6.851: Advanced Data Structures, Fall 2017

Prof. Erik Demaine, Adam Hesterberg, Jayson Lynch

Problem Set 1 Solutions

Due: Wednesday, September 13, 2017 at noon

Problem 1.1 [Here Trees]. Describe and analyze a data structure for storing an ordered set of keys, initially empty. Your data structure should support the following operations and time bounds, where n is the number of keys in the set currently represented by the data structure:

(a) **predecessor**(k) in $O(\log n)$ time: find the largest key $\leq k$ that is in the set, and return a pointer to the node in the data structure representing that key.

(b) **successor**(k) in $O(\log n)$ time: symmetrically

(c) **predecessor-of-here**(k, x) in $O(1)$ time: given a pointer to the node x representing a key k, find the largest key $< k$ that is in the set, and return a pointer to the node representing that key. (If there is no such key, return null.)

(d) **successor-of-here**(k, x) in $O(1)$ time: symmetrically

(e) **insert-after**(k, x) in $O(1)$ amortized time: given a pointer to the node x representing the largest key $< k$, insert k into the data structure (if it doesn't already exist), and return a pointer to the node representing k.

(f) **delete-here**(k, x) in $O(1)$ amortized time: given a pointer to the node x representing a key k, delete k from the data structure.

(g) **split-here**(k, x) in $O(1)$ amortized time: given a pointer to the node x representing a key k, destructively split the data structure into two, one containing all keys $\leq k$ and the other containing all keys $> k$. (Future operations should depend on the newly split sizes.)

(Each cost can be amortized over all operations, not just split-here operations. You should already be comfortable with amortization from a prerequisite class. If not, we recommend that you talk with the course staff for advice.)

Hint: Start from (a, b)-trees.

Hint: In defining a potential function to amortize split-here, think about what changes about the split edges.

Solution 1: This solution was written by Mahi Nur Muhammad Shafiullah working with Thuy Duong Vuong.

We model the data structure as an (a, b) tree (where each node has between a and b keys, with $2a < b$), along with a doubly linked list, both containing all the keys. Additionally, nodes in the linked list contains pointers to associated nodes in the (a, b) tree and vice versa. In the linked list, the elements are in a sorted order.

We define a potential function, Φ, that we will use to analyze the time complexity of the operations:

$$\Phi(T) = 4c\Big(\#\text{ nodes with } a \text{ keys}+\#\text{ nodes with } b \text{ keys}\Big)+c\sum_{\text{keys}\in T}\#\text{ neighbors of key at the same node.}$$

Clearly, this function is 0 at the beginning and positive at any other time.

We will now describe the operations of the DS and analyze their amortized runtime.

- **Predecessor:** Run usual search at the search tree, which takes $O(\log n)$ time.
- **Successor:** Like before, run usual search at the search tree, which also takes $O(\log n)$ time.
- **Predecessor-of-here:** Given the pointer to a node on the tree or the linked list, go to the node on the linked list and go to the predecessor node. Takes $O(1)$ time.
- **Successor-of-here:** Symmetric to Predecessor-of-here.
- **insert-here:** Do a insert of nodes denoting key k at the search tree and the linked list right after the node x. Then, if the node containing k is overfull, do the typical (a, b)-tree splitting of tree nodes (possibly recursively).

  On each level where there is a node split, it takes $O(1)$ time to split the tree node. But this makes our potential go down by $4c$ for the first half of Φ and up by $2c$ on the second half of the function. So overall, we get a $-2c$ on each split, which we can use to do the work for the split. Finally, on the base case, the potential can increase by 2, but this should be covered by the change in potential overall. So, insert-here should be amortized $O(1)$.
- **delete-here:** We can delete the pointed node from the linked list, and then try to delete the node x from the search tree. If we needed to move keys around from siblings, the potential function doesn't change and it only takes $O(1)$ work. Otherwise, if we merge two tree nodes, our potential must go down on every level where we recursively merge because number of a-nodes go down, so this is still $O(1)$ amortized.
- **split-here:** We can start from the node x that has the key k on which we are splitting. Then, going upwards from there, we can split every tree node into two tree nodes, one with all keys $< k$ and another with all keys $> k$. Splitting the tree in such a way will give us two different trees (which may be imbalanced). For each tree, on each level, if it is imbalanced we can move keys around without making any node have a or b keys. Then, for each level, we know that the keys right around the split edges have lost one neighbor each at the same level. So, we will decrease the potential at each level by $2c$, and in the end this decrease in potential decrease will pay for our work on rebalancing and splitting the tree nodes. So overall, this will be a $O(1)$ amortized operation as well.

  Finally, we will need to split our linked list as well, but as it is sorted and we have a pointer to x, that operation is also $O(1)$.

Overall, this data structure gives all the necessary operations in the required runtime, and thus we know that this will work for a suitable choice of c.

Solution 2: This solution was written by Sanjeev Murty (smurty) working with lawli, twang6, seanshi, endrias, and nshafiul.

We use a B+ tree where each non-root internal node must have between d and $2d$ (where $d \geq 2$) children at all times, and the keys are the leaves of the tree. In addition, we maintain that each internal node contains a pointer to the keys in its subtree with the minimum and maximum values. We also maintain that the leaves (keys) will be linked to their predecessors and successors in a doubly linked list. Since we will be analyzing a split operation, we think of the data structure as a forest, and use the following potential function for all the amortized analysis (where $u.c$ is the number of children of node u, $t.n$ is the number of nodes in tree t, F is the current forest, and γ, β are constants we'll set later):

$$\Phi(F) = \gamma \cdot \sum_{\text{internal node } u} \left(u.c - \frac{3d}{2}\right)^2 + \beta \sum_{\text{tree } t} t.n - \lg(t.n) \tag{1}$$

Clearly, $\Phi(F) \geq 0$, since the first summation is a sum of squares, and $x > \lg x$ for $x \geq 0$ (where we define $\lg 0 = 0$), and Φ for the empty data structure is 0. So this is a valid potential function.

(a), (b) Starting from the root, at each level, we use the min/max of each subtree to determine which subtree to recursively search (choosing the subtrees whose interval contains k, and if at the leaves, choosing the predecessor/successor of k in $O(1)$). This requires an $O(d) = O(1)$ search at each level, so the total runtime is $O(\log n)$, since the height of the tree is $O(\log n)$ because we guaranteed that each internal node has at least $d \geq 2$ children.

(c), (d) We simply use the doubly linked list among the keys (as mentioned earlier) to determine the predecessor/successor (or null) of x in $O(1)$.

(e) Let Redistribute-Grandchildren(u) be a method that redistributes its grandchildren among its children so that all its children have either $\lfloor \frac{g}{u.c} \rfloor$ or $\lceil \frac{g}{u.c} \rceil$ children (where g is the number of grandchildren of u). This can be easily done in $O(d^2) = O(1)$ time (while maintaining the min/max pointers of the children of u). We use the standard B+ tree insertion ("splitting" nodes when they exceed capacity, and propagating up), except we call Redistribute-Grandchildren at each level before considering splitting nodes, and after splitting nodes. Updating min/max pointers can easily be done when splitting nodes in $O(1)$. Thus, each time we split a node (and redistribute), the potential decreases by $\Theta(d^2)$ (since $\Theta(d)$ siblings that had $\geq 2d - 2$ children now have 1 less child, which decreases their $\left(u.c - \frac{3d}{2}\right)^2$ term in the potential by $\Theta(d)$). We set γ so that this potential decrease offsets the cost of "splitting" nodes and redistributing grandchildren at each level. The addition of an element only increases the second term in the potential by $O(1)$. Updating the doubly linked list can easily be done in $O(1)$. If the inserted key is the min/max of its parent's subtree, simply overwrite the old min/max's key with the new key, and insert a new node with the old key (we can ensure this is possible by setting $d \geq 3$). This can be done in $O(1)$. Thus, the amortized cost is $O(1)$.

(f) We use the standard B+ tree delete (merging sibling nodes when one gets too few children, and propagating up), except we call Redistribute-Grandchildren at each level before considering merging nodes, and after merging nodes. Updating min/max pointers can easily be done in $O(1)$ when merging nodes. Similar to part (e), merging a node with a sibling and redistributing the children among siblings decreases the potential by $\Theta(d^2)$. Thus, the potential decrease again offsets the cost of merging nodes at each level. Updating the doubly linked list can easily be done in $O(1)$. We use the same trick as in part (e) to maintain the min/max pointers when the min/max of a subtree is deleted in $O(1)$. Thus, the amortized cost is $O(1)$.

$\boxed{\textbf{(g)}}$ Starting from the given node x, we keep following parents until we reach a node whose subtree either contains the min or max of the whole tree. If it contains the max, we'll split off and construct the tree with keys $> k$. Otherwise, we construct the tree with keys $\leq k$. This ensures that the tree we split off and construct will have height $\leq$ the height of the remnant tree. Without loss of generality, assume we are splitting off subtrees with keys $\leq k$. We return to x, and again follow parents up the tree, deleting the subtrees to the left of our path (they contain keys $\leq k$), using part (f). As we delete subtrees, we insert them into a new tree, by choosing the correct parent nodes (by lining the keys up at the same depth), which is easy since we are inserting subtrees covering descending, disjoint ranges. We use part (e) to do the insertions. Assume we are splitting into two subtrees of size a, b. We had to do $O(\log \min(a, b))$ insertions/deletions while constructing the new tree. Splitting causes the second term of the potential to decrease by $\beta \lg \frac{ab}{a+b}$. Assume by symmetry that $a \leq b$. Using the amortization arguments from parts (e) and (f), the amortized cost of the split is $O(\log a) - \beta \cdot (\log a + \log b - \log(a+b))$. We set β so that this is $O(\log(1 + \frac{a}{b})) = O(1)$.