19 Sets

19.1 Introduction

Lists have a linear structure and trees have a two-dimensional structure. We now turn to unstructured collections. The simplest unstructured collection is the set.

> **Set**: An unordered collection in which an element may appear at most once.

We first review the set ADT and then discuss ADT implementation.

19.2 The Set ADT

The *set of T* abstract data type is the ADT of sets of elements of type *T*. Its carrier set is the set of all sets of *T*. This ADT is the abstract data type of sets that we all learned about starting in grade school. Its operations are exactly those we would expect (and more could be included as well):

$e \in s$—Return true if e is a member of the set s.

$s \subseteq t$—Return true if every element of s is also an element of t.

$s \cap t$—Return the set of elements that are in both s and t.

$s \cup t$—Return the the set of elements that are in either s or t.

$s - t$—Return the set of elements of s that are not in t.

$s == t$—Return true if and only if s and t contain the same elements.

The set ADT is so familiar that we hardly need discuss it. Instead, we can turn immediately to the set interface that all implementation of the set ADT will use.

19.3 The Set Interface

The `Set` interface appears in Figure 1. The `Set` interface is a sub-interface of the `Collection` interface, so it inherits all the operations of `Collection` and `Container`. Some of the set ADT operations are included in these super-interfaces (such as `contains?()`), so they don't appear explicitly in the `Set` interface.

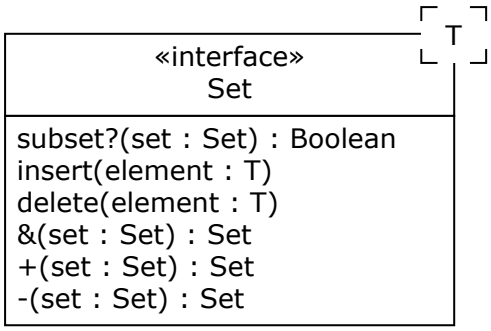

Figure 1: The Set Interface

19.4 Contiguous Implementation of Sets

The elements of a set can be stored in an array or an ArrayList but this approach is not very efficient. To see this, lets consider how the most important set operations—insertion, deletion, and membership check—would be implemented using an array. If the elements are not kept in order, then inserting them is very fast, but deleting them is slow (because a sequential search must be done to find the deleted element), and the membership check is slow (because it also requires a sequential search). If the elements are kept in order, then the membership check is fast (because binary search can be used), but insertion and deletion are very slow because on average half the elements must be moved to open or close a hole for the inserted or deleted element.

There is one way to implement sets using contiguous storage that is very time efficient, though it may not be space efficient. A boolean array, called a *characteristic function*, can be used to represents a set. The characteristic function is indexed by set elements so that the value of the characteristic function at index x is true if and only if x is in the set. Insertion, deletion, and the membership check can all be done in constant time. The problem, of course, is that each characteristic function array must have an element for every possible value that could be in the set, and these values must be able to index the array. If a set holds values from a small sub-range of an integral type, such as the ASCII characters, or integers from 0 to 50, then this technique is feasible (though it still may waste a lot of space). But for large sub-ranges or for non-integral set elements, this technique is no longer possible.

There is one more contiguous implementation technique that is very fast for sets: hashing. We will discuss using hashing to implement sets later on.

19.5 Linked Implementation of Sets

The problem with the contiguous implementation of sets is that insertion, deletion, and membership checking cannot all be done efficiently. The same holds true of linked lists. We have, however, encountered a data structure that provides fast insertion, deletion, and membership checking (at least in the best and average cases): binary search trees. Recall that a binary search tree (if fairly well balanced) can be searched in $O(\lg n)$ time, and elements can be inserted and deleted in $O(\lg n)$ time.

An implementation of sets using binary search trees is called a TreeSet. TreeSets are very efficient implementations of sets provided some care is taken to keep them from becoming too unbalanced. TreeSets have the further advantage of allowing iteration over the elements in the set in sorted order, which can be very useful in some applications. As you will see when you do the exercises, all the hard work creating the BinarySearchTree class will now pay off by making it very easy to implement a TreeSet class.

19.6 Summary and Conclusion

Sets are useful ADTs that can be implemented efficiently using binary search trees. Figure 2 below shows the portion of the container hierarchy culminating in the TreeSet class, including the Enumerable mixin and the BinarySearchTree class, to illustrate how all these interfaces and classes fit together.

