Deques

Step 1

In this day and age, we take web browsing for granted. The plethora of information available at our fingertips is extensive, and with web browsers, we can conveniently explore this information (or more realistically, watch funny cat videos on YouTube or find out which movie character we are via a BuzzFeed survey). As we're browsing, a feature we typically heavily depend on is the browser history, which allows us to scroll back and forward through pages we've recently viewed.

We can think of the browser history as a list of web-pages. As we visit new pages, they get added to the end of the list. If we go back a few pages and then go another route, we remove the pages that were previously at the end of the list. Also, if we browse for too long and this list becomes too large, web-pages from the beginning of the list are removed to save space.

To summarize, we've just listed a set of features: inserting, removing, and looking at elements from the front and back of some sort of list. What we have just described is the first **Abstract Data Type (ADT)** we will explore: the **Double-Ended Queue**, or **Dequeue/Deque** (pronounced "deck").

Step 2

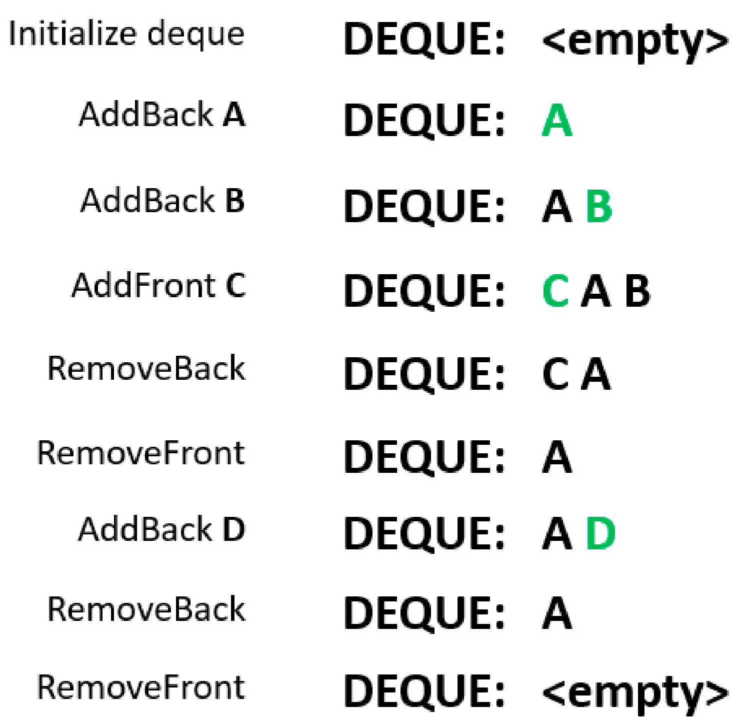
More formally, the **Deque** ADT is defined by the following set of functions:

- **`addFront(element)`:** Add `element` to the front of the Deque
- **`addBack(element)`:** Add `element` to the back of the Deque
- **`peekFront()`:** Look at the element at the front of the Deque
- **`peekBack()`:** Look at the element at the back of the Deque
- **`removeFront()`:** Remove the element at the front of the Deque
- **`removeBack()`:** Remove the element at the back of the Deque

STOP and Think: Do these functions remind us of any data structures we've learned about?

Step 3

Below is an example adding and removing elements in a **Deque**. Note that we make no assumptions about the implementation specifics of the **Deque** because the **Deque** is an **ADT**. In the example, the "front" of the **Deque** is the left side, and the "back" of the **Deque** is the right side.



Initialize deque	DEQUE: <empty>
AddBack **A**	DEQUE: A
AddBack **B**	DEQUE: A B
AddFront **C**	DEQUE: C A B
RemoveBack	DEQUE: C A
RemoveFront	DEQUE: A
AddBack **D**	DEQUE: A D
RemoveBack	DEQUE: A
RemoveFront	DEQUE: <empty>

Step 4

Based on what we have learned so far in this text, there are two approaches we could take that are quite good for implementing a **Deque**: using a **Linked List** or using a **Circular Array**. As you should recall, both data structures have good performance when accessing/modifying elements at their front/back, so how can we make a decision about which of the two we would want to use to implement a **Deque**?

If we were to choose a **Linked List** (a ***Doubly*-Linked List**, specifically), because we would have direct access to both the *head* and *tail* nodes and because inserting/removing elements at the front and back of a **Doubly-Linked List** can be reduced to constant-time pointer rearrangements, we would be guaranteed **O(1)** time for all six of the **Deque** operations described previously, no matter what. However, *if* we were to want access to the middle elements of our **Deque**, a **Doubly-Linked List** would require a **O(*n*)** "find" operation, even if we know exactly which index in our **Deque** we want to access.

If we were to choose a **Circular Array**, because we have direct access to both the *head* and *tail* indices and because the backing array has constant-time access to any element, the "find" and "remove" operations of the **Deque** are guaranteed to have **O(1)** time. However, even though inserting into the **Circular Array** *usually* has O(1) time, recall that, when the backing array is completely full, we need to create a new backing array and copy all of the elements over, which results in a **O(*n*)** worst-case time complexity. Also, *if* we were to want access to the middle elements of our **Deque**, if we knew exactly which index in our **Deque** we wanted to access, the random access property of an array would allow us **O(1)** time access.

Step 5

EXERCISE BREAK: What is the worst-case time complexity of insertion to the front of a **Deque**?

To solve this problem please visit https://stepik.org/lesson/29931/step/5

Step 6

EXERCISE BREAK: If we only care about adding/removing/viewing elements in the front or back of a **Deque** (and not at all in the middle), which of the two implementation approaches we discussed would be the better choice?

To solve this problem please visit https://stepik.org/lesson/29931/step/6

Step 7

In summation, assuming we don't care too much about access to the elements in the middle of the **Deque** (which is typically the case), our best option is to use a **Linked List** to implement our **Deque**. In doing so, we can achieve **O(1)** time for all six of the **Deque** functions we described previously without wasting any extra memory leaving space for future insertions (which we do with an **Array List**).

We already discussed how **Deques** can be used to implement browser history functions of web browsers, but it turns out that **Deques** can also be useful in implementing *other* **Abstract Data Types**. In the next sections, we will discuss two other very useful ADTs that we can use a **Deque** to implement.