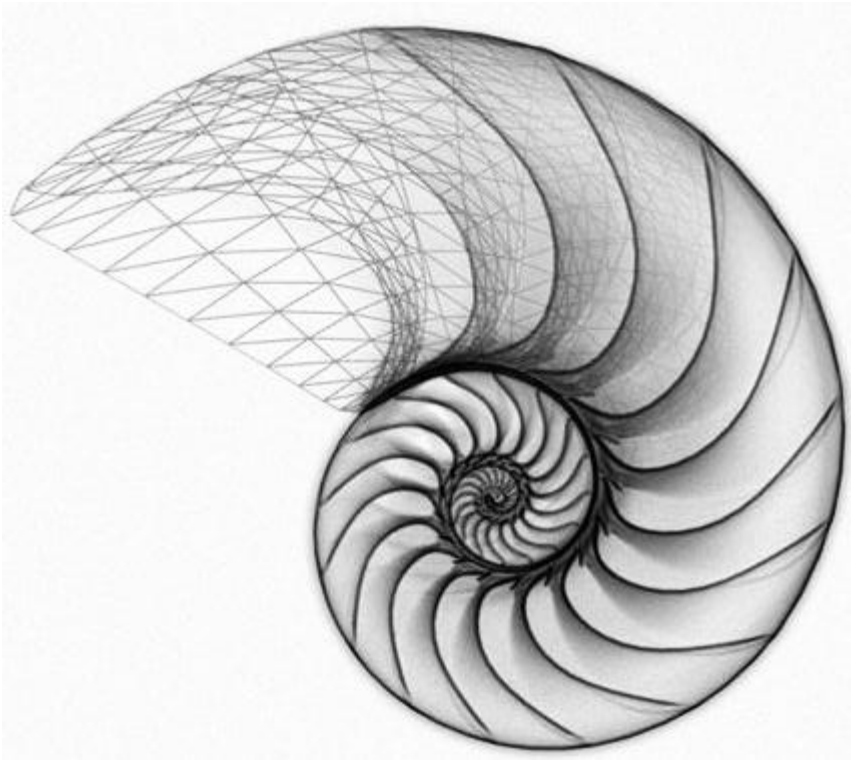


Lecture 16

Abstract Data Type Stack & Queue (Array-based Implementation)

Specification of `StackType`

Structure: Elements are added to and removed from the top of the stack.

Definitions (provided by user):

MAX_ITEMS	Maximum number of items that might be on the stack.
ItemType	Data type of the items on the stack.

Operations (provided by the ADT):

MakeEmpty

Function	Sets stack to an empty state.
Postcondition	Stack is empty.

Boolean IsEmpty

Function	Determines whether the stack is empty.
Precondition	Stack has been initialized.
Postcondition	Returns true if stack is empty and false otherwise.

Boolean IsFull

Function	Determines whether the stack is full.
Precondition	Stack has been initialized.
Postcondition	Returns true if stack is full and false otherwise.

Specification of `StackType`

Push(ItemType newItem)

Function	Adds newItem to the top of the stack.
Precondition	Stack has been initialized.
Postcondition	If (stack is full), exception FullStack is thrown, else newItem is at the top of the stack.

Pop()

Function	Removes top item from the stack.
Precondition	Stack has been initialized.
Postcondition	If (stack is empty), exception EmptyStack is thrown, else top element has been removed from stack.

ItemType Top()

Function	Returns a copy of the top item on the stack.
Precondition	Stack has been initialized.
Postcondition	If (stack is empty), exception EmptyStack is thrown, else a copy of the top element is returned.

