Queue Data Structure

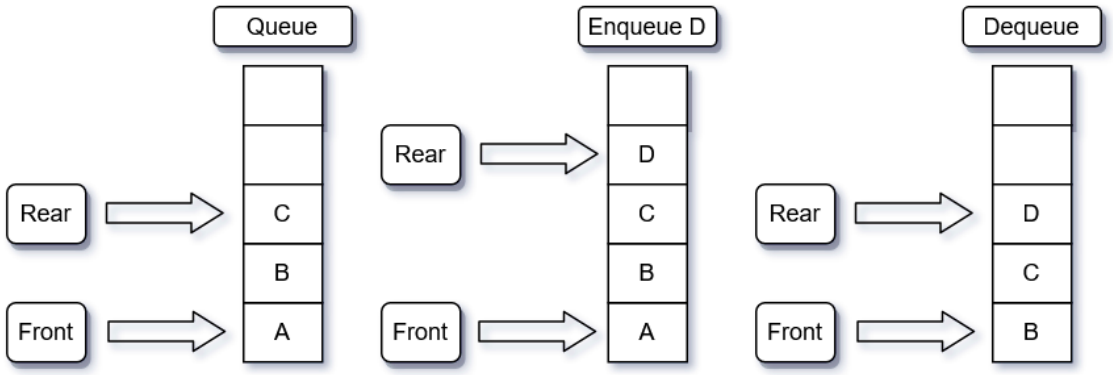
By: João Júlio and Rodrigo Sarroeira

On: 23/july/2020

Queue:

The Queue is a quite common, simple and intuitive data structure. The basic idea is the following: The first element that is inserted, will be the first getting out. This system has a name, **FIFO** (First in First out). This structure can be used when, for example, there are multiple hierarchical tasks to do. If task **A** is more important than task **B**, task **A** should be inserted first, in order to be completed as soon as possible. After task **A** is accomplished task **B** is started. This idea will be explained throughout this essay.

Figure 1 – Queue example (tasks)

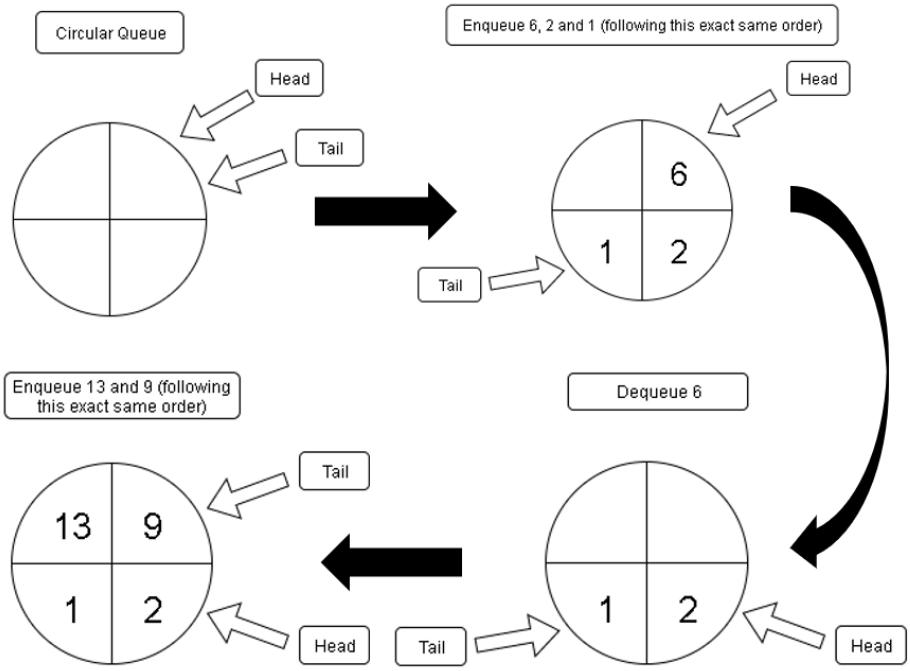


In Figure 1, initially we have 3 tasks in our queue: **A-** First, **B-** Second and **C**-Third. Then using the method enqueue, we add a new task **D** to the rear since there are more tasks waiting in the queue to get completed. After that, we remove task **A** from the queue using the method dequeue, now the task in the front of the queue becomes task **B** since this task was waiting for task **A** to finish, so it can get started and task **D** stays at the rear because no tasks were added to the queue.

