Problem Set 3 Solutions

Problem One: Dynamic Maximum Single-Sell Profit

We can implement this data structure using augmented order statistic trees. Each tree node will be augmented with the following additional information:

- The minimum value in its subtree.
- The maximum value in its subtree.
- The optimal MSSP answer in its subtree.

As we saw in class, augmenting a tree with the minimum and maximum values in the tree can be done using our existing framework, since those values can be computed from the node value itself and the values in its subtrees. The optimal MSSP answer can be computed as the maximum of the following values: the left subtree's MSSP (if the buy and sell are purely before the current time point), the right subtree's MSSP (if the buy and sell are purely after the current time point), the maximum value in the right subtree minus the minimum value in the left subtree (if the buy and sell are from before and after the current time point), the current value minus the left subtree's minimum (if we buy earlier and sell now), the left subtree's maximum value minus the current value (if we buy now and sell high later), and zero (if we immediately buy and then sell back).

We can implement `insert` and `delete` by using traditional red/black tree operations, except that when we'd normally do a search by key, we instead do the search by index using the same technique developed for order statistic trees and then updating cached values using the techniques described in lecture. We can implement `update` by doing a lookup by index and changing the value there (and then updating cached values using the normal augmented tree techniques). These operations run in time O(log n) since they do a baseline O(log n) work, plus O(log n) extra work to update cached values. Finally, we can implement `mssp` in time O(1) by reading the cached value from the root.

Problem Two: Range Excision

One possible algorithm is as follows. First, we split T into two trees T_0 and T_{12} by using the split operation on k_1. Next, split T_{12} into two trees T_1 and T_2 by using the split operation on k_2. This means that all keys strictly less than k_1 are in T_0 and all keys greater than k_2 are in T_2. This collectively takes time O(log n). Next, iterate over all the keys in T_1 and delete all of them, taking time O(z). Then, delete the minimum value from T_2 and use it to join T_0 and T_2 back together, taking an additional O(log n) time. The total time required is then O(log $n + z$), as required.

Problem Three: Fenwick Trees

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).

Begin by building a perfectly balanced BST where each node represents one of the elements in the sequence (this can be done in time O(n) using a simple recursive divide-and-conquer algorithm). Augment each node in the tree with the sum of the values at that node and in its left subtree, along with extra order statistics information so that we can do lookups by index in time O(log n). Since the tree shape never changes, we don't need to worry about preserving these values during a rotation.

To implement `increment`(i, x) we find the node at index i, then add x to that node and to all nodes on the access path where we traversed leftward during the lookup (these nodes correspond to all the nodes whose left subtree's weight changed). To implement `cumulativeFrequency`(i), we search for i in the tree. As we do, we maintain a counter (initially 0), and every time we descend to the right we increment the counter by the cached weight at the node we just took a step right at. We also include the sum for the node corresponding to i itself. This works because every time we descend to the right, we need to include the combined weight of all the nodes we didn't visit, which are those in the left subtree, whereas when we descend to the left we're skipping over nodes that don't contribute to the total.

iii. Give a write-up of your implementation.

As our tree shape, we'll pick a perfect binary tree whose size is as small as possible to hold all of our sequence elements. We'll then remove from that tree any node whose index is past the end of the sequence. We'll then represent this tree implicitly in an array in an inorder fashion.

