Linked list segments

```c
struct list_node {
    int data;
    struct list_node* next;
};

typedef struct list_node list;

bool is_segment(list* start, list* end) {
    if (start == NULL) return false;
    if (start == end) return true;
    return is_segment(start->next, end);
}

struct linkedlist_header {
    list* start;
    list* end;
};

typedef struct linkedlist_header linkedlist;

bool is_linkedlist(linkedlist* L) {
    if (L == NULL) return false;
    return is_segment(L->start, L->end);
}
```

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?

```c
bool eq(list* start1, list* end1, list* start2, list* end2)
{//@requires is_segment(start1, end1);
  //@requires is_segment(start2, end2);
  {
      if (start1 == end1 && start2 == end2) return true;
      if (start1 == end1 || start2 == end2) return false;
      return start1->data == start2->data
              && eq(start1->next, end1, start2->next, end2);
  }
```
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.
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.