Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Основы алгоритмизации и программирования

Отчёт
по учебной практике(ознакомительной)
на тему

Грациозная разметка графа

Студент А.А. Горох

Руководитель Ю.А. Луцик

МИНСК 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В программировании бинарные деревья, и графы в целом, являются одними из наиболее важных структур данных. Они позволяют эффективно хранить и обрабатывать большие объемы информации, а также решать различные задачи, связанные с поиском, сортировкой, оптимизацией и так далее. Деревья используются в различных областях, таких как информационные системы, базы данных, алгоритмы поиска и многие другие.

Использование графов является важным аспектом программирования. Графы в частности используются для моделирования связей между объектами, что позволяет решать множество проблем, таких как оптимизация маршрутов, поиск кратчайшего пути, анализ социальных сетей и т.д.

Теория графов приложима как к наукам о поведении (теории информации, кибернетике, теории игр, теории систем, транспортным сетям), так и к чисто абстрактным дисциплинам (теории множеств, теории матриц, теории групп и так далее). Несмотря на разнообразные, усложнённые, малопонятные термины многие модельные и конфигурационные проблемы легко могут переформулироваться на язык теории графов, что позволяет значительно проще выявлять их концептуальные трудности и решать поставленные задачи.

Математические алгоритмы работы с графами позволяют разработчикам создавать более эффективные и оптимизированные алгоритмы для решения сложных задач. Теория графов помогает легко интерпретировать данные и использовать уже существующие оптимальные алгоритмы. Навыки работы с графами также помогают программистам улучшить производительность многих программ и сделать их более надежными и быстрыми.

1 ПОСТАНОВКА ЗАДАЧИ

Грациозная разметка графов (игра Соломона Голомба). Нужно построить граф и подписать его вершины различными целыми положительными числами (или нулем). Вершины нужно пронумеровать так, чтобы разности чисел, присвоенных соседним вершинам, не совпадали, и при этом наибольший номер вершины был как можно меньшим числом.

Основным решением данной задачи является построение дерева с корневым узлом равным 0 и дальнейшей нумерацией вершин «змейкой» (где каждые листы одного уровня нумеруются одно за одним, меняя направление обхода вершин с новым каждым уровнем на противоположное).

В программе реализованы следующие функции:

− Функция рекурсивного создания дерева до определенного уровня;
− Функция обхода дерева «змейкой»;
− Функция удаления лишних вершин;
− Создание и вывод матрицы смежности;
− Инфиксный вывод дерева.

2 ОПИСАНИЕ АЛГОРИТМА

Рекурсивное создание дерева.

Сначала вычисляем нужное количество уровней: пока количество вершин не равно нулю, вычитаем из него двойку с увеличивающейся степенью. Максимальная степень и будет равна количеству уровней.

На вход функции поступает узел (в самом начале алгоритма это узловой корень) и текущий уровень узла. Если уровень больше одного, то к данному добавляется два листа и из уровня вычитается единица. Далее функция вызывается рекурсивно заново для левого и правого листа от текущего корня. При уровне равном единице функция рекурсивно завершается.

Нумерация «змейкой».

Обход итеративный, то есть мы запускаем функцию обхода для каждого уровня, от узлового корня до максимального уровня. На каждом новом уровне меняем направление обхода на противоположное.

На вход самой функции поступают узел; текущий уровень; булевая переменная «слева направо», показывающая направление обхода; указатель на номер вершины.

Если указатель на узел равен нулю, то выходим из функции. Если уровень равен единице, то приписываем вершине текущий номер и инкрементируем номер для следующего узла. При уровне больше единицы рекурсивно вызываем функцию для левого листа, затем правого, или наоборот в зависимости от текущего направления обхода.

Удаление лишних вершин.

Алгоритм немного схож с нумерацией, но с отличием в том, что функция запускается сразу для нижнего уровня. А вместо нумерации проверяем номера листьев на то, чтобы они были меньше суммарного количества всех вершин в дереве. В данном случае направление обхода не важно, поэтому он идет слева направо.

Создание матрицы смежности.

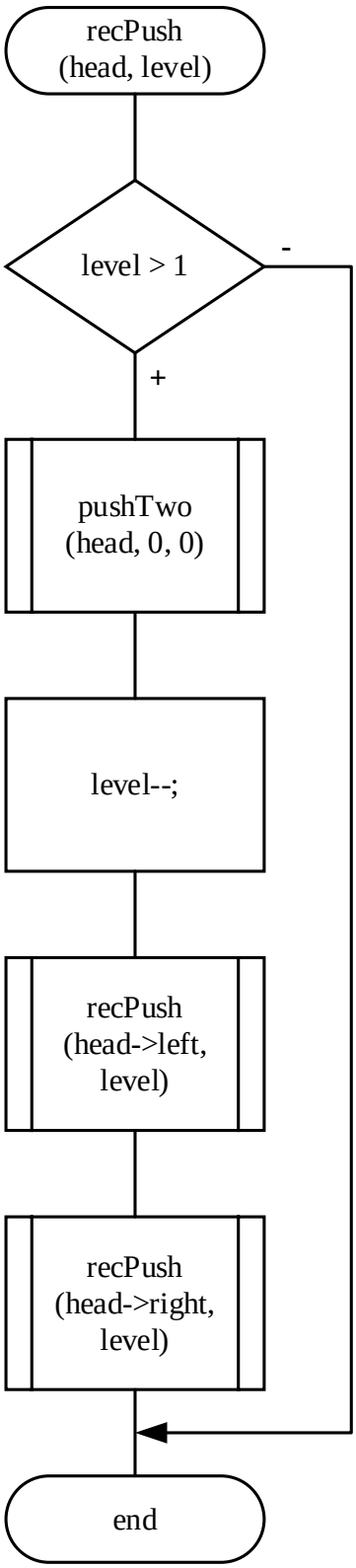
Изначально матрица заполняется нулями, а номера вершин содержатся в отдельном массиве. Потом запускается рекурсивное заполнение единицами на пересечении вершин. Если указатель на узел не равен нулю, то запускается обход на левый лист, потом заполняется матрица по координатам, равным номерам вершин текущего узла и смежного с ним, а затем запускается обход справа. В результате получается квадратная матрица по размерности равной общему количеству вершин.

Инфиксный вывод.

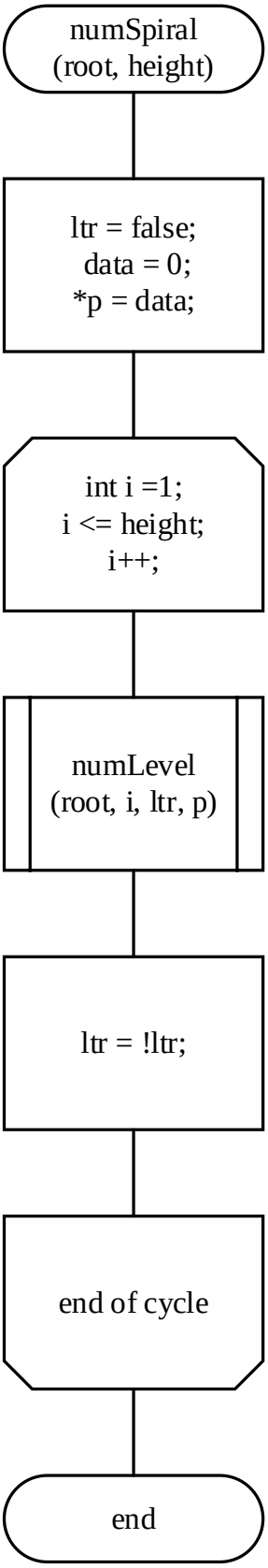
Основная идея такого вывода состоит в том, чтобы каждая вершина выводилась с новой строки и табулировалась в зависимости от положения на своем уровне. С каждой рекурсией табуляция текущей вершины увеличивается (чем больше уровень дерева, тем больше расстояние от узлового корня). Сам обход для вывода такой же (симметрический), как и при создании матрицы смежности.

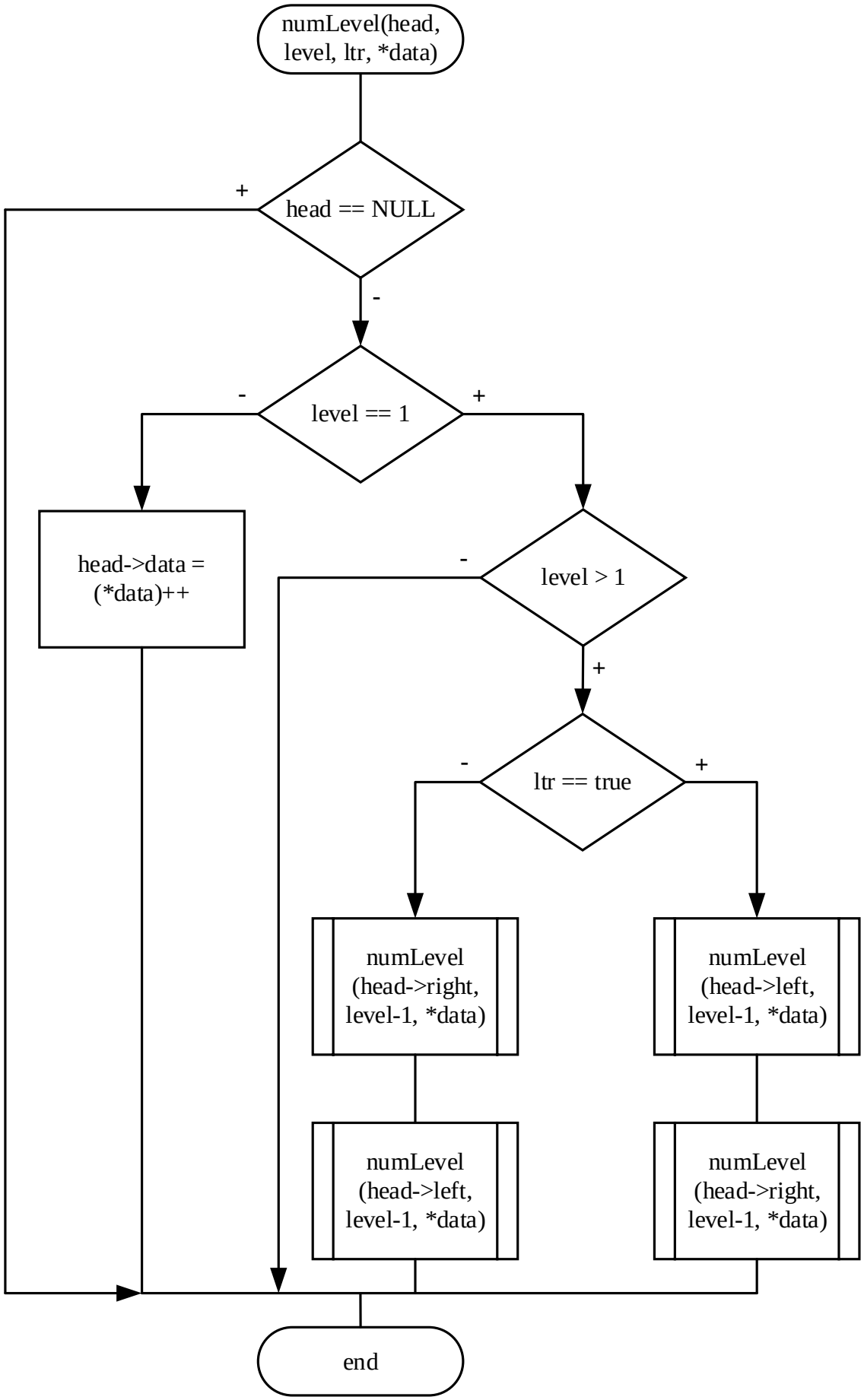
3 СХЕМА ПРОГРАММЫ

Рекурсивное создание дерева.

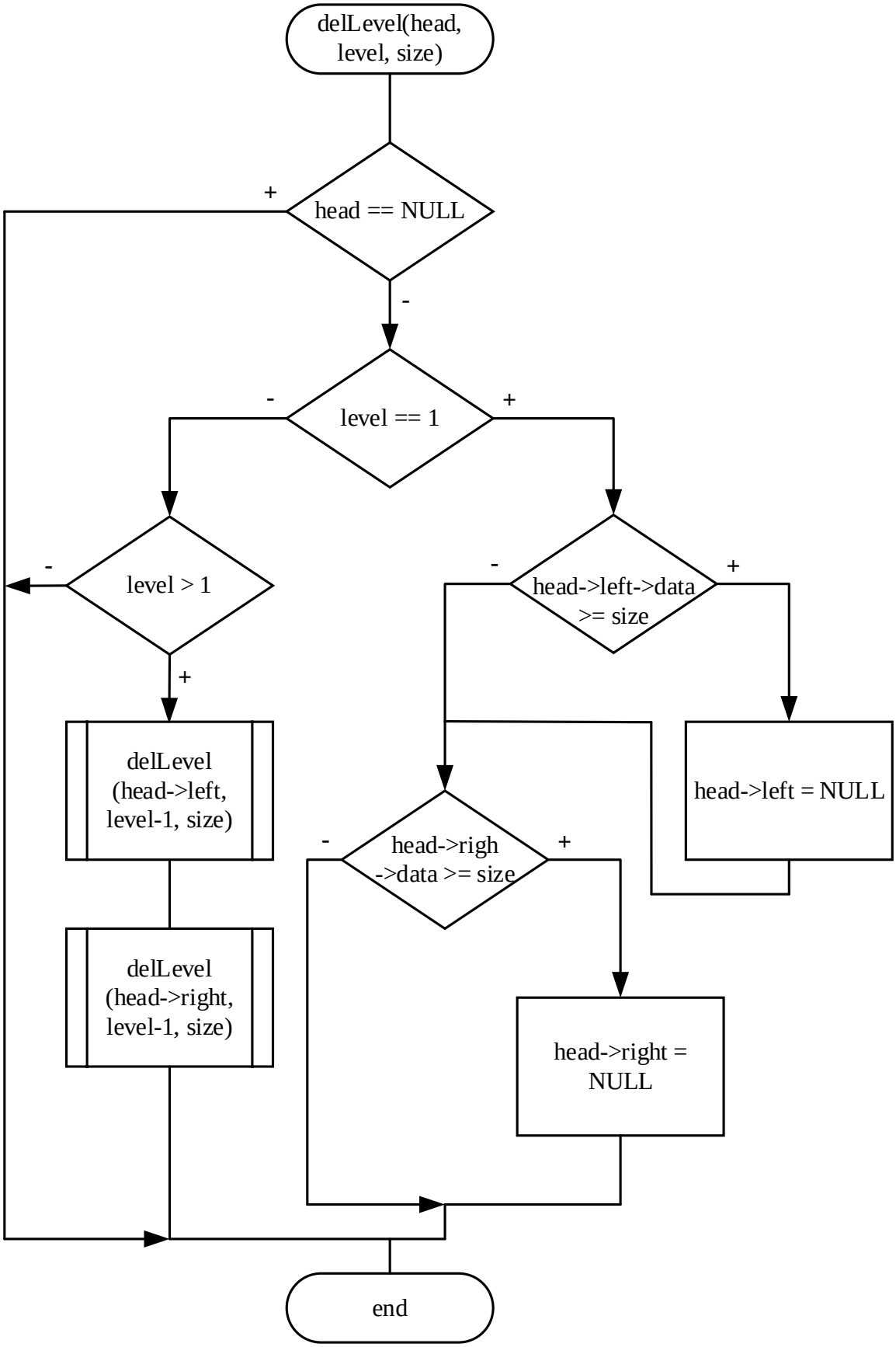


Нумерация «змейкой».




numLevel(head, level, ltr, *data)
head == NULL
+
-
level == 1
-
+
head->data = (*data)++
level > 1
-
+
ltr == true
-
+
numLevel (head->right, level-1, *data)
numLevel (head->left, level-1, *data)
numLevel (head->left, level-1, *data)
numLevel (head->right, level-1, *data)
end

Удаление лишних вершин.



4 КОД ПРОГРАММЫ

4.1 tree.h

```
#pragma once
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <conio.h>
#define COUNT 5

typedef struct Leaf {
    int data;
    struct Leaf *head;
    struct Leaf *left;
    struct Leaf *right;
} leaf;

leaf *pushTwo(leaf **head, int dl, int dr) {
    leaf *tmpl = (leaf*)malloc(sizeof(leaf));
    leaf *tmpr = (leaf*)malloc(sizeof(leaf));
    tmpl->left = tmpl->right = NULL;
    tmpl->data = dl;
    tmpr->left = tmpr->right = NULL;
    tmpr->data = dr;
    if(head != NULL)
    {
        (*head)->left = tmpl;
        (*head)->right = tmpr;
    }
    return *head;
}

int inputPosInt() {
    int scn=0;
    int num=0;
    char temp;
    do {
        rewind(stdin);
        scn=scanf("%d",&num);
        if(scn!=1 || num<=0)
        {
            printf("Wrong input\n");
            scn=0;
            continue;
        }
        while ((temp=getchar())!='\n')
        if(temp>'9' || temp<'0')
        {
            printf("Wrong input\n");
            scn=0;
            break;
```

```
        }
    }
    while(num<=0 || scn!=1);
    return num;
}

void recPush(leaf *head, int level) {
    if(level > 1)
    {
        pushTwo(&head, 0, 0);
        level--;
        recPush(head->left, level);
        recPush(head->right, level);
    }
}

void printTree(leaf *head) {
    if (head != NULL)
    {
        printTree(head->left);
        printf("%d  ", head->data);
        printTree(head->right);
    }
}

void print2D(leaf *head, int tab) {
    if (head == NULL)
        return;
    tab += COUNT;
    print2D(head->right, tab);
    for(int i = COUNT; i < tab; i++)
    {
        printf(" ");
    }
    if(head->left != NULL && head->right != NULL)
        printf("%d <\n", head->data);
    else
        printf("%d\n", head->data);
    print2D(head->left, tab);
}
```

