CSCI-GA.1170-001/002 Fundamental Algorithms	October 29, 2015

Solutions to Problem 1 of Homework 5 (15 points)

Name: GOWTHAM GOLI (N17656180) — *Due: Tuesday, October 13*

Assume you are given a data structure D which supports the following two operations:

- INSERT$(D, value)$. Inserts a value *value* into D. If D has n elements, assume this procedure takes $I(n)$ time.
- SEARCH$(D, value)$. It D contains at least one element equal to *value*, return the pointer to this element (else returns *nil*). Assume this procedure takes $S(n)$ time.
- INORDERWALK(D). Outputs all n elements of D in sorted order. Assume this procedure takes linear time $O(n)$.

Using D, you would like to build a new data structure R, which can deal with many repeated elements more efficiently, by supporting the following operations.

- ADD$(R, value)$. Inserts a value *value* into R.
- FREQUENCY$(R, value)$. Returns the number of elements of R equal to *value* (i.e., how many times was ADD$(R, value)$ called before).
- FASTINORDERWALK(R). Outputs all *distinct* elements of D in sorted order, together with their frequency values.

(a) (5 pts) Using D, show how to implement R, so that the following is true. If R contains n records, but only t of them are distinct, where t could be much less than n, then

- ADD(R, key) should run in time $A(n, t) \approx I(t) + S(t)$;
- FREQUENCY(R, key) should run in time $F(n, t) \approx S(t)$;
- FASTINORDERWALK(R) should run in time $O(t)$.

Namely, all run times are *independent of* n. For example, if ADD has been called 4 times on $(R, 7)$ and 5 times on $(R, 6)$ then FREQUENCY$(R, 3)$ returns 0 but FREQUENCY$(R, 7)$ returns 4, and both calls take time $F(9, 2) \approx S(2)$, where $t = 2$ because only two distinct values were inserted so far (despite $n = 4+5 = 9$). Also, FASTINORDERWALK(R) will output $(6, 5), (7, 4)$ in time $O(2)$.
(**Hint**: Add a field $v.num$ in addition to $v.key$, which counts how many elements are equal to $v.key$.)

Solution:

Implementing R using D

Maintain a field called num for each node in D which maintains the frequency of that node and eliminate all the other duplicates from D. This new data structure formed is R (i.e R is similar to D, the only difference is R has all distinct elements in it and maintains an additional frequency field). Therefore if D has t distinct elements in it then number of elements in R is t.

$\textsc{Insert}(R, value)$ takes $I(t)$ time
$\textsc{Search}(R, value)$ takes $S(t)$ time
$\textsc{InOrderWalk}(R)$ takes $O(t)$ time

$\textsc{Add}(R, key)$

- Search if the key is present in R. Time taken in this step is $S(t)$
- If there is a node v such that $v.key = key$ then $v.num = v.num + 1$. Time taken in this step is $O(1)$
- If there is no such node v then insert the key into R. Time taken in this step is $I(t)$

$\therefore A(n,t) \approx I(t) + S(t)$

$\textsc{Frequency}(R, key)$

- Search if the key is present in R. Time taken in this step is $S(t)$
- If there is a node v such that $v.key = key$ then return $v.num$
- Else return 0

$\therefore F(n,t) = S(t)$

$\textsc{FastInOrderWalk}(R, key)$

- Just calls $\textsc{InOrderWalk}(R)$ and also return the frequency of each node along with it's key. Therefore, the procedure returns the list of tuples $(v.key, v.num)$ sorted by $v.key$

$\therefore$ Time taken is $O(t)$ $\square$

(b) (5 pts) For each of the following implementations of D, compute the running times $A(n,t)$ and $F(n,t)$ of $\textsc{Add}$ and $\textsc{Frequency}$ that you get by using your solution from part (a). Which data structure is the best? Make sure to justify your answers.

- Implement D as a linked list.
- Implement D as a sorted array.
- Implement D as a 2-3-tree.

Solution:

D as a linked list

$S(t) = O(t)$ and $I(t) = O(1)$

$\therefore A(n, t) = O(t + 1) = O(t)$, $F(n, t) = O(t)$ and FastInOrderWalk takes $O(t)$ time

D as a sorted array

$S(t) = O(\log t)$ and $I(t) = O(t)$

$\therefore A(n, t) = O(t + \log t) = O(t)$, $F(n, t) = O(\log t)$ and FastInOrderWalk takes $O(t)$ time

D as a 2-3 tree

$S(t) = O(\log t)$ and $I(t) = O(\log t)$ $\therefore A(n, t) = O(\log t + \log t) = O(\log t)$, $F(n, t) = O(\log t)$ and FastInOrderWalk takes $O(t)$ time

Therefore the best data structure to use would be a 2-3 Tree □

(c) (5 pts) Using the best data structure developed in part (b), give an algorithm for sorting n integers with at most t distinct values in time $O(n \log t)$. Make sure you justify your running time bound.

Solution:

R in this case is a 2-3 Tree. Therefore, each leaf node stores two fields, *key* and *num*. Also, in 2-3 Trees, the leaf nodes when seen from left to right are in a sorted order. So starting from the leftmost node keep calling it's successor until we reach the rightmost node. (Computing successor of a node takes $O(\log t)$ time which is proved in Problem-4)

```
Algorithm: Sort(T)
A ← NewArray(n)
c ← 0
v ← Left-most leaf node in T
while v is not NULL do
    for i ← 1 to v.num do
        A[c] ← v.key
        c ← c + 1
    end
    v ← Successor(v)
end
Return A
```

Algorithm 1: Sorting n integers with t distinct values in $O(n \log t)$ time

Time Complexity

Time taken to find the left most leaf node in the first step is $O(\log t)$. The while loop runs for t times and in each iteration we call SUCCESSOR which takes $O(\log t)$ (proof in Problem-4), the *for* loop iterates for number of repetitions of that element. Hence the overall run time of *for* loop is $O(n)$ and SUCCESSOR is $O(t \log t)$. Therefore, the total running time is $O(n + t\log t) = O(n \log t)$ $\square$