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

/*
 * This recursive helper function inserts a key-value pair into a subtree
 * of the tree, or throws a runtime_error if the key is already present.
 */
template <typename K, typename V>
AVLTreeNode<K,V>*
AVLTree<K,V>::insertInSubtree(AVLTreeNode<K,V>* current, K key, V value) {

  if (current == NULL){
    size++;
    return new AVLTreeNode<K, V>(key, value);
  }
  else if (key == current->key){
    throw std::runtime_error("AVLTree::insertInSubtree" \
      "called on key already in tree.");
  }
  else if (key < current->key){
    current->left = insertInSubtree(current->left, key, value);
  }
  else if (key > current->key){
    current->right = insertInSubtree(current->right, key, value);
  }
  return balance(current);
}

/**
 * This recursive helper function updates key-value pair in the subtree
 * of the tree, or throws a runtime_error if the key is not present.
 */
template <typename K, typename V>
void AVLTree<K,V>::updateInSubtree(AVLTreeNode<K,V>* current, K key, V value) {

  if (current == NULL){
    throw std::runtime_error("Key not found in AVLTree::updateInSubtree.");
  }
  else if (key == current->key){
    current->value = value;
  }
  else if (key < current->key){
    updateInSubtree(current->left, key, value);
  }
  else if (key > current->key){
    updateInSubtree(current->right, key, value);
  }
  return;
}


/**
 * This recursive helper function removes a key-value pair from a subtree
 * of the tree, or throws a runtime_error if that key was not present.
```

```
 *
 * It returns a pointer to the root of the subtree.  This root is often
 * the node that was passed as an argument to the function (current) but
 * might be a different node if current contains the key we are removing
 * from the tree.
 */
template <typename K, typename V>
AVLTreeNode<K,V>*
AVLTree<K,V>::removeFromSubtree(AVLTreeNode<K,V>* current,
                                  K key) {
  if (current == NULL) {
    throw std::runtime_error("AVLTree::remove called on key not in tree.");
  }

  else if (key == current->key) {       // We've found the node to remove
    if ((current->left == NULL) && (current->right == NULL)) {
      size--;
      delete current;
      return NULL;
    }
    else if (current->left == NULL) {
      AVLTreeNode<K,V>* tempNode = current->right;
      delete current;
      size--;
      return balance(tempNode);
    }
    else if (current->right == NULL) {
      AVLTreeNode<K,V>* tempNode = current->left;
      delete current;
      size--;
      return balance(tempNode);
    }
    else {
      AVLTreeNode<K,V>* minimum = current->right;
      while (minimum->left != NULL) {
        minimum = minimum->left;
      }
      current->key = minimum->key;
      current->value = minimum->value;
      current->right = removeFromSubtree(current->right, current->key);
    }
  }

  else if (key < current->key) {
    current->left = removeFromSubtree(current->left, key);
  }
  else {
    current->right = removeFromSubtree(current->right, key);
  }
  return balance(current);
}


/**
 * Returns true if a key is contained in a subtree of the tree, and
 * false otherwise.
 */
template <typename K, typename V>
bool AVLTree<K,V>::containsInSubtree(AVLTreeNode<K,V>* current, K key) {
```

```
  if (current == NULL){
    return false;
  }
  else if (key == current->key){
    return true;
  }
  else if (key < current->key){
    return containsInSubtree(current->left, key);
  }
  else {
    return containsInSubtree(current->right, key);
  }
}


/**
 * Given a key, returns the value for that key from a subtree of the tree.
 * Throws a runtime_error if the key is not in the subtree.
 */
template <typename K, typename V>
V AVLTree<K,V>::findInSubtree(AVLTreeNode<K,V>* current, K key) {
  if (current == NULL) {
    throw std::runtime_error("LinkedBS::findInSubtree called on an empty tree");
  }
  else if (key == current->key) {
    return current->value;
  }
  else if (key < current->key) {
    return findInSubtree(current->left, key);
  }
  else {
    return findInSubtree(current->right, key);
  }
}


/**
 * Returns the largest key in a subtree of the tree.
 */
template <typename K, typename V>
K AVLTree<K,V>::getMaxInSubtree(AVLTreeNode<K,V>* current) {
  if (current->right == NULL) {
    return current->key;
  }
  return getMaxInSubtree(current->right);
}


/**
 * Returns the smallest key in a subtree of the tree.
 */
template <typename K, typename V>
K AVLTree<K,V>::getMinInSubtree(AVLTreeNode<K,V>* current) {
  if (current->left == NULL) {
    return current->key;
  }
  return getMinInSubtree(current->left);
}
```

```
/*
 * Returns the height of a subtree of the tree, or -1 if the subtree
 * is empty.
 *
template <typename K, typename V>
int AVLTree<K,V>::getHeightOfSubtree(AVLTreeNode<K,V>* current) {
  if (current == NULL) {
    return -1;
  }
  int l = getHeightOfSubtree(current->left);
  int r = getHeightOfSubtree(current->right);
  if (l >= r) {
    return ++l;
  }
  else
    return ++r;
}
*/


/**
 * Recursively builds a post-order iterator for a subtree of the tree.
 */
template <typename K, typename V>
void AVLTree<K,V>::buildPostOrder(AVLTreeNode<K,V>* current,
                                       Queue< Pair<K,V> >* it) {
  if (current == NULL) {
    return;
  }
  buildPostOrder(current->left, it);
  buildPostOrder(current->right, it);
  it->enqueue( Pair<K,V>(current->key, current->value) );
}

/**
 * Recursively builds a pre-order iterator for a subtree of the tree.
 */
template <typename K, typename V>
void AVLTree<K,V>::buildPreOrder(AVLTreeNode<K,V>* current,
                                      Queue< Pair<K,V> >* it) {
  if (current == NULL){
    return;
  }
  it->enqueue( Pair<K,V>(current->key, current->value) );
  buildPreOrder(current->left, it);
  buildPreOrder(current->right, it);
}


/**
 * Recursively builds an in-order iterator for a subtree of the tree.
 */
template <typename K, typename V>
void AVLTree<K,V>::buildInOrder(AVLTreeNode<K,V>* current,
                                      Queue< Pair<K,V> >* it) {
  if (current == NULL){
    return;
  }
```

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


/**
 * Performs a post-order traversal of the tree, deleting each node from the
 * heap after we have already traversed its children.
 */
template <typename K, typename V>
void AVLTree<K,V>::traverseAndDelete(AVLTreeNode<K,V>* current) {
  if (current == NULL) {
    return;  //nothing to delete
  }
  traverseAndDelete(current->left);
  traverseAndDelete(current->right);
  delete current;
}

/**
 * Returns true if balance factor of subtree is between -1 and 1 (inclusive)
 * If a child is NULL, we treat the child's height as -1
 */

template<typename K, typename V>
bool AVLTree<K,V>::isBalancedInSubtree(AVLTreeNode<K,V>* current) {
  int leftHeight, rightHeight;

  if (current == NULL){
    return true;
  }
  else{
    if (current->left == NULL) {
      leftHeight = -1;
    }
    else {
      leftHeight = current->left->height;
    }
    if (current->right == NULL) {
      rightHeight = -1;
    }
    else {
      rightHeight = current->right->height;
    }
    int balanceFactor = leftHeight - rightHeight;
    if (balanceFactor >= 2 || balanceFactor <= -2) {
      return false;
    }
    else {
      return true;
    }
  }
}

/**
 * Computes height of a node from heights of children
 * BROUGHT TO YOU BY A HEALTHY COMPUTATION FROM RAHMROMJEFFERS
 * If a child is NULL, we treat the child's height as -1
```

```
 */
template<typename K, typename V>
void AVLTree<K,V>::computeHeightFromChildren(AVLTreeNode<K,V>* current) {
  int leftHeight, rightHeight;

  if (current->left == NULL) {
    leftHeight = -1;
  }
  else {
    leftHeight = current->left->height;
  }
  if (current->right == NULL) {
    rightHeight = -1;
  }
  else {
    rightHeight = current->right->height;
  }

  if (leftHeight >= rightHeight) {
    current->height = leftHeight + 1;
  }
  else {
    current->height = rightHeight + 1;
  }
}

/* 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
*/

template<typename K, typename V>
AVLTreeNode<K,V>* AVLTree<K,V>::rightRotate(AVLTreeNode<K,V>* current) {
  AVLTreeNode<K,V>* b = current;
  AVLTreeNode<K,V>* d = current->left;
  current = d;
  b->left = d->right;
  d->right = b;
  computeHeightFromChildren(b);
  computeHeightFromChildren(current);
  return current;
}

template<typename K, typename V>
AVLTreeNode<K,V>* AVLTree<K,V>::leftRightRotate(AVLTreeNode<K,V>* current) {
  current->left = leftRotate(current->left);
  return rightRotate(current);
}

template<typename K, typename V>
AVLTreeNode<K,V>* AVLTree<K,V>::leftRotate(AVLTreeNode<K,V>* current) {
  AVLTreeNode<K,V>* b = current;
  AVLTreeNode<K,V>* d = current->right;
  current = d;
  b->right = d->left;
  d->left = b;
```

```
  computeHeightFromChildren(b);
  computeHeightFromChildren(current);
  return current;
}

template<typename K, typename V>
AVLTreeNode<K,V>* AVLTree<K,V>::rightLeftRotate(AVLTreeNode<K,V>* current) {
  current->right = rightRotate(current->right);
  return leftRotate(current);
}

/**
 * Balances subtree with current as root, identifying the imbalance and calling
 * the proper rotation method
 */
template<typename K, typename V>
AVLTreeNode<K,V>* AVLTree<K,V>::balance(AVLTreeNode<K,V>* current) {
  if (current == NULL) {
    return current;
  }
  else {
    computeHeightFromChildren(current);
    int leftHeight, rightHeight, balanceFactor;
    if (current->left == NULL) {
      leftHeight = -1;
    }
    else {
      leftHeight = current->left->height;
    }
    if (current->right == NULL) {
      rightHeight = -1;
    }
    else {
      rightHeight = current->right->height;
    }

    balanceFactor = leftHeight - rightHeight;
    if (balanceFactor >= 2){ // left imbalance
      int left_right, left_left;

      if (current->left->left == NULL) {
        left_left = -1;
      }
      else {
        left_left = current->left->left->height;
      }
      if (current->left->right == NULL) {
        left_right = -1;
      }
      else {
        left_right = current->left->right->height;
      }

      if (left_left >= left_right){
        return rightRotate(current);
      }
      else{
        return leftRightRotate(current);
      }
```

```
    }
    else if (balanceFactor <= -2) { // right imbalance
      int right_right, right_left;

      if (current->right->right == NULL) {
        right_right = -1;
      }
      else {
        right_right = current->right->right->height;
      }
      if (current->right->left == NULL) {
        right_left = -1;
      }
      else {
        right_left = current->right->left->height;
      }

      if(right_right >= right_left){
        return leftRotate(current);
      }
      else{
        return rightLeftRotate(current);
      }
    }
  }
  return current;
}
```