Solutions for Problem Set 2

Problem 1. More divide-and-conquer

Solution

The solution is essentially a binary search. The goal is to find two indices $i_a$ and $i_b$ (the first corresponding to a position in $A$ and the second corresponding to a position in $B$) such that the following conditions are satisfied:

1. $i_a + i_b = n$ or $i_a + i_b = n + 1$
2. the $n^{th}$ (or the $(n + 1)^{th}$, since $A \cup B$ actually has two medians) element of $A \cup B$ is $A[i_a]$ or $B[i_b]$.

Algorithm description

The following algorithm determines the median of two vectors $A, B$ having $n$ elements each. The algorithm is called the first time as $\text{Median-of-Union}(A, B, 1, 1, n, n)$

$\text{Median-of-Union}(A, B, s_a, s_b, \text{end}_a, \text{end}_b)$

1. $i_a = \lfloor \frac{s_a + \text{end}_a}{2} \rfloor$
2. $i_b = \lfloor \frac{s_b + \text{end}_b}{2} \rfloor$
5. else if $A[i_a] > B[i_b]$
6. $\text{Median-of-Union}(A, B, s_a, i_a, i_b, \text{end}_b)$
7. else if $A[i_a] < B[i_b]$
8. $\text{Median-of-Union}(A, B, i_a, \text{end}_a, s_b, i_b)$

Correctness

To prove the correctness of the algorithm we need to show the following things:

- when the condition in step 3 is verified then the output is the correct median;
- when the condition in step 4 is verified then the output is the correct median;
• when we take a recursive step, we restrict to a part of the vector that is guaranteed to contain the median.

Note also that the condition $i_a + i_b = n$ (or $i_a + i_b = n + 1$) is always verified.

To prove the correctness of the stopping conditions, we will just show that when the condition in step 3 is verified, then the output is the correct median (a similar argument shows the correctness of the algorithm in the case the condition in step 4 is verified). If $B[i_b] < A[i_a] < B[i_b + 1]$ this means that in $A \cup B$ there are $(n - 1) \leq (i_a - 1) + i_b \leq n$ elements that are less than $A[i_a]$ and $(n - 1) \leq (n - i_a) + (n - i_b) \leq n$ elements that are larger than $A[i_a]$.

To prove the correctness of the recursive step, consider just the case when the condition in step 5 is verified (the analysis is similar when the condition of step 7 is verified). When $A[i_a] > B[i_b]$ then in $A \cup B$ there are at least $i_a + i_b$ elements that are less than $A[i_a + 1]$, and so the median cannot be $A[i_a + 1]$, nor any element bigger than $A[i_a + 1]$. Hence we can restrict our search to the range $s_a - i_a$ in $A[]$. For similar reasons, we can restrict our search in $B[]$ to the elements in the range $i_b$—$end_b$.

**Analysis**

After $k$ steps, we will be looking at subsets of the vector of size $n/2^k$. We will stop by the time $2^k = n$, so we will not execute more than $\log n$ steps. Each step takes constant time. The total time is $O(\log n)$.

**Problem 2. Intersection**

**Solution**

The basic idea of the algorithm is to scan the two vectors and to output the common elements. Since the two vectors are already sorted it is sufficient to scan the arrays just once in order to find all the common elements.

**Algorithm description**

The following algorithm finds all the common elements of two given arrays of $n$ elements each:

1. $i = j = 1$
2. while $(i \leq n)$ and $(j \leq n)$ do
3. \hspace{1em} If $A[i] = B[j]$ then
4. \hspace{2em} $i++$;
5. \hspace{1em} $j++$;
6. \hspace{1em} output $A[i]$;
7. \hspace{1em} else if $A[i] > B[j]$ then $j++$;
8. \hspace{1em} else $i++$;
9. end while.

Correctness

The correctness of the algorithm is clear by inspection.

Analysis

Consider the value $i+j$. At the beginning it is 2, at the end it is $2n$. Each iteration increases $i+j$ at least by one, so there cannot be more than $2n - 2$ iterations. Each iteration takes $O(1)$ time. The total running time is $O(n)$.

Problem 3. Data structures

Solution

The basic idea is to use a stack (an array $S[1, \ldots, M]$ of elements of the type that have to be stored) an index $p \in \{0, 1, \ldots, M\}$ that points to the topmost element of the stack and a vector $T[1, \ldots, M]$ of integers in the range $\{0, 1, \ldots, M\}$.

The array $T$ has the following meaning: if $T[i] = 0$ then the element having key $i$ is not in the stack, if $T[i] \neq 0$ then $T[i]$ is the location where the element has been stored.

Implementation

Initialization

allocate $T[]$, $S[]$, $p = 0$

Insert (a,i) Insert the element $a$ with key $i$

$p = p + 1$
$S[p] = a$
$T[i] = p$

Find (i)

If $T[i] = 0$ return “Not found”
If $T[i] > p$ return “Not found”
let $a = S[T[i]]$
If key of $a$ different from $i$ return “Not found”
return “found”

This is because the information contained in $T[]$ could be wrong (since $T$ has not been initialized). Thus it is necessary to verify that if $T[i]$ points to a legal position in the stack that position is the one we are interested in.

Delete (i)

If not find(i) then return() else
let $j$ be the key of $S[p]$
swap $S[p]$ with $S[T[i]]$
$T[j] = T[i]$
$p = p - 1$
$T[i] = 0$

In order to avoid overflows in the stack whenever we delete an element the above procedure uses the location of such an element for storing the element that is the topmost of the stack. Thus at any moment $p$ is equal to the number of elements in the stack.

**Correctness**

The following statement is easy to prove by induction on the number of operations that we do starting from an empty data structure: “let $(p, S[], T[])$ be the content of the data structure at a certain point: then the data structure contains exactly $p$ elements, these elements are stored in $S[1], \ldots, S[p]$, and for every $i$ such that there is an element with key $i$ in the data structure, such an element is in $S[T[i]]$.

The above property is maintained by insert() and delete(). find() does not change the data structure, and it works correctly assuming that the above property is true.

**Analysis**

Since there are no loops the complexity of each procedure is $O(1)$