# Problem Set 4

This problem set is due **in lecture** on **Monday, October 21.**

*Reading:* Chapter 12; §18.1-18.2, Chapter 13

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered by the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

**Each problem should be done on a separate sheet (or sheets) of three-hole punched paper.**

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions.

---

**Exercise 4-1.** Do exercise 12.4-3 on page 268 of CLRS.

**Exercise 4-2.** Do exercise 12.1-2 on page 256 of CLRS.

**Exercise 4-3.** Do exercise 12.2-7 on page 260 of CLRS.

**Exercise 4-4.** Do exercise 12.3-5 on page 264 of CLRS.

**Exercise 4-5.** Do exercise 18.2-6 on page 449 of CLRS.

**Exercise 4-6.** Do exercise 13.1-5 on page 277 of CLRS.

**Exercise 4-7.** Do exercise 13.2-2 on page 278 of CLRS.

---

**Problem 4-1.** Given two strings $a = a_0a_1 \ldots a_p$ and $b = b_0b_1 \ldots b_q$, where each $a_i$ and $b_i$ is in some ordered set of characters, we say that string $a$ is ***lexically less than*** string $b$ if either:

1. there exists an integer $j$, $0 \leq j \leq \min(p, q)$, such that $a_i = b_i$ for all $i < j$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i \leq p$.

For example, if $a$ and $b$ are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This is similar to the ordering used in English-language dictionaries.

A ***binary trie*** (pronounced "try") is a data structure that allows one to store bit strings. It is a binary tree in which the root-to-node paths correspond to the bits of binary strings. Each node simply has a count indicating how many times the corresponding bit string has been inserted into the structure. When searching for a key $a = a_0a_1 \ldots a_p$, we start at the root. When at a node of depth $i$, we go to the left child if $a_i = 0$ and to the right child if $a_i = 1$, terminating at a node of depth $p + 1$. If the final node's count is greater than zero, the search succeeds; otherwise, $a$ is not in the tree.

**(a)** Give an algorithm implementing INSERT for a binary trie. How does its running time depend on the length of the inserted string? On the number of strings stored in the trie?

**(b)** Let $S$ be a set of binary strings, not necessarily distinct, nor of the same length. Suppose the longest string in $S$ has length $\ell$, and the lengths of all the strings sum to $n$. Give an algorithm to lexically sort $S$ using a binary trie in $\Theta(n)$ time.

**(c)** Let $S$ be as above. Another natural way to sort the strings in $S$ is to use the following variant of RADIX-SORT. Extend each string to length $\ell$ by padding it with a special character ␣ that is less than both 0 and 1; then sort the strings by each bit position, from rightmost to leftmost. For example, if the strings are `10`, `000`, `1011`, `00`, then they would be extended to `10␣␣`, `000␣`, `1011`, `00␣␣` and placed in sorted order `00␣␣`, `000␣`, `10␣␣`, `1011`.

What is the worst-case asymptotic running time of this sorting routine, in terms of $n$? Describe inputs that yield this worst-case behavior.

**Problem 4-2.** Suppose that we would like to keep a set of keys in a binary search tree. New keys will be added at various times, and searches may occur at any moment. Of course we'd like the tree to be balanced at all times, so that we can search and insert in $O(\log n)$ time. One way to try to keep the tree balanced is the following randomized scheme: upon insertion, assign each key a random real-valued ***score*** from $[0, 1]$. Keep the keys in a structure that is a binary search tree on the keys and a heap on the scores. (I.e., if you only look at the keys of the resulting data structure, what you see is a binary search tree. If you only look at the scores, what you see is a heap.) This structure is called a ***treap***, because it is both a tree and a heap. Figure 1 shows a small example with keys $a, b, c, d$ and associated scores $.4, .8, .1, .7$:

**(a)** Prove that when the keys are distinct and each key has a fixed score, there is a *unique* structure that is both a binary search tree on the keys and a heap on the scores. In other

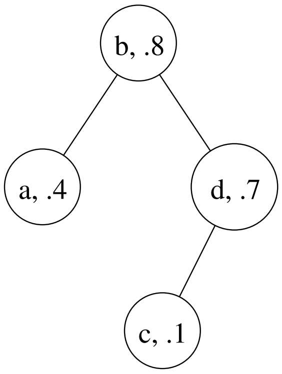


**Figure 1**: A small treap.

words, prove that there is only one possible treap for the given assignment of scores to keys. (You may also assume that the scores are distinct, because with probability 1 they are.)

**(b)** Argue that regardless of the insertion order of the keys, a treap has expected height $O(\log n)$. *Hint:* Relate a treap to a randomly built binary search tree.

**(c)** Give an algorithm for the TREAP-INSERT operation. Assume that a random score has already been attached to the element to be inserted.

**Problem 4-3.** Consider a variant of B-trees in which the keys and data are only stored at the *leaf level* (not in any of the internal nodes). Thus, each key in the dynamic set is stored as the key of a unique leaf. (In contrast, in normal B-trees, each key in the dynamic set is stored as the key of a unique node, which may be an internal node.) Each internal node $v$ now only stores the *minimum key* that can be found in the subtree rooted at $v$, in addition to the pointers to its children. In this structure, the keys appear in sorted order across the leaf level.

**(a)** Explain how SEARCH can still be performed efficiently (i.e., in time proportional to the branching factor $t$ and the height $h$ of the tree) using this structure.

**(b)** What changes must be made to SPLIT-CHILD and INSERT-NONFULL in order to maintain this structure correctly? Argue that their running times remain asymptotically the same. *Remember*: each internal node must maintain the minimum key in its subtree.

**(c)** A common operation in databases is a ***range query*** which asks for all keys occurring in a given ***range*** $R = [k \mathinner{..} k']$. In the variation on B-trees described above, there is a particularly efficient way to represent the answer to a range query. Let $K$ be the set of keys stored in the tree, so that the intersection $K \cap R$ is the answer to the range query. We want to find a short list of nodes $v_1, \ldots, v_m$ in the tree such that the *union* of all keys in the subtrees rooted at $v_1, \ldots, v_m$ is exactly $K \cap R$, and no key appears in more than one of the subtrees. Note that both $K \cap R$ may have up to $n$ elements, but we want to represent it using only logarithmically many subtrees.

Describe how to find such nodes $v_1, \ldots, v_m$ in $O(th)$ time, given $R$. In particular, this running time implies that the number $m$ of these nodes must be at most $O(th)$.

**Problem 4-4.** In this problem we'll explore the rotation operations on binary search trees. As discussed in class, these operations change the structure of a binary search tree without affecting the order of the underlying nodes. See CLRS §13.2 (pp. 277-9) for a description of the rotation operations.

**(a)** Let $\ell(v)$ denote the number of nodes in $v$'s left subtree. Let $L(T)$ be the sum of $\ell(v)$ over all nodes of the tree $T$. Show that a right rotation decreases $L(T)$. Deduce that it is impossible to do more than $O(n^2)$ consecutive right-rotations in an $n$-node tree.

**(b)** Show that on an $n$-node ***left-path*** (a tree where all children are left children) it is possible to do $\Omega(n^2)$ consecutive right rotations (without ever doing any left rotations.)

**(c)** Show that $n - 1$ right rotations are enough to turn any $n$-node tree into a right-path. (The first two parts of this question show that if you do the right-rotations carelessly it is possible to do $\Theta(n^2)$; the point here is to show that if you are clever, you only need $n - 1$.) Deduce that $2n - 2$ rotations are sufficient to turn any binary search tree on $n$ elements into any other binary search tree on the same elements.