|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 4** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Нелинейные структуры данных. Бинарное дерево.»** | |
|  | |
| Выполнил студент группы ИКБО-04-21 | Исаев В.В. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2022

# Цель работы

Получение умений и навыков разработки и реализаций операций над структурой данных бинарное дерево.

# Постановка задачи

Разработать программу в соответствии с требованиями варианта. Выполнить реализацию средствами ООП.

**Для вариантов с 8 по 15**

Вид дерева: дерево выражения.

1. Реализовать операции общие для вариантов с 8 по 15
   1. Создать дерево выражений в соответствии с вводимым выражением. Структура узла дерева включает: информационная часть узла – символьного типа: знак операции +, -, \* или цифра; указатель на левое и указатель на правое поддерево. В дереве выражения операнды выражений находятся в листьях дерева.
   2. Исходное выражение имеет формат:

<формула>::=цифра|<формула><знак операции><формула>

Примеры: 5; 1+2; 1+2+3\*4-5/6.

* 1. Отобразить дерево на экране, повернув его против часовой стрелки.

1. Реализовать операции варианта.
2. Разработать программу, демонстрирующую выполнение всех операций.
3. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Вариант №8. Условия заданий. В таблице 1.1 представлены условия для задания 1.

Таблица 1.1. Условия для задания 1

|  |  |  |
| --- | --- | --- |
| 8 | Символьное значение | Вывод дерева выражений по ширине.  Вернуть самый левый лист дерева.  Вычислить значение выражения |

# Решение

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

Каждый узел в дереве задаёт поддерево, корнем которого он является. У вершины есть два потомка левый и правый и, соответственно, два поддерева левое и правое, корнями которых, они являются.

* 1. Создать дерево выражений в соответствии с вводимым выражением. Структура узла дерева включает: информационная часть узла – символьного типа: знак операции +, -, \* или цифра; указатель на левое и указатель на правое поддерево. В дереве выражения операнды выражений находятся в листьях дерева.

Для дерева была разработана структура «tree\_elem». Структура содержит текущее значение и ссылку на двух её потомков, которые так же являются объектами структуры.

|  |
| --- |
| struct tree\_elem  {  char s;  tree\_elem\* left;  tree\_elem\* right;  }; |

* 1. Исходное выражение имеет формат:

<формула>::=цифра|<формула><знак операции><формула>

Примеры: 5; 1+2; 1+2+3\*4-5/6.

Для реализации задания, была разработана функция, преобразующая вводимую строку в обратную польскую запись по алгоритму. После преобразования введенная строка с использованием структуры стек заносится в дерево.

|  |
| --- |
| string to\_rpn(string str)  {  map<string, int> priority = { {"(", 1}, {")", 1}, {"+", 2}, {"-", 2}, {"\*", 3}, {"/", 3} };  string word;  stack<string> cur\_values;  string pn = "";  istringstream ss(str);  while (ss >> word)  {  bool is\_keyword = false;  for (auto& tmp : priority)  {  if (tmp.first == word)  is\_keyword = true;  }  if (!is\_keyword)  {  pn += " " + word;  }  else  {  if (word == "(")  cur\_values.push(word);  else if (word == ")")  {  while (!cur\_values.empty() && cur\_values.top() != "(")  {  pn += " " + cur\_values.top();  cur\_values.pop();  }  cur\_values.pop();  }  else  {  while (!cur\_values.empty() && priority[cur\_values.top()] >= priority[word])  {  pn += " " + cur\_values.top();  cur\_values.pop();  }  cur\_values.push(word);  }  }  }  while (!cur\_values.empty())  {  pn += " " + cur\_values.top();  cur\_values.pop();  }  return pn;  }  tree\_elem\* parse\_rpn(string rpn\_expression)  {  stack<tree\_elem\*> tree\_stack;  tree\_elem\* te;  for (int i = 0; i < rpn\_expression.size(); i++)  {  if (isdigit(rpn\_expression[i]))  {  te = create\_tree\_element(rpn\_expression[i]);  tree\_stack.push(te);  }  else if (rpn\_expression[i] == '\*' || rpn\_expression[i] == '-' ||rpn\_expression[i] == '+' || rpn\_expression[i] == '/')  {  te = create\_tree\_element(rpn\_expression[i]);  te->right = tree\_stack.top();  tree\_stack.pop();  te->left = tree\_stack.top();  tree\_stack.pop();  tree\_stack.push(te);  }  }  return tree\_stack.top();  }  void insert\_tree\_elem(tree\_elem\* root, tree\_elem\* elem)  {  if (root->s > elem->s)  {  if (root->left == nullptr)  {  root->left = elem;  }  else  {  insert\_tree\_elem(root->left, elem);  }  }  else if (root->s < elem->s)  {  if (root->right == nullptr)  {  root->right = elem;  }  else  {  insert\_tree\_elem(root->right, elem);  }  }  } |

* 1. Отобразить дерево на экране, повернув его против часовой стрелки.

Для реализации задания, была разработана функция, выполняющая обход дерева и выводящая его элементы.

|  |
| --- |
| void print(tree\_elem\* Tree, int l)  {  int i;  if (Tree != nullptr)  {  print(Tree->right, l + 1);  for (i = 1; i <= l; i++)  {  cout << " ";  }  cout << Tree->s << endl;  print(Tree->left, l + 1);  }  } |

1. Реализовать операции варианта.
   1. Вывод дерева выражений по ширине;

Выполнение данного задания представлено в подпункте 1.3. данного пункта.

* 1. Вернуть самый левый лист дерева;

Для реализации задания была написана функция, идущая по левым потомкам корня, пока не найдет последний элемент.

|  |
| --- |
| char find\_left\_elem(tree\_elem\* root)  {  if (root->left == nullptr)  {  return root->s;  }  return find\_left\_elem(root->left);  } |

* 1. Вычислить значение дерева;

Для выполнения задания была разработана функция, которая смотрит значение корня и если оно является знаком, то выполняет необходимую операцию над его потомками. Таким образом, в конце работы функции в переменной содержится значение дерева.

|  |
| --- |
| double find\_result(tree\_elem\* root)  {  if (isdigit(root->s))  {  return double(int(root->s) - 48);  }  if (root->s == '+')  {  return double(find\_result(root->left)) + double(find\_result(root->right));  }  else if (root->s == '-')  {  return double(find\_result(root->left)) - double(find\_result(root->right));  }  else if (root->s == '\*')  {  return double(find\_result(root->left)) \* double(find\_result(root->right));  }  else if (root->s == '/')  {  return double(find\_result(root->left)) / double(find\_result(root->right));  }  return 0.0;  } |

1. Разработать программу, демонстрирующую выполнение всех операций.

