NAME: _____SOLUTIONS___________________________ NETID: _________________________

CS2110 Fall 2010 Prelim 2 Solution Set
November 16, 2010

Write your name and Cornell netid . There are 6 questions on 11 numbered pages and 1 extra credit question on page 12. Check now that you have all the pages. Write your answers in the boxes provided. Use the back of the pages for workspace. Ambiguous answers will be considered incorrect. The exam is closed book and closed notes. Do not begin until instructed. You have 90 minutes. Good luck!

	1	2	3	4	5	6	XTRA	Total
Score	/15	/20	/20	/10	/15	/20	/5	
Grader								

1. (15 points) This first question tests your understanding of the heap structure we covered in class. *Assume all heaps are binary min-heaps.*

a. (4 points) Using the array-backed heap insert operation described in lecture, arrange these elements in a sequence that would give WORST-case performance for constructing a heap:
5, 3, 8, 2, 12, 23

23, 12, 8, 5, 3, 2

b. (4 points) `PriorityQueue.poll()` returns the element with the smallest priority. What priorities would you assign elements to make a FIFO priority queue? [Note: part b is unrelated to part a].

I would just assign the I'th inserted item priority i.

c. (3 points) What priorities would you assign to make a LIFO priority queue?

The I'th inserted item would have priority –i.

d. (4 points) Your friend Gary has written code that carefully maintains trees that are both valid min-heaps and valid binary search trees. He says this results in a heap that can also be searched in O(log n) time. Under what circumstances can a binary search tree also be a min-heap? Is Gary right?

In a min-heap, the lowest priority node for any subtree needs to be at the root of the subtree. In a BST, values smaller than the root of a subtree are to the left and larger to the right. It follows that Gary's construction is feasible but results in a tree that is actually a list: all the left pointers are null. A degenerate BST of this sort (a list) has O(N) search time, so Gary is wrong.

2. (20 points) True or false?

a	[✓ T]	F	Every Java Object supports synchronization (locking), and the wait() and notify() methods. However, primitive types do not support these methods.
b	T	[✓ F]	Suppose a program has a large constant data object, such as a matrix of parameters which is initialized statically and never modified again. Now suppose that the program has a set of threads that all perform read-only access to the shared object. In this situation, Java synchronization is required to ensure correct behavior.
c	T	[✓ F]	A heap is an O(log N) structure for inserting data and for finding the smallest element. If these operations are called "insert" and "find-min", a concurrent program with log N or more threads sharing a heap would achieve O(1) insert and find-min times.
d	T	[✓ F]	If a method has a locally declared object, say X, and two or more threads might concurrently execute that method, it is important to have a synchronization block around X
e	T	[✓ F]	In a heap representing N objects, using the array representation covered in class, the underlying data is stored *in sorted order in* the first N elements of a vector.
f	[✓ T]	F	A non-abstract class that implements an interface need not include code for interface methods that were implemented by some superclass.
g	[✓ T]	F	A Java class can implement more than one interface.
h	T	[✓ F]	A Java class can extend more than one class.
i	[✓ T]	F	A generic defined using a type pattern such as <? extends Animal> can be instantiated for any type that is a subtype of Animal, but not for a type that is a supertype of Animal.
j	[✓ T]	F	The <u>worst-case</u> performance of Quicksort is O(N log N) if the inputs are already in sort order. Assume that for a vector of N elements, the implementation uses element N/2 as its pivot.
k	T	[✓ F]	Synchronization makes it impossible to keep N threads active on an N core machine.
l	[✓ T]	F	If you insert the same element repeatedly into a set, the set should only retain one copy.
m	[✓ T]	F	Although a HashTable has expected lookup cost O(1), key collisions could result in performance as slow as O(N) for lookups and insert operations.
n	T	[✓ F]	If a problem can be solved using algorithm A in time O(log N), or with algorithm B in time O(n^2), there would never be any reason to use B (we would always want to use A)
o	[✓ T]	F	If an application inserts X, Y and Z into a stack, they will pop out in order Z, Y, X.
p	T	[✓ F]	If an application inserts X, Y and Z into a queue, they will be removed in order Z, Y, X.
q	[✓ T]	F	The ordering of objects in a priority queue is determined by the sort-order on their values.
r	T	[✓ F]	If the Frog class is a subclass of Animal, then Frogs can only support a method Jump if all Animals support the Jump method.
s	[✓ T]	F	When building a GUI, the designer can attach application-specific event handlers to standard components such as pull-down menus or buttons.
t	[✓ T]	F	A dynamic layout manager has the job of placing controls onto a GUI form according to a dynamic placement policy, which can vary depending on the choice of layout manager.

