CSCI 230 PA 6 Submission

Due Date: 04/12/2022 Late (date and time):________________

Name(s): Nero Li

Exercise 1 -- need to submit source code and I/O

-- check if completely done ✓; otherwise, discuss issues below

Source code below:
Merge.h:

```
#ifndef M_H
#define M_H

#include <list>

template <typename E>
class Merge {                                      // generic Merge
    public:                                  // global types
        typedef std::list<E> List;                         // list type
        void merge(List& A, List& B, List& C);             // generic merge
function
    protected:                                     // local types
        typedef typename List::iterator Itor;              // iterator type
                                // overridden functions
        virtual void fromA(const E& a, List& C) = 0;
        virtual void fromBoth(const E& a, const E& b, List& C) = 0;
        virtual void fromB(const E& b, List& C) = 0;
};

template <typename E>                        // generic merge
void Merge<E>::merge(List& A, List& B, List& C) {
    Itor pa = A.begin();                     // A's elements
    Itor pb = B.begin();                     // B's elements
    while (pa != A.end() && pb != B.end()) {            // main merging
loop
        if (*pa < *pb)
            fromA(*pa++, C);                       // take from A
        else if (*pa == *pb)
            fromBoth(*pa++, *pb++, C);                   // take from both
        else
            fromB(*pb++, C);                       // take from B
    }
    while (pa != A.end()) { fromA(*pa++, C); }          // take rest from
A
    while (pb != B.end()) { fromB(*pb++, C); }          // take rest from
B
}

template <typename E>                             // set union
class UnionMerge : public Merge<E> {
    protected:
        typedef typename Merge<E>::List List;
        virtual void fromA(const E& a, List& C)
            { C.push_back(a); }                   // add a
        virtual void fromBoth(const E& a, const E& b, List& C)
            { C.push_back(a); }                         // add a only
        virtual void fromB(const E& b, List& C)
            { C.push_back(b); }                         // add b
};

template <typename E>                             // set intersection
class IntersectMerge : public Merge<E> {
    protected:
        typedef typename Merge<E>::List List;
        virtual void fromA(const E& a, List& C)
            { }                                   // ignore
        virtual void fromBoth(const E& a, const E& b, List& C)
            { C.push_back(a); }                   // add a only
        virtual void fromB(const E& b, List& C)
            { }                                   // ignore
};

template <typename E>                             // set subtraction
class SubtractMerge : public Merge<E> {
    protected:
        typedef typename Merge<E>::List List;
        virtual void fromA(const E& a, List& C)
            { C.push_back(a); }                   // add a
        virtual void fromBoth(const E& a, const E& b, List& C)
            { }                                   // ignore
        virtual void fromB(const E& b, List& C)
            { }                                   // ignore
};

#endif
```

exercise_1.cpp:

```
/*  Program: PA_6_exercise_1
    Author: Nero Li
    Class: CSCI 230
    Date: 04/12/2022
    Description:
        Use classes Merge, UnionMerge, IntersectMerge, and SubtractMerge
        from C++ book to perform basic set operations. Set up a driver
        to test the three set operations: union, intersection, and
        difference.
    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include "Merge.h"

using namespace std;

void test()
{
    list<int> a = {1, 2, 3, 5, 7, 9};   // 1 2 3   5   7   9
    list<int> b = {1, 3, 4, 5, 6, 8};   // 1   3 4 5 6   8
    list<int> result;

    UnionMerge<int> test1;
    test1.merge(a, b, result);
    for (int i : result)
        cout << i << ' ';
    cout << endl;

    result.clear();
    IntersectMerge<int> test2;
    test2.merge(a, b, result);
    for (int i : result)
        cout << i << ' ';
    cout << endl;

    result.clear();
    SubtractMerge<int> test3;
    test3.merge(a, b, result);
    for (int i : result)
        cout << i << ' ';
    cout << endl;
}

int main()
{
    test();

    cout << "Modified by: Nero Li\n";

    return 0;
}
```

Input/output below:

```
1 2 3 4 5 6 7 8 9
1 3 5
2 7 9
Modified by: Nero Li
```

Exercise 2 -- need to submit source code and I/O

-- check if completely done ✓; otherwise, discuss issues below

Source code below:

exercise_2.cpp:

```
/*  Program: PA_6_exercise_2
    Author: Nero Li
    Class: CSCI 230
    Date: 04/12/2022
    Description:
        Set up a class named IntSet that keeps track a set of int values
        from 0 to 999 and you can perform basic operations like creating
        a set (constructor that accepts an array of int values),
        insert(e), remove(e), find(e), and print(). Set up three
        methods/functions to perform union, intersection, and difference
        of two sets and return the new set as shown below.

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

const int SIZE = 1000;

class IntSet
{
    private:
        vector<int> set;

    public:
        IntSet(vector<int> newSet = {})
        {
            sort(newSet.begin(), newSet.end());

            for (int i : newSet)
                if (set.empty() || i != *set.end())
```

```
                    set.push_back(i);
        }

        void insert(int e)
        {
            for (int i : set)
                if (e == i)
                    return;
            set.push_back(e);
        }

        void remove(int e)
        {
            vector<int>::iterator itr = set.begin();
            while (itr != set.end())
            {
                if (*itr == e)
                {
                    set.erase(itr);
                    return;
                }
                itr++;
            }
        }

        bool find(int e)
        {
            for (int i : set)
                if (i == e)
                    return true;

            return false;
        }

        void print()
        {
            for (int i : set)
                cout << i << ' ';
            cout << endl;
        }
};

IntSet setUnion(IntSet s1, IntSet s2)
{
    IntSet s3;

    for (int i = 0; i < SIZE; ++i)
        if (s1.find(i) || s2.find(i))
            s3.insert(i);

    return s3;
}
```

```
IntSet setInter(IntSet s1, IntSet s2)
{
    IntSet s3;

    for (int i = 0; i < SIZE; ++i)
        if (s1.find(i) && s2.find(i))
            s3.insert(i);

    return s3;
}

IntSet setDiff(IntSet s1, IntSet s2)
{
    IntSet s3;

    for (int i = 0; i < SIZE; ++i)
        if (s1.find(i) && !s2.find(i))
            s3.insert(i);

    return s3;
}

int main()
{
    vector<int> v1 = {1, 4, 6};
    vector<int> v2 = {2, 4, 8, 10};
    IntSet s1(v1);
    IntSet s2(v2);
    IntSet s3;

    s3 = setUnion(s1, s2);
    s3.print();

    s3 = setInter(s1, s2);
    s3.print();

    s3 = setDiff(s1, s2);
    s3.print();

    cout << "Author: Nero Li\n";

    return 0;
}
```

Input/output below:

```
1 2 4 6 8 10
4
1 6
Author: Nero Li
```

Answer for Question 1:

The template method pattern can be thought of as a class that needs to be used by inheritance and define several specific functions that are provided virtually inside that pattern class. It provided rules as functions for a programmer that we can generate new classes with those basic functions already written.

Answer for Question 2:

The most classical application that we can use the find/union structure is the family questions, which is also a famous question that I remember we should use this algorithm to work with efficiency. We can modify several members in a certain family into a set, and when we want to find a person, the return will be the family lead. This will decrease our searching time for finding that specific person. This will also apply to other questions like searching for an object that is led by another object.

Extra Credit
Source code below:
extra_credit.cpp:

```
/*  Program: PA_6_extra_credit
    Author: Nero Li
    Class: CSCI 230
    Date: 04/12/2022
    Description:
         Implement the find/union partition structure as a tree-based
         approach. Use Partition.java and Position.java from Java
         textbook as a guide to create your own code, add a driver to
         test it for n operations (make cluster/set, find, and union).

    I certify that the code below is my own work.

      Exception(s): N/A

*/

#include <iostream>
#include <vector>

using namespace std;

template <typename E>
class Partition
{
    public:
```

```
    class Locator
    {
        public:
            E elem;
            int size;
            Locator *parent;
            Locator(int elem)
            {
                this->elem = elem;
                size = 1;
                parent = this;
            }

            int getElement()
            {
                return elem;
            }
    };

public:
    Locator *makeSet(E e)
    {
        return new Locator(e);
    }

    Locator *find(Locator *p)
    {
        Locator *loc = p;
        if (loc->parent != loc)
        {
            loc->parent = find(loc->parent);
        }
        return loc->parent;
    }

    void makeUnion(Locator *A, Locator *B)
    {
        Locator *a = find(A);
        Locator *b = find(B);
        if (a != b)
        {
            if (a->size > b->size)
            {
                b->parent = a;
                a->size += b->size;
            }
            else
            {
                a->parent = b;
                b->size += a->size;
            }
        }
        }
};

int main()
{
    Partition<int> test;
    vector<Partition<int>::Locator*> arr;
    const int SIZE = 12;

    for (int i = 1; i <= SIZE; ++i)
        arr.push_back(test.makeSet(i));


    test.makeUnion(arr[3], arr[0]);
    test.makeUnion(arr[3], arr[6]);

    test.makeUnion(arr[8], arr[1]);
    test.makeUnion(arr[8], arr[5]);
    test.makeUnion(arr[2], arr[8]);

    test.makeUnion(arr[10], arr[4]);
    test.makeUnion(arr[11], arr[4]);
    test.makeUnion(arr[9], arr[4]);
    test.makeUnion(arr[7], arr[4]);

    cout << "A = {1, 4, 7}\n";
    cout << "B = {2, 3, 6, 9}\n";
    cout << "C = {5, 8, 10, 11, 12}\n";

    for (int i = 1; i <= SIZE; ++i)
        cout << i << ": " << test.find(arr[i - 1])->getElement() << endl;

    cout << "Modified by: Nero Li\n";

    return 0;
}
```

Input/output below:

```
A = {1, 4, 7}
B = {2, 3, 6, 9}
C = {5, 8, 10, 11, 12}
1: 1
2: 2
3: 2
4: 1
5: 5
6: 2
7: 1
8: 5
9: 2
10: 5
```

```
11: 5
12: 5
Modified by: Nero Li
```