Introduction to Algorithms
10. Balanced Search Trees

Hyungsoo Jung

Scalable Computing Systems Laboratory
Hanyang University

Balanced search trees

Balanced search tree: A search-tree data structure for which a height of $O(\lg n)$ is guaranteed when implementing a dynamic set of n items.

Examples:

- AVL trees
- 2-3 trees
- 2-3-4 trees
- B-trees
- Red-black trees

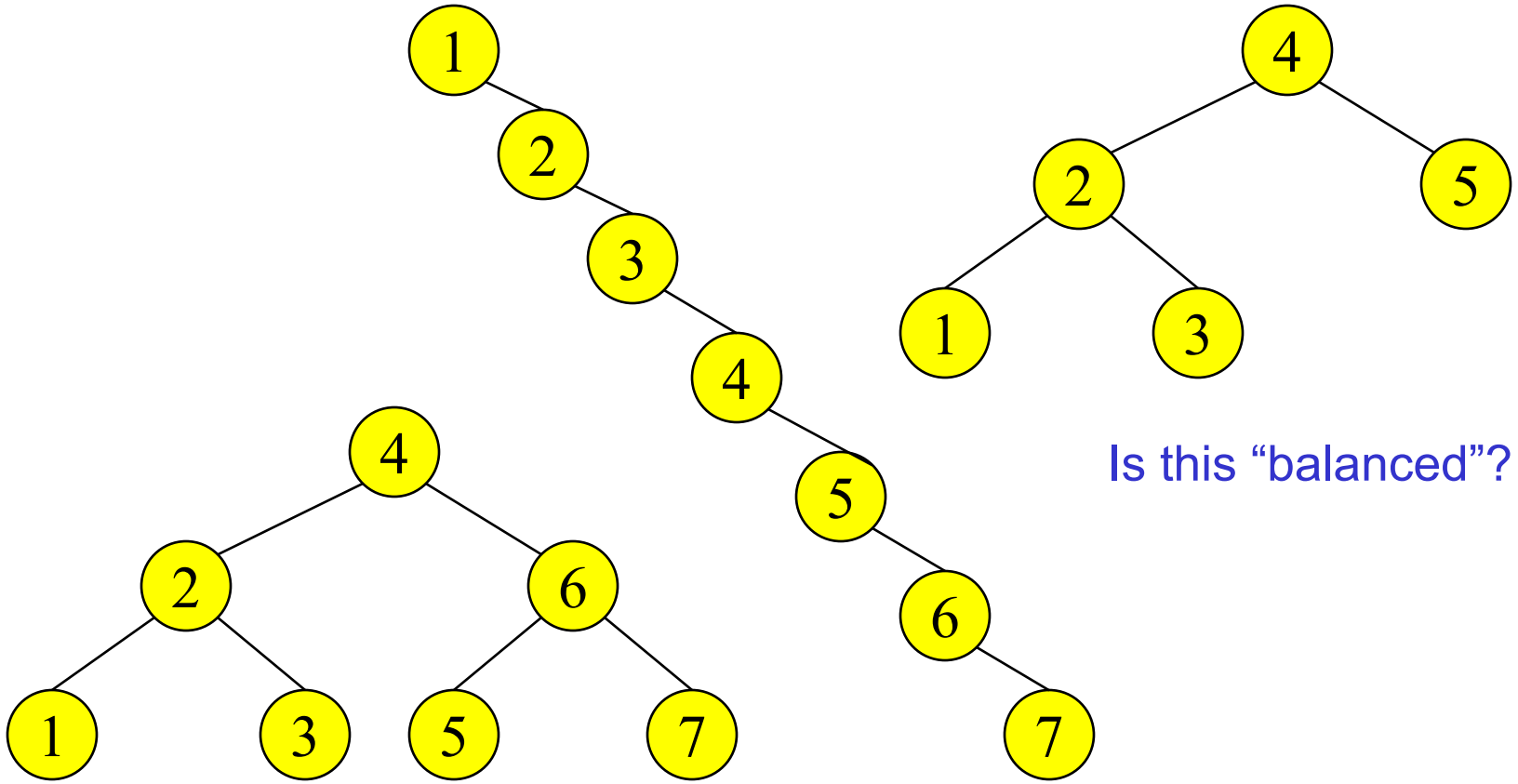
Binary Search Tree - Best Time

- All BST operations are O(d), where d is tree depth
- minimum d is $d = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree - Worst Time

- Worst case running time is O(N)
  - › What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - › Problem: Lack of "balance":
    - compare depths of left and right subtree
  - › Unbalanced degenerate tree

Balanced and unbalanced BST



Is this “balanced”?

Approaches to balancing trees

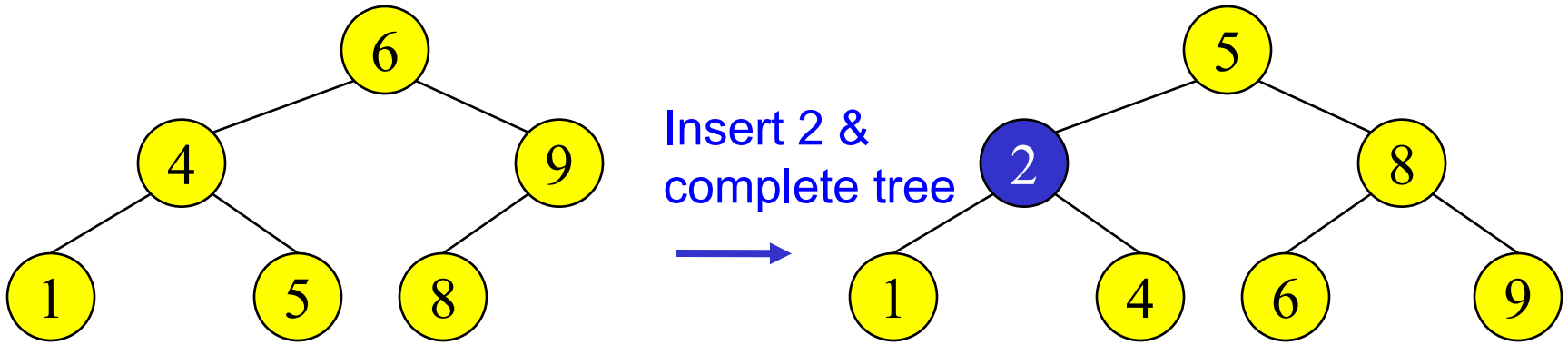
- Don't balance
  - › May end up with some nodes very deep
- Strict balance
  - › The tree must always be balanced perfectly
- Pretty good balance
  - › Only allow a little out of balance
- Adjust on access
  - › Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

Perfect Balance

- Want a complete tree after every operation
  - › tree is full except possibly in the lower right
- This is expensive
  - › For example, insert 2 in the tree on the left and then rebuild as a complete tree

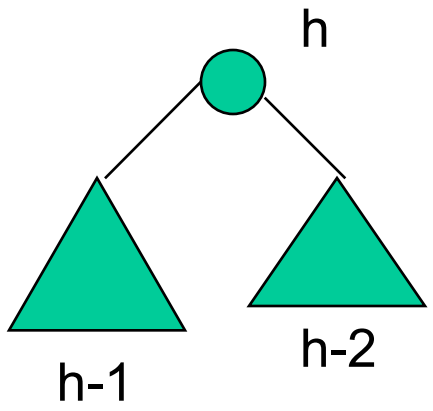


AVL - Good but not Perfect Balance

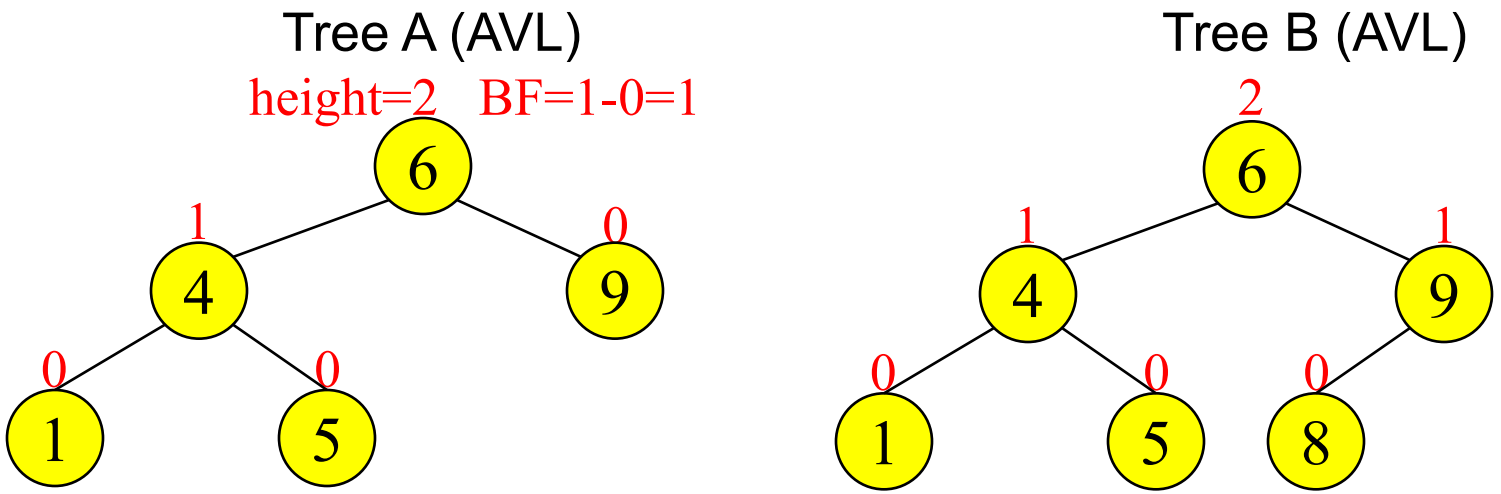
- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

Height of an AVL Tree

- N(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - N(0) = 1, N(1) = 2
- Induction
  - N(h) = N(h-1) + N(h-2) + 1
- Solution (recall Fibonacci analysis)
  - $N(h) \geq \phi^h \quad (\phi \approx 1.62)$

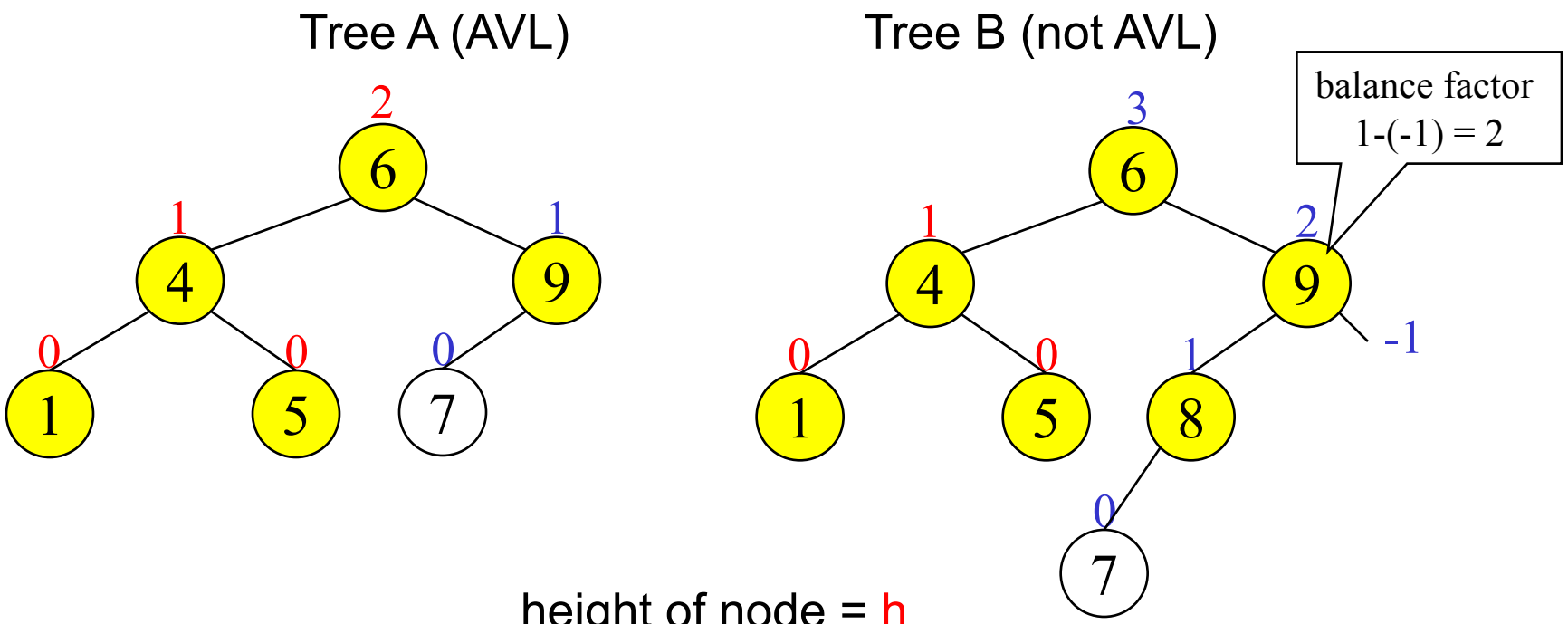


Node Heights



height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

Node Heights after Insert 7

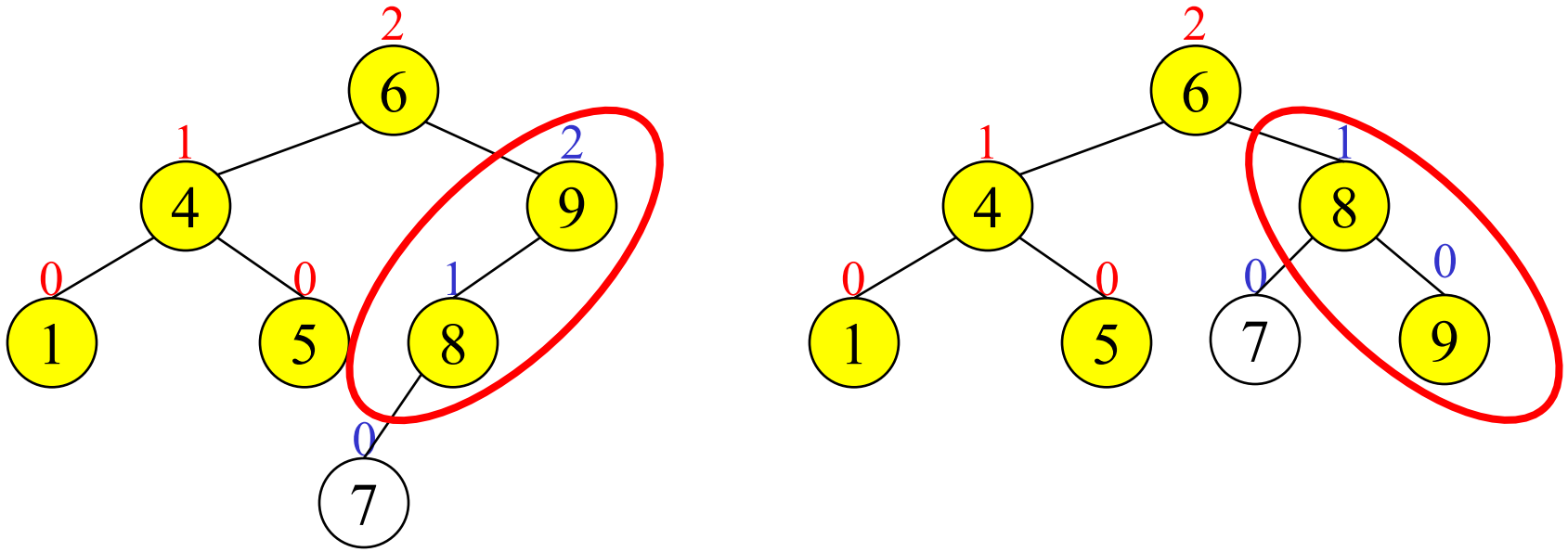


height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree



Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

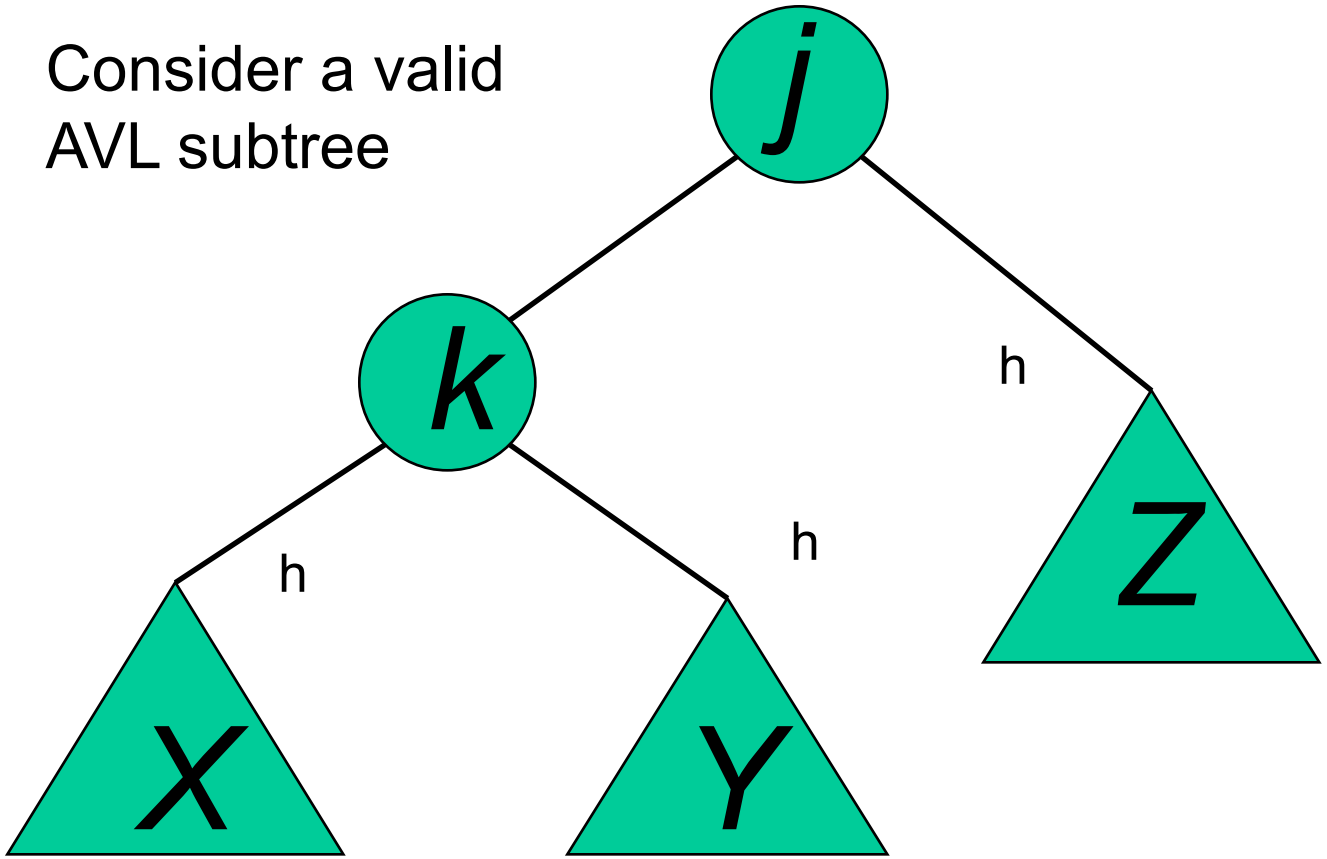
Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

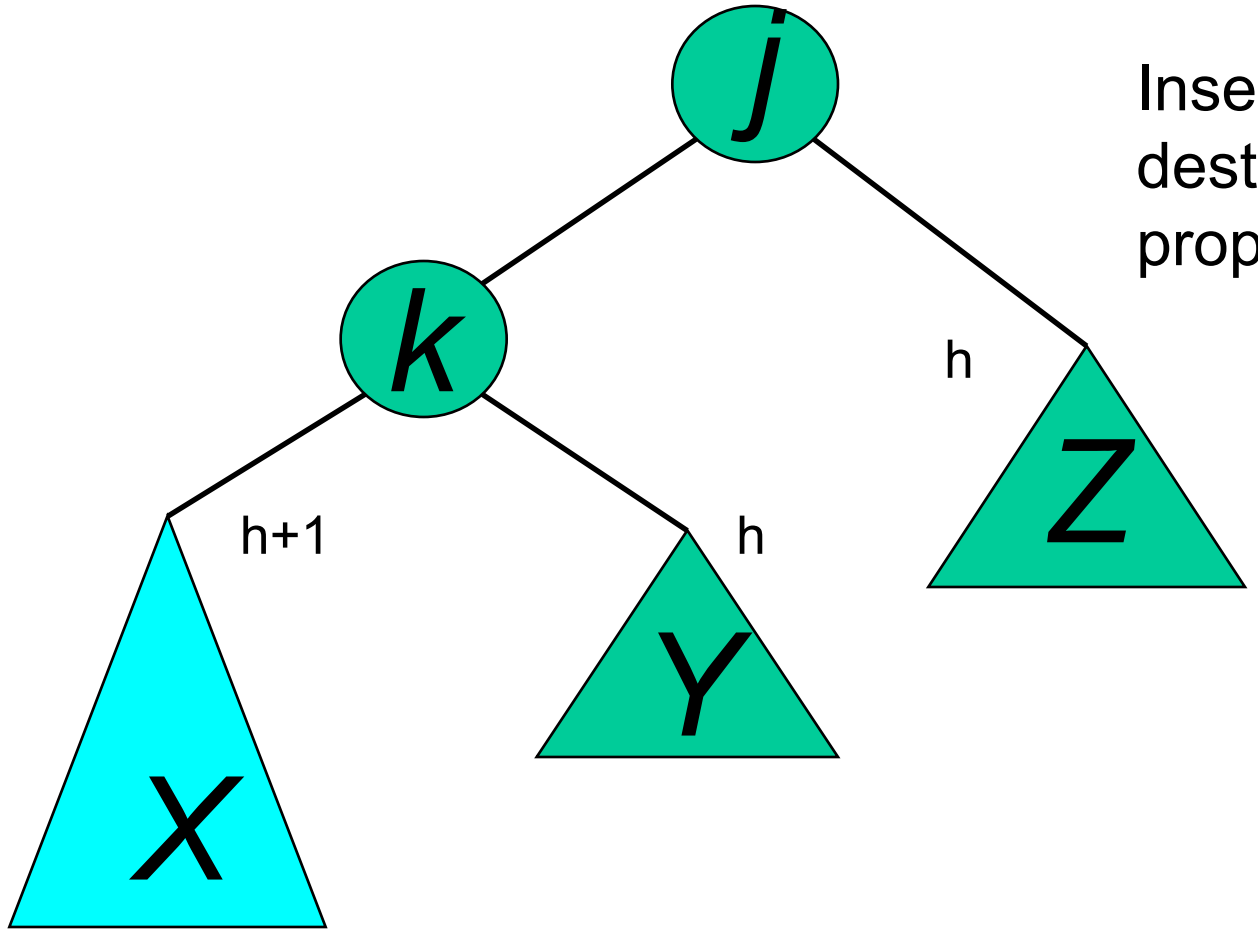
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Outside Case

Consider a valid
AVL subtree

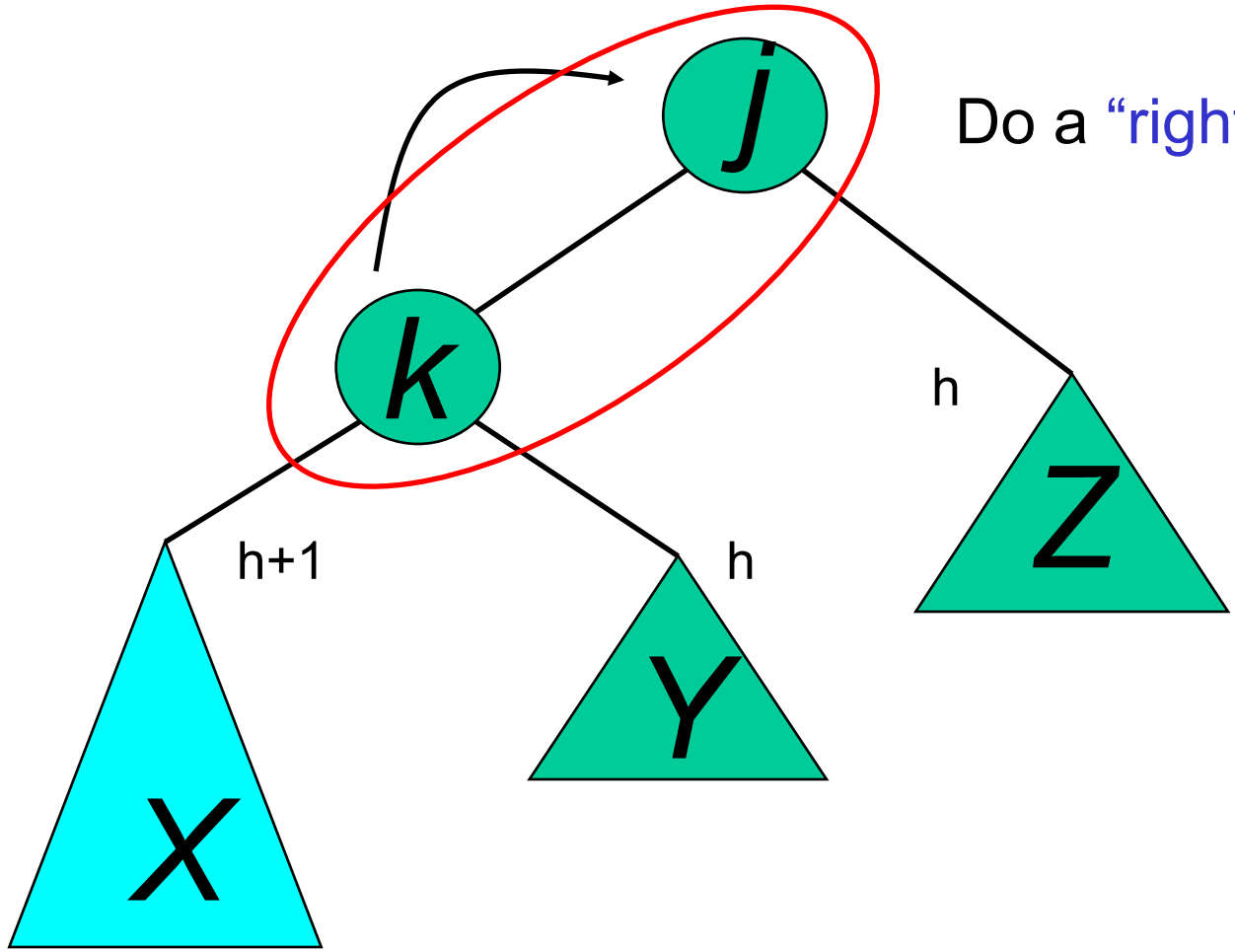


AVL Insertion: Outside Case



Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case



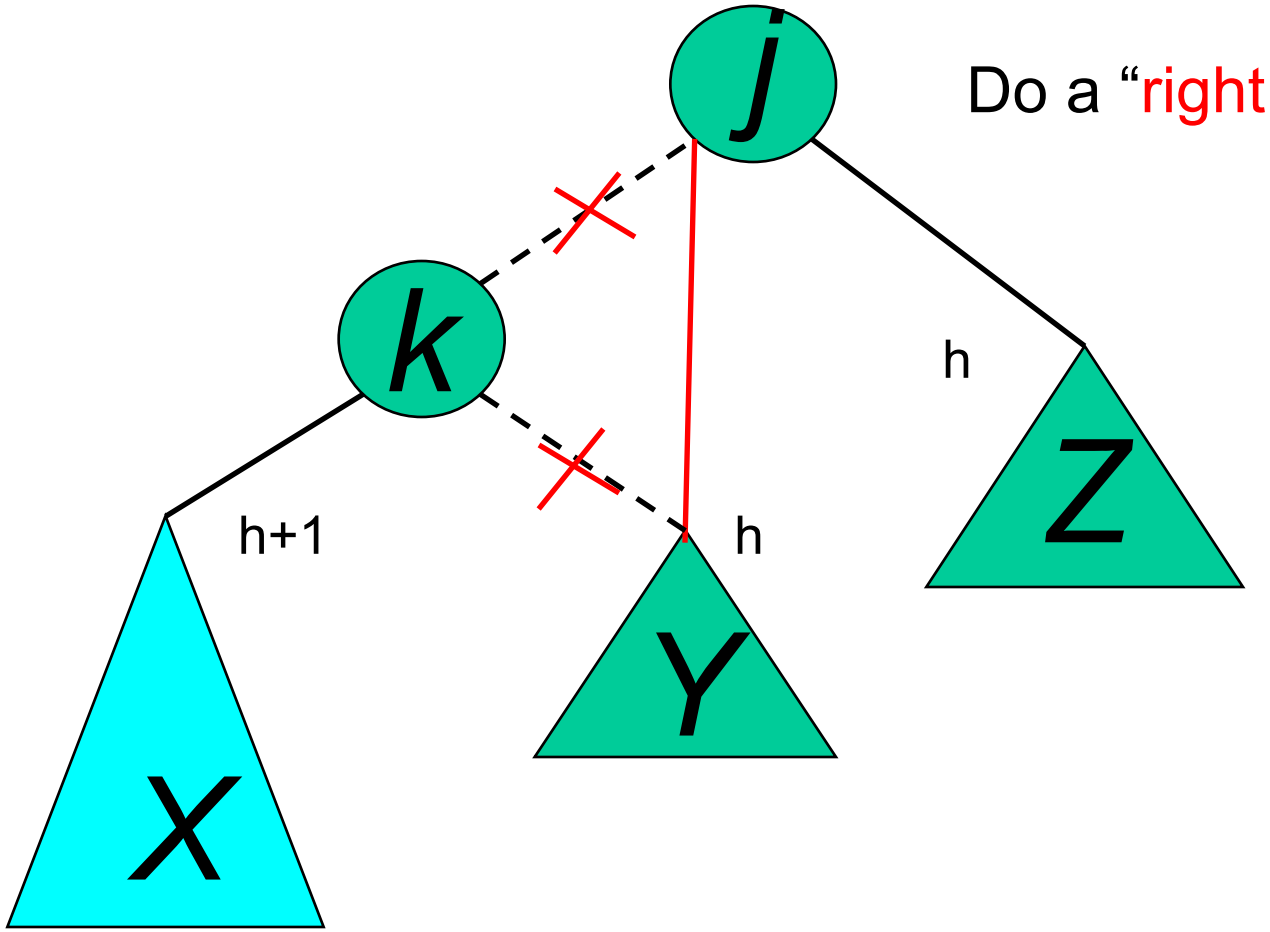
Do a "right rotation"

.....

.....

.....

Single right rotation



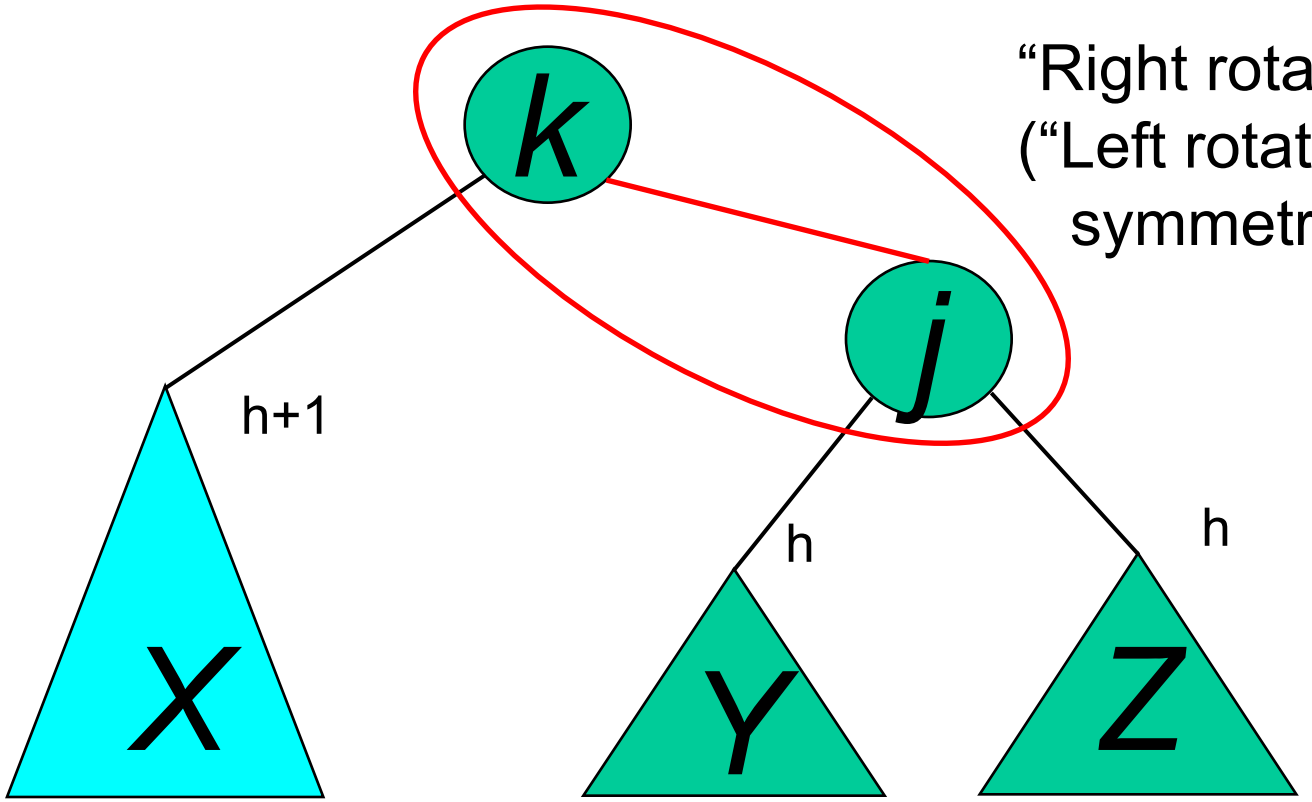
Do a "right rotation"

.....

.....

.....

Outside Case Completed

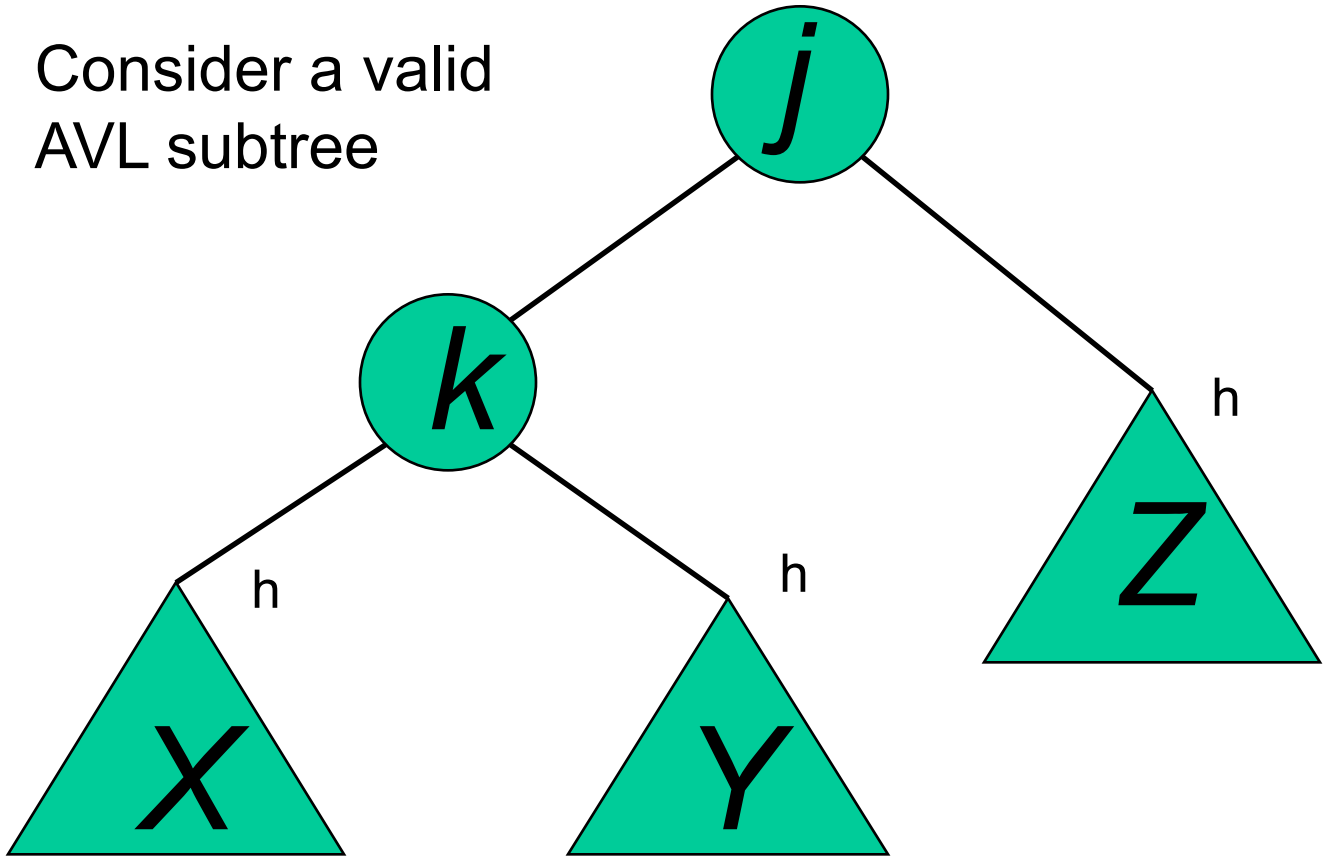


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

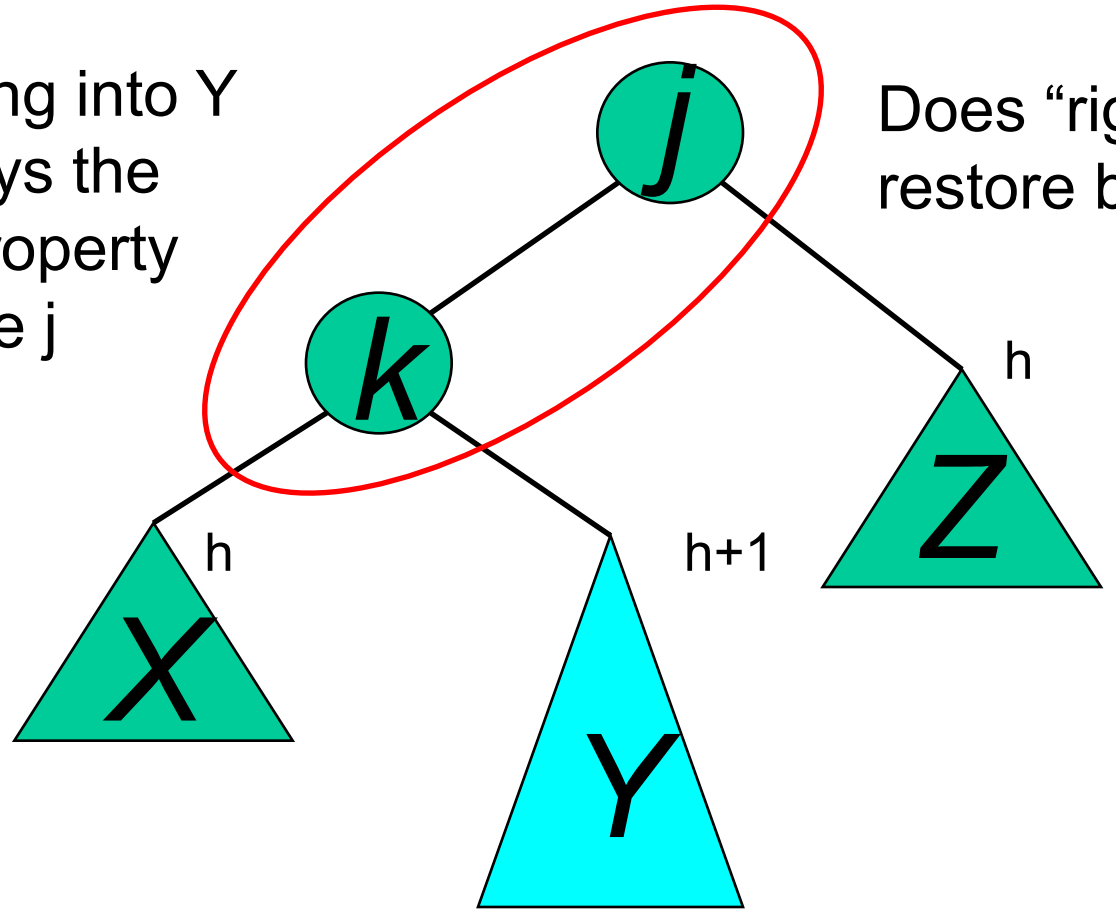
AVL Insertion: Inside Case

Consider a valid
AVL subtree



AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



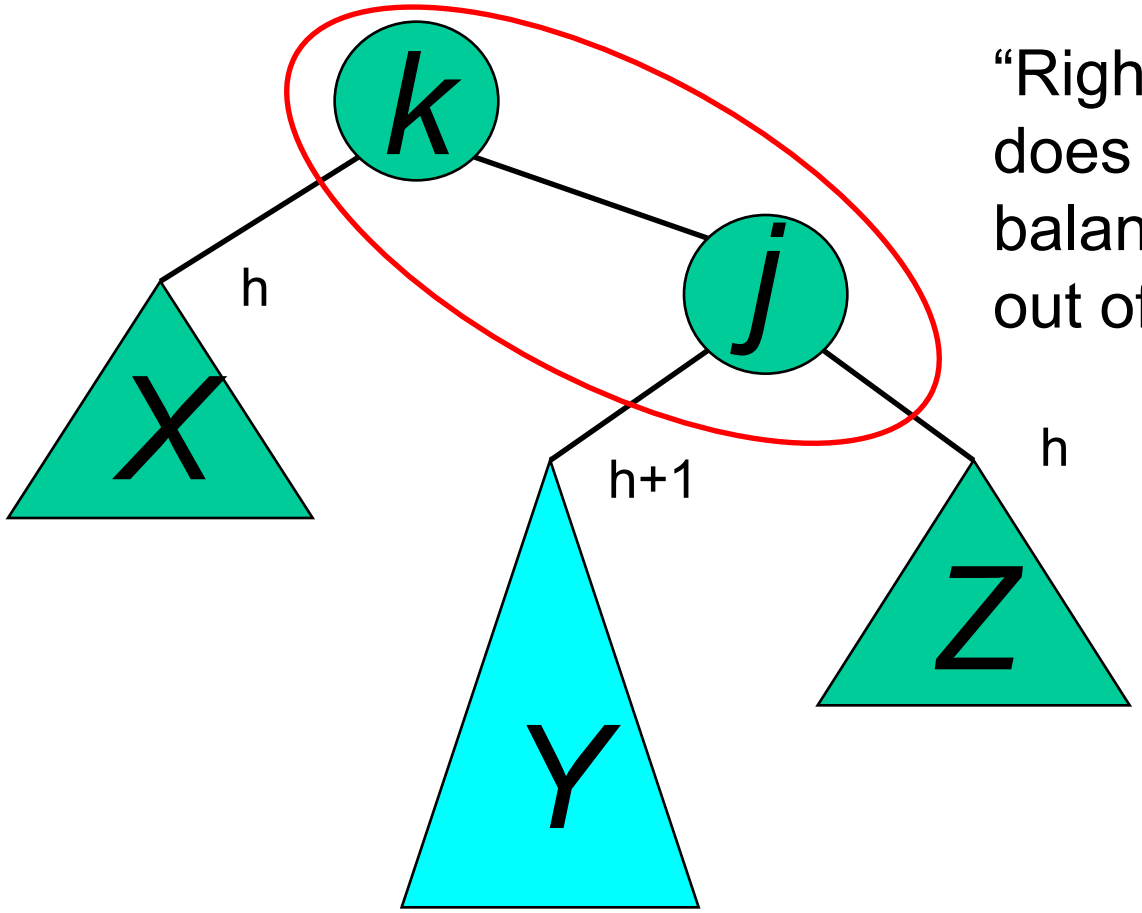
Does “right rotation” restore balance?

.....

.....

.....

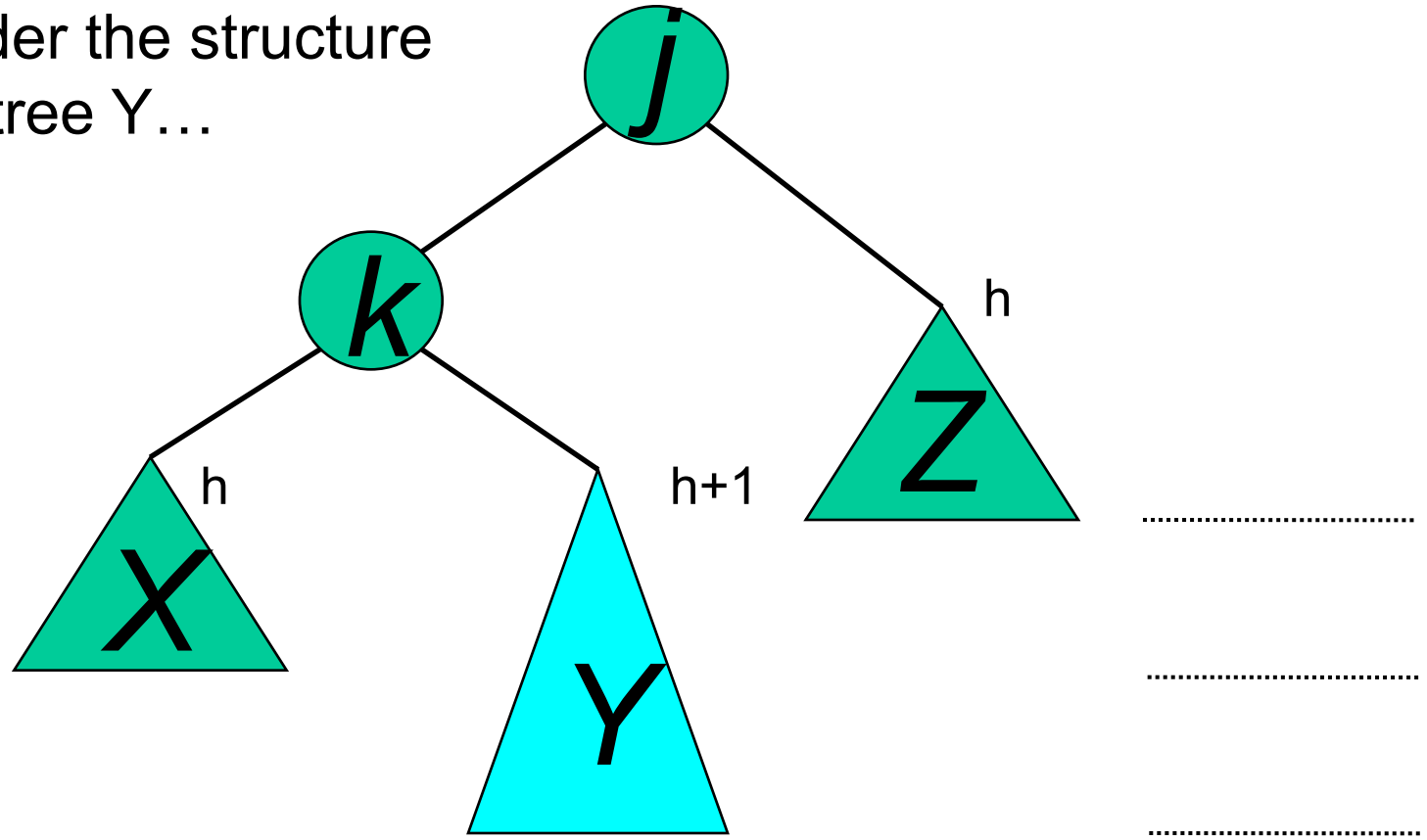
AVL Insertion: Inside Case



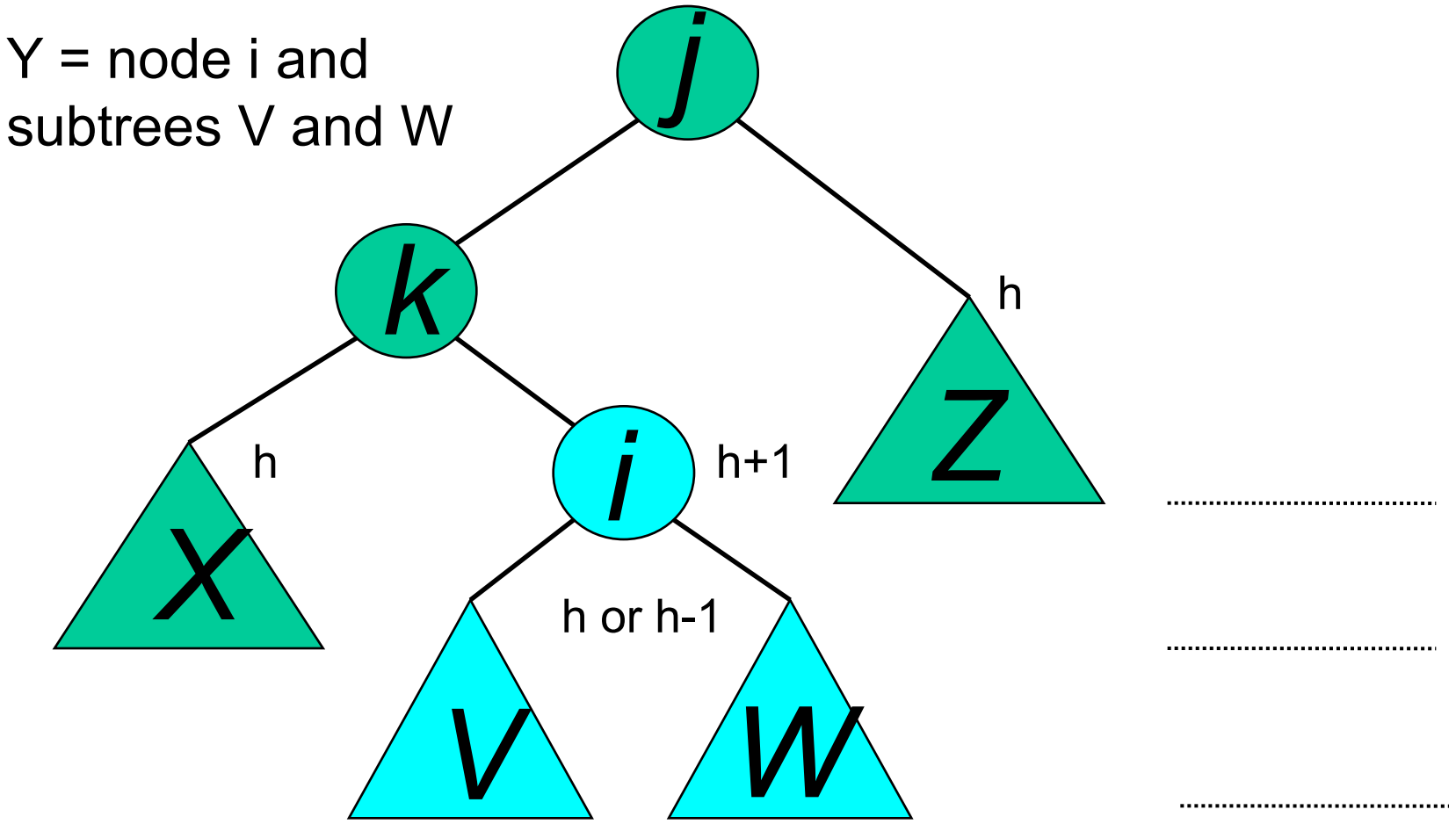
"Right rotation" does not restore balance… now k is out of balance

AVL Insertion: Inside Case

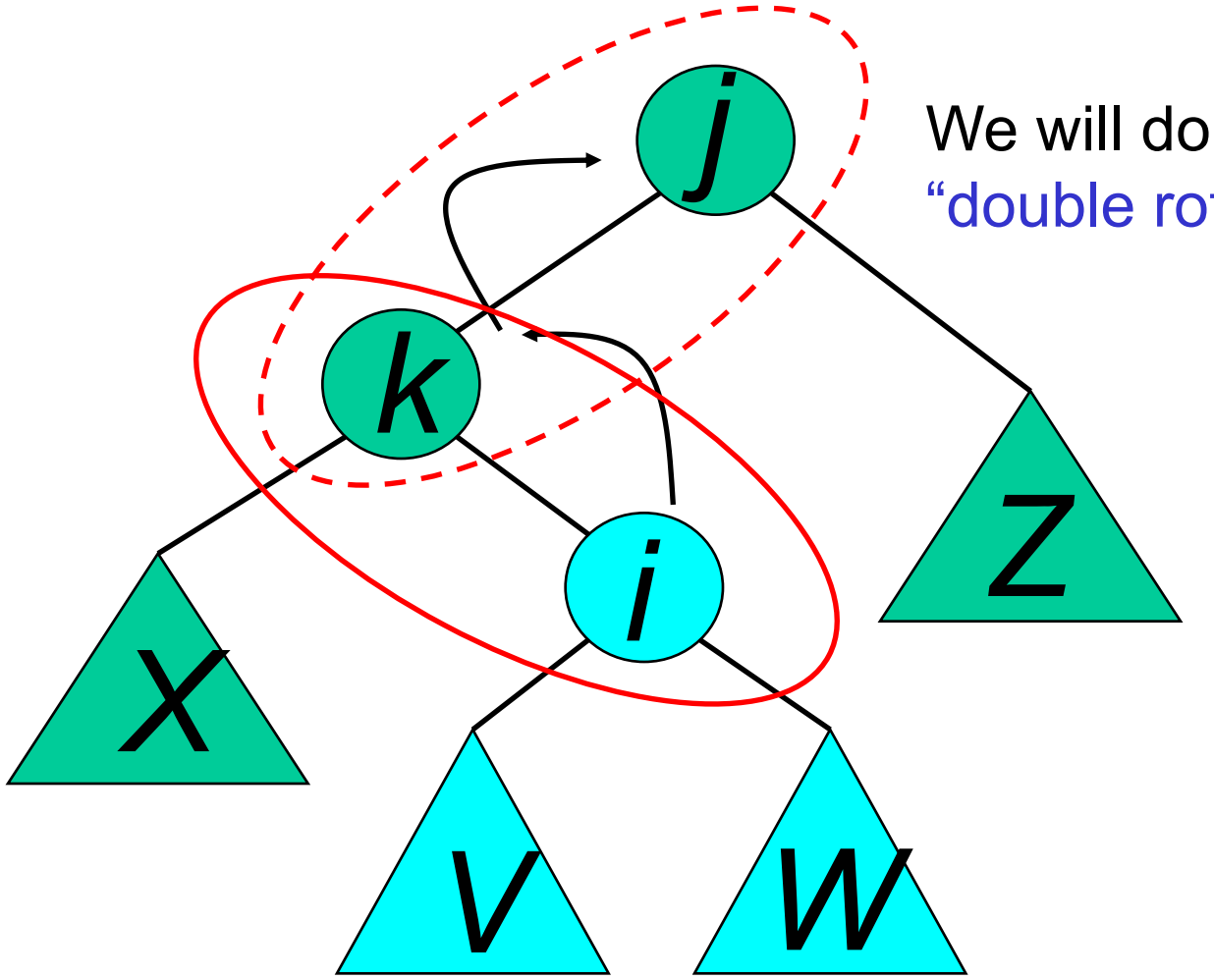
Consider the structure of subtree Y…



AVL Insertion: Inside Case

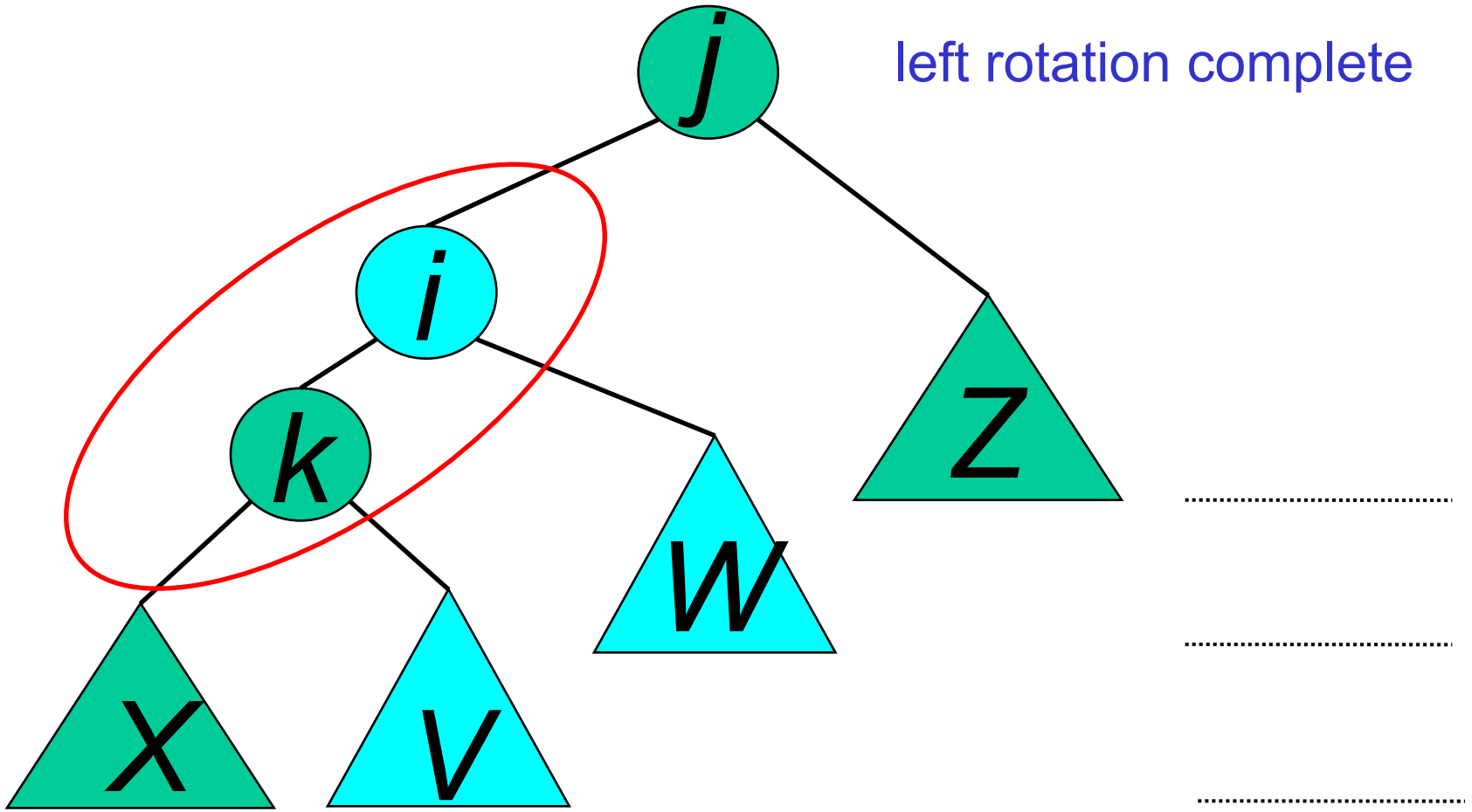


AVL Insertion: Inside Case

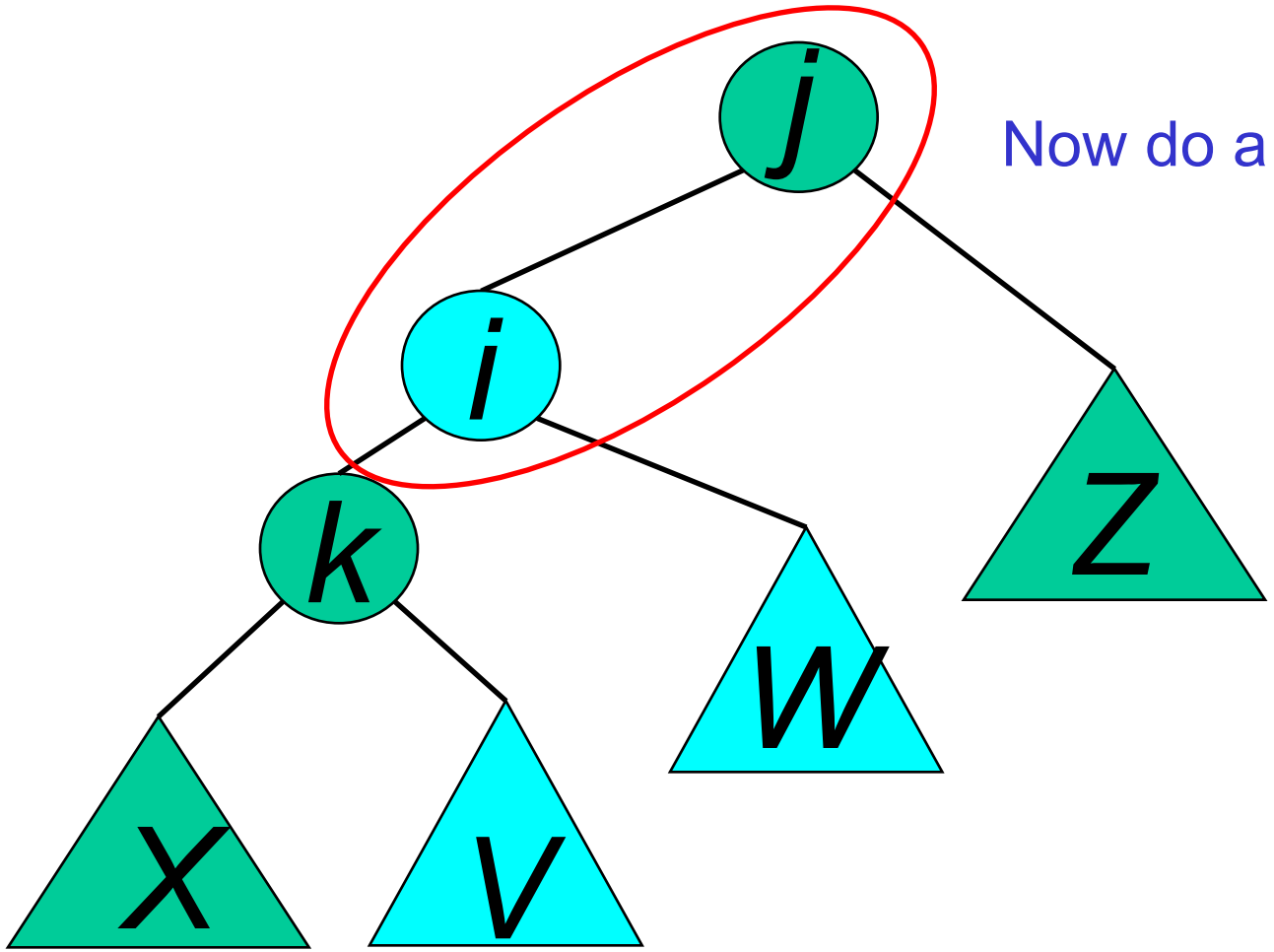


We will do a left-right "double rotation" . . .

Double rotation : first rotation



Double rotation : second rotation

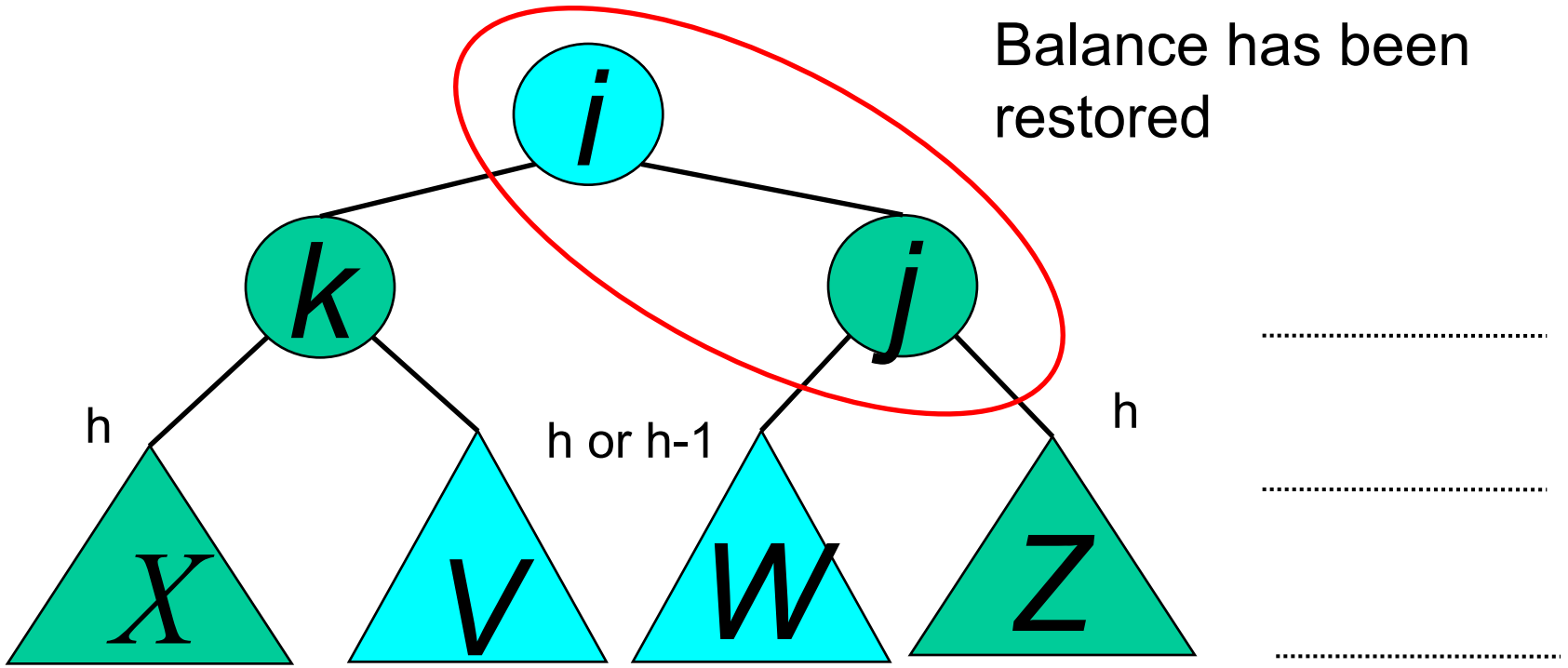


Now do a right rotation

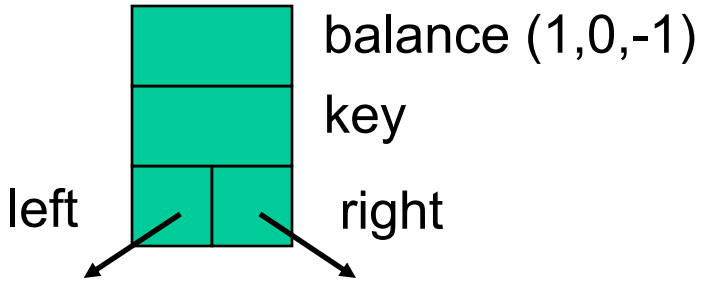
Double rotation : second rotation

right rotation complete

Balance has been restored



Implementation



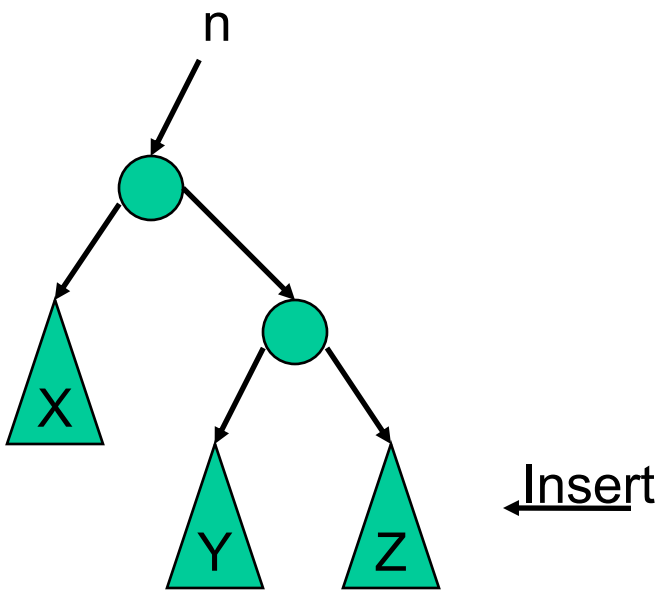
No need to keep the height; just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations

Once you have performed a rotation (single or double) you won't need to go back up the tree

Single Rotation

```
RotateFromRight(n : reference node pointer) {
  p : node pointer;
  p := n.right;
  n.right := p.left;
  p.left := n;
  n := p
}
```

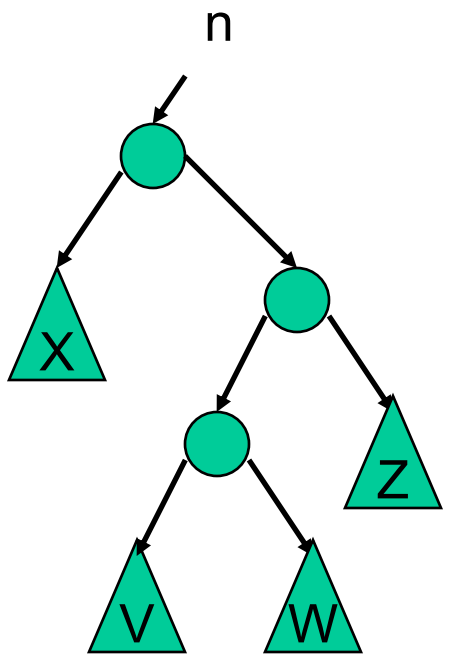
You also need to m
odify the heights or
balance factors of
n and p



Double Rotation

- Implement Double Rotation in two lines.

```
DoubleRotateFromRight(n : reference node pointer) {
  ????
}
```



Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

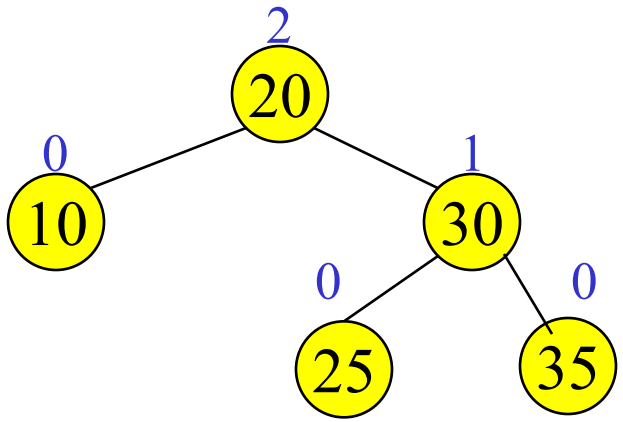
Insert in BST

```
Insert(T : reference tree pointer, x : element) : integer {
  if T = null then
    T := new tree; T.data := x; return 1;
    //the links to children are null
  case
    T.data = x : return 0; //Duplicate do nothing
    T.data > x : return Insert(T.left, x);
    T.data < x : return Insert(T.right, x);
  endcase
}
```

Insert in AVL trees

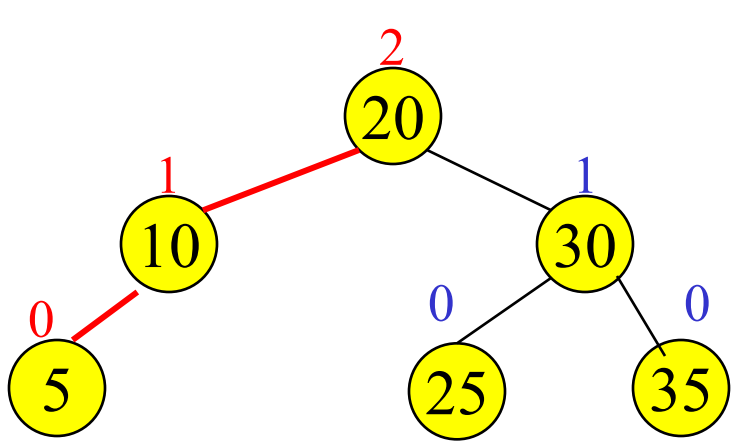
```
Insert(T : reference tree pointer, x : element) : {
  if T = null then
    {T := new tree; T.data := x; height := 0; return;}
  case
    T.data = x : return ; //Duplicate do nothing
    T.data > x : Insert(T.left, x);
                 if ((height(T.left)- height(T.right)) = 2){
                    if (T.left.data > x) then //outside case
                           T = RotatefromLeft(T);
                    else                          //inside case
                           T = DoubleRotatefromLeft(T);}
    T.data < x :  Insert(T.right, x);
                 code similar to the left case
  endcase
  T.height := max(height(T.left),height(T.right)) +1;
  return;
}
```

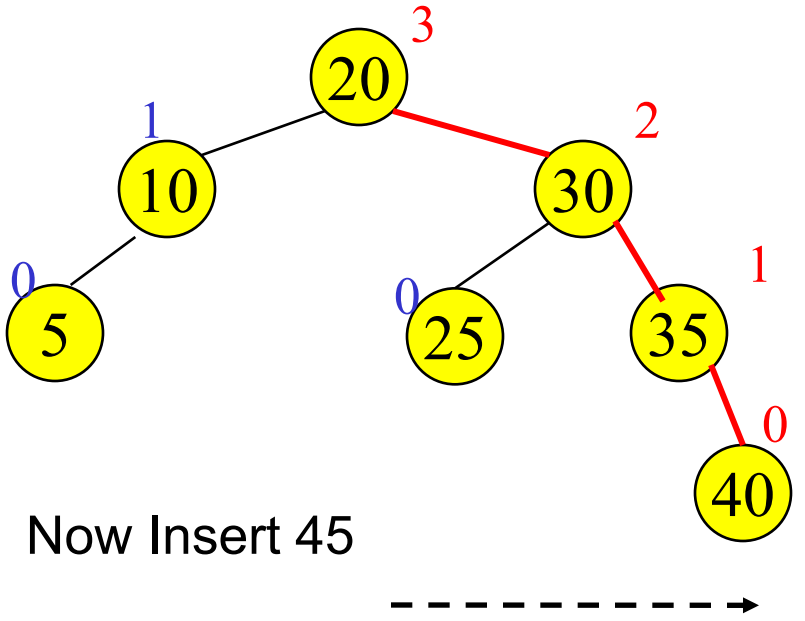
Example of Insertions in an AVL Tree



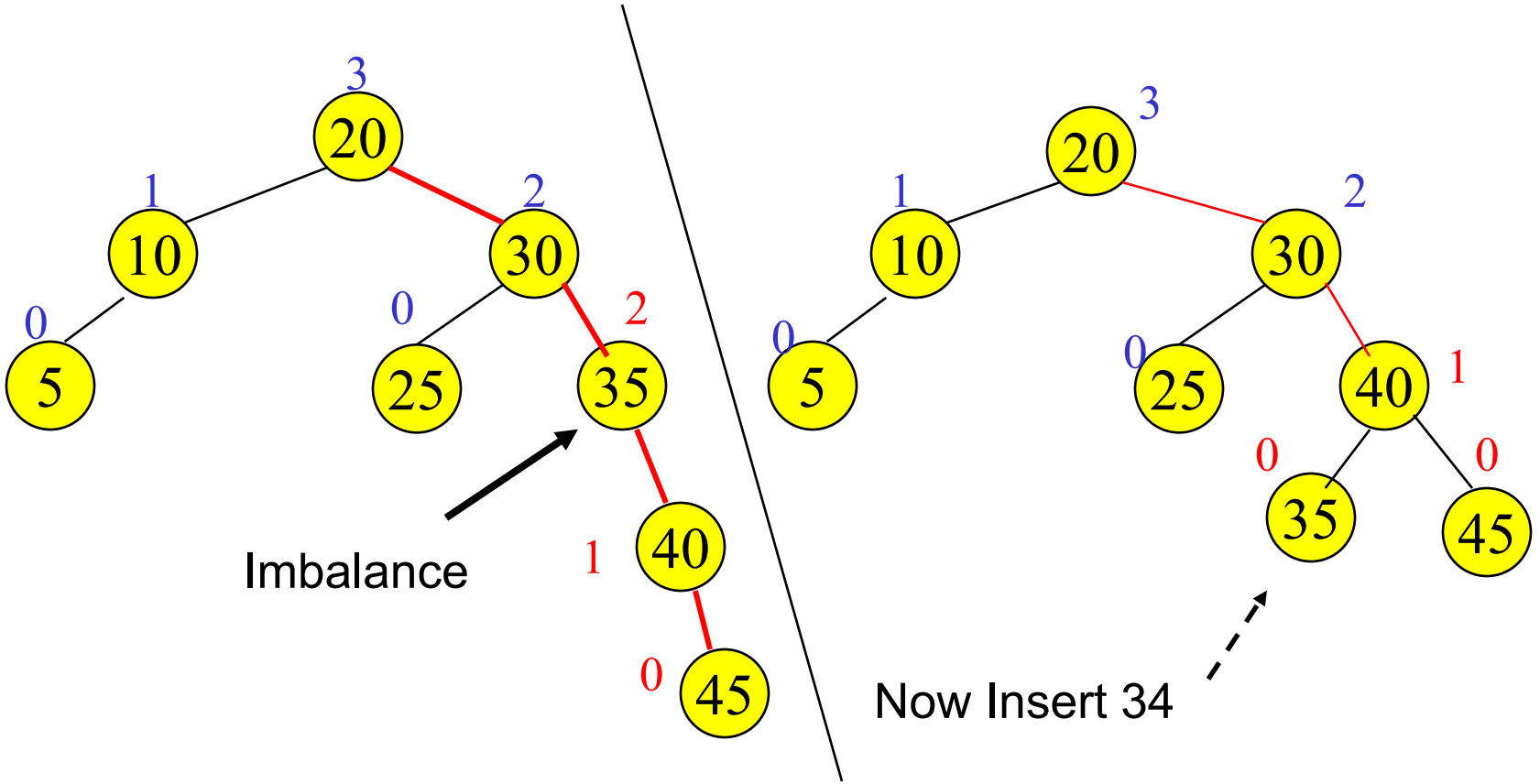
Insert 5, 40

Example of Insertions in an AVL Tree

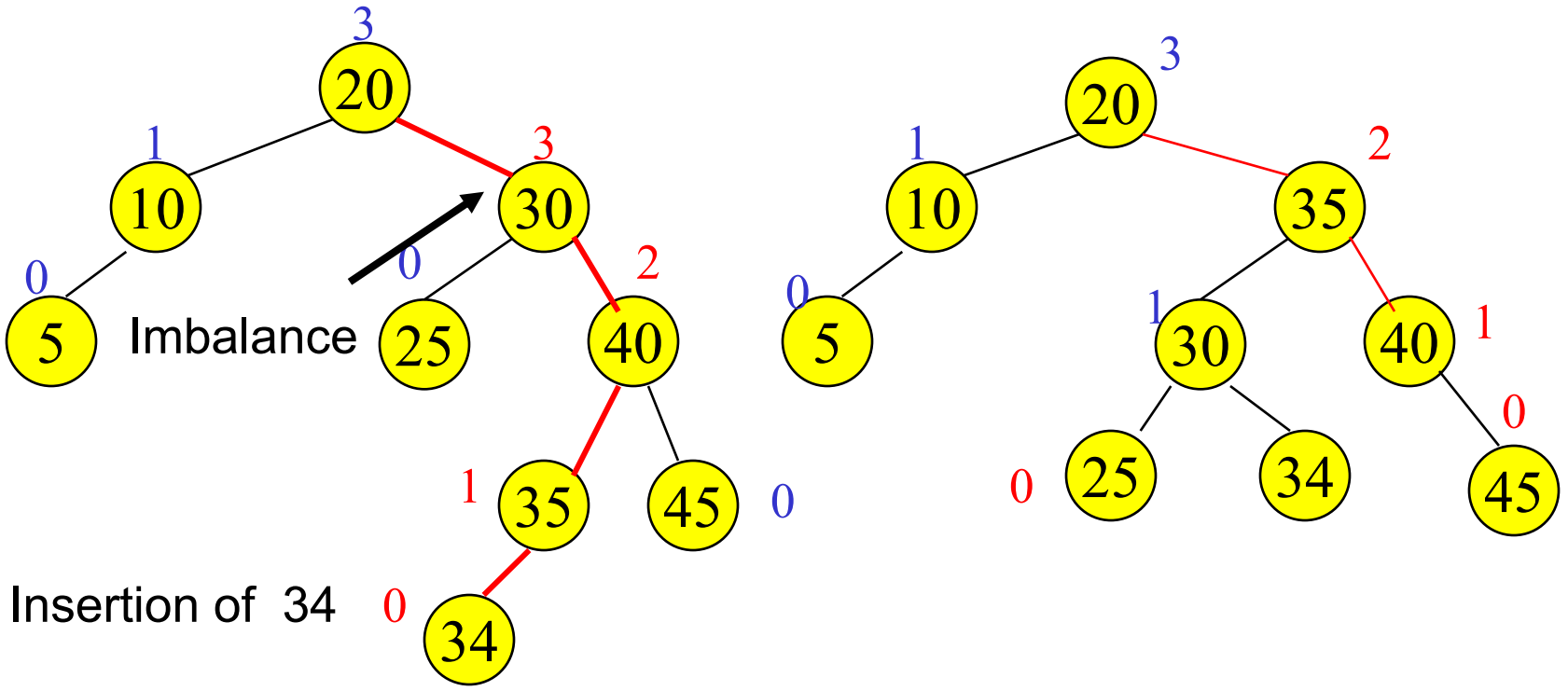




Single rotation (outside case)



Double rotation (inside case)



AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

Pros and Cons of AVL Trees

Arguments for AVL trees:

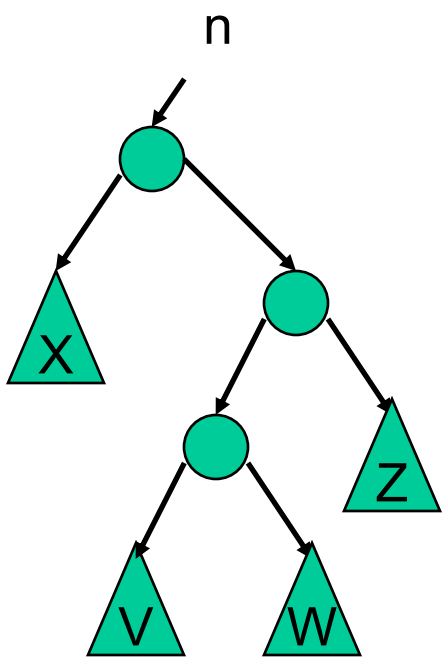
1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of in sertion.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other st ructures (e.g. B-trees).
4. May be OK to have O(N) for a single operation if total run time for many con secutive operations is fast (e.g. Splay trees).

Double Rotation Solution

```
DoubleRotateFromRight(n : reference node pointer) {
  RotateFromLeft(n.right);
  RotateFromRight(n);
}
```



Advantages / Disadvantages of AVL Trees

AVL Trees	
+	-
Guaranteed $\mathcal{O}(\log_\phi n)$ performance of all operations.	Spend **a lot of time** *(almost after every operation!)* on rotations, which are themselves expensive operations
Easy to add balancing functionality to existing BST code (even by overriding)	Need to store at least 4 bits per node to preserve balance information (plus the unit cost of updating it)
• • •	Complex implementation (deletions in particular)
• • •	

Biggest disadvantages of AVL Trees

Maintain the pros, kill the cons

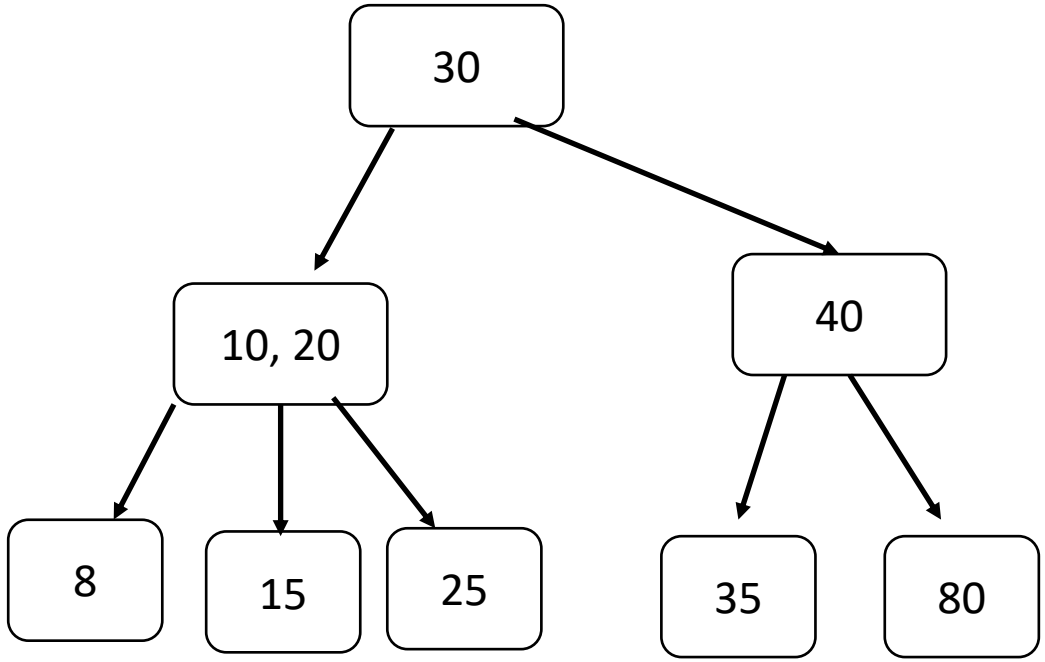
- Is there any way we can maintain mission-critical logarithmic worst-case complexity for search, while simultaneously shedding the storage cost and rotation overhead of AVL Trees?
- Yup! Two solutions:
  - 2-3 trees implemented via Red-Black Binary Search Trees (RBBSTs)
  - B-Trees (BTs)
- Today, we examine the simplest B-tree, known as a 2-3 tree

2-3 Trees

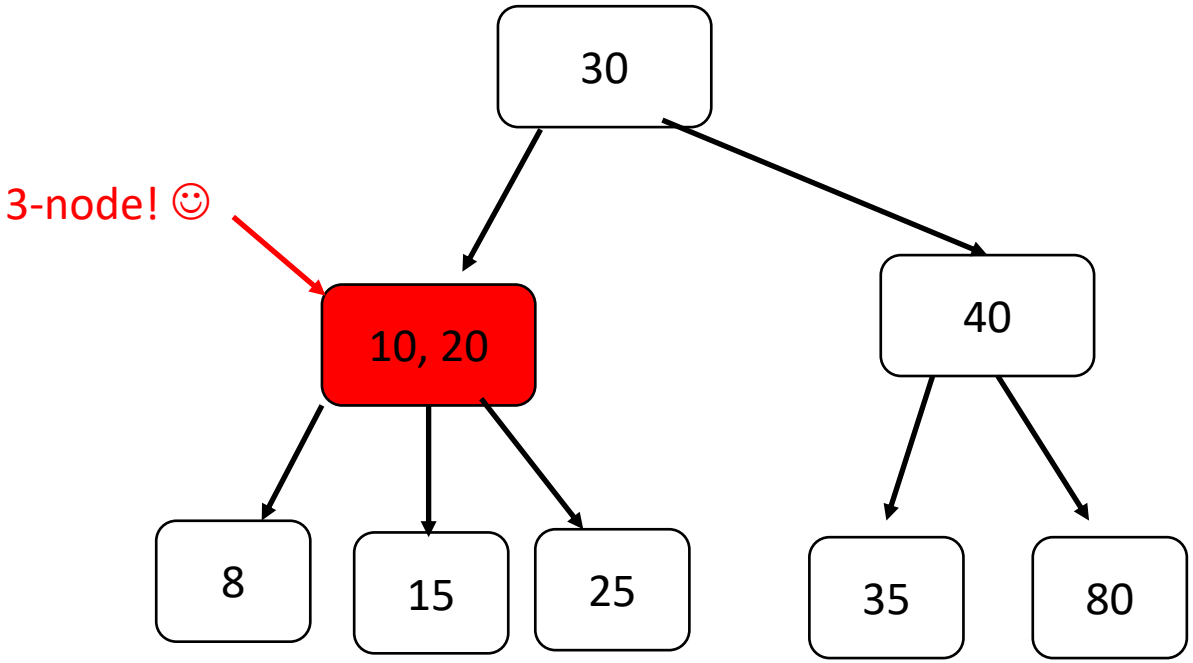
2-3 Trees

- In addition to their other properties, 2-3 trees (and B-trees) will have **perfect balance** all the time!
- Core idea: 2 children per node are ok, more are better (for height).
- 2-3 trees will have two different nodes inside them!
  a) A **BST-like node** with just one key, and two pointers to subtrees **(a 2- node)**
  b) An expanded node with three keys and three pointers to subtrees **(a 3-node)**
    - Leftmost subtree contains keys < Key_1
    - Center subtree contains keys in $[Key_1, Key_2)$
    - Rightmost subtree contains keys $> Key_2$

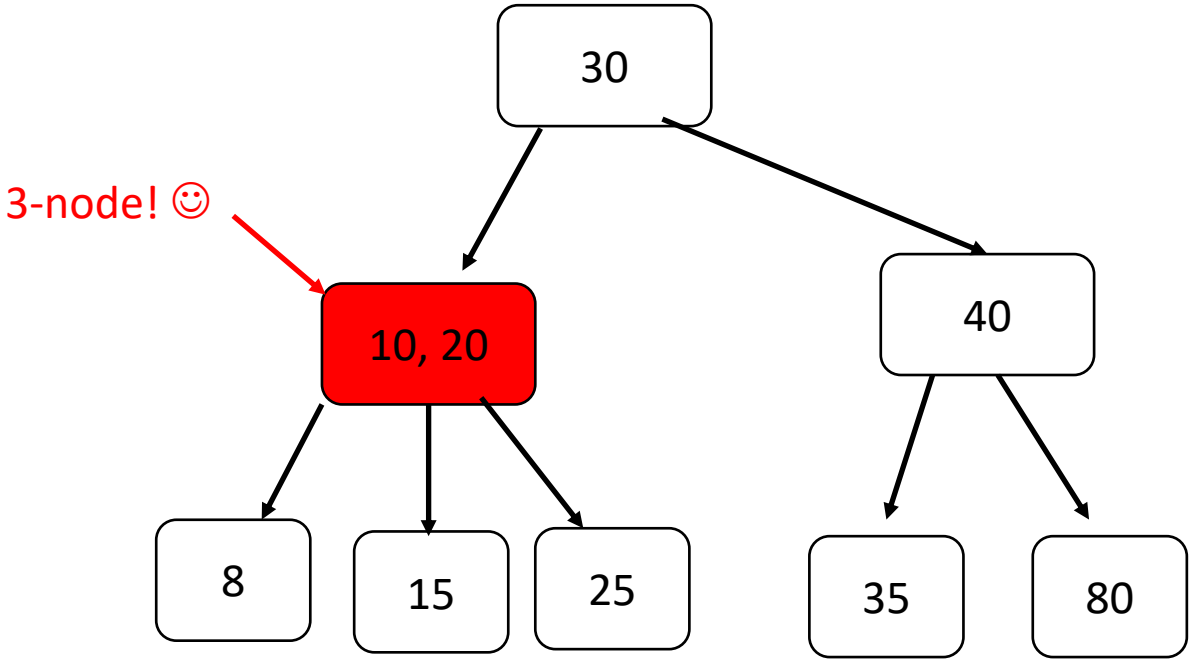
2-3 Tree example



2-3 Tree example



2-3 Tree example



The **minimum** height of a 2-3 tree with n keys is…

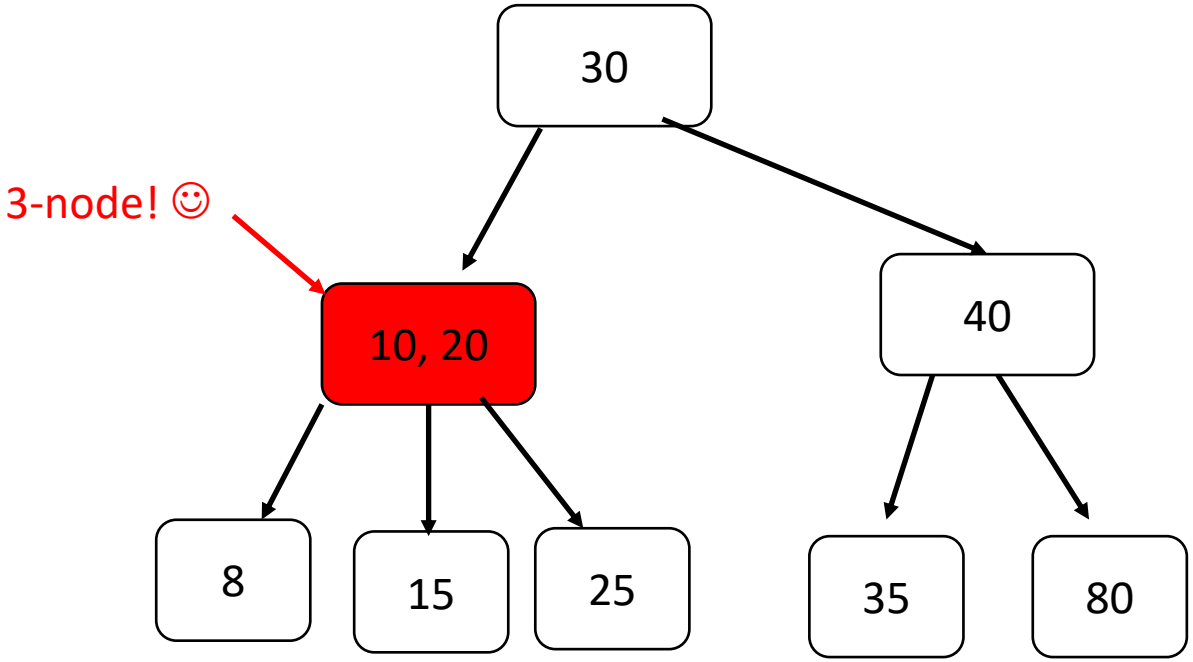
$\lfloor \log_2 n \rfloor$

$\lfloor \log_2 n \rfloor + 1$

$\lceil \log_2 n \rceil$

$\log_3 n$

2-3 Tree example



All 3-nodes!

The **minimum** height of a 2-3 tree with n keys is…

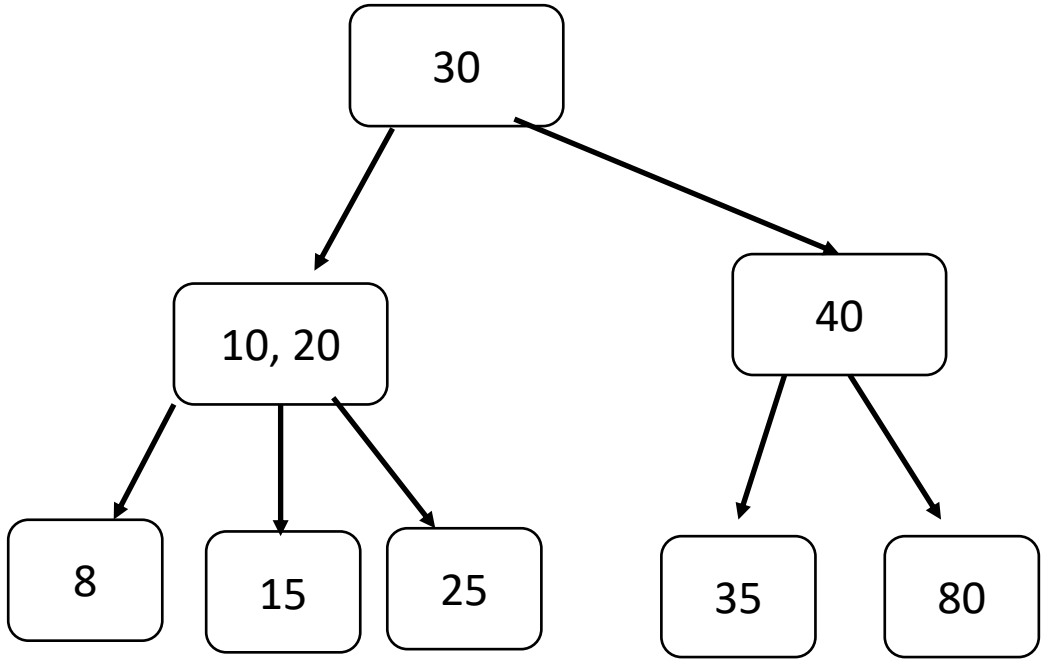
$\lfloor \log_2 n \rfloor$

$\lfloor \log_2 n \rfloor + 1$

$\lceil \log_2 n \rceil$

$\lfloor \log_3 n \rfloor$

Search



- Searching a 2-3-Tree is done in exactly the same way as a BST.
- Stronger efficiency guarantees in place since the height is between $\log_3 n$ and $\log_2 n$! ☺

Insertion

- Let's recall insertion in AVL …
  - **AVL: *Might*** trigger rotations to re-balance the tree according to the AVL condition.
- This **will leave the tree unbalanced** according to the strict most criterion of… same height everywhere!
  - Only exception: AVL insertion places node in the only empty position at the leaves of an AVL Tree.

Insertion

- Insertion on both of these tree has two phases…
  1. A search phase, where we go down to search for an empty position to put the key in (if it's not already in there), and
  2. A rotation phase where we go up to either rebalance subtrees (AVL)…

Insertion

- Insertion on both of these trees has two phases…
  1. A search phase, where we go down to search for an empty position to put the key in (if it's not already in there), and
  2. A rotation phase where we go up to either rebalance subtrees (AVL) …
- B-Trees have the same search phase, BUT! in order to make sure the entire tree is perfectly balanced, the second phase will rotate exactly zero nodes!

Insertion

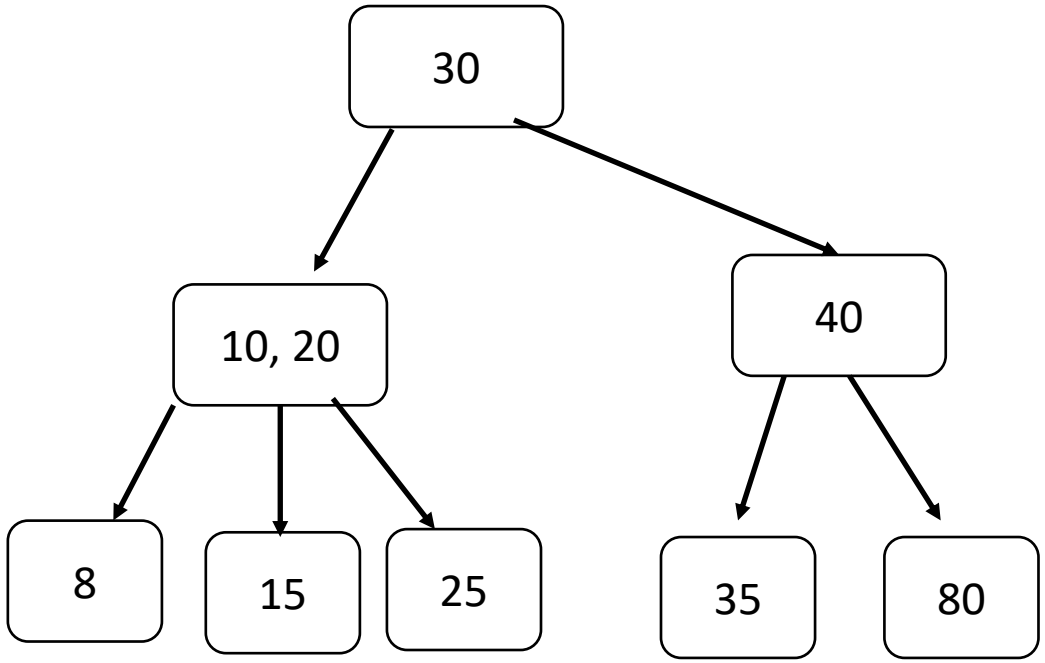
- Insertion on both of these trees has two phases…
  1. A search phase, where we go down to search for an empty position to put the key in (if it's not already in there), and
  2. A rotation phase where we go up to either rebalance subtrees (AVL)…
- B-Trees have the same search phase, BUT! in order to make sure the entire tree is perfectly balanced, the second phase will rotate exactly zero nodes!
- Instead, it *might* either make nodes fatter or blow them up into two nodes.

Insertion

- Insertion on both of these trees has two phases…
  1. A search phase, where we go down to search for an empty position to put the key in (if it's not already in there), and
  2. A rotation phase where we go up to either rebalance subtrees (AVL)…
- B-Trees have the same search phase, BUT! in order to make sure the entire tree is perfectly balanced, the second phase will rotate exactly zero nodes!
- Instead, it *might* either make nodes fatter or blow them up into two nodes.
  - We'll see how this happens **right now**.

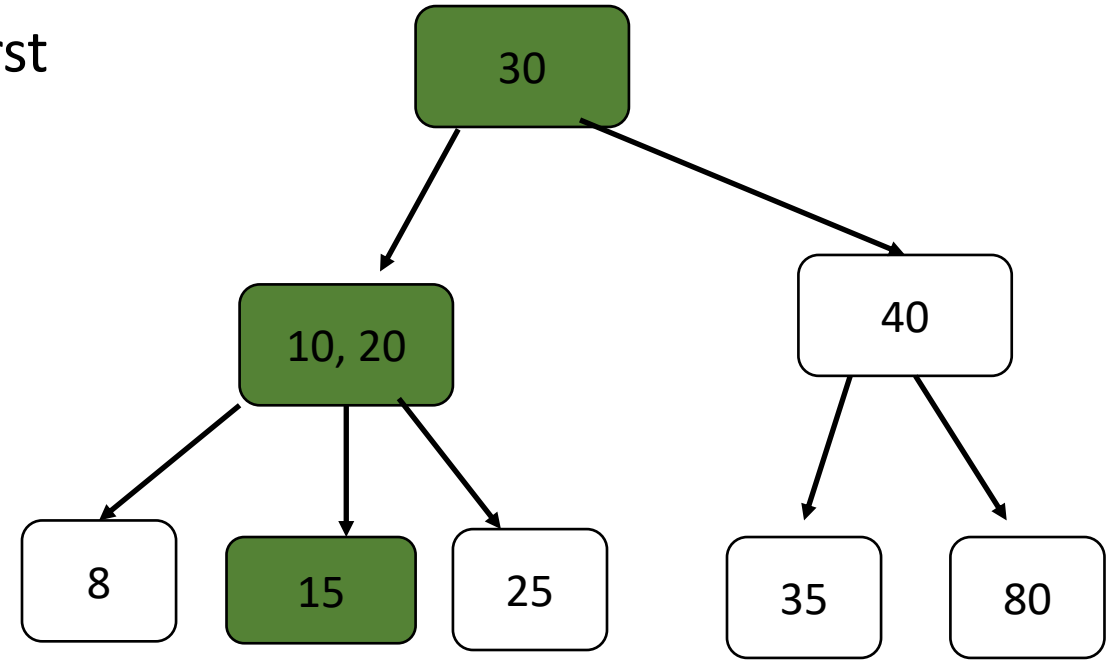
Insertion

- Insert 18



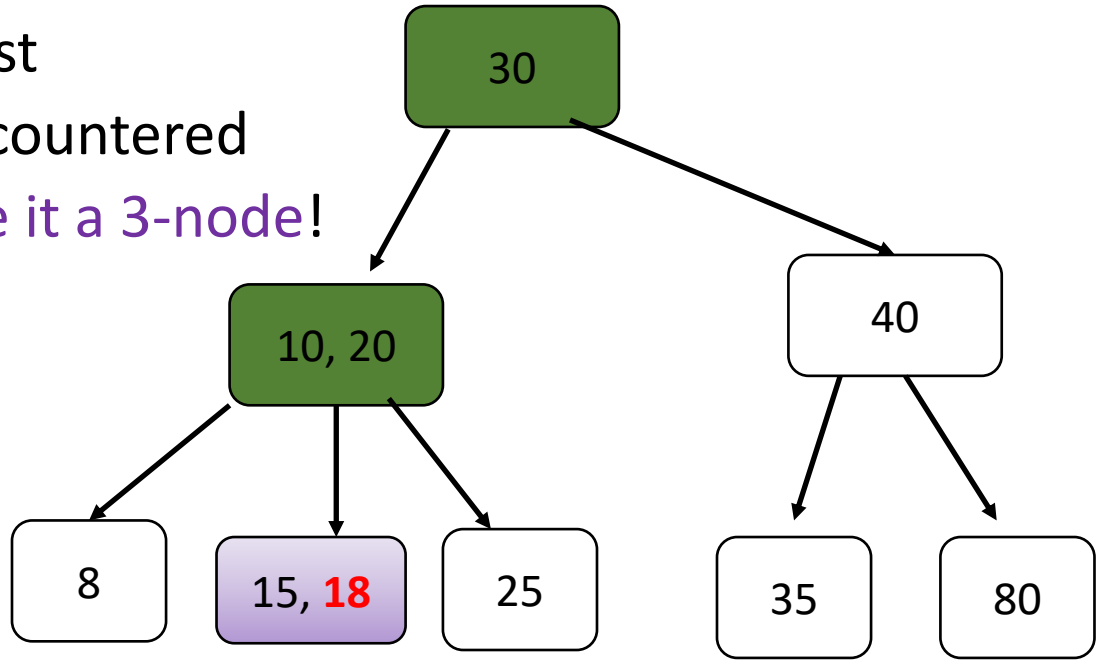
Insertion

- Insert 18
  1. We search first



Insertion

- Insert 18
  1. We search first
  2. Last node encountered
     is 2-node: make it a 3-node!

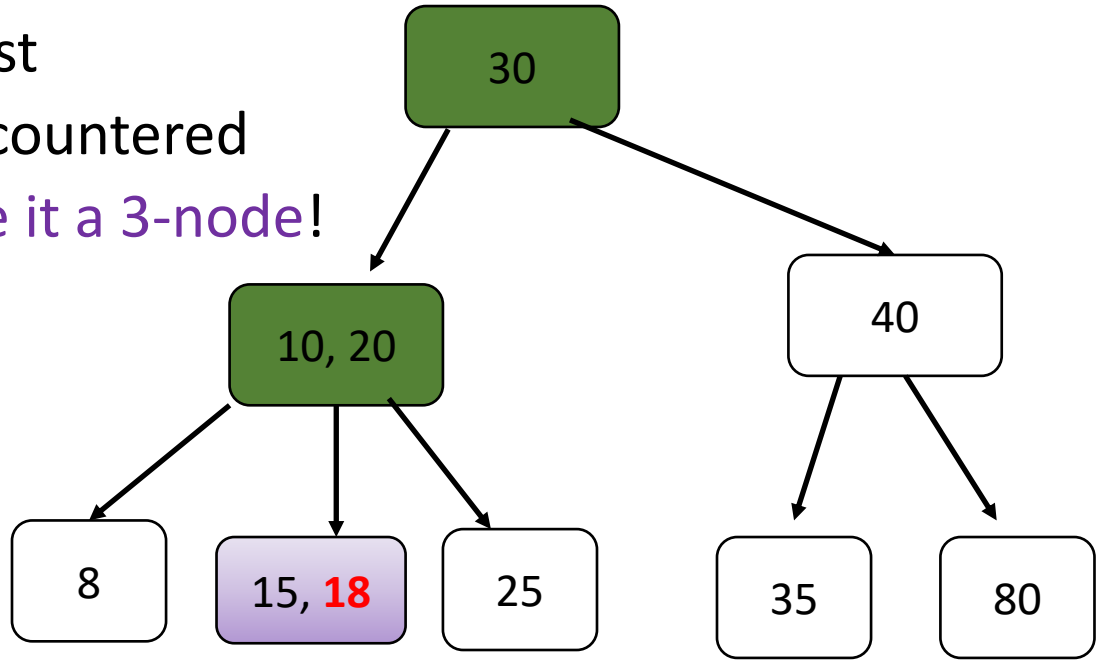


Insertion

- Insert 18
  1. We search first
  2. Last node encountered

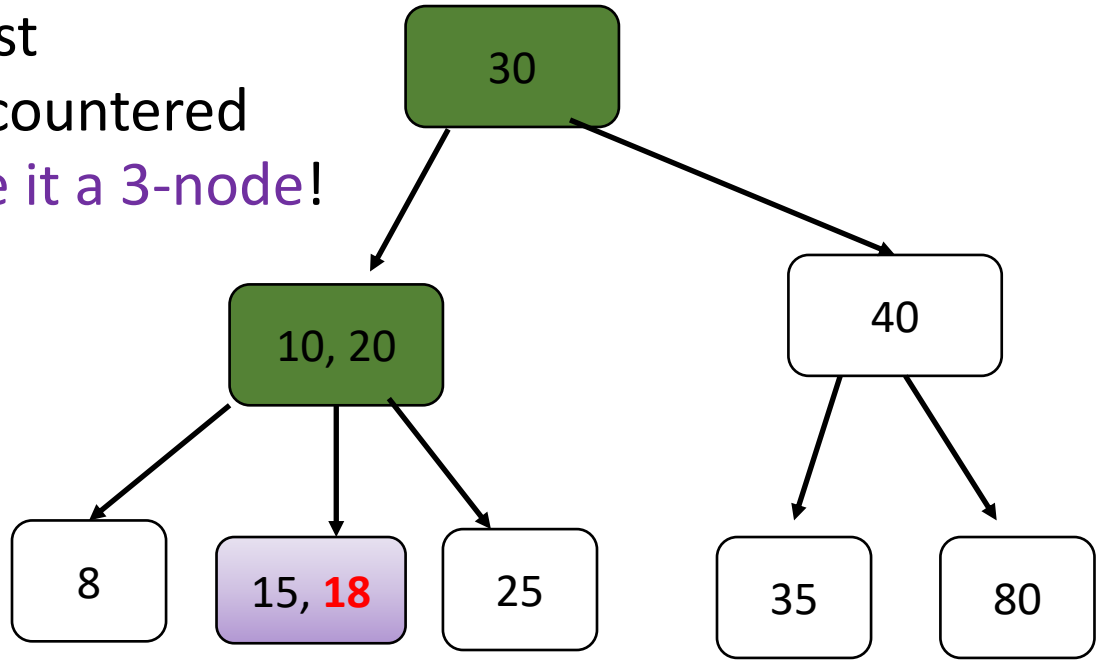
     is 2-node: make it a 3-node!

- Done! ☺



Insertion

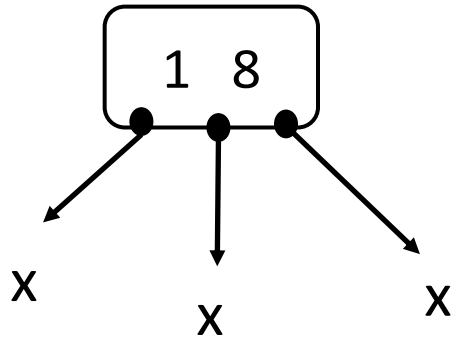
- Insert 18
  1. We search first
  2. Last node encountered is 2-node: make it a 3-node!



- Done! ☺
- But what if I now wanted to insert 17? I don't have space for it!

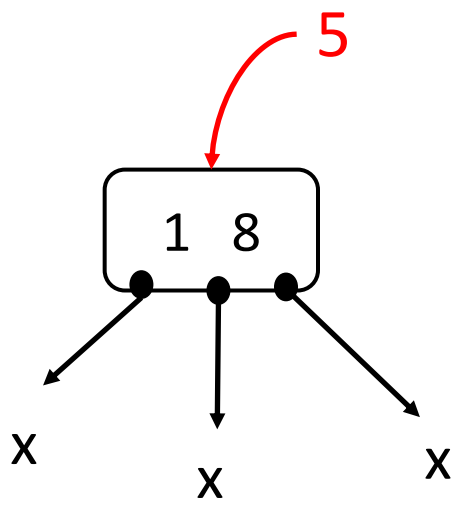
Insertion

- To understand what to do in this case, consider inserting 5 in this "stub" 2-3 tree.



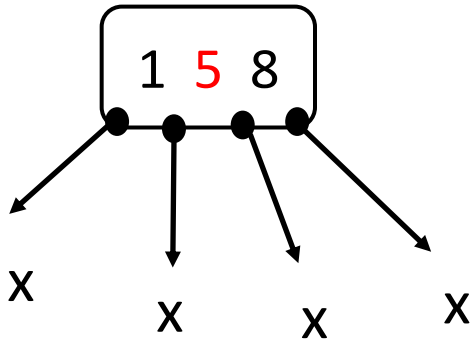
Insertion

- To understand what to do in this case, consider inserting 5 in this "stub" 2-3 tree.



Insertion

- To understand what to do in this case, consider inserting 5 in this "stub" 2-3 tree.

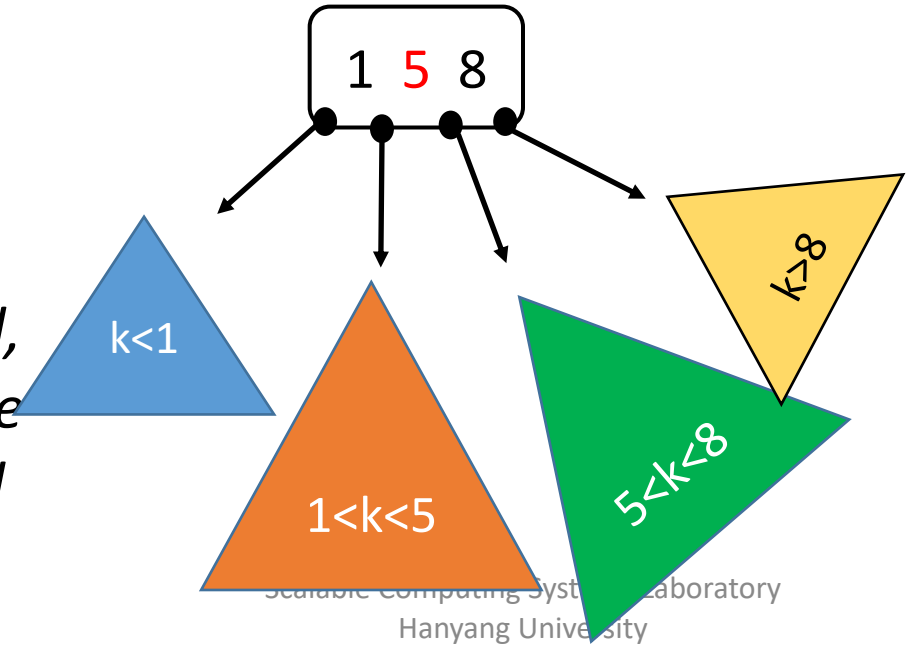


Temporarily assume we can expand the 3-node to a 4-node!

Insertion

- To understand what to do in this case, consider inserting 5 in this "stub" 2-3 tree.

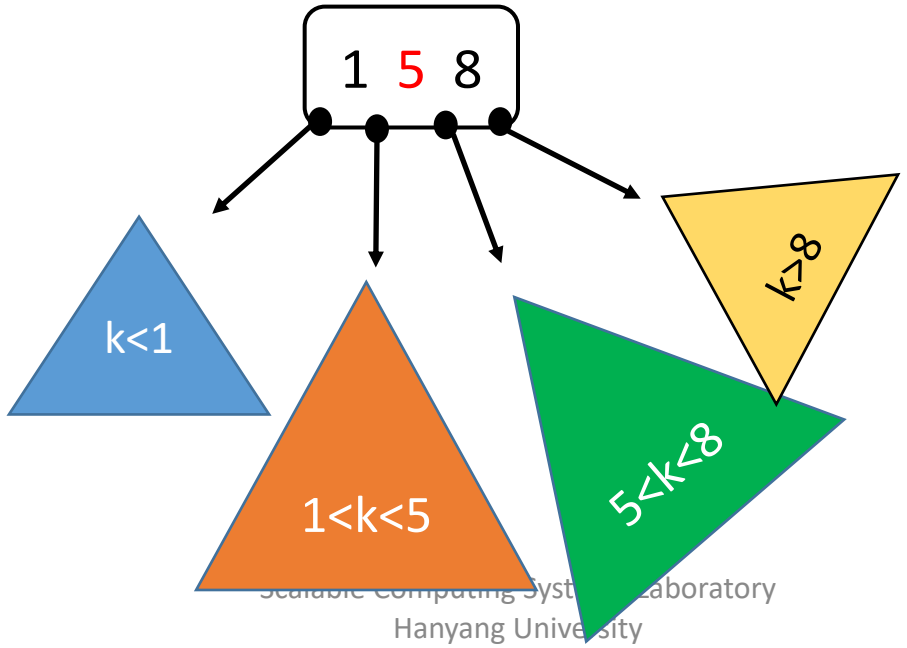
(if the links were non-null, those would be the subtrees / key ranges they would point to)



Temporarily assume we can expand the 3-node to a 4-node!

Insertion

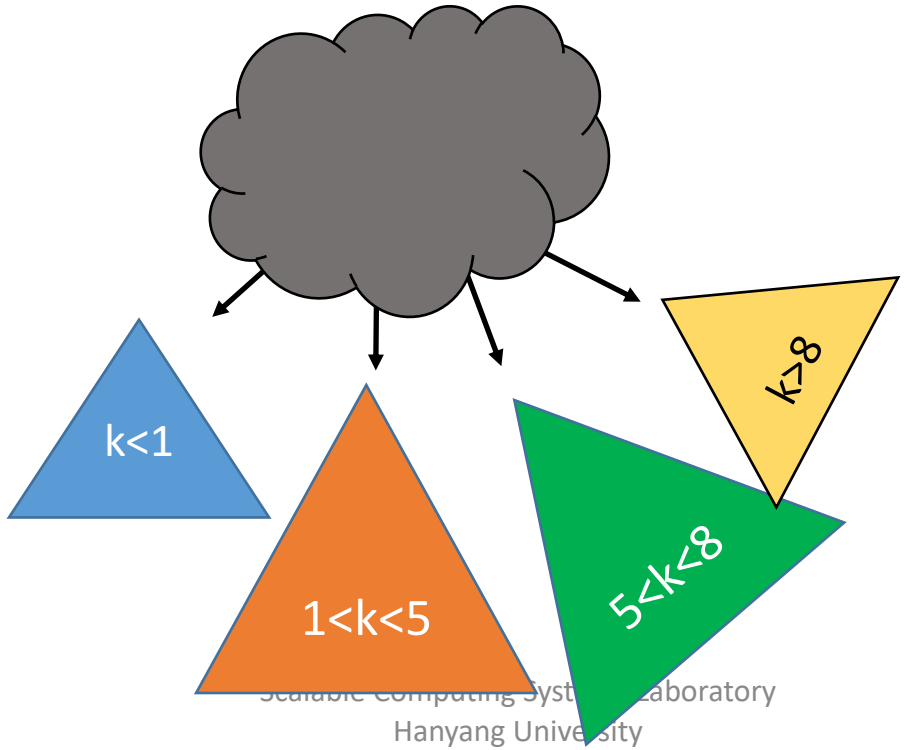
- To understand what to do in this case, consider inserting 5 in this "stub" 2-3 tree.



Of course, this is **unaccep table**. Our solution:

Insertion

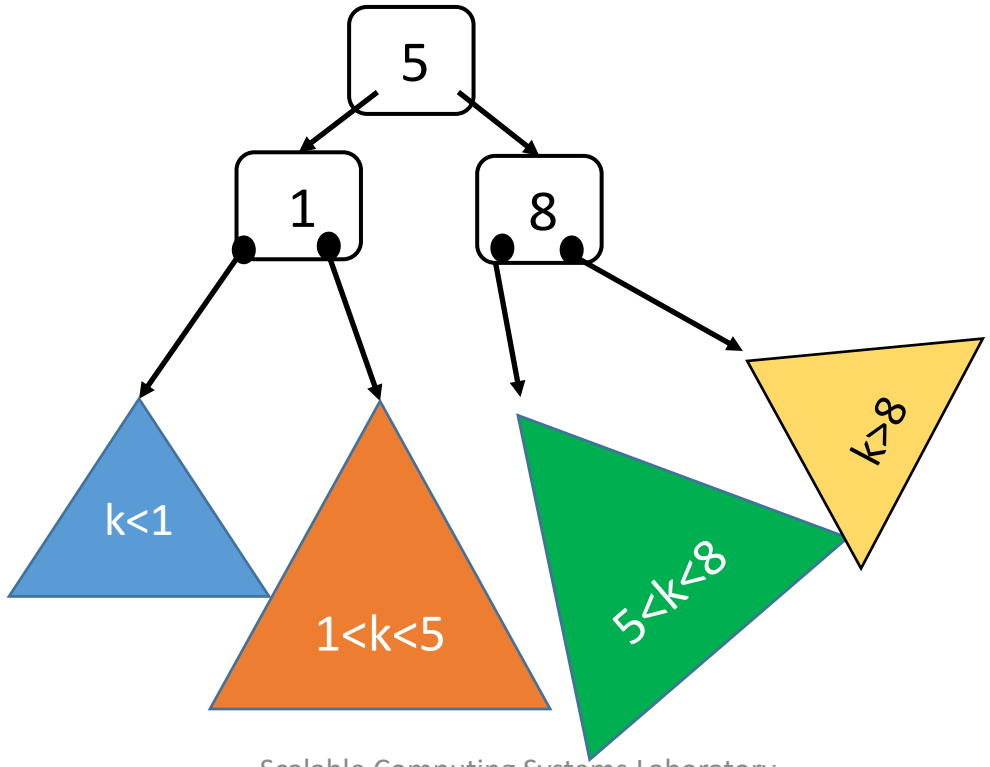
- To understand what to do in this case, consider inserting 5 in this "stub" 2-3 tree.



Of course, this is **unaccep table**. Our solution: **Blow the node apart!**

Insertion

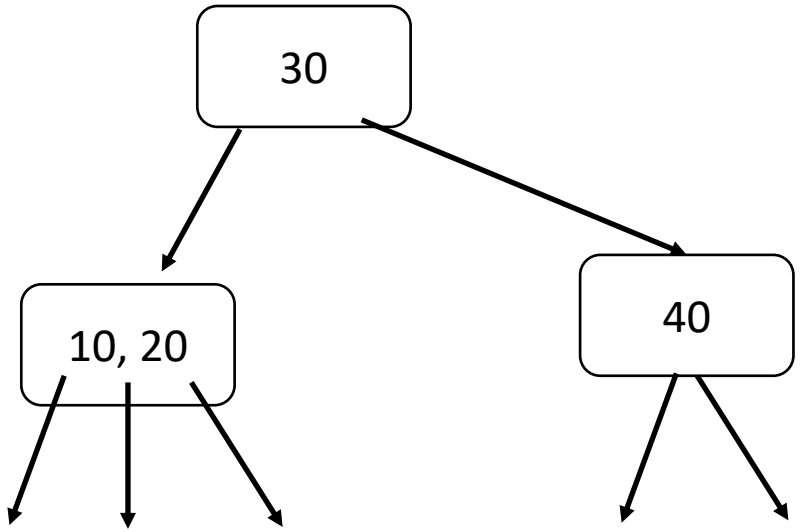
- To understand what to do in this case, consider inserting 5 in this "stub" 2-3 tree.



We are left with three 2-no des that point to the correct subtrees!

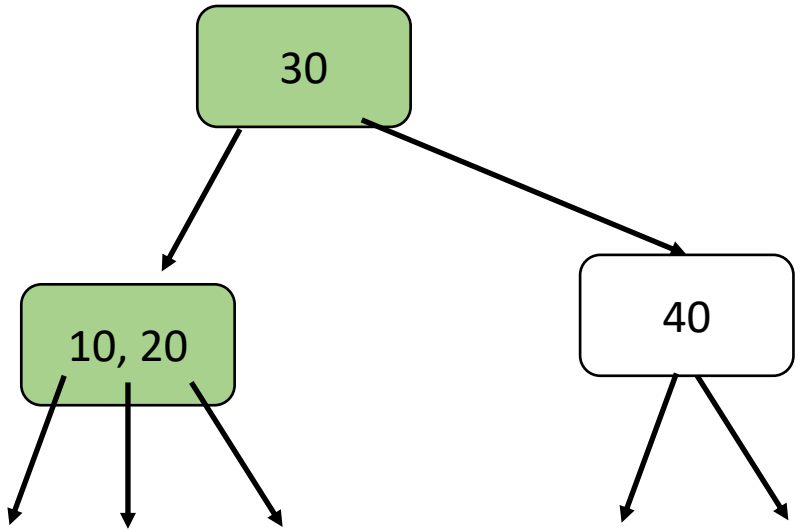
For you!

- Insert 15 in this 2-3 tree please! ☺



Walkthrough…

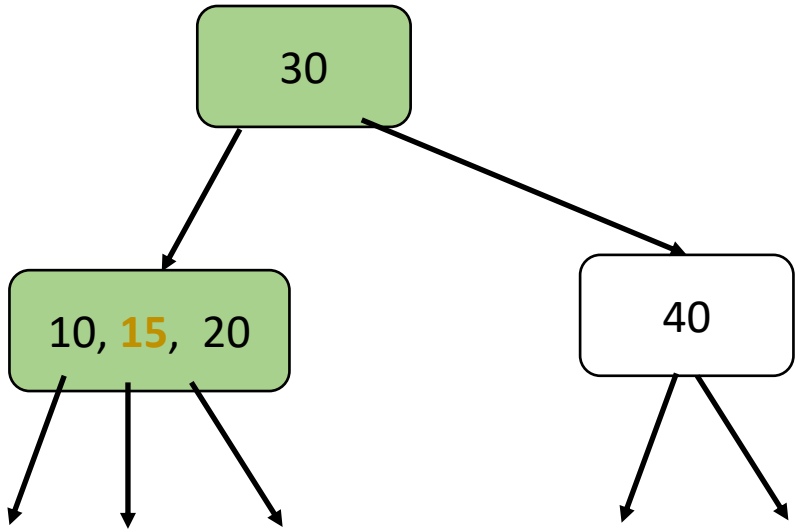
- Insert 15 in this 2-3 tree please! ☺



1. Search

Walkthrough…

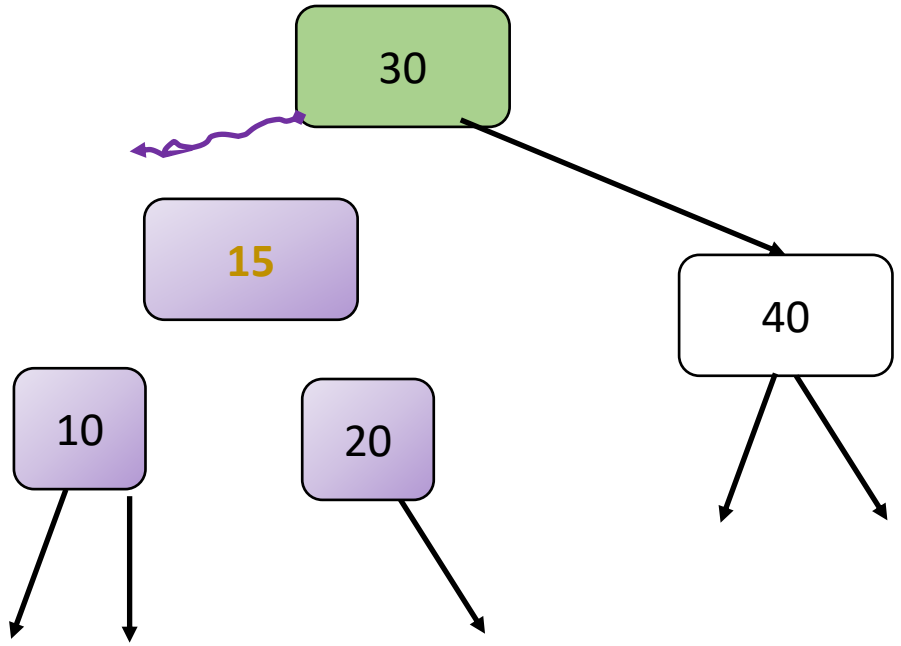
- Insert 15 in this 2-3 tree please! ☺



1. Search
2. Pretend

Walkthrough…

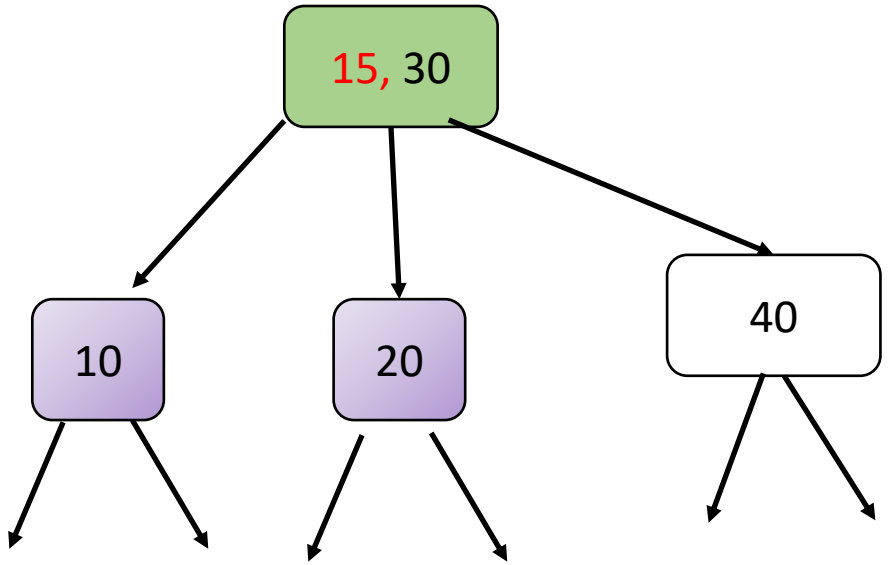
- Insert 15 in this 2-3 tree please! ☺



1. Search
2. Pretend
3. Kaboom

Walkthrough…

- Insert 15 in this 2-3 tree please! ☺

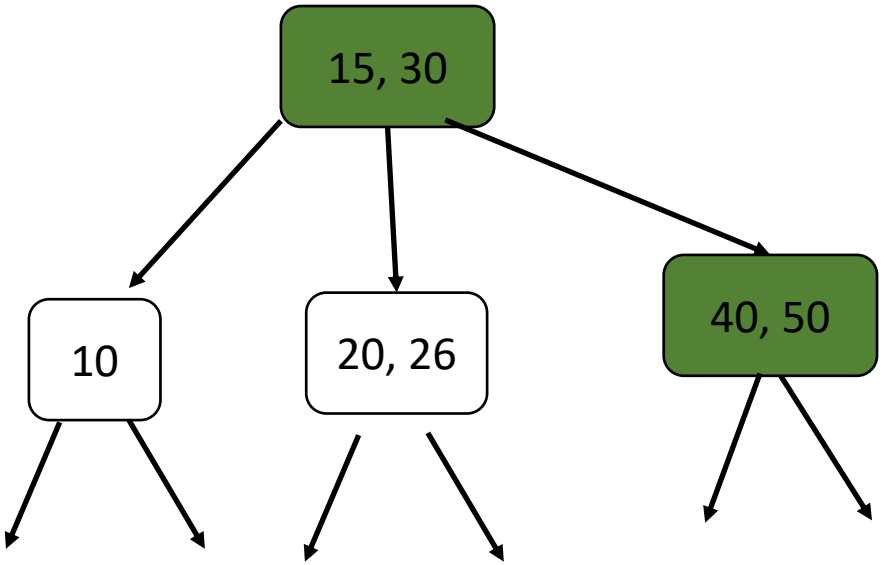


1. Search
2. Pretend
3. Kaboom
4. Merge with 2-node root, distributing children accordingly!

For you!

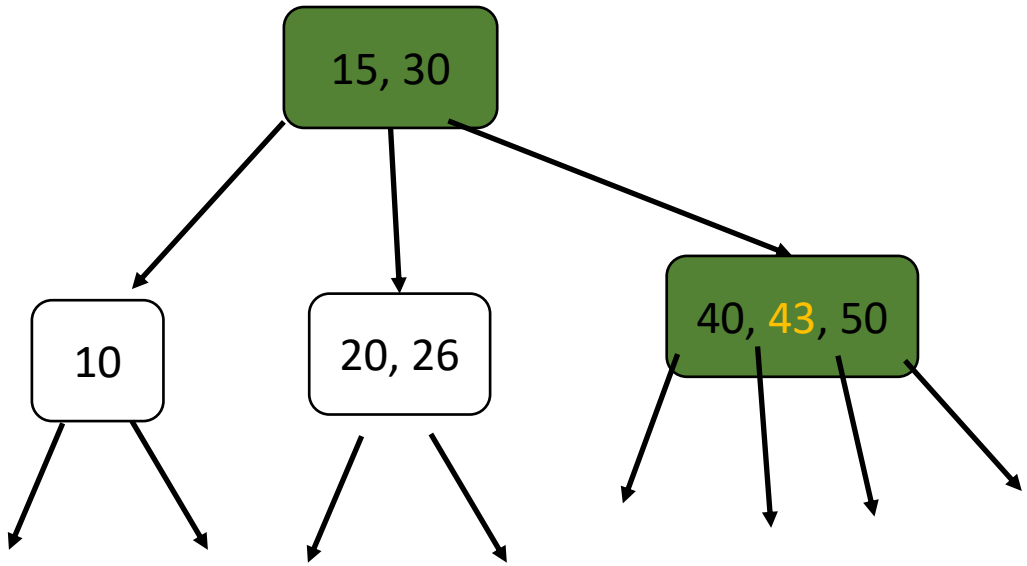
- Let’s now insert 43 in this tree…

1. Search



For you!

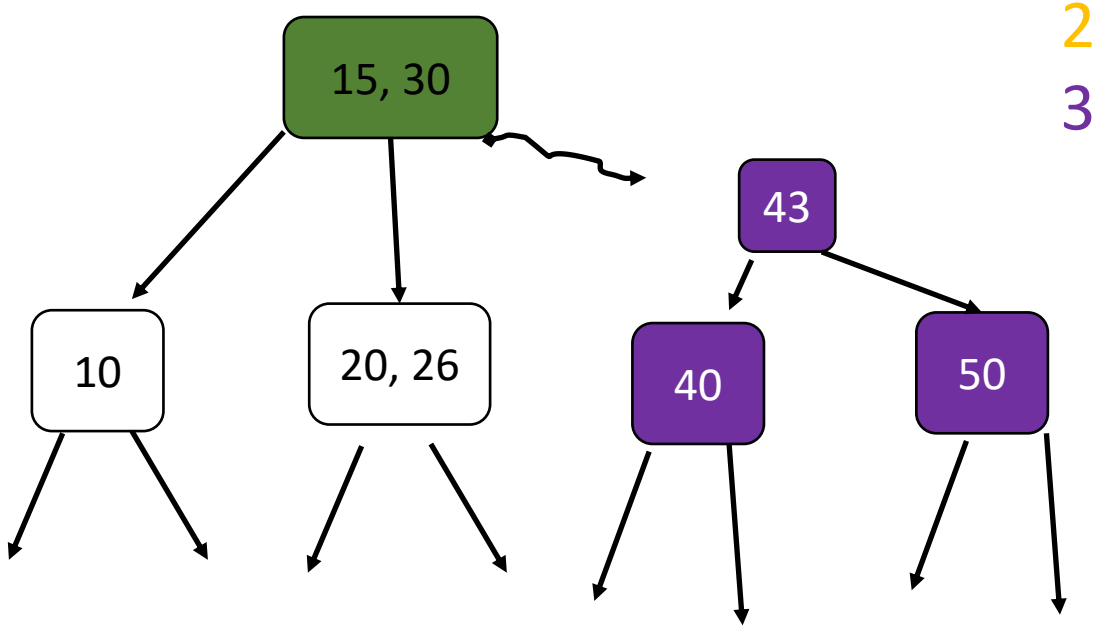
- Let’s now insert 43 in this tree…



1. Search
2. Pretend

For you!

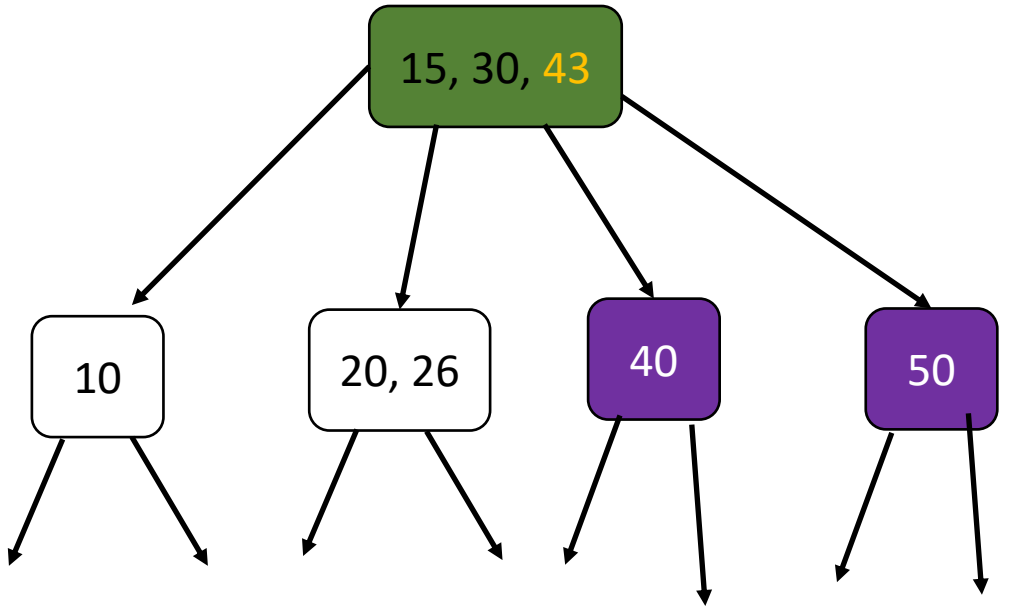
- Let’s now insert 43 in this tree…



1. Search
2. Pretend
3. Split

For you!

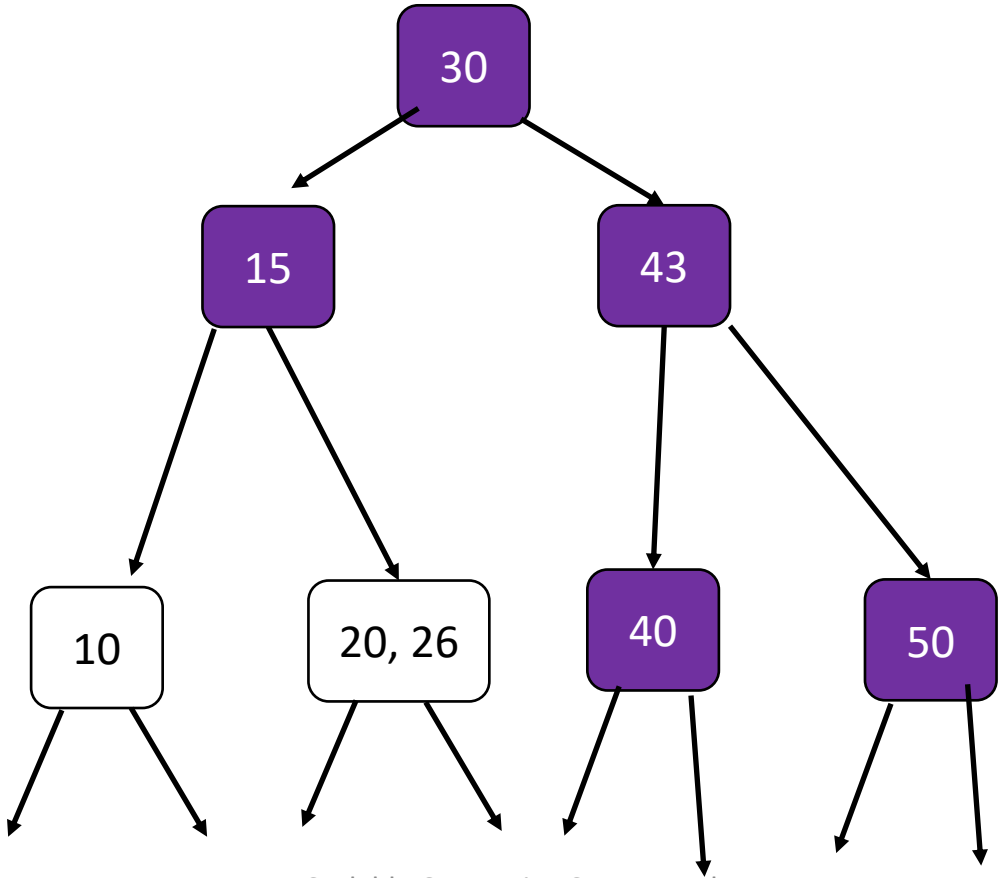
- Let’s now insert 43 in this tree…



1. Search
2. Pretend
3. Split
4. Pretend

For you!

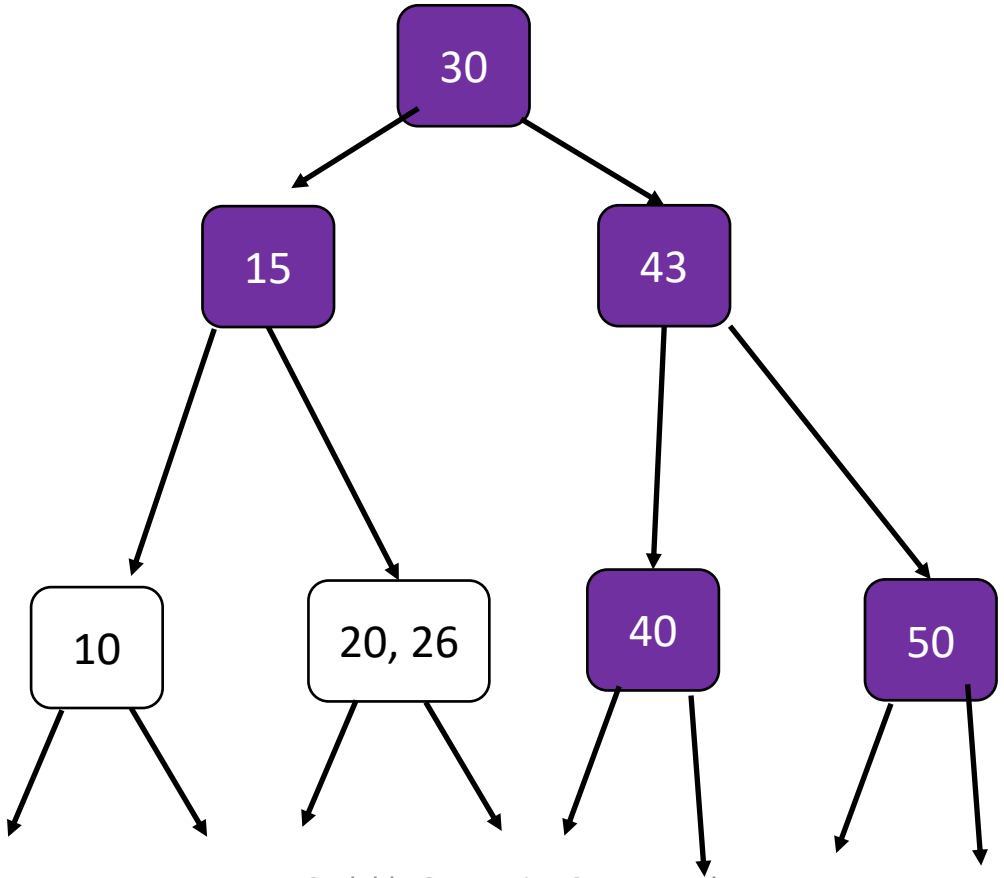
- Let’s now insert 43 in this tree…



1. Search
2. Pretend
3. Split
4. Pretend
5. Split

For you!

- Let’s now insert 43 in this tree…



1. Search
2. Pretend
3. Split
4. Pretend
5. Split

Done!

Deletion

- Insertions in 2-3 trees might transform a 2-node into a 3-node, or blow up a temporary 4-node into 3 2-nodes.
- Deletions, on the other hand, have the potential of not just making 3-nodes into 2-nodes, but making 2-nodes into ...

Deletion

- Insertions in 2-3 trees might transform a 2-node into a 3-node, or blow up a temporary 4-node into 3 2-nodes.
- Deletions, on the other hand, have the potential of not just making 3-nodes into 2-nodes, but making 2-nodes into … ?

Deletion

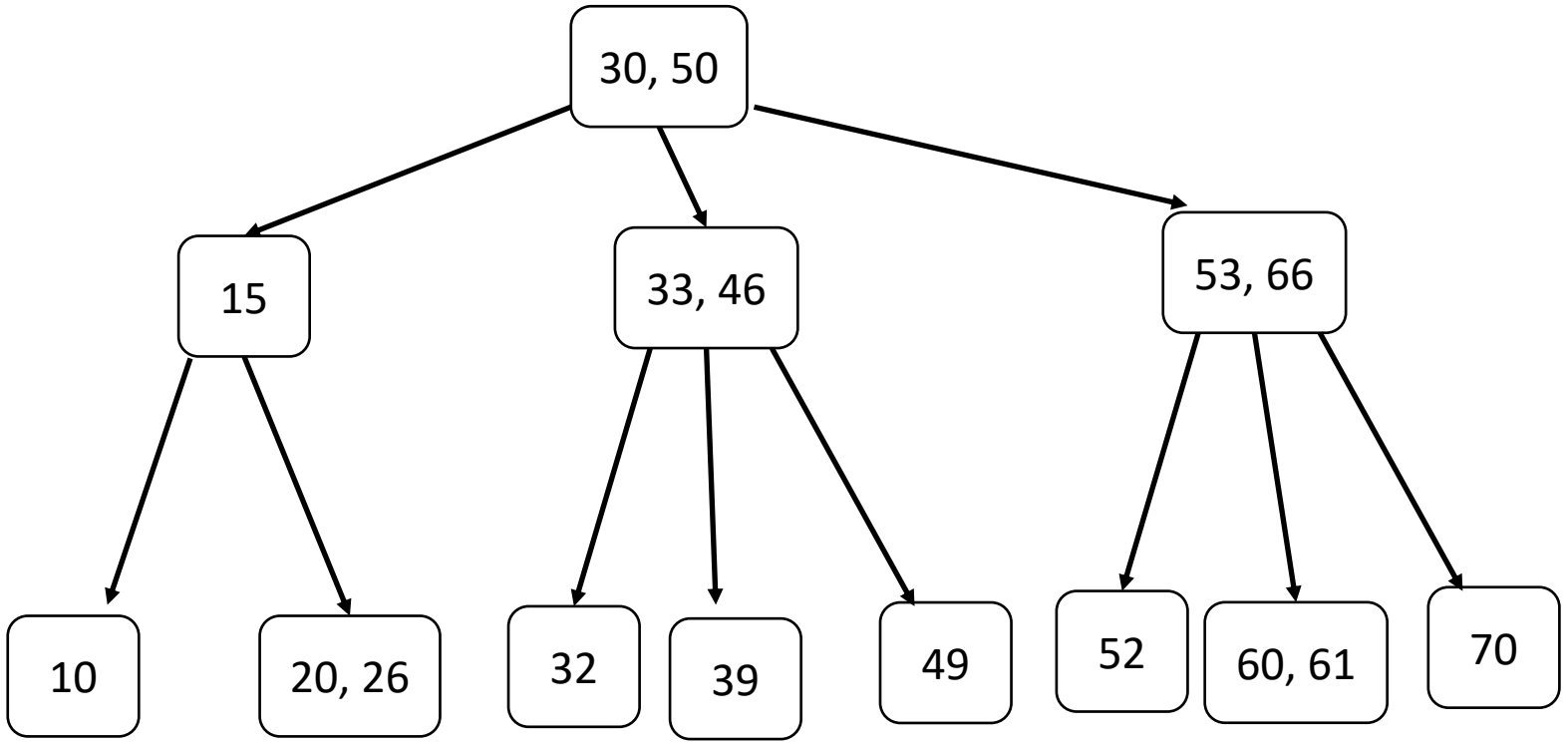
- Insertions in 2-3 trees might transform a 2-node into a 3-node, or blow up a temporary 4-node into 3 2-nodes.
- Deletions, on the other hand, have the potential of not just making 3-nodes into 2-nodes, but making 2-nodes into … ?
- We'll see what the result of that will be soon.

Deletion

- Insertions in 2-3 trees might transform a 2-node into a 3-node, or blow up a temporary 4-node into 3 2-nodes.
- Deletions, on the other hand, have the potential of not just making 3-nodes into 2-nodes, but making 2-nodes into … ?
- We'll see what the result of that will be soon.
- Key point: Since deleting an inner node always ends

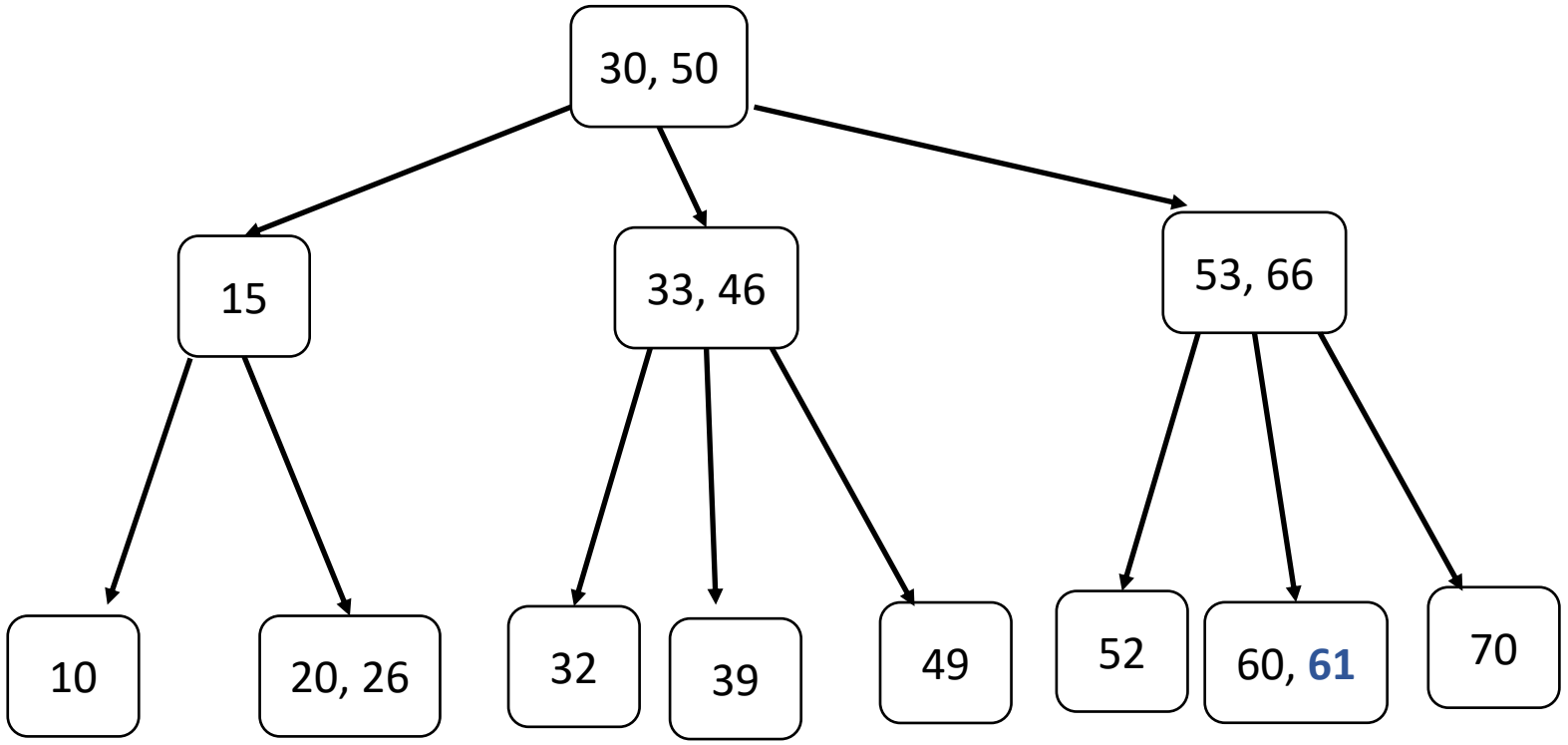
up being a deletion at the leaf level, we only care about leaf deletions.

Deletion



Deletion

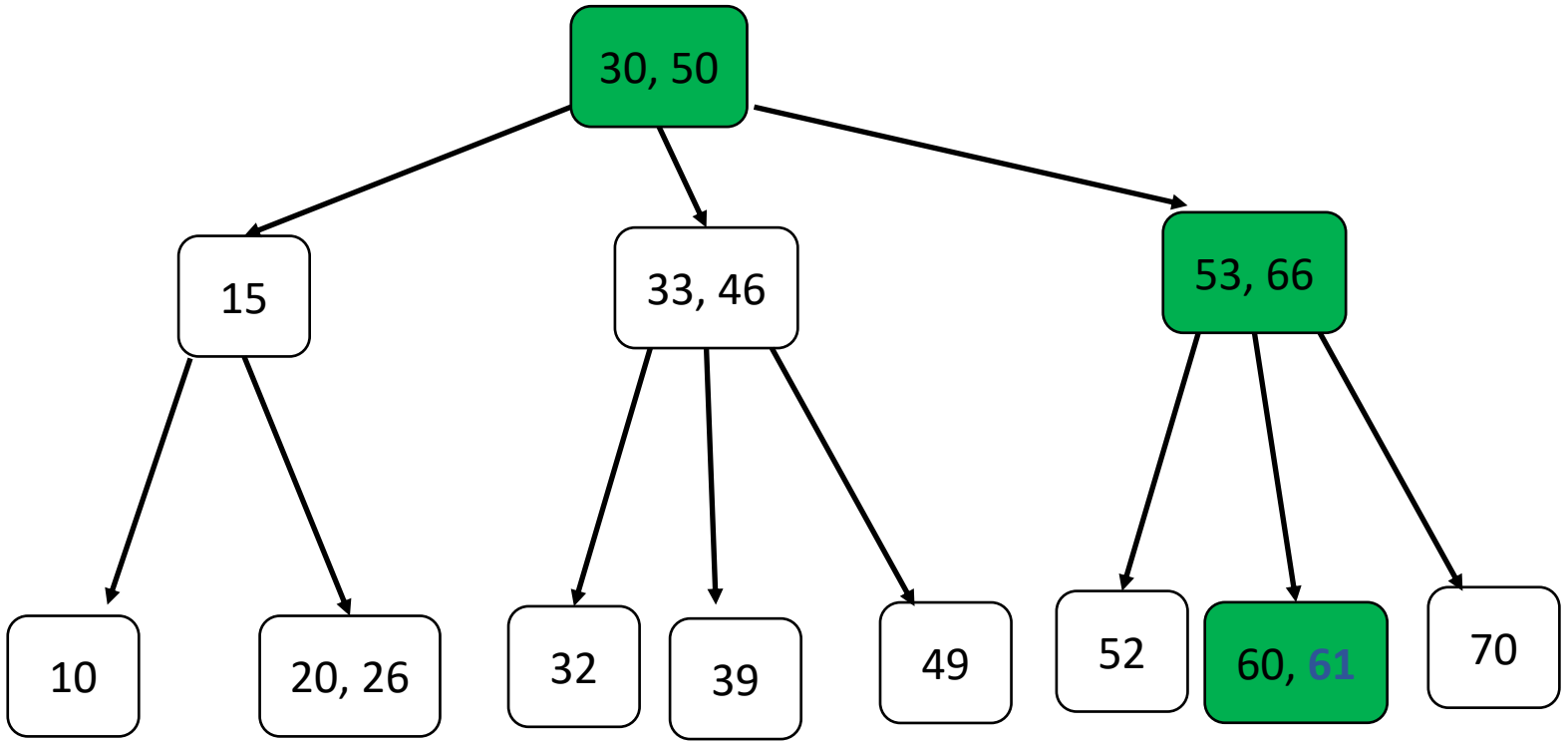
Delete 61



Deletion

Delete 61

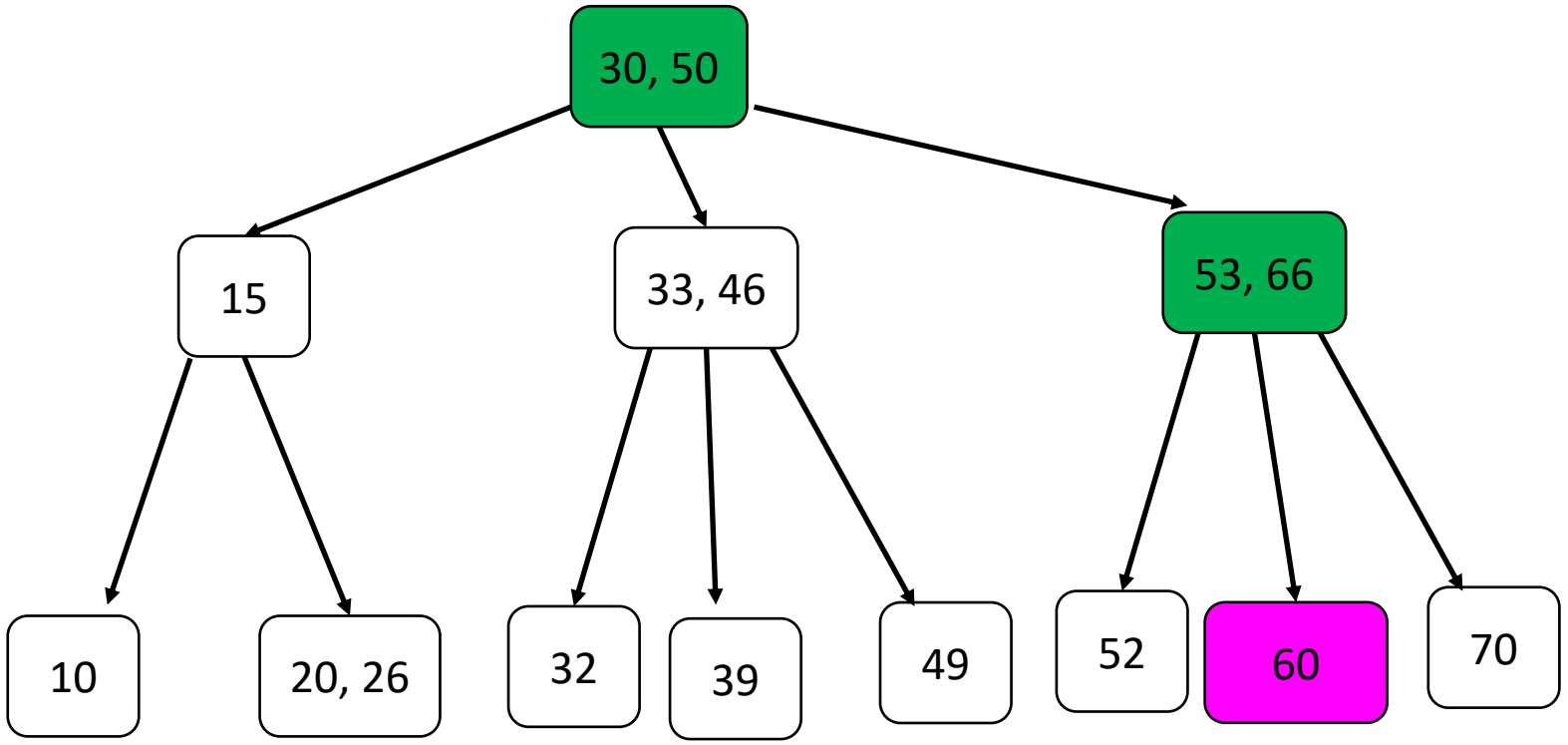
1. Search



Deletion

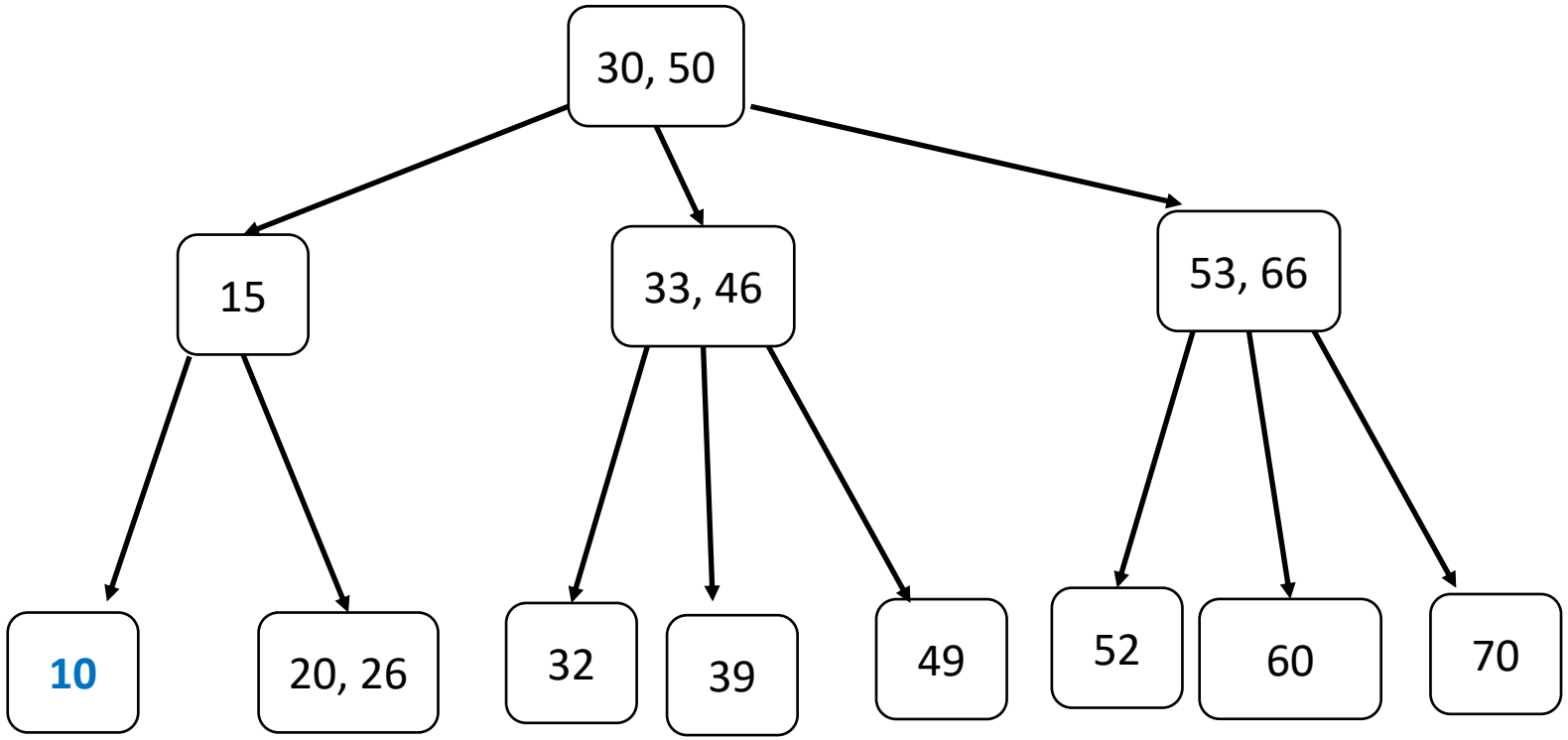
Delete 61

1. **Search**
2. **Since node that co ntains 61 is a 3-no de, we just remov e 61, make it a 2-n ode and we're do ne! ☺**



Deletion

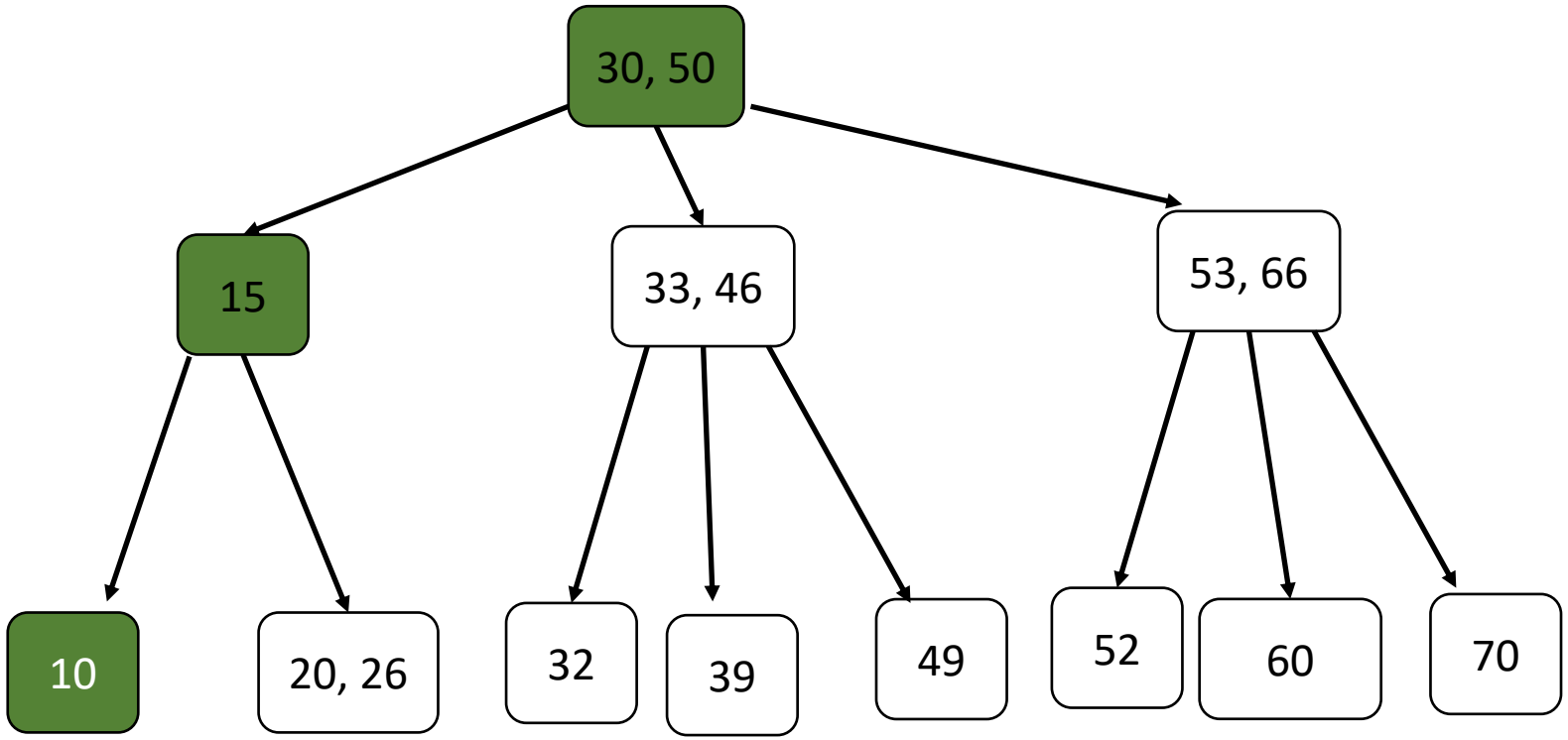
Delete 10



Deletion

Delete 10

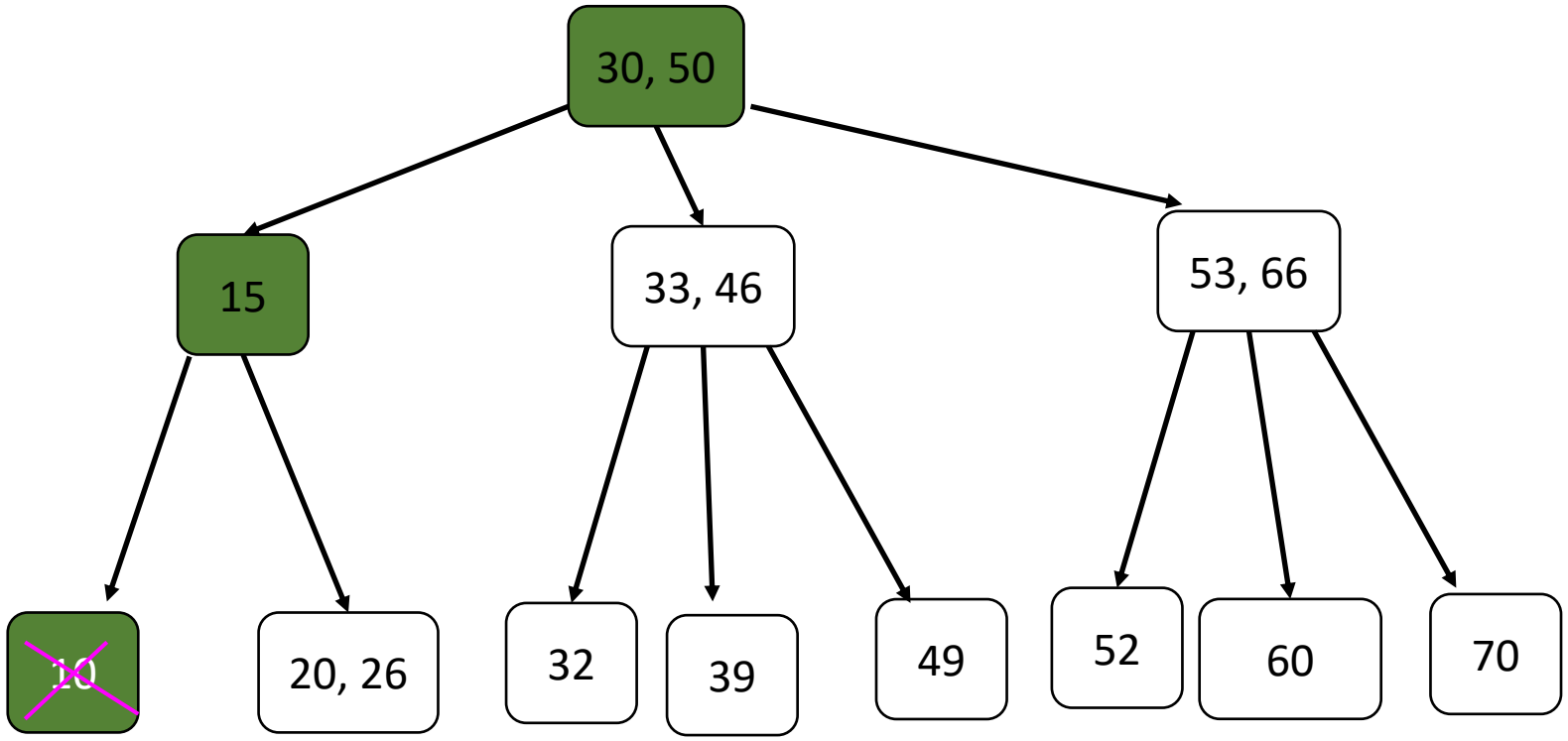
1. Search



Deletion

Delete 10

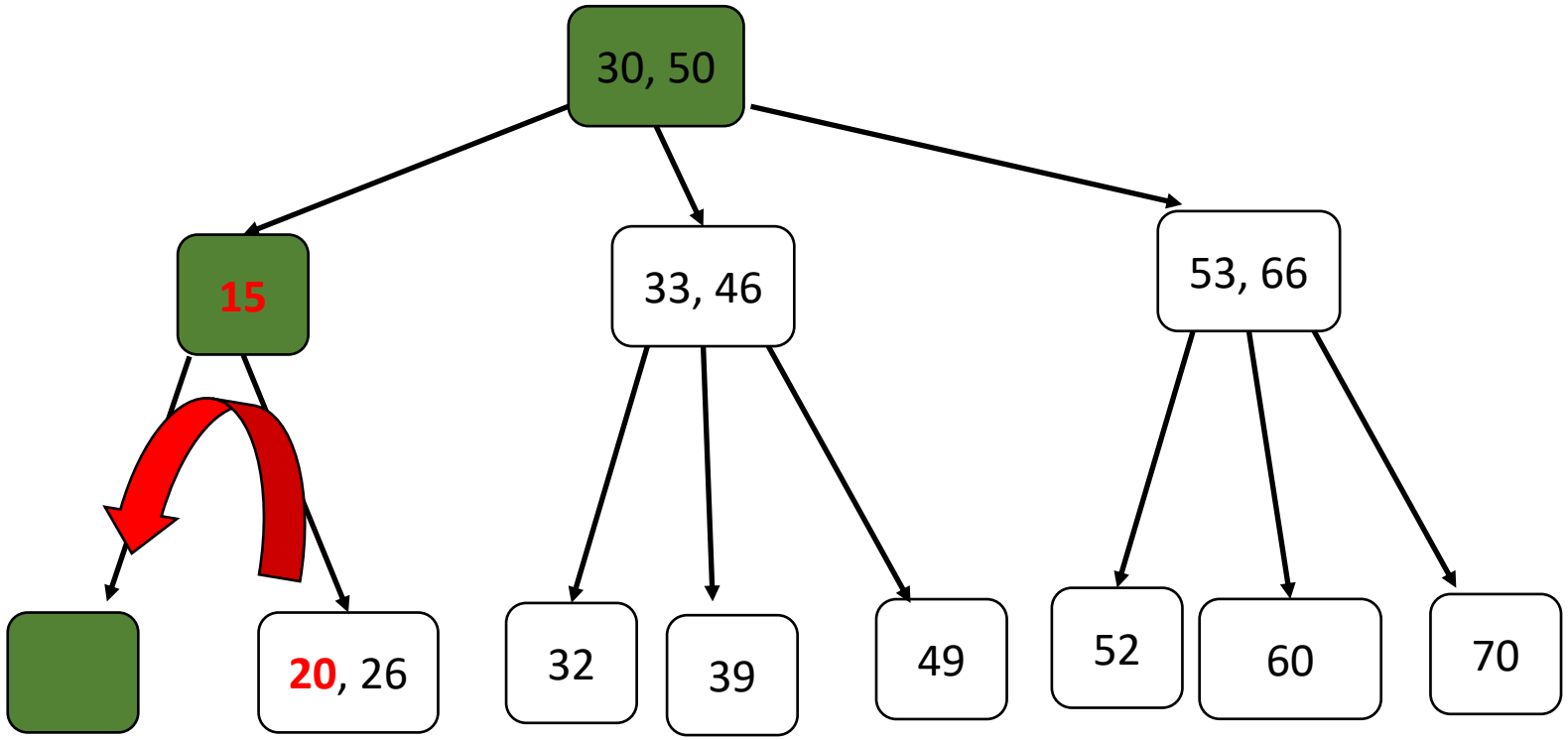
1. **Search**
2. Deleting 10 leads t
   o a node underflo
   w… ☹



Deletion

Delete 10

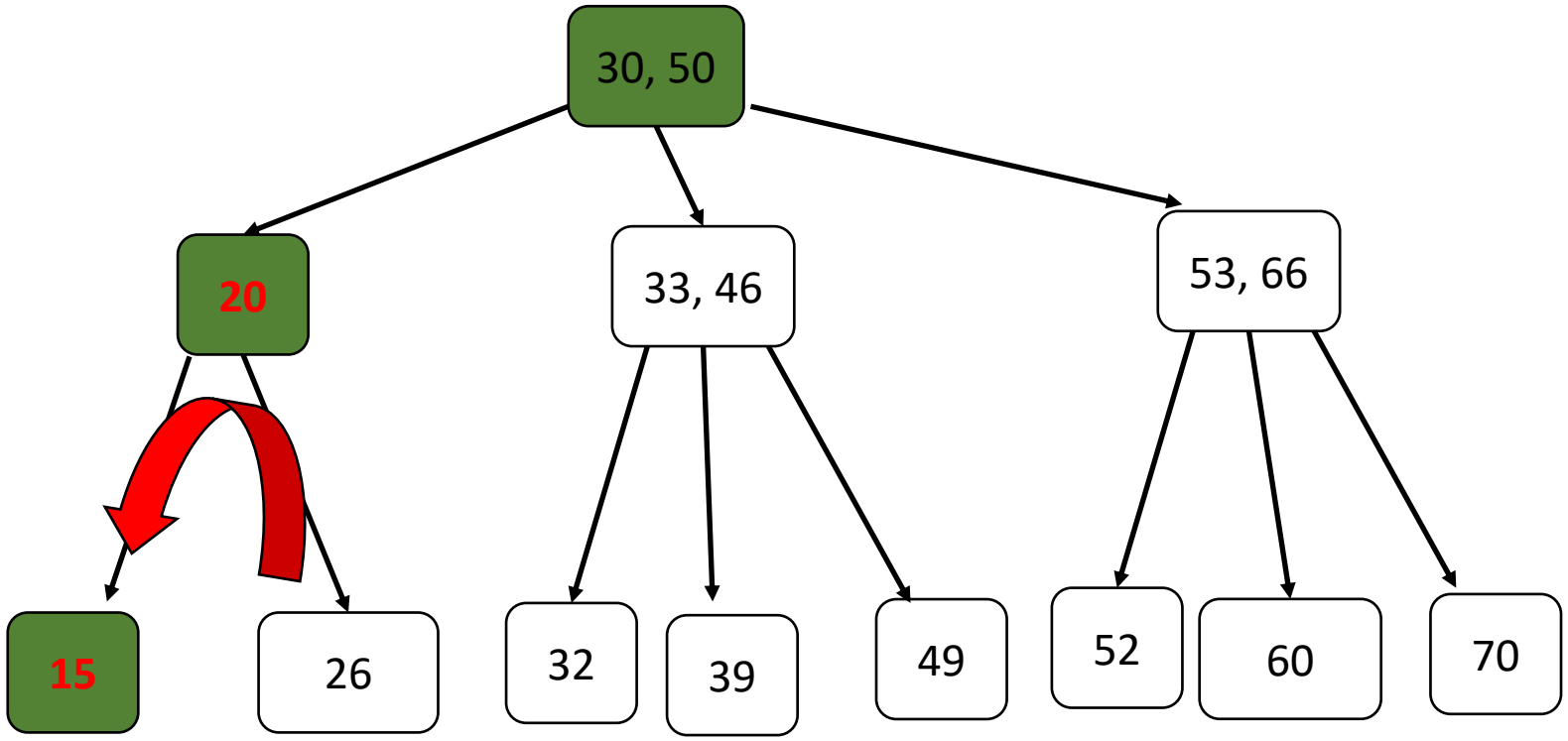
1. **Search**
2. Deleting 10 leads to a node underflow… ☹
3. The solution: rotate a key from our sibling! ☺



Deletion

Delete 10

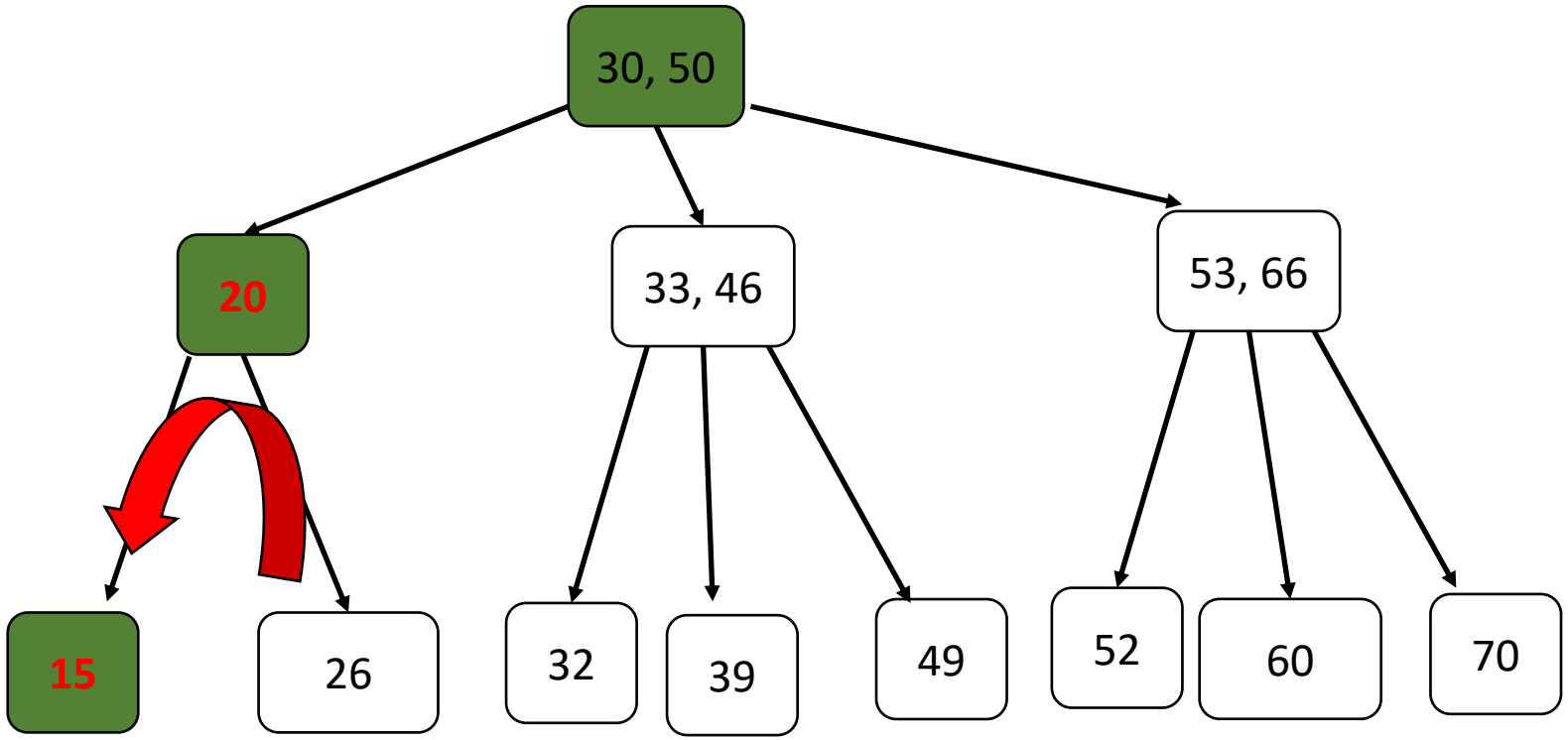
1. **Search**
2. Deleting 10 leads t o a node underflo w… ☹
3. The solution: rotat e a key from our si bling! ☺



Deletion

Delete 10

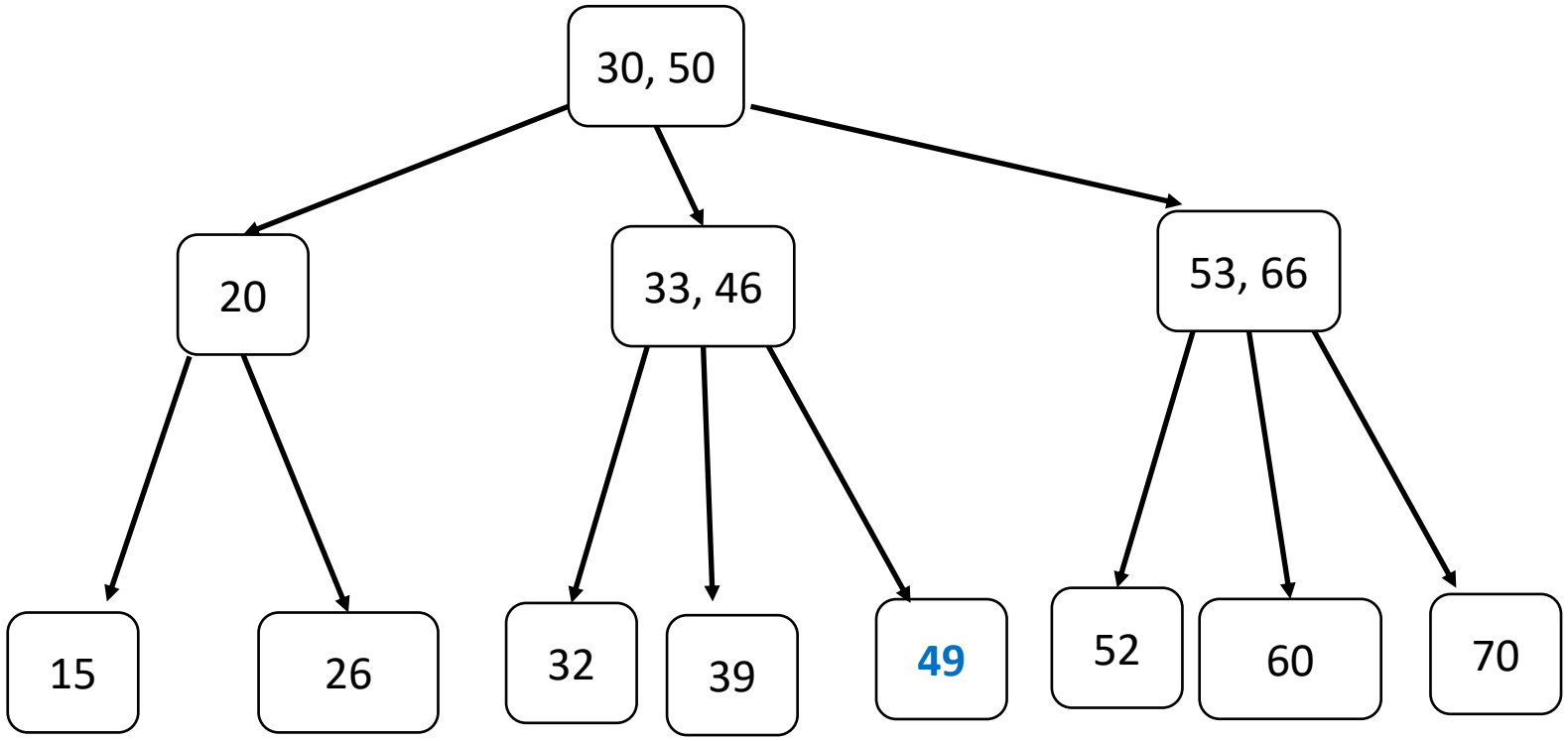
1. **Search**
2. Deleting 10 leads t o a node underflo w… ☹
3. The solution: rotat e a key from our si bling! ☺



These operations are called **"key rotations"**, to disambiguate them from the **nod e rotations** that occur in AVL and Splay trees.

