Data Structures and Algorithms in Java™

Sixth Edition

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Michael H. Goldwasser
Department of Mathematics and Computer Science
Saint Louis University

Instructor's Solutions Manual

WILEY

Chapter 9 Maps, Hash Tables, and Skip Lists

Hints and Solutions

Reinforcement

R-9.1) Hint The first insertion is $O(1)$, the second is $O(2)$, ...

R-9.2) Hint Use the PositionalList.remove method to delete an entry from the map.

R-9.3) Hint Take advantage of the existing findIndex method.

R-9.3) Solution

```
public boolean containsKey(K key) {
  return (findIndex(key) != −1);
}
```

R-9.4) Hint Think about which of the schemes use the array supporting the hash table exclusively and which of the schemes use additional storage external to the hash table.

R-9.5) Hint There is a lot of symmetry and repetition in this string, so avoid a hash code that would not deal with this.

R-9.5) Solution Either polynomial hash codes or cyclic shift hash codes would be good.

R-9.6) Hint Try to mimic the figure in the book.

R-9.6) Solution

0	1	2	3	4	5	6	7	8	9	10
13	94 39				44 88 11			12 23	16 5	20

R-9.7) Hint Try to mimic the figure in the book.

R-9.7) Solution

0	1	2	3	4	5	6	7	8	9	10
13	94	39	16	5	44	88	11	12	23	20

R-9.8) Hint The failure occurs because no empty slot is found. For the drawing, try to mimic the figure in the book.

R-9.8) Solution

0	1	2	3	4	5	6	7	8	9	10
13	94	39	11		44	88	16	12	23	30

The probe sequence when inserting value 5 fails to ever find the sole remaining empty slot (at index 4).

R-9.9) Hint Try to mimic the figure in the book.

R-9.9) Solution

0	1	2	3	4	5	6	7	8	9	10
13	94	23	88	39	44	11	5	12	16	20

R-9.10) Hint Think of the worst-case time for inserting every entry in the same cell in the hash table.

R-9.10) Solution The worst-case time is $O(n^2)$. The best case is $O(n)$.

R-9.11) Hint Mimic the way the figure is drawn.

R-9.11) Solution

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		12	18	41	30	36	25		54			38	10			28		

R-9.12) Hint Combine the hash values of the two components to make a hash value for the pair.

R-9.12) Solution

```
public int hashCode( ) {
  return (a.hashCode( ) ^ b.hashCode( ));
}

public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass( ) != o.getClass( )) return false;
  Pair other = (Pair) o;
  return (a.equals(other.a) && b.equals(other.b));
}
```

R-9.13) Hint There is a subtle flaw—but can you find it?

R-9.13) Solution The problem occurs if null is allowed as a value within the map. In that case, there is ambiguity when bucket.remove(k) returns null, as that may indicate that nothing was removed, or that a key was removed having null as its value. Therefore, we might need to decrease n, even when null is returned.

R-9.14) Hint The load factor can be controlled from within the abstract class, but there must be means for setting the parameter (either through the constructor, or a new method).

R-9.15) Hint It is okay to insert a new entry on "top" of the deactivated entry object.

R-9.16) Hint You will need to keep track of the number of probes in order to apply quadratic probing.

R-9.17) Hint Think of where the entry with minimum key is stored.

R-9.18) Hint Since the map will still contain n entries at the end, you can assume that each remove( ) operation takes the same asymptotic time.

R-9.19) Hint Take advantage of the existing findIndex method.

R-9.19) Solution

```
public boolean containsKey(K key) {
  int j = findIndex(key);
  return (j < table.size( ) && compare(key, table.get(j)) == 0);
}
```

R-9.20) Hint The crucial methods are get and put.

R-9.21) Hint In the new code, what happens when key equals table.get(mid)?

R-9.21) Solution This version is certainly wrong. This can be seen by considering for example a call of findIndex(20, 0, 2) for a table with contents {10, 20, 30}. Such a call should return index 1, yet the newly proposed code will return index 3.

It is worth noting that the alternate version would produce the correct result if only line 4 had been written as

```
if (compare(key, table.get(mid)) <= 0)
```

thereby grouping the equal case with the if clause (not the else clause).

R-9.22) Hint Assume that a skip list is used to implement the sorted map.

R-9.22) Solution For this problem, we assume we are using a skip list as the implementation for the sorted map. (If a SortedTableMap were used, with lower price/performance pairs, each pair is inserted at the beginning of the array list, and thus the insertion time dominates the search time, for overall $O(n^2)$ time.)

With a skip list, the expected running time for maintaining a maxima set if we insert n pairs such that each pair has lower price and performance than one before it is $O(n \log n)$, since each pair is inserted and does not remove any others. Thus, the i^{th} insertion runs in $O(\log i)$ time, and the total expected running time is $\sum_{i=1}^{n} \log i$, which is $O(n \log n)$.

If, on the other hand, each pair had a lower price and higher performance than the one before it, then the overall running time is $O(n)$, because each new pair causes the removal of the pair that came before it. Therefore the size of the map is constant, and thus all its operations run in expected $O(1)$ time.

R-9.23) Hint Mimic the style of the figures in the book.

R-9.24) Hint You must link out the removed entry's tower from all the lists it belongs to.

R-9.25) Hint Compare to the implementation of the addAll method given in Section 10.5.1.

R-9.25) Solution Loop through the second set, while removing each of its elements from the first set (if it exists).

R-9.26) Hint Compare to the implementation of the addAll method given in Section 10.5.1.

R-9.26) Solution Loop through the first set, and remove each element that cannot be found in the second set.

R-9.27) Hint Recall that most skip-list operations run in $O(\log k)$ time for a set with size k.

R-9.27) Solution $O(m \log(n+m))$ time.

R-9.28) Hint Recall that most hash-table operations run in $O(1)$ expected time.

R-9.28) Solution $O(m)$

R-9.29) Hint Recall that most hash-table operations run in $O(1)$ expected time.

R-9.29) Solution $O(m)$

R-9.30) Hint Recall that most hash-table operations run in $O(1)$ expected time.

R-9.30) Solution $O(n)$

R-9.31) Hint Something from this chapter should be helpful!

R-9.31) Solution We should use a sorted multiset, as we want to maintain multiple entries with the same birthday and we want to be able to do neighboring queries.

Creativity

C-9.32) Hint Consider a bootstrapping method for finding the primes up to $\sqrt{2M}$.

C-9.33) Hint Your solution should make a single call to findIndex.

C-9.33) Solution

```
public V putIfAbsent(K key, V value) {
  int j = findIndex(key);
  if (j == −1) {
    table.add(new MapEntry<>(key, value));          // add new entry
    return null;
  } else {
    return table.get(j).getValue( );                // return existing value
  }
}
```

C-9.34) Hint You may assume that such a method is supported by the auxiliary UnsortedTableMap

C-9.35) Hint You must only call the findSlot utility once.

C-9.36) Hint Model your solution after the existing support for other map methods.

C-9.37) Hint You may assume that such a method is supported by the auxiliary UnsortedTableMap

C-9.38) Hint The heart of the process can be performed by the findSlot utility.

C-9.39) Hint When might the load factor fall below the threshold, and how can you detect this from within AbstractHashMap?

C-9.40) Hint Start by defining an appropriate subclass of AbstractMap.MapEntry that includes the new next field that is desired.

C-9.41) Hint You need to do some shifting of entries to close up the "gap" just made, but you should only do this for entries that need to move.

C-9.41) Solution When we remove an entry from a hash table that uses linear probing without using deleted element markers, we must ensure that any cluster of consecutively filled cells remains intact, unless no entries in that cluster after a newly introduced gap have a hash value that falls before such a gap. Assume that we wish to delete an entry stored at index j and that the cluster it belongs to goes until an empty cell at index k. For ease of exposition only, we assume $k > j$, thus the cluster does not wrap around the end of the array. To fill the hole in index j, we wish to find the next entry of the cluster that has a hash value $h \leq j$. If no such entry exists, we may leave a hole at j. Otherwise, assume index i holds the first such entry. We move that entry to j and then we repeat the process to fill the hole that results at index i, finding the next subsequent entry of the cluster with hash value $h \leq i$. Note that the entire process takes time linear in the length of the original cluster of filled cells.

C-9.42) Hint For part a, note that the symmetry will halve the range of possible values. For part b, note that such automatic collisions will not occur.

C-9.43) Hint Perhaps borrow techniques from the Progression hierarchy of Chapter 2, or use Java's iterators.

C-9.44) Hint Maintain a secondary PositionalList instance that represents the FIFO order, and store positions within that list with each entry of the hash table.

C-9.45) Hint Each map entry instance should store its current index within the table.

C-9.46) Hint Each map entry instance should store its current index within the table.

C-9.47) Hint Each map entry instance should store its position within the bucket list.

C-9.48) Hint Try out some examples.

C-9.48) Solution The original version of Code Fragment 10.11 does not guarantee that it finds the leftmost occurrence when duplicates exist, because it stops searching as soon as it finds a match. For example, if the entire table was filled with duplicates, it will report the middle index.

If the version in Exercise R-9.21 had been written as intended, with the $\leq$ operator on line 4, then it is a correct implementation and indeed guarantees that it finds the leftmost of all duplicates.

C-9.49) Hint Do a "double" binary search.

