CSCI 132 Data Structures—Lab #2

Introduction

In today's lab you will practice implementing new Abstract Data Types by working with the `Set` ADT. A set is similar the ADT Bag we discussed in lecture, but while a bag may contain more than one instance of a single entry, a set never contains any duplicates.

Start the lab as follows:

- Log onto your radius account and open a terminal.
- Copy the code for lab2 into your labs folder: `cp -r ~csci132/labs/lab2 ~/labs/`
- Change directories into your new `lab2` directory: `cd ~/labs/lab2/`
- Type `ls` or `ls -l` to verify that your `lab2` directory contains the following files:
  - `SetInterface.h`
  - `ArraySet.h` and `ArraySet.cpp`
  - `ParallelArraySet.h` and `ParallelArraySet.cpp`
  - `TestSet.cpp`

SetInterface and ArraySet

A *set* is an important structure in computer science and discrete math, representing a collection of items with no duplicates and no fixed ordering for the items. `SetInterface.h` defines an interface for a *set* ADT.

`ArraySet.h` and `ArraySet.cpp` give the specification and implementation of a class that satisfies that *set* interface, using an array internally to hold the items. The `ArraySet` class has a `itemCount` member variable to keep track of how many items have been added. `ArraySet` maintains two *invariants*, conditions that must be guaranteed true at all times: (1) it ensures there are never duplicate items in the set; and (2) at all times the items contained in the set are found near the start of the array, with no "gaps" or "holes". In other words, array positions `0` through `itemCount - 1` will contain the items added to the set, and array positions `itemCount` through `maxItems - 1` will contain uninitialized or garbage values.

- Write code to implement the `add` method of the `ArraySet` class.

This will be similar to the implementation of the `add` method for `ArrayBag` we discussed in class. But to maintain the first invariant, the new item should only be added to the array if it was not already present in the array. Your `add` method should return `true` when it succeeds in adding a new, non-duplicate item. If client code attempts to add an entry that is already in the set, or if the array is already full, your function should return `false` and not add the item.

Hint: The private member function `getIndexOf` can be useful here, and will help keep your code simple and tidy. The code for `getIndexOf` is already written for you. Here is the specification of the `getIndexOf` function.

```
Returns either the index of the element in the array items that
contains the given target or -1, if the array does not contain
the target.
```

Checkpoint 1:
Finish the implementation of the `add` function by compiling and running the `TestSet.cpp` file:

```
clang++ -Wall TestSet.cpp -o example
./example
```

SetInterface and ArraySet Client Code

`SetInterface` provides only the most basic set operations, e.g. adding and removing elements. Use those basic operations to implement more advanced operations for sets in `TestSet.cpp`:

- `SetInterface<int>* merge(const SetInterface<int>* first, const SetInterface<int>* second);`
  Return a new set that contains all of the entries that are either in `first` or in `second`.
- `SetInterface<int>* intersection(const SetInterface<int>* first, const SetInterface<int>* second);`
  Return a new set that contains all of the entries that are in both `first` and `second`.
- `SetInterface<int>* difference(const SetInterface<int>* first, const SetInterface<int>* second);`
  Return a new set that contains all of the entries that are in `first` but not in `second`. This operation is not symmetrical—the new set does not include any elements from `second`, regardless of whether those elements are in `first`.

A prototype for each of these functions is provided near the top of `TestSet.cpp`, and a stub implementation can be found near the bottom of that same file. Complete the implementation of these functions.

Note: the parameters and return values for these functions are *pointers*. We have not covered pointers in lecture yet. But simply use the "`->`" operator instead of the "`.`" operator to access members. For example: `first->add(x)` instead of `first.add(x)`, `first->toVector()` instead of `first.toVector()`.

You don't need any error checking for these functions today. Instead, think about what scenarios might cause the function to fail to work properly.

- Add appropriate preconditions for these functions. If any of your functions could fail to work in some cases, specify an appropriate precondition to describe the conditions needed to ensure that function won't fail. If the function can't fail, simply write "`Preconditions: none.`"

When writing client code, like these functions, you will only have access to public methods and data of other classes. You will likely need to use the `toVector` method to iterate through the items in a given set. See the implementation code for the `printSet` function in `TestSet.cpp` for an example of how to iterate through the items in a set using `toVector`.

Checkpoint 2:
Finish implementing all three functions and test them again using `TestSet.cpp`. See sample output below.

Sample Output

Your output should look as follows. Look closely to ensure there are never any duplicates in any set. The order of the items within a set does not matter, but will probably match the example output below in most cases, unless your code does something unusual.

```
Set 1 (contains 5 elements) : 2 5 32 8 17
Set 2 (contains 5 elements) : 4 2 16 8 32
Could not add 9 to set three.
Could not add 10 to set three.
Set 3 (contains 8 elements) : 6 2 3 5 8 4 7 1

Computing union of Set 1 and Set 2... result is 2 5 32 8 17 4 16
Computing intersection of Set 1 and Set 2... result is 2 32 8
Computing difference of Set 1 and Set 3... result is 32 17
Computing union of Set 1 and Set 3... result is 2 5 32 8 17 6 3 4
```

ParallelArraySet

The `ArraySet` code maintains an `itemCount` variable and ensures that any items in the array are contiguous, iwth filled array slots at the start of the array and no "gaps" or "holes". In lecture we discussed an alternative array-based implementation strategy using a second, parallel array of `bool` values, rather than an `itemCount` variable. In this alternative strategy, the elements are still stored in an `items` array, with no duplicates, but a new item can be added in any unoccupied `items` slot, and when an item is removed it can creates a gap in the `items` array. The second `bool` array keeps track of which `items` slots are occupied.

Implement this alternative strategy in class `ParallelArraySet`.

- Examine `ParallelArraySet.h` to see the new variables and invariants for this alternative strategy.
- Complete all methods within the `ParallelArraySet.cpp` file to implement the new strategy. Some methods are already written for you.

Hint: Some functions in `ParallelArraySet.cpp` and `ArraySet.cpp` may be nearly identical, with only small differences needed. Other functions will be completely different, and some may be more complicated now. Most functions will likely involve a loop that iterates through the `bool` array.

- Test your code again using `TestSet.cpp`. In order to test the new class, you should modify `TestSet.cpp` so that it uses `ParallelArraySet` instead of `ArraySet`. Fortunately, because we used proper C++ interfaces, only one or two lines in `TestSet.cpp` will need to be changed.

What to Turn In

- A file named `lab2.txt` with your name and the names of people you worked with during this lab. A full "discussion log" is not necessary. There were no questions to answer in this lab.
- Your modified C++ files: `TestSet.cpp`, `ArraySet.cpp`, and `ParallelArraySet.cpp`

Be sure that the prologue for each C++ file contains your name, course, date, and purpose of the program or a description of the contents of the file (e.g., "`specification of the ParallelArraySet class`" or "`implementation of the ParallelArraySet class`").

Use the following commands to submit your files:

```
cd ~/labs/lab2/
~csci132/bin/submit
```