stacktype.h

```
#ifndef STACKTYPE_H_INCLUDED
#define STACKTYPE_H_INCLUDED

const int MAX_ITEMS = 5;

class FullStack
{};  // Exception class thrown by Push when stack is full.
class EmptyStack
{};  // Exception class thrown by Pop and Top when stack is empty.

template <class ItemType>
class StackType
{
    public:
        StackType();
        bool IsFull();
        bool IsEmpty();
        void MakeEmpty();
        void Push(ItemType);
        void Pop();
        ItemType Top();
    private:
        int top;
        ItemType  items[MAX_ITEMS];
};
#endif // STACKTYPE_H_INCLUDED
```

stacktype.cpp

```
#include "StackType.h"
template <class ItemType>

StackType<ItemType>::StackType()
{
    top = -1;
}
template <class ItemType>
bool StackType<ItemType>::IsEmpty()
{
    return (top == -1);
}
void StackType<ItemType>::MakeEmpty()
{
    top = -1;
}
template <class ItemType>
bool StackType<ItemType>::IsFull()
{
    return (top ==  MAX_ITEMS-1);
}
```

```
template <class ItemType>
void
StackType<ItemType>::Push(ItemType
newItem)
{
    if( IsFull() )
        throw FullStack();
    top++;
    items[top] = newItem;
}
template <class ItemType>
void StackType<ItemType>::Pop()
{
    if( IsEmpty() )
        throw EmptyStack();
    top--;
}
template <class ItemType>
ItemType StackType<ItemType>::Top()
{
    if (IsEmpty())
        throw EmptyStack();
    return items[top];
}
```

stacktype.cpp

```
#include "StackType.h"
template <class ItemType>

StackType<ItemType>::StackType()
{
    top = -1;
}
```

O(1)

```
template <class ItemType>
bool StackType<ItemType>::IsEmpty()
{
    return (top == -1);
}
```

O(1)

```
void StackType<ItemType>::MakeEmpty()
{
    top = -1;
}
```

O(1)

```
template <class ItemType>
bool StackType<ItemType>::IsFull()
{
    return (top ==  MAX_ITEMS-1);
}
```

O(1)

```
template <class ItemType>
void
StackType<ItemType>::Push(ItemType
newItem)
{
    if( IsFull() )
        throw FullStack();
    top++;
    items[top] = newItem;
}
```

O(1)

```
template <class ItemType>
void StackType<ItemType>::Pop()
{
    if( IsEmpty() )
        throw EmptyStack();
    top--;
}
```

O(1)

```
template <class ItemType>
ItemType StackType<ItemType>::Top()
{
    if (IsEmpty())
        throw EmptyStack();
    return items[top];
}
```

O(1)

Application of Stack

```
Pattern Matching Algorithm:
(())()(()())()
```

Application of Stack

Algorithm for matching parentheses string

1. Initialise an empty stack
2. Read next item in the string
   a) If item is an opening parentheses, push it into the stack
   b) Else, if item is a closing parentheses, pop from stack
3. If there are more items to process, go to step 2
4. Pop the answer off the stack.

(( )) ( ) (( ) ( )) ( )

Application of Stack

Infix to Postfix Conversion

`(4+8)*(6-5)/((3-2)*(2+2))`

Application of Stack

Evaluating a postfix expression

Application of Stack

Algorithm for evaluating a postfix expression

1. Initialise an empty stack
2. Read next item in the expression
   a) If item is an operand, push it into the stack
   b) Else, if item is an operator, pop top two items off the stack, apply the operator, and push the answer back into the stack
3. If there are more items to process, go to step 2
4. Pop the answer off the stack.

Application of Stack

Now let’s evaluate this expression.

4 8 + 6 5 - * 3 2 - 2 2 + * /

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 queueto an empty state.
Postcondition	Queue is empty.

Boolean IsEmpty

Function	Determines whether the queueis 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)