3. (20 points) In the following code, the developer was trying to understand how concurrency works, but got confused. His basic idea is to run two computations in parallel, but as part of his debugging tests he added the "count" variable seen below. We're hoping you can help him understand his mistake.

```
public class Q3 {
      private int count = 0;
      public class Worker1 implements Runnable{
            public void run(){
                  for(int i = 0; i < 10000; i++) {
                        //Inner class can access fields defined by parent
                        ComputeSomething(i);
                        count = count+1;
                  }
            }
      }
      public class Worker2 implements Runnable{
            public void run(){
                  for(int i = 0; i < 10000; i++) {
                        ComputeSomethingElse(i);
                        count = count-1;
                  }
            }
      }
      Public void runThreadExperiment() {
            for(int k = 0; k < 10; k++){
                  Thread T1 = new Thread(new Worker1());
                  Thread T2 = new Thread(new Worker2());
                  T1.start();
                  T2.start();

                  T1.join();  // This waits for thread T1 to finish
                  T2.join();  // And this waits for thread T2 to finish
                  System.out.println(count);
                  count = 0;
                  Thread.sleep(100);
            }
      }
}
```

a. Suppose that from Main the developer executes the following code:

```
      Q3 myQ3 = new Q3();
      myQ3.runThreadExperiment();
```

The developer was hoping this program would print 0 each time through the main loop. What might it actually print? (If the value of count is undefined when the println is done, say so and explain why).

In this example the count variable is updated by multiple threads without synchronization. The designer probably intended to see count=0 at the end of each main loop, but in fact the value will be undefined because if synchronization is omitted when two or more threads read and update a shared variable, Java makes no promises about the values that will result – none at all.

Hint: Give short answers for parts b and c, not essays! A one-line correct answer can get full credit and a page-long rambling answer might get no credit at all.

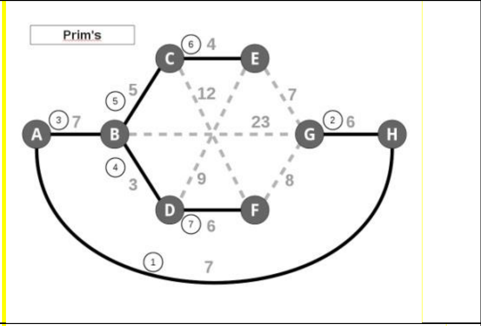
b. Explain why the code, as shown above, is incorrect. What basic Java rule is being violated here? Could the error have any effect on the value of count printed as the program loops? If you say yes, could this error change the number of times ComputeSomething or ComputeSomethingElse are called, or will each be called 10,000 times per execution of the main loop, as seems to be intended?

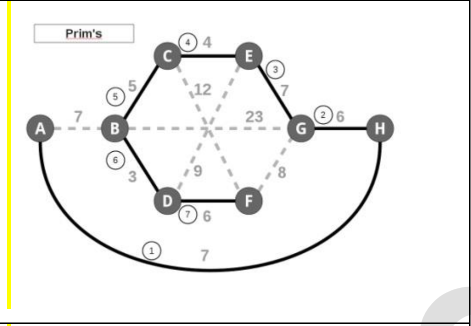
As just explained, the basic rule is that you need to have a synchronization lock on any variables shared by two or more threads that are updated during the sharing period. This error will not impact the number of times ComputeSomething and ComputeSomethingElse are called, but it can cause the program to print gibberish by leaving garbage in count.

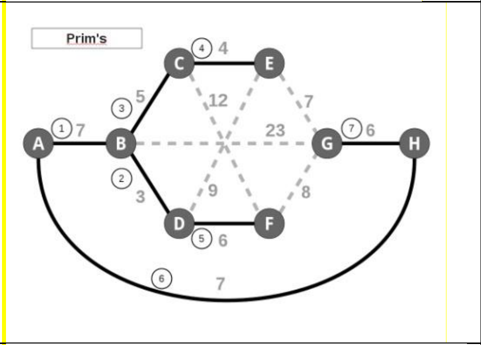
c. How can this code be modified to make sure that the behavior desired in (a) is what really occurs? Explain the effect of your change. If your modified version of the program is run on a two-core machine, would it really obtain any speedup? *Hint: Assume that if two threads "contend" for a lock on some object, each thread loses about 1/10,000th of a second in locking overhead, plus of course any delay until the lock is released, if the object was locked.*

