МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Случайные БДП – вставка и исключение. Текущий контроль.

Студент гр. 7381 _____ Адамов Я.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2018

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Адамов Я.В.

Группа 7381

Тема работы: Случайные БДП - вставка и исключение. Текущий контроль.

Исходные данные:
На вход программе подаются следующие данные: количество заданий, которое необходимо сгенерировать, минимальное и максимальное количество элементов в исходном дереве, минимальное и максимальное количество операций: вставка и исключение, – которое должно выполняться над деревом.

Содержание пояснительной записки:
«Содержание», «Аннотация», «Введение», «Постановка задачи», «Алгоритм работы программы», «Функции и структуры данных», «Пользовательский интерфейс», «Тестирование», «Заключение».

Предполагаемый объём объяснительной записки:
Не менее 20 страниц.

Дата сдачи реферата:

Дата защиты реферата:

Студент _____ Адамов Я.В.

Преподаватель _____ Фирсов М.А.

СОДЕРЖАНИЕ

АННОТАЦИЯ

В данной курсовой работе реализован генератор заданий и ответов к ним на тему «Случайные БДП – вставка и исключение».

Для выполнения этой задачи были реализованы БДП и основные функции для работы с ним: вставка и удаление элемента из дерева, поиск необходимого и минимального элемента дерева, вывод изображения дерева. Реализован удобный и понятный интерфейс, с помощью которого пользователь может настроить сложность генерируемых задач и увидеть процесс генерации.

В качестве языка программирования для создания программы был выбран язык программирования C.

ВВЕДЕНИЕ

Данная курсовая работа основана на работе с бинарным деревом поиска.

Бинарное дерево поиска (БДП) — это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- Оба поддерева — левое и правое — являются БДП.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X.

Пример БДП представлен на рис. 1.

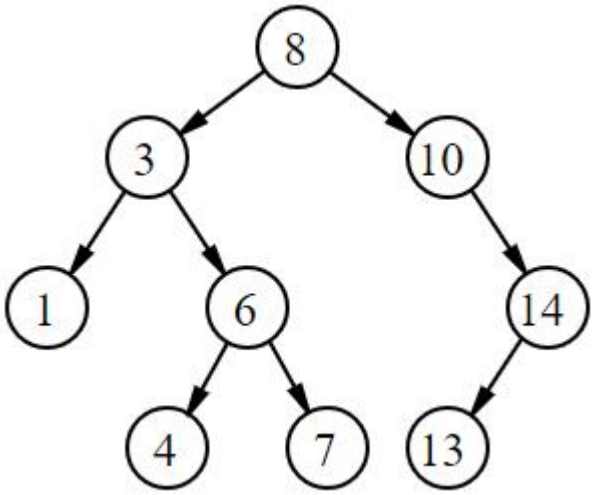


Рисунок 1 – Пример БДП

ПОСТАНОВКА ЗАДАЧИ

Реализация программы для генерации заданий с ответами к ним на тему «Случайные БДП – вставка и исключение» для проведения текущего контроля среди студентов. Задания и ответы должны выводиться в файл в удобной форме: тексты заданий должны быть готовы для передачи студентам, проходящим ТК; ответы должны позволять удобную проверку правильности выполнения заданий.

Программа должна предоставить пользователю настроить сложность генерируемых заданий, а также демонстрировать процесс генерации на экране.

АЛГОРИТМ РАБОТЫ ПРОГРАММЫ

На вход программе подаются следующие данные: количество заданий, которое необходимо сгенерировать (не больше 100), минимальное и максимальное количество элементов в исходном дереве (в пределах от 3 до 26), минимальное и максимальное количество операций: вставка и исключение, – которое должно выполняться над деревом (в пределах от 1 до 10). Если пользователь ввёл некорректные данные, программа предлагает ввести их ещё раз.

Описание алгоритма генерации заданий:

Генерируется два случайных числа *treeSize* – размер исходного БДП, и *operationCount* – количество операций, которое необходимо выполнить над исходным деревом (в пределах, заданных пользователем). Генерируется *treeSize* различных случайных значений, по которой необходимо построить исходное бинарное дерево, и помещаются в строку *task*. Затем *operationCount* раз генерируется пара значение – номер операции (вставка – 1 или исключение – 2) и элемент, который необходимо вставить или удалить.

Пример получившийся строки *task*: *admfd1z2m1e2e*.

Расшифровка: построить БДП *admfd*, добавить элемент *z*, удалить элемент *m*, добавить элемент *e*, удалить элемент *e*.

После генерации происходит проверка на то, что подобных заданий ещё не было сгенерировано (замечание: задания *abcd1e* и *klmn1o* являются аналогичными, так как в них различны только значения, но построения дерева будет одинаковым). Для этого для каждого теста создаётся его аналог, в котором все элементы принимают своё минимальное значение, то есть для задания *klmn1o* аналогом будет *abcd1e*. После генерации очередного задания идёт проверка этих аналогов, если копии не нашлось, значит сгенерированное задание является новым и добавляется в файл вывода *tasks.txt*.

После того, как задания сгенерированы, необходимо их решить и вывести результаты в файл *answers.txt*. Процесс решения заданий выводится на экран.

Описание алгоритма создания БДП по строке:

Программа поочередно обрабатывает символы строки. Первый символ является корнем дерева. Для каждого последующего символа выполняются следующие действия: если текущий обрабатываемый символ меньше ключа корня, то он должен принадлежать левому поддереву, если же больше – правому. Если необходимое поддерево является пустым, то текущий обрабатываемый символ становится ключом нового узла, который является корнем соответствующего правого или левого поддерева, в ином случае аналогичные действия совершаются относительно левого или правого корня поддерева.

Описание алгоритма вставки элемента в БДП:

Если добавляемый элемент меньше ключа корня, то он должен принадлежать левому поддереву, если же больше – правому. Если необходимое поддерево является пустым, то текущий обрабатываемый символ становится ключом нового узла, который является корнем соответствующего правого или левого поддерева, в ином случае аналогичные действия совершаются относительно левого или правого корня поддерева.

Описание алгоритма удаление элемента из БДП:

Программа находит элемент, который необходимо удалить. Если он является висячим (не имеет поддеревьев), то он просто удаляется. Если имеется одно поддерево, то это поддерево занимает место удаляемого элемента. Если имеется два поддерева, то место удаляемого элемента занимает минимальный элемент в правом поддереве.

Алгоритм поиска элемента в БДП:

Если искомый элемент меньше корня, то программа переходит в правое поддерево и рекурсивно повторяет действия, если больше – в левое, если равен – элемент найден.

Алгоритм поиска минимального элемента в БДП:

Если корень имеет левое поддерево, то программа переходит к нему и повторяет действия рекурсивно, в ином случае минимальный элемент найден.

