CS166 Midterm Exam

[Your name goes here]

Problem One: Range Minimum Queries

The ***range distinct query problem*** (or ***RDQ***) is the following problem: preprocess an array A to efficiently support queries of the form "given a subarray of the array A, how many distinct elements are there in that subarray?" For example, consider this array shown below:

Value	2	7	1	8	2	8	1	8	2
Index	0	1	2	3	4	5	6	7	8

Here, performing an RDQ over the range $[2, 5]$ would return 3, as there are three distinct values in the range (namely, 1, 2, and 8). Performing an RDQ over the range $[5, 7]$ would return 2 because there are two distinct values in the range (1 and 8), performing an RDQ over the range $[0, 0]$ would return 1, and performing an RDQ over the range $[0, 8]$ would return 4 (with the distinct values being 1, 2, 7, and 8).

Design a data structure for RDQ that has a (possibly expected) preprocessing time of $O(n)$ and a (worst-case) query time of $O(z)$, where z is the number of distinct elements in the queried subarray.

Some hints:

- Tag each element with the index of the previous occurrence of that element in the overall array (or with -1 if it's the first copy of that element).
- To count the number of distinct elements in a range, search for the first copy of each distinct element within that range. (You don't care about the second, third, etc. copy of each element.)
- Use a divide-and-conquer strategy.

And, of course, given the title of this question, use RMQ to combine the three above ideas together.

Solution: Your solution goes here!

Problem Two: String Data Structures

The ***label*** of a node in a suffix tree is the string formed by tracing the path from the root of the tree to that node. Prove that if the suffix tree for some nonempty string T contains a node with label w, then the suffix tree for T also contains a node with label x for each suffix x of w. (Some algorithms on suffix trees rely on this fact and associate each node in the suffix tree with a pointer to its longest suffix.)

Solution: Your solution goes here!

Problem Three: Balanced Trees

Suppose we will be performing a series of lookups in a binary search tree where the choice of which key to look up next is sampled independently from a fixed probability distribution. More specifically, let the keys in the tree be $x_1 < x_2 < \cdots < x_n$ and let the probability of looking up key x_i be p_i. You can assume that the access probabilities are all positive and that every search is for a key that is present in the tree.

As a reminder, a binary search tree has the ***entropy property*** if the expected cost of a lookup is $O(1+H)$, where H is the Shannon entropy of the probability distribution from which lookups are sampled. (See the lecture on splay trees for a refresher on Shannon entropy.)

In lecture, we claimed that weight-balanced trees have the entropy property, assuming that each key's associated weight is its access probability. Prove this.

Solution: Your solution goes here!

Problem Four: Amortization

Suppose you have a sorted sequence of keys $x_1 < x_2 < \cdots < x_n$ from which you'd like to construct a B-tree of order b. To do so, you could insert each key into an empty tree one at a time. If you were to do this, though, you'd find that you were doing a lot of unnecessary work traversing the tree top-down, since each search is guaranteed to terminate in the rightmost leaf. (Do you see why?) This top-down searching would mean that the runtime of inserting the keys would be $\Omega(n \log_b n)$.

There's a much faster way to build a B-tree from scratch. Maintain a pointer to the rightmost leaf node in the B-tree, and annotate each node in the B-tree with the number of keys it contains. Then, insert each key, one at a time and in sorted order, using the following procedure:

1. Place the key in the next free slot in the rightmost leaf.
2. If the rightmost leaf overflows, split that leaf and propagate any further splits higher up in the tree using the usual node-splitting procedure.

Prove that the amortized cost of inserting each element this way is $O(1)$ using either the banker's method or the potential method. This shows that the cost of building a B-tree from a sorted sequence using this algorithm is $O(n)$, independent of the order of the B-tree.

Although B-trees are typically stored on-disk and analyzed in terms of disk reads and writes, for the purposes of this problem please analyze this algorithm in terms of the total number of operations performed, not the number of block transfers.

Solution: Your solution goes here!

Problem Five: Randomization

You have a randomized data structure D that estimates some (unknown) quantity A. Let's denote the estimate returned by the data structure as $\hat{D}$. Let's further suppose that $\mathrm{E}\left[\hat{D}\right] = A$ (the estimator is unbiased) and also that $\mathrm{Var}\left[\hat{D}\right] = A^2$ (the variance of the estimate is fairly high).

Using D as a black box, design an improved data structure that estimates A with tunable accuracy and confidence parameters. Specifically, your data structure should be parameterized over two user-provided values $\varepsilon \in (0, 1)$ and $\delta \in (0, 1)$ and should satisfy the following guarantees:

- $\Pr\left[|\hat{B} - A| > \varepsilon A\right] < \delta$, where $\hat{B}$ is the estimate returned by your data structure;
- the space usage is at most a factor of $O(\varepsilon^{-2} \log \delta^{-1})$ larger than the space usage for D; and
- the runtime of every operation on your data structure is at most a factor of $O(\varepsilon^{-2} \log \delta^{-1})$ larger than the runtime cost of the corresponding operation on D.

As a hint, you can reduce the variance of an unbiased estimator by running lots of copies of that estimator in parallel and taking the average.

Solution: Your solution goes here!