**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ**

**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

автоматизированныхсистем

Лабораторная работа №7

дисциплина: Алгоритмы и структуры данных

тема: «Структура данных типа «дерево» »

Выполнил ст. группы ВТ

Проверил: проф. Синюк В.Г.

Белгород 20

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

**Задания**

1. Для СД «дерево» определить:

1.1. Абстрактный уровень представления СД:

1.1.1. Характер организованности и изменчивости.

1.1.2. Набор допустимых операций.

1.2. Физический уровень представления СД:

1.2.1. Схему хранения.

1.2.2. Объем памяти, занимаемый экземпляром СД.

1.2.3. Формат внутреннего представления СД и способ его интерпретации.

1.2.4. Характеристику допустимых значений.

1.2.5. Тип доступа к элементам.

1.3. Логический уровень представления СД.

Способ описания СД и экземпляра СД на языке программирования.

2. Реализовать СД «дерево» в соответствии с вариантом индивидуального (табл.17) задания в виде модулей на языке Pascal или С.

3. Разработать программы на языке Pascal или С для решения задачи в соответствии с вариантом индивидуального задания (см. табл.17) с использованием модулей, полученных в результате выполнения пункта 2 задания.

**Задание №1.**

1.1. Абстрактный уровень представления СД:

1.1.1. Характер организованности и изменчивости: **динамическая, иерархическая структура данных - дерево.**

1.1.2. Набор допустимых операций: **инициализация, создание корня, запись данных, чтение данных, проверка — есть ли левый сын, проверка — есть ли правый сын, переход к левому сыну, переход к правому сыну, проверка — пустое ли дерево, удаление листа.**

1.2. Физический уровень представления СД:

1.2.1. Схему хранения: **связная схема хранения.**

1.2.2. Объем памяти, занимаемый экземпляром СД: **Зависит от количества элементов в дереве V = k \* size; size – размер занимаемый одним элементом структуры «дерево».**

1.2.3. Формат внутреннего представления СД и способ его интерпретации: **Может храниться как в статической, так и в динамической памяти.**

1.2.4. Характеристику допустимых значений: CAR(БД) = **((2i)! / ((i + 1)(i!)2))·CAR(BaseType) + 1,i=1…max, где CAR(BaseType) — кардинальное число элемента БД типа BaseType, max — максимальное количество элементов в БД (не всегда определено, т.к. может зависеть от объема свободной динамической памяти).**

1.2.5. Тип доступа к элементам: **иерархический (от корня к листьям).**

1.3. Логический уровень представления СД.

Способ описания СД и экземпляра СД на языке программирования:

typedef <базовыйтип> t\_base;

typedef unsigned char ptrel;

typedef struct element { t\_base data; ptrel lson, rson; } t\_element;

typedef ptrel t\_tree;

**Задание №2**

**tree.h**

#ifndef **ASD\_7\_TREE\_H**#define **ASD\_7\_TREE\_H**#define **SizeMem** 100  
  
#define **TreeOk** 0  
#define **TreeNotMem** 1  
#define **TreeUnder** 2  
  
**extern int** TreeError;  
  
**typedef int** BaseType;  
**typedef unsigned char** PtrEl;  
**typedef struct** element{  
 BaseType Data;  
 PtrEl LSon;  
 PtrEl RSon;  
}element;  
**typedef** PtrEl \*Tree;  
  
element MemTree[**SizeMem**];  
  
**void** InitTree(Tree \*T);  
**void** CreateRoot(Tree \*T);  
**void** WriteDataTree(Tree \*T, BaseType E);  
**void** ReadDataTree(Tree \*T, BaseType \*E);  
**int** IsLSon(Tree \*T);  
**int** IsRSon(Tree \*T);  
**void** MoveToLSon(Tree \*T, Tree \*TS);  
**void** MoveToRSon(Tree \*T, Tree \*TS);  
**int** IsEmptyTree(Tree \*T);  
**void** DellTree(Tree \*T);  
  
