Homework 4
Due: Oct. 20, 2015 at 6:00 PM (early)
Oct. 23, 2015 at 6:00 PM (on time)
Oct. 25, 2015 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.

Problem 1

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 $\{h_p\} : X \rightarrow Y$, 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 $|P|/m$ 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 1b except $m$ is now a fixed power of 2 (instead of a prime).
(d) (3 points) $H$ is the set of all functions 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

(25 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 $\text{Empty}$, 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:

- **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 $\text{Empty}$.

**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](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](http://en.wikipedia.org/wiki/AVL_tree)) very aggressively rebalance the tree;


In each case, rebalancing occurs via a sequence of tree rotations ([http://en.wikipedia.org/wiki/Tree_rotation](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](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.
- 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. (12 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($T$), which must return the minimum difference between any two elements currently stored in the tree $T$.

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 of 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:
A person with height $h$ outside the line moves into unoccupied position $p$ on the line.

A person standing at position $p$ exits the line.

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

In this problem you will be analyzing the style of memory management that languages like C use. 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 $p_1 = \text{Allocate}(1)$, $p_2 = \text{Allocate}(5)$, $p_3 = \text{Allocate}(2)$, 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 $p_1 = 8$, $p_2 = 9$, and $p_3 = 3$. 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 to implement Allocate and Free, parameterized by an amount of memory $N$: 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 $k$th 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. (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 \texttt{Allocate} and \texttt{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: \textit{first think about how you would design \texttt{Allocate} and \texttt{Free}, differently from the scheme above}. This part has no specific requirements, but the more you think about how else \texttt{Allocate} and \texttt{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 \textit{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 \textit{where} different pieces of memory were allocated, and can design future calls to \texttt{Allocate} and \texttt{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 \texttt{Allocate} and \texttt{Free} (no matter how they are implemented!) and which guarantees the following: either some call to \texttt{Allocate}(1) will fail when there is still unused memory, or a single call to \texttt{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 \texttt{Allocate}(1) will fail when there is still unused memory, or a single call to \texttt{Allocate}(\sqrt{m}) will fail when only $\sqrt{m}$ memory is being used.

   (c) (8 points) 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 \textit{ever} using more than a $O\left(\frac{1}{\log m}\right)$ fraction of it.

   The general strategy is as follows: allocate many chunks of size 1, then free some of them, next allocate many chunks of size 2, then free up some chunks, repeating for chunk sizes that are successive powers of two. The tricky part is how to decide which elements to free up each time so as to lead to the most fragmented memory later—specifically, we need just the right induction hypothesis, to capture exactly what properties of the chunks we will enforce in each allocate/free phase.

   In this part you will design an algorithm with very unusual input and output requirements; in the next part you will use this algorithm as the inductive step of an induction proof—you can think of the input requirements of your algorithm as the induction hypothesis. (If you get confused, read through both parts and try to understand the “big
Initially, suppose three conditions:

i. There is \( s \) total memory allocated.

ii. This memory consists of \( n \) chunks.

iii. For a certain number \( k \), each of the regions of memory \( 1, 2, 3, ..., k \), \( k + 1, k + 2, k + 3, ..., 2k \), ..., \( ik + 1, ik + 2, ik + 3, ..., (i + 1)k \) for any \( i \), each contains at most one center of an allocated chunk, where the center of an allocated chunk of memory is the average of its left and right endpoint, rounded down.

Given these three input conditions, call \( \text{Allocate}(2k) \) \( s \) times, to allocate \( s \) more memory (doubling the amount of allocated memory at the moment).

Find an algorithm that, given the above situation as input, chooses how to \( \text{Free} \) memory so that, afterwards, the following three conditions become satisfied: 1) the total amount of allocated memory goes down to at most \( s \); 2) the number of allocated memory chunks is at least half of what it just was, \( \frac{1}{2}n + \frac{s}{2k} \); and 3) if memory is divided into regions of size \( 2k \) (instead of \( k \)), as \( 1, 2, 3, ..., 2k \), \( 2k + 1, 2k + 2, 2k + 3, ..., 4k \), ..., \( 2ik + 1, 2ik + 2, 2ik + 3, ..., 2(i + 1)k \) for any \( i \), then each region contains at most one center of an allocated chunk.

Describe your algorithm in sufficient detail to prove its correctness. (Run-time does not matter.)

(d) (5 points) Consider starting with \( n \) chunks of memory allocated, each of size 1, and let \( k \) initially be 1. The total amount of allocated memory is \( s = n \) now. Consider running your algorithm repeatedly starting in this configuration, where at stage \( i \) your algorithm runs on parameters \( k_i, n_i, s \), that change as described in the previous part: \( k_{i+1} = 2 \cdot k_i \), \( n_{i+1} \geq \frac{1}{2}(n_i + \frac{s}{2k_i}) \), and \( s \) does not change (if your algorithm decreases \( s \), you can increase it back to where it started without ruining our inductive guarantees, by calling \( \text{Allocate}(4k) \) repeatedly until \( s \) memory is currently allocated).

The following intuition you have to make precise: Intuitively, the three conditions at the beginning of each call to your algorithm guarantee that there are at least \( n \) regions of memory of size \( k \) that are partially occupied. More specifically, if we then try to call \( \text{Allocate}(2k) \), the center of the allocated chunk cannot be in any of these size--k regions. (Why?) Thus at the beginning of each run of your algorithm there are \( kn \) memory locations that are “not available for large memory requests”; when your algorithm is run next, this number increases to the product of the new values of \( k \) and \( n \), and you want to show that the product \( kn \) increases by at least \( \frac{s}{k} \) each iteration.

Put the above pieces clearly and cleanly together (this requires work!) to prove that any \( \text{Allocate}/\text{Free} \) scheme, when run on a computer with \( m \) memory, can be made to fail by an algorithm that never uses more than \( O(\frac{m}{\log m}) \) memory. Important: Write this part up cleanly, starting with sentences like: “We show that any \( \text{Allocate}/\text{Free} \) scheme, when run on a computer with \( m \) memory, can be made to fail by an online algorithm/adversary that never uses more than \( O(\frac{m}{\log m}) \) memory. We will prove our claim by showing by induction that…”.

(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.)