Problem Set 4.5

This problem set is for practice purposes and will not be collected.

**Reading:** Chapters 13, 15, 18, 19 of CLR.

**Problem 4.5-1. Assessing the damage**

It’s hurricane season, and the local news station is trying to keep track of the damage on the beach for local property owners. They start with a list of local houses, each with a distance from the shore. They commission you to design a data structure to support the following operations:

- **DECREASE-VALUE(distance, amount):** Decrease the value of every property within `distance` of shore by `amount`.
- **ASSESS-DAMAGE(distance):** Report the total damage to a property `distance` feet from shore.

The idea is that each time a bad wave hits, reaching some distance inland, the station will call `DECREASE-VALUE`, and the homeowners (who are somewhere safe, not at home) can call `ASSESS-DAMAGE` to see how their house is doing. Show how to implement both of these operations in $O(\log n)$ time, where $n$ is the number of houses.

**Problem 4.5-2. Random TREAPs**

Suppose that we have a set of keys that we want to store in a binary search tree. Of course we’d like the search tree to stay balanced, so that we can search in $O(\log n)$ time. One way to try to keep the tree balanced is the following randomized scheme: assign every key a random score, a number in $[0,1]$. Use a binary search tree on the keys that has a heap-order 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 TREP, because it is both a tree and a heap. In figure ?? there is a small example with keys $a, b, c, d$ and associated scores $0.4, 0.8, 0.1, 0.7$:

(a) Describe how to implement a TREAP-insert operation.

(b) Prove that for distinct keys there is a unique binary search tree on the keys that has a heap order on the scores. (Observe that you may also assume that the scores are distinct, because with probability 1 they are.)

(c) Argue that regardless of insertion order, a TREP has expected height $O(\log n)$. (Hint: relate a TREP to a randomly built binary search tree)
Problem 4.5-3.  **Stacking the Deque**

Consider a data structure that supports the following operations:

- **Push**($x$)—adds item $x$ to the front of the data structure.
- **Pop**()—removes the item at the front of the structure and returns it.
- **Inject**($x$)—adds item $x$ to the back of the data structure.
- **Eject**()—removes the item at the back of the structure and returns it.
- **Find-Min**()—returns the minimum item in the data structure.

If we restrict our operations to **Push** and **Pop** only, the data structure is called a *stack*. If we restrict our operations to **Inject** and **Pop** only, the data structure is called a *queue*. (For further discussion, see CLR pages 200–202.) Of course, by symmetry, restricting to **Inject** and **Eject** again results in a stack, and restricting to **Push** and **Eject** again results in a queue. We call a double-ended queue, or **deque** (pronounced “deck”) the hybrid data structure that supports *all four* of these operations. When a deque also supports the **Find-Min** operation, we call it a **min-deque**.

(a) Describe an efficient implementation of a min-stack—that is, a data structure that supports **Push**, **Pop**, and **Find-Min** in $O(1)$ worst-case time per operation.

(b) Describe an efficient implementation of a min-deque—that is, a data structure that supports all five operations in $O(1)$ *amortized* time per operation. (Hint: make a deque out of two stacks.)

Problem 4.5-4.  **Quick Array Inserts**

You have an application which wants to maintain a sorted array of numbers, but is receiving new ones frequently. You decide that instead of inserting each one as it arrives, you will keep a sorted array and an unsorted list of extra items, and every once in a while you will
combine the two to create a new sorted array and an empty list. Your data structure keeps
two extra values, the size \( n \) of the array and the length \( m \) of the list. Insert looks like this:

\[
\text{INSERT}(A, l, x) \\
l \leftarrow \text{CONS}(x, l) \\
m \leftarrow m + 1 \\
\textbf{if } m \geq \log n \\
B \leftarrow \text{SORT}(l) \\
A \leftarrow \text{MERGE}(A, B) \\
n \leftarrow n + m; m \leftarrow 0
\]

Here \text{SORT} is some \( O(n \log n) \) sort which takes a list and returns an array.

(a) What is the amortized running time of \text{INSERT}? (Hint: if \( A \) has size \( n \), what is the
total time required for the next \( \log n \) calls to \text{INSERT}?)

(b) Suppose you implement search by first doing a binary search on \( A \), and if that fails,
then doing a linear search on \( l \). What is the worst case running time of a search?

Problem 4.5-5. Deletion in B-trees

In this problem, we work with B-trees which are covered in chapter 19 of the textbook. The
definition of a B-tree is given on page 385 of the textbook. We demonstrate how the
\text{BTREE-DELETE} operation can be implemented. The \text{BTREE-DELETE} procedure is called
with a key \( k \) to be deleted and the root \( T \) of the B-tree.

In order to delete the key \( k \) from the B-tree \( T \), we first have to find \( k \) in \( T \). We would like
the key \( k \) to be in a leaf, since it makes the deletion procedure easier.

(a) If the key \( k \) is found in an internal node, describe how to modify the B-tree, such
that key \( k \) is transferred to a leaf.

We next describe a series of possible cases and investigate what to do in each case.

(b) Describe what can be done if \( k \) is in the root \( T \) of the B-tree.
(c) Describe what can be done if \( k \) is in a node \( x \) with at least \( t \) keys.
(d) Describe what can be done if \( k \) is in a node \( x \) that has exactly \( t - 1 \) keys, but either
\( x \)'s left sibling or \( x \)'s right sibling has at least \( t \) keys. (Hint: Rotate somehow.)
(e) Describe what can be done if \( k \) is in a node \( x \) that has exactly \( t - 1 \) keys, and none
of \( x \)'s siblings has more than \( t - 1 \) keys either. (Hint: Merge \( x \) with its sibling and
some key from the parent node. Then recursively delete the key from the parent.)