Chapter 9: Searching, Ordered Collections

In Chapter 8 we said that a *bag* was a data structure characterized by three primary operations. These operations were inserting a new value into the bag, testing a bag to see if it contained a particular value, and removing an element from the bag. In our initial description we treated all three operations as having equal importance. In many applications, however, one or another of the three operations may occur much more frequently than the others. In this situation it may be useful to consider alternative implementation techniques. Most commonly the favored operation is searching.

We can illustrate an example with the questions examined back in Chapter 4. Why do dictionaries or telephone books list their entries in sorted order? To understand the reason, Chapter 4 posed the following two questions. Suppose you were given a telephone book and asked to find the number for an individual named Chris Smith. Now suppose you were asked to find the name of the person who has telephone number 543-7352. Which task is easier?

```
Abby Smith   954-9845
Chris Smith  234-7832
Fred Smith   435-3545
Jaimie Smith 845-2395
Robin Smith  436-9834
```

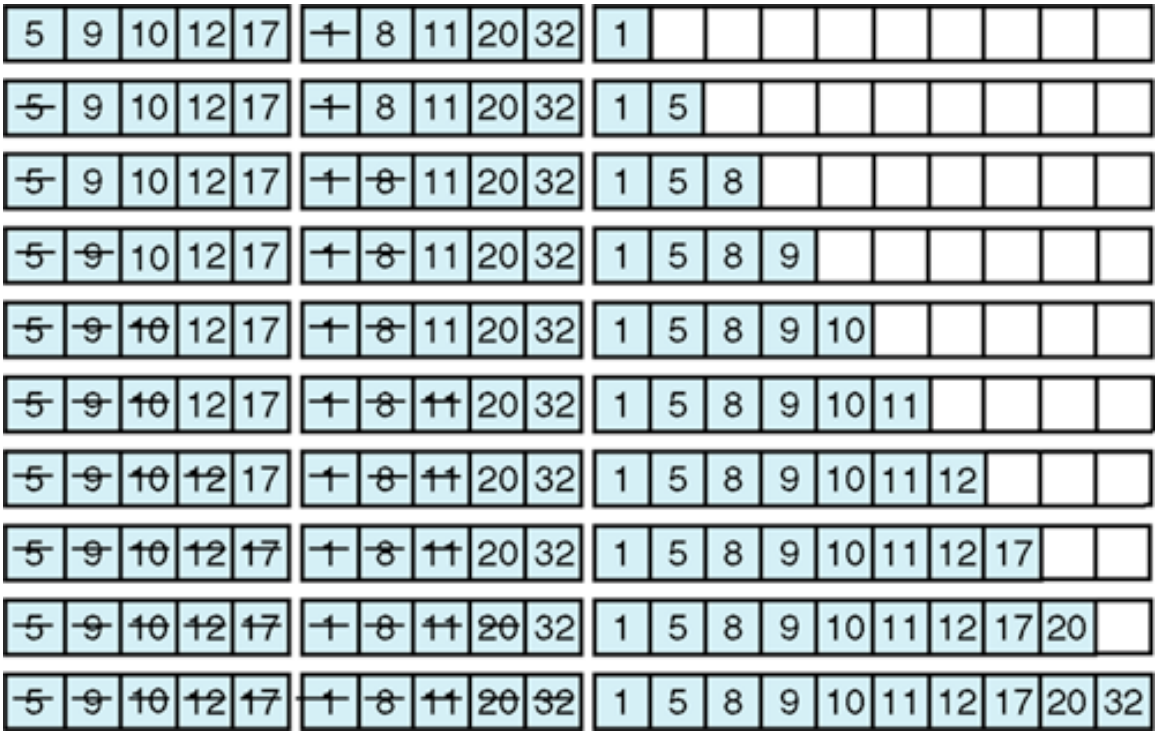
The difference in the two tasks represents the difference between a *sequential*, or *linear* search and a *binary search*. A linear search is basically the approach used in our Bag abstraction in Chapter 8, and the approach that you by necessity need to perform if all you have is an unsorted list. Simply compare the element you seek to each value in the collection, element by element, until either you find the value you want, or exhaust the collection. A binary search, on the other hand, is much faster, but works only for ordered lists. You start by comparing the test value to the element in the middle of the collection. In one step you can eliminate half the collection, continuing the search with either the first half or the last half of the list. If you repeat this halving idea, you can search the entire list in O(log n) steps. As the thought experiment with the telephone book shows, binary search is much faster than linear search. An ordered array of one billion elements can be searched using no more than twenty comparisons using a binary search.

