CS166 Problem Set 3: Balanced Trees

[Your name goes here]

Problem One: Dynamic Overlap (8 Points)

This one is all coding! Make sure to submit your implementation on `myth`.

Problem Two: Range Excision (2 Points)

i. Design and describe an algorithm that, given a red/black tree T and two values k_1 and k_2, deletes all keys between k_1 and k_2, inclusive, that are in T. Your algorithm should run in time $O(\log n + z)$, where n is the number of nodes in T and z is the number of elements deleted. You should assume that it's your responsibility to free the memory for the deleted elements and that deallocating a node takes time $O(1)$.

Solution: Overview:

The general idea is to split the origina tree T into three parts, two which we want to keep and one which we want to delete. Then re-join the two parts we wish to keep.

Algorithm:

To begin, we assume $k_1 \geq k_2$. The first step is to run *split*(T, k_2) to produce T_p and T_r, where T_r contains all of the keys greater than k_2, and T_p all the keys less than or equal to k_2. We then run *split'*(T_p, k_1), a slightly modified version of *split*, so that it produces T_l and T_m. In this case, we'll have T_l contain all keys **strictly** less than k_1 (this is the slight modification, where instead of splitting into $\leq, >$, we split into $<, \geq$), and T_m now has all the keys greater than or equal to k_1 (and also in T_p, so they must be less than or equal to k_2).

We now simply delete the entire T_m tree (all nodes are freed), and run *join*(T_l, k_1, T_r) to produce a new merged tree T'. Finally, we perform a single delete on this tree to delete the value k_1. This final tree is the tree our algorithm returns.

Correctness:

Correctness mainly follows from the correctness of *split*, *join*, and single node deletion in red-black-trees. All of these algorithms were proved in class. The modification to *split* we use is also correct, since it consists of just changing the comparison operator.

Therefore, we know that the algorithm splits our original tree into three, T_l, T_m, T_r whose union of keys are the same as the original. Furthermore, T_m contains all keys with values between k_1 and k_2, inclusive. Therefore joining only T_l and T_r gives us a final tree which is missing these values, as required.

Runtime:

The run-time similarly follows from lecture. We run *split* twice on red/black trees with $O(n)$ nodes, for a runtime of $O(\log n)$. Deleting T_m takes time $\Theta(z)$, since it has z nodes and it takes $\Theta(1)$ time to delete each one. Performing *join* on T_l and T_m takes $\Theta(1 + |bh_1 - bh_2|) = O(\log n)$, since the difference in number of black nodes on any root-null path is at most the height of the origina tree, which is $O(\log n)$. The single delete on the red/black tree will then take $O(\log n)$.

Putting all of the above together, the total run-time is simply $O(\log n + z)$.

Problem Three: Deterministic Skiplists (8 Points)

Although we've spent a lot of time talking about balanced trees, they are not the only data structure we can use to implement a sorted dictionary. Another popular option is the ***skiplist***, a data structure consisting of a collection of nodes with several different linked lists threaded through them.

Before attempting this problem, you'll need to familiarize yourself with how a skiplist operates. We recommend a combination of reading over the Wikipedia entry on skiplists and the original paper "Skip Lists: A Probabilistic Alternative to Balanced Trees" by William Pugh (available on the course website). You don't need to dive too deep into the runtime analysis of skiplists, but you do need to understand how to search a skiplist and the normal (randomized) algorithm for performing insertions.

The original version of the skiplist introduced in Pugh's paper, as suggested by the title, is probabilistic and gives *expected* $O(\log n)$ performance on each of the underlying operations. In this problem, you'll use an isometry between multiway trees and skiplists to develop a fully-deterministic skiplist data structure that supports all major operations in *worst-case* time $O(\log n)$.

i. Briefly explain how to encode a multiway tree as a skiplist. Include illustrations as appropriate.

Solution: Your solution goes here!

To design a deterministic skiplist supporting insertions, deletions, and lookups in time $O(\log n)$ each, we will enforce that the skiplist always is an isometry of a 2-3-4 tree.

ii. Using the structural rules for 2-3-4 trees and the isometry between multiway trees and skiplists you noted in part (i) of this problem, come up with a set of structural requirements that must hold for any skip list that happens to be the isometry of a 2-3-4 tree. To do so, go through each of the structural requirements required of a 2-3-4 tree and determine what effect they will have on the shape of a skiplist that's an isometry of a 2-3-4 tree.

Solution: Your solution goes here!

Going forward, we'll call a skiplist that obeys the rules you came up with in part (ii) a ***2-3-4 skiplist***.

iii. Briefy explain why a lookup on a 2-3-4 skiplist takes worst-case $O(\log n)$ time.

Solution: Your solution goes here!

iv. Based on the isometry you found in part (i) and the rules you developed in part (ii) of this problem, design a deterministic, (optionally amortized) $O(\log n)$-time algorithm for inserting a new element into a 2-3-4 skiplist. Demonstrate your algorithm by showing the effect of inserting the value 8 into the skiplist given in the assignment handout.

Solution: Your solution goes here!

Congrats! You've just used an isometry to design your own data structure! If you had fun with this, you're welcome to continue to use this isometry to fgure out how to delete from a 2-3-4 skiplist or how to implement split or join on 2-3-4 skiplists as well.

Problem Four: Dynamic Prefix Parity (10 Points)

Consider the following problem, called the ***dynamic prefix parity problem***. Your task is to design a data structure that logically represents an array of n bits, each initially zero, and supports these operations:

- `initialize`(n), which creates a new data structure for an array of n bits, all initially 0;
- ds.`flip`(i), which flips the ith bit; and
- ds.`prefix-parity`(i), which returns the *parity* of the subarray consisting of the first i bits of the subarray. (The parity of a subarray is zero if the subarray contains an even number of 1 bits and is one if it contains an odd number of 1 bits. Equivalently, the parity of a subarray is the logical XOR of all the bits in that array).

It's possible to solve this problem with `initialize` taking $O(n)$ time such that `flip` runs in time $O(1)$ and `prefix-parity` runs in time $O(n)$ or vice-versa. (Do you see how?) However, it's possible to do significantly better than this.

i. Let's begin with an initial version of the data structure. Describe how to use augmented binary trees to solve dynamic prefix parity such that `initialize` runs in time $O(()\,n)$ and both `flip` and `prefix-parity` run in time $O(\log n)$. Argue correctness and justify your runtime bounds.

Solution: Overview:

We follow an augmentation process similar to that used for order stastics.

The general idea is to use an augmented binary search tree where the keys for each node are the numbers $0, \cdots, n-1$, and each node is augmented to track:

1. if it has been flipped on (a single bit that is either 0 or 1, initialized to 0)
2. the parity of the left child, and
3. the parity of the right child.

Generally speaking, the *parity* of a tree is just the XOR of all the bits contained therein.

Data Structure:

We can construct this tree in $O(n)$ time, since our keys start "sorted". This is what we do in `initialize`(n).

For ds.`flip`(i), we find the node whose key is i, and flip it's bit. We update all nodes along the root-leaf access path so their left child/ right child parities are consistent. This runs in $O(\log n)$ due to the search and updates.

For ds.`prefix-parity`(i), we perform a search for the node whose key is i. As we search, we keep track of the parity. Each time we follow the left child pointer, the parity tracker remains unchanged. However, when we follow the right child pointer, we update the parity tracker by XOR'ing it with the parity of the left-child and the parity bit stored in the parent.

Once we find the node, we update the parity tracker by XOR'ing it with the parity of the left-child and the parity bit stored in the parent. This is the parity of the subarray consisting of the first i bits.

This takes $O(\log n)$ since it's just a normal search with $O(1)$ operations per node.

Correctness:

The run-time complexity is correct since this fits our generalized augmented tree pattern (the left/right child parities can be computed by XOR'ing the child's parity with its left/right children).

The correcness of the algorithm follows from the fact that ds.`flip`(i) maintains all three invariants of the tree (the bit flip, and the parities of left/right children). If this is true, then ds.`prefix-parity`(i) begins at the root, and combines the parity of all indexes smaller that i as we traverse the tree.

ii. Explain how to revise your solution from part (i) of this problem so that instead of using augmented *binary* trees, you use augmented *multiway* trees. Your solution should have `initialize` take time $O(n)$, `flip` take time $O(\log_k n)$, and `prefix-parity` take time $O(k \log_k n)$. Here, k is a tunable parameter representing the number of keys that can be stored in each node in the multiway tree. Argue correctness and justify your runtime bounds.

We didn't explicitly discuss the idea of augmenting multiway trees in lecture, but we hope that the generalization isn't too tricky, especially since your tree never changes shape.

Solution: Overview:

We do the same thing as before (keys are values $0, \cdots n-1$) and *each* key is augmented with to track:

1. if it has been flipped on (a single bit that is either 0 or 1, initialized to 0)
2. the parity of all subtrees preceeding it (eg, instead of just the left subtree, include the left subtree plus all subtrees of keys in the same node which have a smaller value)
3. the parity of all subtrees following the key (eg, instead of just the right subtree, include the right substree plus all subtrees of keys in the same node which have a larger value)

Data Structure Modifications:

The construction is basically the same as before, with the additional stored information (all parities start at 0).

For ds.`prefix-parity`(i), very little changes. We perform a search for i, and as we walk down the tree, at each step we accumulate the parity of all subtrees to the left of the key we traverse (this is (2)). If we traverse to the right, we also include the parity bit of the key itself, otherwise not (this is (1)). We do the exact same thing when we finally find the node (as if we'd traversed right). Since we're doing $O(1)$ time per node, this will take $O(\log_k n)$ (just a search).

For $ds.$`flip`(i), things get a bit more interesting. We similarly perform a search for i. We'll have to update the stored parities of the nodes along the root-leaf access path. There are $O(\log_k n)$ such nodes. For each node, we'll have to update $O(b)$ keys (we have to update all keys at each node). All keys to the right will have (2) updated, and all keys to the left of our key will have (3) updated. Each of these updates to a key can be done in constant time, since it only depends on its subtrees. This gives a total running time for $ds.$`flip`(i) to be $O(k \log_k n)$.

Correctness:

Correctness follows from a similar argument as BST. Note that each key knows the parity of all subtrees rooted at the same node smaller than itself. As we traverse, we accumulate this information. This means that by the time we've found i, we've accumulated the parity for the first i bits.

We note that `flip`(i) keeps our three augmented properties consistent by updating all affected nodes when a bit is flipped.

Combined, thse two properties lead to the correctness of our data structure.

Runtime:

This was argued as the algorithm was described.

iii. Using the Method of Four Russians, modify your data structure from part (i) so that `initialize` still runs in time $O(n)$, but both `flip` and `prefix-parity` run in time $O(\log n / \log\log n)$.

This last step is probably the trickiest part. Here are some hints:

- It might help to think of the Method of Four Russians as a divide, precompute, and conquer approach. That is, break the problem down into multiple smaller copies of itself, precompute all possible answers to the smaller versions of those problems, then solve the overall problem by looking up precomputed answers where appropriate. That is, this will be less about explicitly sharing answers to subproblems and more about having the answers to all possible small problems written down somewhere. Do you see how your solution to part (ii) implicitly breaks the bigger problem down into lots of smaller copies?
- Remember that $\log_k b = \log b / \log k$ thanks to the change-of-basis formula.
- All basic arithmetic operations are assumed to take time $O(1)$. However, floating-point operations are not considered basic arithmetic operations, nor are operations like "count the number of 1 bits in a machine word" or "find the leftmost 1 bit in a machine word."
- An array of bits can be thought of as an integer, and integers can be used as indices in array-based lookup structures.
- Be precise with your choice of block size. Constant factors matter!

As usual, argue correctness. Be sure to justify your runtime bounds precisely – as with the Fischer-Heun structure, your analysis will hinge on the fact that there aren't "too many" subproblems to compute the answers to all of them.

Solution: Overview

The main insight is that each node in the multi-way tree is itself a prefix parity problem.

The Subproblems

Consider each node in the multi-way tree. As designed, it consists of k keys, each with associated with 2 important bits. The first bit is simply the bit represented by the key, and the second bit is the parity of all subtrees to the left of the key. Instead of having the second bit be the parity of all subtrees to the left of the key, we can instead consider a single bit per key which specifies the parity of the entire subtree rooted at the key. Transforming this size-k array of bits into our original bit can be done by (1) answering the prefix-parity problem for all subtrees left of our key, and (2) XORing this with the parity of the left subtree of our key.

Therefore, by thinking of the problem in this new augmented way (one parity bit-per tree rooted at key, one bit-per tree rooted at a given node, and one bit per key), we have reduced our original problem into many "smaller" sub-problems (e.g, accomplishing (1)).

Pre-Computing for small k

Therefore, let us consider problems (1) and (2) for small k. There are 2^k possible bit-arrays of size k, and for each of these, we can ask k distinct prefix-parity queries. As such, we need to pre-compute the answer for $k2^k$ problems. If we treat the bit-arrays as integers, we can store these answers in an array and do constant time lookups. As such, `prefix-parity`(i) will still take $O(\log_k n)$ time.

However, the update operations required when when doing `flip`(i) can now be done in constant time per-node, rather than $O(k)$ as was the case with the previous approach. This is because the update now consists simply of flipping the bit associated with the key (this is just a simple bitwise-OR operation), followed by flipping the bit for the subtree rooted at the key (if we treat the bit-array as an integer, this can also be done with a simple bitwise-OR operation by additionally keeping a mask for each key), and finally flipping the bit for the node (a simple bitwise-OR).

Choosing k

As argued above, with our $k2^k$ pre-computed answers to the prefix-parity problem, our data structure's runtime for both `flip`(i) and `prefix-parity`(i) is $O(\log_k n) = O\left(\frac{\log n}{\log k}\right)$. However, our `initialize`($n$) time-complexity is now $O\left(n + k2^k\right)$. It takes linear time to construct the multi-way tree, and it takes $O(k)$ time to pre-compute each all possible answers for any prefix-parity problem for a bit-array of size k, of which there are 2^k (using dynamic programming).

Therefore, we need to chose a k small enough such that $k2^k = O(n)$, but also have

$k = \Theta(\log n)$ so that our query-time is $O\left(\frac{\log n}{\log\log n}\right)$. Thankfully, we can just use:

$$k = \frac{\log n}{2}$$

With this choice of k, our complexity for

`initialize`(n)

is:

$$O\left(n + k2^k\right) = O\left(n + \log n\sqrt{n}\right) = O(n)$$

and for both `flip`(i) and `prefix-parity`(i), we have:

$$O\left(\frac{\log n}{\log k}\right) = O\left(\frac{\log n}{\log\log n}\right)$$

which is precisely our desired runtime.

Pat yourself on the back when you finish this problem. Isn't that an amazing data structure?