Queues

Step 1

The next **Abstract Data Type** we will discuss is the **Queue**. If you've ever gone grocery shopping or waited in any type of a line (which is actually called a "queue" colloquially in British English), you've experienced a **Queue**. Just like a grocery store line, we add elements to the back of the **Queue** and we remove elements from the front of the **Queue**. In other words, the *first* element to come *out* of the **Queue** is the *first* element that went *into* the Queue. Because of this, the **Queue** is considered a **"First In, First Out" (FIFO)** data type.

Formally, a **Queue** is defined by the following functions:

- **`enqueue(element)`:** Add `element` to the back of the Queue
- **`peek()`:** Look at the element at the front of the Queue
- **`dequeue()`:** Remove the element at the front of the Queue

STOP and Think: Do these functions remind us of an **Abstract Data Type** we've learned about?

Step 2

As you should have hopefully inferred, we can use a **Deque** to implement a **Queue**: if we implement a **Queue** using a **Deque** as our backing structure (where the **Deque** would have its own backing structure of either a **Doubly-Linked List** or a **Circular Array**, because as you should recall, a **Deque** is also an **ADT**), we can simply reuse the functions of a **Deque** to implement our **Queue**. For example, say we had the following **Queue** class in C++:

```
class Queue {
    private:
        Deque deque;
    public:
        bool enqueue(Data element);
        Data peek();
        void dequeue();
        int size();
};
```

We could extremely trivially implement the **Queue** functions as follows:

```
bool Queue::enqueue(Data element) {
    return deque.addBack(element);
}
```

```
Data Queue::peek() {
    return deque.peekFront();
}
```

```
void Queue::dequeue() {
    deque.removeFront();
}
```

```
int Queue::size() {
    return deque.size();
}
```

Of course, as we mentioned, the **Deque** itself would have some backing data structure as well, but if we use our **Deque** implementation to back our **Queue**, the **Queue** becomes extremely easy to implement.

Watch Out! Notice that, in our implementation of a **Queue**, the `dequeue()` function has a `void` return type, meaning it *removes* the element on the front of the **Queue**, but it does not *return* its value to us. This is purely an implementation-level detail, and in some languages (e.g. Java), the `dequeue()` function removes *and* returns the top element, but in other languages (e.g. C++), the `dequeue()` function *only removes* the front element without returning it, just like in our implementation.

STOP and Think: In our implementation of a **Queue**, we chose to use the `addBack()`, `peekFront()`, and `removeFront()` functions of the backing **Deque**. Could we have chosen `addFront()`, `peekBack()`, and `removeBack()` instead? Why or why not?

