15-122 : Principles of Imperative Computation, Summer 1 2014

Written Homework 4

Due: Thursday, June 12 before recitation

Name: _____

Andrew ID: _____

Recitation: _____

In this assignment, you will work with amortized analysis and hashing.

Question	Points	Score
1	6	
2	10	
3	9	
4	3	
5	7	
Total:	35	

You *must* print this PDF and write your answers *neatly* by hand.
You should hand in the assignment before recitation begins.

Part I: Pushing, Popping, and Paying Tokens

1. **Amortized Analysis**

Consider a counter represented as k bits: $b_{k-1}b_{k-2}\ldots b_1b_0$

The cost of flipping the i^{th} bit is 2^i tokens. For example, b_0 costs 1 token to flip, b_1 costs 2, and b_2 costs 4. We wish to analyze the cost of performing $n = 2^k$ increments of this k-bit counter. (Note that k is *not* a constant.)

(2) (a) The worst case for incrementing the counter is when every bit is set to 1. The increment then causes every bit to flip, the cost of which is

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^{k-1}$$

Show that this cost is $O(n)$.

Solution:

(2) (b) Over the course of n increments, how many tokens does it cost to flip the i^{th} bit? (*Hint:* how many times does the i^{th} bit flip? What is the cost of flipping the i^{th} bit? You may want to work through some examples, such as $i = 0$ and $i = 1$, and look for a pattern.)

Solution:

(2) (c) Based on your answer to the previous part, what is the total cost of performing n increments?

Solution:

Based on your answer above, what is the amortized cost of a single increment?

Solution:

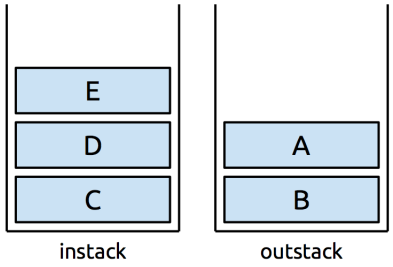
2. **A New Implementation of Queues**

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

```
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);
;
```

We wish to implement a queue using two stacks, called `instack` and `outstack`. When we enqueue an element, we push it on top of the `instack`. When we 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 need to move all of the elements from the `instack` to the `outstack`.

For example, below is one possible configuration of a two-stack queue containing the elements `A` through `E`:



We will use the following C0 code:

```
struct queue {
  stack instack;
  stack outstack;
};
typedef struct queue* queue;

bool is_queue(queue Q)
{
  return Q != NULL;
}

queue queue_new()
//@ensures is_queue(\result);
{
  queue Q = alloc(struct queue);
  Q->instack = stack_new();
  Q->outstack = stack_new();
  return Q;
}
```

(5) (a) Write the function `queue_empty` that returns true if the queue is empty.

Solution:

```
bool queue_empty(queue Q)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{









}
```

Write the function `enqueue` based on the description of the data structure above.

Solution:

```
void enqueue(elem e, queue Q)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{









}
```

Write the function `dequeue` based on the description of the data structure above.

Solution:

```
elem dequeue(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{









}
```

(2) (b) We now determine the runtime complexity of the `enqueue` and `dequeue` 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.

Solution:

enqueue: $O($

dequeue: $O($

(3) (c) 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.**

Solution: (Your answer for the number of tokens should be a constant integer since 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.**

Solution: (Your answer for the number of tokens should be a constant integer since the amortized cost should be $O(1)$.)

Part II: A Mash Of Hash

3. **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 which hashed to that location.
- **Open addressing**: each location in the table stores keys directly. We then *probe* the table to search for keys which may have collided. 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 the index $(h(k) + i) \mod m$.
  - *Quadratic probing*: on the i^{th} attempt (counting from 0), we look at the index $(h(k) + i^2) \mod m$.

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

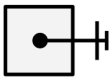
(2) (a) You are given a hash table of size m with n inserted keys that resolves collisions using separate chaining. If $n = 2m$ and the keys are *not* evenly distributed, what is the worst-case Big-O runtime complexity of searching for a specific key?

Solution: $O($ $)$

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

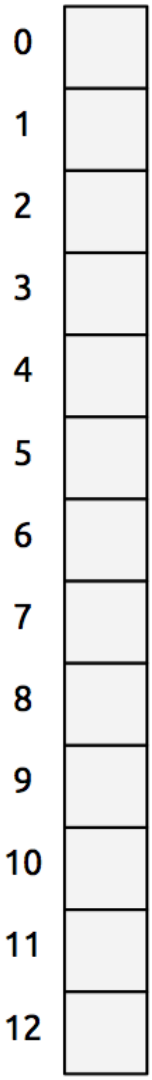
Solution: $O($ $)$

(2) (b) You are given a hash table of capacity $m = 13$ which resolves collisions using separate chaining. The hash function is $h(k) = k$.

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. If a chain is empty, you may indicate a NULL pointer using a diagram such as  where appropriate.

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

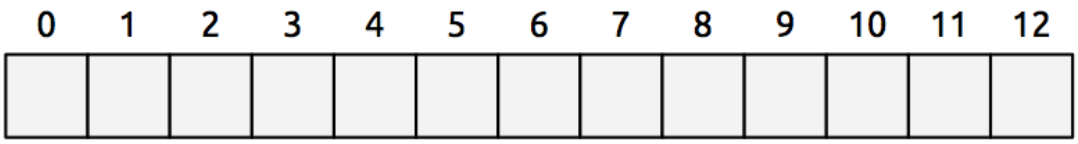
Solution:



(2) (c) Show where the sequence of keys shown below are stored in the hash table if they are inserted one by one, in the order shown, with $h(k) = k$ and $m = 13$, using linear probing to resolve collisions.

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

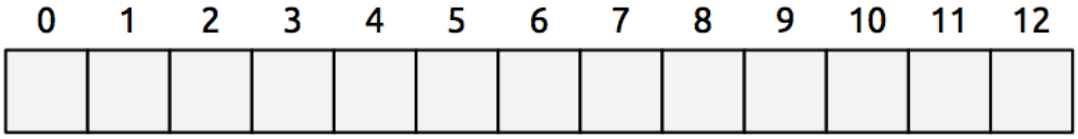
Solution:



(2) (d) Show where the sequence of keys shown below are stored in the hash table if they are inserted one by one, in the order shown, with $h(k) = k$ and $m = 13$, using quadratic probing to resolve collisions.

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

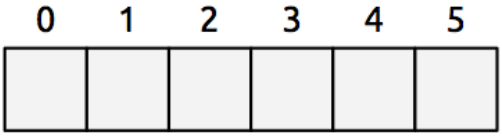
Solution:



(1) (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 $m = 6$ as your table capacity, give an example of a **non-full** hashtable and a key that cannot be successfully inserted using quadratic probing. (*Hint:* start by inserting the keys 36, 78, 12, 90.)

Solution:



Key to insert: _____

4. **Strings as Keys**

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

$$h(s) = 31^{p-1} \cdot s_0 + 31^{p-2} \cdot s_1 + \ldots + 31^1 \cdot s_{p-2} + 31^0 \cdot s_{p-1}$$

where s_i is the ASCII code for the i^{th} character of string s, and p is the length of the string. (Also, don't forget that the result of the hash function is modded by the size of the hash table.)

(1) (a) If 15105 strings were stored in a hash table of size 3021 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 chains?

Solution:

(2) (b) Using the hash function above with a table size of 3021, give an example of two different valid C0 strings that would "collide" in the hash table and would be stored in the same chain. Show your work. (Use short strings please!)

Solution:

5. **Hash Tables: Data Structure Invariants**

Refer to the C0 code below for `is_ht` that checks that a given hash table `ht` is a valid hash table.

```
struct chain_node {
  elem data;
  struct chain_node* next;
};
typedef struct chain_node chain;

struct ht_header {
  chain*[] table;
  int m;     // m = capacity = maximum number of chains table can hold
  int n;     // n = size = number of elements stored in hash table
};
typedef struct ht_header* ht;

bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->m > 0)) return false;
  if (!(H->n >= 0)) return false;
  //@assert H->m == \length(H->table);
  return true;
}
```

An obvious data structure invariant of our hash table is that every element of a chain hashes to the index of that chain. This specification function is incomplete, then: we never test that the contents of the hash table hold to this data structure invariant. That is, we test only on the struct `ht`, and not the properties of the array within.

You may assume the existence of the following client functions as discussed in class:

```
int hash(key k);

bool key_equal(key k1, key k2);

key elem_key(elem e)
//@requires e != NULL;
  ;
```

(5) (a) Extend `is_ht` from above, adding code to check that every element in the hash table matches the chain it is located in, and that each chain is non-cyclic.

Solution:

```
bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->m > 0)) return false;
  if (!(H->n >= 0)) return false;
  //@assert H->m == \length(H->table);

  int nodecount = 0;

  for (int i = 0; i < ____________________________; i++)
  {
    // set p equal to a pointer to first node
    // of chain i in table, if any

    chain* p = ____________________________________________________;

    while (_____________________________________________________)
    {
      elem e = p->data;

      if ((e == NULL) || (_________________________________ != i))

        return false;

      nodecount++;

      if (nodecount > ____________________________________)

        return false;

      p = _____________________________________________________;

    }
  }

  if (_____________________________________________________________)

    return false;

  return true;
}
```

(2) (b) Consider the `ht_lookup` function given below:

```
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
{
  int i = abs(hash(k) % H->m);
  chain* p = H->table[i];
  while (p != NULL)
  //@loop_invariant is_chain(p, i, H->m);
  {
    //@assert p->data != NULL;
    if (key_equal(elem_key(p->data), k))
      return p->data;
    else
      p = p->next;
  }
  /* not in chain */
  return NULL;
}
```

Give a simple postcondition for this function.

Solution:

```
/*@ensures \result == ________________________________________

           || key_equal(k, ____________________________________);
@*/
```