Problem Set 4: Amortized Efficiency

In this problem set, you'll get to explore amortized efficient data structures and a number of their properties. You'll design a few new data structures and explore some of the ones from lecture in more detail. By the time you're done, you'll have a much deeper understanding of amortized efficiency and just why all these data structures are so nifty.

Due Thursday, May 17th at 2:30PM.

Problem One: Stacking the Deque (3 Points)

A *deque* (***d**ouble-**e**nded **que**ue*, pronounced “deck”) is a data structure that acts as a hybrid between a stack and a queue. It represents a sequence of elements and supports the following four operations:

- *deque.*`add-to-front`(*x*), which adds *x* to the front of the sequence.
- *deque.*`add-to-back`(*x*), which adds *x* to the back of the sequence.
- *deque.*`remove-front`(), which removes and returns the front element of the sequence.
- *deque.*`remove-back`(), which removes and returns the last element of the sequence.

Typically, you would implement a deque as a doubly-linked list. In functional languages like Haskell, Scheme, or ML, however, this implementation is not possible. In those languages, you would instead implement a deque using three stacks.

Design a deque implemented on top of three stacks. Each operation should run in amortized time O(1).

The lecture slides on amortized analysis contain an exploration of how to build a queue from two stacks such that each operation runs in amortized O(1). That might be a good starting point to build from when working on this problem.

Problem Two: Constructing Weight-Balanced Trees (10 Points)

Problem Set Two introduced weight-balanced trees in the context of trie representations, but we never actually talked about how to build them. Let's remedy that. 😃

As a refresher, a ***weight-balanced tree*** is a binary search tree built from a list of keys, each of which is annotated with a positive real-valued weight. The key placed in the root node is chosen such that the difference in weights between its left and right subtrees is as close to zero as possible.

We're going to assume that the input is provided to us as an array of n key/weight pairs (k_1, w_1), (k_2, w_2), …, (k_n, w_n) where the keys are in sorted order. The sorting requirement makes it easier to reason about what happens if we pick a particular key to put in the root. Plus, as you'll see later on, if the keys aren't sorted, the cost of sorting will exceed the cost of building the weight-balanced tree!

All of the algorithms we're going to explore in this problem will follow this general template:

- Using some search procedure, locate the key that has the optimum left/right split.
- Place that key into the root of the tree, then recursively build weight-balanced trees for the keys belonging to the left subtree and to the right subtree.

The question, then, is what the best search procedure is for identifying which key gets to be in the root.

Let's begin with a useful observation. Imagine that you scan the keys from left to right, considering each one as a candidate for the root. As you move from the left to the right, the weight in the left subtree will monotonically increase and the weight in the right will monotonically decrease. This means that the difference of the right subtree weight and the left subtree weight monotonically decreases across a left-to-right scan. As a result, that the absolute value of this difference – which is the balance factor between the two subtrees – consists of a monotone decreasing sequence (as the difference gets closer and closer to zero from the positive direction) followed by a monotone increasing sequence (as the differences gets further from zero in the negative direction).

We can exploit this property to find the the optimum splitting point by scanning the array from the left to the right and looking for where the absolute value of the difference in weights increases. The key right before the increase then belongs in the root. However, the time to construct a tree this way is $\Theta(n^2)$ in the worst-case. (Do you see why?) We'll instead opt for another approach: simultaneously scan inwards from the left *and* the right sides of the array, looking for the crossover point, and stopping as soon as we find it.

i. Argue that running this search procedure on an array of length n takes time $\Theta(1 + \min\{k, n - k\})$, where k is the index of the optimal key.

To determine the worst-case runtime of this approach, since this is a recursive algorithm, you might initially expect that we'd want to solve a recurrence relation of some sort. We could do that, but the recurrence is really messy because we have no idea where the optimum element is going to be:

$$T(n) = \max_{1 \le k \le n}\{T(k - 1) + T(n - k) + \Theta(1 + \min\{k, n - k\})\}$$

That's not a fun recurrence by any stretch of the imagination. However, using techniques from amortized analysis, we can solve it cleanly and beautifully in a different way.

ii. Prove that this algorithm's runtime is $O(n \log n)$. To do so, place credits on the keys in the array, each of which is redeemable for $O(1)$ units of work, then show that you can pay for the entire algorithm by spending those credits. Don't try solving the above recurrence; it's a losing game. 😃

(Continued on the next page.)

Let's now see if we can speed this algorithm up a bit.

The above algorithm takes advantage of the fact that the weight disparity monotonically decreases and then monotonically increases as you scan inward from either side. As a result, we don't need to do a linear scan to find the optimal key – we can use a binary search instead! We'll probe a key, then look at it and the key after it. If the split disparities are increasing, we know the optimal key must be to the right. Otherwise, it must be somewhere to the left. We can use this strategy to winnow the range down to a single element, which must then be the optimal key.

For this binary search to work efficiently, we'll need a quick way of determining what the weights in a given node's left and right subtree would be. Fortunately, it's possible to do $O(n)$ preprocessing before our recursive algorithm so that we can query for the weights difference in time $O(1)$. (Do you see how?)

So now let's imagine that we follow the algorithm template and use binary search. Although it might seem like this would improve over our linear scan, this algorithm turns out not to be any more efficient.

iii. Prove that this algorithm runs in time $O(n \log n)$. Then show that there are arbitrarily large inputs for which this algorithm runs in time $\Omega(n \log n)$.

It's surprising that this algorithm isn't faster than before, given that we're using binary search versus linear scans. The algorithm from part (ii) ends up being fast because the work done on each scan depends not on the size of the overall array, but rather on the position of the optimal key within that array. The naive binary search described above doesn't have this property; it might end up searching over the entire array regardless of where the optimal key is. This raises a question: is there a way to modify binary search so that the runtime depends not on the total size of the array, but just on the distance from the optimal key to the closer of its two sides? The answer, amazingly, is yes.

Consider the following variation on binary search. Start from both edges of the array and work inward, probing the keys $2^0 - 1$, $2^1 - 1$, $2^2 - 1$, $2^3 - 1$, …, etc. steps away from the sides of the array, until one of those searches overshoots the optimal key. (You can tell that you've overshot by looking at the probe location and the position one step past that location, moving away from the edge, and seeing if the split costs are starting to increase.) Then, do a regular binary search of just the first 2^t elements on the appropriate side of the array to find the optimal key, where 2^t is the probe size that overshot the optimum.

iv. Argue that this search procedure takes time $\Theta(1 + \min\{\log k, \log n - k\})$, where k is the index of the optimal key.

Our final algorithm is to use the recursive framework with this modified binary search instead of a our prior linear searches. To see how fast this is, we could, as before, "just" solve the following recurrence:

$$T(n) = \max_{1 \le k \le n}\{\ T(k-1) + T(n-k) + \Theta(1 + \min\{\log k, \log (n-k)\})\ \}$$

Again, solving this directly would be messy, and not fun at all. But just like in part (ii) of this problem, we don't actually need to solve this recurrence! We can use techniques from amortization instead.

v. Prove that this algorithm runs in time $O(n)$. There are several ways you could do this. We recommend adapting your approach from part (i) and changing where you place and spend credits. You may find it easiest to envision spending fractional credits at various points. Alternatively, you could directly try to solve this recurrence. (We recommend the former approach!)

So there you have it – by running two parallel modified binary searches, we can build weight-balanced trees in time $O(n)$!

Problem Three: Meldable Heaps with Addition (12 Points)

Meldable priority queues support the following operations:

- `new-pq`(), which constructs a new, empty priority queue;
- *pq*.`insert(`*v*, *k*), which inserts element *v* with key *k*;
- *pq*.`find-min`(), which returns an element with the least key;
- *pq*.`extract-min`(), which removes and returns an element with the least key; and
- `meld`(pq_1, pq_2), which destructively modifies priority queues pq_1 and pq_2 and produces a single priority queue containing all the elements and keys from pq_1 and pq_2.

Some graph algorithms, such as the Chu-Liu-Edmonds algorithm for finding minimum spanning trees in directed graphs (often called "optimum branchings"), also require the following operation:

- *pq*.`add-to-all`(Δk), which adds Δk to the keys of each element in the priority queue.

Using lazy binomial heaps as a starting point, design a data structure that supports all `new-pq`, `insert`, `find-min`, `meld`, and `add-to-all` in amortized time O(1) and `extract-min` in amortized time O(log *n*).

Some hints:

1. You may find it useful, as a warmup, to get all these operations to run in time O(log *n*) by starting with an *eager* binomial heap and making appropriate modifications. You may end up using some of the techniques you develop in your overall structure.
2. Try to make all operations have worst-case runtime O(1) except for `extract-min`. Your implementation of `extract-min` will probably do a lot of work, but if you've set it up correctly the amortized cost will only be O(log *n*). This means, in particular, that you will only propagate the Δk's through the data structure in `extract-min`.
3. If you only propagate Δk's during an `extract-min` as we suggest, you'll run into some challenges trying to `meld` two lazy binomial heaps with different Δk's. To address this, we recommend that you change how `meld` is done to be even lazier than the lazy approach we discussed in class. You might find it useful to construct a separate data structure tracking the `meld`s that have been done and then only actually combining together the heaps during an `extract-min`.
4. Depending on how you've set things up, to get the proper amortized time bound for `extract-min`, you may need to define a potential function both in terms of the structure of the lazy binomial heaps and in terms of the auxiliary data structure hinted at by the previous point.

As a courtesy to your TAs, please start off your writeup by giving a high-level overview of how your data structure works before diving into the details.

Problem Four: Static Optimality in Practice (10 Points)

In lecture, we talked about a number of wonderful theoretical properties of splay trees. How well do they hold up in practice? How do they compare against standard library tree implementations and against weight-balanced trees, which (as mentioned in lecture) are never more than a factor of 1.5 off of a statically optimal tree? In this problem, you'll find out!

Download the assignment starter files from

`/usr/class/cs166/assignments/ps4/`

and extract them somewhere convenient. These starter files contain six implementations of different container classes, of which you'll need to make edits to two:

- `PerfectlyBalancedTree`: A perfectly balanced binary search tree built for a given set of keys. This class is given to you fully-implemented and you don't need to modify anything.
- `StdSetTree`: A collection backed by the `std::set` type, which is the C++ standard library's implementation of a sorted collection of items. It's usually, but not always, implemented as a red/black tree. You don't need to modify any of the code given to you for this type.
- `SortedArray`: A collection backed by a sorted array of keys. You don't need to modify this class.
- `HashTable`: A collection backed by the `std::unordered_set` type, which is the C++ standard library's implementation of an unsorted collection of items. It's usually, but not always, implemented with a hash table. We've already coded up this class for you.
- `WeightBalancedTree`: A weight-balanced tree. You're given a partial implementation of this class and will need to make two changes to it to improve its performance.
- `SplayTree`: A splay tree. You'll need to implement the function to perform lookups.

For simplicity, these types are designed to be constructed directly from a list of the keys they'll be storing. This means that the trees can, in some sense, "brace themselves" for what queries will then be made on them. It also means that you don't need to implement insertion or deletion logic and can focus on the more interesting bits.

To interactively test your trees, run the program `./explore` and follow the prompts. We've also included a program `./run-tests` that will do some basic correctness checks and then get timing information about how well your tree fares on a number of different tests.

Of the five types provided, you'll only need to modify two. Here's your to-do list:

i. The `WeightBalancedTree` that's provided to you contains a recursive function `treeFor` that builds a weight-balanced tree using a naive binary-search-based approach. This implementation as it currently stands is extremely slow for two reasons:

   1. The implementation periodically makes calls to a helper function `costOf` that determines the cost of making a particular split. That function is implemented in a way that takes time O(n), which is unacceptably slow. As was alluded to in Problem Two, there's a way to do O(n) preprocessing work so that querying for the cost of a split takes time O(1). Add some additional preprocessing logic so that `costOf` runs in time O(1).
   2. The implementation provided to you is based on the O(n log n)-time binary search algorithm from part (iii) of Problem Two. Modify the implementation so that it uses the O(n)-time simultaneous search strategy from parts (iv) and (v) of Problem Two.

(Continued on the next page…)

ii. The starter files contain a partial implementation of a `SplayTree` type. Finish the implementation by implementing the `contains` member function. This should perform a lookup and, since this is a splay tree, perform a splay.

In class we presented ***bottom-up splaying*** (where, after a query is performed, we applied a series of splays from the bottom of the tree up to the root). This approach is a little messy in practice because we need to keep track of the access path. There are a few ways to do this, all of which have some problems:

1. We could have each node store a pointer back up to its parent. This increases the overhead associated with each node, which impacts cache-friendliness and reduces the maximum size of the trees that we can hold in memory.
2. We could implement splaying recursively, using the call stack to remember the nodes we've visited. This can cause problems with stack overflows if the tree gets too imbalanced and we do an extremely deep lookup.
3. We could maintain a dynamically-allocated list of the nodes that we visited during a lookup. This requires extra memory allocations per lookup, which dramatically slows down the implementation.

There's an alternative way of implementing splaying called ***top-down splaying*** that's described in Section 4 of Sleator and Tarjan's original paper on splay trees. Top-down splaying works by reshaping the tree during the top-down tree search and requires only O(1) auxiliary storage space. You'll need to do a little legwork to figure out top-down splaying from Sleator and Tarjan's description. As with most research papers, the provided pseudocode skips over some C++-level details, which you will need to figure out by looking at the diagrams and through thorough testing.

There are two versions of top-down splaying presented in Section 4. Please implement the full rather than the simplified version of top-down splaying, since the simplified version is slower in practice.

Once you've implemented everything, edit the Makefile to crank the optimizer up to maximum, rebuild everything, and run the time tests. What do you see? Do any of the numbers surprise you?