15-122: Principles of Imperative Computation, Spring 2023

Written Homework 6

Due on Gradescope: Sunday 19th February, 2023 by 9pm

Name: ______________________________

Andrew ID: ______________________________

Section: ______________________________

This written homework covers doubly-linked lists.

Preparing your Submission You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- ***PDFescape*** or ***DocHub***, two web-based PDF editors that work from anywhere.
- ***Acrobat Pro***, installed on all non-CS cluster machines, works on many platforms.
- ***iAnnotate*** works on any iOS and Android mobile device.

There are many more — use whatever works best for you. If you'd rather not edit a PDF, you can always print this homework, write your answers *neatly* by hand, and scan it into a PDF file — *we don't recommend this option, though.*

Caution Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

Submitting your Work Once you are done, submit this assignment on Gradescope. ***Always check it was correctly uploaded.*** You have unlimited submissions.

Question:	1	2	3	Total
Points:	4	5	6	15
Score:				

1. **Multiple Return Values**

One of the difficulties that both structs and pointers can solve in different ways is the problem of returning more than one piece of information from a function. For instance, a function that tries to parse a string as an integer needs to return both the successfully-parsed integer and information about whether that parse succeeded. Because *any* number could be the result of a successful parse, if the function only returns an **int**, there's no way to distinguish a failed parse from a successful one.

```
int parse_int(string str) {
   int k;
   bool parse_successful;
   ...
   if (parse_successful) return k;
   return ???; /* What do we do now? */
}
```

In each of the following exercises, a `main` function wants to print the value that `parse_int` is storing in the variable `k`, but *only* when the boolean value stored in `parse_successful` is `true`; otherwise we want to print out *"Parse error"*.

You don't have to use all the blank lines we have provided, but you shouldn't use any extra lines. **Double-check your syntax; we will be picky about syntax errors for this question.**

© Carnegie Mellon University 2023

2pts

1.1 Finish this program so that the code will parse the first command-line argument as an **int** if possible. Make sure all your pointer dereferences are provably safe.

```
int* parse_int(string str) {
    int k;
    bool parse_successful;
    // Omitted code that tries to parse the string. It puts
    // the result in the local variable k and sets
    // parse_successful to true if it can, otherwise sets
    // parse_successful to false.

    if (parse_successful) {

        _____________________________________________________;

        _____________________________________________________;

        return _______________________________________________;
    }
    return ___________________________________________________;
}

int main() {
    args_t A = args_parse();
    if (A->argc != 1) error("Wrong number of arguments");
    int* k_ptr = parse_int(A->argv[0]);

    if (____________________) printf("%d\n", _______________);
    else error("Parse error");
    return 0;
}
```

2pts

1.2 Complete another program that works the same way, but that gives a different type to `parse_int`. The missing argument should be a pointer. Make sure all your pointer dereferences are provably safe.

```
bool parse_int(string str, ________________________________)

//@requires ______________________________________________;
{
    int k;
    bool parse_successful;
    // Same omitted code...

    if (parse_successful) {

        ____________________________________________________;

        return _____________________________________________;
    }
    return _________________________________________________;
}

int main() {
    args_t A = args_parse();
    if (A->argc != 1) error("Wrong number of arguments");

    ________________________________________________________;

    bool res = parse_int(A->argv[0], ______________________);

    if (___________________) printf("%d\n", ______________);
    else error("Parse error");
    return 0;
}
```

2. **Reasoning with Linked Lists**

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

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

struct segment_header {
    list* start;
    list* end;
};
typedef struct segment_header* list_segment;
```

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

   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 the access is safe. Your analysis must be precise and minimal: mention only 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.

```
 1 bool is_segment(list* s, list* e) {
 2     if (s == NULL) return false;              // NONE
 3     if (e == NULL) return false;              // NONE
 4     if (s->next == e) return true;            // 2
 5     list* c = s;                              // NONE
 6     while (c != e && c != NULL) {             // NONE
 7         c = c->next;                          // 6
 8     }                                         // NONE
 9     if (c == NULL)                            // NONE
10         return false;                         // NONE
11     return true;                              // NONE
12 }
```

   When we reason that a pointer dereference is safe, the argument applies *only* to that dereference. So, in the example below, we have to use line 42 to prove both line 43 and line 44 safe.

```
42 //@assert is_segment(a, b);  // ASSUME VALID
43 a->next = b;
44 list* l = a->next;
```

   We don't allow you to say that, because line 43 didn't raise an error, `a` must not be `NULL` and therefore line 44 must be safe.

Here's a mystery function:

```
42 void mystery(list_segment a, list_segment b)
43 //@requires a != NULL;                        // NONE
44 //@requires b != NULL;                        // NONE
45 //@requires is_segment(a->start, a->end);     // __________
46 //@requires is_segment(b->start, b->end);     // __________
47 {
48   list* t1 = a->start;                        // __________
49   list* t2 = b->start;                        // __________
50   while (t1 != a->end && t2 != b->end)        // __________
51   //@loop_invariant is_segment(t1, a->end);   // __________
52   //@loop_invariant is_segment(t2, b->end);   // __________
53   {
54       list* t = t2;                           // __________
55       t2 = t2->next;                          // __________
56       t->next = t1->next;                     // __________
57       t1->next = t;                           // __________
58       t1 = t1->next->next;                    // __________
59     }
60     b->start = t2;                            // __________
61 }
```

You can use the blanks on the right for scratchwork — they are not graded.

1pt

2.1 Explain when line 50 is safe: first, clearly state what the conditions for the safety of line 50 are, and then write down the line numbers that support each of them — *only cite the necessary line numbers.*

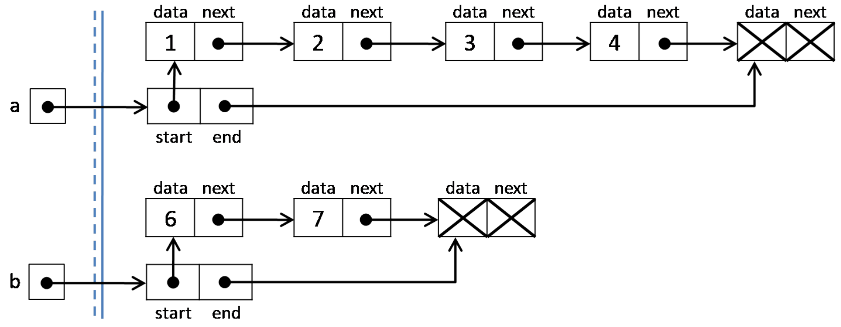
1pt

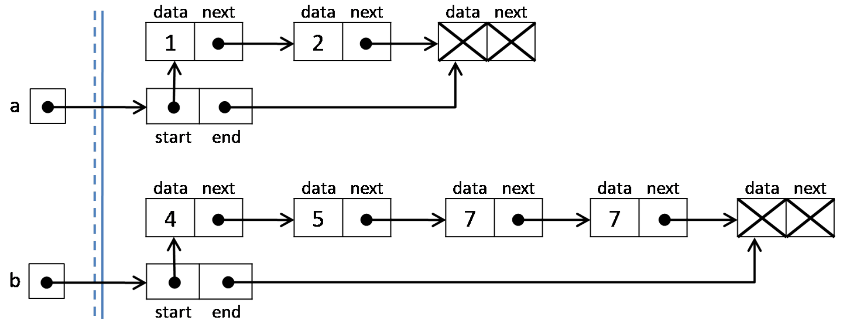
2.2 Why can we not use the combination of line 45 (which tells us that `a->start` is not `NULL`) and line 48 (which tells us that `t1` is `a->start`) to reason that `t1` is not `NULL` and therefore that line 57 is safe?

Why do we actually know line 57 is safe?

1.5pts

2.3 Let `a` and `b` be linked lists with m and n data values in them, respectively. For each of the pictures below, draw the final state of the lists after `mystery(a,b)` executes.





What is the final length of linked list `a` when

- $m \geq n$
- $m < n$

1.5pts

2.4 Complete the function below that removes the maximum integer from a non-empty linked list of integers. The specification function `gt_listseg(x,s,e)` checks that `x` is strictly larger than every element in the list segment between `s` inclusive and `e` exclusive. 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.)

```
int remove_max(list_segment a)

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

  while (________________________________________________)
  //@loop_invariant prev->next == curr;
  {
    if (curr->data > max->data) {

       max_prev = ________________________________________;

       max = _____________________________________________;
    }
    prev = ______________________________________________;

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

    _____________________________________________________;

  else __________________________________________________;
  return max->data;
}
```

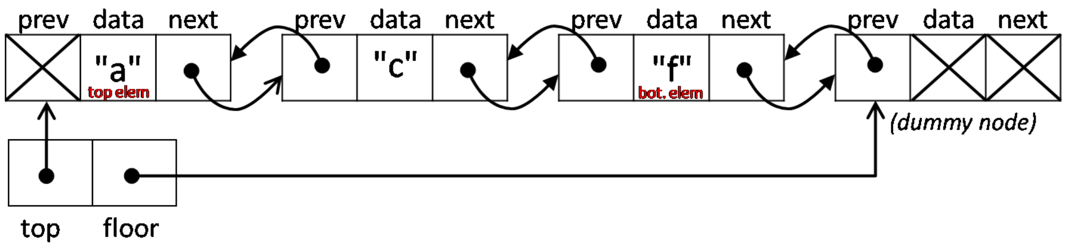
The second postcondition of this function is **not** strong enough. Describe a contract exploit where it holds true but a hypothetical function body does not remove and return the maximum integer from the non-empty input linked list.

3. **Doubly-Linked Lists**

   Consider the following interface for stacks that store elements of the type `string`:

```
// typedef ______* stack_t;

bool stack_empty(stack_t S)              /* O(1) */
/*@requires S != NULL; @*/ ;

stack_t stack_new()                      /* O(1) */
/*@ensures \result != NULL;      @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, string x)           /* O(1) */
/*@requires S != NULL;     @*/
/*@ensures !stack_empty(S); @*/ ;

string pop(stack_t S)                    /* O(1) */
/*@requires S != NULL;       @*/
/*@requires !stack_empty(S); @*/ ;
```

   Suppose we decide to implement the stack (of `string`'s) 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:



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

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

   The top element of the stack (if any) will be stored in the first node of the list (pointed to by `top`), and the bottom element of the stack (if any) will be stored in the second-to-last node in the list, with the last node being a "dummy node" (pointed to by `floor`). Intuitively, the bottom element sits on the `floor`.

   An empty stack consists of a dummy node only: the `prev`, `data`, and `next` fields of that dummy are all unspecified. A non-empty stack has an unspecified `prev` field for the top, and an unspecified `data` and `next` field for the dummy node.

2pts

3.1 Modify the singly-linked list implementation of stacks below to work with the doubly-linked list representation given above. For each function,

- **either** give 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** write "No change needs to be made" if the original code works for the doubly-linked list implementation.

Do not write to unspecified values unless strictly necessary. 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.

```
33 stack* stack_new()
34 //@ensures is_stack(\result);
35 //@ensures stack_empty(\result);
36 {
37     stack* S = alloc(stack);
38     list* L = alloc(list);
39     S->top = L;
40     S->floor = L;
41     return S;
42 }
```

```
44 bool stack_empty(stack* S)
45 //@requires is_stack(S);
46 {
47     return S->top == S->floor;
48 }
```

```
50 void push(stack* S, string x)
51 //@requires is_stack(S);
52 //@ensures is_stack(S);
53 {
54     list* L = alloc(list);
55     L->data = x;
56     L->next = S->top;
57     S->top = L;
58 }
```

```
60 string pop(stack* S)
61 //@requires is_stack(S);
62 //@requires !stack_empty(S);
63 //@ensures is_stack(S);
64 {
65     string e = S->top->data;
66     S->top = S->top->next;
67     return e;
68 }
```

2pts

3.2 We wish to add a new operation `stack_bottom` to our stack **implementation** from the previous part. Here's its interface prototype:

```
string stack_bottom(stack_t S)
/*@requires S != NULL && !stack_empty(S); @*/ ;
```

This operation returns (but does not remove) the bottom element of the stack. It does not modify the input stack at any point.

a. Write this function using the **doubly-linked list** implementation of stacks from the previous part. Be sure that your function is as efficient as possible. *(Remember that the linked list that represents the stack has a dummy node.)*

```
string stack_bottom(stack* S)
//@requires is_stack(S) && !stack_empty(S);
{







}
```

b. Next, write the function using the **singly-linked list** implementation of stacks from lecture. *(Recall that the only difference is that these lists lack the `prev` field.)*

```
string stack_bottom(stack* S)
//@requires is_stack(S) && !stack_empty(S);
{







}
```

c. What is the worst-case asymptotic complexity of each implementation assuming the input stack contains n elements?

Doubly-linked implementation: $O($____________________$)$

Singly-linked implementation: $O($____________________$)$

The rest of this question considers the updated implementation of stacks based on doubly-linked lists.

2pts

3.3 Now, consider a broken `is_stack` function for this new stack implementation.

```
bool is_segment(list* node1, list* node2) {
  if (node1 == NULL) return false;
  if (node1 == node2) return true;
  return is_segment(node1->next, node2);
}

bool is_stack(stack* S) {
  return S != NULL && is_segment(S->top, S->floor);
}
```

Draw a picture of a full stack data structure that

- uses `string` elements,
- contains at least 4 allocated `list_node` structs
- returns `true` from `is_stack` (it should pass the first assertion)

BUT fails the unit test below with a segfault or an assertion failure. **Don't use X anywhere**: give specific values for every field. Your diagram should depict pointers (possibly `NULL`) and `string` values.

Stack picture:

```
// Unit test that your example above should fail
int main() {
  stack* S = // Code that constructs the example above.
  assert(is_stack(S) && !stack_empty(S));  // This must pass
  string x = stack_bottom(S);  // doubly-linked list implementation
  string y = pop(S);
  while (!stack_empty(S)) {
    y = pop(S);
    assert(is_stack(S));
  }
  assert(string_equal(x, y));
  return 0;
}
```