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)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №7

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

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили: асс. Солонченко Роман
Евгеньевич

Белгород 2023г.

Лабораторная работа №7
«Структуры данных типа «дерево» (Pascal/C)»
Вариант 10

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

1. Для СД типа дерево список» определить:

   1.1. Абстрактный уровень представления СД:

      1.1.1. Характер организованности и изменчивости.
      Характер организованности - **дерево**, один ко многим. Характер изменчивости - **динамический**.

      1.1.2. Набор допустимых операций.
      Инициализация, создание корня, запись данных в дерево, чтение данных из дерева, проверка на наличие левого/правого сына, получение правого/левого сына, проверка дерева на пустоту, удаление листа

   1.2. Физический уровень представления СД:

      1.2.1. Схему хранения.
      Схема хранения - **последовательность.**

      1.2.2. Объем памяти, занимаемый экземпляром СД.
      Элемент состоит из трёх полей: указатель произвольного типа, индекс для элемента слева и индекс на элемент справа. Все они имеют размер 4 байт. $V = 12 \cdot N$, где N - количество элемента деревья.

      1.2.3. Формат внутреннего представления СД и способ его интерпретации.
      Элементы содержатся в статическом массиве, свободные элементы объединяются в список (ССЭ), на начало которого указывает левый сын первого элемента массива.

      1.2.4. Характеристику допустимых значений.
      $Car(C) = \sum_1^{max} \frac{(2 \cdot i)!}{(i+1)(i!)^2} \cdot Car(BaseType) + 1.$

      1.2.5. Тип доступа к элементам.
      Тип доступа к элементам - **прямой**.

   1.3. Логический уровень представления СД.

      1.3.1. Способ описания СД и экземпляра СД на языке программирования.

```
InitTree(10);
Tree root = CreateRoot();
```

2. Реализовать СД типа «дерево» в соответствии с вариантом индивидуального задания (см. табл. 17) в виде модуля.
   main.c (тесты)

```c
#include <assert.h>

#include <algc.h>

#define TAKEN_ELEMENTS ((bool *)MemTree[0].data)

void testInitTree() {
    Tree root = InitTree(0);
    assert(TreeError == TreeUnder);

    root = InitTree(10);
    assert(TreeError == TreeOk && Size == 10);

    root = InitTree(TreeBufferSize + 1);
    assert(TreeError == TreeNotMem);
}

void testCreateRoot() {
    Tree init = InitTree(1);
    Tree root = CreateRoot();
    assert(TreeError == TreeNotMem);

    init = InitTree(2);
    root = CreateRoot();
    assert(TreeError == TreeOk);
    assert(IsEmptyTree(root));
    root = CreateRoot();
    assert(TreeError == TreeNotMem);
}

void testMemEmpty() {
    Tree init = InitTree(1);
    assert(EmptyMem());

    init = InitTree(2);
    assert(!EmptyMem());

    init = InitTree(2);
    Tree root = CreateRoot();
    assert(EmptyMem());
}

void testNewMem() {
    Tree init = InitTree(1);
    Tree newEl = NewMem();
    assert(TreeError == TreeNotMem);

    init = InitTree(2);
    newEl = NewMem();
    assert(TreeError == TreeOk && TAKEN_ELEMENTS[newEl]);
    NewMem();
    assert(TreeError == TreeNotMem);
}
```

```
void testDisposeMem() {
    Tree init = InitTree(2);
    Tree newEl = NewMem();
    assert(TreeError == TreeOk && TAKEN_ELEMENTS[newEl]);
    DisposeMem(newEl);
    assert(TreeError == TreeOk && !TAKEN_ELEMENTS[newEl]);
}

void testWriteReadDataTree() {
    Tree init = InitTree(2);
    Tree root = CreateRoot();

    int someVal = 15;
    WriteDataTree(root, &someVal);
    assert(TreeError == TreeOk);
    assert(*(int*)ReadDataTree(root) == someVal && TreeError == TreeOk);
}

void testIsLSonMoveToLSon() {
    Tree init = InitTree(6);
    Tree root = CreateRoot();

    Tree ex = MoveToLSon(root);
    assert(TreeError == TreeUnder);

    Tree newEl = NewMem();
    MemTree[root].LSon = newEl;
    assert(IsLSon(root) && TreeError == TreeOk && MoveToLSon(root) == newEl && TreeError == TreeOk);
}

void testIsRSonMoveToRSon() {
    Tree init = InitTree(6);
    Tree root = CreateRoot();

    Tree ex = MoveToRSon(root);
    assert(TreeError == TreeUnder);

    Tree newEl = NewMem();
    MemTree[root].RSon = newEl;
    assert(IsRSon(root) && TreeError == TreeOk && MoveToRSon(root) == newEl && TreeError == TreeOk);
}

void testDelTree() {
    Tree init = InitTree(6);
    Tree root = CreateRoot();

    Tree newEl1 = NewMem();
    MemTree[root].LSon = newEl1;
    Tree newEl2 = NewMem();
    MemTree[root].RSon = newEl2;

    DelTree(root);
    assert(TreeError == TreeOk &&
    !TAKEN_ELEMENTS[root] && !TAKEN_ELEMENTS[newEl1] && !TAKEN_ELEMENTS[newEl2]);
```

```
}

void test() {
    testInitTree();
    testCreateRoot();
    testMemEmpty();
    testNewMem();
    testDisposeMem();
    testWriteReadDataTree();
    testIsLSonMoveToLSon();
    testIsRSonMoveToRSon();
    testDelTree();
}

int main() {
    test();

    return 0;
}
```

algc.h (заголовки)

```
#ifndef TREE
#define TREE

#include <stdint.h>

#define TreeBufferSize 1000

#define TreeOk 0
#define TreeNotMem 1
#define TreeUnder 2

typedef void* TreeBaseType;
typedef size_t PtrEl;

typedef struct {
    TreeBaseType data;
    PtrEl LSon;
    PtrEl RSon;
} Element;

typedef PtrEl Tree;

extern Element MemTree[TreeBufferSize];
extern int TreeError;
extern size_t Size;
// инициализация дерева
Tree InitTree(unsigned size);

// создание корня
// Эта функция должна вызываться в начале программы
Tree CreateRoot();
```

```
//запись данных
void WriteDataTree(Tree T, TreeBaseType E);

//чтение
TreeBaseType ReadDataTree(Tree T);

//1 – есть левый сын, 0 – нет
int IsLSon(Tree T);

//1 – есть правый сын, 0 – нет
int IsRSon(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 MoveToLSon(Tree T);

//перейти к правому сыну
Tree MoveToRSon(Tree T);

//1 – пустое дерево,0 – не пустое
int IsEmptyTree(Tree T);

//удаление листа
void DelTree(Tree T);

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

/*возвращает 1, если в массиве нет свободных элемен-
тов,0 – в противном случае*/
int EmptyMem();

/*возвращает номер свободного элемента и ис-
ключает его из ССЭ*/
size_t NewMem();

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

int BuildTree(Tree T, char* input);
void CopyTree(Tree dst, Tree src);
bool CompTree(Tree T1, Tree T2);

#endif
```

tree.c (реализации функций)

```
#include <algc.h>
#include <stdbool.h>
#include <stdlib.h>
```

```
Element MemTree[TreeBufferSize];
int TreeError = TreeOk;
size_t Size = 0;

#define TAKEN_ELEMENTS ((bool *)MemTree[0].data)

Tree InitTree(unsigned size) {
    if (size < 1) {
        TreeError = TreeUnder;
        return 0;
    }

    if (size > TreeBufferSize) {
        TreeError = TreeNotMem;
        return 0;
    }
    Size = size;

    TreeError = TreeOk;
    InitMem();

    return 0;
}

void InitMem() {
    if (Size < 1) {
        TreeError = TreeUnder;
        return;
    }

    if (Size > TreeBufferSize) {
        TreeError = TreeNotMem;
        return;
    }

    TreeError = TreeOk;
    bool *takenElements = calloc(Size, sizeof(bool));

    MemTree[0].LSon = 0;
    MemTree[0].RSon = 0;
    MemTree[0].data = takenElements;
    TAKEN_ELEMENTS[0] = true;
}

Tree CreateRoot() {
    size_t newInd = NewMem();
    if (TreeError != TreeOk) return 0;

    TAKEN_ELEMENTS[newInd] = true;
    MemTree[newInd].data = NULL;
    MemTree[newInd].RSon = 0;
    MemTree[newInd].LSon = 0;
```

```
    return newInd;
}


int EmptyMem() {
    TreeError = TreeOk;

    for (size_t i = 0; i < Size; i++)
        if (!TAKEN_ELEMENTS[i]) return false;

    return true;
}

size_t NewMem() {
    TreeError = TreeOk;
    for (size_t i = 0; i < Size; i++) {
        if (!TAKEN_ELEMENTS[i]) {
            TAKEN_ELEMENTS[i] = true;
            MemTree[i].data = NULL;
            MemTree[i].LSon = 0;
            MemTree[i].RSon = 0;

            return i;
        };
    }

    TreeError = TreeNotMem;
    return 0;
}

void DisposeMem(size_t n) {
    TreeError = TreeOk;
    TAKEN_ELEMENTS[n] = false;
}

void WriteDataTree(Tree T, TreeBaseType E) {
    if (!TAKEN_ELEMENTS[T]) {
        TreeError = TreeUnder;
        return;
    }

    TreeError = TreeOk;
    MemTree[T].data = E;
}

TreeBaseType ReadDataTree(Tree T) {
    if (!TAKEN_ELEMENTS[T]) {
        TreeError = TreeUnder;
        return NULL;
    }

    TreeError = TreeOk;
    return MemTree[T].data;
}
```

```
int IsLSon(Tree T) {
    if (!TAKEN_ELEMENTS[T]) {
        TreeError = TreeUnder;
        return false;
    }

    TreeError = TreeOk;
    return MemTree[T].LSon != 0 && TAKEN_ELEMENTS[MemTree[T].LSon];
}

int IsRSon(Tree T) {
    if (!TAKEN_ELEMENTS[T]) {
        TreeError = TreeUnder;
        return false;
    }

    TreeError = TreeOk;
    return MemTree[T].RSon != 0 && TAKEN_ELEMENTS[MemTree[T].RSon];
}

Tree MoveToLSon(Tree T) {
    if (IsLSon(T)) return MemTree[T].LSon;

    TreeError = TreeUnder;
    return 0;
}

Tree MoveToRSon(Tree T) {
    if (IsRSon(T)) return MemTree[T].RSon;

    TreeError = TreeUnder;
    return 0;
}

int IsEmptyTree(Tree T) {
    TreeError = TreeOk;

    return MemTree[T].RSon == 0;
}

void _DelSubTree(Tree T) {
    if (!TAKEN_ELEMENTS[T]) {
        TreeError = TreeUnder;
        return;
    }

    if (IsRSon(T))
        _DelSubTree(MemTree[T].RSon);
    MemTree[T].RSon = 0;

    if (IsLSon(T))
        _DelSubTree(MemTree[T].LSon);
    MemTree[T].LSon = 0;
```

```
    MemTree[T].data = NULL;

    DisposeMem(T);
}

void DelTree(Tree T) {
    TreeError = TreeOk;
    // Please do not the important
    if (T == 0)
        return;

    if (!TAKEN_ELEMENTS[T]) {
        TreeError = TreeUnder;
        return;
    }

    if (IsRSon(T)) {
        _DelSubTree(MemTree[T].RSon);
        MemTree[T].RSon = 0;
    }

    if (IsLSon(T)) {
        _DelSubTree(MemTree[T].LSon);
        MemTree[T].LSon = 0;
    }
    DisposeMem(T);
}
```

3. Разработать программу для решения задачи в соответствии с вариан- том индивидуального задания (см. табл.17) с использованием модуля, по- лученного в результате выполнения пункта 2 задания.
main.c (основная программа)

```
#include <algc.h>
#include <ctype.h>
#include <malloc.h>
#include <stdbool.h>
#include <stdio.h>
#include <assert.h>

int main() {
    InitTree(TreeBufferSize);
    Tree fromBrackets = CreateRoot();
    int res = BuildTree(fromBrackets, "(A(B(C)(D))(e(F)(G)(H)))");

    if (res == -1) {
        printf("unable to parse");
        return 1;
    }

    Tree secondRoot = CreateRoot();
    CopyTree(secondRoot, fromBrackets);
```

```
    assert(CompTree(fromBrackets, secondRoot));

    return 0;
}
```

tree.c (реализации функций)

```
#define NAME_BUFFER_SIZE 100

int BuildTree(Tree T, char* input) {
    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) {
                WriteDataTree(T, buffer);
                shouldWriteData = false;
            }

            size_t newIndex = NewMem();
            if (!anyChild) {
                anyChild = true;
                MemTree[T].RSon = newIndex;
            } else
                MemTree[T].LSon = newIndex;

            int res = BuildTree(newIndex, input);
            if (res == -1) return -1;

            input += res + 1;
            T = newIndex;
        } else if (shouldWriteData)
            buffer[bufferIndex] = *(input++);
        else input++;
    }

    if (shouldWriteData) WriteDataTree(T, buffer);

    return input - startInput;
```

```
}

void CopyTree(Tree dst, Tree src) {
    WriteDataTree(dst, ReadDataTree(src));
    if (TreeError != TreeOk) return;

    Tree RSon;
    if ((RSon = MoveToRSon(src)) && TreeError == TreeOk) {
        Tree newTree = NewMem();
        if (TreeError != TreeOk)
            return;
        MemTree[dst].RSon = newTree;

        CopyTree(newTree, RSon);
    }

    Tree LSon;
    if ((LSon = MoveToLSon(src)) && TreeError == TreeOk) {
        Tree newTree = NewMem();
        if (TreeError != TreeOk)
            return;
        MemTree[dst].LSon = newTree;

        CopyTree(newTree, LSon);
    }
}

bool CompTree(Tree T1, Tree T2) {
    return ((ReadDataTree(T1) == ReadDataTree(T2)) && TreeError == TreeOk) &&
    (IsRSon(T1) == IsRSon(T2) ? !IsRSon(T1) || CompTree(MemTree[T1].RSon, MemTree[T2].RSon) : false ) &&
    (IsLSon(T1) == IsLSon(T2) ? !IsLSon(T1) || CompTree(MemTree[T1].LSon, MemTree[T2].LSon) : false );
}
```

Вывод: в ходе лабораторной работы изучили СД типа «дерево», научились их программно реализовывать и использовать.