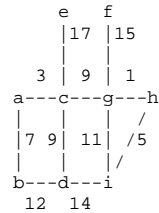CS 61B: Practice for the Final Exam

Please try to answer these questions. We'll release solutions two or three
days before the exam. Starred problems are particularly difficult--much more
difficult than any exam question would be.

Warning: Midterm 1 topics are absent here, but they can reappear on the Final.

[1] Given an array containing the digits 71808294, show how the order of the
     digits changes during each step of [a] insertion sort, [b] selection sort,
     [c] mergesort, [d] quicksort (using the array-based quicksort of Lecture
     31, and always choosing the last element of any subarray to be the pivot),
     and [e] heapsort (using the backward min-heap version discussed in Lecture
     30). Show the array after each swap, except in insertion sort. For
     insertion sort, show the array after each insertion.

[2] Some sorting methods, like heapsort and array-based quicksort, are not
     naturally stable. Suggest a way to make _any_ sorting algorithm stable by
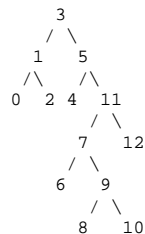   extending the keys (making them longer and adding extra information).

[3] Consider the graph at right.   e   f
                                                                   |17 |15
     [a] In what order are the vertices visited using DFS | |
          starting from vertex a? Where a choice exists, use   3 | 9 | 1
          alphabetical order. What if you use BFS?   a---c---g---h
     [b] A vertex x is "finished" when the recursive call | | | /
          DFS(x) terminates. In what order are the vertices |7 9| 11| /5
          finished? (This is different from the order in | | |/
          which they are visited, when DFS(x) is called.)   b---d---i
     [c] In what order are edges added to the minimum   12 14
          spanning tree by Kruskal's algorithm? List the edges
          by giving their endpoints.

[4] [a] How long does it take to determine if an undirected graph contains
          a vertex that is connected to no other vertex [i] if you use an
          adjacency matrix; [ii] if you use an adjacency list.
     [b] Suppose we use DFS on a binary search tree, starting from the root.
          The edge to a left child is always traversed before an edge to the
          right child. In what order are the nodes visited? Finished?
     [c] An undirected graph contains a "cycle" (i.e., loop) if there are two
          different simple paths by which we can get from one vertex to
          another. Using breadth-first search (not DFS), how can we tell if
          an undirected graph contains a cycle?
     [d] Recall that an undirected graph is "connected" if there is a path
          from any vertex to any other vertex. If an undirected graph is not
          connected, it has multiple connected components. A "connected
          component" consists of all the vertices reachable from a given
          vertex, and the edges incident on those vertices. Suggest an
          algorithm based on DFS (possibly multiple invocations of DFS) that
          counts the number of connected components in a graph.

[5] What does the splay tree at right look like after: 3
                                                                      / \
     [a] max()   [the operation that finds the maximum item]   1   5
                                                                         /\   /\
     [b] insert(4.5) \   0 2 4 11
                        | Start from the _original_ tree,   / \
     [c] find(10) | not the tree resulting from the   7   12
                        | previous operation.   / \
     [d] remove(9)   /   6   9
                                                                                / \
                                                                               8   10

[6] Consider the quick-union algorithm for disjoint sets. We know that a
     sequence of n operations (unions and finds) can take asymptotically
     slightly more than linear time in the worst case.

     [a] Explain why if all the finds are done before all the unions, a
          sequence of n operations is guaranteed to take O(n) time.
     [b] Explain why if all the unions are done before all the finds, a
     *   sequence of n operations is guaranteed to take O(n) time.
          Hint: you can tell the number of dollars in the bank just by looking
          at the forest.
                                                                  -----
[7] [a] Suggest a sequence of insertion operations   4 |3 5|
          that would create the binary tree at right.   / \   -----
     [b] Suggest a sequence of operations that would   2   6   / | \
          create the 2-3-4 tree at right. You are   / \   ----- --- ---
          allowed to use removal as well as insertion.   1   3 |1 2| |4| |6|
                                                                 ----- --- ---

[8] Suppose an application uses only three operations: insert(), find(), and
     remove().

     [a] Under what circumstances would you use a splay tree instead of a hash
          table?
     [b] Under what circumstances would you use a 2-3-4 tree instead of
          a splay tree?
     [c] Under what circumstances would you use an unordered array instead of
          a 2-3-4 tree?
     [d] Under what circumstances would you use a binary heap instead of
          an unordered array?
     [e] Under what circumstances would you use a hash tabel instead of
          a binary heap?

[9] [a] Suppose we are implementing a binary heap, based on reference-based
     *   binary trees (_not_ arrays). We want to implement a deleteRef()
          operation which, given a _reference_ to a node in the tree, can
          delete that node (and the item it contains) from the heap while
          maintaining the heap-order property--even if the node isn't the root
          and its item isn't the minimum. deleteRef() should run in O(log n)
          time. How do we do it?
     [b] Building on your answer to the previous question, explain how to
          combine a min-heap and max-heap (both using reference-based binary
          trees) to yield a data structure that implements insert(),
          deleteMin(), and deleteMax() in O(log n) time. Hint: You will need
          inter-heap pointers. Think of how you deleted edges in Project 3,
          for example.
     [c] How can we accomplish the same thing if we use array-based heaps?
          Hint: Add an extra field to the items stored in each array.

[10] Suppose we wish to create a binary heap containing the keys
      D A T A S T R U C T U R E. (All comparisons use alphabetical order.)

      [a] Show the resulting min-heap if we build it using successive insert()
           operations (starting from D).

      [b] Show the resulting min-heap if we build it using bottomUpHeap().

[11] [a] In Lecture 26, we told you how to implement a method
smallestKeyNotSmaller(k) that returns the smallest key not less than
k in a binary search tree. If the search tree contains an entry
with key k, then an entry with key k is returned.

Describe how to implement a method smallestKeyGreater(k) that
returns the smallest key strictly greater than k in a binary search
tree. Hint: write a slightly modified version of find() that acts
as if it were searching for k + epsilon, where epsilon > 0 is an
infinitesimally small number. Therefore, it is never an exact match
with any key in the tree. (This "hint" is actually a very useful
general-purpose technique worth remembering.)

For extra practice, code it in Java. Use the BinaryTree data
structure from Lecture 26.

[b] You are given a binary search tree that is NOT a splay tree and does
 * not rebalance itself. However, every node of the tree stores a
field that specifies the number of items/nodes in the subtree rooted
at that node (as described at the end of Lecture 40).

Given two search keys x and y, describe an algorithm that computes
the number of keys in the range [x, y] (inclusive) in O(h) time,
where h is the height of the binary search tree.

For extra practice, code it in Java. Assume that every
BinaryTreeNode has an extra int field named "size" that stores the
size of the subtree rooted at that node.

[12] Suppose we modify the array-based quicksort() implementation in the
Lecture 31 notes to yield an array-based quickselect() algorithm, as
described in Lecture 34. Show the steps it would use to find the median
letter in D A T A S T R U C T U R E. (The median in this case is the 7th
letter, which would appear at array index 6 if we sorted the letters.)
As in Question [1], choose the last element of any subarray to be the
pivot, and show the array after each swap.

[13] Suppose our radix-sort algorithm takes exactly n+r microseconds per pass,
where n is the number of keys to sort, and r is the radix (number of
queues). To sort 493 keys, what radix r will give us the best running
time? With this radix, how many passes will it take to sort 420-bit
keys? To answer this question, you'll need to use calculus (and a
calculator), and you'll need to remember that log2 r = (ln r) / (ln 2).

[14] Suppose that while your computer is sorting an array of objects, its
memory is struck by a cosmic ray that changes exactly one of the keys
to something completely different. For each of the following sorting
algorithms, what is the _worst-case_ possibility? For each, answer
[x] the final array won't even be close to sorted, [y] the final array
will have just one or two keys out of place, or [z] the final array will
consist of two separate sorted subsets, one following the other, plus
perhaps one or two additional keys out of place.

[a] Insertion sort
[b] Selection sort
[c] Mergesort
[d] Radix sort

[15] [Note: this is included for those who want some programming practice.
You are not responsible on the Final Exam for knowing anything about the
video Sorting Out Sorting.]

Implement tree sort (as described in the sorting video) in Java. Assume
your treeSort() method's only input parameters are the number of items
and a complete (perfectly balanced) BinaryTree of depth d in which each
leaf has an item; hence, there are 2^d items to sort. All internal nodes
begin with their item field set to null. Use the data structures below
(in which each node knows its left and right child), not a general tree.

Your algorithm should never change a node reference; only the items move.
The centerpiece of your algorithm will be a method that fills an empty
node by (i) recursively filling its left and right children if they're
empty, and (ii) choosing the smaller of its children's items, which is
moved up into the empty node. treeSort() will repeatedly: (i) apply
this method to the root node to find the smallest item remaining in the
tree, (ii) pluck that item out of the root node, leaving the root empty
again, and (iii) put the item into an array of type Comparable[]. Your
treeSort() should allocate and return that array.

```
public class BinaryTreeNode {        |  public class BinaryTree {
  Comparable item;                   |    BinaryTreeNode root;
  BinaryNode leftChild;              |    int size;
  BinaryNode rightChild;             |  }
}                                    |
```