Circular arrays in Java

Table of contents

Introduction

In a ``normal'' *linear* array, the first available slot is at index 0, the second one at index 1, and so on until the last one at index `arr.length - 1` (where `arr` is a reference to the array in question). We cannot start before the first slot at index 0, and we must stop at the last slot at index `arr.length - 1` — any attempt to access a slot at `index < 0` or at `index >= arr.length` will cause an `ArrayIndexOutOfBoundsException` to be thrown.

We iterate over the elements the array in the usual way by advancing a cursor from the first slot to the last. We iterate backwards in a similar way.

```
for (int i = 0; i < size; i++)
    visit(arr[i]);

for (int i = size - 1; i >= 0; i--)
    visit(arr[i]);
```

where `size ≤ arr.length` is the number of elements in the linear array.

In a *circular* array (also called *circular* buffer), the end of the array **wraps around** to the start of the array, just like in musical chairs. Simply watch the second hand of a watch go from `0` to `59`, and then wrap around to `0` again, and you see a circular array at work. You should note that we create the **perception** of a circular array — the underlying data is still kept in a linear array with a start (at index `0`) and an end (at index `arr.length - 1`). We implement the circularity by manipulating the indices such that when we advance past the end of the underlying array, the cursor wraps around to the beginning again. Similarly, when we go past the beginning going backwards, the cursor wraps around to the end of the array. Basically, we ``fake'' the concept of a circular array on top of a ``normal'' linear array.

The following shows a linear array with a capacity of 8, and with 6 elements in it. The start index is assumed to be `0`. The next available slot is at index `size` or `6`, given by `start + size = 0 + 6 = 6`. The last element is in the slot at index `size - 1` or `5`, given by `start + size - 1 = 0 + 6 - 1 = 5`.

```
  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 5 |  7 |  9 |  15 | -4 |  17 |    |   |   size     = 6
-----------------------------------------
                                 ^-- nextAvailPos = size
```

In an equivalent circular array of the same capacity and with the same elements, the first element may start at any index, not necessarily at index `0`. For example, the following circular arrays are both equivalent to each other, and to the linear array shown above.

```
  0    1    2    3    4     5    6   7
-----------------------------------------   capacity = 8
| 5 |  7 |  9 |  15 | -4 |  17 |   |   |   size     = 6
-----------------------------------------
  ^-- start                      ^-- nextAvailPos

  0    1    2    3    4     5    6   7
-----------------------------------------   capacity = 8
|   | 5  |  7 |  9  | 15 | -4  | 17|   |   size     = 6
-----------------------------------------
      ^-- start                      ^-- nextAvailPos

  0    1    2    3    4     5    6   7
-----------------------------------------   capacity = 8
| 15| -4 | 17 |     |    | 5   | 7 | 9 |   size     = 6
-----------------------------------------
 nextAvailPos --^          ^-- start
```

Back to Table of contents

Moving a cursor forward and backward

Before going to iteration, let's take a look at how to advance an cursor (in both directions) in linear and circular arrays. In a linear array, moving an cursor `i` forward and backward by unit stride are simply:

Forward

```
i++, or
i = i + 1, or
i += 1
```

Backward

```
i--, or
i = i - 1, or
i -= 1
```

With the understanding that `i < 0` and `i ≥ arr.length` are invalid indices.

In a circular array, moving moving an cursor `i` forward (or advancing the cursor) is accomplished by first advancing it normally, and then wrapping it if necessary.

```
i++;
if (i == arr.length)
    i = 0;
```

It can also be done using modular arithmetic, by using the modulus operator (`%`) in Java.

```
i++;
i = i % arr.length;
```

Or, by combining the two statements in one.

```
i = (i + 1) % arr.length;
```

Moving backwards however cannot be done using the modulus operator, so we have to wrap it explicitly, if needed.

```
i--;
if (i == -1)
    i = arr.length - 1;
```

We can of course advance the cursor by an `offset > 1`. For a linear array, we can advance a cursor, forward and backward, by a given `offset` in the following way.

Forward

```
i = i + offset
(i >= arr.length will be invalid)
```

Backward

```
i = i - offset
(i < 0 will be invalid)
```

For a circular array, no cursor position is invalid, since it simply wraps around in either direction.

Forward

```
i = i + offset
i = i % arr.length;
```

Backward

```
i = i - offset
if (i < 0)
    i = i + arr.length;
```

(please convince yourself of this before moving on ... There is also a bug in the backward direction code given above when offset is larger than the array's length).

Back to Table of contents

Iteration over the elements in a circular array

Now that we know how to move a cursor through a circular array in both directions, we can now iterate over the elements knowing the start index and the number of elements in the circular array. When iterating over the elements in a circular array, it's easier to use two different variables — one as just a counter that goes `size` (where `size` ≤ `arr.length`) times, and other is the actual cursor into the underlying array. For a circular array with the starting position at index `start`, we can iterate as follows.

Forward

```
int k = start;
for (int i = 0; i < size; i++) {
    visit(arr[k]);
    // advance k, wrapping if necessary
    k = (k + 1) % arr.length;
}
```

Backward

```
// Find the index of the last
// element in the circular array
int k = (start + size - 1) % arr.length;
for (int i = 0; i < size; i++) {
    visit(arr[k]);
    // move k backwards,
    // wrapping if necessary
    k--;
    if (k == -1)
        k = arr.length - 1;
}
```

For the reverse iteration, we need to find the index of the last element in the circular array, which can be accomplished by the following.

```
lastPos = (start + size - 1) % arr.length;
```

Similarly, the index of the next available position is the following.

```
nextAvailPos = (start + size) % arr.length;
```

Back to Table of contents

Linearizing a circular array

Linearizing a circular produces a linear array where the first element is at index `0`, and the last element is at index `size - 1`. It's basically copying the circular array into a linear one, element by element, such that `circArr[start] → linearArr[0]`, `circArr[start + 1] → linearArr[1]`, `circArr[start + 2] → linearArr[2]`, and so on. The following shows the linearized version of a circular array (the input circular array is at the top, with its linearized version at the bottom).

```
  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 15| -4 | 17 |    |    | 5   | 7 | 9 |   size     = 6
-----------------------------------------
 nextAvailPos --^          ^-- start

  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 5 |  7 |  9 |  15 | -4 |  17 |   |   |   size     = 6
-----------------------------------------
  ^-- start                    ^-- nextAvailPos
```

The returned linearized array is a copy of the circular array, with its first element at index `0`. Since we already know how to iterate, it's actually quite simple.

```
/**
 * Returns a linearized copy of the specified circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @return a linear array with the elements in the circular array
 */
public static Object[] linearize(Object[] circArr, int start, int size) {
    Object[] linearArr = new Object[size];
    int k = start;
    for (int i = 0; i < size; i++) {
        linearArr[i] = circArr[k];
        k = (k + 1) % circArr.length;
    }
    return linearArr;
}
```

Back to Table of contents

Resizing a circular array

Resizing a circular array to have a larger capacity, and maintaining the existing elements, is almost exactly the same as linearizing it! The following shows the resized version of a circular array (the input circular array is at the top, with its resized version at the bottom).

```
  0    1    2    3    4    5    6   7
-----------------------------------------   capacity = 8
| 15| -4 | 17 |    |    | 5   | 7 | 9 |   size     = 6
-----------------------------------------
 nextAvailPos --^          ^-- start

  0    1    2    3    4    5    6   7   8   9
-------------------------------------------------   capacity = 10
| 5 |  7 |  9 |  15 | -4 |  17 |   |   |   |   |   size     = 6
-------------------------------------------------
```

```
       ^-- start                      ^-- nextAvailPos
```

The returned resized array is a copy of the circular array, with its first element at index `0`. The code is shown below.

```
  /**
   * Returns a resized copy of the specified circular array.
   * @param circArr the circular array
   * @param start   the index of the first element in the circular array
   * @param size    the number of elements (must be ≤ circArray.length)
   * @param newCap  the new capacity
   * @return a resized copy with the elements in the circular array
   */
  public static Object[] resize(Object[] circArr, int start, int size,
        int newCap) {
      Object[] resizedArr = new Object[newCap];
      int k = start;
      for (int i = 0; i < size; i++) {
          resizedArr[i] = circArr[k];
          k = (k + 1) % circArr.length;
      }
      return resizedArr;
  }
```

Back to Table of contents

Inserting an element in a circular array

Let's say you want to insert a new element at a paritcular position in a circular array. In a linear array, the position is equivalent to the index where the new element would be inserted. In a circular array, the position is **not the index**, but rather it is the **offset** from the **front** element. You can compute the index into the underlying array using the by-now-familiar relation `(front + offset) % circArr.length`.

Let's see the insertion in action using the following example — we insert a new element `13` at position `3`, which results in the circular array shown at the bottom.

```
Insert element 13 in position 3, or at index (5 + 3) % 8 = 0.

       0    1    2    3    4     5    6   7
     ----------------------------------------   capacity = 8
     | 15| -4 | 17 |     |    | 5   | 7 | 9 |   size     = 6
     ----------------------------------------
       ^                        ^-- start
       ^--- this is position 3 relative to front

And the resulting circular array after insertion is show below.

       0    1    2    3    4     5    6   7
     ----------------------------------------   capacity = 8
     |13 | 15 | -4 | 17  |    | 5   | 7 | 9 |   size     = 7
     ----------------------------------------
                              ^-- start
```

Inserting into a circular array is no different that inserting into a linear array — we need to **shift** elements one position to the **right** to create a "hole" for the new element. The only difference is how we shift the elements in linear vs circular array. In either case, to shift elements, we need to know the following:

1. the **index** where to start shifting, and
2. the **number of elements** to shift.

As we noted earlier, in the case of a circular array, we are often given the **offset** from the **front** element (which is exactly what the index is for a linear array!), so we have to first compute the index where to start the shift. Once we know that, the process is exactly the same! The **number of elements** to shift is given by `size - offset` (which is exactly the same for a linear array!).

```
/**
 * Inserts an element at the given position in a circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @param elem    the new element to insert
 * @param pos     the offset of the new element relative to start
 * @return index of the new element just inserted
 */
public static int insert(Object[] circArr, int start, int size,
      Object elem, int pos) {

    // Double the capacity if full
    if (size == circArr.length)
        resize(circArr, start, size, size * 2);

    // Find the number of elements to shift
    int nShifts = size - pos;

    // Since we're shifting elements to the right, we have to start
    // from the right end of the array, and move backwards until we
    // reach the position where the new element is going to be inserted.
    // Start by figuring out the index of the last element or the next
    // available position -- either one is ok.
    int from = (start + size - 1) % circArr.length;
    int to = (from + 1) % circArr.length;
    for (int i = 0; i < nShifts; i++) {
        circArr[to] = circArr[from];

        // move to and from backwards.
        to = from;
        from = from - 1;
        if (from == -1)
            from = circArr.length;
    }

    // Now there is a hole at the given offset, so find the index of
    // the hole in the underlying array, and put the new element in it.
    int index = (start + pos) % circArr.length;
    circArr[index] = elem;

    // Return the index of the new element
    return index;
}
```

Back to Table of contents

Removing an element in a circular array

Now's let's try the opposite — **remove** an element at a given position in a circular array. The problem is exactly the opposite of insertion — we have to **plug the hole** left by removed element by shifting elements to the **left** by one position.

Let's see the removal in action using the following example — we remove the element `15` from a circular array shown at the top, which results in the circular array shown at the bottom. Since we're given the element to remove, we first need to find the **offset** relative to the **front** element, and then the underlying index. In the example below, we can iterate through the circular array starting from **start**, and see that the element `15` is at an offset `4` from the start. We

show two different ways of plugging the "hole" left by the removed element.

```
Remove element 15 in position 4, or at index (5 + 4) % 8 = 1.

       0    1    2    3    4     5    6   7
     ---------------------------------------   capacity = 8
     |13 | 15 | -4 | 17  |    | 5   | 7 | 9 |   size     = 7
     ---------------------------------------
                               ^-- start
            ^--- this is position 4 relative to front

And the resulting circular array after removal is show below.

       0    1    2    3    4     5    6   7
     ---------------------------------------   capacity = 8
     | 13| -4 | 17 |     |    | 5   | 7 | 9 |   size     = 6
     ---------------------------------------
                               ^-- start

The following is also correct - note the difference in which
elements we're shifting to "plug the hole". And note "start"
is now changed.

       0    1    2    3    4     5    6   7
     ---------------------------------------   capacity = 8
     | 9 | 13 | -4 | 17  |    |     | 5 | 7 |   size     = 6
     ---------------------------------------
                                   ^-- start
```

Just like in the case of insertion, we need to shift; just that, in the case of removal, we shift in the opposite direction to plug a hole. And to shift elements, we need to know the following:

1. the **index** where to start shifting, and
2. the **number of elements** to shift.

