Data Structures Homework #5

Ari Feiglin

Question: 5.1:

Describe a data structure where given an array of size n of numbers which supports the following methods:

(1) Construction: Input the data structure with an array of n elements in $\mathcal{O}(n)$ time.

(2) Querying: Given a number k in the range $1 \leq k \leq n$, returns the kth number in size in $\mathcal{O}(k)$ time.

(1) The idea behind the construction is to partially sort the array. We want it so that after the construction phase, the array will be split up into blocks. Each block won't be necessarily ordered, but every number in it will be larger than (or equal to) every number in every previous block.

Suppose the input array is A. We can do the construction iteratively: suppose we have a function `PartitionMedian`(i) which partitions the subarray $A[1 \dots i]$ around its median. It does this using `QuickSelect` on the element of the $\frac{i}{2}$th (or more specifically, $\left\lfloor \frac{i}{2} \right\rfloor$th) size in the subarray. As we know, `QuickSelect` also partitions the array so that for every element with index $< \frac{i}{2}$, its value is less than the median, and for every element with index $> \frac{i}{2}$, its value is greater than the median.

We can run `PartitionMedian` on the whole array: `PartitionMedian`(n)), then `PartitionMedian` $\left(\frac{n}{2}\right)$, and so on. So the algorithm for constructing is:

Input : An array $A[1 \dots n]$ of size n.
Output: A datastructure ready for querying. This is done by partitioning A into blocks.

```
1 while n > 1 do
2 |   PartitionMedian(n);
3 |   n ← ⌊n/2⌋;
4 end
```

After this, the array will be properly partitioned. This is because for every subarray $A\left[1 \dots \left\lfloor \frac{n}{2^{i}} \right\rfloor\right]$, $A\left[1 \dots \left\lfloor \frac{n}{2^{i+1}} \right\rfloor\right] \leq A\left[\left\lfloor \frac{n}{2^{i+1}} \right\rfloor \dots \left\lfloor \frac{n}{2^{i}} \right\rfloor\right]$ As this is exactly what `PartitionMedian` does.

Since `PartitionMedian` $\left(\left\lfloor \frac{n}{2^{i}} \right\rfloor\right) \in \mathcal{O}\left(\frac{n}{2^{i}}\right)$, and the loop is repeated until $\frac{n}{2^{i}}$, the time complexity is:

$$\sum_{i=0}^{\log n} \mathcal{O}\left(\frac{n}{2^{i}}\right) = \mathcal{O}\left(n \cdot \sum_{i=0}^{\log n} 2^{-i}\right) = \mathcal{O}\left(2n \cdot \left(1 - \frac{1}{n}\right)\right) = \mathcal{O}(n)$$

As required.

(2) We can assume that $n = 2^{k}$ for some k to simplify the math (if we want to be rigorous, we can add elements to the original array with values of ∞ in order to make the size a power of two. This wouldn't affect the time complexity since the new size would be $2^{\lceil \log n \rceil}$ which is linear with n.)

Given a number k, let $\ell = \lfloor \log k \rfloor$.

We know that the number at index 2^{ℓ} is the 2^{ℓ}th smallest number in the array since it had to be chosen as a median in the construction phase. And similarly the number at index $2^{\ell+1}$ is the $2^{\ell+1}$th smallest number since it was also chosen as a median. Since every number before 2^{ℓ} is smaller than it and $2^{\ell} \leq k$, the kth

smallest number must be after index 2^ℓ (inclusive). And every number after $2^{\ell+1}$ is larger than it and $2^{\ell+1} \geq k$, the kth smallest number must be in between the indexes 2^ℓ and $2^{\ell+1}$.

Furthermore, every number between these indexes must be between the values at these indexes (since the array is partitioned). So the kth smallest element in A must be the $k-2^\ell$ smallest element in $A\left[2^\ell \dots 2^{\ell+1}\right)$. Since the length of this subarray is $2^{\ell+1} - 2^\ell = 2^\ell$, we can perform `QuickSelect` on it for the $k - 2^\ell$th smallest number in the subarray which gives us the required number. Since `QuickSelect` takes linear time relative to the size of the array, this takes $\mathcal{O}\left(2^\ell\right) = \mathcal{O}\left(k\right)$ time, as required.

Question: 5.2:

> **Definition:**
>
> A **significant element** in an array $A[1 \dots n]$ is an element which is in the array more than $\frac{n}{3}$ times.

Provide an algorithm where given an array $A[1 \dots n]$ returns every significant element in the array. Compute the time complexity of the algorithm.

Notice if B is A sorted, and if m is a significant element in A, then either $B\left[\left\lfloor \frac{n}{3} \right\rfloor + 1\right] = m$ or $B\left[n - \left\lfloor \frac{n}{3} \right\rfloor\right] = m$. This is because the set of all indexes where $B[i] = m$ forms a subarray of B whose length is greater than $\frac{n}{3}$. (Since B is sorted, all the ms must be adjacent and there are more than $\frac{n}{3}$ ms.) Also notice that this subarray must have a length greater than or equal to $\left\lfloor \frac{n}{3} \right\rfloor + 1$ (since it is the first integer greater than $\frac{n}{3}$).

So if $B\left[\left\lfloor \frac{n}{3} \right\rfloor + 1\right] \neq m$, then the subarray of ms must begin after this index, as there are not enough indexes before it. If the subarray starts at the next index, then it must span indexes $\left\lfloor \frac{n}{3} \right\rfloor + 2$ to $2\left\lfloor \frac{n}{3} \right\rfloor + 2$. If the subarray starts at the last index, n, then it must span the indexes $n - \left\lfloor \frac{n}{3} \right\rfloor$ to n.

In the first case, we know that $n - \left\lfloor \frac{n}{3} \right\rfloor$ is in this range since it is less than $2\left\lfloor \frac{n}{3} \right\rfloor + 2$:

$$n - \left\lfloor \frac{n}{3} \right\rfloor \leq 2\left\lfloor \frac{n}{3} \right\rfloor + 2 \iff n - 2 \leq 3\left\lfloor \frac{n}{3} \right\rfloor$$

Which is true.

And in the second case, $n - \left\lfloor \frac{n}{3} \right\rfloor$ is trivially in that range.

What this means is that if m is a significant element in A, then it must either be the $\left\lfloor \frac{n}{3} \right\rfloor + 1$ smallest element or the $n - \left\lfloor \frac{n}{3} \right\rfloor$ smallest. We can use `QuickSelect` to find these two smallest elements, which takes $\mathcal{O}(n)$ time, and then iterate over A and count the number of occurrences of each of these two elements, and verify that they both occur more than $\frac{n}{3}$ times. This takes $\mathcal{O}(n)$ time as well since we're simply iterating over the array.

Then we return the elements out of those two which occur more than $\frac{n}{3}$ times per the count. This takes $\mathcal{O}(n)$ time since each part of this algorithm takes linear time (as explained above), as required.

Question: 5.3:

We need to create a data structure which supports operations on integers. When we create the data structure, we will pass a parameter m to it, and this m never changes. The data structure will contain `(key, value)` pairs where the keys are (distinct) integers. Every number inserted into the data structure will be a natural number between 1 and m (inclusive). Construct the data structure such that it supports the following operations:

(1) Initialization of an empty data structure with parameter m in $\mathcal{O}(1)$ time.

(2) Construction of the data structure given a set of key, value pairs S and the parameter m in $\mathcal{O}(n)$ time.

(3) Insertion of an element x in $\mathcal{O}(\log n)$ time.

(4) Erasure of an element x in $\mathcal{O}(\log n)$ time.

(5) Search for a key x in $\mathcal{O}(\log n)$ time.

Note:

The concept of this data structure storing `(key, value)` pairs was not explicitly stated in the question itself, but if I understand Dvir Fried's answer on Moodle to this question correctly, that is the point of this question.

Nevertheless, I think I may have misunderstood the question since my time complexity is *too* good and my answer too simple. Unfortunately I am quite stressed for time right now due to circumstances beyond my control and I don't have the luxury of the ability to wait for someone to clarify the question for me.

