CS 228: Introduction to Data Structures
Lecture 30
Wednesday, November 9, 2016

Searching for a Key in a BST

To look for a key `k` in a tree, we could iterate though its nodes using an inorder traversal, stopping when we either find the key, or we run out elements to look at. In fact, the implementation in `AbstractSet` actually uses an iterator to implement `contains()`.

A faster way is to exploit the BST property to go down just one path in the tree, starting at the root. By examining the key `c` in the current node, we can decide whether to

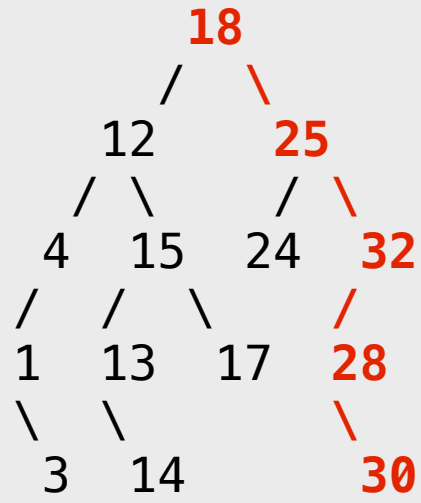
- ***stop***, if `c` is the same as `k`,
- look for `k` in the ***left subtree***, if `k` is less than `c`, or
- look for `k` in the ***right subtree***, if `k` is greater than `c`.

This is like traversing a linked list, but with a conditional statement to decide which way to go. The algorithm is implemented in the `findEntry()` method below.

```
protected Node findEntry(E key)
{
  Node current = root;
  while (current != null)
  {
    int comp =
      current.data.compareTo(key);
    if (comp == 0)
    {
      return current;
    }
    else if (comp > 0)
    {
      current = current.left;
    }
    else
    {
      current = current.right;
    }
  }
  return null;
}
```

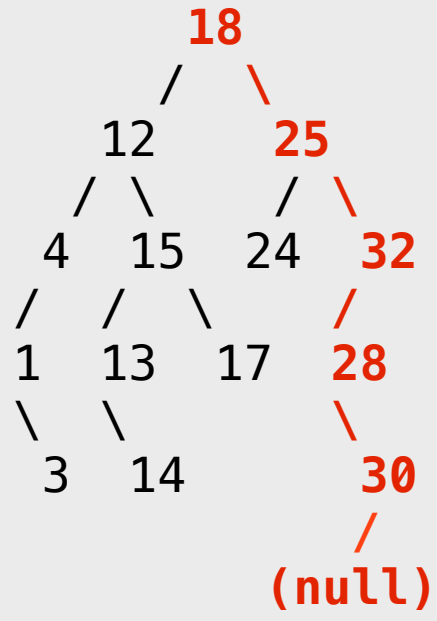
Note the use of `compareTo()`, because type E extends `Comparable<? super E>`.

Example. Here's the path `findEntry()` follows in a search for key 30 on the first of the two preceding trees.



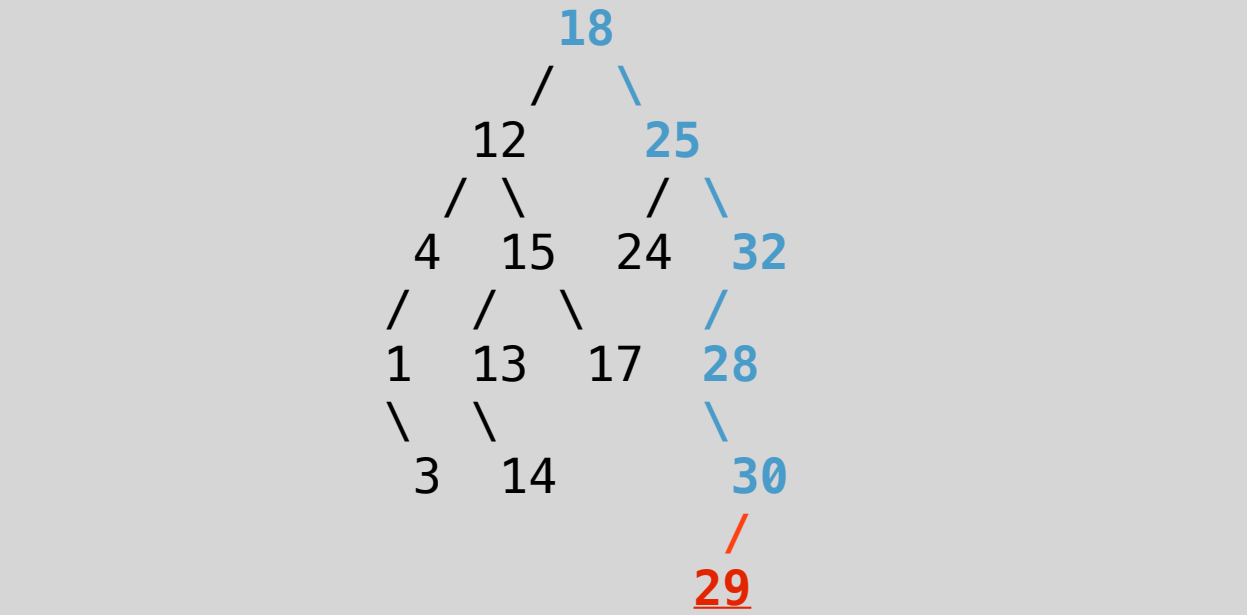
Note that we only had to look at 5 keys, instead of the 12 keys that precede 30.

An unsuccessful search ends up in an empty subtree. For example, here's a search for 29.



Adding a Key to a BST

Suppose we want to ***add*** 29 to the tree in the preceding example. We can just put it in the empty slot where our unsuccessful search ended:



In general, to `add` a key k, we do as follows.

(1) Search for k in the tree.

(2) If k is found, return `false`, indicating that we could not add it, since it's already in the set.

(3) If k is not found, add it in the empty slot where the unsuccessful search ended (and return `true`).

To convince yourselves that this works, try adding some more keys — e.g., 2 or 26 — into the tree above.

Implementation Details

To conform to the specifications of the Java `Set` interface, the implementations of `add()` and `contains()` must address a few technical details. We explain these details next.

`contains()`

The `findEntry()` method does most of what we need to implement `contains()`. There is, however, one issue to consider. It is tempting to write the method as follows.

```
public boolean contains(Object obj)
{
  return findEntry(obj) != null;
}
```

Unfortunately, this will not compile: as far as Java is concerned, `obj` is not of type E. This seems to leave us in a bind. On the one hand, to apply `compareTo()`, we must use type E within `findEntry()`. On the other, the

`Set` API requires the argument of `contains()` to be `Object`. The solution is to add an unsafe cast. The cast is itself meaningless, but it allows the code to compile.

```
public boolean contains(Object obj)
{
  E key = (E) obj;
  return findEntry(key) != null;
}
```

If the runtime type of `obj` is incompatible with type `E`, the `compareTo` method will throw a `ClassCastException`, which is exactly the behavior specified for the `Set` API.

`add()`

The `add()` method begins by trying to find the key we're trying to add. If we *do* find it, then the method should return `false` — remember, there can be no duplicates. Otherwise, our search must have ended in the empty slot where the new key should be inserted. We insert a new node with the given key in this slot and return `true`. The code is on the next pages.

```
public boolean add(E key)
{
  if (root == null)
  {
    root = new Node(key, null);
    ++size;
    return true;
  }

  Node current = root;

  while (true)
  {
    int comp =
      current.data.compareTo(key);
    if (comp == 0)
    {
      // key is already in the tree
      return false;
    }
```

