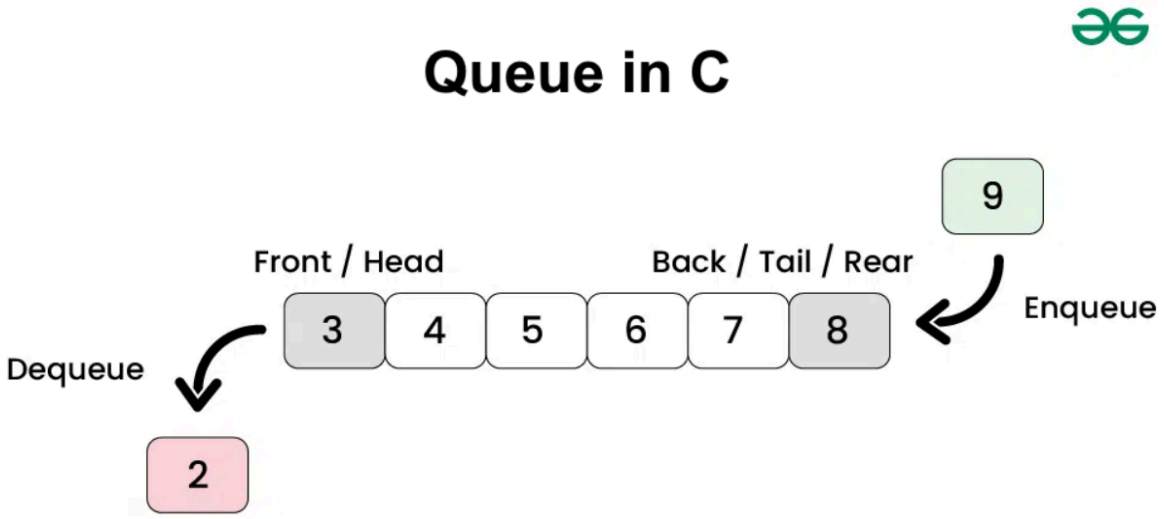
A queue represented by a linked list uses a dynamically allocated chain of nodes, where each node contains data and a pointer to the next node. Two pointers, `front` and `rear`, track the beginning and end of the queue, respectively, enabling First-In-First-Out (FIFO) behavior. New elements are added at the `rear` (enqueue), and elements are removed from the `front` (dequeue).

Queue is a linear data structure that follows the First-In-First-Out (FIFO) order of operations. This means the first element added to the queue will be the first one to be removed. There are different ways using which we can implement a queue data structure in C.
In this article, we will learn how to implement a queue using a linked list in C, its basic operations along with their time and space complexity analysis, and the benefits of a linked list queue in C.

Linked List Implementation of Queue in C

A queue is generally implemented using an array, but the limitation of this kind of queue is that the memory occupied by the array is fixed no matter how many elements are in the queue. In the queue implemented using a linked list, the size occupied by the linked list will be equal to the number of elements in the queue. Moreover, its size is dynamic, meaning that the size will change automatically according to the elements present.



Representation of Linked Queue in C

In C, the queue that is implemented using a linked list can be represented by pointers to both the front and rear nodes of the linked list. Each node in that linked list represents an element of the queue. The type of linked list here is a singly linked list in which each node consists of a data field and the next pointer.

Basic Operations of Linked List Queue in C

Following are the basic operations of the queue data structure that help us manipulate the data structure as needed:

Enqueue Function

The enqueue function will add a new element to the queue. To maintain the time and space complexity of O(1), we will insert the new element at the end of the linked list. The element at the front will be the element that was inserted first.

We need to check for queue overflow (when we try to enqueue into a full queue).

Algorithm for Enqueue Function

Following is the algorithm for the enqueue function:

- *Create a new node with the given data.*
- *If the queue is empty, set the front and rear to the new node.*
- *Else, set the next of the rear to the new node and update the rear.*

Dequeue Function

The dequeue function will remove the front element from the queue. The front element is the one that was inserted first, and it will be present at the front of the linked list.

We need to check for queue underflow (when we try to dequeue from an empty queue).

Algorithm for Dequeue Function

Following is the algorithm for the dequeue function:

- *Check if the queue is empty.*
- *If not empty, store the front node in a temporary variable.*
- *Update the front pointer to the next node.*
- *Free the temporary node.*
- *If the queue becomes empty, update the rear to NULL.*

Peek Function

The peek function will return the front element of the queue if the queue is not empty. The front element is the one at the front of the linked list.

Algorithm for Peek Function

The following is the algorithm for the peek function:

- *Check if the queue is empty.*
- *If empty, return -1.*
- *Else, return the front->data.*

IsEmpty Function

The isEmpty function will check if the queue is empty or not. This function returns true if the queue is empty; otherwise, it returns false.

Algorithm of isEmpty Function

The following is the algorithm for the isEmpty function:

- *Check if the front pointer of the queue is NULL.*
- *If NULL, return true, indicating the queue is empty.*
- *Otherwise, return false, indicating the queue is not empty.*

1. Structure Definitions

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

struct Queue {
    struct Node* front;
    struct Node* rear;
};
```

- **Node**: Each node holds an integer (`data`) and a pointer to the next node (`next`).
- **Queue**: Keeps track of two pointers — `front` (where we remove elements) and `rear` (where we add elements).

---

2. Creating an Empty Queue

```
struct Queue* createQueue() {
    struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
    q->front = q->rear = NULL;
    return q;
}
```

- Allocates memory for a `Queue` structure.
- Initializes both `front` and `rear` to `NULL` because the queue is empty initially.
- Returns the pointer to the created queue.

---

3. Enqueue Operation (Add element to rear)

```
void enqueue(struct Queue* q, int value) {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->data = value;
    temp->next = NULL;

    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }

    q->rear->next = temp;
    q->rear = temp;
}
```

- **Create a new node** (`temp`) with the data to insert and `next` set to `NULL`.
- **If the queue is empty** (`rear == NULL`), both `front` and `rear` point to the new node (first element).

- **Otherwise**, link the current `rear` node's `next` to the new node, then update `rear` to the new node.

---

4. Dequeue Operation (Remove element from front)

```
int dequeue(struct Queue* q) {
    if (q->front == NULL) {
        printf("Queue is empty\n");
        return -1;
    }

    struct Node* temp = q->front;
    int data = temp->data;
    q->front = q->front->next;

    if (q->front == NULL)
        q->rear = NULL;

    free(temp);
    return data;
}
```

- **Check if queue is empty** (`front == NULL`), if yes print message and return error code `-1`.
- Store current `front` node in `temp`.
- Store its data for return.
- Move `front` pointer to the next node.
- **If the queue becomes empty** after removing (i.e., `front` is `NULL`), set `rear` to `NULL` as well.
- Free the old `front` node memory.
- Return the removed data.

---

5. Display Queue Elements

```
void display(struct Queue* q) {
    struct Node* temp = q->front;
    if (temp == NULL) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

- Start from the `front` node.
- If queue is empty (`front == NULL`), print a message.
- Otherwise, traverse the linked list from `front` to `rear`, printing each `data` element.

---

6. Main Function Usage

```
int main() {
    struct Queue* q = createQueue();

    enqueue(q, 10);
    enqueue(q, 20);
    enqueue(q, 30);
    display(q);

    printf("Dequeued element: %d\n", dequeue(q));
    display(q);

    enqueue(q, 40);
    display(q);

    return 0;
}
```

- Creates a queue.
- Enqueues elements 10, 20, 30.
- Displays queue → Output: `10 20 30`.
- Dequeues one element (should remove 10).
- Displays queue → Output: `20 30`.
- Enqueues 40.
- Displays queue → Output: `20 30 40`.

Summary: