Queues using Array

Muhammad Shahwaiz Abbasi

VT 2026

Introduction

This report presents the assignment to achieve a grade **D** in the Data Structures and Algorithms (ID1021) course.

We will build a basic queue using an array. We will add and remove elements from the queue. Adding is called *enqueue* and removing is called *dequeue*. We will also come up with solutions for problems like array being full and empty space in an array.

The objective of this assignment is to understand array based queues, what they are, how they are made, how they work, what problems can occur and how to solve those problems.

What problems can occur?

The following problems can occur during queue operations:

What happens if the array gets full?
A solution is to `resize` the array by calling a function called increase size, which doubles the size of the array to accommodate more elements.

What happens to empty spaces after dequeueing?
Those empty spaces are reused via wrap around method.

What is a wrap around?

A `wrap around` is when you use the empty spaces at the beginning of an array after elements have been removed from the front. Imagine you have a queue with an array of size 4. You fill the array by enqueueing 4 times. Then you dequeue twice, removing the first two elements. Now you have two empty spaces left at the start but you last index is at the end of the

queue. In this situation you normally resize the array to add more elements but this is inefficient because the empty spaces at the front are unused. Here `wrap around` comes and save the day. It allows the `last` to loop back to the start of the array and let the empty spaces be used again. However, a new problem occurs. **What if the `last` becomes equal to the `first` index? Does that mean the queue is empty?** An empty queue is indicated when `First == last`. To avoid this problem we then compare next position after `last` equals `first`, which tell us that the queue is full and then we can resize the array.

The following is the blueprint for Array based Queue:

```
typedef struct queue {
    int *array;
    int front;
    int last;
    int size;
} queue;
```

CODE 0

`CODE 0` contains the structure for the queue. It has an array where the elements are stored along with front and last indicators and At last, it holds the size of that array.

Creating the Queue

```
queue *create_queue() {
    queue *q = (queue*)malloc(sizeof(queue));
    q->array = (int*)malloc(SIZE * sizeof(int));
    q->front = 0;
    q->last = 0;
    q->size = SIZE;
    return q;
}
```

CODE 1

Above is the function to create a queue. It starts by allocating memory for the queue structure and then for the array. Then it sets the front and last pointers to 0 and sets the size to the given size.

Resizing Array

```
void increase_size(queue *q) {
    int size = q->size * 2;
        int *copy=(int*)malloc( size * sizeof(int));
```

```
        for (int i = 0; i < q->size; i++) {
            copy[i] = q->array[i];
        }
        free(q->array);
        q->array = copy;
        q->size = size;
}
```

CODE 2

CODE 2 is a function to resize the array when it is full. It first doubles the size, allocates memory for a copy of the array and copies the elements into it. Then it empties the old array. After that it points the original array to the new copy. Thenit updates the new size.

Enqueue

```
void enque(queue* q, int v) {
    if ((q->last + 1) % q->size == q->front) {
        increase_size(q);
    }
    q->array[q->last] = v;
    q->last = (q->last + 1) % q->size;
}
```

CODE 3

Code 3 is the enqueue operation, where elements are added to the back of the queue. It begins by checking if the queue is full. If true, it increases the size of the array using CODE 2. Then it adds the given value at the last position with the help of the index `last`. After that `last` is move forward by 1 step. When `last` reaches the end it wraps back to the start to use the empty spaces.

Dequeue

```
int dequeue(queue *q) {
    if (empty(q)) {
        printf("Queue is empty\n");
        return -1;
    }
    int val = q->array[q->front];
    q->front = (q->front + 1) % q->size;
    return val;
}
```

CODE 4

`Code 4` is the dequeue operation, where elements are removed from the front of the queue. It begins by checking if the queue is empty. If true, it prints out a message saying *Queue is empty*. Then it gets the value at the Front and removes it. After that `first` is move forward by 1 step. When `first` reaches the end it wraps back to the start to use the empty spaces that were filled afterwards.

Analysis of the Time Performance of Array Based Queues

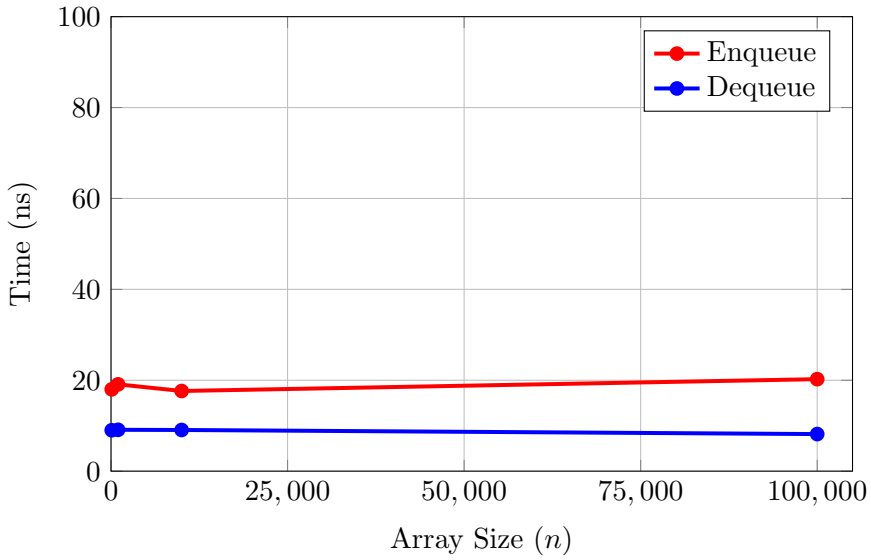

Figure 1: Time Performance of Enqueue and Dequeue Operations on Array Based Queues

`Figure 1` clearly shows that the time complexity of both enqueue and dequeue operations is $\mathcal{O}(1)$. But worst case occurs when the resize function gets triggered which changes the complexity to $\mathcal{O}(n)$ because all n elements are copied to the new copy array. This is shown in `Figure 2` on the next page.

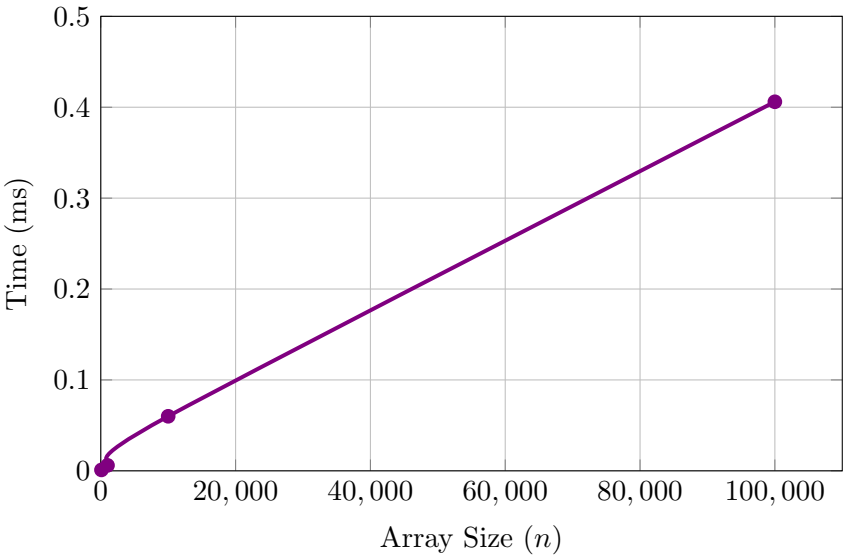

Figure 2: Worst Case Scenario for the Enqueue Operation