CS302 - Data Structures

using C++

Topic: Queues and Priority Queues

Kostas Alexis

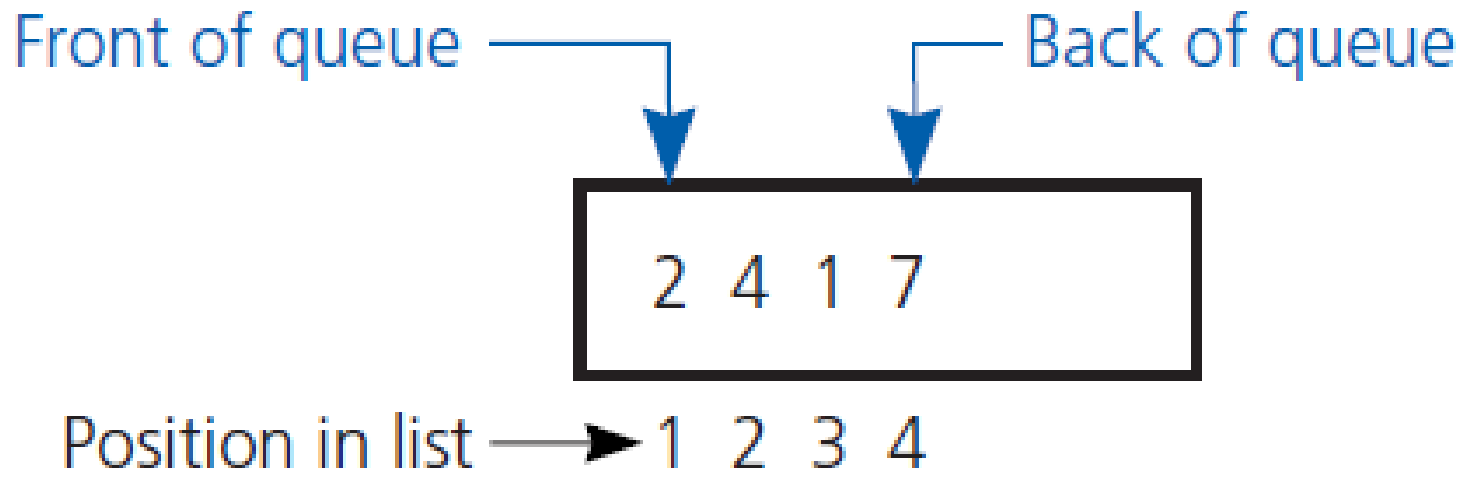
AUTONOMOUS ROBOTS LAB

Implementations of the ADT Queue

- Like stacks, queues can have
  - Array-based or
  - Link-based implementation
- Can also use implementation of ADT list
  - Efficient to implement
  - Might not be most time efficient as possible

An Implementation that uses the ADT list

- An implementation of the ADT queue that stores its entries in a list



An Implementation that uses the ADT list

- Header file for the class ListQueue

```
#ifndef LIST_QUEUE_
#define LIST_QUEUE_

#include "QueueInterface.h"
#include "LinkedList.h"
#include "PrecondViolatedExcept.h"
#include <memory>

template<class ItemType>
class ListQueue : public QueueInterface<ItemType>
{
private:
      std::unique_ptr<LinkedList<ItemType>> listPtr;   // Pointer to list of queue items

public:
      ListQueue();
      ListQueue(const ListQueue& aQueue);
      ~ListQueue();
      bool isEmpty() const;
      bool enqueue(const ItemType& newEntry);
      bool dequeue();

      // @throw PrecondViolatedExcept if this queue is empty
      ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end ListQueue
#include "ListQueue.cpp"
#endif
```

An Implementation that uses the ADT list

- The implementation file for the class ListQue

```
#include "ListQueue.h" // Header file
#include <memory>

template<class ItemType>
ListQueue<ItemType>::ListQueue() : listPtr(std::make_unique<LinkedList<ItemType>>())
{
} // end default constructor

template<class ItemType>
ListQueue<ItemType>::ListQueue(const ListQueue& aQueue) : listPtr(aQueue.listPtr)
{
} // end copy constructor

template<class ItemType>
ListQueue<ItemType>::~ListQueue()
{
} // end destructor

template<class ItemType>
ListQueue<ItemType>::isEmpty();
{
      return listPtr->isEmpty();
} // end isEmpty

template<class ItemType>
ListQueue<ItemType>::enqueue(const ItemType& newEntry)
{
      return listPtr->insert(listPtr->getLength() + 1, newEntry);
} // end enqueue
```

An Implementation that uses the ADT list

- The implementation file for the class ListQue

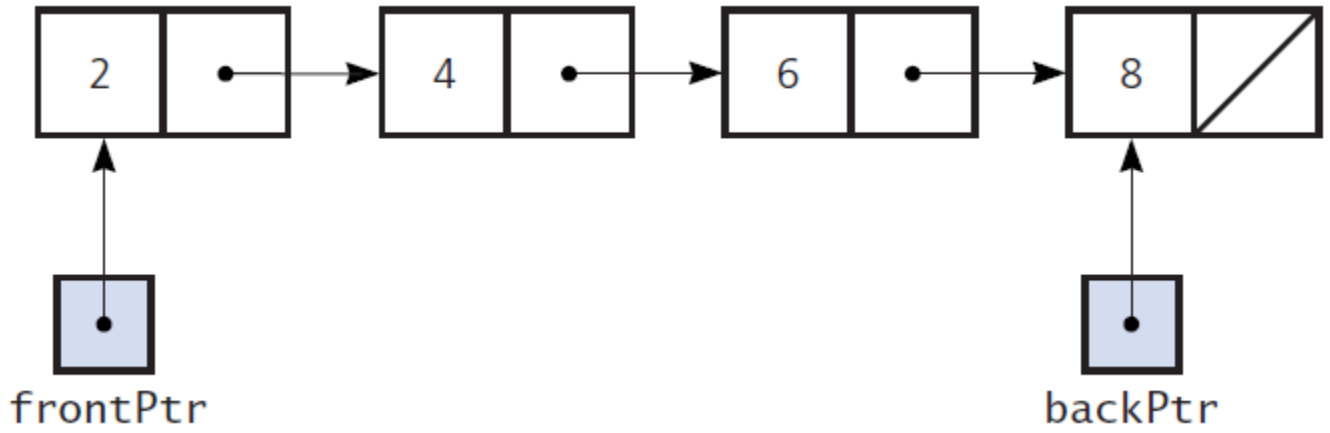
```
template<class ItemType>
ListQueue<ItemType>::dequeue()
{
      return listPtr->remove(1);
} // end dequeuer


template<class ItemType>
ListQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
{
      if (isEmpty())
            throw PrecondViolatedExcpt("peekFront() called with empty queue");

      return listPtr->getEntry(1);
} // end peekFront
// end of implementation file
```

A Link-based Implementation

- Similar to other link-based implementations
- One difference: must be able to remove entries
  - From front
  - From back
- Requires a pointer to chain's last node
  - Called the "tail pointer"

A Link-based Implementation

- A chain of linked nodes with head and tail pointers



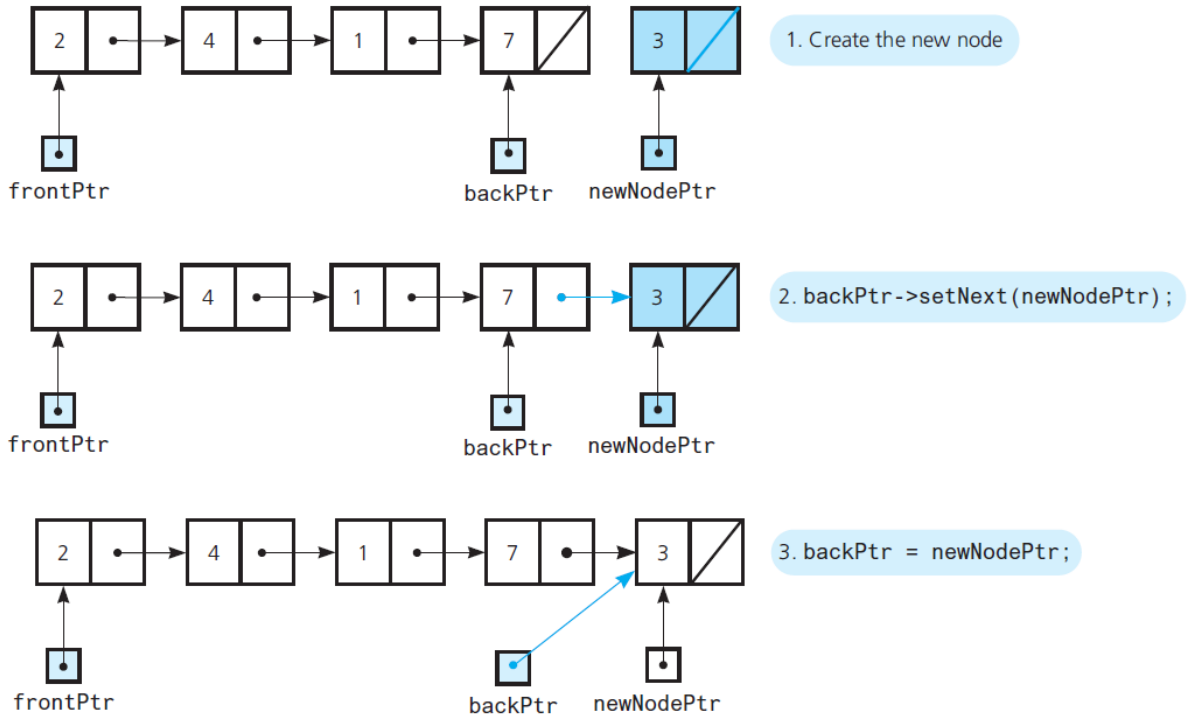
A Link-based Implementation

- The header file for the class LinkedQueue

```
#ifndef LINKED_QUEUE_
#define LINKED_QUEUE_

#include "QueueInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"
#include <memory>

template<class ItemType>
class LinkedQueue : public QueueInterface<ItemType>
{
private:
      // The queue is implemented as a chain of linked nodes that has two external pointers, a head pointer for the front of
      // the que and a tail pointer for the back of the queue
      std::shared_ptr<Node<ItemType>> frontPtr;
      std::shared_ptr<Node<ItemType>> backPtr;
public:
      LinkedQueue();
      LinkedQueue(const LinkedQueue& aQueue);
      ~LinkedQueue();

      bool isEmpty() const;
      bool enqueue(const ItemType& newEntry);
      bool dequeuer();
      // @throw PrecondViolatedExcept if the queue is empty
      ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end LinkedQueue
#include "LinkedQueue.cpp"
#endif
```

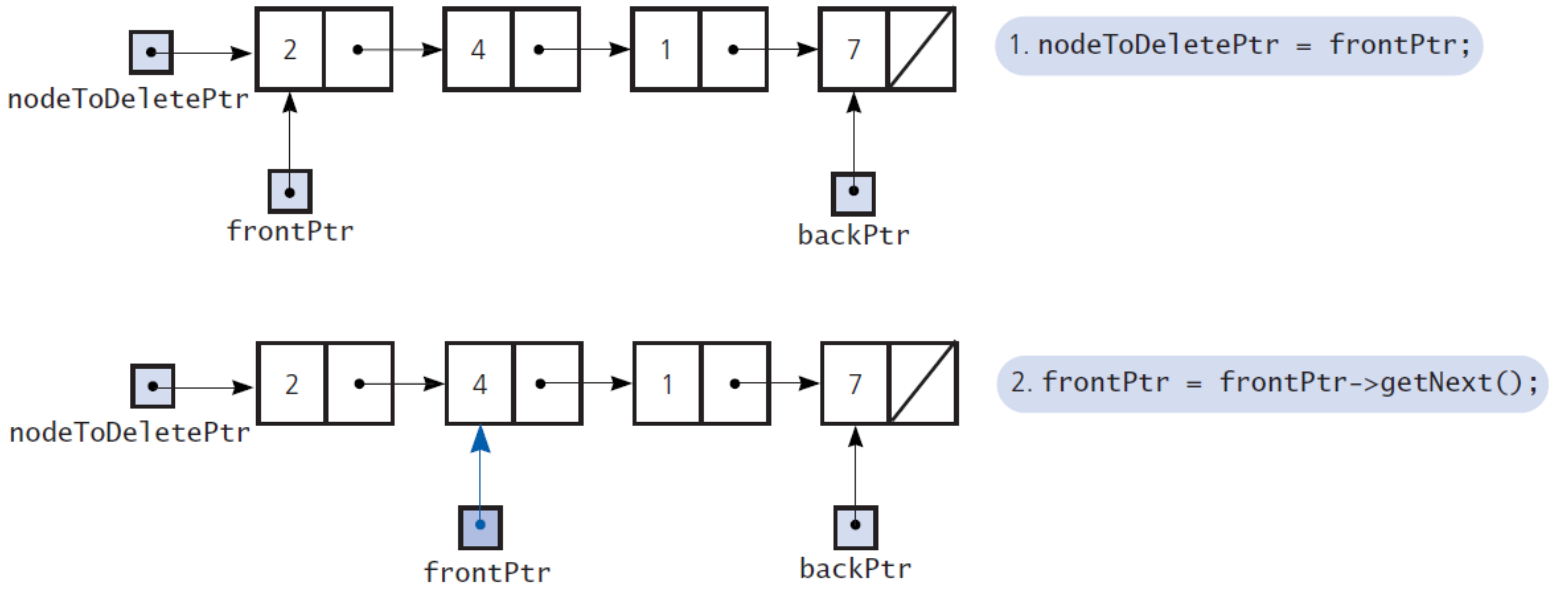
A Link-based Implementation

- Adding an item to a nonempty queue



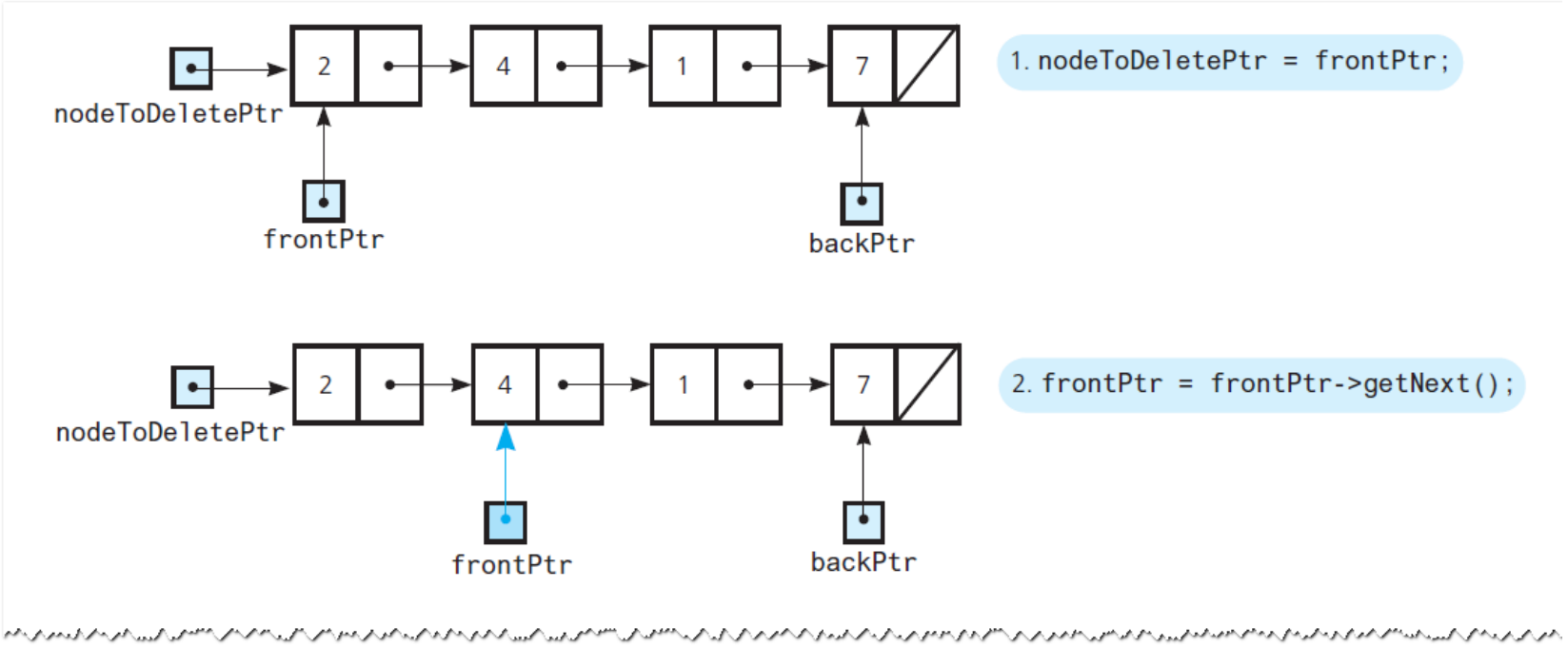
A Link-based Implementation

- Removing an item from a queue of more than one item



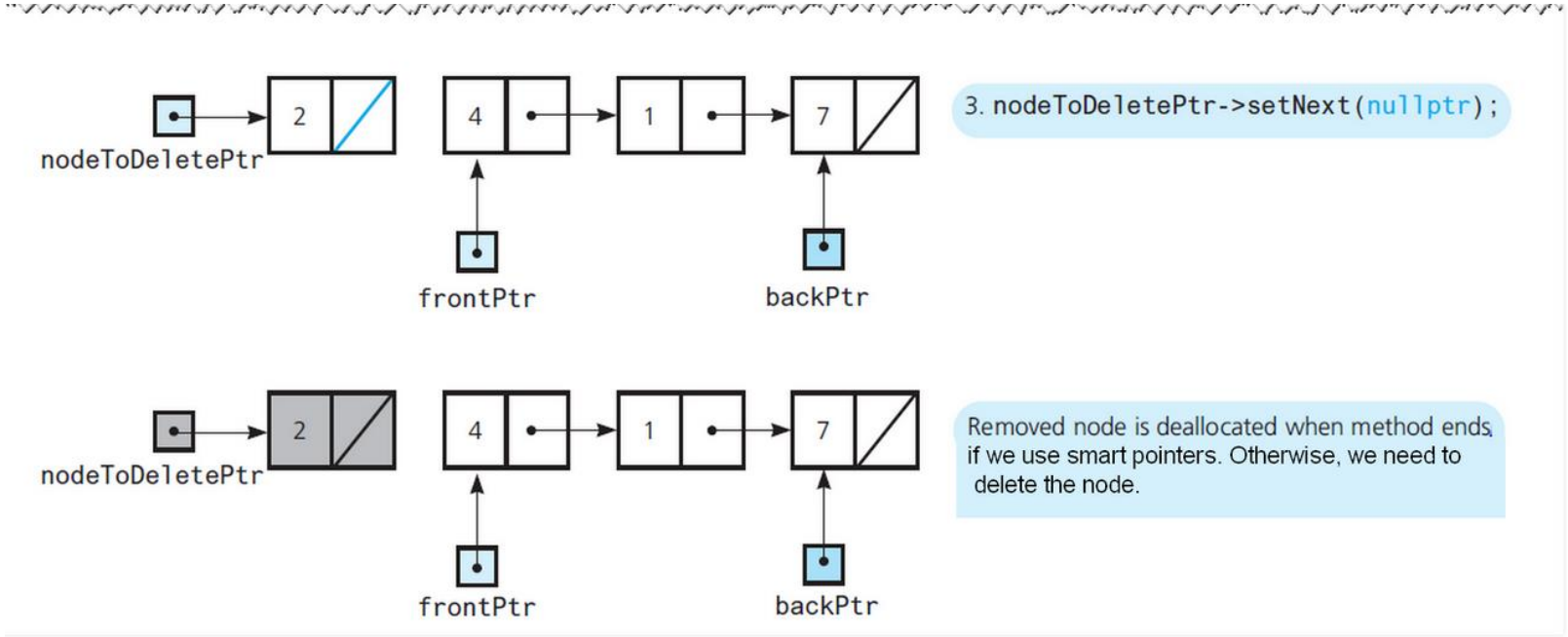
A Link-based Implementation

- Removing an item from a queue of more than one item (cont)



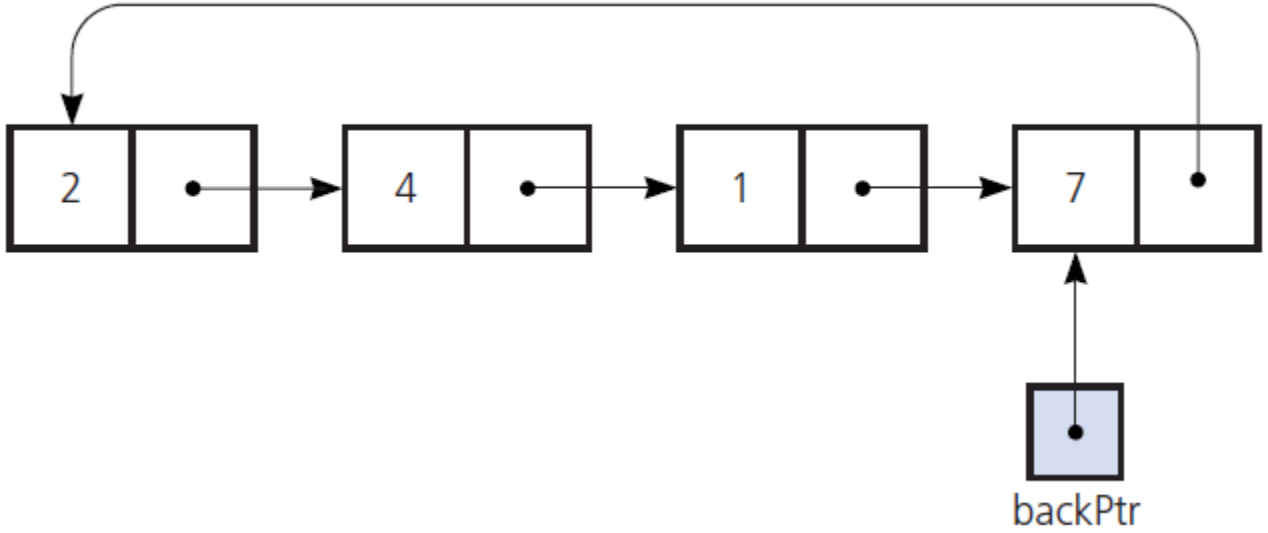
A Link-based Implementation

- Removing an item from a queue of more than one item (cont)



A Link-based Implementation

- **Note:** A circular chain of linked nodes with one external pointer
- One way to get it to work well with a single pointer



An Array-based Implementation

- If a fixed-size is no problem we may implement queues with arrays.

An Array-based Implementation

- If a fixed-size is no problem we may implement queues with arrays.
- At a minimum we need:

```
static const int DEFAULT_CAPACITY = some value
ItemType items[DEFAULT_CAPACITY]
int front;
int back;
```

An Array-based Implementation

- If a fixed-size is no problem we may implement queues with arrays.
- At a minimum we need:

```
static const int DEFAULT_CAPACITY = some value
ItemType items[DEFAULT_CAPACITY]
int front;
int back;
```

- Add item: increment back and place item in items[back]
- Remove item: increment front.

An Array-based Implementation

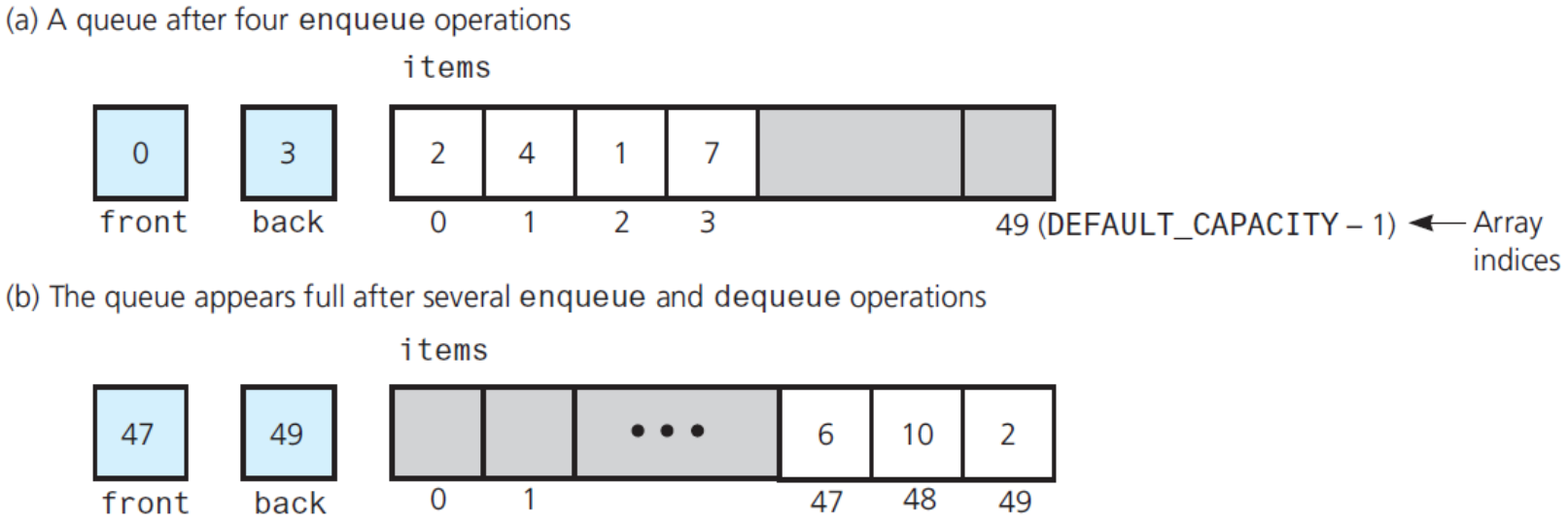
- If a fixed-size is no problem we may implement queues with arrays.
- At a minimum we need:

```
static const int DEFAULT_CAPACITY = some value
ItemType items[DEFAULT_CAPACITY]
int front;
int back;
```

- **Add item:** increment back and place item in items[back]
- **Remove item:** increment front.
- Problem: the queue is full when back equals DEFAULT_CAPACITY-1 and this may happen without the array being fully completed actually.

An Array-based Implementation

- A naïve array-based implementation of a queue for which rightward drift can cause the queue to appear full



An Array-based Implementation

- If a fixed-size is no problem we may implement queues with arrays.
- At a minimum we need:

```
static const int DEFAULT_CAPACITY = some value
ItemType items[DEFAULT_CAPACITY]
int front;
int back;
```

- **Add item:** increment back and place item in items[back]
- **Remove item:** increment front.
- Problem: the queue is full when back equals DEFAULT_CAPACITY-1 and this may happen without the array being fully completed actually.
- Possible Solution: Shift Elements

An Array-based Implementation

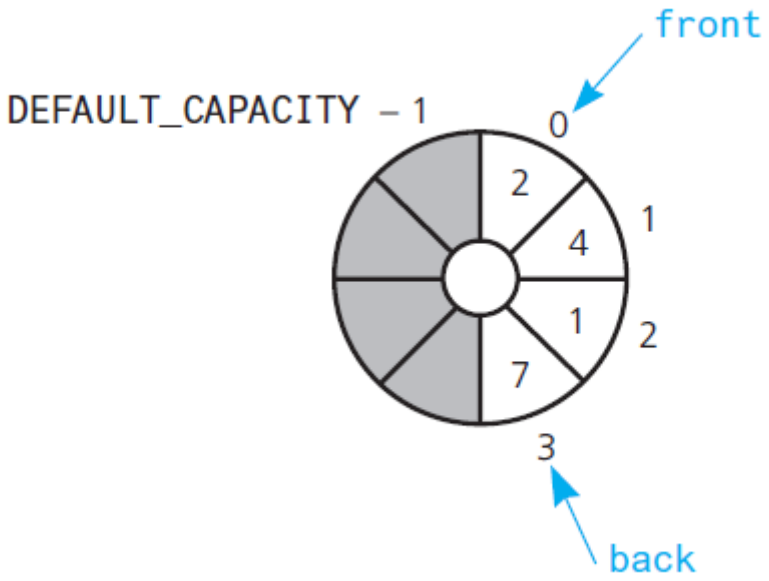
- If a fixed-size is no problem we may implement queues with arrays.
- At a minimum we need:

```
static const int DEFAULT_CAPACITY = some value
ItemType items[DEFAULT_CAPACITY]
int front;
int back;
```

- **Add item:** increment `back` and place item in `items[back]`
- **Remove item:** increment `front`.
- Problem: the queue is full when back equals DEFAULT_CAPACITY-1 and this may happen without the array being fully completed actually.
- Possible Solution: Shift Elements
- Alternative – Elegant Solution: Treat the array as circular!

An Array-based Implementation

- A circular array as an implementation of a queue



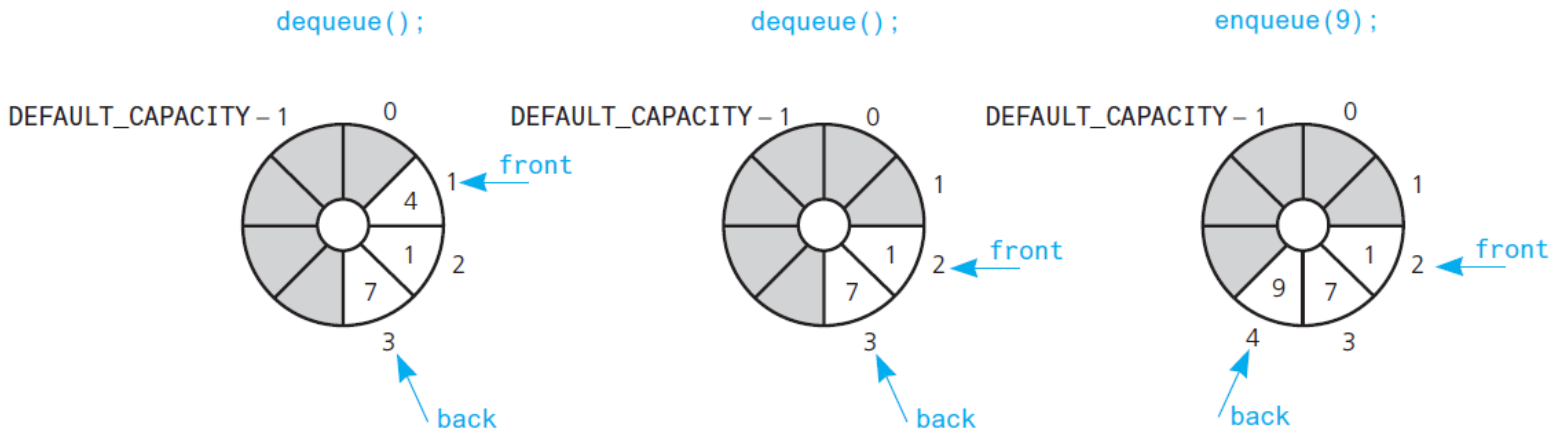
An Array-based Implementation

- In the circular array implementation concept
- **Remove item:** increment the queue index front.
- **Add item:** increment back.
- When either front or back advances past DEFAULT_CAPACITY-1, then wrap around to 0.

```
back = (back + 1) % DEFAULT_CAPACITY;
items[back] = newEntry;
```

