Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department of Computer Science
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger

27th October 2016

Data Structures & Algorithm — Solutions to Sheet 5 — AS 16

Solution 5.1 *Searching for equal numbers.*

a) We compare each element a in A with each element b in B. If we find $a = b$, we output the corresponding element. As we compare each of the m elements in A with each of the n elements of B, we will always use $m \cdot n$ comparisons in this naive approach.

b) A better approach works analogously to merging two arrays with Mergesort. We consider the elements in each array in ascending order. We store two numbers i and j that indicate the currently considered element in A, respectively in B (initially $i = j = 1$). In every step, we compare $A[i]$ with $B[j]$ and distinguish three cases:

Case 1 ($A[i] = B[j]$): We have found two elements that are both in A and in B. We output the element and increase i and j by 1 (as each element occurs at most once in A and in B, $A[i]$ cannot be equal with any $B[j']$ besides $B[j]$. Analogously, no $B[j]$ can be equal with an $A[i']$ besides $A[i]$).

Case 2 ($A[i] < B[j]$): We know that the elements in B are sorted in ascending order. Therefore, $A[i]$ cannot be equal to any element $B[j']$ for $j' \geq j$. We can continue with the next element from A and increase i by 1.

Case 3 ($A[i] > B[j]$): As the elements in A are sorted in ascending order, $B[j]$ cannot be equal to any $A[i']$ for $i' \geq i$. Therefore, we can continue with the next element of B and increase j by 1.

This algorithm terminates as soon as $i = m + 1$ or $j = n + 1$ is reached. Either i or j (or even both) are increased by 1 in every step. Hence, the algorithm terminates after less than $n + m$ steps. As comparisons and calculations of one step take only constant time, the running time of the algorithm is in $\mathcal{O}(m + n)$.

Lösung 5.2 *Extended Heaps.*

The function Restore-Heap-Condition was presented in lecture. It exchanges the node containing a key k with the smaller of its successors until *both* successors contain a key greater or equal than k, or k does not have any successors any more. Analogously we define a function Bottom-Up that exchanges a node of a key k with its predecessor until the predecessor contains a key smaller or equal than k, or k is stored in the root of the heap. Let A be the array storing the heap, and i be the initial position of k in A. In pseudocode we get the following function:

```
BOTTOM-UP($A$, $i$)
1 while $i \geq 2$ do                              ▷ $A[i]$ has a predecessor
2     $j \leftarrow \lfloor i/2 \rfloor$                              ▷ $A[j]$ is predecessor
3     if $A[j] \leq A[i]$ then STOP                     ▷ Heap-Property satisfied: Stop
4     else Swap $A[i]$ and $A[j]$; $i \leftarrow j$
```

Additionally we maintain a variable n that indicates the number of keys stored in the heap, and that is initialized with 0. Since in the above algorithm we have $i \in \{1, \ldots, n\}$, its running time is bounded by $\mathcal{O}(\log n)$.

a) MIN: A is a Min-Heap. Thus, MIN simply returns $A[1]$ (or an error message if $n = 0$). The running time is constant, i.e. in $\mathcal{O}(1)$.

b) REPLACE(i, k): Let $k' = A[i]$ be the key to be replaced. We distinguish two cases:

   1. Case: $k < k'$. Then the subtree rooted at $A[i]$ is still a (Min-)Heap after the substitution (the smallest element just got smaller). However, $A[1..n]$ might not be a heap any more because k might be smaller than its predecessor. To restore the heap property, we invoke BOTTOM-UP(A, i).

   2. Case: $k > k'$. Then, the key stored in node $A[i]$ is still larger than its predecessor after the substitution, but the subtree rooted at $A[i]$ might not be a heap any more (k might be larger than its successors). To restore the heap property, we invoke RESTORE-HEAP-CONDITION(A, i).

   Therefore the operation REPLACE(i, k) can be implemented to run in time $\mathcal{O}(\log n)$.

c) INSERT(k): We increase n by 1 and store the new key k in the entry $A[n]$. Now it might happen that A is not a Min-Heap any more, because the new key might be smaller than its predecessor. To restore the heap property it suffices to invoke BOTTOM-UP(A, n). The running time is in $\mathcal{O}(\log n)$.

d) DELETE(i): We invoke REPLACE$(i, A[n])$, i.e., the key stored at position i is replaced by $A[n]$. Now the key stored in $A[n]$ occurs once too often, thus we decrease n by 1 (the size of the heap shrinks by 1 and the redundant elements gets cut off). The operation has the same running time than REPLACE(i, k), thus it is in $\mathcal{O}(\log n)$.

Lösung 5.3 *Searching in sorted arrays.*

a) The interpolation search is much faster, if the numbers in an array grow as uniformly as possible. An example is the array $A = (1, 2, \ldots, n)$. The key i is stored on position i (the array is indexed from 1 to n). Initially, an interpolation search for key b sets

$$\text{middle} = \left\lfloor \text{left} + \frac{b - A[\text{left}]}{A[\text{right}] - A[\text{left}]} \cdot (\text{right} - \text{left}) \right\rfloor = \left\lfloor 1 + \frac{b-1}{n-1}(n-1) \right\rfloor = b$$

and thus finds b after exactly one comparison. Binary search would bisect the search range multiple times for many choices of b until b is found.

Interpolation search is substantially slower than binary search if the keys do not grow linearly at all. Consider for example the array $A = (1!, 2!, \ldots, n!)$. We show that an interpolation search for key $b = (n-1)!$ considers at least $\lfloor n/2 \rfloor$ many keys. To prove this statement, we show that in every step the left border 'left' is increased by at most 2 and the right border remains n.

For simplicity, we write l instead of left. The interpolation search calculates

$$\begin{aligned} \text{middle} &= \left\lfloor \text{left} + \frac{b - A[\text{left}]}{A[\text{right}] - A[\text{left}]} \cdot (\text{right} - \text{left}) \right\rfloor = \left\lfloor l + \frac{(n-1)! - l!}{n! - l!} \cdot (n-l) \right\rfloor \\ &\leq \left\lfloor l + \frac{(n-1)!}{n! - \frac{1}{2}n!} \cdot (n-l) \right\rfloor = \left\lfloor l + \frac{2}{n} \cdot (n-l) \right\rfloor = \left\lfloor l + 2 - \frac{2l}{n} \right\rfloor \in \{l, l+1\} \end{aligned}$$

and considers $A[\text{middle}]$ (i.e. $A[l]$ or $A[l+1]$). Besides middle $= n-1$, we note that $A[\text{middle}] < (n-1)!$ and thus the procedure sets left $\leftarrow$ middle $+ 1$. The value of left is therefore increased by at most two (and right remains n). Binary search would consider only $\mathcal{O}(\log n)$ many keys.

b) The main idea is to combine interpolation search and binary search in a smart way. We compute alternately a pivot element using interpolation search and using binary search. For the pathological cases, in which the interpolation search takes linear time, the search range is still bisected in every second step by the binary search.

Therefore, this algorithm needs at most twice as many iterations as a simple binary search and also at most twice as many iterations as interpolation search. In particular, the modified algorithm needs at most twice as many steps as the minimal number of steps of binary search and interpolation search.