CSE250, Spring 2014 Final Exam May 14, 2014

Sample for CSE250, Spring 2022. Lightly translated into Scala. Your rules will be the same. Open book, open notes, closed neighbors, 170 minutes. Do ALL FIVE questions in the exam books provided. Please *show all your work*—this may help for partial credit. The exam totals 200 pts., subdivided (48,36,30,56,30) and further as shown.

(1) (6+6+9+6+12+9 = 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. Show the process of inserting the strings `ace, bad, bed, bag, ebb, beg, did` in that order into the following data structures. Show the hash tables and the ~~red-black~~ AVL tree after `bed`, after `ebb`, and then after `did`—if you can! One picture of the BST is enough, while the `BALBOA` should be shown after each split.

(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) An AVL tree.

(f) A `BALBOA ba` with nodes of capacity 4, where a first non-dummy node is created to store `ace`, and then each of the other strings x is inserted after the previous one. Nodes split when they reach (not exceed) size 4.

(2) (6 × 6 = 36 pts.)

Short answer questions: two sentences or formulas at most.

(a) Suppose we begin with an empty `BALBOA` object `ba` and execute `ba.insert(ba.size(), x);` in a loop for n different items x, using the indexing version of `insert` from Project 1. Assume the arrays have capacity roughly $c = \sqrt{n}$. Is the total running time $O(n)$? Justify your answer.

(b) Suppose we begin with an empty `BALBOA` object `ba` and execute `ba.insert(ba.end, x);` in a loop for n different items x, using the iterator version of `insert` from the "ISR" repository. Assume the arrays have capacity roughly $c = \sqrt{n}$. Is the total running time $O(n)$? Justify your answer.

(c) Same question as (b), except that now we use the "pre-allocated" representation of the array, meaning it operates the way the text describes for heaps in chapter 18: When a new linked-list node with an array is allocated, the array is initialized to size c not size zero, but with `end` set equal to 0 to mark the first free cell. Then when the new element x is inserted at the end, an assignment like `elements(end) = x; end += 1` is executed.

(d) Now suppose we insert new elements at the place where they would go to keep the BALBOA in sorted order, rather than at a given index or iterator. Explain why it is impossible for the n inserts to take $O(n)$ time now. [Spring 22: this is a reference to the theorem, not in our text, that every comparison-based sorting algorithm requires $\Omega(n \log n)$ time.]

(e) If $f(n) = o(g(n))$, then is $f(n)^2 = o(g(n)^2)$? Justify briefly.

(f) Why is AIOLI a better choice than BALBOA for an application that involves a lot of insertions and removals in the middle of the data structure, not just building it once and then making heavy use of find and iteration as on assignments 4 and 6?

(3) (10 × 3 = 30 pts. total)

For each task below labeled 1.–10., say which of these best describes its running time:

(a) Guaranteed $O(1)$ time.

(b) Amortized $O(1)$ time.

(c) Usually $O(1)$ time.

(d) Guaranteed $O(\log n)$ time.

(e) Usually $O(\log n)$ time.

(f) Guaranteed $O(\sqrt{n})$ time.

(g) Guaranteed $O(n)$ time.

In all cases n denotes the number of items currently in the underlying data structure, and any other parameters are stated. The variable `arr` stands for an `ArrayBuffer`, `deq` for a deque, `dlist` for a doubly-linked list (unsorted), `ba` for a "BALBOA" data structure with $c \simeq \sqrt{n}$, `bst` for a BST—i.e. a general binary search tree, `avl` for an AVL tree, `itr` for an iterator of the appropriate kind, and `item` for a typical item in the data structure. *All of these objects use the same interface as in the "ISR" repository. Justifications* are not required, but might help for partial credit.

1. For a BST iterator `itr`, the call `remove(itr);`
2. For an AVL tree iterator `itr`, the call `remove(itr);`
3. `ba.remove(ba.begin);`
4. `arr.insert(arr.begin,item);`
5. `dlist.remove(itr);`
6. For a `HashSet` data structure `s`, the call `s.find(item);`
7. For two `BALBOA` iterators `itr1` and `itr2`, the test `itr1.equals(itr2);`

8. For a deque `deq`, n consecutive calls to `popRear();`
9. Given an AVL tree `avl` with n elements and a BST `bst` with only $n/\log_2 n$ elements, copying the latter from `bst` into `avl`.
10. Given two *unsorted* `BALBOA` objects `ba1` and `ba2`, with the same capacity c, creating a new `BALBOA` as the union of the two. [Spring 22: this question is less solid in Scala than it is with the C++ implementation we used.]

(4) (9+3+9+3+9+2+21 = 56 pts. total)

Suppose you have an online trading service for role-playing-game cards, such as Pokemon or Yu-gi-oh or Magic: The Gathering. Each card has a name (such as "Pikachu" or "Voice of Resurgence") and a "par price" in your catalog. Users of your service have ID numbers which are consecutive integers $1, 2, \ldots, U$, while the cards do not have numbers[1] Each user can sell cards to you at the par price p, and can *bid* for cards at a price q that might be over or under p. Bid requests are recorded in a file with N lines of the form:

```
[userid]        [card_name]          [bid_price q]
```

Of course you sell the cards you have in stock to the highest bidders. What you now want to find out are the k users who tend to bid the most over par. That is, for every user u, let b_u be the number of bids u makes. Let S_q be the sum of the bid prices on these cards, let S_p be the sum of the corresponding par prices, and let $P_u = (S_q - S_p)/b_u$. You want the k users u with the highest P_u values.

(a) Of the data structures (i) vector/array, (ii) linked-list, (iii) red-black tree, or (iv) hash-table, which one(s) are most suitable for the *users*? Are any of them *poor*, meaning usual access time more than $O(\log U)$ per user lookup?

(b) Would `BALBOA` have any advantages here? What if many users closed their accounts and got removed?

(c) Of the same data structures (i)–(iv), which one(s) are most suitable for the *cards*? Which ones are *poor*, this time meaning more than $O(\log M)$ time per lookup in average case, where there are M cards?

(d) Suppose you read the N bids from the file into a linked list. Is that enough, or should you subsequently store copies of (or pointers to) bids in instances of a `User` class?

(e) Using the `ISR` and `ISR#Iter` interface, write code to iterate through a `list<Bid>` object called `bids`, look up the user number by a method `size_t getUser()` of the `Bid` class, and store the bid with the corresponding user in a vector `uvec` using a method `def addBid(bid: Bid): Unit` of the `User` class.

(f) Which method(s) in part (e) treat the data as immutable (i.e., would be `const` in C++)? [Exam extra-credit (4 pts.): how might one be "legally `const`" without being "morally const"?]

[1] Or if they did, the numbers would not be consecutive.

(g) Give an algorithm for computing the top-k list. A pseudocode sketch is fine—you may name some functions such as `sort` or `makeHeap` but need not write exact Scala code. Finally and most important, give an asymptotic formula for your algorithm's running time in terms of the number U of users, M of cards, N of bids, and k. (Times that are within logarithmic factors of optimal will not lose credit, and depending on your choices and any reasonable assumptions, not all of U, M, N, k might appear.)

(5) (30 pts.)

Do ONE of the following two programming tasks, *your choice*. Note that one uses indices, the other iterators. [Spring 2022: they are IMPHO a little easier than the C++ tasks actually given in 2014.]

I. Code an indexing function for `BALBOA` so that for any `BALBOA` object `ba` and index $i <$ `ba.size`, `ba(i)` gives element number i in the stored order (which you may assume is sorted order). The function is standardly called `apply(i:Int)` in Scala. You may assume the linked list is doubly linked with fields `next` and `prev` as in `BALBOADLL`, though you may not need to use `prev`. Assuming that each array has size $m = \sqrt{n}$ and there are $r = \sqrt{n}$ arrays, and that the linked list implements `size` in $O(1)$ time, what is the running time of your method?

XOR

II. Give code for a function

```
def merge(ba1: BALBOA[A], ba2: BALBOA[A]): BALBOA[A] = { ... }
```

which outputs a merged `BALBOA[A]` object `ba3` such that `ba3.size = ba1.size + ba2.size`. Use iterators and the iterator version(s) of `insert` (or you may use the methods called `+=`, `++=`, and/or `append` in Scala), in a way that they could work for any container class in the "ISR" repository, not just `BALBOA`. You should assume that there is a comparator `keyComp` for the client type `A`, but you may not assume that the two `BALBOA` objects have the same value of their capacity parameter.

END OF EXAM