CMSC 420 – 0201 – Fall 2019
Lecture 06

2-3, Red-black, and AA trees

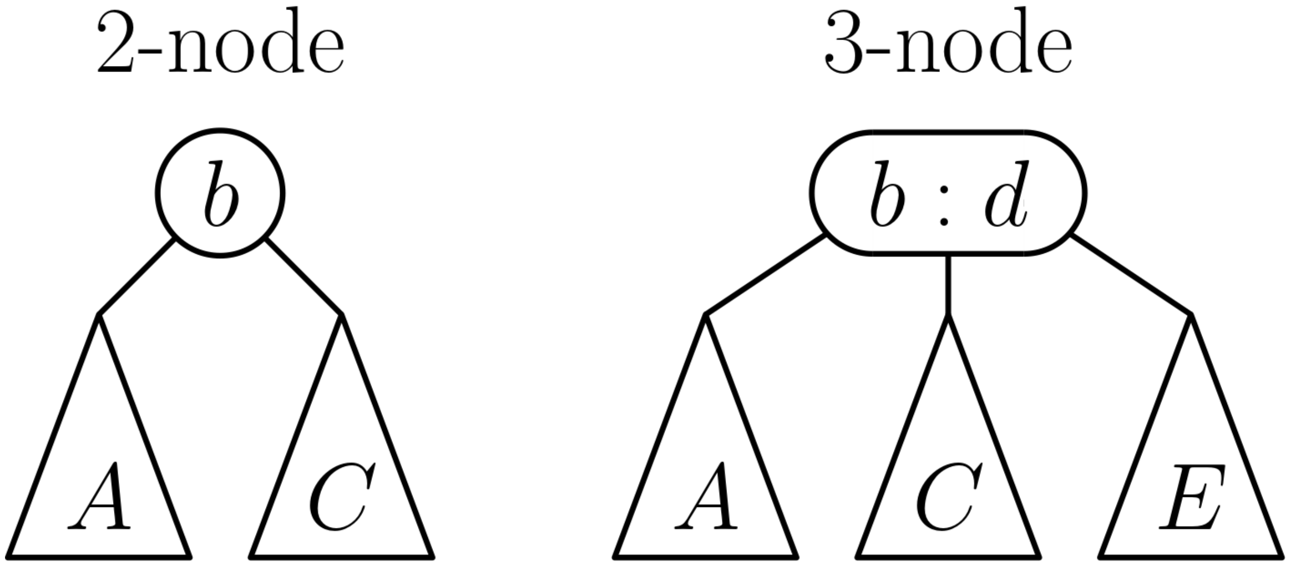
“A rose by any other name…”

- Today, we will consider three search trees, which outwardly look different, but all are equivalent (or nearly so)
- All support find, insert, and delete in $O(\log n)$ time for a tree with n nodes
- These are:
  - 2-3 Trees
  - Red-black Trees
  - AA Trees

2-3 Tree

A Variable Width Tree

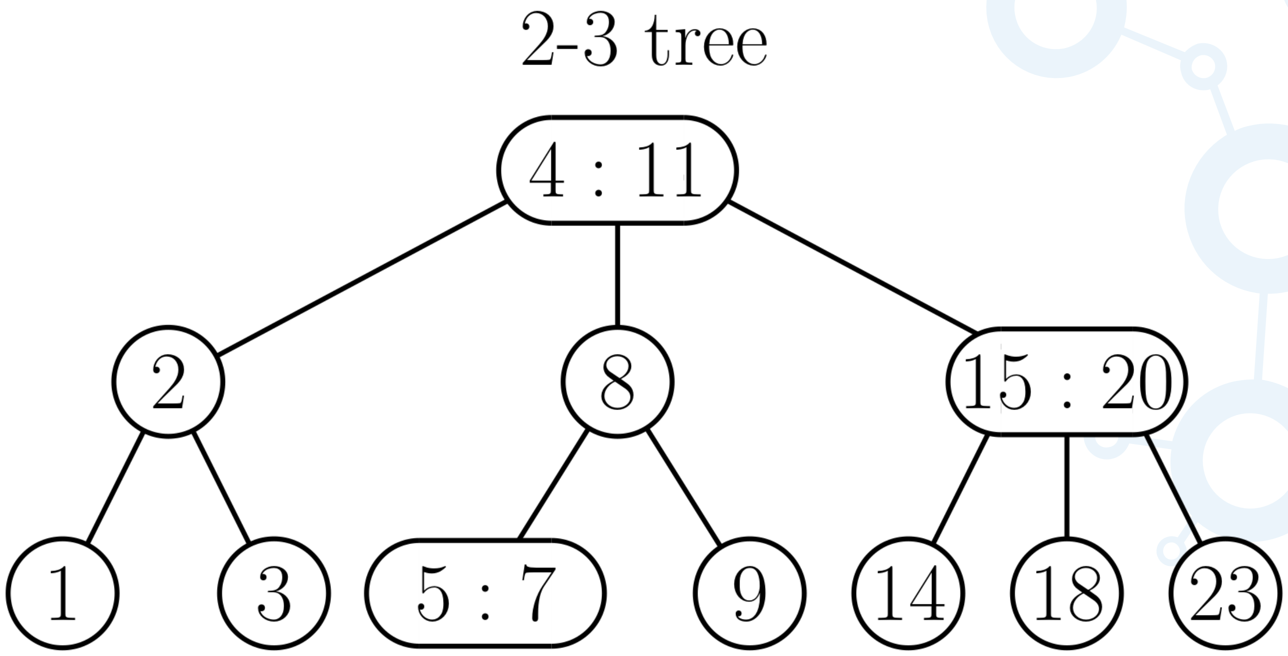
- 2-Node:
  - Two children; stores one key; order: $A < b < C$
- 3-Node:
  - Three children; stores two keys; order: $A < b < C < d < E$



2-3 Tree

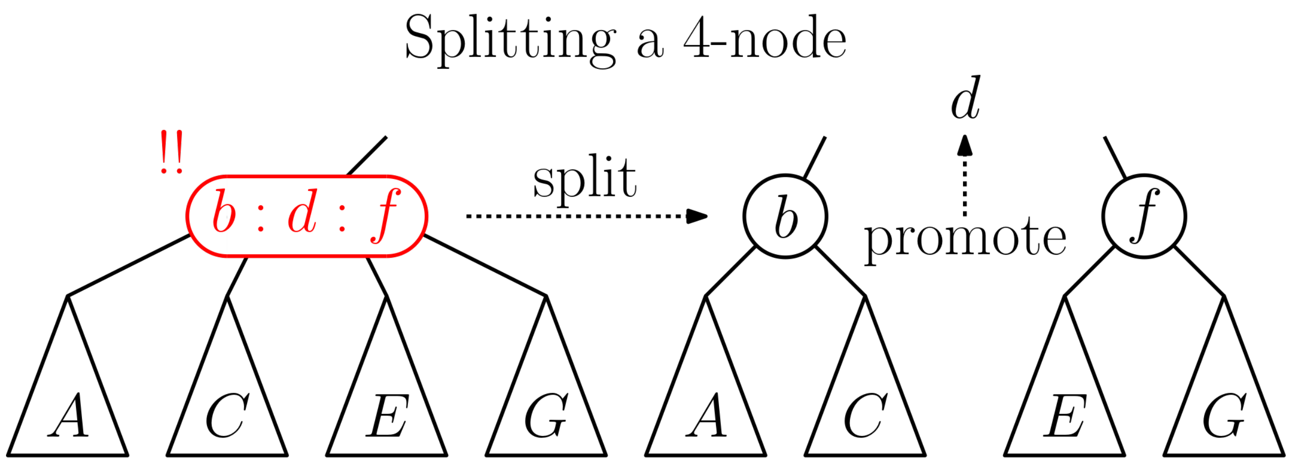
Formal Definition

- A 2-3 tree is:
  - An empty tree (i.e., `null`)
  - Root is a 2-node and two subtrees are 2-3 trees of equal height
  - Root is a 3-node and its three subtrees are 2-3 trees of equal height
- Theorem: A 2-3 tree with n nodes has height $O(\log n)$
- Proof: (Easy) The sparsest tree is already a complete binary tree



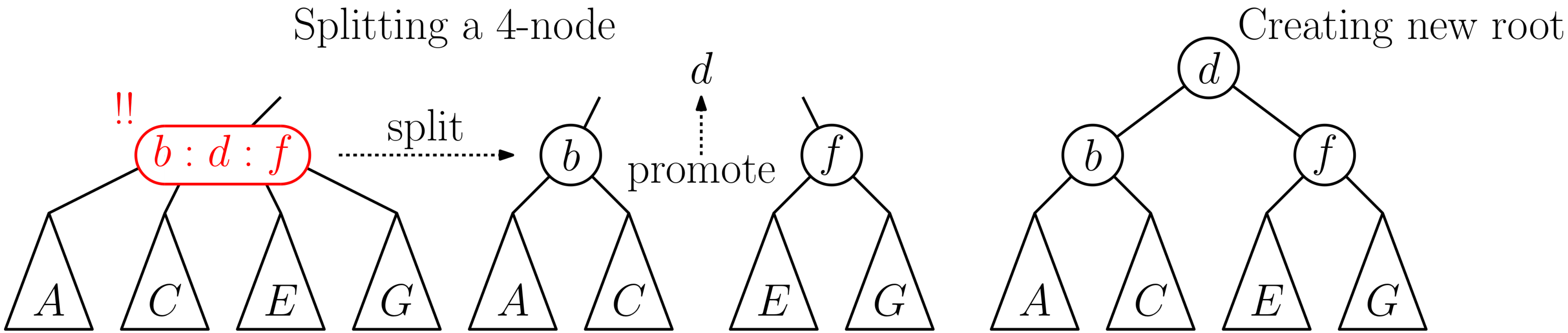
2-3 Tree Insertion

- Start as usual: Find the key and note the leaf node where we fall out of the tree
- Insert new key in this leaf, and restructure if needed:
  - 2-node → 3-node - No problem
  - 3-node → 4-node - !!
    - Split into two 2-nodes; promote middle key to parent; $4 = 2 + 2$



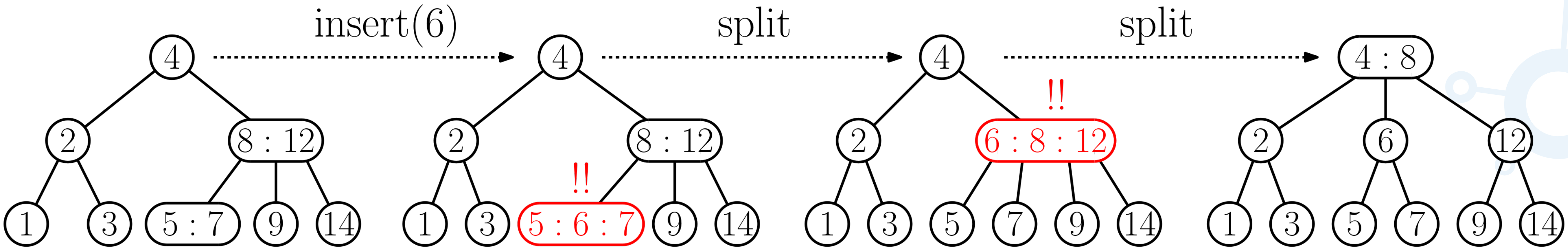
2-3 Tree Insertion

- Start as usual: Find the key and note the leaf node where we fall out of the tree
- Insert new key in this leaf, and restructure if needed:
  - 2-node → 3-node - No problem
  - 3-node → 4-node - !!
    - Split into two 2-nodes; promote middle key to parent; $4 = 2 + 2$
    - May need to fix parent or create a new root



2-3 Tree Insertion

- Example:

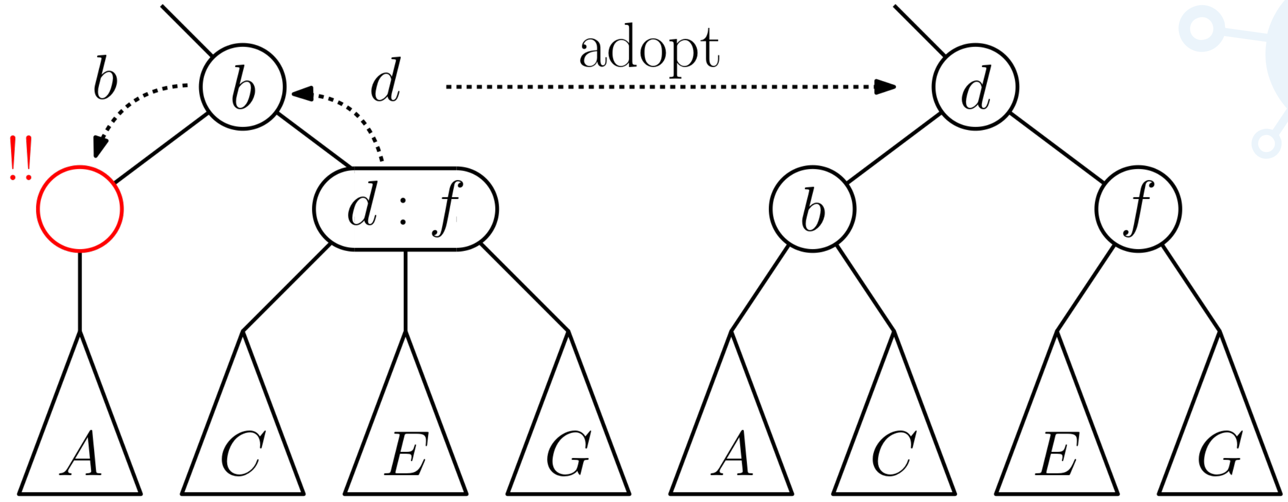


2-3 Tree Deletion

- Deletion as usual:
  - Find the key
  - If it is not a leaf, find the replacement node (inorder successor)
  - Copy replacement-node contents to deleted node
  - Recursively delete the replacement node
  - (We may assume that restructuring always starts at the leaf level)

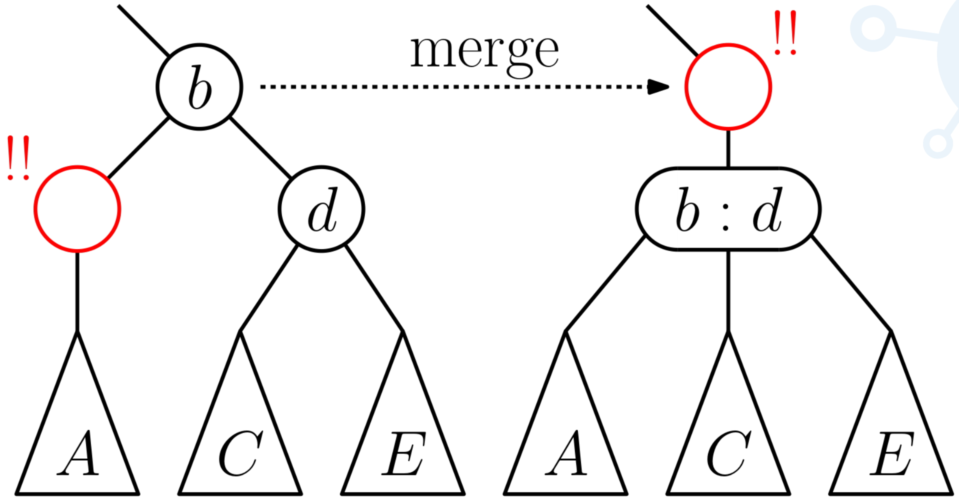
2-3 Tree Deletion

- Restructuring:
  - 3-node → 2-node: No problem
  - 2-node → 1-node: Two possible fixes:
    - Adopt from sibling
    - Merge with sibling
- Adoption:
  - If there is a 3-node sibling
  - Adopt its closest subtree
  - …and associated key
  - $1 + 3 = 2 + 2$



