**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**
**УНИВЕРСИТЕТ ИТМО**

**С.Е. Иванов**

# ПРИКЛАДНЫЕ АЛГОРИТМЫ НА ЯЗЫКЕ ООП C#

**Учебно-методическое пособие**

**Санкт-Петербург**
**2024**

Иванов С.Е. **Прикладные алгоритмы на языке ООП C#.** – СПб: Университет ИТМО, 2022. – 60 с.

Пособие включает лабораторные работы, выполняемые в рамках курса «Объектно-ориентированное программирование». Представлено описание, блок-схемы и реализация различных прикладных алгоритмов на языке ООП C#.

Пособие предназначено для бакалавров и содержит материалы лабораторных работ по дисциплине «Объектно-ориентированное программирование».

# Оглавление

# Введение

В методическом пособии представлены задачи для реализации средствами ООП C# различных прикладных алгоритмов, таких как алгоритм хеширования, генетический алгоритм, алгоритмы обхода деревьев, поисковые алгоритмы на графах, нахождения кратчайших путей и минимального остова на графах, метод «разделяй и властвуй», алгоритмы сортировки и поиска. Также рассмотрена задача оценки сложности алгоритмов. При выполнении работ обучающийся приобретает соответствующие компетенции для понимания схем работы алгоритмов, навыки программирования и умения реализовывать алгоритмы средствами ООП.

В предлагаемом пособии представлены лабораторные работы, выполняемые в рамках курса «Объектно-ориентированное программирование» для профессиональных образовательных программ высшего образования бакалавриата по направлению подготовки 09.03.03 Прикладная информатика.

Перед выполнением лабораторных работ обучающемуся необходимо изучить теоретический материал по рассматриваемому курсу.

Пособие содержит восемь лабораторных работ, включающих постановку задачи, состав отчета по работе, пример реализации алгоритма посредством ООП C# и контрольные вопросы. Также при выполнении лабораторной работы восемь обучающийся приобретает навыки оценки сложности алгоритмов.

После выполнения работы обучающийся предоставляет отчет, который защищает в семестре. В ходе защиты обучающийся поясняет свой отчет и отвечает на вопросы преподавателя. В конце пособия представлен список рекомендуемой литературы по рассмотренным темам.

Методические рекомендации для выполнения представленных в пособии лабораторных работ:

1. Ознакомиться с алгоритмом метода.

2. Разработать схему работы алгоритма.

3. Реализовать алгоритм средствами ООП C#.

4. Выполнить расчет примера с помощью разработанной программы.

5. Проверить результаты работы программы на примерах.

Шаблон отчета по лабораторной работе №____

«_____»

(название лабораторной работы)

1. Цель и задачи лабораторной работы:
2. Задание на лабораторную работу:
3. Результаты лабораторной работы:
4. Выводы:

Отчет по лабораторной работе представляется в электронном виде в формате, предусмотренном шаблоном отчета по лабораторной работе. Защита отчета проходит в форме доклада обучающегося с демонстрацией результатов и ответов на вопросы преподавателя.

Если оформление отчета и доклад обучающегося во время защиты соответствуют указанным требованиям, обучающийся получает максимальное количество баллов.

Основаниями для снижения количества баллов являются:

- небрежное выполнение,
- низкое качество оформления,
- замечания при ответе на контрольные вопросы преподавателя.

Отчет не может быть принят и подлежит доработке в случае:

- отсутствия необходимых разделов,
- отсутствия результатов выполнения.

При выполнении работ обучающийся закрепляет теоретические знания и получает практические навыки по реализации алгоритмов.

В результате выполнения работ обучающийся приобретает:

Знания: принципов построения алгоритмов; основных этапов решения задачи на компьютере; основ языка программирования высокого уровня C#; технологий работы с информационными системами, техническими средствами хранения информации, ее кодирование; типовых структур данных и способов их записи средствами языка высокого уровня.

Умения: принимать участие в разработке задания на программный продукт; выбирать способ представления данных, определять спецификации на отдельные классы, модули и подпрограммы; по математическому описанию задачи разрабатывать алгоритм работы программы; выбирать способ организации программы, на языке высокого уровня.

Навыки: выбирать язык программирования высокого уровня для реализации поставленной задачи; анализировать и алгоритмизировать задачи в различных областях знаний; выбирать технологии разработки для решения поставленных задач; выбирать способ представления данных и алгоритм решения задачи.

Для самостоятельной работы обучающимся рекомендуется предоставить в соответствии с планом СРО 91.2 часа в семестре на подготовку и выполнение лабораторных работ.

# Лабораторная работа №1. Алгоритмы хеширования

## ЦЕЛЬ РАБОТЫ

Изучить алгоритмы хеширования, реализовать алгоритм Рабина-Карпа поиска подстроки в строке с применением хеширования.

## ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритмы хеширования применяют хеш-функцию для преобразования входных данных в выходную битовую строку заданной длины. В результате преобразования получается хеш или хеш-код. Хеш-функции применяют для поиска дублирующих данных, вычисления контрольных сумм, создания уникальных идентификаторов, хранения паролей, для электронной подписи и многих других задачах. В соответствии с правилом Дирихле нет строгого однозначного соответствия между хешем и исходными данными, поэтому возникают коллизии. Любая хорошая хеш-функция должна быстро выполняться и иметь минимальное число коллизий. Для разрешения коллизий разработаны различные методы: цепочек, метод открытой адресации. Хеширование применяется в алгоритме поиска подстроки в тексте, разработанном Майклом Рабином и Ричардом Карпом. Алгоритм Рабина-Карпа применяют для поиска плагиата и совпадений шаблонов одинаковой длины. Этот алгоритм основывается на хешировании строк. Дана строка S и текст T, состоящие из маленьких латинских букв. Требуется найти все вхождения строки S в текст T.

Алгоритм состоит из следующих шагов. Посчитаем хеш для строки S. Посчитаем значения хешей для всех префиксов строки T. Далее переберём все подстроки T длины |S| и каждую сравним с |S|.

## ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1.Реализовать поиск одинаковых строк.

Дан список строк S[1..n], каждая длиной не более m символов. Требуется найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки.

2. Реализовать алгоритм Рабина-Карпа поиска подстроки в строке за O (n).

## ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Этапы работы:

1. Реализовать поиск одинаковых строк

Дан список строк S[1..N], каждая длиной не более M символов. Необходимо найти все повторяющиеся строки и разделить их на группы, чтобы в каждой группе были только одинаковые строки.

Ниже представлен листинг для считывания строк

```
static void Main(string[] args)
{
    int m = 50;
    int n = 10;
    string[] stringArray = new string[n];
    for(int i = 0; i < 10; i++)
    {
        stringArray[i] = Console.ReadLine();
    }
```

Полиномиальным хешем этой строки называется число h = hash(s0..n-1) = s0p + p2s1 + p3s2 +… + pnsn-1, где p - некоторое натуральное число, а si - код i-ого символа строки s (он записывается s[i]).

Ниже представлен листинг для вычисления степеней p:

```
int p = 31;
int[] p_pow = new int[m];
p_pow[0] = 1;
for (int i = 1; i < m; i++)
{
    p_pow[i] = p_pow[i - 1] * p;
}
```

Ниже представлен листинг для вычисления хешей строк:

```
var hashs = new Dictionary<int, int>();
for(int i =0; i < n; i++)
{
    hashs[i] = 0;
    for (int j = 0; j < stringArray[i].Length; j++)
    {
        hashs[i] += (stringArray[i][j] - 'a'+1)*p_pow[j];
    }
}
```

Ниже представлен листинг для сортировки по хешам и вывод ответа:

```
hashs = hashs.OrderBy(pair => pair.Value).ToDictionary(pair => pair.Key, pair => pair.Value);
foreach (var item in hashs)
    Console.WriteLine(item.Key + " " + item.Value);

int n_groups = 0;
var previous_hash = hashs.First();
Console.WriteLine(previous_hash);
foreach(var hash in hashs)
{
    if (hash.Equals(hashs.First()) || hash.Value != previous_hash.Value)
    {
        n_groups++;
        Console.WriteLine();
        Console.Write("Group {0}: ",n_groups);
    }
    Console.Write(hash.Key+" ");
    previous_hash = hash;
}
```

Ниже представлен вывод результатов: хеши и список групп:

```
love
love
sun
sun
love
love
dive
wind
dive
dive
```

```
2 14124
3 14124
7 132920
6 170380
8 170380
9 170380
0 170574
1 170574
4 170574
5 170574
[2, 14124]
```

```
Group 1: 2 3
Group 2: 7
Group 3: 6 8 9
Group 4: 0 1 4 5
```

2. Реализовать алгоритм Рабина-Карпа поиска подстроки в строке за O(N).

Дана строка S и текст T, состоящие из маленьких латинских букв. Требуется найти все вхождения строки S в текст T за время O (|S| + |T|).

Сначала так же считаем степени P. Затем хеши подстрок текста и хеш строки.

Ниже представлен листинг для вычисления степеней и хешей:

```
string s = Console.ReadLine();
string t = Console.ReadLine();
int p = 31;
int n = s.Length;
int m = t.Length;
int[] p_pow = new int[m];
p_pow[0] = 1;
for (int i = 1; i < m; i++)
{
    p_pow[i] = p_pow[i - 1] * p;
}
int[] hashs = new int[m];
for(int i = 0; i < m; i++)
{
    hashs[i] = (t[i] - 'a' + 1) * p_pow[i];
    if (i!=0)
    {
        hashs[i] += hashs[i - 1];
    }
}
int s_hash = 0;
for (int i = 0; i < n; i++)
{
    s_hash+= (s[i] - 'a' + 1) * p_pow[i];
}
```

Используя формулу $hash(s_{0..R}) = hash(s_{0..L-1}) + p^L hash(s_{L..R})$, найдем индексы, где в тексте появляется нужная строка.

Ниже представлен листинг для вывода ответа:

```
int cur_h;
for(int i = 0; (i + n-1) <m; i++)
{
    cur_h = hashs[i + n - 1];
    if (i != 0)
    {
        cur_h -= hashs[i - 1];
    }
    if (cur_h == s_hash * p_pow[i])
    {
        Console.WriteLine(i);
```

Ниже представлены результаты поиска строки в тексте:

```
hi
hinnnnhinnhi hi
0
6
10
13
```

Выводы:

В ходе выполнения работы изучены и реализованы средствами ООП C# алгоритмы хеширования.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие задачи решаются с помощью алгоритма хеширования?
2. Как решается проблема коллизий при хешировании?
3. Какие ограничения и условия на применение алгоритма хеширования?
4. Какие конструкции ООП C# применяются для реализации алгоритма хеширования?
5. Как оценивается сложность алгоритма хеширования?
6. Как можно оценить точность алгоритма хеширования?

# Лабораторная работа №2. Алгоритм генетический

## ЦЕЛЬ РАБОТЫ

Изучить генетический алгоритм и реализовать алгоритм средствами ООП для решения Диофантова уравнения**.**

## ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Генетический алгоритм аналогичен процессу естественного отбора в природе и широко применяется в оптимизационных задачах. В алгоритме важную роль играют операторы кроссинговера, наследования, мутаций и селекции, которые подбираются под определенную задачу. Решение задачи кодируется вектором генов или генотипом. В алгоритме оценивается множество решений (поколение) посредством функции приспособленности или целевой функции в задаче оптимизации. Алгоритм итерационно повторяется до выполнения критерия остановки – нахождения оптимума, прохождения числа поколений или времени.

Алгоритм состоит из следующих шагов: задания функции приспособленности, генерации начального множества решений, в цикле пока критерий остановки не выполнен выполняются операторы кроссинговера, наследования, мутаций, селекции, для сознания нового множества решений. Алгоритм широко применяется для задач оптимизации.

Ниже представлена блок схема генетического алгоритма.

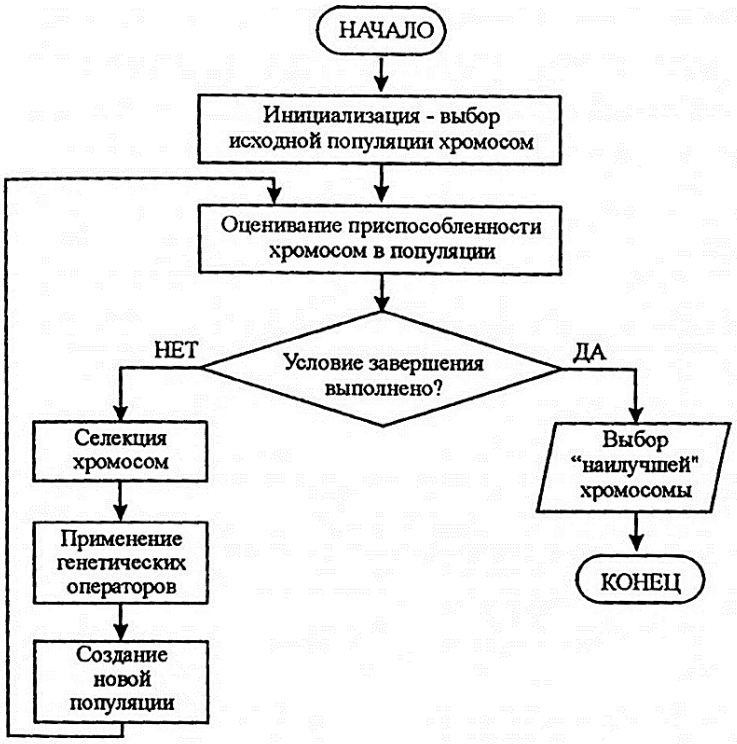



Рисунок 1 – Блок-схема генетического алгоритма

## ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Реализовать средствами ООП генетический алгоритм для решения задачи: найти решение Диофантова (только целые решения) уравнения: a+2b+3c+4d=30, где a, b, c и d - некоторые положительные целые.

## ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

В данном случае посредством генетического алгоритма реализуется решение Диофантова уравнения. Дано уравнение с целыми коэффициентами, определить целое решение.

Диофантово уравнение имеет вид:

D(a, b, c, d) = a + 2b + 3c + 4d = 30,

где a, b, c и d – некоторые положительные целые.

Должно выполняться условие 1 <= a, b, c, d <= 30.

Создается класс Generation, объектами которого будут поколения, а переменные в уравнении – поля класса.

Ex_res – ожидаемый результат, real_res – получаемый.

Ниже представлен листинг, показывающий поля и конструктор Поколения:

```
class Generation
{
    public int a;
    public int b;
    public int c;
    public int d;
    public static Random r = new Random();
    public int ex_res;
    public Func<int, int, int, int, int> f;
    private int? real_res;
    Ссылок: 2
    public int RealResult
    {
        get
        {
            if (real_res == null)
            {
                real_res = f(a, b, c, d);
            }
            return (int)real_res;
        }
    }
    Ссылок: 4
    public Generation(Func<int, int, int, int, int> f, int a, int b, int c, int d, int ex_res)
    {
        this.f = f;
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
        this.ex_res = ex_res;
    }
```

Далее реализуется механизм наследования – в метод передаются два представителя поколения, после чего случайным образом определяется то, как их «ребенок» (новый объект Generation) унаследует хромосомы.

Ниже представлен листинг, реализующий наследование:

```
public static Generation NewGen(Generation p1, Generation p2)
{
    int x = r.Next(1, 4);
    switch (x)
    {
        case 1: return new Generation(p1.f, p1.a, p1.b, p2.c, p2.d, p2.ex_res);
        case 2: return new Generation(p1.f, p1.a, p2.b, p2.c, p2.d, p2.ex_res);
        case 3: return new Generation(p1.f, p1.a, p1.b, p1.c, p2.d, p2.ex_res);
        default: return null;
    }
}
```

Реализуется метод мутаций. Какая хромосома мутирует, определяется также случайно.

Ниже представлен листинг, реализующий мутацию:

```
public void Change()
{
    int imposter = r.Next(1, 5);
    switch (imposter)
    {
        case 1:
            a = r.Next(1, 30);
            break;
        case 2:
            b = r.Next(1, 30);
            break;
        case 3:
            c = r.Next(1, 30);
            break;
        case 4:
            d = r.Next(1, 30);
            break;
        default:
            break;
    }
}
```

Также реализуется метод GetAc() для вычисления, насколько реальный результат отклонился от ожидаемого. Далее с помощью GetAc() в методе SrKoef() определяется средний коэффициент выживания поколения.

Ниже представлен листинг для расчета коэффициента выживания:

```
public int GetAc()
{
    return Math.Abs((int)(ex_res - RealResult));
}
Ссылок: 2
public static double SrKoef(List<Generation> generations)
{
    double sum = 0;
    foreach (Generation item in generations)
    {
        sum += item.GetAc();
    }
    return sum / generations.Count();
}
```

В методе Main создается поколение, в которое добавляется 5 объектов, после чего происходит поиск решения в цикле. Каждая итерация поколения выводится на экран.

Ниже представлен листинг метода Main:

```
class Program
{
    public static int res = 30;
    public static Func<int, int, int, int, int> f = (a, b, c, d) => a + 2 * b + 3 * c + 4 * d;
    public static Random r = new Random();
    Ссылок: 0
    static void Main(string[] args)
    {
        List<Generation> generations = new List<Generation>() ;
        for (int i = 0; i < 5; i++)
        {
            generations.Add(new Generation(f, r.Next(1, 30), r.Next(1, 30), r.Next(1, 30), r.Next(1, 30), res));
        }
        for (int i = 0; ; i++)
        {
            Console.WriteLine("Итерация " + (i + 1));
            foreach (var item in generations)
            {
                Console.Write("a = "+item.a+ " b = " + item.b+ " c = " + item.c+ " d = " + item.d+"\n");
            }
            Console.WriteLine("---");
```

Посредством сравнения коэффициентов выживания, поколения добавляются в список best с последующей проверкой соответствия необходимому результату. Ниже представлен листинг для продолжения Main:

```
double koef1 = Generation.SrKoef(generations);
var best = generations
    .OrderBy(g => g.GetAc())
    .ToList();
if (best.Any(g => g.RealResult == res))
{
    Solution(best[0]);
    break;
}

generations.Clear();
generations.Add(Generation.NewGen(best[0], best[1]));
generations.Add(Generation.NewGen(best[1], best[0]));
generations.Add(Generation.NewGen(best[0], best[2]));
generations.Add(Generation.NewGen(best[2], best[0]));
generations.Add(Generation.NewGen(best[1], best[2]));
double koef2 = Generation.SrKoef(generations);
if (koef2 <= koef1)
{
    Random ran = new Random();
    int x = ran.Next(1, 5);
    for (int j = 0; j < x; j++)
    {
        generations[ran.Next(0, 5)].Change();
    }
}
```

Ниже представлен листинг для вывода решения:

```
public static void Solution(Generation g)
{
    Console.WriteLine("Решение: ");
    Console.Write("a = " + g.a + " b = " + g.b + " c = " + g.c + " d = " + g.d + "\n");
}
```

Ниже представлены результаты расчета примера:

```
Итерация 152
a = 2 b = 10 c = 5 d = 1
a = 2 b = 17 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
a = 18 b = 10 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
---
Итерация 153
a = 2 b = 10 c = 2 d = 1
a = 2 b = 9 c = 2 d = 1
a = 2 b = 10 c = 5 d = 19
a = 2 b = 6 c = 2 d = 1
a = 2 b = 10 c = 5 d = 1
---
Итерация 154
a = 2 b = 10 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
a = 2 b = 10 c = 2 d = 1
a = 6 b = 6 c = 2 d = 1
a = 10 b = 9 c = 2 d = 1
---
Решение:
a = 10 b = 9 c = 2 d = 1
```

Выводы:

В ходе выполнения работы реализован генетический алгоритм на языке C#, позволяющий решить Диофантово уравнение, выбрав наилучшее решение.

**КОНТРОЛЬНЫЕ ВОПРОСЫ**

1. Какие конструкции ООП C# применяются для реализации генетического алгоритма?
2. Как оценивается сложность генетического алгоритма?
3. Какие задачи решаются с помощью генетического алгоритма?
4. Какие ограничения и условия на применение алгоритма?
5. Какие определяются операторы кроссинговера, наследования, мутаций и селекции?
6. Как можно оценить точность генетического алгоритма?

# Лабораторная работа №3. Алгоритмы с бинарными деревьями

## ЦЕЛЬ РАБОТЫ

Изучить алгоритмы работы с деревьями. Реализовать средствами ООП дерево и рекурсивные функции, выполняющие обход дерева в прямом, обратном, концевом порядке. Вычислить значение выражения, заданного деревом.

## ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритмы поиска по дереву или обхода дерева широко применяются к двоичным деревьям. Различают алгоритмы обхода дерева в глубину и ширину. Для обхода в глубину применяют прямой порядок, центрированный и обратный.

Для поиска в глубину двоичного дерева обработка вершин выполняется рекурсивно в различном порядке: вначале рекурсивный обход левого поддерева (L), потом правого (R), далее просматривается корень (N). Представленный порядок обхода слева – направо можно менять: прямой (NLR), центрированный (LNR), обратный обход (LRN). Центрированный обход двоичного дерева получает данные в отсортированном порядке.

## ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализовать средствами ООП дерево и рекурсивные функции, выполняющие обход дерева в прямом, обратном, концевом порядке. Выполнить расчет примера.

   Обход дерева в прямом порядке a b d e c f.

   Обход дерева в обратном порядке d b e a c f.

   Обход дерева в концевом порядке d e b f c a.

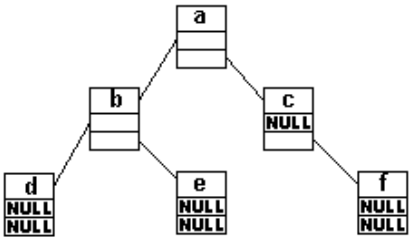



Рисунок 2 – Пример дерева

2. Вычислить значение выражения, заданного деревом. Вычисление выполнять в порядке концевого обхода.

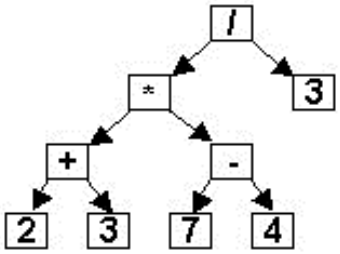



Рисунок 3 – Пример выражения в дереве

## ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

Реализуется класс Tree – узел дерева. Его полями являются левый и правый узлы-потомки, значение, хранимое в узле, булева переменная flag (для последующего использования в функции добавления узла и сообщающая о том, что узел добавлен в нужное место), список значений узлов для обхода и отдельно корень дерева.

Ниже представлен листинг для класса Tree:

```
public class Tree
{
    public Tree Left;
    public Tree Right;
    public char Value;
    public bool flag;
    public static List<char> list = new List<char>();
    public static Tree Node;
    ссылка: 1
    public Tree()
    {
        Node = null;
    }
    Ссылок: 3
    public Tree(Tree left, Tree right, char val)
    {
        Value = val;
        Left = left;
        Right = right;
    }
```

В классе также реализуется метод Add, добавляющий узлы таким образом, что, пока не встретится значение узла ‘.’, добавление будет происходить слева, и только потом – справа. Когда и там, и там дошли до точек, поднимаемся на уровень выше и так до конца.

Ниже представлен листинг для добавления узлов:

```
public void Add(ref Tree node, char value)
{
    if (node == null)
    {
        node = new Tree(null, null, value);
        flag = false;
    }
    else if (node.Value != '.' && flag == false)
    {
        if (node.Left != null)
        {
            Add(ref node.Left, value);
            if (flag == false)
            {
                if (node.Right != null)
                {
                    Add(ref node.Right, value);
                }
                else
                {
                    node.Right = new Tree(null, null, value);
                    flag = true;
                }
            }
        }
        else
        {
            node.Left = new Tree(null, null, value);
            flag = true;
        }
    }
}
```

В методе Main заполнение дерева выглядит следующим образом.

Ниже представлен листинг для заполнения дерева:

```
Tree t = new Tree();
List<char> chars = new List<char>() { 'a', 'b', 'd', '.','.','e','.','.','c','.','f','.','.'};
for (int i = 0; i < chars.Count; i++)
{
    t.flag = false;
    t.Add(ref Tree.Node, chars[i]);
}
```

Дополнительный метод PrintTree служит для более наглядного убеждения в том, что дерево построено верно.

Ниже представлен листинг для метода PrintTree:

```
public void PrintTree(Tree node)
{
    if (node == null) return;
    if (node.Value != '.')
    {
        Console.WriteLine("значение " + node.Value + " левый " + node.Left.Value + " правый " + node.Right.Value);
    }
    PrintTree(node.Left);
    PrintTree(node.Right);
}
```

Для реализации прямого обхода, а именно: корень – левое поддерево – правое поддерево создан метод TreeWalk_Pr, работающий рекурсивно. Значения в нужном порядке добавляются в список list.

Ниже представлен листинг для прямого обхода:

```
public static void TreeWalk_Pr(Tree node)
{
    if (node.Value != '.')
    {
        list.Add(node.Value);
        TreeWalk_Pr(node.Left);
        TreeWalk_Pr(node.Right);
    }
}
```

TreeWalk_Obr реализует обход: левое поддерево – корень – правое поддерево.

Ниже представлен листинг для обратного обхода:

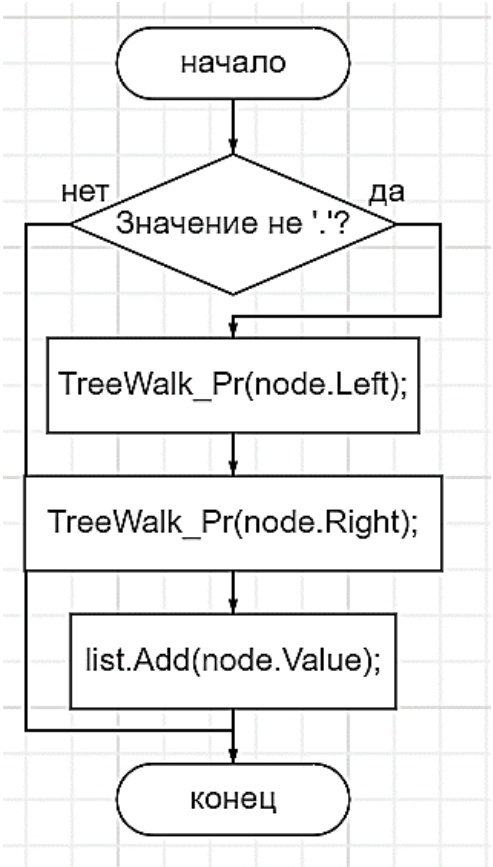
```
public static List<char> TreeWalk_Obr(Tree node)
{
    if (node.Value == '.') return new List<char>();
    var result = TreeWalk_Obr(node.Left);
    result.Add(node.Value);
    result.AddRange(TreeWalk_Obr(node.Right));
    return result;
}
```

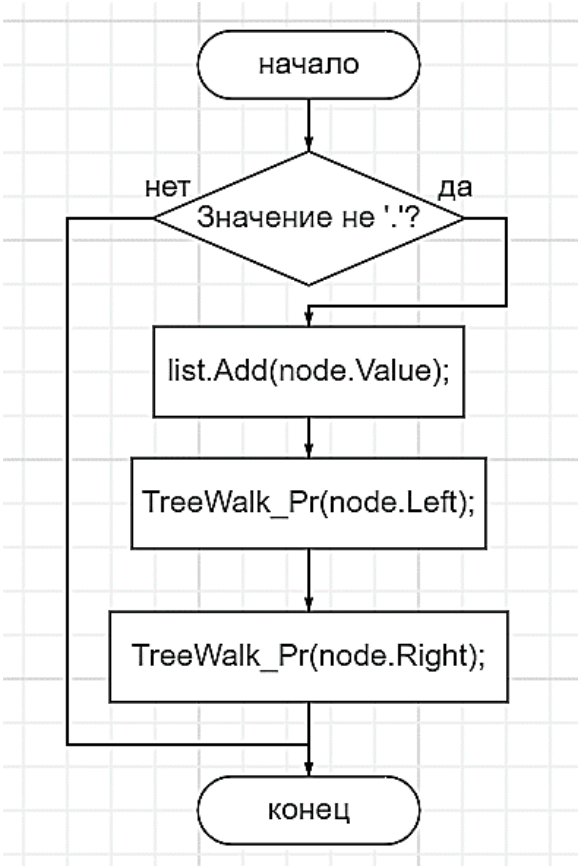
Обход в концевом порядке осуществляет метод TreeWalk_K.

Ниже представлен листинг для концевого обхода: левое поддерево – правое поддерево – корень.

```
public static void TreeWalk_K(Tree node)
{
    if (node.Value != '.')
    {
        TreeWalk_K(node.Left);
        TreeWalk_K(node.Right);
        list.Add(node.Value);
    }
```

Ниже представлены блок-схемы прямого, обратного и концевого обходов.