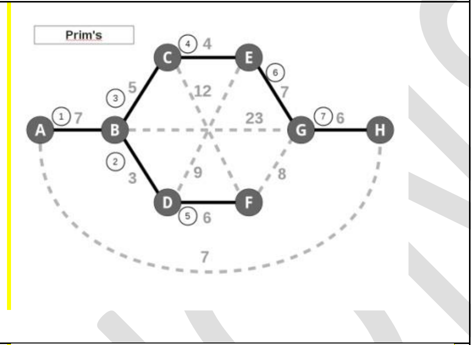
We need to enclose our access to count in a "synchronized" block. One way to do this is to just write
synchronized(this) ++count; and synchronized(this) –count;
We have some choice about where to put this block: if we synchronize the entire run procedure, only one thread runs at a time, and we get no speedup. The same is true if we synchronize to include the for loops. If we synchronize just for the access to count, though, each thread will experience 1 second of locking overheads. This seems like a lot. Best might be to modify the programs to use a local counter inside the threads themselves, and only update count "batch style" at the end of the for loops, using synchronization just on that line.

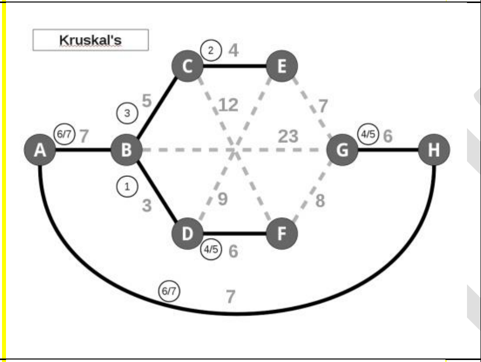
4. (5 points) Minimum spanning trees that we would accept as correct:

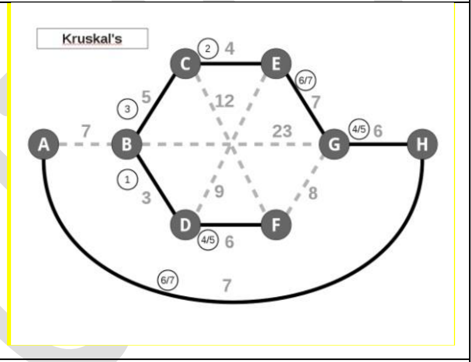


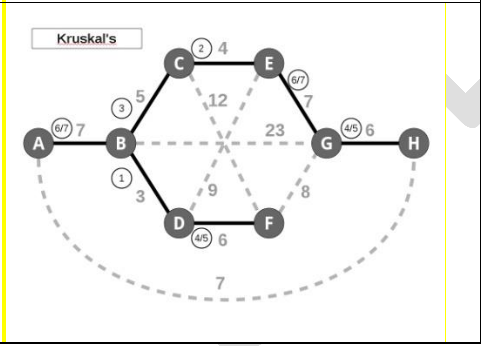




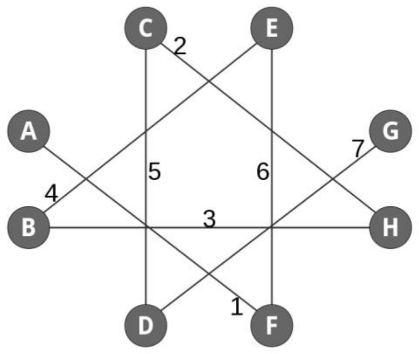








(b) (5 points) By looking at the order in which edges were added to a minimum spanning tree, you can sometimes tell which algorithm was used to construct it. This is true for the graph below. Tell us which minimum spanning tree algorithm (Kruskal's or Prim's) was used to construct this spanning tree. Give a one sentence explanation to justify your answer.

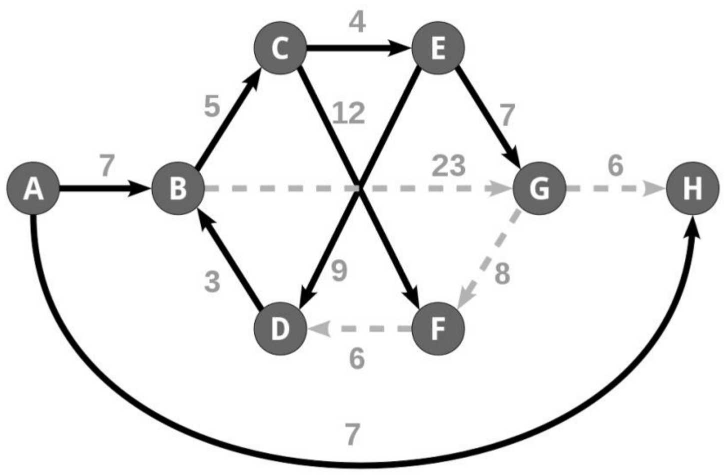


Edges are labeled to show the order they were inserted. Edge <u>weights</u> are not shown.

The answer for 4b is "Kruskal's because the first and second added edges do not have a vertex in common, so it couldn't have been Prim's because it starts from a single vertex and adds edges to it (and its connected component)".

5.(15 points) This question tests your understanding of Dijkstra's algorithm.

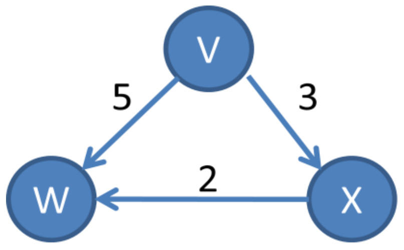
(a)(5 points) Run Dijkstra's algorithm on the graph below. Start Dijkstra's algorithm at node A and mark each node with the distance computed by Dijkstra's method. Then copy the node distances you computed into the table at the bottom of the page, so that we have an easy way to check your results.



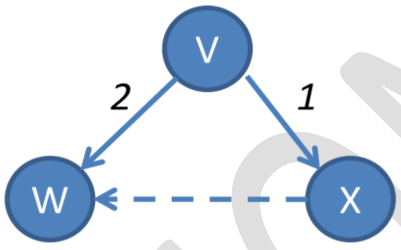
Node label	**Distance in each iteration**									
A	0	0	0	0	0	0	0	0	0	0
B	∞	7	7	7	7	7	7	7	7	7
C	∞	∞	∞	12	12	12	12	12	12	12
D	∞	∞	∞	∞	∞	25	25	25	25	25
E	∞	∞	∞	∞	16	16	16	16	16	16
F	∞	∞	∞	∞	24	24	24	24	24	24
G	∞	∞	∞	30	30	23	23	23	23	23
H	∞	7	7	7	7	7	7	7	7	7
Stage / explored node	init	A	H*	B*	C	E	G	F	D	fini

* The 'H' and 'B' columns can be explored in reverse order (since both costs were equal).

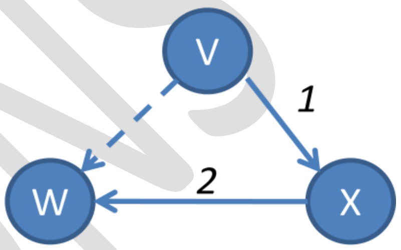
(b) (5 points) Although the two concepts are similar, minimum spanning trees are in fact different from shortest paths trees. Sometimes the MST of a graph happens to be the same as the shortest paths tree rooted at some vertex in that graph, but often it does not. In the next two parts we will illustrate this. Draw a simple graph containing a vertex **V** and at least 2 other vertices such that the tree resulting from running Prim's algorithm (starting at V) **is different from** the result of running Dijkstra's algorithm (also starting at V).



The Graph



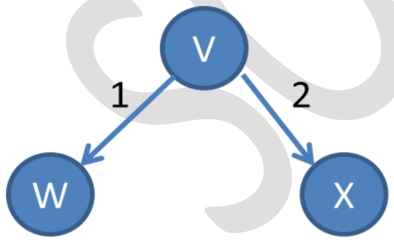
Dijkstras



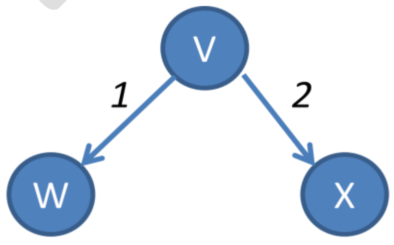
Prim's

The trick here is that there are two distance-five paths to W. The way Dijkstra's is coded causes it to find the length 5 edge from V to W in its first iteration, whereas Prim's follows next-shortest edges so it does the length 3 edge and then the length 2 edge, reaching W via X. (There are also ways to do this with graphs that have more nodes and edges, and in fact the graph used in part 4 is such a graph).

