Homework 4: Codr

Due: Oct. 9, 2018 at 6:00 PM (early)
Oct. 12, 2018 at 6:00 PM (on time)
Oct. 14, 2018 at 6:00 PM (late)

This is a partner homework as usual: work on all problems together; come to all office hours together; you are responsible for everything you and your partner submit and it is an academic code violation to submit something that is not yours. Check for your partner in the Google doc that was sent out. **If you don't have a partner, please let us know.**

One of the features of good writing style is to say everything once and no more than once. When you are writing up these problems, 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.

> We are pivoting to a new business model. Meet `Codr`: It's Omegle for Pair Programming. In this assignment you will help us match programmers with other programmers so that our customers can find new pair programming buddies.

1-credit track: do problems 1, 2, 3 *and indicate "***1-credit***" on each problem.*

Problem 1

> We represent each `Codr` user internally by hashing them to a small ID number, and we hope that each user will get hashed to a unique number so that we can distinguish our users. If two `Codr` users hash to the same ID, we may accidentally assign them to work with the wrong partners. Oops! Help us analyze hash functions so as to minimize collisions.

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

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

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

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

$$h_{p_1,p_2}(x_1, x_2) = (p_1x_1 + p_2x_2) \bmod m.$$

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 $[m]$ ($P = [m] \times [m]$, $X = [m] \times [m]$, $Y = [m]$).

(c) (3 points) H is as in part 1b except m is now a fixed power of 2 (instead of a prime).

(d) (3 points) H is the set of *all* functions (there are a lot!) from pairs $(x_1, x_2) \in [m] \times [m]$ to $[m]$.

2. (3 points) Hacking a hash function: suppose for a member of the hash function family from part 1b you have found two inputs (x_1, x_2) and (x_1', x_2') that hash to the same value. Describe explicitly how to find a third input that collides with both of these inputs.

   (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. Your method above should also let you find a fourth, fifth, etc. inputs that collide, until the server has problems.)

Problem 2

> It turns out that people love collaborating on problems—our userbase has grown so much that we need to transfer our data to another system! We have to place people in a really large array. Can you help us initialize this array quickly?

(20 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 such as 0 or `Empty`, which takes $O(n)$ time.

The goal 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:

- Initialize(n): Initialize the data structure so that it will mimic an array of size n.
- 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`.

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.

Notes:

- Use more than n storage, but do not 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 worst-case *constant* 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 it is 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 3

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. (2 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`.)

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 or a node with only one child is 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-Node-With-At-Most-One-Child, and Edit-Internal-Node-Value. (For context for each of these operations, see how they are needed when we delete an element from a binary search tree: `https://en.wikipedia.org/wiki/Binary_search_tree#Deletion`.)
- In addition, insertion and deletion may 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. (12 points) At Codr, business is booming. The only problem is that often someone who likes functional programming styles will pair with someone who doesn't, and they will spend all their time rewriting `for` loops as `map` invocations and vice versa. To solve this, we want to try an alternative pairing strategy that pairs people together who are most similar in how much they like functional programming. Every user tells Codr how much they like functional programming on a scale from 1 to 10, and your job is to find users who are closest together. For the proof of concept, however, you are just going to find the minimum distance between users (the algorithm for this is similar to the algorithm for finding the two closest users).

   Consider a self-balancing binary search tree whose nodes contain numbers. Augment this data structure so that your data structure can now also respond in *constant* time to Find-Minimum-Difference(T), which must return the difference between the closest pair of numbers currently stored in the tree T. For example, if the tree stores the numbers $1, 4.7, 5.3, 6, 7$, then the closest pair of numbers would be 4.7 and 5.3, with a difference of 0.6, and your algorithm should return 0.6.

   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-Node-With-At-Most-One-Child, 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. Suppose there is an infinitely long, straight line, and people will sometimes step onto the line or step off of it, at locations corresponding to numbers. People standing on the line will always be facing in the positive direction. Unfortunately, some of these people hold gravity-defying throwing knives that kill the next person on the line.

   In particular, you are going to observe a series of events, each of which will have one the following formats:

   [**E1**] A person outside the line moves into unoccupied position p on the line.

   [**E2**] A person standing at position p exits the line.

   [**Q**] A person standing at position p throws a knife in the positive direction.

   Your objective is to efficiently compute what will happen (so you can warn a would-be victim before they get hit by the knife); if no one will get hit by a knife, report this.

   (a) (4 points) Describe how to use a (self-balancing) binary search tree to respond to the series of events as described above. Each event should be responded to in $O(\log n)$ run–time, where n is the total number of people.

   The problem, however, gets more complex than this: in reality, each knife is thrown at a different height, and only someone whose whose height is *strictly greater* than the knife's height will get hit. As above, each event will have one of the following formats:

   [**E1**] A person with height h outside the line moves into unoccupied position p on the line.

   [**E2**] A person standing at position p exits the line.

   [**Q**] A person standing at position p throws a knife in the positive direction at an arbitrary height j.

   (b) (8 points) Augment a self-balancing binary search tree and describe (same rules as above) how to use it to respond to the series of events in this more complicated game. Each event should be responded to in $O(\log n)$ run–time, where n is the total number of people.

   Hint: At each node, in addition to storing "a height of a person," you should figure out what *additional* information to store, so that you can respond to the events efficiently.

Problem 4

> When `Codr` users get matched, they immediately start sending each other memes, and we need to store all the currently used memes in our server's memory (pronounced *MEME*-ory). Because the set of *quality* memes is constantly changing, we need a flexible memory management system that can quickly allocate and free up different sized blocks of memory, analogous to what is done in languages like C.

The ALLOCATE function takes `size` as a parameter, and returns the location of the start of a contiguous chunk of unused memory of the requested size, that your code is now free to use. When you are finished with a chunk, you call the FREE function on the location of the start of that chunk, indicating that the memory may be reallocated for other purposes. For example, after calling $p1 = \text{ALLOCATE}(1)$, $p2 = \text{ALLOCATE}(5)$, $p3 = \text{ALLOCATE}(2)$, the diagram of used memory might look like the following (with numbers indicating which block of memory each location corresponds to, if any):

		3	3				1	2	2	2	2	2

This corresponds to $p1 = 8$, $p2 = 9$, and $p3 = 3$. After calling FREE(9), memory will look like this:

		3	3				1					

Thus if we try calling ALLOCATE(6) next it will fail, because there is no block of 6 adjacent empty memory locations, despite the fact that there are 10 free memory locations total!

Consider the following high-level description of a scheme—in terms of a memory size N—to implement ALLOCATE and FREE: for each k from 0 up to $\log_2 N$, there is a *separate* region of N memory, divided into chunks of size 2^k. Whenever ALLOCATE(*size*) is called, *size* is rounded up to the nearest 2^k, and then an empty chunk in the kth region of memory is returned, if one exists, otherwise the algorithm fails.

1. (a) (1 point) What is the total memory required by this scheme? (Don't count overhead from data structures needed to actually implement such a scheme.)
   (b) (3 points) What is the smallest amount of allocated memory that could make such a scheme fail? ("What is the most embarrassing situation for this scheme?") Specifically, describe a sequence of ALLOCATE requests that is sure to make the algorithm fail, and argue why you have found the sequence with least total memory used—by "used" we mean both the currently allocated memory and the size of the block being requested when the scheme fails. (You should not need to call FREE for this part.)
   (c) (1 point) What is the "efficiency" of this scheme, the ratio between your answers to the previous two parts? Explain this in a sentence.)
2. (5 points) Describe how to implement ALLOCATE and FREE as described above, in constant time per call, using constant amount of memory overhead for each chunk. (You are also allowed to use overhead for each of the unallocated chunks). If your solution leverages specific data structures, be explicit about how they are used, and what properties of them you rely on to successfully implement your procedures. Do not worry about time taken to initialize your data structures, only the time used per call.

3. (3 points) In the next part we will try to show that the bounds you found in part 1 are actually about as good as can be expected, even though this scheme appears wasteful. To help give you some intuition before you start the next part: *first think about how you would design* Allocate *and* Free*, differently from the scheme above.* This part has no specific requirements, but the more you think about how else Allocate and Free might work, the more you will gain from the next parts. Write down an alternate scheme here, along with why you think it might be a good idea.

4. Now we get to the tricky part: showing that the scheme of parts 1 and 2 is close to optimal, by showing that any other scheme can be made to perform as badly. In this part you will construct and analyze some *online adversaries* for **any** allocation scheme. Recall that an online adversary can adaptively respond to decisions made in the past. In this case, your adversary will be able to see *where* different pieces of memory were allocated, and can design future calls to Allocate and Free to take advantage of this. Let m be the total size of memory.

   (a) (3 points) Describe an algorithm that makes a series of calls to Allocate and Free (no matter how they are implemented!) and which guarantees the following: either some call to Allocate(1) will fail when there is still unused memory, or a single call to Allocate(2) will fail when only half the memory is being used. (**Note/hint:** in this part, you are allowed to entirely fill up the memory, and then selectively free parts of it. Remember, your algorithm must work for *all* possible allocation schemes.)

   (b) (1 point) In a sentence or two, describe how to adapt your answer to the previous part so that either some call to Allocate(1) will fail when there is still unused memory, or a single call to Allocate($\sqrt{m}$) will fail when only $\sqrt{m}$ memory is being used.

   (c) (18 points—8 points for algorithm, 10 for proof) What you found in the last two parts is bad news for allocation algorithms, but still not that embarrassing: after all, if you completely fill up memory, then one might expect allocation strategies to struggle with putting things in appropriate places. Our challenge now is to run out of memory *without* **ever** *using more than a $O(\frac{1}{\log m})$ fraction of it.*

   Your task is to **1)** construct an (online) algorithm that makes a series of calls to Allocate and Free (no matter how they are implemented!), where, given m memory, your scheme will cause the memory allocator to fail without ever having more than $O(\frac{m}{\log m})$ memory requested; **2)** prove the correctness of your online algorithm/adversary. Note: you may find it intuitively useful to reparameterize the problem by thinking of $s = c\frac{m}{\log m}$ for some constant c that you choose later, and then saying that your algorithm will never ask for more than s memory, but will cause a system with $O(s \cdot \log s)$ memory to crash. A bit of thought will confirm that $\log m$ and $\log s$ are essentially the same number (if two numbers are close to each other, their logs will be *very* close to each other), so it does not matter whether we are talking about $\log m$ factor memory overhead or $\log s$ factor memory overhead.

   Carefully read all the hints below: The general strategy for your algorithm should be:

Call $\text{ALLOCATE}(1)$ for $\frac{s}{2}$ times, resulting in $\frac{s}{2}$ allocated memory, consisting of $\frac{s}{2}$ chunks of size 1.

for $i = 1, 2, 3, \ldots$

i. Call $\text{ALLOCATE}(2^i)$ for $\frac{s}{2} \cdot \frac{1}{2^i}$ times, resulting in $\frac{s}{2}$ new memory, consisting of $\frac{s}{2} \cdot \frac{1}{2^i}$ chunks of size 2^i;

ii. Carefully free up at least half of the allocated memory (allocated in this i iteration or previous iterations) so as to leave memory maximally "fragmented".

