AN INTRODUCTION TO CIRCULAR ARRAYS

By: Chris Wickens

Chris Wickens
chriswickens@gmail.com

Contents

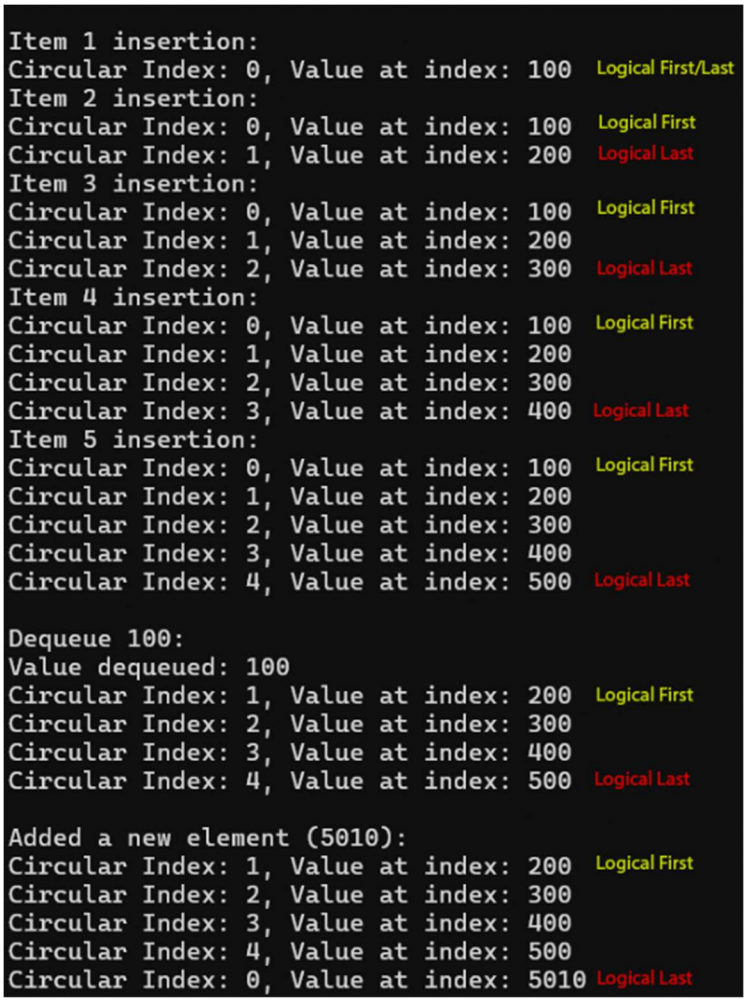
What is a Circular Array?

Typically, a linear array has its data stored in contiguous memory blocks (one after the other) and can only be accessed by indexes or by traversing the array from the beginning to the end, or the end to the beginning to find a specific value stored in the array. However, with a circular array, storing data is done the same way in terms of memory, except accessing data in the array can occur from any index of the array making that point the “logical first“ when that index is being accessed. This means that the “logical last“ would be the index prior to the one accessed first, since this data structure is circular. When adding or removing items from a circular array, it is treated like a Queue in terms of “First In, First Out“ principles, based on which index is being used as the “logical first“ when accessing the values in the array. In contrast, a linear array can have values added or removed from any point so long as the index of the value is known.

The image below (Fig. 1), shows how an empty circular array would use index 0 as its “logical first” and as information is added, the “logical last” will be shifted lower and lower in the array. After dequeuing an item from the full array, index 1 now becomes the “logical first” and the “logical last” remains unchanged until another item is added. When the new item is added, it shows that index 0 is now the “logical last” showing the wraparound that happens in the array as it grows and shrinks.

Figure 1

Console output from a circular array.



Note. From Chris Wickens, 2024.

Advantages and Disadvantages of Circular Arrays

"The main advantage of a circular buffer is that it allows for efficient use of memory by using a constant-size buffer as if it links end-to-end" (DrMax, 2024, p. 5).

A circular array can make better use of memory, as it uses a constant length (size), however this also comes with the issue of older information potentially being overwritten when new data is inserted (if the circular array is coded this way). It is also possible that a circular array could be implemented by "allocating a large amount of memory that we may not fully utilize" (DrMax, 2024, p. 6) which may result in portions of the circular array not being used at all which could be considered a waste of resources. The implementation of a linear array is typically straightforward once the concept is understood, whereas with a circular array, the implementation is more complex and requires expanded knowledge of how the structure works and of what information needs to be tracked to properly make use of the circular nature of the data structure.

Real World Example

Circular arrays, in terms of audio processing, are much more efficient than a standard linear array. For example, when I am recording audio on my DAW (Digital Audio Workstation), I know from my own experience that it uses some implementation of a circular array when recording. The software likely uses a circular array because it will constantly record audio when a track is armed for recording and that track is actively being used for recording. This means that it is constantly taking in new data, processing that data by passing it to the computer to become an audio file (adding the oldest information to the audio track), and then overwriting the oldest audio with the newest audio and repeating constantly. This would be almost impossible to implement with a standard linear array since it does not have the ability to wraparound on itself, which would lead to stuttering and/or latency issues.

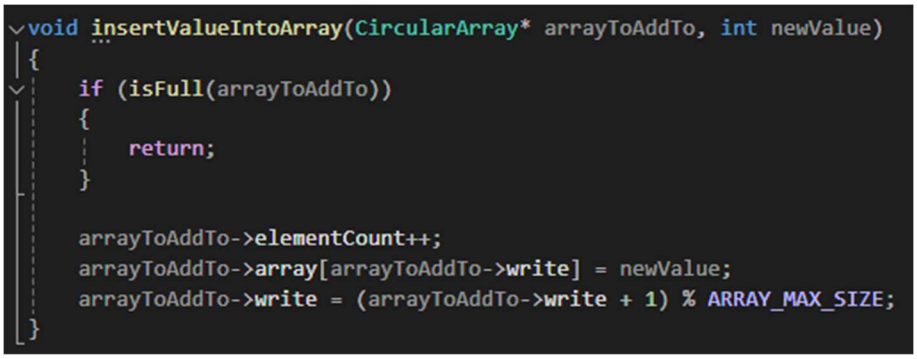
Implementation Explained

Insertion

When inserting new data, the array must be checked to see if it is full. If the circular array is full, the programmer will need to choose whether new data will simply overwrite the oldest data inside of the data structure or whether it will disregard the new data completely. I have implemented my code to simply discard data when an insertion is attempted on a full circular array.

Figure 2

Code for inserting a value into a circular array, not overwriting values.



```
void insertValueIntoArray(CircularArray* arrayToAddTo, int newValue)
{
    if (isFull(arrayToAddTo))
    {
        return;
    }

    arrayToAddTo->elementCount++;
    arrayToAddTo->array[arrayToAddTo->write] = newValue;
    arrayToAddTo->write = (arrayToAddTo->write + 1) % ARRAY_MAX_SIZE;
}
```

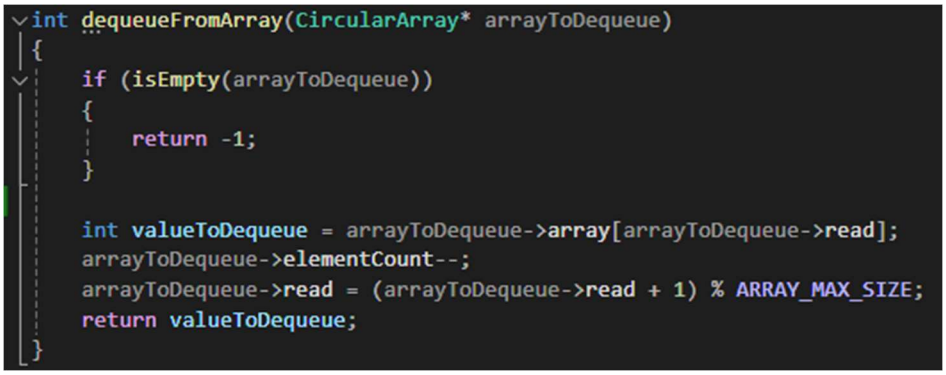
Note. From Chris Wickens, 2024.

Removal

When removing an element from a circular array, much like a circular linked list, the array must be checked to see if the array is empty. If the array is empty, an error would need to be returned to reflect the fact that the array is, in fact, empty. If the array is not empty, the value in the “logical first” of the array should be “removed” by simply updating the “logical first” index to point to the index that was originally before the now removed “logical first”.

