CSE250, Spring 2014 Final Exam May 14, 2014

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 tree after `bed`, after `ebb`, and then after `did`—if you can! One picture of the BST is enough, while the `FlexArray` 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) A red-black tree.

(f) A `FlexArray fa` 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 using the call `fa.insert(fa.begin()++,x);` 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 `FlexArray` object `fa` and execute `fa.insert(fa.size(), x);` in a loop for n different items x, using the indexing version of `insert` from Project 1. Assume the nodes have capacity roughly $c = \sqrt{n}$. Is the total running time $O(n)$? Justify your answer.

(b) Suppose we begin with an empty `FlexArray` object `fa` and execute `fa.insert(fa.end(), x);` in a loop for n different items x, using the iterator version of `insert` from Project 1. Assume the nodes 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 node vector, meaning it operates the way the text describes for `vector` in Chapter 4: When a node is allocated, its `elements` vector is initialized to size c not size zero, and when the new element x is inserted at the end, an assignment like `elements->at(rearSpace++)` is executed.

(d) Now suppose we insert new elements at the place where they would go to keep the `FlexArray` 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. *(questions continue overleaf)*

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

(f) Why was `FlexArray` a better choice for the word-chains application—specifically the part allowing new words to be put in anywhere, not just the ends of a chain—than it would have been for the movie-base or user-base container classes on Project 2?

(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 `vec` stands for a vector, `deq` for a deque, `dlist` for a doubly-linked list (unsorted), `fa` for a "`FlexArray`" data structure with $c \simeq \sqrt{n}$, `bst` for a BST—i.e. a general binary search tree, `rbt` for a red-black 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 STL-compliant interface as on Project 1. Justifications* are not required, but might help for partial credit. "Amortized" and "usually" mean as on Assignment 8.

1. For a BST iterator `itr`, the call `erase(itr);`
2. For a red-black tree iterator `itr`, the call `erase(itr);`
3. `fa.erase(fa.begin());`
4. `vec.insert(vec.begin(),item);`
5. `dlist.erase(itr);`
6. For a `set` data structure `s`, the call `s.find(item);`
7. For two `FlexArray` iterators `itr1` and `itr2`, the test `itr1 == itr2`;
8. For a deque `deq`, n consecutive calls to `popRear();`
9. Given a red-black tree `rbt` with n elements and a BST `bst` with only $n/\log_2 n$ elements, copying the latter from `bst` into `rbt`.
10. Given two FlexArray objects `fa1` and `fa2`, with the same capacity c, creating a new `FlexArray` as the union of the two.

(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 `FlexArray` have any advantages here? What if many users closed their accounts and got erased?

(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 C++ Standard Template Library 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 `void addBid(const Bid& bid)` of the `User` class.

(f) Which method(s) in part (e) should be `const`? [Exam extra-credit (4 pts.): how might one be "legally `const`" without being "morally const"?]

(g) Give an algorithm for computing the top-k list. A pseudocode sketch is fine—you may name some C++ functions such as `sort` or `make_heap` but need not write exact C++ 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.)

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

(5) (30 pts.)

Do One of the following two programming tasks, *your choice*. One is "low-level" with pointers and indices, the other "high-level" with iterators.

I. Inside the `FlexArray` class, revise the body of `at(size_t i)` so that it loops backwards from the end, rather than forward from the first node as on Assignment 6. Here you need to assume that the `ChunkNode<T>` nodes are in a doubly-linked list, with `prev` as well as `next` fields. The effect is that now the rear item and elements near it can be retrieved in $O(1)$ time. Then say how you could combine it with the project version to get a body for `at` that would run in $O(1)$ time for the elements at either end, thus meeting requirements of the C++ `deque` class in particular (which is what `FlexArray` emulates).

XOR

II. Give code for a function

```
template <typename T>
void merge(FlexArray<T>& source, FlexArray<T>& target) { ... }
```

which empties all the items out of `source` and appends them onto `target`. Use iterators and the iterator versions of `insert` and `erase`; note that your code might apply to any STL-compliant container class with the same interface of public methods, not just `FlexArray`. Also note that unlike Problem (3), item 10., you may not assume that the two `FlexArray` objects have the same value of their capacity parameter.

End of Exam