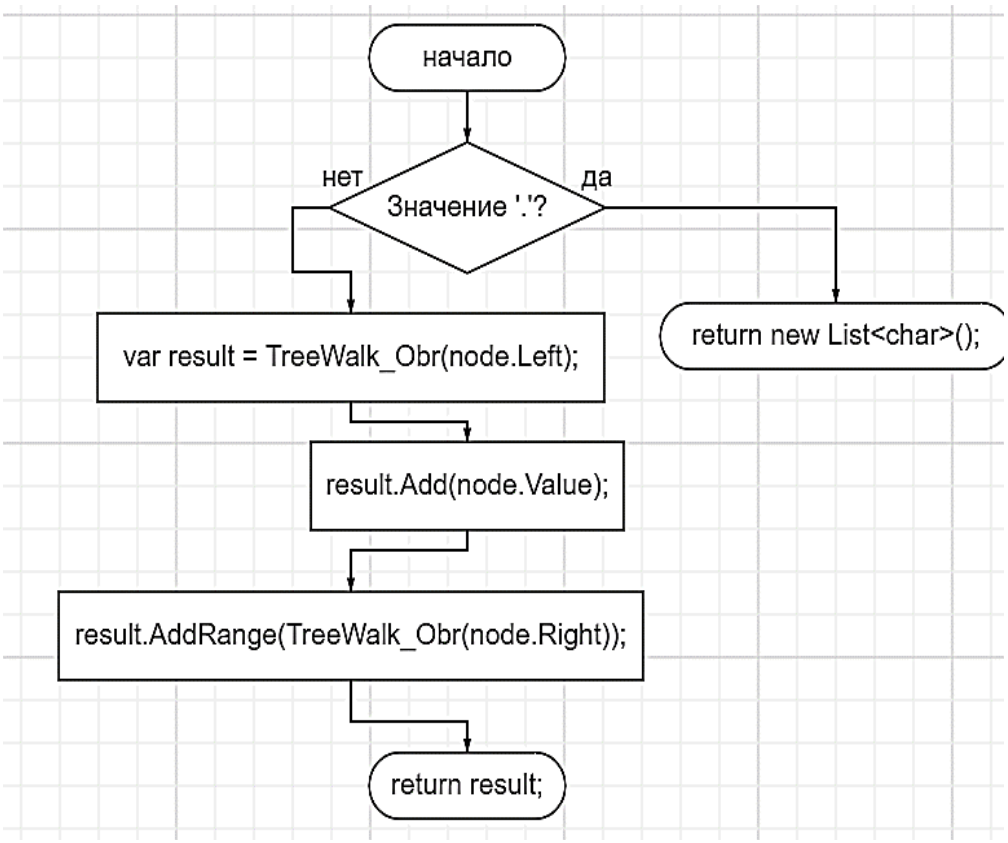



Рисунок 4 – Блок-схемы прямого, обратного и концевого обходов

Ниже представлен листинг для вызова методов:

```
t.PrintTree(Tree.Node);
Console.WriteLine("Прямой порядок");
Tree.TreeWalk_Pr(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item + " ");
}
Tree.list.Clear();
Console.WriteLine() ;
Console.WriteLine("Обратный порядок");
Tree.list = Tree.TreeWalk_Obr(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item+" ");
}
Tree.list.Clear();
Console.WriteLine();
Console.WriteLine("Концевой порядок");
Tree.TreeWalk_K(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item + " ");
}
```

Ниже представлен результат расчета примера:

```
значение a левый b правый c
значение b левый d правый e
значение d левый . правый .
значение e левый . правый .
значение c левый . правый f
значение f левый . правый .
Прямой порядок
a b d e c f
Обратный порядок
d b e a c f
Концевой порядок
d e b f c a
```

Далее реализуется расчет заданного деревом выражения. Сначала дерево обходится в концевом порядке и создается упорядоченный список значений узлов. Далее вызывается метод CalcTree.

Ниже представлен листинг для вычисления выражения:

```
public static int CalcTree(Tree node)
{
    if (char.IsDigit(node.Value)) return node.Value;
    int num1, num2;
    num1 = CalcTree(node.Left);
    num2 = CalcTree(node.Right);
    switch (node.Value)
    {
        case '+': return num1 + num2;
        case '-': return num1 - num2;
        case '*': return num1 * num2;
        case '/': return num1 / num2;
        default: return 0;
    }
}
```

Если рассматриваемое значение узла – число, то оно возвращается. Для символов-операций рекурсивно вычисляются операнды-поддеревья, после чего возвращается необходимый для данной операции результат.

Ниже представлен листинг для метода Main:

```
Tree t = new Tree();
List<char> chars = new List<char>() { '/', '*', '+', '2', '.', '.', '3', '.', '.', '-', '7',
for (int i = 0; i < chars.Count; i++)
{
    t.flag = false;
    t.Add(ref Tree.Node, chars[i]);
}
Console.WriteLine("Концевой порядок");
Tree.TreeWalk_K(Tree.Node);
foreach (var item in Tree.list)
{
    Console.Write(item + " ");
}
Console.Write("Выражение равно ");
Console.Write(Tree.CalcTree(Tree.Node));
```

Ниже представлен результат расчет примера:

```
Концевой порядок
2 3 + 7 4 - * 3 / Выражение равно 5
```

Ниже представлена блок-схема метода вычисления выражения по дереву:

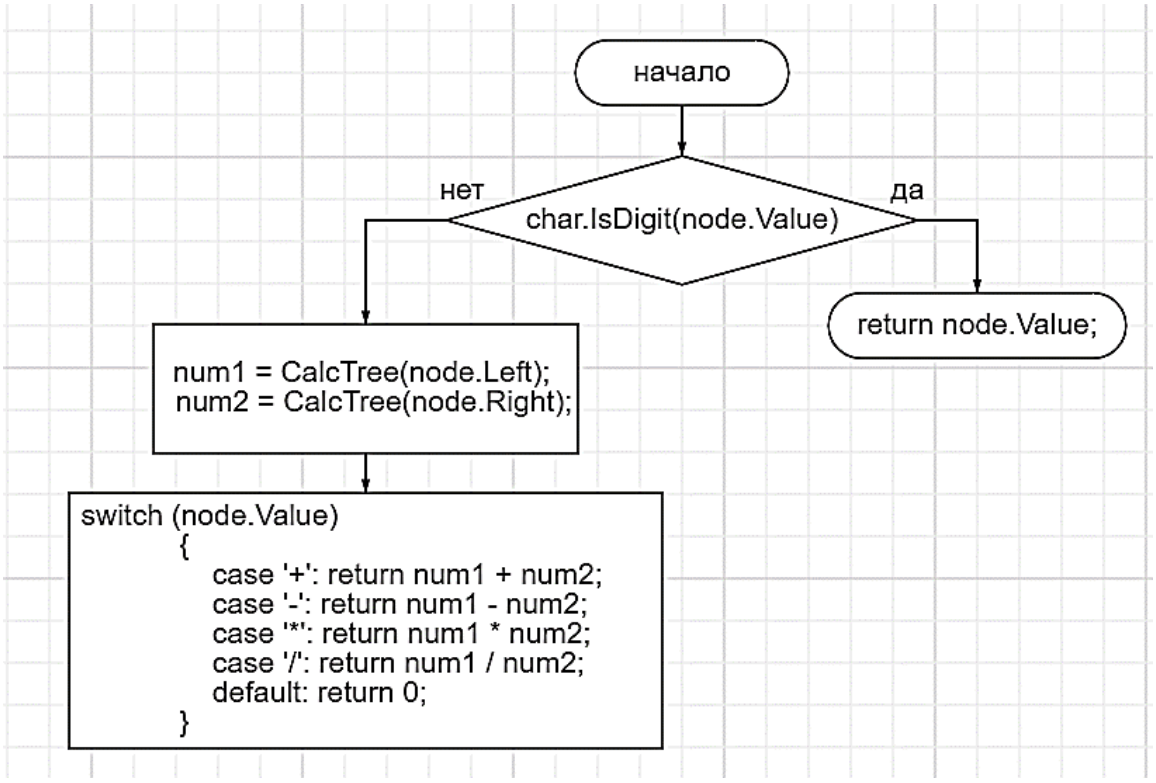



Рисунок 5 – Блок-схема метода вычисления выражения по дереву

Выводы:

В ходе выполнения работы изучены и реализованы такие алгоритмы работы с деревьями, как прямой, обратный и концевой обход, а также алгоритм расчета выражения, заданного деревом.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие ограничения и условия на применения алгоритма обхода?
2. К какому классу сложности относится алгоритм обхода?
3. Какие конструкции ООП C# применяются для реализации прямого, обратного и концевого обхода?
4. Как оценивается сложность алгоритма обхода?
5. Какие задачи решаются с помощью алгоритма обхода?
6. Как можно оценить точность алгоритма обхода?
7. Какой обход дерева применяется для вычисления выражения?

# Лабораторная работа №4. Алгоритмы на графах

## ЦЕЛЬ РАБОТЫ

Научиться реализовывать алгоритмы на графах. Реализовать средствами ООП поиск в глубину и ширину на графе. Реализовать средствами ООП алгоритм нахождение кратчайших путей (Алгоритм Дейкстры). Реализовать алгоритм Крускала.

## ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритмы поиска кратчайшего пути на графе выполняют минимизацию суммы весов рёбер всего пути. Данный алгоритм имеет прикладное значение в GPS-навигаторах.

Алгоритм Дейкстры определяет кратчайшие пути от одной из вершин графа до всех остальных и работает только с положительными весами ребер. Алгоритм широко применяется в протоколах маршрутизации OSPF и IS-IS.
Алгоритм Дейкстры завершается, когда все вершины посещены. До тех пор выбирается вершина с минимальной меткой и рассматриваются всевозможные маршруты, для которых выбранная вершина будет предпоследним пунктом. Для каждого не посещённого соседа вершины определяется новая длина пути как сумма значений текущей метки и длины ребра, соединяющего вершину с этим соседом. Значение метки заменяется значением новой длины, если оно меньше. После рассмотрения вершина отмечается как посещенная и шаги алгоритма повторяются.

Для построения минимального остовного дерева неориентированного графа применяют алгоритм Крускала. При построении остовного дерева минимального веса сначала удаляются все ребра и добавляются ребра с минимальным весом при условии не образования замкнутого цикла.
Ниже представлена блок-схема алгоритма Дейкстры:

