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

**CS61B** **P. N. Hilfinger**
**Fall 2017**

# Test #2 Solutions

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 7 problems on 11 pages. Officially, it is worth 17 points (out of a total of 200).

This is an open-book test. You have 110 minutes to complete it. You may consult any books, notes, or other non-responsive objects available to you. You may use any program text supplied in lectures, problem sets, or solutions. Please write your answers in the spaces provided in the test. Make sure to put your name, login, and TA in the space provided below. Put your login and initials *clearly* on each page of this test and on any additional sheets of paper you use for your answers.

Be warned: my tests are known to cause panic. Fortunately, this reputation is entirely unjustified. Just read all the questions carefully to begin with, and first try to answer those parts about which you feel most confident. Do not be alarmed if some of the answers are obvious. Should you feel an attack of anxiety coming on, feel free to jump up and run around the outside of the building once or twice.

Your name: _____ Login: _____

Login of person to your Left: _____ Right: _____

Discussion TA: _____

*I pledge my honor that during this examination, I have neither given nor received assistance.*

Signature: _____

**1.** [3 points] The following questions concern complexity analysis. Unless otherwise stated, we are looking for asymptotic bounds ($O(\cdot)$, $\Omega(\cdot)$, or $\Theta(\cdot)$) and want the closest bounds you can find (If something is $O(N)$, don't say that it is $O(N^2)$, even though it would be. Don't use $O(\cdot)$ where $\Theta(\cdot)$ would apply.)

(a) Consider the following sorting method on an array:

```
public void sinkSort(int[] A) {
    boolean changed;
    changed = true;
    for (int k = A.length-1; k > 0 && changed; k -= 1) {
        changed = false;
        for (int j = 0; j < k; j += 1) {
            if (A[j] > A[j+1]) {  /* TEST */
                changed = true;
                int tmp = A[j]; A[j] = A[j+1]; A[j+1] = tmp;
            }
        }
    }
}
```

What is the worst-case cost, measured by number of times the line marked `/* TEST */` is executed, for an array of length $N$?

**Answer:** $\Theta(N^2)$

(b) What is the tightest lower bound that you can give for the time cost of executing `sinkSort(A)` for input of size $N$ (that is, what is the largest lower bound you can give that is true for all inputs of size $N$, not just the worst case)?

**Answer:** $\Omega(N)$

(c) Assuming that there are $\leq N$ inversions in the array `A`, what is the worst-case cost of `sinkSort(A)`?

**Answer:** $\Theta(N^2)$

*Problem continues on next page.*

(d) What is the worst-case time for execution of the following program as a function of `N`, assuming that calls to `g` take constant time?

```
int M;
M = 0;
for (int k = 1; k < N; k *= 2) {
    for (int j = 0; j < k; j += 1) {
        M = g(j, M);
    }
}
```

**Answer:** $\Theta(N)$

(e) Suppose that for every value $k$, $g(N,k) \in O(1)$ (that is, if $k$ is any constant, and $h(N) = g(N,k)$, then $h(N) \in O(1)$.) Exhibit a function $g$ for which this is true, and yet $p(N) = \sum_{1 \le k \le N} g(N,k) \in \Theta(N^2)$.

**Answer:** Let $g(N,k) =$ $k$. For fixed $k$ it is $\Theta(1)$, but $p(N)$ is clearly $N(N+1)/2$.

(f) Fill in the blanks to indicate worst-case asymptotic bounds for finding the seventh largest item by the fastest means in the following data structures, assuming each contains $N$ items:

i. A max-heap. **Answer:** $\Theta(\lg N)$

ii. A sorted array. **Answer:** $\Theta(1)$

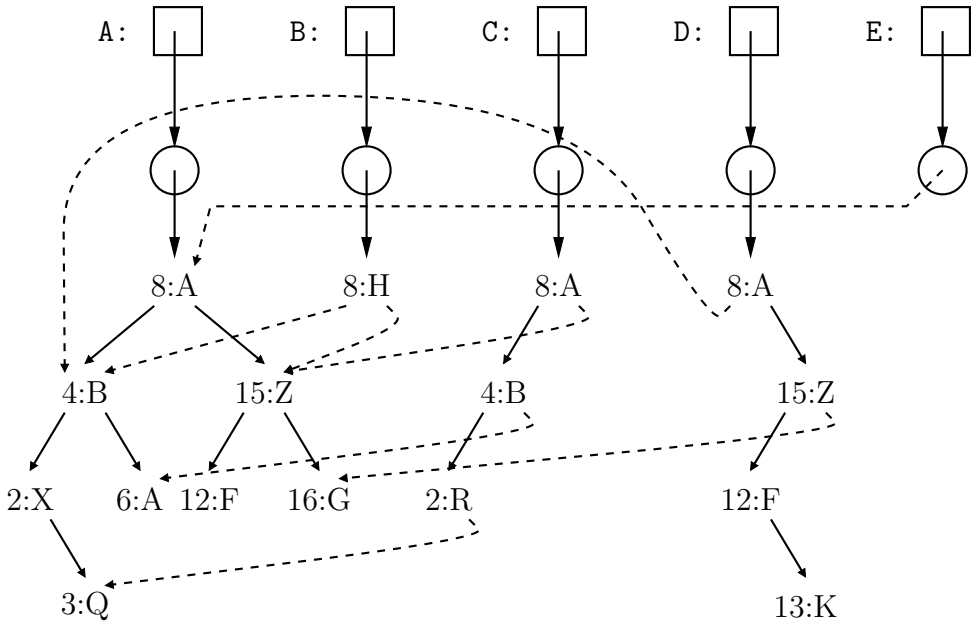
iii. An unsorted array. **Answer:** $\Theta(N)$

iv. A perfect hash table (assume a constant-time hashing function.)

**Answer:** $\Theta(N)$

*This page deliberately left blank.*

**2.** [4 points] In this question, we'll implement part of a binary-search structure called `IntTreeMap` that maps integer keys to strings. Fill in the definition on the next page to create a `IntTreeMap` class that shares as much tree structure as possible among copies of the tree. It creates new tree nodes (type `Node`) only when necessary to ensure that modifications to one `IntTreeMap` do not interfere with the behavior of other `IntTreeMaps`. For example, suppose that `A` is the `IntTreeMap` on the left below. The notation '$K : V$' denotes a `Node` object with a key $K$ and corresponding value $V$. Squares in the diagram are Java variables, and circles are `IntTreeMap` objects.




Suppose we now execute

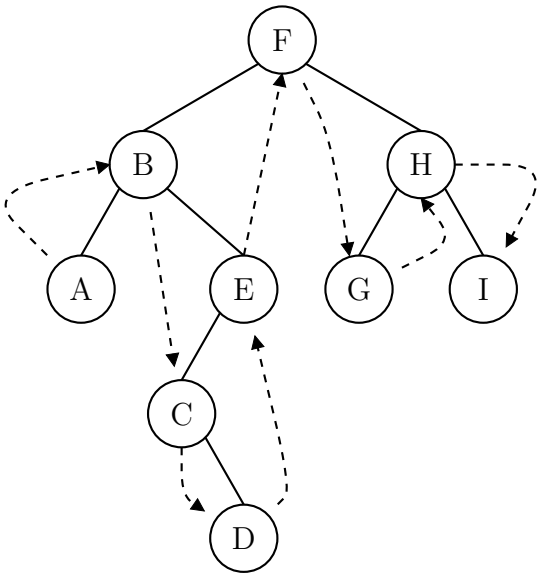
```
IntTreeMap B = new IntTreeMap(A), C = new IntTreeMap(A),
           D = new IntTreeMap(A), E = new IntTreeMap(A);
B.put(8, "H");
C.put(2, "R");
D.put(13, "K");
E.put(2, "X");
```

The intent is to end up with the structures shown for `B, C, D,` and `E`. In particular, since the insertion into `E` does not change the mapping of key 2, `E` and `A` can share all their nodes. Dashed lines denote pointers to previously existing structures shared with other `IntTreeMaps`. Fill in the method and constructor on the next pages to get this effect.

```
public class IntTreeMap {

    private Node root;

    private static class Node {
        Node(int key0, String val0, Node left0, Node right0) {
            key = key0; value = val0; left = left0; right = right0;
        }
        int key;
        String value;
        Node left, right;
    }

    public IntTreeMap() {
        root = null;
    }

    public IntTreeMap(IntTreeMap other) {
        root = other.root;
    }
```

```
    public void put(int key, String val) {
        root = insert(root, key, val);
    }

    private static Node insert(Node t, int key, String val) {
        if (t == null) {
            return new Node(key, val, null, null);
        } else if (key == t.key) {
            if (val.equals(t.value)) {
                return t;
            } else {
                return new Node(key, val, t.left, t.right);
            }
        } else if (key < t.key) {
            Node tmp = insert(t.left, key, val);
            if (tmp == t.left) {
                return t;
            } else {
                return new Node(t.key, val, tmp, t.right);
            }
        } else {
            Node tmp = insert(t.right, key, val);
            if (tmp == t.right) {
                return t;
            } else {
                return new Node(t.key, val, t.left, tmp);
            }
        }
    }

}
```

**3.** [4 points] A *threaded binary tree* is a tree in which each node contains a pointer to the next node of the tree in an in-order traversal, as illustrated by the dashed arrows here:




Here, the thread pointer is in addition to the tree's child pointers, as shown in the `TTree` class below. (There is actually a clever way to thread a tree that avoids having an extra pointer, but we're not using it in this problem.)

```
public class TTree {
    public TTree(String label, TTree left, TTree right) {
        this.label = label; this.left = left; this.right = right;
        this.thread = null;
    }
    public String label;
    public TTree left, right, thread;

    public static void inorder(TTree tree, Consumer<TTree> visitor) {
        if (tree != null) {
            inorder(tree.left, visitor);
            visitor.action(tree);
            inorder(tree.right, visitor);
        }
    }
}
```

