Introduction to Programming and Data Structures

Queue

Malay Bhattacharyya

Associate Professor

MIU, CAIML, TIH
Indian Statistical Institute, Kolkata

October, 2023

Motivating example

Problem: Maintain a list of patients waiting to consult a doctor.
Data structure: Queue (FIFO)
Operations:

- `INSERT` or `ENQUEUE`: insert at the end
- `DELETE` or `DEQUEUE`: remove and return element from the front
- `DISPLAY`: Show the elements
- `LENGTH` or `SIZE`
- `ISEMPTY`, `ISFULL`

Standard implementation – Initialization

Considering the queue size as static:

```
QueueDS = []
SizeQueueDS = 10
```

Standard implementation – Insert operation

```
def QueueINSERT(QueueDS, Data):
    if len(QueueDS) < SizeQueueDS:
        QueueDS.insert(0, Data)
    else:
        print("Queue overflow!!!")
```

Standard implementation – Delete operation

```
def QueueDELETE(QueueDS):
    if len(QueueDS) > 0:
        Data = QueueDS.pop()
        return Data
    else:
        print("Queue underflow!!!")
```

Note: If you do not wish to receive the deleted element, exclude the return statement.

Standard implementation – Displaying the elements

```
def QueueDISPLAY(QueueDS):
    print("The elements in the queue are: ")
    for Element in QueueDS:
        print(Element)
```

Note: You can directly write `print(QueusDS)`.

Efficient implementation – Initialization

Considering the queue size as static:

```
QueueDS = []
SizeQueueDS = 10
```

Efficient implementation – Insert operation

```
def QueueINSERT(QueueDS, Data):
    global rear
    if (rear + 1) % SizeQueueDS == front:
        print("Queue overflow!!!")
    else:
        QueueDS[rear] = Data
        rear = (rear + 1) % SizeQueueDS
```

Efficient implementation – Delete operation

```
def QueueDELETE(QueueDS):
    global front, rear, SizeQueueDS
    if front == rear:
        print("Queue underflow!!!")
    else:
        Data = QueueDS[front]
        front = (front + 1) % SizeQueueDS
        return Data
```

Note: If you wish to receive the deleted element, include a return statement.

Efficient implementation – Displaying the elements

```
def QueueDISPLAY(QueueDS):
    print("The elements in the queue are: ")
    if front <= rear:
        print(QueueDS[front:rear])
    else:
        print(QueueDS[front:] + QueueDS[:rear])
```

Note: You cannot directly write `print(QueusDS)`.

Breadth-first Search (BFS)

We can perform breadth-first search on a graph given in the form of adjacency list in a file. The BFS algorithm works as follows:

i) Insert any one of the graph's vertices in the front of a queue.
ii) Delete the rear data item from the queue and add it to the `Visited` list.
iii) Create a list of that vertex's adjacent nodes. Add the ones which are not in the `Visited` list to the front of the queue.
iv) Repeat steps (ii)-(iii) until the queue is empty.

Implementing BFS

```
# Adjacency list defined as a dictionary
Graph = {
    '0' : ['1','2'], '1' : ['3', '4'], '2' : ['5'],
    '3' : [], '4' : ['5'], '5' : []
}
Visited = [] # Array to keep track of visited vertices
def BFS(Visited, Graph, Vertex):
    if Vertex not in Visited:
        Visited.insert(0, Vertex)
        print(Visited[0]) # Front of queue
        for Adjacent in Graph[Vertex]:
            BFS(Visited, Graph, Adjacent)
BFS(Visited, Graph, '0') # Function call with vertex '0'
```

Problems

1. Given an array and an integer S, find the minimum for each and every contiguous subarray of size S.
2. Given a list of numbers, find out the first non-repeating number in constant time (i.e., it should not depend on the total number of elements in the list).
3. Given a queue having multiple elements, write a recursive function to reverse its elements.