Arjun Biddanda (aab227)
Muhammad Khan (mhk98)
CS 3110 - Fall 2012
2:30 PHL Discussion
11/30/12

Problem Set 6

Part 9: Analysis of Algorithms for the Zardoz Array

1) SEARCH

For a collection of n elements, the zardoz array consists of k sorted arrays, where $k \equiv \lceil \log_2(n+1) \rceil$, such that the binary representation of n is $n_{k-1}n_{k-2}\cdots n_1 n_0$ and each sorted array A_i is empty if $n_i = 0$ and has 2^i elements if $n_i = 1$. Performing a complexity analysis on the search operation for this data structure, we see that in the worst case, the element we seek is in the kth sorted array, which will require $O(k)$ time to reach. In this array, performing a binary search would take $O(log_2 2^i) = O(i)$ time. In the kth sorted array, $i = k-1$, thus the binary search algorithm also takes $O(k)$ time. Thus, together, in the worst case, searching in a zardoz array takes $O(k^2)$ time.

2) INSERT

When inserting an element there are three cases which must be looked at:

$$\begin{gathered}
1)\ \cdots\ 0 \rightarrow \cdots\ 1 \\
2)\ \cdots\ 01 \rightarrow \cdots\ 10 \\
3)\ 111\cdots\ 111 \rightarrow 100\cdots\ 001
\end{gathered}$$

Case (1) corresponds to the situation when the binary representation of n ends in a 0, which means that the binary representation of $n+1$ is equivalent to that of n except this last 0 is replaced by a 1. The original zardoz array (before insertion) will have A_0 empty and A_1 filled with $2^1 = 2$ elements. To insert a new element, we simply set the only element of A_0 to be this newly inserted element, and we have a new zardoz array. This operation takes $O(1)$ time.

Case (2) corresponds to the situation where the last two digits of the binary representation of n are 01. This translates to a zardoz array where A_0 is full with one element and A_1 is empty. If A_1 were to be full, it would have to have two elements. When inserting an element a into a zardoz array of this nature, we simply take a and the element of A_0 and put them into one array which becomes the new A_1. Sorting this two-element array is a constant-time ($O(1)$) operation.

Case (3), likewise, corresponds to the situation when the entire binary representation is a string of 1s. For any n in this case, there are k 1s. Adding one element to this collection will result in $n+1$ elements, which has a binary representation of $100\cdots 001$ ($k-1$ zeroes), which means the only filled sorted arrays are A_0 and A_k. These arrays will be formed by taking the newly added element to fill up the new A_0 (constant time operation). The remaining n elements are sorted amongst each other to fill up A_k. This stage will be implemented using the heapsort algorithm, which in its worst case is linearithmic: $O(n \log n)$. Thus, in this case only, insertion takes $O(n \log n)$ time.

To analyze amortized complexity, we will use the potential method, with a potential function given by

$$\Phi(n) = n - k$$

with n the number of elements and k as defined previously. It is easy to see that $\Phi(0) = 0$. During the fast operations (cases 1 and 2), the value of k remains unchanged, so the amortized time for a fast operation is

$$c + \Phi(n+1) - \Phi(n) = 1 + (n+1) - k - (n-k) = 2$$

The substitution $c = 1$ arises from the fact that a fast operation is $O(1)$.

Likewise, during a slow operation, k increases by 1 after the insertion, so the amortized time is

$$c + \Phi(n+1) - \Phi(n) = n \log n + (n+1) - (k+1) - (n-k) = n \log n$$

After a series of insertions, the value of n gradually increases, so the frequency of slow operations decreases. Thus, the cost of insertion is dominated by the cost of fast operations. Thus, insertion takes amortized constant time $O(1)$ which satisfies the $o(n)$ condition.