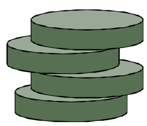
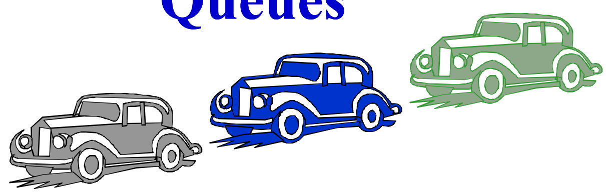
CSE 2215:
Data Structures and Algorithms-I

Stacks



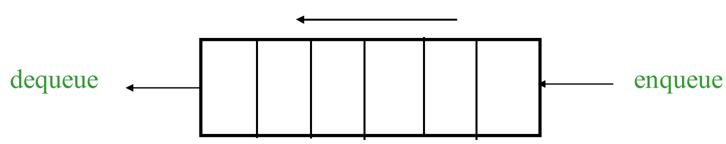
Queues

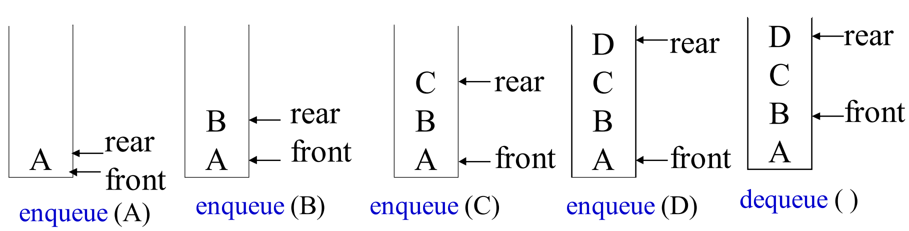


Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Queue: First In First Out

- A **Queue** is an ordered collection of items from which items may be removed at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (the ***rear*** of the queue).
- The operations: **enqueue** (insert) and **dequeue** (delete)





Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Applications of Queues

- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array Implementation of Queues

front = ~~-1~~ 0

rear = -1

myQueue:

0	1	2	3	4	5	6	7
~~10~~	~~25~~	~~35~~	~~22~~	✓	✓	✓	55

front (at 4), rear (at 3)

front=0, rear=0

rear = sizeQ-1

Overflow

rear >= sizeQ -1

enQ(10);

enQ(25); , rear = 1

enQ(35); rear = 2

deQ(); front=1 rear=2

deQ(); front=2 rear=2

enQ(22); front=2 rear=3

deQ() front=3 rear=3

deQ() front=4 rear=3

deQ()

Underflow

for rear+1 = front

Array Implementation of Queues

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)

	0	1	2	3	4	5	6	7
myQ:	17	23	97	44				

front = 0 rear = 3

- **To insert:** put new element in location 4, and set rear to 4

After insertion:	17	23	97	44	33			

front = 0 rear = 4

Array Implementation of Queues

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)

After insertion:	17	23	97	44	33			

front = 0 rear = 4

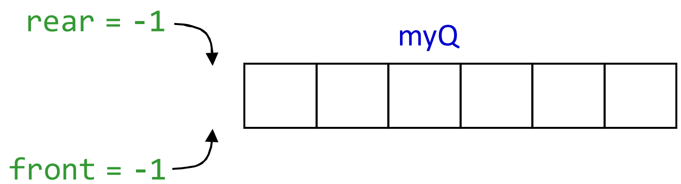
- **To delete:** take element from location 0, and set front to 1

After deletion:	17	23	97	44	33			

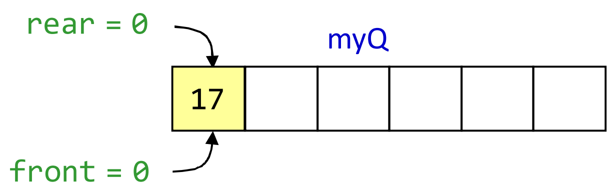
front = 1 rear = 4

Array Implementation: Empty Queue

- Initial Queue, that is Empty Queue

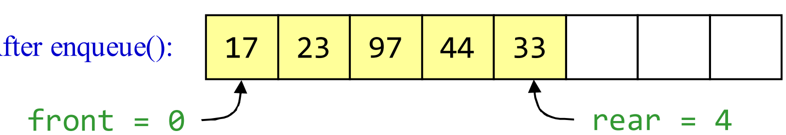


- After inserting 1st element in an Empty Queue, Set front = rear = 0



Array Implementation: Enqueue()

After enqueue():




```
void enQueue(int  x){
   if(rear  >= Qsize - 1)
      printf("\n Queue is over flow");
   else {
      rear++;
      myQ[rear] = x;
   }
}
```

Array Implementation: Dequeue()

After dequeue():

17	23	97	44	33			

front = 1 rear = 4

```
int deQueue() {
   int y;
   if(front  > rear)
      printf("\n Queue is under flow");
   else {
      y = myQ[front];
      front++;
      return y;
   }
}
```

