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

**CS61B** **P. N. Hilfinger**
**Fall 2011**

# Test #2 Solutions

**1.** [3 points] Consider insertion sort, merge sort, and quicksort. For each algorithm, what is the worst-case asymptotic bound on the running time for a list of $N$ elements if you know additionally that:

- the input is already sorted?
- the input is reversely sorted?
- the input is a list containing $N$ copies of the same number?

| Condition | Insertion Sort | Merge Sort | Quicksort | Notes |
|---|---|---|---|---|
| Sorted | $\Theta(N)$ | $\Theta(N \lg N)$ | $\Theta(N \lg N)$ | *The quicksort figure assumes reasonable pivoting (say about the median of first, middle, and last values). Technically, without this provision, the answer $O(N^2)$ also works.* |
| Reverse sorted | $\Theta(N^2)$ | $\Theta(N \lg N)$ | $\Theta(N \lg N)$ | *See preceding note.* |
| $N$ copies | $\Theta(N)$ | $\Theta(N \lg N)$ | $\Theta(N^2)$ | *For quicksort, the pivot step will put almost all elements in one partition.* |

**2.** [3 points] We want a data structure, `T`, on which we can perform the following operations with the given complexity ($N$ being the number of elements already inserted in the data structure):

**insert(x, T)** insert a new element in $O(\lg N)$ time.

**contains(x, T)** check if the structure contains element $x$ in $O(\lg N)$ time.

**delete(k, T)** delete the $k^{\text{th}}$ smallest element in $O(\lg N)$ time, where we take $k$ to be a constant.

Out of all the data structures given below, which one(s) would you pick? For any you reject, explain briefly. Simply circle those that you accept; no explanation is necessary.

- a heap *No: containment check takes $O(N)$. Delete is tricky, but technically can be done with a large constant factor (order $2^k$).*
- a red-black tree *Yes.*
- a binary search tree *No: only works if data is such that tree remains balanced.*
- an unsorted linked list *No: contains and delete require $O(N)$.*
- a sorted linked list *No: contains and delete require $O(N)$ (because access to general element requires $O(N)$.)*
- an unsorted array *No: all operations require $O(N)$.*
- a sorted array *No: contains requires $O(\lg N)$, but insert and delete require $O(N)$.*
- a hash table (nothing said about the hash function) *No: without guarantees on the hash function, same as unsorted list.*

**3.** [2 points] In doing Project #2, Bill Bucket has written the following function:

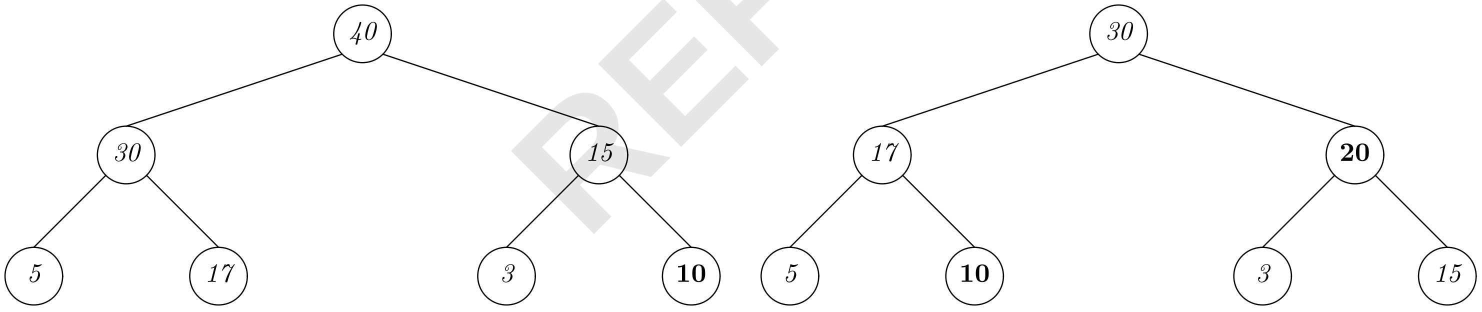
```
/** Returns a copy of A that is unaffected by subsequent operations on
 *  A. */
public static int[][] copy(int[][] A) {
    int[][] result = new int[A.length][];
    for (int i = 0; i < A.length; i += 1) {
        result[i] = A[i];
    }

    return result;
}
```

Devise a unit test that shows that this method does not work (does not fulfill its documentation).

```
@Test
public void checkCopy() {
    int[][] A = { { 1, 2 } };
    int[][] C = copy(A);
    A[0][0] = 0;
    assertArrayEquals(new int { 1, 2 }, C[0]);
}
```

**4.** [3 points] Starting from the max-heap on the left, we remove the largest element and insert 20. The resulting heap is shown on the right. Assume that all values are unique and that 20 is not in the original heap.



Fill in the blank nodes in both heaps to show one possible way that this result could come about.

**5.** [1 point] What non-mathematical property do the numbers 31, 45, 66, 99, and 500 have in common?

> **Solution:** *The numbers are also names of card games.*

**6.** [2 points] Show the steps taken by quicksort on the following unordered list, assuming that the pivot node is always the first item in the (sub)list being sorted, and the array is sorted in place.

```
36  22  15  56  48  90  72  06
```

To show a "step," show the contents of the array at the beginning of each call (initial or recursive) on the quicksort procedure. Assume that we sort all the way down to the trivial case of subarrays of size 1.

> **Solution:** *This solution uses the pivot method from the data-structure notes. The bars show the boundaries of the subsequences. We show the result of sorting all subsequences to save space.*

```
36   22   15   56   48   90   72   06      Initial
06   22   15 | 36 | 48   90   72   56
06 | 22   15 | 36 | 48 | 90   72   56
06 | 15 | 22 | 36 | 48 | 56   72 | 90
06 | 15 | 22 | 36 | 48 | 56 | 72 | 90      Final
```

**7.** [4 points] A certain binary search tree representation uses the following representation

```
/** A binary tree node.  The empty tree is represented by null. */
class BinNode {
      /** Returns my left child. */
      BinNode left() { ... }

      /** Returns my right child. */
      BinNode right() { ... }

      /** Returns my parent, or null if I am the root of the entire
       *  tree. */
      BinNode parent() { ... }

      /** Returns my key value (label). */
      ValueType label() { ... }
      ...
}
```

Fill in the following functions to conform to their comments.

```
/** True iff B is the left child of another node. */
static boolean amLeftChild(BinNode b) {

    return b.parent() != null && b.parent().left()==b;           ;
}

/** Returns the node in B's tree that contains the value next larger
 *  than B's. Assumes that the tree I am part of is a binary search
 *  tree and that all its values are unique.  Returns null if G is
 *  the node containing the largest value.  Does not call label(). */
static BinNode next(BinNode b) {
    if (b.right() == null) {
       while (b != null) {
           if (amLeftChild(b)) {
               return b.parent();
           }
           b = b.parent();
       }
    } else {
       b = b.right();
       while (b.left() != null) {
           b = b.left();
       }
    }
    return b;
}
```

**8.** [2 points] Consider a hash table that uses a very good hashing function—one that (miraculously) always distributes keys evenly across the bins in the table. Assuming the hash table expands at need to maintain a load factor of 3,

- What can you say about the worst-case time for adding an item to the table when it contains $N$ items to begin with?

  **Solution:** *Because the table may have to be resized,* $O(N)$.

- What can you say about the worst-case time for adding $N$ items to the table when it contains $N$ items to begin with?

  **Solution:** $\Theta(N)$. *Given the assumptions, insertion requires amortized constant time.*

**9.** [1 point] I create a list of integers by calling `semiRandomList(B, 10)`, using the following method:

```
static void semiRandomList(int[] A, int jiggle) {
    Random R = new Random();
    for (int i = 0; i < A.length; i += 1) {
        A[i] = i + R.nextInt(jiggle);
    }
}
```

(The `nextInt(jiggle)` call produces a random integer $\geq 0$ and <jiggle.) Having done this, I sort `B` using insertion sort. What can you say about the time required to do so? Be as precise as possible.

**Solution:** *An entry at position $k$ can only be out of order with respect to the entries at positions $i$ for which $k - 10 < i < k + 10$. This limits the total number of inversions, and therefore the running time of insertion sort, to roughly* $10N$.