61B Note 5: Arrays

Kevin Moy

Abstract

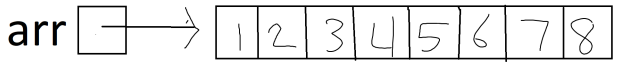
We finally cover our first actual data structure in this note: arrays. They are arguably the most used structure in the programming world today, in its simple usage of storing things in an organized fashion.

1 Arrays: Definition

Remember that **containers** just store things of some type. Arrays, then are a **sequence** of some arbitrary number of containers with the same type. For example, if we have an integer container, an integer array with length 8 is a data structure with 8 of those integer containers in sequence:

1	2	3	4	5	6	7	8

We never work with arrays directly in our code. Instead, we have a **pointer** to the address at the beginning of the array. For example, lets say our pointer to our 8-element array was `arr`. Then we have:



We conceptually treat our array name as the name of its pointer. So to access `arr`'s i-th element (0-indexed), we simply use `arr[i]`. For example, `arr[3]` would return element at index 3, which is `4`.

It's also important to note that **i can actually be any expression that evaluates to an integer**. So `arr[(9+6)/5]` would work, since the expression is evaluated to the integer first. Of course, the evaluated index still must be in range of the array, or you'll get an `ArrayOutOfBoundsException`. The range of all arrays is $[0, arr.length - 1]$, since we zero-index.

2 Arrays In Code

Let's now implement arrays in code. Note that arrays are non-primitive **objects**, so we must declare and allocate space for them in the standard Java syntax with the `new` keyword. For example, to declare our 8-element `arr` above and fill it:

```
int[] arr = new int[8];
for (int i = 1; i < 9; i++) {
    arr[i] = i;
}
```

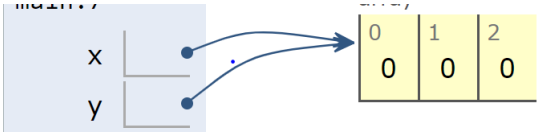
We could also directly add our elements in a one-liner (no for loop/indexing). The new

```
int[] arr = new int[8] {1,2,3,4,5,6,7,8}; // simply
    {1,2,3,4,5,6,7,8} would also work (no new int[8] needed)
```

2.1 Multiple Pointers

Note that arrays are also allowed to have as many pointers as they want! Let's see an example where two pointers point to a single array:

```
int[] x, y;
x = new int[3];
y = x;
```



So now we have two pointers, and utilizing one pointer to edit the array (for example, `y[2] = 100`, would have that change reflected in the other (so printing `x[2]` would also print 100). This should make sense: both pointers point to the *same* object, not different ones.

Note that if we don't **initialize** our array values, the values are all zero by default.

3 Array Insertion

Let's now analyze inserting into arrays in-depth. We already saw insertion by index via `arr[i] = (value)`. That's easy.

Let's now take a step further. What if we wanted to treat the array like a stack? In other words, let's say we want to insert something at index `k`, but then *shift* all higher-indexed elements right? So we'd shift the original `arr[k]` to `arr[k+1]`, `arr[k+1]` to `arr[k+2]`, and so on. Our last element of `arr` would be lost.

Let's write out our skeleton for `insert`. Assume we're working on a `String` array `arr`, although it'll be literally the same logic for arrays of other types.

```
static void insert (String[] arr, int k, String x) {

}
```

So we'll want to insert `x` at `arr[k]`. Game plan: shift all indices $\geq k$ up 1, then insert `x`. We'll want to set each `arr[i]`'s NEW value to the value of the index *before it*- this is a right shift 1.

```
static void insert (String[] arr, int k, String x) {
    for (int i = arr.length-1; i > k; i -= 1)
        arr[i] = arr[i-1];
    arr[k] = x;
}
```

Notice we went *backwards* in our `for` loop, starting from the highest index and right-shifting until we reach index `k`. Why backwards? What if we went forwards, say:

```
static void insert (String[] arr, int k, String x) {
    for (int i = k; i < arr.length; i += 1)
        arr[i+1] = arr[i];
    arr[k] = x;
}
```

Unfortunately, this throws an `ArrayOutOfBoundsException`, because on the last iteration of the for loop we attempt to access `arr[arr.length]`, which

is out of bounds (0-index!!). So we'd actually have to change our for loop condition to `arr.length-1`, which may or may not be unintuitive/annoying. We work backwards because we guarantee we start from an index in-range; it's a lot easier to stay inbounds from that point.

Notice we also returned `void`: our method *destructively* changed `arr` by removing an element and editing others.

4 A Shortcut: System.arraycopy

Java's `System.arraycopy(arr1, i, arr2, j, num)` serves to copy `num` elements starting from (and including) `arr1[i]` into elements starting at `arr2[j]`. Let's see an example:

```
int arr1[] = {0, 1, 2, 3, 4, 5};
int arr2[] = {5, 10, 20, 30, 40, 50};
System.arraycopy(arr1, 0, arr2, 0, 3);
```

would mutate arr2 into `[0,1,2,30,40,50]`, since we copy 3 elements from the start of `arr1` into the start of `arr2`.

5 Array Insertion 2.0

Suppose now that we implement `insert`, but now we actually want to save *all* our elements (not lose the last one). We'd still push everything right. How can we do this?

Well, we now have to actually allocate a *new* array with length `arr.length+1` because we're inserting an element and want to keep everything else. So this just accounts for the worst case where `arr` is full to begin with:

```
static String[] insert(String[] arr, int k, String x) {
    String[] result = new String[arr.length + 1];
    (...)
    return result;
}
```

So now we're creating this *larger* array and returning that as our result.

Our method is no longer destructive- it can't be for arrays. We can't just "tack on" an extra container to arrays (we'd need another special object like `ArrayList` for this). We need to return something new entirely.

Now let's save ourselves the mess of off-by-one and bounds errors with direct indexing and for loops and use our `System.arraycopy` method instead:

```
static String[] insert(String[] arr, int k, String x) {
    String[] result = new String[arr.length + 1];
    arraycopy(arr, 0, result, 0, k);
    arraycopy(arr, k, result, k+1, arr.length-k);
    result[k] = x;
    return result;
}
```

Note that our first array copy simply copies all k elements from `arr` into `result`. Then, our second array copy effectively copies all elements including and ahead of `arr[k]` into the respective right-shifted indices in `result`. Finally, we insert `x` at index `k`, as usual, and return our final inserted array.

6 Implementing Merge

One of the most powerful sorting algorithms ever invented is `MergeSort`. Let's implement the "merging" part of the array, where two **sorted** arrays with lengths k and l are merged to produce a sorted array of length $k + l$. Merging, conceptually, is simple: keep adding the minimum element from the two arrays to the merged array until you run out of elements. This guarantees a sorted array, and is also efficient, because the arrays are already sorted, so it's just comparing `arr1[0]` and `arr2[0]` at each step!

For a recursive implementation, it is useful to generalize the original function to allow merging **subarrays**. Let's look at the code below:

```
static int[] merge(int[] A, int[] B) {
    return mergeTo(A, 0, B, 0, new int[A.length+B.length], 0);
}

/** Merge A[L0:] and B[L1:] into C[K:], assuming A and B sorted. */
static int[] mergeTo(int[] A, int L0, int[] B, int L1, int[] C,
```

```
    int k){
    if (L0 >= A.length && L1 >= B.length) {
        return C;
    } else if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1]))
        {
        C[k] = A[L0];
        return mergeTo(A, L0 + 1, B, L1, C, k + 1);
    } else {
        C[k] = B[L1];
        return mergeTo(A, L0, B, L1 + 1, C, k + 1);
    }
}
```

Let's look at the `mergeTo` recursive helper function in detail. It takes arrays `A` and `B` as well as start-index markers `L0` and `L1` for both arrays. We will recursively call on subarrays of `A` or `B` minus their first element, depending on which array had the minimum element in the initial frame. Once both arrays have "run out of elements", i.e. their index markers have moved past their last index, then we know we've added all elements of A and B to C in a sorted manner. The base case represents this last case, while the other two recursive cases represent the process of finding `min(A,B)` and adding it to C while recursing on appropriately sliced subarrays.

We could also implement `merge` iteratively, *without* a recursive helper function:

```
static int[] merge(int[] A, int[] B) {
    int[] C = new int[A.length + B.length];
    int L0, L1;
    L0 = L1 = 0;
    for (int k = 0; k < C.length; k += 1) {
        if (L1 >= B.length || (L0 < A.length && A[L0] <= B[L1])) {
            C[k] = A[L0];
            L0 += 1;
        } else {
            C[k] = B[L1];
            L1 += 1;
        }
    }
    return C;
```

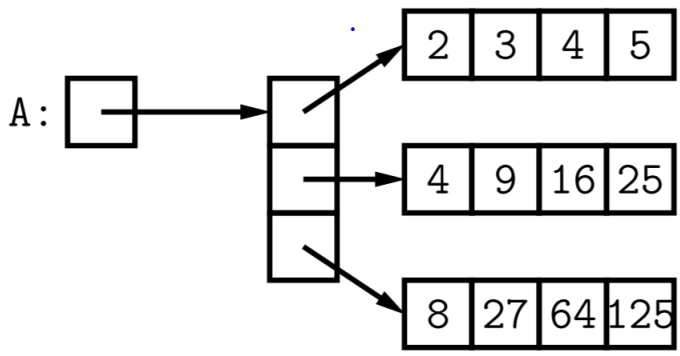
```
}
```

What happens here? Each iteration of the for loop represents finding `min(A,B)` and setting its value at the corresponding index of our merged array. Notice that now `L0` and `L1` markers always just point the the index of the "start" of arrays `A` and `B`- we're "chppping them down" min element by min element, so we need to keep track of what elements in the arrays still need to be considered.

7 Multidimensional Arrays

You can have **multidimensional arrays** in Java. The most common example is the 2D array, which we can simply think of as an array of (pointers to) arrays. Taking the example from class:

```
int[][] A = new int[3][];
A[0] = new int[] {2, 3, 4, 5};
A[1] = new int[] {4, 9, 16, 25};
A[2] = new int[] {8, 27, 64, 125};
```



As you can see above, we declare 2D arrays with two boxes to specify two dimensions instead of one.

There are multiple ways to declare 2D arrays. In our example above, we specify the number in the first box, which indicates how many arrays there

are contained by our outer array (3). However, if we want to add in elements one at a time, we'd have to specify BOTH dimensions (number of arrays AND how many elements in each array) and utilize a nested for loop for element-wise insertion. Elements inserted, then, would have values that follow some function $f(i, j)$. In this case, $f(i, j) = (j + 2)^{i+1}$.

```
int[][] A = new A[3][4];
for (int i = 0; i < 3; i += 1)
    for (int j = 0; j < 4; j += 1)
        A[i][j] = (int) Math.pow(j + 2, i + 1);
```

However, if we're not inserting elements into *specific* indices, the **length of the stored arrays do not actually need to be the same**. Again taking the example from class:

```
int[][] A = new int[5][];
A[0] = new int[] {};
A[1] = new int[] {0, 1};
A[2] = new int[] {2, 3, 4, 5};
A[3] = new int[] {6, 7, 8};
A[4] = new int[] {9};
```

We see the first array has length 0, second has length 2, third length 4, and so on. The moral is that we don't always need to form a rectangle with our 2D arrays.

Note there also exist 3D (and up) arrays, but these are far, far rarer and are only used with conceptual guides, such as storing different-dimension "boards" (project?). You don't need to really understand them in an abstract form.

Notice that accessing and inserting into multidimensional arrays works exactly the same as 1D arrays- you just need extra dimensions to specify the exact container. So to manipulate 2D array `A`, array `i` at index `j` in array `i`, we'd access `A[i][j]`.