

15-122: Principles of Imperative Computation

R5: A Linked List Between Worlds Rob Simmons, Andrew Benson

Linked List Segments

```
1 struct list_node {
2     int data;
3     struct list_node* next;
4 };
5 typedef struct list_node list;
6
7 bool is_segment(list* start, list* end) {
8     if (start == NULL) return false;
9     if (start == end) return true;
10    return is_segment(start->next, end);
11 }
12
13 struct linkedlist_header {
14     list* start;
15     list* end;
16 };
17 typedef struct linkedlist_header linkedlist;
18
19 bool is_linkedlist(linkedlist* L) {
20     if (L == NULL) return false;
21     return is_segment(L->start, L->end);
22 }
```

The `is_segment(start, end)` function tells us that we can go from `start` to `end` by following `next` pointers without ever encountering `NULL` (we ignore cyclic linked lists in this recitation). A `linkedlist` is a non-`NULL` pointer that captures a reference to both the start and end of a linked list.

Now that we know what a linked list is, how do we create one?

Creating a new linked list

The following function creates a new linked list with a single data node. Suppose `linkedlist_new(12)` is called. Draw the final state of the linked list after that line executes. Use `X` for struct fields that we haven't initialized yet.

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

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:

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

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).

1. Justify the safety of each pointer dereference.
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.
3. What is the big-O runtime of this function?

Removing the first item from a linked list

As mentioned earlier, `add_end` can be used to implement `enq` for a queue. To implement `deq`, we would need a function that removes an element from the start of a linked list.

```
1 int remove(linkedlist* L)
2
3 //@requires _____
4
5 //@requires _____
6
7 //@ensures _____
8 {
9     _____
10
11     _____
12
13     _____
14 }
```

1. Fill in the code for the function `remove` that removes the first element of a linked list. Think carefully about what preconditions you would need.
2. What is the big-O runtime of this function?
3. If you were to remove an element from the end of the linked list instead, how would the big-O runtime change? Why?