15-122: Principles of Imperative Computation

Recitation 10 Solutions — Josh Zimmerman

Practice!

(Credit for this section goes to CMU alumna Caroline Buckey; it has been updated since by Alex Cappiello and Rob Simmons.)

Suppose you have the implementation using linked lists shown in lecture. Specifically, you have the following structs:

```
struct list_node {
  int data;
  struct list_node* next;
};
typedef struct list_node list;

struct linkedlist_header {
   list* start;
   list* end;
};
typedef struct linkedlist_header linkedlist;
```

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.

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

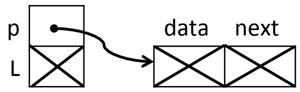
Recall from lecture that we always have one "dummy" node at the end of our linked list segments. Its fields are uninitialized; it simply ensures that we never need to worry about `start` or `end` being null.

Creating a new linked list

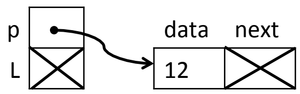
Here's the code that creates a new linked list with one non-dummy node. Suppose `linkedlist_new(12)` is called. For each of lines 4-9 (inclusive) draw a diagram that shows the state of the linked list after that line executes. Use `X` for struct fields that we haven't initialized yet.

```
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 }
```

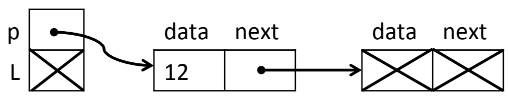
4\.

Solution: 

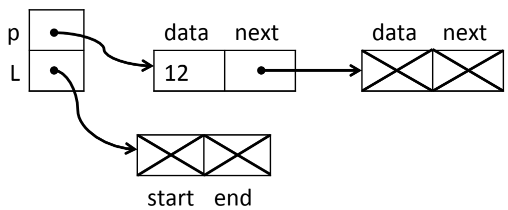
5\.

Solution: 

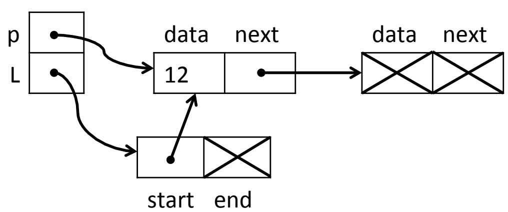
6\.

Solution: 

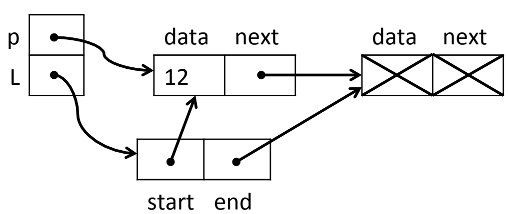
7\.

Solution: 

8\.

Solution: 

9\.

Solution: 

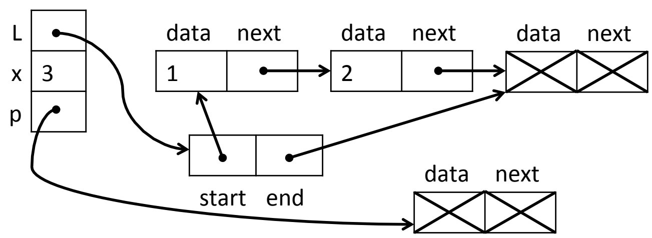
Adding to the end of a linked list

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*.

```
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 }
```

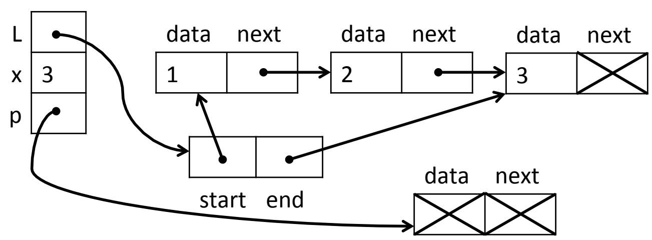
Suppose add_end(L, 3) is called on a linked list L that contains before the call, from start to end, the sequence (1, 2). Draw the state of the linked list after each of lines 5 - 8 (inclusive). Include the list struct separately before it has been added to the linked list.

5.



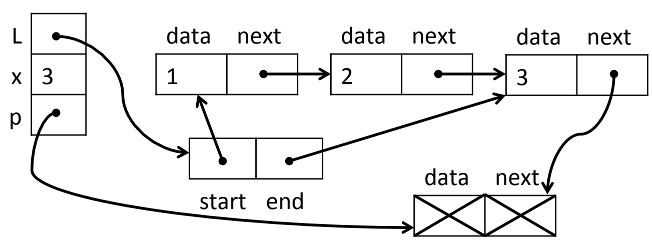
Solution:

6.



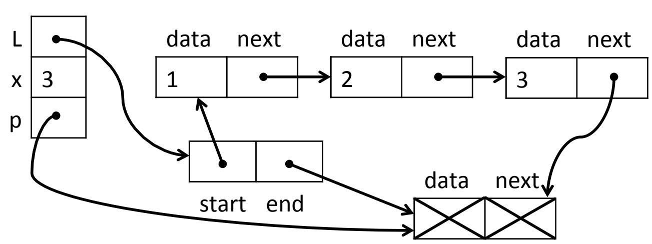
Solution:

7.



Solution:

8.



Solution:

Adding to the start of a linked list

With the previous example in mind, can you think about what code would be necessary if we instead wanted to add a new list node to the *start* of a linked list?

```
void add_start(linkedlist* L, int x)
//@requires is_linkedlist(L);
//@ensures is_linkedlist(L);
{
   list* p = alloc(struct list_nodes);
   p->data = x;
   p->next = L->start;
   L->start = p;
}
```

Removing the first item from a linked list

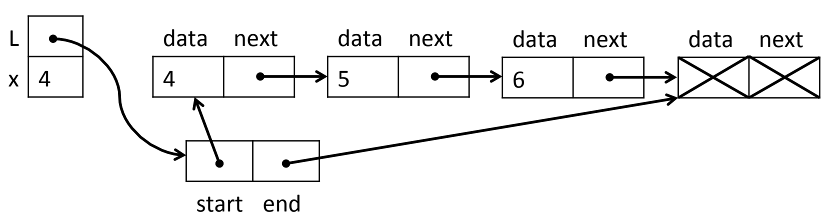
This is the code that removes the first element from a linked list. If it were not for the second precondition, we might remove the dummy node! This would almost certainly cause the postcondition to fail.

```
1 int remove(linkedlist* L)
2 //@requires is_linkedlist(L);
3 //@requires L->start != L->end;
4 //@ensures is_linkedlist(L);
5 {
6    int x = L->start->data;
7    L->start = L->start->next;
8    return x;
9 }
```

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.

6.

Solution:



7.

Solution: