CS350: Data Structures

AA Trees

James Moscola
Department of Engineering & Computer Science
York College of Pennsylvania

Introduction to AA Trees

- **A type of balanced binary search tree**

- **Developed as a simpler alternative to red-black trees and other balanced trees**
  - Introduced by Arne Andersson (hence the AA) in 1993
  - Eliminates many of the conditions that need to be considered to maintain a red-black tree
  - Fewer conditions means AA trees are easier to implement
  - Comparable in performance to red-black trees

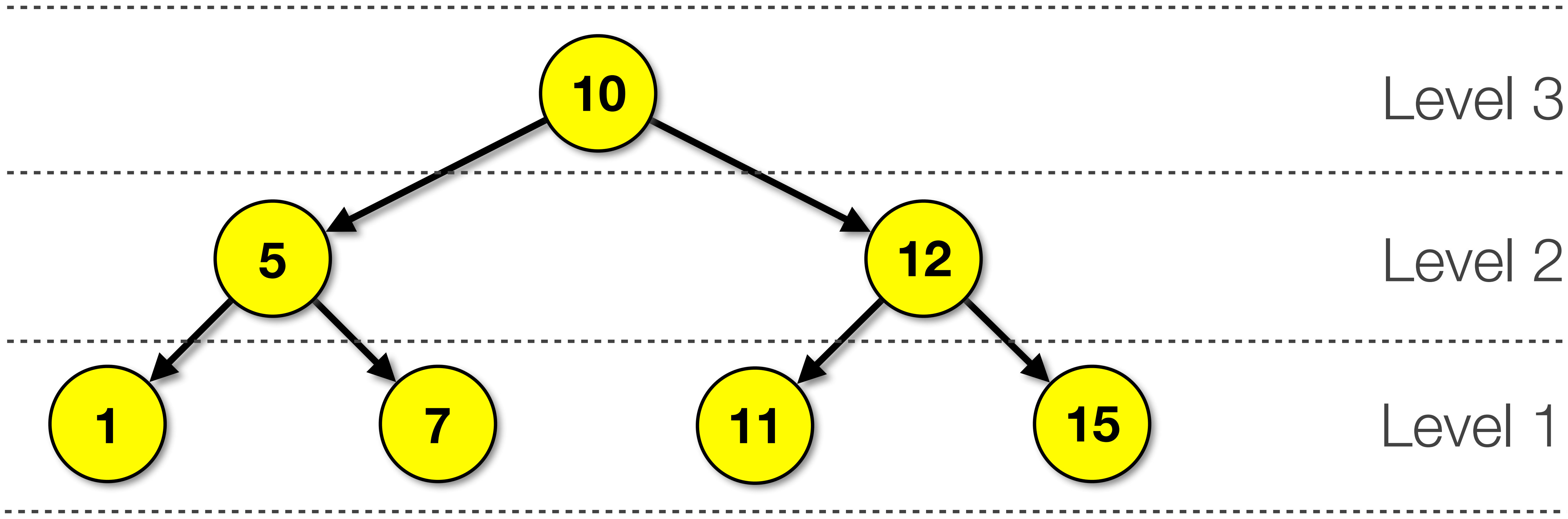
Introduction to AA Trees

- **Similar to red-black trees, but with the addition of a single new rule**
  - Every node is colored either red or black
  - The root node is black
  - If a node is red, its children must be black
  - Every path from a node to a `null` link must contain the same number of black nodes
  - Left children may not be red

New Rule

Levels in AA Trees

- **AA trees utilize the concept of *levels* to aid in balancing binary trees**
  - The level of a node represents the number of left links on the path to the `nullNode` (sentinel node)
- **All leaf nodes are at *level 1***



AA Tree Invariants

- **AA trees must always satisfy the following five invariants:**

1) The *level* of a leaf node is 1

2) The *level* of a left child is strictly less than that of its parent

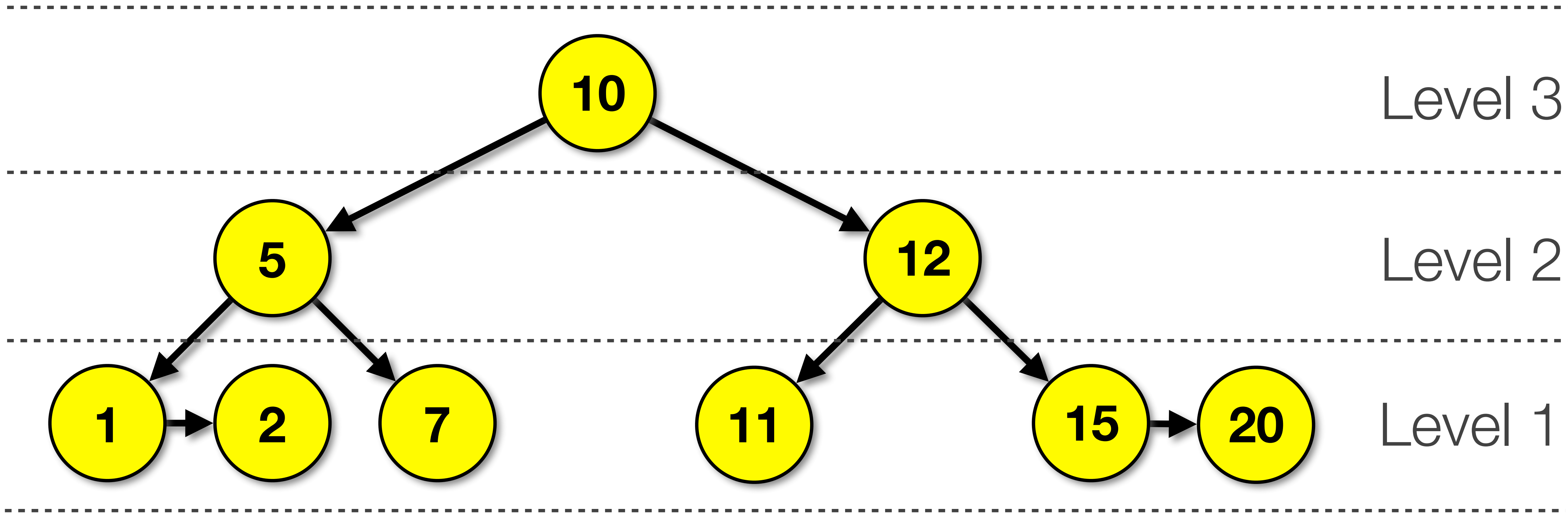
3) The *level* of a right child is less than or equal to that of its parent

4) The *level* of a right grandchild is strictly less than that of its grandparent

5) Every node of *level* greater than one must have two children

Inserting Into AA Trees

- **All nodes are initially inserted as leaf nodes using the standard BST insertion algorithm (tree may require rebalancing after insert)**
- **Since a parent and its <u>right child</u> can be on the same level (*rule #3*), *horizontal links* are possible**

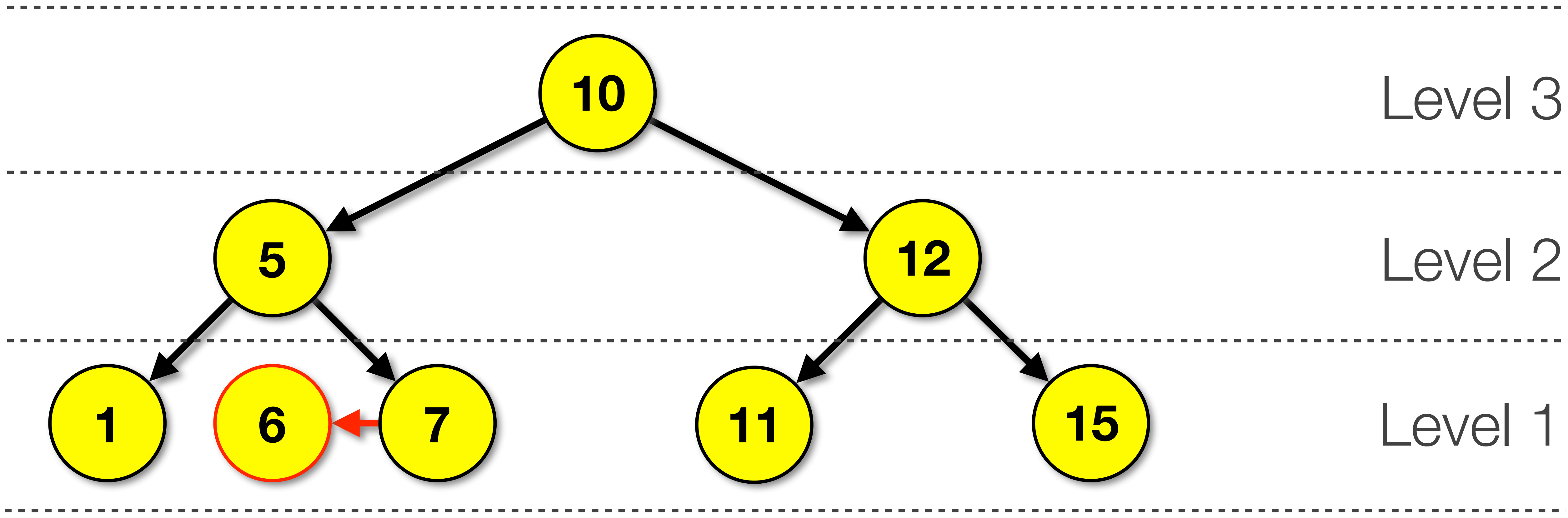


Horizontal Links in AA Trees

- **The five invariants of AA trees impose restrictions on horizontal links**
- **If any of the invariants are violated the tree must be modified until it once again satisfies all five invariants**
- **Only two cases need to be considered and corrected to maintain the balance of an AA tree**

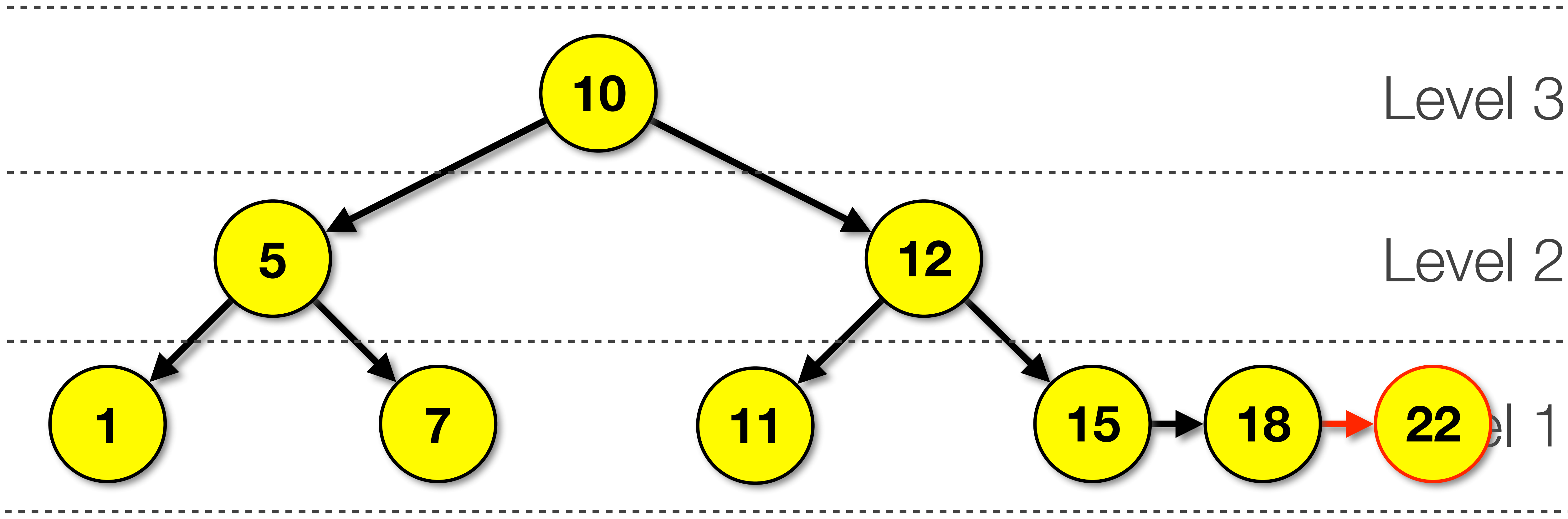
Horizontal Links in AA Trees

- **Case #1 - Left horizontal links are NOT allowed**
  - Violates rule #2 - the *level* a left child is strictly less than that of its parent
  - A ***skew*** operation will be introduced to handle this case



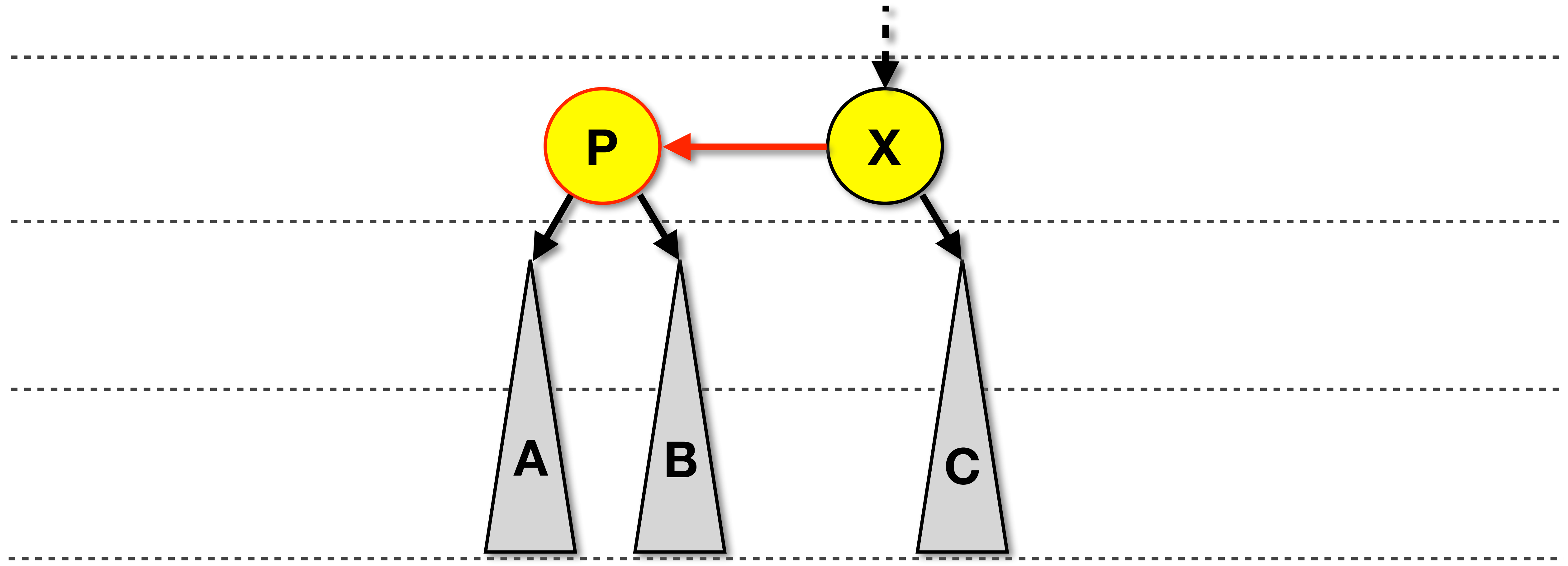
Horizontal Links in AA Trees

- **Case #2 - Two consecutive right horizontal links are NOT allowed**
  - Violates rule #4 - the *level* of a right grandchild is strictly less than that of its grandparent
  - A ***split*** operation will be introduced to handle this case



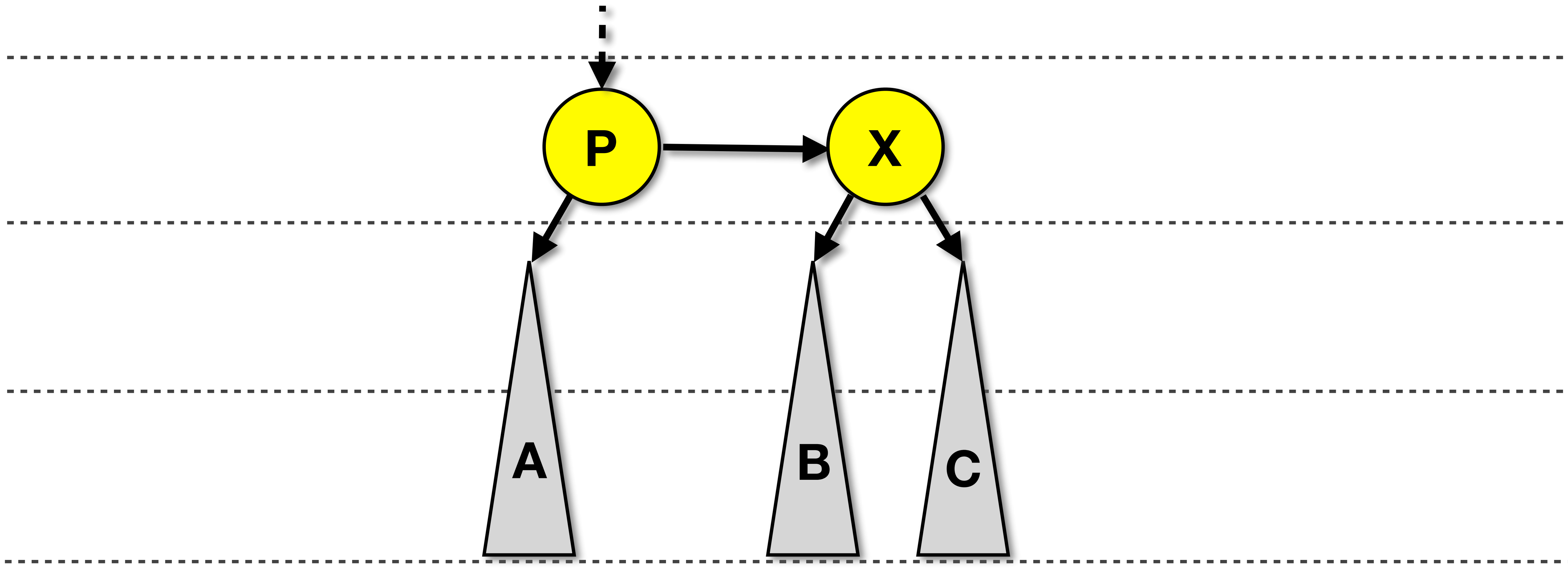
The *skew* Operation

- **The *skew* operation is a single right rotation when an insertion (or deletion) creates a left horizontal link**
  - Removes the left horizontal link
  - May create consecutive right horizontal links in process



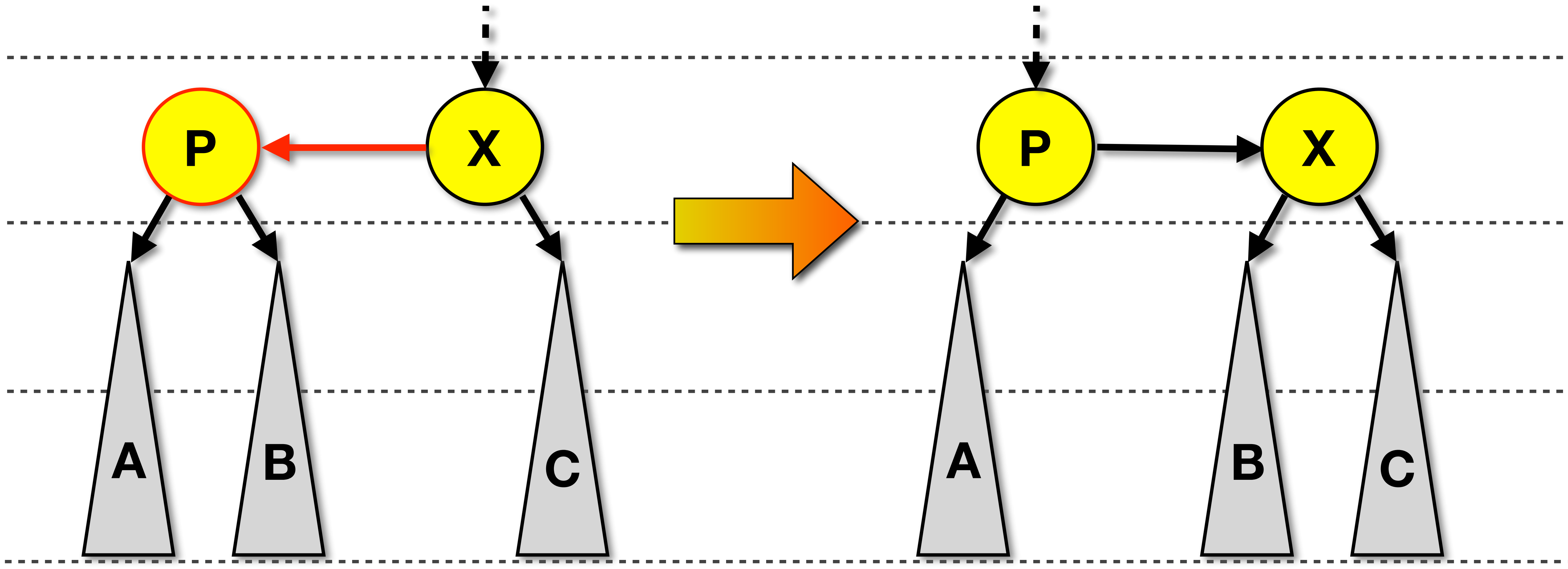
The *skew* Operation

- **The *skew* operation is a single right rotation when an insertion (or deletion) creates a left horizontal link**
  - Removes the left horizontal link
  - May create consecutive right horizontal links in process



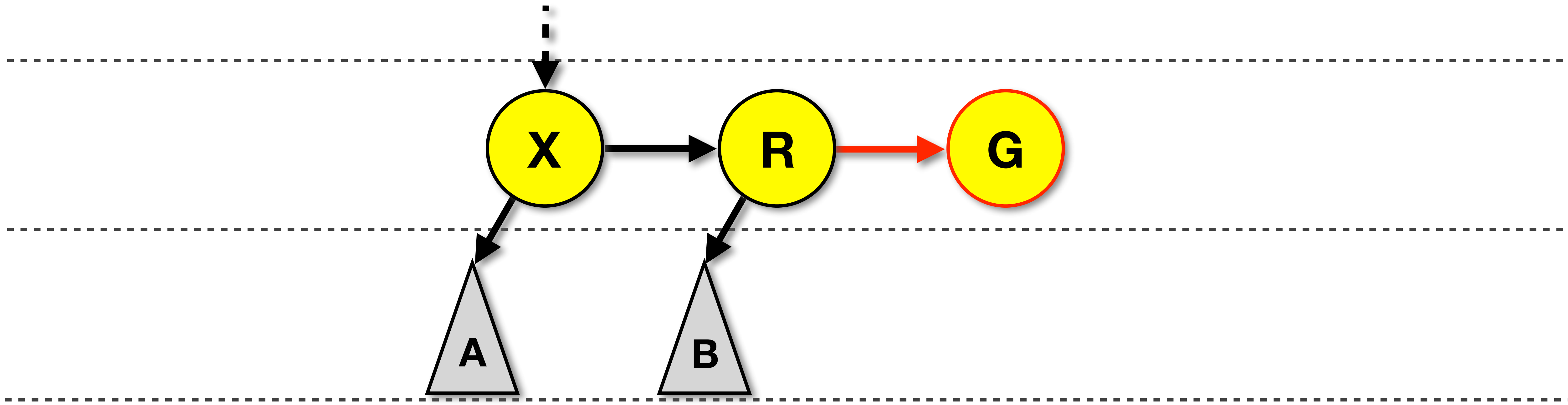
The ***skew*** Operation

- **The *skew* operation is a single right rotation when an insertion (or deletion) creates a left horizontal link**
  - Removes the left horizontal link
  - May create consecutive right horizontal links in process



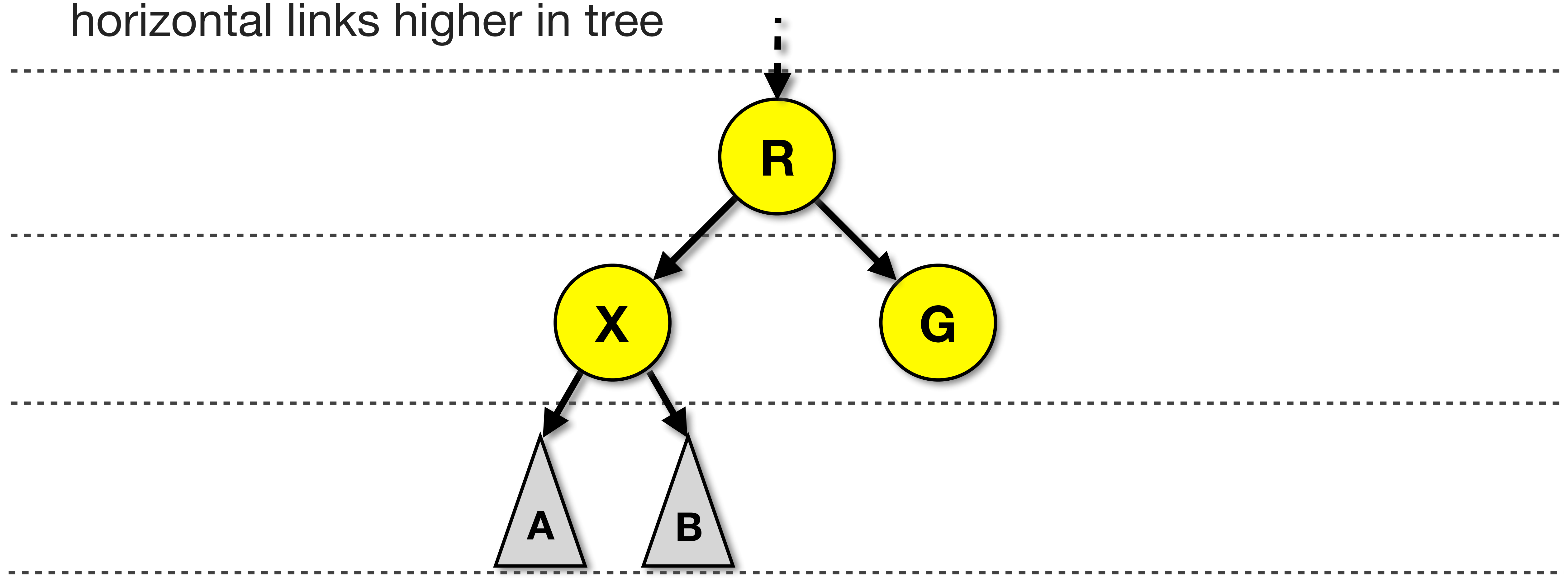
The *split* Operation

- **The *split* operation is a single left rotation when an insertion (or deletion) creates two consecutive right horizontal links**
  - Removes two consecutive right horizontal links
  - Increases level of middle node which may cause problems invalid horizontal links higher in tree



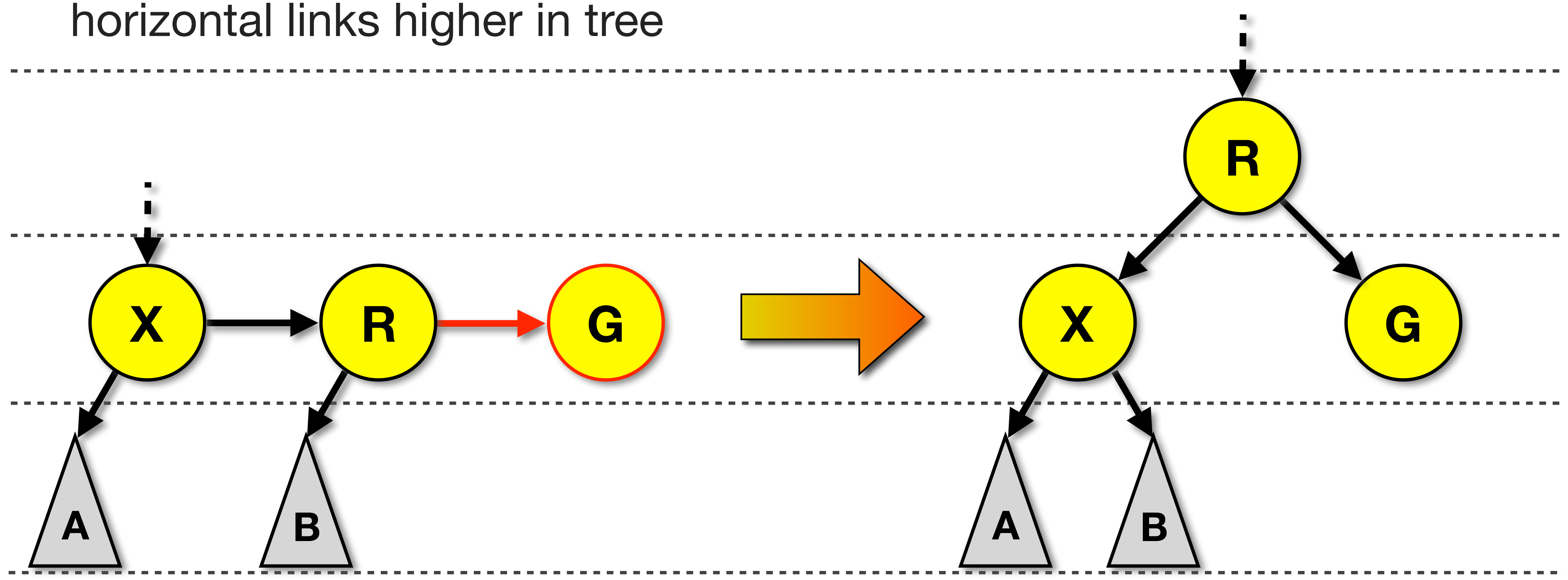
The *split* Operation

- **The *split* operation is a single left rotation when an insertion (or deletion) creates two consecutive right horizontal links**
  - Removes two consecutive right horizontal links
  - Increases level of middle node which may cause problems invalid horizontal links higher in tree



The *split* Operation

- **The *split* operation is a single left rotation when an insertion (or deletion) creates two consecutive right horizontal links**
  - Removes two consecutive right horizontal links
  - Increases level of middle node which may cause problems invalid horizontal links higher in tree



Implementation of ***insert***

```java
/**
  * Internal method to insert into a subtree.
  * @param x the item to insert.
  * @param t the node that roots the tree.
  * @return the new root.
  * @throws DuplicateItemException if x is already present.
  */
 private AANode<AnyType> insert( AnyType x, AANode<AnyType> t )
 {
     if( t == nullNode )
         t = new AANode<AnyType>( x, nullNode, nullNode );
     else if( x.compareTo( t.element ) < 0 )
         t.left = insert( x, t.left );
     else if( x.compareTo( t.element ) > 0 )
         t.right = insert( x, t.right );
     else
         throw new DuplicateItemException( x.toString( ) );

     t = skew( t );
     t = split( t );
     return t;
 }
```

Implementation of ***skew***

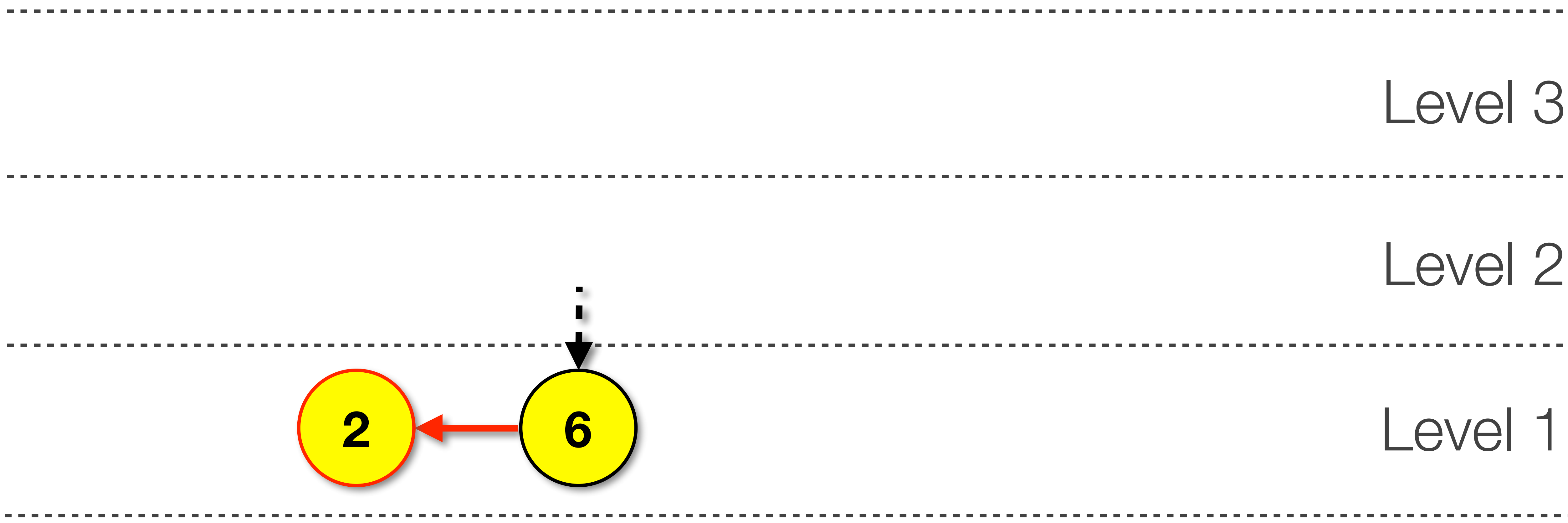
```java
/**
  * Skew primitive for AA-trees.
  * @param t the node that roots the tree.
  * @return the new root after the rotation.
  */
 private static AANode<AnyType> skew( AANode<AnyType> t )
 {
     if( t.left.level == t.level )
         t = rotateWithLeftChild( t );
     return t;
 }
```

Implementation of ***split***

```
/**
   * Split primitive for AA-trees.
   * @param t the node that roots the tree.
   * @return the new root after the rotation.
   */
  private static AANode<AnyType> split( AANode<AnyType> t )
  {
      if( t.right.right.level == t.level )
      {
          t = rotateWithRightChild( t );
          t.level++;
      }
      return t;
```

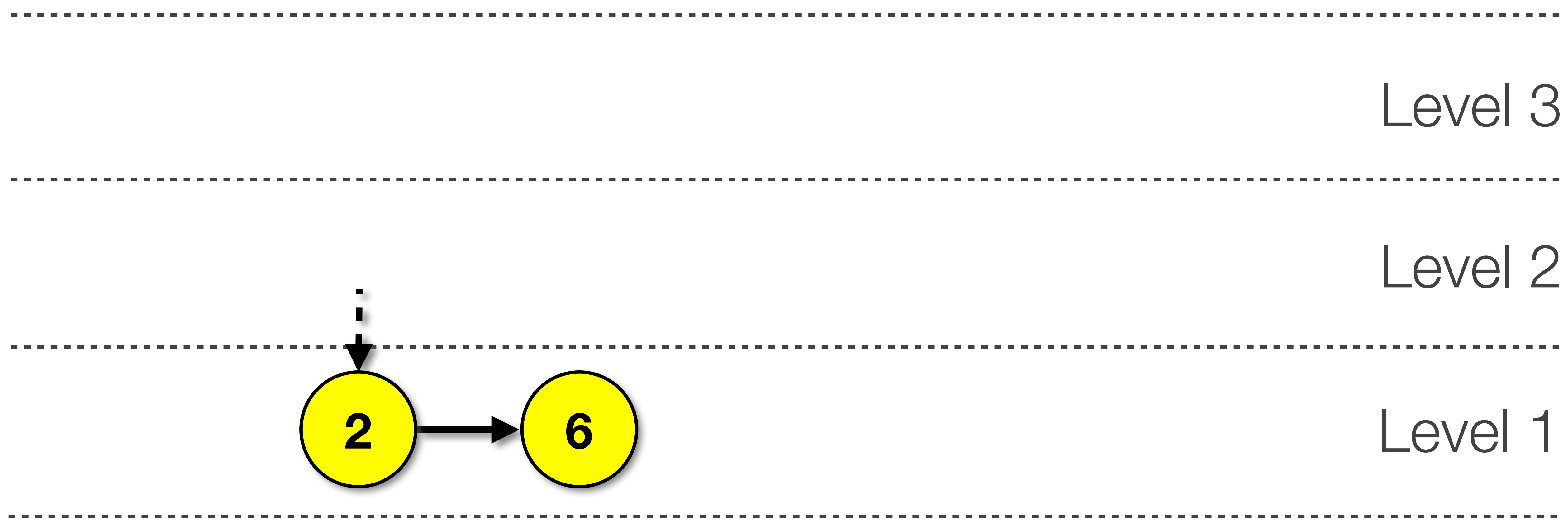
Example of Insertion

Inserts: 6 2



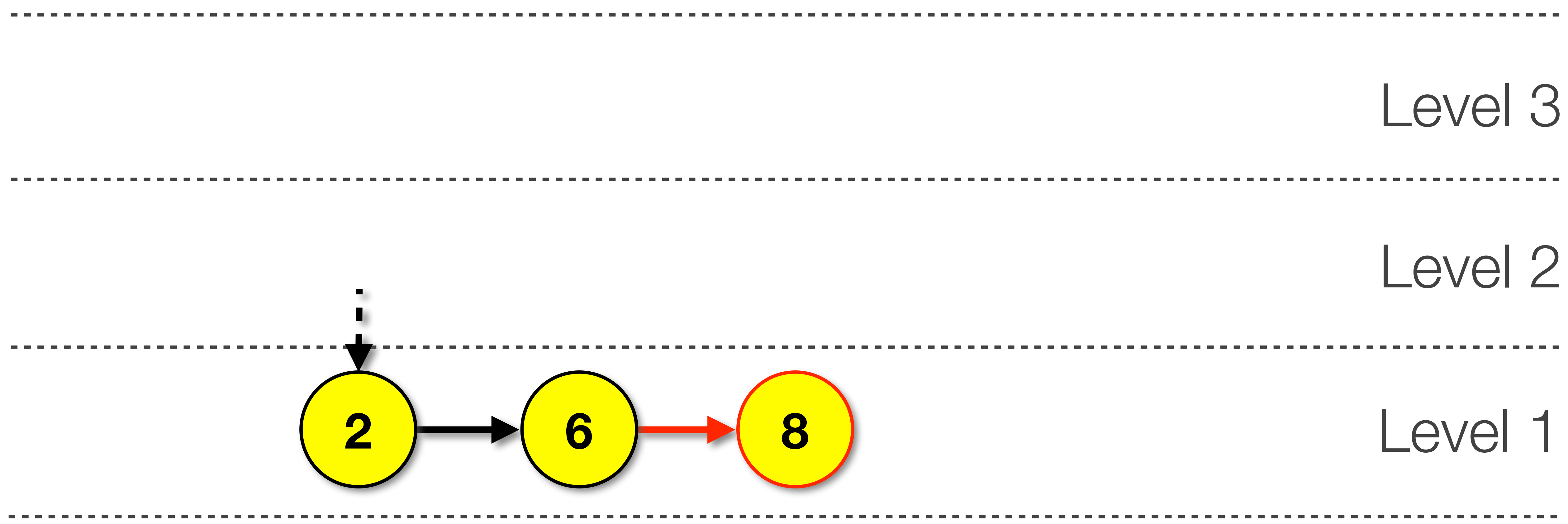
Example of Insertion

Inserts: 6 2



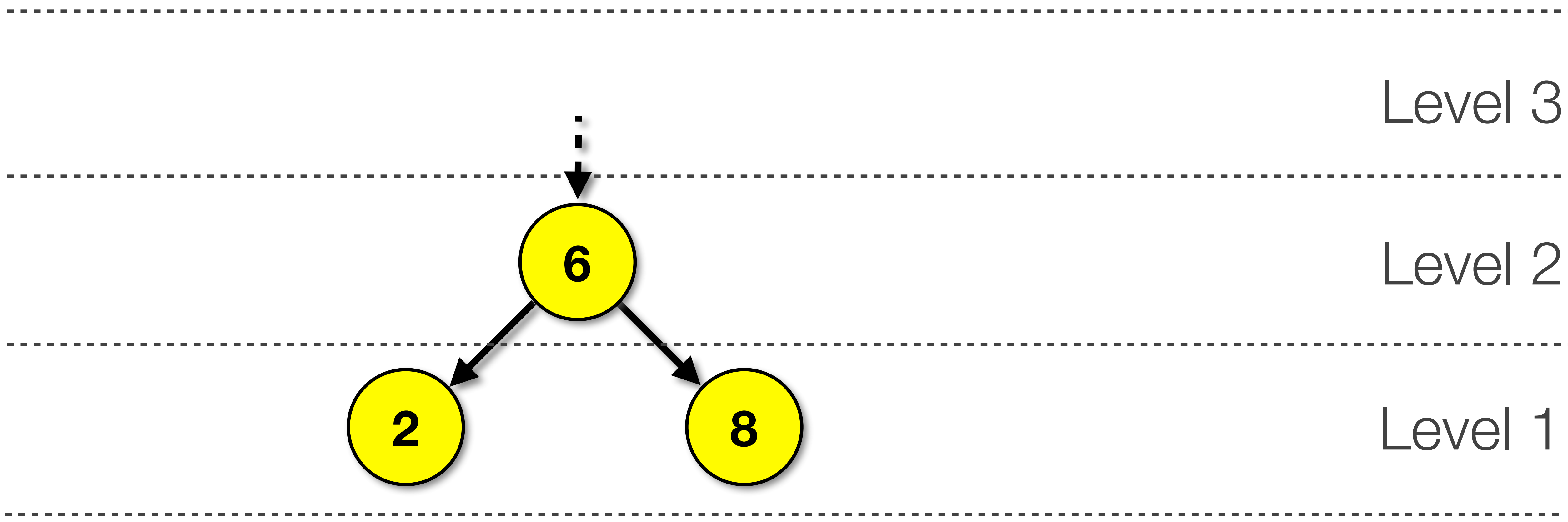
Example of Insertion

Inserts: 6 2 8



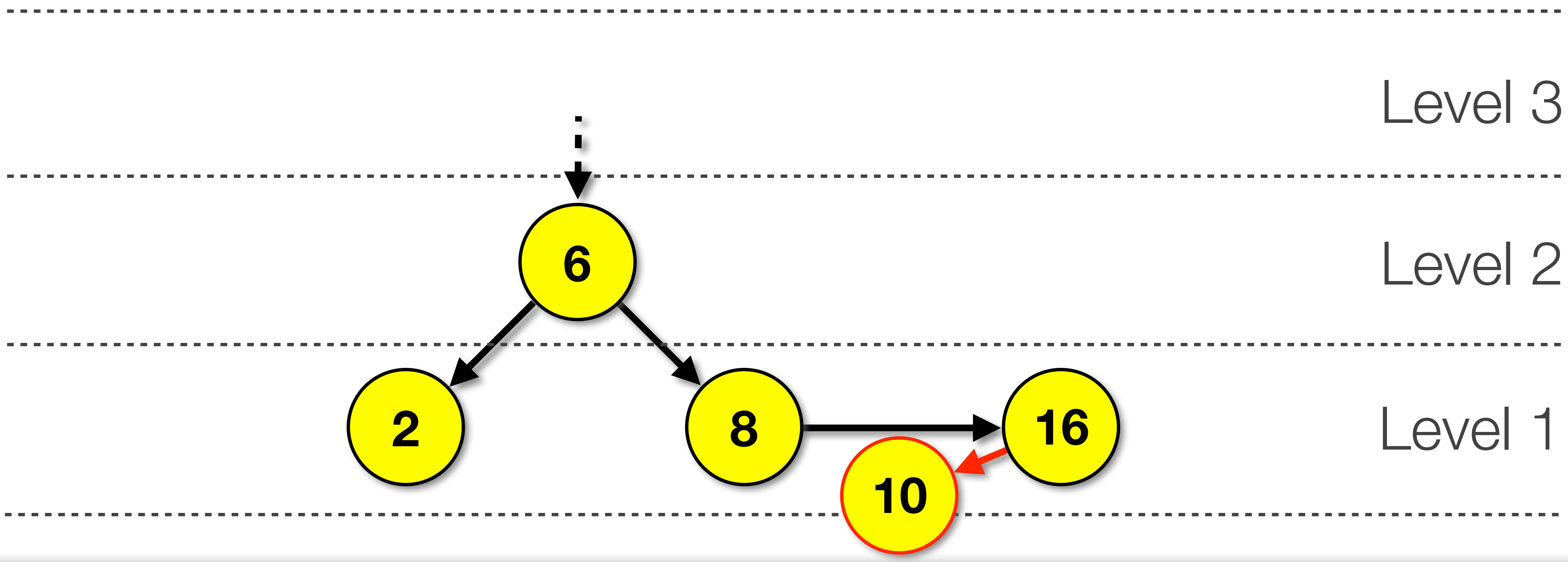
Example of Insertion

Inserts: 6 2 8



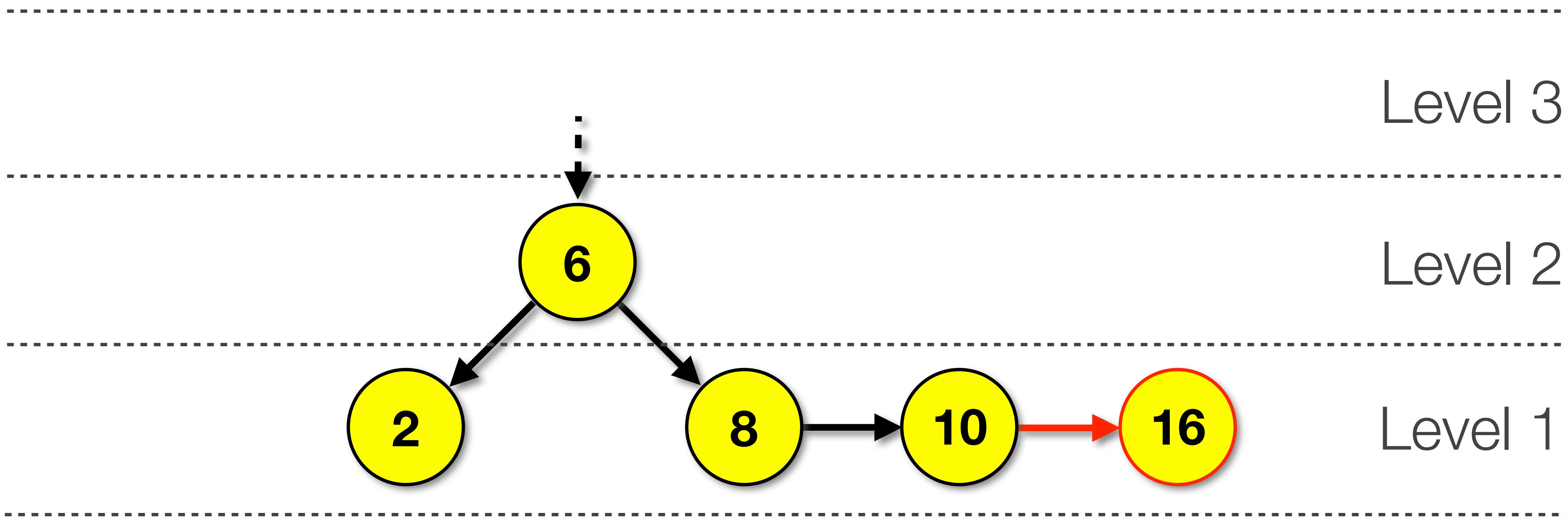
Example of Insertion

Inserts: 6 2 8 16 10



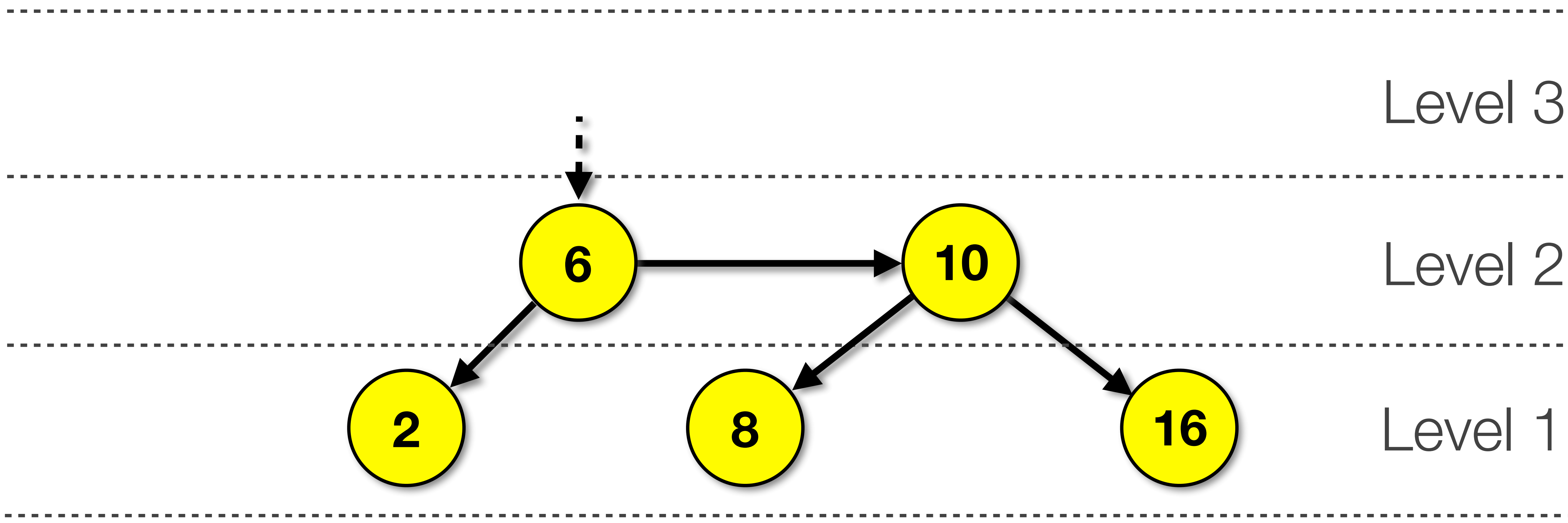
Example of Insertion

<u>Inserts:</u> 6 2 8 16 10



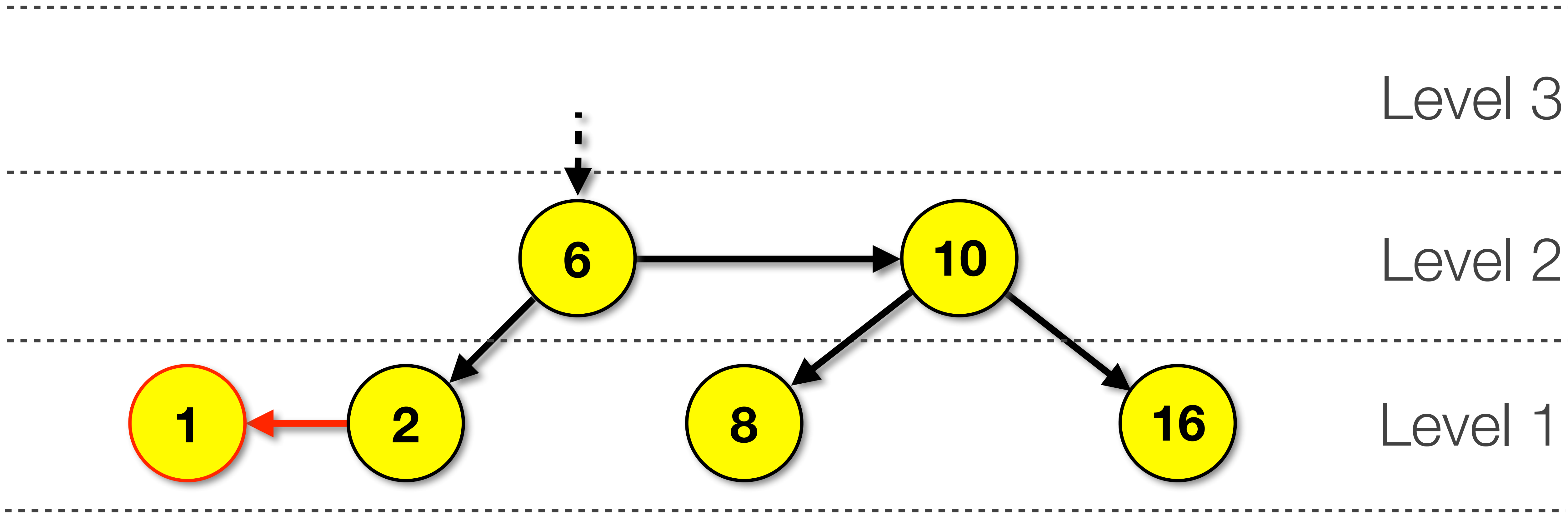
Example of Insertion

Inserts: 6 2 8 16 10



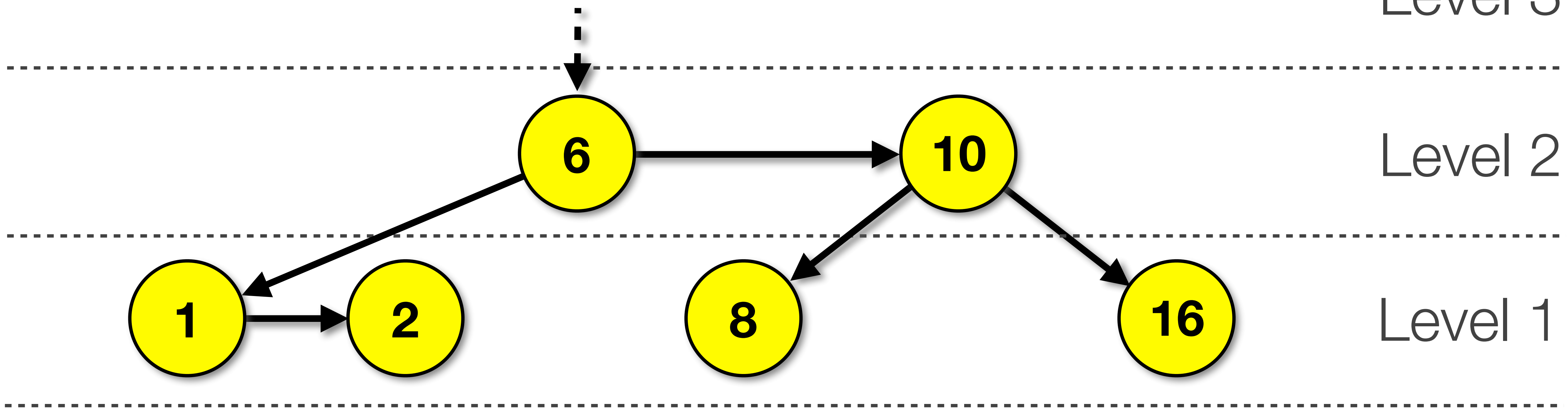
Example of Insertion

Inserts: 6 2 8 16 10 1



Example of Insertion

Inserts: 6 2 8 16 10 1



Deleting From AA Trees

- **Perform a *recursive* deletion just like on other BSTs:**
  - To delete a leaf node (no children), simply remove the node
  - To delete a node with one child, replace node with child (in AA trees the child node will be a right child / both nodes at level 1)
  - To delete an internal node, replace that node with either its successor or predecessor

- **May need to rebalance AA tree after a deletion occurs**

Fixing an Unbalanced AA Tree

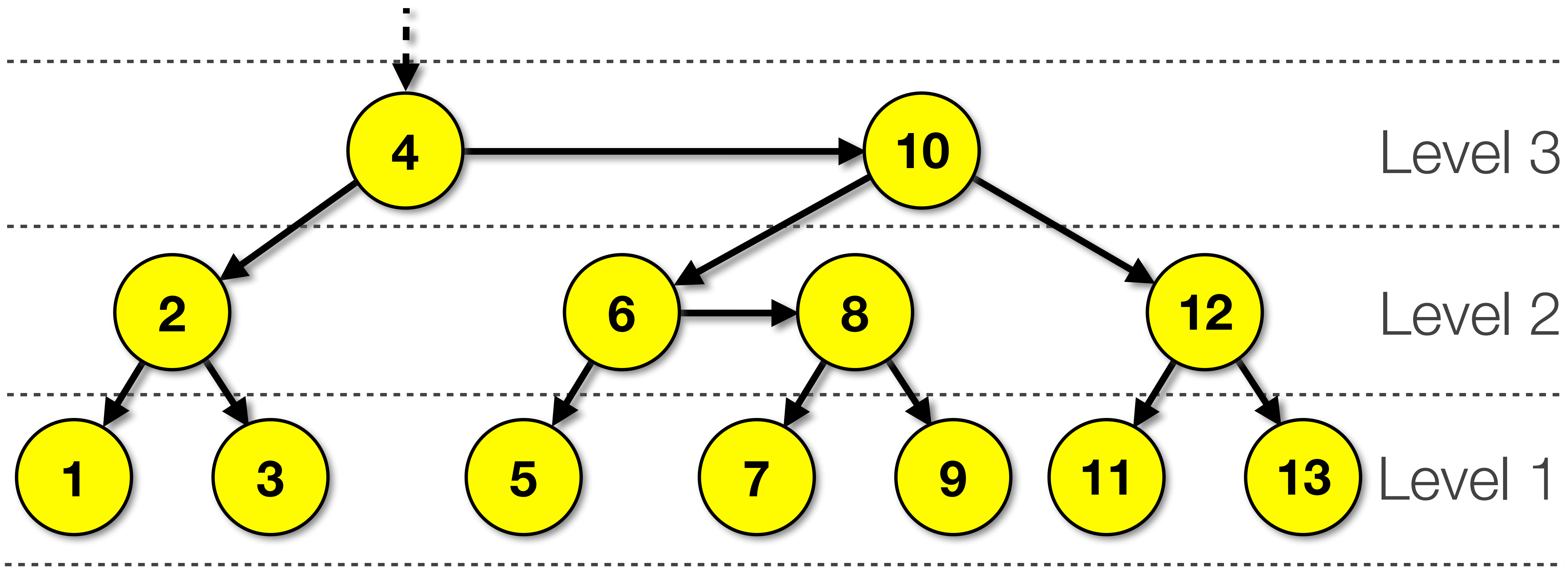
1) **Decrease the level of a node when:**
   - Either of the nodes children are more than one level down
     (Note that a null sentinel node is at level 0)
   - A node is the right horizontal child of another node whose level was decreased
2) **Skew the level of the node whose level was decremented (3 skews)**
   - Skew the subtree from the root, where the decremented node is the root
     (may alter the root node of the subtree)
   - Skew the root node's right child
   - Skew the root node's right-right child
3) **Split the level of the node whose level was decremented (2 splits)**
   - Split the root node of the subtree
     (may alter the root node of the subtree)
   - Split the root node's right child

Excerpt From ***remove***

```
// Rebalance tree
if( t.left.level < t.level - 1 || t.right.level < t.level - 1 )  // check level of children
{
    if( t.right.level > --t.level )         // check level of right horizontal children
        t.right.level = t.level;            //       and decrement if necessary
    t = skew( t );                          // First skew (may alter current root)
    t.right = skew( t.right );              // Second skew
    t.right.right = skew( t.right.right );  // Third skew
    t = split( t );                         // First split (may alter current root)
    t.right = split( t.right );             // Second split
}
```

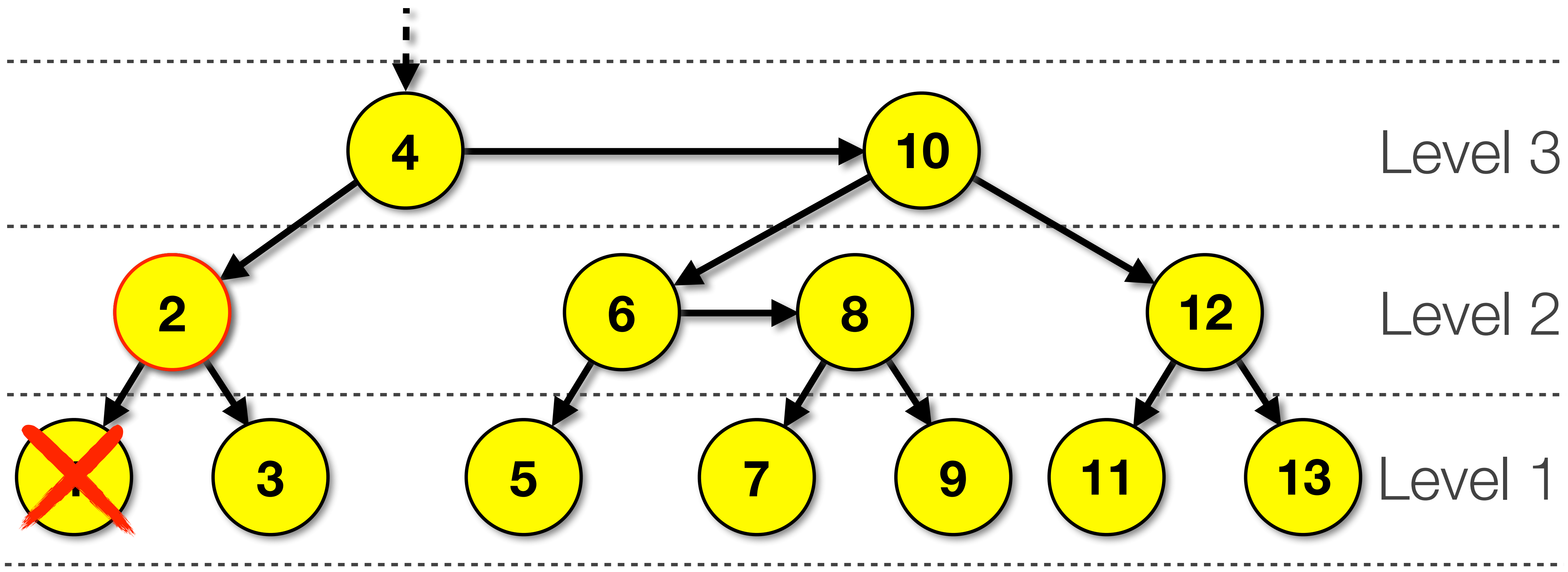
Example of Deletion

This tree can be recreated with the following sequence of inserts:
4, 10, 2, 6, 12, 3, 1, 8, 13, 11, 5, 9, 7



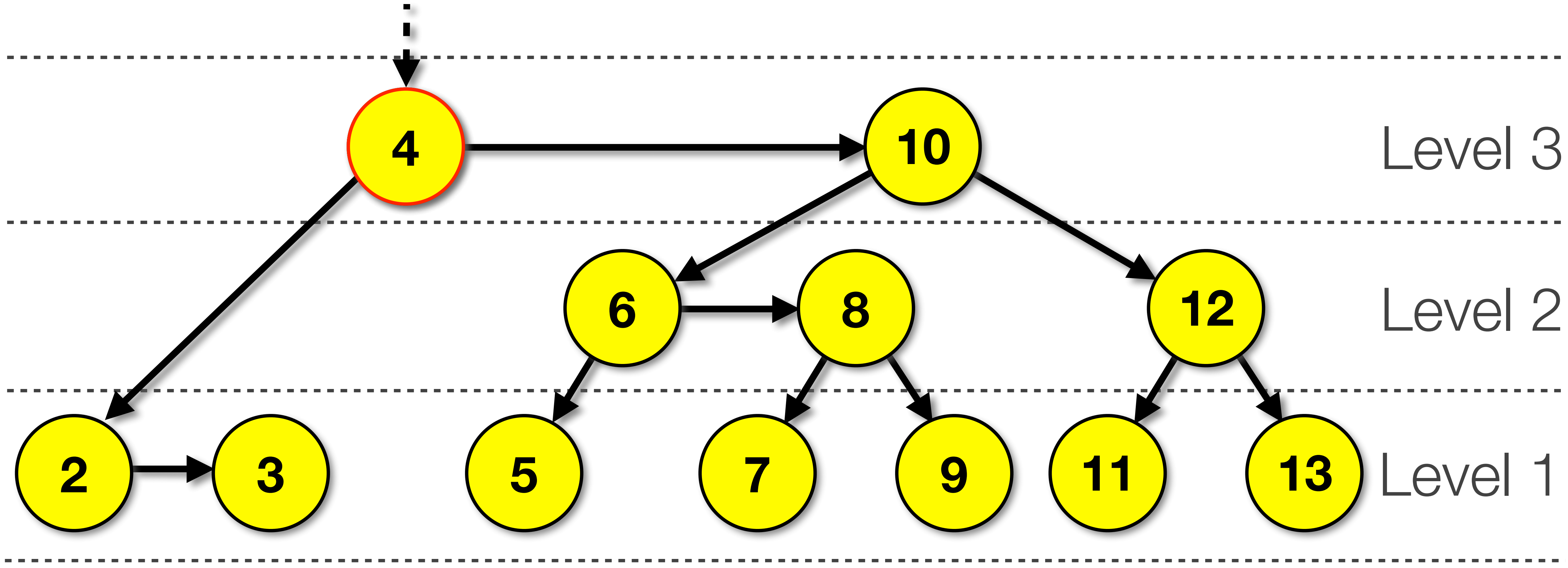
Example of Deletion

Delete node 1
Node 2 now violates rule #5



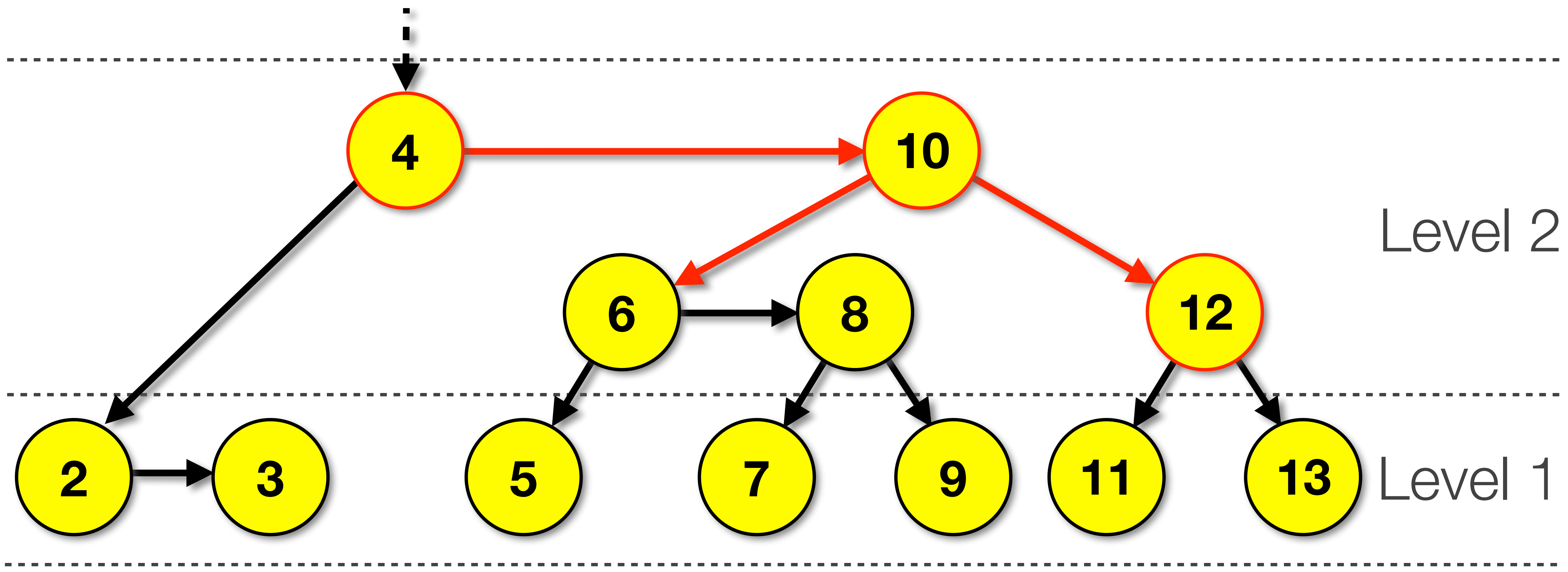
Example of Deletion

Decrement the level of node 2
Node 4 is more than one level above child



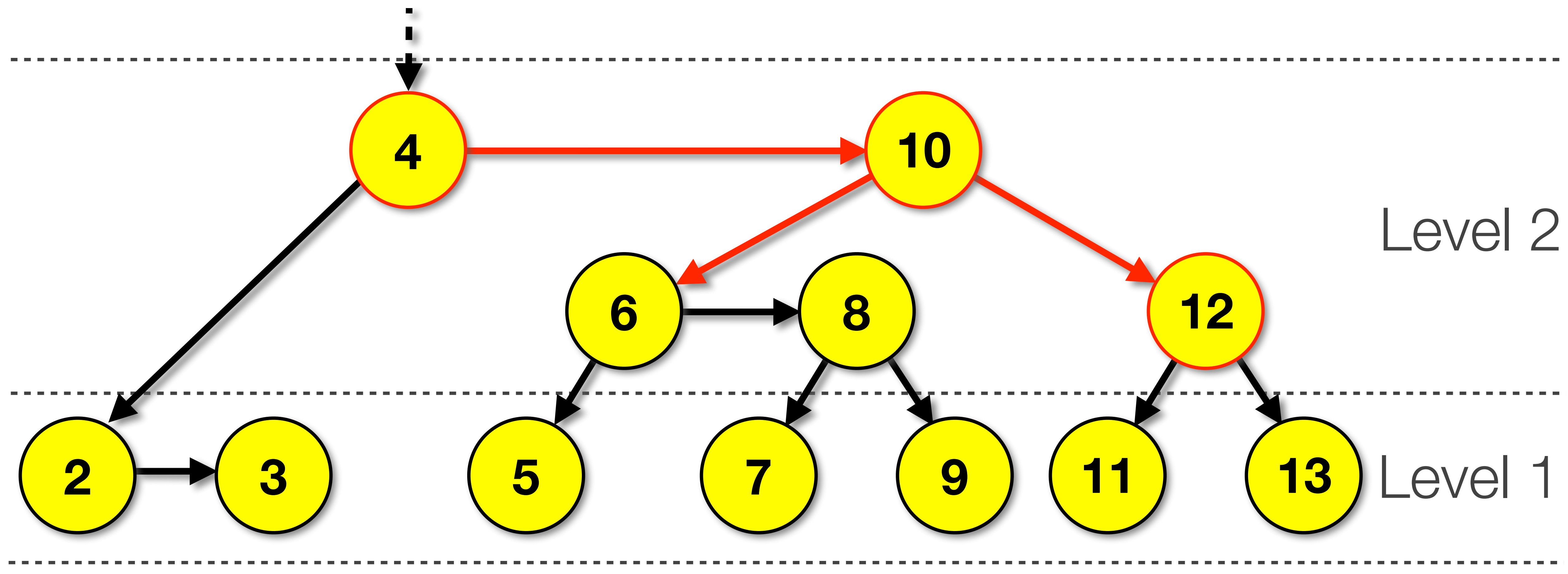
Example of Deletion

Decrement the level of nodes 4 and 10
Node 4 now has two consecutive right links
Node 10 now has left horizontal link



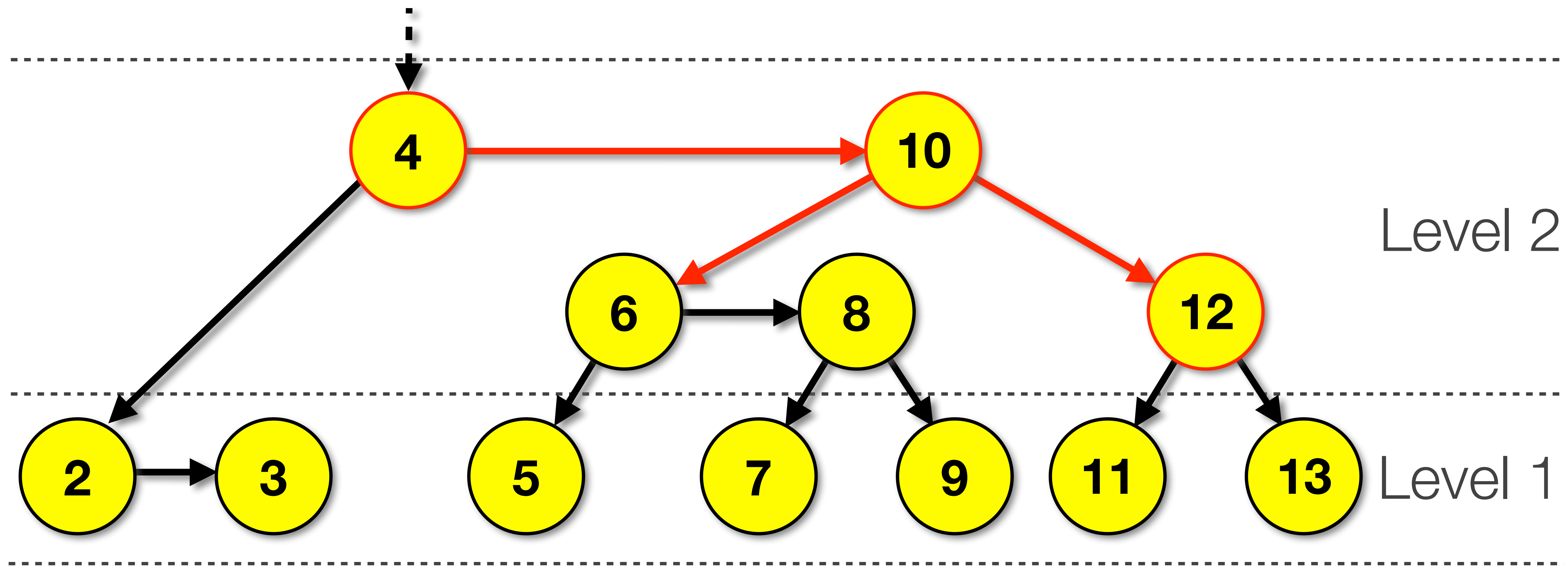
Example of Deletion

No more level decrementing necessary
Start triple-skew, double-split process



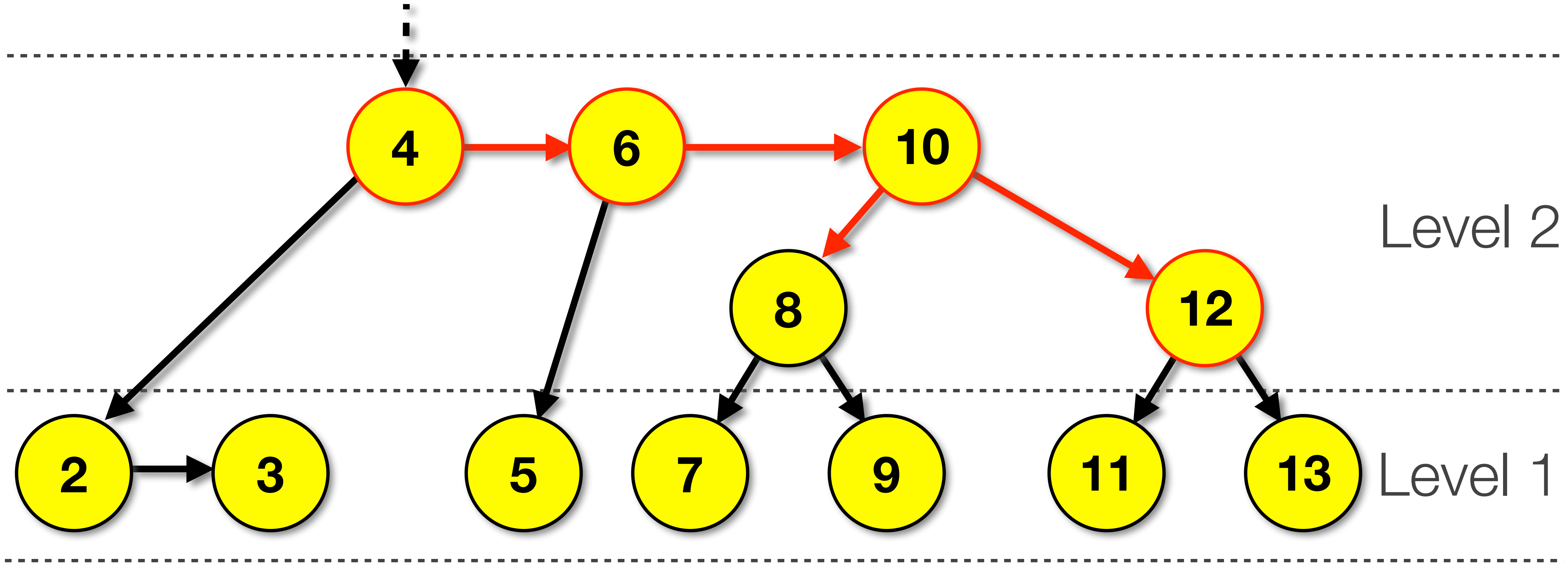
Example of Deletion

Skew node 4 (does nothing)
Next skew 4.right (node 10)



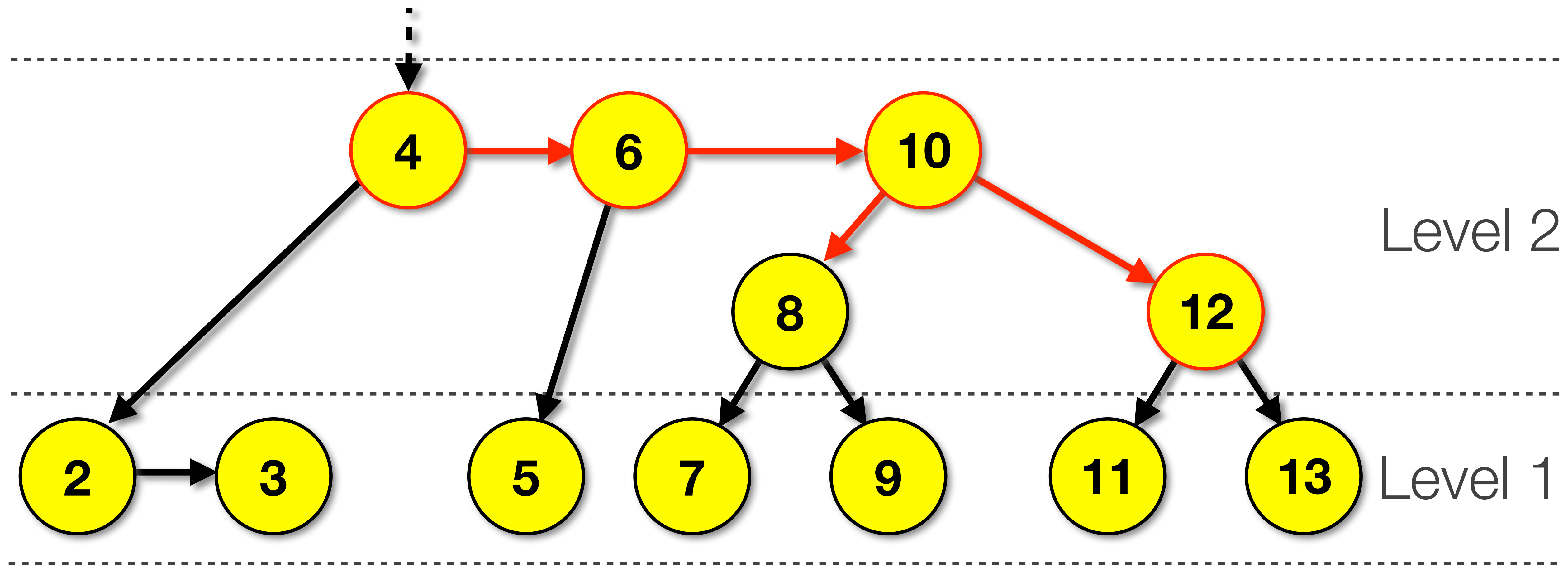
Example of Deletion

After skew 4.right (node 10)
Next skew 4.right.right (node 10 again)



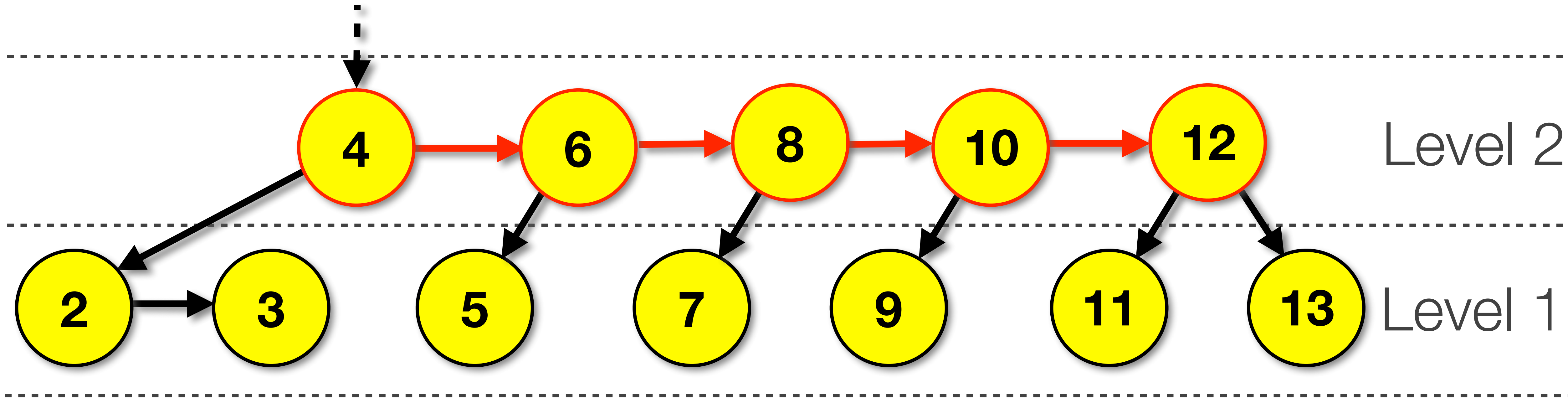
Example of Deletion

After skew 4.right (node 10)
Next skew 4.right.right (node 10 again)



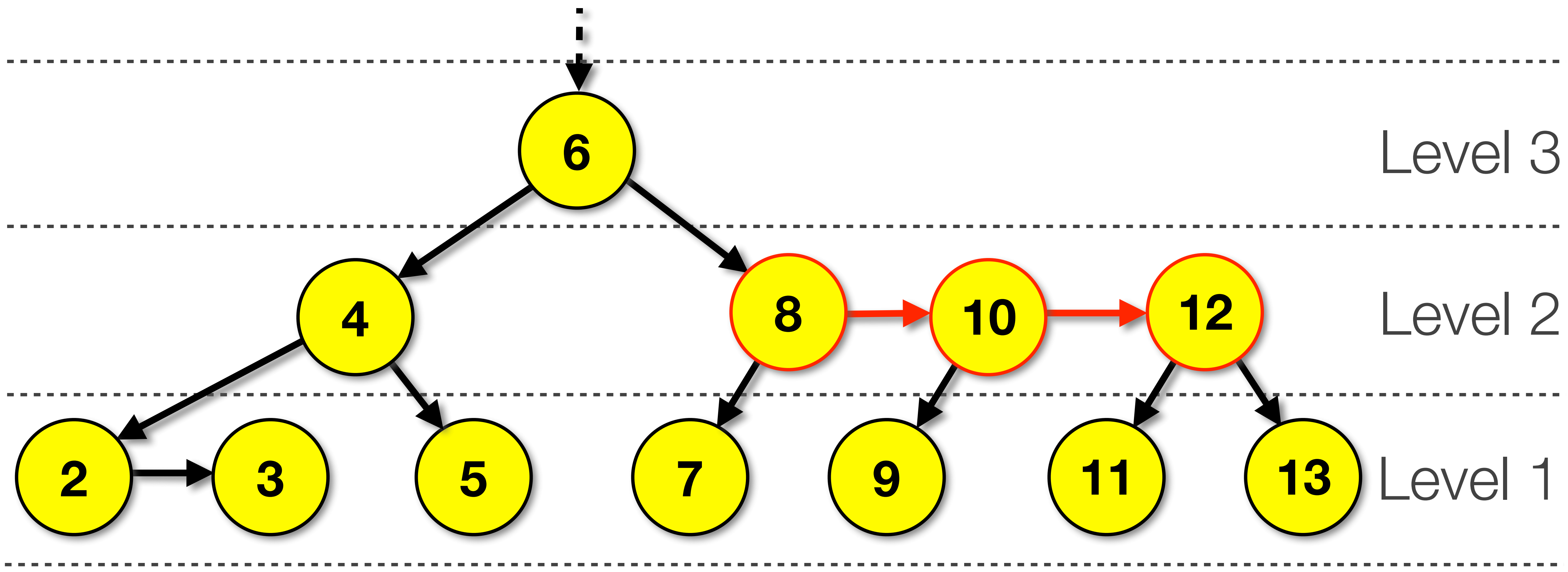
Example of Deletion

After skew 4.right.right (node 10)
Next split node 4



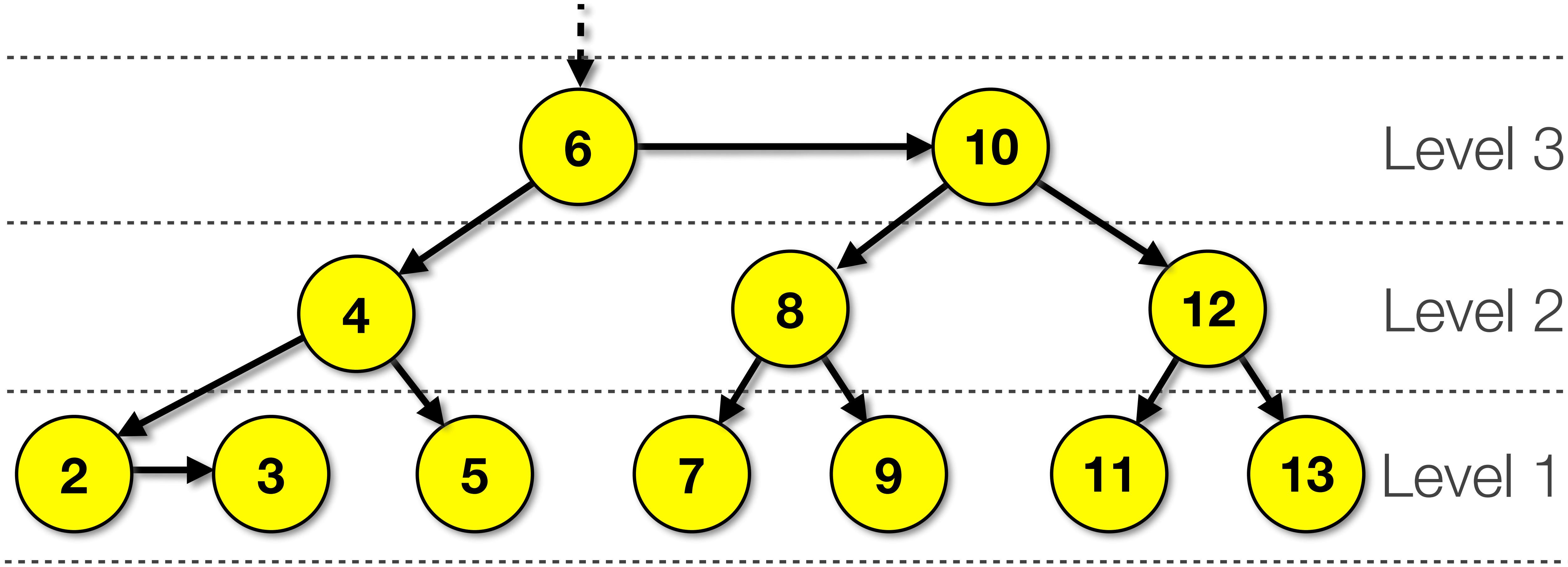
Example of Deletion

After split node 4 (new subtree root)
Next split node 6.right (node 8)



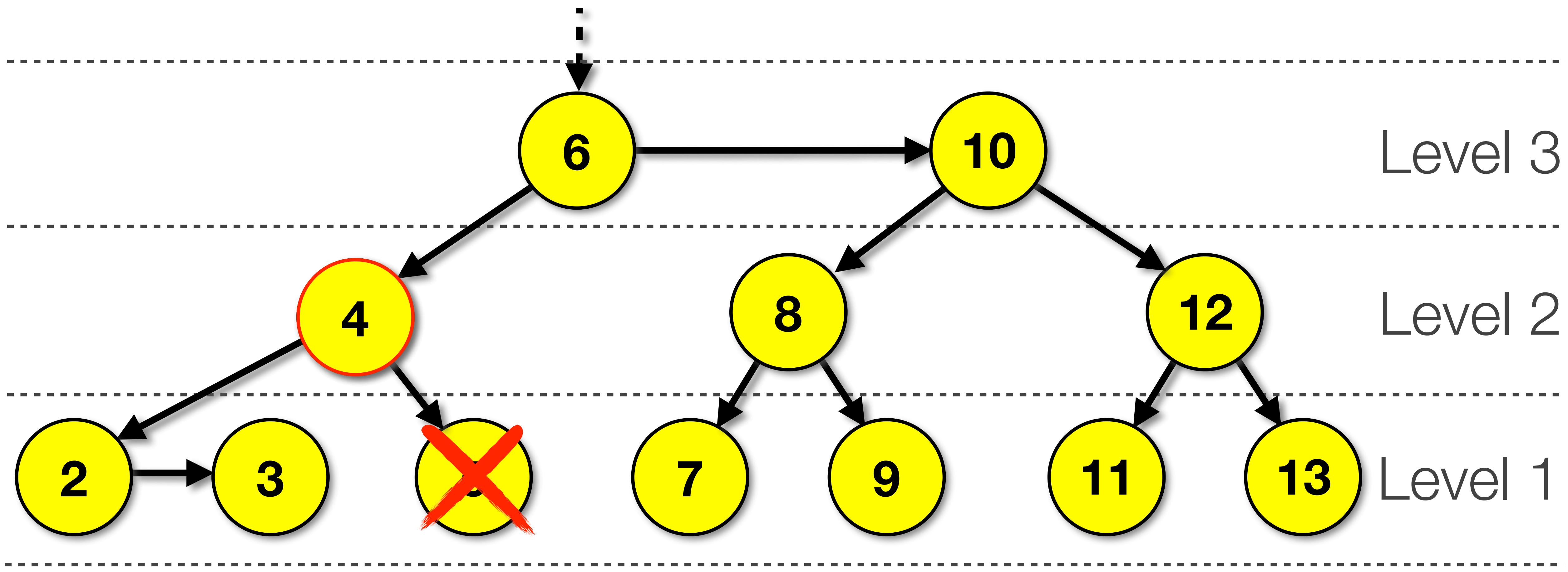
Example of Deletion

After split node 6
Tree is balanced



Example of Deletion

Delete node 5
Node 4 is now violates rule #5

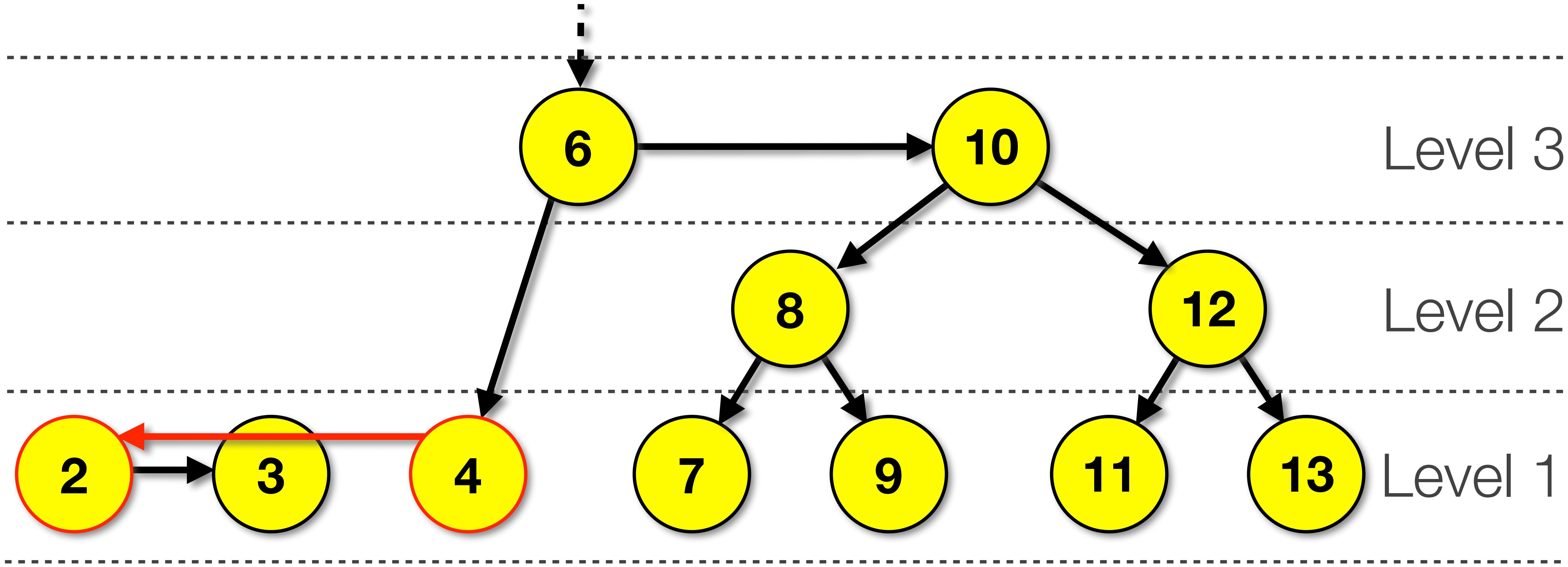


Example of Deletion

Decrement the level of node 4

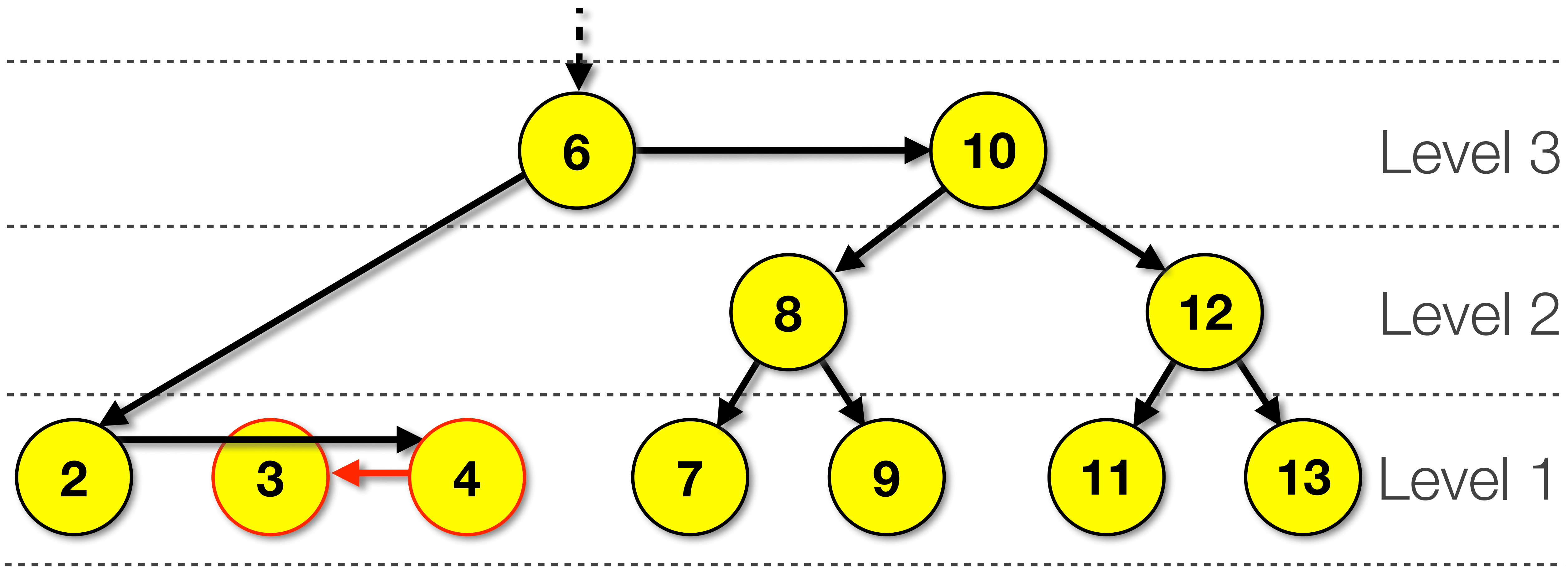
Node 4 now has left horizontal link

Next skew node 4



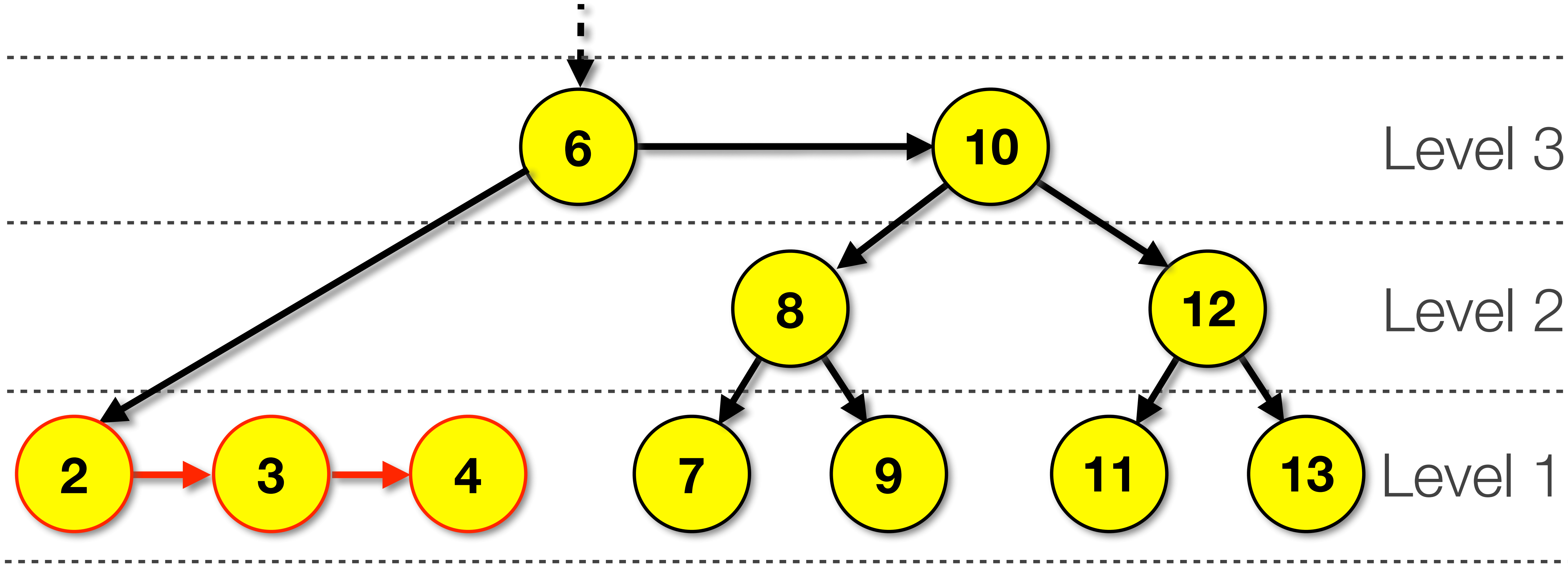
Example of Deletion

After skew node 4 (new subtree root)
Next skew node 2.right (node 4 again)



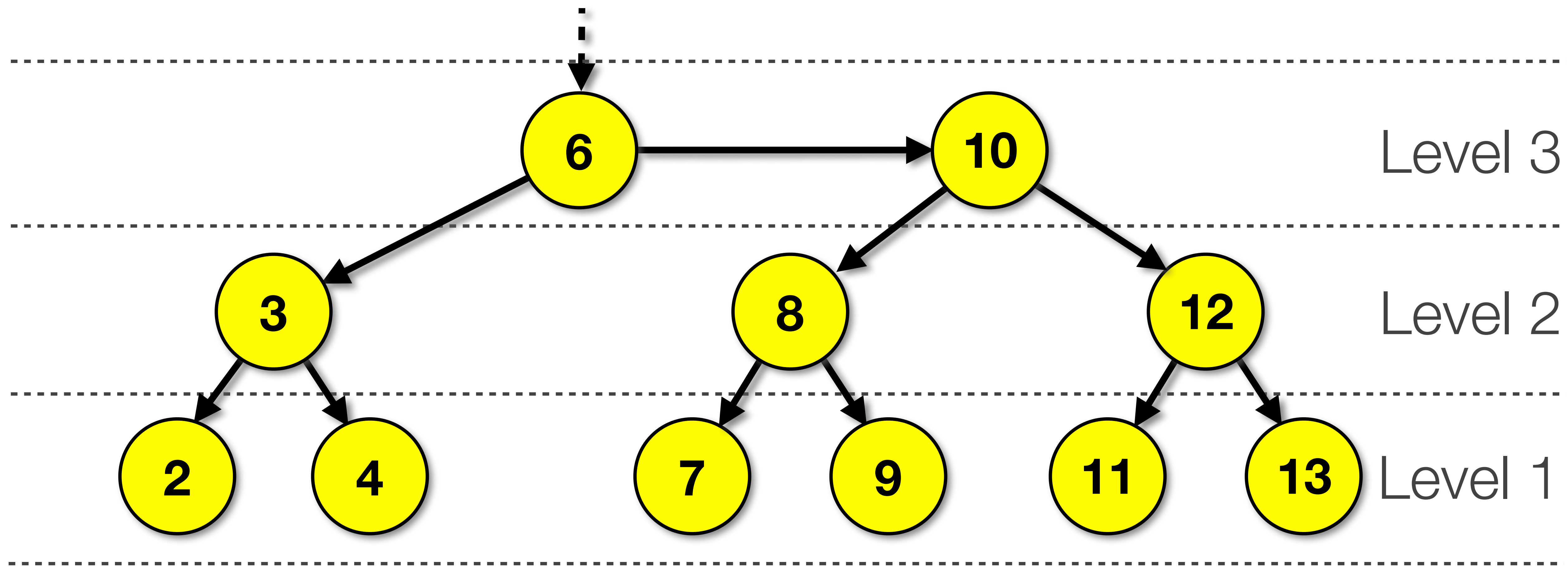
Example of Deletion

After skew node 2.right
Skew node 2.right.right (does nothing)
Split node 2



Example of Deletion

After split node 2 (new subtree root)
Split node 3.right (does nothing)
Tree is balanced



Additional Slides

Implementation of Child Rotations

```java
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 */
static BinaryNode<AnyType> rotateWithLeftChild( BinaryNode<AnyType> k2 )
{
    BinaryNode<AnyType> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}
```

```java
/**
 * Rotate binary tree node with right child.
 * For AVL trees, this is a single rotation for case 4.
 */
static BinaryNode<AnyType> rotateWithRightChild( BinaryNode<AnyType> k1 )
{
    BinaryNode<AnyType> k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}
```