Assignment 5: Splay Trees and Tries

This problem set explores splay trees and and tries. In the course of doing so, you'll explore static optimality from both a theoretical and practical perspective, and will learn about a new data structure, the *weight-balanced tree*, with applications to both static optimality and tries. We hope that this will give you a much deeper understanding of what we've explored this week!

Working in Pairs

We suggest working on this problem set in pairs. If you work in a pair, you should jointly submit a single assignment, which will be graded out of 10 points. If you work individually, the problem set will be graded out of 9 points, but we will not award extra credit if you earn more than 9 points.

Due Wednesday, May 7 at 2:15PM at the start of lecture.

Problem One: Static Optimality in Theory (3 Points)

Let $S = \{ x_1, x_2, \ldots, x_n \}$ be a set where each element has an associated probability p_i. Suppose that you are drawing a sequence of values from S using the probabilities p_i. You and I are communicating over a network in which we are only allowed to send symbols drawn from some alphabet Σ. Your goal is to send me a string in Σ^* describing the sequence you drew from S.

To do so, we agree in advance a function $f : S \to \Sigma^*$ called a *coding function* that assigns to each string $x \in S$ a string $f(x)$ called the *code word*. To send me the sequence of values you drew, you compute the code word for each value, concatenate all those strings together, and then send me the overall string. Assuming we've chosen the code words $f(x)$ intelligently, I can then decode the stream.

For example, suppose that $S = \{ a, b, c, d, e, f \}$ and $\Sigma = \{0, 1\}$. One possible set of code words we could use would be the following:

$$a = 000 \quad b = 001 \quad c = 010 \quad d = 011 \quad e = 100 \quad f = 101$$

Now, If I receive

011100101000010100

I can decode the message by breaking it apart into blocks of three bits each:

011 100 101 000 010 100

and then inverting the code mapping to get back

d e f a c e

In the above case, all of the code words had the same length. However, there's no reason that this must be the case. For example, we could also use these code words:

$$a = 1 \quad b = 01 \quad c = 001 \quad d = 0001 \quad e = 00001 \quad f = 000001$$

Now, if I get the message

0001000010011000001

I could split it apart to get the code words back:

0001 00001 001 1 000001

Which then reads off as

d e c a f

Not all coding functions will work, though. For example, consider these code words:

$$a = 11 \quad b = 00 \quad c = 10 \quad d = 1 \quad e = 0 \quad f = 01$$

To send the message *cafe bbd*, you would send me

101101000001

When I receive this message, I might, however, cut it apart like this:

1 0 11 1 00 0 0 01

Which decodes as *d e a d b e e f*, definitely not the message you intended!

We'll say that a coding function is a *legal* coding function if any coded message sent using that coding system can be uniquely decoded back to a single sequence of code words. One way to ensure this is to use a *prefix code*, in which no code word is a prefix of any other code word. If you've seen Huffman coding before, Huffman coding is one possible way to create a prefix code for a set of elements, though it's not the only way.

There are theoretical limitations to the sizes of the code words used. Given a probability distribution in which each element x_i has probability p_i, the *Shannon entropy* of the distribution, denoted H, is

$$H = \sum_{i=1}^{n} (-p_i \lg p_i)$$

The entropy of the distribution measures how much information, in bits, is generated by sampling from that distribution. At one extreme, if all n elements have probability $1/n$ of being chosen, then the above summation becomes

$$H = \sum_{i=1}^{n} \left(-\frac{1}{n} \lg \frac{1}{n}\right) = \sum_{i=1}^{n} \frac{\lg n}{n} = \lg n$$

So lg n bits of information are generated by sampling from the distribution. Intuitively, this makes sense – there are n possible outcomes, each are equally likely, so we would need lg n bits to write out the index of the entry that was generated. At the other extreme, if one element has probability 1 and the remaining elements have probability 0, then

$$H = \sum_{i=1}^{n} (-p_i \lg p_i) = \sum_{i=1}^{n} 0 = 0$$

So 0 bits of information are generated. This makes sense because we already knew what element was going to be drawn from the distribution.

Shannon's source coding theorem says that if S is a set where each element x_i has probability p_i of being chosen, then for any legal coding function $f : S \to \Sigma^*$, we have

$$\mathrm{E}[|f(x_i)|] \geq H / \lg |\Sigma|$$

Here, the expectation is taken over a random choice of x_i drawn from S using probabilities p_i.

Consider an arbitrary BST whose keys are S and where the access probabilities are p_1, …, p_n. Using Shannon's source coding theorem, prove that the expected number of nodes visited in any successful lookup on that tree is at least H / lg 3, where H is the Shannon entropy of the access distribution.

Some hints:

1. Write an expression for the expected number of nodes visited when doing a lookup in a fixed BST with the given keys and probabilities in terms of the probabilities and the path lengths of the nodes.
2. Show how to turn any BST into a legal coding function over { 0, 1, 2 } such that the lengths of the code words are closely related to the path lengths in the BST.
3. Connect your results from (1) and (2) together.

Problem Two: Static Optimality in Practice (3 Points)

Splay trees have the *static optimality property*: the amortized cost of a successful lookup in a splay tree is O(1 + *H*), where *H* is the Shannon entropy of the access probabilities, as long as each key is accessed at least once. Your result from Problem One says that this is optimal to within a constant factor, so we would expect splay trees to perform well compared to a statically optimal BST. In this problem, you'll compare splay trees against another type of statically optimal BST called a *weight-balanced tree* to see whether theory matches practice.

Suppose that you have a set of keys $x_1, \ldots, x_n$ and associated positive weights $w_1, \ldots, w_n$. We'll define the weight of a tree to be the sum of the weights of all the keys in that tree. A *weight-balanced tree* is then defined as follows: the root node is chosen so that the difference in weights between the left and right subtrees is as close to zero as possible, and the left and right subtrees are then recursively constructed using the same rule. In 1975, Kurt Mehlhorn proved that weight-balanced trees, where the weights on each node correspond to the access probabilities, are within a factor of 1.5 of the statically optimal BST for those nodes given those access probabilities (pretty nifty!)

There is an O(*n* log *n*)-time algorithm for constructing weight-balanced BSTs. Begin by sorting the keys into ascending order and tagging each with their weights. In time O(*n*), compute the total sum of the weights. Then, use the following recursive process:

- Scan from both ends of the array inward, summing up the weights as you go, until a node is found with the optimal balance between its left and right subarrays. You can detect this by looking for a spot where the absolute value of the weight difference increases; the spot right before that is the optimal splitting point.
- Make that node the root of the tree, and repeat this process on the halves of the array.

The runtime of this recursive phase is O(*n* log *n*). Intuitively, at each level, the work done is proportional to the smaller of the two subarrays searched. Using reasoning similar to what you saw in the decremental connectivity structure from Problem Set Three, you can show that each element is part of a smaller array at most O(log *n*) times, so the total work done is O(*n* log *n*).

Using the Java starter code available at

```
/usr/class/cs166/assignments/ps5/
```

implement the `WeightBalancedTree` and `SplayTree` classes and use the provided driver code to test them against one another. You might find your results interesting.

Some advice:

1. Make sure not to recompute the sum of the weights in the subarrays at each level of the recursion, since otherwise the runtime might degrade to $\Theta(n^2)$ on some inputs. (Do you see why?)
2. Your solutions should not cause stack overflows during searches, though you don't need to worry about stack overflows when initially constructing the trees.

Problem Three: Trie Representations (3 Points)

In a trie, each node stores a collection of pointers to child nodes. The performance of the basic operation on tries and the space efficiency of a trie heavily depend on what data structure is used to represent those pointer collections. This problem explores several possibilities and their tradeoffs.

In what follows, let N denote the total number of nodes in a trie and n the number of strings it contains. Denote the alphabet of the trie as Σ. When doing a query, let w be the query string. In the case of a successor search, let $\sigma(w)$ denote the successor of string w in the trie.

i. **(1 Point)** Fill in the table below by writing the time complexities of the specified operations and the space complexity of the overall trie assuming that the indicated data structure is used to store the child pointers. In the case of a hash table, assume that all hash table operations run in their expected, amortized cost of O(1) per operation.

	lookup	***insert***	***delete***	***successor***	Space
Array of Σ pointers, one for each character.	$O(\|w\|)$				$\Theta(N \cdot \|\Sigma\|)$
Red/Black Tree					
Hash table (assume each hash table operation runs in its expected runtime)					

When filling out this table, you can assume that allocating a block of uninitialized memory of any size takes time O(1) and that deallocating a block of memory of any size takes time O(1).

Now, suppose that you have a *static trie* in which the set of words is fixed. Consider the following representation of such a trie: each node in the trie stores its children in a weight-balanced tree (see Problem Two for details). We set the weight of each BST node to be the number of strings in the subtrie corresponding to that node. The space usage of this data structure is the same as in the case of a red/black tree, which as you saw in part (i) is much better than the space usage of an array-based trie. Interestingly, however, the choice of a weight-balanced tree significantly improves lookup times.

ii. **(1 Point)** Suppose that the total weight in a weight-balanced tree is W. Prove that the left subtree and right subtree of a weight-balanced tree each have weight at most $7W / 8$.

iii. **(1 Point)** Prove that any lookup in this representation of a trie requires time $O(|w| + \log n)$. Successor queries also have this runtime, but you don't need to prove this.

Problem Four: Course Feedback (1 Point)

We want this course to be as good as it can be and would really appreciate your feedback on how we're doing. For a free point, please take a few minutes to answer the course feedback questions available at https://docs.google.com/forms/d/1hNeZbZRVUDUpnI5Fa-sEKDU1pYF4Bbpzfoz_BzUL85Q/viewform. If you are submitting in a group, **please have each group member fill this out individually**.