15-122 : Principles of Imperative Computation, Spring 2016

Written Homework 7

Due: Monday 29$^{\text{th}}$ February, 2016

Name: ______________________________

Andrew ID: ______________________________

Section: ______________________________

This written homework covers amortized analysis and hash tables.

Most of this PDF is editable. You can either type your answers in the red boxes/lines, or you can write them *neatly* by hand.

Print out your submission double-sided and staple pages in order.

The assignment is due by 5:30pm on Monday 29$^{\text{th}}$ February, 2016.

You can hand in the assignment during lab
or in the box outside of the lab room.

You must hand in your homework yourself;
do not give it to someone else to hand in.

1. **Reasoning with 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.

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: 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.

```
 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, *only* that dereference is okay. So, in the example below, we have to use line 81 to prove both line 82 and line 83 safe.

```
81 //@assert is_segment(a, b);
82 a->next = b;
83 list* l = a->next;
```

We don't allow you to say that, because line 82 didn't raise an error, `a` must not be `NULL` and therefore 83 must be safe. (This kind of reasoning is error-prone in practice.)

Here's a mystery function:

```
11 void mystery(linkedlist a, linkedlist b)
12 //@requires a != NULL;                         // NONE
13 //@requires b != NULL;                         // NONE
14 //@requires is_segment(a->start, a->end);      // ____________
15 //@requires is_segment(b->start, b->end);      // ____________
16 {
17   list* t1 = a->start;                         // ____________
18   list* t2 = b->start;                         // ____________
19   while (t1 != a->end && t2 != b->end)         // ____________
20   //@loop_invariant is_segment(t1, a->end);    // ____________
21   //@loop_invariant is_segment(t2, b->end);    // ____________
22   {
23     list* t = t2;                              // ____________
24     t2 = t2->next;                             // ____________
25     t->next = t1->next;                        // ____________
26     t1->next = t;                              // ____________
27     t1 = t1->next->next;                       // ____________
28    }
29    b->start = t2;                              // ____________
30 }
```

1pt

(a) Explain why line 19 is safe: first, clearly state what the conditions for the safety of line 19 are, and second, explain why we know those lines are safe.

1pt

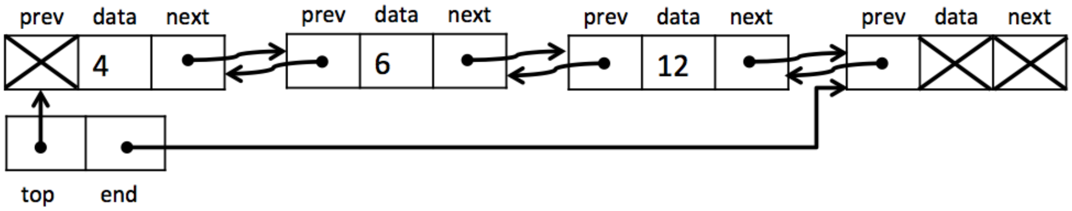
(b) Why can we not use the combination of line 14 (which tells us that `a->start` is not `NULL`) and line 17 (which tells us that `t1` is `a->start`) to reason that `t1` is not `NULL` and therefore that line 26 is safe? Why do we actually know line 26 is safe?

2. **Doubly-Linked Lists**

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

```
typedef struct stack_header* 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, elem x) /* O(1) */
  /*@requires S != NULL; @*/;

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

Suppose we decide to implement the stack (of integers) 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 {
   elem data;
   list* prev;
   list* next;
};

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

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: 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 bottom.

2pts

(a) 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.

```
1  stack* stack_new()
2  //@ensures is_stack(\result);
3  //@ensures stack_empty(\result);
4  {
5     stack* S = alloc(struct stack_header);
6     list* L = alloc(struct list_node);
7     S->top = L;
8     S->bottom = L;
9     return S;
10 }
```

```
11 bool stack_empty(stack* S)
12 //@requires is_stack(S);
13 {
14    return S->top == S->bottom;
15 k}
```

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

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

1pt

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

```
    elem stack_bottom(stack_t S) /* O(1) */
      /*@requires S != NULL && !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. *(Remember that the linked list that represents the stack has a dummy node.)*

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



}
```

1pt (c) 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->bottom);
}
```

Draw a complete picture of a stack data structure (with integer 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 integers.* 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
   elem x = stack_bottom(S);
   elem y = pop(S);
   while (!stack_empty(S)) {
      y = pop(S);
      assert(is_stack(S));
   }
   assert(x == y);
   return 0;
}
```

3. Remove Operation For Unbounded Arrays

The `arr_add` operation adds an element to the end of an unbounded array. Conversely, the `arr_rem` operation removes the element at the end. (Remember that the “end” of the array is from the client’s perspective. There are additional unused positions in the array from the implementation’s perspective.) When removing, we don’t need to resize the array to a smaller size, but we could. However, we need to consider *when* to shrink the array in order to guarantee O(1) amortized runtime.

1pt

(a) If the array resizes to be twice as large as soon as it is full (as in lecture), and resizes to be half as large as soon as it is strictly less than half full, give a sequence of additions and removals, starting from a new array `A` of size 3 (limit 6), that will cause worst-case behavior. End your solution end with “. . . ” after you clearly establish the repeating behavior, and after each operation write the size, limit, and number of array writes for that operation. The first line of the answer is shown.

```
                     // size = 3, limit = 6
arr_add(A, "x");     // size = 4, limit = 6, 1 array write
```

1pt

(b) Generalizing, with the strategy above, what is the worst case runtime complexity, using big-O notation, of performing k operations on an array of size n, where each operation is taken from the set {`arr_add`, `arr_rem`} ?

$O($________________$)$

We haven’t done the type of amortized analysis for this strategy that you saw in lecture (accounting for operations with tokens). However, we can say that the amortized cost of each operation is found by dividing the total cost by the number of operations. (This is known as “aggregate” analysis.)

Using aggregate analysis, what is the amortized cost of each of the k operations in the worst case?

$O($________________$)$

1pt (c) Instead of resizing the array to be half as large as soon as it is strictly less than half full, we could resize the array to half of its current size when it is *exactly* a quarter full. This will lead to O(1) amortized cost per remove operation. Using an array of size 11 (limit 12), show the effect of an add operation followed by the sequence of remove operations that causes the array to resize. As before, show the size and limit of the array after each operation, and indicate how many array writes each step takes. The first two lines are given for you.

```
                    // size = 11, limit = 12
arr_add(A, "x"); // size = 12, limit = 24, 13 array writes
arr_rem(A);         // size = 11, limit = 24, 0 array writes
```

In the answer above, the initial `arr_add` operation doubled the size of the array, consuming any banked tokens. Based on your answer above, what is the minimum number of tokens that should be charged for each `arr_rem` operation so that enough tokens are banked for the resize of the array? In your analysis, the *only* thing we have to pay for with tokens is array writes.

______________ token(s).

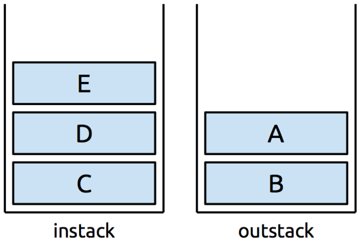
4. A New Implementation of Queues

Recall the interface for a stack that stores elements of the type `elem`:

```
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, elem x)   /* O(1) */
  /*@requires S != NULL; @*/;

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

For this question you will analyze a different implementation of the queue interface. Instead of a linked list, this implementation uses two stacks, called `instack` and `outstack`. To enqueue an element, we push it on top of the `instack`. To dequeue an element, we pop the top off of the `outstack`. If the `outstack` is empty when we try to dequeue, then we will first move all of the elements from the `instack` to the `outstack`, then pop the `outstack`.

For example, below is one possible configuration of a two-stack queue containing the elements `A` through `E` (`A` is at the front and `E` is at the back of the abstract queue):



We will use the following C0 code:

```
typedef struct stackqueue_header stackqueue;
struct stackqueue_header {
  stack_t instack;
  stack_t outstack;
};
```

```
bool is_stackqueue(stackqueue* Q)
{
  return Q != NULL && Q->instack != NULL && Q->outstack != NULL;
}

stackqueue* queue_new()
//@ensures is_stackqueue(\result);
{
  stackqueue* Q = alloc(stackqueue);
  Q->instack = stack_new();
  Q->outstack = stack_new();
  return Q;
}
```

1pt

(a) Given a queue with k elements in it, exactly how many different ways can this queue be represented using two stacks, as a function of k?

________________ way(s).

3pts

(b) Write the function queue_empty that returns true if the queue is empty. Your answer must be based on the description of the data structure above.

```
bool queue_empty(stackqueue* Q)
//@requires is_stackqueue(Q);
//@ensures is_stackqueue(Q);
{

}
```

Write the function enq based on the description of the data structure above.

```
void enq(stackqueue* Q, elem x)
//@requires is_stackqueue(Q);
//@ensures is_stackqueue(Q);
{

}
```

Write the function deq based on the description of the data structure above.

```
elem deq(stackqueue* Q)
//@requires is_stackqueue(Q);
//@requires !queue_empty(Q);
//@ensures is_stackqueue(Q);
{

}
```

1pt (c) We now determine the runtime complexity of the `enq` and `deq` operations. Let k be the total number of elements in the queue.

What is the worst-case runtime complexity of each of the following queue operations based on the description of the data structure implementation given above? Write ONE sentence that explains each answer.

`enq`: $O($______________$)$

`deq`: $O($______________$)$

1pt (d) Using amortized analysis, we can show that the worst-case complexity of a *valid sequence* of n queue operations is $O(n)$. This means that the amortized cost per operation is $O(1)$, even though a single operation might require more than constant time.

In this case, a *valid sequence* of queue operations must start with the empty queue. Each operation must be either an enqueue or a dequeue. Assume that push and pop each consume one token.

How many tokens are required to enqueue an element? **State for what purpose each token is used.** Your answer should be a constant integer – the amortized cost should be $O(1)$.

How many tokens are required to dequeue an element? Once again, you must **state for what purpose each token is used.** Your answer should be a constant integer – the amortized cost should be $O(1)$.

5. **Hash Tables: Dealing with Collisions**

In a hash table, when two keys hash to the same location, we have a *collision*. There are multiple strategies for handling collisions:

- **Separate chaining**: each location in the table stores a chain (typically a linked list) of all keys that hashed to that location.
- **Open addressing**: each location in the table stores a key directly. In case of a collision when inserting, we *probe* the table to search for an available storage location. Similarly, in case of a collision when looking up a key k, we probe to search for k. Suppose our hash function is h, the size of the table is m, and we are attempting to insert or look up the key k:
    - *Linear probing*: on the i^{th} attempt (counting from 0), we look at index $(h(k)+i) \mod m$.
    - *Quadratic probing*: on the i^{th} attempt (counting from 0), we look at index $(h(k)+i^2) \mod m$.

For insertion, we are searching for an empty slot to put the key in. For lookup, we are trying to find the key itself.

1pt

(a) You are given a hash table of size m with n inserted keys that resolves collisions using separate chaining. If $n = 2m$ and the keys are *not* evenly distributed, what is the worst-case runtime complexity of searching for a specific key using big O notation?

$O($______________$)$

Under the same conditions, except that now the keys *are* evenly distributed, what is the worst-case runtime complexity of searching for a specific key using big O notation?

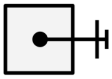
$O($______________$)$

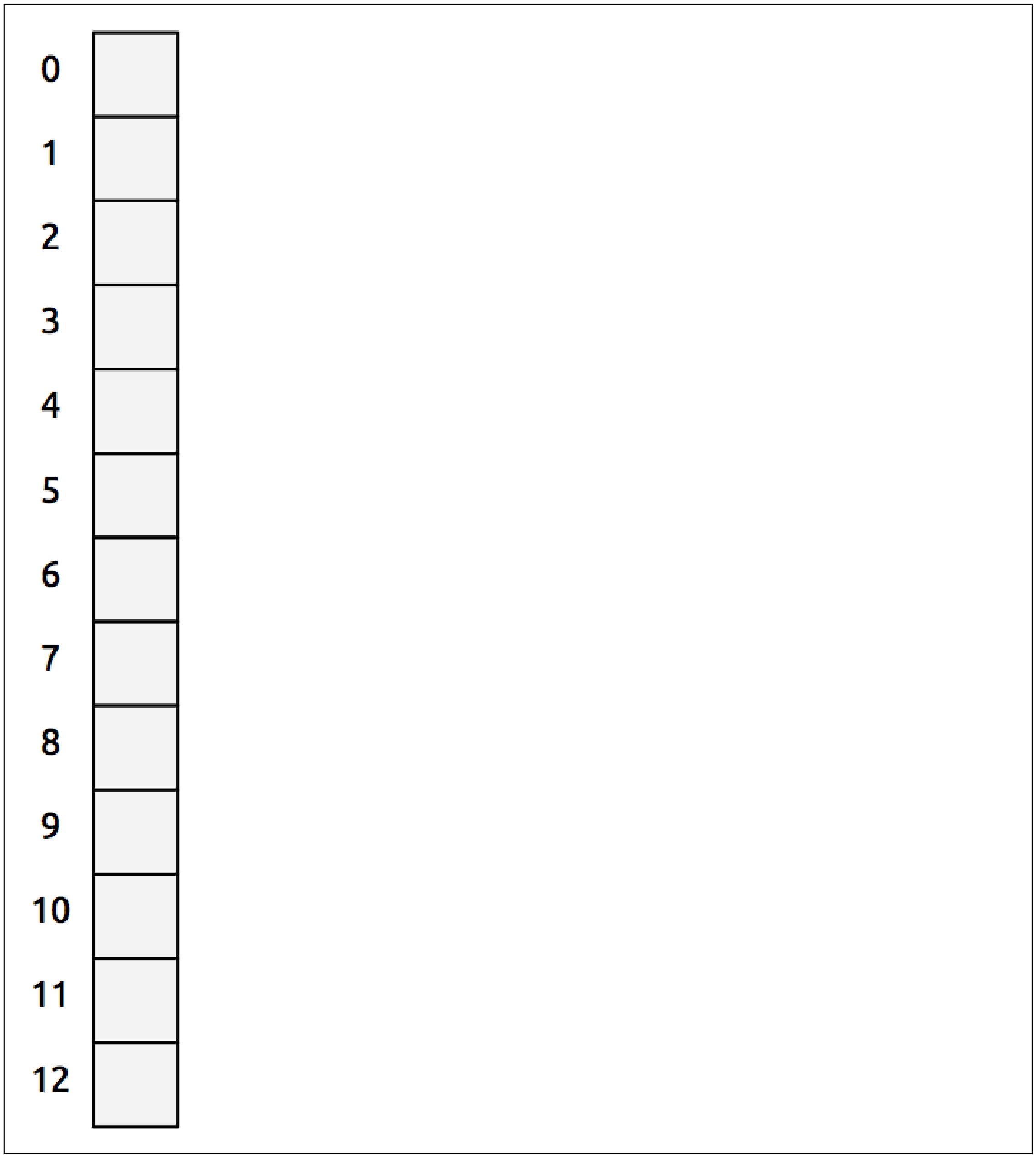
As usual, for both of the answers above, give the tightest, simplest bound.

For the next three questions, you are given a hash table of capacity $m = 13$. The hash function is $h(k) = k$; after hashing we attempt to insert the key k at array index $h(k) \bmod m$.

1pt (b) Assume the table resolves collisions using separate chaining and show how the set of keys below will be stored in the hash table by drawing the *final* state of each chain of the table after all of the keys are inserted, one by one, in the order shown.

$$67,\ 23,\ 54,\ 88,\ 39,\ 75,\ 49,\ 5$$

Wherever they occur, you should indicate `NULL` pointers explicitly by the notation from class: .



1pt (c) Show where the sequence of keys shown below are stored in the same hash table if they are inserted one by one, in the order shown, using linear probing to resolve collisions.

67, 23, 54, 88, 39, 75, 49, 5

0	1	2	3	4	5	6	7	8	9	10	11	12

1pt (d) Show where the sequence of keys shown below are stored in the same hash table if they are inserted one by one, in the order shown, using quadratic probing to resolve collisions.

67, 23, 54, 88, 39, 75, 49, 5

0	1	2	3	4	5	6	7	8	9	10	11	12

1pt (e) Quadratic probing suffers from one problem that linear probing does not. In particular, given a non-full hashtable, insertions with linear probing will always succeed, while insertions with quadratic probing might not (i.e. they may never find an open spot to insert).

Using $h(k) = k$ as your hash function and $m = 7$ as your table capacity, give an example of a table with load factor below 2/3 and a key that cannot be successfully inserted into the table. (*Hint:* start entering different multiples of 7.)

0	1	2	3	4	5	6

Key that cannot successfully be inserted: ______________________