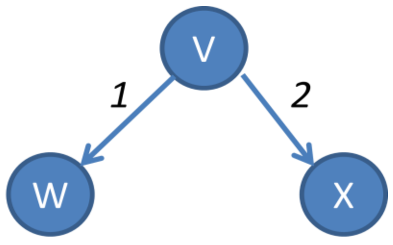
c) (5 points) Draw a simple graph containing a vertex **V** and at least 2 other vertices such that the tree resulting from running Prim's algorithm (starting at V) **is the same as** the result of running Dijkstra's algorithm (also starting at V).



The Graph



Dijkstras



Prim's

No need for any tricks this time. We just do a really simple graph that leaves no choices at all.

6. (20 points)

a) (10 points) Suppose that you have a job at Google Maps and are implementing a Java class to remember the distances between cities, so that they don't need to be recomputed every time the same question arises. Your class should have two methods, and because distances are symmetric, if the distance is known from city1 to city2, the same result should be returned for queries that use city2, city1 as the arguments. Distances are floating point numbers representing travel time.

```
public class Memories {
 // Remember the distance between two cities
 public static void saveDist(String city1, String city2, float d) {
... }

 // Return the saved distance, if known, else return -1.0
 public static float lookupDist(String city1, String city1) {  ... }
}
```

Provide code for these two methods. You can add fields to the Memories class if needed.

```
// My solution stores remembered distances in nested HashMaps
// The "outer" HashMap gets me from city1 to the appropriate inner
// HashMap.  That one is indexed by city1 and stores the distances
// I swap city1/city2 to make city1 always be alphabetically smaller
public class Memories {
 private HashMap<String, HashMap<String, float>> pensieve =
    new HashMap<String, HashMap<String, float>>();

 public static void saveDist(String city1, String city2, float d) {
    if(city1.compareTo(city2) > 0) {
       String tmp = city1; city1 = city2; city2 = tmp;
    }
    HashMap<String,float> inner = pensieve[city1];
    if(inner == null) {
        inner = new HashMap<String,float>();
        pensieve.Add(city1, inner);
    }
    if(inner[city2] == null)
        inner.Add(city2, d);
    else
        inner[city2] = d;
 }

 public static float lookupDist(String city1, String city1) {
    if(city1.compareTo(city2) > 0) {
       String tmp = city1; city1 = city2; city2 = tmp;
    }
    HashMap<String,float> inner = pensieve[city1];
    if(inner != null && inner[city2] != null)
        return inner[city2];
    return -1.0;
 }
```

b) (5 points) What is the expected complexity of your **`SaveDist`** method if called N times, for distinct city pairs? Justify your answer.

In my version, we compare the city strings to swap if needed, which costs O(1), and then do two hashed lookups to save a distance, and those have expected cost O(1). So the expected cost of the procedure is also O(1) (we can ignore constant factors, like the 2 or 3 in this case). But if something needs to call saveDist N times, that other procedure would have complexity O(N) since this O(1) costs are incurred once for each city pair.

c) (5 points) What is the expected complexity of your **`LookupSavedDist`** method, if called N times for pairs of cities that are randomly chosen, half being "known" ones and half "unknown".

The costs are identical (O(1) per operation hence O(N) to do the operation N times). The fact that the cities are half unknown doesn't really matter here since we still look up at least one of them and perhaps both and those operations still cost O(1) time.

Extra credit (5 points, but your total score can't exceed 100). Complex answers probably won't get much extra credit.

In class we developed a Heap class that stores data in a one-dimensional array and can perform insert and find-min operations in time O(log N). Unfortunately, the Heap structure we presented in class can't support both find-min and find-max: you can do one or the other, but not both. Without writing a single line of code, convince us that there is a simple way to support insert, find-min and **find-max**, all in O(log N) time per operation. *Hint: we don't mind if your solution uses a little extra space.*

Well, the rules for this problem suggest a trivial answer. Just keep two min-heaps, one that uses the item values as their priorities (hence it finds mins) and the other that uses the negative of the item values, hence find-min on it actually finds the item with the maximum priority. Yes, this doubles space and doubles the insertion times (since we need to insert each item twice), but apparently, we don't care.

Our challenge now will be to be able to remove items from both heaps if we find them in just one heap. The best way to do this is for each item in the heap to have a pointer to the index of the same object in the other heap. That is, if item X is in the min-heap, a field of X gives me the index to X in the max heap (which won't necessarily be the same, obviously). That way, when removing X from the min-heap I can also reach over and remove X from the max-heap. The same needs to be done in the other direction too: add a field in the max-heap to point to the item in the min-heap. Once again my costs are basically doubled, and I spend a bit of extra space.