Алгоритм вывода изображения дерева на экран:

Сначала дополненное до полного нулевыми элементами исходное дерево «рисуется» в двумерном массиве, причём так, что в каждом столбце находится только один узел. 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: столбцы, в которых нет узлов, удаляются. По данным получившейся таблицы символами на экран выводится бинарное дерево. Пример представлен в разделе «Пользовательский интерфейс».

ФУНКЦИИ И СТРУКТУРЫ ДАННЫХ

1) *struct BST*

Структура бинарного дерева поиска. Для работы с деревом используются следующие функции: leftBST, rightBST, initBST, BST_DeepLevel.

Поля структуры:

- **char key**: ключ.
- **struct BST* left**: указатель на левое поддерево.
- **struct BST* right**: указатель на правое поддерево.

2) *BST* initBST(char key)*

Инициализация бинарного дерева.

Возвращаемое значение: бинарное дерево, значение корня которого – key, указатели *left* и *right* имеют значение *NULL*.

3) *BST* leftBST(BST* tree)*

Функция возвращает указатель на левое поддерево.

Параметры:

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

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

4) *BST* rightBST(BST* tree)*

Функция возвращает указатель на правое поддерево.

Параметры:

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

Возвращаемое указатель на правое поддерево дерева *tree*.

5) *int BST_DeepLevel (BST* tree)*

Функция возвращает глубину дерева.

Параметры:

- **tree**: указатель на дерево.

Возвращаемое значение: глубина дерева *tree*.

6) *BST * BST_create(char * bst_str)*

Функция строит БДП по переданной строке с символами.

Параметры:

- **bst_str**: указатель на строку с последовательностью символов.

Возвращаемое значение: указатель на созданное БДП.

7) *void BST_draw(BST* tree)*

Функция выводит изображение дерева на экран (пример вывода представлен в разделе «Пользовательский интерфейс»).

Параметры:

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

8) *void addUnitsInGrid(BST* tree, char** grid, int level, int deep_level, int horizontalIndex)*

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

Параметры:

- **tree**: 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 freeMemory(BST* tree)*

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

Параметры:

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

10) *void add_element_in_BST(BST* tree, char key)*

Функция добавляет элемент в БДП.

Параметры:

- **tree**: БДП, в которое надо добавить элемент.
- **key**: элемент, который необходимо добавить.

11) *BST * remove_element_in_BST(BST* tree, char key)*

Функция удаляет элемент *key* из БДП, если он там имеется.

Параметры:

- **tree**: БДП, из которого надо удалить элемент.
- **key**: элемент, который необходимо удалить.

Возвращаемое значение: указатель на полученное дерево (если удалялся не корень, то указатель на исходное дерево).

12) *int getRandInt(int min, int max)*

Функция возвращает случайное целое число в диапазоне [min; max].

Параметры:

- **min**: минимальное значение генерируемого числа.
- **max**: максимальное значение генерируемого числа.

Возвращаемое значение: случайное целое число в диапазоне [min; max].

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС

В начале работы программа приветствует пользователя и предлагает настроить количество и сложность генерируемых заданий: минимальное и максимальное количество элементов в исходном дереве, минимальное и максимальное количество операций: вставка и исключение, – которое должно выполняться над деревом. Приветствие и настройка генератора представлены на рис. 2. Обработка некорректных данных представлена на рис. 3.

```
Программа генерирует задания с ответами на тему "Случайные БДП - вставка и исключение".

Введите количество заданий, которое необходимо сгенерировать (не больше 100): 20

Введите минимальное количество элементов в начальном дереве (не меньше 3, но не больше 26): 5

Введите максимальное количество элементов в начальном дереве (не меньше 5, но не больше 26): 13

Введите минимальное количество операций над деревом - вставка и исключение (не меньше 1, но не больше 10): 1

Введите максимальное количество операций над деревом - вставка и исключение (не меньше 1, но не больше 10): 4
```

Рисунок 2 – приветствие и настройка генератора

```
Программа генерирует задания с ответами на тему "Случайные БДП - вставка и исключение".

Введите количество заданий, которое необходимо сгенерировать (не больше 100): 200
Ошибка. Введите число, не превосходящее 100: выф
Ошибка. Введите целое неотрицательное число: 15

Введите минимальное количество элементов в начальном дереве (не меньше 3, но не больше 26): -15
Ошибка. Введите число не меньше 3, но не больше 26: 5

Введите максимальное количество элементов в начальном дереве (не меньше 5, но не больше 26): 28
Ошибка. Введите число не меньше 5, но не больше 26: 10

Введите минимальное количество операций над деревом - вставка и исключение (не меньше 1, но не больше 10): -5
Ошибка. Введите число не меньше 1, но не больше 10: 2

Введите максимальное количество операций над деревом - вставка и исключение (не меньше 2, но не больше 10): 4
```

Рисунок 3 – обработка некорректных данных

После настройки программа начинает генерировать задания, выводя процесс их решения на экран. Демонстрация вывода текста задания и построения БДП представлена на рис. 4. Демонстрация удаления элемента из БДП представлена на рис. 5. Демонстрация удаления элемента из БДП представлена на рис. 6. Вывод конечного результата представлен на рис. 7.