Circular Queue:

The Circular Queue data structure is just the Queue data structure represented by an array instead of a list. This means that instead of having unlimited slots in our queue, we have a limited number of slots. So, we use this data structure (**DST**), usually, when we know how many slots we need in our queue.

Figure 2 - How Circular Queue works



To understand the circular queue better, we can see it as a pie chart. Imagine you have a pie with 4 slices (figure 2), in which slice you can put any number you want, and you have a pointer to the first (**called head**) and the last slice (**called tail**) waiting in the queue to get called. With these pointers you can go around the pie/circular queue and use the same operations that we used before in the Queue data structure. The figure 2 shows how the pie representing the circular queue, works and looks like.

Deque:

The **Deque DST is a mix of the Stack and Queue DST** because we can add elements to the rear (like in both **Stack and Queue**) and remove from both the front (**Queue**) and rear (**Stack**) of our Deque. The extra operation, that is used in the Deque and not used in the Stack and Queue, is adding elements to the front. So, with that said, we can now say that the Deque follows both **FIFO** (First in First out) and **LIFO** (First in Last out) structures. We will see that in Figure 3, that is going to show us how this data structure works.

Figure 3- Deque DST

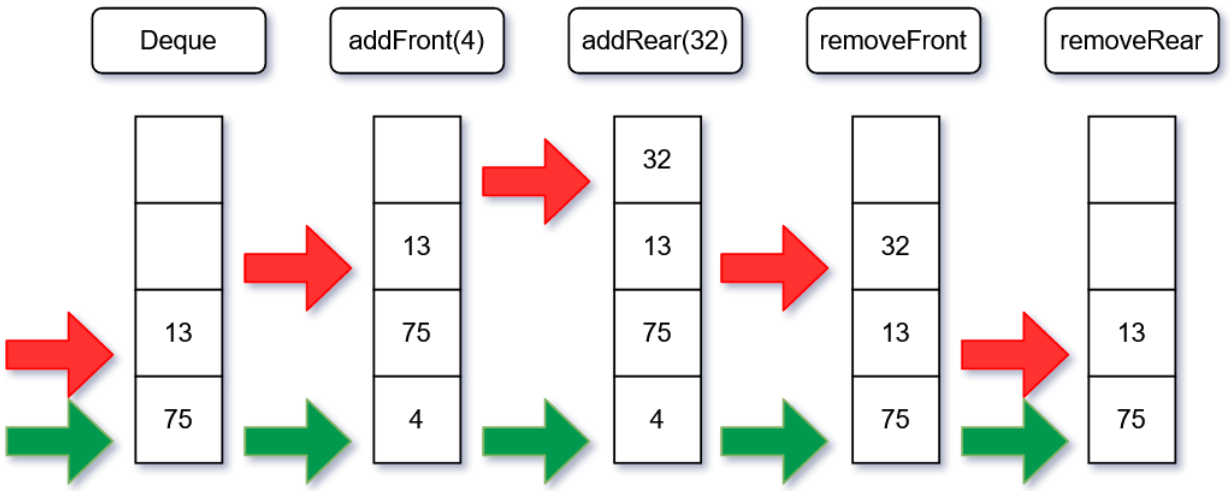
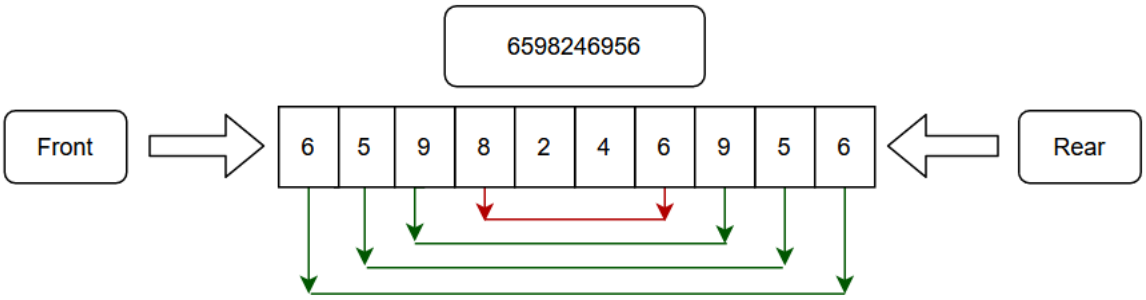


