15-122 : Principles of Imperative Computation, Fall 2013

Written Homework 5

Due: Thursday, October 3, 2013, at 10pm

Name: _____

Andrew ID: _____

Recitation: _____

The written portion of this week's homework will give you some practice working with linked lists, pointers and the principle of an interface. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	9	
2	5	
3	6	
Total:	20	

Write your answers *neatly* on this PDF (or fill out the TeX handout), and then submit the stapled printout to the handin box Thursday before lecture or on Thursday afternoon outside of Tom Cortina's office (GHC 4117).

1. **Linked Lists**

You are given the following C0 type definitions for a linked list of integers:

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

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

An empty list consists of one list_node. All lists have one additional node at the end that does not contain any relevant data, as discussed in class.

(7) (a) In this task, we ask you to analyze a list function and reason that each pointer access is safe. You will do this by indicating the line(s) in the code that you can use to conclude that an access is safe. Your analysis must be precise and minimal: only list the line(s) upon which the safety of a pointer dereference depends. If a line does not include a pointer dereference, indicate this by writing NONE after the line in the space provided. As an example, we show the analysis for an `is_segment` function below.

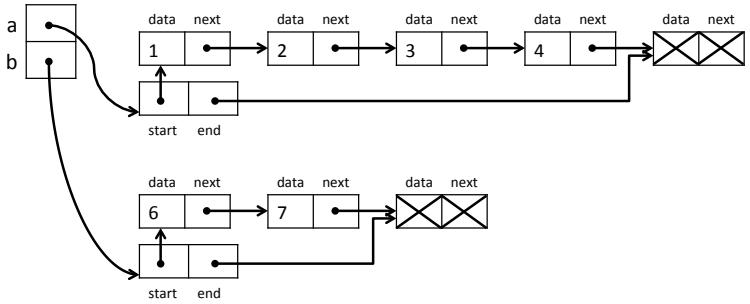
```
          bool is_segment(list* s, list* e) {
/*  1 */      if (s == NULL) return false;            NONE
/*  2 */      if (e == NULL) return false;            NONE
/*  3 */      if (s->next == e) return true;          1
/*  4 */      list* c = s;                            NONE
/*  5 */      while (c != e && c != NULL) {           NONE
/*  6 */          c = c->next;                        5
/*  7 */      }                                       NONE
/*  8 */      if (c == NULL)                          NONE
/*  9 */          return false;                       NONE
/* 10 */      return true;                            NONE
          }
```

Complete the analysis of the `mystery` function on the next page. The first two lines of code are analyzed for you.

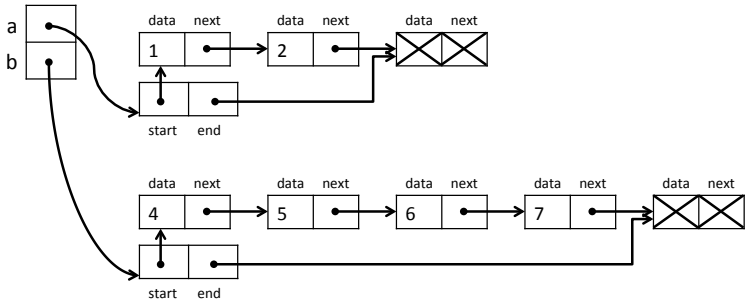
Solution:

```
          void mystery(linkedlist a, linkedlist b)

/*  1 */  //@requires a != NULL;                          ___NONE___

/*  2 */  //@requires b != NULL;                          ___NONE___

/*  3 */  //@requires is_segment(a->start, a->end);      __________

/*  4 */  //@requires is_segment(b->start, b->end);      __________

          {

/*  5 */    list* nptr = b->start;                       __________

/*  6 */    list* t1 = a->start;                         __________

/*  7 */    list* t2 = b->start;                         __________

/*  8 */    while (t1 != a->end && t2 != b->end)         __________

/*  9 */    //@loop_invariant is_segment(t1, a->end);    __________

/* 10 */    //@loop_invariant is_segment(t2, b->end);    __________

            {

/* 11 */        list* t = t2;                            __________

/* 12 */        t2 = t2->next;                           __________

/* 13 */        t->next = t1->next;                      __________

/* 14 */        t1->next = t;                            __________

/* 15 */        t1 = t1->next->next;                     __________

             }

/* 16 */     b->start = t2;                              __________

          }
```

(2) (b) Let `a` and `b` be two linked lists, with m and n data values, respectively, and we call `mystery(a,b);` on these two lists. For each of the following pictures, draw the final state of the lists after the function executes.



Solution:



Solution:

If $m \geq n$, what is the final length of linked list `a`?

Solution:

If $m < n$, what is the final length of linked list `a`?

Solution:

2. **A New Linked List Operation**

For the following question, assume the linked list type definitions from the previous problem. You are also given the following specification function that returns true if and only if `x` is greater than every node in the list from `start` (inclusive) to `end` (exclusive).

```
bool gt(int x, list* start, list* end);
```

(4) (a) Complete the function below that removes the maximum integer from a non-empty linked list of integers. You may assume there are no duplicate elements. (Note that loop invariants are not given so we can't reason about the safety of the code.)

Solution:

```
int remove_max(linkedlist a) {
    //@requires a != NULL;
    //@requires is_segment(a->start, a->end);

    //@requires ______________________________;   // List not empty
    //@ensures  is_segment(a->start, a->end);
    //@ensures gt(\result, a->start, a->end);
    list* first = a->start;
    list* curr = first->next;
    list* prev = first;
    list* max = first;
    list* max_prev = first;

    while (_____________________________________) {
        if (curr->data > max->data) {

            max_prev = ___________________;

            max = ______________________;
        }
        prev = ____________________________;

        curr  = ____________________________;
    }
    if (max == max_prev)

       ____________________________________________________;
    else

       ____________________________________________________;
    return max->data;
}
```

(1) (b) Explain in one sentence why the second postcondition specified in the function above is not strong enough to reason that this function removes and returns the maximum integer from the non-empty linked list.

Solution:

3. **The Stack Interface: Client and Implementation**

Consider the following interface for `stack` that stores elements of the type `elem`:

```
/* Stack Interface */
stack stack_new();              /* O(1) */
bool stack_empty(stack S);      /* O(1) */
void push(elem e, stack S);     /* O(1) */
elem pop(stack S)               /* O(1) */
  //@requires !stack_empty(S);
  ;
```

(2) (a) Write a client function `stack_bottom(stack S)` that returns, but does not remove, the bottom element of the given stack, assuming the stack is not empty. For this question, use only the interface since, as a client, you do not know how this data structure is implemented. Do not use any stack functions that are not in the interface (including specification functions like `is_stack` since these belong to the implementation).

Solution:

```
elem stack_bottom(stack S)
//@requires !stack_empty(S);
{









}
```

Now we look at the implementation side of the `stack` interface. Suppose we decide to implement the stack using a doubly-linked list so that each list node contains two pointers, one to the next node in the list and one to the previous (prev) node in the list:

```
struct list_node {
    elem data;
    struct list_node* prev;
    struct list_node* next;
};
typedef struct list_node list;
```

The top element of the stack will be stored in the first (head) node of the list, and the bottom element of the stack will be stored in the second-to-last node in the list, with the last node being a "dummy node". An empty stack consists of a dummy node only.

```
struct stack_header {
    list* top;
    list* end;   // points to dummy node
};
typedef struct stack_header* stack;
```

(2) (b) Modify the singly-linked list implementation of stacks given below to work with the doubly-linked list representation given above. For each function, either state the modification(s) that need to be made (e.g. "Insert the statement XXXX after line Y", "Remove line Z", "Change line Z to XXXX", etc.) or state "No change needs to be made.". You may assume there is an appropriate `is_stack` specification function already defined. Be sure that your modifications still maintain the O(1) requirement for the stack operations.

Solution:

```
        bool stack_empty(stack S)
        //@requires is_stack(S);
        {
/* 1 */     return S->top == S->end;
        }
```

```
        stack stack_new()
        //@ensures is_stack(\result);
        //@ensures stack_empty(\result);
        {
/* 1 */   stack S = alloc(struct stack_header);
/* 2 */   list* l = alloc(struct list_node);
/* 3 */   S->top = l;
/* 4 */   S->end = l;
/* 5 */   return S;
        }




        void push(elem x, stack S)
        //@requires is_stack(S);
        //@ensures is_stack(S);
        {
/* 1 */     list* l = alloc(struct list_node);
/* 2 */     l->data = x;
/* 3 */     l->next = S->top;
/* 4 */     S->top = l;
        }




        elem pop(stack S)
        //@requires is_stack(S);
        //@requires !stack_empty(S);
        //@ensures is_stack(S);
        {
/* 1 */    elem e = S->top->data;
/* 2 */    S->top = S->top->next;
/* 3 */    return e;
        }
```

(1) (c) We wish to add a new operation `stack_bottom` to our stack implementation from the previous part.

```
elem stack_bottom(stack S);   /* O(1) */
     //@requires !stack_empty(S);
```

This operation returns (but does not remove) the bottom element of the stack. Write an implementation for this function using the doubly-linked list implementation of stacks from the previous part. Be sure that your function runs in constant time. (NOTE: Remember that the linked list that represents the stack has a dummy node.)

Solution:

```
elem stack_bottom(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
{




}
```

(1) (d) If we didn't add the `prev` link to each node in the linked list, how long would it take to return the bottom element of the stack using big O notation if the list had n elements? Why? (NOTE: There is still a dummy node at the end of the linked list.)

Solution: