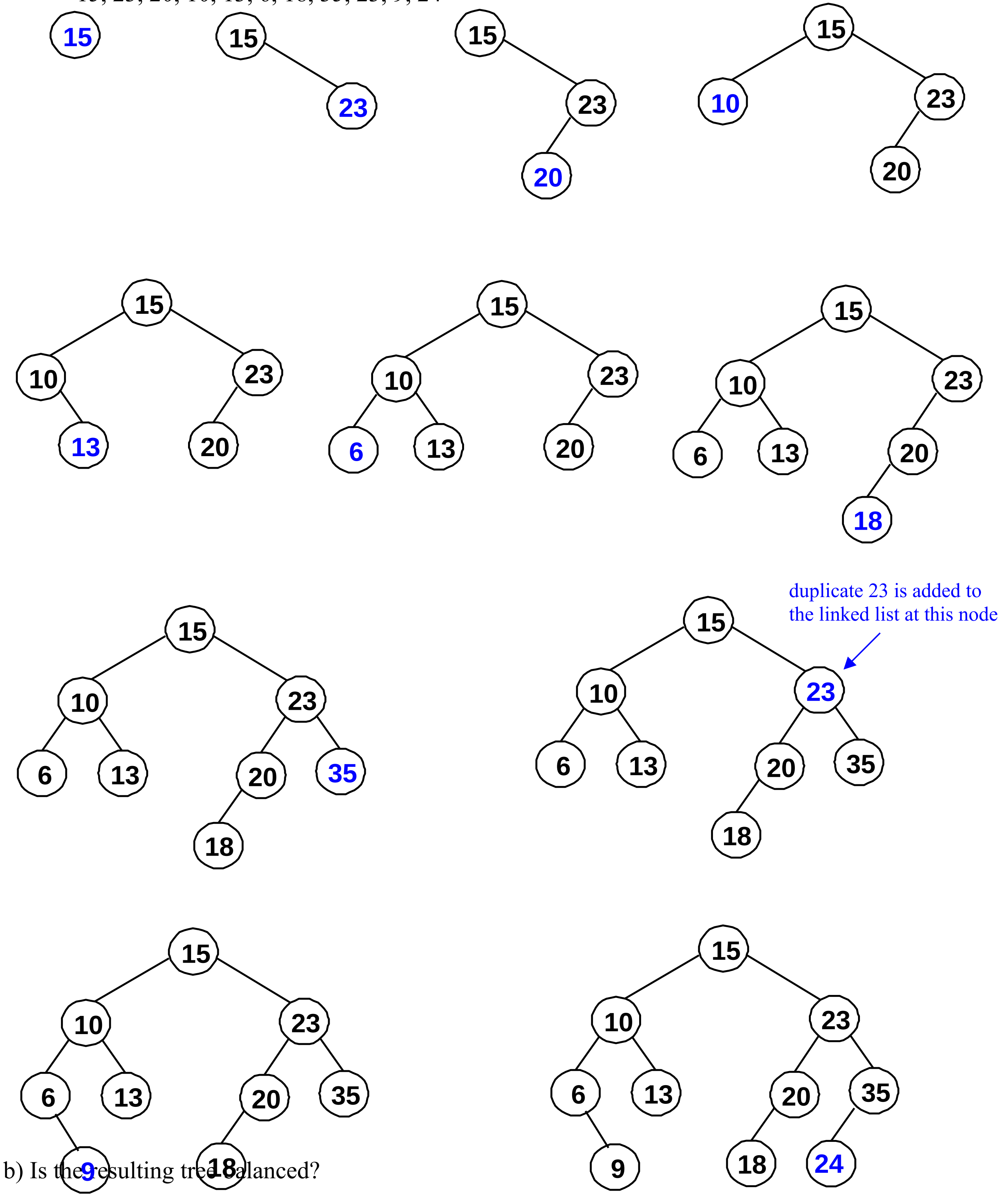
CSCI E-119
Section Notes
Section 8 Solutions

