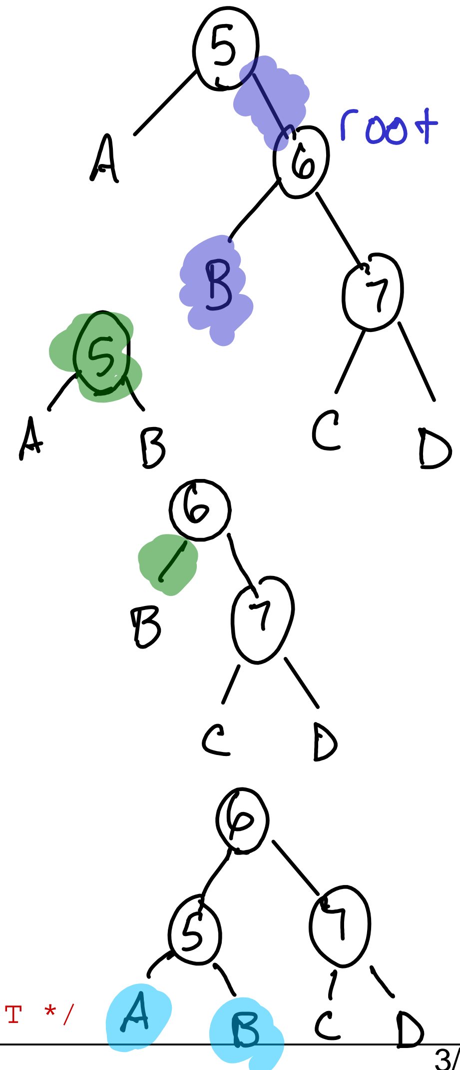
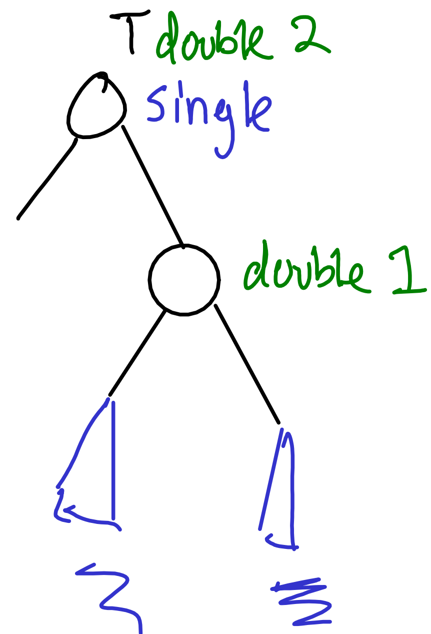
```
/*
 * AVL Trees
 * Ephemeral (imperative) version
 *
 * 15-122 Principles of Imperative Computation, Fall 2012
 * Frank Pfenning
 */

/******************************/
/* 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 */
/**************************/
```

```
typedef struct tree_node tree;

struct tree_node {
  elem data;
  int height;
  tree* left;
  tree* right;
};

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);
}

/* height(T) returns the precomputed height of T in O(1) */
int height(tree* T) {
  return T == NULL ? 0 : T->height;
}

bool is_balanced(tree* T) {
  if (T == NULL) return true;
  int h = T->height;
  int hl = height(T->left);
  int hr = height(T->right);
  if (!(h == (hl > hr ? hl+1 : hr+1))) return false;
  if (hl > hr+1 || hr > hl+1) return false;
  return is_balanced(T->left) && is_balanced(T->right);
}

bool is_avl(tree* T) {
  return is_ordtree(T) && is_balanced(T);
}

tree* leaf(elem e)
//@requires e != NULL;
//@ensures is_avl(\result);
{
  tree* T = alloc(struct tree_node);
  T->left = NULL;
```

```
  T->data = e;
  T->right = NULL;
  T->height = 1;
  return T;
}

/* fix_height(T) calculates the height of T and stores
 * it in T->height, assuming the height of the subtrees
 * is correct.  The result will have correct height, but
 * may not yet be balanced if this is part of a double rotation.
 */
void fix_height(tree* T)
//@requires T != NULL;
//@requires is_balanced(T->left) && is_balanced(T->right);
{
  int hl = height(T->left);
  int hr = height(T->right);
  T->height = (hl > hr ? hl+1 : hr+1);
  return;
}

/* rotate_right(T) may not be balanced if first step
 * of a double rotation, but heights will be accurate.
 */
tree* rotate_right(tree* T)
//@requires is_ordtree(T);
//@requires T != NULL && T->left != NULL;
//@ensures is_ordtree(\result);
//@ensures \result != NULL && \result->right != NULL;
{
  tree* root = T->left;
  T->left = root->right;
  root->right = T;
  fix_height(root->right);      /* must be first */
  fix_height(root);
  return root;
}

/* rotate_left(T) may not be balanced if first step
 * of a double rotation, but heights will be accurate.
 */
tree* rotate_left(tree* T)
//@requires is_ordtree(T);
//@requires T != NULL && T->right != NULL;
//@ensures is_ordtree(\result);
//@ensures \result != NULL && \result->left != NULL;
{
  tree* root = T->right;
  T->right = root->left;
  root->left = T;
  fix_height(root->left);      /* must be first */
  fix_height(root);
  return root;
}

tree* rebalance_left(tree* T)
//@requires T != NULL;
//@requires is_avl(T->left) && is_avl(T->right);
/* also requires that T->left is result of insert into T */
```



```
//@ensures is_avl(\result);
{
  tree* l = T->left;
  tree* r = T->right;
  int hl = height(l);
  int hr = height(r);
  if (hl > hr+1) {
    //@assert hl == hr+2;
    if (height(l->left) > height(l->right)) {
      //@assert height(l->left) == hr+1;
      T = rotate_right(T);
      //@assert height(T) == hr+2;
      return T;
    } else {
      //@assert height(l->right) == hr+1;
      /* double rotate right */
      T->left = rotate_left(T->left);
      T = rotate_right(T);
      //@assert height(T) == hr+2;
      return T;
    }
  } else { //@assert !(hl > hr+1);
    fix_height(T);
    return T;
  }
}

tree* rebalance_right(tree* T)
//@requires T != NULL;
//@requires is_avl(T->left) && is_avl(T->right);
/* also requires that T->right is result of insert into T */
//@ensures is_avl(\result);
{
  tree* l = T->left;
  tree* r = T->right;
  int hl = height(l);
  int hr = height(r);
  if (hr > hl+1) {
    //@assert hr == hl+2;
    if (height(r->right) > height(r->left)) {
      //@assert height(r->right) == hl+1;
      T = rotate_left(T);
      //@assert height(T) == hl+2;
      return T;
    } else {
      //@assert height(r->left) == hl+1;
      /* double rotate left */
      T->right = rotate_right(T->right);
      T = rotate_left(T);
      //@assert height(T) == hl+2;
      return T;
    }
  } else { //@assert !(hr > hl+1);
    fix_height(T);
    return T;
  }
}

bool is_bst(bst B) {
```



```
  if (B == NULL) return false;
  return is_avl(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_avl(T);
//@requires e != NULL;
//@ensures is_avl(\result);
{
  if (T == NULL) {
    T = leaf(e);                /* create new leaf with data e */
  } else {
    int r = key_compare(elem_key(e), elem_key(T->data));
    if (r < 0) {
      T->left = tree_insert(T->left, e);
      T = rebalance_left(T);    /* also fixes height */
    } else if (r == 0) {
      T->data = e;
    } else { //@assert r > 0;
      T->right = tree_insert(T->right, e);
      T = rebalance_right(T);   /* also fixes height */
    }
  }
  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;
}
```