Deletion

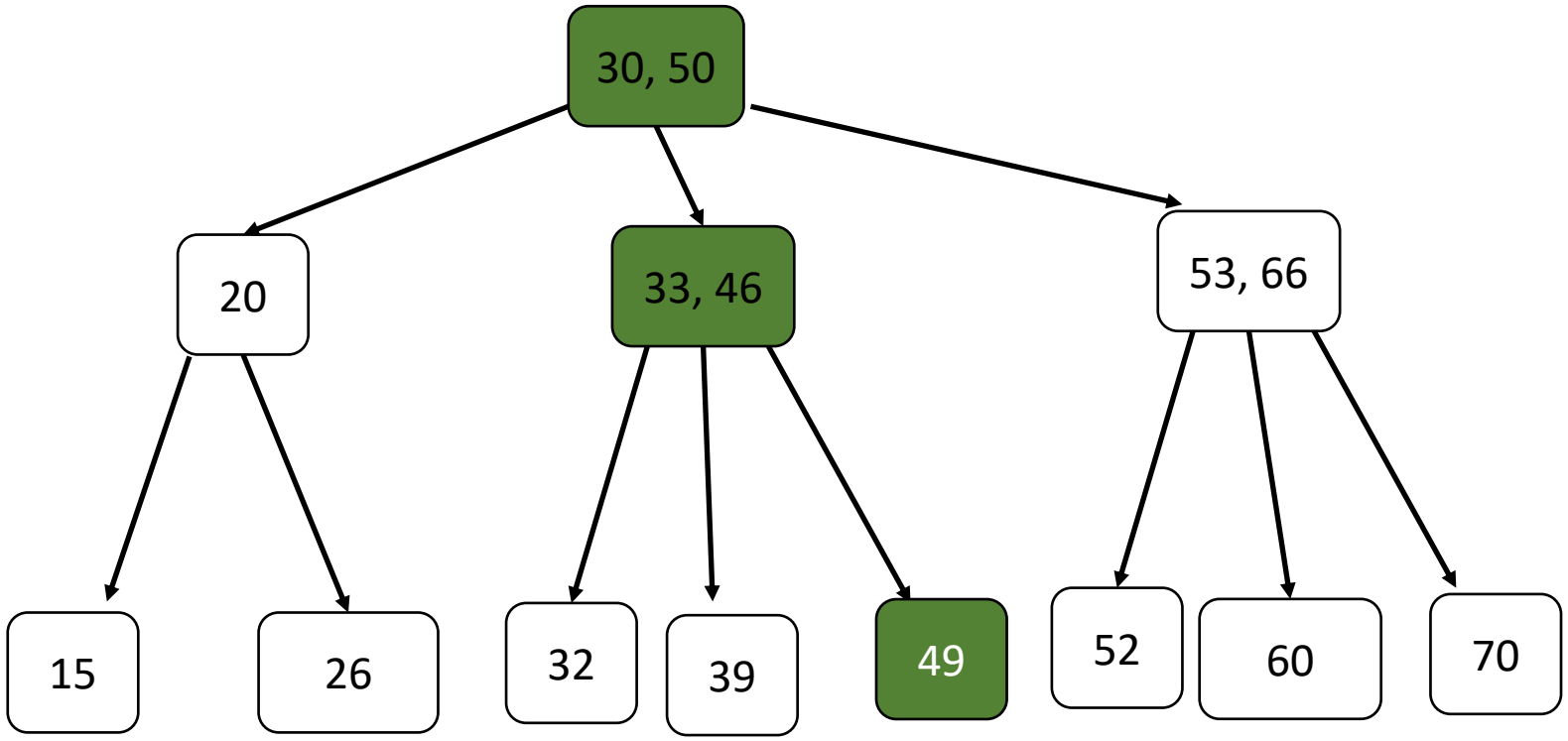
Delete 49



Deletion

Delete 49

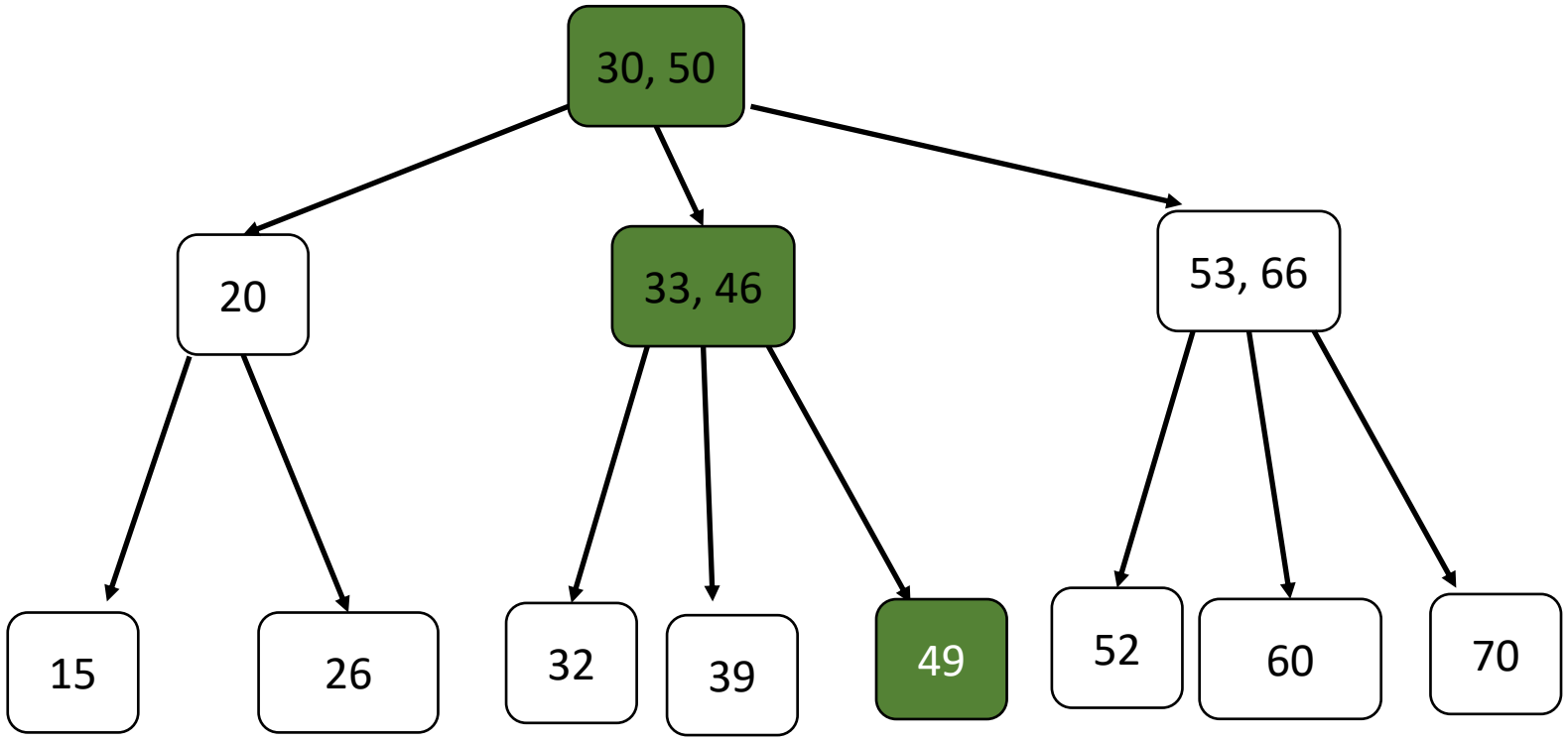
1. Search



Deletion

Delete 49

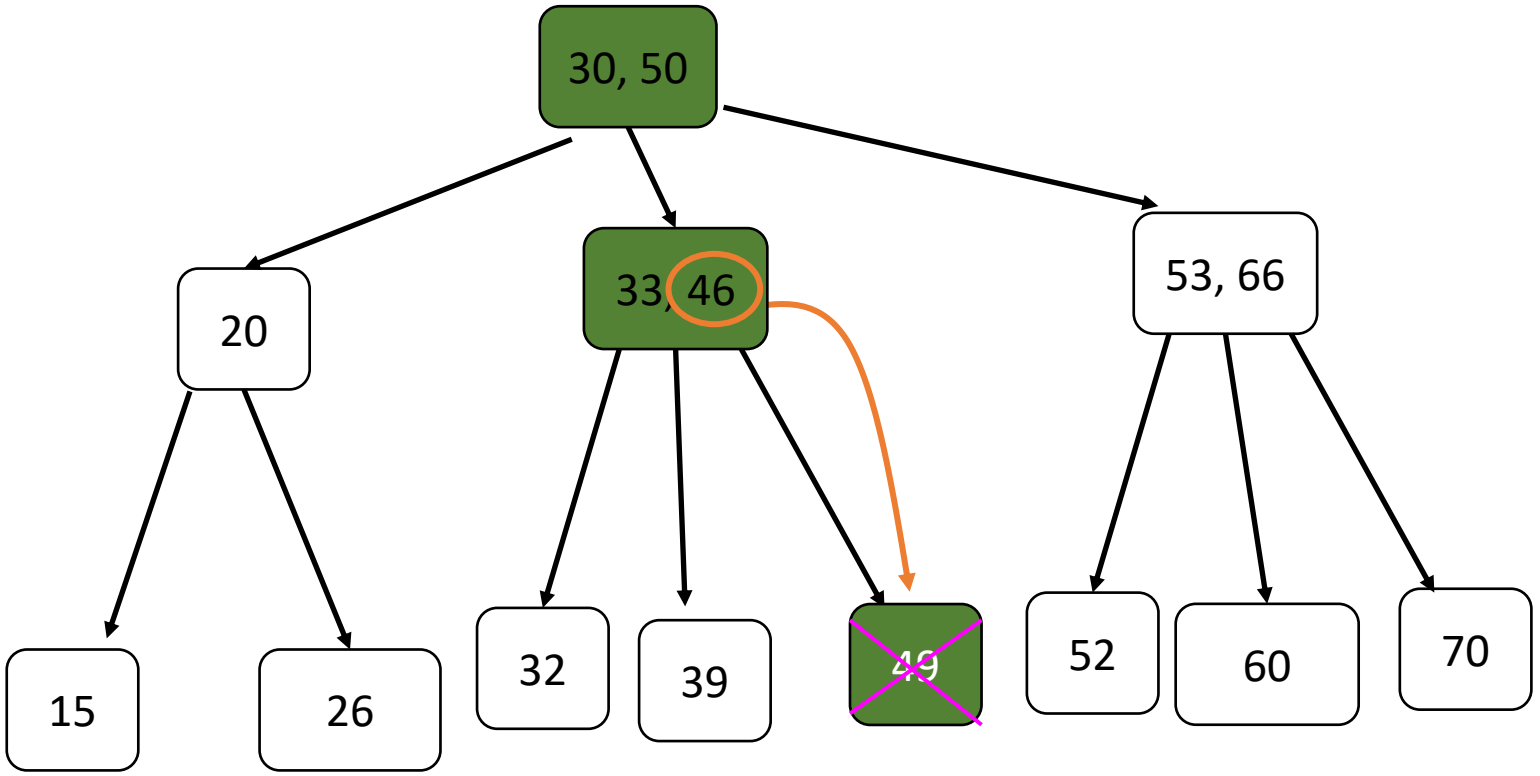
1. **Search**
2. **49 is contained in a 2-node, and both our siblings are 2-nodes as well… ☹**



Deletion

Delete 49

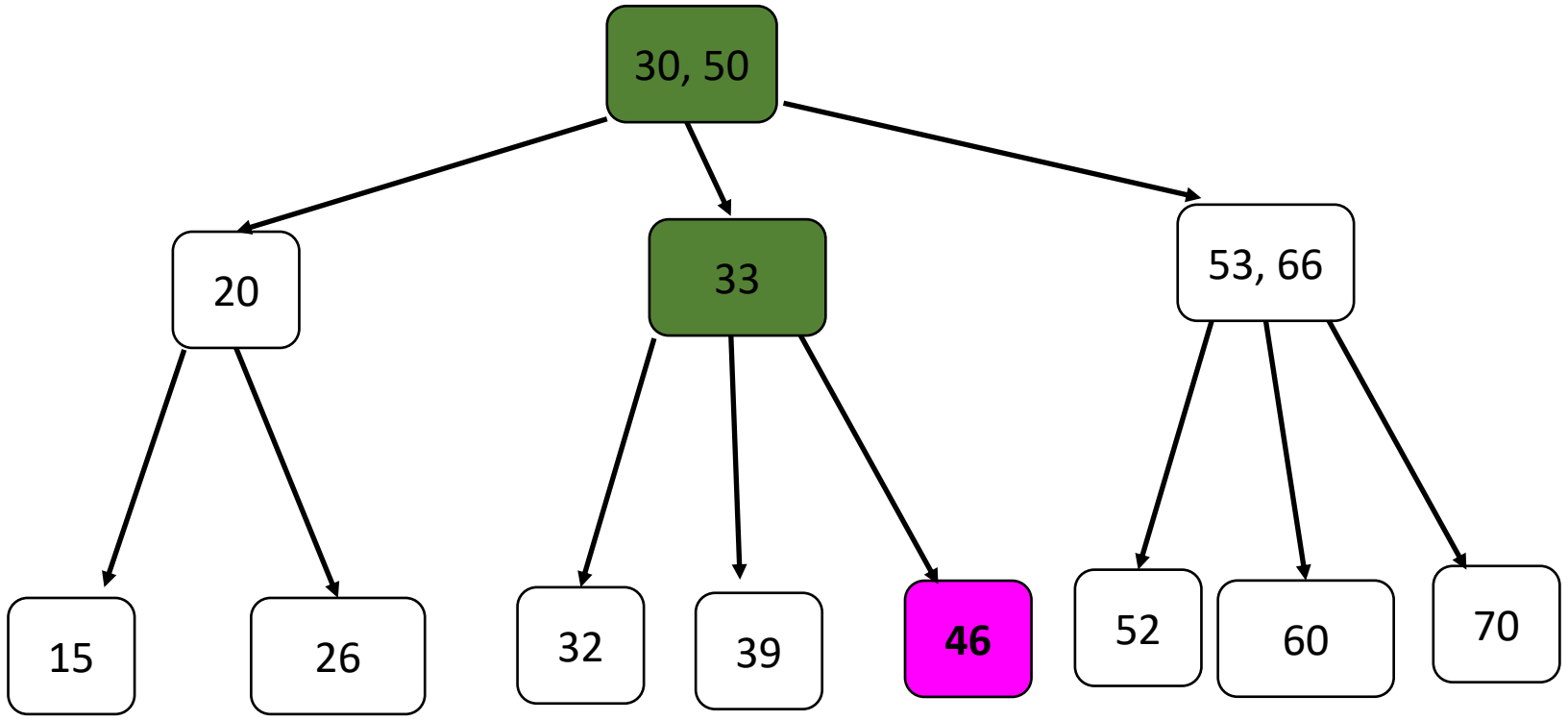
1. **Search**
2. **49 is contained in a 2-node, and both our siblings are 2-nodes as well… ☹**
3. **Solution: Grab a key from the parent!**



Deletion

Delete 49

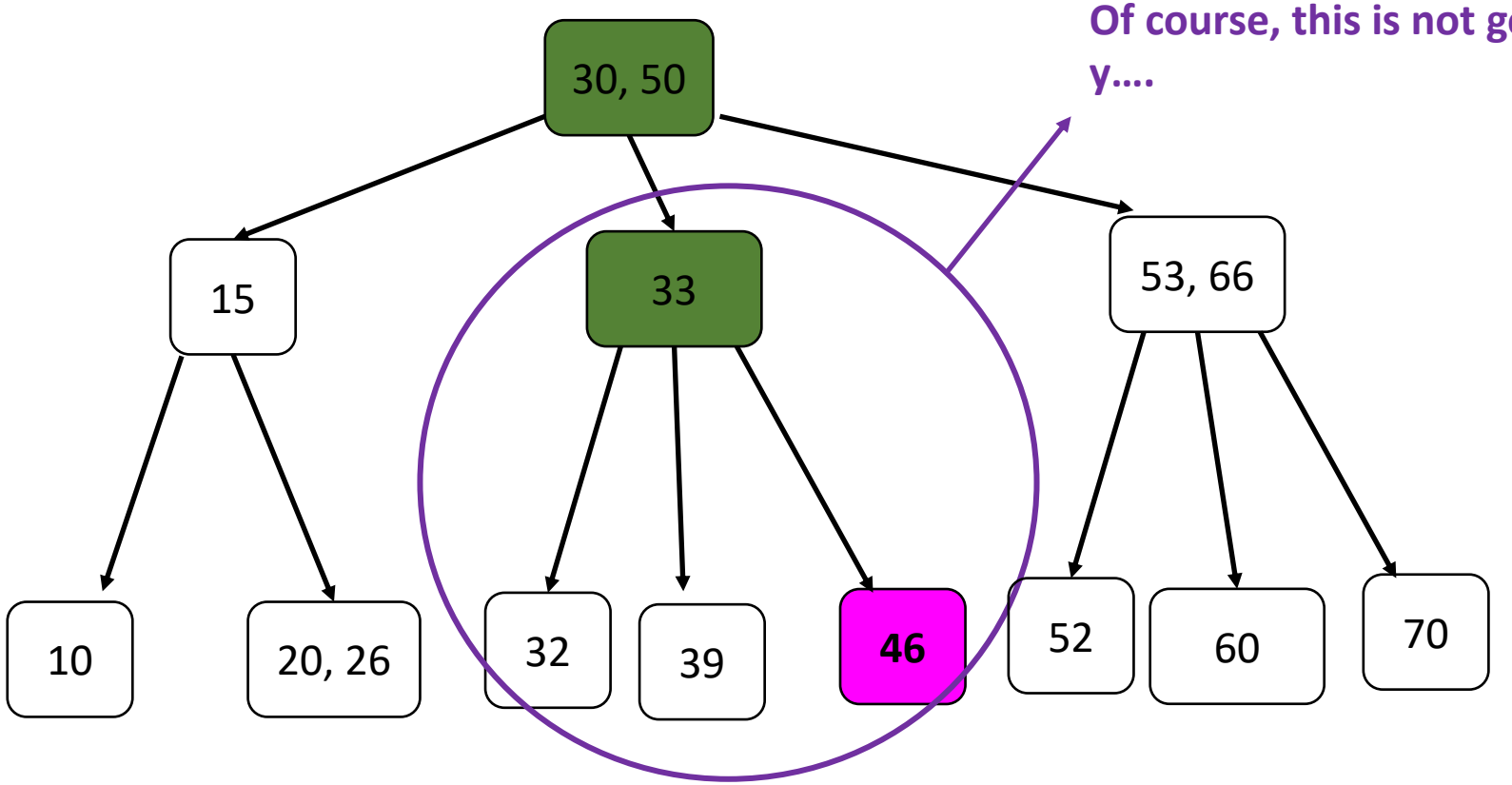
1. **Search**
2. **49 is contained in a 2-node, so deleting the node would create an imbalance… ☹**
3. **Solution: Grab a key from the parent!**



Deletion

Delete 49

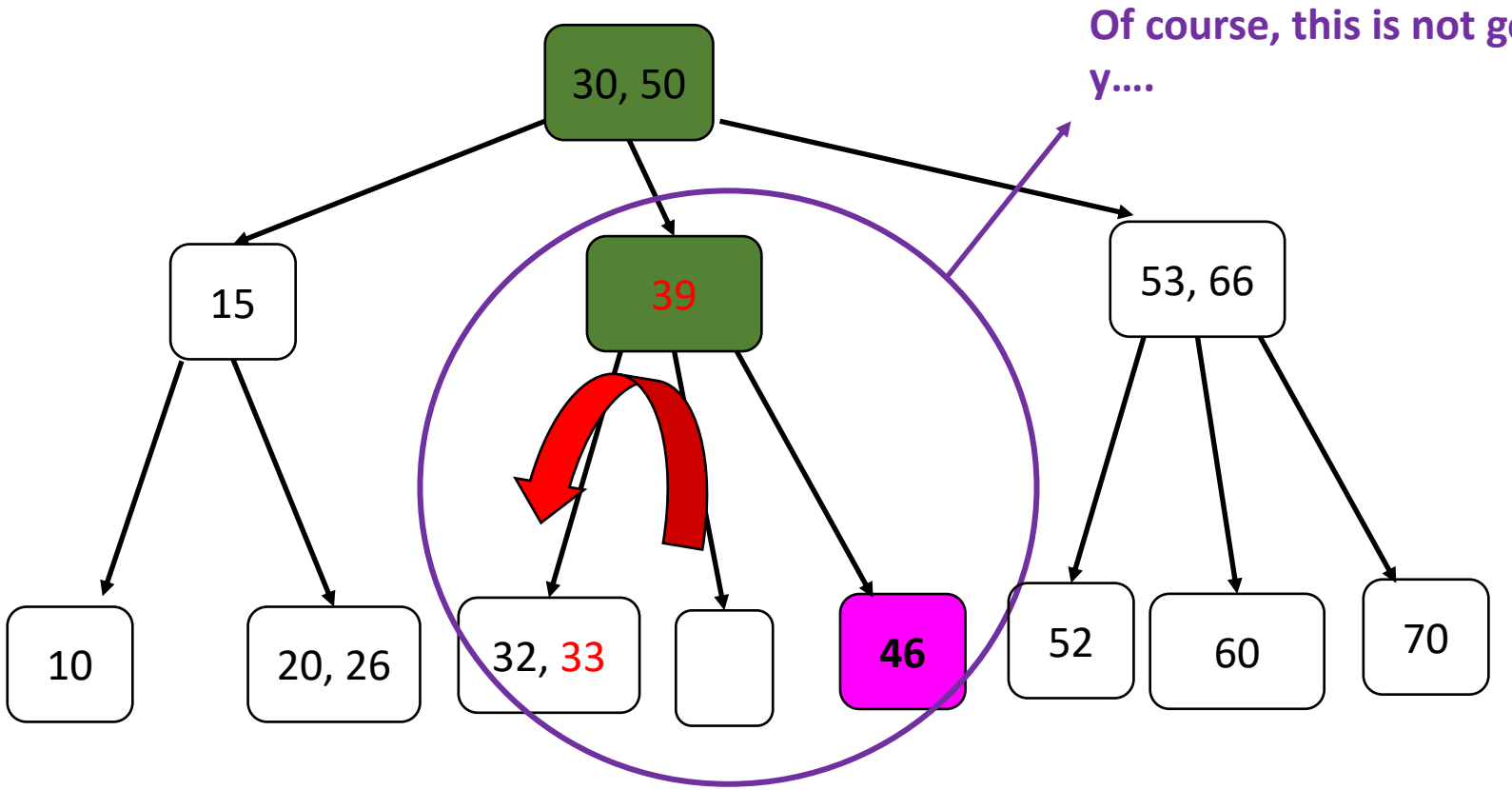
1. **Search**
2. **49 is contained in a 2-node, so deleting the node would create an imbalance… ☹**
3. **Solution: Grab a key from the parent!**



Deletion

Delete 49

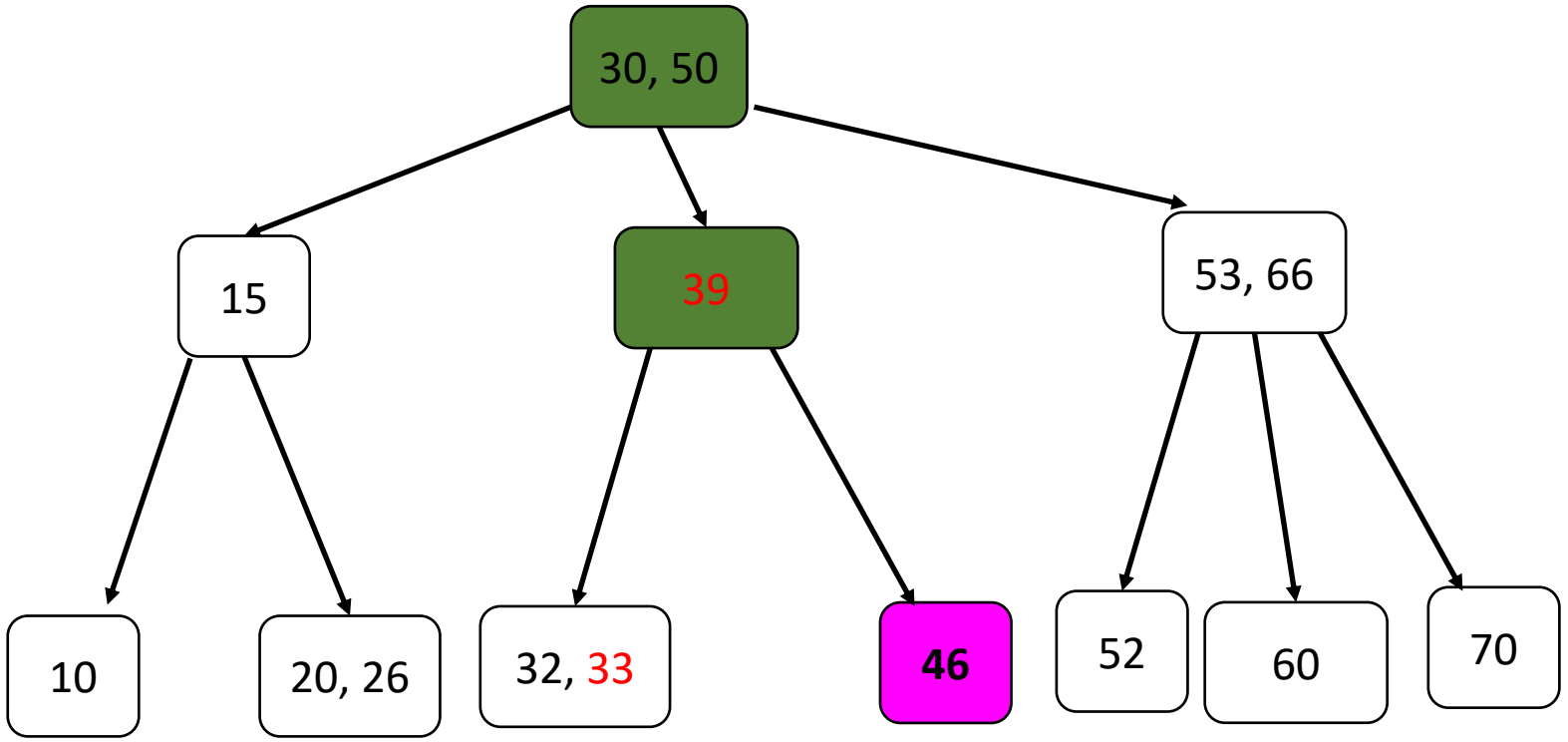
1. **Search**
2. **49 is contained in a 2-node, so deleting the node would create an imbalance… ☹**
3. **Solution: Grab a key from the parent!**
4. **To make the parent be a well-defined 2-node, rotate the middle key to the left!**



Deletion

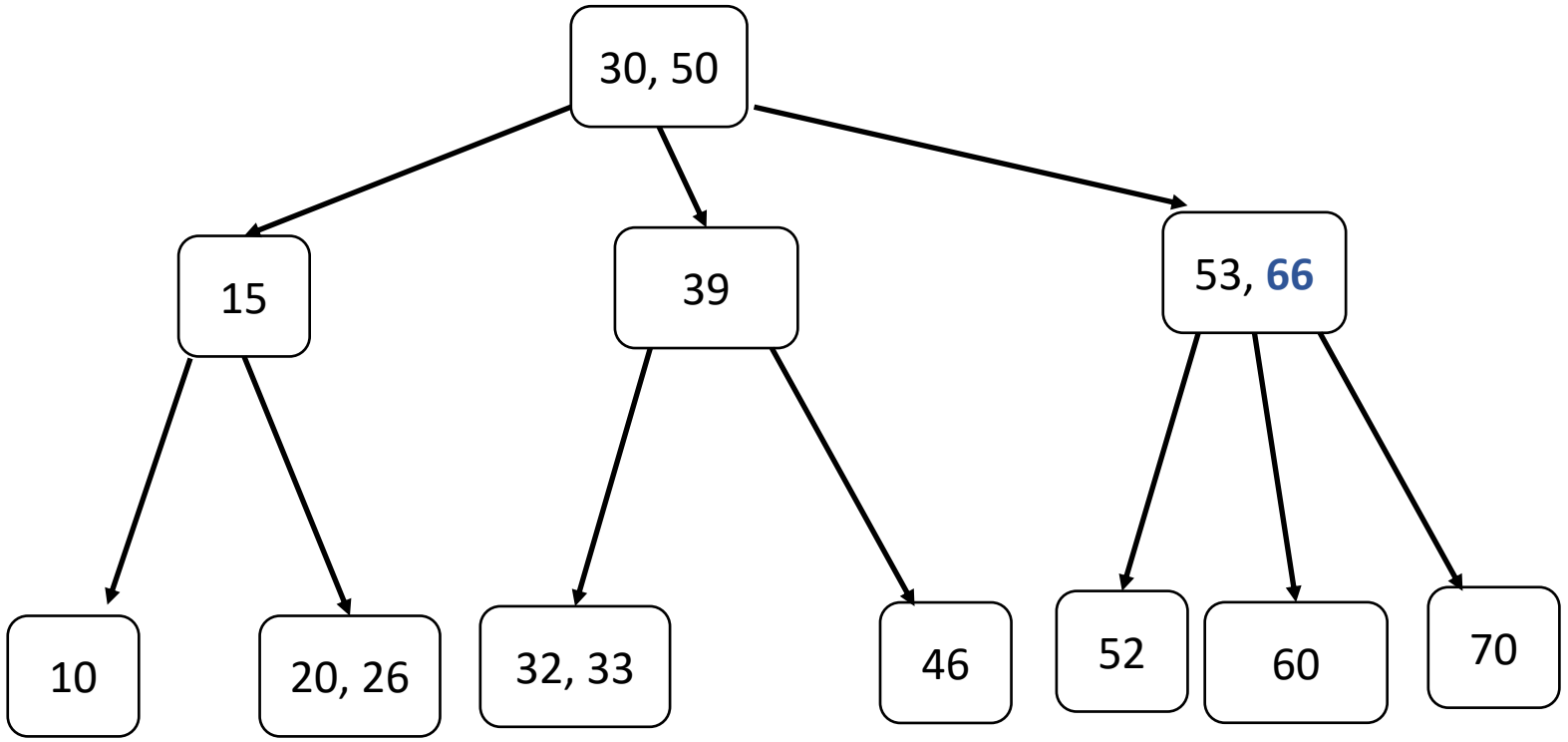
Delete 49

1. **Search**
2. **49 is contained in a 2-node, so deleting the node would create an imbalance… ☹**
3. **Solution: Grab a key from the parent!**
4. **To make the parent be a well-defined 2-node, rotate the middle key to the left!**



Deletion

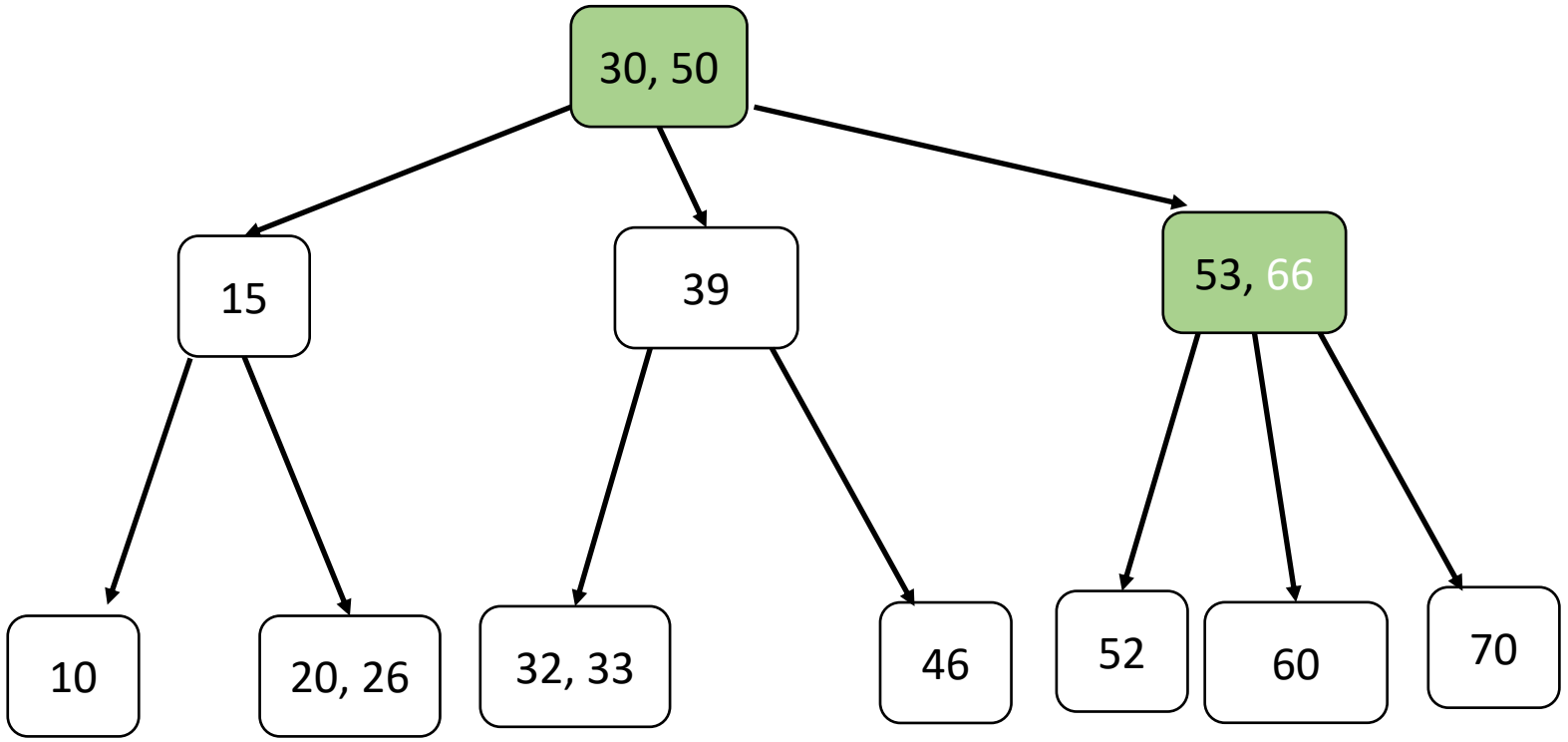
Delete 66



Deletion

Delete 66

1. Search

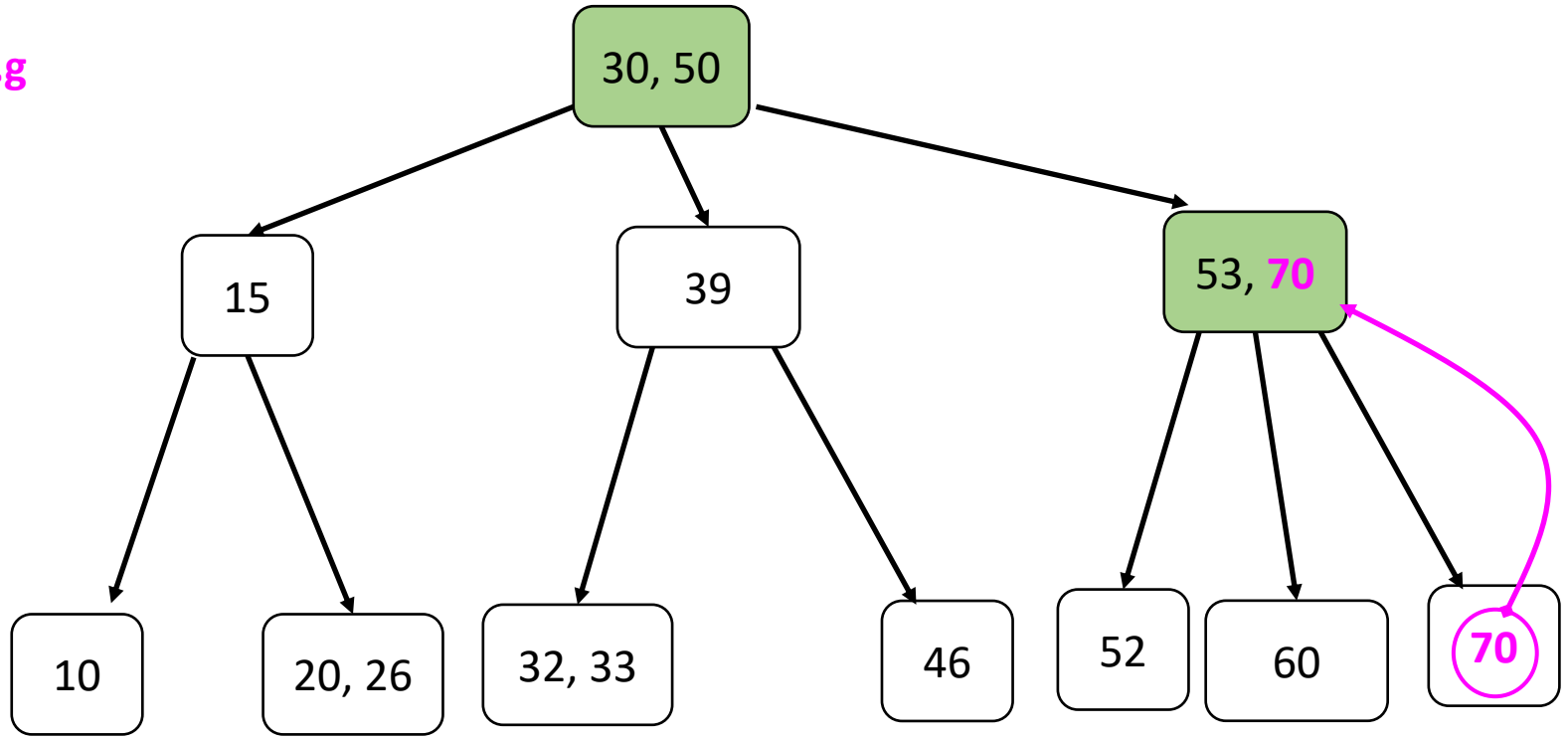


Deletion

Delete 66

1. Search

2. Have to replace key (e.g with inorder successor…)



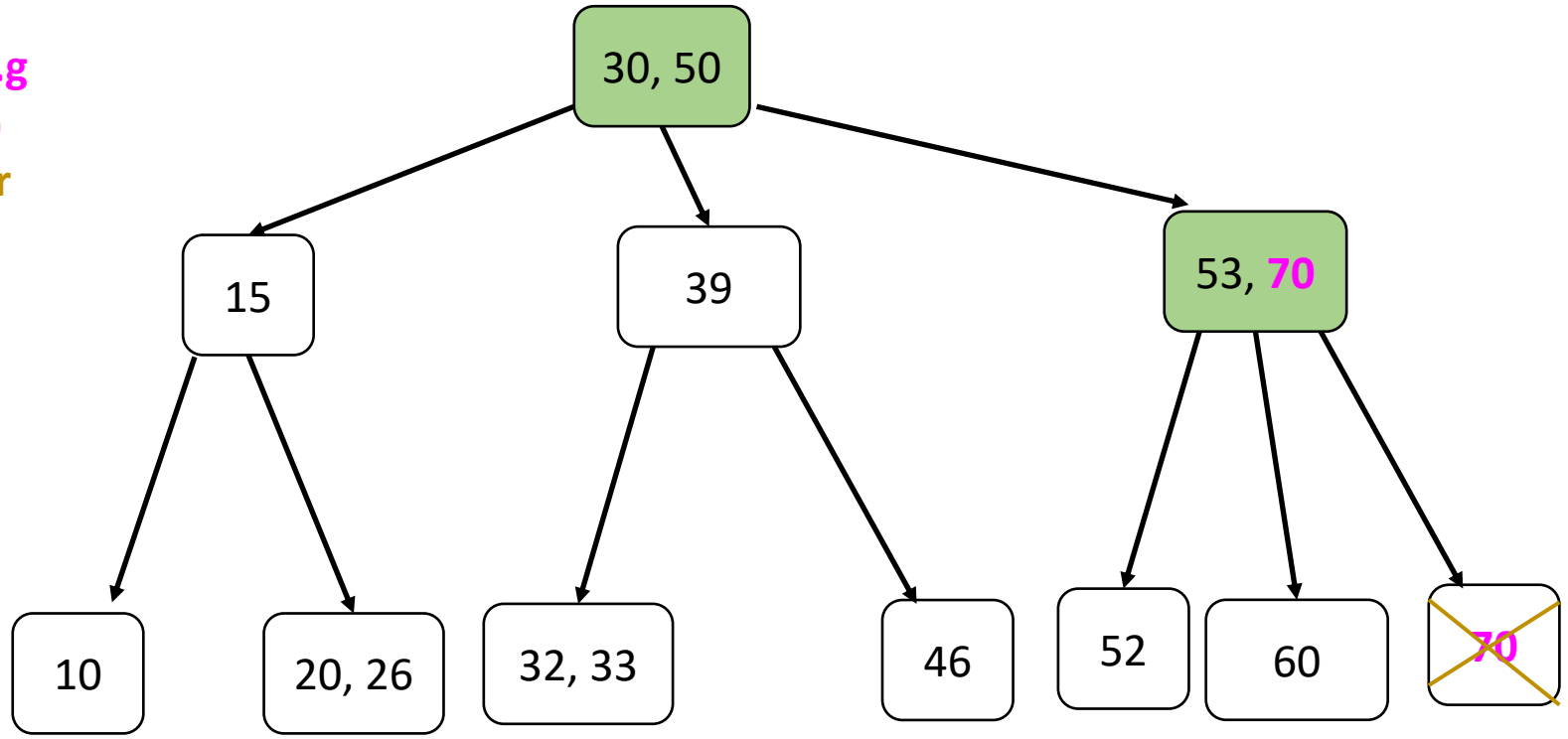
Deletion

Delete 66

1. Search

2. Have to replace key (e.g with inorder successor…)

3. Have to delete the inor der succesor



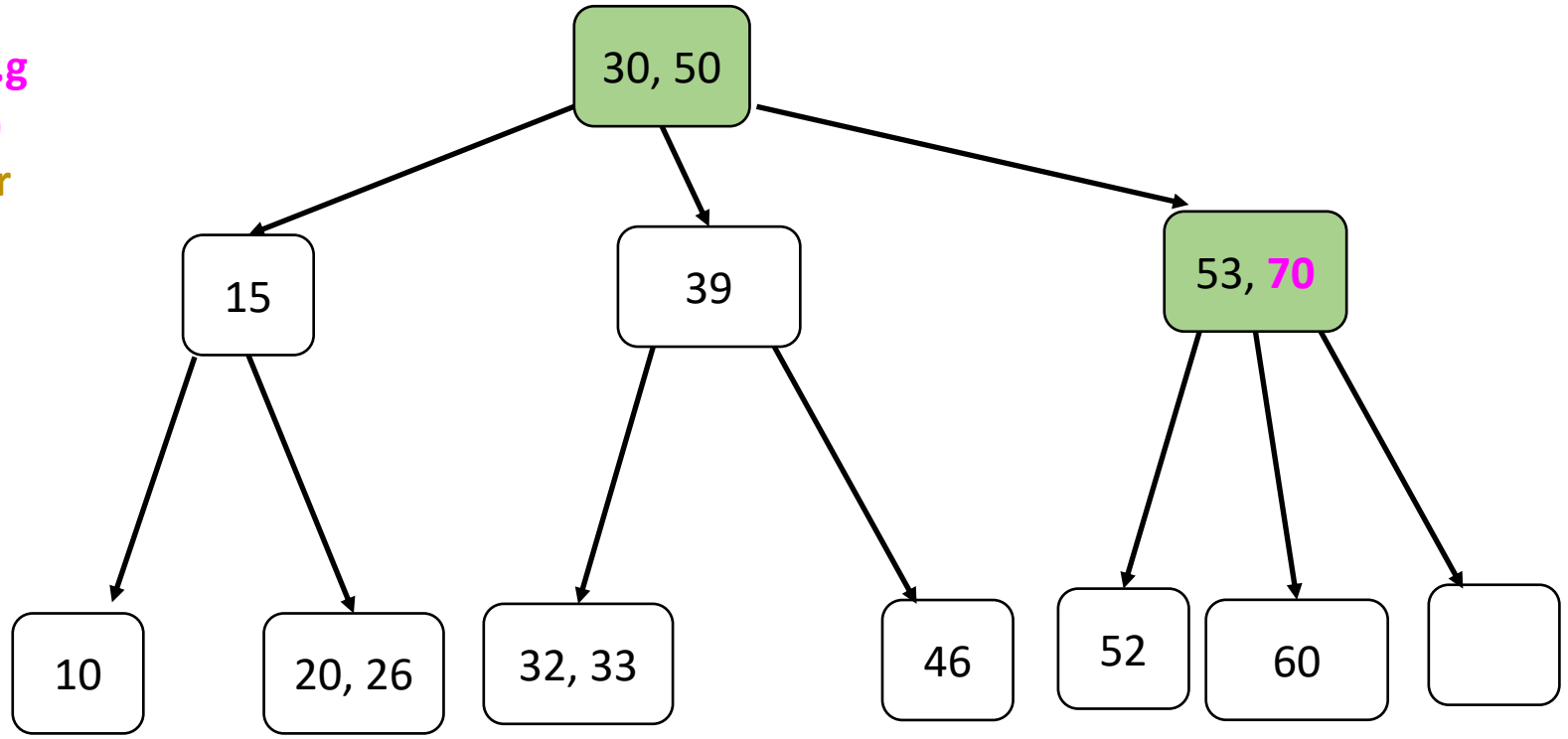
Deletion

Delete 66

1. Search

2. Have to replace key (e.g with inorder successor…)

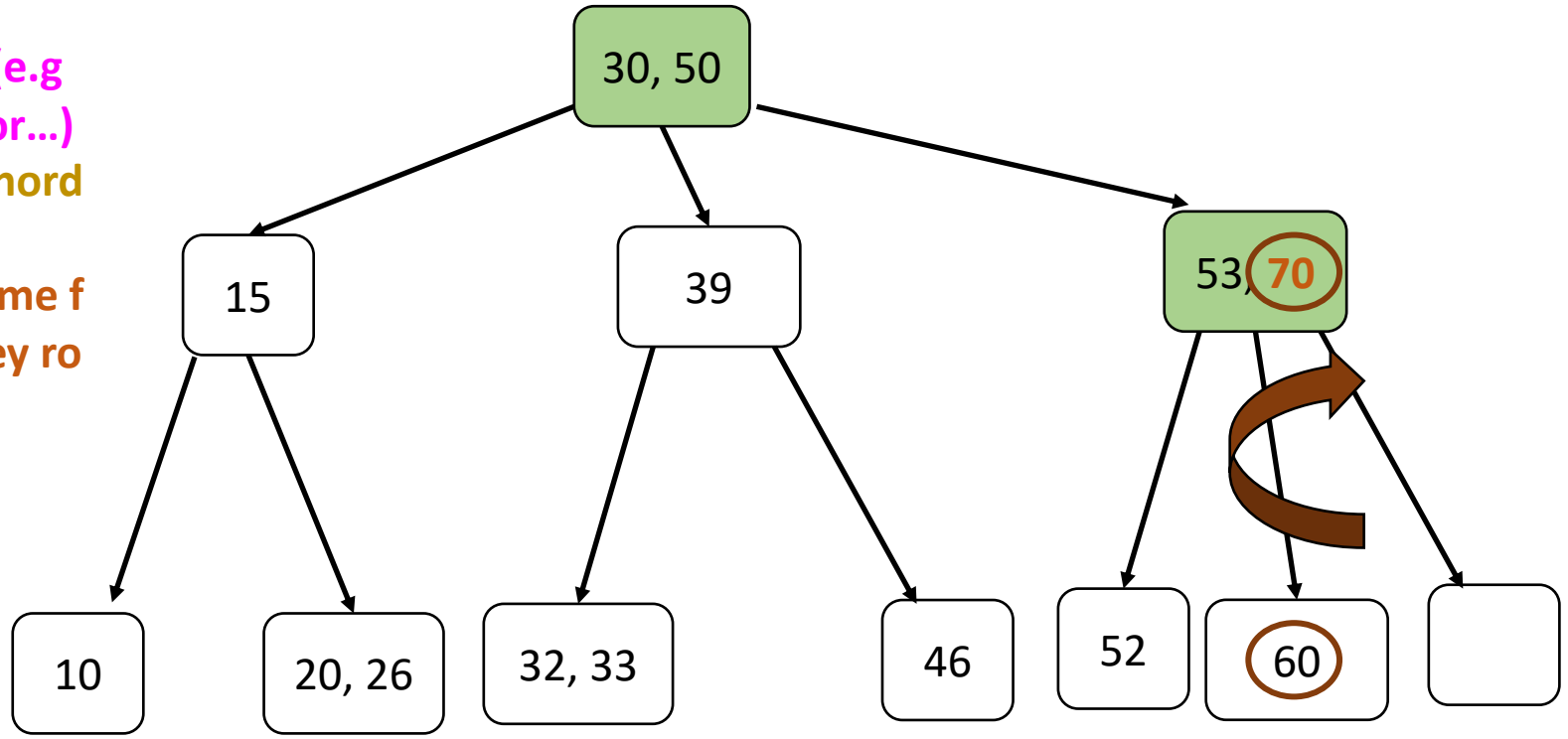
3. Have to delete the inor der succesor



Deletion

Delete 66

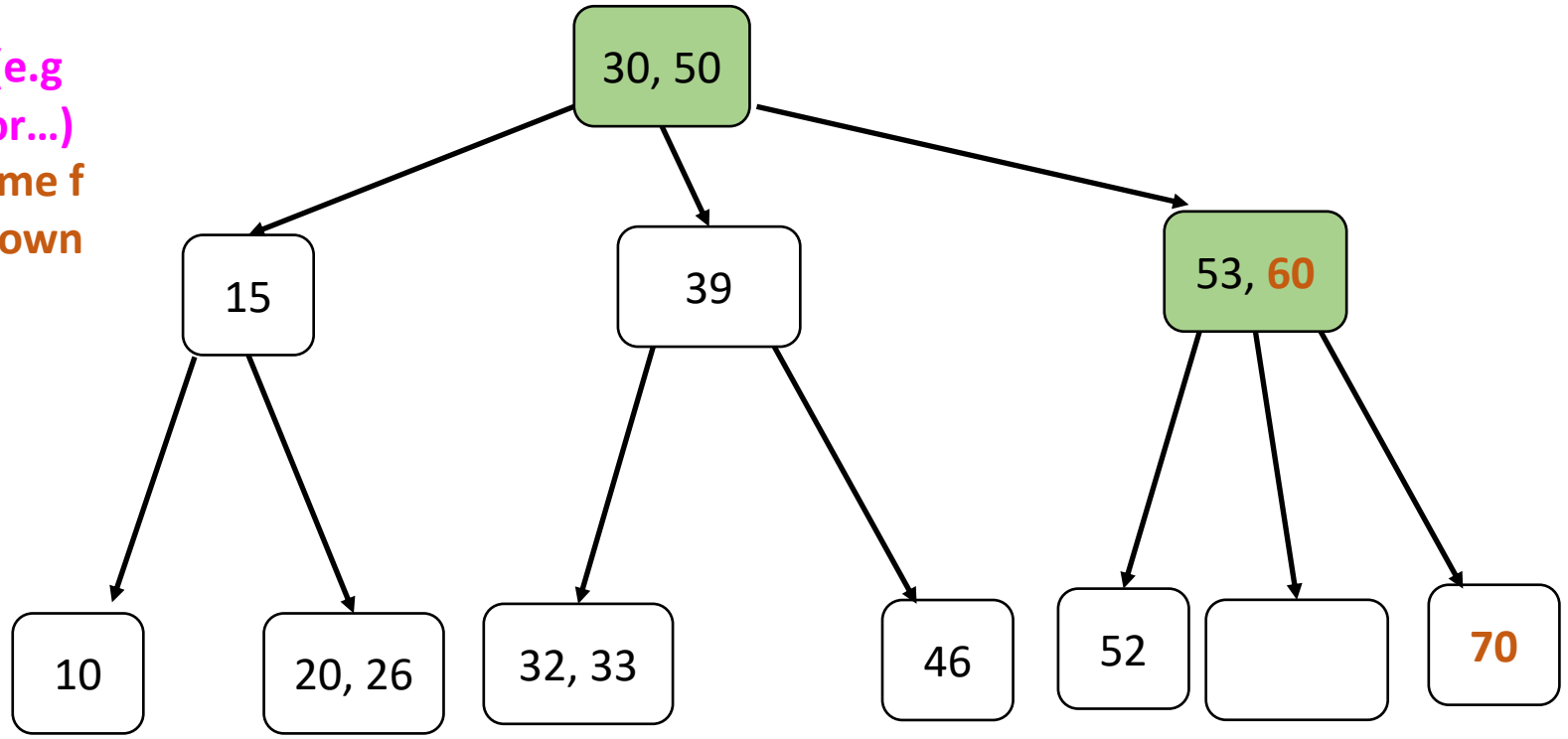
1. **Search**
2. **Have to replace key (e.g with inorder successor…)**
3. **. Have to delete the inorder succesor**
4. **Use our familiar scheme from before to do a key rotation….**



Deletion

Delete 66

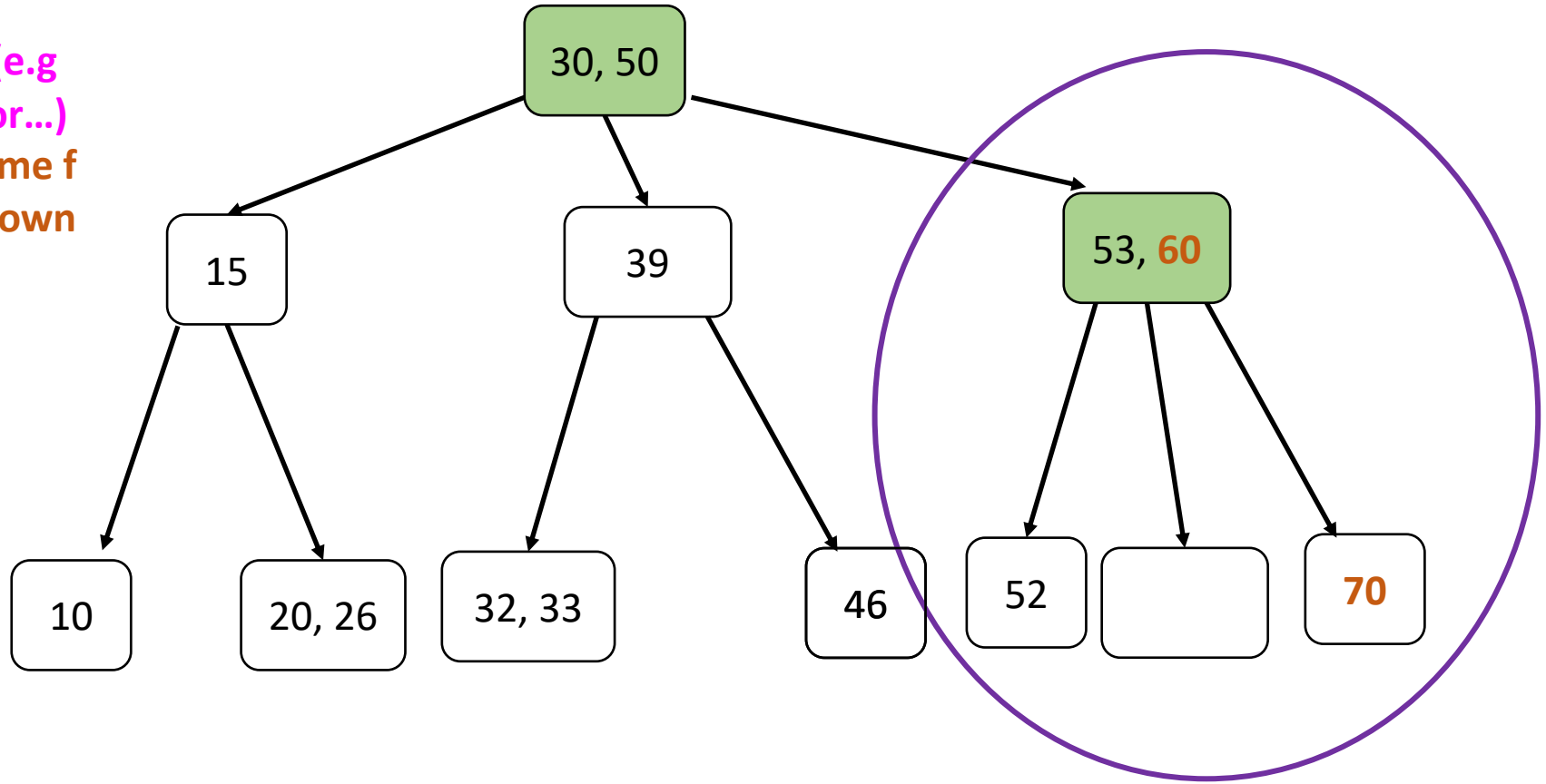
1. **Search**
2. **Have to replace key (e.g with inorder successor…)**
3. **Use our familiar scheme f rom before to push down 70 (again)**



Deletion

Delete 66

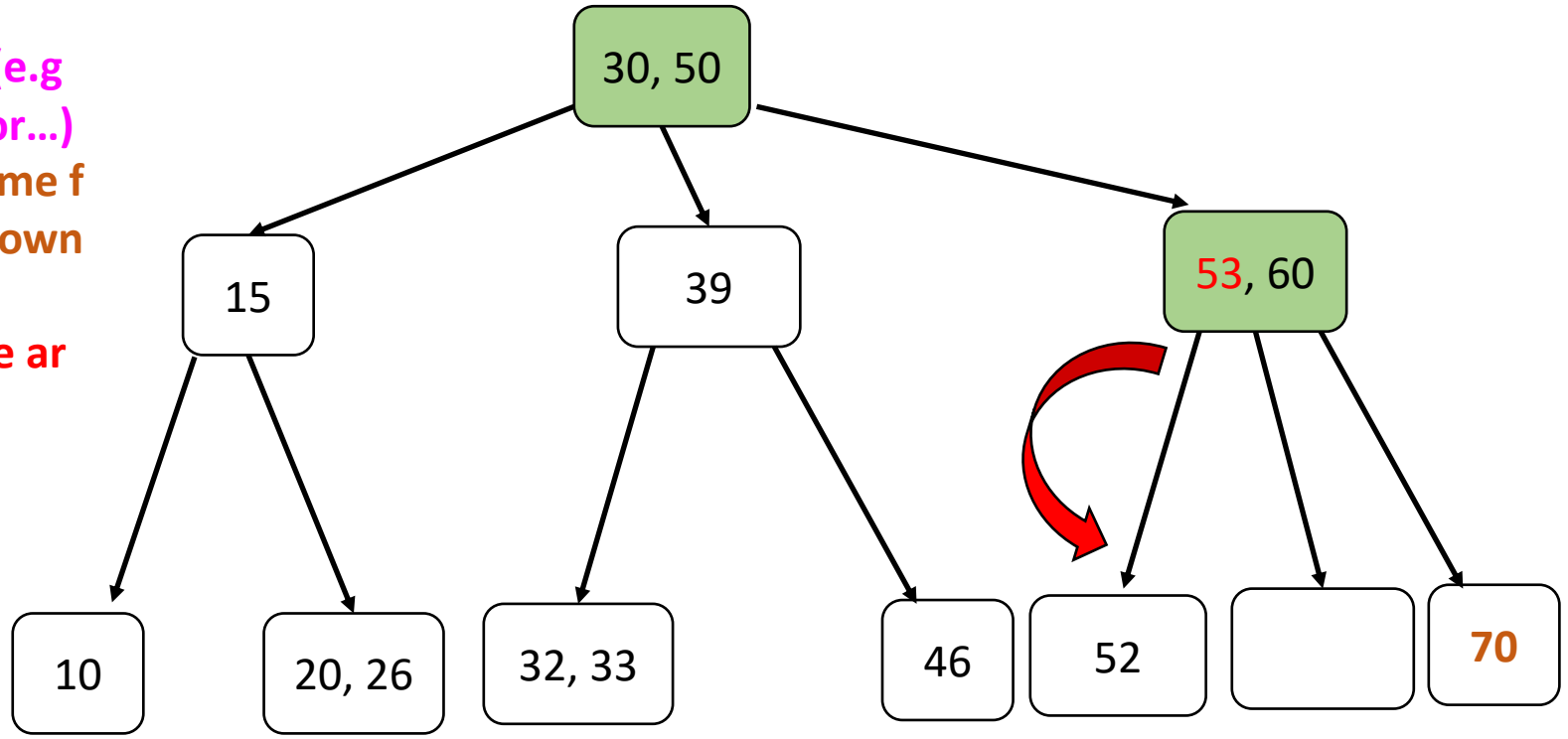
1. **Search**
2. **Have to replace key (e.g with inorder successor…)**
3. **Use our familiar scheme f rom before to push down 70 (again)**



Deletion

Delete 66

1. **Search**
2. **Have to replace key (e.g with inorder successor…)**
3. **Use our familiar scheme f rom before to push down 70 (again)**
4. **Rotate 53 left and we ar e done.**

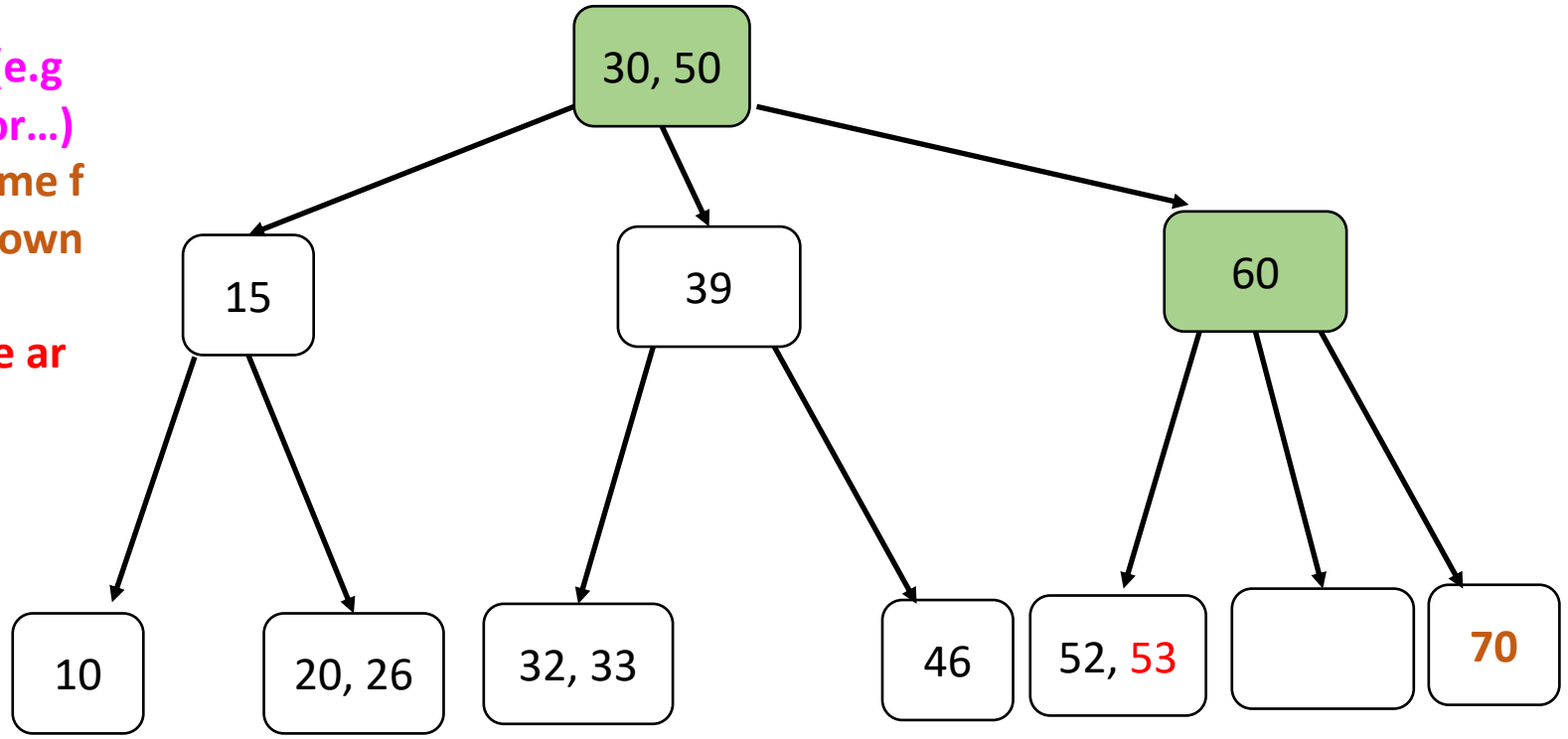


We have to merge the siblings because in this case, no key rotation from the midd

Deletion

Delete 66

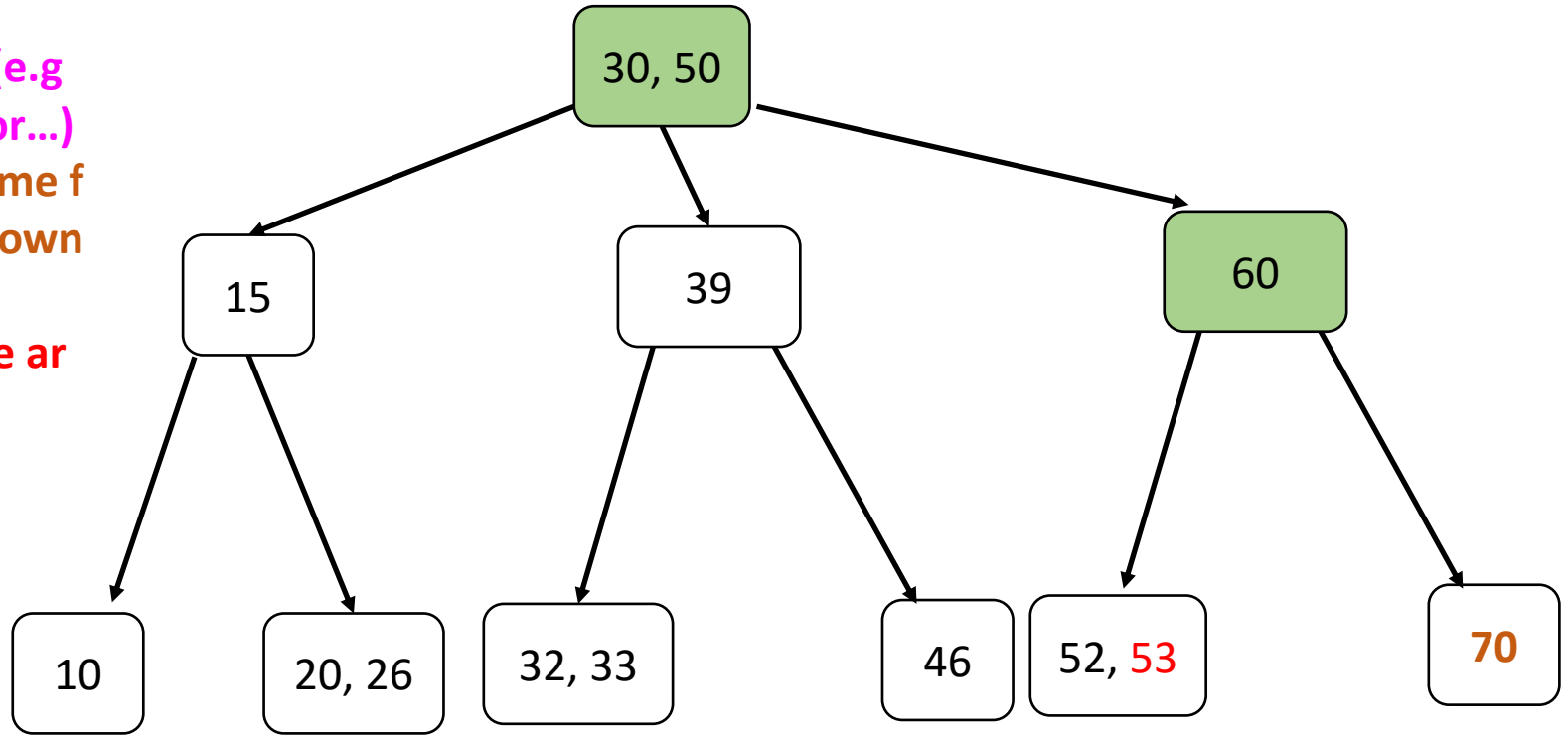
1. **Search**
2. **Have to replace key (e.g with inorder successor…)**
3. **Use our familiar scheme f rom before to push down 70 (again)**
4. **Rotate 53 left and we ar e done.**



Deletion

Delete 66

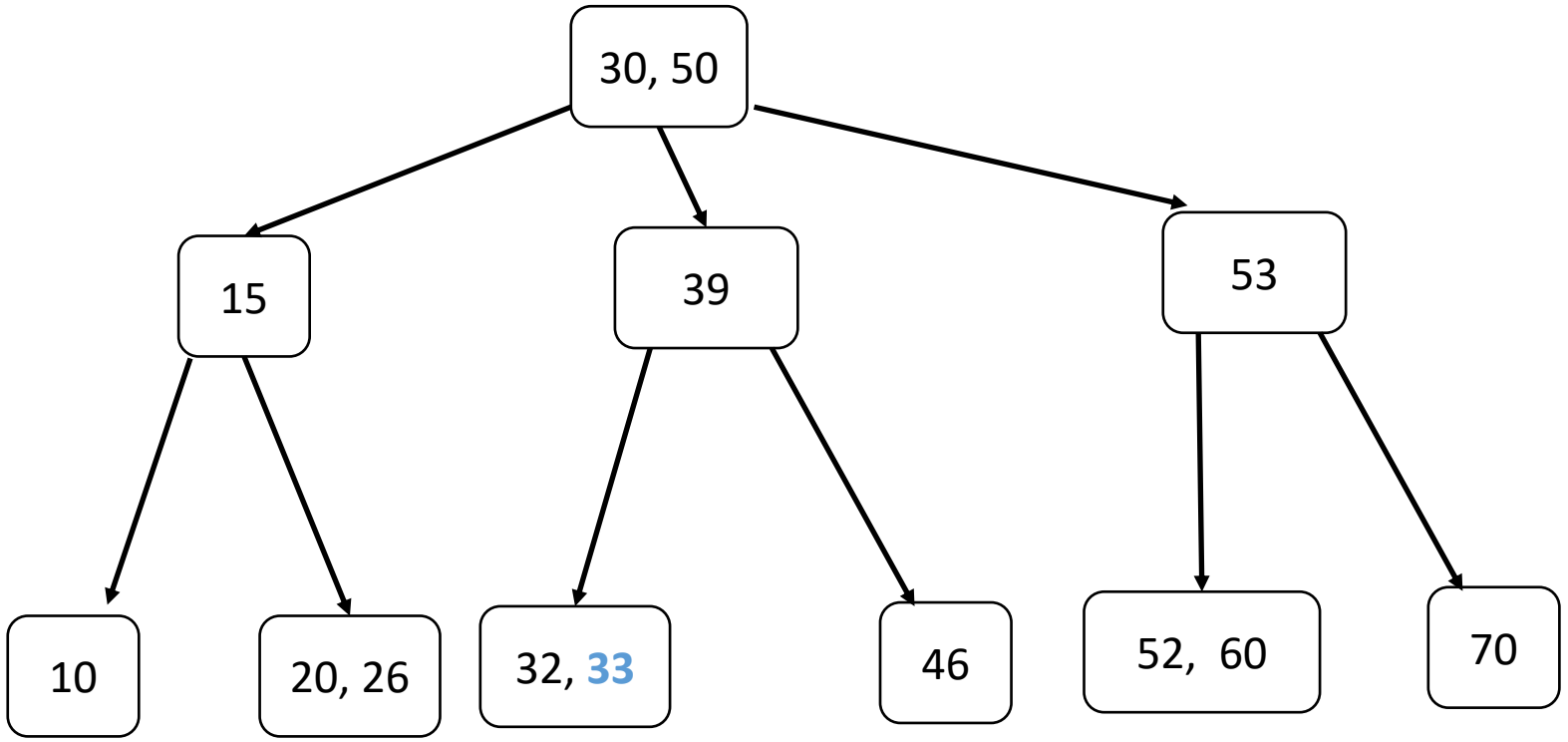
1. **Search**
2. **Have to replace key (e.g with inorder successor…)**
3. **Use our familiar scheme f rom before to push down 70 (again)**
4. **Rotate 53 left and we ar e done.**



DONE ☺

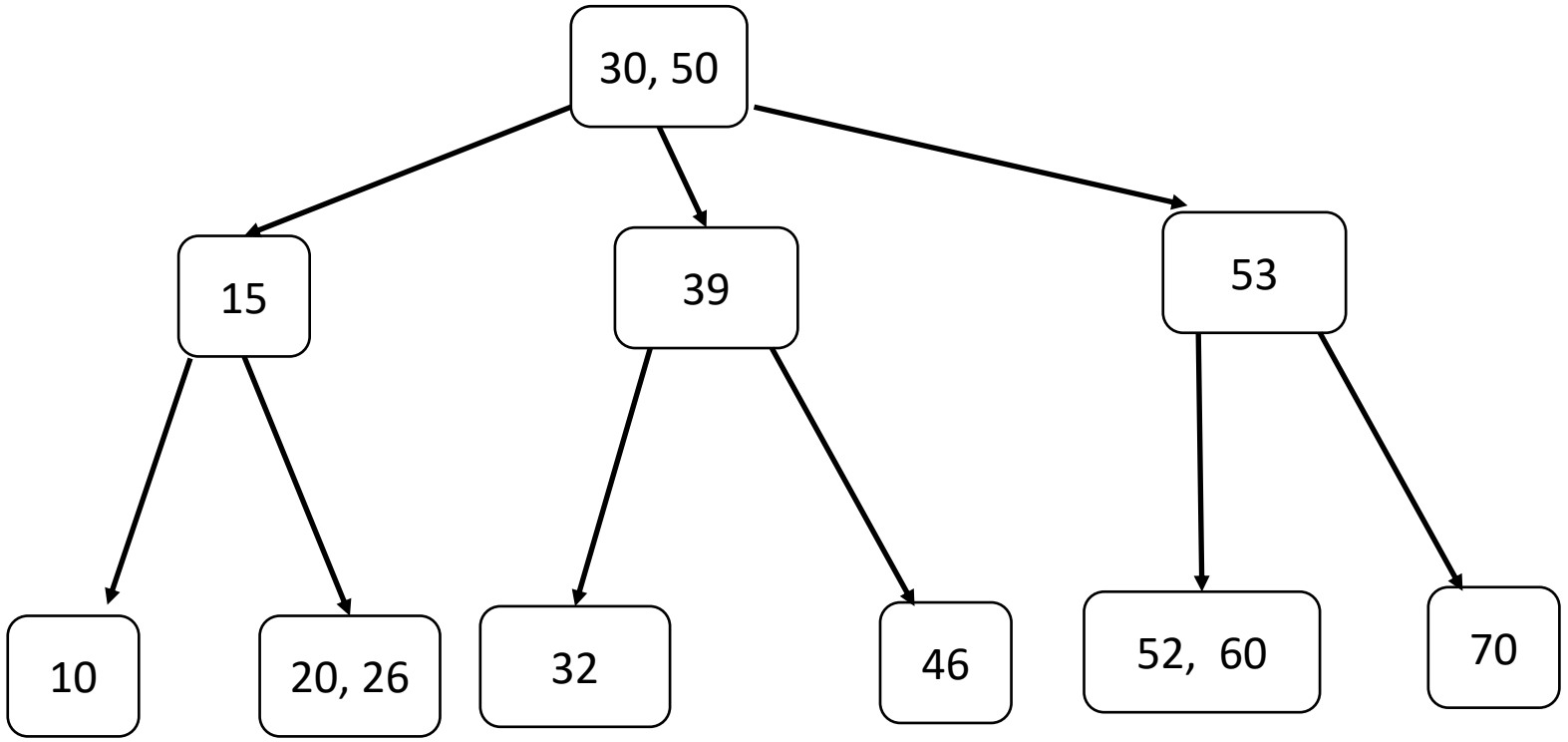
Deletion

Let's delete 33 real quick to make a point…



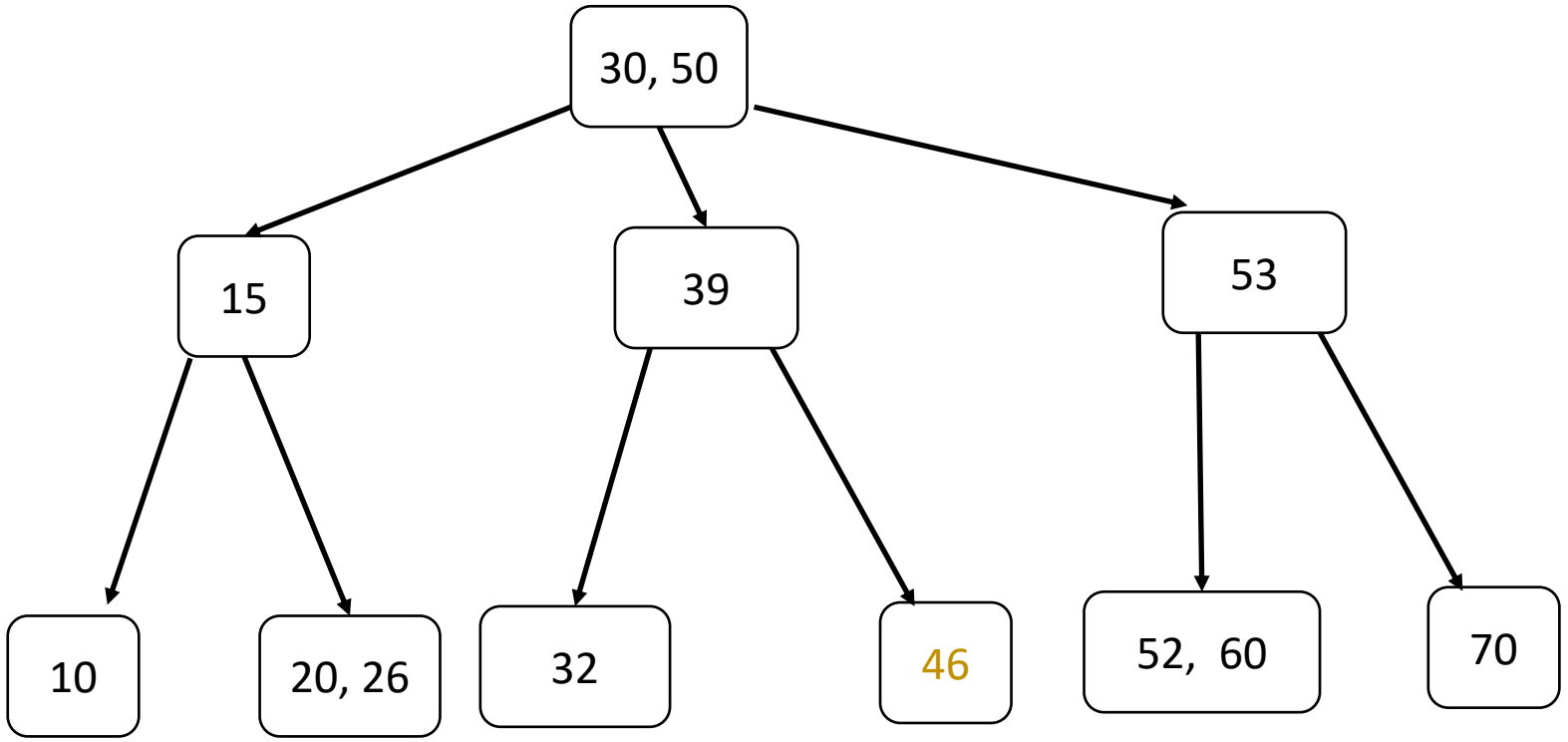
Deletion

Let's delete 33 real quick to make a point…



Deletion

Let's delete 33 real quick to make a point…
And now let's delete 46.

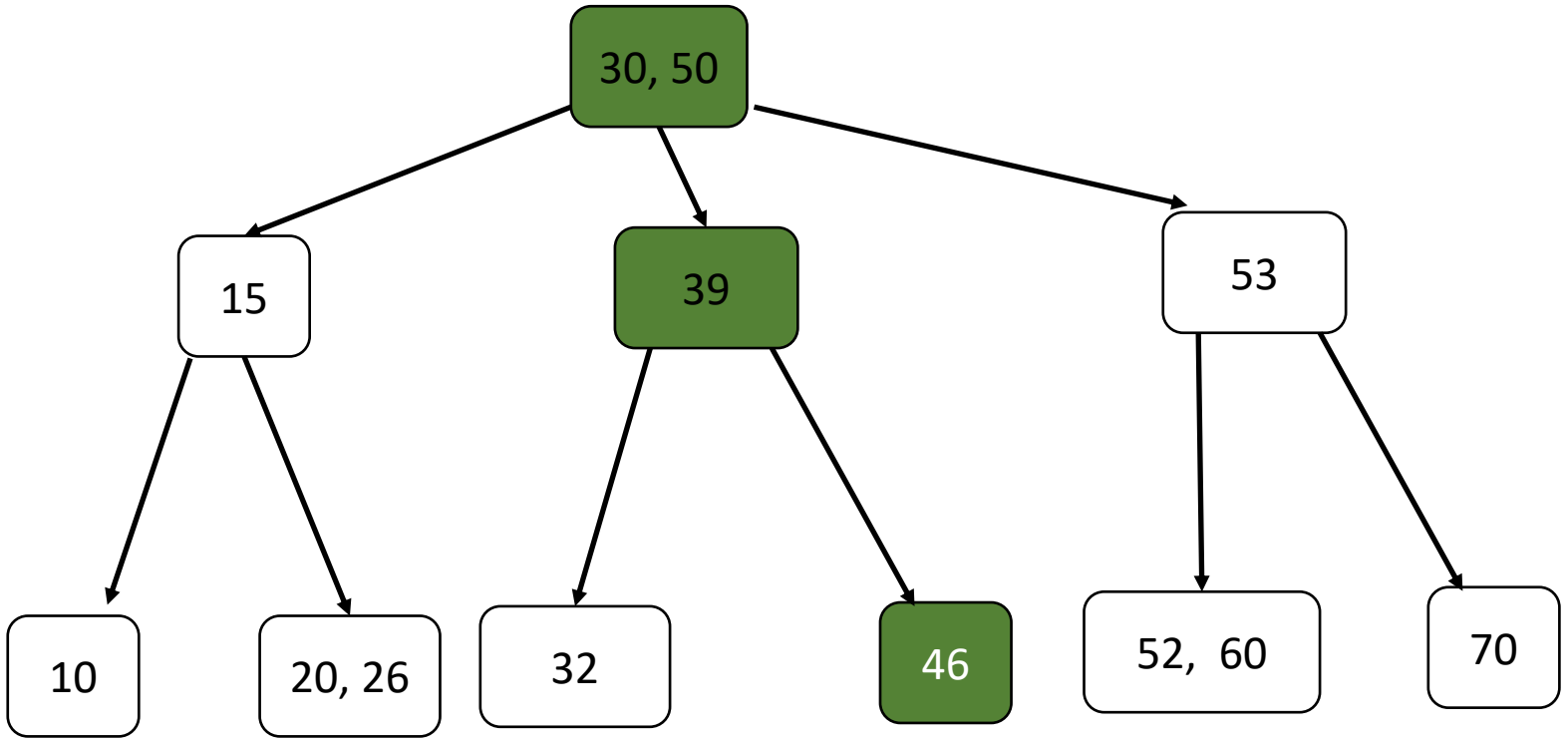


Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. Search

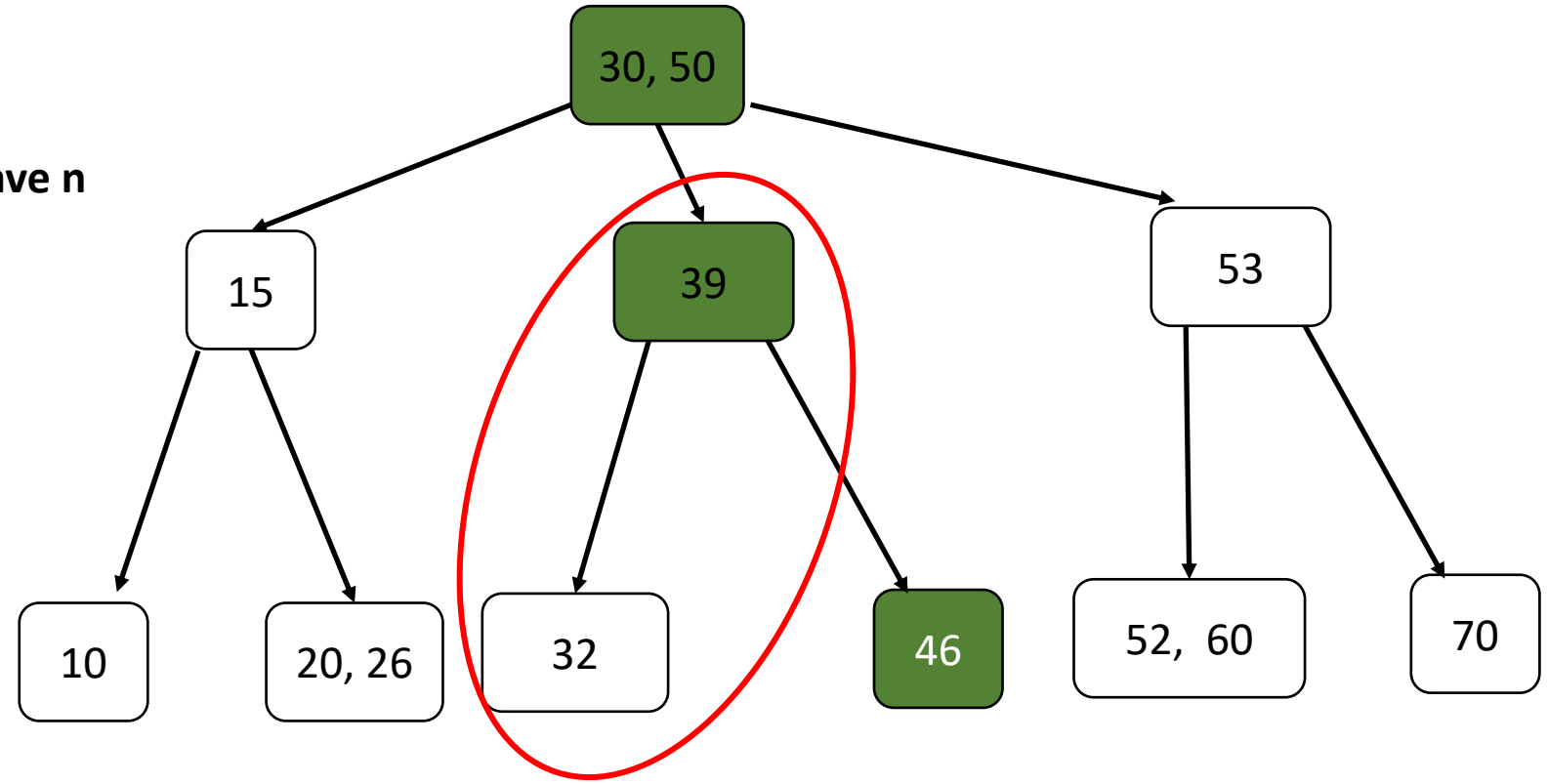


Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. **Search**
2. **Sibling and parent have n o keys to spare! ☹**

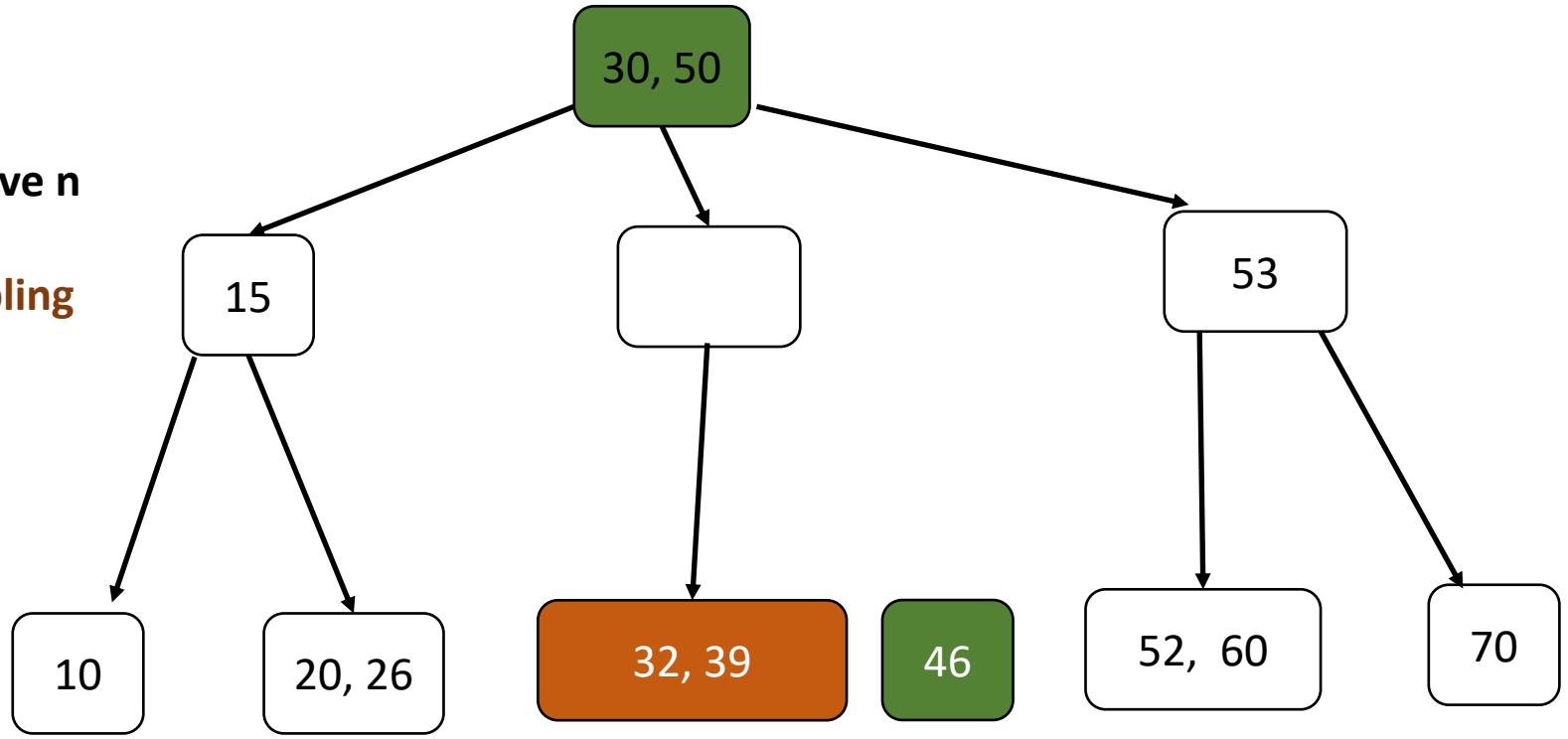


Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. **Search**
2. **Sibling and parent have n o keys to spare! ☹**
3. **Merge parent and sibling into a 3-node…**

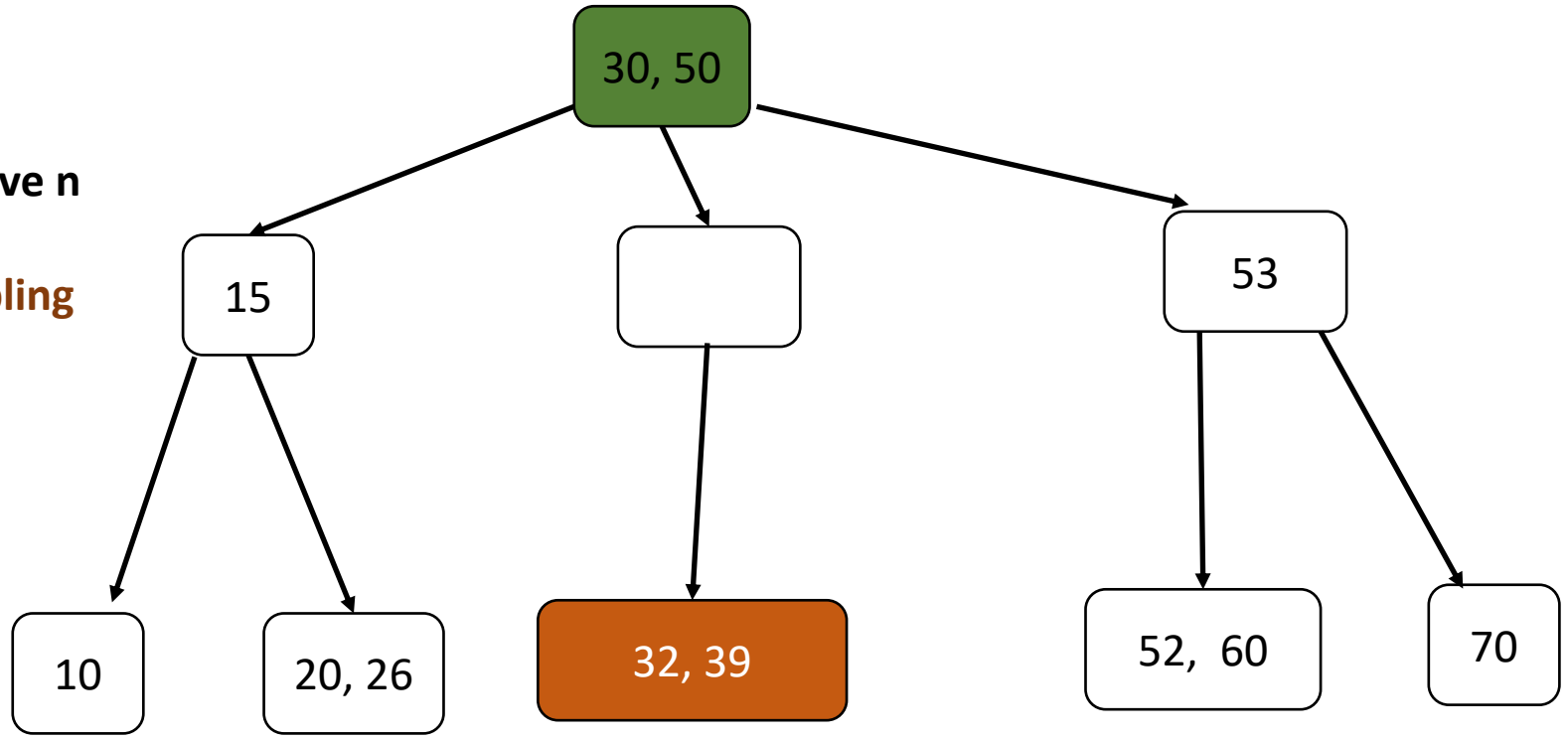


Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. **Search**
2. **Sibling and parent have n o keys to spare! ☹**
3. **Merge parent and sibling into a 3-node…**

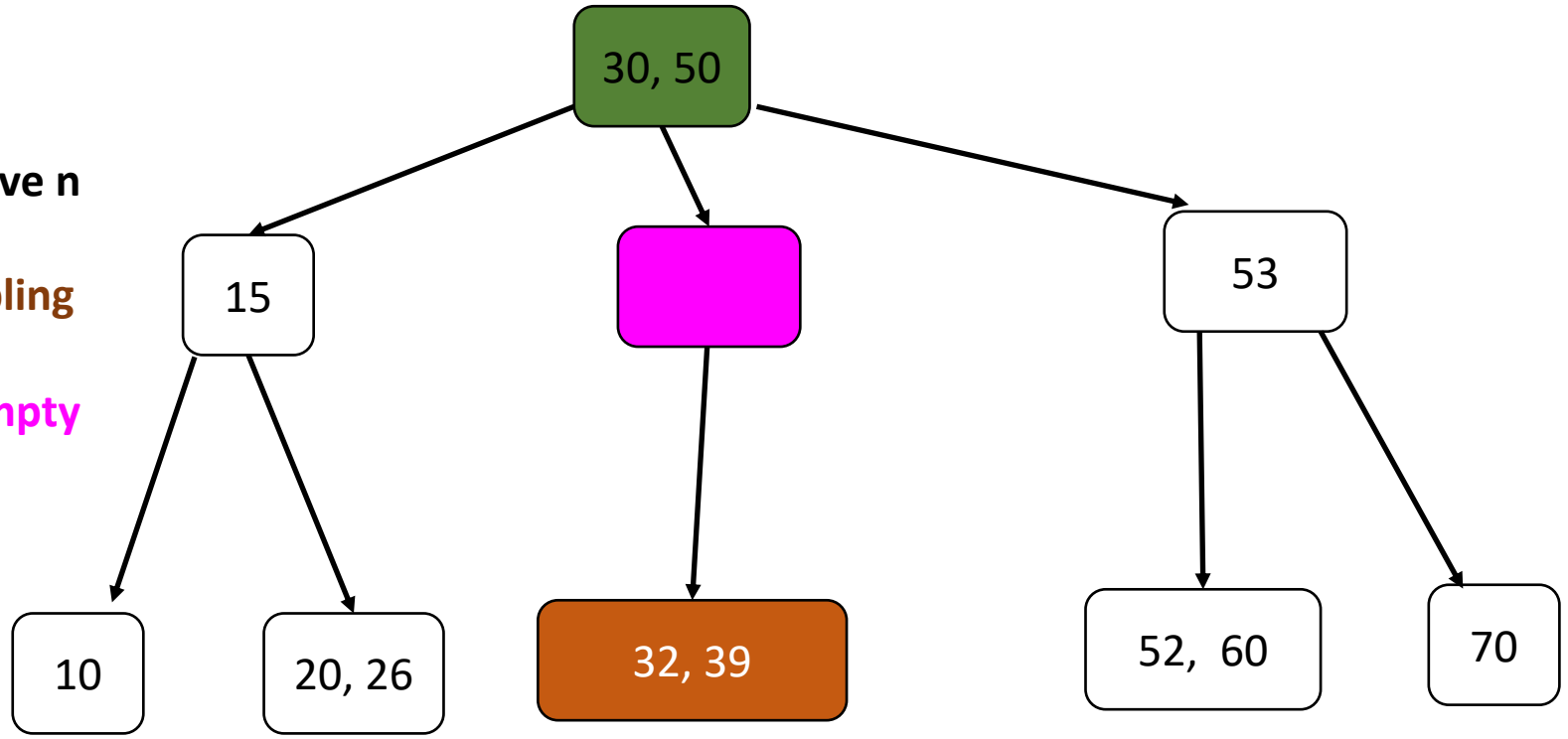


Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. **Search**
2. **Sibling and parent have n o keys to spare! ☹**
3. **Merge parent and sibling into a 3-node…**
4. **Deal with the new empty node recursively!**

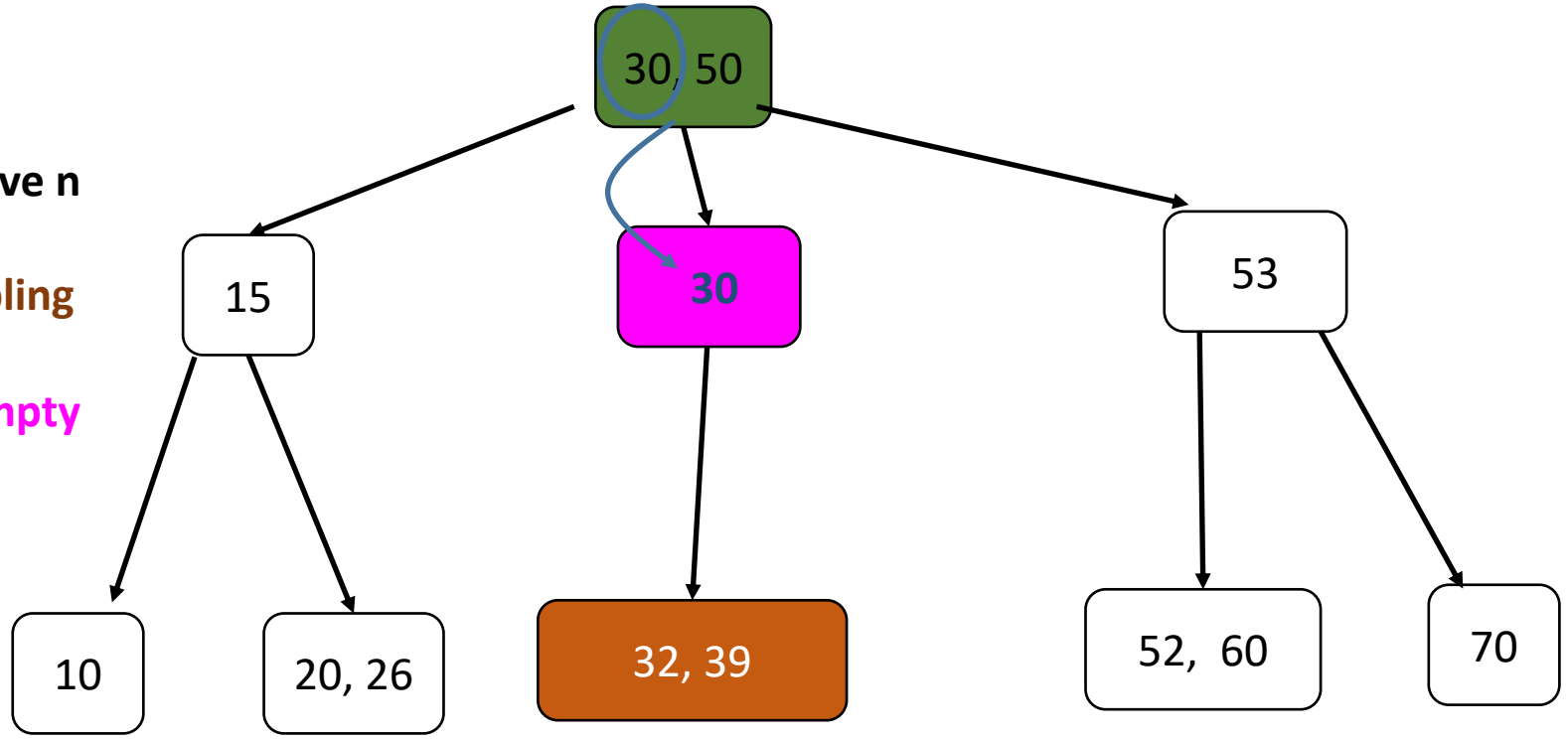


Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. Search
2. Sibling and parent have n o keys to spare! ☹
3. Merge parent and sibling into a 3-node…
4. Deal with the new empty node recursively!



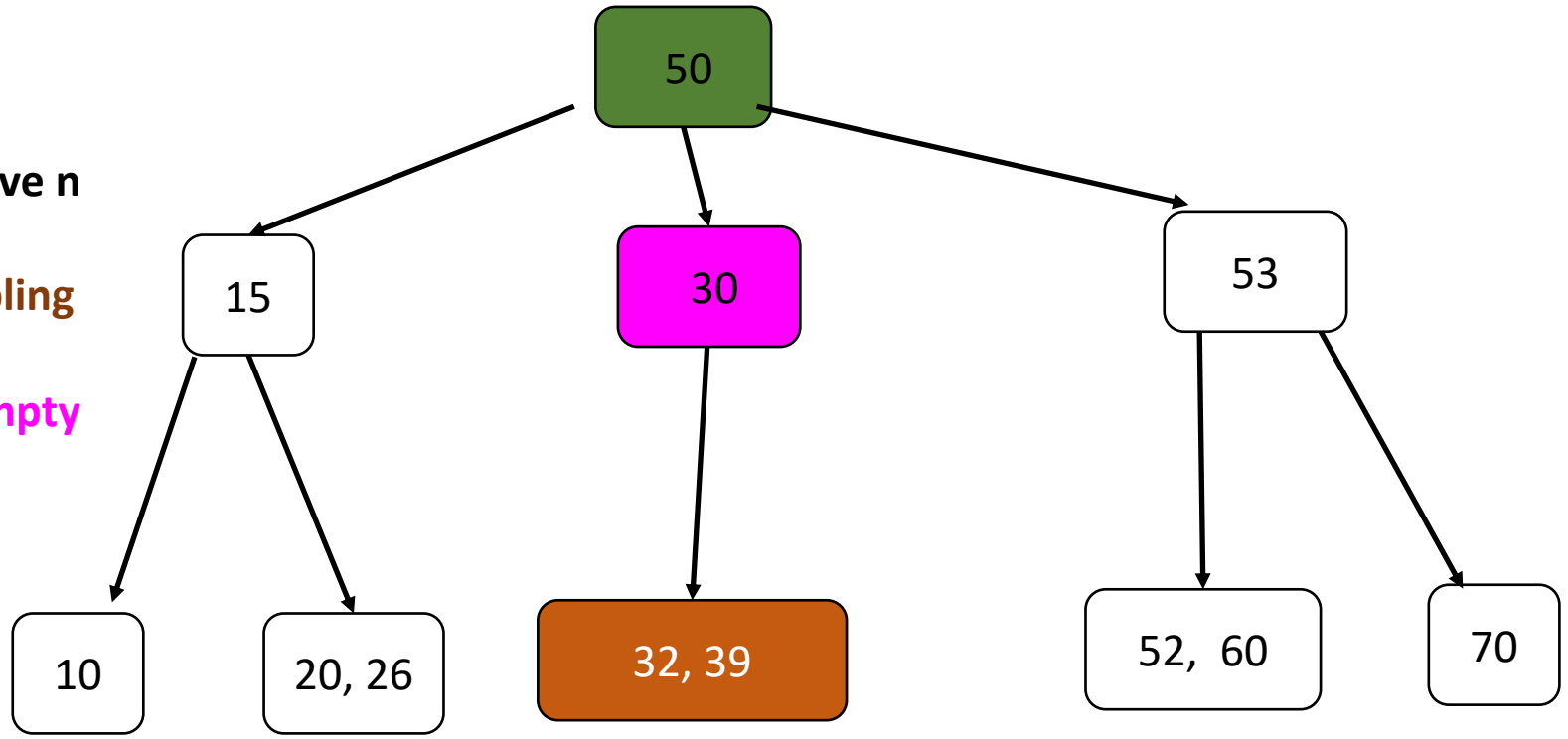
In this case, since only the root is 3-n ode, borrow appropiate key from pa rent and merge node with sibling int o a three-node!

Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. Search
2. Sibling and parent have n o keys to spare! ☹
3. Merge parent and sibling into a 3-node…
4. Deal with the new empty node recursively!



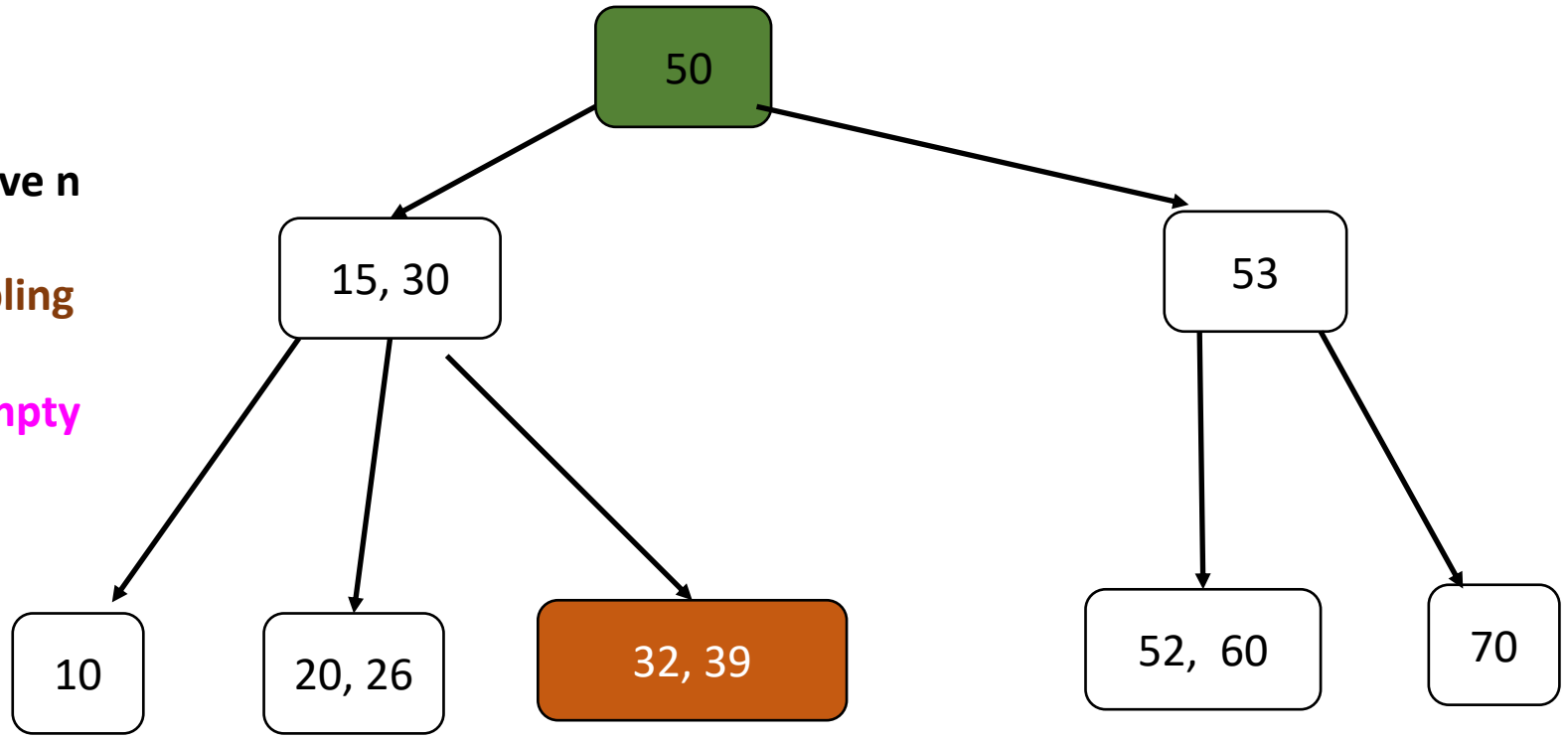
In this case, since only the root is 3-n ode, borrow appropiate key from pa rent and merge node with sibling int o a three-node!

Deletion

Let's delete 33 real quick to make a point…

And now let's delete 46.

1. Search
2. Sibling and parent have n o keys to spare! ☹
3. Merge parent and sibling into a 3-node…
4. Deal with the new empty node recursively!



In this case, since only the root is 3-n ode, borrow appropriate key from p arent and merge node with sibling in to a three-node!

Red-Black Tree

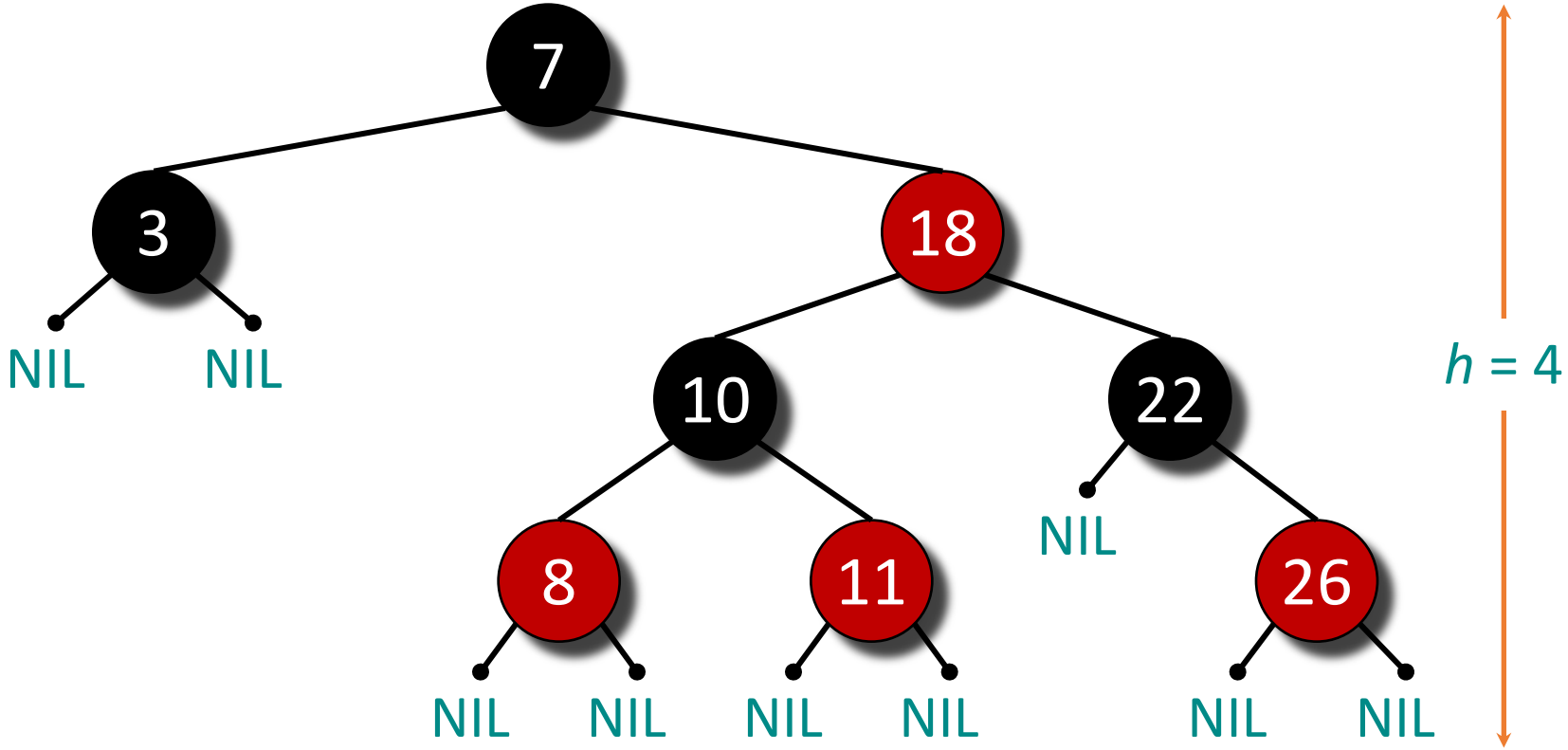
Red-black trees

This data structure requires an extra one-bit color field in each node.

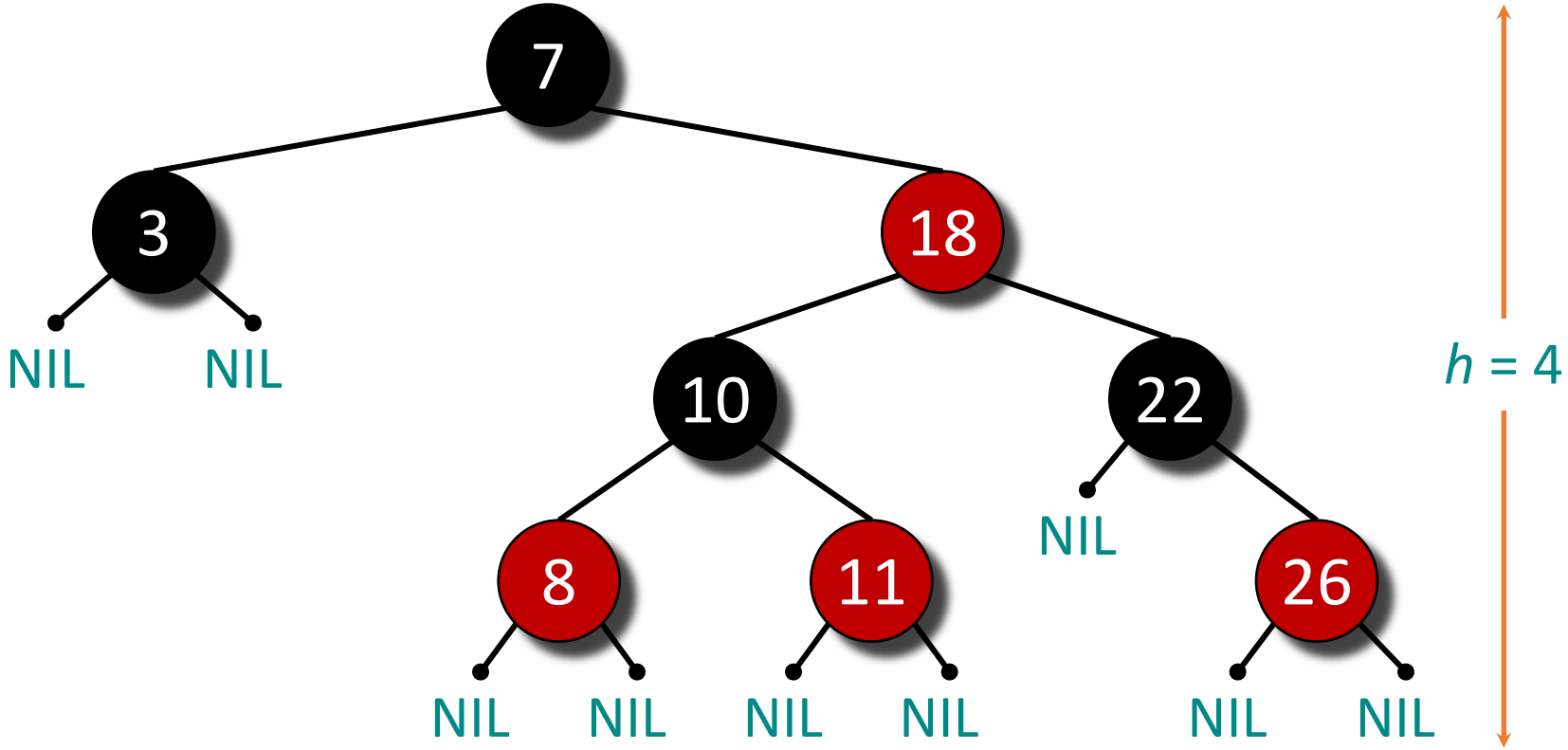
Red-black properties:

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black-height(x).

Example of a red-black tree

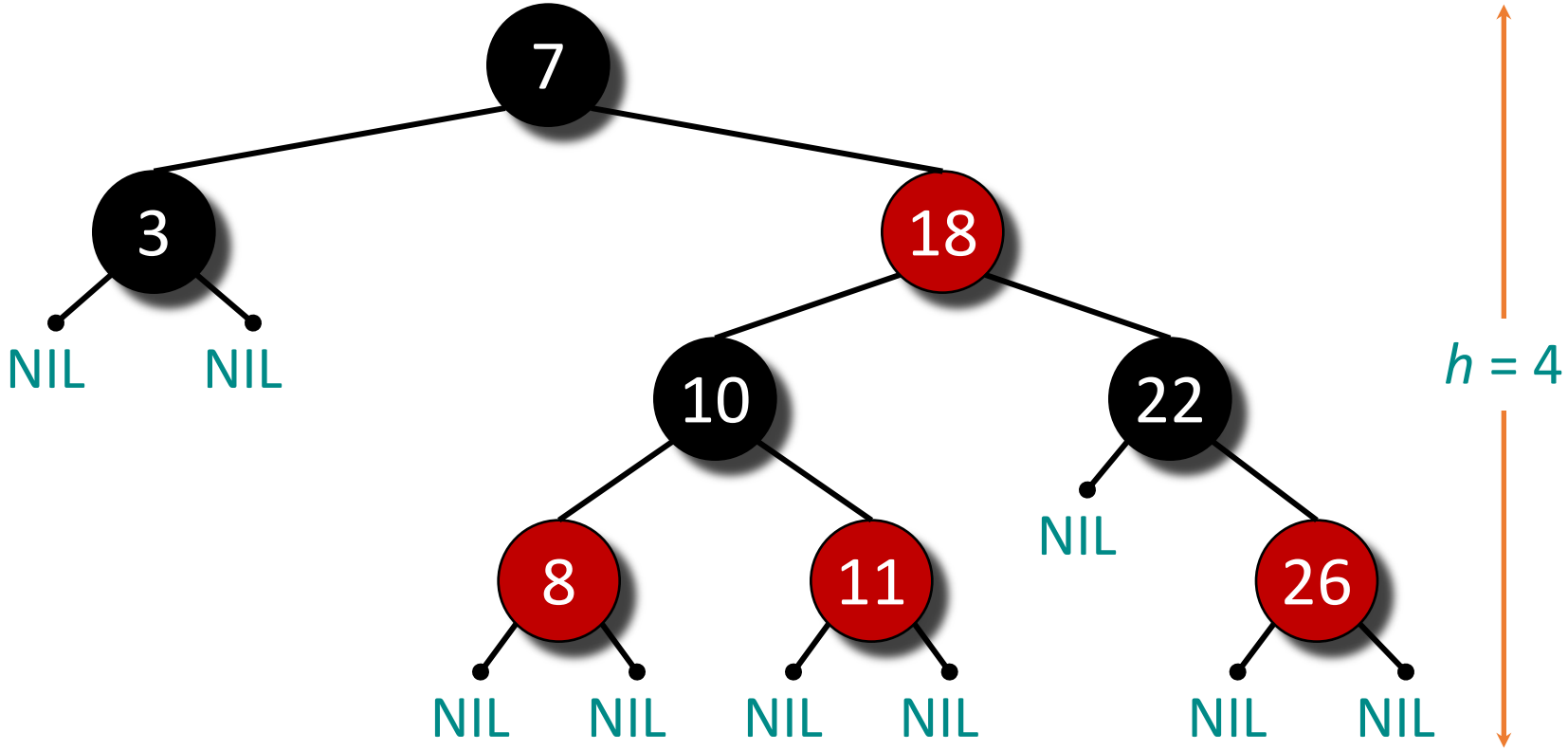


Example of a red-black tree



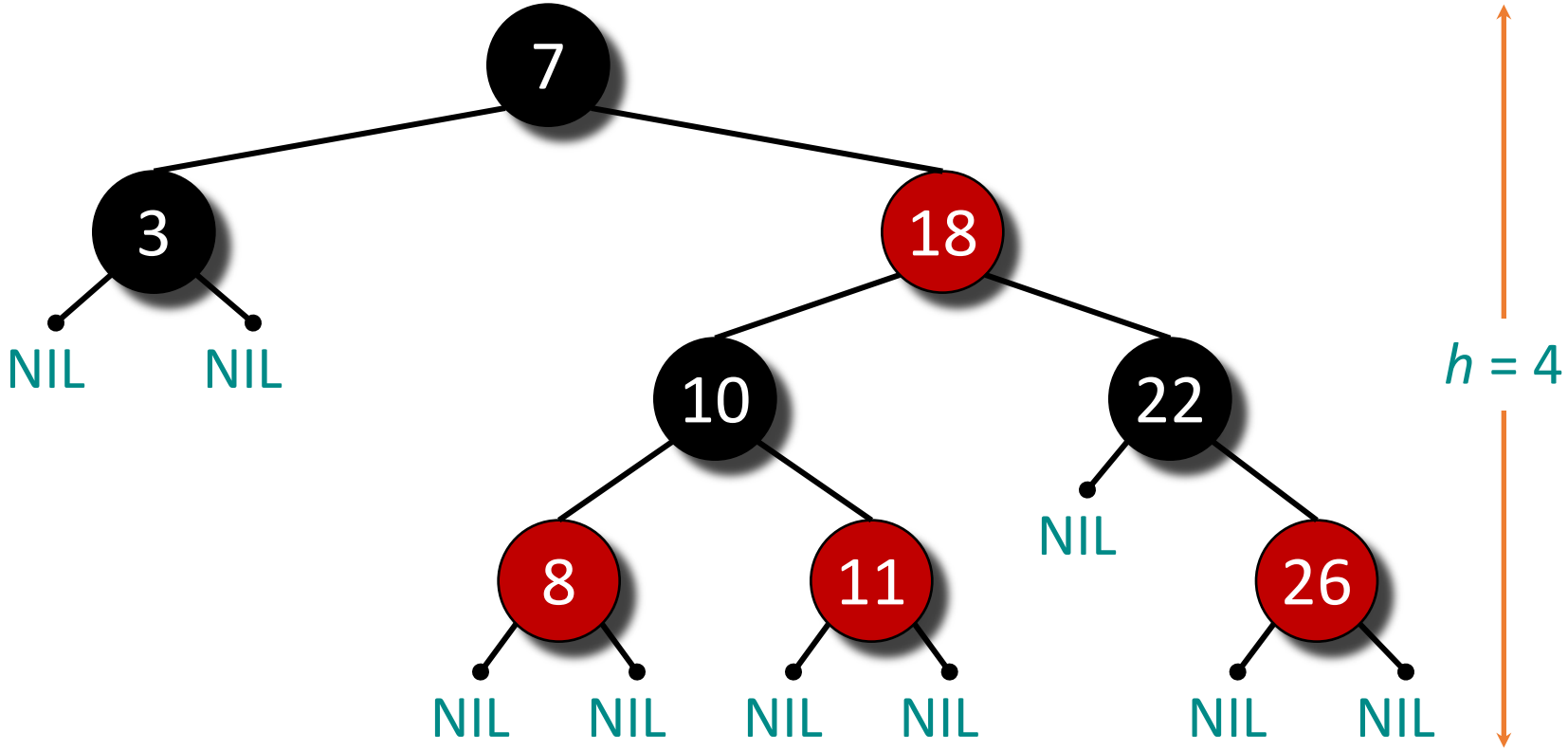
1. Every node is either red or black.

Example of a red-black tree



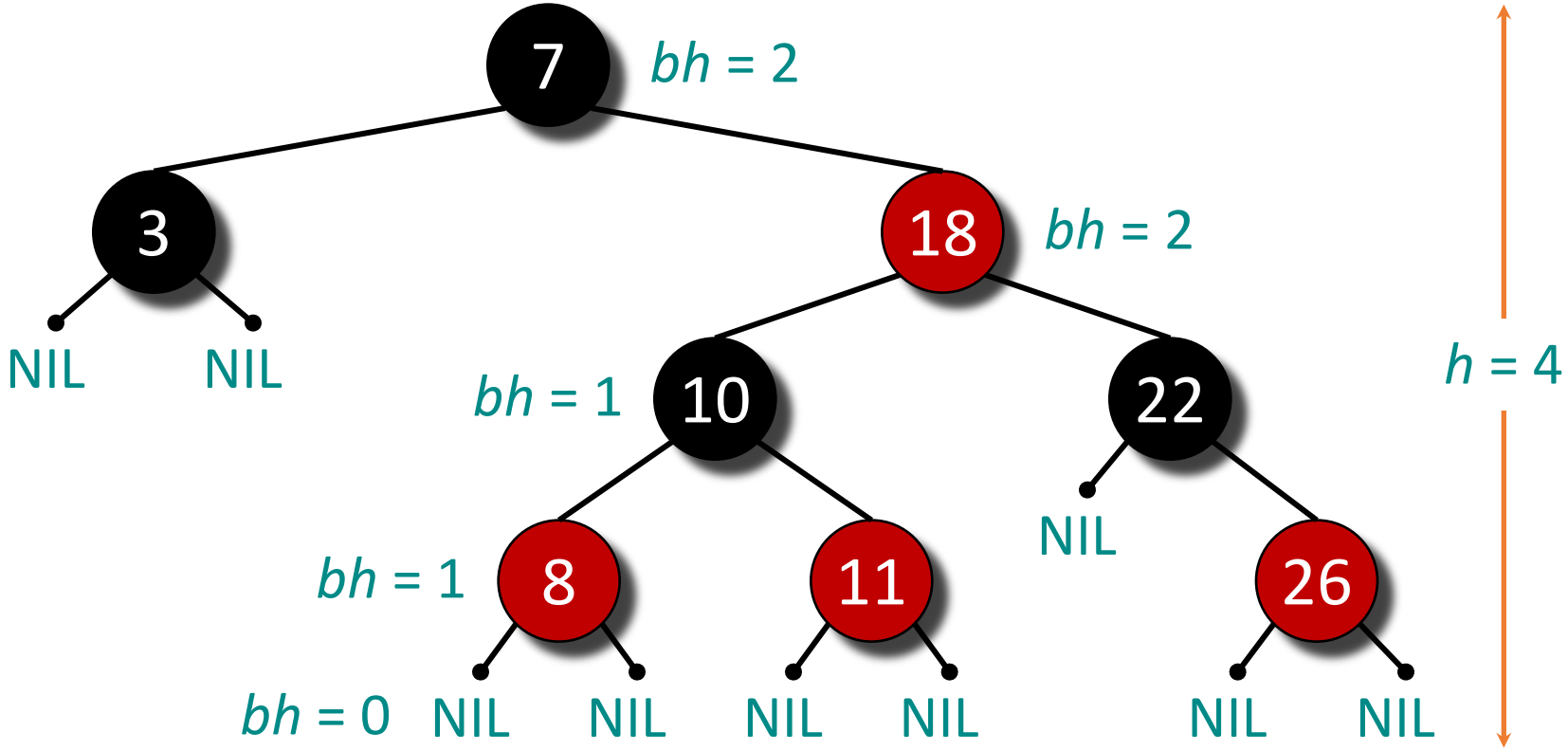
2. The root and leaves (NIL's) are black.