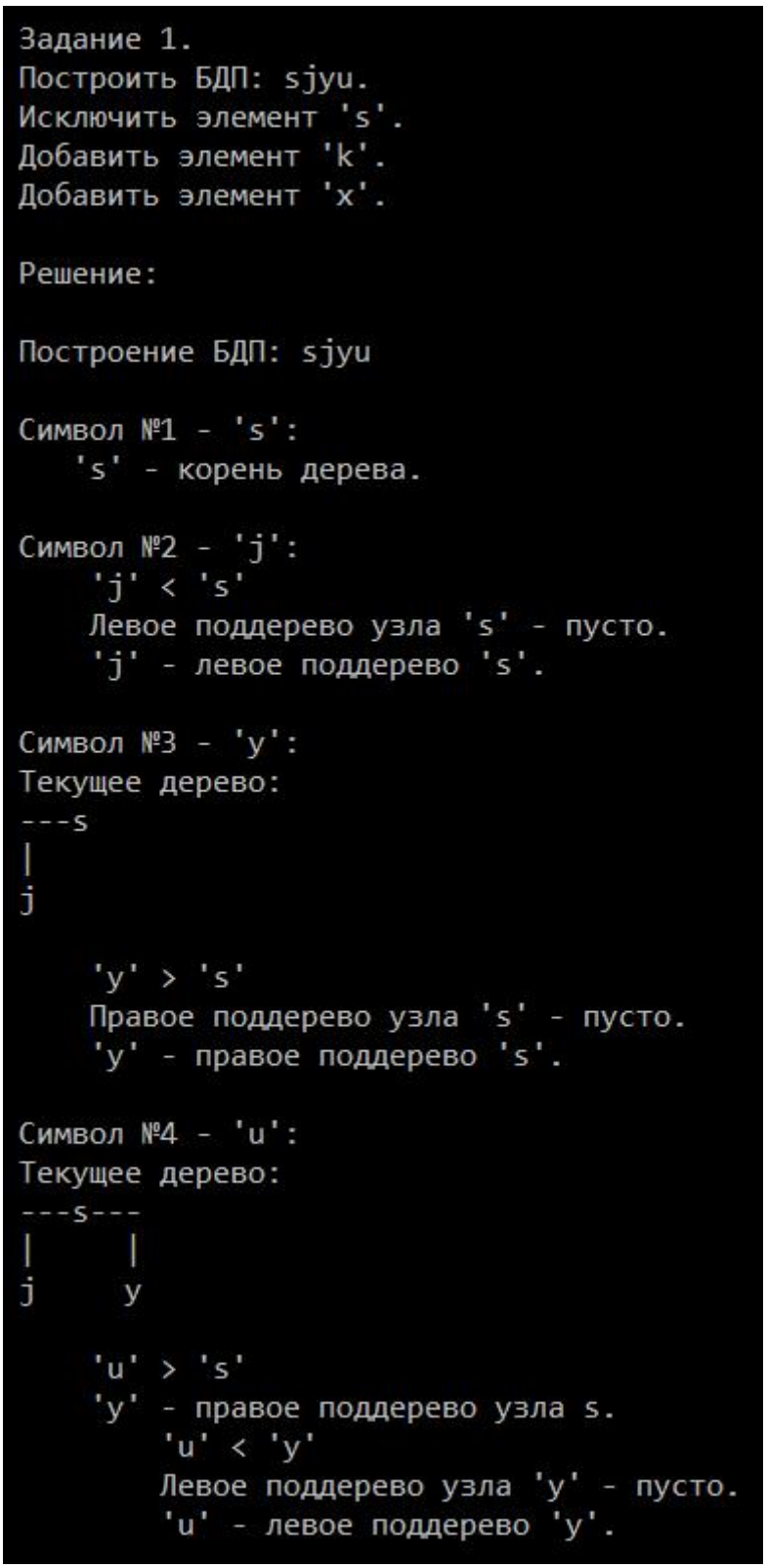


Рисунок 4 – вывод текста задания и построения БДП.

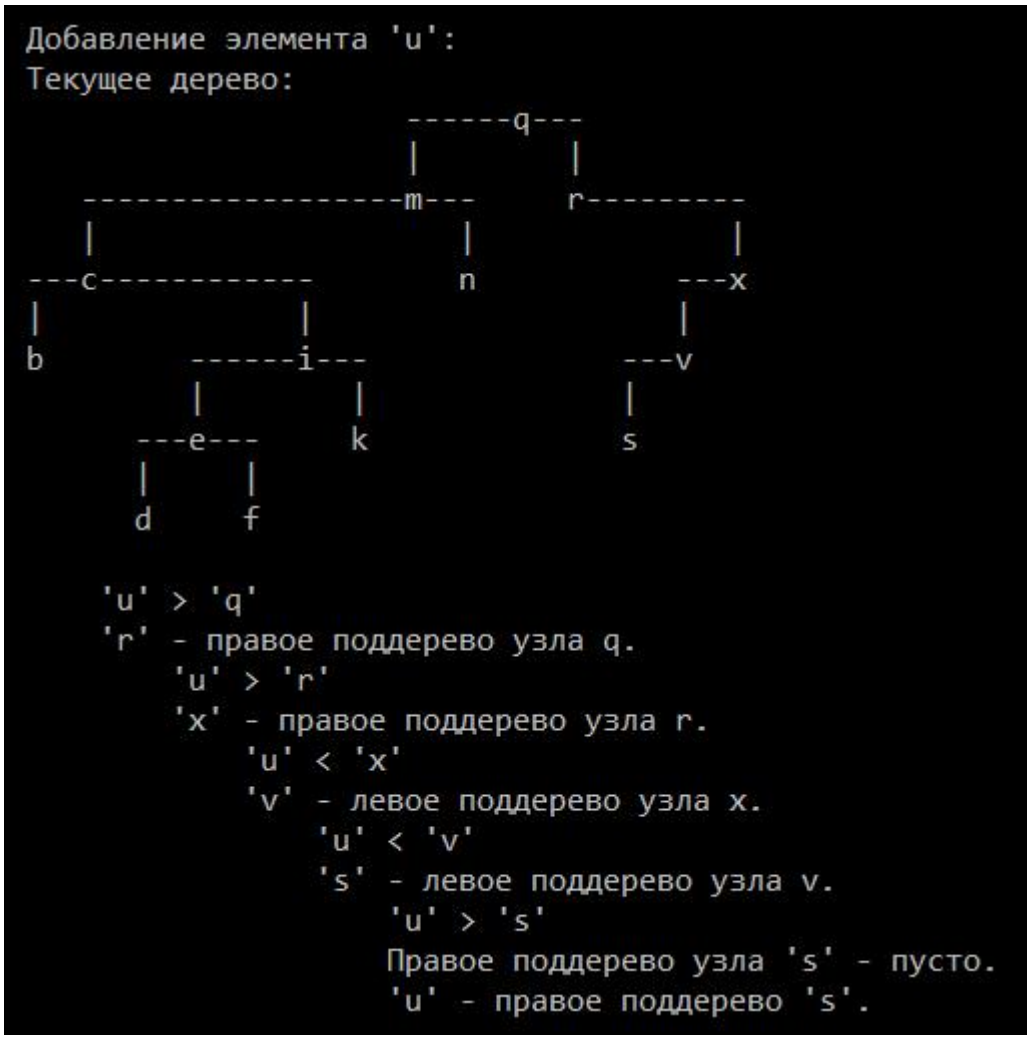


Рисунок 5 – добавление элемента в БДП

Рисунок 6 – удаление элемента из БДП

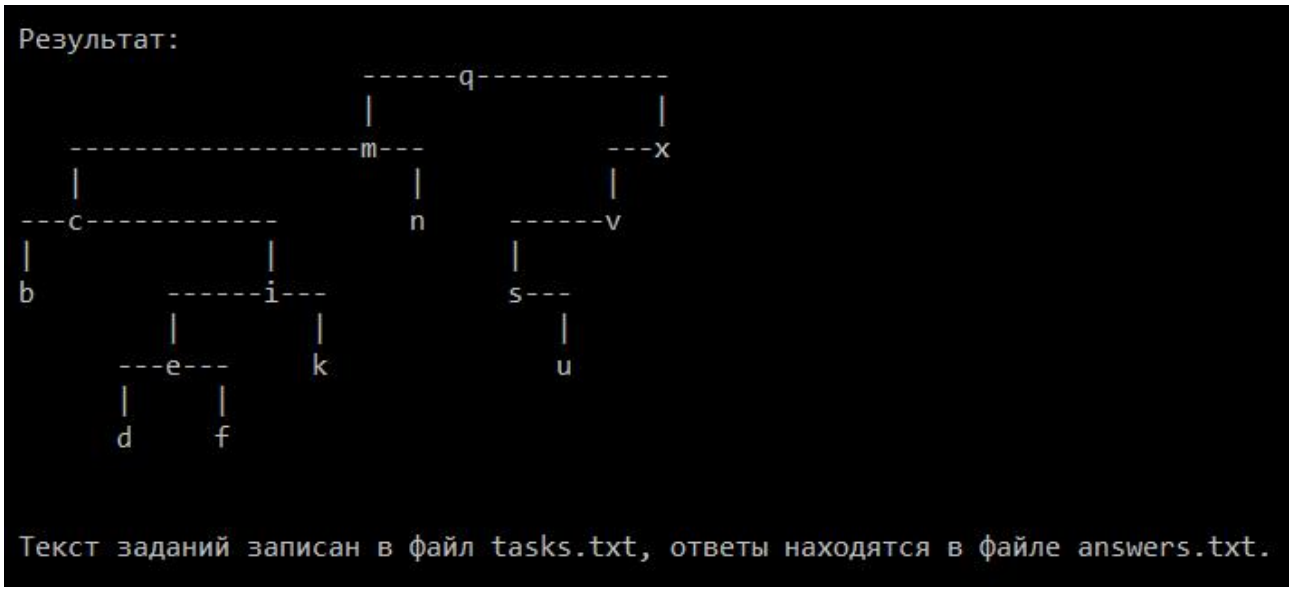


Рисунок 7 – вывод конечного результата

ТЕСТИРОВАНИЕ

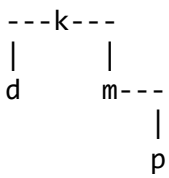
В качестве тестирование были сгенерированы тесты различных сложностей.

Содержание файла tasks.txt:

```
Задание 1.
Построить БДП: ycbjiwumn.
Исключить элемент 'i'.
Исключить элемент 'c'.
Добавить элемент 'i'.
Добавить элемент 'a'.

Задание 2.
Построить БДП: obyuxjhaefzirglcmdkwnpts.
Исключить элемент 'm'.
Добавить элемент 'm'.

Задание 3.
Построить БДП: kmzop.
Добавить элемент 'w'.
Исключить элемент 'o'.
Добавить элемент 'i'.
Исключить элемент 'w'.
Добавить элемент 'd'.
Исключить элемент 'i'.
Исключить элемент 'z'.

Задание 4.
Построить БДП: njpksxhazwfmoytive.
Исключить элемент 'n'.

Задание 5.
Построить БДП: sdg.
Добавить элемент 'm'.

Задание 6.
Построить БДП: ozgslpafjwekimqcthxbrvundy.
Исключить элемент 'z'.
Исключить элемент 'j'.
Добавить элемент 'j'.
Добавить элемент 'z'.
Исключить элемент 'e'.
Добавить элемент 'e'.
Исключить элемент 'r'.
Добавить элемент 'r'.
```

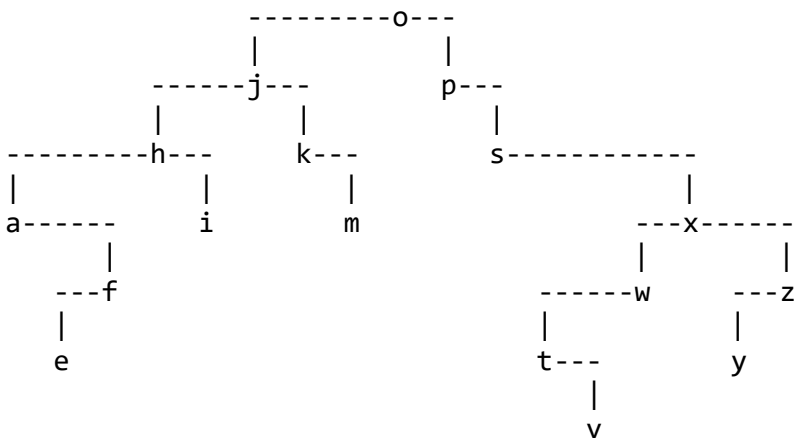
Содержание файла answers.txt:

```
Задание 1.
Построить БДП: ycbjiwumn.
Исключить элемент 'i'.
Исключить элемент 'c'.
Добавить элемент 'i'.
Добавить элемент 'a'.

Ответ:
         ---------------y
         |
   ------j------------
   |                |
---b---           ---w
|     |           |
a     i     ------u
            |
            m---
               |
               n


Задание 2.
Построить БДП: obyuxjhaefzirglcmdkwnpts.
Исключить элемент 'm'.
Добавить элемент 'm'.

Ответ:
   ----------------------------------------o-----------------------
   |                                                              |
---b------------------------                                  ---------y---
|                           |                                  |          |
a                    ------j------                ---------u------     z
                     |           |                |              |
             ---------h---     ---l------     ---r------     ---x
             |           |     |        |     |        |     |
      ------e---         i     k     ---n     p     ---t     w
      |        |                      |              |
      c---     f---                   m              s
         |        |
         d        g


Задание 3.
Построить БДП: kmzop.
Добавить элемент 'w'.
Исключить элемент 'o'.
Добавить элемент 'i'.
Исключить элемент 'w'.
Добавить элемент 'd'.
Исключить элемент 'i'.
Исключить элемент 'z'.

Ответ:
```



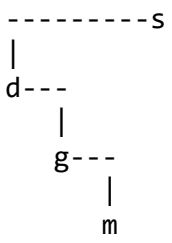
Задание 4.
Построить БДП: njpksxhazwfmoytive.
Исключить элемент 'n'.

Ответ:



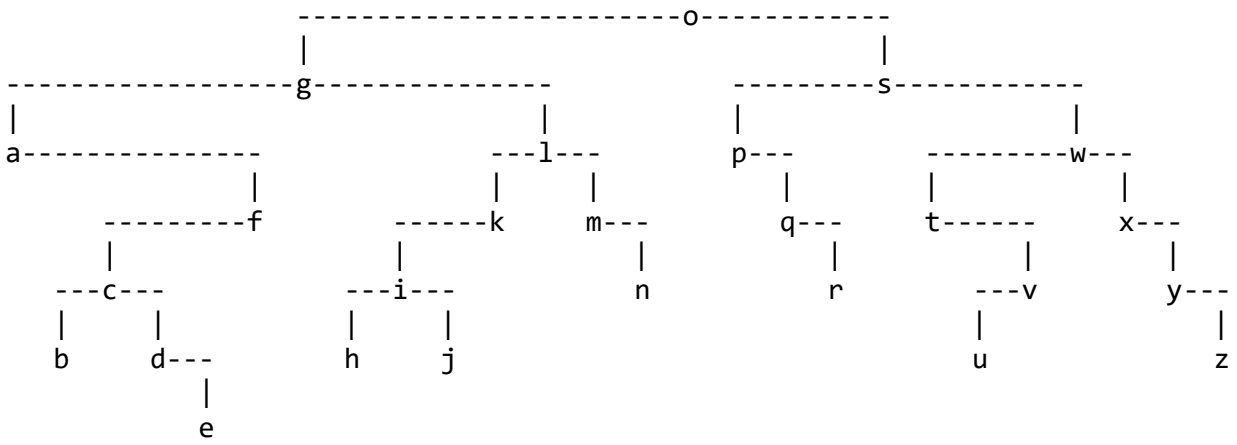
Задание 5.
Построить БДП: sdg.
Добавить элемент 'm'.

Ответ:



Задание 6.
Построить БДП: ozgslpafjwekimqcthxbrvundy.
Исключить элемент 'z'.
Исключить элемент 'j'.
Добавить элемент 'j'.
Добавить элемент 'z'.
Исключить элемент 'e'.
Добавить элемент 'e'.
Исключить элемент 'r'.
Добавить элемент 'r'.

Ответ:



ЗАКЛЮЧЕНИЕ

В ходе работы была создана программа для генерации заданий с ответами к ним для проведения текущего контроля среди студентов по теме «Случайные БДП – вставка и исключение».

В процессе работы были освоены и закреплены навыки работы с основными алгоритмами и структурами данных, используемыми при работе с бинарными деревьями поиска. Также были реализованы алгоритмы вставки и исключения элемента из БДП и вывода изображения бинарного дерева на экран.

ПРИЛОЖЕНИЕ А
КОД ПРОГРАММЫ

```
/*
    Вариант 9:
    Случайные БДП - вставка и исключение. Текущий контроль.
*/



#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>


#define TASK_COUNT_MAX 100 // максимальное количество генерируемых заданий
#define STR_LEN 101 // размер строки
#define TS_MIN 3 // минимальный допустимый размер исходного дерева
#define TS_MAX 26 // максимальный допустимый размер исходного дерева
#define OC_MIN 1 // минимальное допустимое количество операций над деревом
#define OC_MAX 10 // максимальное допустимое количество операций над деревом



// Направления для рисование стрелок при выводе изображения дерева.
enum Direction{
    LEFT = 1,
    RIGHT
};


// Обозначения для строки с заданием.
enum Commands{
    END = 0,
    ADD,
    REMOVE
};


FILE * fileTasks; // файл с заданиями
FILE * fileAnswers; // файл с ответами
```

```c
// Структура БДП.
typedef struct BST{
    char key;
    struct BST * left;
    struct BST * right;
} BST;


// Возврат левого поддерева.
BST* leftBST(BST * tree){
    if (tree != NULL)
        return tree->left;
    else
        return NULL;
}


// Возврат правого поддерева.
BST* rightBST(BST* tree){
    if (tree != NULL)
        return tree->right;
    else
        return NULL;
}


// Инициализация дерева.
BST * initBST(char key){
    BST * tree = (BST*)malloc(sizeof(BST));
    tree->key = key;
    tree->left = NULL;
    tree->right = NULL;
    return tree;
}


// Возврат глубины дерева.
int BST_DeepLevel(BST* tree){
    if (tree == NULL)
        return 0;
    int leftDeep = BST_DeepLevel(leftBST(tree));
    int rightDeep = BST_DeepLevel(rightBST(tree));
    return 1 + (leftDeep > rightDeep ? leftDeep : rightDeep);
}
```

```
// Вывод изображения дерева на экран.


// Создание сетки дерева для дальнейшей оптимизации рисунка.
void addUnitsInGrid(BST* tree, char** grid, int level, int deep_level, int horizontalIndex){
    grid[level-1][horizontalIndex] = tree->key;
    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 (leftBST(tree) != NULL){
        addUnitsInGrid(leftBST(tree), grid, level+1, deep_level, horizontalIndex - a);
    }
    if (rightBST(tree) != NULL){
        addUnitsInGrid(rightBST(tree), grid, level+1, deep_level, horizontalIndex + a);
    }
}
// Вывод изображения дерева.
void BST_draw(BST* tree, int print_in_file_bool){
    if (tree == NULL){
        printf("Вы ввели пустое дерево.\n");
    }
    if (leftBST(tree) == NULL && rightBST(tree) == NULL){
        printf("%c\n\n", tree->key);
        if (print_in_file_bool){
            fprintf(fileAnswers, "%c\n\n", tree->key);
        }
        return;
    }
    // В начале "нарисую" дерево в двумерном массиве.
    int deep_level = BST_DeepLevel(tree); // 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(tree, 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){ // Пустая область
            printf("   ");
            if (print_in_file_bool){
                fprintf(fileAnswers, "   ");
            }
        }
        if (grid[i][j] == LEFT){ // Стрелка влево
            printf("---");
            if (print_in_file_bool){
                fprintf(fileAnswers, "---");
            }
        }
        if (grid[i][j] == RIGHT){ // Стрелка вправо
            // Конец стрелки.
            if (grid[i+1][j] != 0 && grid[i+1][j] != LEFT && grid[i+1][j] != RIGHT){
```

```
                    printf("-  ");
                    if (print_in_file_bool){
                        fprintf(fileAnswers, "-  ");
                    }
                }
                // Середина или начало стрелки.
                else {
                    printf("---");
                    if (print_in_file_bool){
                        fprintf(fileAnswers, "---");
                    }
                }
            }
            if (grid[i][j] != 0 && grid[i][j] != LEFT && grid[i][j] != RIGHT){ // Узел
                if (j < gridWidth-1 && grid[i][j+1] == RIGHT){ // Если есть правое поддерево
                    printf("%c--", grid[i][j]);
                    if (print_in_file_bool){
                        fprintf(fileAnswers, "%c--", grid[i][j]);
                    }
                }
                else { // Если нет правого поддерева
                    printf("%c  ", grid[i][j]);
                    if (print_in_file_bool){
                        fprintf(fileAnswers, "%c  ", grid[i][j]);
                    }
                }
            }
        }
        putchar('\n');
        if (print_in_file_bool){
            fputc('\n', fileAnswers);
        }
        for (int j = 0; j < gridWidth; j++){
            if (grid[i][j] == 0 || (grid[i][j] != LEFT && grid[i][j] != RIGHT)){ // Пустая
область
                printf("   ");
                if (print_in_file_bool){
                    fprintf(fileAnswers, "   ");
                }
            }
            if (grid[i][j] == LEFT || grid[i][j] == RIGHT){ // Стрелки
                // Конец стрелки.
                if (grid[i+1][j] != 0 && grid[i+1][j] != LEFT && grid[i+1][j] != RIGHT){
                    printf("|  ");
                    if (print_in_file_bool){
                        fprintf(fileAnswers, "|  ");
                    }
                }
```

```
                // Середина или начало стрелки.
                else {
                    printf("   ");
                    if (print_in_file_bool){
                        fprintf(fileAnswers, "   ");
                    }
                }
            }
        }
        putchar('\n');
        if (print_in_file_bool){
            fputc('\n', fileAnswers);
        }
    }


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




// Печать пробелов для демонтсрации работы алгоритма.
void print_spaces(int counter){
    for(int i = 0; i < counter; i++){
        printf("    ");
    }
}


// Создание БДП.
BST * BST_create(char * bst_str){
    if (strlen(bst_str) == 0){
        return NULL;
    }

    printf("\nПостроение БДП: %s\n\n", bst_str);
    printf("Символ №1 - '%c':\n", bst_str[0]);
    printf("   '%c' - корень дерева.\n", bst_str[0]);
    BST * tree = initBST(bst_str[0]);


    for (int i = 1; i < strlen(bst_str); i++){
```

```
        printf("\nСимвол №%d - '%c':\n", i+1, bst_str[i]);
        // Вывод изображения дерева
        if (i > 1){
            printf("Текущее дерево:\n");
            BST_draw(tree, 0);