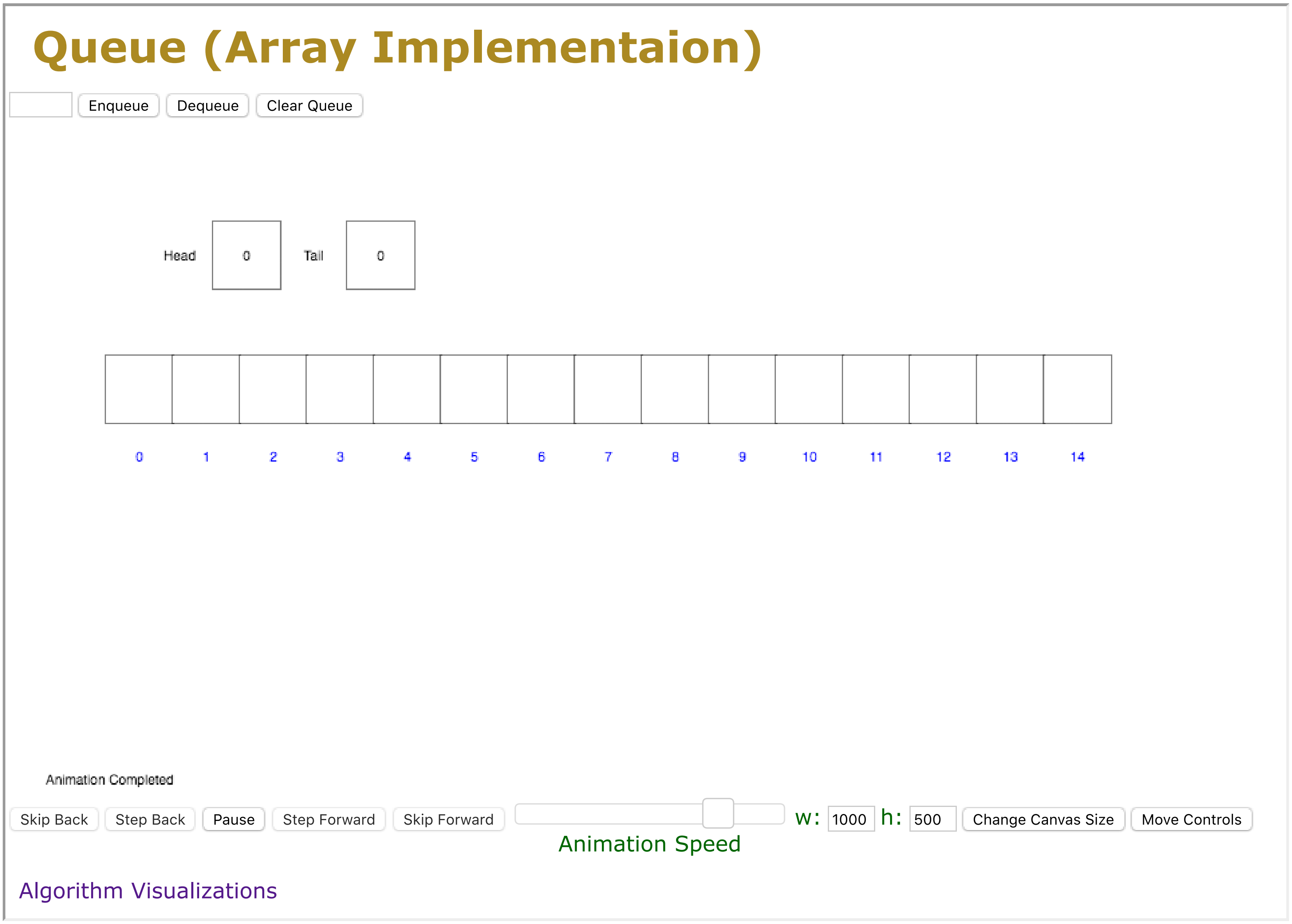
Step 3

Below is an example in which we enqueue and dequeue elements in a **Queue**. Note that we make no assumptions about the implementation specifics of the **Queue** (i.e., we don't use a **Linked List** nor a **Circular Array** to represent the **Queue**) because the **Queue** is an **ADT**.

Initialize queue	**QUEUE: <empty>**
Enqueue **N**	**QUEUE: N**
Enqueue **M**	**QUEUE: N M**
Enqueue **Q**	**QUEUE: N M Q**
Dequeue	**QUEUE: M Q**
Enqueue **D**	**QUEUE: M Q D**
Dequeue	**QUEUE: Q D**
Dequeue	**QUEUE: D**
Dequeue	**QUEUE: <empty>**

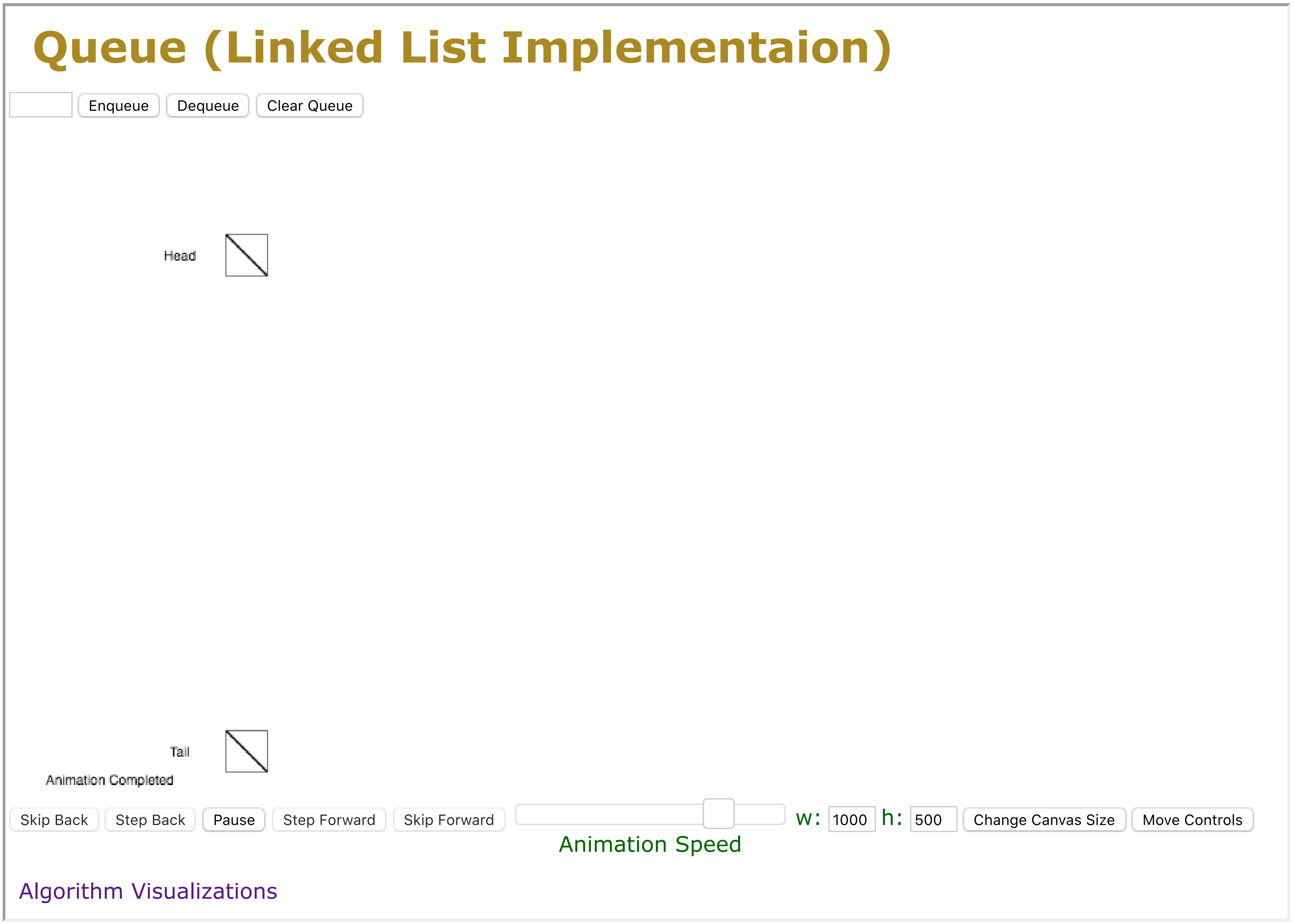
Step 4

Below is a visualization of a **Queue** backed by a **Circular Array**, created by David Galles at the University of San Francisco.



Step 5

Below is a visualization of a **Queue** backed by a **Linked List**, created by David Galles at the University of San Francisco.



Step 6

EXERCISE BREAK: What is the worst-case time complexity of adding an element into a **Queue**, given that we are using a **Deque** as our backing structure in our implementation?

To solve this problem please visit https://stepik.org/lesson/28872/step/6

Step 7

EXERCISE BREAK: What is the worst-case time complexity of adding an element into a **Queue**, given that we are using a **Deque** as our backing structure in our implementation and given that we are using a **Circular Array** as the backing structure of our **Deque**?

To solve this problem please visit https://stepik.org/lesson/28872/step/7

Step 8

EXERCISE BREAK: What is the worst-case time complexity of adding an element into a **Queue**, given that we are using a **Deque** as our backing structure in our implementation and given that we are using a **Doubly-Linked List** as the backing structure of our **Deque**?

To solve this problem please visit https://stepik.org/lesson/28872/step/8

Step 9

Despite its simplicity, the **Queue** is a very powerful ADT because of the numerous applications to which it can be applied, such as:

- Organizing people waiting for their turn (e.g. for a ride, for an event, for a cash register, etc.)
- Keeping track of tasks that need to be completed (in real life, in a computer scheduler, etc.)
- "Graph" exploration via the "BFS" algorithm (which will be covered in the "Graphs" section of this text)

In the next section, we will discuss another simple, yet powerful, ADT that can also be implemented using a **Deque**.