Worksheet 18 - Group 1

Worksheet Group 1 Members

Marc Clinedinst: clinedim@onid.oregonstate.edu
Kelby Faessler: faesslek@onid.oregonstate.edu
James Fitzwater: fitzwatj@onid.oregonstate.edu
Tom Gariepy: gariepyt@onid.oregonstate.edu
Sean Reilly: reillys@onid.oregonstate.edu
Joseph Struth: struthj@onid.oregonstate.edu

Worksheet 18: Linked List Queue

In this worksheet, we implement four functions that define the behavior of the queue data structure when it is implemented on top of a linked list. More specifically, we implement the `listQueueAddBack`, `listQueueFront`, `listQueueRemoveFront`, and `listQueueIsEmpty` functions. The code for these functions is provided below, with accompanying comments where extra explanation is necessary.

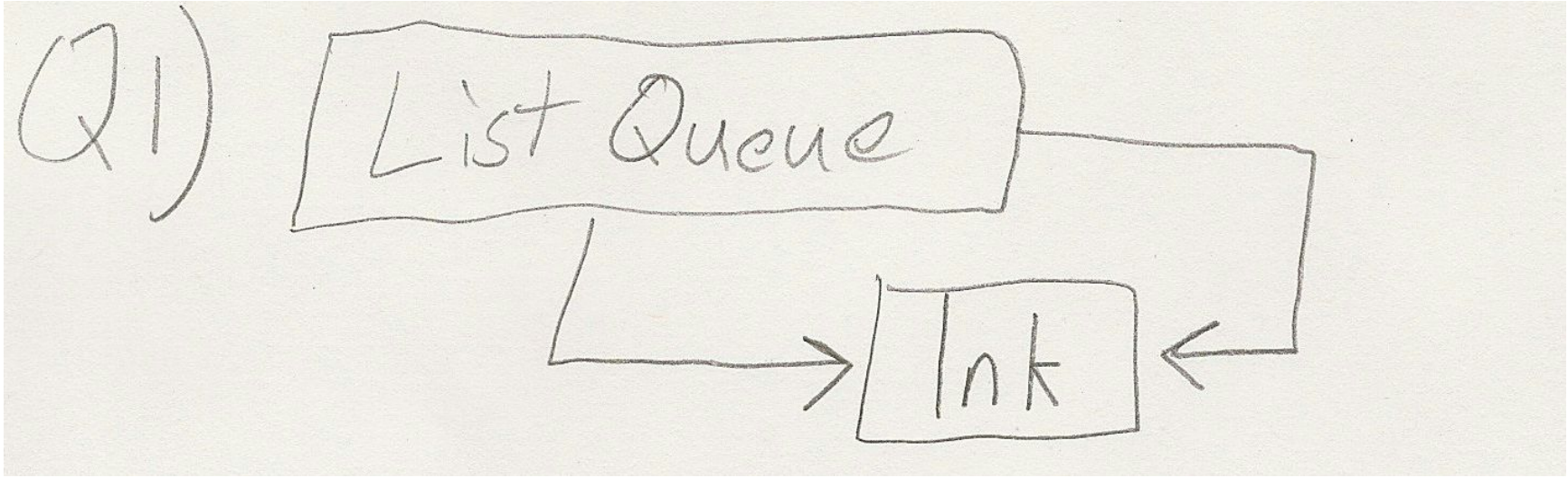
```
/*
     This function adds an element to the back of the queue.  The function first allocates
     memory for a new link and checks that the allocation was successful.  The function then
     sets the value to that which was passed to the function.  It then sets the the new link's
     next pointer to the value of the current last link's next pointer.  Next, the function
     sets the last link's next pointer to the new link.  The function finally sets the last
     link to the newly allocated link.
*/
void listQueueAddBack(struct listQueue *q, TYPE e) {
     struct link *lnk = (struct link *) malloc(sizeof(struct link));
     assert(lnk != 0);
     lnk->value = e;
     lnk->next = q->lastLink->next;
```

```
	q->lastLink->next = lnk;
	q->lastLink = lnk;
}


/*
	This function returns the value at the front of the queue.  Given that this function
	should not be called on an empty queue, it first checks to make sure that the queue is
	not empty.  If the queue is empty, then an assertion error will occur.  If the queue is
	not empty, then the function will return the value at the front of the queue.
*/
TYPE listQueueFront(struct listQueue *q) {
	assert(!listQueueIsEmpty(q));
	return q->firstLink->next->value;
}

/*
	This function removes the value at the front of the queue.  Given that this function
	should not be called on an empty queue, it first checks to make sure that the queue is
	not empty.  If the queue is empty, then an assertion error will occur.  If the queue is
	not empty, then the function creates a new pointer to the first value in the queue called
	garbage. It then adjusts the first value to be the next value.  Finally, it frees the
	memory that was allocated for the garbage pointer.
*/
void listQueueRemoveFront(struct listQueue *q) {
	assert(!listQueueIsEmpty(q));
	struct link *garbage = q->firstLink->next;
	q->firstLink->next = q->firstLink->next->next;
	free(garbage);
}
```

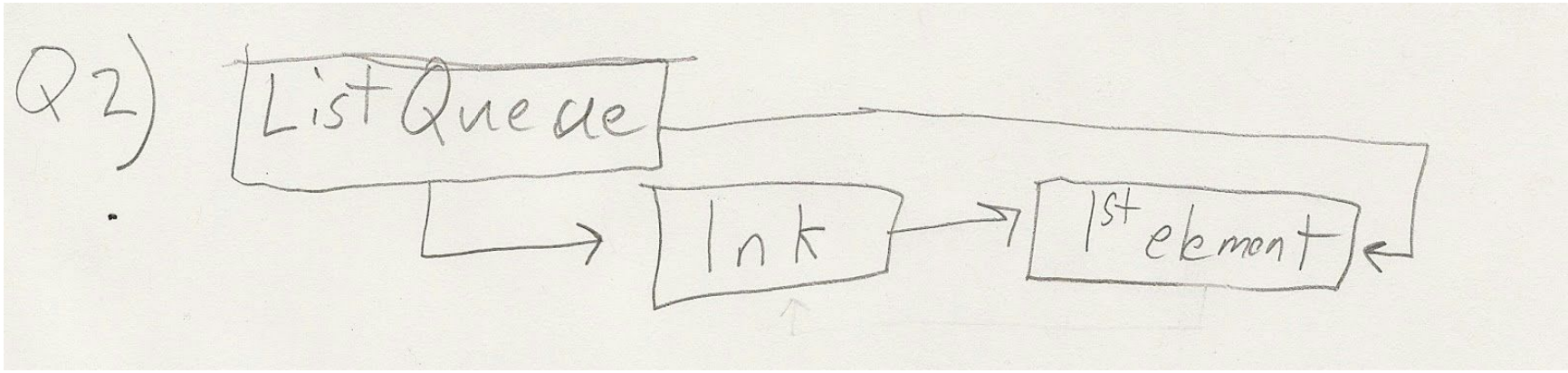
```
/*
      This function returns an integer which represents whether or not the queue is empty.
      It does so by comparing the value stored in q->firstLink->next.  If this value is equal
      to 0, then this indicates that the queue is empty; the value 1 will be returned.
      Otherwise, the value 0 will be returned, indicating that the queue is not empty.
*/
int listQueueIsEmpty(struct listQueue *q) {
      return q->firstLink->next == 0;
}
```

In addition to the functions above, the worksheet also asked us to respond to seven different questions. These are listed below and are accompanied by our group's answers.

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



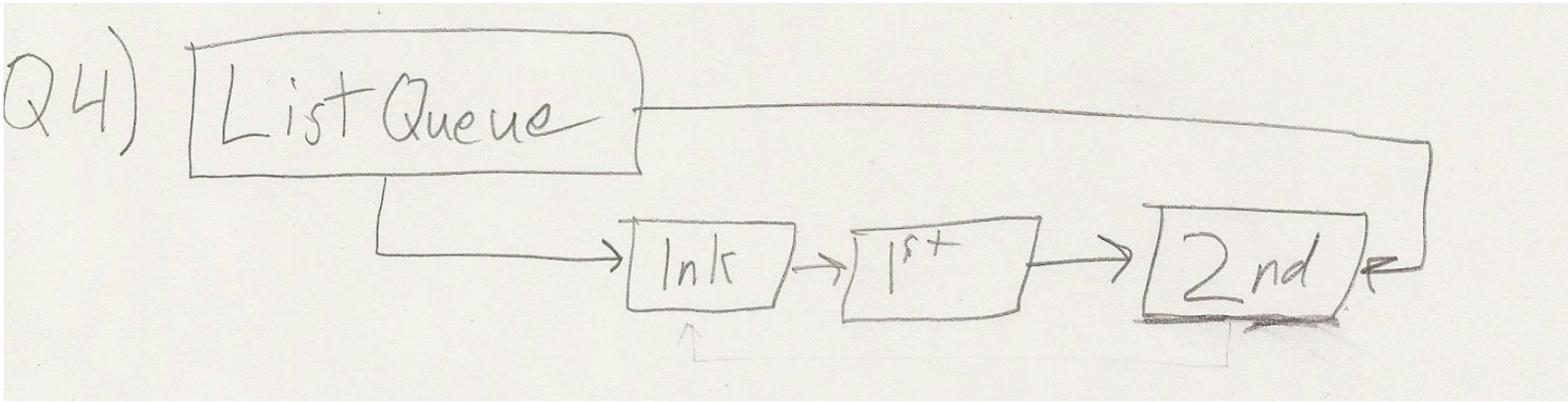
2. Draw a picture showing what it looks like after one element has been inserted.



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

Based on the previous two drawings, we can tell when the list is empty by looking at what the first link is pointing to zero. If it is pointing to another element, it is not empty. If it is pointing to nothing, then it is an empty object.

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?

Operation	Complexity
listQueueInit	O(1)
listQueueAddBack	O(1)
listQueueFront	O(1)
listQueueRemoveFront	O(1)
listQueueIsEmpty	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.

This doesn't sound like a difficult task to do. All you would need to do is create a new link, set that link's next pointer equal to the header's next pointer, then set the header's next pointer to point to the new link that you've created.

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?

It is difficult to remove a value from the back because the link structs don't point to the previous values. Because of this, we would have to search through the queue starting from the front to the back to find the one with the next pointer to the last value.

Piazza Discussion

https://piazza.com/class/ib2kus4hsie528?cid=118