

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

CS61B  
Fall 2013

P. N. Hilfinger

Final Examination Solutions

1. [3 points] Provide short answers to the following questions. In each case, include an explanation of the answer.

a. What does `n & (n - 1) == 0` test?

**Ans:** *Tests whether  $n$  (treated as an unsigned—nonnegative—number is a power of 2 or 0. [We did not insist on worrying about 0 or negative numbers.]*

b. Write a single integer expression that is equal to `x*273` (273 in binary is 100010001), but that does not use `*`, `/`, or `%`. The expression must use fewer than 30 characters. Ignore overflow.

**Ans:** `(x<<8) + (x<<4) + x`



c. Assuming that  $0 \leq x \leq 15$ , write a single integer expression that is equal to `x*273`, as for part (b), but that not use any of the standard arithmetic operators `*`, `/`, `%`, `+`, or `-`. Again, the expression must use fewer than 30 characters.

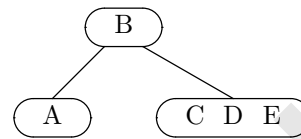
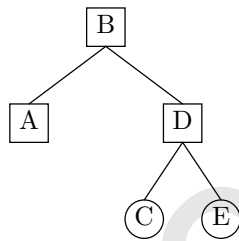
**Ans:** `(x<<8) | (x<<4) | x`

2. [6 points] Starting from an empty set of strings, represented as a 2-4 tree, we insert the letters 'A' through 'G' in order. After inserting 'A' through 'E', we end up with the following tree (leaf nodes, which hold no data, are not shown):

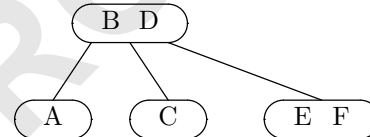
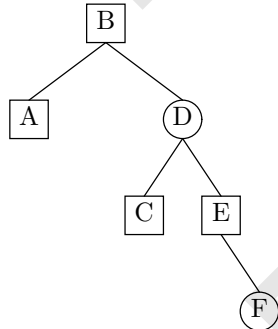
Red-Black Tree

(2,4) Tree

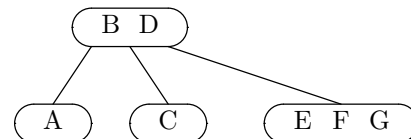
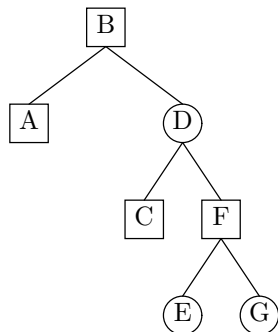
Notation. Black node:  Red node: 



After adding A-E



After adding F



After adding G

- Below the tree on the right, show the (2,4) trees resulting from inserting element 'F' and then from inserting element 'G'.
- To left of each of the three trees in (a), show a corresponding red-black tree. Use square and round nodes to indicate black and red, respectively.

Problem continues on next page.

*Continuation of problem 2.*

- c. True or False: The order in which one deletes items from a  $(2, 4)$  tree has no effect on the final tree.

**Ans:** *False.*

- d. Given a  $(2, 4)$  tree containing  $N$  keys, how would you obtain the keys in sorted order in worst-case  $O(N)$  time? We don't need actual code—pseudocode or an unambiguous description will do.

**Ans:** *Simply generalize an inorder traversal: traverse the left (first) child of the node, emit the first key, traverse the second child of the node, emit the second key, etc.*

3. [6 points] Suppose the types `Stack` and `Queue` both implement `Seq`, an interface that describes generic sequences of values:

```
interface Seq<T> {
    void add(T x);
    /** Removes and returns the last item in the sequence. */
    T remove();
    /** Returns true iff sequence is empty. */
    boolean isEmpty();
}
```

In a `Stack<T>`, `.add` adds to the end of its sequence (LIFO), while in a `Queue<T>`, `.add` adds to the beginning of its sequence (FIFO).

`Node` is a typical binary tree structure:

```
class Node {
    Node left;
    Node right;
    int value;
}
```

Fill in the method `isHeap` to satisfy its comment. Do not add lines outside the blanks. Each blank should contain one expression, statement, or partial expression.

```
/** Assuming T is the root of a binary tree containing unique integers,
 * returns true iff T has the max heap property and is a complete binary
 * tree (that is, each level is filled except possibly the last,
 * all of whose nodes are as far left as possible). */
public static boolean isHeap(Node T) {
    boolean foundLastKey;
    Seq<Node> f = new Queue (); // Stack or Queue?
    f.add(T);
    foundLastKey = false;

    while (!f.isEmpty()) {
        Node n = f.remove();
        if (n == null) {
            foundLastKey = true;
        } else if ( (n.left != null && n.value < n.left.value)
            || (n.right != null && n.value < n.right.value)
            || !foundLastKey )
            return false;
        } else {
            f.add(n.left);
            f.add(n.right);
        }
    }
    return true;
}
```

4. [1 point] According to legend, what did Odysseus do to anger Poseidon?

**Ans:** *Odysseus blinded Poseidon's son Polyphemus (one of the Cyclopes).*

5. [7 points] Indicate whether each of the following is true or false, and give a *brief* explanation for each (you *must* give an explanation to get credit).

- a. We can find the shortest path from a given node to another node in a directed graph with edge weights in the range  $[-\infty, \infty]$  using Dijkstra's algorithm.

**Ans:** *False. Dijkstra's algorithm will not work with negative edge weights.*

- b. For fixed  $k$ , the fastest algorithm to find the  $k^{\text{th}}$  smallest element of a binary search tree of size  $N$  is  $\Theta(N)$  time in the worst case, given that the tree is balanced.

**Ans:** *False. Since the tree is balanced, the time required to traverse to the bottom is  $O(\lg N)$ , and therefore the time to traverse a constant number of nodes (and  $k$  is assumed to be constant) must also be  $O(\lg N)$ .*

- c. If an algorithm for computing  $F(A)$  has a worst-case time bound of  $\Omega(N)$  for values  $A$  of size  $N$ , then computing  $F(B_i)$  for each of the specific values  $B_i$ ,  $0 \leq i < M$ , where the size of each  $B_i$  is  $M$ , takes at least  $M^2$  time in total.

**Ans:** *False. Just because computing  $F(A)$  takes at least  $kN$  time in the worst case (for some constant  $k$ ), it doesn't follow that it does so in the particular cases of  $F(B_i)$ .*

- d. Merge sort runs in  $O(n^2)$  worst-case time for  $n$  the number of items sorted (assuming constant-time comparisons).

**Ans:** *True. Merge sort is a divide-and-conquer algorithm with a linear “conquering” step, so that its time bound is  $\Theta(n \lg n)$  comparisons. However,  $n \lg n \in O(n^2)$ .*

- e. Merge sort runs in  $\Theta(n^2)$  worst-case time for  $n$  the number of items sorted (assuming constant-time comparisons). (WARNING: This is *not* the same problem as (d)!)

**Ans:** *False.  $n \lg n \notin \Theta(n^2)$ , since  $n \lg n \notin \Omega(n^2)$ .*

- f. Consider Kruskal’s algorithm for minimum spanning trees. Upon termination of the algorithm on any undirected connected graph  $G$  with vertices  $V$  and edges  $E$ , the union operation is called exactly  $|V| - 1$  times. ( $|V|$  denotes the number of vertices in  $G$ .)

**Ans:** *True. One edge is added to the tree on each union operation. A tree with  $|V|$  nodes has  $|V| - 1$  edges.*

- g. Assuming that the number of items in each bucket remains limited by a constant, looking up a key in a hash table requires constant time.

**Ans:** *False. We’ve said nothing about the time required to compute the hash function or perform equality comparisons. The statement is true if we assume that both of these take constant time.*

**6.** [4 points] Provide short answers to the following questions. **IMPORTANT:** In each case, include an explanation of the answer.

- a. What is the worst-case running time of the following procedure, as a function of  $N = U - L$ ? Assume that  $H$  executes in constant time.

```
int G(int L, int U) {
    if (L >= U-2) {
        return H(L, U);
    }
    else
        return G(L + 2, U) + G(L, U - 2);
}
```

**Ans:** Each call to  $G$  results in two calls to  $G$  with  $N$  diminished by 2. Hence

$$C_G(N) = \begin{cases} 1, & \text{if } N \leq 2; \\ 2 * C_G(N - 2), & \text{otherwise.} \end{cases}$$

(if we measure time in number of calls to  $H$ ). This has the solution  $C_G(N) \in \Theta(2^{N/2}) = \Theta(\sqrt{2^N})$ . [But we won't complain about  $\Theta(2^N)$ , since this is still exponential growth.]

