CSE 373 Spring 2006

Data Structures and Algorithms
Assignment #3

Due: Wednesday April 19

In this assignment, you will have to implement the basic functions needed to build and search *binary search trees* (abbreviated BST). Your implementation should be such that the nodes of the tree contain an integer and links (possibly null) to the left and right children of the node. In the programming part of the assignment, **you are not allowed to use external nodes** like in your book (i.e., leaves that do not contain an integer element) and **you are not allowed to use parent links**.

It is strongly recommended that you do Paper Exercises 1 and 2 before you start the programming part of the assignment.

Paper Assignment
1. Binary Search Trees(BSTs)
(a) Draw the BST where the data value at each node is an integer and the values are entered in the *following order*: 36, 22, 10, 44, 42
(b) Draw the BST after the following insertions have been done in the tree of part (a): 16, 25, 3, 23, 24 (there is no attempt at being tricky; parts (a) and (b) could be combined into a single one but we want you to check your work so that you don't make a silly mistake)
(c) Now draw the tree after deletions of 42, 23 and 22 *in this order*
(d) Write down the order on which the node values are reached when the BST of part (c) is traversed (i) in inorder (ii) in postorder (iii) in preorder
(e) What is the height of the tree of part (c)? Which nodes have maximal depth in the tree of part (c)?

2. In the implementation of an iterator for a sequence implemented as a BST, the *next()* method must return the *inorder* successor of a node, say A. Of course it is assumed that all predecessors of A in *inorder* have been visited in *inorder*. With the constraint that you are not allowed to start from the root every time you ask for *next()*, what information must be kept so that the successor can be found in $O(h)$ time in the worst case, where h is the maximum height of the tree? What data structure can be used to keep this information? Give a pseudo-code implementation of *next*
(Hint: look at the inorder successors of 16 and 31, for example, in the tree of Exercise 1 part (c). How did you algorithmically reach the successors? Did you have to keep all ancestors? Did you visit these ancestors in any particular order to reach the successor? Look at the Programming assignment hints!

Would it help if you had parent links? By help we mean that you have to show in general (not on a particular example) (i) whether it would be easier to program and (ii) whether it would go faster in an Big-Oh sense.

3. Exercise C-7.7 ("show" means "prove")

4. Exercise C-7.31 only in the case of a binary tree. Write the "method" in pseudo-code.

Programming Assignment

In this assignment you have to implement some basic Java methods for BST's (with a little twist) and an iterator (i.e., an inorder sequencing). Each node in the tree has a data element (an integer) and links (possibly null) to its left and right children (as mentioned above, no parent link and no external node).

Specifically, the methods that you have to implement are:

1. For the public class *NoDupBST*:

   - *insertordelete(item).* If *item* is not in the tree, it is inserted in the tree. If *item* is already in the tree, it is deleted from the tree.
   - *contains(item)* returns true if *item* is in the tree, false otherwise
   - *size()* returns the number of nodes in the tree. This method should run in O(1).
   - *iterator()* returns an iterator for the BST
   - *print()* prints the elements of the tree in inorder. one element per line.

2. The nested class *NoDupBSTIterator* has 2 methods:

   - *hasNext()* returns true if there is another element in the BST when traversed in *inorder*
   - *next()* returns the "next" item in the BST where "next" means *inorder* successor.

Implementation Notes and Hints

In addition to the constraints already mentioned, follow the template that is provided with in particular:

- Private class for a tree node
- A single instance variable to refer to the root of the BST
- Your methods should only traverse the parts of the tree that are necessary. For example in *contains(item)* you should only follow a single path in the BST from the root to the node you are looking for.
- The **trickiest part** of the assignment is the implementation of *next()*. You are not allowed to start from the root of the tree each time *next* is called or even in the cases when you need to back-up to the parent of the node you are at. Instead the iterator should keep track of the nodes that you will have to back-up to before finding the in-order successor (see Paper Exercise # 2). Keeping these nodes on a *stack* is a simple way to do the bookkeeping (use private data in the iterator class for this purpose). It can also help in the implementation of *hasNext()*.
- The method *print()* **must** use the iterator. This is a good way to test if your iterator works.

You should write your own test files.

A template for your program is attached and can be found on the Web also.

Instructions for turnin are forthcoming.

```
import java.util.*;

public class NoDupBST {
    private static class TreeNode {
int data;
TreeNode left;
TreeNode right;

        // add constructors here
    }

    private TreeNode root;
    // add more instance variables here

    public NoDupBST() {
      throw new UnsupportedOperationException();
    }

    // if item is not present, add it and return true.
    // if item is present, delete it from the tree and return false.
    // You are encouraged to use helper methods to make the code
    // more manageable.
    public boolean insertOrDelete(int item) {
      throw new UnsupportedOperationException();
    }

    // you are not allowed to do unnecessary traversing here --
    // take advantage of the sorted structure.
    public boolean contains(int item) {
      throw new UnsupportedOperationException();
    }

    // size() needs to be O(1)
    public int size() {
      throw new UnsupportedOperationException();
    }

    // hint - this method should be one line
    public NoDupBSTIterator iterator() {
      throw new UnsupportedOperationException();
    }

    // Print inorder, one per line.
    // You must use your iterator here.  Recursion not allowed.
    public void print() {
      throw new UnsupportedOperationException();
    }
    public class NoDupBSTIterator {
      // You are allowed to use space proportional to the HEIGHT of the
      // tree, not the size of the tree.
      // Traversing and storing all elements into a list and walking
      // through the list will therefore earn you ZERO points.

      // You do not need to worry about the behavior of next if
      // other operations happen concurrently, i.e. you can assume
      // the list is "frozen" while you are iterating.
      // (Normally a ConcurrentModificationException would be thrown
      // if an insertOrDelete happened in between two next calls of the
      // same iterator.

      // add state variables here

      public NoDupBSTIterator() {
        throw new UnsupportedOperationException();
      }

      // should be O(1)
      public boolean hasNext() {
        throw new UnsupportedOperationException();
      }

      public int next() {
        throw new UnsupportedOperationException();
      }
    }
}
```