Prelim 2 Solution

CS 2110, 24 April 2018, 5:30 PM

	1	2	3	4	5	6	7	Total
Question	Name	Short answer	Heaps	Tree	Collections	Sorting	Graph	
Max	1	16	10	20	11	18	24	100
Score								
Grader								

The exam is closed book and closed notes. Do not begin until instructed.

You have **90 minutes**. Good luck!

Write your name and Cornell **NetID**, **legibly**, at the top of **every** page! There are 6 questions on 8 numbered pages, front and back. Check that you have all the pages. When you hand in your exam, make sure your pages are still stapled together. If not, please use our stapler to reattach all your pages!

We have scrap paper available. If you do a lot of crossing out and rewriting, you might want to write code on scrap paper first and then copy it to the exam so that we can make sense of what you handed in.

Write your answers in the space provided. Ambiguous answers will be considered incorrect. You should be able to fit your answers easily into the space provided.

In some places, we have abbreviated or condensed code to reduce the number of pages that must be printed for the exam. In others, code has been obfuscated to make the problem more difficult. This does not mean that it's good style.

Academic Integrity Statement: I pledge that I have neither given nor received any unauthorized aid on this exam. I will not talk about the exam with anyone in this course who has not yet taken Prelim 2.

_____

(signature)

1. Name (1 point)

Write your name and NetID, **legibly**, at the top of **every** page of this exam.

2. Short Answer (16 points)

(a) True / False (8 points) **Circle** T or F in the table below.

(a)	T	F	All search algorithms take $O(n)$ time if the list is already sorted. False. Mergesort, for example, always takes $O(n \log n)$
(b)	T	F	If a class implements Iterable, it must also implement Iterator. False. Iterator is typically implemented by a private inner class, not the same class.
(c)	T	F	It is best to store dense graphs in an adjacency matrix and sparse graphs in an adjacency list. True. Adjacency list use space $O(\|V\| + \|E\|)$, so they are good for storing graphs with fewer edges. But checking for an edge in an adjacency list takes time proportional to the outdegree of a node, so adjacency matrixes are better for dense graphs.
(d)	T	F	`LinkedList<String>` is not a subtype of `LinkedList<Object>` even though `String` extends `Object`. True. If you don't know why, Look up "generics" in JavaHyperText.
(e)	T	F	An algorithm's time complexity can be both $O(n)$ and $O(n^2)$. True. Any algorithm that is $O(n)$ is also $O(n^2)$.
(f)	T	F	`HashMap<String, int> myMap= new HashMap<>();` is valid Java. False. Type parameters cannot be primitive types.
(g)	T	F	Consider an undirected graph with set of edges E and set of vertices V. It is a tree iff $\|E\| = \|V\|$ and there are no cycles. False. The formula should be $\|E\| = \|V\| - 1$
(h)	T	F	Any directed graph can be topologically sorted. False. The directed graph must be acyclic.

(b) GUI (3 points) What three steps are required to listen to an event in a Java GUI?

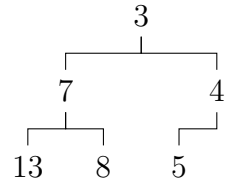
1. Have some class C implement an interface IN that is connected with the event.
2. In class C, override methods required by interface IN; these methods are generally called when the event happens.
3. Register an object of class C as a listener for the event. That object's methods will be called when event happens.

(c) Red-Black Trees (5 points) What is the full red-black tree invariant?

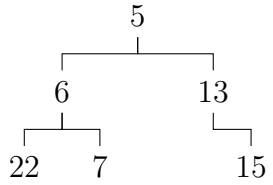
1. The tree is a binary search tree.
2. Every node is either red or black.
3. The root is black.
4. If a node is red, then its (non-null) children are black.
5. For each node, every path to a descendant null node contains the same number of black nodes.

3. Heaps (10 Points)

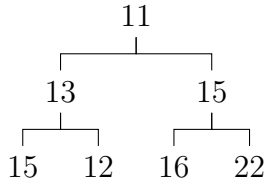
(a) 6 points State whether each tree below is a valid min-heap, in which the priorities are the values. If it is not, state which invariant is unsatisfied. (no partial credit)



valid min-heap



invalid: completeness invariant unsatisfied



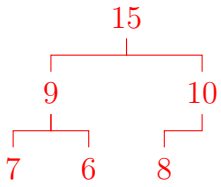
invalid: heap invariant unsatisfied

(b) 4 points The int array b below corresponds to a max-heap of integers in which the values are the priorities. Suppose the heap has size 7 and b[0..6] contains these 7 integers.

```
b =  [22  9  15  7  6  10  8]
```

Draw the state of b after calling b.poll(). Give your solution as an array. (Points deducted if you did not write your solution as an array.)

[15 9 10 7 6 8]

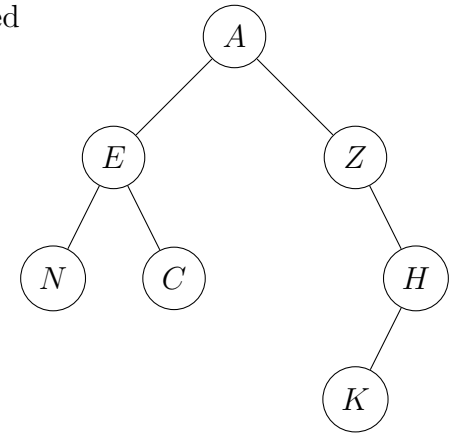


4. Trees (20 Points)

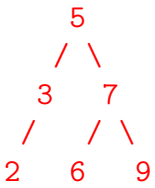
(a) 2 points What is the worst case time complexity for inserting into a:

- Binary Search Tree with n nodes? O(n)
- Red Black Tree with n nodes? O(log n)

(b) 4 points
Write down the order in which this tree's nodes are printed in preorder, inorder, and postorder traversals.
Preorder: A E N C Z H K
Inorder: N E C A Z K H
Postorder: N C E K H Z A



(c) 4 points Draw the binary search tree that results from inserting the following nodes, one by one, into an initially empty tree: 5, 7, 3, 2, 9, 6. Two pts. off for a mistake; 0 if your answer does not resemble the correct tree.

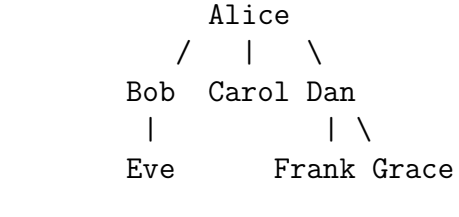


(d) 10 points

Class PhD, to the right, represents a PhD and their PhD committee. In the tree to the right, Alice is generation 0. Her committee members are Bob, Carol, and Dan; they are generation 1. Bob has a committee of 1, Eve, and Dan has a committee of 2, Frank and Grace. Eve, Frank, and Grace are generation 2. Etcetera.

Complete method `oldestGen` below.

```
public class PhD {
  private String name;
  private Set<PhD> committee;
  ...
}
```




```
/** Return the maximum number of generations that this PhD can trace
     * back in their committee history
     *
     * For example, in the tree on the previous page,
     * Alice.oldestGen() = 2 // Tree contains some of Alice's committee
     *                       // member's committee member(s)
     * Bob.oldestGen() = 1   // Tree contains some committee member(s) for Bob
     * Carol.oldestGen() = 0 // Tree contains no committee members for Carol
     * Frank.oldestGen() = 0 // Tree contains no comittee members for Frank*/
   public int oldestGen() {
     int oldest= 0;
     for (PhD cm : committee) {
         oldest= Math.max(oldest, 1 + cm.oldestGen());
     }
     return oldest;



   }
```

5. Collections and Interface (11 Points)

Class **MapBag**⟨E⟩, declared below, implements a bag —like a set but elements can be in it more than once. Implement methods `contains`, `add`, and `remove`. Assume all other methods required by Collection have already been implemented. Note: interpreting size as the number of entries in the map instead of the number elements in the bag is a mistake. Suppose I gave you a bag containing ten dimes. Would you say its size is 1 or 10? 10, of course.

```
/** An instance is a bag --an unordered collection in which there can be
  * multiple instances of the same element. */
public class MapBag<E> implements Collection<E> {
    private int size;              // Number of elements in this bag.
    private Map<E, Integer> map;   // Mapping of element in this bag to
                                   // its number of occurrences (which is > 0)

    /** Constructor: An empty bag. */
    public MapBag() { map= new HashMap<E, Integer>(); }


    /** Return whether this bag contains ob. */  (1 point)
    public boolean contains(Object ob) {
        return map.get(ob) != null;
    }

    /** Add e to the bag. Return whether the map was modified in any way. */  (4 points)

    public boolean add(E e) {
        if (contains(e)) {                OR   Integer s= map.get(e);
            map.put(e, map.get(e)+1);          if (s == null) map.put(e, 1);
        } else {                               else map.put(e, s+1);
            map.put(e, 1);                     size++;
        }.                                     return true;
        size++;
        return true;
    }

    /** Remove ob from the map if present. Return whether it was removed. */ (6 points)

    public boolean remove(Object ob) {
        Integer s= map.get(ob);
        if (s == null) return false;
        if (s == 1) map.remove(ob);
        else map.put((E)ob, s-1);
        size--;
        return true;
     }
}
```

6. Sorting (18 Points)

(a) 4 points. Consider the following class File. A file processing system requires files to arrive for processing in decreasing order of priority. If two files have the same priority, process the larger one first. Complete method compareTo(...).

```
/** An instance represents a comparable file object*/
public class File implements Comparable<File> {
   private String name;
   private int priority;
   private int size;
    ...
   @Override public int compareTo(File ob) {  // There are other solutions
     if (ob.priority == priority)
          return ob.size - size;
     return ob.priority - priority;
    }
}
```

(b) 6 points. Using method compareTo(), complete method insertionSort(), below. That is, complete the comment that begins the body of the for-loop as a high-level statement saying what is done to ensure that the loop invariant remains true. (2 points) Then implement the high-level statement. (4 points)

We developed this method in class, and JavaHyperText entry sort contains a pdf file that develops it. We deducted points for an inner loop that always iterates down to k= 0. It MUST stop as soon as the initial b[i] is in its final position. No regrade request on this will get points back.

```
public void insertionSort(File[] b) {
     // inv: b[0..i-1] is sorted
     for (int i= 0; i < b.length; i= i+1) {
         // Push b[k] down to its sorted position in b[0..k]
         int k= i;
         // inv: b[0..k] is sorted except that b[k-1] could be > b[k]
         while (0 < k  &&  b[k-1].compareTo(b[k]) > 0) {
             int t=  b[k-1];  b[k-1]= b[k];  b[k]= t;
             k= k-1;
         }
     }
}
```

(c) 2 points. What is the worse-case time and expected time of insertion sort? $O(n^2)$, $O(n^2)$

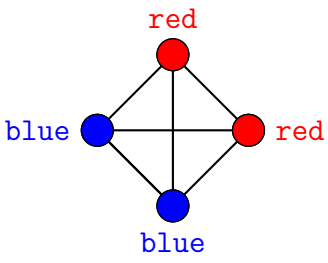
(d) 6 points State the tightest expected *space* complexity of quicksort, mergesort, and selection-sort. For quicksort, assume it is the version that reduces the space as much as possible.

quicksort: $O(log\ n)$ mergesort: $O(n)$ selectionsort: $O(1)$

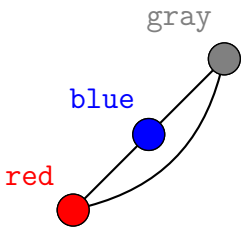
7. Graphs (24 Points)

A *coloring* of a graph is an assignment of colors to the nodes of a graph. An edge is *properly colored* if the two nodes it connects have different colors. A *proper coloring* is a coloring in which all edges are properly colored.

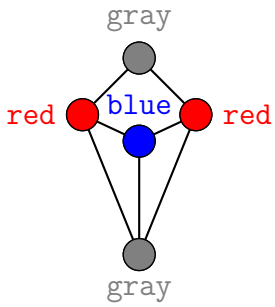
(a) 3 points Which of the following are proper colorings?



no

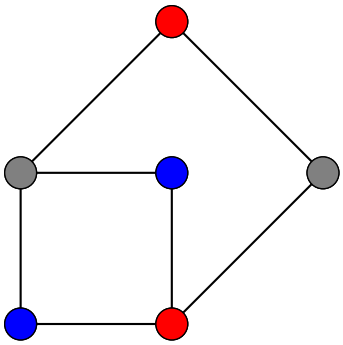


yes

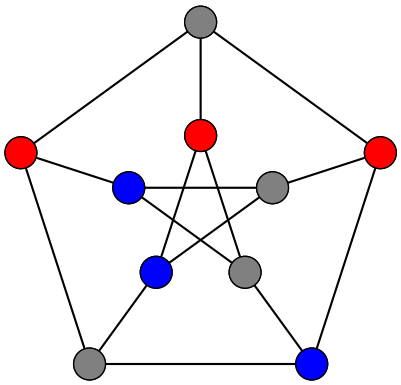


yes

(c) 4 points The *chromatic number of a graph* G, written as (χ_G), is the minimum number of colors needed to properly color the graph. For each of the graphs below, write down χ_G and properly color the graph using χ colors. To assign a color to a node, write the name of the color next to the node, as in part (a).



$\chi_G = 3$

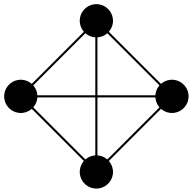


$\chi_G = 3$

(d) 4 points Draw a graph whose chromatic number is the same as the number of nodes in the graph (every node must get a different color). Use at least 4 nodes.

The graph must be a complete graph, where there is an edge between every pair of 2 nodes.

One example:



(e) 7 points Complete method `isProper`, below. The graph is undirected and connected. Starting with all nodes unchecked and `u` some node of the graph, the call `isProper(u)` will determine whether the graph is properly colored. Use the English phrases `n was checked` and `check n` to test whether a node `n` has already been traversed and to mark it as checked, respectively. Also, use a loop `for each neighbor m of n`.
Look at the specification. It does not say that n is checked. Therefore, the first statement MUST be: if (n is checked) return true; Second: n should be checked before its neighbors are traversed. But the precondition of any call implies that n should be checked only if its edges are properly colored. That is the reason for the first loop over the neighbors.

```
/** Let E be the set of edges that are reachable along unchecked paths from u.
    *  Return true iff all edges in E are properly colored.
    *  Precondition: All edges leaving a checked node are properly colored. */
   public boolean isProper(Node n) {
       if (n is checked) return true;
       for each neighbor m of n {
           if (n.color == m.color) return false;
       }
       check n;
       for each neighbor m of n {
           if (!isProper(m)) return false;
       }
       return true;
    }
```

(f) 1 point. Method `isProper` traverses the graphs in some fashion. It is similar to one of the following. Circle the one to which it is similar.

dfs. bfs. shortest path. prim. kruskal. The answer is dfs.

(g) 5 points. Two parts of the invariant of the shortest path algorithm are:

- There are no edges from the settled set to the far-off set.
- For each node u in the settled set, d[u] is the length of the shortest path from the start node to node u.

State (1) the third part of the invariant and (2) the theorem that is proved about the invariant.
(1) For f in the frontier set, d[f] is the length of the shortest path to f that consists of nodes in the settled set except for f.
(2) For f in the frontier set with minimum d-value (over nodes in the frontier set), d[f] is indeed the length of the shortest path from the start node to f.