15-122 : Principles of Imperative Computation, Fall 2015

Written Homework 7

Due: Monday, October 19 2015 by 6:00 PM

Name: _____

Andrew ID: _____

Section: _____

This written homework covers amortized analysis and hash tables.

Print out this PDF double-sided, **staple** pages in order, and write your answers on these pages *neatly*.

The assignment is due by 6:00 PM on Monday, October 19 2015.

You can hand in the assignment to your TA during lab or in the box outside of GHC 4117 (in the CS Undergraduate Program suite). WARNING: The box is removed promptly at 6PM.

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

(This page intentionally left blank. Remember to print double-sided!)

Question	Points	Score
1	3	
2	7	
3	5	
Total:	15	

1. **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.

Solution:

```
                          // 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`} ?

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?

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.

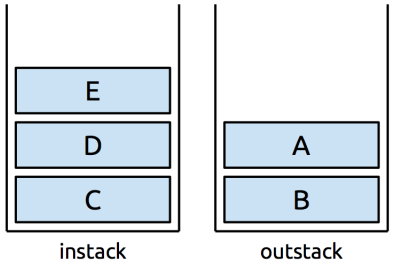
2. **A New Implementation of Queues**

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?

Solution:

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`: $O($

2pts (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 $O(1)$.

3. **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?

Solution: $O($ $)$

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?

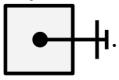
Solution: $O($ $)$

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

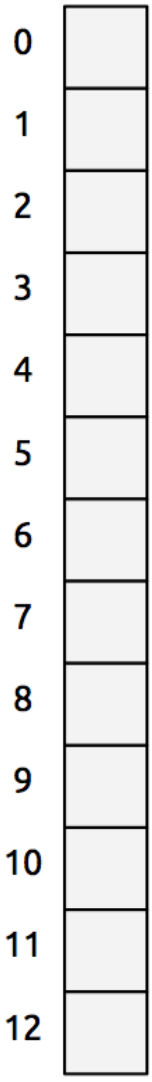
For the next three questions, you are given a hash table of capacity $m = 13$. The hash function is $h(k) = k$; after hashing we attempt to insert the key k at array index $h(k)$ mod m.

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:  .

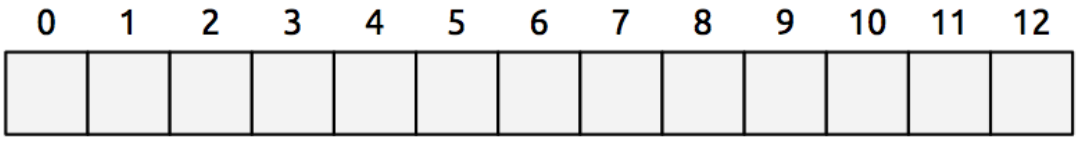
Solution:



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

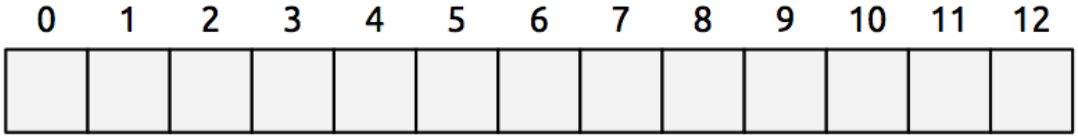
Solution:



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

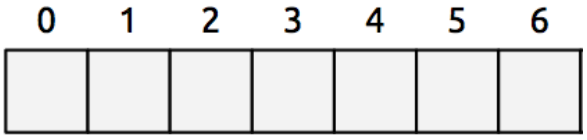
Solution:



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.)

Solution:



Key that cannot successfully be inserted: _____