So while my answer is probably *quite* wrong, mercy, please.

We can implement this data structure with a simple array, A. Since each key, k, is between 1 and m, we can place the value in the kth index of A. Similarly we can erase the element with key k by setting $A[k]$ to some default value. Searching just returns $A[k]$. Thus insertion, erasure, and searching have a time complexity of $\mathcal{O}(1)$, which is trivially a subset of $\mathcal{O}(\log n)$.

Initializing an empty data structure just creates an array of size m, which also takes constant time.

Given a set S and m, we create an array of size m and iterate over S. For every key-value pair (k, v), we set $A[k] = v$. This takes $\mathcal{O}(n)$ time since we simply iterate over S.

Question: 5.4:

We are given an array of integers in base 10. Different numbers can have a different number of digits, but the sum of all the digits in the array is n.

(1) What is the time complexity of Radixsort on such an array? Given an example of an input which gives the worst case time complexity.

(2) Provide an algorithm which sorts the array in $\mathcal{O}(n)$ time.

(1) We know that Radixsort has a time complexity of $\Theta(d(m+R))$ where d is the maximum number of digits in a number, m is the number of numbers in the array, and R is the base which in this case is 10.

We also know that $m \leq n$ since each number must have at least 1 digit, so the sum of all the digits (n) must be greater than or equal to m.

And $d \leq n$ as well since the sum of the number of digits is greater than the maximum number of digits.

So Radixsort in this case has a time complexity of $\mathcal{O}(n(n+10)) = \mathcal{O}(n^2)$.

The worst case time complexity can be obtained with an array of length n where one number has $\frac{n}{2}$ digits and the rest have 1 digit. Thus $d = \frac{n}{2}$ and $m = n$ and the time complexity in this case is $\frac{n^2}{2} \in \Theta(n^2)$ since Radixsort must still sort $\frac{n}{2}$ arrays of length n (which are really the decimal places at a certain point) even though all but 1 of the numbers in each array will be 0 (barring the first one).

(2) Notice that the issue with Radixsort is that it is sorting "extra" arrays. So we want to figure out a way to only sort numbers when necessary.

We can do this by iterating over the array (which I'll call A) and partitioning it into smaller arrays. Suppose $x \in A$ has length k, then we'll append it to the array A_k. This process takes $\mathcal{O}(n)$ time since we just iterate over A (if it is also somehow necessary to iterate over the digits in the number to find how many digits it has, it'll still take $\mathcal{O}(n)$ time since this just requires that we iterate over every digit in the array, and we know there are n digits total).

Let ℓ_k be the length of A_k, so the sum of all ℓ_ks, $\sum \ell_k$, equals the length of the array, which is less than or equal to n (we're only summing over ks where $\ell_k > 0$). And we know that $\sum k\ell_k = n$ since this is the sum of all the digits in A. Also note that $\sum k \leq \sum k\ell_k = n$ if we sum over every k where $\ell_k > 0$.

We can then perform Radixsort on each array A_k. For every k, performing Radixsort on A_k will take $\mathcal{O}(k(\ell_k + 10))$ time. So all in all this takes:

$$\mathcal{O}\left(\sum k(\ell_k + 10)\right) = \mathcal{O}\left(\sum k\ell_k + 10\sum k\right) = \mathcal{O}(n + 10n) = \mathcal{O}(n)$$

(Again we only sum over ks where $\ell_k > 0$ since if $\ell_k = 0$ then we don't do anything to the array.)

Now we have sorted every A_k. Notice that if $k < k'$ then $A_k < A_{k'}$ since the numbers in $A_{k'}$ have longer decimal representations and are thus larger. So we can then iterate over every A_k in order and copy it into a new array (or into A starting from the first index). This array will be sorted since each A_k is sorted.

This will take $\mathcal{O}(n)$ time since it just iterates over every number in A, and there are $\mathcal{O}(n)$ numbers in A. Since this takes $\mathcal{O}(n)$ time and the previous operation (to sort the A_ks) does as well, this algorithm as a whole has a time complexity of $\mathcal{O}(n)$, as required.