CSE250, Spr'13 Final Exam May 6, 2013

Open book, open notes, closed neighbors, 170 minutes. Do ALL SIX questions in the exam books provided. Please *show all your work*—this may help for partial credit. The exam totals 200 pts., subdivided (48,36,30,18,36,32) and further as shown.

(1) (9+9+9+9+12 = 48 pts.)

Let h be the hash function on strings that adds up number values of letters $a = 1$, $b = 2$, $c = 3$ etc., and let binary search trees compare strings in alphabetical order with the earlier (lesser) string on the left. (Unlike as on homeworks, all letters are used for this h.)

Show the process of inserting the strings `add, be, fad, bag, ace, ach, ah` in that order into the following data structures. Show the hash tables after `ace`, and then after `ah`—if you can! One picture of the BST is enough, while the red-black tree should be shown after *each* step, with colors on the nodes.

(a) A size-8 hash table with chaining, with new elements going at end-of-buckets.

(b) A size-8 open-address hash table, using linear probing: $h(k) + i$ for the i-th try.

(c) A size-8 open-address hash table, using the quadratic probe function $h(k) + i^2$.

(d) A simple binary search tree.

(e) A red-black tree—again you must show every step, and in some cases you can vary from the text's algorithm if you make clear that your "moves" are legal.

(2) (6 × 6 = 36 pts.)

Short answer questions: two sentences or formulas at most.

(a) Suppose a red-black tree has a node u that has only one child, call it v. What colors are u and v allowed to have, and how many children is v allowed to have?

(b) Suppose a heap has a u that has only one child, call it v. How many children is v allowed to have?

(c) Show using L'Hôpital's Rule that a running time of $n \log n$ is asymptotically faster than (i.e., little-oh of) a running time of $n^2 / \log n$.

(d) For what value of n and higher will a program with running time $n \log n$ actually start to out-perform a program with running time $n^2 / \log n$? Here logs are to base 2.

(e) Can one guarantee the creation of a simple binary search tree in $O(n \log n)$ time by first making the n elements into a heap, and then popping them one by one to insert into the tree in that order? Explain why or why not.

(f) Why is it impossible for an implementation of a stack using a `vector` to guarantee that every individual push operation will run in $O(1)$ time?

(3) ($10 \times 3 = 30$ pts.)

Let $ac(n)$ and $wc(n)$ respectively stand for the average-case time and the worst-case time of an algorithm running on instances of size n. (The former has the same meaning as "usual time" on Prelim II; the separate concept of *amortized time* for a repeatable operation is not involved here.) For each of the following algorithms, say whether it

(A) should only be run when n is small;

(B) runs well for "most" cases, but has poor performance in some cases—in particular has $ac(n) = o(wc(n))$; or

(C) runs well for all cases—formally, has $wc(n) = \Theta(ac(n))$, and $wc(n)$ is "*within an $O(\log n)$ factor of the best you could possibly hope for*."

Standard implementations of the algorithms and data structures they act on, such as we have seen or used in this course, are assumed. The five sorting algorithms all work on arrays of size n. The five cases of `insert` work one element at a time—i.e., they don't see all n elements "in advance." The algorithms are:

1. `MergeSort`.
2. `InsertionSort`.
3. `HeapSort`.
4. `QuickSort`, implemented by choosing the leftmost element in a sub-array as the "pivot" in every recursive step.
5. "`ValliSort`," implemented by calling `valli.insert(item)` on each of the n items.
6. `insert` n elements into an unsorted (singly) linked list that is behaving like a set.
7. `insert` n elements into an unsorted (singly) linked list that is behaving like a "bag" (a.k.a. multiset).
8. `insert` n elements into a hash table with open addressing and quadratic probing.
9. `insert` n elements into a "simple" binary search tree.
10. `insert` n elements into a red-black tree.

(4) (18 pts. total)

We would love to have a data structure that:

(a) guarantees that any sequence of n elements can be inserted one-at-a-time in $O(n \log n)$ steps,

(b) enables any element to be searched for in $O(\log n)$ time, and

(c) allows iteration through all its elements in sorted order in $O(n)$ time with a very small principal constant.

Here "guarantees" means "...in the worst case." The size of the (unspecified) principal constant in (a) and (b) is considered *not* of primary importance, not as important as the constant in (c). The constant in (c) should be close to the time taken to follow one pointer link or advance one place in an array (say, within a factor of 2, but not 3).

Consider `Valli`, a `Hash Table` with chaining, and `Red-Black Trees`. It is OK to state features for the first two even if you didn't code them—e.g. for `Valli`: refresh with $r = \log_2 n$ every time the input size doubles. For each one, say which of (a), (b), and (c) it satisfies, and which it doesn't. You must *justify* all your negative answers with a reason or an example of a case where it fails. (Brief justifications for positive answers aren't required but are a good idea anyway. Note that some answers repeat problem (3), and are included here just for completeness.)

(5) (12 + 12 + 12 = 36 pts.)

Suppose you are interfacing clients to a generic container class `Container` that uses the Standard Template Library interface, in particular with a `find` method that returns a valid iterator if a stored item with the same key is found, and otherwise returns the `end()` iterator.

(a) First suppose the client programs want to use a non-member function

```
template <class I, class Container>
void retrieve(const Container& cont, I& blankItem)
```

with the idea that the given `blankItem` can be mostly blank except for the key (similar to what was done when processing `Stock` transactions by the ticker symbol). If a stored item with the same key is found, (the rest of) its information is copied to `blankItem`, else `blankItem` is left as it is.

Write a body for `retrieve`, using only `find` and other features of the STL interface. What do you need to require ("REQ") about the template item type `I`?

(b) Now suppose instead that the clients want to use item pointers, and wish to return a pointer to the stored item itself. Code

```
template <class I, class Container>
I* retrieve(const Container& cont, const I& blankItem)
```

instead. Do you still need the same requirement(s) on `I`?

(c) Name two general advantages of doing (b) compared to (a), in terms of (i) the items themselves possibly being large, and (ii) what do you do now in the not-found case? Finally (iii) talk about the use of getting a pointer to the stored item, but also a possible problem given how the method in (b) is coded.

The last problem is overleaf.

(6) (6 + 12 + 6 + 8 = 32 pts.)

Suppose you have two heaps `h1` and `h2`, each with n elements heaped according to the same comparison function, with the maximum element on top. Given any number $k \leq 2n$, you want to generate a list of the top-k elements between the two heaps.

(a) What is the asymptotic running time if you pop off n elements from one heap, insert them one-by-one into the other heap, and pop k elements from the latter?

(b) Find a smarter way, one that works in $o(n)$ time. State your method and give its running time, proving that the time is $o(n)$. You may use any other data structures that you wish.

(c) If you have a vector with $2m$ elements in any order, and you want to find the top m elements, does it matter to the asymptotic running time if you make a heap and pop m times, versus sorting the entire vector?

(d) In (c), what if the vector instead comes with the first m elements initially sorted among themselves, and the second m elements ditto? Harking back to (b), does an $O(m)$ runtime here matter to the entire asymptotic running time of the task in (b)?

End of Exam