Lab 20 Binary Search Trees

Goal

In this lab you will modify a binary search tree so that the nodes are threaded in order. In-order traversal will be implemented using the thread. A client using a binary search tree will be developed.

Resources

- Chapter 23: Trees
- Chapter 24: Tree Implementations
- Chapter 25: A Binary Search Tree Implementation

In `javadoc` directory

- *`TreeInterface.html`*—API documentation for the tree ADT
- *`BinaryTreeInterface.html`*—API documentation for the binary tree ADT
- *`SearchTreeInterface.html`*—API documentation for the binary search tree ADT

Java Files

- *`Identifiers.java`*
- *`TestBST.java`*
- *`TestBinaryTree.java`*

In `TreePackage` directory

- *`BinaryNode.java`*
- *`BinarySearchTree.java`*
- *`BinaryTree.java`*
- *`BinaryTreeAccessInterface.java`*
- *`BinaryTreeInterface.java`*
- *`BinaryTreeWithAccess.java`*
- *`BinaryWithParentsTreeAccessInterface.java`*
- *`SearchTreeInterface.java`*
- *`TreeInterface.java`*

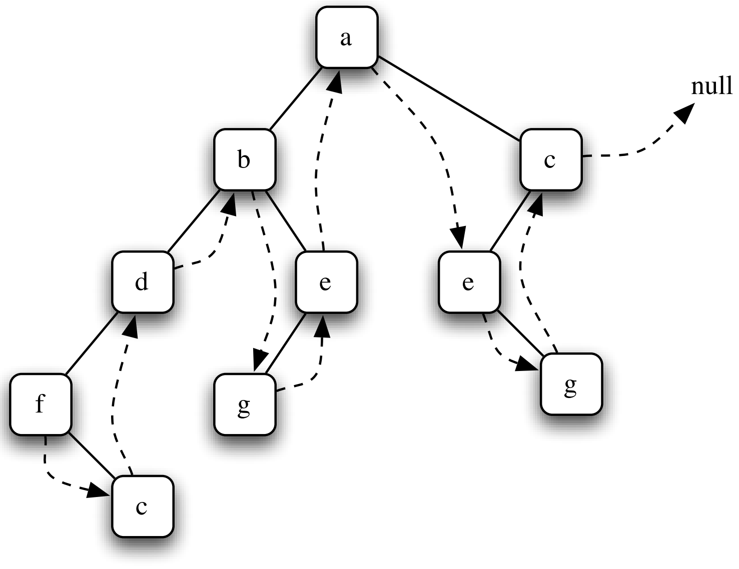
Input Files

- *`Small.java`*
- *`X.java`*

Introduction

Threaded Binary Trees

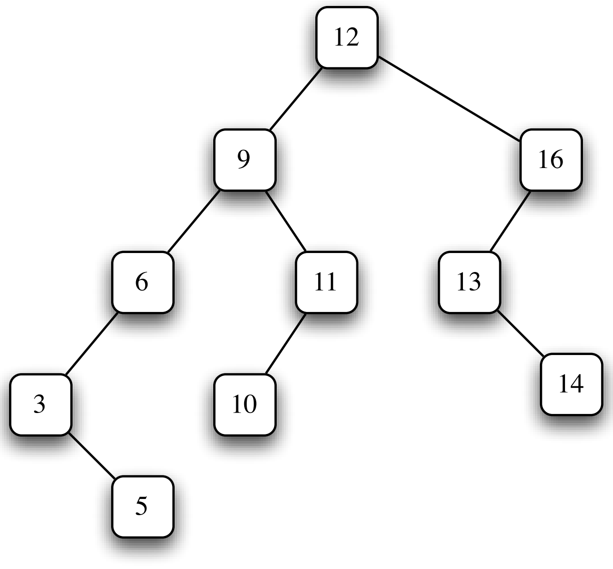
A binary tree is a tree where every node has zero, one, or two children. Each node will have two links, one for each child. In a threaded tree, an extra link is added that threads (links) all of the nodes in some fashion. In this lab, the threading will link the nodes according to an in-order traversal. In the following picture, the dashed lines indicate the thread links.



The first node in the thread is the first node in an in-order traversal.

Binary Search Tree

In a binary search tree, the values stored in the nodes cannot be arbitrary but must be orderable. The search tree property requires that all values in the left subtree must be less than the value stored in the node. Similarly, all values in the right subtree must be greater than the value stored in the node. A binary search tree must satisfy this property at all of its nodes. The following tree is a binary search tree with the same structure as the previous threaded tree.



Examining each node, you see that the search tree property holds. For example, the values to the left of 9 are 3, 5, and 6, while the values to the right are 10 and 11. If any two values in this search tree are switched, it no longer satisfies the search tree property.

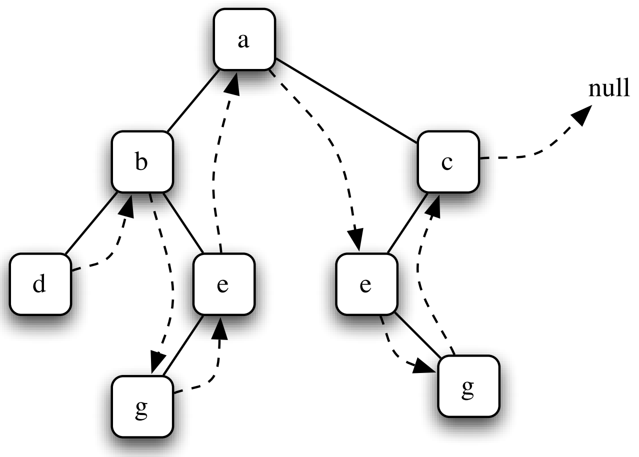
Notice that if the threads from the previous picture are followed, the values traversed will be in ascending order.

One nice property of a search tree is that adding a value in the tree will always be done at a leaf position.

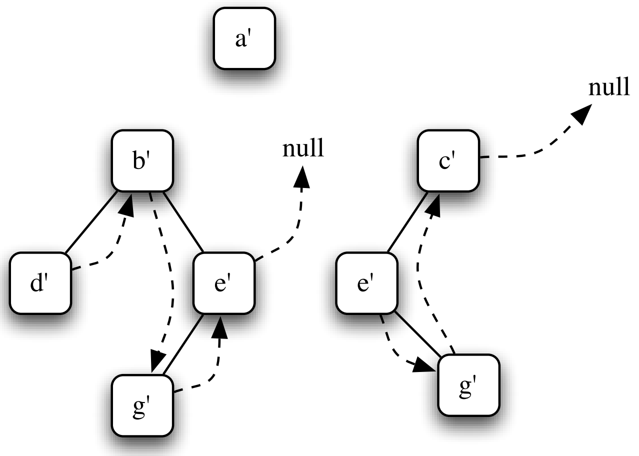
Pre-Lab Visualization

BinaryNode Copy with Threads

One of the responsibilities of the binary node class is to make a copy of the tree rooted by the node. This is done recursively. The threads complicate matters. Consider the following binary tree with threads.



In making a copy, a new node is created for copying the root, and then the left and right subtrees are copied. Notice that the links cannot be a straight copy. If so, they would refer to nodes in the original. Assume that the recursive copy correctly threads all of the nodes within the subtree, with the exception of the last threaded node in the subtree. Here is a picture after the copies have been made.

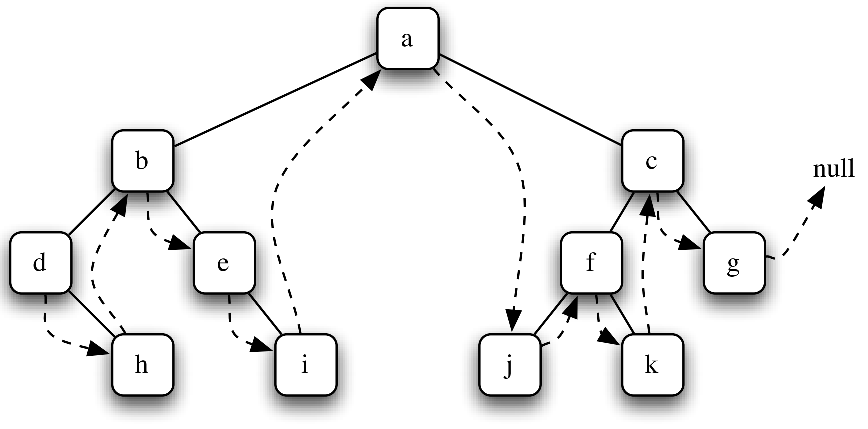


On the picture, indicate the changes that need to be made to complete the copy.

Give an algorithm for a method `linkSubtreeThreadOut(BinaryNode linkTo)` that will link the thread coming out of the left subtree.

Give an algorithm for a method `getSubtreeLeftmost()` that returns the node in the right subtree that should be the target of the thread from the root.

Use the previous two algorithms to show how the following tree is copied.



In-Order Iterator

The in-order iterator for the threaded binary search tree is nearly trivial. It just needs to follow the thread links. It will have a current node. When the `next` method is invoked, the value in the current node will be returned.

What is the condition for `hasNext()` to return true?

What does the iterator do for `next()`?

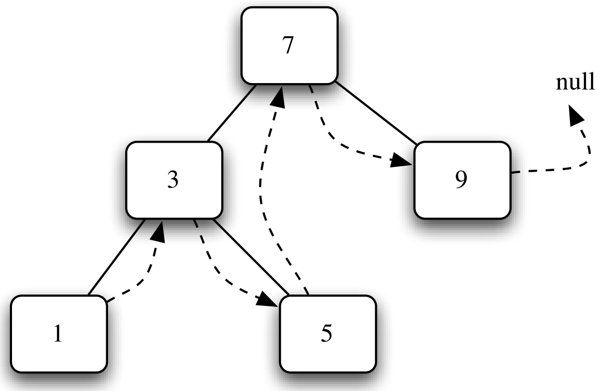
The only minor complication is how the iterator is initialized. It must set the current node to the first node in the thread.

Give an algorithm for finding the first node in the thread.

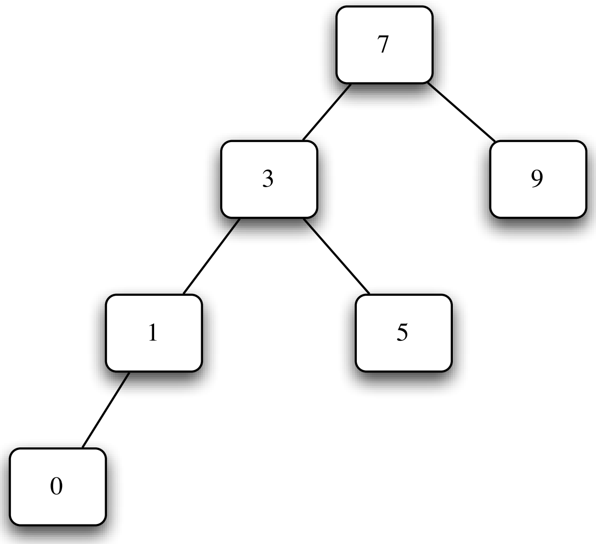
Threading A BinarySearchTree-Add

As nodes are added to the binary search tree, the threads will need to be adjusted.

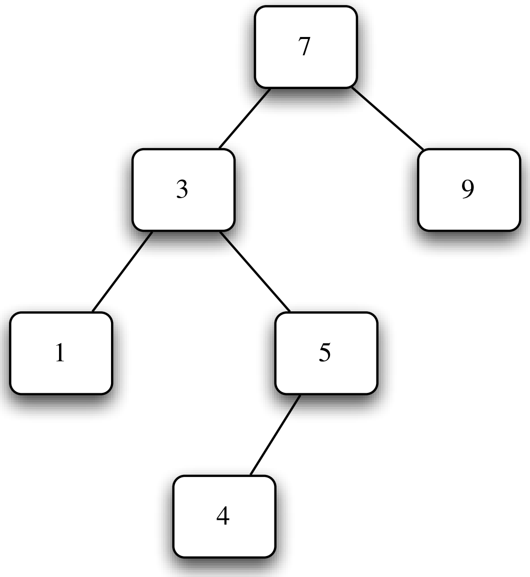
Consider the following initial threaded binary search tree.



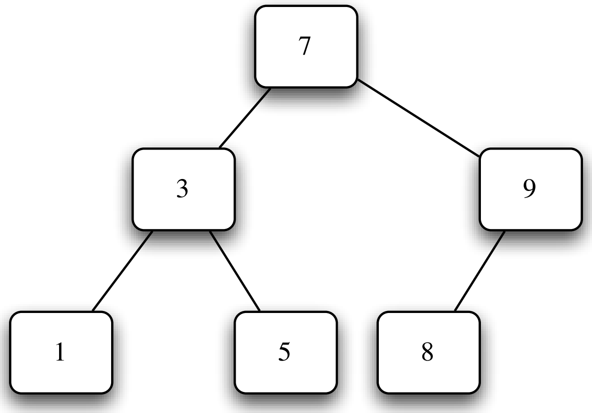
Adding the value 0 will result in the following tree. Put in the thread links and mark the ones that have changed. (Remember that the thread links give an in-order traversal of the tree.)



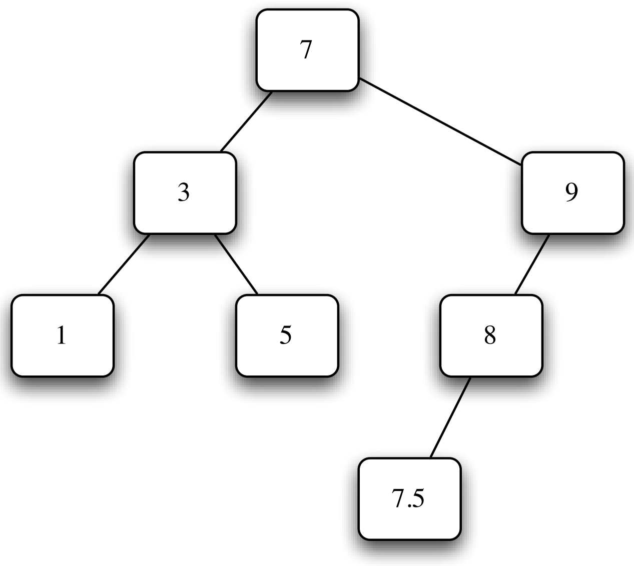
Adding the value 4 to the initial tree will result in the following tree. Put in the thread links and mark the ones that have changed.



Adding the value 8 to the initial tree will result in the following tree. Put in the thread links and mark the ones that have changed.



Adding the value 7.5 to the previous tree will result in the following tree. Put in the thread links and mark the ones that have changed.

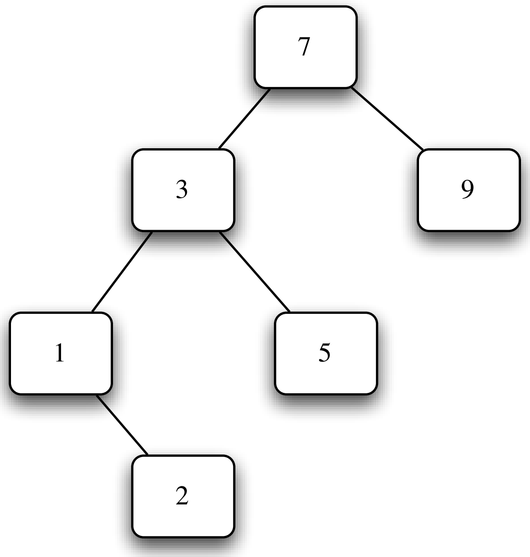


All of the previous examples added the node as a left child. If the insertion is on the left, where must the thread out of the inserted value go?

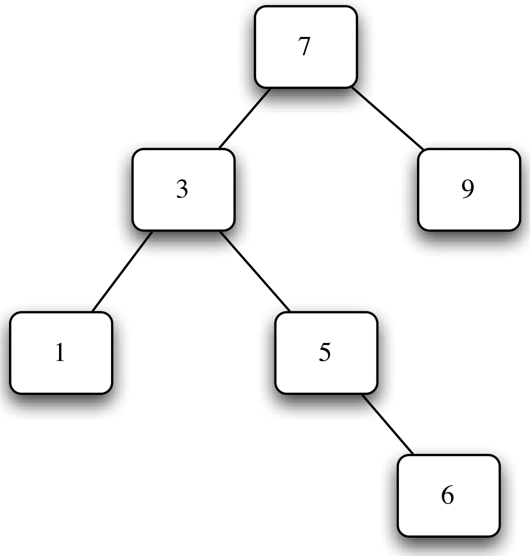
If the insertion is on the left, where is the thread that must be changed to refer to the new node?

Give an algorithm for finding the node with the thread to change. (You may assume that the tree has parent references and use them in the algorithm.)

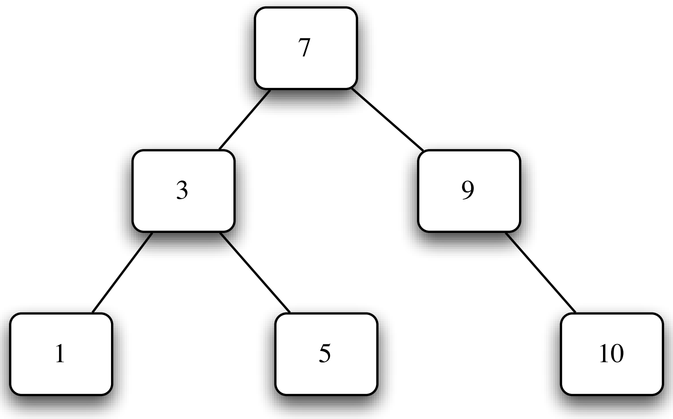
Adding the value 2 to the initial tree will result in the following tree. Put in the thread links and mark the ones that have changed.



Adding the value 6 to the initial tree will result in the following tree. Put in the thread links and mark the ones that have changed.



Adding the value 10 to the initial tree will result in the following tree. Put in the thread links and mark the ones that have changed.



The three previous examples all added the value as a right child. If the insertion is on the right, where must the thread out of the inserted value go?

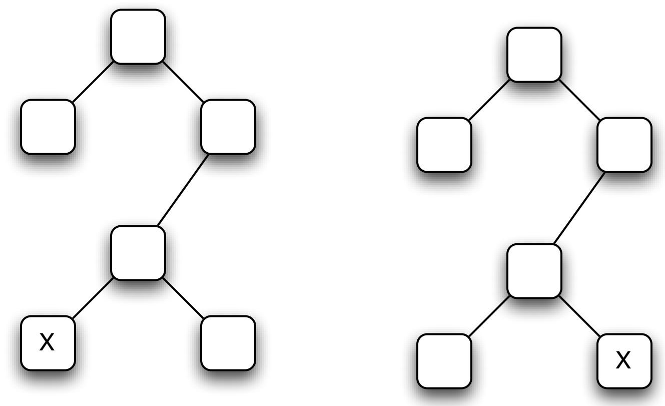
If the insertion is on the right, where is the thread that must be changed to refer to the new node?

Threading BinarySearchTree—Remove

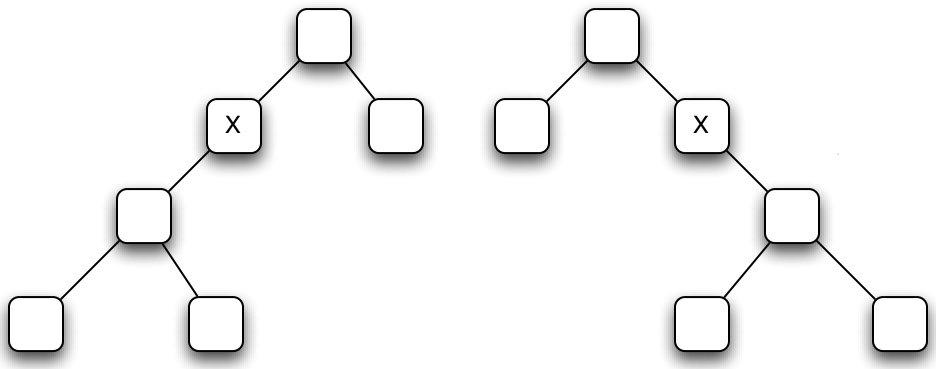
Examine the code for remove in `BinarySearchTree.java`. There are three cases for remove. If the node to be removed has no children, it is just removed. If the node to be removed has one child, its child is moved up in the tree. If the node to be removed has two children, its in-order predecessor's data value is copied up. The predecessor is then removed.

The only changes to the structure of tree will involve a node with zero or one children.

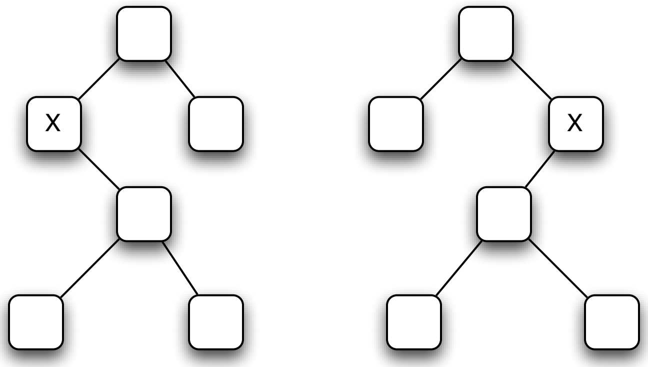
Consider the following trees. The node that will be removed is indicated with an X. Draw the threads and mark the node whose thread will no longer have a target when X is removed.



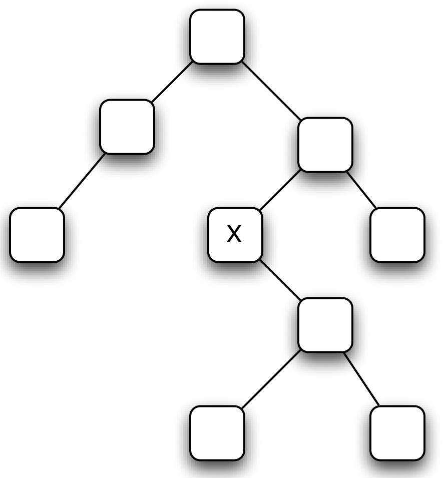
[14]



[15]



[16]



The thread change will always be at the in-order predecessor of the node to be removed. Surprisingly, if the node has no in-order predecessor within the search tree, nothing needs to be done.

Depending on the structure of the tree, there are two cases for finding the in-order predecessor of a node. Identify the cases and give an algorithm for each one.

Case1:

Case 2:

Directed Lab Work

The main classes that you will be working with today are a couple old friends `BinaryTree` and `BinaryNode`, along with a new one `BinarySearchTree`. Take a look at them now, if you have not done so already.

Modifying BinaryTree with Parent References

The binary tree with parent pointers will be the starting point for this lab. If you did not do the last lab, this section has the necessary instructions to add in parent references.

To skip this section, copy the final versions of BinaryNode and BinaryTree from the last lab into the TreePackage folder. Change BinaryTree so that it implements BinaryTreeInterface and then skip ahead to step 10.

Step 1. In the class `BinaryNode`, add a private variable that will hold the parent reference.

Step 2. Add a new constructor that has four arguments: `data`, `left`, `right`, and `parent`.

Step 3. Modify the constructor that takes three arguments to use the new constructor.

Step 4. Create and fully implement three new methods in `BinaryNode`:

```
public BinaryNode<T> getParent()
public void setParent(BinaryNode<T> p)
public boolean hasParent()
```

Checkpoint: Compile BinarySearchTree, BinaryNode and BinaryTree. All tests in TestBinaryTree and TestBST should pass.

Step 5. Make a duplicate of the method `copy()` in `BinaryNode` and add an argument `BinaryNode<T> p` to the duplicate.

Step 6. In the duplicate, set the parent of `newRoot` to be `p`.

Step 7. In the original, set the parent of of `newRoot` to be `null`. (We will assume that if the original version of the copy method is being called, it is the top of the tree being copied and the parent should be null. The second version of copy will be used for all other nodes in the copy.)

Step 8. In both the original and the duplicate, change the two recursive calls to `copy()` so that they pass `newRoot` as the parameter.

Checkpoint: BinaryNode should compile successfully.

Checkpoint: Compile BinaryNode and BinaryTree. All tests in TestBinaryTree and TestBasicAccess should still pass.

The modification of BinaryNode is finished. The next goal is to modify BinaryTree appropriately. Any time a new binary tree is created, parent references for children may need to be set.

Step 9. Anywhere in `BinaryTree` that a left or right child is set, set a parent reference in an appropriate fashion.

Checkpoint: Compile BinaryNode and BinaryTree. All tests in TestBinaryTree and TestBasicAccess should still pass.

Threading the BinaryTree

Step 10. In the class `BinaryNode`, add a private variable that will hold the thread reference.

Step 11. Add a new constructor that has five arguments: `data`, `left`, `right`, `parent`, and `thread`.

Step 12. Modify the constructor that takes four arguments to use the new constructor.

Step 13. Create and fully implement three new methods in `BinaryNode`:

```
public BinaryNode<T> getThread()
public void setThread(BinaryNode<T> target)
public boolean hasThread()
```

Checkpoint: Compile BinarySearchTree, BinaryNode, and BinaryTree. All tests in TestBinaryTree and TestBST should still pass.

Step 14. Create and complete the method `linkSubtreeThreadOut()` in `BinaryNode`. Refer to the pre-lab exercises.

Step 15. In both of the `copy()` methods in `BinaryNode` make a call to `linkSubtreeThreadOut` to thread the left subtree to the root.

Step 16. Create and complete the method `getLeftmostInSubtree()` in `BinaryNode`. Refer to the pre-lab exercises.

Step 17. In the both `copy()` methods in `BinaryNode` add code that will thread the root to the leftmost node in the right subtree.

Checkpoint: Compile BinarySearchTree, BinaryNode, and BinaryTree. All tests in TestBinaryTree and TestBST should still pass.

The modification of BinaryNode is finished. The next goal is to modify BinaryTree appropriately. Any time a new binary tree is created, threads for children may need to be set.

Step 18. Anywhere in `BinaryTree` that a left child is set, set a thread reference for the subtree in an appropriate fashion.

Step 19. Similarly, anywhere in `BinaryTree` that a right child is set, set a thread reference from the root to the leftmost node in the subtree in an appropriate fashion.

Checkpoint: Compile BinarySearchTree, BinaryNode, and BinaryTree. All tests in TestBinaryTree and TestBST should still pass.

It is now time to see if the threads work. The in-order iterator will be changed to use the threads.

Implementing an In-Order Iterator with Threads

Step 20. In the class `BinaryTree`, create a copy of the private inner class `InorderIterator`. Comment out the original.

Step 21. Remove the variable `nodeStack`. Now that threading is available, the stack will not be needed.

Step 22. Refer to the pre-lab exercises and create a private method in `InorderIterator` that will move the current node to the first node to be printed in an in-order traversal.

Step 23. Call the new method in the constructor just after setting the `currentNode` to the root. (Make sure the root is not null before doing so though.)

Step 24. Complete the `hasNext()` method.

Step 25. Complete the `next()` method. It should be much simpler now. It just needs to remember the value to return and then move the current reference. Don't forget to throw `NoSuchElementException` when there are no more elements to be iterated over.

Checkpoint: Compile BinaryNode and BinaryTree. All tests in TestBinaryTree should still pass.

This is the first real test of the threading. To debug the code, it may be helpful to print whenever a node is created (along with the data) and to print whenever a thread is set. When finished, comment out the print statements. They may be useful again in the next section.

Now it is time to make sure that BinarySearchTree respects parent references and threads.

Threading the BinarySearchTree

Step 26. Anywhere in the `add()` method of `BinarySearchTree` that a left or right child is set, parent references and threads must be adjusted. Refer to the pre-lab exercises.

Checkpoint: Compile BinarySearchTree. All the tests except for remove should pass.

Step 27. Anywhere in the `removeNode()` method of `BinarySearchTree` that a left or right child is set or the root is changed, parent references and threads must be adjusted. Refer to the pre-lab exercises. (Of all the methods that collaberate to perform the remove, the only method that affects the structure of the tree is removeNode, so it is the only one where references might need to change.)

Checkpoint: Compile BinarySearchTree. All the tests in TestBST should pass.

Getting Identifiers from a Java Program

Step 1. The application `Identifiers` exists but needs to be completed.

Step 2. Copy *`Small.java`* and *`X.java`* to the default directory that Java's runtime environment uses.

Checkpoint: The application should run. Enter X.java for the file name. The program will open the file for reading then quit.

Step 3. In the method `getPossibleIds()` in `Identifiers` create a loop to read lines from the file.

Step 4. In the loop, read a line using `Scanner` and then use the line to create a `StringTokenizer`.

Step 5. Create another loop that uses the `StringTokenizer` to get tokens (strings) and place them in the binary search tree. (For the delimeters string include any character that would mark the end of a token. For example, in the code `x+=y*eff;` each of the symbols +, * and ; mark the end of an identifier.)

Step 6. In the main, use an in-order iterator to print out the values in the binary search tree.

Final checkpoint: The application should run. Enter X.java for the file name. The list of identifiers should be a b c d e ef g.

Run the application again. Enter Small.java for the file name. This is a very short working java application. The list of possible identifiers should be in alphabetical order and should correctly include the identifiers in the program.

Test the application on other Java files.

Post-Lab Follow Ups

1. Create a new implementation of a threaded binary search tree that uses two thread references, one that links to the in-order predecessor and the other that links to the in-order successor. Do not use a parent reference.

2. Create a new implementation of a threaded binary search tree that uses circular references for the threads. (The last node in an in-order traversal threads back to the first node.)

3. Design and implement a recursive version of the method `linkSubtreeThreadOut()` in `BinaryNode`.

4. Design and implement a recursive version of the method `getLeftmostInSubtree()` in `BinaryNode`.

5. Modify the `Identifiers` application to ignore any text in comments or string constants.

6. Modify the `Identifiers` application to use a second binary search tree that holds Java keywords. Do not add keywords to the list of identifiers.

7. Create a new implementation of a threaded binary search tree that uses thread references, but instead of having the threads give an in-order traversal, create a level-order traversal.

8. An alternate version of a binary search tree allows multiple copies of the same value in the tree. The search tree property will allow equal value nodes in either sub-tree. When adding a duplicate value into the tree, randomly choose a sub-tree of the original value. This will help prevent imbalanced trees. When removing a value from the tree, remove the first copy found.

   Develop and implement this alternate version of a binary search tree. The test classes will need to be changed for the new definition.