CS161 Summer 2022 Problem Session 3

Problem 1: Min-Heap Miscellany

(a) In Problem 4(c) on HW 3, when talking about $O(\log n)$-time deletion from a heap, we assume we have a pointer to the element being deleted. Suppose that we don't have a pointer – what is the running time of the **Delete**(x) operation then? (Can you propose a case that illustrates this?)

(b) Consider a very large min-heap in which every element is distinct. What are the maximum and minimum depths that the fifth-smallest element can be at? (Recall that we consider the depth of the root to be 0.)

(c) Suppose we have a list of n elements and we want to put them all into a new min-heap. (This operation is sometimes called **Heapify**.) What are the best and worst-case running times of this operation? Here, by "best" and "worst", we refer to the input, not to any randomness (since this data structure is fully deterministic).

Problem 2: BST runtimes

Self-balancing binary search trees typically support insertion, search, and deletion in $O(\log n)$ time, and inorder traversal in $O(n)$ time.

(a) Is it possible to design a self-balancing BST with $O(1)$ insertion? (The costs of the other three operations can be whatever you'd like.)

(b) Is it possible to design a self-balancing BST with $O(1)$ insertion and $O(n)$ inorder traversal? (The costs of the other two operations can be whatever you'd like.)

Problem 3: Hashception

No matter how we implement a hash table, performance will degrade as the table gets more and more "full". In a hash table with chaining, the chains in the individual buckets will become very long and hard to search, taking time linear in the length of the chain.

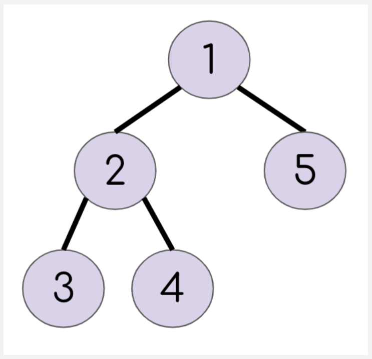
One possible solution is to use, e.g., self-balancing BSTs instead of chains. But Indy doesn't like this idea, since it would mean that the runtime guarantee on **Search**(x) would no longer be $O(1)$, and investors would panic. He instead proposes to solve this problem by putting another hash table in each bucket of the original hash table. What do you think of this strategy?

Solutions to Problem 1

(a) If we don't have a pointer to the element we want to delete, we have to search for it. And unlike BSTs, heaps are not easily searchable: we might have to look through $O(n)$ elements to find an arbitrary value. As an extreme case, imagine a binary min-heap in which every value is 1 except for a single 2. The 2 could be at any of the leaves, and there can be just over $\frac{n}{2} = O(n)$ leaves.

(b) The fifth-smallest element can be no deeper than depth 4, since it must be larger than its parent, which must be larger than *its* parent, and so on. Because there are only four elements that are smaller, this chain can only include four other nodes. (This is still a little discouraging, though! The k-th smallest element can still be $O(k)$ levels deep, rather than, e.g., $O(\log k)$.

However, it is possible for the fifth-smallest element to be a direct child of the root, as in the following example:



(c) Although insertion into a heap is $O(\log n)$ in general, consider a case in which the list we are heapifying is already sorted. As a concrete example, let the list be $[1, 2, ..., n]$. Then the heap never has to do any swaps, since elements are always inserted into the next available slot (at the current bottom level of the heap), and a newly inserted element is never smaller than its parent. So the best-case running time is actually $O(n)$, since each such insertion takes constant time.

The worst case occurs when the list we are heapifying is sorted in reverse order. Every time an element is inserted (into the bottom level), it has to be swapped all the way to the root because it is the new minimum. This takes $O(\log n')$ time each time, where n' is the number of elements in the heap at the time of the insertion. However, we see that the later steps dominate the earlier ones; as in the explanation to (a), the heap can have just over half of its nodes in the leaves. So the overall worst-case running time is $O(n \log n)$.

Solutions to Problem 2

(a) Yes – this is a little silly, but we can just stick inserted elements onto the end of, e.g., a linked list, and then whenever it comes time to do a search, deletion, or traversal, we perform all these insertion(s) and then do the operation we had planned to do. This can dramatically raise the running time of that operation (what if we have built up a million elements that all need to be inserted?), but insertion itself is still technically $O(1)$.

(b) If we could achieve these runtime guarantees, we could sort a list of n comparable items as follows:

- Insert every item into the tree, taking $n \cdot O(1) = O(n)$ time.
- Perform an inorder traversal of the tree, taking $O(n)$ time. This produces the elements in sorted order.

We proved that it is impossible to sort in $o(n \log n)$ time, so we also know that no such BST can exist.

Solution to Problem 3

A hash table of hash tables is not a completely ridiculous idea, assuming that the inner hash tables at least use a different hash function from the outer hash table. (Otherwise, an inner hash table in a bucket wouldn't spread out its values at all!)

However, if Indy is (e.g.) creating a hash table with b buckets, each of which contains its own hash table with b buckets, it would perhaps be simpler for him to just use a single bigger hash table with b^2 buckets?

On the other hand, that strategy would rely on the hash function being very good. In practice, two separate hash functions might be more resistant to pathological data sets. This is one of those situations where we'd have to try both and see!

Problem 4: ThergeSort revisited

Recall that in Problem Session 1, we tried to generalize MergeSort to split the list into k sublists, for $k > 2$. We ran into a difficulty in trying to implement the Merge step in linear time – the time ended up depending on k as well as on n, the total number of elements being merged. Our naive solution was $O(nk)$ since we had to keep checking the pointers of all k sublist to find the smallest element to add to our merged list.

We can't fully get around the dependency on k, but at this point in the course, we can do better than $O(nk)$. Explain how.

Problem 5: One-Way Hashes

We have talked about hash functions in a one-way sense. In general, it does not make sense to think about going the other way – given a hashed value, determine the original value that produced that hash. This is because the universe $\mathcal{U}$ is generally much larger than the number of buckets n, so it would not be possible to determine that value uniquely anyway.

However, can you think of a situation in which we would want a hash function that can't be reverse-engineered at all? e.g., a situation where, given a hashed value, even being able to narrow down a subset of the universe that could have produced it would be highly undesirable...

Problem 6: LRU Caches

(This problem appeared in CS9, but it's good practice here if you haven't seen it!)

Suppose you have a large but slow collection of n key-value pairs. You receive a sequence of requests where each request is a key, and in response you should send the value for that key.

As an optimization, you add a cache. A *Least Recently Used* cache stores the k distinct most recently requested key-value pairs. If a requested key is in the cache, its value can be returned. Otherwise, the least recently requested key-value pair is evicted (hence the name "LRU"), and the newly requested key-value pair is added. Then the value can be returned.

Design an efficient implementation of this data structure. (Assume n can be quite large, and k is relatively smaller.)

Problem 7: UHF

Consider a universe $\mathcal{U}$ consisting of the four strings $00, 01, 10, 11$. You want to design a hash function from this universe to two buckets 0 and 1. Design the smallest universal hash family you can for this scenario.

Solution to Problem 4

We can use a (min-heap-based) priority queue to repeatedly find the next smallest value to add to our merged list. The Merge step now looks like this:

- Input: k sorted sublists, labeled $1, ..., k$, with a combined total of n elements
- Create one "index" variable for each sublist. Initialize each of these indices $p_1, ..., p_k$ to 0.
- Initialize an empty (min-heap-based) priority queue. For each sublist L_i, put the pair $(L_i[0], i)$ into the priority queue.
- Initialize an empty list F to hold the final anwer.
- Repeat the following:
  - Delete the minimum pair from the priority queue.
  - Append the first entry of that pair to F.
  - Let j be the second entry of that pair, i.e., the number of the sublist from which the element that we just appended came. Increment p_j by 1.
  - If p_j equals the length of sublist L_j – i.e, there are no more elements from that sublist to insert – do nothing. Otherwise, insert $(L_j[p_j], j)$ into the priority queue.

