Solution 5.1  Open Hashing.

In the following, the probing methods are used exactly as they were presented in the lecture and in the book, i.e., the probing function is subtracted from the hash function. An addition would also be valid, but it might lead to different solutions.

a)  
- $h(k) = \text{Digit sum of } k$. This function is not suitable for hashing. The value of the hash function must lie between 0 and $p - 1$, but the digit sum can be arbitrarily large.
- $h(k) = k(1 + p + p^2) \mod p$. Since $p \mod p = 0$, and $p^2 \mod p = 0$, we have $h(k) = k \mod p$. In the lecture it was explained that $h(k) = k \mod p$ is a suitable hashing function.
- $h(k) = \lfloor p(rk - \lfloor rk \rfloor) \rfloor$, $r \in \mathbb{R}^+\setminus\mathbb{Q}$. This function is also suitable for hashing, because it corresponds to the multiplicative method presented in the lecture.

b)  
(i)  
- $h(12) = 1$
- $h(19) = 8$
- $h(6) = 6$
- $h(15) = 4$
- $h(13) = 2$
- $h(2) = 2 \rightarrow h(2) - 1 = 1 \equiv 1 \rightarrow h(2) - 2 = 0$
- $h(28) = 6 \rightarrow h(28) - 1 = 5$
- $h(24) = 2 \rightarrow h(24) - 1 = 1 \equiv 1 \rightarrow h(24) - 2 = 0 \rightarrow h(24) - 3 = 10$

(ii)  
- $h(12) = 1$
19: $h(19) = 8$

6: $h(6) = 6$

15: $h(15) = 4$

13: $h(13) = 2$

2: $h(2) = 2 \rightarrow h(2) - 1 \equiv 1 \rightarrow h(2) + 1 \equiv 3$

28: $h(28) = 6 \rightarrow h(28) - 1 \equiv 5$

24: $h(24) = 2 \rightarrow h(24) - 1 \equiv 1 \rightarrow h(24) + 1 \equiv 3$

\[ \rightarrow h(24) - 4 \equiv 9 \]

(iii) 12: $h(12) = 1$

19: $h(19) = 8$

6: $h(6) = 6$

15: $h(15) = 4$

13: $h(13) = 2$

2: $h(2) = 2 \rightarrow h(2) - h'(2) \equiv 10$

28: $h(28) = 6 \rightarrow h(28) - h'(28) \equiv 4 \rightarrow h(28) - 2h'(28) \equiv 2$

\[ \rightarrow h(28) - 3h'(28) \equiv 0 \]

24: $h(24) = 2 \rightarrow h(24) - h'(24) \equiv 6 \rightarrow h(24) - 2h'(24) \equiv 10$

\[ \rightarrow h(24) - 4h'(24) \equiv 3 \]

c) The deletion of a key $k$ is problematic if another key $k'$ with $h(k) = h(k')$ is inserted later than $k$. If $k$ would simply be removed (e.g., by marking the position as free), the key $k'$ could no longer be found, because the probing stops once an empty position is found. In the example above the keys 2 and 24 could no longer be found. Therefore we must mark the position explicitly as deleted, and when searching for a key the probing must continue when such a position is found. Of course, when inserting a new key such a marking may be overwritten if necessary.
If many keys are deleted, then it may happen that the search for a key gets very inefficient (because the probing may visit many positions that are marked as deleted). Therefore hashing is suitable especially if keys are mostly inserted and searched and only rarely deleted.

d)  
- \( h'(k) = \lceil \ln(k + 1) \rceil \mod q \). This function is not suitable as a second hash function, because for the key \( k = 0 \) we have \( h'(0) = \lceil \ln(1) \rceil = 0 \).
- \( s(j, k) = k^j \mod p \). This function is not suitable as a probing function, because for the keys \( k = 0 \) and \( k = 1 \), the function \( s(j, k) \) has constant value of 0 and 1.
- \( s(j, k) = ((k \cdot j) \mod q) + 1 \). This function is also not suitable as a probing function because its value is constant 1 if the key \( k \) is a multiple of \( q \). Moreover, for all other keys, the image of \( s(j, k) \) is \( \{1, \ldots, q\} \), i.e., \( p - q \) addresses of the hash table can not be reached.

e) Quadratic probing generates a sequence of \( s(j, k) \) that does not depend on the key \( k \). If there exist many keys \( k \) that are mapped on the same hash address \( h(k) \), then the probing sequence is the same for all of these keys. Thus there are many collisions when probing. This phenomenon is called secondary clustering. Double hashing avoids secondary clustering because different keys \( k, k' \) with \( h(k) = h(k') \) often have different probing sequences.

Solution 5.2  Binary Search Trees.

a) We obtain the following tree:

b) We replace the key 12 by the key of the symmetric predecessor (i.e., by the largest key smaller than 12), or by the key of the symmetric successor (i.e., by the smallest key greater than 12). After that we remove the corresponding predecessor or successor.

Using the symmetric predecessor:

Using the symmetric successor:

c)  
- Preorder: 12, 6, 2, 19, 15, 13, 28, 24
- Postorder: 2, 6, 13, 15, 24, 28, 19, 12
- Inorder: 2, 6, 12, 13, 15, 19, 24, 28
d) A binary search tree can be reconstructed from its postorder sequence \(k_1, \ldots, k_n\) by inserting the keys \(k_n, \ldots, k_1\) in this order into an initially empty binary search tree. Using this approach we obtain the following search tree:

![Binary Search Tree Diagram]

Solution 5.3 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

\[
\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m. \tag{2}
\]

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

\[
6 \cdot \text{number of elements in the second half of the array (in positions } \frac{n}{2} + 1, \ldots, n). \tag{3}
\]

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

\[ t_i = \frac{2n}{\text{Create new array}} + \frac{n}{\text{Copy elements}} + \frac{1}{\text{Insert new element}} = 3n + 1 \]  

(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

\[ \Phi_i = 3 \cdot \text{number of empty positions in the first half of the array (in positions 1, \ldots, n/2)}. \]  

(6)

Again we have \( \Phi_i \geq 0 \) for \( i \geq 0 \). Obviously every element that we want to delete has been inserted before. It is easy to see that after every insertion operation the array does not contain any empty positions in the first half of the array if we start with an array of size 1 or 2. Thus we have \( \Phi_0 = 0 \).

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

\[ t_i = \frac{n/2}{\text{Create new array}} + \frac{n/4}{\text{Copy elements}} = \frac{3}{4}n, \]  

(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 now easy to see that the potential function

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

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).