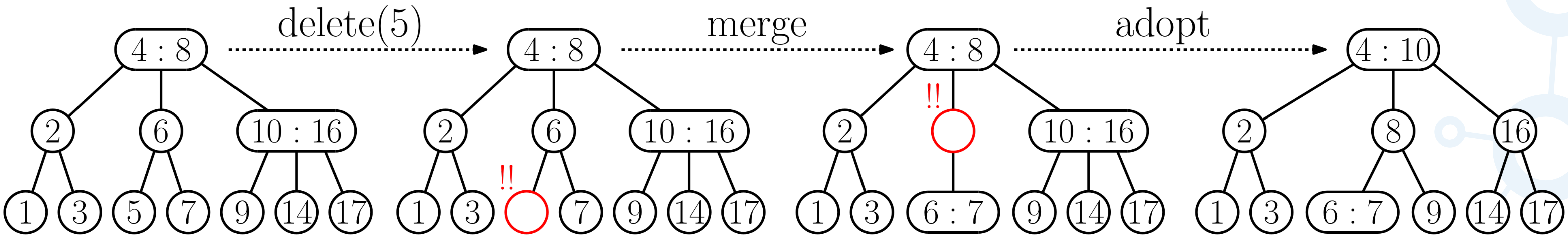
2-3 Tree Deletion

- Restructuring:
  - 3-node → 2-node: No problem
  - 2-node → 1-node: Two possible fixes:
    - Adopt from sibling
    - Merge with sibling
- Merging:
  - No sibling is 3-node ⇒ 2-node
  - Merge these nodes: $1 + 2 = 3$
  - Demote key from parent
  - May need to fix parent or delete root



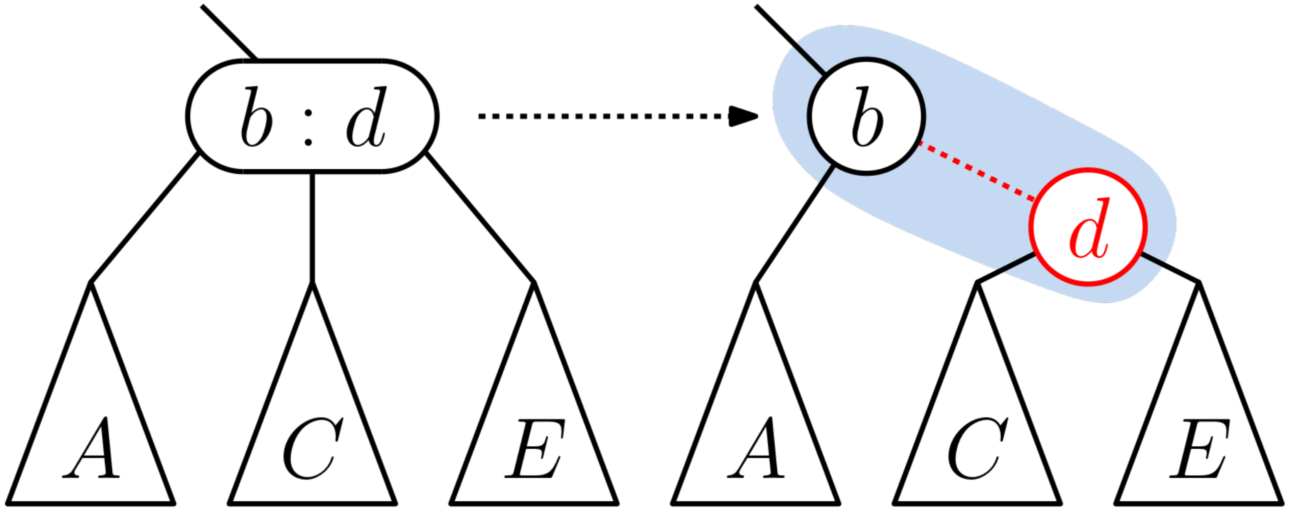
2-3 Tree Deletion

- Example:



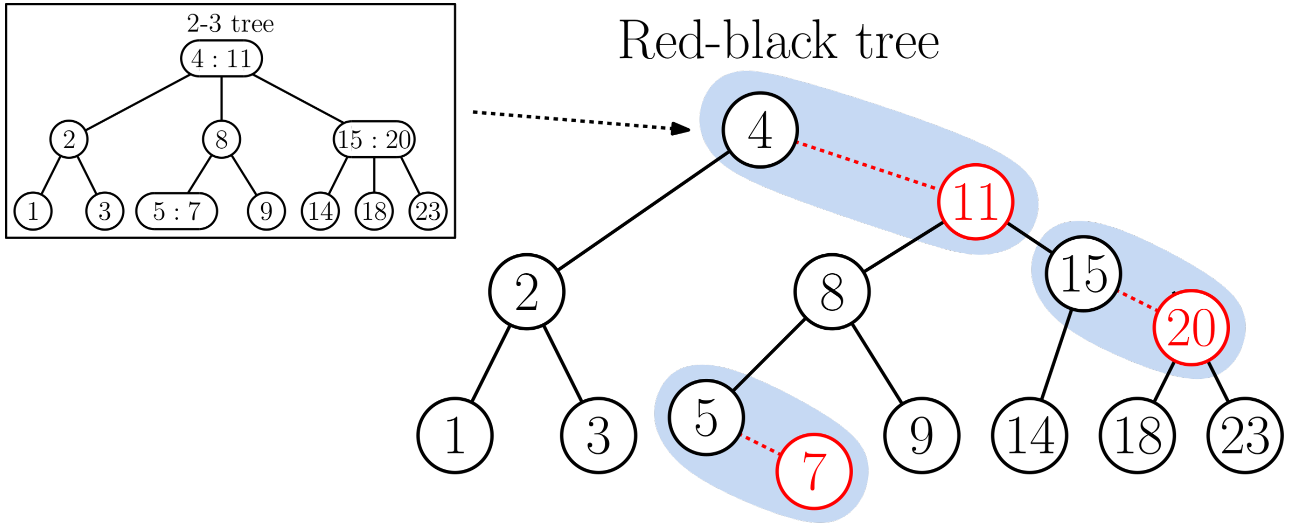
Red-Black Trees

- 2-3 trees are not binary trees – Can we simulate the same idea as a binary tree?
- Replace each 3-node with a pair of nodes:
  - To distinguish them, we'll color the upper node black and the lower node red
  - The result is called a Red-Black Tree



Red-Black Trees

- 2-3 trees are not binary trees – Can we simulate the same idea as a binary tree?
- Replace each 3-node with a pair of nodes:
  - To distinguish them, we'll color the upper node black and the lower node red
  - The result is called a Red-Black Tree



Red-Black Trees

Definition:

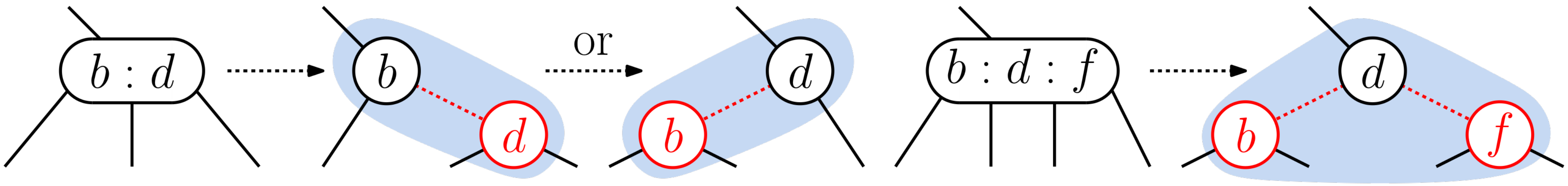
- Each node is either red or black
- The root is black
  - This corresponds to the fact that the root is either a 2-node or the first half of a 3-node
- All `null` pointers are considered black
  - This is just a convenient convention
- If a node is red, then both its children are black
  - This enforces the condition that a child of the second half of a 3-node [red] must either be a 2-node [black] or the first half of a 3-node [black]
- Every path from a given node to any of its `null` descendants contains the same number of black nodes
  - This corresponds to the requirement that all leaves of the 2-3 tree are of equal depth

Red-Black Trees

Lemma: Every 2-3 tree corresponds to a red-black tree

- But the converse does not hold. There are valid red-black trees that are not the encoding of some 2-3 tree
- (a) The red child could be on either the left or right side
- (b) Both children of a black node may be red

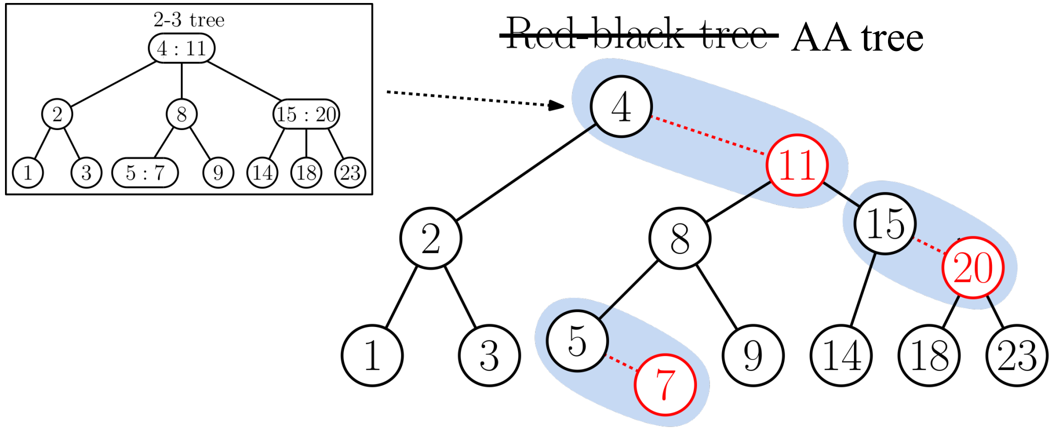
In fact, red-black trees are a binary encoding of a more general tree, a 2-3-4 tree



(a) (b)

AA Trees

- A simpler variant of the red-black tree
- Invented by Arne Anderson (1993) to simplify coding of red-black trees
- Updated definition:
  - If a node is red, ~~then both its children are black~~ it is the right child of a black node
- This fits exactly with our encoding of 2-3 trees as binary trees



AA Trees

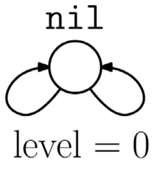
Node Representation:

- No `null` pointers: Use a sentinel node, called `nil`.

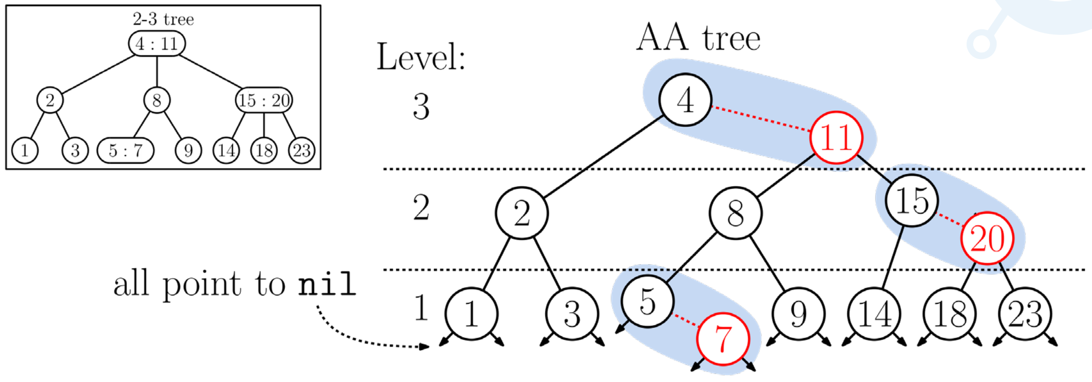
  `nil.left = nil.right = nil`

  Reduces need for checking null pointers.

