Name________________________

# CMSC 420, Fall 2001 – Final Exam

**Instructions:**
- Open book, open notes.
- Write only on the test sheet. If you run out of room, write on the back of the last sheet.
- To get any partial credit for wrong answers, you will need to show your work.

**Problem 1.** Let $f(n)$ and $g(n)$ be any two functions over the set of nonnegative integers.

**1a [5 points].** True or false: if $f(n) = O(g(n))$ then $f(n) = O(g(n)/2000)$.

True.

**1b [5 points].** True or false: if $f(n) = O(g(n))$, then $g(n) = \Omega(2000f(n))$.

True.

**Problem 2 [5 points].** Professor Prune says, "Here's a computer algorithm, along with an analysis showing that $O(n)$ is a lower bound on the algorithm's running time." What is wrong with Professor Prune's statement?

O(n) can be an upper bound, but not a lower bound. For a lower bound, Professor Prune would need to use Ω or Θ.

**Problem 3 [5 points].** Professor Prune says, "I have modified the *TopologicalSort* algorithm to return a list of the nodes in the order that they were visited. To do this, I made *TopologicalSort* start by creating an empty list *L*, and I modified *WalkForSort(v)* so that it appends *v* to *L*. Unfortunately, this gave me a worst-case running time of $O(n^2)$ rather than $O(n)$, where *n* is the number of nodes in the graph *G*." What did Professor Prune do wrong, and how should he fix it?

He shouldn't do a linear search of L to append v to the end of L. Instead, he should either put v onto the front of L and then reverse L when the algorithm terminates, or else maintain a pointer to the last element of L so that he can quickly insert v at the end.

**Problem 4 [5 points].** Professor Prune says, “I’m going to write an algorithm that will be similar to *TopologicalSort*, except that it will return several lists of nodes: one list for each possible topological sort of the nodes of *G*.” Without even knowing the details of this algorithm, what can we say about its worst-case running time?

Consider a graph G of n nodes. The worst case occurs when G has no edges. In this case, there are n! different topological sorts of G, so the running time must be Ω(n!). Since each list has length n, an even better bound is Ω(n!n).

**Problem 5 [10 points].** Draw a binary tree of minimal weighted path length with the weights 10, 12, 14, 18, 28, and 40 at its leaf nodes. What is the tree's weighted path length?

```
          122
         /   \
        /     \
      50      72
     / \     /   \
   22 28   32  40
  /  \      /  \
 /    \    /    \
10    12 14    18
```

WPL = 10*3 + 12*3 + 28*2 + 14*3 + 18*3 + 40*2 = 298

**Problem 6.** Let S and T be two trees, and suppose every key of S is smaller than every key of T. Suppose we want to write a function *concatenate*(S,T) that returns a tree U that contains all of the nodes of S and T. Tell what the worst-case running time would be for this function in each of the following cases:

**(a) [5 points].** S, T, and U are AVL trees.

In the worst case, all we can do is to insert the nodes of one tree one by one into the other tree. If n is the total number of nodes, this will take time Θ(n lg n).

**(b) [5 points].** S, T, and U are 2-3 trees.

$\Theta$(n lg n), for the same reason as in part a.

**(c) [5 points].** S, T, and U are splay trees.

In this case, we can splay S around infinity, and then make T the right child of the root.
This will take time $\Theta$(n), where n is the total number of nodes.

**Problem 7 [5 points].** What is the largest number of keys that a 2-3 tree of height *h* can hold?

Each node can hold at most 2 keys.
For each i, there are at most $3^i$ nodes at depth i.
Thus the answer is $2*3^0 + 2*3^1 + \ldots + 2*3^h = 2*(3^{h+1}-1)/2 = 3^{h+1}-1$.

**Problem 8 [5 points].** What is the largest number of keys that a (2,3) tree of height *h* can hold?

Each node can hold at most 2 keys.
There are at most $3^h$ nodes at depth h.
Thus the answer is $2*3^h$.

**Problem 9 [10 points]**. Professor Prune says that the Splay Tree Theorem (see p. 249) is incorrect. His argument is as follows.

> Suppose we start with an empty tree T and we do 2k+1 insertion operations, to insert the keys 0, –1, 1, –2, 2, …, –k, k. After each insertion operation, the tree will look like a "vine."
>
> For example, here is how the tree will look after 2i+1 insertions:

```
          i
         /
       i–1
       /
     i–2
     /
   ...
   /
 –(i–1)
  /
–i
```

> The next insertion will require 2i rotation operations, which will change the tree into a vine that looks like the following:

```
–(i+1)
   \
   –i
     \
   –(i–1)
       \
       ...
         \
        (i–1)
           \
            i
```

> Thus, over all 2k+1 of the insertion operations, the total number of rotation operations will be Θ(0+1+2+3+…+(k–1)) = Θ($k^2$). This contradicts the Splay Tree Theorem, which says that the amount of time should be only O((2k+1) log (2k+1)) = O(k log k).

Where is the error in Professor Prune's argument?

The tree won't be a vine. Even if it were, the next insertion will make it into a tree that is not a vine.

**Problem 10.** This problem involves *k*-d trees. As described on page 324, each left-child of a node *n* should have a key value that is ≤ the key value of *n*, and each right-child of a node *n* should have a key value that is > the key value of *n*.

**(a) [10 points]** Draw a balanced *k*-d tree for the following set of two-dimensional points:
{(6,15), (8,4), (4,9), (12,17), (16,10), (4,18), (18,6)}.

```
           (8,4)
          /     \    (split around x)
         /       \
     (6,15)     (16,10)
       / \         /  \
      /   \       /    \   (split around y)
     /     \     /      \
(4,9) (4,18) (18,6) (12,17)
```

**(b) [5 points]** Let *S* be a set of *n* two-dimensional points whose *x*-value s and *y*-values are all different. What is the height of a balanced *k*-d tree for *S*? Give an exact value.

$\lceil \lg n \rceil$

**(c) [10 points]** Draw a balanced *k*-d tree for the following set of two-dimensional points:
{(2,15), (2,4), (2,9), (2,17), (2,10), (2,18), (2,6)}.

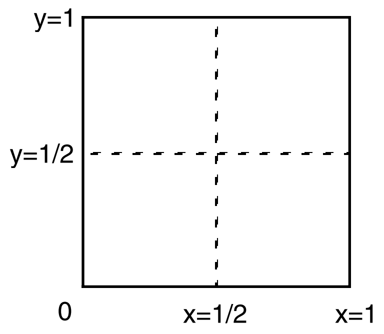
This time, each time we split around x we will get no right children. There are several possible trees we can get this way; here's one of them:

```
          (2,6)
            /         (split around x)
           /
        (2,10)
        /    \      (split around y)
       /      \
    (2,4)   (2,17)
     /          \          (split around x)
    /            \
  (2,9)     (2,15)
                 \          (split around y)
                  \
              (2,18)
```

**(d) [5 points]** Let *S* be a set of two-dimensional points where every point has the same *x*-value but no two points have the same *y*-value. If there are *n* points in *S*, then what is the height of a balanced *k*-d tree for *S*? This time a big-*O* value is OK.

The answer is O(lg n). One might think the answer should be more than this – but even though we have no right children each time we split around x, makes the tree only about
twice as high before, and O(2 lg n) = O(lg n).

**Problem 11 [5 points].** Let $p$ and $q$ be any two points in the interior of a two-dimensional rectangle such as the one shown below. Suppose we want to construct a region-quadtree (the kind of quadtree discussed in the book) such that $p$ and $q$ are in different regions. Give a worst-case upper bound on the height of the tree.

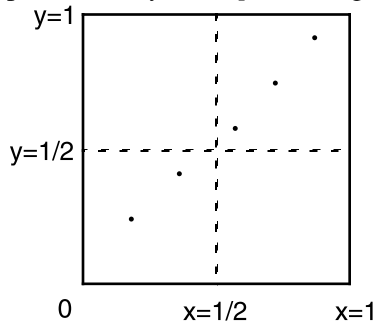


One possible answer is infinity (no tree will actually have infinite height, but no finite constant is a bound on the tree size).

Some of you tried to construct formulas giving the height of the tree as a function of the distance between $p$ and $q$. None of these formulas seemed to be correct—but I gave lots of partial credit for this kind of answer if you had the right kind of idea.

**Problem 12.** If $T$ is a point-quadtree (i.e., the kind of quadtree that V.S. Subrahmanian talked about), then for each point $p$ in $T$, we let *northeast(p), southeast(p), southwest(p), northwest(p)* be the sets of all vertices to the northeast, southeast, southwest, and northwest of $p$, respectively.

**(a) [5 points.]** In the interior of the two-dimensional rectangular region shown below, place five points in such a way that for every possible point-quadtree $T$ that might be constructed from these points, every node $p$ of $T$ is guaranteed to have *southeast(p)* = *northwest(p)* = $\varnothing$.



**(b) [5 points.]** Let $S$ be a set of two-dimensional points whose $x$-values and $y$-values are all different. Here are two possible ways to construct a point-quadtree for $S$:

- **Method 1.** For the root of the tree, choose any point $p$ in $S$ that minimizes the quantity
  $\max(|northeast(p)|, |southeast(p)|, |southwest(p)|, |northwest(p)|)$.
  Then, to choose the children of the root, invoke Method 1 recursively on the sets
  *northeast(p), southeast(p), southwest(p), northwest(p).*

- **Method 2.** For the root of the tree, choose any point $p$ in $S$ that maximizes the quantity
  $\min(|northeast(p)|, |southeast(p)|, |southwest(p)|, |northwest(p)|)$.
  Then, to choose the children of the root, invoke Method 2 recursively on the sets
  *northeast(p), southeast(p), southwest(p), northwest(p).*

If $S$ is a set of $n$ points similar to those described in part (a), which method will produce a more balanced tree?

Method 1. The problem with Method 2 is that $\min(|northeast(p)|, |southeast(p)|, |southwest(p)|, |northwest(p)|)$ will always be 0.

**Problem 13 [5 points].** Professor Prune says, “With coalesced hashing, items with several different hash values can end up in the same list—but with separate hashing, items with different hash values get put into different lists. Thus, separate hashing will never take any more probes than coalesced hashing.” Where is the error in this argument?

Professor Prune isn’t counting the list-header probe that’s needed for separate hashing.

**Problem 14** Suppose we have a memory space big enough to hold 1000 list cells, and we are managing that space using the “collecting by copying” strategy. Suppose the user is running the following procedure:

```
for $i$ = 1 to 10 do
    allocate $k$ list cells
    free all of the cells
end
```

**(a) [10 points]**. For each value of *k*, fill in the following table by giving the number of times we will do garbage collection, and the total number of cells copied during all of the garbage collections.

| | Number of collections | total number of cells copied |
|---|---|---|
| *k* = 200 | 4 | 400 |
| *k* = 300 | 9 | 1800 |
| *k* = 400 | 9 | 900 |

**(b) [5 points]** Explain what’s odd (if anything) about the above table.

We might expect that the more memory a program uses, the more time it would take to do garbage collection. However, between k=300 and k=400, the time goes down.