CS 161 **Problem Set 4**
Spring 2017 **Due:** May 8, 2017, 3pm

Please answer each of the following problems. Refer to the course webpage for the **collaboration policy,** as well as for **helpful advice** for how to write up your solutions.

1. (4 pts) Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that y.key is either the smallest key in T larger than x.key or the largest key in T smaller than x.key.

   [We are expecting: a detailed proof, using the definition of a binary search tree.]

2. **Open-addressing.** Read CLRS Section 11.4, pages 269-276, on open-addressing. The answers to the following questions may be implicit in the text: please answer them in your own words. Suppose that in the following, we consider an open-addressed hash table with n buckets, and a universe of size $M \geq n(n+1)$.

   (a) (1 pt) Suppose that we never delete items from an open-addressed hash table. After inserting $m \leq n$ items, what is the *worst-case*[1] time to `search`?

   [We are expecting: your bound, and an informal argument that it is correct; this includes both an argument that your bound could happen in the worst case, and an argument that the search time won't be any worse than that. A paragraph should be enough.]

   (b) (4 pts) In the discussion about deletions on page 271, we store the value Deleted instead of Nil when an item is deleted. Suppose we did not do this, and simply deleted items by returning their value to Nil. Suppose, as above, that there are currently $m \leq n$ items in the table, but that any number may have been inserted and deleted before. Give a correct algorithm to search for an element, if deletions of the form above were allowed. Above, "correct" means that `search`(x) will always return x if it is the table (and will never return anything if x is not in the table).

   What is the worst-case time for this search algorithm to find an element? Argue that any correct algorithm must have worst-case running time at least this.

   [We are expecting: a description of a correct search algorithm, and informal arguments about its worst-case running time and about why any other correct search algorithm must have worst-case running time at least this. A paragraph or two should be enough.]

   (c) (3 pts) In this question, you will be adversarially generating input for a open-addressed hash table. You insert (in order) the keys $1, 2, 3, \ldots, m$. Given that the next operation will be `search`, you get to choose a key in $\{1, \ldots, m\}$ to search for that will *maximize* the time that the operation takes. You do not know the probe sequence, so there is no way to know the exact answer to this question. Nonetheless, what key would you choose and why?

[1] As always, this means that an adversary who knows everything about the data structure gets to choose m items to insert, and then chooses an item to search for.

[We are expecting: an answer and some informal reasoning (a few sentences) about why you'd hope this would maximize the search time.]

3. In this problem, we will investigate a way to improve the worst-case performance of the chained hash tables that we saw in class. Suppose that $\mathcal{U}$ is a universe of size M, let n be an integer and that $\mathcal{H}$ is a universal family of hash functions that map $\mathcal{U}$ into buckets labeled $1, \ldots, n$. Suppose that $M \geq n^2$.

   (a) (2 pts) What is the worst-case running time for `search` in a chained hash table? That is, suppose that after h is chosen, an adversary chooses $u_1, \ldots, u_n \in \mathcal{U}$ to insert into the table, and then runs a `search` query on an item of their choosing; how long will this last `search` query take in the worst case?

   [We are expecting: One or two sentences with your answer, and a description of how this might happen.]

   (b) (4 pts) Find a way to modify of the chained hash table that we saw in class so that, if at most n items $u_1, \ldots, u_n \in \mathcal{U}$ are ever inserted into your modified data structure:

   - The expected time[2] to perform `search,insert,delete` is still $O(1)$.
   - The worst-case running time (in the sense of part (a)) to perform `search, insert, delete` operations are $O(\log(n))$.

   Your modified data structure should still involve choosing a hash function $h \in \mathcal{H}$ uniformly at random, and should still use only $O(n \log(M))$ space.

   [We are expecting: a description of the data structure and a description of how to perform `search`, `insert`, and `delete` in this data structure. We are also expecting an informal argument (a paragraph or two) about the expected and worse case running time. You do not need to write a formal proof, and you may appeal to results and arguments from class or the textbook.]

4. In this problem, we'll investigate the definition of universal hash functions. Let $\mathcal{U}$ denote a universe of size M, and let n be the number of buckets in a hash table.

   (a) (3pts) Let $\mathcal{H}$ be the family of *all* functions $h : \mathcal{U} \to \{1, \ldots, n\}$. Prove that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h randomly chosen from $\mathcal{H}$,

   $$\Pr[h(x_i) = h(x_j)] = \frac{1}{n}.$$

   This shows that $\mathcal{H}$ is a universal hash family, and moreover that we have *equality* in the definition of the universal hash family property, not just a $\leq$ relationship.

   [We are expecting: a careful and rigorous proof, though it should not need to be more than one or two paragraphs. You should be especially careful about what is random and what is not.]

[2] This is expected time in the formal sense discussed in class: for any fixed set of items $u_1, \ldots, u_n \in \mathcal{U}$, and for any sequence of L `search,insert,delete` operations only involving these elements, the expected amount of time, over the randomness used to choose the data structure, to perform all L operations is $O(L)$.

(b) (3pts) There also exist hash families such that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h randomly drawn from the family, $\Pr[h(x_i) = h(x_j)] < \frac{1}{n}$. (Notice that the inequality here is strict!) We will now explore one such family. Consider $\mathcal{H}' = \mathcal{H} \setminus \{h_1\}$, where h_1 is the function defined by

$$h_1(x_i) = 1 \qquad \text{for all} \qquad x_i \in \mathcal{U}.$$

That is, $\mathcal{H}'$ is the family of all functions from $\mathcal{U}$ to $\{1, ..., n\}$ *except* for the function h_1 which sends all elements of $\mathcal{U}$ to 1.

Prove that, for all $x_i \neq x_j$ in $\mathcal{U}$, for h drawn randomly from $\mathcal{H}'$, we have

$$\Pr[h(x_i) = h(x_j)] < \frac{1}{n}.$$

Notice that the inequality above is strict.

[We are expecting: a clear and rigorous proof. As above, this needn't be more than a paragraph of two, but you should be very careful about what is random and what is not.]

(c) (0pts) **[BONUS: Part (c) is NOT REQUIRED but might be fun to think about. It will not affect your grade but we will give you feedback if you answer it.]** Give a hash family $\mathcal{H}$ so that the collision probability

$$\max_{x \neq y \in \mathcal{U}} \Pr[h(x) = h(y)]$$

is as small as possible (and ideally prove that it is as small as possible). What is this collision probability? Above, the probability is over the choice of a uniformly random $h \in \mathcal{H}$.

[We are expecting: we aren't expecting anything, this is a bonus problem.]

5. Suppose that you are making a lightweight web browser and you have a large, fixed blacklist of w malicious websites. Let M denote the number of websites on the whole internet (this is a huge number!) and suppose for this problem that this number does not change.

Your goal is to use randomness to design a data structure that can be shipped with the browser, that is as small as possible. The data structure can be queried with a function called `isMalicious`, and has the following three properties:

- (**Small space.**) The data structure uses at most b bits, where b is an integer that is a design parameter you'd like to minimize.
- (**No false negatives.**) If a website x is on the blacklist, then `isMalicious`(x) always returns `True`.
- (**Unlikely false positives.**) Fix a website x that is *not* on the blacklist. Then with probability at least 0.99 over the randomness that you used when choosing the data structure, `isMalicious`(x) returns `False`.

Above, we emphasize that the probability works as follows: the blacklist is first fixed and *will never change.* Fix some website x, either malicious or not, and keep it in mind. Now we use randomness to generate the data structure that will ship with the browser. If x was malicious, then with probability 1, `isMalicious(x) = True`. If x was not malicious, then with probability at least 0.99, `isMalicious(x) = False`. The only randomness in the problem has to do with the choice of the data structure.

(a) (1pt) Suppose that we just ship the blacklist directly with the browser (so there is no randomness, and we always correctly classify both malicious and legitimate websites). How many bits does this require? You may assume that a single website (out of all M of them) can be represented using $\log(M)$ bits. [**We are expecting: a single sentence.**]

For parts (b) and (c): Let n be an integer. Let $\mathcal{H}$ be a collection of functions $h : \{x : x \text{ is a website }\} \to \{1, \ldots, n\}$ that map websites to one of n buckets, and suppose that $\mathcal{H}$ is a universal hash family. Notice that the size of the domain of h is M. Suppose that, as with the universal hash family we saw in class, the description of an element $h \in \mathcal{H}$ requires $d = O(\log(M))$ bits.

(b) (3pts) Below, we describe one way to randomly generate a data structure, using the universal hash family $\mathcal{H}$. The pseudocode to construct the data structure is:

```
Choose h in H at random.
Initialize an array T that stores n bits.
Initially, all entries of T are 0.
for x in the black list:
      T[h(x)] = 1.
end for
```

Then we ship T and a description of h with the browser. In order to query the data set, we define

```
function isMalicious(x):
      if T[h(x)] = 1:
            return True
      else:
            return False
      end if
end function
```

Suppose that $n = 100w$. Show that the three requirements above are met, with $b = d + 100w$.

[**We are expecting: a proof that all three properties hold: small space, no false negatives, unlikely false positives.**]

(c) (3 pts) Now consider the following variant on the data structure from the previous part. Let $\mathcal{H}$, d and n be as above. Instead of one array T, we will initialize 10 arrays $T_1, \ldots, T_{10}$, as follows:

```
Choose h_1,...,h_10 independently and uniformly random in H.
Initialize 10 arrays T_1,...,T_10 that store n bits each.
Initially, for all i=1,...,10, all entries of T_i are 0.
for x in the black list:
      for i = 1,...,10:
            T_i[h_i(x)] = 1.
end for
```

We ship $T_1, \ldots, T_{10}$, and a description of $h_1, \ldots, h_{10}$, with the browser. Now, in order to ask if x is on the black list, we define:

```
function isMalicious(x):
      if T_i[h_i(x)] = 1 for all i = 1,...,10:
            return True
      else:
            return False
      end if
end function
```

Suppose that $n = 2w$. Show that all three requirements are met, with $b = 10d + 20w$. Notice that when w is much bigger than d, this is better than part (b)!

[We are expecting: A proof that all three properties hold. You may reference your proof in part (b) for part (c).]