Notice that this priority queue never has more than k elements in it, since after the initial batch of insertions, we must delete an element before inserting another element. Therefore, deletion and insertion both take $O(\log k)$ time, and processing all n elements takes $O(n \log k)$ time. When k is relatively small compared to n, this isn't so bad, and it's definitely better than our previous "score" of $O(nk)$.

Solution to Problem 5

One real-world application of this is in password storage. It is dangerous to store raw "plaintext" passwords on a server – what if someone gets access to that server? So it is more common to instead store hashes of passwords. When a user inputs their password, it is hashed and then checked against the stored hash. (This means that a user might gain access with the wrong password that just happens to hash to the same thing, but we're talking about an enormous hash space here, not like a hash table with a relatively small number of buckets.)

However, what if someone gets access to the list of stored hashes? Then, given a password that they think a user might be using, they can check what that candidate password hashes to and see if that matches the stored hash. Since many people pick their passwords from a relatively small subset of the universe of all possible passwords, this is a potentially devastatingly effective strategy.

There are countermeasures to this ("salts" and sometimes even "pepper"), and countermeasures to those countermeasures ("rainbow tables")... if you're interested, consider taking CS255!

Solution to Problem 6

The answer to this problem is explained in the slides here: `https://web.stanford.edu/class/cs9/slides/w8-2_problem_solving.pdf`. tl;dr the solution involves a hash table and a doubly linked list.

Solution to Problem 7

There is no way to use just a single hash function here, since there will necessarily be two elements that it hashes to the same bucket. Then the probability of those two elements being hashed to the same bucket, over the (trivial) randomness of our choice of (only one) hash function from the family, is 1, which is too high – since there are 2 buckets, we need the probability to be $\leq \frac{1}{2}$.

So we need a family of at least two hash functions. One option that I found after only a little bit of experimentation was the following:

- h_1: choose the first bit of the string
- h_2: choose the second bit of the string

Warning: this does *not* extend nicely to larger strings! Methods like "choose this digit of..." are usually doomed. But this actually works here:

- h_1 hashes 00 and 01 to bucket 0, and 10 and 11 to bucket 1.
- h_2 hashes 00 and 10 to bucket 0, and 01 and 11 to bucket 1.

Pick any of the six possible (unordered) pairs of possible inputs $(00, 01), (00, 10), (00, 11), (01, 10), (01, 11), (10, 11)$. You can see that for each of them, the probability of a collision is actually $\leq \frac{1}{2}$ – by inspection[a], in no case will both of the two possible hash functions hash both inputs to the same bucket.

[a]This is how a mathematician says "you do the work!"

Problem 8: Painted Penguins

A large flock of T painted penguins will be waddling past the Stanford campus next week as part of their annual migration from Monterey Bay Aquarium to the Sausalito Cetacean Institute. Painted Penguins can come in a huge number of colors -— say, M colors – but each flock only has m colors represented, where $m < T$. The penguins will waddle by one at a time, and after they have waddled by they won't come back again. You'd like to design a randomized data structure to keep track of the penguin colors so that, after all the penguins have gone, you'll be able to answer queries about what colors of penguins appeared in the flock; you'd like your answers to these queries to probably be correct.

For example, if $T = 7$, $M = 100000$, and $m = 3$, then a flock of T painted penguins might look like:

`seabreeze, seabreeze, indigo, ultraviolet, indigo, ultraviolet, seabreeze`

You'll see this sequence in order, and only once. After the penguins have gone, you'll be asked questions like "How many `indigo` penguins were there?" (Answer: 2), or "How many `neon orange` penguins were there?" (Answer: 0).

You know m, M and T in advance, and you have access to a universal hash family $\mathcal{H}$, so that each function $h \in \mathcal{H}$ maps the set of M colors into the set $\{0, \dots, n-1\}$, for some integer n. For example, one function $h \in \mathcal{H}$ might have $h(\texttt{seabreeze}) = 5$.

(a) Suppose that $n = 10m$. Suppose also that you only have space to store the following:

- An array B of length n, consisting of numbers in the set $\{0, \dots, T\}$.
- One function h from $\mathcal{H}$.

Use the universal hash family $\mathcal{H}$ to create a randomized data structure that fits in this space and that supports the following operations in time $O(1)$ in the worst case. You can assume that you can evaluate $h \in \mathcal{H}$ in time $O(1)$.

- `Update(color)`: Update the data structure when you see a penguin with color `color`.
- `Query(color)`: Return the number of penguins of color `color` that you have seen so far. For each query, your query should be correct with probability at least 9/10. That is, for all colors `color`,

$$\mathbb{P}\{\texttt{Query(color)} = \text{ the true number of penguins with color } \texttt{color} \text{ }\} \geq \frac{9}{10}.$$

You should present the following description of your data structure. You do not need to justify correctness, but your description must be correct (and must be detailed enough for us to know whether it is correct).

(i) Describe how the array B and the function h are initialized.

(ii) Give pseudocode for (or a clear description of) an implementation of `Query`.

(iii) Give pseudocode for (or a clear description of) an implementation of `Update`.

Hint: While you don't need to provide us with justification of correctness, you may wish to justify to yourself that your data structure is correct. In doing so, you may find it helpful to use the fact that for a finite set of events, the probability that at least one event happens is no greater than the sum of the probabilities of the individual events. This property is called the *union bound*.

(i) We initialize each entry of the array B to 0. We choose a random $h \in H$.

(ii) `Update(color):  B[h(color)] ++`

(iii) `Query(color):  return B[h(color)]`

Each of these operations takes time $O(1)$. The probability that a single `Query` option fails is the probability that any of the m (or $m-1$ other) colors which did appear collided with the color that was queried. That is, we want the following probability to be small:

$$P\{\exists x : x \neq \texttt{color}, h(x) = h(\texttt{color})\}$$

By the universal hash family property, for each color x:

$$P\{h(x) = h(\texttt{color})\} \leq \frac{1}{n}$$

Thus, by the union bound, the probability that there exists an x which appeared that collides with `color` is at most:

$$P\{\exists x : x \neq \texttt{color}, h(x) = h(\texttt{color})\} \leq m * P\{h(x) = h(\texttt{color})\} \leq \frac{m}{n} = \frac{1}{10}$$

(b) Suppose that you now have k times the space you had in part (a). That is, you can store k arrays $B_1, \ldots, B_k$ and k functions $h_1, \ldots, h_k$ from $\mathcal{H}$. Adapt your data structure from part (a) so that all operations run in time $O(k)$, and the `Query` operation is correct with probability at least $1 - \frac{1}{10^k}$.

As in part (a), a description following the outline above (except say how all arrays B_i and functions h_i are initialized) meets the requirements. Again, you do not need to justify correctness, but your description must be correct (and must be detailed enough for us to know whether it is correct).

We will basically just keep k copies of our data structure from part (a). More precisely:

(a) We initialize each entry of each of the k arrays to 0. We choose k hash functions $h_1, ..., h_k \in H$ uniformly at random and independently, with replacement.

(b) `Update(color):  for i = 1, ..., k:  B_i[h_i(color)] ++`

(c) `Query(color):  return min_{i = 1, ..., k} B_i[h_i(color)]`

Both of these operations take time $O(k)$ since they both loop over k things. To compute the success probability of `Query`, notice that this returns the correct value as long as the color `color` is isolated in *any* of the k tables. Since each of these k hash functions are independent, we have:

$$\begin{gathered} P\{\text{for all } i, \exists x : x \neq \texttt{color}, h(x) = h(\texttt{color})\} \\ = (P\{\exists x : x \neq \texttt{color}, h(x) = h(\texttt{color})\})^k \\ \leq (m * P\{h(x) = h(\texttt{color})\})^k \\ \leq (\tfrac{m}{n})^k \\ = \tfrac{1}{10^k} \end{gathered}$$

Thus, with probability at least $1 - 1/10^k$, there is at least one i such that $B_i[h_i(\texttt{color})]$ is equal to the number of times that `color` appeared, and `Query(color)` returns the right thing.