Программа была разработана. Функционал будет представлен в пункте 4 данного отчета, полный код в пункте 6 данного отчета

1. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Тестирование программы будет представлено в пункте 4 данного отчета.

# Тестирование

Тестирование приложения.

Для тестирования приложения были разработаны тестовые случаи, представленные в таблице 2.

Таблица 2. Тестовые случаи для приложения

|  |  |  |  |
| --- | --- | --- | --- |
| № теста | № задания | Вводимые данные | Ожидаемый результат |
| 1 | 1 | 1 + 2 + 3 + 4 + 5 | Дерево создано |
| 2 | 1 | 1+2+3 | Неустранимая ошибка в связи с отсутствием разделителей (пробелов) |
| 3 | 2 | Отсутствуют(после ввода дерева) | 15 |
| 4 | 2 | Отсутствуют(до ввода дерева) | Ошибка об отсутствии дерева |
| 5 | 3 | Отсутствуют(после ввода дерева) | Вывод дерева с вершиной слева |
| 6 | 3 | Отсутствуют(до ввода дерева) | Ошибка об отсутствии дерева |
| 7 | 4 | Отсутствуют(после ввода дерева) | 1 |
| 8 | 4 | Отсутствуют(до ввода дерева) | Ошибка об отсутствии дерева |
| 9 | 5 | Отсутствуют | Приложение закрывается |

Тест 1.

Ввод дерева представлен на рисунке 1.

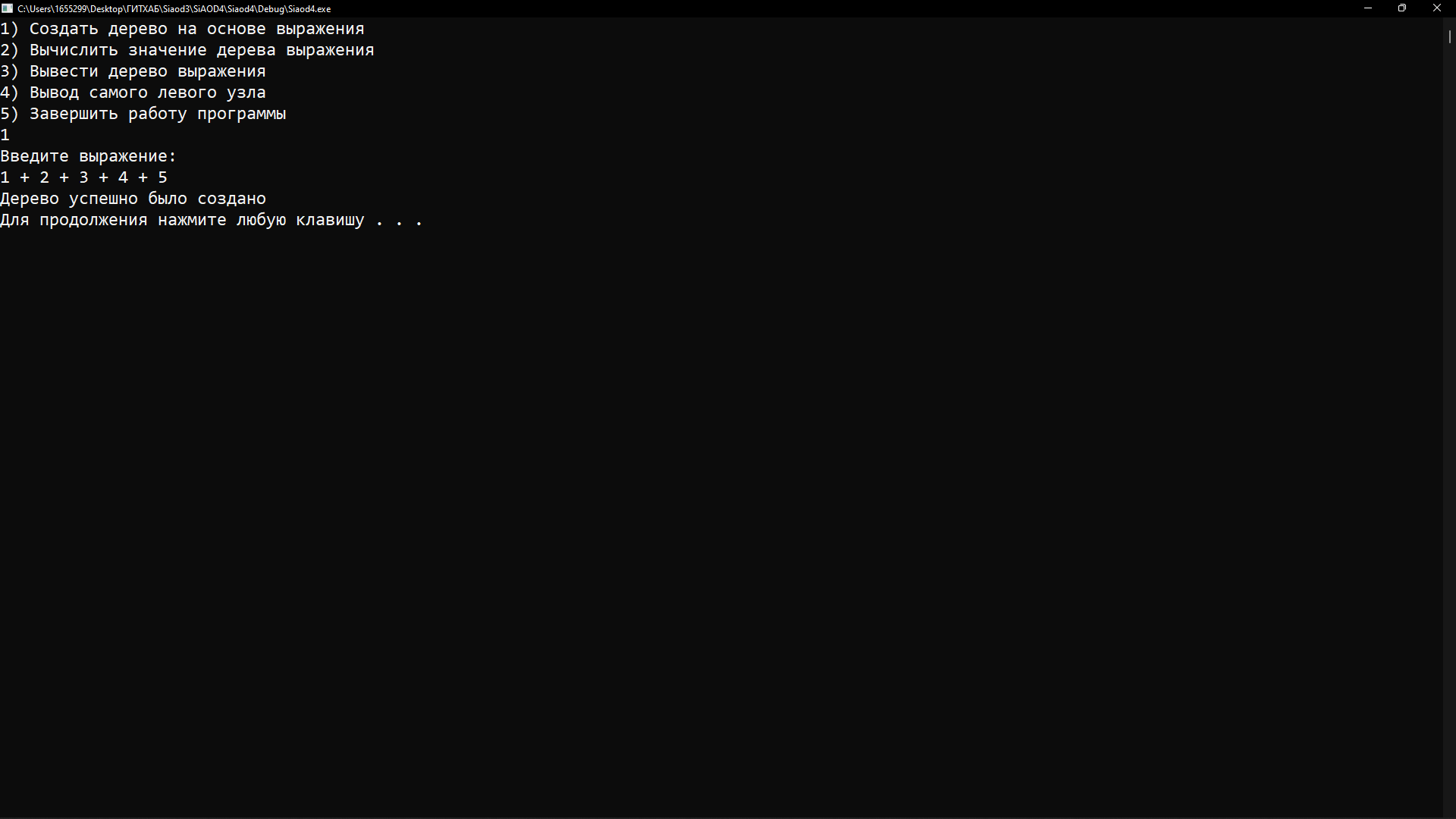


Рис.1 Ввод дерева

Тест 2.

Ввод дерева выражений с ошибкой представлен на рисунке 2.

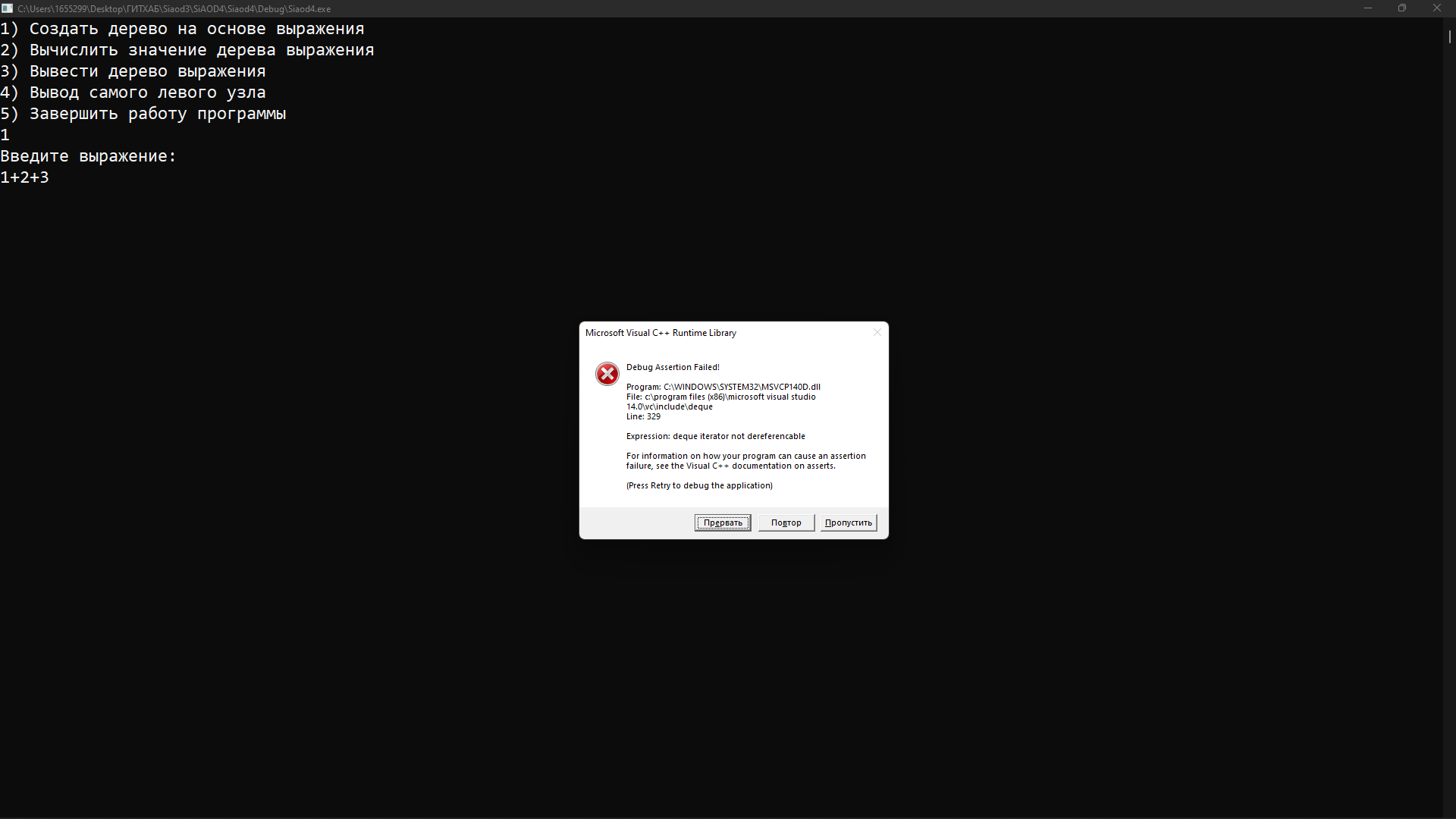


Рис. 2 Ввод дерева выражений с ошибкой

Тест 3.

Вычисление значения выражения представлено на рисунке 3.