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Array Implementation of Queues

front = 0 rear = 3

Initial queue:	17	23	97	44				
After enqueue:	17	23	97	44	33			
After dequeue:	17	23	97	44	33			

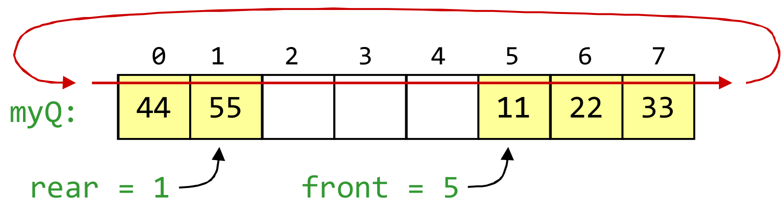
front = 1 rear = 4

- Notice how the array contents “crawl” to the right as elements are enqueued and dequeued
- This will be a problem after a while!

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Circular Queues using Arrays

- We can treat the array holding the queue elements as circular (joined at the ends)

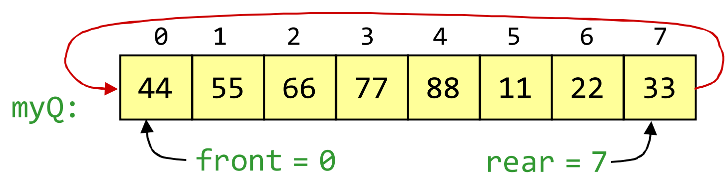


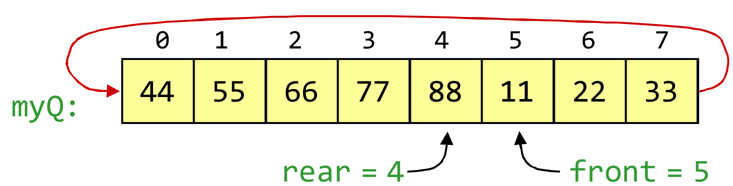
- Elements were added to this queue in the order 11, 22, 33, 44, 55, and will be removed in the same order
- Use: `front = (front + 1) % Qsize;`
  and: `rear = (rear + 1) % Qsize;`

Circular Queues: Full

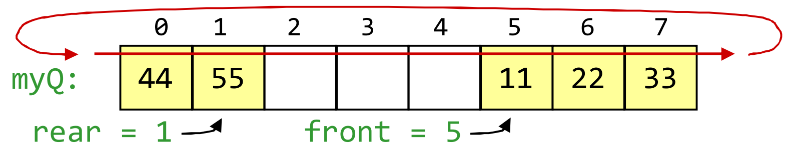
There are two cases in which Queue is Full:

- When front == 0 && rear == Qsize-1,
- When front == rear + 1;
  - ✓ (rear+1) % Qsize == front





Circular Queues using Arrays: EnQueue()

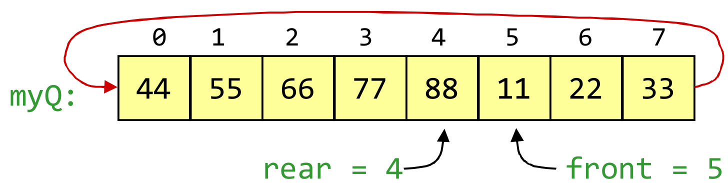



```
void enQueue(int data) {
   if(front == -1 && rear == -1) {  // queue is empty
      front = rear = 0;
      myQ[rear]=data;  }
   else if((rear+1) % Qsize == front)  // check queue is full
      printf("Queue is overflow");
   else {
      rear=(rear+1) % Qsize;      // rear is incremented
      myQ[rear] = data;     // assign a value
   }
}
```

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

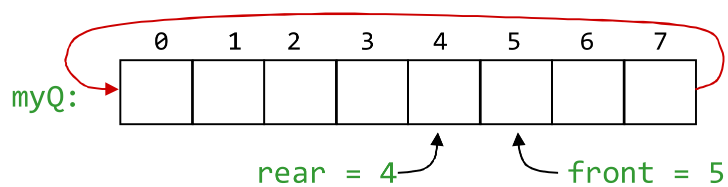
Circular Queues: Empty

- If the queue were to become completely full, it would look like this:



This is a problem!

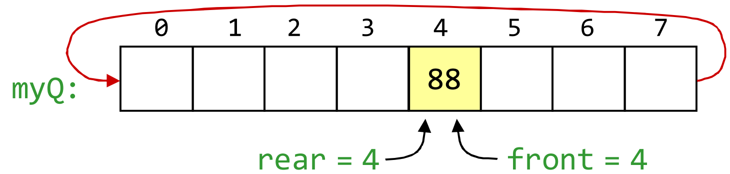
- Again, if we were to remove all eight elements, making the queue completely empty, it would look like this:



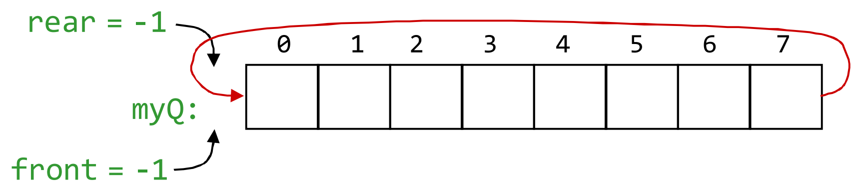
Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Full and Empty Circular Queues: Solutions

- When there is only one element left which is to be deleted, then the front is not incremented, rather the front and rear are reset to -1, i.e,
  - Set front = -1, and Set rear = -1 (Not front++)

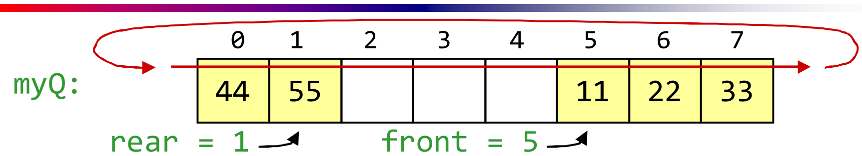


- After deQueue the last element, the empty Queue will be like this



Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Circular Queues using Arrays: deQueue()




```
int deQueue() {
 int y;
  if((front == -1) && (rear == -1)) {
     printf("\n Queue is underflow..");
  else if(front == rear) {   // there is only one element left
     y = myQ[front];  front = -1;  rear = -1;  }
  else  {
    y = myQ[front];
    front = (front+1) % Qsize;
  }
  return y;
}
```

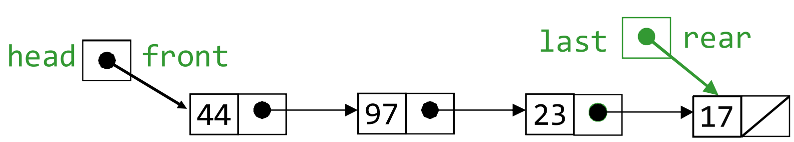
Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Linked-list Implementation of Queues

- In a queue, insertions occur at one end (rear end), deletions at the other end (front end).
- Operations at the head of a singly-linked list (SLL) are $O(1)$, but at the other end they are $O(n)$
  - Because you have to find the last element each time
- BUT: there is a simple way to use a singly-linked list to implement both insertions and deletions in $O(1)$ time
  - You always need a pointer to the *first* element in the list
  - You can keep an additional pointer to the *last* element in the list

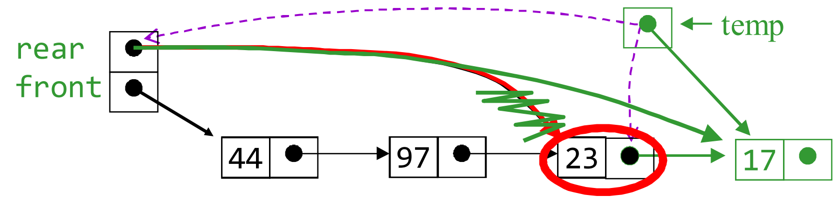
SLL Implementation of Queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it.



- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue

Queues by SLL: Enqueue

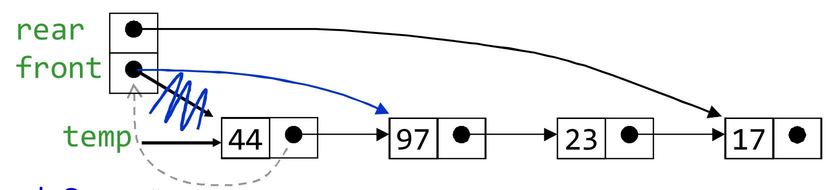



```
void enQueue(int data) {
    struct Node* temp;
    temp = (struct Node *)malloc(sizeof(struct Node));
    // Check if  memory(heap) is full.
    if (!temp){
        cout << "\n Heap Overflow";
        exit(1);
    }
    temp->value = data;
    rear->next = temp;
    rear = temp;
}
```

```
struct Node {
    int value;
    struct Node* next;
};
struct Node* front, rear;
```

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Queues by SLL: Dequeue




```
int deQueue(){
    struct Node* temp;
    int data;
    if (front == NULL) {
        cout << "\n Queue underflow";
        exit(1); }
    else {
        data = front->value
        temp = front;
        front = front->next;
        free(temp);
        return data;
    }
}
```

```
struct Node {
    int value;
    struct Node* next;
};
struct Node* front, rear;
```

Dr. Md. Abul Kashem Mia, Professor, CSE Dept and Pro-Vice Chancellor, UIU

Queue Implementation Details

- With an array implementation:
  - you can have both overflow and underflow
- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition