# 6.046/18.410 Problem Set 2

**Yijun Jiang**
Collaborator: Hengyun Zhou, Eric Lau

October 13, 2015

## 1 Finding the $k$-th smallest element

### 1.1 Part (a)

**Description**:

We can apply merge sort since both $A$ and $B$ are sorted, and stop once we get the $k$-th element of the merged array. This element is the $k$-th smallest element in the union of $A$ and $B$. More efficiently, we do not need to store the merged array.

**Correctness**:

The correctness of this algorithm is guaranteed by the correctness of merge sort. In merge sort, the $k$-th element being added to the merged array is the $k$-th smallest element in the union of two original arrays.

**Runtime**:

In this algorithm, we need to index into an array $k+1$ times, and make $k$ comparisons between two elements. Since finding an element by its index and comparing two elements are $O(1)$ operations, the runtime of this algorithm is $O(k)$.

### 1.2 Part (b)

**Description**:

We can compare the $\lfloor \frac{k}{2} \rfloor$-th element of both $A$ and $B$. If $A[\lfloor \frac{k}{2} \rfloor] \leqslant B[\lfloor \frac{k}{2} \rfloor]$, then $A[1], \cdots, A[\lfloor \frac{k}{2} \rfloor]$ are guaranteed to be no greater than the $k$-th smallest in the union of $A$ and $B$, which for convenience will be referred to as $x$. These $\lfloor \frac{k}{2} \rfloor$ elements are identified. Otherwise, $B[1], \cdots, B[\lfloor \frac{k}{2} \rfloor]$ are no greater than $x$ and are identified.

**Correctness**:

Suppose $A[\lfloor \frac{k}{2} \rfloor] \leqslant B[\lfloor \frac{k}{2} \rfloor]$. Let $i_A$ and $i_B$ be the largest indices in $A$ and $B$ such that $A[i_A] \leqslant x$ and $B[i_B] \leqslant x$. Notice that $A$ and $B$ are sorted, so the ordering of indices is equivalent to the ordering of elements in either array.

If $i_A < \lfloor \frac{k}{2} \rfloor$, then according to the definition of $i_A$, $x < A[\lfloor \frac{k}{2} \rfloor]$. Therefore, $x < B[\lfloor \frac{k}{2} \rfloor]$ and thus $i_B < \lfloor \frac{k}{2} \rfloor$. As a result, in the union of $A$ and $B$, there are only $i_A + i_B < k$ elements that are no greater than $x$, which contradicts the fact that $x$ is the $k$-th smallest element in the union of $A$ and $B$. Therefore, it must be that $i_A \geqslant \lfloor \frac{k}{2} \rfloor$, which implies that $A[1], \cdots, A[\lfloor \frac{k}{2} \rfloor] \leqslant A[i_A] \leqslant x$. The proof is similar if $A[\lfloor \frac{k}{2} \rfloor] > B[\lfloor \frac{k}{2} \rfloor]$. Then we have $B[1], \cdots, B[\lfloor \frac{k}{2} \rfloor] \leqslant B[i_B] \leqslant x$.

**Runtime**:

To identify $\lfloor \frac{k}{2} \rfloor$ of the $k$ smallest elements, we need to indice once into $A$ and once into $B$, and make one comparison. Altogether this takes $O(1)$ time.

### 1.3 Part (c)

**Description**:

Once we identify half of the elements that are no greather than the $k$-th smallest element $x$, the rank of $x$ in the union of unidentified subarrays is halved. We can recursively cut down this rank by half. The pseudocode below shows the details.

**Algorithm 1** Find the $k$-th minimum of $A \cup B$ in $O(\log k)$ time

```
1: procedure RecursiveFind(A, B, k)
2:     if k=1 then return min(A[1], B[1])
3:     else
4:         if A[⌊k/2⌋] ⩽ B[⌊k/2⌋] then return RecursiveFind(A[⌊k/2⌋ + 1 : end], B, ⌈k/2⌉)
5:         else return RecursiveFind(A, B[⌊k/2⌋ + 1 : end], ⌈k/2⌉)
```

**Correctness**:

Still let $x$ be the element we look for, and let $\text{rank}(x)$ be the rank in the union of unidentified subarrays. What we do is we identify and throw away $\lfloor \frac{\text{rank}(x)}{2} \rfloor$ elements in each iteration. Since these elements are no greater than $x$, it is guaranteed that $\text{rank}(x)$ is halved. The base case is when $\text{rank}(x) = 1$, where taking min of the first entries of both unidentified subarrays give the $x$ we want.

**Runtime**:

The runtime of this algorithm is given by the recursion

$$T(k) = T(k/2) + O(1)$$

whose solution, by the master theorem, is $T(k) = O(\log k)$.

# 2 Structures related to van Emde Boas

All the tries in this problem are considered to be nonempty. Otherwise we should raise an exception for relevant operations.

## 2.1 Part (a)

**Description**:

There are two main cases: (1) $x$ exists (2) $x$ does not exist in the trie. The following algorithm deals with both cases.

We first go down the tree trying to reach $x$. This is done by going deeper into the levels to match the prefix of $x$ (or equivalently, we call Find). If $x$ exists, we reach a leaf. Otherwise, we come to a node where any deeper leaf/node does not match (the prefix of) $x$. We then go up from this leaf/node until, either for the first time we reach a node who has an unexplored branch on the right (which means we come from the left), or this never happens, in which case we raise a no-successor exception. If we find the node, go all the way down its right branch, making sure that we take the left branch whenever possible. The leaf $y$ that we eventually reach is the successor of $x$. The pseudocode below makes it clearer.

**Algorithm 2** Find the successor of $x$ in a trie (whose root is $r$) in $O(\log U)$ time

```
1: procedure Successor(x, r)
2:     px=Find(x, r)
3:                 ▷ If x does not exist, suppose Find returns the deepest node that matches the prefix of x.
4:     Go up from px until either (1) reaching the first node n, where px is in left branch of n, and right
       branch of n is non-empty (2) reaching r, but r does not satisfy (1)
5:     if Case (1) then
6:         y=MinKey(n)                    ▷ Find the minimum key in the sub-trie whose root is n
7:     else
8:         No-successor exception: x is (or is greater than if x ∉ trie) the maximum key in the trie
```

**Correctness**:

The correctness of Successor is guaranteed by the trie structure: the successor of $x$ is always "the leftmost leaf in the closest right subtrie".

**Runtime**:

There are three main steps: reaching $x$, going up, reaching $y$. Each step visits $O(\log U)$ levels, thus $T_{\text{SUCCESSOR}} = O(\log U)$.

## 2.2 Part (b)

INSERT still takes $O(\log U)$ time. Think of the worst case where the root has only one branch and we insert a key into the other (originally empty) branch. Then we have to create $\log U$ nodes (including a leaf). Generally we need to create $O(\log U)$ nodes and a leaf, each of which is $O(1)$ time. So $T_{\text{INSERT}} = O(\log U)$.

FIND takes $O(1)$ time, since we can query $H_{leaf}(x)$ in $O(1)$ (say, by hashing into a leaf).

REMOVE still takes $O(\log U)$ time. We still need to remove the leaf and nodes bottomup, until reaching a node that is nonempty. Therefore, $O(\log U)$ removals must be done, each of which is $O(1)$ time. So $T_{\text{REMOVE}} = O(\log U)$.

## 2.3 Part (c)

**Description**:

We need to bisect the levels (or equivalently, bisect the prefix of $x$) in order to reach $O(\log \log U)$ runtime. The goal is to find a node $n$ whose max is $x$ (or predecessor of $x$ if $x$ is not in the trie), but whose father's max is greater than $x$. Then this father node must have a second (and right) child, whose min gives the $y$ we want. The pseudocode is shown below. Still, it can deal with both cases: $x$ exists and $x$ does not exist in the trie.

**Algorithm 3** Find the successor of $x$ in an augmented trie (whose root is $r$) in $O(\log \log U)$ time

```
1:  procedure SUCCESSOR(x, r)
2:      if x ⩾ r.max then
3:          No-successor exception: x is (or is greater than if x ∉ trie) the maximum key in the trie
4:      else if x < r.min then return r.min
5:                                  ▷ These two edge cases cannot he handled by the pseudocode below
6:      else
7:          n1 ←BISECT(x, 1, log U)
8:                                                  ▷ See detailed pseudocode below for BISECT
9:          n2 = n1.father.rightChild
10:         return n2.min
11:
12: procedure BISECT(x, h_min, h_max)
13:     ▷ Return a leaf/node n such that n.max = x (or predecessor of x if x ∉ trie) but n.father.max > x
14:                             ▷ h is the level in the trie, or the length of prefix of x; note that r is level 0
15:     if h_min = h_max then return HASH(x[1 : h_min])
16:             ▷ Except for the above edge cases, this HASH never returns NULL, see the Correctness part
17:     else
18:         h_mid ← ⌊½(h_min + h_max)⌋
19:         n_mid ← HASH(x[1 : h_mid])
20:         if n_mid = NULL or n_mid.max = x then return BISECT(x, h_min, h_mid)
21:                                          ▷ n_mid = NULL means x does not exist in the trie
22:         else return BISECT(x, h_mid + 1, h_max)
```

**Correctness**:

Suppose $r.min \leqslant x < r.max$, otherwise it is an edge case treated separately in $O(1)$ time. First let us suppose $x$ is in the trie. Then it is guaranteed that there is a node $n$ such that $n.max = x$ but $n.father.max > x$. This is because, there is always a path connecting $r$ to the leaf $x$. Going down this path, max decreases monotonically from $r.max$ to $x$. The highest level where max $= x$ gives the desired $n$. If $x$ does not exist in the trie, in the reasoning above replace $x$ by the predecessor of $x$, and the existence of $n$ can still be proved. Thus the base case of BISECT never returns NULL. Finally, $n$ is not $r$ since $r.max > x$, thus $n$ always has a father.

Instead of searching level-by-level, bisection search is more an efficient way to locate $n$. Bisection works since max is monotonically decreasing down this path.

Any leaf in the subtrie $n$ is no greater than $x$. But since $n.father.max > x$, we know that $n.father$ must have another branch containing a bigger max. So this branch is to the right of $n$. This means every key in this branch is larger than $x$. Moreover, the min in this branch is the smallest key that is still larger than $x$. It must be the successor of $x$. Thus the correctness is proved.

**Runtime**:

SUCCESSOR calls BISECT, which recursively calls itself. The following steps in SUCCESSOR takes constant time. So the entire runtime is limited by BISECT.

The original input size for BISECT is $h = \log U$. The input size is halved in each recursive call. The rest operations, including HASH, takes only $O(1)$ time. Then we have the following recursion

$$T(h) = T(h/2) + O(1)$$

whose solution, by the master theorem, is $T(h) = O(\log h) = O(\log \log U)$. Thus, $T_{\text{SUCCESSOR}} = O(\log \log U)$.