
# MULTI-WAY SEARCH TREES

# MULTI-WAY SEARCH TREE

There are data structures that use a multi-way search tree where the following holds:

- Each internal node has **at least 2 children** (a node with $d$ children is called a $d$-node)
- A $d$-node has $d-1$ entries, **sorted** by their keys, i.e.:

$$k_1 \leq \ldots \leq k_{d-1}$$

- Let us define $k_0 = -\infty$ and $k_d = \infty$.
- Then, for every entry $(k, x)$ in the subtree rooted at the $i^{th}$ child, we have:

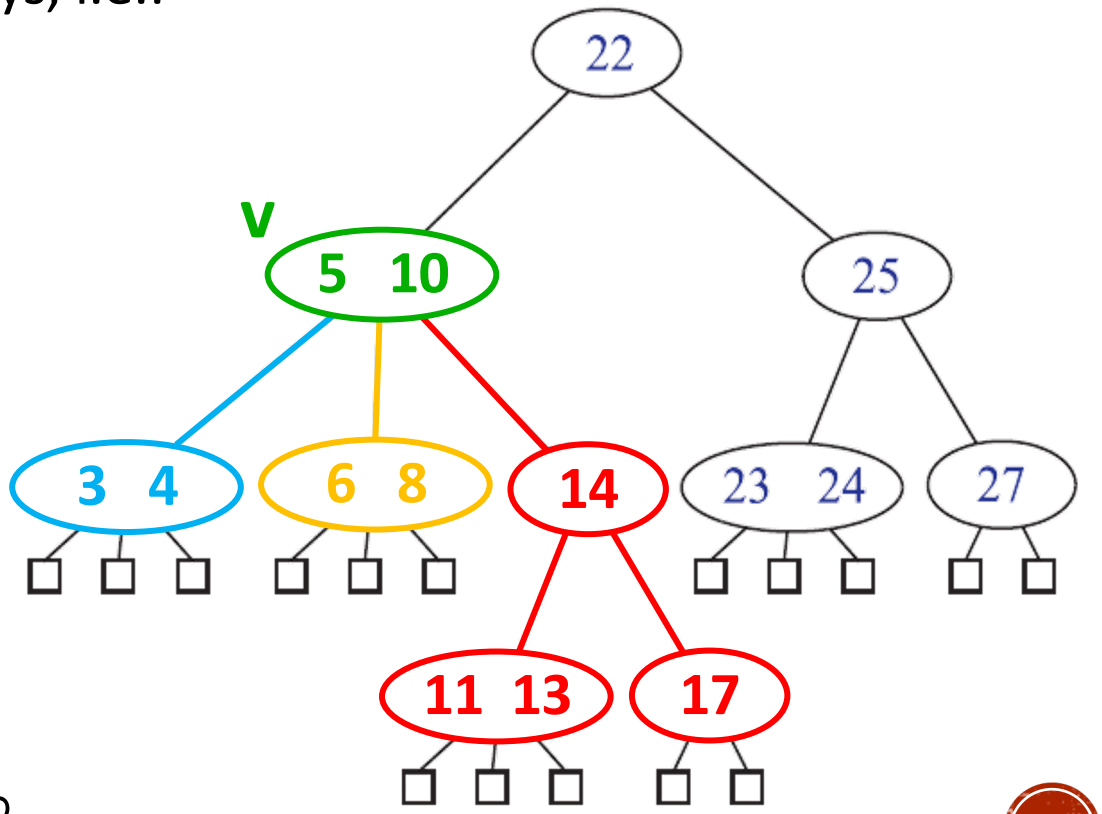
$$k_{i-1} \leq k \leq k_i$$

Example: for node **v**:

$$-\infty \leq (3 \text{ and } 4) \leq 5$$

$$5 \leq (6 \text{ and } 8) \leq 10$$

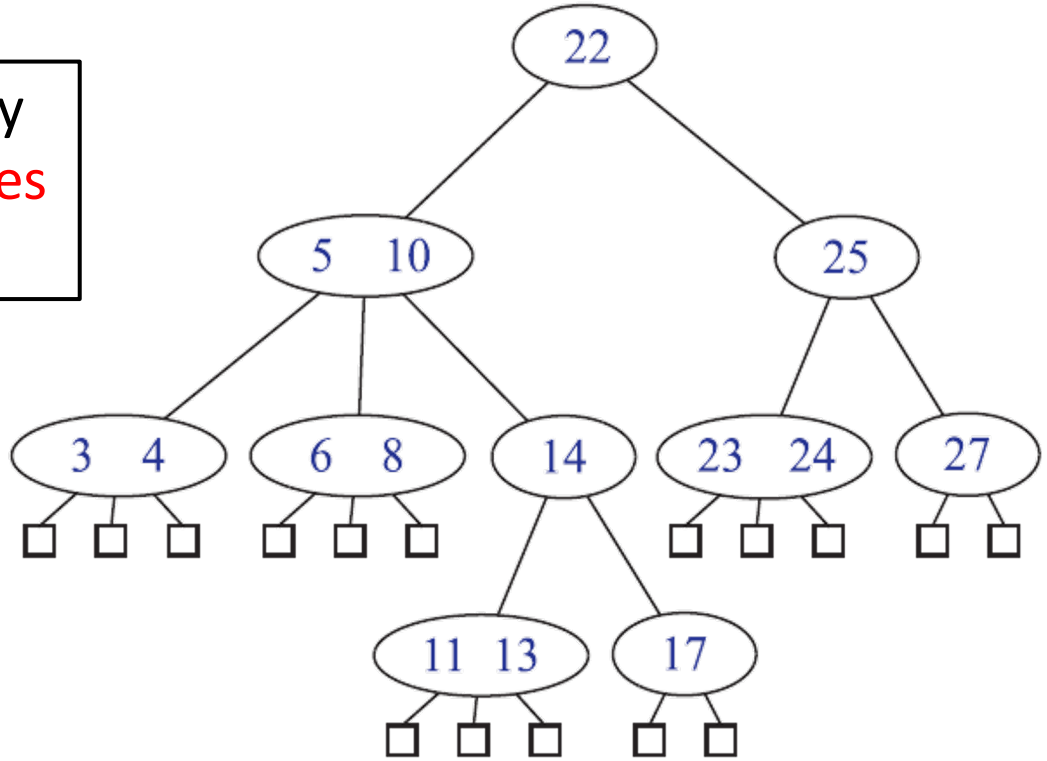
$$10 \leq (11, 13, 14, \text{and } 17) \leq \infty$$



# MULTI-WAY SEARCH TREE

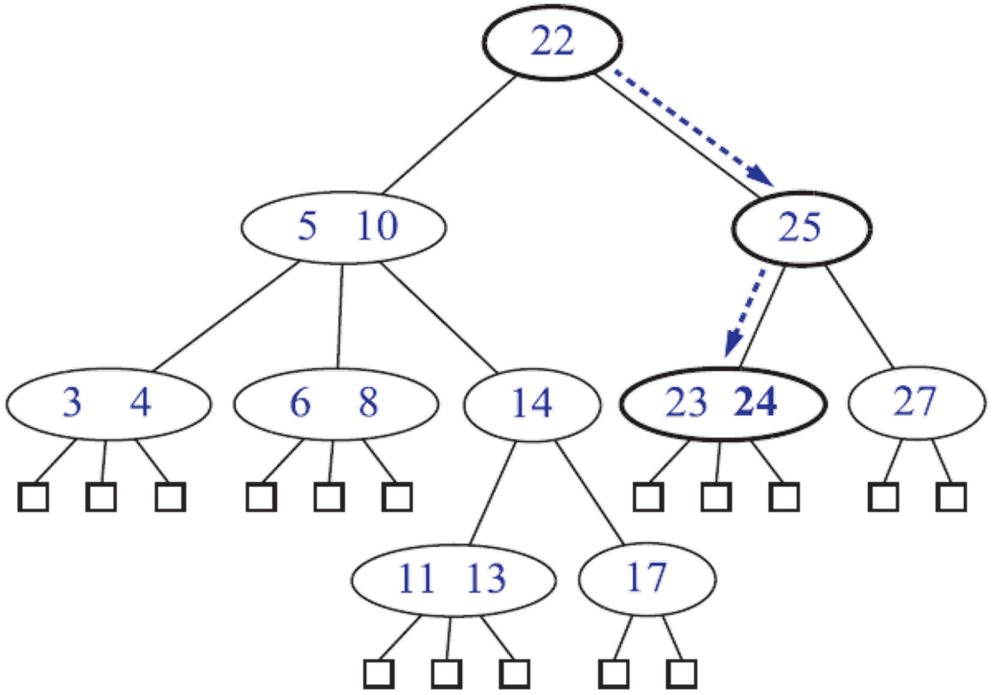
- Notice how, in a multi-way search tree:
  - **External nodes** are "*place holders*" that do not contain any entries
  - The node that has $j$ external children has $j - 1$ entries

Proposition: A multi-way search tree with n entries has n+1 external nodes

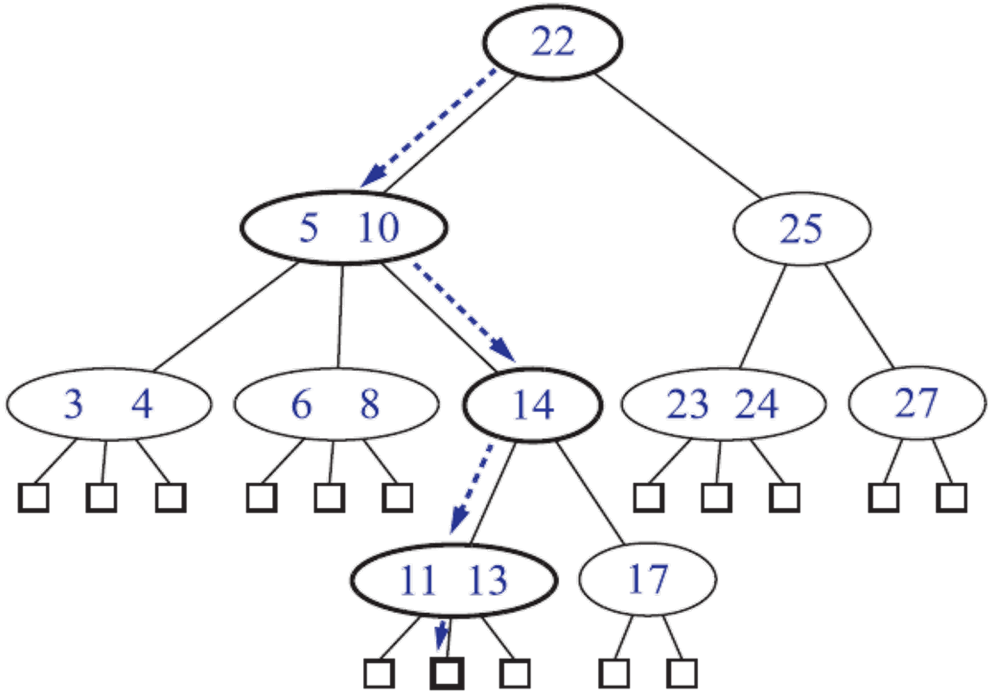


# MULTI-WAY SEARCH TREE

- A binary search tree is actually a **special case** of a multi-way search tree!
- In both types of trees, search is done in exactly the same way!




How we searched for, and found, an entry with key = **24**




How we searched for an entry with key = **12**, and determined that it doesn't exist

# IMPLEMENTING EACH NODE

- In any given node, entries are stored in a map (remember: a map is any data structure that stores **key-value pairs**, called entries)

For node $v$, **suppose the entries were as follows**

$(\mathbf{5}, x_1)$

$(\mathbf{10}, x_2)$

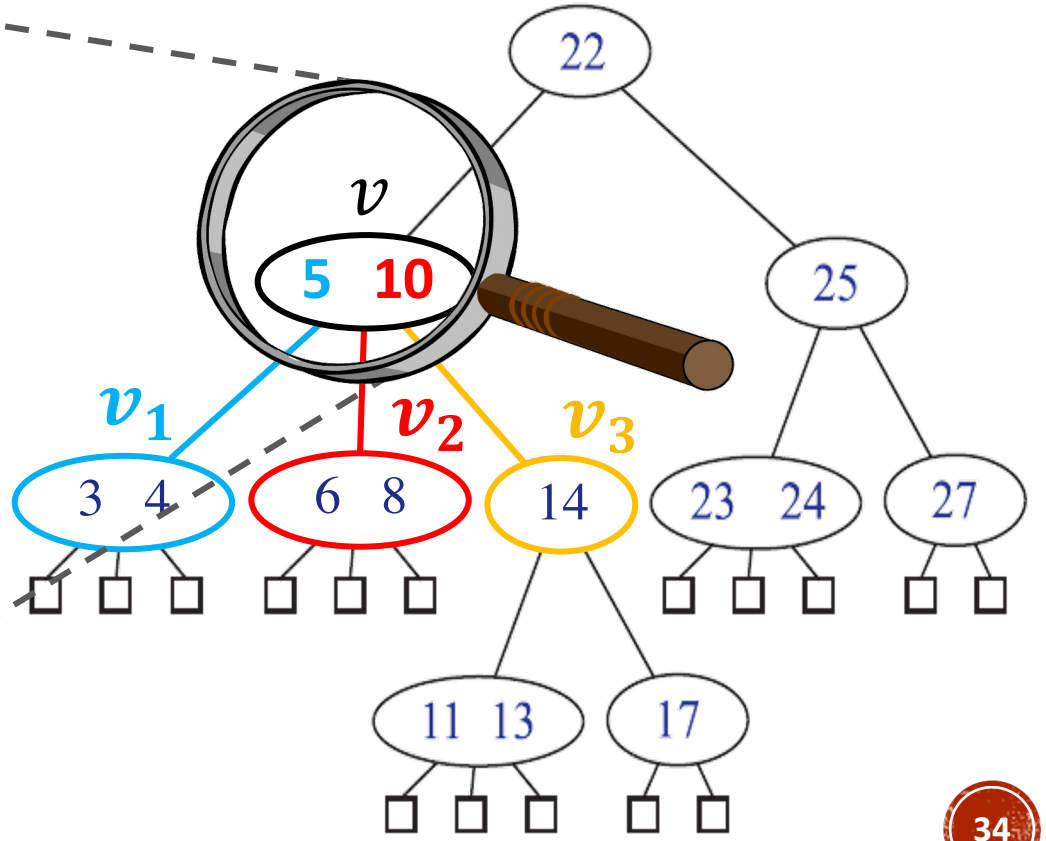
*(recall that the illustration only shows the keys, and omits the values, i.e., omits $x_i$)*

Then the **map** $M(v)$ will have the entries:

$(\mathbf{5}, (x_1, \boldsymbol{v_1}))$

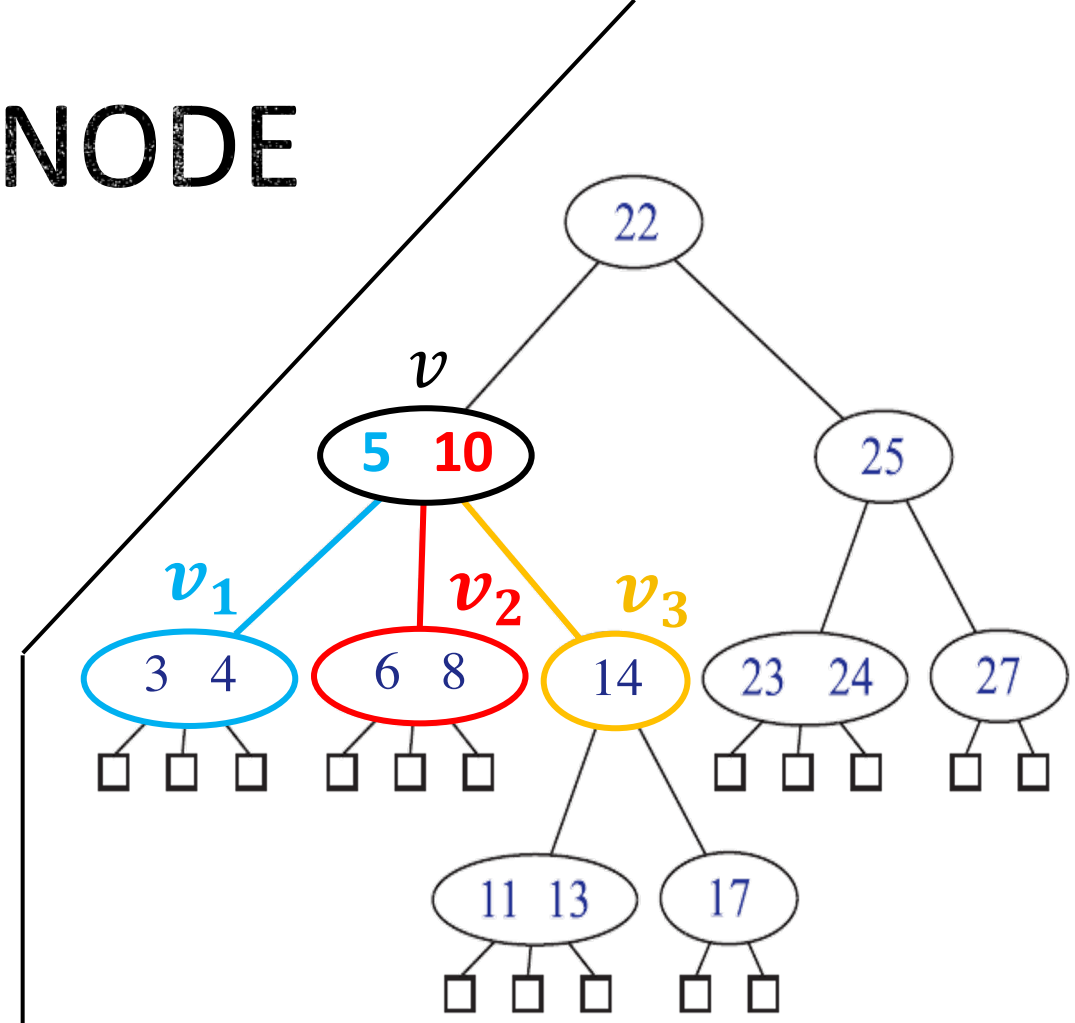
$(\mathbf{10}, (x_2, \boldsymbol{v_2}))$

$(+\infty, (\emptyset, \boldsymbol{v_3}))$



# IMPLEMENTING EACH NODE

- Suppose we are **searching the tree** for an entry with key = $k$.
- Then, if we reach node $v$, we **search the map** $M(v)$ to find the entry $(k_i, (x_i, v_i))$ with the smallest key that is $k_i \geq k$. Now:
  - If $k_i > k$, we continue by the search, but this time by **searching the map** $M(v_i)$
  - If $k_i = k$ the search **terminates** successfully
- If $v$ is a d-node then searching it takes O(log d) if $M(v)$ is implemented as an ordered search table




The **map** $M(v)$ has the entries:

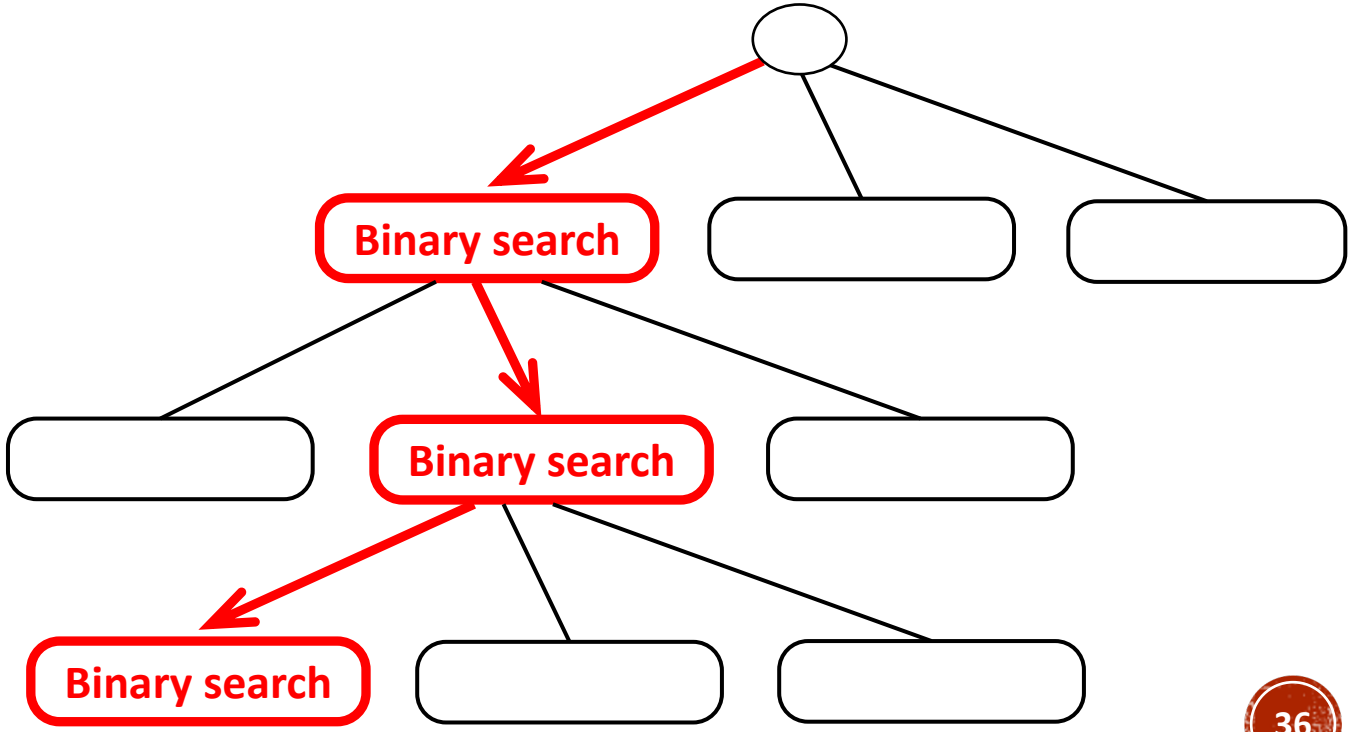
$(5, (x_1, v_1))$

$(10, (x_2, v_2))$

$(+\infty, (\emptyset, v_3))$

# SEARCH COMPLEXITY

- To summarize, we said that searching the map of a d-node takes O(log d) time, and we may have to proceed to searching one of its children, etc.
- Thus, **searching the entire tree** takes $O(h \log d_{max})$ where:
  - $d_{max}$ is the max number of children of a node in the tree
  - $h$ is the height of the tree
- Thus, to search a multi-way tree efficiently, we need to keep the height as small as possible, i.e., a logarithmic function of $n$.
- A search tree with logarithmic height such as this is called a **balanced search tree**.



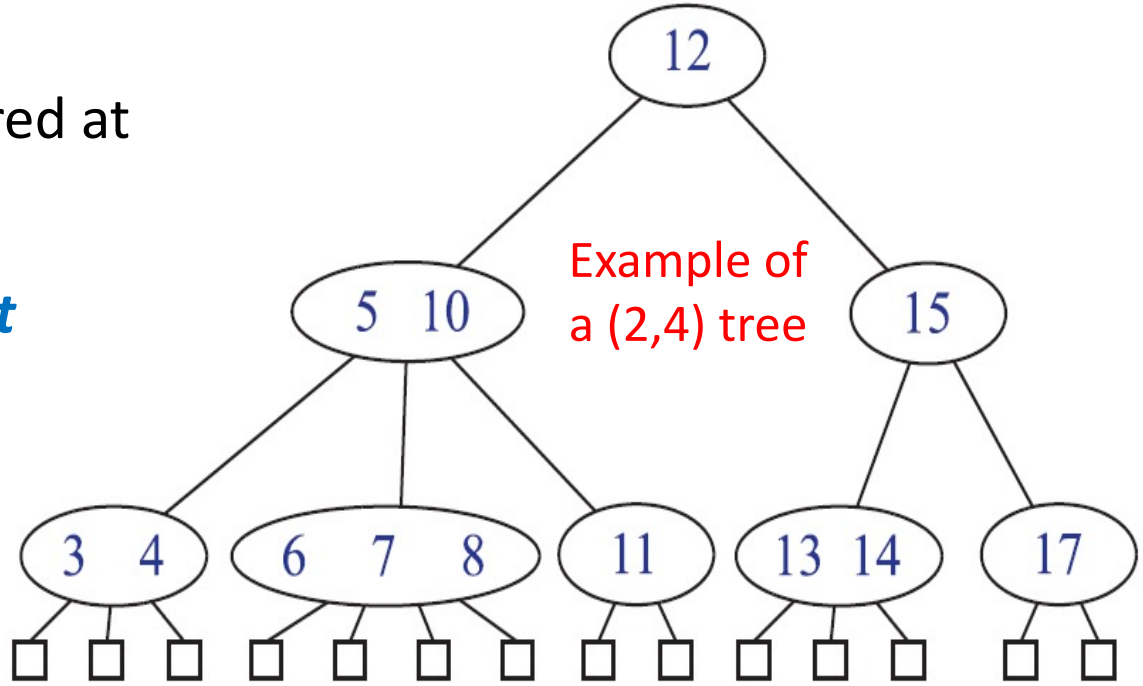
# (2,4) TREES

# (2,4) TREES

- A "(2,4) tree" is a multi-way search tree that satisfies the following properties:
  - **Depth property:** All external nodes have the same depth
  - **Size property:** Every internal node has at most 4 children
- It is called a "(2,4) tree" because each internal node has:
  - At least **2** children by the definition of multi-way search trees
  - At most **4** children by the **size property**
- Note that the **size property** implies that each internal node has at most 3 entries

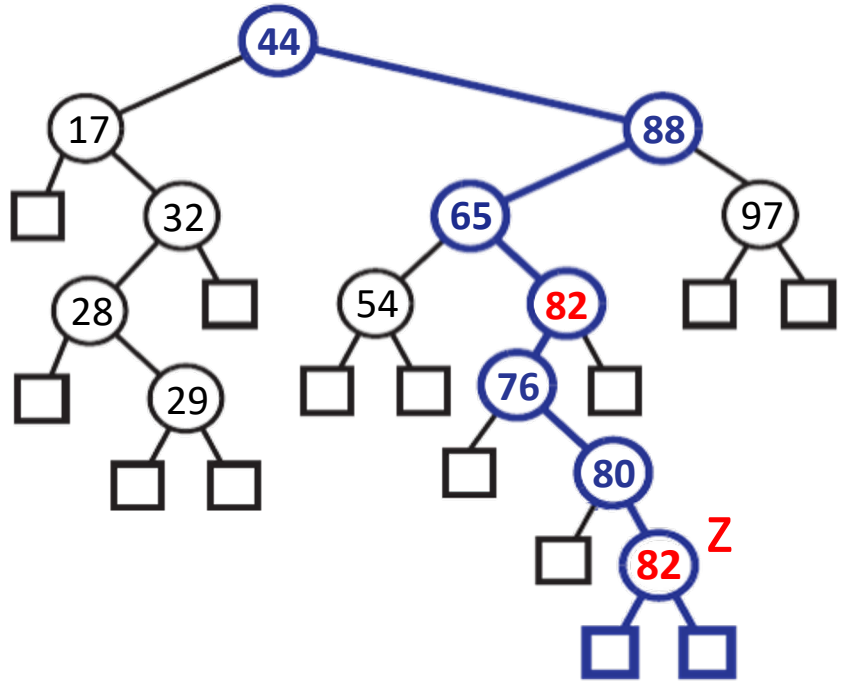
# (2,4) TREES

- A "(2,4) tree" is a multi-way search tree that satisfies the following properties:
  - **Depth property:** All external nodes have the same depth
  - **Size property:** Every internal node has at most 4 children
- Note that:
  - The **size property** keeps the ***map*** stored at each node small
  - The **depth property** makes the ***height*** of the tree logarithmic in n



# BINARY SEARCH TREE – INSERTION

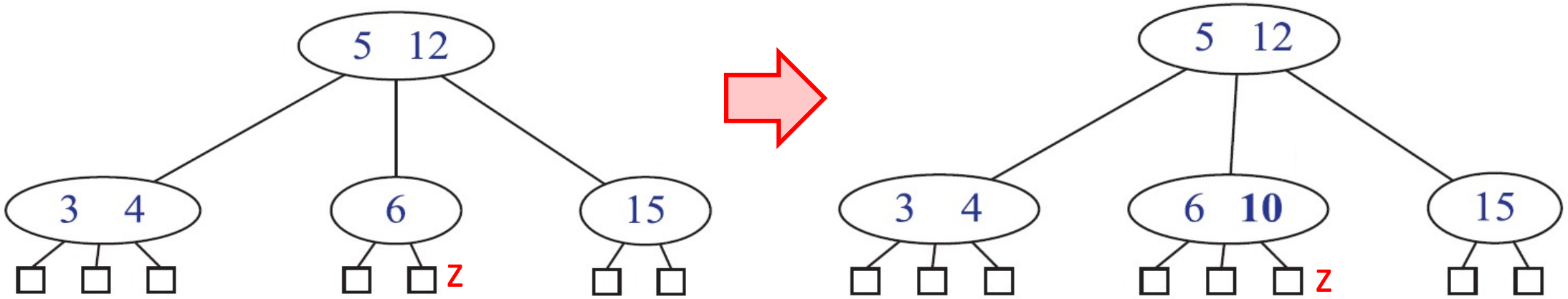
- Remember: In a binary search tree, to insert an entry (k,v), we first search for an entry whose key = k. This search returns either:
  - an **internal** node whose key = k, or
  - an **external** node where we can insert




- If we find an **internal** node, then the tree already contains an entry whose key = k, in which case we search further down the tree, until we reach an external node!
- Thus, insertion always happens at an **external** node!
- **The same idea applies in (2,4) trees**, i.e., to insert an entry (k,v)
  - We repeatedly search for an entry whose key = k, until we reach an external node
  - Then we insert (k,v) at that external node, denoted by z

# (2,4) TREES – INSERTION

- Here is an example of how to insert **10**. We **searched** for **10**, and our search did not find it, but **found an external node** z where we can insert **10**.
- We insert the entry in the **parent** of z, and add an external node to the left of z




- The only problem is if the parent of z already had 4 children, which is **the maximum number of permitted children** in a (2,4) tree!
- In this case, after the insertion, the parent of z will have **5** children, which we call an "**overflow**". To fix this, we need to perform a "**split**" operation.

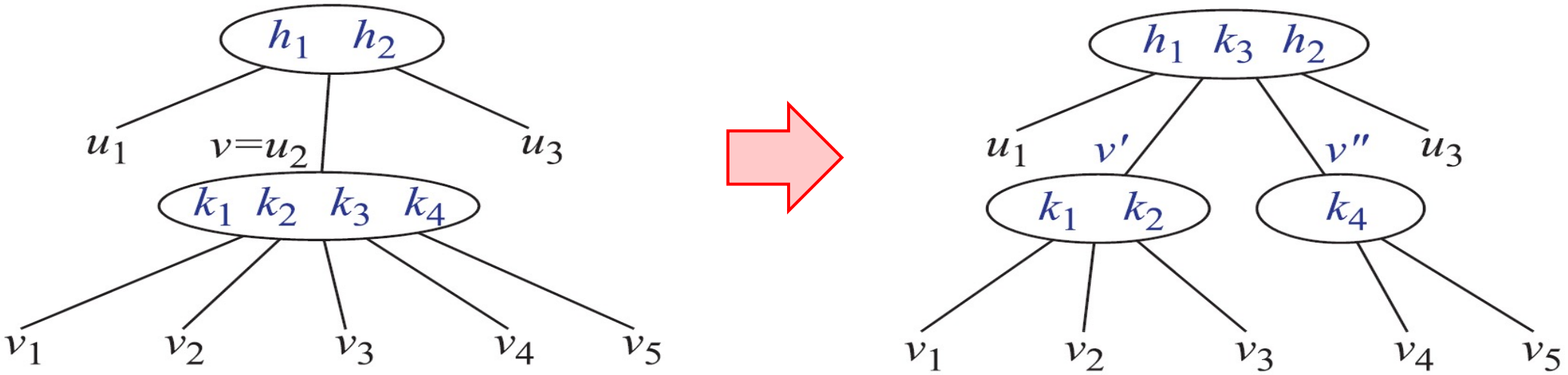
# (2,4) TREES – SPLIT TO FIX AN OVERFLOW

Here is an example where we insert an entry in node v, leaving it with four entries $(k_1, k_2, k_3, k_4)$ and **5 children**, resulting in an **overflow**. Here is how we **split** v:

- v is split into two nodes, $v'$ and $v''$
- $k_1, k_2$ go to $v'$
- $k_3$ goes to the **parent** of $v$
- $k_4$ goes to $v''$

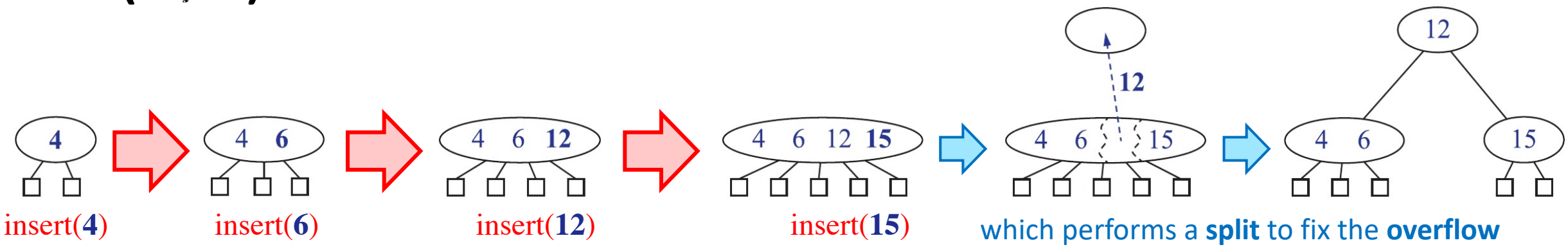
As for the **children**:

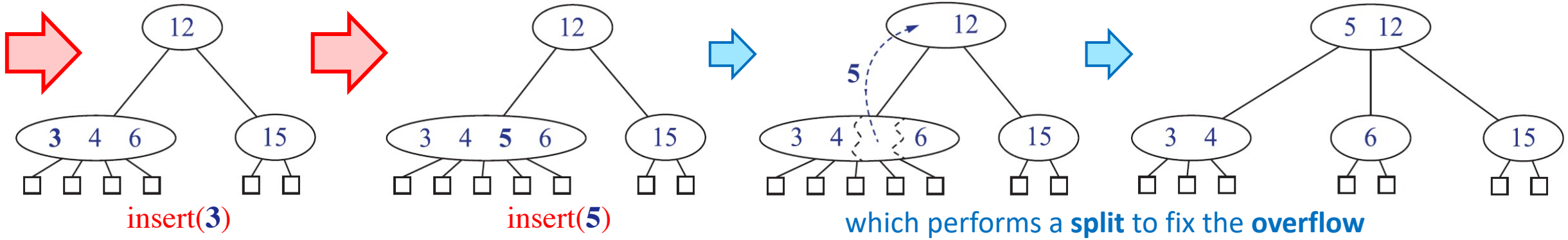
- $v_1, v_2, v_3$ go with $v'$
- $v_4, v_5$ go with $v''$

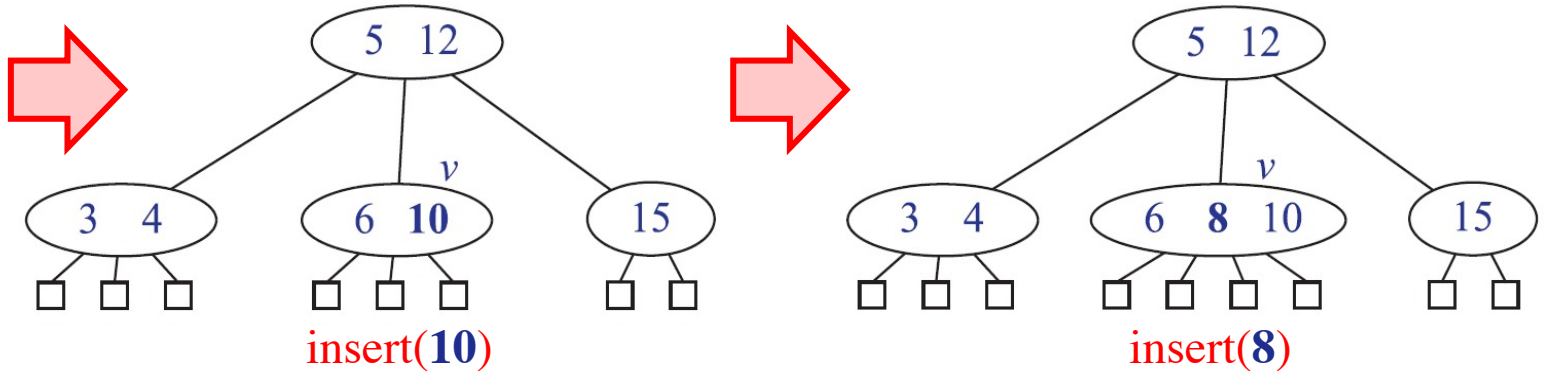



Note: when $k_3$ goes to the **parent** of $v$, if $v$ didn't have a parent, we create one for it!

# (2,4) TREES – INSERTION EXAMPLE



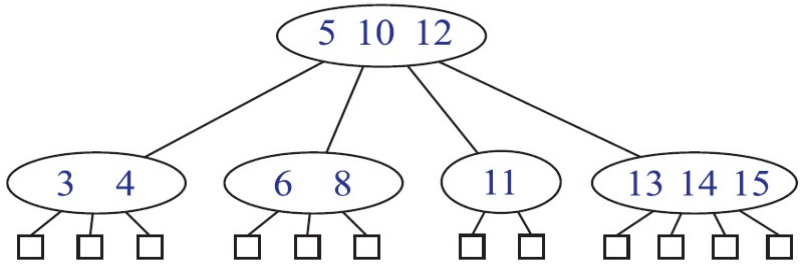



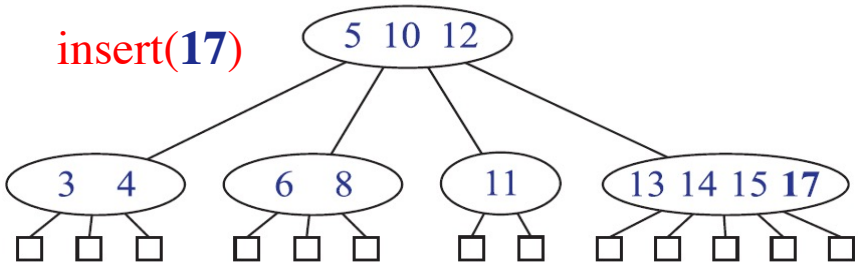




and so on…

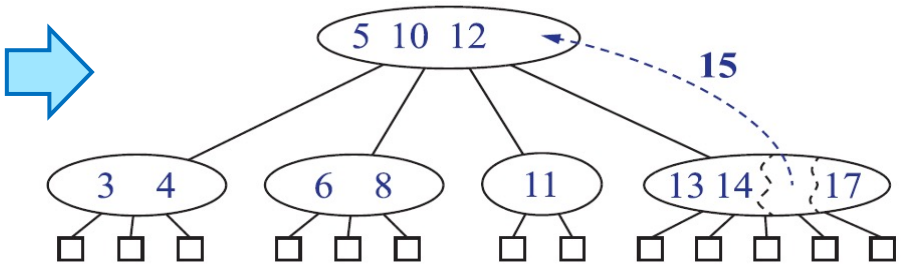
Fixing an overflow may result in a new overflow further up the tree!

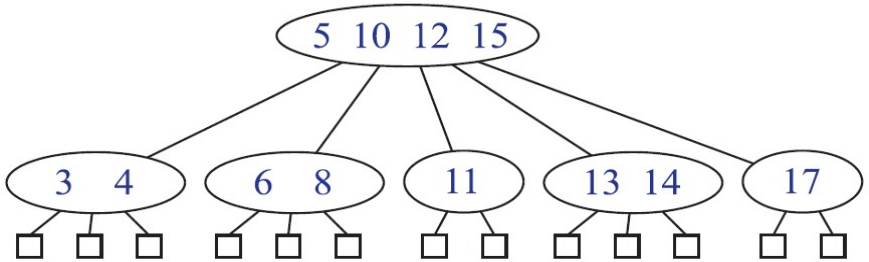
# (2,4) TREES – INSERTION EXAMPLE



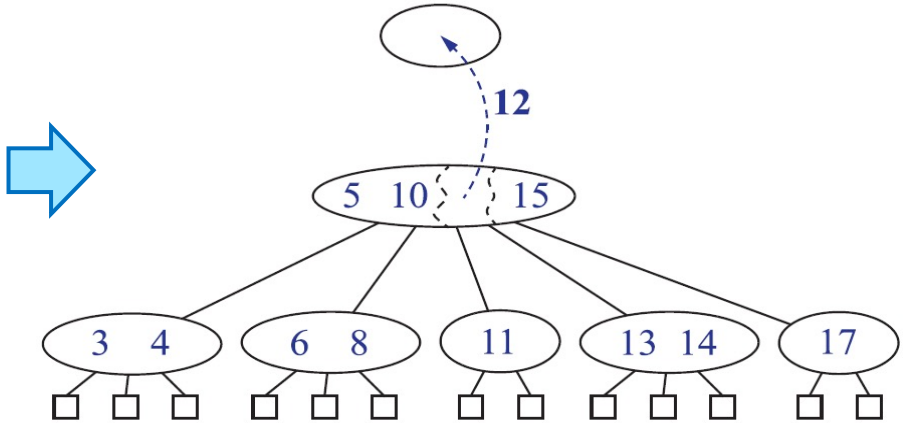



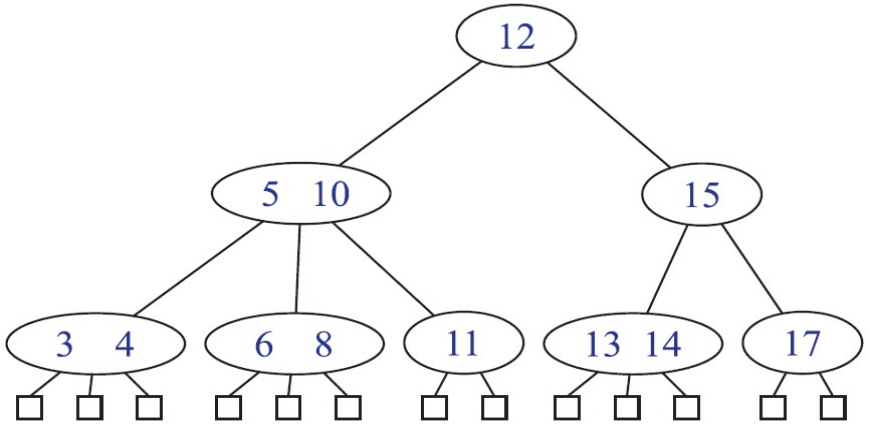

this results in an **overflow** that needs to be fixed by a **split**!




but this results in **another overflow**, which needs to be fixed by yet **another split**!



# (2,4) TREES – INSERTION COMPLEXITY

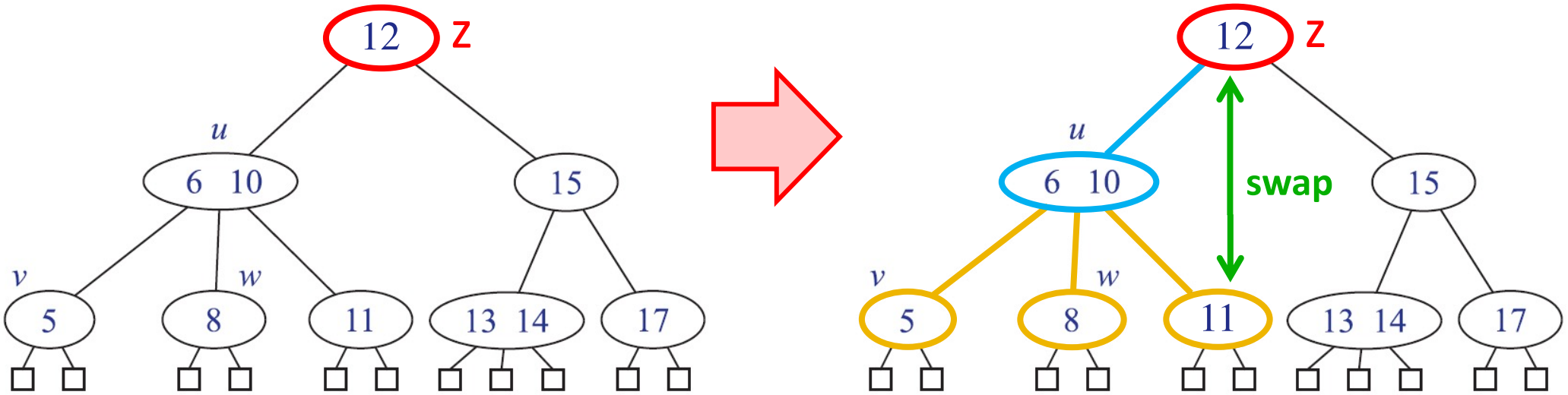
Let's analyze the complexity of **inserting** an entry (k,v):

- First, we repeatedly **search** for an entry whose key = k, until we reach an external node where it would be suitable to insert (k,v)
  - This takes O(log n) time, since the **height** of a (2,4) tree is O(log n)
- Then, we may have to repeatedly perform a **split** from that node all the way up to the root
  - We may perform O(log n) splits, since the **height** of the tree is O(log n)
  - Each split requires a constant no. of operations, i.e., it takes O(1) time

Thus, the **total time** to perform an insertion in a (2,4) tree is O(log n)

# (2,4) TREES – REMOVAL

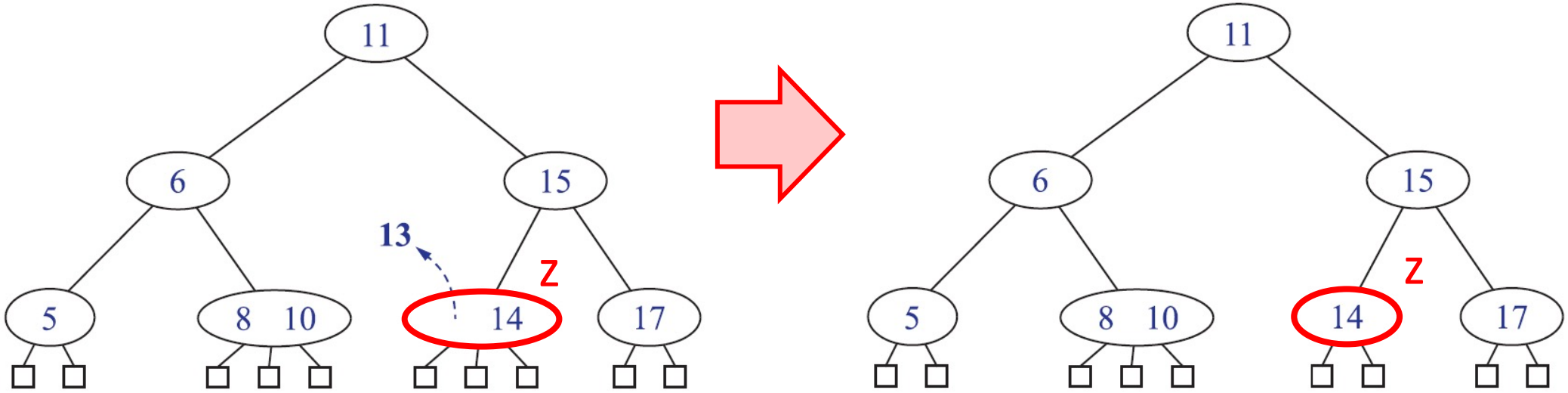
- We will only remove entries from a node whose children are **external nodes**!
- What if we want to remove the $i^{th}$ entry from a node z that has only **internal children**?




- We find the last entry of the **right-most internal node in the subtree** rooted at the $i^{th}$ child of z, and we swap it with $i^{th}$ entry of z
- This way, the entry we want to remove is in a **node whose children are external**!

# (2,4) TREES – REMOVAL

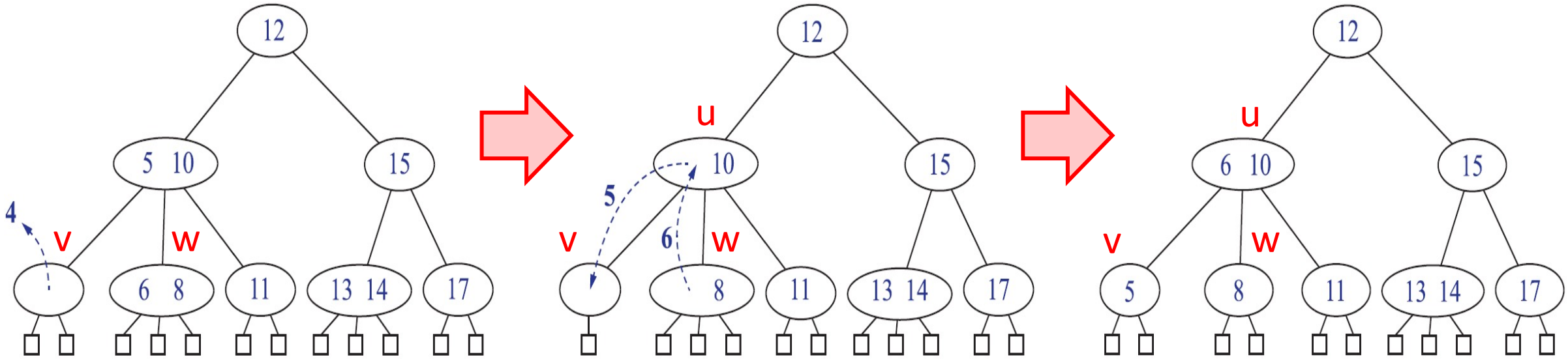
- So far, we ensured that the entry we want to remove is always going to be a node z whose children are external nodes




- Now, when removing an entry from z, we also remove the $i^{th}$ external child of z
- The only problem is if z had only **1 entry** (i.e., **2 children**). Then, after removing that entry, z will be left with no entries and **1 child**, which is not allowed in (2,4) trees! This situation is called an "**underflow**"

# (2,4) TREES – HOW TO FIX AN UNDERFLOW

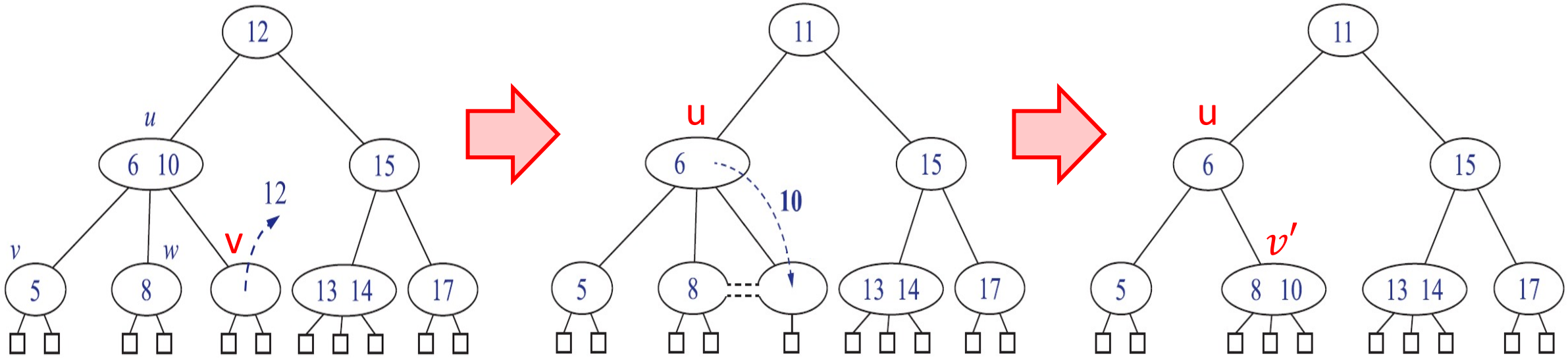
- If v has an **immediate sibling**, w, that has 3 or 4 external nodes, the sibling can afford to lose one of its external nodes (and entries) to help v out. In this case:
  - w gives one of its entries to the parent, u
  - the parent, u, gives one of its entries to v




- This is called a **transfer** operation

# (2,4) TREES – HOW TO FIX AN UNDERFLOW

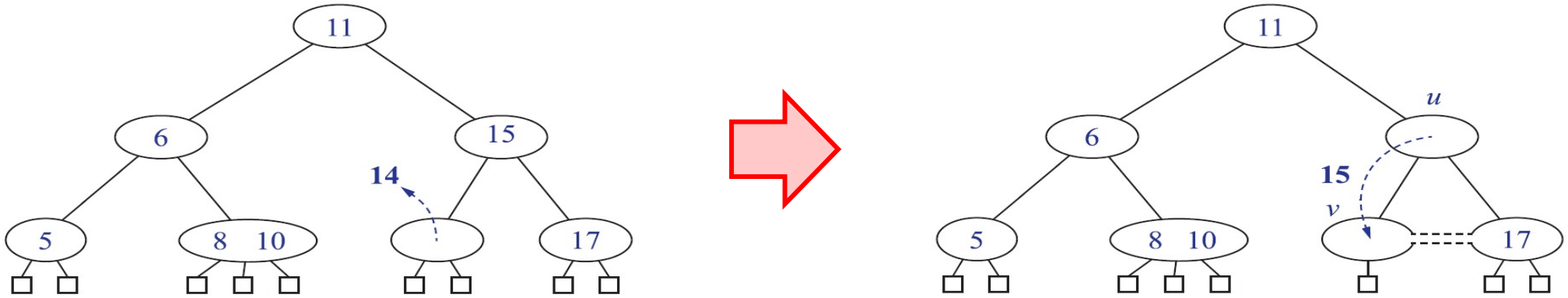
- But what if v **does not have** an immediate sibling, w, with 3 or 4 external nodes?
  - $v$ is merged with its sibling, creating a new node $v'$
  - $v'$ takes an entry from its parent, u



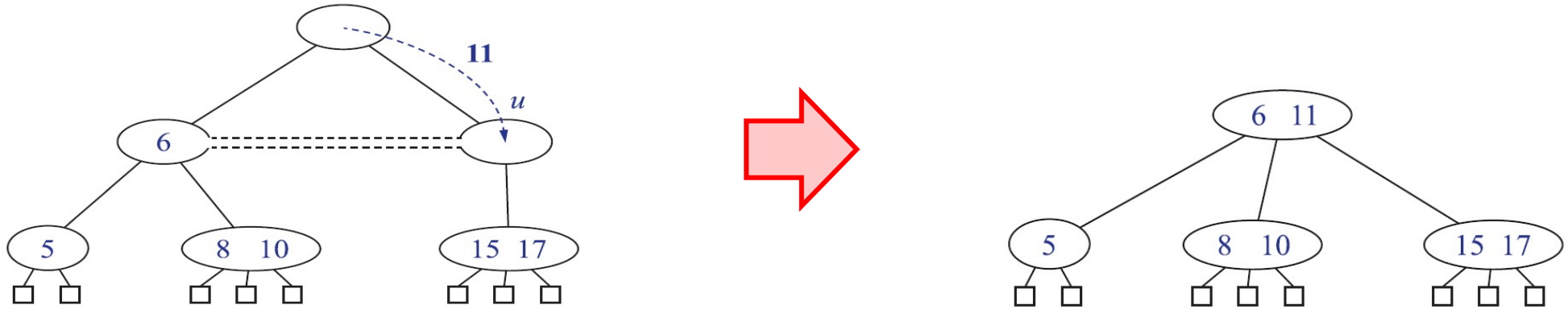

- This is called a **fusion** operation

# (2,4) TREES – HOW TO FIX AN UNDERFLOW

- But what if, after $v'$ takes an entry from its parent u in a **fusion operation,** u had no entries left, thereby causing an **underflow**?




- Just repeat the process for the parent with either a **transfer** or **fusion** operation
  *(in the special case where u is the **root**, just delete u after the fusion!)*



# (2,4) TREES – REMOVAL COMPLEXITY

Let us analyze the complexity of **removing** an entry whose key = k

- First, we **search** for an entry with key = k, and if its children weren't external nodes, we identify the external node that should be swapped with it
  - This takes O(log n) time, since the **height** of a (2,4) tree is O(log n)
- After that, we may have to repeatedly perform a **fusion** operation from that node all the way up to the root (possibly followed by a single **transfer** operation)
  - We may perform O(log n) such operations, since the tree **height** is O(log n)
  - Each such operation requires a constant No. of steps, i.e., takes O(1) time

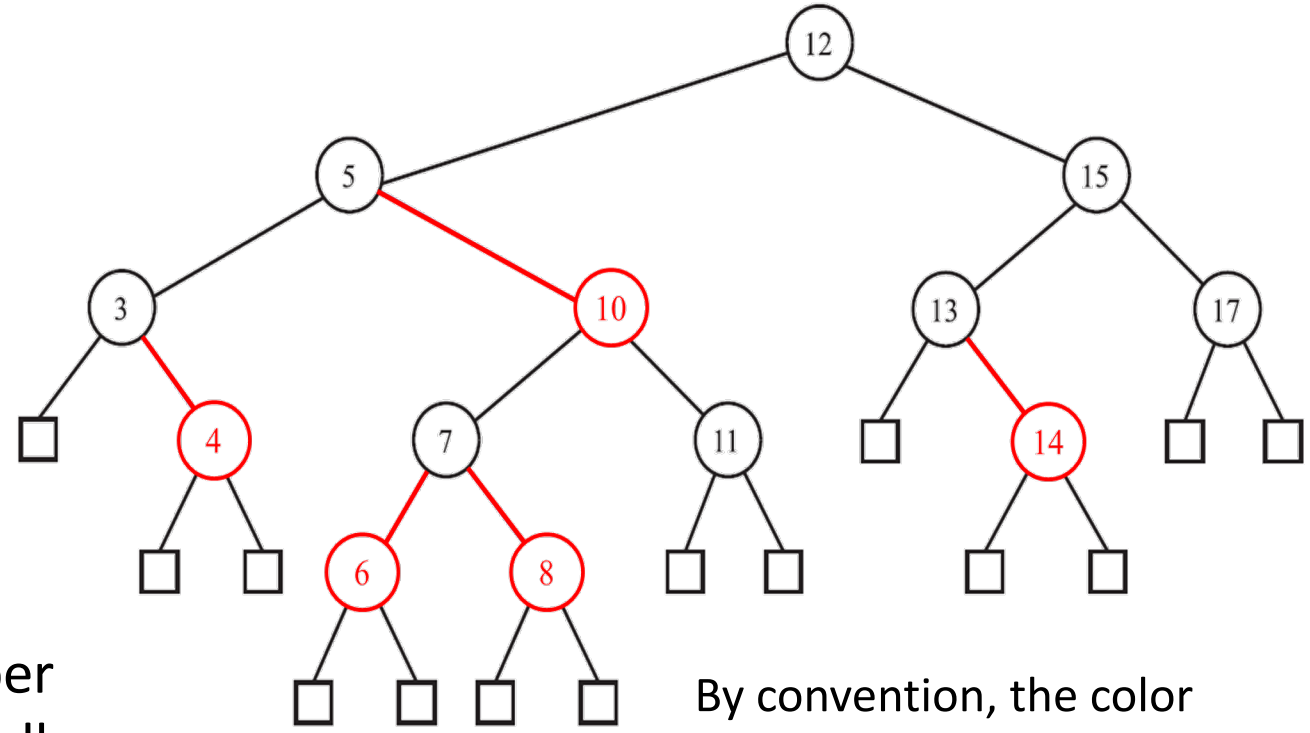
Thus, the **total time** to perform a removal in a (2,4) tree is O(log n)


# RED-BLACK TREES

# RED-BLACK TREES

A Red-**Black** tree is a **BINARY** search tree with nodes colored Red or **Black** such that:

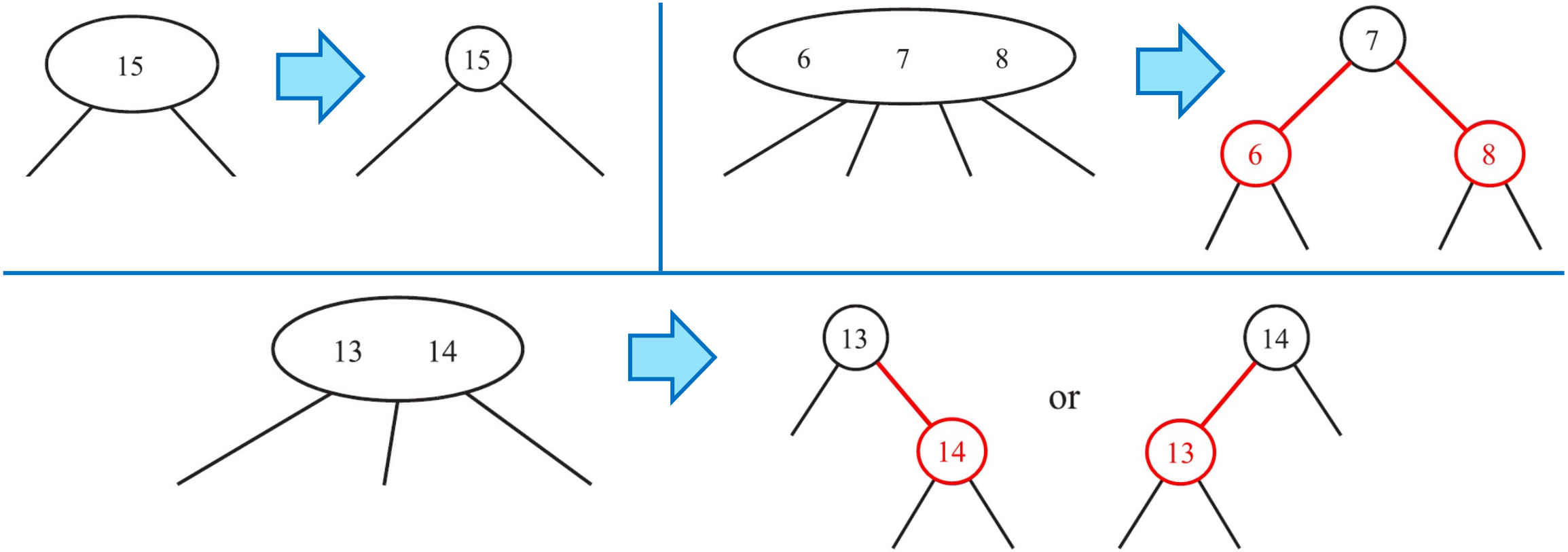
- **Root** property:
  The root is **black**
- **External** property:
  External nodes are **black**
- **Internal** property:
  Children of a red node are **black**
- **Depth** property:
  All external nodes have the same **black-depth**, defined as the number of **black** ancestors minus one (recall that a node is an ancestor of itself!)




By convention, the color of an edge = color of the node underneath it!

# RED-BLACK VS. (2,4) TREES

- You can transform a (2,4) tree into a Red-**Black** tree, and vice versa as follows:

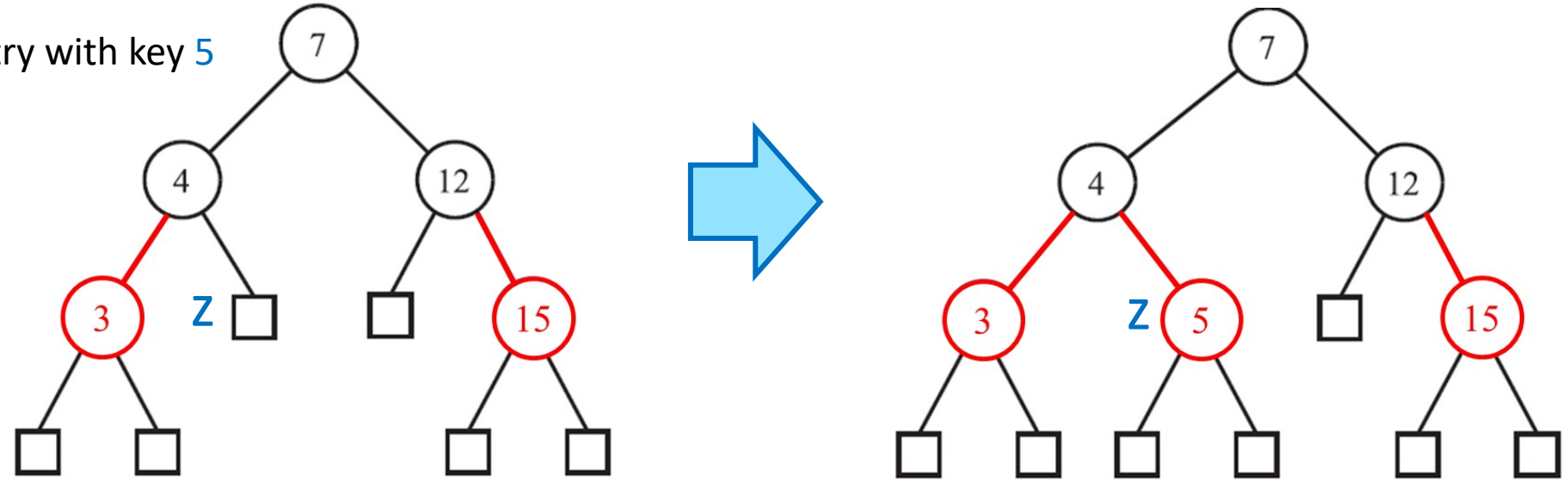



- Also, just like (2,4) trees, the height of a Red-**Black** tree with n entries is O(log n)

# RED-BLACK – INSERTION

- As with any search tree, to insert an entry (k,x) in a Red-**Black** tree, we repeatedly search for an entry whose key = k, until we reach an external node z

E.g. Insert an entry with key 5

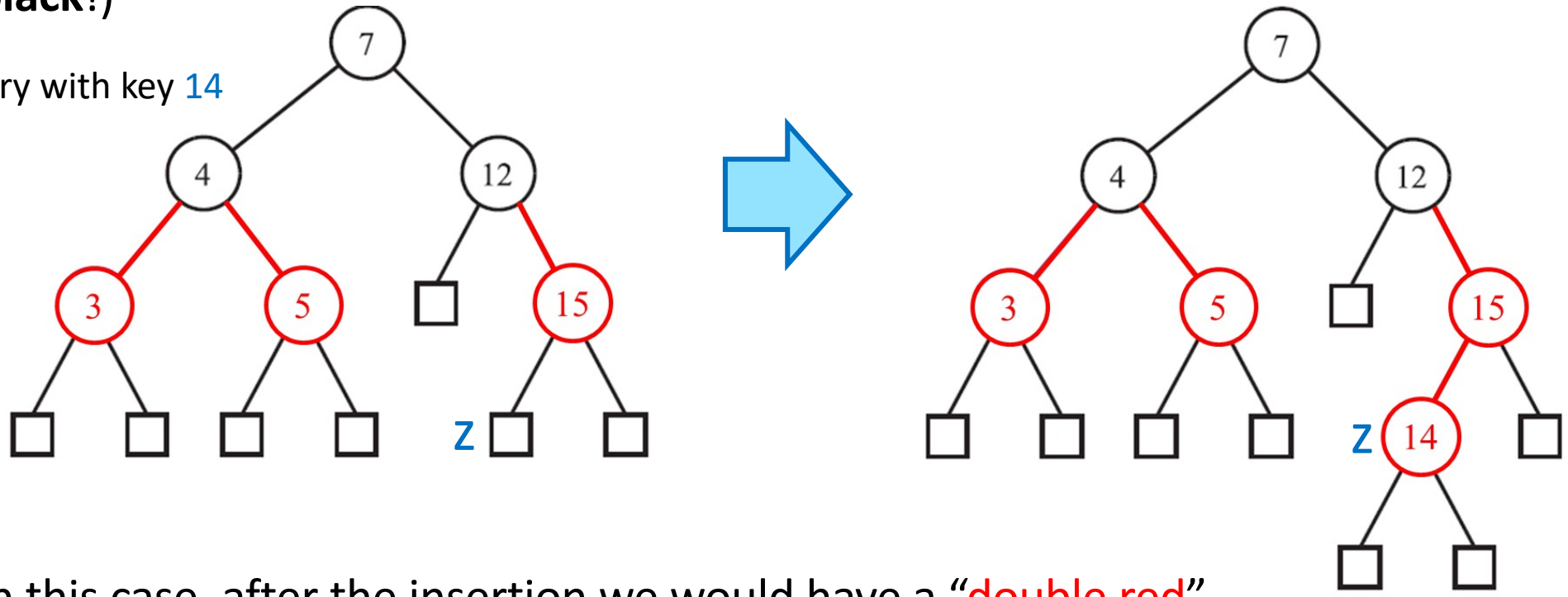



- Insert the entry in z
- Make z red (unless it's the root)
- Give z **black** external children

# RED-BLACK – INSERTION

- The only problem is if z had a **red** parent. In this case, after the insertion we would have a "double red" violation (remember, children of **red** nodes must be **black**!)

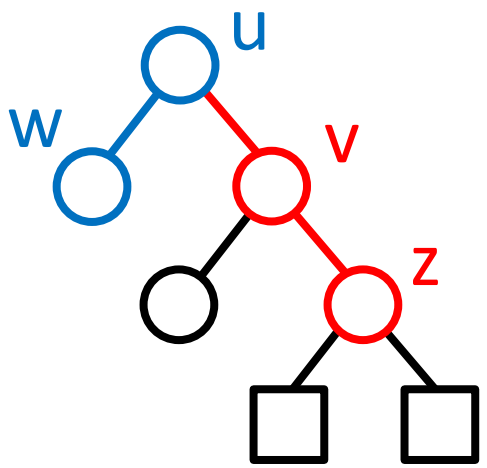
E.g. Insert an entry with key 14




- In this case, after the insertion we would have a "double red" violation (remember, children of red nodes must be **black**!)

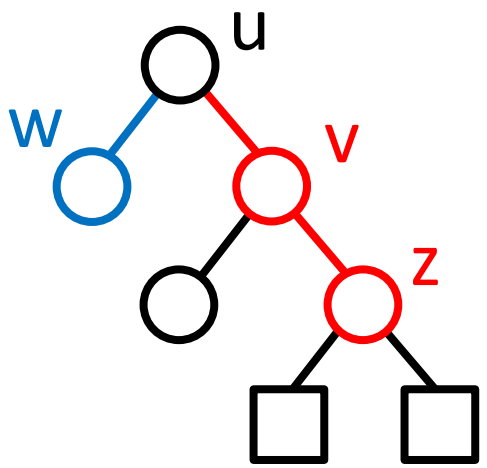
# RED-BLACK – RESOLVING DOUBLE RED

- Let's use blue for when we haven't yet determined whether to use **Red** or **Black**
- Let's analyze the "double red" violation
  - Since v is **red**, it's definitely not the root (because the root of a **Red**-**Black** tree must be **black**).
  - Thus, v has a parent, u



# RED-BLACK – RESOLVING DOUBLE RED

- Let's use blue for when we haven't yet determined whether to use **Red** or **Black**
- Let's analyze the "double red" violation
  - Since v is **red**, it's definitely not the root (because the root of a **Red**-**Black** tree must be **black**).
  - Thus, v has a parent, *u*
  - *u* must be **black**, because if it wasn't, we would have had a double red violation to start with (i.e., before the insertion), and we're assuming that we are inserting in a **Red**-**Black** tree!

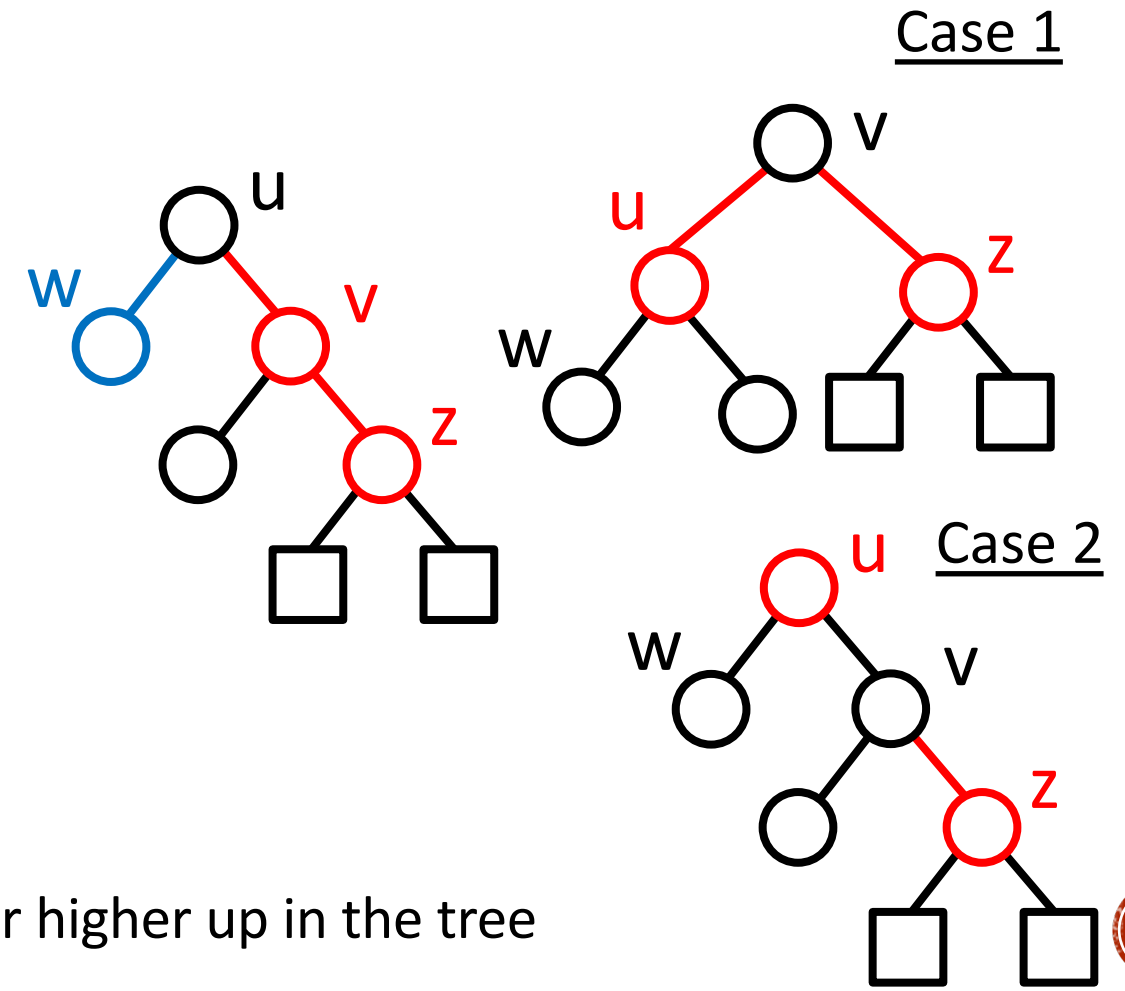


# RED-BLACK – RESOLVING DOUBLE RED

- Let's use blue for when we haven't yet determined whether to use **Red** or **Black**

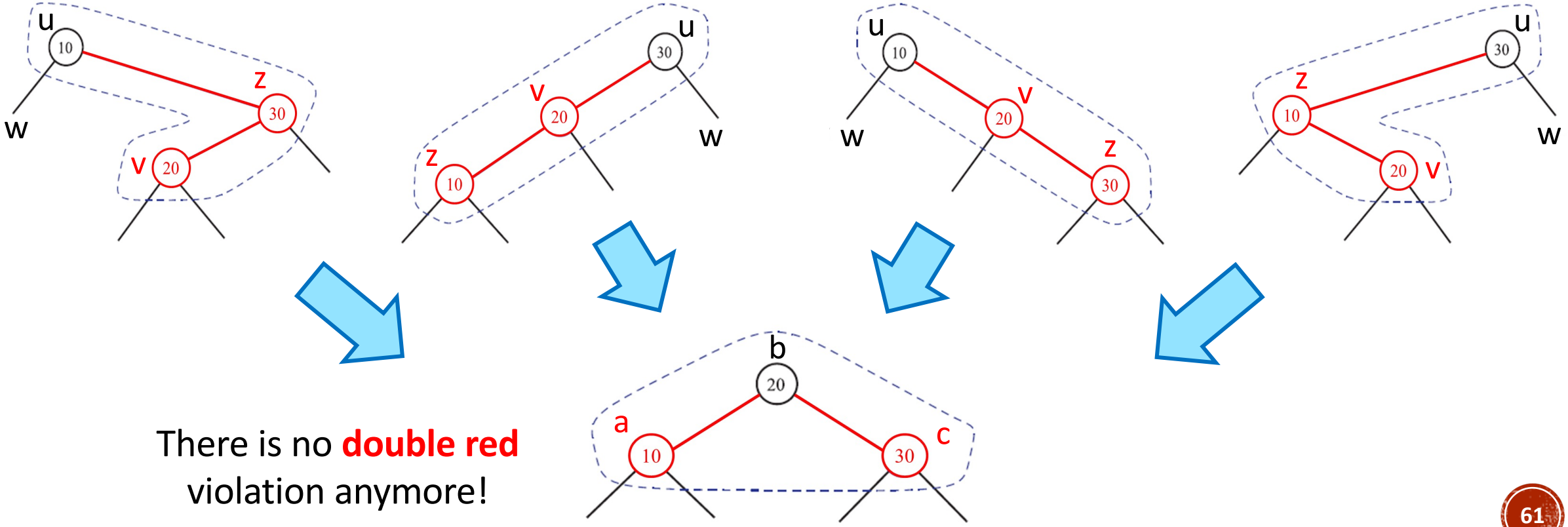
We remedy a double red as follows:

- Case 1: if w (the sibling of v) is **black**
  - Rename (u,v,z) as (a,b,c) and use trinode restructuring
  - Make a and c **red** and b **black**
- Case 2: if w (the sibling of v) is **red**
  - Make v and w **black**
  - Make u **red** (unless it's the root)
  - This is using recoloring
  - After this, double red may re-appear higher up in the tree



# RED-BLACK – RESOLVING DOUBLE RED

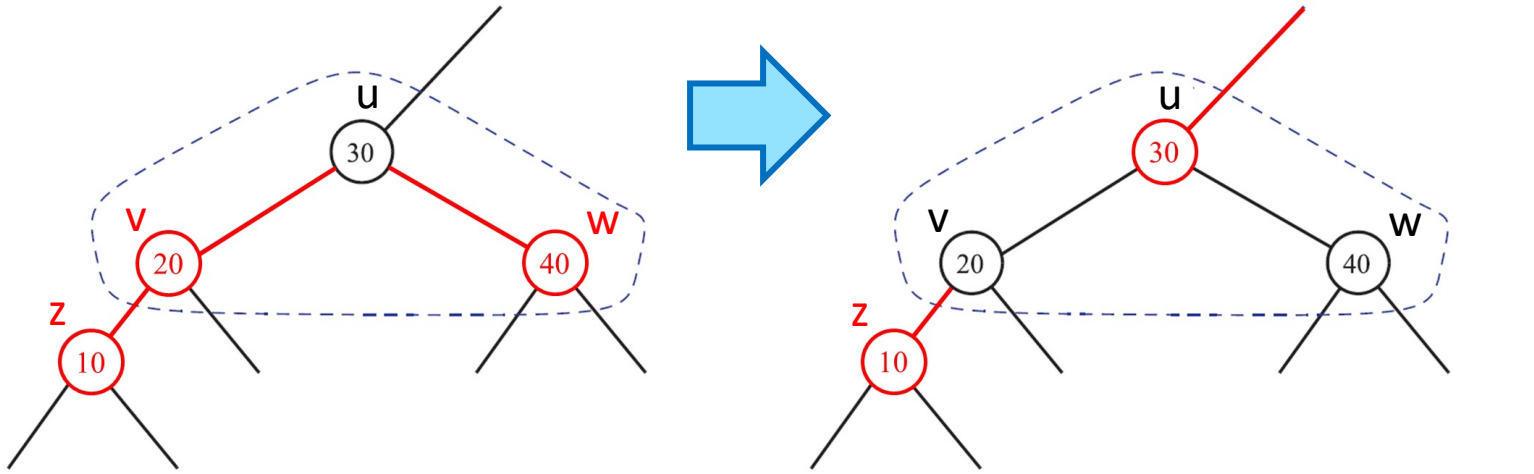
- Case 1: if w is **black**
  - Rename (u,v,z) as (a,b,c) and use trinode restructuring
  - Make a and c **red**, and make b **black**



