# CS3021
# Data Structures and Intermediate Programming
# Program Assignment 4

(Updated Thursday June 5, 2014)

## Overview

In this assignment, you will implement an abstract data type Set and its two subclasses TreeSet and HashSet. The Set class is a pure abstract super class, so you cannot create its instance. The actual set you will use in a program is an instance of either the TreeSet or the HashSet subclass. The TreeSet subclass implements the set abstraction using a binary search tree, while the HashSet does it by using a hash table. The TreeSet instance has a better worst case performance for updates. In contrast, the HashSet instance has a much better average case performance for search.

## Problem Statement

An abstract data type Set realizes the mathematical set, which is a collection of items with no duplicates. A set does not have a notion of order, unlike a list. Your task is to implement the Set type and support basic set operations such as union and intersection. The Set type is realized by three classes, the Set superclass and two subclasses TreeSet and HashSet. The complete declaration for the Set superclass is provided to you. Your task is to implement the concrete subclasses TreeSet and HashSet. You use some variation of a binary tree structure to implement the TreeSet subclass and a hash table for the HashSet subclass. You are not allowed to use any system library classes such as the vector or the list when implementing the internal data structure for the subclasses.

## Specification

Here's the declaration for the Set superclass.

```
class Set {

public:

   //--------- Update Operations ---------------------------//
    virtual void insert(string item) = 0;

    virtual void remove(string item) = 0;

   //--------- Set Operations ---------------------------//
    virtual Set* set_union(Set* s2) = 0; //set union

    virtual Set* set_intersect(Set* s2) = 0; //set intersection

    virtual Set* set_diff(Set* s2) = 0; //set difference
```

```
   //--------- Access Operations --------------------//
    virtual vector<string> scan( ) = 0;

   //--------- Queries ----------------------------//
    virtual int size( ) = 0;

    virtual bool isEmpty( ) = 0;

    virtual bool in(string item) = 0;
};
```

## Operations

*Please read the specification for each function very, very carefully.*

| Function | Specification |
|---|---|
| `void insert(string item)` | Insert an item into the set. If the item already exists in the set, do nothing. |
| `void remove(string item)` | Delete the designated item from the set. If the designated item is not found, do nothing. |
| `Set* set_union(Set* s2)` | Return the set union of this set and s2. This set and s2 remain unchanged. |
| `Set* set_intersect(Set* s2)` | Return the set intersection of this set and s2. This set and s2 remain unchanged. |
| `Set* set_diff(Set* s2)` | Return the set difference of this set and s2. This set and s2 remain unchanged. Remove all items in s2 from this set. An item in this set but not in s2 remains in the result. |
| `vector<string> scan( );` | Return the items in the set as a vector of items. If the set is empty, then return an empty vector, i.e., a vector with no elements, not a NULL. |
| `int size( );` | Return the number of items in the set. |
| `bool isEmpty( );` | Return true if the set is empty; otherwise, return false. |
| `bool in(string item);` | Return true if the item is in the set. Otherwise, return false. |

Here's a sample use:

```
Set * ps1 = new TreeSet();
Set * ps2 = new TreeSet();
Set * ps3;
```

```
ps1->insert("Hello");
ps1->insert("World");

ps2->insert("C++");
ps2->insert("Hello");

ps3 = ps1->set_union(ps2);
```

# Requirements

1. All functions must have good performance. For example, when taking a union of two set, you should aim for an algorithm that performs better than inserting all elements from s1 and s2, one by one, into s3.
2. There should be absolutely NO input and output statements in the functions. You may place temporary I/O statements in a function for testing/debugging purposes during the development, but you must remove them before the submission.
3. DO NOT CHANGE the public member functions defined in the file Set.h. The instructor's test program will import functions from your module so it is critical that the function signatures are not changed.
4. DO NOT ADD any new public member functions to the Set class. But, you may add as many private data members and private helper functions as you wish.
5. Functions you define in the program should not access any global variables, those define outside of the class. Use of the global symbolic constants is fine.
6. In your source files, include a header comment with your name, the course number, the assignment number, and any detailed explanation of your functions as appropriate.
7. Format the program in a clean manner. Group statements into logical groups and include a descriptive comment for each group. Group comments do not have to be verbose. One sentence would suffice if the statements in a group are written correctly and precisely. Include enough blank lines between logical groups.
8. Insert one or more blank lines between statements as necessary to make the code more readable and easier to follow. Do not pack statements too densely (e.g., 20 statements with no blank lines between them).
9. Use comments judiciously. Do not include redundant and meaningless comments.

# Starter Code

The following source files are provided to you.

| Filename | Description |
|---|---|
| set.h | The source file for the abstract superclass Set. *Do not make any changes to the public segment of the class declaration. The private segment, however, is under your complete control.* |
| prog4.cpp | This sample main program illustrates a basic interaction with a set. |
| vectorset.h<br>vectorset.cpp | This is a partially completed sample subclass to illustrate how the set union can be implemented using a vector. This class is provided solely for the purpose of giving you some idea. You do not have to do anything to this class. |

# Development Ideas

1. Implement the TreeSet class using some variation of a binary search tree. Consider making modifications to the base BST class so the set union, intersection, and difference can be implemented efficiently.
2. Implement the HashSet class using one of the hashing techniques. Again, aim for the hashing technique that will allow you to implement the set operations efficiently.

# Testing the Functions

As always, you need to plan effective testing methodology to verify the implementation.

# Submission

Submit the source files for the HashSet and TreeSet class—hashset.h, hashset.cpp, treeset.h, and treeset.cpp—that fully implements the Set class, via the Sakai course website's Program Assignment 4 page.