COMP 250, Winter 2009
Assignment 4
Prof. Michael Langer

Posted Wednesday April 1
Due Tuesday April 14 at 1:30 PM
LATE ASSIGNMENTS WILL NOT BE ACCEPTED
Please follow the appropriate hand-in instructions for each question

Prepared by Mansoor Siddiqui, Neeraj Tickoo, Julien Villemure.
Questions about this assignment should be directed to the TAs above.

Question 1 – (25 points) – IntegerSet ADT

Consider the IntegerSequence ADT, which models an ordered list of integers. Such a sequence is not necessarily sorted and may contain duplicates. The following **constant-time** operations are defined on sequences:

- empty(S)
  Returns whether the sequence S contains any elements at all. (True/False)

- add(a,S)
  Returns the sequence composed of the elements of S with element "a" in front.
  Example: add(1, (3, 5, 1, 2)) = (1, 3, 5, 1, 2)

- first(S)
  Returns the first element of S.
  Example: first((3, 5, 1, 2)) = 3

- rest(S)
  Returns the sequence composed of all elements of S except the first.
  Example: rest((3, 5, 1, 2)) = (5, 1, 2)

Now consider another ADT, called IntegerSet, which is a collection of **unique** integers (**no duplicates**). The following operations are defined on those sets:

a) insert(a,X)
   Returns the set of all elements of set X with element "a" added if it is not present.

b) delete(a,X)
   Returns the set of all elements of set X with element "a" removed if it is present.

c) member(a,X)
   Returns whether set X contains element "a". (True/False)

d) union(X,Y)
   Returns the set corresponding to the union of sets X and Y.

e) intersection(X,Y)
   Returns the set corresponding to the intersection of sets X and Y.

We can build one abstract data type on top of another. For instance, we can model an IntegerSet as an IntegerSequence with no duplicates. Furthermore, if we restrict ourselves to sorted IntegerSequences, the above operations on IntegerSets can be made more efficient.

Your task is to write pseudocode for the 5 operations of the IntegerSet ADT (5 points each), where IntegerSets are implemented with IntegerSequences of **sorted unique elements**. All operations must have a **linear running time**, i.e. O(n) where n is the size of the set (for union and intersection, n is the size of X plus the size of Y).

You are only allowed to use sequences and operations on sets/sequences, as defined above. Any use of arrays or object allocation (such as the `new` keyword in Java) or other data structures will immediately result in a grade of zero. This is meant to be an abstract exercise and you should not be thinking about Java.

Please turn in your pseudocode **on paper in class**.

Question 2 – (35 points) – Binary Search Trees

In Node.java you are given starter code for a binary tree, wherein each node has a value and two children nodes (*left* and *right*). Each node also has a reference to its parent. The tree is generic and therefore values are, at compilation time, of unknown parametric type T.

In order to implement a binary search tree, values must be compared. In assignment 2 you have become familiar with the use of a Comparator to compare objects of parametric types. The same idea will be applied here. The tree's constructor requires a Comparator object; you must use it whenever values must be compared.

Uniqueness of nodes is assumed. **There cannot be 2 nodes with equal values in the tree.**

a) (5 points) Implement `insert(T element)` **recursively**.

   This method inserts the given element into the tree such that the binary search property is preserved. In other words, this is a *sorted insert* operation. Your implementation must be **O(h)** where h is the height of the tree.

b) (5 points) Implement `inorderPrint()` **recursively**.

This method does an in-order traversal of the tree, as seen in class, and prints values of nodes (one value per line). Since an in-order traversal of a binary search tree visits nodes in sorted order, your implementation should output sorted values. Running time must be **O(n)** where n is the number of elements in the tree.

c) (25 points) Implement `makeRoot(T element)`

This method first searches for the given element in the tree. If it is not found, there is nothing further to be done. If the element is found, it is "promoted" to the root position. In other words, the tree is re-arranged so that

- The element is now the root
- The tree is *still* a binary search tree (everything is *still sorted*)

You will notice that many nodes will be shifted around. This is okay as long as the binary search property is preserved. The method returns the node that is now the root.

Your algorithm should execute in time **O(h)** where h is the height of the tree.

Hints and notes:

- Read the comments in the starter code! They are meant to guide you.
- See TestNode.java for an example of how you can test your code. You may modify this file as you wish. In fact, you are strongly encouraged to write your own tests.
- Note the following methods implemented for you in class Node:

```
insert(Collection<? extends T> elements)
insert(T[] elements)
```

  These methods add multiple elements to the tree in one convenient call. They call the other insert() method (that you must implement) on each element of the Collection or Array. See TestNode.java for examples.
- The output produced by inorder traversal should not be affected by makeRoot(), since the latter should keep the tree sorted. You can use this fact to test your algorithm.

Submit **Node.java** on WebCT. You do not need to submit TestNode.java or whatever tests you have written, as they will not count toward your grade.

Question 3 – (40 points) – Hashing

In the following questions, you will again be using the Song class described in Assignment 2. You will now be storing a list of songs in a hash table. This allows song data to be quickly retrieved given the song's name.

a) (15 points) Hash Table Implementation

   Implement the following methods in the provided HashTable class: the constructor, put(), get(), and remove(). Your implementation should use the separate chaining approach discussed in lecture 29. You can test your implementation by running the tests defined in the main method of the HashTableTester class.

b) (10 points) Polynomial Hash Code Implementation

   Implement the method getHashCode() in the PolynomialHashCode class. This method takes a string and converts it to an integer using the "hashing strings" method described in lecture 29, page 3. Your method should work for any integer p (which is the parameter of the PolynomialHashCode constructor).

   Note: Java does not automatically detect integer overflow, so if an integer becomes too big, it will just wrap around and take on a negative value. Integer overflow is okay, and is to be expected sometimes when using polynomial hash codes.

Submit **all java files** on WebCT.

Provide brief answers to the following discussion questions and **submit them electronically in a plain text file called discussion.txt**:

c) (2 points) In Assignment 2, we used an ArrayList to represent a song playlist. Why would it not be a good idea to use a hash table to represent a song playlist?

d) (2 points) In the HashTable.HashEntry class, why does it make sense to have a setValue() method, but not a setKey() method?

e) (9 points) Test your hash table for various values of the parameter p in the PolynomialHashCode constructor, and include a selection of these tests in your HashTableTester.main(). For your selected tests, briefly discuss which values of p give a better spread of the songs over the hash table bucket, and which give a worse spread. Your results sometimes may be ambiguous -- this is ok. Just state what your tests seem to show.

f) (2 points) Describe what the getHashCode() method of class PolynomialHashCode does when p = 1. For what kinds of data might this hash function work poorly?