15-122 : Principles of Imperative Computation, Spring 2013

Homework 5 Theory [Update 1]

Due: Tuesday, March 26, 2013, at the **beginning** of lecture

Name: ______________________________

Andrew ID: ______________________________

Recitation: ______________________________

The written portion of this weekï£¡s homework will give you some practice working with hash tables and trees. You can either type up your solutions or write them *neatly* by hand in the spaces provided. You should submit your work in class on the due date just before lecture or recitation begins. Please remember to *staple* your written homework before submission.

Question	Points	Score
1	4	
2	6	
3	10	
Total:	20	

You *must* use this printout, include this cover sheet,
and staple the whole thing together before turning it in.
Either type up the assignment using 15122-theory5.tex,
or print this PDF and write your answers *neatly* by hand.

1. **Hash Tables using Separate Chaining.**

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

```
struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node list;

struct ht_header {
  list*[] 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, int m)
//@requires m > 0;
//@ensures 0 <= \result && \result < m;
  ;

bool key_equal(key k1, key k2);

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

(4) (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 numnodes = 0;

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

    list* p = __________________________________________________;

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

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

        return false;

      numnodes++;

      if (numnodes > ___________________________________________)

        return false;

      p = ______________________________________________________;

    }
  }

  if (__________________________________________________________)

    return false;

  return true;
}
```

2. **Linear and quadratic probing**

Consider two alternate implementations of hash tables that use an array only (no chains), using linear/quadratic probing to resolve collisions. In linear probing, if a key k is inserted/lookup'd, on the $(i+1)$st attempt we look at index $(h(k)+i) \mod m$, where h is the hash function being used and m is the size of the table. Similarly, in quadratic probing, we look at index $(h(k)+i^2) \mod m$ on the $(i+1)$st attempt.

Note that for this question, the hash function $h(k)$ does not perform a modulus by the table size. Also, for this question, you may assume that there is no integer overflow (i.e. even for large i, i^2 will still be non-negative).

(4) (a) Assume that we hash a set of integer keys into a hash table of capacity $m = 11$ using a hash function $h(k) = k$.

In the first set of boxes, show where the following sequence of keys are stored in the hash table if they are inserted in the order shown using linear probing to resolve collisions.

In the second set, show the same but with quadratic probing.

```
54, 67, 23, 88, 39, 98, 72
```

Solution:

```
With Linear Probing:

      0    1    2    3    4    5    6    7    8    9   10
     ------------------------------------------------------
    |    |    |    |    |    |    |    |    |    |    |    |
     ------------------------------------------------------


With Quadratic Probing:

      0    1    2    3    4    5    6    7    8    9   10
     ------------------------------------------------------
    |    |    |    |    |    |    |    |    |    |    |    |
     ------------------------------------------------------
```

(2) (b) Quadratic probing suffers from one problem that linear probing does not. In particular, given a non-full hashtable, insertions will linear probing will always succeed, while insertions with quadratic probing may or may not succeed (i.e. they may never find an open spot to insert).

Give an example of a non-full hashtable and a key that cannot be successfully inserted using quadratic probing. Use $h(k) = k$ as your hash function and $m = 6$ as your table capacity.

Solution:

```
Non-Full Hashtable:                     Key to Insert:

       0    1    2    3    4    5
      -----------------------------           ----
     |    |    |    |    |    |    |         |    |
      -----------------------------           ----
```

3. **BSTs.**

In this question, we will think about the implementation of binary search trees. The `is_bst` and `is_ordered` functions are implemented as follows:

```
bool is_ordered(tree* T, elem lower, elem upper) {
  if (T == NULL) return true;
  if (T->data == NULL) return false;
  key k = elem_key(T->data);
  if (!(lower == NULL || key_compare(elem_key(lower),k) < 0))
    return false;
  if (!(upper == NULL || key_compare(k,elem_key(upper)) < 0))
    return false;
  return is_ordered(T->left, lower, T->data)
    && is_ordered(T->right, T->data, upper);
}

bool is_bst(bst B) {
  if (B == NULL) return false;
  return is_ordered(B->root, NULL, NULL);
}
```

Remember that we cannot compare elements or keys directly with the C0 comparison operations; we have to use the client function `elem_key` to extract a key from an element and use the client function `key_compare` to compare keys.

(2) (a) It is possible to implement `bst_search` as an iterative function rather than a recursive one. Fill in the blanks so that the below function correctly implements `bst_search`.

The lines involving the variables `lower` and `upper` are used only to prove that the loop invariant is preserved. You should not use `lower` or `upper` when filling in the blanks.

Solution:

```
/*  1 */ elem bst_search(bst B, key k)
/*  2 */ //@requires is_bst(B);
/*  3 */ /*@ensures \result == NULL
/*  4 */         || key_compare(k, elem_key(\result)) == 0; @*/
/*  5 */ {
/*  6 */   tree* T = B->root;
/*  7 */   elem lower = NULL;
/*  8 */   elem upper = NULL;
/*  9 */
/* 10 */   while (_______________________________________________)
/* 11 */   //@loop_invariant is_ordered(T, lower, upper);
/* 12 */   {
/* 13 */     if (____________________________________________) {
/* 14 */       upper = T->data;
/* 15 */       T = T->left;
/* 16 */     }
/* 17 */     else {
/* 18 */       lower = T->data;
/* 19 */       T = T->right;
/* 20 */     }
/* 21 */   }
/* 22 */
/* 23 */   if (T == NULL) return NULL;
/* 24 */   return T->data;
/* 25 */ }
```

(4) (b) Prove that the code you wrote on lines 10 and 13 is safe – that pointer dereferences are safe and that the preconditions of any function you call are satisfied.

Solution:

Prove that the loop invariant is true initially.

Solution:

Prove that the loop invariant is preserved by any iteration of the loop. There are two cases that to show; just show the case where the conditional on line 13 evaluates to `true`.

Solution:

(4) (c) The ordering invariant of BSTs makes it possible to print the elements of the BST in descending order, without having to use a temporary data structure to store and/or sort any of the elements. Write a function `bst_print_desc(bst B)` that prints the elements of a BST in *descending order* using a helper function `tree_print_desc(tree* T)`.

You can assume that you have a client-provided function `elem_print(elem e)` that prints an element. Your solution should be simple and straightforward, so try hard to think of a way to do this elegantly; overly complex code will cost points.

Solution:

```
void tree_print_desc(tree* T)
//@requires is_ordered(T, NULL, NULL);
{

}

void bst_print_desc(bst B)
//@requires is_bst(B);
{

      __________________________________________________________;
      return;
}
```