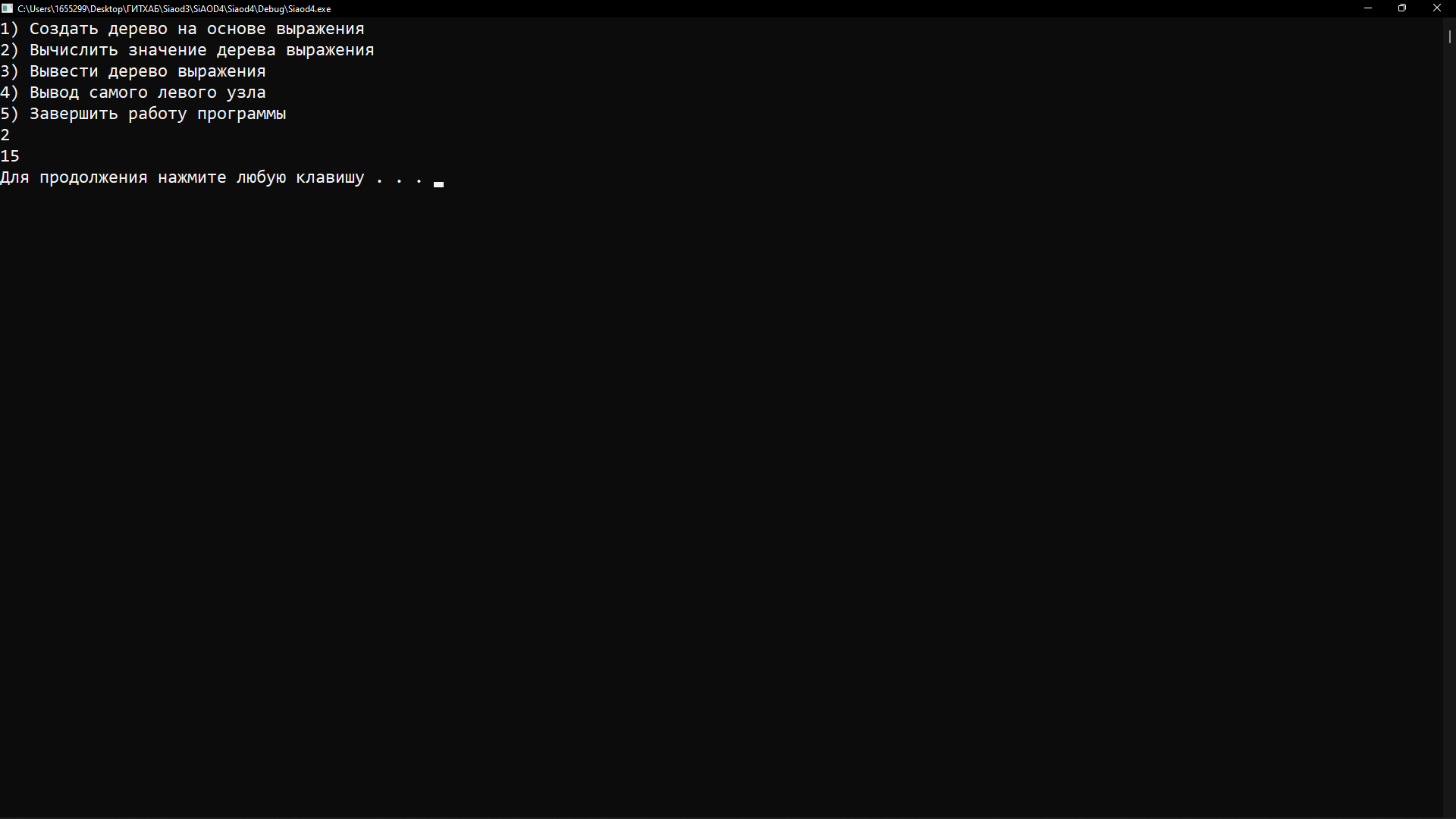


Рис. 3 Вычисление значения выражения

Тест 4.

Ошибка об отсутствии дерева выражений для теста 4 представлена на рисунке 2.

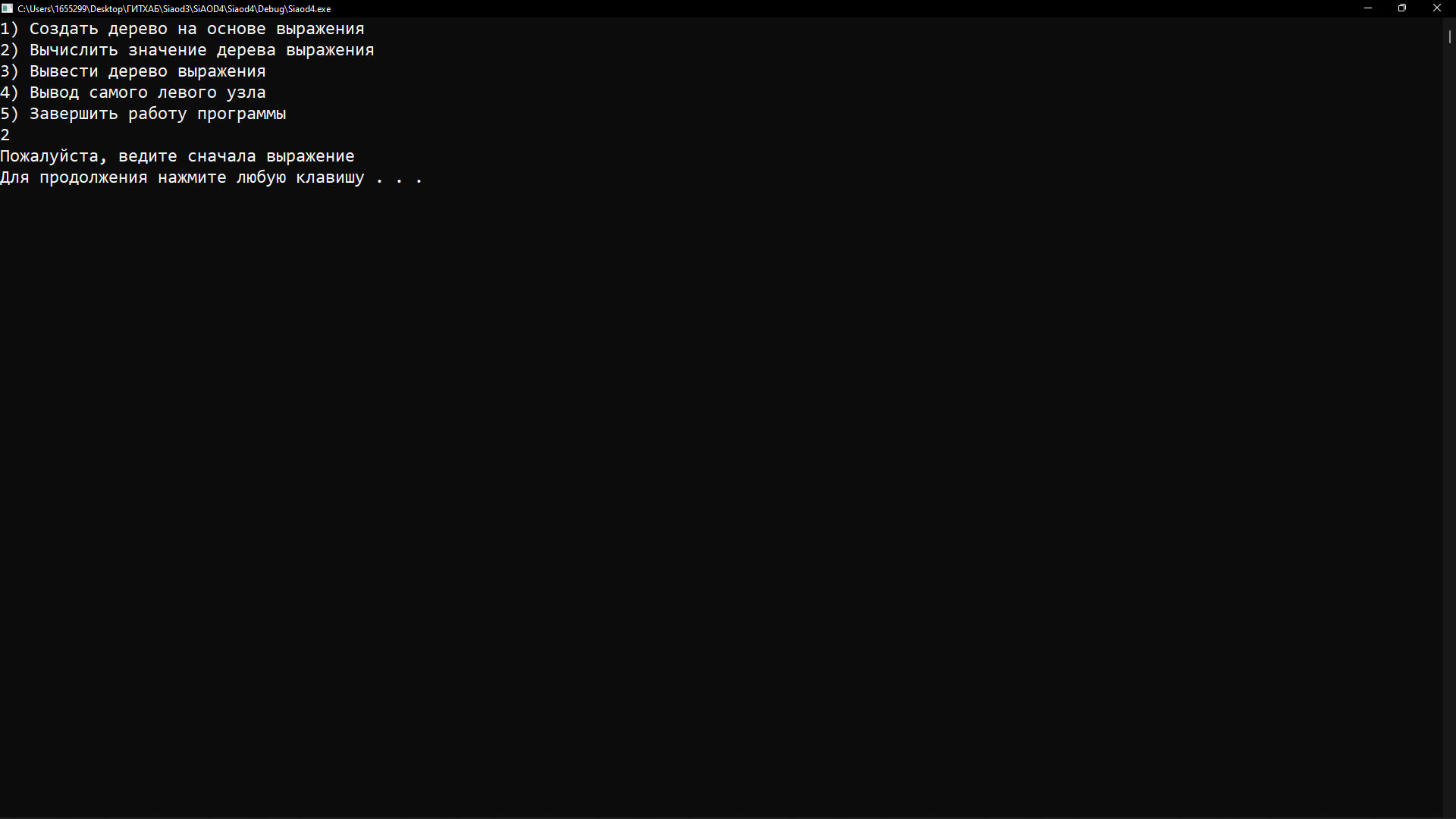


Рис. 4 Ошибка об отсутствии дерева выражений для теста 4

Тест 5.

Вывод дерева выражений представлен на рисунке 5.