As we noted earlier, in the case of a circular array, we are often given the **offset** from the **front** element (which is exactly what the index is for a linear array!), so we have to first compute the index where to start the shift. Once we know that, the process is exactly the same! The **number of elements** to shift is given by `size - offset - 1` (note the `-1` at the end — it's different than the case of insertion).

We can do it one of two ways:

1. Shift all subsequent elements to the **left** by one position, which is the way we remove an element from a linear array (so we call it the *usual* way). The code to do that is shown below. This is the first way of doing it in the illustration above.

```
/**
 * Removes an element at the given position in a circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @param elem    the new element to insert
 * @param pos     the offset of the new element relative to start
 * @return the object that was removed
 */
public static Object remove(Object[] circArr, int start, int size,
      Object elem, int pos) {

    // Save a reference to the element that is to be removed, so find
    // its index first.
    int index = (start + pos) % circArr.length;
    Object removed = circArr[index];
```

```
    // Find the number of elements to shift
    int nShifts = size - pos - 1;

    // Since we're shifting elements to the left, we have to start
    // from the left end of the array where the element to be removed
    // is currently at, and move forwards until we reach the last
    // element. Start by figuring out the index of the element to be
    // removed.
    int to = index;
    int from = (to + 1) % circArr.length;
    for (int i = 0; i < nShifts; i++) {
        circArr[to] = circArr[from];

        // advance to and from forwards.
        to = from;
        from = (from + 1) % circArr.length;
    }

    // The last slot is now unused, so help GC by null'ing it.
    circArr[from] = null;

    // Return the element removed.
    return removed;
}
```

2. Shift the elements from the **front** to the given element **right** by one position, and then advance **front**. This is only possible for a circular array, since we can treat any index as the starting position. This is the second way of doing it in the illustration above.

```
/**
 * Removes an element at the given position in a circular array.
 * @param circArr the circular array
 * @param start   the index of the first element in the circular array
 * @param size    the number of elements (must be ≤ circArray.length)
 * @param elem    the new element to insert
 * @param pos     the offset of the new element relative to start
 * @return the object that was removed
 */
public static Object remove(Object[] circArr, int start, int size,
      Object elem, int pos) {

    // Save a reference to the element that is to be removed, so find
    // its index first.
    int index = (start + pos) % circArr.length;
    Object removed = circArr[index];

    // We'll shift each element from start to this index one
    // position to the right.

    // Find the number of elements to shift. Note that it's not
    // the same number as in the first method.
    int nShifts = size - pos;

    // Since we're shifting elements to the right, we have to start
    // from the right end of the array where the element to be removed
    // is currently at, and move backwards until we reach the front
    // element. Start by figuring out the index of the element to be
    // removed.
    int to = index;
    int from = to - 1;
    if (from == -1)
        from = circArray.length - 1;
    for (int i = 0; i < nShifts; i++) {
        circArr[to] = circArr[from];
```

```
            // move to and from backwards.
            to = from;
            from = from - 1;
            if (from == -1)
                from = circArray.length - 1;
        }

        // The first slot is now unused, so help GC by null'ing it.
        circArr[start] = null;

        // And advance start to point to the shifted front element.
        start = (start + 1) % circArray.length;

        // Return the element removed.
        return removed;
    }
```

Back to Table of contents

Questions to ponder

Some questions for you to ponder:

- How do you shift he elements in a cicular array? For example, how would you shift the first j elements in a circular array one place to the right? Same, but to the left? See how it's done when inserting a new element in a circular array.
- Why do we care about circular arrays? Think of one application where it was nice to have (and why).

Back to Table of contents