Worksheet 19: Linked List Deque   Name:

Group 11:

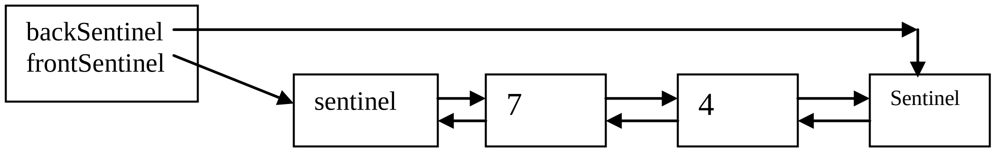
Danny Mejia

Tatyana Vlaskin
Katherin Jensen

Worksheet 19: Linked List Deque

In Preparation: Read Chapter 7 to learn more about the Deque data type. If you have not done so already, you should complete worksheets 17 and 18 to learn about the basic features of a linked list.

In this lesson we continue looking at variations on the theme of linked lists, this time including double links and sentinels on both the front and the back of the list. This will be the first of several lessons that will develop a very general purposed linked list abstraction. In this lesson we will emphasize the deque aspects of the list. In the following lesson we will add more operations.



A deque, you recall, allows insertions at both the beginning and the end of the container. The conceptual interface is shown at right. (Recall that the actual functions may differ from the conceptual interface in two ways. First, the names are likely differ to accommodate the C restriction that all functions have unique names. Second, the actual functions will pass the underlying data area as an argument).

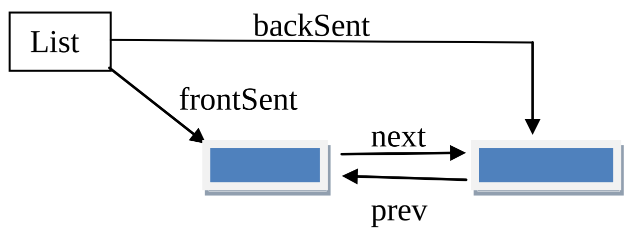
```
Conceptual Deque operations

void addFront (TYPE e)
void addBack (TYPE e)
TYPE front ()
TYPE back ()
void removeFront ()
void removeBack ()
int isEmpty ()
```

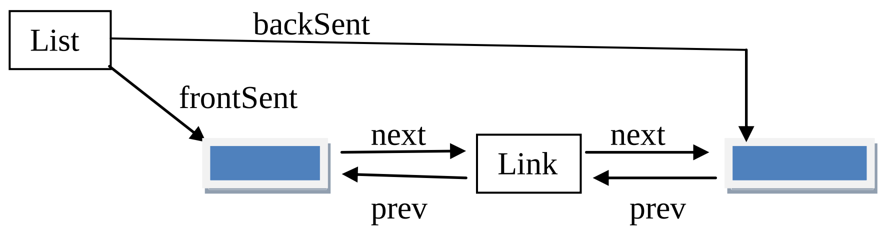
Our linkedList data structure will have two major variations from the linked list stack and queue you have examined earlier. First, an integer data field will remember the number of elements (that is, the size) in the container. Second, we will use sentinels at both the front and back of the linked list.

Sentinels ensure the list is never completely empty. They also mean that all instructions can be described as special cases of more general routines. The internal method _addBefore will add a link before the given location. The second routine _removeLink, will remove its argument link. Both of these use the underscore convention to indicate they are internal methods. When a value is removed from a list make sure you free the associated link fields. Use an assertion to check that allocations of new links are successful, and that you do not attempt to remove a value from an empty list. The following instructions will help you understand how to implement these operations.

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



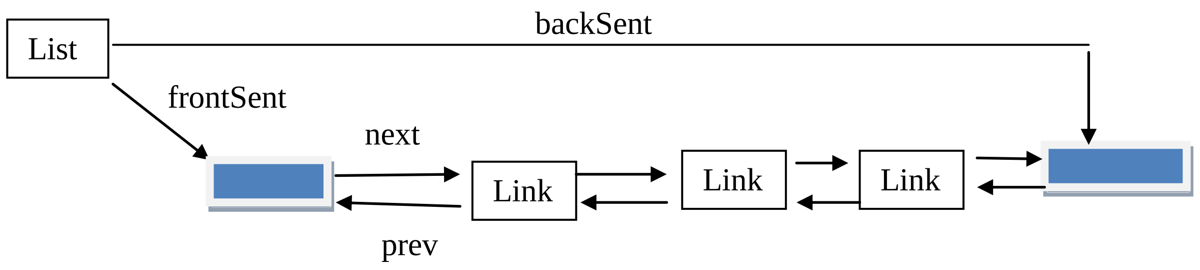
2. Draw a picture of the LinkedList after the insertion of one value.



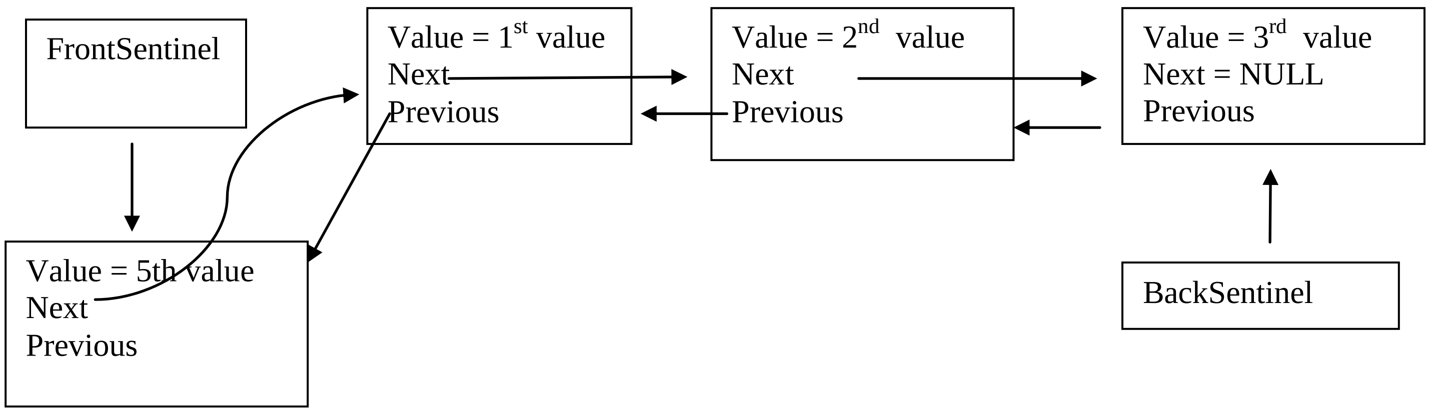
3. Based on the previous two pictures, can you describe what property characterizes an empty collection?

The empty collection has two sentinels and that's it. No other links.

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

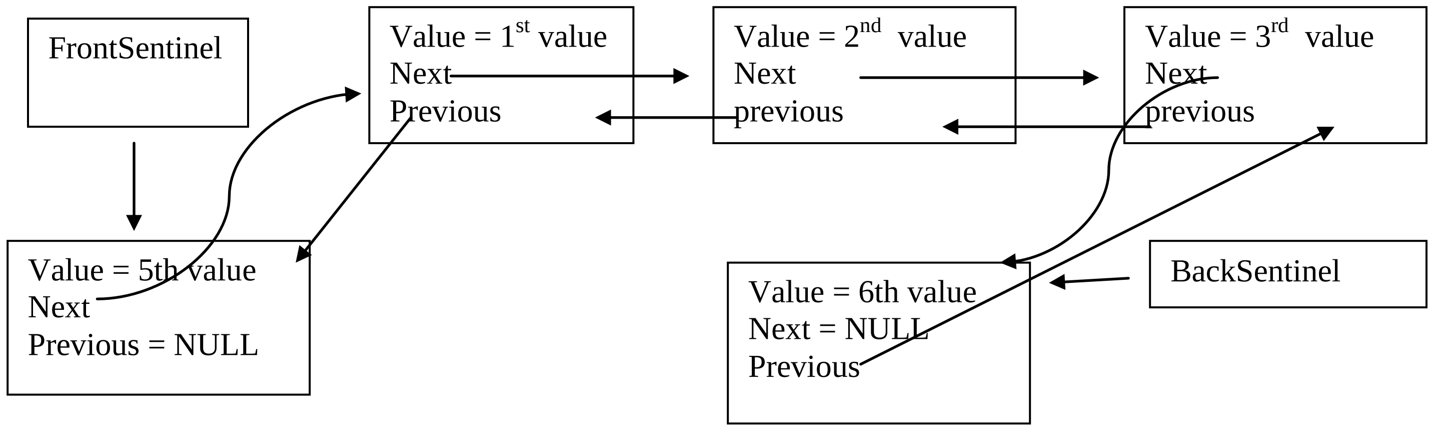


5. Draw a picture after a value has been inserted into the front of the collection. Notice that this is between the front sentinel and the following element. Draw a picture showing an insertion into the back. Notice that this is again between the last element and the ending sentinel. Abstracting from 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.
ELEMENT IS INSERTED IN THE FRONT:



Worksheet 19: Linked List Deque   Name:

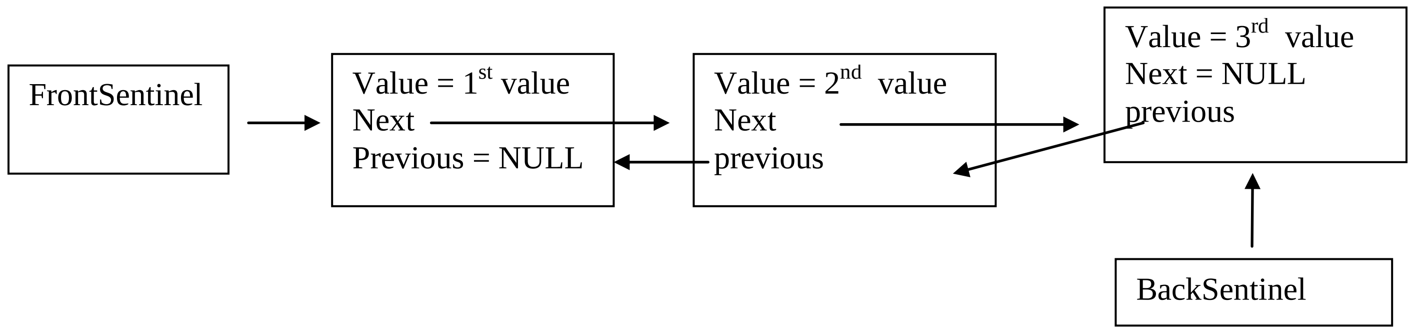
ELEMENT IS INSERTED IN THE BACK



The function addBefore needs to do the following:

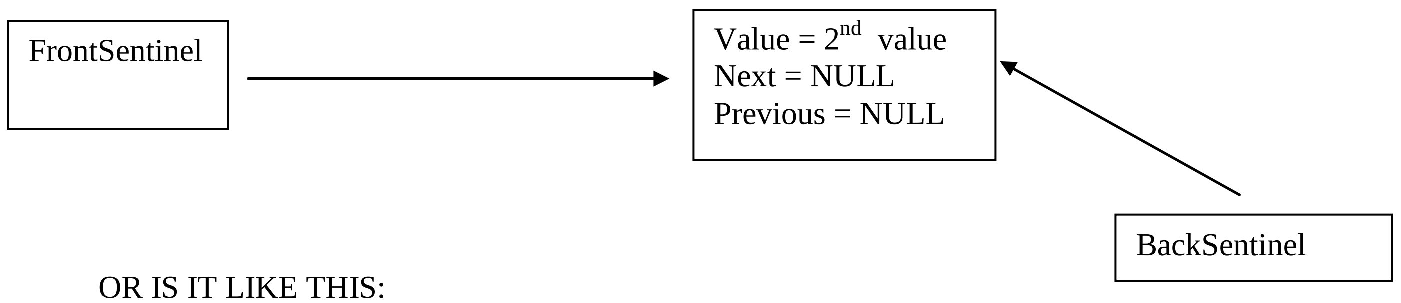
1. Add a new node
2. Set the next pointer of the new node to point to the previous 1st node in the linked list
3. Set the previous pointer of the previous 1st node to point to the newly added node
4. Change the pointer in the FrontSentinel and make it point to the newly added node.

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?

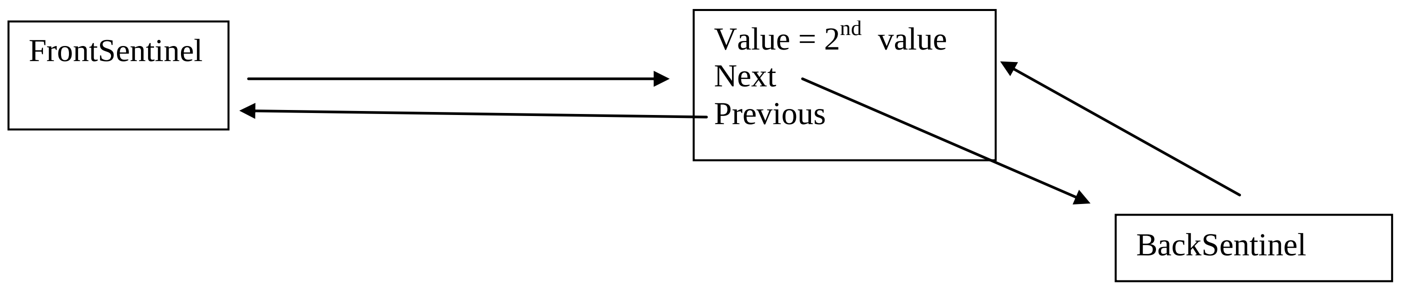


To remove the 1st and the last element (in the picture it is the element 3), we need to reassign the pointers of the frontSentinel and BackSentinel. The frontSentinel pointer needs to start pointing at the next (2nd in the picture element/node) and the lastSentiel needs to start point at the previous (2nd on the picture node). At the same time, once the node is removed from the front, we need to reassign previous pointer in the node that become new 1st node. We were not sure if the previous pointer needs to be assigned to NULL or previous pointer will point to the frontsentinel (see pictures below). Similar situation is with the next pointer of the node that becomes the last node. We need to change the next pointer to either point to the back sentinel or set it to Null.
It is not clear to us if 1st and last links in the linked list point to the sentinels or they will be assigned to NULL if there are no nodes in front and in the back?

We have provided 2 pictures:



OR IS IT LIKE THIS:



7. What is the algorithmic complexity of each of the deque operations?
O(1) because we have 2 sentinels. One that points to the front of the linked list and other one that point to the back of the linked list. As a result of this we can easily remove a node either from the end or from the front of the list.

```
struct dlink {
  TYPE value;
  struct dlink * next;
  struct dlink * prev;
};

struct linkedList {
  int size;
  struct dlink * frontSentinel;
  struct dlink * backSentinel;
};

/* these functions are written for you */
void LinkedListInit (struct linkedList *q) {
  q->frontSentinel = malloc(sizeof(struct dlink));
  assert(q->frontSentinel != 0);
  q->backSentinel = malloc(sizeof(struct dlink));
  assert(q->backSentinel);
  q->frontSentinel->next = q->backSentinel;
  q->backSentinel->prev = q->frontSentinell;
  q->size = 0;
}

void linkedListFree (struct linkedList *q) {
  while (q->size > 0)
    linkedListRemoveFront(q);
  free (q->frontSentinel);
  free (q->backSentinel);
  q->frontSentinel = q->backSentinel = null;
}

void LinkedListAddFront (struct linkedList *q, TYPE e)
  { _addBefore(q, q->frontSentinel_>next, e); }

void LinkedListAddback (struct linkedList *q, TYPE e)
  { _addBefore(q, q->backSentinel, e); }

void linkedListRemoveFront (struct linkedList *q) {
  assert(! linkedListIsEmpty(q));
  _removeLink (q, q->frontSentinel->next);
}

void LinkedListRemoveBack (struct linkedList *q) {
  assert(! linkedListIsEmpty(q));
  _removeLink (q, q->backSentinel->prev);
}
```

```
int LinkedListIsEmpty (struct linkedList *q) {
   return q->size == 0;
}


/* write addLink and removeLink. Make sure they update the size field correctly */
/* _addBefore places a new link BEFORE the provide link, lnk */
void _addBefore (struct linkedList *q, struct dlink *lnk, TYPE e) {
        struct dlink * newlink = (struct dlink *) malloc (sizeof(struct dlink));
        asser(newlink !=0);
        newlink->value = e;
        newlink->prev = lnk->prev;
        newlink->next = lnk;
        lnk->prev->next= newlink;
        lnk->prev= newlink;
        q->size++

}

void _removeLink (struct linkedList *q, struct dlink *lnk) {
        lnk->prev->next = lnk->next;
        lnk->next->prev = lnk ->prev;
        free(lnk);
        q->size--;
}

TYPE LinkedListFront (struct linkedList *q) {

        Assert(!LinkedListIsEmpty(q));
        Return q->frontSentinel->next->value;
}

TYPE LinkedListBack (struct linkedList *q) {
        Assert(!LinkedListIsEmpty(q));
        Return q->backSentinel->prev->value;
}
```

REFERENCES:
https://gist.github.com/BAKERDAR/7048438