Worksheet 26: Ordered Bag using a Sorted Array

In Preparation: Read Chapter 8 to learn more about the Bag data type and Chapter 9 to learn more about the advantages of ordered collections. If you have not done it previously, you should review Chapter 2 on the binary search algorithm, and do Worksheets14 and 16 to learn about the basic features of the dynamic array.

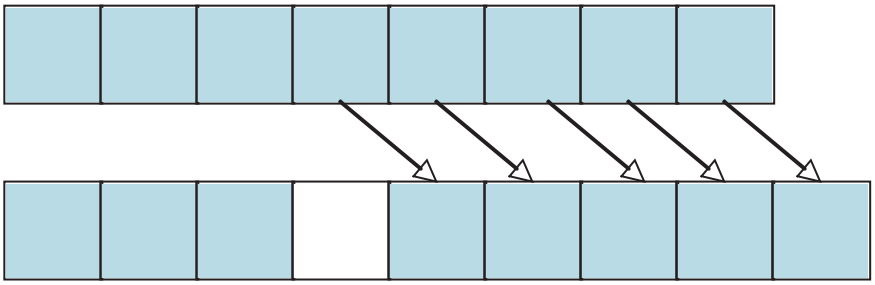
In an earlier lesson you encountered the *binary search* algorithm. The version shown below takes as argument the value being tested, and returns, in O(log n) steps, either the location at which the value is found, *or if it is not in the collection* the location the value can be inserted and still preserve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

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 the bags developed in Worksheets 21 and 22.

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 Worksheet 21. 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 method will find the correct location at which to insert a value, then call another method that will insert an element at a given location:

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

The method addAt must check that the size is less than the capacity, calling dyArrayDoubleCapacity 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 *da, int index, TYPE newElement) {
  int i;
  assert(index >= 0 && index <= da->size);
  if (da->size >= da->capacity)
          _dyArrayDoubleCapacity(da);
  …  /* you get to fill this in */
}
```

The method remove could use the same implementation as you developed in Chapter B. 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 function `dyArrayRemoveAt` that you wrote in Worksheet14 to remove the value at the indicated position.

On Your Own

Fill in the following Chart with the big-oh execution times for the simple unordered dynamic array bag (Lesson X), the linked list bag (Lesson x) and the ordered dynamic array bag (this worksheet).

	Dynamic Array Bag	LinkedListBag	Ordered Array Bag
Add	O(	O(	O(
Contains	O(	O(	O(
Remove	O(	O(	O(

Worksheet 26: Ordered Bag using a Sorted Array Name:

Which operations are faster in an unordered collection? Which operations are faster in an ordered collection? Based on this information, can you think of an example problem where it would be better to use an unordered collection? What about an example problem where it would be better to use an ordered collection?

Short Answers

1. What is the algorithmic complexity of the binary search algorithm?

2. Using your answer to the previous question, what is the algorithmic complex of the method contains for an OrderedArrayBag?

3. What is the algorithmic complexity of the method addAt?

4. Using your answer to the previous question, what is the algorithmic complexity of the method add for an OrderedArrayBag?

5. What is the complexity of the method removeAt? of the method remove?

```
/*These have been written already and you can use them */
void _dyArrayDoubleCapacity (struct dyArray * dy);

void dyArrayInit (struct dyArray * dy, int initialCapacity);
void dyArrayRemoveAt (struct dyArray * dy, int position);
void dyArrayAddAt(struct dyArray *dy, int position, TYPE val);
int _binarySearch(TYPE *data, int size, TYPE testValue);
```

```
/* These are the new functions to take advantage of the ordered dynamic array */

int dyArrayBinarySearch (struct dyArray * dy, TYPE testValue) {
  return _binarySearch (dy->data, dy->size, testValue); }

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

Worksheet 26: Ordered Bag using a Sorted Array Name:

```
    /* you must complete the following */
int orderedArrayContains (struct dyArray *dy, TYPE testElement) {









}
 void orderedArrayRemove (struct dyArray *dy, TYPE testElement) {









}
```