Figure 3 shows how the four principle methods of the **Deque DST** work (*addFront, addRear, removeFront, removeRear*). In the first position, the **Deque** only has two elements, the 75 (*Front*) and the 13 (*Rear*). In all the following examples the red arrow represents the *Rear* and the green one represents the *Front.* In the second position, 4 is inserted at the *Front* of the **Deque**, becoming the most priority element (the first one). On the next position 32 is inserted at the *Rear*, being the less priority element. On the two last positions happen two removal operations, first the removal of 4 (*Front*) and finally the removal of 32, that was on the *Rear*. This **DST** is basically a variation of the **Queue DST**, making them remarkably similar.

Examples:

In this first example, the objective is to create a code that uses one **deque** data structure that receives and input number or word and return True if the input is palindrome, False if not. A **deque** is like a **queue**, where it is possible to insert elements not only on the *rear*, but also on the *front*. It is also possible to remove elements from both positions. Now that this new data structure is explained let us jump into the real problem. How can we check if an input number or word is a palindrome?

Figure 2 – Check if a string is palindrome



The idea is simple, looking at figure 2, we can see that the string is 6598246956. The first step is to add all the digits to the **dequeue**, in order. After the insertion of all the elements, the real process starts. The idea is to remove the element of the *Front* and of the *Rear*, comparing them. If the comparison returns True, the algorithm repeats. If the comparison returns False, the algorithm stops. If the algorithm is repeated $int\left(\frac{n}{2}\right)$ times, it is because the string is a palindrome (with n being the length of the string). On figure 2, we see that the first three comparisons return True, but the fourth comparison returns False, ending the algorithm.

Another good example of how queues can be used to solve real life problems are the social security queues. Imagine you go to the social security, based on your ID number the program directs you to one of the queues (priority and regular). If you have one of the following conditions, pregnancy, 65+ years, have any disability or health related condition, you are placed into the **priority queue**. In any other case, you are placed into the **"regular"** queue (Queue for the people that do not have at least one of the following characteristics).

So, for example, you are pregnant, you are added to the **priority queue**. This means that to get called and removed from the **priority queue**, you must wait for everyone in your queue to get called. But if do not have any of the 3 characteristics referred before, you will be added to the **"regular queue"**. To get called, you must wait until everyone from the **priority queue** and for everyone that is front of you, in the **"regular" queue**, to get called.

Basic methods (queue and deque):

- **Enqueue (queue) –** This method helps us adding elements to our queue, using the function append to add the elements at the end of the collection data type.
- **Dequeue (queue) –** We use this method to remove elements from our queue. First, we check if the queue is empty, if it is not, we remove the element in the first position of our collection data type. If it is, print "Queue is empty!".
- **addFront (deque) –** This method is used to insert elements to the front of our deque. We use the function insert with the first argument 0, to add the elements at the first position of collection data type.
- **addRear (deque) -** This method does the same as the method above, the only difference is that the elements are added at the end of the collection data type (instead of being added to the first position).
- **removeFront (deque) -** We use this method to remove elements from the front of our deque. First, we check if the deque is empty, if it is not, we remove the element in the first position of our collection data type. If not, print "Deque is empty!".
- **removeRear (deque) -** This method does the same as the method above, the only difference is that if the deque is not empty we remove the element at the end of the collection data type (instead of being added to the first position).
- **Is_empty (queue & deque) –** Returns "True" if the Queue is empty and "False" if not.
- **Size (queue & deque) -** Returns the number of elements in the Queue.
- **Clear (queue & deque) -** Restarts the Queue.