|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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-33-21 | Зарожина Я.А. |
| Принял преподаватель | Муравьева.Е.А. |

|  |  |  |
| --- | --- | --- |
| 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.

1. Разработать программу в соответствии с требованиями варианта. Выполнить реализацию средствами ООП.
2. Вид дерева: дерево выражения.
3. 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.

Условие задания:

|  |  |  |
| --- | --- | --- |
| Вариант | Значение информационной части | Операции варианта |
|  | Символьное значение | Вывод дерева выражений по ширине.  Вернуть самый левый лист дерева.  Вычислить значение выражения |

КОНТРОЛЬНЫЕ ВОПРОСЫ.

1. Что определяет степень дерева?

2. Какова степень сильноветвящегося дерева?

3. Что определяет путь в дереве?

4. Как рассчитать длину пути в дереве?

5. Какова степень бинарного дерева?

6. Может ли дерево быть пустым?

7. Дайте определение бинарного дерева?

8. Дайте определение алгоритму обхода.

9. Приведите рекуррентную зависимость для вычисления высоты дерева.

10. Изобразите бинарное дерево, корень которого имеет индекс 6, и которое представлено в памяти таблицей

11. Укажите путь обхода дерева по алгоритму: прямой; обратный; симметричный

12. Какая структура используется в алгоритме обхода дерева методом в «ширину»?

13. Выведите путь при обходе дерева в «ширину». Продемонстрируйте использование структуры при обходе дерева.

14. Какая структура используется в не рекурсивном обходе дерева методом в «глубину»?

15. Выполните прямой, симметричный, обратный методы обхода дерева выражений.

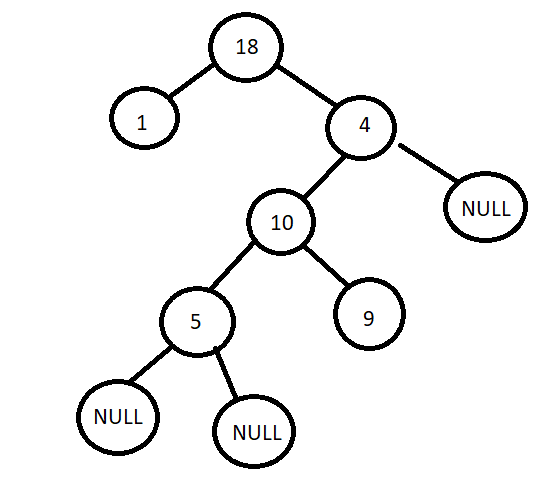
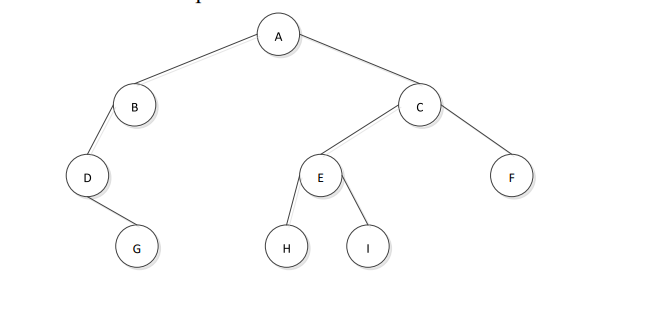
16. Для каждого заданного арифметического выражения постройте бинарное дерево выражений: 1) a+b-c\*d+e 2) /a-b\*c d 3) a b c d / - \* 4) \* - / + a b c d e

17. В каком порядке будет проходиться бинарное дерево, если алгоритм обхода в ширину будет запоминать узлы не в очереди, а в стеке?

18. Постройте бинарное дерево поиска, которое в результате симметричного обхода дало бы следующую последовательность узлов?

19. Приведенная ниже последовательность получена путем прямого обхода бинарного дерева поиска. Постройте это дерево.

# ОТВЕТЫ НА ВОПРОСЫ.

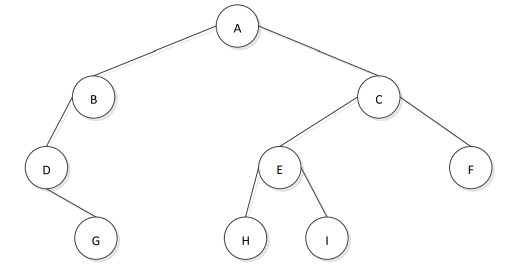
* 1. Степень дерева – это максимальная степень вершин, входящих в дерево. Строгое бинарное дерево – это дерево, у которого вершины имеют степень ноль (у листьев) или два (у узлов). Упорядоченное дерево – это дерево, у которого ветви, исходящие из каждой вершины, упорядочены по определенному критерию. Уровень вершины – это количество дуг от корня дерева до вершины.
  2. Сильноветвящиеся – степень дерева произвольная.
  3. Количество дуг, которые нужно пройти, чтобы продвинуться от корня к вершине X, называется длиной пути к вершине X. Очевидно, что вершина, расположенная на уровне i, имеет длину пути i. Ветвью будем называть путь от корня дерева к любому ее листу. Длина пути дерева определяется как сумма длин путей ко всем его вершинам. Она также называется длиной внутреннего пути дерева.
  4. Длина внутреннего пути может быть определена по следующей рекурсивной формуле: Длина внутреннего пути = Длина внутреннего пути в левом поддереве + длина внутреннего пути в правом поддереве + количество узлов в дереве - 1.
  5. Двоичное (бинарное) дерево – это упорядоченное дерево со степенью вершин не выше 2. У каждой вершины такого дерева есть не более 2 поддеревьев, называемых левым и правым поддеревьями.
  6. Бинарное (двоичное) дерево поиска может быть либо пустым, либо оно обладает таким свойством, что корневой элемент имеет большее значение узла, чем любой элемент в левом поддереве, и меньшее или равное, чем элементы в правом поддереве.
  7. Бинарное дерево (Двоичное дерево) — 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.
  8. обход графа — это переход от одной его вершины к другой в поисках свойств связей этих вершин. Связи (линии, соединяющие вершины) называются направлениями, путями, гранями или ребрами графа. Вершины графа также именуются узлами.  
     Двумя основными алгоритмами обхода графа являются поиск в глубину (Depth-First Search, DFS) и поиск в ширину (Breadth-First Search, BFS).
  9. Если повторение имеет вид T (n) = aT (n / b) + f (n), то глубина дерева равна логарифмической основе b числа n. Например, повторение 2T (n / 2) + n будет иметь дерево глубины lg (n) (логарифмическая база 2 из n).
  10. 
  11. 

Прямой порядок – от корня к левой ветви, затем к правой: A B D G C E H I F

Обратный порядок – проходится левая ветвь, затем правая, затем корень: G D B H I E F C A

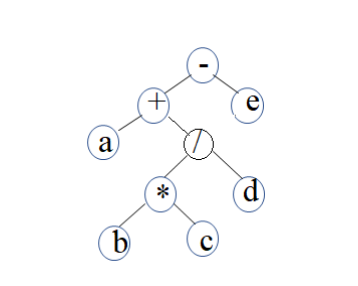
Симметричный порядок – дерево проходится, начиная с левой ветви вверх к корню, затем к правой ветви: D G B A H E I C F

* 1. Обход в ширину подразумевает, что сначала мы посещаем корень, затем, слева направо, все ветви первого уровня, затем все ветви второго уровня и т.д. Пусть мы находимся в корне дерева. Далее необходимо посетить всех наследников корня. Таким образом, нужно засунуть в контейнер сначала узел, затем его наследников, при этом узел далее должен быть обработан первым. То есть, элемент, который вошёл первым должен быть обработан первым.



Путь при обходе дерева в «ширину»: A B C D E F G H I

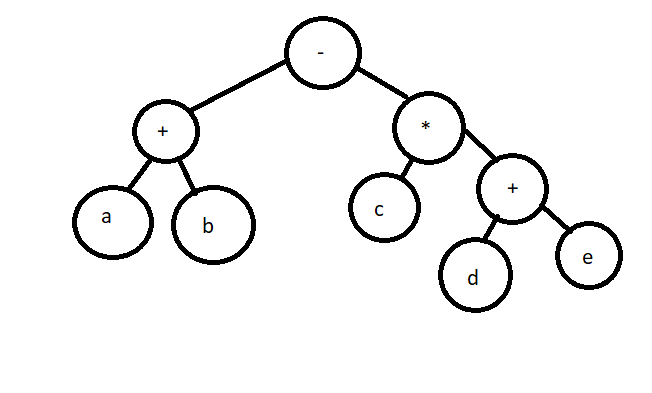
* 1. Поиск по глубине легко реализуется с помощью стека, в том числе рекурсивно (через стек вызовов), в то время как поиск по ширине легко реализуется с помощью очереди, в том числе и рекурсивно.

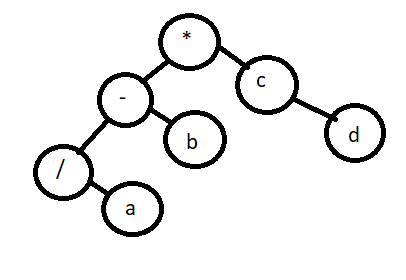


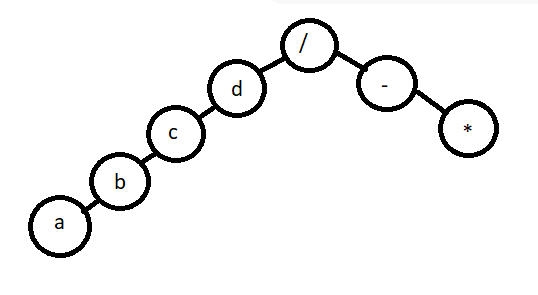
Прямой метод обхода: -+a/\*bcde

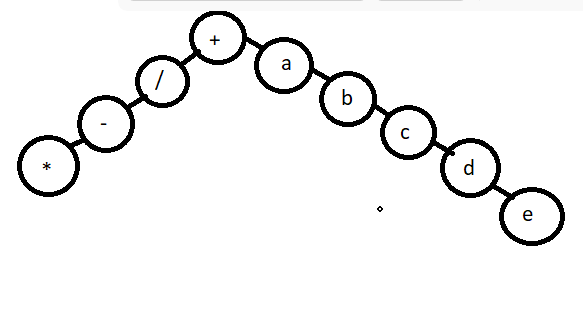
Симметричный метод обхода – a+b\*c/d-e

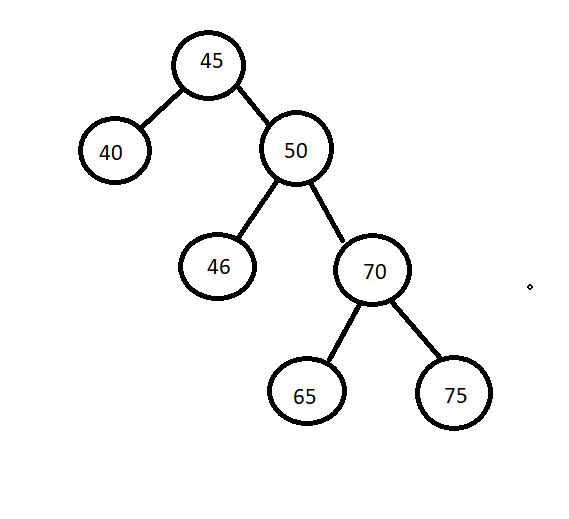
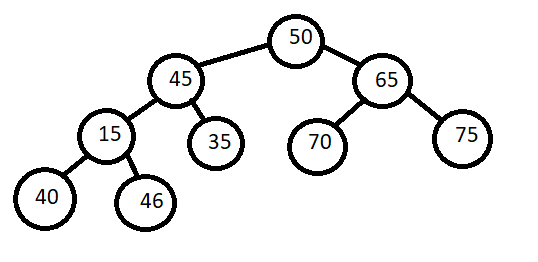
Обратный метод обхода – abc\*d/+e-

 a+b-c\*d+e

/a-b\*c d

 a b c d / - \*

\* - / + a b c d e

* 1. Если заменить очередь стеком при реализации поиска в ширину, вместо этого вы будете выполнять поиск в глубину. Поиск в ширину найдет кратчайший путь в невзвешенном графике, когда он впервые встретит искомый элемент. Если ребра имеют веса, то вам нужно будет либо выполнить исчерпывающий поиск, чтобы найти все пути к элементу, либо использовать вместо этого алгоритм, подобный алгоритму Джикстры. Поиск в глубину не обязательно найдет кратчайший путь, когда он впервые найдет искомый элемент. Поэтому нужно будет выполнить поиск по всем путям.
  2.  40 45 46 50 65 70 75
  3.  50 45 35 15 40 46 65 75 70

4. ОПИСАНИЕ МОДЕЛИ РЕШЕНИЯ.

Был разработана структура дерева выражения. Сама структура:

|  |
| --- |
| struct Tree {  string value;  Tree\* leftLeaf= NULL;  Tree\* rightLeaf= NULL;  };  Tree\* TreeHead = NULL;  Tree\* Child = NULL; |

Метод добавления новой ветви в дереве:

|  |
| --- |
| Tree\* AddLeaf(string value)  {  Tree\* Leaf = new Tree;  Leaf->value = value;  Leaf->leftLeaf = nullptr;  Leaf->rightLeaf = nullptr;  return Leaf;  } |

Далее была разработана функция заполнения дерева. Она получает на вход само выражение и его длину. Затем начинается цикл, который разбирает выражение, начиная с конца. Сначала он записывает операнд в дерево, затем начинается следующий цикл, куда уже поступает на рассмотрение оператор. Далее идет проверка на закрытость ветви, если ветвь не закрыта, то идет проверка на приоритетность операторов текущего и предыдущего (если такой имеется) и затем нужный оператор ставится на нужное место. В следующий цикл поступает уже новый операнд, тот вносится в дерево и закрывает ветвь и эта последовательность начинается сначала. Сама функция записи в дерево:

|  |
| --- |
| void TreeWiriter(string s, int length)  {  string value = "";  Tree\* Leaf = nullptr;  Tree\* oper = nullptr;  Tree\* LastChild = nullptr;  while (length != 0)  {  if ((s[length - 1] != '\*') && (s[length - 1] != '/') && (s[length - 1] != '+') && (s[length - 1] != '-'))  {  value += s[length - 1];  }  else  {  Leaf = AddLeaf(value);  if (Child == NULL)//Если нет незакрытой ветки  {  value = s[length - 1];  oper = AddLeaf(value);  oper->leftLeaf = Leaf;  Child = oper;  }  else//Если ветка не закрыта  {  if ((s[length - 1] == '\*' || s[length - 1] == '/')&&(oper->value == "+" || oper->value == "-"))//если следующий введенный символ выше по исполнимости  {  value = s[length - 1];  oper = AddLeaf(value);  Child->rightLeaf = oper;  oper->leftLeaf = Leaf;  LastChild = oper;  }  else  {  if ((Child->value == "+" || Child->value == "-") && (oper->value == "/" || oper->value == "\*") && (s[length - 1] == '\*' || s[length - 1] == '/'))  {  oper->rightLeaf = Leaf;  LastChild = oper;  value = s[length - 1];  oper = AddLeaf(value);  Child->rightLeaf = oper;  oper->leftLeaf = LastChild;  LastChild = oper;  }  else  {  value = s[length - 1];  oper = AddLeaf(value);  if (LastChild == nullptr)//Если была создана незакрытая ветка с \* или /  {  Child->rightLeaf = Leaf;  }  else  {  LastChild->rightLeaf = Leaf;  LastChild = nullptr;  }  oper->leftLeaf = Child;  Child = oper;  }  }  }  value = "";  }  length--;  }  if (value != "" && LastChild != nullptr)  {  LastChild->rightLeaf = AddLeaf(value);  }  else  {  if (value != "" && Child != nullptr)  Child->rightLeaf = AddLeaf(value);  else  if (value != "")  Child = AddLeaf(value);  }  TreeHead = Child;  } |

