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-11-22 Гришин А. В.

Принял преподаватель Скворцова Л. А.

Самостоятельная работа выполнена «__»_____202__ г. *(подпись студента)*

«Зачтено» «__»_____202__ г. *(подпись руководителя)*

Москва 2023

Содержание

1. Условие задания

Разработать программу вывода дерева выражений в соответствии с вводимым выражением

1.1. Формулировка

Вариант	Значение информационной части	Операции варианта
15	Символьное значение	Вычислить значение выражения в левом поддереве. Вычислить значение выражения в правом поддереве. Вернуть корень дерева и вычислить значение выражения, используя значения левого и правого подвыражений.

1.2. Требования

1) Создать дерево выражений в соответствии с вводимым выражением. Структура узла дерева включает: информационная часть узла – символьного типа: либо знак операции +, -, * либо цифра, указатель на левое и указатель на правое поддерево. В дереве выражения операнды в лист ьях дерева. Исходное выражение имеет формат: ::=цифра|
2) Отобразить дерево на экране, используя алгоритм ввода дерева повернутым справа налево.

2. Описание свойства дерева варианта

Бинарное дерево выражений – это структура данных, предназначенная для представления арифметических выражений в виде дерева. Основные характеристики этой структуры включают следующее:

1) Структура: Бинарное дерево выражений состоит из узлов, где каждый узел может представлять оператор (+, -, *, /) или операнд (например, числовое значение). У каждого оператора есть два дочерних узла, которые также могут быть операторами или операндами.
2) Вычисление: Это дерево используется для представления арифметических выражений, которые можно вычислить, обходя дерево в соответствии с определенными правилами. Например, для вычисления значения выражения можно использовать рекурсивный обход дерева.
3) Приоритет операторов: Дерево учитывает приоритет операторов, размещая операторы с более высоким приоритетом ближе к корню, а операторы с более низким приоритетом - ближе к листьям.
4) Вложенность: Бинарное дерево выражений учитывает вложенность операторов и операндов, что позволяет представлять сложные выражения с правильным порядком вычислений.
5) Обход и преобразование: Дерево может быть обойдено с использованием различных алгоритмов обхода (инфиксный, префиксный, постфиксный) для анализа, вычисления и преобразования выражений.
6) Графическое представление: Бинарное дерево выражений может быть визуально представлено с использованием графических инструментов, что облегчает визуальное понимание структуры выражения.
7) Проверка синтаксиса: Дерево может служить инструментом для проверки синтаксиса арифметических выражений, выявляя ошибки в их структуре.
8) Возможность оптимизации: Дерево может использоваться для оптимизации арифметических выражений, например, для упрощения выражений или удаления избыточных операторов.

3. Примеры дерева со значениями

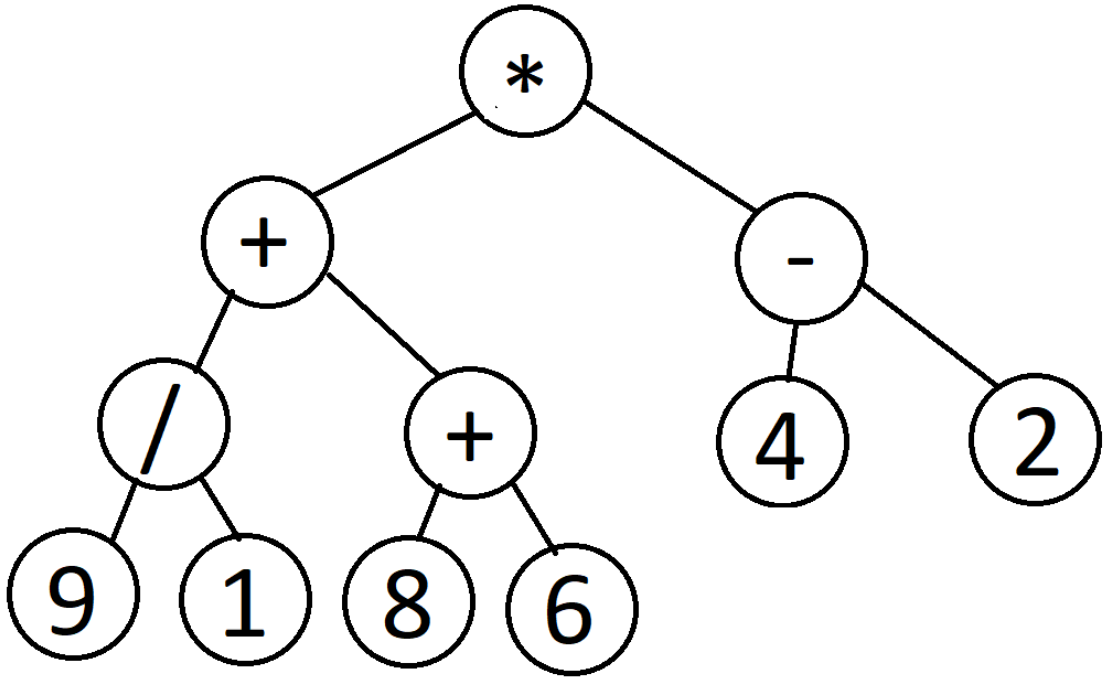

Рис 1. Пример дерева.

4. Результат выполнения операций варианта

```
Enter an expression in prefix form:* + / 9 1 + 8 6 - 4 2

                        2
                - <
                        4
        * <
                                6
                        + <
                                8
                + <
                                1
                        / <
                                9

Value in the left subtree: 23
Value in the right subtree: 2
Root of the tree: *
Value of the expression: 46
```

Рис. 2. Результат выполнения программы.

5. Прототипы функций, реализующих операции задания

1) Конструктор по умолчанию ExprTree::ExprTree():
    - Текст операции (задачи): Конструктор по умолчанию инициализирует объект типа ExprTree пустым деревом.
    - Предусловие: Нет.
    - Постусловие: Объект ExprTree инициализирован, и его корневой узел root установлен в NULL.
    - Прототип: template ExprTree::ExprTree()
    - Описание алгоритма: Просто устанавливает корневой узел root в NULL, создавая пустое дерево.
2) Конструктор копирования ExprTree::ExprTree(const ExprTree& source):
    - Текст операции (задачи): Конструктор копирования создает копию дерева, переданного в аргументе source, и инициализирует текущий объект этой копией.
    - Предусловие: Объект source должен быть корректным деревом.
    - Постусловие: Объект ExprTree инициализирован копией дерева source.
    - Прототип: template ExprTree::ExprTree(const ExprTree& source)
    - Описание алгоритма: Вызывает оператор присваивания operator= для копирования дерева source в текущий объект.
3) Метод для построения дерева void ExprTree::build():
    - Текст операции (задачи): Метод для построения дерева, который позволяет пользователю ввести префиксное арифметическое выражение и создать дерево, представляющее это выражение.
    - Предусловие: Нет.
    - Постусловие: Дерево построено и готово для дальнейших операций.

- Прототип: template void ExprTree::build()
- Описание алгоритма: Метод вызывает вспомогательную функцию buildHelper, которая рекурсивно строит дерево, начиная с корневого узла.

4) *Метод для построения дерева (вспомогательный) void ExprTree::buildHelper(ExprTreeNode& node)**:
    - Текст операции (задачи): Вспомогательный метод для построения дерева, который рекурсивно создает узлы для введенных символов, представляющих операторы и операнды.
    - Предусловие: Нет.
    - Постусловие: Дерево построено, и корень node содержит введенное выражение.
    - Прототип: template void ExprTree::buildHelper(ExprTreeNode*& node) • Описание алгоритма: Метод читает символы из входного потока (например, с клавиатуры) и создает узлы для каждого символа. Если символ - оператор (+, -, *, /), метод рекурсивно вызывает себя для создания левого и правого поддеревьев.
5) Метод для отображения структуры дерева void ExprTree::showStructure() const:
    - Текст операции (задачи): Метод для отображения структуры дерева в виде текстовой диаграммы.
    - Предусловие: Нет.
    - Постусловие: Структура дерева отображена в консоли.
    - Прототип: template void ExprTree::showStructure() const
    - Описание алгоритма: Метод вызывает вспомогательную функцию showHelper, которая рекурсивно обходит дерево и выводит его структуру.
6) Метод для отображения структуры дерева (вспомогательный) void ExprTree::showHelper(ExprTreeNode p, int level) const*:

- Текст операции (задачи): Вспомогательный метод для отображения структуры дерева в виде текстовой диаграммы.
- Предусловие: Дерево должно быть построено.
- Постусловие: Структура дерева отображена в консоли.
- Прототип: template void ExprTree::showHelper(ExprTreeNode* p, int level) const
- Описание алгоритма: Метод рекурсивно обходит дерево и выводит его структуру, включая данные узлов и соединители (стрелки).

7) Метод для вычисления значения выражения дерева void ExprTree<DataType>::evaluate() const:
   - Текст операции (задачи): Метод для вычисления значения выражения, представленного в виде дерева.
   - Предусловие: нет.
   - Постусловие: Значение выражения дерева вычислено и возвращено.
   - Прототип: template<typename DataType> DataType ExprTree<DataType>::evaluate() const
   - Описание алгоритма: Метод вызывает вспомогательный метод evaluateHelper, передавая корень дерева в качестве аргумента, и возвращает результат вычисления значения выражения дерева.
8) Метод для вычисления значения выражения дерева (вспомогательный) DataType ExprTree<DataType>::evaluateHelper(ExprTreeNode* node) const:
   - Текст операции (задачи): Вспомогательный метод для рекурсивного вычисления значения выражения, представленного в виде дерева.
   - Предусловие: Узел node не равен nullptr.

- Постусловие: Значение выражения, представленного поддеревом с корнем в узле node, вычислено и возвращено.
- Прототип: template<typename DataType> DataType ExprTree<DataType>::evaluateHelper(ExprTreeNode* node) const
- Описание алгоритма: Метод рекурсивно вычисляет значение выражения, представленного поддеревом с корнем в узле node, используя рекурсивные вызовы для левого и правого поддеревьев и операцию, заданную в узле node. В случае операнда, значение операнда возвращается. Если оператор неизвестен или происходит деление на ноль, метод выводит сообщение об ошибке и возвращает некоторое значение по умолчанию.

9) Конструктор узла ExprTree::ExprTreeNode::ExprTreeNode(char elem, ExprTreeNode leftPtr, ExprTreeNode rightPtr):
    - Текст операции (задачи): Конструктор создает узел с указанными данными и указателями на левое и правое поддеревья.
    - Предусловие: Нет.
    - Постусловие: Узел создан с заданными данными и поддеревьями.
    - Прототип: template ExprTree::ExprTreeNode::ExprTreeNode(char elem, ExprTreeNode* leftPtr, ExprTreeNode* rightPtr)
    - Описание алгоритма: Конструктор устанавливает данные узла (dataItem) и указатели на левое и правое поддеревья (left и right) в соответствии с переданными параметрами.

6. Код основной программы

Листинг main.cpp

```
#include <iostream>
#include "ExpressionTree.cpp"

using namespace std;

int main() {
    while (true) {
        ExprTree<float> testExprTree;
```

```
        cout << "Enter an expression in prefix form: ";

        testExprTree.build();
        testExprTree.showStructure();

        cout << "Value in the left subtree: " <<
testExprTree.evaluateHelper(testExprTree.getRoot()->left) << endl;
        cout << "Value in the right subtree: " <<
testExprTree.evaluateHelper(testExprTree.getRoot()->right) << endl;
        cout << "Root of the tree: " << testExprTree.getRoot()-
>dataItem << endl;
        cout << "Value of the expression: " <<
testExprTree.evaluate() << endl;
    }
}
```

Листинг ExpressionTree.h

```
#ifndef __4_EXPRESSIONTREE_H
#define __4_EXPRESSIONTREE_H

#include <stdexcept>
#include <iostream>

template <typename DataType>
class ExprTree {
public:
    // Default constructor
    ExprTree();

    // Copy constructor
    ExprTree(const ExprTree& source);

    // Building the tree
    void build();

    // Calculating the value of an expression
    DataType evaluate() const;

    // Displaying the tree structure
    void showStructure() const;

    // Get the root of the tree
    class ExprTreeNode;
    // Expression tree node definition
    class ExprTreeNode {
    public:
        // Node constructor
        ExprTreeNode(char elem, ExprTreeNode* leftPtr, ExprTreeNode*
rightPtr);

        char dataItem; // Expression tree data item
        ExprTreeNode* left; // Pointer to the left child
        ExprTreeNode* right; // Pointer to the right child
    };

    // Calculating the value of an expression (helper function)
    DataType evaluateHelper(ExprTreeNode* node) const;
```

```
    ExprTreeNode* getRoot() const {
        return root;
    }

private:
    // Displaying the tree structure (helper function)
    void showHelper(ExprTreeNode* p, int level) const;

    // Building the tree (helper function)
    void buildHelper(ExprTreeNode*& node);

    // Pointer to the root of the tree
    ExprTreeNode* root;
};

#endif //__4_EXPRESSIONTREE_H
```

Листинг ExpressionTree.cpp

```
#include "ExpressionTree.h"

using namespace std;

template<typename DataType>
ExprTree<DataType>::ExprTree() {
    root = NULL; // Initialize the root pointer to NULL
}

template<typename DataType>
ExprTree<DataType>::ExprTree(const ExprTree& source) {
    *this = source;
}

template<typename DataType>
void ExprTree<DataType>::build() {
    buildHelper(root);
}

template<typename DataType>
void ExprTree<DataType>::buildHelper(ExprTreeNode *&node) {
    char c;
    cin >> c;
    node = new ExprTreeNode(c, NULL, NULL);

    if (c == '+' || c == '-' || c == '*' || c == '/') {
        buildHelper(node->left);
        buildHelper(node->right);
    }
}

template<typename DataType>
void ExprTree<DataType>::showStructure() const {
    if (root == NULL) {
        cout << "Empty tree" << endl;
    } else {
        cout << endl;
        showHelper(root, 1);
        cout << endl;
    }
```

```
}

template<typename DataType>
void ExprTree<DataType>::showHelper(ExprTree::ExprTreeNode *p, int
level) const {
    int j; // Loop counter

    if (p != 0) {
        showHelper(p->right, level + 1); // Output right subtree
        for (j = 0; j < level; j++) // Tab over to level
            cout << "\t";
        cout << " " << p->dataItem; // Output dataItem
        if ((p->left != 0) && // Output "connector"
            (p->right != 0))
            cout << " <";
        else if (p->right != 0)
            cout << "/";
        else if (p->left != 0)
            cout << "\\";
        cout << endl;
        showHelper(p->left, level + 1); // Output left subtree
    }
}

template<typename DataType>
DataType ExprTree<DataType>::evaluate() const {
    return evaluateHelper(root);
}

template<typename DataType>
DataType ExprTree<DataType>::evaluateHelper(ExprTreeNode* node) const
{
    if (node == nullptr) {
        // Empty subtree, return some default value (you may want to
handle this differently)
        return DataType();
    }

    if (isdigit(node->dataItem)) {
        // Operand, return its value
        return static_cast<DataType>(node->dataItem - '0');
    } else {
        // Operator, perform the operation based on the operator
        DataType leftValue = evaluateHelper(node->left);
        DataType rightValue = evaluateHelper(node->right);

        switch (node->dataItem) {
            case '+':
                return leftValue + rightValue;
            case '-':
                return leftValue - rightValue;
            case '*':
                return leftValue * rightValue;
            case '/':
                // Handle division by zero if necessary
                if (rightValue != 0) {
                    return leftValue / rightValue;
                } else {
```

```
                    // Handle division by zero (you may want to
handle this differently)
                    cout << "Error: Division by zero" << endl;
                    return DataType(); // Return some default value
                }
            default:
                // Handle unknown operator (you may want to handle
this differently)
                cout << "Error: Unknown operator" << endl;
                return DataType(); // Return some default value
        }
    }
}

template<typename DataType>
ExprTree<DataType>::ExprTreeNode::ExprTreeNode(char elem,
ExprTree::ExprTreeNode *leftPtr,
                                                ExprTree::ExprTreeNode
*rightPtr) {
    dataItem = elem;
    left = leftPtr;
    right = rightPtr;
}
```