```
/*testSplavl.cpp
 *Dylan Jeffers
 *Tahmid Rahman
 *
 *Test script for SPLAVL Trees
 *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 "AVLTree.h"
#include "SPLAVL.h"
#include "SPLAVL.h"

using namespace std;

void insertTest();
void updateTest();
void findTest();
void testMaxMin();
void testgetHeight();
void SPLAYRemoveTest();
void AVLRemoveTest();
void AVLInsertTest();
void SPLAVLinsertTest();


int main() {
  insertTest();
  SPLAVLinsertTest();
  findTest();
  updateTest();
  SPLAYRemoveTest();
  AVLRemoveTest();
  AVLInsertTest();
  cout << "Passed SPLAVL 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() {
  SPLAVL<int,int> BST;
  BST.setMaxCount(1000);
  //Queue< Pair<int,int> >* queue0;

  SPLAVL<int, char> SBST1, SBST2, SBST3, SBST4, SBST5;
  SBST1.setMaxCount(1000);
  SBST2.setMaxCount(1000);
  SBST3.setMaxCount(1000);
  SBST4.setMaxCount(1000);
  SBST5.setMaxCount(1000);

  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);
  }
  for (int i = 0; i < 100; ++i) {  // Checks that keys are in the tree.
    assert(BST.contains(-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 == 'B');
assert(queue1->dequeue().second == 'A');
assert(queue1->dequeue().second == 'C');

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

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

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

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;
}


/* 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 the element found is splayed to root
 */
void findTest() {
  SPLAVL<int,int> BST;
  SPLAVL<int, char> BST2;
```

```
BST.setMaxCount(1000);
BST2.setMaxCount(1000);


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 elementary subtrees (i.e. with 3 nodes)
 *
 *    *checks that updated element is splayed to root
 */
void updateTest(){
  SPLAVL<int, char> SBST1, SBST2, SBST3, SBST4, SBST5;
  Queue< Pair<int,char> >* queue;
  SPLAVL<int, int> BST;

  SBST1.setMaxCount(1000);
  SBST2.setMaxCount(1000);
  SBST3.setMaxCount(1000);
  SBST4.setMaxCount(1000);
  SBST5.setMaxCount(1000);
  BST.setMaxCount(1000);


  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));
   BST.update(2*i + 1, 2*i);
   assert(BST.getSize() == 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));
 }

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

 queue = SBST1.getPostOrder();

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

  delete queue;
}


/* SPLAYremoveTest - accomplishes the following
* *Note: This tests splay functionality of SPLAVL
*
* *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 various remove
* situations
*/

void SPLAYRemoveTest() {
  SPLAVL<int, char> BST;
  BST.setMaxCount(1000);
  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 SPLAVL::remove
 * function. We have designed the tree to test all possible
 * removal algorithms (right/left leaves, parent nodes with one
 * right/left child node, and parent node with two subchildren).
 * Tested with all four search algorithms.
 * The original tree looks like:
 */
queue1 = BST.getPostOrder();
queue2 = BST.getPreOrder();
queue3 = BST.getInOrder();
queue4 = BST.getLevelOrder();

assert(queue1->dequeue().second == 'B');
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 == 'B');
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 == 'A');
assert(queue4->dequeue().second == 'B');

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;
}

/* AVLremoveTest - accomplishes the following
 *
 * Note: this tests the AVL functionality of SPLAVL
 *
 * *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 various remove
 * situations
 */

void AVLRemoveTest(){
  SPLAVL<int,int> AVL;
  AVL.setMaxRatio(-1);

  //Our insertTest method makes sure that the balancing calls
  //do what they should. The only case to take care of is
  //when we delete a leaf node (and we call balance on NULL.

  //This adds a bunch of elements and then
```

```
  //deletes a bunch of elements, and makes sure things still stay
  //balanced.

  assert(AVL.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) {
    AVL.insert(2*i + 1, i);
    assert(AVL.isBalanced());
    assert(AVL.getSize() == i+1);
  }
  for (int i = 0; i < 100; ++i) {  // Checks that keys are in the tree.
    assert(AVL.contains(2*i + 1));
  }

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

  for (int i = 0; i < 100; ++i) {
    AVL.remove(-2*i - 1);
    assert(AVL.isBalanced());
    assert(AVL.getSize() == 199 - i);
  }

  for (int i = 0; i < 100; ++i) {
    AVL.remove(2*(99-i) + 1);
    assert(AVL.isBalanced());
    assert(AVL.getSize() == 99 - i);
  }

  assert(AVL.getSize() == 0);
}

/* AVLInsertTest - accomplishes the following
 *
 *    *Note: This tests AVL functionality of SPLAVL
 *
 *    *tests tree is balanced after series of insertions and removals
 *
 *    *tests that each kind of imbalance is properly adjusted for
 */
void AVLInsertTest(){
  SPLAVL<int,int> AVL;
  SPLAVL<int, char> SAVL1, SAVL2, SAVL3, SAVL4, SAVL5;
  AVL.setMaxRatio(-1);

  SAVL1.setMaxRatio(0);
  SAVL2.setMaxRatio(0);
  SAVL3.setMaxRatio(0);
  SAVL4.setMaxRatio(0);
  SAVL5.setMaxRatio(0);
```

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

assert(AVL.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) {
  AVL.insert(2*i + 1, i);
  assert(AVL.isBalanced());
  assert(AVL.getSize() == i+1);
}
for (int i = 0; i < 100; ++i) {  // Checks that keys are in the tree.
  assert(AVL.contains(2*i + 1));
}

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

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

assert(AVL.isBalanced());
//================================================

// The following tests use inserts that create LL, LR, RL, RR imbalances.
// They ensure that the tree is balanced in the right way.

assert(SAVL1.getSize() == 0);

SAVL1.insert(1, 'A');
SAVL1.insert(2, 'B');
SAVL1.insert(3, 'C');

queue1 = SAVL1.getPostOrder();
queue2 = SAVL1.getPreOrder();
queue3 = SAVL1.getInOrder();
queue4 = SAVL1.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;

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

assert(SAVL2.getSize() == 0);

SAVL2.insert(3, 'C');
SAVL2.insert(2, 'B');
SAVL2.insert(1, 'A');

queue1 = SAVL2.getPostOrder();
queue2 = SAVL2.getPreOrder();
queue3 = SAVL2.getInOrder();
queue4 = SAVL2.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;


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

assert(SAVL3.getSize() == 0);

SAVL3.insert(2, 'B');
SAVL3.insert(1, 'A');
SAVL3.insert(3, 'C');

queue1 = SAVL3.getPostOrder();
queue2 = SAVL3.getPreOrder();
queue3 = SAVL3.getInOrder();
```

```
queue4 = SAVL3.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;



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

assert(SAVL4.getSize() == 0);

SAVL4.insert(3, 'C');
SAVL4.insert(1, 'A');
SAVL4.insert(2, 'B');

queue1 = SAVL4.getPostOrder();
queue2 = SAVL4.getPreOrder();
queue3 = SAVL4.getInOrder();
queue4 = SAVL4.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;
```

```
  //==============================================

  assert(SAVL5.getSize() == 0);

  SAVL5.insert(1, 'A');
  SAVL5.insert(3, 'C');
  SAVL5.insert(2, 'B');

  queue1 = SAVL5.getPostOrder();
  queue2 = SAVL5.getPreOrder();
  queue3 = SAVL5.getInOrder();
  queue4 = SAVL5.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;
}

/* SPLAVLinsertTest
 *
 * *This function tests hybridized inserts
 *
 * *It ensures elements go to the right
 *  spot depending on maxRatio and maxCount
 */
void SPLAVLinsertTest(){
  SPLAVL<int, int> SPLAVL;
  Queue< Pair<int,int> >* queue1;
  Queue< Pair<int,int> >* queue2;
  Queue< Pair<int,int> >* queue3;
  Queue< Pair<int,int> >* queue4;

  SPLAVL.setMaxRatio(0);
  SPLAVL.setMaxCount(5);

  for (int i = 1; i<=4; i++){
    SPLAVL.insert(i, i);
  }

  //==============================================
```

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

assert(queue1->dequeue().second == 1);
assert(queue1->dequeue().second == 2);
assert(queue1->dequeue().second == 3);
assert(queue1->dequeue().second == 4);

assert(queue2->dequeue().second == 4);
assert(queue2->dequeue().second == 3);
assert(queue2->dequeue().second == 2);
assert(queue2->dequeue().second == 1);

assert(queue3->dequeue().second == 1);
assert(queue3->dequeue().second == 2);
assert(queue3->dequeue().second == 3);
assert(queue3->dequeue().second == 4);

assert(queue4->dequeue().second == 4);
assert(queue4->dequeue().second == 3);
assert(queue4->dequeue().second == 2);
assert(queue4->dequeue().second == 1);

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

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

SPLAVL.insert(5,5);

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

assert(queue1->dequeue().second == 1);
assert(queue1->dequeue().second == 2);
assert(queue1->dequeue().second == 5);
assert(queue1->dequeue().second == 4);
assert(queue1->dequeue().second == 3);

assert(queue2->dequeue().second == 3);
assert(queue2->dequeue().second == 2);
assert(queue2->dequeue().second == 1);
assert(queue2->dequeue().second == 4);
assert(queue2->dequeue().second == 5);

assert(queue3->dequeue().second == 1);
assert(queue3->dequeue().second == 2);
assert(queue3->dequeue().second == 3);
assert(queue3->dequeue().second == 4);
assert(queue3->dequeue().second == 5);

assert(queue4->dequeue().second == 3);
assert(queue4->dequeue().second == 2);
```

```
    assert(queue4->dequeue().second == 4);
    assert(queue4->dequeue().second == 1);
    assert(queue4->dequeue().second == 5);

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

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

    SPLAVL.insert(6,6);

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

    assert(queue1->dequeue().second == 1);
    assert(queue1->dequeue().second == 2);
    assert(queue1->dequeue().second == 5);
    assert(queue1->dequeue().second == 4);
    assert(queue1->dequeue().second == 3);
    assert(queue1->dequeue().second == 6);

    assert(queue2->dequeue().second == 6);
    assert(queue2->dequeue().second == 3);
    assert(queue2->dequeue().second == 2);
    assert(queue2->dequeue().second == 1);
    assert(queue2->dequeue().second == 4);
    assert(queue2->dequeue().second == 5);

    assert(queue3->dequeue().second == 1);
    assert(queue3->dequeue().second == 2);
    assert(queue3->dequeue().second == 3);
    assert(queue3->dequeue().second == 4);
    assert(queue3->dequeue().second == 5);
    assert(queue3->dequeue().second == 6);

    assert(queue4->dequeue().second == 6);
    assert(queue4->dequeue().second == 3);
    assert(queue4->dequeue().second == 2);
    assert(queue4->dequeue().second == 4);
    assert(queue4->dequeue().second == 1);
    assert(queue4->dequeue().second == 5);

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

}
```