Exercise 6.1  *Self-organizing Lists.*

a) Using the frequency-count rule, the given sequence of accesses results in the following shifts (each with the corresponding access time):

<table>
<thead>
<tr>
<th>K</th>
<th>L</th>
<th>A</th>
<th>N</th>
<th>G</th>
<th>O</th>
<th>P</th>
<th>F</th>
<th>E</th>
<th>R</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>K</td>
<td>L</td>
<td>A</td>
<td>N</td>
<td>G</td>
<td>O</td>
<td>P</td>
<td>F</td>
<td>E</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>K</td>
<td>L</td>
<td>A</td>
<td>N</td>
<td>G</td>
<td>O</td>
<td>P</td>
<td>F</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>E</td>
<td>G</td>
<td>K</td>
<td>L</td>
<td>A</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>R</td>
<td>G</td>
<td>K</td>
<td>L</td>
<td>A</td>
<td>N</td>
<td>O</td>
<td>P</td>
<td>F</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>R</td>
<td>G</td>
<td>N</td>
<td>K</td>
<td>L</td>
<td>A</td>
<td>O</td>
<td>P</td>
<td>F</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>R</td>
<td>G</td>
<td>N</td>
<td>K</td>
<td>L</td>
<td>O</td>
<td>A</td>
<td>P</td>
<td>F</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>R</td>
<td>G</td>
<td>N</td>
<td>K</td>
<td>L</td>
<td>O</td>
<td>P</td>
<td>F</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

In total, 79 steps are required, with an average of \( \frac{79}{12} \approx 6.58 \) steps per access.

Using the transpose rule, the sequence of accesses results in the following shifts:
In total, 77 steps are required, with an average of $77/12 \approx 6.42$ steps per access. The frequency-count rule is a bit worse than the Transpose rule in this example.

b) A bad sequence for the transpose rule on the given list is the one repeating ,,R,E`` $n$ times. Every pair requires 20 steps, and $20n$ steps are required in total (10 per request). Using the move-to-front rule on the same sequence yields 20 steps for the first pair, and only 4 steps for the following pairs. For $n \to \infty$, the transpose rule requires 5 times the number of steps of the move-to-front rule. In general, transpose is bad if the elements at the end of the list are accessed.

The move-to-front rule behaves poorly when a rarely used element is accessed. This is, for example, the case when repeating ,,R, E, F, P, O, G, N, A, L, K`` $n$ times. Move-to-front requires 10 steps for every element in the sequence, i.e., an overall of $100n$ steps. The transpose rule requires 60 steps for each sequence, and since afterwards the list is in its original state, the overall number of steps is $60n$. In this case, move-to-front requires 1.66 times as many steps as the transpose rule.

The frequency-count rule is bad when an element that was accessed rarely in the past is now accessed often. It takes a long time for the element to get to the front of the list. This is for example the case for the sequence that accesses $n$ times the element ,,K``, then $n$ times ,,L``, then $n$ times ,,A``, and so on. An element is moved to the front of the list only after its $n$-th access, and the overall number of steps is $n \sum_{i=1}^{10} i = 55n$. Using the move-to-front rule, each element is always moved to the front, and the overall number of steps is $\sum_{i=1}^{10} (i + 1 \cdot (n - 1)) = 10n + 45$. For $n \to \infty$, the frequency-count rule requires (asymptotically) 5.5 times as many steps.

Note: Both the frequency-count and the transpose rule can in general be much worse than another strategy, i.e., they are not $k$-competitive for any constant $k$: Let $m$ be the number of accesses, and $n$ the number of elements. If the last two elements are accessed alternatingly, the transpose rule requires $\Omega(mn)$ steps, and move-to-front requires $O(m)$ steps. If one of each element in the original order is accessed $k + n$ times, Frequency-count costs $\Omega(kn^2) = \Omega(mn)$ steps, but the optimal would be $O(m)$. Move-to-front never requires more than twice as many steps as any other solution, i.e., it is 2-competitive.

Exercise 6.2  
Splay Trees & Optimal Search Trees.

a) We obtain the given tree by inserting the keys 1, 2, 3, 7, 6, 5, 4 in this order.

b) An example is $a_1, \ldots, a_7 = 1, 3, 11, 43, 11, 3, 1$ and $b_0, \ldots, b_7 = 0, 0, 1, 5, 5, 1, 0, 0$ with keys 1, $\ldots$, 7 (the access frequencies are given next to the nodes/leaves):
This example is obtained by defining the weights in bottom-up fashion. In every sub-tree the root has a frequency that is higher than the total frequencies of its two sub-trees, and the two sub-trees have same frequency (these conditions are slightly stronger than necessary).

Exercise 6.3  Design of Optimal Search Trees.

The tables \( r(i, j) \), \( P(i, j) \) and \( W(i, j) \) contain the root, the number of comparisons, and the number of accesses of an optimal search tree, as described in Chapter 5.7 of the book.

\[
\begin{array}{cccccc}
  i/j & 0 & 1 & 2 & 3 & 4 & 5 \\
  \hline \\
  0 & 5 & 6 & 10 & 19 & 22 & 33 \\
  1 & 0 & 4 & 13 & 16 & 27 \\
  2 & 0 & 9 & 12 & 23 \\
  3 & 2 & 5 & 16 \\
  4 & 1 & 12 \\
  5 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccc}
  i/j & 0 & 1 & 2 & 3 & 4 & 5 \\
  \hline \\
  0 & 0 & 6 & 14 & 33 & 41 & 68 \\
  1 & 0 & 4 & 17 & 25 & 52 \\
  2 & 2 & 0 & 9 & 17 & 40 \\
  3 & 3 & 0 & 5 & 21 \\
  4 & 4 & 0 & 12 \\
  5 & 5 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccc}
  i/j & 0 & 1 & 2 & 3 & 4 & 5 \\
  \hline \\
  0 & 1 & 1 & 3 & 3 & 3 \\
  1 & 2 & 3 & 3 & 3 \\
  2 & 3 & 3 & 5 \\
  3 & 3 & 4 & 5 \\
  4 & 4 & 5 \\
\end{array}
\]

The corresponding tree is (the access frequencies are given next to the nodes/leaves):
Exercise 6.4  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 a amortized constant cost 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 an „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. The actual cost of the \( i \)-th operation is \( t_i \). The amortized cost of the \( i \)-th operation is defined as \( a_i := t_i + \Phi_i - \Phi_{i-1} \). From this definition it follows that, for a sequence of \( m \) operations:

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

and therefore

\[
\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m.
\]

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 \( m \), for instance, it follows that \( \sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i \), i.e., we can estimate the actual total cost by the sum of the amortized costs.

a) Insertion in amortized constant time:

We define the potential function (i.e., the cash balance) of an array of size \( n \) as

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

Note that \( n \) changes when the array is resized. From the definition it follows that \( \Phi_0 = 0 \) (at the beginning the array is empty), and because \( \Phi_i \) can never be negative, it is also clear that for every \( i > 0 \) we have \( \Phi_i \geq 0 \).
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 is not doubled (i.e., it 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 is doubled to size $2n$ in the $i$-th insert operation, the actual costs are

$$t_i = \frac{2n}{\text{resizing}} + \frac{n}{\text{copying}} + \frac{1}{\text{inserting new element}} = 3n + 1$$

and the potential difference is

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

The amortized cost in this case is $a_i = 3n + 7 - 3n = 7$.

The amortized cost of an insertion is therefore constant (precisely: $a_i \leq 7$). This completes the amortized analysis for the insertion.

b) Deleting in amortized constant time:

We show that amortized constant time is possible for deletions only. To obtain this, we shrink an array of size $n$ to size $n/2$ when it has only $n/4$ elements left, and not when it has $n/2$ elements 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

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

If in delete operation $i$ the array is not halved, we have that $a_i = 1 + 0$ 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 we half the array in delete operation $i$, we have

$$t_i = \frac{1}{\text{delete the element}} + \frac{n/2}{\text{create new array}} + \frac{n/4 - 1}{\text{copy}} = \frac{3}{4}n$$

(we could avoid the explicit deletion) and the difference in potential function is

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

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

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

This completes the amortized analysis for the deletion.

It is now easy to see that the potential function

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

can be used to show that for every sequence of both insert and delete operations, in any order, 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).