CSE326 Homework #5

Due: Friday July 29.

1. In class we saw an implementation of binary heaps using an array. We could also just as easily use the standard pointer based tree structure. In this case, merging two heaps of size m and n can be done in time $O(\log(n+m))$. Give a merge algorithm that achieves this bound. Why can't the same thing be done for the array based implementation?

2. One problem with array based data structures is that we need to know the maximum size of the array before hand. One possible way around this is a "stretchy" array: whenever the array becomes full a new array of twice the size is allocated, and all existing items are copied to the new array.

   In the following problems, our measure of time is one unit per array write. Let N be the size of the array before performing additional operations. For simplicity, assume we are doing stack operations, so that all reads and writes happen at the end of the array.

   (a) What is the best-case runtime for insert? What is the worst-case runtime?

   (b) Say the array starts out at size $N = 2$, and we insert M elements. What is the total runtime for the M inserts? Provide an exact answer, not just the order of growth. (Hint 1: Some operations are fast and some are slow. Write these as two summations and solve. Hint 2: Explain what you are doing–an answer that is just a series of equations will get little credit.)

   (c) Using the above answer, argue that the amortized runtime for *any* large enough sequence of operations is $O(1)$ per insert.

   (d) If our array shrinks we may want to copy to a smaller array to save space. An obvious first try is to cut the array size in half whenever less than half of the elements are being used. What is the amortized runtime of insert and delete in this case? (A simple example is fine, don't do a rigorous proof.)

   (e) We've shown that stretchy arrays have good theoretic guarantees. Do you think a stretchy array makes sense as something to do in practice? Hint: Look up how Java's Vector class handles this.

3. A bipartite graph $G = (V_1, V_2, E)$ is a graph such that the vertices are divided into two sets and all edges connect a vertex in V_1 with a vertex in V_2. Give an algorithm to determine if a given connected graph is bipartite. If the graph is not bipartite, your algorithm should return an odd-length cycle as proof that the graph is not bipartite. The algorithm should run in linear time. As always, you don't need to worry about the low-level details, just give the ideas behind the algorithm. (Hint: Use depth-first search. As you go, mark vertices as even or odd.)