Solution 5.1  Joining AVL Trees.

Let $T_1$ and $T_2$ be two AVL trees of heights $h_1$ and $h_2$. For a tree $T$, we denote its height by $h(T)$. Let $T$ be a tree with a root $v$, a left subtree $T_l(v)$, and a right subtree $T_r(v)$. Then we define the balance factor of $v$ as $\text{bal}(v) := h(T_r(v)) - h(T_l(v))$. For an AVL tree we have $\text{bal}(v) \in \{-1, 0, 1\}$ for every vertex $v$ of the tree. Especially we have $\text{bal}(v) = -1$ if the left subtree is higher than the right one.

Now the union of two AVL trees can be computed as follows:

1) We first calculate the values of $h_1$ and $h_2$ by calculating the length of a longest path from the root to a leaf in $T_1$ and $T_2$. To do this efficiently, we use the balance factors in the vertices. We start at the root, and when located at a vertex $v$, we choose as a successor the vertex at which the higher subtree of $v$ is rooted: if $\text{bal}(v) = -1$, we proceed to the left child, otherwise to the right one (for $\text{bal}(v) = 0$, both successors can be used). The values of $h_1$ and $h_2$ can be determined in time $O(h_1 + h_2)$.

2) Remove the minimum element $x$ of $T_2$. After the removal, we obtain a tree $T'_2$ with height $h \in \{h_2 - 1, h_2\}$. This operation can be performed in time $O(h_2)$.

3) If $|h_1 - h| \leq 1$ (the height difference of $T_1$ and $T'_2$ is at most 1), then the algorithm terminates and outputs a new tree rooted at $x$, with $T_1$ as left subtree and $T'_2$ as right subtree. This operation needs only $O(1)$ time.

4) Without loss of generality, let $h_1 > h + 1$ (the case $h_1 < h - 1$ is symmetric). We start at the root of $T_1$ and follow the right child vertices until we find a vertex $v$ that is the root of a subtree $T'_1$ with height $h$ or $h + 1$. We also determine the predecessor $u$ of this vertex $v$ (i.e., $T'_1$ is the right subtree of $u$, and $v$ is the root of $T'_1$). We can find the vertices $u$ and $v$ in time $O(h_1)$ as follows.

1  $v \leftarrow \text{Root}(T_1)$  
2  $h' \leftarrow h_1$  
3  while $h' > h + 1$ do  
4    $u \leftarrow v$  
5    if $\text{bal}(v) = -1$ then $h' \leftarrow h' - 2$ else $h' \leftarrow h' - 1$  
6  $v \leftarrow \text{RightChild}(v)$

5) Next, replace the right subtree of $u$ by the tree that has $x$ as root, $T'_1$ as left subtree and $T'_2$ as right subtree.

This tree with $x$ as the root is a search tree, since all keys in $T_1$ (and thus in particular in $T'_1$) are smaller than $x$, and all keys in $T'_2$ are greater or equal to $x$. Moreover, it is an AVL tree, because the balance factor of $x$ is $h - h' \in \{-1, 0\}$. Analogically, one can argue that the whole tree that we constructed is still a search tree. However, the AVL property might be violated since the new right subtree of $u$ now has height $h' + 1$ (instead of height $h$ that it had originally).
We now have the following schematic situation:

The whole operation can be performed in time $O(1)$.

6) Analogically with a situation of inserting a new vertex, we need to check the balancing factors of the vertices on the path from $u$ to the root of the tree, and, if necessary, repair them using rotations. This can be done in time $O(h_1)$.

In overall, the steps 1 – 6 can be performed in time $O(h_1 + h_2)$. Since $h_1, h_2 \in O(\log n)$, the whole union can be realized in time $O(\log n)$.

**Solution 5.2 Amortized Analysis.**

A good choice is $k = 2n$. This means that, once the array is full and we need to insert a new element, a new array of double the size is created. To show that this choice leads to amortized constant costs for the insert operations, we perform an amortized analysis. We define a potential function that assigns a value to every state of the array (we can intuitively think of this value as a “cash balance”).

As a reminder, amortized analysis using potential functions works as follows: We define $\Phi_i$ as the potential for the $i$-th operation. Let the actual cost of the $i$-th operation be $t_i$. The amortized cost of the $i$-th operation is defined as $a_i := t_i + \Phi_i - \Phi_{i-1}$. With this definition, it follows for a sequence of $m$ operations that

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{m} t_i\right) + \Phi_m - \Phi_0,$$

thus we obtain

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m.$$  \hspace{1cm} (1)

Once we have an estimate of the amortized cost for each operation as well as an estimate for $\Phi_0 - \Phi_m$, we also have an estimate of the actual total costs. If the potential function is chosen such that $\Phi_m \geq \Phi_0$ for every $m$, then it follows that $\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i$, i.e., the sum of the amortized costs is an upper bound for the actual total cost.
a) We define the potential function (i.e., the cash balance) of an array of size $n$ as

$$6 \cdot \text{number of elements in the second half of the array (in positions } \frac{n}{2} + 1, \ldots, n).$$

(3)

Note that $n$ changes when the array is resized. From the definition it follows that $\Phi_0 = 0$ (initially the array is empty), and because $\Phi_i$ can never be negative, it is also clear that $\Phi_i \geq 0$ for every $i > 0$, thus, $\Phi_m \geq \Phi_0$. We need to examine how much an insertion costs. We distinguish two cases: If in the $i$-th operation the array size is not doubled (i.e., the array is not full), then $t_i = 1$, $\Phi_i - \Phi_{i-1} \leq 6$ (= 0 if the second half is empty, and = 6 otherwise), and $a_i \leq 1 + 6 = 7$. If the array size is doubled to $2n$ in the $i$-th insert operation, the actual costs are

$$t_i = \frac{2n}{n} + \frac{n}{n} + \frac{1}{1} = 3n + 1$$

(4)

and the potential difference is

$$\Phi_i - \Phi_{i-1} = 6 \cdot (1 - \frac{n}{2}) = 6 - 3n.$$

(5)

In this case, the amortized costs are $a_i = 3n + 7 - 3n = 7$, thus they are also constant.

b) We show that also for a sequence of only deletions, amortized constant time is possible. To obtain this, we shrink an array of size $n$ to size $n/2$ when only $n/4$ elements are left, and not already when $n/2$ elements are left. This prevents us from repeatedly doubling and halving the array by first inserting $n/2$ elements and then starting to alternate between insertions and removals.

For the amortized analysis, we define the potential function (i.e., the cash balance) of an array of size $n$ as

$$3 \cdot \text{number of empty positions in the first half of the array (in positions } 1, \ldots, \frac{n}{2}).$$

(6)

If the array is not halved in the $i$-th delete operation, then we have $a_i = 1 + 0$ if the deleted element is in the upper half of the array and $a_i = 1 + 3$ if the deleted element is in the lower half. If the array is halved in the $i$-th delete operation, then we have

$$t_i = \frac{n/2}{n/2} + \frac{n/4}{n/4} = \frac{3}{4} n,$$

(7)

and the potential difference is

$$\Phi_i - \Phi_{i-1} = 3 \cdot (1 - n/4).$$

(8)

The amortized cost in this case is $a_i = \frac{3}{4} n + 3 \cdot (1 - n/4) = 3$. For every deletion, the amortized cost is constant (precisely, $a_i \leq 4$).

It is easy to see that $\Phi_0 - \Phi_m \leq m$. For the actual costs we thus obtain

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m \leq 4m + m \in \mathcal{O}(m).$$

(9)

This completes the amortized analysis for the deletion.
It is now easy to see that the potential function

\[6 \cdot \text{(number of elements in the second half of the array + number of empty positions in the first half of the array)}\]

\[\text{(10)}\]

can be used to show that for arbitrary sequences of insert and delete operations, the amortized cost of each operation is constant.

Note: We could also include additional costs in the analysis, e.g., assuming that the deletion of an array of length \(n\) costs \(\Theta(n)\) (and not 0).

Solution 5.3 Blum’s Median-of-Median Strategy.

a) We divide the first sequence in \(\lceil N/5 \rceil\) groups of exactly 5 elements, and one group with 2 elements:

\[8, 13, 17, 5, 11, 29, 3, 4, 11, 10, 15, 7, 30, 57, 1, 2, 6, 9, 17, 7, 14, 13\]

The first recursive call is invoked on the medians of these groups. For the last group, the median is 14 by definition. Therefore the first sequence is

\[11, 10, 15, 7, 14\]

The result of this call is the median-of-medians 11. We use 11 as pivot element for a pivoting step similarly to the one in Quicksort (we first interchange 11 with 13, do the pivoting step and finally interchange 15 with 11):

\[8, 7, 9, 5, 6, 2, 3, 4, 1, 10, 7, 11, 30, 57, 11, 29, 13, 17, 17, 13, 14, 15\]

The first sequence has more elements than the second one has. Since we look for the element on the position \(\lceil N/2 \rceil\), the second recursive call of \textit{Auswahl} is invoked on the larger sequence

\[8, 7, 9, 5, 6, 2, 3, 4, 1, 10, 7\]

Note: Depending on the implementation of the pivoting step, the elements of the sequence could be in a different order.

b) The first recursive call of the procedure \textit{Auswahl} is always invoked on exactly \(\lceil N/5 \rceil\) elements. In the best case, the median-of-medians is exactly the median we are looking for, then the second call is not needed at all. If the median-of-medians is exactly the median that we are looking for, then the second call is not needed at all. If the median-of-medians is exactly “next” to the pivot element, the second recursive call is invoked on \(\lfloor N/2 \rfloor\) elements. In the worst case, there are at least \(3(\lfloor N/5 \rfloor - 2) + 2 + 1\) elements smaller (or greater) than the median-of-medians (there exist \((\lfloor N/5 \rfloor - 2)\) groups of 5 that contribute 3 each, the group of the pivot element contributes 2, the group with fewer than 5 elements may only contribute 1). The number of elements in the second recursive call is then

\[N - 3 \left( \left\lceil \frac{1}{2} \left\lceil \frac{N}{5} \right\rceil \right\rceil - 1 \right) \approx \frac{7}{10} N + 3.\]  \(\text{(11)}\)