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

CS61B **P. N. Hilfinger**
Fall 2018

Test #2

READ THIS PAGE FIRST. *Please do not discuss this exam with people who haven't taken it.* Your exam should contain 8 problems on 13 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: __________

Your SID: ____________________

Login of person to your Left: ______________ Right: ______________

Discussion TA: ________________________________

1. [2 points] In this question, we will assess a variety of hashing functions for the following class:

```
class Room {

    /** E.g., new Room("Cory", "521", 52, true); */
    Room(String building, String roomNum, int capacity, boolean reserve) {
        _building = building;
        _roomNum = roomNum;
        _capacity = capacity;
        _reservable = reserve;
    }

    @Override
    public int hashCode() { HASHING FUNCTION HERE }

    @Override
    public boolean equals(Object other) {
        Room o = (Room) other;
        return _building.equals(o._building) && _roomNum.equals(o._roomNum)
               && _capacity == o._capacity && _reservable == o._reservable;
    }

    // Other methods not shown.

    private String _building;
    private String _roomNum;
    private int _capacity;
    private boolean _reservable;
}
```

Assume that the `hashCode` functions for the `String` class is uniformly distributed and deterministic on the data we deal with.

We are given a `HashMap<Room, Integer>`, `map`, which we'll assume uses external chaining and keeps its load factor below some constant value. For each of the following `hashCode` functions for `Room`, fill in the appropriate bubble (1, 2, or 3) that describes what happens on a call to `map.get`.

1. `map.get(x)` will always or sometimes return the wrong value.
2. `map.get(x)` will always return the correct value, but is likely to take significantly more than constant time.
3. `map.get(x)` will always return the correct value and is likely to run in constant time.

Assume that the `map` can get arbitrarily large, as can the number of rooms in any building.

(a) `return building.hashCode();`

Description: 1. ○ [✓ 2.] 3. ○

(b) `return building.hashCode() + _roomNum.hashCode();`

Description: 1. ○ 2. ○ [✓ 3.]

(c) `return building.hashCode() + _roomNum.hashCode() + _capacity;`

Description: 1. ○ 2. ○ [✓ 3.]

(d)
```
return building.hashCode() + _roomNum.hashCode()
       + (int) System.currentTimeMillis();
```

Description: [✓ 1.] 2. ○ 3. ○

(e)
```
int boolHash = _reservable ? 1 : 0; // 1 if _reservable else 0
return (building.hashCode() + _roomNum.hashCode()) * boolHash;
```

Description: 1. ○ [✓ 2.] 3. ○

(f) `return (building.hashCode() + _roomNum.hashCode()) >>> _capacity;`

Description: 1. ○ [✓ 2.] 3. ○

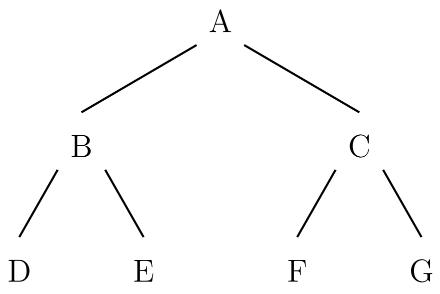
(g) Suppose we introduce the following bug into the `Room.equals` method:

```
public boolean equals(Object other) {
    return true;
}
```

Compared to the previous implementation of the `.equals` method, when using this new implementation, a call to the `get` method will (fill in the bubble for the correct answer):

- ○ Always return the correct value
- [✓ Sometimes return the correct value]
- ○ Never return the correct value

2. [2 points] We are given the following min-heap, where each letter represents a different number.



Given this initial min-heap, assume we insert a new value, H.

(a) Which letter(s) could represent the smallest number in the resulting min-heap? (Fill in the squares for all that apply.)

A ■ B □ C □ D □ E □ F □ G □ H ■

(b) Which letter(s) could represent the last item (rightmost on the bottom row) in the resulting min-heap?

A □ B □ C □ D ■ E □ F □ G □ H ■

(c) Which letter(s) could represent the largest item in the resulting min-heap?

A □ B □ C □ D ■ E ■ F ■ G ■ H ■

Now return to the original heap (just A–G; no H inserted). After removing the minimum value from the original heap:

(d) Which letter(s) could represent the smallest in the resulting min-heap?

A □ B ■ C ■ D □ E □ F □ G □

(e) Which letter(s) could represent the second-smallest number in the resulting min-heap?

A □ B ■ C ■ D ■ E ■ F ■ G ■

(f) Which letter(s) could be the right child of the root in the resulting min-heap?

A □ B □ C ■ D □ E □ F ■ G ■

3. [5 points] The binary search tree class `BSTSet` on page 6 contains an incomplete implementation of a set type represented as a binary search tree whose labels are strings. In this problem, you will be completing its representation and several of its instance methods.

(a) If we add no additional instance variables to the representation, what is the asymptotic cost of executing the `size` method as a function of N, the number of items in the tree?

$\Theta(\lg N)$ ○ $\Theta(\sqrt{N})$ ○ [✓ $\Theta(N)$] $\Theta(N^2)$ ○ $\Theta(2^N)$ ○

(b) If we restrict our trees to ones that are as bushy as possible, so that they have as few levels as possible, how does the answer to part (a) change (if at all)?

$\Theta(\lg N)$ ○ $\Theta(\sqrt{N})$ ○ [✓ $\Theta(N)$] $\Theta(N^2)$ ○ $\Theta(2^N)$ ○

Problem continues on page 7

```
public class BSTSet {

    /** The empty BSTSet (all empty BSTSets are the same object). */
    public static final BSTSet EMPTY = new EmptyBST();

    /** Destructively add ITEM to this set, returning the modified
     *  set. Has no effect (and returns this) if ITEM is already
     *  contained in the set. */
    public BSTSet add(String item) {  /* Implemented in part (c) */ }
    /** Return the number of items in this set.  */
    public int size() { /* implemented in parts (c) */ }
    /** Return the Kth smallest item in this set. Item 0 is the
     *  smallest item, item 1 is the next smallest, etc. */
    public String get(int k) { /* implemented in part (d) */ }

    /** A new BSTSet containing only the item ITEM. */
    private BSTSet(String item) {
        _left = _right = EMPTY;
        _item = item;
    }

    // There may be other methods that are not shown.

    private static class EmptyBST extends BSTSet {
        @Override
        public BSTSet add(String x) { /* implemented in part (c) */ }

        @Override
        public int size() { /* implemented in part (c) */ }

        @Override
        public String get(int k)
        { throw new java.util.NoSuchElementException(); }

        private EmptyBST() { super(null); }
    }

    private String _item;
    private BSTSet _left, _right;
}
```

(c) Implement the `add` and `size` methods in `BSTSet` so that `size` is a constant time operation and the `add` method executes in $O(h)$ time, where h is the height of the tree. (The implementation of `EmptyBST` is in part (d)). Add any necessary instance variables to `BSTSet`.

```
public class BSTSet {
    ...
    /** Destructively add ITEM to this set, returning the
     *  modified set. Has no effect (and returns this)
     *  if ITEM is already contained in the set. */
    public BSTSet add(String item) {
        int c = item.compareTo(_item);
        if (c < 0) {
            _left = _left.add(item);
        } else if (c > 0) {
            _right = _right.add(item);
        }
        _size = _left.size() + _right.size() + 1;
        return this;
    }

    /** Return the number of items in this set.  */
    public int size() {
        return _size;
    }

    private int _size = 1;
    ...
```

(d) Implement the `add` and `size` methods in `EmptyBST`.

```
          private static class EmptyBST extends BSTSet {

               public BSTSet add(String item) {
                    return new BSTSet(item);
               }

               public int size() {
                    return 0;
               }
          ...
     }
```

(e) Implement the `get` method in `BSTSet` so that it executes in time $O(h)$, where h is the height of the tree.

```
     public class BSTSet {
          ...
          public String get(int k) {
               if (k < _left.size()) {
                    return _left.get(k);
               } else if (k == _left.size()) {
                    return _item;
               } else {
                    return _right.get(k - _left.size() - 1);
               }
          }
          ...
     }
```

4. [2 points] Fill in the following function to agree with its comment. For example, `extract2(0xdeadbeef, 3, 1)` should return `0xdebe` and `extract2(0xdeadbeef, 1, 3)` should return `0xbede`. (`0x` indicates hexadecimal—base 16—in which the possible digits are 0–9 and a–f for a total of 16 possible digits. Each hexadecimal digit therefore stands for 4 bits.)

```
/** Return (as an int) the non-negative 16-bit value whose
 *  low-order byte is byte LOW of VAL and whose high-order
 *  byte is byte HIGH of VAL. Assume 0 <= LOW < 4, 0 <= HIGH < 4.
 *  Byte 0 is the low-order (least significant) byte, and byte 3
 *  is the high-order byte. */
static int extract2(int val, int high, int low) {

    /* You may OPTIONALLY use the following two lines to define
     * intermediate values. */


    int v1 = (val >> (low * 8)) & 255;


    int v2 = (val >> (high * 8)) & 255;


    return v1 | (v2 << 8);

}
```

