Data Structures

10. Queues

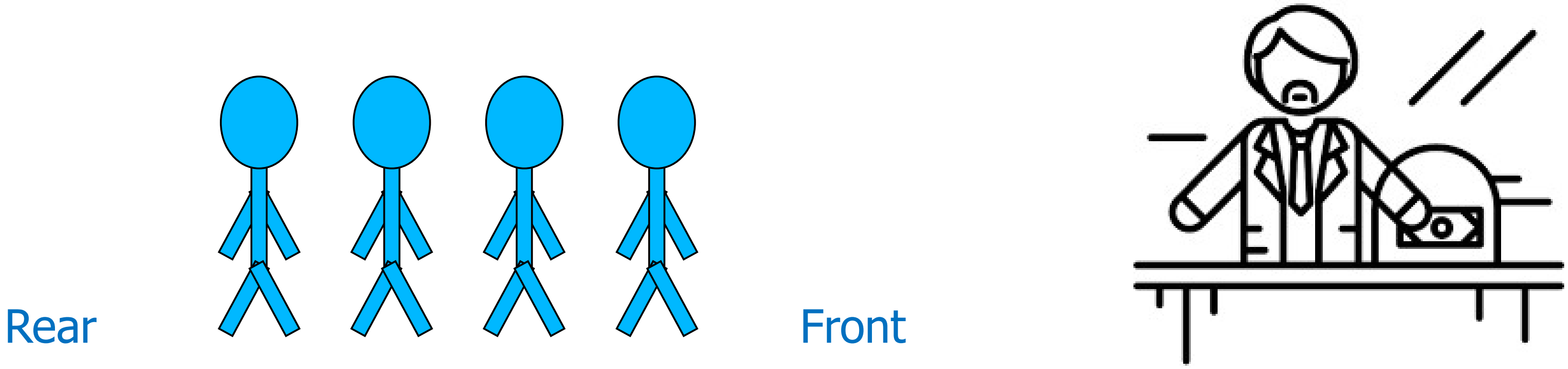
Queues

- Queue is First-In-First-Out (FIFO) data structure
  - First element added to the queue will be first one to be removed

- Queue implements a special kind of list
  - Items are inserted at one end (the rear)
  - Items are deleted at the other end (the front)

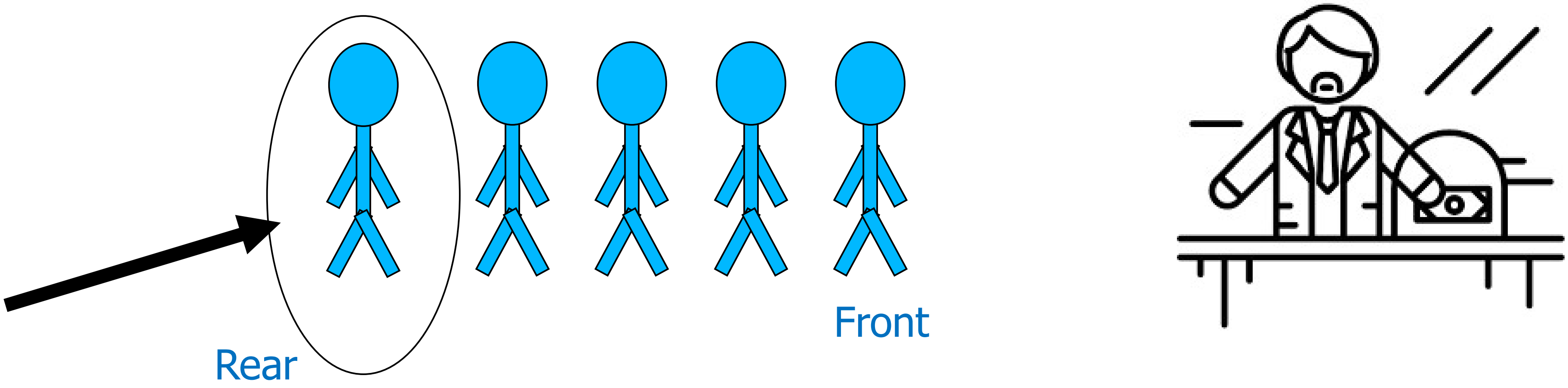
Queue – Analogy (1)

