Worksheet 32: Tree Sort
Group 11

In Preparation: Read chapter 10 on the Tree data type.

Both the Skip List and the AVL tree provide a very fast O(log n) execution time for all three of the fundamental Bag operations: addition, contains, and remove. They also maintain values in sorted order. If we add an iterator then we can loop over all the values, in order, in O(n) steps. This makes them a good general purpose container.

Here is an application you might not have immediately thought about: sorting. **Recall some of the sorting algorithms we have seen. Insertion sort and selection sort are O(n2), although insertion sort can be linear if the input is already nearly sorted. Merge sort and quick sort are faster at O(n log n), although quick sort can be O(n2) in its worst case if the input distribution is particularly bad.**

Consider the following sorting algorithm:

How to sort a array A
Step 1: copy each element from A into an AVL tree
Step 2: copy each element from the AVL Tree back into the array

Assuming the array has n elements, what is the algorithmic execution time for step 1?
O(n log n)
What is the algorithmic execution time for step 2?
O(n)
Recall that **insertion sort and quick sort** are examples of algorithms that can have very different execution times depending upon the distribution of input values. Is the execution time for this algorithm dependent in any way on the input distribution?

Yes it does. Adding one item to a binary search tree is on average an $O(\log n)$ process (in big O notation), so adding n items is an $O(n \log n)$ process, making tree sort a so-called fast sort. But adding an item to an unbalanced binary tree needs $O(n)$ time in the worst-case, when the tree resembles a linked list (degenerate tree), causing a worst case of $O(n^2)$ for this sorting algorithm. This worst case occurs when the algorithm operates on an already sorted set, or one that is nearly sorted. Expected $O(\log n)$ time can however be achieved in this case by shuffling the array.

The worst-case behavior can be improved upon by using a self-balancing binary search tree. Using such a tree, the algorithm has an $O(n \log n)$ worst-case performance, thus being degree-optimal for a comparison sort.

http://en.wikipedia.org/wiki/Tree_sort

A sorted dynamic array bag also maintained elements in order. Why would this algorithm not work with that data structure? What would be the resulting algorithmic execution time if you tried to do this?

As already stated in the answer to the previous question, when dynamic array maintains elements in order, this is equivalent to adding an item to an unbalanced binary tree, which is equilavent to complexity of $O(n)$ time in the worst-case.

Build the binary search tree: O(n^2)

Traverse the binary tree: O(n)
Total: O(n^2) + O(n) = O(n^2)

Reference: http://www.cs.cmu.edu/~mrmiller/15-121/Lectures/39-bucketsort-pq.pdf

Can you think of any disadvantages of this algorithm?
It might be more costly in terms of time and space.

Algorithm shown above is known as tree sort (because it is traditionally performed with AVL trees). Complete the following implementation of this algorithm. Note that you have two ways to form a loop. You can use an indexing loop, or an iterator loop.
Which is easier for step 1? Which is easier for step 2?
Step 1 - index, step 2 - iterator.

void treeSort (TYPE *data, int n) { /* sort values in array data */
AVLtree tree;
int i=0, count=0;
AVLtreeInit (&tree);
for(i=0; i < n ; i++)
AVLAddTree(&tree, data[i]);
/* Assuming no iterator */
_treeSortHelper(tree->root, data, &count);
}
_treeSortHelper(AVLNode *cur, TYPE *data, int *count)
{
if(cur != 0)
{
_treeSortHelper(cur->left, data, count);
data[*count] = cur->val;
***count = *count + 1;**
_treeSortHelper(cur->right, data, count);
}

http://classes.engr.oregonstate.edu/eecs/fall2013/cs261-001/Week_10/Worksheet32.ans.recursive.pdf