БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Лабораторная работа № 9

по дисциплине «Структуры и алгоритмы обработки данных»

Вариант № 4

Выполнил студент: Ефименко Павел Викторович,

группа 981063

Минск 2020

**Условие:**

Ввести 10-15 целых чисел и построить из них бинарное дерево поиска.

1.  Выполнить симметричную прошивку бинарного дерева поиска. Обойти его согласно симметричному порядку следования элементов. Реализовать вставку и удаление элементов из симметрично прошитого бинарного дерева.

2.   Выполнить прямую прошивку бинарного дерева поиска. Обойти его согласно прямому порядку следования элементов. Реализовать вставку и удаление элементов из прямо прошитого бинарного дерева.

**Листинг:**

class Program

{

// Прошитая структура дерева дерево

public class MyStitchedTree

{

// Данные

private int data;

public int Data

{

get { return data; }

}

// Левый блок

private MyStitchedTree leftUnit;

public MyStitchedTree LeftUnit

{

get { return leftUnit; }

set { leftUnit = value; }

}

// Правый блок

private MyStitchedTree rightUnit;

public MyStitchedTree RightUnit

{

get { return rightUnit; }

set { rightUnit = value; }

}

// Bool удаления (true | false)

private bool isDeleted;

public bool IsDeleted

{

get { return isDeleted; }

}

// Конструктор

public MyStitchedTree(int value)

{

data = value;

}

// Удаления

public void Delete()

{

isDeleted = true;

}

// Поиск прямой

public MyStitchedTree Search(int value)

{

MyStitchedTree thisEl = this;

while (thisEl != null)

{

if (value == this.data && isDeleted == false)

{

return thisEl;

}

else if (value > thisEl.data)

{

thisEl = thisEl.rightUnit;

}

else

{

thisEl = thisEl.leftUnit;

}

}

return null;

}

// Поиск рекурсивный

public MyStitchedTree SearchRec(int value)

{

if (value == data && isDeleted == false)

{

return this;

}

else if (value < data && leftUnit != null)

{

return leftUnit.SearchRec(value);

}

else if (rightUnit != null)

{

return rightUnit.SearchRec(value);

}

else

{

return null;

}

}

// Добавления значений

public void Add(int value)

{

if (value >= data)

{

if (rightUnit == null)

{

rightUnit = new MyStitchedTree(value);

}

else

{

rightUnit.Add(value);

}

}

else

{

if (leftUnit == null)

{

leftUnit = new MyStitchedTree(value);

}

else

{

leftUnit.Add(value);

}

}

}

// Ищем наименьшее значение

public Nullable<int> MinValue()

{

// Когда достигем левого блока - возвращаем значение или продолжаем выбирать наименьшее значение левого блока

if (leftUnit == null)

{

return data;

}

else

{

return leftUnit.MinValue();

}

}

// Ищем наибольшее значение

internal Nullable<int> MaxValue()

{

if (rightUnit == null)

{

return data;

}

else

{

return rightUnit.MaxValue();

}

}

public void InOrderTraversal()

{

if (leftUnit != null)

{

leftUnit.InOrderTraversal();

}

Console.Write(data + " ");

if (rightUnit != null)

{

rightUnit.InOrderTraversal();

}

}

public void PreOrderTraversal()

{

Console.Write(data + " ");

if (leftUnit != null)

{

leftUnit.PreOrderTraversal();

}

if (rightUnit != null)

{

rightUnit.PreOrderTraversal();

}

}

public void PostorderTraversal()

{

if (leftUnit != null)

{

leftUnit.PostorderTraversal();

}

if (rightUnit != null)

{

rightUnit.PostorderTraversal();

}

Console.Write(data + " ");

}

public int Height()

{

if (this.leftUnit == null && this.rightUnit == null)

{

return 1;

}

int left = 0;

int right = 0;

if (this.leftUnit != null)

{

left = this.leftUnit.Height();

}

if (this.rightUnit != null)

{

right = this.rightUnit.Height();

}

if (left > right)

{

return (left + 1);

}

else

{

return (right + 1);

}

}

public int NumberOfLeafNodes()

{

if (this.leftUnit == null && this.rightUnit == null)

{

return 1;

}

int leftLeaves = 0;

int rightLeaves = 0;

if (this.leftUnit != null)

{

leftLeaves = leftUnit.NumberOfLeafNodes();

}

if (this.rightUnit != null)

{

rightLeaves = rightUnit.NumberOfLeafNodes();

}

return leftLeaves + rightLeaves;

}

public bool IsBalanced()

{

int LeftHeight = LeftUnit != null ? LeftUnit.Height() : 0;

int RightHeight = RightUnit != null ? RightUnit.Height() : 0;

int heightDifference = LeftHeight - RightHeight;

if (Math.Abs(heightDifference) > 1)

{

return false;

}

else

{

return ((LeftUnit != null ? LeftUnit.IsBalanced() : true) && (RightUnit != null ? RightUnit.IsBalanced() : true));

}

}

}

// Прошитое дерево

public class MyStitchedBinaryTree

{

// Основной элемент

private MyStitchedTree main;

public MyStitchedTree Main

{

get { return Main; }

}

// Поиска по значению

public MyStitchedTree Search(int data)

{

if (main != null)

{

return main.Search(data);