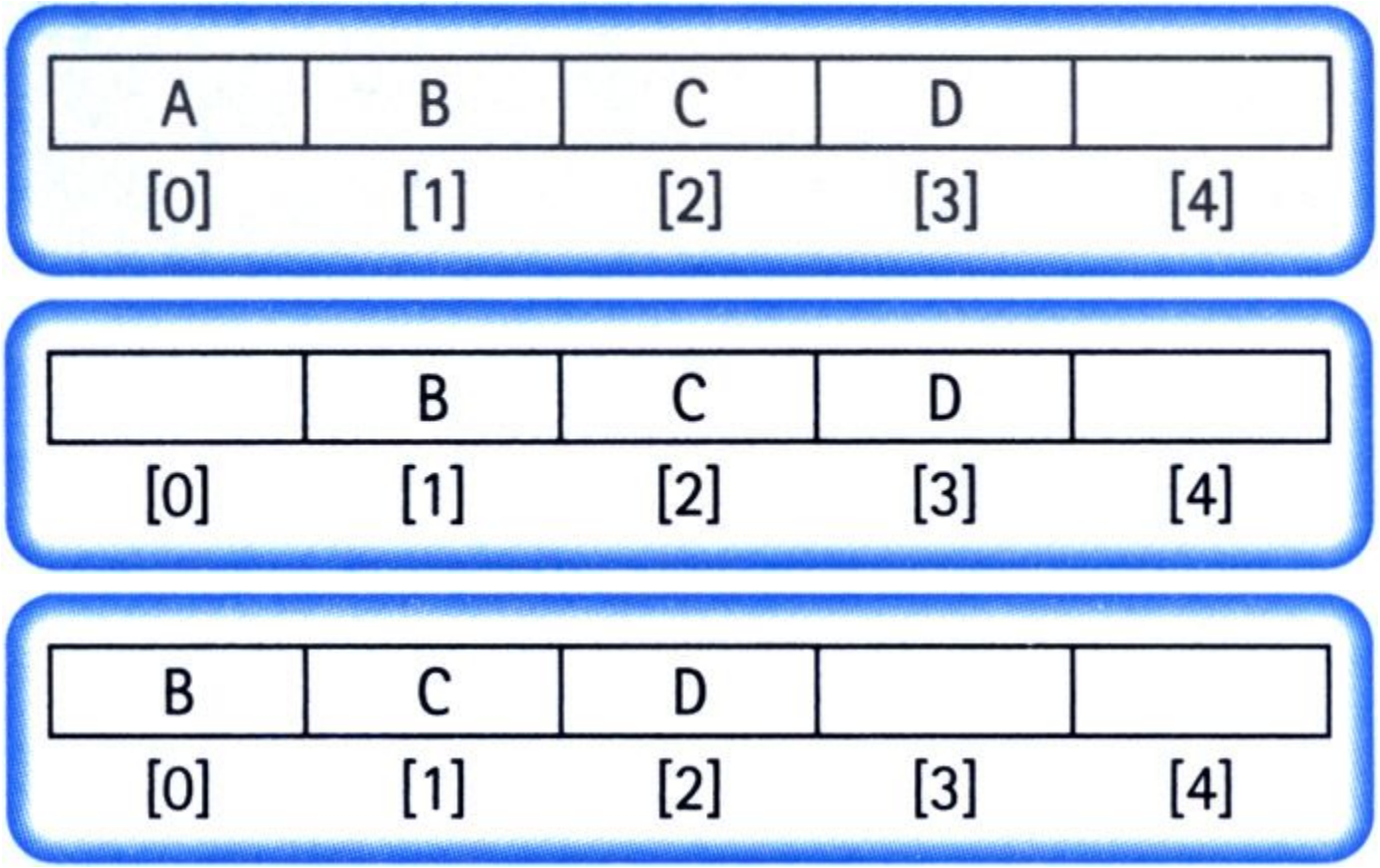
Function	Adds newItem to the rear of the queue.
Precondition	Queue has been initialized.
Postcondition	If (queue is full), FullQueue exception is thrown, else newItem is at rear of queue.

Dequeue(ItemType& item)

Function	Removes front item from the queue and returns it as 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.

