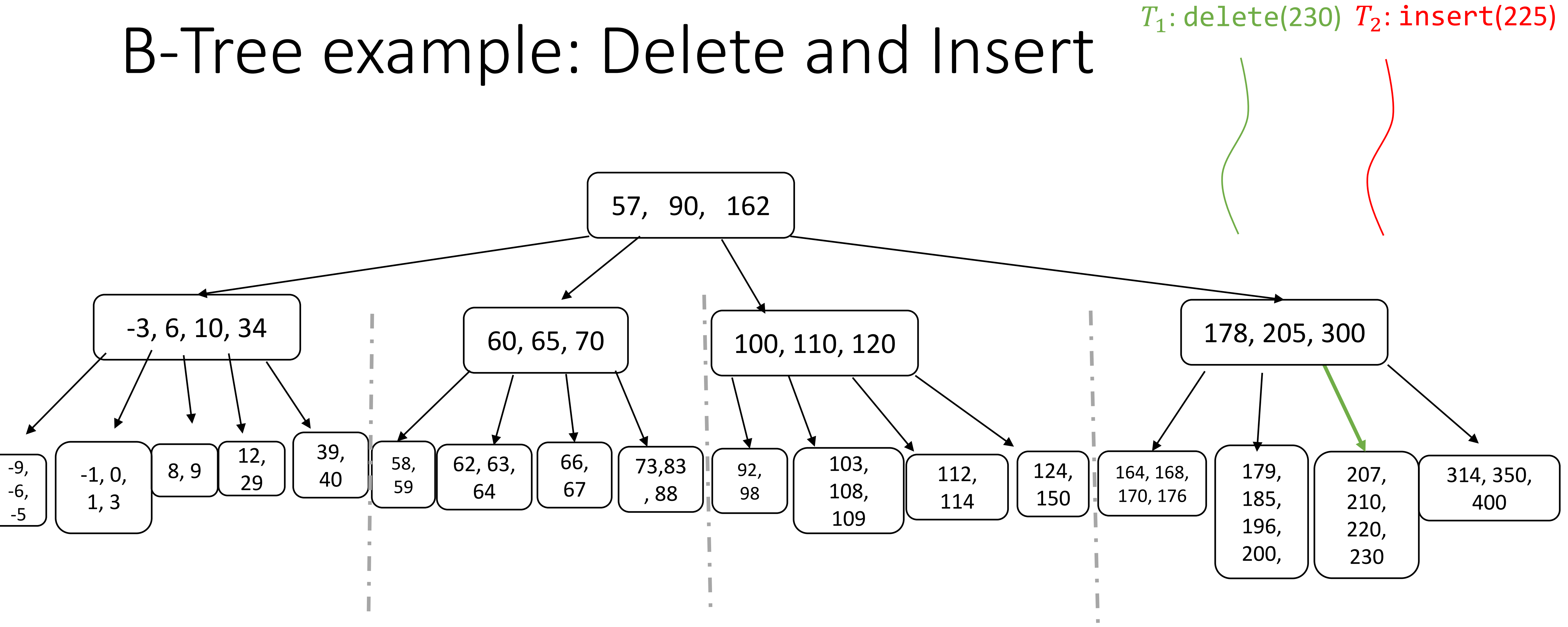


B-Tree example: Delete and Insert

T_1: delete(230) T_2: insert(225)

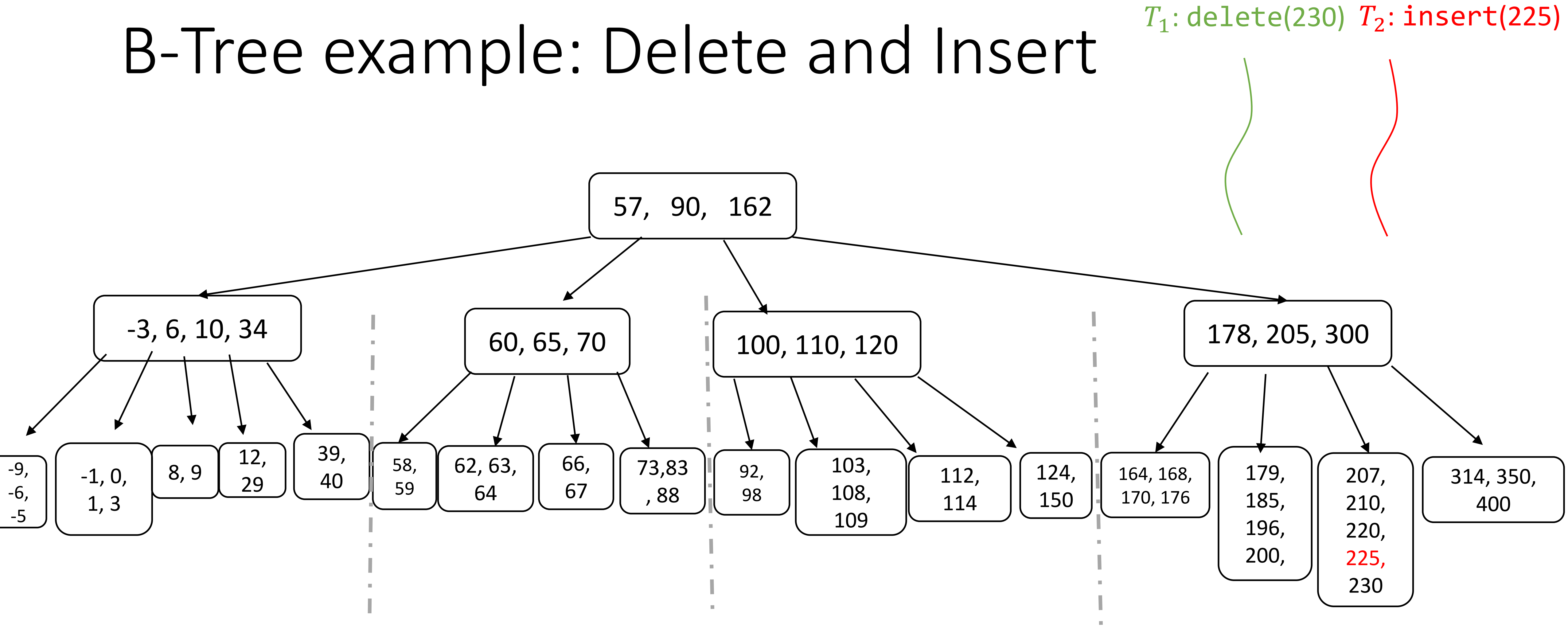


B-Tree example: Delete and Insert

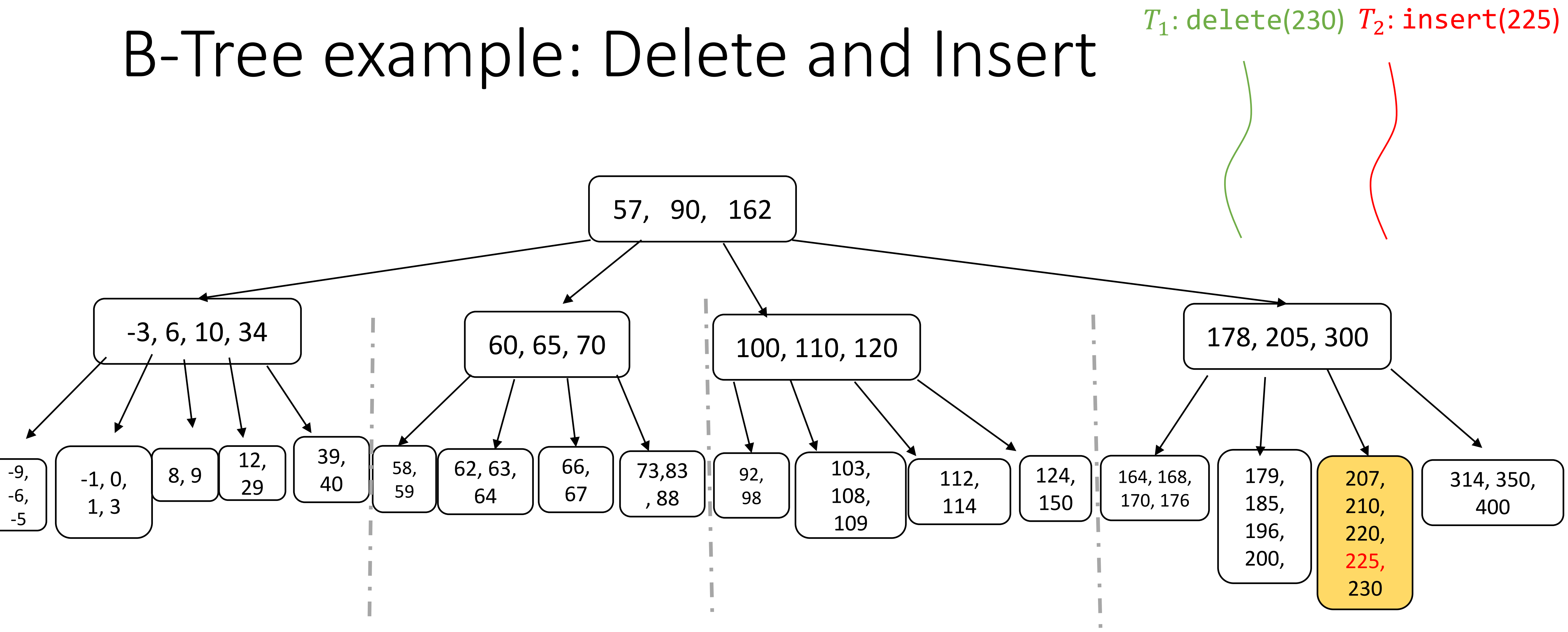


T_1 now yields the CPU to T_2, which will search for the appropriate leaf node to insert 290 into.

B-Tree example: Delete and Insert

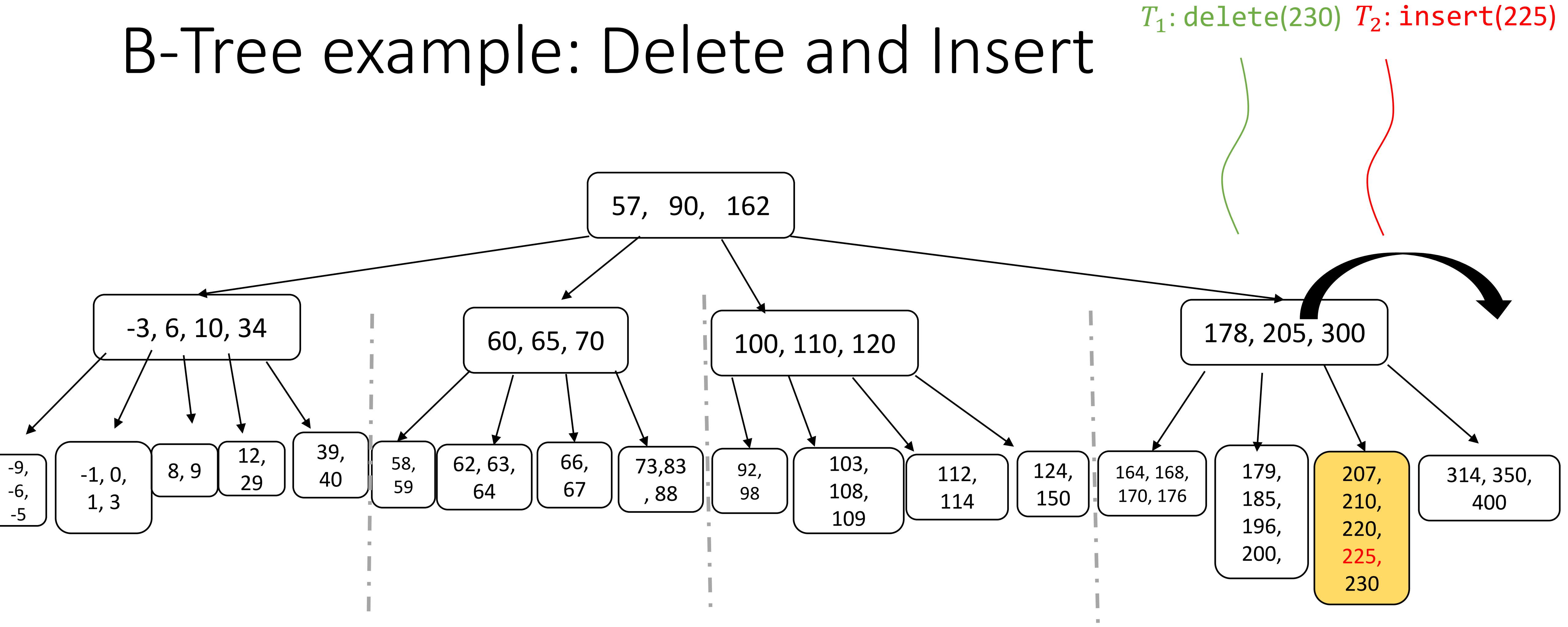


B-Tree example: Delete and Insert



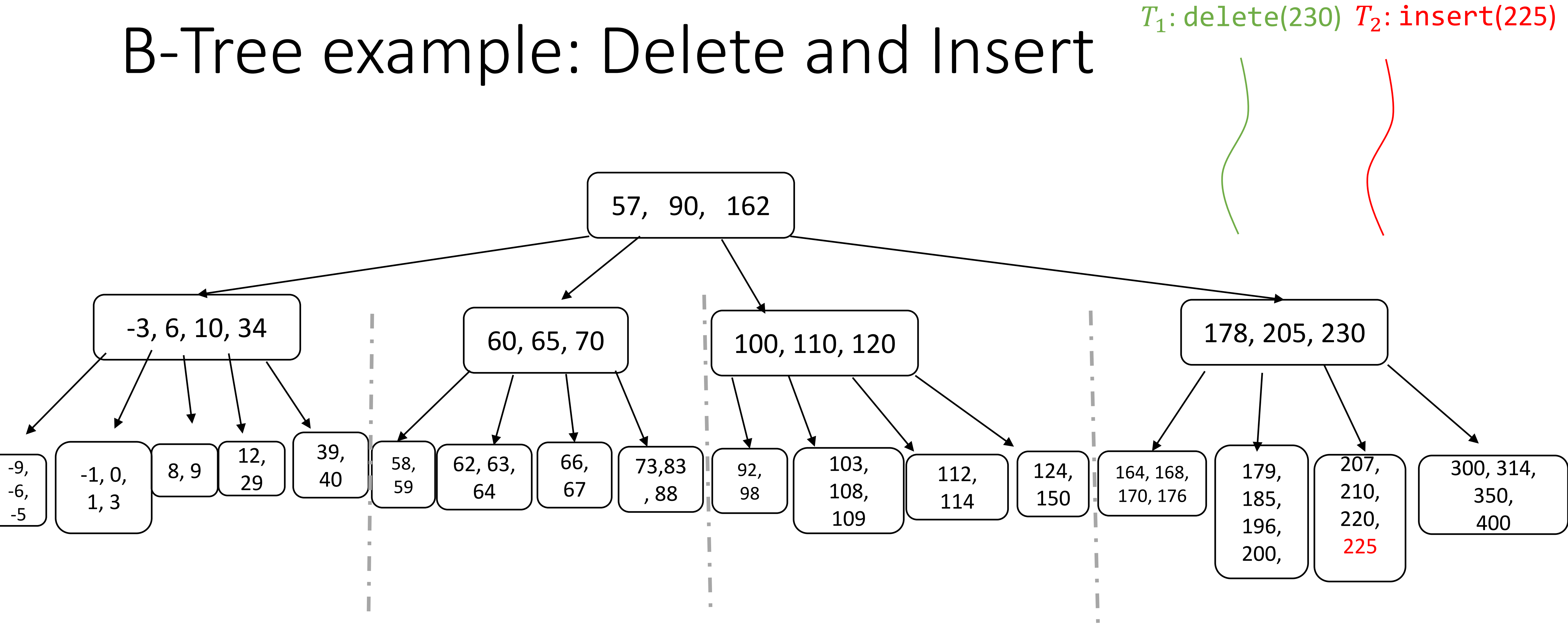
Leaf node overflows… ☹

B-Tree example: Delete and Insert

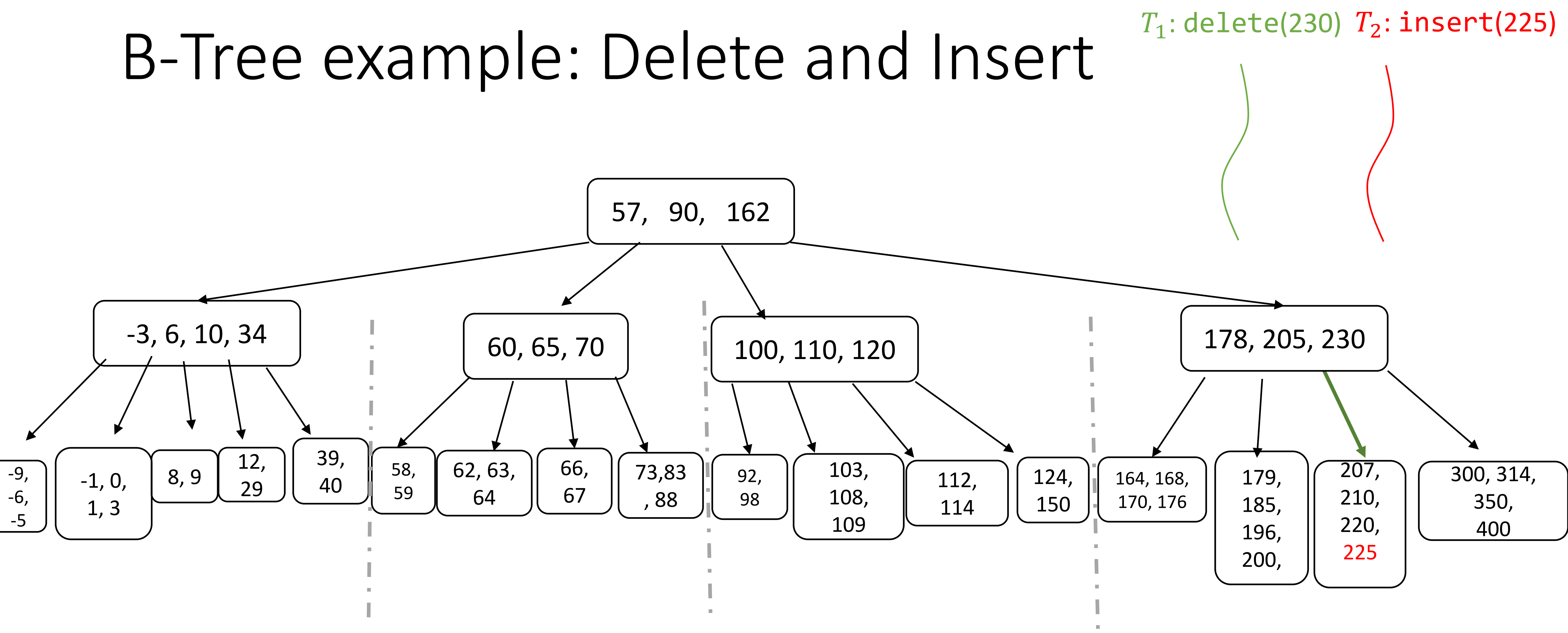


So let's solve this overflow with a right key rotation…

B-Tree example: Delete and Insert

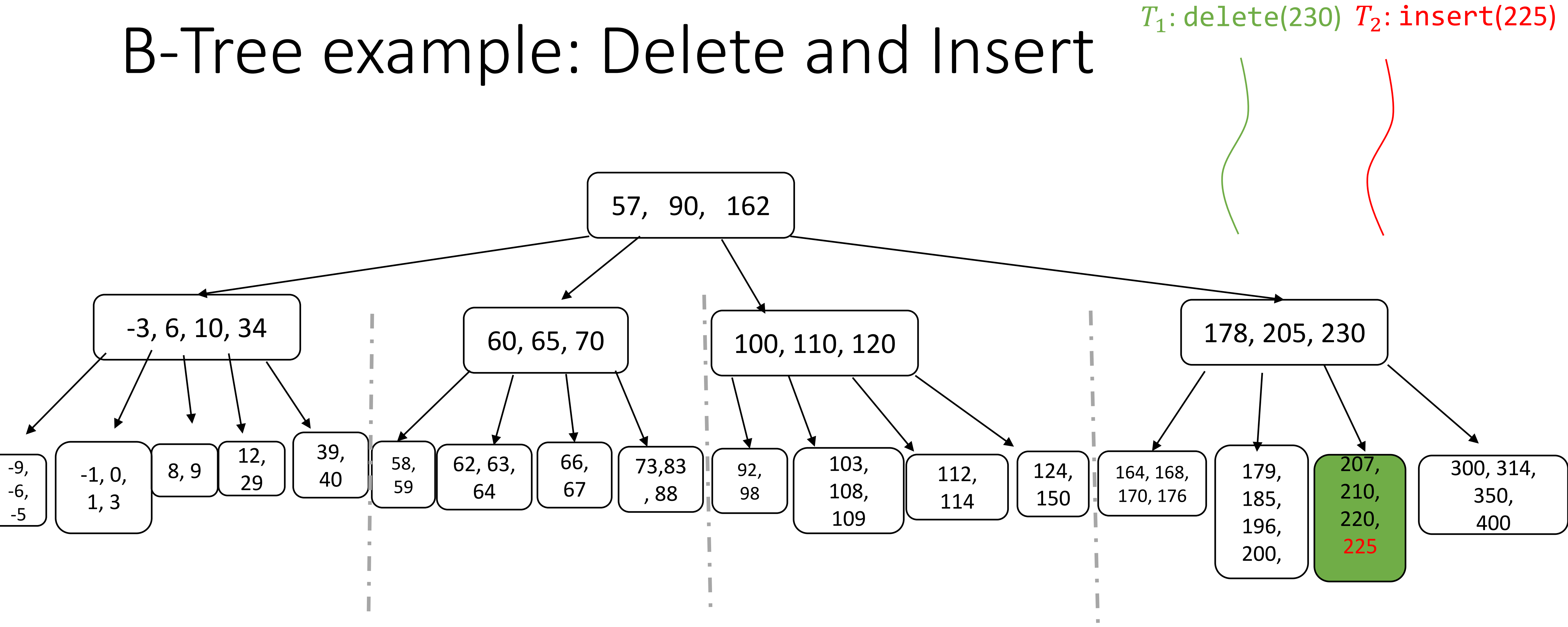


B-Tree example: Delete and Insert



Now we yield the CPU back to T_1, which we last left looking at the leaf node where 230 *used* to be…

B-Tree example: Delete and Insert



- 230 is nowhere to be found, because it's been rotated to the parent!
- Another false negative: T_1 cannot delete 230!

Take-home message

- Because of their node or key rotations, balanced trees pose serious risks when operated upon by multiple threads.
- The only safe thing to do is lock the entire tree and only allow one thread to access it at a time.
    - So, **Blocking** implementations.
    - Exception: You can allow multiple *searcher* threads simultaneously in the tree.
- People have tried coming up with solutions for more finer-grained locking of those trees, and they have largely failed.
- But it turns out we can do something better and easier, but it’s not through a tree-like structure!