Analysis of Algorithm, Homework# 4

Fei Dai   SS# 900-01-9382

November 20, 2001

1. **Making binary search dynamic**

   Binary search of sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

   Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. We have k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, there is no particular relationship between elements in different arrays.

   (a) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

   Solution: Suppose $\text{BSEARCH}(A, size, key)$ is the binary search function, which returns the position of key in array A or returns 0 when key is not in A. The following algorithm searches key in the set of sorted arrays A holding n elements altogether.

   $\text{SEARCH}(A, n, key)$

   1: $i \leftarrow 0$
   2: **while** $n > 0$ **do**
   3:   **if** n is odd **then**
   4:     $pos \leftarrow \text{BSEARCH}(A_i, 2^i, key)$
   5:     **if** $pos \neq 0$ **then**
   6:       **return** (i, pos)
   7:     **end if**
   8:   **end if**
   9:   $n \leftarrow n/2$
   10:   $i \leftarrow i + 1$
   11: **end while**

12: **return** "not found"

In the worst case, BSEARCH will be called $k = \lceil \lg(n+1) \rceil$ times, with $size = 2^i$, for $i = 0, 1, \ldots, k-1$. For each invocation of BSEARCH$(A_i, 2^i, key)$, the search time is $O(i)$. Therefore, the total complexity is $\sum_{i=0}^{k-1} O(i) = O(k^2) = O(\lg(n)^2)$. □

(b) Describe how to insert a new element into this data structure. Analyze its worst-case and amortized running times.

Solution: Suppose MERGE(A_1, A_2, A_3) combines two sorted arrays A_1 and A_2 into a sorted array A_3. The following INSERT algorithm insert a new element e into the set of sorted arrays A.

INSERT(A, n, e)

1: $i \leftarrow 0$
2: $B_0 \leftarrow \{e\}$
3: **while** n is odd **do**
4:     MERGE(A_i, B_i, B_{i+1})
5:     $n \leftarrow n/2$
6:     $i \leftarrow i+1$
7: **end while**
8: $A_i \leftarrow B_i$

The cost to produce B_i for $i = 0, 1, \ldots, k-1$ is $O(2^i)$ (lines 2,4). We omit the the cost of array assignment (line 8) because it can be avoided via a simple condition test in lines 2 and 4. Therefore, the total cost for a INSERT operation is $\sum_{i=0}^{k_n} O(2^i) = O(2^{k_n+1})$ where k_n is the least i satisfying $n_i = 0$.

For a sequence of INSERT operations with $n = 1, 2, \ldots, N$, B_0 is produced for N times, B_1 is produced for $\lfloor \frac{N}{2} \rfloor$ times, ..., B_i is produced for $\lfloor \frac{N}{2^i} \rfloor$ times, ..., and B_k is produced once. So the total cost is

$$\sum_{i=0}^{k-1} \lfloor \frac{N}{2^i} \rfloor O(2^i) \leq \sum_{i=0}^{k-1} O(2^k) \leq O(2^k k) = O(N \lg N)$$

Therefore, the amortized cost for each of the N operations is $O(\lg N)$. □

(c) Discuss how to implement DELETE.

Solution: Suppose REPLACE(A_i, pos, e) assign a new value to $A_i[pos]$ and adjust its position within A_i to maintain its sorted property. The following DELETE algorithm delete a element $A_i[pos]$ from the set of sorted arrays A.

DELETE(A, n, i, pos)

1: $k \leftarrow 0$
2: **while** n is even **do**
3:     $n \leftarrow n/2$
4:     $k \leftarrow k + 1$
5: **end while**
6: **if** $i = k$ **then**
7:     REPLACE$(A_k(1 : 2^k - 1), pos, A_k[2^k])$
8: **else**
9:     REPLACE$(A_i, pos, A_k[2^k])$
10: **end if**
11: **for** $i \leftarrow 0$ **to** $k - 1$ **do**
12:     $A_i \leftarrow A_k(2^i : 2^{i+1} - 1)$
13: **end for**
14: $A_i \leftarrow \emptyset$

$\square$

2. (a) You are given n keys and an integer k such that $1 \leq k \leq n$. Given an efficient algorithm to find *any one* of the k smallest keys. (For example, if $k = 3$, the algorithm may provide the first-, second-, or third-smallest key. It need not know the exact rank of the key it outputs.) How many key comparisons does your algorithm do? *Hint*: Don't look for something complicated. One insight gives a short, simple algorithm.

   (b) Give a lower bound, as a function of n and k, on the number of comparisons needed to solve this problem.

Solutions:

   (a) The following algorithm solve the problem with $n - k$ comparisons.

   ONE-OF-k-SMALLEST-KEYS(A, n, k)
   1: $min \leftarrow A[1]$
   2: **for** $i \leftarrow 2$ **to** $n - k + 1$ **do**
   3:     **if** $A[i] < min$ **then**
   4:        $min \leftarrow A[i]$
   5:     **end if**
   6: **end for**
   7: **return** min

   (b) We can prove that the low bound for this problem is also $n - k$ comparisons. Suppose there is an algorithm that can find one of the k smallest keys, say v, in less than $n - k$

3

comparisons. However, this algorithm can produce at most $n - k - 1$ winners (i.e., keys found to be the larger one in at least one comparison). Among the other $k + 1$ non-winners, we can assign values smaller than v to at least k keys. Therefore v is not one of the k smallest keys, which is a contradiction.

$\square$