```
/*testSplay.cpp
 *
 *Dylan Jeffers
 *Tahmid Rahman
 *
 *Test script for SplayTrees
 *are at the bottom of the script
 *
 *The nature and structure of this
 *code was inspired by our test code
 *for AVL trees from CS31, a class
 *taken Fall, 2014 at Swarthmore
 */


#include <stdlib.h>  // Used for pseudo-random number generation.
#include <assert.h>  // Used for testing below.

#include <iostream>
#include "pair.h"
#include "BST.h"
#include "library/circularArrayList.h"
#include "library/queue.h"
#include "SplayTree.h"

using namespace std;

void insertTest();
void updateTest();
void removeTest();
void findTest();
void testMaxMin();
void testgetHeight();

int main() {
  insertTest();
  findTest();
  updateTest();
  testMaxMin();
  removeTest();
  testgetHeight();

  cout << "Passed SplayTree tests!" << endl;

  return 0;
}

/* insertTest - accomplishes the following
 *    *tests getSize
 *
 *    *ensures a new tree is indeed empty
 *
 *    *ensures each insert increases the size by 1
 *
 *    *tests that each inserted element is in the tree
 *    *tests inserting lots of elements in increasing order,
 *     then lots of elements in decreasing order
 *
 *    *tests that each inserted element is splayed to the right spot
```

```
 *     by checking all four traversal algorithms on five small subtrees
 *
 */
void insertTest() {
  SplayTree<int,int> BST;
  //Queue< Pair<int,int> >* queue0;

  SplayTree<int, char> SBST1, SBST2, SBST3, SBST4, SBST5, SBST6;
  Queue< Pair<int,char> >* queue1;
  Queue< Pair<int,char> >* queue2;
  Queue< Pair<int,char> >* queue3;
  Queue< Pair<int,char> >* queue4;

  assert(BST.getSize() == 0);  // Checks that initial size is correct.  assert
                               // causes the program to immediately crash if
                               // the condition is false.

  for (int i = 0; i < 100; ++i) {
    BST.insert(2*i + 1, i);
    assert(BST.getSize() == i+1);
    assert(BST.getRootKey() == 2*i+1);
  }

  for (int i = 0; i < 100; ++i) {  // Checks that keys are in the tree.
    assert(BST.contains(2*i + 1));
    assert(BST.getRootKey() == 2*i+1);
  }

  for (int i = 0; i < 100; ++i) {
    BST.insert(-2*i - 1, i);
    assert(BST.getSize() == i+1 + 100);
    cout.flush();
  }

  for (int i = 0; i < 100; ++i) {  // Checks that keys are in the tree.
    assert(BST.contains(-2*i - 1));
    assert(BST.getRootKey() == -2*i-1);
  }

  for (int i = 0; i < 100; ++i) {  //Error returned if key already exists.
    try{
      BST.insert(2*i + 1, i);
      assert(false);
    } catch(runtime_error& exc){}
  }

  //================================================

  /* The following tests each tree traversal algorithm on
   * five elementary subtrees.
   */
  SBST1.insert(1, 'A');
  SBST1.insert(2, 'B');
  SBST1.insert(3, 'C');

  queue1 = SBST1.getPostOrder();
  queue2 = SBST1.getPreOrder();
  queue3 = SBST1.getInOrder();
  queue4 = SBST1.getLevelOrder();
```

```
assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'B');
assert(queue1->dequeue().second == 'C');

assert(queue2->dequeue().second == 'C');
assert(queue2->dequeue().second == 'B');
assert(queue2->dequeue().second == 'A');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'B');
assert(queue3->dequeue().second == 'C');

assert(queue4->dequeue().second == 'C');
assert(queue4->dequeue().second == 'B');
assert(queue4->dequeue().second == 'A');

delete queue1;
delete queue2;
delete queue3;
delete queue4;

//================================================

SBST2.insert(3, 'C');
SBST2.insert(2, 'B');
SBST2.insert(1, 'A');

queue1 = SBST2.getPostOrder();
queue2 = SBST2.getPreOrder();
queue3 = SBST2.getInOrder();
queue4 = SBST2.getLevelOrder();

assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'B');
assert(queue1->dequeue().second == 'A');

assert(queue2->dequeue().second == 'A');
assert(queue2->dequeue().second == 'B');
assert(queue2->dequeue().second == 'C');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'B');
