January 18, 2019 at 11:18

1. Queue implementation using a Circular Array. This is an implementation of the Pascal pseudocode provided in pages 74-75 of the book “Data Structures and Algorithms” by Aho et al. (This is my first real `CWEB` program.)

A Queue is a list datastructure where items are inserted at one end (*enqueue*), and taken out from the other end (*dequeue*). It can also be called a FIFO i.e. “first-in first-out” list.

In a linked-list implementation of a queue, we can dequeue by removing the first element of the list. However we must traverse to the end of the list to enqueue. The linked list implementation also requires piece-wise allocation and de–allocation of the items in the list.

If we have a reasonable idea of the maximum possible size of the queue before–hand, then we can improve on this by using an array of fixed length and using it in a circular fashion.

To make this possible, we need a key addition to the abstraction of an array. When we increment the index of the array beyond it’s length, it should reset to zero, effectively making it a never ending traversal.

Once we have this abstraction, we can go on and implement the queue procedures using this abstraction of a circular array.

Finally, we will write some test code which can be built and run with a flag. If the flag for test code is not set, one can build this as library instead. The flag is called `TEST_CODE`

2. Here’s how this program is structured. The program has four sections viz. headers, circular array, queue implementation, and test program (optional).

```
⟨ Header files to include 3 ⟩
⟨ Circular Array 4 ⟩
⟨ Queue implementation 5 ⟩
⟨ Test program 13 ⟩
```

3. Let’s get the headers out of the way first. We include `stdio.h` if we want to run the test. `stdlib.h` is always included, as we use malloc to create the queue when defining *make_queue*.

```
⟨ Header files to include 3 ⟩ ≡
#ifdef TEST_CODE
#include <stdio.h>
#endif
#include <stdlib.h>
#include <stdbool.h>
```

This code is used in section 2.

4. Circular Array: So let’s start with this one function abstraction of a circular array. The idea is to use this function to go to the next item in an array when traversing (instead of just incrementing the index).

If the index i is a multiple of $max\_length - 1$, the last index of the array, then we start from 0.

```
⟨ Circular Array 4 ⟩ ≡
  long next(long i, long max_length)
  {
    return ((i + 1) % (max_length − 1));
  }
```

This code is used in section 2.

5. Queue implementation: We define the queue datastructure, and procedures to create, and destroy queues and also functions to access the queue.

```
⟨ Queue implementation 5 ⟩ ≡
  ⟨ Errors 6 ⟩
  ⟨ Queue Datastructure 7 ⟩
  ⟨ Make queue 8 ⟩
  ⟨ Free queue 9 ⟩
  ⟨ Enqueue 10 ⟩
  ⟨ Empty 11 ⟩
  ⟨ Dequeue 12 ⟩
```

This code is used in section 2.

6. Errors: All errors are returned as numeric error codes which are defined in an enum

```
⟨ Errors 6 ⟩ ≡
  typedef enum {
    SUCCESS = 0, E_ALLOC_FAILED = −1, E_QUEUE_FULL = −2
  } q_err_t;
```

This code is used in section 5.

7. Queue Data Structure: Before we define the queue functions, we will first define a queue datastructure based on an array.

```
⟨ Queue Datastructure 7 ⟩ ≡
  typedef struct {
    void **array;
    long max_length;
    long front, rear;
  } queue_t;
```

This code is used in section 5.

8. Queue creation: This is a standard pattern in C to create new structs, where the return value is an argument. The return value of the function is just an error code, but if the creation is successful then the value of the *q* pointer is set to newly created but empty queue. This allows us the possibility of returning multiple error codes for failures.

The *make_queue* function one argument *max_length* for the maximum possible length of the queue.

To distinguish between a full and an empty queue, we must consider the queue full when the *next* of *next* of *rear* is *front*. If we ever got to the point of the *next*(*rear*) being equal to front while filling the queue, then we would not be able to distinguish this from an empty queue.

⟨ Make queue 8 ⟩ ≡

```
q_err_t make_queue(queue_t **q, long max_length)
{
  (*q) = (queue_t *) calloc(1, sizeof(queue_t));
  if (¬(*q)) {
    return E_ALLOC_FAILED;
  }
  (*q)→array = (void **) calloc(max_length, sizeof(void *));
  if (¬(*q)→array) {
    return E_ALLOC_FAILED;
  }
  (*q)→max_length = max_length;
  (*q)→front = 0;
  (*q)→rear = max_length − 2;
  return SUCCESS;
}
```

This code is used in section 5.

9. Queue destroy: To destroy the queue, we first free the array, and then free the queue struct itself.

⟨ Free queue 9 ⟩ ≡

```
void free_queue(queue_t *q)
{
  if (q) {
    free(q→array);
    free(q);
  }
}
```

This code is used in section 5.

10. Enqueue: If the *next* of *next* index of *rear* is *front* then the queue is full. And we return an error. If the queue is not full, we add the item to the queue, and we change the value of *rear* to *next*(*rear*).

```
⟨ Enqueue 10 ⟩ ≡
  q_err_t enqueue(queue_t *q, void *item)
  {
    if (next(next(q→rear, q→max_length), q→max_length) ≡ q→front) {
      return E_QUEUE_FULL;
    }
    else {
      q→rear = next(q→rear, q→max_length);
      q→array[q→rear] = item;
      return SUCCESS;
    }
  }
```

This code is used in section 5.

11. Empty queue: The queue is empty if the *next* of *rear* is *front*.

```
⟨ Empty 11 ⟩ ≡
  bool empty(queue_t *q)
  {
    if (next(q→rear, q→max_length) ≡ q→front) {
      return true;
    }
    return false;
  }
```

This code is used in section 5.

12. Dequeue: If the queue is empty then we can't remove an element from it, so we return an error. Otherwise we return the first element and change *front* to *next*(*front*).

```
⟨ Dequeue 12 ⟩ ≡
  void *dequeue(queue_t *q)
  {
    void *ret = Λ;
    if (¬empty(q)) {
      ret = q→array[q→front];
      q→front = next(q→front, q→max_length);
    }
    return ret;
  }
```

This code is used in section 5.

13. Test program: A program to create a queue, add a few numbers and read them back, then free the queue.

```
⟨ Test program 13 ⟩ ≡
#ifdef TEST_CODE
  int main(int argc, char **argv)
  {
    q_err_t ret = 0;
    queue_t *q = Λ;
    if (make_queue(&q, 10) ≡ 0) {
      for (int i = 0; i < 9; i++) {
        int *x = (int *) calloc(1, sizeof(int));
        if (x ≠ Λ) {
          *x = i * 200;
          int ret = enqueue(q, x);
          if (ret ≡ SUCCESS) {
            printf("Enqueue'd␣%d\n", *x);
          }
          else {
            printf("Queue␣is␣full␣now.\n");
          }
        }
      }
      for (int i = 0; i < 9; i++) {
        int *val;
        if ((val = (int *) dequeue(q)) ≠ Λ) {
          printf("Dequeue'd␣%d.\n", *val);
        }
        else {
          printf("Queue␣is␣empty␣now.\n");
        }
        free(val);
      }
    }
    free_queue(q);
    exit(0);
  }
#endif
```

This code is used in section 2.

14. Index. Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition.

QUEUE˙CIRCULAR˙ARRAY