|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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-03-21 | Хречко С.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.

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.

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 №9. Условие задания:

|  |  |
| --- | --- |
| Символьное значение | Проверить, является ли дерево деревом выражений.  Вывести дерево, отобразить его формулу.  Определить, содержит ли дерево операцию \* |

# **Решение**

Хеш-функция — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Двоичное дерево является упорядоченным ориентированным деревом.

По заданию требовалось реализовать бинарное дерево средствами ООП. Каждый узел дерева является объектом, который хранит ссылки на свои дочерние элементы.

|  |
| --- |
| class BinaryTree {  private:      char data;      BinaryTree \*left;      BinaryTree \*right; |

По заданию требовалось создать дерево выражений из выражения, записанного строкой. Метод рекурсивно обрабатывает строку посимвольно, добавляя левый операнд и вызывая метод рекурсивно, чтобы добавить правое поддерево с оператором и/или операндами/ом.

|  |
| --- |
| void addRightNodeRec(const string& s, int it){          if(it >= s.size()){              return;          }          if(it+1 >= s.size()){              right = new BinaryTree(s[it]);              return;          }          right = new BinaryTree( s[it+1], new BinaryTree(s[it]) ); //adding a block of a left operand and an operator          right->addRightNodeRec(s, it+2);          return;      } |

По заданию требовалось определять, является ли выражение алгебраическим. Метод рекурсивно проходит по дереву и проверяет правильность расположения операторов и операндов относительно друг друга, в случае обнаружения нарушения выражение считается не алгебраическим, в случае если противоречий не обнаружено, выражение является алгебраическим.

|  |
| --- |
| bool isExpression(){          if(data == '+' || data == '-' || data == '\*' || data == '/'){ //it's an operator              if(left == nullptr || right == nullptr){                  return false; //if at least one of the operands is missing and the data is an operator then it's not an expression              }              //operator and both operands are present, they must be expressions              return (left->isExpression() && right->isExpression());          }          //not an operator          if(left == nullptr && right == nullptr){              return true; //operand can't have operands so it's an expression          }          return false;      } |

По заданию требовалось определять, присутствует ли операнд \* в выражении. Метод проходит рекурсивно по дереву и проверяет узлы на наличие операнда \*.

|  |
| --- |
| bool containsMultiply(){          if(data == '\*'){              return true;          }          if(left != nullptr && left->containsMultiply()){              return true;          }          if(right != nullptr && right->containsMultiply()){              return true;          }          return false;      } |

По заданию требовалось выводить дерево. Метод рекурсивно проходит по дереву, выводя его элементы, для большего удобства чтения хранится глубина, по которой выводятся дополнительные пробелы, таким образом получается древообразное изображение в консоли.

|  |
| --- |
| void printTree() {          printTree(0);      }      void printTree(int level) {          if (right != nullptr) {              right->printTree(level + 1);          }          for (int i = 0; i < level; i++) {              cout << "|   ";          }          cout << "|" <<data << endl;          if (left != nullptr) {              left->printTree(level + 1);          }      } |

По заданию требовалось отобразить формулу выражения записанного в дереве. Сначала метод рекурсивно вызывается для левого поддерева, затем в строку записывается содержимое узла, для которого вызван метод, затем рекурсивно метод вызывается для правого поддерева.

|  |
| --- |
| string buildExpression(string s = ""){          if(left != nullptr){              s = left->buildExpression(s);          }          s += data;          if(right != nullptr){              s = right->buildExpression(s);          }          return s;      } |

Пользовательский интерфейс программы предлагает пользователю ввести выражение. Затем выводит информацию по этому выражению: построенное дерево, является ли выражение алгебраическим, и затем выводит восстановленное выражение и проверяет присутствует ли в нем символ \*.

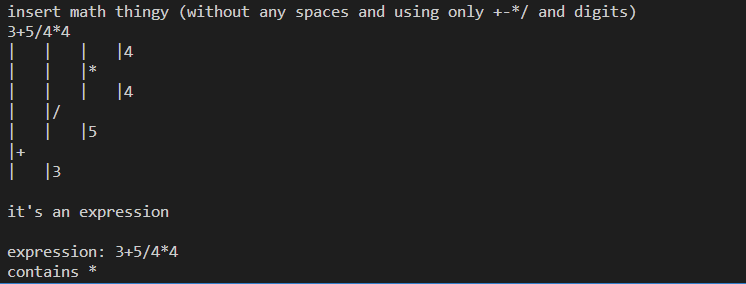


Рисунок 1. Интерфейс программы

# **Тестирование**

Тестирование построения дерева. Как в случае, когда выражение является алгебраическим так и в том случае, когда оно им не является дерево должно быть построено. Тестирование для этих случаев приведено на рисунках 2 и 3.

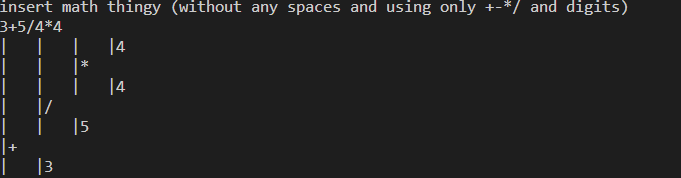


Рисунок 2. Тестирование построения дерева

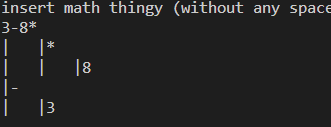


Рисунок 3. Тестирование построения дерева

Тестирование проверки на то, является ли выражение алгебраическим. Проверим для тех же выражений, что и ранее. Результат приведен на рисунках 4 и 5

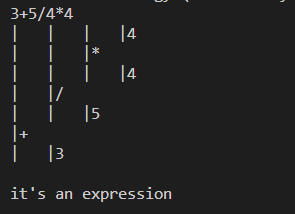


Рисунок 4. Тестирование определения алгебраичности

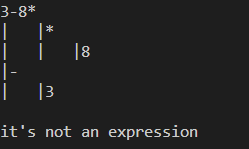


Рисунок 5. Тестирование определения алгебраичности

Тестирование проверки содержания оператора \*. Проверка проводится на случаи, когда такой оператор содержится и не содержится. Также в данную проверку включена проверка восстановления выражения по бинарному дереву. Результаты приведены на рисунках 6 и 7.

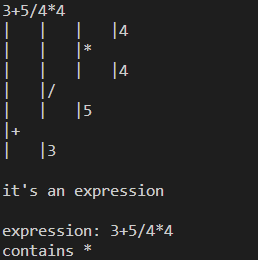


Рисунок 6. Тестирование содержания символа \*

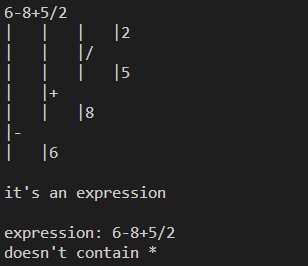


Рисунок 7. Тестирование содержания символа \*

Из результатов выполнения программы видно:

1. Программа работает корректно, все функции работоспособны.

# **Вывод**

В результате выполнения работы я:

1. Освоил базовую работу с бинарными деревьями.
2. Научился программировать создание бинарных деревьев выражений по данному выражению.

# **Исходный код программы**

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  class BinaryTree {  private:      char data;      BinaryTree \*left;      BinaryTree \*right;  public:      BinaryTree(char data) {          this->data = data;          left = nullptr;          right = nullptr;      }      BinaryTree(char data, BinaryTree \*left){          this->data = data;          this->left = left;          right = nullptr;      }      BinaryTree(char data, BinaryTree \*left, BinaryTree \*right) {          this->data = data;          this->left = left;          this->right = right;      }      ~BinaryTree() {          delete left;          delete right;      }      void setLeft(BinaryTree \*left) {          this->left = left;      }      void setRight(BinaryTree \*right) {          this->right = right;      }      BinaryTree \*getLeft() {          return left;      }      BinaryTree \*getRight() {          return right;      }      char getData() {          return data;      }      void addRightNodeRec(const string& s, int it){          if(it >= s.size()){              return;          }          if(it+1 >= s.size()){              right = new BinaryTree(s[it]);              return;          }          right = new BinaryTree( s[it+1], new BinaryTree(s[it]) ); //adding a block of a left operand and an operator          right->addRightNodeRec(s, it+2);          return;      }      bool isExpression(){          if(data == '+' || data == '-' || data == '\*' || data == '/'){ //it's an operator              if(left == nullptr || right == nullptr){                  return false; //if at least one of the operands is missing and the data is an operator then it's not an expression              }              //operator and both operands are present, they must be expressions              return (left->isExpression() && right->isExpression());          }          //not an operator          if(left == nullptr && right == nullptr){              return true; //operand can't have operands so it's an expression          }          return false;      }      bool containsMultiply(){          if(data == '\*'){              return true;          }          if(left != nullptr && left->containsMultiply()){              return true;          }          if(right != nullptr && right->containsMultiply()){              return true;          }          return false;      }      void printTree() {          printTree(0);      }      void printTree(int level) {          if (right != nullptr) {              right->printTree(level + 1);          }          for (int i = 0; i < level; i++) {              cout << "|   ";          }          cout << "|" <<data << endl;          if (left != nullptr) {              left->printTree(level + 1);          }      }      string buildExpression(string s = ""){          if(left != nullptr){              s = left->buildExpression(s);          }          s += data;          if(right != nullptr){              s = right->buildExpression(s);          }          return s;      }  };  int main()  {      setlocale(LC\_ALL, "");      cout << "insert math thingy (without any spaces and using only +-\*/ and digits)" << endl;      string s;      getline(cin, s);      if(s.length() == 0){          cout << "seriosly?" << endl;          return 0;      }      if(s.length() == 1){          BinaryTree \*tree = new BinaryTree(s[0]);          tree->printTree();          delete tree;          return 0;      }      BinaryTree \*tree = new BinaryTree(s[1], new BinaryTree(s[0]));      tree->addRightNodeRec(s, 2);      tree->printTree();      cout << endl;      tree->isExpression() ? cout << "it's an expression" << endl : cout << "it's not an expression" << endl;      cout << endl;      cout << "expression: " << tree->buildExpression() << endl;      tree->containsMultiply() ? cout << "contains \*" << endl : cout << "doesn't contain \*" << endl;      delete tree;      return 0;  } |