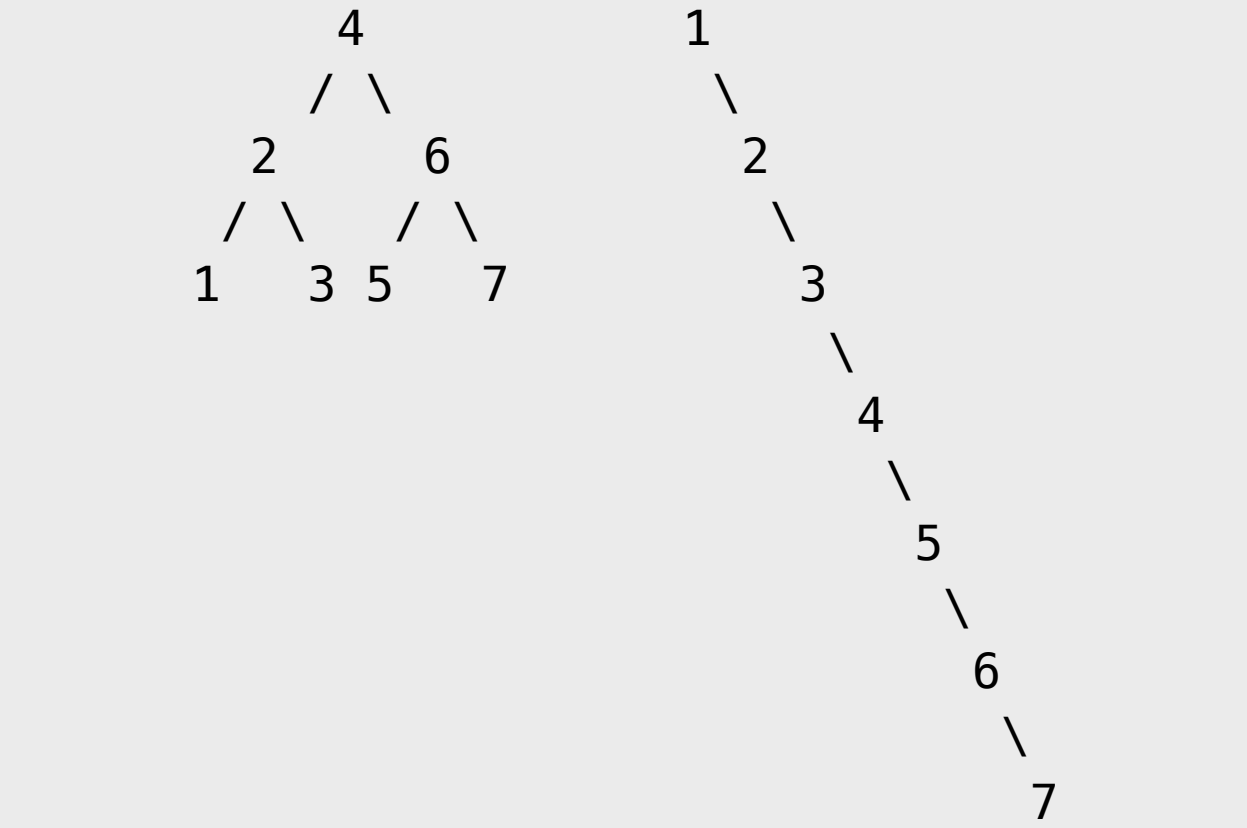
```
      else if (comp > 0)
      {
        if (current.left != null)
        {
          current = current.left;
        }
        else
        {
          current.left =
            new Node(key, current);
          ++size;
          return true;
        }
      }
      else
      {
        if (current.right != null)
        {
          current = current.right;
        }
        else
        {
          current.right =
            new Node(key, current);
          ++size;
          return true;
        }
      }
    }
  }
```

The Time Complexity of Searching and Insertion

Searching for a key and adding a new key require following a path from the root, doing a comparison at each step. Suppose we make the (very reasonable) assumptions that a key comparison takes O(1) time and that following a pointer and linking in a single node also take O(1) time. Then, the time complexity to search for a key or add a key is proportional to the length of the path from the root that is followed when executing these operations. Since the longest path from the root in a tree T has height(T) + 1 nodes, we have the following.

Fact. The worst-case time needed to search for a key or to add a key to a BST T is O(height(T)).

As we will see, the time to delete a key is also O(height(T)). Note that the height of a tree can vary considerably. For example, here are the minimum- and maximum-height trees on 7 keys.



Let n be the number of nodes in the BST. In general, there are two extreme cases:

- **Best case: well-balanced tree.** At each node, the number of nodes in the left subtree is roughly the same as the number on the right subtree. Then, the height is O(log n) and so is the time complexity of the operations.

- **Worst case: very unbalanced (degenerate) tree.** Each node has only one child. Then, the height of the tree is n-1 and the operations are O(n).

The actual height will fall somewhere in between. It can be proved mathematically that, under certain conditions, the *expected*[1] height is O(log n), so the expected running time is O(log n). In practice, this kind of argument is unsatisfactory, and we would prefer a performance *guarantee*. Fortunately, there are more advanced BSTs with clever extra features to keep them from getting too far out of balance, ensuring O(log n) time for all operations — AVL trees[2] and red-black trees[3] are two examples. Indeed, `TreeSet`, the Java library implementation of a sorted set, uses red-black trees. Next week, we will see splay trees, a kind of "self-adjusting" BST where the *amortized time* of all operations is O(log n).

`remove()`

To delete a key, we first have to find it; `findEntry()` takes care of this. If the key is not in the tree, we just return `false`. Otherwise, `findEntry()` returns a reference to the node containing the key. We then leave the actual removal to a helper method called

---

[1] For a formal definition of expected value, see http://en.wikipedia.org/wiki/Expected_value.

[2] http://en.wikipedia.org/wiki/AVL_tree. For a nice demo of AVL trees, see http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html.

[3] http://en.wikipedia.org/wiki/Red%E2%80%93black_tree

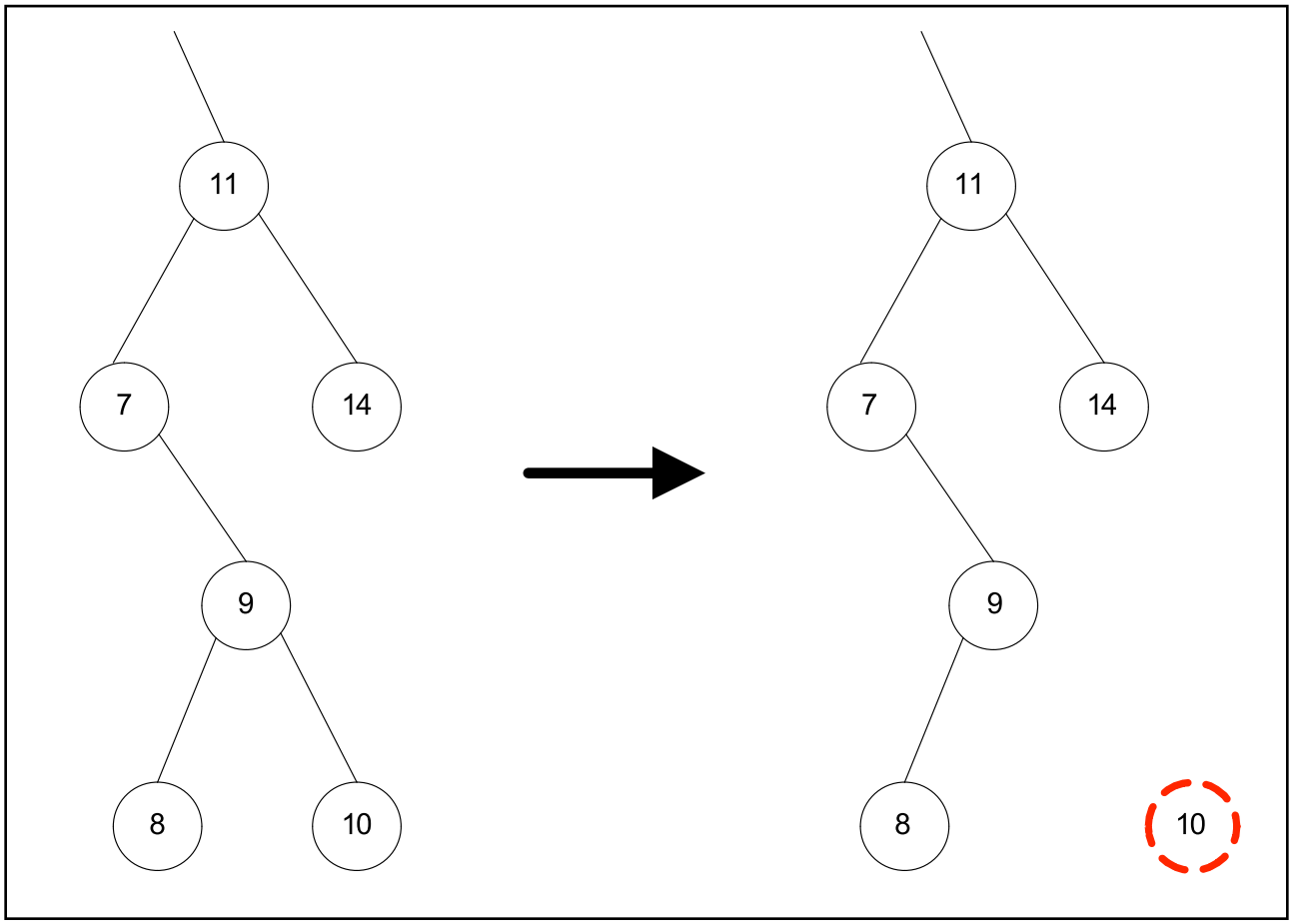
`unlinkNode()`, which we'll describe shortly. The code for `remove()` is now simply this.

```
public boolean remove(Object obj)
{
  E key = (E) obj;
  Node n = findEntry(key);
  if (n == null)
  {
    return false;
  }
  unlinkNode(n);
  return true;
}
```

