# 15-451 Algorithms, Fall 2012

**Homework # 3** **Due: October 9, 2012**

Please hand in each problem on a separate sheet and put your **name** and **recitation** (time or letter) at the top of each page. You will be handing each problem into a separate box in lecture, and we will then give homeworks back in recitation.

Remember: written homeworks are to be done *individually*. Group work is only for the oral-presentation assignments.

---

**1. Dequeue.** [25 pts]

A dequeue is data structure that represents an ordered list of items. It allows access to both ends of the list. (The ends are called the *front* and *back* of the list.) Specifically it supports the following operations:

d.isempty(): return `true` if the dequeue is empty

d.push($x$): add $x$ to the front of the dequeue

d.pop(): remove and return the element at the front of the dequeue

d.inject($x$): add $x$ to the back of the dequeue

d.eject(): remove and return the element at the back of the dequeue

A stack is a similar data structure that just allows the s.isempty() s.push() and s.pop() operations.

Describe how to implement a dequeue using two stacks. (Your data structure can maintain additional bookkeeping information, such as the number of elements in the dequeue. But it should not make use of an array or other data structures to store the data in the dequeue.)

Your algorithm should be constant amortized time per operation. To be concrete, measure the cost of an operation in terms of the number of stack pushes and pops that it does. Find small constants $c_1$ and $c_2$ such that the amortized costs of d.push() and d.inject() are at most $c_1$ and the amortized costs of d.pop() and d.eject() are at most $c_2$. Prove your result using a potential function.

**2. The List-Update Problem.** [25 pts]

Suppose we have $n$ data items $x_1, x_2, \ldots, x_n$ that we wish to store in a linked list in some order. Let's say the cost for performing a *lookup*($x$) operation is \$1 if $x$ is in the head of the list, \$2 if $x$ is the second element in the list, and so on.

For instance, say there are 4 items and it turns out that we end up accessing $x_1$ 3 times, $x_2$ 5 times, $x_3$ once, and $x_4$ twice. In this case, in hindsight, the best ordering for a linked list would have been $(x_2, x_1, x_4, x_3)$ with a total cost of \$21.

The *Move-to-Front* (MTF) strategy is the following algorithm for organizing the list if we don't know in advance how many times we will access each element. We begin with the elements in their initial order $(x_1, x_2, \ldots, x_n)$. Then, whenever we perform a *lookup*($x$)

operation, we move the item accessed to the front of the list. Let us say that performing the movement is free. For instance, if the first operation was $lookup(x_3)$, then we pay \$3, and afterwards the list will look like $(x_3, x_1, x_2, x_4 \ldots)$.

1. Suppose $n = 4$ and we use MTF starting from the order $(x_1, x_2, x_3, x_4)$. If we perform the following 4 operations:

   $$lookup(x_4), lookup(x_2), lookup(x_4), lookup(x_2).$$

   What does the list look like in the end and what was the total cost?

2. Your job is to prove that the total cost of the MTF algorithm on a sequence of $m$ operations (think of $m$ as much larger than $n$) is at most $2C_{static} + n^2$ where $C_{static}$ is the cost of the best static list in hindsight for those $m$ operations (like in our first example). We will prove this in two steps.

   (a) First prove the somewhat easier statement that the cost of Move-to-Front is at most $2C_{initial}$ where $C_{initial}$ is the cost of the original ordering $(x_1, x_2, \ldots, x_n)$.

   *Hint*: If $i < j$ but $x_j$ is in front of $x_i$ in the MTF list, let's say that $x_j$ has "cut in line" in front of $x_i$. Now, imagine that each element $x_i$ has a bank account with \$1 for everyone that is currently cutting in line in front of it.

   (b) Now prove the $2C_{static} + n^2$ bound.

Note: one nice use of this is for *data compression*. You store each ascii character in a list in this way, and then when reading a string of text, for each character you output its index $i$ in the list before moving the character to the front (this requires only $O(\log i)$ bits, which will be small if the item was close to the front of the list).

**3. Road trip!** [25 pts]

You and your friends decide to go on a road trip to San Francisco immediately after midterms. To plan the trip, you have laid out a map of the U.S., and marked all the places you think might be interesting to visit along the way. However, the requirements are:

1. Each stop on the trip must be strictly closer to SF than the previous stop.
2. The total length of the trip can be no longer than $D$.

You and your friends want to visit the most places subject to these conditions. As a first step, you create a directed graph with $n$ nodes (one for each location of interest) and an edge from $i$ to $j$ if there is a road from $i$ to $j$ and $j$ is closer to SF than $i$. Let $d_{ij}$ be the length of edge $(i, j)$ in this graph.

Give an $O(mn)$-time algorithm to solve your optimization problem. Specifically, given a directed acyclic graph (DAG) $G$ with $n$ nodes, $m$ edges and with lengths on the edges, and given a start node $s$, a destination node $t$, and a distance bound $D$, your algorithm should find the path in $G$ from $s$ to $t$ that visits the most intermediate nodes, subject to having total length $\leq D$.

(Note that in general graphs, this problem is NP-complete: in particular, a solution to this problem would allow one to solve the *traveling salesman problem*. However, the case that $G$ is a DAG is much easier.)

**4. Optimal BSTs.** [25 pts]

Consider a binary search tree storing a set of keys $x_1 < x_2 < x_3 < \ldots < x_n$. Let's define the *cost* of handling a request for some key to be the number of comparisions made in searching for it (1 plus the distance of the node from the root of the tree). For example, if the root is requested, the cost is 1.

Given a particular sequence of requests, one can calculate the cost that would be incurred on that sequence by different possible binary search trees. (We are thinking of the trees as static, not self-adjusting as in Splay Trees.) Out of all possible BSTs, the one that attains the minimum cost is called the *optimal binary search tree* for that sequence.[1]

1. For a fixed tree, the cost of a given sequence of requests clearly only depends on the number of times each key is requested, not on their order. Suppose that $n = 4$ and that $x_1$ is accessed twice, $x_2$ is accessed 4 times, $x_3$ is accessed 3 times, and $x_4$ is accessed 6 times. It turns out there are multiple different optimal binary trees for this set of requests. Find two of them.

2. In general, suppose the optimal binary search tree for a given set of requests has $x_i$ at the root, with $L$ as its left subtree and $R$ as its right subtree. Prove that $L$ must be an optimal binary search tree for the requests to elements $x_1, \ldots, x_{i-1}$ and $R$ must be an optimal binary search tree for the requests to elements $x_{i+1}, \ldots, x_n$.

3. Give a general algorithm for constructing the optimal binary tree given a sequence of counts $c_1, c_2, \ldots, c_n$ ($c_i$ is the number of times $x_i$ is accessed). The running time of your algorithm should be $O(n^3)$. Hint: use dynamic programming.

Note #1: the notion of an optimal binary search tree is a lot like the notion of a Huffman tree, except that we also require the keys to be in search-tree order. This requirement is the reason that the greedy Huffman-tree algorithm doesn't work for finding optimal BSTs.

Note #2: it's actually possible to improve the running time to $O(n^2)$ by a simple modification to this dynamic-programming solution. But proving correctness for this faster version is very tricky.

[1] This is the exactly analogous to the "best static list in hindsight" in the previous problem. The big difference is that for *lists*, the best static list in hindsight is very simple: you just put the most-requested item in the front, then the next-most-requested, etc. For *trees*, computing the optimal binary search tree will be more complicated, requiring dynamic programming, which is what this whole problem is about.