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

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

/*SPLAVLNode Implementation */

//default constructor

template <typename K, typename V>
SPLAVLNode<K,V>::SPLAVLNode() {
    height = - 1;
    left = NULL;
    right = NULL;
}

// standard constructor

template <typename K, typename V>
SPLAVLNode<K,V>::SPLAVLNode(K k, V v) {
    height = 0;
    key = k;
    value = v;
    left = NULL;
    right = NULL;
}


/*SPLAVL Implemenation */

//standard constructor
template <typename K, typename V>
SPLAVL<K,V>::SPLAVL() {
    size = 0;
    root = NULL;
    currentCount = 0;
    maxCount = 1;
    currentRatio = 0;
    maxRatio = 1;
}

template <typename K, typename V>
SPLAVL<K,V>::SPLAVL(int maxC, int maxR) {
    size = 0;
    root = NULL;
    currentCount = 0;
    maxCount = maxC;
    currentRatio = 0;
    maxRatio = maxR;
}

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

```
}

template <typename K, typename V>
int SPLAVL<K,V>::getSize() {
    return size;
}

template <typename K, typename V>
bool SPLAVL<K,V>::isEmpty() {
    return size == 0;
}

template <typename K, typename V>
K SPLAVL<K,V>::getMax() {
  if (isEmpty()) {
    throw std::runtime_error("SPLAVL::getMax called on an empty tree.");
  }
  return getMaxInSubtree(root);
}

template <typename K, typename V>
K SPLAVL<K,V>::getMin() {
  if (isEmpty()) {
    throw std::runtime_error("SPLAVL::getMin called on an empty tree.");
  }
  return getMinInSubtree(root);
}

template <typename K, typename V>
int SPLAVL<K,V>::getHeight() {
    if (root == NULL)
        return -1;
    else
        return root->height;
}

template <typename K, typename V>
void SPLAVL<K,V>::setMaxCount(int maxC) {
  if(maxC <= 0) {
    throw std::runtime_error("maxCount needs to be larger than 0");
  }
  maxCount = maxC;
}

template <typename K, typename V>
void SPLAVL<K,V>::setMaxRatio(int maxR) {
  maxRatio = maxR;
}

template <typename K, typename V>
void SPLAVL<K,V>::insert(K key, V value) {
  currentCount++;
  if (currentCount >= maxCount){
    int height = getHeight();
    float size = getSize();
    float denom = log (size);
    currentRatio = height/denom;
    currentCount = 0;
  }
```

```
  root = insertInSubtree(root, key, value);
  if (currentRatio > maxRatio){
    currentRatio = 0;
  }
}

template <typename K, typename V>
void SPLAVL<K,V>::update(K key, V value) {
    currentCount++;
    if (currentCount >= maxCount){
      int height = getHeight();
      float size = getSize();
      float denom = log (size);
      currentRatio = height/denom;
      currentCount = 0;
    }
    //updateInSubtree(root, key, value);
    if (contains(key)){
      root->value = value;
      if (currentRatio > maxRatio){
        currentRatio = 0;
      }
    }
    else{
      throw std::runtime_error("SPLAVL:update called on nonexistent node");
    }
}


template <typename K, typename V>
bool SPLAVL<K,V>::contains(K key) {
  currentCount++;
  if (currentCount >= maxCount){
    int height = getHeight();
    float size = getSize();
    float denom = log (size);
    currentRatio = height/denom;
    currentCount = 0;
  }
  return containsInSubtree(root, key);
  if (currentRatio > maxRatio){
    currentRatio = 0;
  }
}

template <typename K, typename V>
void SPLAVL<K,V>::remove(K key) {
  root = removeFromSubtree(root, key);
}

template <typename K, typename V>
V SPLAVL<K,V>::find(K key) {
  currentCount++;
  if (currentCount >= maxCount){
    int height = getHeight();
    float size = getSize();
    float denom = log (size);
    currentRatio = height/denom;
    currentCount = 0;
```

```
  }
  if (contains(key)){
    if (currentRatio > maxRatio){
      currentRatio = 0;
    }
    return root->value;
  }
  else{
    throw std::runtime_error("SPLAVL:find called on nonexistent node");
  }
}


template <typename K, typename V>
Queue< Pair<K,V> >* SPLAVL<K,V>::getPreOrder() {
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();
  buildPreOrder(root, it);
  return it;
}


template <typename K, typename V>
Queue< Pair<K,V> >* SPLAVL<K,V>::getInOrder() {
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();
  buildInOrder(root, it);
  return it;
}

template <typename K, typename V>
Queue< Pair<K,V> >* SPLAVL<K,V>::getPostOrder() {
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();
  buildPostOrder(root, it);
  return it;
}

template <typename K, typename V>
Queue< Pair<K,V> >* SPLAVL<K,V>::getLevelOrder() {
  ArrayQueue< SPLAVLNode<K,V>* > levelQ;
  Queue< Pair<K,V> >* it = new ArrayQueue< Pair<K,V> >();

  levelQ.enqueue(root);
  while (!levelQ.isEmpty()) {
    SPLAVLNode<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;
}

template <typename K, typename V>
K SPLAVL<K,V>::getRootKey() {
  return root->key;
}

/* isBalanced- returns true if the tree is balanced
```

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