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

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

/*SplayTreeNode implmentation */

//default constructor

template <typename K, typename V>
SplayTreeNode<K,V>::SplayTreeNode() {
    left = NULL;
    right = NULL;
}

// standard constructor

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


/*SplayTree Implemenation */


//standard constructor
template <typename K, typename V>
SplayTree<K,V>::SplayTree() {
    size = 0;
    root = NULL;
}

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

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

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

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

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

template <typename K, typename V>
int SplayTree<K,V>::getHeight() {
  return getHeightOfSubtree(root);
}

template <typename K, typename V>
void SplayTree<K,V>::insert(K key, V value) {
    bool inserted = false;
    bool skip = false;
    root = insertInSubtree(root, key, value, &inserted, &skip);
}

template <typename K, typename V>
void SplayTree<K,V>::update(K key, V value) {
    //updateInSubtree(root, key, value);
    if (contains(key)){
      root->value = value;
    }
    else{
      throw std::runtime_error("SplayTree:update called on nonexistent node");
    }
}


template <typename K, typename V>
bool SplayTree<K,V>::contains(K key) {
  bool skip = false;
  return containsInSubtree(root, key, &skip);
}

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

template <typename K, typename V>
V SplayTree<K,V>::find(K key) {
  if (contains(key)){
    return root->value;
  }
  else{
    throw std::runtime_error("SplayTree:find called on nonexistent node");
  }
}
```

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

  levelQ.enqueue(root);
  while (!levelQ.isEmpty()) {
    SplayTreeNode<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 SplayTree<K,V>::getRootKey() {
  return root->key;
}
```