

This homework is due Thursday, April 16, in class. Please place it in the folder for homework at the start of class. The questions are drawn from the material in Chapter 17 of the text on amortized analysis, and from material in class on splay trees. The homework is worth a total of 100 points.

Please write on just one side of a page, do not use scrap paper, put your answers in the correct order, and staple your pages together. If you can't solve a problem, state this, and write only what you know to be correct. Neatness and conciseness count.

- (1) **(Simulating a queue using stacks)** (25 points) Show how to implement the *queue* data structure by using two *stacks*, so that the amortized time for queue operations in the stack-based implementation matches their worst case time in a standard queue implementation. More specifically, show how to implement the operations

- $\text{Put}(x, Q)$, which adds element x to the rear of queue Q , and
- $\text{Get}(Q)$, which removes the element x on the front of queue Q and returns x ,

so that both operations run in $O(1)$ amortized time. Use the *potential method* for your analysis.

- (2) **(bonus) (Stack with Multipush)** (10 points) Suppose you want to support a stack that has the operations Push , Pop , and Multipop as discussed in class, as well as the new operation

- $\text{Multipush}(A, k, S)$, which pushes all elements in the array $A[1:k]$ onto stack S .

This is equivalent to doing $\text{Push}(A[k], S)$, $\text{Push}(A[k-1], S)$, \dots , $\text{Push}(A[1], S)$.

Can Push , Pop , Multipop , and Multipush all be supported in $O(1)$ amortized time per operation?

(Note: If you feel this can be achieved, give an amortized analysis demonstrating it. If not, give an argument showing it is impossible.)

- (3) **(Deleting the larger half)** (35 points) Design a data structure that supports the following two operations on a set S of integers:

- $\text{Insert}(x, S)$, which inserts element x into set S , and
- $\text{DeleteLargerHalf}(S)$, which deletes the largest $\lceil |S|/2 \rceil$ elements from S .

Show how to implement this data structure so both operations take $O(1)$ amortized time. You may use the charging method or the potential method for your analysis.

(Hint: It may be useful to recall that the median element of an unsorted list can be found in linear time.)

- (4) **(Splittable heaps)** (40 points) A *splittable heap* is a data structure whose elements are (key, item)-pairs. The keys are real numbers, and the items are also drawn from a totally-ordered set. (Thus there are total orders on both keys and items, and these ordering relations may be different.) The keys in the pairs in a splittable heap are not necessarily distinct (in contrast to search trees), but the items are all distinct.

The operations on a splittable heap are:

- $\text{Insert}(k, x, H)$, which adds item x to heap H with associated key k . This assumes x is not already in H .

- **Extract**(H), which removes a pair from H that has the smallest key, and returns the item in the pair.
- **Decrease**(k, x, H), which updates the key associated with item x in H to be k , if k is smaller than the current key associated with x . This assumes item x is already in heap H .
- **Split**(x, H), which splits heap H into two heaps, H_1 and H_2 , according to item x . Heap H_1 contains all pairs in H whose items do not come after x in the total order on items, and heap H_2 contains all pairs in H whose items do come after x in the item order. **Split** destroys heap H and returns the two heaps H_1 and H_2 .
- **Join**(H_1, H_2), which joins heaps H_1 and H_2 into one heap, assuming that all items in H_1 come before all items in H_2 in the item order. **Join** destroys heaps H_1 and H_2 , and returns the merged heap.

Note that **Split** and **Join** are with respect to the *item order*, while **Extract** and **Decrease** are with respect to the *key order*. Note also that splitable heaps are different from the heap data-structure used in heap sort.

Show how to implement each of these five operations using the *splay tree* data-structure so that each operation takes $O(\log n)$ amortized time, where n is the largest number of pairs in the heap at any time. To do this, you will want to add and maintain some new fields in tree nodes. Be sure to explain the ideas behind your algorithms clearly.

(Remark: Splitable heaps are used in some implementations of graph algorithms, most notably Edmonds' algorithm for maximum-weight matching in a sparse general graph.)

Note that Problem (2) is a *bonus question*, and is not required (except for the honors section).