**Array Deletions**

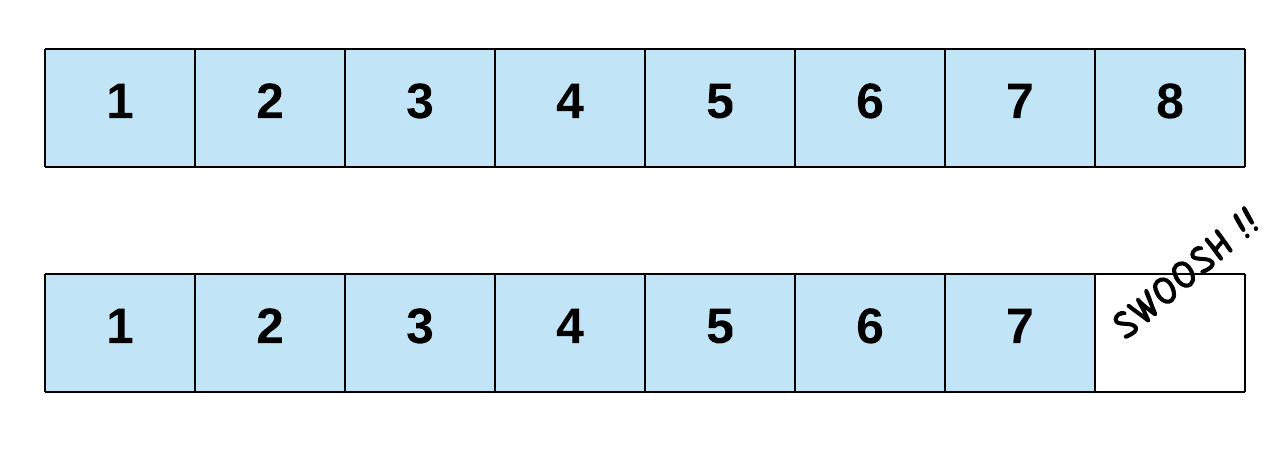
Now that we know how insertion works, it's time to look at its complement—deletion!

Deletion in an Array works in a very similar manner to insertion, and has the same three different cases:

1. Deleting the last element of the Array.
2. Deleting the first element of the Array.
3. Deletion at any given index.

Deleting From the End of an Array

Deletion at the end of an Array is similar to people standing in a line, also known as a queue. The person who most recently joined the queue (at the end) can leave at any time without disturbing the rest of the queue. Deleting from the end of an Array is the least time consuming of the three cases. Recall that insertion at the end of an Array was also the least time-consuming case for insertion.



So, how does this work in code? Before we look at this, let's quickly remind ourselves what the length of an Array means. Here is some code that creates an Array with room for 10 elements, and then adds elements into the first 6 indexes of it.

|  |
| --- |
| // Declare an integer array of 10 elements.  int[] intArray = new int[10];  // The array currently contains 0 elements  int length = 0;  // Add elements at the first 6 indexes of the Array.  for(int i = 0; i < 6; i++) {  intArray[length] = i;  length++;  } |

Notice the length variable. Essentially, this variable keeps track of the next index that is free for inserting a new element. This is always the same value as the overall length of the Array. Note that when we declare an Array of a certain size, we simply fix the maximum number of elements it could contain. Initially, the Array doesn't contain anything. Thus, when we add new elements, we also increment the length variable accordingly.

Anyway, here's the code for deleting the last element of an Array.

|  |
| --- |
| // Deletion from the end is as simple as reducing the length  // of the array by 1.  length--; |

Remember how insertion we were using this printArray function?

|  |
| --- |
| for (int i = 0; i < intArray.length; i++) {  System.out.println("Index " + i + " contains " + intArray[i]);  } |

Well, if we run it here, we'll get the following result, regardless of whether we run it before or after removing the last element.

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

Index 3 contains 3.

Index 4 contains 4.

Index 5 contains 5.

Index 6 contains 0.

Index 7 contains 0.

Index 8 contains 0.

Index 9 contains 0.

What's gone wrong? Well, remember how there's two different definitions of length? When we use intArray.length, we're looking *every* valid index of the Array. When in fact, we only want to look at the ones that we've put values into. The fix is easy, we just iterate up to our own length variable instead.

|  |
| --- |
| for (int i = 0; i < length; i++) {  System.out.println("Index " + i + " contains " + intArray[i]);  } |

Run this, and you'll get the following before the deletion:

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

Index 3 contains 3.

Index 4 contains 4.

Index 5 contains 5.

