SELF-BALANCING SEARCH TREES

Chapter 9

Chapter Objectives

- To understand the impact that balance has on the performance of binary search trees
- To learn about the AVL tree for storing and maintaining a binary search tree in balance
- To learn about Red-Black trees for storing and maintaining a binary search tree in balance
- To learn about 2-3 trees, 2-3-4 trees, and B-trees and how they achieve balance
- To learn about skip-lists and their properties similar to balanced search trees properties
- To understand the process of search and insertion in each of these structures and to be introduced to removal

Self-Balancing Search Trees

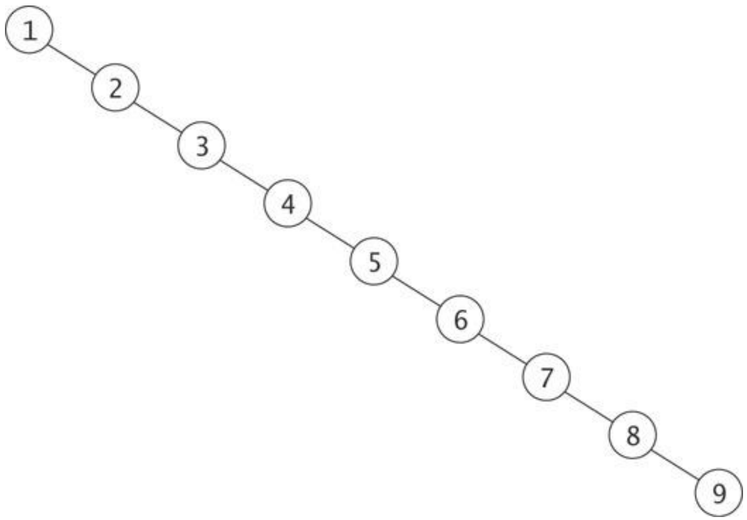
- The performance of a binary search tree is proportional to the *height of the tree*
- A full binary tree of height k can hold $2^k - 1$ items
- If a binary search tree is full and contains n items, the expected performance is $O(\log n)$
- However, if a binary tree is not full, the actual performance is worse than expected
- To solve this problem, we introduce *self-balancing* trees to achieve a balance so that the heights of the right and left subtrees are equal or nearly equal
- We also look at other non-binary search trees and the skip-list

Tree Balance and Rotation

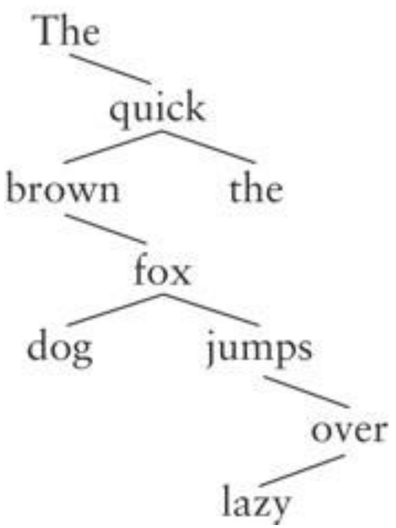
Section 9.1

Why Balance is Important

- Searches into this unbalanced search tree are O(*n*), not O(log *n*)

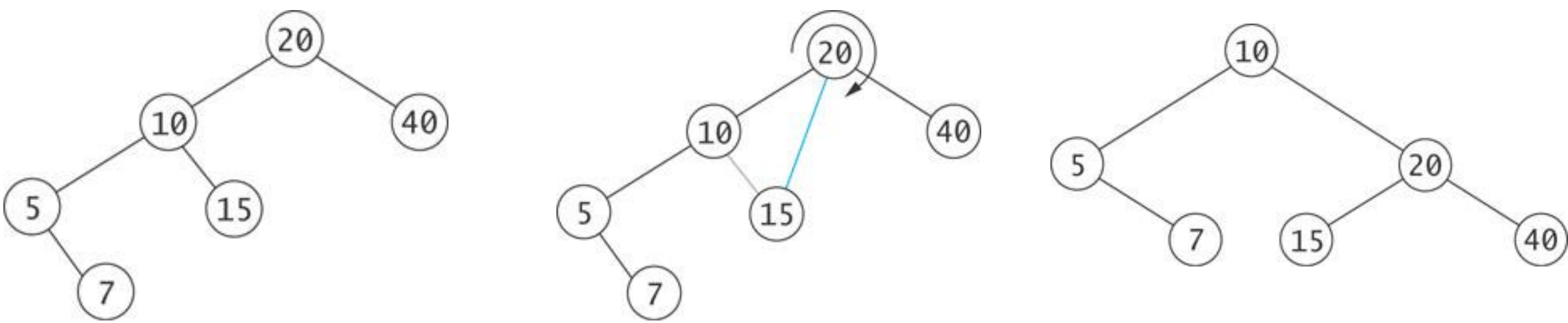


- A realistic example of an unbalanced tree

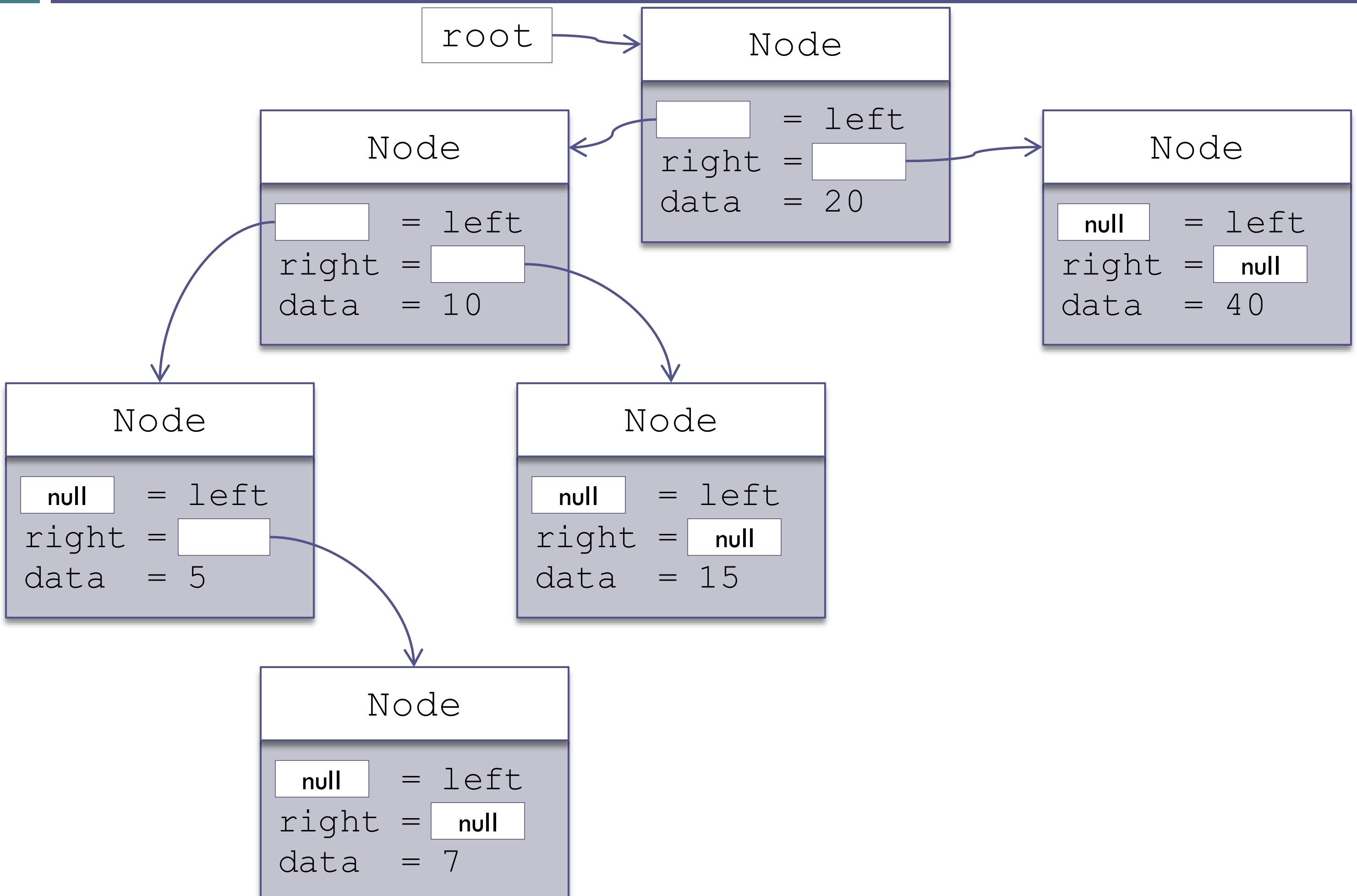


Rotation

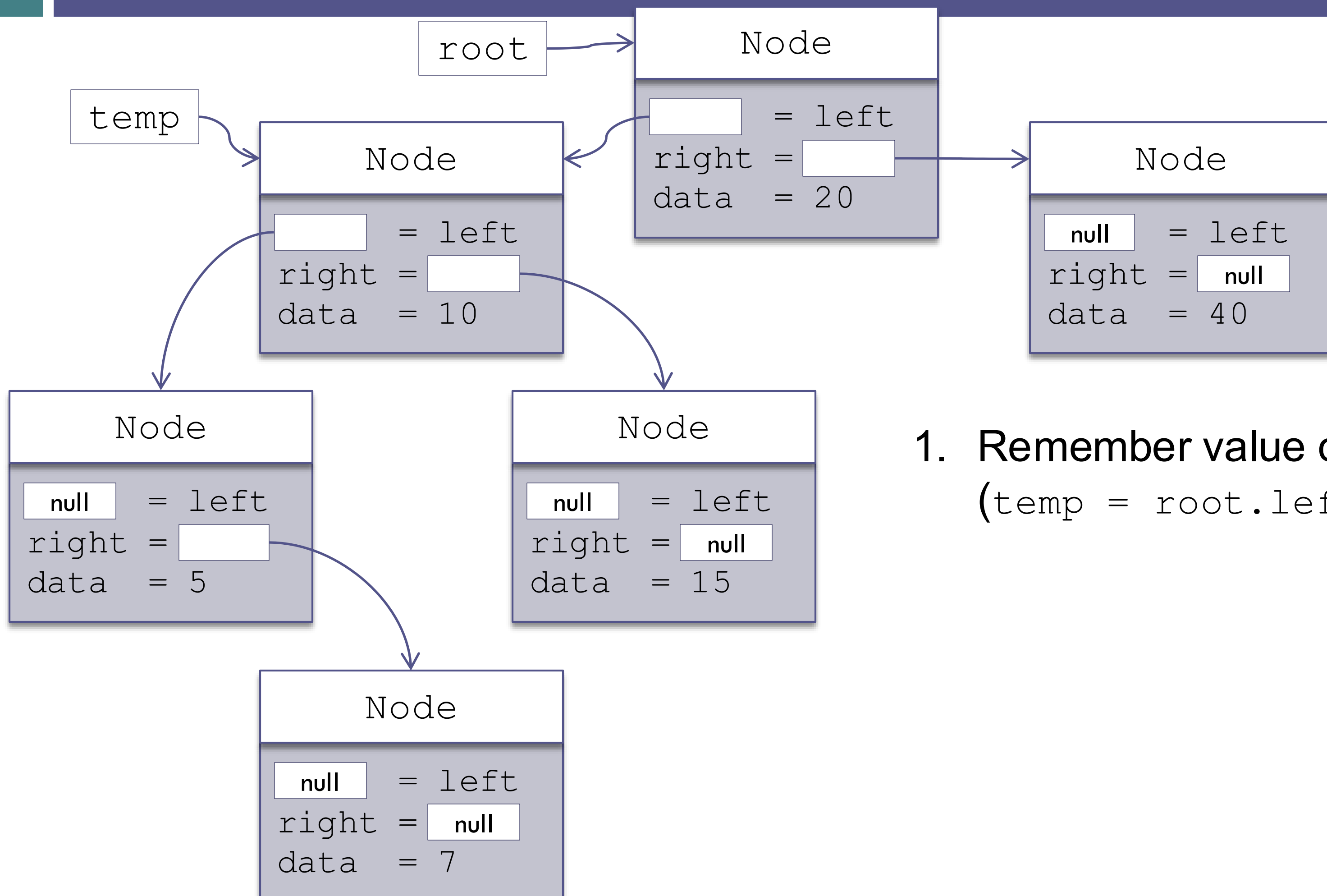
- We need an operation on a binary tree that changes the relative heights of left and right subtrees, but preserves the binary search tree property



Algorithm for Rotation

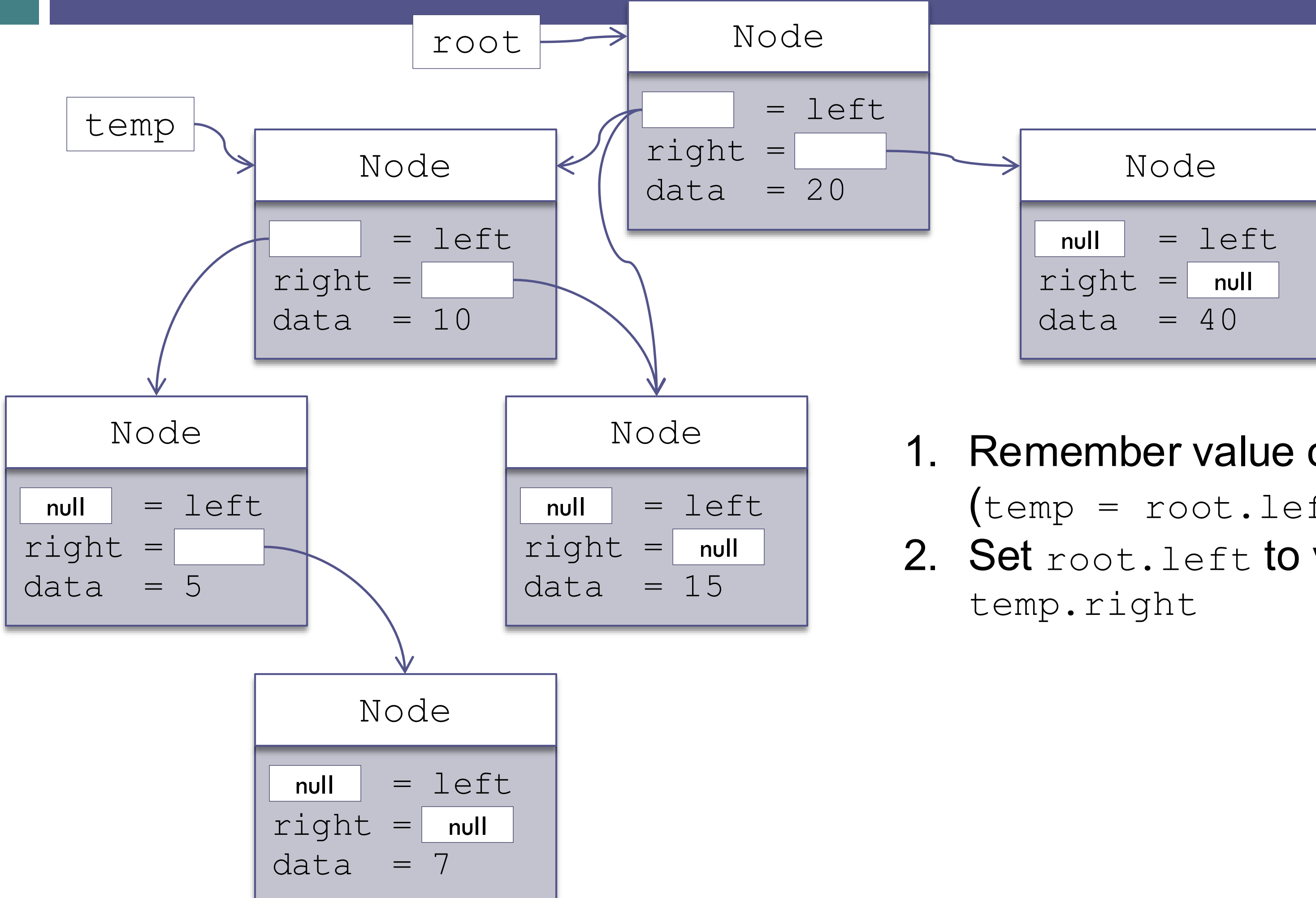


Algorithm for Rotation (cont.)



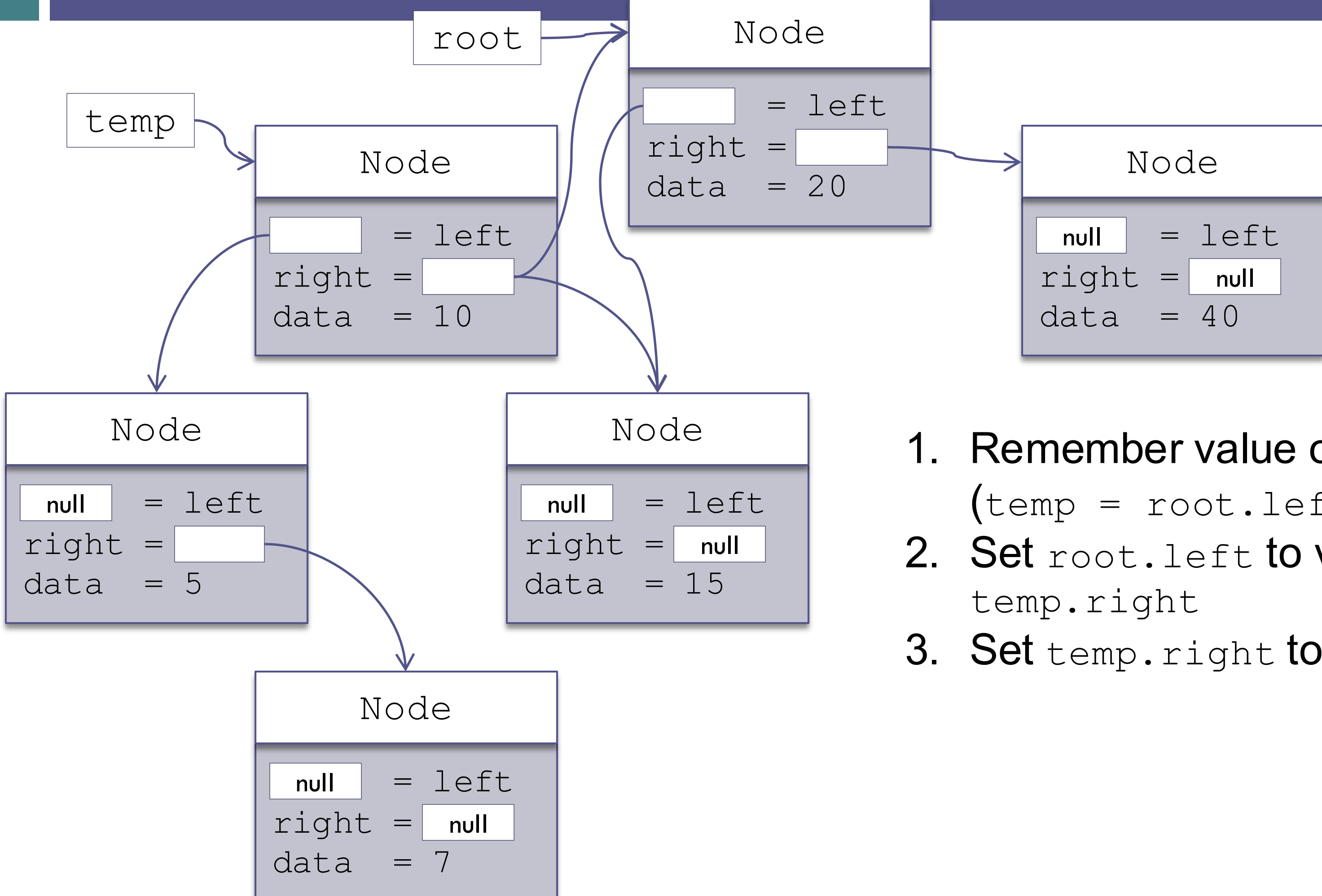
1. Remember value of `root.left` (`temp = root.left`)

Algorithm for Rotation (cont.)



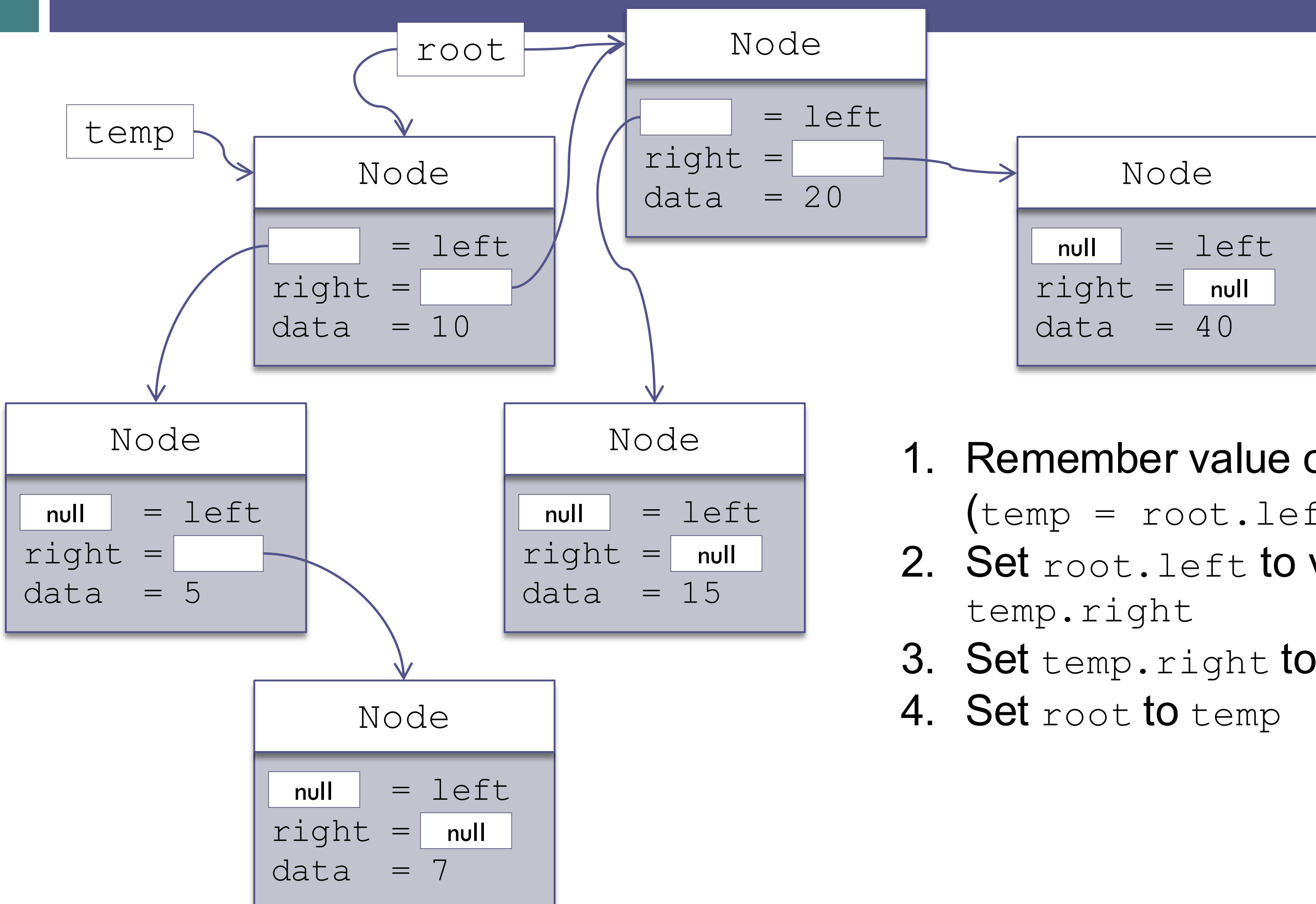
1. Remember value of `root.left` (`temp = root.left`)
2. Set `root.left` to value of `temp.right`

Algorithm for Rotation (cont.)



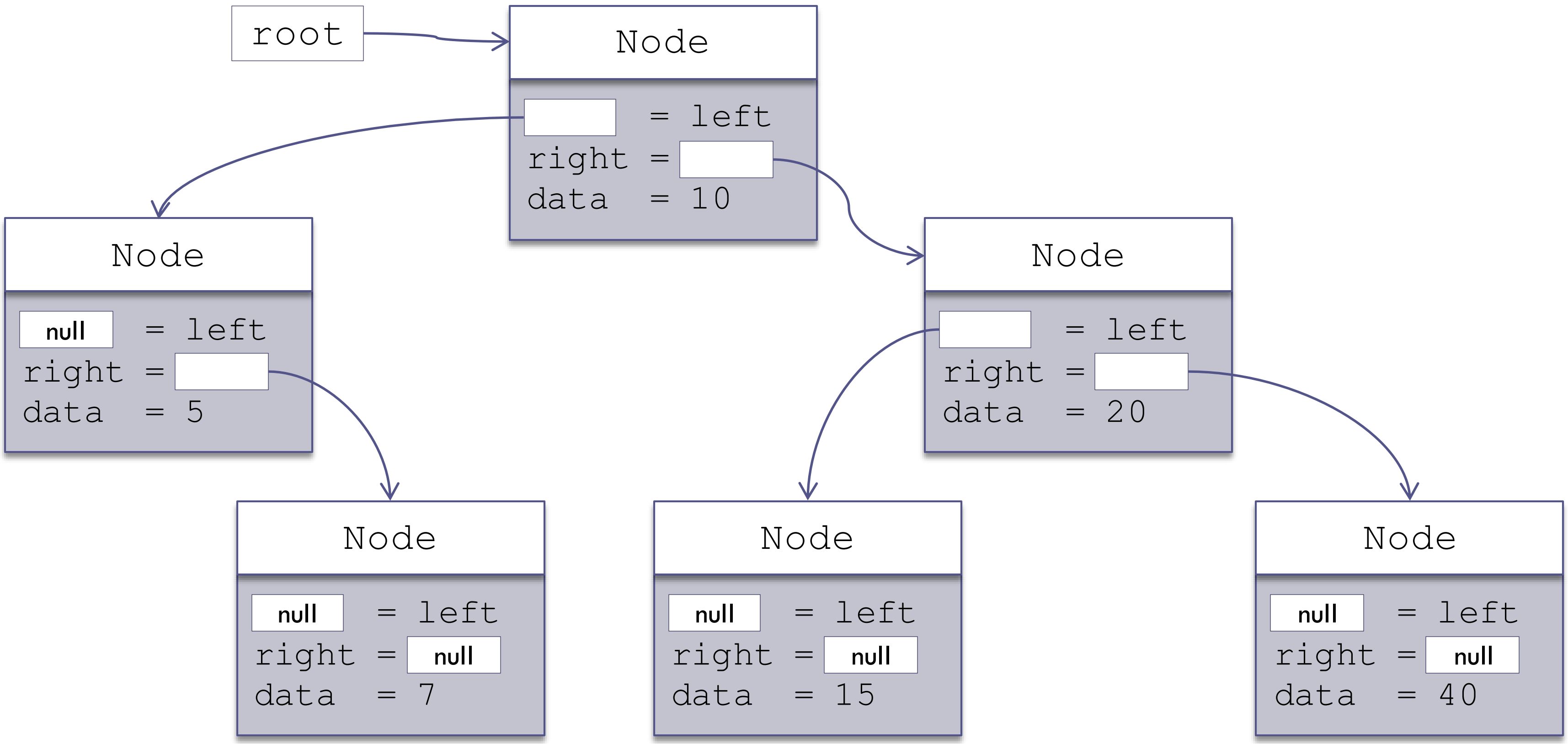
1. Remember value of `root.left` (`temp = root.left`)
2. Set `root.left` to value of `temp.right`
3. Set `temp.right` to `root`

Algorithm for Rotation (cont.)

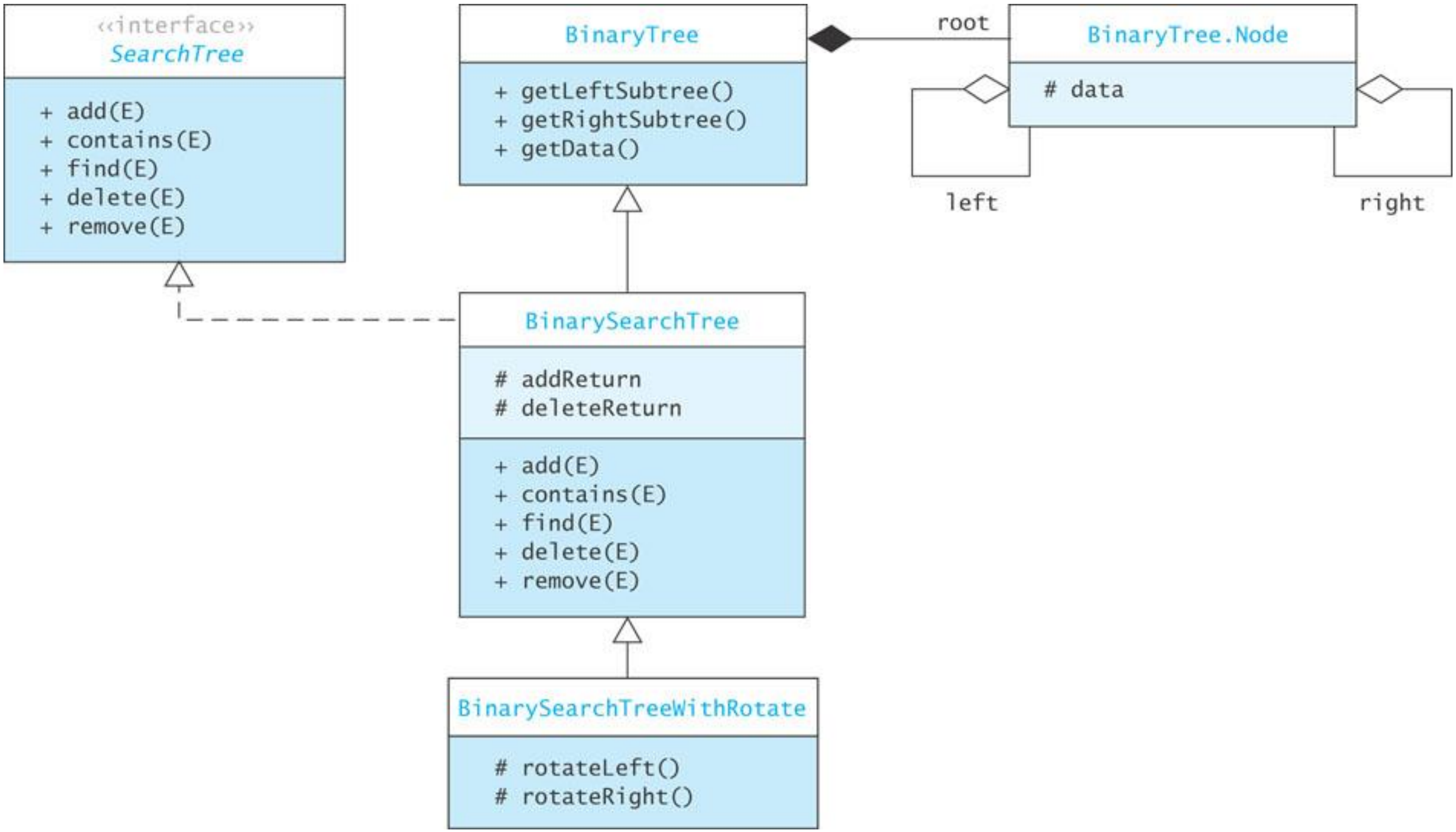


1. Remember value of `root.left` (`temp = root.left`)
2. Set `root.left` to value of `temp.right`
3. Set `temp.right` to `root`
4. Set `root` to `temp`

Algorithm for Rotation (cont.)



Implementing Rotation



Implementing Rotation (cont.)

- Listing 9.1 (`BinarySearchTreeWithRotate.java`, page 476)

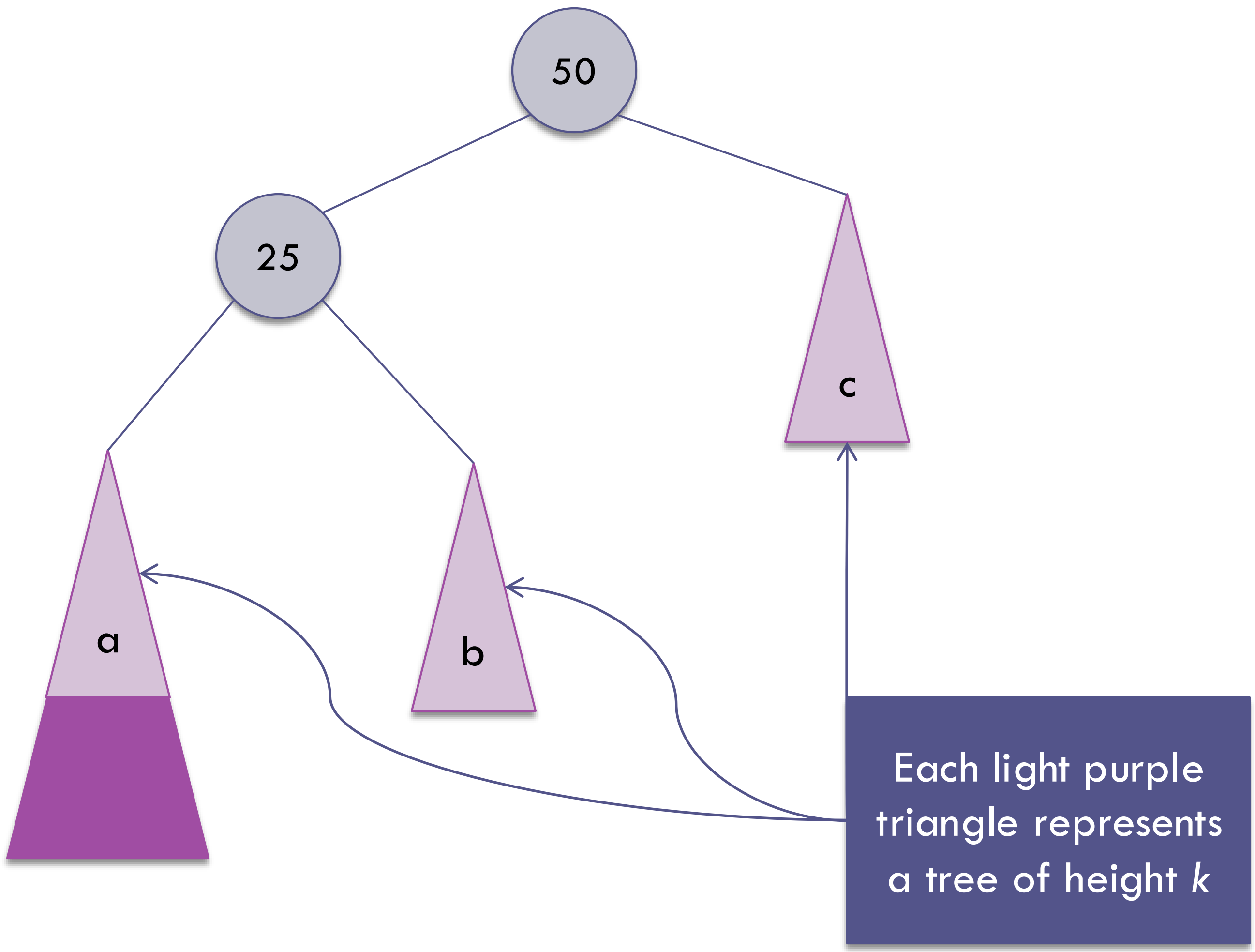
AVL Trees

Section 9.2

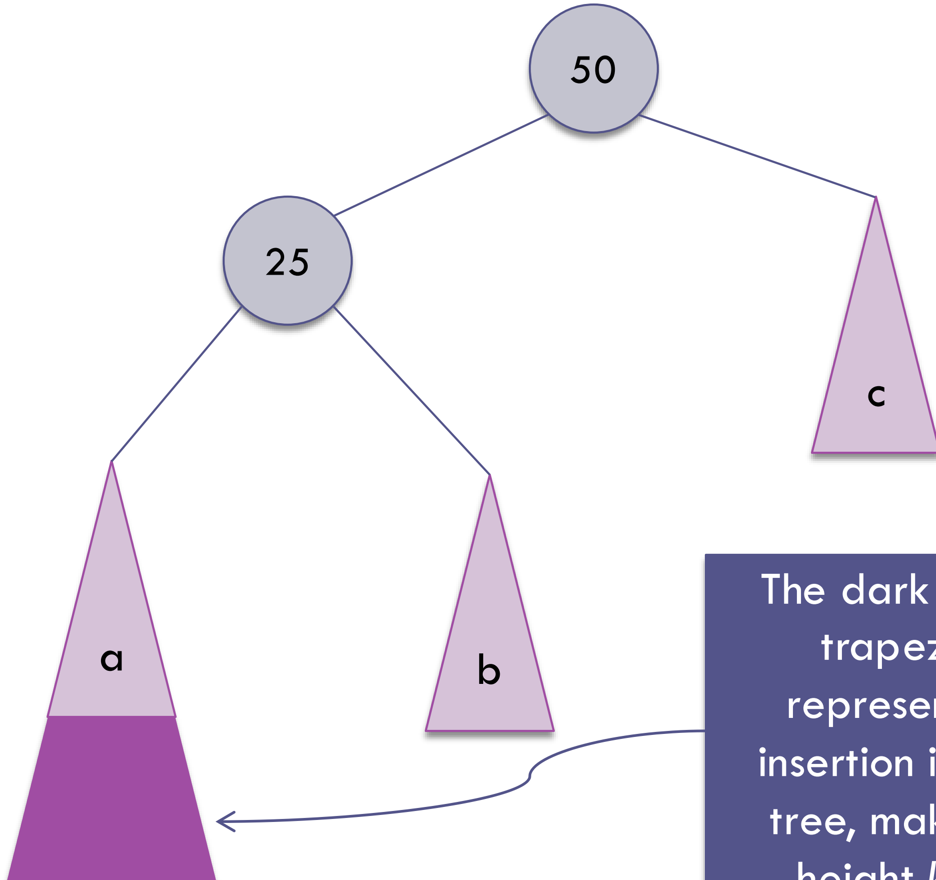
AVL Trees

- In 1962 G.M. Adel´son-Vel´skiî and E.M. Landis developed a self-balancing tree. The tree is known by their initials: *AVL*
- The AVL tree algorithm keeps track of the difference in height of each subtree
- As items are added to or removed from a tree, the balance of each subtree from the insertion or removal point up to the root is updated
- If the balance gets out of the range -1 to +1, the tree is rotated to bring it back into balance

Balancing a Left-Left Tree



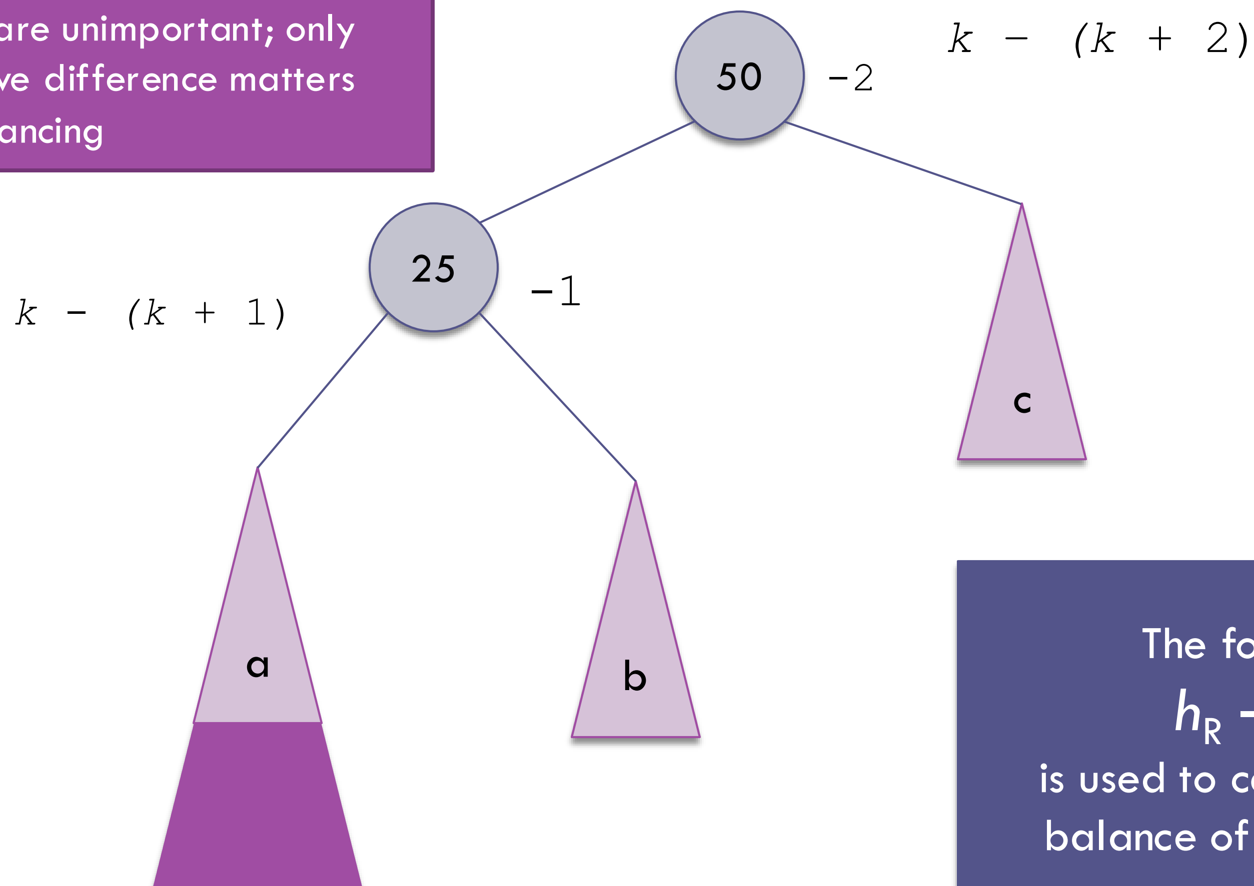
Balancing a Left-Left Tree (cont.)



The dark purple trapezoid represents an insertion into this tree, making its height $k + 1$

Balancing a Left-Left Tree (cont.)

The heights of the left and right subtrees are unimportant; only the relative difference matters when balancing

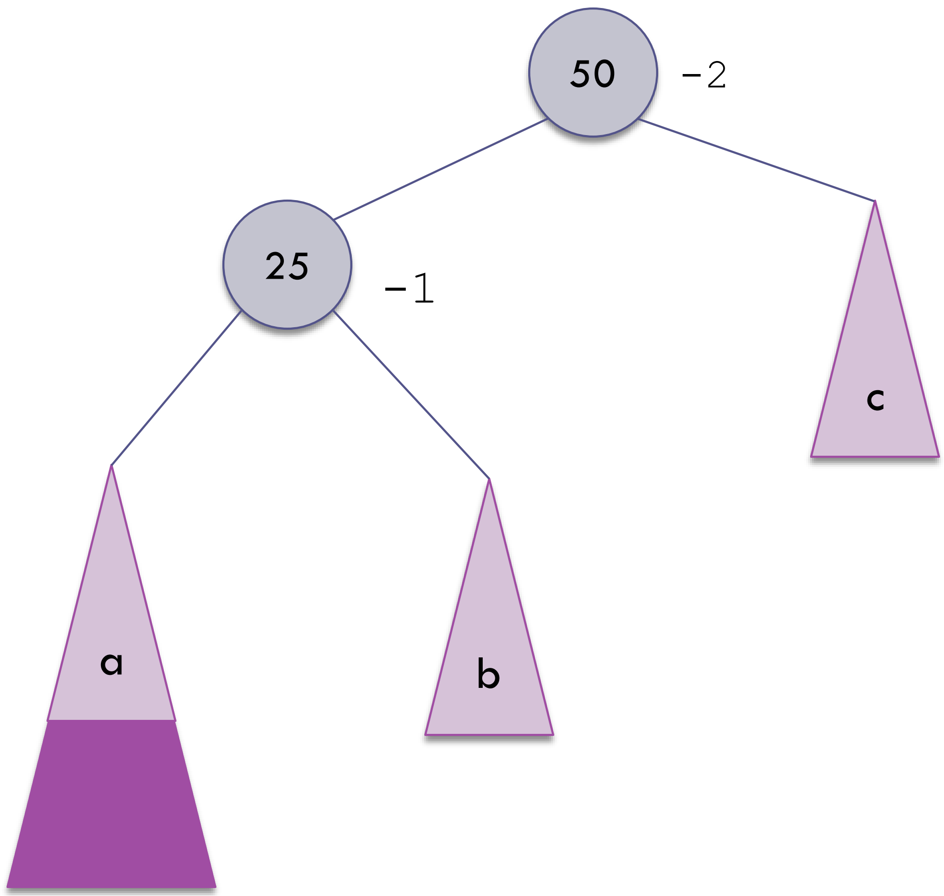


The formula

$h_R - h_L$

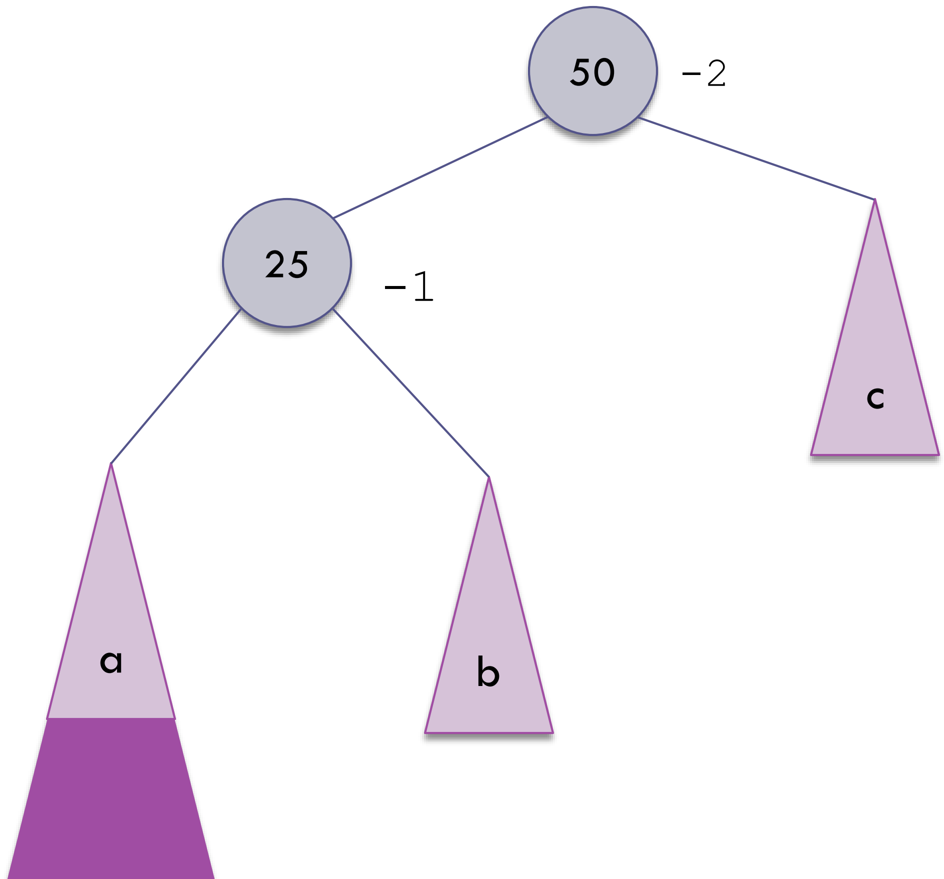
is used to calculate the balance of each node

Balancing a Left-Left Tree (cont.)



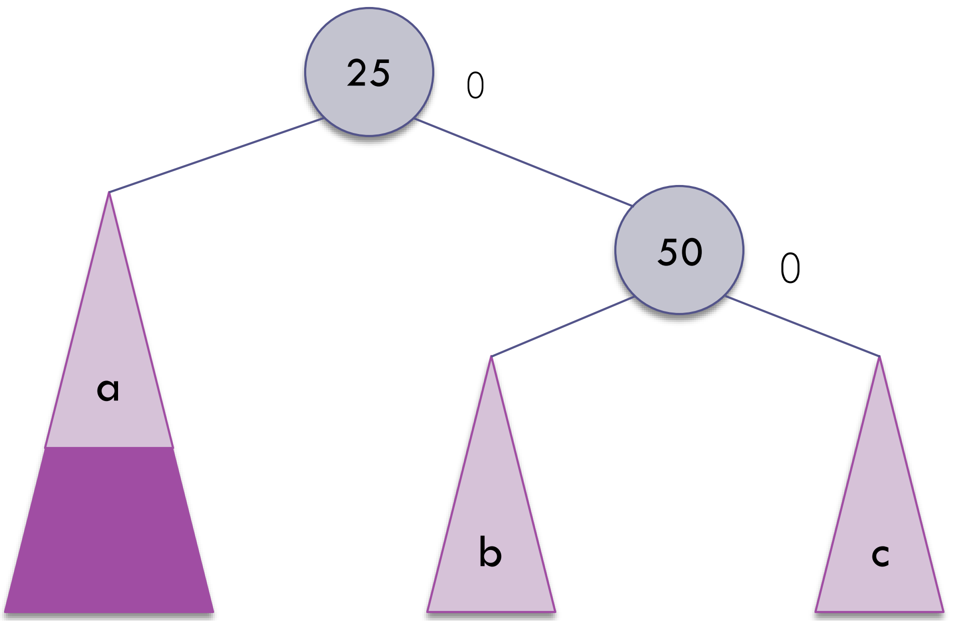
When the root and left subtree are both left-heavy, the tree is called a Left-Left tree

Balancing a Left-Left Tree (cont.)



A Left-Left tree can be balanced by a rotation right

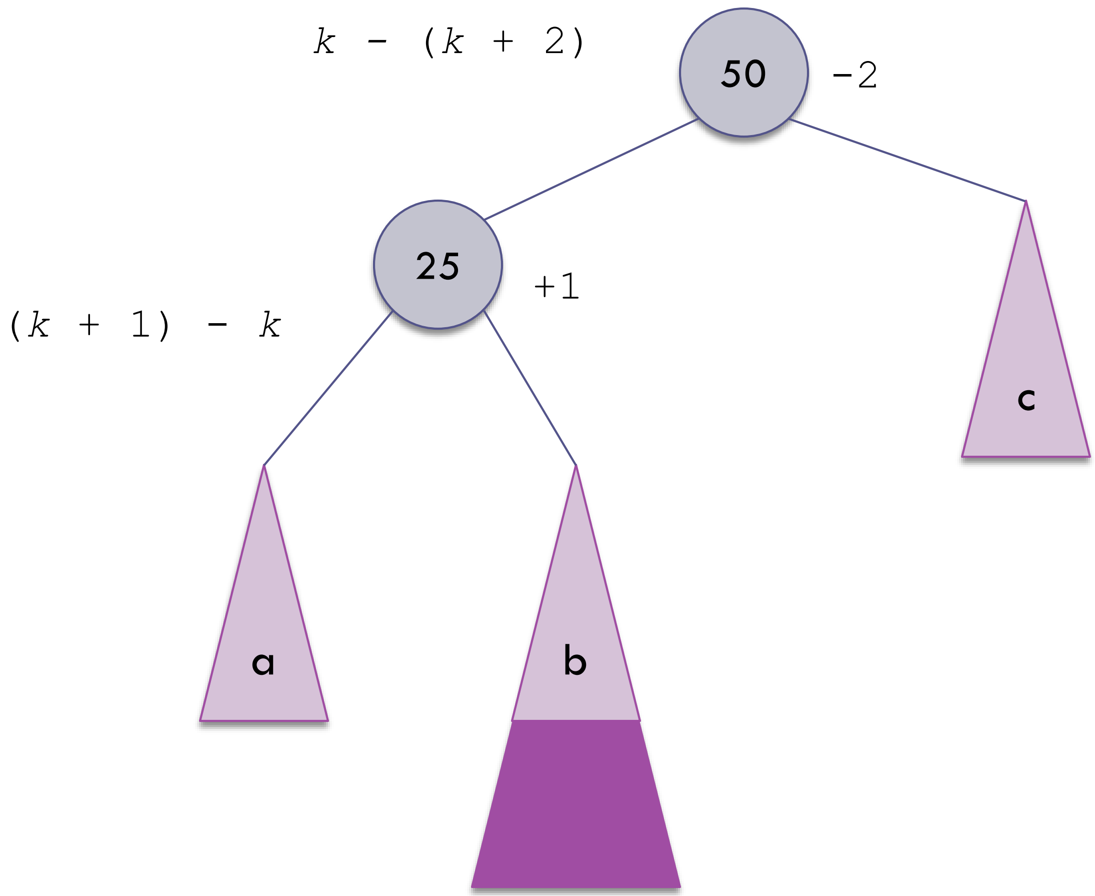
Balancing a Left-Left Tree (cont.)



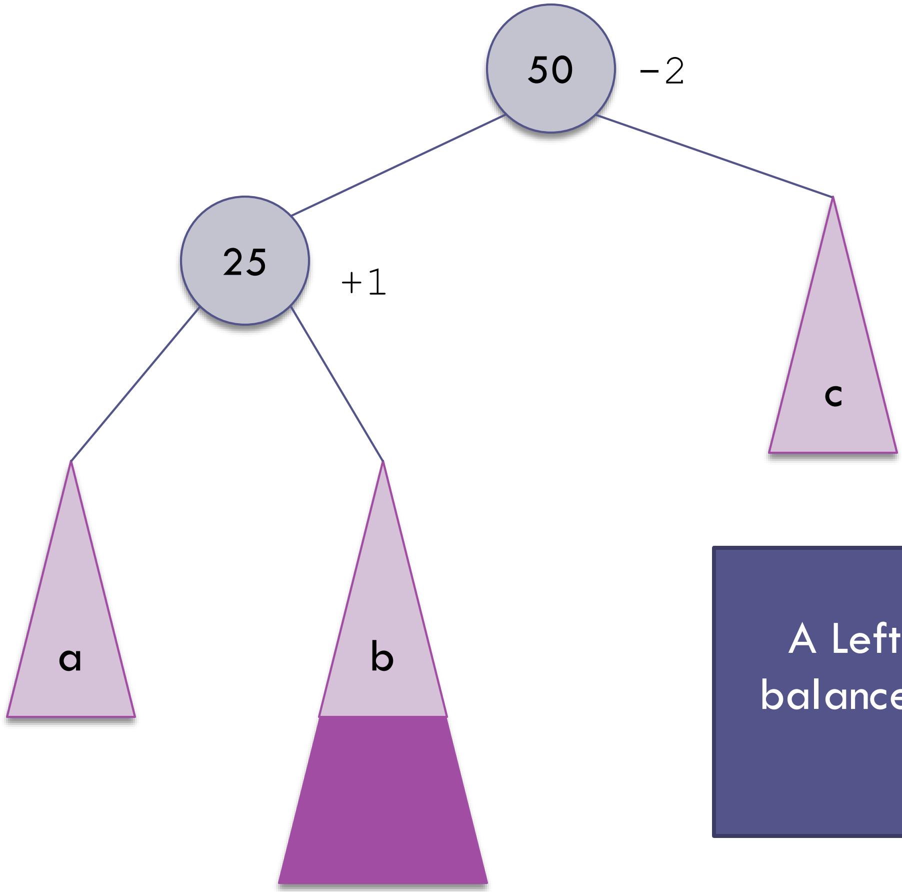
Balancing a Left-Left Tree (cont.)

- Even after insertion, the overall height has not increased

Balancing a Left-Right Tree

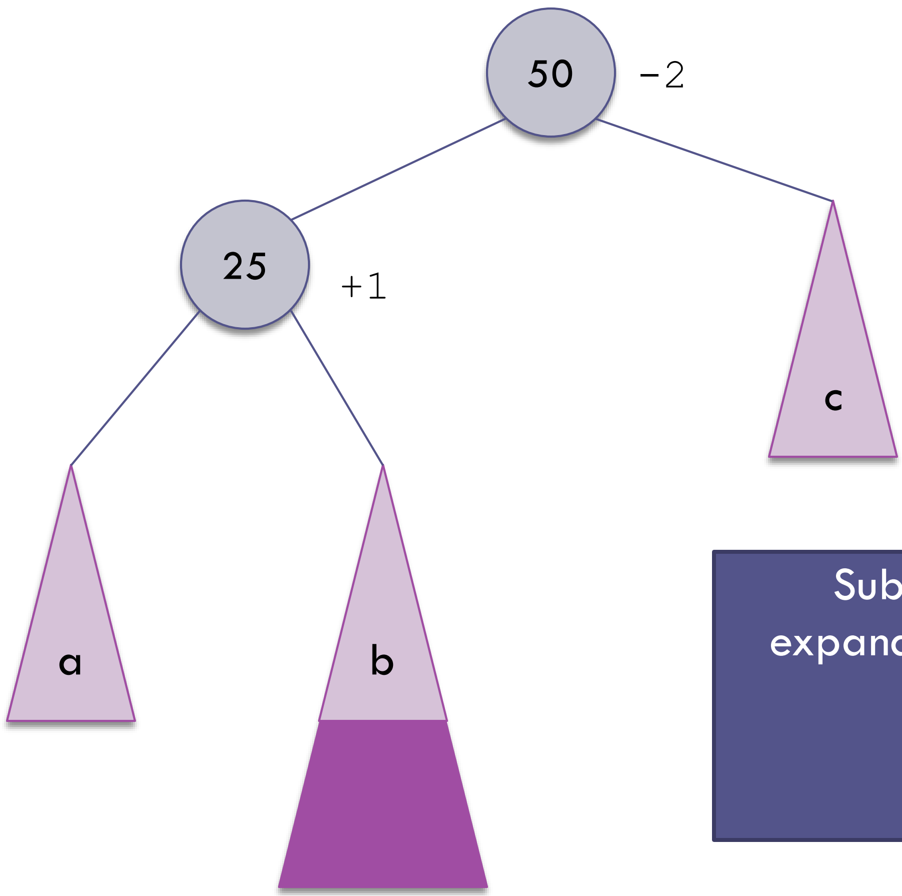


Balancing a Left-Right Tree (cont.)



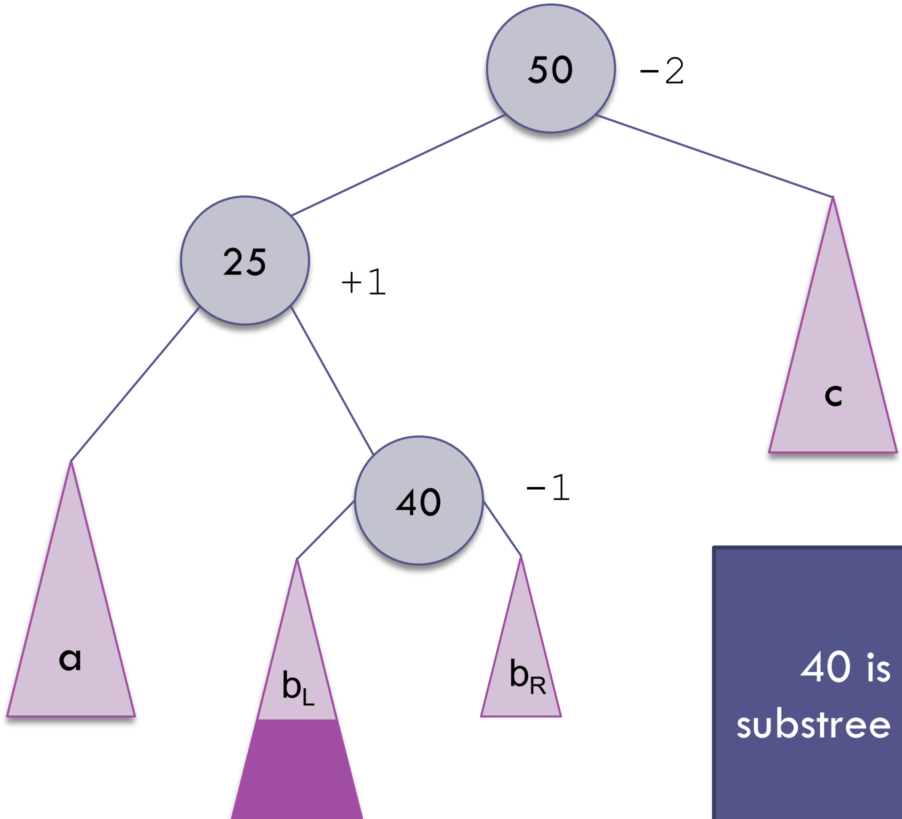
A Left-Right tree cannot be balanced by a simple rotation right

Balancing a Left-Right Tree (cont.)



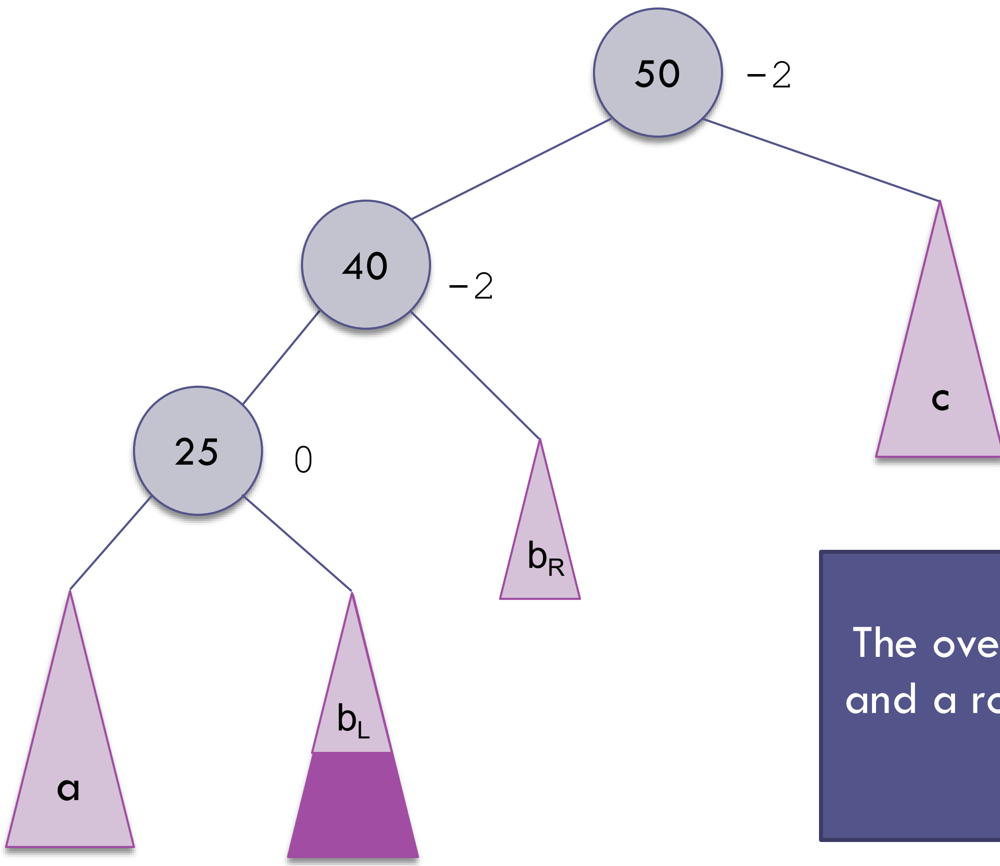
Subtree b needs to be expanded into its subtrees b_L and b_R

Balancing a Left-Right Tree (cont.)



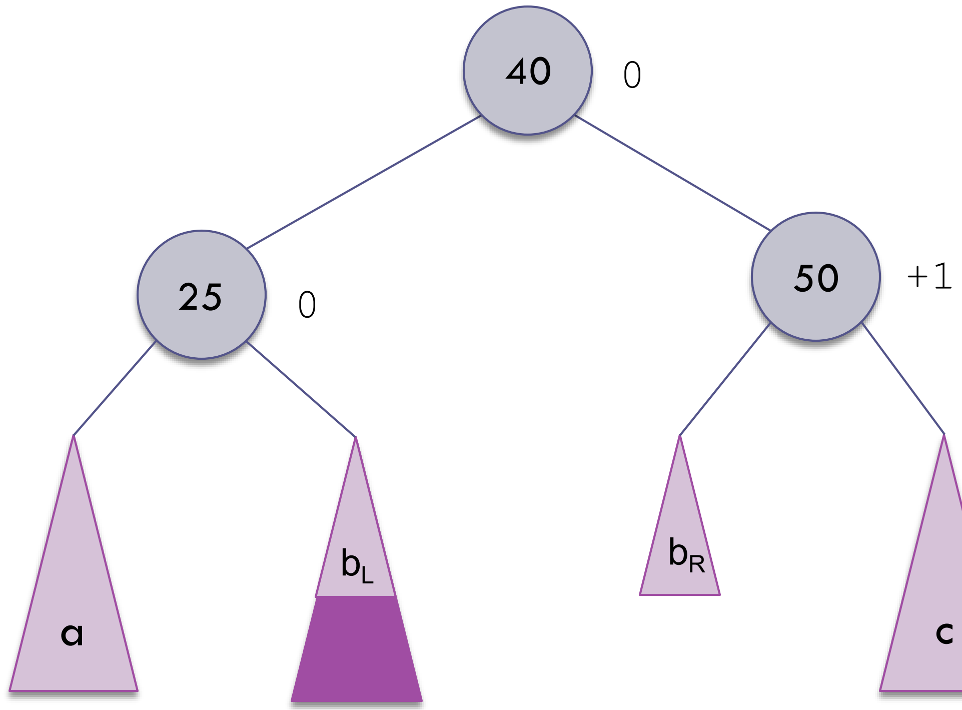
40 is left-heavy. The left substree can now be rotated left

Balancing a Left-Right Tree (cont.)

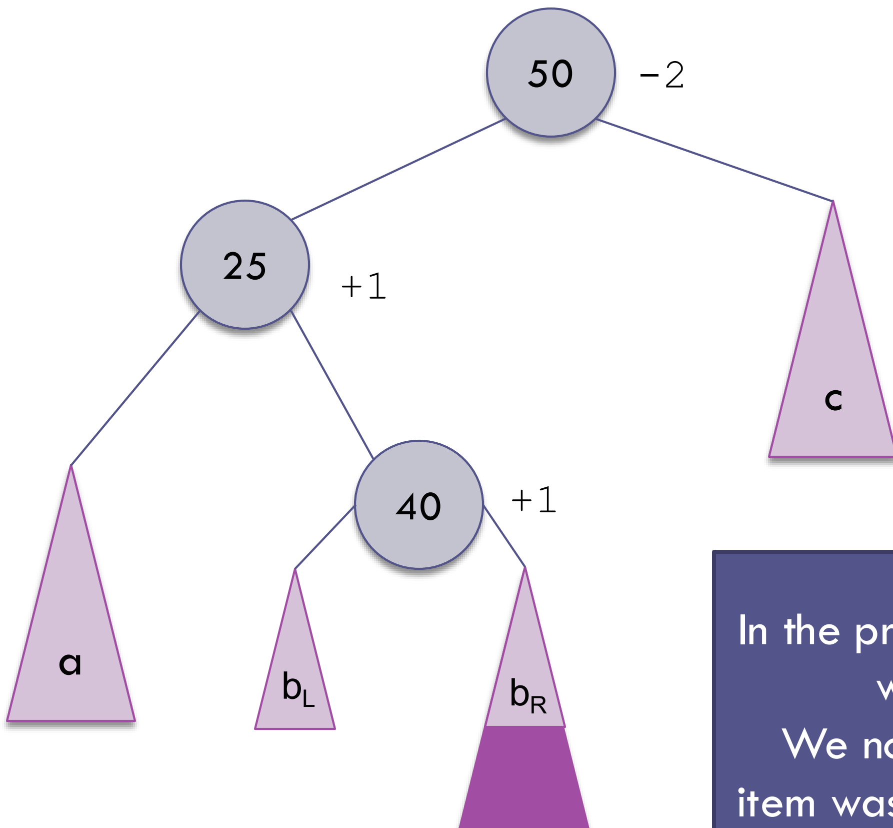


The overall tree is now Left-Left and a rotation right will balance it.

Balancing a Left-Right Tree (cont.)

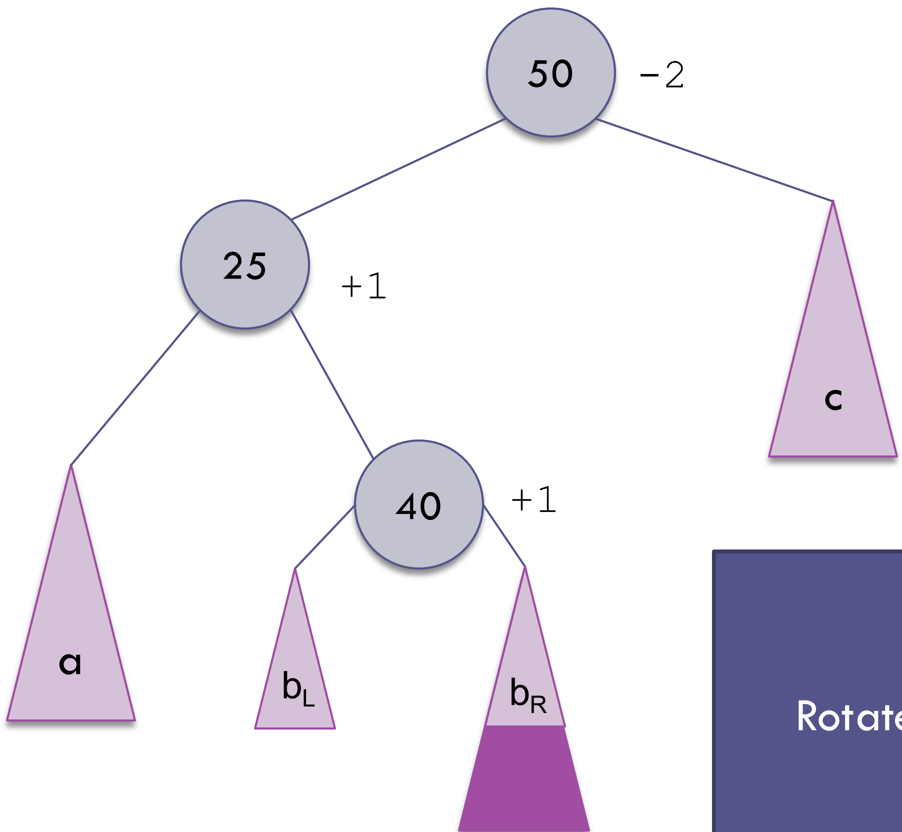


Balancing a Left-Right Tree (cont.)



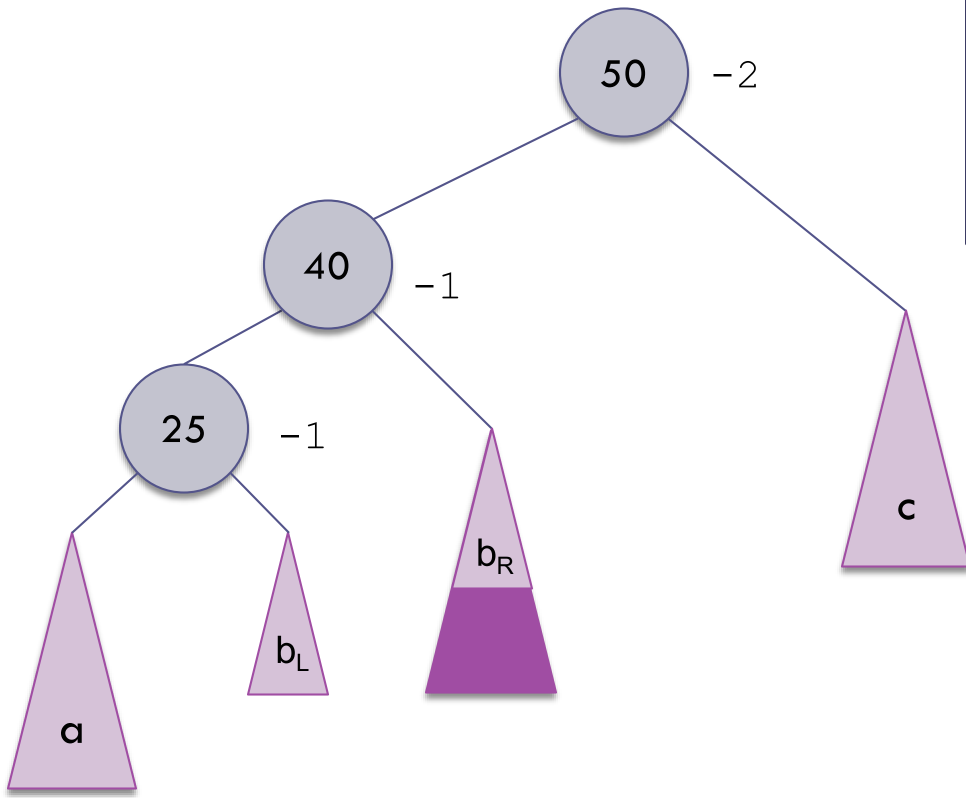
In the previous example, an item was inserted in b_L.
We now show the steps if an item was inserted into b_R instead

Balancing a Left-Right Tree (cont.)



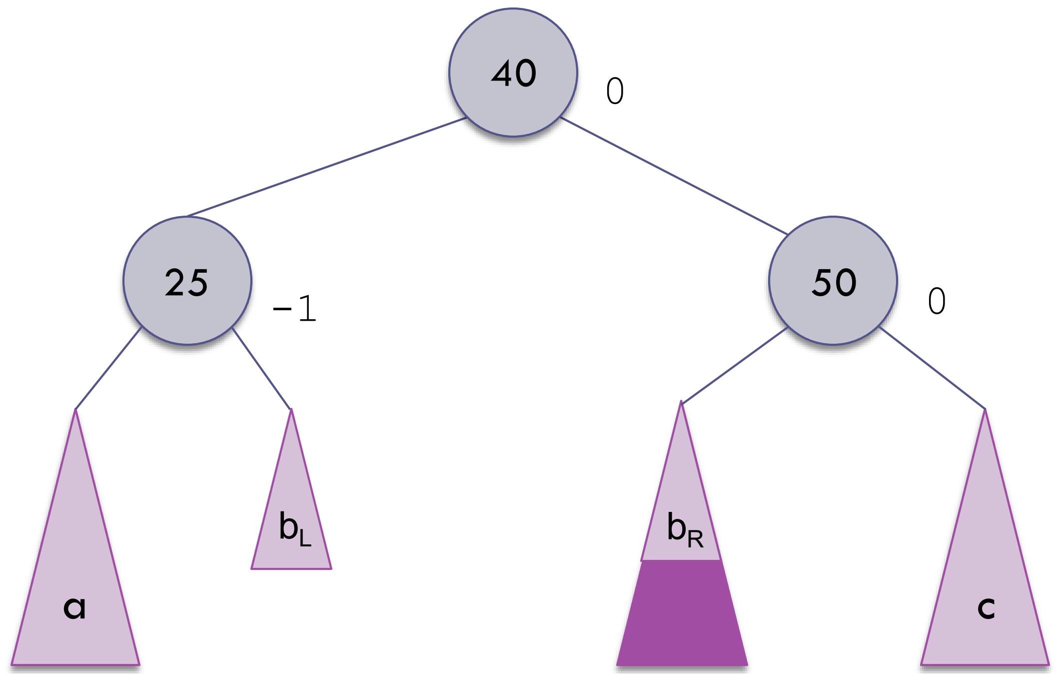
Rotate the left subtree left

Balancing a Left-Right Tree (cont.)



Rotate the tree right

Balancing a Left-Right Tree (cont.)



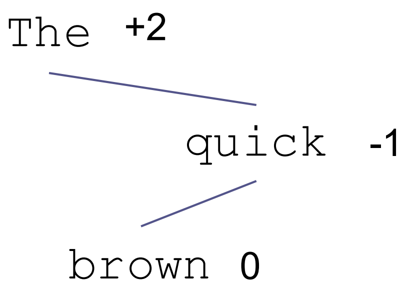
Four Kinds of Critically Unbalanced Trees

- Left-Left (parent balance is -2, left child balance is -1)
  - Rotate right around parent
- Left-Right (parent balance -2, left child balance +1)
  - Rotate left around child
  - Rotate right around parent
- Right-Right (parent balance +2, right child balance +1)
  - Rotate left around parent
- Right-Left (parent balance +2, right child balance -1)
  - Rotate right around child
  - Rotate left around parent

AVL Tree Example

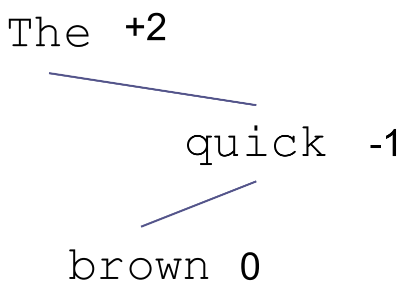
- Build an AVL tree from the words in
  "The quick brown fox jumps over the lazy dog"

AVL Tree Example (cont.)



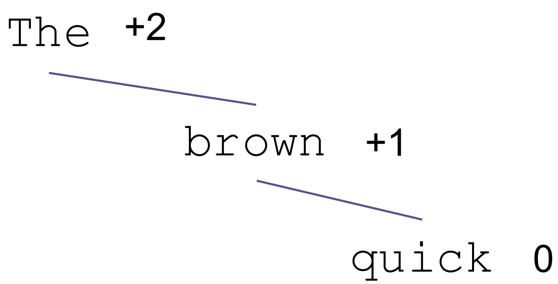
The overall tree is right-heavy (Right-Left)
parent balance = +2
right child balance = -1

AVL Tree Example (cont.)



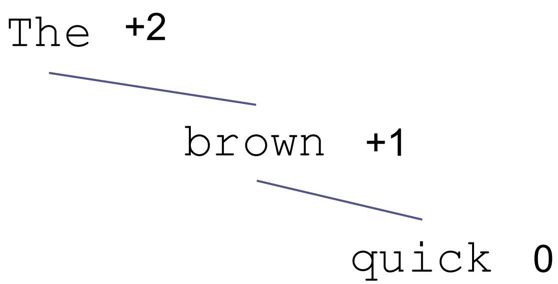
1. Rotate right around the child

AVL Tree Example (cont.)



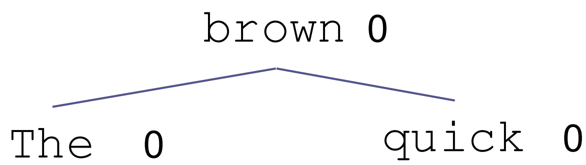
1. Rotate right around the child

AVL Tree Example (cont.)



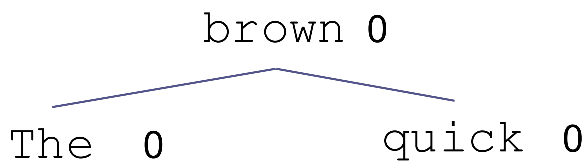
1. Rotate right around the child
2. Rotate left around the parent

AVL Tree Example (cont.)



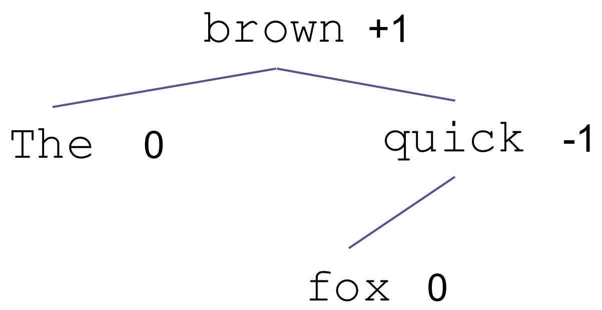
1. Rotate right around the child
2. Rotate left around the parent

AVL Tree Example (cont.)



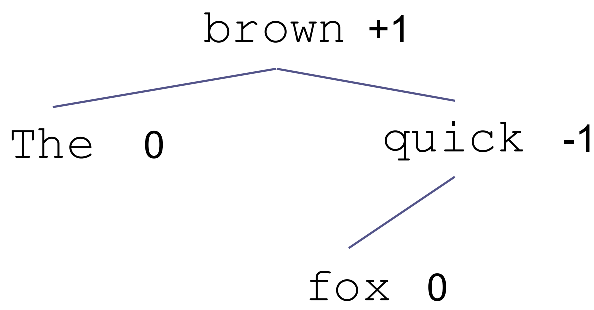
Insert *fox*

AVL Tree Example (cont.)



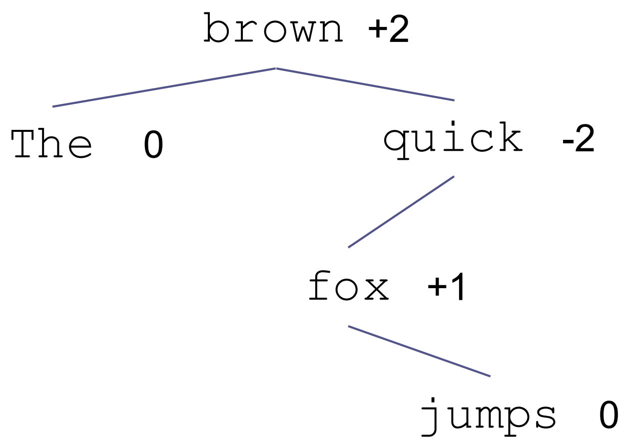
Insert *fox*

AVL Tree Example (cont.)



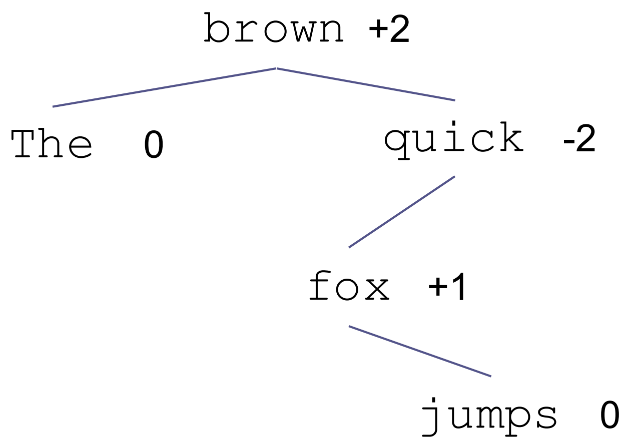
Insert *jumps*

AVL Tree Example (cont.)



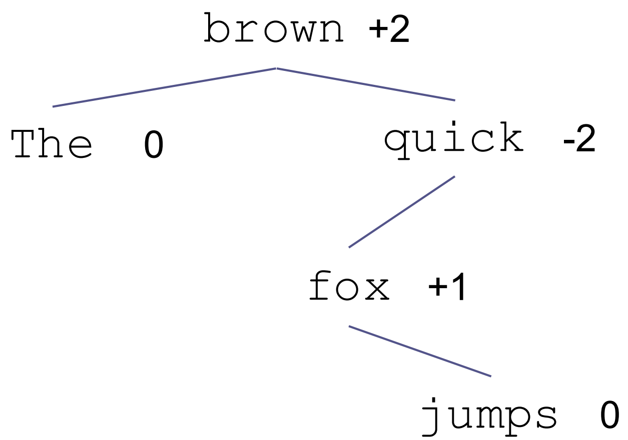
Insert *jumps*

AVL Tree Example (cont.)



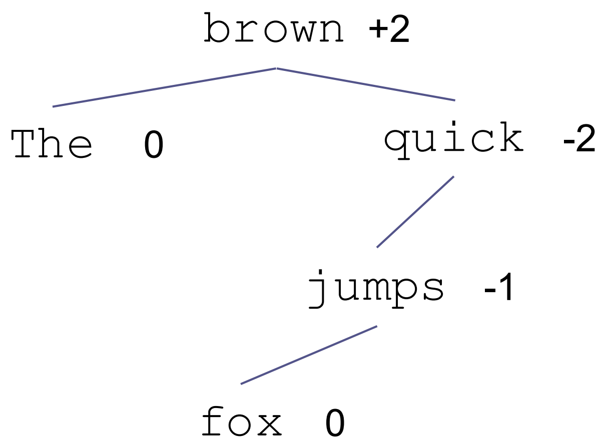
The tree is now left-heavy about *quick* (Left-Right case)

AVL Tree Example (cont.)



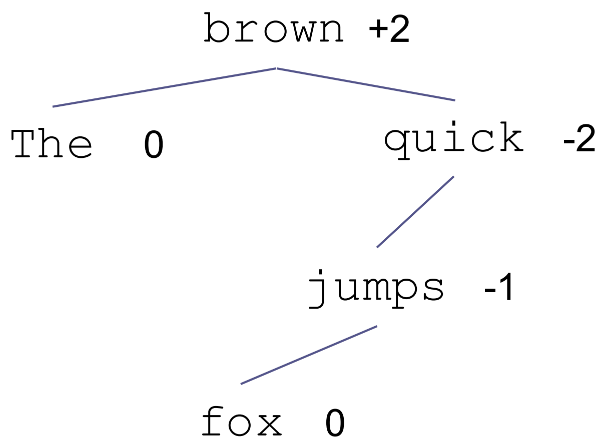
1. Rotate left around the child

AVL Tree Example (cont.)



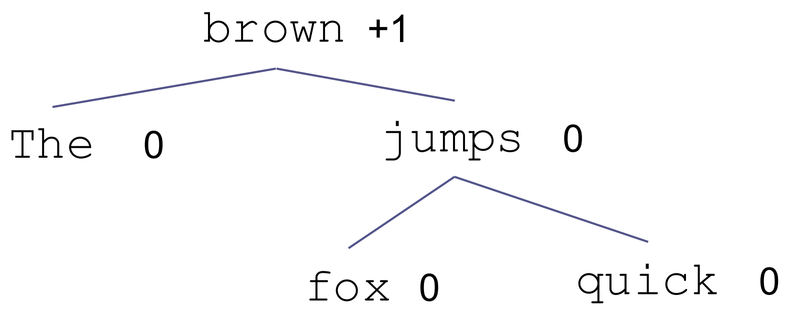
1. Rotate left around the child

AVL Tree Example (cont.)



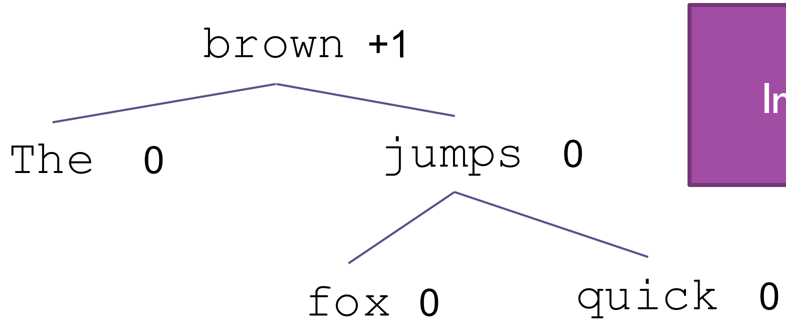
1. Rotate left around the child
2. Rotate right around the parent

AVL Tree Example (cont.)



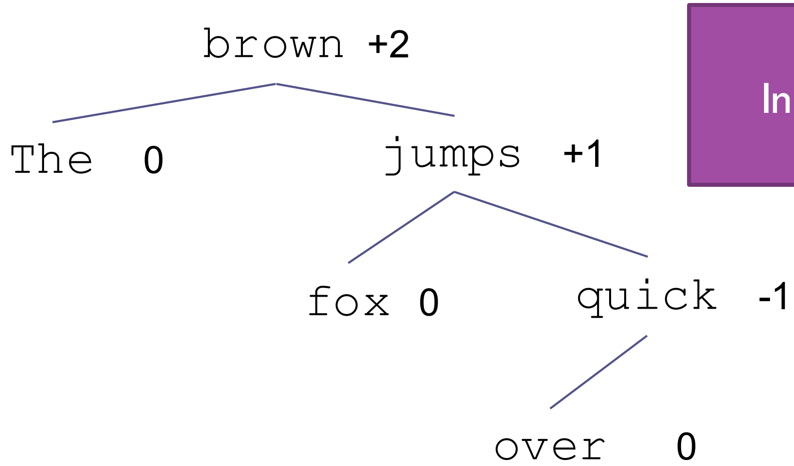
1. Rotate left around the child
2. Rotate right around the parent

AVL Tree Example (cont.)



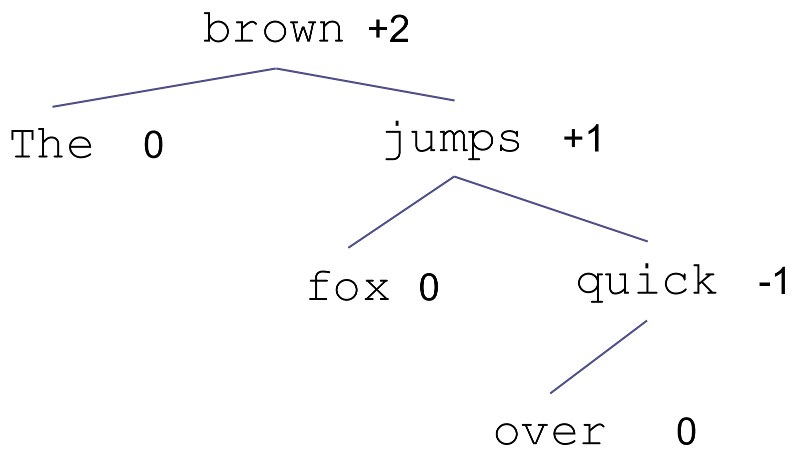
Insert over

AVL Tree Example (cont.)



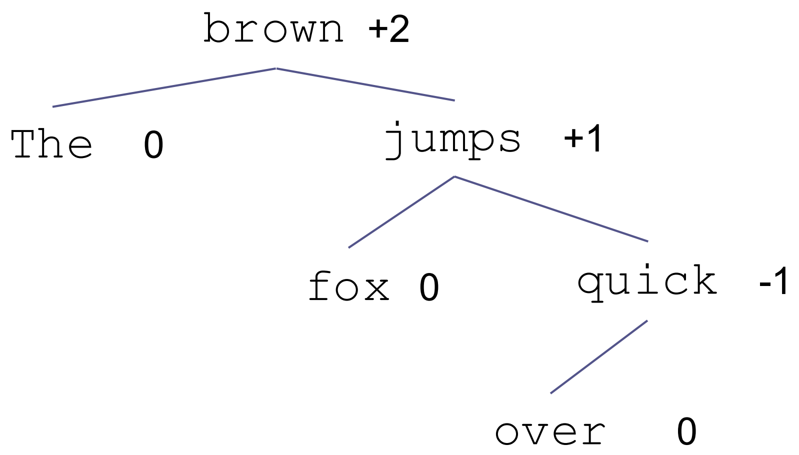
Insert *over*

AVL Tree Example (cont.)



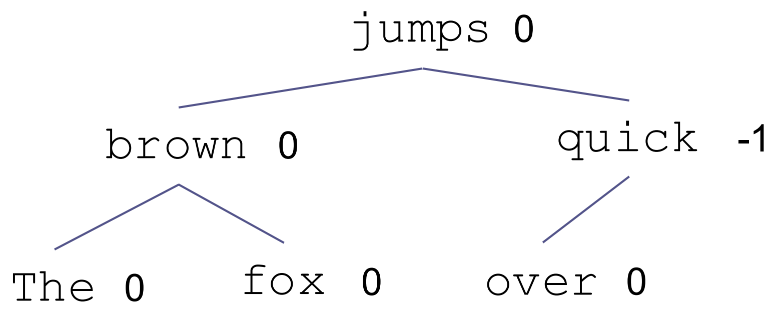
We now have a Right-Right imbalance

AVL Tree Example (cont.)



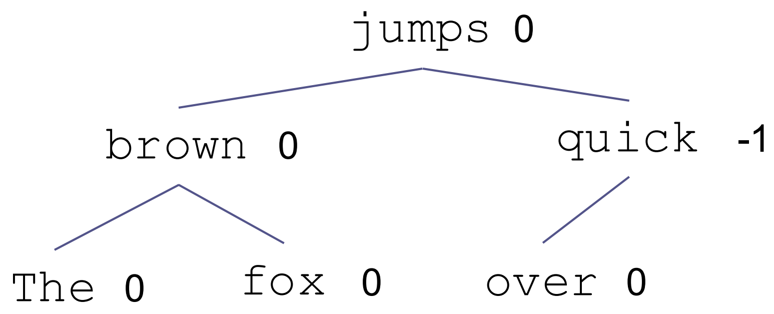
1. Rotate left around the parent

AVL Tree Example (cont.)



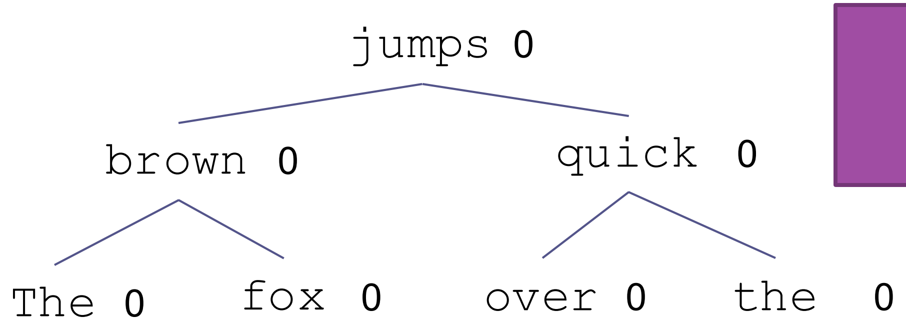
1. Rotate left around the parent

AVL Tree Example (cont.)



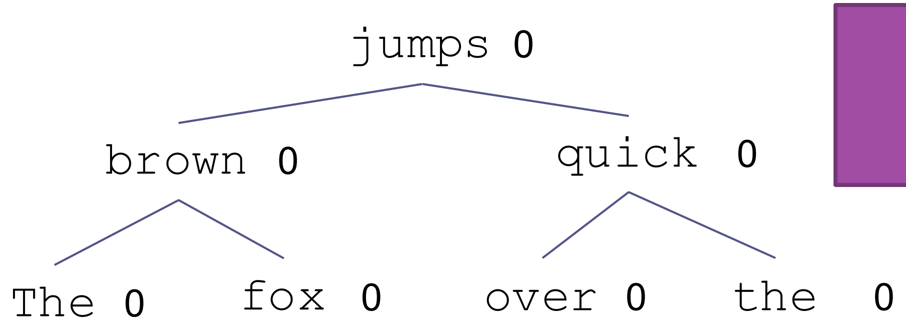
Insert *the*

AVL Tree Example (cont.)



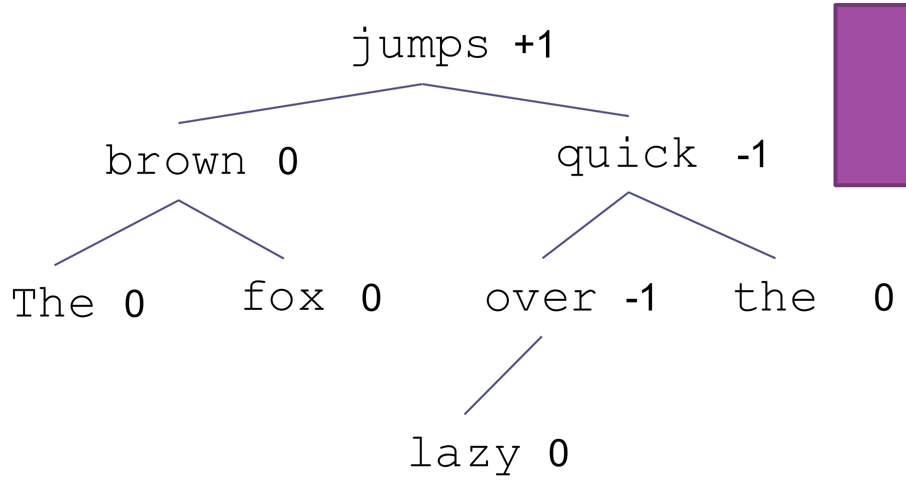
Insert *the*

AVL Tree Example (cont.)



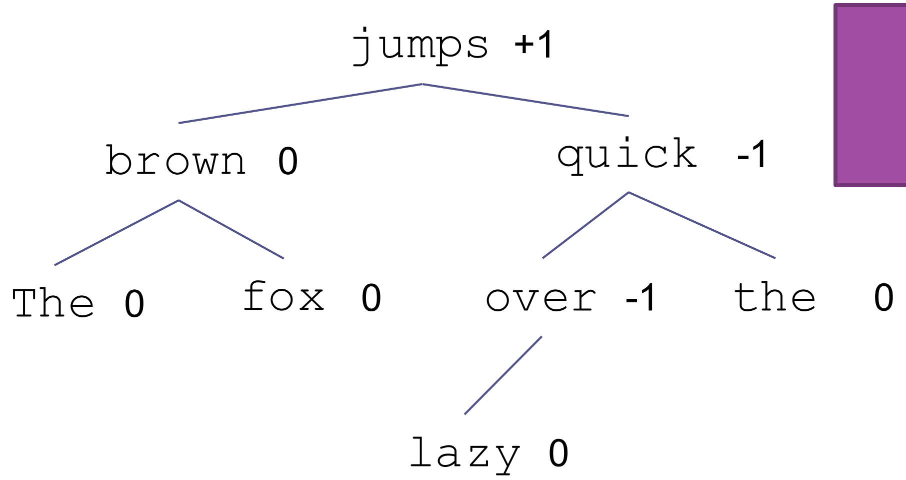
Insert *lazy*

AVL Tree Example (cont.)



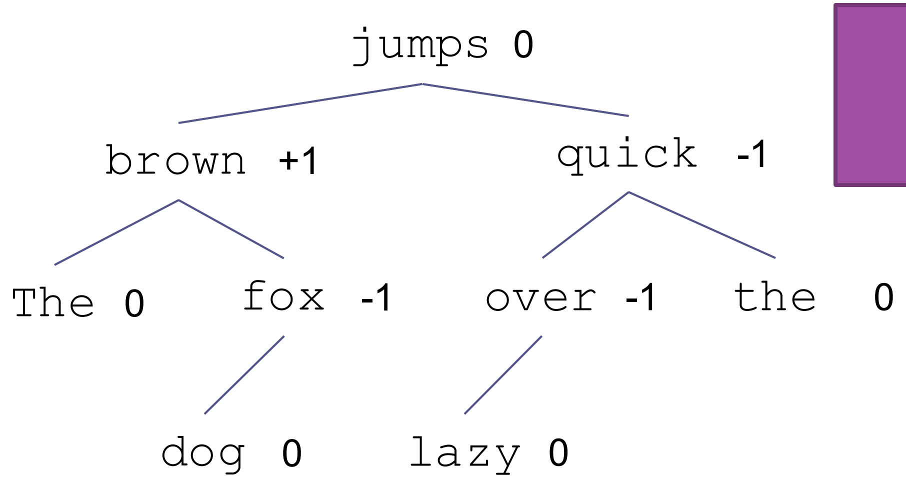
Insert *lazy*

AVL Tree Example (cont.)



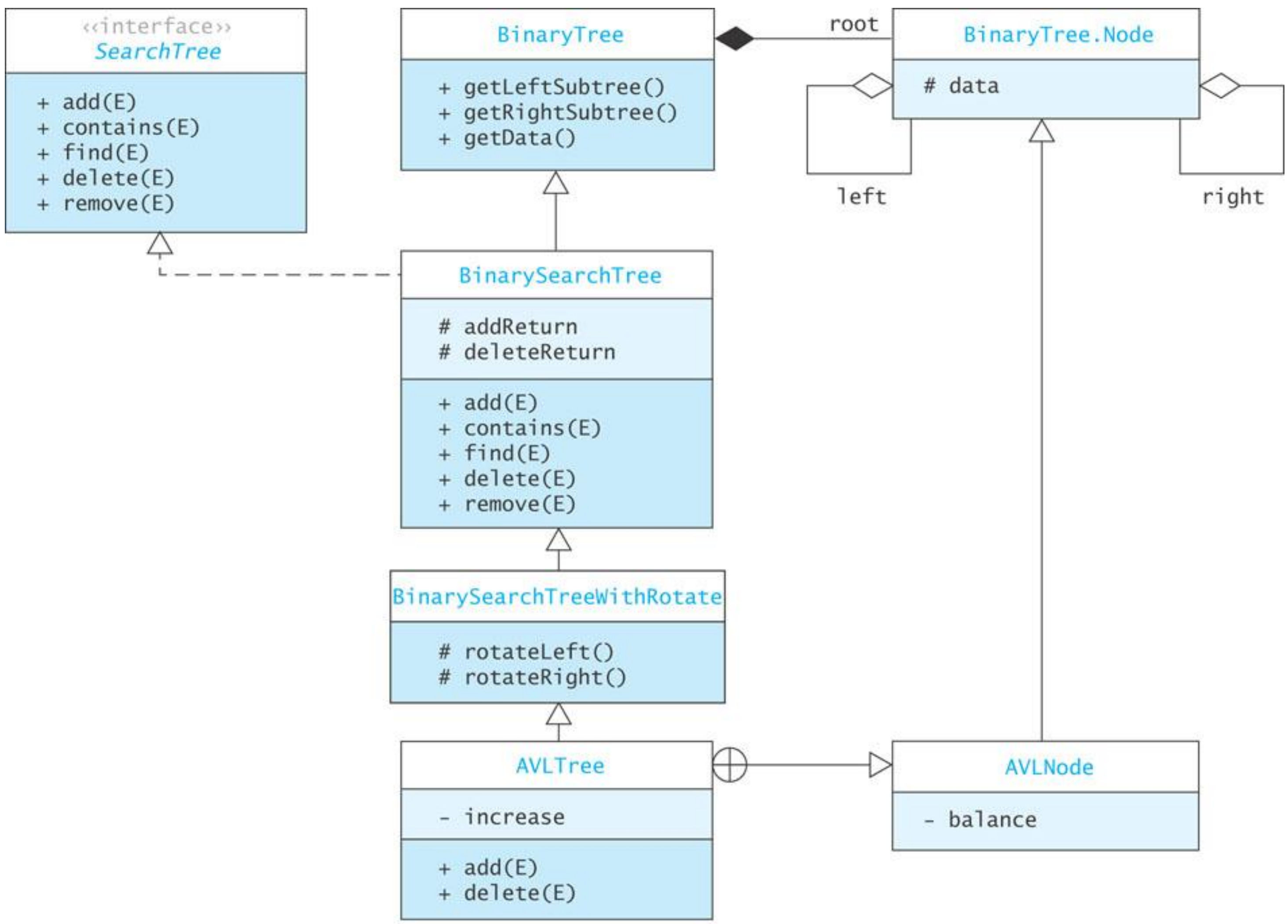
Insert *dog*

AVL Tree Example (cont.)



Insert *dog*

Implementing an AVL Tree



The `AVLNode` Class

- Listing 9.2 (`The AVLNode Class`, pages 482-483)

Inserting into an AVL Tree

- The easiest way to keep a tree balanced is never to let it become unbalanced
- If any node becomes critical, rebalance immediately
- Identify critical nodes by checking the balance at the root node as you return along the insertion path

Inserting into an AVL Tree (cont.)

Algorithm for Insertion into an AVL Tree

```
1.  if the root is null
2.        Create a new tree with the item at the root and return true.
    else if  the item is equal to root.data
3.        The item is already in the tree; return false.
    else if  the item is less than root.data
4.        Recursively insert the item in the left subtree.
5.        if the height of the left subtree has increased (increase is true)
6.              Decrement balance.
7.              if balance  is zero, reset increase to false.
8.              if balance is less than –1
9.                   Reset increase to false.
10.                  Perform a rebalanceLeft.
    else if  the item is greater than root.data
11.       The processing is symmetric to Steps 4 through 10. Note that balance
          is incremented if increase is true.
```

add Starter Method

```
/** add starter method.
    pre: the item to insert implements the Comparable interface.
    @param item The item being inserted.
    @return true if the object is inserted; false
         if the object already exists in the tree
    @throws ClassCastException if item is not Comparable
*/
@Override
public boolean add(E item) {
    increase = false;
    root = add((AVLNode<E>) root, item);
    return addReturn;
}
```

Recursive add method

```
/** Recursive add method. Inserts the given object into the tree.
     post: addReturn is set true if the item is inserted,
        false if the item is already in the tree.
     @param localRoot The local root of the subtree
     @param item The object to be inserted
     @return The new local root of the subtree with the item
        inserted
*/
private AVLNode<E> add(AVLNode<E> localRoot, E item)
if (localRoot == null) {
     addReturn = true;
     increase = true;
     return new AVLNode<E>(item);
}
if (item.compareTo(localRoot.data) == 0) {
     // Item is already in the tree.
     increase = false;
     addReturn = false;
     return localRoot;
}
```

Recursive add method (cont.)

```
else if (item.compareTo(localRoot.data) < 0) {
     // item < data
     localRoot.left = add((AVLNode<E>) localRoot.left, item);
. . .

if (increase) {
     decrementBalance(localRoot);
     if (localRoot.balance < AVLNode.LEFT_HEAVY) {
         increase = false;
         return rebalanceLeft(localRoot);
     }
}
return localRoot; // Rebalance not needed.
```

Initial Algorithm for `rebalanceLeft`

Initial Algorithm for `rebalanceLeft`

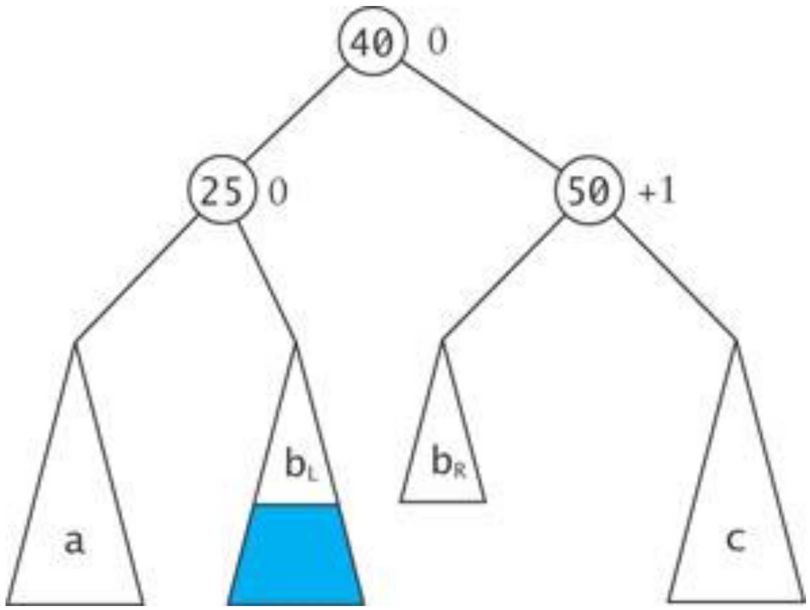
1. **`if`** the left subtree has positive balance (Left-Right case)
2. Rotate left around left subtree root.
3. Rotate right.

Effect of Rotations on Balance

- The rebalance algorithm on the previous slide was incomplete as the balance of the nodes was not adjusted
- For a Left-Left tree the balances of the new root node and of its right child are 0 after a right rotation
- Left-Right is more complicated:
  - the balance of the root is 0

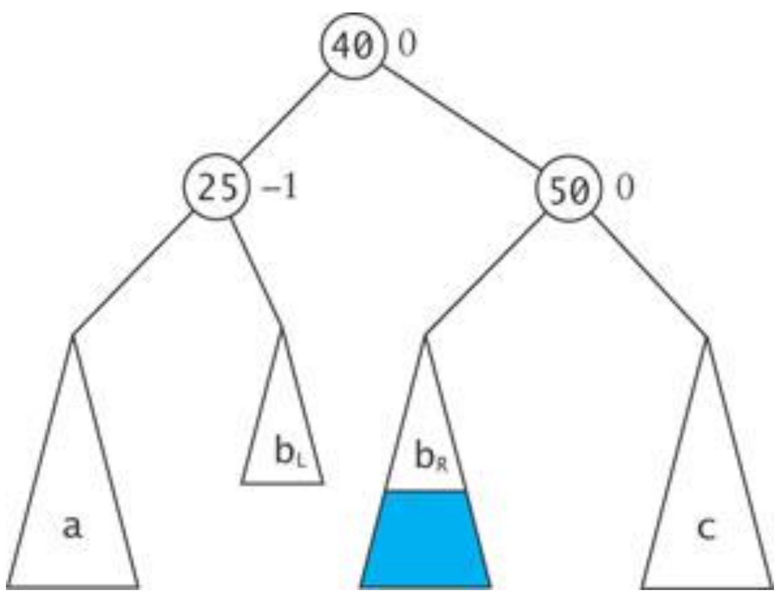
Effect of Rotations on Balance (cont.)

- if the critically unbalanced situation was due to an insertion into
  - subtree b_L (Left-Right-Left case), the balance of the root's left child is 0 and the balance of the root's right child is +1



Effect of Rotations on Balance (cont.)

- if the critically unbalanced situation was due to an insertion into
  - subtree b_R (Left-Right-Right case), the balance of the root's left child is -1 and the balance of the root's right child is 0



Revised Algorithm for `rebalanceLeft`

Revised Algorithm for `rebalanceLeft`

```
1.  if the left subtree has a positive balance (Left-Right case)
2.      if the left-left subtree has a negative balance (Left-Right-Left case)
3.          Set the left subtree (new left subtree) balance to 0.
4.          Set the left-left subtree (new root) balance to 0.
5.          Set the local root (new right subtree) balance to +1.
        else (Left-Right-Right case)
6.          Set the left subtree (new left subtree) balance to –1.
7.          Set the left-left subtree (new root) balance to 0.
8.          Set the local root (new right subtree) balance to 0.
9.      Rotate the left subtree left.
    else (Left-Left case)
10.     Set the left subtree balance to 0.
11.     Set the local root balance to 0.
12. Rotate the local root right.
```

Method `rebalanceLeft`

- Listing 9.3 (`The rebalanceLeft Method`, page 487)

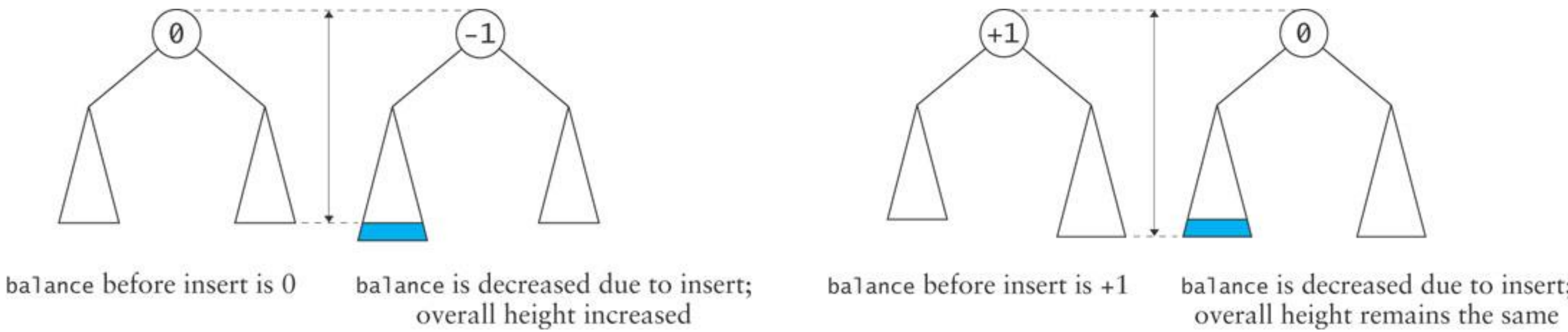
Method `rebalanceRight`

- The `rebalanceRight` method is symmetric with respect to the `rebalanceLeft` method

Method `decrementBalance`

- As we return from an insertion into a node's left subtree, we need to decrement the balance of the node
- We also need to indicate if the subtree height at that node has not increased (setting `increase` to `false`)

Method `decrementBalance` (cont.)



- Two cases to consider:
  - a balanced node – insertion into its left subtree will make it left-heavy and its height will increase by 1
  - a right-heavy node – insertion into its left subtree will cause it to become balanced and its height will not increase

Method `decrementBalance` (cont.)

```
private void decrementBalance(AVLNode<E> node) {
    // Decrement the balance.
    node.balance--;
    if (node.balance == AVLNode.BALANCED) {
       /** If now balanced, overall height has not increased. */
       increase = false;
    }

}
```

Removal from an AVL Tree

- Removal
  - from a left subtree, increases the balance of the local root
  - from a right subtree, decreases the balance of the local root
- The binary search tree removal method can be adapted for removal from an AVL tree
- A data field `decrease` tells the previous level in the recursion that there was a decrease in the height of the subtree from which the return occurred
- The local root balance is incremented or decremented based on this field
- If the balance is outside the threshold, a rebalance method is called to restore balance

Removal from an AVL Tree (cont.)

- Methods `decrementBalance`, `incrementBalance`, `rebalanceLeft`, and `rebalanceRight` need to be modified to set the value of `decrease` and `increase` after a node's balance is decremented
- Each recursive return can result in a further need to rebalance

Performance of the AVL Tree

- Since each subtree is kept as close to balanced as possible, the AVL has expected O(log n)
- Each subtree is allowed to be out of balance ± 1 so the tree may contain some holes
- In the worst case (which is rare) an AVL tree can be 1.44 times the height of a full binary tree that contains the same number of items
- Ignoring constants, this still yields O(log n) performance
- Empirical tests show that on average log $n + 0.25$ comparisons are required to insert the nth item into an AVL tree – close to a corresponding complete binary search tree

Red-Black Trees

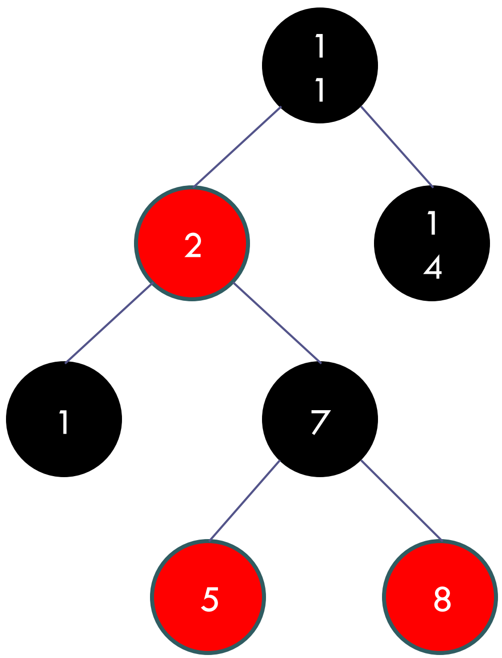
Section 9.3

Red-Black Trees

- Rudolf Bayer developed the red-black tree as a special case of his B-tree
- Leo Guibas and Robert Sedgewick refined the concept and introduced the color convention

Red-Black Trees (cont.)

- A red-black tree maintains the following invariants:
  1. A node is either red or black
  2. The root is always black
  3. A red node always has black children (a `null` reference is considered to refer to a black node)
  4. The number of black nodes in any path from the root to a leaf is the same



Red-Black Trees (cont.)

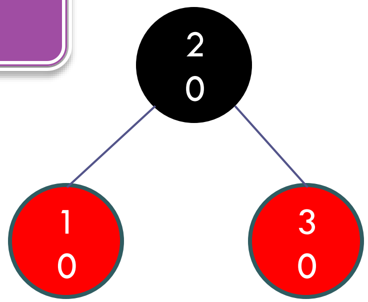
- Height is determined by counting only black nodes
- A red-black tree is always balanced because the root node's left and right subtrees must be the same height

Insertion into a Red-Black Tree

- The algorithm follows the same recursive search process used for all binary search trees to reach the insertion point
- When a leaf is found, the new item is inserted and initially given the color red
- If the parent is black, we are done; otherwise there is some rearranging to do
- We introduce three situations ("cases") that may occur when a node is inserted; more than one can occur after an insertion

Insertion into a Red-Black Tree (cont.)

