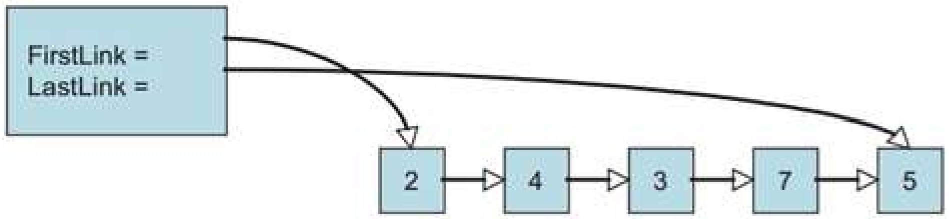
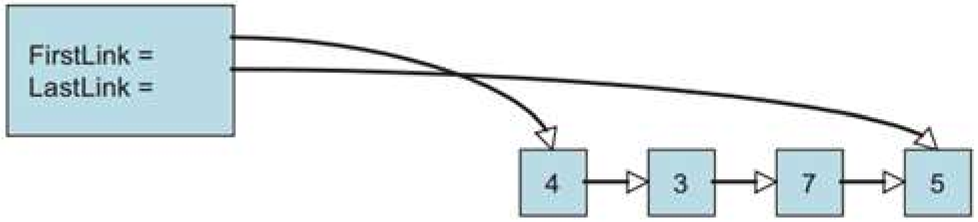
Worksheet 18: Linked List Queue, pointer to Tail

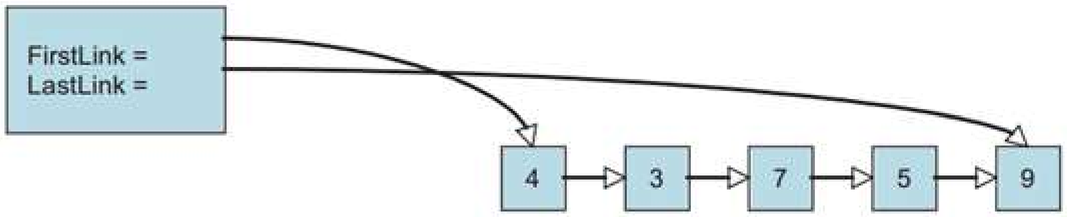
In Preparation: Read Chapters 6 and 7 to learn more about the Stack and Queue data types. If you have not done so already, you should do Worksheet 17 to learn about the basic features of a linked list.
One problem with the linked list *stack* is that it only permits rapid access to the front of the collection. This was no problem for the stack operations described in Worksheet 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:



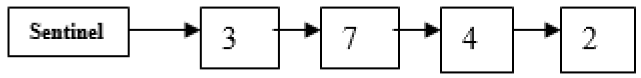
To reduce the size of the image the fields in the links have been omitted, replaced with the graphic arrow. Both the firstLink and lastLink fields can be **null** if a list is empty. Removing a data field from the front is the same as before, with the additional requirement that when the last value is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



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

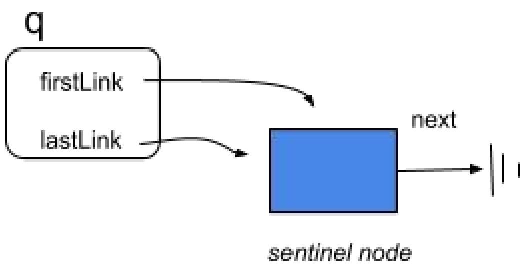


We will add another variation to our container. A *sentinel* is a special link, one that does not contain a value. The sentinel is used to mark either the beginning or end of a chain of links. In our case we will use a sentinel at the front. This is sometimes termed a list header. The presence of the 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.

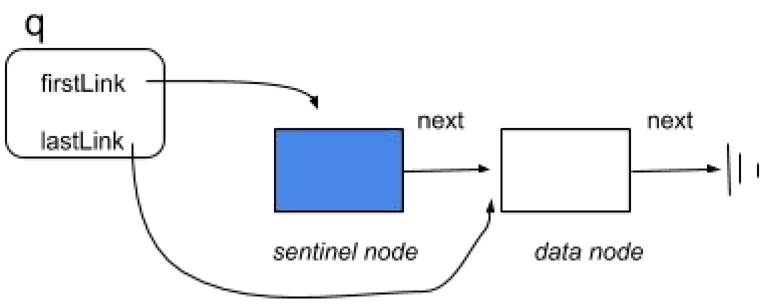


Values are removed from the front, as with the stack. But because of the sentinel, the element removed will be the element right after the sentinel. Make sure that when you remove a value from the collection you free the memory for the associated link.
You should complete the following skeleton code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

