15-122: Principles of Imperative Computation, Spring 2023

Written Homework 8

Due on Gradescope: Sunday 12th March, 2023 by 9pm

Name: _____

Andrew ID: _____

Section: _____

This written homework covers amortized analysis, hash tables, and generics.

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

- ***PDFescape*** or ***DocHub***, two 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.*

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	Total
Points:	2.5	7.5	5	15
Score:				

1. **Amortized Analysis Revisited**

Consider a special binary counter represented as n bits: $b_{n-1}b_{n-2}\ldots b_1b_0$. For this special counter, 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 tokens to flip, b_2 costs 4 tokens to flip, etc. We wish to analyze the cost of performing $k = 2^n$ increments of this n-bit counter. (Note that n is *not* a constant.)

Observe that if we begin with our n-bit counter containing all 0s, and we increment k times, where $k = 2^n$, the final value stored in the counter will again be 0.

1pt **1.1** The worst case for a single increment of the counter is when every bit is set to 1. The increment then causes every bit to flip, the cost of which is

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

Find a closed form in terms of n for the formula above. Using this fact, explain in one or two sentences why this cost is $O(k)$ — again recall that $k = 2^n$.

Closed form: _____

The cost is $O(k)$ because _____

_____

_____

1.5pts **1.2** Now, we will use amortized analysis to show that although the worst case for a single increment is $O(k)$, the amortized cost of a single increment is asymptotically less than this. Remember, $k = 2^n$. Run some experiments for small values of n and see if you can find a pattern.

Over the course of k increments, how many tokens in total does it cost to flip the i^{th} bit the necessary number of times?

Based on your answer to the previous part, what is the total cost in tokens of performing k increments? (In other words, what is the total cost of flipping each of the n bits through k increments?) Write your answer as a function of k **only**. (Hint: what is n as a function of k?)

Based on your answer above, what is the amortized cost of a single increment as a function of k **only**?

$O($_____$)$ amortized

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

The C0 code below for `is_hdict` checks that a given hash dictionary is valid.

```
// typedef _________* entry;
typedef struct chain_node chain;
struct chain_node {
  entry data;             // != NULL
  chain* next;
};

struct hdict_header {
  int size          // number of entries stored in hash table
  int capacity;     // maximum number of chains table can hold
  chain*[] table;
};
typedef struct hdict_header hdict;

bool is_array_expected_length(chain*[] table, int length) {
  //@assert \length(table) == length;
  return true;  }

bool is_hdict(hdict* H) {
  return H != NULL && H->capacity > 0 && H->size >= 0
      && is_array_expected_length(H->table, H->capacity);
}
```

An obvious data structure invariant of our hash table is that every entry in a chain hashes to the index of that chain. Then, the above specification function is incomplete: we never test that the contents of the hash table satisfies this data structure invariant. That is, we test only on the struct `hdict_header`, and not the properties of the array within.

On the next page, extend `is_hdict` from above, adding a helper function to check that every entry in the hash table belongs in the chain it is located in, and that each chain is acyclic. You should assume we will use the following three functions for extracting the key (type `key`) of an `entry`, hashing a key and for comparing two keys for equivalence:

```
key entry_key(entry e)              /*@requires e != NULL;@*/ ;
int key_hash(key k);
bool key_equiv(key k1, key k2);
```

Additionally, the constant-time function

```
int index_of_key(hdict* H, key k)
/*@requires H-> capacity > 0; @*/
/*@ensures 0 <= \result && \result < H->capacity; @*/ ;
```

maps a key to a valid index. It is provided for your convenience.

2pts

2.1 Note: your answer needs only to work for hash tables containing a few hundred million entries — do not worry about the number of entries exceeding `int_max()`.

```
bool has_valid_chains(hdict* H)
// Preconditions (H != NULL, H->size >= 0...) omitted for space
{
    int nodecount = 0;

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

        chain* p = ____________________________________;

        while (____________________________________) {
            entry e = p->data;

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

                return false;

            nodecount++;

            if (nodecount > ____________________)

                return false;

            p = ____________________________________;
      }
    }

    if (____________________________________)

        return false;

    return true;
}

bool is_hdict(hdict* H) {
    return H != NULL && H->capacity > 0 && H->size >= 0
        && is_array_expected_length(H->table, H->capacity)
        && has_valid_chains(H);
}
```

0.5pts

2.2 Consider the function `hdict_lookup` given below:

```
entry hdict_lookup(hdict* H, key k)
//@requires is_hdict(H);
{
  int i = index_of_key(H, k);
  chain* p = H->table[i];
  while (p != NULL) {
    //@assert p->data != NULL;
    if (key_equiv(entry_key(p->data), k))
      return p->data;
    p = p->next;
  }
  return NULL;    // not in chain
}
```

Give a simple postcondition for this function.

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

        || key_equiv(k, _____________________________________);@*/
```

1pt

2.3 The function `hdict_remove(H,k)` below removes the entry with key `k` from hash table `H`. Complete the helper function `remove_from_chain(p,k)` that returns the chain obtained by removing the entry with key `k` from chain `p` while supporting the post-conditions of `hdict_remove`. Recall that chains in our hash table implementation do not have duplicates. *[Hint: a recursive solution may be worth considering.]*

```
chain* remove_from_chain(chain* p, key k) {
  if (p == NULL) return NULL;

  if (_____________________________________) return ____________;

  _____________________________________________________________;

  return _____________________________________________________;
}

void hdict_remove(hdict* H, key k)
//@requires is_hdict(H);
//@ensures  is_hdict(H) && hdict_lookup(H, k) == NULL;
{
  if (hdict_lookup(H, k) != NULL) (H->size)--;
  int i = index_of_key(H, k);
  H->table[i] = remove_from_chain(H->table[i], k);
}
```

2.5pts

2.4 Emma is implementing this hash dictionary with an initial table capacity of 8. She wants it to be very fast even if it contains lots of entries. She would like to know what it would cost to insert n entries into an empty dictionary and then look one up, assuming an **optimal** hash function (i.e., that distributes key uniformly).

We assume that each array or pointer access costs one unit of time. As always, give the simplest, tightest bounds.

a. At first, Emma implements the table that *never resizes*. For each insertion, she needs to find the correct chain, check if an entry with that key is already in there, and add it if it isn't. What is the cost?

n insertions:	$O($_____$)$
then, 1 lookup:	$O($_____$)$

b. This is too slow for her needs! Emma decides to resize the table every time its load factor exceeds 1.5. To do so, she allocates a new table and rehashes all the entries into it. She figures that, since she needs space for one entry, she'll *resize the table by one*. What is the cost now?

n insertions:	$O($_____$)$
then, 1 lookup:	$O($_____$)$

c. This is still slow! Stumped, Emma looks through her notes, and decides to try *doubling the table* when the load factor exceeds 1.5.

n insertions:	$O($_____$)$
then, 1 lookup:	$O($_____$)$

d. Notice that the last technique is what we used in unbounded arrays. What is the amortized cost of a single insertion?

1 insertion (amortized):	$O($_____$)$

e. This is much faster, but she's unsure why. She expected the same performance as in attempt (b). In one clear sentence, explain to Emma why doubling the size of the table yields such a boost in performance.

1.5pts

2.5 In our `hdict` implementation, we use a library helper function `index_of_key` that takes an element, computes its hash value using the client's `key_hash` function and converts this hash value to an integer. Here is this function, with the return expression missing:

```
1 int index_of_key(hdict* H, key k)
2 //@requires H->capacity > 0;
3 //@ensures 0 <= \result && \result < H->capacity;
4 {
5   int h = key_hash(k);
6   return ____________________________;  // What should go here?
7 }
```

Answer the following questions about what to have on line 6:

a. Assume line 6 is

```
6   return abs(h) % H->capacity;
```

In this case, the function fails on exactly one value for `h`.

It fails when `h` = _____

b. Changing line 6 to

```
6   return h < 0 ? 0 : h % H->capacity;
```

solves this issue, but this is not a great solution. In one sentence, explain what feature of the resulting hash table makes this solution undesirable.

_____

_____

_____

(The *ternary operator* `b ? e1 : e2` evaluates to the value of expression `e1` if the boolean test `b` is `true`, and to the value of `e2` if `b` is `false`.)

c. Complete line 6 so it avoids the problems of the previous two attempts.

```
6   return (h < 0 ? ____________________ : h) % H->capacity;
```

3. **Generic Algorithms**

A generic comparison function might be given a type as follows in C1:

```
typedef int compare_fn(void* x, void* y);
```

(Note: there's no precondition that `x` and `y` are necessarily non-`NULL`.)

If we're given such a function, we can treat `x` as being less than `y` if the function returns a negative number, treat `x` as being greater than `y` if the function returns a positive number, and treat the two arguments as being equal if the function returns `0`.

Given such a comparison function, we can write a function to check that an array is sorted even though we don't know the type of its elements (as long as it is a pointer type):

```
bool is_sorted(void*[] A, int lo, int hi, compare_fn* cmp)
  //@requires 0 <= lo && lo <= hi && hi <= \length(A) && cmp != NULL;
```

1pt **3.1** Complete the generic binary search function below. You don't have access to generic variants of `lt_seg` and `gt_seg`. Remember that, for sorted integer arrays, `gt_seg(x, A, 0, lo)` was equivalent to `lo == 0 || A[lo - 1] < x`.

```
int binsearch_generic(void* x, void*[] A, int n, compare_fn* cmp)
 //@requires 0 <= n && n <= \length(A) && cmp != NULL;
 //@requires is_sorted(A, 0, n, cmp);
 {
   int lo = 0;
   int hi = n;

   while (lo < hi)
   //@loop_invariant 0 <= lo && lo <= hi && hi <= n;

   //@loop_invariant lo == _____ || _____________________ < 0;

   //@loop_invariant hi == _____ || _____________________ > 0;
   {
     int mid = lo + (hi - lo)/2;

     int c = ____________________________________________________;

     if (c == 0) return mid;
     if (c < 0) lo = mid + 1;
     else hi = mid;
   }
   return -1;
 }
```

Suppose you have a generic sorting function, with the following contract:

```
void sort_generic(void*[] A, int lo, int hi, compare_fn* cmp)
  //@requires 0 <= lo && lo <= hi && hi <= \length(A) && cmp != NULL;
  //@ensures is_sorted(A, lo, hi, cmp);
```

1.5pts

3.2 Write an integer comparison function `compare_ints` that can be used with this generic sorting function. The contracts on your `compare_ints` function *must* be sufficient to ensure that no precondition-passing call to `compare_ints` can possibly cause a memory error.

```
int compare_ints(void* x, void* y)

//@requires x != NULL && \hastag(_____________________________);

//@requires y != NULL && \hastag(_____________________________);
{

   if (_____________________________________) return __________;

   if (_____________________________________) return __________;

   return ____________________________________________________;
}
```

2.5pts

3.3 Using `sort_generic` (which you may assume has already been written) and `compare_ints`, fill in the body of the `sort_ints` function below so that it will sort the array `A` of integers. You can omit loop invariants. But of course, when you call `sort_generic`, the preconditions of `compare_ints` must be satisfied by any two elements of the array `B`.

```
void sort_ints(int[] A, int n)
//@requires \length(A) == n;
{
   // Allocate a temporary generic array of the same size as A

   void*[] B = ____________________________________;

   // Store a copy of each element in A into B









   // Sort B using sort_generic and compare_ints from task 2



   // Copy the sorted ints in your generic array B into array A









}
```