15-122: Principles of Imperative Computation, Spring 2024

Written Homework 8

Due on Gradescope: Monday 11th March, 2024 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:

- **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	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 of a single increment 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 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 constant-time 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()`.

```
// in_chain will be modified in a later task
bool in_chain(chain* p, elem x) { return true; }

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 (in_chain(p->next, x)) return false;

        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?

$O($_____$)$

1pt **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, update the function `in_chain` so that it actually detects duplicate elements.

```
bool in_chain (chain* p, elem x) {




}
```

Given a hash set containing n elements, what is the cost of `is_hset` with this additional update?

Cost: $O($_____$)$

2.5pts

2.4 Sheng is implementing this hash set with an initial table capacity of 10. He wants it to be very fast even if it contains lots of elements. He would like to know what it would cost to insert n elements into an empty hash set and then look one up, assuming an **optimal** hash function (i.e., one that distributes key uniformly).

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

1. At first, Sheng implements a table that *never resizes*. For each insertion, he needs to find the correct chain, check if the element is already in there, and add it if it isn't. What is the cost? What is the amortized cost of an operation in this sequence?

   n insertions: $O($_____$)$

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

   Amortized cost: $O($_____$)$

2. This is too slow for his needs! Sheng decides to resize the table every time its load factor exceeds 1.6. To do so, he allocates a new table and rehashes all the elements into it. He figures that, since he needs space for one element, he'll *resize the table by one*. What are the costs now?

   n insertions: $O($_____$)$

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

   Amortized cost: $O($_____$)$

3. Better but still very slow! Maybe resizing by 1 is too little. Next, once the load factor exceeds 1.6, Sheng resizes the table to accommodate 15122 additional elements. What are the costs?

   n insertions: $O($_____$)$

   15122 lookups: $O($_____$)$

   Amortized cost: $O($_____$)$

4. This is still slow! Stumped, Sheng looks through his notes, and decides to try *doubling the table* when the load factor exceeds 1.6.

   n insertions: $O($_____$)$

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

   Amortized cost: $O($_____$)$

5. This is much faster, but he's unsure why: both resizing by 15122 and doubling the size of the table result in less frequent resizes. In one clear sentence, explain to Sheng what makes doubling the table so much more efficient.

1.5pts

2.5 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. Here is this function, with the return expression missing:

```
1 int index_of_elem(hset* H, elem x)
2 //@requires H->capacity > 0;
3 //@ensures 0 <= \result && \result < H->capacity;
4 {
5   int h = elem_hash(x);
6   return ____________________________;  // What should go here?
7 }
```

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

1. 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` = _____

2. 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`.)

3. Complete line 6 so it avoids the problems of the previous two attempts, thereby making `h` a **good** hash function.

   ```
   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 a string comparison function `compare_strings` that can be used with this generic sorting function. Strings are compared lexicographically, something the `<string>` library function `string_compare` does already. The contracts on your `compare_strings` function *must* be sufficient to ensure that no precondition-passing call to `compare_strings` can possibly cause a memory error. Depending on how you write your solution, you may or may not need the extra space at the beginning of this function.

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

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

//@requires y != NULL && \hastag(_____________________________);
{
                                        // Extra space if needed









  return ____________________________________________________;
}
```

2.5pts

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

```
void sort_strings(string[] 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_strings from task 2



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









}
```