

# 15-122: Principles of Imperative Computation, Spring 2024

## Written Homework 7

Due on [Gradescope](#): Monday 4<sup>th</sup> March, 2024 by 9pm

Name: \_\_\_\_\_

Andrew ID: \_\_\_\_\_

Section: \_\_\_\_\_

This written homework covers amortized analysis and hash tables.

**Preparing your Submission** You can prepare your submission with any PDF editor that you like. Here are a few that prior-semester students recommended:

- [Kami](#), [Adobe Acrobat Online](#), or [DocHub](#), some web-based PDF editors that work from anywhere.
- *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*.

**Please do not add, remove or reorder pages.**

**Caution** Recent versions of Preview on Mac are buggy: annotations get occasionally deleted for no reason. **Do not use Preview as a PDF editor.**

**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	4	Total
Points:	3.5	5	5	1.5	15
Score:					





## 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)           /* 0(1) */
/*@requires S != NULL; @*/;

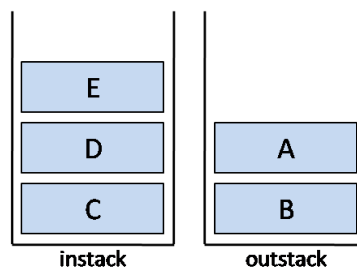
stack_t stack_new()                   /* 0(1) */
/*@ensures \result != NULL; @*/
/*@ensures stack_empty(\result); @*/;

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

elem pop(stack_t S)                   /* 0(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;
};

typedef stackqueue* queue_t;
```

```
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;
}

bool queue_empty(stackqueue* Q)
//@requires is_stackqueue(Q);
{
    return stack_empty(Q->instack) && stack_empty(Q->outstack);
}

void enq(stackqueue* Q, elem x)
//@requires is_stackqueue(Q);
//@ensures is_stackqueue(Q);
{
    push(Q->instack, x);
}

elem deq(stackqueue* Q)
//@requires is_stackqueue(Q);
//@requires !queue_empty(Q);
//@ensures is_stackqueue(Q);
{
    if (stack_empty(Q->outstack)) {
        while (!stack_empty(Q->instack))
            push(Q->outstack, pop(Q->instack));
    }
    return pop(Q->outstack);
}
```

0.5pts

2.1 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).
---------------

2pts

2.2 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:  $O(\underline{\hspace{2cm}})$ , because  $\underline{\hspace{2cm}}$

$\underline{\hspace{2cm}}$

deq:  $O(\underline{\hspace{2cm}})$ , because  $\underline{\hspace{2cm}}$

$\underline{\hspace{2cm}}$

2.5pts

2.3 Using amortized analysis, we can show that the worst-case complexity of a *valid sequence* of  $n$  enqueue/dequeue operations starting from an empty queue is  $O(n)$ . This means that the amortized cost per operation is  $O(1)$ , even though the worst-case cost of an individual operation may not be constant.

Here, a *valid sequence* of queue operations must start with the empty queue, each operation must be either an enq or a deq, and you must have enough tokens. Assume that push and pop each consume one token (and emptiness tests are free).

How many tokens should be charged to enqueue an element? How many to dequeue an element? Give the smallest possible number of tokens so that the amortized cost is  $O(1)$ . Justify each answer by briefly stating for what purpose of each token is used in each operation (*you may not need all lines*).

Cost of enq:  $\underline{\hspace{2cm}}$  token(s), to be used as follows:

1 token to  $\underline{\hspace{2cm}}$

1 token to  $\underline{\hspace{2cm}}$

1 token to  $\underline{\hspace{2cm}}$

1 token to  $\underline{\hspace{2cm}}$

Cost of deq:  $\underline{\hspace{2cm}}$  token(s), to be used as follows:

1 token to  $\underline{\hspace{2cm}}$

1 token to  $\underline{\hspace{2cm}}$

1 token to  $\underline{\hspace{2cm}}$

1 token to  $\underline{\hspace{2cm}}$

### 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 capacity of the table is  $m$ , and we are attempting to insert or look up the key  $k$ :
  - *Linear probing:* on the  $i^{\text{th}}$  attempt (counting from 0), we look at index  $(h(k) + i) \bmod m$ .
  - *Quadratic probing:* on the  $i^{\text{th}}$  attempt (counting from 0), we look at index  $(h(k) + i^2) \bmod m$ .

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

- 3.1 You are given a hash table of capacity  $m$  with  $n$  inserted keys. Collisions are resolved 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?

$O(\rule{1.5cm}{0.4pt})$

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?

$O(\rule{1.5cm}{0.4pt})$


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

3.2 Assume the table resolves collisions using separate chaining. 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

You may insert keys either always at the beginning or always at the end of a chain, but be consistent. Wherever they occur, you should indicate NULL pointers explicitly: either draw  or write NULL inside the box.

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



In the next two tasks, recall that the capacity of the table is  $m = 13$ .

1pt

- 3.3 Show where the keys in the sequence 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. Leave cells with no data blank.

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

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

1pt

- 3.4 Show where the keys in the sequence 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. Leave cells with no data blank.

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

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

1pt

- 3.5 Quadratic probing suffers from one problem that linear probing does not. Given a non-full hash table, 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  $m = 8$  as your table capacity, give an example of a table with load factor not above  $2/3$  and a key that cannot be successfully inserted into the table. (*Hint*: start entering different multiples of 8.) Leave cells with no data blank.

0	1	2	3	4	5	6	7

Key that cannot successfully be inserted: \_\_\_\_\_

## 4. Strings as Keys

In a popular programming language, non-empty strings are hashed using the following function:

$$H(s) = (s_0 \times 31^{|s|-1} + s_1 \times 31^{|s|-2} + \dots + s_{|s|-2} \times 31^1 + s_{|s|-1} \times 31^0) \% m$$

where  $s_i$  is the ASCII code (<http://www.asciitable.com/>) for the  $i^{\text{th}}$  character of string  $s$  (starting at 0 and counting from left to right),  $|s|$  is the length of  $s$ , and  $m$  is the size of the hash table. We consider 7-bit ASCII codes, which are the characters with ASCII value between 0 and 127 (included).

0.5pts

- 4.1 If 15217 strings were stored in a hash table of size 412 using separate chaining, what would the load factor of the table be? If the strings above were equally distributed in the hash table, what does the load factor tell you about the possible lengths of the chains?

The load factor is \_\_\_\_\_

Possible length(s) for each chain \_\_\_\_\_

1pt

- 4.2 Using the hash function above with a table size of 412, give an example of two different strings  $s_1$  and  $s_2$  that would “collide” in the hash table and would be stored in the same chain. Show your work. Use strings of up to 2 **printable** characters.

$s_1 =$  \_\_\_\_\_ (type or write clearly and unambiguously)

$s_2 =$  \_\_\_\_\_ (type or write clearly and unambiguously)

collide because