CSE225: Data Structure and Algorithms

Lec 12: QUEUE

Queue

- A list
- Data items can be **added** and **deleted**
- Maintains **First In First Out (FIFO)** order

Specification of `QueueType`

Structure:	**Elements are added to the rear and removed from the front of the queue.**
Definitions (provided by user):	
MAX_ITEMS	Maximum number of items that might be on the queue.
ItemType	Data type of the items on the queue.
Operations (provided by the ADT):	
MakeEmpty	
Function	Sets queue to an empty state.
Postcondition	Queue is empty.
Boolean IsEmpty	
Function	Determines whether the queue is empty.
Precondition	Queue has been initialized.
Postcondition	Returns true if queue is empty and false otherwise.
Boolean IsFull	
Function	Determines whether the queue is full.
Precondition	Queue has been initialized.
Postcondition	Returns true if queue is full and false otherwise.

Specification of `QueueType`

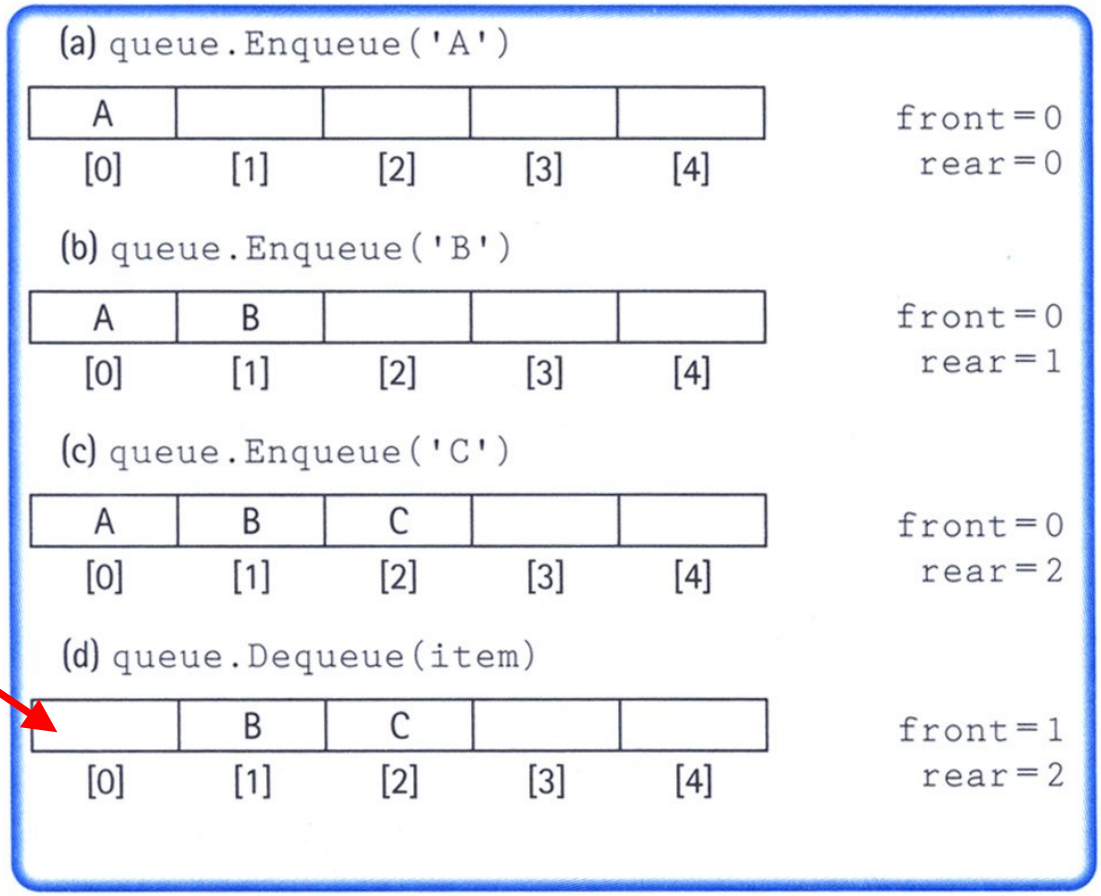
Enqueue(ItemType newItem)	
Function	Adds newItem to the rear of the queue.
Precondition	Queue has been initialized.
Postcondition	If (queue is full), FullQueue exception is thrown, else newItem is at rear of queue.
Dequeue(ItemType& item)	
Function	Removes front item from the queue and returns it in item.
Precondition	Queue has been initialized.
Postcondition	If (queue is empty), EmptyQueue exception is thrown and item is undefined, else front element has been removed from queue and item is a copy of removed element.

Implementation Issues

- Always insert elements at the back of the array.
- Complexity of deletion: **O(N)**

Implementation Issues

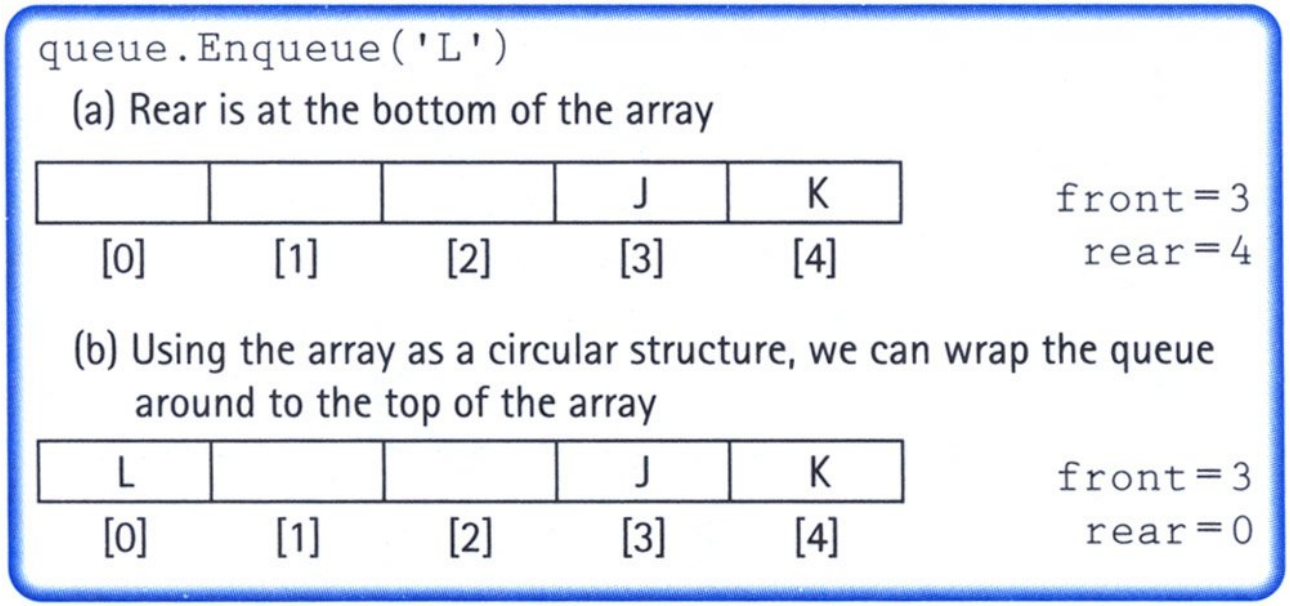
- Maintain two indices: **`front`** and **`rear`**
- Increment the indices as additions and deletions are performed (**`rear++`** for addition and **`front++`** for deletion)



Implementation Issues

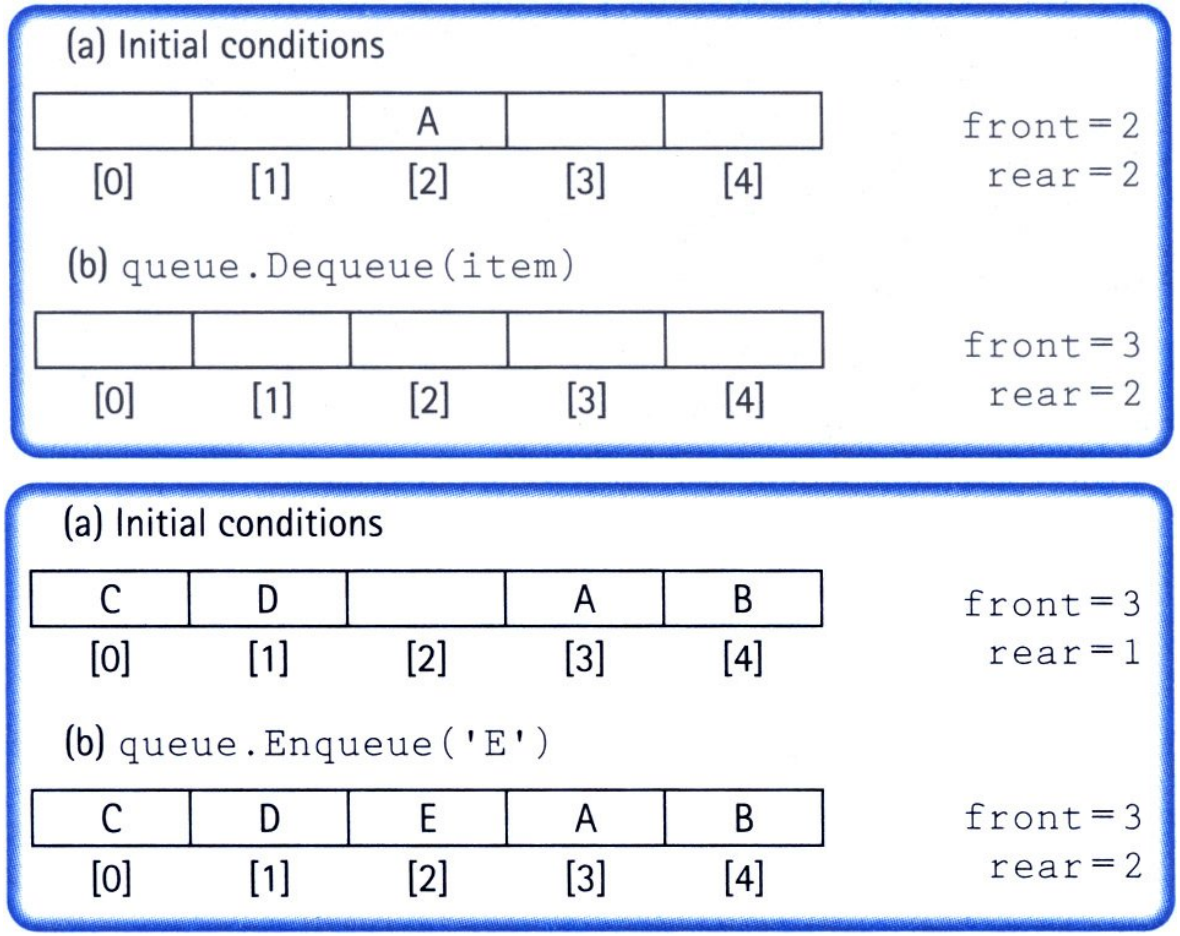
- Maintain two indices: **`front`** and **`rear`**
- Make the indices "wrap around" when they reach the end of the array

```
rear = (rear + 1) % maxQue; //maxQue=size of array
front = (front + 1) % maxQue;
```



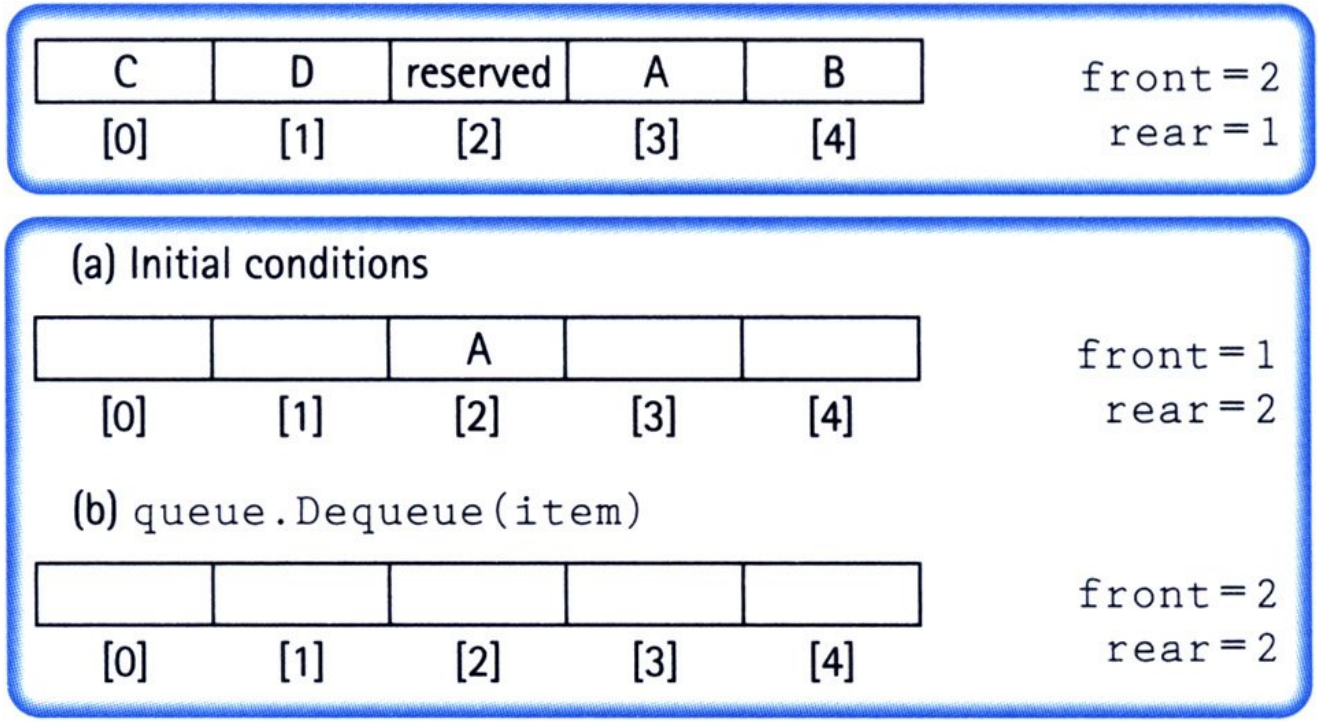
Implementation Issues

- How do we differentiate between the empty state and the full state?



