**Міністерство освіти і науки України**

**Національний авіаційний університет**

**Навчально-науковий інститут комп’ютерних**

**інформаційних технологій**



Лабораторна робота №3.2

з дисципліни «Об’єктивно-орієнтоване програмування»

«Дослідження колекцій та узагальнених колекцій С#»

Виконав студент:

групи ПІ-224Б

Ляшенко Б.М.

Перевірив викладач:

Домків Т.С

Київ 2022

Історія контролю

|  |  |  |
| --- | --- | --- |
| **Дата** | **Опис** | **Автор** |
| 20.09.2022 | Дослідження колекцій та узагальнених колекцій. | Ляшенко Богдан Михайлович |

**Мета**: дослідити класи-колекції та узагальнені колекції у мові програмування С#.

**Завдання:**

1. Описати клас, заданий варіантом у мові програмування С# (табл. 1).
2. Створити список об’єктів класу, вказаного в п.1, використовуючи будь-яку бібліотечну узагальнену (Generic), звичайну (non-generic) колекцію С#, а також масив. Продемонструвати основні операції з колекціями та масивами: додавання, видалення, оновлення, пошук елементу та прохід по набору даних. Порівняти поведінку та пояснити відмінності. Максимальна оцінка за виконання завдання 1-2 - «задовільно».
3. Створити бінарне дерево об'єктів класу, вказаного в п.1. При цьому для бінарного дерева передбачити можливість зберігати дані будь-якого типу (створити узагальнений тип). Елементами дерева не можуть бути структури. Максимальна оцінка за виконання завдання 1-3 - «добре».
4. Доповнити бінарне дерево з п.3 операцію порівняння (використати узагальнені типи та інтерфейси IComparer<T> і/чи IComparable<T>). Продемонструвати функціонал, реалізований з інтерфейсів порівняння через виклик методів сортування. А також реалізувати власний ітератор, використовуючи інтерфейси <IEnumerable> та <IEnumerator>. Обхід дерева реалізувати відповідно до віріанту з табл 1 (колонка «Порядок обходу дерева\*»). Максимальна оцінки за виконання завдань 1-4 – «відмінно».



**Хід роботи**

MyString

using System;

namespace My\_String

{

public class MyString : IComparable<MyString>// +

{

char[] value;

public MyString(string value)

{

this.value = value.ToCharArray();

}

public string Value

{

get { return new string(value); }

private set

{

this.value = value.ToCharArray();

}

}

public int Length

{

get

{

return Value.Length;

}

}

#region Task Methods

public (bool, char[]) IsSubString(string subString)

{

if (subString != null && subString.Length <= Value.Length)

{

int rows = Value.Length;

int columns = subString.Length;

char[] res = new char[Value.Length];

int[,] lcs = new int[rows + 1, columns + 1];

for (int i = 1; i <= rows; i++)

{

for (int j = 1; j <= columns; j++)

{

if (Value[i - 1].Equals(subString[j - 1]))

{

lcs[i, j] = lcs[i - 1, j - 1] + 1;

res[i - 1] = Value[i - 1];

}

else

lcs[i, j] = Math.Max(lcs[i, j - 1], lcs[i - 1, j]);

}

}

if (subString.Length == lcs[rows, columns])

return (true, res);

}

return (false, null);

}

public void InsertSubString(string subString, int index)

{

if (index <= Value.Length)

{

char[] res = new char[Value.Length + subString.Length];

char[] chars = Value.ToCharArray();

int i = 0;

for (; i < index; i++)

res[i] = chars[i];

for (int j = 0; j < subString.Length; j++, i++)

res[i] = subString.ToCharArray()[j];

for (int j = index; j < chars.Length; j++, i++)

res[i] = chars[j];

Value = new string(res);

}

}

public void ChangeSubString(string subString, string newSubString)

{

if (IsSubString(subString).Item1)

{

int index = 0;

char[] sub = IsSubString(subString).Item2;

while (sub[index] == '\0')

index++;

char[] res = new char[Value.Length + subString.Length];

char[] chars = Value.ToCharArray();

int i = 0;

for (; i < index; i++)

res[i] = chars[i];

for (int j = 0; j < newSubString.Length; j++, i++)

res[i] = newSubString.ToCharArray()[j];

for (int j = subString.Length + index; j < chars.Length; j++, i++)

res[i] = chars[j];

Value = new string(res);

}

else

throw new ArgumentException($"this string: {Value} does not contain subString: {subString}");

}

#endregion

#region Object

public override int GetHashCode()

{

return Value.GetHashCode();

}

public override bool Equals(object obj)

{

if (obj is MyString)

{

MyString my = (MyString)obj;

return 0 == CompareTo(my);

}

return false;

}

public override string ToString()

{

string res = $"{Value}({Length})";

return res;

}

#endregion

public int CompareTo(MyString other)

{

return Value.CompareTo(other.Value);

}

public static implicit operator MyString(string value)

{

MyString my = new MyString(value);

return my;

}

}

}

**BinaryTreeNode**

using System;

namespace MyBinaryTree

{

public class BinaryTreeNode<TNode> : IComparable<TNode> where TNode : class, IComparable<TNode>

{

public BinaryTreeNode(TNode value)

{

Value = value;

}

public TNode Value { get; private set; }

public BinaryTreeNode<TNode> Left { get; set; }

public BinaryTreeNode<TNode> Right { get; set; }

#region IComparable<TNode>

public int CompareTo(TNode other)

{

return Value.CompareTo(other);

}

public int CompareNode(BinaryTreeNode<TNode> other)

{

return Value.CompareTo(other.Value);

}

#endregion

}

}

**BinaryTree**

using System;

using System.Collections;

using System.Collections.Generic;

namespace MyBinaryTree

{

public class BinaryTree<T> : IEnumerable<T>, ICollection<T> where T : class, IComparable<T>

{

BinaryTreeNode<T> root;

int count;

public BinaryTree()

{

root = null;

count = 0;

}

public BinaryTree(IEnumerable storage)

{

foreach (T item in storage)

{

Add(item);

}

}

#region ICollection<T>

public int Count => count;

public bool IsReadOnly => false;

public void Add(T item)

{

BinaryTreeNode<T> current = new BinaryTreeNode<T>(item);

if (root == null)

root = current;

else

add(root, current);

count++;

}

void add(BinaryTreeNode<T> root, BinaryTreeNode<T> current)

{

int res = current.CompareNode(root);

if (res < 0)

{

if (root.Left != null)

add(root.Left, current);

else

root.Left = current;

}

else

{

if (root.Right != null)

add(root.Right, current);

else

root.Right = current;

}

}

public void Clear()

{

root = null;

count = 0;

}

public bool Contains(T item)

{

return FindNodeWithParent(item).node != null;

}

(BinaryTreeNode<T> node, BinaryTreeNode<T> parent) FindNodeWithParent(T Value)

{

BinaryTreeNode<T> node = root;

BinaryTreeNode<T> parent = null;

while (node != null)

{

int res = Value.CompareTo(node.Value);

if (res > 0)

{

parent = node;

node = node.Right;

}

else if (res < 0)

{

parent = node;

node = node.Left;

}

else

break;

}

return (node, parent);

}

public void CopyTo(T[] array, int arrayIndex)

{

for (int i = arrayIndex; i < array.Length; i++)

{

array[i] = PostOrderTraversal()[i];

}

}

public bool Remove(T item)

{

if (Contains(item))

{

count--;

BinaryTreeNode<T> node = FindNodeWithParent(item).node;

BinaryTreeNode<T> parent = FindNodeWithParent(item).parent;

if (node.Right == null)

{

if (node == root)

root = node.Left;

else

{

int res = node.CompareNode(parent);

if (res > 0)

parent.Right = node.Left;

else if (res < 0)

parent.Left = node.Left;

}

}

else if (node.Right.Left == null)

{

node.Right.Left = node.Left;

if (node == root)

root = node.Right;

else

{

int res = node.CompareNode(parent);

if (res > 0)

parent.Right = node.Right;

else if (res < 0)

parent.Left = node.Right;

}

}

else

{

BinaryTreeNode<T> mostLeft = node.Right.Left;

BinaryTreeNode<T> mostLeftParent = node.Right;

while (mostLeft != null)

{

mostLeftParent = mostLeft;

mostLeft = mostLeft.Left;

}

mostLeftParent.Left = mostLeft.Right;

mostLeft.Left = node.Left;

mostLeft.Right = node.Right;

if (node == root)

root = mostLeft;

else

{

int res = node.CompareNode(parent);

if (res > 0)

parent.Right = mostLeft;

else if (res < 0)

parent.Left = mostLeft;

}

}

return true;

}

return false;

}

#endregion

#region IEnumerator<T>

public IEnumerator<T> GetEnumerator()

{

//return PreOrderTraversal().GetEnumerator();

return PostOrderTraversal().GetEnumerator();

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

#endregion

#region Post Order Traversal (left right root)

public List<T> PostOrderTraversal()

{

List<T> list = new List<T>();

return PostOrderTraversal(root, ref list);

}

List<T> PostOrderTraversal(BinaryTreeNode<T> root, ref List<T> list)

{

if (root == null)

throw new Exception("Tree is empty");

if (root.Left != null)

PostOrderTraversal(root.Left, ref list);

if (root.Right != null)

PostOrderTraversal(root.Right, ref list);

list.Add(root.Value);

return list;

}

#endregion

#region Pre Order Traversal (root left right)

public List<T> PreOrderTraversal()

{

List<T> list = new List<T>();

return PreOrderTraversal(root, ref list);

}

List<T> PreOrderTraversal(BinaryTreeNode<T> root, ref List<T> list)

{

if (root == null)

throw new Exception("Tree is empty");

list.Add(root.Value);

if (root.Left != null)

PreOrderTraversal(root.Left, ref list);

if (root.Right != null)

PreOrderTraversal(root.Right, ref list);

return list;

}

#endregion

#region In Order Traversal (left root right)

public List<T> InOrderTraversal()

{

List<T> list = new List<T>();

return InOrderTraversal(root, ref list);

}

List<T> InOrderTraversal(BinaryTreeNode<T> root, ref List<T> list)

{

if (root == null)

throw new Exception("Tree is empty");

if (root.Left != null)

InOrderTraversal(root.Left, ref list);

list.Add(root.Value);

if (root.Right != null)

InOrderTraversal(root.Right, ref list);

return list;

}

#endregion

}

}

**Storages**

using My\_String;

using MyBinaryTree;

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace InteractWithStorages

{

internal class Storages<T> where T : class, IComparable<T>

{

MyString[] initArray;

T[] array;

List<T> list;

ArrayList arrList;

BinaryTree<T> binarytree;

public Storages()

{

array = new T[0];

list = new List<T>();

arrList = new ArrayList();

binarytree = new BinaryTree<T>();

}

public void Init()

{

initArray = new MyString[]

{

new MyString("ABCDE"),

new MyString("ABC"),

new MyString("AB"),

new MyString("ABCD"),

new MyString("ABCDEFG"),

new MyString("ABCDEF"),

new MyString("ABCDEFGK")

};

array = new T[initArray.Length];

System.Array.Copy(initArray, array, array.Length);

CreateArrayList();

CreateList();

CreateBinaryTree();

}

#region List

List<T> CreateList()

{

list.AddRange(array);

return list;

}

public List<T> List

{

get => list;

}

#endregion

#region ArrayList

ArrayList CreateArrayList()

{

arrList.AddRange(array);

return arrList;

}

public ArrayList ArrayList

{

get => arrList;

}

#endregion

#region BinaryTree

BinaryTree<T> CreateBinaryTree()

{

binarytree = new BinaryTree<T>(array);

return binarytree;

}

public BinaryTree<T> BinaryTree

{

get => binarytree;

}

#endregion

#region Array

public T[] Array

{

get => array;

}

public void AddToArray(T value)

{

if (array.Length == 0)

{

array = new T[1];

array[0] = value;

return;

}

T[] newArr = new T[array.Length + 1];

int i = 0;

for (; i < array.Length; i++)

newArr[i] = array[i];

newArr[newArr.Length - 1] = value;

array = newArr;

}

public void DeleteFromArray(T value)

{

if (array.Contains<T>(value))

{

int i = 0;

for (; i < array.Length; i++)

{

if (array[i].Equals(value))

break;

