Advanced Algorithms, Fall 2014

Prof. Bernard Moret

Homework Assignment #1

due Sunday night, Sep. 28

Write your solutions in LaTeX using the template provided on the Moodle and web sites and upload your PDF file through Moodle by 4:00 am Tuesday morning, Oct. 1.

Question 1.

The following algorithm, when given a permutation, generates the next permutation in lexicographic order, changing the given permutation in-place.

```
Find the largest index k such that a[k] < a[k+1]. If no such index exists, exit (the permutation is the last permutation).
Find the largest index l such that a[k] < a[l].
Swap a[k] with a[l].
Reverse the sequence from a[k+1] up to and including the final element a[n].
```

Verify that the next permutation is generated in constant amortized time.

Question 2.

Using arrays, as in a simple queue or a binary heap, creates a problem: when the needs grow beyond the allocated array size, there is no support for simply increasing the array. Therefore consider the following solution.

- When the next insertion encounters a full array, the array is copied into one twice its size (and the old array returned to free storage) and the insertion then proceeds.
- When the next deletion reduces the number of elements below one quarter of the allocated size, the array is copied into one half its size (and the old array returned to free storage) and the deletion then proceeds.

Assume that returning an array to free storage takes constant time, but that creating a new array takes time proportional to its size.

- Prove that, in the context of a data structure that grows or shrinks one element at a time, the amortized cost per operation of array doubling and array halving is constant.
- Why not halve the array as soon as the number of elements falls below one half of the allocated size?

Question 3.

For a priority queue that supports insert and delete-min in $O(\log n)$ worst-case time (implemented, for example, using a binary search tree), design a potential function that makes the amortized costs of insert and delete-min $O(\log n)$ and $O(1)$, respectively.

Question 4.
An infinite stack is an unbounded series of stacks $S_0, S_1, \ldots$, where the i^{th} stack can contain at most 2^i elements. Operations pop and push on the infinite stack are implemented as follows. The operations are always started on the smallest stack, S_0. When an element is pushed on to a stack S_i that is already full, all elements in S_i are popped and then pushed onto S_{i+1} (one element at a time). If this shift causes stack S_{i+1} to become full in turn, all of its elements are popped and pushed onto stack S_{i+2}, and so on. If a pop operation empties a stack S_i (note that it must always empty stack S_0), then S_i is filled by popping the top 2^i elements of S_{i+1} and pushing them onto S_i; and this shift is again recursive, in that, if it results in emptying S_{i+1}, then S_{i+1} is filled by popping 2^{i+1} elements from S_{i+2}, and so on. Popping or pushing a single element on any stack takes $O(1)$ time.

1. Suppose we push 15 elements (let the elements be $1 \ldots 15$) onto an empty infinite stack and then pop 7 elements out. List the contents of stacks S_0, S_1, S_2, S_3, S_4 and S_5 in correct order after these 22 operations.

2. Analyse the worst-case time complexity of pop and push on the infinite stack.

3. Show that pop and push on the infinite stack *cannot* be achieved in amortized constant time—that is, give an initial state of the infinite stack and a (short) sequence of operations that returns the infinite stack to its initial state such that the cost of this sequence of operations grows faster than the number of operations in the sequence.