20 Maps

20.1 Introduction

A very useful collection is one that is a hybrid of lists and sets, called a *map*, *table*, *dictionary*, or *associative array*. A map (as we will call it), is a collection whose elements (which we will refer to as *values*) are unordered, like a set, but whose values are accessible via a *key*, akin to the way that list elements are accessible by indices.

> **Map**: An unordered collection whose values are accessible using a key.

Another way to think of a map is as a function that maps keys to values (hence the name), like a map or function in mathematics. As such, a map is a set of ordered pairs of keys and values such that each key is paired with a single value (though a value may be paired with several keys).

20.2 The Map ADT

Maps store values of arbitrary type with keys of arbitrary type, so the ADT is *map of (K,T)*, where *K* is the type of the keys and *T* is the type of the values in the map. The carrier set of this type is the set of all ordered pairs whose first element is of type *K* and whose second element is of type *T*. The carrier set thus includes the empty map, the maps with one ordered pair of values of types *K* and *T*, the maps with two ordered pairs of values of types *K* and *T*, and so forth.

The essential operations of maps, in addition to those common to all collections, are those for inserting, deleting, searching and retrieving keys and values.

empty?(*m*)—Return true if and only if *m* is the empty map.

size(*m*)—Return the number of pairs in map *m*.

has_key?(*m*, *k*)—Return true if and only if map *m* contains an ordered pair whose first element is *k*.

has_value?(*m*, *v*)—Return true if and only if map *m* contains an ordered pair whose second element is *v*.

m[*k*]=*v*—Return a map just like *m* except that the ordered pair <*k*, *v*> has been added to *m*. If *m* already contains an ordered pair whose first element is *k*, then this ordered pair is replaced with the new one.

delete(*m, k*)—Return a map just like *m* except that an ordered pair whose first element is *k* has been removed from *m*. If no such ordered pair is in *m*, then the result is *m* (in other words, if there is no pair with key *k* in *m*, then this operation has no effect).

m[*k*]—Return the second element in the ordered pair whose first element is *k*. Its precondition is that *m* contains an ordered pair whose first value is *k*.

There is considerable similarity between these operations and the operations for lists and sets. For example, the *delete_at*() operation for lists takes a list and an index and removes the element at the designated index, while the map operation takes a map and a key and removes the key-value pair matching the key. On the other hand, when the list index is out of range, there is a precondition violation, while if the key is not present in the map, the map is unchanged. This latter behavior is the same as what happens with sets when the set *delete*() operation is called with an argument that is not in the set.

20.3 The Map Interface

The diagram below in Figure 1 shows the Map interface, which is a sub-interface of Collection. It includes all the operations of the map of (*K*, *T*) ADT. As usual, the operation parameters are a bit different from those in the ADT because the map is an implicit parameter of all operations.

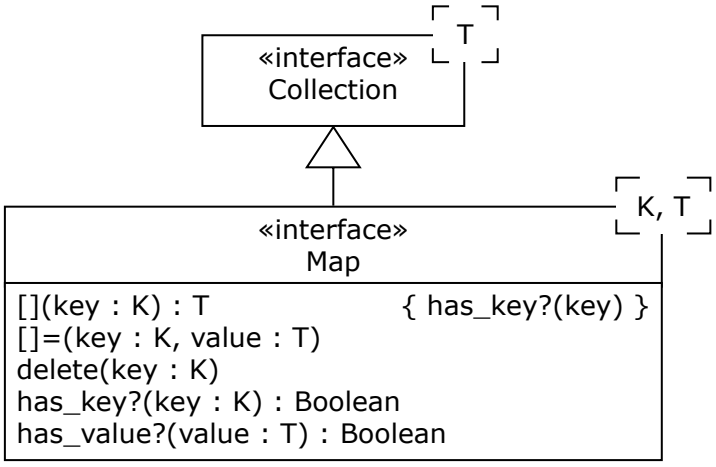

Figure 1: The Map Interface

The contains?() operation inherited from Collection is a synonym for has_value?(). As a Collection, a Map has an associated Iterator (returned by the Collection iterator() operation). In Ruby, it is convenient to have this Iterator traverse the Map and returns its key-value pairs as an array with two elements, the first being the key and the second being the value.

20.4 Contiguous Implementation of the Map ADT

As with sets, using an array or ArrayList to store the ordered pairs of a map is not very efficient because only one of the three main operations of insertion, deletion, and search can be done quickly. Also as with sets, a characteristic function can be used if the key set is a small integral type (or a small sub-range of an integral type), but this situation is rare. Finally as with sets, hashing provides a very efficient contiguous implementation of maps and we will discuss how this works later on.

20.5 Linked Implementation of the Map ADT

As with sets, using linked lists to store map elements is not much better than using an array. But again as with sets, binary search trees can store map elements and provide fast insertion, deletion, and search operations on keys. Furthermore, using binary search trees to store map elements allows the elements in the map to be traversed in sorted key order, which is sometimes very useful. A TreeMap is thus an excellent implementation of the Map interface.

The trick to using binary search tree to store map elements is to create a class to hold key-value pairs, redefining its relational operators to compare keys, and using this as the datum stored in nodes of the binary search tree. Dummy class instances with the correct key field can then be used to search the tree and retrieve key-value pairs.

20.6 Summary and Conclusion

Maps are extremely important collections because they allow values to be stored using keys, a very common need in programming. The map of (*K*, *T*) ADT specifies the essential features of the type, and the Map interface captures these features and places maps in the Container hierarchy. Contiguous implementations are not well suited for maps (except hash tables, which we discuss in the next chapter). Binary search trees, however, provide very efficient implementations, so a TreeMap class is a good realization of the Map interface. Figure 2 shows how Maps and TreeMaps fit into the container hierarchy.