CASE 1

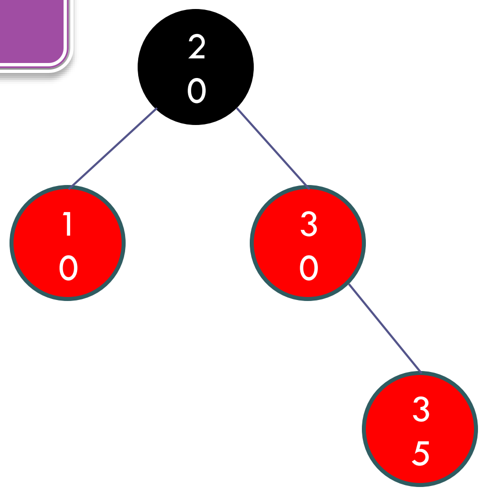


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 1



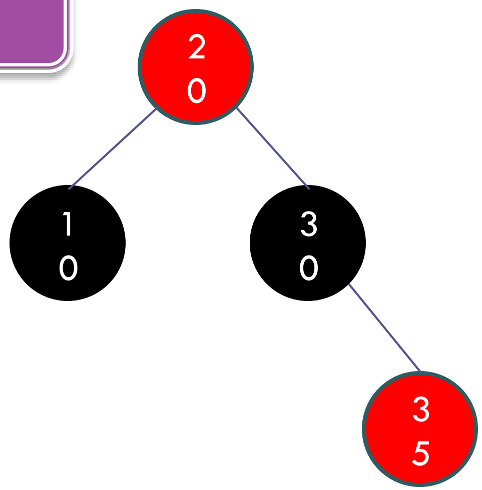
If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 1



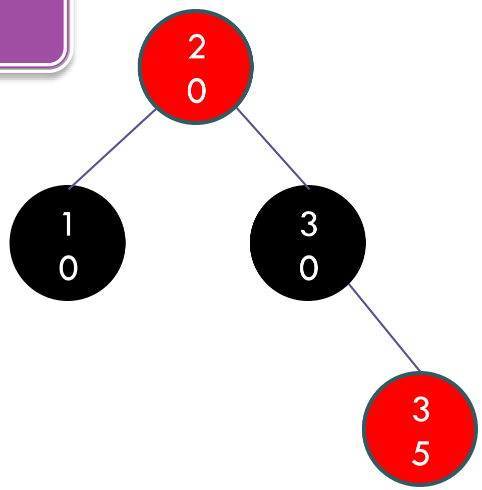
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red

Insertion into a Red-Black Tree (cont.)

CASE 1



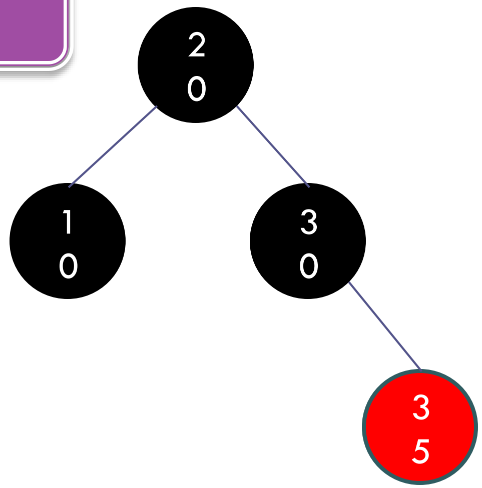
The root can be changed to black and still maintain invariant 4

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 1



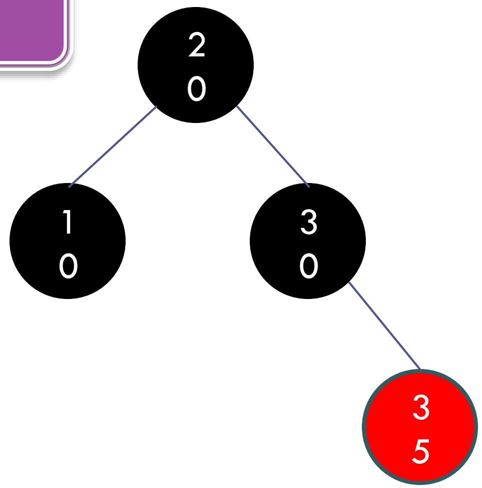
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

The root can be changed to black and still maintain invariant 4

Insertion into a Red-Black Tree (cont.)

CASE 1



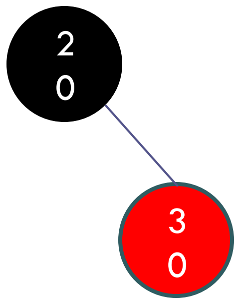
Balanced tree

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 2

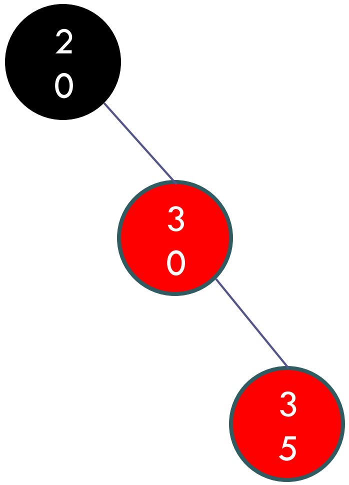


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 2



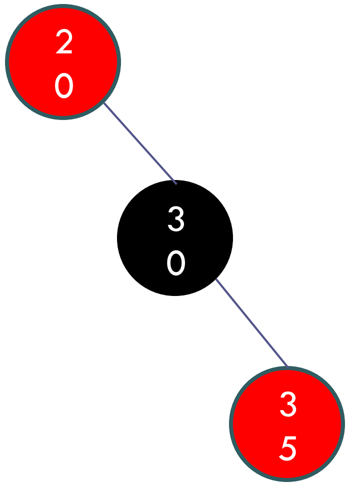
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red

Insertion into a Red-Black Tree (cont.)

CASE 2



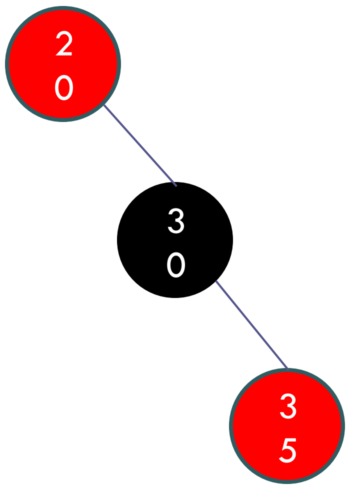
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red

Insertion into a Red-Black Tree (cont.)

CASE 2



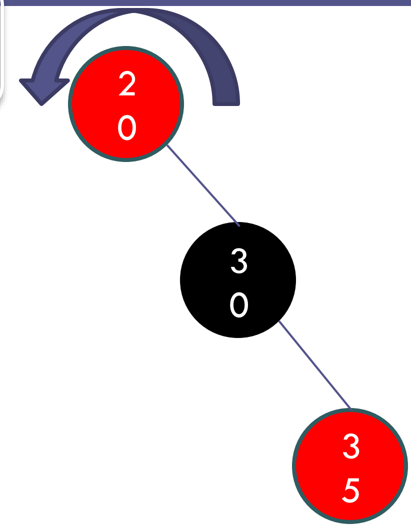
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

There is one black node on the right and none on the left, which violates invariant 4

Insertion into a Red-Black Tree (cont.)

CASE 2



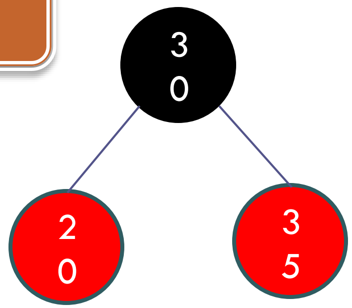
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate left around the grandparent to correct this

Insertion into a Red-Black Tree (cont.)

CASE 2



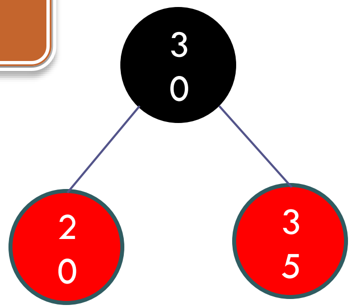
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate left around the grandparent to correct this

Insertion into a Red-Black Tree (cont.)

CASE 2



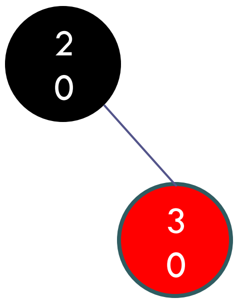
Balanced tree

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 3

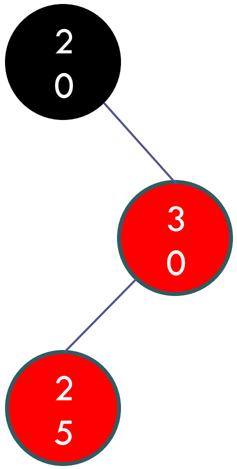


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 3



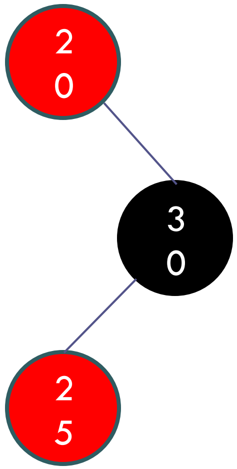
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red

Insertion into a Red-Black Tree (cont.)

CASE 3



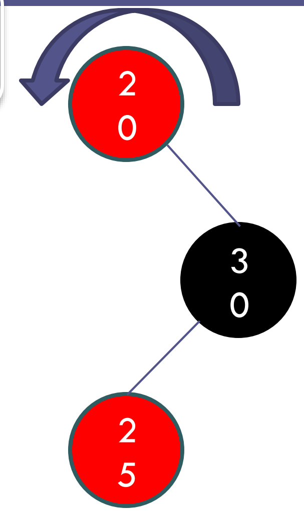
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

If a parent is red (with no sibling), it can be changed to black, and the grandparent to red

Insertion into a Red-Black Tree (cont.)

CASE 3



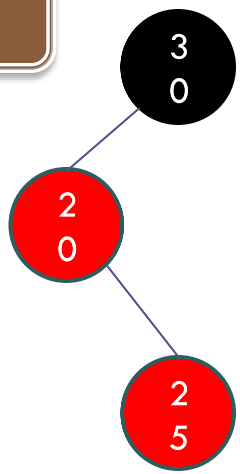
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

A rotation left does not fix the violation of #4

Insertion into a Red-Black Tree (cont.)

CASE 3



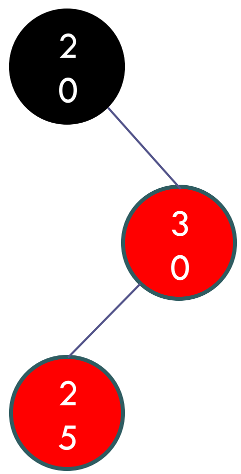
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

A rotation left does not fix the violation of #4

Insertion into a Red-Black Tree (cont.)

CASE 3



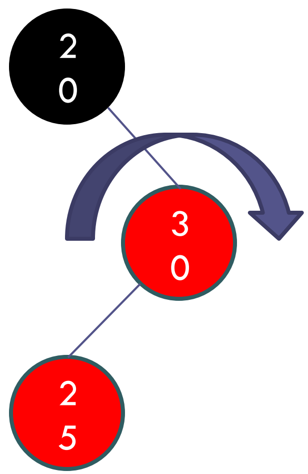
Back-up to the beginning (don't perform rotation or change colors)

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 3



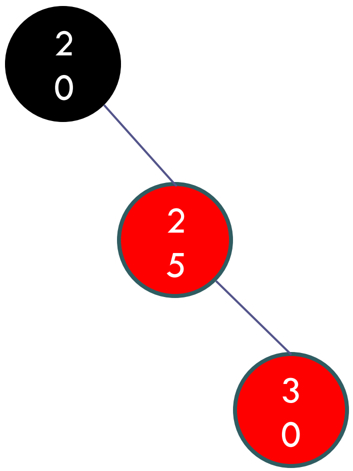
Rotate right about the parent so that the red child is on the same side of the parent as the parent is to the grandparent

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)

CASE 3



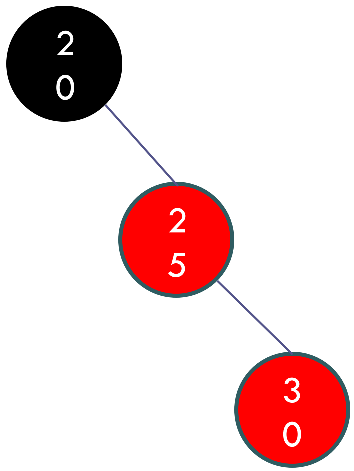
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Rotate right about the parent so that the red child is on the same side of the parent as the parent is to the grandparent

Insertion into a Red-Black Tree (cont.)

CASE 3



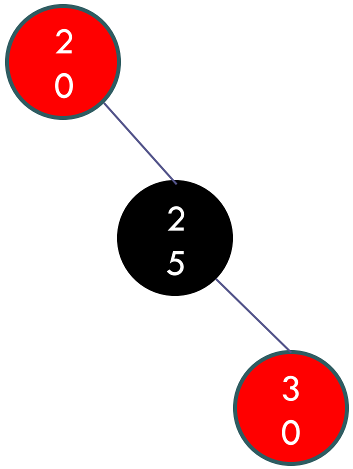
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

NOW, change colors

Insertion into a Red-Black Tree (cont.)

CASE 3



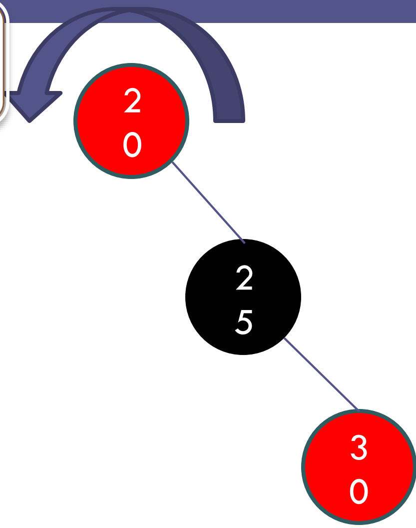
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

NOW, change colors

Insertion into a Red-Black Tree (cont.)

CASE 3



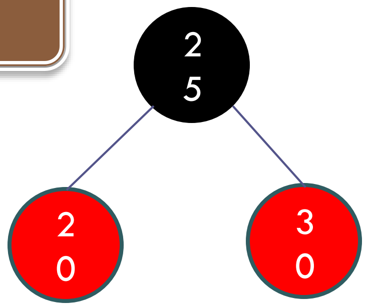
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

and rotate left . . .

Insertion into a Red-Black Tree (cont.)

CASE 3



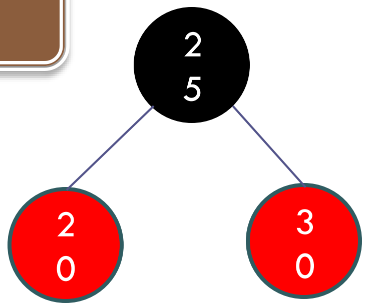
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

and rotate left. . .

Insertion into a Red-Black Tree (cont.)

CASE 3

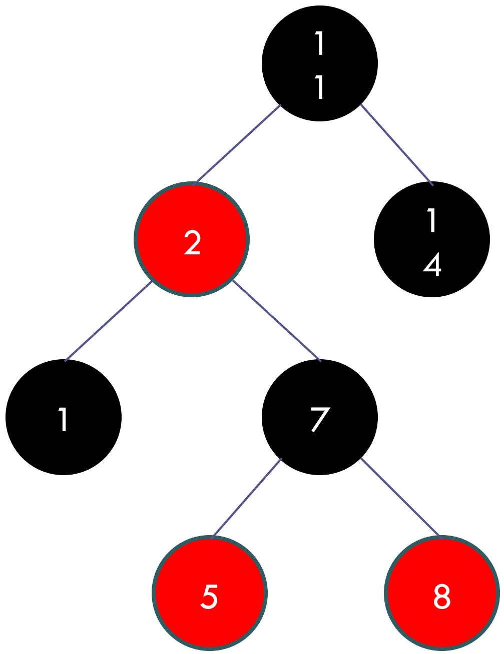


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Balanced tree

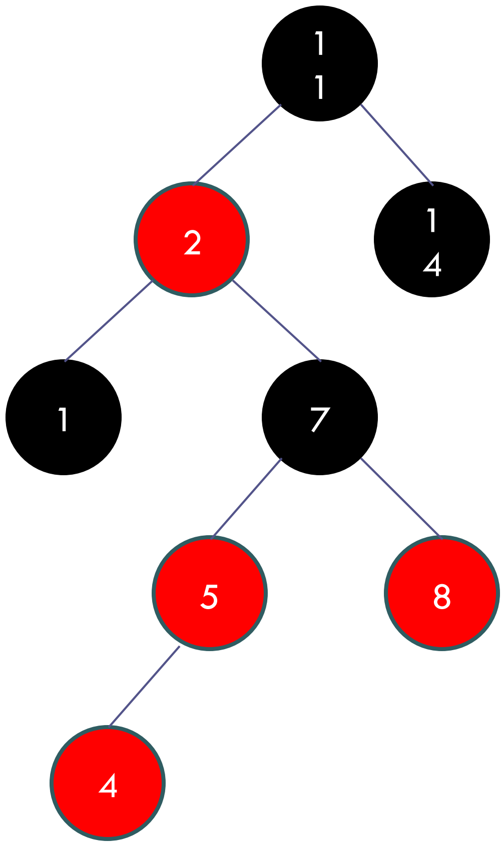
Insertion into a Red-Black Tree (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

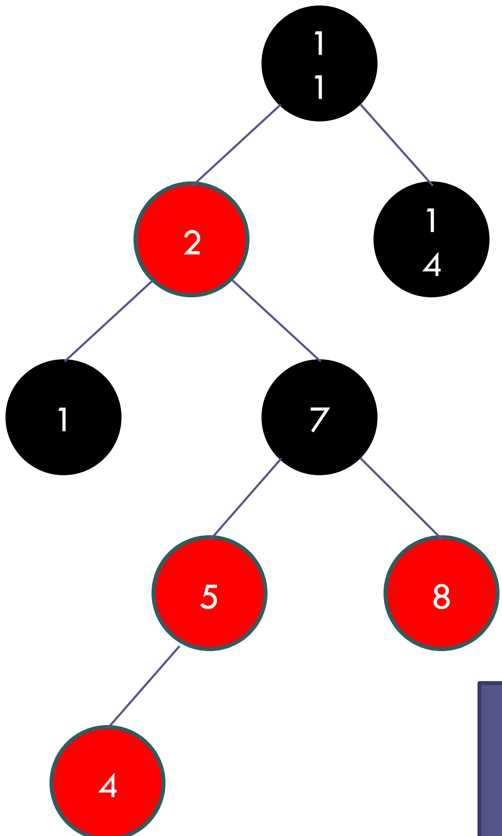
Insertion into a Red-Black Tree (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Insertion into a Red-Black Tree (cont.)



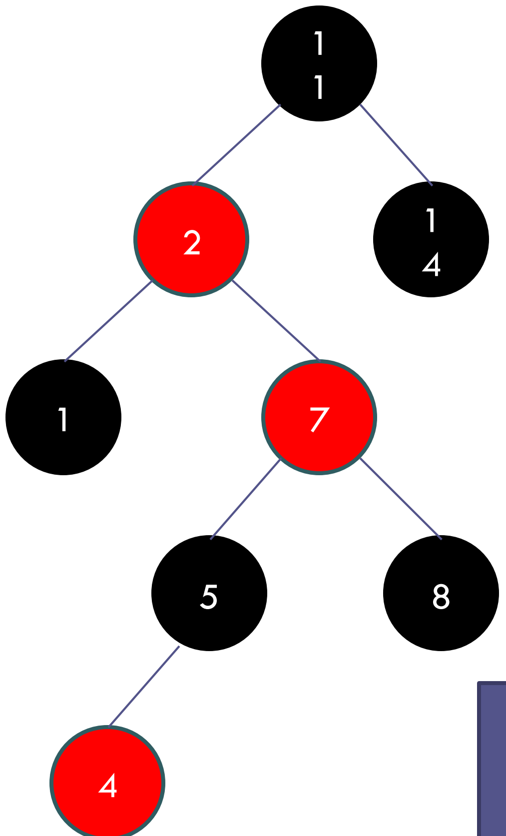
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red

Insertion into a Red-Black Tree (cont.)



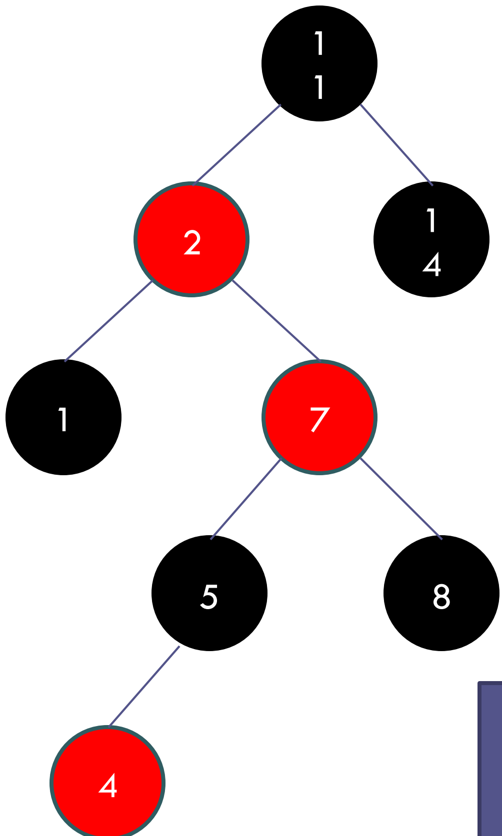
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

If a parent is red, and its sibling is also red, they can both be changed to black, and the grandparent to red

Insertion into a Red-Black Tree (cont.)

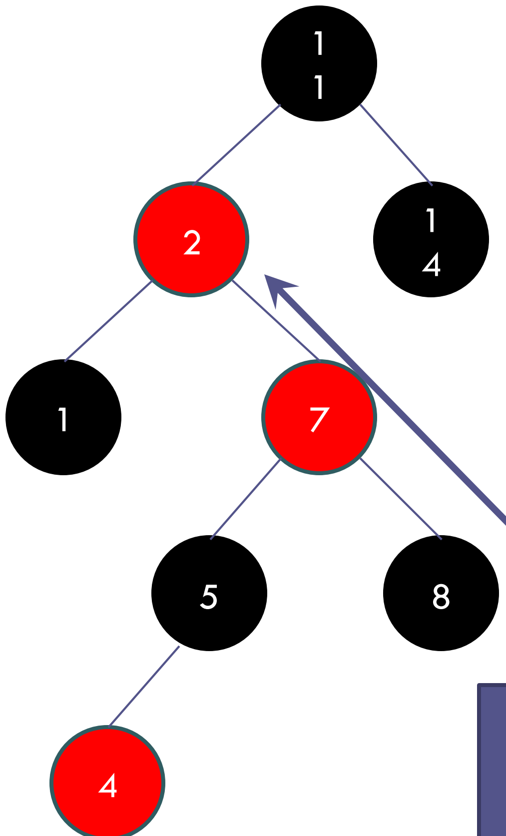


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

The problem has now shifted up the tree

Insertion into a Red-Black Tree (cont.)



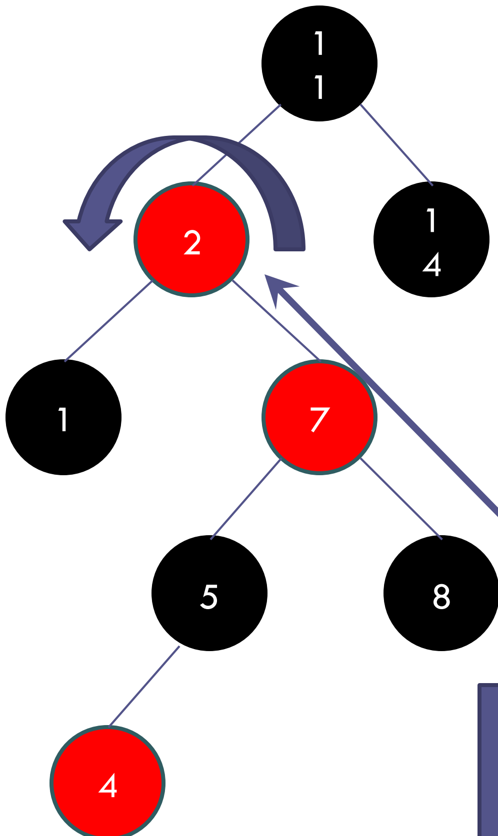
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

We cannot change 2 to black because its sibling 14 is already black (both siblings have to be red (unless there is no sibling) to do the color change

Insertion into a Red-Black Tree (cont.)



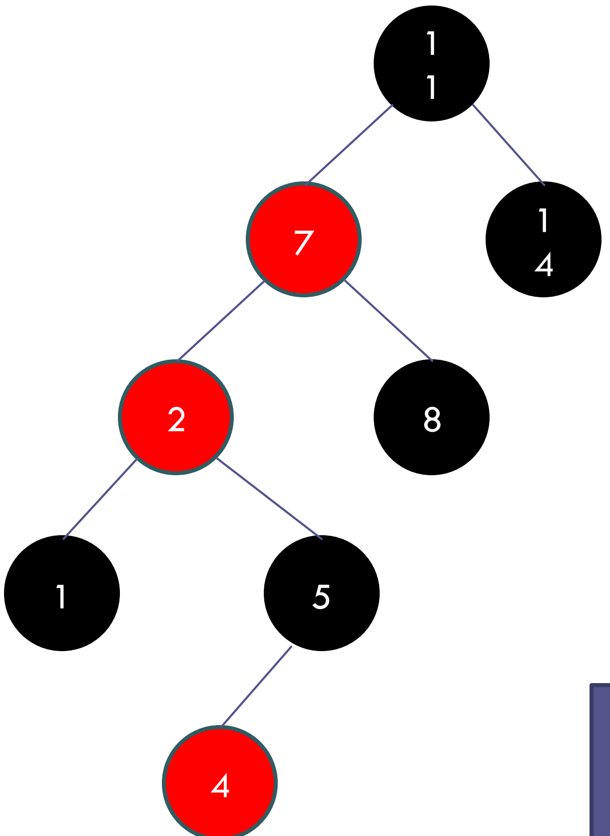
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

We need to rotate left around 2 so that the red child is on the same side of the parent as the parent is to the grandparent

Insertion into a Red-Black Tree (cont.)



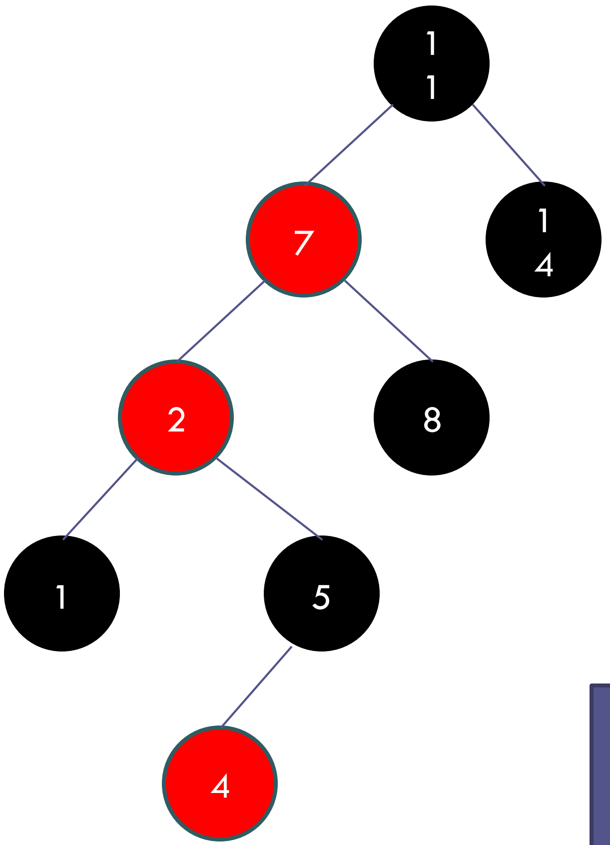
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

We need to rotate left around 2 so that the red child is on the same side of the parent as the parent is to the grandparent

Insertion into a Red-Black Tree (cont.)



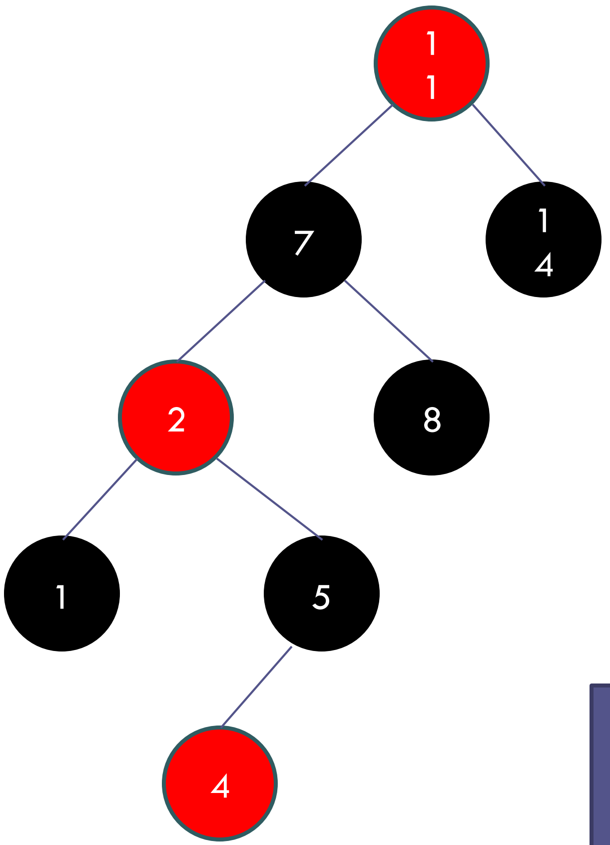
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Change colors

Insertion into a Red-Black Tree (cont.)



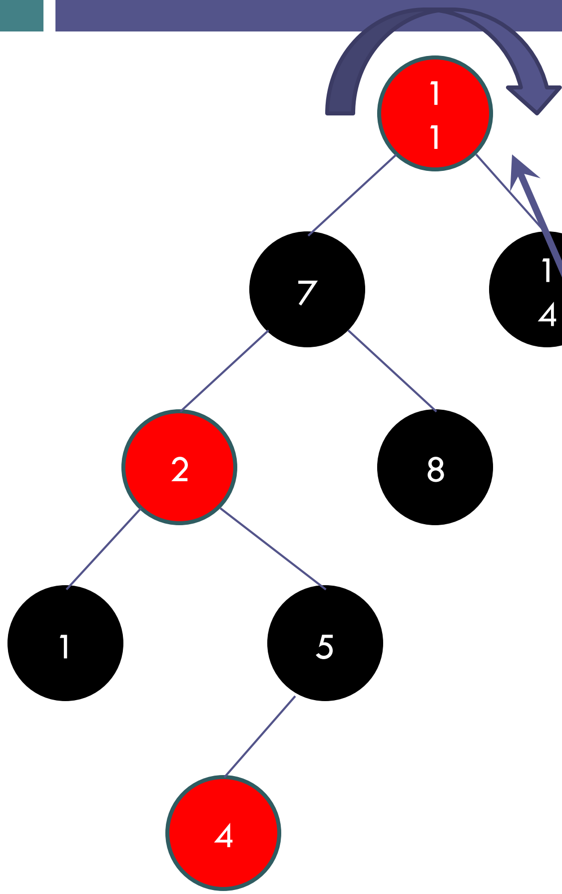
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Change colors

Insertion into a Red-Black Tree (cont.)



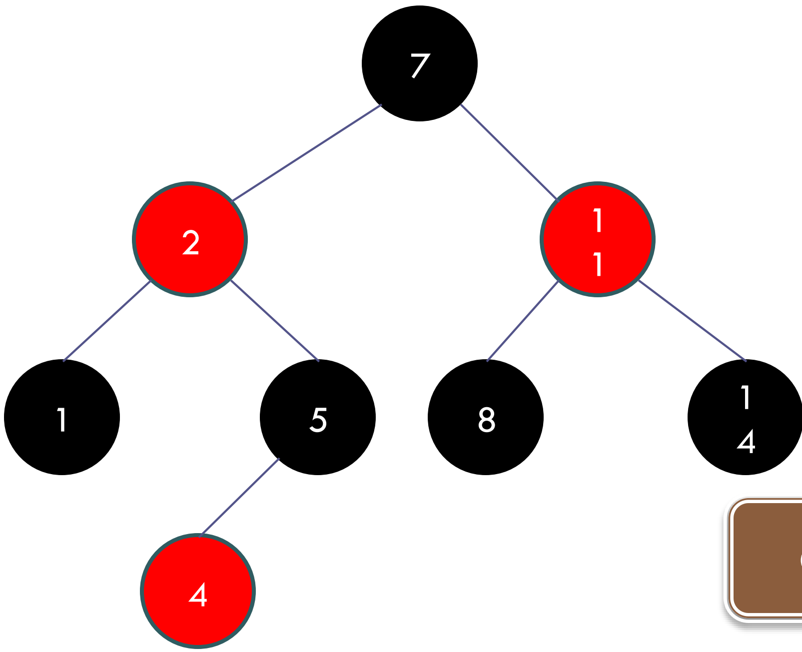
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Rotate right around 11 to restore the balance

Insertion into a Red-Black Tree (cont.)



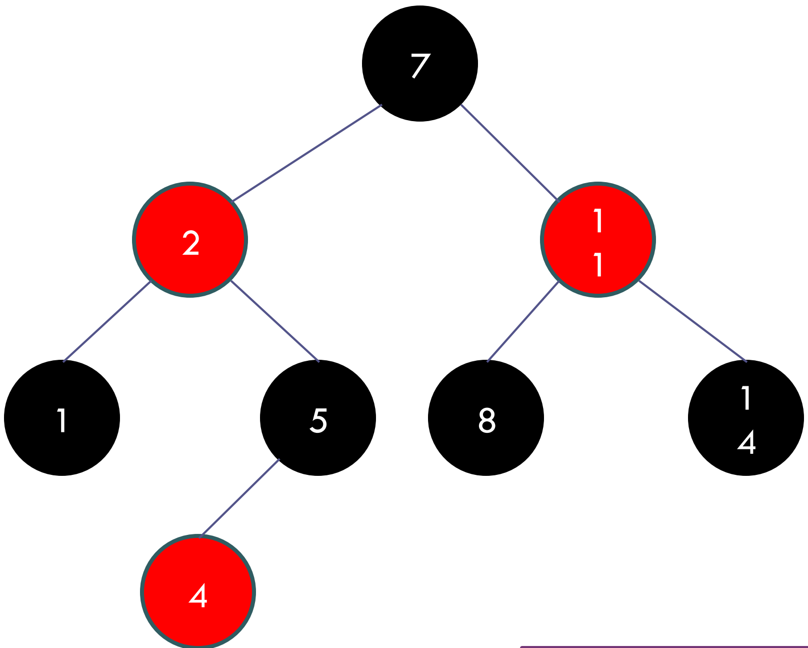
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Rotate right around 11 to restore the balance

Insertion into a Red-Black Tree (cont.)



Invariants:

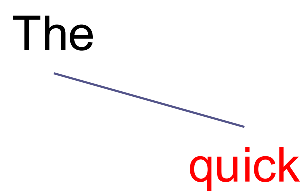
1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Balanced tree

Red-Black Tree Example

- Build a Red-Black tree for the words in "The quick brown fox jumps over the lazy dog"

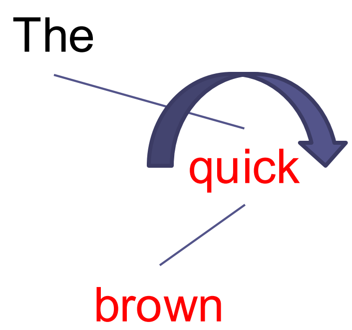
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



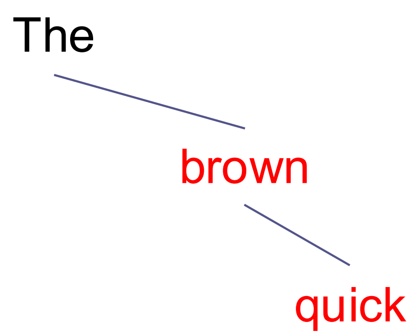
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Rotate so that the child is on the same side of its parent as its parent is to the grandparent

Red-Black Tree Example (cont.)



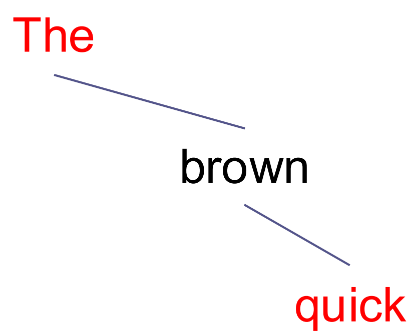
CASE 3

Change colors

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



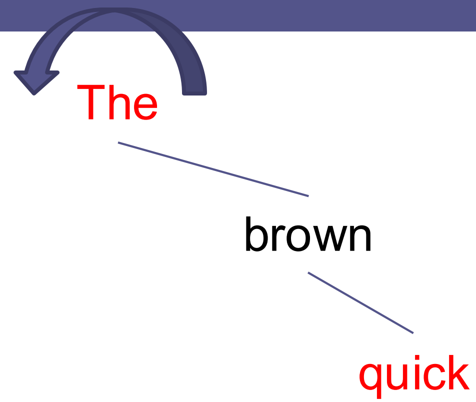
CASE 3

Change colors

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



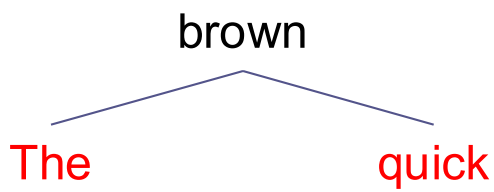
CASE 3

Rotate left

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

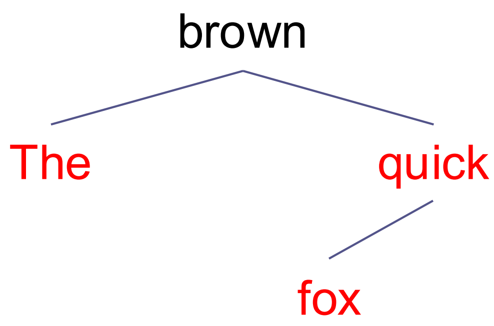
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

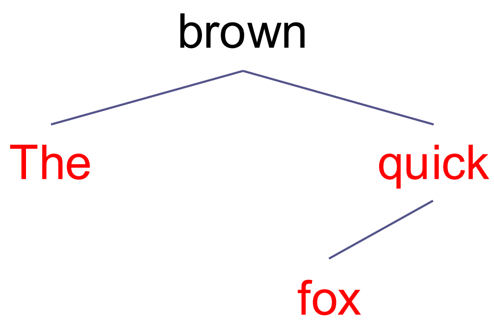
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



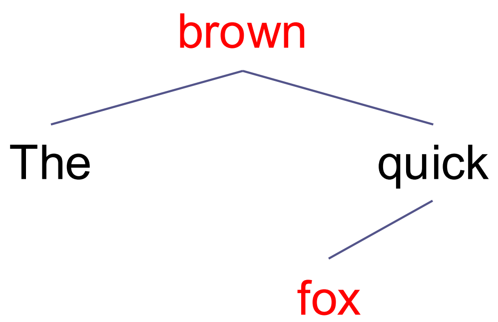
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

fox's parent and its parent's sibling are both red. Change colors.

Red-Black Tree Example (cont.)



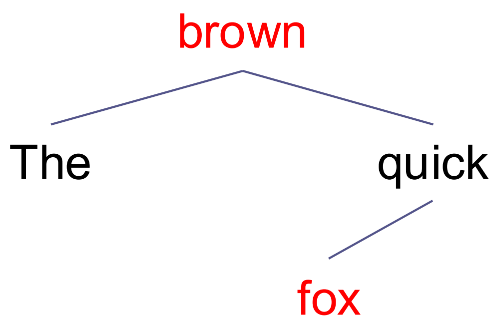
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

fox's parent and its parent's sibling are both red. Change colors.

Red-Black Tree Example (cont.)



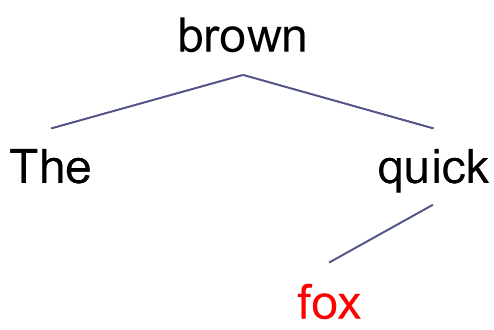
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

We can change *brown*'s color to black and not violate #4

Red-Black Tree Example (cont.)



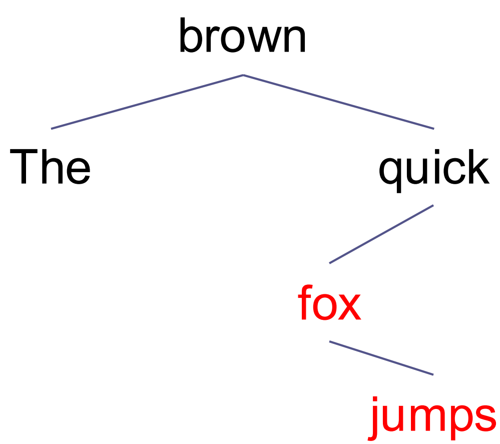
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 1

We can change *brown*'s color to black and not violate #4

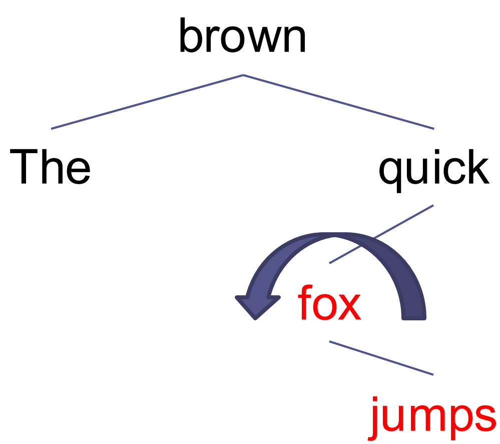
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



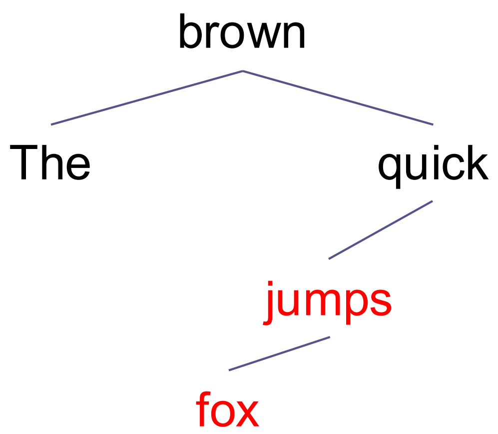
CASE 3

Rotate so that red child is on same side of its parent as its parent is to the grandparent

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



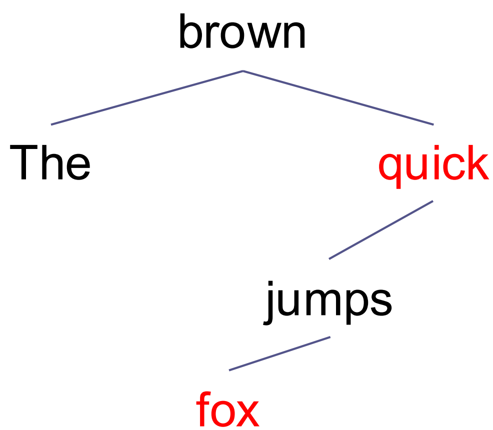
CASE 3

Change *fox*'s parent and grandparent colors

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



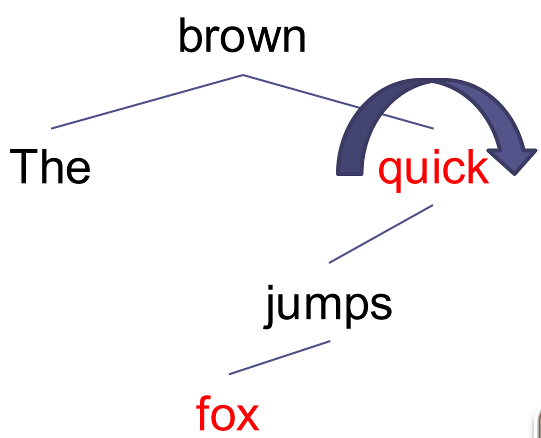
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Change *fox*'s parent and grandparent colors

Red-Black Tree Example (cont.)



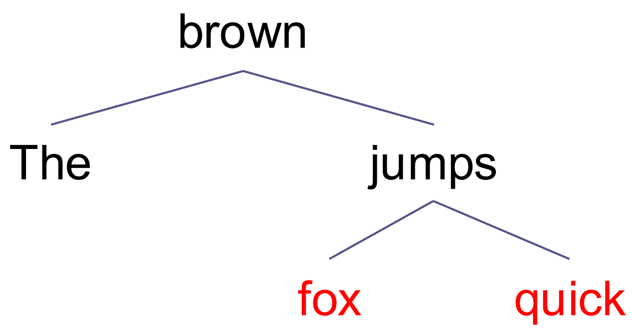
CASE 3

Rotate right about *quick*

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



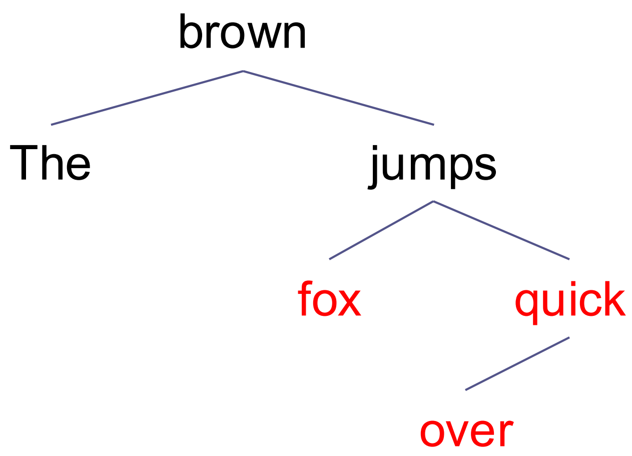
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 3

Rotate right about *quick*

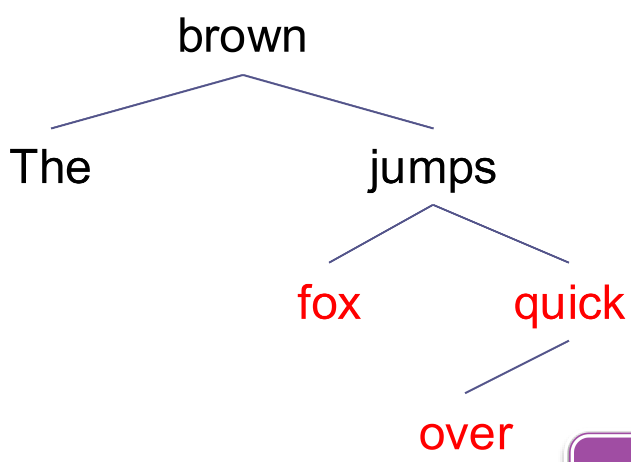
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



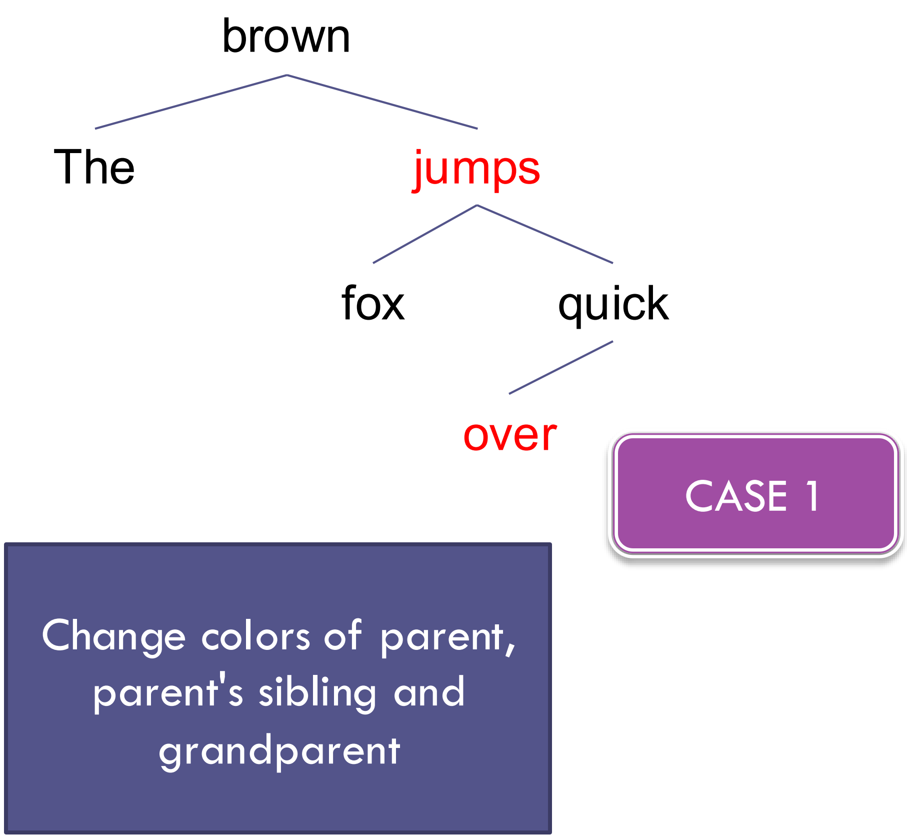
CASE 1

Change colors of parent, parent's sibling and grandparent

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

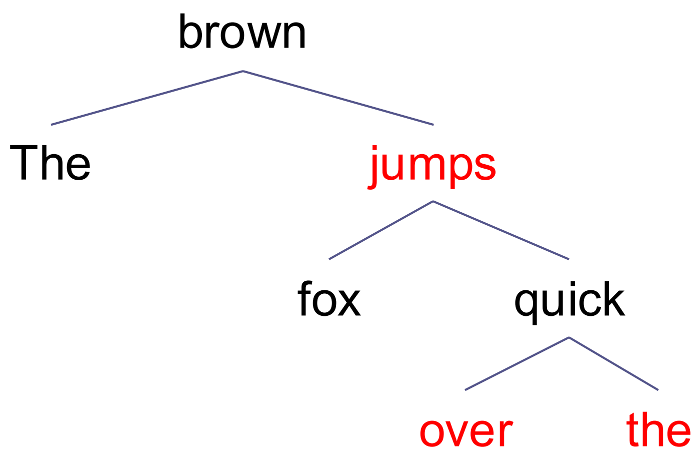
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)

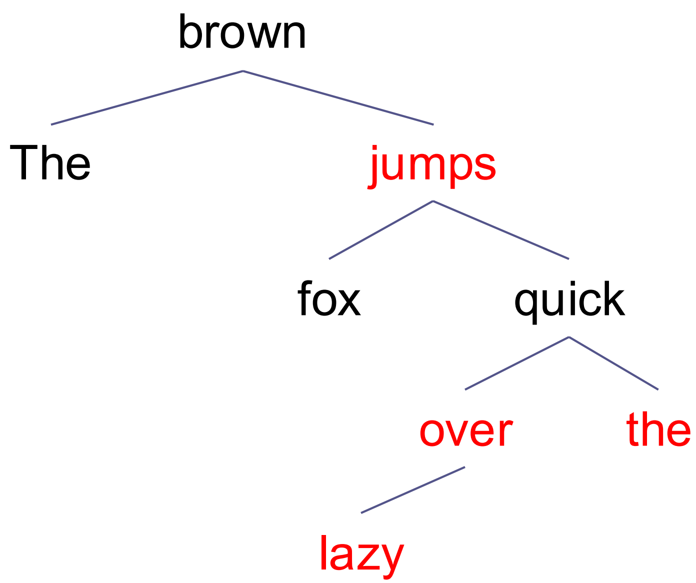


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

No changes needed

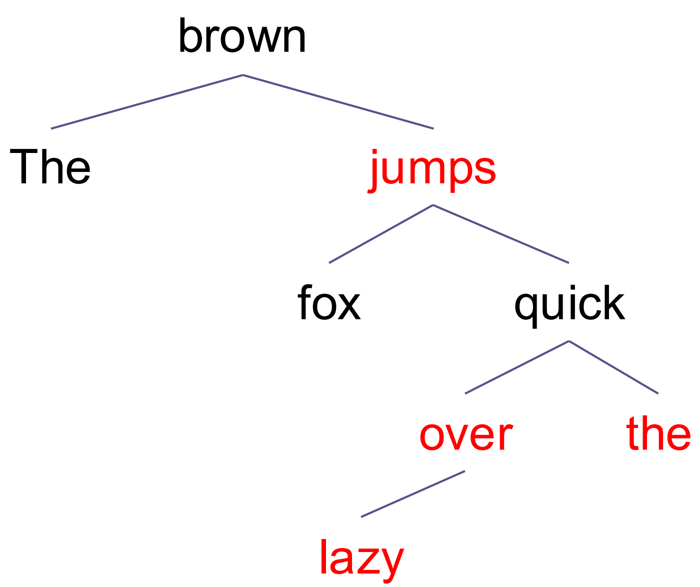
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)



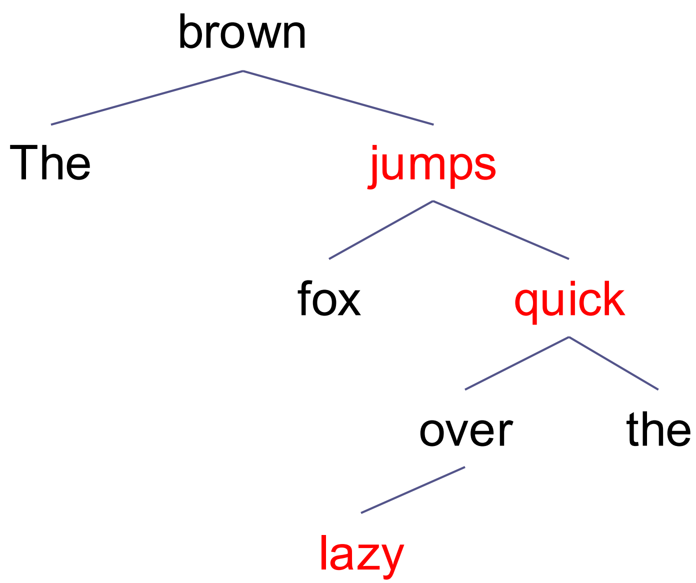
Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Because *over* and *the* are both red, change parent, parent's sibling and grandparent colors

CASE 1

Red-Black Tree Example (cont.)

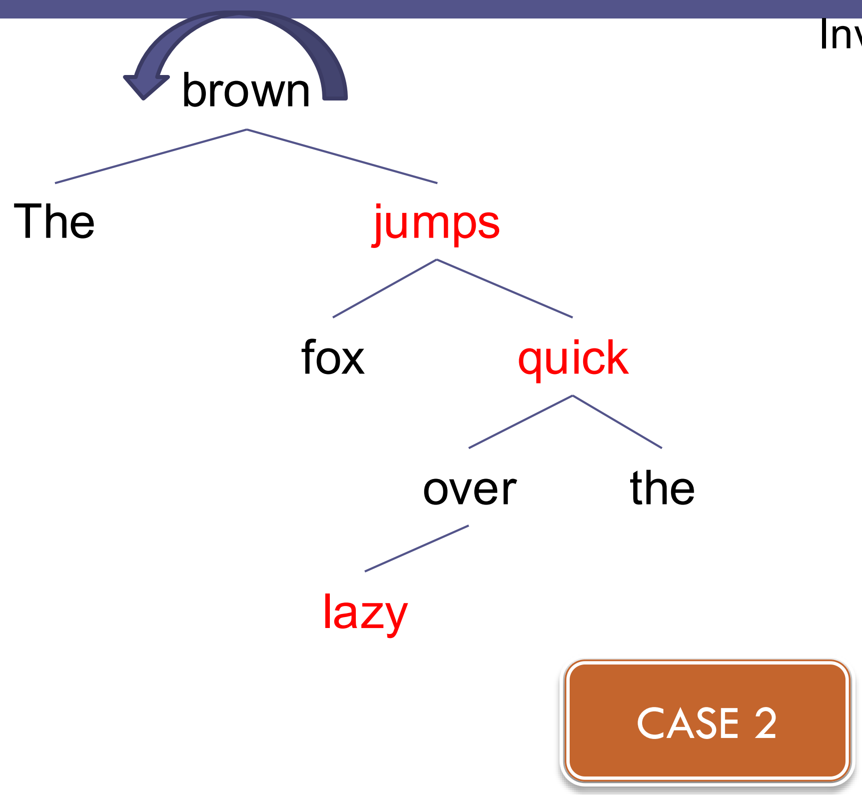


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 2

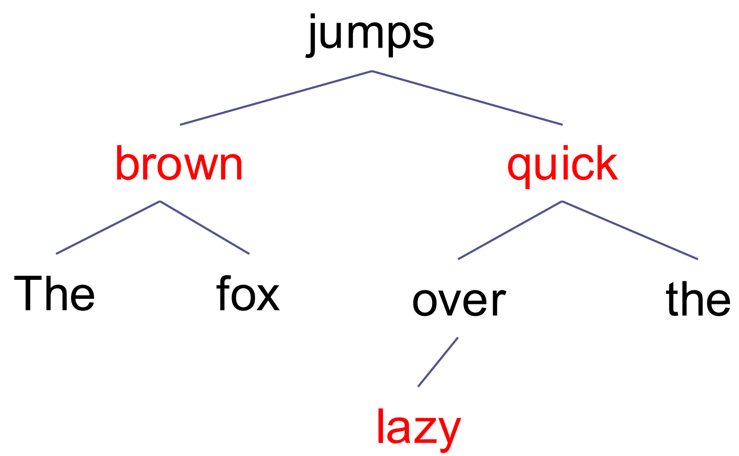
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)

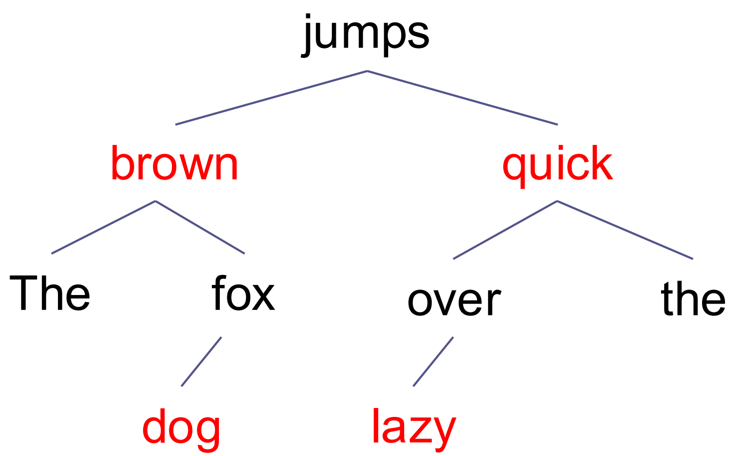


Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

CASE 2

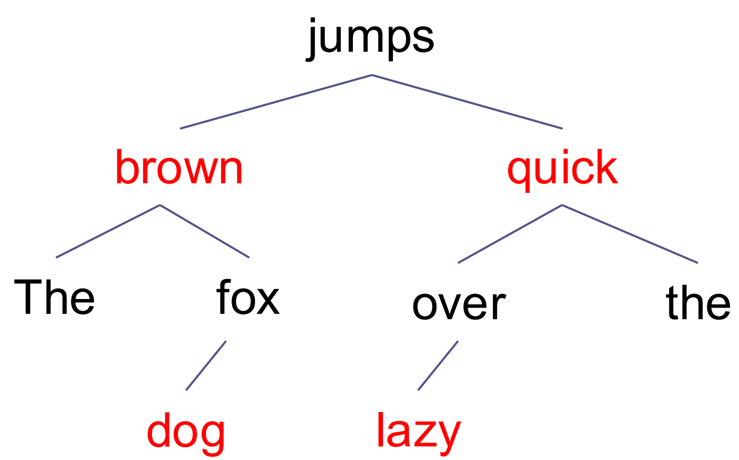
Red-Black Tree Example (cont.)



Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Red-Black Tree Example (cont.)

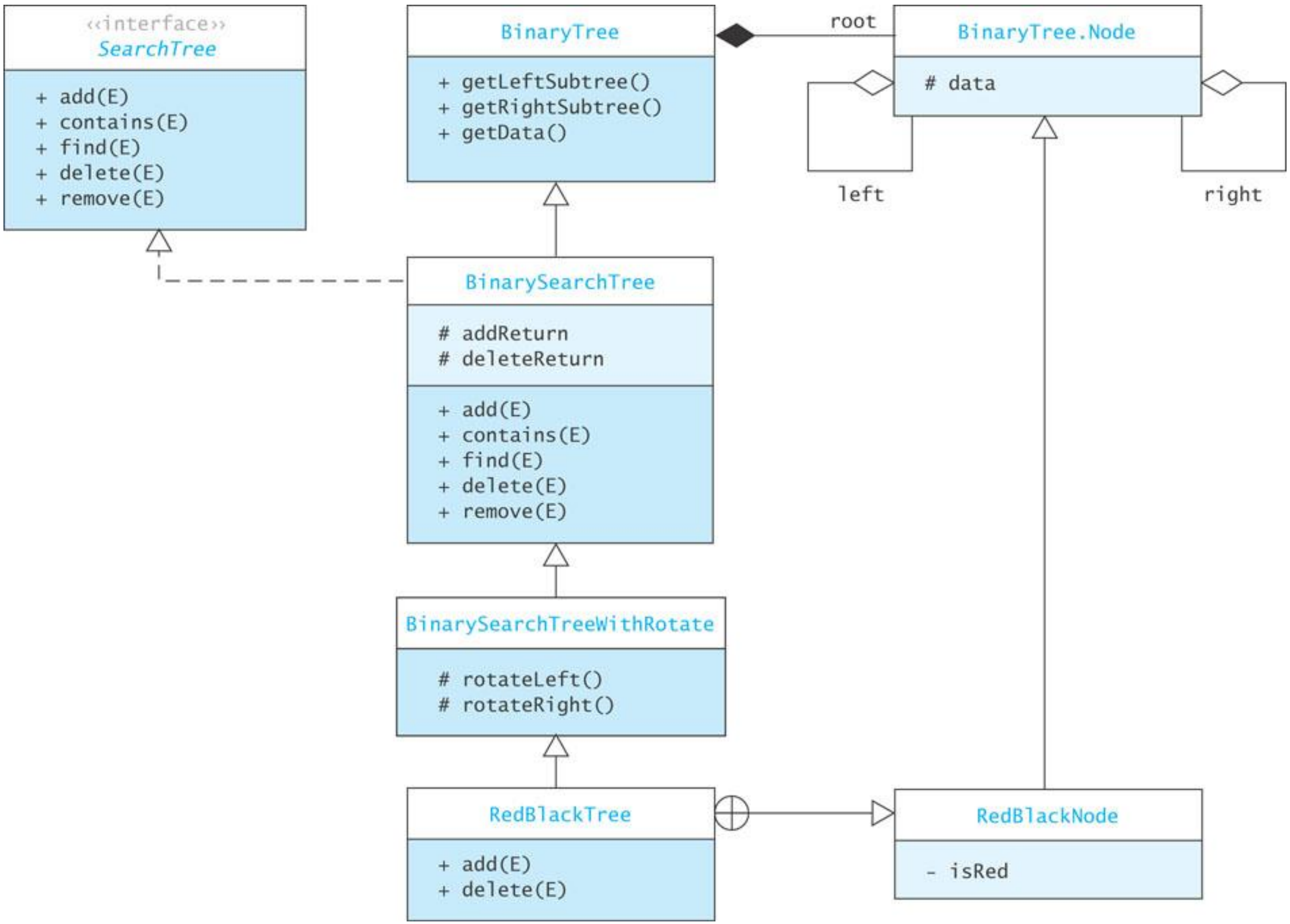


Balanced tree

Invariants:

1. A node is either red or black
2. The root is always black
3. A red node always has black children (a `null` reference is considered to refer to a black node)
4. The number of black nodes in any path from the root to a leaf is the same

Implementation of a Red-Black Tree Class



Implementation of a Red-Black Tree Class (cont.)

- Listing 9.4 (`RedBlackTree.java`, page 497)

Algorithm for Red-Black Tree Insertion

- The insertion algorithm can be implemented with a data structure that has a reference to the parent of each node
- The following algorithm detects the need for fix-ups from the grandparent level
- Also, whenever a black node with two children is detected on the way down the tree, it is changed to red and the children are changed to black; any resulting problems can be fixed on the way back up

Algorithm for Red-Black Tree Insertion (cont.)

Algorithm for Red-Black Tree Insertion

```
1.  if the root is null
2.      Insert a new Red-Black node and color it black.
3.      Return true.
4.  else if the item is equal to root.data
5.      The item is already in the tree; return false.
6.  else if the item is less than root.data
7.      if the left subtree is null
8.          Insert a new Red-Black node as the left subtree and color it red.
9.          Return true.
10.     else
11.         if both the left child and the right child are red
12.             Change the color of the children to black and change local
                root to red.
13.         Recursively insert the item into the left subtree.
14.         if the left child is now red
15.             if the left grandchild is now red (grandchild is an "out-
                side" node)
16.                 Change the color of the left child to black and
                    change the local root to red.
17.                 Rotate the local root right.
18.             else if the right grandchild is now red (grandchild is an
                "inside" node)
19.                 Rotate the left child left.
20.                 Change the color of the left child to black and
                    change the local root to red.
21.                 Rotate the local root right.
22. else
23.     Item is greater than root.data; process is symmetric and is left as an
        exercise.
24. if the local root is the root of the tree
25.     Force its color to be black.
```

add Starter Method

```
public boolean add(E item) {
    if (root == null) {
        root = new RedBlackNode<E>(item);
        ((RedBlackNode<E>) root).isRed = false; // root is black.
        return true;
    }
. . .

else {
    root = add((RedBlackNode<E>) root, item);
    ((RedBlackNode<E>) root).isRed = false; // root is always black.
    return addReturn;
}
```

The Recursive add Method

```
private Node<E> add(RedBlackNode<E> localRoot, E item) {
    if (item.compareTo(localRoot.data) == 0) {
          // item already in the tree.
          addReturn = false;
          return localRoot;
    }
. . .

else if (item.compareTo(localRoot.data) < 0) {
    // item < localRoot.data.
    if (localRoot.left == null) {
          // Create new left child.
          localRoot.left = new RedBlackNode<E>(item);
          addReturn = true;
          return localRoot;
    }
. . .

else { // Need to search.
    // Check for two red children, swap colors if found.
    moveBlackDown(localRoot);
    // Recursively add on the left.
   localRoot.left = add((RedBlackNode<E>) localRoot.left, item);
. . .

// See whether the left child is now red
    if (((RedBlackNode<E>) localRoot.left).isRed) {
. . .
```

The Recursive add Method (cont.)

```
if (localRoot.left.left != null
    && ((RedBlackNode<E>) localRoot.left.left).isRed) {
    // Left-left grandchild is also red.
. . .

// Single rotation is necessary.
((RedBlackNode<E>) localRoot.left).isRed = false;
localRoot.isRed = true;
return rotateRight(localRoot);

else if (localRoot.left.right != null
          && ((RedBlackNode<E>) localRoot.left.right).isRed) {
    // Left-right grandchild is also red.
    // Double rotation is necessary.
    localRoot.left = rotateLeft(localRoot.left);
    ((RedBlackNode<E>) localRoot.left).isRed = false;
    localRoot.isRed = true;
    return rotateRight(localRoot);
}
```

Removal from a Red-Black Tree

- Remove a node only if it is a leaf or has only one child
- Otherwise, the node containing the inorder predecessor of the value being removed is removed
- If the node removed is red, nothing further is done
- If the node removed is black and has a red child, then the red child takes its place and is colored black
- If a black leaf is removed, the black height becomes unbalanced
- A programming project at the end of the chapter describes other cases

Performance of a Red-Black Tree

- The upper limit in the height for a Red-Black tree is $2 \log_2 n + 2$ which is still $O(\log n)$
- As with AVL trees, the average performance is significantly better than the worst-case performance
- Empirical studies show that the average cost of searching a Red-Black tree built from random values is $1.002 \log_2 n$
- Red-Black trees and AVL trees both give performance close to that of a complete binary tree

`TreeMap` and `TreeSet` Classes

- The Java API has a `TreeMap` class that implements a Red-Black tree
- It implements `SortedMap` so some of the methods it defines are:
  - `get`
  - `put`
  - `remove`
  - `containsKey`
- All are O(log *n*) operations
- `TreeSet` implements `SortedSet` and is an adapter of the `TreeMap` class

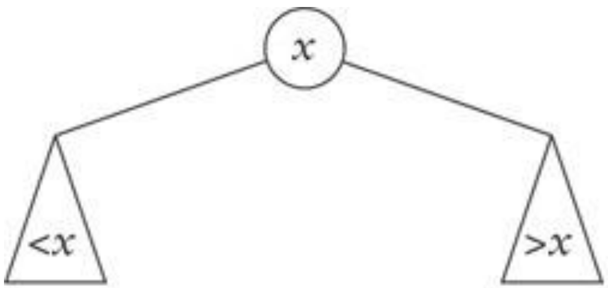
2-3 Trees

Section 9.4

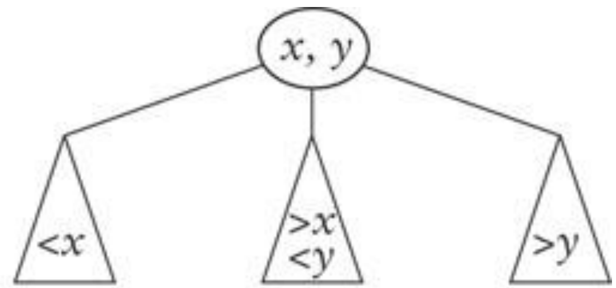
2-3 Trees

- A 2-3 tree is made up of nodes designated as either *2-nodes* or *3-nodes*
- A 2-node is the same as a binary search tree node:
  - it contains a data field and references to two child nodes
  - one child node contains data less than the node's data value
  - the other child contains data greater than the node's data value
- A 3-node
  - contains two data fields, ordered so that first is less than the second, and references to three children
    - One child contains data values less than the first data field
    - One child contains data values between the two data fields
    - One child contains data values greater than the second data field
- All the leaves of a 2-3 tree are at the lowest level

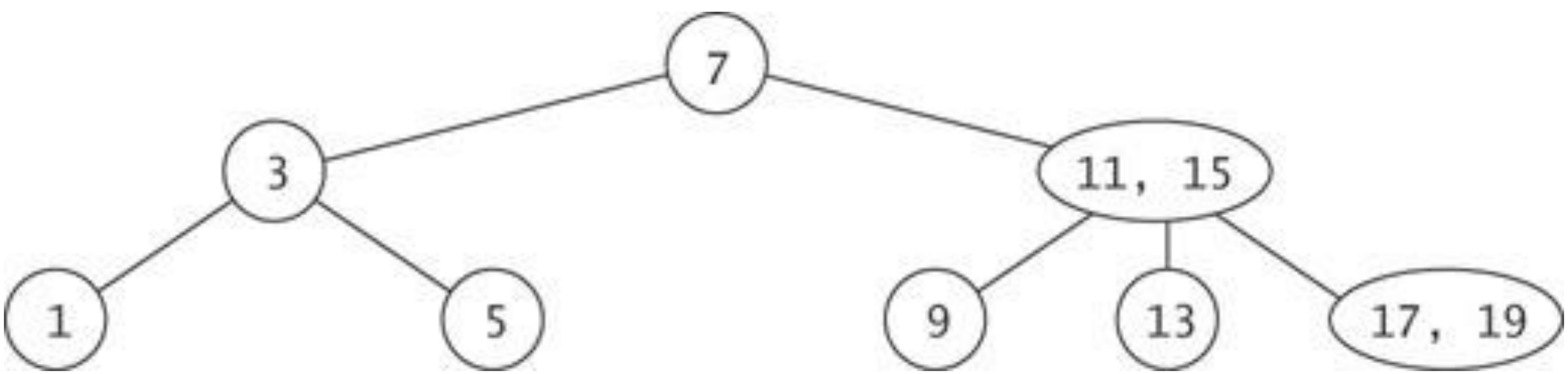
2-3 Trees (cont.)



2-node



3-node



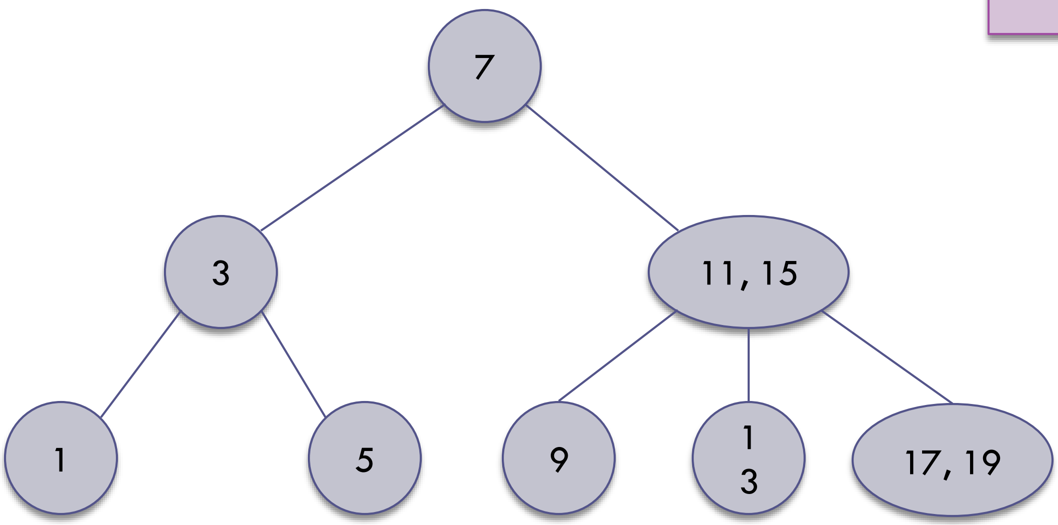
Searching a 2-3 Tree

Searching a 2-3 Tree

Searching a 2-3 tree is very similar to searching a binary search tree.

```
if the local root is null
    Return null; the item is not in the tree.
else if this is a 2-node
    if the item is equal to the data1 field
        Return the data1 field.
    else if the item is less than the data1 field
        Recursively search the left subtree.
    else
        Recursively search the right subtree.
else  // This is a 3-node
    if the item is equal to the data1 field
        Return the data1 field.
    else if the item is equal to the data2 field
        Return the data2 field.
    else if the item is less than the data1 field
        Recursively search the left subtree.
    else if the item is less than the data2 field
        Recursively search the middle subtree.
    else
        Recursively search the right subtree.
```

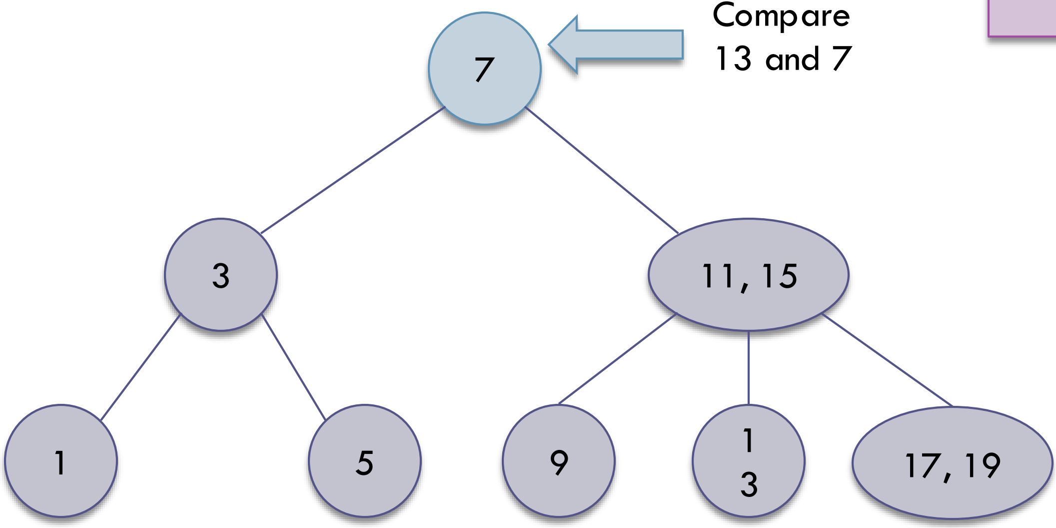
Searching a 2-3 Tree (cont.)



To search for 13

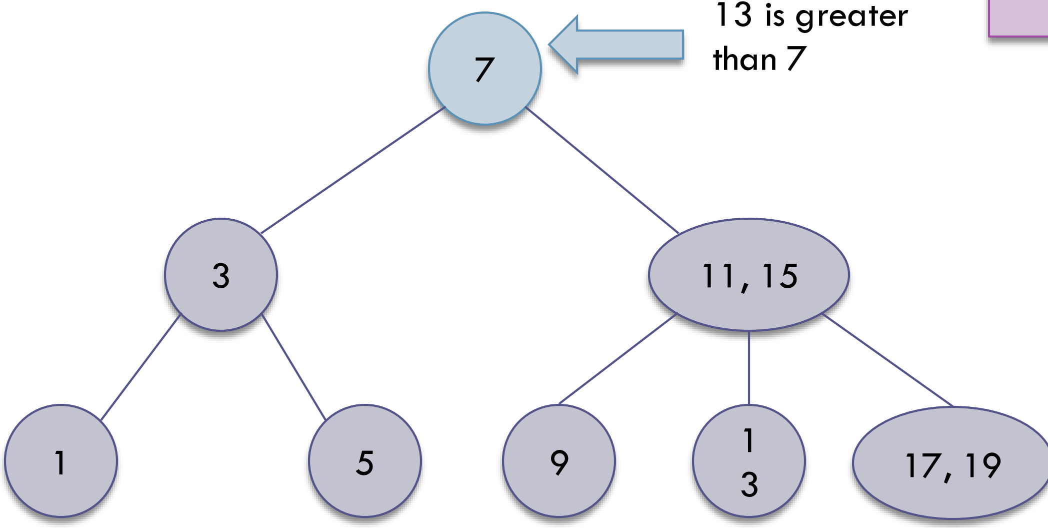
Searching a 2-3 Tree (cont.)

To search for 13

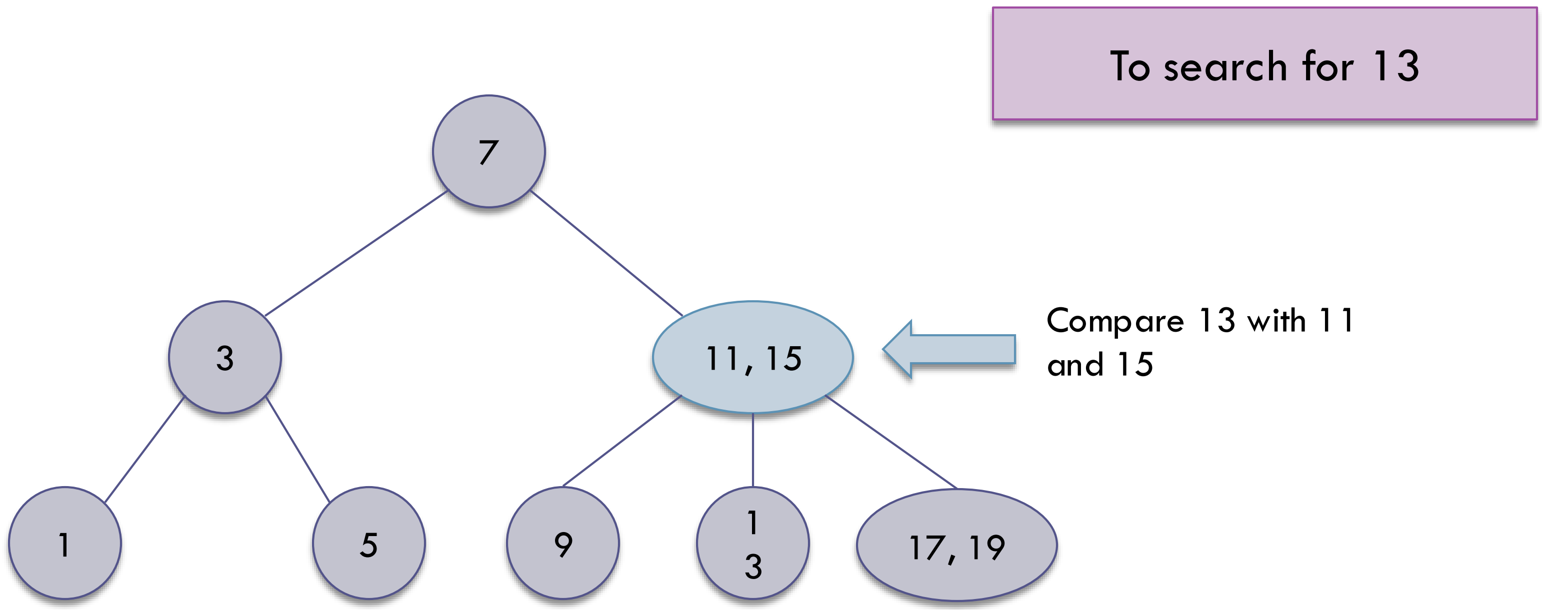


Searching a 2-3 Tree (cont.)

To search for 13

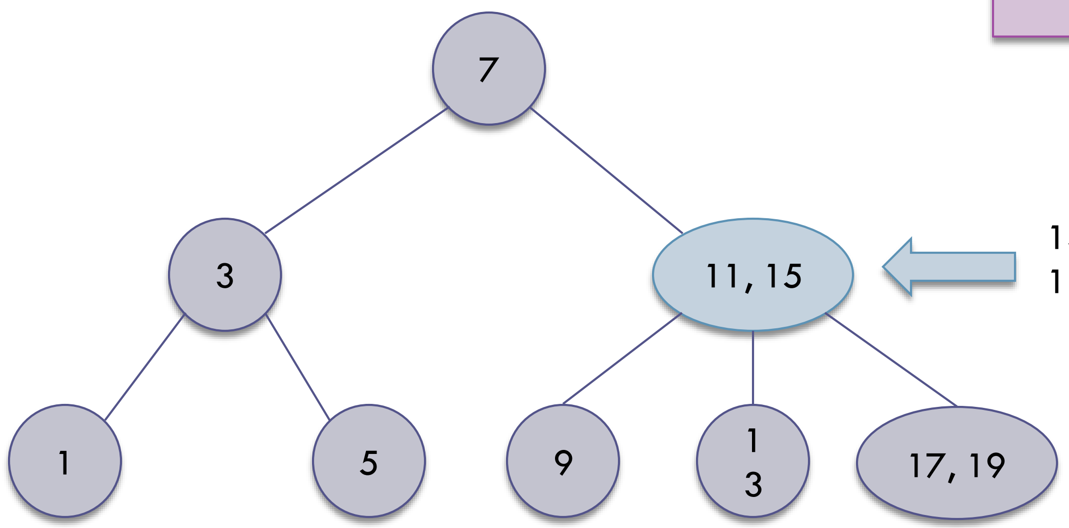


Searching a 2-3 Tree (cont.)



Searching a 2-3 Tree (cont.)

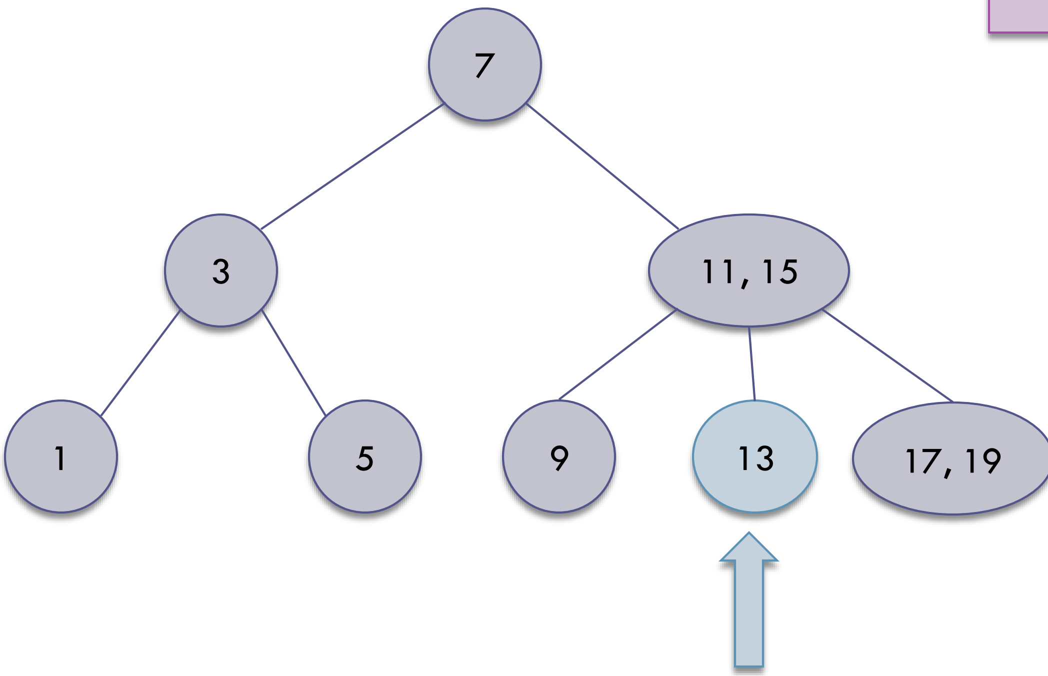
To search for 13



13 is in between 11 and 15
11 < 13 < 15

Searching a 2-3 Tree (cont.)

To search for 13

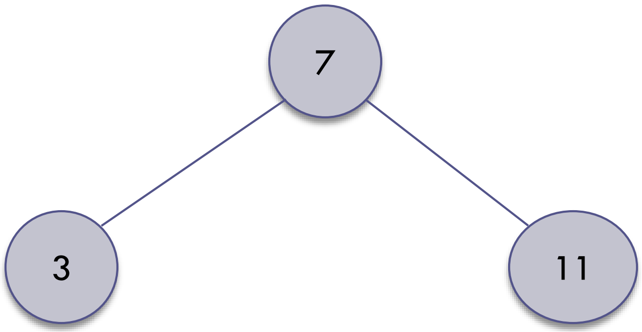


13 is in the middle child

Inserting an Item into a 2-3 Tree

- A 2-3 tree maintains balance by being built from the bottom up, not the top down
- Instead of hanging a new node onto a leaf, we insert the new node into a leaf

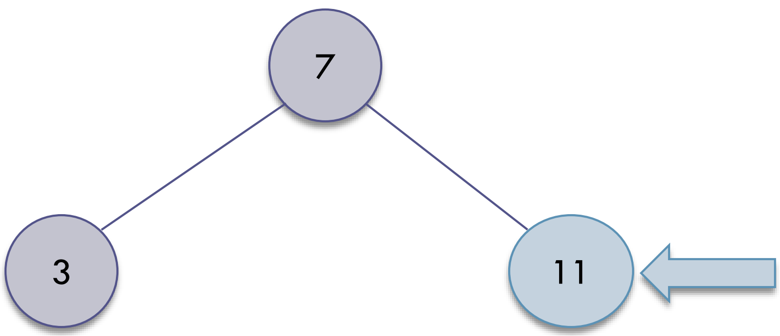
Inserting an Item into a 2-3 Tree (cont.)



Insert 15

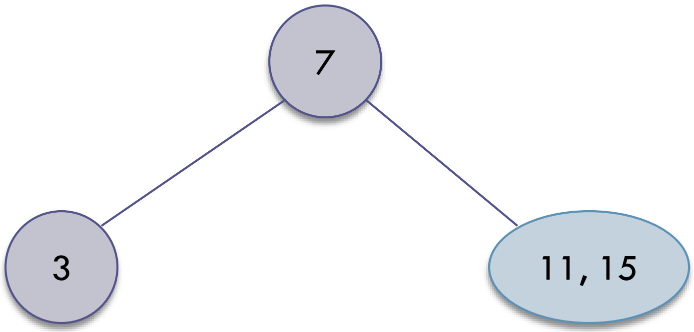
Inserting an Item into a 2-3 Tree (cont.)

Insert 15



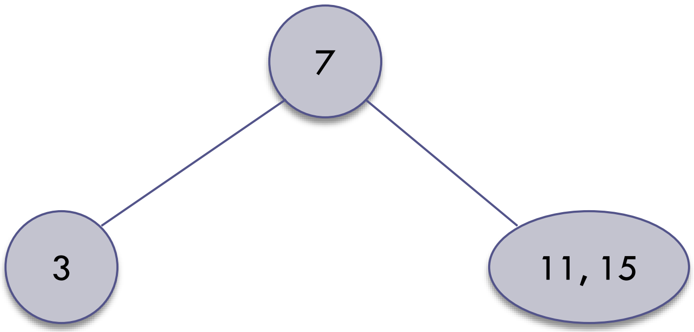
Because this node is a 2-node, we insert directly into the node creating a 3-node

Inserting an Item into a 2-3 Tree (cont.)



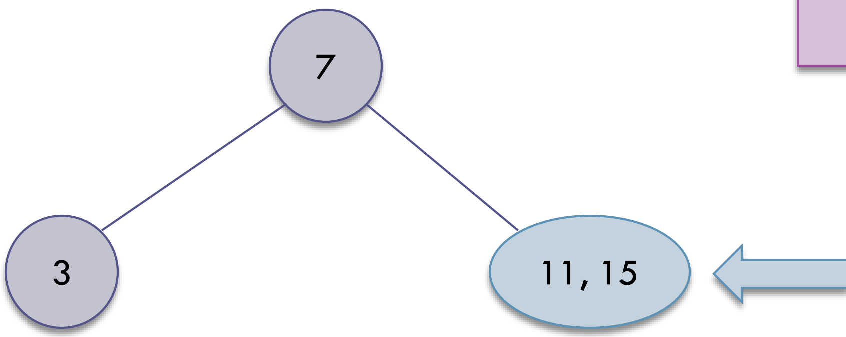
Insert 15

Inserting an Item into a 2-3 Tree (cont.)



Insert 17

Inserting an Item into a 2-3 Tree (cont.)

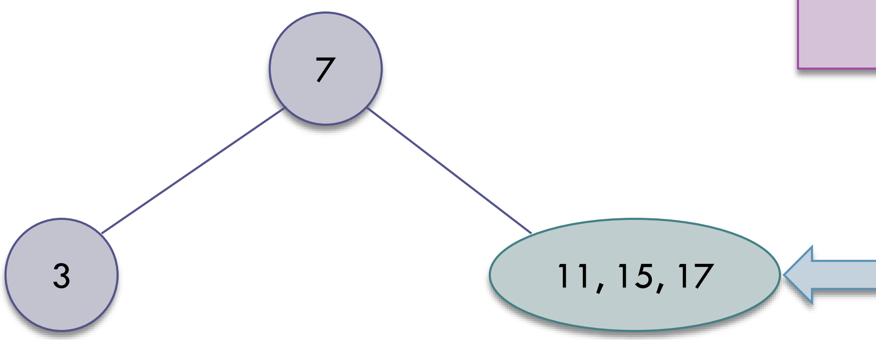


Insert 17

Because we insert into leaves, 17 is virtually inserted into this node

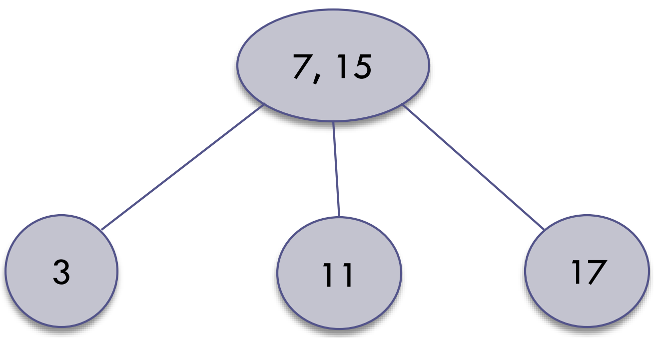
Inserting an Item into a 2-3 Tree (cont.)

Insert 17



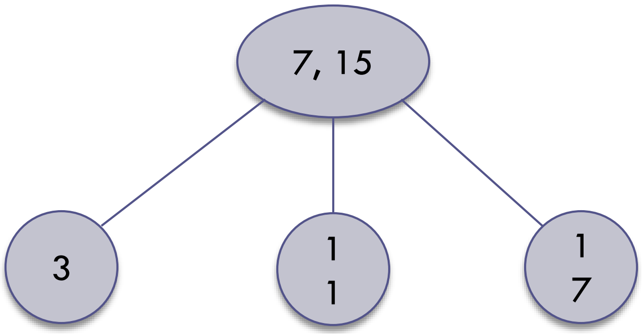
Because a node can't store three values, the middle value propagates up to the 2-node parent and this leaf node splits into two new 2-nodes

Inserting an Item into a 2-3 Tree (cont.)



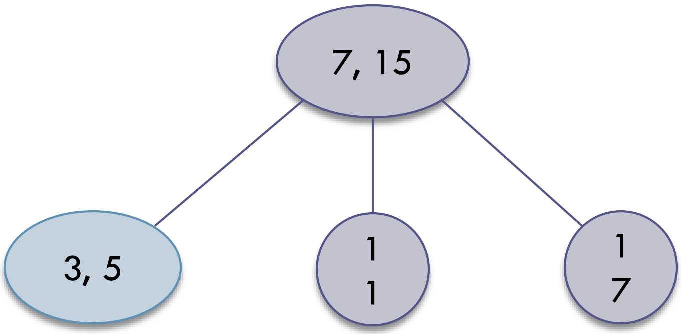
Insert 17

Inserting an Item into a 2-3 Tree (cont.)



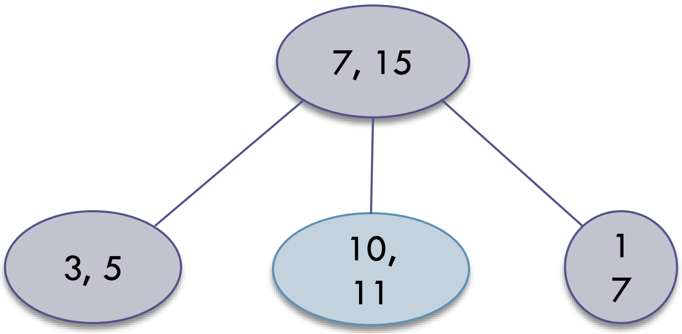
Insert 5, 10, 20

Inserting an Item into a 2-3 Tree (cont.)



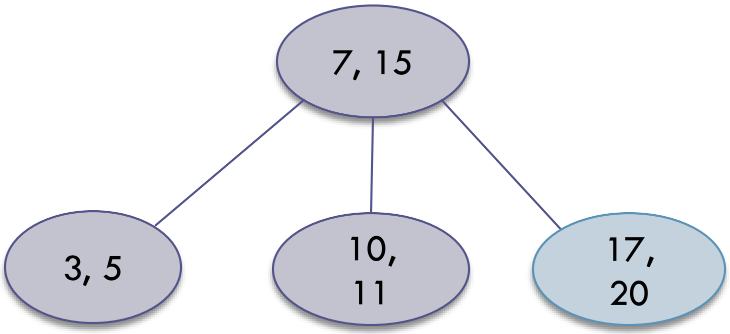
Insert 5, 10, 20

Inserting an Item into a 2-3 Tree (cont.)



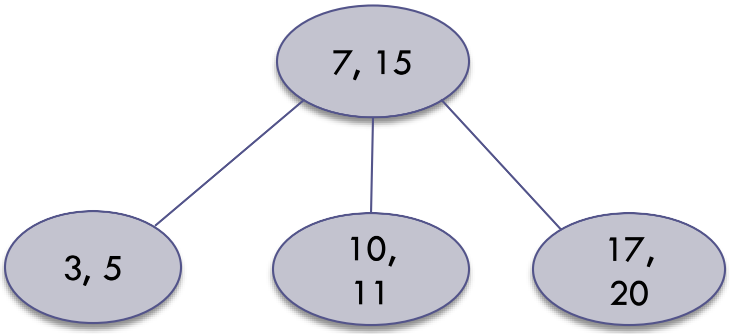
Insert 5, 10, 20

Inserting an Item into a 2-3 Tree (cont.)



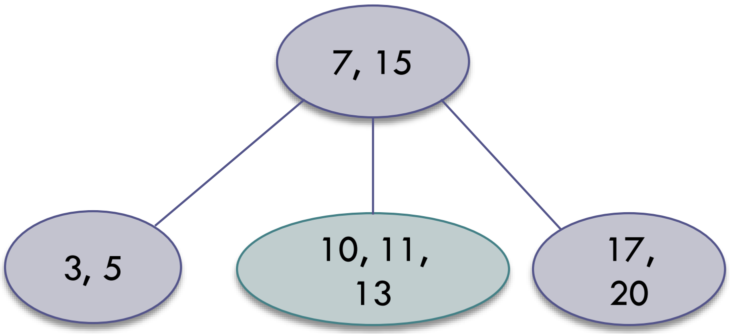
Insert 5, 10, 20

Inserting an Item into a 2-3 Tree (cont.)



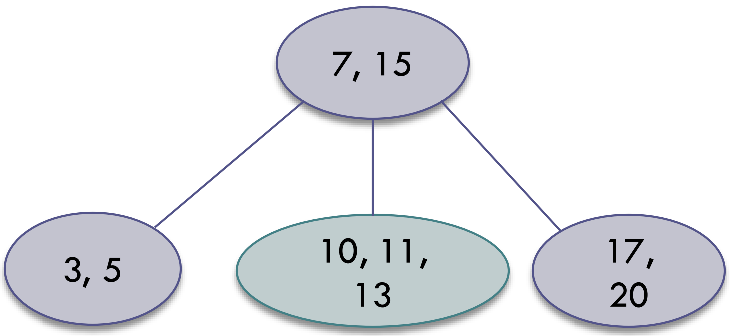
Insert 13

Inserting an Item into a 2-3 Tree (cont.)



Insert 13

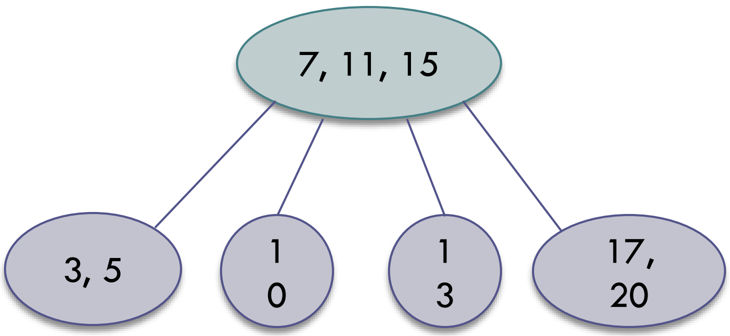
Inserting an Item into a 2-3 Tree (cont.)



Insert 13

Since a node with three values is a virtual node, move the middle value up and split the remaining values into two nodes

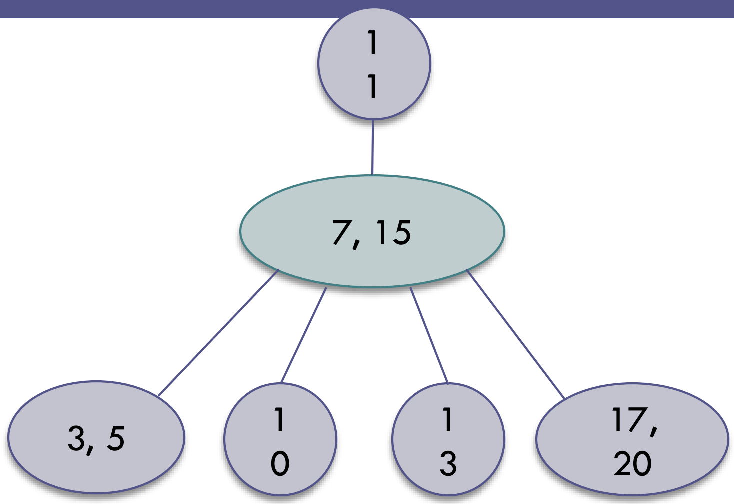
Inserting an Item into a 2-3 Tree (cont.)



Insert 13

Repeat

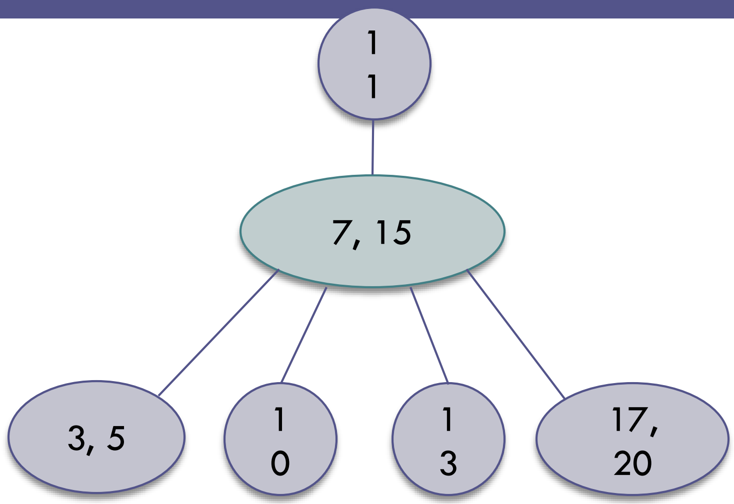
Inserting an Item into a 2-3 Tree (cont.)



Insert 13

Move the middle value up

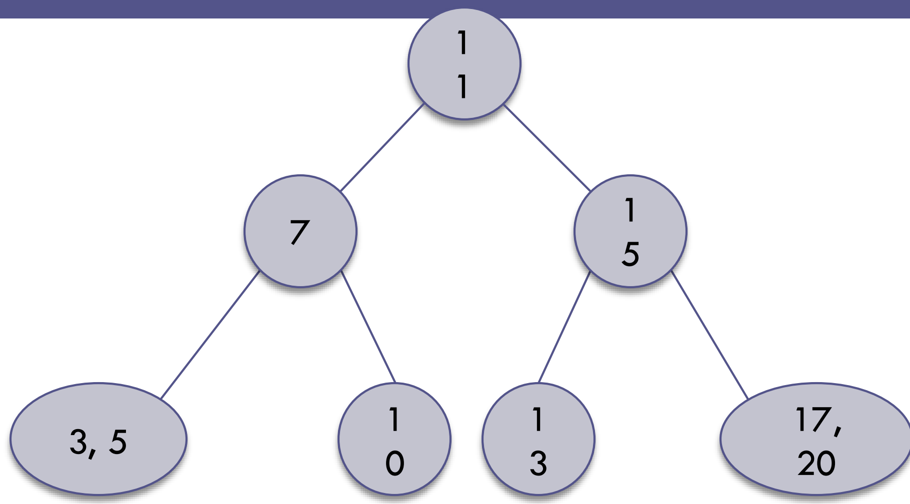
Inserting an Item into a 2-3 Tree (cont.)



Insert 13

Split the remaining values into two nodes

Inserting an Item into a 2-3 Tree (cont.)



Insert 13

Split the remaining values into two nodes

Algorithm for Insertion into a 2-3 Tree

Algorithm for Insertion

```
1.  if the root is null
2.      Create a new 2-node that contains the new item.
3.  else if the item is in the local root
4.      Return false.
5.  else if the local root is a leaf
6.      if the local root is a 2-node
7.          Expand the 2-node to a 3-node and insert the item.
8.      else
9.          Split the 3-node (creating two 2-nodes) and pass the new parent
            back up the recursion chain.
10. else
11.     if the item is less than the smaller item in the local root
12.         Recursively insert into the left child.
13.     else if the local root is a 2-node
14.         Recursively insert into the right child.
15.     else if the item is less than the larger item in the local root
16.         Recursively insert into the middle child.
17.     else
18.         Recursively insert into the right child.
19. if a new parent was passed up from the previous level of recursion
20.     if the new parent will be the tree root
21.         Create a 2-node whose data item is the passed-up parent, left
            child is the old root, and right child is the passed-up child. This
            2-node becomes the new root.
22.     else
23.         Recursively insert the new parent at the local root.
24. Return true.
```

Insertion Example

- Create a 2-3 tree using the words "The quick brown fox jumps over the lazy dog"

Insertion Example (cont.)

The

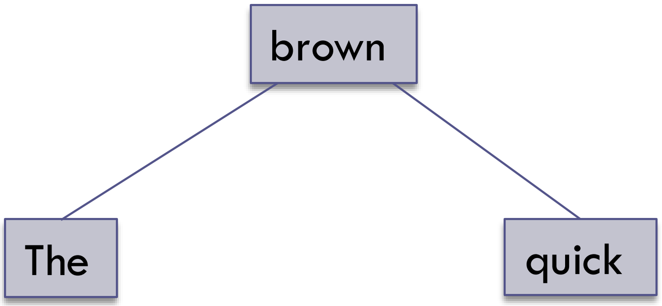
Insertion Example (cont.)

The, quick

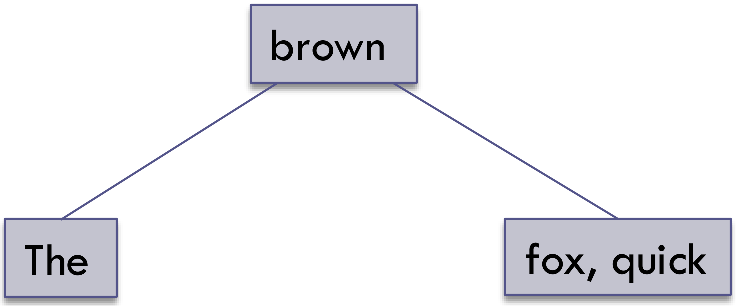
Insertion Example (cont.)

The, brown, quick

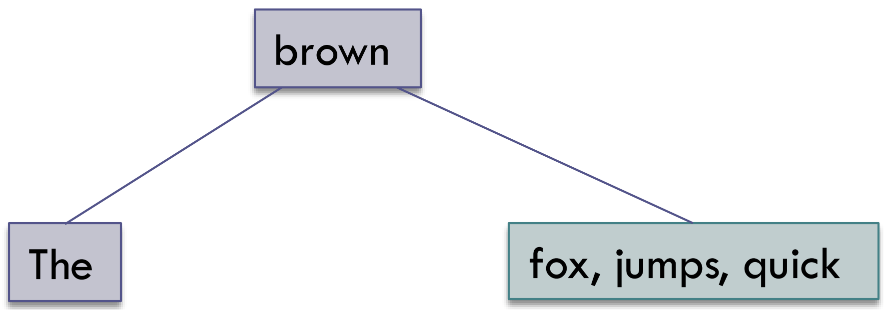
Insertion Example (cont.)



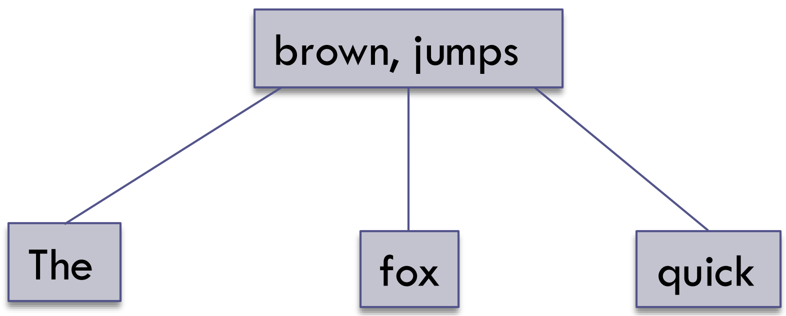
Insertion Example (cont.)



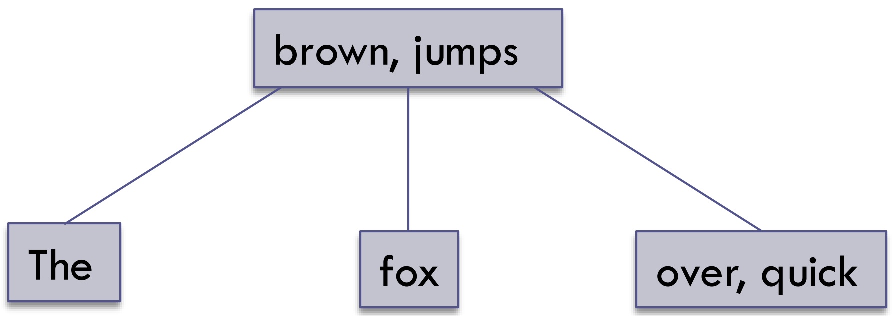
Insertion Example (cont.)



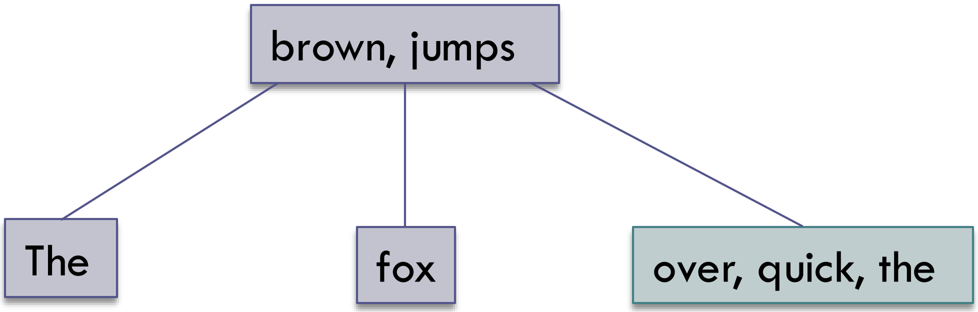
Insertion Example (cont.)



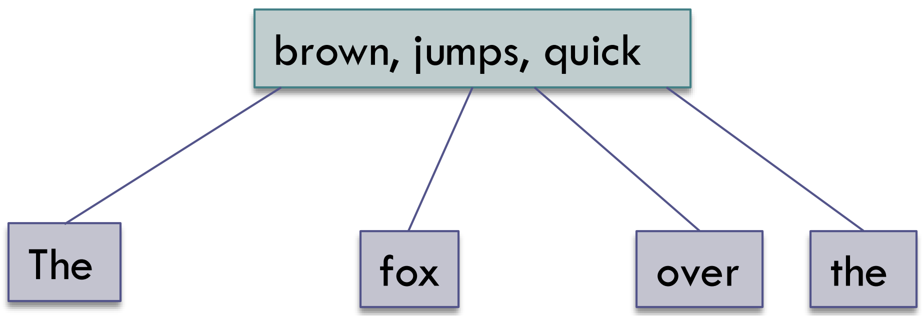
Insertion Example (cont.)



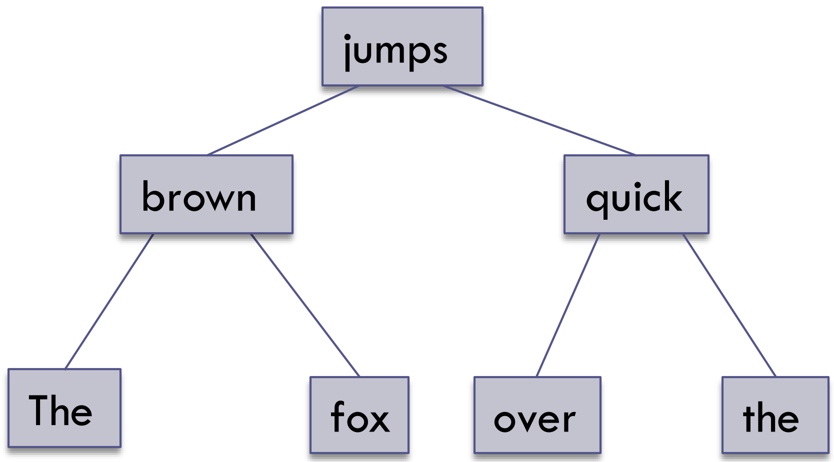
Insertion Example (cont.)



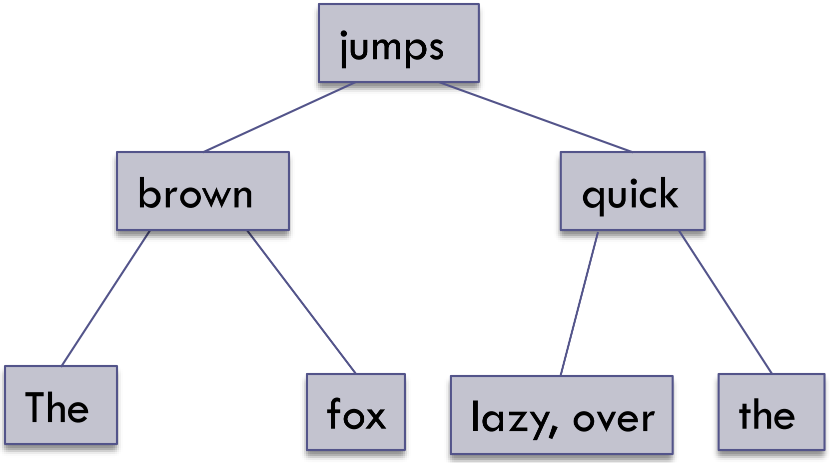
Insertion Example (cont.)



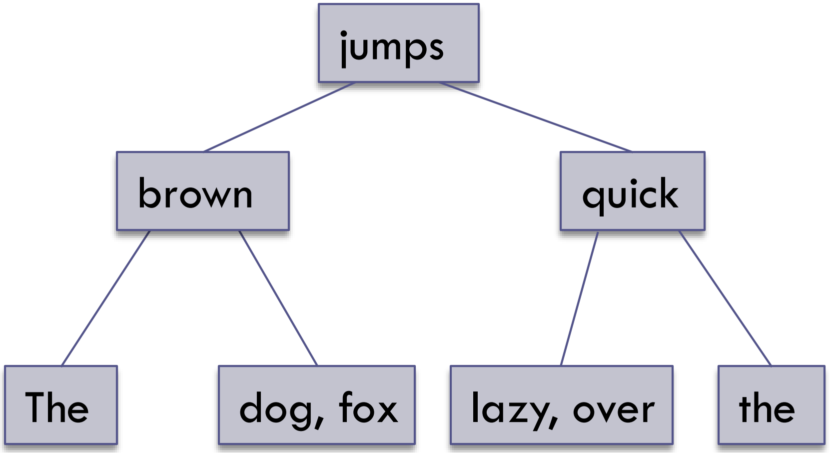
Insertion Example (cont.)



Insertion Example (cont.)



Insertion Example (cont.)



