15-122 : Principles of Imperative Computation, Spring 2013

Homework 3 Theory [Update 1]

Due: Thursday, February 21, 2013, 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 queues. 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	5	
2	11	
3	9	
Total:	25	

You *must* use this printout, include this cover sheet,
and staple the whole thing together before turning it in.
Either type up the assignment using 15122-theory3.tex,
or print this PDF and write your answers *neatly* by hand.

1. **Stacks and Queues**

   This question focuses on using interface functions for stacks and queues. The division between an implementation (the details of a data structure's low-level details) and an interface (how a client is able to interact with the data structure) is one of the most crucial distinctions in computer science.

   Consider the following interfaces for `queue` and `stack` that store elements of the type `elem`:

```
/* Queue Interface */
queue queue_new();              /* O(1) */
bool queue_empty(queue Q);      /* O(1) */
void enqueue(elem e, queue Q); /* O(1) */
elem dequeue(queue Q)           /* O(1) */
  //@requires !queue_empty(Q);
  ;

/* 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);
  ;
```

   For this question, assume that we do not know how these data structures are implemented. That is, we don't know if the programmer used arrays or linked lists, or something else—just that they somehow implemented the functions shown above, and that they have the same observable behavior as a typical stack or queue.

   As a client, you can only use the interface functions specified above to interact with these data structures.

(2) (a) Write a function `stack_reverse(stack S)` that uses a *queue* to reverse the order of elements on the stack S that is passed in.

Include proper annotations in your code.

Solution:

```
void stack_reverse(stack S)



{









}
```

(3) (b) Write a function `stack_reverse(stack S)` that uses *stacks* to reverse the order of elements on the stack S that is passed in.

Include proper annotations in your code.

Solution:

```
void stack_reverse(stack S)



{









}
```

2. **Quacks.** *This question focuses on writing code to implement a particular data structure according to some given specifications. This involves making the implementation:*

   1. *Generic (it does not make any assumptions regarding the data the client wishes to place in the data structure)*
   2. *Accurate (it does not violate any of the data structures properties)*
   3. *Safe (it rejects any malformed input that the client may enter)*

   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`.

```
bool quack_empty(quack Q);             /* O(1) */
bool quack_new();                      /* O(1) */
void enqueue(elem e, quack Q);         /* O(1) */
elem dequeue(quack Q);                 /* O(1) */
void push(elem e, quack Q);            /* O(1) */
elem pop(quack Q);                     /* O(1) */
bool quack_equal(quack Q1, quack Q2); /* O(n) */
```

   Quacks accept elements of the type `elem`, which is just a generic type name – if we wanted to put `int`s in the `quack` we would use `typedef int elem`. The actual type of `elem` is provided by the *client*; the implementation of quacks should *not* rely upon the concrete details of `elem`.

   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;
```

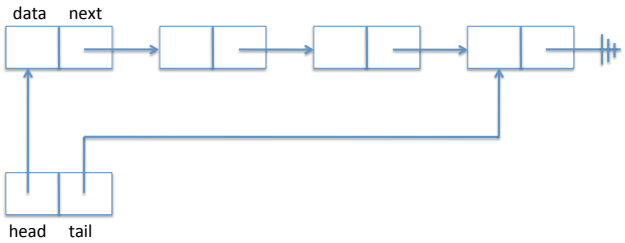
   Fred thinks he'll be able to make lots of money off of his `quack` data structure, so he's kept its implementation secret. All we know is that his implementation uses the above `struct`s to store its data and that each function has the runtime specified in the interface, that it acts like a stack when only `push` and `pop` are used, and that it acts like a queue when only `enqueue` and `dequeue` are used.

   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 for our implementation of queues and stacks.

1. 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`.
2. If `tail` is `NULL`, then `head` must also be `NULL` and the `quack` is empty.
3. In the case where there is only one thing in the `quack`, *both* `head` and `tail` should point to it.

Your quack implementation, in other words, will look like this when there are four elements in the quack. (The `head` and `tail` must be assigned as shown below for relevant quack operations to have O(1) time; you should consider why this must be the case.)



You may assume you have the functions, which you may use for any pre/post conditions:

- `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, that `Q->head` is `NULL` if `Q->tail` is `NULL` and vice-versa, etc.)
- `bool elem_equal(elem e1, elem e2)`, which compares two elements of type `elem` and checks if they are equal or not. This is a *client-side function*, along with the type `elem`.
- `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.

In the following sub-parts, you will write the code necessary to implement quacks as per the description given above. Bear the following points in mind:

1. You must include all necessary pre-conditions/post conditions to guarantee that the operations work as specified above. Refer to the code given in lecture on the implementation of stacks and queues to get idea of what contracts to write.
2. Your functions must have the runtime performance specified in the interface.
3. Try to write the code on paper, with the aid of diagrams, instead of using `coin` or `c0`, as you will be expected to write code on the exams without access to either of these tools.

(1) (a) Write the `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.*

Solution:

```
bool quack_empty(quack Q)



{



}
```

(1) (b) Write a `quack_new` function that returns a valid, empty `quack`

Solution:

```
quack quack_new()

{







}
```

(2) (c) Write an `enqueue` function which, given a `quack Q`, adds the element `e` to the `tail` of the `quack`. Include a contract that guarantees that the element `e` has been added to the end of the quack.

Solution:

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



{









}
```

(2) (d) Write a `dequeue` function which, given a non-empty `quack`, removes the element at the `head` of the `quack` and returns it.

Solution:

```
elem dequeue(quack Q)



{









}
```

(2) (e) Write a `push` function which, given a `quack` and an element `e`, adds `e` to the `head` of the `quack`. Include a contract that guarantees that the element `e` has been added to the front of the quack.

Solution:

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


{









}
```

(1) (f) Write a `pop` function which, given a nonempty `quack`, pops an element off the `head` of the `quack` and returns it.
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)


{



}
```

(2) (g) Write a `quack_equal` function which checks if two quacks are equal (ie they have the same number of elements and the elements are equal and are arranged in the same order.

Your function should not assume anything about the type of data stored in the quacks. You can assume that both quacks have data of type `elem`*. Your function may NOT modify the original quacks passed in*

Solution:

```
bool quack_equal(quack Q1, quack Q2)
{
}
```

3. **Mergesort** This question concerns the implementation of `mergesort`; an implementation is given below. The effect of `copy_array` in line 49 is to copy the array `B`$[0, \mathit{length})$ into `A`$[\mathit{lower}, \mathit{upper})$. `A` is changed so that `A[lower + i] == B[i]` for each `i < length`.

```
1 void mergesort (int[] A, int lower, int upper)
2 //@requires 0 <= lower && lower <= upper && upper <= \length(A);
3 //@ensures is_sorted(A, lower, upper);
4 {
5  if (upper-lower <= 1) return;
6  int mid = lower + (upper-lower)/2;
7
8  mergesort(A, lower, mid);
9  //@assert is_sorted(A, lower, mid);
10
11 mergesort(A, mid, upper);
12 //@assert is_sorted(A, mid, upper);
13
14 merge(A, lower, mid, upper);
15 return;
16 }
17
18 void merge (int[] A, int lower, int mid, int upper)
19 /*@requires 0 <= lower && lower <= mid
20       && mid <= upper && upper <= \length(A);*/
21 //@requires is_sorted(A, lower, mid);
22 //@requires is_sorted(A, mid, upper);
23 //@ensures is_sorted(A, lower, upper);
24 {
25  int length = upper - lower;
26  int[] B = alloc_array(int, length);
27  int i1 = lower;
28  int i2 = mid;
29
30  for(int k = 0; k < length; k++)
31   //@loop_invariant 0 <= k && k <= length;
32   //@loop_invariant lower <= i1 && i1 <= mid;
33   //@loop_invariant mid <= i2 && i2 <= upper;
34   //@loop_invariant (i1 - lower) + (i2 - mid) == k;
35   //@loop_invariant is_sorted(B, 0, k);
36   /*@loop_invariant k == 0 ||
37                     ((i1 == mid || A[i1] >= B[k-1])
38                      && (i2 == upper || A[i2] >= B[k-1]);*/
39   {
40    if(i2 == upper || (i1 < mid && A[i1] <= A[i2])) {
41     B[k] = A[i1];
42     i1++;
43    } else {
44     B[k] = A[i2];
45     i2++;
46    }
47   }
48  //@assert is_sorted(B, 0, length);
49  copy_array(A, lower, upper, B, 0, length);
50 }
```

In part (a) of this question we will reason operationally about the `mergesort`, and in part (b) we will reason logically about safety.

(4) (a) Suppose we have the array `A[]` below, and call `mergesort(A, 0, 8)`. Step through the execution of `mergesort(A, 0, 8)`, and write down the contents of `A[]` after each `merge` call terminates. Thus, the first line should contain `A[]` immediately after the first `merge` call terminates, the second line should contain `A[]` immediately after the second `merge` call terminates, and so on. Note that `mergesort` does not always call `merge`, and that even when `merge` is called on a smaller subarray of `A[]`, you should fill in the **entire** contents of `A[]`. Finally, the number of lines provided may be more than the number of times `merge` is called; any excess lines should be left blank.

`merge` #	`A[]`							
	7	5	0	3	4	1	2	6
1								
2								
3								
4								
5								
6								
7								
8								
9								

(5) (b) The `merge` function contains both a number of array accesses. For the following array accesses, list the facts that establish the safety of that array access, as well as the line that immediately provides each of those facts. You do not need to give the intermediate reasoning, just the raw facts that would allow you to establish safety and the lines of code that immediately justify those facts. Do not include any extra or unnecessary facts.

Solution:

- Line 37, `A[i1]`

  `0 <= lower` (line 19)
  `lower <= i1` (line 32)
  `i1 <= mid` (line 32)
  `i1 != mid` (line 37)
  `mid <= upper` (line 20)
  `upper <= \length(A)` (line 20)

- Line 37, `B[k-1]`

- Line 38, `A[i2]`

- Line 38, `B[k-1]` (same as the justification from line 37)

- Line 40, `A[i1]`

- Line 40, `A[i2]`

  `0 <= lower && lower <= mid` (line 19)
  `mid <= i2 && i2 <= upper` (line 33)
  `i2 != upper` (line 40)
  `upper <= \length(A)` (line 20)

- Line 41, `B[k]`

  `0 <= k` (line 31)
  `k < length` (line 30)
  `length = \length(B)` (line 26)

- Line 41, `A[i1]`

- Line 44, `B[k]` (same as the justification from line 41)

- Line 44, `A[i2]`