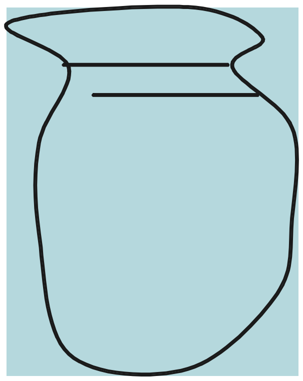
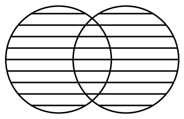
Chapter 8: Bags and Sets

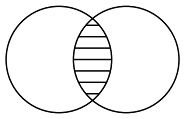
In the stack and the queue abstractions, the order that elements are placed into the container is important, because the order elements are removed is related to the order in which they are inserted. For the *Bag*, the order of insertion is completely irrelevant. Elements can be inserted and removed entirely at random.



By using the name *Bag* to describe this abstract data type, the intent is to once again to suggest examples of collection that will be familiar to the user from their everyday experience. A bag of marbles is a good mental image. Operations you can do with a bag include inserting a new value, removing a value, testing to see if a value is held in the collection, and determining the number of elements in the collection. In addition, many problems require the ability to loop over the elements in the container. However, we want to be able to do this without exposing details about how the collection is organized (for example, whether it uses an array or a linked list). Later in this chapter we will see how to do this using a concept termed an *iterator*.

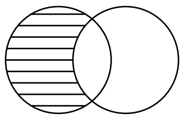
A *Set* extends the bag in two important ways. First, the elements in a set must be unique; adding an element to a set when it is already contained in the collection will have no effect. Second, the set adds a number of operations that combine two sets to produce a new set. For example, the set *union* is the set of values that are present in either collection.





The *intersection* is the set of values that appear in both collections.

A set *difference* includes values found in one set but not the other.



Finally, the subset test is used to determine if all the values found in one collection are also found in the second. Some implementations of a set allow elements to be repeated more than once. This is usually termed a *multiset*.

The Bag and Set ADT specifications

The traditional definition of the Bag abstraction includes the following operations:

Add (newElement)	Place a value into the bag
Remove (element)	Remove the value
Contains (element)	Return true if element is in collection
Size ()	Return number of values in collection
Iterator ()	Return an iterator used to loop over collection

As with the earlier containers, the names attached to these operations in other implementations of the ADT need not exactly match those shown here. Some authors prefer “insert” to “add”, or “test” to “contains”. Similarly, there are differences in the exact meaning of the operation “remove”. What should be the effect if the element is not found in the collection? Our implementation will silently do nothing. Other authors prefer that the collection throw an exception in this situation. Either decision can still legitimately be termed a bag type of collection.

The following table gives the names for bag-like containers in several programming languages.

operation	Java Collection	C++ vector	Python
Add	Add(element)	Push_back(element)	Lst.append(element)
remove	Remove(element)	Erase(iterator)	Lst.remove(element)
contains	Contains(element)	Count(iterator)	Lst.count(element)

The set abstraction includes, in addition to all the bag operations, several functions that work on two sets. These include forming the intersection, union or difference of two sets, or testing whether one set is a subset of another. Not all programming languages include set abstractions. The following table shows a few that do:

operation	Java Set	C++ set	Python list comprehensions
intersection	retainAll	Set_intersection	[x for x in a if x in b]
union	addAll	Set_union	[x if (x in b) or (x in a)]
difference	removeAll	Set_difference	[x for x in a if x not in b]
subset	containsAll	includes	Len([x for x in a if x not in b]) != 0

Python list comprehensions (modeled after similar facilities in the programming languages ML and SETL) are a particularly elegant way of manipulating set abstractions.

Applications of Bags and Sets

The bag is the most basic of collection data structures, and hence almost any application that does not require remembering the order that elements are inserted will use a variation on a bag. Take, for example, a spelling checker. An on-line checker would place a dictionary of correctly spelled words into a bag. Each word in the file is then tested against the words in the bag, and if not found it is flagged. An off-line checker could use set operations. The correctly spelled words could be placed into one bag, the words in the document placed into a second, and the difference between the two computed. Words found in the document but not the dictionary could then be printed.

Bag and Set Implementation Techniques

For a Bag we have a much wider range of possible implementation techniques than we had for stacks and queues. So many possibilities, in fact, that we cannot easily cover them in contiguous worksheets. The early worksheets describe how to construct a bag using the techniques you have seen, the dynamic array and the linked list. Both of these require the use of an additional data abstraction, the iterator. Later, more complex data structures, such as the skip list, avl tree, or hash table, can also be used to implement bag-like containers.

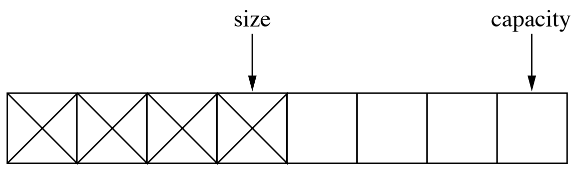
Another thread that weaves through the discussion of implementation techniques for the bag is the advantages that can be found by maintaining elements in order. In the simplest there is the sorted dynamic array, which allows the use of binary search to locate elements quickly. A skip list uses an ordered linked list in a more subtle and complex fashion. AVL trees and similarly balanced binary trees use ordering in an entirely different way to achieve fast performance.

The following worksheets describe containers that implement the bag interface. Those involving trees should be delayed until you have read the chapter on trees.

Worksheet 21	Dynamic Array Bag
Worksheet 22	Linked List Bag
Worksheet 23	Introduction to the Iterator
Worksheet 24	Linked List Iterator
Worksheet 26	Sorted Array Bag
Worksheet 28	Skip list bag
Worksheet 29	Balanaced Binary Search Trees
Worksheet 31	AVL trees
Worksheet 37	Hash tables

Building a Bag using a Dynamic Array

For the Bag abstraction we will start from the simpler dynamic array stack described in Chapter 6, and not the more complicated deque variation you implemented in Chapter 7. Recall that the Container maintained two data fields. The first was a reference to an array of objects. The number of positions in this array was termed the *capacity* of the container. The second value was an integer that represented the number of elements held in the container. This was termed the *size* of the collection. The size must always be smaller than or equal to the capacity.



As new elements are inserted, the size is increased. If the size reaches the capacity, then a new array is created with twice the capacity, and the values are copied from the old array into the new. This process of reallocating the new array is an issue you have already solved back in Chapter 6. In fact, the function *add* can have exactly the same behavior as the function *push* you wrote for the dynamic array stack. That is, add simply inserts the new element at the end of the array.