Fundamental Algorithms, Section 003
Homework 6, Fall 22. Due date: Wednesday, October 19, 4:55pm on Gradescope

1. This problem continues the question on the additional problem handout.

Suppose we are maintaining a collection of DNA strings. We wish to support one more operation: Reverse a string: i.e. $u = u_1 u_2 \dots u_n$ is replaced by $u_n, \dots u_2 u_1$.

Show how to support this operation and the previous operations so they run in time $O(\log n)$, where n is the sum of the lengths of the strings involved in an operation.

Present your algorithm so it is clear why it works and justify the runtime briefly. It suffices to explain how to modify the solution to the additional problem. You may state known results about data structures.

2. Let S be a set of n items which have $k < n$ distinct values. Show how to sort S in expected $O(n + k \log k)$ time.

3. Suppose you are given a set S of items with two attributes, value and size, which are both positive integer values in the range $(0, m-1)$. Give a data structure to store these items so that the following operations can be supported in the stated times.
i. Insertion, in expected $O(1)$ time.
ii. Deletion, in expected $O(1)$ time.
iii. Deletion of all items of size s in expected time proportional to the number of these items.
iv. Report all items of value v in expected time proportional to the number of these items.

Present your solution so it is clear why it works and justify the runtime briefly.

4. In some applications of hashing it is important to have zero collisions, because testing equality is expensive. e.g. imagine storing a database of known chemicals, and wanting to identify wherether an unknown chemical is in the database, where the hashing is based on various features of the chemicals, but the equality test is based on X-ray imaging (or some other expensive process).

We suppose that for each integer s, there is a family $\mathcal{H}_s$ of hash functions that maps to tables of size s, and you can select a hash function uniformly at random from $\mathcal{H}_s$ in $O(1)$ time.

Suppose n items are being stored in a table of size $2n$. We guarantee that a random $h \in \mathcal{H}_{2n}$ will have at most n pairs of collisions with probability at least $\frac{1}{2}$; for example if 4 items are hashed to the same location they will cause $\frac{1}{2} \cdot 3 \cdot 4 = 6$ pairs of collisions.

Also, suppose that m items are being stored in a table of size $m(m-1)$. We guarantee that a random $h \in \mathcal{H}_{m(m-1)}$ will have zero collisions with probability at least $\frac{1}{2}$.

We will build a two-level hash table. The first level uses a hash function h_1 into a table H_1 of size $2n$. Then, for each location $H_1[i]$ to which m_i items are hashed, we have a second hash function $h_{2,i}$ and hash table $H_{2,i}$ of size $2m_i(m_i - 1)$.

The 2-level hash table will be used to store a fixed set of n items and then to answer membership queries with respect to this set.

Show how to build such a two-level hash table so that there are no collisions at the second level and so that it uses $O(n)$ space in the worst case. Show that it can be built in expected $O(n)$ time.
Hint. You can draw a hash function uniformly at random from $\mathcal{H}_s$ for a desired value of

s, and if it is does not have the properties you desire, then draw another function again uniformly at random.

Challenge problem. Do not submit.
This is a continuation of Problem 1 on the additional problem set.

One can also support copying when the data structure is a 2–3 tree. The way to do this is to view each operation on a string as creating a new 2–3 tree. The new 2–3 tree will share any unchanged subtrees with the old 2–3 tree, but the new tree will replace all the nodes on the search path wtih new nodes, as well as any other nodes of the old tree that need to change. The overall structure is no longer a tree, but a more complex structure of nodes and pointers (it's actually a directed acyclic graph). However, it contains every 2–3 tree that occurs as a result of the series of operations. Argue that the edges descending from each 2–3 tree created in this way actually specify a tree. Conclude that each copy and paste needs to change only $O(\log n)$ nodes.

This idea of copying changes is how one can efficiently support version histories for any type of text file, e.g. programs.