        }
        int counter = 1;
        BST * current_tree = tree;
        while(1){
            if (bst_str[i] < current_tree->key){ // Переход к левому поддереву
                print_spaces(counter);
                printf("'%c' < '%c'\n", bst_str[i], current_tree->key);
                if (current_tree->left == NULL){ // Оно пусто - добавляю элемент.
                    print_spaces(counter);
                    printf("Левое поддерево узла '%c' - пусто.\n", current_tree->key);
                    print_spaces(counter);
                    printf("'%c' - левое поддерево '%c'.\n", bst_str[i], current_tree->key);
                    current_tree->left = initBST(bst_str[i]);
                    break;
                } else { // Иду дальше вниз по дереву
                    print_spaces(counter);
                    printf("'%c' - левое поддерево узла %c.\n", leftBST(current_tree)->key,
current_tree->key);
                    counter ++;
                    current_tree = current_tree->left;
                }
            } else { // Переход к правому поддереву
                print_spaces(counter);
                printf("'%c' > '%c'\n", bst_str[i], current_tree->key);
                if (current_tree->right == NULL){ // Оно пусто - добавляю элемент.
                    print_spaces(counter);
                    printf("Правое поддерево узла '%c' - пусто.\n", current_tree->key);
                    print_spaces(counter);
                    printf("'%c' - правое поддерево '%c'.\n", bst_str[i], current_tree-
>key);
                    current_tree->right = initBST(bst_str[i]);
                    break;
                } else { // Иду дальше вниз по дереву
                    print_spaces(counter);
                    printf("'%c' - правое поддерево узла %c.\n", rightBST(current_tree)-
>key, current_tree->key);
                    counter ++;
                    current_tree = current_tree->right;
                }
            }
        }
    }
```

```
    return tree;
}


// Добавление элемента в БДП.
void add_element_in_BST(BST* tree, char key){
    printf("Добавление элемента '%c':\n", key);
    printf("Текущее дерево:\n");
    BST_draw(tree, 0); // Вывод изображения дерева.
    BST * current_tree = tree;
    int counter = 1;
    while(1){
        if (key < current_tree->key){ // Переход к левому поддереву
            print_spaces(counter);
            printf("'%c' < '%c'\n", key, current_tree->key);
            if (current_tree->left == NULL){ // Оно пусто - добавляю элемент.
                print_spaces(counter);
                printf("Левое поддерево узла '%c' - пусто.\n", current_tree->key);
                print_spaces(counter);
                printf("'%c' - левое поддерево '%c'.\n", key, current_tree->key);
                current_tree->left = initBST(key);
                break;
            } else { // Иду дальше вниз по дереву
                print_spaces(counter);
                printf("'%c' - левое поддерево узла %c.\n", leftBST(current_tree)->key,
current_tree->key);
                counter ++;
                current_tree = current_tree->left;
            }
        } else { // Переход к правому поддереву
            print_spaces(counter);
            printf("'%c' > '%c'\n", key, current_tree->key);
            if (current_tree->right == NULL){ // Оно пусто - добавляю элемент.
                print_spaces(counter);
                printf("Правое поддерево узла '%c' - пусто.\n", current_tree->key);
                print_spaces(counter);
                printf("'%c' - правое поддерево '%c'.\n", key, current_tree->key);
                current_tree->right = initBST(key);
                break;
            } else { // Иду дальше вниз по дереву
                print_spaces(counter);
                printf("'%c' - правое поддерево узла %c.\n", rightBST(current_tree)->key,
current_tree->key);
                counter ++;
                current_tree = current_tree->right;
            }
        }
```

```
    }
}


// Удаление элемента в БДП.
BST * remove_element_in_BST(BST* tree, char key){
    printf("Удаление элемента '%c':\n", key);
    printf("Текущее дерево:\n");
    BST_draw(tree, 0);
    BST* parentTree = tree;
    BST* childTree = tree;


    // Если нужно удалить корень.
    if (tree->key == key){
        if (leftBST(tree) == NULL && rightBST(tree) == NULL){ // Дерево состоит из одного
узла.
            free(tree);
            return NULL;
            // Если есть только одно поддерево.
        } else if ((leftBST(tree) != NULL && rightBST(tree) == NULL) || (leftBST(tree) ==
NULL && rightBST(tree) != NULL)){
            if (leftBST(tree) != NULL){
                printf("    Узел '%c' имеет только левое поддерево.\n", key);
                printf("    Узел '%c' занимает место узла '%c'.\n", leftBST(tree)->key,
key);
                BST* childTree = leftBST(tree);
                free(tree);
                return childTree;
            } else {
                printf("    Узел '%c' имеет только правое поддерево.\n", key);
                printf("    Узел '%c' занимает место узла '%c'.\n", rightBST(childTree)-
>key, key);
                BST* childTree = rightBST(tree);
                free(tree);
                return childTree;
            }
        } else { // Если есть два поддерева.
            printf("    Узел '%c' имеет два поддерева.\n", key);
            parentTree = tree;
            childTree = rightBST(tree);
            while (leftBST(childTree) != NULL){
                parentTree = childTree;
                childTree = leftBST(childTree);
            }
            printf("    Минимальный элемент в правом поддереве - '%c'.\n", childTree->key);
            printf("    Узел '%c' занимает место узла '%c'.\n", childTree->key, key);
            tree->key = childTree->key;
```

```
            if (childTree == leftBST(parentTree)){
                parentTree->left = rightBST(childTree);
            } else {
                parentTree->right = rightBST(childTree);
            }
            free(childTree);
            return tree;
        }
    }


    // В ином случае поиск нужного элемента в дереве.
    while (1){
        if (childTree->key == key){
            break;
        } else if (childTree->key > key){
            parentTree = childTree;
            childTree = leftBST(childTree);
        } else {
            parentTree = childTree;
            childTree = rightBST(childTree);
        }
    }
    if (leftBST(childTree) == NULL && rightBST(childTree) == NULL){ // Если нету
поддеревьев.
        printf("    Узел '%c' не имеет поддеревьев.\n", key);
        printf("    Удаление узла '%c'.\n", key);
        if (childTree == leftBST(parentTree)){
            parentTree->left = NULL;
        } else {
            parentTree->right = NULL;
        }
        free(childTree);
        // Если есть одно поддерево.
    } else if ((leftBST(childTree) != NULL && rightBST(childTree) == NULL) ||
(leftBST(childTree) == NULL && rightBST(childTree) != NULL)){
        if (leftBST(childTree) != NULL){
            printf("    Узел '%c' имеет только левое поддерево.\n", key);
            printf("    Узел '%c' занимает место узла '%c'.\n", leftBST(childTree)->key,
key);
        } else {
            printf("    Узел '%c' имеет только правое поддерево.\n", key);
            printf("    Узел '%c' занимает место узла '%c'.\n", rightBST(childTree)->key,
key);
        }
        if (childTree == leftBST(parentTree)){
            if (leftBST(childTree) != NULL){
                parentTree->left = leftBST(childTree);
```

```
            } else {
                parentTree->left = rightBST(childTree);
            }
        } else {
            if (leftBST(childTree) != NULL){
                parentTree->right = leftBST(childTree);
            } else {
                parentTree->right = rightBST(childTree);
            }
        }
        free(childTree);
    } else { // Если есть два поддерева.
        printf("    Узел '%c' имеет два поддерева.\n", childTree->key);
        BST* deletedRoot = childTree;
        parentTree = deletedRoot;
        childTree = rightBST(deletedRoot);
        while (leftBST(childTree) != NULL){
            parentTree = childTree;
            childTree = leftBST(childTree);
        }
        printf("    Минимальный элемент в правом поддереве - '%c'.\n", childTree->key);
        printf("    Узел '%c' занимает место узла '%c'.\n", childTree->key, deletedRoot-
>key);
        deletedRoot->key = childTree->key;
        if (childTree == leftBST(parentTree)){
            parentTree->left = rightBST(childTree);
        } else {
            parentTree->right = rightBST(childTree);
        }
        free(childTree);
    }
    return tree;
}


// Возврат случайного целого числа в диапазоне [min, max].
int getRandInt(int min, int max){
    return rand() % (max - min + 1) + min;
}




// Очистка памяти.
void freeMemory(BST* tree){
    if(tree != NULL){
        freeMemory(leftBST(tree));
        freeMemory(rightBST(tree));
```

```
        free(tree);
    }
}
```

```
int main(){
    int task_count; // количество заданий
    int tree_size_min; // минимальный размер исходного дерева
    int tree_size_max; // максимальный размер исходного дерева
    int operation_count_min; // минимальное количество операций над деревом
    int operation_count_max; // максимальное количество операций над деревом
    char str[STR_LEN]; // строка для разных нужд
    int counter; // счетчик для разных нужд
    BST * tree = NULL; // бинарное дерево поиска
    char ** tasks; // массив с заданиями

    srand(time(NULL));


    fileTasks = fopen("tasks.txt", "w"); // файл с заданиями
    fileAnswers = fopen("answers.txt", "w"); // файл с ответами


    // Приветствие.
    printf("\nПрограмма генерирует задания с ответами на тему ");
    printf("\"Случайные БДП - вставка и исключение\".\n");


    // Ввод количества генерируемых заданий и настройка сложности.
    // Ввод Количества заданий.
    printf("\nВведите количество заданий, которое необходимо сгенерировать (не больше %d):
", TASK_COUNT_MAX);
    while(1){
        fgets(str, STR_LEN, stdin);
        counter = 0;
        task_count = 0;
        for (int i = 0; i < strlen(str)-1; i++){
            if (!isdigit(str[i])){
                printf("Ошибка. Введите целое неотрицательное число: ");
                counter++;
```

```
                break;
            }
        }
        if (counter == 0){
            for (int i = 0; i < strlen(str)-1; i++){
                task_count *= 10;
                task_count += str[i] - '0';
            }
            if (task_count > 100){
                printf("Ошибка. Введите число, не превосходящее %d: ", TASK_COUNT_MAX);
            } else{
                break;
            }
        }
    }
    if (task_count == 0){
        printf("\nПрограмма завершила работу.\n");
        return 0;
    }
    // Ввод минимального размера дерева.
    printf("\nВведите минимальное количество элементов в начальном дереве (не меньше %d, но не больше %d): ", TS_MIN, TS_MAX);
    while(1){
        fgets(str, STR_LEN, stdin);
        counter = 0;
        tree_size_min = 0;
        for (int i = 0; i < strlen(str)-1; i++){
            if (!isdigit(str[i])){
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ", TS_MIN, TS_MAX);
                counter++;
                break;
            }
        }
        if (counter == 0){
            for (int i = 0; i < strlen(str)-1; i++){
                tree_size_min *= 10;
                tree_size_min += str[i] - '0';
            }
            if (tree_size_min < TS_MIN || tree_size_min > TS_MAX){
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ", TS_MIN, TS_MAX);
            } else{
                break;
            }
        }
    }
    // Ввод максимального размера дерева.
```

```
    printf("\nВведите максимальное количество элементов в начальном дереве (не меньше %d, но не больше %d): ", tree_size_min, TS_MAX);
    while(1){
        fgets(str, STR_LEN, stdin);
        counter = 0;
        tree_size_max = 0;
        for (int i = 0; i < strlen(str)-1; i++){
            if (!isdigit(str[i])){
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ", tree_size_min, TS_MAX);
                counter++;
                break;
            }
        }
        if (counter == 0){
            for (int i = 0; i < strlen(str)-1; i++){
                tree_size_max *= 10;
                tree_size_max += str[i] - '0';
            }
            if (tree_size_max < tree_size_min || tree_size_max > TS_MAX){
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ", tree_size_min, TS_MAX);
            } else{
                break;
            }
        }
    }
    // Ввод минимального количества операции над деревом.
    printf("\nВведите минимальное количество операций над деревом - вставка и исключение (не меньше %d, но не больше %d): ", OC_MIN, OC_MAX);
    while(1){
        fgets(str, STR_LEN, stdin);
        counter = 0;
        operation_count_min = 0;
        for (int i = 0; i < strlen(str)-1; i++){
            if (!isdigit(str[i])){
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ", OC_MIN, OC_MAX);
                counter++;
                break;
            }
        }
        if (counter == 0){
            for (int i = 0; i < strlen(str)-1; i++){
                operation_count_min *= 10;
                operation_count_min += str[i] - '0';
            }
            if (operation_count_min < OC_MIN || operation_count_min > OC_MAX){
```

```
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ", OC_MIN,
OC_MAX);
            } else{
                break;
            }
        }
    }
    // Ввод максимального количества операцйи над деревом.
    printf("\nВведите максимальное количество операций над деревом - вставка и исключение
(не меньше %d, но не больше %d): ", operation_count_min, OC_MAX);
    while(1){
        fgets(str, STR_LEN, stdin);
        counter = 0;
        operation_count_max = 0;
        for (int i = 0; i < strlen(str)-1; i++){
            if (!isdigit(str[i])){
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ",
operation_count_min, OC_MAX);
                counter++;
                break;
            }
        }
        if (counter == 0){
            for (int i = 0; i < strlen(str)-1; i++){
                operation_count_max *= 10;
                operation_count_max += str[i] - '0';
            }
            if (operation_count_max < operation_count_min || operation_count_max > OC_MAX){
                printf("Ошибка. Введите число не меньше %d, но не больше %d: ",
operation_count_min, OC_MAX);
            } else{
                break;
            }
        }
    }


    // Генерация заданий.
    printf("\nГенерация заданий:\n\n");
    // Выделение памяти.
    tasks = (char**)malloc(task_count * sizeof(char*));
    char ** minimalized_tasks = (char**)malloc(task_count * sizeof(char*));
    for (int i = 0; i < task_count; i++){
        tasks[i] = (char*)malloc((tree_size_max + 1 + 2 * operation_count_max) *
sizeof(char));
        minimalized_tasks[i] = (char*)malloc((tree_size_max + 1 + 2 * operation_count_max) *
sizeof(char));
```

```
        int treeSize;
        int operation_count;
        // Генерация задания.
        while (1){
            treeSize = getRandInt(tree_size_min, tree_size_max);
            int current_treeSize = treeSize;
            operation_count = getRandInt(operation_count_min, operation_count_max);
            char * current_list_of_element =
(char*)malloc((treeSize+operation_count)*sizeof(char));
            // Генерация дерева.
            for (int j = 0; j < treeSize; j++){
                tasks[i][j] = getRandInt('a', 'z');
                current_list_of_element[j] = tasks[i][j];
                for (int k = 0; k < j; k++){
                    if (tasks[i][j] == tasks[i][k]){
                        j--;
                        break;
                    }
                }
            }
            // Генерация операций над деревом.
            for (int j = 0; j < operation_count; j++){
                if (current_treeSize < 3){
                    tasks[i][treeSize+j*2] = ADD;
                } else if (current_treeSize == 26){
                    tasks[i][treeSize+j*2] = REMOVE;
                } else {
                    tasks[i][treeSize+j*2] = getRandInt(ADD, REMOVE);
                }
                if (tasks[i][treeSize+j*2] == ADD){
                    // Вставка элемента.
                    char added_element;
                    while (1){
                        added_element = getRandInt('a', 'z');
                        int checker = 0;
                        for (int m = 0; m < current_treeSize; m++){
                            if (added_element == current_list_of_element[m]){
                                checker++;
                                break;
                            }
                        }
                        if (checker == 0){
                            break;
                        }
                    }
                    tasks[i][treeSize+j*2+1] = added_element;
                    current_list_of_element[current_treeSize] = added_element;
                    current_treeSize++;
```

```
                } else {
                    // исключение элемента.
                    int num_of_remove_element = getRandInt(1, current_treeSize);
                    tasks[i][treeSize+j*2+1] =
current_list_of_element[num_of_remove_element-1];
                    for (int m = num_of_remove_element-1; m < current_treeSize-1; m++){
                        current_list_of_element[m] = current_list_of_element[m+1];
                    }
                    current_treeSize--;
                }
            }
            tasks[i][treeSize+2*operation_count] = END;

            // Проверка на отсутствие аналогичных заданий.
            for (int j = 0; j <= treeSize+2*operation_count; j++){
                minimalized_tasks[i][j] = tasks[i][j];
            }
            char current_value = REMOVE + 1;
            for (char c = 'a'; c <= 'z'; c++){
                int checker = 0;
                for (int j = 0; j <= treeSize+2*operation_count; j++){
                    if (minimalized_tasks[i][j] == c){
                        minimalized_tasks[i][j] = current_value;
                        checker++;
                    }
                }
                if (checker){
                    current_value++;
                }
            }
            free(current_list_of_element);
            counter = 0;
            for (int j = 0; j < i; j++){
                if (strcmp(minimalized_tasks[j], minimalized_tasks[i]) == 0){
                    counter++;
                    break;
                }
            }
            // Если аналогичное задание уже существует, то сгенерировать новое.
            if (counter == 0){
                break;
            }
        }

        // Текст задания.
        putchar('\n');
        printf("Задание %d.\n", i+1);
        fprintf(fileTasks, "Задание %d.\n", i+1);
```

```
fprintf(fileAnswers, "Задание %d.\n", i+1);
printf("Построить БДП: ");
fprintf(fileTasks, "Построить БДП: ");
fprintf(fileAnswers, "Построить БДП: ");
for (int j = 0; j < treeSize; j++){
    putchar(tasks[i][j]);
    fputc(tasks[i][j], fileTasks);
    fputc(tasks[i][j], fileAnswers);
}
printf(".\n");
fprintf(fileTasks, ".\n");
fprintf(fileAnswers, ".\n");
for (int j = treeSize; j <= treeSize+2*operation_count; j++){
    if (tasks[i][j] == END){
        putchar('\n');
        fputc('\n', fileTasks);
        fputc('\n', fileAnswers);
    } else if (tasks[i][j] == ADD){
        printf("Добавить элемент ");
        fprintf(fileTasks, "Добавить элемент ");
        fprintf(fileAnswers, "Добавить элемент ");
    } else if (tasks[i][j] == REMOVE){
        printf("Исключить элемент ");
        fprintf(fileTasks, "Исключить элемент ");
        fprintf(fileAnswers, "Исключить элемент ");
    } else {
        printf("\'%c\'.\n", tasks[i][j]);
        fprintf(fileTasks, "\'%c\'.\n", tasks[i][j]);
        fprintf(fileAnswers, "\'%c\'.\n", tasks[i][j]);
    }
}

// Решение задания.
printf("Решение:\n");
char c = tasks[i][treeSize];
tasks[i][treeSize] = '\0';
tree = BST_create(tasks[i]);
tasks[i][treeSize] = c;
putchar('\n');
for (int j = 0; j < operation_count; j++){
    if (tasks[i][treeSize+2*j] == ADD){
        add_element_in_BST(tree, tasks[i][treeSize+2*j+1]);
        putchar('\n');
    } else {
        tree = remove_element_in_BST(tree, tasks[i][treeSize+2*j+1]);
        putchar('\n');
    }
}
```

```
            printf("Результат:\n");
            fprintf(fileAnswers, "Ответ:\n");
            BST_draw(tree, 1);
            fputc('\n', fileAnswers);
            freeMemory(tree);
        }

        printf("\nТекст заданий записан в файл tasks.txt, ответы находятся в файле
answers.txt.\n");

        // Освобождение памяти и закрытие файлов.
        for (int i = 0; i < task_count; i++){
            free(tasks[i]);
            free(minimalized_tasks[i]);
        }
        free(tasks);
        free(minimalized_tasks);


        fclose(fileTasks);
        fclose(fileAnswers);

        return 0;
}
```