МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №4 (вариант 7-д)

по дисциплине «Алгоритмы и структуры данных»

Тема: «Деревья»

Студент гр. 7381 _____ Адамов Я.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2018

Цель работы.

Ознакомиться с такой структурой данных, как дерево, и научиться применять деревья на практике.

Основные теоретические положения.

Дерево – это конечное множество *T*, состоящее из одного или более узлов, таких, что

а) имеется один специально обозначенный узел, называемый корнем данного дерева;

б) остальные узлы (исключая корень) содержатся в $m \gg 0$ попарно не пересекающихся множествах *T1*, *T2*, ..., *Tm*, каждое из которых, в свою очередь, является деревом.

Деревья *T1, T2, ..., Tm* называются поддеревьями данного дерева.

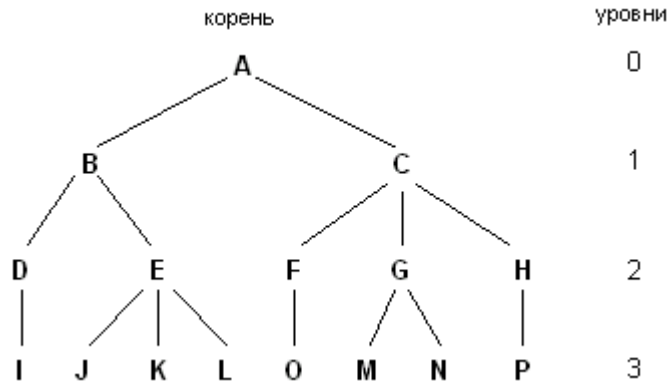

Рисунок 1 – Пример дерева

Бинарное дерево – дерево, в котором каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.

Рекурсивное определение бинарного дерева:

<BinTree> ::= (<данные> < BinTree > < BinTree >) | nil .

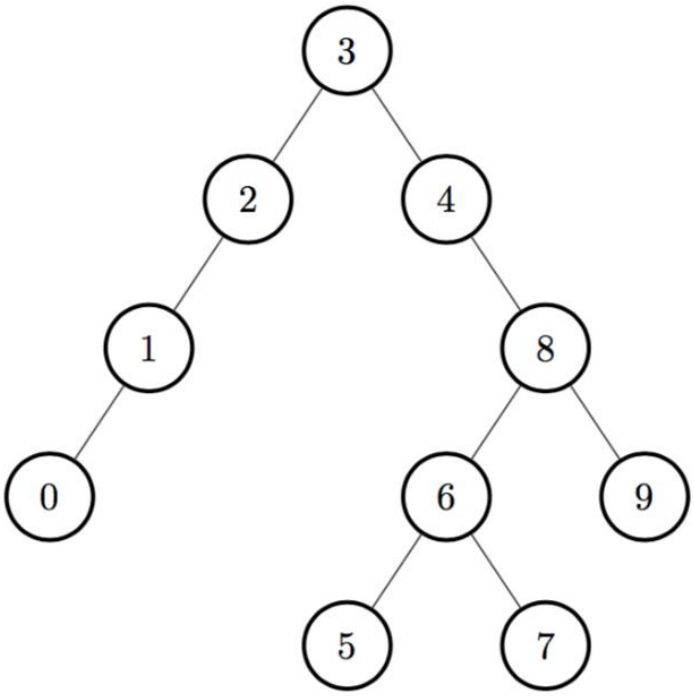

Рисунок 2 – Пример бинарного дерева

Задание.

Вариант 7-д.

Рассматриваются бинарные деревья с элементами типа char. Заданы перечисления узлов некоторого дерева b в порядке КЛП и ЛКП. Требуется:

а) восстановить дерево b и вывести его изображение;

б) перечислить узлы дерева b в порядке ЛПК.

Пояснение задания.

Вариант 7-д.

На вход программе подаются две строки: обходы дерева в порядках КЛП и ЛКП. По данным строкам нужно восстановить дерево и вывести его изображение на экран или в файл. Также необходимо перечислить элементы восстановленного дерева в порядке ЛПК.

Ход работы.

Программа будет писаться на языке C.

Исходный файл: main.c

Программа считывает 2 строки *traversal_KLP* (обход в порядке КЛП) и *traversal_LKP* (обход в порядке ЛКП) из файла или с клавиатуры. После чего строки передаются функции *treeRecovering*(), которая проверяет строки на валидность и восстанавливает бинарное дерево, если ошибок не возникло, в ином случае программа выводит соответствующее сообщение об ошибке и завершает работу.

