|  |  |  |
| --- | --- | --- |
|  |  |  |
| 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»

по теме «Нелинейные структуры данных. Бинарное дерево.»

|  |  |
| --- | --- |
| **Выполнил:**  Студент группыИКБО-13-22 | Козлов К.И. |
| **Проверил:** | ассистент Муравьёва Е.А. |

МОСКВА 2023 г.

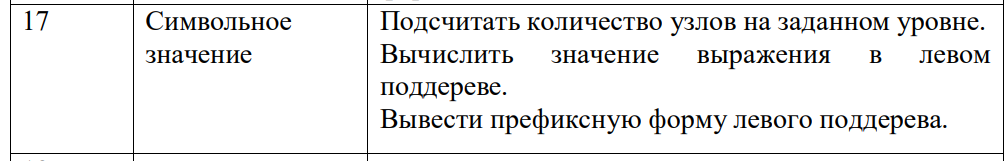
1. **ВВЕДЕНИЕ**

**Цель**: получение умений и навыков разработки и реализаций операций над структурой данных бинарное дерево.

* 1. **Задания (Вариант 17)**

Значение информационной части: **символьное значение**

Операции варианта:

* 
  + 1. **Формулировка задания 1**

Ответьте на вопросы и выполните упражнения.

* + 1. **Формулировка задания 2**

Разработать программу в соответствии с требованиями варианта.

*Совет*. Выполните реализацию средствами ООП, операции будут

методами класса.

Вид дерева: дерево выражения.

1. Реализовать операции общие для вариантов с 11 по 20:

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 листьях дерева.

Исходное выражение имеет формат:

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

1. **ХОД РАБОТЫ**
   1. **Ответы на вопросы**
      1. **Что определяет степень дерева?**

Степень дерева определяется количеством потомков, или детей, у каждой вершины (узла) в дереве. Вершины дерева называются узлами, и степень узла определяет, сколько непосредственных потомков у этого узла.

Узел степени 0 (листовой узел): Узел, у которого нет потомков, называется листовым узлом. Листовые узлы находятся в конце ветвей дерева и не имеют дополнительных поддеревьев.

Узел степени 1: Узел, у которого есть только один непосредственный потомок, называется узлом степени 1. Такие узлы соединяют другие узлы в древовидной структуре.

Узел степени 2 и более: Узел, у которого есть два или более непосредственных потомка, называется узлом степени 2 или более.

Степень дерева — это максимальное количество потомков (детей), которое имеет узел в дереве. Она может варьироваться в зависимости от конкретной структуры дерева.

* + 1. **Какова степень сильноветвящегося дерева?**

Сильноветвящееся дерево — это дерево, у которого вершины имеют большое количество потомков (детей). Степень сильноветвящегося дерева определяется как максимальное количество потомков, которое имеет вершина в этом дереве. Степень сильноветвящегося дерева может быть очень высокой, и она зависит от конкретной структуры дерева.

В примере сильноветвящегося дерева, степень каждой вершины может быть значительно больше, чем в типичных деревьях. Например, в бинарном дереве каждая вершина имеет не более двух потомков (степень 2), в то время как в сильноветвящемся дереве вершины могут иметь множество потомков. Степень сильноветвящегося дерева зависит от его конкретной структуры и потребностей задачи, для которой оно используется.

* + 1. **Что определяет путь в дереве?**

Путь в дереве определяется последовательностью вершин, которые соединяются друг с другом от корня до определенной целевой вершины внутри дерева. Путь в дереве представляет собой цепь или последовательность вершин и рёбер, которые соединяются между собой.

Основные характеристики пути в дереве:

* Начальная вершина (или корень): Путь начинается с какой-то начальной вершины в дереве, которая может быть корневой вершиной или любой другой.
* Целевая вершина: Путь заканчивается в целевой вершине, которая может быть любой внутри дерева.
* Вершины и рёбра на пути: Путь включает в себя все вершины, которые проходят от начальной вершины к целевой вершине, а также все рёбра, которые соединяют эти вершины.
* Длина пути: Длина пути определяется количеством вершин или рёбер на этом пути. Например, длина пути может быть равна количеству вершин минус один, или по-другому количеству рёбер.

Путь в дереве может быть использован для определения связей и отношений между вершинами внутри дерева. Также путь может использоваться для нахождения определенной информации или выполнения навигации в дереве, в том числе поиск пути между вершинами или вычисление расстояния между ними.

* + 1. **Как рассчитать длину пути в дереве?**
* По количеству вершин: Длина пути в дереве может быть определена как количество вершин (узлов), которые включены в этот путь. Если есть последовательность вершин, соединенных рёбрами, можно просто подсчитать количество вершин на этом пути. Длина пути будет равна количеству вершин минус один, так как количество рёбер на пути всегда на один меньше количества вершин.
* По количеству рёбер: Длина пути также может быть рассчитана через количество рёбер на пути. Это делается путем подсчета всех рёбер, которые соединяют вершины на пути. Количество рёбер равно длине пути.
* Весовой путь (взвешенное дерево): если дерево имеет весовые рёбра (рёбра имеют числовые значения, называемые весами), то длина пути может быть определена как сумма весов рёбер на пути. В этом случае длина пути будет равна сумме весов всех рёбер, которые соединяют вершины на пути.

Выбор метода зависит от конкретного контекста и структуры дерева. Важно учесть, что в зависимости от задачи можно использовать один из этих методов для вычисления длины пути в дереве.

* + 1. **Какова степень бинарного дерева?**

Степень бинарного дерева определяется максимальным числом потомков, которое может иметь вершина (узел) в этом дереве. В бинарном дереве каждая вершина может иметь не более двух потомков: левого и правого.

Следовательно, степень бинарного дерева равна 2, так как каждая вершина может иметь максимум два потомка. Это особенность бинарных деревьев и делает их структуру очень удобной для множества задач, включая поиск, сортировку и балансировку данных.

* + 1. **Может ли дерево быть пустым?**

Да, дерево может быть пустым. Пустое дерево не содержит ни одной вершины (узла) и, следовательно, не имеет никакой структуры. В программировании и структурах данных пустые деревья часто используются как начальное состояние перед добавлением данных или как специальное состояние для представления отсутствия информации или структуры.

Пустое дерево полезно, когда вы хотите начать построение дерева или когда ваши данные могут отсутствовать. Когда дерево пусто, оно не содержит ни одной вершины, и его структура не определена. Как только вы начинаете добавлять вершины в дерево, оно начинает приобретать форму и структуру.

Пустые деревья широко используются в алгоритмах и структурах данных, таких как бинарные деревья поиска, кучи (heap), деревья AVL и других. Они представляют собой важный элемент для обеспечения корректной работы алгоритмов и обработки различных случаев.

* + 1. **Дайте определение бинарного дерева.**

Бинарное дерево — это иерархическая структура данных, в которой каждая вершина (узел) имеет не более двух потомков: левого и правого. Бинарные деревья широко используются в информатике для различных задач, таких как поиск, сортировка, хранение данных и многие другие.

Основные характеристики бинарного дерева:

* Корневая вершина (Root): Это вершина, которая находится в самом верхнем уровне бинарного дерева. Все остальные вершины происходят от корневой вершины.
* Левый и правый потомки: Каждая вершина может иметь не более двух потомков: левого и правого. Левый потомок находится слева от родительской вершины, а правый - справа.
* Листовые вершины (Leaves): Листовые вершины — это вершины без потомков, то есть вершины, у которых не существует левого и правого поддерева. Они находятся в конце ветвей дерева.
* Поддеревья: Каждый узел бинарного дерева может быть корнем для своего собственного поддерева, включая все вершины, которые являются его потомками.