assert(queue3->dequeue().second == 'C');

assert(queue4->dequeue().second == 'A');
assert(queue4->dequeue().second == 'B');
assert(queue4->dequeue().second == 'C');

delete queue1;
delete queue2;
delete queue3;
delete queue4;


//================================================

SBST3.insert(2, 'B');
SBST3.insert(1, 'A');
```

```
SBST3.insert(3, 'C');

queue1 = SBST3.getPostOrder();
queue2 = SBST3.getPreOrder();
queue3 = SBST3.getInOrder();
queue4 = SBST3.getLevelOrder();

assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'B');
assert(queue1->dequeue().second == 'C');

assert(queue2->dequeue().second == 'C');
assert(queue2->dequeue().second == 'B');
assert(queue2->dequeue().second == 'A');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'B');
assert(queue3->dequeue().second == 'C');

assert(queue4->dequeue().second == 'C');
assert(queue4->dequeue().second == 'B');
assert(queue4->dequeue().second == 'A');

delete queue1;
delete queue2;
delete queue3;
delete queue4;



//================================================

SBST4.insert(3, 'C');
SBST4.insert(1, 'A');
SBST4.insert(2, 'B');

queue1 = SBST4.getPostOrder();
queue2 = SBST4.getPreOrder();
queue3 = SBST4.getInOrder();
queue4 = SBST4.getLevelOrder();

assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'B');

assert(queue2->dequeue().second == 'B');
assert(queue2->dequeue().second == 'A');
assert(queue2->dequeue().second == 'C');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'B');
assert(queue3->dequeue().second == 'C');

assert(queue4->dequeue().second == 'B');
assert(queue4->dequeue().second == 'A');
assert(queue4->dequeue().second == 'C');

delete queue1;
delete queue2;
```

```
delete queue3;
delete queue4;


//================================================

SBST5.insert(1, 'A');
SBST5.insert(3, 'C');
SBST5.insert(2, 'B');

queue1 = SBST5.getPostOrder();
queue2 = SBST5.getPreOrder();
queue3 = SBST5.getInOrder();
queue4 = SBST5.getLevelOrder();

assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'B');

assert(queue2->dequeue().second == 'B');
assert(queue2->dequeue().second == 'A');
assert(queue2->dequeue().second == 'C');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'B');
assert(queue3->dequeue().second == 'C');

assert(queue4->dequeue().second == 'B');
assert(queue4->dequeue().second == 'A');
assert(queue4->dequeue().second == 'C');

delete queue1;
delete queue2;
delete queue3;
delete queue4;

//================================================

SBST6.insert(2, 'B');
SBST6.insert(3, 'C');
SBST6.insert(1, 'A');

queue1 = SBST6.getPostOrder();
queue2 = SBST6.getPreOrder();
queue3 = SBST6.getInOrder();
queue4 = SBST6.getLevelOrder();

assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'B');
assert(queue1->dequeue().second == 'A');

assert(queue2->dequeue().second == 'A');
assert(queue2->dequeue().second == 'B');
assert(queue2->dequeue().second == 'C');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'B');
assert(queue3->dequeue().second == 'C');
```

```
  assert(queue4->dequeue().second == 'A');
  assert(queue4->dequeue().second == 'B');
  assert(queue4->dequeue().second == 'C');

  delete queue1;
  delete queue2;
  delete queue3;
  delete queue4;

}


/* findTest - accomplishes the following
 *
 *    *tests that each removed element
 *     is not in the tree
 *
 *    *tests that right value is returned when find is called
 *
 *    *tests that find lifts the found element to the root
 */
void findTest() {
  SplayTree<int,int> BST;
  SplayTree<int, char> BST2;

  assert(BST.getSize() == 0);  // Checks that initial size is correct.

  for (int i = 0; i < 100; ++i) {
    BST.insert(2*i + 1, i);
    assert(BST.getSize() == i+1);
  }
  for (int i = 0; i < 100; ++i) {  // Checks that keys are in the tree.
    assert(BST.find(2*i + 1) == i);
    assert(BST.getRootKey() == 2*i+1);
  }
  for (int i = 0; i < 100; ++i) { // Checks that key returns proper update.
