Circular Arrays

Step 1

Imagine we have a set of elements we want to store in some data structure. The elements don't necessarily have a particular order (they might, but we don't know in advance). In the previous sections of the text, we learned about two data structures we could theoretically use to store our elements: **Array Lists** and **Linked Lists**.

We learned that **Array Lists** are great if we want to be able to randomly access elements *anywhere* in my dataset. In terms of inserting elements, however, they do fine if we're always adding to the end of the list. If we want to insert elements at the beginning of the list however, we have to waste time moving other elements to make room for the insertion.

Linked Lists, on the other hand, are great at inserting elements to the beginning or end of the list. If we want to access elements in the middle however, we have to waste time iterating through the elements until we reach the desired element.

Is there any way we can create a new data structure that allows us to enjoy some of the perks of both data structures? In this section, we will discuss the **Circular Array**, our attempt at reaping the benefits of both **Array Lists** and **Linked Lists**.

Step 2

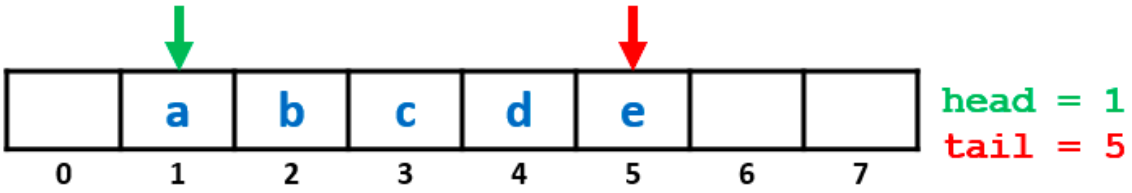
At the lowest level, a **Circular Array** is really just a regular **Array List** with a clever implementation: we will try to mimic the implementation of a **Linked List**. Recall that a **Linked List** has a *head* pointer and a *tail* pointer, and when we want to insert an element to the beginning or end of a **Linked List**, all we do is update a few pointers (which is a constant-time operation).

What if, similar to *head* and *tail* pointers in a **Linked List**, we take our **Array List** and use *head* and *tail* indices? The first element would be the element at the *head* index, and the last element would be the element at the *tail* index. As we add to the tail of the **Circular Array**, we can simply increment *tail*, and likewise, as we add to the head of the **Circular Array**, we can simply decrement *head*. As we increment *tail*, if we ever go out of bounds (i.e., *tail* becomes equal to the size of the array), we can simply wrap around to index 0. Likewise, as we're decrementing *head*, if we ever go out of bounds (i.e., *head* becomes -1), we can simply wrap around to the last index of the array (i.e., *size*–1).

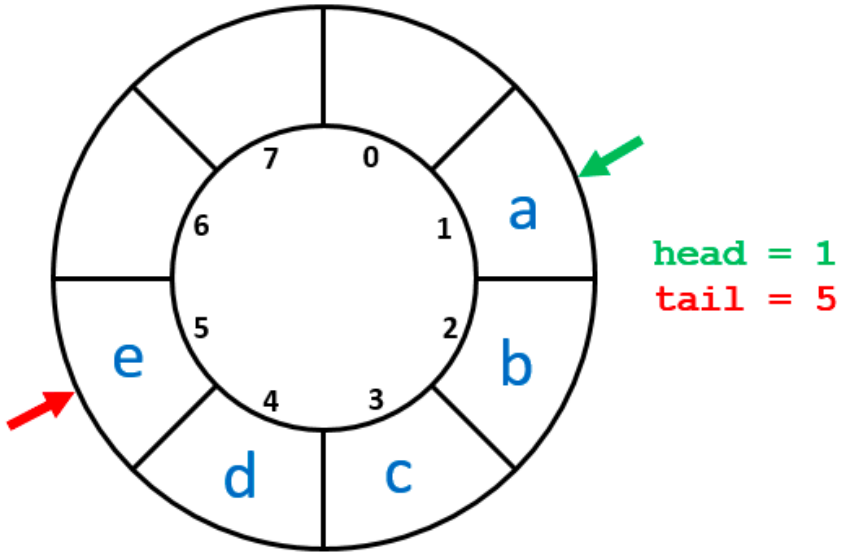
Note that, just like with an **Array List**, the elements of our **Circular Array** will be contiguous in the backing array. The reason for this is that we will only allow users to insert to the *head* or the *tail* of the **Circular Array**.

Step 3

Below is an example of a **Circular Array** containing 5 elements but with a capacity of 8 elements. The backing array's indices are written below the backing array in black, and the *head* and *tail* indices have been written and are also drawn onto the figure with arrows.



When adding/removing elements, we only really care about elements' locations with respect to the *head* and *tail* indices, not with respect to the actual indices in the backing array. Because of this, as well as because of the fact that the "wrap around" performed by both *head* and *tail* is a constant-time operation, many people prefer to picture the **Circular Array** as exactly that: a circular array. Below is a circularized representation of the same example shown above.



You can choose either representation (linear or circular) to visualize the **Circular Array** in your own mind: both representations are equally valid. The linear representation is closer to the implementation "truth", whereas the circular representation better highlights the purpose of the structure.

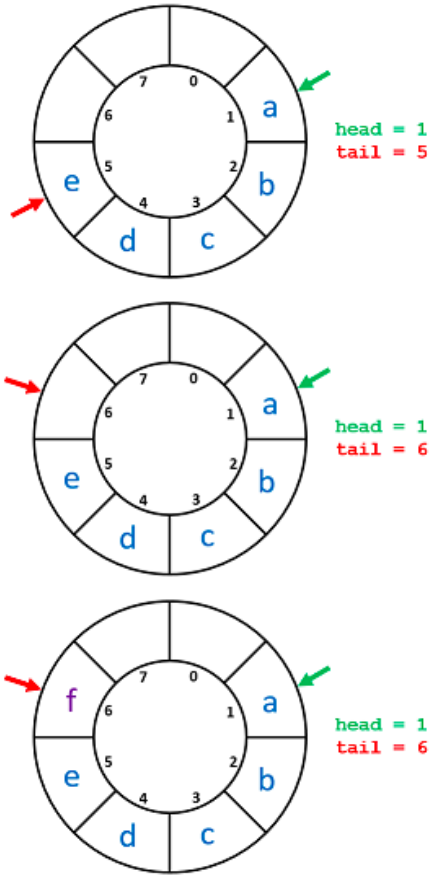
Step 4

EXERCISE BREAK: You are given an arbitrary **Circular Array** with a capacity of 8 that contains 3 elements. Which of the following pairs of *head* and *tail* indices could be valid? Assume indices are 0-based (i.e., the first index of the backing array is 0 and the last index of the backing array is 7). (Select all that apply)

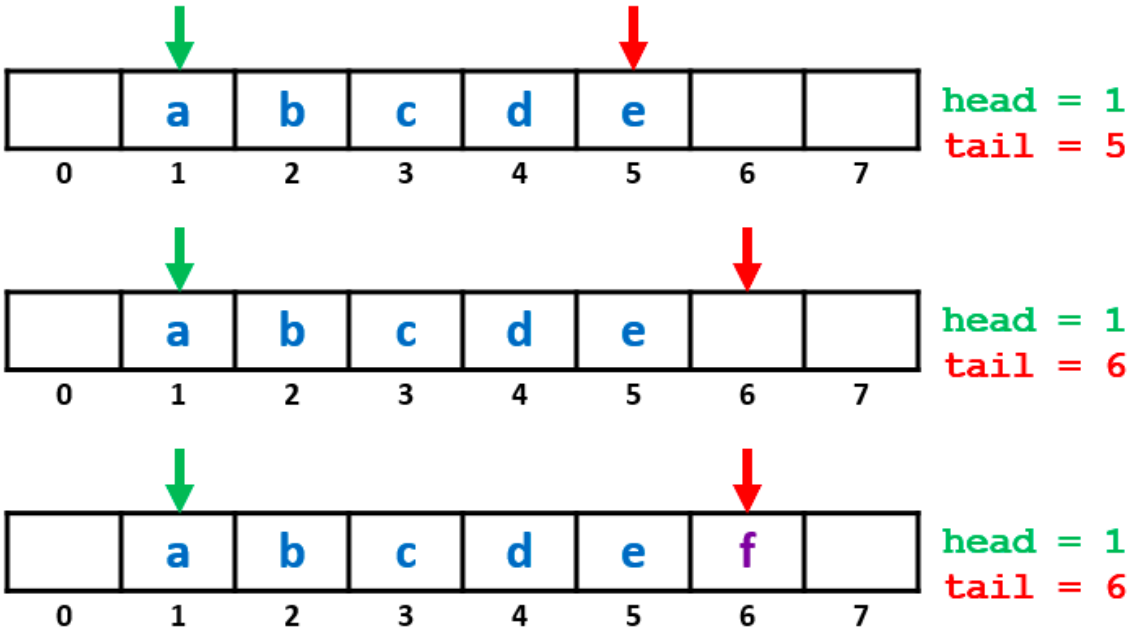
To solve this problem please visit https://stepik.org/lesson/28869/step/4

Step 5

Below is an example of insertion at the end of a **Circular Array** with some elements already in it:

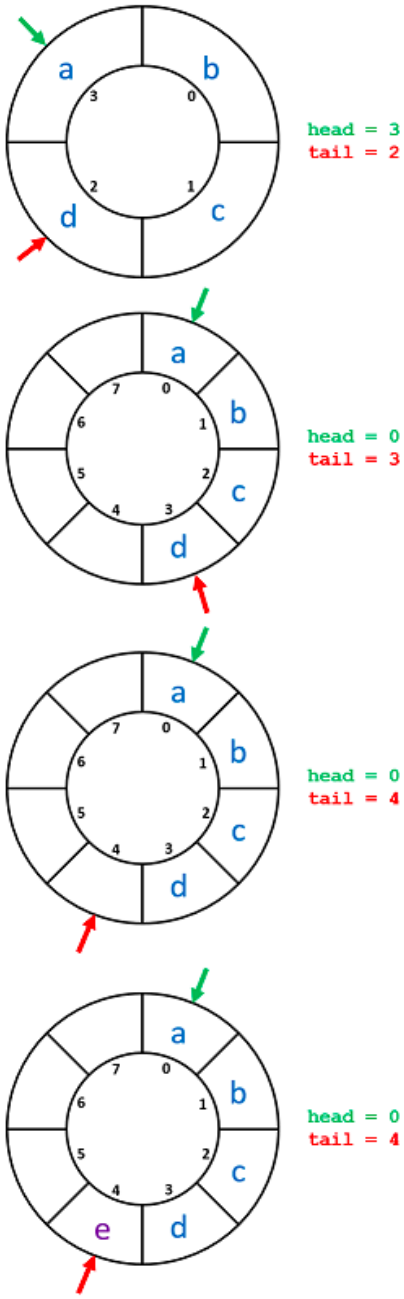


Below is the same example, but in the linear representation:

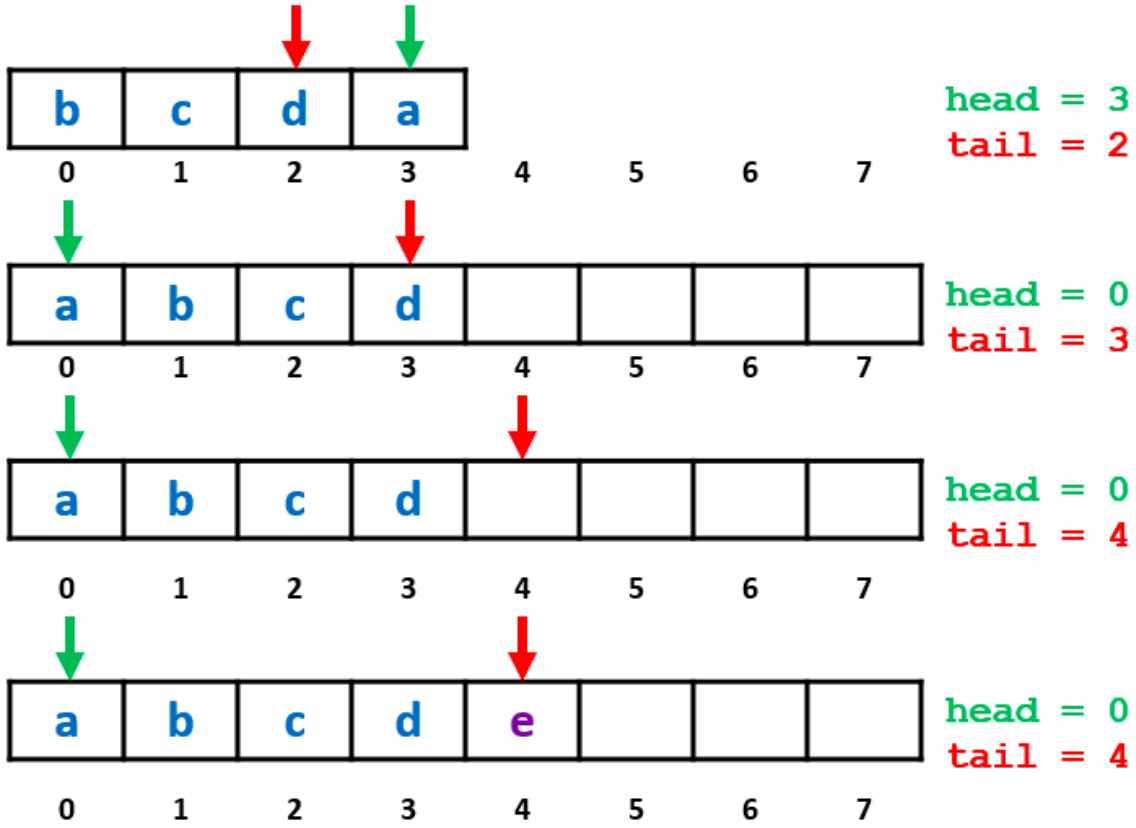


Step 6

Note that, just like in the **Array List** section, if the backing array becomes full, we can create a new backing array (typically of twice the size) and simply copy all elements from the old backing array into the new backing array. To ensure that our elements stay in the same order, it's best to set them to indices 0 through *n*-1 in the new array. Below is an example of insertion at the end of a full **Circular Array**:



Below is the same example, but in the linear representation:



Step 7

We briefly discussed the insertion algorithm at a higher level, but let's formalize it. In the following pseudocode, assume our **Circular Array** has a backing array named `array` (which has length `array.length`), an integer `n` that keeps track of the number of elements added to the **Circular Array**, and two indices `head` and `tail`. For our purposes, let's assume you can only add elements to the front or back of the **Circular Array** to keep the problem relatively simple.

```
insertFront(element): // inserts element at the front of the Circular Array
    // check array size
    if n == array.length:
        newArray = empty array of length 2*array.length
        for i from 0 to n-1:       // copy all elements from array to newArray
            newArray[i] = array[(head+i)%array.length]
        array = newArray           // replace array with newArray
        head = 0                   // fix head and tail indices
        tail = n-1

    // insertion algorithm
    head = head - 1                // decrement head index
    if head == -1:                 // if we went out of bounds, wrap around
        head = array.length-1
    array[head] = element          // perform insertion
    n = n + 1                      // increment size
```

```
insertBack(element): // inserts element at the back of the Circular Array
    // check array size
    if n == array.length:
        newArray = empty array of length 2*array.length
        for i from 0 to n-1:       // copy all elements from array to newArray
            newArray[i] = array[(head+i)%array.length]
        array = newArray           // replace array with newArray
        head = 0                   // fix head and tail indices
        tail = n-1

    // insertion algorithm
    tail = tail + 1                // increment tail index
    if tail == array.length:       // if we went out of bounds, wrap around
        tail = 0
    array[tail] = element          // perform insertion
    n = n + 1                      // increment size
```

STOP and Think: How could we generalize this idea to allow for insertion into the middle of the **Circular Array**?

Step 8

EXERCISE BREAK: What is the worst-case time complexity for an "insert" operation at the front or back of a **Circular Array**?

To solve this problem please visit https://stepik.org/lesson/28869/step/8

Step 9

EXERCISE BREAK: What is the worst-case time complexity for an "insert" operation at the front or back of a **Circular Array**, given that the backing array is not full?

To solve this problem please visit https://stepik.org/lesson/28869/step/9

Step 10

EXERCISE BREAK: What is the worst-case time complexity for an "remove" operation at the front or back of a **Circular Array**?

To solve this problem please visit https://stepik.org/lesson/28869/step/10

Step 11

Removal at the front or back of a **Circular Array** is fairly trivial. To remove from the front of a **Circular Array**, simply "erase" the element at the *head* index, and then increment the *head* index (wrapping around, if need be). To remove from the back of a **Circular Array**, simply "erase" the element at the *tail* index, and then decrement the *tail* index (wrapping around, if need be).

```
removeFront(): // removes the element at the front of the Circular Array
    erase array[head]
    head = head + 1
    if head == array.length:
        head = 0
    n = n - 1
```

```
removeBack(): // removes the element at the back of the Circular Array
    erase array[tail]
    tail = tail - 1
    if tail == -1:
        tail = array.length - 1
    n = n - 1
```

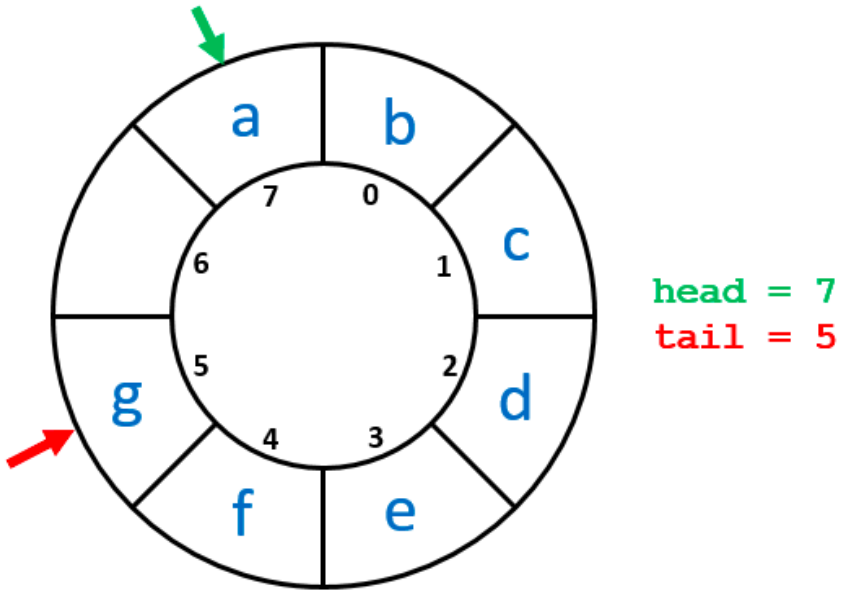
STOP and Think: Is it necessary for us to perform the "erase" steps in these algorithms? What would happen if we didn't?

Step 12

So far, we've only looked at adding or removing elements from the front or back of a **Circular Array**, which is no better than adding or removing elements from the front or back of a **Linked List**. So what was the point of all this added complexity? Recall that, with a **Linked List**, if we want to access an element somewhere in the middle of the list, even if we know which index of our list we want to access, we need to perform an O(*n*) iteration to follow pointers and reach our destination. However, with a **Circular Array**, the backing structure is an **Array List**, meaning we have random access to any element given that we know which index of the backing array we need to query.

If we want to access the element of a **Circular Array** at index *i*, where *i* is with respect to the *head* index (i.e., *head* is *i* = 0, irrespective of what index it is in the backing array), we can simply access the element at index ***(head + i) % array.length*** in the backing array. By modding by the backing array's length, we ensure that the index we get from the operation (*head* + *i*) wraps around to a valid index if it exceeds the boundaries of the array.

Below is an example of a **Circular Array** where the *head* index wrapped around. Notice that, if we consider *head* to be "*i* = 0" of our elements (even though it's index 7 of the backing array), we can access any *i*-th element of our list via the formula above. For example, the element at *i* = 2 of our list is at index (7 + 2) % 8 = 9 % 8 = 1 of the backing array, which corresponds to c.



STOP and Think: Under what condition(s) would we be safe to omit the mod operation in the formula above? In other words, under what condition(s) would the formula *head* + *i* be valid?

Step 13

Previously, we discussed **Array Lists** and **Linked Lists**, and now, we've also introduced and analyzed the **Circular Array**, a data structure that is based on an **Array List**, but that attempts to mimic the efficient front/back insertion/removal properties of a **Linked List**.

In the next sections, we will explore some real-life applications in which we only care about adding/removing elements from the front/back of a list, which will give purpose to our exploration of **Linked Lists** and **Circular Arrays**.