```
/*SPLAVL-private-inl.h
 *Tahmid Rahman
 *Dylan Jeffers
 */

/* This recursive definition inserts a new node
 * or continues following down the subtrees to find
 * where to place the new node
 *
 * It splays or balances on the way up, depending
 * on the current and max ratio values
 */

template <typename K, typename V>
SPLAVLNode<K,V>*
SPLAVL<K,V>::insertInSubtree(SPLAVLNode<K,V>* current, K key, V value) {

  if (current == NULL){
    size++;
    return new SPLAVLNode<K, V>(key, value);
  }
  else if (key == current->key){
    throw std::runtime_error("SPLAVL::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);
  }

  if (currentRatio > maxRatio){
    return balance(current);
  }
  else{
    return splay(current, key);
  }
}


/**
 * This recursive helper function removes a key-value pair from a subtree
 * of the tree (balancing as we go when necessary), or, if the key was not
 * found, it throws a runtime_error.
 *
 * 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>
SPLAVLNode<K,V>*
SPLAVL<K,V>::removeFromSubtree(SPLAVLNode<K,V>* current,
                               K key) {
  if (current == NULL) {
    throw std::runtime_error("SPLAVL::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) {
      SPLAVLNode<K,V>* tempNode = current->right;
      delete current;
      size--;
      if (currentRatio > maxRatio){
        return balance(tempNode);
      }
      else{
        return tempNode;
      }
    }
    else if (current->right == NULL) {
      SPLAVLNode<K,V>* tempNode = current->left;
      delete current;
      size--;
      if (currentRatio > maxRatio){
        return balance(tempNode);
      }
      else{
        return tempNode;
      }
    }
    else {
      SPLAVLNode<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);
  }

  if (currentRatio > maxRatio){
    return balance(current);
  }
  else{
    return current;
  }
}


/**
 * Returns true if a key is contained in a subtree of the tree, and
 * false otherwise.
 */
```

```
template <typename K, typename V>
bool SPLAVL<K,V>::containsInSubtree(SPLAVLNode<K,V>* current, K key) {
  if (current == NULL){
    return false;
  }

  else if ((current->key == root->key) && (current->key == key)){
    return true;
  }

  else if (current->left == NULL && current->right == NULL
    && current->key != key){
    return false;
  }

  else if (current->left == NULL){
    if (current->right->key == key){
      if (current->key == root->key){
        if (currentRatio > maxRatio) {
          root = balance(root);
        }
        else {
          root = splay(root, key);
        }
      }
      return true;
    }
    else if (current->key > key){
      return false;
    }
    else if (current->key < key){
      if (containsInSubtree(current->right, key)){
        if (currentRatio > maxRatio){
          current->right = balance(current->right);
        }
        else{
          current->right = splay(current->right, key);
        }
        if (current->key == root->key){
          if (currentRatio > maxRatio) {
            root = balance(root);
          }
          else {
            root = splay(root, key);
          }
        }
        return true;
      }
      else {
        return false;
      }
    }
  }

  else if (current->right == NULL){
    if (current->left->key == key){
      if (current->key == root->key){
          root = splay(root, key);
      }
```

```
    return true;
  }
  else if (current->key < key){
    return false;
  }
  else if (current->key > key){
    if (containsInSubtree(current->left, key)){
      if (currentRatio > maxRatio) {
        current->left = balance(current->left);
      }
      else {
        if(currentRatio > maxRatio) {
          current->left = balance(current->left);
        }
        else {
          current->left = splay(current->left, key);
        }
      }
      if (current->key == root->key){
        if (currentRatio > maxRatio) {
          root = balance(root);
        }
        else {
          root = splay(root, key);
        }
      }
      return true;
    }
    else{
      return false;
    }
  }
}

else{
  if (current->right->key == key){
    if (current->key == root->key){
      if (currentRatio > maxRatio) {
        root = balance(root);
      }
      else {
        root = splay(root, key);
      }
    }
    return true;
  }
  else if (current->left->key == key){
    if (current->key == root->key){
      if (currentRatio > maxRatio) {
        root = balance(root);
      }
      else {
        root = splay(root, key);
      }
    }
    return true;
  }
  else if (current->key > key){
    if (containsInSubtree(current->left, key)){
```

```
        if (currentRatio > maxRatio) {
          current->left = balance(current->left);
        }
        else {
          current->left = splay(current->left, key);
        }
        if (current->key == root->key){
          if (currentRatio > maxRatio) {
            root = balance(root);
          }
          else {
            root = splay(root, key);
          }
        }
        return true;
      }
      else{
        return false;
      }
    }
    else if (current->key < key){
      if (containsInSubtree(current->right, key)){
        if (currentRatio > maxRatio) {
          current->right = balance(current->right);
        }
        else {
          current->right = splay(current->right, key);
        }
        if (current->key == root->key){
          if (currentRatio > maxRatio) {
            root = balance(root);
          }
          else {
            root = splay(root, key);
          }
        }
        return true;
      }
      else{
        return false;
      }
    }
  }

  throw std::runtime_error("failed call to SPLAVL::containsInSubtree");
}



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


/**
 * Recursively builds a post-order iterator for a subtree of the tree.
 */
template <typename K, typename V>
void SPLAVL<K,V>::buildPostOrder(SPLAVLNode<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 SPLAVL<K,V>::buildPreOrder(SPLAVLNode<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 SPLAVL<K,V>::buildInOrder(SPLAVLNode<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 SPLAVL<K,V>::traverseAndDelete(SPLAVLNode<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 SPLAVL<K,V>::isBalancedInSubtree(SPLAVLNode<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
 * If a child is NULL, we treat the child's height as -1
 */
template<typename K, typename V>
void SPLAVL<K,V>::computeHeightFromChildren(SPLAVLNode<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 in SPLAVL
*   (1) Right rotation for a left-left imbalance,
*       and the key to splay is to the left
*
*   (2) Left rotation for a right-right imbalance,
*       and the key to splay is to the right
*
*   (3) LeftRight rotation for left-right imbalance
*
*   (4) RightLeft rotation for a right-left imbalance
*/

template<typename K, typename V>
SPLAVLNode<K,V>* SPLAVL<K,V>::rightRotate(SPLAVLNode<K,V>* current) {
  SPLAVLNode<K,V>* b = current;
  SPLAVLNode<K,V>* d = current->left;
  current = d;
  if (d->right == NULL){
    b->left = NULL;
  }
  else{
    b->left = d->right;
  }
  d->right = b;
  computeHeightFromChildren(b);
  computeHeightFromChildren(current);
  return current;
}

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

template<typename K, typename V>
SPLAVLNode<K,V>* SPLAVL<K,V>::leftRotate(SPLAVLNode<K,V>* current) {
  SPLAVLNode<K,V>* b = current;
  SPLAVLNode<K,V>* d = current->right;
  current = d;
  if (d->left == NULL){
```

```
    b->right = NULL;
  }
  else{
    b->right = d->left;
  }
  d->left = b;
  computeHeightFromChildren(b);
  computeHeightFromChildren(current);
  return current;
}

template<typename K, typename V>
SPLAVLNode<K,V>* SPLAVL<K,V>::rightLeftRotate(SPLAVLNode<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>
SPLAVLNode<K,V>* SPLAVL<K,V>::balance(SPLAVLNode<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;
}

template<typename K, typename V>
SPLAVLNode<K,V>* SPLAVL<K,V>::splay(SPLAVLNode<K,V>* current, K toSplay){
  if (current->right == NULL){
    if (current->left->key == toSplay){
        return rightRotate(current);
    }
    else{
      throw std::runtime_error("Splay function failure in SPLAVL::splay.");
    }
  }
  else if (current->left == NULL){
    if (current->right->key == toSplay){
        return leftRotate(current);
    }
    else{
      throw std::runtime_error("Splay function failure in SPLAVL::splay.");
    }
  }

  else if (current->left->key == toSplay){
    return rightRotate(current);
  }
  else if (current->right->key == toSplay){
    return leftRotate(current);
  }
```

```
  else{
    throw std::runtime_error("Splay function failure in SPLAVL::splay.");
  }
}
```