CS1102 Semester 2 AY 2003/2003

Annotated Solutions for Lab 4A

Problem 40: Augmented Binary Search Tree

Introduction

This problem is meant to be a easy problem, that could be completed within two hours.

To solve this problem, you need to understand how the given code works. Drawing pictures of BSTs, and thinking about how insertion/deletion can change the five data fields are important. Trying to solve this problem straight away before understanding the code or how insert/delete can affect the tree would cause wasted hours of debugging. It also requires knowledge of how to delete and insert from a doubly linked list.

The given source code is shown in the appendix. The unrelated part of the source code has been removed for simplicity. Therefore the line numbers referred to in this document does not match those in the java file. Let's look at how each data field can be affected by insertion and deletion. We will begin by looking at the simplest one, the parent.

Parent

The parent of a node N should either point to itself, if N is a root, or points to the parent. Therefore, the parent of node N changes, if either N becomes a new root, or if some other node sets its left or right child to N.

When can a node becomes a new root? This happens when the member "root" of BinarySearchTree changed and only happens in line 2 to 10.

We can reset the parent pointer here:

```
  2     public void insert(KeyedItem newItem)
  3     {
  4       root = insertItem(root, newItem);
          root.parent = root;
  5     }
  6
  7     public void delete(Comparable searchKey) throws TreeException
  8     {
  9       root = deleteItem(root, searchKey);
          if (root != null) root.parent = root;
 10     }
```

Note that we do not have to check if root is null after insertItem.

The other case where a parent pointer could change is when some other node has changed its child pointer. This happens in line 135 to 146.

We can update the parent pointer here:

```
135     public void setLeft(TreeNode left)
136     {
137       // Sets the left child reference to left.
138       leftChild  = left;
          if (left != null) left.parent = this;
139     }
140
141     public void setRight(TreeNode right)
142     {
143       // Sets the right child reference to right.
144       rightChild  = right;
```

```
        if (right != null) right.parent = this;
145   }
146 }
```

Size and Height

Initialization of size and height are easy. When a new TreeNode is created, just set both of them to 1.

Since size and height of a node N indicate the number of nodes and the height of the subtree rooted at N, these two data fields need to be updated whenever the subtree of N is changed. The subtree rooted at a node N can only be changed when an item is inserted/deleted from one of its subtree. For insertion, this happens between line 24 and 34 after insertItem() is called on either the left or right subtree.

```
23      // search for the insertion position
24      if (newItem.getKey().compareTo(nodeItem.getKey()) < 0) {
25        // search the left subtree
26        newSubtree = insertItem(tNode.getLeft(), newItem);
27        tNode.setLeft(newSubtree);
          tNode.updateSizeAndHeight();
28        return tNode;
29      }
30      else { // search the right subtree
31        newSubtree = insertItem(tNode.getRight(), newItem);
32        tNode.setRight(newSubtree);
          tNode.updateSizeAndHeight();
33        return tNode;
34      }
```

For deletion, this happens in four different places, whenever deleteItem or deleteLeftmost is called.

```
49        // else search for the item
50        else if (searchKey.compareTo(nodeItem.getKey()) < 0) {
51          // search the left subtree
52          newSubtree = deleteItem(tNode.getLeft(), searchKey);
53          tNode.setLeft(newSubtree);
            tNode.updateSizeAndHeight();
54        }
55        else { // search the right subtree
56          newSubtree = deleteItem(tNode.getRight(), searchKey);
57          tNode.setRight(newSubtree);
            tNode.updateSizeAndHeight();
58        }

90      // retrieve and delete the inorder successor
91      else {
92        replacementItem = findLeftmost(tNode.getRight());
93        tNode.setItem(replacementItem);
94        tNode.setRight(deleteLeftmost(tNode.getRight()));
          tNode.updateSizeAndHeight();

95        return tNode;
96      }
97    }

99    protected TreeNode deleteLeftmost(TreeNode tNode)
100   {
101     if (tNode.getLeft() == null) {
102       return tNode.getRight();
103     }
104     else {
105       tNode.setLeft(deleteLeftmost(tNode.getLeft()));
          tNode.updateSizeAndHeight();
```

```
        return tNode;
      }
    }

```

Method updateSizeAndHeight() can be written as a method in class TreeNode as follows:

```
private void updateSizeAndHeight()
{
  if (leftChild == null && rightChild == null) {
    size = height = 1;
  } else if (leftChild == null) {
    size = rightChild.size + 1;
    height = rightChild.height + 1;
  } else if (rightChild == null) {
    size = leftChild.size + 1;
    height  = leftChild.height + 1;
  } else {
    size = leftChild.size + rightChild.size + 1;
    height = Math.max(leftChild.height, rightChild.height) + 1;
  }
}
```

Someone observed that we can update the size and height in setLeft() and setRight(). This is correct for this particular implementation. However, for other implementation, it might not be the case that any modification to the subtree rooted at node N results in a change to the left or right child of N.

Predecessor and Successor

The predecessor and successor fields are not related to the structure of the tree, but to the values of the items stored in the tree. An important observation is that these two references form a doubly linked list. Thus, updates of predecessor and successor when insert/delete is performed on the tree are the same as updates of next/prev references in a doubly linked list.

To insert a new item N, we first need to find out the position of N in the doubly linked list. This is easy once you made the observation that the parent of N must be either the successor (if it is inserted to the left) or the predecessor (if inserted to the right) of N. The next question is, when to update the predecessor and successor? Obviously this must be done when a new node is created, that is, after line 18. But at line 18, we have no access to the parent of the newly created tNode. We can solve this by passing in the parent node to insertItem.

```
12    protected TreeNode insertItem(TreeNode tNode, KeyedItem newItem,
                                      TreeNode parent)
13    {
14      TreeNode newSubtree;
15      if (tNode == null) {
16        // position of insertion found; insert after leaf
17        // create a new node
18        tNode = new TreeNode(newItem, null, null);
          // if parent is larger than tNode,
          //    insert tNode before parent
          // else
          //    insert tNode after parent
19        return tNode;
20      }
```

Finally, we consider how to update successor and predecessor when a node N has been deleted. Deletion of a node is done in deleteNode, in which the argument tNode is removed from the tree. Case 1, 2 and 3 are straight forward – since we are removing tNode from the tree, we just remove it from the double linked list formed by the successor and predecessor reference. In case 4, tNode is not deleted, but instead, was replaced by its successor. Hence, we do not remove tNode, but we remove its successor instead.

```java
  protected TreeNode deleteNode(TreeNode tNode)
  {
    // Algorithm note: There are four cases to consider:
    //   1. The tNode is a leaf.
    //   2. The tNode has no left child.
    //   3. The tNode has no right child.
    //   4. The tNode has two children.
    // Calls: findLeftmost and deleteLeftmost
    KeyedItem replacementItem;

    // test for a leaf
    if ( (tNode.getLeft() == null) &&
         (tNode.getRight() == null) ) {
      // remove tNode from doubly linked list
      return null;
    }

    // test for no left child
    else if (tNode.getLeft() == null) {
      // remove tNode from doubly linked list
      return tNode.getRight();
    }

    // test for no right child
    else if (tNode.getRight() == null) {
      // remove tNode from doubly linked list
      return tNode.getLeft();
    }

    // there are two children:
    // retrieve and delete the inorder successor
    else {
      replacementItem = findLeftmost(tNode.getRight());
      tNode.setItem(replacementItem);
      tNode.setRight(deleteLeftmost(tNode.getRight()));
      // remove tNode's successor from doubly linked list
      return tNode;
    }
  }
```

One interesting note is that we can simplify the code for case 4 in deletion. We do not really need to traverse the tree to find the replacement item using findLeftmost( ) anymore, because the successor of tNode points to the replacement item directly.

Appendix: Original Source Code for Binary Search Tree.

```java
class BinarySearchTree extends BinaryTreeBasis {
  public void insert(KeyedItem newItem)
  {
    root = insertItem(root, newItem);
  }

  public void delete(Comparable searchKey) throws TreeException
  {
    root = deleteItem(root, searchKey);
  }

  protected TreeNode insertItem(TreeNode tNode, KeyedItem newItem)
  {
    TreeNode newSubtree;
    if (tNode == null) {
      // position of insertion found; insert after leaf
      // create a new node
      tNode = new TreeNode(newItem, null, null);
      return tNode;
```

```
    }
    KeyedItem nodeItem = (KeyedItem)tNode.getItem();

    // search for the insertion position
    if (newItem.getKey().compareTo(nodeItem.getKey()) < 0) {
      // search the left subtree
      newSubtree = insertItem(tNode.getLeft(), newItem);
      tNode.setLeft(newSubtree);
      return tNode;
    }
    else { // search the right subtree
      newSubtree = insertItem(tNode.getRight(), newItem);
      tNode.setRight(newSubtree);
      return tNode;
    }
  }

  protected TreeNode deleteItem(TreeNode tNode, Comparable searchKey)
  {
    TreeNode newSubtree;
    if (tNode == null) {
      throw new TreeException("TreeException: Item not found");
    }
    else {
      KeyedItem nodeItem = (KeyedItem)tNode.getItem();
      if (searchKey.compareTo(nodeItem.getKey()) == 0) {
        // item is in the root of some subtree
        tNode = deleteNode(tNode);  // delete the item
      }
      // else search for the item
      else if (searchKey.compareTo(nodeItem.getKey()) < 0) {
        // search the left subtree
        newSubtree = deleteItem(tNode.getLeft(), searchKey);
        tNode.setLeft(newSubtree);
      }
      else { // search the right subtree
        newSubtree = deleteItem(tNode.getRight(), searchKey);
        tNode.setRight(newSubtree);
      }
    }
    return tNode;
  }

  protected TreeNode deleteNode(TreeNode tNode)
  {
    // Algorithm note: There are four cases to consider:
    //   1. The tNode is a leaf.
    //   2. The tNode has no left child.
    //   3. The tNode has no right child.
    //   4. The tNode has two children.
    // Calls: findLeftmost and deleteLeftmost
    KeyedItem replacementItem;

    // test for a leaf
    if ( (tNode.getLeft() == null) &&
         (tNode.getRight() == null) ) {
      return null;
    }

    // test for no left child
    else if (tNode.getLeft() == null) {
      return tNode.getRight();
    }

    // test for no right child
    else if (tNode.getRight() == null) {
      return tNode.getLeft();
    }

```

```java
    // there are two children:
    // retrieve and delete the inorder successor
    else {
      replacementItem = findLeftmost(tNode.getRight());
      tNode.setItem(replacementItem);
      tNode.setRight(deleteLeftmost(tNode.getRight()));
      return tNode;
    }
  }

  protected TreeNode deleteLeftmost(TreeNode tNode)
  {
    if (tNode.getLeft() == null) {
      return tNode.getRight();
    }
    else {
      tNode.setLeft(deleteLeftmost(tNode.getLeft()));
      return tNode;
    }
  }

}


class TreeNode {
  private Object item;
  private TreeNode leftChild;
  private TreeNode rightChild;

  public TreeNode(Object newItem)
  {
    // Initializes tree node with item and no children.
    item = newItem;
    leftChild  = null;
    rightChild = null;
  }

  public TreeNode(Object newItem, TreeNode left, TreeNode right)
  {
    // Initializes tree node with item and
    // the left and right children references.
    item = newItem;
    leftChild  = left;
    rightChild = right;
  }

  public void setLeft(TreeNode left)
  {
    // Sets the left child reference to left.
    leftChild  = left;
  }

  public void setRight(TreeNode right)
  {
    // Sets the right child reference to right.
    rightChild  = right;
  }
}
```