```
#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 < DEFAULT_CAPACITY) new_capacity =
DEFAULT_CAPACITY;
   capacity = new_capacity;
   int* newArr = new int[capacity];
   for (int i = 0; i < used; ++i) newArr[i] = data[i];
   delete [] data;
   data = newArr;
}

IntSet::IntSet(int initial_capacity) : capacity(initial_capacity),
used(0) {
   if (capacity < 1) capacity = DEFAULT_CAPACITY;
   data = new int[capacity];
}

IntSet::IntSet(const IntSet& src) : capacity(src.capacity),
used(src.used) {
   data = new int[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* newArr = new int[rhs.capacity];
      for (int i = 0; i < rhs.used; i++) newArr[i] = rhs.data[i];
      delete [] data;
      data = newArr;
      capacity = rhs.capacity;
      used = rhs.used;
   }
   return *this;
}

int IntSet::size() const { //remains the same
   return used;
}

bool IntSet::isEmpty() const { //remains the same
   return used == 0;
}

bool IntSet::contains(int anInt) const { //remains the same
   bool found = false;
   for (int i = 0; i < used; i++) {
       if (data[i] == anInt){
           found = true;
           break;
       }
   }
   return found;
}
```

```
bool IntSet::isSubsetOf(const IntSet& otherIntSet) const {
//remains the same
   bool success = true;
   for (int i=0; i < used;i++) {
       if (!otherIntSet.contains(data[i])) {
           success = false;
           break;
       }
   }
   return success;
}

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 {
//problem with silent failure fixed due to array being dynamic
    IntSet temp(used + otherIntSet.used); //this sets the size of the
temp array to be large enough to take the union data
   for (int i = 0; i < used; i++){
      temp.add(data[i]);
   }
   for (int i = 0; i < otherIntSet.size(); i++){
      temp.add(otherIntSet.data[i]);
   }
    return temp;
}

IntSet IntSet::intersect(const IntSet& otherIntSet) const {
//remains the same
   IntSet temp(used); //reduces load on resizing
   for (int i = 0; i < used; i++) {
       if (otherIntSet.contains(data[i])) temp.add(data[i]);
   }
   return temp;
}

IntSet IntSet::subtract(const IntSet& otherIntSet) const {
//remains the same
   IntSet temp(used);
   for (int i=0;i<used;i++) {
       if (!otherIntSet.contains(data[i])) temp.add(data[i]);
   }
   return temp;
}

void IntSet::reset() { //hope this observes the invariant better
   for (int i = used-1; i > -1; i--) data[i] = 0;
   used = 0;
}

bool IntSet::add(int anInt) { //I believe this should be fine?
   bool success = false;
   if (!contains(anInt)){
      if (used == capacity) {
         resize(1.5 * capacity + 1);
      }
      data[used] = anInt;
      success = true;
      used++;
   }
   return success;
}

bool IntSet::remove(int anInt) { //does not need to alter capacity
   bool success = false;
   int location = -1;
    for (int i = 0; i < used; i++) { //fixed out of bound
        if (anInt == data[i]) {
            location = i;
            break;
        }
    }
    if (location>-1) {
        for (int i = location + 1; i < used; i++){ //hopefully fixed
invariant observation
            data[i-1] = data[i];
        }
        success = true;
        used--;
    }
    return success;
}

bool operator==(const IntSet& is1, const IntSet& is2) {
    if (is1.size() != is2.size()) return false;
    if (!is1.isSubsetOf(is2)) return false;
    return true;
}
```