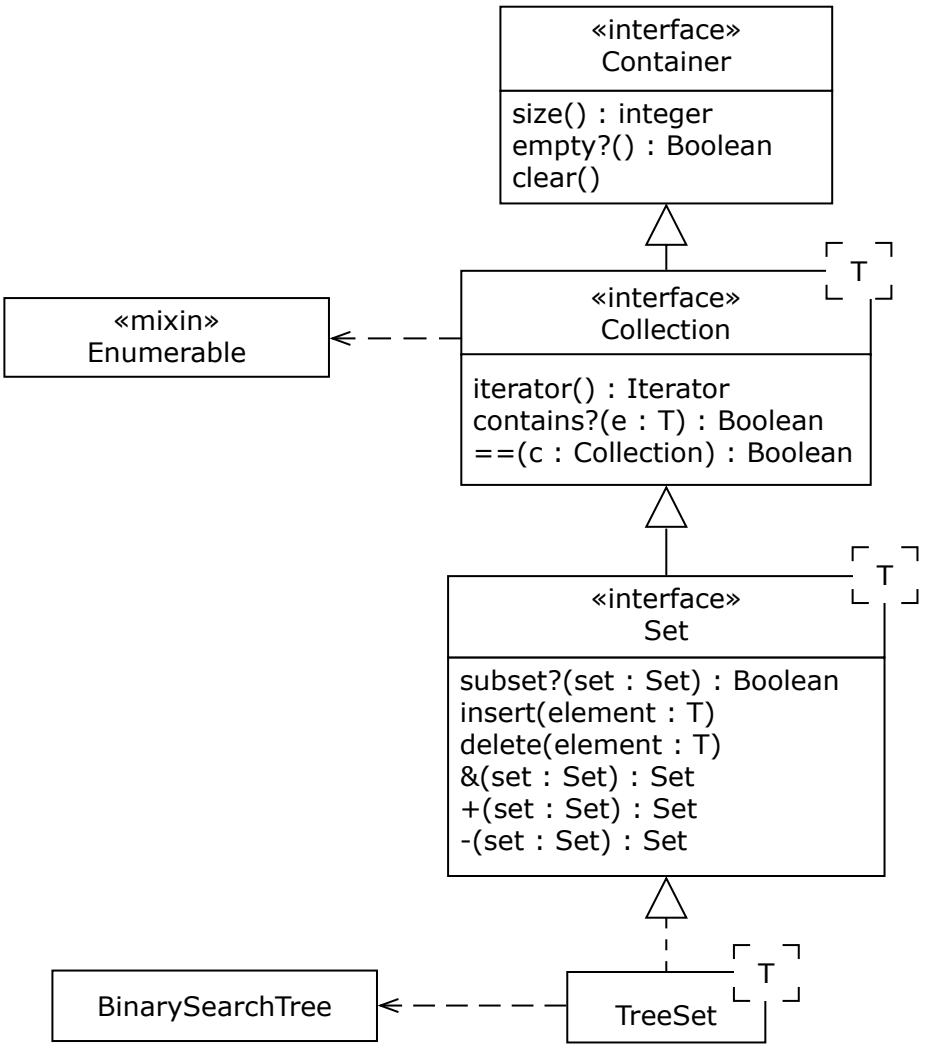

Figure 2: Sets and TreeSets in the Container Hierarchy

19.7 Review Questions

1. Which set operation appear sin the set ADT but does not appear explicitly in the Set interface? Why is it not present?
2. What is a characteristic function?
3. Why is an array or an ArrayList not a good data structure for implementing the set ADT?
4. Why is a LinkedList not a good data structure to implement the set ADT?
5. Why is a binary search tree a good data structure for implementing the set ADT?

19.8 Exercises

1. What other operation do you think might be useful to add to the set of *T* ADT?
2. Is an iterator required for sets? How does this compare with the situation for lists?

3. Write the beginnings of a TreeSet class in Ruby that includes its attributes, invariant, constructor, and the operations inherited from the Collection interface.
4. Continue the implementation begun in exercise 3 by writing the subset?() operation for TreeSet.
5. Continue the implementation begun in exercise 3 by writing the insert() and delete() operations for TreeSet.
6. Continue the implementation begun in exercise 3 by writing the +() (union) operation for TreeSet. Note that you need to create and return a brand new TreeSet.
7. Continue the implementation begun in exercise 3 by writing the &() (intersection) operation for TreeSet. Note that you need to create and return a brand new TreeSet.
8. Continue the implementation begun in exercise 3 by writing the -() (relative complement) operation for TreeSet. Note that you need to create and return a brand new TreeSet.

19.9 Review Question Answers

1. The membership operation in the set ADT does not appear explicitly in the Set interface because the contains() operation from the Collection interface already provides this functionality.
2. A characteristic function is a function created for a particular set that takes a value as an argument and returns true just in case that value is a member of the set. For example, the characteristic function $f(x)$ for the set $\{a, b, c\}$ would have the value true for $f(a)$, $f(b)$, and $f(c)$, but false for any other argument.
3. If an array or an ArrayList is used to implement the set ADT, then either the insertion, deletion, or set membership operations will be slow. If the array or ArrayList is kept in order, then the set membership operation will be fast (because binary search can be used), but insertion and deletion will be slow because elements will have to be moved to keep the array in order. If the array or ArrayList is not kept in order, then insertions will be fast, but deletions and set membership operations will require sequential searches, which are slow.
4. If a LinkedList is used to implement the set ADT, then deletion and membership testing will be slow because a sequential search will be required. If the elements of the list are kept in order to speed up searching (which only helps a little because a sequential search must still be used), then insertion is made slower.
5. A binary search tree is a good data structure for implementing the set ADT because a binary search tree allows insertion, deletion, and membership testing to all be done quickly, in $O(\lg n)$ time (provided the tree is kept fairly balanced).