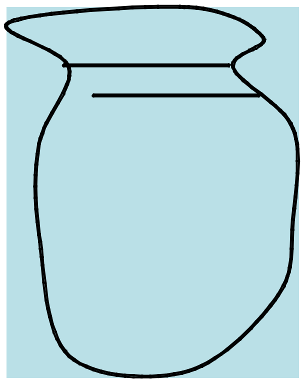
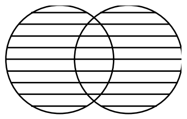
Chapter 8: Bags and Sets

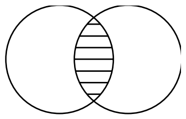
In the stack and the queue abstractions, the order that elements are placed into the container is important, because the order elements are removed is related to the order in which they are inserted. For the *Bag*, the order of insertion is completely irrelevant. Elements can be inserted and removed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

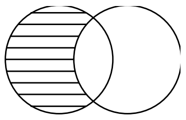
A *Set* extends the bag in two important ways. First, the elements in a set must be unique; adding an element to a set when it is already contained in the collection will have no effect. Second, the set adds a number of operations that combine two sets to produce a new set. For example, the set *union*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

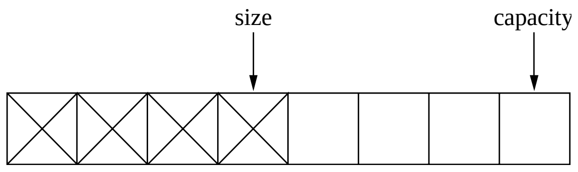
Another thread that weaves through the discussion of implementation techniques for the bag is the advantages that can be found by maintaining elements in order. In the simplest there is the sorted dynamic array, which allows the use of binary search to locate elements quickly. A skip list uses an ordered 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.



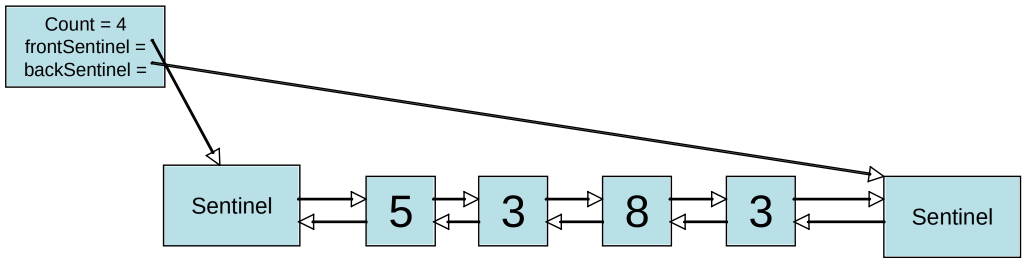
As new elements are inserted, the size is increased. If the size reaches the capacity, then a new array is created with twice the capacity, and the values are copied from the old array into the new. This process of reallocating the new array is an issue you have already solved back in Chapter 6. In fact, 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.

The contains function is also relatively simple. It simply uses a loop to cycle over the index values, examining each element in turn. If it finds a value that matches the argument, it returns true. If it reaches the end of the collection without finding any value, it returns false.

The remove function is the most complicated of the Bag abstraction. To simplify this task we will divide it into two distinct steps. The remove function, like the contains function, will loop over each position, examining the elements in the collection. If it finds one that matches the desired value, it will invoke a separate function, removeAt, that removes the value held at a specific location. You will complete this implementation in Worksheet 21.

Constructing a Bag using a Linked List

To construct a Bag using the idea of a Linked List we will begin with the list deque abstraction you developed in Chapter 7. Recall that this implementation used a sentinel at both ends and double links.



The *contains* function must use a loop to cycle over the chain of links. Each element is tested against the argument. If any are equal, then the Boolean value true is returned. Otherwise, if the loop terminates without finding any matching element, the value False is returned.

The *remove* function uses a similar loop. However, this time, if a matching value is found, then the function *removeLink* is invoked. The remove function then terminates, without examining the rest of the collection. (As a consequence, only the first occurrence of a value is removed. Repeated values may still be in the collection. A question at the end of this chapter asks you to consider different implementation techniques for the *removeAll* function.)

Introduction to the Iterator

As we noted in Chapter 5, one of the primary design principles for collection classes is *encapsulation*. The internal details concerning how an implementation works are hidden behind a simple and easy to remember interface. To use a Bag, for example, all you need know is the basic operations are add, collect and remove. The inner workings of the implementation for the bag are effectively hidden.

