6.851 Advanced Data Structures (Spring’14)
Prof. Erik Demaine TAs: Timothy Kaler, Aaron Sidford

Problem 1 Sample solution

This solution is modeled off Section 5.2 in [1].

Before describing how to implement the retroactive data structure, we will briefly sketch an implementation of a non-retroactive dynamic read-only array with the required operations. This data structure can be thought of as a deque with an extra operation \( \text{get}(i) \) to obtain its \( i \)th element.

Suppose that you know the maximum size of the read-only array is \( m \). Maintain an array \( A \) of size \( 2m+1 \) indexed from \(-m, -m+1, \ldots, m\) with the 0th element omitted. In addition, maintain two counters \( L \) and \( R \) that maintain the net number of elements added to the left and right end of the array. The operation \( \text{addR}(x) \) sets \( A[R+1] = x \) and increments \( R \), and \( \text{remR}(x) \) sets \( A[R] = 0 \) and decrements \( R \). \( \text{addL} \) and \( \text{remL} \) are implemented analogously, but write to the left half of \( A \), e.g. \( \text{addL} \) writes to \( A[-L-1] \). The query operations can be implemented using \( A, R \), and \( L \): the operation \( \text{size()} \) returns \( R + L \) and \( \text{get}(i) \) returns \( A[i-L] \).

To implement a fully retroactive version of this data structure, rather than maintaining this array explicitly we maintain a balanced binary search tree where the leaves are the update operations \( \text{addR}, \text{addL}, \text{remR}, \text{remL} \) ordered by time. For each leaf node we associate numbers \( U_R \) and \( L_R \) such that \( U_R = 1 \) for \( \text{addR} \), \( U_R = -1 \) for \( \text{remR} \), \( U_L = 1 \) for \( \text{addL} \), and \( U_L = -1 \) for \( \text{remL} \). The numbers \( U_R \) and \( L_R \) are zero for all other nodes.

The value of \( L \) at time \( t \) is the sum of all \( U_L \) values for update operations that occurred at time \( t < t \) and \( R \) is the sum of all \( U_R \) values for update operations that occurred at time \( t < t \). The value of \( A[j] \) at time \( t \) is simply the result one of the two last update operations when \( -L - 1 = R + 1 \) was \( j \).

Therefore, it suffices to be able to compute the value of \( L \) and \( R \) at every time \( t \) and to find the last update operation when \( L \) or \( R \) was some specified value. For this purpose, we augment each node in the balanced binary search tree with six values, the sum of the \( U_L \) and \( U_R \) values in its subtree and the nodes with the smallest and largest \( L \) and \( R \) values in its subtree. We maintain these values in \( O(\log m) \) time for each \textbf{Insert} or \textbf{Delete} by updating the modified nodes’ ancestors.

- **Insert** \((t, \text{update}(x))\) where update \( \in \{\text{addL}(x), \text{addR}(x), \text{remL}, \text{remR}\} \): Insert a new leaf node into the binary search tree with the appropriate \( U_L \) and \( U_R \) values, updating the auxiliary information as needed.
- **Delete** \((t, \text{update}(x))\) where update \( \in \{\text{addL}(x), \text{addR}(x), \text{remL}, \text{remR}\} \): Delete the corresponding leaf node from the binary search tree, updating the auxiliary information as needed.
- **Query** \((t, \text{size})\): Find the leaf node corresponding to the last update performed before time \( t \). Compute \( L \) and \( R \) for \( t \) by adding the subtree sums of \( U_L \) and \( U_R \) of the left children of the ancestors of this node. Return \( R + L \).
- **Query** \((t, \text{get}(i))\): Compute the values of \( L \) and \( R \) as in the previous bullet and check that \( i < R + L \). Next we compute the value of \( A[j] \) for \( j = i - L \). To do this we find the leaf node corresponding to the last update operation that occurred before time \( t \). We then perform two walks up and down the tree to find the last update operations that occurred before time \( t \) when \( -L - 1 = j \) or \( R + 1 = j \). At least one of these operations must be a \( \text{addL} \) or a \( \text{addR} \) and we return the one that was performed last.

Using a balanced binary search tree all these operations can be performed in \( O(\log m) \) time.

References


\(^1\)Note that few small tweaks are required to adjust indices by 1 to account for the unused 0th array index.