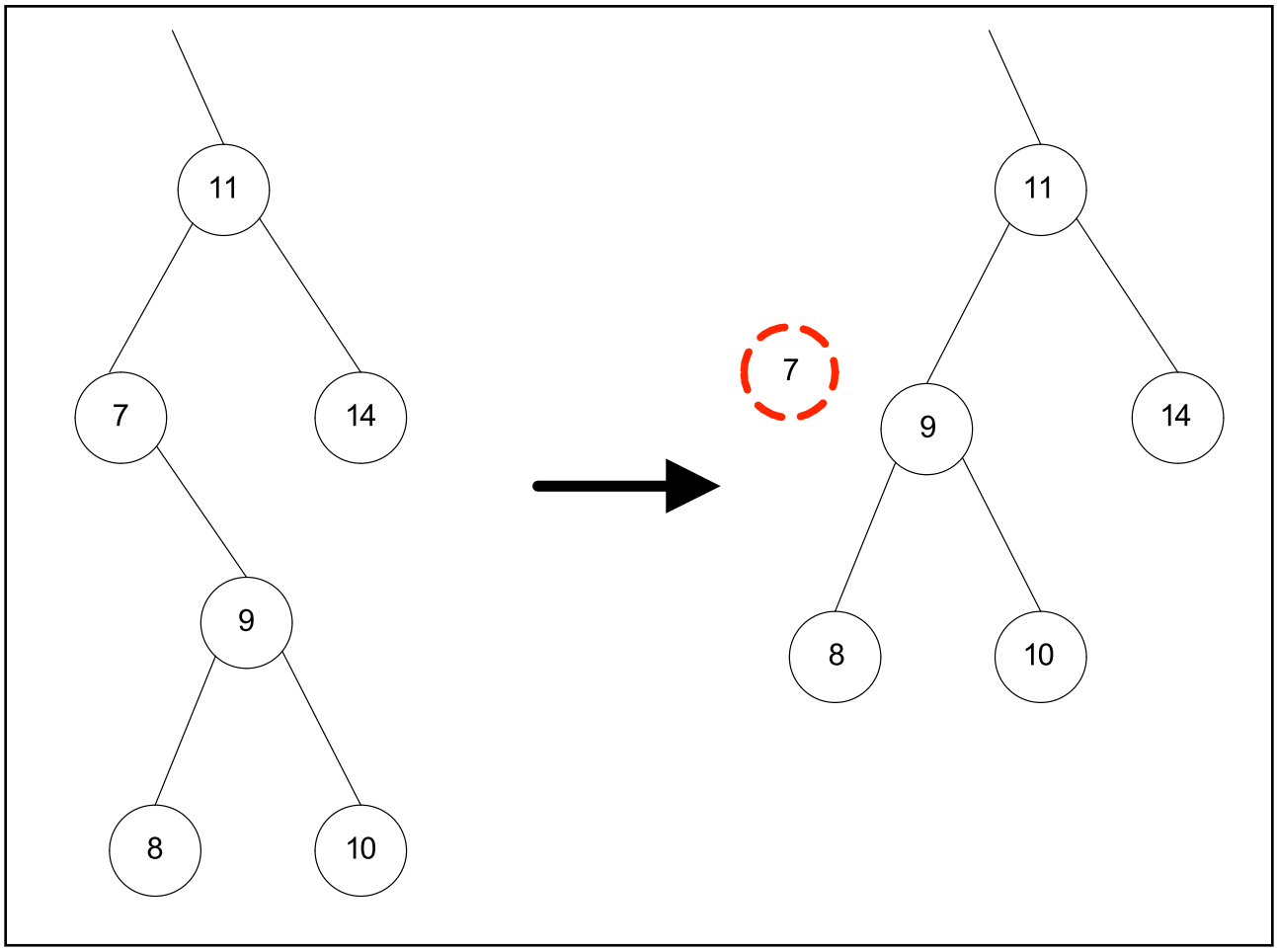
`unlinkNode()`

There are three possibilities.

Case 1: The node to be removed has no children. Then, we simply delete the node.



Case 2: The node to be removed has one child. This is easy too. The BST property is preserved by replacing the deleted node with its child.



Case 3: The node to be removed has two children.
For instance, suppose we want to remove the node "4" in the tree in p. 8. Unlike cases 1 and 2, we cannot just delete the node; we must replace it with something else. If we put either of 4's children in its place, we then have to move one of the child's children, and so on. This looks like too much restructuring.

The solution is to leave the node where it is, and find some other element in the tree that can go there without

violating the BST property. The ***successor*** of 4 in the tree ordering, which is 7, is a good choice. This is because of the following.

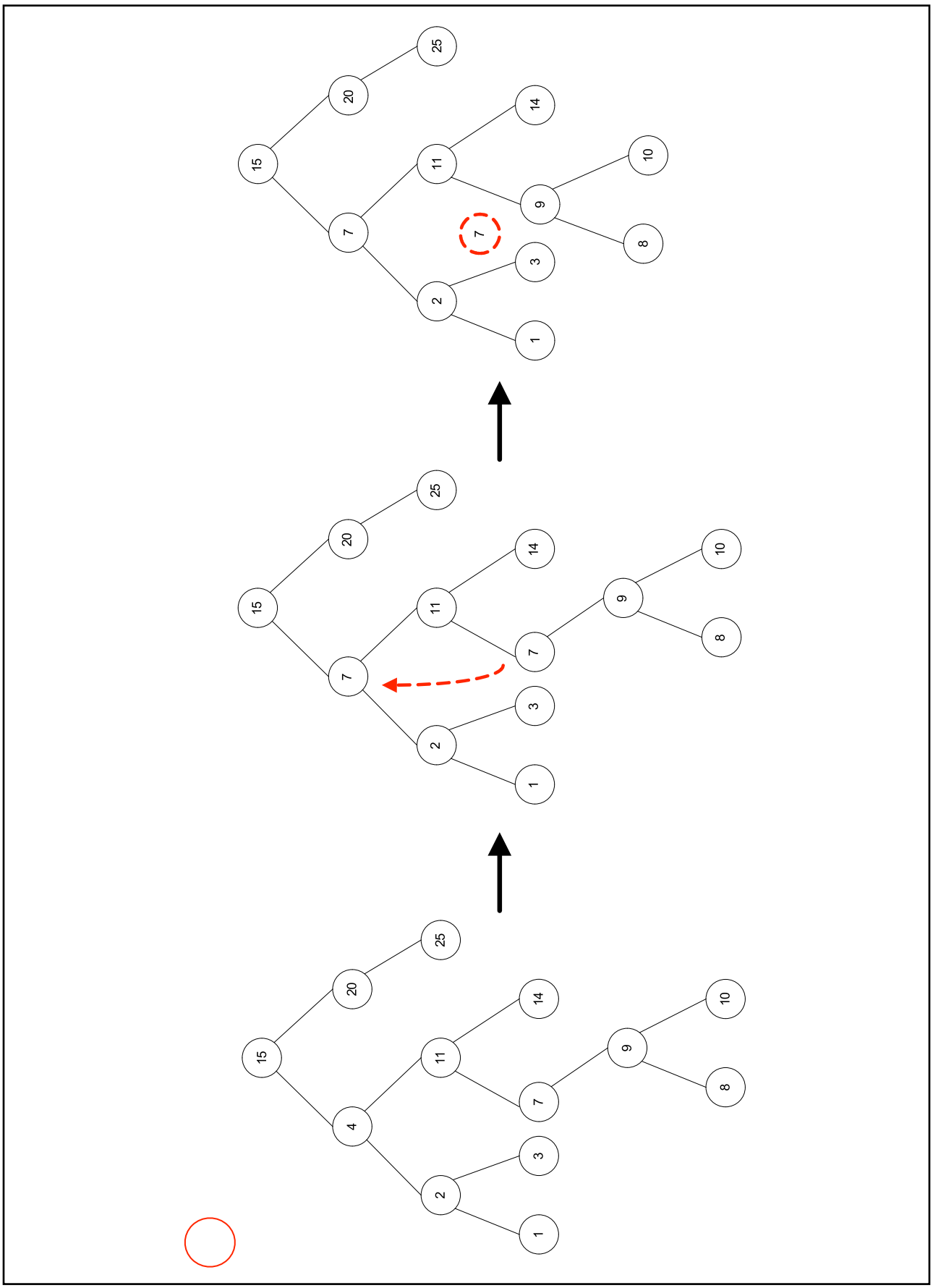
Fact. *If `t` is a node with two children and node `s` contains the successor of `t`, then `s` cannot have a left child.*

Exercise. Prove this.

Thus, deleting the successor of a node with two children falls under the (easy) Cases 1 or 2. In our example, the node `s` containing 7 only has one child. We copy the 7 into the node t containing 4, and then delete node `s`.

Note. In the preceding example, instead of replacing 4 by its successor, we could have replaced it by its ***predecessor*** in the tree ordering, which is 3. The predecessor of a node with two children cannot have a right child, so deleting it is as easy as deleting the successor.

The pseudocode for `unlinkNode` on pp. 9-10 assumes that we have a `successor()` method, which, given a node `n`, returns the successor of `n` or `null`, if `n` has no successor.


25
20
15
14
11
10
9
7
8
7
3
2
1
25
20
15
14
11
10
9
7
8
3
2
1
25
20
15
14
11
10
9
4
7
8
3
2
1

```
unlinkNode(n):
    if n has two children
        // Copy the successor's data into n; then arrange
        // to delete the successor.
        s = successor(n)
        n.data = s.data
        n = s

    // Now n has at most one child, so we delete n and
    // replace it with the child (or possibly null).

    // First, figure out whether the replacement node
    // should be the left child, right child, or null.
    replacement = null
    if n has a left child
        replacement = n.left
    else if n has a right child
        replacement = n.right

    // Now, link the replacement node to n's parent.
    if n is the root
        root = replacement
    else
        if n is a left child of its parent (*)
            n.parent.left = replacement
        else
            n.parent.right = replacement
```

```
    if replacement != null
        replacement.parent = n.parent

    --size
```

(*) To test if n is a left child of its parent, we do

```
n == n.parent.left
```