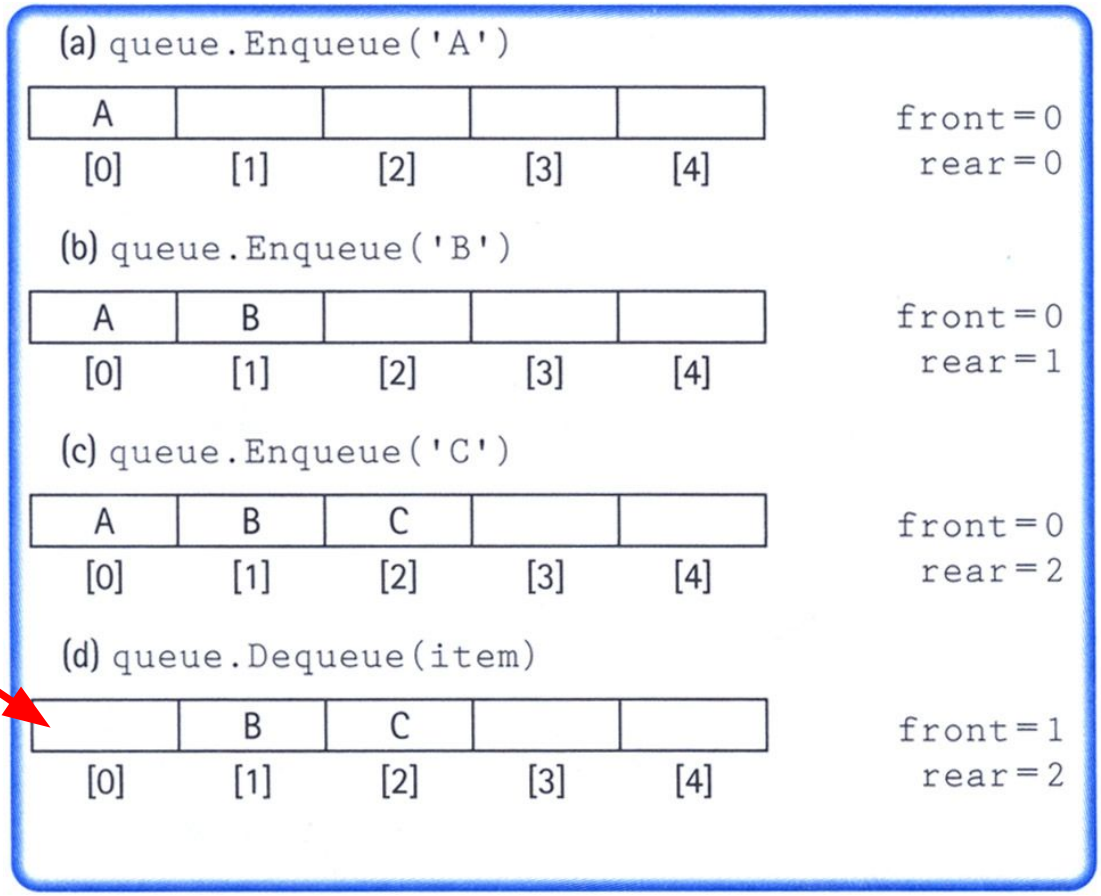
Implementation Issues

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



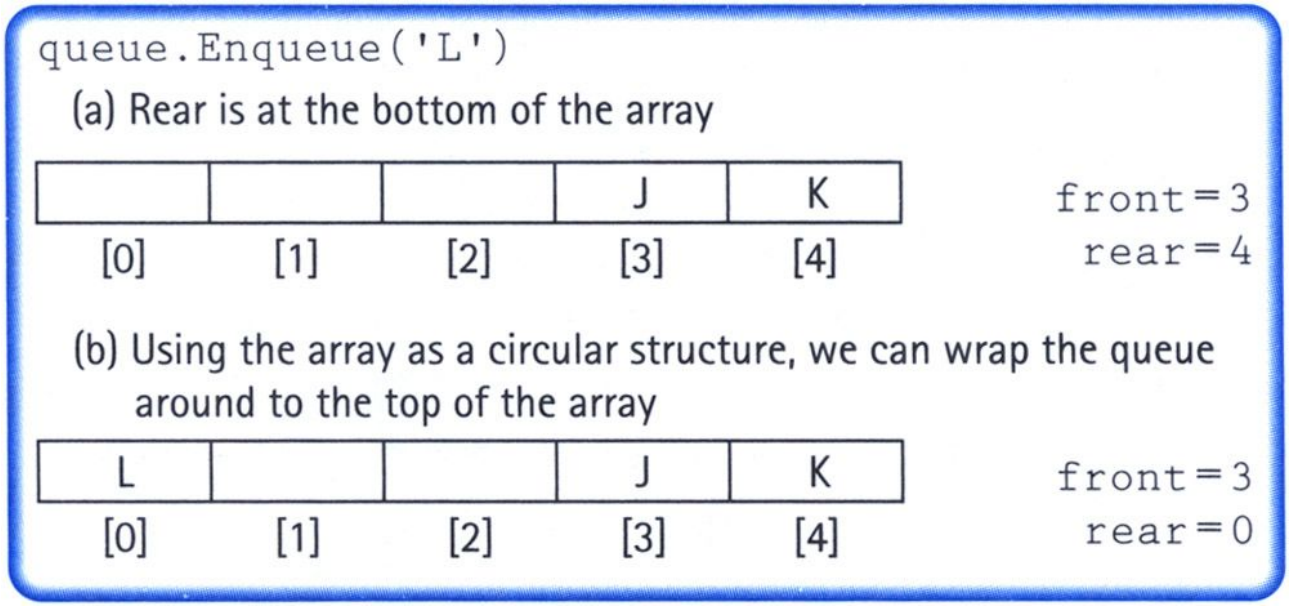
Implementation Issues

- Maintain two indices: **`front`** and **`rear`**
- Increment the indices as enqueue and dequeue are performed (**`rear++`** for enqueue and **`front++`** for dequeue)



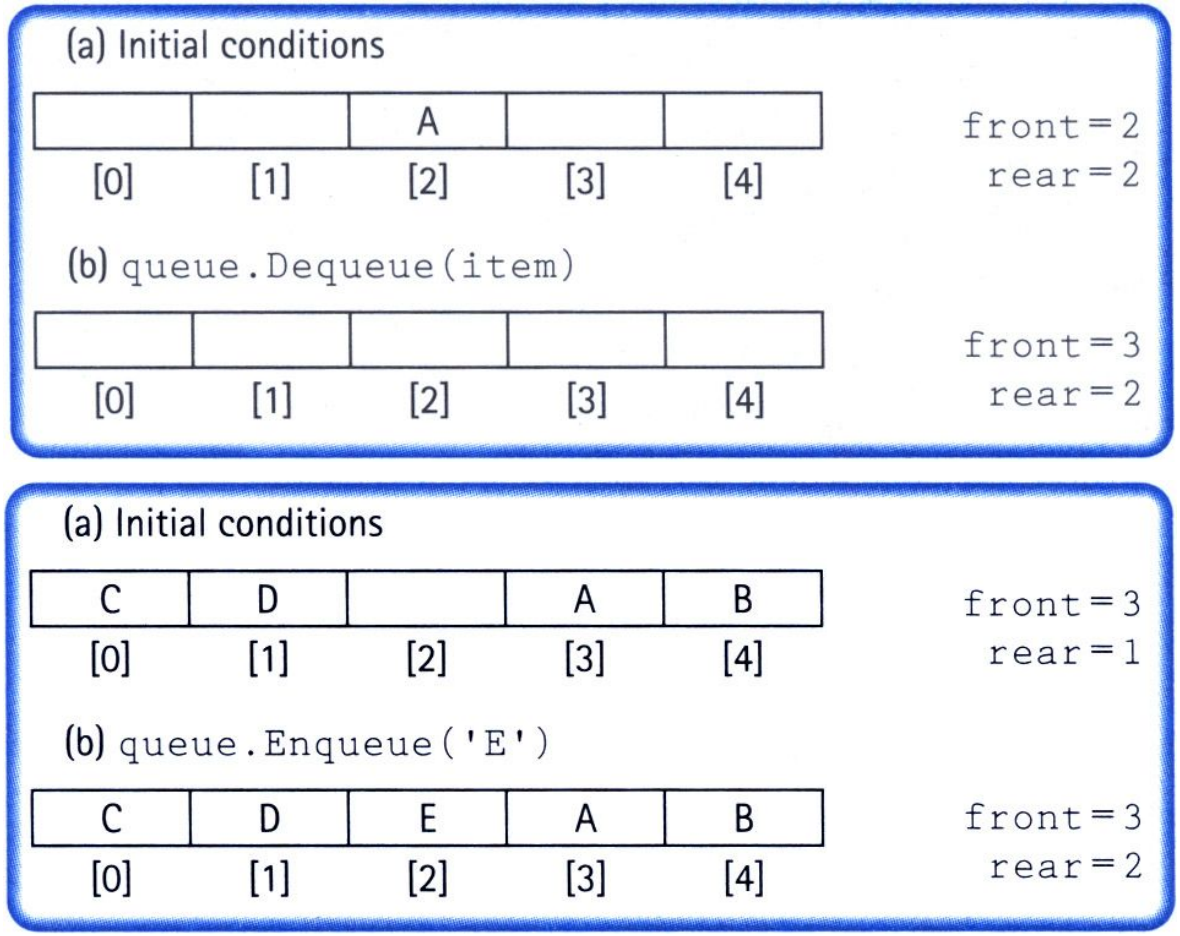
Implementation Issues

- Maintain two indices: **`front`** and **`rear`**
- Make the indices "wrap around" when they reach the end of the array:
- For Dequeue: **`rear = (rear + 1) % maxQue;`**`//maxQue=array size`
- For Enqueue: **`front = (front + 1) % maxQue;`**



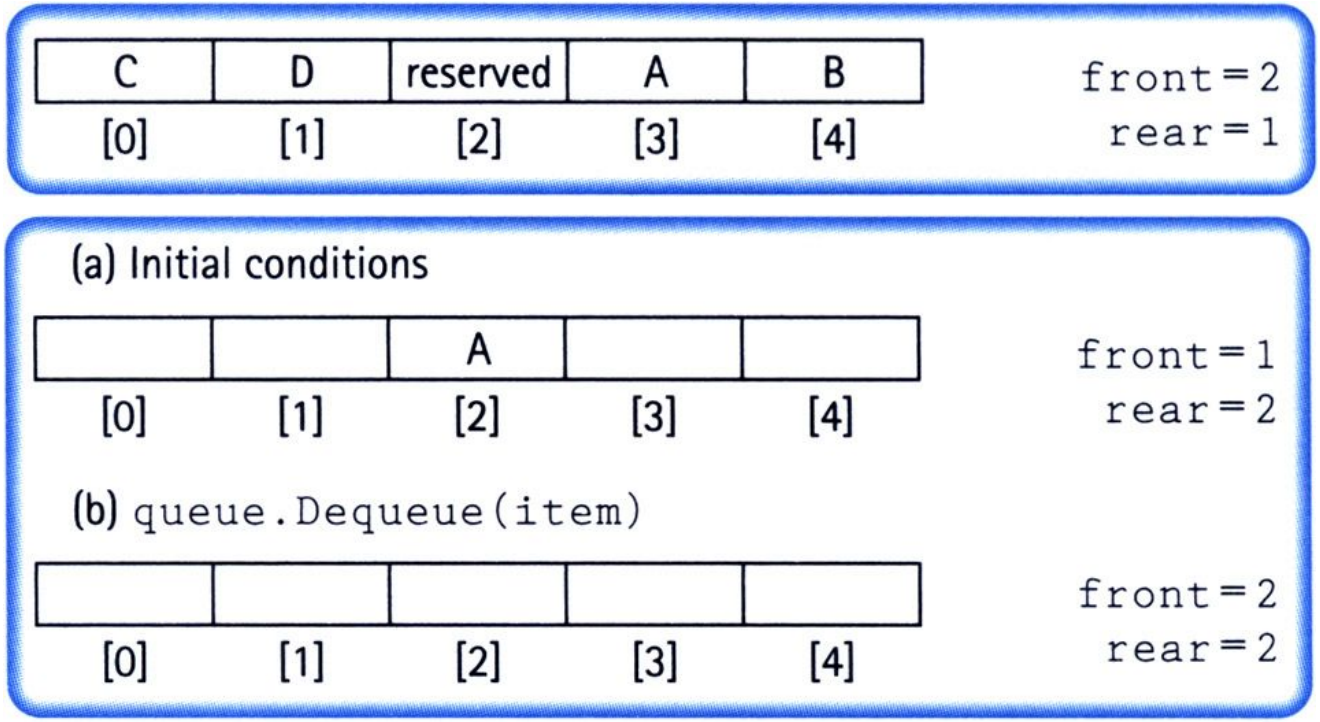
Implementation Issues

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



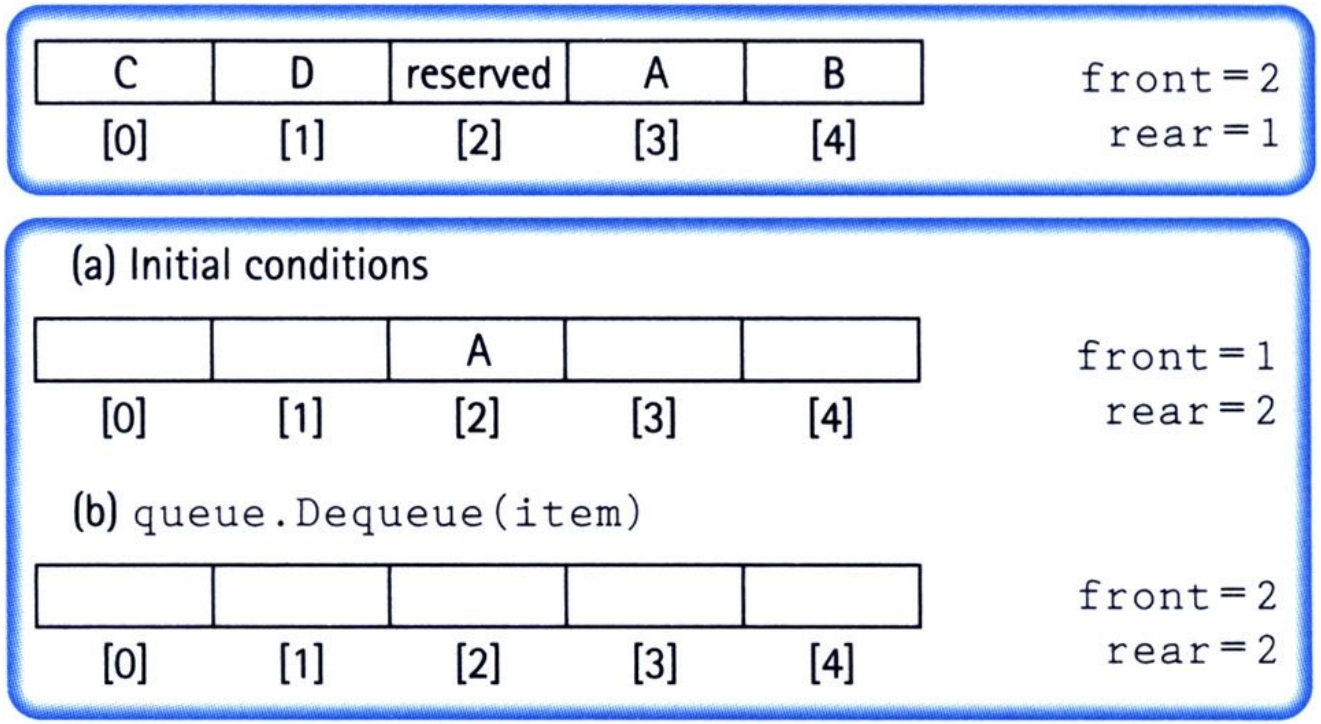
Implementation Issues

- 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

```
#define DEFAULT_SIZE 500
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 = DEFAULT_SIZE + 1;
  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];
}

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

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

O(1)

O(1)

O(1)

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

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

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

O(1)

O(1)

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)