

# 15-122: Principles of Imperative Computation, Spring 2020

## Written Homework 6

Due on Gradescope: Monday 24<sup>th</sup> February, 2020 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.
- *Preview*, the Mac's PDF viewer.
- *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*.

**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	3.5	4.5	12
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 assignable 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.**

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* p = parse_int(A->argv[0]);

    if ( _____ ) printint( _____ );
    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 ( _____ ) printint( _____ );
    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 list_header {
    list* start;
    list* end;
};
typedef struct list_header* linkedlist;
```

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 67 to prove both line 68 and line 69 safe.

```
67 //@assert is_segment(a, b); // ASSUME VALID
68 a->next = b;
69 list* l = a->next;
```

We don't allow you to say that, because line 68 didn't raise an error, `a` must not be `NULL` and therefore line 69 must be safe. (This kind of reasoning is error-prone in practice.)

Here's a mystery function:

```

42 void mystery(linkedlist a, linkedlist 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

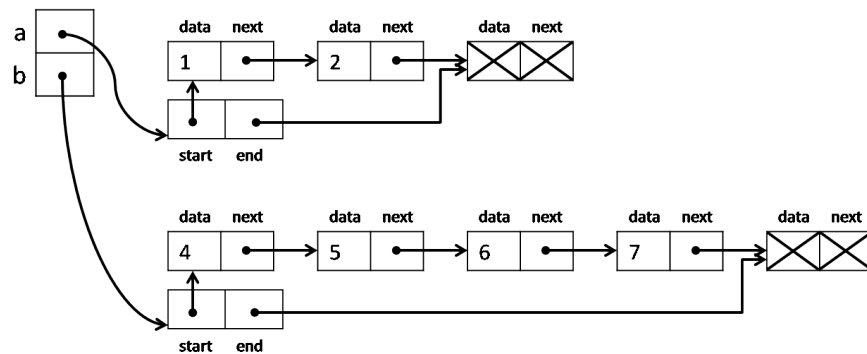
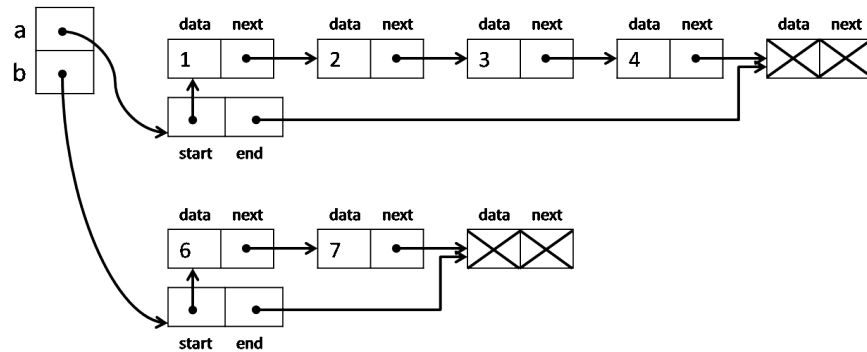
2.1 Explain why line 50 is safe: first, clearly state what the conditions for the safety of line 50 are, and second, explain why we know those lines are safe.

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

•  $m \geq n$

•  $m < n$

### 3. Doubly-Linked Lists

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

```
// typedef _____* stack_t;

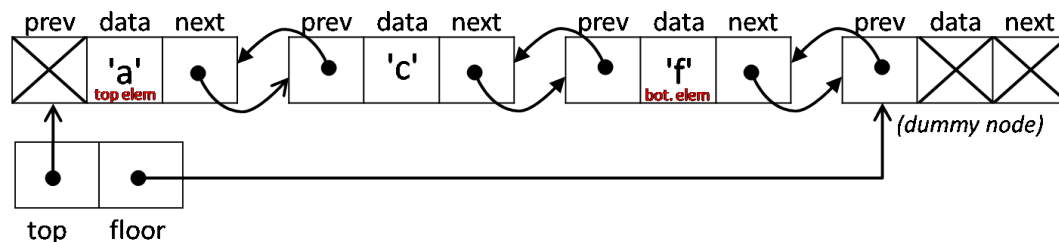
bool stack_empty(stack_t S)           /* 0(1) */
/*@requires S != NULL; @*/ ;

stack_t stack_new()                   /* 0(1) */
/*@ensures \result != NULL;      @*/
/*@ensures stack_empty(\result); @*/ ;

void push(stack_t S, char x)          /* 0(1) */
/*@requires S != NULL; @*/ ;

char pop(stack_t S)                   /* 0(1) */
/*@requires S != NULL;      @*/
/*@requires !stack_empty(S); @*/ ;
```

Suppose we decide to implement the stack (of `char`'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 {
    char data;
    list* prev;
    list* next;
};

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

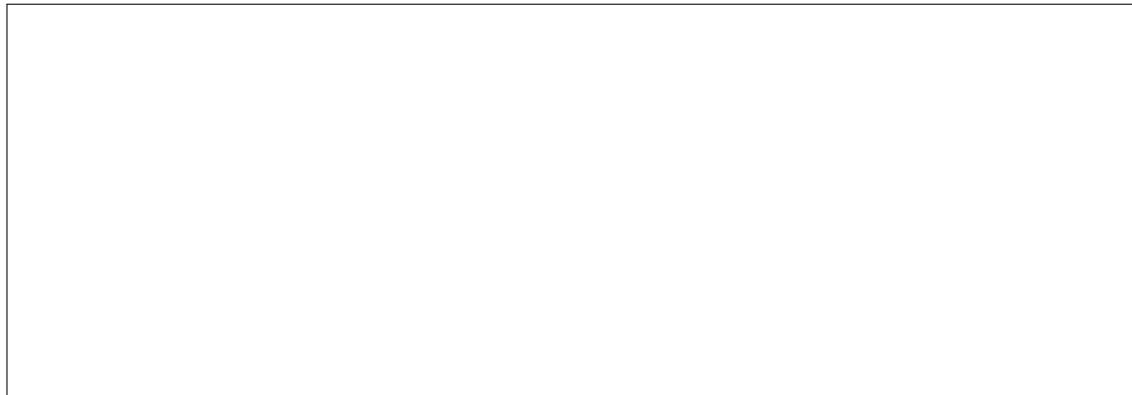
The top element of the stack (if any) will be stored in the first node of the list (pointed to by `top`), and the bottom element of the stack (if any) will be stored in the second-to-last node in the list, with the last node being a "dummy node" (pointed to by `floor`). Intuitively, the bottom element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

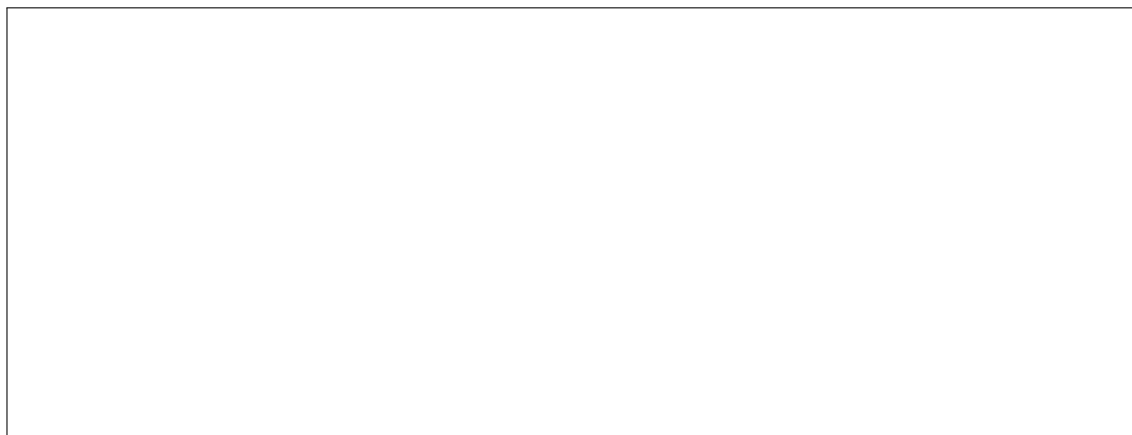
2pts

- 3.1 Modify the singly-linked list implementation of stacks 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.

```
42 stack* stack_new()
43 //@ensures is_stack(\result);
44 //@ensures stack_empty(\result);
45 {
46     stack* S = alloc(stack);
47     list* L = alloc(list);
48     S->top = L;
49     S->floor = L;
50     return S;
51 }
```



```
53 bool stack_empty(stack* S)
54 //@requires is_stack(S);
55 {
56     return S->top == S->floor;
57 }
```





```
59 void push(stack* S, char x)
60 //@requires is_stack(S);
61 //@ensures is_stack(S);
62 {
63     list* L = alloc(list);
64     L->data = x;
65     L->next = S->top;
66     S->top = L;
67 }
```

```
69 char pop(stack* S)
70 //@requires is_stack(S);
71 //@requires !stack_empty(S);
72 //@ensures is_stack(S);
73 {
74     char e = S->top->data;
75     S->top = S->top->next;
76     return e;
77 }
```

1pt

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

```
char stack_bottom(stack_t S)    /* 0(1) */
    /*@requires S != NULL && !stack_empty(S); @*/ ;
```

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

```
char stack_bottom(stack* S)
//@requires is_stack(S);
//@requires !stack_empty(S);
{
}
}
```

If we didn't add the `prev` link to each node, how long would it take to return the bottom element of the stack in big-O notation for a list of  $n$  elements? Why? (*Note that there is still a dummy node at the end of the linked list.*)

$O(\underline{\hspace{2cm}})$

Because \_\_\_\_\_

\_\_\_\_\_

1.5pts

3.3 Now, consider the following broken implementation of `is_stack` for this 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 complete picture of a stack data structure (with char elements) that contains at least 4 allocated `list_node` structs and that returns true from `is_stack` yet would not be well-formed. *Give specific values everywhere. Don't use Xs anywhere; they are for unspecified values. So your diagram should depict pointers (possibly NULL) and values of type char.* For full credit your example struct must fail the unit test below with a segfault or an assertion failure after passing the initial assertion.

Stack picture:

```
// Unit test that your example above should fail
int main() {
    stack* S = // Code that constructs the example above.
              // By necessity, this won't respect the interface
    assert(is_stack(S) && !stack_empty(S)); // This must pass
    char x = stack_bottom(S);
    char y = pop(S);
    while (!stack_empty(S)) {
        y = pop(S);
        assert(is_stack(S));
    }
    assert(x == y);
    return 0;
}
```