Name: NetID:

Prelim 2 Solution

CS 2110, November 19, 2015, 7:30 PM

	1	**2**	**3**	**4**	**5**	**6**	**Total**
Question	True False	Short Answer	Complexity	Searching Sorting Invariants	Trees	Graphs	
Max	20	15	13	14	17	21	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** at the top of EACH 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 booklet is 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 its good style.

1. True / False (20 points)

a)	[✓ T]	F	Calculating the width of a tree (as in A4, the largest number of nodes at the same depth) always requires examining all of its nodes.
b)	[✓ T]	F	Let t be a node of a tree. Determining whether t is not a leaf takes constant time.
c)	[✓ T]	F	Let t be a balanced BST with n values. The worst-case time for inserting a value into t is $O(\log n)$.
d)	[✓ T]	F	To obtain the values of a BST in ascending order, use an inorder traversal.
e)	[✓ T]	F	Given only the preorder and inorder traversals of a binary tree, one can construct the tree (uniquely).
f)	T	[✓ F]	The language of a grammar that contains this rule must be infinite: $\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E}$ [If that is the only rule for E, there is no way to create a sentence —a sequence of terminal symbols.]
g)	T	[✓ F]	An unconnected graph with n nodes must have at least n edges. [An unconnected graph can have 0 edges.]
h)	[✓ T]	F	Topological sort requires that at each step, at least one unprocessed node has indegree 0.
i)	T	[✓ F]	A bipartite graph must have an even number of nodes. [The example of a bipartite graph in the lecture slides has an odd number of nodes.]
j)	T	[✓ F]	When using the adjacency matrix representation of a directed graph, in the worst case, determining whether there exists an edge from node u to node v requires more than constant time. [It's constant time; just look at matrix entry for (u, v).]
k)	T	[✓ F]	Breadth-first search can be written recursively or iteratively, but depth-first search can be written only iteratively. [BFS uses a queue, which makes writing it recursively extremely unnatural and difficult. DFS uses a stack.]
l)	[✓ T]	F	Dijkstra's shortest-path on a graph with all edge weights being 2 is a breadth-first search, not a depth-first search.
m)	[✓ T]	F	In the **best case**, there is a comparison-based sorting algorithm that can sort an array in $O(n)$ time.
n)	T	[✓ F]	All `JButtons` in a GUI must be "listened to" with the same `actionPerformed` procedure; it's not possible to have different `actionPerformed` procedures for different buttons. [We showed two actionPerformed procs. in lecture.]
o)	T	[✓ F]	It's best to declare all local variables used in a method at the beginning of the method, in order to save time allocating and deallocating space for them. [Our guidelines say to put them as close to their first use as possible.]
p)	T	[✓ F]	`String[]` is a subclass of `Object[]`, just as `HashSet<String>` is a subclass of `HashSet<Object>`. [Slides 11..13 of lecture 16.]
q)	T	[✓ F]	Methods in a class `C` with an inner class `IC` cannot call methods in `IC` that are declared `private`. [You did this kind of thing in A3.]

r)	T	F	If the value of function `equals(Object)` in a class depends only on fields `b` and `c`, function `hashCode()` in that class has to depend also on only those two fields.
s)	T	F	One can't use a "for-each" statement to process the entries of a `HashMap` because the keys of a `HashMap` are not ordered in any way.
t)	T	F	The purpose of interface `Iterable` is to make it possible to use a for-each statement.

2. Short Answer (15 points)

2.a Hashing (9 points)

Consider implementing a set using hashing with linear probing. Assume an array of 6 elements of class `Integer`, as shown below. We define the hash function $f(i) = (3 * i) \mod 6$, where 6 is the table size. For example, hashing 4 gives 12 mod 6, which is 0.

0	1	2	3	4	5
4 ~~null~~	null ~~XXX~~	null	5 ~~null~~	3 ~~null~~	1 ~~null~~

(i) 5 points Insert the values 5, 3, 5, 4, and 1 into the set, in that order —write the values in the appropriate element in the table above, crossing off the value currently in that element. Do not re-size the array, even though this is the standard way to implement a hash table.

(ii) 2 points Now remove the value 1 from the set. Do you simply set the array element to `null`? Explain why or why not.

No. We set the value of `isInSet` to `false`. If we simply set the array element to `null`, then the linear probing algorithm might stop searching for an element before it should.

(iii) 2 points Consider the insertion of values in point (i) above. Which insertion (if any) caused the load factor to surpass 0.4?

Insertion of the "4" changes the load factor to .5.

2.b Exception Handling 6 points

(i) 2 points Consider the statement below, appearing in a method `m`, where `b` is an int array. Does its execution result in a `RuntimeException` being thrown out to the call of `m`? Write your answer and an explanation for it to the right of the statement.

```
try {
    throw new RuntimeException();
}
catch (Exception e) {
    int x=  b[-1];
}
```

Yes. The line `int x= b[-1];` in the catch block throws an `ArrayIndexOutOfBoundsException`, which is a subclass of `RuntimeException`, that is not caught in `m`.

(ii) 4 points To the right below, write down what is printed by the `println` statements during execution of the call `mm(1)`, where method `mm()` is defined as follows:

```
public static void mm(int x) {
    try {
       System.out.println("11");
       int b= 5/(x-1);
       System.out.println("12");
       return;
    } catch (RuntimeException e) {
       System.out.println("13");
       int c= 5/(x-2);
       System.out.println("14");
    }
    System.out.println("15");
    int d= 5/(x-1);
    System.out.println("16");
    return;
}
```

11
13
14
15

3. Complexity (13 points)

(a) 4 points For each of the functions f below, state the function $g(n)$ such that $f(n)$ is $O(g(n))$. $g(n)$ should be as small as possible. (e.g. one could say that $f(n) = 2n^2$ is $O(n^3)$, but the best answer, the one we want, is $O(n^2)$.)

(i) $f(n) = n\log(n) + n + n^2$. $g(n) = n^2$

(ii) $f(n) = 2 + \frac{1500}{n} + 42n^3$. $g(n) = n^3$

(iii) $f(n) = 2^{n+4} + 300n^2$. $g(n) = 2^n$

(iv) $f(n) = 56$ $g(n) = 1$

(b) 3 points State the tightest (smallest) asymptotic time complexity (in terms of n) of the following statement sequence:

```
int s= 0;
for (int k= 0; k < 7; k= k+1) {
   for (int j= k-n; j < k; j= j+1) {
       s= s + j*k;
    }
}
```

$O(n)$. The outer loop has 7 iterations only; that does not depend on n.

(c) 6 points Give a formal proof that $f(n) = 30n + 2n^2$ is $O(n^2)$.

```
         f(n)
   =        <definition of f>
         30n + 2n^2
   <=      <for n >= 1>
         30n^2 + 2n^2
   =        <arithmetic>
         32n^2
So, use c = 32, N = 1
```

4. Seaching, Sorting, and Invariants (14 points)

(a) 6 points Assume that this procedure has already been written:

```
/** Partition b[e..f] by a random pivot in b[e..f]
  * Return the index j of the pivot, so that
  *       b[e..j-1] <= b[j] <= b[j+1..f] */
public static int partition(int[] b, int e, int f) {...}
```

Below, write the body of procedure `QS`, completely in Java.

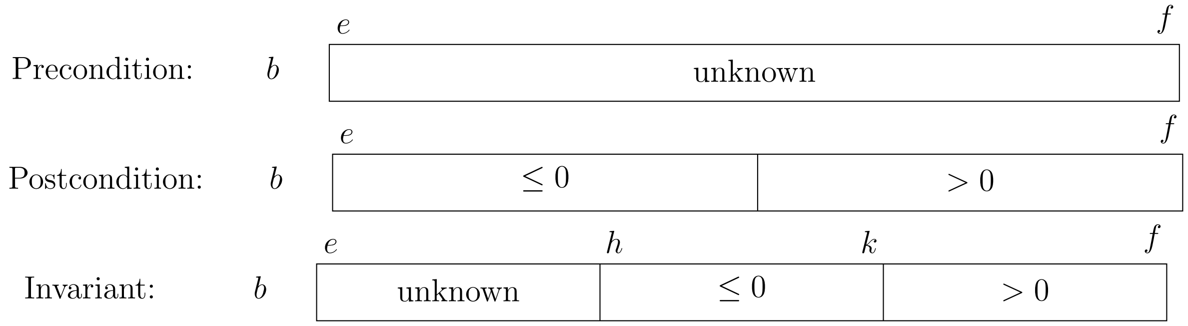
Solution

```
/** Sort b[e..f], using the quicksort algorithm. */
public static void QS(int[] b, int e, int f) {
   if (f <= e) return;
   int p= partition(b, e, f);
   QS(b, e, p - 1);
   QS(b, p + 1, f);
}
```

(b) 8 points Below is the precondition and postcondition of an algorithm to swap the positive values in $b[e..f]$ into the end of $b[e..f]$. Note that e is not necessarily 0 and f is not necessarily `b.length-1`. Do not change e and f. You may assume the following procedure has already been written:

```
/** Swap b[i] and b[j]. */
public static void swap(int[] b, int i, int j) {...}
```

Complete procedure `p` below using a loop that uses the given loop invariant to accomplish this task.



Solution

```
/** Modify b[e..f] as defined above. */
public static void p(int[] b, int e, int f) {
    int h = f + 1;            // 2 pts for init making inv true
    int k = f;                // 2 pts for inv && !cond imply result
    while (e != h) {          // 2 pts for progress toward termination
        if (b[h - 1] <= 0) {  // 2 pts for maintaining inv
            h= h - 1;         // pts may be deducted for doing what
        } else {              // shouldn't be done (e.g. change e or f)
            swap(b, h - 1, k);
            h= h - 1;
            k= k - 1;
        }
    }
}
```

5. Trees (17 points)

5.a Binary Search Trees (9 points)

Assume that each node of a binary search tree (BST), of class `Node`, has these fields:

- `Node left`: the left subtree (`null` if empty)
- `Node right`: the right subtree (`null` if empty)
- `int data`: the data of the node

Write the body of the following method, which appears in class `Node`:

Solution

```
/** Return true if v is in this tree and false otherwise.
  * Takes O(d) time, where d is the maximum depth of this tree
  * Precondition this is a BST */
public boolean contains(int v) {
    if (v == data) return true;
    if (v < data) return left != null && left.contains(v);
    return right != null && right.contains(v);
}
```

5.b Heaps (8 points)

Consider writing heapsort to sort an int array b into descending order. Complete the implementation of step (2) in the method below. You do not have to concern yourself with the implementation of step (1). In implementing step (2):

- Remember that $b[0]$ is the root of the heap and is the smallest value (it is a min-heap).
- Assume that function `int poll(int[] b, int k)` has already been written and can be used. It assumes that $b[0..k-1]$ is a heap, removes the root, does what is necessary to make $b[0..k-2]$ back into a heap, and returns the removed value.

Solution

```
/** Sort array b using heapsort */
public static void heapsort(int[] b) {
   // (1) Make b[0..b.length-1] into a min-heap
   heapify(b);

   // (2) Poll values from the heap and put them into their
   //       sorted (in descending order) position in b
   for (int i= b.length - 1; i >= 0; i--) {
        b[i]= poll(b, i + 1);
   }
}
```

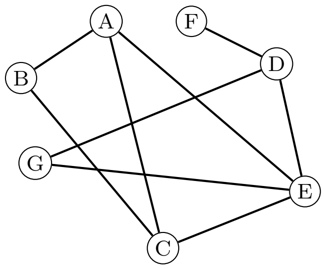
6. Graphs (21 points)

(a) 3 points There are two basic ways to implement a graph: (1) an adjacency matrix and (2) an adjacency list. Let a graph have n nodes and e edges. Below, for each point, state to the right which representation of a graph has that property:

(i) Takes time $O(n)$ to iterate over the edges that leave given node n: Adjacency matrix
(ii) Takes time $O(n)$ to determine whether there is an edge from node n_1 to node n_2: Adjacency list
(iii) Uses space $O(n^2)$: Adjacency matrix

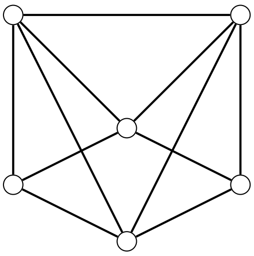
(b) 5 points For the graph below, give a list of the nodes that are visited by the recursive depth-first search algorithm starting at node A, in the order visited. Whenever there is a choice of processing nodes in any way, process them in alphabetical order by their names. Example:

to do something with nodes G, A, D, first do A, then D, and then G.



A, B, C, E, D, F, G

(c) 3 points Is the following graph a planar graph? Write yes or no to its right.



Yes

(d) 3 points State the difference between Prim's algorithm and Kruskal's algorithm for constructing a spanning tree of an undirected graph.

Kruskal's algorithm selects the smallest edge that does not introduce a cycle and adds it to the tree each iteration. Prim's algorithm starts with a specific node and selects the smallest edge leaving the connected component being built each iteration.

(e) 7 points Complete recursive algorithm `dfs`, given below. Do not be concerned about how visiting occurs. You may simply say "visit m" and "if m is unvisited".

Solution

```
/** Visit all nodes that are reachable along unvisited paths from m.
  * Precondition: m is unvisited. */
public static void dfs(Node m) {
    visit m;
    for (Node v : m.neighbors) {   // Could have said this a bit differently.
        if (v is unvisited) {      // We were not particular as long as it
            dfs(v);                // was understandable and correct
        }
    }
}
```