МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №5 (вариант 9)

по дисциплине «Алгоритмы и структуры данных»

Тема: «Бинарное дерево поиска»

Студент гр. 7381 _____ Адамов Я.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2018

Цель работы.

Ознакомиться с такой структурой данных, как бинарное дерево поиска, и научиться применять БДП на практике.

Основные теоретические положения.

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.

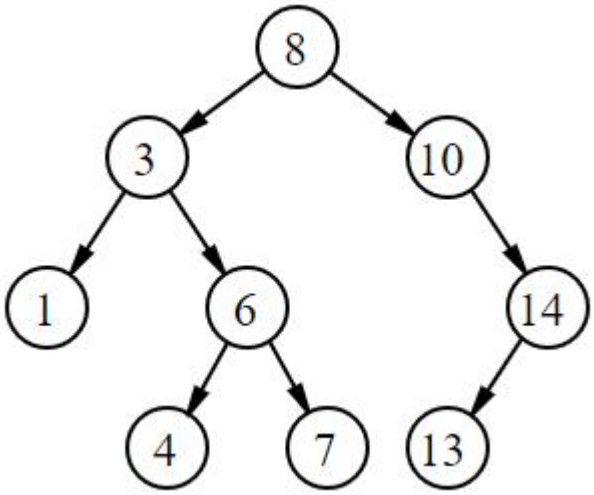


Рисунок 1 – Пример БДП

Задание.

Вариант 9.

1) По заданному файлу F (типа file of Elem), все элементы которого различны, построить БДП определённого типа.
2) Записать в файл элементы построенного БДП в порядке их возрастания; вывести построенное БДП на экран.

Пояснение задания.

Вариант 9.

На вход программе подается строка, содержащая имя файла, в который записана последовательность символов. По этой строке программа должна построить случайное БДП. Программа должна продемонстрировать алгоритм работы. Построенное дерево необходимо вывести на экран, а узлы этого дерева записать в отдельный файл в порядке возрастания.

Ход работы.

Программа будет писаться на языке C.

Исходный файл: main.c

Программа считывает строку *fileName* из файла или с клавиатуры. Строка *fileName* содержит имя файла, который необходимо открыть для считывания содержащейся в нём последовательности символов. Если такого файла не существует или произошла ошибка при его открытии, программа выводит соответствующее сообщение об ошибке. Также программа выведет сообщение об ошибке, если последовательность символов будет не валидной: строка состоит не из цифр и латинских букв, имеются повторяющиеся символы. Если же введённые данные корректны, то вызывается функция *BST_create()*, которая строит по этой строке БДП и выводит алгоритм построения на экран.

После удачного построения бинарного дерева вызывается функции *treeDrawing*() и *printResult* (), которые выводят на экран изображение дерева и перечисление узлов в порядке возрастания.

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 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 БСТ по переданной строке с символами.

Параметры:

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

Описание алгоритма:

Функция поочередно обрабатывает символы строки *bst_str*. 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.

7) *void BST_draw(BST* tree)*

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

Параметры:

- **tree**: 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 рекурсивная 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.

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 printResult(BST * tree, FILE * fileWrite)*

Функция выводит на экран и в файл перечисление элементов дерева в порядке возрастания.

Параметры:

- **tree**: БДП, элементы которого необходимо перечислить.
- **fileWrite**: указатель на файл, куда будет записываться результат.

Описание алгоритма: если левое поддерево переданного БДП не является пустым, то функция вызывается рекурсивно для левого поддерева, затем выводится значение ключа этого дерева, после чего аналогичное действие для правого поддерева..

10) *void freeMemory(BST* tree)*

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

Параметры:

- **tree**: 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 поиска. Получены навыки по работе с БДП.

Приложение А. Тестирование.

Демонстрация работы программы:

Test 1:

```
ceafbdg
```

output:

```
Программа считывает строку из файла и строит из полученных данных БДП,
выводит его изображение на экран и записывает элементы БДП в порядке
возрастания в файл.

Введите имя файла (не больше 80 символов): test1.txt
Входная строка: ceafbdg
Алгоритм построения БДП:

Символ №1 - 'c':
   'c' - корень дерева.
Символ №2 - 'e':
    'e' > 'c'
    Правое поддерево узла 'c' - пусто.
    'e' - правое поддерево 'c'.
Символ №3 - 'a':
    'a' < 'c'
    Левое поддерево узла 'c' - пусто.
    'a' - левое поддерево 'c'.
Символ №4 - 'f':
    'f' > 'c'
    'e' - правое поддерево узла c.
        'f' > 'e'
        Правое поддерево узла 'e' - пусто.
        'f' - правое поддерево 'e'.
Символ №5 - 'b':
    'b' < 'c'
    'a' - левое поддерево узла c.
        'b' > 'a'
        Правое поддерево узла 'a' - пусто.
```

```
        'b' - правое поддерево 'a'.
Символ №6 - 'd':
    'd' > 'c'
    'e' - правое поддерево узла c.
        'd' < 'e'
        Левое поддерево узла 'e' - пусто.
        'd' - левое поддерево 'e'.
Символ №7 - 'g':
    'g' > 'c'
    'e' - правое поддерево узла c.
        'g' > 'e'
        'e' - правое поддерево узла e.
            'g' > 'f'
            Правое поддерево узла 'f' - пусто.
            'g' - правое поддерево 'f'.

Изображение дерева:

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


Элементы БДП в порядке возрастания: abcdefg
(также записаны в файл result.txt).
```

Пояснение к выводу программы:

После того, как пользователь ввёл строку *bst_str* (последовательность символов), программа строит БДП. Процесс обработки каждого символа выводится на экран: если текущий символ меньше ключа корня, то переходим вниз к левому поддереву,

в ином случае – к правому. Если это поддерево пусто, то создаётся новый узел, в ином случае аналогичным способом программа идёт ниже по дереву.

После создания БДП на экран выводится его изображение и перечисление элементов этого дерева в порядке возрастания.

Тестирование:

Test 2:

```
abcdef
```

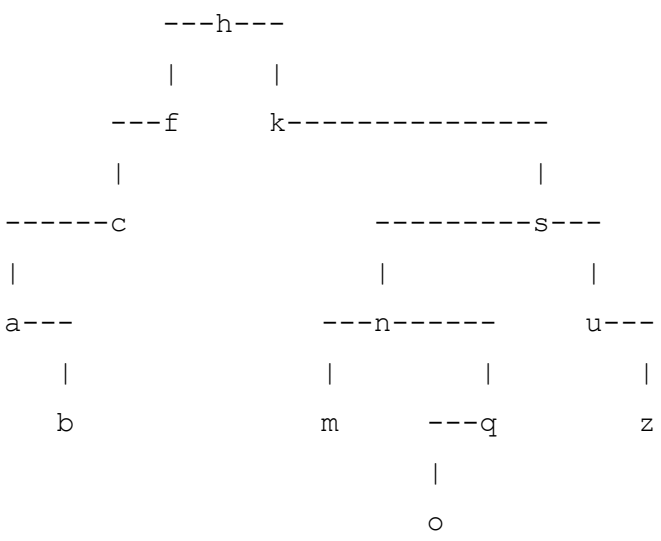
output:

```
Изображение дерева:

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


Элементы БДП в порядке возрастания: abcdef
```

Test 3:

```
hfksucnzabmqo
```

output:




```
Элементы БДП в порядке возрастания: abcfhkmnoqsuz
```

Test 4:

```
abdfcadqqjb
```

output:

```
Введены некорректные данные:
'a', 'b', 'd', 'q' встречаются больше одного раза.
```

Test 5:

```
a-d.,gb vx
```

output:

```
Введены некорректные данные:
Символ №2 - '-'.
Символ №4 - '.'.
Символ №5 - ','.
Символ №8 - ' '.

Строки должны состоять только из цифр и латинских букв.
```

Приложение Б. Файл main.c.

```
/*
    Вариант 9:
    1) По заданному файлу F (типа file of Elem), все элементы
    которого различны, построить БДП определённого типа;
    2) Записать в файл элементы построенного БДП в порядке их возрастания;
    вывести построенное БДП на экран.
*/



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


#define STR_LENGTH 81


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



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;
}



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



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);
}



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

```
BST * BST_create(char * bst_str){
    if (strlen(bst_str) == 0){
        return NULL;
    }


    printf("Алгоритм построения БДП:\n\n");
    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("Символ №%d - '%c':\n", i+1, bst_str[i]);
        int counter = 1;
        BST * current_tree = tree;
        while(1){
            if (bst_str[i] < current_tree->key){
                print_spaces(counter);
                printf("'%c' < '%c'\n", bst_str[i], current_tree->key);
                if (current_tree->left == NULL){
                    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 {
                    print_spaces(counter);
                    printf("'%c' - левое поддерево узла %c.\n", leftBST(tree)->key, current_tree-
>key);
                    counter ++;
                    current_tree = current_tree->left;
                }
            } else {
                print_spaces(counter);
                printf("'%c' > '%c'\n", bst_str[i], current_tree->key);
                if (current_tree->right == NULL){
                    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 {
                    print_spaces(counter);
                    printf("'%c' - правое поддерево узла %c.\n", rightBST(tree)->key,
current_tree->key);
                    counter ++;
                    current_tree = current_tree->right;
                }
            }
        }
    }


    return tree;
}




// 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){
    if (tree == NULL){
        printf("Вы ввели пустое дерево.\n");
    }
```

```
if (leftBST(tree) == NULL && rightBST(tree) == NULL){
    printf("%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)
                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);
}


// Вывод результата.
void printResult(BST * tree, FILE * fileWrite){
    if(leftBST(tree) != NULL){
        printResult(leftBST(tree), fileWrite);
    }
    if (fileWrite != NULL){
```

```
        printf("%c", tree->key);
        fprintf(fileWrite, "%c", tree->key);
    }
    if (rightBST(tree) != NULL){
        printResult(rightBST(tree), fileWrite);
    }
}




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




int main(){
    char fileName[STR_LENGTH];
    char bst_str[STR_LENGTH]; // входная страка из файла
    FILE * fileRead; // файл с данными
    FILE * fileWrite; // файл, куда будет записан результат
    BST * tree = NULL; // бинарное дерево поиска
    int counter = 0; // счетчик для разных нужд

    // Приветствие.
    printf("\nПрограмма считывает строку из файла и строит из полученных данных БДП,\n");
    printf("выводит его изображение на экран и записывает элементы БДП в порядке возрастания в
файл.\n");


    // Считывание входных данных.
    printf("\nВведите имя файла (не больше %d символов): ", STR_LENGTH-1);
    fgets(fileName, STR_LENGTH, stdin);
    if (fileName[strlen(fileName)-1] == '\n'){
        fileName[strlen(fileName)-1] = '\0';
    }
    fileRead = fopen(fileName, "r");
    if (fileRead == NULL){
```

```
    printf("\nОшибка: файл не найден.\n");
    return 1;
}
fgets(bst_str, STR_LENGTH, fileRead);
if (bst_str[strlen(bst_str)-1] == '\n'){
    bst_str[strlen(bst_str)-1] = '\0';
}
fclose(fileRead);


// Проверки на ошибки.
if (strlen(bst_str) == 0){
    printf("Вы ввели пустую строку.\n");
    return 0;
}
printf("\nВходная строка: %s\n", bst_str);
// Проверка на использование недопустимых символов.
for (int i = 0; i < strlen(bst_str); i++){
    if (!isdigit(bst_str[i]) && !isalpha(bst_str[i])){
        if (!counter)
            printf("\nВведены некорректные данные:\n");
        printf("Символ №%d - '%c'.\n", i+1, bst_str[i]);
        counter++;
    }
}
if (counter){
    printf("\nСтроки должны состоять только из цифр и латинских букв.\n");
    return 1;
}
// Проверка на отсутствие повторяющихся символов.
for (int i = 0; i < strlen(bst_str); i++){
    for (int j = i; j >= 0; j--){
        if (bst_str[i] == bst_str[j] && i != j){
            break;
        }
        if (j == 0){
            for (int k = i; k < strlen(bst_str); k++){
                if (bst_str[i] == bst_str[k] && i != k){
                    if (!counter){
                        printf("\nВведены некорректные данные:\n");
                        printf("'%c'", bst_str[i]);
                    } else {
                        printf(", '%c'", bst_str[i]);
                    }
                    counter++;
```

```
                    break;
                }
            }
        }
    }
    if (counter){
        if (counter > 1){
            printf(" встречаются больше одного раза.\n");
        } else {
            printf(" встречается больше одного раза.\n");
        }
        return 1;
    }


    tree = BST_create(bst_str);


    printf("\nИзображение дерева:\n\n");
    BST_draw(tree);


    printf("\nЭлементы БДП в порядке возрастания: ");
    fileWrite = fopen("result.txt", "w");
    printResult(tree, fileWrite);
    fclose(fileWrite);
    printf("\n(также записаны в файл result.txt).\n");


    freeMemory(tree);


    return 0;
}
```