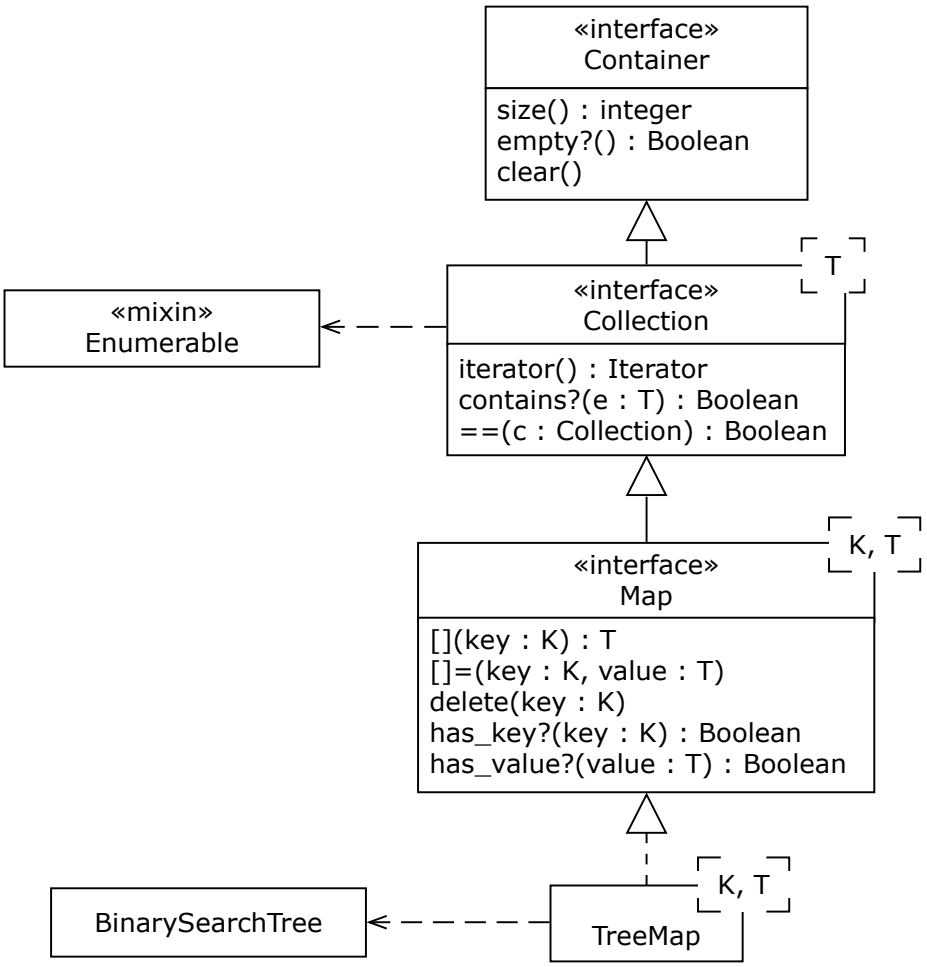

Figure 2: Maps and TreeMaps in the Container Hierarchy

20.7 Review Questions

1. Why is a map called a map?
2. The Collection interface has a generic parameter T, and the Map interface has generic parameters K and T. What is the relationship between them?
3. Why is an array or an ArrayList not a good data structure for implementing the map ADT?
4. Why is a LinkedList not a good data structure to implement the map ADT?
5. Why is a binary search tree a good data structure for implementing the map ADT?

20.8 Exercises

1. Make a function mapping the states California, Virginia, Michigan, Florida, and Oregon to their capitals. If you wanted to store this function in a map ADT, which values would be the keys and which the elements?
2. Represent the map described in the previous exercise as a set of ordered pairs. If this map is *m*, then also represent as a set of ordered pairs the map that results when the operation *remove*(*m*, Michigan) is applied.
3. Is an iterator required for maps? How does this compare with the situation for lists?
4. To make a `TreeMap` class that uses the `BinarySearchTree` class discussed in Chapter 18, you will need to make a class to hold key-value pairs with comparison operations that work on the keys. Write such a `Pair` class in Ruby.
5. Write the beginnings of a `TreeMap` class in Ruby that includes its attributes, invariant, constructor, and the operations inherited from the `Collection` interface. You will need to make use of the `Pair` class from the previous exercise.
6. Continue the implementation begun in exercise 5 by writing the `[]=`, `[]`, and `delete()` operations for `TreeMap`.

7. Continue the implementation begun in exercise 5 by writing the `Enumerable.each()` operation for TreeMap. This operation should produce each key-value pair, yielding the key and value as two elements, just as this operation does for the Ruby `Hash` class. Also write `each_key()` and `each_value()` operations, again modeled on the Ruby `Hash` class.
8. Continue the implementation begun in exercise 5 by writing the iterator() operation (you will need a TreeMapIterator class for TreeMap). The Iterator.current() operation should return a key-value pair as an array with two elements: the key and the value.

20.9 Review Question Answers

1. A map associates keys and values such that each key is associated with at most one values. This is the definition of a function from keys to values. Functions are also called *maps*, and we take the name of the collection from this meaning of the word.
2. The Collection interface generic parameter T is the same as the Map interface generic parameters T: the elements of a Collection are also the values of a Map. But Maps have an additional data item—the key—whose type is K.
3. An array or an ArrayList is not a good data structure for implementing the map ADT because the key-value pairs would have to be stored in the array or ArrayList in order or not in order. If they are stored in order, then finding a key-value pair by its key is fast (because we can use binary search), but adding and removing pairs is slow. If pairs are not stored in order, then they can be inserted quickly by appending them at the end of the collection, but searching for them or finding them when they need to be removed are slow operations because they must use sequential search.
4. A LinkedList is not a good data structure to implement the map ADT because although key-value pairs can be inserted quickly into a LinkedList, searching for pairs or finding them when they need to be removed are slow operations because the LinkedList must be traversed node by node.
5. A binary search tree is a good data structure for implementing the map ADT because (assuming that the tree remains fairly balanced), adding key-value pairs, searching for them by key, and removing them by key, are all done very quickly. Furthermore, if the nodes in the tree are traversed in order, then the key-value pairs are accessed in key-order.