When using collections a common requirement is the need to loop over all the elements in the collection, for example to print them to a window. Once again it is important that this process be performed without any knowledge of how the collection is represented in memory. For this reason the conventional solution is to use a mechanism termed an *Iterator*.

```
/* conceptual interface */
 int  hasNext ( );
 TYPE next ( );
 void remove ( );
```

```
LinkedListIterator itr;
TYPE current;
…
ListIteratorInit (aList, itr);
while (ListIteratorHasNext(itr)) {
  current = ListIteratorNext(itr);
  … /* do something with current
*/
}
```

Each collection will be matched with a set of functions that implement this interface. The functions next and hasNext are used in combination to write a simple loop that will cycle over the values in the collection.

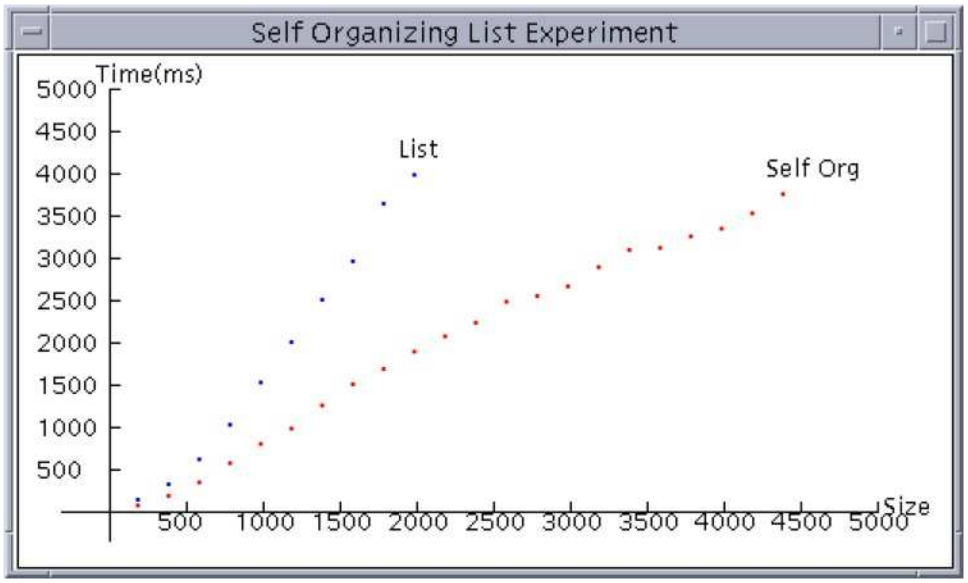
The iterator loop exposes nothing regarding the structure of the container class. The function *remove* can be used to delete from the collection the value most recently returned by next.

Notice that an iterator is an object that is separate from the collection itself. The iterator is a *facilitator object*, that provides access to the container values. In worksheets 23 and 24 you complete the implementation of iterators for the dynamic array and for the linked list.

Self Organizing Lists

We have treated all list operations as if they were equally likely, but this is not always true in practice. Often an analysis of the frequency of operations will suggest ways that a data structure can be modified in order to improve performance. For example, one common situation is that a successful search will frequently be followed relatively soon by a search for the same value. One way to handle this would be for a successful search to remove the value from the list and reinsert it at the front. By doing so, the subsequent search will be much faster.

A data structure that tries to optimize future performance based on the frequency of past operations is called *self-organizing*. We will subsequently encounter a number of other self-organizing data structures. Given the right circumstances self-organization can be very effective. The following chart shows the results of one simple experiment using this technique.



Simple Set Operations

Any bag can be used to construct a set. Among the basic functions the only change is in the addition operation, which must check that the value is not already in the collection. Operations such as set union, intersection and difference can all be implemented with simple loops. The following pseudo-code shows set union:

```
setUnion (one, two, three) /* assume set three is initially empty */
   for each element in one
      if value is found in two
         add to set three
```

The algorithmic execution time for this operation depends upon a combination of the time to search the second set, and the time to add into the third set. If we use a simple dynamic array or linked list bag, then both of these operations are O(n) and so, since the outer loop cycles over all the elements in the first bag, the complete union operation is $O(n^2)$.

