

Homework 1, CS 520, McConnell, Fall '09, due February 3 before class

I will give you instructions about how to submit your program later.

1. Problem 4-1 on page 85 of Cormen (review).
2. For each of the following pairs $f(n)$ and $g(n)$, tell whether $f(n) = o(g(n))$, $f(n) = \Theta(g(n))$, or $f(n) = \omega(g(n))$. The identities on pages 52-54 are helpful.
 - (a) $f(n) = n$ vs. $g(n) = 4^{\lg n}$ (Consider rewriting the 4 as 2^2 .)
 - (b) $\lg n$ vs. $\log_{10} n$
 - (c) n^2 vs $4^{\lg n}$
 - (d) 2^n vs. 2^{n+1}
 - (e) 2^n vs 2^{2n}
 - (f) $n!$ vs. $(n+1)!$
 - (g) n^{100} vs 2^n
 - (h) $(\lg n)^{(\lg n)}$ vs n^3
 - (i) 2^n vs e^n
 - (j) $\lg n!$ vs n
 - (k) $\lg n$ vs $\lg \lg n$
 - (l) $\lg^* n$ vs $\lg^*(\lg n)$
 - (m) $\lg^*(\lg n)$ vs $\lg(\lg^* n)$

3. I sketched the VEB tree implementation of a priority queue in class.

For our purposes, a priority queue is an abstract data type that maintains a set of integers. It supports adding an integer to the set (**insert**), asking what the minimum element in the current set is (**find-min**), and extracting the minimum element from the set (**extract-min**).

We can use a VEB tree for implementing a priority queue that can only store elements in the range $[i, j]$, where i and j are specified when the data structure is created. The full definition of the tree appears in what follows. Let $n = j - i + 1$ be the number of elements in the range.

A disadvantage is the restriction on the range of values, and the fact that the data structure takes $\Theta(n)$ time to initialize. An advantage is that priority-queue operations take $O(\log \log n)$ time. Since priority-queue operations are often the bottleneck in an algorithm, this can be used to speed up the running time of many algorithms. The speedup is of theoretical interest. More importantly for our purposes, the data structure give some insight into how we can use induction to understand and invent structures that are so complicated that they can't be understood any other way.

The trick is to divide the range into *buckets*, where the first bucket handles elements in the range $[i, i + \lceil \sqrt{n} \rceil]$, the second is in the range $[i + \lceil \sqrt{n} \rceil + 1, i + 2\lceil \sqrt{n} \rceil]$, etc. This gives $O(\sqrt{n})$ buckets, each with a range of size $O(\sqrt{n})$. We implement the buckets with an array of recursive VEB trees. This indexes the recursive VEB trees by integers

in the range from 1 through \sqrt{n} . We use a *side tree* to implement a priority queue that holds the indices of the nonempty buckets.

The height of this tree is given by $H(n) = H(\sqrt{n}) + 1$. This can't be solved directly by the master theorem. However, letting $H'(m) = H(2^m)$, we get the recurrence $H'(m) = H'(m/2) + 1 = O(\log m)$ by the master theorem. Since $m = O(\log n)$, this is $O(\log \log n)$.

- (a) As a first attempt to get the better time bound, we can perform an **extract-min** by performing a **find-min** on the side tree to get the index i of the first nonempty bucket, and then an **extract-min** element from subtree i . If bucket i goes empty, then we must perform an **extract-min** on the side tree to get i out of it. Even if we ignore the cost of the **find-min**, we get a recurrence of $T(n) = 2T(\sqrt{n}) + 1$ for the running time.

Give a big- Θ bound on this recurrence by adapting the trick we used for bounding the height. You should find that the result is a disappointment.

- (b) If we could revise it to get the recurrence $T(n) = T(\sqrt{n}) + 1$, then we will get an $O(\log \log n)$ time bound, since this is the same as the recurrence for the height.

Let us modify the structure so that the minimum element is not stored in a bucket, but rather, stored in a separate variable, **min**, and the number of stored elements in a separate variable **num**. This gives an $O(1)$ time bound for **find-min**. It also reduces the time to extract the minimum element to $O(1)$ *if it is the last element in the tree*, since we just set **num** to 0 and return **min**.

Suppose our tree stores more than one element. To extract the minimum element, we perform a **find-min** on the side tree to find minimum index i of a nonempty bucket in $O(1)$ time. If bucket i has only one element, it takes $O(1)$ time to extract it from the bucket, and we remove i from the side tree in $T(\sqrt{n})$ time with a recursive **extract-min** operation. If bucket i has more than one element, the **extract-min** on it takes $T(\sqrt{n})$ time, but since bucket i didn't go empty, we don't need to extract i from the side tree. In either case, we have extracted the minimum element k that's stored in a subtree in $T(\sqrt{n})$ time.

Since the minimum element in the whole tree is stored in **min**, k is the second-smallest element stored in the tree. We decrement **num**, save the current value of **min** in a temporary variable, **temp**, store k in **min** and return **temp**.

This gives a new recurrence: $T(n) = T(\sqrt{n}) + 1$, which gives the $O(\log \log n)$ bound for **extract-min**.

Using this strategy as a model, describe how to get an $O(\log \log n)$ bound for **insert**.

- (c) We can use a naive data structure for small ranges. What is wrong with using a VEB tree to implement a bucket whose range has two elements?
- (d) I've posted some Java files that implemented a priority queue with a VEB tree, except that I have removed some of the code. You have to fill in the missing code. *Don't mess with the i/o interface, since I will depend on it for testing.*