The fact that the \texttt{insert} and \texttt{removeMin} methods in a sufficiently bushy binary search tree take $O(\log N)$ time suggests a way of using binary search trees for sorting arrays, exactly analogous to the algorithm we saw earlier in the semester that used the similar methods of priority queues. Specifically, we can create an empty binary search tree, insert the elements of the array to be sorted into that binary search tree, one by one, and then repeatedly call \texttt{removeMin} to extract them in sorted order, overwriting the elements of the unsorted array from left to right.

If $N$ is the size of the array, there will be $N$ calls to the \texttt{insert} method, each of which (we hope) will take $O(\log N)$ time, followed by $N$ calls to the \texttt{removeMin} method, each of which (we hope) will also take $O(\log N)$ time. The overall running time, therefore, will be $O(N \log N)$.

This won’t work in the worst case, where the binary search tree constructed in the insertion phase consists of one long stringy “branch” that actually has no branching nodes in it. In that case, each of the insertions will be $O(N)$ rather than $O(\log N)$. In the absolutely worst case, which comes up when the elements of the array are reverse-sorted to begin with, the nodes along the one stringy branch all have empty right subtrees, so that each \texttt{removeMin} operation also takes $O(N)$ time.

One way to mitigate the effects of the worst cases is to permute the array elements randomly before transferring its contents to the binary search tree. This has no effect on the final result, since the elements of the original array are going to be overwritten anyway, and they are the same elements before and after the permutation. Shuffling takes $O(N)$ time, and the analysis in section 19.3 of the textbook shows that the running time for $N$ insertions averaged over all $N!$ of the possible permutations of the inserted elements is $O(N \log N)$.

A better approach is to strengthen the invariants that binary search trees must observe so that they also constrain the shape of the acceptable trees, forcing them to be kept somewhat bushy at all times. This will ensure that the running time of treesort is $O(N \log N)$ even in the worst case, regardless of the order in which the elements of the array are transferred into the binary search tree. We’ll study this possibility in future classes that deal with tree balancing.

If we want to sort the array into descending order rather than ascending order, we can simply adjust the overwriting phase of the algorithm so that the elements of the array are overwritten from right to left rather than from left to right.

Copyright © 2019 Shervin Hajiamini

I am indebted to my colleague, John David Stone, for assistance with this handout.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/deed.en_US

or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.