5. [1 point] Starting from the north pole, you consult your (magnetic) compass and start along the great circle route in the direction it initially says is north. You continue in the same direction for a distance of 4200 miles. In what country do you end up?

Mexico

6. [2 points] For each of the following methods, write down the tightest asymptotic bounds you can for the actual (not just worst-case) asymptotic running time as a function of $n \geq 0$ ("tightest" means smallest for $O(\cdot)$ and largest for $\Omega(\cdot)$). Provide either a single $\Theta(\cdot)$ bound, if applicable, and otherwise a $O(\cdot)$ and $\Omega(\cdot)$ bound.

(a)
```
public void flip(int n) {
    if (n < 1) {
        return;
    }
    flip(n/2);
}
```
Bound(s): $\lg n$

(b)
```
public void flop(int n) {
    if (n < -1000) {
        return;
    }
    flop(n - 1);
}
```
Bound(s): n

(c)
```
public void drop(int n) {

    for (int i = 0; i < Math.max(61.0 * 61.5, Math.PI * 1000.0); i += 1) {
        System.out.println(n * n);
    }
}
```
Bound(s): 1

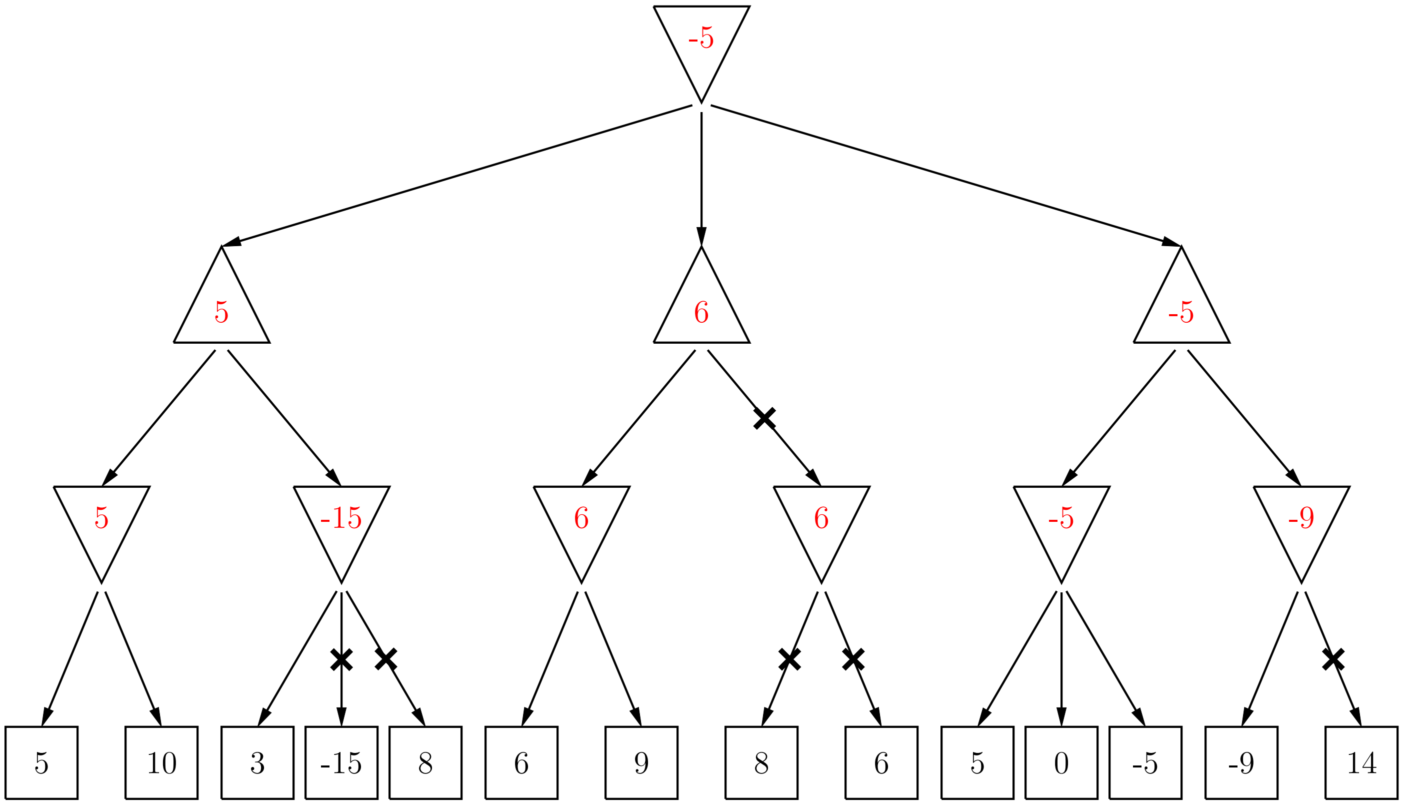
(d)
```
public void blob(int n) {
    if (n < 1) {
        return;
    }
    blob(n/2);
    blob(n/2);
    blob(n/2);
    blob(n/2);
}
```
Bound(s): n^2