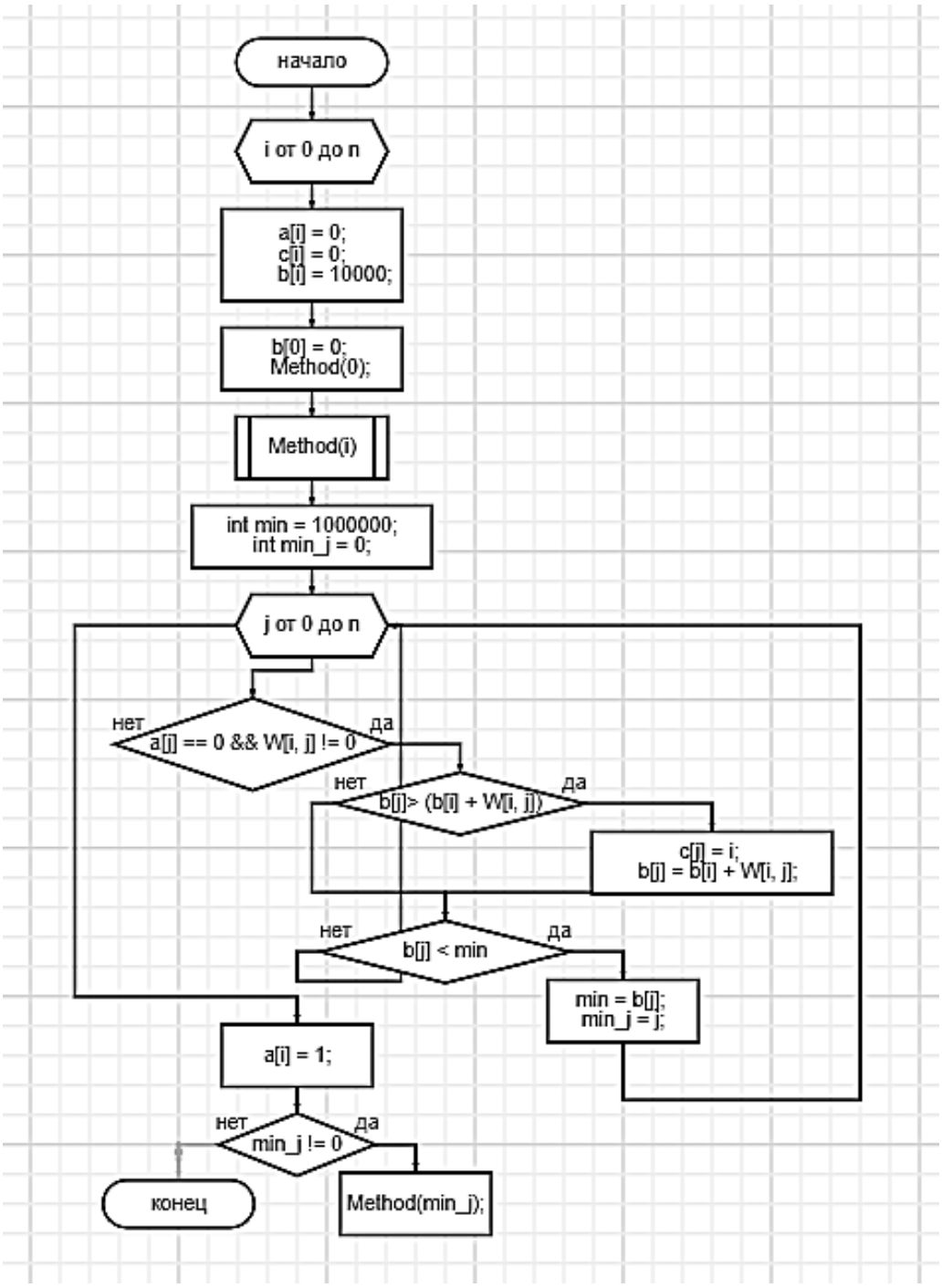
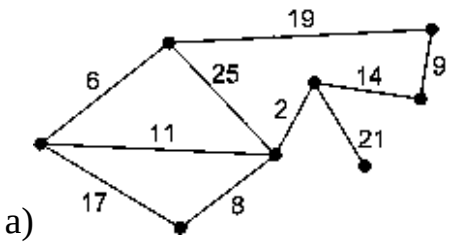



Рисунок 6 – Блок-схема алгоритма Дейкстры

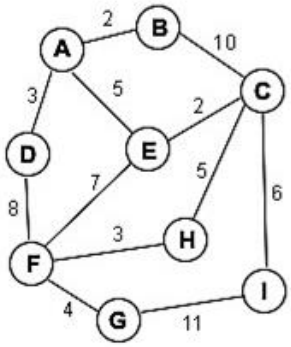
## ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

1. Реализовать средствами ООП поиск в глубину и ширину на графе. Выполнить расчет примера.
2. Реализовать средствами ООП алгоритм нахождения кратчайших путей из одного источника (Алгоритм Дейкстры). Выполнить расчет примера.

3. Найти минимальный остов неориентированного взвешенного графа (Алгоритм Крускала), представленного ниже.




a)




b)

Рисунок 7 – Неориентированный взвешенный графа для алгоритма Крускала.

## ПРИМЕР ВЫПОЛНЕНИЯ РАБОТЫ

1.Реализуется поиск в глубину и ширину на графе.

Идея поиска в глубину заключается в том, что мы двигаемся от начальной вершины (точки, места) в определенном направлении (по определенному пути) до тех пор, пока не достигнем конца пути или пункта назначения (искомой вершины). Если мы достигли конца пути, но он не является пунктом назначения, то мы возвращаемся назад (к точке разветвления или расхождения путей) и идем по другому маршруту.

Создается программа, реализующая данный алгоритм. В полях класса n – число вершин графа, список used – список пройденных вершин, массив массивов g – матрица связей между вершинами.

Ниже представлен листинг для полей класса:

```
class Program
{
    static Random rand = new Random();
    static int n = 6;
    static List<int> used = new List<int>();
    static int[][] g = new int[6][];
```

В методе Main заполняется матрица связей. Алгоритм организован так, что между элементами g[i][j] и g[j][i] будет одинаковая связь 0 или 1 – то есть по условию граф является неориентированным.

Ниже представлен листинг создания матрицы связей:

```
for (int i = 0; i < n; i++)
{
    g[i] = new int[n];
}
for (int i = 0; i < n; i++)
{
    Console.Write("\n(" + (i) + ") вершина -->[");
    for (int j = i; j < n; j++)
    {
        g[i][j] = rand.Next(0, 2);
        g[j][i] = g[i][j];
    }
    foreach (var item in g[i])
    {
        Console.Write(item);
    }
    Console.Write("]\n");
}
```

Далее в методе Main реализуется вызов метода Method, который и реализует поиск в глубину. Принимая номер вершины, он начинает обходить смежные с ней вершины. Для каждой из них сразу же вызывается свой Method, чтобы поиск был именно в глубину. Когда все вершины пройдены, на экран выводится порядок их прохождения – список used.

Ниже представлен листинг для метода обхода в глубину:

```
Method(0);
Console.WriteLine("Вершины были пройдены в следующем порядке:");
foreach (var item in used)
{
    Console.Write(item + " ");
}
```

```
public static void Method(int i)
{
    used.Add(i);
    for (int j = 0; j < n; j++)
    {
        if (g[i][j] == 1 && !used.Contains(j))
        {
            Method(j);
        }
    }
}
```

Для алгоритма обхода в ширину в Method происходит добавление каждой вершины, смежной с данной, в очередь. Вызывается Method последовательно для вершин в очереди. Вершина, для которой подошла очередь, из очереди удаляется. Также осуществляется вывод на экран текущего вида очереди.

Ниже представлен листинг для метода обхода в ширину:

```
public static void Method(int i)
{
    used.Add(i);
    for (int j = 0; j < g[i].Length; j++)
    {
        if (g[i][j] == 1 && !used.Contains(j) && !queue.Contains(j))
        {
            queue.Enqueue(j);
        }
    }
    foreach (var item in queue)
    {
        Console.Write(item+" ");
    }
    Console.WriteLine();

    if (queue.Count != 0)
    {
        Method(queue.Dequeue());
    }
}
```

Ниже представлена блок-схема метода обхода в глубину и в ширину:

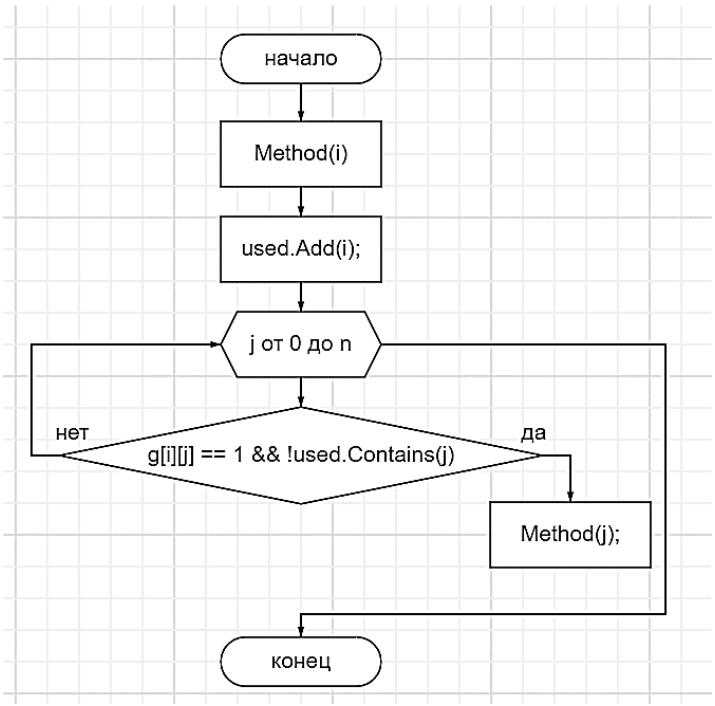



Рисунок 8 – Блок-схема метода обхода в глубину

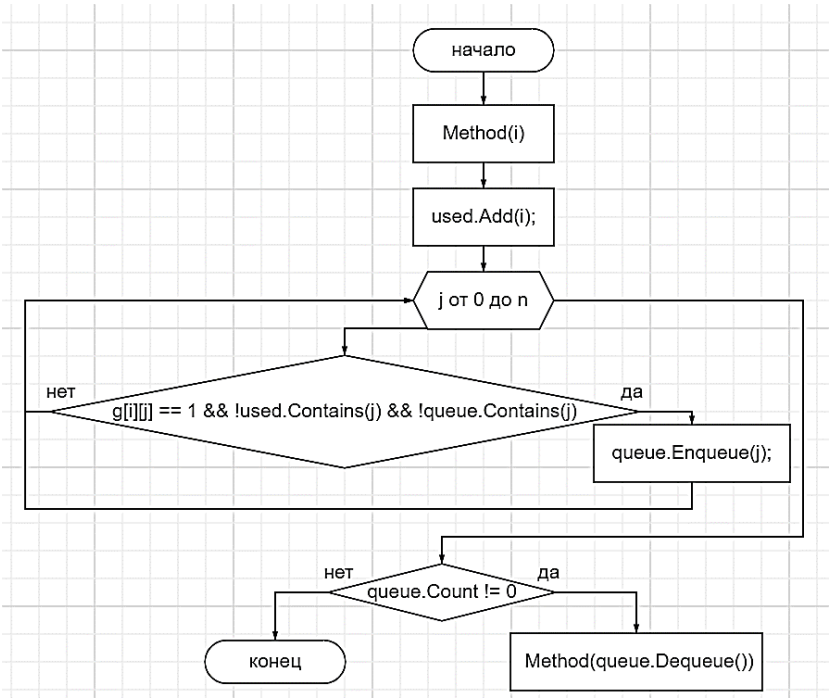



Рисунок 9 – Блок-схема метода обхода в ширину

Ниже представлен результат расчета примера алгоритма обхода в глубину:

```
(0) вершина -->[100111]

(1) вершина -->[011100]

(2) вершина -->[010101]

(3) вершина -->[111100]

(4) вершина -->[100011]

(5) вершина -->[101011]
Вершины были пройдены в следующем порядке:
0 3 1 2 5 4
```

Ниже представлен результат расчета примера алгоритма обхода в ширину:

```
(0) вершина -->[000101]

(1) вершина -->[011111]

(2) вершина -->[010101]

(3) вершина -->[111000]

(4) вершина -->[010000]

(5) вершина -->[111000]
Вид очереди:
3 5
5 1 2
1 2
2 4
4

Вершины были пройдены в следующем порядке:
0 3 5 1 2 4
```

2. Реализуется алгоритм нахождения кратчайших путей из одного источника (Алгоритм Дейкстры).

Для данного алгоритма создается три массива:

1) массив a, такой что a[i]=1, если вершина уже рассмотрена, и a[i]=0, если нет.

2) массив b, такой что b[i] – длина текущего кратчайшего пути из заданной вершины x в вершину i;

3) массив c, такой что c[i] – номер вершины, из которой нужно идти в вершину i в текущем кратчайшем пути.

W – массив массивов, представляющий собой матрицу связей, заполняется вручную.

Сам алгоритм реализует метод Method.

Ниже представлен листинг для метода Main:

```
for (int i = 0; i < n; i++)
{
    a[i] = 0;
    c[i] = 0;
    b[i] = 10000;
}
b[0] = 0;
Method(0);
for (int i = 1; i < n; i++)
{
    Console.WriteLine("Путь до вершины "+i+" от вершины 0:");
    Console.Write(i);
    int z = i;
    do
    {
        Console.Write(" <- " + c[z]);
        z = c[z];
    }
    while (z != 0);
    Console.WriteLine();
}
```

Ниже представлен листинг для алгоритма Дейкстры:

```
public static void Method(int i)
{
    int min = 1000000;
    int min_j = 0;
    for (int j = 0; j < n; j++)
    {
        if (a[j] == 0 && W[i, j] != 0)
        {
            if (b[j]> (b[i] + W[i, j]))
            {
                c[j] = i;
                b[j] = b[i] + W[i, j];
            }
            if (b[j] < min)
            {
                min = b[j];
                min_j = j;
            }
        }
    }
    a[i] = 1;
    if (min_j != 0)
    {
        Method(min_j);
    }
```

Ниже представлен результат расчет примера алгоритма Дейкстры:

```
Путь до вершины 1 от вершины 0:
1 <- 0
Путь до вершины 2 от вершины 0:
2 <- 0
Путь до вершины 3 от вершины 0:
3 <- 2 <- 0
Путь до вершины 4 от вершины 0:
4 <- 5 <- 2 <- 0
Путь до вершины 5 от вершины 0:
5 <- 2 <- 0
```

3. Реализация алгоритма нахождения минимального остова неориентированного взвешенного графа (Алгоритм Крускала).

Заполняется матрица связей, вместе с этим создаются объекты класса Edge – ребра между вершинами, которые добавляется в список ребер.

Ниже представлен листинг для создания матрицы связи и ребер:

```
for (int i = 0; i < n; i++)
{
    Console.Write("\n(" + (i) + ") вершина -->[");
    for (int j = i; j < n; j++)
    {
        {
            g[i][j] = rand.Next(0, 2);
            if (g[i][j] == 1)
            {
                g[i][j] = rand.Next(1, 21);
                Edge edge = new Edge(i, j, g[i][j]);
                Edge.reb.Add(edge);
            }
            g[j][i] = g[i][j];
        }
    }
    foreach (var item in g[i])
    {
        Console.Write(item +"; ");
    }
    Console.Write("]\n");
}
```

Ниже представлен листинг для класса Edge:

```
public class Edge
{
    Ссылок: 8
    public int I { get; set; }
    Ссылок: 8
    public int J { get; set; }
    Ссылок: 8
    public int Weight { get; set; }
    public static List<Edge> reb = new List<Edge>();
    public static List<HashSet<int>> toplist = new List<HashSet<int>>();
    ссылка: 1
    public Edge(int i, int j, int weight)
    {
        I = i;
        J = j;
        Weight = weight;
    }
```

Класс ребер содержит следующие свойства: первая вершина, вторая и вес. Также имеется список ребер и список подмножеств графа, не соединенных друг с другом.

В методе Main далее ребра сортируются по весу, после чего последовательно для каждого ребра вызывается метод Edge_Check.

Ниже представлен листинг для сортировки ребер, вызова метода, вывода результатов:

```
Console.WriteLine("Отсортированный список ребер:");
Edge.reb.Sort((a, b) => a.Weight.CompareTo(b.Weight));
foreach (var item in Edge.reb)
{
    Console.Write(item.Weight + " ");
}
foreach (var item in Edge.reb)
{
    item.Edge_Check();
}
Console.WriteLine();
foreach (var item in Edge.reb)
{
    if (item.Weight != -1)
    {
        Console.WriteLine("ребро между вершинами "+item.I+" и "+item.J+" вес"+item.Weight);
    }
}
```

Метод Edge_Check() работает следующим образом: Последовательно в цикле for проходятся все подмножества графа с целью поиска в них вершин данного ребра. Если найдена вершина I или J, устанавливаются true значения булевых

переменных flag_i или flag_j соответственно. Однако подмножество не может содержать в себе сразу две вершины – в таком случае значение веса ребра станет равным -1, то есть его не будет в остове.

Переменная n_top хранит в себе номер подмножества, в котором находится первая найденная вершина, чтобы, в случае нахождения второй вершины в другом подмножестве, они могли объединиться в одно и цикл бы прекратился. Если цикл закончен, а найдена только одна вершина – вторая добавляется в подмножество первой. Если ни одна не найдена – создается новое подмножество.

Ниже представлен листинг для метода Edge_Check:

```
public void Edge_Check()
{
    bool flag_i = false;
    bool flag_j = false;
    int n_top = -1;
    for (int n = 0; n < toplist.Count;n++)
    {
        if (toplist[n].Contains(this.I) && !toplist[n].Contains(this.J))
        {
            flag_i = true;
            if (n_top != -1)
            {
                toplist[n].UnionWith(toplist[n_top]);
                toplist[n_top].Clear();
                break;
            }
            else
            {
                n_top = n;
            }
        }
```

```
else if (!toplist[n].Contains(this.I) && toplist[n].Contains(this.J))
{
    flag_j = true;
    if (n_top != -1)
    {
        toplist[n].UnionWith(toplist[n_top]);
        toplist[n_top].Clear();
        break;
    }
    else
    {
        n_top = n;
    }
}
else if (toplist[n].Contains(this.I) && toplist[n].Contains(this.J))
{
    flag_i = true;
    flag_j = true;
    this.Weight = -1;
    break;
}
```

Ниже представлен листинг для продолжения метода:

```
if (flag_i == false && flag_j == false)
{
    if (this.I != this.J)
    {
        toplist.Add(new HashSet<int>() { this.I, this.J });
    }
    else
    {
        this.Weight = -1;
    }
}
else if (flag_i == true && flag_j == false)
{
    toplist[n_top].Add(this.J);
}
else if (flag_i == false && flag_j == true)
{
    toplist[n_top].Add(this.I);
}
```

Когда цикл закончен, все вершины объединены в одно подмножество.

Ниже представлена блок-схема алгоритма Крускала:

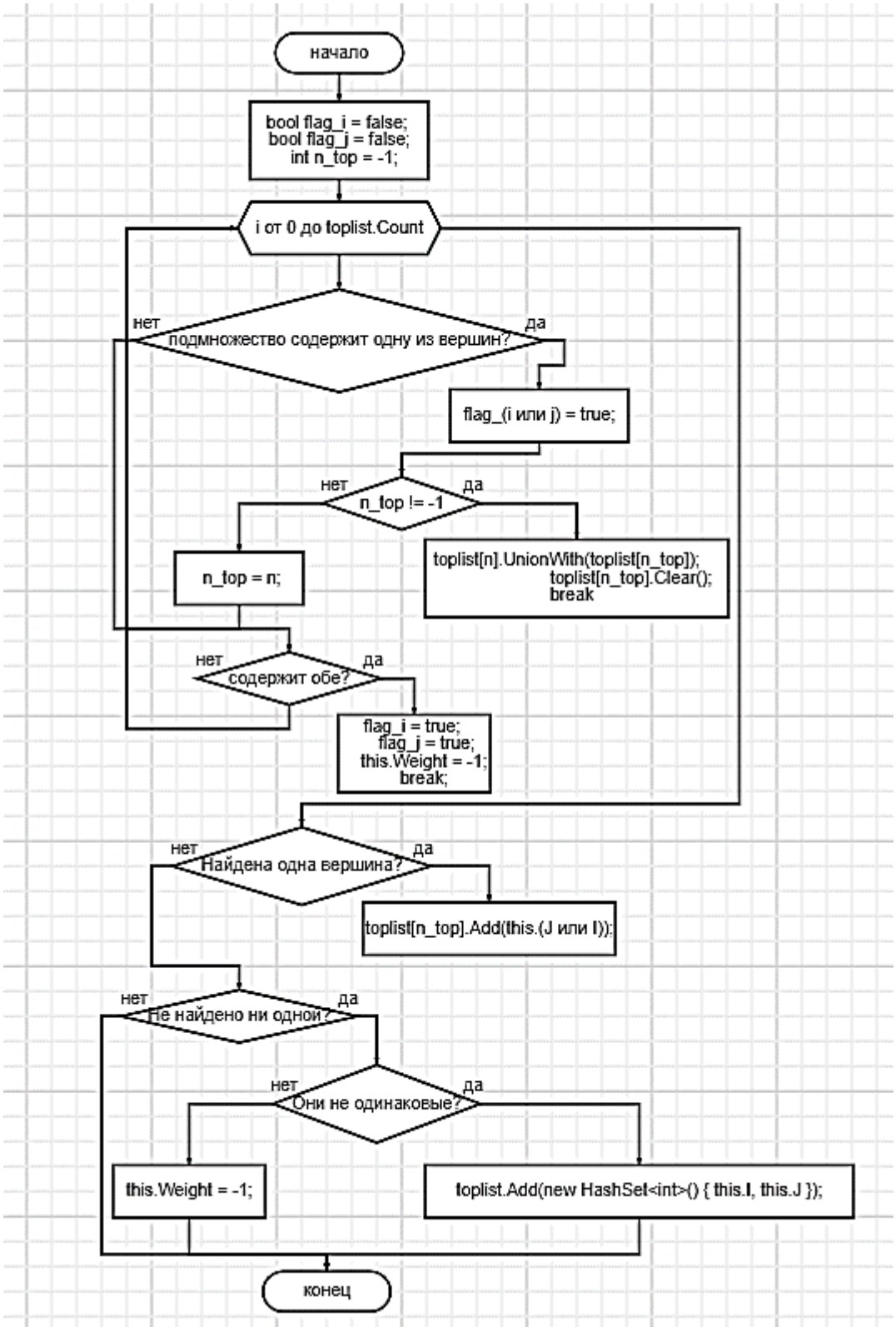



Рисунок 10 – Блок-схема алгоритма Крускала

Ниже представлен результат расчета примера:

```
(0) вершина -->[7; 12; 10; 0; 11; 12; ]

(1) вершина -->[12; 12; 0; 14; 0; 18; ]

(2) вершина -->[10; 0; 9; 0; 0; 5; ]

(3) вершина -->[0; 14; 0; 0; 0; 0; ]

(4) вершина -->[11; 0; 0; 0; 1; 12; ]

(5) вершина -->[12; 18; 5; 0; 12; 12; ]
Отсортированный список ребер:
1 5 7 9 10 11 12 12 12 12 12 14 18
ребро между вершинами 2 и 5 вес5
ребро между вершинами 0 и 2 вес10
ребро между вершинами 0 и 4 вес11
ребро между вершинами 0 и 1 вес12
ребро между вершинами 1 и 3 вес14
```

Выводы:

В ходе выполнения работы изучены и реализованы на языке C# основные алгоритмы на графах – алгоритм поиска в ширину и в глубину, алгоритм Дейкстры и алгоритм Крускала.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие задачи решаются с помощью алгоритма Дейкстры и алгоритма Крускала?
2. Какие ограничения и условия на применение алгоритма Дейкстры?
3. Какие конструкции ООП C# применяются для реализации алгоритма поиска в ширину и в глубину?
4. Как оценивается сложность алгоритма Дейкстры?
5. К какому классу сложности относится алгоритм Дейкстры?
6. Как можно оценить точность алгоритма Дейкстры?

**Приложение 1. Алгоритм Дейкстры**

Класс для описания информации о вершинах графа

```
/// <summary>
/// Информация о вершине
/// </summary>
public class GraphVertexInfo
{
    /// <summary>
    /// Вершина
    /// </summary>
    public GraphVertex Vertex { get; set; }

    /// <summary>
    /// Не посещенная вершина
    /// </summary>
    public bool IsUnvisited { get; set; }

    /// <summary>
    /// Сумма весов ребер
    /// </summary>
    public int EdgesWeightSum { get; set; }

    /// <summary>
    /// Предыдущая вершина
    /// </summary>
    public GraphVertex PreviousVertex { get; set; }

    /// <summary>
    /// Конструктор
    /// </summary>
    /// <param name="vertex">Вершина</param>
    public GraphVertexInfo(GraphVertex vertex)
    {
        Vertex = vertex;
        IsUnvisited = true;
        EdgesWeightSum = int.MaxValue;
        PreviousVertex = null;
    }
}
```

Класс с реализацией поиска кратчайшего пути

```
using System.Collections.Generic;

/// <summary>
/// Алгоритм Дейкстры
/// </summary>
public class Dijkstra
{
    Graph graph;
```

```
List<GraphVertexInfo> infos;

/// <summary>
/// Конструктор
/// </summary>
/// <param name="graph">Граф</param>
public Dijkstra(Graph graph)
{
   this.graph = graph;
}

/// <summary>
/// Инициализация информации
/// </summary>
void InitInfo()
{
   infos = new List<GraphVertexInfo>();
   foreach (var v in graph.Vertices)
   {
      infos.Add(new GraphVertexInfo(v));
   }
}

/// <summary>
/// Получение информации о вершине графа
/// </summary>
/// <param name="v">Вершина</param>
/// <returns>Информация о вершине</returns>
GraphVertexInfo GetVertexInfo(GraphVertex v)
{
   foreach (var i in infos)
   {
      if (i.Vertex.Equals(v))
      {
         return i;
      }
   }

   return null;
}

/// <summary>
/// Поиск непосещенной вершины с минимальным значением суммы
/// </summary>
/// <returns>Информация о вершине</returns>
public GraphVertexInfo FindUnvisitedVertexWithMinSum()
{
   var minValue = int.MaxValue;
   GraphVertexInfo minVertexInfo = null;
   foreach (var i in infos)
   {
```

```
        if (i.IsUnvisited && i.EdgesWeightSum < minValue)
        {
            minVertexInfo = i;
            minValue = i.EdgesWeightSum;
        }
    }

    return minVertexInfo;
}

/// <summary>
/// Поиск кратчайшего пути по названиям вершин
/// </summary>
/// <param name="startName">Название стартовой вершины</param>
/// <param name="finishName">Название финишной вершины</param>
/// <returns>Кратчайший путь</returns>
public string FindShortestPath(string startName, string finishName)
{
    return FindShortestPath(graph.FindVertex(startName), graph.FindVertex(finishName));
}

/// <summary>
/// Поиск кратчайшего пути по вершинам
/// </summary>
/// <param name="startVertex">Стартовая вершина</param>
/// <param name="finishVertex">Финишная вершина</param>
/// <returns>Кратчайший путь</returns>
public string FindShortestPath(GraphVertex startVertex, GraphVertex finishVertex)
{
    InitInfo();
    var first = GetVertexInfo(startVertex);
    first.EdgesWeightSum = 0;
    while (true)
    {
        var current = FindUnvisitedVertexWithMinSum();
        if (current == null)
        {
            break;
        }

        SetSumToNextVertex(current);
    }

    return GetPath(startVertex, finishVertex);
}

/// <summary>
/// Вычисление суммы весов ребер для следующей вершины
/// </summary>
/// <param name="info">Информация о текущей вершине</param>
void SetSumToNextVertex(GraphVertexInfo info)
{
```

```
            info.IsUnvisited = false;
            foreach (var e in info.Vertex.Edges)
            {
                var nextInfo = GetVertexInfo(e.ConnectedVertex);
                var sum = info.EdgesWeightSum + e.EdgeWeight;
                if (sum < nextInfo.EdgesWeightSum)
                {
                    nextInfo.EdgesWeightSum = sum;
                    nextInfo.PreviousVertex = info.Vertex;
                }
            }
        }

        /// <summary>
        /// Формирование пути
        /// </summary>
        /// <param name="startVertex">Начальная вершина</param>
        /// <param name="endVertex">Конечная вершина</param>
        /// <returns>Путь</returns>
        string GetPath(GraphVertex startVertex, GraphVertex endVertex)
        {
            var path = endVertex.ToString();
            while (startVertex != endVertex)
            {
                endVertex = GetVertexInfo(endVertex).PreviousVertex;
                path = endVertex.ToString() + path;
            }

            return path;
        }
    }
```

Программа для поиска кратчайшего пути между заданными вершинами графа с использованием алгоритма Дейкстры

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

class Program
{
    static void Main(string[] args)
    {
        var g = new Graph();

        //добавление вершин
        g.AddVertex("A");
        g.AddVertex("B");
        g.AddVertex("C");
        g.AddVertex("D");
        g.AddVertex("E");
        g.AddVertex("F");
        g.AddVertex("G");
```

```
        //добавление ребер
        g.AddEdge("A", "B", 22);
        g.AddEdge("A", "C", 33);
        g.AddEdge("A", "D", 61);
        g.AddEdge("B", "C", 47);
        g.AddEdge("B", "E", 93);
        g.AddEdge("C", "D", 11);
        g.AddEdge("C", "E", 79);
        g.AddEdge("C", "F", 63);
        g.AddEdge("D", "F", 41);
        g.AddEdge("E", "F", 17);
        g.AddEdge("E", "G", 58);
        g.AddEdge("F", "G", 84);

        var dijkstra = new Dijkstra(g);
        var path = dijkstra.FindShortestPath("A", "G");
        Console.WriteLine(path);
        Console.ReadLine();
    }
}
```

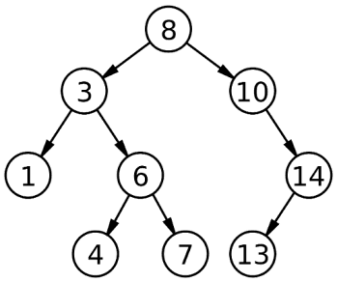
**Приложение 2. Алгоритмы работы с деревьями**

**Бинарное дерево** (*binary tree*) – это структура данных, которая состоит из узлов, при этом каждый узел может иметь не более двух дочерних. Первый узел называется корневым или родительским, а дочерние – правым и левым наследником(потомком).

Бинарное дерево поиска

**Двоичное дерево поиска** (*binary search tree*) – это бинарное дерево, которое соответствует следующим условиям:

- Дочерние узлы являются двоичными деревьями поиска;
- Для произвольного узла:
  - Все значения **левого** поддерева, **меньше** значения родительского узла;
  - Все значения **правого** поддерева, **больше** значения родительского узла.




Класс для описания узла

Для начала необходимо создать класс для описания узла бинарного дерева. Для того, чтобы иметь возможность хранить произвольный тип данных, используем обобщенный тип **T**, к которому применима операция сравнения, иными словами, можно будет использовать любой тип данных, реализующий интерфейс **IComparable**:

```
/// <summary>
/// Расположения узла относительно родителя
/// </summary>
public enum Side
{
   Left,
   Right
}

