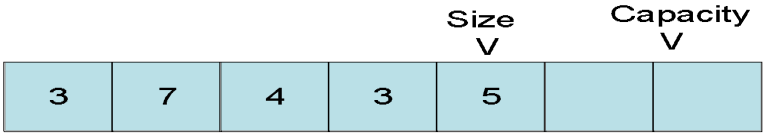
Worksheet 21: Building a Bag using a Dynamic Array

In preparation: Read Chapter 8 to learn more about the Bag data type. If you have not done so already, complete Worksheet 14 to learn about the basic features of the dynamic array.

```
Conceptual Bag interface
void add (TYPE newValue);
Boolean contains (TYPE testValue);
void remove (TYPE testValue);
```

The stack, queue and deque data abstractions are all characterized by maintaining values in the order they were inserted. In many situations, however, it is the values themselves, and not their time of insertion, that is of primary importance. The simplest data structure that is simply concerned with the values, and not their time of insertion, is the Bag. A conceptual definition of the Bag operations is shown at right. In subsequent lessons we will encounter several different implementation techniques for this abstraction. In this lesson we will explore how to create a bag using a dynamic array as the underlying storage area.

Recall that the dynamic array structure maintained three data fields. The first was a reference to an array of objects. The number of positions in this array, held in an integer data field, was termed the *capacity* of the container. The third 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 internal array is created with twice the capacity, and the values are copied from the old array into the new. In Worksheet14 you wrote a routine `_setCapacityDynArr`, to perform this operation.

To add an element to the dynamic array you can simply insert it at the end. This is exactly the same behavior as the function `addDynArray` you wrote in Worksheet 14.

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. Because we want the container to be generalized, we define equality using a macro definition. This is similar to the symbolic constant trick we used to define the type TYPE. The macro is defined as follows:

```
# ifndef EQ
# define EQ(a, b) (a == b)
# endif
```

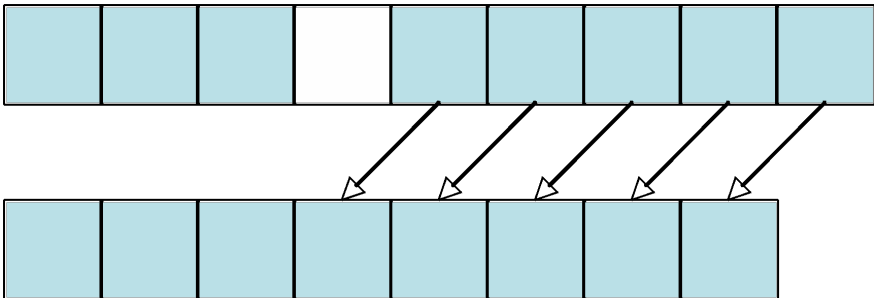
The ifndef preprocessor instruction allows the user to provide an alternative definition for the EQ function. If none is provided, the primitive equality operator will be used.

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 function contains, 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 (from Worksheet 14), that removes the value held at a specific location.

```
void removeDynArr (struct DynArr * da, TYPE test) {
   int i;
   for (i = 0; i < da->size; i++) {
      if (EQ(test, da->data[i])) { /* found it */
         _dynArrayRemoveAt(da, i);
         return;
      }
   }
}
```

Notice two things about the remove function. First, if no matching element is found, the loop will terminate and the function return without making any change to the data. Second, once an element has been found, the function returns. This means that if there were two or more occurrences of the value that matched the test element, only the first would be removed.

The removeAt function takes as argument an index position in the array, and removes the element stored at that location. This is complicated by the fact that when the element is removed, all values stored at locations with higher index values must be “moved down”.



Once the values are moved down, the count must be decremented to indicate the size of the collection is decreased.

Based on these ideas, complete the following skeleton implementation of the bag functions for the dynamic array. You can use any of the functions you have previously written in earlier lessons.

```
struct DynArr {
        TYPE * data;
        int size;
        int capacity;
};

            /* the following were written in earlier lessons */
void initDynArr (struct DynArr * da, int initCap);
void addDynArr(struct DynArr * da, TYPE e);

            /* remove was shown earlier, to use removeAt */
void removeDynArr (struct DynArr * da, TYPE test) {
   int i;
   for (i = 0; i < da->size; i++) {
      if (EQ(test, da->data[i]) { /* found it */
         _dynArrayRemoveAt(da, i);
         return;
      }
   }
}

/* you must write the following */

int containsDynArr (struct DynArr * da, TYPE e) {

            assert(da != 0);
            assert((arraySize(da) > 0);

            for(int i = 0; i < sizeDynArr(da); i++)
            {
                        if(EQ(getDynArr(da, i), e))
                        {
                                    return 1;
                        }
            }
            return 0;

}
```

1. What should the removeAt method do if the index given as argument is not in range?

   The way we wrote removeAt (in Worksheet 14), it includes an assertion to check that the given value is within range (i.e., between zero and the size of the array). If it is given an index that is not in range, the program will abort with an assertion failure error message.

2. What is the algorithmic complexity of the method removeAt? Given your answer to the previous question, what is the worst-case complexity of the method remove?

   The removeAt() function removes a value from a single element, but then must shift all subsequent elements over to fill the gap. Best-base, it is passed the last index, in which case it only needs to operate on one element and its complexity is O(1). However, worst-case, it is passed the first index and needs to operate on every index in an array of size n, so worst-case complexity is O(n).
   The remove method for the Bag ADT must search for the given value before calling removeAt(). Because the array is not sorted, this means using an exhaustive search, which worst-case operates in linear time O(n). Therefore, the algorithmic complexity of the Bag remove function—which includes both the search and removeAt() complexities—is O(n) + O(n), or just O(n).

3. What are the algorithmic complexities of the operations add and contains?

	Best Case	Worst Case	Average Case
Add	O(1) if size < capacity	O(n) if size >= capacity	O(1)+
Contains	O(1) if found at index 0	O(n) as it is linear	O(n)