Linked List Implementation of the Deque
Deque Interface (Review)

```c
void addFront(TYPE val); // Add value at front of deque.
void addBack (TYPE val); // Add value at back of deque.
void removeFront(); // Remove value at front.
void removeBack (); // Remove value at back.
TYPE front(); // Get value at front of deque.
TYPE back(); // Get value at back of deque.
```
Linked List Deque

- What if we want to add and remove elements from both front and back?
Modification #3: Double Links

- Point forward to the `next` element
- Point backwards to the `previous` element

```c
struct DLink {
    TYPE val;
    struct DLink *next; /* Link to prev node. */
    struct DLink *prev; /* Link to next node. */
};
```

![Diagram of double-linked list](image-url)
struct linkedList {
    int size;
    struct dlink * frontSentinel;
    struct dlink * backSentinel;
};
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->frontSentinal;
    q->size = 0;
}

How can we test for an empty deque?
Advantage of Sentinels

- Consider a deque, with two sentinels A, B:
  - Pointer to front sentinel: `frontSent`
  - Pointer to back sentinel: `backSent`
- Add to front and add to back are now special cases of more general “add before” operation

This is similar to most standard library Deque implementations (Java LinkedList)
Consider a deque, with two sentinels:

- Pointer to front sentinel: `frontSent`
- Pointer to back sentinel: `backSent`

Add to front and add to back are now special cases of more general “add before” operation.

This is similar to most standard library Deque implementations (Java LinkedList)
void addBackListDeque(struct ListDeque *q, TYPE val) {
    _addBefore(q->backSent, val);
}

void addFrontListDeque(struct ListDeque *q, TYPE val) {
    _addBefore(q->frontSent->next, val);
}
Adding to the LL Deque

```c
void addBackListDeque(struct ListDeque *q, TYPE val) {
    _addBefore(q->backSent, val);
}

void addFrontListDeque(struct ListDeque *q, TYPE val) {
    _addBefore(q->frontSent->next, val);
}
```

![Diagram of list with backSent and frontSent pointers](image)
Removing from the LL Deque

```c
void removeFirstListDeque(struct ListDeque *q) {
    assert(!isEmptyListDeque(q));

    _removeLink(q->frontSent->next);
}

void removeLastListDeque(struct ListDeque *q) {
    assert(!isEmptyListDeque(q));

    _removeLink (q->backSent->prev);
}
```
Your Turn...

Worksheet #19: _addBefore, _removeLink

<table>
<thead>
<tr>
<th></th>
<th>DynArrDeque best, ave, worst</th>
<th>LLDeque best, ave, worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>addLast</td>
<td>O(1), O(1+), O(N)</td>
<td>O(1), O(1), O(1)</td>
</tr>
<tr>
<td>removeLast</td>
<td>O(1), O(1), O(1)</td>
<td>O(1), O(1), O(1)</td>
</tr>
<tr>
<td>addFirst</td>
<td>O(1), O(1+), O(N)</td>
<td>O(1), O(1), O(1)</td>
</tr>
<tr>
<td>removeFirst</td>
<td>O(1), O(1), O(1)</td>
<td>O(1), O(1), O(1)</td>
</tr>
</tbody>
</table>
_addBefore

```c
void _addBefore(struct DLink *link, TYPE val) {
    struct DLink *newLink = malloc(...);
    assert(newLink != 0);
    newLink->val = val;
    newLink->next = link;
    newLink->prev = link->prev;

    link->prev->next = newLink;
    link->prev = newLink;
}
```

```c
struct DLink {
    TYPE val;
    struct DLink *next;
    struct DLink *prev;
}
```