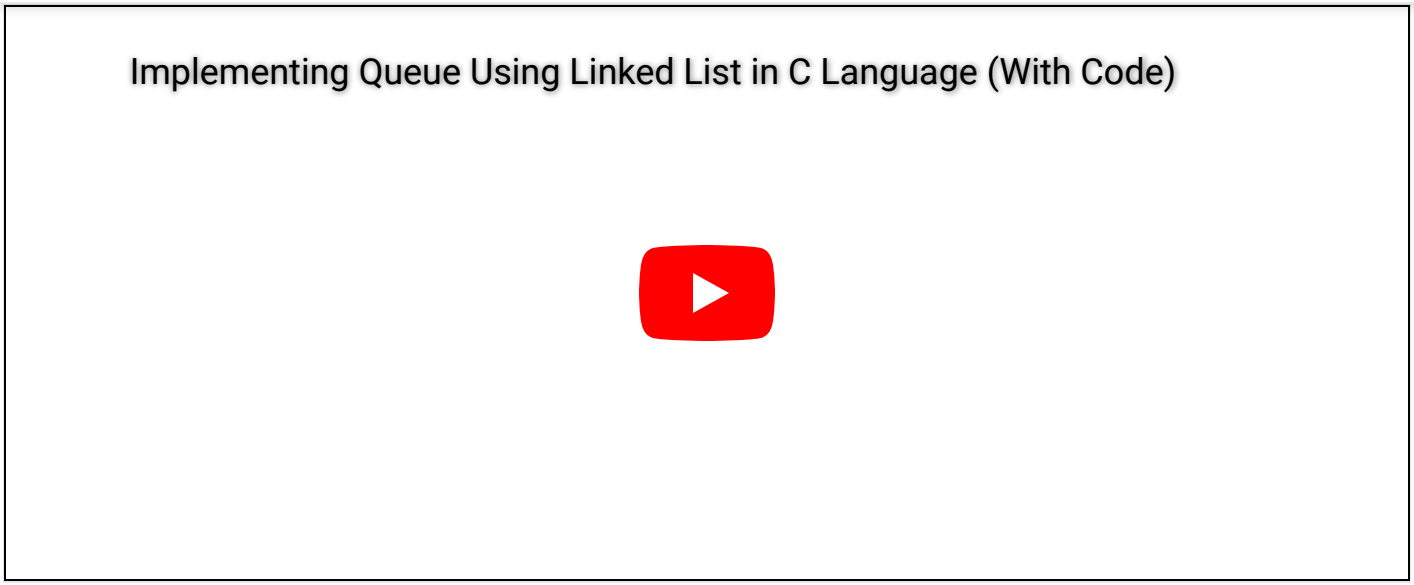


Implementing Queue Using Linked List in C Language (With Code)

In the last lecture, we learned to implement queues using linked lists. We talked about the enqueue and dequeue methods. In the queue linked list, we saw the conditions for its full or empty state. Today, we'll code these implementations in C. Before proceeding, make sure you have finished till here. I would recommend seeing that first if you somehow missed the last lecture since we discussed the concepts there.

Let's move onto our editors. I have attached the source code below for your reference.

Understanding the below code snippet:

1. First of all, I'll make you all aware of the things we have already completed. And I'll make you feel confident about linked lists and queues.
2. We had studied linked lists before, where we studied the traversal methods, insertion at different positions, deletion at different positions, cases of empty

and full. And we have completed queues as well and their basic operations.

3. Today, we'll integrate our knowledge of both to implement queues using linked lists.
4. We don't need to copy everything we learned there in the linked lists. We will move with the basics, and this might feel like a revision of the past lectures.
5. Create a struct Node with two of its members, one integer variable *data* to store the data, and another struct Node pointer *next* to store the address of the next node.

```
struct Node
{
    int data;
    struct Node *next;
};
```

Code Snippet 1: Creating the struct Node

6. Globally, create two struct Node pointers *f and r,* which would be used to mark the front and the rear ends. Declaring globally helps us use them in functions.

Creating Enqueue:

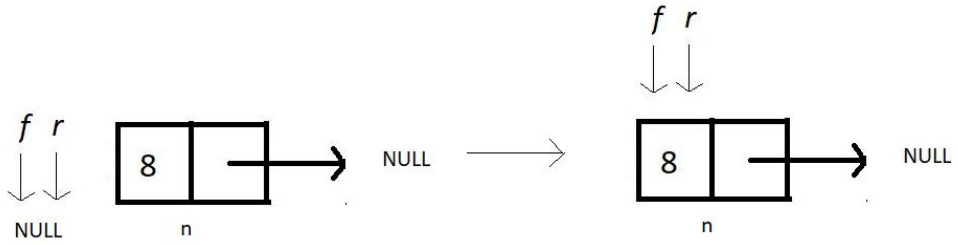
We learned in the last lecture that to enqueue, we only use the rear pointer and add a new node at the end of the list. So, create a void function *enqueue,* and the value to enqueue is the only parameter since we have declared the pointers *f and r* globally. In the function, create a new struct Node pointer *n,* and assign its memory in heap dynamically using malloc. Don't forget to include the header file<stdlib.h>. Then check if the queue is full or, in other words, if there is no space in the heap. And that can be done by checking if the new pointer *n* equals NULL. If it does, then print the condition of the queue overflow and return. Else, insert the val in the *data* member of *n,* and make this node point to NULL. If you recall, we discussed a special case, where we were inserting in the list for the first time, when both *f and r* equals NULL. For this case, make both *f and r* equal to *n,* and for all the other cases, just make the *r* point the new node *n.* Ultimately make r equal to n since n becomes our new rear end. And that would be all.

```
void enqueue(int val){
```

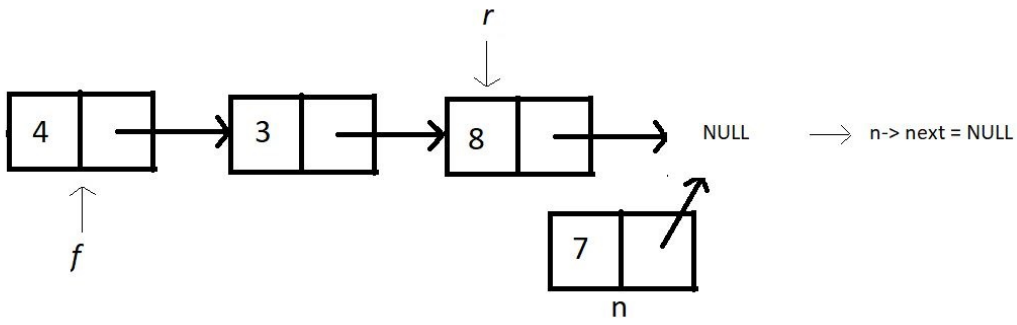
```
        struct Node *n = (struct Node *) malloc(sizeof(struct Node));
        if(n==NULL){
            printf("Queue is Full");
        }
        else{
            n->data = val;
            n->next = NULL;
            if(f==NULL){
                f=r=n;
            }
            else{
                r->next = n;
                r=n;
            }
        }
    }
```

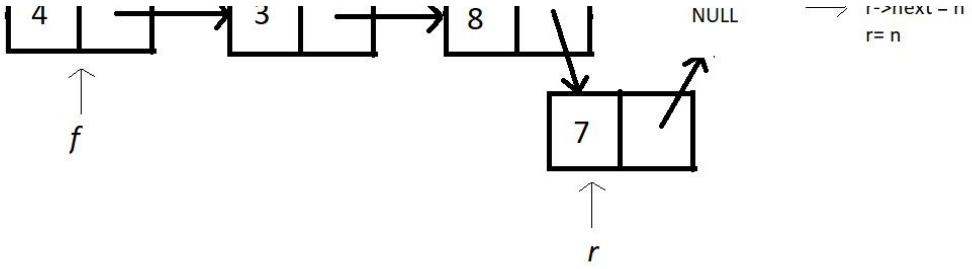
Code Snippet 2: Creating the enqueue function

