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

#include <stdexcept>
#include "library/arrayQueue.h"

////////////////////////////////////////////////////////////////////////
////////////////////////////AVLTREE NODE IMPLEMENTATION/////////////////////
////////////////////////////////////////////////////////////////////////
/**
 * Default constructor for the AVLTreeNode class.
 *   Does not set the key-value pair, and initializes the subtrees to NULL.
 */
template <typename K, typename V>
AVLTreeNode<K,V>::AVLTreeNode() {
  height = -1;  //An empty tree has height -1
  left = NULL;
  right = NULL;
}


/**
 * Standard constructor for the AVLTreeNode class.
 * Stores the given key-value pair and initializes the subtrees to NULL.
 * @param k - key for the element (index in the overall BST)
 * @param v - value for the element
 */
template <typename K, typename V>
AVLTreeNode<K,V>::AVLTreeNode(K k, V v) {
  key = k;
  value = v;
  height = 0;
  left = NULL;
  right = NULL;
}

////////////////////////////////////////////////////////////////////////
////////////////////////////AVLTree IMPLEMENTATION/////////////////////
////////////////////////////////////////////////////////////////////////

/**
 * Standard constructor for the AVLTree class.
 *   Constructs an empty tree (size 0, with a NULL root).
 */
template <typename K, typename V>
AVLTree<K,V>::AVLTree() {
  size = 0;
  root = NULL;
}

/**
 * The AVLTree destructor uses a recursive helper function which
 * traverses the tree and frees each node on the heap as it traverses.
 */
```

```
template <typename K, typename V>
AVLTree<K,V>::~AVLTree() {
  traverseAndDelete(root);
}

/* getSize - returns the size of the BST
 * @return int: the number of key-value pairs in the data structure
 */
template <typename K, typename V>
int AVLTree<K,V>::getSize() {
  return size;
}

/* isEmpty  - returns true if the tree is empty
 * @return bool: true if there are no elements in the BST
 */
template <typename K, typename V>
bool AVLTree<K,V>::isEmpty() {
  return size == 0;
}

/* isBalanced- returns true if the tree is balanced
 * @return bool: true if the tree is balanced.
 */
template <typename K, typename V>
bool AVLTree<K,V>::isBalanced() {
  return isBalancedInSubtree(root);
}

/* insert - inserts the key-value pair into the tree
 * @param key - key for indexing the new element
 * @param value - value associated with the given key
 * @error runtime_error if they key already exists
 */
template <typename K, typename V>
void AVLTree<K,V>::insert(K key, V value) {
  root = insertInSubtree(root, key, value);
}

/* update - finds the element indexed by the given key and updates
 *          its value to the provided value parameter
 * @param key - key for finding the existing element
 * @param value - the new value to store for the given key
 * @error runtime_error if the key is not found in the BST
 */
template <typename K, typename V>
void AVLTree<K,V>::update(K key, V value){
  updateInSubtree(root, key, value);
}

/* remove - deletes the element with given key from the tree
 * @param key - index key to search for and remove
 * @error: runtime_error if they key is not found
 */
template <typename K, typename V>
void AVLTree<K,V>::remove(K key) {
  root = removeFromSubtree(root, key);
}
```

```
/* contains - returns true if there exists an element in the BST
 *   with the given key
 * @param key - index key to search for
 * @return bool: true if the given key exists in the BST
 */
template <typename K, typename V>
bool AVLTree<K,V>::contains(K key) {
  return containsInSubtree(root, key);
}

/* find - returns the value associated with the given key
 * @param key - index key for element to find
 * @error runtime_error if the key is not found in the BST
 * @return V: value associated with given key
 */
template <typename K, typename V>
V AVLTree<K,V>::find(K key) {
  return findInSubtree(root, key);
}


/* getMin - returns the smallest key in the data structure
 * @error runtime_error if BST is empty
 * @return K: minimum key in the BST
 */
template <typename K, typename V>
K AVLTree<K,V>::getMin() {
  if (isEmpty()) {
    throw std::runtime_error("AVLTree::getMin called on an empty tree.");
  }
  return getMinInSubtree(root);
}

/* getMax - returns the largest key in the data structure
 * @error runtime_error if BST is empty
 * @return K: maximum key in the BST
 */
template <typename K, typename V>
K AVLTree<K,V>::getMax() {
  if (isEmpty()) {
    throw std::runtime_error("AVLTree::getMax called on an empty tree.");
  }
  return getMaxInSubtree(root);
}

/* getHeight - returns a height for the tree (i.e., largest
 *   depth for any leaf node)
 * @return int: height of tree, -1 if tree is empty
 */
template <typename K, typename V>
int AVLTree<K,V>::getHeight() {
  if(root == NULL){
    return -1;
  } else{
    return root->height;
  }
}
```

```
/* getPreOrder - returns a pointer to an iterator (a queue here) containing
 *               all key-value pairs in the data structure.  Uses a pre-order
 *               traversal to obtain all elements
 * @return Queue< Pair<K,V>>*: a pointer to a dynamically allocated
 *   Queue with key-value pairs.  The caller is responsible for handling
 *   the heap memory deallocation
 */
template <typename K, typename V>
Queue< Pair<K,V> >* AVLTree<K,V>::getPreOrder() {
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();
  buildPreOrder(root, it);
  return it;
}


/* getInOrder - returns a pointer to an iterator (a queue here) containing
 *               all key-value pairs in the data structure.  Uses an in-order
 *               traversal to obtain all elements
 * @return Queue< Pair<K,V>>*: a pointer to a dynamically allocated
 *   Queue with key-value pairs.  The caller is responsible for handling
 *   the heap memory deallocation
 */
template <typename K, typename V>
Queue< Pair<K,V> >* AVLTree<K,V>::getInOrder() {
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();
  buildInOrder(root, it);
  return it;
}

/* getPostOrder - returns a pointer to an iterator (a queue here) containing
 *               all key-value pairs in the data structure.  Uses a post-order
 *               traversal to obtain all elements
 * @return Queue< Pair<K,V>>*: a pointer to a dynamically allocated
 *   Queue with key-value pairs.  The caller is responsible for handling
 *   the heap memory deallocation
 */
template <typename K, typename V>
Queue< Pair<K,V> >* AVLTree<K,V>::getPostOrder() {
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();
  buildPostOrder(root, it);
  return it;
}

/* getLevelOrder - returns a pointer to an iterator (a queue here) containing
 *               all key-value pairs in the data structure.  Uses a level-order
 *               traversal to obtain all elements
 * @return Queue< Pair<K,V>>*: a pointer to a dynamically allocated
 *   Queue with key-value pairs.  The caller is responsible for handling
 *   the heap memory deallocation
 */
template <typename K, typename V>
Queue< Pair<K,V> >* AVLTree<K,V>::getLevelOrder() {
  ArrayQueue< AVLTreeNode<K,V>* > levelQ;
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();

  levelQ.enqueue(root);
  while (!levelQ.isEmpty()) {
    AVLTreeNode<K,V>* current = levelQ.dequeue();
    if (current != NULL) {
```

```
      it->enqueue( Pair<K,V>(current->key, current->value) );
      levelQ.enqueue(current->left);
      levelQ.enqueue(current->right);
    }
  }

  return it;
}
```