МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №7

по дисциплине: Алгоритмы и структуры данных
по теме: «Структуры данных типа «дерево» (Pascal/C)»

Выполнил: студент группы ПВ-223
Мелехов Артём Дмитриевич

Проверили:
асс. Солонченко Роман Евгеньевич

Белгород 2023 г.

Лабораторная работа №7. Структуры данных типа «дерево» (C)

Цель работы: изучить СД типа «дерево», научиться их программно реализовывать и использовать.

Содержание отчета:

Решения заданий. Для каждого задания указаны:

– Название задания;

– Условие задания;

– Решение задания;

Вывод;

Полный листинг программы (с названиями директорий и файлов) и ссылка на репозиторий GitHub с выполненной работой.

Вариант №7.

Номер модуля: 7

Вариант задачи: 7

Задание №1.

Для СД типа «дерево» определить:

Пункт 1:

Абстрактный уровень представления СД.

Решение:

Характер организованности:

Дерево состоит из узлов, связанных ветвями.
Каждый узел имеет родительский узел (кроме корневого узла) и может иметь несколько дочерних узлов.
Узлы, не имеющие дочерних узлов, называются листьями.
Узлы, имеющие хотя бы одного потомка, называются внутренними узлами.

Характер изменчивости:

Деревья могут быть статическими или динамическими.
В статическом дереве количество узлов и их связи не меняются после создания дерева.
В динамическом дереве узлы могут добавляться или удаляться, а связи между узлами могут изменяться.

Набор допустимых операций:

Вставка: Добавление нового узла в дерево.
Удаление: Удаление узла из дерева.
Поиск: Поиск узла в дереве.
Обход: Посещение всех узлов дерева в определенном порядке.
Чтение: Получение значения узла.
Запись: Изменение значения узла.

Пункт 2:

Физический уровень представления СД.

Решение:

Схема хранения:

Деревья обычно хранятся в памяти компьютера как набор связанных узлов. Каждый узел содержит значение и ссылки на его дочерние узлы.

Объем памяти, занимаемый экземпляром СД:

Объем памяти, занимаемый деревом, зависит от количества узлов в дереве и размера данных, хранящихся в каждом узле. Каждый узел обычно занимает небольшой объем памяти, но общий объем памяти может быстро увеличиваться с ростом количества узлов.

Формат внутреннего представления СД и способ его интерпретации:

Внутреннее представление дерева обычно включает структуру узла, которая содержит значение узла и ссылки на дочерние узлы. Эта структура интерпретируется как дерево, где каждый узел имеет одного родителя и ноль или более детей.

Характеристика допустимых значений:

Допустимые значения в узлах дерева могут варьироваться в зависимости от конкретного типа дерева. Например, в бинарном дереве поиска значения в левом поддереве любого узла меньше значения узла, а значения в правом поддереве - больше.

Тип доступа к элементам:

Доступ к элементам дерева обычно осуществляется через обход дерева, начиная с корневого узла. Можно использовать различные стратегии обхода, такие как прямой (pre-order), симметричный (in-order) или обратный (post-order) обход.

Пункт 3:

Логический уровень представления СД.

Решение:

Определение:

Дерево – это иерархическая структура данных, состоящая из узлов, которые связаны ветвями. Каждый узел имеет родительский узел (кроме корневого узла) и может иметь несколько дочерних узлов.

Свойства:

Корневой узел: Узел, не имеющий родителя.

Листовые узлы: Узлы, не имеющие детей.

Внутренние узлы: Узлы, имеющие хотя бы одного ребенка.

Уровень узла: Глубина узла в дереве. Корневой узел находится на уровне 0, его дети на уровне 1 и т.д.

Высота дерева: Максимальный уровень любого узла в дереве.

Операции:

Вставка узла: Добавление нового узла в дерево.

Удаление узла: Удаление узла из дерева.

Поиск узла: Поиск узла в дереве по значению.

Обход дерева: Посещение всех узлов дерева в определенном порядке.

Задание №2.

Реализовать СД типа «стек» и «очередь» в соответствии с вариантом инди-видуального задания в виде модуля.

Решение:

Файл `data_structures/tree.h`:

```
#ifndef TREE
#define TREE

#include <stdint.h>
#include <stdbool.h>
#include <stdlib.h>
#include <ctype.h>

#define TREE_BUFFER_SIZE 1000

#define TREE_OK 0
#define TREE_NOT_MEM 1
#define TREE_UNDER 2

typedef void *Tree_Base_Type;
typedef size_t Ptr_El;

typedef struct {
    Tree_Base_Type data;
    Ptr_El l_son;
    Ptr_El r_son;
} Element;

typedef Ptr_El Tree;

extern Element mem_tree[TREE_BUFFER_SIZE];
extern int tree_error;
extern size_t Size;
```

```
// инициализация дерева
Tree init_tree(unsigned size);

// создание корня
Tree create_root();

//запись данных
void write_data_tree(Tree t, Tree_Base_Type e);

//чтение
Tree_Base_Type read_data_tree(Tree t);

//1 — есть левый сын, 0 — нет
int is_l_son(Tree t);

//1 — есть правый сын, 0 — нет
int is_r_son(Tree t);

// перейти к левому сыну, где t — адрес ячейки, содержащей адрес
текущей вершины, TS — адрес
// ячейки, содержащей адрес корня левого поддерева(левого сына)
Tree move_to_l_son(Tree t);

//перейти к правому сыну
Tree move_to_r_son(Tree t);

//1 — пустое дерево,0 — не пустое
int is_empty_tree(Tree t);

//удаление листа
void del_tree(Tree t);

/*связывает все элементы массива в список свободных элементов*/
void init_mem();

/*возвращает 1, если в массиве нет свободных элементов,0 — в
противном случае*/
int empty_mem();

/*возвращает номер свободного элемента и исключает его из ССЭ*/
size_t new_mem();

/*делает n-й элемент массива свободным и включает его в ССЭ*/
void dispose_mem(size_t n);

// Строит дерево t по его скобочному представлению input
int build_tree(Tree t, char *input);

// Копирует дерево src в dst.
void copy_tree(Tree dst, Tree src);
```

```
// Возвращает true, если деревья t1 и t2 равны.
bool comp_tree(Tree t1, Tree t2);

#endif
```

Файл data_structures/tree.c:

```
#include "tree.h"

Element mem_tree[TREE_BUFFER_SIZE];
int tree_error = TREE_OK;
size_t Size = 0;

Tree init_tree(unsigned size) {
    if (size < 1) {
        tree_error = TREE_UNDER;

        return 0;
    }

    if (size > TREE_BUFFER_SIZE) {
        tree_error = TREE_NOT_MEM;

        return 0;
    }

    Size = size;

    tree_error = TREE_OK;
    init_mem();

    return 0;
}
```

```
void init_mem() {
    tree_error = TREE_OK;

    if (Size < 1) {
        tree_error = TREE_UNDER;

        return;
    }

    if (Size > TREE_BUFFER_SIZE) {
        tree_error = TREE_NOT_MEM;

        return;
    }

    for (Ptr_El i = 0; i < Size - 1; i++)
        mem_tree[i].l_son = i + 1;

    mem_tree[Size - 1].l_son = 0;
}

Tree create_root() {
    tree_error = TREE_OK;

    size_t new_ind = new_mem();

    if (tree_error != TREE_OK)
        return 0;

    mem_tree[new_ind].r_son = 0;
    mem_tree[new_ind].l_son = 0;

    return new_ind;
}


int empty_mem() {
    tree_error = TREE_OK;

    return mem_tree[0].l_son == 0;
}
```

```
size_t new_mem() {
    tree_error = TREE_OK;

    Ptr_El result = mem_tree[0].l_son;

    if (!result) {
        tree_error = TREE_NOT_MEM;

        return 0;
    }

    mem_tree[0].l_son = mem_tree[result].l_son;

    return result;
}

void dispose_mem(size_t n) {
    tree_error = TREE_OK;

    if (!n)
        return;

    Ptr_El oldElement = mem_tree[0].l_son;

    mem_tree[0].l_son = n;
    mem_tree[n].l_son = oldElement;
}

void write_data_tree(Tree t, Tree_Base_Type e) {
    tree_error = TREE_OK;
    mem_tree[t].data = e;
}

Tree_Base_Type read_data_tree(Tree t) {
    tree_error = TREE_OK;

    return mem_tree[t].data;
}

int is_l_son(Tree t) {
    tree_error = TREE_OK;

    return mem_tree[t].l_son != 0;
}

int is_r_son(Tree t) {
    tree_error = TREE_OK;

    return mem_tree[t].r_son != 0;
}
```

```
Tree move_to_l_son(Tree t) {
    if (is_l_son(t))
        return mem_tree[t].l_son;

    tree_error = TREE_UNDER;

    return 0;
}

Tree move_to_r_son(Tree t) {
    if (is_r_son(t))
        return mem_tree[t].r_son;

    tree_error = TREE_UNDER;

    return 0;
}

int is_empty_tree(Tree t) {
    tree_error = TREE_OK;

    return !mem_tree[t].r_son && !mem_tree[t].l_son;
}

void del_tree(Tree t) {
    tree_error = TREE_OK;

    if (!t)
        return;

    del_tree(mem_tree[t].r_son);
    del_tree(mem_tree[t].l_son);

    dispose_mem(t);
}
```

```c
#define NAME_BUFFER_SIZE 100

int build_tree(Tree t, char *input) {
    mem_tree[t].l_son = 0;

    char *startInput = input;

    while (isspace(*input))
        input++;

    if (*input != '(')
        return -1;

    input++;

    char *buffer = calloc(NAME_BUFFER_SIZE, sizeof(char));
    int bufferIndex = 0;
    bool shouldWriteData = true;
    bool anyChild = false;

    while (*input != ')')
        if (*input == '\0')
            return -1;
        else if (*input == '(') {
            if (shouldWriteData) {
                write_data_tree(t, buffer);
                shouldWriteData = false;
            }

            size_t newIndex = new_mem();

            if (!anyChild) {
                anyChild = true;
                mem_tree[t].l_son = newIndex;
            } else
                mem_tree[t].r_son = newIndex;

            int res = build_tree(newIndex, input);

            if (res == -1)
                return -1;

            input += res + 1;
            t = newIndex;
            mem_tree[t].r_son = 0;
        } else if (shouldWriteData)
            buffer[bufferIndex] = *(input++);
        else
            input++;

    if (shouldWriteData)
        write_data_tree(t, buffer);
```

```
    return input - startInput;
}

void copy_tree(Tree dst, Tree src) {
    write_data_tree(dst, read_data_tree(src));

    if (tree_error != TREE_OK)
        return;

    Tree r_son = move_to_r_son(src);
    mem_tree[dst].r_son = r_son;

    if (is_r_son(src) && tree_error == TREE_OK) {
        Tree new_tree = new_mem();

        if (tree_error != TREE_OK)
            return;

        mem_tree[dst].r_son = new_tree;

        copy_tree(new_tree, r_son);
    }

    Tree l_son = move_to_l_son(src);

    mem_tree[dst].l_son = l_son;

    if (is_l_son(src) && tree_error == TREE_OK) {
        Tree new_tree = new_mem();

        if (tree_error != TREE_OK)
            return;

        mem_tree[dst].l_son = new_tree;

        copy_tree(new_tree, l_son);
    }
}

bool comp_tree(Tree t1, Tree t2) {
    return ((read_data_tree(t1) == read_data_tree(t2)) &&
tree_error == TREE_OK) &&
           (is_r_son(t1) == is_r_son(t2) ? !is_r_son(t1) ||
comp_tree(mem_tree[t1].r_son,

mem_tree[t2].r_son) : false) &&
           (is_l_son(t1) == is_l_son(t2) ? !is_l_son(t1) ||

comp_tree(mem_tree[t1].l_son, mem_tree[t2].l_son) : false);
}
```

Задание №3.

а) Procedure BildTree(var T:Tree);

Строит дерево в глубину.

б) Function CalcLevel(T:Tree; n:byte):byte;

Определяет количество вершин в дереве T на n-ом уровне.

в) Procedure WriteWays(T:Tree);

Выводит все пути от листьев до корня (в i-ю строку вывода — i-ый путь).

Решение:

Файл lab7/lab7.h:

```
//
// Created by Artyom on 16.11.2023.
//

#ifndef ALGORITMS_AND_DS_LAB7_H
#define ALGORITMS_AND_DS_LAB7_H

#include <stdio.h>

#include "../data_structure/tree.h"

size_t calc_level(Tree t, size_t level);

void write_ways(Tree t);

#endif //ALGORITMS_AND_DS_LAB7_H
```

Файл lab7/lab7.c:

```
//
// Created by Artyom on 16.11.2023.
//

#include "lab7.h"

size_t calc_level(Tree t, size_t level) {
    if (!t)
        return 0;

    if (!level)
        return 1;

    return calc_level(mem_tree[t].l_son, level - 1) +
calc_level(mem_tree[t].r_son,

level - 1);
}
```

```
void print_path(int path[], int path_len) {
    int i;

    for (i = path_len - 1; i >= 0; i--)
        printf("%d ", path[i]);

    printf("\n");
}

void print_paths_recur(Tree t, int path[], int path_len) {
    if (!t)
        return;

    path[path_len] = *(int *) mem_tree[t].data; // явное
приведение типа
    path_len++;

    if (!mem_tree[t].l_son && !mem_tree[t].r_son)
        print_path(path, path_len);
    else {
        print_paths_recur(mem_tree[t].l_son, path, path_len);
        print_paths_recur(mem_tree[t].r_son, path, path_len);
    }
}

void write_ways(Tree t) {
    int path[1000];

    print_paths_recur(t, path, 0);
}
```

Вывод: в ходе выполнения лабораторной работы изучена ипрограммно реализована СД типа «дерево».

Ссылка на GitHub (lab7): https://github.com/SStaryi/algorithms_and_DS