**/\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\* INTERFACE \*/**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**// typedef \_\_\_\_\_\_\* elem;**

**typedef void\* elem;**

**typedef int compare\_fn(elem e1, elem e2)**

**/\*@requires e1 != NULL && e2 != NULL; @\*/**

**/\*@ensures -1 <= \result && \result <= 1; @\*/ ;**

**// typedef \_\_\_\_\_\_\* bst\_t;**

**typedef struct bst\_header\* bst\_t;**

**bst\_t bst\_new(compare\_fn\* compare)**

**/\*@requires compare != NULL; @\*/**

**/\*@ensures \result != NULL; @\*/ ;**

**bool bst\_lookup(bst\_t B, elem x)**

**/\*@requires B != NULL && x != NULL; @\*/ ;**

**void bst\_insert(bst\_t B, elem x)**

**/\*@requires B != NULL && x != NULL; @\*/**

**/\*@requires !bst\_lookup(B, x); @\*/ ;**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\* IMPLEMENTATION \*/**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**typedef struct tree\_node tree;**

**struct tree\_node {**

**elem data;**

**tree\* left;**

**tree\* right;**

**};**

**typedef struct bst\_header bst;**

**struct bst\_header {**

**tree\* root;**

**compare\_fn\* compare;**

**};**

**bool is\_ordered(tree\* T, elem lower, elem upper, compare\_fn\* compare)**

**//@requires compare != NULL;**

**{**

**if (T == NULL) return true;**

**if (T->data == NULL) return false;**

**elem mid = T->data;**

**if (!(lower == NULL || (\*compare)(lower, mid) < 0)) return false;**

**if (!(upper == NULL || (\*compare)(mid, upper) < 0)) return false;**

**return is\_ordered(T->left, lower, mid, compare)**

**&& is\_ordered(T->right, mid, upper, compare);**

**}**

**bool is\_tree(tree\* T, compare\_fn\* compare)**

**//@requires compare != NULL;**

**{**

**return is\_ordered(T, NULL, NULL, compare);**

**}**

**bool is\_bst(bst\* B) {**

**return B != NULL && B->compare != NULL && is\_tree(B->root, B->compare);**

**}**

**bst\* bst\_new(compare\_fn\* compare)**

**//@requires compare != NULL;**

**//@ensures is\_bst(\result);**

**{**

**bst\* B = alloc(struct bst\_header);**

**B->root = NULL;**

**B->compare = compare;**

**return B;**

**}**

**bool tree\_lookup(tree\* T, elem x, compare\_fn\* compare)**

**//@requires compare != NULL && is\_tree(T, compare);**

**{**

**if (T == NULL) return false;**

**int r = (\*compare)(x, T->data);**

**if (r == 0) {**

**return true;**

**} else if (r < 0) {**

**return tree\_lookup(T->left, x, compare);**

**} else {**

**//@assert r > 0;**

**return tree\_lookup(T->right, x, compare);**

**}**

**}**

**bool bst\_lookup(bst\* B, elem x)**

**//@requires is\_bst(B) && x != NULL;**

**{**

**return tree\_lookup(B->root, x, B->compare);**

**}**

**tree\* leaf(elem x) {**

**/\* create new node and return it \*/**

**tree\* T = alloc(struct tree\_node);**

**T->data = x;**

**return T;**

**}**

**tree\* tree\_insert(tree\* T, elem x, compare\_fn\* compare)**

**//@requires x != NULL && compare != NULL && is\_tree(T, compare);**

**//@requires !tree\_lookup(T, x, compare);**

**//@ensures is\_tree(\result, compare);**

**{**

**if (T == NULL) return leaf(x);**

**if ((\*compare)(x, T->data) < 0) {**

**T->left = tree\_insert(T->left, x, compare);**

**} else {**

**//@assert (\*compare)(x, T->data) > 0;**

**T->right = tree\_insert(T->right, x, compare);**

**}**

**return T;**

**}**

**void bst\_insert(bst\* B, elem x)**

**//@requires is\_bst(B) && x != NULL;**

**//@requires !bst\_lookup(B, x);**

**//@ensures is\_bst(B);**

**{**

**B->root = tree\_insert(B->root, x, B->compare);**

**return;**

**}**