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



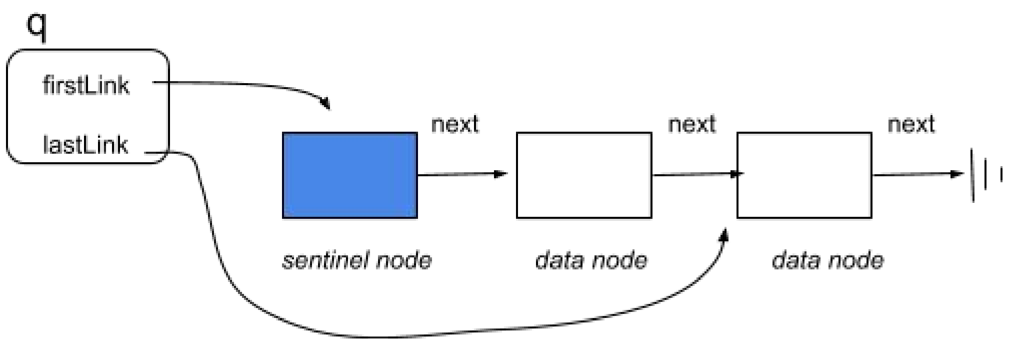
2. Draw a picture showing what it looks like after one element has been inserted.



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

Without a sentinel, we can use whether the first link pointer points to null.
With a sentinel, we can use whether the sentinel's 'next' member points to null.

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?

It is always constant (O(1)).

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.

It would be quite simple. A new link would be created, it would be made the first data node, and its 'next' attribute would point to the former first data node. It would be the same procedure as push for the stack.

Pseudocode:

```
newLnk = malloc(...)
newLnk->next = firstLink->next;
firstLink->next = newLnk;
```

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?

   If we remove the last node we would need to set the "next" pointer of the previous node to NULL. In order to do that we need to be able to address the previous-to-last node. There is no direct way to get that address other than traversing the entire queue from beginning to almost end. The complexity of finding the addressof the previous-to-last node would be O(n), because you would have to traversethe entire list, 'asking' each link for the address of the next element, up until the previous-to-last node.

```
struct link {
  TYPE value;
  struct link * next;
};
struct listQueue {
  struct link *firstLink;
  struct link *lastLink;
};
void listQueueInit (struct listQueue *q) {
  struct link *lnk = (struct link *) malloc(sizeof(struct link));
  assert(lnk != 0); /* lnk is the sentinel */
  lnk->next = 0;
  q->firstLink = q->lastLink = lnk;
}
void listQueueAddBack (struct listQueue *q, TYPE e) {
        struct link * newLink = (struct link *) malloc(sizeof(struct link));
        assert (newLink != 0);

        q->lastLink->next = newLink;
        q->lastLink = newLink;

        newLink->value = e;
        newLink->next = NULL;
}
TYPE listQueueFront (struct listQueue *q) {
        assert (q->firstLink != q->lastLink);
        return q->firstLink->next->value;
}
void listQueueRemoveFront (struct listQueue *q) {
        assert(q->firstLink != q->lastLink);
        struct link* temp = q->firstLink->next;

        if (q->firstLink->next == q->lastLink){
                q->lastLink = q->firstLink;
                q->firstLink->next = NULL;
        }else{
                q->firstLink->next = q->firstLink->next->next;
        }

        free(temp);
        temp = NULL;




}
```

int listQueueIsEmpty (struct listQueue *q) {

```
        if (q->firstLink == q->lastLink){
                return 1;
        }else{
                return 0;
        }
}
```

Function Test Results

```
Create and Initialize Queue

Queue is Empty: TRUE

Add to Back: 1
Queue Contents = 1
Add to Back: 2
Queue Contents = 1 2
Add to Back: 3
Queue Contents = 1 2 3
Add to Back: 4
Queue Contents = 1 2 3 4
Add to Back: 5
Queue Contents = 1 2 3 4 5

Remove Front: 1
Queue Contents = 2 3 4 5

Remove Front: 2
Queue Contents = 3 4 5

Remove Front: 3
Queue Contents = 4 5

Remove Front: 4
Queue Contents = 5

Remove Front: 5
Queue Contents =
Queue is Empty: TRUE
```