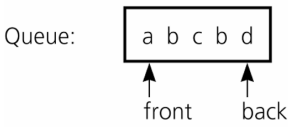
Queue

- FIFO: first in first out
- Operations on a queue occur only at its two ends: front and back (rear)
- Application:
  - Waiting line
  - Simulations



What is coming

- Queue applications
- Implementation of Queue
  - Pointer-based Q
  - Array-based Q

UML diagram for the class **Queue**

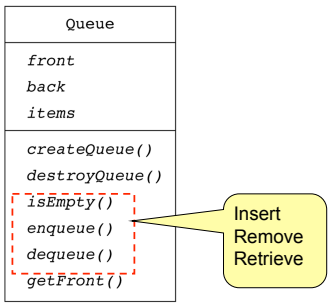


Figure 7-2: Some queue operations

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

```
class Queue   {
  public:
    Queue();                        // default constructor
    Queue(const Queue& Q);          // copy constructor
    ~Queue();                       // destructor
    bool isEmpty( ) const;
    void enqueue(QueueItemType newItem);
    void dequeue( );
    void dequeue(QueueItemType& queueFront);
    void getFront(QueueItemType& queueFront) const;
  private:
    struct QueueNode   {
        QueueItemType item;
        QueueNode *next;
    }; // end struct
    QueueNode *frontPtr;
    QueueNode *backPtr;
}; // end class
```

Application of Queue

- Reading String of characters
- Recognizing palindromes

Example 1: Reading String of characters

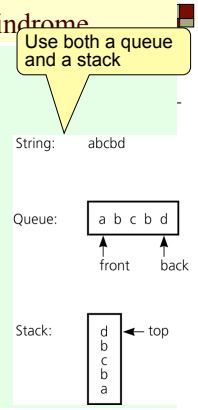
- Converts digits in a queue into a decimal integer result
  Queue may store leading whitespaces

```
//  247 = 2*100 + 4*10 +7 = 10*(10*2 + 4) + 7
do {     // skip leading blanks
     aQueue.dequeue(ch)
} while (ch is blank)
result =0          // result to be stored
done=false
do {
      result =10* result + int(ch-'0')
     if (!aQueue.isEmpty())
          aQueue.dequeue(ch)
     else
          done=true
} while(!done and ch is a digit)
```

Example 2: Recognizing palindrome

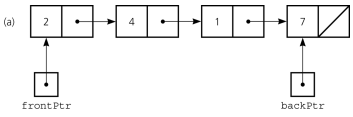
Use both a queue and a stack

```
isPal (in str:string): boolean
     aQ.createQueue ( )
     aStack.createStack ( )
     length = length of str
     for (i=1 though length) {
          aQ.enqueue(ith character of str)
          aStack.push(ith character of str)
     }
     isEqual = true
     while (!aQ.isEmpty() and isEqual) {
          aQ.getFront(qFront)
          aStack.getTop(stackTop)
          if (qFront equals stackTop) {
                    aQ.dequeue ( )
                    aStack.pop ( )
          } else
                    isEqual = false
     }
     return isEqual
```



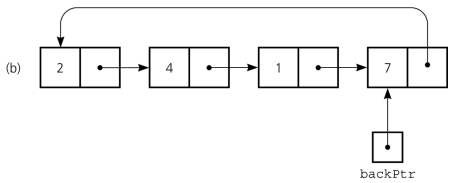
Pointer-based implementation

- Linear linked list with two external pointer
- Insert at the back and delete from the front



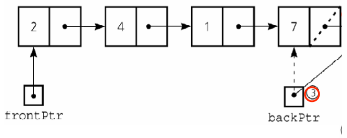

```
class Queue   {
  public:
    Queue();                        // default constructor
    Queue(const Queue& Q);  // copy constructor
    ~Queue();                       // destructor
    bool isEmpty( ) const;
    void enqueue(QueueItemType newItem);
    void dequeue( );
    void dequeue(QueueItemType& queueFront);
    void getFront(QueueItemType& queueFront) const;
  private:
    struct QueueNode   {
        QueueItemType item;
        QueueNode *next;
    }; // end struct
  QueueNode *frontPtr;
  QueueNode *backPtr;
}; // end class
```

A pointer-based implementation of a queue: a circular linear linked list with one external pointer



Back is backPtr
Front is backPtr->next

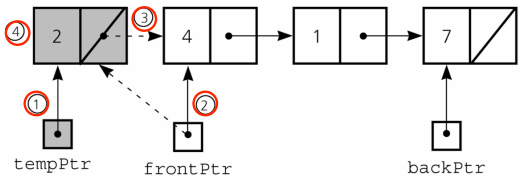
Inserting an item into a nonempty queue



1. newPtr->next = NULL;
2. backPtr->next = newPtr;
3. backPtr = newPtr;

Inserting an item into an empty queue:
(a)before insertion;
(b)after insertion

Deleting an item from a queue of more than one item



1. tempPtr = frontPtr;
2. frontPtr = frontPtr->next;
3. tempPtr->next = NULL;
4. Delete tempPtr;

Queue: array-based implementation

- If fixed sized queue is not a problem→ array
- A naive array-based implementation of a queue
- Rightward drift can cause the queue to appear full

Queue: array-based Q

- Naïve array-based implementation
  - front = 0; back = -1; when empty
  - enQ :
    - back++;
    - items[back];
  - deQ : front++;
  - if (back < front) → empty
  - if(back == maxQ –1) → full
  - → right drift

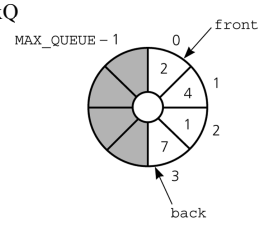
Queue: array-based Q

- Solution 1:
  - shift array element
  - Too expensive: it would dominate the cost of the implementation

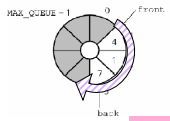
Queue: array-based Q

- Solution 2: use circular array
  - deQ → (++front) % maxQ
  - enQ → (++back) % maxQ

How do we detect
Q empty or full?



The effect of some operations of the queue



dequeue

front = 1
back = 3

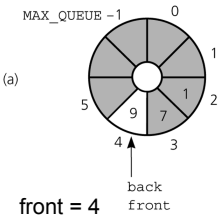
Enqueue 9

front = 2
back = 3

front = 2
back = 4

Is this full or empty?

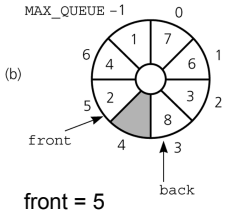
Queue with single item



front = 4
back = 4

front passes **back** when the queue becomes empty
front = 5 and back = 4

Queue with single empty slot



front = 5
back = 3

back catches up to **front** when the queue becomes full
front = 5 and back = 4

Array-based Q (Circular array)
If front is one slot ahead of back, is Q full or empty?

1. Keep a count of the number of items in the Q
   - new empty Q: front = 0; back = maxQ −1; size =0;
   - Check the size before deleting or inserting an item
   - Enqueue:

```
if (size<maxQ) { // not full
    back = (back+1) % maxQ;
    items[back] = newItem;
    ++size;
}
```

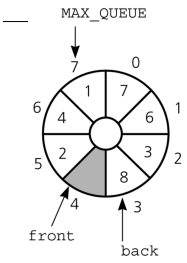
   - Dequeue:

```
if(size >0) { // not empty
        front = (front+1) % maxQ;
        --size;
}
```

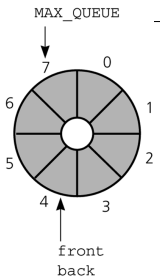
Array-based Q (Circular array)
If front is one slot ahead of back, is Q full or empty?

2. Use only maxQ element with maxQ+1 size array
   - Sacrifice one element and make front (actual front-1)
   - full : if front == (back+1) % (maxQ+1)
   - empty : if front == back
3. No need for keeping tract of queue size

Use MAX_QUEUE-1 number of elements in the array.
Sacrifice one element, front is (actual front-1)



A full queue



An empty queue