    BST.update(2*i + 1, 2*i);
    assert(BST.find(2*i + 1) == 2*i);
    assert(BST.getRootKey() == 2*i+1);
  }

  try{ // Testing edge cases
    BST.find(0);
    assert(BST.getRootKey() == 199);
    assert(false);
  } catch(runtime_error& exc){}

  try{
    BST.find(2*99 + 2);
    assert(false);
  } catch(runtime_error& exc){}

  BST2.insert(4, 'D');
  assert(BST2.getRootKey() == 4);
  BST2.insert(3, 'C');
  assert(BST2.getRootKey() == 3);
  BST2.insert(5, 'E');
  assert(BST2.getRootKey() == 5);
  BST2.insert(1, 'A');
```

```
  assert(BST2.getRootKey() == 1);
  BST2.insert(2, 'B');
  assert(BST2.getRootKey() == 2);

  assert(BST2.find(4) == 'D');
  assert(BST2.getRootKey() == 4);
  BST2.remove(4);

  try{
    BST2.find(4);
    assert(false);
  } catch(runtime_error& exc){}

  assert(BST2.find(2) == 'B');
  assert(BST2.getRootKey() ==  2);
  BST2.remove(2);

  try{
    BST2.find(2);
    assert(false);
  } catch(runtime_error& exc){}

  assert(BST2.find(1) == 'A');
  assert(BST2.getRootKey() == 1);
  BST2.remove(1);

  try{
    BST2.find(1);
    assert(false);
  } catch(runtime_error& exc){}

  assert(BST2.find(3) == 'C');
  assert(BST2.getRootKey() == 3);
  BST2.remove(3);

  try{
    BST2.find(3);
    assert(false);
  } catch(runtime_error& exc){}

  assert(BST2.find(5) == 'E');
  assert(BST2.getRootKey() == 5);
  BST2.remove(5);

  try{
    BST2.find(5);
    assert(false);
  } catch(runtime_error& exc){}
}


/* updateTest - accomplishes the following
 *    *tests that update returns error if key is
 *     not in tree
 *
 *    *tests that each updated element is in the right spot
 *     by checking on five small subtrees (i.e. with 3 nodes)
 *
 *    *ensures that updated element is lifted to root
```

```
 */
void updateTest(){
  SplayTree<int, char> SBST1, SBST2, SBST3, SBST4, SBST5;
  Queue< Pair<int,char> >* queue;
  SplayTree<int, int> BST;

  assert(BST.getSize() == 0);  // Checks that initial size is correct.

 try{ // errors returned when key does not exist in subtree
      BST.update(5, 10);
      assert(false);
  } catch(runtime_error& exc){}

  for (int i = 0; i < 100; ++i) { //inserts and updates 100 elements
    BST.insert(2*i + 1, i);
    assert(BST.contains(2*i + 1));
    assert(BST.getRootKey() == 2*i + 1);
    BST.update(2*i + 1, 2*i);
    assert(BST.getSize() == i+1);
    assert(BST.getRootKey() == 2*i + 1);
  }

 try{
      BST.update(2*99 + 2, 100);
      assert(false);
  } catch(runtime_error& exc){}

  for (int i = 0; i < 100; ++i) {  // Checks that keys haven't been changed
    assert(BST.contains(2*i + 1));
    assert(BST.getRootKey() == 2*i + 1);
  }

  SBST1.insert(1, 'A');
  SBST1.insert(2, 'B');
  SBST1.insert(3, 'C');
  SBST1.update(2, 'D');

  queue = SBST1.getPostOrder();

  assert(SBST1.getRootKey() == 2);

  assert(queue->dequeue().second == 'A');
  assert(queue->dequeue().second == 'C');
  assert(queue->dequeue().second == 'D');

  delete queue;

  //================================================

  SBST2.insert(3, 'C');
  SBST2.insert(2, 'B');
  SBST2.insert(1, 'A');
  SBST2.update(1, 'E');
  SBST2.update(2, 'B');

  queue = SBST2.getPostOrder();

  assert(SBST2.getRootKey() == 2);
```

```
assert(queue->dequeue().second == 'E');
assert(queue->dequeue().second == 'C');
assert(queue->dequeue().second == 'B');

delete queue;

//================================================

SBST3.insert(2, 'B');
SBST3.insert(1, 'A');
SBST3.insert(3, 'C');
SBST3.update(3, 'F');

queue = SBST3.getPostOrder();

assert(SBST3.getRootKey() == 3);

assert(queue->dequeue().second == 'A');
assert(queue->dequeue().second == 'B');
assert(queue->dequeue().second == 'F');

delete queue;

//================================================

SBST4.insert(3, 'C');
SBST4.insert(1, 'A');
SBST4.insert(2, 'B');
SBST4.update(3, 'D');
SBST4.update(1, 'E');
SBST4.update(2, 'F');


queue = SBST4.getPostOrder();

assert(SBST4.getRootKey() == 2);

assert(queue->dequeue().second == 'E');
assert(queue->dequeue().second == 'D');
assert(queue->dequeue().second == 'F');

delete queue;

//================================================

SBST5.insert(1, 'A');
SBST5.insert(3, 'C');
SBST5.insert(2, 'B');
SBST5.update(2, 'G');
SBST5.update(2, 'E');
SBST5.update(3, 'F');
SBST5.update(2, 'M');

queue = SBST5.getPostOrder();

assert(SBST5.getRootKey() == 2);

assert(queue->dequeue().second == 'A');
assert(queue->dequeue().second == 'F');
```

```
  assert(queue->dequeue().second == 'M');

  delete queue;
}


/*    removeTest - accomplishes the following
 *
 *    *ensures we can't delete in empty tree
 *
 *    *ensures each remove decreases the size by 1
 *
 *    *for tests that check each removed element
 *     is not in the tree, look at findTest
 *
 *    *tests that each removed element results in tree
 *     with elements in the right spot
 *     by checking all four traversal algorithms on all remove
 *     situations
 */
void removeTest() {
    SplayTree<int, char> BST;

    Queue< Pair<int,char> >* queue1;
    Queue< Pair<int,char> >* queue2;
    Queue< Pair<int,char> >* queue3;
    Queue< Pair<int,char> >* queue4;

    try{ // testing removing on an empty tree
      BST.remove(1);
      assert(false);
    } catch(runtime_error& exc){}

    BST.insert(2, 'B');
    BST.insert(1, 'A');
    BST.insert(5, 'E');
    BST.insert(3, 'C');
    BST.insert(4, 'D');

    assert(BST.getSize() == 5);
    try { // Testing remove on non-existant keys
      BST.remove(6);
      assert(false);
    } catch(runtime_error& exc){}

    try {
      BST.remove(0);
      assert(false);
    } catch(runtime_error& exc){}

    //=============================================
    /* The following is a comprehensive test of the SplayTree::remove
     * function.
     */
    queue1 = BST.getPostOrder();
    queue2 = BST.getPreOrder();
    queue3 = BST.getInOrder();
    queue4 = BST.getLevelOrder();
```

```
assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'B');
assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'E');
assert(queue1->dequeue().second == 'D');

assert(queue2->dequeue().second == 'D');
assert(queue2->dequeue().second == 'C');
assert(queue2->dequeue().second == 'B');
assert(queue2->dequeue().second == 'A');
assert(queue2->dequeue().second == 'E');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'B');
assert(queue3->dequeue().second == 'C');
assert(queue3->dequeue().second == 'D');
assert(queue3->dequeue().second == 'E');

assert(queue4->dequeue().second == 'D');
assert(queue4->dequeue().second == 'C');
assert(queue4->dequeue().second == 'E');
assert(queue4->dequeue().second == 'B');
assert(queue4->dequeue().second == 'A');

delete queue1;
delete queue2;
delete queue3;
delete queue4;

//================================================

BST.remove(2);
assert(BST.getSize() == 4);
queue1 = BST.getPostOrder();
queue2 = BST.getPreOrder();
queue3 = BST.getInOrder();
queue4 = BST.getLevelOrder();

assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'E');
assert(queue1->dequeue().second == 'D');

assert(queue2->dequeue().second == 'D');
assert(queue2->dequeue().second == 'C');
assert(queue2->dequeue().second == 'A');
assert(queue2->dequeue().second == 'E');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'C');
assert(queue3->dequeue().second == 'D');
assert(queue3->dequeue().second == 'E');

assert(queue4->dequeue().second == 'D');
assert(queue4->dequeue().second == 'C');
assert(queue4->dequeue().second == 'E');
assert(queue4->dequeue().second == 'A');

delete queue1;
```

```
delete queue2;
delete queue3;
delete queue4;

//================================================

BST.remove(5);

assert(BST.getSize() == 3);

queue1 = BST.getPostOrder();
queue2 = BST.getPreOrder();
queue3 = BST.getInOrder();
queue4 = BST.getLevelOrder();

assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'D');

assert(queue2->dequeue().second == 'D');
assert(queue2->dequeue().second == 'C');
assert(queue2->dequeue().second == 'A');

assert(queue3->dequeue().second == 'A');
assert(queue3->dequeue().second == 'C');
assert(queue3->dequeue().second == 'D');

assert(queue4->dequeue().second == 'D');
assert(queue4->dequeue().second == 'C');
assert(queue4->dequeue().second == 'A');

delete queue1;
delete queue2;
delete queue3;
delete queue4;

//================================================

BST.remove(1);

assert(BST.getSize() == 2);

queue1 = BST.getPostOrder();
queue2 = BST.getPreOrder();
queue3 = BST.getInOrder();
queue4 = BST.getLevelOrder();

assert(queue1->dequeue().second == 'C');
assert(queue1->dequeue().second == 'D');

assert(queue2->dequeue().second == 'D');
assert(queue2->dequeue().second == 'C');

assert(queue3->dequeue().second == 'C');
assert(queue3->dequeue().second == 'D');

assert(queue4->dequeue().second == 'D');
assert(queue4->dequeue().second == 'C');
```

```
    delete queue1;
    delete queue2;
    delete queue3;
    delete queue4;

    //================================================

    BST.remove(4);

    assert(BST.getSize() == 1);

    queue1 = BST.getPostOrder();
    queue2 = BST.getPreOrder();
    queue3 = BST.getInOrder();
    queue4 = BST.getLevelOrder();

    assert(queue1->dequeue().second == 'C');

    assert(queue2->dequeue().second == 'C');

    assert(queue3->dequeue().second == 'C');

    assert(queue4->dequeue().second == 'C');

    delete queue1;
    delete queue2;
    delete queue3;
    delete queue4;
    //================================================

    BST.remove(3);

    assert(BST.getSize() == 0);

    queue1 = BST.getPostOrder();
    queue2 = BST.getPreOrder();
    queue3 = BST.getInOrder();
    queue4 = BST.getLevelOrder();

    assert(queue1->getSize() == 0);
    assert(queue2->getSize() == 0);
    assert(queue3->getSize() == 0);
    assert(queue4->getSize() == 0);

    delete queue1;
    delete queue2;
    delete queue3;
    delete queue4;
}

/* testgetHeight - accomplishes the following
 *
 *    *tests that empty tree yields height of -1
 *
 *    *tests that tree with one element returns height of 0
 *
 *    *tests height by creating tree and incrementally
 *     removing nodes
 */
```

```
void testgetHeight(){
  SplayTree<int, char> BST;

  BST.insert(4, 'D');
  BST.insert(2, 'B');
  BST.insert(1, 'A');
  BST.insert(3, 'C');
  BST.insert(6, 'F');
  BST.insert(5, 'E');
  BST.insert(7, 'G');

  for (int i = 1; i <= 7; i++) {
    assert(BST.getHeight() == 7-i);
    BST.remove(i);
  }
  assert(BST.getHeight() == -1);
}

/* testMaxMin - accomplishes the following
 *
 *    *tests that calling getMax or getMin on empty tree
 *     results in error thrown
 *
 *    *tests that tree with one element returns same max
 *     and min key
 *
 *    *tests max and min by creating tree and incrementally
 *     removing nodes
 */
void testMaxMin(){
  SplayTree<int, char> BST;

  try{
    BST.getMax();
    assert(false);
  } catch(runtime_error& exc){}

  try{
    BST.getMin();
    assert(false);
  } catch(runtime_error& exc){}

  BST.insert(6, 'A');
  assert(BST.getMax() == 6);
  assert(BST.getMin() == 6);
  assert(BST.getMax() == BST.getMin());

  BST.insert(1, 'B');
  BST.insert(5, 'C');
  BST.insert(2, 'D');
  BST.insert(4, 'E');
  BST.insert(3, 'F');
  BST.insert(11, 'G');
  BST.insert(7, 'H');
  BST.insert(10, 'I');
  BST.insert(8, 'J');
  BST.insert(9, 'K');

  assert(BST.getMax() == 11);
```

```
  assert(BST.getMin() == 1);

  BST.remove(11);
  BST.remove(1);

  assert(BST.getMax() == 10);
  assert(BST.getMin() == 2);

  BST.remove(10);
  BST.remove(2);

  assert(BST.getMax() == 9);
  assert(BST.getMin() == 3);

  BST.remove(9);
  BST.remove(3);

  assert(BST.getMax() == 8);
  assert(BST.getMin() == 4);

  BST.remove(8);
  BST.remove(4);

  assert(BST.getMax() == 7);
  assert(BST.getMin() == 5);

  BST.remove(7);
  BST.remove(5);

  assert(BST.getMax() == 6);
  assert(BST.getMin() == 6);
  assert(BST.getMax() == BST.getMin());
}
```