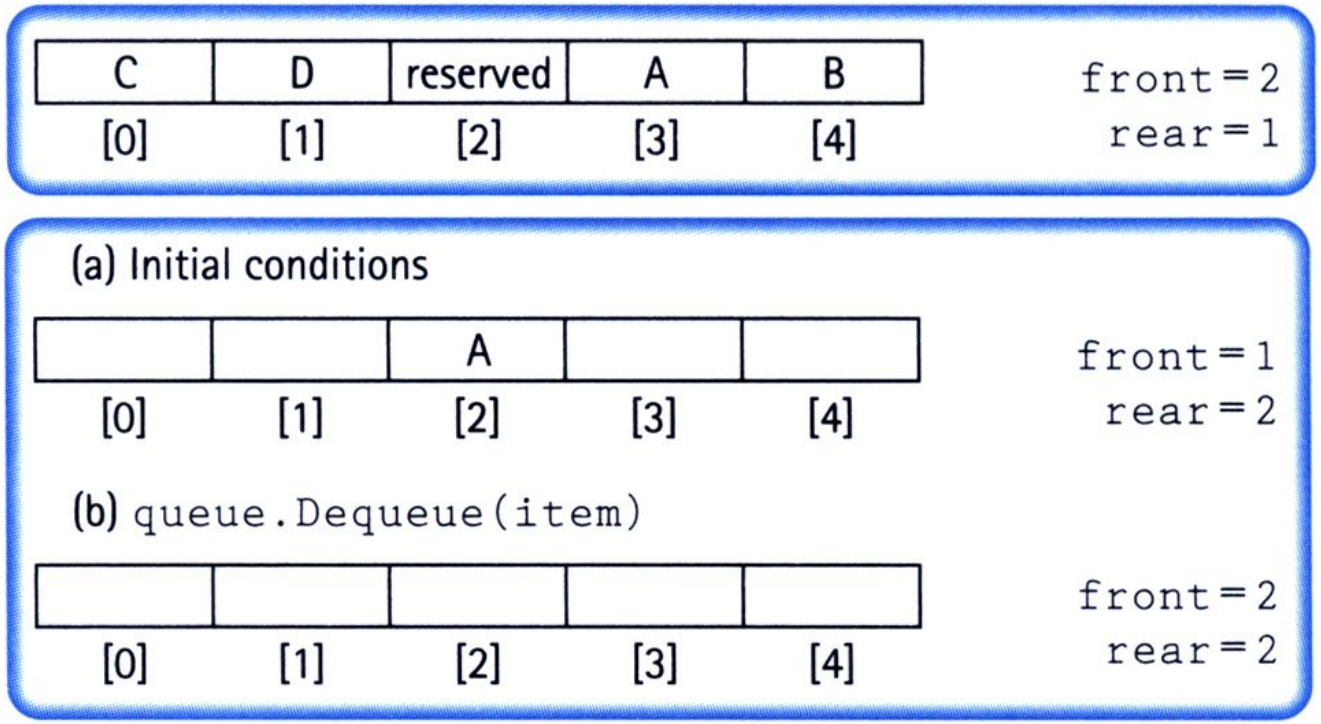
Implementation Issues

- Let front indicate the index of the array slot preceding the front element.
- The array slot preceding the front element is reserved and items are not assigned in that slot.



Implementation Issues

- Full queue when **`(rear + 1) % maxQue == front`**
- Empty queue when **`front == rear`**



queuetype.h

```
typedef char ItemType;
class FullQueue
{};
class EmptyQueue
{};
class QueType
{
public:
    QueType();
    QueType(int max);
    ~QueType();
    void MakeEmpty();
    bool IsEmpty();
    bool IsFull();
    void Enqueue(ItemType newItem);
    void Dequeue(ItemType& item);
private:
    int front;
    int rear;
    ItemType* items;
    int maxQue;
};
```

queuetype.cpp

```
#include "QueType.h"

QueType::QueType(int max)
{
  maxQue = max + 1;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new
ItemType[maxQue];
}

QueType::QueType()
{
  maxQue = 501;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new
ItemType[maxQue];
}

QueType::~QueType()
{
  delete [] items;
}
```

```
void QueType::MakeEmpty()
{
  front = maxQue - 1;
  rear = maxQue - 1;
}

bool QueType::IsEmpty()
{
  return (rear == front);
}

bool QueType::IsFull()
{
  return ((rear+1)%maxQue == front);
}
```

queuetype.cpp

```
void QueType::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    rear = (rear +1) % maxQue;
    items[rear] = newItem;
  }
}

void QueType::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    front = (front + 1) % maxQue;
    item = items[front];
  }
}
```

queuetype.cpp

```
#include "QueType.h"

QueType::QueType(int max)
{
  maxQue = max + 1;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new
ItemType[maxQue];
}
```

O(1)

```
QueType::QueType()
{
  maxQue = 501;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new
ItemType[maxQue];
}
```

O(1)

```
QueType::~QueType()
{
  delete [] items;
}
```

O(1)

```
void QueType::MakeEmpty()
{
  front = maxQue - 1;
  rear = maxQue - 1;
}
```

O(1)

```
bool QueType::IsEmpty()
{
  return (rear == front);
}
```

O(1)

```
bool QueType::IsFull()
{
  return ((rear+1)%maxQue == front);
}
```

O(1)

queuetype.cpp

```
void QueType::Enqueue(ItemType newItem)
{
  if (IsFull())
    throw FullQueue();
  else
  {
    rear = (rear +1) % maxQue;
    items[rear] = newItem;
  }
}
```

O(1)

```
void QueType::Dequeue(ItemType& item)
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    front = (front + 1) % maxQue;
    item = items[front];
  }
}
```

O(1)

- **Applications of Queue Data Structure**

1. Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.
2. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
3. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.)