15-122 : Principles of Imperative Computation

Fall 2012

Assignment 5 – Selected Solutions

(Theory Part)

The following solutions are provided to you to help you study this semester. They are not to be distributed to others outside of the class nor are they intended to be used by students in future semesters as a substitute for completing one's own homework assignments.

1. **Collision Resolution.**

   Consider an alternate implementation of hash tables that uses an array only (no chains) and linear probing to resolve collisions. Recall that in linear probing, if a collision occurs, we search through the array linearly (with wraparound if necessary) for the first available position.

(0) (a) Omitted.

(0) (b) Omitted.

(3) (c) Consider an update to the hash table that adds a boolean array to the hash table data structure such that cell i is true if the corresponding table cell is or has been occupied, and false otherwise. When a new hash table is created, all cells in the `occupied` array are initially `false`.

```
struct ht_header {
    elem[] table;
    bool[] occupied;
    int m;          // m = capacity
};
typedef struct ht_header* ht;
```

Complete the functions below for `ht_insert`, `ht_lookup` and `ht_delete` for the hash table that uses linear probing to resolve collisions and the `occupied` array as described above. You may assume there is an appropriate `is_ht` function already defined, and you may assume the existence of the client functions `hash`, `elem_key`, and `key_equal` as described in problem 1.

```
void ht_insert(ht H, elem e)
//@requires is_ht(H);
//@requires e != NULL;
//@ensures is_ht(H);
```

```
{
    key k = elem_key(e);
    int i = hash(k, H->m);
    int j = 0;
    while (j < H->m) {
        if (____)   // write missing condition below in solution box
           H->table[i] = e;
           H->occupied[i] = true;
           return;
        }
        i = (i + 1) % m;
        j++;
    }
    return;     // table full, element gets dropped
}
```

Solution:

```
!H->occupied[i]
|| (H->table[i] != NULL && key_equal(elem_key(H->table[i]), k))
```

Notice that we make use of the short-circuit evaluation of `||` such that we're guaranteed that `H->occupied[i]` within the parenthesized expression.

Here's a state table that might help clarify the variety of conditions that a correct solutions needs to handle. Remember that a position in the array may become occupied, but then later be deleted; that delete operation will assign NULL to that position in the array, but it will *not* set the occupied flag in the hash table to false. Why? So that lookups will not terminate prematurely during probing.

	`H->table[i] == NULL`	`H->table[i] != NULL`
`H->occupied[i]`	(1) Don't write!	(2) Write only if key is the same
`!H->occupied[i]`	(3) Write.	(4) Impossible state.

Why?

1. If element i in the array is already occupied, and the element was deleted (has a NULL value) then that array position is now unusable due to the need to preserve probing lookups.
2. If the element i in the array is already occuped, and the key is the same, then we can safely overwrite the existing value.
3. If the element is not yet occupied, we can of course safely write.

4. It is not possible for an element to have a non-null value without having the occupied flag set (This could be enforced by a data structure invariant).

```
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
//@ensures \result == NULL || key_equal(elem_key(\result), k);
{
    int i = hash(k, H->m);
    int j = 0;
    while (j < H->m) {
        elem e = H->table[i];
        if (____)   // write missing condition below in solution box
            return e;
        i = (i + 1) % m;
        j++;
    }
    return NULL;
}
```

Solution:

```
!H->occupied[i]
|| (H->table[i] != NULL && key_equal(elem_key(H->table[i]), k))
```

	`H->table[i] == NULL`	`H->table[i] != NULL`
`H->occupied[i]`	(1) Deleted.	(2) Check for key match.
`!H->occupied[i]`	(3) Never wrote here.	(4) Impossible state.

Why?

1. We have previously inserted and then deleted here. There's no longer anything to find in our lookup.
2. We have previously inserted here, but not yet deleted. Check if the key at this location matches the requested key.
3. We have never inserted here. Nothing to match.
4. It is not possible for an element to have a non-null value without having the occupied flag set (This could be enforced by a data structure invariant).

```
void ht_delete(ht H, key k)
//@requires is_ht(H);
//@ensures is_ht(H);
{
    int i = hash(k, H->m);
    int j = 0;
    while (j < H->m) {
        elem e = H->table[i];
        if (____)    // write missing condition #1 below in solution box
           return;
        if (____) {  // write missing condition #2 below in solution box
           H->table[i] = NULL;
           return;
        }
        i = (i + 1) % m;
        j++;
    }
}
```

Solution:

Condition 1: `!H->occupied[i]`

Condition 2:

`H->table[i] != NULL && key_equal(elem_key(H->table[i]), k)`

	`H->table[i] == NULL`	`H->table[i] != NULL`
`H->occupied[i]`	(1) Already deleted.	(2) Needs to be deleted.
`!H->occupied[i]`	(3) Never wrote here.	(4) Impossible state.

Why?

1. We have previously inserted and then deleted here. Nothing further needs to be done, so just return.
2. We have previously inserted here, but not yet deleted. We must now delete this element.
3. We have never inserted here. No need to delete.
4. It is not possible for an element to have a non-null value without having the occupied flag set (This could be enforced by a data structure invariant).