Worksheet 27: Sorted Array Sets

In Preparation: Read Chapter 8 to learn more about the Set data type. If you have not done so previously, you should do Lesson 26 to learn more about the basic features of the ordered array.

There are two important reasons to keep an array in order. One is so you can use binary search to perform a fast searching operation. We explored this in the previous worksheet when you developed the ordered dynamic array bag. But there is a second reason you might way to keep array elements in sequence. This is because two sorted arrays can be rapidly merged into a third.

But the idea that two arrays can be rapidly merged into a third, as shown in the following picture. Simply walk down both collections in sequence, at each step copying the smallest element into a new array:

![Image of merging two arrays]

We can use this insight to make a much faster Set abstraction. Recall from Chapter 8 that the naive implementations of set union, intersection and difference are each $O(n^2)$. In this lesson you will make a container that is much faster. We will build on the ordered array abstraction you created in Lesson X. The add, contains and remove operations you implemented in Lesson X can be used as before.

Now consider the Set operations. The simplest operation to understand is intersection. To form an intersection simply walk through the two lists, forming a new list that will
contain the intersection. If an element is found in both, add it to the new intersection list. Otherwise if an element is found in only one of the collections, ignore it.

Only slightly more complex is set union. Remember that elements in a union must be unique. If a value is found in only one of the collections add it to the union. If it is found in both, add only one copy to the union. A complication to the union code is that when the first loop finishes you still need to copy the remainder of the elements from the other list to the collection.

A set difference is the elements from one set that are not found in the second. Forming this is similar to intersection.

Finally, to determine if a set is a subset of another walk down both lists in parallel. If you ever find a value that is in the first set but not in the second, then return false. If you finish looping over the first set and have not yet returned false, then return true.
void orderedArrayIntersection (struct dyArray * dy1, struct dyArray * dy2, struct dyArray * dy3) {
    int i = 0;
    int j = 0;
    while (i < dy1->size && j < dy2->size) {
        if (LT(dy1->data[i], dy2->data[j])) {
            i++;
        } else if (EQ(dy1->data[i], dy2->data[j])) {
            dyArrayAdd(dy3, dy1->data[i]);
            i++;
            j++;
        } else {
            j++;
        }
    }
}

void orderedArrayUnion (struct dyArray * dy1, struct dyArray * dy2, struct dyArray * dy3) {
    int i = 0;
    int j = 0;
    while (i < dy1->size && j < dy2->size) {
        if (LT(dy1->data[i], dy2->data[j])) {
            i++;
        } else if (EQ(dy1->data[i], dy2->data[j])) {
            i++;
            j++;
        } else {
            j++;
        }
    }
}
void orderedArrayDifference (struct dyArray * dy1, struct dyArray * dy2, struct dyArray * dy3) {
    int i = 0;
    int j = 0;
    while (i < dy1->size && j < dy2->size) {
        if (LT(dy1->data[i], dy2->data[j])) {
            i++;
        } else if (EQ(dy1->data[i], dy2->data[j])) {
            i++;
            j++;
        } else {
            j++;
        }
    }
}

int orderedArraySubset (struct dyArray * dy1, struct dyArray * dy2) {
    int i = 0;
    int j = 0;
    while (i < dy1->size && j < dy2->size) {
        if (LT(dy1->data[i], dy2->data[j])) {
            i++;
        } else if (EQ(dy1->data[i], dy2->data[j])) {
            i++;
            j++;
        } else {
            j++;
        }
    }
}