Homework 6

Due: Mar 11, 2013 11:59 PM (early)
Mar 13, 2013 11:59 PM (on time)
Mar 16, 2013 2:00 PM (late)

The written portion of the homework must be handed in to the CS157 hand-in bin located on the CIT 2nd floor between the Fishbowl and the locker. The programming portion of homeworks (when applicable) must be handed in electronically via the hand-in script.

Each problem must be handed in separately with your name on the top and the time of hand-in. You may turn in different problems for different deadlines. For more information, please review the assignment and due date policy in the course missive on the course webpage.

Please ensure that your solutions are complete and communicated clearly: use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

You are allowed, and often encouraged, to use Wikipedia to help with your homework. See the collaboration policy on the course webpage for more details.

In this homework you will be working in pairs. These pairs are for this assignment only, and any future pair assignments must be done with a different partner. Please email the HTA list if you want us to find you a partner. You and your partner should write and hand in one solution per problem and put both your names on it.

Working in pairs will give you an opportunity to improve your thinking, communication, and writing skills. If something you write requires a verbal explanation for your partner to understand it, consider this a valuable sign that this explanation should be included in your writeup. In particular, you are responsible for everything you and your partner submit. It is an academic code violation to sign your name to something that is not yours. Further, however, the material covered on the homework will help prepare you for the exam, so aim for mastery of all of it.

Problem 1

(30 points) You can allocate a block of n memory locations on your computer in constant time, however the contents of the memory in the block may be arbitrary. Typically, you will *initialize* these memory locations before you use them, by setting them all to a special symbol `Empty`, which takes $O(n)$ time.

The object of this problem is to create a new data structure that mimics the properties of an array, while being much faster to initialize, but while still ensuring that any values returned by this data structure are meaningful, and not uninitialized garbage.

You need to come up with a data structure that behaves like a 0-indexed array A of n elements. The following operations must take a constant time:

- `set(index, value)`: Assign the `value` to A[`index`].
- `get(index)`: Return the value from A[`index`]. If no value has yet been assigned, return `Empty`.

- `initialize(n)`: Initialize the data structure so that it will mimic an array of size n, and where any `get` call returns `Empty`.

Warning: Keep in mind that, initially, the entries in memory can be *arbitrary* and may imitate valid parts of whatever data structure you design—your data structure should work *no matter what* is in memory initially.

Hints:

- Use more than n storage (but you do not need to use more than $O(n)$ storage).
- Most memory locations may be garbage, but think about how you can be sure *some* memory locations are meaningful.
- Because all operations in this data structure must take *constant* (worst case) time, you cannot use anything fancy: no hash tables, no binary search trees, no heaps, etc.

Important: If you are using extra space, please explain in sentences how they are used. As always, you need to communicate clearly that your proposed data structure works correctly, and that the running time for each operation, including initialization, is constant.

Problem 2

In this problem, you will investigate *self-balancing binary search trees* and how to *augment* them to be even more useful. Recall binary search trees: `http://en.wikipedia.org/wiki/Binary_search_tree`. One of the potential pitfalls of binary search trees is that they can become *unbalanced*, meaning that some nodes are much farther from the root than others. In particular, if we are storing n items, we would like all elements to have distance at most some small multiple of $\log n$ from the root. There are two standard notions of self-balancing binary search trees, which each guarantee that no matter how the elements are inserted or deleted, when there are n elements in the tree all elements will have distance at most some small multiple of $\log n$ from the root. AVL trees (`http://en.wikipedia.org/wiki/AVL_tree`) very aggressively rebalance the tree. Red-black trees (`http://en.wikipedia.org/wiki/Red-black_tree`) take a slightly more relaxed approach. In each case, rebalancing occurs via a sequence of *tree rotations* (`http://en.wikipedia.org/wiki/Tree_rotation`).

Skim the descriptions of red-black trees and AVL trees on Wikipedia, as you will be using them in the parts below; however, the internal details of how these trees work do not matter from our algorithm design perspective.

1. (3 points) From the Wikipedia articles (do not prove or justify this, just find it in the articles): How many rotations do red-black trees require for an insertion or deletion? How many rotations do AVL trees require for an insertion or deletion?

2. (4 points) For an arbitrary red-black tree, prove that the longest path from the root to a leaf contains at most twice as many nodes as the shortest path from the root to a leaf. (Hint: use properties 4 and 5 of red-black trees, as listed in `http://en.wikipedia.org/wiki/Red-black_tree#Properties`.)

(Note: if the following paragraph does not make sense to you, review the basics of binary search trees, and in particular make sure you understand the diagram under `http://en.wikipedia.org/wiki/Binary_search_tree#Deletion` .)

In the next two parts your challenge is to figure out how to *augment* such a self-balancing tree so that it stores additional information that will help you solve certain algorithmic challenges. This additional information must be easy to maintain: each of the operations on your data structure must take $O(\log n)$ time. While the internal details of red-black trees and AVL trees are very complicated, and rather different, for the purpose of building effective algorithms you may view them as being essentially the same: to insert or delete an item in these data structures, first an ordinary binary tree search is performed, and then at most one leaf is added or removed, and the value of at most one internal node is edited (which happens when a node is "swapped" during the ordinary process of node deletion). Then a complicated series of *rotations* is performed to rebalance the tree, but the number of such rotations is bounded by the expression you found in part 1. (In addition, $O(\log n)$ work may be done to update internals of the red-black tree or AVL tree data structure, but these internals do not affect the values or the structure of the tree.) In summary:

- The n items are stored as a binary search tree where each leaf has depth at most $O(\log n)$.
- Insertion and deletion in this data structure involve at most a constant number of the more fundamental operations: `add-leaf`, `remove-leaf`, `edit-internal-node-value`.
- In addition, insertion and deletion involve some number of calls to `rotate`, as you found in part 1.

In this problem there are two separate tasks that you must augment these data structures to handle. (Chapter 14.2 of the CLRS textbook has an introduction to this idea.)

3. (15 points) Augment a self-balancing binary search tree whose nodes contain numbers so that your data structure can also respond in *constant* time to `find-minimum-difference()`, which must return the minimum difference between any two elements currently stored in the tree.

   You must state what additional data you are storing, as well as how to update this information when performing each of the four fundamental operations `add-leaf`, `remove-leaf`, `edit-internal-node-value`, and `rotate`. Given these four subroutines, and the basic facts above, conclude that the total time spent by your algorithm for each insertion and deletion is $O(\log n)$. Additionally, make sure to analyze how long it takes to compute the minimum difference between two elements given your data structure.

   (**Hint:** For every node, store the minimum difference between any two elements in its subtree; the challenge is to figure out what *else* to store at each node so that you can update these minimum differences efficiently.)

4. Bilbo is organizing an army in a battle for Middle-Earth, and his most important task is being able to tell the currently surviving soldiers who their commander is. The army has the following particularly simple structure: each soldier is identified with a number corresponding to his or her rank, and the soldier's commander is the currently-alive soldier with the least rank greater than his or her own (or "no one" if the soldier is has the highest rank of any surviving soldier).

   (a) (3 points) Describe how to implement the `get-commander(i)` function. When called on a number i that is a rank of a soldier, it returns the least rank greater than i in the tree.

However, the battle for Middle-Earth is vastly more confusing than this simple model might suggest. Bilbo can receive messages of the following forms:

- "Soldier i is in our army."
- "Ignore whatever you heard previously; soldier i is *not* in our army."
- "Soldier i died at time t."
- "Ignore whatever you heard previously; soldier i survived the battle."
- "Soldier i at time t wants to know who his or her commander is."

Because the battle is so confusing, the sequence of messages *will not* be the same as the sequences of times contained in them. For example, if Bilbo hears that soldiers 1, 2, and 3 are in the army, and then hears that at time 10 soldier 2 died, and then gets a message that at time 5 soldier 1 wants to know her commander, Bilbo should return "2", despite having already heard about soldier 2's future death.

(b) (15 points) Augment a self-balancing binary search tree so that you can help Bilbo react or respond to each of the 5 kinds of messages in $O(\log n)$ time.

You must state what additional data you are storing, as well as how to update this information when performing each of the four fundamental operations `add-leaf`, `remove-leaf`, `edit-internal-node-value`, and `rotate`. Given these four subroutines, and the basic facts above, conclude that the total time spent by your algorithm for of the five kinds of messages is $O(\log n)$.

(**Hint:** At each node, in addition to storing a "time of death"—which may be ∞—you should figure out what *additional* information to store, so that you can compute soldiers' commanders efficiently.)

One of the features of good writing style is to say everything once and no more than once. When you are writing up this problem, please find a way to organize your presentation so that similar repeated parts of your argument are instead compressed into a single unit. This will make it easier for you to write and for us to read, and will sound more professional.

Problem 3

1. When choosing a hash function, we want to make sure that collisions are unlikely. One way to ensure this is to randomly choose a hash function from a large family, where different functions in the hash function family scramble elements in different ways.

   A hash function family H is a family of functions $\{h_p\} : X \to Y$, where p ranges over *parameters* in a set P. Throughout this problem, we will let m be the size of the range of the hash function family, $m = |Y|$. Typically, a hash function is parameterized by several parameters; for example, if h is parameterized by triples $p = (p_1, p_2, p_3)$, where p_1 ranges over some set P_1, p_2 ranges over some set P_2, and p_3 ranges over some set P_3, then the universe of parameters P consists of all values of these triples. Specifically, $P = P_1 \times P_2 \times P_3$, and $|P| = |P_1| \cdot |P_2| \cdot |P_3|$.

   A hash function family H is called *universal* if for each pair $a, b \in X$ with $a \neq b$, at most $\frac{|P|}{m}$ out of the $|P|$ parameters p make a and b collide as $h_p(a) = h_p(b)$.

For each of the following hash function families, either prove it is universal or give a counterexample. Additionally, compute how many bits are needed to choose a random element of the family (namely, compute $\log_2 |P|$ in each case).

The notation $[m]$ denotes the set of integers $\{0, 1, 2, \ldots, m-1\}$.

(a) (3 points) $H = \{h_p : p \in [m]\}$ where m is a fixed prime and

$$h_p(x) = px \mod m.$$

Each of these functions is parameterized by an integer p in $[m]$, and maps an integer x in $[m]$ to an output in $[m]$.

(b) (3 points) $H = \{h_{p_1,p_2} : p_1, p_2 \in [m]\}$ where m is a fixed prime and

$$h_{p_1,p_2}(x_1, x_2) = p_1 x_1 + p_2 x_2 \mod m.$$

Each of these functions is parameterized by a pair of integers p_1 and p_2 in $[m]$, and maps a pair of integers x_1 and x_2 in $[m]$ to an output in $[m]$.

(c) (3 points) H is as in part 3.1b except m is now a fixed power of 2 (instead of a prime).

(d) (4 points) H is the set of all functions from pairs $x_1, x_2 \in [m]$ to $[m]$.

2. (3 points) Hacking a hash function: suppose for a member of the hash function family from part 3.1b you have found two inputs (x_1, x_2) and (x'_1, x'_2) that hash to the same value. Describe how to find further inputs that collide.

   (Suppose you are interacting with a server, and you start to suspect that the server is using a hash function like this. This sort of technique might be used to crash the server, if their hash function data structures are not implemented well.)

3. (7 points) A much stronger property than universal hashing is *k-independent hashing*. A hash function family $\{h_p\} : X \to Y$ is k-independent if for any distinct $x_1, \ldots, x_k \in X$ and any $y_1, \ldots, y_k \in Y$, for exactly a $\frac{1}{m^k}$ fraction of parameters p we will have $h_p(x_1) = y_1$ and $h_p(x_2) = y_2$ and ... $h_p(x_k) = y_k$.

   For a prime m consider the family of hash functions from $[m]$ to $[m]$ parameterized by $p = (p_0, \ldots, p_{k-1})$, where $h_p(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_{k-1} x^{k-1} \mod m$. Show that this family is k-independent.

   (**Hint:** Recall the familiar fact that for any k distinct real numbers $x_1, \ldots, x_k$, and any k real numbers $y_1, \ldots, y_k$, there is a *unique* degree $k-1$ polynomial that passes through these k pairs (x_i, y_i). The same fact is true modulo a prime m. Assume and use this fact.)

   Intuitively, if a hash-function is 10-independent, this means that for any 10 elements, their hash destinations will look as though they had been chosen uniformly at random. This is very useful for analyzing (and preventing) unfortunate hashing patterns involving up to 10 elements, because you can deduce that such patterns will occur no more often than if the hash destinations had been chosen at random.

4. (7 points) The hash function family $h_p(x)$ of the previous part is a bit odd because it maps $[m]$ to itself. Consider instead a prime q that is smaller than m, and consider instead a new hash function family $h'_p(x)$ from $[m]$ to $[q]$ computed as: $h'_p(x) = h_p(x) \mod q = (p_0 + p_1 x + p_2 x^2 + \ldots + p_{k-1} x^{k-1} \mod m) \mod q$—the hash function from the previous part,

parameterized identically, but then taken modulo q. This new hash function family $h'_p(x)$ will *not* be k-independent, but in many cases it will be "close enough for practical purposes". (Note that, unlike for the previous parts of this problem, the range of h' has size q instead of m.)

Find bounds on the fraction of parameters p such that $h'_p(x_1) = y_1$ and $h'_p(x_2) = y_2$ and ... $h'_p(x_k) = y_k$, when $x_1, \ldots, x_k$ are distinct elements of $[m]$ and $y_1, \ldots, y_k$ are elements of $[q]$.

Use these bounds and the approximation $e^x \approx 1 + x$ for small x to show that (subject to this approximation), when q is smaller than m/k, then the fraction of p such that $h'_p(x_1) = y_1$ and $h'_p(x_2) = y_2$ and ... $h'_p(x_k) = y_k$ is within a factor of e of $\frac{1}{q^k}$. (Thus, the hash function family $h'_p(x)$ is "e-close to being k-independent".)

(**Hint:** Given that $h'_p(x_1) = y_1$ and $h'_p(x_2) = y_2$ and ... $h'_p(x_k) = y_k$, what does this say about the hash function of the previous part, h_p? Specifically, for each i between 1 and k, we have $h_p(x_i) \mod q = y_i$. Let $\bar{y_i}$ be the set of all integers in $[m]$ that are equivalent to y_i modulo q. We thus have that $h_p(x_i) \in \bar{y_i}$ for each i. If you come up with bounds on the size of each $\bar{y_i}$ you can finish the problem from the results of the previous part.)