- A queue is like a line of people waiting for a bank teller
- The queue has a front and a rear



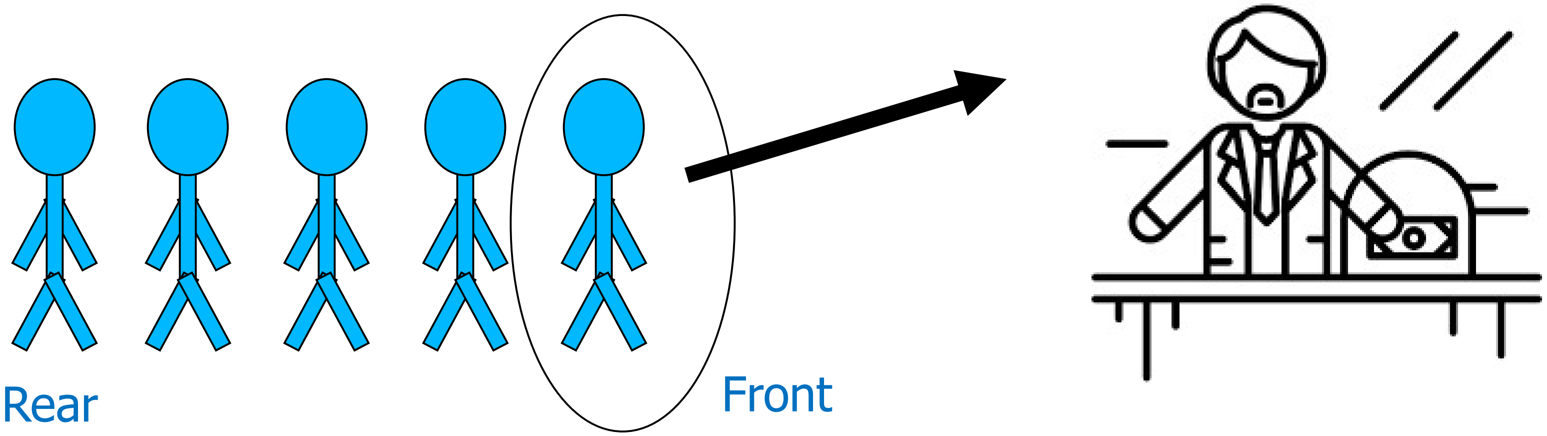
Queue – Analogy (2)

- New people must enter the queue at the rear



Queue – Analogy (3)

- An item is always taken from the front of the queue



Queues – Examples

- Billing counter
  - Booking movie tickets
  - Queue for paying bills
- A print queue
- Vehicles on toll-tax bridge
- Luggage checking machine
- And others?

Queues – Applications

- Operating systems
  - Process scheduling in multiprogramming environment
  - Controlling provisioning of resources to multiple users (or processing)

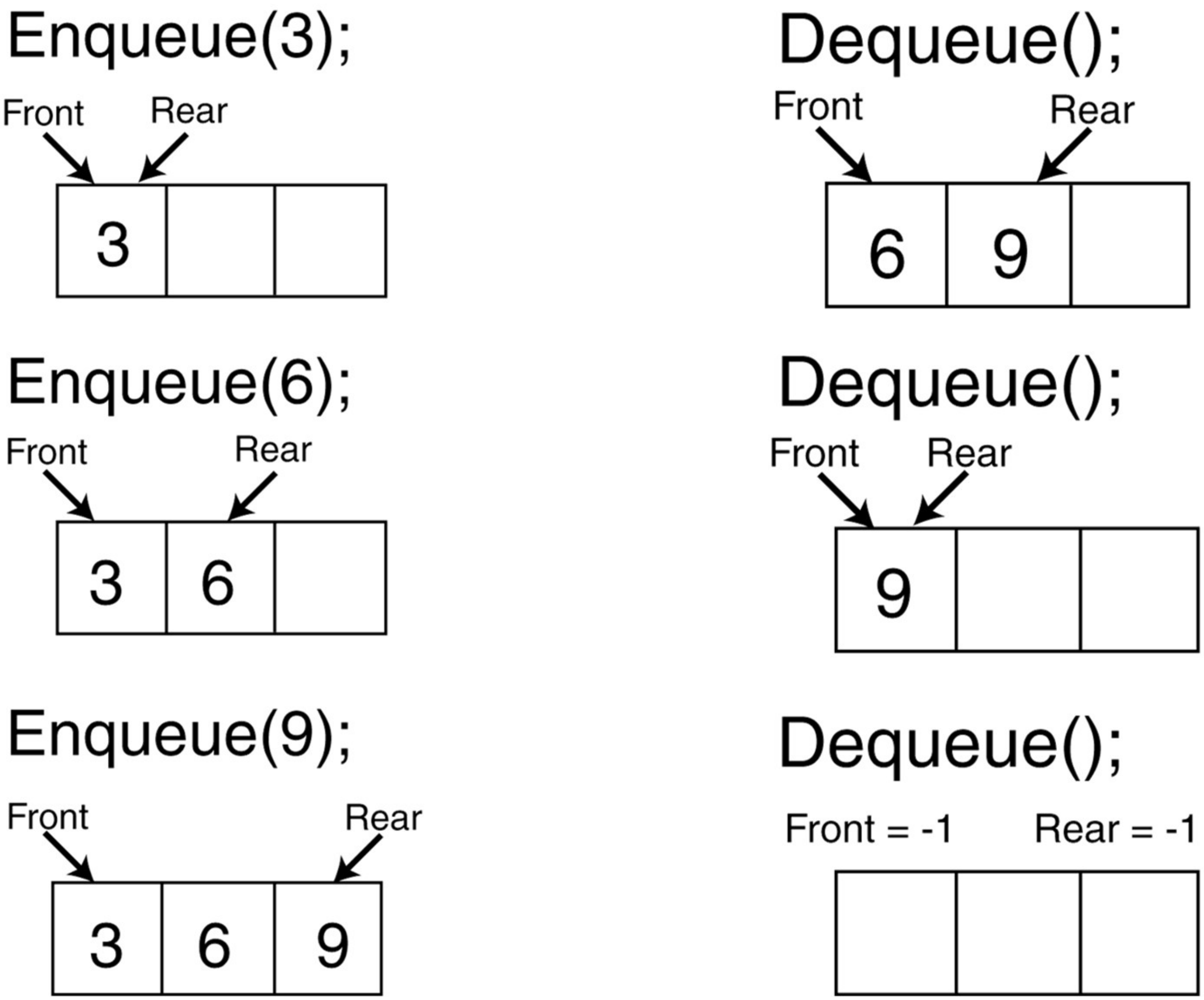
- Middleware/Communication software
  - Hold messages/packets in order of their arrival
    - ➢ Messages are usually transmitted faster than the time to process them

  - The most common application is in client-server models
    - ➢ Multiple clients may be requesting services from one or more servers
    - ➢ Some clients may have to wait while the servers are busy
    - ➢ Those clients are placed in a queue and serviced in the order of arrival

Basic Operations (Queue ADT)

- `MAKENULL(Q)`
  - Makes Queue Q be an empty list
- `FRONT(Q)`
  - Returns the first element on Queue Q
- `ENQUEUE(x,Q)`
  - Inserts element x at the end of Queue Q
- `DEQUEUE(Q)`
  - Deletes the first element of Q
- `EMPTY(Q)`
  - Returns true if and only if Q is an empty queue

Enqueue And Dequeue Operations



Enqueue(3);

Front Rear

3

Enqueue(6);

Front Rear

3 6

Enqueue(9);

Front Rear

3 6 9

Dequeue();

Front Rear

6 9

Dequeue();

Front Rear

9

Dequeue();

