Отчет по лабораторной работе № 23 по курсу по курсу “Практикум на ЭВМ”

Студент группы **М8О-107Б-20** Чекменев Вячеслав Алексеевич, № по списку 27

Контакты e-mail: chekmenev031@gmail.com, telegram: @suraba03

Работа выполнена: «21» мая 2021 г.

Преподаватель: каф. 806 Найденов Иван Евгеньевич

Отчет сдан « » _____20 ___ г., итоговая оценка _____

Подпись преподавателя _____

1. **Тема:** Динамические структуры данных. Обработка деревьев.

2. **Цель работы:** Составить программу на языке Си для построения и обработки дерева общего вида или упорядоченного двоичного дерева.

3. **Задание:** (вариант 15) выяснить, все ли листья находятся на одном уровне

4. **Оборудование** (студента):

   Процессор *Intel Core i5-8265U* с ОП 7851 Мб, НМД 256 Гб. Монитор *1920x1080*

5. **Программное обеспечение (студента):**

   Операционная система семейства UNIX: linux, наименование: manjaro, версия: 20.1 Mikah
   интерпретатор команд: bash, версия: 5.0.18.
   текстовый редактор: atom, версия: 5.2
   Утилиты операционной системы --
   Прикладные системы и программы --
   Местонахождение и имена файлов программ и данных –

6. Идея, метод, алгоритм решения задачи (в формах: словесной, псевдокода, графической [блок-схема, диаграмма, рисунок, таблица] или формальные спецификации с пред- и постусловиями)

1) Написать структуру дерева, состоящую из целого числа(значения, хранимого в узле), указателей на правого и левого сына.

2) Напишем основные функции для обработки BST.
- Для создания дерева создадим его корень с некоторым значением целого типа, выделив память под структуру и заполнив поля указателей нулевыми значениями, а поле целого типа-некоторым значением.
- напишем функцию поиска мин значения в поддереве
- напишем функцию поиска узла по значению в нем
- Для добавления вершины в дерево по значению напишем функцию, которая по значению узла дерева возвращает указатель на эту вершину. Напишем функцию добавления вершины в дерево по правилам BST
- Для удаления вершины дерева рассмотрим три случая.
- Для графического представления будем выводить значения, хранимые в узлах слева направо.

3). Напишем функцию, проверяющую, все ли листья находятся на одном уровне, будем рекурсивно ходит в глубину, запоминая уровни листьев, на вовод будем подавать 1 или 0.

7. Сценарий выполнения работы [план работы, первоначальный текст программы в черновике (можно на отдельном листе) и тесты либо соображения по тестированию].

тесты:

Входные данные	**Выходные данные**	**Описание тестируемого случая**
c 20 a 5 a 30 a 1 a 15 a 25 a 40 a 9 a 7 a 12 a 45 a 42 p l	а вот и дерево: 1 5 7 9 12 15 20 25 30 40 42 45 Не все листья на одном уровне	Проверка на разные удаления, разные добавления + вывод дерева и логическая функция
c 20 a 5 a 30 a 1 a 15 a 25 a 40 a 9 a 7 a 12 a 45 a 42 d 40 d 15 p l	а вот и дерево: 1 5 7 9 12 20 25 30 42 45 Не все листья на одном уровне	Другая проверка на логическую функцию
c 20 a 5 a 30 a 1 a 15 a 25 a 40 a 9 a 7 a 12 a 45 a 42 d 40 d 15 d 42 d 7 d 12 p l	а вот и дерево: 5 9 20 25 30 45 Все листья на одном уровне	Случай, когда все листья на одном уровне

8. Распечатка протокола (подклеить листинг окончательного варианта программы с тестовыми примерами, подписанный преподавателем).

Tree.h

```
#ifndef _TREE_H_
#define _TREE_H_

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

typedef struct tree {
   int value;
   struct tree *right;
   struct tree *left;
} tree;

tree *tree_node_create(int crt_value);
tree *search(tree *root, int srch_value);
void print_tree(tree *root, int level);
tree *tree_find_minimum(tree *root);
tree *tree_add_node(tree *root, int add_value);
tree *tree_delete_node(tree *root, int del_value);
void inorder(tree *root);
int checkUtil(tree *root, int level, int *leafLevel);
int check(tree *root);
void delete_tree(tree *curr);

#endif
```

Tree_func.c

```
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

tree *tree_node_create(int crt_value)
{
   tree *result = (tree *)malloc(sizeof(tree));

   if (result != NULL) {
      result->right = NULL;
      result->left = NULL;
      result->value = crt_value;
      return result;
   } else {
      printf("Error: not enough memory to create a node\n");
      return NULL;
   }
}

tree *search(tree *root, int srch_value) // srch_value is the searching value
{
   if (root == NULL || root->value == srch_value) { //  if root->value == srch_value then the element found or we reached the null node (reached the leaf's child)
      return root;
   }
   else if (srch_value > root->value) { // if srch_value greater then root->value, so we will search the right subtree
      return search(root->right, srch_value);
   }
   else { // if srch_value smaller then root->value, so we will search the left subtree
      return search(root->left, srch_value);
   }
}

void print_tree(tree *root, int level)
{
```

```
    if (root != NULL) {
        print_tree(root->left, level + 1);

        for (int i = 0; i < level; i++) {
            printf("\t");
        }
        printf("%d\n\n", root->value);
        print_tree(root->right, level + 1);
    }
}


// tree_find_minimum function
// call the function, while the 1st condition is false (root (root->left from last call of func) == NULL).
// When 1st condition will become true,
// program will return current root value.

tree *tree_find_minimum(tree *root)
{
    if (root == NULL) {
        return NULL;
    }
    else if (root->left != NULL) {
        return tree_find_minimum(root->left);
    }
    return root;
}

// tree_add_node function
// Assign function value to right (left) child's node, while the 1st condition is false.
// When 1st condition will become true,
// I'll create new node in place of the last tree_add_node function's output.

tree *tree_add_node(tree *root, int add_value)
{
    if (root == NULL) {
        return tree_node_create(add_value);
    }
    else if (add_value > root->value) {
        root->right = tree_add_node(root->right, add_value);
    }
    else {
        root->left = tree_add_node(root->left, add_value);
    }
    return root;
}

tree *tree_delete_node(tree *root, int del_value)
{
    if (root == NULL) {
        printf("Error: a node with value = %d does not exist in the tree\n", del_value);
        return NULL;
    }
    if (del_value > root->value) {
        root->right = tree_delete_node(root->right, del_value);
    }
    else if (del_value < root->value) {
        root->left = tree_delete_node(root->left, del_value);
    }
    else {

        // no children
        if (root->right == NULL && root->left == NULL) {
            free(root);
            return NULL;
        }
```

```
        // one child
        else if (root->right == NULL || root->left == NULL) {
            tree *temp;

            if (root->left == NULL) {
                temp = root->right;
            } else {
                temp = root->left;
            }
            free(root);
            return temp;
        }

        // two children
        else {
            tree *temp = tree_find_minimum(root->right);
            root->value = temp->value;
            root->right = tree_delete_node(root->right, temp->value);
        }
    }
    return root;
}

void inorder(tree *root)
{
    if (root != NULL) { // checking if the root is not null
        inorder(root->left); // visiting left child
        printf(" %d ", root->value); // printing data at root
        inorder(root->right); // visiting right child
    }
}

int checkUtil(tree *root, int level, int *leafLevel)
{
    // Базовый вариант
    if (root == NULL) {
        return 1;
    }
    // Если встречается листовой узел
    if (root->left == NULL && root->right == NULL)
    {
        // Когда листовой узел найден впервые
        if (*leafLevel == 0)
        {
            *leafLevel = level; // Установить уровень первого найденного листа
            return 1;
        }

        // Если это не первый листовой узел, сравниваем его уровень с
        // уровень первого листа
        return (level == *leafLevel);
    }

    // Если этот узел не листовой, рекурсивно проверяем левое и правое поддеревья
    return checkUtil(root->left, level + 1, leafLevel) && checkUtil(root->right, level + 1, leafLevel);

}

int check(tree *root)
{
   int level = 0, leafLevel = 0;
   return checkUtil(root, level, &leafLevel);
}

void delete_tree(tree *curr)
```

```
{
    if (curr) {
        delete_tree(curr->left);
        delete_tree(curr->right);
        tree_delete_node(curr, curr->value);
    }
}
```

client.c

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


int main(int argc, char *argv[])
{
    printf("Напишите '?' для получения помощи в использовании программы:\n");

    char c;
    int value, node_v;
    int cnt_root = 0;
    tree *root;

    while ((c = getchar()) != EOF) {
        if (c == '?') {
            printf("Набор команд:\n");
            printf("c - создать корень (введите значение корня).\n");
            printf("a - добавить узел в дерево (введите значение нового узла).\n");
            printf("d - удалить узел из дерева (введите значение удаляемого узла).\n");
            printf("p - вывести дерево.\n");
            printf("l - все ли листья на одном уровне?\n"); // лучше f
            printf("b - удалить дерево.\n"); // лучше r
        } else if (c == 'c' && cnt_root == 0) {
            cnt_root++;
            printf("введите значение корня: ");
            scanf("%d", &value);
            root = tree_node_create(value);
        } else if (c == 'c' && cnt_root != 0) {
            printf("Корень уже существует!\n");
        } else if (c == 'a' && cnt_root != 0) {
            printf("введите значение узла: ");
            scanf("%d", &node_v);
            tree_add_node(root, node_v);
        } else if (c == 'a' && cnt_root == 0) {
            printf("Дерево не существует.\n");
        } else if (c == 'd' && cnt_root != 0) {
            printf("введите значение узла: ");
            scanf("%d", &node_v);
            root = tree_delete_node(root, node_v);
        } else if (c == 'd' && cnt_root == 0) {
            printf("Дерево не существует.\n");
        } else if (c == 'p' && cnt_root != 0) {
            printf("а вот и дерево: \n");
            print_tree(root, 0);
        } else if (c == 'p' && cnt_root == 0) {
            printf("Дерево не существует.\n");
        } else if (c == 'l' && cnt_root != 0) {
            if (check(root)) {
                printf("Все листья на одном уровне\n");
            } else {
                printf("Не все листья на одном уровне\n");
            }
        } else if (c == 'l' && cnt_root == 0) {
            printf("Дерево не существует.\n");
        } else if (c == 'b' && cnt_root != 0) {
            delete_tree(root);
            cnt_root--;
```

```
        } else if (c == 'b' && cnt_root == 0) {
            printf("Дерево не существует.\n");
        }
    }
    return 0;
}
```

9. Дневник отладки должен содержать дату и время сеансов отладки и основные события (ошибки в сценарии и программе, нестандартные ситуации) и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании других ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы.

№	Лаб. или дом.	Дата	Время	Событие	Действие по исправлению	Примечание
23	дома	21.05	20 00	закончил	никаких	

10. Замечания автора по существу работы

11. Выводы

В лабораторной работе я познакомился с BST. Работа была сложной при работе с динамической памятью, при построении дерева и реализации различных алгоритмов деревьев и при отладке программы, ведь с памятью нужно работать аккуратно. Деревья имеют огромное значения для хранения, представления различных данных + для поиска минимальльных, максимальных значений. Я уверен, что полученные знания мне пригодятся в будущем(например для будущих ЛР).

Подпись студента: