Jonathan Lam
Prof. Sable
ECE 264
DSA 1
12 / 19 / 18

# Data Structures and Algorithms I
# Fall 2018
# Homework #3

## 1. Sorting

a) Insertion sort adds an element onto an already-sorted array— if it is the largest element of the sorted array, then no swaps are made; otherwise it is swapped into its correct position in the sorted array. If an array is near sorted, few swaps will need to be made, and most of the swaps that will need to be made will be placed near the end of the already-sorted array. (For an already-sorted array, this will be linear, because N comparisons and 0 swaps will be made.)

b) Bubble sort only swaps adjacent elements that are in the wrong order, and then checks if any swaps were made in the last pass. If most of the elements are in the right order, few swaps and few passes will need to occur; if it is already sorted, this is linear because no swaps are made.

c) Selection sort does many comparisons (O(N^2)) but few swaps (O(N)), whereas insertion sort usually does fewer comparisons (O(N^2)) but more swaps (O(N^2)). Movement in memory is generally much slower than comparing objects, especially for larger data, so the reduction in memory swaps with selection sort outweighs the fact that it is generally slower than insertion sort.

d) For sorted or nearly-sorted arrays, this creates O(N^2) behavior, the worst-case behavior for quicksort. This is because the partition operation performs N operations but almost no swaps, which is followed by a recursive quicksort with size N-1, which is similarly unoptimized.

e) Mergesort is typically not linear (its running time is O(NlogN)), but it is possible to implement it to look for natural “runs” of numbers to ignore while sorting. This would take linear time (one pass) for already sorted arrays. (Timsort includes a merge sort with natural runs for better performance on nearly-sorted arrays).

f) It is possible using a radix sort — the integers would then be sorted digit by digit into bins, and none of the integers would be compared against each other.

g) An indirect sort is useful for reducing the number of swaps of the actual data, which is useful when swapping elements in memory is slow, i.e., when the data is large or there is no constant time access to elements (which is the case in a linked list).

h) A typical implementation of radix sort is stable, because the order that elements were placed into bins is the same order that they are taken out. (Identical elements must be placed in the same bin, and the order they are taken out is the same as the order they were relative to each other in the original sequence.)

i) Linear does not mean always faster than linearithmic (NlogN)— in this case, the high overhead of the median-of-median-of-five pivot selection makes it generally slower than the median-of-three pivot selection for most sequences.

j) It is not efficient— essentially it would be recursively performing a quicksort on the smaller halves until a singleton sequence is obtained (the smallest number). A one-pass search for the smallest element (comparing every element against the minimum value so far in the sequence) would be much faster.

---

**2. Trees**

a) Preorder traversal, inorder traversal, and postorder traversal are generally implemented recursively. (Preorder recursively traverses children after processing current node, postorder recursively traverses children before processing current node, and inorder traverses left subtree, current node, and then right subtree).

b) A pre-order traversal is used to evaluate an expression tree: the two subtrees of an operator have to be recursively evaluated to numbers first before the operator itself can get evaluated (evaluate children before parent).

c) The storing of pointers to left children and right sibling is useful when the number of children of a node is unknown, in order to reduce space but still fully describe the tree's structure. However, to store pointers to the two children of a binary search tree is preferred because it still describes the whole tree, doesn't increase space (still only two pointers to nodes, because nodes can only have two children) and it will be advantageously performant to access the right child of a node directly (as opposed to accessing it as the right sibling of the left child).

d) Yes. This would be a sequence that oscillates at each depth with a lessening amplitude. For example, the sequence 1, 100, 2, 99, 3, 98, … , 49, 50 would be such a sequence that would create a tree of depth 99.

e) If a grandchild exists in an AVL tree, its height must be equal to the height of the tree minus two (grandchild) plus or minus one (AVL tree condition). Thus, its minimum height is h-3.

f) Insertion operations in an AVL tree may be slightly slower because of the rotations and keeping track of height of a node. If there are few elements to enter to a search tree, and if the elements are randomly ordered (i.e., not likely to yield worst-case linear behavior), then it might be more efficient to use a simple binary search tree.

g) An inorder traversal can be used to iterate through an AVL tree in order based on its keys.

h) Only one rotation is necessary to restore balance (this is true for any insertion into an AVL tree).

Jonathan Lam
Prof. Sable
ECE 264
DSA 1
12 / 19 / 18

i) Because each node of a balanced binary search tree can only have two children, it might take many node accesses to reach the desired leaf node. For very large amounts of data in a tree, the data is usually stored on a hard drive, and every access is a (slow) memory access. In a B+ tree, having more children nodes per node allows for memory accesses to be much more efficient (usually optimized to memory block sizes).

j) The minimum depth would be three, because there would have to be at least enough leaf nodes to store all of the data values (100^3 leaves on the third depth, assuming that the tree is perfectly filled with 100 children per non-leaf node, is one million).