UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61B
Fall 2011

P. N. Hilfinger

Test #3 Solutions

1. [6 points] Provide simple and tight asymptotic bounds for the running times of the following methods (including any methods they call) as a function of the value of N, the length of `A`. If possible, give a single Θ bound that always describes the running time, regardless of the input. Otherwise give an upper and a lower bound. Give brief justifications of your answers.

a. [1 point]

```
static List<Integer> reverse(List<Integer> A) {
    ArrayList<Integer> R = new ArrayList<Integer>();
    for (Integer x : A) {
        R.add(0, x);
    }
    return R;
}
```

Solution: *We insert at the beginning of a list stored as an array each time. The time required is therefore* $1 + 2 + \ldots + N \in \Theta(N^2)$.

b. [1 point]

```
static List<Integer> reverse(List<Integer> A) {
    ArrayList<Integer> R = new ArrayList<Integer>();
    for (int i = A.length - 1; i >= 0; i -= 1) {
        R.add(A.get(i));
    }
    return R;
}
```

Solution: *We insert at the end of a list stored as an array each time. The amortized time per insertion is therefore constant, giving a bound on the loop of* $\Theta(N)$.

c. [1 point]

```
/** True iff X is among elements #L through #U of A,
 *  assuming A is sorted. */
static boolean contains(ArrayList<Integer> A, int L, int U, int x) {
    if (L > U)
        return false;
    int m = (L + U) / 2;
    if (A.get(m) < x)
        return contains(A, m + 1, U, x);
    else if (A.get(m) > x)
        return contains(A, L, m - 1, x);
    else
        return true;
}
```

Initial call: `contains(someList, 0, someList.size()-1, someValue)`.

Solution: *This is a standard binary search, giving a running time in* $O(\lg N)$. *The search stops on finding* `x`*, which can happen immediately, so the only thing we can say about the lower bound is that it is* $\Omega(1)$.

d. [1 point]

```
/** True iff X is among elements #L through #U of A,
 *  assuming A is sorted. */
static boolean contains(LinkedList<Integer> A, int L, int U, int x) {
    if (L > U)
        return false;
    int m = (L + U) / 2;
    if (A.get(m) < x)
        return contains(A, m + 1, U, x);
    else if (A.get(m) > x)
        return contains(A, L, m - 1, x);
    else
        return true;
}
```

Initial call: `contains(someList, 0, someList.size()-1, someValue)`.

Solution: *Although this is a binary search, the* `.get` *method on a linked list requires* $\Theta(N)$ *time in the worst case. If we cleverly use a ListIterator, so as to access each midpoint element starting from the last one, we get an upper bound of* $N/2 + N/4 + \ldots + 1 \in O(N)$*, but if we just use* `.get` *as given, we get a bound of* $O(N \lg N)$ *(*$\lg N$ `.get`*s, each taking* $O(N)$ *time).*
The lower bound is $\Omega(1)$*, as for (c), and for the same reason.*

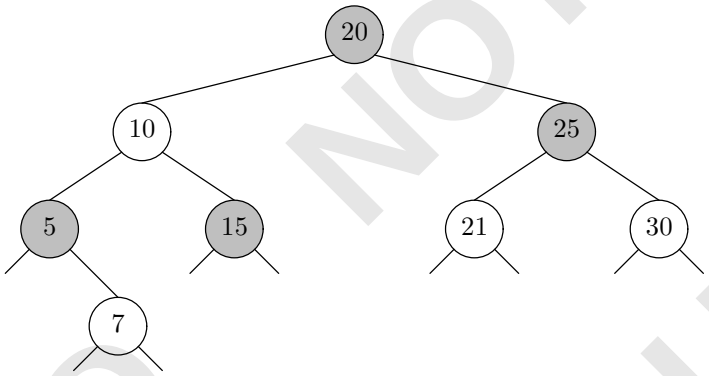
e. [2 [points]

```
static void munge(int[] A, Random R) {
   for (int i = 1; i < A.length; i += 1) {
       if (A[i-1] > A[i])
           return;
   }

   for (int i = 0; i < 8; i += 1) {
       k = R.nextInt(A.length); /* 0 <= k < A.length */
       int t = A[k]; A[k] = A[0]; A[0] = t;
   }

   insertionSort(A);
}
```

