In lecture, we talked about the `is_segment(start, end)` function that tells us we can start at `start`, follow `next` pointers, and get to `end` without ever encountering a `NULL`. (We won’t worry about the problems with getting `is_segment` to terminate in this recitation.) A `linkedlist` is a non-`NULL` pointer that captures a reference to both the start and end of a linked list.

Here’s an example of a specification function that uses `is_segment` as a precondition. Why are the pointer dereferences on line 7 and 8 safe?
Creating a new linked list

Here's the code that creates a new linked list with one non-dummy node. Suppose `linkedlist_new(12)` is called. For each of lines 4-9 (inclusive) draw a diagram that shows the state of the linked list after that line executes. Use X for struct fields that we haven't initialized yet.

```c
linkedlist* linkedlist_new(int data)
/*@ensures is_linkedlist(result);*/
{
    list* p = alloc(struct list_node);
    p->data = data;
    p->next = alloc(struct list_node);
    linkedlist* L = alloc(struct linkedlist_header);
    L->start = p;
    L->end = p->next;
    return L;
}
```
Adding to the end of a linked list

We can add to either the start or the end of a linked list. When we discussed the implementation of stacks in lecture, we were adding to the front. The following code adds a new list node to the end, the way a queue would:

```c
void add_end(linkedlist* L, int x)
    //@requires is_linkedlist(L);
    //@ensures is_linkedlist(L);
{
    list* p = alloc(struct list_node);
    L->end->data = x;
    L->end->next = p;
    L->end = p;
}
```

Suppose `add_end(L, 3)` is called on a linked list `L` that contains before the call, from start to end, the sequence `(1, 2)`. Draw the state of the linked list after each of lines 5 - 8 (inclusive). Include the list struct separately before it has been added to the linked list.

5.

6.

7.

8.
Removing the first item from a linked list

This is the code that removes the first element from a linked list. If it were not for the second precondition, we might remove the dummy node! This would almost certainly cause the postcondition to fail.

```c
int remove(linkedlist* L) {
    //@requires is_linkedlist(L);
    //@requires L->start != L->end;
    //@ensures is_linkedlist(L);
    { 
        int x = L->start->data;
        L->start = L->start->next;
        return x;
    }
}
```

Suppose `remove(L)` is called on a linked list `L` that contains before the call, from `start` to `end`, the sequence `(4, 5, 6)`. Draw the state of the linked list after lines 6 and 7 execute. Include an indication of what data the variable `x` holds.