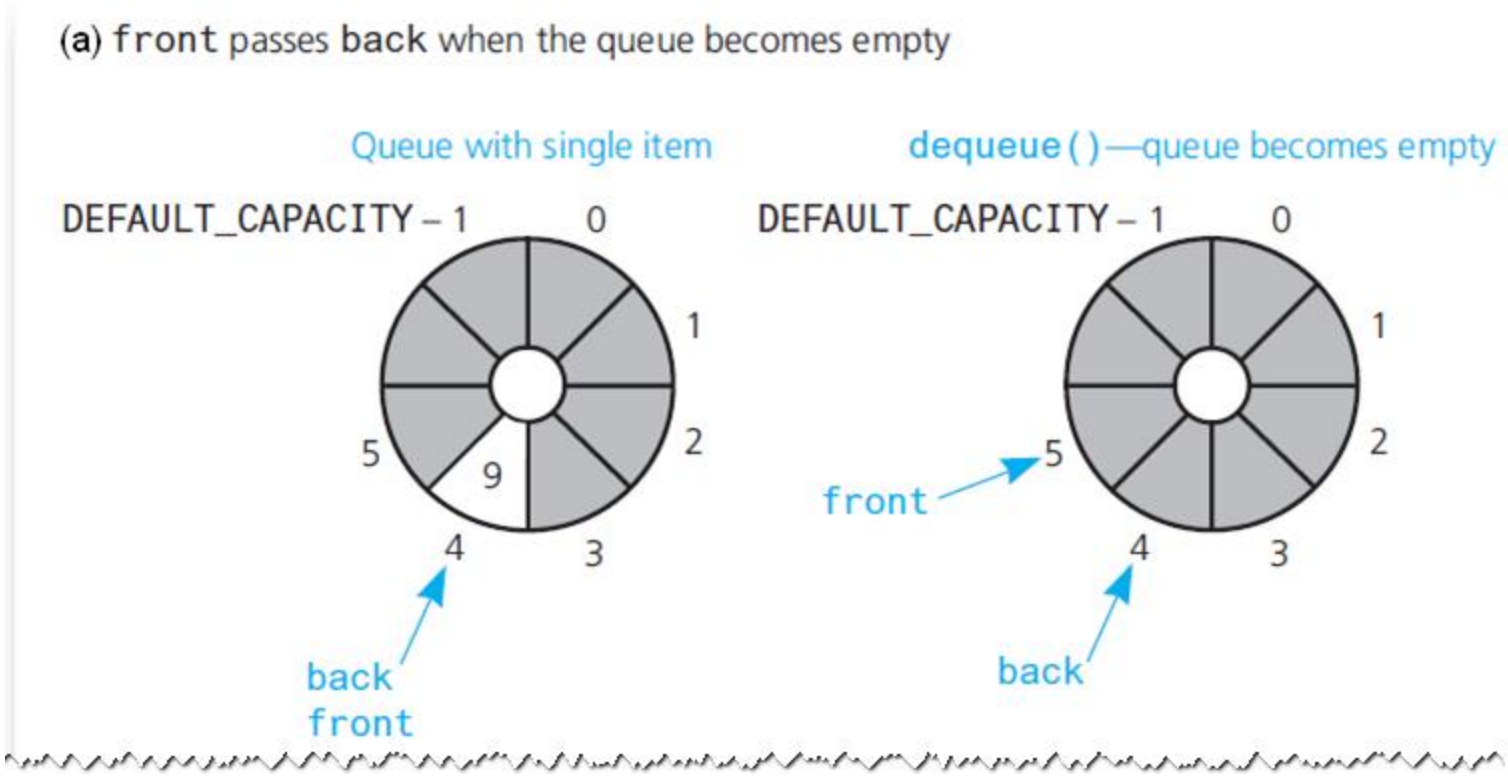
An Array-based Implementation

- The effect of three consecutive operations on the queue



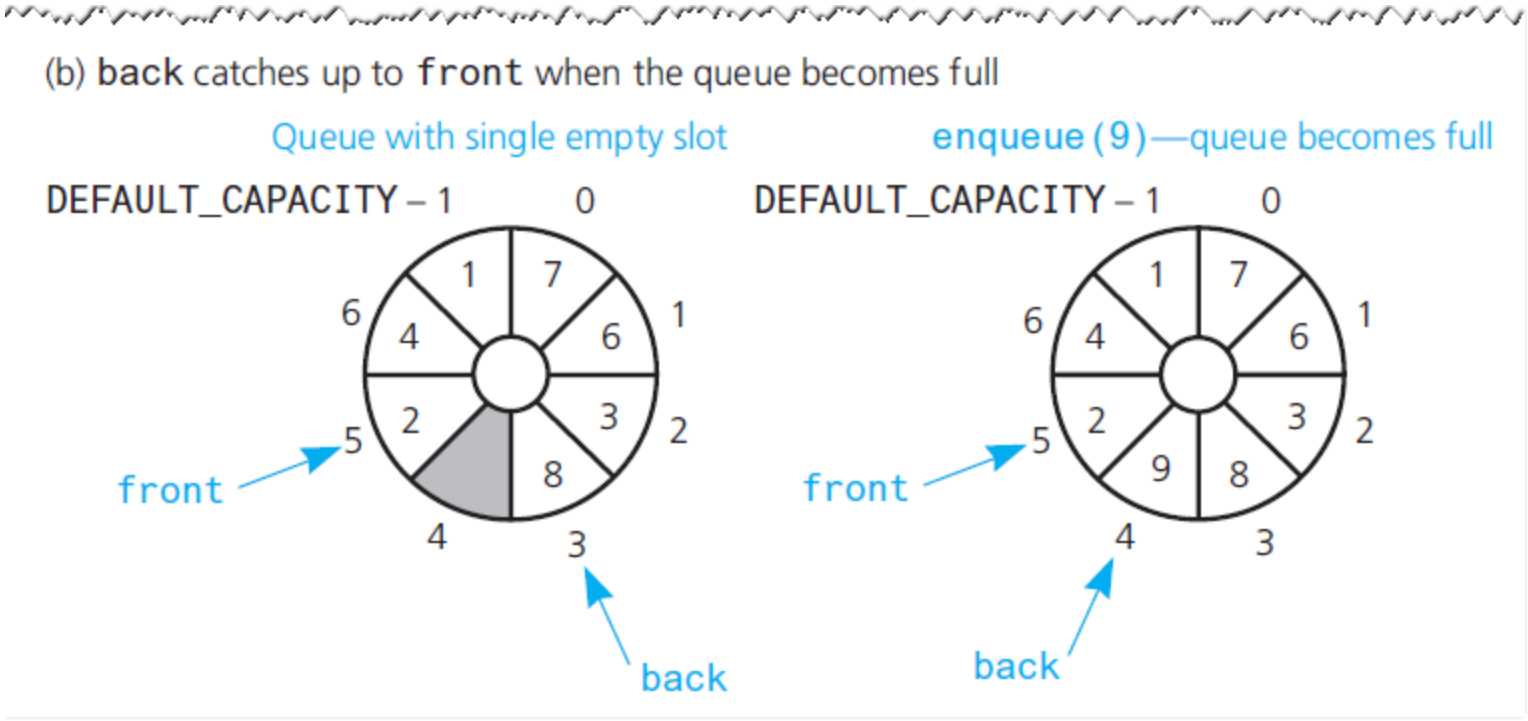
An Array-based Implementation

- Front and back as the queue becomes empty and as it becomes full



An Array-based Implementation

- Front and back as the queue becomes empty and as it becomes full (cont)



An Array-based Implementation

- The header file for the class ArrayQueue

```
#ifndef ARRAY_QUEUE_
#define ARRAY_QUEUE_

#include "QueueInterface.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class ArrayQueue : public QueueInterface<ItemType>
{
private:
      static const int DEFAULT_CAPACITY = 50;
      ItemType items[DEFAULT_CAPACITY];     // Array of queue items
      int front;                            // Index to front of queue
      int back;                             // Index to back of queue
      int count;                            // Number of items currently in the queue
public:
      ArrayQueue();
      bool isEmpty() const;
      bool enqueue(const ItemType& newEntry);
      bool dequeue();

      // @throw  PrecondViolatedExcept if queue is empty
      ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end LinkedQueue
#include "ArrayQueue.cpp"
#endif
```

An Array-based Implementation

- The implementation file for the class ArrayQueue

```
#include "ArrayInterface.h"

template<class ItemType>
ArrayQueue<ItemType>::ArrayQueue() : front(), back(DEFAULT_CAPACITY-1), count()
{
} // end default constructor

template<class ItemType>
bool ArrayQueue<ItemType>::isEmpty() const
{
      return count == 0;
} // end isEmpty

template<class ItemType>
bool ArrayQueue<ItemType>::enqueue(const ItemType& newEntry)
{
      bool result = false;
      if (count < DEFAULT_CAPACITY)
      {
            // Queue has room for another item
            back = (back + 1) % DEFAULT_CAPACITY;
            items[back] = newEntry;
            count++;
            result = true;
      } // end if
      return result;
} // end enqueue
```

An Array-based Implementation

- The implementation file for the class ArrayQueue

```
template<class ItemType>
ArrayQueue<ItemType>::dequeue()
{
      bool result = false;
      if (!isEmpty())
      {
            front = (front + 1) % DEFAULT_CAPACITY;
            count--;
            result = true;
      } // end if
      return result;
} // end dequeue

template<class ItemType>
bool ArrayQueue<ItemType>::peekFront() const throw(PrecondViolatedExcept)
{
      // Enforce precondition
      if (isEmpty())
            throw PrecondViolatedExcept("peekFront() called with empty queue");

      // Queue is not empty; return front
      return items[front];

{
      return count == 0;
} // end peekFront
// end of implementation file
```

Comparing Implementations

- Issues
  - Fixed size (array-based) versus dynamic size (link-based)
  - Reuse of already implemented class saves time

Assignment #3

- Formal document release will take place (this is only an introduction)
- Task: Implemented the ADT Priority Queue
- Additional Tasks defined in the announcement

Assignment #3

- Formal document release will take place (this is only an introduction)
- Main Task: Implemented the ADT Priority Queue
  - Subject to the Header File we will provide
  - Using a Sorted List
- Additional Tasks defined in the announcement

Assignment #3

- Header file for the class SL_PriorityQueue

```
#ifndef PRIORITY_QUEUE_
#define PRIORITY_QUEUE _

#include “PriorityQueueInterface.h”
#include “LinkedSortedList.h”
#include “PrecondViolatedExcept.h”
#include <memory>


template<class ItemType>
class SL_PriorityQueue : public PriorityQueueInterface<ItemType>
{
private:
      std::unique_ptr<LinkedSortedList<ItemType>> slistPtr;       // Ptr to sorted list of items

public:
      SL_PriorityQueue();
      SL_PriorityQueue(const SL_PriorityQueue& pq);
      ~SL_PriorityQueue();

      bool isEmpty() const;
      bool enqueue(const ItemType& newEntry);
      bool dequeue();

      // @throw  PrecondViolatedExcept if priority queue is empty
      ItemType peekFront() const throw(PrecondViolatedExcept);
}; // end SL_PriorityQueue
#include “ArrayQueue.cpp”
#endif
```

Thank you