Figure 3

Code for removing an element from a circular array.



```
int dequeueFromArray(CircularArray* arrayToDequeue)
{
    if (isEmpty(arrayToDequeue))
    {
        return -1;
    }

    int valueToDequeue = arrayToDequeue->array[arrayToDequeue->read];
    arrayToDequeue->elementCount--;
    arrayToDequeue->read = (arrayToDequeue->read + 1) % ARRAY_MAX_SIZE;
    return valueToDequeue;
}
```

Note. From Chris Wickens, 2024.

Accessing

Accessing information directly in a circular array is no different than a linear array, one simply provides the index to look at, and that is all that is needed. Due to the nature of a circular array being a wraparound style of data structure, referencing an index would be nearly useless since tracking the “logical first/last” is what gives the data structure its circular nature, and one cannot discern exactly what data will be stored at the index being accesses.

Figure 4

Code for accessing an element directly in a circular array.

```
printf("Access item at index 4: %d\n", theArray->array[4]);
```

Note. From Chris Wickens, 2024.

Searching

Searching a circular array works much like searching a linear array. The only difference between the two is that typically, with a linear array, one iterates over the array from the beginning to the end (or vice versa) to ensure that all values are checked and compared against the value being searched for. The two outcomes are either finding the value or reaching the end of the array. With a circular array, the search should begin from the “logical first” index, to properly iterate over the data structure the way it is intended to be used. It is possible to search through the entire circular array for the value and return the value once it is found or to return an error if the search function ended up back at the “logical first” after not having found the value being searched for. It would only make sense to start searching from the “logical first”, as one would not be using it as intended by randomly picking an index to start from. In the example below, a simple counter is used to determine when the entire circular array has been searched.

Figure 5

Code for searching for an element in a circular array.

```
void searchCircularArray(CircularArray* arrayToSearch, int valueToFind)
{
    if (isEmpty(arrayToSearch))
    {
        return;
    }

    int currentIndex = arrayToSearch->read;
    int count = 0;
    while (count < arrayToSearch->elementCount)
    {
        if (arrayToSearch->array[currentIndex] == valueToFind)
        {
            printf("Found item at: %d\nValue being searched for: %d\nValue in array: %d", currentIndex, valueToFind, arrayToSearch->array[currentIndex]);
            return;
        }

        count++;
        currentIndex = (currentIndex + 1) % ARRAY_MAX_SIZE;
    }

    printf("Value not found!\n");
}
```

Note. From Chris Wickens, 2024.

Complexity Analysis

For this section, I referred to (GeeksforGeeks.com, 2023) for linear array complexity information, as well as the slides provided in the Conestoga Data Structures course material. I referenced (DrMax, 2024) for information regarding circular array time complexity as well.

Linear Array (Removing items from the beginning)

When simply removing items from the array without shifting data around and the location is already known, it will have a time complexity of O(1), since it is possible to simply remove/null the element by using the 0 index. If the intent is to shift data when removing an element, it will have a time complexity of O(N) since the data will need to be shifted after removal, which changes the time complexity to be dependent on the number of items in the array.

Circular Array (Removing Items from the beginning)

Since the location of the “logical first” is tracked within a circular array, and the circular array acts like a queue, the time complexity for removing an item would be O(1). Since the element will be dequeued based on the “logical first”, the “logical first” will be updated to reflect the “removal” of the element. “Removal” is used loosely here because in my example the data is not removed. The “removed” data will simply be overwritten when new data is added to the circular array, as the previous “logical first” will become the “logical last” when inserting a new element.

Linear Array (Removing items from the end)

If the index for removing the last element of the array is obtained by simply subtracting 1 from the arrays size, the time complexity would be O(1) without shifting data.

Circular array (Removing items from the end)

As with most operations performed on a circular array, tracking of the “logical first/last” without needing to shift the data results in a time complexity of O(1). The “logical last” index location is simply updated to reflect this change.

Linear Array (Inserting at the beginning)

Much like removing an item, if the index is known and it is not necessary to shift the data, the time complexity is O(1). If the intent is to insert data and shift the existing data, the time complexity will be O(N) as it will be dependent on the number of items in the array which need to be moved to complete the insertion.

Circular Array (Inserting at the beginning)

One benefit of a circular array and of the way that it functions is that the insertion location is already known and does not require shifting the data around. The new value can be inserted at the “logical first” location without needing to shift the data and the time complexity will simply be O(1). This operation would be contrary to the idea of a queue using FIFO which a circular array is based on.