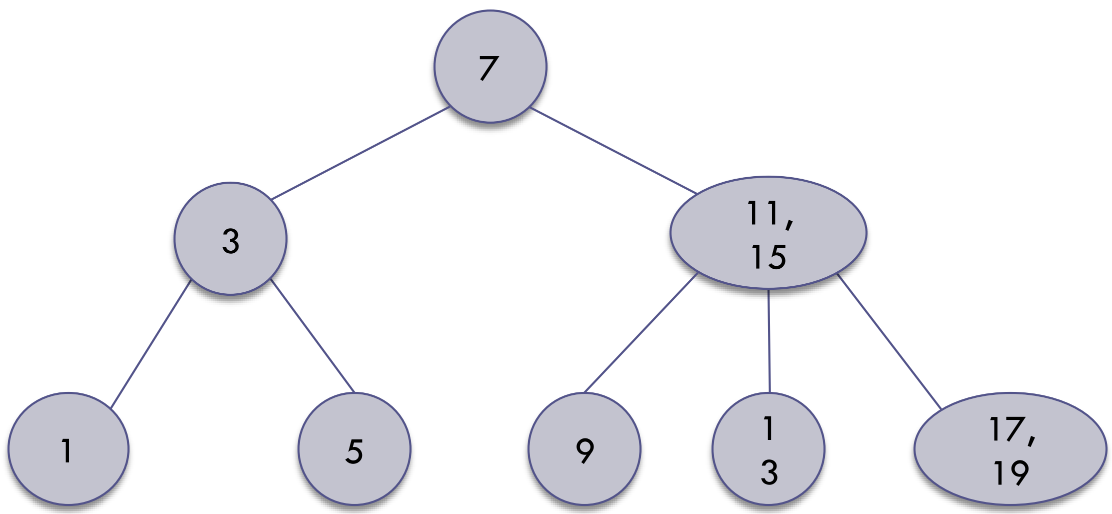
Analysis of 2-3 Trees and Comparison with Balanced Binary Trees

- 2-3 trees do not require the rotations needed for AVL and Red-Black trees
- The number of items that a 2-3 tree of height h can hold is between 2^h -1 (all 2 nodes) and $3^h - 1$ (all 3-nodes)
- Therefore, the height of a 2-3 tree is between $\log_3 n$ and $\log_2 n$
- The search time is $O(\log n)$

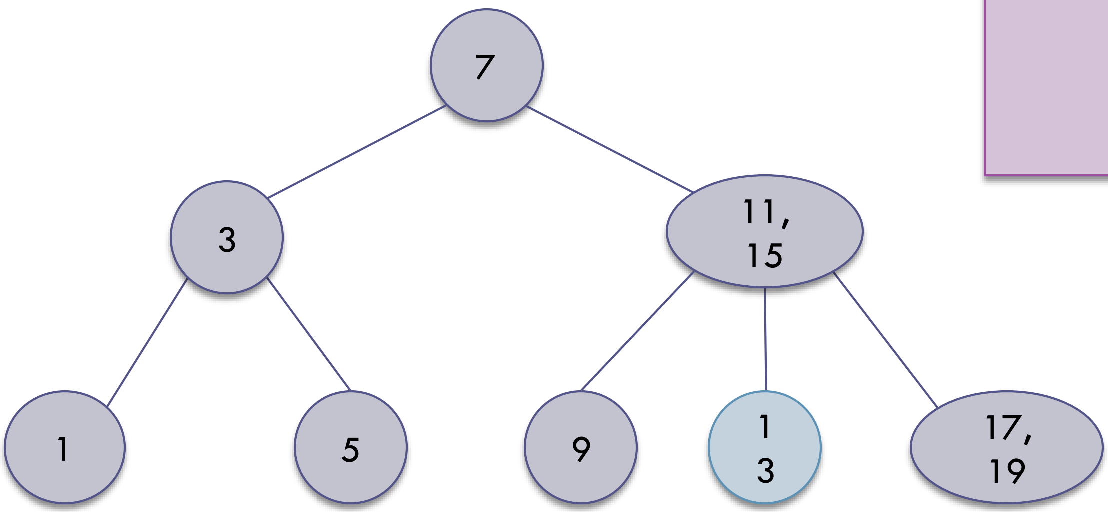
Removal from a 2-3 Tree

- Removing an item from a 2-3 tree is generally the reverse of the insertion process
- If the item to be removed is in a leaf, simply delete it
- If it's not in a leaf, remove it by swapping it with its inorder predecessor in a leaf node and deleting it from the leaf node
- If removing a node from a leaf causes the leaf to become empty,
  - items from the sibling and parent can be redistributed into that leaf
  - or the leaf can be merged with its parent and sibling nodes

Removal from a 2-3 Tree (cont.)

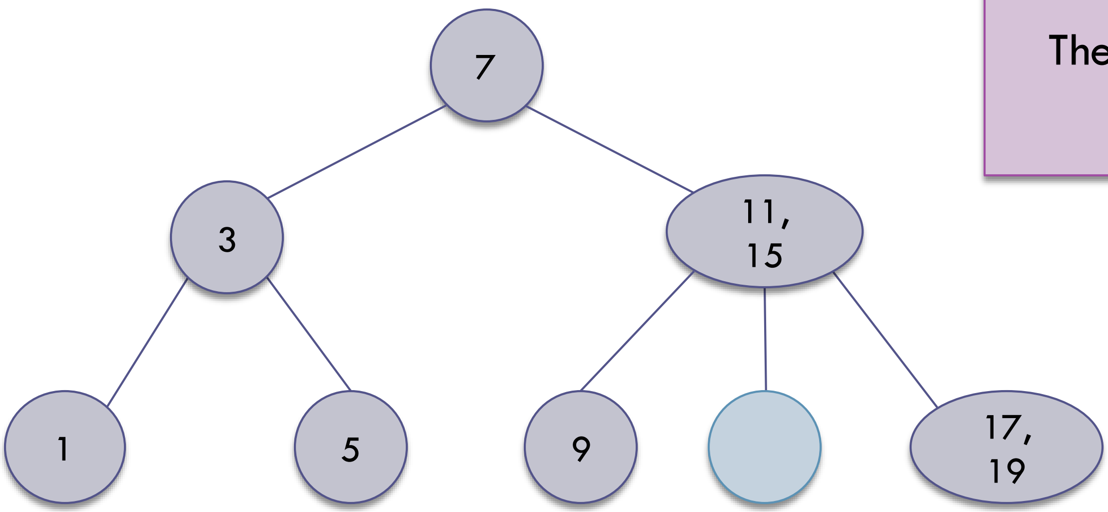


Removal from a 2-3 Tree (cont.)



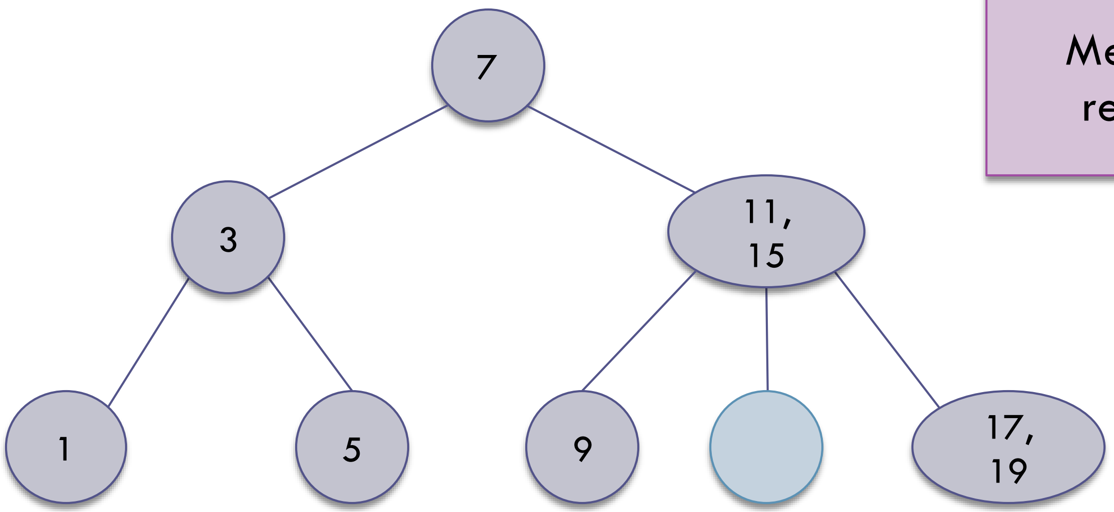
Remove 13

Removal from a 2-3 Tree (cont.)



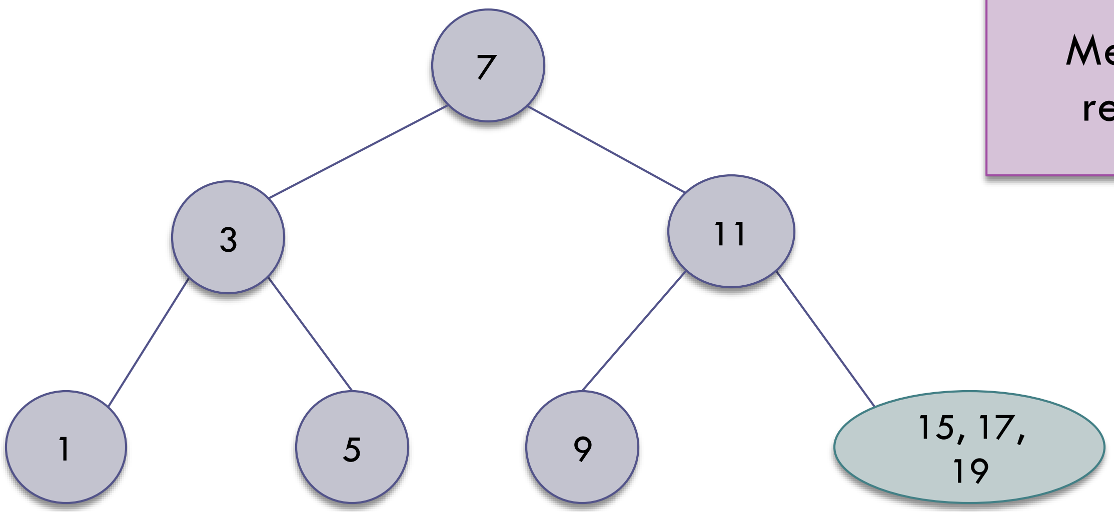
The node becomes empty

Removal from a 2-3 Tree (cont.)



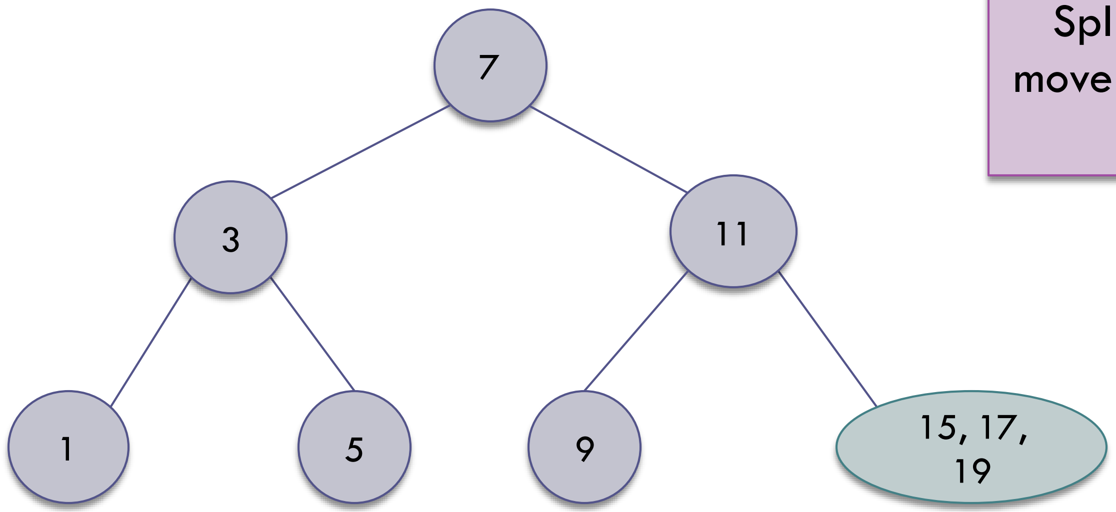
Merge 15 with a remaining child

Removal from a 2-3 Tree (cont.)



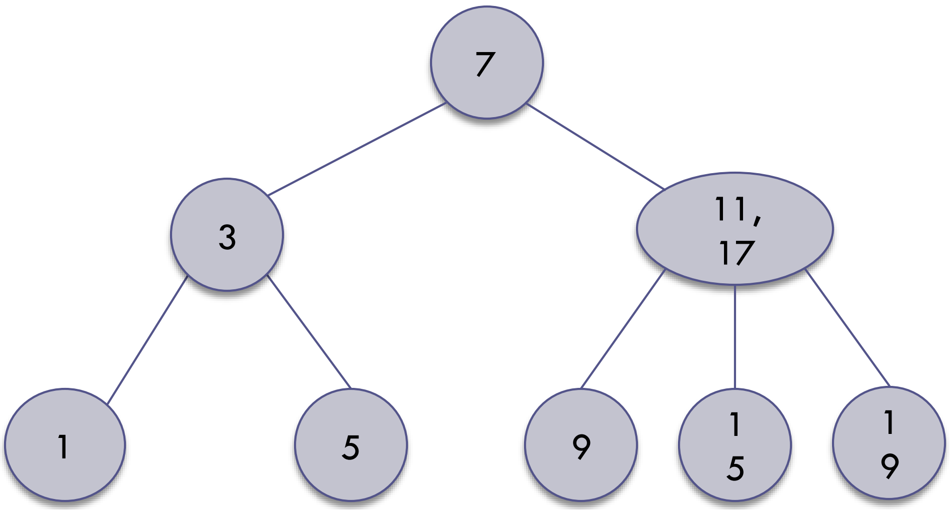
Merge 15 with a remaining child

Removal from a 2-3 Tree (cont.)



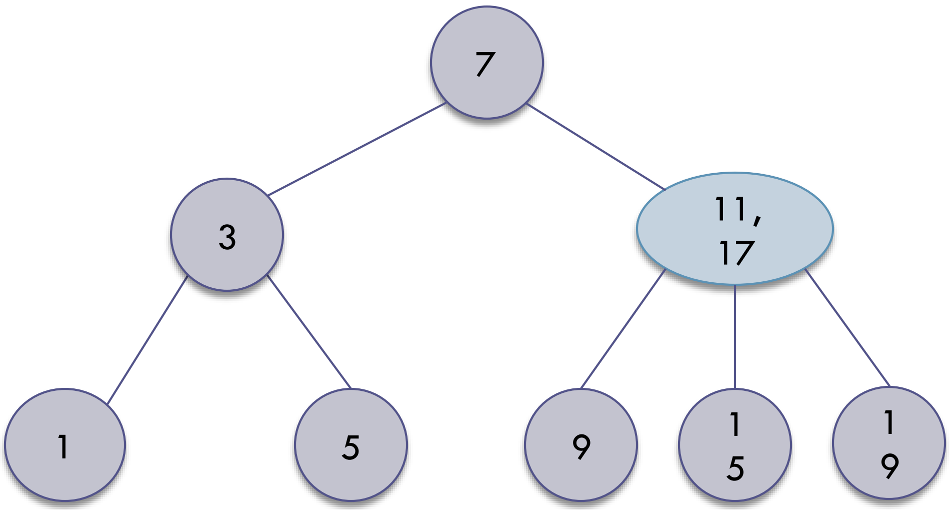
Split the node and move the middle value (17) up

Removal from a 2-3 Tree (cont.)



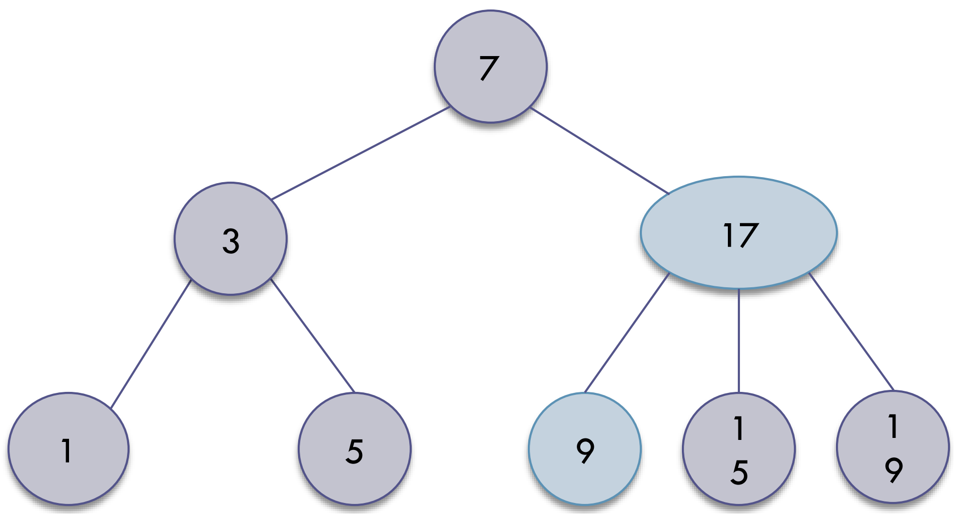
Split the node and move the middle value (17) up

Removal from a 2-3 Tree (cont.)



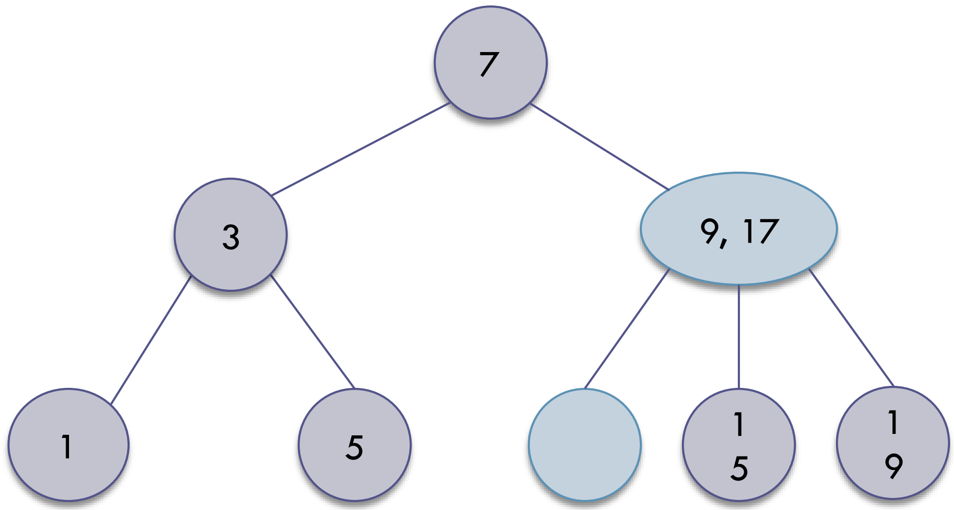
Remove 11

Removal from a 2-3 Tree (cont.)



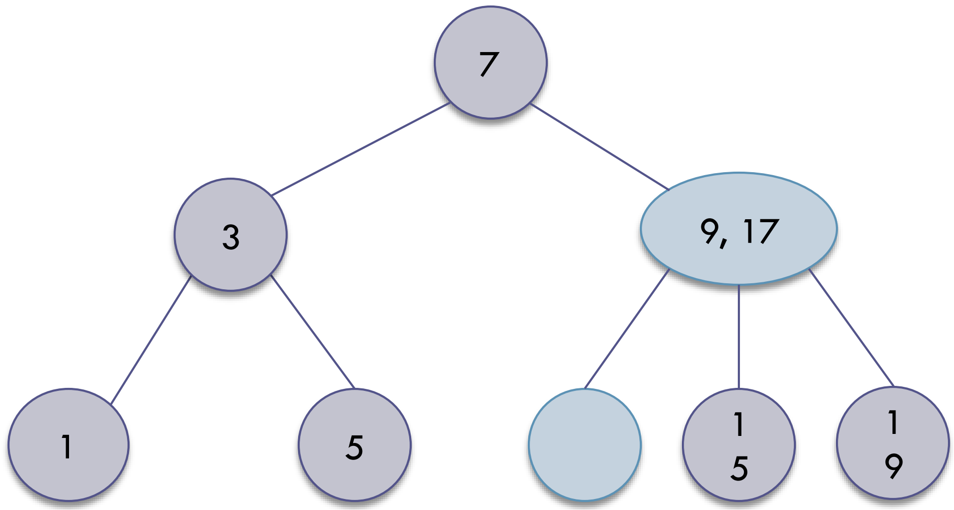
Because 11 is not in a leaf, replace it with its leaf predecessor (9)

Removal from a 2-3 Tree (cont.)



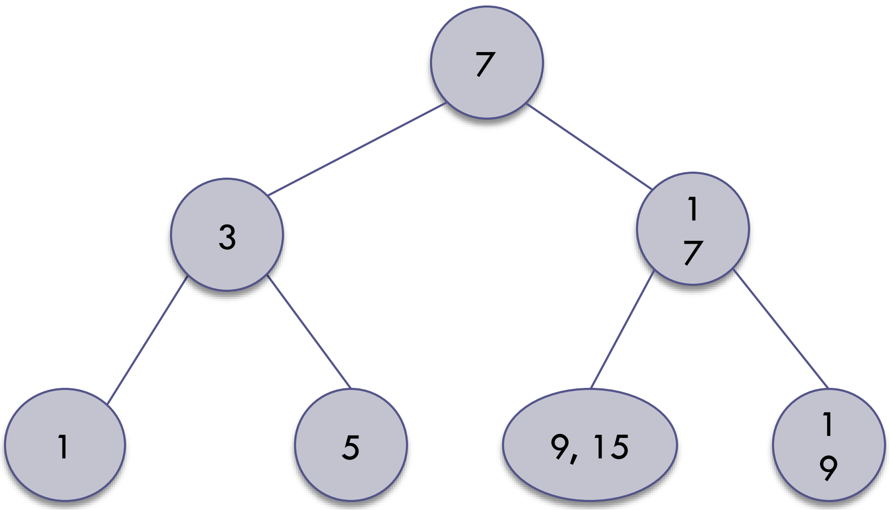
Because 11 is not in a leaf, replace it with its leaf predecessor (9)

Removal from a 2-3 Tree (cont.)



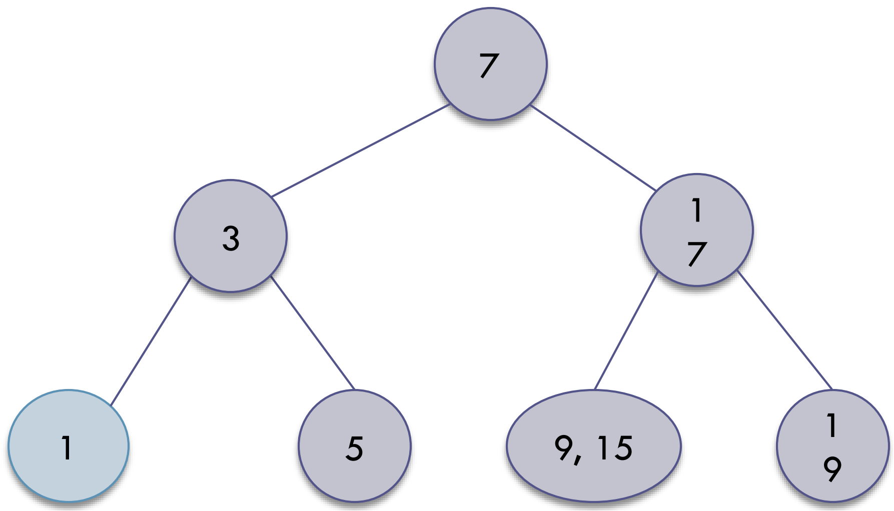
The left leaf is now empty. Merge the parent (9) into its right child (15)

Removal from a 2-3 Tree (cont.)



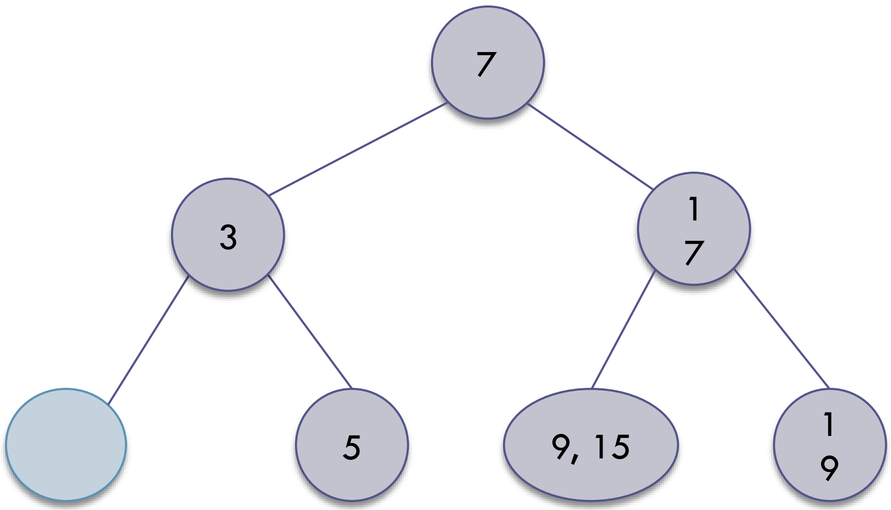
The left leaf is now empty. Merge the parent (9) into its right child (15)

Removal from a 2-3 Tree (cont.)



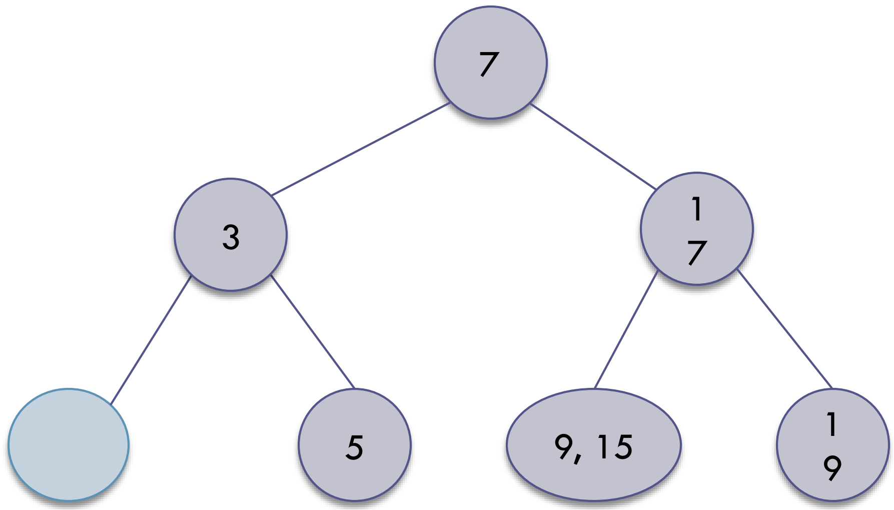
Remove 1

Removal from a 2-3 Tree (cont.)



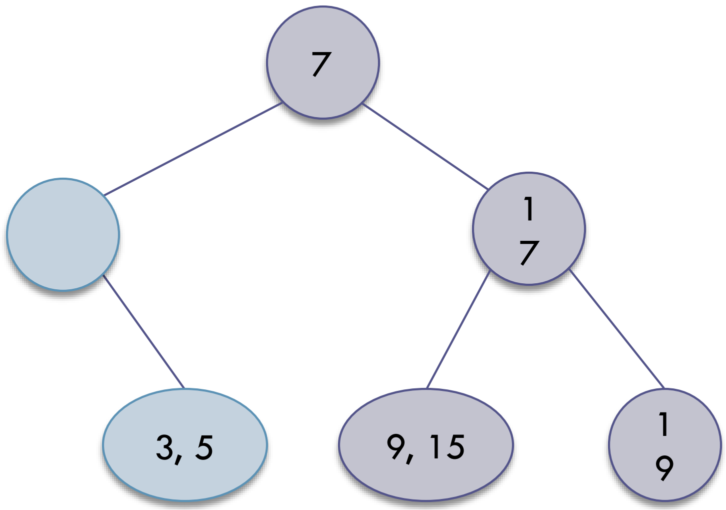
Remove 1

Removal from a 2-3 Tree (cont.)



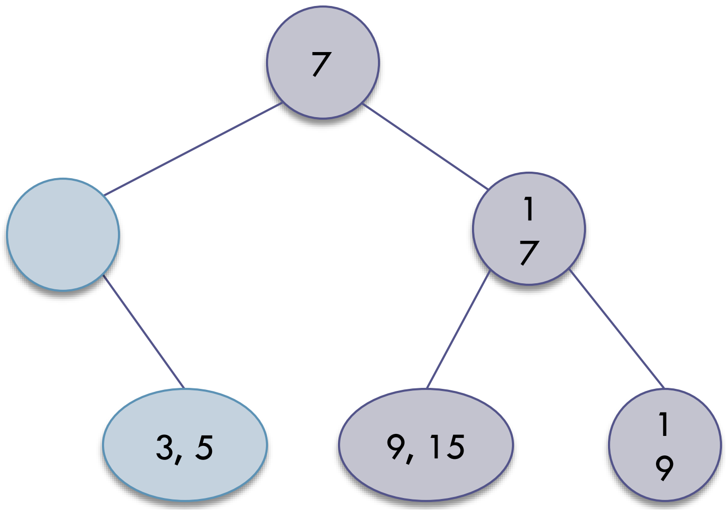
Merge the parent (3) with its right child (5)

Removal from a 2-3 Tree (cont.)



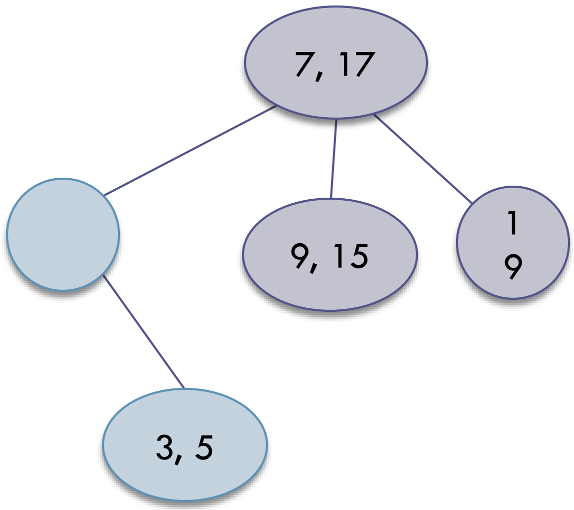
Merge the parent (3) with its right child (5)

Removal from a 2-3 Tree (cont.)



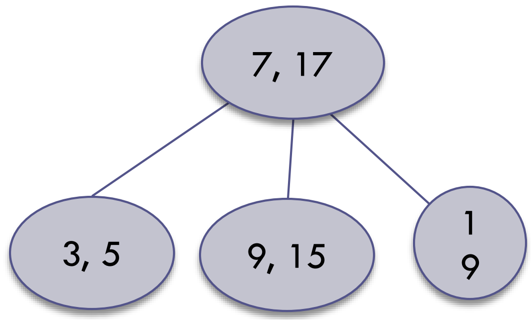
Repeat on the next level.
Merge the parent (7) with its right child (17)

Removal from a 2-3 Tree (cont.)



Repeat on the next level.
Merge the parent (7) with its right child (17)

Removal from a 2-3 Tree (cont.)



Repeat on the next level.
Merge the parent (7) with its right child (17)

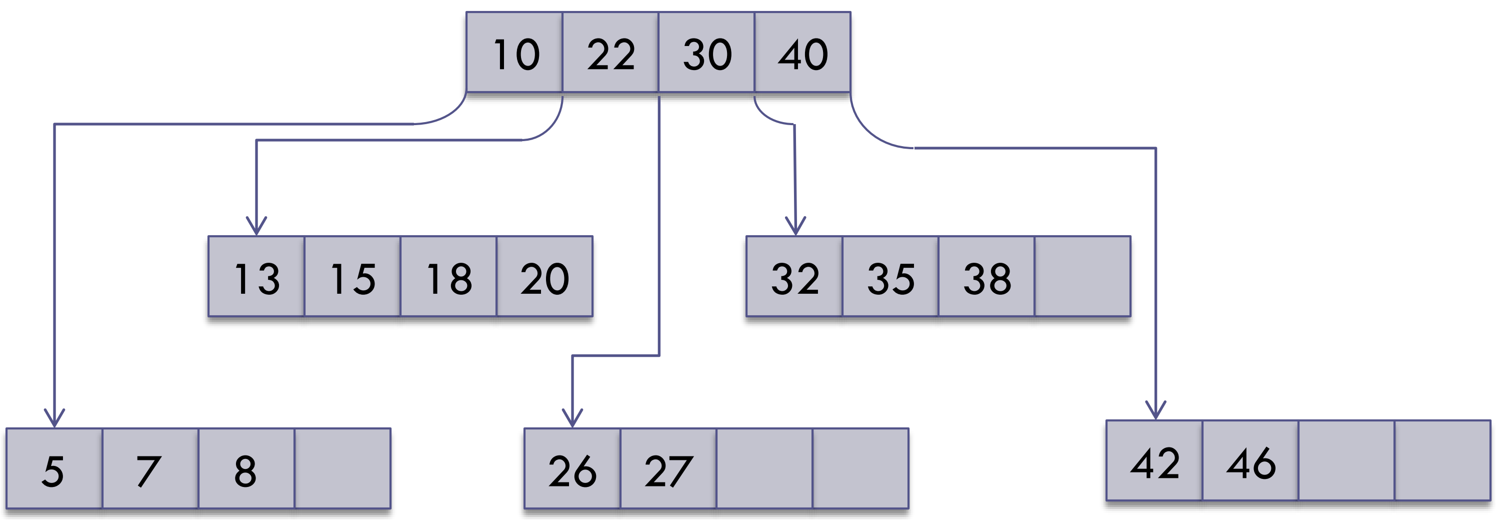
B-Trees and 2-3-4 Trees

Section 9.5

B-Trees and 2-3-4 Trees

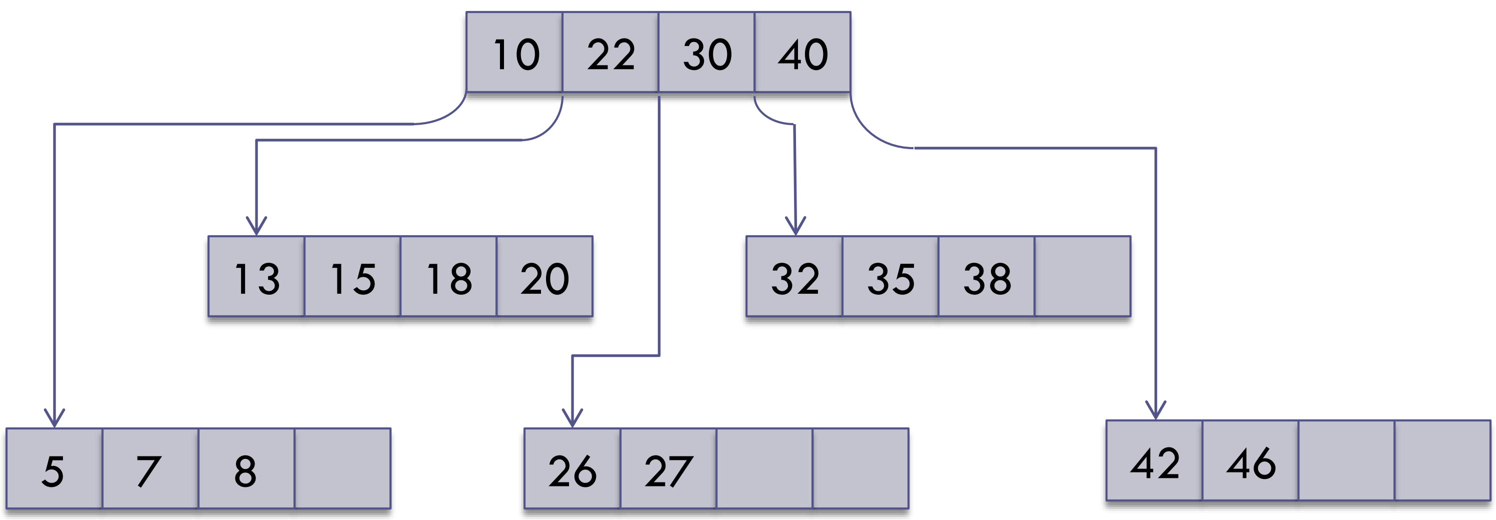
- The 2-3 tree was the inspiration for the more general B-tree which allows up to *n* children per node
- The B-tree was designed for building indexes to very large databases stored on a hard disk
- The 2-3-4 tree is a specialization of the B-tree

B-Trees



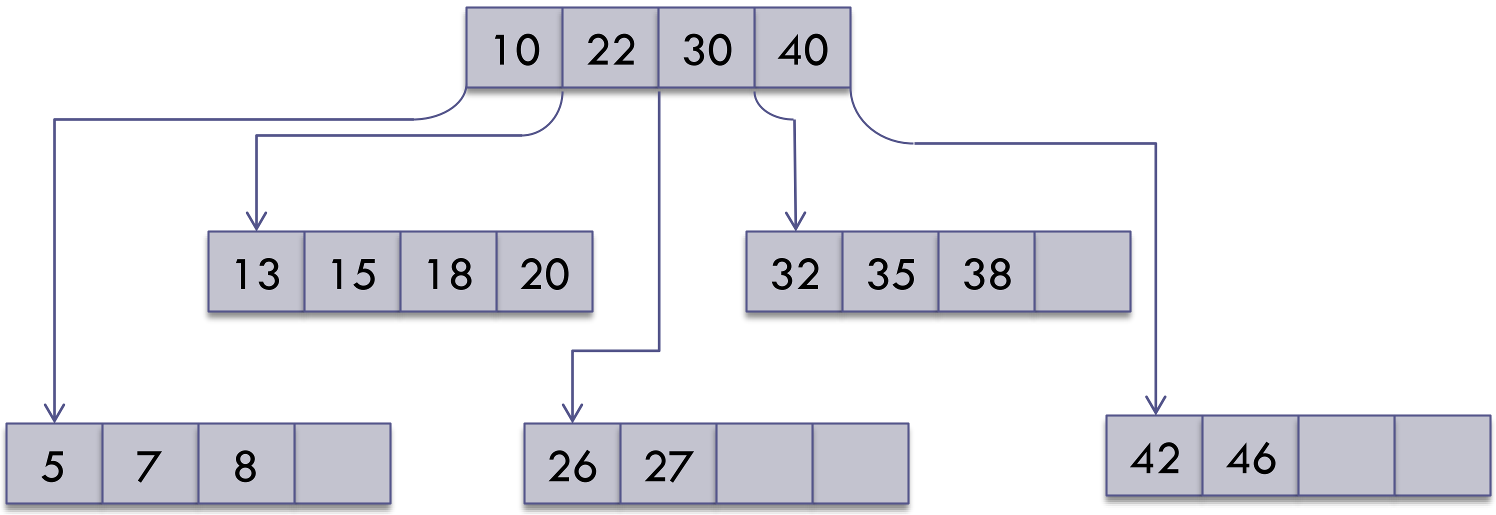
B-Trees (cont.)

The maximum number of children is the *order* of the B-tree, which we represent as the variable `order`



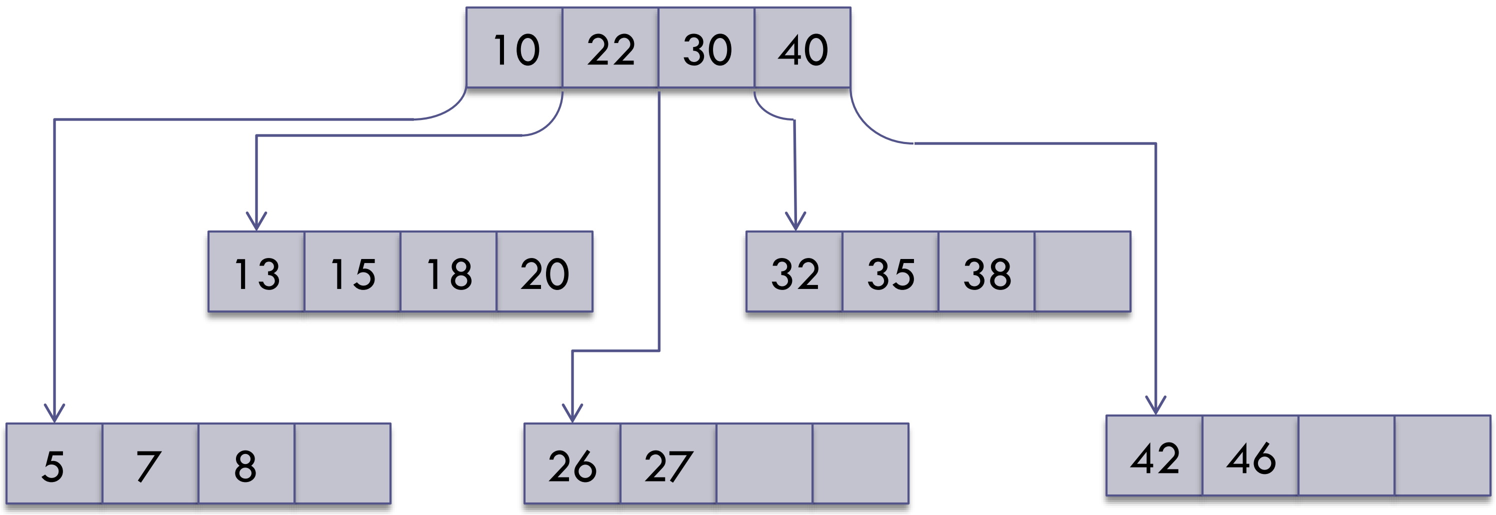
B-Trees (cont.)

The `order` of the B-tree below is 5



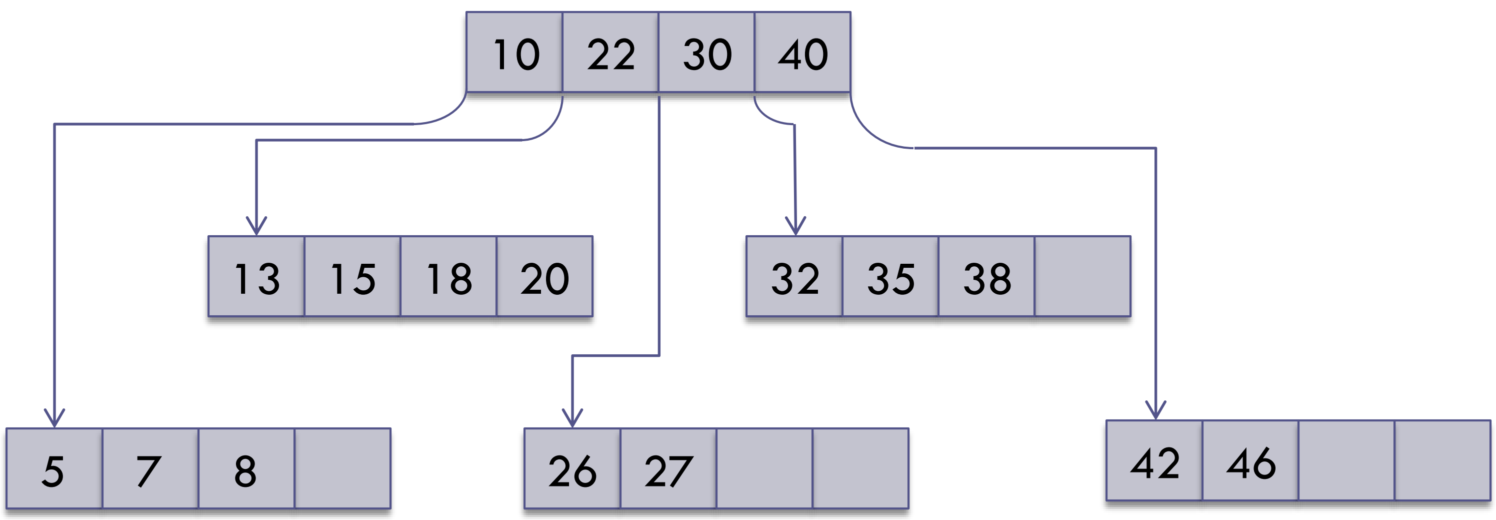
B-Trees (cont.)

The number of data items in a node is 1 less than the number of children (the order)



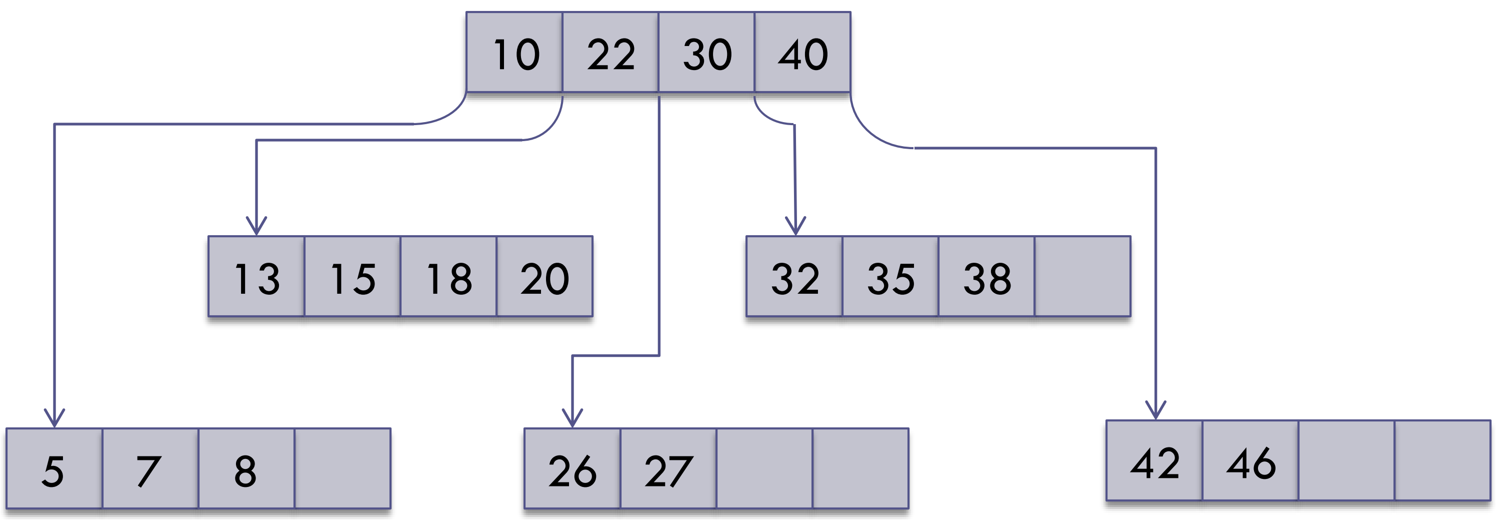
B-Trees (cont.)

Other than the root, each node has between `order/2` and `order` children



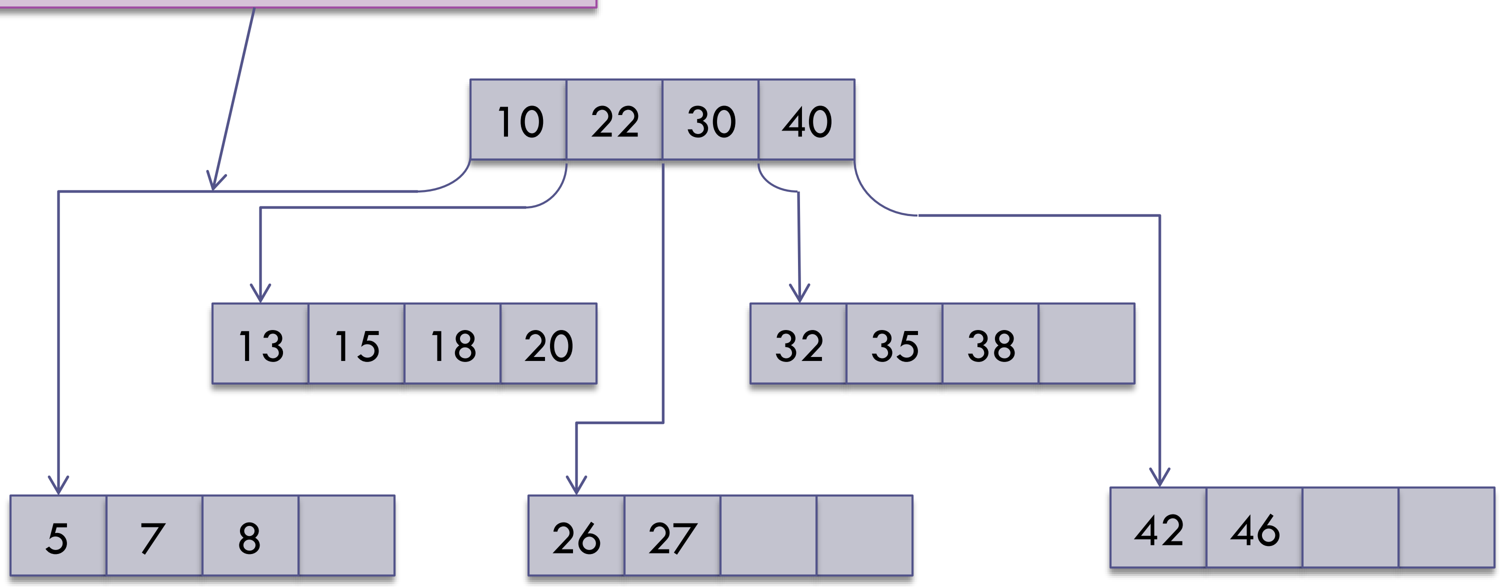
B-Trees (cont.)

The data items in each node are in increasing order



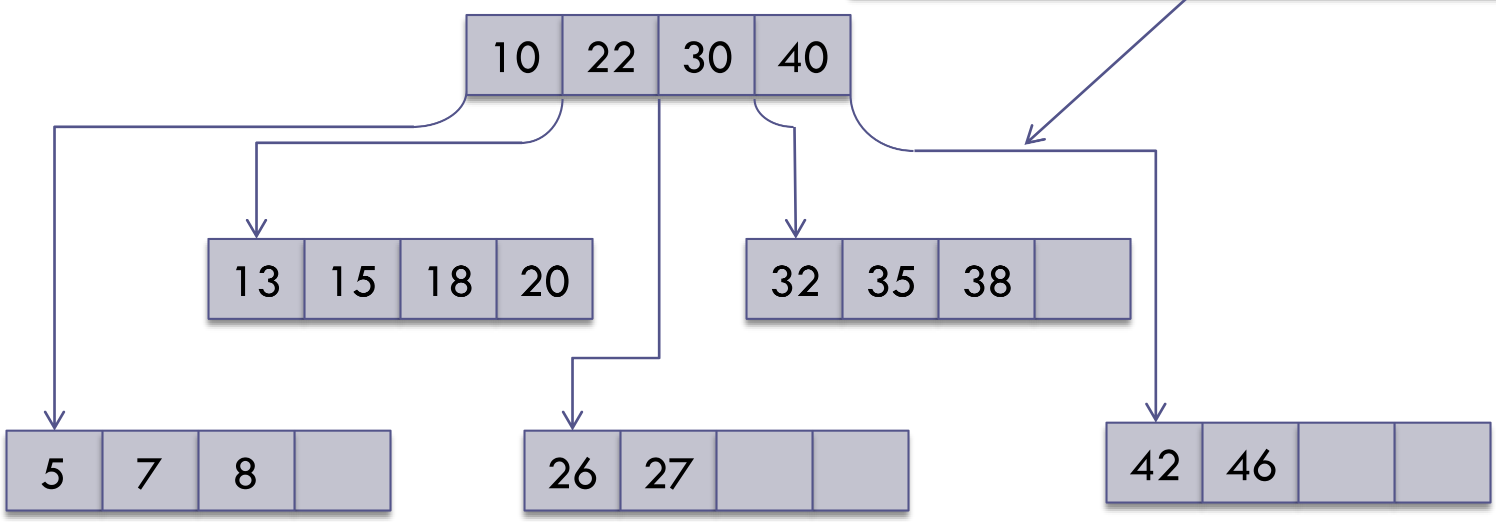
B-Trees (cont.)

The first link from a node connects it to a subtree with values smaller than the parent's smallest value



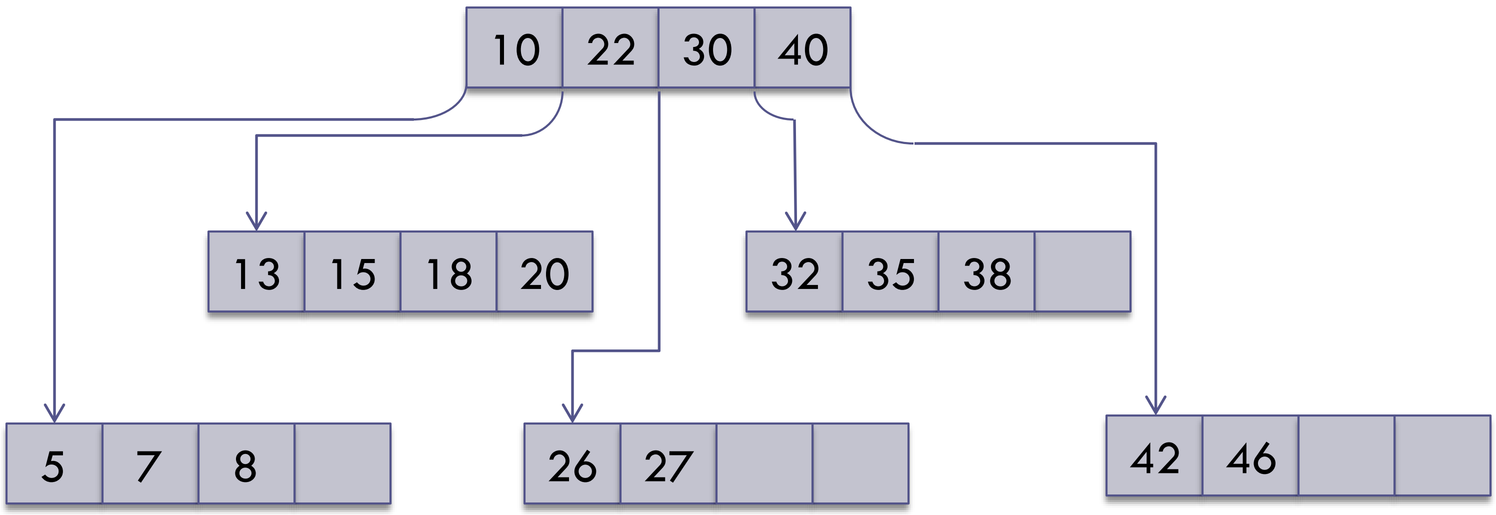
B-Trees (cont.)

The last link from a node connects it to a subtree with values greater than the parent's largest value



B-Trees (cont.)

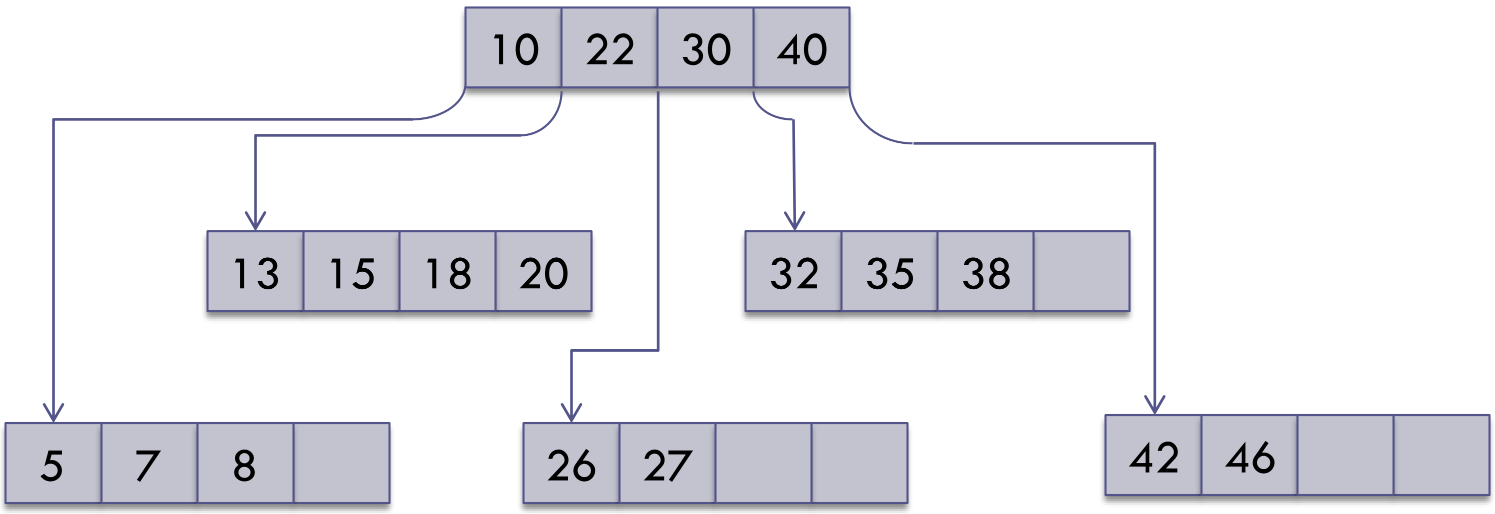
The other links are to subtrees with values between each pair of consecutive values in the parent node.



B-Trees (cont.)

