// FILE: IntSet.cpp - header file for IntSet class

// Implementation file for the IntStore class

// (See IntSet.h for documentation.)

// INVARIANT for the IntSet class:

// (1) Distinct int values of the IntSet are stored in a 1-D,

// dynamic array whose size is stored in member variable

// capacity; the member variable data references the array.

// (2) The distinct int value with earliest membership is stored

// in data[0], the distinct int value with the 2nd-earliest

// membership is stored in data[1], and so on.

// Note: No "prior membership" information is tracked; i.e.,

// if an int value that was previously a member (but its

// earlier membership ended due to removal) becomes a

// member again, the timing of its membership (relative

// to other existing members) is the same as if that int

// value was never a member before.

// Note: Re-introduction of an int value that is already an

// existing member (such as through the add operation)

// has no effect on the "membership timing" of that int

// value.

// (4) The # of distinct int values the IntSet currently contains

// is stored in the member variable used.

// (5) Except when the IntSet is empty (used == 0), ALL elements

// of data from data[0] until data[used - 1] contain relevant

// distinct int values; i.e., all relevant distinct int values

// appear together (no "holes" among them) starting from the

// beginning of the data array.

// (6) We DON'T care what is stored in any of the array elements

// from data[used] through data[capacity - 1].

// Note: This applies also when the IntSet is empry (used == 0)

// in which case we DON'T care what is stored in any of

// the data array elements.

// Note: A distinct int value in the IntSet can be any of the

// values an int can represent (from the most negative

// through 0 to the most positive), so there is no

// particular int value that can be used to indicate an

// irrelevant value. But there's no need for such an

// "indicator value" since all relevant distinct int

// values appear together starting from the beginning of

// the data array and used (if properly initialized and

// maintained) should tell which elements of the data

// array are actually relevant.

//

// DOCUMENTATION for private member (helper) function:

// void resize(int new\_capacity)

// Pre: (none)

// Note: Recall that one of the things a constructor

// has to do is to make sure that the object

// created BEGINS to be consistent with the

// class invariant. Thus, resize() should not

// be used within constructors unless it is at

// a point where the class invariant has already

// been made to hold true.

// Post: The capacity (size of the dynamic array) of the

// invoking IntSet is changed to new\_capacity...

// ...EXCEPT when new\_capacity would not allow the

// invoking IntSet to preserve current contents (i.e.,

// value for new\_capacity is invalid or too low for the

// IntSet to represent the existing collection),...

// ...IN WHICH CASE the capacity of the invoking IntSet

// is set to "the minimum that is needed" (which is the

// same as "exactly what is needed") to preserve current

// contents...

// ...BUT if "exactly what is needed" is 0 (i.e. existing

// collection is empty) then the capacity should be

// further adjusted to 1 or DEFAULT\_CAPACITY (since we

// don't want to request dynamic arrays of size 0).

// The collection represented by the invoking IntSet

// remains unchanged.

// If reallocation of dynamic array is unsuccessful, an

// error message to the effect is displayed and the

// program unconditionally terminated.

#include "IntSet.h"

#include <iostream>

#include <cassert>

using namespace std;

void IntSet::resize(int new\_capacity)

{

if (new\_capacity < used)

new\_capacity = used;

if (new\_capacity < 1)

new\_capacity = 1;

capacity = new\_capacity;

int \* newData = new int[capacity];

for (int i = 0; i < used; ++i)

newData[i] = data[i];

delete [] data;

data = newData;

}

IntSet::IntSet(int initial\_capacity) : capacity(DEFAULT\_CAPACITY), used(0)

{

if (initial\_capacity >= 1)

capacity = initial\_capacity;

data = new int[capacity];

}

IntSet::IntSet(const IntSet& src)

: capacity(src.capacity), used(src.used)

{

data = new int[src.capacity];

for (int i = 0; i < used; i++)

data[i] = src.data[i];

}

IntSet::~IntSet()

{

delete [] data;

}

IntSet& IntSet::operator=(const IntSet& rhs)

{

if (this != &rhs)

{

int \*newData = new int[rhs.capacity];

for (int i = 0; i < rhs.used; ++i)

newData[i] = rhs.data[i];

delete [] data;

data = newData;

capacity = rhs.capacity;

used = rhs.used;

}

return \*this;

}

int IntSet::size() const { return used; }

bool IntSet::isEmpty() const

{

if (used == 0) return true;

return false;

}

bool IntSet::contains(int anInt) const

{

for (int i = 0; i < used; i++)

if (data[i] == anInt) return true;

return false;

}

bool IntSet::isSubsetOf(const IntSet& otherIntSet) const

{

if (isEmpty() == true) return true;

for (int i = 0; i < used; i++)

if (otherIntSet.contains(data[i]) == false) return false;

return true;

}

void IntSet::DumpData(ostream& out) const

{ // already implemented ... DON'T change anything

if (used > 0)

{

out << data[0];

for (int i = 1; i < used; ++i)

out << " " << data[i];

}

}

IntSet IntSet::unionWith(const IntSet& otherIntSet) const

{

IntSet newSet = \*this;

if (used + otherIntSet.used > capacity)

newSet.resize( int(1.5 \* (used + otherIntSet.used) + 1) );

for (int i = 0; i < otherIntSet.used; i++)

newSet.add(otherIntSet.data[i]);

return newSet;

}

IntSet IntSet::intersect(const IntSet& otherIntSet) const

{

IntSet newSet = \*this;

for (int i = 0; i < newSet.used; i++) {

if (otherIntSet.contains(newSet.data[i]) == false)

{

newSet.remove(newSet.data[i]);

i -= 1;

}

}

return newSet;

}

IntSet IntSet::subtract(const IntSet& otherIntSet) const

{

IntSet newSet = \*this;

for (int i = 0; i < otherIntSet.used; i++)

newSet.remove(otherIntSet.data[i]);

return newSet;

}

void IntSet::reset() { used = 0; }

bool IntSet::add(int anInt)

{

if (contains(anInt) == true) return false;

else {

if (used == capacity) {

resize( int(1.5 \* used) + 1);

}

used += 1;

data[used - 1] = anInt;

return true;

}

}

bool IntSet::remove(int anInt)

{

if (contains(anInt) == true)

{

for (int i = 0; i < used; i++)

if (data[i] == anInt)

{

for (int x = i; x < used; x++)

data[x] = data[x + 1];

used -= 1;

}

return true;

}

return false;

}

bool operator==(const IntSet& is1, const IntSet& is2)

{

if (is1.size() != is2.size()) return false;

if (is1.isEmpty() == true && is2.isEmpty() == true) return true;

bool isEqual = is2.isSubsetOf(is1);

return isEqual;

}