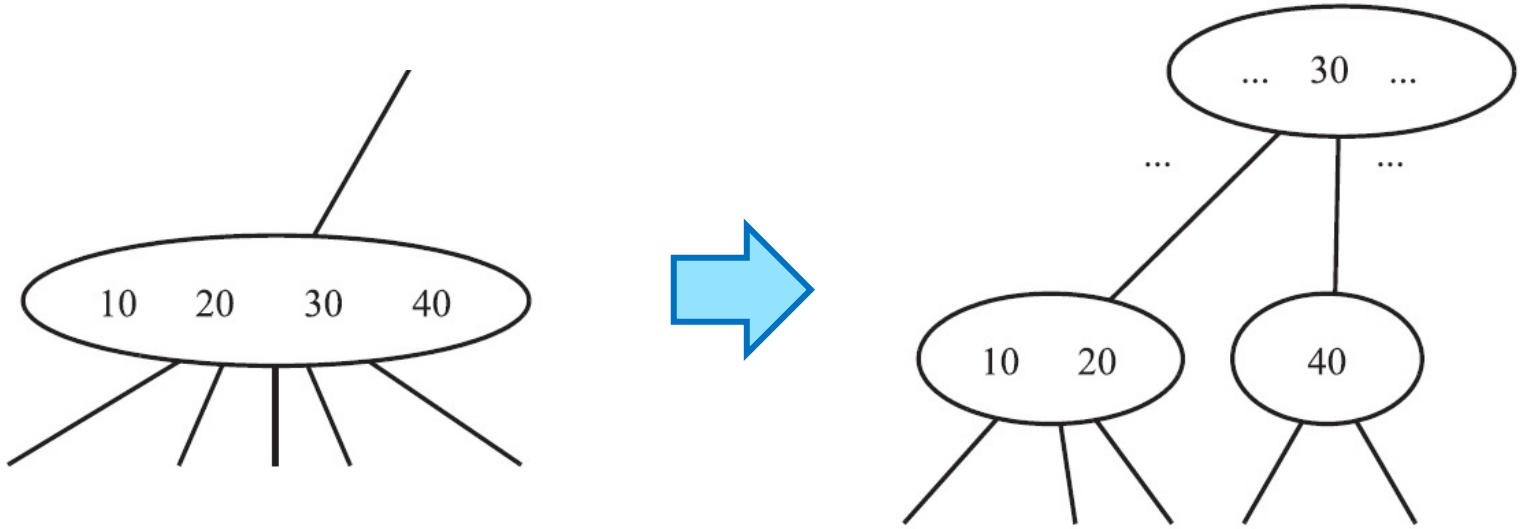

There is no **double red** violation anymore!

# RED-BLACK – RESOLVING DOUBLE RED

- Case 2: if w is **red**
  - Make v and w **black**
  - Make u **red** (unless it's the root)

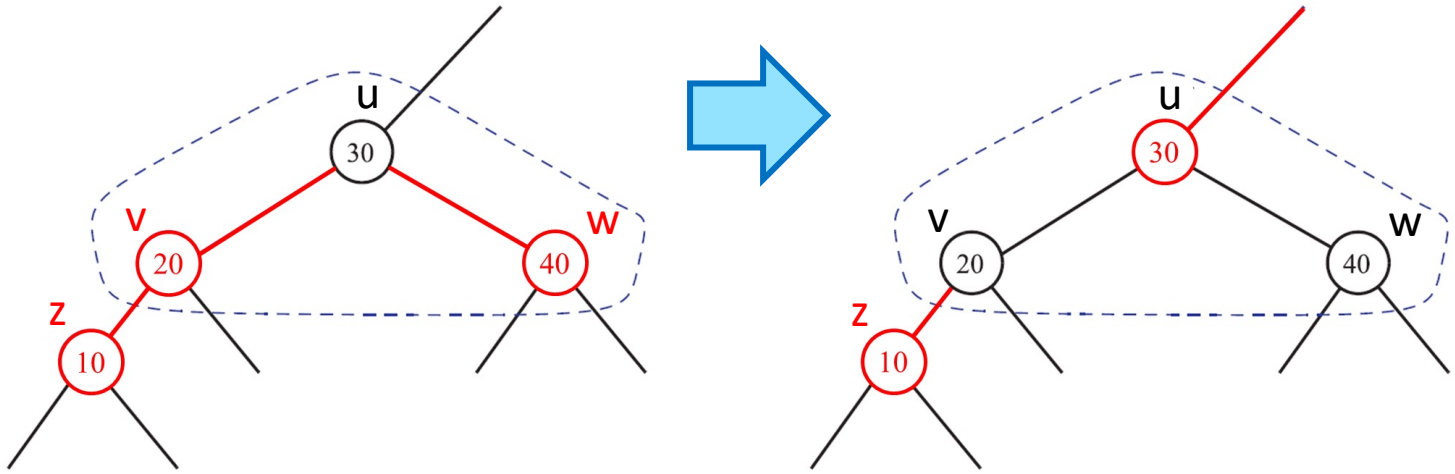



  - This operation, called "recoloring" is equivalent to a "split" operation in the corresponding (2,4) tree!



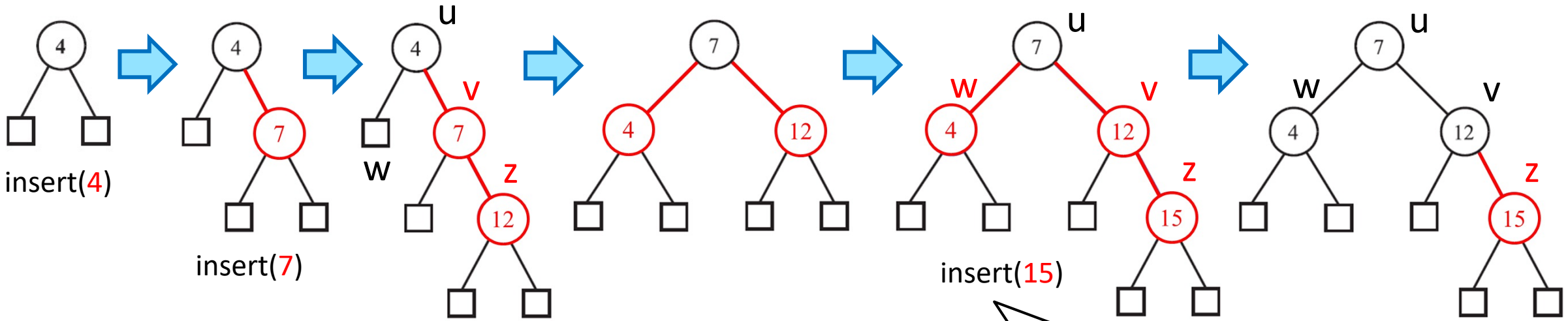
# RED-BLACK – RESOLVING DOUBLE RED

- Case 2: if w is **red**
  - Make v and w **black**
  - Make u **red** (unless it's the root)




- A recoloring operation may introduce a double red violation higher up the tree, but such a violation is guaranteed to be solvable via a recoloring!
- A recoloring operation may introduce yet another double red violation, which would need to be resolved by either trinode restructuring or recoloring, etc.
- Thus, we may need O(log n) recoloring operations, and at most one trinode restructuring (the latter operation never results in another double red violation)

# RED-BLACK – INSERTION EXAMPLES



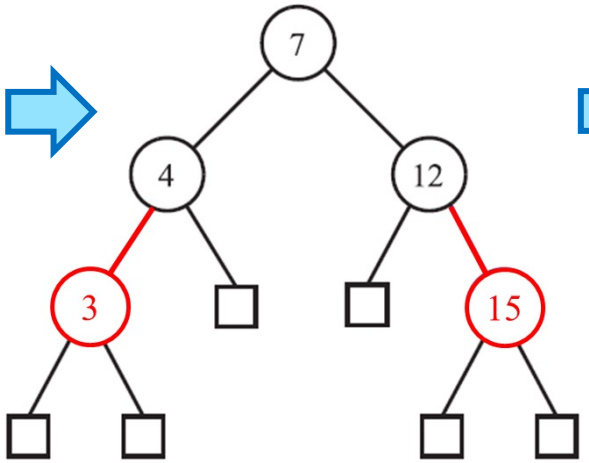

Double red violation!

- Case 1: w is **black**
  - Rename (u,v,z) as (a,b,c) and use trinode restructuring
  - Make a and c **red** and b **black**

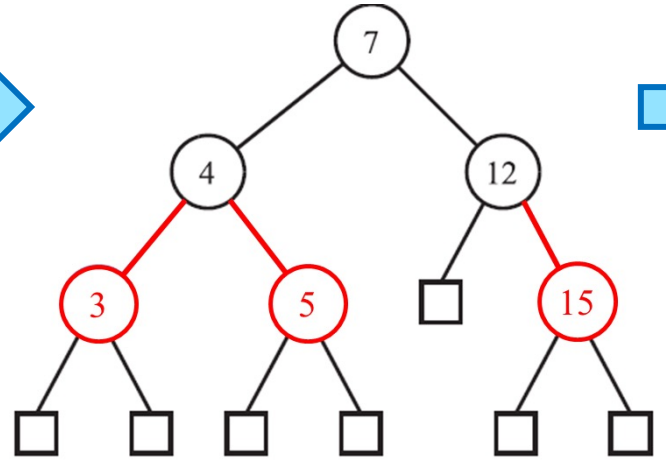
Double red violation!

- Case 2: w is **red** (do recoloring)
  - Make v and w **black**
  - Make u **red** (unless it's the root)

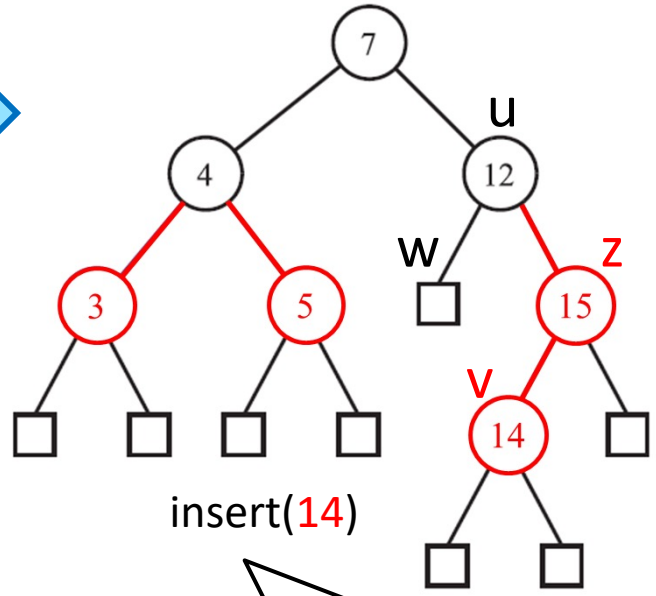
# RED-BLACK – INSERTION EXAMPLES



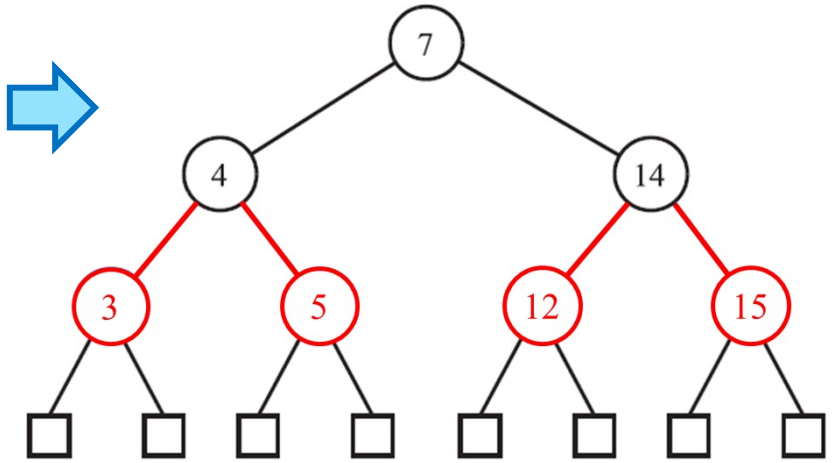

insert(3)




insert(5)




insert(14)




Double red violation!

- Case 1: w is **black**
  - Rename (u,v,z) as (a,b,c) and use trinode restructuring
  - Make a and c red and b **black**