Module 4

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 4 Study Guide and Deliverables

Topics:	• Priority Queues and Heaps • Maps and Hash Tables
Readings:	• Module 4 online content • Textbook: Chapter 9, Chapter 10 (except 10.4)
Assignments:	• Assignment 4 due **Tuesday, April 12 at 6:00 AM ET**
Assessments:	• Quiz 4 due **Tuesday, April 12 at 6:00 AM ET**
Live Classrooms:	• **Tuesday, April 5 from 8:00-9:30 PM** • **Thursday, April 7 from 8:00-10:00 PM** • Another one-hour live office hour session led by your facilitator: TBD

Module 4 Learning Objectives

In this section, we discuss priority queues, maps, and hash tables.

After successfully completing this module, you will be able to:

1. Implement a priority queue using a sorted or unsorted list.
2. Implement a heap data structure and heap operations.
3. Implement a priority queue using a heap.
4. Illustrate how heap-sort works.
5. Implement a heap-sort algorithm.
6. Implement an adaptable priority queue using a heap.
7. Implement a simple map using an unsorted list.
8. Implement a hash table with chaining method.
9. Implement a hash table with open addressing.

Section 4.1 Priority Queues

Section 4.1. Priority Queues

Overview

We discussed a FIFO queue ADT in Section 2.3.2. Elements are removed from the FIFO queue in the same order in which they arrived. In this section, we describe a different type of queue called a *priority queue*, from which elements are removed based on some *priority* criterion, not on when they arrived.

Section 4.1.1 Priority Queue ADT

To understand the behavior of a priority queue, let us look at a real-world example as an analogy. Assume that airline travelers are waiting for boarding at an airport boarding area. If boarding is done on the first-come-first-serve basis, then this is a FIFO queue. However, some airline companies may give higher priority to a certain group of travelers, or assign different priorities to different groups of travelers. For example, people with special assistance are given the highest priority and they are allowed to board before any other travelers. First class passengers and US Military members board next. All other passengers board after that. This is how a priority queue works.

We assume that an element in a priority queue is associated with *key*. When an element is removed from the queue, an element with a *minimal* key is removed. So, an element with a smaller key has a higher priority. Usually, a key is a numeric value. However, objects of any type can be used as far as there is an ordering among the objects.

Now we describe a priority queue ADT. In this design, an element of the queue is referred to as an *entry* and it consists of a key *k* and a value *v*. The priority of an entry is determined by its key. The ADT supports the following operations:

- insert(*k*, *v*): Create an entry with key *k* and value *v* in the priority queue.
- min( ): Returns (but does not remove) an entry (*k*, *v*) with the minimum key. Returns null if the priority queue is empty.
- removeMin( ): Removes and returns an entry (*k*, *v*) with the minimum key. Returns null if the priority queue is empty.
- size( ): Returns the number of entries in the priority queue.
- isEmpty( ): Returns true if the priority queue is empty. Returns false, otherwise.

In the priority queue, it is possible that multiple entries have the same key or equivalent keys. If there are multiple entries with the same minimal key, then the *min* and *removeMin* operations select one among them arbitrarily.

The following example shows a sequence operations performed on a priority queue. Initially the queue is empty and the value of an entry is a character.

Figure 4.1 Demonstration of Priority Queue Operations

Method	Return Value	Priority Queue Contents
insert(17, A)		{(17, A)}
insert(4, P)		{(4, P), (17, A)}
insert(15, X)		{(4, P), (15, X), (17, A)}
size( )	3	{(4, P), (15, X), (17, A)}
isEmpty( )	false	{(4, P), (15, X), (17, A)}
min( )	(4, P)	{(4, P), (15, X), (17, A)}
removeMin( )	(4, P)	{(15, X), (17, A)}
removeMin( )	(15, X)	{(17, A)}
removeMin( )	(17, A)	{ }
removeMin( )	null	{ }
size( )	0	{ }
isEmpty( )	true	{ }

Section 4.1.2. Implementing a Priority Queue

In this section, we first implement the priority queue ADT as a Java abstract base class. This abstract base class defines variables and methods that will be shared by subsequent concrete implementations. Then, we describe two concrete implementations that use a positional list *L* (which we discussed in Section 3.1.3) for storage.

Section 4.1.2.1. Entry Composite

An entry in the priority queue consists of a key and a value, and it is implemented with a *composition design pattern*. We define an interface that stores key-value pairs. A Java code for the interface follows:

Code Segment 4.1

```
1   public interface Entry<K,V> {
2     K getKey();
3     V getValue();
4   }
```

Then we define the interface for the priority queue using the *Entry* interface, the Java code of which is shown below:

Code Segment 4.2

```
1   public interface PriorityQueue<K,V> {
2     int size();
3     boolean isEmpty();
4     Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
5     Entry<K,V> min();
6     Entry<K,V> removeMin();
7   }
```

Section 4.1.2.2. Comparing Keys with Total Orders

As mentioned earlier, any type of objects can be used as keys as far as there is a total ordering among those objects. A total ordering guarantees that we can compare any two objects with a deterministic result, and the results of such comparisons are not contradictory to each other. A *total ordering*, ρ, is a binary relation among a set of objects that satisfies the following properties. Let k_1, k_2, and k_3 be objects of the set.

- Comparability property—$k_1 \ \rho \ k_2$ or $k_2 \ \rho \ k_1$.
- Antisymmetric property—If $k_1 \ \rho \ k_2$ and $k_2 \ \rho \ k_1$, then $k_1 \ \rho = k_2$.
- Transitive property—If $k_1 \ \rho \ k_2$ and $k_2 \ \rho \ k_3$, then $k_1 \ \rho \ k_{23}$.

The comparability property guarantees that any pair of keys can be compared. This property also implies the following property:

- Reflexivity property—$k \ \rho \ k$

A total ordering defines a *linear* ordering among all elements in the set.

The comparison rule we use to compare keys of the priority queue, for which we use the symbol $\leq$, must satisfy this total-ordering property. Then we can compare any two keys, there can be no contradictory results, and a *minimal key* is well defined. A *minimal key,* key_{min}, is a key with the following property: $key_{min} \leq k$, for any other key k in the set.

In Java, there are two ways to compare objects. One way is to use the *compareTo* method defined in the *java.util.Comparable* interface. We implement the *Comparable* method and override the *compareTo* method to define a total ordering of objects, called the *natural ordering*. Suppose the *compareTo* method is invoked with *a.compareTo*(*b*). Then the return value is one of the following:

- A negative number, if $a < b$
- Zero, if $a = b$
- A positive number, if $a > b$

Many Java classes implemented the *Comparable* interface, which include *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Character*, *String*, and *Date*. For example, String objects can be compared via the *compareTo* method, and the comparison is based on the natural ordering, or lexicographical ordering, of strings.

Java provides another interface that we can use to specify a total ordering among objects, the *java.util.Comparator* interface. The *Comparator* interface has single abstract method, *compare*. We use the *Comparator* interface when we want to compare objects, but not by their natural ordering. For example, we may want to compare strings based on their lengths. Then we can write a class that implements the *Comparator* interface and use the *comparator* object for such comparison. A *comparator* object is external to the class of keys being compared. It can be passed as an argument, for example, to a sorting method. Then the

sorting method will use the *comparator* when comparing objects.

The following example illustrates how to use the *Comparator* interface to compare strings by length. We first define a comparator class that implements the *Comparator* interface. The comparator class compares strings based on their lengths. Then we use that comparator to compare three strings.

Code Segment 4.3

```
1   public class StringLengthComparator implements Comparator<String> {
2      public int compare(String a, String b){
3          if (a.length() < b.length()) return -1;
4          else if (a.length() == b.length()) return 0;
5          else return 1;
6      }
7   }

8   public class ComparatorTest {
9      public static void main(String[] args) {
10         StringLengthComparator c = new StringLengthComparator();
11         String s1 = "tiger";
12         String s2 = "sugar";
13         String s3 = "coffee";
14         String s4 = "cat";
15         System.out.println("Compare s1 and s2: " + c.compare(s1, s2));
16         System.out.println("Compare s1 and s3: " + c.compare(s1, s3));
17         System.out.println("Compare s1 and s4: " + c.compare(s1, s4));

18         ArrayList<String> stringList = new ArrayList<>();
19         stringList.add(s1);
20         stringList.add(s2);
21         stringList.add(s3);
22         stringList.add(s4);
23         stringList.sort(c);
24         System.out.print("Sort string by length: ");
25         for (String s : stringList)
26              System.out.print(s + " ");
27     }
28  }
```

Note that, in line 23, the comparator *c* is passed to the *sort* method as an argument. The expected output is as follows:

```
Compare s1 and s2: 0
Compare s1 and s3: -1
Compare s1 and s4: 1
Sort string by length: cat tiger sugar coffee
```

The implementation of the priority queue allows keys of any type. A user needs to send a comparator to the constructor of the priority queue so that the priority may use this comparator when comparing keys.

A default comparator is also defined for convenience; it relies on the natural ordering of the keys. A Java code for the default comparator is shown below:

Code Segment 4.4

```
1   public class DefaultComparator<E> implements Comparator<E> {
2     public int compare(E a, E b) throws ClassCastException {
3       return ((Comparable<E>) a).compareTo(b);
4     }
5   }
```

In line 3, two objects *a* and *b* are compared using the *compareTo* method of the *Comparable* interface, which provides a natural ordering of the objects.

Section 4.1.2.3. The AbstractPriorityQueue Base Class

The *AbstractPriorityQueue* class is an abstract base class that provides common features for different concrete implementations. The *Entry* interface (of Section 4.1.2.1) is implemented as a nested class *PQEntry*, which is shown below:

Code Segment 4.5

```
1   protected static class PQEntry<K,V> implements Entry<K,V> {
2     private K k;  // key
3     private V v;  // value
4     public PQEntry(K key, V value) {
5       k = key;
6       v = value;
7     }
8     public K getKey() { return k; }
9     public V getValue() { return v; }
10    protected void setKey(K key) { k = key; }
11    protected void setValue(V value) { v = value; }
12  }
```

The abstract base class also includes the following:

- Instance variable *comp*—This is the comparator used to compare keys.
- Two constructors—One receives a comparator as an argument. The other is a default constructor that uses the default comparator described in the previous section.
- *compare* method—This method compares two keys using the comparator *comp*.
- *checkKey* method—This method checks whether a given key is valid.
- *isEmpty* method—This method checks whether the queue is empty.

A Java code for *AbstractPriorityQueue* class can be found at the *AbstractPriorityQueue.java* file.

Section 4.1.2.4. Implementing a Priority Queue with an Unsorted List

This section describes the implementation of the priority queue as an *UnsortedPriorityQueue* class. It extends, and inherits from, the *AbstractPriorityQueue* class. It uses an *unsorted* list as an underlying storage. Entries are stored in an unsorted *PositionalList* (discussed in Section 3.1.3). Its members include the following:

- Instance variable *list*—This is a positional list of *LinkedPostionalList* type. As discussed in Section 3.1.3.3, it uses a doubly linked list.
- Two constructors—One creates an empty priority queue that uses a natural ordering. The other creates an empty priority queue that uses the comparator that is passed as an argument.
- *min* method—This method returns (but does not remove) the entry with a minimal key.
- *removeMin* method—This method removes the entry with a minimal key.
- *findMin* method—Since the list is unsorted, the *min* and *removeMin* methods require the list to be scanned to find an entry with a minimal key. This *findMin* method is a helper method that performs the scanning and finds and returns the position of the entry with a minimal key.
- *insert* method—This method first checks that the given key is valid. Then it creates a new entry (with the key/value pair that is passed) and adds it to the queue.
- *size* method—This method returns the number of elements in the priority queue by invoking the *size* method of *list* (because the number of elements in the

queue is the same as the number of elements in the list).

The *min* and *removeMin* methods scan the list to find an entry with a minimal key (via the *findMin* method). So, the worst-case running time of these method is $O(n)$. All other methods take $O(1)$ time.

A Java code of the *UnsortedPriorityQueue* class can be found at the *UnsortedPriorityQueue.java* file.

Section 4.1.2.5. Implementing a Priority Queue with a Sorted List

This implementation uses the same positional list. The difference is that the entries in the list are always sorted. The advantage is that the *min* and *removeMin* operations now require only $O(1)$ time because an entry with a minimal key is always at the head of the list.

However, to maintain the order of entries in the list, the *insert* method must do more work. It must find the correct position for each entry added to the sorted list. In this implementation, the *insert* method scans the list from the tail backward until it finds the entry with a key that is smaller than or equal to that of the new entry, and the new entry; the new entry is then inserted after that position. This requires $O(n)$ time. The code of the *insert* method is shown below:

Code Segment 4.6

```
1   public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
2     checkKey(key);    // auxiliary key-checking method (could throw exception)
3     Entry<K,V> newest = new PQEntry<>(key, value);
4     Position<Entry<K,V>> walk = list.last();
5     // walk backward, looking for smaller key
6     while (walk != null && compare(newest, walk.getElement()) < 0)
7       walk = list.before(walk);
8     if (walk == null)
9       list.addFirst(newest);                  // new key is smallest
10    else
11      list.addAfter(walk, newest);            // newest goes after walk
12    return newest;
13  }
```

Line 2 ensures that the given key is valid. Line 3 creates a new entry with the given key/value pair. In the *while* loop of lines 6 and 7, while walking backward, the method finds the first entry of which the key is smaller than or equal to the given key. Then the new entry is added after that entry in line 11. If the new key is the smallest, then the entry is added at the head of the list in line 9.

A complete Java code of the *SortedPriorityQueue* can be found at the *SortedPriorityQueue.java* file.

In the implementation with an unsorted list, the *min* and *removeMin* methods take $O(n)$ time, and the *insert* method takes $O(1)$ time. In the implementation with a sorted list, however, the *min* and *removeMin* methods take $O(1)$ time and the *insert* method takes $O(n)$ time. So, there is a trade-off between the two implementations. Based on the behavior of the application that uses the priority queue—or, more specifically, based on the respective frequencies of those three operations—a user should choose an appropriate implementation for the application. The following table summarizes the running times of operations in the two implementations:

Figure 4.2. Performance of Priority Queue Implemented Using Unsorted List and Sorted List

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

Section 4.1.3. Heaps

In the previous section, we saw how a priority queue ADT can be implemented using positional list, which is based on a doubly linked list. We also saw that there is a trade-off between the implementation using a sorted list and that using a sorted list. In both implementations, some operations take $O(1)$ time and others take $O(n)$ time.

In this section, we discuss a different implementation, which uses a data structure called a *binary heap*. With this implementation, the running times of the *insert* and *remove* operations are *logarithmic* time.

Section 4.1.3.1. Heap Data Structure

A *heap* is a binary tree T that satisfies two properties. One property specifies the relationship between the key of a node and that of its parent. The other property specifies the shape of the tree T. The former is referred to as the *heap-order property* and the latter, the *complete-binary tree property*.

Heap-order property: In a heap T, for every position p except the root, the key stored at p is greater than or equal to the key stored at p's parent.

There are two important additional properties we can draw from the heap-order property. First, when we traverse a path from the root to any leaf node, keys along the path are in nondecreasing order. The other property is that the key at the root node is always a minimal key. Because of this second property, the *removeMin* operation can be performed efficiently.

Before we state the *complete-binary tree property*, we first define a *complete binary tree*.

A binary tree T with height h is a *complete binary tree* if both of the following are true:

- Levels $0, 1, ..., h-1$ of T have the maximal number of nodes (in other words, level i has 2^i nodes, where $0 \leq i \leq h-1$.
- Nodes at level h are in the leftmost possible positions at that level.

In Figure 4.3, the tree (a) is a complete binary tree, but the tree (b) is not.

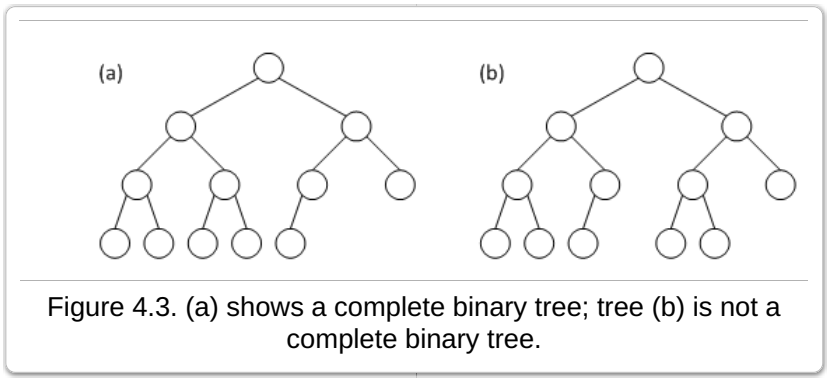


Figure 4.3. (a) shows a complete binary tree; tree (b) is not a complete binary tree.

Now we state the *complete binary tree property*.

Complete-binary tree property: A heap is a complete binary tree.

The following figure is an example of a priority queue, which is implemented using a heap data structure. In the queue, a position (or a node in the heap) stores a (key, value) pair, in which a key is an integer and a value is a character.

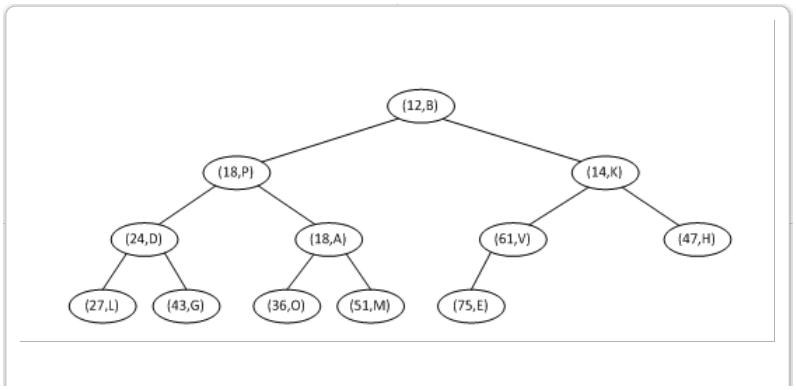


Figure 4.4. A priority queue implemented using a heap.

One important consequence of the complete-binary tree property is that the height of a heap is a logarithmic function of the total number of nodes, n, in the heap. More specifically, the height of a heap storing n entries is defined as $h = \lfloor \log n \rfloor$. We can prove this as follows.

Consider a heap T storing n entries. Let h be the height of the heap. According to the complete binary tree property, the number of nodes at levels 0 through $n-1$ is $1 + 2 + ... + 2^{h-1} = 2^h - 1$. Further, the number of nodes at level h is at least 1 and at most 2^h. So, the total number of nodes, n, in a heap T with height h is as follows:

$$2^h - 1 + 1 \leq n \leq 2^h - 1 + 2^h$$
$$2^h \leq n \leq 2^{h+1} - 1$$

From the lower end of the inequality, we have $2^h \leq n$. If we take the logarithm of the both sides, we have

$$h \leq \log n.$$

From the upper end of the inequality, we have $n \leq 2^{h+1} - 1$, which can be rearranged to give the following:

$$n + 1 \leq 2^{h+1}$$

If we take the logarithm of the both sides, we proceed as follows:

$$\log(n+1) \leq \log(2^{h+1})$$
$$\log(n+1) \leq \log(2^h.2)$$
$$\log(n+1) \leq \log 2^h + \log 2$$
$$\log(n+1) \leq h + 1$$
$$\log(n+1) - 1 \leq h$$

So, we have $\log(n+1) - 1 \leq h \leq \log n$, and the integer that satisfies this inequality is $h = \lfloor \log n \rfloor$.

Section 4.1.3.2. Implementing a Priority Queue with a Heap

In the previous section, it was shown that the height of a heap, h, is a logarithmic function of the number of entries in the heap, n. So, if we design update operations of which the performance is proportional to the height of the tree, then their running time will be $O(\log n)$.

Adding an Entry to a Heap

The insertion of an entry into a priority queue implemented with a heap is performed in two steps. In the first step, the new entry is added to the "end" of the current heap. If the bottom level of the heap is not full, then the new entry becomes the rightmost node at that level. If the bottom level of the heap is full, then the new entry becomes the leftmost node at the next level.

The new entry is added to the end of the heap may violate the heap-order property. For example, the key of the new entry may be smaller than the key of its parent. So, in the second step, the heap is reorganized, if needed, to maintain the heap-order property. This is done in a bottom-up fashion.

If the heap is empty and the new entry is added as the root, then nothing needs to be done. Otherwise, we begin at the new entry and repeatedly perform the following. Let e be the new entry. If the key of e is smaller than the key of its parent, then e is swapped with its parent. Otherwise, we stop. If the swap occurs, the new entry e is now in its previous parent's position. We then repeat the process with its new parent—i.e., if the key of e is smaller than the key of its new parent, then e is swapped with its new parent, and so on. This process is called *up-heap bubbling*. It continues until it reaches the root or a swap does not occur. The following figure shows how a new entry is inserted into a heap and illustrates the up-heap-bubbling process.

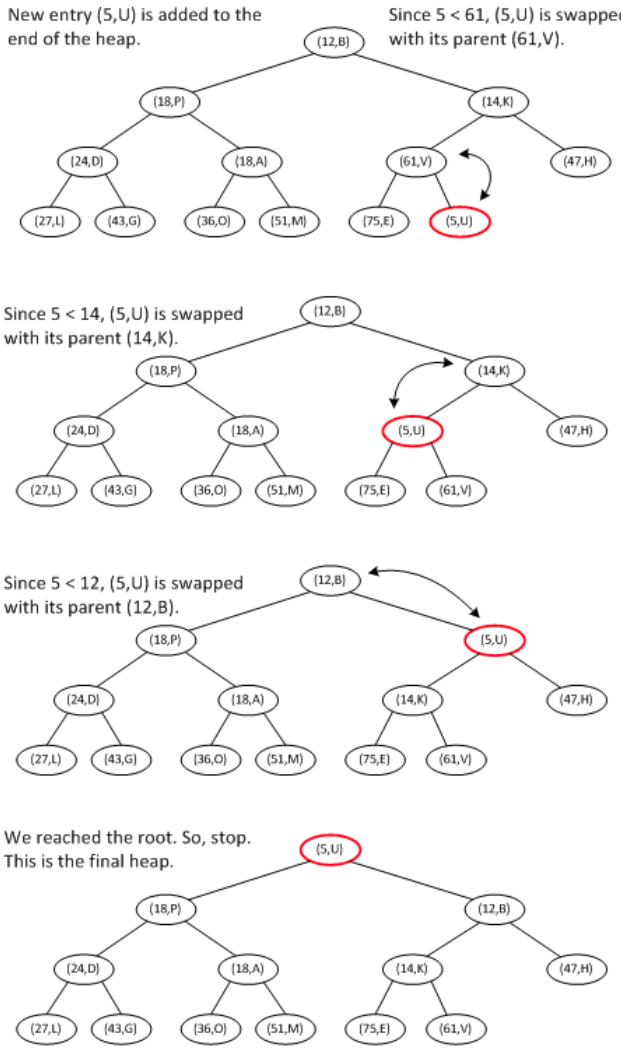


Figure 4.5. Adding an entry to a priority queue implemented with a heap.

The up-heap-bubbling process needs, at most, h swaps. So, its running time is $O(h)$ or $O(\log n)$. Note that when we do asymptotic running time analysis, we ignore floor and ceiling functions. So, the running time of the insertion operation is $O(\log n)$.

Removing the Entry with Minimal Key

Since an entry with a minimal key is at the root of a heap, removing the entry itself is simple. However, after removing the root entry, the heap needs to be reorganized, and we have to make sure that the heap-order property still holds in the new heap.

The removal is also performed in two steps. In the first step, the root entry is removed, and the last node in the heap is moved up to the root position. Then, in the second step, the new heap with the new root is reorganized to maintain the heap-order property.

The reorganization of the heap is performed in a top-down manner. This process is called *down-heap bubbling* and is the opposite of up-heap bubbling.

If the heap has only one entry, which is the root, then the new heap becomes empty and nothing needs to be done. Otherwise, we begin at the root node and move downward, correcting any violation of the heap-order property.

Suppose that we are at an entry e (initially the root is e). Let c be one of e's two children, with the smaller key of the two. In other words, if the left child's key is smaller than the right child's key, then the left child becomes c. Otherwise, the right child becomes c. If there is only one child, then that child is c. If the key of e is smaller than or equal to the key of c, then we stop. Otherwise, e is swapped with c. This process continues until we reach a leaf node or a swap does not occur.

The following figure shows the removal of an entry from a priority queue implemented using a heap. It also illustrates the down-heap bubbling.

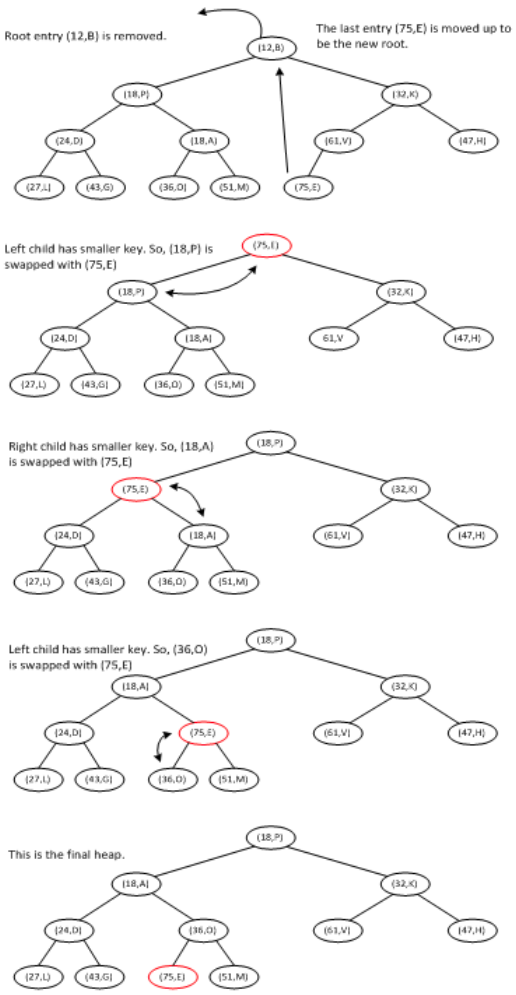


Figure 4.6. Removing an entry from a priority queue implemented with a heap.

Representing a Complete Binary Tree Using an Array

An advantage of a heap data structure, which is a complete binary tree, is that it can be easily represented as an array, making access operations very efficient. Assume that we have a heap with n elements, or size n. Then the heap can be stored in an array with n elements, and each entry in the heap can be accessed with an index between 0 and $n - 1$.

Let $f(p)$ be the level number of the position p of an entry in a heap (refer to Section 3.2.3.2 for the discussion of level numbering). Then, that entry is stored in the array at index $f(p)$. As discussed in Section 3.2.3.2, the index of the root node's position is defined to be 0. The indexes of its children are easily calculated using the following:

- Index of left child of $p = 2f(p) + 1$
- Index of right child of $p = 2f(p) + 2$

In addition, the index of a position *p* can be easily calculated as follows:

- Index of parent of $p = \lfloor (f(p) - 1)/2 \rfloor$

The following figure shows a heap and the array that stores the heap elements:

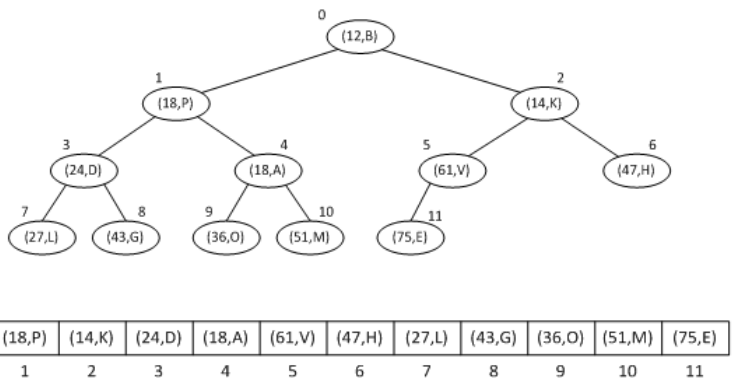


Figure 4.7. A heap, which is a complete binary tree, and an array that stores heap elements.

Java Implementation of Heap-Based Priority Queue

We now introduce a Java implementation of the heap-based priority queue. It is implemented as the *HeapPriorityQueue* class, which extends the abstract base class *AbstractPriorityQueue*. In this case, a heap is implemented as an *ArrayList* of composite entries, which keeps a paired key and value. The position of an entry in the heap is the index of the entry in the array list.

It has one instance variable *heap* of *ArrayList* type. Its declaration and the declaration part of the *HeapPriorityQueue* class are shown below:

```
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {

  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();
```

It includes utility methods that calculate the indexes of the parent, left child, and right child of an entry. It implements methods for adding an entry to and removing an entry from a priority queue. It also implements the *swap* method, the *upheap* method (an implementation of the up-heap bubbling), and the *downheap* method (which is an implementation of the down-heap bubbling).

We briefly discuss the *upheap*, *downheap*, *insert*, and *removeMin* methods.

A Java code of the *upheap* method is given below:

Code Segment 4.7

```
1   protected void upheap(int j) {
2     while (j > 0) { // continue until reaching root (or break statement)
3       int p = parent(j);
4       if (compare(heap.get(j), heap.get(p)) >= 0) break; // no violation
5       swap(j, p);
6       j = p;        // continue from the parent's location
7     }
8   }
```

The argument j is the index of the entry where the up-heap bubbling begins. Line 3 gets the index of its parent. If the key of the entry j is larger than or equal to the key of its parent, then the heap-order property is satisfied and we stop (line 4). Otherwise, it is swapped with its parent in line 5, and the same is repeated from its parent position in line 6 (which now has the entry that was at the index j location).

The following is a Java code of the *downheap* method:

Code Segment 4.8

```
1   protected void downheap(int j) {
2     while (hasLeft(j)) {     // continue to bottom (or break statement)
3       int leftIndex = left(j);
```

```
4       int smallChildIndex = leftIndex; // although right may be smaller
5       if (hasRight(j)) {
6           int rightIndex = right(j);
7           if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
8             smallChildIndex = rightIndex;  // right child is smaller
9       }
10      if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
11        break;                      // heap property has been restored
12      swap(j, smallChildIndex);
13      j = smallChildIndex;        // continue at position of the child
14    }
15  }
```

The argument j is the index of the entry where the down-heap bubbling begins. The local variable *smallChildIndex* stores the index of the child that has a smaller key. Initially, it is set to the index of the left child in line 4. If the right child exists (which is tested in line 5), then its key and that of the left child are compared. If the key of the right child is smaller (line 7), then *smallChildIndex* is set to the index of the right child (line 8). Next, the key of the entry at index *smallChildIndex* is compared with the key of the entry at *j*. If the key of the entry at *smallChildIndex* is greater, there is no violation of the heap-order property, and we stop. Otherwise a swap occurs (in line 12) and we continue moving downward move at the child position.

