15-122 : Principles of Imperative Computation, Spring 2016

Written Homework 8

Due: Monday 14$^{\text{th}}$ March, 2016

Name: ______________________________

Andrew ID: ______________________________

Section: ______________________________

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

The assignment is due by 1:30pm on Monday 14$^{\text{th}}$ March, 2016.

This assignment is to be completed by editing the PDF file and then submitted through Gradescope.

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.)

Note 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 (a) 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

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

Explain in one or two sentences why this cost is $O(n)$. (HINT: Find a closed form for the formula above.)

2pts (b) 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?

Solution: ____________

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?)

Solution: ____________

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

Solution: O(____________) amortized

3pts 2. **Hash Tables: Data Structure Invariants**

Refer to the C0 code below for `is_hset`, which checks that a given separate-chaining hash set containing only strings is valid.

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

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

int hashindex(hset* H, string x)
//@requires H != NULL && H->capacity > 0;
//@ensures 0 <= \result && \result < H->capacity;
{
  return abs(string_hash(x) % H->capacity);
}

bool is_table_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_table_expected_length(H->table, H->capacity);
}
```

An obvious data structure invariant of our hash table is that every element of a chain hashes to the index of that chain. Thus, this 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 non-cyclic. You should assume we will use the following two functions for hashing strings and for comparing them for equivalence:

```
int string_hash(string x);
bool string_equiv(string x, string y);
```

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 (____________________) {

         string 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_table_expected_length(H->table, H->capacity)
      && has_valid_chains(H);
}
```

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

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

1pt (a) The following function has a bug in it. 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 hashindex(hset H, elem x)
//@requires H != NULL && H->capacity > 0;
//@requires x != NULL;
//@ensures 0 <= \result && \result < H->capacity;
{
  int h = elem_hash(x);
  return abs(h) % H->capacity;
}
```

Solution: This function fails when `h` = ________________

1pt (b) 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.

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

1pt (c) Complete the following function so it avoids the problems in the previous two implementations of `hashindex`.

```
int hashindex(hset H, elem x)
//@requires H != NULL && H->capacity > 0;
//@requires x != NULL;
//@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* comp)
  //@requires 0 <= lo && lo <= hi && hi <= \length(A) && comp != NULL;
```

2pts (a) 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* comp)
//@requires 0 <= n && n <= \length(A) && comp != NULL;
//@requires is_sorted(A, 0, n, comp);
{
  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;
    else 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* comp)
  //@requires 0 <= lo && lo <= hi && hi <= \length(A) && comp != NULL;
  //@ensures is_sorted(A, lo, hi, comp);
```

2pts

(b) Write an integer comparison function `compare_ints` that can be used with this generic sorting function, which you should assume is already written. You can leave out the postcondition that the result of `compare_ints` is between `-1` and `1` inclusive. However, 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 -1;

  if (____________________) return 1;
  return 0;
}
```

2pts (c) Using the above generic sorting function and `compare_ints`, fill in the body of the `sort_ints` function below so that it will sort the array `A` of integers using the generic sort function specified above. 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 part b



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








}
```