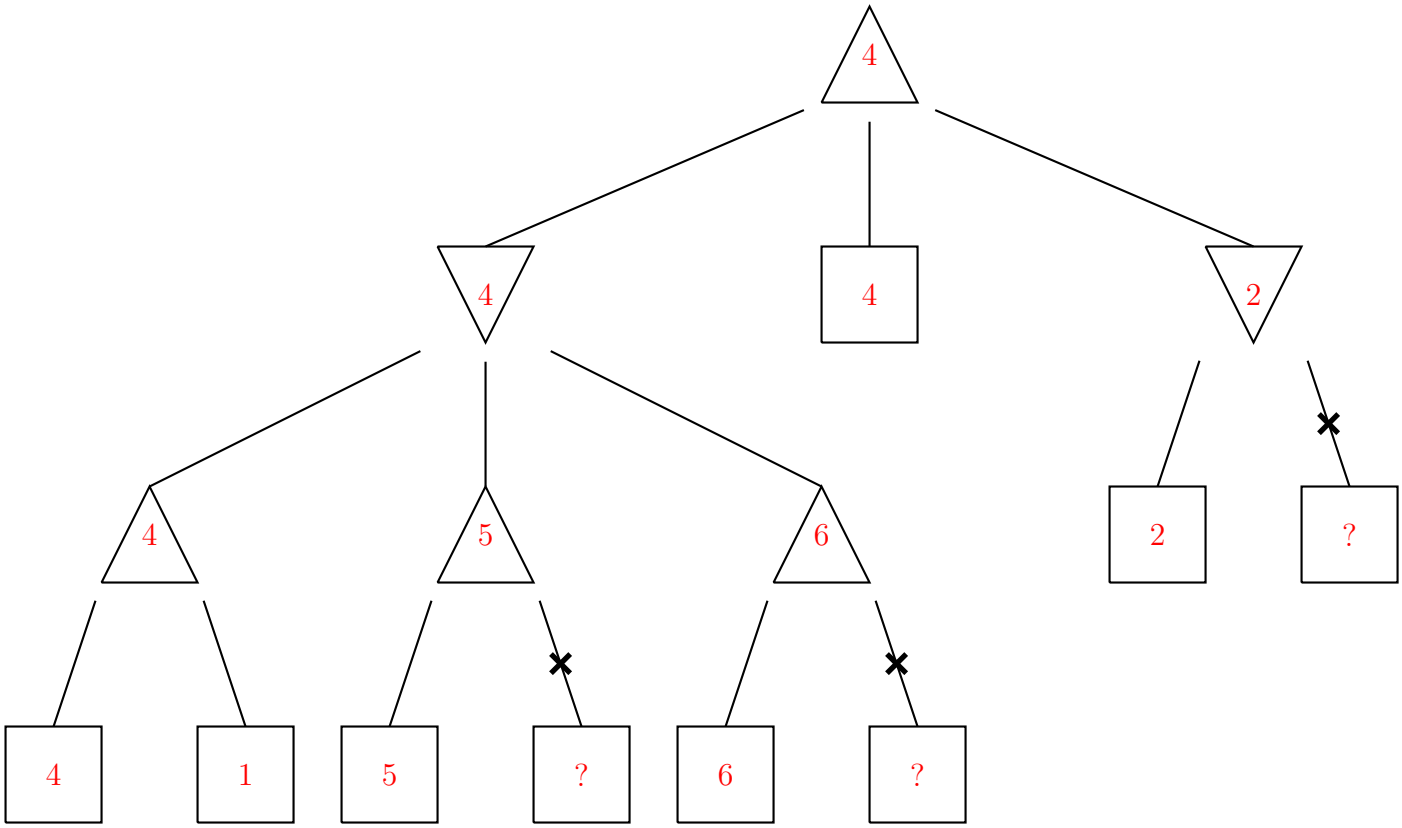
There is also the following utility class:

```
public interface Consumer<T> {
    void action(T x);
}
```

We add the following `threadTree` method to the class `TTree`. Fill it in and add any other declarations needed in the class to fulfill its comment. Your solution must not use any iteration or recursion other than that provided in the other, previously implemented methods of `TTree`.

```
/** Properly thread the nodes of TREE, as described in the problem,
 *  returning a pointer to the first node in in-order. */
public TTree threadTree(TTree tree) {
    Threader visitor = new Threader();
    TTree.inorder(tree, visitor);
    return visitor.start();
}

private static class Threader implements Consumer<TTree> {

    private TTree first, last;

    public void action(TTree node) {
        if (first == null) {
            first = last = node;
        } else {
            last.thread = node;
            last = node;
        }
        node.thread = null;
    }

    public TTree start() {
        return first;
    }

}
```

**4.** [1 point] In the partial game tree below, maximizing nodes are denoted by △; minimizing nodes by ▽; and □ nodes represent positions with static values (as you can see, for this tree not all statically evaluated positions are at the same level.) Crossed-out edges indicate branches that are pruned by the alpha-beta algorithm, and therefore have no effect on the root value. Fill in values for the unpruned nodes (△, ▽, and □) that result in exactly the indicated subtrees being pruned, and no others (you need not fill in values for any pruned subtrees). Use only values in the range 0–9, inclusive (values may be used as often as desired.)



**5.** [1 point] Fill in the definition of `patn` with a Java pattern string that matches valid Java arithmetic expressions containing just identifiers, additions, multiplications, and divisions (no numerals, whitespace, parentheses, or other operators), and bracketed by `<` and `>`. Assume that Java identifiers consist only of letters. The pattern should capture the expression as its first capturing group, not including the `<>`.

```
static String patn = "<([a-zA-Z]+([*+/][a-zA-Z]+)*>                    ";
static void printFirstExpr(String S) {
    Matcher m = Pattern.compile(patn).matcher(S);
    if (m.matches()) {
       System.out.println(m.group(1));
    }
}
```

For example if `S` is

```
"This only looks like an expression: x*y/z+q, but this is one: <x/z*foo+bar>.
```

then `printFirstExpr(S)` should print

```
x/z*foo+bar
```

**6.** [1 point] Who was the only U.S. Secretary of Commerce to become President?

Herbert Hoover

**7.** [2 points] Fill in the program below to fulfill its comment. Do not use any arithmetic operators (`+`, `-`, `*`, `/`), function calls, or conditional expressions (`?:`).

```
/** The integer value consisting of every Kth bit of X, starting with
 *  bit 0, packed adjacent to each other. In other words,
 *  bit j of the result is bit j * K of X.  Here, bit 0 is the
 *  least-significant (units) bit, bit 1 is the 2's bit, etc., and
 *  bits 32 and beyond are 0.
 *
 *  For example, pack(83, 2) == 13, because 83 in binary is 1010011,
 *  and taking bits 0, 2, 4, and 6 gives 1101, which is 13 in decimal.
 *  Also, pack(-1, 31) == 3, because -1 is represented as 32 1-bits in
 *  binary, so bit 0 and bit 31 are both 1, giving 11 in binary or 3
 *  in decimal. */
static int pack(int x, int k) {
    int b, r;
    r = 0;
    b = 1;
    while (x != 0) {
        if ((x & 1) == 1) {
            r = r | b;
        }
        b = b << 1;
        x = x >>> k;
    }
    return r;
}
```

**8.** [2 points] Here are some questions concerning hashing.

(a) Consider the implementation of a hash table for a class `Pixel`, defined

```
public class Pixel {
    private int _row, _col;
    /** Keep track of row and column of last Pixel created. */
    private static int _lastRow, _lastCol;

    public Pixel(byte row, byte col) {
        if (row < 0 || col < 0) {
           throw new IllegalArgumentException();
        }
        _lastRow = _row = row; _lastCol = _col = col;
    }

    @Override
    public boolean equals(Object other) {
        Pixed p = (Pixel) other;
        return _row == p._row && _col == p._col;
    }

    @Override
    public int hashCode() {

        return See options below ;
    }

    // Other methods not shown.
}
```

For each of the following possible replacements for the blank, indicate (by completely filling in the appropriate boxes) whether the resulting hash function is valid (i.e., makes the library classes `HashSet` and `HashMap` work properly,) and, if so, whether the resulting function can always serve as a perfect hashing function (at least for a sufficiently large table).

| Statement | Valid | Invalid | Perfect |
|---|---|---|---|
| (i) `return _lastRow ^ _lastCol` | ☐ | ☒ | ☐ |
| (ii) `return _row ^ _col` | ☒ | ☐ | ☐ |
| (iii) `return _row * _col` | ☒ | ☐ | ☐ |
| (iv) `return (_row << 7) + _col` | ☒ | ☐ | ☒ |

(b) Suppose that a certain hash table that uses open addressing contains $N$ items and that its hash function determined a different bucket number for each of those existing $N$ values (it's won't necessarily do that for other values than those in the table). Suppose furthermore that the table's current load factor is 1/2. What is the worst-case bound for the number of equality comparisons required to insert a new value that is not currently in the table, assuming the table does not have to be resized? (We did present open-address hashing in lectures, so please don't ask us about it now.)

**Bound:** $\Theta($N_____$)$

(c) Under the same assumptions as part (b), what bound can you place on the worst-case time to insert a value that is already in the table (which would not change the contents of the table)?

**Bound:** $\Theta($1_____$)$

(d) Again under the same assumptions as part (b), suppose that the implementer decides to resize the table as soon as the load factor exceeds 1/2. After inserting one more node into the table, what is the worst-case bound for the next search in that table?

**Bound:** $\Theta($N_____$)$