Example of a red-black tree



3. If a node is red, then its parent is black.

Example of a red-black tree



4. All simple paths from any node *x* to a descendant leaf have the same number of black nodes = *black-height*(*x*).

Height of a red-black tree

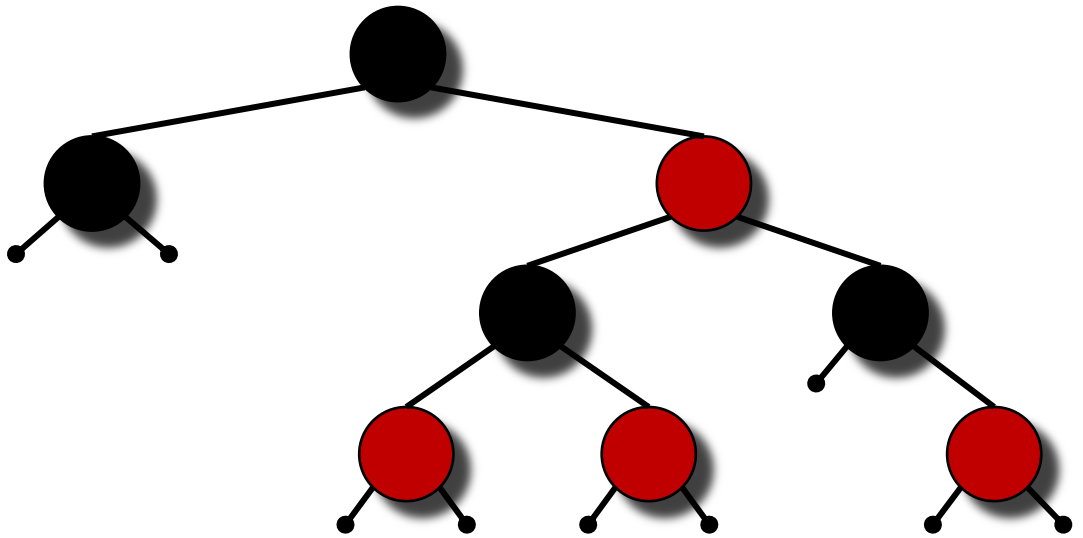
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

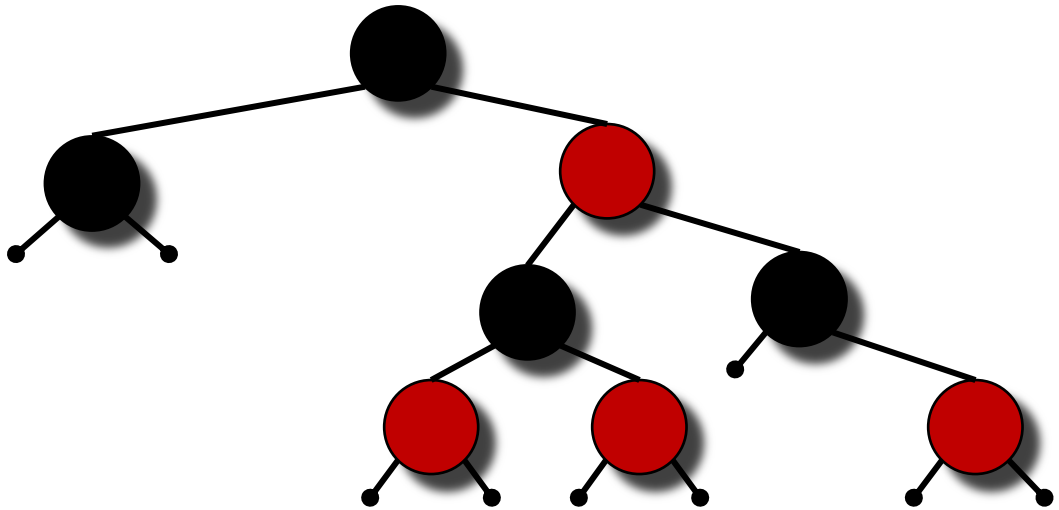
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

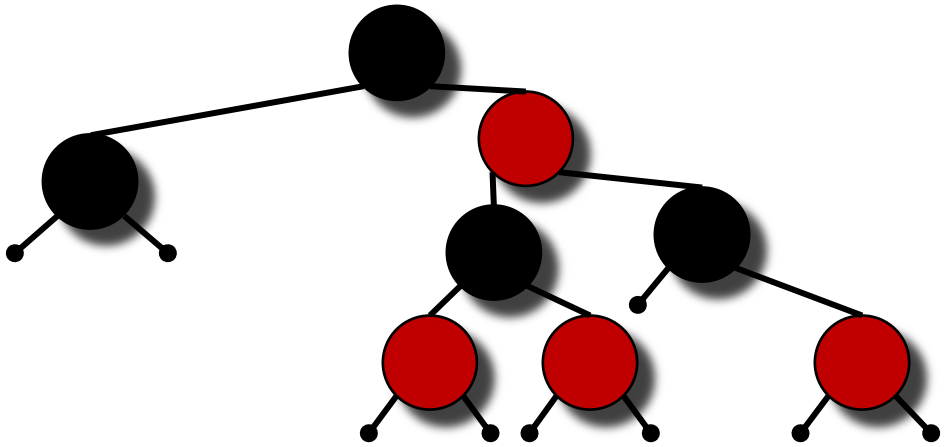
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

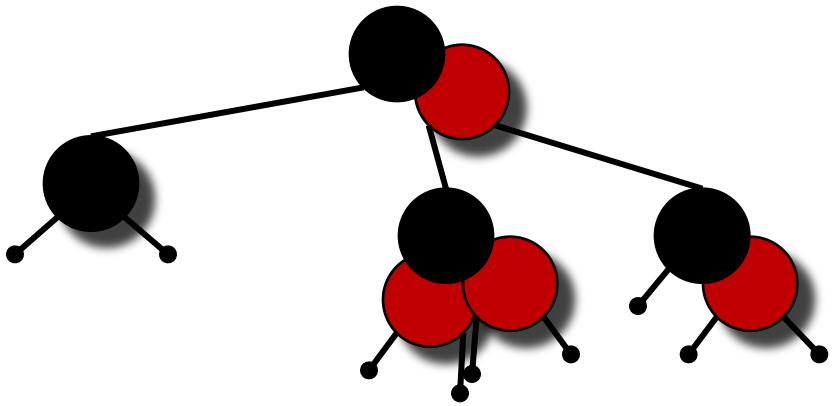
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

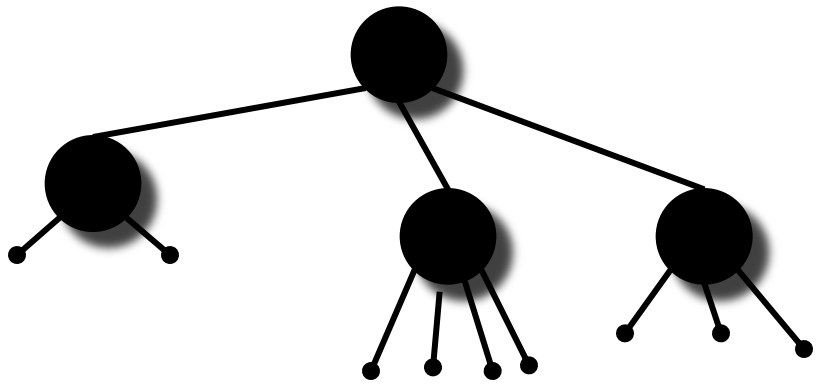
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



Height of a red-black tree

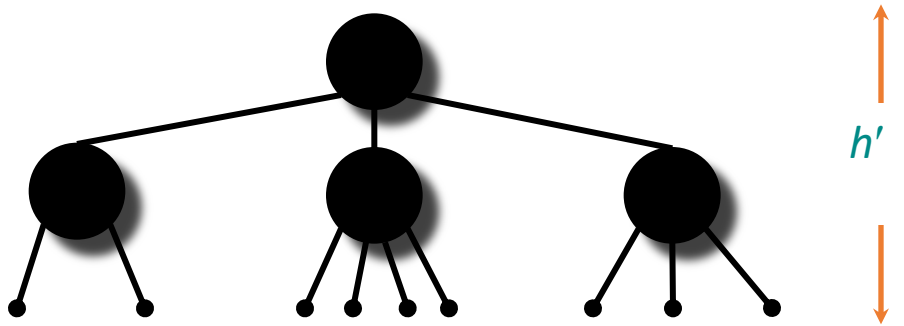
Theorem. A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1).$$

Proof. (The book uses induction. Read carefully.)

INTUITION:

- Merge red nodes into their black parents.



- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves.

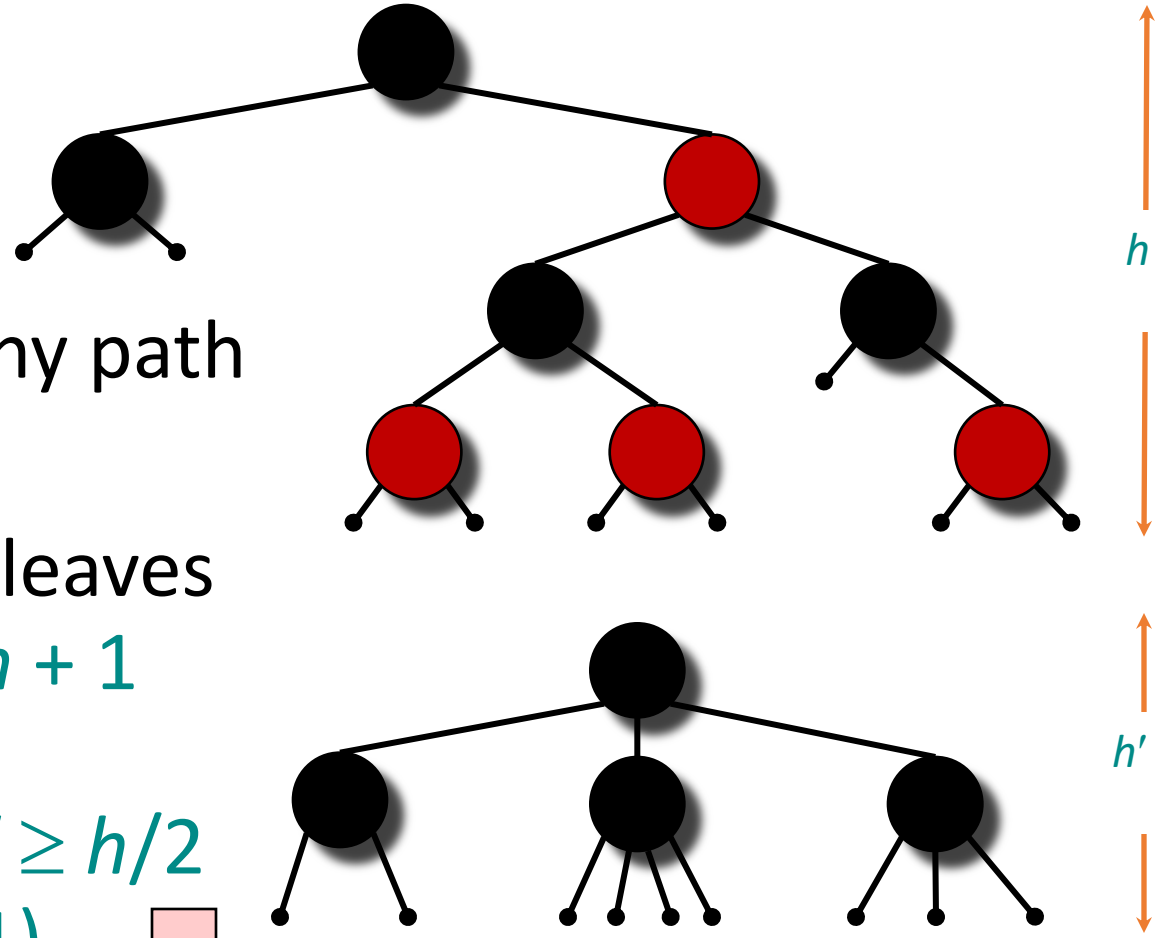
Proof (continued)

- We have $h' \geq h/2$, since at most half the leaves on any path are red.
- The number of leaves in each tree is $n + 1$

$\Rightarrow n + 1 \geq 2^{h'}$

$\Rightarrow \lg(n + 1) \geq h' \geq h/2$

$\Rightarrow h \leq 2 \lg(n + 1)$. □



Query operations

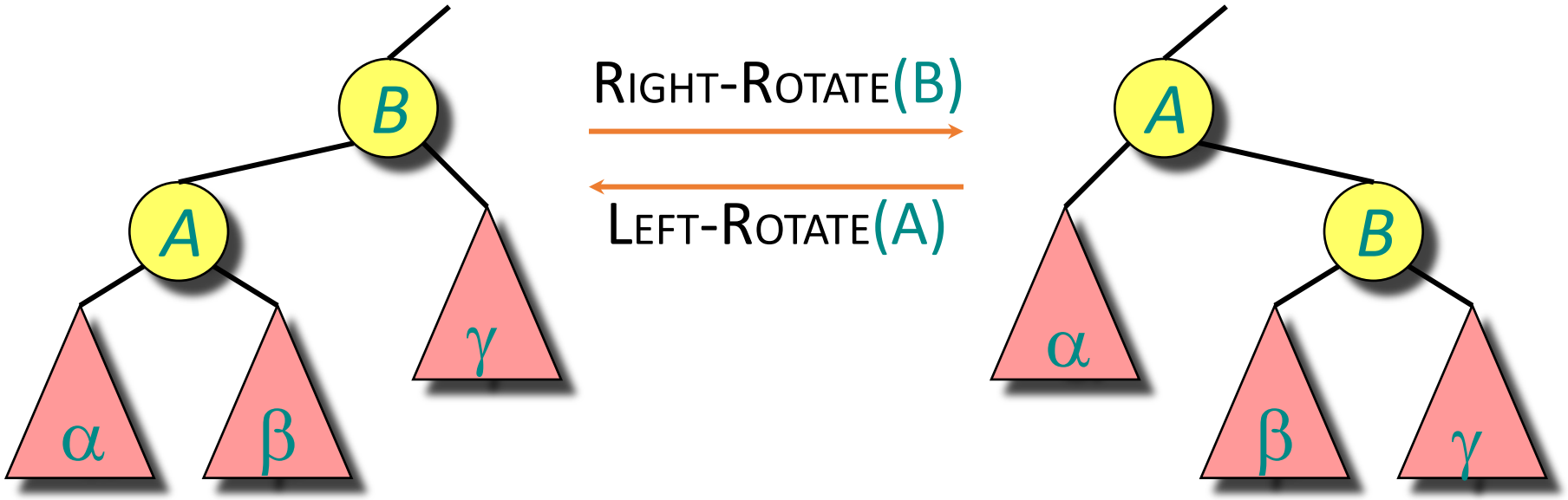
Corollary. The queries SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR all run in $O(\lg n)$ time on a red-black tree with n nodes.

Modifying operations

The operations INSERT and DELETE cause modifications to the red-black tree:

- the operation itself,
- color changes,
- restructuring the links of the tree via ***"rotations"***.

Rotations



Rotations maintain the inorder ordering of keys:

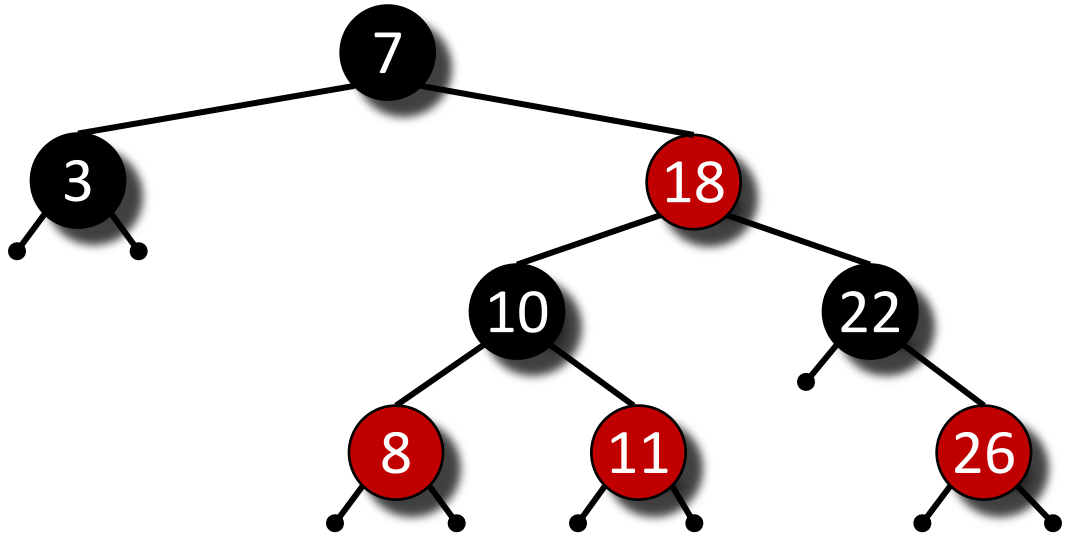
- $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c.$

A rotation can be performed in $O(1)$ time.

Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

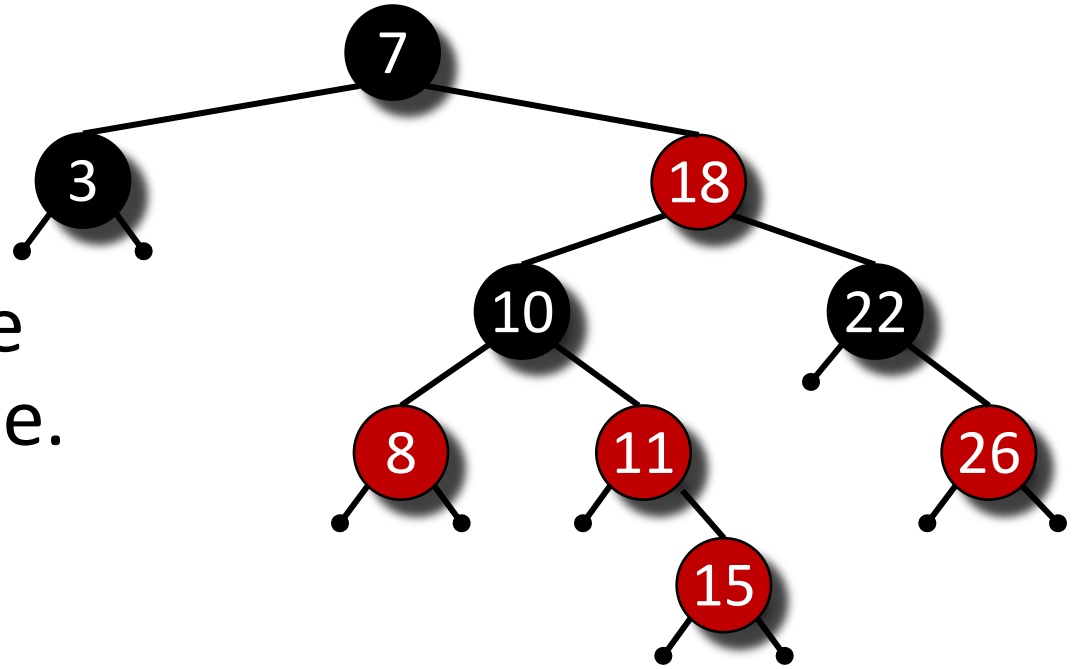


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.

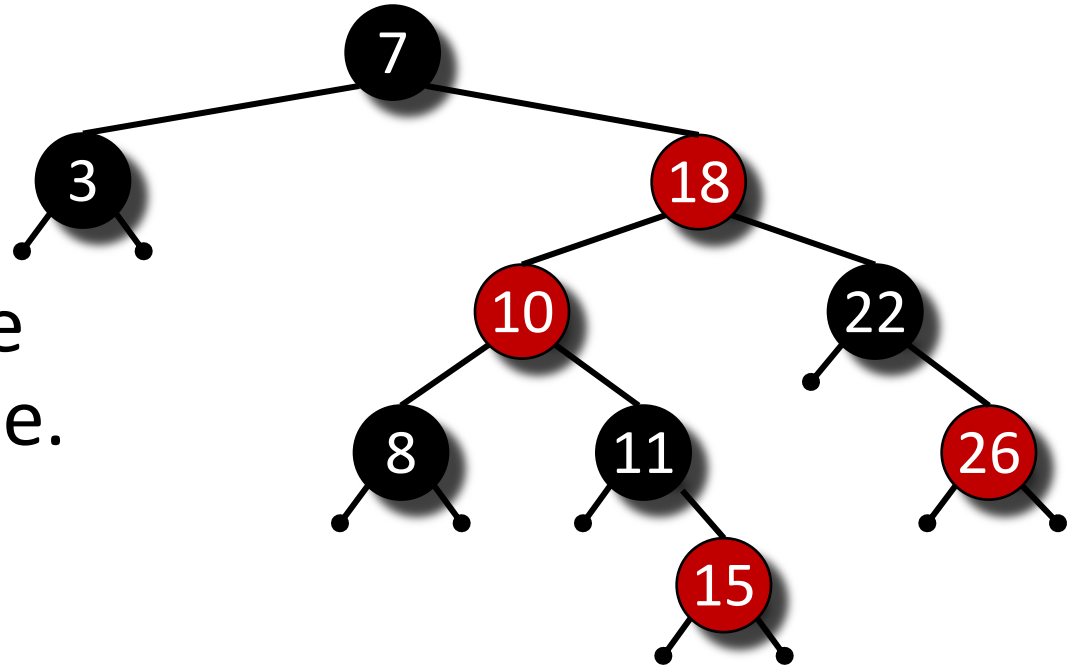


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).

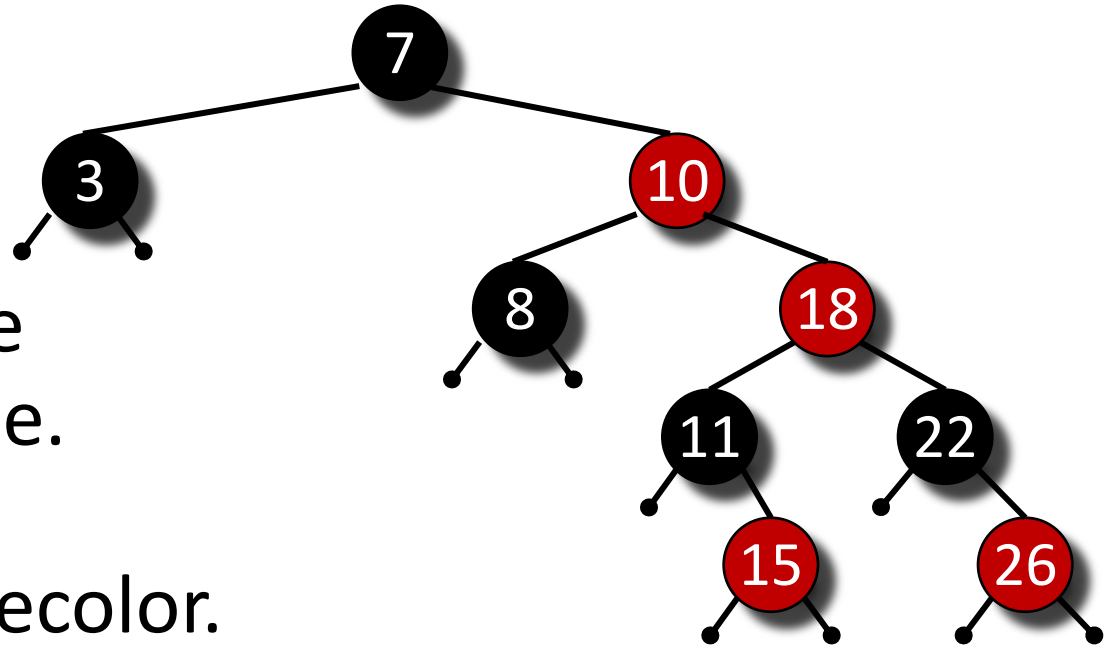


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.

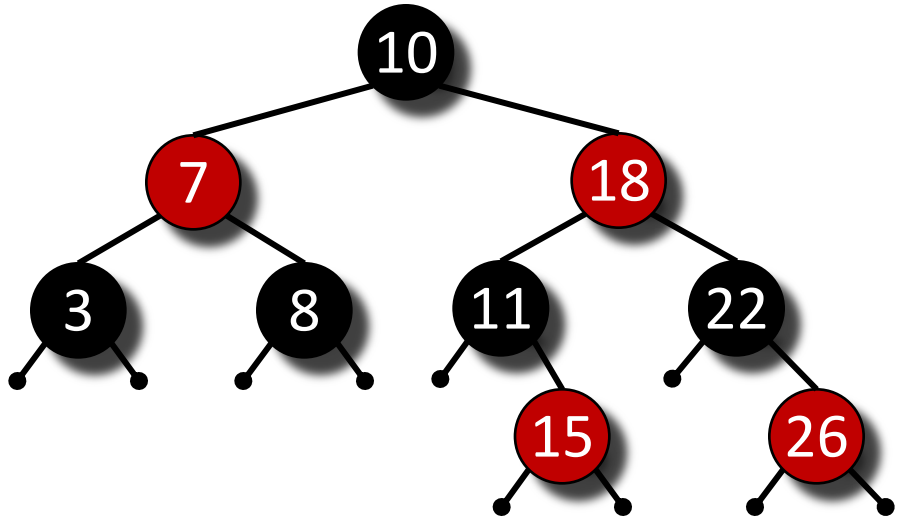


Insertion into a red-black tree

IDEA: Insert x in tree. Color x red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring.

Example:

- Insert $x = 15$.
- Recolor, moving the violation up the tree.
- RIGHT-ROTATE(18).
- LEFT-ROTATE(7) and recolor.



Pseudocode

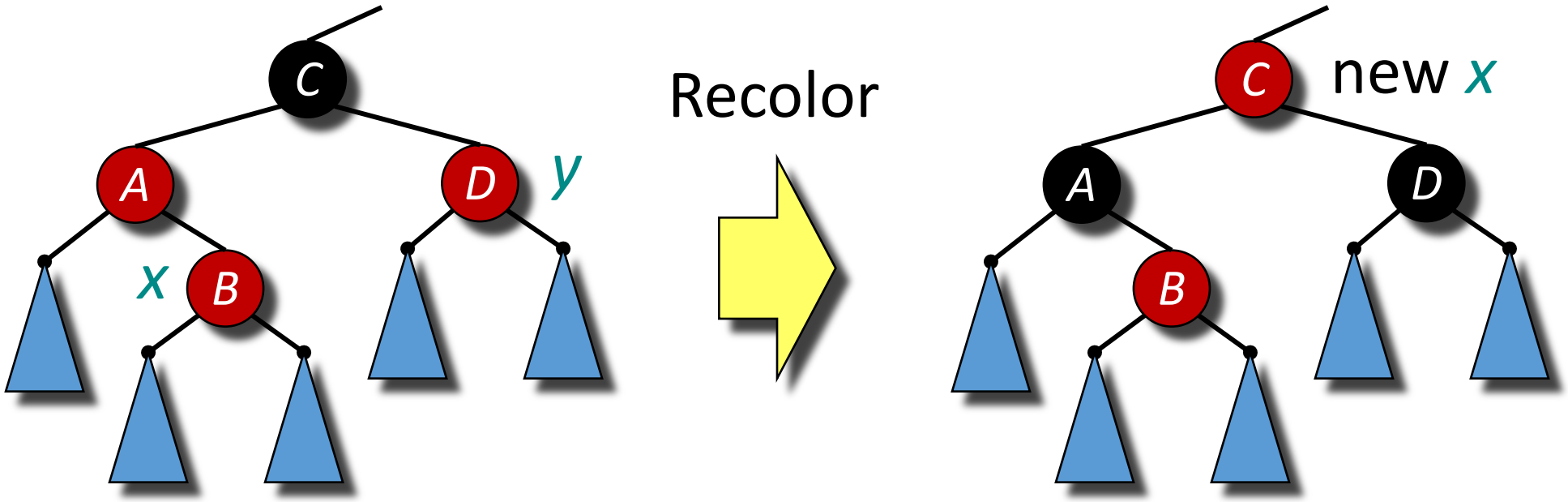
```
RB-INSERT(T, x)
    TREE-INSERT(T, x)
    color[x] ← RED          ▷ only RB property 3 can be violated
    while x ≠ root[T] and color[p[x]] = RED
        do if p[x] = left[p[p[x]]
            then y ← right[p[p[x]]          ▷ y = aunt/uncle of x
                 if color[y] = RED
                  then ⟨Case 1⟩
                  else  if x = right[p[x]]
                          then ⟨Case 2⟩      ▷ Case 2 falls into Case 3
                               ⟨Case 3⟩
            else ⟨"then" clause with "left" and "right" swapped⟩
    color[root[T]] ← BLACK
```

Graphical notation

Let  denote a subtree with a black root.

All  's have the same black-height.

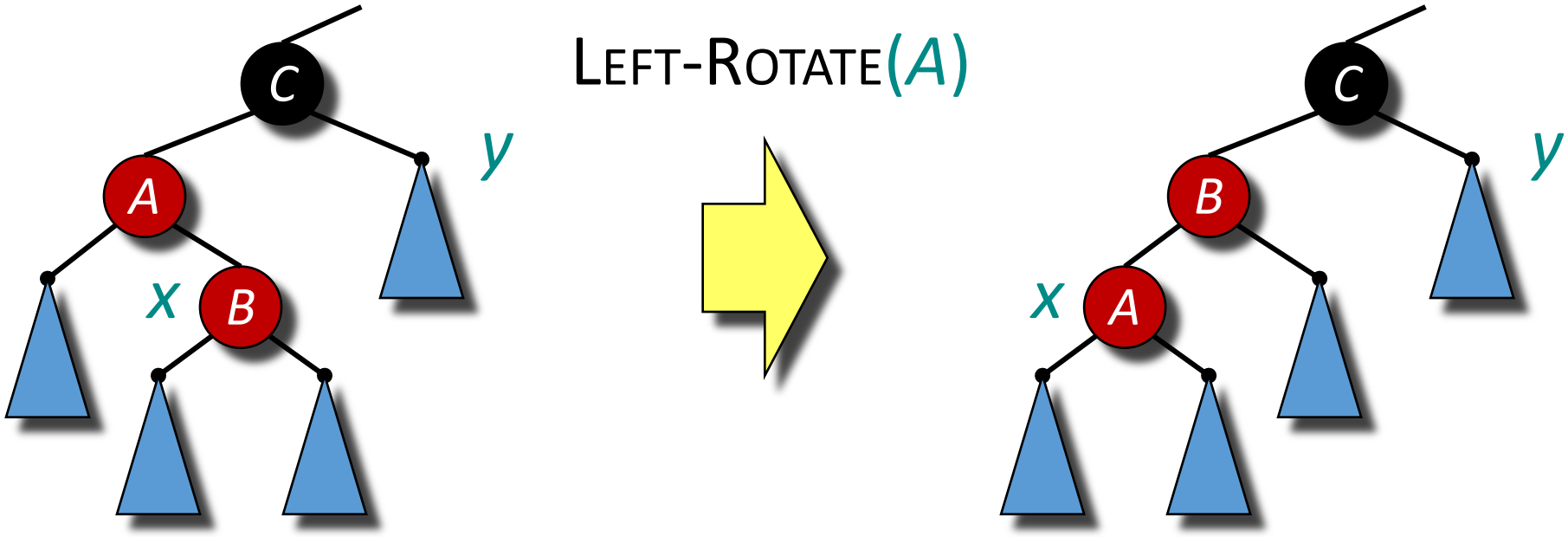
Case 1



(Or, children of *A* are swapped.)

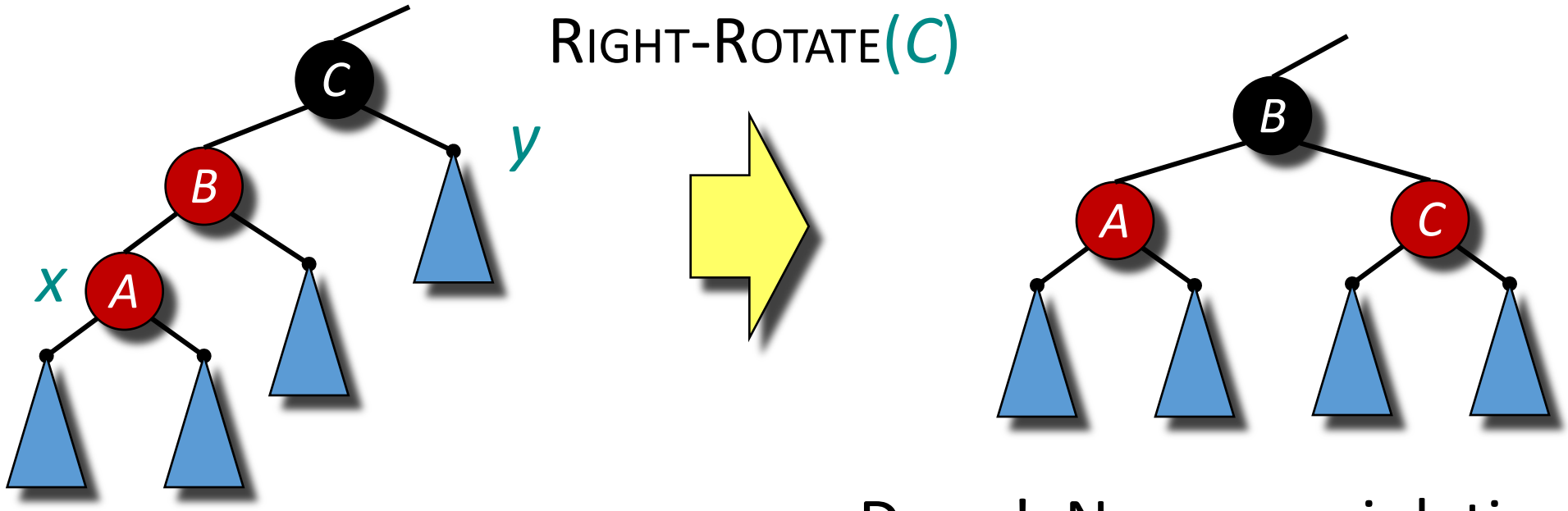
Push *C*'s black onto *A* and *D*, and recurse, since *C*'s parent may be red.

Case 2



Transform to Case 3.

Case 3



Done! No more violations of RB property 3 are possible.

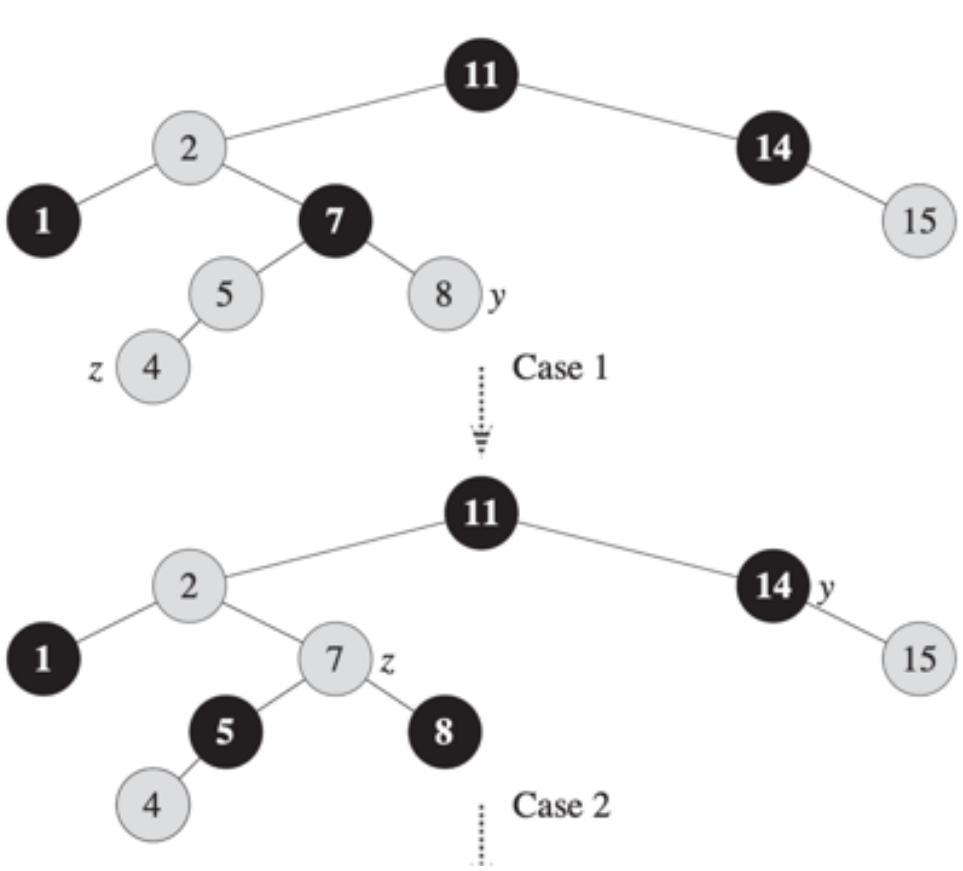
Analysis

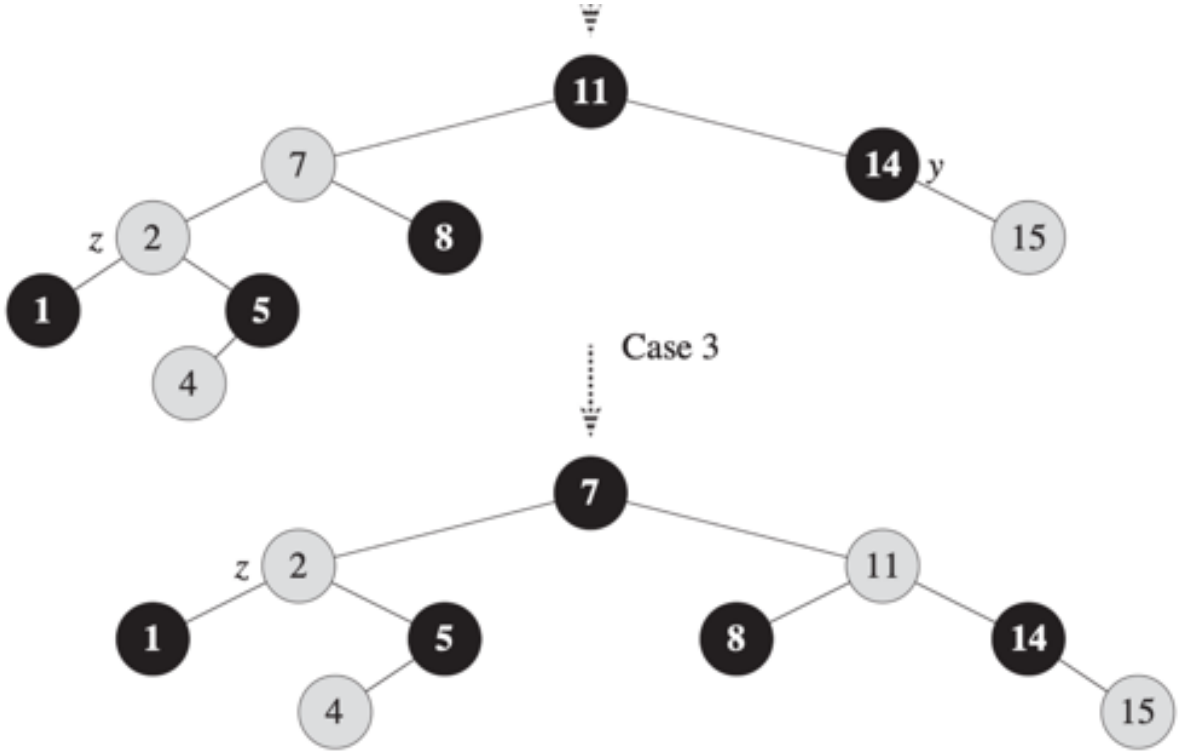
- Go up the tree performing Case 1, which only recolors nodes.
- If Case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

Running time: $O(\lg n)$ with $O(1)$ rotations.

RB-DELETE — same asymptotic running time and number of rotations as RB-INSERT (see textbook).

Insertion Example

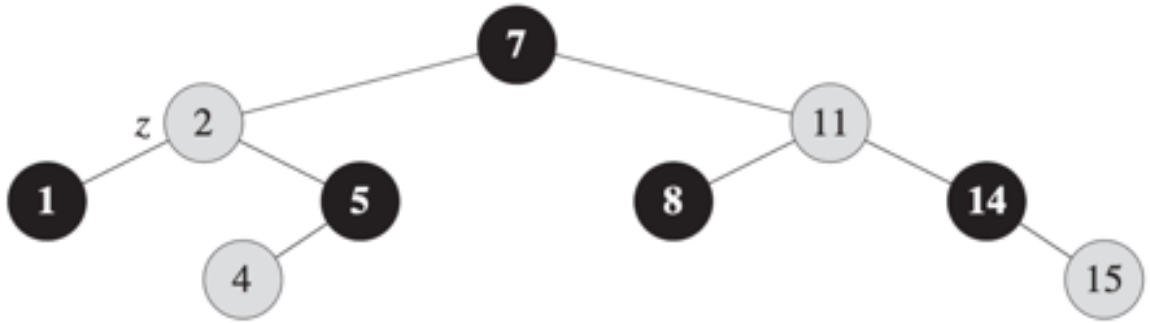




Deletion

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13            x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15            y.right = z.right
16            y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21  if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

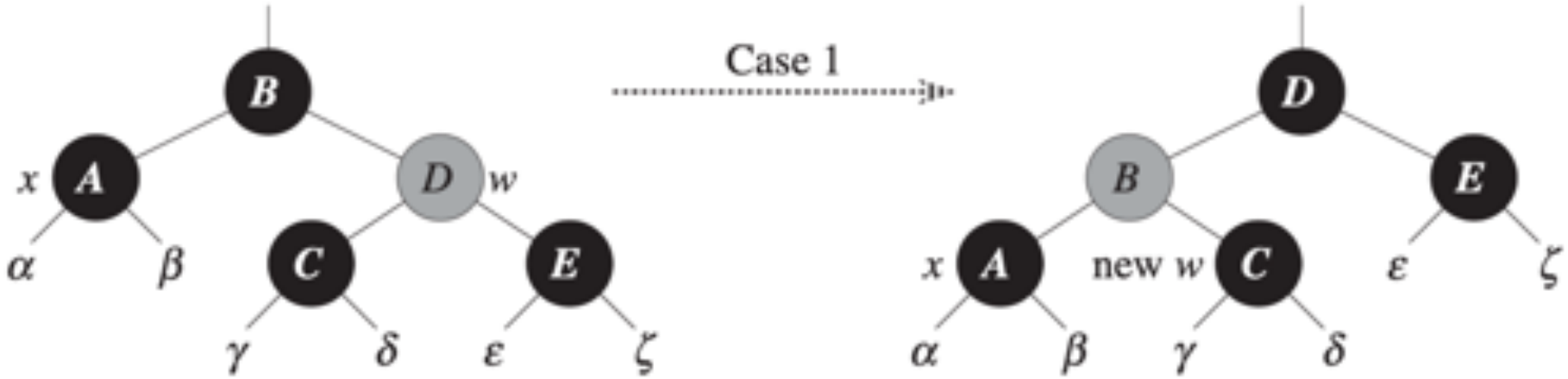
```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2       T.root = v
3  elseif u == u.p.left
4       u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```



```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

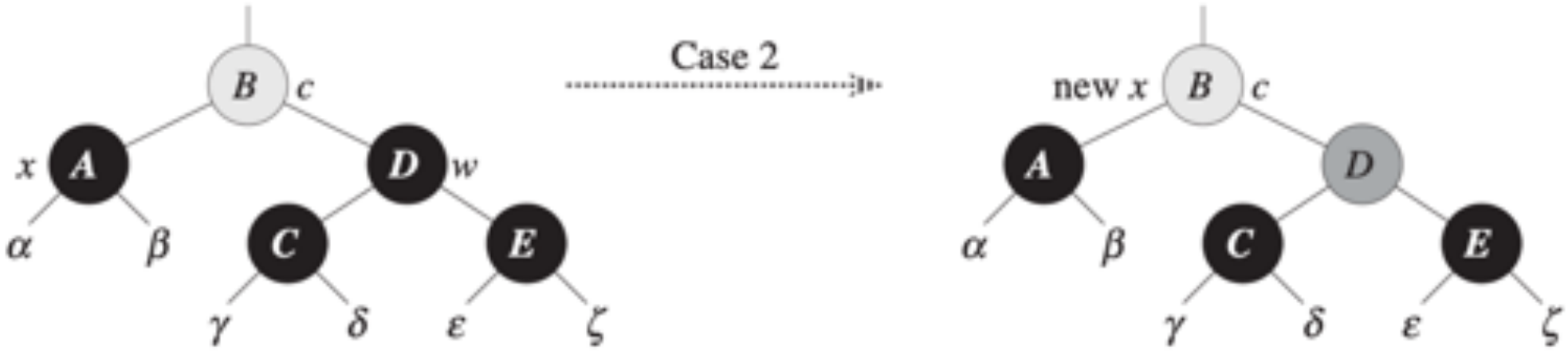
Deletion

- Case 1: x's sibling w is red



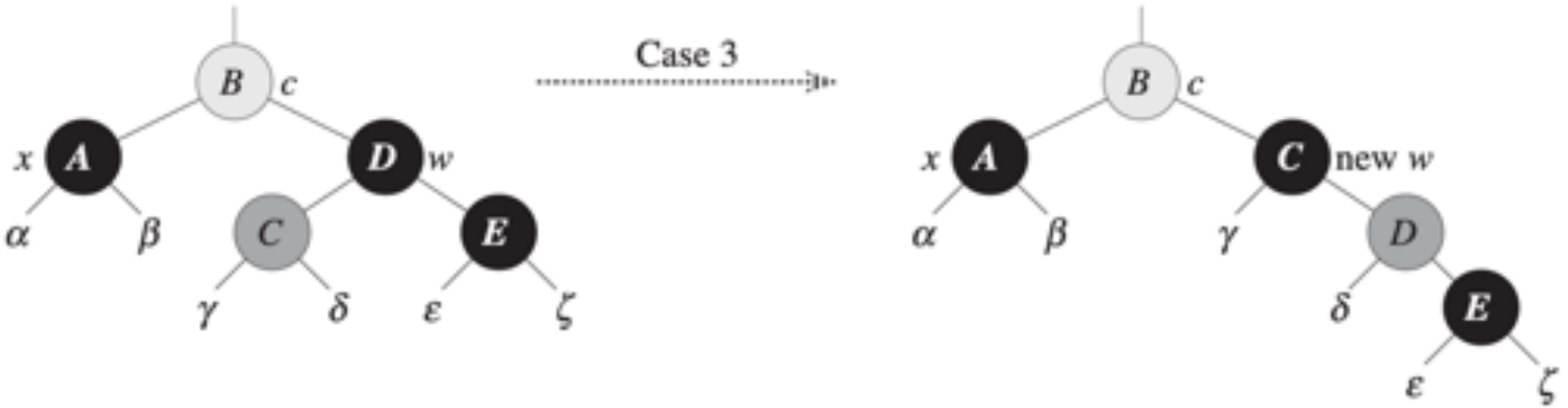
Deletion

- Case 2: x's sibling w is black, and both of w's children are black



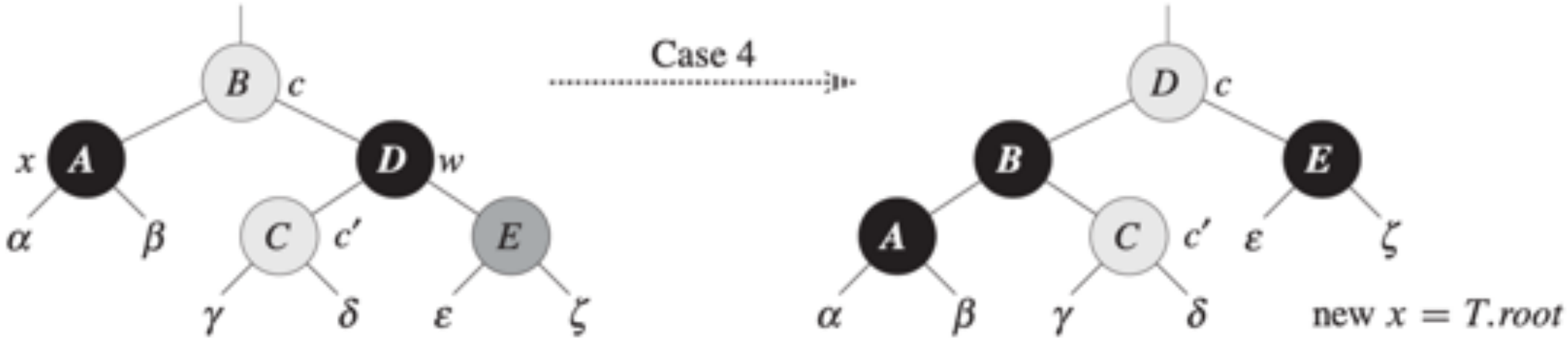
Deletion

- Case 3: x's sibling w is black, w's left child is red, and w's right child is black



Deletion

- Case 4: x's sibling w is black, and w's right child is red



Thank You