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» (C)»**

Выполнил/а: ст. группы ПВ-231
Чупахина София Александровна
Проверил:
Акиньшин Даниил Иванович

Белгород, 2024

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

Задания:

1. Для СД типа «дерево» определить:

   (a) Абстрактный уровень представления СД:

      i. Характер организованности и изменчивости,
      ii. Набор допустимых операций.

   (b) Физический уровень представления СД:

      i. Схему хранения;
      ii. Объем памяти, занимаемый экземпляром СД;
      iii. Формат внутреннего представления СД и способ его интерпретации;
      iv. Характеристику допустимых значений;
      v. Тип доступа к элементам.

   (c) Логический уровень представления СД. Способ описания СД и экземпляра СД на языке программирования.

2. Реализовать СД типа «дерево» в соответствии с вариантом индивидуального задания в виде модуля;

3. Разработать программу для решения задачи в соответствии с вариантом индивидуального задания с использованием модуля, полученного в результате выполнения пункта 2 задания.

Содержание

Задание 1:

Абстрактный уровень:

Задание 1.1.1:

СД типа «дерево» — нелинейная структура, в отличие от остальных СД, реализованных в лабораторных работах ранее. Дерево относится к иерархическим структурам; оно состоит из множества элементов (для дерева они называются вершинами), для которых выполняется следующее условие. В дереве обязательно есть одна начальная вершина, которая называется корнем дерева. Остальные вершины разбиты на несколько попарно непересекающихся множеств, каждое из которых тоже является деревом (поддеревом для исходного). Таким образом, формируется иерархическая связь: у корня дерева есть несколько элементов, зависящих от него, которые называются его потомками — это как раз и будут корни поддеревьев. Но и у каждого из них есть свои несколько потомков — корней соответствующих поддеревьев. Для каждого поддерева цепочка будет длиться, пока в множестве вершин, образующих некоторое поддерево, не останется только один элемент — он будет корнем последнего поддерева без потомков. Такие вершины называются листьями. Количество потомков у вершины называется степенью данной вершины: максимальная степень вершины среди всех вершин дерева называется степенью дерева. Деревья со степенью $m = 2$ называются бинарными деревьями, и в дальнейшем в этой работе, говоря о деревьях, мы будем рассматривать только их.

«Дерево» является динамической СД: в ней могут меняться как сами элементы, так и их количество. Однако если дерево характеризуется еще каким-либо свойством (сбалансированностью, является пирамидой и т. д.), следует проводить добавление и удаление элементов так, чтобы сохранялось этой свойство.

Задание 1.1.2:

Для СД типа «дерево» определены следующие базовые операции, которые должны присутствовать в любой реализации этой структуры:

1. Инициализация,
2. Создание корня,
3. Запись данных,
4. Чтение данных,
5. Проверка — есть ли левый сын,
6. Проверка — есть ли правый сын,
7. Переход к левому сыну
8. Переход к правому сыну
9. Проверка — пустое ли дерево
10. Удаление листа

Физический уровень:

Задание 1.2.1:

Поскольку СД типа «дерево» является не последовательной, а иерархической, то для нее может использоваться только связная схема хранения. В дереве один элемент-родитель относится к нескольким и элементам-потомкам, и адекватно отобразить эти отношения между элементами может лишь связная схема. Таким образом, значение каждой вершины должно быть включено в структуру типа «запись», которая хранит, помимо этого значения, указатели на записи, где хранятся значения вершин-потомков заданной вершины. Количество указателей зависит от степени дерева — для бинарных деревьев оно равно двум. Если вершина является листом, все указатели на вершины-потомки должны быть пустыми; если же вершина имеет меньшее количество потомков, чем значение степени дерева, то только часть указателей являются пустыми.

Задание 1.2.2:

Количество памяти, занимаемой СД типа «дерево», зависит от того, какое количество памяти занимает его базовый тип, какое максимальное количество потомков может иметь вершина — соответственно, сколько указателей должна содержать запись, и сколько вершин содержит дерево. Если базовый тип занимает T байт, для каждого элемента необходимо также хранить 2 (поскольку здесь и далее мы рассматриваем бинарное дерево) указателей типа *unsigned long*, занимающего 8 байт, а дерево содержит K вершин, то количество памяти, занимаемой экземпляром СД «дерево», будет равно $(T + 2) * K$ байт.

Задание 1.2.3:

Найти количество Так как СД типа «дерево» с количеством вершин K использует связную схему хранения, то значения записей, представляющих отдельные вершины дерева, могут храниться в памяти независимо друг от друга. В некоторых реализациях дерево может представляться массивом, где хранятся записи, отображающие вершины, но и в этом массиве они не обязательно идут подряд. Значения в полях записей также могут храниться в памяти компьютера не подряд (либо в порядке, отличном от заданного). Значение элемента переводится в двоичный код в зависимости от базового типа; указатели кодируются как положительные целые числа — переводятся в двоичную систему счисления и дополняются нулями слева до размера в 8 байт.

Задание 1.2.4:

Найти количество допустимых значений для СД типа «бинарное дерево» сложнее, чем для рассматриваемых ранее структур: ведь помимо допустимых значений самих вершин и их количества в дереве, необходимо учесть и их порядок, а он гораздо сложнее, чем таковой у линейных структур. Впрочем, для фиксированного набора значений для i вершин существует формула, показывающая, сколько различных бинарных деревьев можно составить из такого набора: их количество будет равно $\frac{(2i)!}{(i+1)(i!)^2}$. Соответственно, просуммировав все значения этого выражения по i от 1 до некоторого max (максимальное количество элементов определяется объемом памяти, отведенным для хранения дерева), получим, сколько различных конфигураций бинарных

деревьев существует вообще:

$$\sum_{i=1}^{max} \frac{(2i)!}{(i+1)(i!)^2}$$

Кардинальное число для СД типа «бинарное дерево» с базовым типом BaseType в итоге находится как:

$$(\sum_{i=1}^{max} \frac{(2i)!}{(i+1)(i!)^2} \times CAR(BaseType)) + 1$$

.

Задание 1.2.5:

СД типа «дерево» имеет последовательный доступ к элементам: поиск нужного элемента для чтения или записи осуществляется с помощью перехода к левому или правому сыну текущего элемента.

Логический уровень:

СД типа «дерево» является производной СД, и потому сначала необходимо реализовать ее тем или иным способом. Только после этого ее можно описать на логическом уровне (представить на языке программирования). Приведем здесь описания для той реализации, которая будет выполнена в задании 2 этой лабораторной работы (дерево в динамической памяти с базовым типом «целое число»).

```c
typedef int BaseType;

typedef struct Element {
    BaseType data;
    struct Element *LSon;
    struct Element *RSon;
} element;
typedef element* PtrEl;
typedef PtrEl *Tree;

int main() {
    Tree example_tree;
}
```

../АСД 7 си/example.c

Заполнение экземпляра СД типа «дерево» значениями и добавление в него новых элементов без модуля с функциями для работы с этой СД весьма трудоемко, а потому не будем его демонстрировать. Займемся написанием этого модуля в задании 2.

Индивидуальное задание; вариант 21

Модуль 1: Дерево в динамической памяти (базовый тип определяется задачей).

Реализация на языке C:

#if !defined(__TREE1_H)

const TreeOk = 0;

const TreeNotMem = 1;

const TreeUnder = 2;

*typedef /*определить !!!*/ BaseType;*

*typedef struct element *ptrel;*

typedef struct element{basetype data;

ptrel LSon;

ptrel RSon; }

*typedef PtrEl *Tree;*

short TreeError;

*void InitTree(Tree *T)// инициализация — создается элемент, который будет содержать корень дерева*

*void CreateRoot(Tree *T) //создание корня*

*void WriteDataTree(Tree *T, BaseType E) //запись данных*

*void ReadDataTree(Tree *T,BaseType *E)//чтение*

*int IsLSon(Tree *T)//1 — есть левый сын, 0 — нет*

*int IsRSon(Tree *T)//1 — есть правый сын, 0 — нет*

*void MoveToLSon(Tree *T, Tree *TS)// перейти к левому сыну, где T — адрес ячейки, содержащей адрес текущей вершины, TS — адрес ячейки, содержащей адрес корня левого поддерева(левого сына)*

*void MoveToRSon(Tree *T, Tree *TS)//перейти к правому сыну*

*int IsEmptyTree(Tree *T)//1 — пустое дерево,0 — не пустое*

*void DelTree(Tree *T)//удаление листа*

#endif

Задача 2:

а) *Procedure BildTree(var T:Tree)*;

Строит дерево арифметического выражения, заданного в ППЗ. Операнды — целочисленные константы. Операции — «+», «−», «*» и «div».

б) *Procedure WritePostfix(T:Tree);*

Выводит арифметическое выражение в ОПЗ.

в) *Function WriteCalc(T:Tree):integer;*

Вычисляет значение по дереву арифметического выражения и выводит результат выполнения каждой операции в виде: <операнд><операция><операнд>=<значение>

Задание 2:

Разделим код выполнения этого задания на два файла: заголовочный и реализации. В заголовочном файле зададим константы с кодами трех основных ошибок, которые могут возникнуть в ходе работы со списками: TreeNotMem — не удалось выделить место под хранение нового элемента-записи ; TreeUnder — попытка записать элемент в пустое дерево либо прочесть элемент из него. Под хранение кода ошибки отводится переменная. После этого дадим название используемым типам данных: BaseType — тип, элементы которого хранит список (в нашем случае это целые числа int: почему был выбран именно этот тип, поясним при рассмотрении задания 3); element — запись, состоящая из значения вершины и двух указателей на такие же записи — на левый и правый потомки вершины; PtrEl — указатель на экземпляр element; Tree — указатель на экземпляр PtrEl. Немного подкорректировав данный для варианта 21 шаблон, получим такой заголовочный файл:

```
#ifndef __TREE1_H
#define __TREE1_H
const short TreeOk = 0;
const short TreeNotMem = 1;
const short TreeUnder = 2;
short TreeError;

typedef int BaseType;

typedef struct Element {
    BaseType data;
    struct Element *LSon;
    struct Element *RSon;
} element;
typedef element* PtrEl;
typedef PtrEl *Tree;

//Инициализация – создается элемент, который будет содержать корень
   дерева
void InitTree(Tree *T);
//Создание корня
void CreateRoot(Tree *T);
//Запись данных
void WriteDataTree(Tree *T, BaseType E);
//Чтение
void ReadDataTree(Tree *T,BaseType *E);
//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
   поддеревалевого( сына)
```

```
31 void MoveToLSon(Tree *T, Tree *TS);
32 //Перейти к правому сыну
33 void MoveToRSon(Tree *T, Tree *TS);
34 //1 – пустое дерево, 0 – не пустое
35 int IsEmptyTree(Tree *T);
36 //Удаление листа
37 void DelTree(Tree *T);
38
39 #endif
```

../АСД 7 си/tree.h

Реализация функций будет вынесена в отдельный файл со следующим содержанием.

```
1  #ifndef __TREE1_C
2  #define __TREE1_C
3  #include "tree.h"
4  #include <malloc.h>
5
6  void InitTree(Tree *T) {
7      *T = malloc(sizeof(PtrEl));
8      if (*T == NULL) {
9          TreeError = TreeNotMem;
10         exit(TreeError);
11     } else {
12         (**T) = NULL;
13         TreeError = TreeOk;
14     }
15 }
16
17 void CreateRoot(Tree *T) {
18     **T = malloc(sizeof(element));
19     if (**T == NULL) {
20         TreeError = TreeNotMem;
21         exit(TreeError);
22     } else {
23         (**T)->LSon = NULL;
24         (**T)->RSon = NULL;
25         TreeError = TreeOk;
26     }
27 }
28
29 void WriteDataTree(Tree *T, BaseType E) {
30     if (**T == NULL) {
31         TreeError = TreeUnder;
32         exit(TreeError);
33     } else {
34         (**T)->data = E;
35     }
```

```c
}

void ReadDataTree(Tree *T, BaseType *E) {
    if (**T == NULL) {
        TreeError = TreeUnder;
        exit(TreeError);
    } else {
        *E = (**T)->data;
    }
}

int IsLSon(Tree *T) {
    return ((**T)->LSon != NULL);
}

int IsRSon(Tree *T) {
    return ((**T)->RSon != NULL);
}

void MoveToLSon(Tree *T, Tree *TS) {
    *TS = &(**T)->LSon;
}

void MoveToRSon(Tree *T, Tree *TS) {
    *TS = &(**T)->RSon;
}

int IsEmptyTree(Tree *T) {
    return (**T) == NULL;
}

void DelTree(Tree *T) {
    (**T) = NULL;
}

#endif
```