Before a collection can be searched using a binary search it must be placed in order. There are two ways this could occur. Elements can be inserted, one by one, and ordered as they are entered. The second approach is to take an unordered collection and rearrange the values so that they become ordered. This process is termed *sorting*, and you have seen several sorting algorithms already in earlier chapters. As new data structures are introduced in later chapters we will also explore other sorting algorithms.

Fast Set Operations

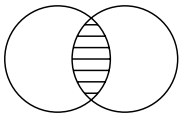
Keeping a dynamic array in sorted order allows a fast search by means of the binary search algorithms. But there is another reason why you might want to keep a dynamic array, or even a linked list, in sorted order. This is that two sorted collections can be *merged* very quickly into a new, also sorted collection. Simply walk down the two collections in order, maintaining an index into each one. At each step select the smallest

value, and copy this value into the new collection, advancing the index. When one of the pointers reaches the end of the collection, all the values from the remaining collection are copied.



The merge operation by itself is sometimes a good enough reason to keep a sorted collection. However, more often this technique is used as a way to provide fast set operations. Recall that the *set* abstraction is similar to a bag, with two important differences. First, elements in a set are unique, never repeated. Second, a set supports a number of collection with collection operations, such as set union, set intersection, set difference, and set subset.

All of the set operations can be viewed as a variation on the idea of merging two ordered collections. Take, for example, set intersection. To form the intersection simply walk down the two collections in order. If the element in the first is smaller than that in the second, advance the pointer to the first. If the element in the second is smaller than that in the first, advance the pointer to the second. Only if the elements are both equal is the value copied into the set intersection.



This approach to set operations can be implemented using either ordered dynamic arrays, or ordered linked lists. In practice ordered arrays are more often encountered, since an ordered array can also be quickly searched using the binary search algorithm.

Implementation of Ordered Bag using an Ordered Array

In an earlier chapter you encountered the *binary search* algorithm. The version shown below takes as argument the value being tested, and returns in O(log n) steps either the location at which the value is found, *or if it is not in the collection* the location the value can be inserted and still preserve order.

```
int binarySearch (TYPE * data, int size, TYPE testValue) {
  int low = 0;
  int high = size;
  int mid;
  while (low < high) {
    mid = (low + high) / 2;
    if (LT(data[mid], testValue))
      low = mid + 1;
    else
      high = mid;
  }
  return low;
}
```

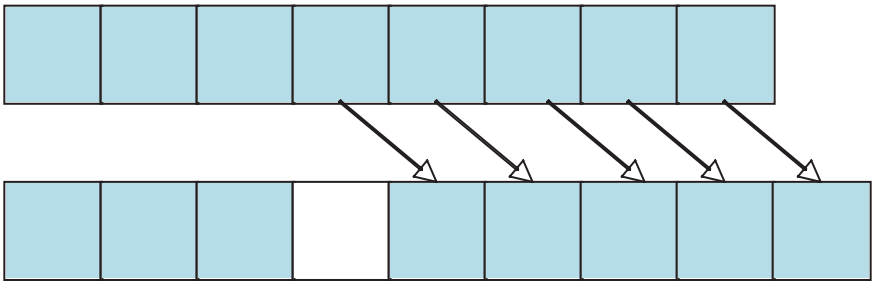
Consider the following array, and trace the execution of the binary search algorithm as it searches for the element 7:

2	4	5	7	8	12	24	37	40	41	42	50	68	69	72

Do the same for the values 1, 25, 37 and 76. Notice that the value returned by this function need not be a legal index. If the test value is larger than all the elements in the array, the only position where an insertion could be performed and still preserve order is the next index at the top. Thus, the binary search algorithm might return a value equal to size, which is not a legal index.