- No node colors: Every node stores a level number:
  - `nil` is at level 0
  - Leaves at level 1
  - If you are a red node, you are at the same level as your parent
  - If you are a black node, you are at one level less than your parent
  - Levels match levels of 2-3 tree



(a)

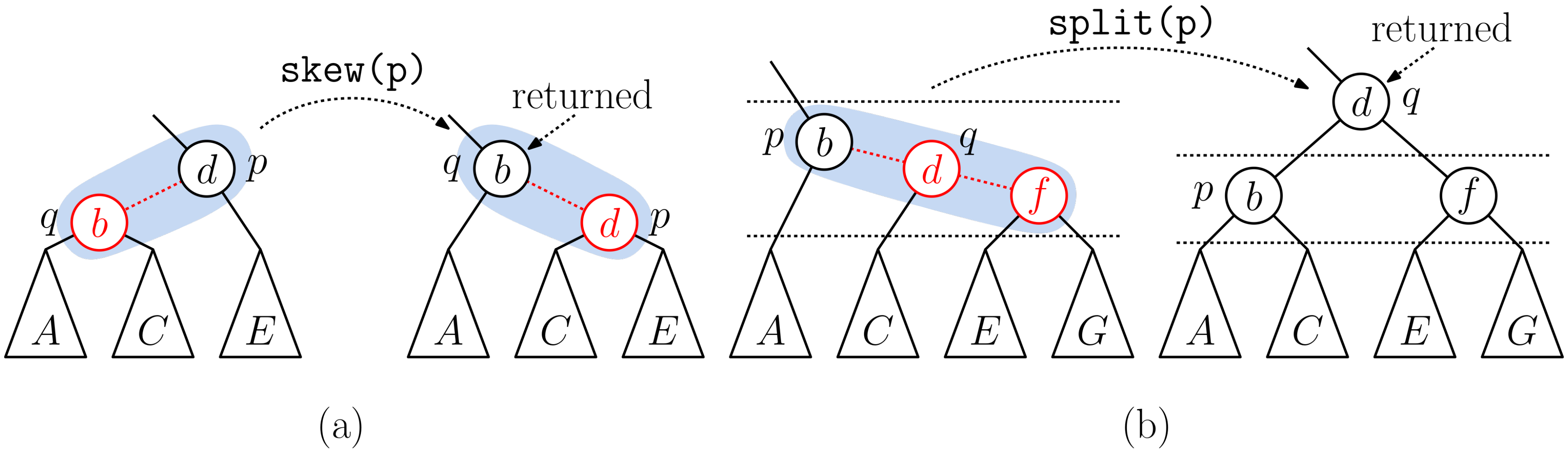


(b)

AA-Trees

Restructuring

- `skew(p)`: If p is black and has a red left child, rotate so that the red child is now on the right
- `split(p)`: If p is black and has a right chain of two consecutive red nodes, split this triple, promoting p's right child to the next higher level



AA Trees

Restructuring Operations

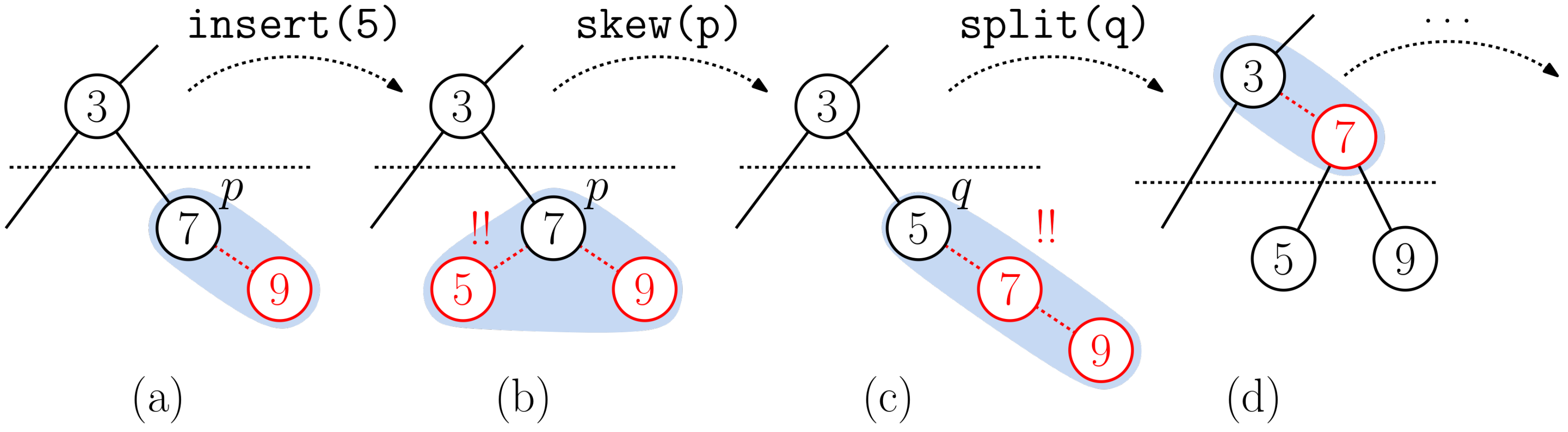
```
AANode skew(AANode p) {
    if (p.left.level == p.level) {  // red node to our left?
        AANode q = p.left;          // do a right rotation at p
        p.left = q.right;
        q.right = p;
        return q;                   // return pointer to new upper node
    }
    else return p;                  // else, no change needed
}

AANode split(AANode p) {
    if (p.right.right.level == p.level) { // right-right red chain?
        AANode q = p.right;         // do a left rotation at p
        p.right = q.left;
        q.left = p;
        q.level += 1;               // promote q to next higher level
        return q;                   // return pointer to new upper node
    }
    else return p;                  // else, no change needed
}
```

AA Trees - Insertion

Insertion

- Search for the new key and note where we fall out of the tree
- Insert a new (red) leaf node here (at level 1)
- Work back towards the root and restructure along the way
  - Left child is red? → skew
  - Two red children to the right? → split



AA Trees

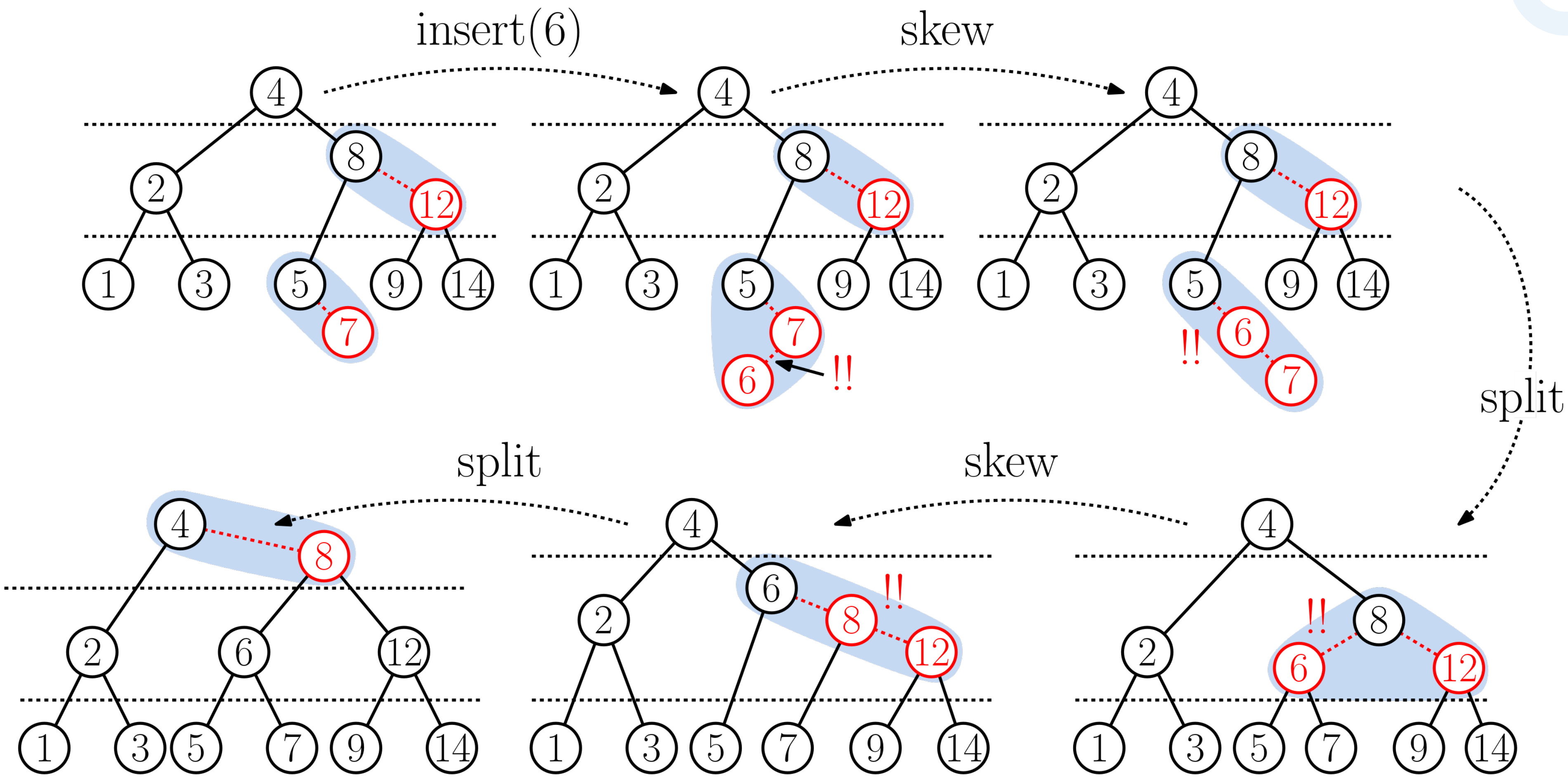
Insertion

```
AANode insert(Key x, Value v, AANode p) {
  if (p == nil)                          // fell out of the tree?
       p = new AANode(x, v, 1, nil, nil); // ... create a new leaf node here
   else if (x < p.key)                   // x is smaller?
       p.left = insert(x, v, p.left);    // ...insert left
   else if (x > p.key)                   // x is larger?
       p.right = insert(x, v, p.right);  // ...insert right
   else
       throw DuplicateKeyException;      // duplicate key!
   return split(skew(p));                // restructure and return result
}
```

