Chapter 7-1, Lecture notes

The Abstract Data Type Queue

1. The term queue is used in England to describe a line of people.
   1.1 The first person to enter a line at a ticket office is the first one to get service and the first to leave from the "front".
   1.2 New people joining the line are added to the "back."

   1.3 The queue is called a First In, First Out or FIFO.
      1.3.1 This is unlike the stack that basically had only one end to push and pop items off of.
      1.3.2 A queue would have the following ADT operations:

         - Create an empty queue
         - Destroy a queue
         - Determine whether a queue is empty
         - Add a new item to the queue
         - Remove from the queue the item that was added earliest
         - Retrieve from the queue the item that was added earliest

2 Queues are appropriate for many real-life applications (a grocery line, the ticket line, automated teller machine, and so on).
   2.1 Computer science also has applications for queues.

      2.1.2 When you print to a printer, the computer sends faster than the printer can print.
      2.1.3 Your print job is held in a queue until it is printed.
      2.1.4 If another print job had been sent while the first printing is still going on, it is put in the queue to wait its turn.

   2.2 All these applications involve waiting.
      2.2.1 There have been "simulations" to see if the wait can be reduced.
      2.2.2 For instance, banks used to have one line per teller which could cause waiting if you happened to be in a line unlucky enough to have someone take a LOT of the teller's time.
      2.2.3 By having one long queue for all the tellers, it goes much faster since even with one teller having to spend a lot of time with one customer the other tellers come available relatively quickly.

3 Figures 7-1 and 7-2 on page 345 would give the UML contract and some queue operations.
   3.1 We use the terms of enqueue to add an item to the back and dequeue to remove an item from the front.

3.2 And, like the stack, we have two kinds of dequeues: one that just removes and one that retrieves THEN removes.

3.3 The UML contract is like so:

Operation Contract for the ADT Queue
QueueItemType is the type of the items stored in the queue

isEmpty():boolean {query}
Determines whether this queue is empty

enqueue(in newItem:QueueItemType) throw QueueException
Inserts newItem at the back of this queue. Throws QueueException if the insertion is not successful

dequeue() throw QueueException
Removes the front of this queue; that is, removes the item that was added earliest. Throws QueueException if the deletion is not successful

dequeue(out queueFront:QueueItemType) throw QueueException
Retrieves into queueFront and then removes the front of this queue. That is, retrieves and removes the item that was added earliest. Throws QueueException if the deletion is not successful

getFront(out queueFront the front of this queue. That is, retrieves the item that was added earliest. Throws QueueException if the retrieval is not successful. The queue is unchanged.

3.4 The operations and their effect on the queue is like so:

Operation	Queue after operation
	front |
aQueue.createQueue()	V
aQueue.enqueue(5)	5
aQueue.enqueue(2)	5 2
aQueue.enqueue(7)	5 2 7
aQueue.getFront(queueFront)	5 2 7 (queueFront is 5)
aQueue.dequeue(queueFront)	2 7 (queueFront is 5)
aQueue.dequeue(queueFront)	7 (queueFront is 2)

Chapter 7-2, Simple Applications of the ADT Queue

4 Remember that the applications for a queue would use the operations independent of the actual implementations (more on that later).

4.1 Let's try a few applications using a queue.

4.1.1 For instance, entering text at a keyboard could have a queue to store the characters in order they were typed.

4.1.2 For instance:

```
// read a string of character form a single line of input into a queue
aQueue.createQueue()
while (not end of line)
{       Read a new character ch
        aQueue. Enqueue(ch)
}       // end while
```

4.2 Once in the queue, the system can process them.

4.2.1 So if you typed in an integer without any imbedded blanks but could have a leading blanks or a trailing blanks, you could process the queue.

4.2.2 If you entered the characters of 2, 4, and 7, the system would convert them from characters into an integer like so:

10 * (10*2 + 4) + 7

4.3 The following pseudo code performs this conversion in like:

```
// convert digits in a queue aQueue into a decimal integer n

// get first digit, ignoring any leading blanks
do
{       aQueue.dequeue(ch)
}       while (ch is blank)

// Assertion: ch contains first digit

// compute n from digits in queue
n = 0
done = false
do
{       n = 10 * n + integer that ch represents
        if (!aQueue.isEmpty()
                aQueue.dequeue(ch)
        else
                done = true;
} while (!done and ch is a digit)

// Assertion: n is result
```

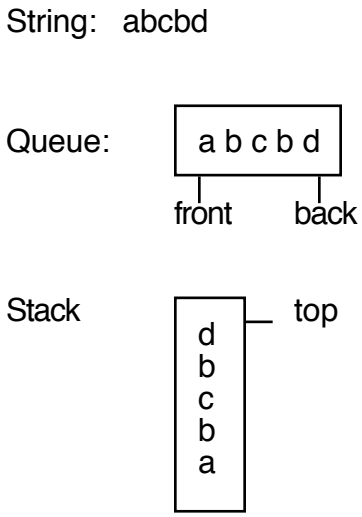
4.4 Now one method to get an integer digit out of a character is to do this:

Digit = ch – '0'

4.4.1 That would take the ASCII value of 0 and subtract it from the character
4.4.2 Thus resulting in an integer digit.

5 Let's take the example of a palindrome – a string that reads the same from left to right as right to left.
5.1 With a stack, you can reverse the order of characters.
5.2 However, with a queue, you can preserve the order.
5.2.1 If you take the characters as they come in and enqueue then AND push them onto the stack, you can dequeue and pop and check to see if they match.

5.2.2 For instance Figure 7-3 on page 347 has:



5.2.3 The pseudo code would look like:

```
isPal(in str:string):boolean
// Determines whether str is a palindrome
// create an empty queue and an empty stack
aQueue.createQueue()
aStack.createStack()
// insert each character of the string into both the queue and the stack
length = length of str
for (i = 1 through length)
{       nextChar = i^th character of str
        aQueue.enqueue(nextChar)
        aStack.push(nextChar)
}       // end for
```

```
// compare the queue characters with the stack characters
charactersAreEqual = true
while (aQueue is not empty and charactersAreEqual)
{       aQueue.getFront(queueFront)
        aStack.getTop(stackTop)

        if (queueFront equals stackTop)
        {       aQueue.dequeue()
                aStack.pop()
        }
        else
                charactersAreEqual = false
}       // end while
```

Chapter 7-3, Implementations of the ADT Queue

1 Like stacks, queues can have array-based or pointer-based implementation.

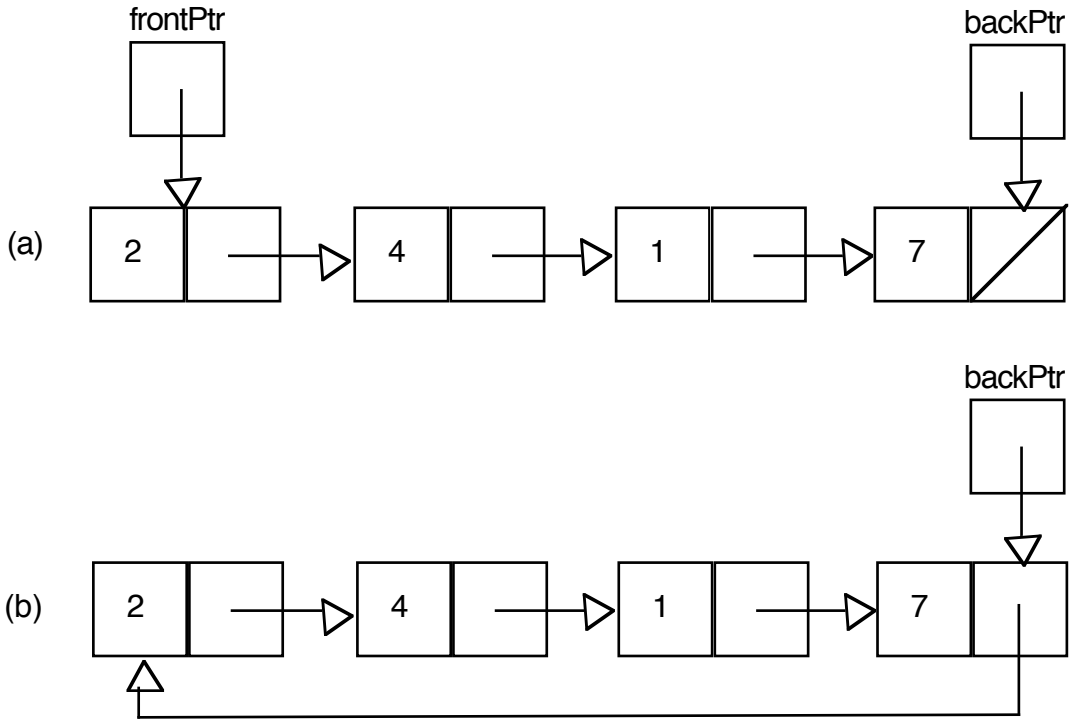
1.1 And the queue would have the following definition of QueueException:

```
#include <stdexcept>
#include <string>
using namespace std;

class QueueException : public logic_error
{
public:
        QueueException(const string & message = “”) : logic_error(message.c_str())
        {} // end constructor
{;      // end QueueException
```

1.2 The pointer-based implementation is actually more straightforward than the array-based one so that will be first.

1.3 Figure 7-4 on page 349 shows a couple of options. Like so:



1.4 Figure 7-4 a has two pointers: a frontPtr and a backPttr.

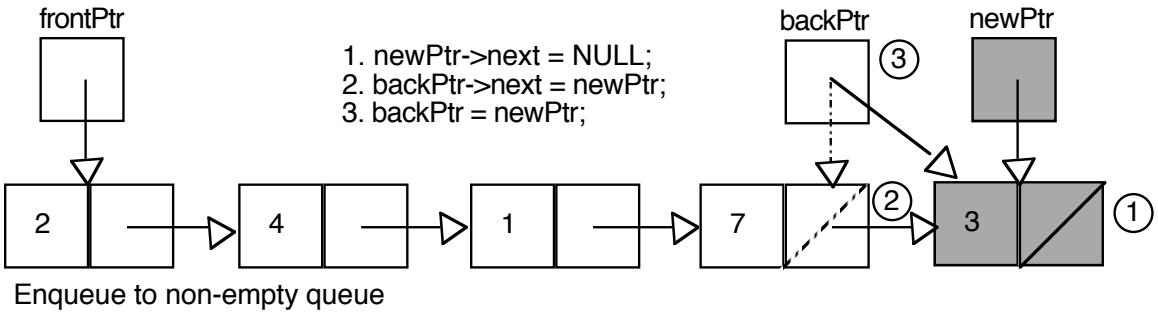
1.4.1 When you enqueue, you put the new node on the end of the queue and all you need do is change the backPtr.

1.4.2 With dequeue, you remove from the font of the queue and change frontPtr.

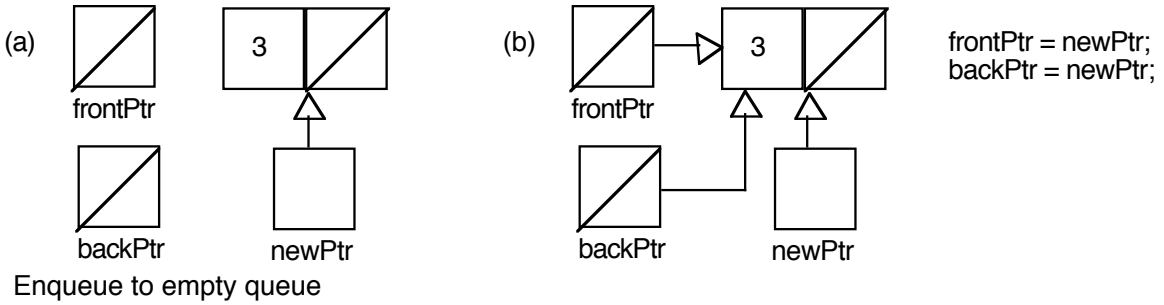
1.4.3 You might have to change backPtr also if you wind up with an empty list.

1.4.4 With Figure 7-4 b you have only one pointer but you could readily find the front for dequeue and the back for enqueue.

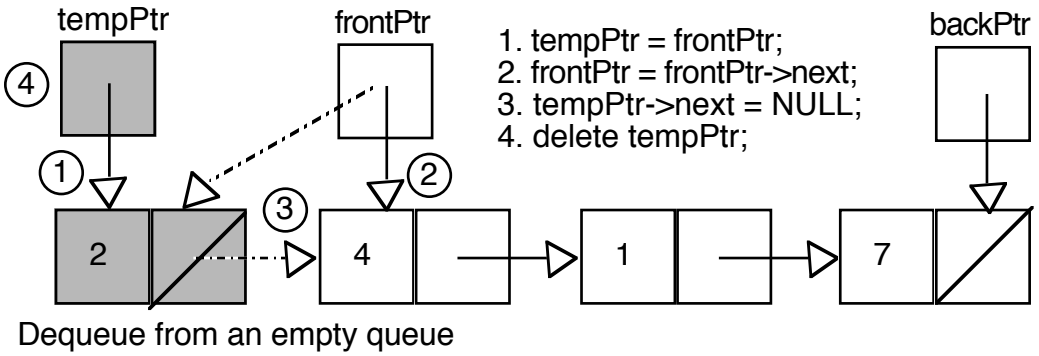
1.4.5 Figure 7-5 on page 350 shows enqueue to a non-empty:



1.4.6 Figure 7-6 shows enqueue to an empty queue:



1.4.7 Then Figure 7-7 shows dequeuing from a non-empty queue:



1.5 Note that you only need to change the frontPtr.

1.5.1 Removing the last node would be a special case that would involve the backPtr.
1.5.2 Let's have the code that does the pointer-based implementation.
1.5.3 Note, with pointers, you must have both a constructor and a destructor

2 The code for the header files and implementation files for pointer-based queues are under queuep in the Documents folder on Blackboard/WebCT.
2.1 That folder also has code for pointer based stacks and a "pal.cpp" program that does a version of the palindrome code that uses stacks and queues.
2.2 Note: it does also show how to convert a string (use str variable name) into a c-string (use c_str variable name).
2.3 Doing so can be a bit of a pain without the correct code, like so:

```
char *c_str = new char[str.c_str()+1];
strcpy(c_str, str.c_str());
```

2.4 The memory allocated must have an extra space (i.e. str.c_str()+1) for the terminating null character (i.e. '\0') that all c-string must have.

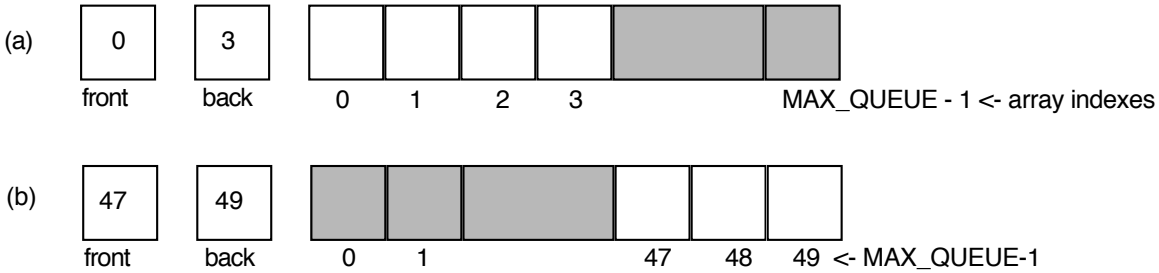
3 The array-based implementations can be used where you are reasonably assured of a fixed-sized queue.
3.1 We need a front and back index and an array, like so:

```
const int MAX_QUEUE = maxium_size_of_queue;
typedef desired-type-of-queue-item QueueItemType;

QueueItemType items[MAX_QUEUE];
int           front;
int           back;
```

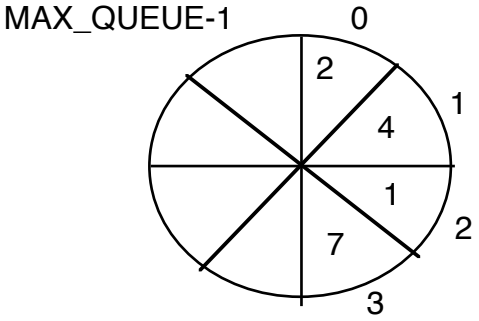
3.2 Now if we make MAX_QUEUE size be 50 and initially set the indexes of front and back to zero, we can have back incremented when we add to the end of the queue and front increment when we take from the front.

3.3 And – uh, ho. This will have problems as Figure 7-8 on page 355 shows like so:



3.4 The indexes of front and back do indeed work fine like in part a until we hit the end in part b (called "rightward drift").

- 3.4.1 Then we run out of spaces.
- 3.4.2 We need to reuse the spaces.
- 3.4.3 But how?
- 3.4.4 We could shuffle all the nodes to the left but that is inefficient.
- 3.4.5 There is another way: make the queue circular, like shown on page 355, figure 7-8.



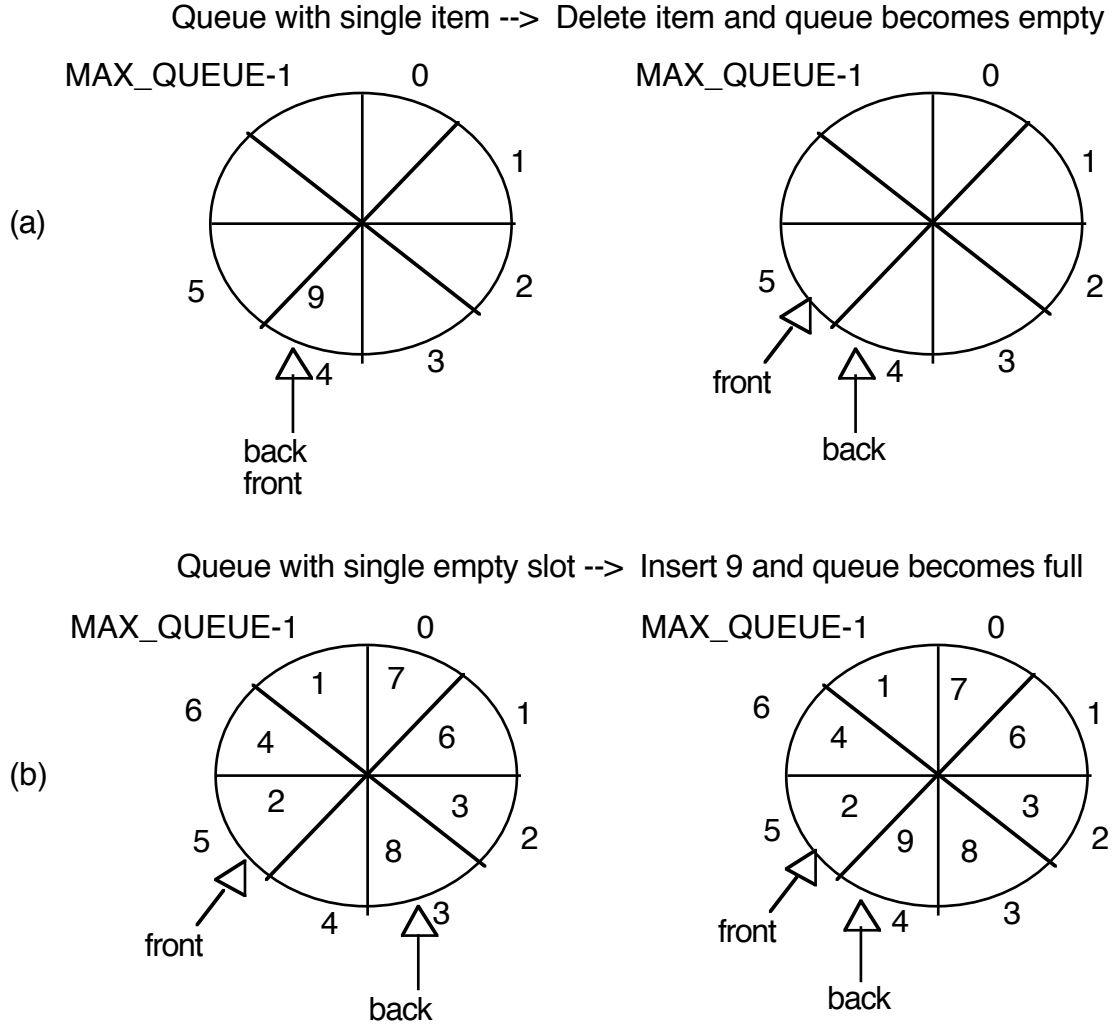
3.5 When you keep incrementing front and back, if you reach the end, just have the indexes start at 0 again!

- 3.5.1 That allows all the slots to be reused.
- 3.5.2 The only problem is how to tell when the queue is full.
- 3.5.3 You can simply say:

front is one slot ahead of back

3.5.4 It appears that front "passes" back though back will eventually "catch up to front".

3.5.5 Figure 7-11 on page 357 shows the problem:



3.6 Part a shows that front will pass back when the last item is deleted.

3.6.1 Also, part b shows what happens when the queue is full – yes, front is past back.

3.6.2 So how to distinguish between the two conditions.

3.6.3 One way is to use a counter.

3.6.4 The following code implements the counter and shows how to make an index circular by using the modulus operation of "%", like so:

```
back = (back+1) % MAX_QUEUE;
items[back] = newItem;
++count;
```

3.6.5 The back index would automatically wrap around to the front. To increment the front do:

```
front = (front+1) % MAX_QUEUE;
--count;
```

3.6.6 The code for the array-based implementation is under Documents, queuea.

3.6.7 It also had the array-based stacks and the palindrome program that uses both queue and stack.

3.7 Instead of using a counter, you can use a isFull flag.

3.7.1 However, that takes about as much time as a counter.

3.7.2 A faster way to see if the queue is full is to sacrifice one slot by checking:

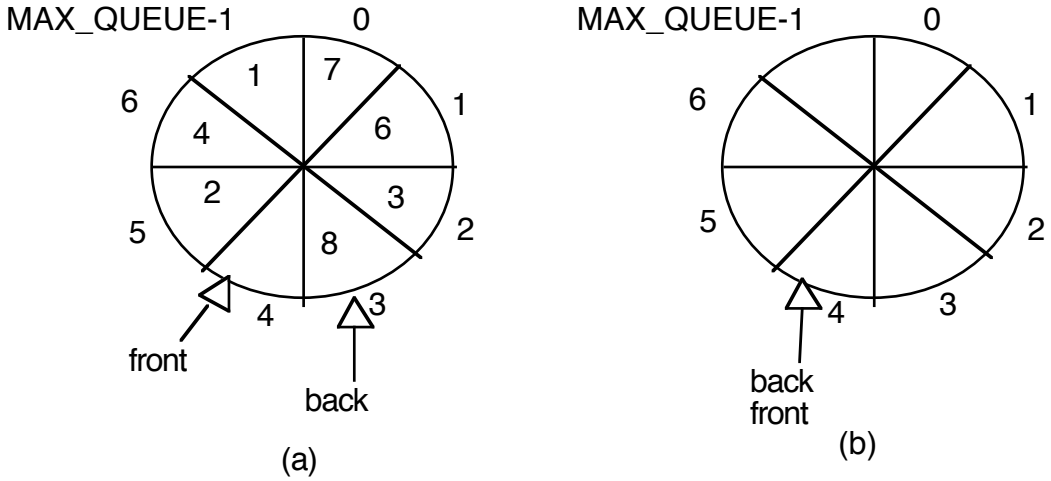
front equals (back+1) % (MAX_QUEUE+1)

3.7.3 but if the queue is empty:

front equals back

3.7.4 Figure 7-12 on page 360 shows how it works:

(b)



3.7.5 It does sacrifice one slot but it is more efficient.

4 You can also implement the queue using the ADT list.

4.1 The Blackboard/WebCT Documents folder has queuel folder with a copy of the palindrome.

4.2 Of course there is a queue option in the STL.

4.2.1 With the STL queue, to get an item from the front you use the "front" method and not the "getFront" we've been using.

4.2.2 Also, enqueue would be “push” and dequeue would be “pop” – like the stack operations only it does work with front and back.

4.2.3 The following is a simplified version of the operations for the STL queue:

```
Template <typename T, typename Container = deque <T> >
class queue
{
public:
  /** Default constructor, initializes an empty queue
   *  @pre None
   * @post An empty queue exists */
    explicit queue(const Container& cnt = Container());

  /** Determines whether the queue is empty
    * @pre None
    * @post None
    * @return True   if the queue is empty, otherwise returns false */
  bool empty() const;

  /** Determines the size of the queue. size_type is an integral type
    * @pre None
    * @post None
    * @return The number of items that are currently in the queue */
  size_type size() const;

  /** Returns a reference to the first item in the queue
   * @pre None
   * @post The item is not removed from the queue
   * @return a reference to the first item in the queue */
  T &front();

  /** Returns a reference to the last item in the queue
   * @pre None
   * @post The item is not removed from the queue
   * @return A reference to the last item in the queue */
  T &back();

  /** Removes the first item in the queue
   * @pre None
   * @post The item at the front of the queue is removed. */
  void pop();

  /** Inserts an item at the back of the queue
   * @pre None
   * @post The item x is at the back of the queue
```

```
   *  @param x  The item to push */
  void push(const T& x);

}; // end std::queue
```

4.3 STL class queue uses more basic container types.

4.3.1 Classes whose implementations use other classes are called “adapter containers”.

4.3.2 The three basic container types used by adapter containers are: list, vector, and deque.

4.3.3 We’ve used the list in chapter 4 and used the vector in chapter 5.

4.3.4 The deque type supports “double-ended queue” – a slight variation on the queue.

4.3.5 If you do the following:

```
queue<int> myQueue;
```

4.3.6 You are using the default of deque. However, if you wanted the STL queue to use the list instead, you do:

```
queue<int, list<int> > myQueue;
```

4.4 And example of using the STL queue and stack is:

```
#include <iostream>
#include <list>
#include <queue>
#include <stack>

using namespace std;

int main()
{
        list<int> myList;        // create an empty list
        list<int>::iterator i = myList.begin();

        for (int j = 1; j  < 5; j++)
        {       i = myList.insert(i, j);
                i++;
        } // end for

        cout << “myList:   “;
        i = myList.begin();
        while (i != myList.end())
        {       cout << *i  << “ “;
                i++;
```

```
        } // end while
        cout << endl;

        // assume the front of the list is the front of the queue
        queue<int, list<int> > myQueue(myList);

        // assumes the back of the list is the top of the stack
        stack<int, list<int> > myStack(myList);

        cout << "myQueue:  ";
        while (!myQueue.empty())
        {       cout << myQueue.front() << " ";
                myQueue.pop();
        } // end while

        cout << "myStack:  ";
        while (!myStack.empty())
        {       cout << myStack.top() << "  ";
                myStack.pop();
        } // end while
        cout << endl;
        return 0;
} // end main
```

5 Which implement to use?

6 An array-based implementation is faster but limited in space. If you do not know how many items can be in a queue, use the pointer based.