(e)
```
public void coffee(int n) {
    for (int i = 0; i < n; i++) {
        while (i < n) {
            if (Math.random() > 0.5) {
                System.out.print("single");
            } else if (Math.random() < 0.5) {
                System.out.print("double");
                System.out.print("double");
            }
            n = n >> 1;
        }
    }
}
```
Bound(s): $\Theta(\lg n)$

(f)
```
public void pancake(Stack<Boolean> s) {
    int n = s.size();
    while (!s.empty()) {
        if (s.pop()) {  // Removes and returns top, which is true/false
            pancake(s);
        } else {
            for (int i = 0; i < n; i++) System.out.println(i);
        }
    }
}
```
Bound(s): $O(n^2)$, $\Omega(n)$

(g)
```
public void whippedCream(int n, int m) {
    if (n == 2) {
        return;
    } else if (n == 1) {
        for (int i = 0; i < m; i++) {
            System.out.println(i);
        }
    } else {
        whippedCream(n - 1, m);
    }
}
```
Bound(s): $\Theta(n)$

7. [2 points] In the partial game tree below, we represent maximizing nodes as △; minimizing nodes as ▽; and nodes with static values as □. Determine the values for the nodes that would be determined by the minimax algorithm without pruning (write them inside the nodes), and then cross out branches that would not be traversed (would be pruned) as a result of alpha-beta pruning. Assume we evaluate children of a node from left to right. (Careful! The root of this tree is a minimizing node.)



8. [2 points] In the following questions, assume that k is a fixed positive integer constant. Also assume that in each case, A is an array or linked list that is initially sorted in ascending order. The value n in each case refers to the initial size of A. When asked for bounds, give the tightest, simplest asymptotic bounds you can given the information available (i.e., single $\Theta(\cdot)$ bounds where feasible, or else the largest $\Omega(\cdot)$ and smallest $O(\cdot)$ bounds possible.) Assume that all elements originally in and added to A are distinct. When the initial data are stored in an array, assume that they completely fill the array (with no extra space at the end). References to insertion sort, merge sort, and heap sort refer to the versions of these algorithms presented in lecture. Assume that in applying any of the sorting algorithms, you do not take advantage of your knowledge of how the input is constructed to specialize (and speed up) the algorithm.

(a) What bound(s) can you put on the time required to destructively add k elements to the end of A and perform insertion sort on the result as a function of n, assuming that the A is represented as a doubly linked list?

Answer: $\Theta(n)$

(b) What bound(s) can you put on the time required to destructively add k elements to the end of A and perform insertion sort on the result as a function of n, assuming that the A is represented as an array?

Answer: $\Theta(n)$

(c) What bound(s) can you put on the time to destructively add k elements to the end of A and perform heap sort on the result as a function of n, assuming that the A is represented as an array?

Answer: $\Theta(n \lg n)$

(d) What bound(s) can you put on the time to destructively add k elements to end of A and perform merge sort on the result as a function of n, assuming that the A is represented as an array?

Answer: $\Theta(n \lg n)$

(e) What bound(s) can you put on the time to destructively add k elements to the end of A and perform merge sort on the result as a function of n, assuming that the A is represented as a doubly linked list?

Answer: $\Theta(n \lg n)$