|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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-33-22 | Шило Ю.С. |
| Принял преподаватель | Филатов А.С. |

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

Москва 2023

# **Цель работы**

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

1. **Постановка задачи**

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 на экране 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.

|  |  |  |
| --- | --- | --- |
| № | Значение информационной части | Операция варианта |
| 9 | Символьное значение | Проверить, является ли дерево деревом выражений.  Вывести дерево, отобразить его формулу.  Определить, содержит ли дерево операцию \*. |

# **Решение**

Нелинейные структуры данных предоставляют гибкость в организации и хранении данных. Одной из наиболее распространенных нелинейных структур данных является бинарное дерево.

Бинарное дерево состоит из узлов, каждый из которых имеет не более двух потомков: левого и правого. Корень дерева — это верхний узел, который не имеет родителя, а листья — это узлы, не имеющие потомков.

Каждый узел бинарного дерева содержит значение и ссылки на его потомков. Правильное распределение значений между узлами позволяет эффективно выполнять операции поиска, добавления и удаления элементов.

Бинарные деревья широко используются для решения различных задач, включая поиск, сортировку, обход и многое другое. Их гибкость и эффективность делают их важным инструментом в разработке программного обеспечения.

**Задание**

1. Чтобы пользователь мог удобнее взаимодействовать с нашей программы был написан простенький интерфейс. Мы запрашиваем у пользователя строку.

|  |
| --- |
| int main() {  string infixExpression;  cout << "Enter the expression" << std::endl;  getline(cin, infixExpression);  string postfixExpression = convertInfixToPostfix(infixExpression);  Node\* root = buildExpressionTree(postfixExpression);  cout << "Infix -> " << infixExpression << endl  << "Postfix -> " << postfixExpression << endl  << "ExpressionTree -> " << endl;  printExpressionTree(root);   cout << "Get expression from tree -> " << getExpressionFromTree(root) << endl  << "Is Expression tree -> " << (isExpressionTree(root) ? "Yes" : "No") << endl  << "Contains multiplication -> " << (containsMultiplication(root) ? "Yes" : "No") << endl;  return 0; } |

1. Перед выполнением работы нам потребуется создать структуру данных, которая будет описывать узды наши древа. Она включает в себя: указатели на правую и левую ветку, а поле char, которое будет отвечать за хранение информации.

|  |
| --- |
| struct Node {  char data;  Node\* left;  Node\* right;  Node(char data) {  this->data = data;  this->left = nullptr;  this->right = nullptr;  } }; |

1. Для постройки же самого дерева был написана функция buildExpressionTree. Она проверяет какой символ встретился в строке и на основе этого выполняет определенные действия.

|  |
| --- |
| Node\* buildExpressionTree(string postfix) {  stack<Node\*> s;  for (char elem : postfix) {  if (isdigit(elem)) {  Node\* newNode = new Node(elem);  s.push(newNode);  }  else if (!isdigit(elem)) {  Node\* newNode = new Node(elem);  newNode->right = s.top();  s.pop();  newNode->left = s.top();  s.pop();  s.push(newNode);  }  }  return s.top(); } |

1. Для того чтобы пользователю узнать является ли данное древо древом выражений была написана функция isExpressionTree.

|  |
| --- |
| bool isExpressionTree(Node\* root) {  if (root == nullptr) return false;  // Если текущий узел цифра то левая и правая ветка должны быть nullptr || Если текущий узел операция то его левая и правая ветка должны быть деревьями выражений  if (isdigit(root->data))  return root->left == nullptr && root->right == nullptr;  else  return root->left != nullptr && root->right != nullptr && isExpressionTree(root->left) &&  isExpressionTree(root->right); } |

1. Для вывода древа в консоль была написана функция printExpressionTree.

|  |
| --- |
| void printExpressionTree(Node\* root, int level = 0){  if (root == nullptr) return;  printExpressionTree(root->right, level + 1);  for (int i = 0; i < level; i++) std::cout << " ";  std::cout << root->data << std::endl;  printExpressionTree(root->left, level + 1); } |

1. Для того что бы узнать есть ли знак ‘\*’ в текущем дереве была написана функция containsMultiplication.

|  |
| --- |
| bool containsMultiplication(Node\* root) {  if (root == nullptr)  return false;  if (root->data == '\*')  return true;  return containsMultiplication(root->left) || containsMultiplication(root->right); } |

1. Для того что бы получить выражение из нашего дерева была написана функция getExpressionFromTree.

|  |
| --- |
| std::string getExpressionFromTree(Node\* root) {  if (root == nullptr)  return "";  if (isdigit(root->data)) {  string temp = " ";  temp[0] = root -> data;  return temp;  }  std::string leftExpression = getExpressionFromTree(root->left);  std::string rightExpression = getExpressionFromTree(root->right);  return "(" + leftExpression + root->data + rightExpression + ")"; } |

Интерфейс нашей программы представлена на рисунке 1.

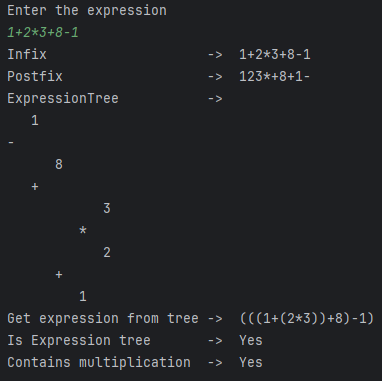


Рисунок 1. Интерфейс программы

# **Тестирование**

Протестируем написанную нами программу. Для этого мы введем строку, которое содержит знак умножения и является выражением. При таком вводе мы получим следующие, вывод в консоли предоставлен на рисунке 2.

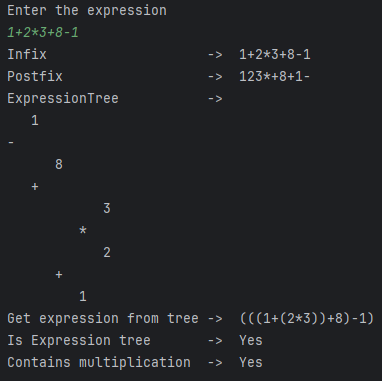


Рисунок 2. Подтверждение работы программы

Теперь введем строку, которая не содержит знака умножения. При таком вводе мы получим следующие, вывод в консоли предоставлен на рисунке 3.

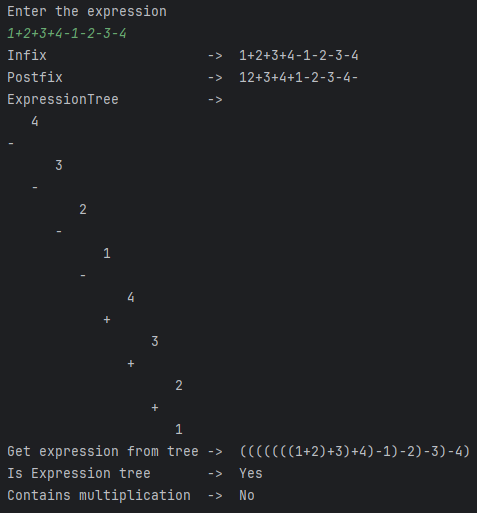


Рисунок 3. Подтверждение работы программы

# **Вывод**

Изучение бинарных деревьев и других нелинейных структур данных помогает разработчикам эффективно организовывать и обрабатывать данные. Эти структуры данных полезны для решения различных задач, таких как поиск, сортировка и обход данных.

# **Исходный код программы**

Исходный код файла для выполнения задания 1 -main.cpp

|  |
| --- |
| #include <iostream> #include <string> #include <stack>  using namespace std;  int getPrecedence(char digit){  if (digit == '+' || digit == '-') return 1;  else return 2; }  string convertInfixToPostfix(string infix){  stack<char> temp;  string res;   for (char elem : infix){  if (isdigit(elem)) res += elem;  else {  while (!temp.empty() && getPrecedence(elem) <= getPrecedence(temp.top())){  res += temp.top();  temp.pop();  }  temp.push(elem);  }  }   while (!temp.empty()){  res+=temp.top();  temp.pop();  }   return res; }  struct Node {  char data;  Node\* left;  Node\* right;   Node(char data) {  this->data = data;  this->left = nullptr;  this->right = nullptr;  } };  //// ================================================================================================================== //// Node\* buildExpressionTree(string postfix) {  stack<Node\*> s;   for (char elem : postfix) {   if (isdigit(elem)) {  Node\* newNode = new Node(elem);  s.push(newNode);  }  else if (!isdigit(elem)) {  Node\* newNode = new Node(elem);  newNode->right = s.top();  s.pop();  newNode->left = s.top();  s.pop();  s.push(newNode);  }  }   return s.top(); }  // Является ли дерево деревом выражений. bool isExpressionTree(Node\* root) {  if (root == nullptr) return false;  // Если текущий узел цифра то левая и правая ветка должны быть nullptr || Если текущий узел операция то его левая и правая ветка должны быть деревьями выражений  if (isdigit(root->data))  return root->left == nullptr && root->right == nullptr;  else  return root->left != nullptr && root->right != nullptr && isExpressionTree(root->left) &&  isExpressionTree(root->right); }  // Вывод дерева, повернутого на экране против часовой стрелки. void printExpressionTree(Node\* root, int level = 0){  if (root == nullptr) return;   printExpressionTree(root->right, level + 1);   for (int i = 0; i < level; i++) std::cout << " ";   std::cout << root->data << std::endl;   printExpressionTree(root->left, level + 1); }  bool containsMultiplication(Node\* root) {  if (root == nullptr) {  return false;  }   if (root->data == '\*') {  return true;  }   return containsMultiplication(root->left) || containsMultiplication(root->right); }  std::string getExpressionFromTree(Node\* root) {  if (root == nullptr) {  return "";  }   if (isdigit(root->data)) {  string temp = " ";  temp[0] = root -> data;  return temp;  }   std::string leftExpression = getExpressionFromTree(root->left);  std::string rightExpression = getExpressionFromTree(root->right);   return "(" + leftExpression + root->data + rightExpression + ")"; }  int main() {  string infixExpression;   cout << "Enter the expression" << std::endl;  getline(cin, infixExpression);   string postfixExpression = convertInfixToPostfix(infixExpression);  Node\* root = buildExpressionTree(postfixExpression);   cout << "Infix -> " << infixExpression << endl  << "Postfix -> " << postfixExpression << endl  << "ExpressionTree -> " << endl;  printExpressionTree(root);   cout << "Get expression from tree -> " << getExpressionFromTree(root) << endl  << "Is Expression tree -> " << (isExpressionTree(root) ? "Yes" : "No") << endl  << "Contains multiplication -> " << (containsMultiplication(root) ? "Yes" : "No") << endl;   return 0; } |