Problem Set 4 Solutions

(Exercises were not to be turned in, but we're providing the solutions for your own interest.)

Exercise 4-1. Consider $n = 3$, where the elements are $1, 2, 3$. There are 6 ways to insert the elements, but only 5 different trees. The tree with 2 at the root is created by two different insertion orders: $2, 1, 3$ and $2, 3, 1$.

Exercise 4-2. The binary search tree property says that all elements to the *left* are smaller, and all to the *right* are larger. The min-heap property says that all elements both to the left and right are smaller. The min-heap property cannot be used to print out the keys of an n-element heap in sorted order in $O(n)$ time, because otherwise we could comparison sort in linear time: just build a heap of the elements ($O(n)$ time) and output them in sorted order (by assumption). This contradicts the comparison-model sorting lower bound of $\Omega(n \log n)$.

Exercise 4-3. Consider the binary search tree as a graph with edges between each parent and its children. Every node (except the root) has a unique parent, so there are $n - 1$ edges. Then the algorithm, as described, traverses each edge exactly once in each direction (once going "down," the other going "up"), for a total of twice per edge. This takes $O(n)$ time, as desired.

Exercise 4-4. No, deletion is not commutative. To see why, consider the tree with root 5, children 3 and 7, and 6 as a child of 7. We will delete 5 and 3: if 5 is deleted first, then we get the tree with root 6, children 3 and 7; after deleting 3 we get the tree with root 6 and right child 7. If 3 is deleted first, 5 remains the root with only a right child; after deleting 5 we get the tree with root 7 and left child 6.

Exercise 4-5. The CPU time with linear search is $O(th) = O(t \log_t n)$. With binary search, the t becomes $\lg t$ for a total CPU time of $O(\lg t \log_t n)$. By change of base, we know that $\log_t n = \frac{\lg n}{\lg t}$, so $\lg t \log_t n = \lg n$ as desired.

Exercise 4-6. Consider the shortest simple path from x to a descendant leaf, which has length ℓ; then it contains at most ℓ black nodes. Any other simple path from x to a descendant leaf has the same number of black nodes, and cannot have two red nodes in a row. Therefore it has length at most 2ℓ, as desired.

Exercise 4-7. Every rotation (left or right) is determined by two vertices which have a parent-child relationship, and vice-versa. Every node (except the root) has a unique parent, so the number of parent-child pairs is exactly $n - 1$, yielding $n - 1$ possible rotations.

Problem 4-1. Binary tries

(a) Given a string $a = a_0 \ldots a_p$ to insert into the trie, we do the following: follow the search procedure, creating a new node (with counter zero) whenever the appropriate child does not yet exist. At the final node, increment its counter. The running time is $\Theta(p)$ (where p is the length of the string), and does not depend at all upon the number of strings stored in the trie.

(b) First, insert all of the strings of S into a trie. Next, do a *preorder* tree walk of the trie (where we first visit the root, then recursively visit the left subtree, then the right subtree), keeping track of the string corresponding to the current node. More specifically, every time we descend from a parent to a left (or right) child, we append a 0 (or 1) to the current string; every time we ascend to a parent, we strip off the last character of the current string. When visiting each node, we output c copies of the current string, where c is the count of the node.

The correctness of the algorithm comes from the preorder walk. It is easy to see that a node's string lexically precedes all of the strings of its left subtree, and all of those strings precede all of the strings in the right subtree.

For the running time, note that inserting all of the strings into a trie takes $\Theta(n)$ time, by the previous part. Printing out all of the strings also takes $\Theta(n)$ time, so we need only analyze the running time of the preorder walk. Note that there are at most n different strings (because none of the strings have length 0, by the problem description), so there are at most n unique nodes in the trie. Therefore a tree walk takes $O(n)$ time because each edge in the trie is traversed exactly twice (once down, once up), and there are at most $n-1$ edges. So we get that the running time of the sorting algorithm is $\Theta(n)$.

(c) This sorting algorithm would take $\Omega(n^2)$ time, when given inputs of the following form: $n/2$ copies of the string 0, and one string of length $n/2$ that consists of all 0s. (The zeros are not important; only the lengths are.) After padding, we are left with about $n/2$ strings each of length $n/2$. RADIX-SORT takes $\Theta(d(m+k))$ time, where $d = n/2$, $m = n/2$, and $k = 2$. This simplifies to $\Theta(n^2/4) = \Omega(n^2)$.

Problem 4-2. Treaps

(a) We prove this fact by (strong) induction. Clearly when $n = 1$, there is only one treap on n elements when the key and score are fixed. Assume now that for all $n < k$, there is a unique treap. Suppose now that we have k elements with fixed (distinct) scores. Then there is some element r which has the largest score, which must be the root of any treap on these elements (by the score-heap property). Now exactly those elements which are less than r must go into the left subtree, and there are at most $k-1$ of them. By the inductive hypothesis, there is exactly one treap of those elements forming the left subtree. Likewise, there is exactly one treap of the elements greater than r for the right subtree. Therefore there is only one possible treap on all k elements, and the result follows by induction.

(b) We have seen that once the scores are fixed, there is exactly one treap that results, regardless of insertion order. Notice also that if we set all the scores ahead of time, inserting the elements in *decreasing* order of scores into a *binary search tree* will yield the treap (because new elements are always inserted below existing ones). Because the scores are random, every permutation of the elements (when put in decreasing order of score) is equally likely, so the treap with random scores is identical to a binary search tree with the elements inserted in random order. We have seen in class that a binary search tree has expected height $O(\log n)$ when the elements are inserted in random order; therefore a treap has expected height $O(\log n)$ when the scores are assigned randomly.

(c) The main idea is the following: first we insert the element with BST-INSERT (which ensures that the data structure has the BST property), which may violate the heap property. Then we use rotations to "fix-up" these violations, while retaining the BST property.

More specifically, the algorithm is the following: to insert an element x into the trie, first insert it using BST-INSERT. Then if x's score is larger than its parent's score, do a rotation of the proper type around x's parent (if x is a right child, do a left rotation; if x is a left child, do a right rotation). Repeat until x's score is less than its parent's, or until x is at the root of the trie.

Now we argue correctness: after insertion, the trie is still a binary search tree on the elements, and rotations preserve this property. Furthermore, after insertion the only violation of the heap property (that is: "each node's score is larger than both of its children's scores") can be between x and its parent. It is easy to verify that performing a rotation puts x above its old parent, and does not introduce any new violations of the heap property, except possibly between x and its new parent. Therefore our algorithm preserves both the BST and heap properties, which means the data structure remains a treap.

For running time, note that the normal BST insertion requires $O(h)$ time, where h is the height of the tree (and is $O(\log n)$ in expectation). The number of rotations we do is also at most h, so the total running time is $O(\log n)$. (In fact, it can be proven that the expected number of rotations per insert is at most 2, though this doesn't improve the asymptotic running time.)

Problem 4-3. Modified B-trees

(a) To SEARCH for a key x in a modified B-tree, start at the root. Check each of the children in order, and recursively SEARCH for x in the last child whose minimum leaf is at most x (if the first child's minimum is greater than x, then x is not in the tree). At the leaf level, return the leaf only if its key is x.

We now argue correctness: the keys found in a particular subtree are all between that subtree's minimum and the next subtree's minimum (inclusive). Therefore if x is in the tree we are searching, it is in the last subtree whose minimum is still at most x.

For running time, notice that each child is examined at most once, for $O(t)$ time per level. There are h levels in the tree, for a total of $O(th)$ time.

(b) Changes to SPLIT-CHILD: first we note that keys do not ascend. Also note that internal nodes no longer maintain any keys, so this simplifies the code significantly. We simply convert a node having $2t$ children to two nodes, each with t. The node having the smaller children keeps the same minimum, while the other node gets the minimum of its first child. If a new parent (i.e., root) is created, its minimum is the minimum of the lesser child.

Changes to INSERT-NONFULL: when inserting a new key at a node, we compare it to the node's current minimum and update the minimum if the key is smaller. This is the only change.

In both cases, there is only a constant amount of new work (over the original versions) performed: a lookup of the node's minimum in the first case, and a comparison in the second. Therefore the asymptotic running times are the same.

(c) To find a small list of subtrees containing exactly $K \cap R$, search for k and k' "in parallel," that is, switch between the searches so that they descend levels in step with each other. When the searches diverge (that is, descend to different children of the same node), output all children that are strictly between the two children followed in the search. After that, whenever searching a node for k, output all of its children that strictly follow the descended child. In the search for k', output all children that strictly precede the descended child.

First we analyze the running time: the algorithm examines and outputs $O(t)$ nodes at each level of each search, for a total running time (and output length) of $O(th)$.

For correctness, note that the keys in the intersection $K \cap R$ appear in one continuous block across the leaf level, and they are those keys between k and k'. If we draw a path from k to the root and k' to the root, then we want all the subtrees which "hang off" the insides of these paths. It is easy to see that all leaves in these subtrees are between k and k', and that they cover the full range.

Problem 4-4. Rotations

(a) Suppose we are right-rotating around y, which has left child x and right-subtree γ, where x has right- and left-subtrees α and β, respectively. The right-rotation does not change $\ell(v)$ for any v that is outside the subtree orignally rooted at y, so we need only examine how L changes on that subtree. Let $C(T)$ be the number of elements in a tree T. Before the rotation, $L = (1 + C(\alpha) + C(\beta)) + C(\alpha) + L(\alpha) + L(\beta) + L(\gamma)$ by simply adding $\ell(y)$, $\ell(x)$, and L of each subtree. After the rotation $L = C(\alpha) + C(\beta) + L(\alpha) + L(\beta) + L(\gamma)$, so it decreases by $1 + C(\alpha) \geq 1$ as desired.

In any n-node tree, $\ell(v) < n$ for every node v, so $L(T) < n^2$ by summing over all v. Because $L(T) \geq 0$ for any tree T, and each right rotation decreases L by at least 1, the number of consecutive right-rotations can be no more than $n^2 = O(n^2)$.

(b) Note that for a left-path tree T, $L(T) = 1 + 2 + \cdots + (n-1) = \Omega(n^2)$, and for a right-path tree U, $L(U) = 0$. Also note that in the notation from the previous part, a right-rotation decreases L by $1 + C(\alpha)$. Therefore it is enough to perform a series of right-rotations, where each respective α is an empty tree; this series would include $\Omega(n^2)$ right-rotations. This is indeed possible: for any tree, consider the set of all nodes that have non-empty left subtrees. If the tree is not a right-path, then this set is non-empty. Therefore it has some node y with largest depth, which has left child x. Then x has no left child (otherwise x would be in the set and have larger depth than y), so a right-rotation around y decreases L by only 1, as desired.

(c) We can turn any tree into a right-path with at most $n - 1$ rotations, by increasing the number of nodes on the "right spine" (those elements reachable from the root by following only right children) with each rotation. Note that initially the right spine contains at least one element (the root itself). In addition, if the tree is not a right-path, then there is some element x which is the left child of an element y on the right spine. We can then right-rotate around y, which adds x to the right spine without removing any other nodes from it. Hence, the number of elements on the right spine increases by one. Therefore after at most $n-1$ rotations of this type, the right-spine has n nodes and is, in fact, a right-path.

To turn any tree A into any other tree B, first turn A into a right-path using at most $n-1$ right-rotations. Then consider how B would be turned into a right-path using at most $n - 1$ right-rotations; from the right-path (obtained from A), do the corresponding *left*-rotations *in reverse order* to arrive at B. This process requires at most $2n - 2$ rotations total, as desired.