**void** InitMem();  
**int** EmptyMem();  
**int** NewMem();  
**void** DisposeMem(**int** n);  
  
#endif *//ASD\_7\_TREE\_H*

**tree.c**

#include **"tree.h"**#include **<stdio.h>**#include **<stdlib.h>**#include **"tree.h"  
  
int** TreeError;  
  
**void** InitTree(Tree \*T)  
{  
 InitMem();  
 \*T = **NULL**;  
 TreeError = **TreeOk**;  
}  
  
**void** CreateRoot(Tree \*T)  
{  
 **if** (!EmptyMem())  
 {  
 **int** k = NewMem();  
 MemTree[k].LSon = 0;  
 MemTree[k].RSon = 0;  
 \*T = k;  
 TreeError = **TreeOk**;  
 }  
 **else** TreeError = **TreeNotMem**;  
}  
  
**void** WriteDataTree(Tree \*T, BaseType E)  
{  
 **int** k = \*T;  
 MemTree[k].Data = E;  
 TreeError = **TreeOk**;  
}  
  
**void** ReadDataTree(Tree \*T, BaseType \*E)  
{  
 **int** k = \*T;  
 \*E = MemTree[k].Data;  
 TreeError = **TreeOk**;  
}  
  
**int** IsLSon(Tree \*T)  
{  
 **int** k = \*T;  
 TreeError = **TreeOk**;  
 **return** (MemTree[k].LSon != 0);  
}  
  
**int** IsRSon(Tree \*T)  
{  
 **int** k = \*T;  
 TreeError = **TreeOk**;  
 **return** (MemTree[k].RSon != 0);  
}  
  
**void** MoveToLSon(Tree \*T, Tree \*TS)  
{  
 **int** k = \*T;  
 **if** (MemTree[k].LSon)  
 {  
 \*TS = MemTree[k].LSon;  
 TreeError = **TreeOk**;  
 }  
 **else** TreeError = **TreeUnder**;  
}  
  
**void** MoveToRSon(Tree \*T, Tree \*TS)  
{  
 **int** k = \*T;  
 **if** (IsRSon(T))  
 {  
 \*TS = MemTree[k].RSon;  
 TreeError = **TreeOk**;  
 }  
 **else** TreeError = **TreeUnder**;  
}  
  
**int** IsEmptyTree(Tree \*T)  
{  
 **return** !(IsLSon(T) || IsRSon(T));  
}  
  
**void** DellTree(Tree \*T)  
{  
 Tree \*TS;  
 **if** (IsRSon(\*T))  
 {  
 MoveToRSon(\*T, \*TS);  
 DellTree(\*TS);  
 }  
 **if** (IsLSon(\*T))  
 {  
 MoveToLSon(\*T, \*TS);  
 DellTree(\*TS);  
 }  
 **int** k = \*T;  
 DisposeMem(k);  
}  
  
  
**void** InitMem()  
{  
 **for** (**int** i = 0; i < **SizeMem** - 1; i++)  
 {  
 MemTree[0].Data = **NULL**;  
 MemTree[0].LSon = **NULL**;  
 MemTree[i].RSon = i + 1;  
 }  
}  
  
**int** EmptyMem()  
{  
 **return** (MemTree[**SizeMem** - 1].Data != **NULL**);  
}  
  
**int** NewMem()  
{  
 **unsigned char** t = MemTree[0].RSon;  
 MemTree[0].RSon = MemTree[t].RSon;  
 **return** t;  
}  
  
**void** DisposeMem(**int** n)  
{  
 MemTree[n].Data = **NULL**;  
 MemTree[n].LSon = **NULL**;  
 MemTree[n].RSon = MemTree[0].RSon;  
 MemTree[0].RSon = n;  
}

**Задание №3**

а) Построить дерево арифметического выражения, заданного в ППЗ. Операнды — целочисленные константы.

Операции — «+», «–», «\*» и «div».

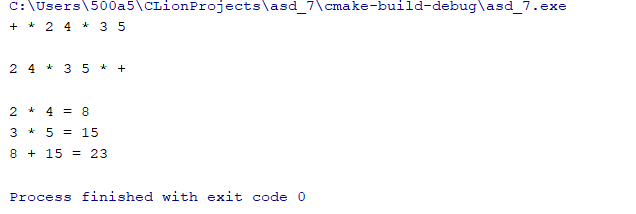
б) Вывести арифметическое выражение в ОПЗ.

в) Вычислить значение по дереву арифметического выражения и выводит результат выполнения каждой операции в виде:

<операнд><операция><операнд>=<значение>

**main.c**:

#include **<stdio.h>**#include **<stdlib.h>**#include **"tree.h"  
  
char** \*s = **"+ \* 2 4 \* 3 5"**;  
  
**int** isOperation(**char** c)  
{  
 **return** (c == **'/'** || c == **'+'** || c == **'-'** || c == **'\*'**);  
}  
  
**int** isDigit(**char** c)  
{  
 **return** (c >= **'0'**&& c <= **'9'**);  
}  
  
**void** BuildTree(Tree \*T)  
{  
 Tree \*TS;  
 **int** k;  
 **if** (\*s != **'\0'**)  
 {  
 **if** (\*s == **' '**)  
 s++;  
 **if** (isOperation(\*s))  
 {  
 WriteDataTree(T,\*s);  
 s++;  
 CreateRoot(&TS);  
 k = \*T;  
 MemTree[k].LSon = TS;  
 MoveToLSon(T, &TS);  
 BuildTree(&TS);  
 CreateRoot(&TS);  
 k = \*T;  
 MemTree[k].RSon = TS;  
 MoveToRSon(T, &TS);  
 BuildTree(&TS);  
 }  
 **if** (isDigit(\*s))  
 {  
 WriteDataTree(T,\*s - **'0'**);  
 s++;  
 }  
 }  
}  
  
**void** WritePostfix(Tree \*T)  
{  
 Tree \*TS;  
 **if** (IsLSon(T))  
 {  
 MoveToLSon(T,&TS);  
 WritePostfix(&TS);  
 }  
 **if** (IsRSon(T))  
 {  
 MoveToRSon(T,&TS);  
 WritePostfix(&TS);  
 }  
 **char** c;  
 ReadDataTree(T, &c);  
 **if** (isDigit(c + **'0'**))  
 c = c + **'0'**;  
 printf(**"%c "**,c);  
}  
  
**int** WriteCalc(Tree \*T)  
{  
  
 Tree \*TS;  
 **int** op1,op2,res;  
 **char** c;  
 **if** (IsLSon(T))  
 {  
 MoveToLSon(T,&TS);  
 op1 = WriteCalc(&TS);  
 }  
 **if** (IsRSon(T))  
 {  
 MoveToRSon(T,&TS);  
 op2 = WriteCalc(&TS);  
 ReadDataTree(T, &c);  
 **switch** (c)  
 {  
 **case '\*'** :  
 res = op1 \* op2;  
 **break**;  
 **case '/'** :  
 res = op1 / op2;  
 **break**;  
 **case '+'** :  
 res = op1 + op2;  
 **break**;  
 **case '-'** :  
 res = op1 - op2;  
 **break**;  
 }  
 printf(**"%i %c %i = %i\n"**, op1, c, op2, res);  
 }  
 ReadDataTree(T, &c);  
 **if** (isOperation(c))  
 **return** res;  
 **else  
 return** c;  
}  
  
**int** main()  
{  
 Tree \*T;  
 InitTree(&T);  
 CreateRoot(&T);  
 printf(**"%s\n\n"**,s);  
 BuildTree(&T);  
 WritePostfix(&T);  
 puts(**"\n"**);  
 WriteCalc(&T);  
}



optimTree.h

#ifndef **ASD\_7\_OPTIMTREE\_H**#define **ASD\_7\_OPTIMTREE\_H**#include**"tree.h"**#include**"search.h"***//Ввод массива длинны n***int**\*inputArr(size\_t n);  
  
*//Возвращает оптимальное дерево поиска по введенным частотам*tree\_searchgetOptimTree();  
  
*//Выделяет память для матрицы n x n***int**\*\*matrLoc(size\_t n);  
  
*//Освобождает память занятую матрицей matr n x n*voidmatrFree(**int**\*\*matr, size\_t n);  
  
*//Строит дерево по матрице R*tree\_searchgetTree(**int**\*\*R, **int**\*a, size\_t n, size\_t i, size\_t j);  
  
#endif *//ASD\_7\_OPTIMTREE\_H*

optimTree.c

#include **"optimTree.h"  
int**\*inputArr(size\_t n) {  
 **int** \*rez = calloc(n, **sizeof**(**int**));  
 **for** (size\_t i = 0; i < n; ++i)  
 scanf(**"%d"**, rez + i);  
  
 **return** rez;  
}  
  
*//Возвращаетоптимальноедеревопоискаповведеннымчастотам*tree\_searchgetOptimTree()  
{  
  
 printf(**"Input n: "**);  
 size\_t n;  
 scanf(**"%d"**, &n);  
  
 printf(**"Input a: "**);  
 **int**\*a = inputArr(n - 1);  
  
 printf(**"Input key a: "**);  
 **int**\*keyA = inputArr(n - 1);  
  
 printf(**"Input b: "**);  
 **int**\*b = inputArr(n);  
 printf(**"Input keyb: "**);  
 **int**\*keyB = inputArr(n);  
 **int**\*\*C = matrLoc(n);  
 **int**\*\*R = matrLoc(n);  
 **int**\*\*W = matrLoc(n);  
*//Инициализация h=0* **for** (size\_t i = 0; i< n; ++i) {  
 C[i][i] = 0;  
 W[i][i] = b[i];  
 R[i][i] = keyB[i];  
 }  
 **for** (size\_t h = 1; h < n; ++h) {  
 **for** (size\_t i = 0; i< n - h; ++i) {  
 size\_t j = i + h;  
 W[i][j] = W[i][j - 1] + a[j - 1] + b[j];  
  
 **int** min = C[i][j - 1] + C[j][j];  
 size\_t m = j;  
 **for** (size\_t k = i + 1; k <= j; ++k) {  
 **if** (C[i][k - 1] + C[k][j] < min) {  
 min = C[i][k - 1] + C[k][j];  
 m = k;  
 }  
 }  
 C[i][j] = W[i][j] + min;  
 R[i][j] = keyA[m - 1];  
 }  
 }  
 **for** (**int** l = 0; l < n; ++l) {  
 **for** (**int** i = 0; i< n; ++i) {  
 printf(**"%d "**, R[l][i]);  
 }  
 printf(**"\n"**);  
 }  
 Tree tree = getTree(R, keyA, n, 0, n - 1);  
 matrFree(C, n);  
 matrFree(W, n);  
 matrFree(R, n);  
 free(a);  
 free(b);  
 free(keyA);  
 free(keyB);  
 **return** tree;  
  
}  
*//Выделяетпамятьдляматрицы n x n***int**\*\*matrLoc(size\_t n)  
{  
  
 **int**\*\*matr = calloc(n, **sizeof**(**int**\*));  
 **for** (size\_t i = 0; i< n; ++i)  
 matr[i] = calloc(n, **sizeof**(**int**));  
 **return** matr;  
  
}  
  
*//Освобождаетпамятьзанятуюматрицейmatr n x n***void** matrFree(**int**\*\*matr, size\_t n)  
{  
  
 **for** (size\_t i = 0; i< n; ++i)  
 free(matr[i]);  
 free(matr);  
  
}  
  
*//Строитдеревопоматрице R*tree\_searchgetTree(**int**\*\*R, **int**\*a, size\_t n, size\_t i, size\_t j)  
{  
  
 Tree tree;  
 initTree(&tree, R[i][j]);  
 **if**(i != j){  
 size\_t rootIndex = unorderedLineSearch(a, n, R[i][j]);  
 tree->L\_Son = getTree(R, a, n, i, rootIndex);  
 tree->R\_Son = getTree(R, a, n, rootIndex + 1, j);  
 }  
  
  
 **return** tree;  
}

Текст программы

**int**main() {  
  
tree\_search tree = getOptimTree();  
  
DelTree(tree);  
**return** 0;  
}