The hard part is figuring out how to free your memory in step 2 so that you can prove your algorithm works. You need to design your algorithm with the following proof strategy in mind.[1] At the end of the i^{th} iteration (which is also the start of the $i + 1^{\text{st}}$ iteration) consider drawing lines to divide memory into regions[2] of size 2^i. For $i = 1$ we consider regions $\boxed{1,2}, \boxed{3,4}, \boxed{5,6}, \ldots, \boxed{2j+1, 2j+2}, \ldots$ all of size 2, and for general i, the regions are $\boxed{1, 2, 3, \ldots, 2^i}, \boxed{2^i+1, 2^i+2, 2^i+3, \ldots, 2 \cdot 2^i}, \ldots,$ $\boxed{j \cdot 2^i + 1, j \cdot 2^i + 2, j \cdot 2^i + 3, \ldots, (j+1) \cdot 2^i}, \ldots$.

The crucial property you want (for your algorithm and proof) is: at the end of iteration i, each region contains *at most 1* **center** *of an allocated chunk*, where the *center* of an allocated chunk of memory is the average of its left and right endpoints, rounded down. How do you prove such a strange property? You need to rely on the assumption that your algorithm was correct at the end of the previous iteration—put this strange claim about centers of allocated chunks in an induction hypothesis, and induct on i. Namely, at the start of iteration i, you can assume that your algorithm correctly ended iteration $i-1$, meaning that, when dividing memory into (smaller) regions of size 2^{i-1}, each (small) region contains at most 1 center of an allocated chunk at the start of the ith iteration. Now, you need to consider the next two steps of the algorithm: when step 1 calls $\text{ALLOCATE}(2^i)$ repeatedly, how does this affect our understanding of regions/centers? (What tweak of this induction hypothesis is true in between steps 1 and 2 of iteration i?) Next, you need to figure out a clever plan to free roughly half the memory, so that when pairs of (small) regions of size 2^{i-1} get merged into a single (big) region of size 2^i, only one chunk center remains in each such (big) region.

Details and conclusion: You should design step 2 so that it frees up **exactly half** *of the total number of allocated chunks* (ignoring off-by-1 issues, which we mostly do not care about in this course), while preferring to free larger chunks over smaller chunks so as to free up **at least half** *of the total allocated bytes.*

Suppose you have figured out how to design and analyze a good scheme for step 2, why does this lead to breaking the memory allocator? After i iterations, you should carefully count and/or bound the parameters of the current memory configuration. The intuition is that, for each region of size 2^i that contains a chunk center, none of the $\text{ALLOCATE}(2^{i+1})$ calls in the next iteration can return a chunk centered in this region

[1] This idea came up in the dynamic programming section of the course: sometimes when you are stuck designing an algorithm, you can get guidance by trying to "think ahead" to the proof, and figure out what kind of algorithm could possibly be proven correct.

[2] This is related to the proof technique for analyzing the competitive ratio of LRU cache we saw in class—these regions exist only on paper, as a proof technique; we can draw anything we want on our paper, without affecting what happens on the computer; drawing lines and carefully counting what is between the lines is merely an analysis technique.

because it would overlap the chunk already centered there (draw a diagram for this, and prove it!). Thus, if iteration i ends with ℓ chunks allocated, each centered in a different region of size 2^i, this means that each of these $\ell \cdot 2^i$ locations in these regions *cannot be a center of a chunk allocated at the start of the next iteration* $i+1$. Restated, any region that already contains a chunk center will be "not available for future large memory requests". Carefully track the product, $\ell \cdot 2^i$ computing the size of all these regions, and, when this value exceeds the total memory m, point out that step 1 in iteration $i+1$ must crash, thus concluding your proof that your algorithm will make the memory allocator crash.

After each iteration i, the memory will get more and more fragmented; what is the maximum value of i for which your analysis makes sense? (Your analysis certainly will not make sense for $i > \log_2 m$, since then we are talking about allocating chunks of memory of size $2^i > m$.)

Try to write up your algorithm and proof as cleanly as possible. **Do not** refer to this handout in your writeup, as this will lead to a confusing presentation. Explain and prove everything from scratch, aiming for clear communication throughout. Talk to the TAs if you are having trouble expressing your ideas.

(To understand what you have done, go back and think about how what you have just found would defeat the scheme you came up with in part 3.)