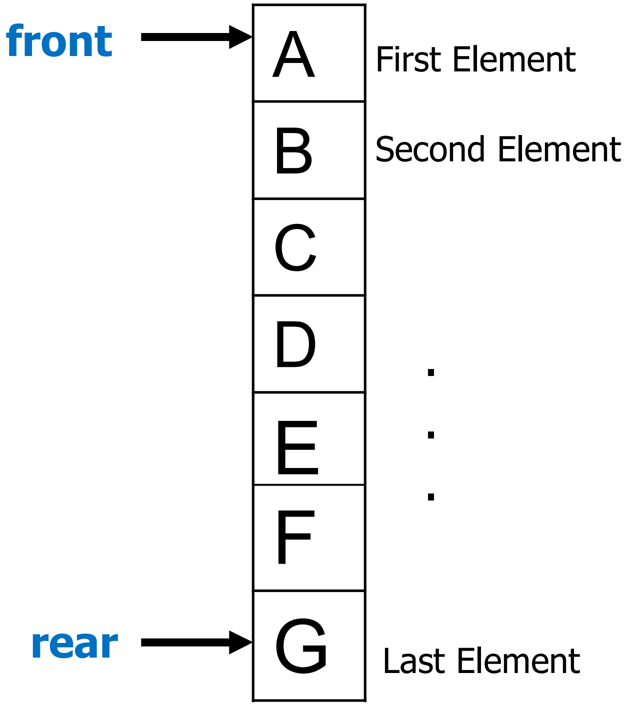
Front = -1 Rear = -1

Implementation

- Static
  - Queue is implemented by an array
  - Size of queue remains fix
- Dynamic
  - A queue can be implemented as a linked list
  - Expand or shrink with each enqueue or dequeue operation

Array Implementation

- Use two counters that signify rear and front
- When queue is empty
  - Both front and rear are set to -1
- When there is only one value in the Queue,
  - Both rear and front have same index
- While enqueueing increment rear by 1
- While dequeueing, increment front by 1



Array Implementation Example (1)

0	1	2	3	4	5	6	7	8

front= -1
rear = -1

5								
0	1	2	3	4	5	6	7	8

front= 0
rear = 0

5	4							
0	1	2	3	4	5	6	7	8

front= 0
rear = 1

Array Implementation Example (2)

5	4	6	7	8	7	6		
0	1	2	3	4	5	6	7	8

front=0
rear=6

				8	7	6		
0	1	2	3	4	5	6	7	8

front=4
rear=6

					7	6	12	67
0	1	2	3	4	5	6	7	8

front=5
rear=8

Problem: How can we insert more elements?
Rear index can not move beyond the last element….

Using Circular Queue

- Allow rear to wrap around the array

```
if(rear == queueSize-1)
      rear = 0;
else
      rear++;
```

- Alternatively, use modular arithmetic

```
rear = (rear + 1) % queueSize;
```

Example

					7	6	12	67
0	1	2	3	4	5	6	7	8

front=5
rear=8

Enqueue 39

- `Rear = (Rear+1) mod queueSize = (8+1) mod 9 = 0`

39					7	6	12	67
0	1	2	3	4	5	6	7	8

front=5
rear=0

Problem: How to avoid overwriting an existing element?

How to Determine Empty and Full Queues?

- A counter indicating number of values/items in the queue
  - Covered in first array-based implementation

- Without using an additional counter (only relying on front and rear)
  - Covered in alternative array-based implementation

Array-based Implementation

Array Implementation – Code (1)

```
class IntQueue
{
   private:
        int *queueArray; // Pointer to array implemented as Queue
        int queueSize;   // Total size of the Queue
        int front;
        int rear;
        int numItems;    // Number of items currently in the Queue
   public:
        IntQueue(int);
        ~IntQueue(void);
        void enqueue(int);
        int dequeue(void);
        bool isEmpty(void);
        bool isFull(void);
        void makeNull(void);

};
```

Array Implementation – Code (2)

```
class IntQueue
{
   private:
        int *queueArray; // Pointer to array implemented as Queue
        int queueSize;   // Total size of the Queue
        int front;
        int rear;
        int numItems;    // Number of items currently in the Queue
   public:
        IntQueue(int);
        ~IntQueue(void);
        void enqueue(int);
        int dequeue(void);
        bool isEmpty(void);
        bool isFull(void);
        void makeNull(void);

};
```

Clears the queue by resetting the `front` and `rear` indices, and setting the `numItems` to 0.

Array Implementation – Code (3)

- Constructor

```
IntQueue::IntQueue(int s) //constructor
{
   queueArray = new int[s];
   queueSize = s;
   front = -1;
   rear = -1;
   numItems = 0;
}
```

- Destructor

```
IntQueue::~IntQueue(void) //destructor
{
   delete [] queueArray;
}
```

Array Implementation – Code (4)

- `isFull()` returns true if the queue is full and false otherwise

```
bool IntQueue::isFull(void)
{
    if (numItems < queueSize)
           return false;
    else
           return true;
}
```

- `makeNull()` resets `front` & `rear` indices and sets `numItems= 0`

```
void IntQueue::makeNull(void)
{
    front = - 1;
    rear = - 1;
    numItems = 0;
}
```

Array Implementation – Code (5)

- Function `enqueue` inserts the value in `num` at the end of the Queue

```
void IntQueue::enqueue(int num)
{
   if (isFull())
      cout << "The queue is full.\n";

   else {
      // Calculate the new rear position
      rear = (rear + 1) % queueSize;
      // Insert new item
      queueArray[rear] = num;
      // Update item count
      numItems++;
   }
}
```

Array Implementation – Code (6)

- Function `dequeue` removes and returns the value at the front of the Queue

```
int IntQueue::dequeue(void)
{
   int num = -1;
   if (isEmpty())
      cout << "The queue is empty.\n";
   else{
      // Move front
      front = (front + 1) % queueSize;
      // Retrieve the front item
      num = queueArray[front];
      // Update item count
      numItems--;
   }
   return num;
}
```

Using Queues

Output:

```
Enqueuing 5 items...
Now attempting to enqueue again...
The queue is full
The values in the queue were:
0
1
2
3
4
```

```
void main(void)
{
   IntQueue iQueue(5);
   cout << "Enqueuing 5 items...\n";
   // Enqueue 5 items.
   for (int x = 0; x < 5; x++)
      iQueue.enqueue(x);
   // Attempt to enqueue a 6th item.
   cout << "Now attempting to enqueue again...\n";
   iQueue.enqueue(5);
   // Deqeue and retrieve all items in the queue
   cout << "The values in the queue were:\n";
   while (!iQueue.isEmpty()){
      int value;
      value = iQueue.dequeue();
      cout << value << endl;
   }
}
```

Alternative Array-based Implementation

Alternative Implementation – Code (1)

```
class CQueue
{
   Private:
      int *queueArray; // Pointer to array implemented as Queue
      int queueSize;   // Total size of the Queue
      int front;
      int rear;
   public:
      CQueue(int size);
      ~CQueue( );
      bool IsFull();
      bool IsEmpty();
      void enqueue(int num);
      int dequeue();
      void MakeNull();
};
```

Alternative Implementation – Code (2)

- `isEmpty()` returns true if the queue is empty and false otherwise

```
bool CQueue::IsEmpty()
{
   if (front==-1)
      return true; // we can check “rear” too
   else
      return false;
}
```

- `isFull()` returns true if the queue is full and false otherwise

```
bool CQueue::IsFull()
{
   if ( ( (rear+1)%queueSize ) == front )
      return true;
   else
      return false;
}
```

Alternative Implementation – Code (3)

- Function `enqueue` inserts the value in `num` at the end of the Queue

```
void CQueue ::enqueue(int num);
{
   if ( IsFull() ) {
      cout<<“Overflow”;
      return;
   }
   if (IsEmpty())
      rear = front = 0;
   else
      rear=(rear+1) % queueSize;
   queueArray[rear] = num;
}
```

Comparison: dequeue Operation

```
void CQueue ::enqueue(int num);
{
   if ( IsFull() ) {
      cout<<“Overflow”;
      return;
   }
   if (IsEmpty())
      rear = front = 0;
   else
      rear=(rear+1) % queueSize;
   queueArray[rear] = num;
}
```

```
void IntQueue::enqueue(int num)
{
   if (isFull())
      cout << "The queue is full.\n";

   else {
      // Calculate the new rear position
      rear = (rear + 1) % queueSize;
      // Insert new item
      queueArray[rear] = num;
      // Update item count
      numItems++;
   }
}
```

Alternative Implementation – Code (4)

- Function `dequeue` removes and returns the value at the front of the Queue

```
int CQueue ::dequeue()
{
   if ( IsEmpty() ) {
      cout<<“Underflow”;
      return;
   }
   int ReturnValue = queueArray[front];

   if ( front == rear ) //only one element in the queue
      front = rear = -1;
   else
      front = (front+1) % queueSize;

   return ReturnValue;
}
```

Any Question So Far?