SACHIN PRASANNA
211IT058

Camlin Page
Date / /

# ASSIGNMENT - 4

## 1 Queue

We implemented our Queue through a SINGLY linked list.
It follows the FIFO assignment.

## 2 enQueue ()

This function inserts an element at the end of a queue.

One pointer (rear) keeps track of the end element.
So, if we want to enqueue an element, we just have to update the rear pointer.

Updating the rear pointer takes constant time.

So, Worst case Time Complexity → O(1)

## 3 deQueue ()

This function removes an element at the front of the queue.

Again the first element is referenced through a pointer, and when an element is dequeued, this pointer is updated to its next.
This again takes constant time, Since just a pointer is updated.

Date / /

- The dequeued node is also deleted, which again takes constant time.

Hence, Worst case Time Complexity → O(1)

## (3) isEmpty()

- This function checks if the Queue is empty.

- To do this, it just checks if the front pointer is pointing to NULL.

- This again takes constant time as it is just an if condition.

So, Worst Case Time Complexity → O(1)

## (4) Size()

- This functions returns the number of element in the queue.

- It works on the principle of traversing the linked list.

- We ~~start~~ Start from the front pointer and keep moving it till the front pointer reaches the rear pointer.
We exit when front pointer equals rear pointer.

LL → Linked List

Since we have traversed the linked list (queue),

The Worst Case Time Complexity → O(n)

n → size of queue / LL

③ Display ()

This function prints all the elements in queue.

Since we have to traverse each element one by one, ~~through the~~

the Worst Case Time Complexity becomes → O(n)

n → size of queue / LL

④ De Que

We implemented a deque through a DOUBLY Linked List.
We can do more operations in a deque than a Simple queue.

① enQueue Front ()

~~Enqueue~~ This function inserts an element in the beginning of the

deque.

This involves altering pointers, that is if we are inserting a newNode, this newNode will become the new front of the queue.

This operation of making
newNode -> next = front
front -> prev = newNode
front = newNode.

All take constant time.

So, Worst case Time Complexity → O(1)

(2) enQueueRear ()

This function inserts an element in the end of the deque

The pointer rear which keeps track of the last node, needs to be updated to the newNode itself.

And finally, the newNode becomes the new rear.

```
rear -> next = newNode;
newNode -> prev = rear;
rear = newNode.
```

All this takes constant time.

So, Worst Case Time Complexity → O(1)

(3) de Queue Front ()

This function removes the first element of the deque.

As before, when the first element is removed, front is moved to its subsequent element.

Now, the dequeued element is deleted (freed).

All this takes constant time, since only pointers are being altered.

So, Worst Case Time Complexity → O(1)

(4) de Queue Rear ()

This function removes the last element of the deque.

In this case, the rear pointer (the pointer which keeps tracks of the last element) is moved to the previous element. (the element preceding it).

Finally, the element to be dequeued is deleted (freed).

All this takes constant time, since only pointers are being altered.

So, Worst Case Time Complexity → O(1)

(5) Size ()

This function returns the number of element in the deque.

It works by traversing the deque and stops when front pointer equals the rear pointer.

Since traversing is an O(n) time operation,

Worst Case Time Complexity → O(n)
n → size of deque

(6) ~~is Empty ()~~ is Empty ()

This function checks whether the ~~queue~~ deque is empty.

If front == NULL, then the deque is empty or else it is not empty.

Since it is just an if operation,
Worst Case Time Complexity → O(1)

Camlin Page
Date / /

(7) display ()

- This function prints all the element present in the deque.

- We traverse through each element and print it. This is O(n) in nature.

So, Worst case Time complexity → O(n)

n → size of deque