НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ”

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

Кафедра системного програмування та спеціалізованих комп‘ютерних систем

**Лабораторна робота №4  
*З дисципліни: «Операційні системи»***

«**Файлові системи**»

Студентка групи КВ-71

Вернік Михайло

Перевірив(ла):

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Київ 2020

**Завдання**

1. Написати програму, що моделює роботу складових заданої файлової системи згідно варіанта (перелік варіантів представлений нижче у табл.4.3).

Вхідні дані студент задає самостійно з урахуванням особливостей індивідуального варіанта завдання.

1. Зробити візуалізацію роботи програми і кінцевих результатів на різних наборах вхідних даних.

**Варіант 2:**

**Побудова таблиці ідентифікаторів за методом бінарного дерева**

Дерево повинно бути ідеально сбалансованим і впорядкованим. Забезпечити можливість додаткового включення та виключення ідентифікаторів при збереженні сбалансованості.

Забезпечити можливість пошуку заданого ідентифікатора з роздруківкою шляху пошуку та відображенням вигляду дерева.

У якості ідентифікаторів використовувати три букви.

**Текст програми**

**AVLTree.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Lab4.AVL

{

public class AVLTree<T> : ICollection<T>, IList<T>

where T : IComparable<T>

{

public class AVLTreeNode : ICollection<T>, IList<T>

{

public AVLTreeNode(T value, AVLTree<T> tree)

{

this.Value = value;

this.Level = 1;

this.Count = 1;

this.Tree = tree;

}

public AVLTree<T> Tree { get; private set; }

public T Value { get; private set; }

public AVLTreeNode Parent { get; private set; }

public AVLTreeNode LeftHand { get; private set; }

public AVLTreeNode RightHand { get; private set; }

int Level { get; set; }

public int Count { get; private set; }

public void Add(T item)

{

var compare = item.CompareTo(this.Value);

if (compare < 0)

if (this.LeftHand == null)

((this.LeftHand = new AVLTree<T>.AVLTreeNode(item, this.Tree)).Parent = this).Reconstruct(true);

else this.LeftHand.Add(item);

else

if (this.RightHand == null)

((this.RightHand = new AVLTree<T>.AVLTreeNode(item, this.Tree)).Parent = this).Reconstruct(true);

else this.RightHand.Add(item);

}

public void Clear()

{

if (this.LeftHand != null) this.LeftHand.Clear();

if (this.RightHand != null) this.RightHand.Clear();

this.LeftHand = this.RightHand = null;

}

public bool Contains(T item)

{

var compare = item.CompareTo(this.Value);

if (compare < 0)

return this.LeftHand == null ? false : this.LeftHand.Contains(item);

else if (compare == 0)

return true;

else

return this.RightHand == null ? false : this.RightHand.Contains(item);

}

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

{

if (this.LeftHand != null)

{

this.LeftHand.CopyTo(array, arrayIndex);

arrayIndex += this.LeftHand.Count;

}

array[arrayIndex++] = this.Value;

if (this.RightHand != null)

this.RightHand.CopyTo(array, arrayIndex);

}

public bool IsReadOnly { get { return false; } }

public bool Remove(T item)

{

var compare = item.CompareTo(this.Value);

if (compare == 0)

{

if (this.LeftHand == null && this.RightHand == null)

if (this.Parent != null)

{

if (this.Parent.LeftHand == this) this.Parent.LeftHand = null;

else this.Parent.RightHand = null;

this.Parent.Reconstruct(true);

}

else this.Tree.RootNode = null;

else if (this.LeftHand == null || this.RightHand == null)

{

var child = this.LeftHand == null ? this.RightHand : this.LeftHand;

if (this.Parent != null)

{

if (this.Parent.LeftHand == this) this.Parent.LeftHand = child;

else this.Parent.RightHand = child;

(child.Parent = this.Parent).Reconstruct(true);

}

else (this.Tree.RootNode = child).Parent = null;

}

else

{

var replace = this.LeftHand;

while (replace.RightHand != null) replace = replace.RightHand;

var temp = this.Value;

this.Value = replace.Value;

replace.Value = temp;

return replace.Remove(replace.Value);

}

this.Parent = this.LeftHand = this.RightHand = null;

return true;

}

else if (compare < 0)

return this.LeftHand == null ? false : this.LeftHand.Remove(item);

else

return this.RightHand == null ? false : this.RightHand.Remove(item);

}

public IEnumerator<T> GetEnumerator()

{

if (this.LeftHand != null)

foreach (var item in this.LeftHand)

yield return item;

yield return this.Value;

if (this.RightHand != null)

foreach (var item in this.RightHand)

yield return item;

}

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

void Reconstruct(bool recursive)

{

this.Count = 1;

int leftLevel = 0, rightLevel = 0;

if (this.LeftHand != null)

{

leftLevel = this.LeftHand.Level;

this.Count += this.LeftHand.Count;

}

if (this.RightHand != null)

{

rightLevel = this.RightHand.Level;

this.Count += this.RightHand.Count;

}

if (leftLevel - rightLevel > 1)

{

var leftLeft = this.LeftHand.LeftHand == null ? 0 : this.LeftHand.LeftHand.Level;

var leftRight = this.LeftHand.RightHand == null ? 0 : this.LeftHand.RightHand.Level;

if (leftLeft >= leftRight)

{

this.LeftHand.Elevate();

this.Reconstruct(true);

}

else

{

var pivot = this.LeftHand.RightHand;

pivot.Elevate(); pivot.Elevate();

pivot.LeftHand.Reconstruct(false);

pivot.RightHand.Reconstruct(true);

}

}

else if (rightLevel - leftLevel > 1)

{

var rightRight = this.RightHand.RightHand == null ? 0 : this.RightHand.RightHand.Level;

var rightLeft = this.RightHand.LeftHand == null ? 0 : this.RightHand.LeftHand.Level;

if (rightRight >= rightLeft)

{

this.RightHand.Elevate();

this.Reconstruct(true);

}

else

{

var pivot = this.RightHand.LeftHand;

pivot.Elevate(); pivot.Elevate();

pivot.LeftHand.Reconstruct(false);

pivot.RightHand.Reconstruct(true);

}

}

else

{

this.Level = Math.Max(leftLevel, rightLevel) + 1;

if (this.Parent != null && recursive)

this.Parent.Reconstruct(true);

}

}

void Elevate()

{

var root = this.Parent;

var parent = root.Parent;

if ((this.Parent = parent) == null) this.Tree.RootNode = this;

else

{

if (parent.LeftHand == root) parent.LeftHand = this;

else parent.RightHand = this;

}

if (root.LeftHand == this)

{

root.LeftHand = this.RightHand;

if (this.RightHand != null) this.RightHand.Parent = root;

this.RightHand = root;

root.Parent = this;

}

else

{

root.RightHand = this.LeftHand;

if (this.LeftHand != null) this.LeftHand.Parent = root;

this.LeftHand = root;

root.Parent = this;

}

}

public int IndexOf(T item)

{

var compare = item.CompareTo(this.Value);

if (compare == 0)

if (this.LeftHand == null) return 0;

else

{

var temp = this.LeftHand.IndexOf(item);

return temp == -1 ? this.LeftHand.Count : temp;

}

else if (compare < 0)

if (this.LeftHand == null) return -1;

else return this.LeftHand.IndexOf(item);

else

if (this.RightHand == null) return -1;

else return this.RightHand.IndexOf(item);

}

public void Insert(int index, T item) { throw new InvalidOperationException(); }

public void RemoveAt(int index)

{

if (this.LeftHand != null)

if (index < this.LeftHand.Count)

{

this.LeftHand.RemoveAt(index);

return;

}

else index -= this.LeftHand.Count;

if (index-- == 0)

{

this.Remove(this.Value);

return;

}

if (this.RightHand != null)

if (index < this.RightHand.Count)

{

this.RightHand.RemoveAt(index);

return;

}

throw new ArgumentOutOfRangeException("index");

}

public T this[int index]

{

get

{

if (this.LeftHand != null)

if (index < this.LeftHand.Count) return this.LeftHand[index];

else index -= this.LeftHand.Count;

if (index-- == 0) return this.Value;

if (this.RightHand != null)

if (index < this.RightHand.Count) return this.RightHand[index];

throw new ArgumentOutOfRangeException("index");

}

set { throw new InvalidOperationException(); }

}

}

public AVLTreeNode RootNode { get; private set; }

private void Path(AVLTreeNode root, string value, List<string> way)

{

if (root == null) return;

if (root.Value.ToString() == value)

{

way.Add(root.Value.ToString());

}else

if (String.Compare(root.Value.ToString(), value) > 0)

{

way.Add(root.Value.ToString());

Path(root.LeftHand, value, way);

}

else

{

way.Add(root.Value.ToString());

Path(root.RightHand, value, way);

}

}

public List<string> FindPathToNodeFromRoot(String value)

{

List<string> path = new List<string>();

Path(this.RootNode, value, path);

return path;

}

public void Add(T item)

{

if (this.RootNode == null) this.RootNode = new AVLTree<T>.AVLTreeNode(item, this);

else this.RootNode.Add(item);

}

public void Clear()

{

if (this.RootNode == null) return;

this.RootNode.Clear();

this.RootNode = null;

}

public bool Contains(T item) { return this.RootNode == null ? false : this.RootNode.Contains(item); }

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

{

if (array == null) throw new ArgumentNullException("array");

if (arrayIndex < 0) throw new ArgumentOutOfRangeException("arrayIndex");

if ((array.Length <= arrayIndex) || (this.RootNode != null && array.Length < arrayIndex + this.RootNode.Count))

throw new ArgumentException();

if (this.RootNode != null)

this.RootNode.CopyTo(array, arrayIndex);

}

public int Count { get { return this.RootNode.Count; } }

public bool IsReadOnly { get { return false; } }

public bool Remove(T item) { return this.RootNode == null ? false : this.RootNode.Remove(item); }

public IEnumerator<T> GetEnumerator()

{

if (this.RootNode != null)

foreach (var item in this.RootNode)

yield return item;

else

yield break;

}

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

public int IndexOf(T item) { return this.RootNode != null ? this.RootNode.IndexOf(item) : -1; }

public void Insert(int index, T item) { throw new InvalidOperationException(); }

public void RemoveAt(int index) { if (this.RootNode != null) this.RootNode.RemoveAt(index); }

public T this[int index]

{

get

{

if (this.RootNode != null) return this.RootNode[index];

else throw new ArgumentOutOfRangeException("index");

}

set { throw new InvalidOperationException(); }

}

}

}

**Lab4.cs**

using Lab4.AVL;

using Lab4.RBTree;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace Lab4

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

class TreeNodeAVL

{

public TreeNodeAVL(int val)

{

this.val = val;

}

public TreeNodeAVL left;

public TreeNodeAVL right;

public int val;

}

void PrintTree(AVLTree<string>.AVLTreeNode root, TreeNode treeViewRoot)

{

if (root == null) return;

if (treeViewRoot == null)

{

treeViewRoot = new TreeNode(root.Value);

}

else if (root.Value != null)

{

treeViewRoot.Text = root.Value;

}

bool left = false;

bool right = false;

if (root.LeftHand != null && root.LeftHand.Value != null)

{

treeViewRoot.Nodes.Add(root.LeftHand.Value);

left = true;

}

else

{

treeViewRoot.Nodes.Add("null");

}

if (root.RightHand != null && root.RightHand.Value != null)

{

treeViewRoot.Nodes.Add(root.RightHand.Value);

right = true;

}

else

{

treeViewRoot.Nodes.Add("null");

}

PrintTree(root.LeftHand, left ? treeViewRoot.Nodes[0] : null);

PrintTree(root.RightHand, right ? treeViewRoot.Nodes[1] : null);

}

void PrintTree(RedBlackTreeNode<Int32, string> root, TreeNode treeViewRoot)

{

if (root == null) return;

if (treeViewRoot == null)

{

treeViewRoot = new TreeNode(root.Value);

}

else if (root.Value != null)

{

treeViewRoot.Text = root.Value;

}

bool left = false;

bool right = false;

if (root.Left != null && root.Left.Value != null)

{

treeViewRoot.Nodes.Add(root.Left.Value);

left = true;

}

else

{

treeViewRoot.Nodes.Add("null");

}

if (root.Right != null && root.Right.Value != null)

{

treeViewRoot.Nodes.Add(root.Right.Value);

right = true;

}

else

{

treeViewRoot.Nodes.Add("null");

}

PrintTree(root.Left, left ? treeViewRoot.Nodes[0] : null);

PrintTree(root.Right, right ? treeViewRoot.Nodes[1] : null);

}

int nodes = 0;

TreeNodeAVL AddValueToTree(TreeNodeAVL root, int val)

{

if (root == null)

{

return new TreeNodeAVL(val); ;

}

if (root.val > val)

root.left = AddValueToTree(root.left, val);

else

root.right = AddValueToTree(root.right, val);

return root;

}

RedBlackTree<int, string> redBlackTree = new RBTree.RedBlackTree<int, string>();

AVLTree<string> aVLTree = new AVL.AVLTree<string>();

private void btnAdd\_Click(object sender, EventArgs e)

{

if (textBox.TextLength == 3)

{

//redBlackTree.Add(nodes++, textBox.Text);

aVLTree.Add(textBox.Text);

treeView.Nodes.Clear();

treeView.Nodes.Add(redBlackTree.Root.Value);

PrintTree(aVLTree.RootNode, treeView.Nodes[0]);

treeView.ExpandAll();

listBox.Items.Add(textBox.Text);

textBox.Text = "";

}

else

{

}

//TreeNodeAVL root = new TreeNodeAVL(1);

////root.left = new TreeNodeAVL(2);

////root.right = new TreeNodeAVL(3);

////root.right.left = new TreeNodeAVL(4);

////root.left.left = new TreeNodeAVL(5);

//AddValueToTree(root, 12);

//AddValueToTree(root, 13);

//AddValueToTree(root, 14);

//AddValueToTree(root, 15);

//AddValueToTree(root, 16);

//AddValueToTree(root, 22);

//AddValueToTree(root, 100);

//AddValueToTree(root, 1);

//AddValueToTree(root, 4);

//AddValueToTree(root, -12);

//AddValueToTree(root, -122);

}

private void btnDeleteAll\_Click(object sender, EventArgs e)

{

aVLTree = new AVLTree<string>();

redBlackTree = new RedBlackTree<int, string>();

treeView.Nodes.Clear();

treeView.ExpandAll();

richTextBox.Text = "";

listBox.Items.Clear();

}

private void btnDelete\_Click(object sender, EventArgs e)

{

var selected = listBox.SelectedItem.ToString();

if (aVLTree.Remove(selected) == true)

{

// deleted;

treeView.Nodes.Clear();

treeView.Nodes.Add(aVLTree.RootNode.Value);

PrintTree(aVLTree.RootNode, treeView.Nodes[0]);

treeView.ExpandAll();

listBox.Items.Remove(selected);

}

}

private void btnFind\_Click(object sender, EventArgs e)

{

var selected = listBox.SelectedItem.ToString();

var path = aVLTree.FindPathToNodeFromRoot(selected);

if (path != null)

richTextBox.Text = string.Join(" -> ", path.ToArray());

}

}

}

**Тест**

