

**15-122 : Principles of Imperative Computation, Spring 2014**  
**Written Homework 8**

Due before class: Thursday, March 20, 2014

Name: \_\_\_\_\_

Andrew ID: \_\_\_\_\_

Recitation: \_\_\_\_\_

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

Question	Points	Score
1	5	
2	10	
Total:	15	

You must do this assignment in one of two ways and bring the stapled printout to the handin box on Thursday:

- 1) Write your answers *neatly* on a printout of this PDF.
- 2) Use the TeX template at <http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15122-s14/www/theory8.tgz>

## 1. Hash Tables: Data Structure Invariants

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

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

struct ht_header {
    chain*[] 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);

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 nodecount = 0;

    for (int i = 0; i < H->m; i++)
    {
        // set p equal to a pointer to first node
        // of chain i in table, if any

        chain* p = H->table[i];

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

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

                return false;

            nodecount++;

            if (nodecount > H->n)

                return false;

            p = p->next;
        }
    }

    if (nodecount!=H->n)

        return false;

    return true;
}
```

- (1) (b) Consider the `ht_lookup` function given below:

```
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
{
    int i = abs(hash(k) % H->m);
    chain* p = H->table[i];
    while (p != NULL)
        //@loop_invariant is_chain(p, i, H->m);
    {
        //@assert p->data != NULL;
        if (key_equal(elem_key(p->data), k))
            return p->data;
        else
            p = p->next;
    }
    /* not in chain */
    return NULL;
}
```

Give a simple postcondition for this function.

**Solution:**

```
/*@ensures \result == NULL
                || key_equal(k, elem_key(\result));
@*/
```

**2. Binary Search Trees**

- (1) (a) Draw the binary search tree that results from inserting the following keys in the order given:

75 92 99 13 84 42 71 98 73 20

Be sure all branches in your tree are clearly drawn so we can distinguish left branches from right branches.

**Solution:**

- (1) (b) How many different binary search trees can be constructed using the following five keys: 73, 28, 52, -9, 104 if they can inserted in any arbitrary order?

**Solution:** 42

Refer to the implementation of binary search trees discussed in class that is available on our course website.

- (3) (c) Write an implementation of a new library function, `bst_height`, that returns the height of a binary search tree. The height of a binary search tree is defined as the maximum number of nodes as you follow a path from the root to a leaf. As a result, the height of an empty binary search tree is 0. Your function must include a **recursive** helper function `tree_height`.

HINT: In general, the height of a tree rooted at node T is one more than the height of its deepest subtree.

**Solution:**

```
int tree_height(tree* T)
//@requires is_ordered(T, NULL, NULL);
{

}

int bst_height(bst B)
//@requires is_bst(B);
//@ensures is_bst(B);
{
    return _____;
}
```

- (5) (d) Consider extending the BST library implementation with the following function which deletes an element from the tree with the given key.

```
void bst_delete(bst B, key k)
//@requires is_bst(B);
//@ensures is_bst(B);
{
    B->root = tree_delete(B->root, key k);
}
```

Complete the code for the recursive helper function `tree_delete` which is used by the `bst_delete` function. This function should return a pointer to the tree rooted at `T` once the key is deleted (if it is in the tree).

You will need to complete an additional helper function `largest_child` that removes and returns the largest child rooted at a given tree node `T`.

**Solution:**

```
tree* tree_delete(tree* T, key k)
{
    if (T == NULL) {                                // key is not in the tree
        return NULL;
    }

    if (key_compare(k, elem_key(T->data)) < 0) {
        T = tree_delete(T->left, k);
        return T;
    } else if (key_compare(k, elem_key(T->data)) > 0) {
        T = tree_delete(T->right, k);
        return T;
    } else {    // key is in current tree node T
        if (T->left == NULL)    // node has only right child
            return _____;

        else if (T->right == NULL)    // node has only left child
            return _____;
    }
}
```

```

        else {                // Node to be deleted has two children

            if (T->left->right == NULL) {

                // Replace the data in T with the data
                // in the left child.

                -----;

                // Replace the left child with its left child.

                -----;

                return T;
            }
            else {
                // Search for the largest child in the
                // left subtree of T and replace the data
                // in node T with this data after removing
                // the largest child in the left subtree.
                T->data = largest_child(T->left);
                return T;
            }
        }
    }

elem largest_child(tree* T)
//@requires T != NULL && T->right != NULL;
{
    if (T->right->right == NULL) {

        elem e = -----;

        T->right = -----;

        return e;
    }

    return largest_child(-----);
}

```