}

System.Array.Copy(array, i + 1, array, i, array.Length - i - 1);

T[] my = new T[array.Length - 1];

System.Array.Copy(array, my, array.Length - 1);

array = my;

}

}

#endregion

public void AddToStorage(dynamic storage, T value) => storage.Add(value);

public void DeleteFromStorage(dynamic storage, T value) => storage.Remove(value);

}

}

**ConsoleMenu**

using My\_String;

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using System.Text.RegularExpressions;

namespace InteractWithStorages

{

public static class ConsoleMenu

{

static Storages<MyString> my = new Storages<MyString>();

static Exception wrongName = new Exception("Wrong name of collection");

#region Regexes

static Regex regexShow = new Regex(@"/show\s+(?<name>[A-Za-z]+)$");

static Regex regexShowBT = new Regex(@"/show\s+(?<name>[A-Za-z]+)\s+(?<order>[A-Za-z]+)");

static Regex regexSearch = new Regex(@"/search\s+(?<name>[A-Za-z]+)\s+(?<value>\w+)");

static Regex regexAdd = new Regex(@"/add\s+(?<name>[A-Za-z]+)\s+(?<value>\w+)");

static Regex regexDelete = new Regex(@"/delete\s+(?<name>[A-Za-z]+)\s+(?<value>\w+)");

#endregion

#region Start

public static void Start()

{

Info();

string input;

do

{

Console.Write("\nEnter the command:");

input = Console.ReadLine();

Console.WriteLine();

try

{

CheckForLongExpression(input);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

}

} while (input.ToLower() != "/end");

}

static void Info()

{

Console.WriteLine(

"All commands:\n" +

" /init - inizialize storages with default values;\n" +

" /storages - list of available storages;\n\n" +

" /add - add new MyString to selected storage;\n" +

" /delete - delete selected MyString from selected storage;\n" +

" /clear - clears all storages;\n\n" +

" /show - show selected storage;\n" +

" /search - search specific element in selected storage;\n\n" +

" /end - end program.");

}

static void Storages()

{

PropertyInfo[] propertyInfos = typeof(Storages<MyString>).GetProperties();

Console.Write("Storages are available: ");

foreach (PropertyInfo property in propertyInfos)

{

Console.Write(property.Name + ", ");

}

Console.WriteLine();

}

static void Init()

{

my.Init();

}

static void CheckForLongExpression(string input)

{

bool isMatch = regexShow.IsMatch(input) || regexSearch.IsMatch(input)

|| regexDelete.IsMatch(input) || regexAdd.IsMatch(input) || regexShowBT.IsMatch(input);

if (isMatch)

{

Match show = regexShow.Match(input);

Match search = regexSearch.Match(input);

Match add = regexAdd.Match(input);

Match delete = regexDelete.Match(input);

Match showBT = regexShowBT.Match(input);

if (show.Success)

{

ShowStorage(show.Groups["name"].ToString());

}

else if (add.Success)

{

AddToStorage(add.Groups["name"].ToString().ToLower(),

add.Groups["value"].ToString());

}

else if (search.Success)

{

SearchInStorage(search.Groups["name"].ToString().ToLower(),

search.Groups["value"].ToString());

}

else if (delete.Success)

{

DeleteFromStorage(delete.Groups["name"].ToString().ToLower(),

delete.Groups["value"].ToString());

}

else if (showBT.Success)

{

string name = showBT.Groups["name"].ToString() + " " +

showBT.Groups["order"].ToString();

ShowStorage(name);

}

}

else

{

switch (input.ToLower())

{

case "/info":

Info();

break;

case "/init":

Init();

break;

case "/add":

Storages();

AddToStorage();

break;

case "/storages":

Storages();

break;

case "/show":

Storages();

ShowStorage();

break;

case "/search":

SearchInStorage();

break;

case "/delete":

Storages();

DeleteFromStorage();

break;

case "/clear":

Clear();

break;

case "/cls":

Console.Clear();

Info();

break;

default:

Console.WriteLine("Ther is no such a command");

break;

}

}

}

#endregion

#region Show

static void ShowStorage(string name = null)

{

if (name == null)

{

Console.Write("Enter name of the storage you want to get:");

name = Console.ReadLine();

}

switch (name.ToLower())

{

case "list":

Display(my.List);

break;

case "arraylist":

Display(my.ArrayList);

break;

case "array":

Display(my.Array);

break;

case "binarytree":

Display(my.BinaryTree);

break;

case "binarytree postorder":

Display(my.BinaryTree.PostOrderTraversal());

break;

case "binarytree preorder":

Display(my.BinaryTree.PreOrderTraversal());

break;

case "binarytree inorder":

Display(my.BinaryTree.InOrderTraversal());

break;

default:

throw wrongName;

}

}

static void Display(IEnumerable storage)

{

int i = -1;

foreach (MyString myString in storage)

{

Console.WriteLine($"{++i}." + myString);

}

if (i == -1)

throw new Exception("Storage is Empty!");

Console.WriteLine();

}

#endregion

#region Add

static void AddToStorage(string name = null, string value = null)

{

if (name == null)

{

Console.Write("Enter name of the storage where you want to add MyString:");

name = Console.ReadLine();

}

switch (name)

{

case "list":

my.AddToStorage(my.List, add(value));

Display(my.List);

break;

case "arraylist":

my.AddToStorage(my.ArrayList, add(value));

Display(my.ArrayList);

break;

case "array":

my.AddToArray(add(value));

Display(my.Array);

break;

case "binarytree":

my.AddToStorage(my.BinaryTree, add(value));

Display(my.BinaryTree);

break;

default:

throw wrongName;

}

}

static MyString add(string value)

{

if (value == null)

{

Console.Write("Enter value which you want to add:");

value = Console.ReadLine();

}

MyString myString = new MyString(value);

return myString;

}

#endregion

#region Delete

static void DeleteFromStorage(string name = null, string value = null)

{

if (name == null)

{

Console.Write("Enter name of storage from which you want to delete MyString:");

name = Console.ReadLine().ToLower();

}

switch (name)

{

case "list":

Display(my.List);

my.DeleteFromStorage(my.List, del(value));

break;

case "arraylist":

Display(my.ArrayList);

my.DeleteFromStorage(my.ArrayList, del(value));

break;

case "array":

Display(my.Array);

my.DeleteFromArray(del(value));

break;

case "binarytree":

Display(my.BinaryTree);

my.DeleteFromStorage(my.BinaryTree, del(value));

break;

default:

throw wrongName;

}

ShowStorage(name);

}

static MyString del(string value)

{

if (value == null)

{

Console.Write("Enter value of MyString you want to delete: ");

value = Console.ReadLine();

}

return value;

}

#endregion

#region Search

static void SearchInStorage(string name = null, string value = null)

{

if (name == null)

{

Console.Write("Enter name of storage where you want to find element: ");

name = Console.ReadLine().ToLower();

}

switch (name)

{

case "list":

contains(name, value, my.List);

break;

case "arraylist":

contains(name, value, null, my.ArrayList);

break;

case "array":

contains(name, value, my.Array);

break;

case "binarytree":

contains(name, value, my.BinaryTree);

break;

default:

throw wrongName;

}

}

static void contains(string name, string value, IEnumerable<MyString> stor, ArrayList arr = null)

{

if (stor == null && arr == null)

throw new Exception("Storage is Empty!");

else if (name == "arraylist")

{

if (value == null)

{

Console.Write("Enter value of item which you want to find: ");

value = Console.ReadLine();

}

Console.WriteLine($"{name} contains {value}: {my.ArrayList.Contains(new MyString(value))}");

return;

}

else if (value == null)

{

Console.Write("Enter value of item which you want to find: ");

value = Console.ReadLine();

}

Console.WriteLine($"{name} contains {value}: {stor.Contains<MyString>(new MyString(value))}");

}

#endregion

static void Clear()

{

my = new Storages<MyString>();

}

}

}

**MainProgram**

using InteractWithStorages;

namespace MainProgram

{

internal class Program

{

static void Main(string[] args)

{

ConsoleMenu.Start();

}

}

}

**Висновок:**

**Під час виконная індивідуального завдання лабораторної роботи я дослідив класи-колекції та узагальнені колекції у мові програмування С#.**