COP 3502 – Computer Science 1

Lecture 06

Dr. Sahar Hooshmand
sahar.hooshmand@ucf.edu

Department of Computer Science

Slides modified from Dr. Ahmed, with permission

Content

- Abstract Data Type (ADT)
- Queue
  - Applications of Queue
  - Queue Operations and Implementation
  - Example and Simulation

Abstract Data Type (ADT)

- Can you give me some examples of data type?
- What happens when you declare a variable of a data type? (e.g., int a;)
- What can you do with an integer variable?
  - You can just store data and get data from there
  - It does not provide any functionalities to store and retrieve the data in a specific way
- What is an array?
- ADT is a data structure and a set of operations which can be performed on it
  - Example: Stack, Queue, Linked List

Queue

What is QUEUE?

- A Queue is a First In, First Out (FIFO) ADT
- Anything added goes to the “rear or back”
- Anything removed from the Queue is taken from the “front”
- Things are removed in the same order from that in which they were inserted
- Analogy:
  - Vehicles in a one way single lane road
  - Other examples?

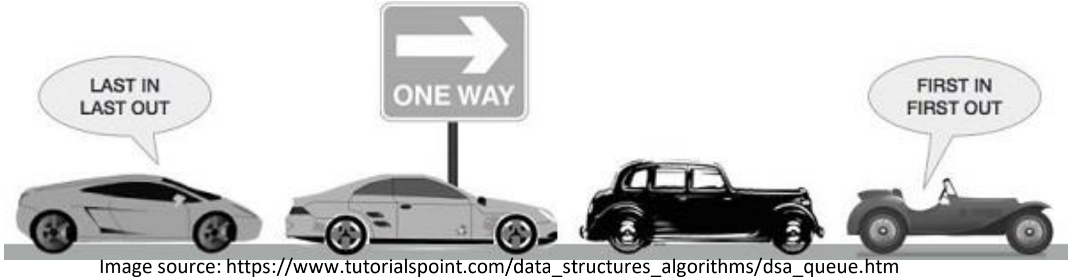


Image source: https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm

Applications of Queue

- Variety of applications of Queue:
  - Network printer
  - Job scheduling by OS
  - Call center phone call queue
  - Queue of people at any service point such as ticketing etc.
  - Algorithm such as BFS (Breath First Search)

Queue operations

- Basic Queue operations
  - **enQueue :** Add a new item
  - **deQueue :** Get and remove an from the front of the Queue
  - **IsEmpty:** Determine whether the Queue is empty
  - **IsFull:** Determine whether the Queue is full
  - **Peek:** Get the element at the front of the queue without removing it

Queue Implementation

- Queues can be implemented using:
  - Array
  - Linked list

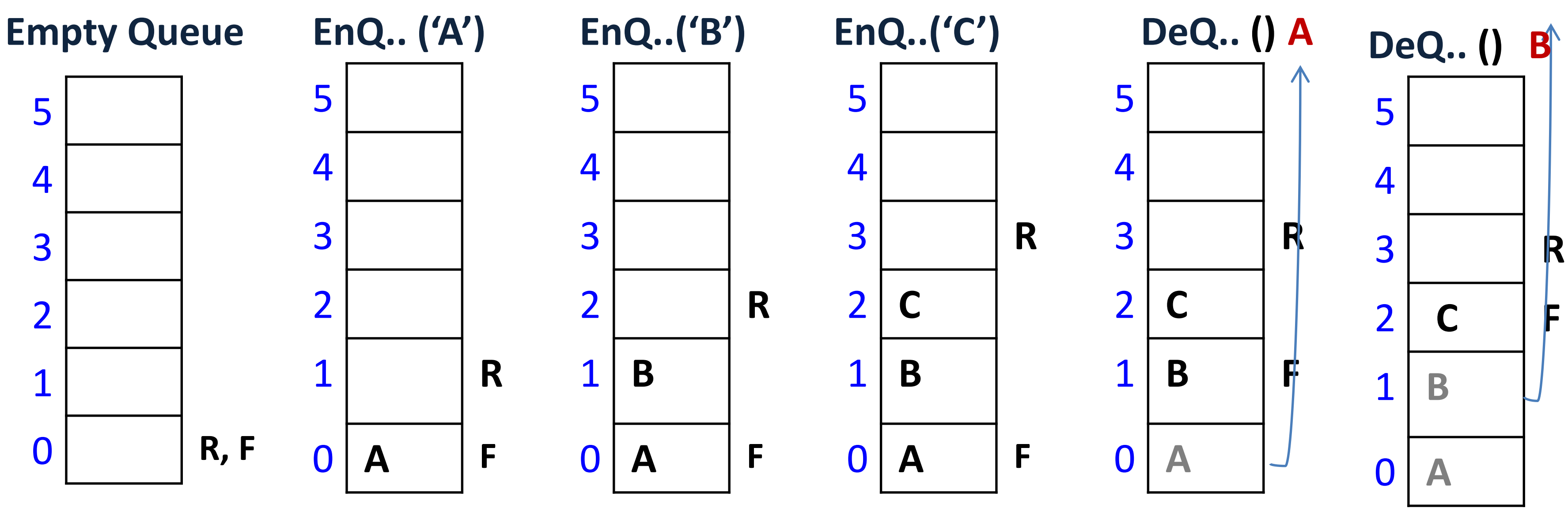
- We will see both of the implementations

