Московский авиационный институт

(Национальный исследовательский университет)

Факультет №3

Кафедра 304

Отчёт по лабораторной работе

По учебной дисциплине

Структуры и алгоритмы обработки данных

На тему:  
“Работа с деревьями”

Выполнил:

Плотников Кирилл

Крещук Николай

Группа: М30-224БК-17

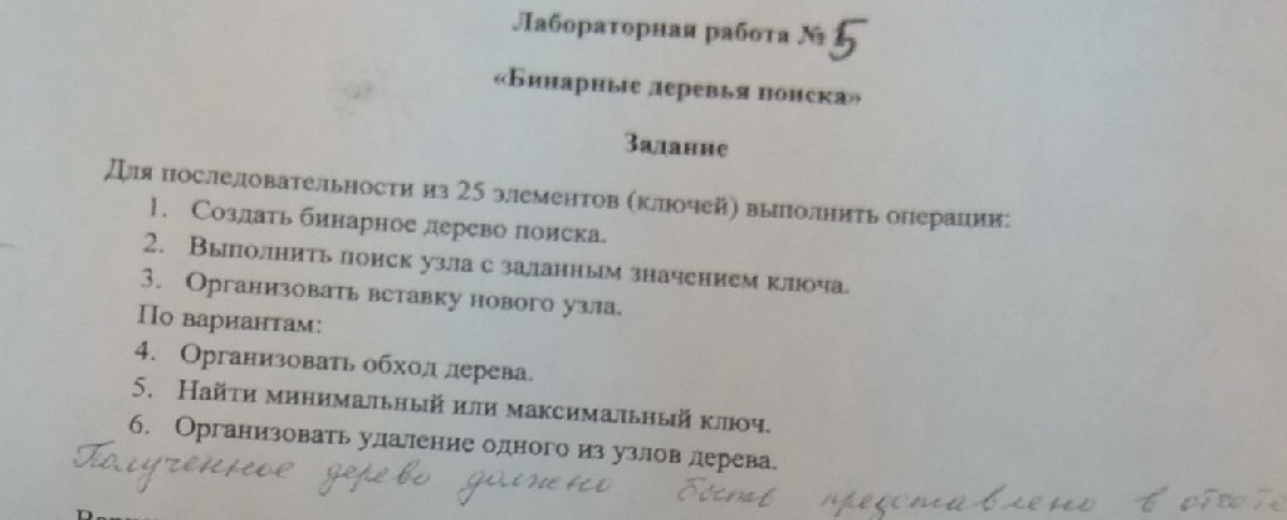
Бригада №7

Принял:

Силаев А.В.

Москва, 2018

**Задание.**



**Номер варианта: 7.**

1. Порядок обхода: прямой.
2. Поиск ключа: минимальный.
3. Удаляемые узел должен иметь один дочерний узел.

**Класс Дерева.**

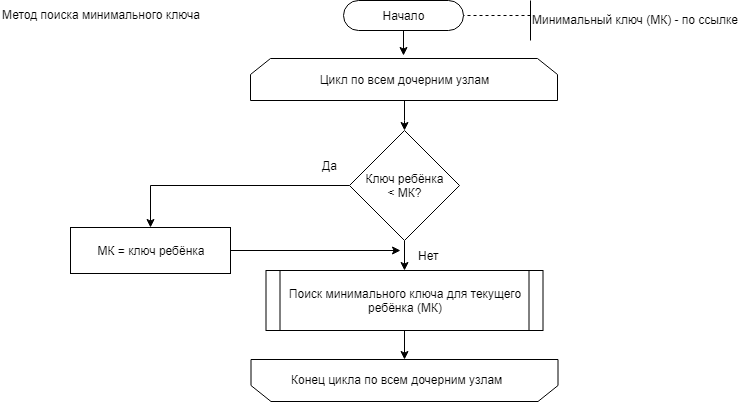
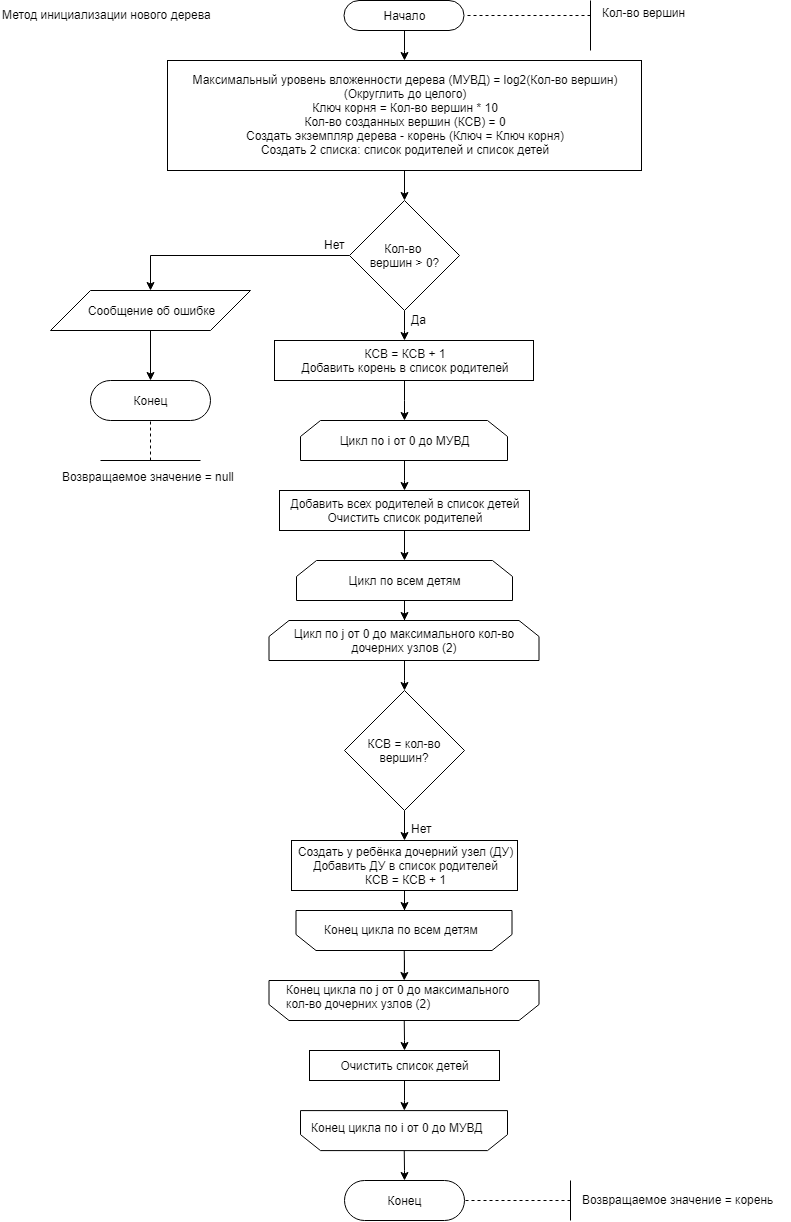
**Имя:** Tree

