Worksheet 19 - 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 19: Linked List Deque

In this worksheet, we implement four functions that define the behavior of the dque data structure when it is implemented on top of a linked list. More specifically, we implement the _addBefore, _removeLink, LinkedListFront, and LinkedListBack functions. The code for these functions is provided below, with accompanying comments where extra explanation is necessary.

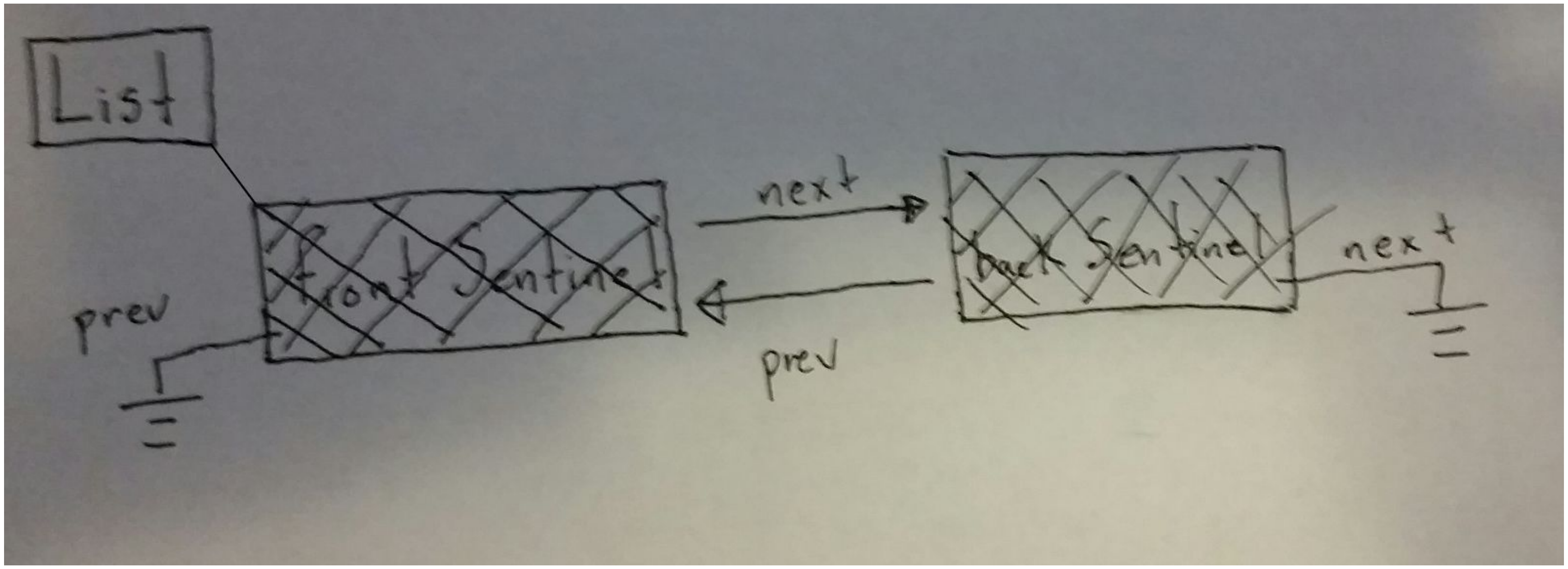
```
/*
     This function adds a new link before a particular link that is already in the
     deque.  The first two steps completed in this function are that of allocating memory for
     the new link and checking to make sure that the allocation was successful.  Once the new
     link has been created, the function sets the value of the new link's value field. The
     function then adjusts the pointers for the new link and existing link so that the new
     link appears before the particular link that was already in the deque.  Finally, the size
     of the deque is incremented.
*/
void _addBefore (struct linkedList *q, struct dlink *lnk, TYPE e) {
     struct dlink *newLink = (struct dlink *)malloc(sizeof(struct dlink));
     assert(newLink != 0);
     newLink->value = e;
     newLink->next = lnk;
```

```
	newLink->prev = lnk->prev;
	lnk->prev->next = newLink;
	lnk->prev = newLink;
	q->size++;
}

/*
	This function removes a link from the deque.  It begins by marking as garbage the link
	that is to be deleted.  It then adjusts the links for the link that appears prior to
	garbage and after garbage point to one another.  It then frees the memory that was
	allocated for garbage.  Finally, the size of garbage is decremented.
*/
void _removeLink(struct linkedList *q, struct dlink *lnk) {
	struct dlink *garbage = lnk;
	lnk->prev->next = lnk->next;
	lnk->next->prev = lnk->prev;
	free(garbage);
	q->size--;
}

/*
	This function returns the value that appears at the front of the deque.  Given that this
	function should not be called on an empty deque, the function first checks that the deque
	is not empty.  If the deque is empty, then an assertion error occurs.  Otherwise, the
	value at the front of the deque is returned.
*/
TYPE LinkedListFront(struct linkedList *q) {
	assert(!LinkedListIsEmpty(q));
	return q->frontSentinel->next->value;
}
```

