# 6.851 Advanced Data Structures (Spring'14)

Prof. Erik Demaine TAs: Timothy Kaler, Aaron Sidford

## Problem 4 *Sample solution*

As pointed out by one of the students, the key insight to solving this problem is that the `next`() and `prev`() operations should be destructive, i.e. they need to modify the data structure. Without this it is difficult to ensure that `insert`($x$) and `delete`() do not cause memory to be spread out in such a way that a significantly long set of `next`() and `prev`() operations do not cause many cache misses. Here we present two solutions which use this insight. The first is our intended solution and then a clever and perhaps simpler solution noted by some of the students which support all operations in $O(1/B)$ memory transfers amortized.

## Intended Solution:

The main idea of the solution is to maintain the data as a standard linked list and whenever the `next`() or `prev`() operations occur attempt to rewrite the data to array and use the fact that copying between arrays can be done in $O(\lceil \frac{1}{B} \rceil)$ time amortized. We maintain our data as a linked list but attempt to replace standard linked list nodes with "super-nodes" that contain many of the elements in the array.

At all times we maintain the number of elements in data structure with the number, denoted $N$, and the position of the current element in the data structure, denoted $j$. We maintain our data in a linked list where each node in the linked list contains an array storing containing a consecutive subset of the elements of the data structure stored in order. We also maintain an array $A$ of size $\theta(N)$. During a sequence of `next`() and `prev`() operations we traverse the linked list rewriting the data to $A$ in order. When data needed for a `next`() or `prev`() operations has already been written to $A$ we read it from $A$ rather than the linked list. When a `insert`($x$) or `delete`() operation occurs, we perform the appropriate operation on the current array, create a new node, and copy the portion of $A$ corresponding data that was read during this sequence of `next`() and `prev`() operations. We insert this new node into the linked list deleting nodes it replaces and updating pointers of the neighbors. If $A$ contains at least half of the elements of one of the neighbors, we rewrite that nodes data into this new node as well.

Clearly the `insert`($x$) and `delete` operations take $O(1 + K/B)$ time where $K$ is the number of `next`() and `prev`() operations that occurred in sequence. Furthermore, so long as `next`() and `prev`() read from the same node or from $A$ the cost of these operations are $O(\frac{1}{B})$. The only time `next`() and `prev`() may be more expensive is when we move between nodes. However, we can attribute each of these transfers to one of the `insert`($x$) or `delete`() operations and we only do this once for each operation. Consequently the amortized cost of the operations is as desired. Finally, since nodes overlap in at most half the elements we see that the total space used is $O(N)$.

## Improved Solution:

We store the data in two stacks, $S_1$ and $S_2$. Stack $S_1$ contains all the elements in the data structure before or at the current finger and stack $S_2$ contains all the elements in the data structure after the current finger. Stacks $S_1$ and $S_2$ are stored as arrays of size $K$ where $K \in [N/2, 2N]$ and $N$ is the number of elements in the data structure. If $K \notin [N/2, 2N]$ we simply copy all the data to new stacks of the appropriate size which takes $O(N/B)$ memory transfers amortized by the $O(N)$ `insert`($x$) and `delete`() operations that made the resizing necessary.

To implement `next`() we pop from $S_2$ and push this element to $S_1$ and to implement `prev`() we pop from $S_1$ and push this element to $S_2$. To implement `insert`($x$) we push $x$ to $S_1$ and to implement `delete` we just pop from $S_1$. Since we are always reading and writing from the top block in $S_1$ and $S_2$ we see that for a cache miss to occur there must have been $B$ operations. Consequently the amortized cost of all operations is $O(1/B)$.[1]

[1] Some students stored the arrays for the two stacks overlayed. This is cool, but unnecessary. We just need the blocks for the top of the two stacks as well as the block to store $N$ to fit in memory. However we may assume $M = O(B)$.