# 15-122: Principles of Imperative Computation — Spring 2024

## Recitation 06: Link it All Together — Wednesday February 14th

## Linked list segments

```
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 valid value of type `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.

```
1 bool eq(list* start1, list* end1, list* start2, list* end2)
2 //@requires is_segment(start1, end1);
3 //@requires is_segment(start2, end2);
4 {
5   if (start1 == end1 && start2 == end2) return true;
6   if (start1 == end1 || start2 == end2) return false;
7   return (start1->data == start2->data)
8       && eq(start1->next, end1, start2->next, end2);
9 }
```

## Checkpoint 0

Why are the pointer dereferences on lines 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.

```
1 linkedlist* linkedlist_new(int data)
2 //@ensures is_linkedlist(\result);
3 {
4   list* p = alloc(list);
5   p->data = data;
6   p->next = alloc(list);
7   linkedlist* L = alloc(linkedlist);
8   L->start = p;
9   L->end = p->next;
10  return L;
11 }
```

## Checkpoint 1

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 if we *don't care* about their values.

4.

5.

6.

7.

8.

9.

## Adding to the end of a linked list

We can add to either the start or the end of a linked list. The following code adds a new list node to the *end*, the way a queue would:

```
1 void add_end(linkedlist* L, int x)
2 //@requires is_linkedlist(L);
3 //@ensures is_linkedlist(L);
4 {
5   list* p = alloc(list);
6   L->end->data = x;
7   L->end->next = p;
8   L->end = p;
9 }
```

## Checkpoint 2

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 elements

Complete the function below which removes every second element from the input linked list by modifying it. For example, when called on a list `L` with 1, 2, 3, 4 as data values, `L` will contain 1, 3 as data elements when the function returns.

```
void remove_every_second_node(linkedlist* L)

//@requires ______________________________________;

//@ensures _______________________________________;
{
  list* p = ______________________________________;

  while (_________________________________________)
  //@loop_invariant is_segment(L->start, p);
  //@loop_invariant is_segment(p, L->end);
  {
    ______________________________________________;

    ______________________________________________;
  }
}
```

## Using `is_segment` as a loop invariant

A loop invariant must be initially true, must be preserved by every iteration of the loop, and together with the negation of the loop guard must imply the postcondition.

## Checkpoint 3

What are the loop invariants we need to prove the correctness of this function?

```
linkedlist* copy(linkedlist* L)
//@requires is_linkedlist(L);
//@ensures is_linkedlist(\result);
//@ensures \result != L;
//@ensures eq(L->start, L->end, \result->start, \result->end);
{
  linkedlist* N = alloc(linkedlist);
  N->start = alloc(list);
  list* o = L->start;
  list* n = N->start;
  while (o != L->end)
  //@loop_invariant ______________________________;
  //@loop_invariant ______________________________;
  //@loop_invariant ______________________________;
  //@loop_invariant ______________________________;
  {
    n->data = o->data;
    n->next = alloc(list);
    o = o->next;
    n = n->next;
  }
  N->end = n;
  return N;
}
```

## Termination

Termination arguments about linked lists are often a bit more abstract than what we have experienced in the past.

## Checkpoint 4

The loop terminates because ______________________________

is strictly decreasing at each iteration of the loop and can never get smaller than ______________

where the loop guard evaluates to ______________.