- b. How fast does the program in part (a) run after it is memoized?

**Ans:** The number of pairs of arguments  $(L, U)$  that can come up is roughly  $\frac{1}{2}(N/2)^2$  (the  $\frac{1}{2}$  results from the fact that  $L \leq U$ ). It follows that the time bound becomes  $\Theta(N^2)$ ,

*Problem continues on the next page.*

*Continuation of problem 6.*

- c. Given an undirected connected graph  $G(V, E)$ , give an algorithm that determines whether the vertices of the graph can be colored in two colors, such that no two neighboring nodes have the same color. High-level pseudo-code is fine; we're not looking for complete Java code.

**Ans:** *Let's assume our colors are red and green, just to be concrete. Perform a depth-first traversal, starting from an arbitrary node. To visit a node,  $n$ , check the colors of its neighbors. If none are colored (this happens only for the starting node), color  $n$  red. Otherwise, if those that are colored are all colored red, color  $n$  green. If those that are colored are all colored green, color  $n$  red. If none of these cases hold, the graph is not 2-colorable.*

- d. Here is an algorithm that multiplies two integers:

```
/** Return the product of positive integers A and B. Does not use *. */
public static Int multiply(Int a, Int b) {
    Int product;
    if (a == 0 || b == 0) {
        return 0;
    }
    product = 0;
    for (Int i = 0; i < a; i += 1) {
        product += b;
    }
    return product;
}
```

If we assume that `Int` is a (new) integer type for arbitrarily large integers, which of the following describes the running time of this algorithm as a function of  $N$ , the minimum number of bits needed to represent `a` and `b`? Ignore overflow and assume addition takes constant time. Briefly explain your answer on the right.

- I.  $O(N)$ ;
- II.  $O(N \lg N)$ ;
- III.  $O(N^2)$ ;
- IV.  $O(2^N)$ ;

**Ans:** IV. *If  $a$  is an  $N$ -bit number, then  $a$ 's value can be as high as  $2^N$ .*



7. [3 points] In project 3, the three kinds of traversal implemented by `graph.Traversal` all have the same structure. Ignoring pre- and post-visits:

- Initialize the fringe to a starting vertex.
- while the fringe is not empty:
  - Remove an item,  $v$ , from the fringe.
  - If  $v$  is not marked,
    - \* Mark  $v$ ;
    - \* Visit  $v$ ;
    - \* For each successor,  $s$ , of  $v$  that is not marked:
      - add  $s$  to the fringe

This traversal may be continued (`continueTraversing`) from a new starting vertex. When this happens, the program is supposed to perform the same kind of traversal as before, treating all the previously marked nodes as already marked and visited. Describe concisely, but in sufficient detail how one can implement the traversal methods so as to avoid duplicating the algorithm above in each method, and for `continueTraversing` so as to avoid using any kind of conditional statements (`if`, `switch`, `while`, `for`) to determine what kind of traversal to continue.

**Ans:** Define a class `Fringe`, with one subtype for each kind of traversal, and let a `Traversal` contain a pointer to a `Fringe`. The idea is that `Fringe` implements methods for performing the steps “add  $s$  to the fringe,” “while the fringe is not empty,” and “remove an item,  $v$ , from the fringe.” Each traversal method creates a `Fringe` of the appropriate type and uses it with the generic algorithm given above. It also stores the fringe for use by `continueTraversing`.

8. [4 points] Consider the following binary-search tree type:

```
class BST {
    private int key;
    private BST left, right;
    private int count;
    ...
    public static int kth(BST T, int k) { see below
        ...n
    }
}
```

The value null represents the empty tree. The methods of BST maintain the `count` field so that `T.count` always contains the total number of nodes in the subtree rooted at `T` (its minimum value, therefore, is 1.) Assume that all keys in these BSTs are unique. Fill in `BST.kth` to agree with its comment.

```
/** Return the Kth smallest value in the tree rooted at T. Thus,
 * kth(T, 0) is the smallest value in T, kth(T, 1) the next larger,
 * etc. It is an error if there is no Kth item in T. */
public static int kth(BST T, int k) {
    if (k < 0) {
        throw new IllegalArgumentException("negative index");
    } else if (T == null) {
        throw new IllegalArgumentException("no such element");
    } else if (k == (T.left == null ? 0 : T.left.count)) {
        return T.key;
    } else if (T.left != null && T.left.count > k) {
        return kth(T.left, k);
    } else {
        return kth(T.right, k - 1 - (T.left == null ? 0 : T.left.count));
    }
}
```

[NOTE: However, we were lenient about handling null children, since in retrospect, we should have had empty trees be represented by nodes with a count of 0.]

9. [3 points] Billy Joe is designing an algorithm to solve the following problem: Given a set of airline tickets *possibly* detailing an itinerary from some starting location to some (distinct) ending location, return the particular ordering of tickets that provides the proper itinerary. For example, suppose the tickets provided are: NY→JPN, CA→NY, and JPN→TX. Then, the proper itinerary would be: CA→NY, NY→JPN, JPN→TX. For sets of tickets that don't allow such an ordering (such as NY→JPN, CA→JPN) or for which the order is not unique (such as NY→JPN, JPN→NY), return null.

- a. His initial solution is shown below. There are some problems with it! Billy has ignored a couple of edge cases concerning his input. Indicate the appropriate fixes to his implementation at the commented points.

```
class Ticket {
    String start;
    String end;
}

/** Given a list T containing tickets in arbitrary order, provide a
 * list of tickets in the proper order, or null if no order
 * can be found, or the order is not unique. */
public static ArrayList<Ticket> findItinerary(ArrayList<Ticket> t) {
    HashMap<String, Ticket> tickets = new HashMap<>();
    Set<String> ends = new HashSet<>();
    ArrayList<Ticket> itinerary = new ArrayList<>();

    for (Ticket ticket : t) {
        tickets.put(ticket.start, ticket);
        ends.add(ticket.end);
    }
    for (String start : tickets.keySet()) {
        if (!ends.contains(start)) {
            itinerary.add(tickets.get(start));
        }
    }
    if (itinerary.size() != 1) return null; // Ans: Here
    tickets.remove(itinerary.get(0).start);
    // Ans: Nothing needed here.
    while (tickets.size() != 0) {
        Ticket curr = itinerary.get(itinerary.size() - 1);
        Ticket next = tickets.remove(curr.end);
        if (next == null) return null; // Ans: Here
        itinerary.add(next);
    }
    return itinerary;
}
```

- b. As a function of  $N = t.size()$ , what is the running time of Billy's algorithm (assuming String comparison takes constant time)?

**Ans:** *If we assume that our hashing function also takes constant time and is well-behaved on this set of data, then each tickets.put, ends.add, ends.contains, and tickets.remove takes constant time. Each operation on itinerary takes either constant or amortized*

constant time. Thus, overall time is  $\Theta(N)$  in the worst case, since each loop iterates  $N$  times. If, however, we allow for the possibility that our hashing function misbehaves (say, maps everything to the same value), the worst-case time goes to  $\Theta(N^2)$ .

As a practical matter, of course, nobody has itineraries with, say, thousands of stops, so one could get good results even if one used `ArrayList` for `ends` and `ArrayLists` of pairs for `tickets`.

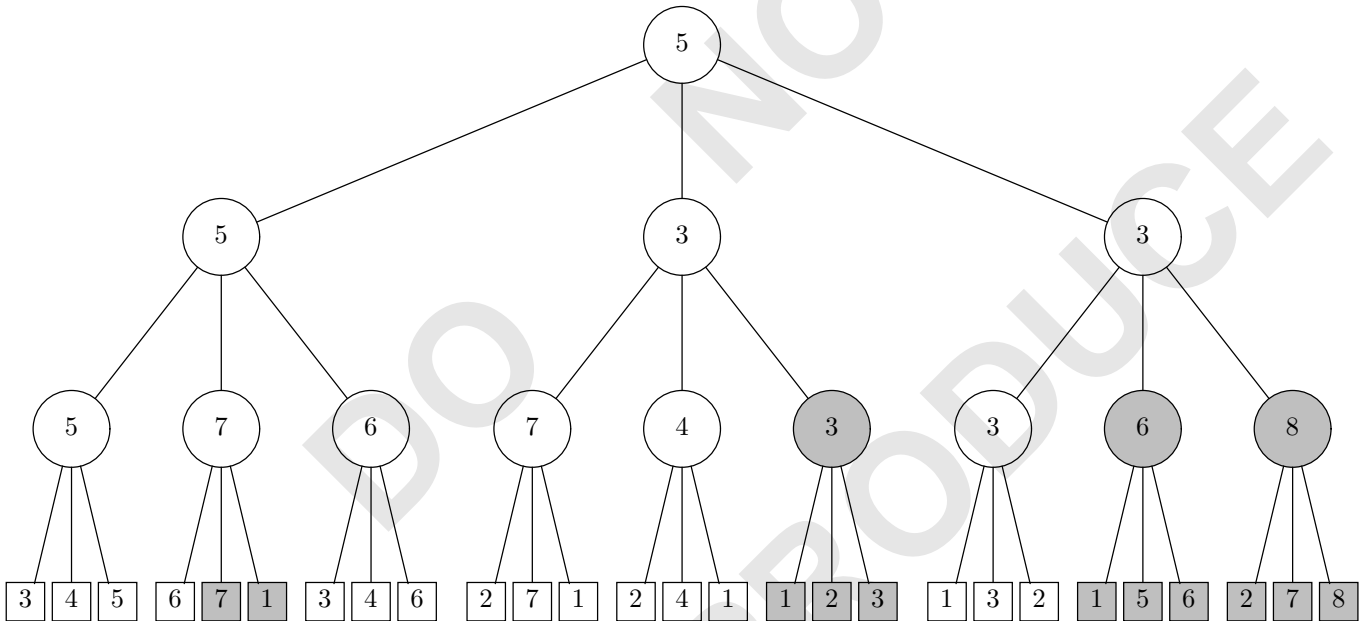
10. [6 points] Match up the sorting algorithms I–VI to the sequences a–f, which represent an array being sorted at some intermediate steps in the computation (not necessarily consecutive or evenly spaced). In each case, the original array to be sorted consists of the integers

985 228 678 819 906 602 650 385 805 424 590 185 838 749 546

	Intermediate Steps	Sorting Method
a.	228 985 678 819 906 602 650 385 805 424 590 185 838 749 546 185 228 385 424 590 602 650 678 805 819 906 985 838 749 546	<u>III</u>
b.	185 228 385 424 590 546 678 602 650 749 819 805 838 985 906 185 228 385 424 546 590 602 650 678 749 805 819 838 906 985	<u>VI</u>
c.	185 228 385 424 546 602 650 678 805 819 590 906 838 749 985 185 228 385 424 546 749 650 678 805 602 590 819 838 906 985 185 228 385 424 546 590 650 602 678 749 805 819 838 906 985	<u>II</u>
d.	650 590 602 424 985 385 805 185 906 546 228 678 838 819 749 602 805 906 819 424 228 838 546 749 650 678 985 385 185 590	<u>V</u>
e.	185 228 385 424 546 602 650 678 805 819 590 985 838 749 906 185 228 385 424 546 590 602 650 805 819 678 985 838 749 906	<u>IV</u>
f.	985 906 805 838 678 602 819 546 228 424 590 185 650 749 385 906 838 805 819 678 602 749 546 228 424 590 185 650 385 985	<u>I</u>

- I. Heap sort
- II. Quicksort
- III. Insertion sort
- IV. Straight selection sort
- V. LSD radix sort
- VI. MSD radix sort

11. [4 points] Consider the following game tree. The values in the square nodes at the bottom are static evaluations of positions where it is Player 2's move. The top node represents a position in which it is Player 1's move. All values represent the position values from the standpoint of Player 1 (larger means better for Player 1).



- Fill in the values for the positions in the tree above (represented by circles). Again, all values are from Player 1's standpoint (higher is better for Player 1).
- Show the effect of alpha-beta pruning, by circling the nodes (square and round) whose values the move evaluator need not compute (assume that each player evaluates positions left to right).

**Ans:** Here, we have grayed the nodes that can be pruned rather than circling them.