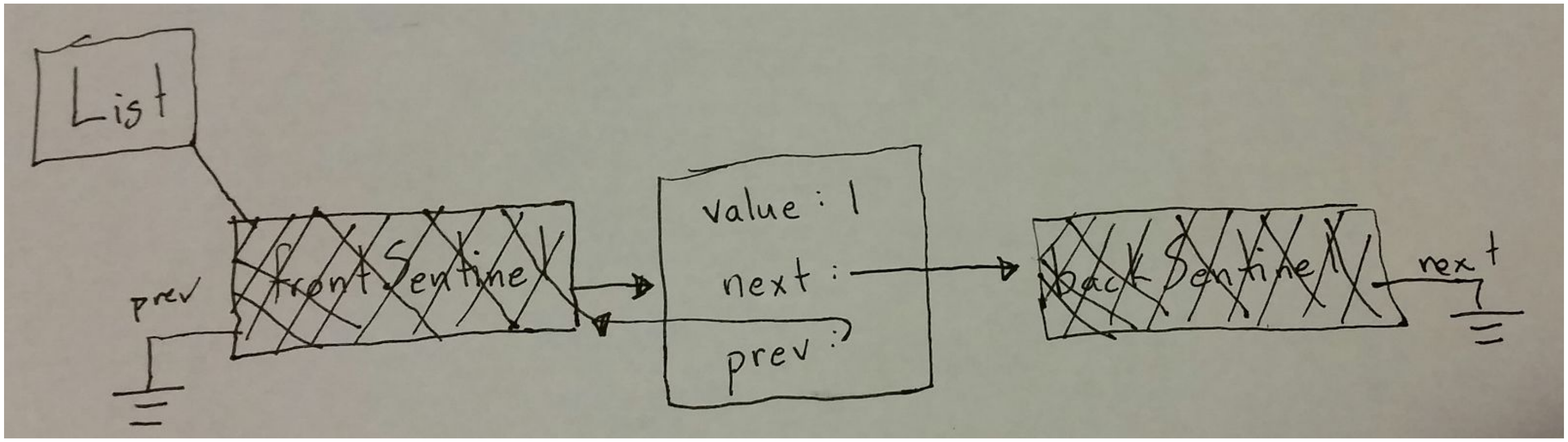
```
/*
     This function returns the value that appears at the front of the deque.  Given that this
     function should not be called on an empty deque, the function first checks that the deque
     is not empty.  If the deque is empty, then an assertion error occurs.  Otherwise, the
     value at the front of the deque is returned.
*/
TYPE LinkedListBack(struct linkedList *q) {
     assert(!LinkedListIsEmpty(q));
     return q->backSentinel->prev->value;
}
```

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

1. Draw a picture of an initially empty LinkedList, including the two sentinels.



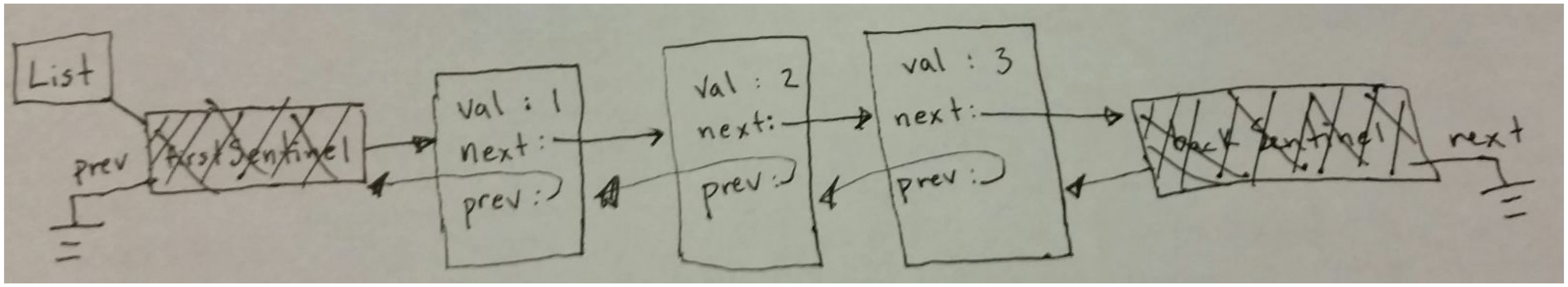
2. Draw a picture of the LinkedList after the insertion of one value.



3. Based on the previous two pictures, can you describe what property characterizes an empty collection?

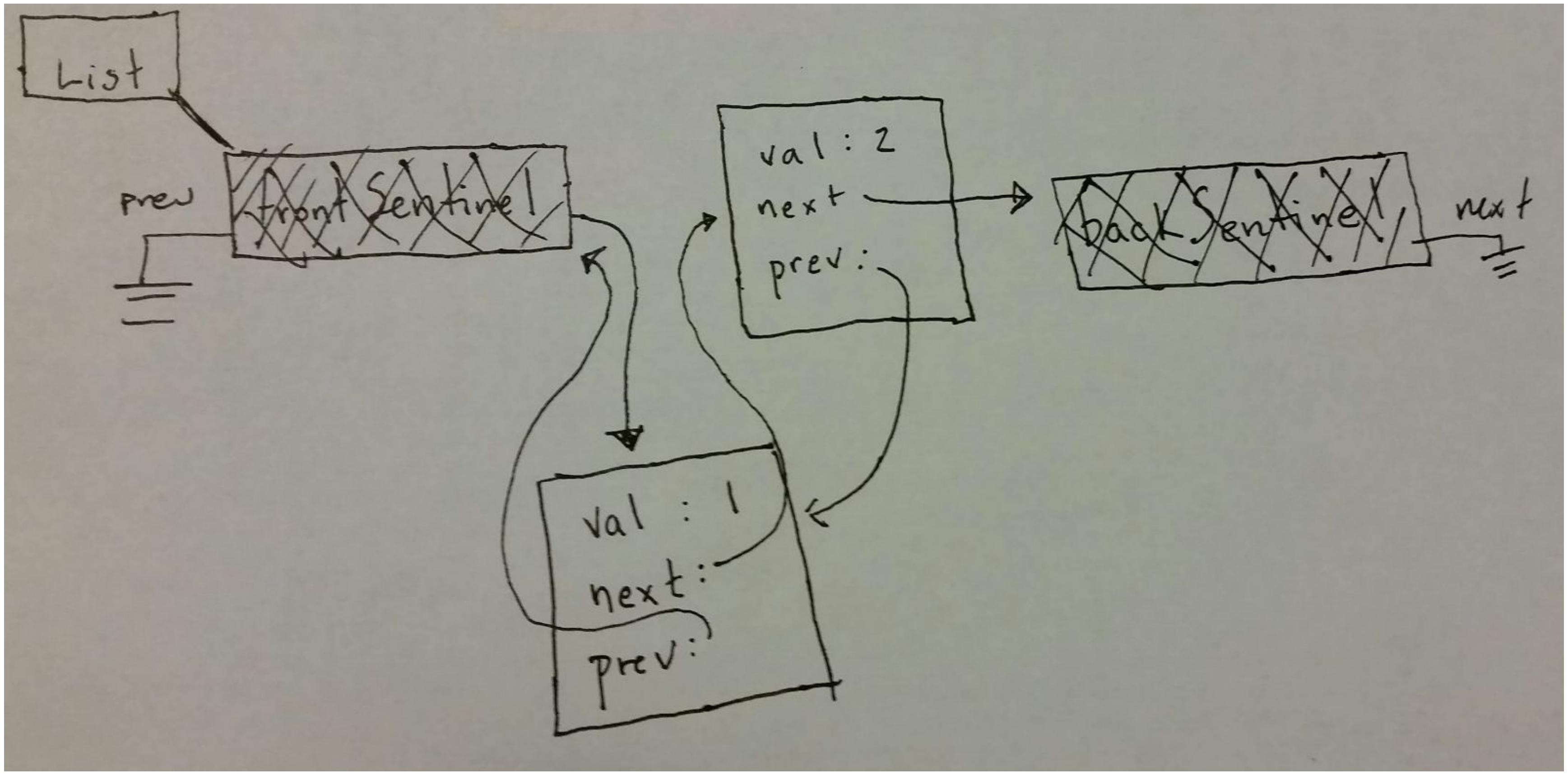
An empty collection exists when the frontSentinel's next field points to backSentinel; this could also be determined if the backSentinel's prev field points to frontSentinel.

4. Draw a picture of a LinkedList with three or more values (in addition to the sentinels).

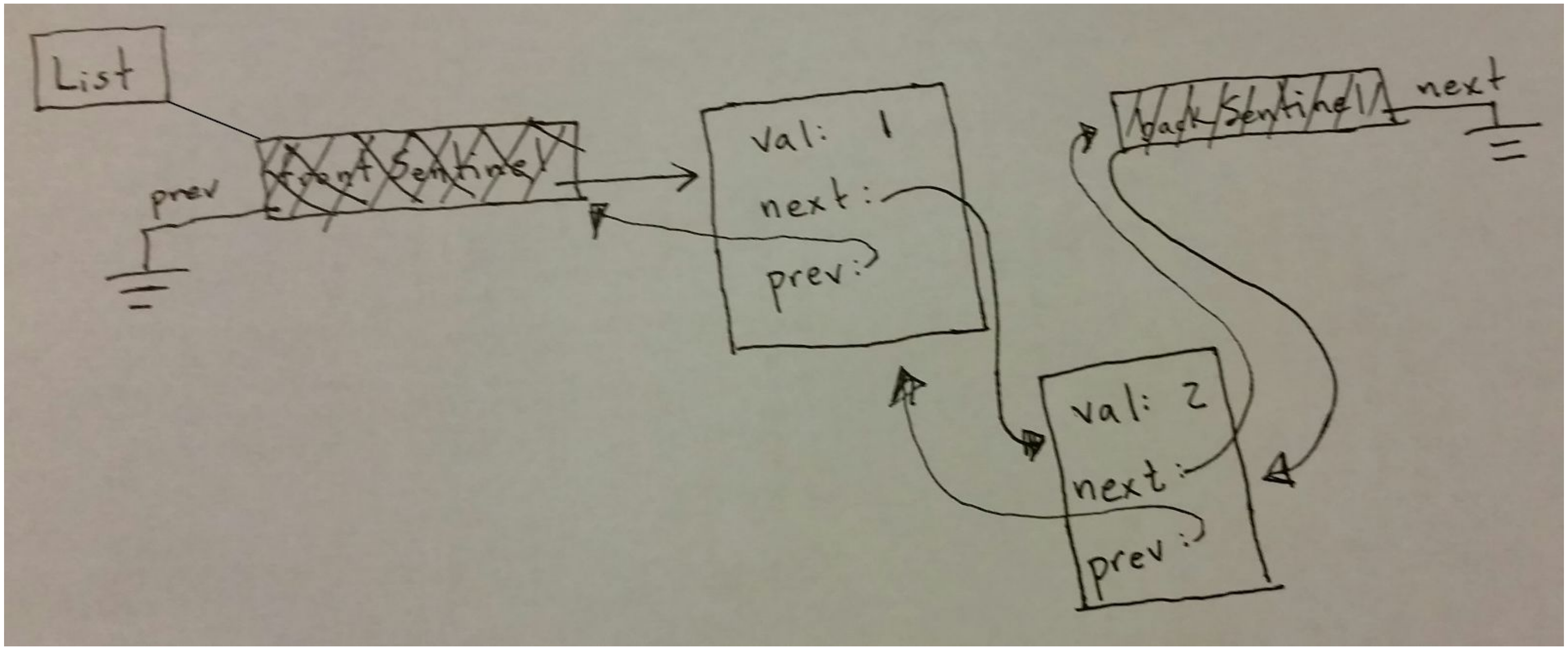


5. Draw a picture after a value has been inserted into the front of the collection. Notice that this is between the front sentinel and the following element. Draw a picture showing an insertion into the back. Notice that this is again between the last element and the ending sentinel. Abstracting from these pictures, what would the function addBefore need to do, where the argument is the link that will follow the location in which the value is inserted.

This picture shows the process of inserting a value at the front of the deque:



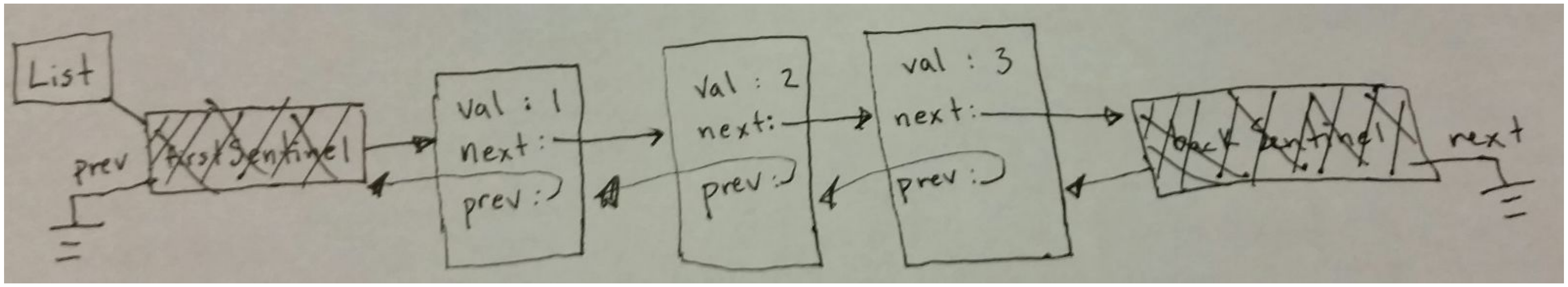
This picture shows the process of inserting a value at the back of the deque:



The addBefore function allocates memory for the new link and stores the value in the new link. It then adjusts the pointers for the links that will appear before and after the new link.

6. Draw a picture of a linkedList with three or more values, then examine what is needed to remove both the first and the last element. Can you see how both of these operations can be implemented as a call on a common operation, called _removeLink?

We can simply reuse the picture that we drew earlier:



In order to remove a link at the front or the pack, we will first begin by marking the link to be removed as garbage. We then set the next field for the link that appears before the garbage link to point to the link that appears after the garbage link. Following that, we set the prev field for the link that appears after the garbage link to point to the link that appears before the garbage link. We then free the memory reserved for the garbage link. These same actions will need to be performed for links at both the front and the back, so it can be abstracted into a function.

7. What is the algorithmic complexity of each of the deque operations?

The algorithmic complexity of each of the deque operations is shown below:

Operation	**Complexity**
LinkedListAddFront	O(1)
LinkedListAddBack	O(1)
LinkedListRemoveFront	O(1)
LinkedListRemoveBack	O(1)
LinkedListFront	O(1)

LinkedListBack	O(1)
LinkedListIsEmpty	O(1)

Piazza Discussion

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