And the following after:

Index 0 contains 0.

Index 1 contains 1.

Index 2 contains 2.

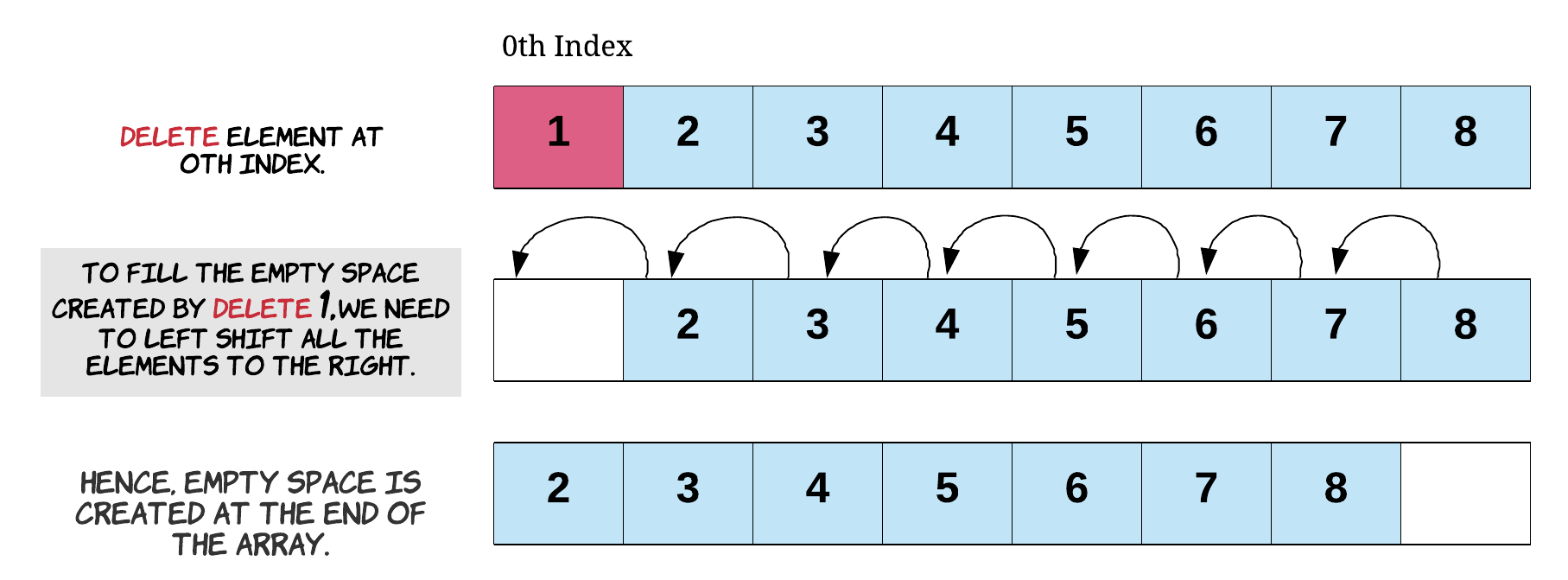
Index 3 contains 3.

Index 4 contains 4.

Yup, that's it! Even though we call it a deletion, its not like we actually freed up the space for a new element, right? This is because we don't actually need to free up any space. Simply overwriting the value at a certain index deletes the element at that index. Seeing as the length variable in our examples tells us the next index where we can insert a new element, reducing it by one ensures the next new element is written over the deleted one. This also indicates that the Array now contains one less element, which is exactly what we want programmatically.

Deleting From the Start of an Array

Next comes the costliest of all deletion operations for an Array—deleting the first element. If we want to delete the first element of the Array, that will create a vacant spot at the 0th index. To fill that spot, we will shift the element at index 1 one step to the left. Going by the ripple effect, every element all the way to the last one will be shifted one place to the left. This shift of elements takes O(N)*O*(*N*) time, where N*N* is the number of elements in the Array.



Here is how deleting the first element looks in code.

|  |
| --- |
| // Starting at index 1, we shift each element one position  // to the left.  for (int i = 1; i < length; i++) {  // Shift each element one position to the left  int\_array[i - 1] = int\_array[i];  }  // Note that it's important to reduce the length of the array by 1.  // Otherwise, we'll lose consistency of the size. This length  // variable is the only thing controlling where new elements might  // get added.  length--; |

Starting from index 0, we'll move every element one position to its left, effectively "deleting" the element at index 0. We also need to reduce length by 1 so that the next new element is inserted in the correct position.

And here's the output we'll get, with our updated printArray function.