../АСД 7 си/Tree.c

Для нормальных (не вызывающих ошибки и аварийного завершения) сценариев этих функций можно составить автоматизированные тесты и вынести их в отдельный файл тестирования. Будем тестировать функции следующим образом. Сначала рассматривая сценарии выполнения для дерева из одного элемента, а после написания тестов для функций перемещения по дереву, MoveToLSon и MoveToRSon, для некоторых функций рассмотрим также и сценарии, где дерево имеет больше одного элемента. Также везде, кроме как в тестах функций записи и чтения значений, WriteDataTree и ReadDataTree, не будем заполнять значениями созданные деревья.

```c
#include "tree.h"
#include "tree.c"
```

```c
#include <malloc.h>
#include <assert.h>
#include <stdio.h>

void Test_InitTree() {
    Tree T;
    InitTree(&T);
    assert(T != NULL);
}

void Test_CreateRoot() {
    Tree T;
    InitTree(&T);
    CreateRoot(&T);
    assert(T != NULL && *T != NULL);
    assert((*T)->LSon == NULL && (*T)->RSon == NULL);
}

void Test_WriteDataTree() {
    Tree T;
    InitTree(&T);
    CreateRoot(&T);
    WriteDataTree(&T, 5);
    assert(T != NULL && *T != NULL);
    assert((*T)->data == 5 && (*T)->LSon == NULL && (*T)->RSon ==
   NULL);
}

void Test_ReadDataTree() {
    Tree T;
    InitTree(&T);
    CreateRoot(&T);
    WriteDataTree(&T, 5);
    BaseType value;
    ReadDataTree(&T, &value);
    assert(T != NULL && *T != NULL);
    assert((*T)->data == 5 && (*T)->LSon == NULL && (*T)->RSon ==
   NULL);
    assert(value == 5);
}

void Test_IsLSon_IsNot() {
    Tree T;
    InitTree(&T);
    CreateRoot(&T);
    assert(!IsLSon(&T));
}
```

```
48
49 void Test_IsRSon_IsNot() {
50     Tree T;
51     InitTree(&T);
52     CreateRoot(&T);
53     assert(!IsRSon(&T));
54 }
55
56 void Test_MoveToLSon_Empty() {
57     Tree T, TL;
58     InitTree(&T);
59     InitTree(&TL);
60     CreateRoot(&T);
61     MoveToLSon(&T, &TL);
62     assert(T != NULL && *T != NULL);
63     assert((*T)->LSon == NULL && *TL == NULL);
64 }
65
66 void Test_MoveToRSon_Empty() {
67     Tree T, TR;
68     InitTree(&T);
69     InitTree(&TR);
70     CreateRoot(&T);
71     MoveToRSon(&T, &TR);
72     assert(T != NULL && *T != NULL);
73     assert((*T)->RSon == NULL && *TR == NULL);
74 }
75
76 void Test_IsLSon_Is() {
77     Tree T, TL;
78     InitTree(&T);
79     InitTree(&TL);
80     CreateRoot(&T);
81     MoveToLSon(&T, &TL);
82     assert(!IsLSon(&T));
83     CreateRoot(&TL);
84     assert(IsLSon(&T));
85 }
86
87 void Test_IsRSon_Is() {
88     Tree T, TR;
89     InitTree(&T);
90     InitTree(&TR);
91     CreateRoot(&T);
92     MoveToRSon(&T, &TR);
93     CreateRoot(&TR);
94     assert(IsRSon(&T));
```

```
}

void Test_IsEmptyTree_IsEmpty() {
    Tree T;
    InitTree(&T);
    assert(IsEmptyTree(&T));
}

void Test_IsEmptyTree_IsNotEmpty() {
    Tree T;
    InitTree(&T);
    CreateRoot(&T);
    assert(!IsEmptyTree(&T));
}

void Test_DelTree_Root() {
    Tree T;
    InitTree(&T);
    CreateRoot(&T);
    DelTree(&T);
    assert(IsEmptyTree(&T));
}

void Test_DelTree_Subtree() {
    Tree T, TR, TRR;
    InitTree(&T);
    InitTree(&TR);
    InitTree(&TRR);
    CreateRoot(&T);
    MoveToRSon(&T, &TR);
    CreateRoot(&TR);
    MoveToRSon(&TR, &TRR);
    CreateRoot(&TRR);
    DelTree(&TR);
    assert(!IsEmptyTree(&T) && !IsRSon(&T));
    assert(IsEmptyTree(&TR));
    assert(!IsEmptyTree(&TRR));
}

void Test_Tree() {
    Test_InitTree();
    Test_CreateRoot();
    Test_WriteDataTree();
    Test_ReadDataTree();
    Test_IsLSon_IsNot();
    Test_IsRSon_IsNot();
    Test_MoveToLSon_Empty();
```

```
142     Test_MoveToRSon_Empty();
143     Test_IsLSon_Is();
144     Test_IsRSon_Is();
145     Test_IsEmptyTree_IsEmpty();
146     Test_IsEmptyTree_IsNotEmpty();
147     Test_DelTree_Root();
148     Test_DelTree_Subtree();
149 }
150 
151 int main() {
152     Test_Tree();
153     printf("All is OK!");
154     return 0;
155 }
```

../АСД 7 си/tree_test.c

Запустив программу, можем самостоятельно убедиться, что все тесты прошли успешно:

```
"C:\Users\sovac\Desktop\ASD_third_semester\АСД 7 си\tree_test.exe"
All is OK!
Process finished with exit code 0
```

Задание 3:

Теперь, когда модуль для СД типа «дерево» реализован, можем перейти к решению задачи для варианта 21, описанной выше. Базовым типом дерева для этой задачи будет целое число. Как сказано в условии, операндами выражения должны быть целые числа; сами же операции (их всего четыре: сложение, вычитание, умножение, деление нацело) можно обозначать константами 0, 1, 2, 3 соответственно. Ведь при построении дерева арифметического выражения операнды будут его листьями, а операции — вершинами с потомками. И определить, как стоит толковать целое значение вершины — как число или как код операции — очень просто: нужно лишь знать, имеет ли вершина потомков. Затем объявим два вспомогательных массива для связи целочисленных «кодов» операций с их обозначениями в строке («+», «-», «*», «/» соответственно) и функциями, которым соответствуют эти операции. В этих массивах символы, обозначающие операции, и функции идут в указанном выше порядке.

Откорректировав прототипы функций, данные в индивидуальных заданиях, для языка C, можем реализовать их следующим образом.

Примечание: при вводе с клавиатуры рассчитываем, что соблюдаются следующие правила: операнды и операции отделены друг от друга одним пробелом, унарный минус применяется только к отдельным числам и не отделяется пробелом от числа, с которым связан.

```
1 #include "tree.h"
2 #include "tree.c"
3 
```

```c
#include <stdio.h>
#include <ctype.h>

int add(int x, int y) {
    return x + y;
}
int subtract(int x, int y) {
    return x - y;
}
int multiply(int x, int y) {
    return x * y;
}
int divide(int x, int y) {
    return x / y;
}

//Массив символов, обозначающих операции сложения, вычитания, умножения, деления нацело
char marks[4] = {'+', '-', '*', '/'};
//Массив функций, выполняющих операции сложение, вычитание, умножение, деление нацело двух целых чисел
int (*operations[4]) (int, int)  = {add, subtract, multiply, divide};

//Возвращает числовой код операции (0, 1, 2, 3) по символу, обозначающему этому операцию
int getCodeByMark(char marking) {
    for (int i = 0; i < 4; i++) {
        if (marks[i] == marking) {
            return i;
        }
    }
    return -1;
}

//Строит дерево арифметического выражения, заданного в ППЗ. Операнды — целочисленные константы.
//Операции — «»+, «-», «»* и «»div.
void BildTree(Tree *T) {
    char cur_sym = getchar();
    if (cur_sym == '+' || cur_sym == '-' || cur_sym == '*' || cur_sym == '/') {
        char next_sym = getchar();
        if (cur_sym != '-' || isspace(next_sym)) {
            WriteDataTree(T, getCodeByMark(cur_sym));
            Tree TL, TR;
            InitTree(&TL);
            InitTree(&TR);
```

```
46              MoveToLSon(T, &TL);
47              CreateRoot(&TL);
48              WriteDataTree(&TL, 0);
49              BildTree(&TL);
50              MoveToRSon(T, &TR);
51              CreateRoot(&TR);
52              WriteDataTree(&TR, 0);
53              BildTree(&TR);
54          }
55          else {
56              int cur_value = -(next_sym - '0');
57              WriteDataTree(T, cur_value);
58              BildTree(T);
59          }
60      } else if (cur_sym >= '0' && cur_sym <= '9') {
61          while(!isspace(cur_sym)) {
62              int cur_value;
63              ReadDataTree(T, &cur_value);
64              cur_value = cur_value * 10 + (cur_sym - '0') * ((cur_value
    < 0) ? -1 : 1);
65              WriteDataTree(T, cur_value);
66              cur_sym = getchar();
67          }
68      }
69  }
70
71  //Выводит арифметическое выражение в ОПЗ.
72  void WritePostfix(Tree *T) {
73      if (IsLSon(T)) {
74          Tree TL, TR;
75          InitTree(&TL);
76          InitTree(&TR);
77          MoveToLSon(T, &TL);
78          WritePostfix(&TL);
79          MoveToRSon(T, &TR);
80          WritePostfix(&TR);
81          int value;
82          ReadDataTree(T, &value);
83          printf("%c ", marks[value]);
84      } else {
85          int value;
86          ReadDataTree(T, &value);
87          printf("%d ", value);
88      }
89  }
90
91  //Вычисляет значение по дереву арифметического выражения и выводит
```

```
   результат выполнения каждой операции в виде:
92 //операндоперацияоперандзначение<><><>=<>
93 int WriteCalc(Tree *T) {
94     if (IsLSon(T)) {
95         Tree TL, TR;
96         InitTree(&TL);
97         InitTree(&TR);
98         MoveToLSon(T, &TL);
99         int first = WriteCalc(&TL);
100        MoveToRSon(T, &TR);
101        int second = WriteCalc(&TR);
102        int op;
103        ReadDataTree(T, &op);
104        printf("%d %c %d = %d\n", first, marks[op], second,
   operations[op](first, second));
105        return operations[op](first, second);
106    } else {
107        int value;
108        ReadDataTree(T, &value);
109        return value;
110    }
111 }
112
113 int main() {
114    Tree T;
115    InitTree(&T);
116    CreateRoot(&T);
117    BildTree(&T);
118    WritePostfix(&T);
119    printf("\n");
120    WriteCalc(&T);
121 }
```

../АСД 7 си/task.c

Для того, чтобы протестировать полученную программу, будем использовать следующие выражения: а) 3 * 6 - 25 / 4 + 1 — выражение, для отображения которого в инфиксной записи не требуются скобки, б) (6 + 231 / 15) * (4 + 3) — выражение, для отображения которого в инфиксной записи скобки необходимы, в) 5 * (-7 - 9 / -4) — пример с отрицательными числами. В префиксной записи они будут выглядеть так: а) + - * 3 6 / 25 4 1; б) * + 6 / 231 15 + 4 3; в) * 5 - -7 / 9 -4. Для каждой из полученных префиксных записей запустим программу и введем ее с клавиатуры. Вывод будет следующим:

```
"C:\Users\sovac\Desktop\ASD_third_semester\АСД 7 си\task.exe"
+ - * 3 6 / 25 4 1
3 6 * 25 4 / - 1 +
3 * 6 = 18
25 / 4 = 6
18 - 6 = 12
12 + 1 = 13
```

```
"C:\Users\sovac\Desktop\ASD_third_semester\АСД 7 си\task.exe"
* + 6 / 231 15 + 4 3
6 231 15 / + 4 3 + *
231 / 15 = 15
6 + 15 = 21
4 + 3 = 7
21 * 7 = 147
```

```
"C:\Users\sovac\Desktop\ASD_third_semester\АСД 7 си\task.exe"
* 5 - -7 / 9 -4
5 -7 9 -4 / - *
9 / -4 = -2
-7 - -2 = -5
5 * -5 = -25
```

Как можно убедиться, постфиксные (обратные польские) записи построены верно, значения выражений также вычислены без ошибок.

Вывод:

В ходе лабораторной работы дали характеристику СД типа «дерево», форматам ее представления, реализовали один из них в соответствии с вариантом (дерево в динамической памяти с базовым типом «целое число»), написали ряд базовых функций для работы с деревьями в этом формате, а также решили задачу ввода и хранения арифметического выражения с операциями сложения, вычитания, умножения и деления нацело в формате дерева, вывода этого выражения на экран и его вычисления.