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) $\text{predecessor}(k)$ 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) $\text{successor}(k)$ in $O(\log n)$ time: symmetrically

(c) $\text{predecessor-of-here}(k, x)$ 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) $\text{successor-of-here}(k, x)$ in $O(1)$ time: symmetrically

(e) $\text{insert-after}(k, x)$ 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) $\text{delete-here}(k, x)$ in $O(1)$ amortized time: given a pointer to the node $x$ representing a key $k$, delete $k$ from the data structure.

(g) $\text{split-here}(k, x)$ 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.)

$\text{Hint:}$ Start from $(a, b)$-trees.

$\text{Hint:}$ In defining a potential function to amortize split-here, think about what changes about the split edges.
**Solution:** 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, \(\Phi\), that we will use to analyze the time complexity of the operations:

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

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 \(\Phi\) 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 \(c\).