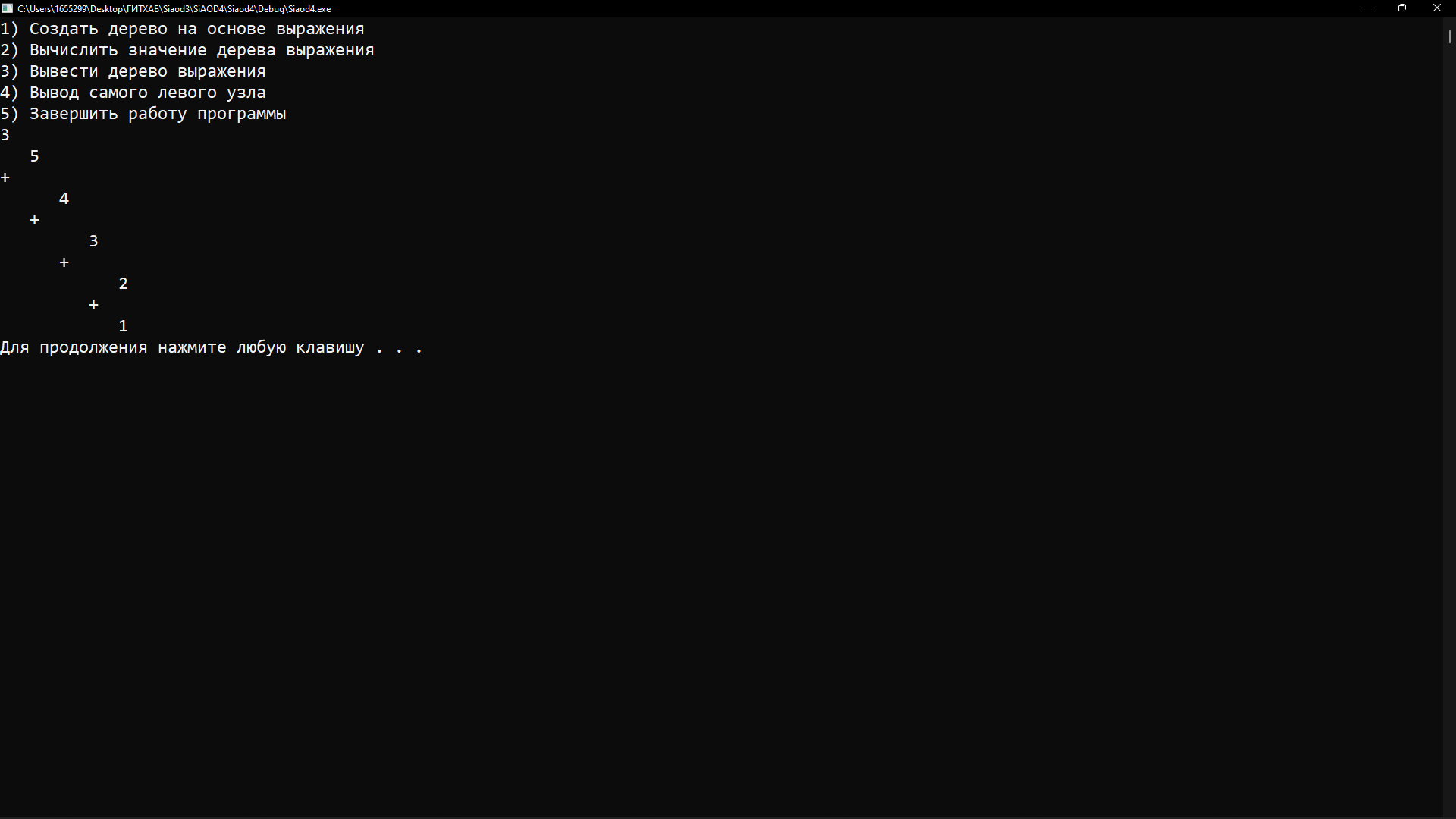


Рис. 5 Вывод дерева выражений

Тест 6.

Ошибка об отсутствии дерева выражений для теста 6 представлена на рисунке 6.

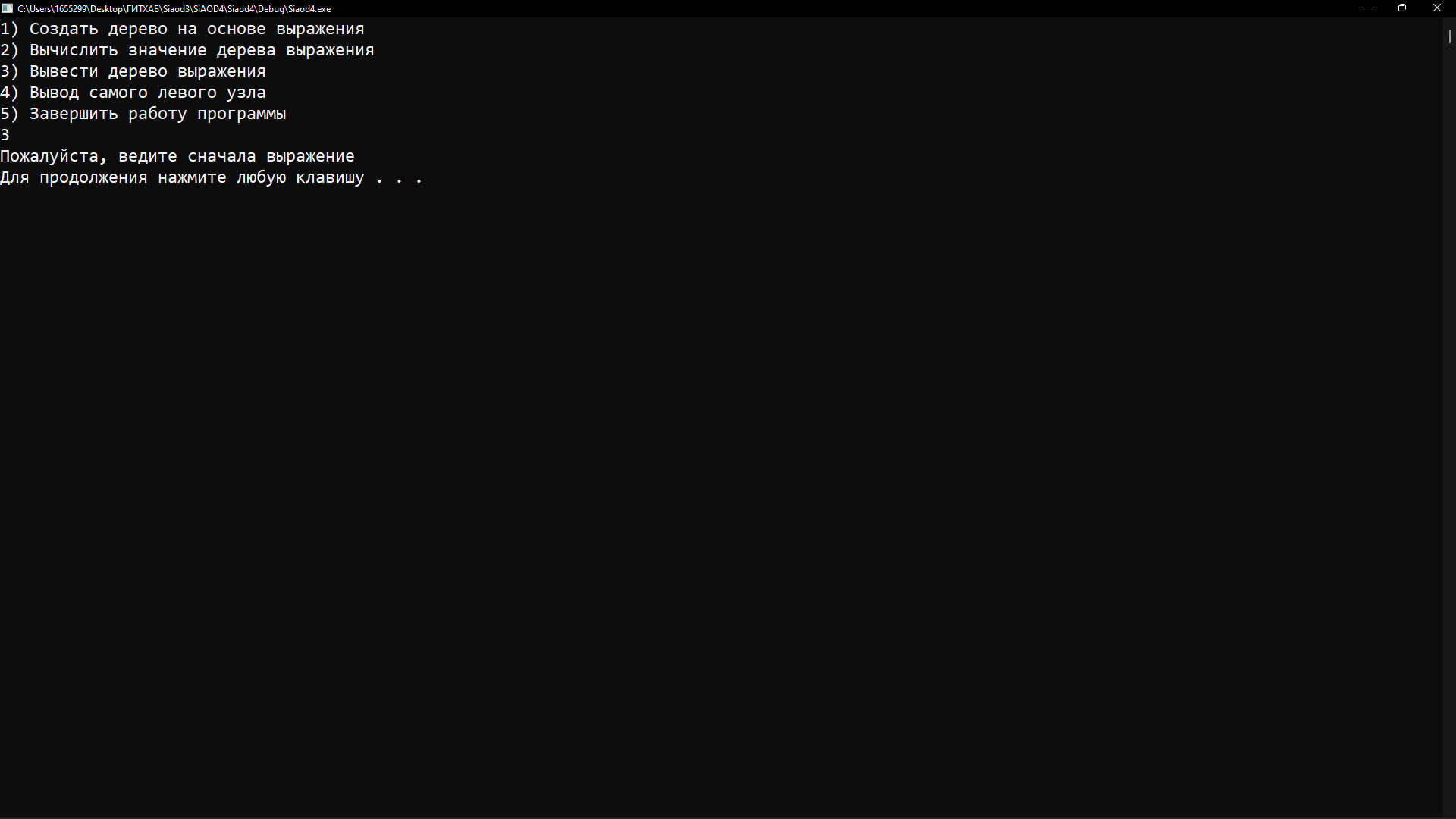


Рис. 6 Ошибка об отсутствии дерева выражений

Тест 7.

Вывод самого левого узла дерева представлен на рисунке 7.

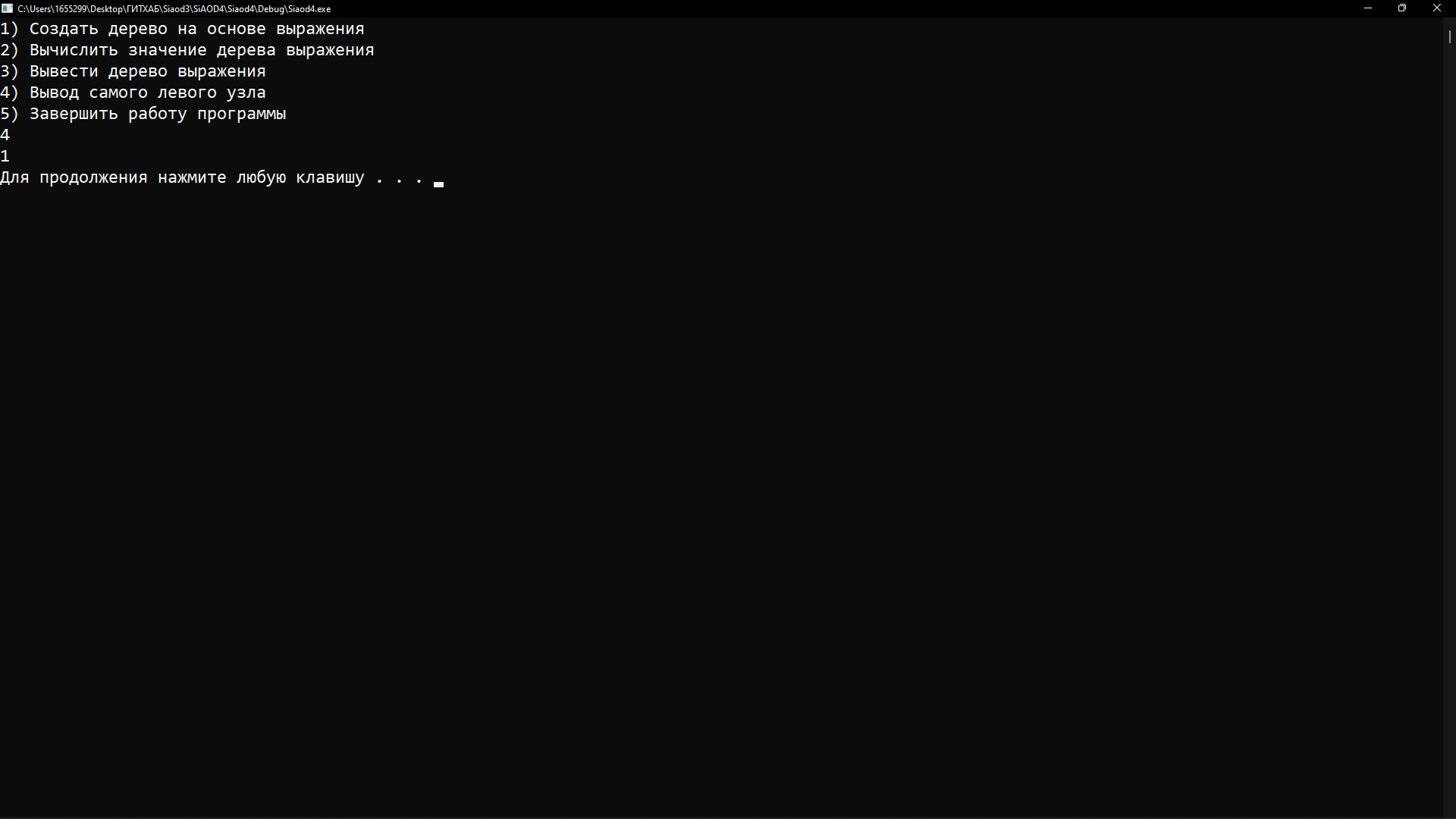


Рис. 7 Вывод самого левого узла дерева

Тест 8.

Ошибка об отсутствии дерева выражений для теста 8 представлена на рисунке 8.