1. Binary Search Trees

a) Insert the following sequence of keys into an empty binary search tree:

15, 23, 20, 10, 13, 6, 18, 35, 23, 9, 24

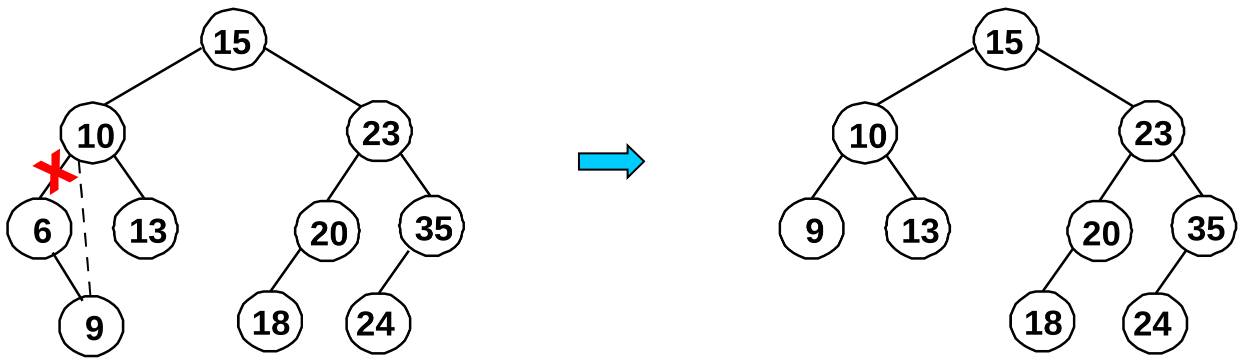


b) Is the resulting tree balanced?

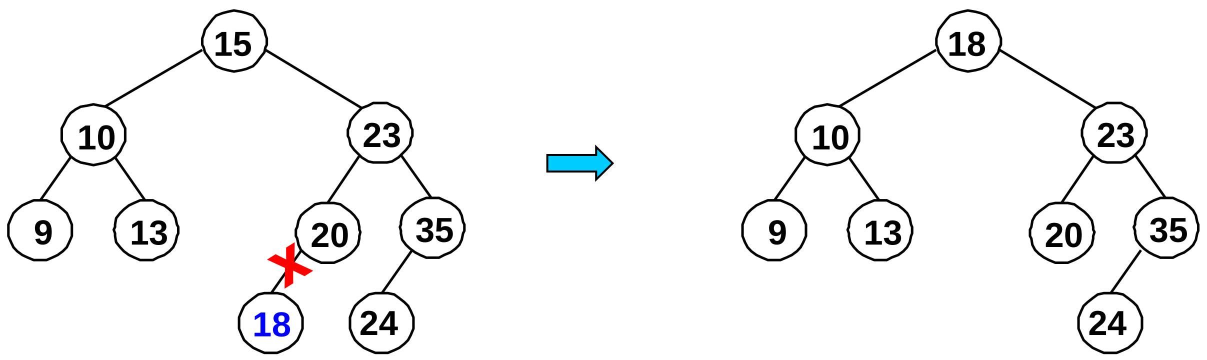
Yes – for each node, the heights of the subtrees of the node differ by at most 1.

c) What will the tree from part (a) look like after deleting the following items: 6, then 15, then 20

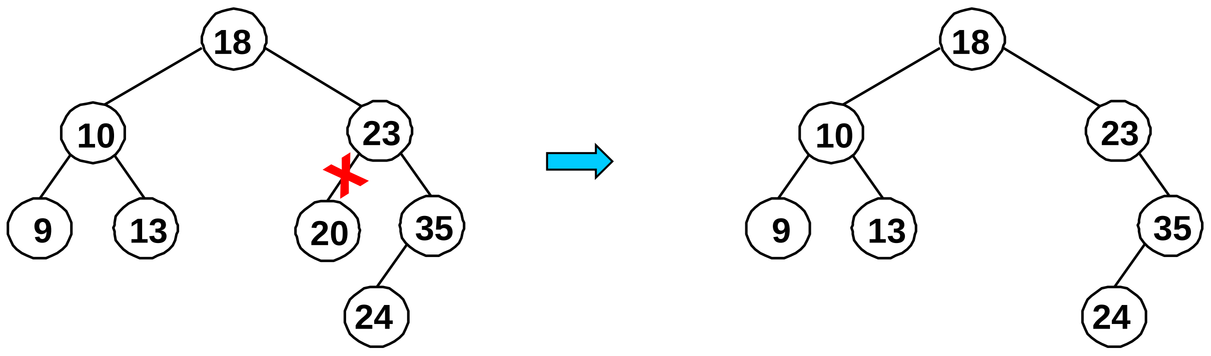
First we delete the 6 (an example of case 2):



Then we delete the 15 (an example of case 3). We begin by replacing the 15 with the smallest item in its right subtree (the 18), then we delete the 18 (an example of case 1):



Then we delete the 20 (another example of case 1):



d) Is the resulting tree balanced?

No – the 23 has an empty left subtree (which can be thought of as having a height of -1), but its right subtree has a height of 1 – so the heights of its subtrees differ by more than 1.

2. Binary Tree Methods

We want to write a method which counts the number of leaf nodes in a given tree. Leaf nodes are defined as nodes which have no children.

What parts of the tree will we need to process to accomplish this?

We must process the entire tree, and cannot leave any nodes out, because all the leaves are at the bottom of the tree.

Assume we have a node **root** which is the root node of a tree. How can we recursively define the number of leaf nodes in that tree?

The number of leaves in a tree for which **root** is the root node is the sum of the number of leaves in **root**'s left subtree and the number of leaves in **root**'s right subtree. In other words, we can say that

leafCount(root) = leafCount(root.left) + leafCount(root.right).

What will be our base case(s)?

We need to terminate the recursion if we are either at a leaf or if we are given an empty tree or subtree.

What will our function return? An integer value.

```
public int leafCount( Node root ) {
    if ( root == null )
        return 0;
    else if ( (root.left != null) || (root.right != null) )
        return (leafCount(root.left) +
                leafCount(root.right));
    else
        return 1; //if we got here this must be a leaf
}
```

What would we need to do if we wanted to write this method iteratively? What sort of data structures would we need?

We would need to maintain a stack onto which we could push the left and right subtree's root nodes as we iterated.

Notice we could also implement this algorithm iteratively using a level-order traversal of the binary tree. Would this implementation be less efficient for balanced trees? Why or why not?

No, it would not be less efficient, because we would still need to touch every node in the tree once, just like we do in our depth-first recursive traversal above.

3. Implementing an iterator for a binary tree

In Problem Set 4, you will implement an *postorder* iterator for the LinkedTree class. In this section, we will implement a *preorder* iterator to give you a sense of the types of issues that you will need to consider.

In each case, the Java class for the iterator should implement the following interface:

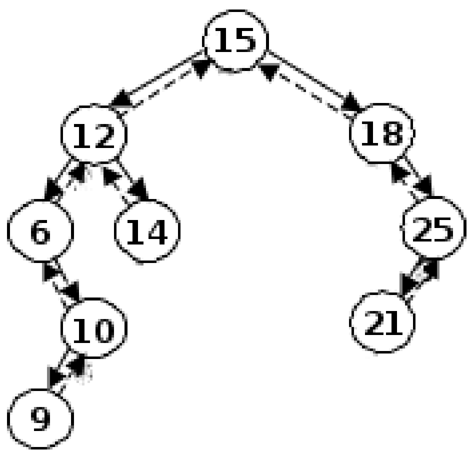
```
public interface LinkedTreeIterator {
   // Are there other nodes to see in this traversal?
   boolean hasNext();

   // Return the value of the key in the next node in the
   // traversal, and advance the position of the iterator.
   int next();
}
```

If we create a *preorder* iterator object and repeatedly invoke its next() method, we should visit the nodes of the tree in the order taken by a preorder traversal. If we create an *inorder* iterator object and repeatedly invoke its next() method, we should visit the nodes of the tree in the order taken by an inorder traversal.

Each type of iterator will be implemented as a private inner class of the LinkedTree class. If we name the class for our preorder iterator PreorderIterator, the header of the class will look like this:

```
public class LinkedTree {
   …
   private class PreorderIterator implements LinkedTreeIterator {
      …
```

Before we implement this class, we need to mention a related change to the LinkedTree class that is needed to implement the iterators. We need to add a parent field to each node in the tree; it will be used to hold a reference to the node's parent. Below is an example of a tree in which these references have been added. They are shown as arrows that have dotted lines.



In the assignment, you will need to modify the LinkedTree code so that these parent references are correctly maintained. For the sake of this exercise, we will assume that these references are already fully supported.

What instance variable or variables do we need in our PreorderIterator class?

We only need a single instance variable, a reference to the next node to be visited:

```
    private class PreorderIterator implements LinkedTreeIterator {
        private Node nextNode;
        …
```

What should the constructor for the class do?

It needs to make nextNode refer to the first node to be visited by the iterator. Since a preorder traversal begins by visiting the root of the tree, the constructor is very simple:

```
        private PreorderIterator() {
            nextNode = root;
        }
```

Note that root is an instance variable of the LinkedTree class. Because PreorderIterator is an inner class, it has access to the instance variables of the tree object that was used to create it. (We'll say more about how the iterator is created in a moment.)

Note also that the constructor for your inorder iterator will be more complicated, because an inorder traversal doesn't necessarily begin with the root of the entire tree.

What should the hasNext() method look like?

It's also quite simple. There are more nodes to visit so long as nextNode is non-null.

```
        public boolean hasNext() {
            return (nextNode != null);
        }
```

The next() method must do two things:
1) it returns the value of the next node to be visited (in this case, the value of the node's key)
2) it advances the iterator so that it is ready for the next call to this method

If there are no additional nodes to visit, what should we do?

throw an exception:

```
        public int next() {
            if (nextNode == null)
                throw new NoSuchElementException();
            …
```

The second task involves updating the state of the iterator. What variable(s) need to be updated?

nextNode

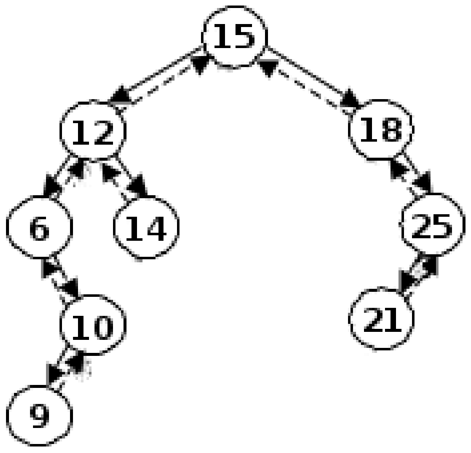
In order to accomplish the first task, what do we need to do *before* updating the iterator's state?

We need to store the key of the node currently pointed to by nextNode in a local variable:

```
public int next() {
    if (nextNode == null)
        throw new NoSuchElementException();

    int key = nextNode.key;
    …
```

When advancing the iterator, there are several possible cases to consider, depending on the number and types of children of the node whose key we are about to return (which we will call the "current node")..

Looking at our earlier example (shown again below), what are these cases, and how would you advance the iterator in each case?



Case 1: the current node has a left child (examples: if the current node is the 15, 12, or 10 nodes)
A preorder traversal visits the root, then it recursively traverses the left subtree, then it recursively traverses the right subtree.

As a result, if the current node has a left child, then the next node to be visited is the left child, since it is the root of the left subtree. We can code this as follows:

```
public int next() {
    …
    if (nextNode.left != null)
        nextNode = nextNode.left;
```

Case 2: the current node does not have a left child, but it does have a right child (examples: if the current node is the 6 node or 18 node above)
In this case, because there is no left subtree, the traversal goes next to the right subtree, and thus the next node to be visited is the current node's right child, since it is the root of the right subtree:

```
public int next() {
    …
    if (nextNode.left != null)
        nextNode = nextNode.left;
    else if (nextNode.right != null)
        nextNode = nextNode.right;
```

Case 3: the current node is a leaf node (examples: if the current node is 9, 14, or 21)
In this case, we need to go back up the tree, looking for nodes that have yet to be visited.

Because a preorder traversal visits a node as soon as it encounters it (i.e., the root is visited first) and then visits the left subtree, the only unvisited nodes are in right subtrees of nodes that we've already visited. Thus, we can trace back up using parent references until we find a node with a right child on a different path from the one that we took to get to the currentNode.

The following code almost works:

```
public int next() {
    …
    if (nextNode.left != null)
        nextNode = nextNode.left;
    else if (nextNode.right != null)
        nextNode = nextNode.right;
    else {
        // XXX: doesn't work correctly in all cases
        Node parent = nextNode.parent;
        // look for a node with a right child
        while (parent != null &&
          parent.right == null) {
            parent = parent.parent;
        }
        if (parent == null)
            nextNode = null;    // the iteration is complete
        else
            nextNode = parent.right;
    }
    …
```

However, this code fails to distinguish between right children that are on the path from the root to the current node (which have already been visited) and right children that are not on that path (and have yet to be visited).

For example, if the current node is the 9 node, the above code would tell us to revisit the 10 node, since 10 is the right child of its parent. Instead, we want the code to tell us to visit the 14 node next.
We can get around this problem by using two different references as we trace back up the tree: one to perform the traversal, and one to stay one node "behind" the first reference. That will allow us to find previously unvisited right children:

```
public int next() {
    …
    if (nextNode.left != null)
        nextNode = nextNode.left;
    else if (nextNode.right != null)
        nextNode = nextNode.right;
    else {
        Node parent = nextNode.parent;
        Node child = nextNode;
        // look for a node with an unvisited right child
        while (parent != null &&
          (parent.right == child || parent.right == null)) {
            child = parent;
            parent = parent.parent;
        }
        if (parent == null)
            nextNode = null;    // the iteration is complete
        else
            nextNode = parent.right;
    }
    …
```

Once the iterator has been advanced, what final thing needs to be done?

We need to return the previously stored key:

```
public int next() {
    …
    int key = nextNode.key;

    …

    return key;
}
```

Below is the complete PreorderIterator inner class from LinkedTree.java, along with the preorderIterator() method that returns an instance of it:

```
/** Returns a preorder iterator for this tree. */
public LinkedTreeIterator preorderIterator() {
    return new PreorderIterator();
}

/*** inner class for a preorder iterator ***/
private class PreorderIterator implements LinkedTreeIterator {
    private Node nextNode;

    private PreorderIterator() {
        // The traversal starts with the root node.
        nextNode = root;
    }

    public boolean hasNext() {
        return (nextNode != null);
    }

    public int next() {
        if (nextNode == null)
            throw new NoSuchElementException();

        // Store a copy of the key to be returned.
        int key = nextNode.key;

        // Advance nextNode.
        if (nextNode.left != null)
            nextNode = nextNode.left;
        else if (nextNode.right != null)
            nextNode = nextNode.right;
        else {
            // We've just visited a leaf node.
            // Go back up the tree until we find a node
            // with a right child that we haven't seen yet.
            Node parent = nextNode.parent;
            Node child = nextNode;
            while (parent != null &&
              (parent.right == child || parent.right == null)) {
                child = parent;
                parent = parent.parent;
            }

            if (parent == null)
                nextNode = null;  // the traversal is complete
            else
                nextNode = parent.right;
        }

        return key;
    }
}
```