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


/* insertInSubtree
 * creates a new node or finds which subtree to place the new node
 * calls splay as appropriate to boost the created node up to root
 */

template <typename K, typename V>
SplayTreeNode<K,V>*
SplayTree<K,V>::insertInSubtree(SplayTreeNode<K,V>* current, K key, V value, bool*
inserted, bool* skip) {

  if (current == NULL){
    size++;
    *inserted = true;
    *skip = true;
    return new SplayTreeNode<K, V>(key, value);
  }

  else if (key == current->key && !(*inserted)){
    throw std::runtime_error("SplayTree::insertInSubtree" \
      "called on key already in tree.");
  }
  else if (key < current->key){
    current->left = insertInSubtree(current->left, key, value, inserted, skip);
  }
  else if (key > current->key){
    current->right = insertInSubtree(current->right, key, value, inserted, skip);
  }
  if (current->key == root->key && *inserted) {
    return splay(current, key);
  }
  else if (*skip && (current != root) && *inserted) {
    *skip = false;
    return current;
  }
  else {
    *skip = true;
    return splay(current, key);
  }
}


/**
 * 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>
SplayTreeNode<K,V>*
SplayTree<K,V>::removeFromSubtree(SplayTreeNode<K,V>* current, K key) {
```

```
  if (current == NULL) {
    throw std::runtime_error("SplayTree::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) {
      SplayTreeNode<K,V>* tempNode = current->right;
      delete current;
      size--;
      return tempNode;
    }
    else if (current->right == NULL) {
      SplayTreeNode<K,V>* tempNode = current->left;
      delete current;
      size--;
      return tempNode;
    }
    else {
      SplayTreeNode<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 current;
}


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

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

```
      }
    }
  }

  else{
    if (current->right->key == key){
      if (current->key == root->key){
          root = splay(root, key);
      }
      return true;
    }
    else if (current->left->key == key){
      if (current->key == root->key){
          root = splay(root, key);
      }
      return true;
    }
    else if (current->key > key){
      if (containsInSubtree(current->left, key, skip)){
        if (*skip == true) {
          *skip = false;
        }
        else {
          current->left = splay(current->left, key);
        }
        if (current->key == root->key){
          root = splay(root, key);
        }
        return true;
      }
      else{
        return false;
      }
    }
    else if (current->key < key){
      if (containsInSubtree(current->right, key, skip)){
        if (*skip == true) {
          *skip = false;
        }
        else {
          current->right = splay(current->right, key);
        }
        if (current->key == root->key){
          root = splay(root, key);
        }
        return true;
      }
      else{
        return false;
      }
    }
  }

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


/**
 * Returns the largest key in a subtree of the tree.
```

```
 */
template <typename K, typename V>
K SplayTree<K,V>::getMaxInSubtree(SplayTreeNode<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 SplayTree<K,V>::getMinInSubtree(SplayTreeNode<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 SplayTree<K,V>::buildPostOrder(SplayTreeNode<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 SplayTree<K,V>::buildPreOrder(SplayTreeNode<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 SplayTree<K,V>::buildInOrder(SplayTreeNode<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 SplayTree<K,V>::traverseAndDelete(SplayTreeNode<K,V>* current) {
  if (current == NULL) {
    return;  //nothing to delete
  }
  traverseAndDelete(current->left);
  traverseAndDelete(current->right);
  delete current;
}


/* The four rotations needed to fix each of the six possible rotations
*   in an SplayTree
*   (1) Right rotation for splaying from left
*   (2) Left rotation for splaying from right
*   (3) LeftRight rotation for splaying from left-right
*   (4) RightLeft rotation for splaying from right-left
*   (5) RightRight rotation for splaying from right-right
*   (6) LeftLeft rotation for splaying from left-left
*/

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


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

```
  d->left = b;
  return current;
}

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

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

template<typename K, typename V>
SplayTreeNode<K,V>* SplayTree<K,V>:: rightRightRotate(SplayTreeNode<K,V>* current)
{
    current = rightRotate(current);
    return rightRotate(current);
}

template<typename K, typename V>
SplayTreeNode<K,V>* SplayTree<K,V>:: leftLeftRotate(SplayTreeNode<K,V>* current) {
    current = leftRotate(current);
    return leftRotate(current);
}


/* This function takes in a node and a key to splay
 * It assume that the key is at most a grandchild of the node
 * It performs the correct rotation based on the key's location
 */
template<typename K, typename V>
SplayTreeNode<K,V>* SplayTree<K,V>::splay(SplayTreeNode<K,V>* parent, K key){
  if (parent == NULL){
    throw std::runtime_error("1 Splay function failure in SplayTree::splay.");
  }
  else {
    if (parent->right == NULL){
      if (parent->left == NULL) {
        throw std::runtime_error("2 Splay function failure in SplayTree::splay.");
      }
      else if (parent->left->key == key){
          return rightRotate(parent);
      }
      else {
        if (parent->left->left == NULL && parent->left->right == NULL) {
          throw std::runtime_error("3 Splay function failure in
SplayTree::splay.");
        }
        else if (parent->left->right == NULL) {
          if (parent->left->left->key == key) {
            return rightRightRotate(parent);
          }
          else {
            throw std::runtime_error("4 Splay function failure in
SplayTree::splay.");
```

```
          }
        }
        else if (parent->left->left == NULL) {
          if (parent->left->right->key == key) {
            return leftRightRotate(parent);
          }
          else {
            throw std::runtime_error("5 Splay function failure in
SplayTree::splay.");
          }
        }
        else {
          if (parent->left->right->key == key) {
            return leftRightRotate(parent);
          }
          else if (parent->left->left->key == key) {
            return rightRightRotate(parent);
          }
          else {
            throw std::runtime_error("6 Splay function failure in
SplayTree::splay.");
          }
        }
      }
    }


    else if (parent->left == NULL) {
      if (parent->right == NULL) {
        throw std::runtime_error("2 Splay function failure in SplayTree::splay.");
      }
      else if (parent->right->key == key){
          return leftRotate(parent);
      }
      else{
        if (parent->right->left == NULL && parent->right->right == NULL) {
          throw std::runtime_error("7 Splay function failure in
SplayTree::splay.");
        }
        else if (parent->right->left == NULL) {
          if (parent->right->right->key == key) {
            return leftLeftRotate(parent);
          }
          else {
            throw std::runtime_error("8 Splay function failure in
SplayTree::splay.");
          }
        }
        else if (parent->right->right == NULL) {
          if (parent->right->left->key == key) {
            return rightLeftRotate(parent);
          }
          else {
            throw std::runtime_error("9 Splay function failure in
SplayTree::splay.");
          }
        }
        else {
          if (parent->right->left->key == key) {
```

```
            return rightLeftRotate(parent);
          }
          else if (parent->right->right->key == key) {
            return leftLeftRotate(parent);
          }
          else {
            throw std::runtime_error("10 Splay function failure in
SplayTree::splay.");
          }
        }
      }
    }

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

        else if (parent->key > key){
         if (parent->left->key > key){
          if (parent->left->left == NULL){
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
          else if (parent->left->left->key == key){
            return rightRightRotate(parent);
          }
          else{
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
        }
        else{
          if (parent->left->right == NULL){
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
          else if (parent->left->right->key == key){
            return leftRightRotate(parent);
          }
          else{
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
        }
      }

      else if (parent->key < key){
         if (parent->right->key > key){
          if (parent->right->left == NULL){
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
          else if (parent->right->left->key == key){
            return rightLeftRotate(parent);
          }
```

```
          else{
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
        }
        else{
          if (parent->right->right == NULL){
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
          else if (parent->right->right->key == key){
            return leftLeftRotate(parent);
          }
          else{
            throw std::runtime_error("Splay function failure in
SplayTree::splay.");
          }
        }
      }

      throw std::runtime_error("12 Splay function failure in SplayTree::splay.");
    }
  }
}

/**
 * Returns the height of a subtree of the tree, or -1 if the subtree
 * is empty.
 */
template <typename K, typename V>
int SplayTree<K,V>::getHeightOfSubtree(SplayTreeNode<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;
}
```