```
/*AVLTree.h
 *
 *Dylan Jeffers
 *Tahmid Rahman
 *This file taken from Joshua Brody's CS31 Class
 *Taught Fall of 2014 @ Swarthmore College
 */
#ifndef AVLTREE_H_
#define AVLTREE_H_

#include "BST.h"
#include "pair.h"
#include "library/queue.h"

// Forward declaration of the AVLTreeNode class
template <typename K, typename V> class AVLTreeNode;

/**
 *  A AVLTree is a templated binary search tree, implementing the
 *  BST interface (see BST.h).  This implementation is similar
 *  to LinkedBST except:
 *    (1) Each AVLTreeNode stores the height of its sub-tree
 *    (2) An AVLTree is balanced according to the AVL property:
 *         the difference between the height of two child nodes is at most 1
 */
template <typename K, typename V>
class AVLTree : public BST<K,V> {
  private:
    int size;            // Current number of items in the tree.
    AVLTreeNode<K,V>* root;  // Pointer to the root node (possibly NULL).

  public:
    AVLTree();
    ~AVLTree();

    /* All public functions declared/detailed in BST.h*/
    /* These methods are defined in AVLTree-inl.h*/

    /* sizing operations */
    int getSize();
    bool isEmpty();
    int  getHeight();

    /* test operations */
    bool isBalanced();

    /* Key operations */
    K    getMax();
    K    getMin();

    /* dictionary operations */
    void insert  (K key, V value);
    void update  (K key, V value);
    bool contains(K key);
    void remove  (K key);
    V    find    (K key);

    /* traversal operations */
    Queue< Pair<K,V> >* getPreOrder();
    Queue< Pair<K,V> >* getInOrder();
    Queue< Pair<K,V> >* getPostOrder();
    Queue< Pair<K,V> >* getLevelOrder();

  private:
    /* Private recursive internal methods that
     * correspond to the public methods defined above.
     * These methods are defined in AVLTree-private-inl.h
     */

    /*
     * isBalancedInSubtree - Recursive function that tests whether a
     *                       a subtree is balanced
     * Note: all AVLTrees _should_ be balanced, so if this tree is not,
     *  there's something wrong with the implementation.
     *
     * @param current : a pointer to the root of the subtree
     * @return bool: true iff the AVL subtree is indeed balanced.
     */
    bool isBalancedInSubtree(AVLTreeNode<K,V>* current);

    /* insertInSubtree - Recursive function that inserts a new node into
     *                   a sub-tree pointed to by current
     * @param current : a pointer to the root of the sub-tree
     * @param key : the key for the new node being inserted
     * @param value : the value for the new node being inserted
     * @error runtime_error if the key already exists
     * @return AVLTreeNode<K,V>*: the root of the sub-tree.
     */
    AVLTreeNode<K,V>* insertInSubtree(AVLTreeNode<K,V>* current, K key, V value);

    /* updateInSubtree - Recursive function that updates a key-value pair
     *                   in the tree
     * @param current : pointer to root node of the sub-tree
     * @param key : the key being searched for
     * @param value : the new value to associate with the given key
     * @error runtime_error if key not found
     */
    void updateInSubtree(AVLTreeNode<K,V>* current, K key, V value);

    /* removeFromSubtree - Recursive function that searches for and removes
     *                   the element associated with the given key in the
     *                   sub-tree pointed to by current
     * @param current : a pointer to the root of the sub-tree
     * @param key : the key for the node to be removed
     * @error runtime_error if the key does not exist
     * @return AVLTreeNode<K,V>*: the root of the sub-tree.
     */
    AVLTreeNode<K,V>* removeFromSubtree (AVLTreeNode<K,V>* current, K key);

    /* containsInSubtree - Recursive function that checks if the sub-tree
     *                 pointed by current contains the given key
     * @param current : pointer to root node of the sub-tree
     * @param key : the key being searched for
     * @return bool : true if the key was found
     */
    bool containsInSubtree (AVLTreeNode<K,V>* current, K key);

    /* findInSubtree - Recursive function that returns the value associated
```

```
 *                  with the given key in the sub-tree
 *                  pointed by current
 *@param current : pointer to root node of the sub-tree
 *@param key : the key being searched for
 *@error runtime_error if the key is not in the sub-tree
 *@return V : the value associated with the search key
 */
V findInSubtree(AVLTreeNode<K,V>* current, K key);

/* getMaxInSubtree - Recursive function that retrieves the maximal key
 *                    in the sub-tree pointed to by current
 *
 * @param  current: pointer to root node of the sub-tree
 * @return K : the maximal key in the subtree
 */
K  getMaxInSubtree(AVLTreeNode<K,V>* current);

/* getMinInSubtree - Recursive function that retrieves the minimal key
 *                    in the sub-tree pointed to by current
 *
 * @param  current: pointer to root node of the sub-tree
 * @return K : the minimal key in the subtree
 */
K getMinInSubtree(AVLTreeNode<K,V>* current);

/* build{Pre,In,Post} - Recursive helper functions for building
 *                       iterators for the data structure.
 *                       Each enqueues all key-value pairs for the
 *                       sub-tree pointed to by current
 *
 * @param current : pointer to root node of the sub-tree
 * @param it : a pointer to the Queue to fill with the key-value
 *                              pairs based on the traversal order
 */
void buildPreOrder (AVLTreeNode<K,V>* current, Queue< Pair<K,V> >* it);
void buildInOrder  (AVLTreeNode<K,V>* current, Queue< Pair<K,V> >* it);
void buildPostOrder(AVLTreeNode<K,V>* current, Queue< Pair<K,V> >* it);

/* traverseAndDelete - Recursive helper function for the destructor
 *                     Performs a post-order traversal of the sub-tree
 *                     freeing memory for all nodes in the sub-tree
 *                     pointed to by current
 * @param current : pointer to root node of the sub-tree to be deleted
 */
void traverseAndDelete (AVLTreeNode<K,V>* current);

/*These methods are unique to the AVLTree relative to LinkedBST. Each
 * maintains/ensures a balanced tree according to the AVL property
 */

/* computeHeightFromChildren - updates height for a node by checking
 *       child nodes
 * @param current - root of sub-tree whose height needs to be
 *                  updated
 */
void computeHeightFromChildren(AVLTreeNode<K,V>* current);

/* balance - updates heights after insert/remove, detects imbalances,
 *            and invokes rotation to fix an imbalance
```

```
     * @param current : pointer to the root node of sub-tree to be balanced
     * @return AVLTreeNode<K,V>* pointer to root of sub-tree (current if
     *         no rotations are made)
     */
    AVLTreeNode<K,V>* balance(AVLTreeNode<K,V>* current);

    /* The four rotations needed to fix each of the four possible imbalances
     *   in an AVLTree
     *   (1) Right rotation for a left-left imbalance
     *   (2) Left rotation for a right-right imbalance
     *   (3) LeftRight rotation for left-right imbalance
     *   (4) RightLeft rotation for a right-left imbalance
     * @param current : pointer to root of sub-tree to be balanced
     * @return AVLTreeNode<K,V>* pointer to root of updated sub-tree
     */
    AVLTreeNode<K,V>* rightRotate(AVLTreeNode<K,V>* current);
    AVLTreeNode<K,V>* leftRightRotate(AVLTreeNode<K,V>* current);
    AVLTreeNode<K,V>* leftRotate(AVLTreeNode<K,V>* current);
    AVLTreeNode<K,V>* rightLeftRotate(AVLTreeNode<K,V>* current);
};


/******************************************************************************/


/**
 * The AVLTreeNode is a templated class that stores data for each node
 * in the AVLTree.
 */
template <typename K, typename V>
class AVLTreeNode {
  private:
    K key;                      // The key stored in this node.
    V value;                    // The value stored in this node.
    AVLTreeNode<K,V>* left;         // Pointer to this node's left subtree.
    AVLTreeNode<K,V>* right;        // Pointer to this node's right subtree.
    int height;

    /* default constructor */
    AVLTreeNode();

    /* preferred constructor to initialize key and value to given
     * parameters; left and right are set to NULL
     */
    AVLTreeNode(K k, V v);

    /* getHeight checks for NULL for you, returning -1*/
    int getHeight();

  /* indicates that AVLTree is a friend class so that it can directly
   * access private aspects of this class.
   */
  friend class AVLTree<K,V>;
};

//all the public methods are defined here as well as the AVLTreeNode class
#include "AVLTree-inl.h"

//all the private methods are defined here
```

```
#include "AVLTree-private-inl.h"

#endif  // AVLTREE_H_
```