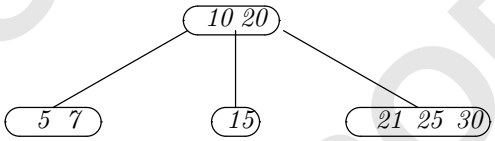
Solution: *If the array is unsorted, the first loop will cause it to exit in time bounded by* $O(N)$ *and* $\Omega(1)$*. If the array is sorted, the first loop requires* $\Theta(N)$ *time. The second loop then introduces up to* $O(8N)$ *inversions in* $\Theta(1)$ *time. This in turn means that insertion sorting will require* $\Theta(8N) = \Theta(N)$ *time. Overall then, we have bounds of* $O(N)$ *and* $\Omega(1)$*.*

2. [5 points]

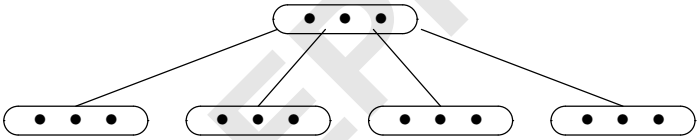
a. Convert the following red-black tree into the corresponding 2-4 tree (shaded nodes are black).



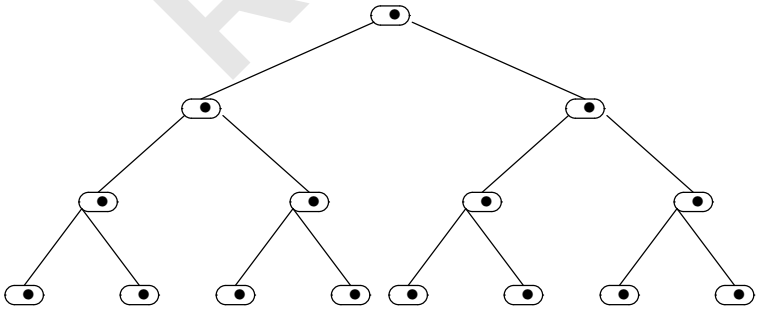
Solution:



b. Show a (2,4) tree of minimum height that contains 15 keys (we're not interested in specific key values).

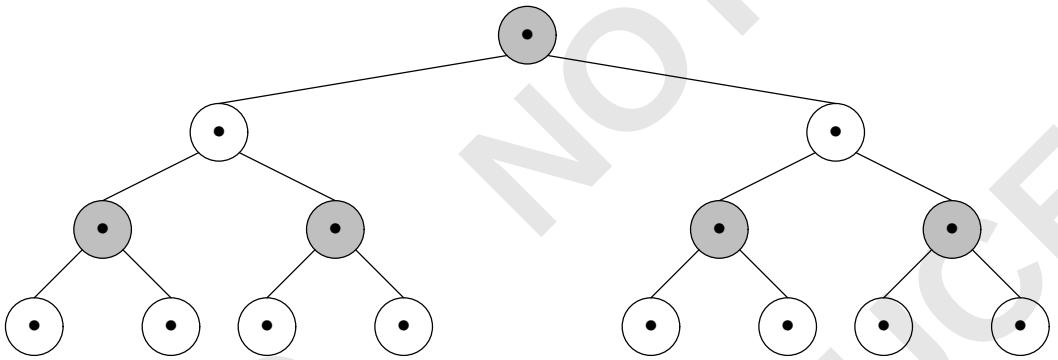


c. Show a (2,4) tree of maximum height that contains 15 keys.



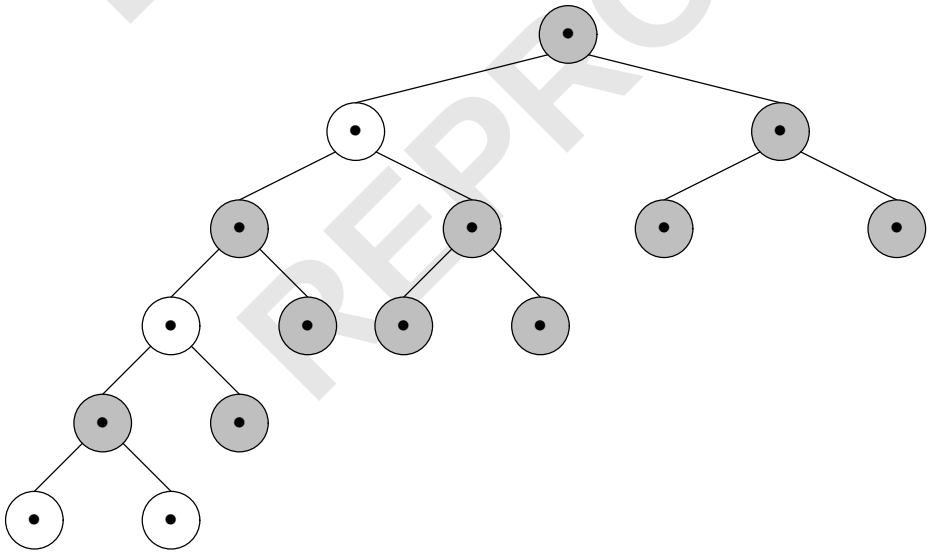
d. Show a minimum-height red-black tree containing 15 keys.

Solution:



e. Show a maximum-height red-black tree containing 15 keys.

Solution: *The idea is to extend one path from the root alternating red and black and to extend the other subtrees to obey the rules while using as few nodes as possible. The height of such a tree is constrained by its relationship to (2,4) trees and the fact that at most, a (2,4) node translates to a subtree of height 1.*



3. [2 points] A certain proposed implementation of a set type uses an open-address hash table implementation (with an ArrayList to hold the values), and does not allow null values in the set. It has the following implementation for its `remove` method:

```
/** Remove OBJ != null from the table.  Returns true if OBJ was in
 *  the table, and otherwise false. */
public boolean remove(Object obj) {
    if (obj == null)
        throw new IllegalArgumentException("null value in set");
    int hash = obj.hashCode() % _values.size();
    if (hash < 0)
        hash = -hash;
    int i;
    i = hash;
    do {
        if (_values.get(i) == null)
            break;
        if (obj.equals(_values.get(i))) {
            _values.put(i, null);
            return true;
        }
        i = (i + 1) % _values.size();
    } while (i != hash);
    return false;
}
```

A simple unit test for this method seems to show that it works. But although the hash-table implementation throws no exceptions for valid input (in this or other methods), there seems to be a problem, which you suspect might be due to `remove`. What is wrong, and what test would reveal it?

> **Solution:** *When you remove an entry, you set the array slot that contained it to null, which cuts off any search for other members in the same bucket of the hash table. A possible test would be to insert two objects known to have the same hash code, remove the first, and then search for the second.*

4. [1 point] What is the source of the following lines?

La fleur que tu m'avais jetée,
Dans ma prison m'était restée,
Flétrie et sèche, cette fleur
Gardait toujours sa douce odeur....

Solution: *Bizet's opera Carmen.*

5. [2 points] The following fragment of a program compiles, but has what are almost certainly two bugs. Indicate clearly how to fix them.

```
public class Distributor {

    /** A new Distributor with N outlets. */
    public Distributor(int N) {
        // ERROR: local variable _outlets shadows the private instance variable.
        ArrayList<Outlet> _outlets = new ArrayList<Outlet>();
        // CORRECTED LINE: _outlets = new ArrayList<Outlet>();
        for (int i = 0; i < N; i += 1) {
            _outlets.add(new Outlet(i));
        }
    }

    /** Current number of outlets. */
    public int size() {
        return _outlets.size();
    }

    /** Turn off outlet #K. */
    public void close(int k) {
        _outlets.get(k).close();
    }

    /** Remove all closed outlets. */
    public void demolish() {
        for (Iterator<Outlet> it = _outlets.iterator(); it.hasNext(); ) {
            Outlet out = it.next();
            if (out.isClosed())
                // ERROR: attempt to modify _outlets while iterating through it.
                _outlets.remove(out);
                // CORRECTED LINE: it.remove();
        }
    }

    private ArrayList<Outlet> _outlets;
}
```

6. [5 points] *Lazy* data structures resemble their non-lazy (or *strict*) counterparts except that they compute the results of accessors (e.g., `.get` on a list) only when they are needed the first time, saving the result. Any subsequent calls to an accessor with the same arguments then simply returns the previously computed value. Thus, the data structure grows only in response to need. In CS61A, streams had this property.

The type `LazyTree`, below, is one such structure. A `LazyTree` is a binary tree node that has operations `getLabel`, `getLeft`, and `getRight`, which provide its label and its left and right children, as for an ordinary binary tree. To create a `LazyTree` node, one supplies a `TreeFormer` argument, which returns the node label and subtrees upon demand. However, a given `TreeFormer` is to be consulted at most three times during the lifetime of its tree node: once for the label, once for the left child, and once for the right. If the program asks a `LazyTree` for the label or a subtree from a node more than once, the second and subsequent requests return the previously computed value without reconsulting the `TreeFormer`.

```
/** Supplies the label (of generic type T) and tree formers for the
 *  left and right children of a tree node. */
public abstract class TreeFormer<T> {
    public abstract T label();
    public abstract LazyTree<T> left();
    public abstract LazyTree<T> right();
};

/** A binary tree whose contents are constructed as needed
 *  from TreeFormers.  The empty tree is represented by null. */
public class LazyTree<T> {
     public LazyTree(TreeFormer<T> former) {
         _former = former;
     }

     T getLabel() {
         if (!_haveLabel) { _haveLabel = true; _label = _former.label(); }
         return _label;
     }

     T getLeft() {
         if (!haveLeft) { _haveLeft = true; _left = _former.left(); }
         return _left;
     }

     T getRight() {
         if (!_haveRight) { _haveRight = true; _right = _former.right(); }
         return _right;
     }

     private TreeFormer<T> _former;
     private boolean _haveLeft, _haveRight, _haveLabel;
     private T _label;
     private LazyTree<T> _left, _right;
}
```

a. [3 points] Fill in the function below to obey its comment. Feel free to add any additional classes you need.

```
/** Returns a lazy tree that contains elements L through U of a sorted
 *  array A as a balanced binary search tree. For example, if A is
 *  {1, 2, 3, 4, 5, 6, 7, 8}, the tree resulting from toBST(A, 0, 7) is
 *              4
 *            /   \
 *          2       6
 *         / \     / \
 *        1   3   5   7
 *                     \
 *                      8
 *  As illustrated here, when the subtrees are uneven, the extra element goes
 *  on the right.
 */
LazyTree<String> toBST(String[] A, int L, int U) {
    if (L > U)
        return null;
    else
        return new LazyTree(new Balanced(A, L, U));
}

class Balanced extends TreeFormer<String> {
    Balanced(String[] A, int L, int U) {
      _A = A; _L = L; _U = U;
    }

    String label() {
        return _A[(_L+_U) / 2];
    }

    LazyTree<String> left() {
        if (_L > _U)
           return null;
        else
           return new Balanced(_A, _L, _U-1);
    }

    LazyTree<String> right() {
        if (_L > _U)
           return null;
        else
           return new Balanced(_A, _L+1, _U);
    }

    private String[] _A;
    private int _L, _U;
}
```

b. [2 points] Suppose I wanted to test another implementation of `LazyTree` by checking the properties that its nodes consult their `TreeFormer`s at most once to get their labels and children and do not consult their `TreeFormer`s for any labels or children that the client does not request (by calls to the `getLeft`, `getRight`, and `getLabel` methods). Describe in sufficient detail JUnit tests to check these properties.

```
There are many solutions.  One way is to define a
TreeFormer extension that keeps a static variable that counts the
total number of times the label, left, and right methods are called.
Then create a tree, partially traverse some of its nodes in some
irregular pattern, and check the value of the static variable.
```