Algorithms and Data Structures
Spring 2018
Exercises for Units 12 and 13

1. In class we discussed dynamic arrays that allow constant time access of the i-th elements and whose length can be increased by one in constant amortized time, while using linear space. Develop a method that makes such arrays fully dynamic in that the length can also be decreased by one in constant amortized time (making the element in the formerly last position lost for good).

   So overall your method should allow constant worst case time access and constant amortized length increase and decrease by one, while the space that is used is always proportional to the current lengthe of the array.

   It may be easiest to use the bank account method for the amortized analysis, but you are free to use any correct approach for your analysis.

2. If you feel uneasy with splay trees, draw a few examples and test insertions and deletions.

   You can also search the web for splay tree animations and experiment with them.

3. Let n elements $e_1, e_2, \ldots, e_n$ be stored in the nodes of a splay tree. Consider a sequence S of m accesses where element e_i is accessed in total $m_i \geq 1$ times $(1 \leq i \leq n)$. The cost of accessing an element is proportional to 1 plus the depth of the element in the current tree. All accesses will be performed using the *splay method.*

   Prove that the total time for performing S is

$$O\left(m + \sum_{1 \leq i \leq n} \left(m_i \cdot \log\left(\frac{m}{m_i}\right)\right)\right).$$

4. If you feel uneasy with treaps, draw a few examples and test insertions and deletions.

   You can also search the web for treap/randomized search tree animations and experiment with them.

5. Let T be some binary search tree with root r. For node v in T define its *left ancestor* to be the first node on the path from v to r whose key is smaller than the key of v. Analogously the *right ancestor* is defined to be the first node on this path with larger key. Such a left ancestor or right ancestor need not exist. (Can you characterize those nodes?)

   Let us call an implementation of a binary search tree *lush* if it stores with every node four pointers: left child, right child, left ancestor, and right ancestor.

   (a) Can rotations be realized in constant time in a lush implementation?

   (b) Consider treaps with lush implementations. Give implementations of SPLIT and CONCATENATE that do not rely on rotations.

6. A *max-insertion* into a treap T inserts a new key that is larger than all keys currently in T.

   (a) Give a simple implementation of max-insertions that relies on maintaining a pointer ("finger") to the maximum key node and parent pointers.

   (b) Can you express the cost of the max-insertion in terms of the structure of the resulting treap?

   (c) Show that the expected cost of a max-insertion into a randomized search tree is constant.

   (d) Show that the *amortized* cost of a max-insertion into a general treap is constant (no matter which priorities are used).

   In other words, assume a sequence of n items is inserted into an intially empty treap in increasing key order using max-insertions. Show that the total cost of these n insertions is $O(n)$.

   (e) By symmetry whatever you proved for max-insertions also holds for the corresponding min-insertions.

   Is the statement of (d) still true for a sequence of arbitrarily interlaced max-insertions and min-insertion?

7. Let $x_1 < x_2 < \cdots < x_n$ be a sequence of n keys. In a randomized search tree storing this sequence what is the expected size of the subtree whose root stores x_k ?