If we used a dynamic array as the underlying container, and if we kept the elements in sorted order, then we could use the binary search algorithm to perform a very rapid contains test. Simply call the binary search algorithm, and save the resulting index position. Test that the index is legal; if it is, then test the value of the element stored at the position. If it matches the test argument, then return true. Otherwise return false. Since the binary search algorithm is O(log n), and all other operations are constant time, this means that the contains test is O(log n), which is much faster than either of the implementations you developed in the preceding chapter.

Inserting a new value into the ordered array is not quite as easy. True, we can discover the position where the insertion should be made by invoking the binary search algorithm. But then what? Because the values are stored in a block, the problem is in many ways the opposite of the one you examined in Chapter 8. Now, instead of moving values down to delete an entry, we must here move values up to make a “hole” into which the new element can be placed:



As we did with remove, we will divide this into two steps. The add function will find the correct location at which to insert a value, then call another function that will insert an element at a given location:

```
void orderedArrayAdd (struct dyArray *dy, TYPE newElement) {
  int index = binarySearch(dy->data, dy->size, newElement);
  dyArrayAddAt (dy, index, newElement);
}
```

The method addAt must check that the size is less than the capacity, calling setCapacity if not, loop over the elements in the array in order to open up a hole for the new value, insert the element into the hole, and finally update the variable count so that it correctly reflects the number of values in the container.

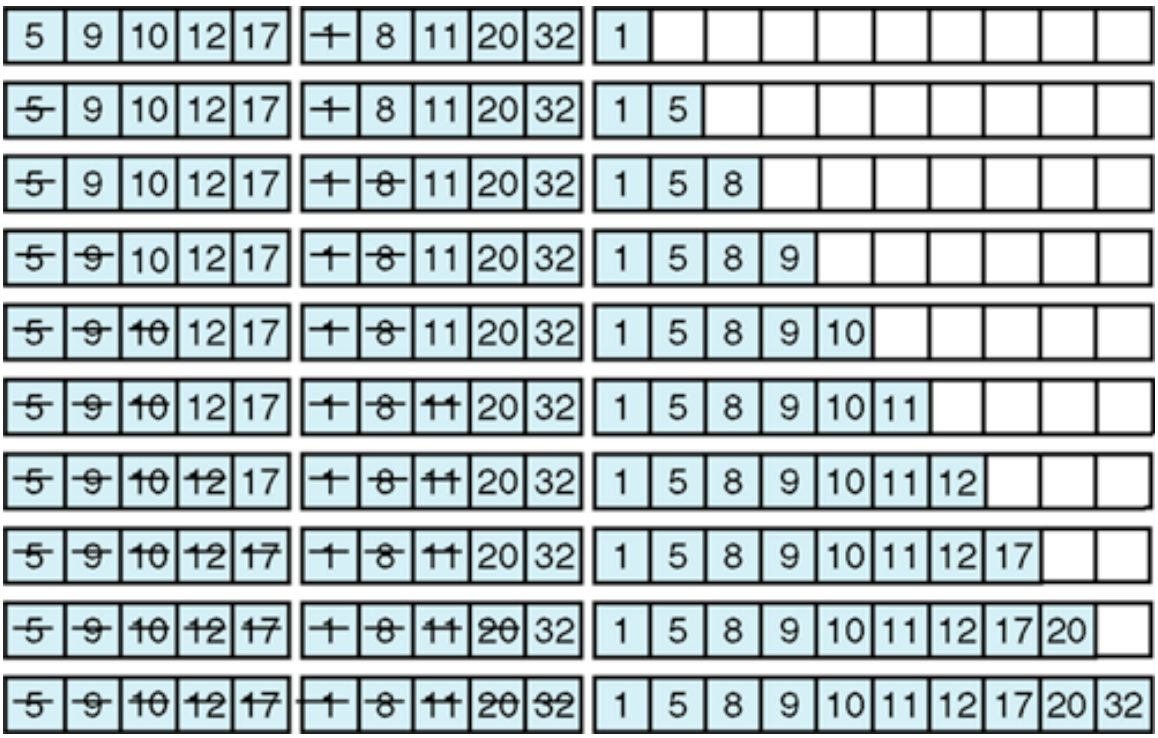
```
void dyArrayAddAt (struct dyArray *dy, int index, TYPE newElement) {
  int i;
  assert(index > 0 && index <= dy->size);
  if (dy->size >= dy->capacity)
          _dyArraySetCapacity(dy, 2 * dy->capacity);
  …  /* you get to fill this in */
}
```

The function remove could use the same implementation as you developed in Chapter 8. However, whereas before we used a linear search to find the position of the value to be deleted, we can here use a binary search. If the index returned by binary search is a legal position, then invoke the method removeAt that you wrote in Chapter 8 to remove the value at the indicated position.

In worksheet 26 you will complete the implementation of a bag data structure based on these ideas.

Fast Set Operations for Ordered Arrays

Two sorted arrays can be easily merged into a new array. Simply walk down both input arrays in parallel, selecting the smallest element to copy into the new array:



The set operations of union, intersection, difference and subset are each very similar to a merge. That is, each algorithm can be expressed as a parallel walk down the two input collections, advancing the index into the collection with the smallest value, and copying values into a new vector.

```
void arraySetIntersect (struct dyArray *left, struct dyArray *right, struct dyArray *to) {
	int i = 0;
	int j = 0;
	while ((i < dyArraySize(left)) && (j < dyArraySize(right))) {
		if (LT(dyArrayGet(left, i), dyArrayGet(right, j)) {
			i++;
		} else if (EQ(dyArrayGet(left, i), dyArrayGet(right, j))) {
			dyArrayAdd(to, dyArrayGet(left, i)); i++; j++;
		} else {
			j++;
		}
	}
}
```

Take, for example, set intersection. The intersection copies a value when it is found in both collections. Notice that in this abstraction it is more convenient to have the set operations create a new set, rather than modifying the arguments. Union copies the smaller element when they are unequal, and when they are equal copies only one value and advances both pointers (remember that in a set all elements are unique, each value appears only once). The difference copies values from the first collection when it is smaller than the current element in the second, and ignores elements that are found in both collections. Finally there is the subset test. Unlike the others this operation does not

produce a new set, but instead returns false if there are any values in the first collection that are not found in the second. But this is the same as returning false if the element from the left set is ever the smallest value (indicating it is not found in the other set).

Question: The parallel walk halts when one or the other array reaches the end. In the merge algorithm there was an additional loop after the parallel walk needed to copy the remaining values from the remaining array. This additional step is necessary in some but not all of the set operations. Can you determine which operations need this step?

In worksheet 27 you will complete the implementation of the sorted array set based on these ideas.

In Chapter 8 you developed set algorithms that made no assumptions concerning the ordering of elements. Those algorithms each have $O(n^2)$ behavior, where n represents the number of elements in the resulting array. What will be the algorithmic execution times for the new algorithms?

	Dynamic Array Set	Sorted Array Set
unionWith	$O(n^2)$	O(
intersectionWith	$O(n^2)$	O(
differenceWith	$O(n^2)$	O(
subset	$O(n^2)$	O(

Set operations Using Ordered Linked Lists

We haven't seen ordered linked lists yet for the simple reason that there usually isn't much reason to maintain linked lists in order. Searching a linked list, even an ordered one, is still a sequential operation and is therefore O(n). Adding a new element to such a list still requires a search to find the appropriate location, and there therefore also O(n). And removing an element involves finding it first, and is also O(n). Since none of these are any better than an ordinary unordered linked list, why bother?

