1. [7 points] Please give short answers to the following, giving reasons where called for. Unless a question says otherwise, time estimates refer to asymptotic bounds ($O(\cdot \cdot \cdot)$, $\Omega(\cdot \cdot \cdot)$, $\Theta(\cdot \cdot \cdot)$). Always give the simplest bounds possible ($O(f(x))$ is “simpler” than $O(f(x) + g(x))$ and $O(Kf(x))$).

a. The hash tables we presented in lecture use (unordered) linked lists to hold the contents of each bucket. Suppose that we replace these with some kind of balanced binary search tree (e.g., red-black trees), so that each bucket’s data is stored in its own tree. Assuming also that the number of buckets remains fixed regardless of the number of data items, $N$, stored the table, give a worst-case bound for the time to search for an item in this table, as a function of $N$. We make no assumption about how “good” the hash function is.

In the worst case, all keys go into one bucket, so that the hash table reduces to a balanced search tree. Thus, the search time is $\Theta(lg N)$.

b. How does your answer to (a) change if in addition I say that we only consider sets of data for which the hash function is “good”, and divides the data evenly among the buckets?

Now there are only $N/k$ keys in any given bucket. But since the problem says that $k$, the number of buckets, is a fixed constant, the search time is $\Theta(lg N/k) = \Theta(lg N - lg K) = \Theta(lg N)$, as before.

c. If we represented Set2Ds in Project 2 as simple lists of points, what would be the running time of the simulate command, as a function of $N$, the number of particles, and $S$, the number of time steps needed?

We have to process each pair of points to search for the next collision, and we would do this $S$ times. There are $N(N-1)/2$ pairs of points, so the result is $\Theta(SN(N-1)/2) = \Theta(SN^2)$. 
d. Suppose that you knew that node $k$ (and only that one node) in a heap containing $N$ items violated the heap constraint. How long would it take to reorganize the heap to fix the problem, as a function of $N$?

You would either have to move node $k$ up or down in the heap. In either case, the time is limited by the height of the heap, so the time required is $\Theta(\lg N)$.

e. Suppose that you knew that exactly one node in a heap containing $N$ items violated the heap constraint, but did not know which one. How long would it take to reorganize the heap to fix the problem, as a function of $N$?

Moving the heap item still has the same cost: $\Theta(\lg N)$. But this time, you also have to find it, which requires $\Theta(N)$ cost, giving an overall total of $\Theta(N + \ln N) = \Theta(N)$.

f. Starting with a sorted sequence of $N$ items, 10 randomly chosen pairs of items are swapped (we don’t know which). How would you go about restoring the list to proper order, so as to do so as fast as possible? How long would this take, as a function of $N$? Assume that the only operation you may perform on the items is to compare them (i.e., $< \text{or} \leq$). You may assume the sequence is an array or doubly linked structure, but state your assumption if it matters.

Swapping 10 pairs of items introduces at most $10N$ inversions, so a simple insertion sort will work in linear ($\Theta(N)$) time, which is probably best.

g. Draw a 2-4 tree that contains 7 keys and whose height is 4, if possible. If this is not possible, say why. (To be clear, the height is the distance from the root to the (empty) leaves at the bottom; a 2-4 tree containing just a root node and its leaves has height 1).

This is not possible. Each node of a (non-empty) tree has at least 2 children and one key. Thus, there will be at least 1 key at the root, 2 keys at the next level, and 4 keys at the next. Thus the height of the tree is limited to 3.

2. [1 point] Lewis Carroll wrote a take-off on Longfellow’s epic poem The Song of Hiawatha called Hiawatha’s Photographing, and attached the following preface:

“In an age of imitation, I can claim no special merit for this slight attempt at doing what is known to be so easy. Any fairly practised writer, with the slightest ear for rhythm, could compose, for hours together, in the easy running metre of ‘The Song of Hiawatha.’ Having, then, distinctly stated that I challenge no attention in the following little poem to its merely verbal jingle, I must beg the candid reader to confine his criticism to its treatment of the subject.” [from Lewis Carroll, Selected Poems, Keith Silver, ed., Carcanet Press Limited, 1995.]

This preface seems kind of pointless to me. Why do you suppose he wrote it?

The preface is itself a joke. Though formatted to look like ordinary prose, it is actually in the same trochaic meter as the poem.
3. [7 points] Given the following class definition:

```java
class IntTree {
    public IntTree (int data, IntTree left, IntTree right) {
        this.data = data; this.left = left; this.right = right;
    }

    public final int data;
    public final IntTree left, right;
}
```

fill in the following method to agree with its comment. For example, if T contains the tree below and x is 15, the value returned is 27 (this is the first value > 15 encountered in the sequence 7, 10, 8, 2, 3, 14, 27, 17, 16, 39). Add any methods and classes you need.

```java
/** Assuming that all data in T is non-negative (T is NOT necessarily a * binary search tree), returns the first value in T that is > X, or * -1 if there is none. Here, the "first" value is the first that * occurs top to bottom, left to right. */
int firstBigger (IntTree T, int x) {
    LinkedList<IntTree> queue = new LinkedList<IntTree> ();
    queue.add (T);
    while (queue.size () > 0) {
        IntTree node = queue.removeFirst ();
        if (node == null)
            continue;
        if (node.data > x)
            return node.data;
        queue.add (node.left);
        queue.add (node.right);
    }
    return -1;
}
```
4. [6 points] Here is another example of the Visitor Pattern, discussed in lecture:

```java
/** An Action acts on nodes of a BinTree, and also guides traversals
 * over the tree. */
interface Action<Data> {
    /** Operate on tree node T. Return 0 if T's children are not to
     * be traversed, 1 if only the left child is to be traversed,
     * 2 if only the right child, and 3 if both children are to be
     * traversed. */
    int act (BinTree<Data> T);
}

class BinTree<Data> {
    /** A BinTree whose label data is DATA, and whose left and right
     * children are LEFT and RIGHT. LEFT and RIGHT must not be
     * null. */
    public BinTree (Data data, BinTree<Data> left, BinTree<Data> right) {
        this.data = data; this.left = left; this.right = right;
    }

    /** Apply ACTION.act to each node in THIS in preorder, using the
     * values returned by action to determine which children are
     * traversed (see comment on act, above). */
    void preorderTraverse (Action<Data> action) {
        int next = action.act (this);
        if (next == 1 || next == 3)
            left.preorderTraverse (action);
        if (next == 2 || next == 3)
            right.preorderTraverse (action);
    }

    public final Data data;
    private BinTree<Data> left, right;
}

/** Represents an empty tree. Its data field is meaningless. */
class EmptyTree<Data> extends BinTree<Data> {
    public EmptyTree () {
        super (null, null, null);
    }

    void preorderTraverse (Action<Data> action) {
    }
}
```

Because the `left` and `right` fields are private, the only way to traverse these trees is by using the method `preorderTraverse`. Using these definitions, fill in the methods on the next page to fulfill their comments. Add any additional methods or classes you need.

Copyright (c) 2008 by P. N. Hilfinger. Reproduction prohibited.
/** Assuming T is a binary search tree, prints "FOUND" (once) iff X is 
* present in T. In the worst case, operates in time proportional 
* to the height of T. */
static void contains (BinTree<String> T, String X) {
    T.preorderTraverse (new Finder (X));
}

/** Assuming T is a binary search tree, returns a list of all strings 
* in T that are >= L and <= U (string comparison). The list may be 
* in any order. In the worst case, operates in time proportional 
* to the height of T plus the size of the returned list. */
static List<String> inRange (BinTree<String> T, String L, String U) {
    ArrayList<String> result = new ArrayList<String> ();
    T.preorderTraverse (new Accum (L, U, result));
    return result;
}

/* See next page for Finder and Accum. */
class Finder implements Action<String> {
    Finder (String x) {
        this.x = x;
    }

    private String x;

    public int act (BinTree<String> T) {
        int cmp = T.data.compareTo (x);
        if (cmp == 0) {
            System.out.println ("FOUND");
            return 0;
        } else if (cmp < 0)
            return 2;
        else
            return 1;
    }
}

class Accum implements Action<String> {
    Accum (String L, String U, List<String> result) {
        this.L = L; this.U = U; this.result = result;
    }

    private String L, U;
    private List<String> result;

    public int act (BinTree<String> T) {
        int cL = T.data.compareTo (L),
            cR = T.data.compareTo (U);
        if (cL >= 0 && cR <= 0)
            result.add (T.data);
        int v;
        v = 0;
        if (cL >= 0)
            v += 1;
        if (cR <= 0)
            v += 2;
        return v;
    }
}