Exception case:



All the other cases:





Creating Dequeue:

As we discussed in the last lecture, Dequeue needs you to just delete the head node, which is the *f node here. So,* create an integer function *dequeue.* And we have no parameters to pass. Create a struct Node pointer *ptr* to hold the node we will delete. Make *ptr* equal to *f.* In the function, check if the queue is already not empty by checking if our front *f* is NULL or not. If it is NULL, then print the condition of the queue underflow and return. Else, make *f* equal to the next node to *f.* Store the data of *ptr* in an integer variable *val.* We can now free the pointer *ptr.* And return *val,* which is the data of the node we deleted.

```
int dequeue()
{
    int val = -1;
    struct Node *ptr = f;
    if(f==NULL){
        printf("Queue is Empty\n");
    }
    else{
        f = f->next;
        val = ptr->data;
        free(ptr);
    }
    return val;
}
```

Code Snippet 3: Creating the dequeue function

After every operation, we may need to have the traversal function, which you can copy from the previous lectures. Nothing in that needs to be modified.

Here is the whole source code:

```c
#include <stdio.h>
#include <stdlib.h>

struct Node *f = NULL;
struct Node *r = NULL;

struct Node
{
    int data;
    struct Node *next;
};

void linkedListTraversal(struct Node *ptr)
{
    printf("Printing the elements of this linked list\n");
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

void enqueue(int val)
{
    struct Node *n = (struct Node *) malloc(sizeof(struct Node));
    if(n==NULL){
        printf("Queue is Full");
    }
    else{
        n->data = val;
        n->next = NULL;
        if(f==NULL){
            f=r=n;
```

```
        }
        else{
            r->next = n;
            r=n;
        }
    }
}

int dequeue()
{
    int val = -1;
    struct Node *ptr = f;
    if(f==NULL){
        printf("Queue is Empty\n");
    }
    else{
        f = f->next;
        val = ptr->data;
        free(ptr);
    }
    return val;
}

int main()
{
    linkedListTraversal(f);
    printf("Dequeuing element %d\n", dequeue());
    enqueue(34);
    enqueue(4);
    enqueue(7);
    enqueue(17);
    printf("Dequeuing element %d\n", dequeue());
    printf("Dequeuing element %d\n", dequeue());
    printf("Dequeuing element %d\n", dequeue());
    printf("Dequeuing element %d\n", dequeue());
```

```
    linkedListTraversal(f);
    return 0;
}
```

Code Snippet 4: Implementing queues using linked lists

Now, let's check if these methods work well. First, we'll enqueue some elements in the queue, and to check if that actually happens, we'll display the elements using traversal.

```
    enqueue(34);
    enqueue(4);
    enqueue(7);
    enqueue(17);
    linkedListTraversal(f);
```

Code Snippet 5: Using the enqueue function

And the output we received was:

```
Printing the elements of this linked list
Element: 34
Element: 4
Element: 7
Element: 17
```

Figure 1: Output of the above program

Let us now dequeue everything. And focus on the order of elements being dequeued.

```
    printf("Dequeuing element %d\n", dequeue());
    printf("Dequeuing element %d\n", dequeue());
    printf("Dequeuing element %d\n", dequeue());
    printf("Dequeuing element %d\n", dequeue());
    linkedListTraversal(f);
```

Code Snippet 6: Using the dequeue function

And the output this time was;

```
Dequeuing element 34
Dequeuing element 4
Dequeuing element 7
Dequeuing element 17
Printing the elements of this linked list
```

Figure 2: Output of the above program

As you can observe, there was no element left to display, and element 34 went out first since it entered the first. And here, we finish the implementation of queues using linked lists. The **queue** has been holding us all for quite some time now. Ah, that metaphor!

I appreciate your support throughout. I hope you enjoyed the tutorial. If you genuinely appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial, where we'll learn briefly about another data structure called double-ended queues. Till then, keep coding.

Previous Next