Array based Queue Implementation

- There can be two types of array based implementation of Queues:
  - Linear Queue: Cannot re-use the empty spaces after deleting data
  - Circular Queue: Re-use empty slots from the front
- We will learn about both of them in next couple of slides!

Example of Linear Queue operations

- Max_Size = 6, Queue[Max_Size], Front = 0, Rear=0
- EnQueue(char x): Queue[Rear++] = x,
- DeQueue(): return Queue[Front++],
- Peek(): return Queue[Front]



- Discussion: DeQue(), DeQue()
- When Queue is full? Or Queue is empty?

Example Queue Implementation

```
void EnQueue(int x)
 {
     if( Rear < Max_Size)
      {
        Queue[Rear++] = x;
        printf("Inserted");
      }
     else
         printf("Full");
}
```

```
int DeQueue()
{
    if(Front==Rear)
     {
       printf("Empty");
       return -9999;
     }
    else
        return Queue[Front++];
}
```

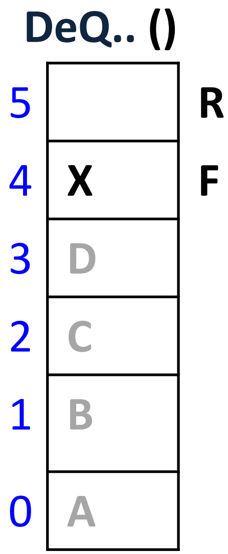
```
int isEmpty()
{
    The logic is already available in the
    DeQueue() function above
}
```

```
int isFull()
{
    The logic is already available in the
    EnQueue() function above
}
```

Note about DeQueue

- In an array based implementation of DeQueue, we are not really deleting the data from the front, we are just hiding it.
- While printing the Queue, you should print only valid Queue part of the array!

Linear Queue vs. Circular Queue



- Consider the Queue shown in the the left side
- Can you add more items in this Queue?
- Yes we can add an item in R and R becomes 6.
- Then, can you add more?
- We cannot add more as R > Max_size
- Is it really full?
  - See F is at 4, it means we have removed the items from 0 to 3.
- How about utilizing the empty spaces and make it circular?
- In order to go back to the empty spaces, we can use **Mod (%) operation**
- Still we need to check whether the queue is full or empty, for enque and deque
- If we can track how many elements do we have, it can help us to keep track whether the queue full or it has empty slot or not

Let’s review mod (%) operation

0	1	2	3	4	5
G	F	**I**	**K**	**A**	C

F ↑

- The size of the above array is 6.
- Mod (%) operation gives you the remainder of the division.
- What is the value of :
  - 0%6 = 0
  - 1%6 = 1
  - 2%6 = 2
  - 3%6 = 3
  - 4%6 = 4
  - 5%6 = 5
  - 6%6 = 0 //wow it is coming back to zero!
- What is the value of 7%6?

Understanding Circular Queue-EnQueue

- Consider the following Queue

0	1	2	3	4	5
G	F	**I**	**K**	**A**	C

F ↑ (at index 2)

- The **size or capacity** of the Queue is 6
- Number of elements (**noe**) is 3
- What items do we have in the Queue?
  - Read 3 elements starting from front (I, K, A)
- **EnQueue('L'):**
  - Where L should be inserted?
  - At 5 as the last item is at 4.
  - But, how can you generate that index, where L need to be inserted?
  - Isn't it at the position **F + noe** ?
  - **2 + 3 = 5, so L will be added to index 5**
  - **And we have to increase noe++; (So noe is now 4)**

0	1	2	3	4	5
G	F	**I**	**K**	**A**	**L**

F ↑ (at index 2)

Understanding Circular Queue-EnQueue

0	1	2	3	4	5
G	F	I	K	A	L

F ↑

- Now, <u>noe</u> is 4
- **<u>EnQueue ('M'):</u>**
  - Do we have space to insert M? (Yes, we have at 0 and 1)!, but how do you know? ***(noe<capacity)***
  - Where M should be inserted?
  - At 0 as the last item is at 5, so we go back to the beginning
  - But, how can you generate that index?
  - Isn't it the position at (**F + noe)%size** ?
  - (2 + 4)%6 = ?
  - So, L will be added to index 0
  - **And we have to increase noe++; <u>(So noe is now 5)</u>**

0	1	2	3	4	5
M	F	I	K	A	L

F ↑

Understanding Circular Queue-EnQueue

0	1	2	3	4	5
M	**F**	**I**	**K**	**A**	**L**
		F ↑			

- Now, **noe** is 5
- EnQueue (‘N’):
  - Do we have space to insert N? (Yes, we have at 1)! ***(noe<capacity)***
  - Where N should be inserted?
  - At 1 as the last item is at 0
  - But, how can you generate that index?
  - (F + noe)%size = (2 + 5)%6 = 7%6 = 1
  - So, N will be added to index 1
  - **And we have to increase noe++; (So noe is now 6)**

0	1	2	3	4	5
M	**N**	**I**	**K**	**A**	**L**
		F ↑			

Understanding Circular Queue-EnQueue

0	1	2	3	4	5
M	**N**	**I**	**K**	**A**	**L**
		F ↑			

- Now, **noe** is 6
- EnQueue (‘O’):
  - Do we have space to insert N?
    - Is noe<size (or capacity)?
  - So, we don’t have space and the Queue is FULL!
  - So, what can we do?
    - We can give a message to the user that it is full and we can stop further insertion
    - Or we can realloc and increase the capacity!

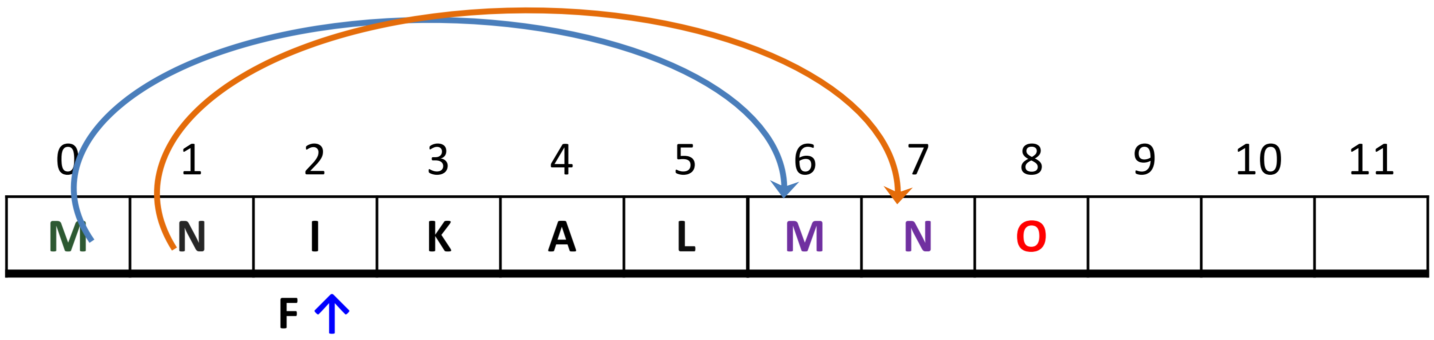
Challenges using realloc in EnQueue

- Let's say, we use realloc when the queue is full and we increase the size to size*2

0	1	2	3	4	5	6	7	8	9	10	11
M	**N**	**I**	**K**	**A**	**L**						
		F ↑									

- So, now the size or capacity is 12.
- noe = 6
- If we keep processing EnQueue ('O'), the index will be (f+ noe)%size = (2+6)%12 = 8
- However, there will be a gap between L and O (index 6 and 7 are empty)
- It will mess-up our queue.
- Also, if I ask you to print the Queue, you will start at Front and keep printing until L, and then it is confusing should I go back to 0 or should I continue. We still have spaces in the Queue after 5, so whey to go back to 0?
- So, what is the solution?
  - Copy everything before F to after 5 (which actually depend on the old size)

Challenges using realloc in EnQueue



- So, the solution is:
  - After realloc
  - Copy all the data before F (if any) to the index starting from the old size and afterwards
  - Finally, add the item to the next slot
- What will happen to the position before F?
  - They will be re-used after completing the insertion up to the new size using our mod formula.

Summary of EnQueue Steps

- Get the item you want to insert
- Check if the queue is full or not by comparing the noe and the size
  - If not full
    - Add the item at index (F+noe)%size
    - Increase noe
  - If full:
    - Realloc with more capacity
    - Copy data from the left of Front to the right side
    - Add the item after that
    - Increase noe
    - Increase capacity

Understanding Circular Queue-DeQueue

- As the main criteria of a queue, we always remove data from the FRONT
- Steps:
  - Check if the queue is empty
    - If noe == 0, queue is empty!
  - Copy the data from the front into val
  - Increase front by:
    - front = (front+1)%size //to make it circular
  - noe—
  - return val

Example of DeQueue

- Consider the following Queue.
  - Size = 6, noe = 3

0	1	2	3	4	5
M	N	I	K	**A**	**L**
				F ↑	

- **DeQueue():**
  - **Queue is not empty**
  - **Val = ‘A’**
  - **F = (F + 1)%6 = (4+1)%6 = 5**
  - **noe-- (noe will become 2)**
  - **Return val (return ‘A’)**

0	1	2	3	4	5
M	N	I	K	A	**L**
					F ↑

Example of DeQueue

- Size = 6, noe = 2

0	1	2	3	4	5
M	N	I	K	A	**L**

F ↑ (under index 5)

- **DeQueue():**
  - **Queue is not empty**
  - **Val = ‘L’**
  - **F = (F + 1)%6 = (5+1)%6 = 0**
  - **noe-- (noe will become 1)**
  - **Return val (return ‘L’)**

0	1	2	3	4	5
M	N	I	K	A	L

F ↑ (under index 0)

Example of DeQueue

- Size = 6, noe = 1

0	1	2	3	4	5
M	N	I	K	A	L

F ↑

- **DeQueue():**
  - **Queue is not empty**
  - **Val = ‘M’**
  - **F = (F + 1)%6 = (0+1)%6 = 1**
  - **noe-- (noe will become 0)**
  - **Return val (return ‘M’)**

0	1	2	3	4	5
M	N	I	K	A	L

F ↑

- **DeQueue():**
  Que is EMPTY as noe == 0

Circular Queue implementation

- Let us define a structure for Que:

```
struct queue {
    int* elements;
    int front;
    int numElements;
    int queueSize;
};
```

- An array for the elements of the queue
- An integer for the index into the front of the queue
- An integer for the number of elements in the queue
- An integer representing the current size of the queue

Circular Queue implementation

- We will implement the following functions to utilize our queues

- void init(struct queue* qPtr);
- int enqueue(struct queue* qPtr, int val);
- int dequeue(struct queue* qPtr);
- int empty(struct queue* qPtr);
- int peek(struct queue* qPtr);

- The functions and the code is pretty big
- The complete code will be available in webcourses
- We will discuss each of the functions in the class

Linked List implementation of Queue

- Implement Queue using Linked List
  - Where an enQueue item will be added?
  - Where a dQueue item will be deleted from?
  - What can be the drawback?
    - In case of array based implementation we just need to access one index of the array for both enQueue, and deQueue.
    - We did not need to go through all items for any of the cases
    - However, depending on your implementation, either enQueue or DeQueue any one of them will need to access all the items to reach to the end of the list.
    - O(1) vs O(n) // we will learn about Big-O in another lecture

Linked List implementation of Queue

- Maintaining two pointer can help:
  - One for front of the list
  - One for the back
- One way to think about this is that our struct to store the queue would actually store two pointers to linked list structs.
  - The first would point to the head of the list and
  - the second would always point to the last node in that list.

```
//stores one node of the linked list
struct node {
   int data;
   struct node* next;
};
```

```
// Stores our queue.
struct queue {
          struct node* front;
           struct node* back;
};
```

Linked List implementation of Queue

- **Let's consider how we'd carry out some operations:**

<u>init function:</u> this function should make front and rear of the que as NULL.

<u>enqueue</u>

- **1) Create a new node and store the inserted value into it.**
- **2) Link the back node's next pointer to this new node.**
- **3) Move the back node to point to the newly added node.**

<u>dequeue</u>

- **1) Store a temporary pointer to the beginning of the list**
- **2) Move the front pointer to the next node in the list**
- **3) Free the memory pointed to by the temporary pointer.**

<u>front</u>

- **1) Directly access the data stored in the first node through the front pointer to the list.**

<u>empty</u>

- **1) Check if both pointers (front, back) are null.**

- **A very good exercise for you would be to use the above concepts to implement queue using linked list. You must try it!**