Problem Set 3: Balanced Trees

This problem set explores augmented binary trees, data structure isometries, and some of the more advanced operations on binary search trees. By the time you've finished working through this problem set, you'll have a much richer feeling for just how powerful this particular family of techniques can be and how they can be used to design simple, elegant, and fast data structures.

Due Thursday, April 28 at the start of lecture.

Problem One: Dynamic Maximum Single-Sell Profit (3 Points)

The ***maximum single-sell profit problem*** (or ***MSSPP***) is a classic algorithms question given as follows. You're given a sequence of n real numbers $p_1, p_2, \ldots, p_n$ representing the prices of some object over a period of time. The prices are listed in chronological order, so, for example, the value p_3 represents the price of the object at some time between p_2 and p_4. You are required to buy the object at some particular time point p_i and to sell it at some time p_j (where $i \leq j$) in a way that maximizes $p_j - p_i$, the profit you've made you've made. Note that you are allowed to sell the object back as soon as you buy it for zero profit if you'd like.

The ***dynamic MSSP problem*** is similar to the one above, except that the underlying list of object prices is allowed to change over time. Specifically, new object prices might be added between existing time points, and prices in the range might change.

Design a data structure that supports the following operations:

- *ds*.`insert`(*i*, *p*), which inserts a new time point with price *p* at index *i* in the list.
- *ds*.`update`(*i*, *p*), which replaces the price at index *i* with the new price *p*.
- *ds*.`remove`(*i*), which removes the price at index *i* from the list.
- *ds*.`mssp`(), which returns the maximum single-sell profit for the current list.

Your data structure should support the first three operations in time O(log n) and the `mssp` operation in time O(1), where n is the total number of elements in the sequence.

Problem Two: Range Excision (2 Points)

Let T be a red/black tree with n elements and let $k_1 \leq k_2$ be two arbitrary values. Give an O(log n + z)-time algorithm for deleting all keys from T that are between k_1 and k_2, where z is the total number of elements deleted this way. In designing your algorithm, you should assume it's your responsibility to free the memory for the deleted elements and can assume that deallocating a block of memory takes time O(1).

Problem Three: Fenwick Trees (4 Points)

Consider the following problem: you are given a fixed-size, 1-indexed sequence of n integers, all initially zero. Your task is to support the following operations as efficiently as possible:

- *ds*.`increment`(*i*, *x*), which adds *x* to the *i*th element of the sequence, and
- *ds*.`cumulativeFrequency`(*i*), which returns the sum of the elements from index 1 to *i*, inclusive.

It's possible to solve this problem naïvely such that `increment` runs in time O(1) and `cumulativeFrequency` runs in time O(n) or vice-versa (do you see how?), but by being a bit more clever with our data structures we can significantly improve on these time bounds.

i. Describe how to solve this problem using augmented binary trees so that initialization takes time O(n) and both `increment` and `cumulativeFrequency` queries run in time O(log n).

This particular application of augmented trees is interesting in that the tree shape never changes. As a result, rather than representing the tree as an explicit tree structure, you can represent it implicitly in a similar spirit to how binary heaps (a tree structure) are often represented implicitly using arrays. The resulting data structure is called a ***Fenwick tree*** or a ***binary indexed tree*** and is remarkably fast in practice.

ii. Implement the data structure you described in part (i) implicitly in an array without storing any pointers between nodes. To do so, download the requisite starter files from

    `/usr/class/cs166/assignments/ps3`

    and implement the appropriate types and functions in `fenwick-tree.c`. As before, to receive full credit, your code should compile with no warnings and should not have any memory errors. We'll test your code on the `corn` cluster.

    The biggest challenge you're likely to encounter in implementing this data structure is determining how to lay out the elements of the binary tree so that you can move from node to node in the tree by performing calculations on the indices. As a starting point, think about how you'd approach this problem in the case where n is of the form $2^k - 1$, then generalize from there. As a hint, consider storing nodes in an inorder fashion (as opposed to, say, the level-by-level fashion used in binary heaps). When generalizing your answer to work for arbitrary sizes, we recommend structuring the tree so that the root node always corresponds to an index that's a power of two.

    When implementing your data structure, you'll need to be able to navigate around in the implicitly-represented tree. The calculations you'll need to do are a bit more involved than in binary heaps, but don't require too much code. As a hint, write out the (1-indexed) indices of the nodes in binary, look at the nodes you'll need to touch or update to implement each operation, and see if you notice a pattern. You may find it useful to think about what the bitwise operation `x & -x` returns if `x` is a nonzero unsigned integer.

iii. Give a write-up of your implementation. In particular, explain how the specific array and bitwise operations you're performing correspond to analogous operations on the balanced tree representation you developed in part (i) of this problem. Given that the code you'll ultimately end up writing will probably bear little resemblance to the higher-level data structure you described in part (i), please be as specific as possible in your explanations.

Problem Four: Deterministic Skiplists (4 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 time O(log *n*).

i. There is a beautiful isometry between multiway trees and skiplists. Describe how to encode a skiplist as a multiway tree and a multi-way tree as a skiplist. Include illustrations as appropriate.

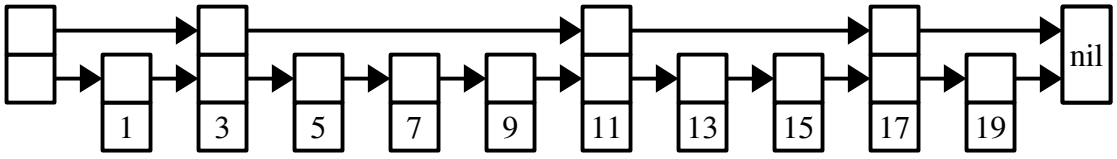
To design a deterministic skiplist supporting insertions, deletions, and lookups in time O(log *n*) each, we will enforce that the skiplist always is an isometry of a 2-3-4 tree.

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 that will have on the shape of a skiplist that's an isometry of a 2-3-4 tree.

Going forward, we'll call a skiplist that obeys the rules you came up with in part (ii) a ***1-2-3 skiplist***.

iii. Briefly explain why a lookup on a 1-2-3 skiplist takes worst-case O(log *n*) time.

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 1-2-3 skiplist. Demonstrate your algorithm by showing the effect of inserting the value 8 into the skiplist given below:



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 figure out how to delete from a 1-2-3 skiplist or how to implement split or join on 1-2-3 skiplists as well.