Mariam Tsirekidze

COMPE 361

Prof: Tsintsadze.

Assignment 4

Description:

In this assignment, you'll implement a generic Set<T> class that is parameterized by the type of object it contains. The set will be used as a basic container of objects of type T. Since this is a set, there is only one copy of any object in it. For example: {1,2,2,3} and {1,2,3} are the same set.

Next, write a generic class SortedSet<T> that is a subclass of Set<T>. Just like its name indicates, the set is sorted (The difference is that, for {2, 3,1}, enumerating Set would give you 2, 3, 1, and enumerating SortSet set would always produce 1, 2, 3), so the type T should have a constraint that T should implement the IComparable interface (Find more information of IComparable on MSDN). Override Add and Remove for SortedSet<T>.

Solution:

I am using another namespace for classes Set and SortedSet, the namespace is : COMPE361, that's why I am implementing "using ProjectAssignment4.COMPE361" in test file.

In Set<T> class

- I have implemented IEnumerable <T> interface
- List<T> _ mySet to save data
- public int Count _ property to return the length of the set
- public bool IsEmpty _ property to return true if the set is empty and false otherwise.
- public Set()_ default constructor
- public Set(IEnumerable<T> e) _ constructor
- public static Set<T> operator +(Set<T> lhs, Set<T> rhs) _ "+"operator overload function that will combine two sets in one(without repetition of the numbers) and returns combined set
- public virtual bool Add(T item) _ that will add the item in the set in case of the item isn't in the set and returns true if the item is added
- public virtual bool Remove(T item) _ removes the item in case of the item is in the set and returns true if the item is removed
- public bool Contains(T item) _ returns true if the set contains item
- public delegate bool F<T>(T elt); _ delegate function
- public Set<T> Filter(F<T> filterFunction) _ that receives function and returns filtered function (that is filtered according passed function)
- public IEnumerator<T> GetEnumerator() and IEnumerator IEnumerable.GetEnumerator() _ functions for implementing IEnumerable interface

in SortedSet<T> class (subclass of Set<T>)

- where T : IComparable<T> _ the type of T should be comparable
- public SortedSet(IEnumerable<T> e) : base(e) _ constructor
- public override bool Add(T item) _ override Add function from base class that function will add the item in its own place in sorted set. Returns true if the item is added in the set otherwise false
- public override bool Remove(T item) _ overrides Remove function from base class that function will remove item from sorted class. Returns true if the item is removed
- public static SortedSet<T> operator +(SortedSet<T> lhs, SortedSet<T> rhs) _ does exactly same as in base class but returns sorted set.

in SetEnumerator<T> class I am defining IEnumerator

- public bool MoveNext() _ returns true if there is another member
- public void Reset() _ resets index

In my program.cs I am testing every method end property of my classes.

Source:

Set class and SortedSet class

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;
using System.Collections;

namespace ProjectAssignment4.COMPE361
{
    public class Set<T> : IEnumerable<T>
    {
        // I am storing the data in list
        protected List<T> mySet;
        // make Count and IsEmpty properties only readable and not writable
        public int Count
        {
            get
            {
                return this.mySet.Count();
            }
        }

        public bool IsEmpty
        {
            get
            {
                return !(this.mySet.Any());
            }
        }
        //default constructor
        public Set()
        {
            this.mySet = new List<T>();
        }
        //constructor
        public Set(IEnumerable<T> e)
        {
            this.mySet = new List<T>();
            foreach (var item in e)
            {
                if (!mySet.Contains(item))
                    this.mySet.Add(item);
            }
        }
        // + operator overload
        public static Set<T> operator +(Set<T> lhs, Set<T> rhs)
        {
            Set<T> newSet = new Set<T>();
            // add all the element that is in the rhs to the new set
            foreach (var item1 in rhs)
            {
                newSet.mySet.Add(item1);
            }
            // add all remaining element from the lhs to the new set
            foreach (var item2 in lhs)
            {
                if (!rhs.Contains(item2))
                {
                    newSet.mySet.Add(item2);
                }
            }
            // return new set
            return newSet;
        }
        //add item to set (using list.Add)
        public virtual bool Add(T item)
        {
            bool added = false;
            if (!mySet.Contains(item))
            {
                this.mySet.Add(item);
                added = true;
            }


            return added;
        }
        // remove item from set (list.Remove)
        public virtual bool Remove(T item)
        {
            bool removed = false;
            if (mySet.Contains(item))
            {
                this.mySet.Remove(item);
                removed = true;
            }
            return removed;
        }
        //returns true if the item is in the set otherwise false
        public bool Contains(T item)
        {
            return mySet.Contains(item);
        }
        //delegate function
        public  delegate bool F<T>(T elt);

        //Filters set acording the passed function
        public Set<T> Filter(F<T> filterFunction)
        {
            Set<T> newSet = new Set<T>();
            foreach (var item in mySet)
            {
                if (filterFunction(item))
                {
                    newSet.mySet.Add(item);
                }
            }
            return newSet;
        }



//implementing IEnumerable interface
    public IEnumerator<T> GetEnumerator()
    {
        foreach (var item in mySet)
        {
            yield return item;
        }

    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }



}
// derived class which sorts the set
public class SortedSet<T> : Set<T> where T : IComparable<T>
{
    //constructor from base class then sort
    public SortedSet(IEnumerable<T> e) : base(e)
    {
        this.mySet.Sort();

    }
    //default constructor
    public SortedSet()
    {
        this.mySet = new List<T>();
    }
    //add item at specific place
    public override bool Add(T item)
    {
        int count = 0;
        bool added = false;
        if (!mySet.Contains(item))
        {
            foreach (var i in mySet.ToList())
            {
                if (item.CompareTo(i) <= 0)
                {
                    this.mySet.Insert(count, item);
                    added = true;
                    break;
                }
                count++;
            }
        }
        return added;
    }
```

```
        //same remove as in base class
        public override bool Remove(T item)
        {
            int count = 0;
            bool removed = false;
            foreach (var i in mySet.ToList())
            {
                if (item.CompareTo(i) == 0)
                {
                    this.mySet.RemoveAt(count);
                    removed = true;
                    break;
                }
                count++;
            }
            return removed;

        }
        //+operator overload
        public static SortedSet<T> operator +(SortedSet<T> lhs, SortedSet<T> rhs)
        {
            SortedSet<T> newSet = new SortedSet<T>();
            foreach (var item1 in rhs)
            {
                newSet.mySet.Add(item1);
            }
            // add all remaining element from the lhs to the new set
            foreach (var item2 in lhs)
            {
                if (!rhs.Contains(item2))
                {
                    newSet.mySet.Add(item2);
                }
            }
            newSet.mySet.Sort();
            // return new set
            return newSet;
        }


    }

}
```

Enumerator class:

```
using System;
using System.Collections.Generic;
using System.Collections;

namespace ProjectAssignment4.COMPE361
{
    // in this class I am defining IEnumerator
    public class SetEnumerator<T> : IEnumerator<Set<T>>
    {
        private Set<T> _set;
        private int _indx;

        //constructor
        public SetEnumerator(Set<T> set)
        {
            _set = set;
            _indx = -1;
        }

        public Set<T> Current => throw new NotImplementedException();

        object IEnumerator.Current => throw new NotImplementedException();

        public void Dispose()
        {
        }

        // if there is another member after index returns true if not returns false
        public bool MoveNext()
        {
            return ++_indx < _set.Count;
        }
        // resets index
        public void Reset()
        {
            _indx = -1;
        }
    }


}
```

Test them:

```
using System;
using ProjectAssignment4.COMPE361;

namespace ProjectAssignment4
{
    class Program
    {
        static void Main(string[] args)
        {
            Set<int> set1 = new Set<int>();
            Console.WriteLine($"is set1 empty? {set1.IsEmpty}"); //outputs true
            foreach (var i in set1)
            {
                Console.Write($"{i} ");
            }

            int[] array1 = new int[] { 1, 3, 5, 7, 9, 1 };
            Set<int> set2 = new Set<int>(array1);

            Console.WriteLine($"is set2 empty?{set2.IsEmpty}"); //outputs false
            foreach(var i in set2)
            {
                Console.Write($"{i} ");
            }

            Console.WriteLine();
            Console.WriteLine($"is 10 added to set1? {set1.Add(10)}");//outputs true
            Console.WriteLine($"length of set1 {set1.Count}"); //outputs 1
            Console.WriteLine($"does set1 contains 10? {set1.Contains(10)}\n"); //outputs
true

            Set<int> set3 = set1 + set2;  //check operator + overload
            Console.WriteLine($"does set3 contains 10? {set3.Contains(10)}");  //true
            Console.WriteLine($"does set3 contains 1? {set3.Contains(1)}");   //true
            Console.WriteLine($"does set3 contains 2? {set3.Contains(2)}");   //false
            Console.WriteLine($"does set3 contains 3? {set3.Contains(3)}");   //true
            Console.WriteLine($"does set3 contains 4? {set3.Contains(4)}");   //false
            Console.WriteLine($"does set3 contains 5? {set3.Contains(5)}");   //true

            Console.WriteLine($"does 10 removed from set3? {set3.Remove(10)}"); //true
            Console.WriteLine($"does set3 contains 10? {set3.Contains(10)}");  //true
            Console.WriteLine($"does 10 removed from set3? {set3.Remove(10)}"); //false
            Console.WriteLine($"does set3 contains 10? {set3.Contains(10)}");  //false

            // if passed value is greater 5 return true otherwise false
            bool filter(int elt)
            {
                if(elt> 5)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
            //filter set3, left only numbers greater then 5
            Set<int> set4 = set3.Filter(filter);
            Console.WriteLine($"\nlength of filtered set3 is {set4.Count}"); //output
should be 2 (numbers: 7, 9)
            Console.WriteLine($"does filtered set3 contains 7? {set3.Contains(7)}");
//true
            Console.WriteLine($"does filtered set3 contains 9? {set3.Contains(9)}\n");
//true


            int[] array3 = new int[] { 100, 3, 500, 9, 7, 1 };
            Set<int> mari5 = new Set<int>(array3);
            var sortedset = new Set<int>.SortedSet<int>(array3);
            Console.WriteLine("sortedset contains: ");
            foreach(var i in sortedset)
            {
                Console.WriteLine(i);
            }
            Console.WriteLine($"is 99 added to sortedset? {sortedset.Add(99)}");
            foreach (var i in sortedset)
            {
                Console.WriteLine(i);
            }
            Console.WriteLine();
            Console.WriteLine($"is 99 removed from the sorted list?
{sortedset.Remove(99)}"); //true
            Console.WriteLine($"is 199 removed from the sorted list?
{sortedset.Remove(199)}"); //false
            foreach (var i in sortedset)
            {
                Console.WriteLine(i);
            }

        }
    }
}
```