CSCI2100: Regular Exercise Set 7

Prepared by Yufei Tao

Problems marked with an asterisk may be difficult.

Problem 1. Let S_1 and S_2 be two sets of integers (they are not necessarily disjoint). We know that $|S_1| = |S_2| = n$ (i.e., each set has n integers). Design an algorithm to report the *distinct* integers in $S_1 \cup S_2$ using $O(n)$ expected time. For example, if $S_1 = \{1, 5, 6, 9, 10\}$ and $S_2 = \{5, 7, 10, 13, 15\}$, you should output: 1, 5, 6, 7, 9, 10, 13, 15.

Solution. First, output everything in S_1. Then, create a hash table on S_1 in $O(|S_1|)$ time. For every value $x \in S_2$, probe the hash table to see if $x \in S_1$. If not, output x. Each probe takes $O(1)$ expected time. Hence, the total cost of all the probes is $O(|S_2|)$ expected. The overall cost is therefore $O(n)$ expected.

Problem 2 (No Single Hash Function Works for All Sets). Let U and m be integers satisfying $U \geq m^2$. Fix a hash function h from $[U]$ to $[m]$, where $[x]$ represents the set of integers $\{1, 2, ..., x\}$. Prove that there must be a set $S \subseteq [U]$ such that (i) $|S| = m$, and (ii) h maps all the elements of S to the same hash value.

Solution. For each $i \in [m]$, define $S_i = \{x \in [U] \mid h(x) = i\}$. Since $\sum_{i=1}^{m} |S_i| = U \geq m^2$, there is at least one $j \in [m]$ such that $|S_j| \geq U/m \geq m$. Construct a set S to include m arbitrary distinct elements from S_j. This S fulfills our purposes.

Problem 3*. Let S be a multi-set of n integers. Define the *frequency* of an integer x as the number of occurrences of x in S. Design an algorithm to produce an array that sorts the *distinct* integers in S by frequency. Your algorithm must terminate in $O(n)$ expected time. For example, suppose that $S = \{75, 123, 65, 75, 9, 9, 65, 9, 93\}$. Then you should output $(123, 93, 65, 75, 9)$. Note that if two integers have the same frequency, their relative ordering is unimportant. For example, $(93, 123, 75, 65, 9)$ is another legal output.

Solution. We can collect the set T of distinct integers in S by hashing as follows. For every integer $x \in S$, check whether the hash table has already contained a copy of x. This takes $O(1)$ in expectation. If so, ignore x; otherwise, insert x into the hash table in $O(1)$ time. The collection requires $O(n)$ time overall.

We can then obtain the frequency of every distinct integer as follows. For each integer $x \in S$, find its copy in the hash table, and increase the counter of the copy by 1 (the counter initially set to 0). This takes $O(1)$ time per integer, and hence, $O(n)$ time overall.

Now we simply sort all the distinct integers by frequency. Note that the frequencies are in the domain from 1 to n. Hence, counting sort gets this done in $O(n)$ time.

Problem 4*. Let S be a set of n key-value pairs of the form (k, v), where k is the key and v is the value. Preprocess S into a data structure so that the following queries can be answered efficiently. Given a pair (q_k, q_v), a query

- Returns nothing if S contains no pair with key q_k;

- Otherwise, it returns the number of pairs $(k, v) \in S$ such that $k = q_k$ and $v \le q_v$.

Define the *frequency* of a key k as the number of pairs in S with key k. Define f as the maximum frequency of all keys. Your structure must use $O(n)$ space, and answer a query in $O(\log f)$ expected time. Furthermore, it must be possible to construct the structure $O(n \log f)$ time.

For example, suppose that $S = \{(75, 35), (123, 6), (65, 32), (75, 22), (9, 1), (9, 10), (65, 74), (9, 8), (93, 23)\}$. Then, given $(63, 33)$, the query returns nothing. Given $(65, 33)$, the query returns 1. Given $(65, 2)$, the query returns 0. In this example, $f = 3$.

Solution. Collect the set T of distinct keys in S, and obtain their frequencies in $O(n)$ time (see the solution of Problem 2). Create a hash table on T in $O(n)$ time. For every key $k \in T$, create an array A_k whose length is equal to the frequency of k. Store in A_k all the values v such that (k, v) is a pair in S. Sort A_k in ascending order. The sorting takes $O(|A_k| \log |A_k|) = O(|A_k| \log f)$ time. Store the beginning address of A_k at the copy of k in the hash table. The overall construction time is $O(\sum_k |A_k| \log f) = O(n \log f)$. The space consumption is obviously $O(n)$.

To answer a query (q_k, q_v), first probe the hash table to see if $q_k \in T$. If not, terminate the algorithm. Otherwise, perform binary search in A_{q_k} in $O(\log f)$ time. The overall query time is $O(1)$ expected plus $O(\log f)$ worst case, which is $O(\log f)$ expected.

Problem 5 (Dynamic Hashing).** Consider the following *dynamic dictionary search* problem. Let S be a dynamic set of integers. At the beginning, S is empty. We want to support the following operations:

- `Insert`(e): Adds an integer e to S.
- `Delete`(e): Removes an integer e from S.
- `Query`(q): Determines whether q belongs to the current set.

Design a data structure with the following guarantees:

- At all times, the space consumption is $O(|S|)$, i.e., linear to the number of elements currently in S.
- For any sequence of n operations (each being an `insert`, `delete`, or `query`), your algorithm must use $O(n)$ expected time in total.

Solution. If $|S| \le 4$, we simply store the entire $|S|$ in an array of length 4. If $|S| > 4$, we will maintain the hash function h whose output domain is $[m]$, with m being a power of 2 and satisfying $|S| \le m \le 4|S|$. Accordingly, we also maintain a hash table T computed using h. `Insert`(e) is processed by inserting e into the linked list in T corresponding to the hash value $h(e)$. Similarly, `delete`(e) is processed by scanning the entire linked list of $h(e)$, and removing e from there.

If after an insertion $|S|$ reaches m, we double m, and reconstruct the hash table by randomly selecting a new hash function h whose output domain is $[m]$ (note that the domain size has doubled). If after a deletion $|S|$ equals $m/4$, we halve m, and reconstruct the hash table by randomly selecting a new hash function h whose output domain is $[m]$. The amortized insertion/deletion cost is $O(1)$ by the same analysis we did for dynamic arrays.

A query is answered in the same way as discussed in the class.

An insertion obviously is handled in $O(1)$ time. The expected running time of a deletion is the same as that of a query, which is $O(1)$ when we choose h from universal family explained in the class. The space consumption is $O(|S|)$ at all times.