4.2 main.c

```
#include "tree.h"
#include <math.h>

void numLevel(leaf *head, int level, bool ltr, int *data) {
    if (head == NULL)
        return;
    if (level == 1)
    {
        head->data = (*data)++;
```

```
    }
    else
    if (level > 1)
    {
        if (ltr)
        {
            numLevel(head->left, level - 1, ltr, data);
            numLevel(head->right, level - 1, ltr, data);
        }
        else
        {
            numLevel(head->right, level - 1, ltr, data);
            numLevel(head->left, level - 1, ltr, data);
        }
    }
}

void numSpiral(leaf *root, int height) {
    bool ltr = false;
    int data = 0, *p = &data;
    for (int i = 1; i <= height; i++)
    {
        numLevel(root, i, ltr, p);
        ltr = !ltr;
    }
}

void delLevel(leaf *head, int level, int size) {
    if (head == NULL)
        return;
    if (level == 1)
    {
        if(head->left->data >= size)
        {
            leaf *tmp = head->left;
            head->left = NULL;
            free(tmp);
        }
        if(head->right->data >= size)
        {
            leaf *tmp = head->right;
            head->right = NULL;
            free(tmp);
        }
    }
    else
    if (level > 1)
    {
        delLevel(head->left, level - 1, size);
        delLevel(head->right, level - 1, size);
    }
}
```

```
    leaf *fillTree(leaf *root, int size) {
        int height = 1, buf = size;
        while(buf > 0)
            buf -= pow(2, height++);
        recPush(root, height);
        numSpiral(root, height);
        delLevel(root, height-1, size);
        return root;
    }

    void fillOrder(leaf *head, int **adjacy) {
        if (head != NULL)
        {
            fillOrder(head->left, adjacy);
            if(head->left != NULL)
                adjacy[head->data][head->left->data] =
adjacy[head->left->data][head->data] = 1;
            if(head->right != NULL)
                adjacy[head->data][head->right->data] =
adjacy[head->right->data][head->data] = 1;
            fillOrder(head->right, adjacy);
        }
    }

    void printMatrix(int size, int **adjacy, int *peak) {
        printf("\n      ");
        for(int i = 0; i < size; i++)
            printf("%4d", peak[i]);

        printf("\n  ____");
        for(int i = 0; i < size; i++)
            printf("____");

        printf("\n");
        for(int i = 0; i < size; i++)
        {
            printf("%4d |", peak[i]);
            for(int j = 0; j < size; j++)
                printf("%4d", adjacy[i][j]);
            printf("\n");
        }
    }

    void fillMatrix(leaf *root, int size, int **adjacy, int
*peak) {
        for(int i = 0;  i < size; i++)
            peak[i] = i;

        for(int i = 0; i < size; i++)
            for(int j = 0; j < size; j++)
                adjacy[i][j] = 0;

        fillOrder(root, adjacy);
```

```
}

int main() {
    leaf *root = (leaf*)malloc(sizeof(leaf));
    root->head = NULL;
    printf("Enter amount of graph vertices:\t");
    int size = inputPosInt();
    int *peak = (int*)malloc(size*sizeof(int));
    int **adjacy = (int**)malloc(size*sizeof(int*));
    for(int i = 0; i < size; i++)
        *(adjacy + i) = (int*)malloc(size*sizeof(int));
    root = fillTree(root, size);
    printf("\n");
    fillMatrix(root, size, adjacy, peak);
    printMatrix(size, adjacy, peak);
    printf("\n\n");
    print2D(root, 0);
    getch();
    return 0;
}
```

5 РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ПРОГРАММЫ

```
Enter amount of graph vertices: 13


        0   1   2   3   4   5   6   7   8   9  10  11  12
  ____________________________________________________________
   0 |    0   1   1   0   0   0   0   0   0   0   0   0   0
   1 |    1   0   0   0   0   1   1   0   0   0   0   0   0
   2 |    1   0   0   1   1   0   0   0   0   0   0   0   0
   3 |    0   0   1   0   0   0   0   0   0   0   0   0   0
   4 |    0   0   1   0   0   0   0   0   0   0   0   1   1
   5 |    0   1   0   0   0   0   0   0   0   1   1   0   0
   6 |    0   1   0   0   0   0   0   1   1   0   0   0   0
   7 |    0   0   0   0   0   0   1   0   0   0   0   0   0
   8 |    0   0   0   0   0   0   1   0   0   0   0   0   0
   9 |    0   0   0   0   0   1   0   0   0   0   0   0   0
  10 |    0   0   0   0   0   1   0   0   0   0   0   0   0
  11 |    0   0   0   0   1   0   0   0   0   0   0   0   0
  12 |    0   0   0   0   1   0   0   0   0   0   0   0   0


          3
     2 <
               12
          4 <
               11
0 <
               10
          5 <
               9
     1 <
               8
          6 <
               7
```

ЗАКЛЮЧЕНИЕ

В ходе данной работы были изучены основные алгоритмы работы с такой структурой данных, как бинарное дерево. Разработаны и использованы на практике различные алгоритмы обхода деревьев, которые пригодятся для дальнейшего изучения алгоритмизации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Луцик, Ю. А. «Основы алгоритмизации и программирования: язык Си» − учеб.-метод. пособие / Ю. А. Луцик, А. М. Ковальчук, Е. А. Сасин. – Минск : БГУиР, 2015. – 170 с.

[2] «Язык программирования Си» / Б. Керниган, Д. Ричи – 3-е издание.

[3] Бхаргава А. «Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих» / А. Бхаргава.

[4] Прата С. «Язык программирования С» / С. Прата – 6-е издание.