C-9.49) Solution In order to the find the k^{th} smallest key in the union of the keys from S and T, we can do a "double" binary search. In other words, we will begin by examining the $k/2^{\text{th}}$ element in the array list S. Next, we will find the largest element in T that is less than $S[k/2]$ by binary search. Then, we will add the indices of the elements we were examining in S and T. If the sum equals k, then the max of the two elements is our result. If the sum is greater than k, then we will do a binary search to the right (upwards) in S. If the sum is less than k, then we will do a binary search to the left (downwards) in S. This is followed once again by searching in T for largest element less than the current element in S, etc. This method does a binary search on S which requires $O(\log n)$ "probes." However, for each probe of the search, it does a binary search on T which takes $O(\log n)$ time. Thus, the entire method takes $O(\log^2 n)$ time.

C-9.50) Hint Dove-tail two binary searches.

C-9.51) Hint Do a lazy iteration through indices of the underlying array list.

C-9.52) Hint Manage a primary iteration through all nonempty buckets, and then a secondary iteration through each element of a bucket.

C-9.53) Hint Do a lazy iteration through the nonempty cells of the table.

C-9.54) Hint Think about first sorting the pairs by cost.

C-9.54) Solution Sort the pairs by cost. Then scan this list looking at the performance values. Remove any that have performance values worse than the (unremoved) pair that came before.

C-9.55) Hint In the insertion algorithm, first repeatedly flip the coin to determine the level where you should start inserting the new entry.

C-9.56) Hint Consider augmenting each node v in a higher level with the number of missing entries in the gap from v to the next node over.

C-9.57) Hint Consider augmenting each node v in a higher level with the number of missing entries in the gap from v to the next node over.

C-9.58) Hint Think first about how you can determine the number of 1's in any row in $O(\log n)$ time.

C-9.58) Solution To count the number of 1's in A, we can do a binary search on each row of A to determine the position of the last 1 in that row. Then we can simply sum up these values to obtain the total number of 1's in A. This takes $O(\log n)$ time to find the last 1 in each row. Done for each of the n rows, then this takes $O(n \log n)$ time.

C-9.59) Hint Consider a two-pass solution, with use of an auxiliary structure.

C-9.59) Solution

```
public void retainAll(Set<E> other) {
  // let's first build a list of elements to remove
  ArrayList<E> leaving = new ArrayList<E>();
  for (E element : this)
    if (!other.contains(element))
      leaving.add(element);
  // now let's remove those elements from the set
  for (E element : leaving)
    remove(element);
}
```

C-9.60) Hint Think of describing your algorithms in terms of boolean operations on the bit vectors.

C-9.61) Hint Think of how you could transform D into L.

C-9.62) Hint Consider the way subMap was implemented for SortedTableMap.

C-9.62) Solution Simply do a binary search to find an element equal to k. Then step back through the array until you reach the first element equal to

k. Finally, step forward through the area reporting entries until you reach the first key that is not equal to k. This takes $O(\log n)$ time for the search and then at most s time to search back to the beginning of the run of k's and s time return all of the elements with key k. Therefore, we have a solution running in at most $O(s+\log n)$ time.

C-9.63) Hint You need some way of grouping together entries with the same key.

C-9.63) Solution Use a skip list map, with only r nodes in the bottom level, and with each of these having a secondary linked list of elements with the same key. Thus, all the query and update operations will run in $O(\log r)$ expected time, and the getAll method will run in $O(\log r+s)$ expected time.

C-9.64) Hint You need some way of grouping together elements with the same key.

C-9.64) Solution The solution is actually quite simple–just use something like the HashMultimap class given in the book, but make sure that the capacity of the underlying hash table is always $O(n)$. Another possibility is to use a linked hash table, which keeps its entries in a linked list in addition to the hash table. Either way we would get that the entries( ) method would run in $O(n)$ time.

Projects

P-9.65) Hint The biggest challenge is detecting the case of an infinite loop.

P-9.66) Hint When searching for an existing key, make sure to consider both of the possible buckets.

P-9.67) Hint Maintain a secondary PositionalList instance that represents the FIFO order, and store positions within that list with each entry of the hash table.

P-9.68) Hint We've already implemented them for you; all that's left is the experiments.

P-9.69) Hint In a Unix/Linux system, a good place to start is /usr/dict.

P-9.70) Hint It is okay to generate these phone numbers more-or-less at random.

P-9.71) Hint Sentinels can be used in place of the theoretical $-\infty$ and $+\infty$.

P-9.72) Hint Try to make your screen images mimic the skip list figures in the book.

P-9.73) Hint You need to find some way to let the intermediate nodes in the skip list keep track of the number of elements that have been skipped over.

P-9.74) Hint For each word t that results from a minor change to s, you can test if t is in W in $O(1)$ time.