После удачного построения бинарного дерева вызываются функции *treeDrawing*() и *print_traversal_LPK*(), которые выводят на экран изображение дерева и обход его элементов в порядке ЛПК.

Для демонстрации работы было написано несколько тестов, а также скрипт perform_tests.sh, который запускает все эти тесты. Полная демонстрация работы программы, а также её тестирование находится в Приложении А.

Описание функций и структур.

1) *struct BinTree*

Структура бинарного дерева. Для работы деревом используются следующие функции: leftBinTree, rightBinTree, initBinTree, findBinTree, binTreeDeepLevel.

Поля структуры:

- **char a**: значение элемента.
- **struct binTree* leftBinTree**: указатель на левое поддерево.
- **struct binTree* rightBinTree**: указатель на правое поддерево.

2) *BinTree* initBinTree(char a)*

Инициализация бинарного дерева.

Возвращаемое значение: бинарное дерево, значение корня которого – *a*, указатели *leftBinTree* и *rightBinTree* имеют значение *NULL*.

3) *BinTree* leftBinTree(BinTree* binTree)*

Функция возвращает указатель на левое поддерево.

Параметры:

- **binTree**: указатель на дерево, левое поддерево которого необходимо вернуть.

Возвращаемое значение: указатель на левое поддерево дерева *binTree*.

4) *BinTree* rightBinTree(BinTree* binTree)*

Функция возвращает указатель на правое поддерево.

Параметры:

- **binTree**: указатель на дерево, правое поддерево которого необходимо вернуть.

Возвращаемое указатель на правое поддерево дерева *binTree*.

5) *BinTree* findBinTree(BinTree* binTree, char a)*

Функция ищет указанный элемент в дереве.

Параметры:

- **binTree**: указатель на дерево.
- **a**: искомый элемент

Возвращаемое значение: указатель на поддерево, корнем которого является элемент *a*. Если такой элемент отсутствует в дерево, то функция возвращает *NULL*.

6) *int binTreeDeepLevel(BinTree* binTree)*

Функция возвращает глубину дерева.

Параметры:

- **binTree**: указатель на дерево.

Возвращаемое значение: глубина дерева *binTree*.

7) *int treeRecovering(BinTree** binTree, char* traversal_KLP, char* traversal_LKP)*

Функция строит бинарное дерево по перечислению его элементов в порядках КЛП и ЛКП.

Параметры:

- **binTree**: указатель на указатель, содержащий адрес, по которому будет записываться восстановленное дерево.
- **traversal_KLP**: указатель на строку, содержащую перечисление элементов дерева в порядке КЛП
- **traversal_LKP**: указатель на строку, содержащую перечисление элементов дерева в порядке ЛКП.

Описание алгоритма:

В начале функция проверяет строки *traversal_KLP* и *traversal_LKP* на валидность: длины строк должны быть равны, содержать только допустимые символы, не содержать повторяющихся символов, состоять из одинакового набора символов. Если одно из условий не выполняется, функция выводит соответствующее сообщение об ошибке и возвращает значение 1. Если ошибок не возникло, функция строит дерево.

Алгоритм восстановления дерева:

Индексы *index_KLP* и *index_LKP*, указывают на начала соответствующих строк обходов дерева. Функция идёт по символам строки *traversal_KLP* и каждый элемент делает левым поддеревом предыдущего элемента, пока текущий символ строки *traversal_KLP* не станет равным текущему символу строки *traversal_LKP*, после чего начинают перечисляться символы строки *traversal_KLP* до тех пор, пока не встретится новый элемент, которого ещё нет в дереве, то есть идёт обход обратно к корню (при этом происходит проверка на соответствие двух обходов одному дереву: в обратном обходе узлы дерева должны следовать обратному порядку, указанном в обходе ЛПК). Этот элемент является правым поддеревом дерева со значением

предыдущего символа строки *traversal_LKP*. Действия повторяются рекурсивно, пока не будут перечислены все символы строк.

Возвращаемое значение: 1 – если возникла ошибка, 0 – если ошибок не возникло и дерево построено.

8) *void treeDrawing(BinTree* binTree)*

Функция выводит изображение дерева на экран (пример вывода представлен в приложении А).

Параметры:

- **binTree**: указатель на дерево, которое необходимо вывести.

Описание алгоритма:

Сначала дополненное до полного нулевыми элементами исходное дерево «рисуется» в двумерном массиве, причём так, что в каждом столбце находится только один узел, для чего вызывается вспомогательная рекурсивая функция *void addUnitsInGrid*(). Элементами массива будут являются:

0 – пустая область

1 – ветвь к левому поддереву

2 – ветвь к правому поддереву

иное значение – элемент дерева

После чего происходит оптимизация изображения: нулевые столбцы удаляются. По данным получившейся таблицы символами на экран выводится бинарное дерево.

9) *void addUnitsInGrid(BinTree* binTree, char** grid, int level, int deep_level, int horizontalIndex)*

Вспомогательная функция для функции *void treeDrawing*(), которая строит «изображение» дерева в двумерном массиве.

Параметры:

- **binTree**: указатель на дерево.

- **grid**: указатель на двумерный массив, в котором будет хранится изображение дерева.
- **level**: глубина текущего элемента дерева.
- **deep_level**: глубина дерева *binTree*.
- **horizontalIndex**: индекс текущего элемента по горизонтали

9) *void print_traversal_LPK(BinTree* binTree)*

Функция выводит на экран перечисление элементов дерева в порядке ЛПК.

Параметры:

- **binTree**: бинарное дерево, элементы которого необходимо перечислить.

Описание алгоритма: если указатель *binTree* не равен нулю, то функция вызывается рекурсивно для левого и правого поддерева, после чего на экран выводится значение корня *binTree*.

10) *void freeMemory(BinTree* binTree)*

Функция очищает память, которая была выделена динамически для бинарного дерева.

Параметры:

- **binTree**: указатель на корень дерева.

Тестирование программы.

Было создано несколько тестов для проверки работы программы. Помимо тестов, демонстрирующих работу алгоритма, были написаны тесты, содержащие некорректные данные, для демонстрации вывода сообщений об ошибках введенных данных (см. Приложение А).

Вывод.

В ходе выполнения работы была изучена новая структура данных – бинарное дерево. Получены навыки по работе с бинарным деревом, разными обходами двоичных деревьев.

Приложение А. Тестирование.

Демонстрация работы программы:

Test 1:

```
abdefcglkm
edbfagckml
```

output:

```
Программа, используя перечисления узлов дерева в порядке КЛП и ЛКП:
а) восстанавливает дерево и выводит его изображение.
б) перечисляет узлы дерева в порядке ЛПК.

Перечислите узлы дерева в порядке КЛП (не больше 500 символов):
Перечислите узлы дерева в порядке КЛП (не больше 500 символов):

Алгоритм восстановления дерева по КЛП и ЛКП:

Шаг 1:
       КЛП: (a)bdefcglkm
       ЛКП: (e)dbfagckml
       'a' - корень дерева.
Шаг 2:
       КЛП: a(b)defcglkm
       ЛКП: (e)dbfagckml
       'b' - корень левого поддерева 'a'.
Шаг 3:
       КЛП: ab(d)efcglkm
       ЛКП: (e)dbfagckml
       'd' - корень левого поддерева 'b'.
Шаг 4:
       КЛП: abd(e)fcglkm
       ЛКП: (e)dbfagckml
       'e' - корень левого поддерева 'd'.
Шаг 5:
       КЛП: abde(f)cglkm
       ЛКП: edb(f)agckml
```

```
        'f' - корень правого поддерева 'b'.
Шаг 6:
        КЛП: abdef(c)glkm
        ЛКП: edbfa(g)ckml
        'c' - корень правого поддерева 'a'.
Шаг 7:
        КЛП: abdefc(g)lkm
        ЛКП: edbfa(g)ckml
        'g' - корень левого поддерева 'c'.
Шаг 8:
        КЛП: abdefcg(l)km
        ЛКП: edbfagc(k)ml
        'l' - корень правого поддерева 'c'.
Шаг 9:
        КЛП: abdefcgl(k)m
        ЛКП: edbfagc(k)ml
        'k' - корень левого поддерева 'l'.
Шаг 10:
         КЛП: abdefcglk(m)
         ЛКП: edbfagck(m)l
         'm' - корень правого поддерева 'k'.

Дерево построено.
```

```
Изображение дерева:

      ------a------
      |           |
   ---b---     ---c---------
   |     |     |           |
---d     f     g     ------l
|                    |
e                    k---
                        |
                        m

Обход ЛПК для данного дерева:
edfbgmklca
```

Пояснение к выводу программы:

После того, как пользователь ввёл две строки *traversal_KLP* (обход в порядке КЛП) и *traversal_LKP* (обход в порядке ЛКП), программа восстанавливает по ним бинарное дерево. Процесс обработки строк выводится на экран – на каждом шаге показывается, какие символы обрабатываются в данный момент и результат очередного шага, например:

```
Шаг <4>:
       КЛП: abd(e)fcglkm
       ЛКП: (e)dbfagckml
       'e' - корень левого поддерева 'd'.
```

После чего на экран выводится изображение дерева, построенное из символов, и обход этого дерева в порядке ЛПК.

Тестирование:

Test 2:

```
abfgcdklnoemywzx
fbganlokdceywmxz
```

output:

```
Изображение дерева:

   ------a------------------
   |                       |
---b---                 ---c---
|     |                 |     |
f     g              ---d     e---------
                     |                 |
                ------k            ------m------
                |                  |           |
             ---l---               y---      ---z
             |     |                  |      |
             n     o                  w      x

Обход ЛПК для данного дерева:
fgbnolkdwyxzmeca
```

Test 3:

```
abdktulvxemzynwhcfoijpq1gr23s45
tkudvlxbzmyewnhaiojfqp1c2r3g4s5
```

output:

```
Изображение дерева:

                  ------------------------a-----------------------
                  |                                               |
        ------------b------------                       ------------c------------
        |                       |                       |                       |
   ------d------           ------e------           ------f------           ------g------
   |           |           |           |           |           |           |           |
---k---     ---l---     ---m---     ---n---     ---o---     ---p---     ---r---     ---s---
|     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
t     u     v     x     z     y     w     h     i     j     q     1     2     3     4     5

Обход ЛПК для данного дерева:
tukvxldzymwhnebijoq1pf23r45sgca
```

Test 4:

```
a
a
```

output:

```
Изображение дерева:
a

Обход ЛПК для данного дерева:
a
```

Test 5:

```
1234567ab
ba7654321
```

output:

```
Изображение дерева:

                    ---1
                    |
                ---2
                |
            ---3
            |
        ---4
        |
    ---5
    |
---6
|
---7
|
---a
|
b
```

```
Обход ЛПК для данного дерева:
ba7654321
```

Test 6:

abcdef

abcdef

output:

```
Изображение дерева:

a---
   |
   b---
      |
      c---
         |
         d---
            |
            e---
               |
               f

Обход ЛПК для данного дерева:
fedcba
```

Test 7:

abcefklmdr

aeclkmfbrd

output:

```
Изображение дерева:

a--------------------
                     |
      ---------------b------
      |                    |
   ---c------------     ---d
   |              |     |
   e         ------f     r
             |
          ---k---
          |     |
          l     m

Обход ЛПК для данного дерева:
elmkfcrdba
```

Test 8:

```
abdsajk23
djsakljdkasm9
```

output:

```
Ошибка: введены некорректные данные.
Длина обхода КЛП - 9, длина обхода ЛКП - 13.
```

Test 9:

```
kla!8*z-- -!l
ldakl/.11klaf
```

output:

```
Ошибка: введены некорректные данные.
Символ №4 в обходе КЛП - '!'.
Символ №6 в обходе КЛП - '*'.
Символ №6 в обходе ЛКП - '/'.
Символ №7 в обходе ЛКП - '.'.
Символ №8 в обходе КЛП - '-'.
```

```
Символ №9 в обходе КЛП - '-'.
Символ №10 в обходе КЛП - ' '.
Символ №11 в обходе КЛП - '-'.
Символ №12 в обходе КЛП - '!'.
Строки должны состоять только из цифр и латинских букв.
```

Test 10:

```
aafjkdf1vwa
1lkd1lllvwl
```

output:

```
Ошибка: введены некорректные данные.
Узел 'a' встречается в обходе КЛП больше одного раза.
Узел '1' встречается в обходе ЛКП больше одного раза.
Узел 'l' встречается в обходе ЛКП больше одного раза.
Узел 'f' встречается в обходе КЛП больше одного раза.
```

Test 11:

```
abc123efxzq
fl421cdaqxz
```

output:

```
Ошибка: введены некорректные данные.
Узел 'b' встречается в обходе КЛП, но отсутствует в обходе ЛКП.
Узел 'l' встречается в обходе ЛКП, но отсутствует в обходе КЛП.
Узел '4' встречается в обходе ЛКП, но отсутствует в обходе КЛП.
Узел '3' встречается в обходе КЛП, но отсутствует в обходе ЛКП.
Узел 'e' встречается в обходе КЛП, но отсутствует в обходе ЛКП.
Узел 'd' встречается в обходе ЛКП, но отсутствует в обходе КЛП.
```

Test 12:

output:

```
Вы ввели пустое дерево.
```

Test 13:

abcdefx

abcfdex

output:

```
Программа, используя перечисления узлов дерева в порядке КЛП и ЛКП:
а) восстанавливает дерево и выводит его изображение.
б) перечисляет узлы дерева в порядке ЛПК.

Перечислите узлы дерева в порядке КЛП (не больше 500 символов):
Перечислите узлы дерева в порядке КЛП (не больше 500 символов):

Алгоритм восстановления дерева по КЛП и ЛКП:

Шаг 1:
       КЛП: (a)bcdefx
       ЛКП: (a)bcfdex
       'a' - корень дерева.
Шаг 3:
       КЛП: a(b)cdefx
       ЛКП: a(b)cfdex
       'b' - корень правого поддерева 'a'.
Шаг 4:
       КЛП: ab(c)defx
       ЛКП: ab(c)fdex
       'c' - корень правого поддерева 'b'.
Шаг 5:
       КЛП: abc(d)efx
       ЛКП: abc(f)dex
       'd' - корень правого поддерева 'c'.
Шаг 6:
       КЛП: abcd(e)fx
       ЛКП: abc(f)dex
       'e' - корень левого поддерева 'd'.
Шаг 7:
       КЛП: abcde(f)x
```

ЛКП: abc(f)dex

'f' - корень левого поддерева 'e'.

Ошибка: обходы КЛП и ЛПК не соответствуют одному дереву.

Корень 'e' следует после корня 'd' в обоих обходах при обходе в сторону корня.

Приложение Б. Файл main.c.

```
// По перечислению узлов дерева в порядке КЛП и ЛКП:
// а) восстановить дерево и вывести его изображение.
// б) перечислить узлы дерева в порядке ЛПК.


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "binTree.h"


// Максимальный размер строки.
#define STR_LENGTH 501
// Направления для восстановления дерева по КЛП И ЛКП.
#define LEFT 1
#define UP 2
#define RIGHT 3




// Печать пробелов (для демонстрации работы алгоритма).
// Количество пробелов = количеству знаков строки:
// "Шаг N: ", где N - номер шага.
void printSpaces(int a){
    int count = 6;
    for (int i = 1; a%i < a; i*=10)
        count++;
    for (int i = 0; i < count; i++)
        putchar(' ');
}


// Вывод строк КЛП и ЛКП с указанием,
// какие узлы обрабатываются на текущем шаге
// (для демонстрации работы алгоритма).
void printCurrentStep(char* traversal_KLP, char* traversal_LKP, int index_KLP, int index_LKP, int
counter){
    printSpaces(counter);
    printf("КЛП: ");
    for (int i = 0; i < strlen(traversal_KLP)-1; i++){
        if (i == index_KLP)
```

```
            printf("(%c)", traversal_KLP[i]);
        else
            printf("%c", traversal_KLP[i]);
    }
    putchar('\n');
    printSpaces(counter);
    printf("ЛКП: ");
    for (int i = 0; i < strlen(traversal_LKP)-1; i++){
        if (i == index_LKP)
            printf("(%c)", traversal_LKP[i]);
        else
            printf("%c", traversal_LKP[i]);
    }
    putchar('\n');
}
```

```
// Восстановление дерева из КЛП и ЛКП.
// возвращает: 1 - возникли ошибки, 0 - строки валидны
int treeRecovering(BinTree** binTree, char* traversal_KLP, char* traversal_LKP){
    int index_KLP = 0;
    int index_LKP = 0;
    int counter = 0; // счетчик для разных нужд
    int direction; // направление обхода для восстановления дерева
    int str_len; // длина строк обхода

    putchar('\n'); // Для красивого вывода :-)

    // Проверка на ревенство длин строк КЛП и ЛКП.
    if (strlen(traversal_KLP) != strlen(traversal_LKP)){
        printf("Ошибка: введены некорректные данные.\n");
        printf("Длина обхода КЛП - %zu, длина обхода ЛКП - %zu.\n\n", strlen(traversal_KLP)-1,
strlen(traversal_LKP)-1);
        return 1;
    }
    str_len = strlen(traversal_KLP) - 1;

    // Проверка на то, что строки не пустые
    if (str_len == 0){
        printf("Вы ввели пустое дерево.\n\n");
        return 1;
```

```
    }

    // Проверка на использование недопустимых символов.
    for (int i = 0; i < str_len; i++){
        if (!isdigit(traversal_KLP[i]) && !isalpha(traversal_KLP[i])){
            if (!counter)
                printf("Ошибка: введены некорректные данные.\n");
            printf("Символ №%d в обходе КЛП - '%c'.\n", i+1, traversal_KLP[i]);
            counter++;
        }
        if (!isdigit(traversal_LKP[i]) && !isalpha(traversal_LKP[i])){
            if (!counter)
                printf("Ошибка: введены некорректные данные.\n");
            printf("Символ №%d в обходе ЛКП - '%c'.\n", i+1, traversal_LKP[i]);
            counter++;
        }
    }
    if (counter){
        printf("Строки должны состоять только из цифр и латинских букв.\n\n");
        return 1;
    }


    // Проверка на отсутствие повторяющихся узлов в каждом обходе.
    for (int i = 0; i < str_len; i++){
        for (int j = i; j >= 0; j--){
            if (traversal_KLP[i] == traversal_KLP[j] && i != j){
                break;
            }
            if (j == 0){
                for (int k = i; k < str_len; k++){
                    if (traversal_KLP[i] == traversal_KLP[k] && i != k){
                        if (!counter)
                            printf("Ошибка: введены некорректные данные.\n");
                        printf("Узел '%c' встречается в обходе КЛП больше одного раза.\n",
traversal_KLP[i]);
                        counter++;
                        break;
                    }
                }
            }
        }
        for (int j = i; j >= 0; j--){
            if (traversal_LKP[i] == traversal_LKP[j] && i != j){
                break;
```

```
            }
            if (j == 0){
                for (int k = i; k < str_len; k++){
                    if (traversal_LKP[i] == traversal_LKP[k] && i != k){
                        if (!counter)
                            printf("Ошибка: введены некорректные данные.\n");
                        printf("Узел '%c' встречается в обходе ЛКП больше одного раза.\n",
traversal_LKP[i]);
                        counter++;
                        break;
                    }
                }
            }
        }
    }
    if (counter){
        putchar('\n'); // Для красивого вывода :-)
        return 1;
    }

    // Проверка на соответствие узлов в КЛП и ЛКП.
    for (int i = 0; i < str_len; i++){
        for (int j = 0; j < str_len; j++){
            if (traversal_KLP[i] == traversal_LKP[j]){
                break;
            }
            if (j == str_len - 1){
                if (!counter)
                    printf("Ошибка: введены некорректные данные.\n");
                printf("Узел '%c' встречается в обходе КЛП, но отсутствует в обходе ЛКП.\n",
traversal_KLP[i]);
                counter++;
            }
        }
        for (int j = 0; j < str_len; j++){
            if (traversal_LKP[i] == traversal_KLP[j]){
                break;
            }
            if (j == str_len - 1){
                if (!counter)
                    printf("Ошибка: введены некорректные данные.\n");
                printf("Узел '%c' встречается в обходе ЛКП, но отсутствует в обходе КЛП.\n",
traversal_LKP[i]);
                counter++;
            }
```

```
        }
    }
    if (counter){
        putchar('\n'); // Для красивого вывода :-)
        return 1;
    }


    // Создание дерева.
    direction = LEFT;
    counter = 1;
    printf("Алгоритм восстановления дерева по КЛП и ЛКП:\n\n");

    printf("Шаг %d:\n", counter);
    printCurrentStep(traversal_KLP, traversal_LKP, index_KLP, index_LKP, counter);
    printSpaces(counter);
    printf("'%c' - корень дерева.\n", traversal_KLP[index_KLP]);
    *binTree = initBinTree(traversal_KLP[index_KLP]);
    counter++;
    if (traversal_KLP[index_KLP] == traversal_LKP[index_LKP]){
        direction = UP;
    }
    index_KLP++;

    while(index_KLP < str_len){
        if (direction == LEFT){
            printf("Шаг %d:\n", counter);
            printCurrentStep(traversal_KLP, traversal_LKP, index_KLP, index_LKP, counter);
            printSpaces(counter);
            printf("'%c' - корень левого поддерева '%c'.\n", traversal_KLP[index_KLP],
traversal_KLP[index_KLP-1]);
            findBinTree(*binTree, traversal_KLP[index_KLP-1])->leftTree =
initBinTree(traversal_KLP[index_KLP]);
            if (traversal_KLP[index_KLP] == traversal_LKP[index_LKP]){
                direction = UP;
            }
            index_KLP++;
        } else if (direction == UP){
            for (int i = 0; i < index_KLP; i++){
                if (traversal_LKP[index_LKP] == traversal_KLP[i]){
                    index_LKP++;
                    break;
                }
                if (i == index_KLP-1){
                    direction = RIGHT;
```

```
                }
            }
        } else if (direction == RIGHT){
            printf("Шаг %d:\n", counter);
            printCurrentStep(traversal_KLP, traversal_LKP, index_KLP, index_LKP, counter);
            printSpaces(counter);
            printf("'%c' - корень правого поддерева '%c'.\n", traversal_KLP[index_KLP],
traversal_LKP[index_LKP-1]);
            findBinTree(*binTree, traversal_LKP[index_LKP-1])->rightTree =
initBinTree(traversal_KLP[index_KLP]);
            if (traversal_KLP[index_KLP] != traversal_LKP[index_LKP]){
                direction = LEFT;
            }
            else {
                for (int i = 0; i < index_KLP; i++){
                    if (traversal_KLP[i] == traversal_LKP[index_LKP+1]){
                        direction = UP;
                        break;
                    }
                }
                index_LKP++;
            }
            index_KLP++;
        }
        if (direction != UP){
            counter++;
        }
    }

    return 0;
}




// Вывод изображения дерева на экран.
// Создание сетки дерева для дальнейшей оптимизации рисунка.
void addUnitsInGrid(BinTree* binTree, char** grid, int level, int deep_level, int horizontalIndex){
    grid[level-1][horizontalIndex] = binTree->a;
    int a = 1;
    for (int i = 0; i < deep_level-level-1; i++)
        a*=2;
    if (level < deep_level){
        for (int i = 1; i <= a; i++){
            grid[level-1][horizontalIndex-i] = LEFT;
```

```
            grid[level-1][horizontalIndex+i] = RIGHT;
        }
    }
    if (leftBinTree(binTree) != NULL){
        addUnitsInGrid(leftBinTree(binTree), grid, level+1, deep_level, horizontalIndex - a);
    }
    if (rightBinTree(binTree) != NULL){
        addUnitsInGrid(rightBinTree(binTree), grid, level+1, deep_level, horizontalIndex + a);
    }
}
// Вывод изображения дерева.
void treeDrawing(BinTree* binTree){
    if (binTree == NULL){
        printf("Вы ввели пустое дерево.\n");
    }
    if (leftBinTree(binTree) == NULL && rightBinTree(binTree) == NULL){
        printf("%c\n\n", binTree->a);
        return;
    }
    // В начале "нарисую" дерево в двумерном массиве.
    int deep_level = binTreeDeepLevel(binTree); // глубина дерева
    int gridWidth = 1; // ширина сетки
    for (int i = 0; i < deep_level; i++)
        gridWidth*=2;
    gridWidth--;
    int horizontalTab = 0; // расстояние между узлами по горизонтали


    // Создам и заполню сетку нулями.
    char** grid = (char**)malloc(sizeof(char*)*deep_level);
    for (int i = 0; i < deep_level; i++){
        grid[i] = (char*)malloc(sizeof(char)*gridWidth);
        for (int j = 0; j < gridWidth; j++){
            grid[i][j] = 0;
        }
        horizontalTab = horizontalTab*2+1;
    }
    // Внесение в сетку узлов дерева и веток.
    addUnitsInGrid(binTree, grid, 1, deep_level, horizontalTab/2);
    // Оптимизация сетки.
    for (int j = 0; j < gridWidth; j++){
        for (int i = 0; i < deep_level; i++){
            if (grid[i][j] != 0 && grid[i][j] != LEFT && grid[i][j] != RIGHT)
                break;
            if (i == deep_level-1){
```

```
            for (int k = j; k < gridWidth-1; k++){
                for (int l = 0; l < deep_level; l++){
                    grid[l][k] = grid[l][k+1];
                }
            }
            gridWidth--;
            j--;
        }
    }
}


// Построение дерева по сетке.
for (int i = 0; i < deep_level; i++){
    for (int j = 0; j < gridWidth; j++){
        if (grid[i][j] == 0)
            printf("   ");
        if (grid[i][j] == LEFT)
            printf("---");
        if (grid[i][j] == RIGHT){
            if (grid[i+1][j] != 0 && grid[i+1][j] != LEFT && grid[i+1][j] != RIGHT)
                printf("-  ");
            else
                printf("---");
        }
        if (grid[i][j] != 0 && grid[i][j] != LEFT && grid[i][j] != RIGHT){
            if (j < gridWidth-1 && grid[i][j+1] == RIGHT)
                printf("%c--", grid[i][j]);
            else
                printf("%c  ", grid[i][j]);
        }
    }
    putchar('\n');
    for (int j = 0; j < gridWidth; j++){
        if (grid[i][j] == 0 || (grid[i][j] != LEFT && grid[i][j] != RIGHT))
            printf("   ");
        if (grid[i][j] == LEFT || grid[i][j] == RIGHT){
            if (grid[i+1][j] != 0 && grid[i+1][j] != LEFT && grid[i+1][j] != RIGHT)
                printf("|  ");
            else
                printf("   ");
        }
    }
    putchar('\n');
}
```

```
    // Освобождение памяти.
    for (int i = 0; i < deep_level; i++)
        free(grid[i]);
    free(grid);
}


// Перечисление узлов дерева в порядке ЛПК.
void print_traversal_LPK(BinTree* binTree){
    if (binTree != NULL){
        print_traversal_LPK(leftBinTree(binTree));
        print_traversal_LPK(rightBinTree(binTree));
        putchar(binTree->a);
    }
}




// Освобождение памяти.
void freeMemory(BinTree* binTree){
    if(binTree != NULL){
        freeMemory(leftBinTree(binTree));
        freeMemory(rightBinTree(binTree));
        free(binTree);
    }
}




int main(){
    char traversal_KLP[STR_LENGTH]; // обход дерева в порядке КЛП
    char traversal_LKP[STR_LENGTH]; // обход дерева в порядке ЛКП
    BinTree* binTree = NULL; // бинарное дерево

    // приветствие
    printf("\nПрограмма, используя перечисления узлов дерева в порядке КЛП и ЛКП:\n");
    printf("а) восстанавливает дерево и выводит его изображение.\n");
    printf("б) перечисляет узлы дерева в порядке ЛПК.\n");
```

```
    // считывание строк КЛП и ЛКП
    printf("\nПеречислите узлы дерева в порядке КЛП (не больше %d символов):\n", STR_LENGTH-1);
    fgets(traversal_KLP, STR_LENGTH, stdin);
    printf("Перечислите узлы дерева в порядке КЛП (не больше %d символов):\n", STR_LENGTH-1);
    fgets(traversal_LKP, STR_LENGTH, stdin);

    // Восстановление дерева.
    if (treeRecovering(&binTree, traversal_KLP, traversal_LKP) == 0){
        printf("\nДерево построено.\n");
        // Вывод изображения и обхода ЛПК.
        printf("\nИзображение дерева:\n\n");
        treeDrawing(binTree);
        printf("Обход ЛПК для данного дерева:\n");
        print_traversal_LPK(binTree);
        printf("\n\n");
    }

    // Освобождение памяти.
    freeMemory(binTree);

    return 0;
}
```

Приложение В. Файл binTree.c.

```c
#include <stdlib.h>
#include "binTree.h"



// Возврат левого поддерева.
BinTree* leftBinTree(BinTree* binTree){
    if (binTree != NULL)
        return binTree->leftTree;
    else
        return NULL;
}


// Возврат правого поддерева.
BinTree* rightBinTree(BinTree* binTree){
    if (binTree != NULL)
        return binTree->rightTree;
    else
        return NULL;
}


// Создание дерева.
BinTree* initBinTree(char a){
    BinTree* binTree = (BinTree*)malloc(sizeof(BinTree));
    binTree->a = a;
    binTree->leftTree = NULL;
    binTree->rightTree = NULL;
    return binTree;
}


// Поиск по дереву.
BinTree* findBinTree(BinTree* binTree, char a){
    if (binTree == NULL)
        return NULL;
    if (binTree->a == a)
        return binTree;
    if (findBinTree(leftBinTree(binTree), a) == NULL)
        return findBinTree(rightBinTree(binTree), a);
    else
```

```
        return findBinTree(leftBinTree(binTree), a);
}


// Глубина дерева.
int binTreeDeepLevel(BinTree* binTree){
    if (binTree == NULL)
        return 0;
    int leftDeep = binTreeDeepLevel(leftBinTree(binTree));
    int rightDeep = binTreeDeepLevel(rightBinTree(binTree));
    return 1 + (leftDeep > rightDeep ? leftDeep : rightDeep);
}
```

Приложение Г. Файл binTree.h.

```
#pragma once

// Структура бинарного дерева.
typedef struct BinTree{
    char a;
    struct BinTree* leftTree;
    struct BinTree* rightTree;
} BinTree;


// Возврат левого поддерева.
BinTree* leftBinTree(BinTree*);


// Возврат правого поддерева.
BinTree* rightBinTree(BinTree*);


// Создание дерева.
BinTree* initBinTree(char);


// Поиск по дереву.
BinTree* findBinTree(BinTree*, char);


// Глубина дерева.
int binTreeDeepLevel(BinTree*);
```