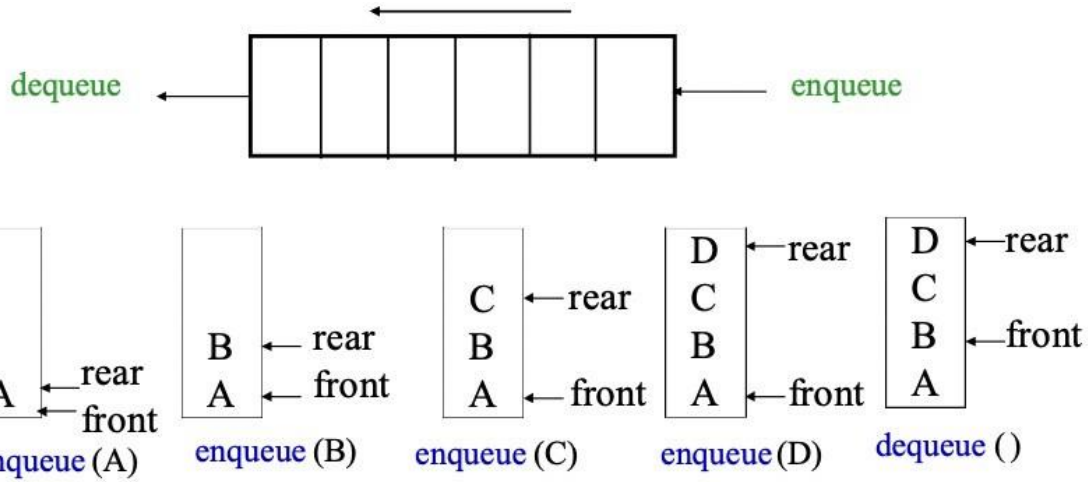
Queue

Charles Aunkan Gomes
Lecturer, Dept. of CSE
United International University
charles@cse.uiu.ac.bd

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)

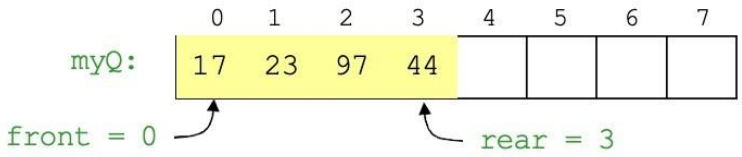


Queue: Applications

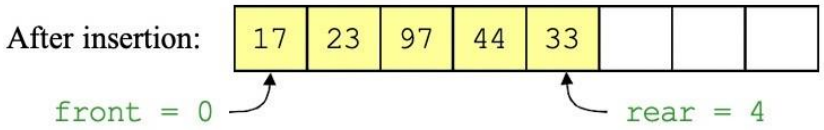
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Array Implementation of Queue

- 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)

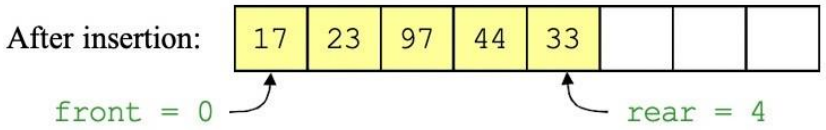


- **To insert:** put new element in location 4, and set rear to 4

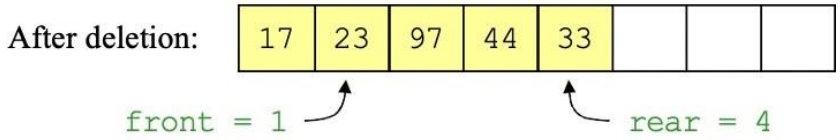


Array Implementation of Queue

- 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)

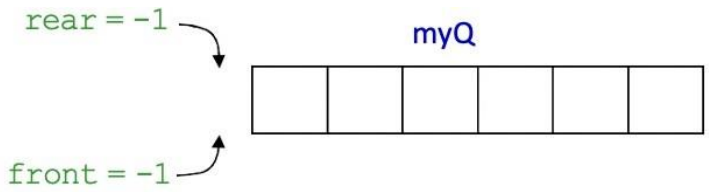


- **To delete:** take element from location 0, and set front to 1

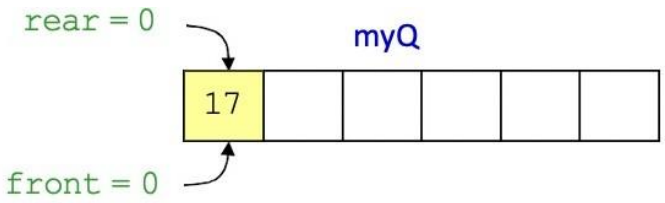


Array Implementation: Empty Queue

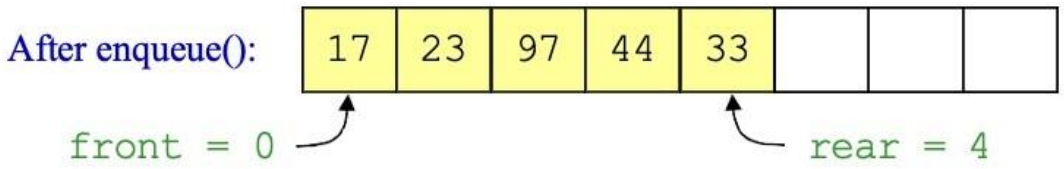
- Initial Queue, that is Empty Queue



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

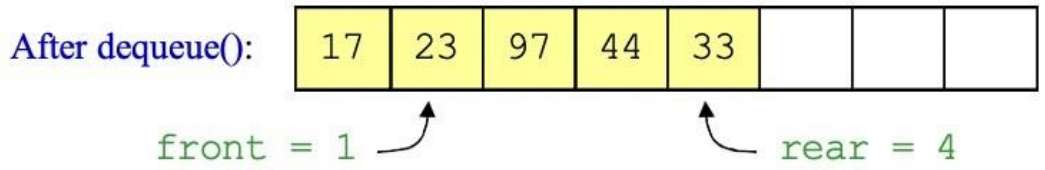


Array Implementation: Enqueue()



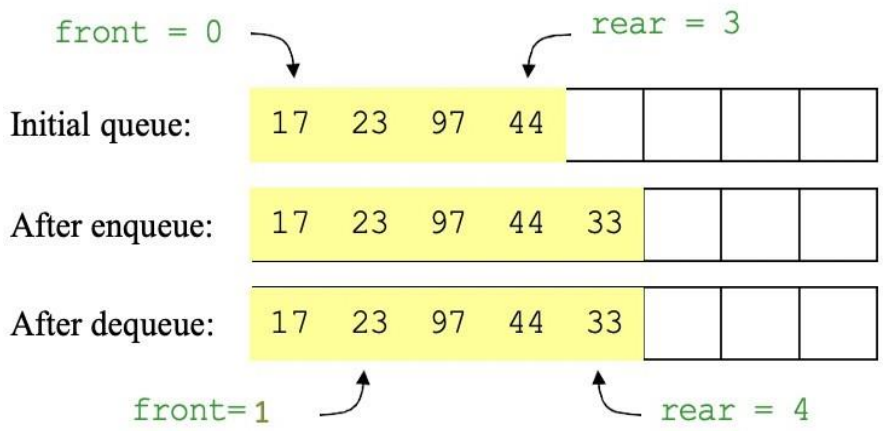

```
void enQueue(int x){
        if(rear >= Qsize - 1)
         printf("\n Queue is over flow");
        if( front == -1 && rear ==-1){
         front = rear = 0;
         myQ[rear] = x;
        }else {
              rear++;
              myQ[rear] = x;
              }
}
```

Array Implementation: Dequeue()




```
int deQueue() {
        int y;
        if((front > rear) || (front==-1 && rear==-1))
          printf("\n Queue is under flow");
        else {
          y = myQ[front];  front++;
          return y;
        }
}
```

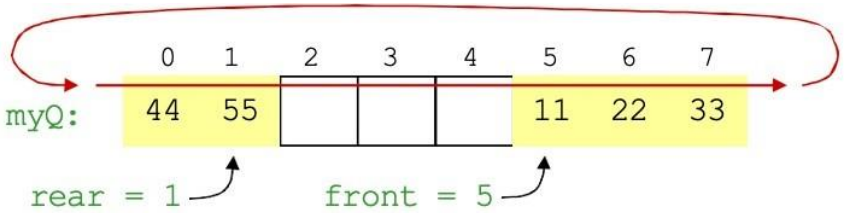
Array Implementation of Queues



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

Circular Queues using Array

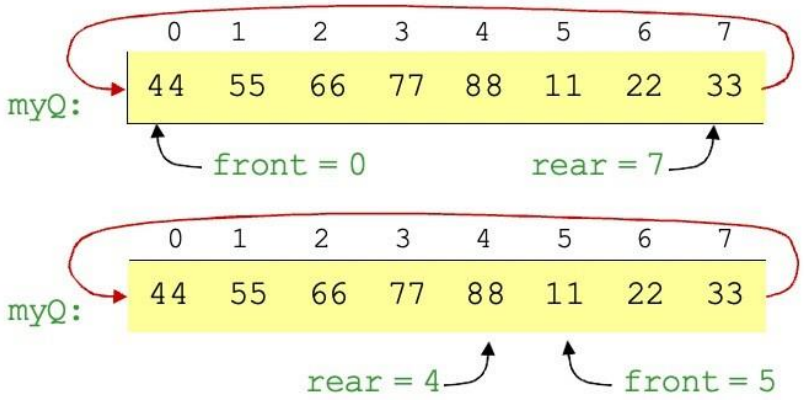
- We can treat the array holding the queue elements as circular (joined at the ends)



- 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;

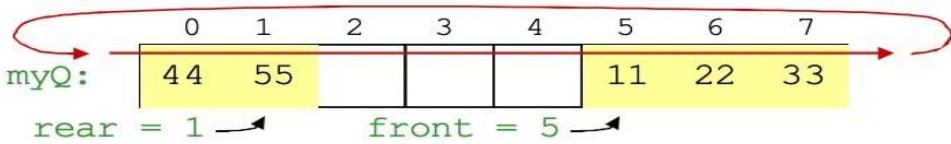
Circular Queues: Full

- There are two cases in which Queue is Full:
  - When front == 0 && rear == Qsize-1,
  - When front == rear + 1;
    - ✓(rear+1) % Qsize == front



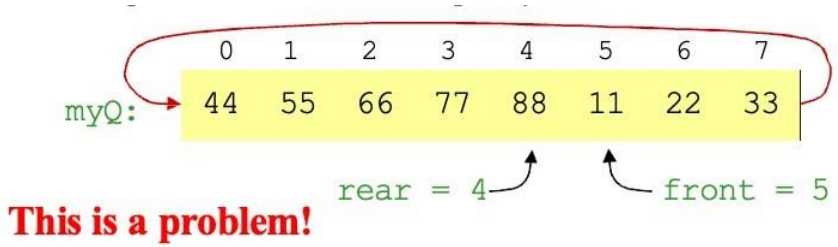
Circular Queue using Array: EnQueue()

```
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
        }
}
```

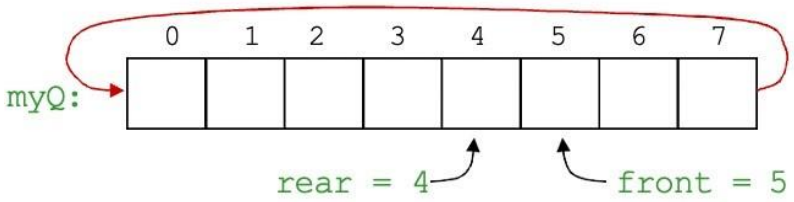


Circular Queue: Empty

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

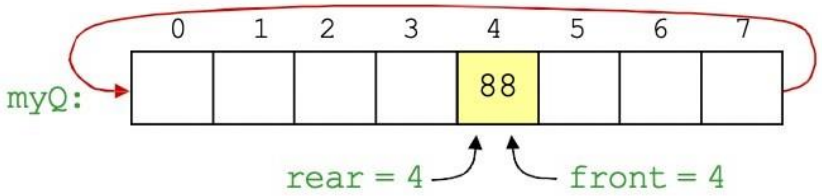


- Again, if we were to remove all eight elements, making the queue completely empty, it would look like this:

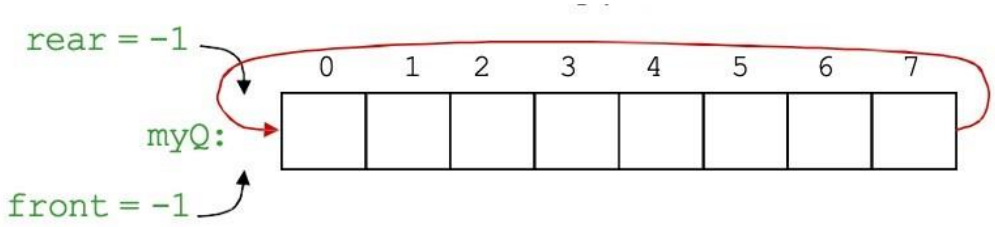


Empty Circular Queue: Solution

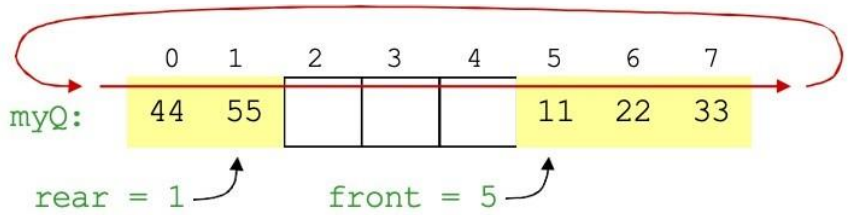
- When there is only one element left which is to be deleted, then the front is not incremented, rather the front and rear are reset to -1, i.e,
  - Set front = -1, and Set rear = -1 (Not front++)



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



Circular Queue using Array: 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 = rear = -1;
        }
        else {
                y = myQ[front];

                front = (front+1) % Qsize;
       }
       return y;
  }
```

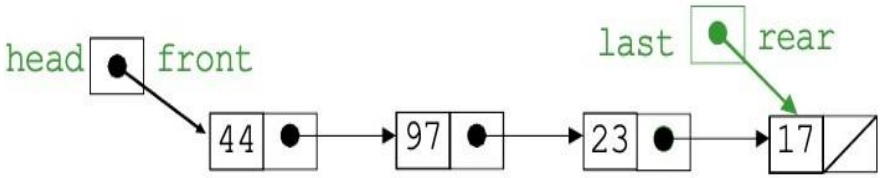
Linked List Implementation of Queue

- 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 Queue

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way

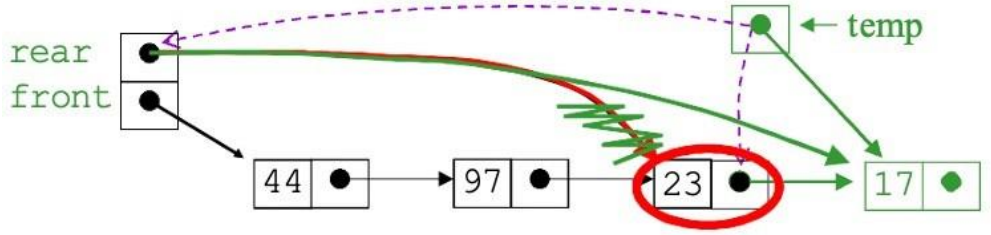
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it.



- 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

Queue by SLL: EnQueue()

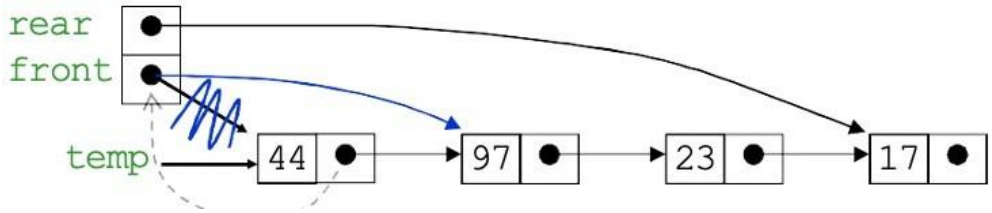
```
void enQueue(int data) {
        struct Node* temp;
        temp = (struct Node *)malloc(sizeof(struct Node));
        // Check if memory(heap) is full.
        if (!temp){
                cout << "\n Heap Overflow";
                exit(1);
        }
        temp->value = data;
        rear->next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



Queue 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;
        }

  }
```



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

THANK YOU