15-122 : Principles of Imperative Computation

Fall 2012

Assignment 3

(Theory Part)

Due: Thursday, October 4 at the **beginning** of lecture.

Name: ____________________

Andrew ID: ____________________

Recitation: ____________________

The written portion of this week's homework will give you some practice working with sorting algorithms, stacks and queue data structures and performing some asymptotic analysis tasks. 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.

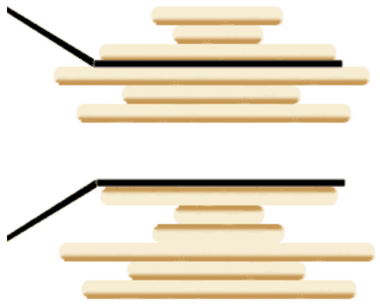
Question	Points	Score
1	8	
2	7	
Total:	15	

You *must* include this cover sheet.
Either type up the assignment using
15-122-theory3.tex, or print out this PDF.

1. **Pancake sort.** In lecture we talked about stacks, a data structure with a last-in, first-out behavior. This question deals with a much more delicious and *entirely unrelated* sort of stack: stacks of pancakes!

   Suppose that you have a stack of pancakes that you would like to sort from smallest to largest such that largest pancake is at the bottom of the stack after sorting. However, the only operation allowed on this data structure is flipping a portion of the stack over. To flip a portion of the stack, you choose a starting pancake and flip the entire portion of stack from that pancake up to the top, including the starting pancake.

   For example, here is what happens when the top 3 pancakes are flipped upside down. The first image shows the original pancake stack before flipping the top 3 pancakes. The second image shows the pancake stack after flipping the top 3 pancakes.



   Here is a possible algorithm for sorting a stack with `n` pancakes. You can assume that there is some abstract type `pancake_stack` that represents a stack of pancakes. We index the pancakes with pancake 0 on the top of the stack.

```
struct pancake {
    int size;
    string flavor;
};

void pancakesort(pancake_stack pancakes, int n)
{
    for (int i = n; i > 1; i--)
    {
        int max = findmax(pancakes, 0, i);
        flip(pancakes, max + 1);
        flip(pancakes, i);
    }
}
```

The abstract type `pancake_stack` implements the following interface:

```
int pancake_stack_size(pancake_stack pancakes)
//@ensures \result >= 0;
;

bool is_sorted(pancake_stack pancakes, int top, int bottom)
/*@requires 0 <= top && top <= bottom
    && bottom <= pancake_stack_size(pancakes);@*/
;

/* findmax returns the index of the largest size pancake that's
 * between indices top and bottom (not including bottom)
 */
int findmax(pancake_stack pancakes, int top, int bottom)
/*@requires 0 <= top && top < bottom
    && bottom <= pancake_stack_size(pancakes);*@/
//@ensures top <= \result && \result < bottom;
;

/* flip flips the substack containing the n top items over.
 * for example, flip(pancakes, 3) is in the picture above.
 */
void flip(pancake_stack pancakes, int n)
//@requires 0 <= n && n <= pancake_stack_size(pancakes);
;

/* le_seg returns true if and only if the size of p is less
 * than or equal to that of every pancake in the stack from
 * lower to upper (excluding upper)
 */
bool le_seg(struct pancake p, pancake_stack pancakes, int lower, int upper)
/*@requires 0 <= lower && lower <= upper
    && upper <= pancake_stack_size(pancakes);*@/
;

/* ge_seg returns true if and only if the size of p
 * greater than or equal to every pancake in the stack from
 * lower to upper (excluding upper)
 */
bool ge_seg(struct pancake p, pancake_stack pancakes, int lower, int upper)
/*@requires 0 <= lower && lower <= upper
    && upper <= pancake_stack_size(pancakes);@*/
;
```

```
/* le_segs returns true if and only if the size of everything in the
 * stack from lower1 to upper1 is less than or equal to the size of
 * everything in the stack from lower2 to upper2
 * (not including upper1 or upper2)
 */
bool le_segs(pancake_stack pancakes, int lower1, int upper1,
    int lower2, int upper2)
/*@requires 0 <= lower1 && lower1 <= upper1
    && upper1 <= pancake_stack_size(pancakes);@*/
/*@requires 0 <= lower2 && lower2 <= upper2
    && upper2 <= pancake_stack_size(pancakes);@*/
;

/* ge_segs returns true if and only if the size of everything in the
 * stack from lower1 to upper1 is greater than or equal to the size of
 * everything in the stack from lower2 to upper2
 * (not including upper1 or upper2)
 */
bool ge_segs(pancake_stack pancakes, int lower1, int upper1,
    int lower2, int upper2)
/*@requires 0 <= lower1 && lower1 <= upper1
    && upper1 <= pancake_stack_size(pancakes);@*/
/*@requires 0 <= lower2 && lower2 <= upper2
    && upper2 <= pancake_stack_size(pancakes);@*/
;
```

(1) (a) What is the tightest upper bound for the worst case asymptotic complexity of your algorithm using big O notation in terms of n where n is the number of pancakes? Explain your answer.

Assume that we have specialized state-of-the-art pancake-flipping hardware, so `flip(pancakes, max)` is a constant time operation ($O(1)$).
Also assume `findmax(pancakes, 0, i)` is a linear time operation ($O(i)$).

Solution: $O($

(3) (b) Insert the missing preconditions, postconditions and loop invariants in the code. You can use only the functions declared above for your contracts.

Solution:

```
void pancakesort(pancake_stack pancakes, int n)




{
    for (int i = n; i > 1; i--)






    {
        int max = findmax(pancakes, 0, i);
        flip(pancakes, max + 1);
        flip(pancakes, i);
    }
}
```

(1) (c) Prove that the loop invariants hold before the first iteration of the loop.

Solution:

(1) (d) Prove that the loop invariants are preserved (if the loop invariants are true before some iteration, they'll also be true after it).

Solution:

(1) (e) Prove that the loop invariants combined with the negated loop guard imply the postcondition. *Hint: This part should be straightforward if your loop invariants are strong enough.*

Solution:

(1) (f) Prove that the loop terminates.

Solution:

2. **Quacks.** *This question aims to give you more practice thinking about stacks and queues, thinking about some simple time complexity analysis, and ensuring correctness of code through contracts. We suggest that you write the code for this problem without using* `coin` *or* `cc0` *so that you can get practice writing code without a compiler, which you'll need to do on the midterms and final. We will accordingly be lenient on minor syntax errors in grading.*

   After learning about queues and stacks in 15-122, Fred Hacker decided he'd try to combine the data structures to get the best of both worlds.

   Consider the following interface for a data structure that Fred calls a `quack`. It accepts elements of the type `elem`. (Note: `elem` is just a generic type name, used to indicate that we can accept data of any type. If we wanted to put `int`s in the `quack` we could do `typedef int elem;` and if we wanted to put `string`s in we could do `typedef string elem;`.)

```
struct quack* quack_new();
bool quack_empty(struct quack *Q);
void enqueue(elem e, struct quack *Q);
elem dequeue(struct quack *Q);
void push(elem e, struct quack *Q);
elem pop(struct quack *Q);
```

   Fred decided to use a linked list to implement his `quack`. The linked list he used is defined with these `struct`s:

```
struct list_node {
    elem data;
    struct list_node *next;
}
typedef struct list_node node;
struct quack_header {
    node *head;
    node *tail;
}
typedef struct quack_header* quack;
```

Fred thinks he'll be able to make lots of money off of his `quack` data structure, so he's kept its implementation secret. All we know is that his implementation implements every every interface function of the `quack` in $O(1)$ time and that it uses the above `struct`s to store its data. Note that you may assume that `alloc` runs in constant time, even though this is not true in practice.

Your task is to implement Fred's `quack` before he can release it, so *you* can make any money that it's worth.

In your implementation, do not use an extra node at the end of the list as we've done in class. In other words, if `tail` is not `NULL`, then `tail -> data` should contain actual data that we want to store, and `tail -> next` should be `NULL`. If `tail` is `NULL`, then `head` must also be `NULL` and the `quack` is empty. In the case where there is only one thing in the `quack`, both `head` and `tail` should point to it.

You may assume you have the functions:

- `bool is_quack(quack Q)`, which checks that the given `quack` is valid (it makes sure it's not `NULL`, that everything in the `quack` is as it should be, that there are no cycles in the list, etc.)
- `quack quack_new()`, which generates a new, empty `quack`
- `int occurrences(elem e, quack Q)`, which returns the number of times `e` appears in `Q`
- `bool quack_equal(quack Q1, quack Q2)`, which returns true if and only if the two `quack`s are equal. Note that this requires us to have some function that compares values of type `elem` and tells us whether they are equal. You can assume you have a `bool elem_equal(elem e1, elem e2)` function that returns true if and only if `e1` and `e2` are equal.
- `elem peek_last(quack Q)`, which returns the last element of `Q` and does NOT modify it.
- `elem peek_first(quack Q)`, which returns the first element of `Q` and does NOT modify it.

(1) (a) Explain briefly why the struct definition for a `struct quack_header` will work and allow all of the interface functions defined on the top of page 8 to be done in $O(1)$ time.

Solution:

(1) (b) Write a `quack_empty` function that returns true if and only if the given `quack` is empty. Your function should not modify the `quack` that is passed in. Include all relevant annotations. `quack_empty` should run in $O(1)$ time, but you don't need to justify that it does.

Solution:

```
bool quack_empty(quack Q)



{


}
```

(2) (c) Write an `enqueue` function which, given a `quack Q`, adds the element `e` to the end of the quack. Use all necessary annotations. Your function should run in $O(1)$ time and you should include a short justification explaining why it does.

Solution:

```
void enqueue(elem e, quack Q)





{















}
```

(1) (d) Write a `dequeue` function which, given a nonempty `quack`, removes the first element and returns it. Use all necessary annotations. Your function should run in $O(1)$ time, and you should include a short justification explaining why it does.

Solution:

```
elem dequeue(quack Q)




{




}
```

(1) (e) Write a `push` function which, given a `quack` and an element `e`, adds `e` to the front of the quack. Use all necessary annotations. Your function should run in $O(1)$ time, and you should include a short justification explaining why it does.

Solution:

```
void push(elem e, quack Q)



{









}
```

(1) (f) Write a `pop` function which, given a nonempty `quack`, pops an element off the front and returns it. Use all necessary annotations. Your function should run in $O(1)$ time, and you should include a short justification explaining why it does. *Hint: it should be possible to write this function in one simple line of code using functions you have already written.*

Solution:

```
elem pop(quack Q)


{


}
```