```
/* Hash tables (fixed size)
 * 15-122 Principles of Imperative Computation, Fall 2012
 * Frank Pfenning
 */

#use <string>
#use "hash-string.c0"

/*****************************/
/* client-side implementation */
/*****************************/

struct wcount {
  string word;
  int count;
};

int hash(string s, int m) {
  return hash_string(s, m);     /* from hash-string.c0 */
}

bool key_equal(string s1, string s2) {
  return string_equal(s1, s2);
}

string elem_key(struct wcount* wc)
//@requires wc != NULL;
{
  return wc->word;
}

/*************************/
/* client-side interface */
/*************************/
typedef struct wcount* elem;
typedef string key;

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;
  ;

/*************************/
/* library side interface */
/*************************/
struct ht_header;
typedef struct ht_header* ht;

ht ht_new(int capacity)
//@requires capacity > 0;
  ;
elem ht_lookup(ht H, key k);    /* O(1) avg. */
void ht_insert(ht H, elem e)    /* O(1) avg. */
```

Key

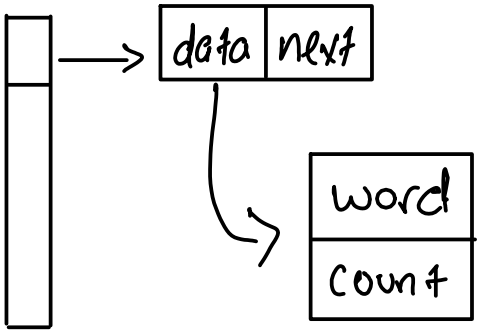
data

A wrapper function for our string hashing code. The library expects (needs) a function with this name and type.

Connecting the types between the client and the library.

Notice how short the client interface with the library is.

```
//@requires e != NULL;
  ;

/*******************************/
/* library-side implementation */
/*******************************/
struct list_node {
  elem data;                    /* data != NULL */
  struct list_node* next;
};
typedef struct list_node list;

struct ht_header {
  int size;                     /* size >= 0 */
  int capacity;                 /* capacity > 0 */
  list*[] table;                /* \length(table) == capacity */
};

/* to be filled in */
/*
bool is_chain(...);
*/

bool is_ht(ht H) {
  if (H == NULL) return false;
  if (!(H->size >= 0)) return false;
  if (!(H->capacity > 0)) return false;
  //@assert \length(H->table) == H->capacity;
  /* check that each element of table is a valid chain */
  /* includes checking that all elements are non-null */
  return true;
}

ht ht_new(int capacity)
//@requires capacity > 0;
//@ensures is_ht(\result);
{
  ht H = alloc(struct ht_header);
  H->size = 0;
  H->capacity = capacity;
  H->table = alloc_array(list*, capacity);
  /* initialized to NULL */
  return H;
}

/* ht_lookup(H, k) returns NULL if key k not present in H */
elem ht_lookup(ht H, key k)
//@requires is_ht(H);
{
  int i = hash(k, H->capacity);
  list* p = H->table[i];
  while (p != NULL)
    // loop invariant: p points to chain
    {
    //@assert p->data != NULL;
    if (key_equal(elem_key(p->data), k))
      return p->data;
    else
      p = p->next;
```

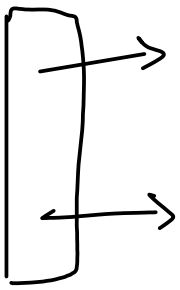


int ref_count

ht H;

Load Factor;

H->size / H->capacity;



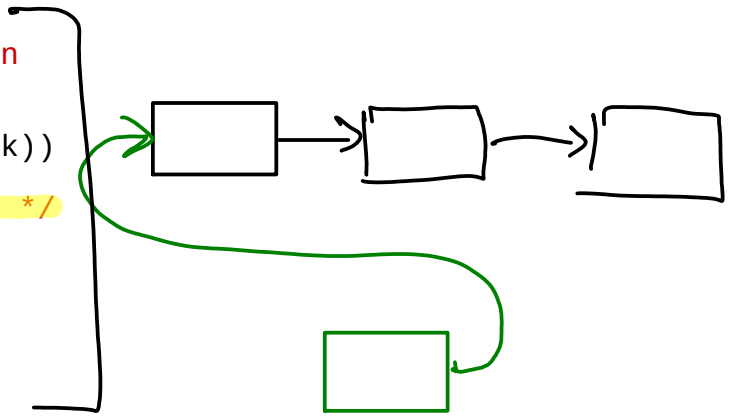
This is a really weak specification function. How can we improve it?

NULL-terminated linked lists--no dummy nodes.

Hash to find the correct bucket and then look through everything in that linked list.

Client functions in action.

```
  }
  /* not in list */
  return NULL;
}

void ht_insert(ht H, elem e)
//@requires is_ht(H);
//@requires e != NULL;
//@ensures is_ht(H);
//@ensures ht_lookup(H, elem_key(e)) != NULL;
{
  key k = elem_key(e);
  int i = hash(k, H->capacity);

  list* p = H->table[i];
  while (p != NULL)
    // loop invariant: p points to chain
    {
      //@assert p->data != NULL;
      if (key_equal(elem_key(p->data), k))
        {
          /* overwrite existing element */
          p->data = e;
          return;
        } else {
        p = p->next;
      }
    }
  //@assert p == NULL;
  /* prepend new element */
  list* q = alloc(struct list_node);
  q->data = e;
  q->next = H->table[i];
  H->table[i] = q;
  (H->size)++;
  return;
}
```



Pay close attention to where we insert!