С помощью рекурсии на экран пользователя выводится само дерево против часовой стрелки, начиная с самого правого листа и до левого. Функция печатания дерева:

|  |
| --- |
| void PrintTree(Tree\* Leaf, int i, int& maxI)//печатание дерева  {  i = i + 1;  if (Leaf->rightLeaf)  PrintTree(Leaf->rightLeaf, i, maxI);  if (maxI < i)  maxI = i;  for (int j = 0; j < i; j++)  cout << "\t";  cout << Leaf->value<<"\n";  if (Leaf->leftLeaf)  PrintTree(Leaf->leftLeaf, i, maxI);    } |

Функция печатания левого элемента:

|  |
| --- |
| void PrintLeftEl()//печатание левого элемента  {  Tree\* Leaf = TreeHead;  while (Leaf->leftLeaf)  Leaf = Leaf->leftLeaf;  cout << Leaf->value<<"\n";  } |

Функция обхода дерева в ширину создает динамический массив, в который мы записываем значения данных в листьях дерева, а затем вызывает вспомогательную рекурсивную функцию, которая записывает значения каждого листа в ячейку массива. Функция обхода дерева в ширину:

|  |
| --- |
| void WidthTrav(int maxI)  {  Tree\* Leaf = TreeHead;  string \*s = new string[maxI+1];  Weid(TreeHead, s, -1);  for (int i = 0; i < maxI + 1; i++)  cout << s[i];  delete[] s;  } |

Вспомогательная рекурсивная функция вывода для вывода дерева в ширину:

|  |
| --- |
| void Weid(Tree\* Leaf, string\* s, int i)  {  i = i + 1;  if (Leaf->leftLeaf!=NULL)  Weid(Leaf->leftLeaf, s, i);  s[i] = s[i] + Leaf->value;  if (Leaf->rightLeaf!=NULL)  Weid(Leaf->rightLeaf, s, i);  } |

Рекурсивная функция подсчёта выражения:

|  |
| --- |
| int CountingExpression(Tree\* Leaf)//подсчёт выражения  {  int s1 = 0, s2 = 0;  string s;  s = Leaf->value;  if (Leaf->leftLeaf)  s1 = CountingExpression(Leaf->leftLeaf);  if (Leaf->rightLeaf)  s2=CountingExpression(Leaf->rightLeaf);  switch (s[0])  {  case '+':  {  return (s1 + s2);  }  case '-':  {  return (s1 - s2);  }  case '\*':  {  return (s1 \* s2);  }  case '/':  {  return (s1 / s2);  }  default:  return stoi(s);  }  } |

При первом заходе в программу, пользователь видит интерфейс программы, а именно предложение программы ввести выражение (Рис. 1), при вводе выражения программа сразу выполняет все необходимые по варианту действия и выводит их на экран .

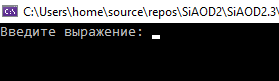


Рис. 1 Интерфейс программы

# **Тестирование**

В первом прогоне используется выражение 1+2+3\*4-5/6. Сначала выводится само дерево против часовой стрелки (Рис. 2).

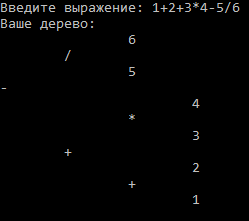


Рис. 2 Заполнение файла первого тестового прогона

Затем выводится самый левый элемент дерева (Рис. 3).



Рис. 3 Вывод самого левого элемента дерева

Потом печатается значение всего выражения (Рис. 4).



Рис. 4 Вывод значения выражения

И в конце дерево выводится по ширине (Рис. 5)



Рис. 5 Вывод дерева по ширине

Во втором прогоне используется выражение 22+14-13\*2\*4/3+25. Сначала выводится само дерево против часовой стрелки (Рис. 6).

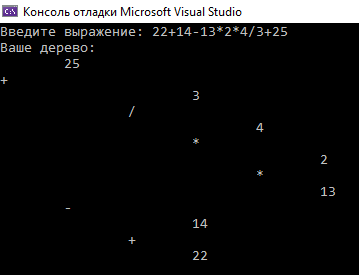


Рис. 6 Заполнение файла первого тестового прогона

Затем выводится самый левый элемент дерева (Рис. 7).



Рис. 7 Вывод самого левого элемента дерева

Потом печатается значение всего выражения (Рис. 8).



Рис. 8 Вывод значения выражения

И в конце дерево выводится по ширине (Рис. 9)



Рис. 9 Вывод дерева по ширине

В третьем прогоне используется выражение 5. Сначала выводится само дерево против часовой стрелки (Рис. 10).



Рис. 10 Заполнение файла первого тестового прогона

Затем выводится самый левый элемент дерева (Рис. 11).



Рис. 11 Вывод самого левого элемента дерева

Потом печатается значение всего выражения (Рис. 12).



Рис. 12 Вывод значения выражения

И в конце дерево выводится по ширине (Рис. 13)



Рис. 13 Вывод дерева по ширине

Из результатов выполнения программы видно:

Тестирование операций над деревом проведено успешно. При вводе любого корректно написанного выражения, программа выводит всё верно.

# ВЫВОД.

В результате выполнения работы я получила навыки разработки бинарного дерева и его применения.

# ИСХОДНЫЙ КОД ПРОГРАММЫ.

|  |
| --- |
| #include <iostream>  #include <string>  using namespace std;  struct Tree {  string value;  Tree\* leftLeaf= NULL;  Tree\* rightLeaf= NULL;  };  Tree\* TreeHead = NULL;  Tree\* Child = NULL;  Tree\* AddLeaf(string value)  {  Tree\* Leaf = new Tree;  Leaf->value = value;  Leaf->leftLeaf = nullptr;  Leaf->rightLeaf = nullptr;  return Leaf;  }  void TreeWiriter(string s, int length)  {  string value = "";  Tree\* Leaf = nullptr;  Tree\* oper = nullptr;  Tree\* LastChild = nullptr;  while (length != 0)  {  if ((s[length - 1] != '\*') && (s[length - 1] != '/') && (s[length - 1] != '+') && (s[length - 1] != '-'))  {  value += s[length - 1];  }  else  {  Leaf = AddLeaf(value);  if (Child == NULL)//Если нет незакрытой ветки  {  value = s[length - 1];  oper = AddLeaf(value);  oper->leftLeaf = Leaf;  Child = oper;  }  else//Если ветка не закрыта  {  if ((s[length - 1] == '\*' || s[length - 1] == '/')&&(oper->value == "+" || oper->value == "-"))//если следующий введенный символ выше по исполнимости  {  value = s[length - 1];  oper = AddLeaf(value);  Child->rightLeaf = oper;  oper->leftLeaf = Leaf;  LastChild = oper;  }  else  {  if ((Child->value == "+" || Child->value == "-") && (oper->value == "/" || oper->value == "\*") && (s[length - 1] == '\*' || s[length - 1] == '/'))  {  oper->rightLeaf = Leaf;  LastChild = oper;  value = s[length - 1];  oper = AddLeaf(value);  Child->rightLeaf = oper;  oper->leftLeaf = LastChild;  LastChild = oper;  }  else  {  value = s[length - 1];  oper = AddLeaf(value);  if (LastChild == nullptr)//Если была создана незакрытая ветка с \* или /  {  Child->rightLeaf = Leaf;  }  else  {  LastChild->rightLeaf = Leaf;  LastChild = nullptr;  }  oper->leftLeaf = Child;  Child = oper;  }  }  }  value = "";  }  length--;  }  if (value != "" && LastChild != nullptr)  {  LastChild->rightLeaf = AddLeaf(value);  }  else  {  if (value != "" && Child != nullptr)  Child->rightLeaf = AddLeaf(value);  else  if (value != "")  Child = AddLeaf(value);  }  TreeHead = Child;  }  void PrintTree(Tree\* Leaf, int i, int& maxI)//печатание дерева  {  i = i + 1;  if (Leaf->rightLeaf)  PrintTree(Leaf->rightLeaf, i, maxI);  if (maxI < i)  maxI = i;  for (int j = 0; j < i; j++)  cout << "\t";  cout << Leaf->value<<"\n";  if (Leaf->leftLeaf)  PrintTree(Leaf->leftLeaf, i, maxI);    }  void PrintLeftEl()//печатание левого элемента  {  Tree\* Leaf = TreeHead;  while (Leaf->leftLeaf)  Leaf = Leaf->leftLeaf;  cout << Leaf->value<<"\n";  }  int CountingExpression(Tree\* Leaf)//подсчёт выражения  {  int s1 = 0, s2 = 0;  string s;  s = Leaf->value;  if (Leaf->leftLeaf)  s1 = CountingExpression(Leaf->leftLeaf);  if (Leaf->rightLeaf)  s2=CountingExpression(Leaf->rightLeaf);  switch (s[0])  {  case '+':  {  return (s1 + s2);  }  case '-':  {  return (s1 - s2);  }  case '\*':  {  return (s1 \* s2);  }  case '/':  {  return (s1 / s2);  }  default:  return stoi(s);  }  }  void Weid(Tree\* Leaf, string\* s, int i)  {  i = i + 1;  if (Leaf->leftLeaf!=NULL)  Weid(Leaf->leftLeaf, s, i);  s[i] = s[i] + Leaf->value;  if (Leaf->rightLeaf!=NULL)  Weid(Leaf->rightLeaf, s, i);  }  void WidthTrav(int maxI)  {  Tree\* Leaf = TreeHead;  string \*s = new string[maxI+1];  Weid(TreeHead, s, -1);  for (int i = 0; i < maxI + 1; i++)  cout << s[i];  delete[] s;  }  void Deleter(Tree\* Leaf)  {  if (Leaf->rightLeaf)  Deleter(Leaf->rightLeaf);  if (Leaf->leftLeaf)  Deleter(Leaf->leftLeaf);  delete (Leaf);  }  int main()  {  setlocale(LC\_ALL, "RUSSIAN");  string s;  int maxI;  cout << "Введите выражение: ";  cin >> s;  for (int i = 0; i < s.length() / 2; i++)  {  char buff;  buff = s[i];  s[i] = s[s.length() - 1-i];  s[s.length() - 1 - i] = buff;  }  TreeWiriter(s, s.length());  cout << "Ваше дерево:" << "\n";  PrintTree(TreeHead,-1, maxI);  cout << "\n" << "Самый левый элемент: ";  PrintLeftEl();  cout << "\n" << "Значение выражения: " << CountingExpression(TreeHead);  cout << "\n\nОбход в ширину: ";  WidthTrav(maxI);  Deleter(TreeHead);  } |