Homework 6, Solution set

1. Again, each string is stored in a 2–3 tree. To enable string reversals to be easily implemented we keep a direction bit for each subtree, stored at the root of the subtree. If the bit has value 1 the substring stored in the subtree is reversed. Stated recursively, the string represented by a 2–3 tree is the following: if the direction bit is 0, it is the concatenation of the strings represented by the root's subtrees, in left to right order; if the direction bit is 1, it is the concatenation of the reversal of the strings represented by the root's subtrees, in right to left order. The string represented by a leaf node is the one character it stores.

To reverse a string, we flip the bit at the root of the subtree. This takes $O(1)$ time. Also, we define the following clean(v) operation for any node v. If the flip bit at v is 0, the operation does nothing; if it has value 1, it is reset to 0, the bits at its children are flipped, and the order of its children is reversed. This operation takes $O(1)$ time. Whenever we perform an operation on the 2–3 tree, as each node v on the search path is reached, we perform a clean(v) step; likewise, whenever we want to combine two nodes v and w, or have them repartition their children, we perform clean(v) and clean(w) operations first. This ensures that the standard 2–3 tree operations (search, insert, delete) can be performed as before. Performing the clean( ) operations only increases the runtimes by a constant factor, as we still traverse length $O(\log n)$ paths for each operation, and the work done per node is still constant.

As before, we keep a count of the number of characters in each subtree. These counts are stored at the parent of a subtree. We implement the cut and paste as before, enhanced by cleaning each node that is touched by these operations. Again, these operations will still take $O(\log n)$ time, as these operations each touch $O(\log n)$ nodes.

2. We begin by hashing the items into a table of size n in order to identify duplicates, and more importantly to identify the k non-duplicate items. This takes expected $O(n)$ time. We then sort the non-duplicate items using quick sort. This takes expected $O(k \log k)$ time. (Alternatively, use merge sort.)

We perform hashing with chaining. In addition, for each distinct item, we keep a separate doubly linked list of the copies of this item. So each entry of the hash table is a list of lists. When we insert an item into the hash table, we either find that it is a new item, in which case we add it to the list for that entry in the hash table, or we find it is a duplicate, in which case we add it to the list for that value in $O(1)$ time (either we add the item to the front of this list, or we keep a pointer to the rear of the list and add it to the rear). Each hash takes expected $O(1)$ time, as there is only one copy of each item in the list at its hash location, and there are at expected $O(1)$ distinct items at each location. So over all n items, this is expected $O(n)$ time.

To obtain the non-duplicate items, we simply record them as they are added to the hash table.

Finally, to report the sorted set of n items, in turn, for each non-duplicate, output the items in its list of duplicates. This takes $O(n)$ time.

Thus, overall, the algorithm runs in expected time $O(n + k \log k)$.

3. The idea is to use three hash functions, one hashing on both attributes, one hashing the value attribute, and one hashing the size attribute. Let us call them h_b, h_s, and h_v, respectively, and suppose they use arrays H_b, H_s, and H_v, respectively. As in the previous question, in table H_s, for each size s inserted into the table, we keep a doubly linked list of the items with that size. So there is just one representative of each size actually in this hash table. Similarly, in table H_v, for each value v inserted into the table, we keep a linked list of the items with that value.

Inserting an element $e = (\overline{v}, \overline{s})$
We insert the item in H_b (at location $h_b(\overline{v}, \overline{s})$), in H_v (at location $h_v(\overline{v})$), and in H_s (at location $h_s(\overline{s})$). In addition, we keep pointers with each copy of the item so that we can navigate between the 3 copies in $O(1)$ time. As each insertion takes expected $O(1)$ time, the overall operation also takes expected $O(1)$ time.

Deleting an element $e = (\overline{v}, \overline{s})$
We delete the item from H_b, and use the pointers to navigate to the copies in H_v and H_s and delete these copies from the linked lists for the value $\overline{v}$ (in H_v) and for the size $\overline{s}$ (in H_s). Note directly hashing into H_v and then searching the list for value $\overline{v}$ may take too long as there may be many items with this value. Likewise, directly hashing into H_s may be too expensive. The deletion on H_b takes expected $O(1)$ time, and the pointer navigation to remove the copies in H_v and H_s takes a further $O(1)$ time.

Deleting all items of size $\overline{s}$
Compute $h_s(\overline{s})$ to identify where the items with size $\overline{s}$ are stored in H_s. We then remove all the items from this location (location $H_s[h_s(\overline{s})]$). For each item $e = (\overline{v}, \overline{s})$ deleted from H_s, by following pointers, we also delete e from H_b and H_v. It takes $O(1)$ time to compute $h_s(\overline{s})$. We then spend $O(1)$ work for each item stored at $H[h_s(\overline{s})]$ in deleting it from H_b and H_v. Therefore, the total expected work is linear in the number of items of size $\overline{s}$.

Reporting all items of value $\overline{v}$.
Compute $h_v(\overline{v})$ to identify where the items with value v are stored in H_v. Then simply output the list of items at location $H_v[h_v(\overline{v})]$. This takes $O(1)$ time to compute $h_v(v)$ and $O(1)$ time per item at this location in H_v; so the runtime time is linear in the number of items with value $\overline{v}$.

4. We select $h \in \mathcal{H}$ uniformly at random and hash our set of n items to a table of size $2n$. If there are at most n collisions we continue to the next step. Otherwise, we draw another $h \in \mathcal{H}$, repeating until we obtain an h which causes at most n collisions. As we have at least a $\frac{1}{2}$ probability of succeeding at each step, this takes expected time $cn(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots) \leq 2cn$, where cn is the cost of testing a single h. Note that to count the number of collisions, it suffices to keep a running total. If it reaches $n+1$, we simply stop the test of that h. Let h_1 be the function chosen by this process.

Let m_i be the number of items in $H_1[i]$. Note that the number of pairs of collisions in this location is $\frac{1}{2}m_i(m_i - 1)$. Thus the total number of collisions, $\sum_{i=0}^{2n-1} \frac{1}{2}m_i(m_i - 1) \leq n$, by the construction in the previous paragraph.

For the second level of the hash table, the construction proceeds as follows. If $m_i > 1$, we draw a hash function $h_{2,i}$ from $\mathcal{H}$ which maps to a table of size $m_i(m_i - 1)$. Here, we test if $h_{2,i}$ causes zero collisions, and if not keep drawing until we obtain a zero-collision hash function. For each hash function we test, this will take $O(m_i^2)$ time to initialize the hash table to empty and a further $O(m_i)$ time to test if it produces zero collisions. As the probability of success for each hash function we test is at least $\frac{1}{2}$, the process of finding a hash function that causes zero collisions will take expected time $O(m_i^2)$ when $m_i > 1$. While if $m_i = 1$ we create a size 1 level 2 hash table, and the corresponding level 2 hash function is the identity map. Thus the total expected time for this second step is $O(n + \sum_i m_i^2) = O(n)$, as $\frac{1}{2}\sum_i m_i(m_i - 1) \leq n$, and hence $\sum_i m_i^2 = O(n)$ also.

It follows that the overall expected runtime is $O(n)$.

The space used by this two-level hash table is $O(n)$ for H_1 and $O(\sum_{i=1}^{n} m_i(m_i - 1))$ for the second level tables when $m_i > 1$, and $O(n)$ for the second level tables when $m_i = 1$. But this sum is the number of pairs of collisions, and the first level hash function was chosen so that this is bounded by n. Thus the total space is $O(n)$.