Simple algorithms for set intersection, difference, and subset are similar, with similar performance. These are analyzed in questions at the end of this chapter. Faster set algorithms require adding more structure to the collection, such as keeping elements in sorted order.

The Bit Set

A specialized type of set is used to represent positive integers from a small range, such as a set drawn from the values between 0 and 75. Because only integer values are allowed, this can be represented very efficiently as binary values, and is therefore termed a *bit set*. Worksheet 25 explores the implementation of the bit set.

Advanced Bag Implementation Techniques

Several other bag implementation techniques use radically different approaches. Indeed, three of the following chapters will be devoted in some fashion or another to bag data structures. Chapter 9 explores the saving possible when elements are maintained in order. Chapter 10 introduces a new type of list, the skip list, as well as an entirely different form of organization, the binary tree. Finally, Chapter 12 introduces yet another technique, hashing, which produces some of the fastest implementations of bag operations.

Self Study Questions

1. What features characterize the Bag data type?

2. How is a set different from a bag?

3. When a dynamic array is used as a bag, how is the add operation different from operations you have already implemented in the dynamic array stack?

4. When a dynamic array is used as a set, how is the add operation different from operations you have already implemented in the dynamic array stack?

5. Using a dynamic array as the implementation technique, what is the big-oh algorithmic execution time for each of the bag operations? Is this different if you use a linked list as the implementation technique?

6. What is an iterator? What purpose does the iterator address?

7. Any bag can be used as the basis for a set. When using a dynamic array or linked list as the bag, what is the algorithmic execution time for the set operations?

8. What is a bit set?

Short Exercises

1. Assume you wanted to define an equality operator for bags. A bag is equal to another bag if they have the same number of elements, and each element occurs in the other bag the same number of times. Notice that the order of the elements is unimportant. If you implement bags using a dynamic array, how would you implement the equality-testing operator? What is the algorithmic complexity of your operation? Can you think of any changes to the representation that would make this operation faster?

Analysis Exercises

1. There are two simple approaches to implementing bag structures; using dynamic arrays or linked list. The only difference in algorithmic execution time performance between these two data structures is that the linked list has guaranteed constant time insertion, while the dynamic array only has amortized constant time insertion. Can you design an experiment that will determine if there is any measurable difference caused by this? Does your experiment yield different results if a test for inclusion is performed more frequently that insertions?

2. What should the remove function for a bag do if no matching value is identified? One approach is to silently do nothing. Another possibility is to return a Boolean flag indicating whether or not removal was performed. A third possibility is to throw an exception or assertion error. Compare these three possibilities and give advantages and disadvantages of each.

3. Finish describing the naïve set algorithms that are built on top of bag abstractions. Then, using invariants, provide an informal proof of correctness for each of these algorithms.

4. Assume that you are assigned to test a bag implementation. Knowing only the bag interface, what would be some good example test cases? What are some of the boundary values identified in the specification?

5. The entry for Bag in the NIST *Dictionary of Algorithms and Data Structures* suggests a number of axioms that can be used to characterize a bag. Can you define test cases that would exercise each of these axioms? How do your test cases here differ from those suggested in the previous question?

6. Assume that you are assigned to test a set implementation. Knowing only the set interface, what would be some good example test cases? What are some of the boundary values identified in the specification?

7. The bag data type allows a value to appear more than once in the collection. The remove operation only removes the first matching value it finds, potentially leaving other occurrences of the value remaining in the collection. Suppose you wanted to implement a removeAll operation, which removed all occurrences of the argument. If you did so by making repeated calls on remove what would be the algorithmic complexity of this operation? Can you think of a better approach if you are using a dynamic array as the underlying container? What about if you are using a linked list?

Programming Projects

1. Implement an experiment designed to test the two simple bag implementations, as described in analysis exercise 1.

2. Using either of the bag implementations implement the simple set algorithms and empirically verify that they are $O(n^2)$. Do this by performing set unions for two sets with n elements each, of various values of n. Verify that as n increases the time to perform the union increases as a square. Plot your results to see the quadratic behavior.

On the Web

The wikipedia has (at the time of writing) no entry for the bag abstract data type , but does have an entry for the related data type termed the *multiset*. Other associated entries include bitset (termed a bit array or bitmap). The NIST *Dictionary of Algorithms and Data Structures* does have an entry that describes the bag.