Linear Array (Inserting at the end)

If data is being directly inserted into the last index of the array, this would result in a time complexity of O(1), assuming not data needs to be shifted when completing this operation. If data was to be shifted, the operation would have a time complexity of O(N).

Circular Array (Inserting at the end)

Since the “logical first/last” are tracked, this results in a time complexity of O(1) and a simple reference to the “logical last” value to be used as the insertion point in the circular array.

Linear Array (Searching)

Finding a value in a linear array requires the value being searched for to be potentially compared against every value inside of the array. With luck, the first index will hold the value and the time complexity would be O(1). However, assuming that it is not the first index, the complexity becomes O(N), since the entire array will potentially need to be checked.

Circular Array (Searching)

Finding data follows the same principle as a linear array, regardless of the “logical first/last”. The value being searched for could be at the “logical first”, resulting in O(1). If the value is not in the “logical first”, the entire array will potentially need to be traversed to find that specific value, resulting in a complexity of O(N).

Case Study

As previously discussed and based on my experience using recording software, a circular array is likely used as the buffer since it supports wrapping around itself to overwrite older data and using FIFO to get the recorded audio onto the hard drive of the computer used as the workstation host. The wraparound nature of the circular array makes it the most efficient way to complete the complex operation of having an audio interface take in audio and convert that audio into a digital signal that a computer can store. This process is something that can be seen happening in near real-time: as audio is being recorded, the play head is constantly moving forward in the track (from left to right), and the actual wave forms can be seen being generated in near real-time as each sample is processed. In the simplest terms, the sample RATE is how often the computer is taking a “sample” of the incoming audio from the interface.

“Sample rates are usually measured per second, using kilohertz (kHz) or cycles per second. CDs are usually recorded at 44.1kHz - which means that every second, 44,100 samples were taken.” (Adobe, 2024). The sample rates on most interfaces range from 44.1kHz up to 192kHz, meaning that it could potentially take 44,100 samples, all the way up to 192,000 samples *per second*. This setting is extremely important when it comes to audio production, as it will dictate the quality of the audio being captured.

Imagine trying to capture that much information per second by using a linear array. The computer would need to process the information in the array and then clear out the array after it becomes full instead of the way a circular array functions, which

means it constantly wraps back around on itself without removing the audio that has been transferred to the computer until that portion of the circular array is overwritten. This would take precious clock cycles in a linear array, in a setting that requires very strict timing to ensure that the performing artist hears their audio in near real-time as they are recording. I say *near* real-time, as there is always latency when using digital recording interfaces, although it is nearly imperceptible if one is using a good interface with a powerful computer.

Conclusion

The use of circular arrays instead of linear arrays happens in many different applications. I could discuss the application of circular arrays in audio production *ad nauseum*, as it is used for nearly every aspect of processing and playback of audio in that environment. These types of circular arrays are used without most people even knowing it: “all of your keystrokes are held in a circular buffer” (Oswold, 2019, p. 4) and the buffering of audio/video streams from services like Spotify and YouTube, for example. These data structures are used in everyday tasks without anyone being the wiser because they are the most efficient way of processing streaming data.

As the internet becomes more advanced and the amount of data that can be transferred becomes ever larger, people will want that data to be presented as quickly as possible, which means that circular arrays will likely be the most efficient way to accomplish this given the current computer architecture that is used in consumer products.

None can say what the future holds for technology nor how these data structures will be used to implement new and exciting technologies requiring large quantities of data to be streamed, even as it relates to audio production with the increased throughput of newer USB/Thunderbolt standards. This could mean that there may be a new data structure concept that could be even more efficient – someone just needs to design it first (not me).

References

Adobe. (2024). *Sample rates and audio sampling: A guide for beginners | adobe*. Retrieved from Adobe.com: https://www.adobe.com/uk/creativecloud/video/discover/audio-sampling.html

DrMax. (2024, March 18). *Circular Buffer*. Retrieved from Baeldung: https://www.baeldung.com/cs/circular-buffer

GeeksforGeeks.com. (2023, November 03). *Time and Space complexity of 1-D and 2-D Array Operations*. Retrieved from Geeks for Geeks: https://www.geeksforgeeks.org/time-and-space-complexity-of-1-d-and-2-d-array-operations/

Oswold, C. (2019, May 7). *Data Structures: Your Quick Intro to Circular Buffers*. Retrieved from Medium: https://betterprogramming.pub/now-buffering-7a7d384faab5