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

CS61B
Fall 2016

P. N. Hilfinger

Test #2 Solution

Reference Material.

```
/** Binary tree. */
public class BinTree<Label> {
    /** The label of this node. */
    public final Label label;
    /** My left and right children. */
    public BinTree<Label> left, right;

    public BinTree(Label label, BinTree<Label> left, BinTree<Label> right) {
        this.label = label;
        this.left = left;
        this.right = right;
    }
}
```

1. [2 points]

Give $\Theta(\cdot)$ bounds for the best-case and worst-case running times of the following calls. Where there are multiple parameters to the call, you may be asked to come up with some expression, E, involving the parameters, as well as bounds on the value of E. For example, given a method call with arguments `X` and `Y`, you might choose $E = X + Y$ and say that the worst-case running time is in $\Theta(E^2)$ and the best case is in $\Theta(E)$.

Assume that `print` and `println` are constant-time operations. For Java library classes, determine time bounds based on your knowledge of the data structures that underlie them.

(a) What is the complexity of `eval(a, b)`? Specify a suitable expression for E involving a and b and bounds on that expression. Assume `a <= b`.

```
public static void eval(int N, int M) {
    for (int i = N; i < M; i += 1) {
        System.out.println(i + " dabs.");
    }
}
```

Best-case bound: $\Theta($*E*$)$ Worst-case bound: $\Theta($*E*$)$

where $E =$ *b* − *a*.

(b) For the same program as in part (a), what is the complexity of `eval(N, 100000000)` as a function of N, assuming that $N \geq 0$?

Best-case bound: $\Theta($1$)$ Worst-case bound: $\Theta($1$)$

(c) What is the complexity of `eval(L, v)` as a function of $N =$ `L.size()`?

```
public static void eval(LinkedList<Integer> list, int value) {
    for (int x : list) {
        if (x == value) {
            for (int j = 0; j < list.size(); j += 1) {
                System.out.println(list.get(j));
            }
        }
    }
}
```

Best-case bound: $\Theta($*N*$)$ Worst-case bound: $\Theta($*N*^{3}$)$

(d) What is the complexity of `eval(A, K)`? Specify a suitable expression for E involving `K` and `A.length` and bounds on that expression. Assume `K>=0`.

```
public static void eval(int[] array, int M) {
    if (M > 0) {
        eval(array, M / 2);
    }
    for (int i = 0; i < array.length; i += 1) {
        System.out.println(M + array[i]);
    }
}
```

Best-case bound: $\Theta($*E*$)$ Worst-case bound: $\Theta($*E*$)$

where $E =$ `A.length`$\cdot \lg K$

2. [2 points] For each of the following, give the tightest (smallest) and simplest (shortest) bounds you can. Use $\Theta(\cdot)$ if possible, and otherwise $O(\cdot)$.

(a) $n^2 + 1/n$ Bound: $\Theta(n^2)$

(b) $n \cdot \sin(n)$ Bound: $O(n)$

(c) $\ln(x) + \ln(1/x)$ Bound: $\Theta(1)$

(d) $4^n + n2^n$ Bound: $\Theta(4^n)$

3. [2 points] Consider a hash table that uses external chaining and also keeps track of the number of keys that it contains. It stores each key at most once; adding a key a second time has no effect. It takes the steps necessary to ensure that the number of keys is always less than or equal to twice the number of buckets (i.e., that the load factor is ≤ 2). Assume that its hash function and comparison of keys take constant time. All bounds should be a function of N, the number of elements in the table.

(a) Give $\Theta(\cdot)$ bounds on the worst-case times of adding an element to the table when the load factor is 1 and when it is exactly 2 before the addition.

Bound for load factor 1: $\Theta(N)$. Bound for load factor 2: $\Theta(N)$,

(b) Assume that the hashing function is so good that it always evenly distributes keys among buckets. What now are the bounds on the worst-case time of adding an element?

Bound for load factor 1: $\Theta(1)$. Bound for load factor 2: $\Theta(N)$,

(c) Making no assumption about the goodness of the hashing function, suppose that instead of using linked lists for the buckets, we use some kind of binary search tree that somehow keeps itself "bushy." What bound can you place on the worst-case time for testing to see if an item is in the table?

Bound: $\Theta(\lg N)$

(d) Using the same representation as in part (c), but with a very good hash function, as in part (b), what bound can you place on the worst-case time for testing to see if an item is in the table?

Bound: $\Theta(1)$

4. [2 points] Consider a heap of distinct integers implemented using an array, as discussed in lecture:

```
public class MaxHeap {

      public MaxHeap(int maxSize) {
          _data = new int[maxSize];
          _size = 0;
      }

      /** Assuming that all elements of _data[0.._size-1] are distinct
       *  and satisfy the heap property, except that element #K
       *  (0 <= K < _size) may be out of order with respect to its ancestors,
       *  re-arrange _data to make the heap property apply to all
       *  elements in _data[0 .. _size-1]. */
      private void reheapifyUp(int k) { /* Implementation not shown */ }

      /** Assuming that all elements of _data[0.._size-1] are distinct
       *  and satisfy the heap property, except that element #K
       *  (0 <= K < _size) may be out of order with respect to its
       *  descendants, re-arrange _data to make the heap property apply
       *  to all elements in _data[0 .. _size-1]. */
      private void reheapifyDown(int k) { /* Implementation not shown */ }

      // reheapify and modify methods are on the next page.
      // add, removeLargest, getLargest, and other methods not shown.
}
```

(a) Implement `reheapify` on the next page.

(b) Implement `modify` on the next page.

(c) Worst-case running bound for `reheapify`, where N = `_size`: $\Theta(\lg N)$ _____.

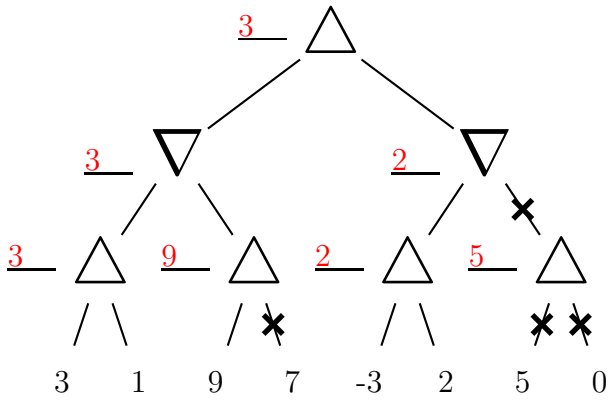
(d) Worst-case running bound for `modify`, where N = `_size`: $\Theta(N)$ _____.

In your solutions to (a) and (b), do not use extra space or pack multiple statements on a line.

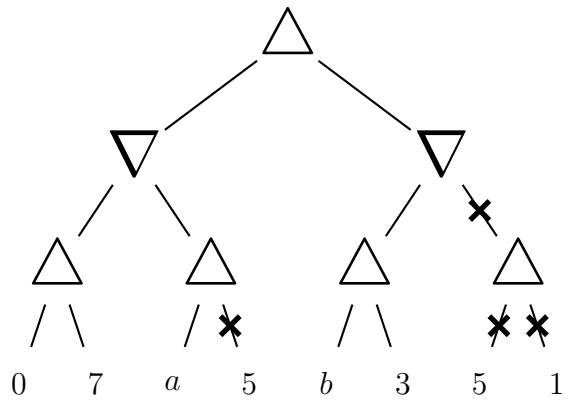
```
/** Assuming that all elements of _data[0.._size-1] satisfy the
 *  heap property, ignoring item K (0 <= K < _size), re-arrange
 *  _data[0 .. _size-1] to make the heap property apply to all
 *  elements in _data[0 .. _size-1].
private void reheapify(int k) {  // Part (a): Fill in
    reheapifyUp(k);
    reheapifyDown(k);
}

/** Remove value OLDVAL from this heap and replace it with NEWVAL,
 *  maintaining the heap property.  Has no effect if OLDVAL is not
 *  present. */
public void modify(int oldVal, int newVal) { // Part (b): Fill in
    for (int i = 0; i < _size; i += 1) {
        if (_data[i] == oldVal) {
            _data[i] = newVal;
            reheapify(i);
            break;
        }
    }
}
```

5. [2 points]

(a) In the blanks to the left of the game-tree nodes below, put the position values as determined by minimax, assuming the values at the bottom are as shown. Cross out the branches that would be pruned in an alpha-beta search. Maximizing nodes are denoted by △; minimizing nodes by ▽.

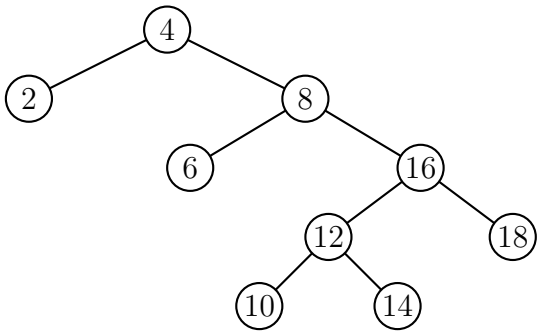


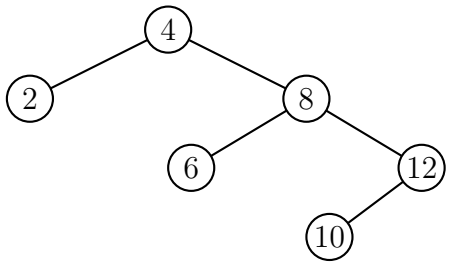
(b) Give a possible value for a and a value for b such that the indicated branches would be pruned in an alpha-beta search. If no possible values for a or b exist, state "not possible."



a value: 8 b value: 6

6. [2 points] Fill in the function below to obey its comment. Assume that `BinTree` is as defined on page 2. For example, if `A` is the `BinTree<Integer>` on the left below, then `trimHigh(A, 13)` produces the tree on the right. Your implementation must *not* create any new nodes. Also, the `label` field is marked `final`; therefore you must not assign to this field.






```
/** Destructively modify binary search tree BST, removing all keys
 *  that are greater than LIMIT.  Returns the modified tree. */

public static BinTree<Integer> trimHigh(BinTree<Integer> bst, int limit) {

       if (bst == null) {
            return null;
       } else if (bst.label > limit) {
            return trimtree(bst.left, limit);
       } else {
            bst.right = trimtree(bst.right, limit);
            return bst;
       }
}
```

7. [2 points]

(a) Give a single expression that produces the same value as `((x/512) % 64) + 384` for integral $x \geq 0$, but does not use `+`, `-`, `*`, `/`, or `%`.

