Quiz/Homework Assignment #4
CSE326
Homework due date:
Wednesday, February 16th, beginning of class

February 19, 2000

This quiz is homework assignment #4. Any question you answered correctly in class were considered correct for the purposes of the assignment. Any question you did not answer or which you answered incorrectly was a short answer question for the homework assignment due in class on Wednesday.

In *italics* are instructions on how to answer each question for homework. The correct letter answer and an explanation of that answer is listed after each question.

Good luck!

1. *Describe the (most precise) asymptotic relationship relationship between each pair. Explain why (a) is the answer and why each of (b)-(e) breaks one of the conditions.*

   In which of the following is

   - $f(n) \in o(n)$
   - $g(n) \in \omega(n \log n)$ and
   - $f(n) > g(n)$ for some finite $n \geq 1$

   (a) $f(n) = \sqrt{4n}$, $g(n) = n^2$
   (b) $f(n) = n$, $g(n) = n \log n$
   (c) $f(n) = 2n$, $g(n) = n \log n$
   (d) $f(n) = 3$, $g(n) = 16^n$
   (e) $f(n) = 10$, $g(n) = 8$

   Correct answer(s): **(a)**
   $\sqrt{4n}$ is $\Theta(\sqrt{n})$, and that's asymptotically strictly less than n. So, $\sqrt{4n} \in o(n)$. We know that $n \in \omega(\log n)$ and so $n * n \in \omega(n * \log n)$ or $n^2 \in \omega(n \log n)$. Finally, for $n = 1$, $\sqrt{4 * 1} = \sqrt{4} = 2$ and $1^2 = 1$ so $f(1) > g(1)$.
   Precise asymptotic relationships:

   (a) $\sqrt{4n} \in o(n^2)$
   (b) $n \in o(n \log n)$
   (c) $2n \in o(n \log n)$
   (d) $3 \in o(16^n)$

(e) $10 \in \Theta(8)$

Why (b)-(e) don't work:

(a) $n \notin o(n)$

(b) $2n \notin o(n)$

(c) For all $n \geq 1$, $3 \leq 16^n$

(d) $8 \notin \omega(n \log n)$

2. *Explain why none of (b)-(d) actually is a flaw. In terms of the definition of Ω time, explain why (a) is a flaw.*

   The following is a *flawed* best case analysis of merging skew heaps in terms of n, the total size of the heaps. Which option below describes the flaw?

   > If $n = 4$ and both heaps are leftist heaps with half of the elements, neither one's root will have a right child. Therefore, the merge will take $\Omega(1)$ time.

   (a) The analysis does not generalize to arbitrary n.

   (b) The analysis does not generalize for different sizes of each heap with the same n.

   (c) Best case analyses derive tight bounds and upper bounds, not lower bounds

   (d) The analysis improperly assumes that the skew heaps are leftist heaps.

   (e) None of the above. The analysis is correct.

   Correct answer(s): **(a)**

   Since this is a best case analysis, we can choose the most fortuitous arrangement of the two skew heaps that we want. So, it's not a flaw to decide that the two heaps are the same size, and it's not a flaw to decide that they will be leftist (in shape). Both of these are reasonable best case assumptions. Also, a best case analysis can result in an upper, lower, or tight bound just as a worst or average case analysis can.

   Finally, part (a) does hit on a flaw because the definition of $f(n) \in \Omega(g(n))$ requires that there exist some c and n_0 such that *for all* $n > n_0$, $f(n) \geq c * g(n)$. In other words, the analysis must prove a property for every n larger than some constant. This analysis only proves a property for $n = 4$.

3. *Give the time required by the naive version of each data structure's "build" operation.* **Briefly** *explain and give the running time of the best "build" operation for each data structure.*

   We can build each of the following structures from a set of initial data by repeatedly calling insert until the data structure is complete, taking $O(n)$ times the time for an insert.

   However, we can do better with several of these data structures. Which of the following *cannot* be built asymptotically faster than the naive method described here?

   Hint: remember that any binary heap is a leftist heap!

   (a) AVL tree

   (b) Hash table

   (c) Splay tree

(d) Skew heap
(e) Leftist heap

Correct answer(s): **(b)**

(a) The naive build method for AVL trees takes $O(n \log n)$ (loosely, $O(\log n)$ for each of n inserts). A better build is the AVL buildTree method we described in class: assuming that the input is sorted, we just choose the middle value for the root, and recursively build the left and right subtrees out of the left and right halves of the array. This takes $O(n)$.
(b) The naive build method for hash tables takes $O(n)$ from n inserts taking $O(1)$ each. No matter what, each element will take some time to insert; so, we cannot build the table faster than this.
(c) It turns out that this is a reasonable alternate answer. Naive takes $O(n \log n)$ in general, but only $O(n)$ for sorted input. BuildTree with sorted input also gives us $O(n)$ (and a better resulting tree).
(d) The naive method takes $O(n \log n)$, but we can put all the data in the array, call the binary heap method "buildHeap", and turn the result into a skew heap in $O(n)$ time.
(e) As in the previous part, naive is $O(n \log n)$ but buildHeap gives us $O(n)$.

4. *Explain what properties each type of tree/heap does ensure. Explain why those properties result in a nearly balanced tree for (a), (b), and (d) and why they do not for (c).*

Which of these does *not* ensure a balanced or nearly balanced tree?

(a) binary heap
(b) d-Heap
(c) leftist heap
(d) AVL Tree
(e) All of these ensure at least a nearly balanced tree

Correct answer(s): **(c)**

(a) A binary heap ensures the heap order and heap shape properties. Heap shape means the tree is complete except the one level fringe which is filled in from left to right. Therefore, except for the fringe, a binary heap is perfectly balanced.
(b) A d-Heap ensures the same properties as a binary heap but may have higher arity, so it is also almost perfectly balanced.
(c) A leftist heap ensures the leftist property which states that the Null Path Length of any node's left child is at least as large as that node's right child's NPL. Perfectly balanced trees fit this description, but so do highly unbalanced trees, and leftist heaps make no effort to keep themselves balanced.
(d) AVL trees get their guaranteed good performance by ensuring balance. They are within a factor of about 1.44 of perfectly balanced even in the worst case. Their balance property ensures that the height of the left subtree of any node is no more than one different from the height of the right subtree.

5. *Give an example of a tree for which the given testing method would succeed but the tree is still not a BST. Explain what testing method is needed instead.*

   Can we ensure that a tree is a Binary Search Tree just by checking that every node's left child has a key less than the node's key and every node's right child has a key greater than the node's key? (Choose the most general answer.)

   (a) Yes, always.

   (b) No, not for any type of BST.

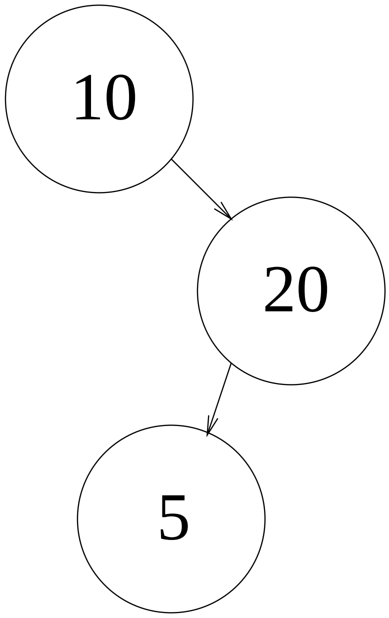
   (c) No, not if it's also an AVL tree.

   (d) No, not if it's also a splay tree.

   (e) No, not on Wednesday.

   Correct answer(s): **(b)**

   The testing method that is actually needed is to ensure that every key in every node's left *subtree* is less than the key, and every key in every node's right *subtree* is greater than the key.



6. *Review heaps and the heap-order property! Explain why all of these have the heap-order property (that is, what advantage do they gain from having heap-order?).*

   Which of these heaps enforces the property that a node's children's keys must have larger priority values than the node's key?

   (a) Binary heap

   (b) Leftist heap

   (c) Skew heap

   (d) d-Heap

   (e) All of the above

Correct answer(s): **(e)**

All of these gain the advantage of being able to find the minimum in constant time (and *very* fast constant time) from the heap order property.

7. *List how long (asymptotically) merge does take for each of these. Explain why the splay tree join operation cannot be used to support general merging.*

   Which of the following data structures supports a worst-case merge in $O(\log n)$ time?

   (a) Hash table
   (b) Binary search tree
   (c) Splay tree
   (d) AVL tree
   (e) None of these

   Correct answer(s): **(e)**

   (a) Hash tables take $O(n)$ time to merge (assuming that the load factor is $O(1)$). This is because each element of one table must be inserted into the other table.
   (b) Binary search trees take $O(n)$ time to merge. This is accomplished by reading all the values of each tree into two arrays, merging the arrays, and then using buildTree to construct the merged tree (each of these takes $O(n)$ time).
   (c) Splay trees take the same time as BSTs. The join operation can't be used in general because the trees may not have the property that one tree's keys are all less than the least of the other tree's keys (which is required for splay join to work).
   (d) AVL trees take the same time as BSTs.

8. *Explain how find minimum must be performed in a hash table. What's another operation that performs (similarly) poorly on hash tables?*

   Assuming we have a comparator for keys, hash tables support the find minimum operation in:

   (a) $O(1)$
   (b) $o(\log n)$
   (c) $\Theta(\log n)$
   (d) $O(n + tableSize)$
   (e) Not at all. This operation cannot be performed on a hash table.

   Correct answer(s): **(d)**

   In order to find the minimum in a hash table, we need to scan through every element of the table (of which there are $O(tableSize)$) and, for open hashing tables, every element of the chains (of which there are $O(n)$). Other bad operations are any dealing with order information: find maximum, find predecessor, find successor, range queries, etc.

9. *Explain why open hashing is always better for an insert under the condition in (c) if resizing is not allowed. Explain how open hashing could be worse for the conditions in (a) and (b).*

   An insert into an open hashing table (without resizing) is always better than an insert into a closed hashing table (without resizing) under which of the following conditions:

(a) $0 \leq \lambda < \frac{1}{2}$

(b) $\frac{1}{2} \leq \lambda < 1$

(c) $1 \leq \lambda$

(d) Open hashing is always better than closed hashing.

(e) Open hashing is never better than closed hashing.

Correct answer(s): **(c)**

A closed hashing table with $\lambda = 1$ is full, and the next insert will fail (or "take forever"; that is, run until the program crashes). An open hashing table will not. Therefore, open hashing is more efficient under these conditions.

However, as we saw in class, a closed hashing table can take approximately the same number of probes as an open hashing table for λ near $\frac{1}{2}$, and a closed table need not follow pointer chains. So, it may be practically more efficient under these conditions.