```
/*
 * Binary search tree without balancing
 *
 * 15-122 Principles of Imperative Computation, Spring 2011
 * Frank Pfenning
 */

/*****************************/
/* Client-side implementation */
/*****************************/
#use <string>

struct wcount {
  string word;                  /* key */
  int count;                    /* data = wordcount */
};

int key_compare(string s1, string s2) {
  return string_compare(s1, s2);
}

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

/*************************/
/* Client-side interface */
/*************************/

typedef struct wcount* elem;
typedef string key;

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

int key_compare(key k1, key k2)
//@ensures -1 <= \result && \result <= 1;
  ;

/*********************/
/* Library interface */
/*********************/

typedef struct bst_header* bst;

bst bst_new();
void bst_insert(bst B, elem e)  /* replace if elem with same key as x in B */
//@requires e != NULL;
  ;
elem bst_lookup(bst B, key k);  /* return NULL if not in tree */

/*************************/
/* Library implementation */
/*************************/

struct tree_node {
  elem data;
  struct tree_node* left;
  struct tree_node* right;
};
typedef struct tree_node tree;

struct bst_header {
```

```
  tree* root;
};

/* is_ordered(T, lower, upper) checks if all elements in T
 * are strictly in the interval (elem_key(lower),elem_key(kupper)).
 * lower = NULL represents -infinity; upper = NULL represents +infinity
 */
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_ordtree(tree* T) {
  /* initially, we have no bounds - pass in NULL */
  return is_ordered(T, NULL, NULL);
}

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

bst bst_new()
//@ensures is_bst(\result);
{
  bst B = alloc(struct bst_header);
  B->root = NULL;
  return B;
}

elem tree_lookup(tree* T, key k)
//@requires is_ordtree(T);
//@ensures \result == NULL || key_compare(elem_key(\result), k) == 0;
{
  if (T == NULL) return NULL;
  int r = key_compare(k, elem_key(T->data));
  if (r == 0)
    return T->data;
  else if (r < 0)
    return tree_lookup(T->left, k);
  else //@assert r > 0;
    return tree_lookup(T->right, k);
}

elem bst_lookup(bst B, key k)
//@requires is_bst(B);
//@ensures \result == NULL || key_compare(elem_key(\result), k) == 0;
{
  return tree_lookup(B->root, k);
}

/* tree_insert(T, e) returns the modified tree
 * this avoids some complications in case T = NULL
 */
tree* tree_insert(tree* T, elem e)
//@requires is_ordtree(T);
//@requires e != NULL;
//@ensures is_ordtree(\result);
{
```

Base cases

```
  if (T == NULL) {
    /* create new node and return it */
    T = alloc(struct tree_node);
    T->data = e;
    T->left = NULL; T->right = NULL;
    return T;
  }
  int r = key_compare(elem_key(e), elem_key(T->data));
  if (r == 0)
    T->data = e;                    /* modify in place */
  else if (r < 0)
    T->left = tree_insert(T->left, e);
  else //@assert r > 0;
    T->right = tree_insert(T->right, e);
  return T;
}

void bst_insert(bst B, elem e)
//@requires is_bst(B);
//@requires e != NULL;
//@ensures is_bst(B);
{
  B->root = tree_insert(B->root, e);
  return;
}
```

base case

worst -> O(n)
ideal -> O(log n)

① Base Case

100

② Recursive Case

③ Termination