15-122: Principles of Imperative Computation, Spring 2020

Written Homework 8

Due on Gradescope: Thursday 19$^{\text{th}}$ March, 2020 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.
- ***Preview***, the Mac's PDF viewer.
- ***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*.

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:	2.5	4	1.5	4	12
Score:					

1. **Amortized Analysis Revisited**

Consider a special binary counter represented as k bits: $b_{k-1}b_{k-2}\ldots b_1b_0$. 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 $n = 2^k$ increments of this k-bit counter. (Note that k is *not* a constant.)

Observe that if we begin with our k-bit counter containing all 0s, and we increment n times, where $n = 2^k$, 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

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

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

Closed form: ______________________________

The cost is $O(n)$ because ______________________________

____

____

1.5pts

1.2 Now, we will use amortized analysis to show that although the worst case for a single increment is $O(n)$, the amortized cost of a single increment is asymptotically less than this. Remember, $n = 2^k$.

Over the course of n 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 n increments? (In other words, what is the total cost of flipping each of the k bits through n increments?) Write your answer as a function of n **only**. (Hint: what is k as a function of n?)

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

$O($______________________________$)$ amortized

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

A *hash set* is a hash table where keys and entries coincide: it is a convenient data structure to implement sets whose elements are these keys/entries. The type `hset` defines hash sets similarly to separate-chaining hash tables. The code below checks that a given hash set is valid.

```
// typedef _________ elem;       // type of elements -- client defined
typedef struct chain_node chain;
struct chain_node {
  elem data;
  chain* next;
};

struct hset_header {
  int size;            // number of elements stored in hash set
  int capacity;        // maximum number of chains in hash set
  chain*[] table;
};
typedef struct hset_header hset;

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

bool is_hset(hset* 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 set is that every element of 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 satisfy this additional invariant. That is, we test only on the struct `hset`, and not on the properties of the array within.

On the next page, extend `is_hset` from above, adding a helper function to check that every element 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 two functions for hashing elements and for comparing them for equality:

```
int elem_hash(elem x);
bool elem_equiv(elem x, elem y);
```

Additionally, the function

```
int index_of_elem(hset* H, elem x)
/*@requires H->capacity > 0; @*/
/*@ensures 0 <= \result && \result < H->capacity; @*/ ;
```

maps an element 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 elements — do not worry about the number of elements exceeding `int_max()`.

```
bool has_valid_chains(hset* 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 (______________________________________________) {
        elem x = p->data;

        if (____________________________________________ != i)
          return false;

        nodecount++;

        if (nodecount > _____________________________________)
          return false;

        p = _________________________________________________;
    }
  }

  if (_______________________________________________________)

      return false;

  return true;
}

bool is_hset(hset* 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 We generally don't care about the cost of specification functions, but what is the worst case complexity of `has_valid_chains` as a function of the number n of elements in the hash set?

$O($______________________________$)$

1.5pts

2.3 The updated function `is_hset` still falls short of flagging all possible invalid hash sets: nothing prevents a chain from containing multiple occurrences of an element. Given the above declarations, describe how you could check whether the hash set contains duplicate elements without allocating any extra memory. What is the cost?

____

____

Cost: $O($______________________________$)$

Assume you have a comparison function, **`int`** `elem_compare(elem x, elem y)` which returns -1 if `x` is to be considered less than `y`, 0 if they are equal, and 1 if `x` is greater than `y`. How could you modify the behavior of the hash set to make the cost of finding duplicates asymptotically faster?

Change: ____

____

____

Cost: $O($______________________________$)$

3. **Hash Tables: Mapping Hash Values to Hash Table Indices**

   In our `hset` implementation, we use a library helper function `index_of_elem` that takes an element, computes its hash value using the client's `elem_hash` function and converts this hash value to an integer. The first two functions below try to implement `index_of_elem` but have issues.

0.5pts

3.1 The following function has a bug. For one specific hash value `h`, this function does not return an index that is valid for a hash table. Identify the specific hash value.

```
int index_of_elem(hset* H, elem x)
//@requires H->capacity > 0;
//@ensures 0 <= \result && \result < H->capacity;
{
  int h = elem_hash(x);
  return abs(h) % H->capacity;
}
```

This function fails when `h` = ______________________________

0.5pts

3.2 The following function has an undesirable feature, although it always returns a valid index. Identify the flaw and, in one sentence, explain why it's a problem. (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`.)

```
int index_of_elem(hset* H, elem x)
//@requires H->capacity > 0;
//@ensures 0 <= \result && \result < H->capacity;
{
  int h = elem_hash(x);
  return h < 0 ? 0 : h % H->capacity;
}
```

0.5pts

3.3 Complete the following function so it avoids the problems in the previous two implementations of `index_of_elem`.

```
int index_of_elem(hset* H, elem x)
//@requires H->capacity > 0;
//@ensures 0 <= \result && \result < H->capacity;
{
  int h = elem_hash(x);

  return (h < 0 ? ____________________________ : h) % H->capacity;
}
```

4. **Generic Algorithms**

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

```
typedef int compare_fn(void* x, void* y)
  //@ensures -1 <= \result && \result <= 1;
```

(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 -1, treat x as being greater than y if the function returns 1, 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

4.1 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 == _____ || ____________________ == -1;

   //@loop_invariant hi == _____ || ____________________ == 1;
   {
     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);
```

Recall also the **abstract** pixel interface seen early in the course:

```
//typedef ______ pixel_t;  // pixel_t is not necessarily int

int get_red(pixel_t p)   /*@ensures 0 <= \result && \result < 256; @*/ ;
int get_green(pixel_t p) /*@ensures 0 <= \result && \result < 256; @*/ ;
int get_blue(pixel_t p)  /*@ensures 0 <= \result && \result < 256; @*/ ;
int get_alpha(pixel_t p) /*@ensures 0 <= \result && \result < 256; @*/ ;

pixel_t make_pixel(int alpha, int red, int green, int blue)
/*@requires 0 <= alpha && alpha < 256; @*/
/*@requires 0 <= red   &&   red < 256; @*/
/*@requires 0 <= green && green < 256; @*/
/*@requires 0 <= blue  &&  blue < 256; @*/ ;
```

1pt

4.2 Write a pixel comparison function `compare_red` that can be used with the above generic sorting function, which you should assume is already written. The function `compare_red` compares pixels based uniquely on the intensity of their red component. For example, pixel `p1` with red component 122 is considered smaller than pixel `p2` with red component 210, irrespective of the values of their other components.

As you write this function, the contracts on your `compare_red` function *must* be sufficient to ensure that no precondition-passing call to `compare_red` can possibly cause a memory error.

```
int compare_red(void* x, void* y)
//@requires x != NULL && \hastag(____________________________);

//@requires y != NULL && \hastag(____________________________);
//@ensures -1 <= \result && \result <= 1;
{



  if (______________________________________) return ___________;

  if (______________________________________) return ___________;

  return ________________________________________________________;
}
```

2pts

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

```
void sort_red(pixel_t[] 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_red from task 2



  // Copy the sorted pixels from generic array B into array A








}
```