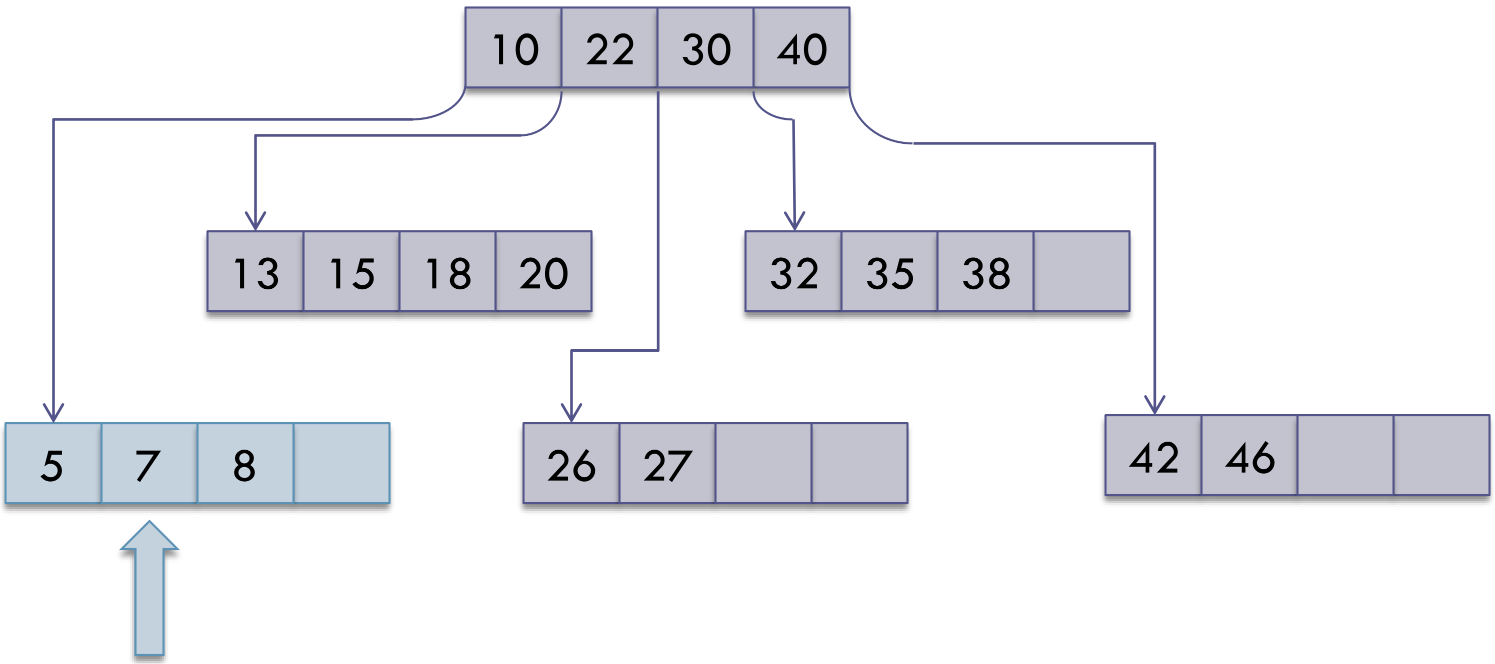
- B-Trees were developed to store indexes to databases on disk storage.
  - disk storage is broken into blocks
  - the nodes of a B-tree are sized to fit in a block
  - each disk access to the index retrieves exactly one B-tree node
  - the time to retrieve a block off the disk is large compared to the time to process it in memory
  - by making tree nodes as large as possible, we reduce the number of disk accesses required to find an item in the index
- Assuming a block can store a node for a B-tree of order 200, each node would store at least 100 items.
- This enables 100^4 or 100 million items to be accessed in a B-tree of height 4

B-Tree Insertion



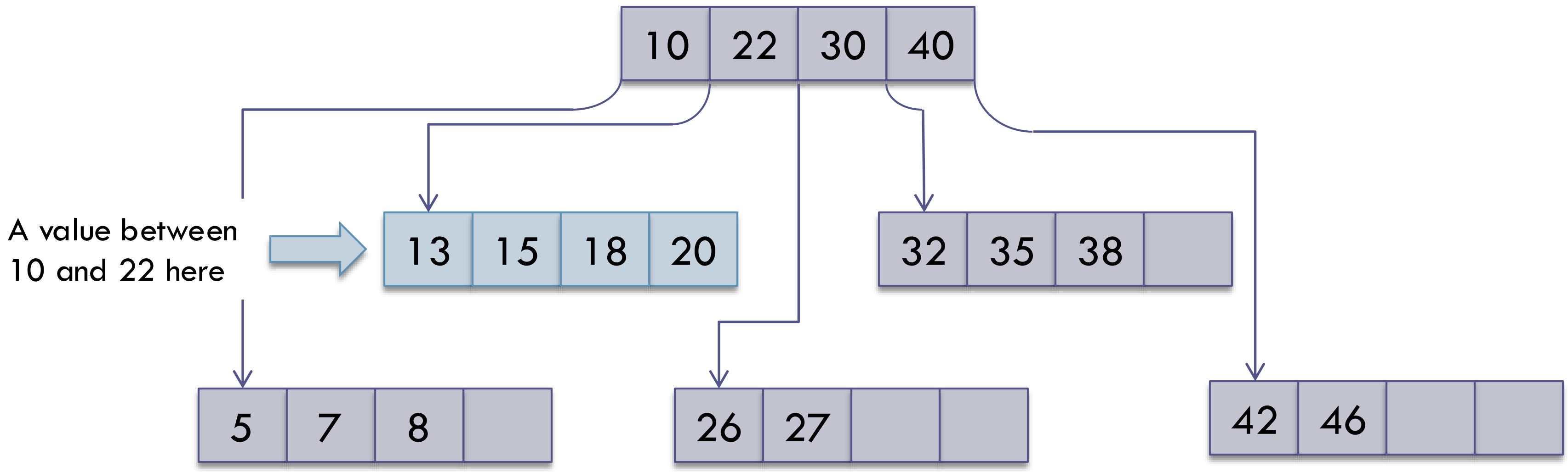
Similar to 2-3 trees, insertions take place in leaves

B-Tree Insertion (cont.)

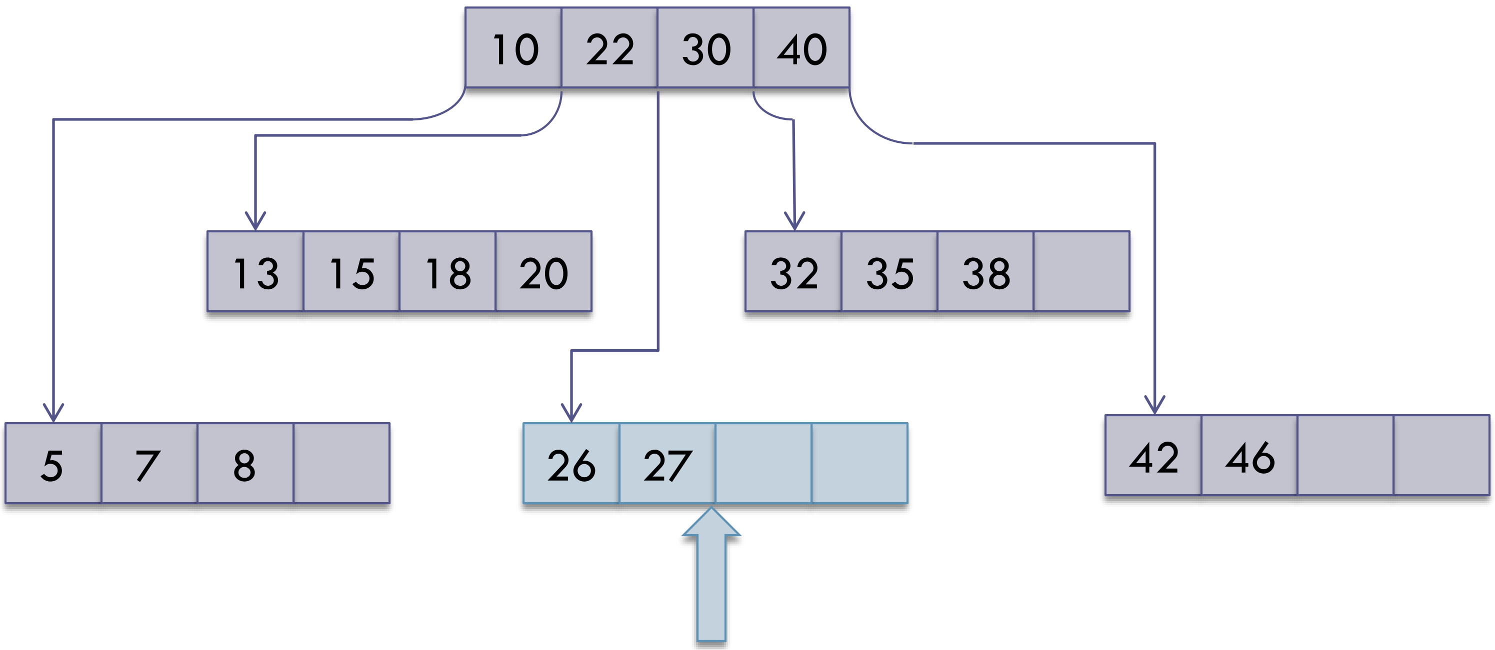


A value less than 10
would be inserted here

B-Tree Insertion (cont.)



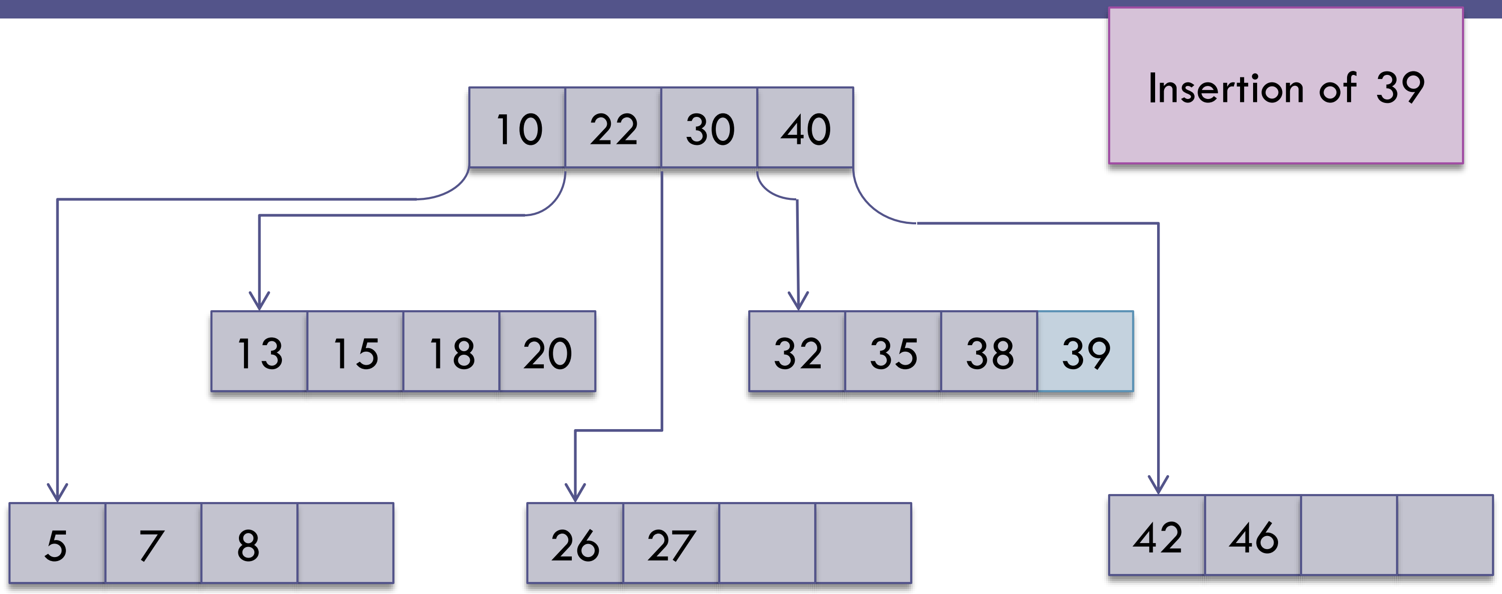
B-Tree Insertion (cont.)



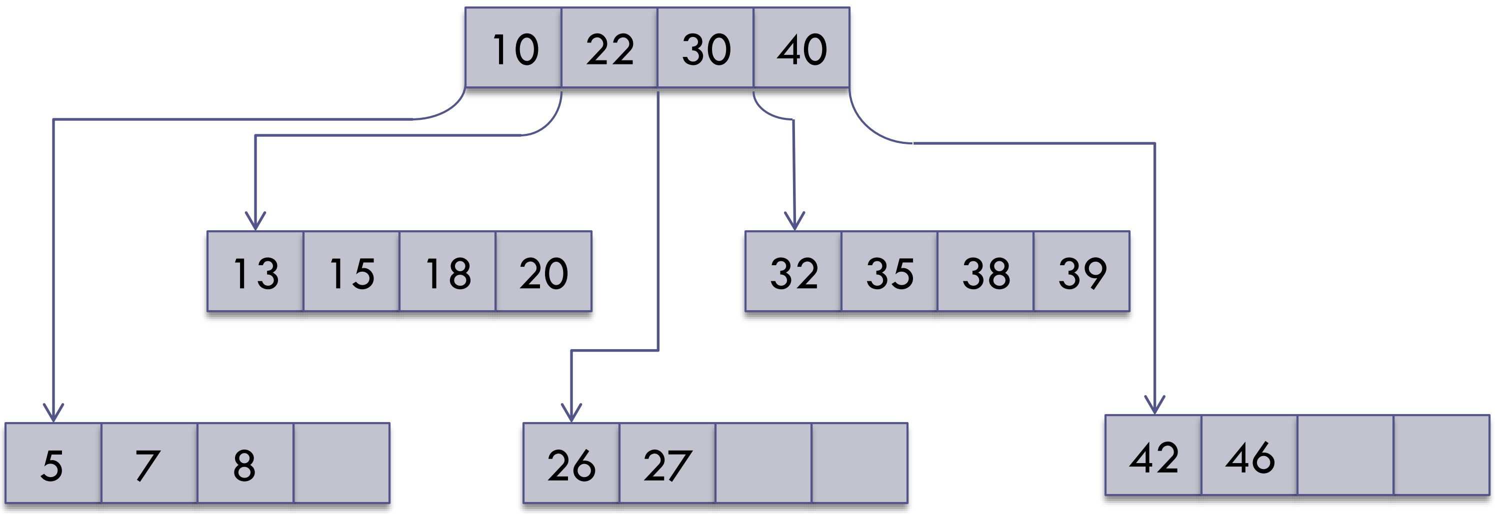
A value between 22
and 30 here

and so on . . .

B-Tree Insertion (cont.)

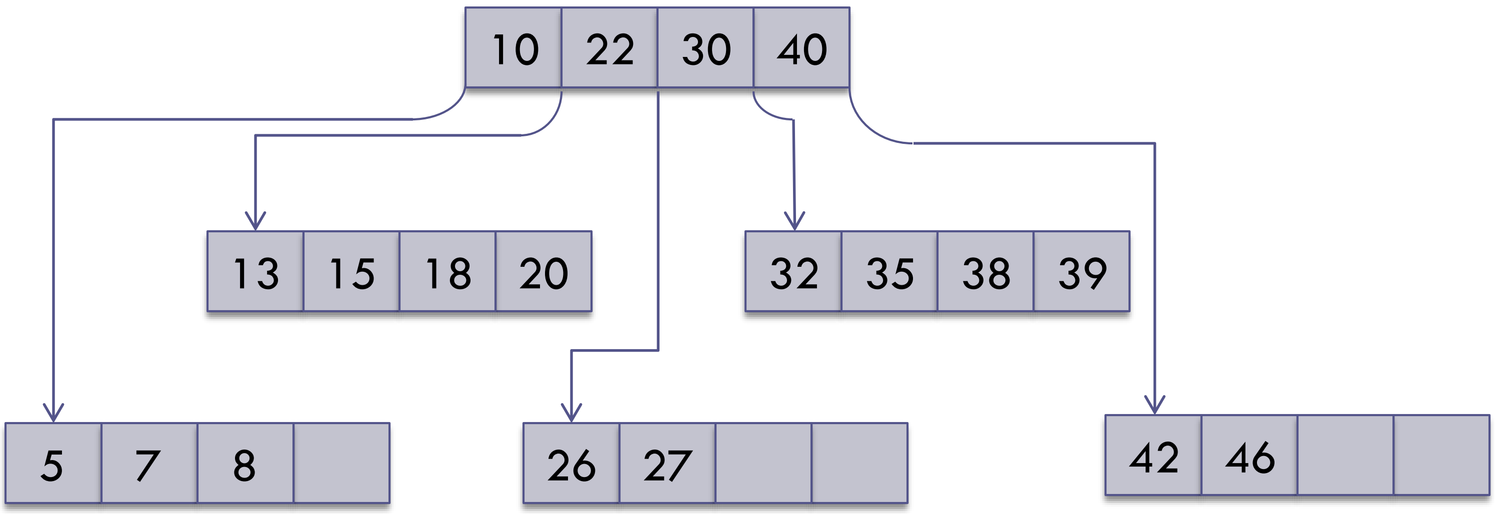


B-Tree Insertion (cont.)



If a leaf to receive the insertion is full, it is split into two nodes, each containing approximately half the items, and the middle item is passed up to the split node's parents

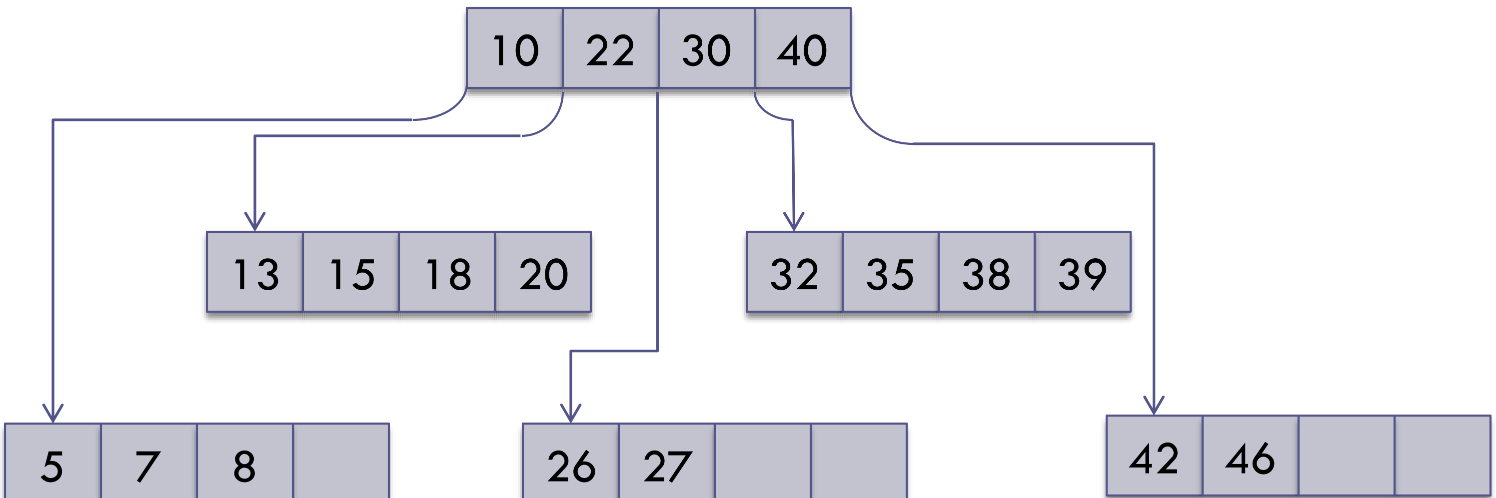
B-Tree Insertion (cont.)



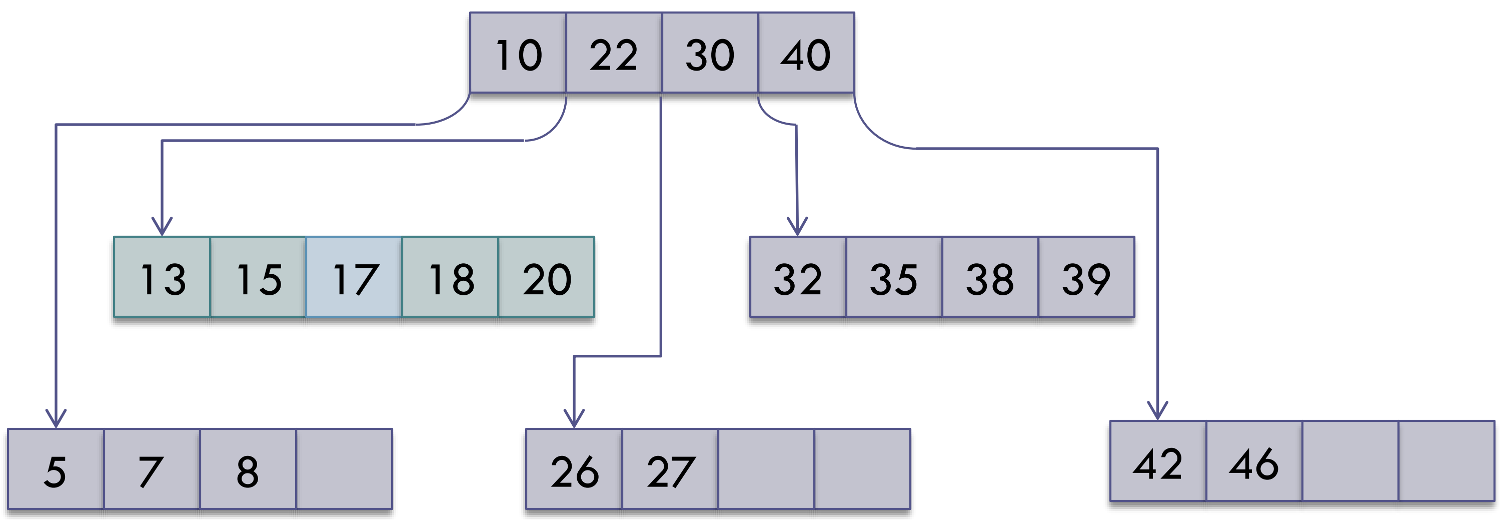
If the parent is full, it is split and its middle item is passed up to its parent, and so on

B-Tree Insertion (cont.)

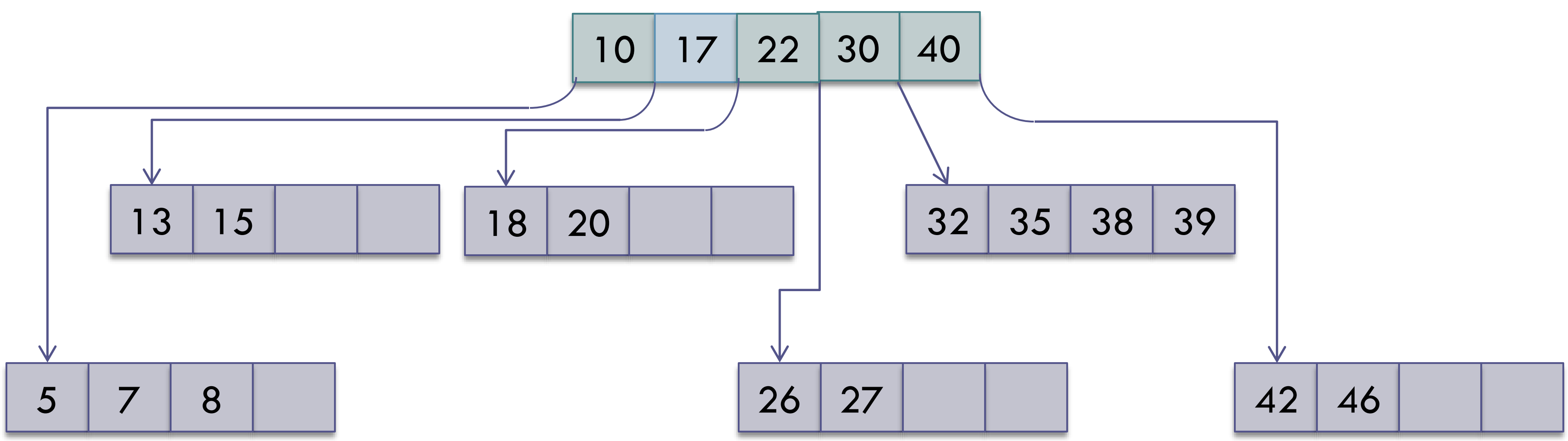
Insert 17



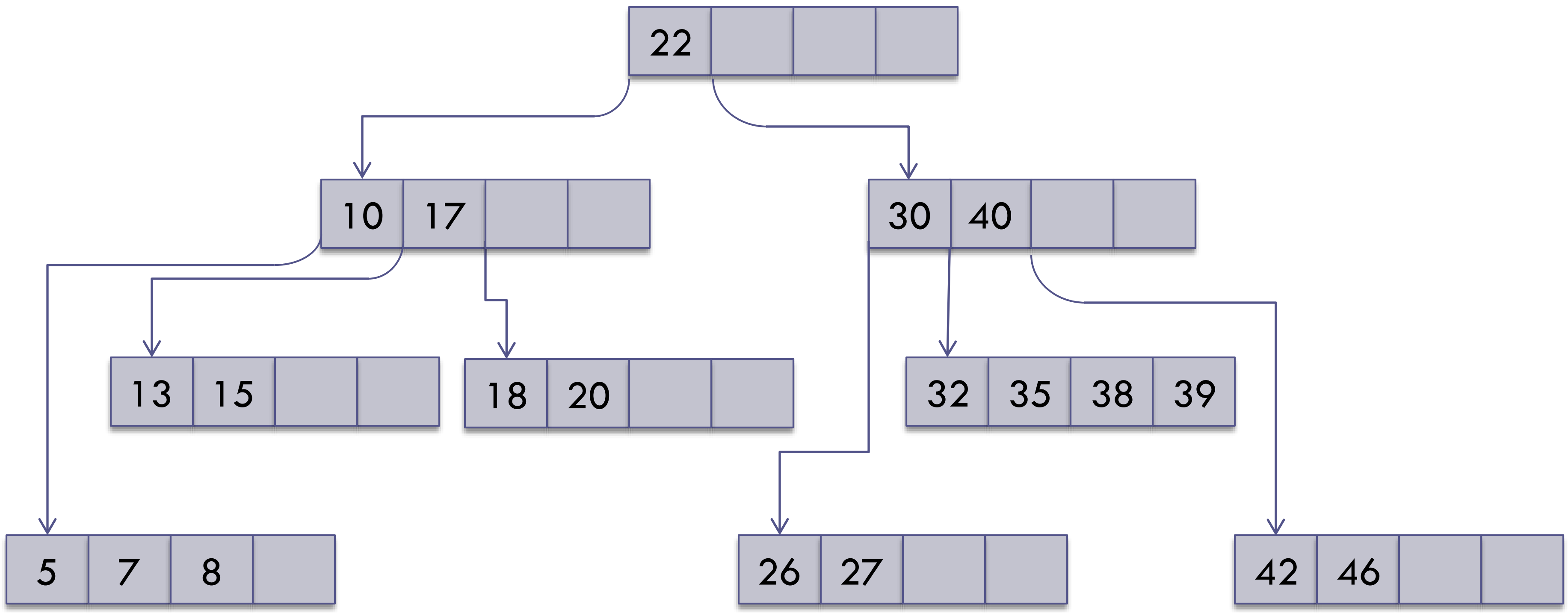
B-Tree Insertion (cont.)



B-Tree Insertion (cont.)



B-Tree Insertion (cont.)



Implementing the B-Tree

```
/** A Node represents a node in a B-tree. */
private static class Node<E> {
    // Data Fields

    /** The number of data items in this node */
    private int size = 0;
    /** The information */
    private E[] data;
    /** The links to the children. child[i] refers to
         the subtree of children < data[i] for i < size
         and to the subtree of children > data[size-1]
         for i == size */
    private Node<E>[] child;

    /** Create an empty node of size order
         @param order The size of a node
    */
    @SuppressWarnings("unchecked")
    public Node(int order) {
         data = (E[]) new Comparable[order - 1];
         child = (Node<E>[]) new Node[order];
         size = 0;
    }
}
```

Implementing the B-Tree (cont.)

```
/** An implementation of the B-tree. A B-tree is a
    search tree in which each node contains n data items where
    n is between (order-1)/2 and order-1. (For the root, n may be
    between 1 and order-1.) Each node not a leaf has n+1 children. The
    tree is always balanced in that all leaves are on the same level,
    i.e., the length of the path from the root to a leaf is constant.
    @author Koffman and Wolfgang
*/

    public class BTree<E extends Comparable<E>>
    // Nested class
    /** A Node represents a node in a B-tree. */
    private static class Node<E> {
    . . .
    }

    /** The root node. */
    private Node<E> root = null;
    /** The maximum number of children of a node */
    private int order;

    /** Construct a B-tree with node size order
    @param order the size of a node
    */
    public BTree(int order) {
    this.order = order;
    root = null;
    }
```

Implementing the B-Tree (cont.)

Algorithm for `insertion`

```
1.  if the root is null
2.      Create a new Node that contains the inserted item
3.  else search the local root for the item
4.      if the item is in the local root
5.          return false
6.      else
7.          if the local root is a leaf
8.              if the local root is not full
9.                  insert the new item
10.                 return null as the new_child
                    and true to indicate successful insertion
11.             else
12.                 split the local root
13.                 return the newParent and a newChild
                    and true to indicate successful insertion
14.         else
15.                 recursively call the insert method
16.             if the returned newChild is not null
17.                 if the local root is not full
18.                         insert the newParent and newChild into the
                            local root
19.                         return null as the newChild
                            and true to indicate successful insertion
20.                 else
21.                         split the local root
22.                         return the newParent and the newChild
                            and true to indicate successful insertion
23.             else
24.                 return the success/fail indicator for the insertion
```

Code for the `insert` Method

```
if (root.size < order - 1) {
    insertIntoNode(root, index, item, null);
    newChild = null;
} else {
    splitNode(root, index, item, null);
}
return true;

boolean result = insert(root.child[index], item);
if (newChild != null) {
    if (root.size < order - 1) {
        insertIntoNode(root, index, newParent, newChild);
        newChild = null;
    } else {
        splitNode(root, index, newParent, newChild);
    }
}
return result;
```

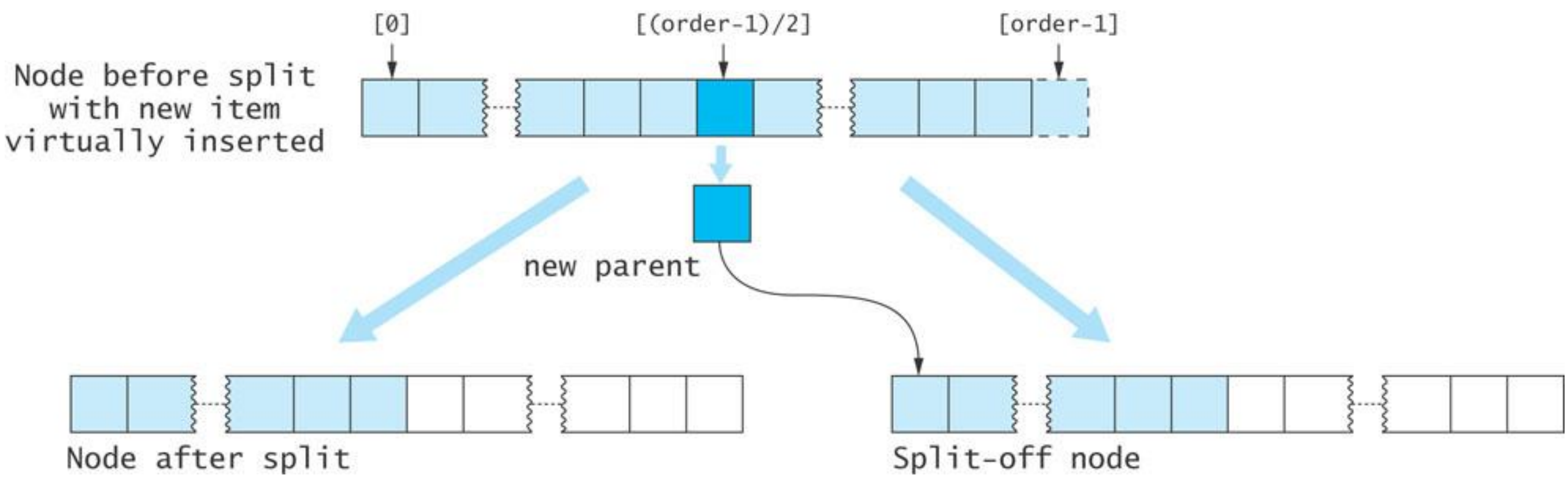
Code for the `insert` Method (cont.)

- Listing 9.5 (`The insert Method`, page 515)

The `insertIntoNode` Method

```
/** Method to insert a new value into a node
    pre: node.data[index-1] < item < node.data[index];
    post: node.data[index] == item and old values are moved
        right one position
    @param node The node to insert the value into
    @param index The index where the inserted item is to be placed
    @param item The value to be inserted
    @param child The right child of the value to be inserted
*/
private void insertIntoNode(Node<E> node, int index,
                            E obj, Node<E> child) {
    for (int i = node.size; i > index; i--) {
        node.data[i] = node.data[i - 1];
        node.child[i + 1] = node.child[i];
    }
    node.data[index] = obj;
    node.child[index + 1] = child;
    node.size++;
}
```

The `splitNode` Method



The `splitNode` Method (cont.)

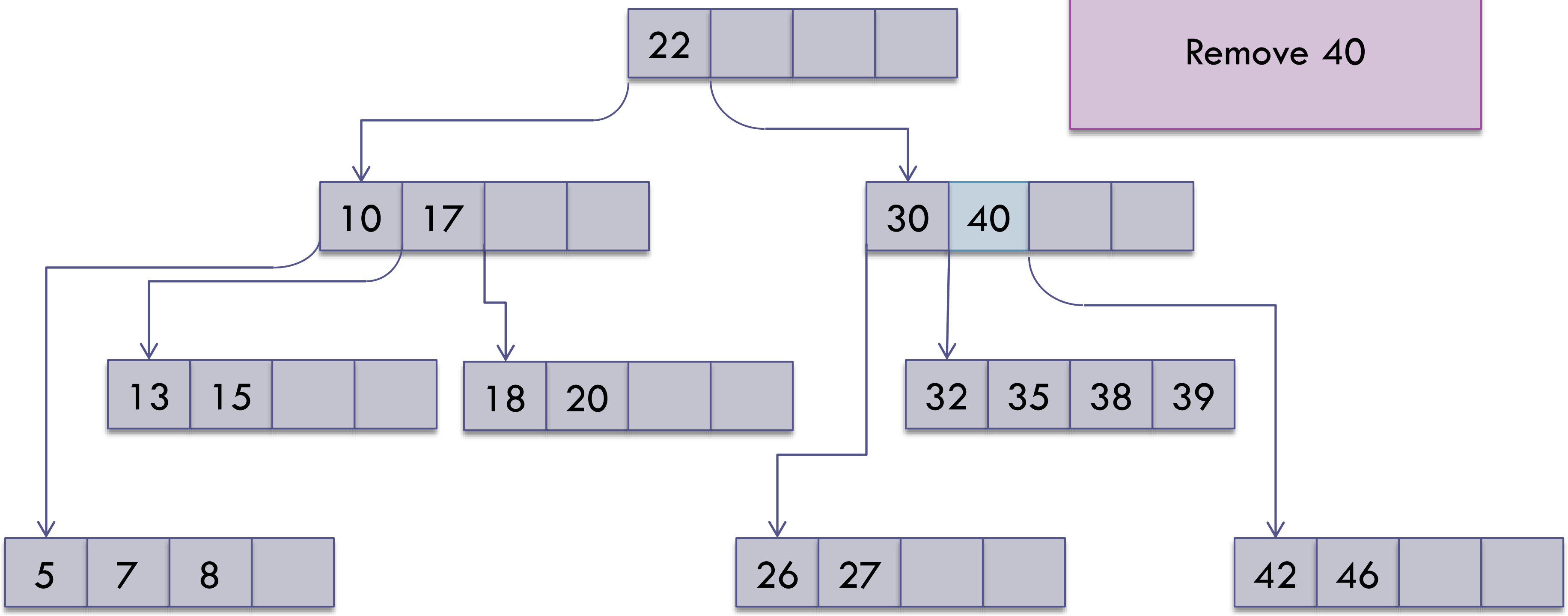
- Listing 9.6 (`The splitNode Method`, pages 517-18)

Removal from a B-Tree

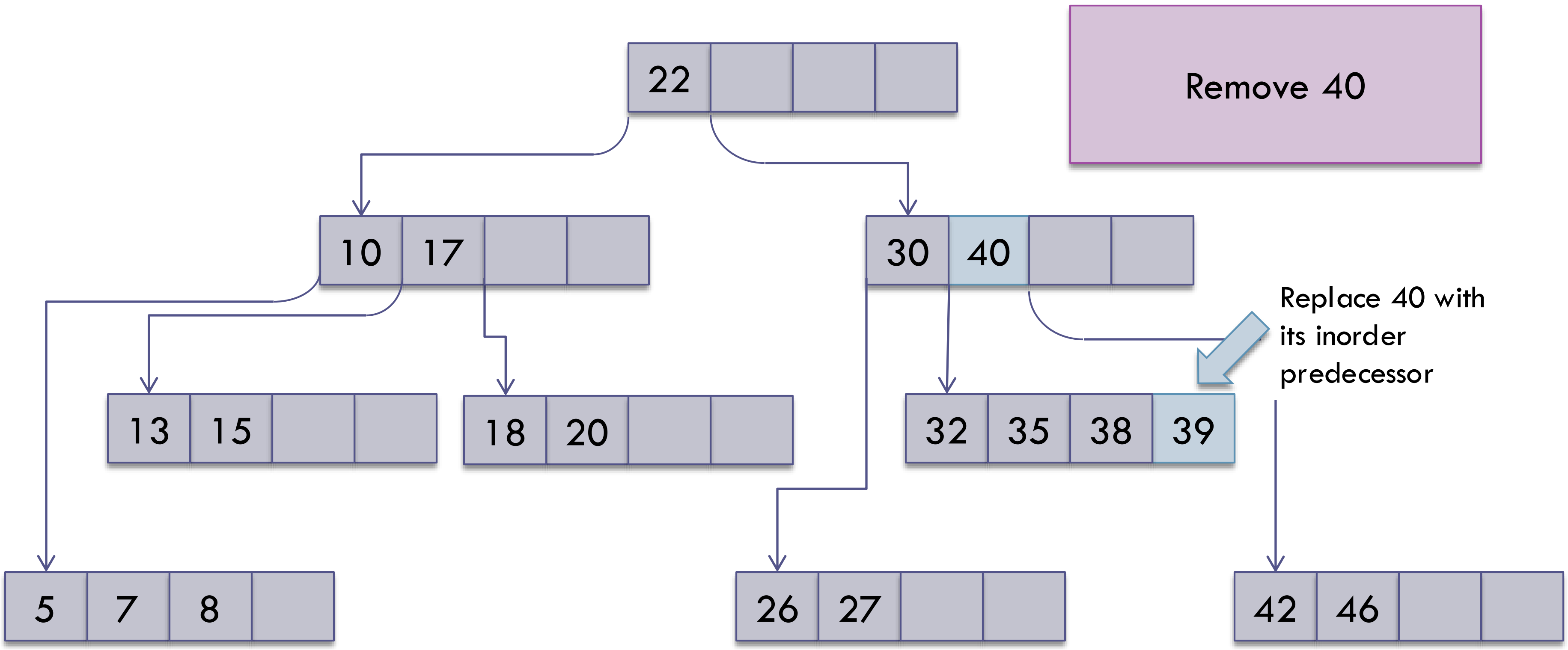
- Removing an item is a generalization of removing an item from a 2-3 tree
- The simplest removal is deletion from a leaf
- When an item is removed from an interior node, it must be replaced by its inorder predecessor (or successor) in a leaf
- If removing an item from a leaf results in the leaf being less than half full, redistribution needs to occur

Removal from a B-Tree (cont.)

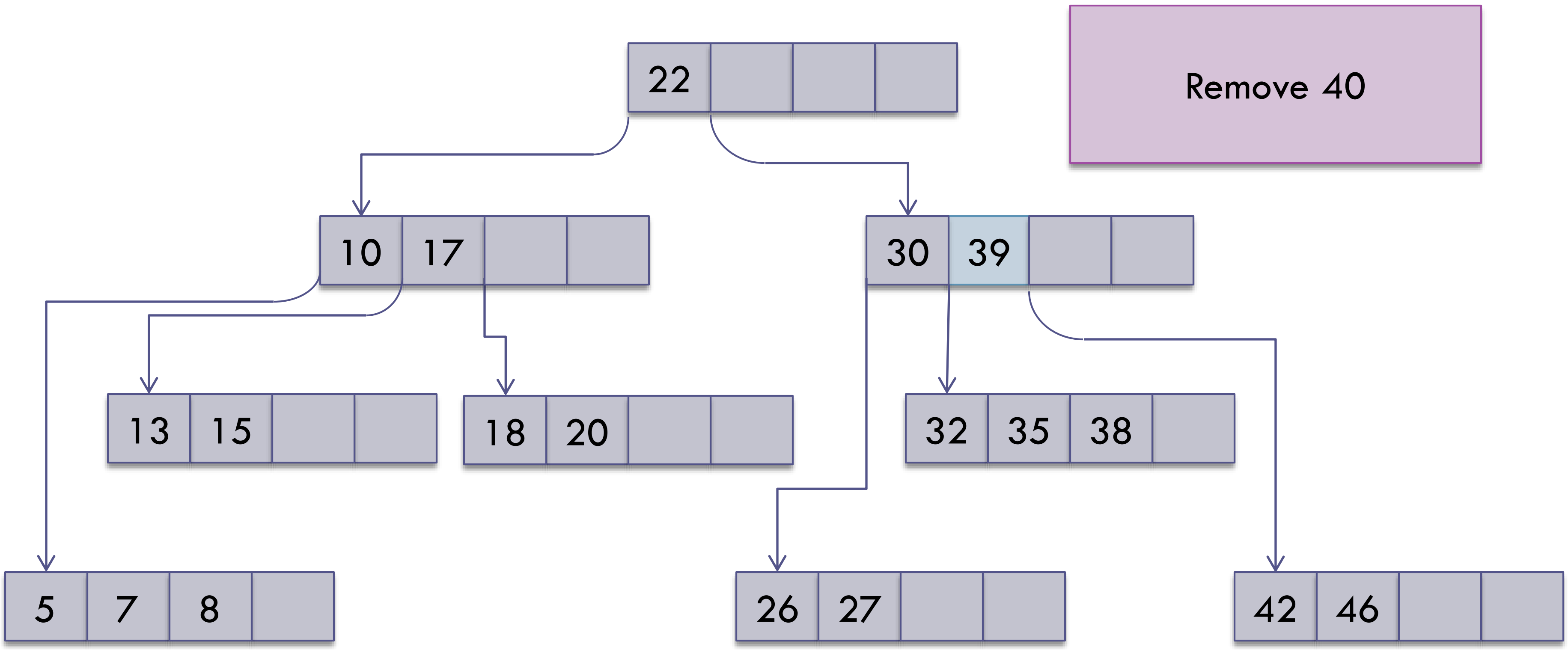
Remove 40



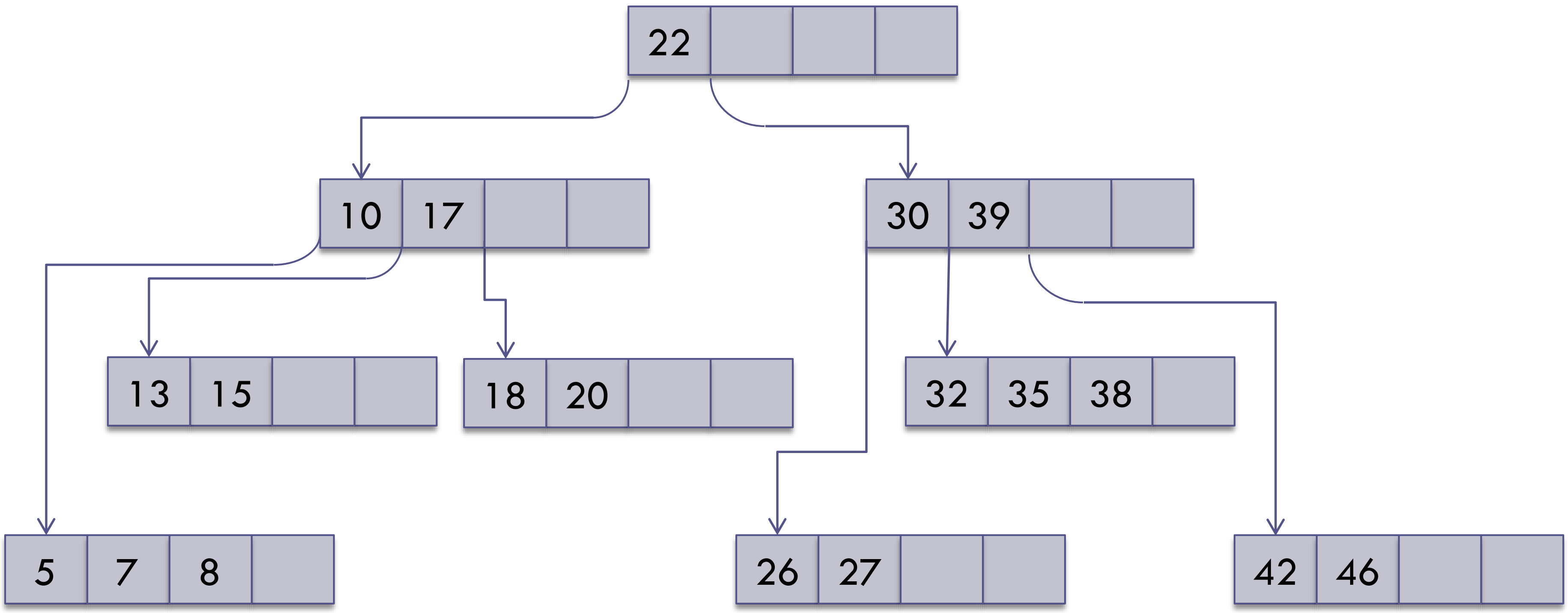
Removal from a B-Tree (cont.)



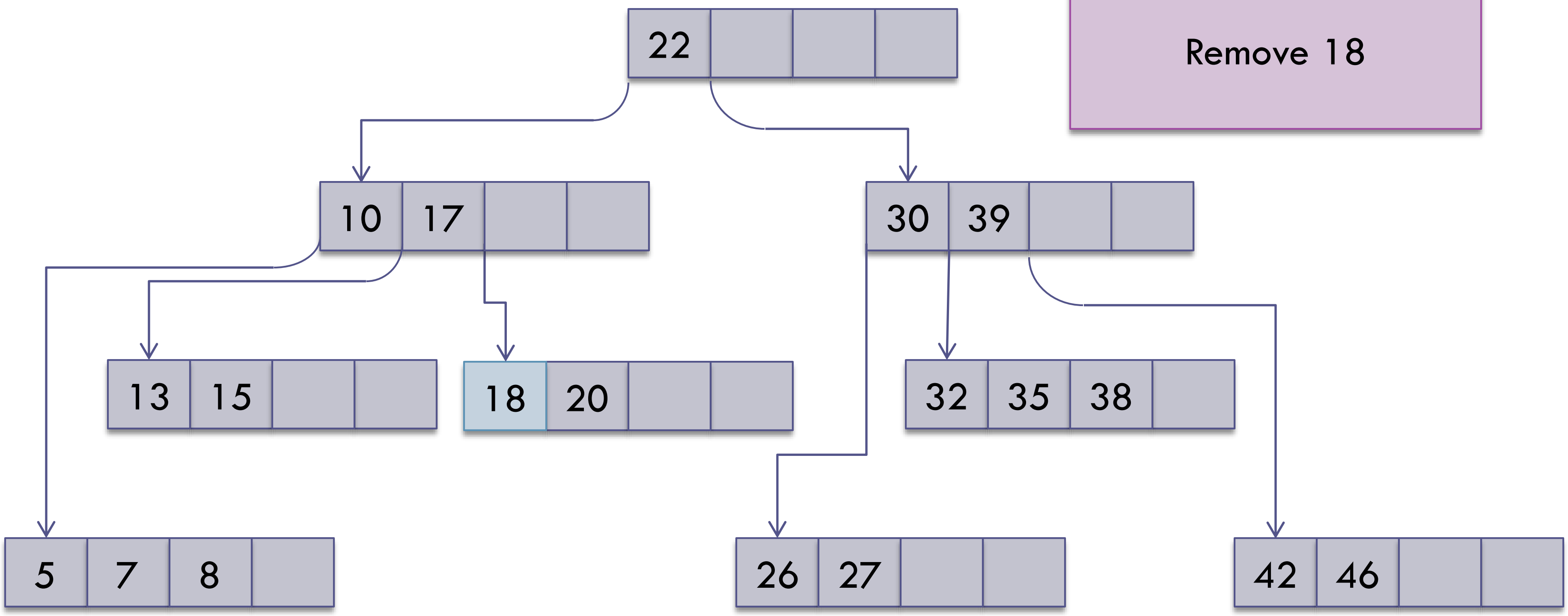
Removal from a B-Tree (cont.)



Removal from a B-Tree (cont.)

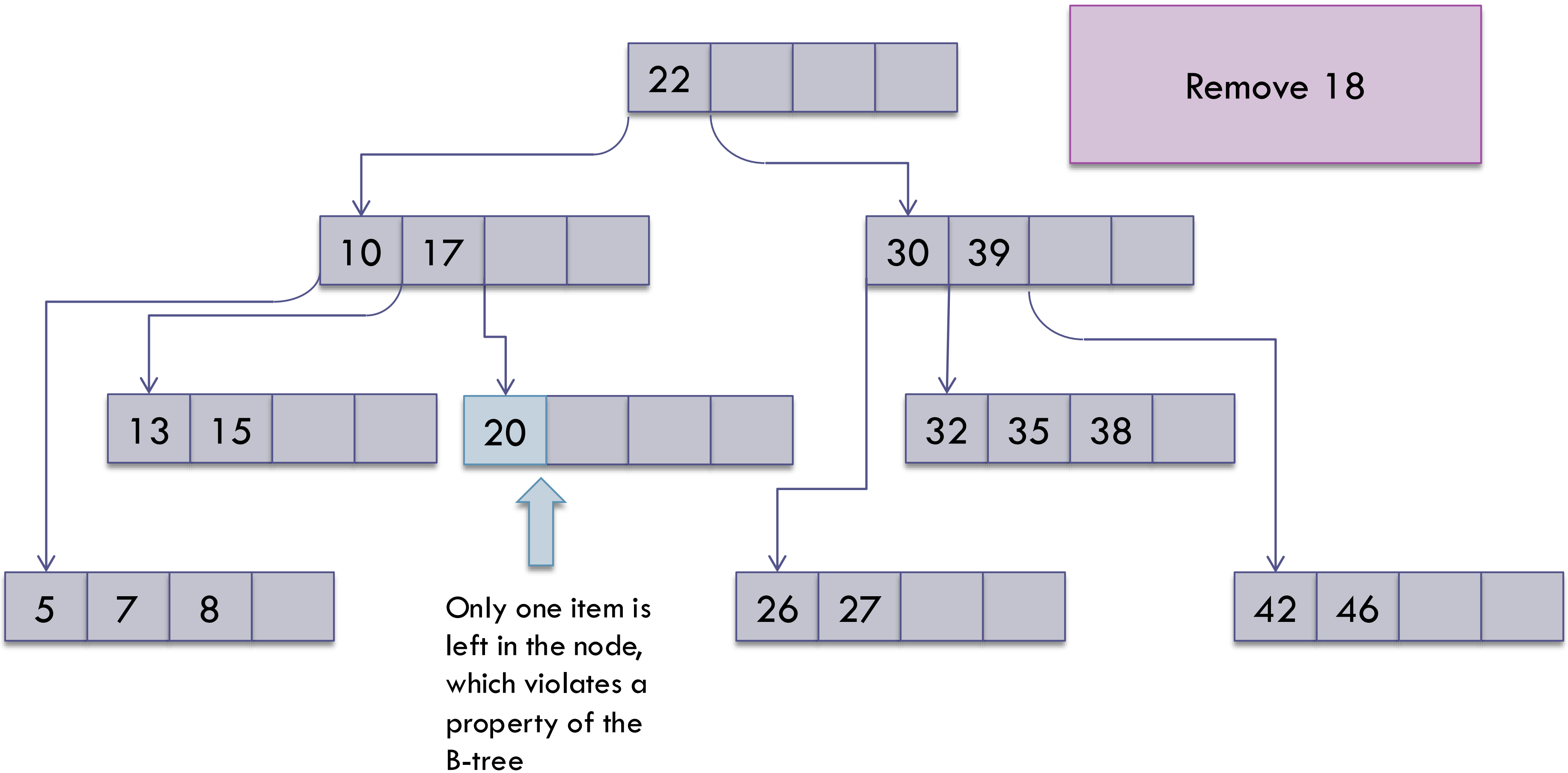


Removal from a B-Tree (cont.)

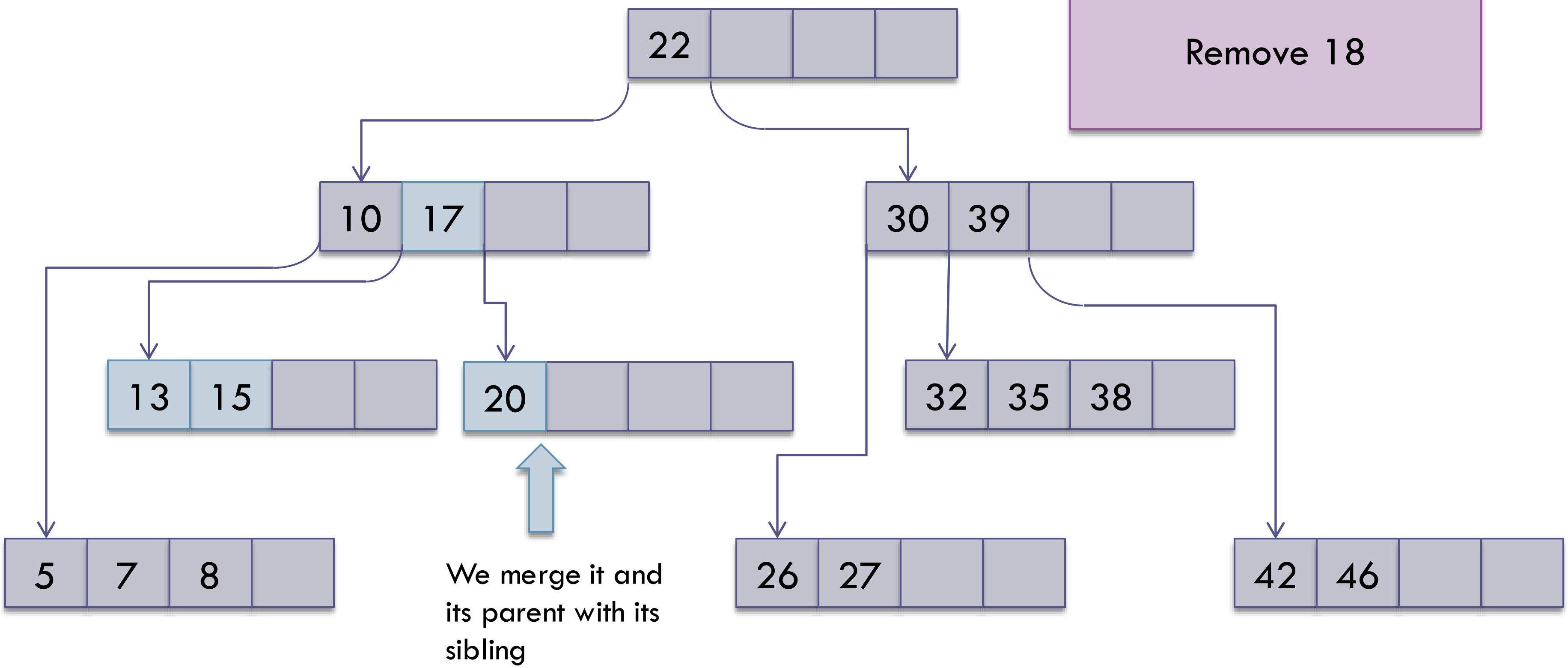


Remove 18

Removal from a B-Tree (cont.)

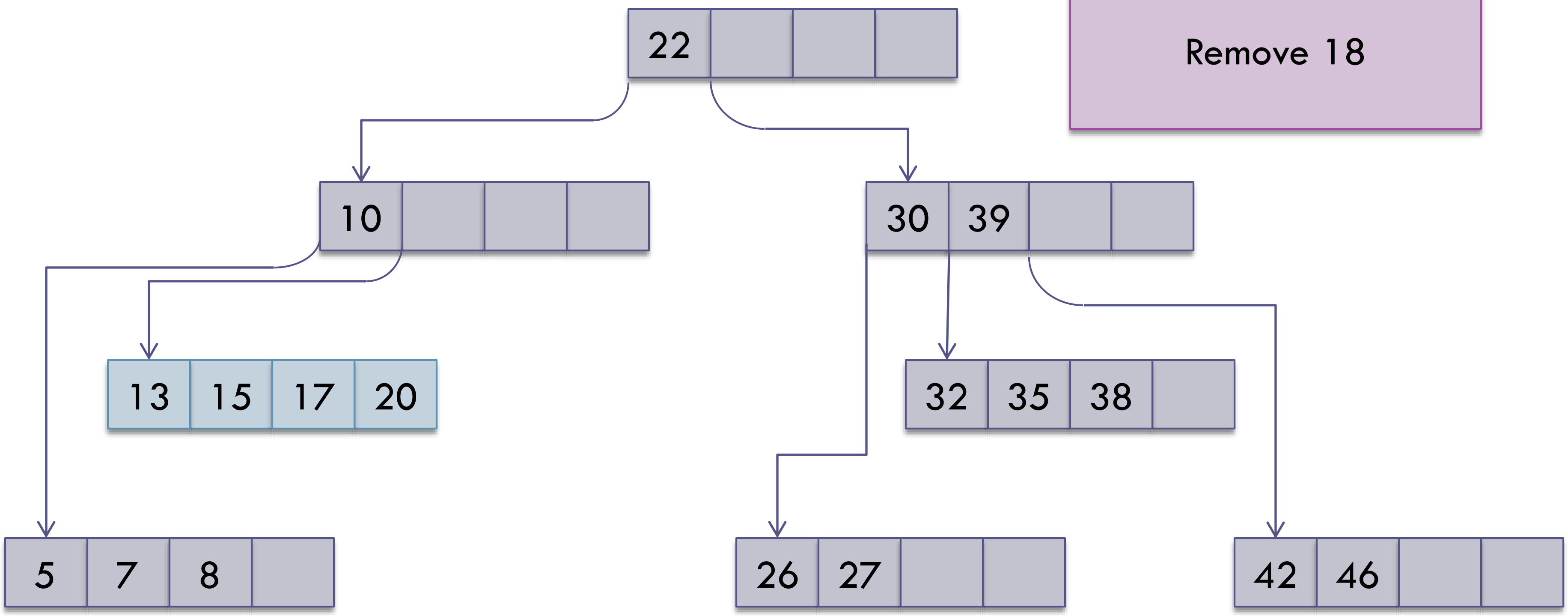


Removal from a B-Tree (cont.)



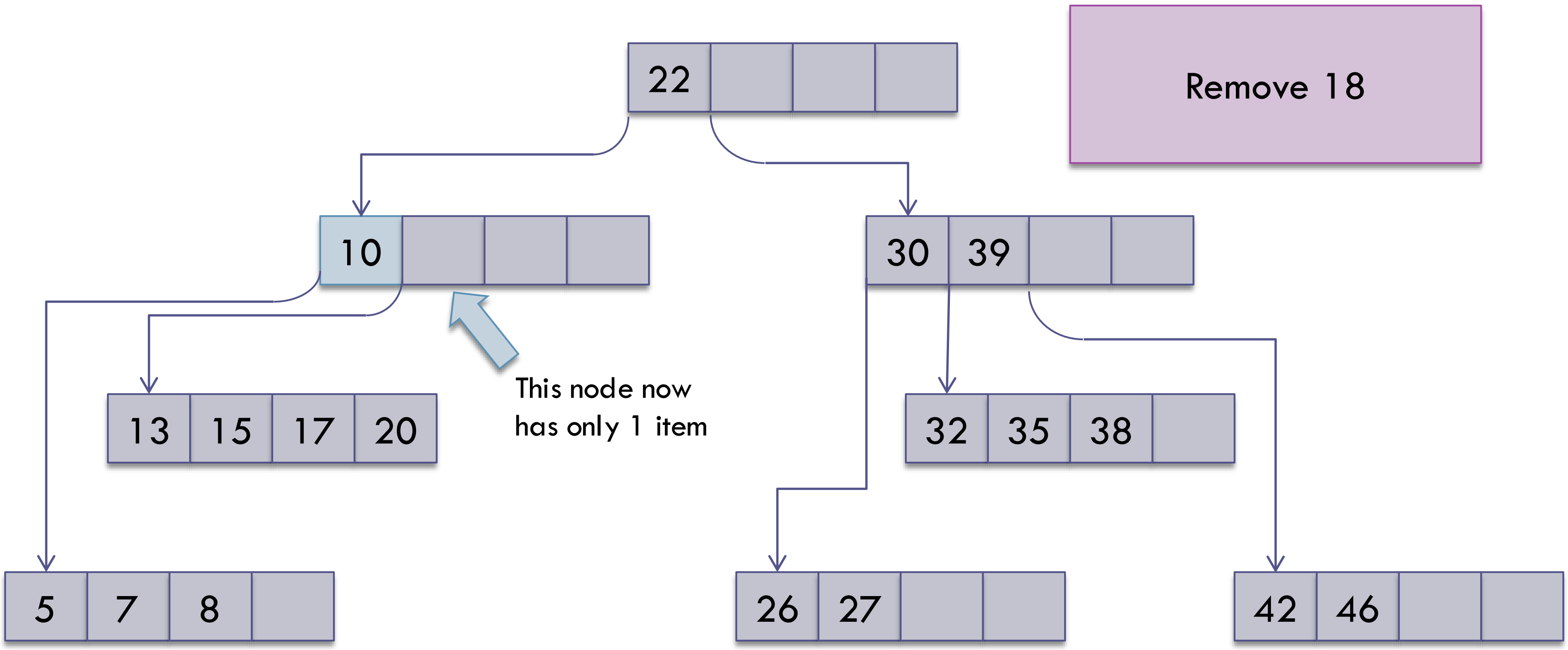
Remove 18

Removal from a B-Tree (cont.)

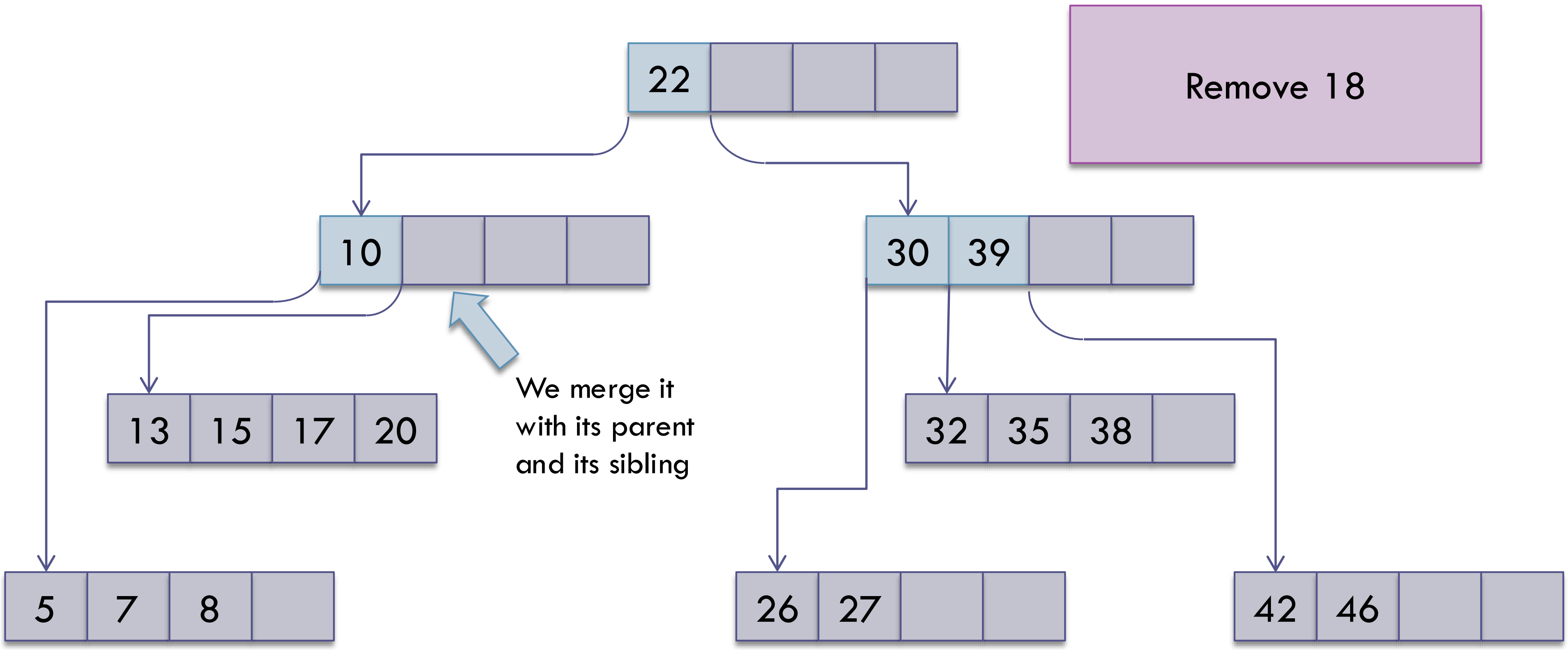


Remove 18

Removal from a B-Tree (cont.)

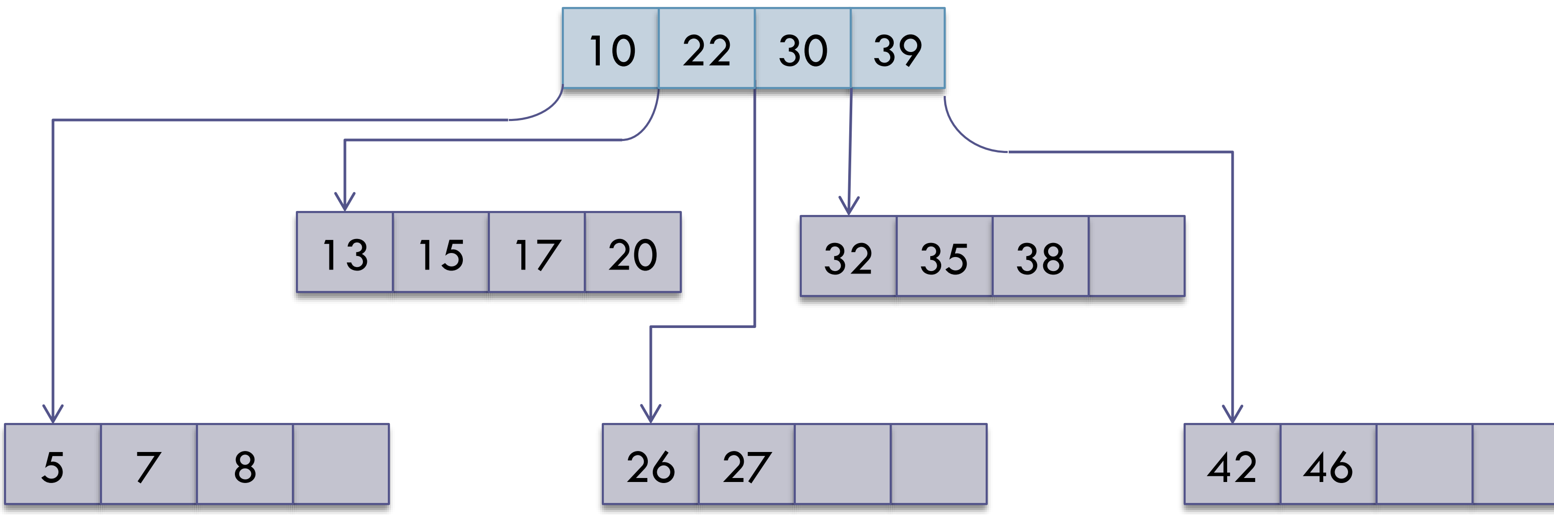


Removal from a B-Tree (cont.)

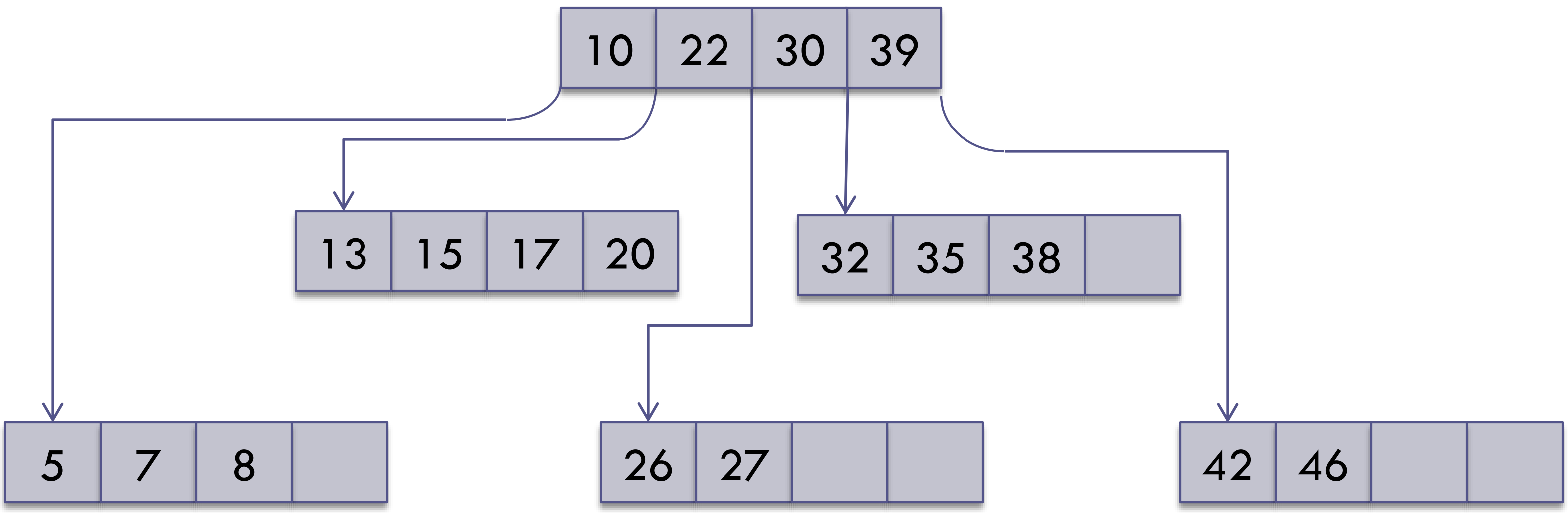


Removal from a B-Tree (cont.)

Remove 18



Removal from a B-Tree (cont.)



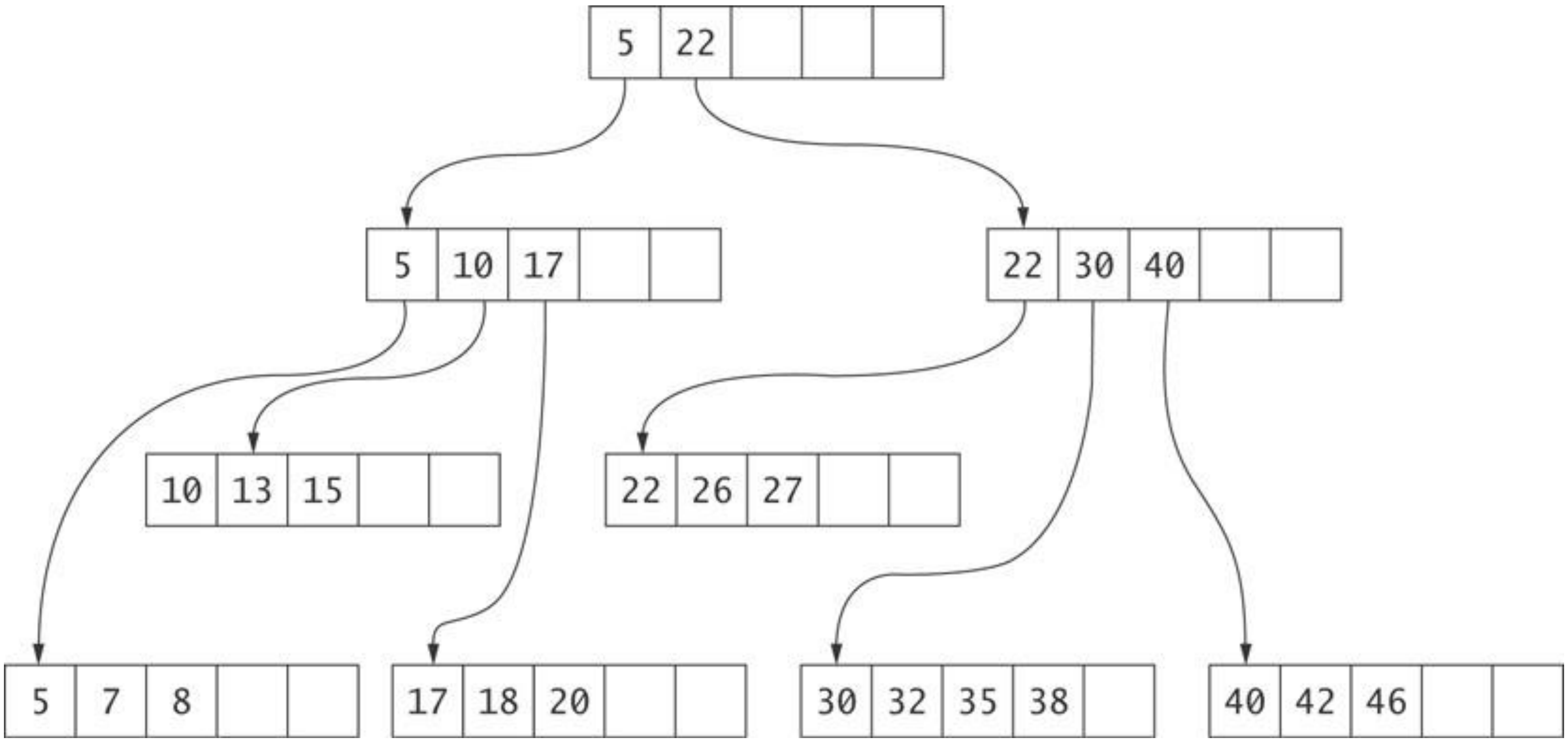
B+ Trees

- The B-tree was developed to create indexes for databases
  - the `Node` is stored on a disk block
  - the `Node` pointers are pointers to disk blocks instead of memory addresses
  - the `E` is a key-value pair where the value is also a pointer to a disk block
- Since in the leaf nodes all child pointers are null, there is a significant waste of space

B+ Trees (cont.)

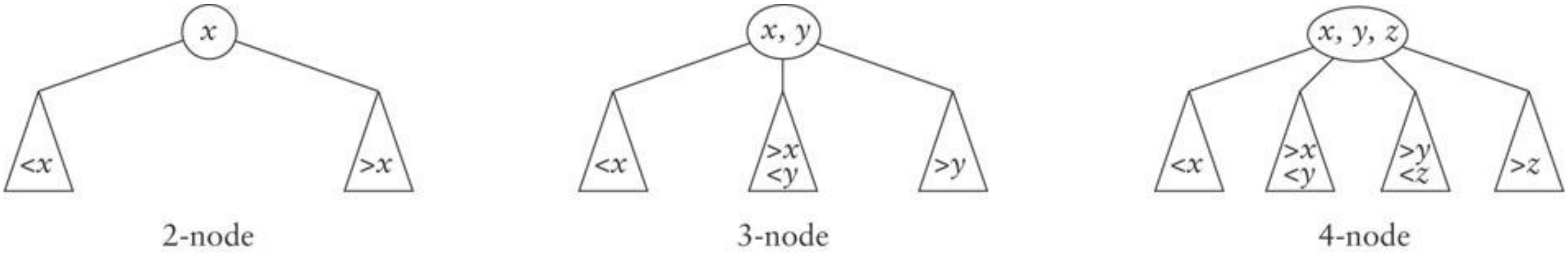
- A B+ tree addresses this wasted space
- In a B+ tree,
  - the leaves contain the keys and pointers to their corresponding values
  - the internal nodes contain only keys and pointers to the children
  - the parent's value is repeated as the first value
  - there are `order` pointers and `order` values

B+ Trees (cont.)

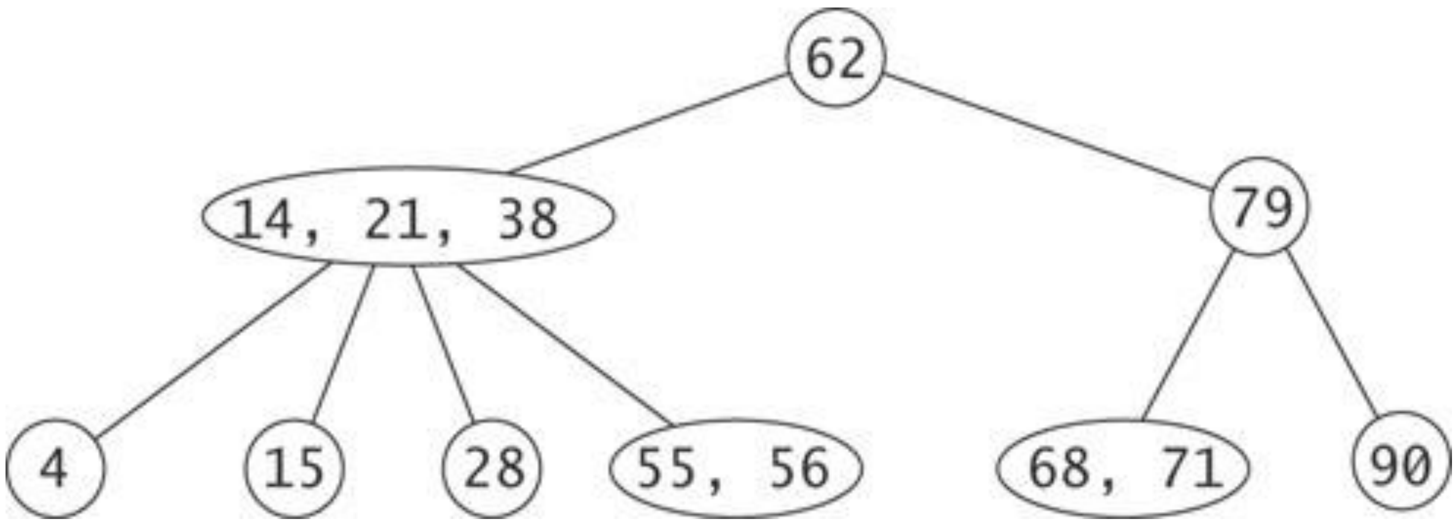


2-3-4 Trees

- 2-3-4 trees are a special case of the B-tree where order is fixed at 4
- A node in a 2-3-4 tree is called a 4-node
- A 4-node has space for three data items and four children



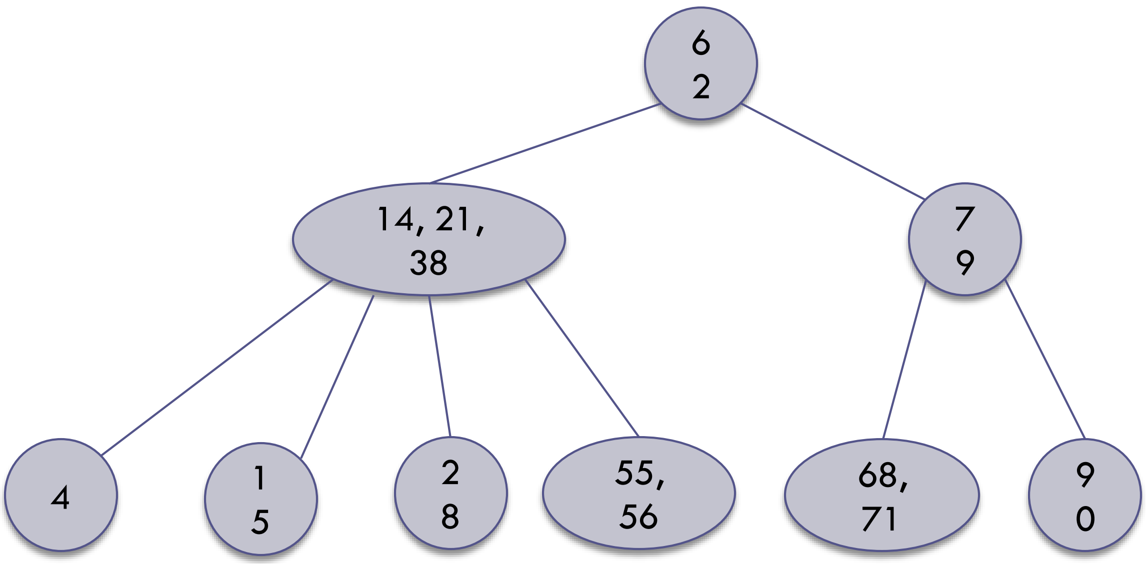
2-3-4 Tree Example



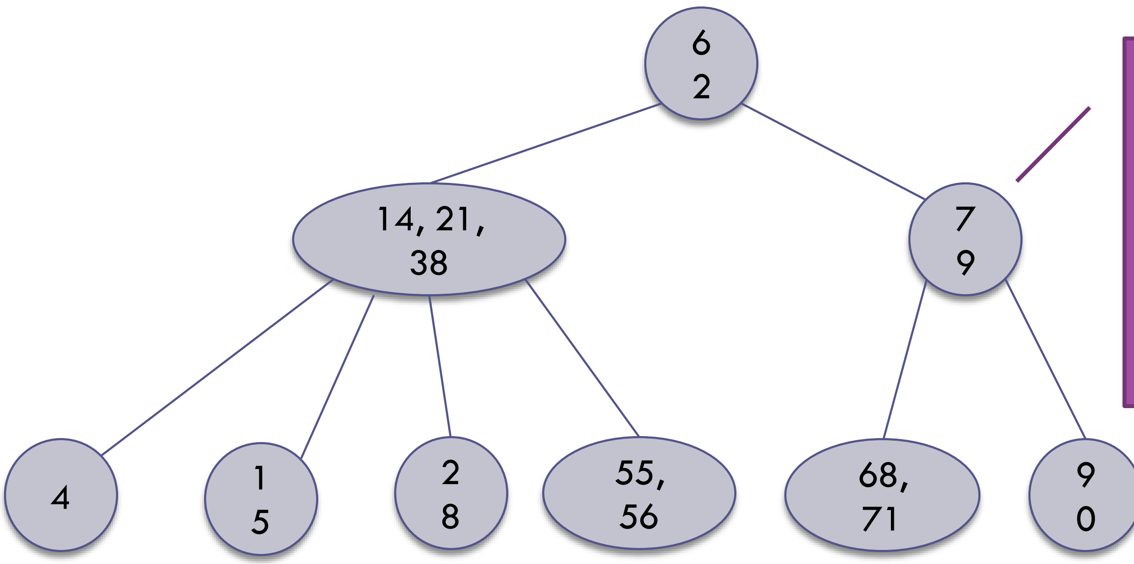
2-3-4 Trees (cont.)

- Fixing the capacity of a node at three data items simplifies the insertion logic
- A search for a leaf is the same as for a 2-3 tree or B-tree
- If a 4-node is encountered, we split it
  - When we reach a leaf, we are guaranteed to find room to insert an item

Insertion into a 2-3-4 Tree

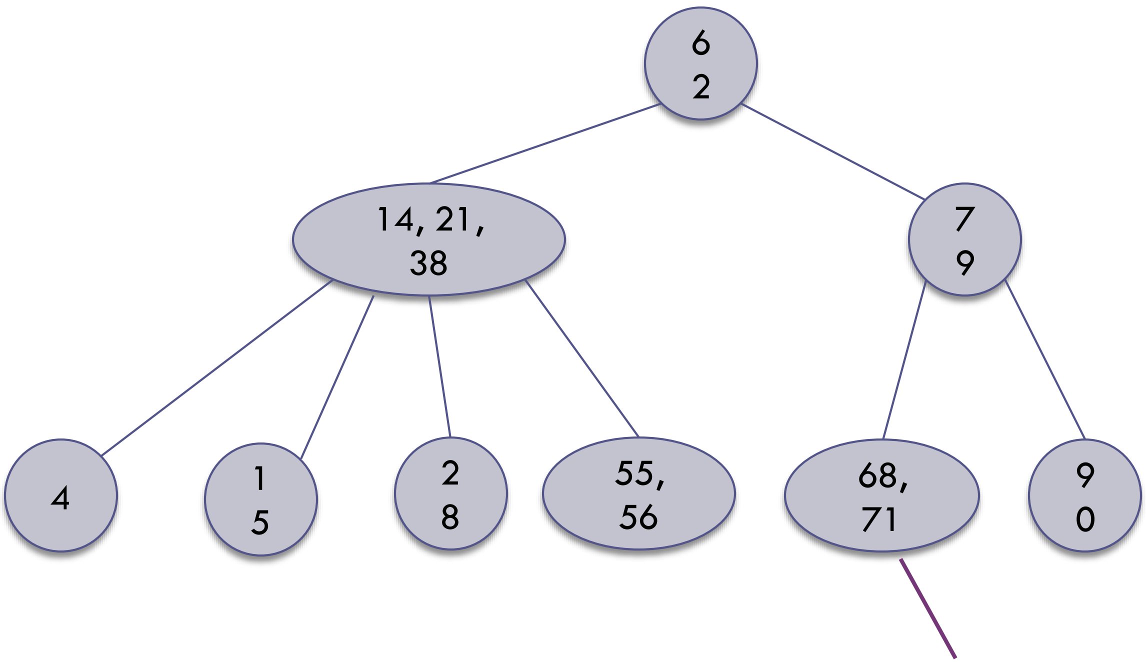


Insertion into a 2-3-4 Tree (cont.)



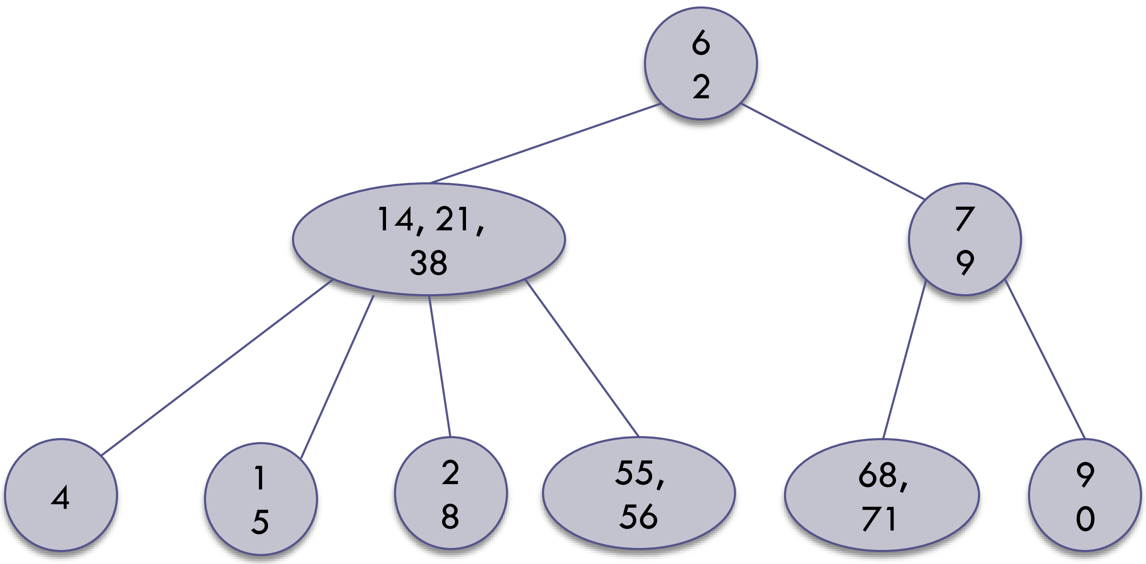
A number larger than 62 is inserted into one of this subtree's leaf nodes

Insertion into a 2-3-4 Tree (cont.)



A number between 63 and 78, inclusive, is inserted into this 3-node making it a 4-node

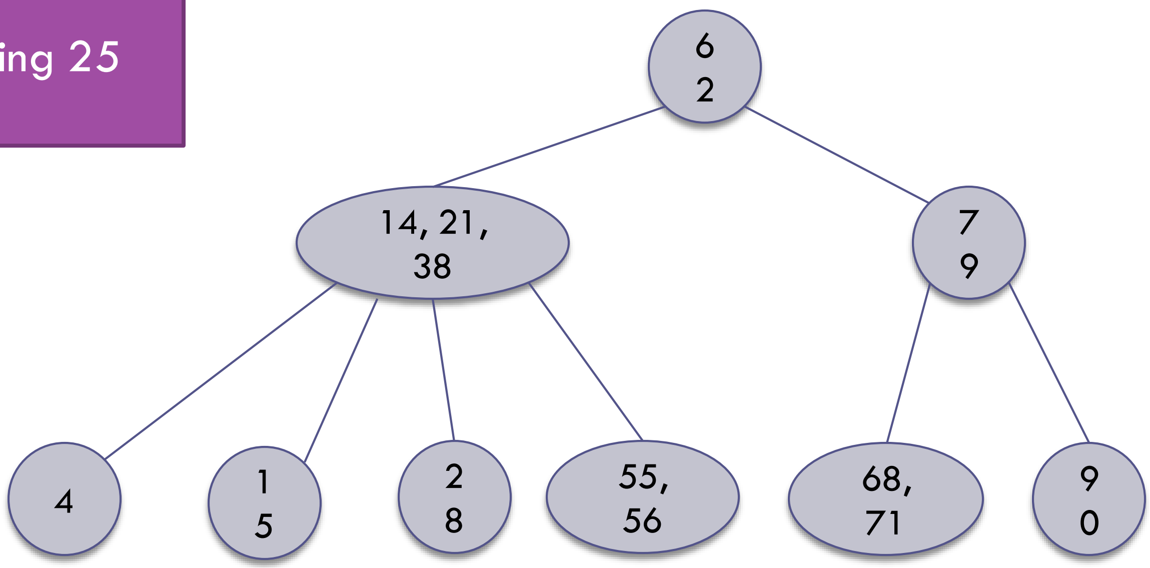
Insertion into a 2-3-4 Tree (cont.)



A number larger than 79 is inserted into this 2-node making it a 3-node

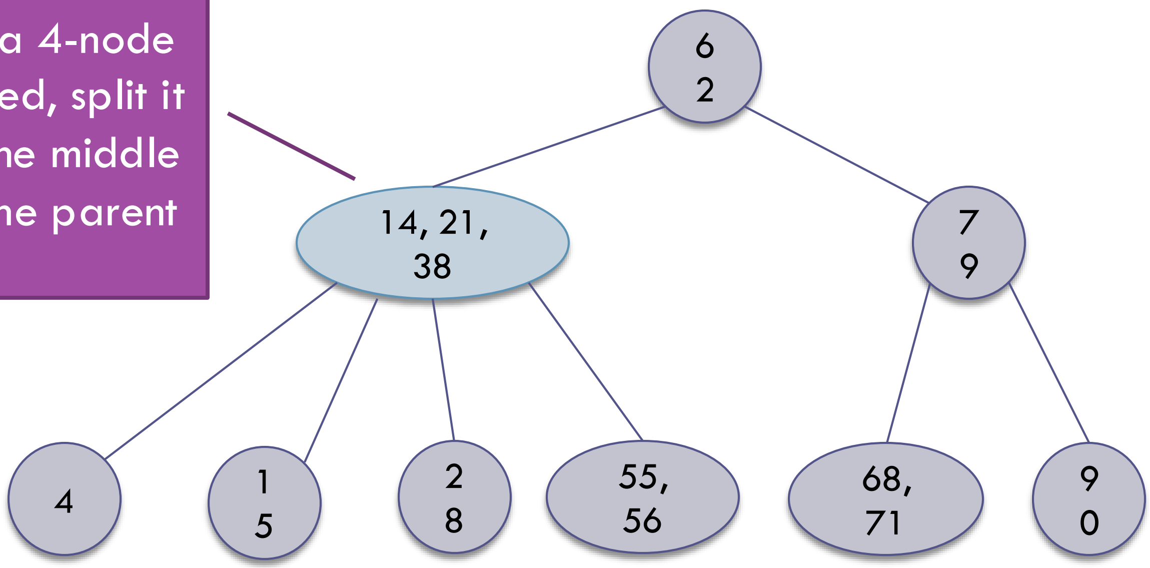
Insertion into a 2-3-4 Tree (cont.)

Inserting 25



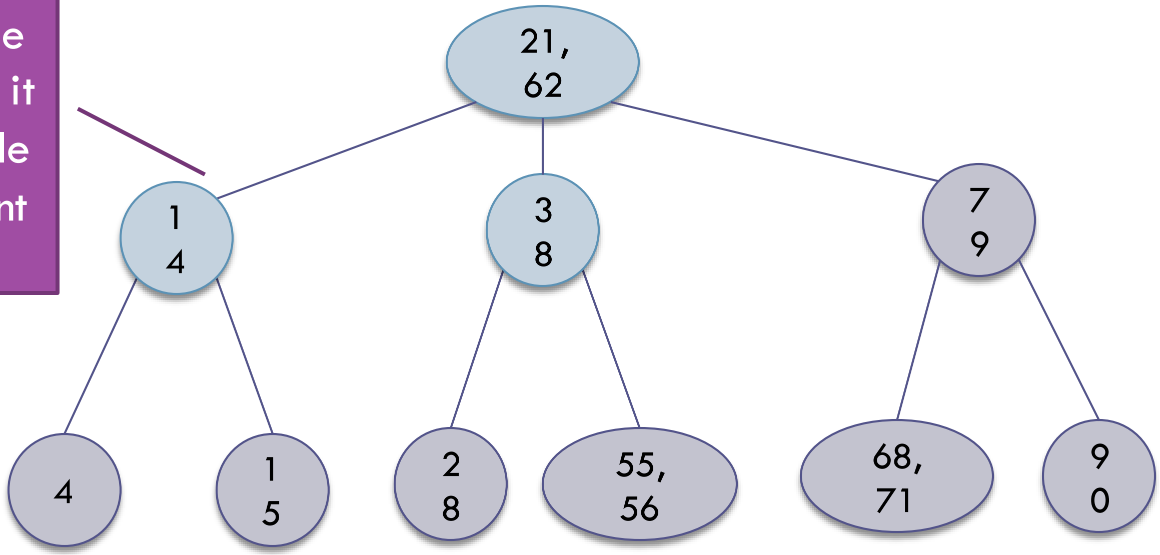
Insertion into a 2-3-4 Tree (cont.)

As soon as a 4-node is encountered, split it and move the middle value into the parent

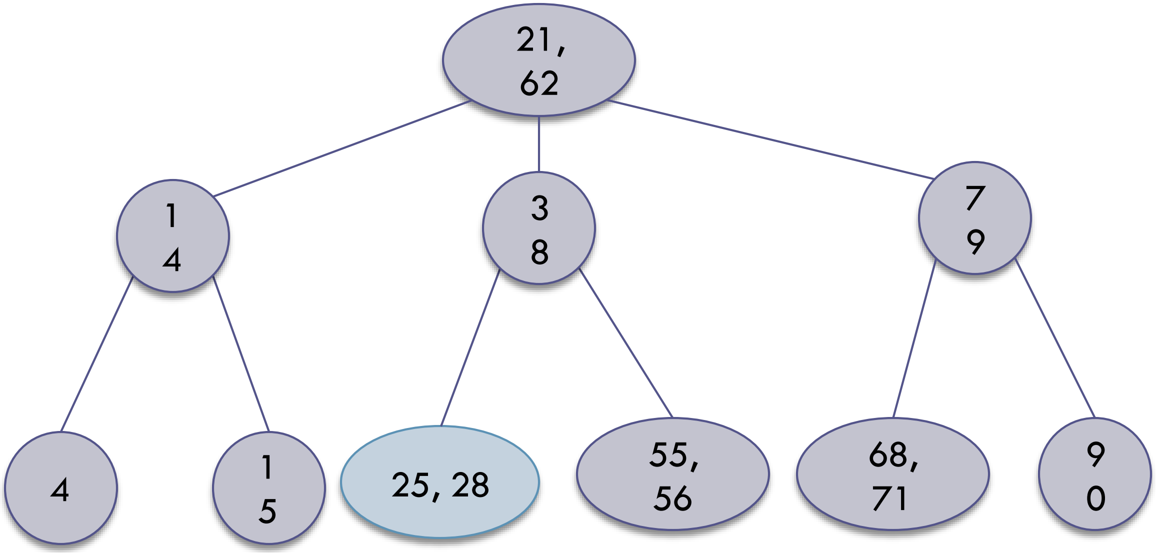


Insertion into a 2-3-4 Tree (cont.)

As soon as a 4-node is encountered, split it and move the middle value into the parent

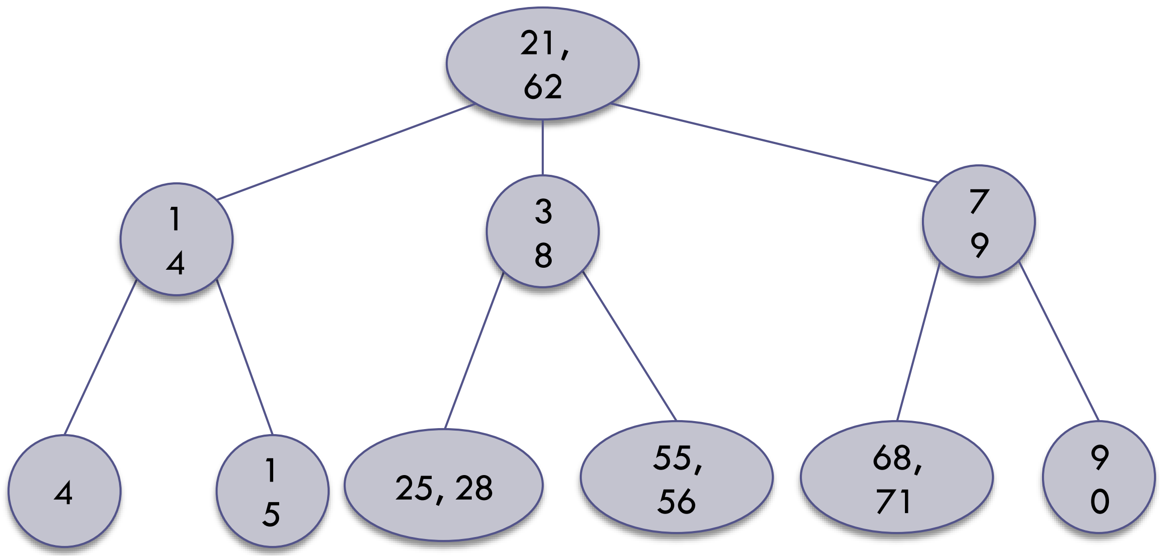


Insertion into a 2-3-4 Tree (cont.)



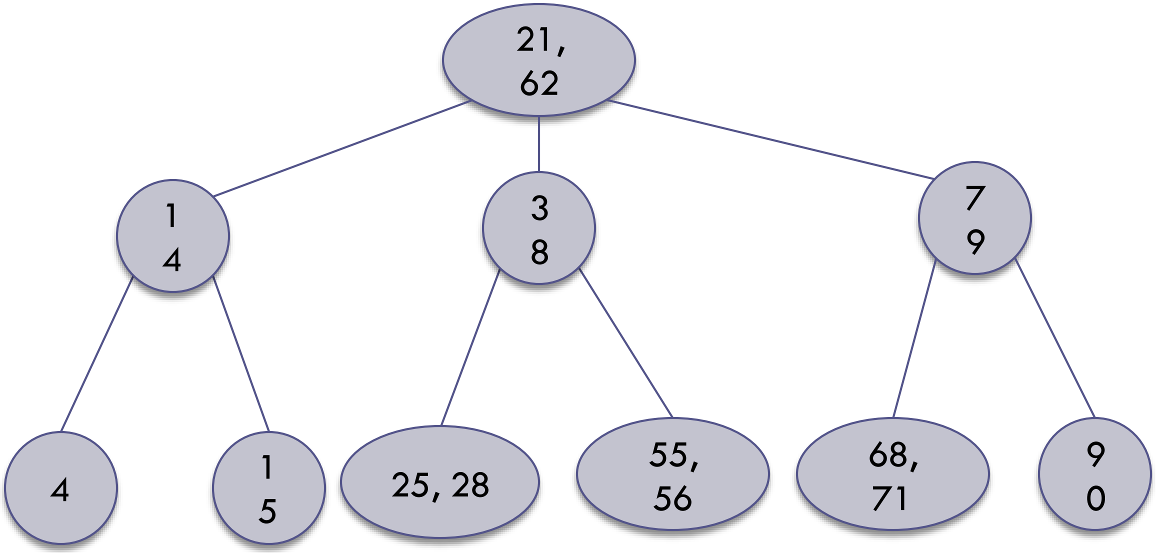
Then add the value (25) in a leaf node

Insertion into a 2-3-4 Tree (cont.)



This immediate split guarantees that a parent will not be a 4-node, and we will not need to propagate a child or its parent back up the recursion chain. The recursion becomes tail recursion.

Insertion into a 2-3-4 Tree (cont.)



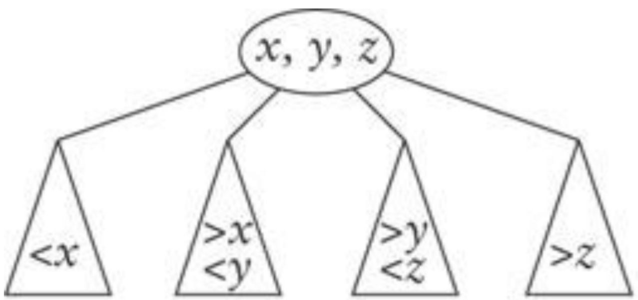
25 *could* have been inserted into the leaf node without splitting the parent 4-node, but always splitting a 4-node when it is encountered simplifies the algorithm with minimal impact on overall performance

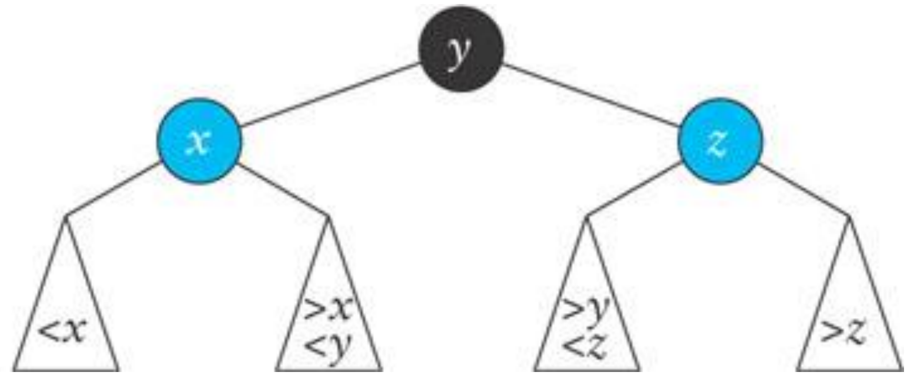
Relating 2-3-4 Trees to Red-Black Trees

- A Red-Black tree is a binary-tree equivalent of a 2-3-4 tree
- A 2-node is a black node

Relating 2-3-4 Trees to Red-Black Trees (cont.)

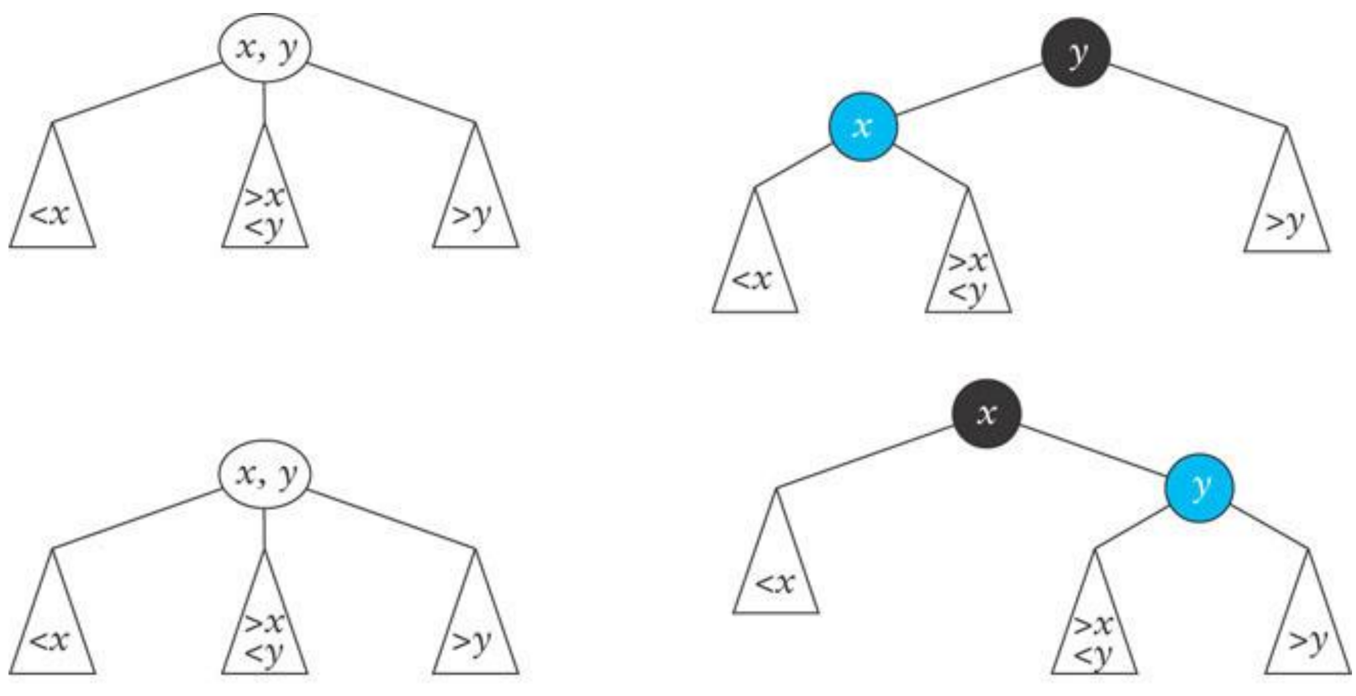
- A 4-node is a black node with two red (blue) children





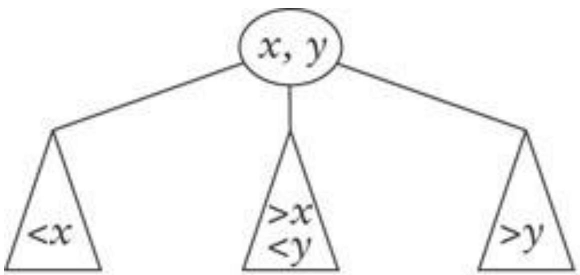
Relating 2-3-4 Trees to Red-Black Trees (cont.)

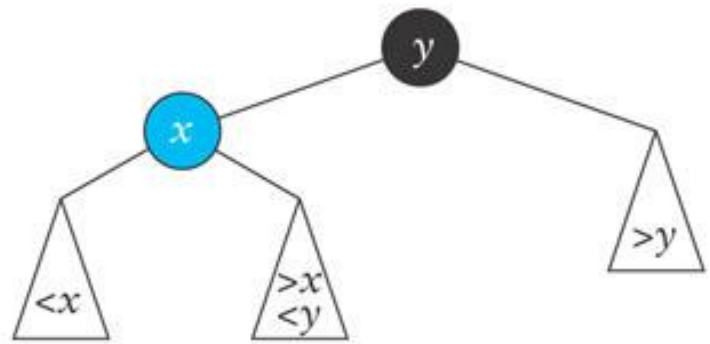
- A 3-node can be represented as either a black node with a left red (blue) child or a black node with a right red (blue) child



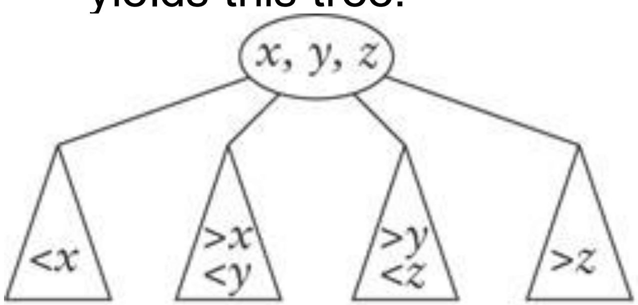
Relating 2-3-4 Trees to Red-Black Trees (cont.)

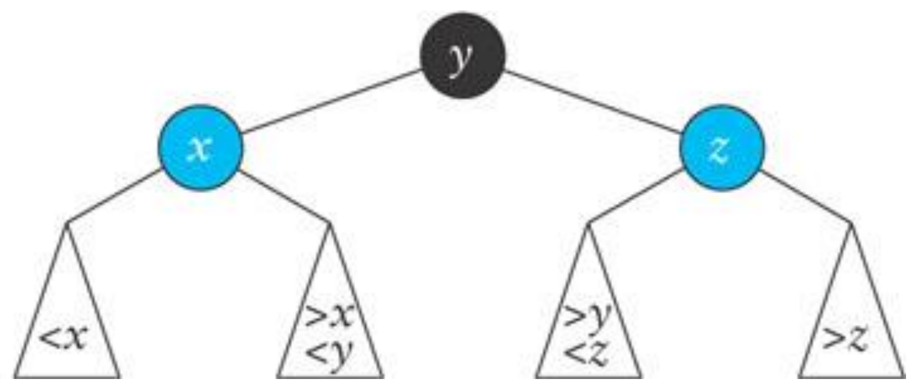
Inserting a value *z* greater than *y* in this tree:





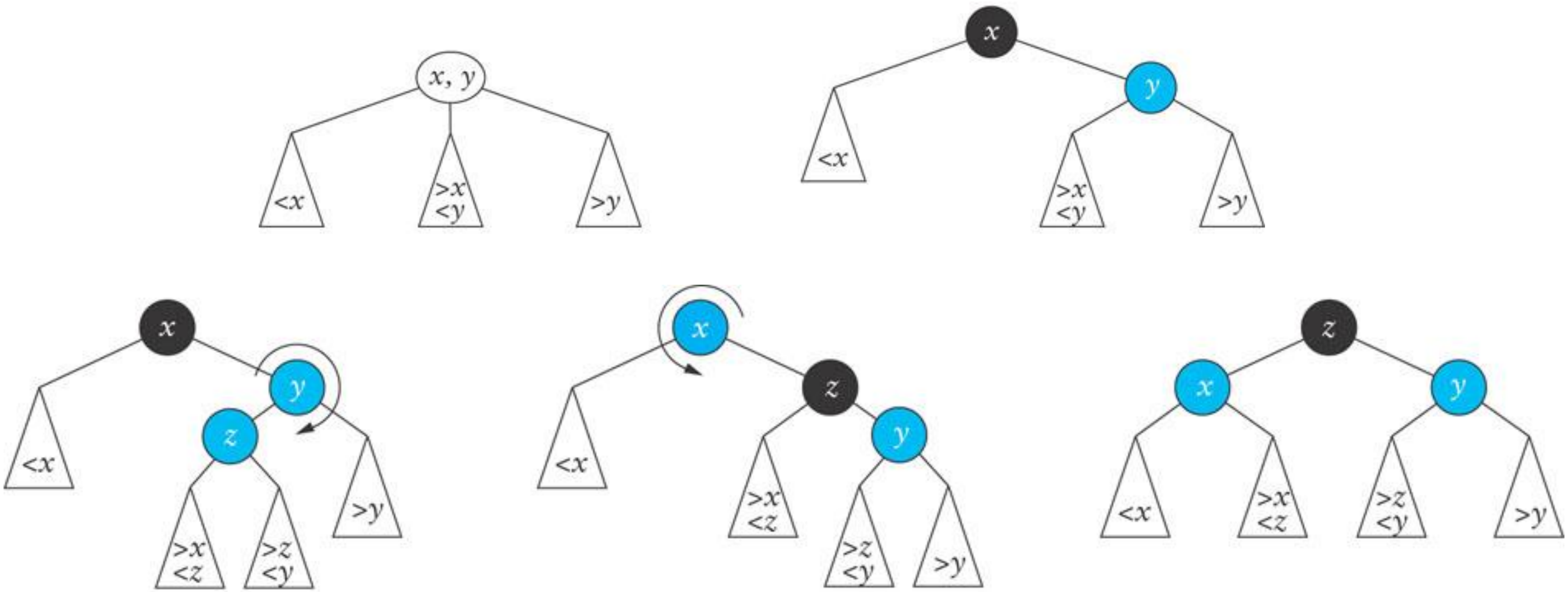
yields this tree:





Relating 2-3-4 Trees to Red-Black Trees (cont.)

Inserting value *z* that is between *x* and *y*



Skip-Lists

Section 9.6

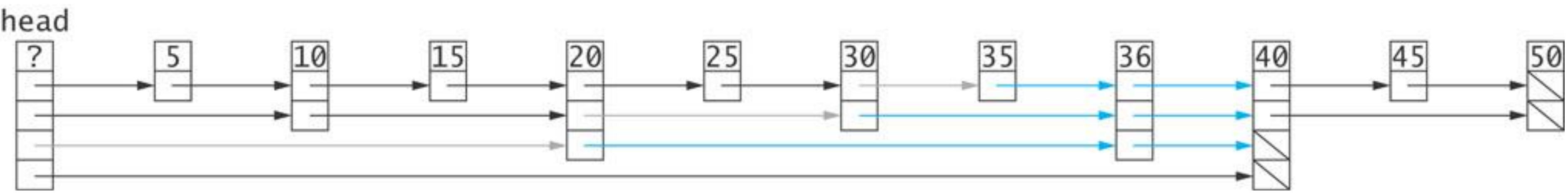
Skip-Lists

- A skip-list is another data structure that can be used as the basis for the `NavigableSet` or `NavigableMap` and as a substitute for a balanced tree
- It provides O(log *n*) search, insert, and remove
- It has the advantage over a Red-Black tree-based `TreeSet` in that concurrent references resulting from multiple threads are easier to achieve
- The concurrency features are beyond the scope of the course, and deal with multiple threads making modifications on a data set

Skip-List Structure

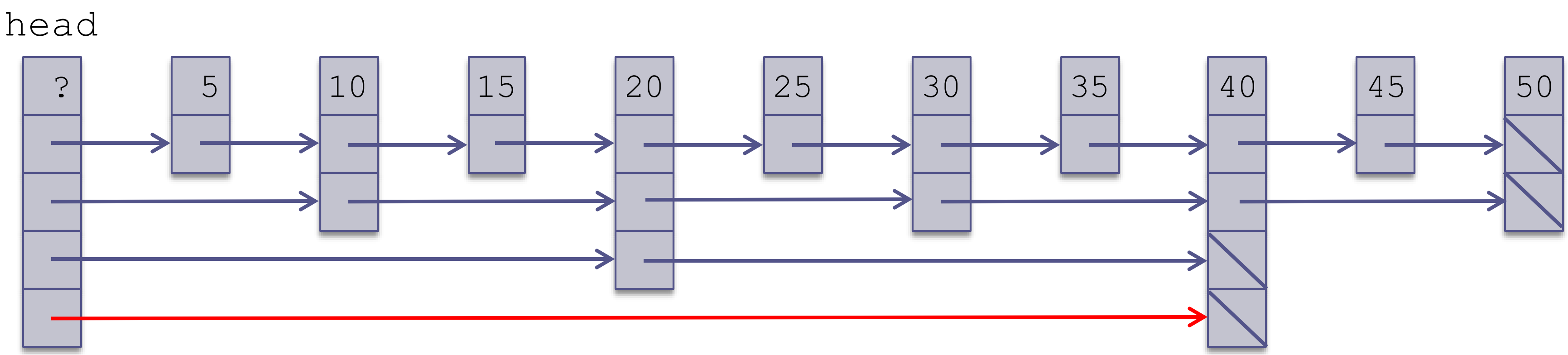
- A skip-list is a list of lists
  - Each node contains a data element with a key
  - The elements in each list are in increasing order by key
  - The nodes can contain a varying number of forward links determined by the level of the node
  - A level-*m* node has *m* forward links
  - The level of a new node is chosen randomly in such a way that, for a 4-level skip-list, approximately 50% are level 1 (one forward link), 25% are level 2 (2 forward links), 12.5% are level 3 (3 forward links), and so on

Skip-List Structure (cont.)



- The level of a skip-list is defined as its highest node level, or 4, in this list
- The list above is an *ideal skip-list*; most skip-lists will not have exactly this structure, but will behave similarly

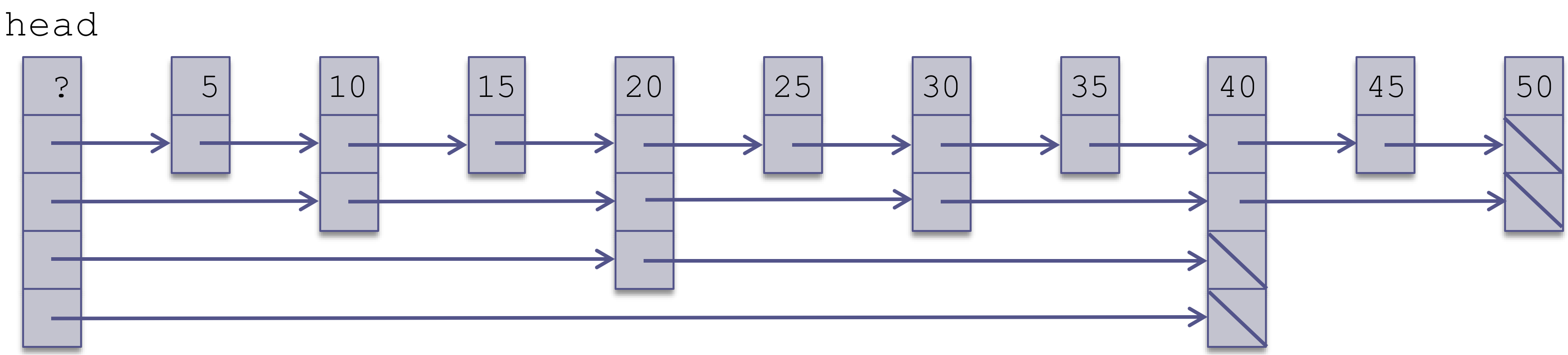
Searching a Skip-List



A search always begins in the highest level list (the list with the fewest elements)

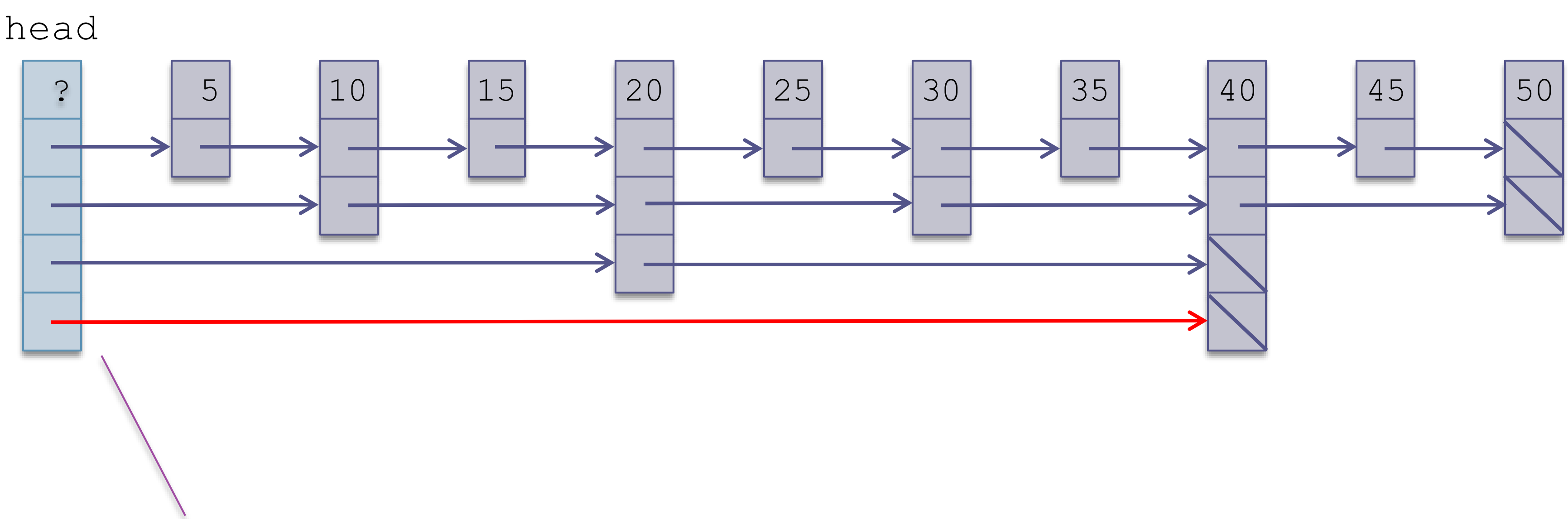
Searching a Skip-List (cont.)

Search for 35



Searching a Skip-List (cont.)

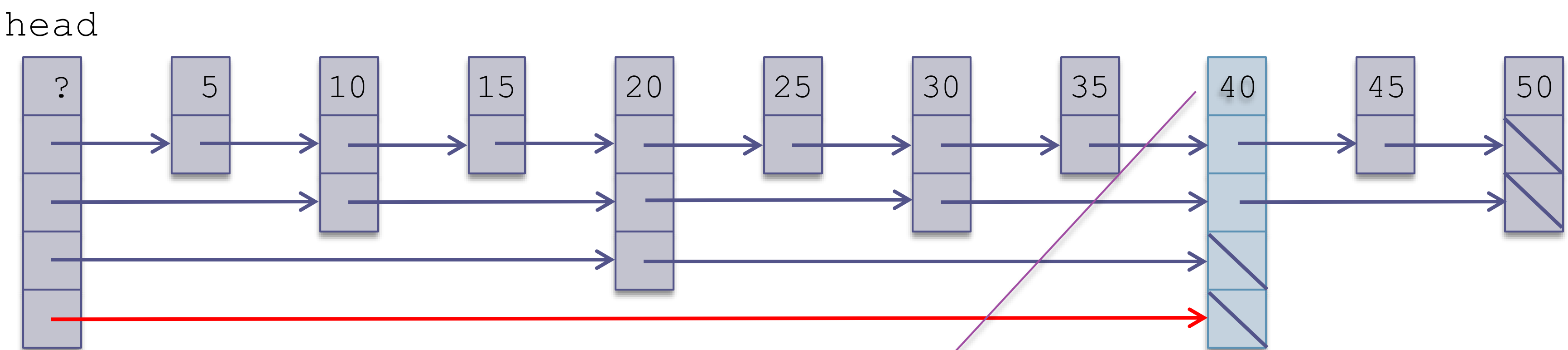
Search for 35



Start with the highest list, in this case, level 4

Searching a Skip-List (cont.)

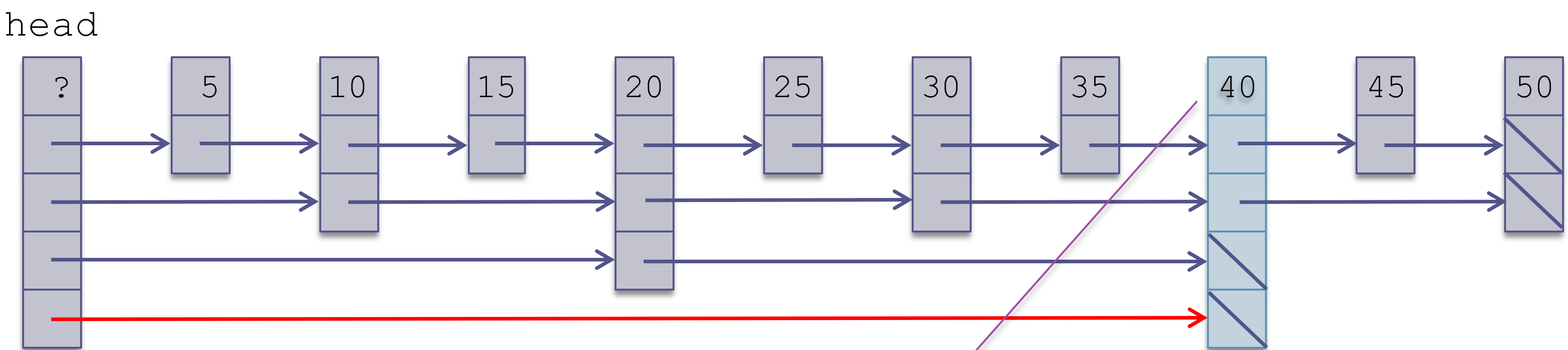
Search for 35



The value of this node is 40 (we have also reached the end of this list)

Searching a Skip-List (cont.)

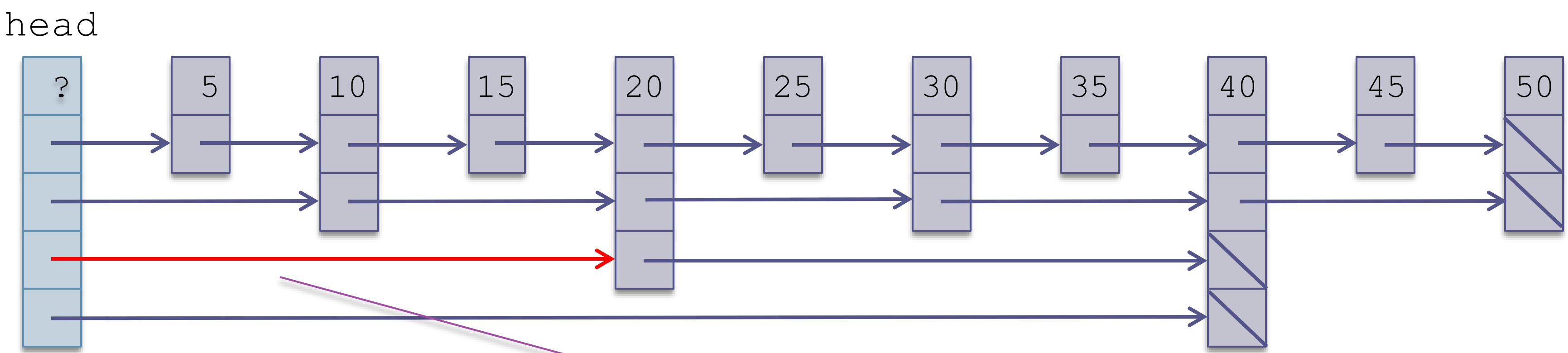
Search for 35



Since 35 < 40, we move back to the predecessor node and search the next lower level

Searching a Skip-List (cont.)

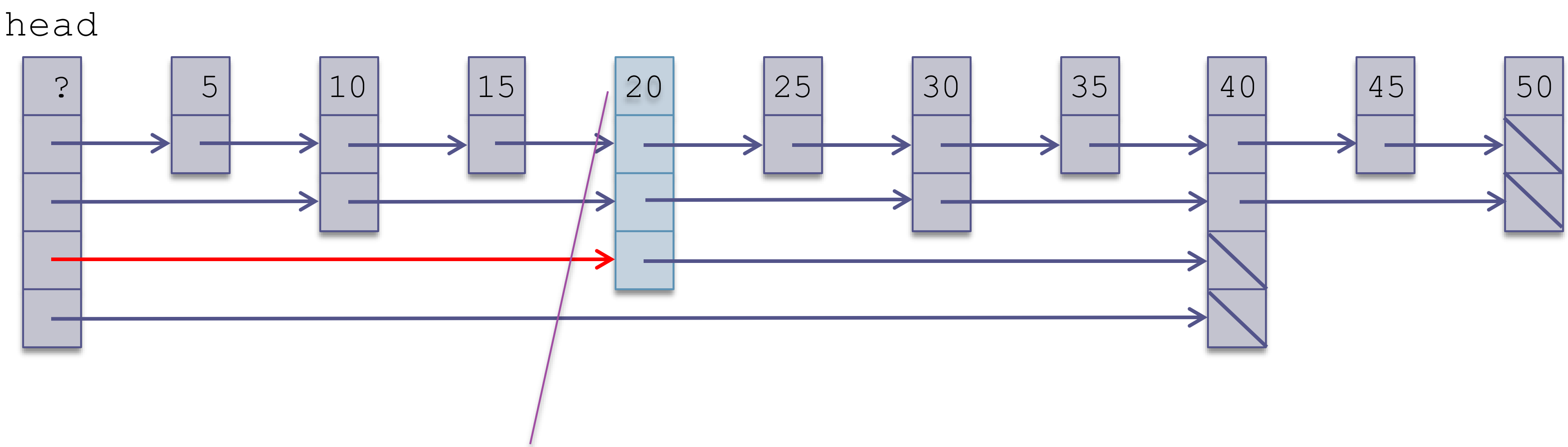
Search for 35



Since 35 < 40, we move back to the predecessor node and search the next lower level

Searching a Skip-List (cont.)

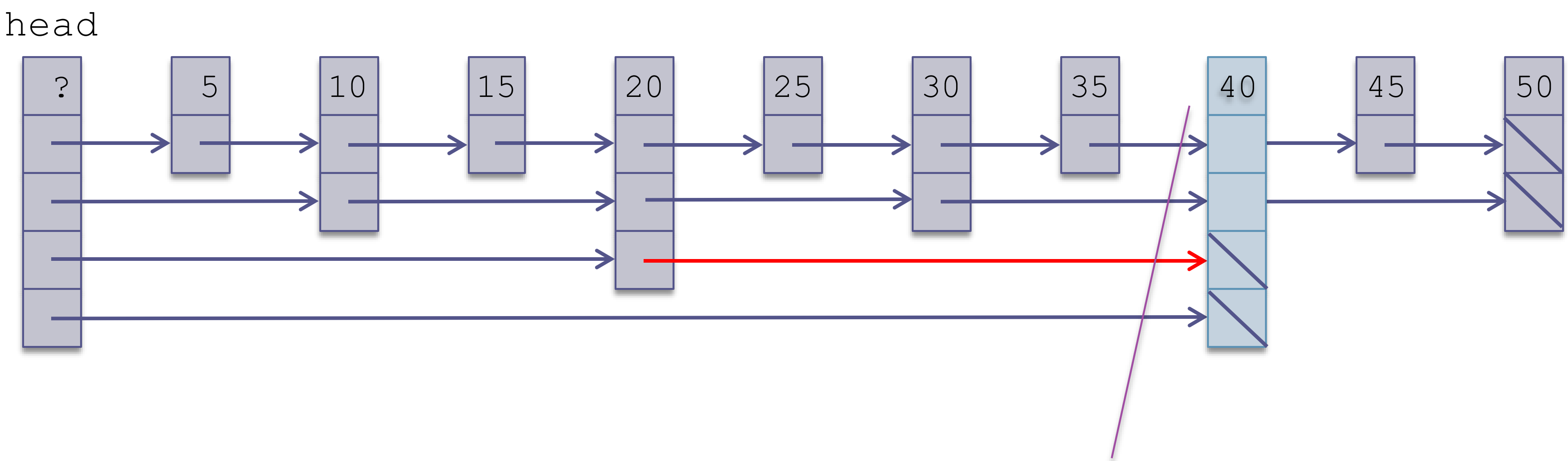
Search for 35



The value of the next node is 20. 35 > 20, so we move to the next node on this level

Searching a Skip-List (cont.)

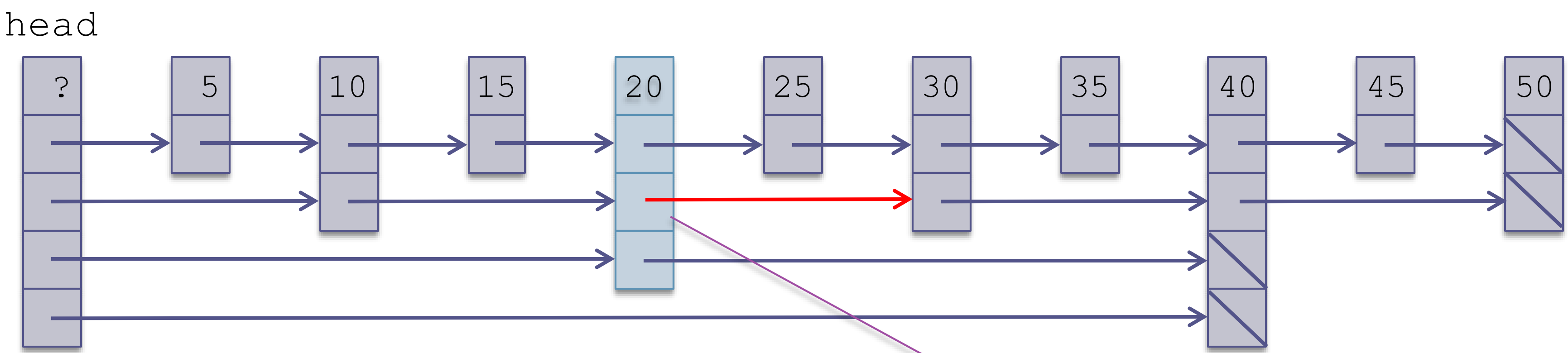
Search for 35



35 < 40 so we move to the predecessor and search the next lower level

Searching a Skip-List (cont.)

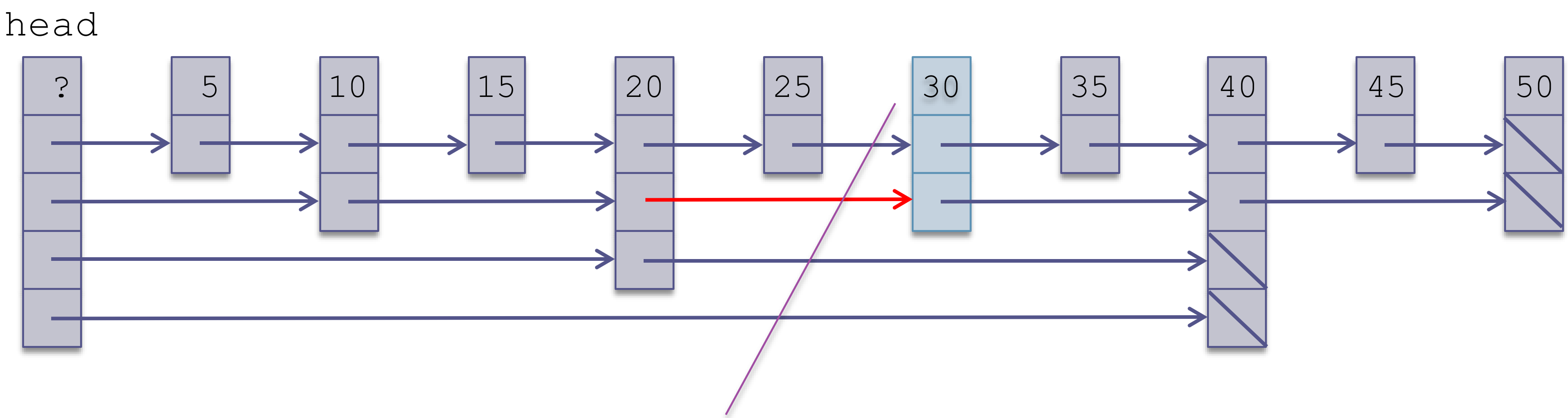
Search for 35



35 < 40 so we move to the predecessor and search the next lower level

Searching a Skip-List (cont.)

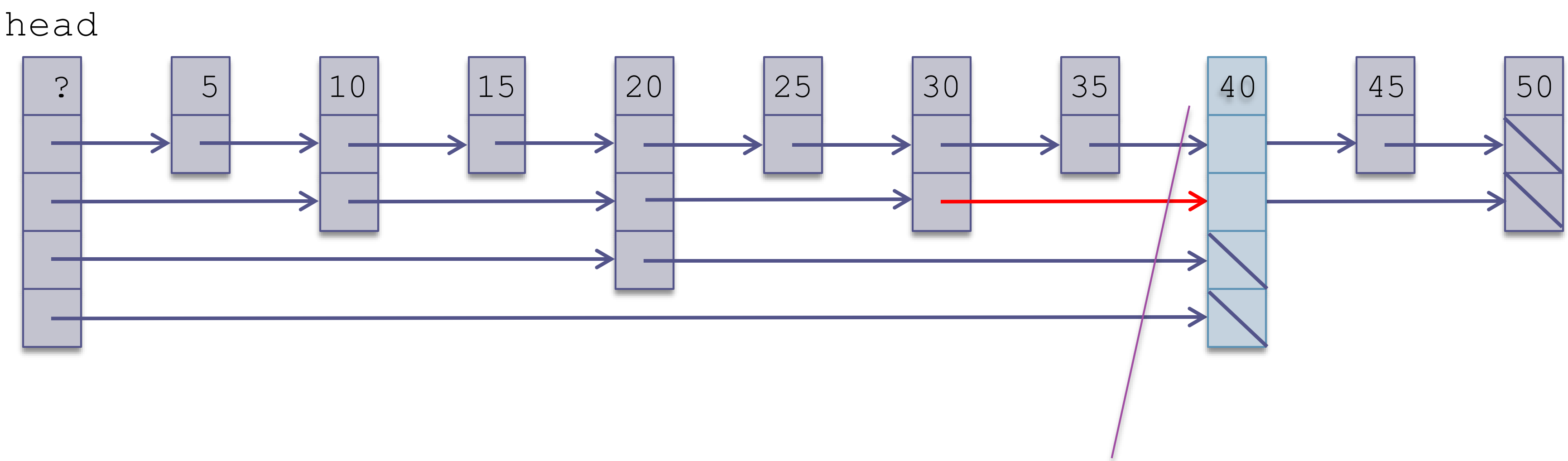
Search for 35



35 > 30, so we move to the next node

Searching a Skip-List (cont.)

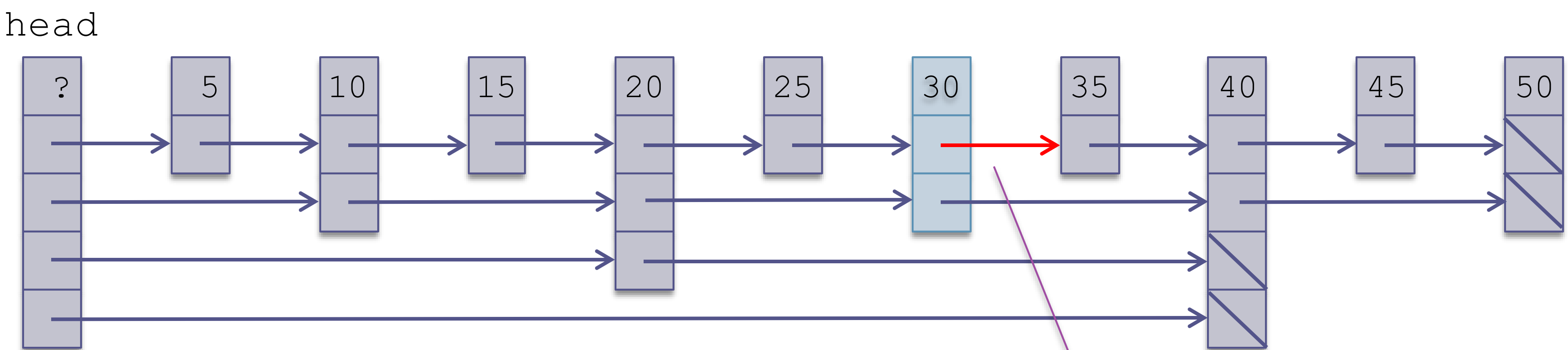
Search for 35



35 < 40, so we stop and move to the predecessor and search the next lower level

Searching a Skip-List (cont.)

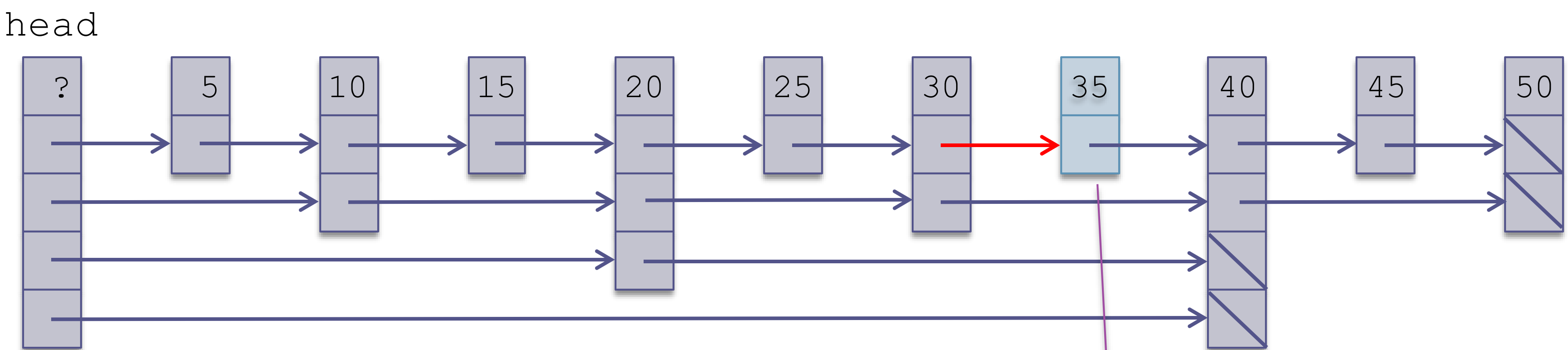
Search for 35



35 < 40, so we stop and move to the predecessor and search the next lower level

Searching a Skip-List (cont.)

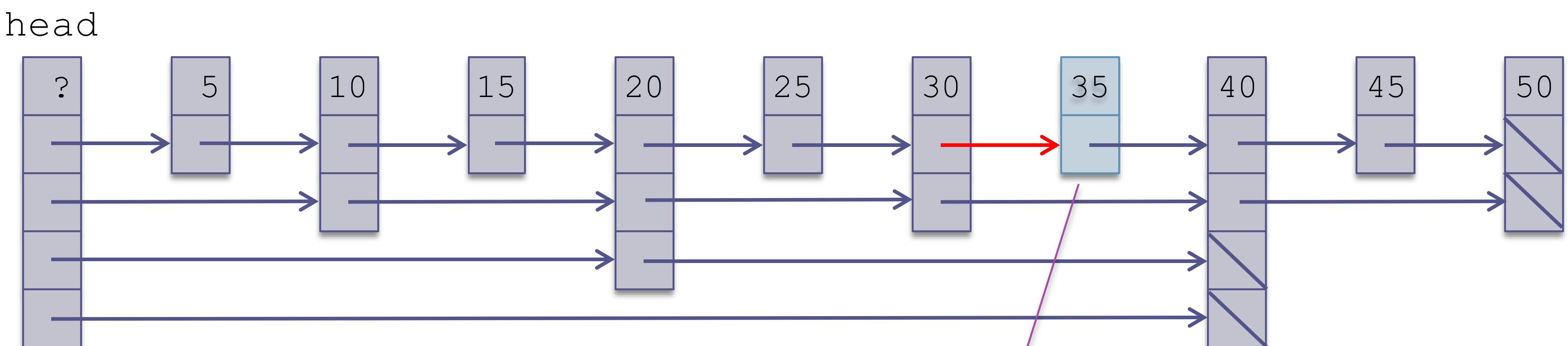
Search for 35



35 is found!

Searching a Skip-List (cont.)

Search for 35



If we reach a value greater than our search target on the lowest level, the search target is not in the list

Searching a Skip-List Algorithm

1. Let *m be the highest-level node.*
2. **while** *m > 0*
3. Following the level-*m* links, find the node with the largest value that is less than or equal to the target.
4. If it is equal to the target, the target has been found—exit loop.
5. Set *m to m - 1*
6. If *m = 0, the target is not in the list.*

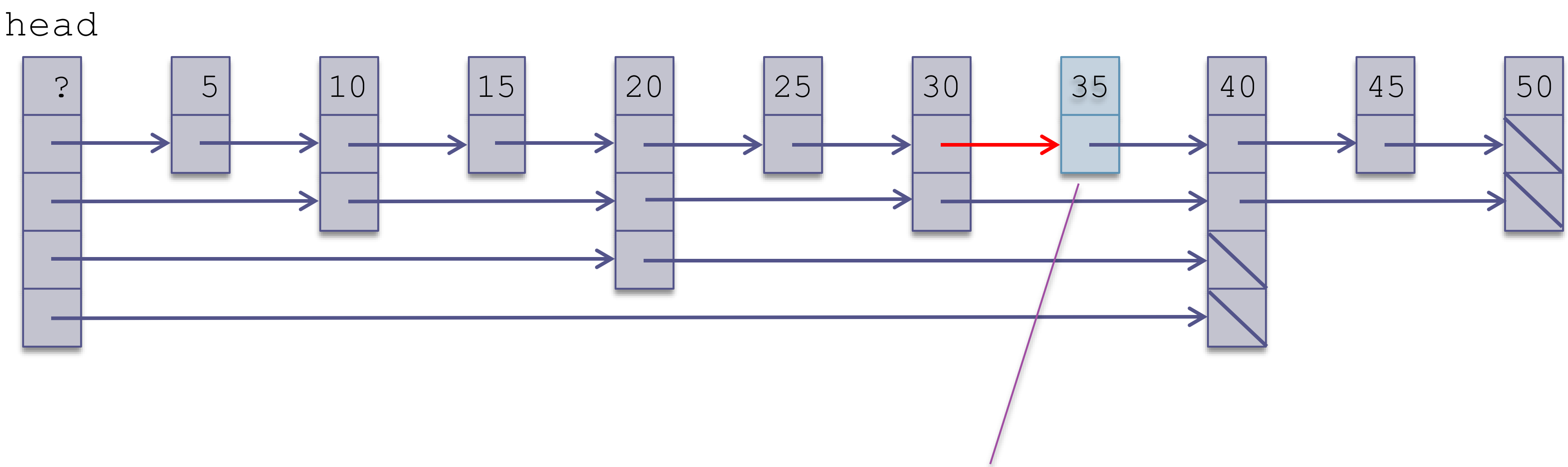
Performance of a Skip-List Search

- Because the first list searched has the fewest elements, and each subsequent-level list has approximately half as many elements as the current list, the search performance is O(log n), which is similar to that of a binary search

Insertion into a Skip-List

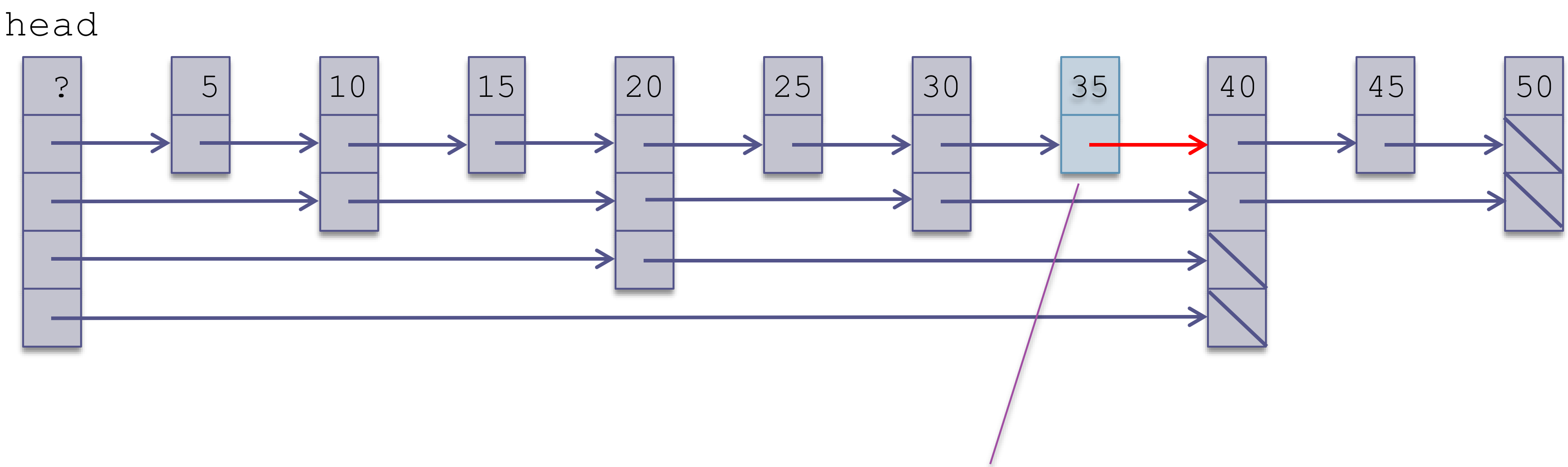
- If the search algorithm fails to find the target, it will find its predecessor in the level-1 list, which is the target's insertion point
- While we know the insertion point, we need to determine the level of the new node
- The level is chosen at random based on the number of items currently in the skip-list
- The random number is chosen with a logarithmic distribution; for a level-4 skip-list
  - half the time a level-1 node is chosen
  - a quarter of the time a level-2 node is chosen
  - And, generally, $1/2^m$ of the time a level-*m* node is chosen

Insertion into a Skip-List (cont.)



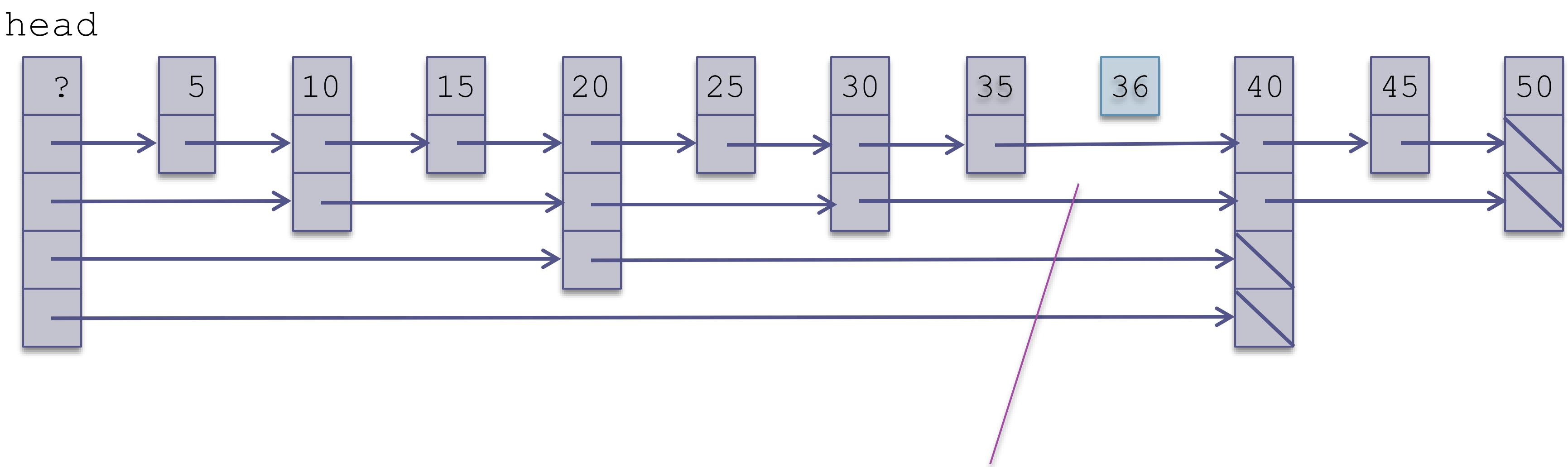
Previously we searched for 35. The same search would be run to insert 36

Insertion into a Skip-List (cont.)



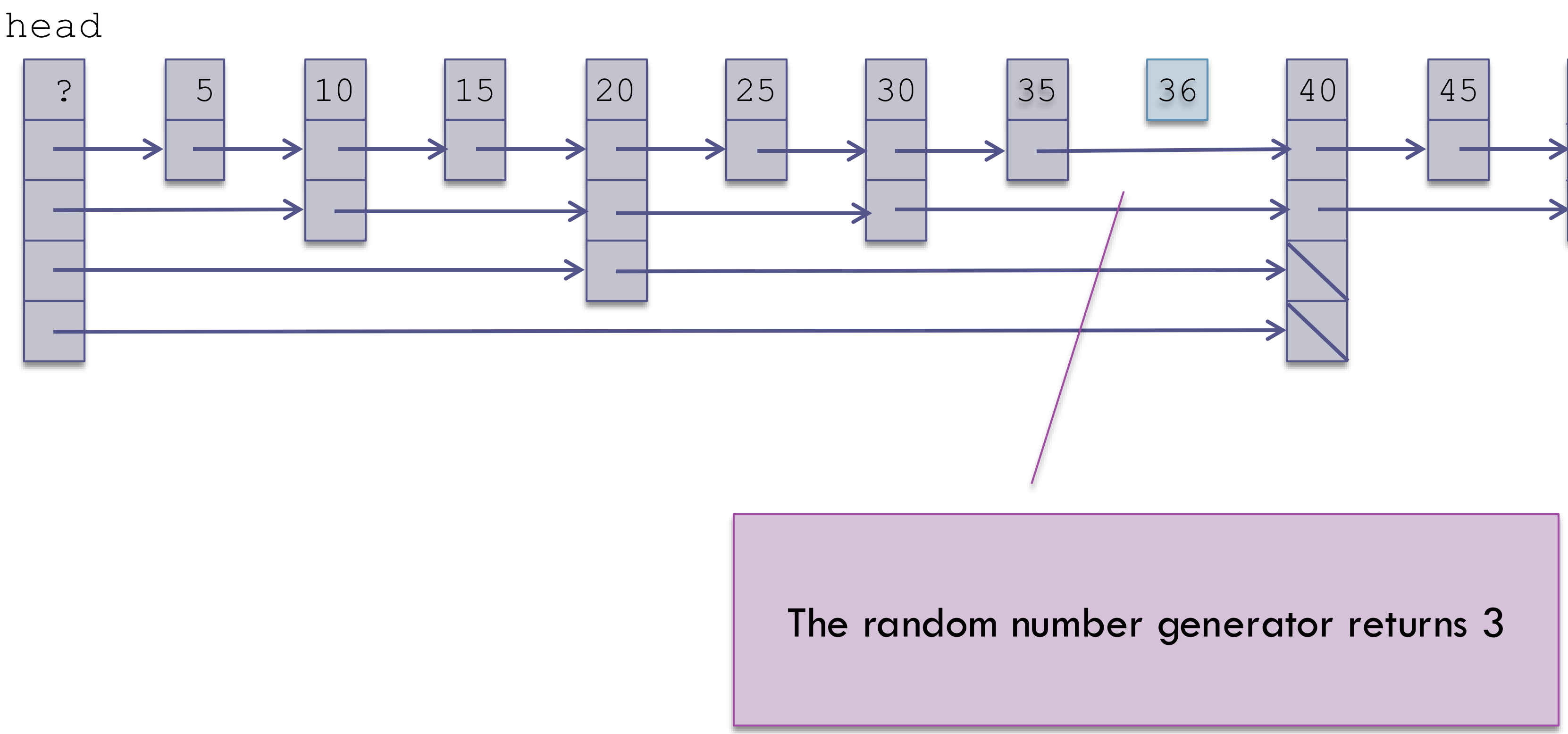
The next node's value is 40, which is greater than 36, so the insertion point is after predecessor (35)

Insertion into a Skip-List (cont.)

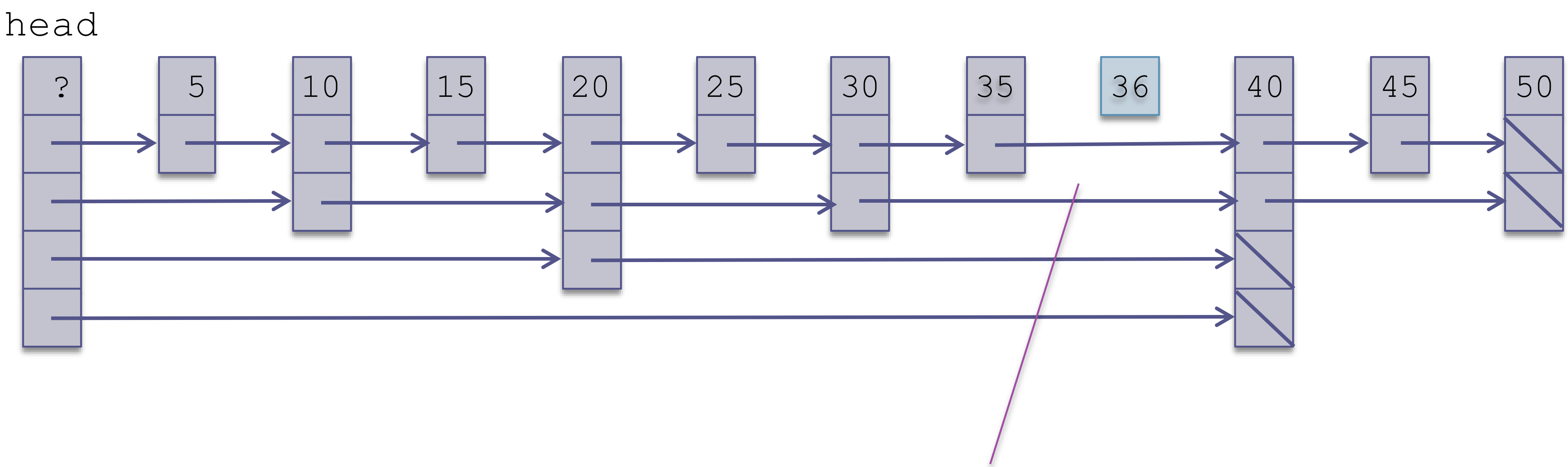


The next node's value is 40, which is greater than 36, so the insertion point is after predecessor (35)

Insertion into a Skip-List (cont.)

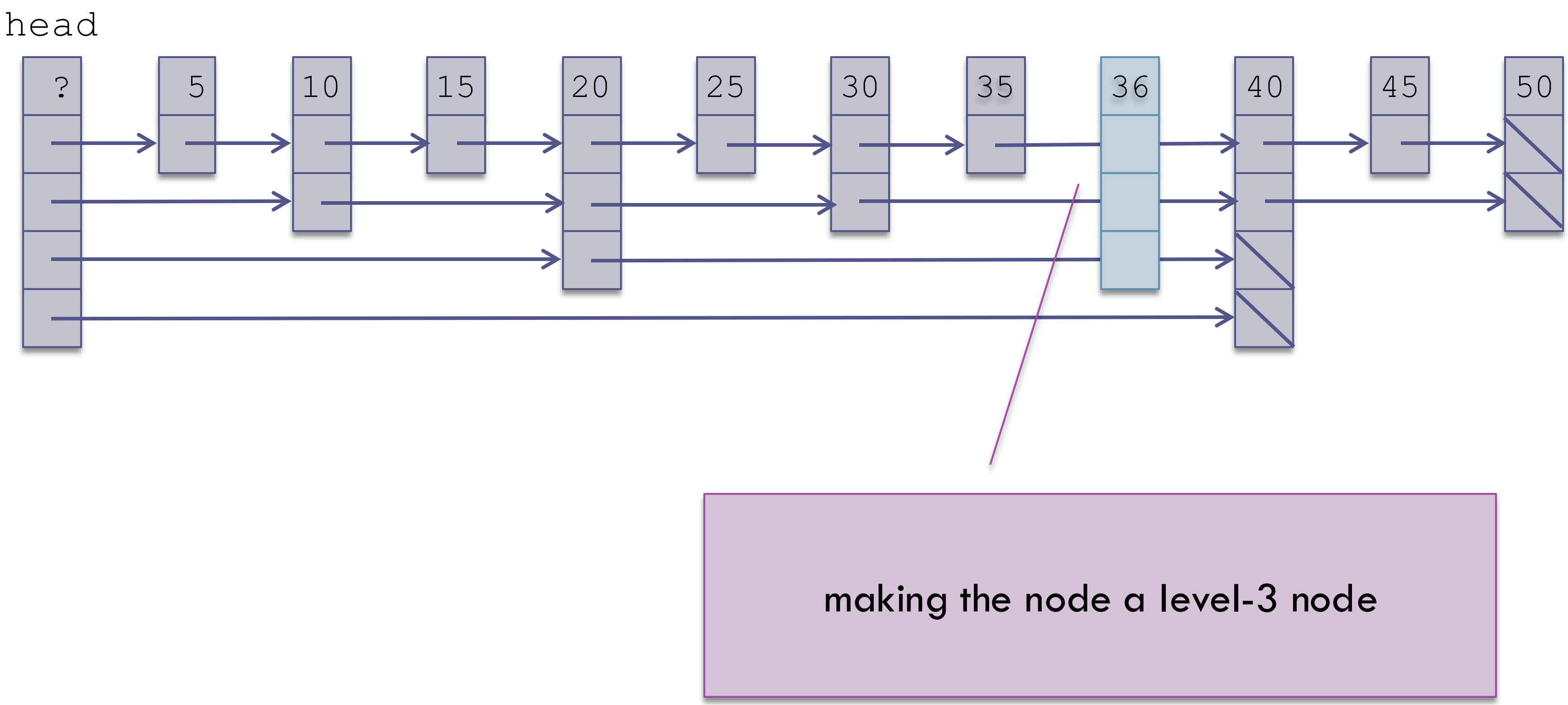


Insertion into a Skip-List (cont.)

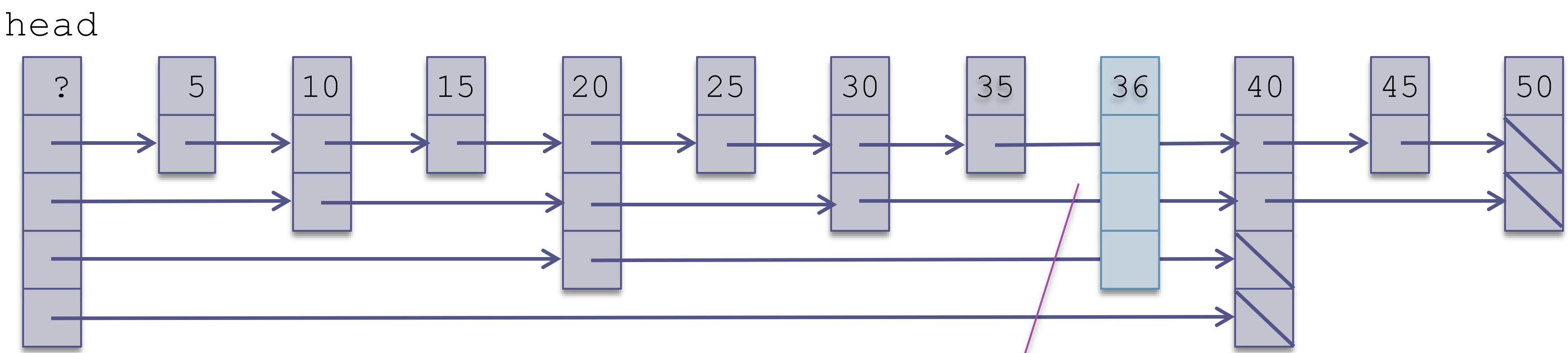


making the node a level-3 node

Insertion into a Skip-List (cont.)

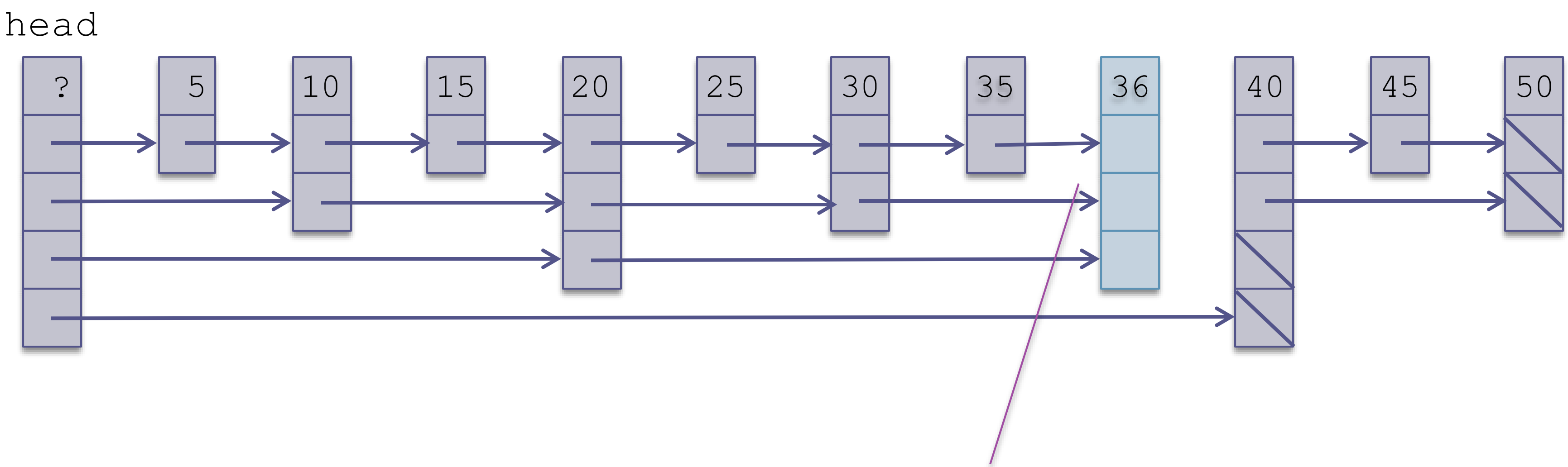


Insertion into a Skip-List (cont.)



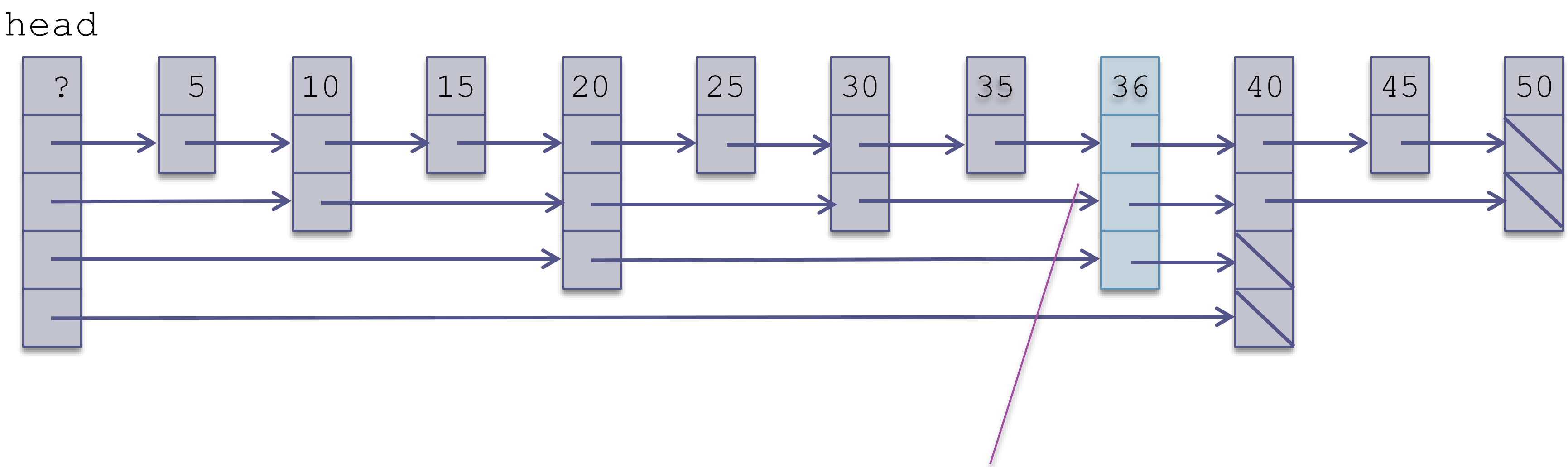
Along the way we recorded the last node visited at each level. We use these nodes to link up the new node

Insertion into a Skip-List (cont.)



Along the way we recorded the last node visited at each level. We use these nodes to link up the new node

Insertion into a Skip-List (cont.)



and to point the new node to their previous targets

Increasing the Height of a Skip-List

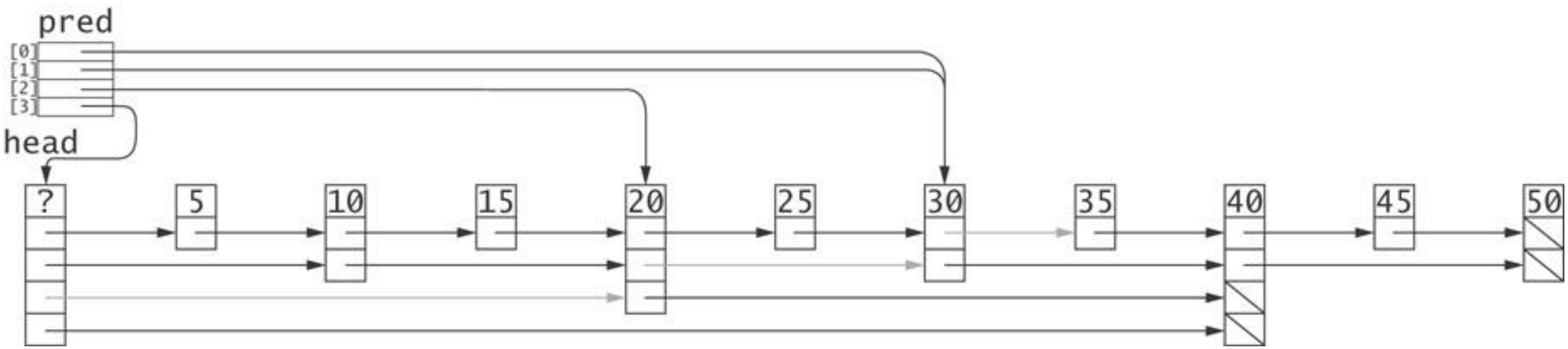
- A level-*m* skip-list can hold between 2^{m-1} and $2^m - 1$ items
- A level-4 skip-list, as in our example,
  - can efficiently hold up to 15 items ($2^4 - 1$)
  - When a 16^{th} item is inserted, the level is increased by 1

Implementing a Skip-List

```
/** Static class to contain the data and the links */
static class SLNode<E> {
    SLNode<E>[] links;
    E data;

    /** Create a node of level m */
    SLNode (int m, E data) {
       links = (SLNode<E>[]) new SLNode[m];  // create links
       this.data = data;                     // store item
    }
}
```

Searching a Skip-List

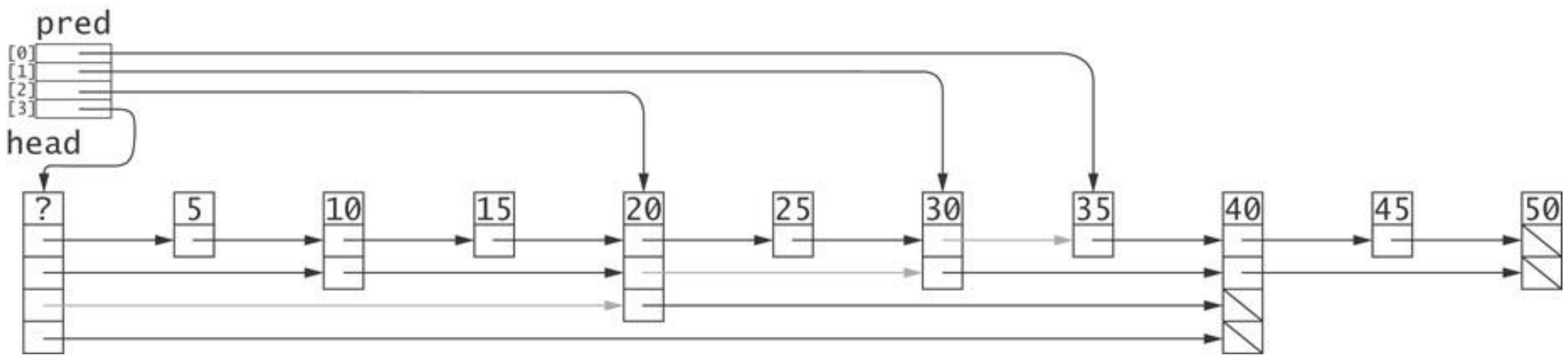


- Method `search` will return an array `pred` which holds references to the predecessors at each level

Searching a Skip-List (cont.)

- Listing 9.7 (`Methods for Searching a Skip-List`, pages 528-529)

Insertion




```
newNode.links[i] = pred[i].links[i];
pred[i].links[i] = newNode;
```

Determining the Size of the Inserted Node

```
/** Natural Log of 2 */
static final double LOG2 = Math.log(2.0);

/** Method to generate a logarithmic distributed integer between
    1 and maxLevel. i.e., 1/2 of the values returned are 1, 1/4
    are 2, 1/8 are 3, etc.
    @return a random logarithmic distributed int between 1 and
        maxLevel
*/
private int logRandom() {
    int r = rand.nextInt(maxCap);
    int k = (int) (Math.log(r + 1) / LOG2);
    if (k > maxLevel - 1) {
        k = maxLevel - 1;
    }
    return maxLevel - k;
}
```

Completing the Insertion Process

```
if (size > maxCap) {
    maxLevel++;
    maxCap = computeMaxCap(maxLevel); // maximum capacity
    head.links = Arrays.copyOf(head.links, maxLevel);
    pred = Arrays.copyOf(update, maxLevel);
    pred[maxLevel - 1] = head;
}
```

Performance of a Skip-List

- In an ideal skip-list, every other node is at level 1, and every 2^mth node is at least level m
- With this ideal structure, performance matches that of a binary search at O(log n)
- By randomly choosing the levels of inserted nodes to have an exponential distribution, the skip-list will have the desired distribution of nodes
- However, the nodes are randomly positioned throughout the skip-list—making the average time for search and distribution O(log n)