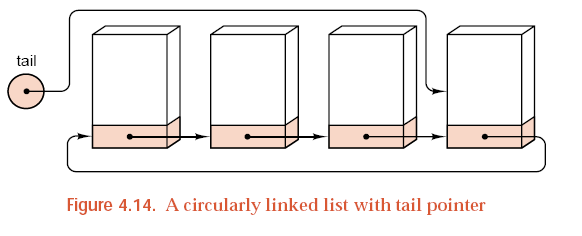
****

**E5.** *A* ***circularly linked list****, illustrated in Figure 4.14, is a linked list in which the node at the tail of the list, instead of having a* NULL *pointer, points back to the node at the head of the list. We then need only one pointer* tail *to access both ends of the list, since we know that* tail->next *points back to the head of the list.*

1. *If we implement a queue as a circularly linked list, then we need only one pointer* tail *(or* rear*) to locate both the front and the rear. Write the methods needed to process a queue stored in this way.*

*Answer* The definition for the **class** Queue is as follows.

**class** Queue {

**public:**

**//** *standard* Queue *methods*

Queue( )**;**

**bool** empty( ) **const;**

Error\_code append(**const** Queue\_entry &item)**;**

Error\_code serve( )**;**

Error\_code retrieve(Queue\_entry &item) **const;**

**//** *safety features for linked structures*

*~*Queue( )**;**

Queue(**const** Queue &original)**;**

**void operator** = (**const** Queue &original)**;**

**protected:**

Node \*tail**;**

}**;**

The implementations of the methods follow.

**bool** Queue **::** empty( ) **const**

**/**\* **Post:** *Return* **true** *if the* Queue *is empty, otherwise return* **false***.* \***/**

{

**return** tail == NULL**;**

}

Queue **::** Queue( )

**/**\* **Post:** *The* Queue *is initialized to be empty.* \***/**

{

tail = NULL**;//错误：初始化时将tail->next=tail**

}

Error\_code Queue **::** append(**const** Queue\_entry &item)

**/**\* **Post:** *Add* item *to the rear of the* Queue *and return a code of* success *or return a code of*

overflow *if dynamic memory is exhausted.* \***/**

{

Node \*new\_rear = **new** Node(item)**;**

**if** (new\_rear == NULL) **return** overflow**;**

**if** (tail == NULL) {

tail = new\_rear**;**

tail ->next = tail**;**

}

**else** {

new\_rear ->next = tail ->next**;**

tail->next = new\_rear**;**

tail = new\_rear**;**

}

**return** success**;**

}

Error\_code Queue **::** retrieve(Queue\_entry &item) **const**

**/**\* **Post:** *The front of the* Queue *is reported in* item*. If the* Queue *is empty return an* Error\_code

*of* underflow *and leave the* Queue *unchanged.* \***/**

{

**if** (tail == NULL) **return** underflow**;**

item = (tail->next)->entry**;**

**return** success**;**

}

Error\_code Queue **::** serve( )

**/**\* **Post:** *The front of the* Queue *is removed. If the* Queue *is empty, return an* Error\_code *of* underflow*.* \***/**

{

**if** (tail == NULL) **return** underflow**;**

Node \*old\_front = tail->next**;**

**if** (tail == old\_front) tail = NULL**;**

**else** tail->next = old\_front->next**;**

**delete** old\_front**;**

**return** success**;**

}

Queue **::~**Queue( )

{

**while** (!empty( ))

serve( )**;**

}

Queue **::** Queue(**const** Queue &copy)

{

tail = NULL**;**

**if** (copy**.**tail == NULL) **return;**

Node \*copy\_node = (copy**.**tail)->next**;**

**do** {

append(copy\_node->entry)**;**

copy\_node = copy\_node->next**;**

} **while** (copy\_node != (copy**.**tail)->next)**;**

}

**void** Queue **:: operator** = (**const** Queue &copy)

{

**while** (!empty( ))

serve( )**;**

tail = NULL**;**

**if** (copy**.**tail == NULL) **return;**

Node \*copy\_node = (copy**.**tail)->next**;**

**do** {

append(copy\_node->entry)**;**

copy\_node = copy\_node->next**;**

} **while** (copy\_node != (copy**.**tail)->next)**;**

}

**(b)** *What are the disadvantages of implementing this structure, as opposed to using the version requiring two pointers?*

*只用一个指针，省略了队头这个指针，当出队时，需要从tail指针向后查找定位至队头，算法实现中由于没有充分体现队头队尾的属性，实现逻辑没有原来清楚。*

*总的来说，用空间的这点节约换来的是程序处理逻辑不清晰，并不值得。*