Data Structures

BS (CS) _Fall_2024

Lab_08 Manual

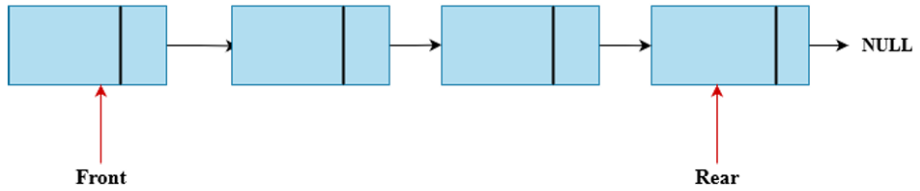
Learning Objectives:

1. Queue

Introduction to Queues

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. In a queue, the first element added to the queue is the first one to be removed. Queues are often used to manage tasks or data in a way that ensures that the oldest item is processed or removed before newer items.

A common way to implement a queue is by using a linked list. In a linked list-based queue, each element of the queue is represented as a node in the linked list.



Queue Operations

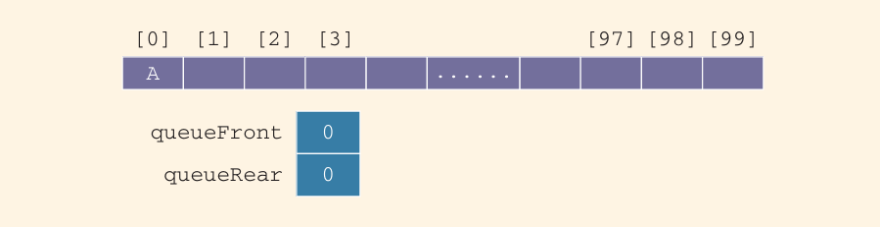
Some of the queue operations are:

- **initializeQueue**: Initializes the queue to an empty state.
- **isEmptyQueue**: Determines whether the queue is empty. If the queue is empty, it returns the value true; otherwise, it returns the value false.
- **isFullQueue**: Determines whether the queue is full. If the queue is full, it returns the value true; otherwise, it returns the value false.
- **front**: Returns the front, that is, the first element of the queue. Input to this operation consists of the queue. Prior to this operation, the queue must exist and must not be empty.
- **back**: Returns the last element of the queue. Input to this operation consists of the queue. Prior to this operation, the queue must exist and must not be empty.
- **addQueue**: Adds a new element to the rear of the queue. Input to this operation consists of the queue and the new element. Prior to this operation, the queue must exist and must not be full.
- **deleteQueue**: Removes the front element from the queue. Input to this operation consists of the queue. Prior to this operation, the queue must exist and must not be empty.

Implementation of Queue as Arrays

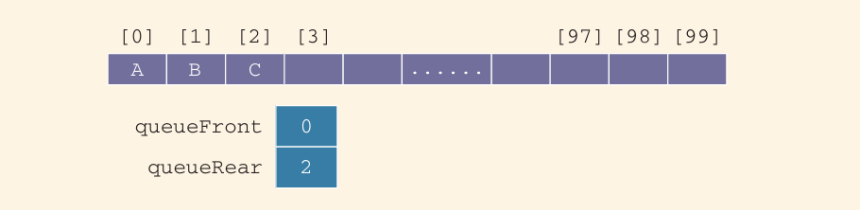
Let us see what happens when queueFront changes after a deleteQueue operation and queueRear changes after an addQueue operation. Assume that the array to hold the queue elements is of size 100.Initially, the queue is empty. After the operation:

addQueue(Queue,'A');



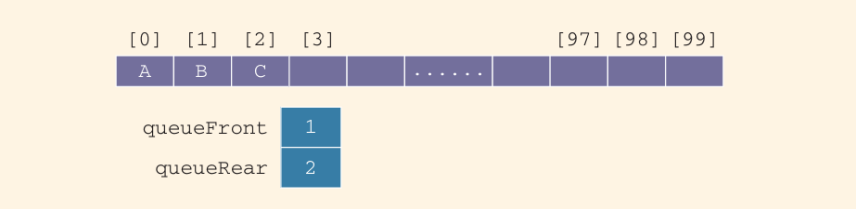
After two more addQueue operations:

addQueue(Queue,'B');
addQueue(Queue,'C');



Now consider the delete Queue operation:

deleteQueue();



Working of Queue

Queue operations work as follows:

- two variables FRONT and REAR
- FRONT track the first element of the queue
- REAR track the last element of the queue
- initially, set value of FRONT and REAR to -1

```
Queue() {
    front = -1;
    rear = -1;
  }
```

Enqueue Operation

- check if the queue is full
- for the first element, set the value of FRONT to 0
- increase the REAR index by 1
- add the new element in the position pointed to by REAR

```
void enQueue(int element) {
    if (isFull()) {
      cout << "Queue is full";
    } else {
      if (front == -1) front = 0;
      rear++;
     items[rear] = element;
    }
}
```

Dequeue Operation

- check if the queue is empty
- return the value pointed by FRONT
- increase the FRONT index by 1
- for the last element, reset the values of FRONT and REAR to -1

```
int deQueue() {
    if (isEmpty()) {
      return (-1);
    } else {
      Int element = items[front];
      if (front >= rear) {
        front = -1;
        rear = -1;
      } /* Q has only one element, so we reset the queue after deleting it.
*/
      else {
        front++;
      }
      return (element);
    }
  }
```

Implementation of Queue as Linked List

We maintain two pointers, front, and rear. The front points to the first item of the queue and rear points to the last item.

```
// Node class representing a single node in the linked list

      class Node {
      public:
          int data;
          Node* next;
          Node(int new_data)
          {
              this->data = new_data;
              this->next = nullptr;
          }
      };
```

- enQueue(): This operation adds a new node after the rear and moves the rear to the next node.

```
      void enqueue(int new_data) {

              // Create a new linked list node
              Node* new_node = new Node(new_data);

              if (this->isEmpty()) {
                  // If queue is empty, the new node is both the front and rear
                  return;
              }

              // Add the new node at the end of the queue and
              // change rear
      }
```

- deQueue(): This operation removes the front node and moves the front to the next node

```
      void dequeue() {

        // If queue is empty, return
        if (this->isEmpty())
            return;
        // Store previous front and move front one node ahead

        if (front == nullptr)
            // If front becomes nullptr, then change rear also to nullptr

        // Deallocate memory of the old front node
        delete old_front;
      }
```