FIFO Queues

The logical level

- What is a queue?
  - Regulate processing of tasks to ensure *fair* treatment
    - A waiting line
  - Ordered homogeneous group of elements in which the elements are added at one end (rear) and removed from the other end (front)
  - The only two accessible elements in a queue are the front element (to remove it) and the last/rear element (to add another element to the queue
  - The middle elements of a queue are logically inaccessible
  - The ends of the queue are themselves abstractions, and may or may not correspond to any physical characteristics of the implementation
  - The essential property of a queue is the FIFO access – "First-in-First-out"
  - The items in the queue cannot be manipulated directly; to do so, we should take the element off the queue, change it, and put it back on
- Operations on FIFO queues
  - Queue is a dynamic data structure and changes as the elements are added to or removed from it
  - Queue ADT specification

    Structure. Elements are added to the rear and removed from the front of the queue
    Create queue. Initializes a queue to an empty state

    ```
    queue_type create_queue ( queue_type queue );
    ```

      Function. Initializes queue to an empty state
      Input. None
      Preconditions. None
      Output. A queue
      Postconditions. Queue is empty

    Destroy queue. Remove the queue's existence

      Function. Removes all elements from queue, leaving the queue empty
      Input. A queue
      Preconditions. The queue exists
      Output. Queue
      Postconditions. The queue is empty

    Empty queue? Checks to see if the queue is empty

      Function. Tests whether the queue is empty
      Input. A queue
      Preconditions. Queue has been created
      Output. True or False (1 or 0)
      Postconditions. True if queue is empty, false otherwise

    Full queue? Checks to see if the queue is full

      Function. Tests whether the queue is full
      Input. A queue
      Preconditions. Queue has been created
      Output. True or False (1 or 0)
      Postconditions. True if queue is full, false otherwise

    Enqueue an element

      Function. Adds a new element to the rear of the queue

Input. Queue as well as the new element
Preconditions. Queue has been created and is not full
Output. Queue
Postconditions. The new element is added to the rear of the queue

Dequeue an element

Function. Removes the front element from the queue and returns it as the dequeued element
Input. Queue
Preconditions. Queue has been created and is not empty
Output. Queue, dequeued element
Postconditions. Queue is returned with the front element removed which is returned as the dequeued element

The user level

The implementation level

- Implementation of a queue as a static array
  - All elements of a queue are the same type, allowing the queue to be implemented as an array
  - First element goes in position 0, second element in position 1, and so on
  - We need to keep track of the front and rear of queue
  - Can be accomplished through a structure

```
#define MAX_QUEUE 100

typedef struct
{
    int items[MAX_QUEUE];
    int front, rear;
} queue_type;

queue_type q;
q.front = q.rear = MAX_QUEUE - 1;
```

  - `q.front` and `q.rear` are initialized to the last index of the array, rather than to -1 or 0, because the last element of the array immediately precedes the first one in this implementation
  - Since `q.rear` equals `q.front`, the queue is initially empty

- Queue operations with the array implementation
  - Creating the queue

```
queue_type * create_queue ( queue_type * queue )
{
    /* Allocate memory and initialize the queue        */

    queue = ( queue_type * ) malloc ( sizeof ( queue_type ) );
    queue -> front = queue -> rear = MAX_QUEUE - 1;

    return ( queue );
}
```

  - Destroying the queue

```
void destroy_queue ( queue_type * queue )
{
    free ( queue );
}
```

- Function to check for empty queue

  ```
  int empty_queue ( queue_type * queue )
  {
      return ( queue -> front == queue -> rear );
  }
  ```

- Function to check for queue being full
  * Being a dynamic data structure, queue cannot be full
  * We are limited to the maximum size of the array for queue due to our choice of implementation

    ```
    int full_queue ( queue_type * queue )
    {
        return ( ( ( queue -> rear + 1 ) % MAX_QUEUE ) == queue -> front );
    }
    ```

- Inserting an element into the queue (enqueue)

  ```
  queue_type * enqueue ( queue_type * queue, char element )
  {
      queue -> rear = ( ( queue -> rear )++ % MAX_QUEUE );
      queue -> items[queue->rear] = element;

      return ( queue );
  }
  ```

  * We should check for the condition that the queue is not full
  * *Queue overflow* is the condition resulting from trying to enqueue an element into a full queue

  ```
  #define OVERFLOW 1
  queue_type * enqueue ( queue_type * queue, q_el_type element, int * error )
  {
      if ( full_queue ( queue ) )
      {
          *error = OVERFLOW;
          return ( queue );    /* Return queue without modification */
      }

      *error = 0;
      queue -> rear = ( ( queue -> rear )++ % MAX_QUEUE );
      queue -> items[queue->rear] = element;

      return ( queue );
  }
  ```

- Removing an element from the queue (dequeue)

  ```
  #define UNDERFLOW 2

  q_el_type dequeue ( queue_type * queue, int * error )
  {
      if ( empty ( queue ) )
      {
          *error = UNDERFLOW;
          return ( queue );
      }

      *error = 0;
      queue -> front = ( queue -> front )++ % MAX_QUEUE;
  ```

```
    return ( queue -> items[queue->front] );
}
```

- Implementation of a queue as a linked structure
  - True dynamic memory allocation-based algorithm
  - Designing the structure
    * Need to create the space for each element on the fly
    * The space created for the element should also have room to point to another element, which has followed just behind
    * We'll call each element in this queue a *node*
    * Each node contains an information field and a field for connection with the rest of the queue
    * The entire node can be declared as follows

```
typedef struct queue_node_type
{
    queue_element_type     element;  /* The information field                */
    struct queue_node_type *next;    /* Pointer to the next element          */
};

typedef struct
{
    struct queue_node_type *front, *rear;   /* Front and rear of queue */
} queue_type;
```

  - Creating the queue

```
queue_type * create_queue ( queue_type * queue )
{
    queue = ( queue_type * ) malloc ( sizeof ( queue_type ) );
    queue -> front = queue -> rear = NULL;

    return ( queue );
}
```

  - Function to check for empty queue

```
int empty_queue ( queue_type * queue )
{
    return ( ! queue -> front );       /* Returns true if queue is empty */
}
```

  - Inserting an element into the queue

```
queue_type * enqueue ( queue_type * queue, queue_element_type element )
{
    queue_node_type * node;        /* A node in the queue                   */

    node = ( queue_node_type * ) malloc ( sizeof ( queue_node_type ) );

    node -> element = element;
    node -> next = NULL;

    if ( queue -> rear )           /* is not NULL */
        ( queue -> rear ) -> next = node;
    else
        queue -> front = node;
```

```
    queue -> rear = node;

    return ( queue );
}
```

– Removing an element from the queue (dequeue)

```
#define UNDERFLOW 2
queue_element_type dequeue ( queue_type * queue, int * error )
{
    queue_element_type element;
    queue_node_type    * tmp;

    if ( empty_queue ( queue ) )
    {
        *error = UNDERFLOW;
        return ( );    /* Return nothing */
    }

    /* Save the element and the pointer to the front of the queue       */

    element = ( queue -> front ) -> element;
    tmp = queue -> front;

    /* Set queue front to point to the next element in the queue        */

    queue -> front = ( queue -> front ) -> next;

    if ( queue -> front == NULL )
        queue -> rear = NULL;

    /* Free the space occupied by the top node of the queue            */

    free ( tmp );

    return ( element );
}
```

– Destroying the queue

```
void destroy_queue ( queue_type * queue )
{
    int error;
    queue_element_type element;

    while ( ! empty_queue ( queue ) )
        element = dequeue ( queue, error );
}
```