Бинарные деревья применяются для решения различных задач, таких как бинарные деревья поиска (BST), кучи (heap), деревья AVL и др. Каждый из этих видов бинарных деревьев имеет свои собственные правила и свойства, которые обеспечивают эффективную работу алгоритмов и структур данных.

* + 1. **Дайте определение алгоритму обхода.**

Алгоритм обхода (или алгоритм прохода) — это специфическая процедура, которая определяет порядок посещения элементов (например, узлов или вершин) в структуре данных, такой как дерево, граф, массив и другие. Обход используется для систематического и последовательного доступа ко всем элементам структуры данных.

Алгоритмы обхода могут быть применены к различным задачам, таким как поиск, вывод данных, анализ и обработка информации внутри структуры. Обходы бывают двух видов: в ширину и в глубину.

Поиск в ширину (BFS) идет из начальной вершины, посещает сначала все вершины, находящиеся на расстоянии одного ребра от начальной, потом посещает все вершины на расстоянии два ребра от начальной и так далее. Алгоритм поиска в ширину является по своей природе нерекурсивным (итеративным). Для его реализации применяется структура данных очередь (FIFO).

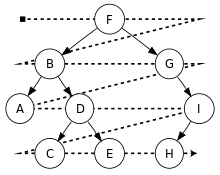


Рисунок 1 – пример обхода бинарного дерева в ширину

Поиск в глубину (DFS) идет из начальной вершины, посещая еще не посещенные вершины без оглядки на удаленность от начальной вершины. Алгоритм поиска в глубину по своей природе является рекурсивным. Для эмуляции рекурсии в итеративном варианте алгоритма применяется структура данных стек. Обходу в глубину в графе соответствуют три вида обходов бинарного дерева: прямой (pre-order), симметричный (in-order) и обратный (post-order).

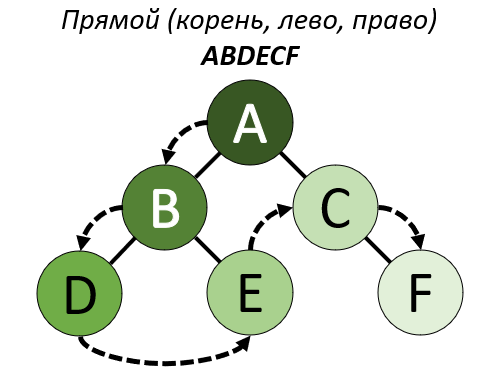


Рисунок 2 – Прямой обход дерева (корень, лево, право)

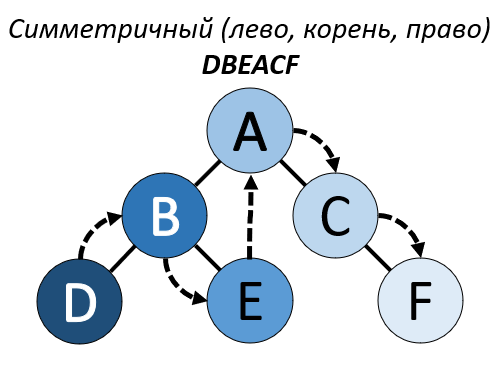


Рисунок 3 – Симметричный обход дерева (лево, корень, право)

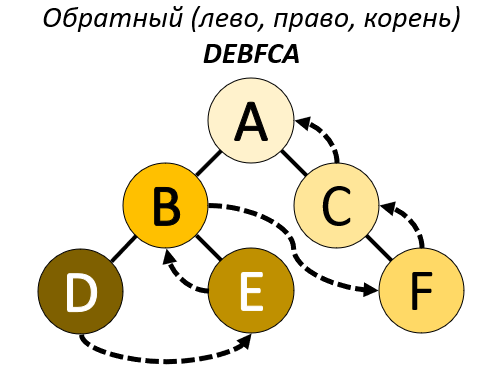


Рисунок 4 – Обратный обход дерева (лево, право, корень)

Каждый тип алгоритма обхода имеет свои уникальные применения и может использоваться в разных сценариях в зависимости от конкретной задачи и структуры данных.

* + 1. **Приведите рекуррентную зависимость для вычисления высоты дерева.**

Для вычисления высоты бинарного дерева можно использовать рекуррентную зависимость. Высота дерева определяется как максимальная длина пути от корневой вершины до самой удаленной листовой вершины. Рекуррентная зависимость для вычисления высоты дерева может быть описана следующим образом:

* Пусть height(root) обозначает высоту дерева с корнем в вершине root.
* Если root равен null (то есть дерево пусто), то height(root) равна -1 (или 0, в зависимости от условия).
* Иначе, height(root) можно выразить как:

Формула гласит, что высота дерева с корнем в вершине root равна максимальной из высот левого поддерева height(root.left) и высоты правого поддерева height(root.right), увеличенной на 1. Это означает, что для вычисления высоты дерева, мы выбираем более высокое из двух поддеревьев (левого и правого) и добавляем 1, чтобы учесть текущий уровень, на котором находится корневая вершина.

Эта рекуррентная зависимость рекурсивно вычисляет высоту дерева, начиная с корневой вершины и продолжая до листовых вершин. Результатом будет высота всего дерева.

* + 1. **Изобразите бинарное дерево, корень которого имеет индекс 6, и которое представлено в памяти таблицей вида**

1. Таблица 1 – Задание 10

|  |  |  |  |
| --- | --- | --- | --- |
| Индекс | key | left | right |
| 1 | 12 | 7 | 3 |
| 2 | 15 | 8 | NULL |
| 3 | 4 | 10 | NULL |
| 4 | 10 | 5 | 9 |
| 5 | 2 | NULL | NULL |
| 6 | 18 | 1 | 4 |
| 7 | 7 | NULL | NULL |
| 8 | 14 | 6 | 2 |
| 9 | 21 | NULL | NULL |
| 10 | 5 | NULL | NULL |

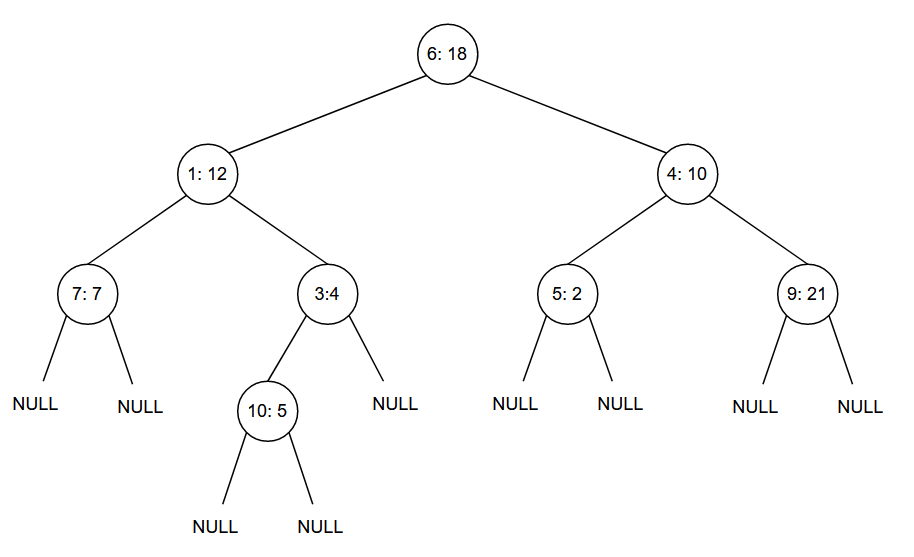


Рисунок 5 – бинарное дерево, представленное таблицей

* + 1. **Укажите путь обхода дерева по алгоритмам: прямой, обратный, симметричный**

Представление трех видов обхода дерева в глубину:

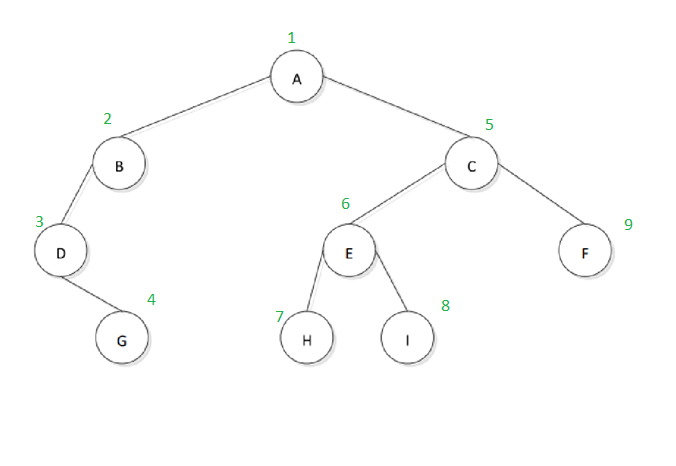
****

Рисунок 6 – Прямой обход дерева

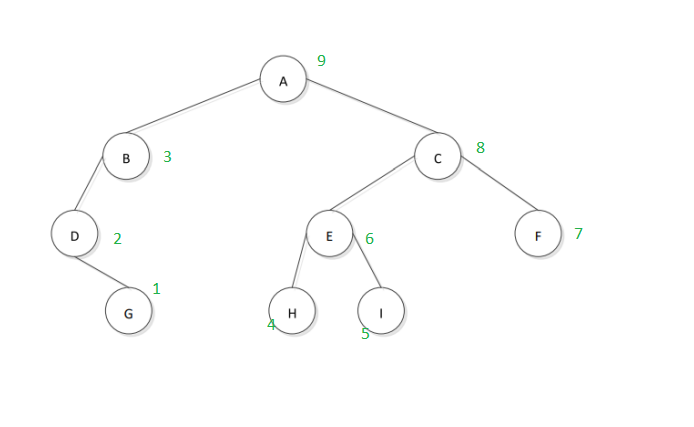
****

Рисунок 7 – Обратный обход дерева

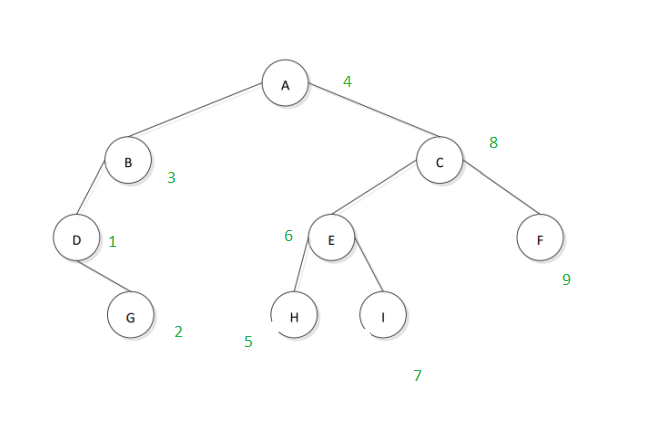
****

Рисунок 8 – Симметричный обход дерева

* + 1. **Какая структура используется в алгоритме обхода дерева методом в «ширину»**

В алгоритме обхода дерева методом в "ширину" (Breadth-First Search, BFS) для хранения вершин, которые ожидают обработки, обычно используется структура данных, называемая очередью (queue). Очередь работает по принципу "первым пришел - первым ушел" (FIFO - First-In-First-Out). Это означает, что вершины добавляются в очередь в порядке их появления, и они извлекаются из очереди в том же порядке.

В контексте обхода дерева методом BFS, процесс может быть описан следующим образом:

* Начинаем с корневой вершины дерева и помещаем ее в очередь.
* Затем выполняем следующие шаги в цикле, пока очередь не опустеет:
  + Извлекаем вершину из начала очереди.
  + Обрабатываем эту вершину (например, выводим ее значение или выполняем другие операции).
  + Добавляем в очередь всех непосещенных потомков этой вершины.
* Повторяем шаги 2 до тех пор, пока в очереди не останется вершин для обработки.