One reason is the same as the motivation for maintaining a dynamic array in sorted order. We can quickly merge two ordered linked lists to produce a new list that is also sorted. And, as we discovered in the last worksheet, the set operations of intersection, union, difference and subset can all be thought of as simple variations on the idea of a merge. Thus we can quickly make fast implementations of all these operations as well.

The motivation for keeping a list sorted is not the same as it was for keeping a vector sorted. With a vector one could use binary search quickly find whether a collection contained a specific value. The sequential nature of a linked list prevents the use of the binary search algorithm (but not entirely, as we will see in a later chapter).

Self Study Questions

1. What two reasons are identified in this chapter for keeping the elements of a collection in sorted order?

2. What is the algorithmic execution time for a binary search? What is the time for a linear search?

3. If an array contains n values, what is the range of results that the binary search algorithm might return?

4. The function dyArrayGet produced an assertion error if the index value was larger than or equal to the array size. The function dyArrayAddAt, on the other hand, allows the index value to be equal to the size. Explain why. (Hint: How many locations might a new value be inserted into an array).

5. Explain why the binary search algorithm speeds the test operation, but not additions and removals.

6. Compare the algorithm execution times of an ordered array to an unordered array. What bag operations are faster in the ordered array? What operations are slower?

7. Explain why merging two ordered arrays can be performed very quickly.

8. Explain how the set operations of union and instersection can be viewed as variations on the idea of merging two sets.

Short Exercises

1. Show the sequence of index values examined when a binary search is performed on the following array, seeking the value 5. Do the same for the values 14, 41, 70 and 73.

2	4	5	7	8	12	24	37	40	41	42	50	68	69	72

Analysis Exercises

1. The binary search algorithm as presented here continues to search until the range of possible insertion points is reduced to one. If you are searching for a specific value, however, you might be lucky and discover it very quickly, before the values low and high meet. Rewrite the binary search algorithm so that it halts if the element examined at position mid is equal to the value being searched for. What is the algorithmic complexity of your new algorithm? Perform an experiment where you search for random values in a given range. Is the new algorithm faster or slower than the original?

2. Provide invariants that could be used to produce a proof of correctness for the binary search algorithm. Then provide the arguments used to join the invariants into a proof.

3. Provide invariants that could be used to produce a proof of correctness for the set intersection algorithm. Then provide the arguments used to join the invariants into a proof.

4. Do the same for the intersection algorithm.

5. Two sets are equal if they have exactly the same elements. Show how to test equality in O(n) steps if the values are stored in a sorted array.

6. A set is considered a subset of another set if all values from the first are found in the second. Show how to test the subset condition in O(n) steps if the values are stored in a sorted array.

7. The binary search algorithm presented here finds the midpoint using the formula (low + high) / 2. In 2006, Google reported finding an error that was traced to this formula, but occurred only when numbers were close to the maximum integer size. Explain what error can occur in this situation. This problem is easily fixed by using the alternative formula low + (high – low) / 2. Verify that the values that cause problems with the first formula now work with the second. (See http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html for a discussion of the bug)

Programming Projects

1. Rewrite the binary search algorithm so that it halts if it finds an element that is equal to the test value. Create a test harness to test your new algorithm, and experimentally compare the execution time to the original algorithm.

2. Write the function to determine if two sets are equal, as described in an analysis exercise above. Write the similar function to determine if one set is a subset of the second.

3. Create a test harness for the sorted dynamic array bag data structure. Then create a set of test cases to exercise boundary conditions. What are some good test cases for this data structure?

On the Web

Wikipedia contains a good explanation of binary search, as well as several variations on binary search. Binary search is also explained in the *Dictionary of Algorithms and Data Structures.*