Worksheet 18: Linked List Queue, pointer to Tail

**In Preparation:** Read Chapters 6 and 7 to learn more about the Stack and Queue data types. If you have not done so already, you should do Lesson 17 to learn about the basic features of a linked list.

One problem with the linked list stack is that it only permits rapid access to the front of the collection. This was no problem for the stack operations described in Lesson 17, but would present a difficulty if we tried to implement a queue, where elements were inserted at one end and removed from the other.

A solution to this problem is to maintain two references into the collection of links. As before, the data field firstLink refers to the first link in the sequence. But now a second data field named lastLink will refer to the final link in the sequence:

![Diagram of linked list structure with firstLink and lastLink fields]

To reduce the size of the image the fields in the links have been omitted, replaced with the graphic arrow. Both the firstLink and lastLink fields can be **null** if a list is empty. Removing a data field from the front is the same as before, with the additional requirement that when the last value is removed and the list becomes empty the variable lastLink must also be set to null.

![Diagram of linked list with last value removed and lastLink set to null]

Because we have a link to the back, it is easy to add a new value to the end of the list.

![Diagram of linked list with new value added at the end]
Worksheet 18: Linked list Queue  Name:

We will add another variation to our container. A **sentinel** is a special link, one that does not contain a value. The sentinel is used to mark either the beginning or end of a chain of links. In our case we will use a sentinel at the front. This is sometimes termed a list header. The presence of the sentinel makes it easier to address special cases. For example, the list of links is never actually empty, even when it is logically empty, since there is always at least one link. A new value is inserted after the end.

![Sentinel Diagram]

Values are removed from the front, as with the stack. But because of the sentinel, the element removed will be the element right after the sentinel. Make sure that when you remove a value from the collection you free the memory for the associated link.

You should complete the following skeleton code for the ListQueue. The structures have been written for you, as well as the initialization routine. The function isEmpty must determine whether or not the collection has any elements. What property characterizes an empty queue?

**On Your Own**

1. Draw a picture showing the values of the various data fields in an instance of ListQueue when it is first created.

2. Draw a picture showing what it looks like after one element has been inserted.

3. Based on the previous two drawings, can you determine what feature you can use to tell if a list is empty?

4. Draw a picture showing what it looks like after two elements have been inserted.

5. What is the algorithmic complexity of each of the queue operations?

6. How difficult would it be to write the method addFront(newValue) that inserts a new element into the front of the collection? A container that supports adding values at either and, but removal from only one side, is sometimes termed a **scroll**.

7. Explain why removing the value from the back would be difficult for this container. What would be the algorithmic complexity of the removeLast operation?
struct link {
    EleType value;
    struct link * next;
};

struct listQueue {
    struct slink *firstLink;
    struct slink *lastLink;
};

void ListQueueInit (struct listQueue *q) {
    struct slink *lnk = (struct slink *) malloc(sizeof(struct slink));
    assert(lnk != 0); /* lnk is the sentinel */
    lnk->next = 0;
    q->firstLink = q->lastLink = lnk;
}

void listQueueAddBack (struct listQueue *q, EleType e) {

}  

EleType listQueueFront (struct listQueue *q) {

}

void listQueueRemoveFront (struct listQueue *q) {

}

int listQueueIsEmpty (struct listQueue *q) {

}