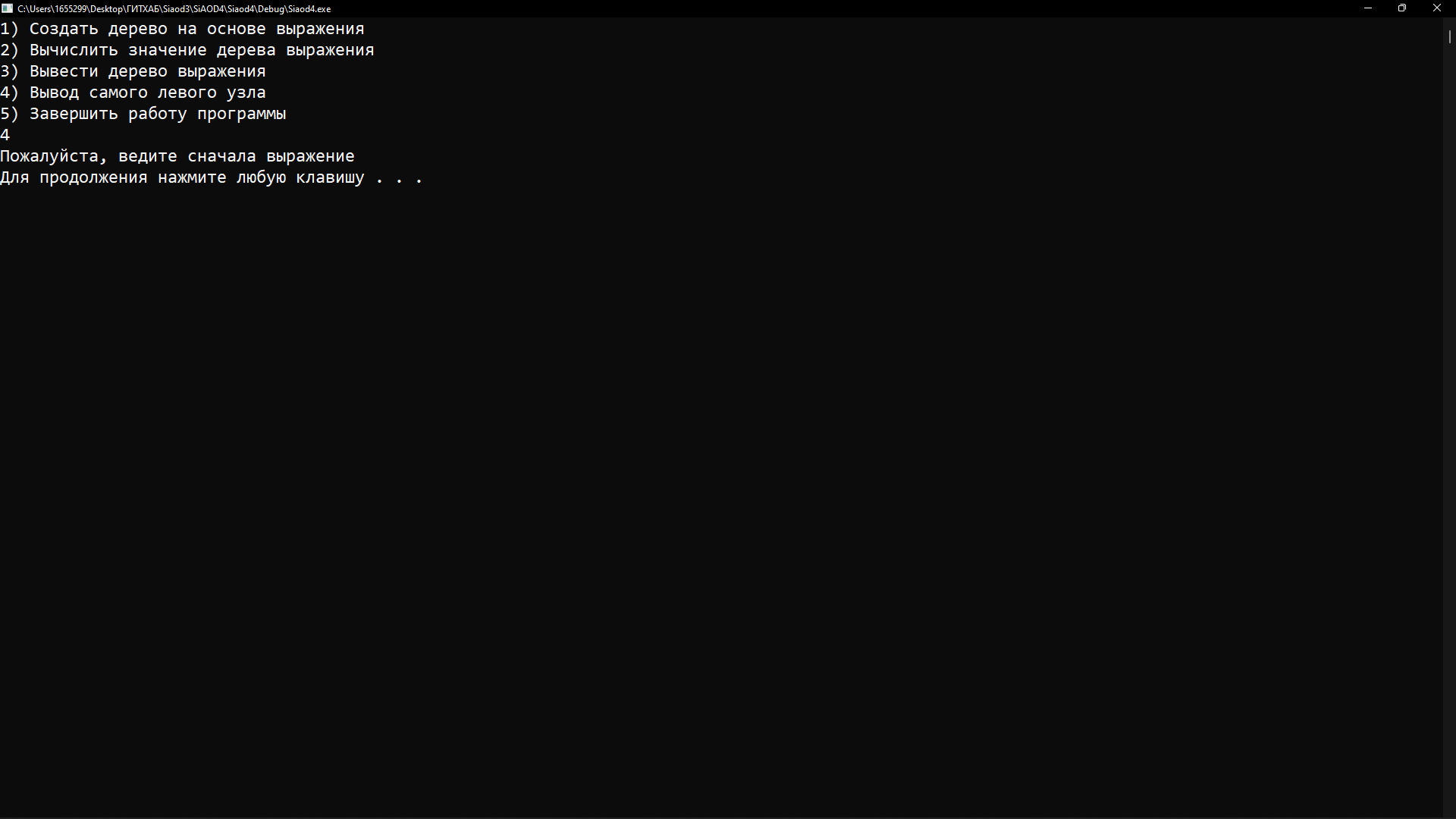


Рис. 8 Ошибка об отсутствии дерева выражений для теста 8

Тест 9.

Проверка функции завершения программы представлена на рисунке 9.

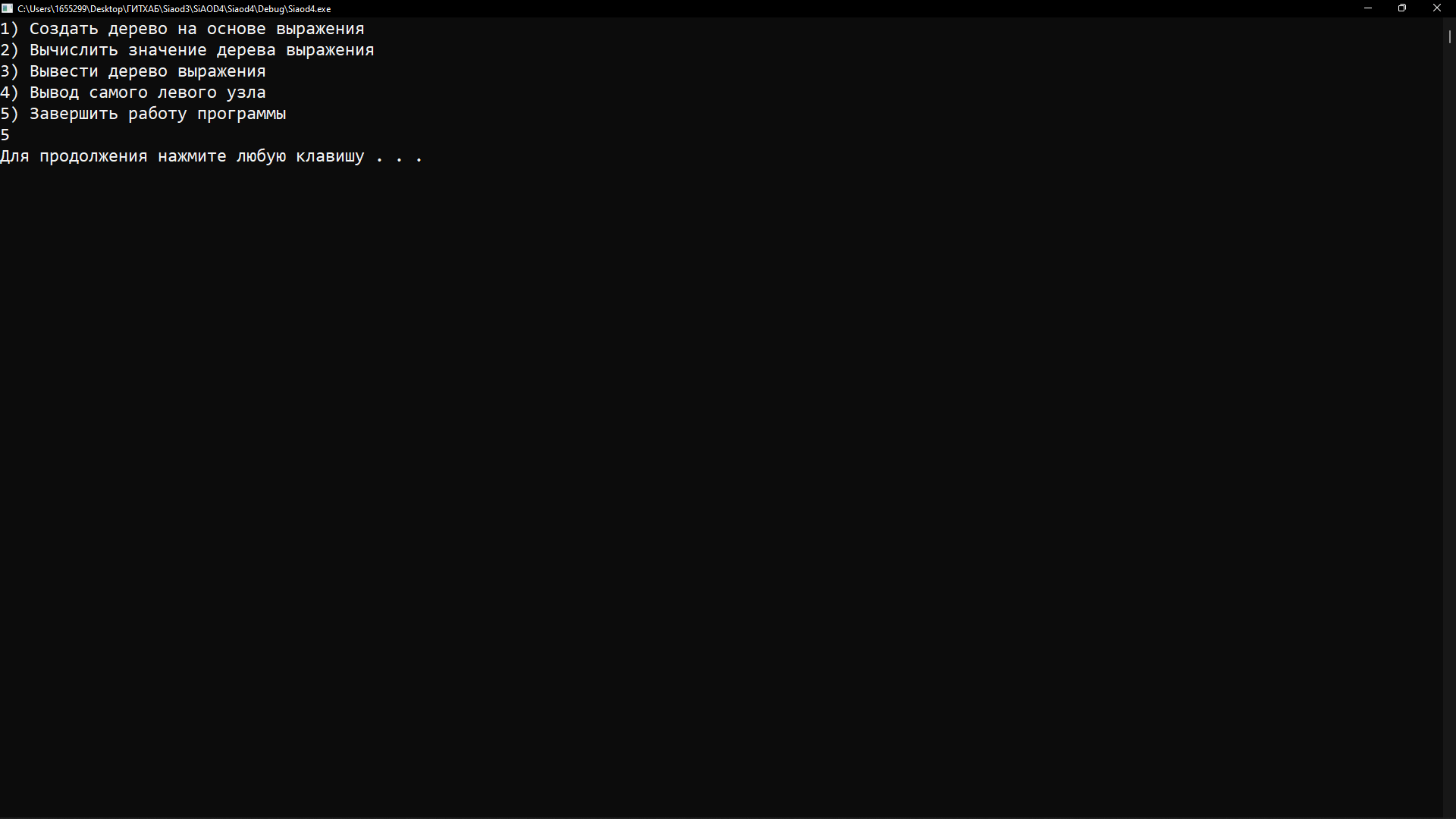


Рис. 9 Проверка функции завершения программы

Таким образом, после проведения тестирования можно подтвердить, что программа работает правильно.

# Вывод

В ходе выполнения практической работы были выполнены все задания, а также:

1. Изучен принцип работы бинарных деревьев;
2. Улучшены знания о рекурсии;
3. Улучшены знания о составлении обратной польской записи

# Исходный код программы

main.cpp

|  |
| --- |
| #include "binary\_tree.h"  void main\_menu()  {  setlocale(0, "");  system("pause");  system("cls");  cout << "1) Создать дерево на основе выражения\n2) Вычислить значение дерева выражения\n3) Вывести дерево выражения\n4) Вывод самого левого узла\n5) Завершить работу программы" << endl;  }  int main()  {  setlocale(0, "");  tree\_elem\* exp\_tree = nullptr;  int choice;  bool is\_not\_executed = true;  while (is\_not\_executed)  {  main\_menu();  cin >> choice;  switch (choice)  {  case 1:  {  string str;  cout << "Введите выражение: " << endl;  cin.get();  getline(cin, str);  str = to\_rpn(str);  exp\_tree = parse\_rpn(str);  cout << "Дерево успешно было создано\n";  }  break;  case 2:  {  if (exp\_tree == nullptr)  {  cout << "Пожалуйста, ведите сначала выражение" << endl;  continue;  }  cout << find\_result(exp\_tree) << endl;  }  break;  case 3:  {  if (exp\_tree == nullptr)  {  cout << "Пожалуйста, ведите сначала выражение" << endl;  continue;  }  print(exp\_tree, 0);  }  break;  case 4:  {  if (exp\_tree == nullptr)  {  cout << "Пожалуйста, ведите сначала выражение" << endl;  continue;  }  cout << find\_left\_elem(exp\_tree) << endl;  }  break;  case 5:  {  is\_not\_executed = false;  }  break;  }  }  system("pause");  } |

binary\_tree.cpp

|  |
| --- |
| #include "binary\_tree.h"  tree\_elem\* create\_tree\_element(char s)  {  tree\_elem\* elem = new tree\_elem;  elem->s = s;  elem->left = nullptr;  elem->right = nullptr;  return elem;  }  void insert\_tree\_elem(tree\_elem\* root, tree\_elem\* elem)  {  if (root->s > elem->s)  {  if (root->left == nullptr)  {  root->left = elem;  }  else  {  insert\_tree\_elem(root->left, elem);  }  }  else if (root->s < elem->s)  {  if (root->right == nullptr)  {  root->right = elem;  }  else  {  insert\_tree\_elem(root->right, elem);  }  }  }  string to\_rpn(string str)  {  map<string, int> priority = { {"(", 1}, {")", 1}, {"+", 2}, {"-", 2}, {"\*", 3}, {"/", 3} };  string word;  stack<string> cur\_values;  string pn = "";  istringstream ss(str);  while (ss >> word)  {  bool is\_keyword = false;  for (auto& tmp : priority)  {  if (tmp.first == word)  is\_keyword = true;  }  if (!is\_keyword)  {  pn += " " + word;  }  else  {  if (word == "(")  cur\_values.push(word);  else if (word == ")")  {  while (!cur\_values.empty() && cur\_values.top() != "(")  {  pn += " " + cur\_values.top();  cur\_values.pop();  }  cur\_values.pop();  }  else  {  while (!cur\_values.empty() && priority[cur\_values.top()] >= priority[word])  {  pn += " " + cur\_values.top();  cur\_values.pop();  }  cur\_values.push(word);  }  }  }  while (!cur\_values.empty())  {  pn += " " + cur\_values.top();  cur\_values.pop();  }  return pn;  }  char find\_left\_elem(tree\_elem\* root)  {  if (root->left == nullptr)  {  return root->s;  }  return find\_left\_elem(root->left);  }  tree\_elem\* parse\_rpn(string rpn\_expression)  {  stack<tree\_elem\*> tree\_stack;  tree\_elem\* te;  for (int i = 0; i < rpn\_expression.size(); i++)  {  if (isdigit(rpn\_expression[i]))  {  te = create\_tree\_element(rpn\_expression[i]);  tree\_stack.push(te);  }  else if (rpn\_expression[i] == '\*' || rpn\_expression[i] == '-' ||rpn\_expression[i] == '+' || rpn\_expression[i] == '/')  {  te = create\_tree\_element(rpn\_expression[i]);  te->right = tree\_stack.top();  tree\_stack.pop();  te->left = tree\_stack.top();  tree\_stack.pop();  tree\_stack.push(te);  }  }  return tree\_stack.top();  }  double find\_result(tree\_elem\* root)  {  if (isdigit(root->s))  {  return double(int(root->s) - 48);  }  if (root->s == '+')  {  return double(find\_result(root->left)) + double(find\_result(root->right));  }  else if (root->s == '-')  {  return double(find\_result(root->left)) - double(find\_result(root->right));  }  else if (root->s == '\*')  {  return double(find\_result(root->left)) \* double(find\_result(root->right));  }  else if (root->s == '/')  {  return double(find\_result(root->left)) / double(find\_result(root->right));  }  return 0.0;  }  void print(tree\_elem\* Tree, int l)  {  int i;  if (Tree != nullptr)  {  print(Tree->right, l + 1);  for (i = 1; i <= l; i++)  {  cout << " ";  }  cout << Tree->s << endl;  print(Tree->left, l + 1);  }  } |

binary\_tree.h

|  |
| --- |
| #ifndef BINARY\_FILES\_H  #define BINARY\_FILES\_H  #include <iostream>  #include <string>  #include <stack>  #include <map>  #include <sstream>  using namespace std;  struct tree\_elem  {  char s;  tree\_elem\* left;  tree\_elem\* right;  };  string to\_rpn(string expression);  tree\_elem\* create\_tree\_element(char s);  void insert\_tree\_elem(tree\_elem\* root, tree\_elem\* elem);  char find\_left\_elem(tree\_elem\* root);  tree\_elem\* parse\_rpn(string rpn\_expression);  double find\_result(tree\_elem\* root);  void print(tree\_elem\* tree, int l);  #endif |