/// <summary>
/// Узел бинарного дерева
/// </summary>
/// <typeparam name="T"></typeparam>
public class BinaryTreeNode<T> where T : IComparable
{
   /// <summary>
   /// Конструктор класса
   /// </summary>
   /// <param name="data">Данные</param>
```

```
    public BinaryTreeNode(T data)
    {
        Data = data;
    }

    /// <summary>
    /// Данные которые хранятся в узле
    /// </summary>
    public T Data { get; set; }

    /// <summary>
    /// Левая ветка
    /// </summary>
    public BinaryTreeNode<T> LeftNode { get; set; }

    /// <summary>
    /// Правая ветка
    /// </summary>
    public BinaryTreeNode<T> RightNode { get; set; }

    /// <summary>
    /// Родитель
    /// </summary>
    public BinaryTreeNode<T> ParentNode { get; set; }

    /// <summary>
    /// Расположение узла относительно его родителя
    /// </summary>
    public Side? NodeSide =>
        ParentNode == null
        ? (Side?)null
        : ParentNode.LeftNode == this
            ? Side.Left
            : Side.Right;

    /// <summary>
    /// Преобразование экземпляра класса в строку
    /// </summary>
    /// <returns>Данные узла дерева</returns>
    public override string ToString() => Data.ToString();
}
```

Класс бинарное дерево

Создадим новый класс **BinaryTree**, который будет описывать бинарное дерево поиска и содержать в себе все необходимые методы для работы с данными. Для начала добавим в него родительский элемент:

```
/// <summary>
/// Бинарное дерево
/// </summary>
/// <typeparam name="T">Тип данных хранящихся в узлах</typeparam>
```

```
public class BinaryTree<T> where T : IComparable
{
   /// <summary>
   /// Корень бинарного дерева
   /// </summary>
   public BinaryTreeNode<T> RootNode { get; set; }
}
```

Добавление данных

Первый метод, который мы реализуем, будет добавлять новый узел в дерево:

```
/// <summary>
/// Добавление нового узла в бинарное дерево
/// </summary>
/// <param name="node">Новый узел</param>
/// <param name="currentNode">Текущий узел</param>
/// <returns>Узел</returns>
public BinaryTreeNode<T> Add(BinaryTreeNode<T> node, BinaryTreeNode<T> currentNode
= null)
{
   if (RootNode == null)
   {
      node.ParentNode = null;
      return RootNode = node;
   }

   currentNode = currentNode ?? RootNode;
   node.ParentNode = currentNode;
   int result;
   return (result = node.Data.CompareTo(currentNode.Data)) == 0
      ? currentNode
      : result < 0
         ? currentNode.LeftNode == null
            ? (currentNode.LeftNode = node)
            : Add(node, currentNode.LeftNode)
         : currentNode.RightNode == null
            ? (currentNode.RightNode = node)
            : Add(node, currentNode.RightNode);
}
```

Данный метод сначала проверяет корневой узел и если он пустой, то присваивает ему значение нового узла. В противном случае производит сравнение данных нового узла с значением узла в который вставляется узел, и если:

- Значения совпадают, то значение узла остается прежним;
- Новое значение меньше текущего – вставляем в левую ветку дерева;
- Новое значение больше текущего – вставляем в правую ветку дерева.

Для упрощения непосредственного добавления данных, создадим перегрузку метода:

```
/// <summary>
```

```
/// Добавление данных в бинарное дерево
/// </summary>
/// <param name="data">Данные</param>
/// <returns>Узел</returns>
public BinaryTreeNode<T> Add(T data)
{
   return Add(new BinaryTreeNode<T>(data));
}
```

Поиск узла дерева по значению

Напишем рекурсивный метод для поиска узла по его значению:

```
/// <summary>
/// Поиск узла по значению
/// </summary>
/// <param name="data">Искомое значение</param>
/// <param name="startWithNode">Узел начала поиска</param>
/// <returns>Найденный узел</returns>
public BinaryTreeNode<T> FindNode(T data, BinaryTreeNode<T> startWithNode = null)
{
   startWithNode = startWithNode ?? RootNode;
   int result;
   return (result = data.CompareTo(startWithNode.Data)) == 0
      ? startWithNode
      : result < 0
         ? startWithNode.LeftNode == null
            ? null
            : FindNode(data, startWithNode.LeftNode)
         : startWithNode.RightNode == null
            ? null
            : FindNode(data, startWithNode.RightNode);
}
```

Метод начинает поиск с указанного узла(если не задан, то с корня дерева), сравнивая значения текущего элемента с искомым, если значение найдено, возвращается ссылка на узел, если нет, то производится рекурсивный обход дочерних веток дерева.

Удаление узла бинарного дерева

```
/// <summary>
/// Удаление узла бинарного дерева
/// </summary>
/// <param name="node">Узел для удаления</param>
public void Remove(BinaryTreeNode<T> node)
{
   if (node == null)
   {
      return;
   }

   var currentNodeSide = node.NodeSide;
```

```
//если у узла нет подузлов, можно его удалить
if (node.LeftNode == null && node.RightNode == null)
{
   if (currentNodeSide == Side.Left)
   {
      node.ParentNode.LeftNode = null;
   }
   else
   {
      node.ParentNode.RightNode = null;
   }
}
//если нет левого, то правый ставим на место удаляемого
else if (node.LeftNode == null)
{
   if (currentNodeSide == Side.Left)
   {
      node.ParentNode.LeftNode = node.RightNode;
   }
   else
   {
      node.ParentNode.RightNode = node.RightNode;
   }

   node.RightNode.ParentNode = node.ParentNode;
}
//если нет правого, то левый ставим на место удаляемого
else if (node.RightNode == null)
{
   if (currentNodeSide == Side.Left)
   {
      node.ParentNode.LeftNode = node.LeftNode;
   }
   else
   {
      node.ParentNode.RightNode = node.LeftNode;
   }

   node.LeftNode.ParentNode = node.ParentNode;
}
//если оба дочерних присутствуют,
//то правый становится на место удаляемого,
//а левый вставляется в правый
else
{
   switch (currentNodeSide)
   {
      case Side.Left:
         node.ParentNode.LeftNode = node.RightNode;
         node.RightNode.ParentNode = node.ParentNode;
         Add(node.LeftNode, node.RightNode);
         break;
```

```
            case Side.Right:
                node.ParentNode.RightNode = node.RightNode;
                node.RightNode.ParentNode = node.ParentNode;
                Add(node.LeftNode, node.RightNode);
                break;
            default:
                var bufLeft = node.LeftNode;
                var bufRightLeft = node.RightNode.LeftNode;
                var bufRightRight = node.RightNode.RightNode;
                node.Data = node.RightNode.Data;
                node.RightNode = bufRightRight;
                node.LeftNode = bufRightLeft;
                Add(bufLeft, node);
                break;
        }
    }
}
```

Поскольку в метод для удаления, необходимо передавать экземпляр класса, то для удобства сделаем перегрузку для поиска и удаления:

```
/// <summary>
/// Удаление узла дерева
/// </summary>
/// <param name="data">Данные для удаления</param>
public void Remove(T data)
{
    var foundNode = FindNode(data);
    Remove(foundNode);
}
```

Вывод бинарного дерева на экран

Напишем рекурсивный метод для вывода дерева в экран консоли:

```
/// <summary>
/// Вывод бинарного дерева
/// </summary>
public void PrintTree()
{
    PrintTree(RootNode);
}

/// <summary>
/// Вывод бинарного дерева начиная с указанного узла
/// </summary>
/// <param name="startNode">Узел с которого начинается печать</param>
/// <param name="indent">Отступ</param>
/// <param name="side">Сторона</param>
private void PrintTree(BinaryTreeNode<T> startNode, string indent = "", Side? side = null)
{
    if (startNode != null)
```

```
    {
        var nodeSide = side == null ? "+" : side == Side.Left ? "L" : "R";
        Console.WriteLine($"{indent} [{nodeSide}]- {startNode.Data}");
        indent += new string(' ', 3);
        //рекурсивный вызов для левой и правой веток
        PrintTree(startNode.LeftNode, indent, Side.Left);
        PrintTree(startNode.RightNode, indent, Side.Right);
    }
}
```

Программа для работы с бинарным деревом поиска

Напишем программу для работы с двоичным деревом, для простоты будем использовать целочисленный тип данных:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        var binaryTree = new BinaryTree<int>();

        binaryTree.Add(8);
        binaryTree.Add(3);
        binaryTree.Add(10);
        binaryTree.Add(1);
        binaryTree.Add(6);
        binaryTree.Add(4);
        binaryTree.Add(7);
        binaryTree.Add(14);
        binaryTree.Add(16);

        binaryTree.PrintTree();

        Console.WriteLine(new string('-', 40));
        binaryTree.Remove(3);
        binaryTree.PrintTree();

        Console.WriteLine(new string('-', 40));
        binaryTree.Remove(8);
        binaryTree.PrintTree();

        Console.ReadLine();
    }
}
```

**Литература**


1. Рутковская Д., Пилиньский М., Рутковский Л. Нейронные сети, генетические алгоритмы и нечеткие системы: Пер.с польск. И.Д.Рудинского. М.: Горячая линия-Телеком, 2013. - 384 с. - Режим доступа: http://e.lanbook.com/books/element.php?pl1_cid=25&pl1_id=11843
2. Окулов, С.М. Алгоритмы обработки строк [Электронный ресурс] : учебное пособие / С.М. Окулов. — Электрон. дан. — Москва : Издательство "Лаборатория знаний", 2015. — 258 с. — Режим доступа: https://e.lanbook.com/book/66113. — Загл. с экрана.
3. Костюкова Н.И. Графы и их применение. Комбинаторные алгоритмы для программистов. М., 2007.
4. Дж. Клейнберг, Е. Тардос. Алгоритмы: разработка и применение. // Спб: Питер. - 2016. - 800 с.
5. Клейнберг Дж., Тардос Е. Алгоритмы: анализ и разработка. Санкт-Петербург: Питер, 2016, 800 с.
6. Клейнберг Дж., Тардос Е. Алгоритмы: разработка и применение. Классика Computers Science / Пер. с англ. Е. Матвеева. — СПб.: Питер, 2016. — 800 с.