Algorithms
Computer Science 140 & Mathematics 168
Instructor: B. Thom
Fall 2004
Homework 3b
Due on Tuesday, 09/21/04 (beginning of class)

1. **[15 Points] Multipop Stacks with Automatic Backup!**

   CLRS Excercise 17.2-1 (page 412). You should also consider why this problem has a restricted stack size k. Could anything in your analysis change if k were constant? What about if k were a fraction of n?

2. **[25 Points] Extendible Arrays Revisited.** In class we examined extendible arrays. We showed that by doubling the length of the array each time we need to perform an extension, a sequence of n insertions takes time $O(n)$. Now imagine that we allow deletions in addition to insertions. Just as an insertion always inserted an element at the end of the array, a deletion always deletes the last element in the array. You can imagine that an array may become quite large but later, due to deletions, most of the array becomes empty. In this kind of scenario, it would be nice to contract the size of the array to give the memory back to the operating system.

   Professor I. Lai of the Pasadena Institute of Technology has suggested the following rule: Use the regular doubling rule to extend arrays. However, when an array becomes less than half full (because of deletions, presumably), allocate a new array of half the length of the current array and copy the elements into the new array. (At that point the old array is released and its memory is recovered by the operating system.)

   As in class, in these analyses you should assume that you start with an initially empty array and that requests for memory are handled in constant time.

   (a) Play the role of the "malicious adversary" and describe a sequence of n insert and delete operations for which Professor Lai's rule would incur a cost of $\Theta(n^2)$. Explain your analysis.

   (b) Professor Lai was not granted tenure and he was replaced by Professor Anna Litik. Professor Litik had a much better idea: Use the regular doubling rule to extend arrays. However, only contract an array when it becomes less than or equal to 1/4 full. At that point, allocate a new array of half the length of the current array and copy the elements into the new array. Use an amortization argument to explain why under this scheme any sequence of n inserts and deletes costs a total of $O(n)$ time. Still use 3 rubles for the insertions but now explain how much the deletions should pay and why this all works! (While allocating a new block of memory can be done in constant time, copying k elements from one array to another takes k time.)

   (c) Professor Litik's colleague, Professor Polly Nomial, has proposed another variant of extendible arrays: Still just double the array when it becomes full. However,

we contract the array (due to deletions, presumably) only when it becomes less than or equal to 1/3 full. At that point, allocate a new array of 2/3 the length of the current array and copy the elements into that new array. Does a sequence of n inserts and deletes still cost a total of $O(n)$ time in this scheme? If so, give a clear amortization argument to explain why. If not, describe a sequence of n inserts and deletes which would cause this scheme to use more than $O(n)$ time.

3. **[20 points] Amortizing your Heap!** Recall that, for any node i in a heap, `heapify`(i) runs in at most the height of the subtree rooted at i. This fact is true because heapify percolates i downwards, swapping with one of i's children until either a leaf is reached or the desired heap property is maintained. Thus, `heapify` can be used to build a heap as follows:

```
BuildHeap(A[1,...,i,...,n]) {
  for i = n downto 1
    heapify(i)
}
```

Since the height of the overall tree is $O(\log n)$, the function's runtime is clearly $O(n \log n)$. Your task in this problem is to use amortization (the accounting method) to argue the tighter bound mentioned in class: $\Theta(n)$. As appropriate, use the following notation: $h(i)$ is the height of the subtree rooted at i.

Here's how to proceed: Before `BuildHeap` even runs, give every node some number of rubles in which to pay for future work. Your goal is then to argue that, with your proposed payment scheme, heapify's work at each node i is paid for. This will be easiest to show via induction. In particular, you'll want to prove the following invariant: when `BuildHeap` is called on node i, there are $h(i)$ rubles available to pay for that `heapify` operation. (You'll want to make this argument with respect to whatever accounting scheme you propose.) After proving this invariant, link this back into the main task, i.e. why is `BuildHeap` $\Theta(n)$?

4. **[15 Points] Order Statistics Revisited.** In class we showed that the recursive **Select** algorithm runs in time $O(n \lg n)$ if the array is partitioned into groups of 3 but runs in time $O(n)$ if the array is partitioned into groups of 5. Now we'll investigate what happens if the algorithm is implemented so that the array is partitioned into groups of 7. Give the recurrence relation that arises when groups of 7 are used, and explain why your recurrence relation is correct. Then use the analysis technique that we used in class to derive the Big-Oh running time of this variant of the algorithm. Explain your analysis very carefully!