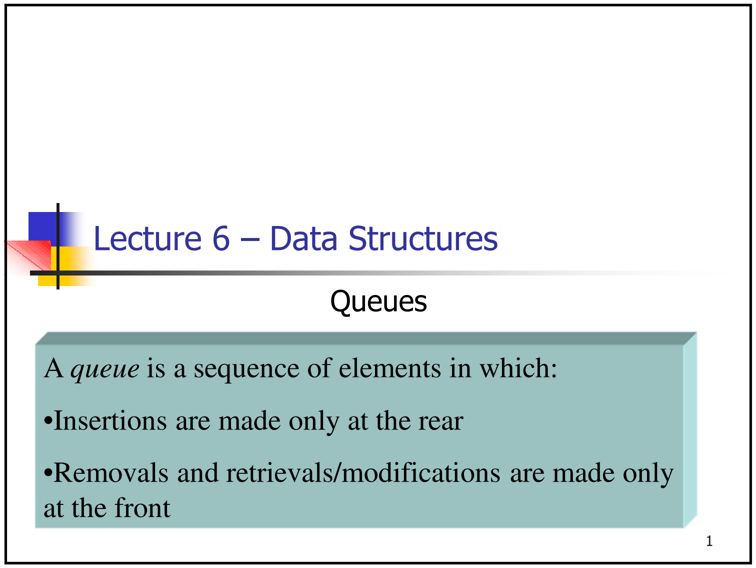


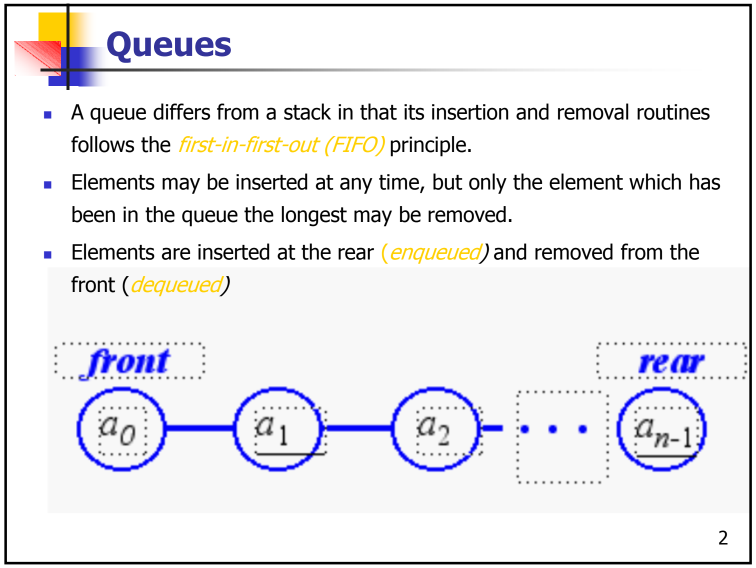
Lecture 6 – Data Structures

Queues

A *queue* is a sequence of elements in which:

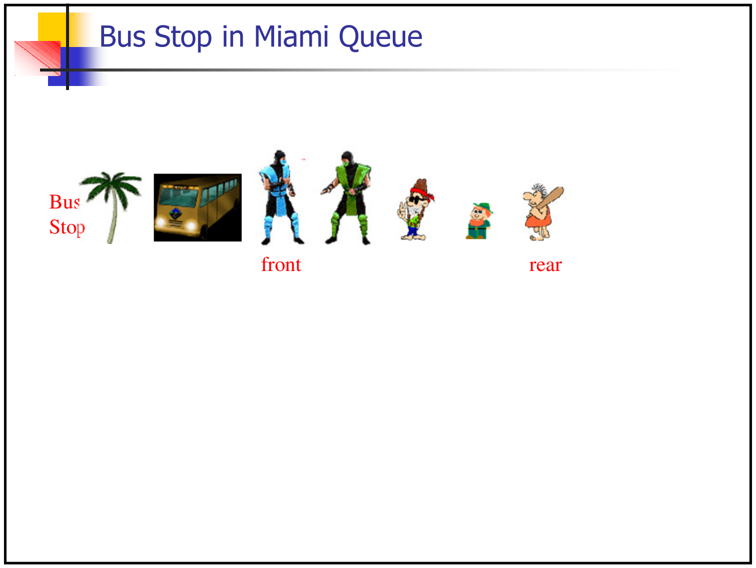
•Insertions are made only at the rear

•Removals and retrievals/modifications are made only at the front

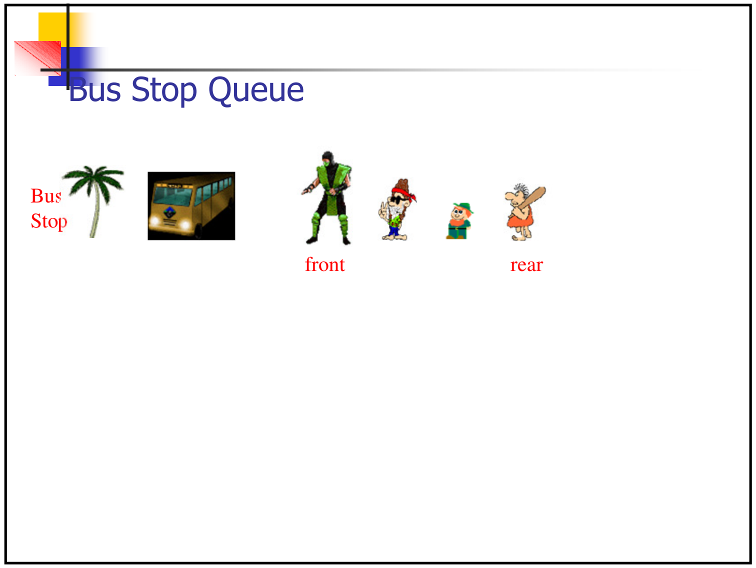


Queues

- A queue differs from a stack in that its insertion and removal routines follows the *first-in-first-out (FIFO)* principle.
- Elements may be inserted at any time, but only the element which has been in the queue the longest may be removed.
- Elements are inserted at the rear *(enqueued)* and removed from the front (*dequeued)*

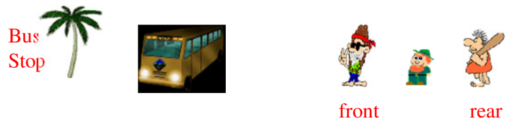


Bus Stop in Miami Queue

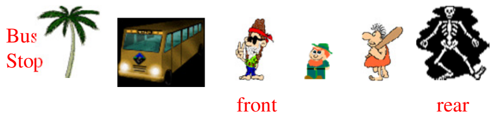


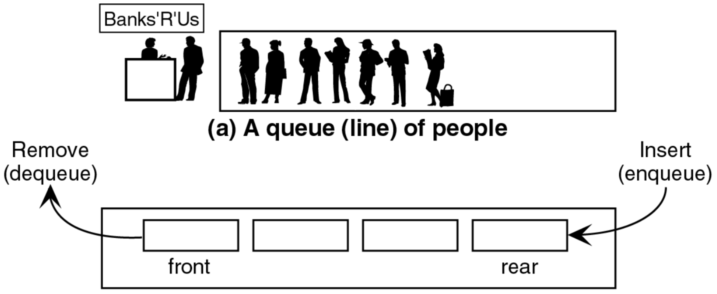
Bus Stop Queue

Bus Stop Queue



Bus Stop Queue



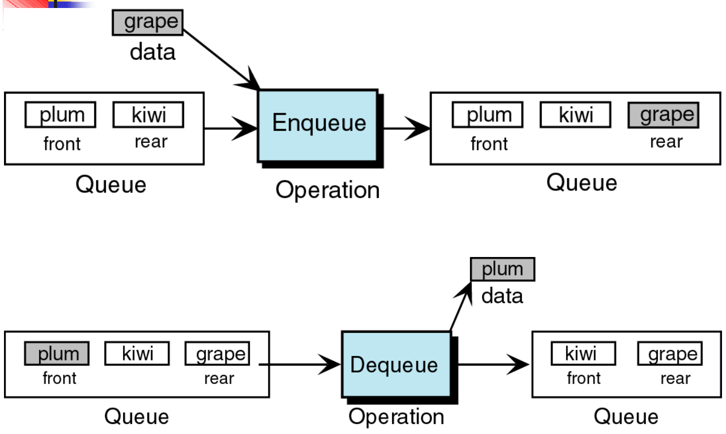


(a) A queue (line) of people

(b) A computer queue

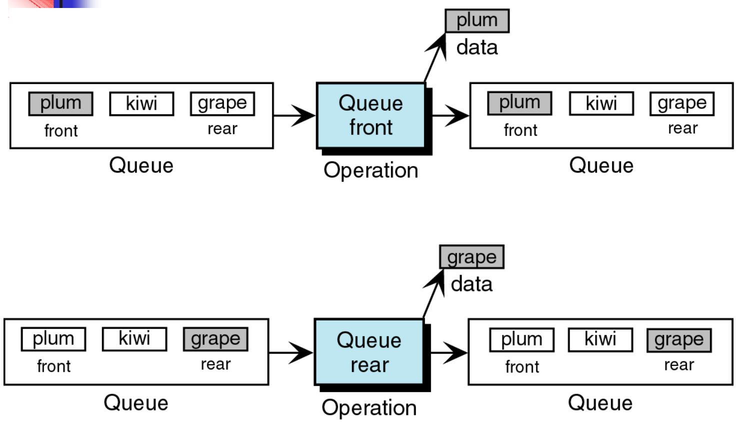
The Queue Abstract Data Type

- The queue has two fundamental methods:
  - **enqueue(o):** *Insert* object *o* at the *rear* of the queue
  - **dequeue():** *Remove* the object from the *front* of the queue and *return* it; *an error occurs if the queue is empty*
- These support methods should also be defined:
  - **size():** Return the *number of objects* in the queue
  - **isEmpty():** Return a *boolean* value that indicates whether the queue is *empty*
  - **front():** *Return*, but *do not remove*, the *front* object in the queue; *an error occurs if the queue is empty*



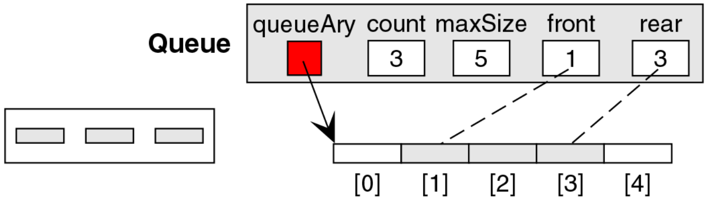
grape
data
plum kiwi
front rear
Queue
Enqueue
Operation
plum kiwi grape
front rear
Queue

plum
data
plum kiwi grape
front rear
Queue
Dequeue
Operation
kiwi grape
front rear
Queue



plum
data
plum kiwi grape
front rear
Queue
Queue front
Operation
plum kiwi grape
front rear
Queue

grape
data
plum kiwi grape
front rear
Queue
Queue rear
Operation
plum kiwi grape
front rear
Queue



Queue
queueAry count maxSize front rear
3 5 1 3
[0] [1] [2] [3] [4]

(a) Conceptual array (b) Physical array

QUEUES

A **queue** is an *ordered collection of homogeneous data* items such that:

- items can be *removed only at one end* (*front* or *head* of the queue)
- items can be *added only at the other end* (*back* or *rear* of the queue)
- A queue is a **FIFO** *"first in, first out"* structure.

Basic operations are:

construct: Create an empty queue

empty: Check if a queue is empty

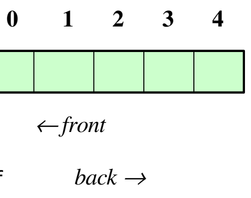
addQ: Add a value at the back of the queue

front: Retrieve the value at the front of the queue

removeQ: Remove the value at the front of the queue

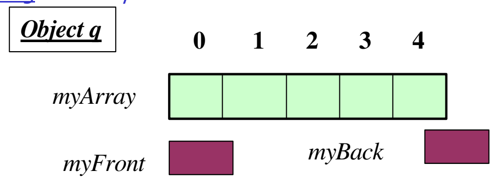
Array-Based Implementation of Queues

- *myArray* to store the elements of the queue



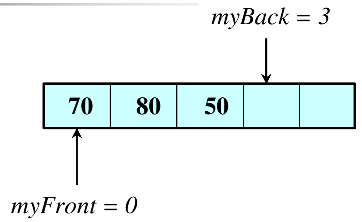
two integer variables

- *myFront* The position in the array of the element that can be removed – the position of the *front queue element*
- *myBack* The position in the array of the element that can be added – the position *following the last queue element*

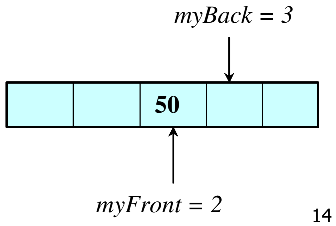


Example: (a) *QUEUE_CAPACITY = 5*

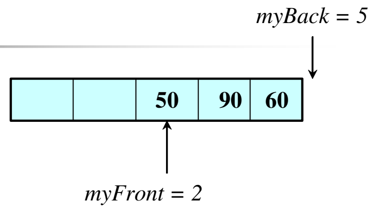
- **(b)** The sequence of operations is *addQ 70, addQ 80,* and *addQ 50*



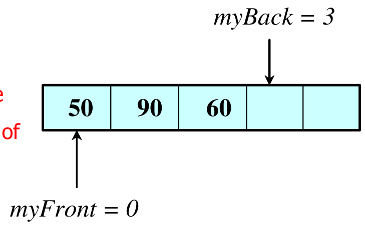
- Two elements are removed

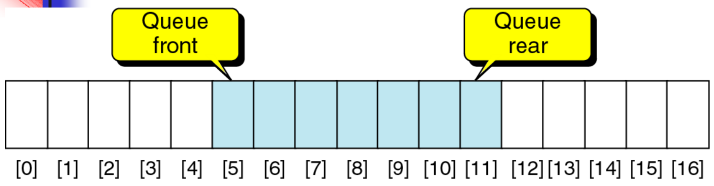


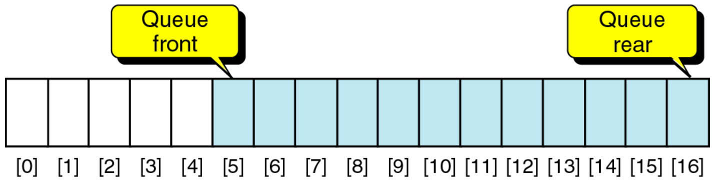
- *90* and *60* are then *added*



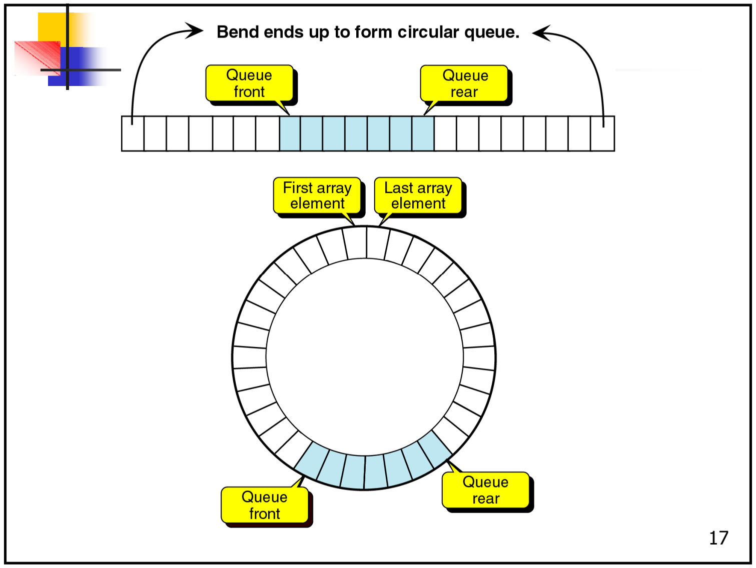
- Before another item can be inserted into the queue, the elements in the array must be shifted back to the beginning of the array:





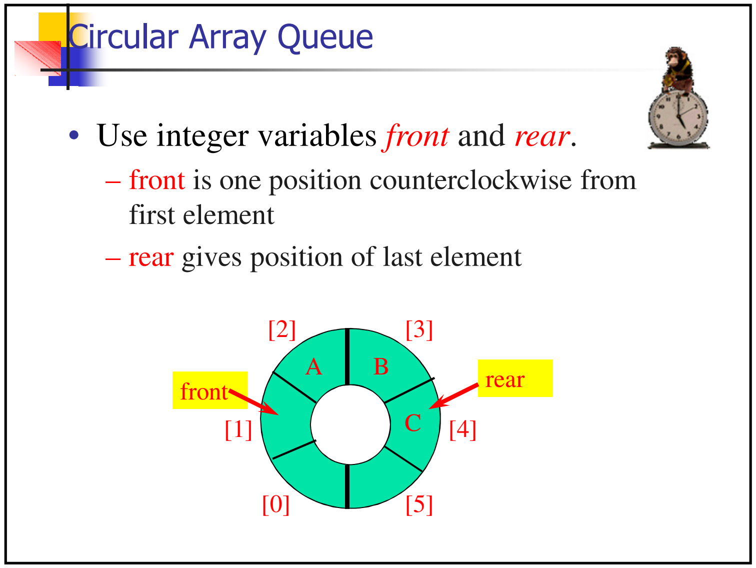


maxsize
emptyQ -> front == back   fullQ ->(back+1)%maxsize
increment -> back = (back+1) % maxsize   front -> front = (front+1) % maxsize



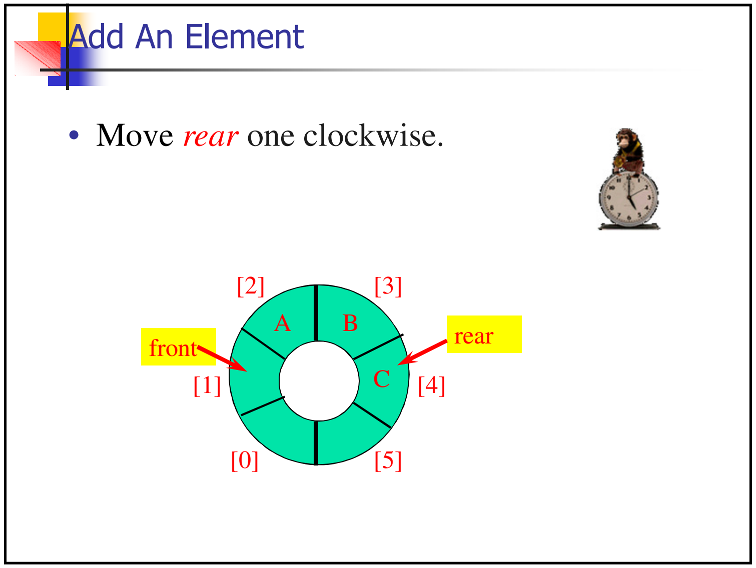
Bend ends up to form circular queue.

17



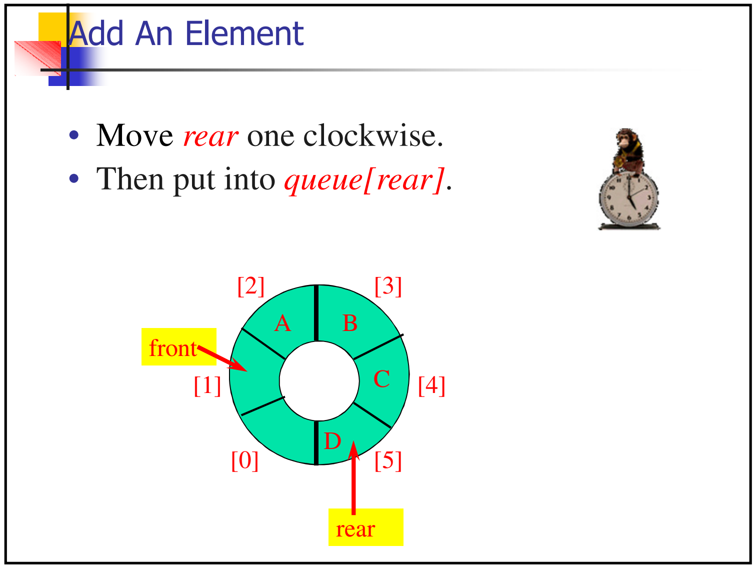
Circular Array Queue

- Use integer variables *front* and *rear*.
  - front is one position counterclockwise from first element
  - rear gives position of last element



Add An Element

- Move *rear* one clockwise.

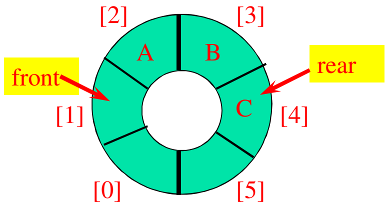


Add An Element

- Move *rear* one clockwise.
- Then put into *queue[rear]*.

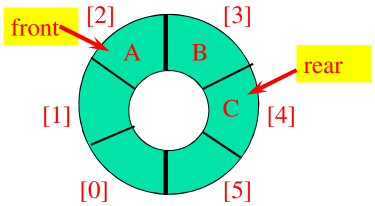
Remove An Element

- Move *front* one clockwise.

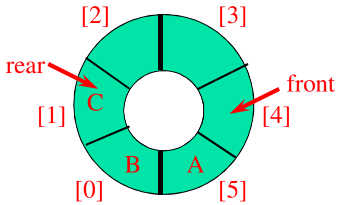


Remove An Element

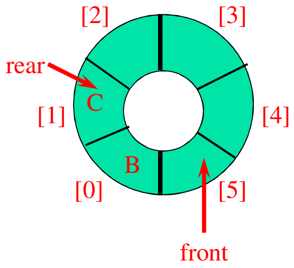
- Move *front* one clockwise.
- Then extract from *queue[front]*.



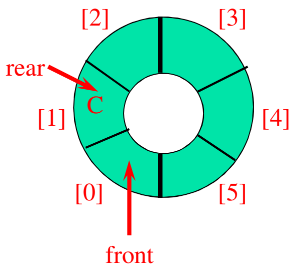
Empty That Queue



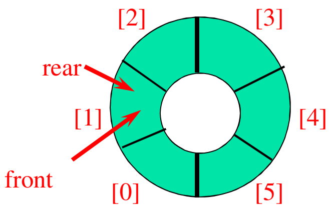
Empty That Queue



Empty That Queue

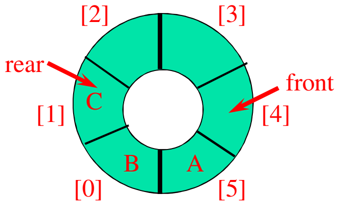


Empty That Queue

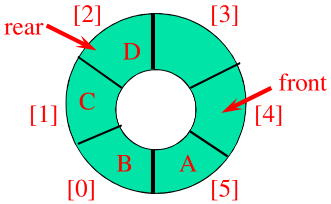


- When a series of removals causes the queue to become empty, *front = rear*.
- When a queue is constructed, it is empty.
- So initialize *front = rear = 0*.

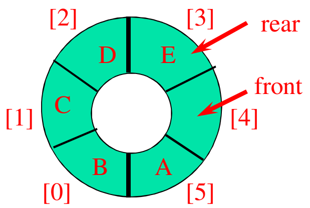
A Full Tank Please



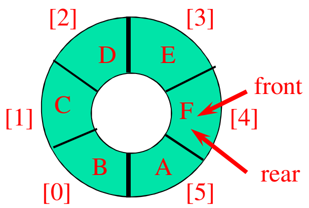
A Full Tank Please



A Full Tank Please



A Full Tank Please

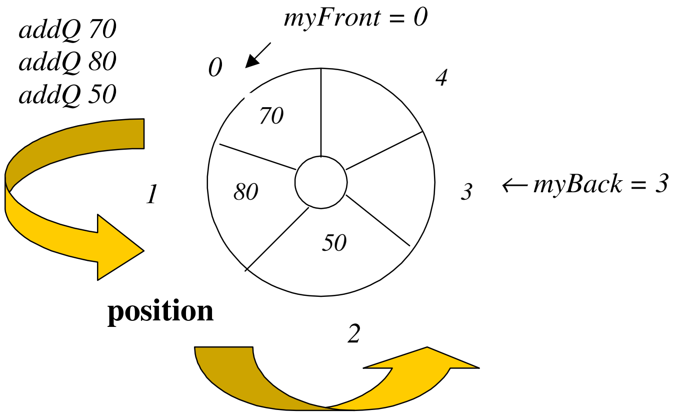


- When a series of adds causes the queue to become full, *front = rear.*
- So we cannot distinguish between a full queue and an empty queue!

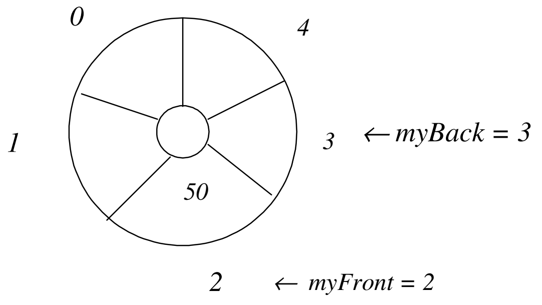
Ouch!!!!!

- Remedies.
  - Don’t let the queue get full.
    - When the addition of an element will cause the queue to be full, increase array size.
    - This is what the text does.
  - Define a boolean variable *lastOperationIsPut*.
    - Following each *put* set this variable to *true*.
    - Following each *remove* set to *false*.
    - Queue is empty if *(front == rear) && !lastOperationIsPut*
    - Queue is full if *(front == rear) && lastOperationIsPut*
  - Performance is slightly better when first strategy is used.

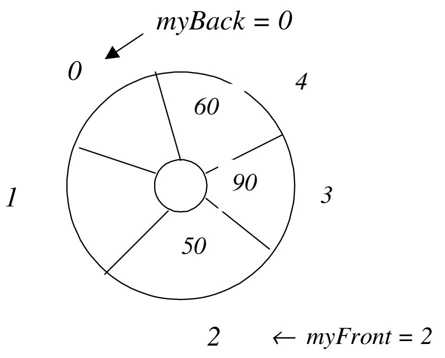
The shifting of array elements is very inefficient and can be avoided by thinking of the array as *circular.*



removeQ
removeQ



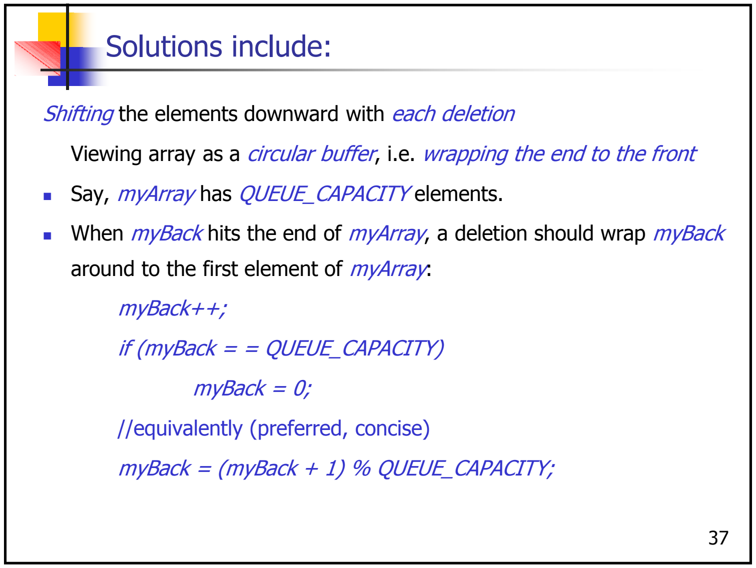
addQ 90
addQ 60



- To determine if a *queue is empty*, we need only check the condition *myFront = = myBack*. The queue constructor will initialize *myFront* and *myBack* both to ***0***.
- To distinguish between an *empty* queue and a *full* queue we maintain *one empty position in the array*. The condition indicating that a queue is *full* then becomes

  (myBack + 1) % QUEUE_CAPACITY == myFront.

- **An array-based implementation would need structures like**
  - *myArray*, an array to store the *elements of the queue*
  - *myFront*, an index to track the *front queue element*
  - *myBack*, an index to track the position *following* last queue element
- **Additions** to the queue would result in incrementing *myBack*.
- **Deletions** from the queue would result in incrementing *myFront*.
- *Clearly, we'd run out of space soon!*



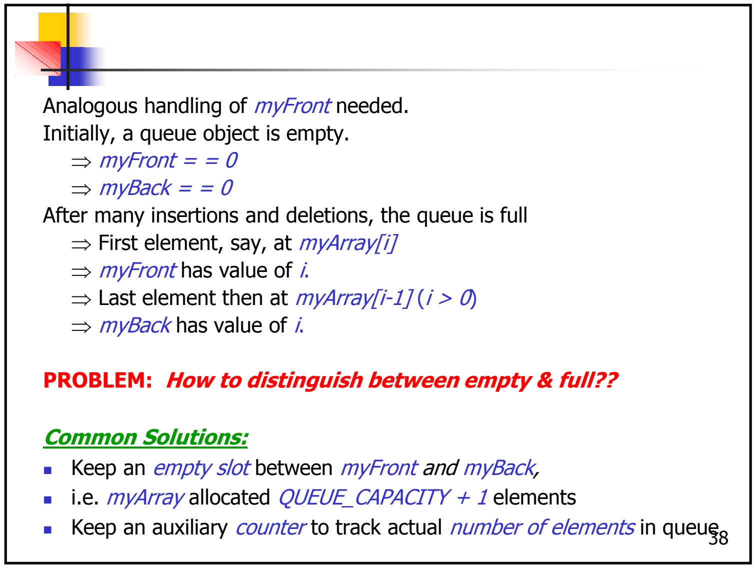
Solutions include:

Shifting the elements downward with *each deletion*

Viewing array as a *circular buffer*, i.e. *wrapping the end to the front*

- Say, *myArray* has *QUEUE_CAPACITY* elements.
- When *myBack* hits the end of *myArray*, a deletion should wrap *myBack* around to the first element of *myArray*:

```
myBack++;
if (myBack = = QUEUE_CAPACITY)
        myBack = 0;
//equivalently (preferred, concise)
myBack = (myBack + 1) % QUEUE_CAPACITY;
```

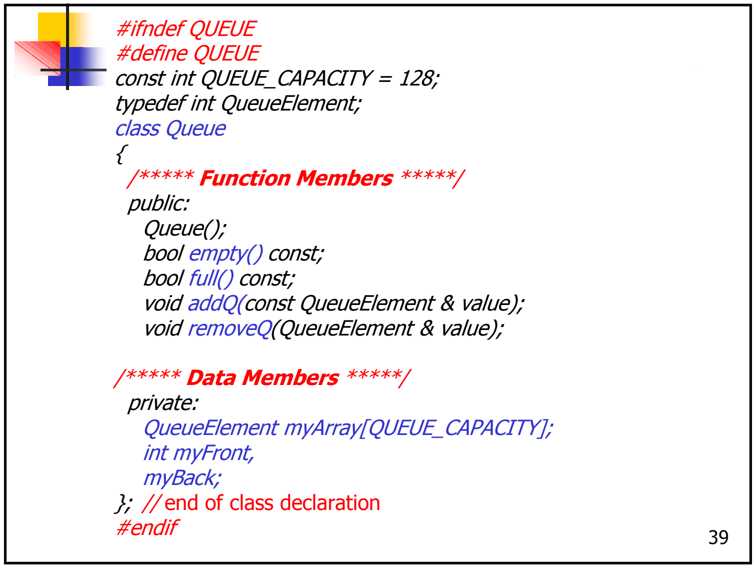


Analogous handling of *myFront* needed.
Initially, a queue object is empty.
⇒ *myFront = = 0*
⇒ *myBack = = 0*
After many insertions and deletions, the queue is full
⇒ First element, say, at *myArray[i]*
⇒ *myFront* has value of *i.*
⇒ Last element then at *myArray[i-1]* (*i > 0*)
⇒ *myBack* has value of *i.*

PROBLEM: ***How to distinguish between empty & full??***

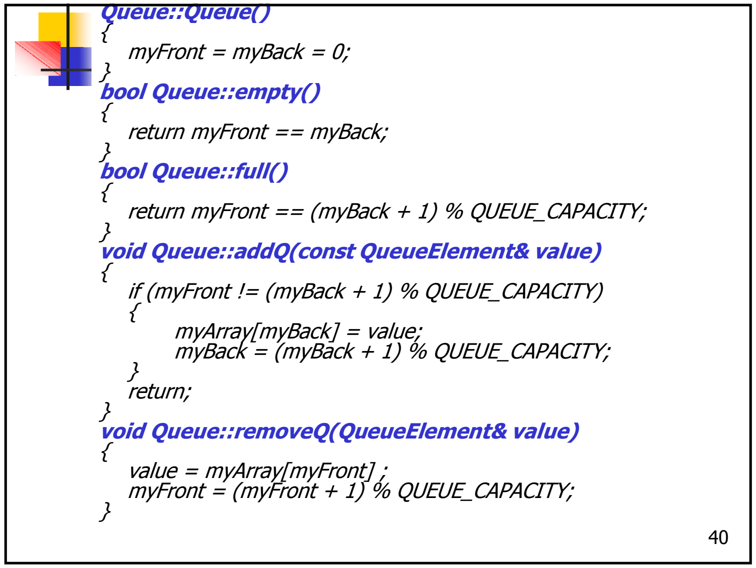
Common Solutions:

- Keep an *empty slot* between *myFront and myBack,*
- i.e. *myArray* allocated *QUEUE_CAPACITY + 1* elements
- Keep an auxiliary *counter* to track actual *number of elements* in queue

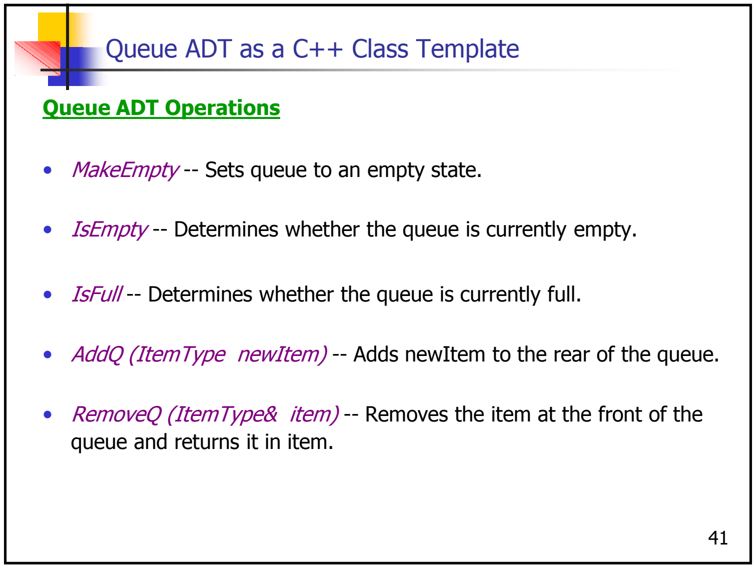


```
#ifndef QUEUE
#define QUEUE
const int QUEUE_CAPACITY = 128;
typedef int QueueElement;
class Queue
{
 /***** Function Members *****/
 public:
   Queue();
   bool empty() const;
   bool full() const;
   void addQ(const QueueElement & value);
   void removeQ(QueueElement & value);

/***** Data Members *****/
 private:
   QueueElement myArray[QUEUE_CAPACITY];
   int myFront,
   myBack;
}; //end of class declaration
#endif
```



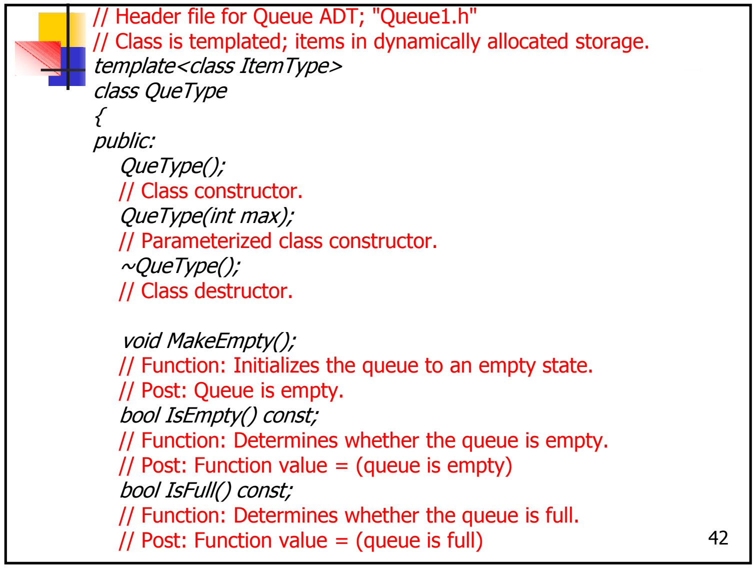
```
Queue::Queue()
{
   myFront = myBack = 0;
}
bool Queue::empty()
{
   return myFront == myBack;
}
bool Queue::full()
{
   return myFront == (myBack + 1) % QUEUE_CAPACITY;
}
void Queue::addQ(const QueueElement& value)
{
   if (myFront != (myBack + 1) % QUEUE_CAPACITY)
   {
        myArray[myBack] = value;
        myBack = (myBack + 1) % QUEUE_CAPACITY;
   }
   return;
}
void Queue::removeQ(QueueElement& value)
{
   value = myArray[myFront] ;
   myFront = (myFront + 1) % QUEUE_CAPACITY;
}
```



Queue ADT as a C++ Class Template

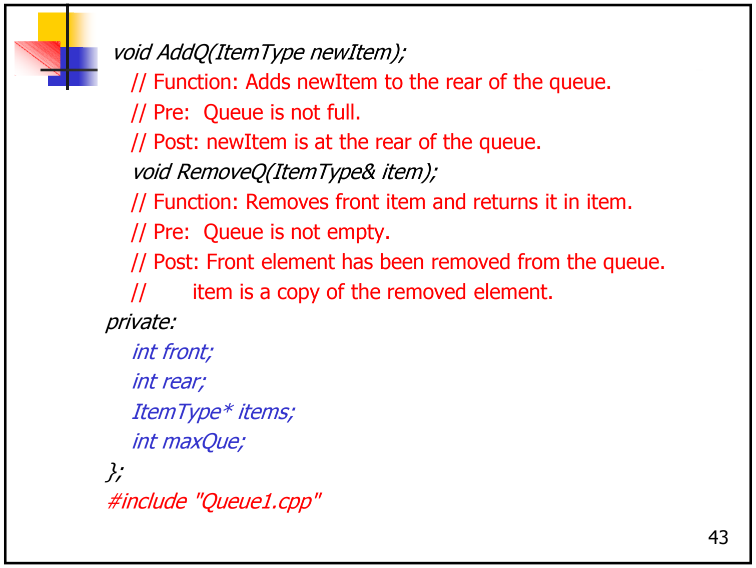
Queue ADT Operations

- *MakeEmpty* -- Sets queue to an empty state.
- *IsEmpty* -- Determines whether the queue is currently empty.
- *IsFull* -- Determines whether the queue is currently full.
- *AddQ (ItemType newItem)* -- Adds newItem to the rear of the queue.
- *RemoveQ (ItemType& item)* -- Removes the item at the front of the queue and returns it in item.

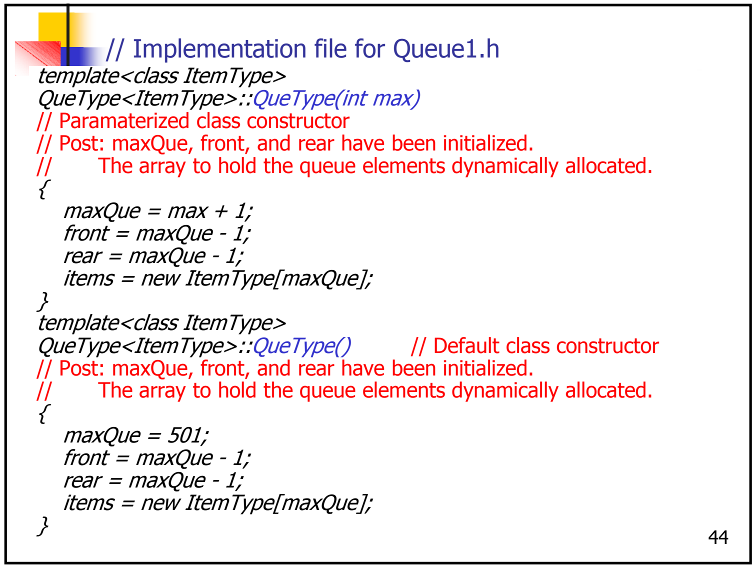


```
// Header file for Queue ADT; "Queue1.h"
// Class is templated; items in dynamically allocated storage.
template<class ItemType>
class QueType
{
public:
    QueType();
    // Class constructor.
    QueType(int max);
    // Parameterized class constructor.
    ~QueType();
    // Class destructor.

    void MakeEmpty();
    // Function: Initializes the queue to an empty state.
    // Post: Queue is empty.
    bool IsEmpty() const;
    // Function: Determines whether the queue is empty.
    // Post: Function value = (queue is empty)
    bool IsFull() const;
    // Function: Determines whether the queue is full.
    // Post: Function value = (queue is full)
```



```
    void AddQ(ItemType newItem);
      // Function: Adds newItem to the rear of the queue.
      // Pre:  Queue is not full.
      // Post: newItem is at the rear of the queue.
      void RemoveQ(ItemType& item);
      // Function: Removes front item and returns it in item.
      // Pre:  Queue is not empty.
      // Post: Front element has been removed from the queue.
      //       item is a copy of the removed element.
    private:
      int front;
      int rear;
      ItemType* items;
      int maxQue;
    };
    #include "Queue1.cpp"
```



// Implementation file for Queue1.h

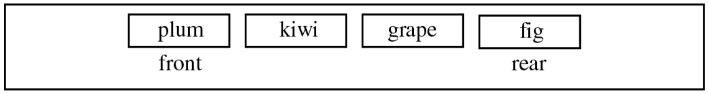
```
template<class ItemType>
QueType<ItemType>::QueType(int max)
// Paramaterized class constructor
// Post: maxQue, front, and rear have been initialized.
//       The array to hold the queue elements dynamically allocated.
{
   maxQue = max + 1;
   front = maxQue - 1;
   rear = maxQue - 1;
   items = new ItemType[maxQue];
}
template<class ItemType>
QueType<ItemType>::QueType()        // Default class constructor
// Post: maxQue, front, and rear have been initialized.
//       The array to hold the queue elements dynamically allocated.
{
   maxQue = 501;
   front = maxQue - 1;
   rear = maxQue - 1;
   items = new ItemType[maxQue];
}
```

```
template<class ItemType>
QueType<ItemType>::~QueType()        // Class destructor
{
   delete [] items;
}
template<class ItemType>
void QueType<ItemType>::MakeEmpty()
// Post: front and rear have been reset to the empty state.
{
   front = maxQue - 1;
   rear = maxQue - 1;
}
template<class ItemType>
bool QueType<ItemType>::IsEmpty() const
// Returns true if the queue is empty; false otherwise.
{
   return (rear == front);
}
```

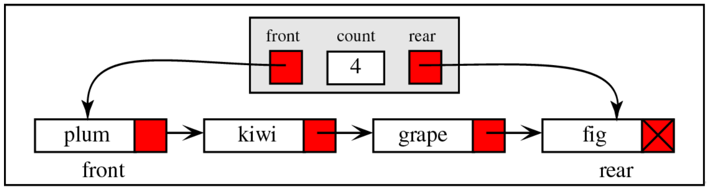
```
template<class ItemType>
bool QueType<ItemType>::IsFull() const
// Returns true if the queue is full; false otherwise.
{
   return ((rear + 1) % maxQue == front);
}
template<class ItemType>
void QueType<ItemType>::AddQ(ItemType newItem)
// Post: newItem is at the rear of the queue.
{
   rear = (rear +1) % maxQue;
   items[rear] = newItem;
}
template<class ItemType>
void QueType<ItemType>::RemoveQ(ItemType& item)
// Post: The front of the queue removed and a copy returned in item.
{
   front = (front + 1) % maxQue;
   item = items[front];
}
```

Implementing a Queue as a Linked Structure

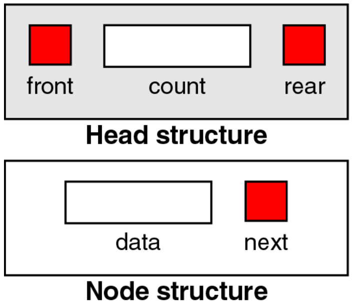
- In an *array-based implementation* of a queue, we keep track of *two indexes* that point to the ***front*** and ***rear*** boundaries of the data in the queue.
- In a *linked* representation we can use *two pointers*, *qFront* and *qRear*, to mark the front and the rear of the queue.



(a) Conceptual queue

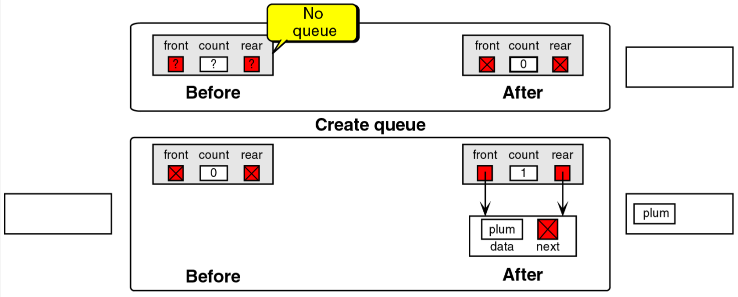


(b) Physical queue



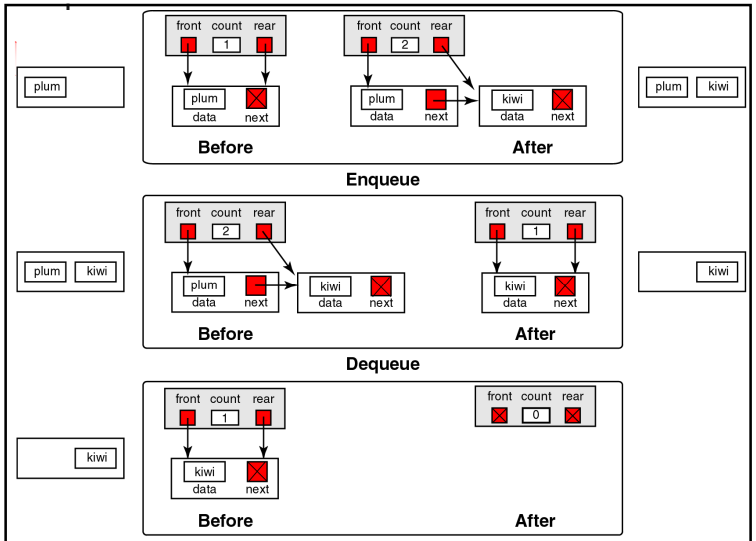
```
queueHead
 front    <node pointer>
 count    <integer>
 rear     <node pointer>
end queueHead

node
  data    <dataType>
  next    <node pointer>
end node
```



Create queue

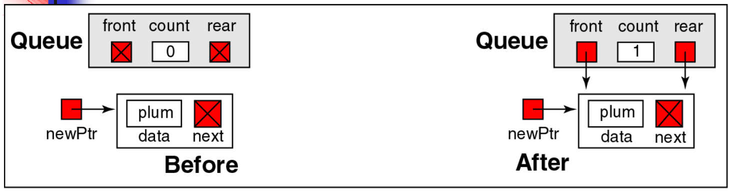
Enqueue



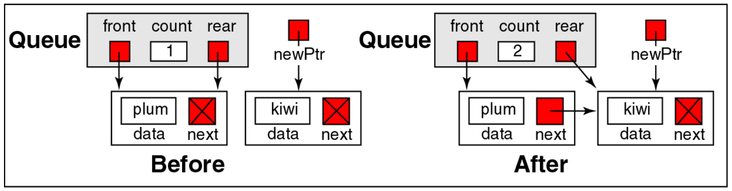
Enqueue

Dequeue

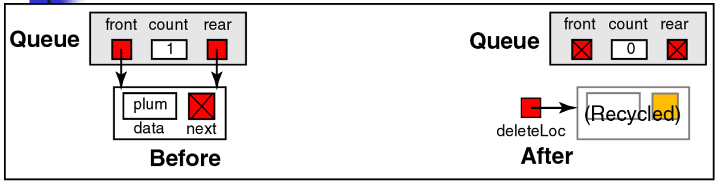
Destroy queue



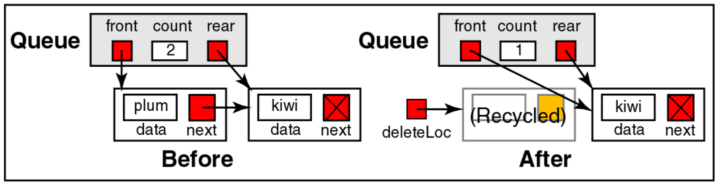
(a) Case 1: insert into null queue



(b) Case 2: insert into queue with data

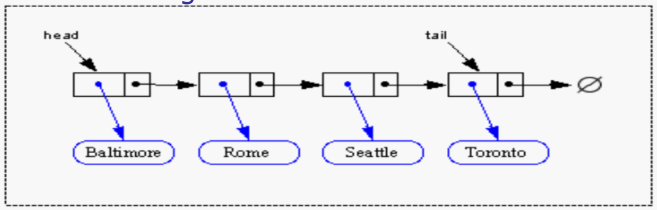


(a) Case 1: delete only item in queue

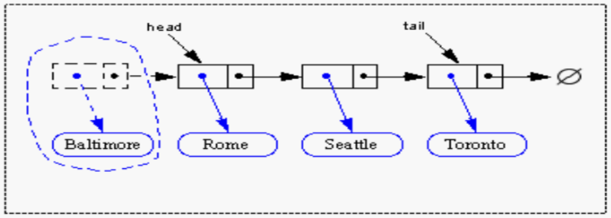


(b) Case 2: delete item at front of queue

Removing at the Head



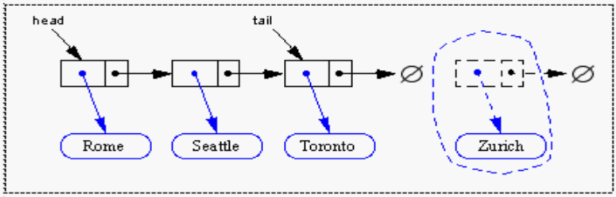
- advance head reference



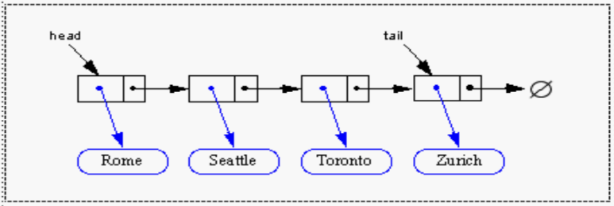
- inserting at the head is just as easy

Inserting at the Tail

- create a new node

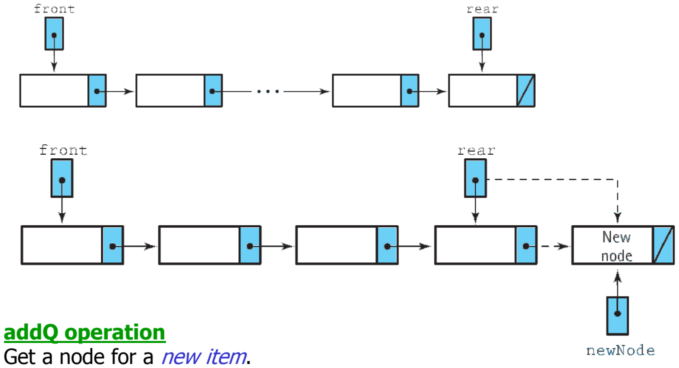


- chain it and move the tail reference



- how about removing at the tail?

Because we add new elements to the queue by inserting after the last node, we need a new *addQ (Enqueue)* algorithm.



addQ operation

Get a node for a *new item*.

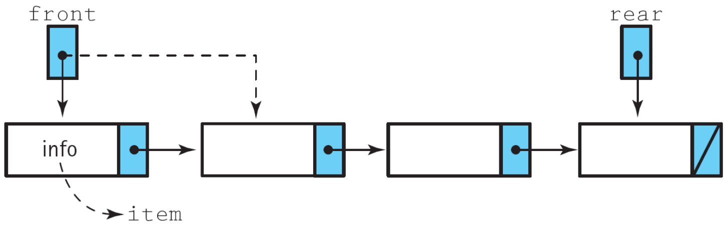
Insert the new node at the *rear* of the queue.

Update pointer to the rear of the queue.

```
// Get a node for the new item
    Set newNode to the address of a newly allocated node.
    Set Info(newNode) to newItem.
    Set Next(newNode) to NULL.
// Insert the new node at the rear of the queue
    Set Next(qRear) to newNode.
// Insert the new node at the rear of the queue
    IF the queue is empty
        Set qFront  to newNode
    ELSE
        Set Next(qRear)  to newNode
//  Set qRear
    Set qRear to newNode
```

removeQ operation

- If *qFront* is NULL after we have deleted the front node, we know that the queue is now empty. *qRear* must be also set to NULL .



removeQ

- Set *tempPtr* to *qFront* // Save it for deallocating
- Set *item* to *Info(qFront)*
- Set *qFront* to *Next(qFront)*

```
      IF queue is now empty
            Set qRear to NULL
      Deallocate Node(tempPtr)
```

- How do we know when the queue is empty? Both *qFront* and *qRear* should then be NULL pointers.
- What about function *IsFull?* We can use the same *IsFull* we wrote for the **Stack ADT**.

// File Queue2.h: Header file for Queue ADT.

```
// Class is templated.
template <class ItemType>
struct NodeType;
template <class ItemType>
class QueType
{
  public:
    QueType();
    // Class constructor.
   ~QueType();
    // Class destructor.
    void MakeEmpty();
    // Function: Initializes the queue to an empty state.
    // Post: Queue is empty.
```

```
 bool IsEmpty() const;
   // Function: Determines whether the queue is empty.
   // Post: Function value = (queue is empty)
   bool IsFull() const;
   // Function: Determines whether the queue is full.
   // Post: Function value = (queue is full)
   void AddQ(ItemType newItem);
   // Function: Adds newItem to the rear of the queue.
   // Pre:  Queue is not full.
   // Post: newItem is at the rear of the queue.
   void RemoveQ(ItemType& item);
   // Function: Removes front item and return it in item.
   // Pre:  Queue is not empty.
   // Post: Front element has been removed from the queue.
   //       item is a copy of the removed element
private:
   NodeType<ItemType>* qFront;
   NodeType<ItemType>* qRear;
};
```

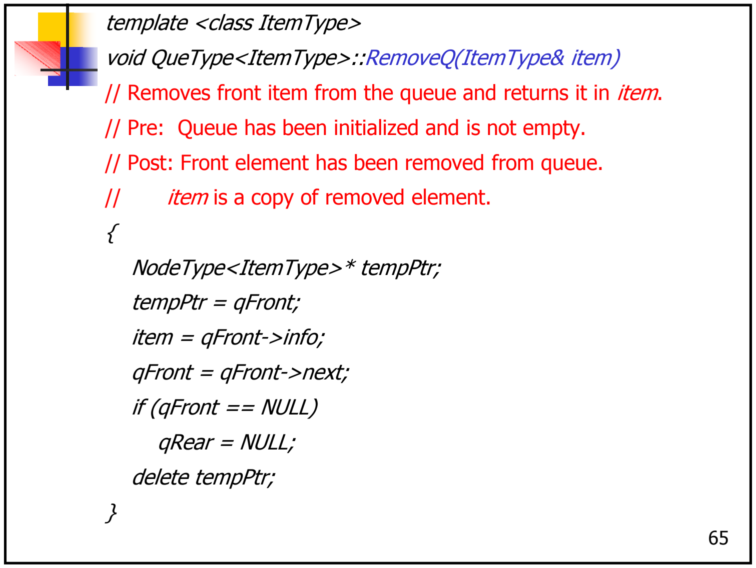
// Implementation file for Queue ADT

```
template <class ItemType>
struct NodeType
{
    ItemType info;
    NodeType* next;
};
template <class ItemType>
QueType<ItemType>::QueType()                    // Class constructor.
// Post:  qFront and qRear are set to NULL.
{
    qFront = NULL;
    qRear = NULL;
}
```

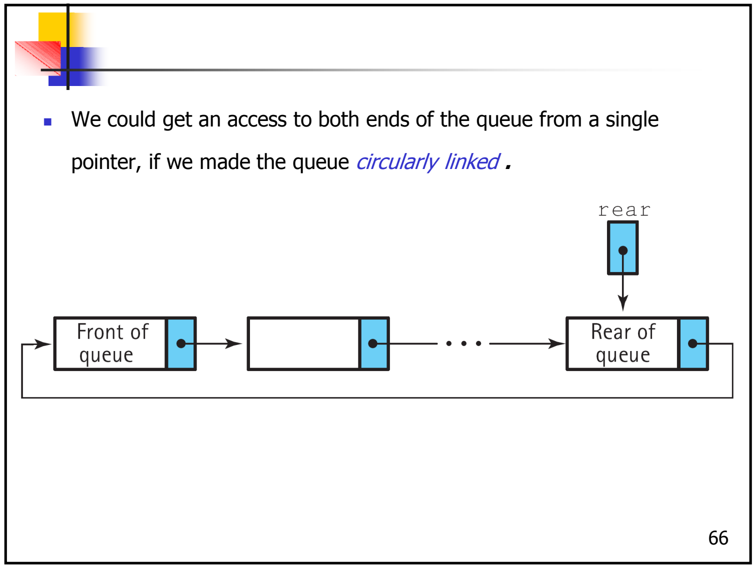
```
template <class ItemType>
void QueType<ItemType>::MakeEmpty()
// Post: Queue is empty; all elements have been deallocated.
{
    NodeType<ItemType>* tempPtr;
    while (qFront != NULL)
    {
        tempPtr = qFront;
        qFront = qFront->next;
        delete tempPtr;
    }
    qRear = NULL;
}
template <class ItemType>                   // Class destructor.
QueType<ItemType>::~QueType()
{
    MakeEmpty();
}
```

```
template <class ItemType>
bool QueType<ItemType>::IsFull() const
// Returns true if no room for another ItemType on the free store;
// false otherwise.
{
    NodeType<ItemType>* ptr;
    ptr = new NodeType<ItemType>;
    if (ptr == NULL)
        return true;
    else
    {
        delete ptr;
        return false;
    }
}
template <class ItemType>
bool QueType<ItemType>::IsEmpty() const
// Returns true if there are no elements on the queue; false otherwise.
{
    return (qFront == NULL);
}
```

```
template <class ItemType>
void QueType<ItemType>::AddQ(ItemType newItem)
// Adds newItem to the rear of the queue.
// Pre:  Queue has been initialized and is not full.
// Post: newItem is at rear of queue.
{
    NodeType<ItemType>* newNode;
    newNode = new NodeType<ItemType>;
    newNode->info = newItem;
    newNode->next = NULL;
    if (qRear == NULL)
        qFront = newNode;
    else
        qRear->next = newNode;
    qRear = newNode;
}
```



```
template <class ItemType>
void QueType<ItemType>::RemoveQ(ItemType& item)
// Removes front item from the queue and returns it in item.
// Pre:  Queue has been initialized and is not empty.
// Post: Front element has been removed from queue.
//       item is a copy of removed element.
{
    NodeType<ItemType>* tempPtr;
    tempPtr = qFront;
    item = qFront->info;
    qFront = qFront->next;
    if (qFront == NULL)
        qRear = NULL;
    delete tempPtr;
}
```



- We could get an access to both ends of the queue from a single pointer, if we made the queue *circularly linked* **.**