B-Trees

CMSC 420

Improving our index

- B-Tree: A generalization of a 2-3 tree.
  - Parameter p (for $p$ointer) controls the ***fan-out (#children)*** of every node.
  - For efficiency purposes, usually nodes allocated with static array of size p that contain pointers to subtrees.
  - Any **non-root** node can have between $\left\lceil \frac{p}{2} \right\rceil - 1$ and $p - 1$ ***keys inclusive***.
  - The root itself is the only node allowed to have a **single key (2 children)**
    - Just ***allowed***. Not necessarily **"has to have".**

Next-level quiz

- **Quiz**: To separate p subtrees I need the following #separators…

Next-level quiz

- **Quiz**: To separate p subtrees I need the following #separators…

$p-1$	p	$p+1$	Something else (what?)

Next-level quiz

- **Quiz**: To separate p subtrees I need the following #separators…

$p-1$	p	$p+1$	Something else (what?)

So, **all B-Tree nodes** have $(p-1)$ ***4***-byte separators and p ***8***-byte pointers, for a total memory footprint of $12p - 4 = \mathcal{O}(p)$ ***each***!

Insertion / Deletion into B-Trees

- Generalization of insertion /deletion in 2-3 trees.
  - Difference: arrays of keys and references inside every node
- Some new constraints, though:
  1. Any **non-root** node can have between $\left\lceil\frac{p}{2}\right\rceil - 1$ and $p-1$ keys.
     - Exception: root can have 2 children! (we'll see why)
  2. If a key is inserted into a node with $p-1$ keys, we have an overflow.
  3. If a key is deleted from a leaf node with $\left\lceil\frac{p}{2}\right\rceil - 1$ keys, we have an underflow.

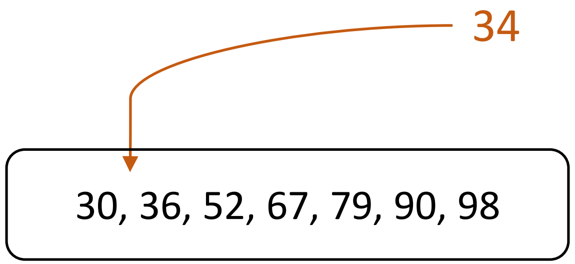
What’s with the constraint on the root?

- Suppose we have $p = 8$ and the following “stub” B-Tree consisting of just the root…

30, 36, 52, 67, 79, 90, 98

What’s with the constraint on the root?

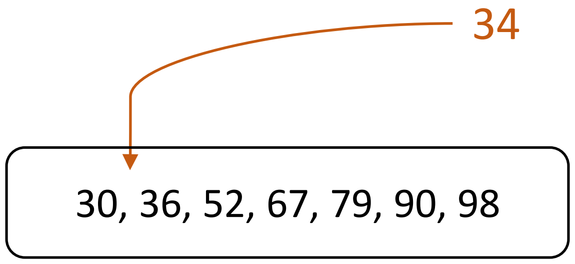
- Suppose we have $p = 8$ and the following “stub” B-Tree consisting of just the root…



- And now I want to insert they key 34 in my tree.

What’s with the constraint on the root?

- Suppose we have $p = 8$ and the following “stub” B-Tree consisting of just the root…



- And now I want to insert they key 34 in my tree.

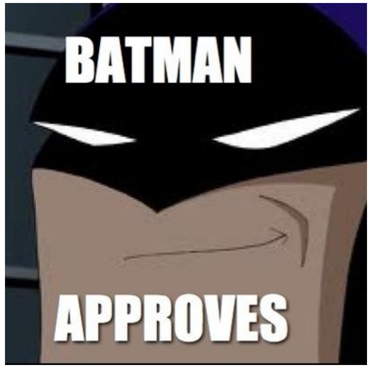
What's with the constraints on root and leaves?

- Suppose we have $p = 8$ and the following "stub" B-Tree consisting of just the root…

34

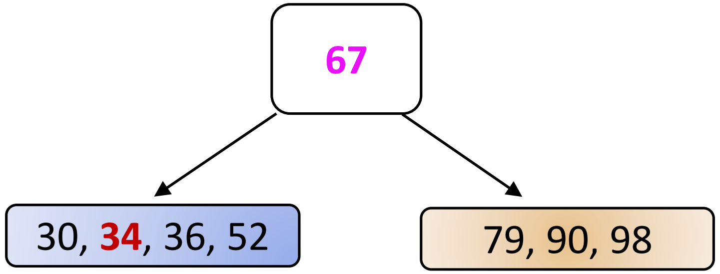
30, 36, 52, **67**, 79, 90, 98

- And now I want to insert they key 34 in my tree.
- Solution: Split the root!



What’s with the constraints on root and leaves?

- Suppose we have $p = 8$ and the following “stub” B-Tree consisting of just the root…



- And now I want to insert they key 34 in my tree.
- Solution: Split the root!
  - Left child: holds $4 = \left\lceil \frac{p}{2} \right\rceil \geq \left\lceil \frac{p}{2} \right\rceil - 1$ keys which is our lower bound
  - Right child: holds $3 = \left\lceil \frac{p}{2} \right\rceil \geq \left\lceil \frac{p}{2} \right\rceil - 1$, our lower bound

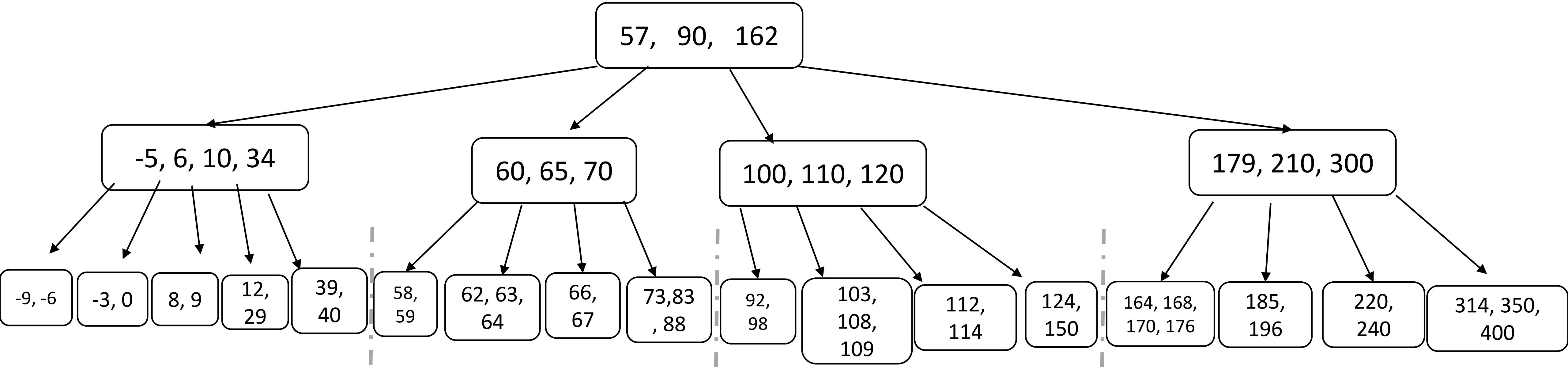
Dealing with overflows

- Unlike in 2-3 trees, where splitting is practically the only option allowed, in B-trees, we will always attempt key rotations first.
  - To avoid asking the heap for **large contiguous storage!**

Dealing with overflows

- Unlike in 2-3 trees, where splitting is practically the only option allowed, in B-trees, we will always attempt key rotations first.
  - To avoid asking the heap for **large contiguous storage!**
- Idea: Look at your sibling nodes (≤ 2) and locate at least one that would not overflow if you were to add a key to it
  - If you find one such sibling, rotate the key “closest” to the sibling node to that node through the parent.
  - This is always either the smallest or the largest key of the current node

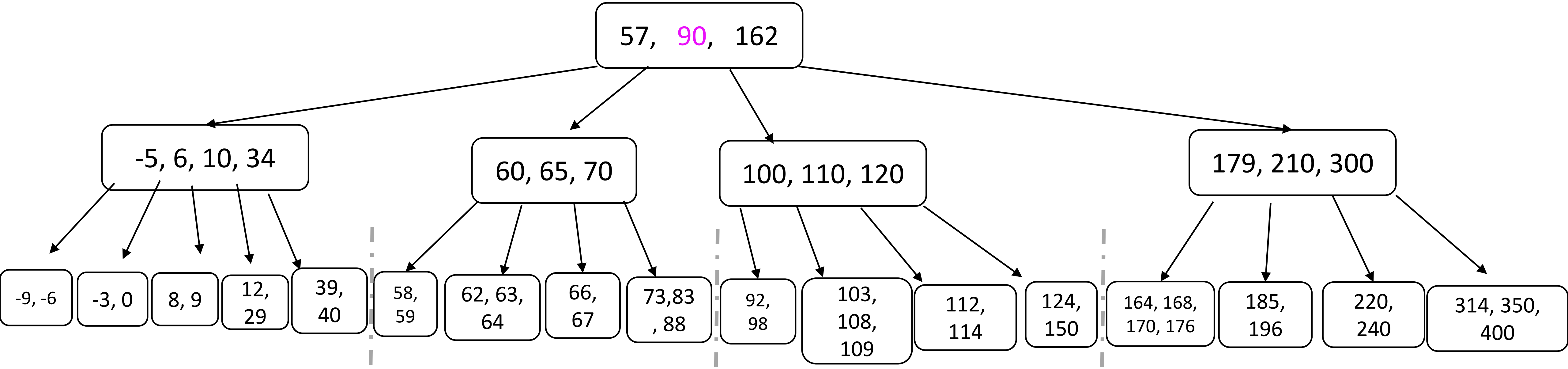
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

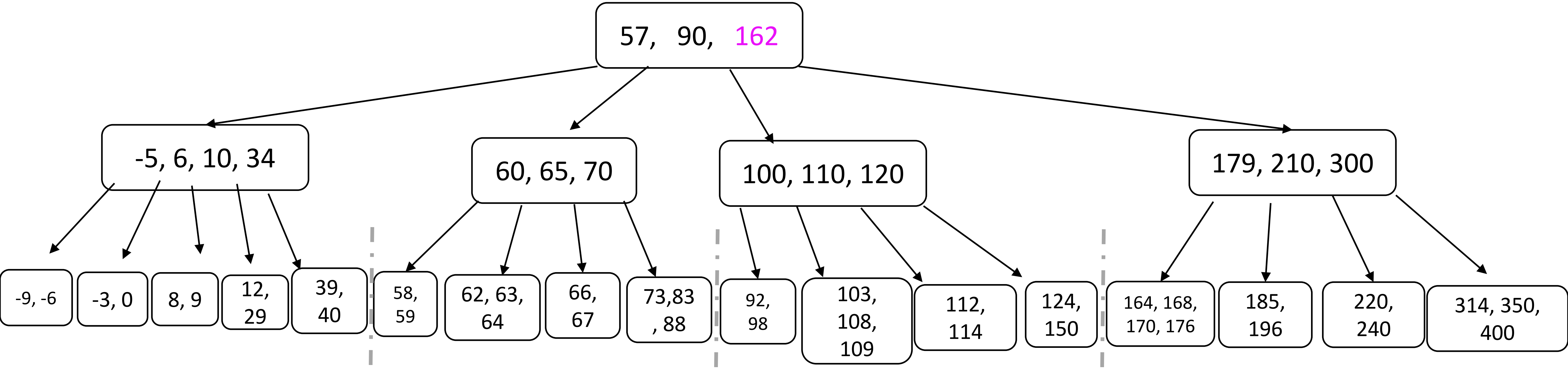
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

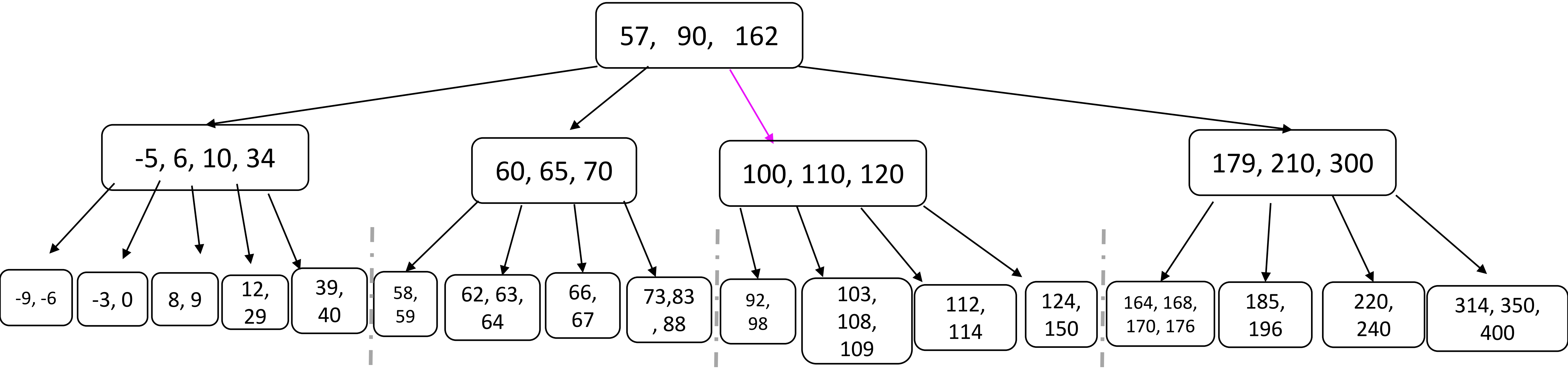
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

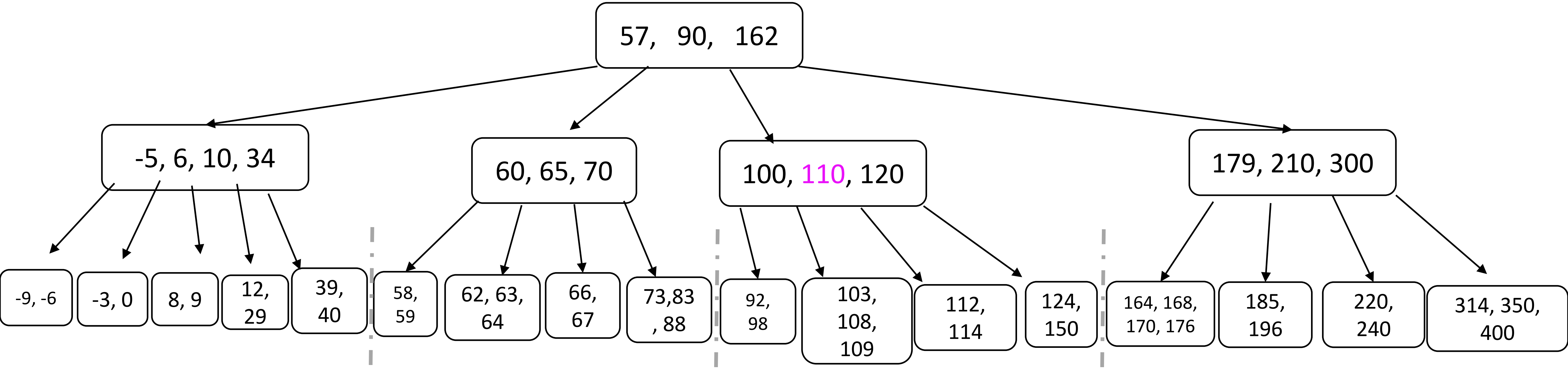
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

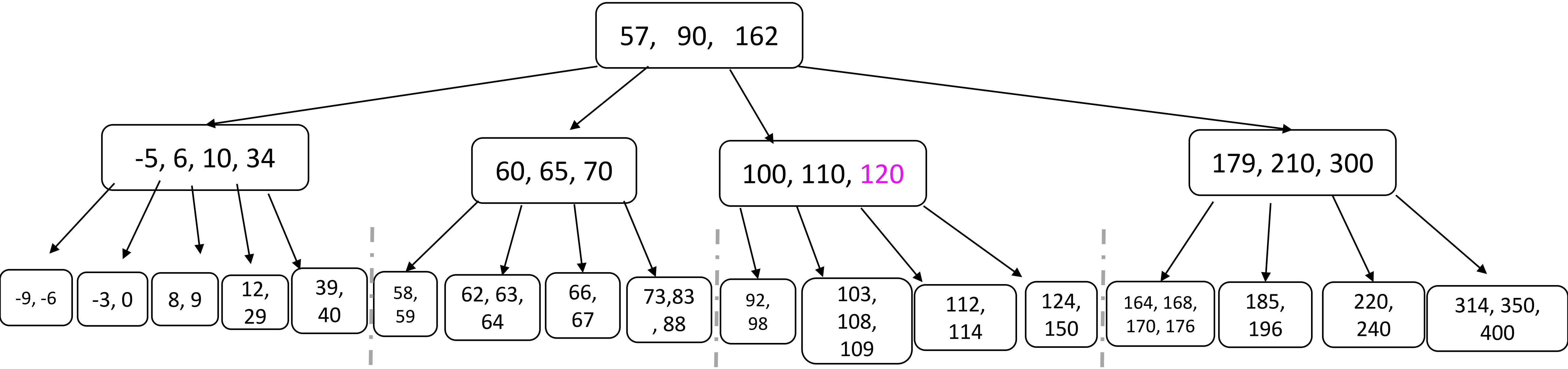
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

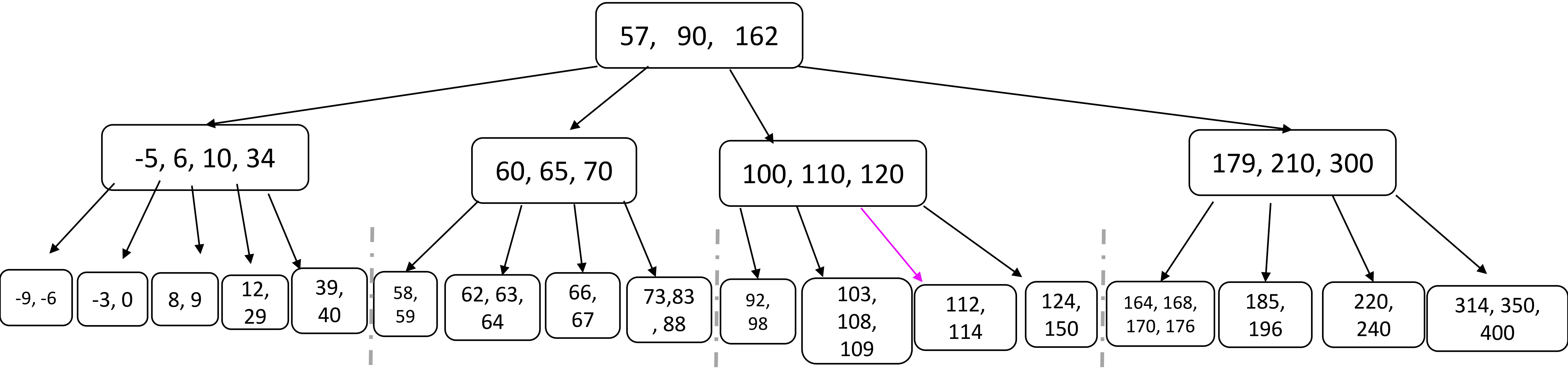
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

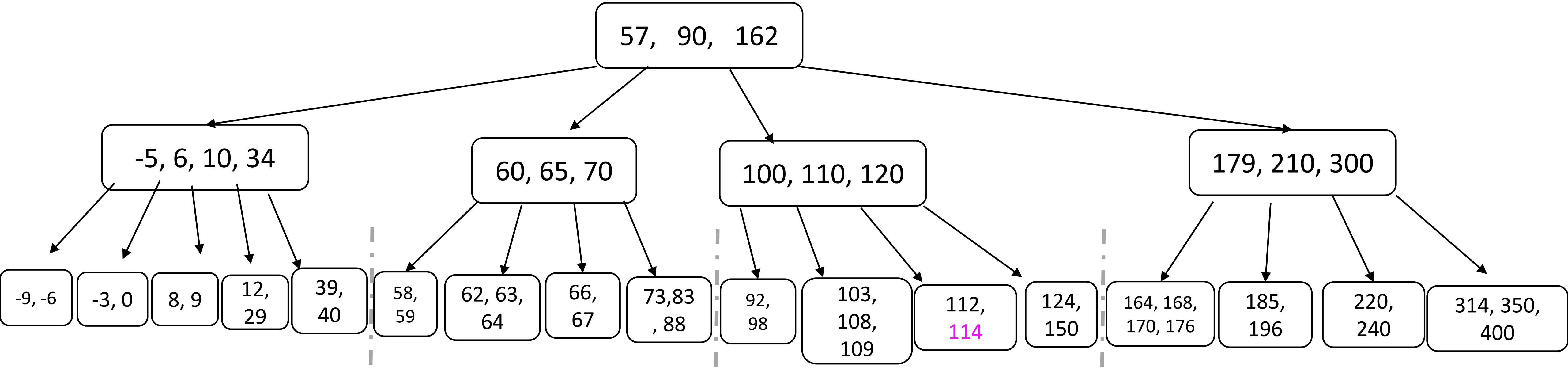
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

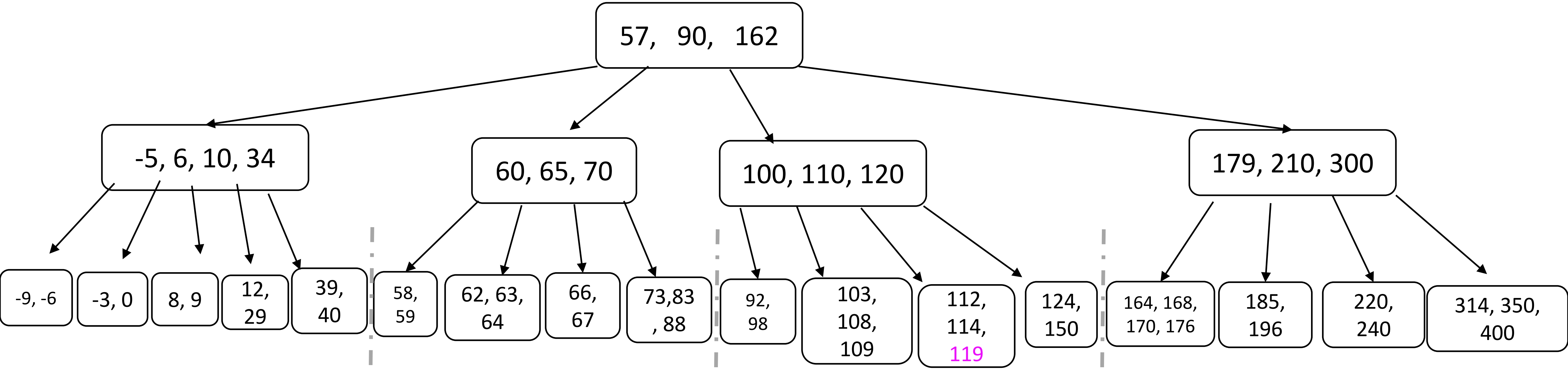
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

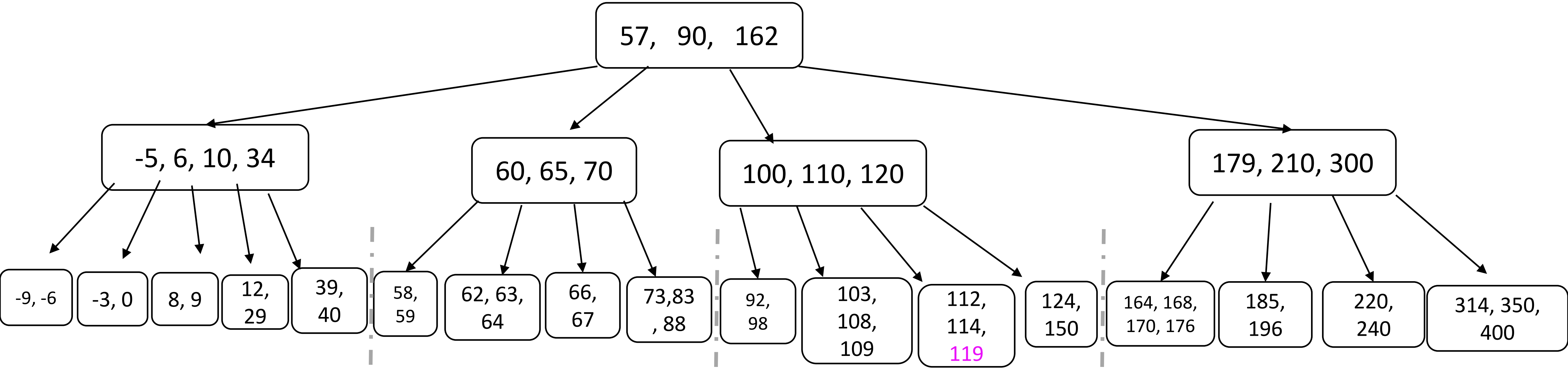
Example #0: No overflows



- Task: Insert 119

In all examples, $p = 5$

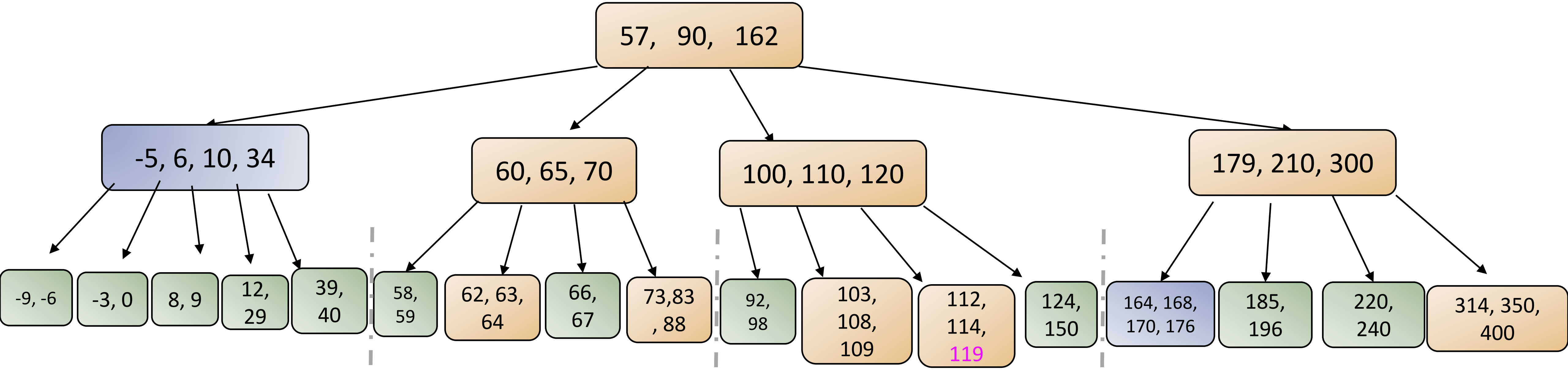
Example #0: No overflows



- Task: Insert 119
- No overflow at the leaf, so no problem

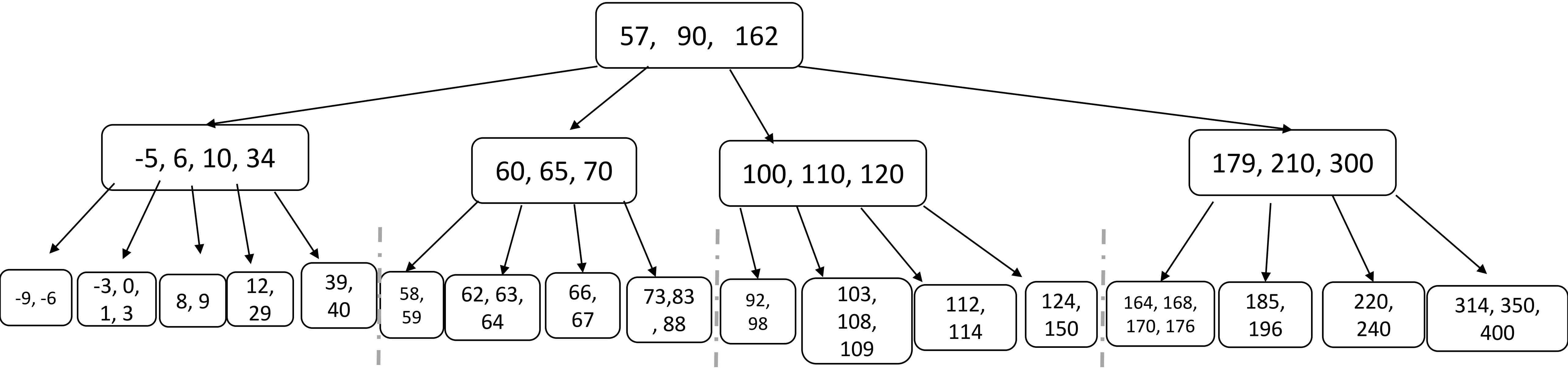
In all examples, $p = 5$

Example #0: No overflows



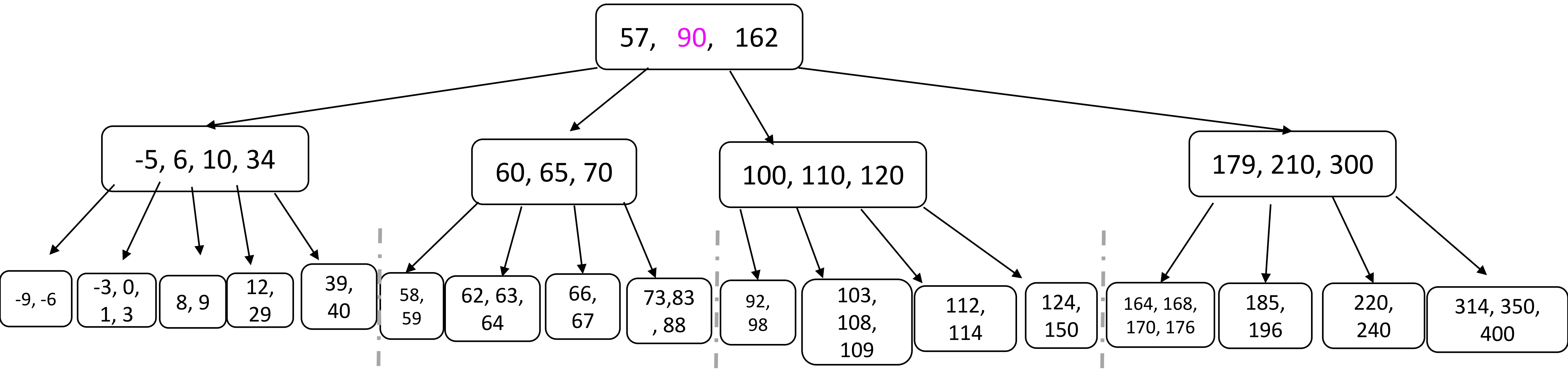
- **Side notes:**
  1. Only 2 nodes with maximum amount of $p - 1 = 4$ keys: That's ok.
  2. Bunch of leaf nodes holding only $2 = \left\lceil \frac{p}{2} \right\rceil - 1$ keys; also ok
  3. Bunch of inner and leaf nodes holding $3 = \left\lceil \frac{p}{2} \right\rceil$ keys; also ok

Example #1: Pick either sibling



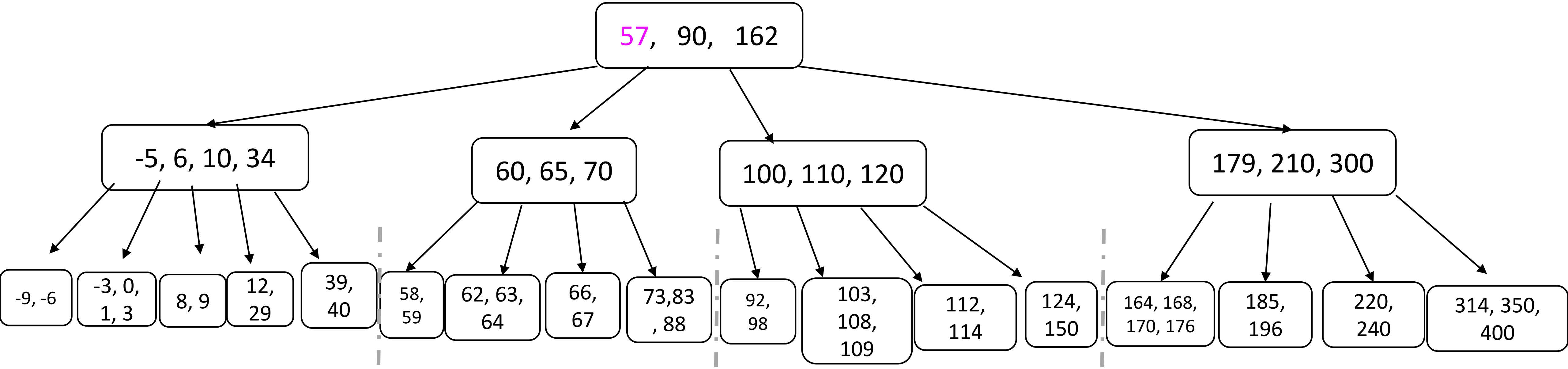
- Task: Insert -1

Example #1: Pick either sibling



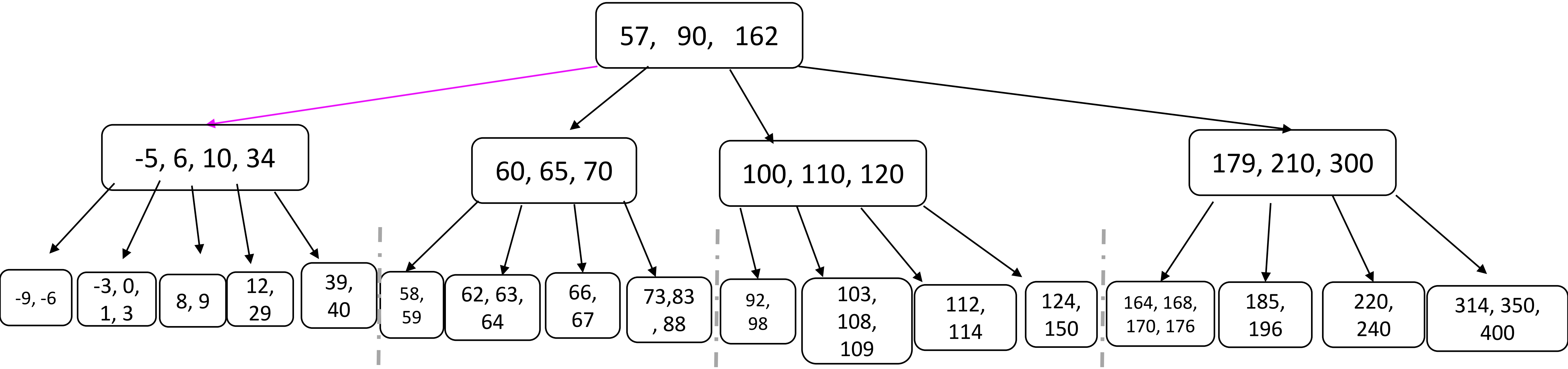
- Task: Insert -1

Example #1: Pick either sibling



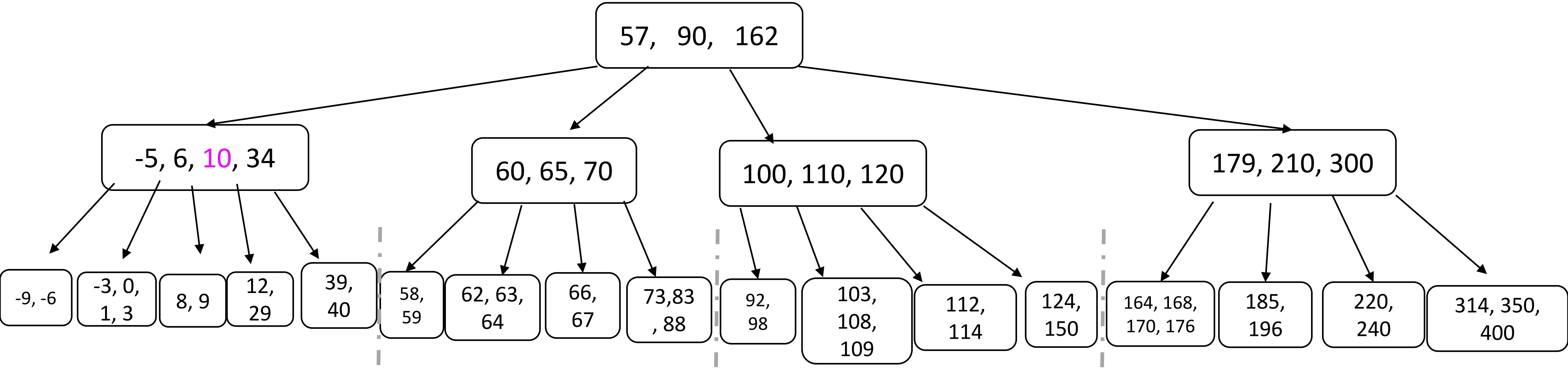
- Task: Insert -1

Example #1: Pick either sibling



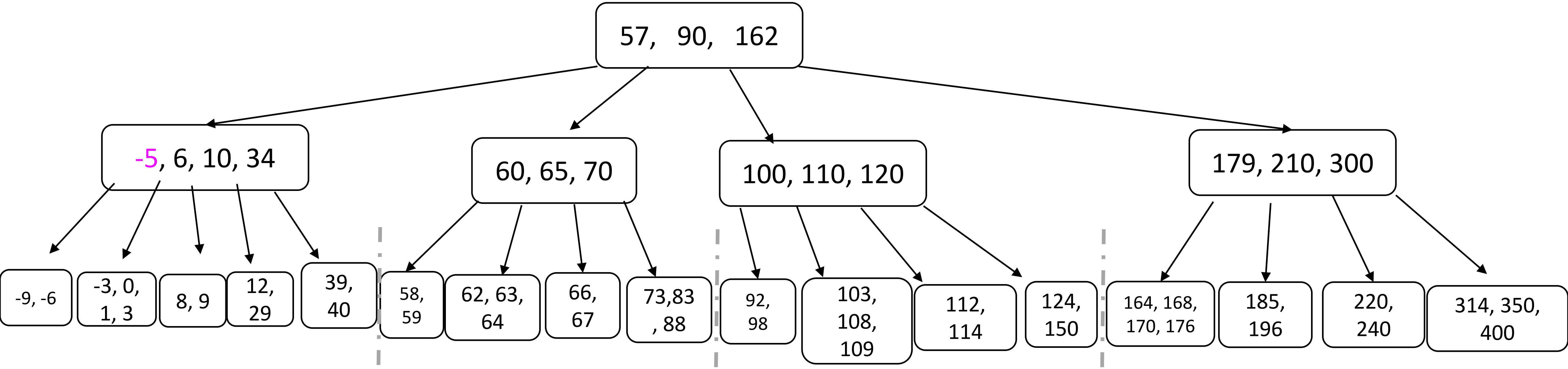
- Task: Insert -1

Example #1: Pick either sibling



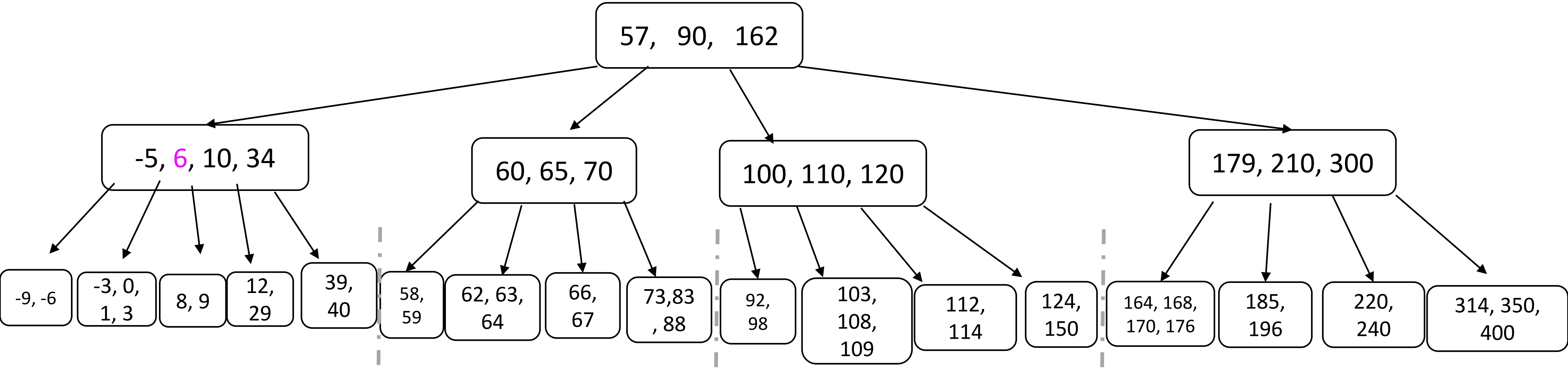
- Task: Insert -1

Example #1: Pick either sibling



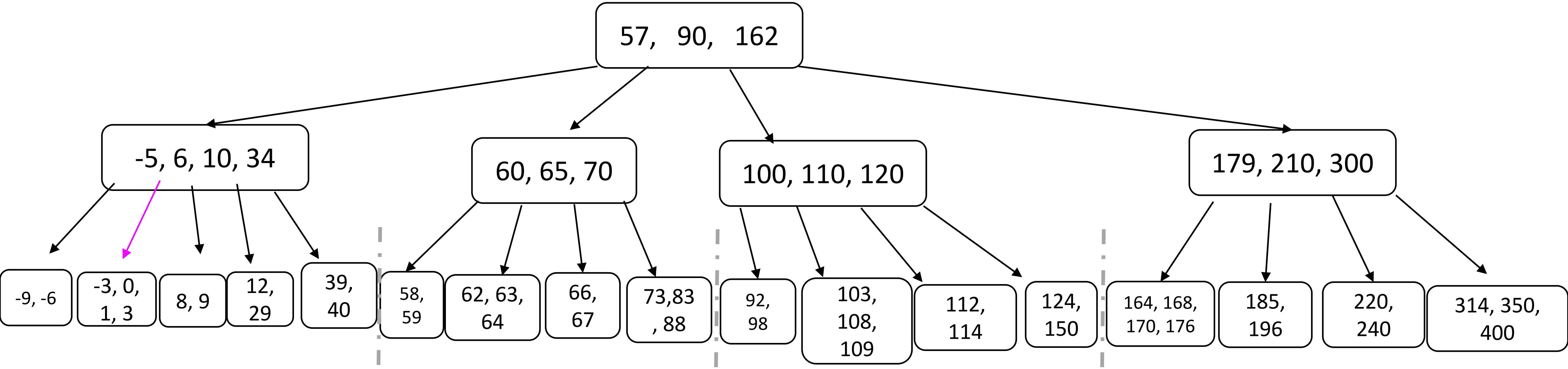
- Task: Insert -1

Example #1: Pick either sibling



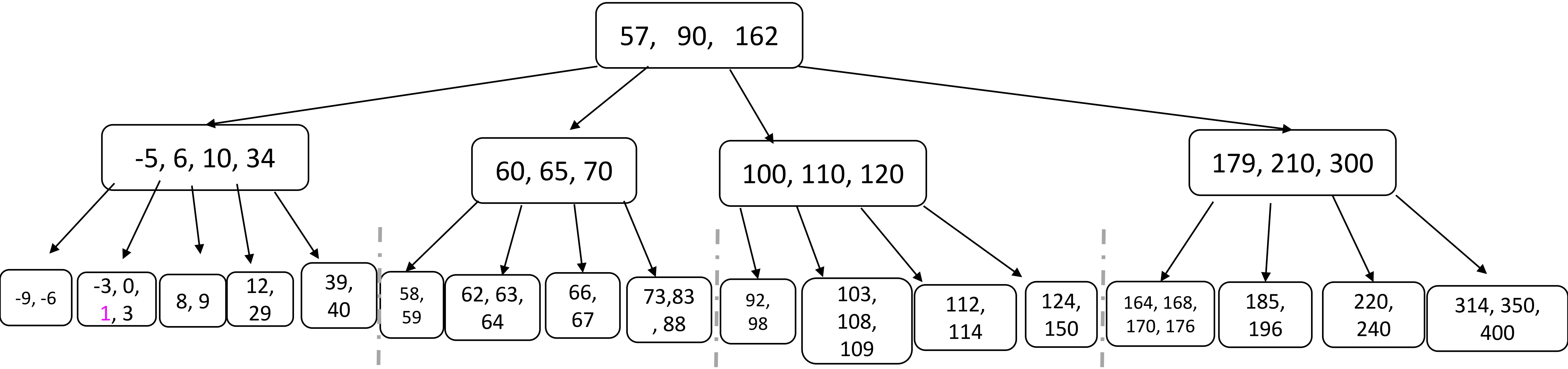
- Task: Insert -1

Example #1: Pick either sibling



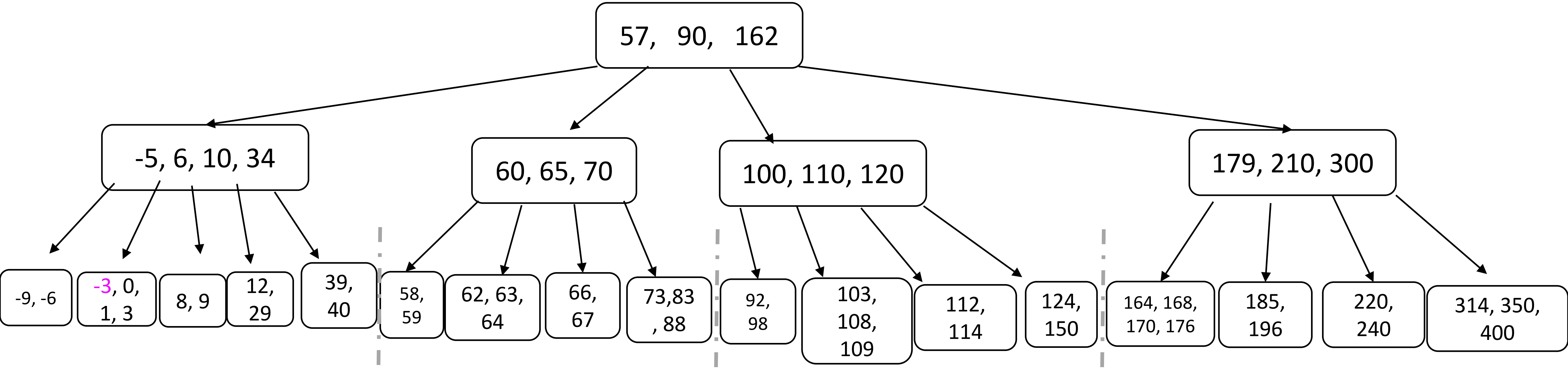
- Task: Insert -1

Example #1: Pick either sibling



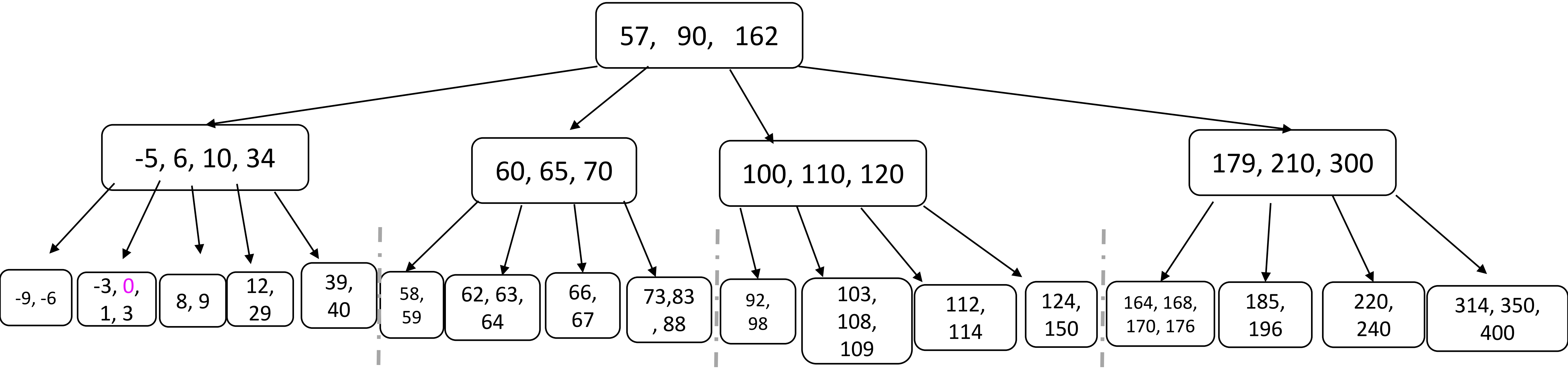
- Task: Insert -1

Example #1: Pick either sibling



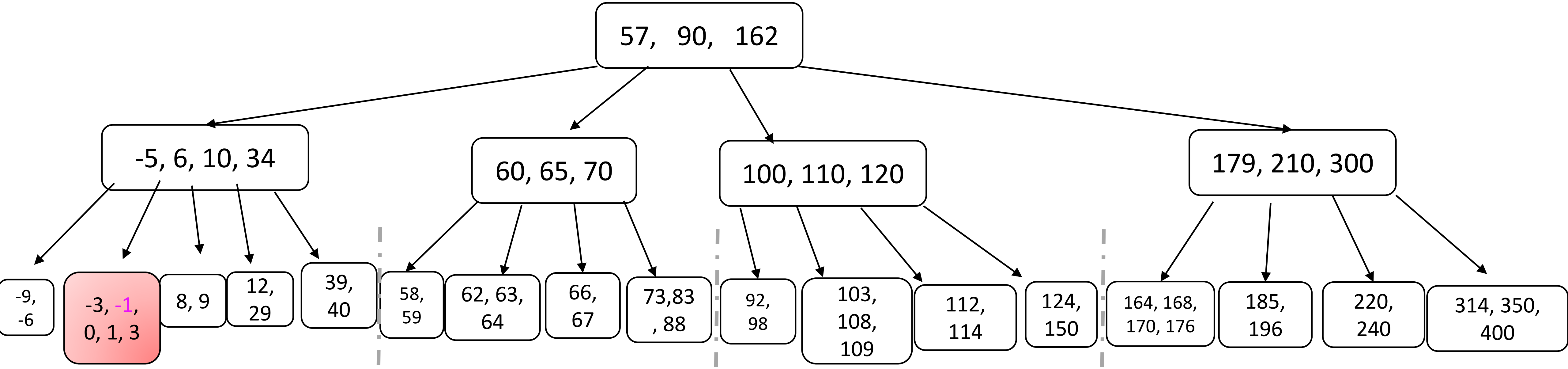
- Task: Insert -1

Example #1: Pick either sibling



- Task: Insert -1

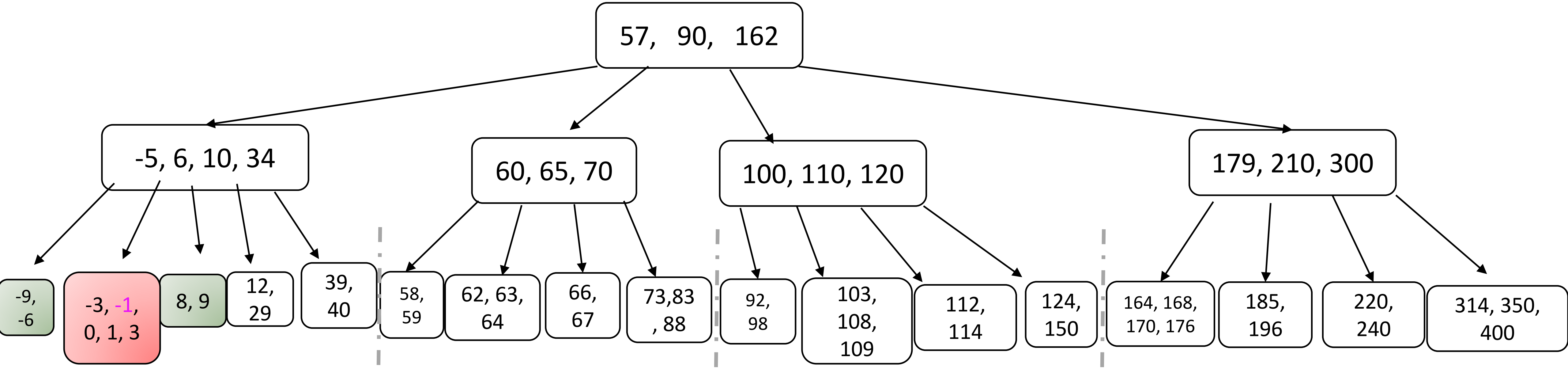
Example #1: Pick either sibling



- Task: Insert -1

Overflow ☹

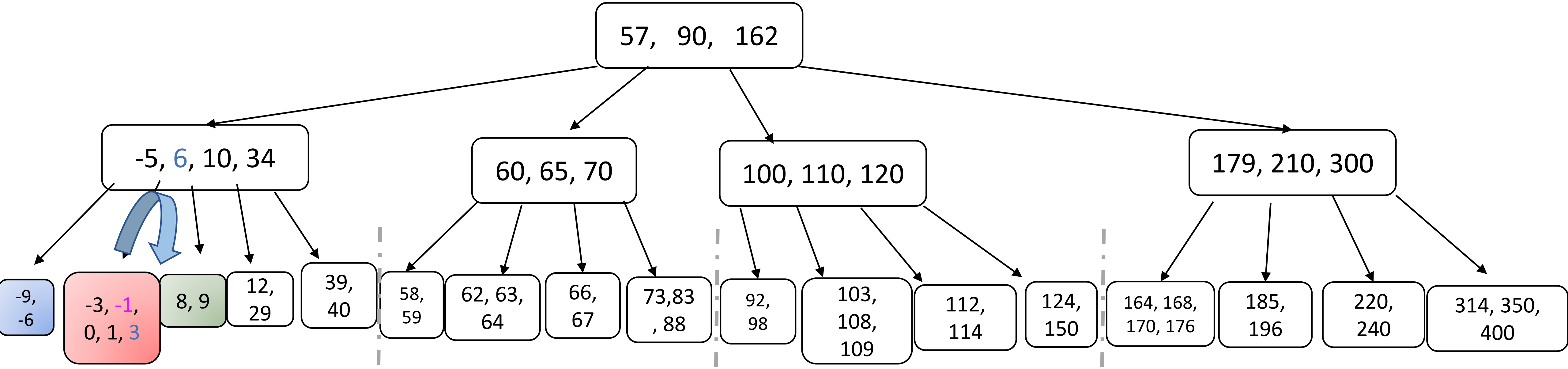
Example #1: Pick either sibling



- Task: Insert -1

No worries! I can rotate to either sibling ☺

Example #1: Pick either sibling

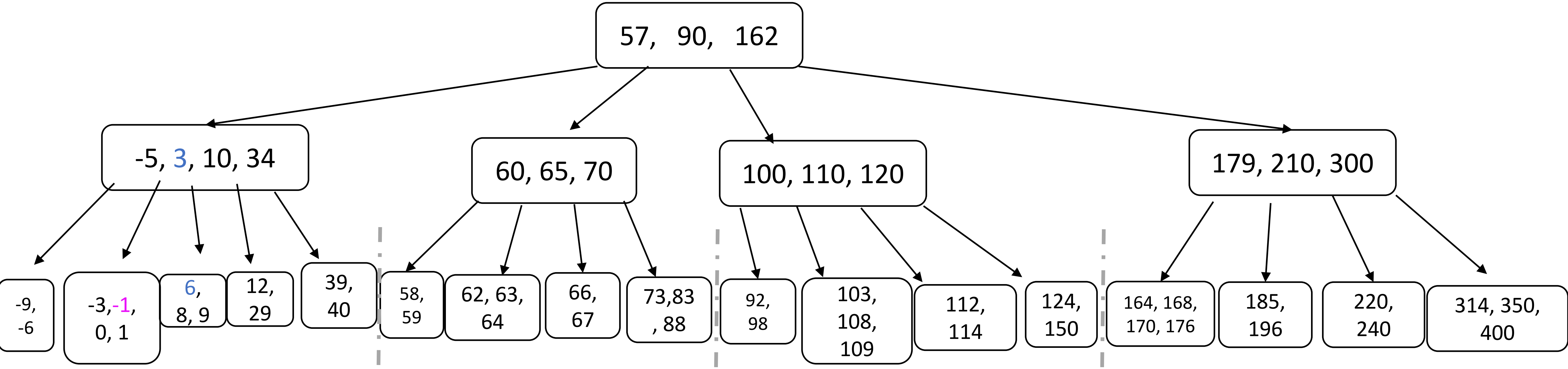


- Task: Insert -1

E.g pick right sibling, rotate 3 rightwards

(This is a remnant from an older semester: In Fall 19, we make the convention of rotation rightwards first)

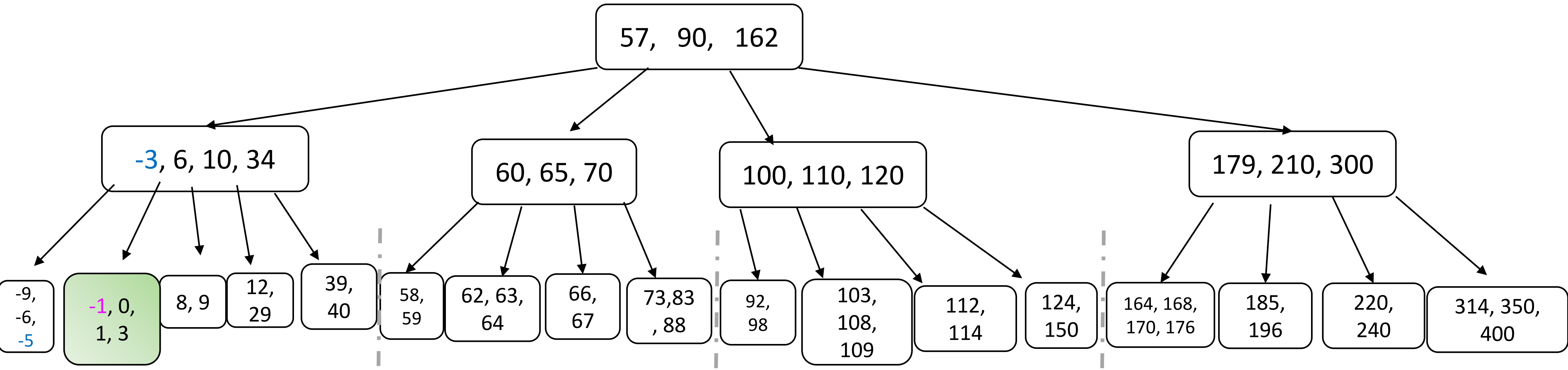
Example #1: Pick either sibling



- Task: Insert -1

Careful: This means that ***the parent's key*** (6) is the one that lands on the sibling node… 3 goes to the parent!

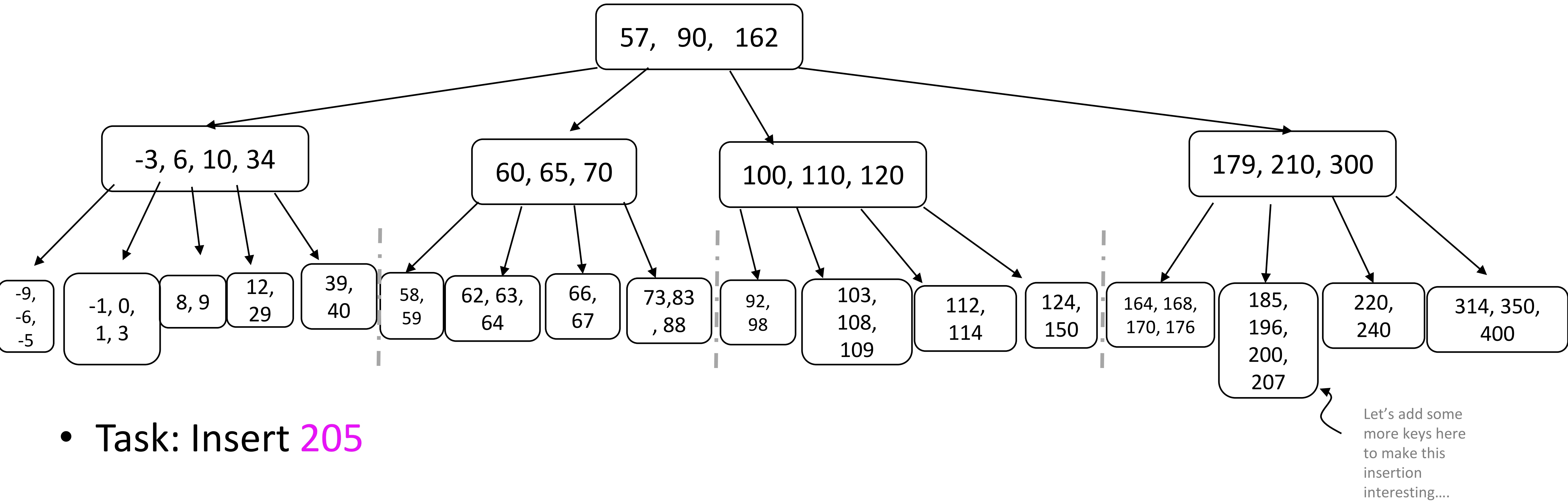
Example #1: Pick either sibling



- Task: Insert -1

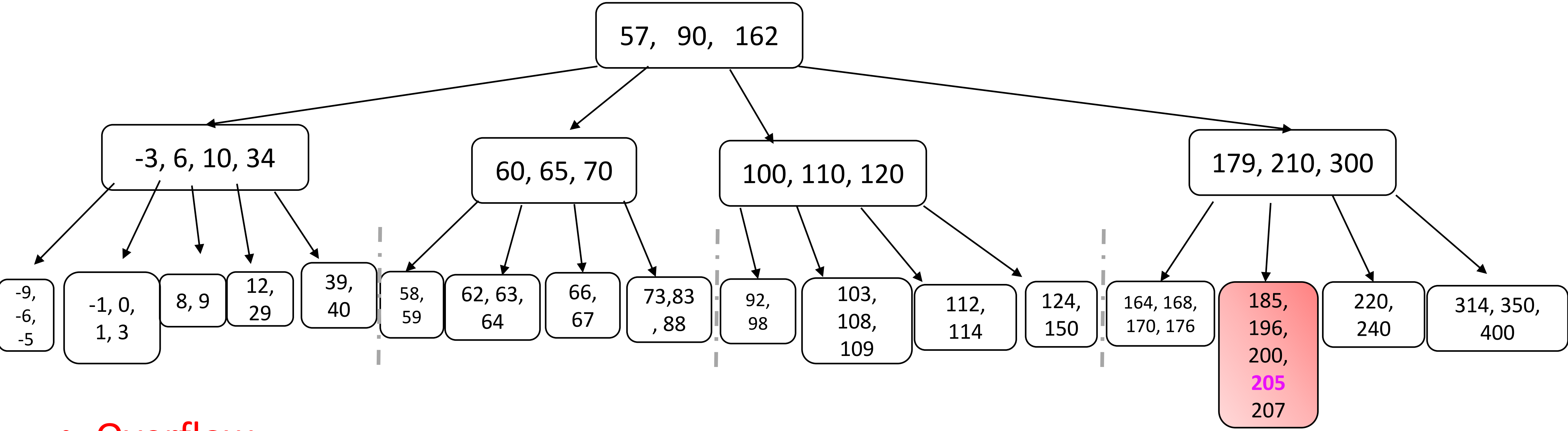
Overflow fixed, key inserted ☺

Example #2: Only one sibling with space!



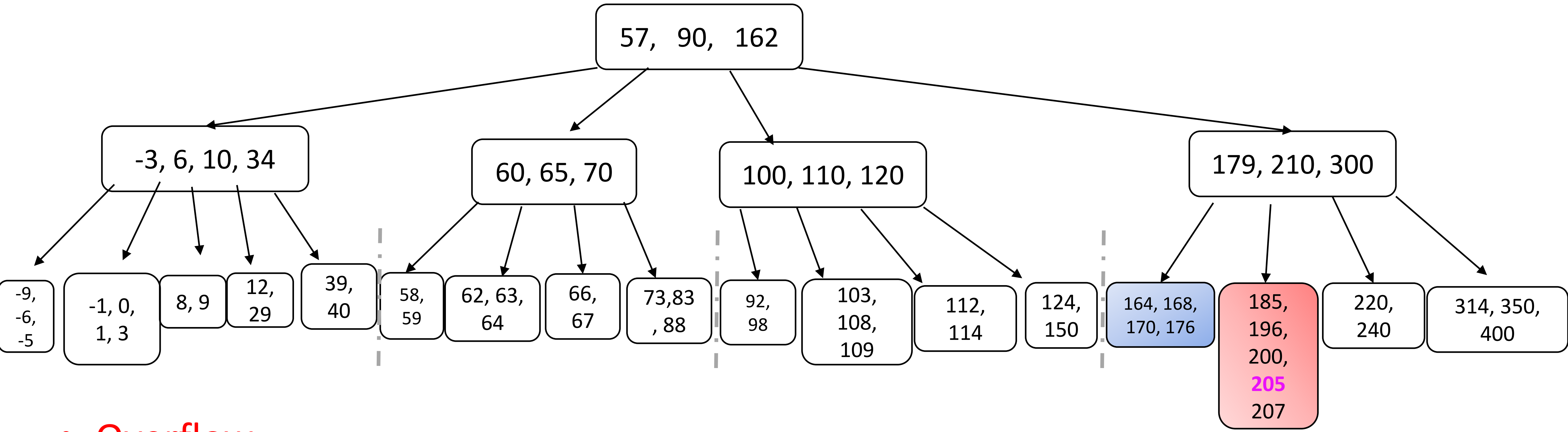
- Task: Insert 205

Example #2: Only one sibling with space!



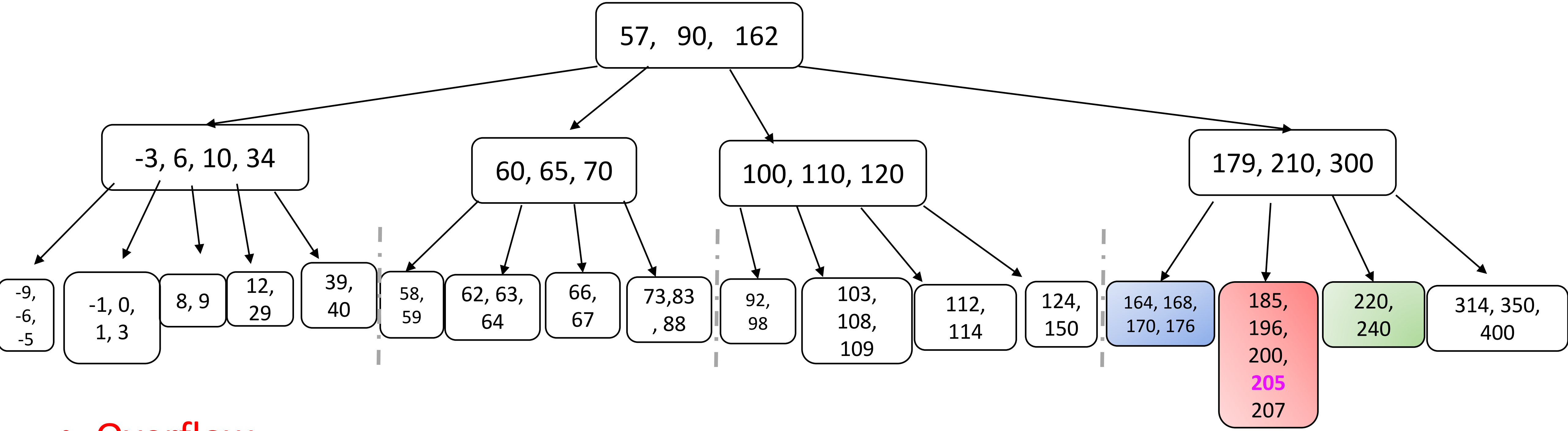
- Overflow…

Example #2: Only one sibling with space!



- Overflow…
- Can’t rotate 185 leftwards because left sibling full ☹

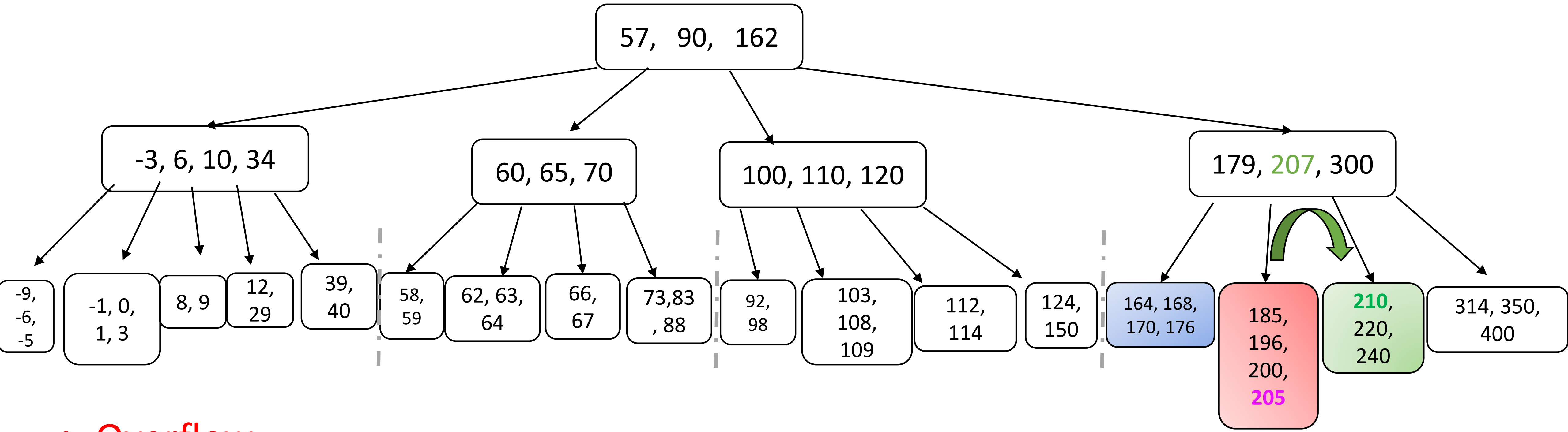
Example #2: Only one sibling with space!



- Overflow…
- Can’t rotate 185 leftwards because left sibling full ☹
- But can rotate 207 rightwards because right sibling non-full ☺

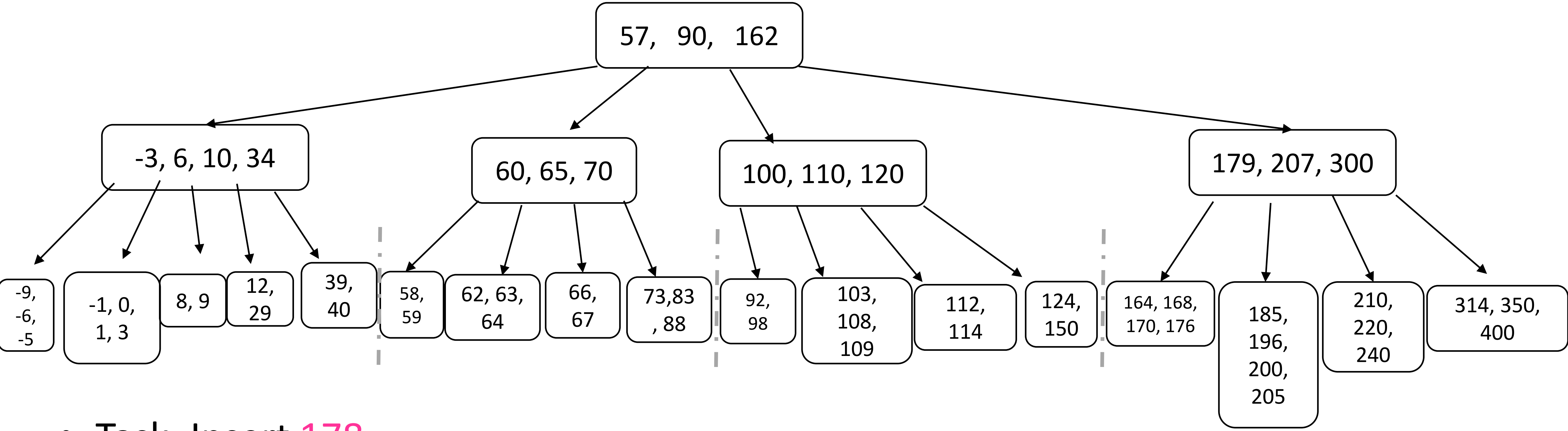
Once again, for Fall 19, we would make the convention of going right first (so we would check it ***first***).

Example #2: Only one sibling with space!



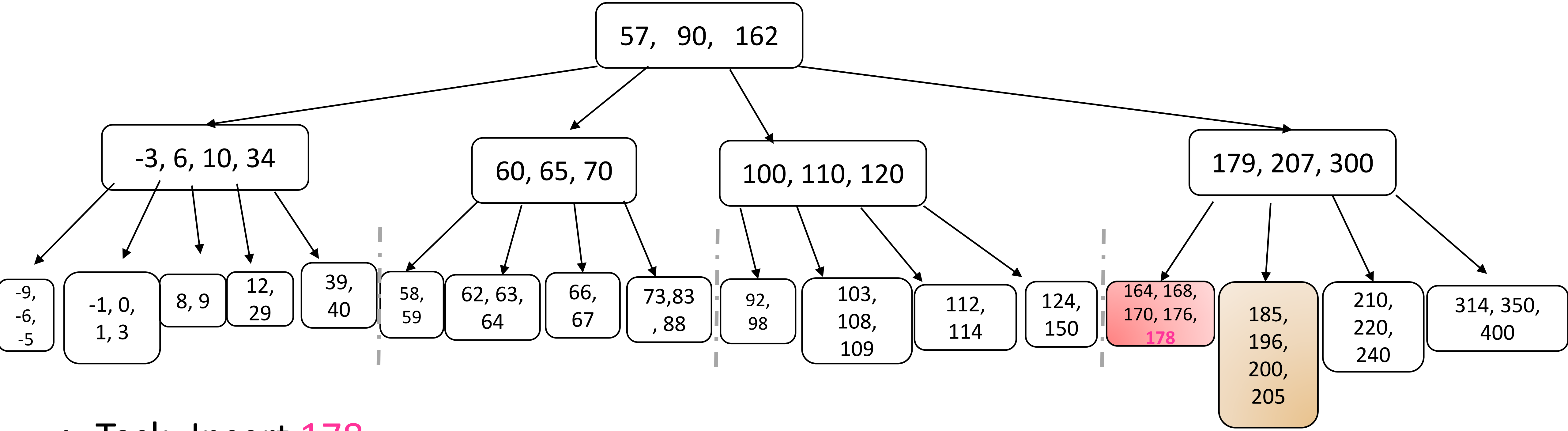
- Overflow…
- Can’t rotate 185 leftwards because left sibling full ☹
- But can rotate 207 rightwards because right sibling non-full ☺

Example #3: Examine first sibling node!



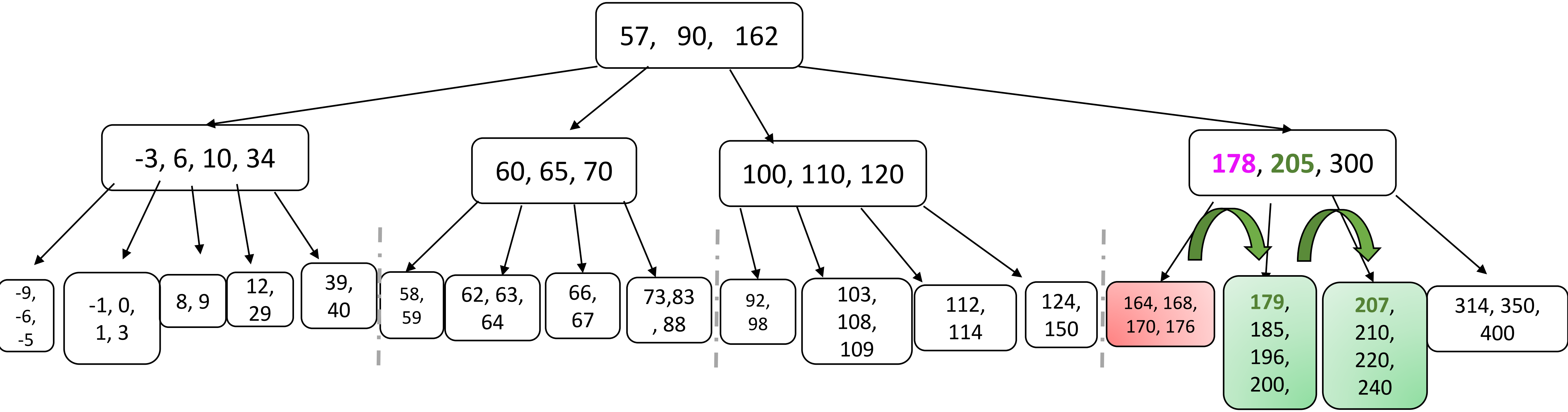
- Task: Insert 178

Example #3: Examine first sibling node!



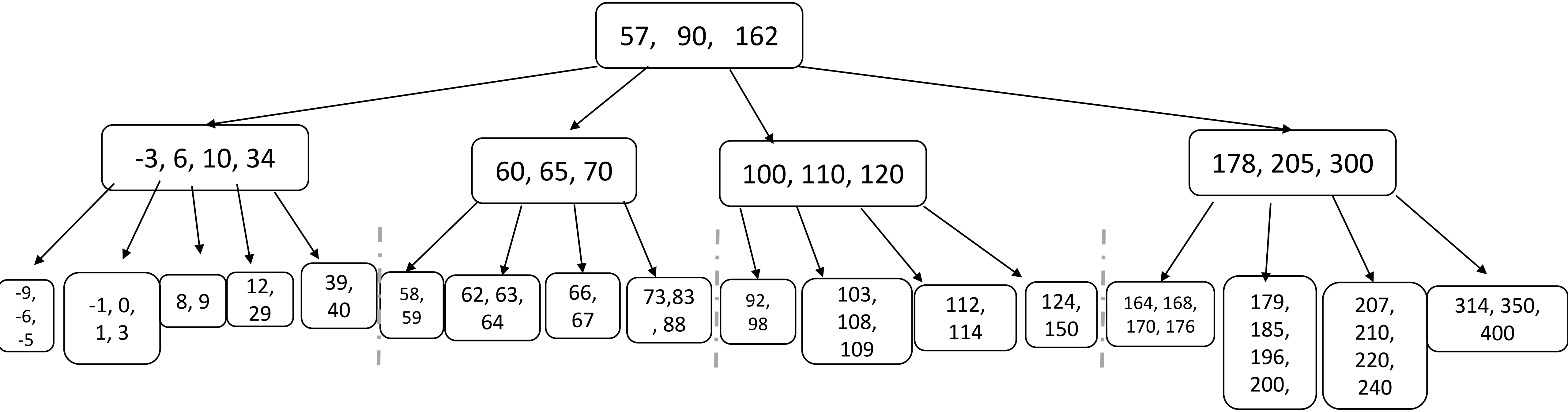
- Task: Insert 178
- Overflow… ☹
- Only sibling also full ☹

Example #3: Examine first sibling node!



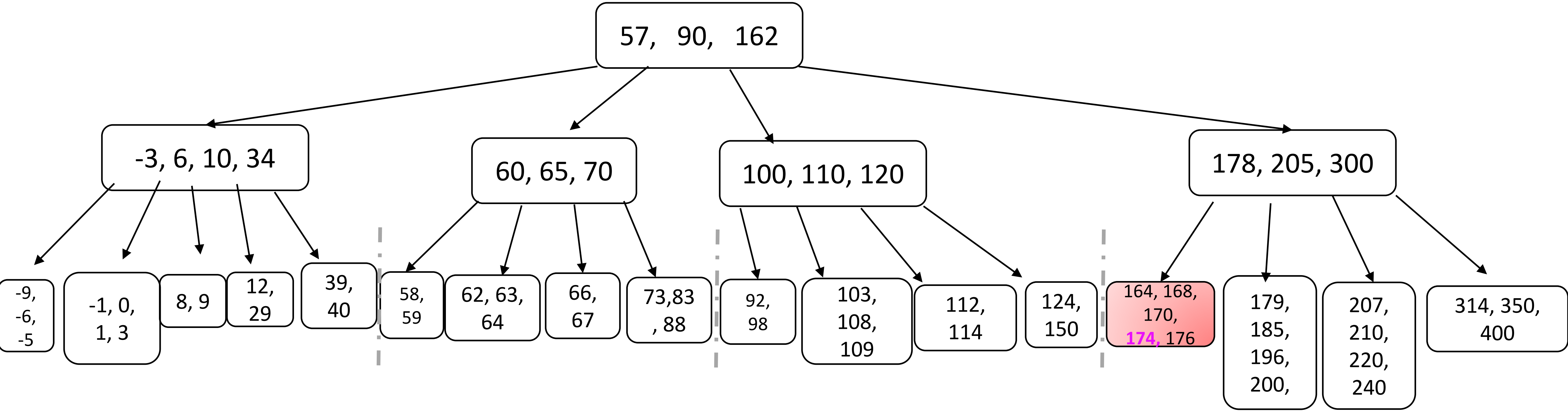
- Task: Insert 178
- Overflow… ☹
- Only sibling also full ☹
- Solution: Rotate 205 rightwards and 178 (new key!) rightwards! ☺

Example #3: Examine first sibling node!



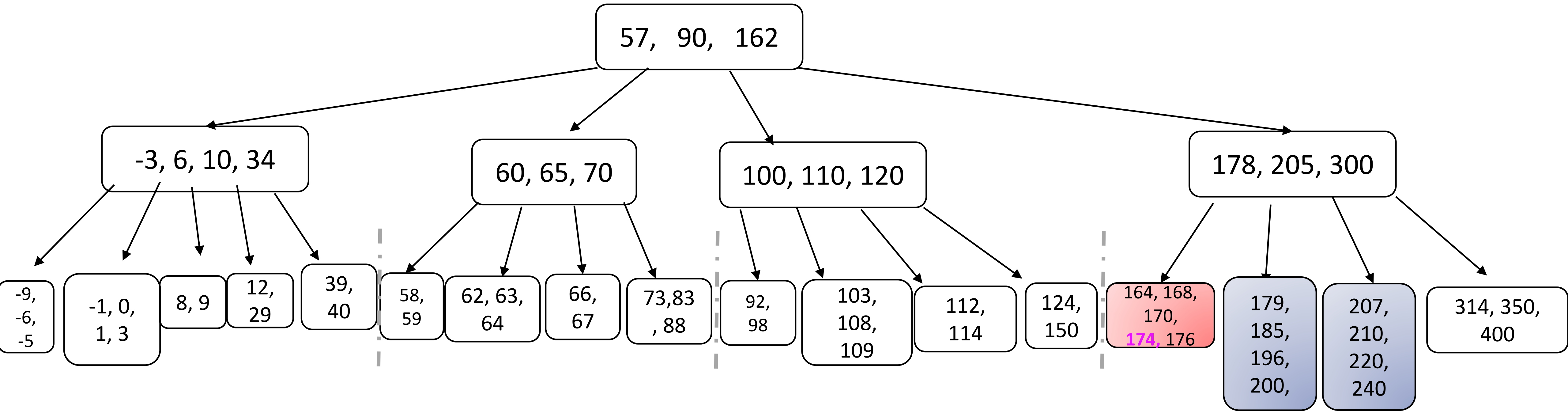
- Task: Insert 174

Example #3: Examine first sibling node!



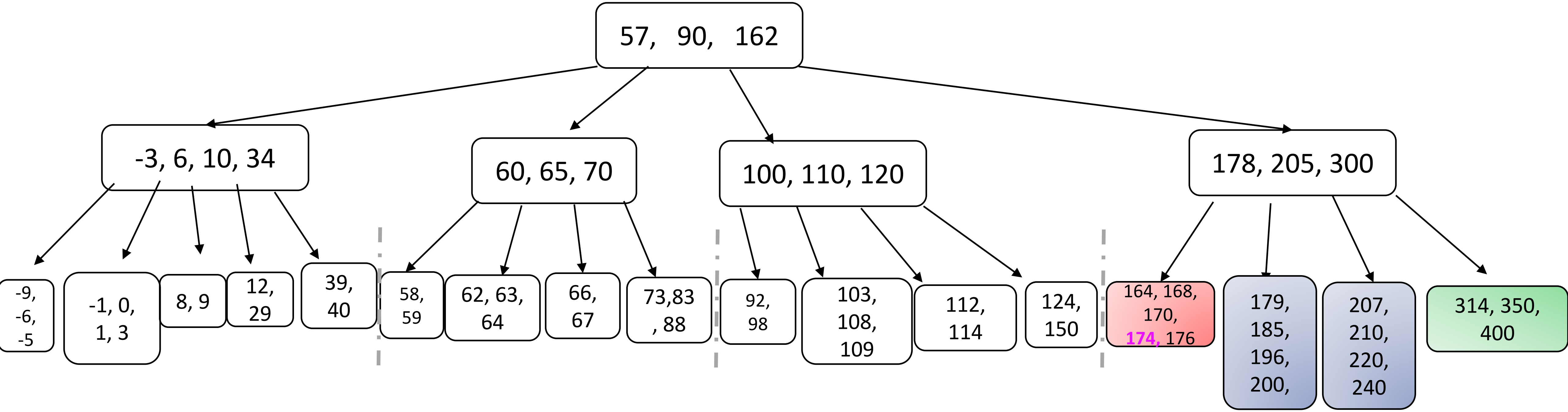
- Task: Insert 174
- Overflow ☹

Example #3: Examine first sibling node!



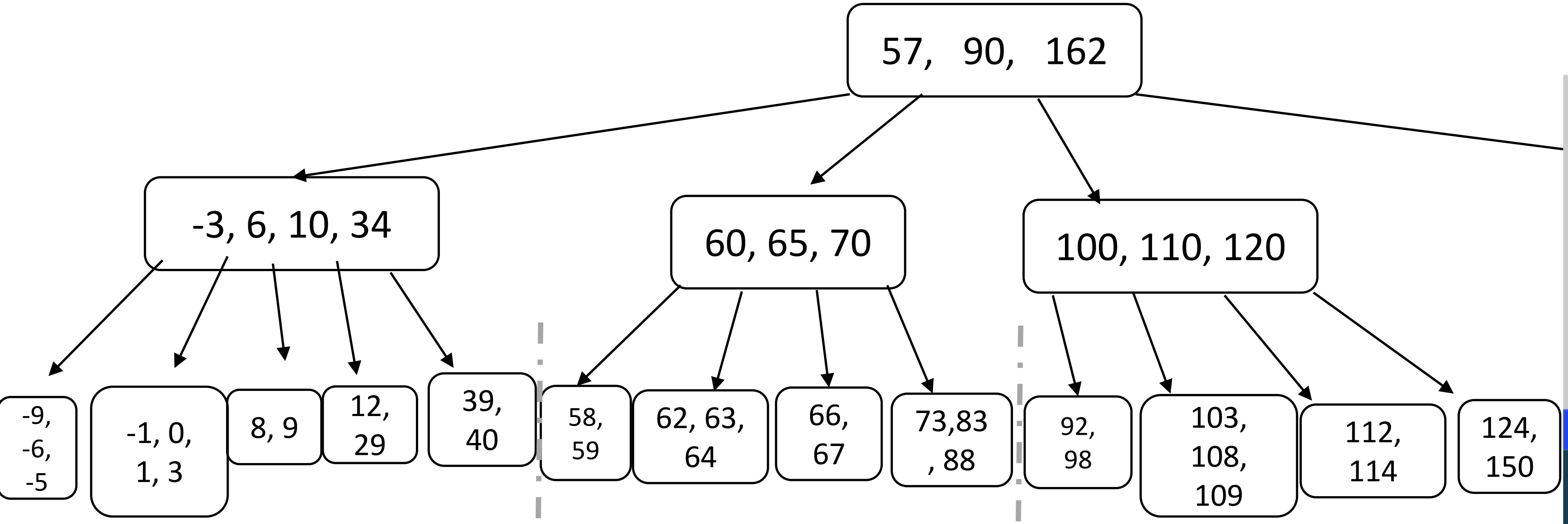
- Task: Insert 174
- Overflow ☹
- 1st and 2nd sibling don't have space to rotate keys to ☹

Example #4: Examine second sibling node!



- Task: Insert 174
- Overflow ☹
- 1st and 2nd sibling don't have space to rotate keys to ☹
- But 3rd sibling does! ☺

Example #4: Examine second sibling node!



- Task: Insert 174
- Overflow ☹
- 1^{st} and 2^{nd} sibling don’t have space to rotate keys to ☹
- But 3^{rd} sibling does! ☺

WE INTERRUPT YOUR REGULARLY SCHEDULED PROGRAM FOR AN INVESTIGATION INTO SIBLING SEARCH!

So, should I always rotate as far away as I can?

So, should I always rotate as far away as I can?

- **UNCLEAR.**

So, should I always rotate as far away as I can?

- **UNCLEAR.**
- In the 70s and 80s, yes, absolutely
  - Memory expensive: inner nodes in disk as well means we want to minimize height.

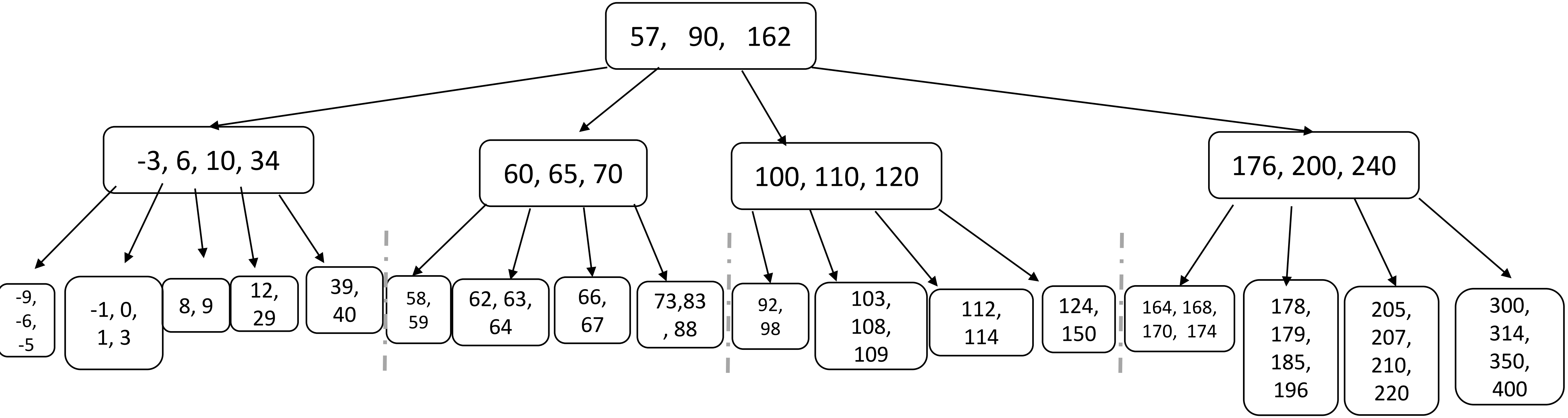
So, should I always rotate as far away as I can?

- **UNCLEAR.**
- In the 70s and 80s, yes, absolutely
  - Memory expensive: inner nodes in disk as well means we want to minimize height.
- Nowadays, can fit the index (B-Tree) in memory.
  - So might not care about adding to the height that much

So, should I always rotate as far away as I can?

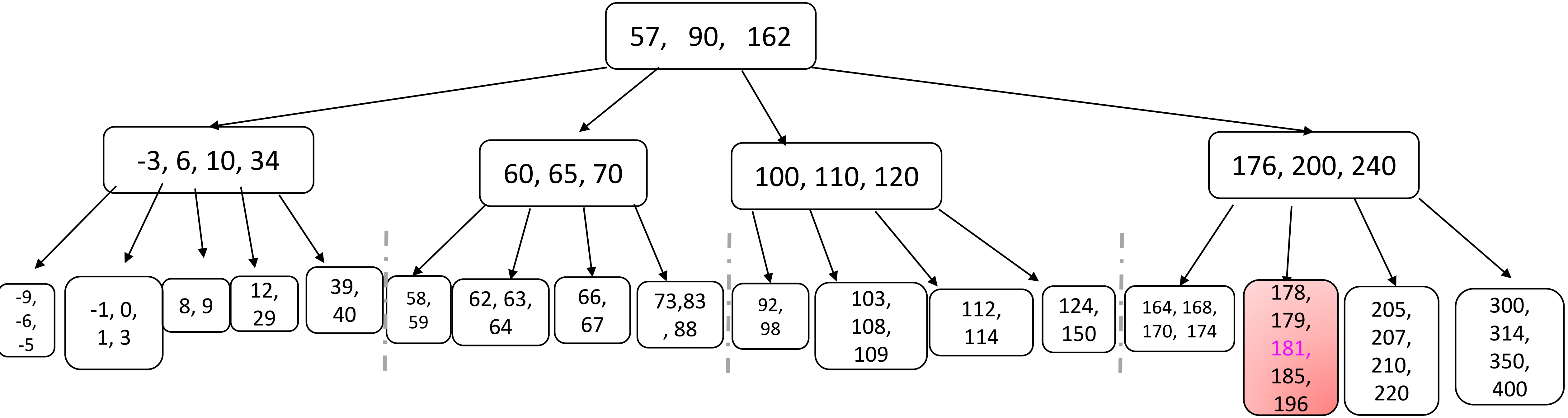
- **UNCLEAR.**
- In the 70s and 80s, yes, absolutely
  - Memory expensive: inner nodes in disk as well means we want to minimize height.
- Nowadays, can fit the index (B-Tree) in memory.
  - So might not care about adding to the height that much
- **Our solution (for exams, projects, etc)**: supply an `int` parameter $hops$ to either insertion method or B-Tree constructor to indicate how far away you are willing to search for key space when you overflow.
  - $hops = -1$ can be used to indicate *"as far away as you want".*

Example #5: Splitting, parent has space



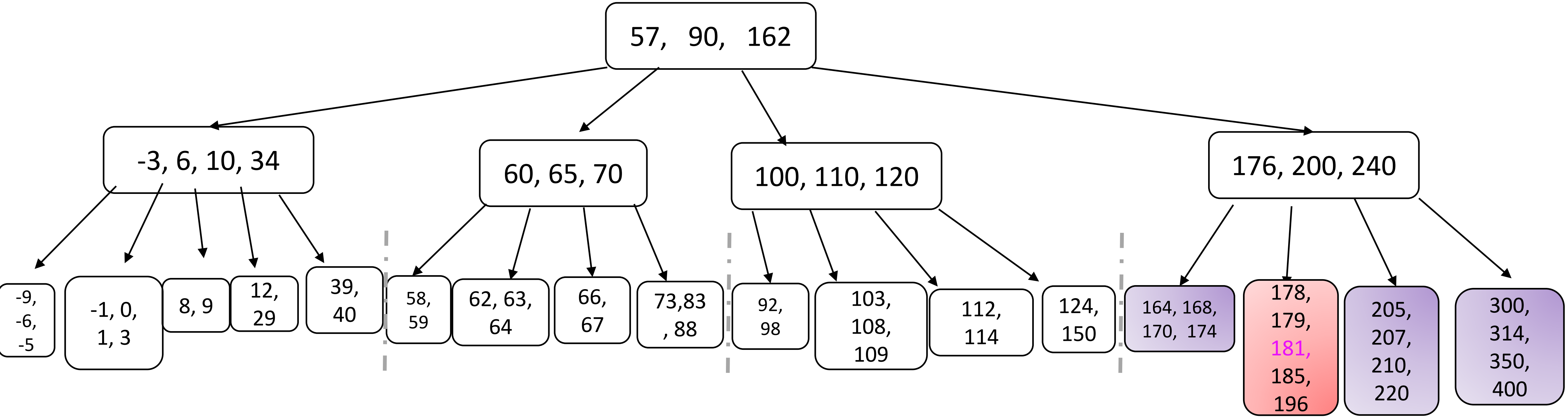
- Task: Insert 181

Example #5: Splitting, parent has space



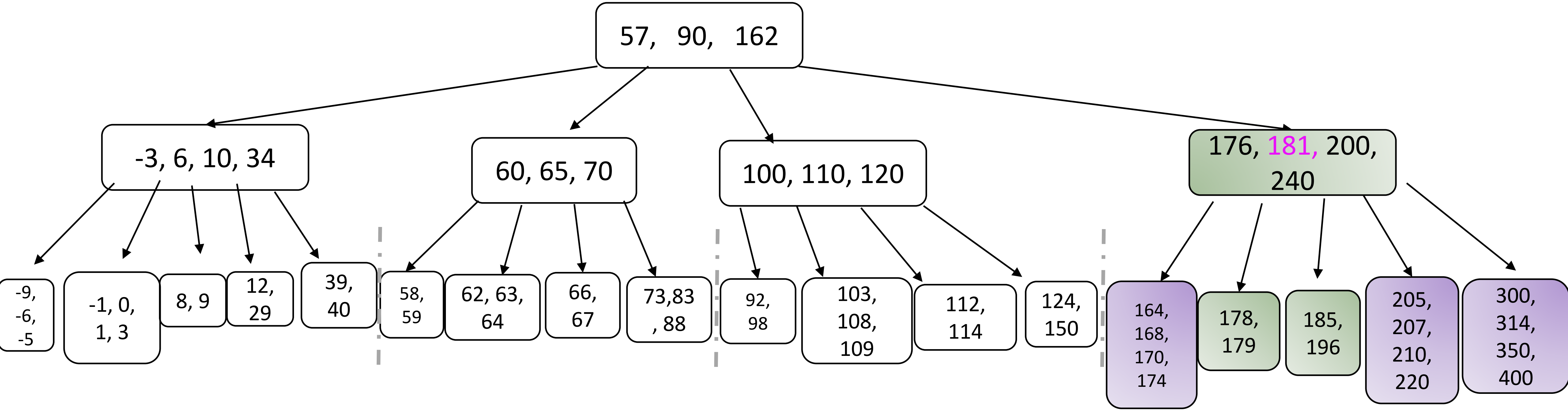
- Task: Insert 181
- Overflow ☹

Example #5: Splitting, parent has space



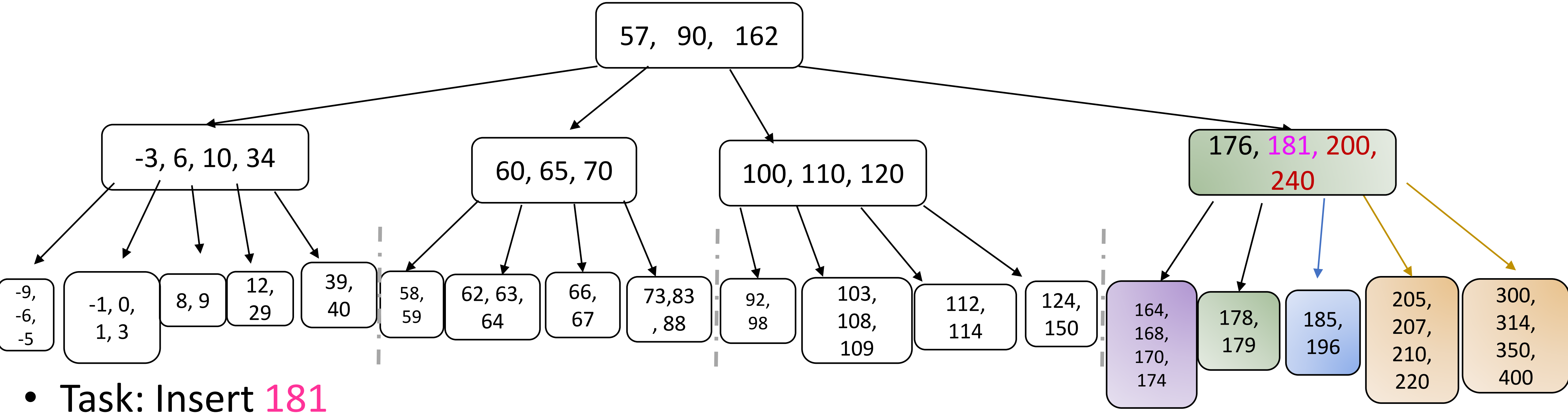
- Task: Insert 181
- Overflow ☹
- No sibling or cousin has space for rotating keys to! ☹

Example #5: Splitting, parent has space



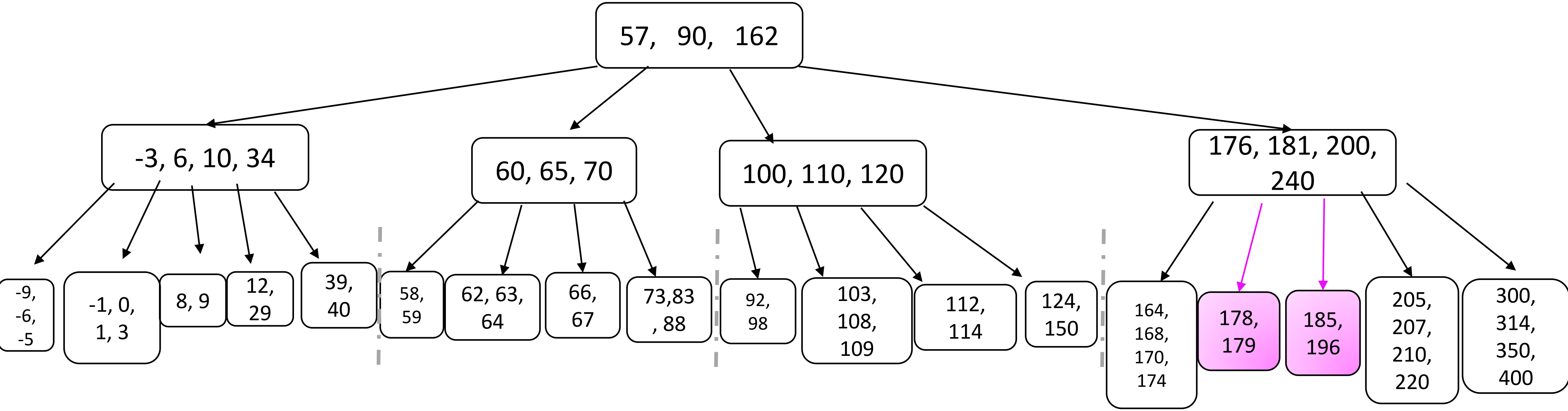
- Task: Insert 181
- Overflow ☹
- No sibling or cousin has space for rotating keys to! ☹
- Solution: Split the current node, and elevate middle key to the parent ☺

Example #5: Splitting, parent has space



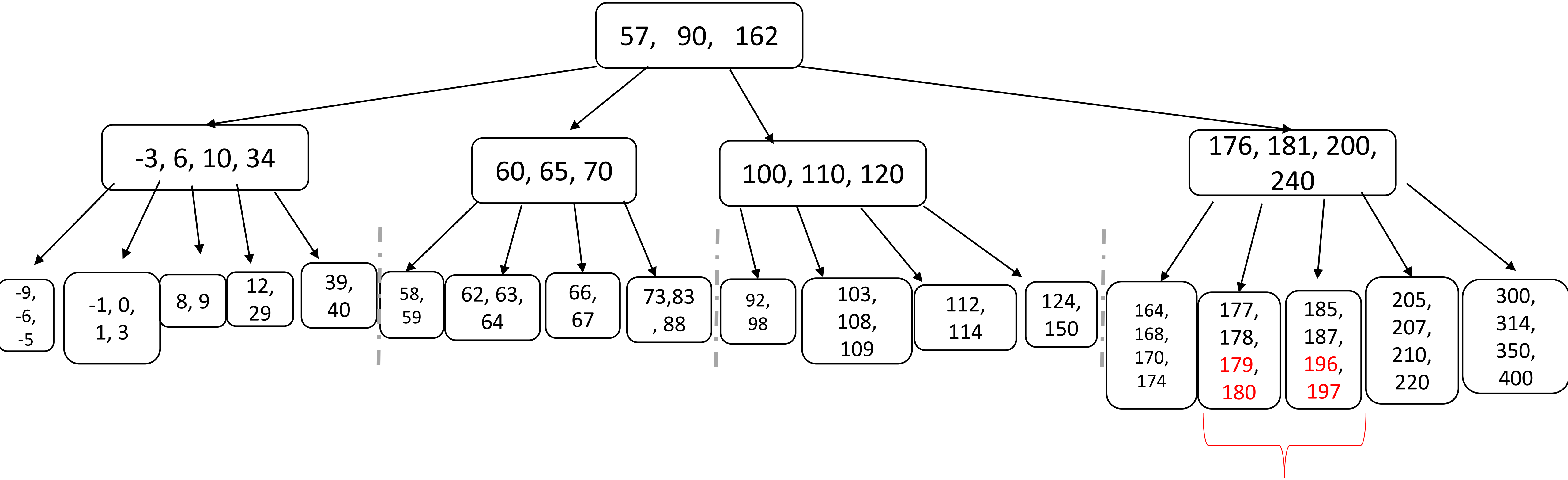
- Task: Insert 181
- Overflow ☹
- No sibling or cousin has space for rotating keys to! ☹
- Solution: Split the current node, and elevate middle key to the parent ☺
  - Parent has 1 additional child subtree now.

Interesting notes about splitting



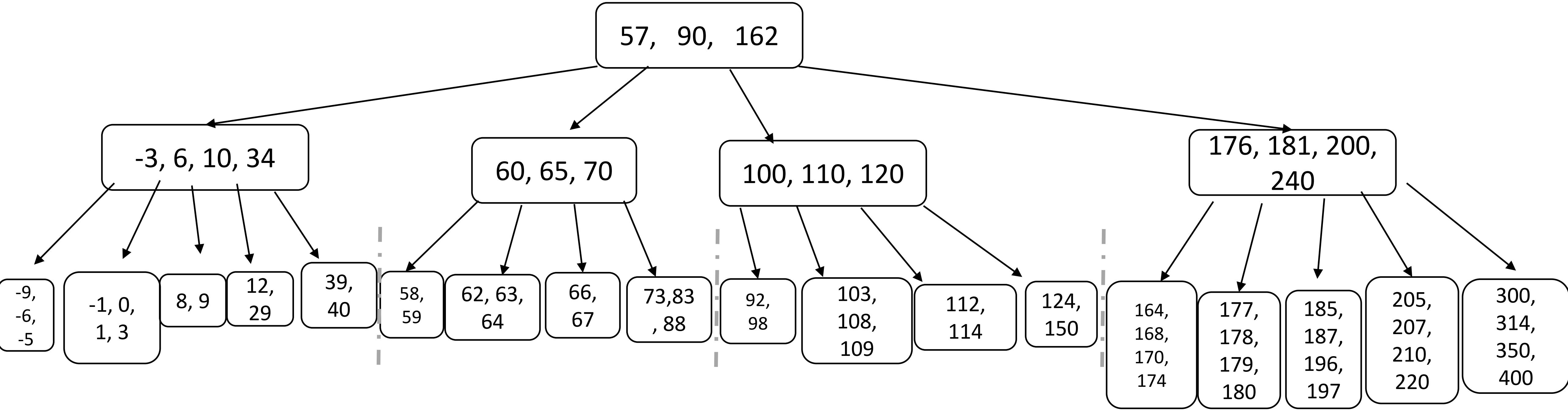
- **4** key insertions in the range [177, 199] will be solvable with **key rotations to the left or right**!
- Point: when a splitting in a certain range occurs, it's unlikely to happen again in the near future (*locality of reference*)

Example #6: Splitting, parent overflows and rotates



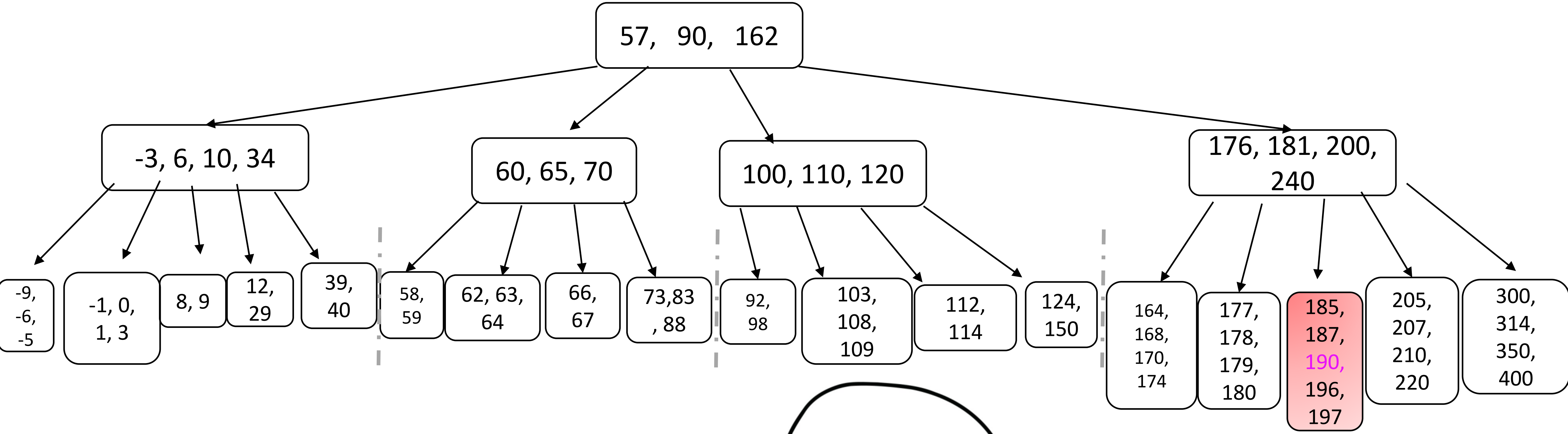
Added some keys here to make example interesting...

Example #6: Splitting, parent overflows and rotates

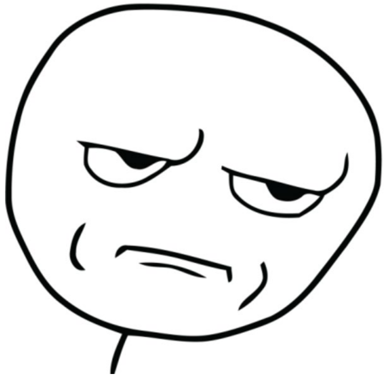


- Task: Insert 190

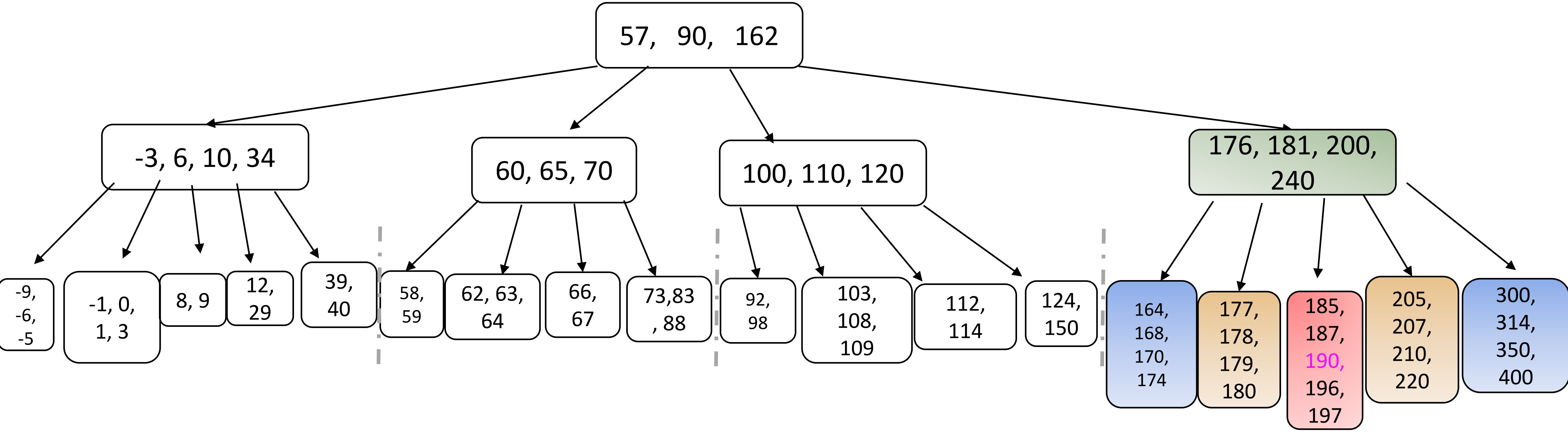
Example #6: Splitting, parent overflows and rotates



- Task: Insert 190
- Overflow ☹

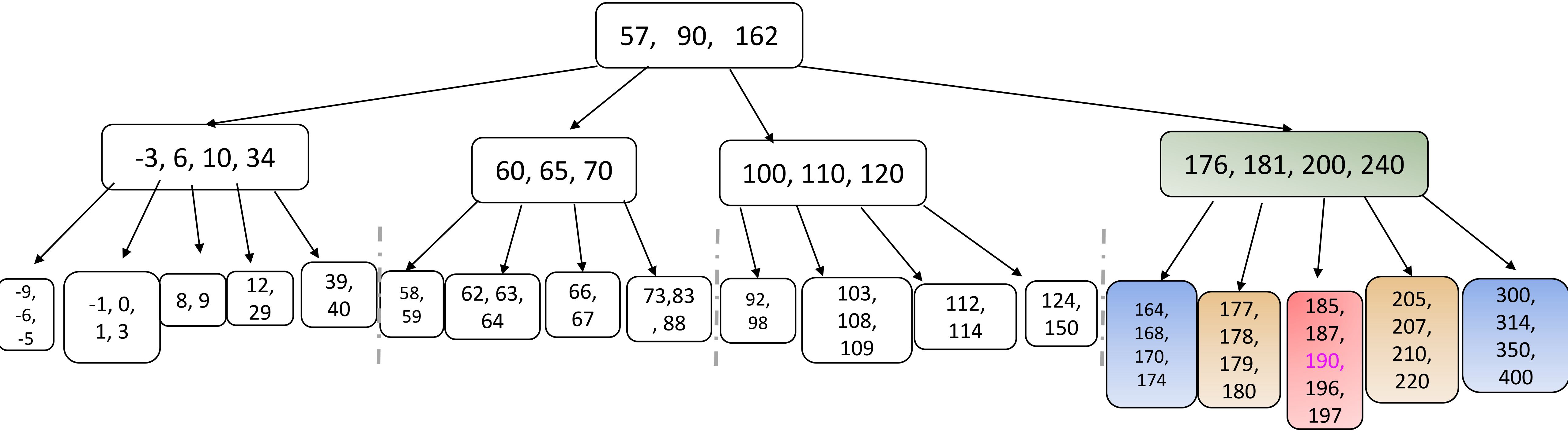


Example #6: Splitting, parent overflows and rotates



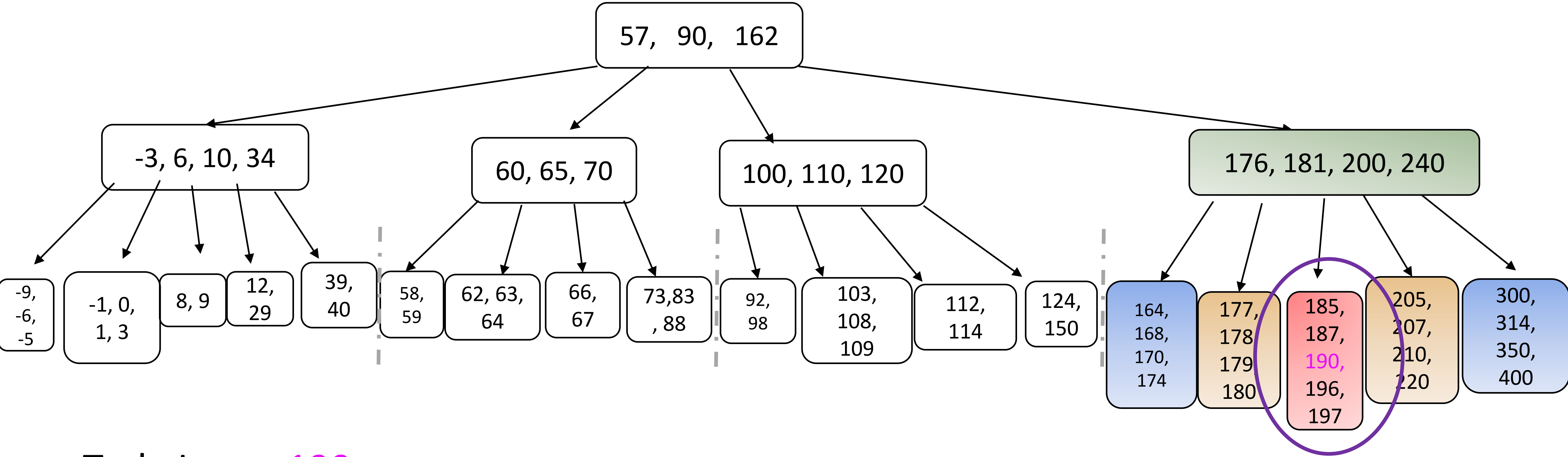
- Task: Insert 190
- Overflow ☹
- 1st Siblings, 2nd siblings and parent all full! ☹

Example #6: Splitting, parent overflows and rotates



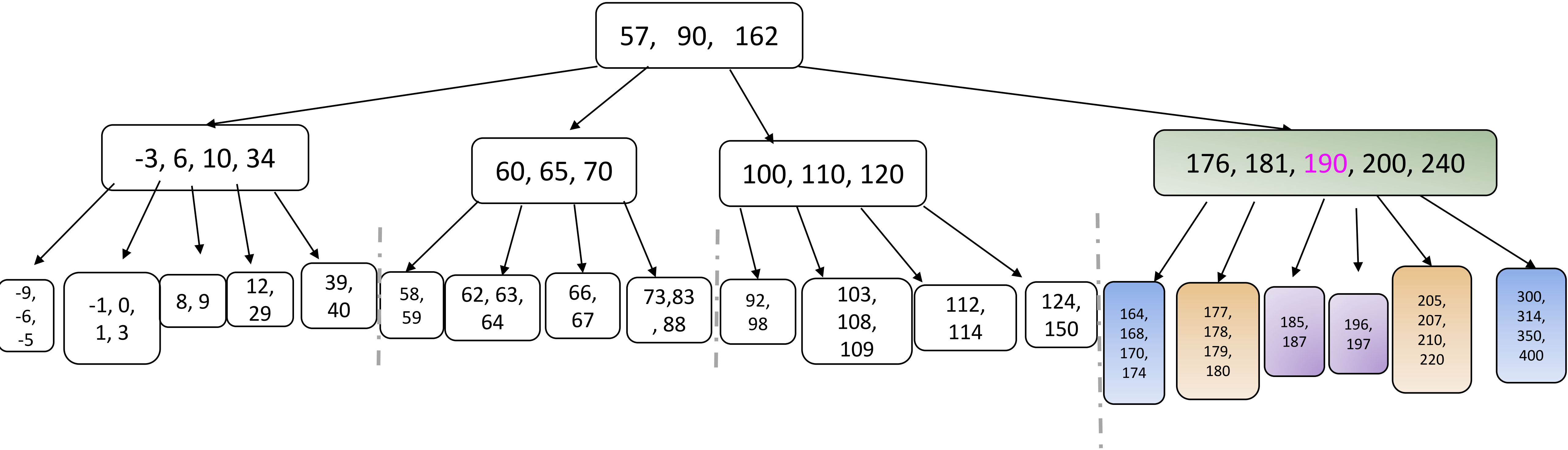
- Task: Insert 190
- Overflow ☹
- 1st Siblings, 2nd siblings and parent all full! ☹
- Solution: Split **node**, elevating 190 to **parent** and then rotate **176** to the left!

Example #6: Splitting, parent overflows and rotates



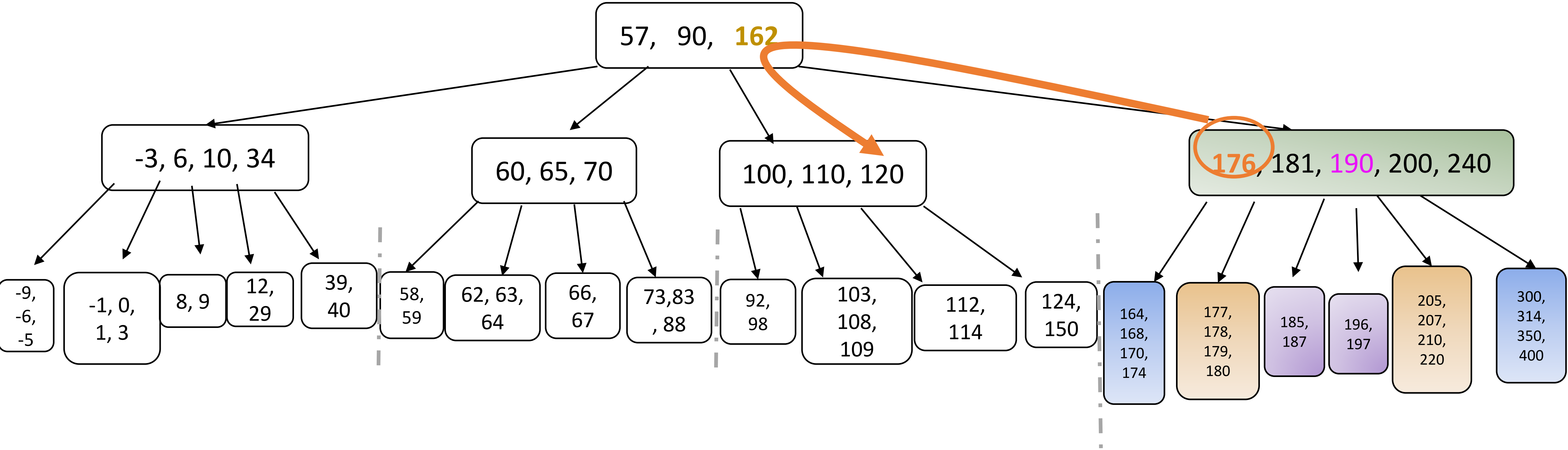
- Task: Insert 190
- Overflow ☹
- 1st Siblings, 2nd siblings and parent all full! ☹
- Solution: Split **node**, elevating 190 to **parent** and then rotate **176** to the left!

Example #6: Splitting, parent overflows and rotates



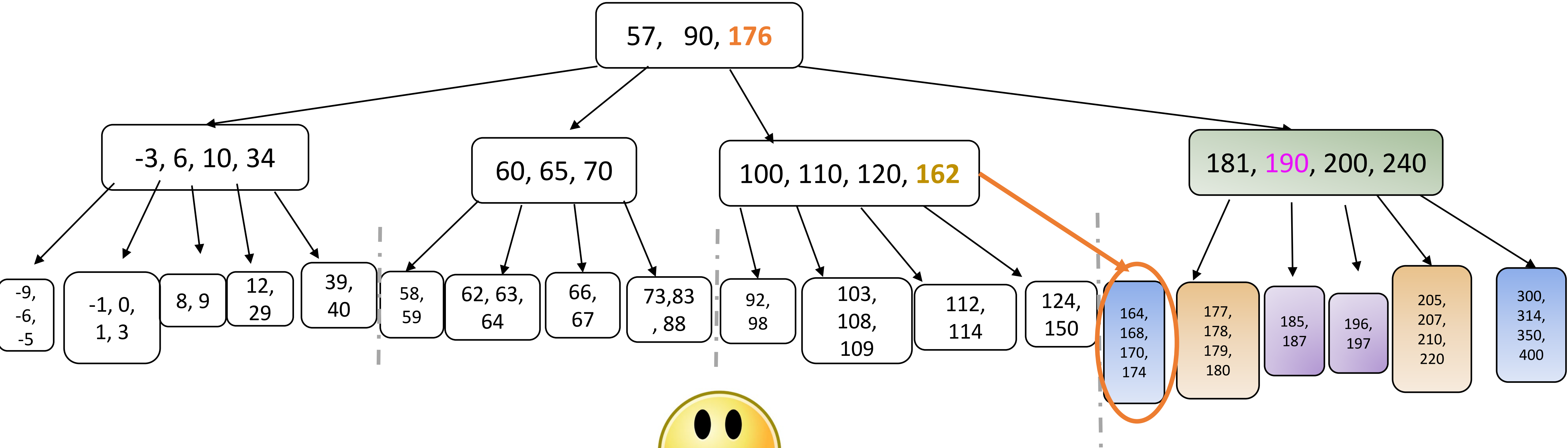
- Task: Insert 190
- Overflow ☹
- 1st Siblings, 2nd siblings and parent all full! ☹
- Solution: Split **node**, elevating 190 to **parent** and then rotate **176** to the left!

Example #6: Splitting, parent overflows and rotates



- Task: Insert 190
- Overflow ☹
- 1st Siblings, 2nd siblings and parent all full! ☹
- Solution: Split **node**, elevating 190 to **parent** and then rotate **176** to the left!

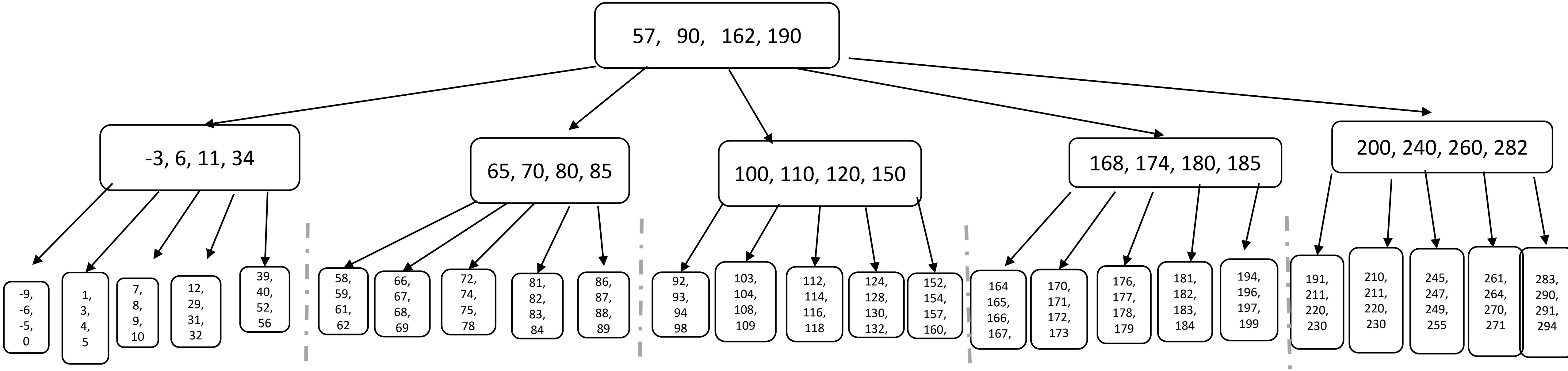
Example #6: Splitting, parent overflows and rotates



- Task: Insert 190
- Overflow ☹
- 1st Siblings, 2nd siblings and parent all full! ☹
- Solution: Split **node**, elevating 190 to **parent** and then rotate **176** to the left!

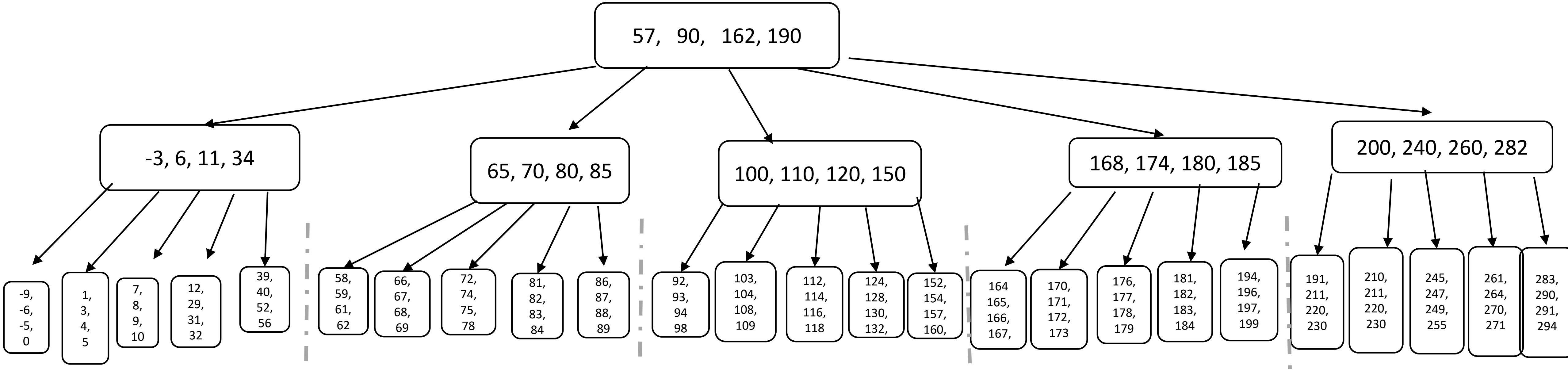
Don't forget to re-distribute subtrees when rotating keys above the leaf level!

Example #7: Worst-case, splitting the root



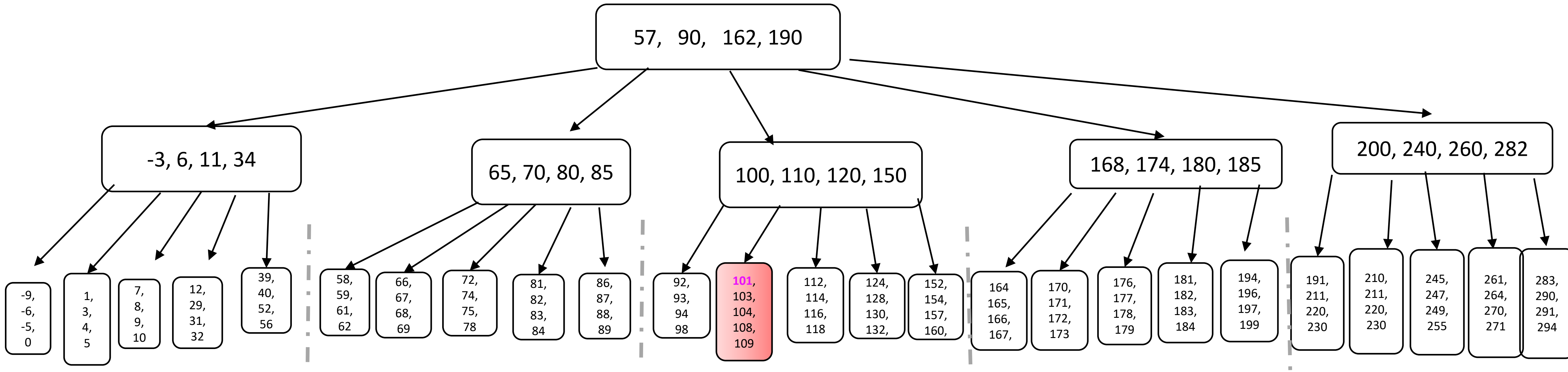
- Added a bunch of keys to make the tree filled to the brim!
- $4 + 4^2 + 4^3 = 20 + 64 = 84$ keys with $h = 2$ and $p = 5$.

Example #7: Worst-case, splitting the root



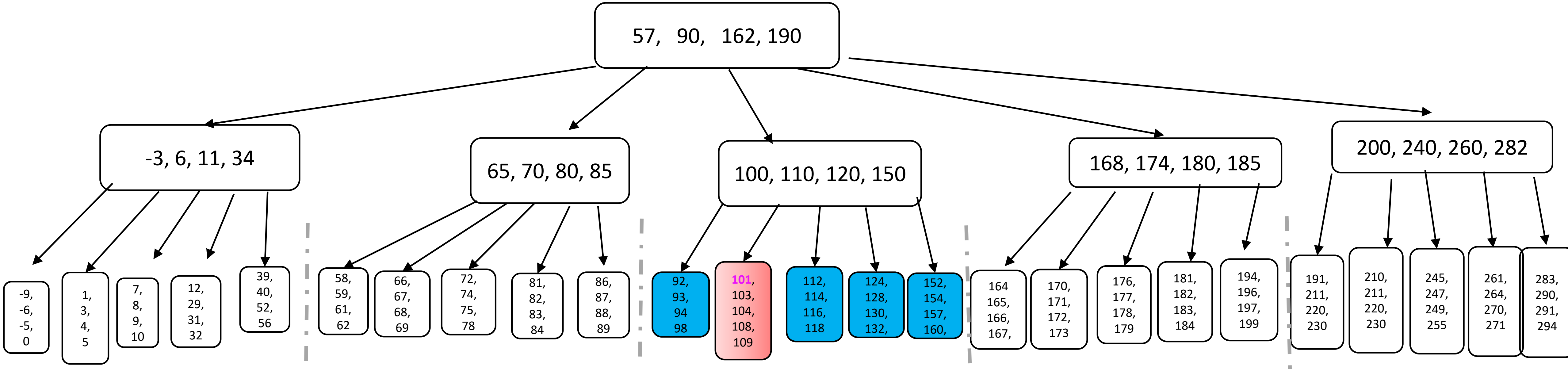
- Task: Insert 101

Example #7: Worst-case, splitting the root



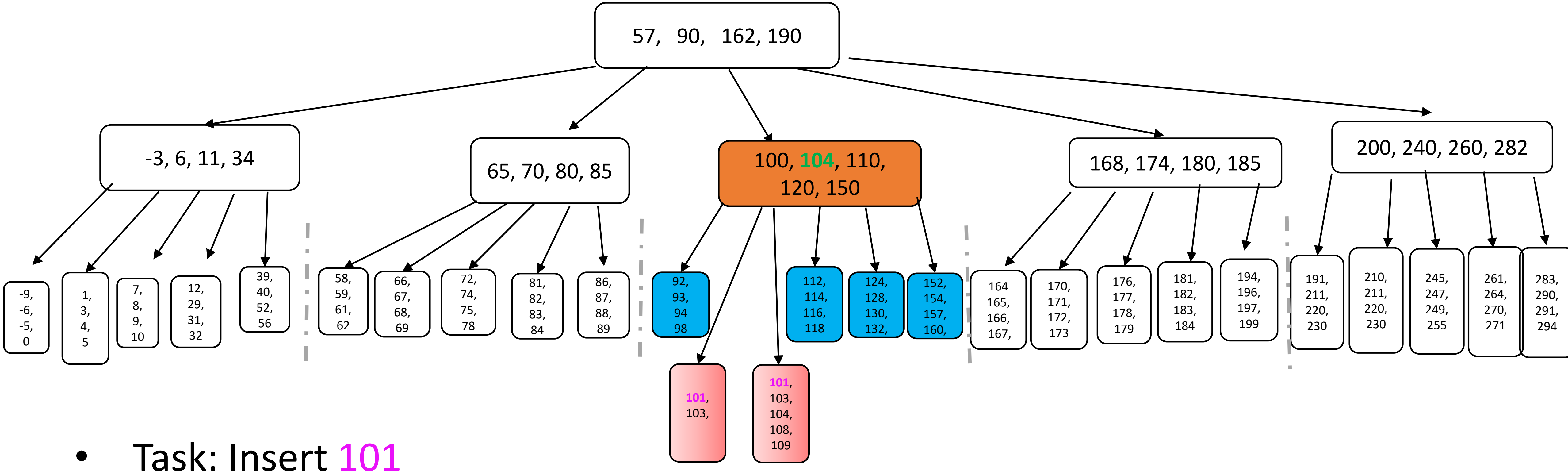
- Task: Insert 101
- Overflow ☹

Example #7: Worst-case, splitting the root



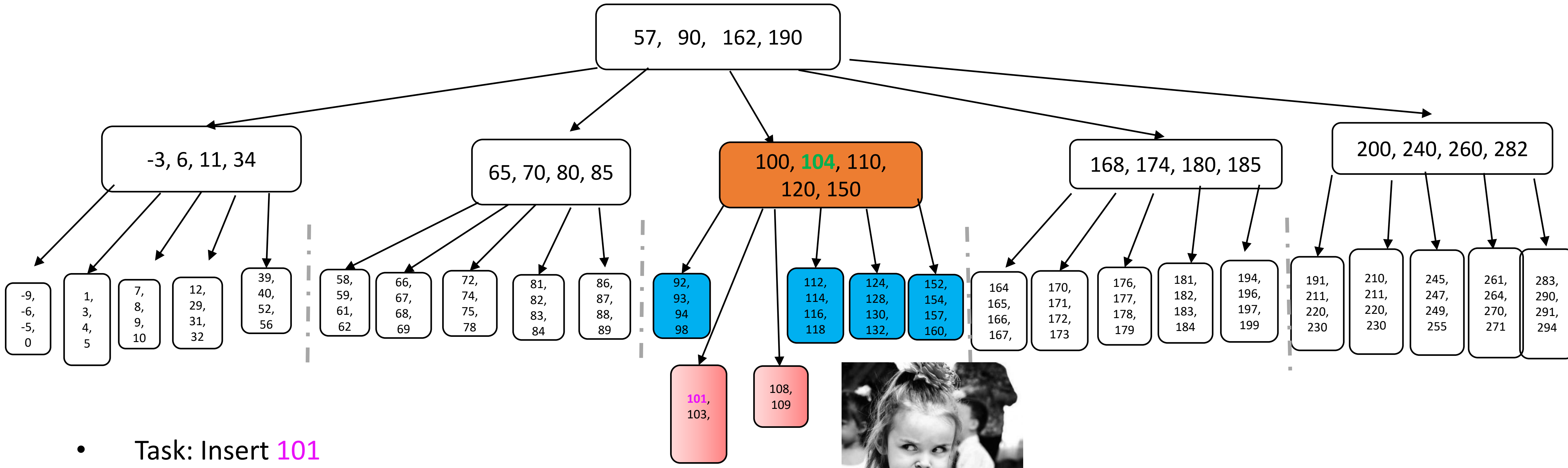
- Task: Insert 101
- Overflow ☹
- No possible key rotation to siblings! ☹

Example #7: Worst-case, splitting the root



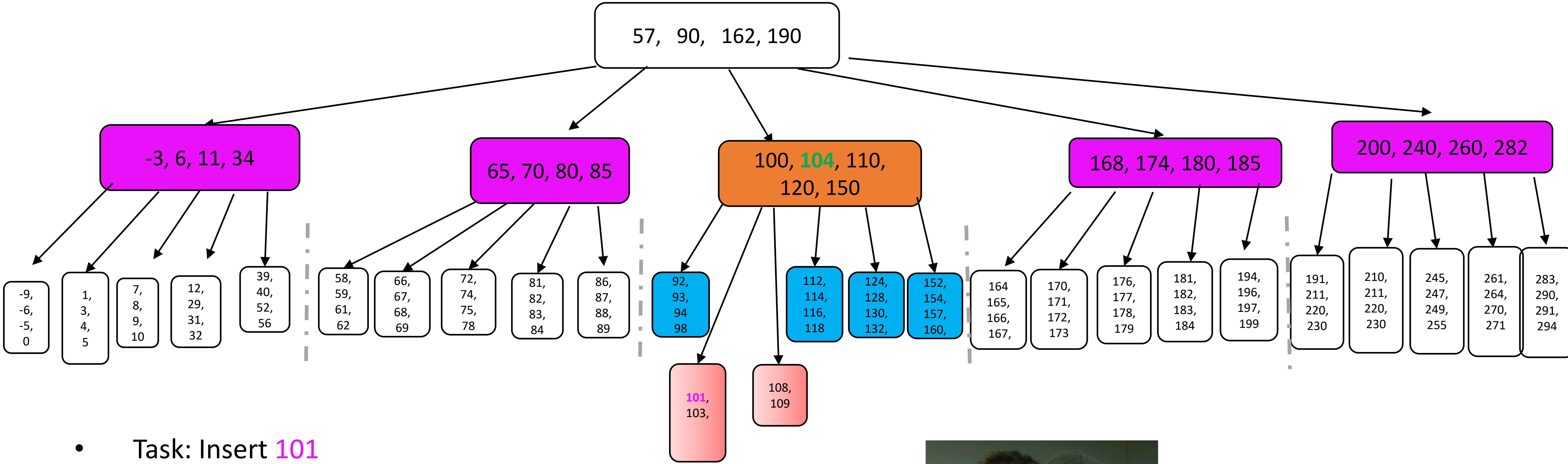
- Task: Insert 101
- Overflow ☹
- No possible key rotation anywhere in the tree! ☹
- **Split** and elevate 104 (middle key) to parent…

Example #7: Worst-case, splitting the root



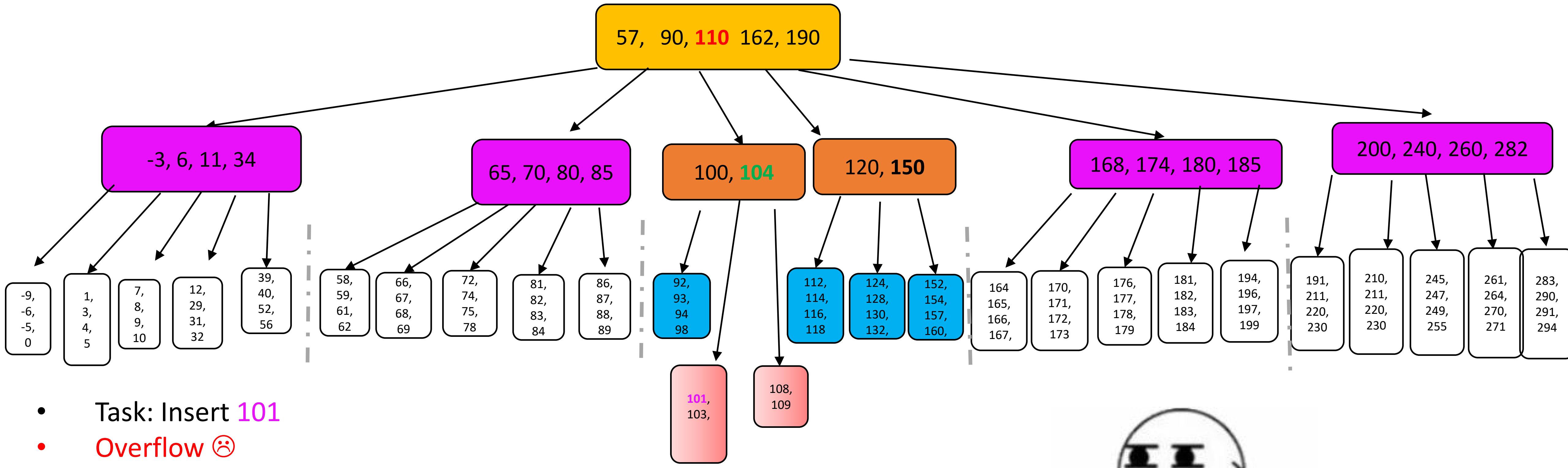
- Task: Insert 101
- Overflow ☹
- No possible key rotation anywhere in the tree! ☹
- **Split** and elevate 104 (middle key) to parent…
- **Which also overflows!**

Example #7: Worst-case, splitting the root



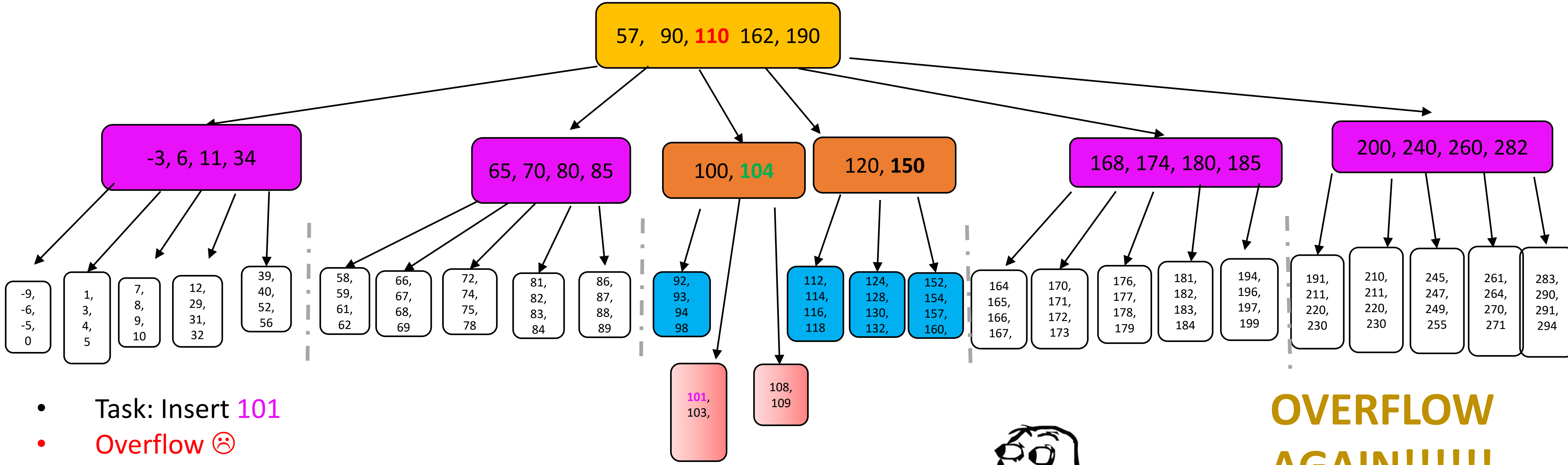
- Task: Insert 101
- Overflow ☹
- No possible key rotation anywhere in the tree! ☹
- **Split** and elevate 104 (middle key) to parent…
- **Which also overflows!**
- **AND ITS OWN SIBLINGS CAN'T HELP WITH ROTATIONS!**

Example #7: Worst-case, splitting the root

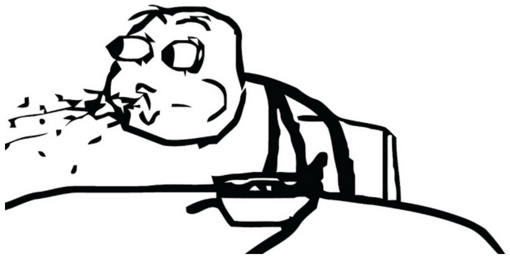


- Task: Insert 101
- Overflow ☹
- No possible key rotation anywhere in the tree! ☹
- **Split** and elevate 104 (middle key) to parent…
- **Which also overflows!**
- **AND ITS OWN SIBLINGS CAN'T HELP WITH ROTATIONS!**
- **Split current node** and elevate **110** (middle key) to the **parent (the root)**.

Example #7: Worst-case, splitting the root

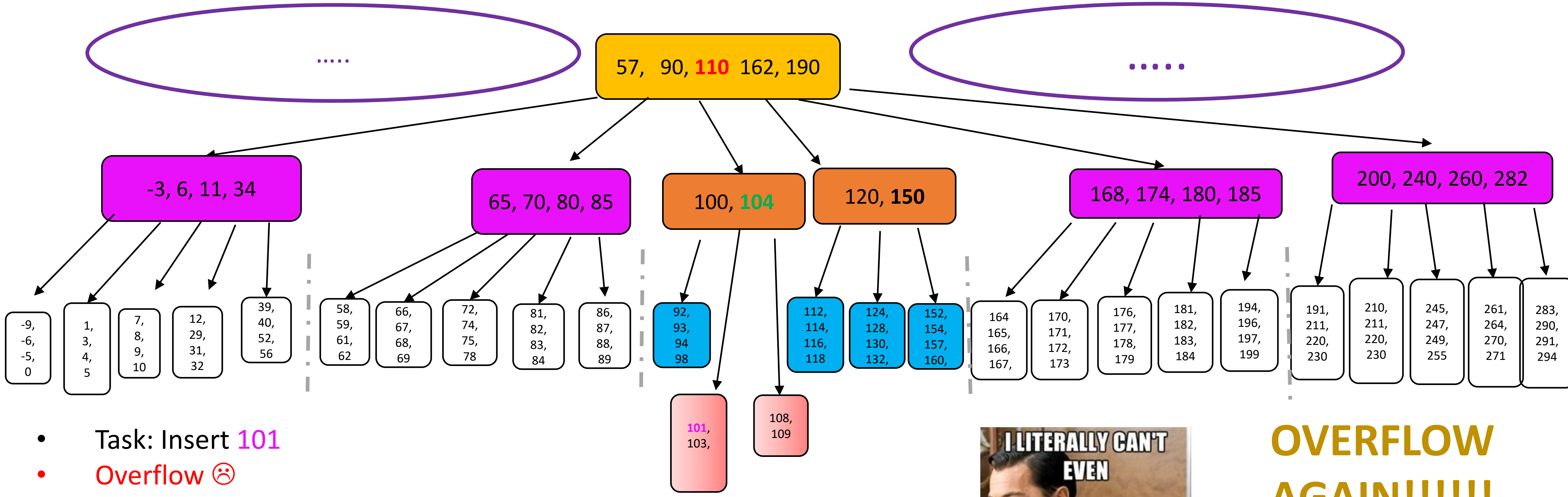


- Task: Insert 101
- Overflow ☹
- No possible key rotation anywhere in the tree! ☹
- **Split** and elevate 104 (middle key) to parent…
- **Which also overflows!**
- **AND ITS OWN SIBLINGS CAN'T HELP WITH ROTATIONS!**
- **Split current node** and elevate **110** (middle key) to the **parent (the root)**.

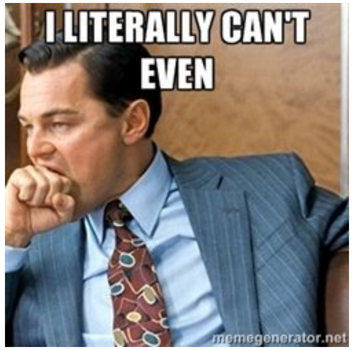


OVERFLOW AGAIN!!!!!!

Example #7: Worst-case, splitting the root



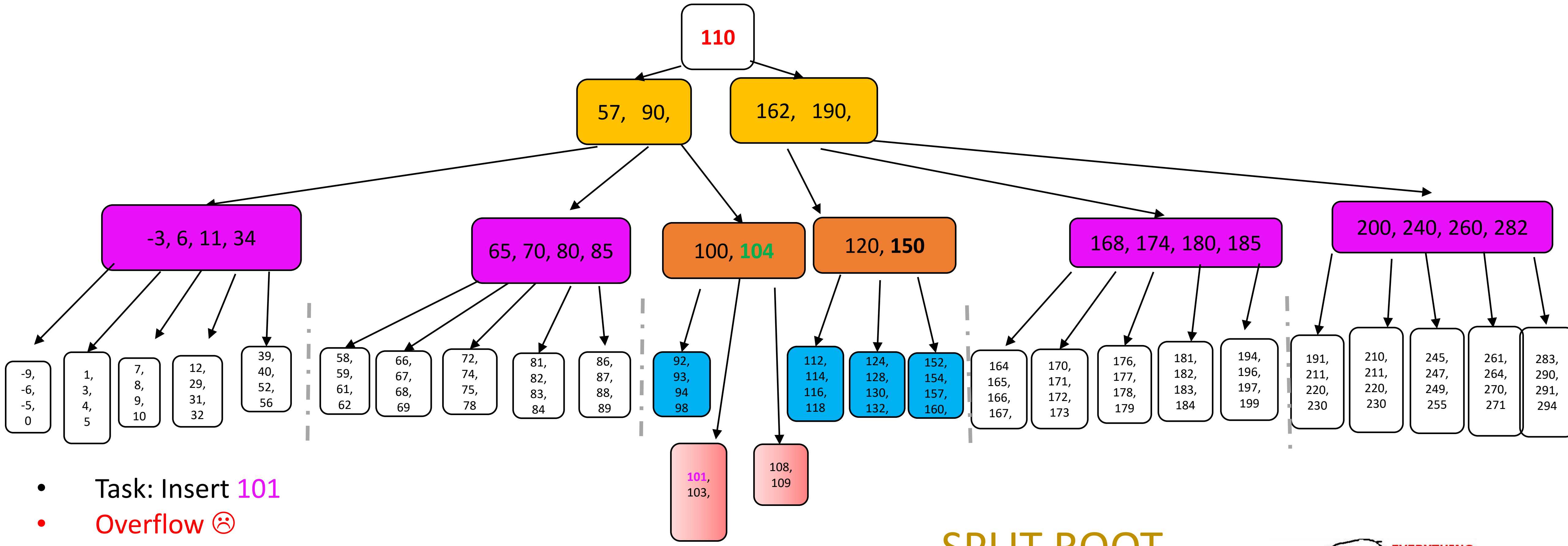
- Task: Insert 101
- Overflow ☹
- No possible key rotation anywhere in the tree! ☹
- **Split** and elevate 104 (middle key) to parent…
- **Which also overflows!**
- **AND ITS OWN SIBLINGS CAN'T HELP WITH ROTATIONS!**
- **Split current node** and elevate **110** (middle key) to the **parent (the root)**.



OVERFLOW AGAIN!!!!!!

NO SIBLINGS TO ROTATE TO!

Example #7: Worst-case, splitting the root



- Task: Insert 101
- Overflow ☹
- No possible key rotation anywhere in the tree! ☹
- **Split** and elevate 104 (middle key) to parent…
- **Which also overflows!**
- **AND ITS OWN SIBLINGS CAN'T HELP WITH ROTATIONS!**
- **Split current node** and elevate **110** (middle key) to the **parent (the root)**.

SPLIT ROOT,
ELEVATING **110**,
AND DONE.

Dealing with underflows

- Since deleting any key will always boil down to deleting some key from the leaf level, underflows can only occur at the leaf level

Dealing with underflows

- Since deleting any key will always boil down to deleting some key from the leaf level, underflows can only occur at the leaf level
- We will once again attempt key rotations first.

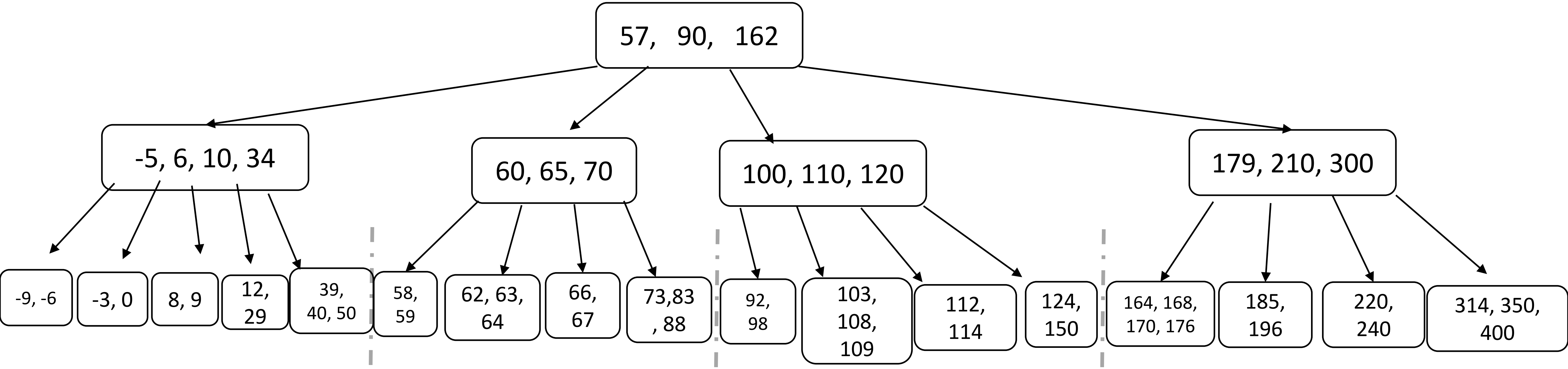
Dealing with underflows

- Since deleting any key will always boil down to deleting some key from the leaf level, underflows can only occur at the leaf level
- We will once again attempt key rotations first.
- Idea: Look at your sibling nodes (≤ 2) and locate at least one that would not underflow if you were to take a key from it
  - If you find one such sibling, rotate the key "closest" to the current node from that sibling through the parent.
  - This is always either the smallest or the largest key of the sibling

Dealing with underflows

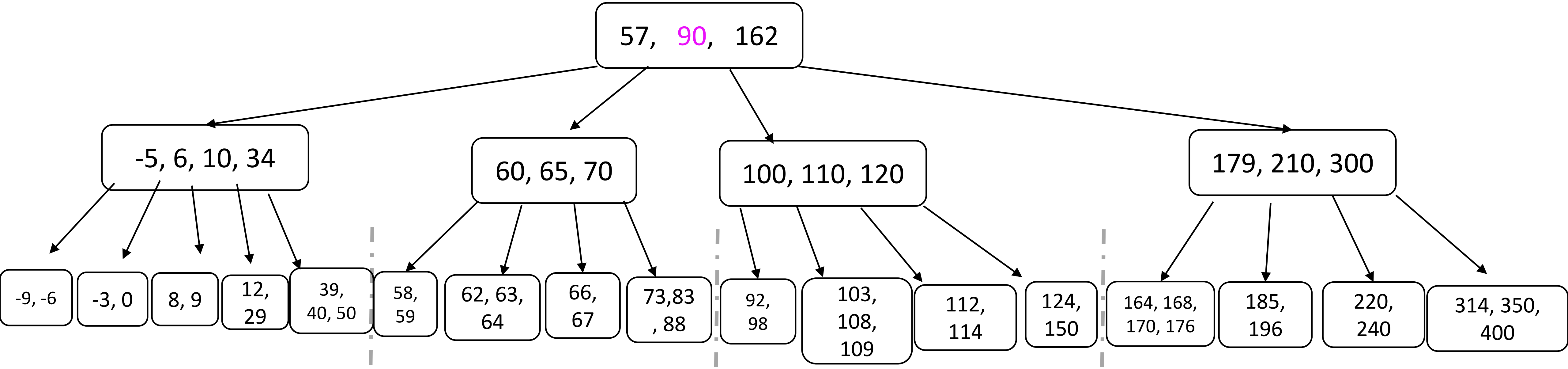
- Since deleting any key will always boil down to deleting some key from the leaf level, underflows can only occur at the leaf level
- We will once again attempt key rotations first.
- Idea: Look at your sibling nodes (≤ 2) and locate at least one that would not underflow if you were to take a key from it
  - If you find one such sibling, rotate the key “closest” to the current node from that sibling through the parent.
  - This is always either the smallest or the largest key of the sibling
- Familiar generalization to “further” siblings applies here too.
- If all fails, then merge current node with parent key
  - This may propagate an underflow in the parent, which should be dealt with recursively…

Example #0: Leaf deletion, no underflows



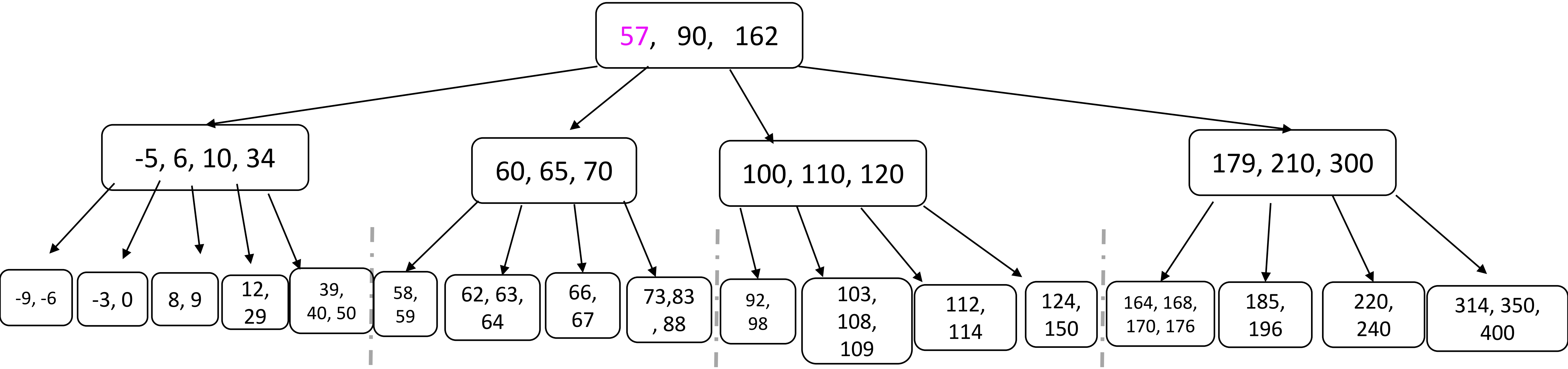
- Task: delete 62

Example #0: Leaf deletion, no underflows



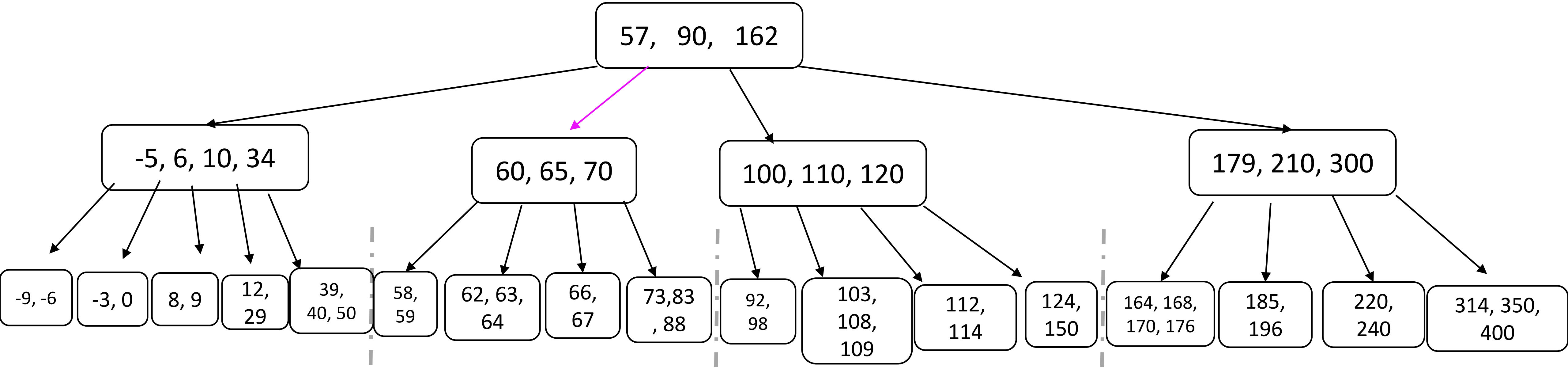
- Task: delete 62

Example #0: Leaf deletion, no underflows



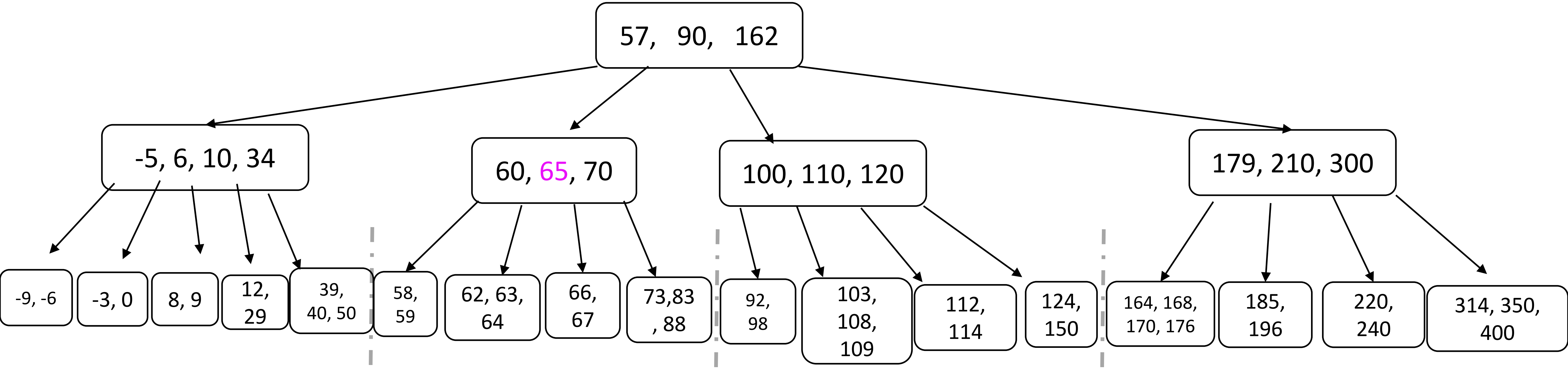
- Task: delete 62

Example #0: Leaf deletion, no underflows



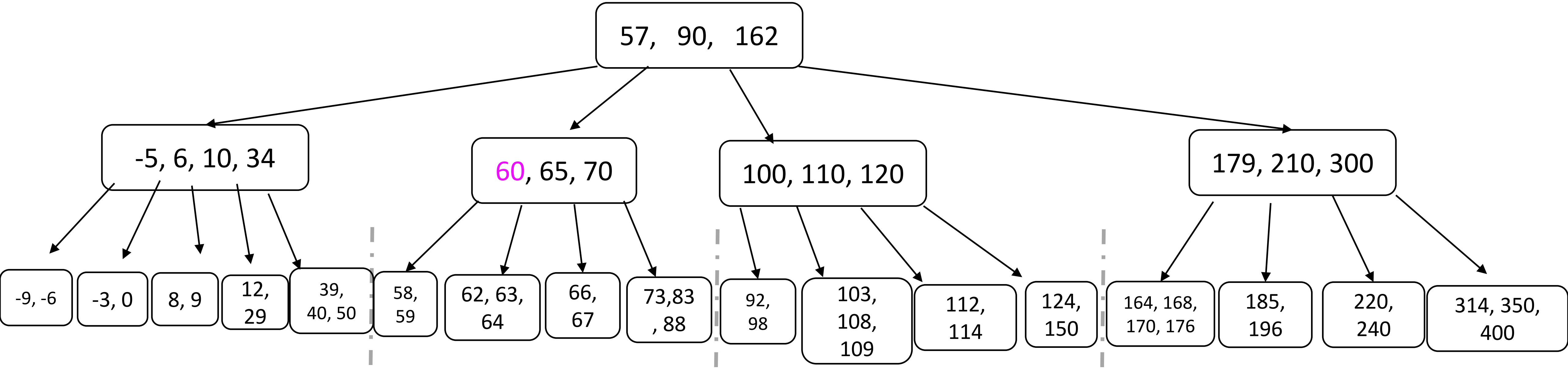
- Task: delete 62

Example #0: Leaf deletion, no underflows



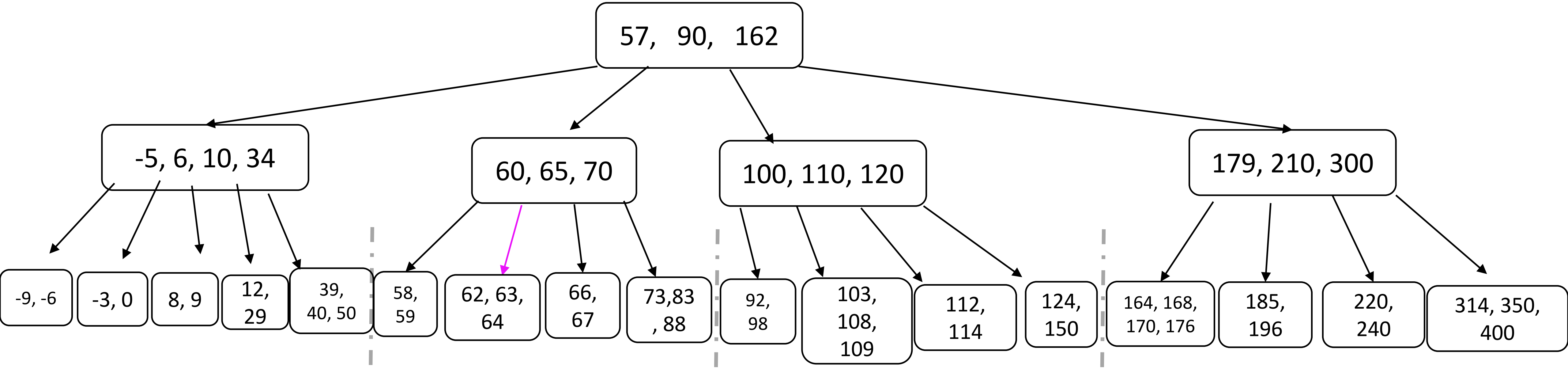
- Task: delete 62

Example #0: Leaf deletion, no underflows



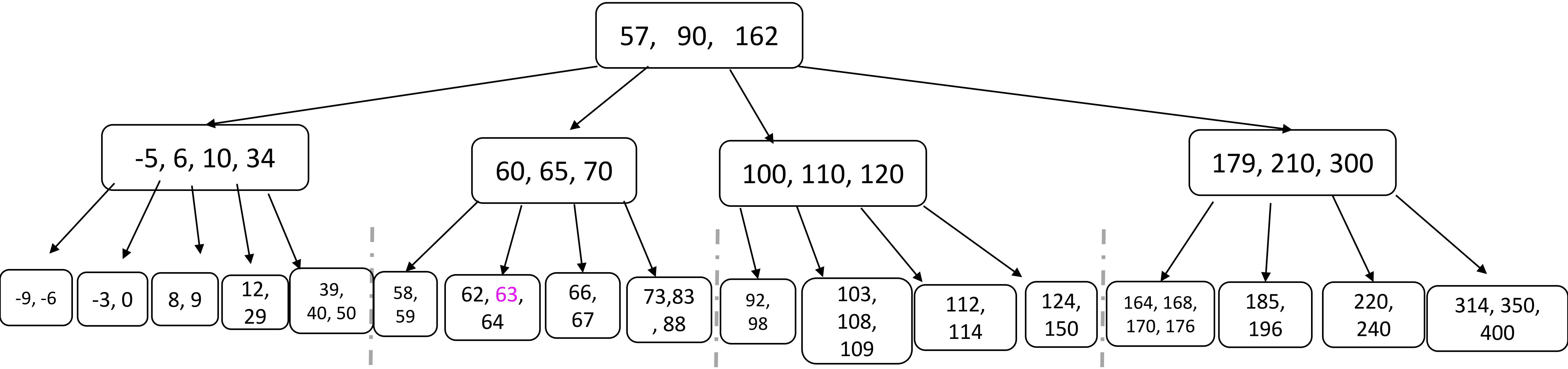
- Task: delete 62

Example #0: Leaf deletion, no underflows



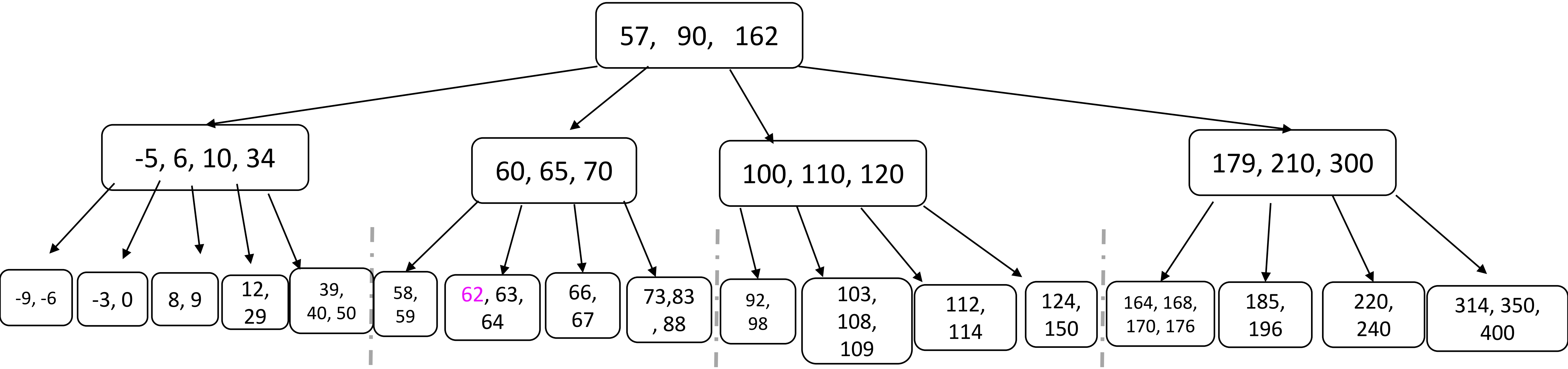
- Task: delete 62

Example #0: Leaf deletion, no underflows



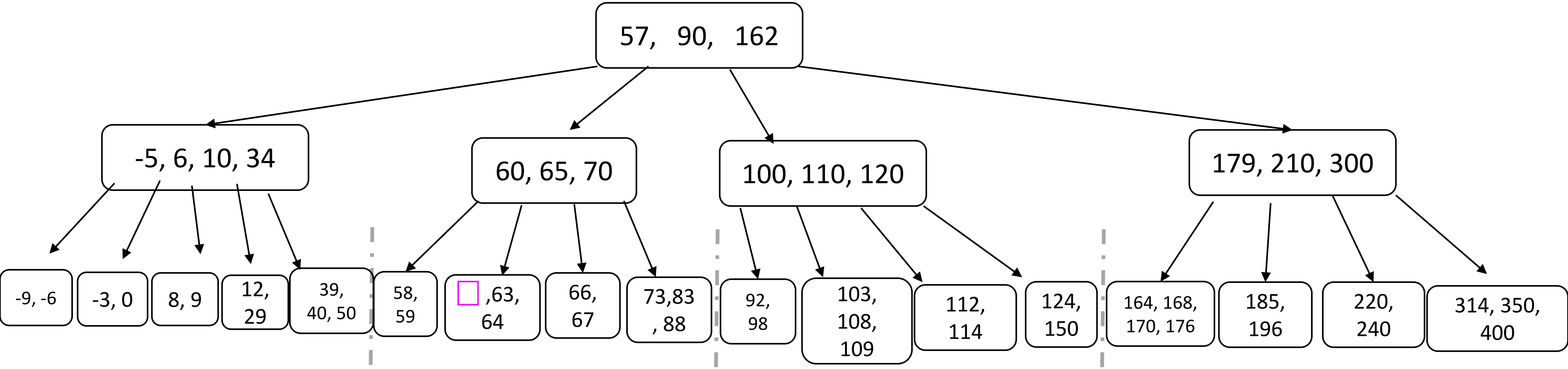
- Task: delete 62

Example #0: Leaf deletion, no underflows



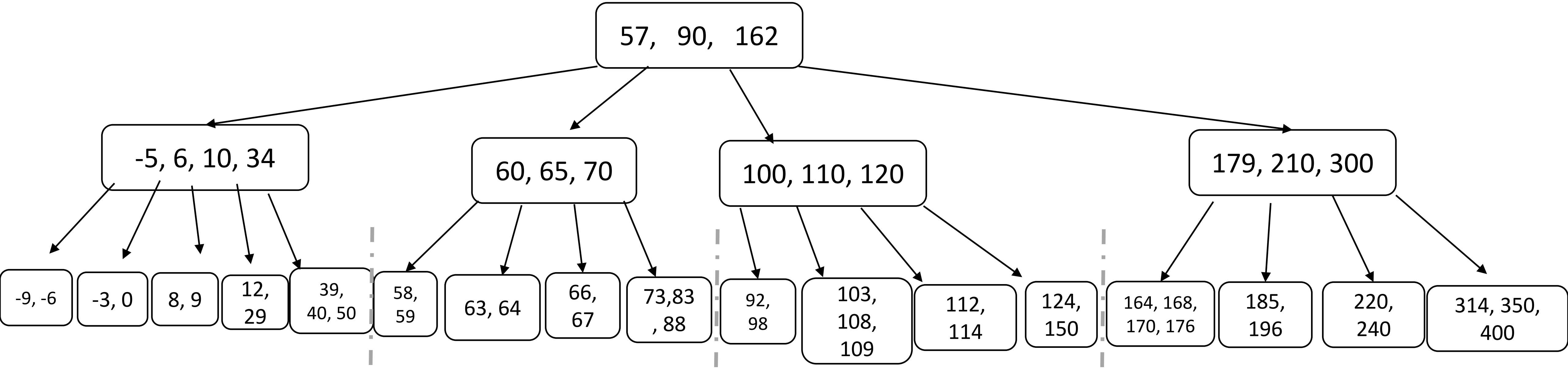
- Task: delete 62

Example #0: Leaf deletion, no underflows



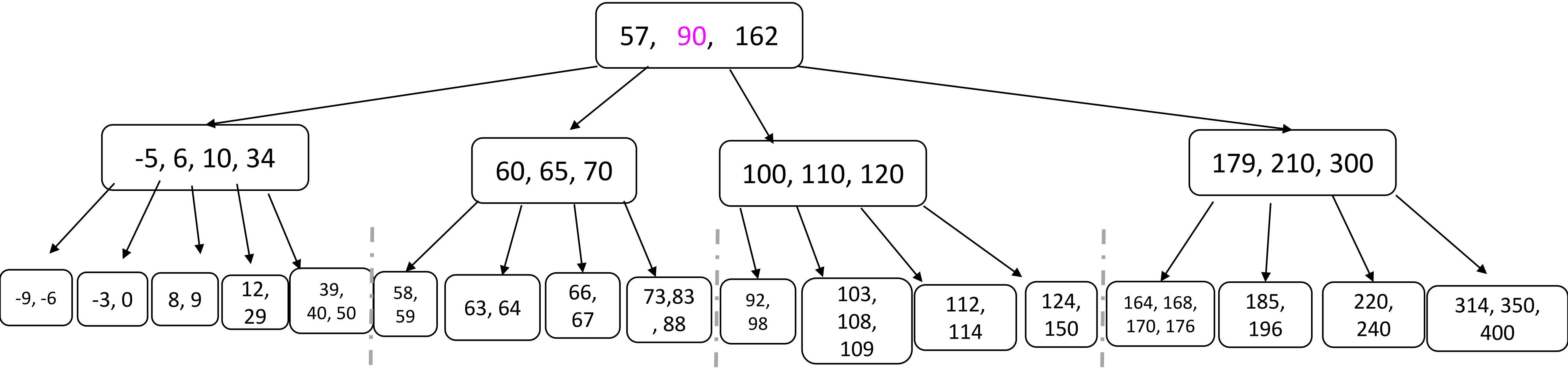
- Task: delete 62

Example #1: Inner node deletion, no underflows



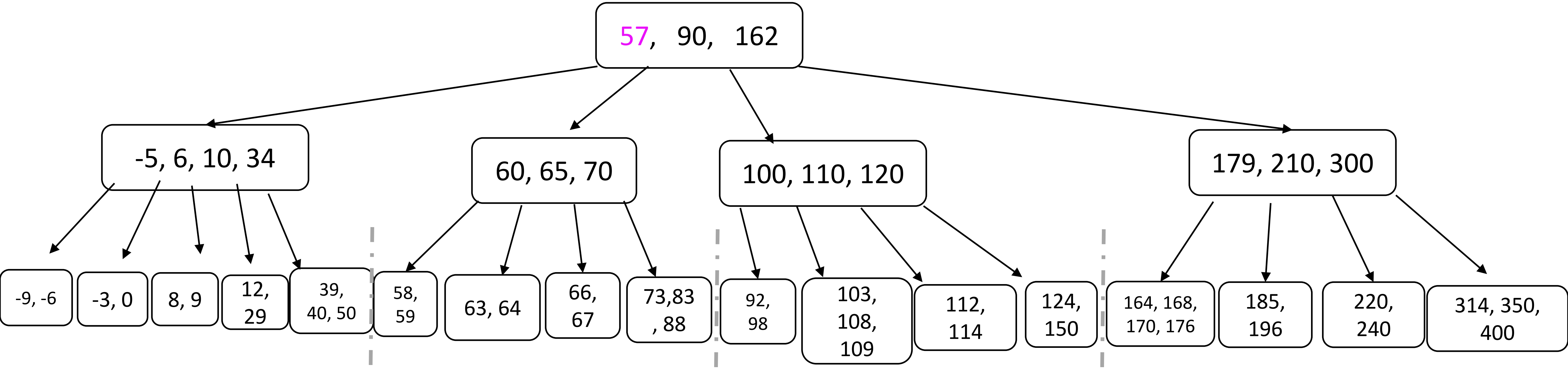
- Task: delete 34

Example #1: Inner node deletion, no underflows



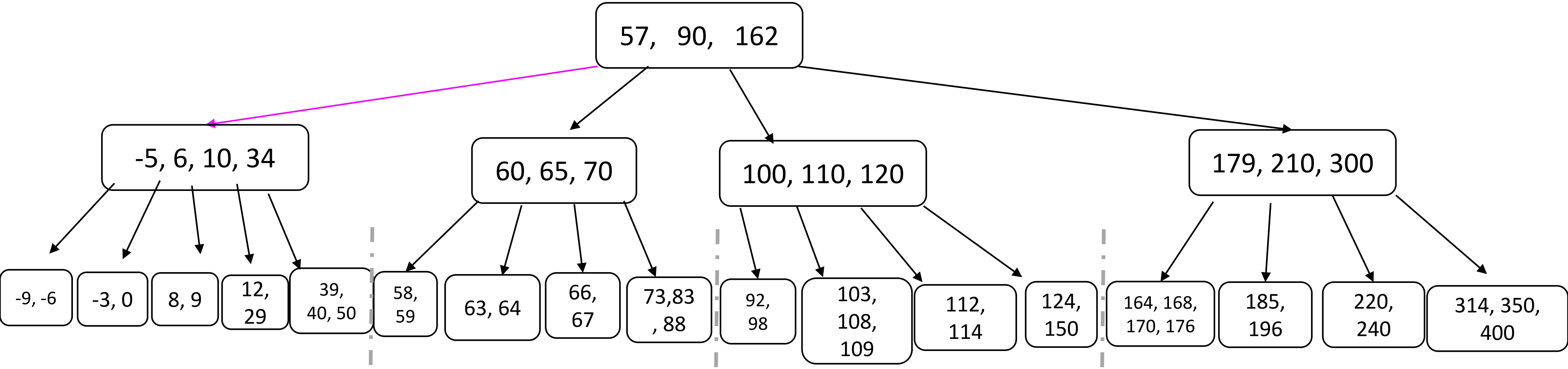
- Task: delete 34

Example #1: Inner node deletion, no underflows



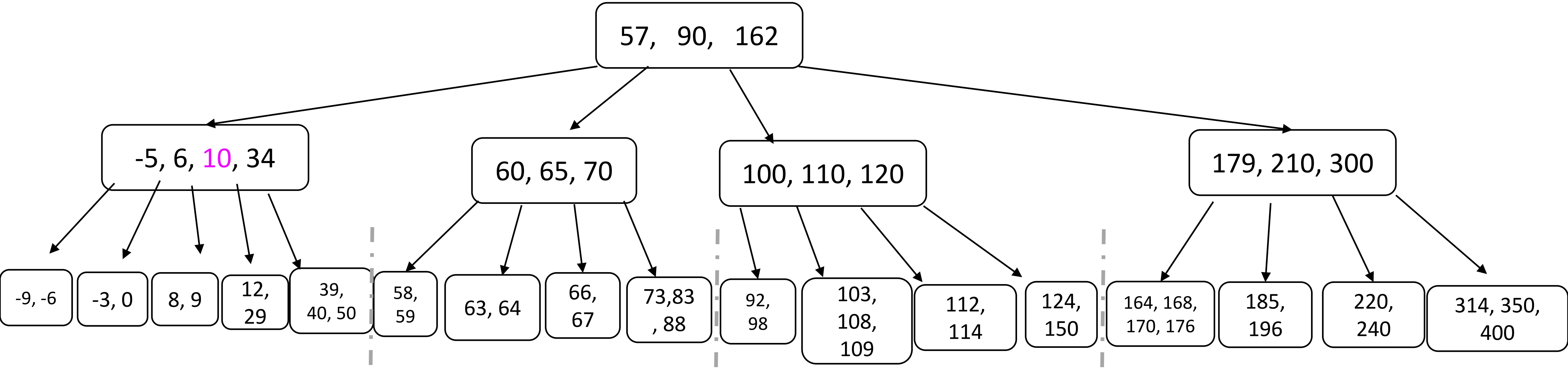
- Task: delete 34

Example #1: Inner node deletion, no underflows



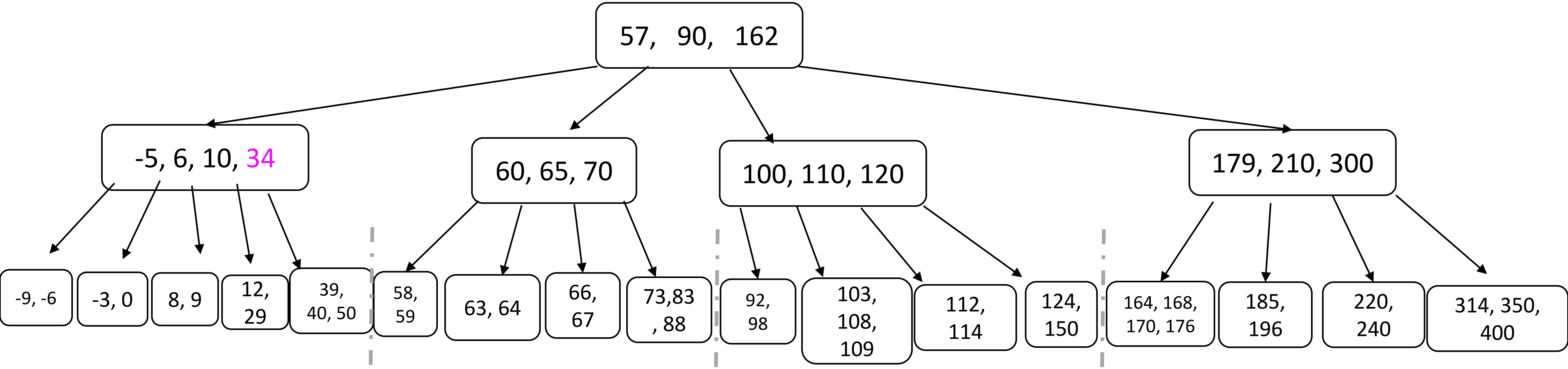
- Task: delete 34

Example #1: Inner node deletion, no underflows



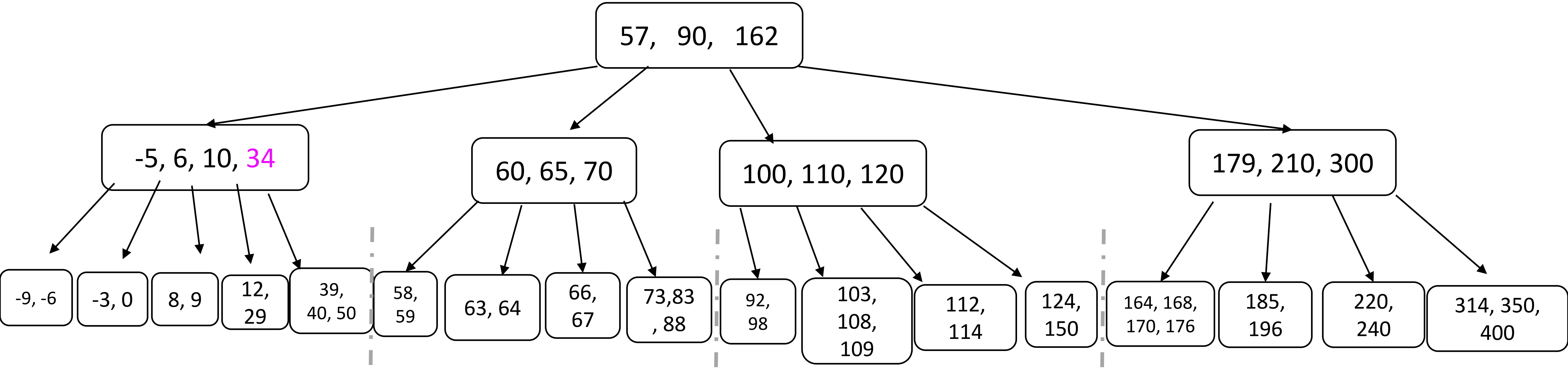
- Task: delete 34

Example #1: Inner node deletion, no underflows



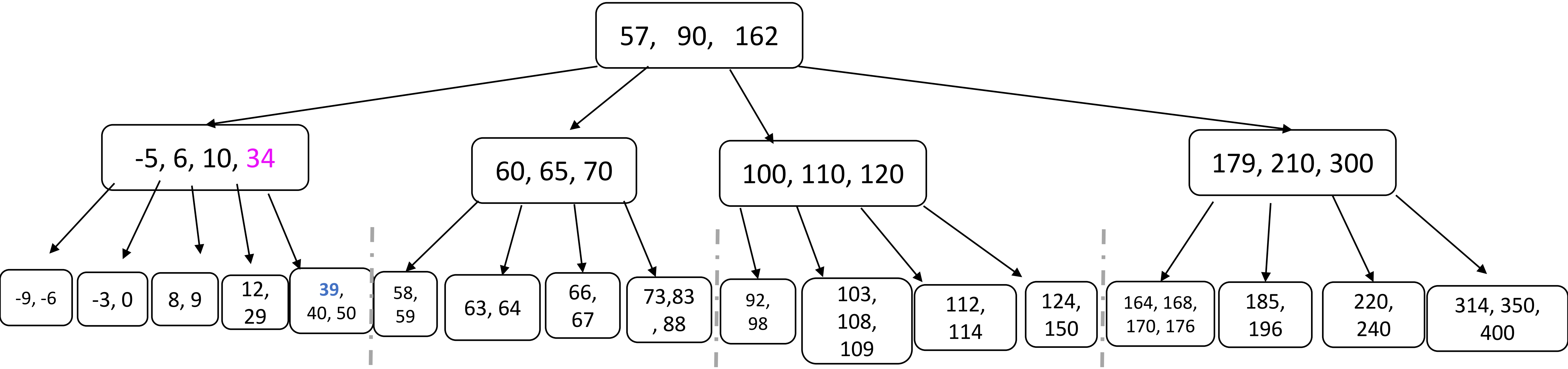
- Task: delete 34

Example #1: Inner node deletion, no underflows



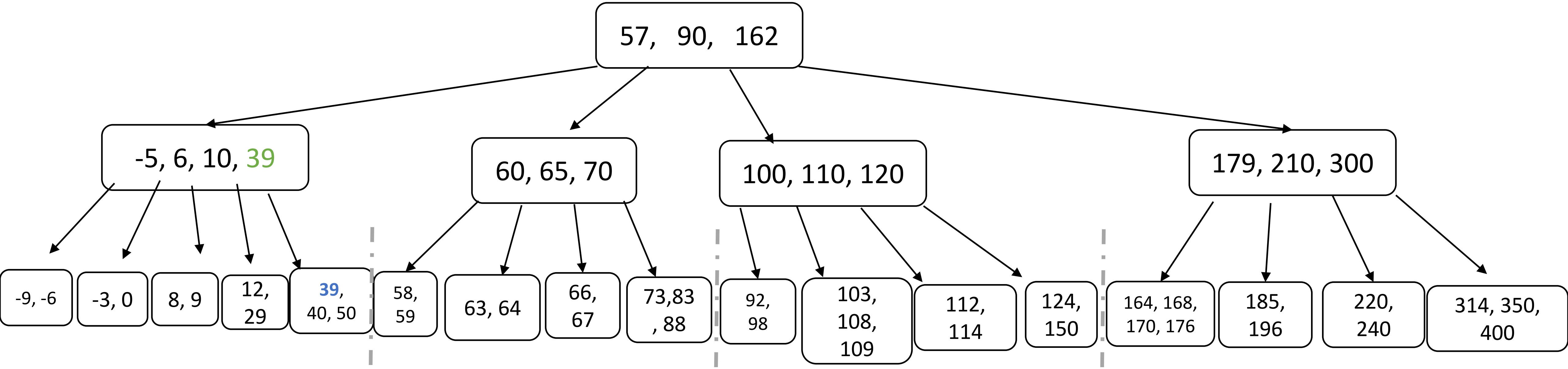
- Task: delete 34
  - Finding inorder successor…

Example #1: Inner node deletion, no underflows



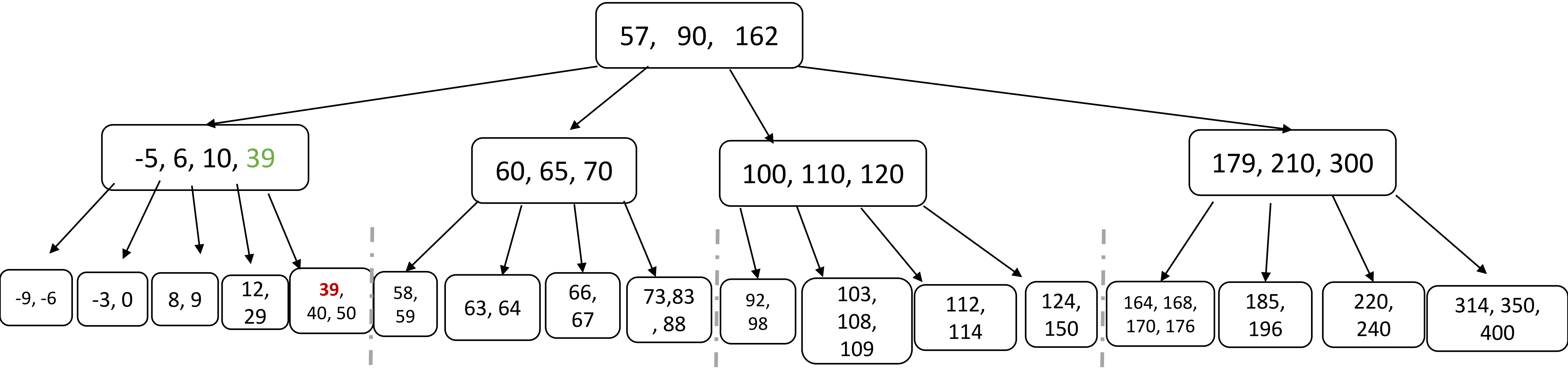
- Task: delete 34
  - Finding inorder successor…

Example #1: Inner node deletion, no underflows



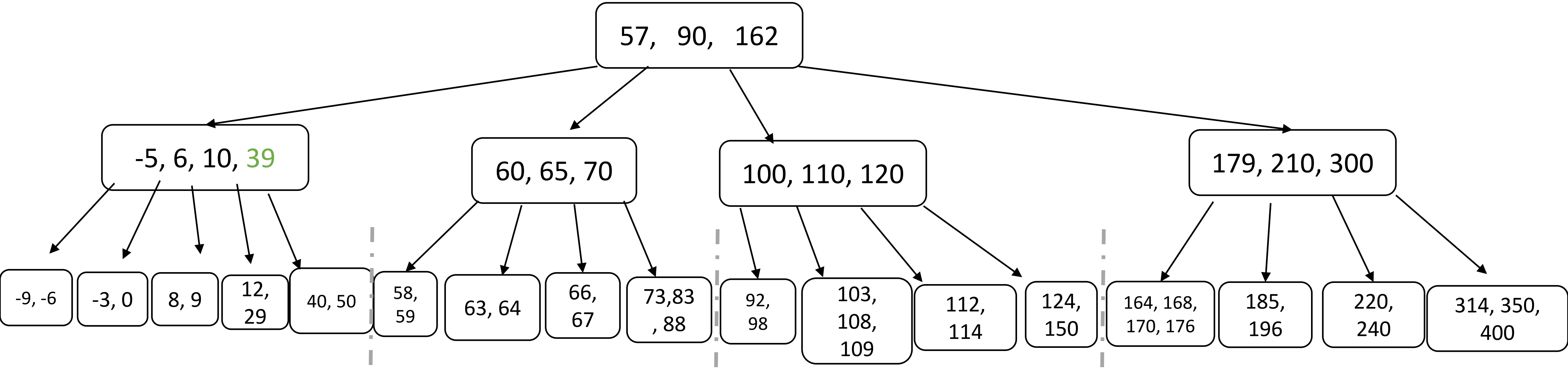
- Task: delete 34
  - Finding inorder successor…
  - Copy key

Example #1: Inner node deletion, no underflows

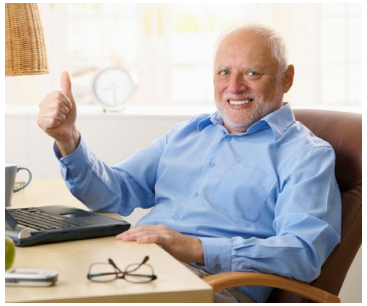


- Task: delete 34
  - Finding inorder successor…
  - Copy key
  - Recursively delete 39

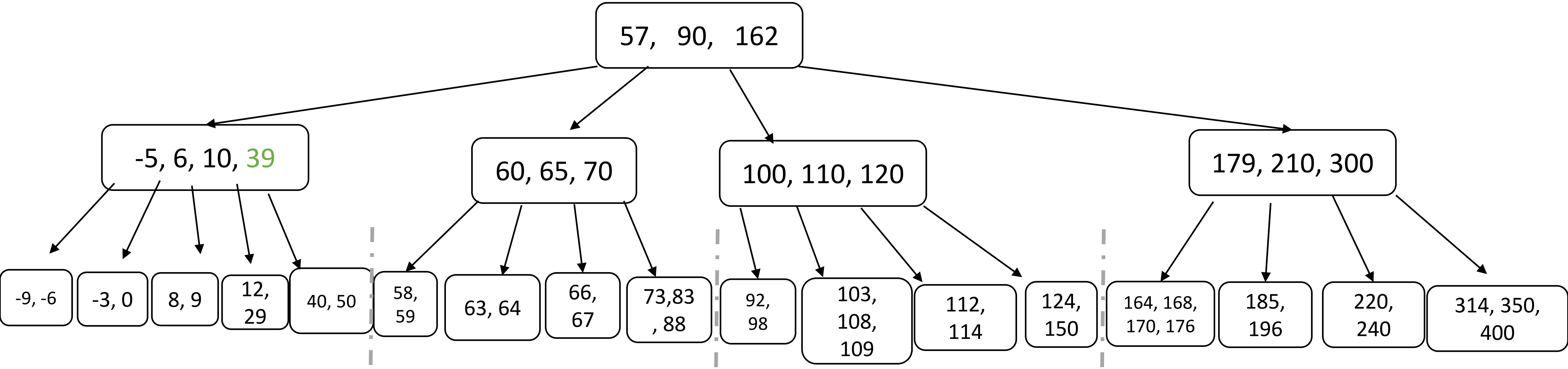
Example #1: Inner node deletion, no underflows



- Task: delete 34
  - Finding inorder successor…
  - Copy key
  - Recursively delete 39
    - No underflow, no problem!

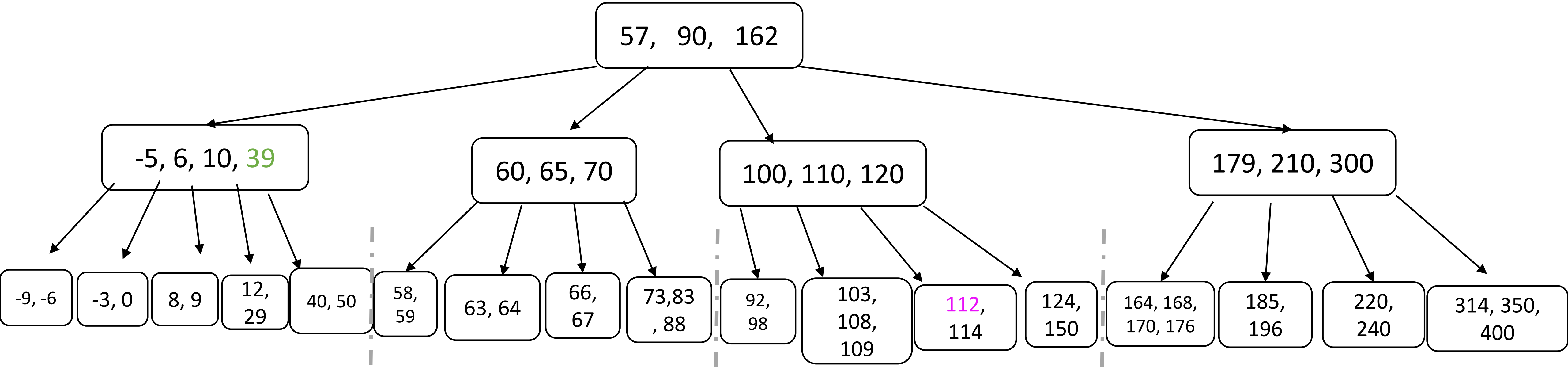


Example #2: Key rotation



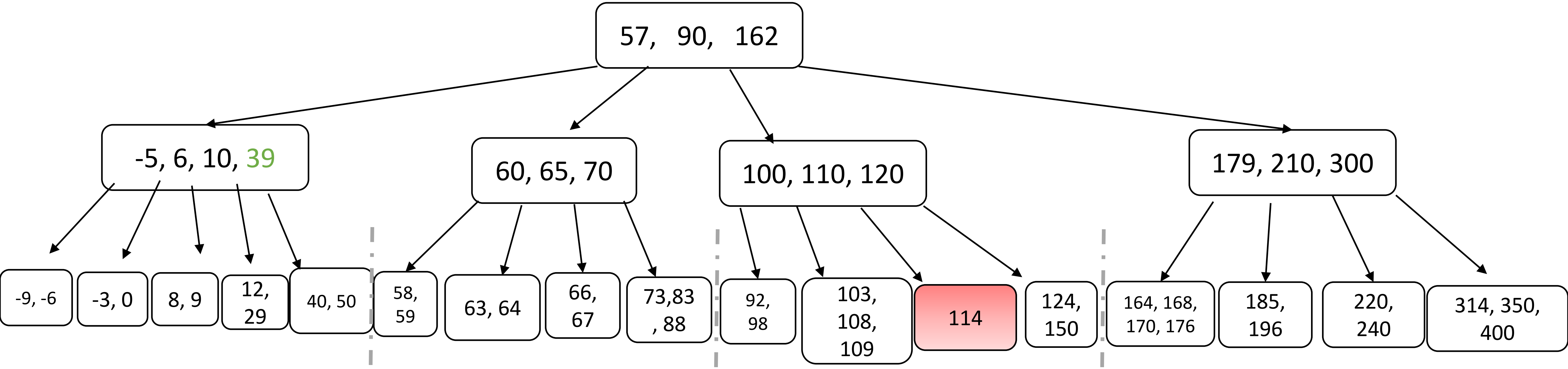
- Task: delete 112

Example #2: Key rotation



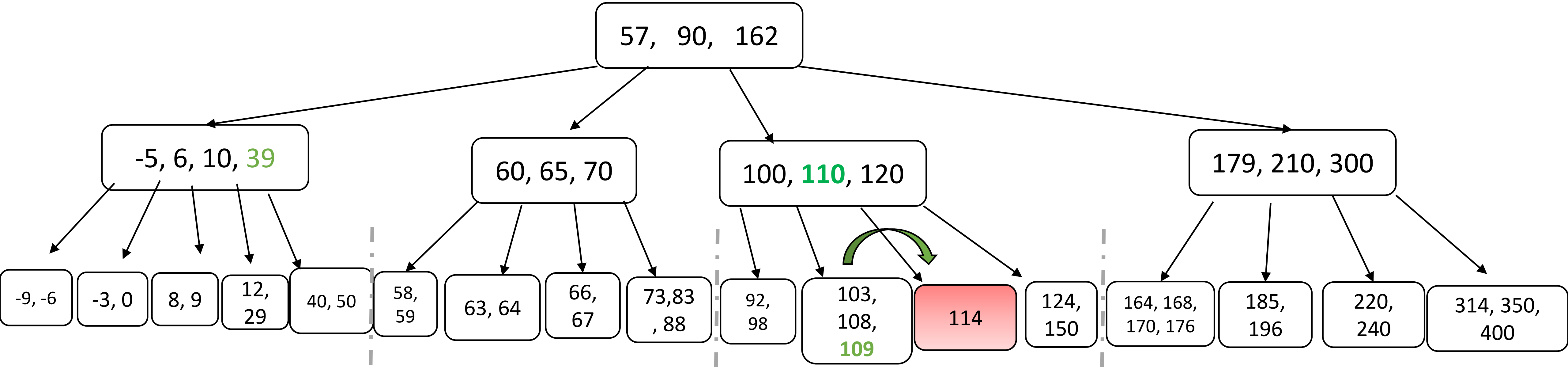
- Task: delete 112

Example #2: Key rotation



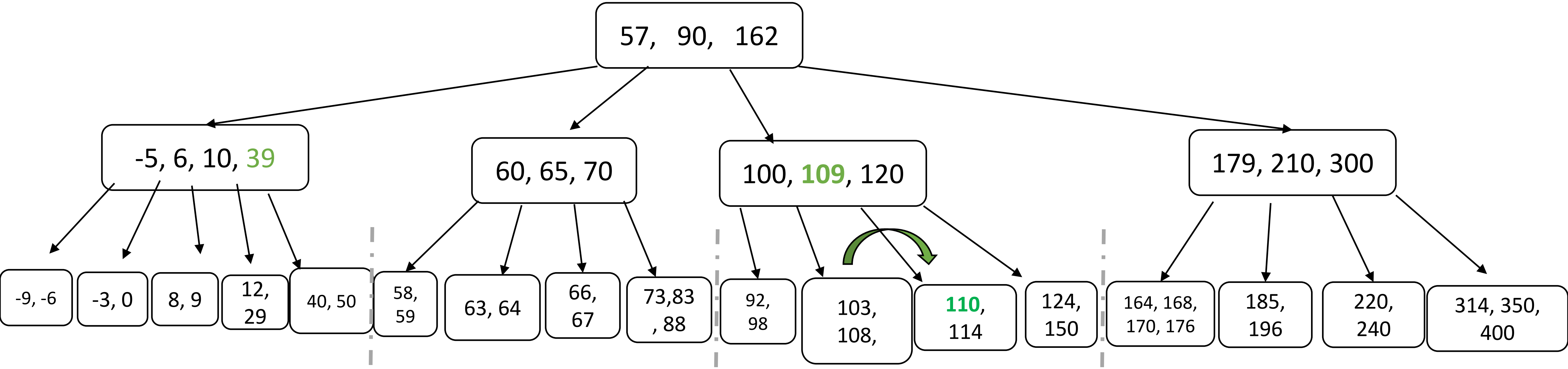
- Task: delete 112
- Underflow ☹

Example #2: Key rotation



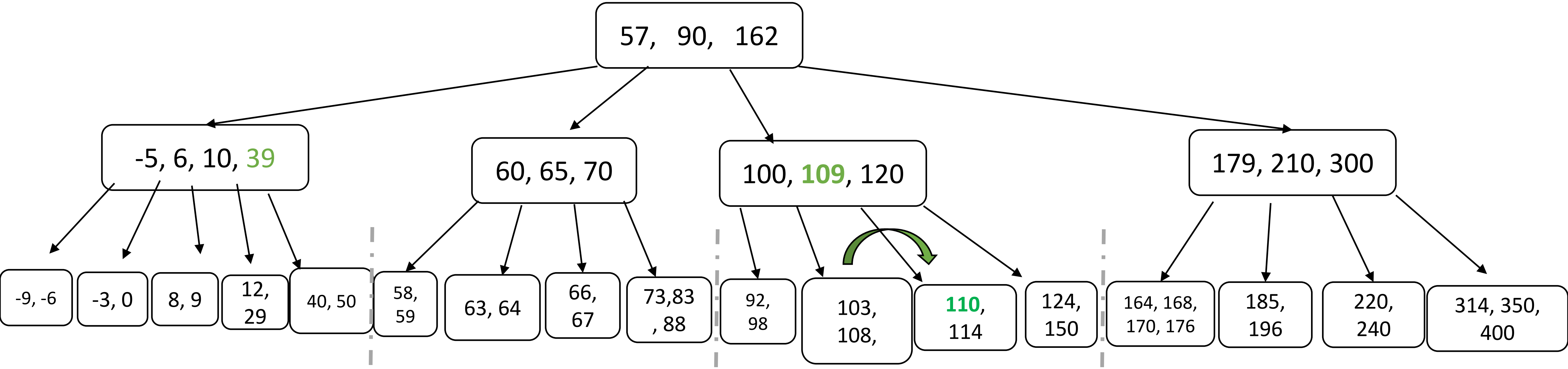
- Task: delete 112
- Underflow ☹
- Solvable via key rotation ☺

Example #2: Key rotation



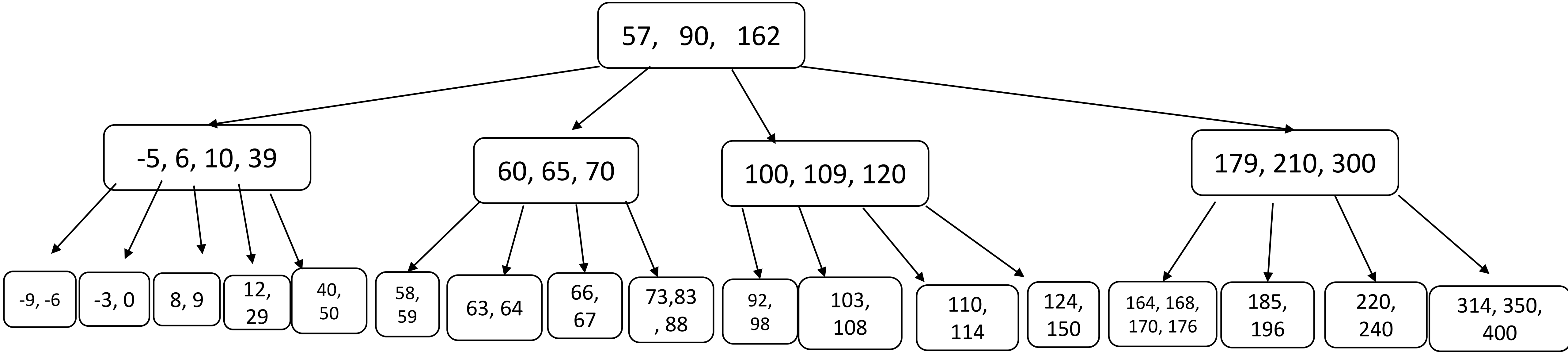
- Task: delete 112
- Underflow ☹
- Solvable via key rotation ☺

Example #2: Key rotation



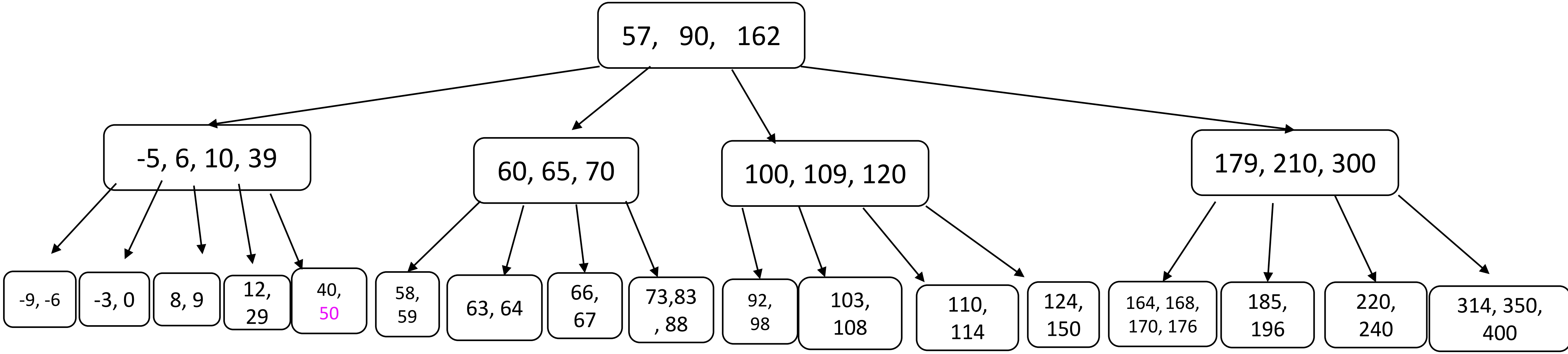
- Task: delete 112
- Underflow ☹
- Solvable via key rotation ☺
- The *hops* trick that we learned from insertion applies here as well!
    - Look as far away as *hops* tells you to.

Example #3: Merging with sibling node



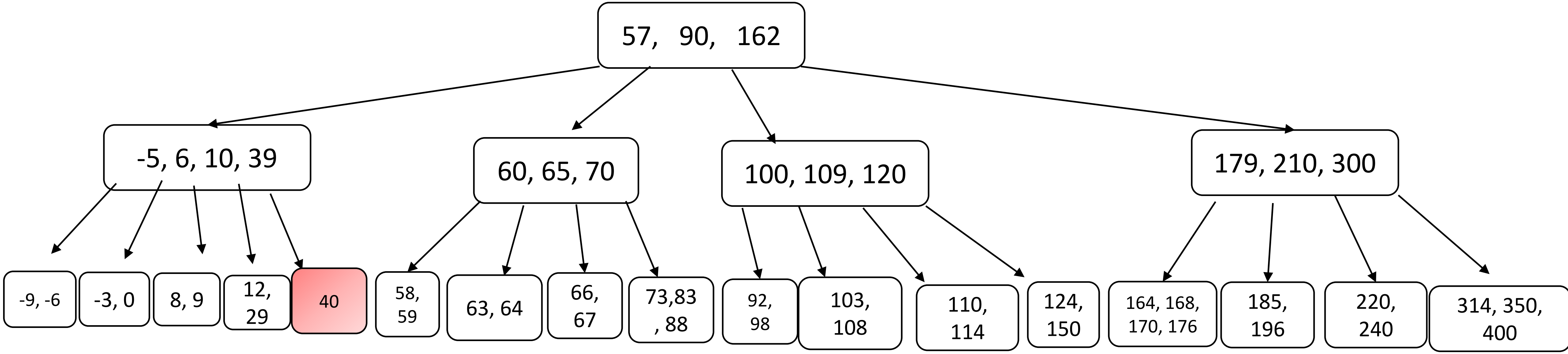
- Task: delete 50

Example #3: Merging with sibling node



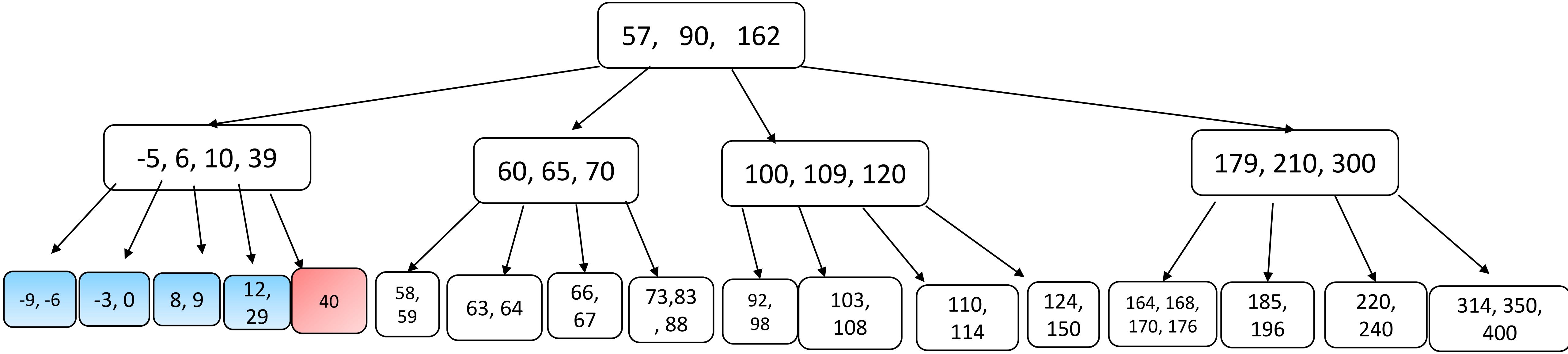
- Task: delete 50

Example #3: Merging with sibling node



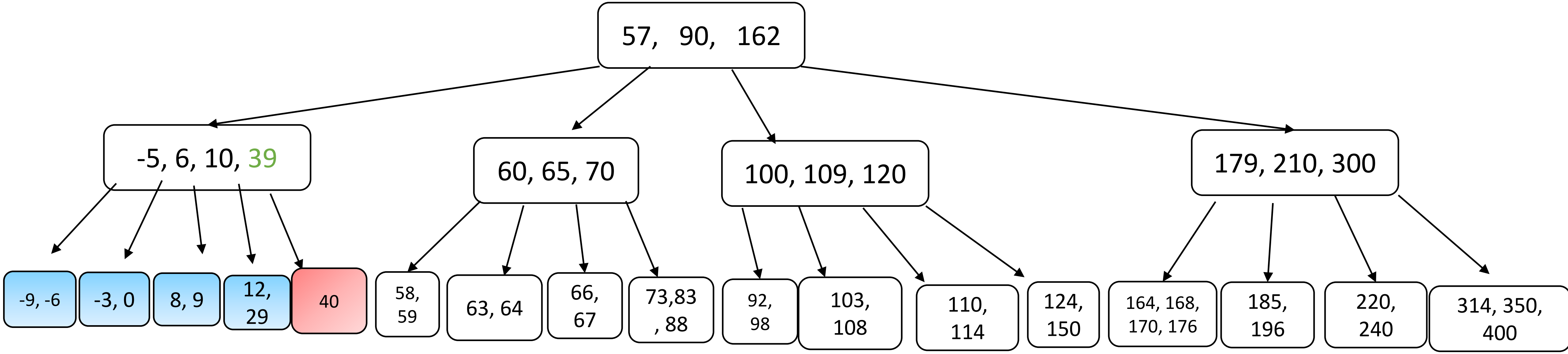
- Task: delete 50
- Underflow ☹

Example #3: Merging with sibling node



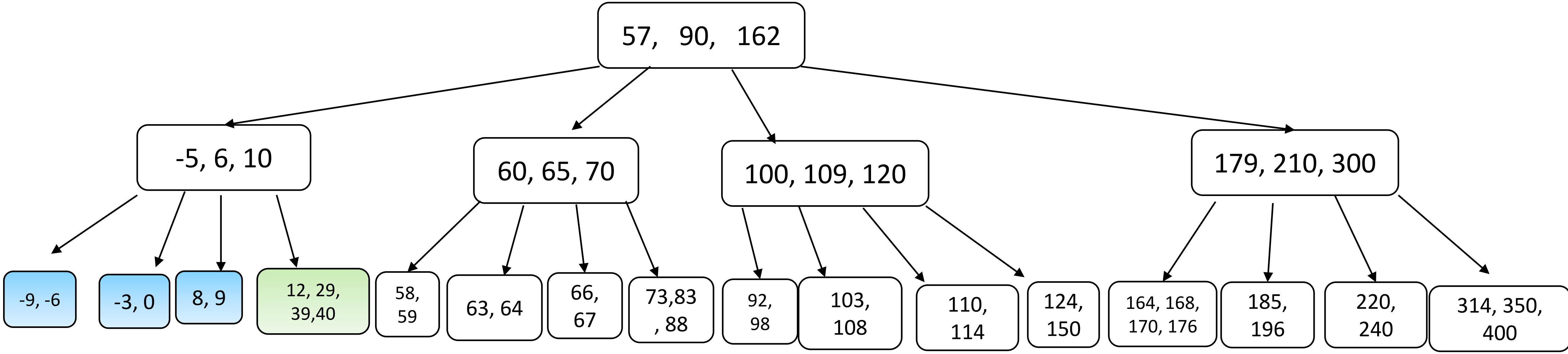
- Task: delete 50
- Underflow ☹
- Siblings can't help with key rotations ☹ (because min # keys= $\left\lceil\frac{p}{2}\right\rceil - 1 = 3 - 1 = 2$)

Example #3: Merging with sibling node



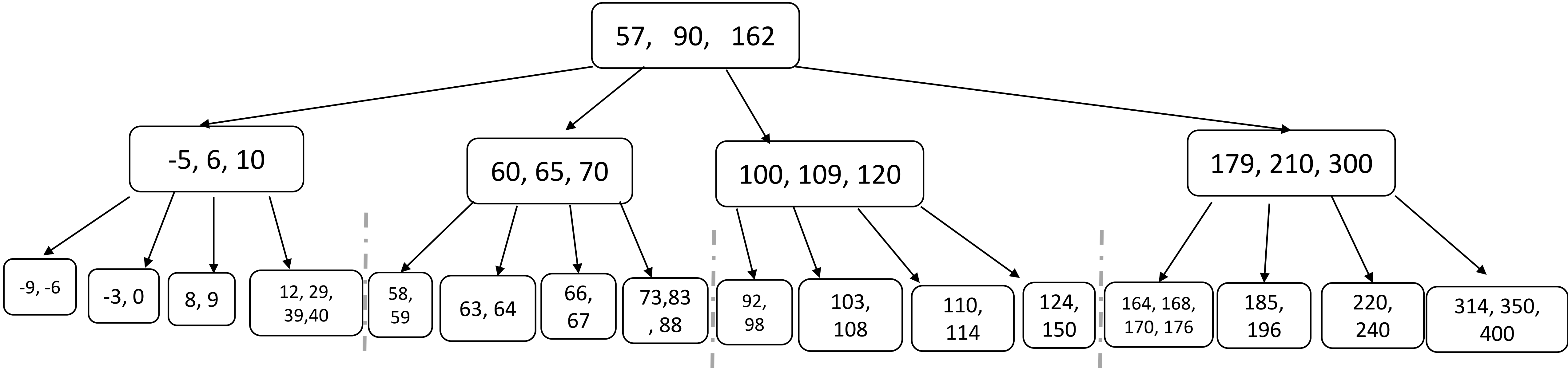
- Task: delete 50
- Underflow ☹
- Siblings can't help with key rotations ☹ (because min # keys= $\left\lceil \frac{p}{2} \right\rceil - 1 = 3 - 1 = 2$)
- Solution: Merge current node with sibling node and parent key.

Example #3: Merging with parent node



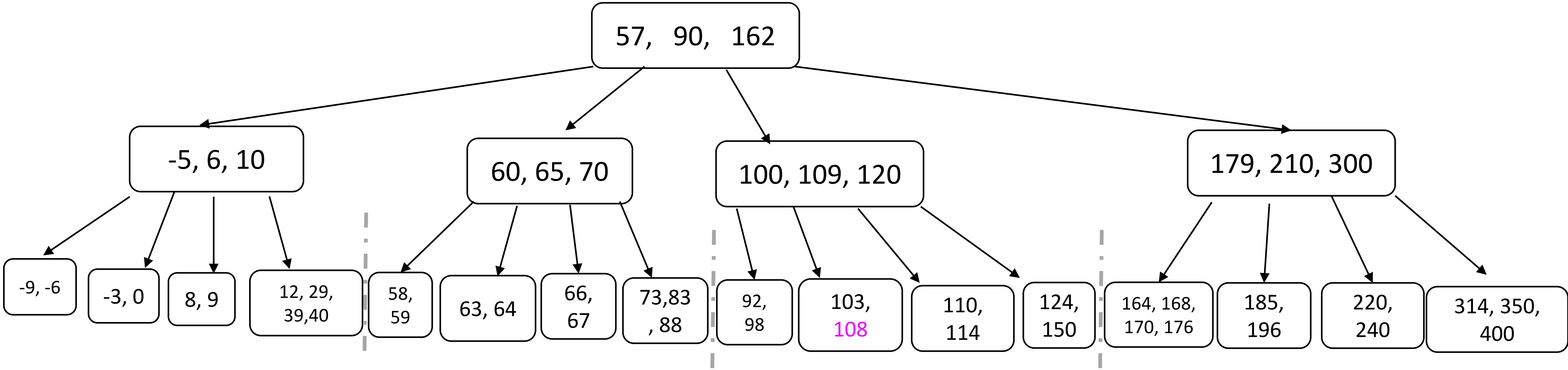
- Task: delete 50
- Underflow ☹
- Siblings can't help with key rotations ☹ (because min # keys= $\left\lceil \frac{p}{2} \right\rceil - 1 = 3 - 1 = 2$)
- Solution: Merge current node with sibling node and parent key.

Example #4: Merging with either sibling



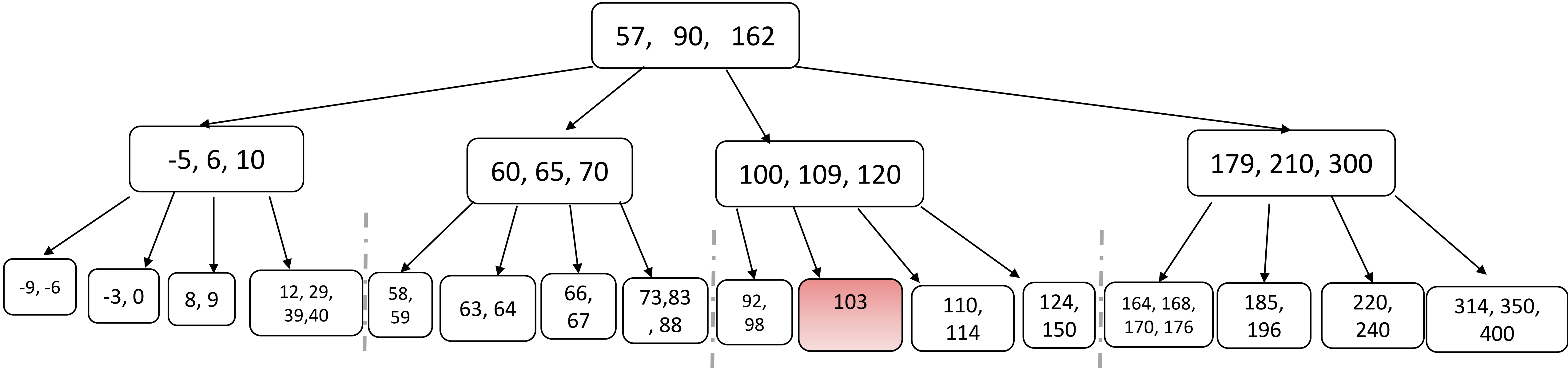
- Task: delete 108

Example #4: Merging with either sibling



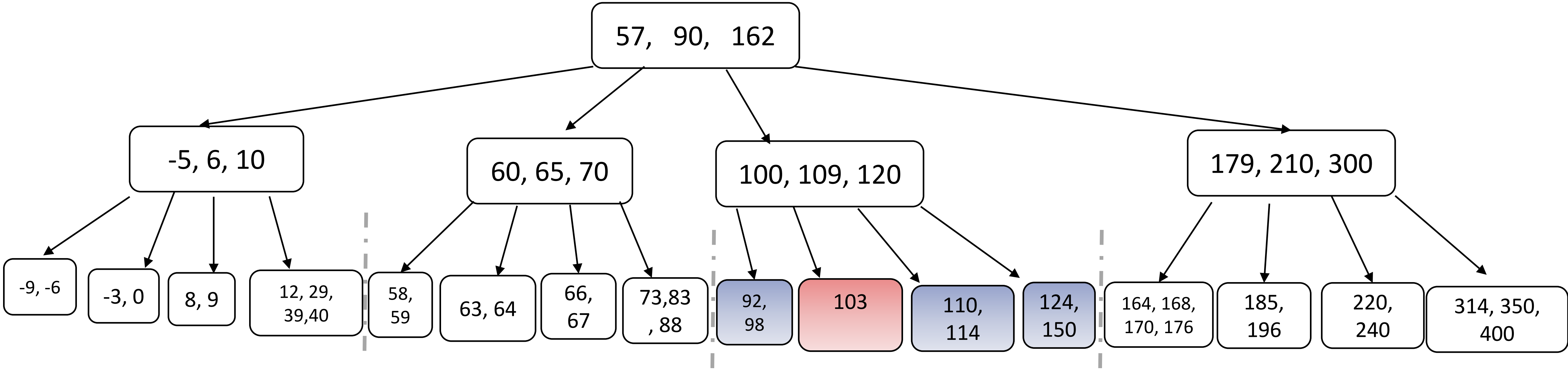
- Task: delete 108

Example #4: Merging with either sibling



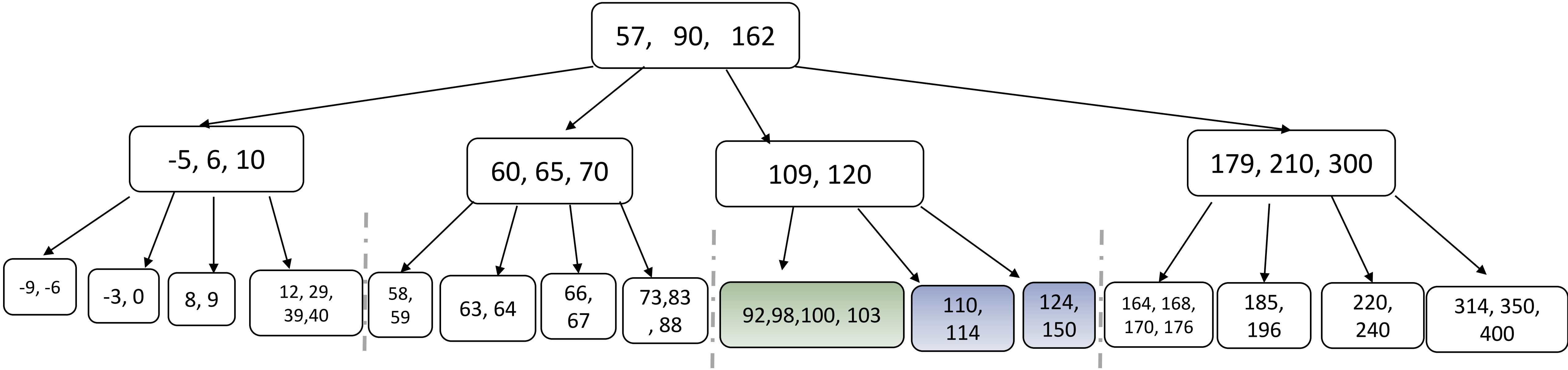
- Task: delete 108
- Underflow ☹

Example #4: Merging with either sibling



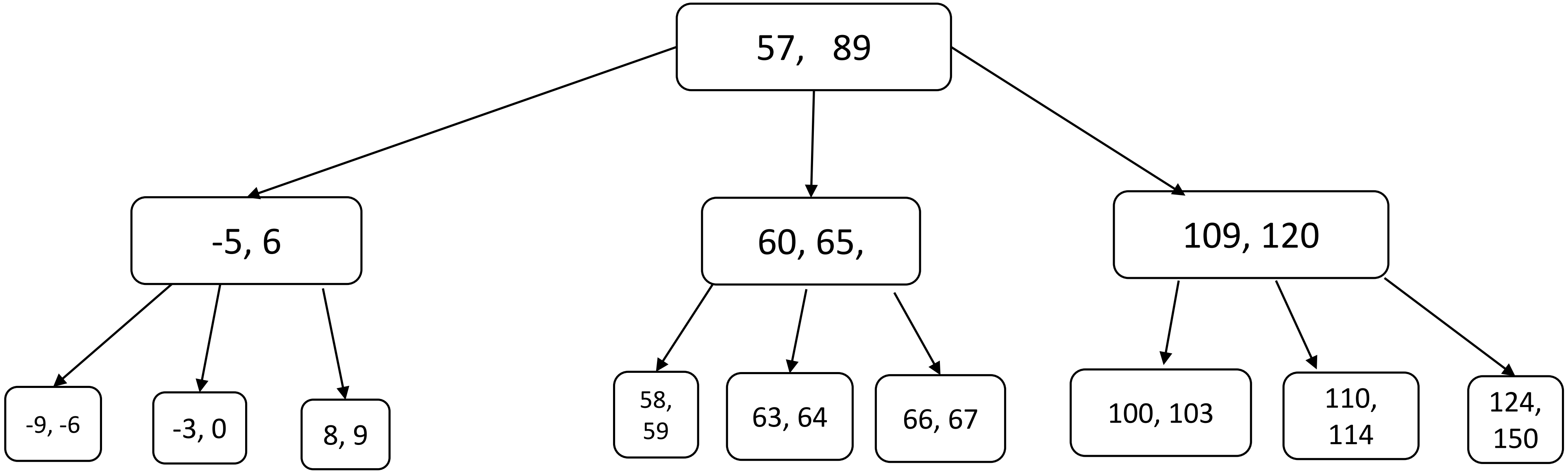
- Task: delete 108
- Underflow ☹
- Siblings can't help with key rotations ☹ (same reason)

Example #4: Merging with either sibling



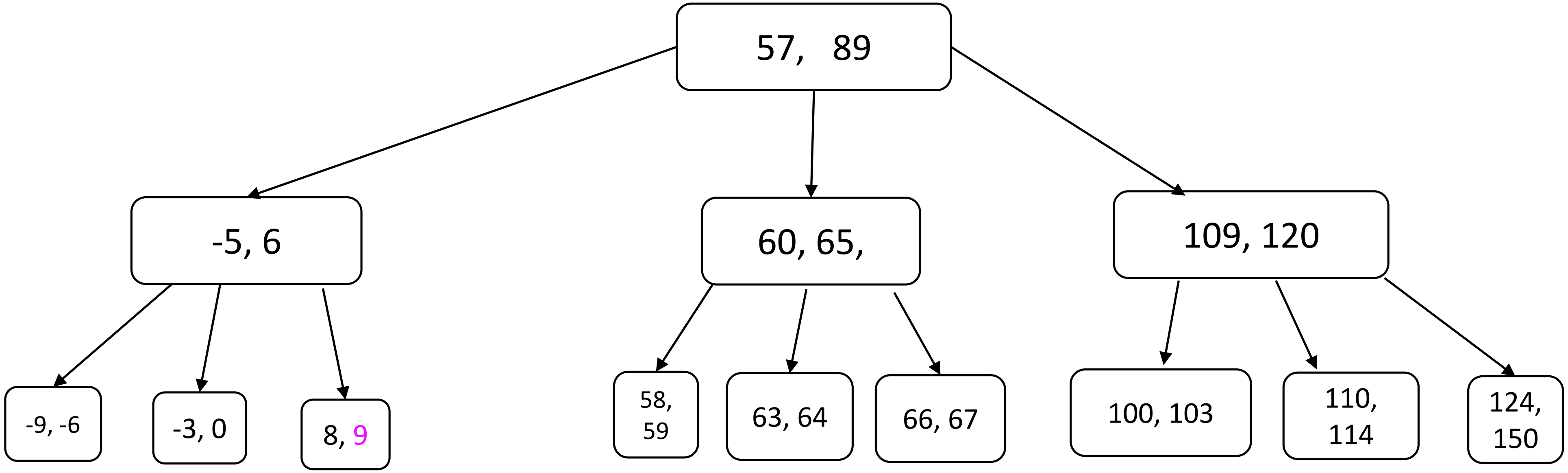
- Task: delete 108
- Underflow ☹
- Siblings can't help with key rotations ☹ (same reason)
- A relief force appears in the form of a sibling-parent key merging ☺

Example #5: Merging all the way to the root.



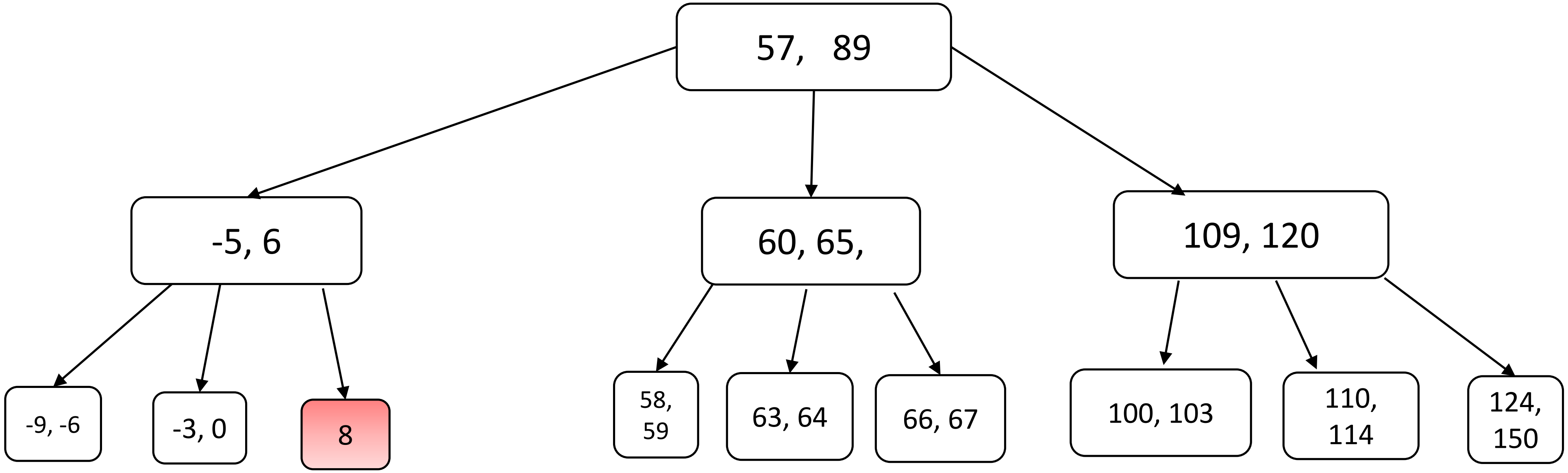
- Task: delete 9

Example #5: Merging all the way to the root.



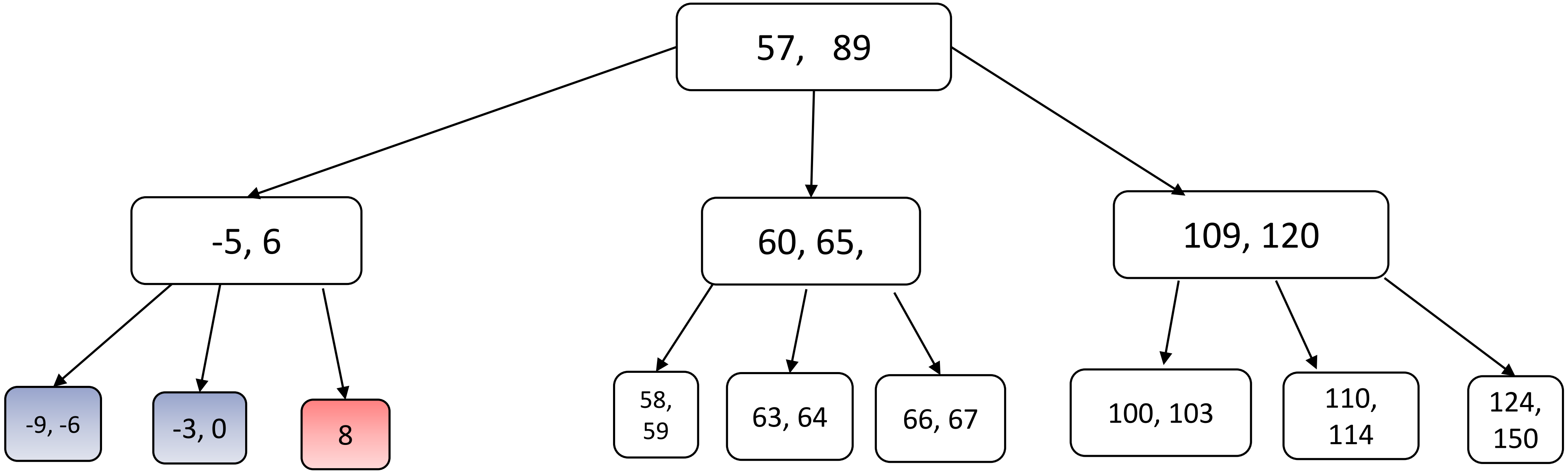
- Task: delete 9

Example #5: Merging all the way to the root.



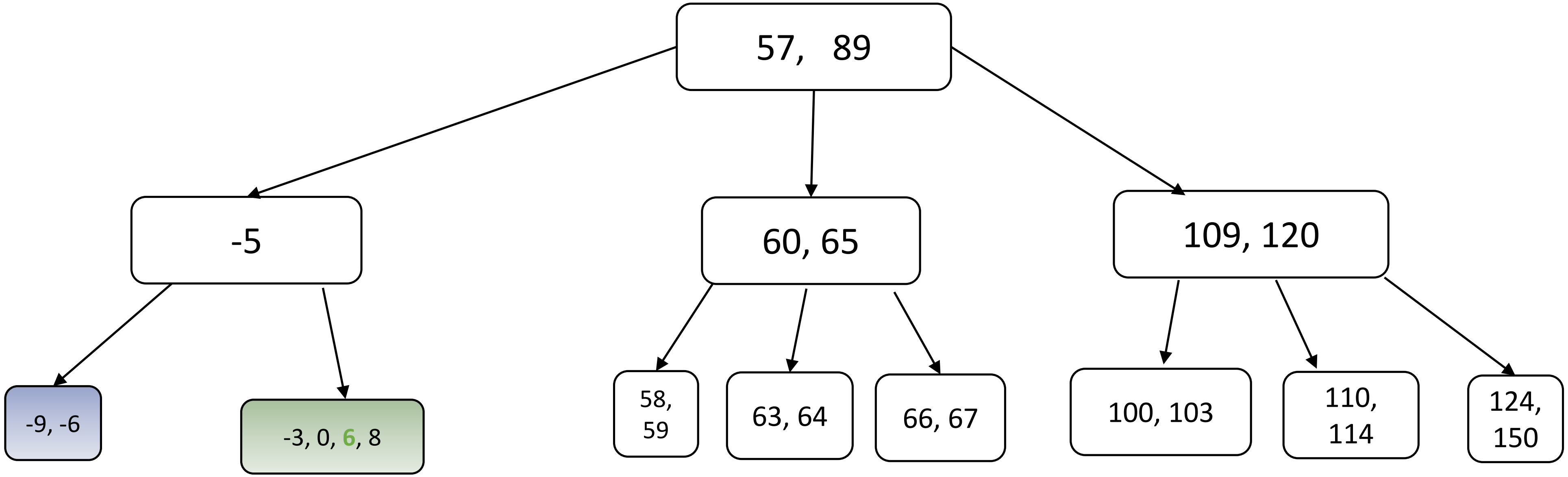
- Task: delete 9
- Underflow ☹

Example #5: Merging all the way to the root.



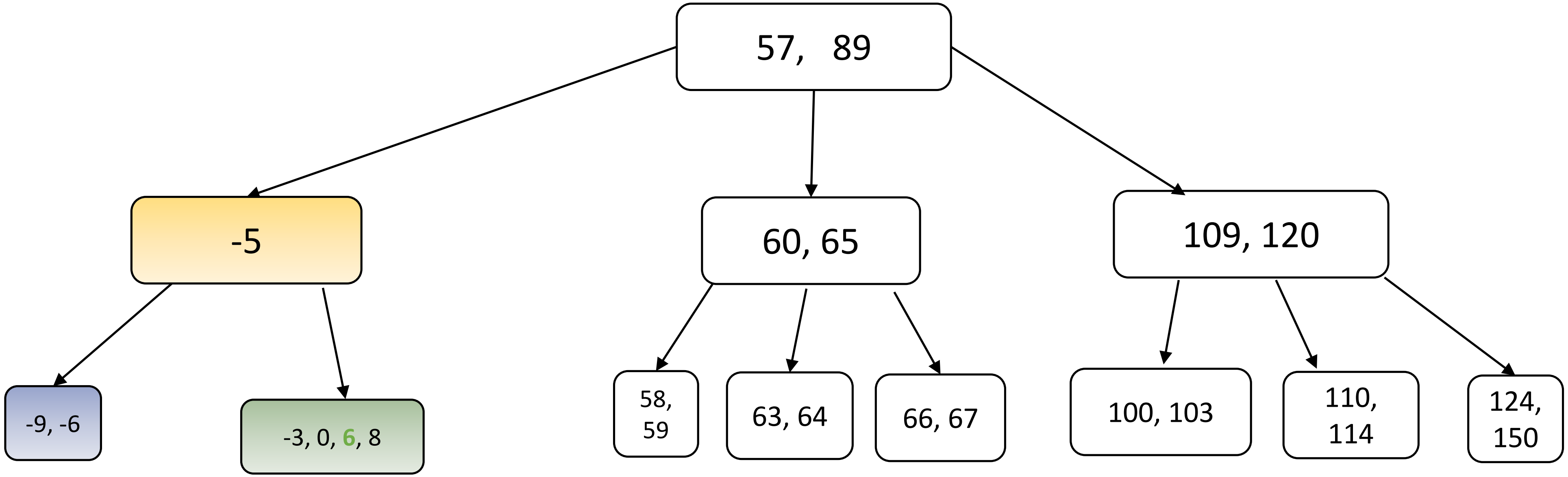
- Task: delete 9
- Underflow ☹
- Siblings can’t help with key rotations! ☹

Example #5: Merging all the way to the root.



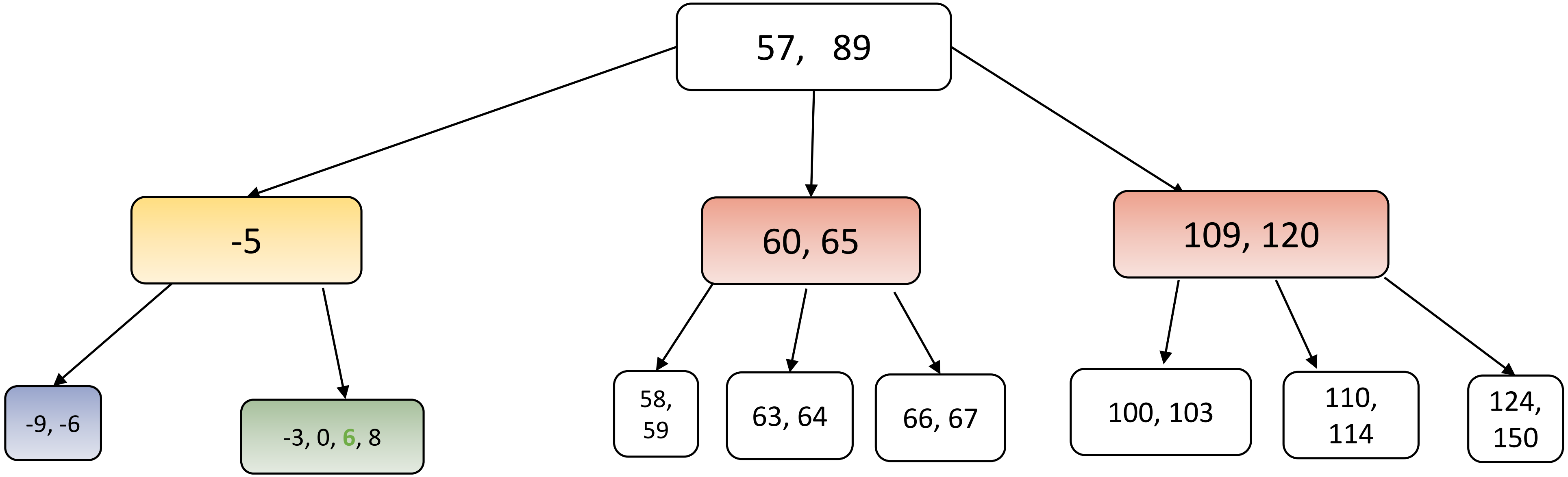
- Task: delete 9
- Underflow ☹
- Siblings can't help with key rotations! ☹
- Merging with parent key and sibling ☺

Example #5: Merging all the way to the root.



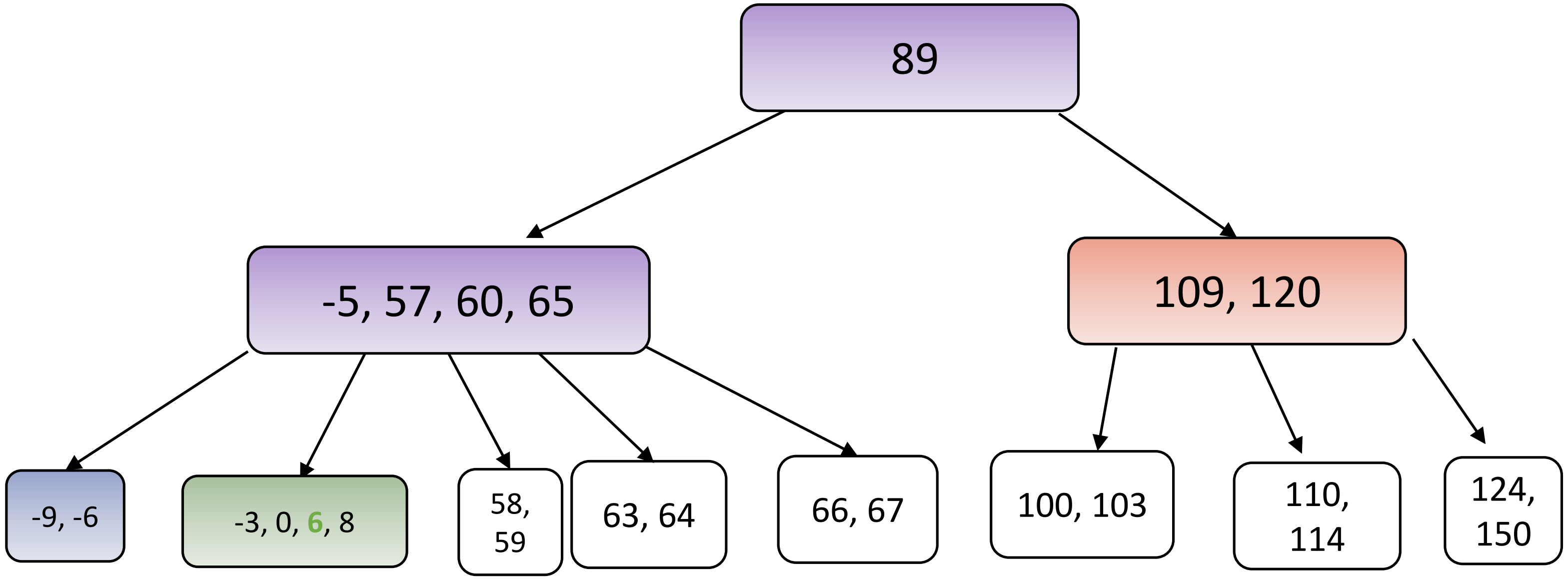
- Task: delete 9
- Underflow ☹
- Siblings can't help with key rotations! ☹
- Merging with parent key and sibling ☺
- Parent underflows ☹

Example #5: Merging all the way to the root.



- Task: delete 9
- Underflow ☹
- Siblings can’t help with key rotations! ☹
- Merging with parent key and sibling ☺
- Parent underflows ☹
- Siblings can’t help with rotations☹

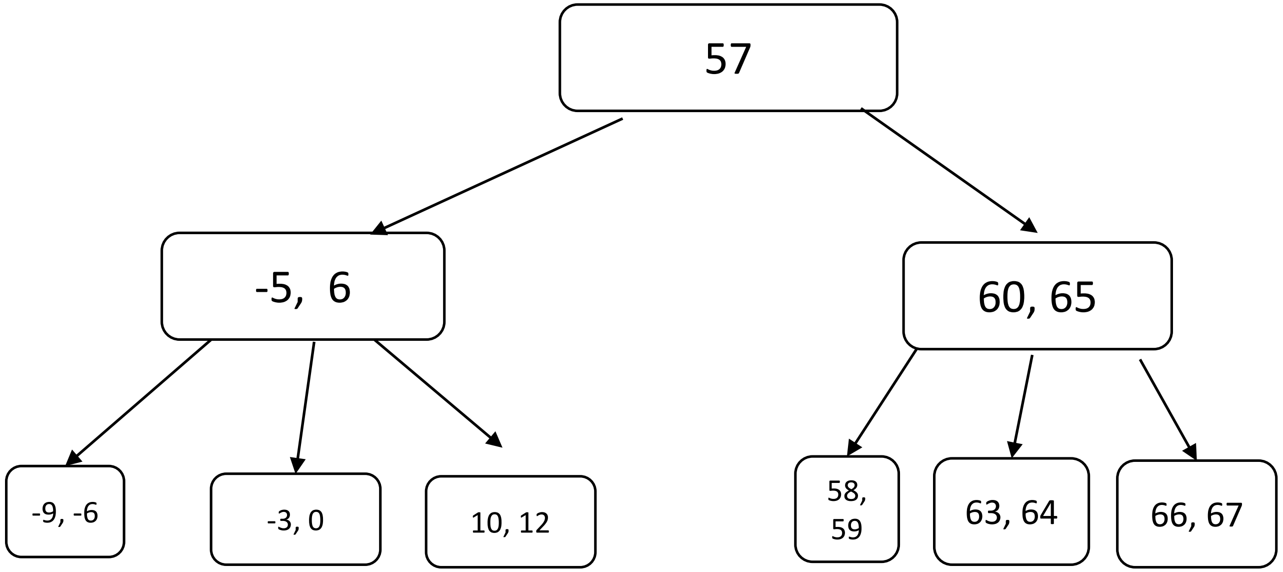
Example #5: Merging all the way to the root.



- Task: delete 9
- Underflow ☹
- Siblings can’t help with key rotations! ☹
- Merging with parent key and sibling ☺
- Parent underflows ☹
- Siblings can’t help with rotations ☹

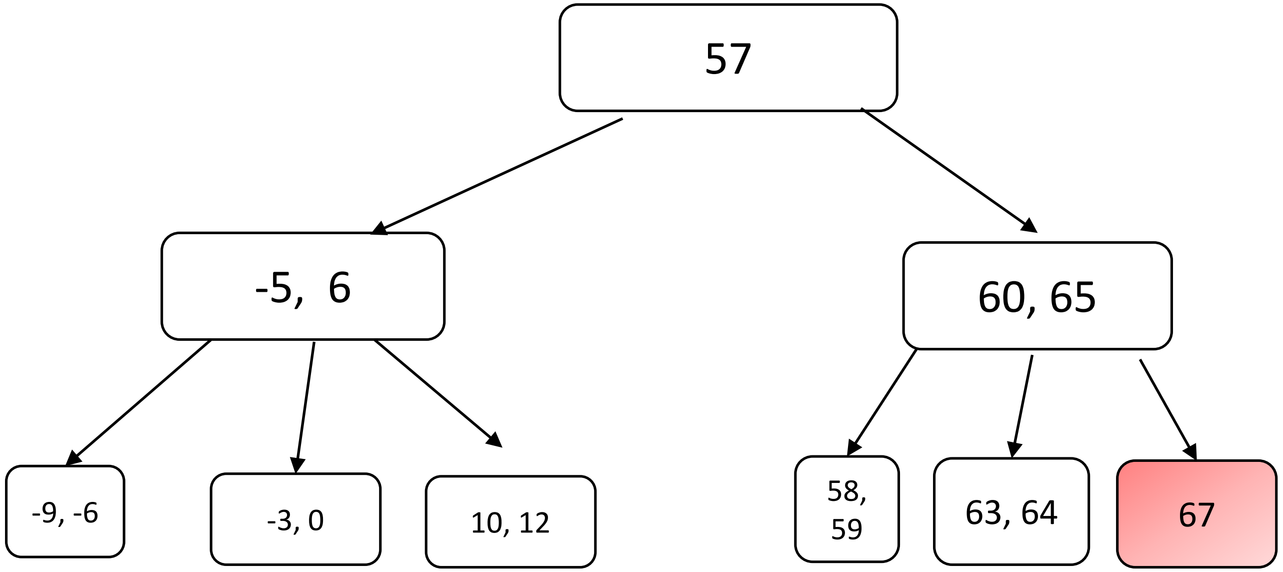
Solution: Merge with parent key and sibling node! ☺

Example #6: Root underflow



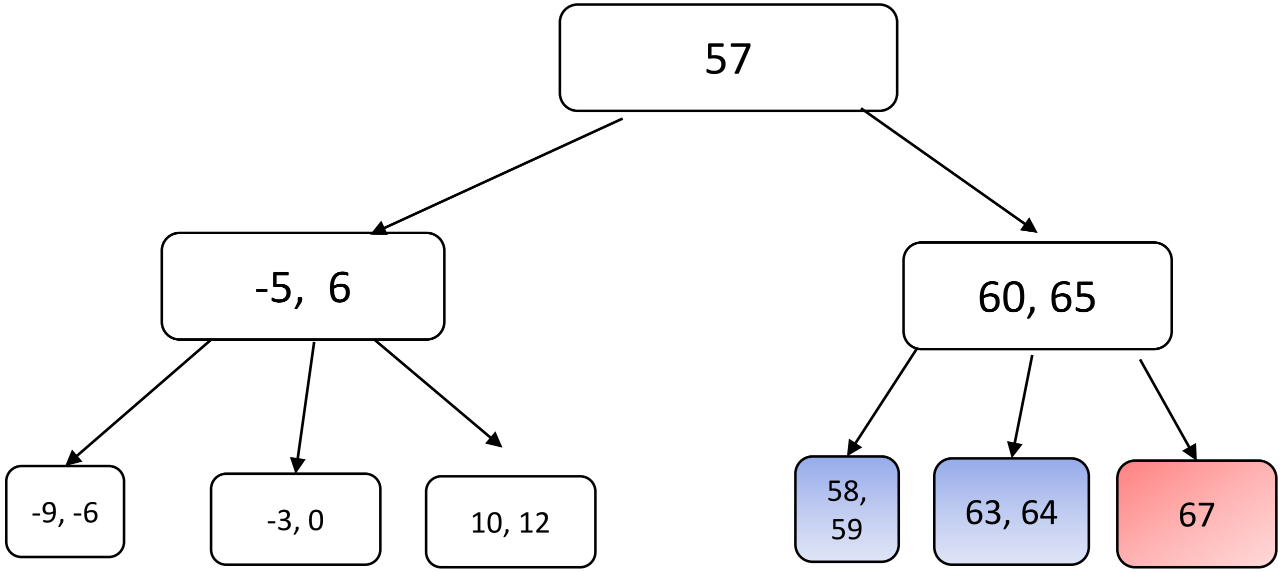
- Task: delete 66

Example #6: Root underflow



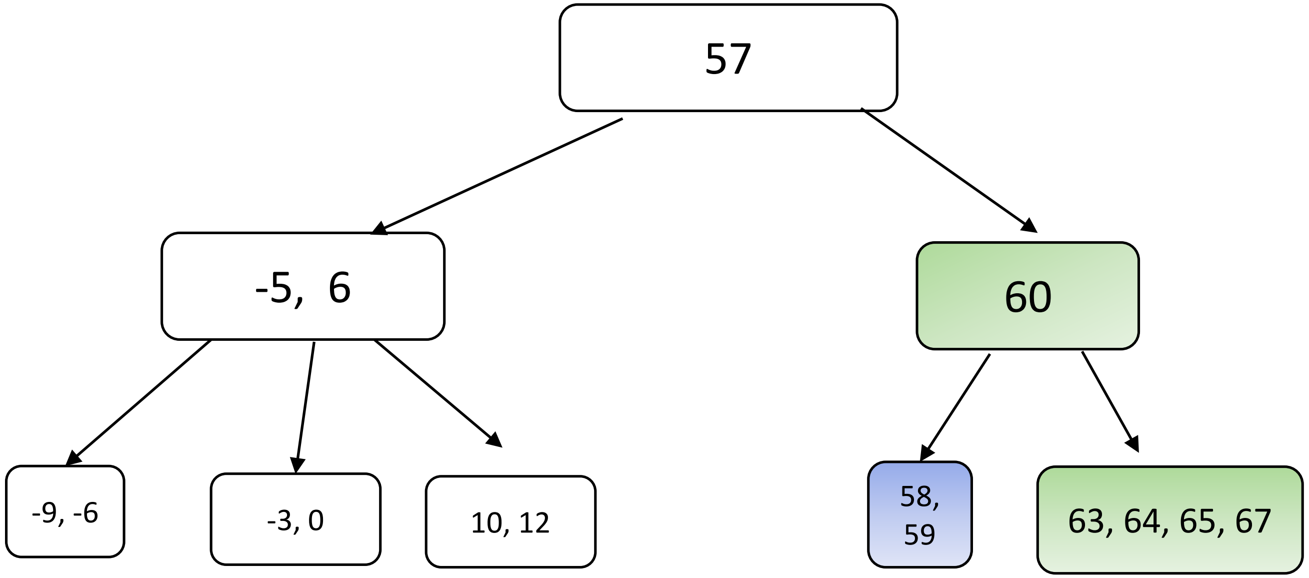
- Task: delete 66
- Underflow ☹

Example #6: Root underflow



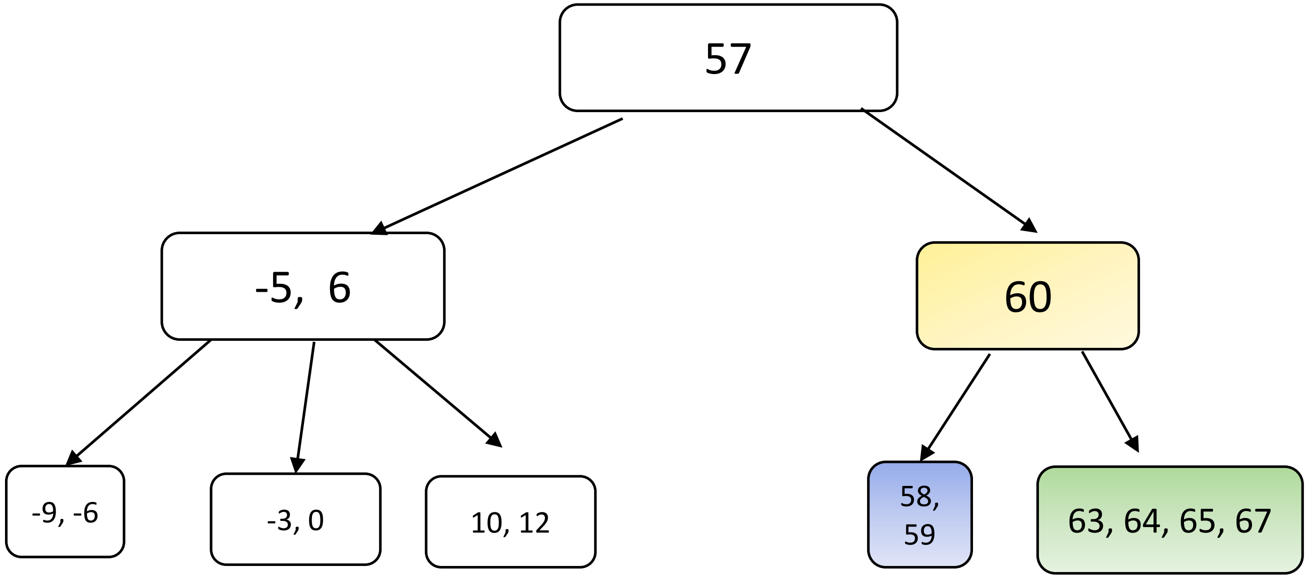
- Task: delete 66
- Underflow ☹
- Siblings can't help with key rotations ☹

Example #6: Root underflow



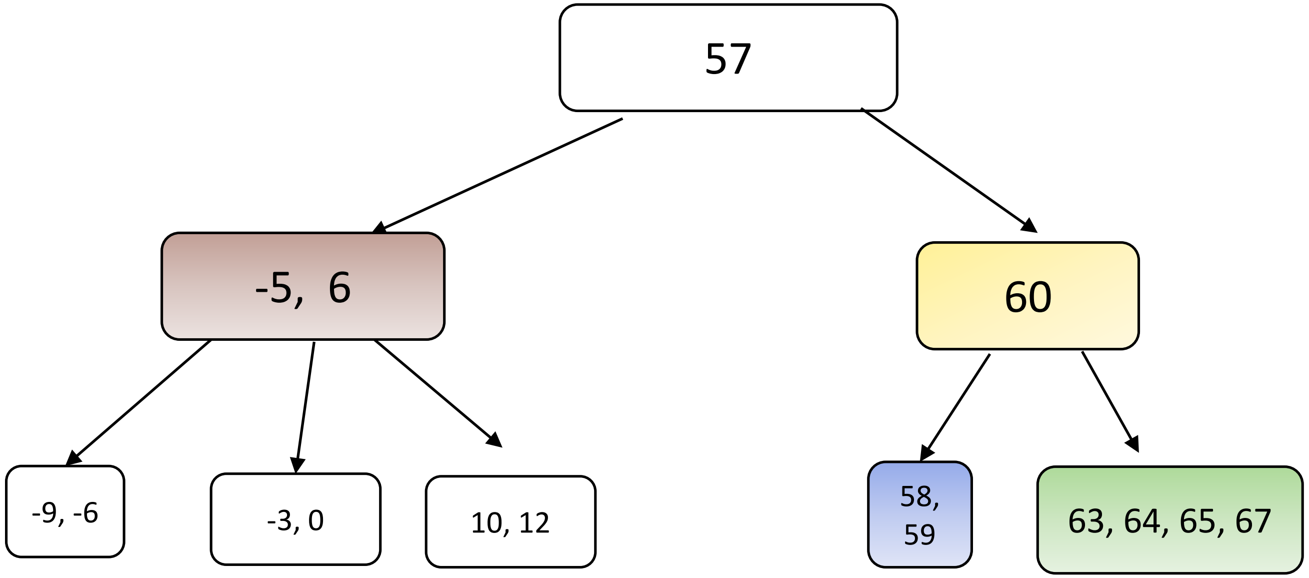
- Task: delete 66
- Underflow ☹
- Siblings can't help with key rotations ☹
- Merge ☺

Example #6: Root underflow



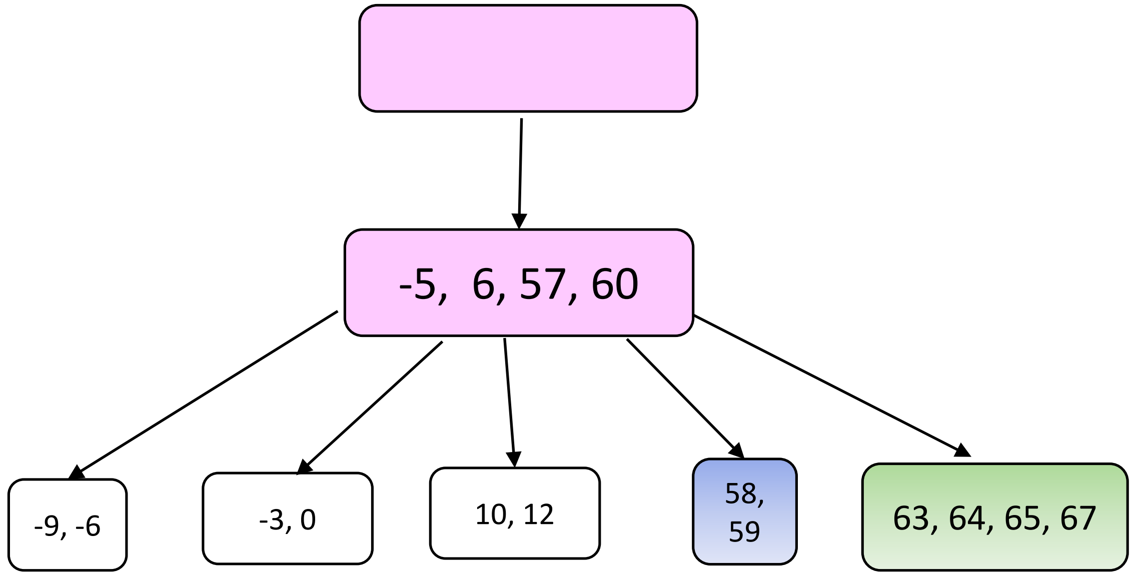
- Task: delete 66
- Underflow ☹
- Siblings can’t help with key rotations ☹
- Merge ☺
- Underflow ☹

Example #6: Root underflow



- Task: delete 66
- Underflow ☹
- Siblings can't help with key rotations ☹
- Merge ☺
- Underflow ☹
- Siblings can't help with key rotations ☹

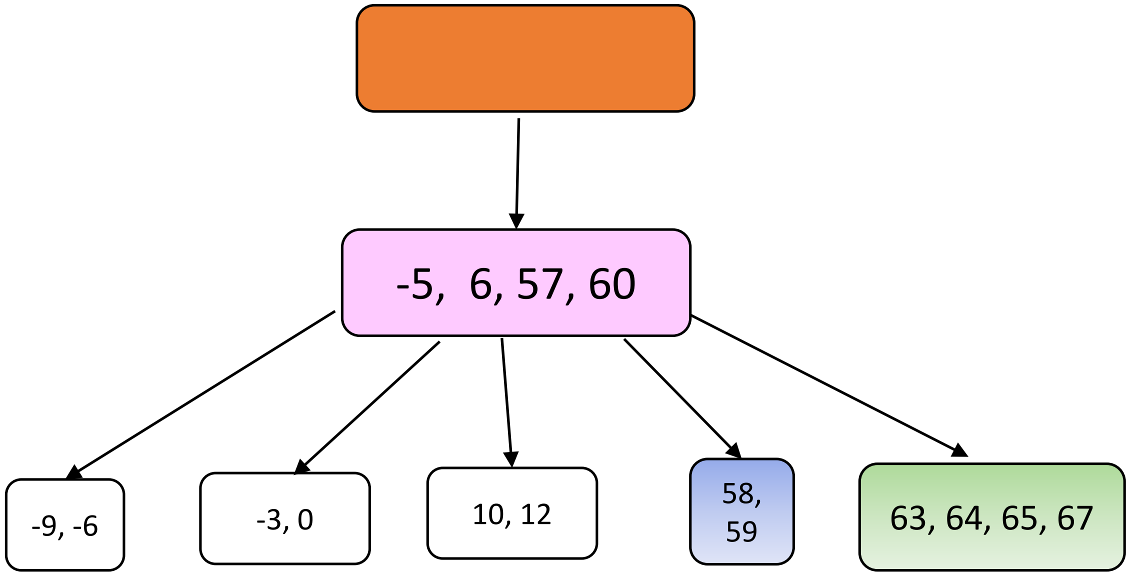
Example #6: Root underflow



- Task: delete 66
- Underflow ☹
- Siblings can’t help with key rotations ☹
- Merge ☺
- Underflow ☹
- Siblings can’t help with key rotations ☹

Merge ☺

Example #6: Root underflow

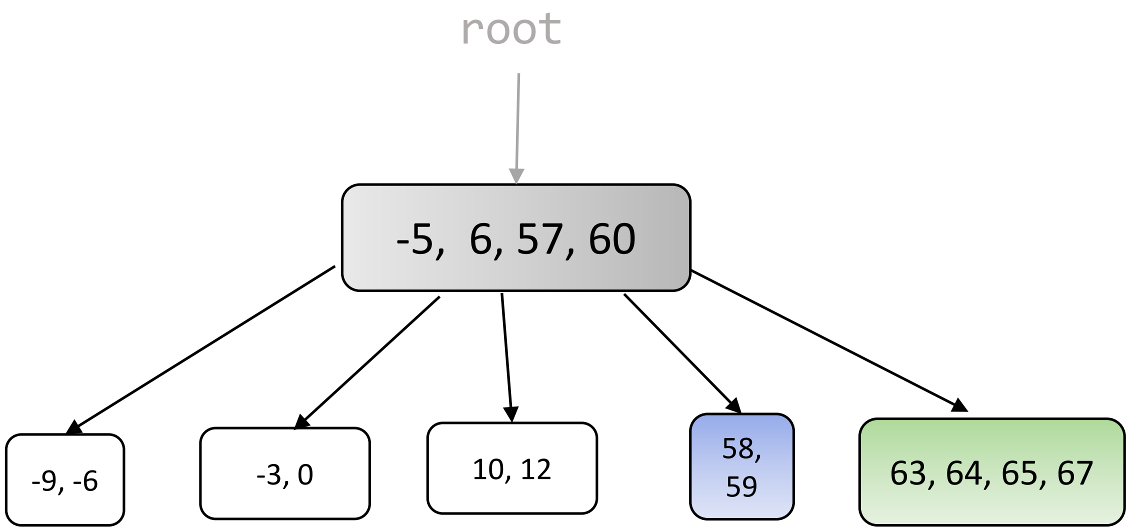


- Task: delete 66
- Underflow ☹
- Siblings can't help with key rotations ☹
- Merge ☺
- Underflow ☹
- Siblings can't help with key rotations ☹

Merge ☺

Root underflows ☹

Example #6: Root underflow



- Task: delete 66
- Underflow ☹
- Siblings can’t help ☹
- Merge ☺
- Underflow ☹
- Siblings can’t help ☹

- Merge ☺
- Root underflows ☹
- Delete it, and set new root to its only child! ☺

Searching a B-Tree

- Searching has effectively been covered by presenting insertion and deletion.
- At every level, I will query the node via binary search. Either:
  - I will find the key, so I'm done, or
  - I will follow the relevant pointer to a subtree, and repeat.
- Worse case: I have to go all the way to the leaves.

Quiz

- I want to measure the worst-case search cost in a B-Tree with branching factor p.
- **Unit cost**: comparison of two keys via `compareTo()`.

Quiz

- I want to measure the worst-case search cost in a B-Tree with branching factor p.
- **Unit cost**: comparison of two keys via `compareTo()`.
- Question: **in the worst case**, how many comparisons will I make for a key search in a B-Tree with n ***keys***?

Quiz

- I want to measure the worst-case search cost in a B-Tree with branching factor p.
- **Unit cost**: comparison of two keys via `compareTo()`.
- Question: **in the worst case**, how many comparisons will I make for a key search in a B-Tree with n ***keys***?

$\log_p n$

$p * log_p n$

$(p-1) * log_p n$

Something else (what?)

Quiz

- I want to measure the worst-case search cost in a B-Tree with branching factor p.
- **Unit cost**: comparison of two keys via `compareTo().`
- Question: **in the worst case**, how many comparisons will I make for a key search in a B-Tree with n ***keys***?

$$\log_2(p-1) * \log_p n \approx \log_2 n$$

$\log_p n$

$p * log_p n$

$(p-1) * log_p n$

Something else (what?)

With the keys **sorted by construction**, it really is dumb to not do binary search in every node! ;)

So, why even care?

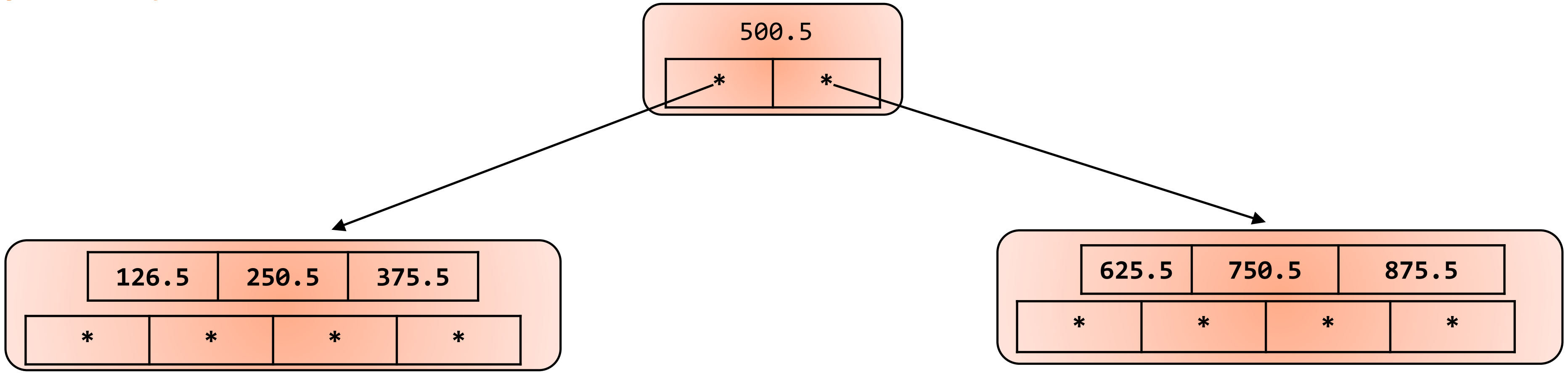
- Why even care about B-Trees if their search is $O(\log_2 n)$?

So, why even care?

- Why even care about B-Trees if their search is $\mathcal{O}(\log_2 n)$?
- Because of the **hugely successful** ***B+ ("B-plus") trees***, of which they are the main component!

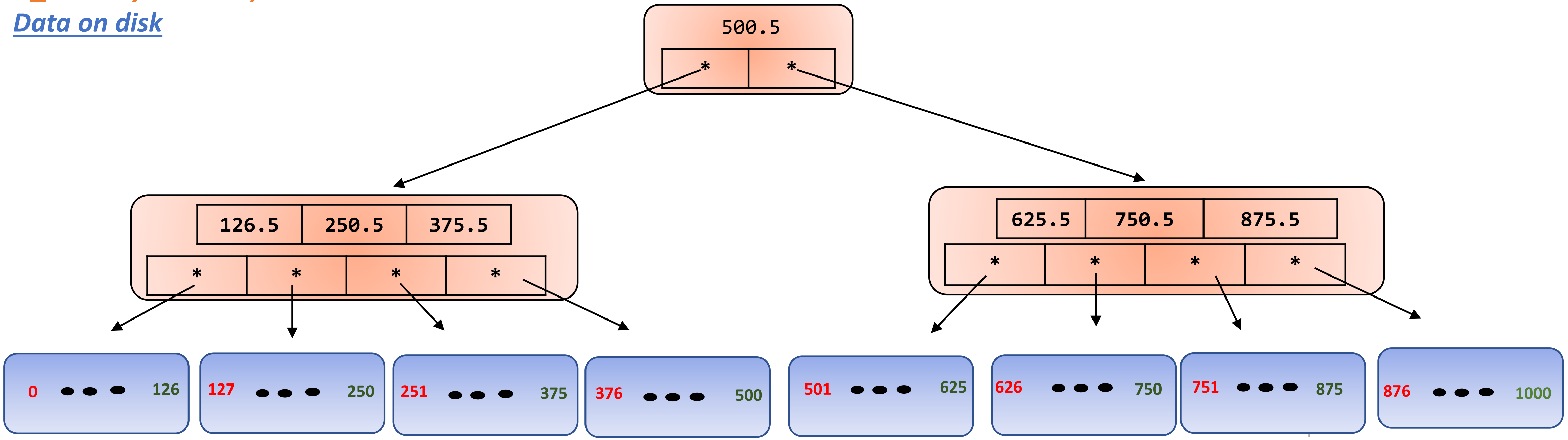
The components of a B+-tree

(1) A (memory-resident) B-Tree



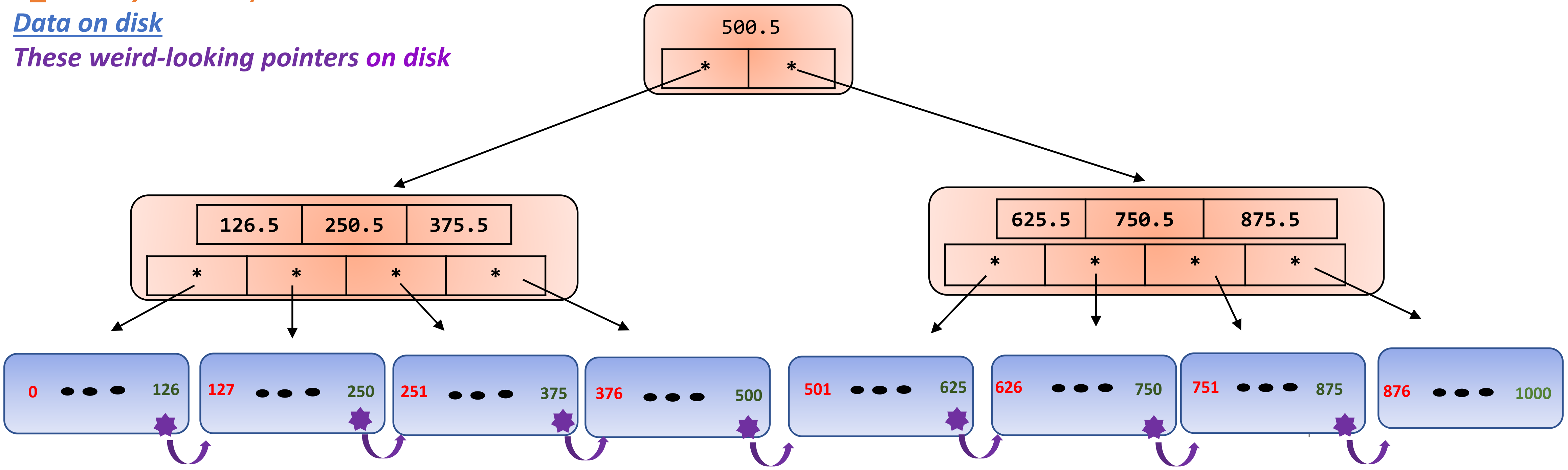
The components of a B+-tree

(1) A (memory-resident) B-Tree
(2) Data on disk



The components of a B+-tree

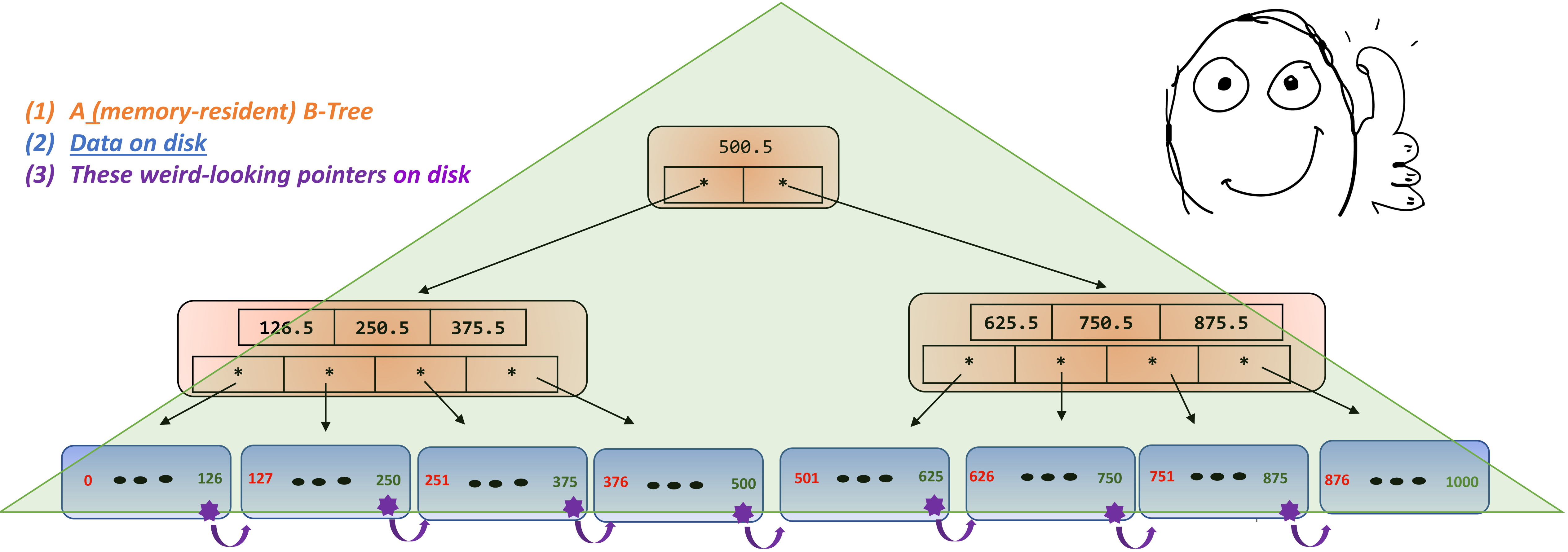
(1) A (memory-resident) B-Tree
(2) Data on disk
(3) These weird-looking pointers on disk



The components of a B+-tree

Give us a B$^+$-tree ("B – plus tree") index!

(1) A (memory-resident) B-Tree
(2) Data on disk
(3) These weird-looking pointers on disk



Are buffers a good storage choice?

- **Issue**: Static storage buffers for keys and references waste memory space for nodes.
    - Proposal: Use linked lists instead!
- This is actually not a good idea.
- Counter-arguments:
    1. Buffers waste space **MINIMAL** compared to the size of the DB (recall earlier results)
    2. Static buffers allow us to perform super-efficient binary search for seeking keys at every node
    3. In 2019, memory is cheaper than it used to be in 1970; no reason to over-optimize!

Are buffers a good storage idea?

- **Issue**: Static storage buffers for keys and references waste memory space for nodes.
  - Proposal: Use linked lists instead!
- This is actually not a good idea.
- Counter-arguments:
  1. Buffers waste space **MINIMAL** compared to the size of the DB (recall earlier results)
  2. Static buffers allow us to perform super-efficient binary search for seeking keys at every node
  3. In 2019, memory is cheaper than it used to be in 1970;

  no reason to over-optimize!
- Conclusion: yes, they are an excellent storage choice.