`((x/512) % 64) + 384 ==` `((x >>> 9) & 63) | 384`

(b) Describe succinctly what the number printed by the following program is a count of. Your answer should involve properties of `x` and `y` only (not `z`), and should not make any mention of the bitwise operations. Assume that all variables are of type `int`.

```
z = x ^ y;
n = 0;
for (int i = 0; i < 32; i += 1) {
    n += 1 & z;
    z >>>= 1;
}
System.out.println(n);
```

The number of bits at which `x` and `y` disagree.

8. [1 point] Which name among the following does not belong, and why?

Alice Cary, Robert Browning, William Cullen Bryant, Samuel Coleridge-Taylor, Paul Laurence Dunbar, Eugene Field, Edgar A. Guest, Langston Hughes

9. [1 point] Fill in the definition of `patn` with a Java pattern string such that the following method prints the first sequence in `S` of one or more decimal integer numerals separated by commas, or nothing if no such sequence exists. You may assume there are no blanks embedded in the sequences.

```
static String patn = ".*?((\\d+,)*\\d+).*?";
static void printNumberList(String S) {
    Matcher m = Pattern.compile(patn).matcher(S);
    if (m.matches()) {
       System.out.println(m.group(1));
    }
}
```

For example if `S` is

```
   "The lottery numbers are 10,7,9,31,22, and yesterday's were 19,5,29,48,30."
```

then `printNumberList(S)` should print

```
10,7,9,31,22
```

10. [2 points] Consider a program for playing a two-person board game, where the board is represented by the following `Board` class:

```
public class Board implements Comparable<Board> {
    /** Return the Boards that result from taking each of the possible
     *  legal moves from this position. Always non-empty unless gameOver(). */
    public Collection<Board> children() { ... }
    /** Return true iff this is won or drawn position. */
    public boolean gameOver() { ... }
    @Override
    public int compareTo(Board b) { ... }
}
```

That is, a `Board` is a kind of tree whose children are possible next positions in the game. `Board` objects are comparable; if `A` and `B` are `Boards`, then `A` is larger than `B` iff the position represented by `A` is more favorable to the first player (whom we'll call "white.") Just exactly how `Board` determines this ordering is irrelevant to this problem (you do not have to implement minimax search.)

We want an iterator over `Boards` that produces a sequence of `Boards` that would result from optimal play, starting with white's move. That is, a `new BoardIterator(b0)` (where `b0` must not be null) would first return `b0`, then the largest child, `b1`, of `b0`, then the smallest child of `b1`, etc. Complete the following to have this behavior.

There are two possible solutions to this problem. One maintains the invariant that you have already returned the board you are keeping track of (except perhaps before the first call to next()), the other maintains the invariant that you haven't yet returned the board you are keeping track of. The former requires an additional if-statement in next, the latter doesn't. hasNext() differs for the two solutions.

Solution 1

```
/* These might come in handy. Both take a single argument,
 * a Collection, and return the object in the collection that is
 * last (max) or first (min) in natural order. */
import static java.util.Collections.max;
import static java.util.Collections.min;

public class BoardIterator<Board> implements Iterator<Board> {

    private Board _current;
    private int _maximize;

    public BoardIterator(Board startingBoard) {
        _current = startingBoard;
        _maximize = -1;
```

```
    }
    public boolean hasNext() {
        return !_current.gameOver();
    }
    /** Assumes hasNext(). */
    public Board next() {
        if (_maximize == -1) {
            _maximize == 1
            return _current;
        } else if (_maximize == 0) {
           _current = max(_current.children());
           _maximize = 1;
        } else {
           _current = min(_current.children());
           _maximize = 0;
        }
        return _current;
    }
}
```

Solution 2

```
/* These might come in handy. Both take a single argument,
 * a Collection, and return the object in the collection that is
 * last (max) or first (min) in natural order. */
import static java.util.Collections.max;
import static java.util.Collections.min;

public class BoardIterator<Board> implements Iterator<Board> {

    private Board _current;
    private boolean _maximize;

    public BoardIterator(Board startingBoard) {
        _current = startingBoard;
        _maximize = true;
    }
```

```
    public boolean hasNext() {
        return _current != null;
    }
    /** Assumes hasNext(). */
    public Board next() {
        Board result = _current;
        if (_maximize) {
           _current = max(_current.children());
        } else {
           _current = min(_current.children());
        }
        _maximize = !_maximize;
        return result;
    }
}
```