Java implementations of the *insert* and *removeMin* methods are shown below:

Code Segment 4.9

```
1   public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
2     checkKey(key);  // auxiliary key-checking method (could throw exception)
3     Entry<K,V> newest = new PQEntry<>(key, value);
4     heap.add(newest);                     // add to the end of the list
5     upheap(heap.size() - 1);              // upheap newly added entry
6     return newest;
7   }

8   public Entry<K,V> removeMin() {
9     if (heap.isEmpty()) return null;
10    Entry<K,V> answer = heap.get(0);
11    swap(0, heap.size() - 1);             // put minimum item at the end
12    heap.remove(heap.size() - 1);         // and remove it from the list;
13    downheap(0);                          // then fix new root
14    return answer;
15  }
```

The *insert* method receives, as its arguments, the key and the value of the entry to be inserted. Line 2 checks that the key is valid. In line 3, a new entry with the key and the value is created. Then it is to the heap (which is *ArrayList*) in line 4 and the *upheap* method is invoked to start the up-heap-bubbling process.

The *removeMin* method first checks whether the heap is empty in line 9. If it is not empty, the root node is stored in a temporary variable *answer*. Next, the root node is swapped with the last node in line 11, and the last node—which was at the root—is removed in line 12. Now, the node that was at the last position is at the root. In line 13, the *downheap* method is invoked to maintain the heap-order property. Line 14 returns the removed entry.

A complete Java code of the *HeapPriorityQueue* class can be found at the *HeapPriorityQueue.java* file.

Section 4.1.3.3. Analysis of Heap-Based Priority Queue

The *size* method returns the size of *heap*, which is an *ArrayList*. So, it takes $O(1)$. The *isEmpty* method returns ($size() == 0$) and takes $O(1)$. The *min* method returns (but does not remove) the entry at the root of the priority queue. So, it takes $O(1)$.

The *insert* and *removeMin* methods take $O(\log n)$ time. The up-heap bubbling swaps a node with its parent while moving upward from a leaf node. In the worst case, it has to move all the way up to the root, and the number of swaps is the height of the tree, which is $\lfloor \log n \rfloor$. So, the running time of up-heap bubbling is $O(\log n)$. We can do the same analysis for the down-heap bubbling and determine that it takes $O(\log n)$. So, the *insert* method, which relies

on the up-heap bubbling, takes $O(\log n)$ and the *removeMin* method, which relies on the down-heap bubbling, takes $O(\log n)$.

The following table is a summary of the running times:

Figure 4.8. Performance of heap-based priority queue.

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)$
removeMin	$O(\log n)$

Section 4.1.3.4. Bottom-Up Heap Construction

To build a heap with n entries, we can insert the entries one at a time. This takes $O(n \log n)$ time (n insertions with each taking $O(\log n)$).

If all n entries are given in advance, there is a more efficient way of constructing a heap. This method constructs a heap in a bottom-up manner and takes $O(n)$.

In this section, we discuss this bottom-up heap construction method. To make the description simple, we assume that a heap stores $n = 2^{h+1} - 1$ entries. In other words, the heap is a complete binary tree, and every level of it is full. The height of the heap $h = \log(n + 1) - 1$. The bottom-up construction consists of $h + 1$ steps, beginning at the leaf level, as follow:

1. Step 1: We construct $(n + 1)/2$ elementary heaps. Each elementary heap stores a single entry. These heaps are at height 0.
2. Step 2: We construct $(n + 1)/4$ heaps. These heaps are at height 1. Each heap has three entries, consisting of a pair of two heaps (with one entry each) from height 0 and a new entry. The new entry is the root of the heap at the beginning. Since, the heap-order property may be violated, down-heap bubbling is performed on the root node.
3. Step 3: We construct $(n + 1)/8$ heaps. These heaps are at height 2. Each heap has seven entries, consisting of a pair of two heaps (with three entries each) from height 1 and a new entry. The new entry is the root of the heap at the beginning. Since the heap-order property may be violated, down-heap bubbling is performed on the root node.

   …

   In general, at step $i, 2 \le i \le h, (n + 1)/2^i$ heaps are formed at height $i - 1$. Each heap stores $2^i - 1$ entries, consisting of a pair of two heaps (with $2^i - 1$ entries each) from height $i - 2$ and a new entry at the root. Then down-heap bubbling is performed on the root.

   …

   $h + 1$. In the last step, the final heap is formed at height h. Two heaps at height $h - 1$—with $(n - 1)/2$ entries each—are merged, and a new entry is placed at the root. Again, down-heap bubbling is performed to preserve the heap-order property.

Figure 4.9 illustrates the bottom-up heap-construction process. It is assumed that the following sequence of 15 entries is given (only keys are shown, for simplicity):

14, 5, 8, 25, 9, 11, 17, 16, 15, 4, 12, 6, 7, 23, 20

From this sequence, we will build a heap with $n = 15$ nodes with height $h = 1$. Note that the order of entries in the given sequence is not important. It is not important, nor is the order in which the entries are added to the heap in the bottom-up process. In this example, we add entries to the heap from the end of the given sequence while moving backward. The heap-construction process takes four steps.

At step 1, we add the last eight entries from the sequence—16, 15, 4, 12, 6, 7, 23, 20—to build eight elementary heaps at height 0. At step 2, we add the next four entries—25, 9, 11, 17—to build four heaps at height 1. Each heap contains three entries: one each from two heaps of height 0 and one new entry. At step 3, we add the next two entries—5, 8—to form two heaps at height 2. Each heap has seven entries—three each from two heaps at height 2 and one new entry. Finally, at step 4, the entry 14 is added to form a single heap with all 15 entries.

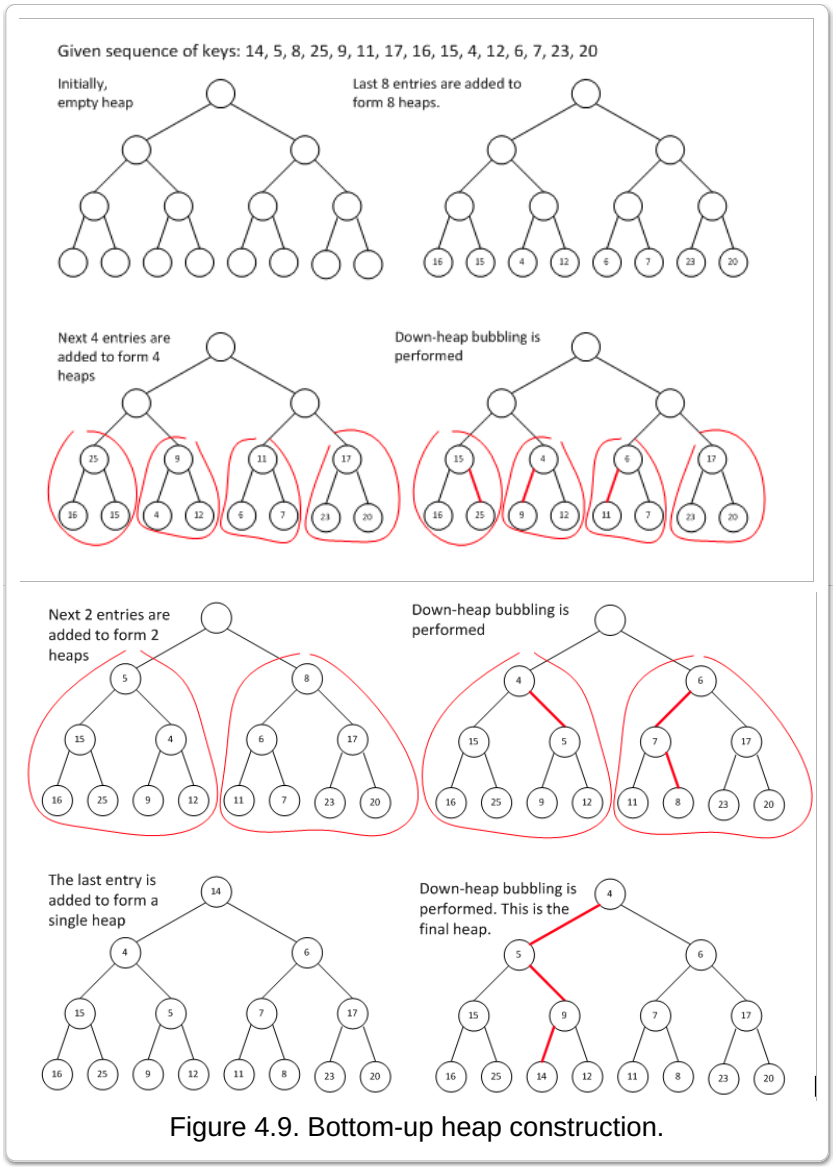


Figure 4.9. Bottom-up heap construction.

The running time of the bottom-up heap construction is $O(n)$. We will not prove this running time. Interested students are referred to the analysis on page 383 in our textbook.

Java Implementation of Bottom-Up Heap Construction

Implementation of the bottom-up heap construction in Java is straightforward. In this section, we discuss it.

As we discussed earlier, a heap can be stored in an array. So, we initially store all n entries in an array in an arbitrary order. Then we apply down-heap bubbling to each entry in the tree. Since the leaf nodes do not have children, we do not need to apply down-heap bubbling to them. Instead, we start with the deepest internal position. In the implementation discussed in this section, we begin with the last node with a child and move backward in the array. Assume that we are given the same 15 entries that we used as an example in the previous section, copied below:

14, 5, 8, 25, 9, 11, 17, 16, 15, 4, 12, 6, 7, 23, 20

The following figure shows the array that stores the 15 entries, and the corresponding heap:

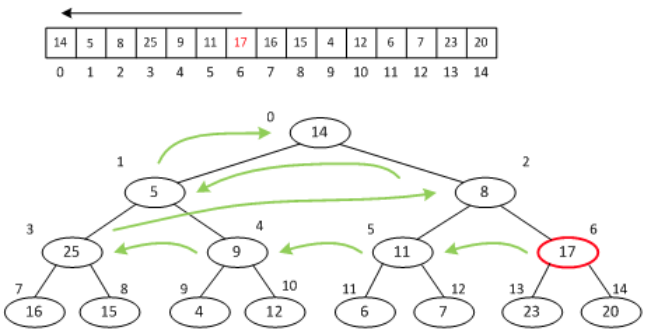


Figure 4.10. Bottom-up heap-construction example. It shows a sequence of down-heap bubbling call.

In the heap, the last node with a child is the entry with key 17, at position 6. So, we apply the down-heap bubbling at position 6, followed by position 5 and so on, until we reach the root. The order of the down-heap bubbling's application is indicated with arrows in the figure. The process is effectively the same as the one shown in the previous section.

A Java code implementing the bottom-up heap construction is shown below. This code segment is included in the code of *HeapPriorityQueue.java*.

Code Segment 4.10

```
1   public HeapPriorityQueue(K[] keys, V[] values) {
2     super();
3     for (int j=0; j < Math.min(keys.length, values.length); j++)
4       heap.add(new PQEntry<>(keys[j], values[j]));
5     heapify();
6   }

7   protected void heapify() {
8     int startIndex = parent(size()-1);  // start at PARENT of last entry
9     for (int j=startIndex; j >= 0; j--) // loop until processing the root
10      downheap(j);
11  }
```

The constructor *HeapPriorityQueue* of lines 1 through 6 creates an empty heap and then add entries from the given arrays of keys and values to the heap, which is an *ArrayList*.

The *heapify* method performs the actual construction of the heap. The *startIndex* in line 8 is the index of the last node that has a child. In the *for* loop of lines 9 and 10, the *downheap* method is invoked, beginning at the *startIndex* and moving backward in the *ArrayList* (note that the index is decremented in line 9).

Section 4.1.3.5. Java's Priority-Queue Class

In Java, the *java.util.PriorityQueue* class implements a priority queue. It is a subclass of *java.util.AbsractQueue*, which is an abstract base class for some queue operations.

In Java's priority queue, an entry is a single element. Elements are ordered, by default, according to either their natural ordering or a comparator that is passed to the constructor used to create a priority queue.

Some operations of the *PriorityQueue* class are shown below. Here, *E* is a generic type parameter.

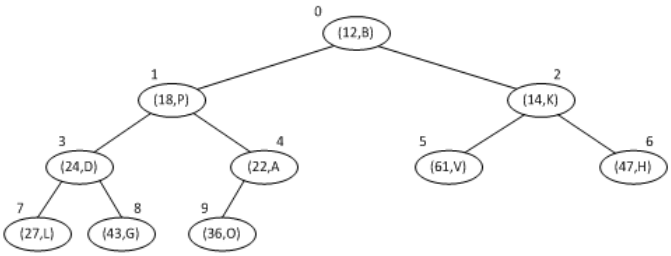
- add(E e)—Inserts the specified element e into the priority queue.
- isEmpty( )—Returns *true* if the priority queue contains no element.
- peek( )—Retrieves, but does not remove, a minimal element from the priority queue.
- remove( )—Removes a minimal element from the priority queue.

- size( )—Returns the number of elements in the priority queue.

Java's priority queue is implemented using a heap. The *add* and *remove* methods run in $O(\log n)$ time. The *isEmpty*, *peek*, and *size* methods take $O(1)$ time.

Test Yourself 4.1

A heap can be represented as an array. Consider the following heap:



This heap can be represented as the following array. Note that, for simplicity, the array is shown as a sequence of keys.

<12, 18, 14, 24, 22, 61, 47, 27, 43, 36>

Suppose that we insert a new entry (15, X) using the *insert* method in the Code Segment 4.9 (this code is also shown in page 378 of the textbook). Show, step by step, the change of the contents of the array.

Please think carefully, write your answer, and then click "Show Answer" to compare yours to the suggested answer.

Suggested answer:

```
<12,18,14,24,22,61,47,27,43,36,15> //15 is added at the end
<12,18,14,24,15,61,47,27,43,36,22> //15 is swapped with 22
<12,15,14,24,18,61,47,27,43,36,22> //15 is swapped with 18
Stop.
```

The following is the final heap:

Section 4.1.4. Sorting with a Priority Queue

We can sort elements using a priority queue in nondecreasing order. We assume that the elements to be sorted are given as a sequence *S*, and we use a priority queue *P*, which is initially empty. A general approach consists of two phases.

1. We insert all elements in *S*, one at a time, into *P*.
2. We remove all elements from *P*, one at a time, using the *removeMin* operation and put them back in *S* in that order.

A Java implementation of this general approach is given below. This implementation assumes that *S* is implemented using a positional list.

Code Segment 4.11

```
1   public static <E> void pqSort(PositionalList<E> S, PriorityQueue<E,?>P){
2     int n = S.size();
3     for (int i=0; i<n; i++){
4       E element = S.remove(S.first());
4       E element = S.remove(S.first());
5       P.insert(element, null);
6     }
7     for (int i=0; i<n; i++){
8       E element = P.removeMin().getKety();
9       S.addLast(element);
10    }
11  }
```

The first *for* loop (lines 3 through 6) implements the first phase, and the second *for* loop (lines 7 through 10) implements the second phase.

Depending on which data structure is used as an underlying storage, we can develop a different sorting algorithm. If an implementation uses an unsorted list, we can develop a *selection-sort* algorithm. If an implementation uses a sorted list, an *insertion-sort* algorithm can be developed. Both sorting algorithms run in $O(n^2)$ time. These two algorithms are discussed in Section 9.4.1 of the textbook.

In this lecture, we discuss only a *heap-sort* algorithm, which uses a priority queue implemented via a heap data structure. We use the general approach discussed earlier and adapt it for a heap-based priority queue.

We first show that the heap-sort algorithm runs in $O(n \log n)$ time.

In the first phase, we insert n elements into the priority queue one at a time. Each *insert* operation takes, in the worst case, $O(\log n)$ time (refer to the analysis in Section4.1.3.3). So, inserting all elements takes $O(n \log n)$.

In the second phase, n elements are removed from the priority queue one at a time. Since each *removeMin* operation takes $O(\log n)$ in the worst case, the second phase also takes $O(n \log n)$.

So, the total running time of a heap-sort is $O(n \log n)$.

We now describe a heap-sort algorithm. The algorithm uses an array-based heap data structure and sorts elements in an array without using additional storage. In other words, during the sorting process, all elements remain in the initial array, though they may move from one place to another. This type of sorting algorithm is referred to as an *in-place* sorting algorithm.

To achieve the in-place sorting, the heap structure is modified. In the previous heap, the heap-order property guaranteed that a key in a node was *greater* than or equal to the key of its parent. This property is now redefined as follows:

> In a heap T, for every position p except the root, the key stored at p is *smaller* than or equal to the key stored at p's parent.

This type of heap is called a *maximum-oriented* heap. In a maximum-oriented heap, the node at the root has the largest key—or the root node has an entry with a maximal key. Let's call the previous type of heap a *minimum-oriented* heap. We can implement a maximum-oriented heap by modifying the minimum-oriented heap. In the minimum-oriented heap, during the up-heap-bubbling process, a node is swapped with its parent if its key is *smaller* than the key of its parent. In the maximum-oriented heap, we modify the up-heap bubbling in such a way that a swap occurs if the key of a node is *greater* than the key of its parent. Similar modification is done to the down-heap bubbling. A node is swapped with the child with a *larger* key, instead of the child with a *smaller* key as was done in the minimum-oriented heap. The *removeMin* method is renamed the *removeMax* method, but its operation remains unchanged. Other minor adjustments may have to be made depending on actual implementation, but these are the main changes.

The sorting is accomplished in two steps:

In the first step, the given n elements are inserted into a maximum-oriented heap.

In the second step, the root node is first swapped with the last node. This moves an entry with a maximal key to the end of the heap (or to the end of the array). Then the heap size is decremented (this excludes the last node from the heap), and the down-heap bubbling is applied to the root node, correcting a potential violation of the heap-order property. This process is then repeated until there is only one node left in the heap.

The following figure illustrates the sorting process. The figure shows both the array, which is an actual data structure storing the elements, and the heap, which is a logical data structure. In the array, those elements that are already sorted and excluded from the heap are highlighted with yellow. The current, proper part of the heap is indicated in red in the figure.

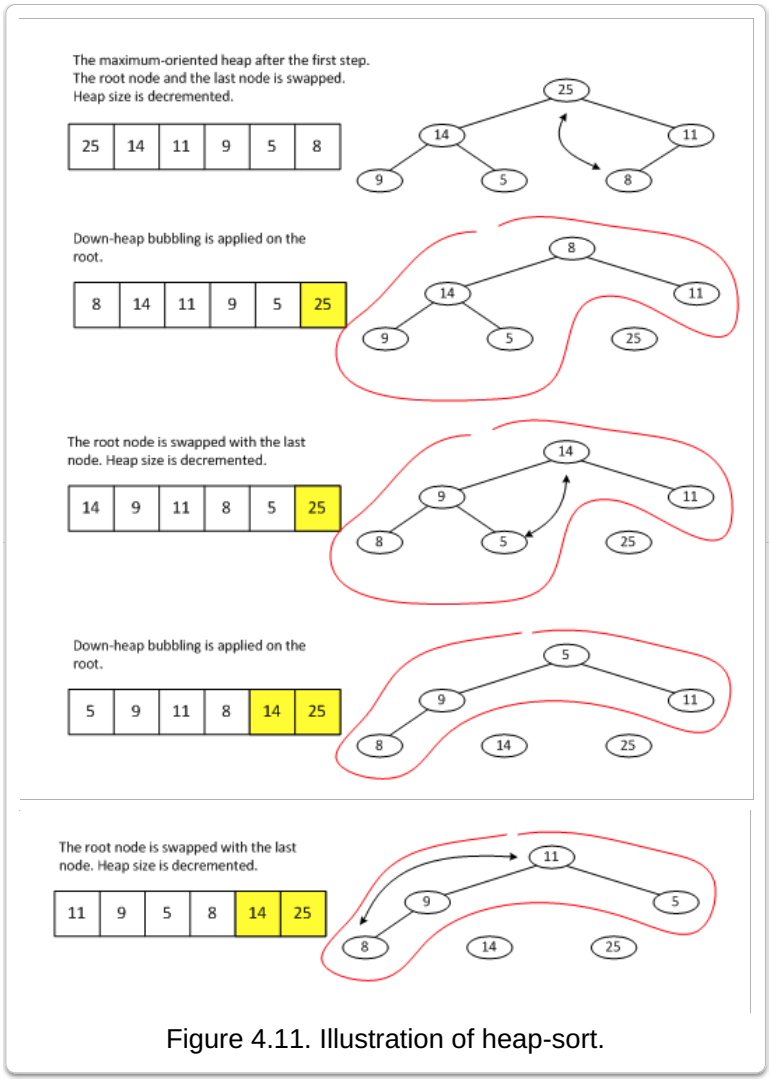
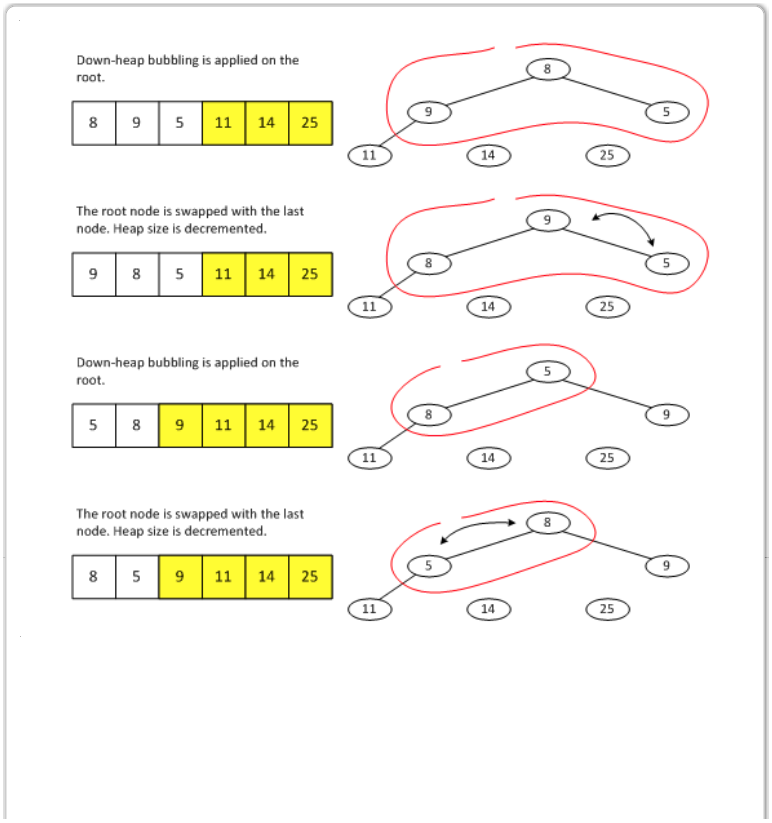


Figure 4.11. Illustration of heap-sort.



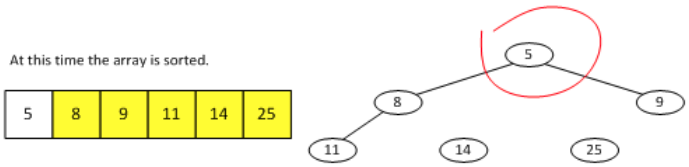


Figure 4.12. Illustration of heap-sort (continued).

Section 4.1.5. Adaptable Priority Queues

In this section, we describe a priority queue that supports additional operations. Consider the following scenarios. We assume that standby passengers are waiting for a certain flight at an airport. When a seat is available, a standby passenger is chosen based on some criteria, such as frequent-flyer status.

- A standby passenger becomes tired of waiting and tasks to be removed from the waiting list. We cannot use the *removeMin* method to remove this passenger from the priority queue because this passenger may not be the one at the top of the queue. We need a new method that removes an arbitrary entry from the queue.
- A standby passenger realizes that she has a gold frequent-flyer card, which she thought she had forgotten to bring with her. She shows it to the airline agent, and it is necessary to change her priority. For this, we need an operation that changes the key of an entry in the queue.
- A standby passenger finds that her name is misspelled on the ticket and requests that it be corrected. For this, we need an operation that changes the value of an entry in the queue.

A priority queue ADT that supports the above operations is called an *adaptable-priority queue* ADT. The adaptable priority queue supports the following methods:

- remove(e)—Removes entry e from the priority queue.
- replaceKey(e, k)—Replaces the key of existing entry e with k.
- replaceValue(e, v)—Replaces the value of existing entry e with v.

Section 4.1.5.1. Location-Aware Entries

To implement the above methods, it is necessary to find an entry in the priority queue efficiently. One way to achieve this is to keep, for each entry, one more field, which denotes the current index of the entry in the array-based heap. When an entry is relocated while a certain operation is performed, its index needs to be update to reflect the new location. Figure 4.13 illustrates the *removeMin* method and shows the changes to the indexes of entries that are relocated within the priority queue. It shows entries with the third field, which is the index of an entry, along with the change of indexes when an entry is moved. The figure also shows the heap (tree structure) to illustrate how the operation is performed.

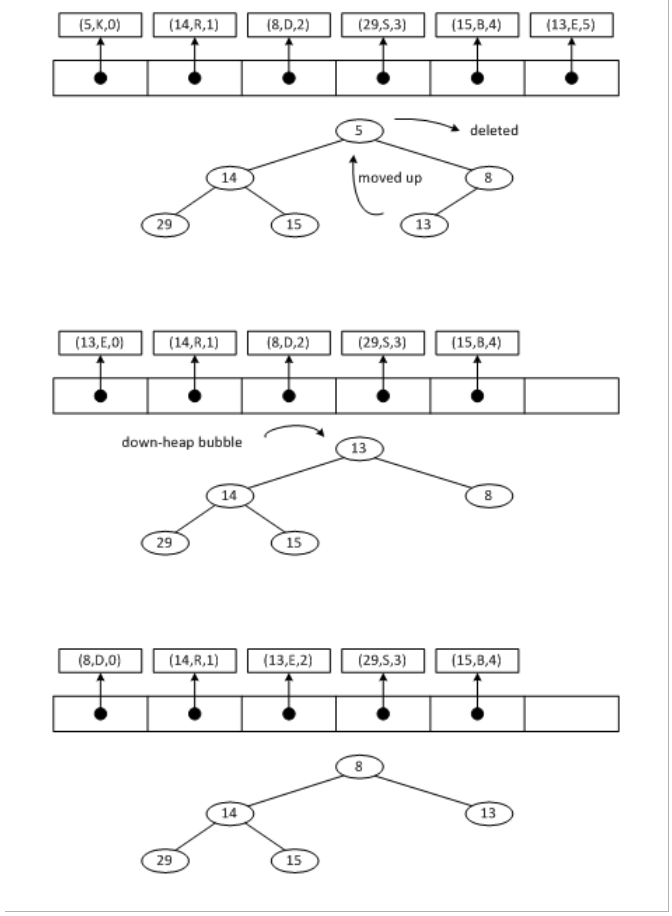


Figure 4.13 Illustration of the *removeMin* method.

Section 4.1.5.2. Implementing an Adaptable Priority Queue

In this section, we discuss a Java implementation of the adaptable–priority queue ADT. It is implemented as the *HeapAdaptablePriorityQueue* class, and it extends the *HeapPriorityQueue* class.

An entry in a queue is implemented as a nested class *AdaptablePQEntry*, of which the code is shown below:

Code Segment 4.12

```
1   protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
2     private int index;       // entry's current index within the heap
3     public AdaptablePQEntry(K key, V value, int j) {
4       super(key, value);     // this sets the key and value
5       index = j;             // this sets the new field
6     }
7     public int getIndex() { return index; }
8     public void setIndex(int j) { index = j; }
9   }
```

This class extends the *PQEntry* class, which is a nested class in the *AbstractPriorityQueue* class. It inherits the instance variables k (key) and v (value). The new field *index* is declared in line 2. The constructor in line 3 receives three arguments—*key*, *value*, and *j*—and the index of the newly created entry is set to *j* (in line 5). Lines 7 and 8 define the *get* and *set* methods, respectively.

Before we discuss three new methods, we first discuss the *bubble* and *swap* methods.

The *bubble* method is a general bubble method. It performs the up-heap bubbling if the key of the entry is smaller than the key of its parent, and performs the

down-heap bubbling if the key of the entry is larger than or equal to the key of its parent. Its Java code is shown below:

Code Segment 4.13

```
1   protected void bubble(int j) {
2     if (j > 0 && compare(heap.get(j), heap.get(parent(j))) < 0)
3       upheap(j);
4     else
5       downheap(j);         // may not need to move down
6   }
```

The *swap* method overrides the *swap* method in the *HeapPriorityQueue* class and adds the update of indexes. A Java implementation is shown below:

```
1   protected void swap(int i, int j) {
2     super.swap(i,j);                                  // perform the swap
3     ((AdaptablePQEntry) heap.get(i)).setIndex(i);     // reset entry's index
4     ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(j); // reset entry's index
5   }
```

Now we describe three new methods: *remove*, *replaceKey*, and *replaceValue*.

Removal of an entry is performed in three steps:

1. Swap the entry to be removed with the last element.
2. The last entry (which now has the entry to be removed) is removed.
3. Invoke the *bubble* method to correct a potential violation of heap-order property.

Figure 4.14 shows the removal process. The figure shows, for simplicity, only the keys in the nodes.

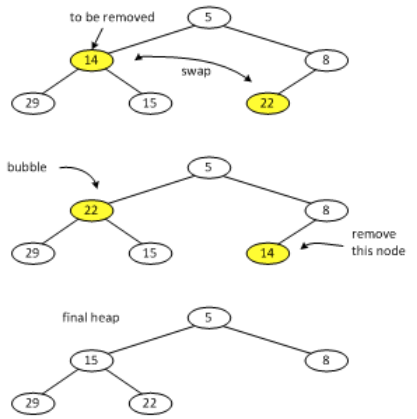


Figure 4.14. Illustration of the remove method.

A Java code is shown below:

Code Segment 4.14

```
1   public void remove(Entry<K,V> entry) throws IllegalArgumentException {
2     AdaptablePQEntry<K,V> locator = validate(entry);
3     int j = locator.getIndex();
4     if (j == heap.size() - 1)       // entry is at last position
5       heap.remove(heap.size() - 1); // so just remove it
6     else {
7       swap(j, heap.size() - 1);     // swap entry to last position
8       heap.remove(heap.size() - 1); // then remove it
```

```
9       bubble(j);                  // and fix entry displaced by the swap
10    }
11  }
```

Line 2 checks whether the given entry is valid. If the entry to be removed is the last entry in the queue, it is simply removed, in lines 4 and 5. Line 7 swaps the entry to be removed and the last entry, line 8 removes the last entry, and line 9 invokes the *bubble* method.

The following is a Java code implementing the *replaceKey* method:

Code Segment 4.15

```
1   public void replaceKey(Entry<K,V> entry, K key)
2                          throws IllegalArgumentException {
3     AdaptablePQEntry<K,V> locator = validate(entry);
4     checkKey(key);              // might throw an exception
5     locator.setKey(key);        // method inherited from PQEntry
6     bubble(locator.getIndex()); // with new key, may need to move entry
7   }
```

In lines 2 and 3, the validity of the entry and the key is checked. Then, the key of the entry is updated with the given key in line 5 using the *setKey* method, and line 6 calls the *bubble* method.

The following illustrates this method. In the figure, both the key and the element in each node are shown.

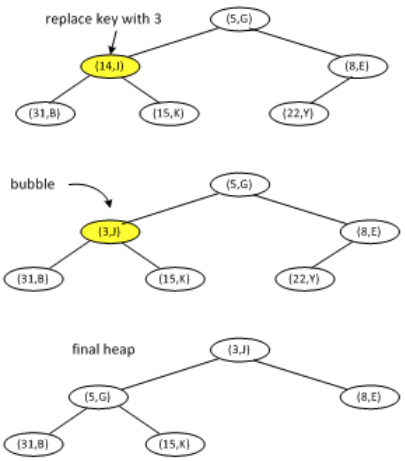


Figure 4.15. Illustration of the replaceKey method.

The *replaceValue* method is the simplest. It first ensures that the given entry is valid and then updates the value with the *setValue* method.

Code Segment 4.16

```
1   public void replaceValue(Entry<K,V> entry, V value)
2                            throws IllegalArgumentException {
3     AdaptablePQEntry<K,V> locator = validate(entry);
4     locator.setValue(value);         // method inherited from PQEntry
5   }
```

Section 4.2 Maps and Hash Tables

Section 4.2. Maps and Hash Tables

Section 4.2.1. Maps

A *map* is an abstract data type that supports the efficient storage and retrieval of entries. In a map, an entry has a (*key*, *value*) pair, and retrieval and update operations are performed based primarily on the keys. All keys in a map are unique and are called *search keys*.

Some example applications of maps follow:

- You want to store information about movies and be able to search by title. You can use the title of a movie as a *key* and other information—such as synopsis, director, and cast—as *values*.
- The parts division of an appliance manufacturer needs to store information about a large number of parts in a system. In such a system, a search is usually done by unique part number: When a user enters a part number (which is a *key*), the system maps that part number to other relevant information about the part, such as its name, where it is used, and its price (which are *values*).
- When you reserve a room at a hotel online, usually you are given a reservation number. Later, you can find information about your reservation by using that number. In this application, the reservation number is a *key*, and the information about your reservation is a *value*.

Our textbook lists other applications of maps on page 402 in Section 10.1.

Section 4.2.1.1. The Map ADT

A map is viewed as a collection of (*key*, *value*) pairs. The following are operations defined in the map ADT. Here, *M* denotes a map.

- size( )—Returns the number of entries in *M*.
- isEmpty( )—Returns *true* if *M* is empty. Returns *false* otherwise.
- get(*k*)—Returns the value *v* associated with the key *k*, if such entry exists. Returns *null* otherwise.
- put(*k*, *v*)—If there is no entry in *M* with a key equal to *k*, then adds the entry (*k*, *v*) to *M* and returns *null*. Otherwise, replaces the existing value associated with the key *k* with *v* and returns the old value.
- remove(*k*)—Removes from *M* the entry with the key *k* and returns its value. If there is no entry in *M* with the key *k*, returns *null*.
- keySet( )—Returns an iterable collection containing all keys in *M*.
- values( )—Returns an iterable collection containing all values in *M*. If multiple keys map to the same value, then the value appears multiple times in the returned collection.
- entrySet( )—Returns an iterable collection containing all (*key*, *value*) entries in *M*.

The following table shows a sequence of map operations applied to an initially empty map:

Figure 4.16. Demonstration of Map Operations

Method	Return Value	Map
put(10,A)	null	{(10,A)}
put(5,B)	null	{(10,A), (5,B)}
put(17,C)	null	{(10,A), (5,B), (17,C)}
put(17,Z)	C	{(10,A), (5,B), (17,Z)}
get(10)	A	{(10,A), (5,B), (17,Z)}
get(17)	Z	{(10,A), (5,B), (17,Z)}
get(2)	null	{(10,A), (5,B), (17,Z)}
size( )	3	{(10,A), (5,B), (17,Z)}
remove(20)	null	{(10,A), (5,B), (17,Z)}
remove(5)	B	{(10,A), (17,Z)}

isEmpty( )	false	{(10,A), (17,Z)}
keyset( )	{10, 17}	{(10,A), (17,Z)}
values( )	{A, Z}	{(10,A), (17,Z)}
entrySet( )	{(10,A), (17,Z)}	{(10,A), (17,Z)}

A Java code that implements the map ADT as an interface is shown below:

Code Segment 4.17

```
1   public interface Map<K,V> {
2     int size();
3     boolean isEmpty();
4     V get(K key);
5     V put(K key, V value);
6     V remove(K key);
7     Iterable<K> keySet();
8     Iterable<V> values();
9     Iterable<Entry<K,V>> entrySet();
10  }
```

Java provides the *java.util.Map* interface, which furnishes a more extensive set of operations than those defined above.

Section 4.2.1.2. Application: Word Frequencies

In this section, we describe a problem that can be solved easily using a map: Count the number of times each distinct word appears in a given text document.

First, we create an empty map in which a word is used as a *key*, and the frequency of that word is used as a *value*. Then, we scan the document and, for each word *w*, we perform the following:

- If *w* is not in the map, we add (*w*, 1) to the map.
- If *w* is already in the map, we increment the frequency of that word in the map.

A Java implementation is shown below:

Code Segment 4.18

```
1   public class WordCount {
2     public static void main(String[] args) {
3       Map<String,Integer> freq = new ChainHashMap<>();  // or any concrete map
4       // scan input for words, using all nonletters as delimiters
5       Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
6       while (doc.hasNext()) {
7         String word = doc.next().toLowerCase();// convert next word to lowercase
8         Integer count = freq.get(word);  // get the current count for this word
9         if (count == null)
10          count = 0;                 // if not in map, previous count is zero
11        freq.put(word, 1 + count);    // (re)assign new count for this word
12      }
13      int maxCount = 0;
14      String maxWord = "no word";
15      for (Entry<String,Integer> ent : freq.entrySet()) // find max-count word
16        if (ent.getValue() > maxCount) {
17          maxWord = ent.getKey();
18          maxCount = ent.getValue();
```

```
19        }
20      System.out.print("The most frequent word is '" + maxWord);
21      System.out.println("' with " + maxCount + " occurrences.");
22    }
23  }
```

In line 3, an empty map is created. Note that the *ChainHashMap* class, which is used in creating the map, is not discussed yet. It will be discussed in Section 4.2.2.4. The *while* loop of lines 6 through 12 scans the document and processes one word at a time. In line 8, the current count of the word being processed is retrieved. If a word is not in the map—which is tested with *count* == *null* in line 9—then the initial count is set to zero in line 10. Then, in line 11, the entry (word, count+1) is added to the map. The remaining part of the code finds the word with the highest frequency.

Section 4.2.1.3. Abstract Map Base Class

In this section and the sections that follow, we describe different implementations of the map ADT using various data structures. To maximize code reuse, many classes and interfaces are designed to support various implementations. The hierarchy of interfaces and classes is shown below.

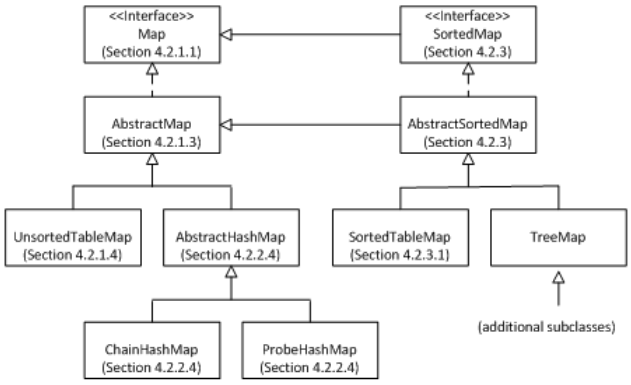


Figure 4.17. Inheritance hierarchy of map-related classes and interfaces.

In this section, we design an abstract base class *AbstractMap*, which is a basis for all map implementations described in Section 4.2.

The *AbstractMap* class provides the following supports, which can be used in its concrete subclasses:

- *isEmpty* method—This method assumes the implementation of the *size*( ) method.
- *MapEntry* nested class—The nested class stores a (*key*, *value*) pair and implements the *Entry* interface. It also defines the *get* and *set* methods.

Code Segment 4.19

```
1   protected static class MapEntry<K,V> implements Entry<K,V> {
2     private K k;  // key
3     private V v;  // value
4     public MapEntry(K key, V value) {
5       k = key;
6       v = value;
7     }
8     public K getKey() { return k; }
9     public V getValue() { return v; }
10    protected void setKey(K key) { k = key; }
11    protected V setValue(V value) {
12      V old = v;
13      v = value;
14      return old;
```

```
15    }
16    public String toString() {return "<" + k + ", " + v +">";}
17  }
```

- *keySet* method—This method returns the iterable collection of values. It involves two nested classes— *KeyIterable* and *KeyIterator*. The *KeyIterator* class returns an iterator to the *KeyIterable* class, which, in turn, returns an iterable to the *keySet* method.

Code Segment 4.20

```
1   private class KeyIterator implements Iterator<K> {
2     private Iterator<Entry<K,V>> entries = entrySet().iterator();
3     public boolean hasNext() {return entries.hasNext();}
4     public K next() {return entries.next().getKey();}    // return key!
5     public void remove()
          {throw new UnsupportedOperationException("remove not supported");}
6   } //----------- end of nested KeyIterator class -----------

7   private class KeyIterable implements Iterable<K> {
8     public Iterator<K> iterator() {return new KeyIterator();}
9   } //----------- end of nested KeyIterable class -----------

10  public Iterable<K> keySet() {return new KeyIterable();}
```

- *values* method—Returns the iterable collection of values. It also involves two nested classes—*ValueIterable* and *ValueIterator*. The *ValueIterator* class returns an iterator to the *ValueIterable* class, which, in turn, returns an iterable to the keySet method.

Code Segment 4.21

```
1   private class ValueIterator implements Iterator<V> {
2     private Iterator<Entry<K,V>> entries = entrySet().iterator();
3     public boolean hasNext() {return entries.hasNext();}
4     public V next() {return entries.next().getValue();} // return value!
5     public void remove()
6       {throw new UnsupportedOperationException("remove not supported");}
7   } //----------- end of nested ValueIterator class -----------

8   private class ValueIterable implements Iterable<V> {
9     public Iterator<V> iterator() {return new ValueIterator();}
10   } //----------- end of nested ValueIterable class -----------

12   public Iterable<V> values() {return new ValueIterable();}
13  }
```

A complete code of the *AbstractMap* can be found at the *AbstractMap.java* file.

Section 4.2.1.4. Simple Unsorted-Map Implementation

As an example of a simple, concrete implementation of the *AbstractMap* base class, we describe a simple, unsorted-map implementation, which is based on Java's *ArrayList*. It is implemented as the *UnsortedTableMap* class.

It extends the *AbstractMap* class and has one instance variable, *table*. The *table* is a map that is implemented using an *ArrayList.*

The three basic methods—*get*(*k*), *put*(*k*, *v*), and *remove*(*k*) methods—need to scan the array to check whether an entry with a key equal to *k* exists. So, a utility method *findIndex*(*key*) is defined to perform that function, and it is shared by these three methods. A Java code is shown below:

Code Segment 4.22

```
1   private int findIndex(K key) {
2     int n = table.size();
3     for (int j=0; j < n; j++)
4       if (table.get(j).getKey().equals(key))
5         return j;
6     return -1; // special value denotes that key was not found
7   }
```

In the code, the *for* loop of lines 3 to 5 scans the array to find an entry with the given key. If no such entry exists, a special value "-1" is returned.

The *get*(*key*) method invokes the *findIndex* method and returns the associated value if an entry with the given key exists. A Java code is shown below:

Code Segment 4.23

```
1   public V get(K key) {
2     int j = findIndex(key);
3     if (j == -1) return null;  // not found
4     return table.get(j).getValue();
5   }
```

The *put*(*key*, *value*) method adds a new entry containing the given *(key, value)* pair, if there is no entry with the given key. If there exists an entry with the given key, then its value is replaced with the given value. A Java implementation is shown below:

Code Segment 4.24

```
1   public V put(K key, V value) {
2     int j = findIndex(key);
3     if (j == -1) {
4       table.add(new MapEntry<>(key, value)); // add new entry
5       return null;
6     } else                                  // key already exists
7       return table.get(j).setValue(value); // replaced value is returned
8   }
```

The *remove*(*key*) method removes the entry with the specified key, if it exists. Let e denote the entry with the given key. First, e is replaced with the last entry in the array. Then the last entry is removed from the array. A Java code is shown below:

Code Segment 4.25

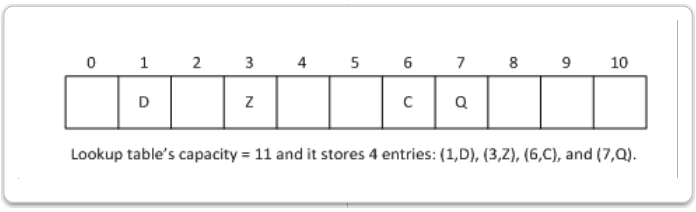
```
1   public V remove(K key) {
2     int j = findIndex(key);
3     int n = size();
4     if (j == -1) return null;       // not found
5     V answer = table.get(j).getValue();
6     if (j != n - 1)
7       table.set(j, table.get(n-1)); // move last entry to 'hole' created by removal
8     table.remove(n-1);              // remove last entry of table
9     return answer;
10  }
```

A complete code of the *UnsortedTableMap* class can be found at the *UnsortedTableMap.java* file.

Section 4.2.2. Hash Tables

A *hash table* is a data structure that implements a map. It provides efficient search and update operations.

Consider a lookup table that is implemented as an array-based map with the capacity N. Suppose that we want to store n entries, each containing a *(key, value)* pair, and that the keys are integers. Then we can use the keys as the indexes of corresponding entries in the array. The following figure shows an example of such a map, where $N = 11$ and $n = 4$:



On this lookup table, we can implement the basic map operations *get*, *put*, and *remove* in $O(1)$ worst-case time.

There are some problems with this approach. First, if the keys are not consecutive numbers, then there will be many unused spaces in the array. Second, if we do not know the minimum and maximum of the keys (in other words, if we do not know n, then we have to estimate the size of the array N, and we may under- or an overestimate it. Third, even though we know n, if $N > n$, then there will be many unused spaces. Finally, keys may not be integers.

A *hash table* solves these problems. A hash table uses a hash function to map a key to an integer, which is a valid array index. More specifically, a hash function h is a mapping from a set of keys to $\{0, 1, ..., N - 1\}$, where N is the capacity of an array. Given a $(key\ value)$ pair, we apply a hash function h to key, and we store $value$ in the array at index $h(key)$.

As an example, let's assume the following:

- We have a hash table of size $N = 11$. Note that the hash table is an array.
- Hash function $h = key$ mod 11, where *mod* is a modulo operator.
- We want to store the following entries: (1, X), (4, K), (8, Y), (10, E), (12, M), (19, A), (41, G), and (54, H).

To determine the location in the array where each entry is to be stored, we first apply the hash function to each key:

- h(1) = 1 mod 11 = 1
- h(4) = 4 mod 11 = 4
- h(8) = 8 mod 11 = 8
- h(10) = 10 mod 11 = 10
- h(12) = 12 mod 11 = 1
- h(19) = 19 mod 11 = 8
- h(41) = 41 mod 11 = 8
- h(54) = 54 mod 11 = 10

As we can see, some multiple keys are mapped to the same array index. For example, keys 8, 19, and 41 are all mapped to index 8. This is called *collision*, and we need to resolve collisions in a certain way. We will discuss collision resolution later. For now, we assume that each array slot has a bucket and if multiple keys are mapped to the same slot, we keep those entries in the bucket. The following figure shows how the above entries are stored in a hash table with buckets. Note that, to make the figure less cluttered, empty buckets are not shown.

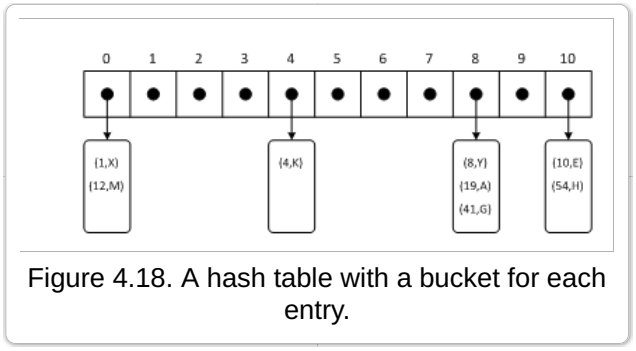


Figure 4.18. A hash table with a bucket for each entry.

In what follows, we use the term *bucket* to refer to a slot in the array. So, the terms *bucket* and *array slot* are used interchangeably.

Section 4.2.2.1 Hash Functions

As mentioned in the previous section, a hash function h is a mapping from a set of keys to an integer in the range $[0, N-1]$:

$$h: U \rightarrow \{0, 1, ..., N-1\}$$

Here, N is the capacity of the hash table (or the underlying array), and U is a set of keys.

Let A be the bucket array, which was shown in the previous section. Then an entry with the key k will be stored in the bucket $A[h(k)]$.

As discussed earlier, when two keys are mapped to the same array slot, a collision occurs. Such collisions can be resolved but collision resolutions incur overhead. So, a good hash function must ideally distribute keys evenly across the hash table. In practice, this is not achievable, and we use heuristic hash functions. Another important feature of a hash function is efficiency. A hash function should be able to perform the mapping fast.

In general, when keys are arbitrary objects, a hash function consist of two steps: a *hash code* that maps a key k to an integer, and a *compression function* that maps that hash code to an integer in the range $[0, N-1]$. Figure 4.19 illustrates these two steps:

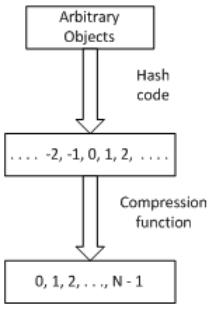


Figure 4.19. Two steps of a hash function.

Hash Codes

The *hash code* takes a key k of arbitrary type and maps it to an integer. The result does not have to be in the range $[0, N-1]$ and can be a negative number. A hash code should avoid collisions as much as possible. If two keys are mapped to the same integer, then they will certainly be mapped to the same hash-table index by a compression function. Note that the integer generated by hash code mapping of a key k is also called a *hash code* for k.

Java programming language uses 32-bit hash codes. If the data type of keys is a base type—such as *byte*, *short*, *int*, or *char*—a hash code can be created by simply casting the type to *int*.

If the representation of the type of keys requires more than 32 bits, or the keys are arbitrary objects, then usually some mathematical method is employed as a hash code. In our textbook, some mathematical methods are discussed on pages 412 to 414.

In Java, the *Object* class has a default *hashCode*( ) method that returns a 32-bit integer hash code of *int* type. This default *hashCode* method returns the memory address of the object.

When you define your own class and you override and define your own *equals* method, then you also need to override the default *hashCode* method. When overriding the *equals* and *hashCode* methods, you have to make sure that two "equal" object (as determined by the *equals* method) produce the same hash code.

Compression

After a key is converted to an integer hash code, a *compression function* maps the hash code to an integer in the range $[0, N-1]$ to fit in the hash-table index range. To minimize collisions, a good compression function must distribute hash codes (of keys) relatively uniformly across the hash table. We briefly describe two compression functions.

A simple and popular compression function is the *division method*. It uses the modulo operator and is defined as follows:

$$i \mod N$$

Here, i is an integer (such as an integer hash code) and N is the size of the hash table. When we use the *division* method, we choose the value of N carefully. If we choose a prime number for N, the possibility of collisions is smaller. If N is not a prime number, then the possibility is larger. Note that choosing

a prime number for N does not guarantee that there will be no collisions; it simply reduces their probability. A prime number that is not too close to an exact power of two often works well.

Another method, which spreads integer keys reasonable well, is the *Multiply-Add-and-Divide* (or *MAD*) method. It is defined as follows:

$$[(ai + b) \mod p] \mod N$$

Here, p is a prime number larger than N, and a and b are integers randomly chosen from $[0, N-1]$, with $a > 0$.

The *MAD* method is better than the *division* method. However, the *division* method is more efficient, for it needs to perform one simple operation, whereas the *MAD* method requires multiple operations.

Section 4.2.2.2. Collision-Handling Schemes

In this section, we discuss two collision-resolution methods: *chaining* and *open addressing*.

Chaining Method

The chaining method, also referred to as *separate chaining*, is an implementation of the array-with-bucket approach described in Section 4.2.2. We keep an unsorted list for each array slot to store entries that are mapped to the same array index. The separate-chaining method is illustrated below:

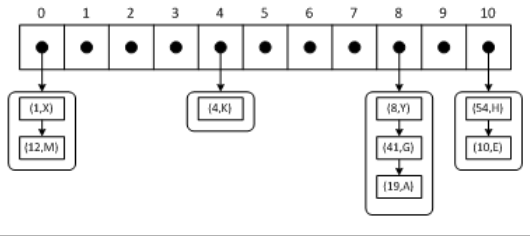


Figure 4.20. Collision resolution with chaining method.

An advantage of the chaining method is that it is easy to implement. However, a drawback is that it requires additional storage space to keep the lists.

Open-Addressing Method

In *open addressing*, all entries are stored in a hash table itself. So, no additional data structure or storage space is needed. One implication of this is that the capacity of the hash table, N, must be large enough to store all elements within itself.

We briefly discuss three open-addressing techniques: *linear probing*, *quadratic probing*, and *double hashing*.

The linear probing is the simplest of the three. Let A be the array of a hash table. Suppose that we are inserting an entry (k, v) into the hash table, and k is mapped to the array index j—i.e., $h(k) = j$. If $A[j]$ is empty, then the entry is stored in that slot. If, however, that slot is already occupied by another entry, then the next bucket, $A[j+1]$, is *probed* to see whether it is available. If it is empty, the entry is stored there. Otherwise, the next bucket, $A[j+2]$, is probed, and so on, until an empty slot is found or all slots have been probed. Note that if we reach the end of the array while probing next slots, we go back to the beginning of the array and continue. So, more accurately, we probe $A[(j+1) \mod N]$, $A[(j+2) \mod N]$, and so on. In general, the sequence of array slots probed—or a *probe sequence*—is determined by $A[(j+1) \mod N]$, for $i = 0, 1, 2, \ldots, N-1$. The "i" is called a *probe number*. The following figure illustrates the linear-probing method:

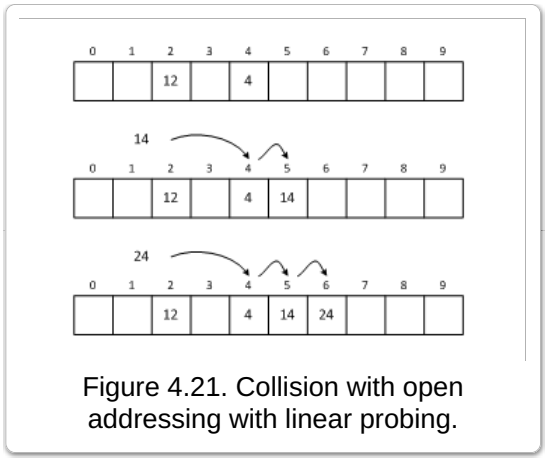


Figure 4.21. Collision with open addressing with linear probing.

In the figure, the hash function used is $h = k \mod N$, where $N = 10$. For simplicity, only keys are shown in the array. Keys are inserted in the following order: 4, 12, 14, and 24.

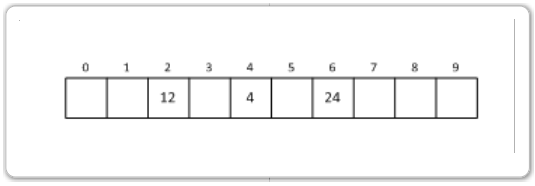
First, 4 is inserted in $A[h(4)] = A[4]$. Next, 12 is inserted in $A[h(12)] = A[2]$. The next key, 14, is mapped to $A[4]$, which is occupied. So, the next slot, $A[5]$, is probed. Since it is empty, 14 is placed in $A[5]$. The key 24 is also mapped to $A[4]$, which is occupied. The next bucket, $A[5]$, is also occupied. Since $A[6]$ is empty, it is inserted here.

When we search the hash table for an entry (k, v), we first map the given key to get the array index—j, for example. If we find an entry with the key k in that slot, the search succeeds. If the slot is occupied, then we probe the consecutive slots. There are three cases:

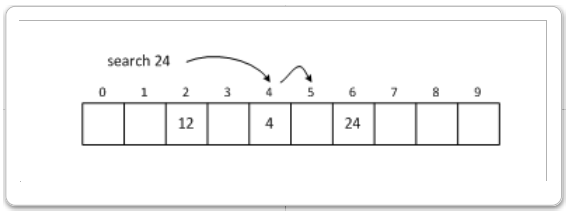
- We find an entry with the key. This is a successful search.
- We encounter an empty slot. The entry we are looking for is not in the hash table. This is a failed search.
- We have probed all slots in the array. The entry we are looking for is not in the hash table. This is a failed search.

If we search the hash table for 24, we will follow the same sequence of probes as when we inserted it, and we will find it in $A[6]$. Suppose we search for 34. It is also mapped to $A[4]$, and we will follow the same sequence of probes: $A[4]$, $A[5]$, and $A[6]$. They are all occupied, so the next bucket, $A[7]$, is probed. Since it is empty, we conclude that there is no entry with key 34.

Deleting an entry from the hash table is not straightforward. Consider the following scenario. We insert keys 4, 12, 14, and 24, in that order. Later, we delete the entry with key 14. The resulting array is shown below:

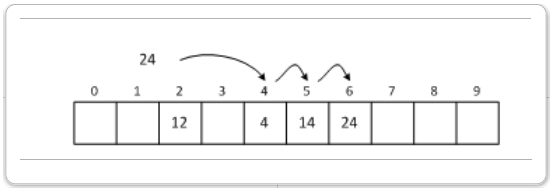


Now, assume we search for 24. It is mapped to $A[4]$, which we find occupied. So, we probe $A[5]$, and it is empty. We conclude that 24 is not in the hash table. But actually, it is in the hash table in $A[6]$. This is illustrated in the following figure:



One solution to this problem is to put a "special object" in the slot from which an entry is deleted. For example, we may place the symbol ϕ in that slot. When we insert an entry, the special symbol is considered an empty slot. When we perform a search, a slot with the special symbol is considered to contain a key, which is different from the key that is being searched for.

The linear probing has one issue, which is referred to as *primary clustering*. A *cluster* is a consecutive array of slots that are filled with entries that experienced collisions. For example, in the following figure, the buckets $A[4]$, $A[5]$, and $A[6]$ form a cluster. In the linear probing, once a cluster is formed, it tends to grow (because it increases the probability of subsequent collisions).



Another open-addressing method, called *quadratic probing*, tries to alleviate the primary clustering problem. When there is a collision, the next slot to be probed is not the slot immediately following the current slot in the array, as it would be with linear probing; instead, it is determined by a quadratic function of a probe number. For example, we may probe slots in accordance with the following expression: $A[(h(k) + f(i) \mod N]$, for $i = 0, 1, 2, ..., N - 1$, where $f(i) = i^2$. Suppose that we are inserting a key 24, which is mapped to $A[4]$, and suppose that it is occupied. The linear probing will probe subsequent buckets $A[5]$, $A[6]$, $A[7]$, and so on. However, the quadratic probing, as defined above, probes the following:

$A[(4 + 1^2) \mod 10] = A[5]$,
$A[(4 + 2^2) \mod 10] = A[8]$,
$A[(4 + 3^2) \mod 10] = A[3]$,
...

The quadratic probing does not have the primary-clustering issue. However, it has its own clustering problem, called *secondary clustering*.

The third type of open addressing is *double hashing*. The double hashing method does not suffer from the clustering problems of linear and quadratic probing. It uses a secondary hash function h'. The probe sequence is determined as follows:

$A[(h(k) + i \cdot h'(k)) \mod N]$, for $i = 0, 1, 2, ..., N - 1$

One common secondary hash function is $h'(k) = q - (k \mod q)$, for some prime number $q < N$. Another popular secondary hash function is $h'(k) = 1 + (k \mod N')$, where N' is a prime number that is two smaller than N. In both cases, N is a prime number.

Section 4.2.2.3. Load Factors, Rehashing, and Efficiency

One important parameter of a hash table is a *load factor*, defined as follows:

$\lambda = n / N$

A larger value of λ means there is higher probability of collisions. Therefore, the smaller λ is, the better. With the chaining method, λ could be larger than 1. However, it is desirable to keep λ below 1. A theoretical analysis shows that the average number of slots that need to be probed for a successful search is approximately $1 + \frac{\lambda}{2}$.

Let C be the average number of slots that need to be probed for a successful search. The following shows the values of *C* for some instances of λ:

λ	C
0.5	1.25
0.7	1.35
1.0	1.5
2.0	2

In Java, which uses the chaining method for hash tables, λ is set to 0.75 or less by default.

In open addressing, λ cannot be larger than 1.0. According to a theoretical analysis, the average number of slots to be probed for a successful search, C, is approximately $\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$. Usually, we keep λ below 0.5. The following shows the values of *C* for some λ:

λ	C
0.3	1.19

0.5	1.39
0.7	1.72
0.9	2.56

Suppose that you want to keep λ below a certain threshold. When there are many insertions, then λ may go above the specified threshold. Then the size of the hash table needs to be increased. If this happens, then all entries in the hash table need to be *rehashed* into the new, resized hash table. When resizing a hash table, a prime number that is approximately double the previous size is usually chosen as the size of the new hash table.

The following table compares the running times of the unsorted-list and hash-table implementations of a map for different methods:

Figure 4.22. Performance of Both a Map Implemented Using an Unsorted List and a Map Implemented Using a Hash Table

Method	Unsorted List	Hash Table	
		Expected	Worst Case
get	$O(n)$	$O(1)$	$O(n)$
put	$O(n)$	$O(1)$	$O(n)$
remove	$O(n)$	$O(1)$	$O(n)$
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
entrySet, keySet, values	$O(n)$	$O(1)$	$O(n)$

Section 4.2.2.4. Java Hash-Table Implementation

In this section, we describe how to implement a hash table using the chaining method and using open addressing with linear probing. These two collision-resolution methods reflect completely different approaches. However, some functionalities are common to both methods. We first develop an abstract base class *AbstractHashMap*, which will provide such common functionalities.

Abstract Base Class

The abstract class does not define functionalities that are dependent on implementation details. For example, the chaining method needs a separate data structure to implement a *bucket* for each array slot, while the open-addressing methods do not need such additional data structures. Those methods that depend on such implementation details are declared only as abstract methods, and actual implementations are left to the concrete subclasses of the abstract class. In the *AbstractHashMap* class, which serves as an abstract base class for *ChainHashMap* and *ProbeHashMap* classes, the following methods are declared as abstract methods:

- createTable( )—Creates an initially empty hash table.
- bucketGet(h, k)—Returns the value of the entry with the key k, which is initially mapped to bucket h.
- bucketPut(h, k, v)—Inserts the entry with (k, v) to the hash table, which is initially mapped to bucket h.
- bucketRemove(h, k)—Removes the entry with key k, which is initially mapped to bucket h.
- entrySet( )—Returns the iterable collection of all entries in the hash table.

The *AbstractHashMap* class extends the *AbstractMap* class (described in Section 4.2.1.3). It has the following instance variables:

```
protected int n = 0;       // number of entries in the dictionary
protected int capacity;    // length of the table
private int prime;         // prime factor
private long scale, shift; // the shift and scaling factors
```

The *AbstractHashMap* class implements the *MAD* compression function, which is defined as [(*ai* + *b*) mod *p*] mod *N* (refer to Section 4.2.2.1). The instance variables *scale*, *shift, prime*, and *capacity* correspond to *a*, *b*, *p*, and *N*, respectively, in the definition. The prime factor, *prime*, is also used when generating *scale*

and *shift*.

The *AbstractHashMap* class implements the following methods: *size*, *get*, *remove*, *put*, *hashValue*, and *resize*. We briefly discuss the *put*, *hashValue*, and *resize* methods.

The *put* method adds the given entry by invoking the *bucketPut* method. Then, if the number of elements becomes the half of the hash table's capacity (i.e., the load factor goes above 0.5), the capacity is doubled by calling the *resize* method. A Java code is shown below:

Code Segment 4.26

```
1   public V put(K key, V value) {
2     V answer = bucketPut(hashValue(key), key, value);
3     if (n > capacity / 2)       // keep load factor <= 0.5
4       resize(2 * capacity - 1); // (or find a nearby prime)
5     return answer;
6   }
```

The *hashValue* method maps the given key to the index of the bucket in the hash table. It first obtains the key's hash code by invoking the *hashCode*( ) method. Then, it converts the hash code to the bucket index using the *MAD* compression function, and returns the index. A Java implementation is given below:

Code Segment 4.27

```
1   private int hashValue(K key) {
2     return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
3   }
```

The *resize* method doubles the capacity of the hash table, the Java code of which is shown below:

Code Segment 4.28

```
1   private void resize(int newCap) {
2     ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
3     for (Entry<K,V> e : entrySet())
4       buffer.add(e);
5     capacity = newCap;
6     createTable();  // based on updated capacity
7     n = 0;          // will be recomputed while reinserting entries
8     for (Entry<K,V> e : buffer)
9       put(e.getKey(), e.getValue());
10  }
```

In lines 2 through 4, all entries in the hash table are temporarily copied into *buffer*, which is an array list, before the capacity is increased in line 5. Line 6 creates a new hash table with the increased capacity. The *for* loop in lines 8 and 9 inserts all entries in *buffer* into the new hash table.

A complete code of *AbstractHashMap* can be found at the *AbstractHashMap.java* file.

Implementation with Chaining

In this section, we discuss how to implement a hash table using the chaining method. It is implemented as the *ChainHashMap* class. In the *ChainHashMap* class, in which a bucket of a hash table is implemented using the *UnsortedTableMap* class (discussed in 4.2.1.4). So, we are using a simpler, secondary map to implement more complex map. In general, this type of approach—using a simpler solution to solve a more complex problem—is known as *bootstrapping*.

The whole hash table is implemented as an array *table* of secondary maps, which are unsorted table maps. Each bucket *table*[h] is either *null*, if there are no entries in that bucket, or the reference to a secondary map. It is declared as an instance variable, as follows:

```
private UnsortedTableMap<K,V>[] table;   // initialized within createTable
```

The *ChainHashMap* class extends the *AbstractHashMap* class and provides concrete implementations for the five abstract methods declared in the *AbstractHashMap* class. The *bucketPut* method relies on the *put* method of the *UnsortedTableMap* class, and the *bucketGet* method relies on the *get* method of

the *UnsortedTableMap* class.

We discuss those five methods below.

The *createTable* method creates a new *table* as an array of the *UnsortedTableMap* class:

Code Segment 4.29

```
1   protected void createTable() {
2     table = (UnsortedTableMap<K,V>[]) new UnsortedTableMap[capacity];
3   }
```

The *bucketGet* method invokes the *get* method of the *UnsortedTableMap* class. It returns the value of the entry with the given key. If the bucket, to which the key is mapped is empty, or an entry with the key does not exists in the bucket, it returns *null*.

Code Segment 4.30

```
1   protected V bucketGet(int h, K k) {
2     UnsortedTableMap<K,V> bucket = table[h];
3     if (bucket == null) return null;
4     return bucket.get(k);
5   }
```

The *bucketPut* method inserts an entry with the given *(k, v)* pair to the table. A Java code is shown below:

Code Segment 4.31

```
1   protected V bucketPut(int h, K k, V v) {
2     UnsortedTableMap<K,V> bucket = table[h];
3     if (bucket == null)
4       bucket = table[h] = new UnsortedTableMap<>();
5     int oldSize = bucket.size();
6     V answer = bucket.put(k,v);
7     n += (bucket.size() - oldSize); // size may have increased
8     return answer;
9   }
```

In line 3, it checks whether the bucket, to which the key is mapped is empty or not. If it is empty, a new bucket (an unsorted table map) is created and assigned to *table*[h]. In line 6, a new entry with (k, v) is added to the bucket. Line 7 increases the number of entries n, but only if a new entry is added to the hash table. (Recall that the *put* method of the *UnsortedTableMap* class adds a new entry only if there is no entry with the given key. Otherwise, it replaces the existing value with the new value. In the latter case, there is no change in the number of entries.)

The following is a Java code of the *bucketRemove*method.

Code Segment 4.32

```
1   protected V bucketRemove(int h, K k) {
2     UnsortedTableMap<K,V> bucket = table[h];
3     if (bucket == null) return null;
4     int oldSize = bucket.size();
5     V answer = bucket.remove(k);
6     n -= (oldSize - bucket.size()); // size may have decreased
7     return answer;
8   }
```

If the bucket in the table at index *h* is empty, *null* is returned in line 3. Otherwise, the *remove* method of the *UnsortedTableMap* class is invoked in line 5. The *remove* method returns the value of the entry with the key *k*. If such an entry does not exists in the bucket, it returns *null*. As was done in the *bucketPut* method, the number of entries is decreased only if an entry was removed, in line 6.

The *entrySet* method returns the iterable collection of all entries in the hash table. A Java code is shown below:

Code Segment 4.33

```
1   public Iterable<Entry<K,V>> entrySet() {
2     ArrayList<Entry<K,V>> buffer = new ArrayList<>();
3     for (int h=0; h < capacity; h++)
4       if (table[h] != null)
5         for (Entry<K,V> entry : table[h].entrySet())
6           buffer.add(entry);
7     return buffer;
8   }
```

An empty *ArrayList* object *buffer* is created in line 2. The outer *for* loop (of line 3) scans all buckets in the table. If a slot is not empty, then the inner *for* loop (of line 5) retrieves all entries in the bucket and adds them to the *buffer*. Note that, in line 5, the *entrySet* method of the *UnsortedTableMap* class is used to get all entries from a bucket. Then the *buffer* is returned in line 7.

A complete code of the *ChainHashMap* class can be found at the *ChainHashMap.java* file.

Implementation Using Open Addressing with Linear Probing

We describe how to implement a hash table using open addressing with linear probing. It is implemented as the *ProbeHashMap* class.

As discussed in Section 4.2.2.2, when an entry is deleted, we need to put a "special object" in the array slot from which the entry was removed. In this implementation, a sentinel object called *DEFUNCT* is used for that purpose. The *DEFUNCT* is a *MapEntry* object with *null* for both key and value.

When we search a hash table for an entry with a key k, first we compute the hash value, h, of the key k. If the bucket h is occupied by an entry with a different key, then we need to start the probing process. If a *DEFUNCT* is found in a slot during probing, the slot is considered to be occupied with an entry with a different key, and the probing must continue. Since the removal of an entry requires the search for the entry to be removed, the same steps are taken first.

When we insert a new entry into the hash table, we first need to check whether there already is an entry with the given key. So, we need to perform the search operation. Again, the probing must continue beyond the *DEFUNCT*. If there is no entry with the given key in the hash table, then the new entry is added to the first slot with *DEFUNCT*.

Since all three operations—search, insert, and remove—may require probing, a separate utility method *findSlot* is implemented to perform probing.

We first describe the *findSlot* method and then other methods.

The *findSlot* method receives two arguments: h and k. The first argument, h, is the hash code of the key k, which is the index of the bucket to which k is initially mapped. The second argument, k, is the key. The value returned by the *findSlot* method is determined as follows:

- A slot is said to be "available" if it contains *null* or *DEFUNCT*.
- If an entry with the key k exists in the hash table, the index of the slot is returned. The return value is always greater than or equal to zero.
- If an entry with the key k does not exist in the hash table, the "negative version" of the index of the first available slot is returned. Let *avail* be the index of the first available slot. The method first converts it to—(avail + 1) and returns the "negative version." The return value is always a negative number.

A Java implementation of the *findSlot* method is shown below:

Code Segment 4.34

```
1   private int findSlot(int h, K k) {
2     int avail = -1;                 // no slot available (thus far)
3     int j = h;                      // index while scanning table
4     do {
5       if (isAvailable(j)) {         // may be either empty or defunct
6         if (avail == -1) avail = j;  // this is the first available slot!
7         if (table[j] == null) break; // if empty, search fails immediately
8       }
9       else if (table[j].getKey().equals(k))
10        return j;                    // successful match
11      j = (j+1) % capacity;          // keep looking (cyclically)
```

```
12    } while (j != h);                // stop if we return to the start
13    return -(avail + 1);             // search has failed
14  }
```

The *if* statement of line 5 checks whether the slot being probed is available or not. If it is available and this is the first such slot, the variable *avail* is set to the index of that slot. If the slot has *null*, this means there is no entry with the given key. Then, we exit the *do-while* loop and the "negative version" of the index is returned in line 13. If the slot has *DEFUNCT*, we jump to line 11 and continue the search. If the slot is not available, then the given key is compared with the key of the entry in that slot (line 9). If they are identical, the entry is found, and the index of the slot is returned (line 10). Otherwise, we continue to probe the next slot, in line 11.

The *bucketGet* method first calls the *findSlot* method. If the value returned by the *findSlot* method is a negative number, then, since there is no entry with the given key, *null* is returned. Othewise, we retrieve the value of the entry using the index of the slot, which was returned by the *findSlot* method. A Java code is shown below:

Code Segment 4.35

```
1   protected V bucketGet(int h, K k) {
2     int j = findSlot(h, k);
3     if (j < 0) return null; // no match found
4     return table[j].getValue();
5   }
```

A Java code of the *bucketPut* method is shown below:

Code Segment 4.36

```
1   protected V bucketPut(int h, K k, V v) {
2     int j = findSlot(h, k);
3     if (j >= 0)                  // this key has an existing entry
4       return table[j].setValue(v);
5     table[-(j+1)] = new MapEntry<>(k, v); // convert to proper index
6     n++;
7     return null;
8   }
```

The *bucketPut* method calls the *findSlot* method first in line 2. If the returned value is a non-negative number, this means there is an entry with the given key. In this case, the value of the entry is replaced with the given value and the old value is returned in line 4. If the return value is a negative number, this means there is no entry with the given key, and the new entry with the given (k, v) is added to the slot using the index returned by the *findSlot* method in line 5. Note that, the returned index is the "negative version" of the real index. So, it is converted to the original index in line 5.

The following is a Java code of the *bucketRemove* method:

Code Segment 4.37

```
1   protected V bucketRemove(int h, K k) {
2     int j = findSlot(h, k);
3     if (j < 0) return null;   // nothing to remove
4     V answer = table[j].getValue();
5     table[j] = DEFUNCT;  // mark this slot as deactivated
6     n--;
7     return answer;
8   }
```

If the *findSlot* method returns a negative number, *null* is returned in line 3. Otherwise, the entry is removed in line 4, and *DEFUNCT* is put in that slot in line 5.

Test Yourself 4.2

Assume that you designed a hash table using the *division* function, which is defined as

$h(i) = i \mod N$, where $N = 12$.

Also assume that you use the open addressing with linear probing to resolve collisions.

Show the final contents of the hash table after the following sequence of keys are mapped to the hash table, in the given order.

<19, 14, 67, 36, 53, 84, 50, 15, 33, 32>

Please think carefully, write your answer, and then click "Show Answer" to compare yours to the suggested answer.

Suggested answer:

Section 4.2.3. Sorted Map

The traditional map ADT, which we discussed, performs an *exact search*. However, some applications require searches with more flexible search conditions. In this section, we describe an ADT, called the *Sorted Map ADT*, which includes all behaviors of the traditional map plus the following methods:

- firstEntry( )—Returns the entry with the smallest key. Returns *null* if the map is empty.
- lastEntry( )—Returns the entry with the largest key. Returns *null* if the map is empty.
- ceilingEntry(*k*)—Returns the entry with the smallest key that is larger than or equal to *k*. Returns *null* if no such entry exists.
- floorEntry(*k*)—Returns the entry with the largest key that is smaller than or equal to *k*. Returns *null* if no such entry exists.
- lowerEntry(*k*)—Returns the entry with the largest key that is strictly smaller than *k*. Returns *null* if no such entry exists.
- higherEntry(*k*)—Returns the entry with the smallest key that is strictly larger than *k*. Returns *null* if no such entry exists.
- subMap(k_1, k_2)—Returns an iteration of all entries with keys larger than or equal to k_1, but strictly smaller than k_2.

Section 4.2.3.1. Sorted Search Table

One simple data structure that can be used to implement the sorted-map ADT is an array list. An implementation using an array list is referred to as a *sorted search table*.

In this implementation, entries are sorted in the array by their keys. The following figure is a sorted array used to implement a map, which shows only keys for simplicity:

0	1	2	3	4	5	6	7	8	9
4	7	21	32	38	40	41	54	58	79

The primary advantage of the sorted search table is that we can use the binary search when implementing some map operations. Because of this, many operations run in $O(\log n)$ time. A disadvantage is that, since the order of entries in the array must be maintained, some operations requiring the reordering of entries need $O(n)$ time. The following table summarizes the performance of a sorted map:

Figure 4.23. Performance of a Sorted Map

Method	Running Time
size	$O(1)$
get	$O(\log n)$
put	$O(n)$; $O(\log n)$ if an entry with key is already in the map
remove	$O(n)$
firstEntry, lastEntry	$O(1)$

ceilingEntry, floorEntry lowerEntry, higherEntry	$O(\log n)$
subMap	$O(s + \log n)$ where s items are retrieved
entrySet, keyset,values	$O(n)$

The *subMap* operation works as follows. Let s be the number of entries in the given range. It first performs the binary search to find the entry with the key k_1 (the first entry within the range). This takes $O(\log n)$ time. After that, it scans the array to retrieve s entries. So, the *subMap* operation takes $O(s + \log n)$ time.

A Java implementation of the sorted map ADT can be found at the *SortedTableMap.java* file.

Section 4.2.3.2. Application of Sorted Maps

In this section, we describe an application that can take advantage of a sorted map.

When a user makes an airline reservation online, she usually, issues a query with the origin, destination, departure date, and departure time to the system. Then system returns a flight plan that "matches" the user query—either one that exactly matches the user query or one that is "close" to what the user requested. For example, suppose that the user has entered the following information:

Origin: NY
Destination: Dallas
Departure date: October 15, 2016
Departure time: 13:00

If the system can find a flight plan that exactly matches the query, that plan will be given to the user. However, if it cannot find the exact match, the system may present a plan that is close to the user query, such as the following:

Origin: NY
Destination: Dallas
Departure date: October 15, 2016
Departure time: 11:00

One way of implementing such an application is to use a sorted map, along with its inexact search operations. In a sorted map, a key is an object consisting of the above four elements:

k = (origin, destination, date, time)

The value is other relevant information about the flight plan, such as the flight number, class, flight duration, price, available seats, and whether the flight is nonstop or not.

Suppose that a user wants a flight that would depart at 10 AM. If no flight is available that departs at that exact time, she is okay with taking the last flight that departs before 10 AM. This query can be answered by the *floorEntry* method.

A user is also allowed to specify a range of departure times, such as 11 AM to 1 PM. Then the system can use the *subMap* method to answer the query. For example, suppose the user specifies the following two keys:

k_1 = (NY, Dallas, 10/15/2016, 11:00)
k_2 = (NY, Dallas, October 15, 2016, 11:00)

The following would be a one possible answer:

(NY, Dallas, 10/15/2016, 11:25) : (Delta 1234, Coach, 3:45, $250, 5, nonstop)
(NY, Dallas, 10/15/2016, 12:10) : (AA 2345, First Class, 3:10, $450, 5, nonstop)
(NY, Dallas, 10/15/2016, 12:45) : (Delta 3456, Coach, 5:45, $150, 5, onestop)

Section 4.2.4. Sets, Multisets, and Multimaps

In this section, we briefly discuss three collection ADTs that are closely related to the map ADT:

- *Set*—A set is an unordered collection of elements without duplicates. A set ADT typically supports an efficient set-membership test as well as standard set operations, such as union, intersection, and set difference.
- *Multiset*—A multiset is an unordered collection of elements that allows duplicates. A multiset is also referred to as a *bag*.
- *Multimap*—A multimap is a data structure similar to a map, in which a key can be mapped to more than one value.

Section 4.2.4.1. Set ADT

In Java, a set ADT is defined in *java.util.Set*. In the following description, each method is invoked on a set *S*.

It defines the following fundamental methods:

- add(*e*)—Adds the element *e* to *S*, if it is not already in the set.
- remove(*e*)—Removes the element *e* from *S*, if it is in the set.
- contains(*e*)—Returns *true* if *e* is in *S*, *false* otherwise.
- iterator( )*e*Returns an iterator of the elements in *S*.

Standard set operations are defined as the following methods:

- addAll(*T*)—This method defines the union operation. It updates *S* to also include all elements of *T*.
- retainAll(*T*)—This method defines the intersection operation. It updates *S* so that it only keeps those elements that are also in *T*.
- removeAll(*T*)—This method defines the set-difference operation. It updates *S* by removing any elements that are also in *T*.

The *addAll*, *retainAll*, and *removeAll* methods can be implemented using the *template method pattern* design by calling the fundamental methods, as shown below:

Code Segment 4.38

```
// S union T
1   public void addAll(Set<E> other) {
2     for (E element : other)
3       add(element);

// S intersection T
4   public void removeAll(Set<E> other) {
5     for (E element : other)
6       remove(element);

// S – T
7   public void retainAll(Set<E> other) {
8     ArrayList<E> leaving = new ArrayList<E>( );
9     for (E element : this)
10      if (other.contains(element))
11        leaving.add(element);
12    for (E element : leaving)
13      remove(element);
14  }
```

In line 8, a temporary array list, *leaving*, is created. In the *for* loop of lines 9 through 11, if an element in *S* (*this*) is also in *T* (*other*), it is added to *leaving*. After that, in lines 12 and 13, elements in *leaving* are removed from *S*.

A set can also be implemented as a *sorted set* if the elements within it are *comparable*. Recall that objects in a class are comparable if they belong to a *Comparable* class, or a *Comparator* object is provided. The following operations are supported in the *sorted-set* ADT:

- first( ): Returns the smallest element in *S*.
- last( ): Returns the largest element in *S*.
- ceiling(*e*): Returns the smallest element that is larger than or equal to *e*.
- floor(e)*:* Returns the largest element that is smaller than or equal to *e*.
- lower(*e*): Returns the largest element that is strictly smaller than *e*.
- higher(*e*): Returns the smallest element that is strictly larger than *e*.

- subset(e_1, e_2): Returns an iteration of all elements larger than or equal to e_1, but strictly smaller than e_2.
- pollFirst( ): Removes and returns the smallest element in *S*.
- pollLast( ): Removes an returns the largest element in *S*.

A set can be implemented using a map. In this case, keys are elements, but there are no values associated with keys. We can assign a special value, such as *null*, to the values of all entries. However, this approach is not efficient and, instead of using the *Entry* composite, we can just store set elements directly in a data structure.

In Java, the following set implementations are provided:

- *java.util.HashSet*—Implements the unordered-set ADT using a hash table.
- *java.util.concurrent.ConcurrentSkipListSet*—Implements the sorted-set ADT using a skip list (refer to pages 436 through 444 of our textbook for the skip-list data structure).
- *java.util.TreeSet*—Implements the sorted-set ADT using a balanced search tree.

Section 4.2.4.2. The Multiset ADT

A multiset can contain duplicates of the same element. For example, a multiset {a, c, g, c, a, k, a, w, w} contains three *a*'s two *c*'s, and two *w*'s. A typical multiset ADT supports the following operations. Here, *S* denotes a multiset.

- add(*e*)—Adds a single occurrence of *e* to *S*.
- contains(*e*)—Returns *true* if *e* is in *S*, *false* otherwise.
- count(*e*)—Returns the number of occurrences of *e* in *S*.
- remove(*e*)—Removes a single occurrence of *e* from *S*.
- remove(*e*, *n*)—Removes *n* occurrences of *e* from *S*.
- size( )—Returns the number of elements in *S*, including duplicates.
- iterator( )—Returns an iterator of the elements in *S* (repeating those with multiplicity greater than one).

Additionally, standard set operations can be defined.

A multiset allows duplicates. There are two notions of duplicates in Java. In the first, two elements are references to the same object. In the second, two elements are different objects, but are considered equivalent based on the *equals* method redefined for the object class. For example, consider the following code segment:

```
String s1 = new String ("multiset");
String s2 = s1;
String s3 = new String ("multiset");
```

The strings *s*1 and *s*2 are references to the same object. The string *s*1 and *s*3 are two different objects but are considered equivalent (and duplicates) based on the *equals* method defined in the *String* class, for they have the same sequence of characters. When we implement a multiset, all three of them are considered equivalent, and duplicates.

If a multiset is implemented using a map, an entry that is inserted into a map with *s*1 as a key can be retrieved later with *s*3 as a key. In actual implementation, we use one of the three strings as a key and store the remaining two strings in a container object; we then use that container object as the associated value, or we use one of them as a key and the number of duplicates as its value.

Section 4.2.4.3 The Multimap ADT

A multimap is a map that stores entries, each of which is a k, v pair, with additional properties. In a multimap, multiple entries can have the same key. For example, two entries k, v and k, v' are allowed to exist in a multimap.

There are two ways of implementing a multimap. We can store, for example, k, v and k, v' as two independent entries in the underlying data structure. We can also map the key k to a secondary container object including all associated values, such as v and v'.

The multimap ADT supports the following methods. The M in the following description denotes a multmap.

- get(k)—Returns the collection of all values associated with k in M.
- put(k, v)—Adds a new entry to M associating k with v, without overwriting the existing mappings for k.
- remove(k, v)—Removes an entry mapping k to v from M.
- removeAll(k)—Removes all entries with key k from M.

- size( )—Returns the number of entries in M, including multiple associations.
- entries( )—Returns the collection of all entries in M.
- keys( )—Returns the collection of keys for all entries in M, including duplicates for keys with multiple associations.
- keyset( )—Returns the nonduplicative (or distinct) collection of keys in M.
- values( )—Returns the collection of values for all entries in M.

Note that Java does not provide any interface/class for multisets and multimaps. The Google Core Libraries for Java (Guava) include *MultiSet* and *SortedMultiSet* interfaces and the *HashMultimap* class.

Module 4 Practice Questions

The following are some review questions for you to practice. Please read each question, think carefully, figure out your own answer or write your own program first, and then click "Show Answer" to compare yours to the suggested answer or the possible solution.

Test Yourself 4.3

Consider the priority queue ADT described in Section 4.1.1 (it is also described in Section 9.1.2 of the textbook). In the following table, show the return value and priority queue contents for the execution of each method shown in the first column. Assume that the priority queue is initially empty.

Method	Return Value	Priority Queue Contents
removeMin( )		
size( )		
insert(13, R)		
insert(8, K)		
isEmpty( )		
removeMin( )		
insert(26, G)		
min( )		
insert(17, W)		

Suggested answer:

Test Yourself 4.4

The priority queue discussed in Section 4.1.2 (also described in Section 9.2 of the textbook) requires a set of elements to have a *total ordering*. A total order must satisfy three properties. State these three properties.

Suggested answer:

Test Yourself 4.5

In Java, the *compareTo* method defined in a class *C*, which implements the *Comparable* interface, compares two objects based on natural ordering of the objects of *C*. True or False?

True.

Your option is right.

False.

Your option is wrong.

Test Yourself 4.6

Consider the following class definition:

```
public class Student implements Person {
  String id;
  String name;
  int age;
  public String getID() { return id;}
  public String getName() { return name; }
public int getAge() { return age; }
}
```

Write a Java comparator, named *StudentComparator*, which compares students based on their age.

Suggested answer - one possible solution:

```
public class StudentComparator implements Comparator<Student>{
       public int compare(Student a, Student b){
              if (a.getAge() < b.getAge()) return -1;
              else if (a.getAge() == b.getAge()) return 0;
              else return 1;
       }
}
```

Test Yourself 4.7

The following code segment is a part of the *UnsortedPriorityQueue* described in Section 4.1.2.4 (the code is also shown in Section 9.2.4 of the textbook). This class implements a priority queue using the *LinkedPositionalList* described in Section 3.1.3 (which is also described in Section 7.3 of the textbook).

```
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

. . .

  private Position<Entry<K,V>> findMin() {    // only called when nonempty
    Position<Entry<K,V>> small = list.first();
    for (Position<Entry<K,V>> walk : list.positions())
      if (compare(walk.getElement(), small.getElement()) < 0)
        small = walk;      // found an even smaller key
    return small;
  }
  . . .
}
```

What is the running time of the *findMin*method?

Suggested answer: $O(n)$.

Test Yourself 4.8

Preorder traversal of a minimum-oriented heap visits keys in non-decreasing order. True of False?

True.

Your option is wrong.

False.

Your option is right.

Test Yourself 4.9

In a minimum-oriented heap *T*, the position with the largest key is stored at the last position of *T*. True or False?

True.

Your option is wrong.

False.

Your option is right. The position with the largest key is stored at one of leaf nodes (not necessarily at the last position).

Test Yourself 4.10

A heap can be represented as an array. Let *p* be a position in a heap *T*. Let *index*(*p*) be the index of *p* in the array that represents *T*. Suppose *q* is the left child of *p* and *r* is the right child of *q*. Express the index of *r* in terms of *index*(*p*).

Suggested answer:

Test Yourself 4.11

Assume that you have n integers stored in a collection. Describe an algorithm that selects the k largest integer in $O(n \log k)$ time using $O(k)$ additional space.

Suggested answer: We use a minimum-oriented heap that can store k integers. We begin by inserting the first k integers. Then, from that point on, if the next integer is greater than the smallest integer in the heap, we remove the smallest integer and then insert the new integer. There will be at most $2n$ heap operations, each of which takes $O(\log k)$ time since the heap has at most k entries. Therefore, the total running time is $O(n \log k)$ and the heap needs $O(k)$ space.

Test Yourself 4.12

What is the worst-case running time of the *remove* method of Code Segment 4.25 (you can also find this code segment in page 409 of the text book)?

Suggested answer: $O(n)$. It invokes the *findIndex* method, which has to scan the whole *ArrayList* in the worst case.

Test Yourself 4.13

When two or more keys are mapped to the same hash value, it is called *collision*. True or False?

True.

Your option is right.

False.

Your option is wrong.

Test Yourself 4.14

The evaluation of a hash function consists of two steps. What are these two steps?

Suggested answer: *Hash code* and *compression*.

A hash code maps a key to an integer.

A compression function maps the hash code to an integer within a range of indices, [0, *N*-1], for a bucket array.

Test Yourself 4.15

Assume that you designed a hash table using the *multiply-add-and-divide* (*MAD*) function, which is defined as

$$h(i) = [(ai + b) \mod p] \mod N$$

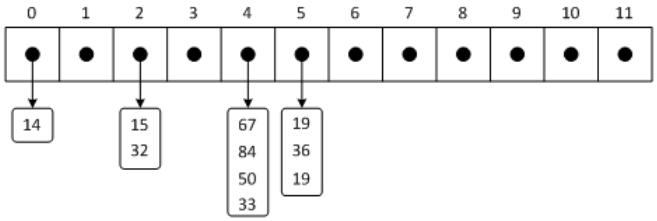
Here, $N = 12, p = 17, a = 2,$ and $b = 1$.

Also assume that you use the chaining method to resolve collisions.

Illustrate how the following sequence of keys are mapped to the hash table of size $12 (N = 12)$, in the give order, using a figure similar to Figure 4.20 (a similar figure is also shown in page 417 of the textbook).

<19, 14, 67, 36, 53, 84, 50, 15, 33, 32>

Suggested answer:



Test Yourself 4.16

The load factor is always smaller than 1 when the chaining method is used but it can be greater than 1 when the open addressing method is used. True or False?

True.

Your option is wrong.

False.

Your option is right.

Test Yourself 4.17

This question is about the *UnsortedTableMap* described in Section 4.2.1.4 (which is also described in Section 10.1.4 of the textbook). Suppose that the value, *v*, of an entry (*k*, *v*) can be *null*. Then, there is one potential issue: When the *get*(*k*) method returns *null*, we cannot know whether it is because there is no entry with key = *k* or it found a valid entry (*k*, *null*). To resolve this ambiguity, we can define a *containsKey*(*k*) method which returns true if an entry with key = *k* exists in the map. Write a Java code that implements the *containsKey*(*k*) method for the *UnsortedTableMap* class.

Suggested answer - one possible solution:

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

Test Yourself 4.18

This question is about the *UnsortedTableMap* described in Section 4.2.1.4 (which is also described in Section 10.1.4 of the textbook). Write a Java method, named *putIfAbsent*(key *k*, value *v*), for the *UnsortedTableMap* class. If there is no entry with key = *k*, it put the new entry in the map and returns *null*. If there exists such an entry, it returns the value of that entry. This method should use the *get* method and the *put* method of the *UnsortedTableMap* class and it must run in $O(n)$ time.

Suggested answer:

```
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
  }
}
```

Test Yourself 4.19

Consider a hash table that uses the separate chaining method to resolve collisions. What is the worst-case and best-case running time of a search operation on the table?

Suggested answer: The worst-case time is $O(n)$. The best case is $O(1)$.

Test Yourself 4.20

This question is about the *addAll* method described in Code Segment 4.38 (it is also described in page 446 of the textbook). Let *n* be the size of a set *S* and *m* be the size of a set *T*. What are the expected and worst-case running times of *S.addAll*(*T*)? Assume that both sets are implemented using a hash table.

Suggested answer: The expected running time of adding an element to a hash table is $O(1)$ and the worst-case running time of adding an element to a hash table is $O(n)$. Therefore, the expected running time of *S.addAll*(*T*) is $O(m)$ and the worst-case running time of *S.addAll*(*T*) is $O(mn)$.

References

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.) Wiley.
- Gosling, J., Joy, B., Steele, G., Bracha, G., & Buckley, A. (2015). *The Java® language specification, Java SE 8 edition.* Oracle America Inc.
- Oracle. *The Java Tutorials*. Retrived from https://docs.oracle.com/javase/tutorial/index.html.