Index 0 contains 1.

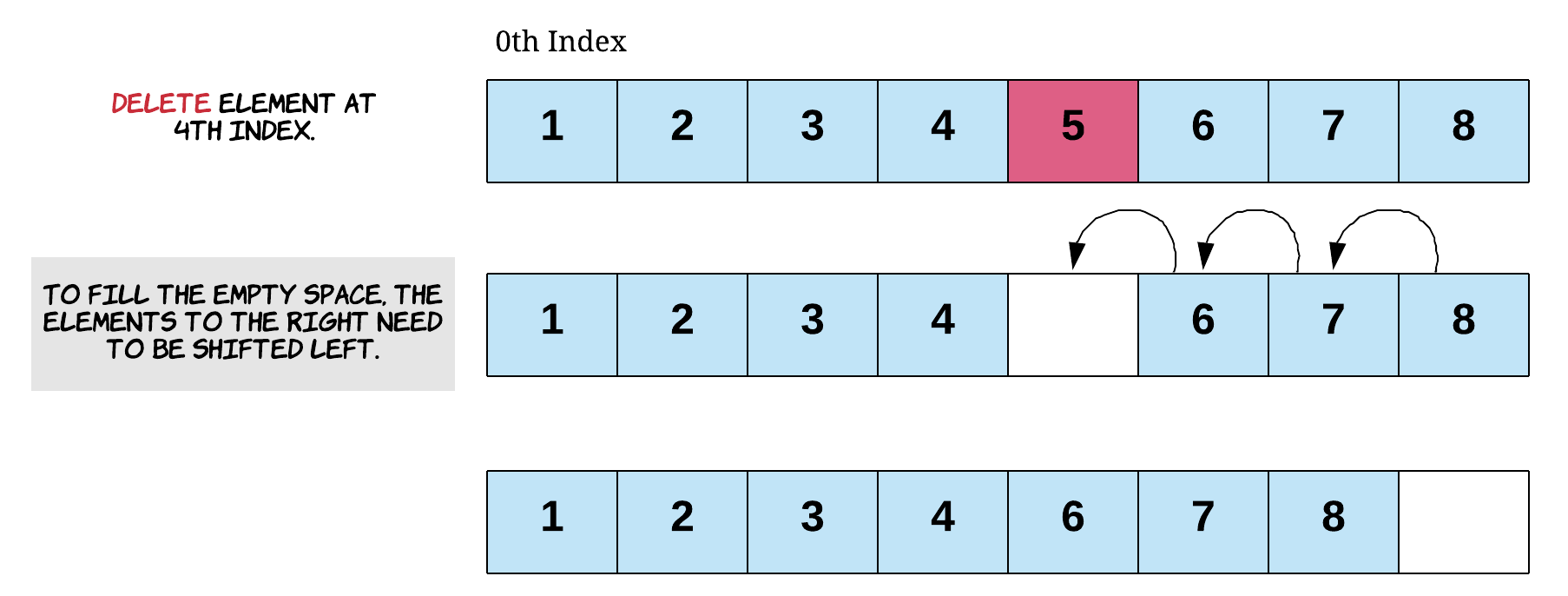
Index 1 contains 2.

Index 2 contains 3.

Index 3 contains 4.

Deleting From Anywhere in the Array

For deletion at any given index, the empty space created by the deleted item will need to be filled. Each of the elements to the *right* of the index we're deleting at will get shifted to the *left* by one. Deleting the first element of an Array is a special case of deletion at a given index, where the index is 0. This shift of elements takes O(K)*O*(*K*) time where K*K* is the number of elements to the right of the given index. Since *potentially* K = N*K*=*N*, we say that the time complexity of this operation is also O(N)*O*(*N*).



Here is the code to delete the element at index 1. To do this, we'll need to move over the elements after it in the Array.

|  |
| --- |
| // Say we want to delete the element at index 1  for (int i = 2; i < length; i++) {  // Shift each element one position to the left  int\_array[i - 1] = int\_array[i];  }  // Again, the length needs to be consistent with the current  // state of the array.  length--; |

Notice that this works exactly like deleting the first element, except that we don't touch the elements that are at *lower* indexes than the element we're deleting.

Here is the output from the printArray function.

Index 0 contains 1.

Index 1 contains 3.

Index 2 contains 4.

Did that all make sense? To help you cement what you've learned, here's a couple of programming problems for you to try. You should try to solve them *without* making a new Array. Do this by using the deletion techniques we've investigated above.

Once you're done, we'll look at searching Arrays!