Programming Assignment 1
CSCI 235
Ilya Korsunsky
Due Friday September 27 (midnight)

Follow the instructions presented in the Programming Rules documents. Submit only the header and source files as well as your Makefile. Note that your program needs to compile in order to get any credit at all.

Adhere to the style guidelines.

Start Early!

---

a) Provide a complete templated array-based implementation of the Set Abstract Data Type. The Set ADT is the same as the Bag ADT with the only difference that duplicate items are not allowed. The items do not have to be in sorted order.

You can (and should) copy the code provided for the implementation of the Bag ADT. The only function that needs modification is *Add()* and the function *GetFrequencyOf() is* not needed.

Additionally, add a constructor that constructs a Set for a single item:
Set(const ItemType &an_item);

Similarly to the Bag implementation provide four files: SetInterface.h, Set.h, Set.cpp and TestSet.cpp.

In the file TestSet.cpp provide a number of test cases that demonstrate that your implementation is correct. Do this by writing a function named *TestSetImplementation()* that contains code that does the following:

i) Ensures that an empty set contains no items. This can be done as follows:
Set<int> a_set;
cout << "This is the empty set, and IsEmpty() returns " << a_set.IsEmpty() << endl;
// Use the above logic for the all the following tests.
ii) Adds one item to an empty set, and then searches for it (it should be there).
iii) Creates a set by adding the following items: 1, 10, 3, 10, 5, 10 in that order.

iv) Ensures that the set now holds only 4 items.

v) Ensures that the search (function *Contains()*) of an item in the Set returns true, and that the search of an item not in the Set returns false.

vi) Ensures that trying to add more items than the maximum capacity, results in a false flag returned by *Add()*.

vii) Ensures that adding an item already in the Set results in a false flag returned by *Add()*.

viii) Ensures that removing an item from an empty set results in a false flag returned by the function *Remove().*

ix) Ensures that you can remove a given item from the Set.

b) Write two templated client functions (i.e. functions that are not part of the Set class; you can place them at the end of the Set.cpp file, but after the class implementation):

1) A function named *DisplaySet()* that gets as an argument a set, and displays the contents of the set. Here is the signature of the function:

```
template <class ItemType>
void DisplaySet(const Set<ItemType> &a_set) {
…
}
```

2) A function named *UniteSets()* that gets as arguments two Sets set1 and set2. The function *UniteSets()* creates a new set that contains all elements in sets set1 and set2. This new set is returned by the function. Note that sets set1 and set2 shouldn't be modified. Here is the signature of the function:

```
template <class ItemType>
Set<ItemType> UniteSets(const Set<ItemType> &set1, const Set<ItemType> &set2) {
…
}
```

In the file TestSet.cpp provide a number of test cases for the *UniteSets()* function. Do this by writing a function *TestUniteSets()* that contains code that does the following:

i) Ensures that the union of two empty sets is an empty set.

ii) Ensures that if one of the two input sets is empty, the result equals the non-empty set. Use the function *DisplaySet()* to print the sets (same holds for the next tests).

iii) Use two sets that do not have common elements (for instance {1, 20, 30} and {40, 50}) and ensure that the result is correct.

iv) Use two sets with common elements (for instance {1, 20, 30} and {20, 30, 0, 40}) and ensure that the result is correct.