}

else

{

return null;

}

}

// Рекурсивный способ поиска

public MyStitchedTree SearchRec(int data)

{

if (main != null)

{

// Вызываем рекурсивный способ поиска

return main.SearchRec(data);

}

else

{

// Или возвращаем пустой

return null;

}

}

// Вставки

public void Add(int data)

{

if (main != null)

{

main.Add(data);

}

else

{

main = new MyStitchedTree(data);

}

}

// Удаление

public void Remove(int data)

{

MyStitchedTree current = main;

MyStitchedTree parent = main;

bool isLeftChild = false;

// Если элемент пустой, то пустой return

if (current == null)

{

return;

}

// Цикл пока не найдем совпадений

while (current != null && current.Data != data)

{

// Устанавливаем текущую в настоящую

parent = current;

if (data < current.Data)

{

// Устанавливаем в левый блок

current = current.LeftUnit;

isLeftChild = true;

}

else

{

// Устанавливаем в правый блок

current = current.RightUnit;

isLeftChild = false;

}

}

if (current == null)

{

return;

}

// Если правая и левая ветки пустые

if (current.RightUnit == null && current.LeftUnit == null)

{

// Если текущая равна основной

if (current == main)

{

main = null;

}

else

{

// Если это левая ветка

if (isLeftChild)

{

// Удаляем левый блок

parent.LeftUnit = null;

}

else

{

// Удаляем правый блок

parent.RightUnit = null;

}

}

}

else if (current.RightUnit == null)

{

// Если текущее значение равно основному

if (current == main)

{

// Основному присваивается левая ветвь

main = current.LeftUnit;

}

else

{

if (isLeftChild)

{

// Если это левый

parent.LeftUnit = current.LeftUnit;

}

else

{

// Присваиваем правму блоку левую ветвь

parent.RightUnit = current.LeftUnit;

}

}

}

else if (current.LeftUnit == null)

{

if (current == main)

{

main = current.RightUnit;

}

else

{

if (isLeftChild)

{

parent.LeftUnit = current.LeftUnit;

}

else

{

parent.RightUnit = current.RightUnit;

}

}

}

else

{

MyStitchedTree successor = GetSuccessor(current);

if (current == main)

{

main = successor;

}

else if (isLeftChild)

{

parent.LeftUnit = successor;

}

else

{

parent.RightUnit = successor;

}

}

}

private MyStitchedTree GetSuccessor(MyStitchedTree node)

{

MyStitchedTree parentOfSuccessor = node;

MyStitchedTree successor = node;

MyStitchedTree current = node.RightUnit;

while (current != null)

{

parentOfSuccessor = successor;

successor = current;

current = current.LeftUnit;

}

if (successor != node.RightUnit)

{

parentOfSuccessor.LeftUnit = successor.RightUnit;

successor.RightUnit = node.RightUnit;

}

successor.LeftUnit = node.LeftUnit;

return successor;

}

public void SoftDelete(int data)

{

MyStitchedTree toDelete = Search(data);

if (toDelete != null)

{

toDelete.Delete();

}

}

public Nullable<int> Smallest()

{

if (main != null)

{

return main.MinValue();

}

else

{

return null;

}

}

public Nullable<int> Largest()

{

if (main != null)

{

return main.MaxValue();

}

else

{

return null;

}

}

public void InOrderTraversal()

{

if (main != null)

{

main.InOrderTraversal();

}

}

public void PreorderTraversal()

{

if (main != null)

{

main.PreOrderTraversal();

}

}

public void PostorderTraversal()

{

if (main != null)

{

main.PostorderTraversal();

}

}

public int NumberOfLeafNodes()

{

if (main == null)

{

return 0;

}

return main.NumberOfLeafNodes();

}

public int Height()

{

if (main == null)

{

return 0;

}

return main.Height();

}

public bool IsBalanced()

{

if (main == null)

{

return true;

}

return main.IsBalanced();

}

}

static void Main()

{

MyStitchedBinaryTree myStitchedBinaryTree = new MyStitchedBinaryTree();

// Заполняем дерево значениями

for (int i = 1; i <= 10; i++)

{

myStitchedBinaryTree.Add(i);

}

Console.WriteLine("Прямой порядок следования элементов: ");

myStitchedBinaryTree.InOrderTraversal();

Console.WriteLine("\n\nСимметричный порядок следования элементов: ");

myStitchedBinaryTree.PreorderTraversal();

Console.WriteLine($"\n\nРезультат поиска: {myStitchedBinaryTree.Search(1).Data}");

myStitchedBinaryTree.Remove(2);

Console.WriteLine("\nПрямой порядок следования элементов после удаления:");

myStitchedBinaryTree.InOrderTraversal();

}

public static int getHeight(MyStitchedTree main)

{

if (main == null)

{

return 0;

}

return Math.Max(getHeight(main.LeftUnit), getHeight(main.RightUnit)) + 1;

}

public static bool isBalanced(MyStitchedTree main)

{

if (main == null)

{

return true;

}

int heightDiff = getHeight(main.LeftUnit) - getHeight(main.RightUnit);

if (Math.Abs(heightDiff) > 1)

{

return false;

}

else

{

return (isBalanced(main.LeftUnit) && isBalanced(main.RightUnit));

}

}

}

**Выполнение:**