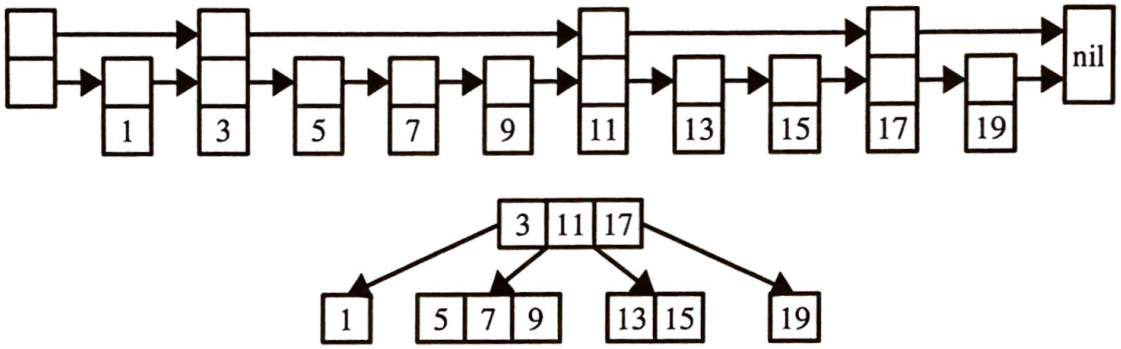
We'll actually navigate this tree *in reverse*, moving from the leaves up to the root rather than going the other way around. To do an update, we start at the appropriate index, then need to update the cached frequencies in all the nodes we'd visit in the leaf-root path where we follow a left pointer. It turns out that we can jump directly from a node to its lowest ancestor where we moved left by dropping the least-significant-set 1 bit in the node index, so we can sit in a loop and repeatedly subtract out `index & -index` from the current index until we hit the root of the tree. (To see why this works, notice that `index & -index` yields a number whose binary representation consists of a single 1 bit at the index of the least-significant 1 bit of `index`.)

To determine cumulative frequencies, we need to walk from the leaf up to the root, accumulating the values at all the nodes where we moved to the right. If you inspect the binary representations of the indices of the nodes visited this way, you'll notice that they're determined by taking the last block of 1 bits in the current index, flipping them all to 0's, then flipping the preceding bit to a 1. This corresponds to adding in the value `index & -index` to the current index, so we can navigate up the tree to the root by adding in this value until we eventually walk off the tree.

Problem Four: Deterministic Skiplists

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.

The isometry is given as follows: each skiplist node corresponds to a node in a multiway tree whose height is given by the height of the skiplist node. All nodes at the same height with no intervening nodes of a larger height are grouped together into a single multiway tree node. Here's an example illustration:



ii. 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.

The first rule for 2-3-4 trees is that every node must have exactly 1, 2, or 3 keys in it. Mapping this to our skiplist, this means that if we find a group of nodes that are all at the same height and have no intervening nodes of a larger height between them, then that group must have either 1, 2, or 3 nodes in it.

The second rule for 2-3-4 trees is that all root-null paths must pass through the same number of nodes. In a 2-3-4 tree, this means that in each non-leaf node, in the space before the first key, after the last key, and between any two keys, there will be a child node. Mapping this to skiplists, we see that this means that if we can find a group of keys at the same height $h \geq 1$ with no intervening nodes of larger heights, then in the gap before the first key, after the last key, and between any of those two keys, there will be at least one key whose height is exactly h-1.

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

We know that at the top level of the skiplist there can be at most three nodes, since if there were more nodes than that, we'd have to have an even higher node between them. This means that a skiplist lookup will make at most three comparisons at the top level. When we drop down a level below that, we can similarly say that we need to only consider at most three keys, since there can be at most three keys of height h-1 between any two nodes of height h. This pattern continues all the way down the skiplist, so the work done is proportional to the height of the skiplist. The isometry with 2-3-4 trees guarantees that this height is O(log n), so the total lookup time is O(log n).

iv. Design a deterministic, (optionally, amortized) O(log n)-time algorithm for inserting a new element into a 1-2-3 skiplist.

Our insertion procedure is extremely similar to what we'd do for a normal 2-3-4 tree insertion. Begin by inserting a node of height 1 into the skiplist by doing a normal lookup and inserting the element at the appropriate spot. Now, count how many nodes there are at height 1 in its section without crossing nodes of height 2. If there are 1, 2, or 3, we're done. Otherwise, there will be exactly four. Take the middle element, increase its height by one, and wire it in between its boundary nodes. Then repeat this process at the next layer, promoting a node if there are four nodes of the same level without intervening larger nodes, etc., potentially increasing the height of the entire skiplist.

The initial insertion takes time O(log n) courtesy of our argument from part (iii). At each layer, we do O(1) work to determine how many elements there are of each height, and overall we make at most O(log n) height increases. If we represent the pointers at each node as dynamic arrays, then this makes at most O(log n) appends to those dynamic arrays and each append is accompanied by a constant number of pointer rewirings. Overall, this takes amortized time O(log n). *(Fun challenge: Show that it's actually worst-case O(log n).)*