**CHAPTER 11**

**Amortized Analysis**

**11.1** When the number of trees after the insertions is more than the number before.

**11.2** Although each insertion takes roughly log *N*, and each *deleteMin* takes 2 log *N* actual time, our accounting system is charging these particular operations as 2 for the insertion and 3 log *N* − 2 for the *deleteMin*. The total time is still the same; this is an accounting gimmick. If the number of insertions and *deleteMins* are roughly equivalent, then it really is just a gimmick and not very meaningful; the bound has more significance if, for instance, there are *N* insertions and *O*(*N*/log *N*) *deleteMins* (in which case, the total time is linear).

**11.3** Insert the sequence *N*, *N* + 1, *N*−1, *N* + 2, *N*−2, *N* + 3, . . . , 1, 2*N* into an initially empty skew heap. The right path has *N* nodes, so any operation could take **Ω** (*N*) time.

**11.5** We implement *decreaseKey(x)* as follows: If lowering the value of *x* creates a heap order violation, then cut *x* from its parent, which creates a new skew heap *H*1 with the new value of *x* as a root, and also makes the old skew heap (*H*) smaller. This operation might also increase the potential of *H*, but only by at most log *N*. We now merge *H* and *H*1. The total amortized time of the *merge* is *O*(log *N*), so the total time of the *decreaseKey* operation is *O*(log *N*).

**11.8** For the *zig-zig* case, the actual cost is 2, and the potential change is *Rf* (*X*) + *Rf* (*P*) + *Rf* (*G*)− *Ri* (*X*)−*Ri* (*P*)−*Ri* (*G*). This gives an amortized time bound of



Since *Rf* (*X*) = *Ri* (*G*), this reduces to



Also, *Rf* (*X*) > *Rf* (*P*) and *Ri* (*X*) < *Ri* (*P*) , so



Since *Si* (*X*) + *Sf* (*G*) < *Sf* (*X*), it follows that *Ri* (*X*) + *Rf* (*G*) < 2 *Rf* (*X*)−2. Thus



**11.9 (a)** Choose *W*(*i*) = 1/*N* for each item. Then for any access of node *X*, *Rf* (*X*) = 0, and *Ri* (*X*) ≥ −log *N*, so the amortized access for each item is at most 3 log *N* + 1, and the net potential drop over the sequence is at most *N* log *N*, giving a bound of *O*(*M* log *N* + *M* + *N* log *N*), as claimed.

**(b)** Assign a weight of *qi* /*M* to items *i*. Then *Rf* (*X*) = 0, *Ri* (*X*) ≥ log(*qi*/*M*), so the amortized cost of accessing item *i* is at most 3 log(*M*/*qi*) + 1, and the theorem follows immediately.

**11.10 (a)** To merge two splay trees *T*1 and *T*2, we access each node in the smaller tree and insert it into the larger tree. Each time a node is accessed, it joins a tree that is at least twice as large; thus a node can be inserted log *N* times. This tells us that in any sequence of *N* − 1 merges, there are at most *N* log *N* inserts, giving a time bound of *O*(*N* log2 *N*). This presumes that we keep track of the tree sizes. Philosophically, this is ugly since it defeats the purpose of self-adjustment.

**(b)** Port and Moffet [6] suggest the following algorithm: If *T*2 is the smaller tree, insert its root into *T*1. Then recursively merge the left subtrees of *T*1 and *T*2, and recursively merge their right subtrees. This algorithm is not analyzed; a variant in which the median of *T*2 is splayed to the root first is with a claim of *O*(*N* log *N*) for the sequence of merges.

**11.11** The potential function is *c* times the number of insertions since the last rehashing step, where *c* is a constant. For an insertion that doesn’t require rehashing, the actual time is 1, and the potential increases by *c*, for a cost of 1 + *c*.

If an insertion causes a table to be rehashed from size *S* to 2*S*, then the actual cost is 1 + *dS*, where *dS* represents the cost of initializing the new table and copying the old table back. A table that is rehashed when it reaches size *S* was last rehashed at size *S*/2, so *S*/2 insertions had taken place prior to the rehash, and the initial potential was *cS*/2. The new potential is 0, so the potential change is − *cS*/2, giving an amortized bound of (*d* − *c*/2)*S* + 1. We choose *c* = 2*d*, and obtain an *O*(1) amortized bound in both cases.

**11.12** We can (inductively) take a Fibonacci heap consisting of a single degenerate tree that extends as deep as possible. Insert three very small items; do a *deleteMin* to force a merge. This leaves two of the newer small items, one as a root, the other as a child of the root. Then do a *decreaseKey* on that child, and then a *deleteMin*; now we have a longer degenerate tree. Thus the worst case is *O*(*N*) depth.

**11.13 (a)** This problem is similar to Exercise 3.25. The first four operations are easy to implement by placing two stacks, *SL* and *SR* , next to each other (with bottoms touching). We can implement the fifth operation by using two more stacks, *ML* and *MR* (which hold minimums).

If both *SL* and *SR* never empty, then the operations can be implemented as follows:

*push(x):* push *x* onto *SL*; if *X* is smaller than or equal to the top of *ML*, push *x* onto *ML* as well.

*inject(x):* same operation as *push*, except use *SR* and *MR*.

*pop():* pop *SL*; if the popped item is equal to the top of *ML*, then pop *ML* as well.

*eject():* same operation as *pop*, except use *SR* and *MR*.

*findMin():* return the minimum of the top of *ML* and *MR*.

These operations don’t work if either *SL* or *SR* is empty. If a *pop* or *eject* is attempted on an empty stack, then we clear *ML* and *MR*. We then redistribute the elements so that half are in *SL* and the rest in *SR*, and adjust *ML* and *MR* to reflect what the state would be. We can then perform the *pop* or *eject* in the normal fashion. The following figure shows a transformation.



Define the potential function to be the absolute value of the number of elements in *SL* minus the number of elements in *SR*. Any operation that doesn’t empty *SL* or *SR* can increase the potential by only 1; since the actual time for these operations is constant, so is the amortized time.

To complete the proof, we show that the cost of a reorganization is *O*(1) amortized time. Without loss of generality, if *SR* is empty, then the actual cost of the reorganization is |*SL*| units. The potential before the reorganization is |*SL*|; afterward, it is at most 1. Thus the potential change is 1 − |*SL*|, and the amortized bound follows.

**11.15** No; a counterexample is that we alternate between finding a node at the root (with splay) and a node that is very deep (without splay).

**11.16** The maximum potential is *O*(*N* log *N*) for a splay tree of maximum depth. The minimum potential is *O*(*N*) for a balanced splay tree (the logic is similar to that used in analyzing the *fixHeap* operation for binary heaps.) The potential can decrease by at most *O*(*N* log *N*), on that tree (half the nodes lose all their potential). The potential can only increase by *O*(log *N*), based on the splay tree amortized bound.

**11.17 (a)** *O*(*N* log *N*).

**(b)** *O*(*N* log log *N*).