CS 162C+
Ordered Arrays

Ordered Arrays

Ordered arrays are similar in all ways to unordered arrays, except that the elements are maintained in a particular order, either from smallest to largest or largest to smallest. For the following examples, we will be considering arrays sorted from largest to smallest.

Adding new items to an Ordered Array

Since the array must be maintained in order, it is necessary to insert any new items in the appropriate slot.

Let's consider the following array of integers, where the count indicates how many items are in the array and the elements are sorted from largest to smallest.

array[0]	33
array[1]	21
array[2]	16
array[3]	12
array[4]	9
array[5]	6
array[6]	3
array[7]	
array[8]	
count	7

If we want to add the number 10, we need to move the numbers 9, 6, and 3 to higher slots to make room as follows:

array[0]	33
array[1]	21
array[2]	16
array[3]	12
array[4]	9
array[5]	9
array[6]	6
array[7]	3
array[8]	
count	8

If we then add the number 10, it should look like this:

array[0]	33
array[1]	21
array[2]	16
array[3]	12
array[4]	10
array[5]	9
array[6]	6
array[7]	3
array[8]	
count	8

Remember that we do not care what happens to be in memory at the locations in the array that are blank.

Coding this functionality results in:

```
addItem (int value)

   int index = 0

   while (array[index] < value)
      array[index+1] = array[index]
      index += 1

   array[index] = value
   count += 1
```

Finding items in an Ordered Array

There is no value in arbitrarily removing an item from an ordered array, they are used primary for find methods. Since the items are sorted, it is possible to use a divide can conquer approach to searching as follows:

Let's consider the following array of integers, where the count indicates how many items are in the array and the elements are sorted from largest to smallest.

array[0]	33
array[1]	21
array[2]	16
array[3]	12
array[4]	10
array[5]	9
array[6]	6
array[7]	3
array[8]	
count	8

Let's search for the number **15**. First we determine that the section being sorted as **array[0]** to **array[count-1]**. Since the number of items is 8, the middle of this is **array[4].**

Array[0]	33	
array[1]	21	
array[2]	16	
array[3]	12	
array[4]	10	←
array[5]	9	
array[6]	6	
array[7]	3	

When we compare, we find that **array[4] is < 15**, so we know that we need to search the ½ of the array with smaller indices. That is, now we are searching from **array[0]** to **array[3].**

array[0]	33	
array[1]	21	
array[2]	16	←
array[3]	12	

Since there are 4 elements, we look at **array[2]** and **16 > 15**, so now we search from **array[3]** to **array[3].**

array[3]	12	←

We compare the value at array[3] and find that **12 != 15,** so the number is not present.

This algorithm can be represented as:

```
binSearch (int value)

   int min = 0
   int max = count

   while (max > min)
      int mid = (min+max)/2

      if (array[mid] == value)
         return true

      if (array[mid] < value])
         min = mid+1

      else
         max = mid-1

   return false
```

Removing items in an Ordered Array

To remove an item from an ordered array, you simply do a search and then compress the array as in the following algorithm:

```
binRemove (int value)

   int min = 0
   int max = count

   while (max > min)
      int mid = (min+max)/2

      if (array[mid] == value)

         for (int i = mid; i < count; i++)
            array[i] = array[i+1]

         count -= 1

         return true

      if (array[mid] < value])
         min = mid+1

      else
         max = mid-1

   return false
```

In this algorithm, the compression loop replaces the return true of the previous function.