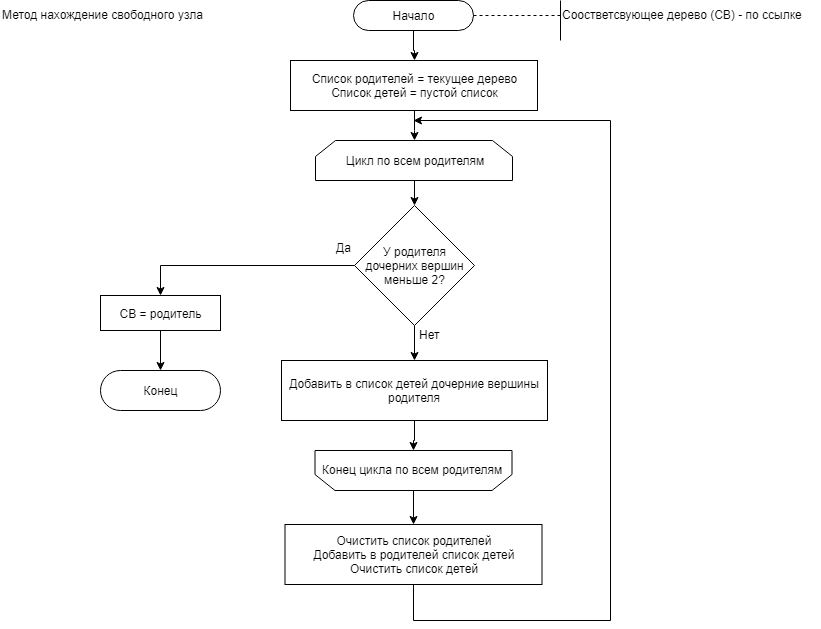
**Поля:**

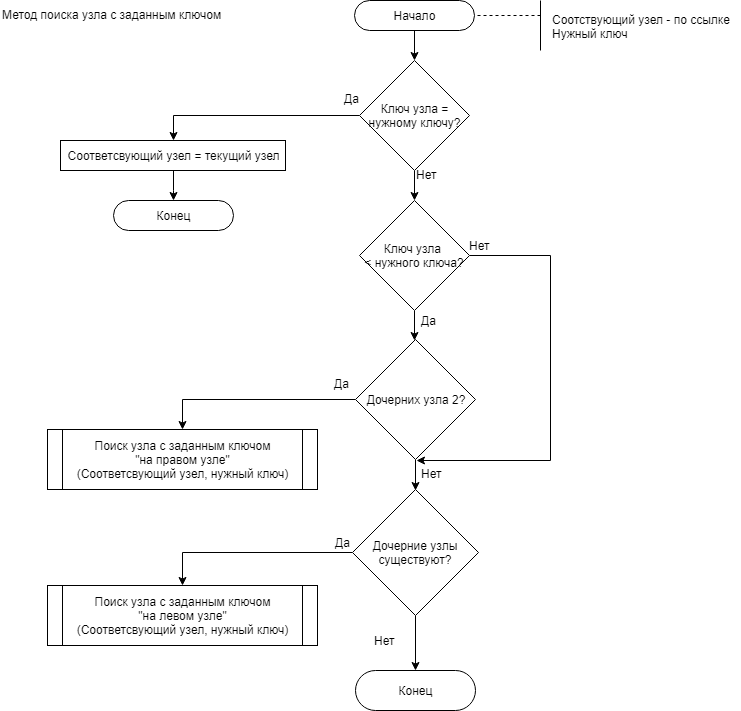
1. Максимальное кол-во вершин (целочисленная константа).
2. Ключ (целочисленное значение).
3. Список из Tree.
4. Номер вершины (целочисленное значение).

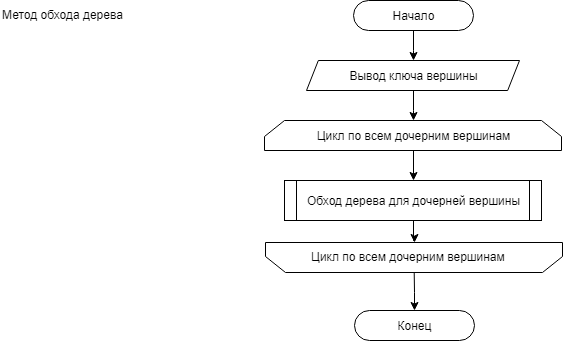
**Методы:**

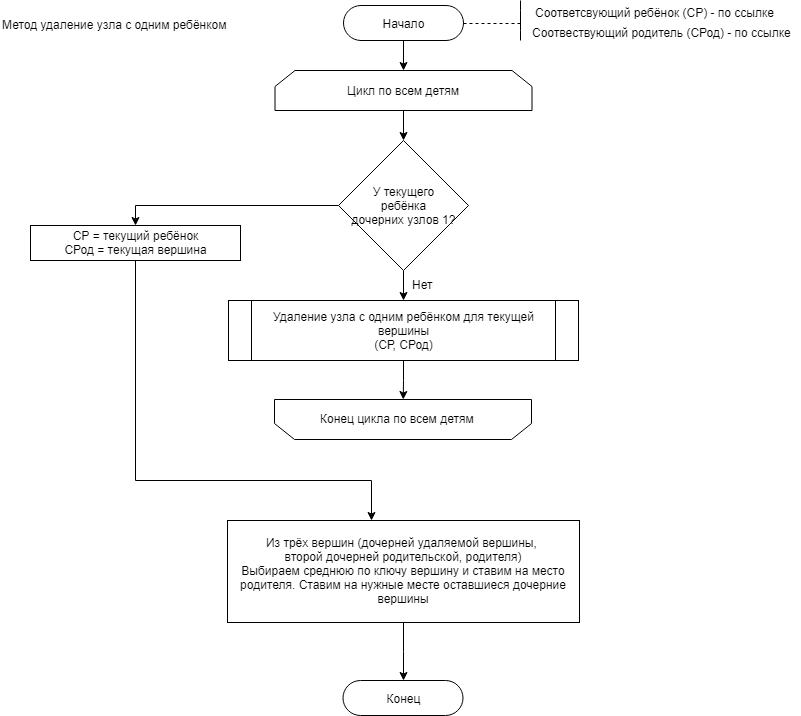
1. Конструктор – инициализация полей данных.
2. Добавление вершины – создаёт новый экземпляр класса Tree, вызывает метод поиска свободной вершины, передавая новый экземпляр, затем на нём вызывается метод добавление новой вершины.
3. Удаление узла с одной дочерней вершиной – производится поиск вершины, у которой один дочерний узел, затем производится его удаление со смещением оставшихся вершин.
4. Обход – производит рекурсивный обход дерева в прямом порядке.
5. Поиск ключа – производит поиск ключа в дереве посредством рекурсивного обхода если текущий ключ не равен искомому по принципу: если текущий ключ < искомого, то вызывается метод на правом узле, иначе на левом.
6. Поиск минимального ключа – производит рекурсивный обход дерева с сравнением ключей с целью поиска минимального.
7. Инициализация дерева – создаёт новый экземпляр класса Tree и заполняет его нужным кол-вом вершин. Заполнение происходит сбалансированно, то есть пока текущий уровень не заполнен, новый уровень также не заполняется.
8. Создание новой вершины – создаёт и заполняет узел случайным числом в диапазонах в зависимости от того, левый узел или правый, учитывая условия бинарного дерева поиска.
9. Создание новой вершины по заданному ключу – находит нужное место для узла и вставляет его на это место.











**Текст программы.**

using System;

using System.Collections.Generic;

namespace Lab\_5

{

class Tree : IComparable<Tree>

{

public const int MAX\_NUMBER\_NODES = 2;

public static int CountNodes { get; private set; }

private int \_currentLvl;

private List<Tree> \_subTrees;

private int \_key;

private Tree()

{

\_subTrees = new List<Tree>();

}

public string AddNode()

{

CountNodes++;

string result = "Добавление случайного узла...\n";

Tree findMatch = null;

SearchNodeWithFreeSpace(ref findMatch);

if (findMatch is null)

return result + "Не удалось создать узел.";

findMatch.CreateNewNode(this);

return result + "Узел успешно добавлен.";

}

public string AddNode(int key)

{

CountNodes++;

string result = "Добавление узла по выбранному ключу...\n";

bool isFind = false;

Tree findNode = null;

SearchNodeWithMatchKey(ref findNode, key, ref isFind);

if (findNode is null)

return result + "Не удалось добавить узел.";

findNode.\_subTrees.Add(new Tree() { \_key = key, \_currentLvl = findNode.\_currentLvl + 1 });

findNode.\_subTrees.Sort();

return result + "Узел успешно добавлен.";

}

public string RemoveNodeWithOneChild()

{

Tree findChildMatch = null;

Tree findParentMatch = null;

SearchParentNodeWithOneChild(ref findChildMatch, ref findParentMatch);

if (findChildMatch is null || findParentMatch is null)

{

return "Удаление узла с одним дочерним узлом невозможно.";

}

else

{

findParentMatch.\_subTrees.Add(findChildMatch.\_subTrees[0]);

findParentMatch.\_subTrees.Remove(findChildMatch);

findParentMatch.\_subTrees.Sort();

List<Tree> subsetTree = new List<Tree>() { findParentMatch, findParentMatch.\_subTrees[0], findParentMatch.\_subTrees[1] };

subsetTree.Sort();

Tree temp = findParentMatch;

findParentMatch = subsetTree[1];

subsetTree[1] = temp;

subsetTree.Remove(findParentMatch);

subsetTree.Sort();

return $"Узел с ключом {findChildMatch.\_key} удалён.";

}

}

public void Display()

{

Console.WriteLine($"Уровень: {\_currentLvl}, ключ: {\_key}");

foreach (Tree subTree in \_subTrees)

subTree.Display();

}

public string SearchKey(int key)

{

Tree findMatch = null;

GetMatchingKey(ref findMatch, key);

if (findMatch is null)

return $"Ключ {key} не найден.";

else

return $"Ключ {key} найден.";

}

public string SearchMinKey()

{

int curMin = \_key;

GetMinKey(ref curMin);

return $"Минмальный ключ = {curMin}.";

}

public static Tree InitializeNewTree(int numberNodes)

{

CountNodes = 0;

// Максимальный уровень узлов дерева.

int maxLvl = (int)(Math.Log(numberNodes, 2));

int rootKey = 10000 \* numberNodes;

Tree root = new Tree() { \_key = rootKey };

List<Tree> parents = new List<Tree>();

List<Tree> children = new List<Tree>();

if (numberNodes > 0)

{

parents.Add(root);

CountNodes++;

}

else

{

Console.WriteLine("Дерево с нулём вершин не может быть создано.");

return null;

}

Console.WriteLine($"Текущий (кол-во вершин = {numberNodes}) максимальный уровень дерева: {maxLvl} (отсчёт с нуля).");

for (int i = 0; i < maxLvl; i++)

{

children.AddRange(parents);

parents.Clear();

foreach (Tree child in children)

{

for (int j = 0; j < MAX\_NUMBER\_NODES; j++)

{

if (CountNodes != numberNodes)

{

CountNodes++;

child.CreateNewNode(root);

parents.Add(child.\_subTrees[j]);

}

}

}

children.Clear();

}

return root;

}

private void SearchNodeWithMatchKey(ref Tree matchingTree, int key, ref bool isFind)

{

if (\_key <= key)

{

if (\_subTrees.Count > 0)

{

if (\_subTrees.Count == 2)

{

\_subTrees[1].SearchNodeWithMatchKey(ref matchingTree, key, ref isFind);

}

else if (\_subTrees[0].\_key >= \_key)

{

\_subTrees[0].SearchNodeWithMatchKey(ref matchingTree, key, ref isFind);

}

else

{

if (!isFind)

{

matchingTree = this;

isFind = true;

return;

}

}

}

else

{

if (!isFind)

{

matchingTree = this;

isFind = true;

return;

}

}

}

else

{

if (\_subTrees.Count == 2 || (\_subTrees.Count == 1 && \_subTrees[0].\_key < \_key))

{

\_subTrees[0].SearchNodeWithMatchKey(ref matchingTree, key, ref isFind);

}

else

{

if (!isFind)

{

matchingTree = this;

isFind = true;

return;

}

}

}

return;

}

private void CreateNewNode(Tree root)

{

Random rnd = new Random();

int newKey = 0;

int maxDelta = 500;

int minDelta = 100;

int maxOnLvl = (int)(Math.Pow(2, \_currentLvl + 1));

double countPrevCurNodes = 0.0;

for (int i = \_currentLvl; i >= 0; i--)

countPrevCurNodes += Math.Pow(2, i);

int numberNodeOnLvl = CountNodes - (int)countPrevCurNodes;

if (\_subTrees.Count == 0)

{

if (numberNodeOnLvl <= maxOnLvl / 2)

{

do

{

maxDelta -= 1;

minDelta += 1;

newKey = rnd.Next(\_key - maxDelta, \_key - minDelta);

} while (!(newKey < \_key && newKey < root.\_key));

}

else

{

do

{

minDelta -= 1;

maxDelta += 1;

newKey = rnd.Next(\_key - maxDelta, \_key - minDelta);

} while (!(newKey < \_key && newKey >= root.\_key));

}

\_subTrees.Add(new Tree() { \_key = newKey, \_currentLvl = \_currentLvl + 1 });

}

else

{

if (numberNodeOnLvl <= maxOnLvl / 2)

{

do

{

maxDelta -= 1;

newKey = rnd.Next(\_key, \_key + maxDelta);

} while (!(newKey >= \_key && newKey < root.\_key));

}

else

{

do

{

maxDelta += 1;

newKey = rnd.Next(\_key, \_key + maxDelta);

} while (!(newKey >= \_key && newKey >= root.\_key));

}

\_subTrees.Add(new Tree() { \_key = newKey, \_currentLvl = \_currentLvl + 1 });

}

}

private void SearchNodeWithFreeSpace(ref Tree matchingNode)

{

List<Tree> parents = new List<Tree>() { this };

List<Tree> children = new List<Tree>();

while (true)

{

for (int i = 0; i < parents.Count; i++)

{

if (parents[i].\_subTrees.Count < MAX\_NUMBER\_NODES)

{

matchingNode = parents[i];

return;

}

else

{

children.Add(parents[i].\_subTrees[0]);

children.Add(parents[i].\_subTrees[1]);

}

}

parents.Clear();

parents.AddRange(children);

children.Clear();

}

}

private void GetMatchingKey(ref Tree matchingNode, int key)

{

if (\_key == key)

{

matchingNode = this;

return;

}

if (\_key < key)

{

if (\_subTrees.Count == 2)

\_subTrees[1].GetMatchingKey(ref matchingNode, key);

}

if (\_subTrees.Count > 0)

\_subTrees[0].GetMatchingKey(ref matchingNode, key);

}

private void GetMinKey(ref int min)

{

foreach (Tree subTree in \_subTrees)

{

if (subTree.\_key < min)

min = subTree.\_key;

subTree.GetMinKey(ref min);

}

}

private void SearchParentNodeWithOneChild(ref Tree matchingNode, ref Tree matchingParentTree)

{

foreach (Tree subTree in \_subTrees)

{

if (subTree.\_subTrees.Count == 1)

{

matchingNode = subTree;

matchingParentTree = this;

return;

}

subTree.SearchParentNodeWithOneChild(ref matchingNode, ref matchingParentTree);

}

}

int IComparable<Tree>.CompareTo(Tree other)

{

return \_key.CompareTo(other.\_key);

}

}

}

**Результаты работы программы.**