Only difference with standard binary search tree insertion

AA-Trees

Insertion Example



AA-Trees - Deletion

- Find the node to delete
- If it is not a leaf, find replacement at the leaf level and delete replacement
- Work back towards the root and restructure along the way
  - More cases than with insertion
  - Basic issue is that a node's level may decrease
- Possibly 3 skew invocations:
  - skew(p), skew(p.right), skew(p.right.right)
- Possibly 2 split invocations:
  - split(p), split(p.right)

AA Trees

Deletion – Restructuring Utilities

```
AANode updateLevel(AANode p) {                         // update p's level
    int idealLevel = 1 + min(p.left.level,  p.right.level);
    if (p.level > idealLevel) {                        // p's level is too high?
        p.level = idealLevel;                          // decrease its level
        if (p.right.level > idealLevel)                // p's right child red?
            p.right.level = idealLevel;                // ...fix its level as well
    }
    return p;
}
```

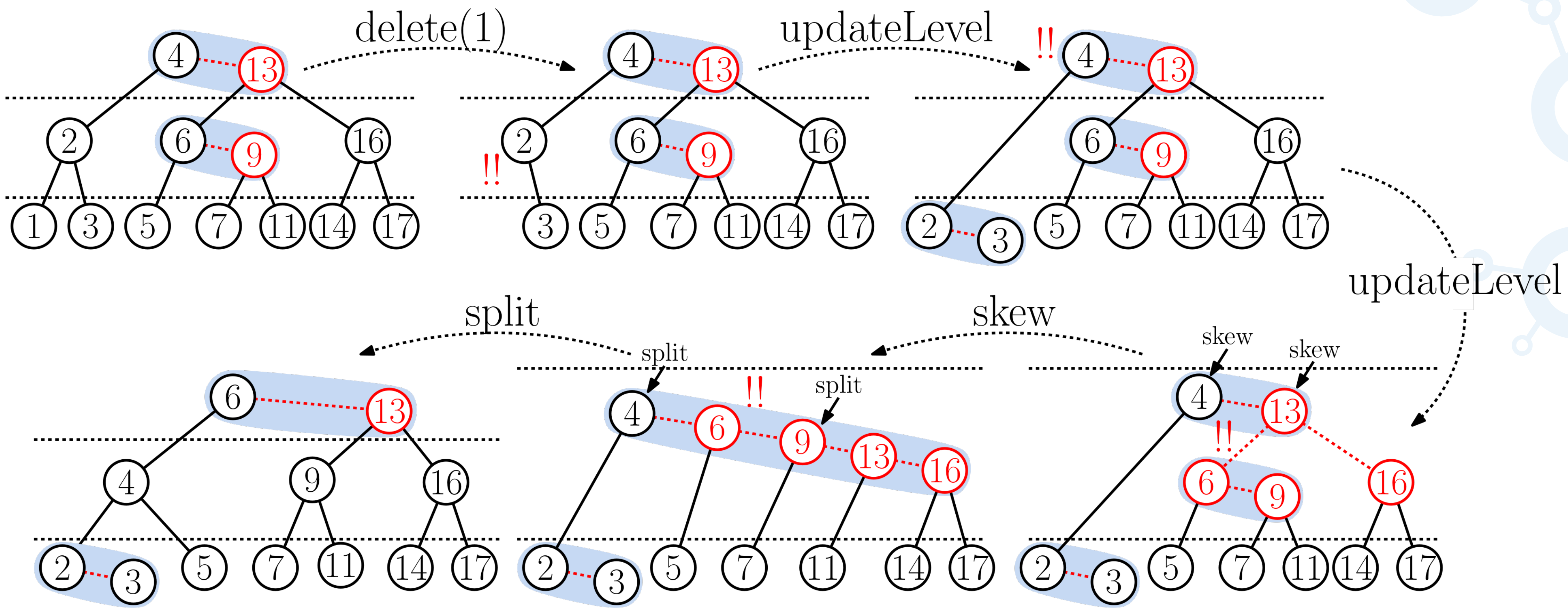
```
AANode fixupAfterDelete(AANode p) {
    p = updateLevel(p);                                // update p's level
    p = skew(p);                                       // skew p
    p.right = skew(p.right);                           // ...and p's right child
    p.right.right = skew(p.right.right);               // ...and p's right-right grandchild
    p = split(p);                                      // split p
    p.right = split(p.right);                          // ...and p's (new) right child
    return p;
}
```

AA Trees - Deletion

```
AANode delete(Key x, AANode p) {
  if (p == nil)                             // fell out of tree?
       throw KeyNotFoundException;          // ...error - no such key
   else {
       if (x < p.key)                       // look in left subtree
           p.left = delete(x, p.left);
       else if (x > p.key)                  // look in right subtree
           p.right = delete(x, p.right);
       else {                               // found it!
           if (p.left == nil && p.right == nil)// leaf node?
               return nil;                  // just unlink the node
           else if (p.left == nil) {        // no left child?
               AANode r = inorderSuccessor(p); // get replacement from right
               p.copyContentsFrom(r);       // copy replacement contents here
               p.right = delete(r.key, p.right);// delete replacement
           }
           else {                           // no right child?
               AANode r = inorderPredecessor(p);// get replacement from left
               p.copyContentsFrom(r);       // copy replacement contents here
               p.left = delete(r.key, p.left); // delete replacement
           }
       }
       return fixupAfterDelete(p);          // fix structure after deletion
   }
}s
```

AA-Trees

Deletion Example



Summary

- 2-3 Trees
- Insertion
  - Splitting nodes
- Deletion
  - Adoption
  - Merging
- Red-black trees – Model 2-3-4 trees
- AA trees – Simplified red-black trees
  - Skew and split to restructure