Linked List Implementation of the Queue
Time complexity of **ListStack** operations:

- Push: $O(1)$ always
- Pop: $O(1)$ always
- Top: $O(1)$ always

How would this compare to a **DynArr** (a dynamic array implementation of a stack)?

- Push: $O(1+)$ average, $O(n)$ worse, $O(1)$ best
- Pop: $O(1)$ always
- Top: $O(1)$ always
- In practice, dynamic array is slightly faster in real timings
• Could we use our linked list as is, to implement a queue?
Modification#1: Tail Pointer

Which side should we make the ‘front’ of the queue?
Modification#2: Sentinel

- A sentinel is a special marker at the front and/or back of the list
- Has no value and never removed
- Helps us avoid special cases in the code associated with null references since it’s never null (e.g. first/last never point to null)
- Simplifies some operations
- An empty list always has a sentinel
struct listQueue {
    struct Link *firstLink; /* Always pts to Sent */
    struct Link *lastLink;
}

After additions
void listQueueInit (struct listQueue *q) {
  struct link *lnk = malloc(sizeof(struct link));
  assert(lnk != 0); /* lnk is the sentinel */
  lnk->next = 0;
  q->firstLink = q->lastLink = lnk;
}
/* No Sentinel */

void addBackListQueue (struct listQueue *q, TYPE e) {
    struct Link * lnk = ...
    assert(lnk != 0);
    lnk->next = 0;
    lnk->value = e;
    /* lastLink may be null!! */

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

Empty Case?

List

firstLink

List

lastLink

Link

next

... next

Link

next
```c
/* Sentinel */
void addBackListQueue (struct listQueue *q, TYPE e) {
    struct Link * lnk = malloc(....)
    assert(lnk != 0);
    lnk->next = 0;
    lnk->value = e;
    /* we know it has a lastLink. */
    q->lastLink->next = lnk;
    q->lastLink = lnk;
}
```
Your Turn

- Worksheet #18
  - Linked List Queue Implementation