Использование очереди в алгоритме BFS позволяет обеспечить обход вершин на текущем уровне перед переходом к вершинам следующего уровня. Это обеспечивает обход в "ширину", где сначала обрабатываются вершины на текущем уровне, затем на следующем уровне и так далее, пока не пройдены все уровни дерева.

* + 1. **Выведите путь при обходе дерева в «ширину». Продемонстрируйте использование структуры при обходе дерева.**

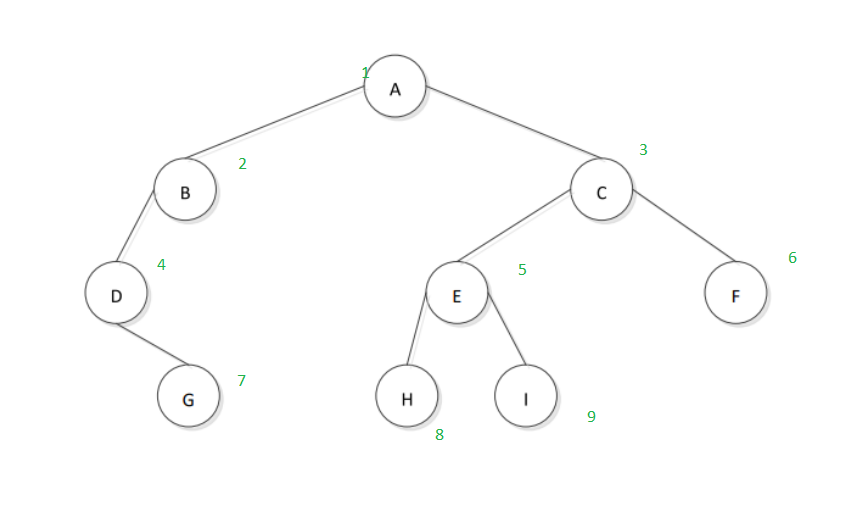
****

Рисунок 9 – обход дерева в ширину

Структура обхода дерева в ширину заключается в добавлении в очередь каждого потомка искомого узла слева направо. То есть сначала в очереди у нас уже есть корневой узел, потом при обходе значений мы как-бы достаём его из очереди, производим с ним какие-либо операции (например, считываем) и засовываем в очередь его левого и правого потомка. Потом циклично повторяем тоже самое, доставая из очереди левых и правых потомков узла и уже помещая в очередь их потомков и т.д.

* + 1. **Какая структура используется в не рекурсивном обходе дерева методом в «глубину»?**

В нерекурсивном обходе дерева методом в "глубину" (Depth-First Search, DFS), часто используется структура данных, называемая ***стеком*** (stack), для выполнения обхода вершин. Стек работает по принципу "последним пришел - первым ушел" (LIFO - Last-In-First-Out), что означает, что вершины добавляются и извлекаются из стека в обратном порядке.

Процесс не-рекурсивного обхода дерева методом в "глубину" с использованием стека может быть описан следующим образом:

* Начинаем с корневой вершины дерева и помещаем ее в стек.
* Затем выполняем следующие шаги в цикле, пока стек не опустеет:
  + Извлекаем вершину из стека.
  + Обрабатываем эту вершину (например, выводим ее значение или выполняем другие операции).
  + Помещаем в стек всех непосещенных потомков этой вершины. При этом, вершины добавляются в стек в обратном порядке, так чтобы вершина, которая должна быть обработана следующей, оказывается на вершине стека.
* Повторяем шаги 2 до тех пор, пока в стеке не останется вершин для обработки.

Использование стека в алгоритме DFS позволяет реализовать обход вершин на текущем уровне перед переходом к вершинам более глубокого уровня. Это обеспечивает обход в "глубину", где сначала обрабатываются вершины на текущем пути вниз по дереву, затем вершины на более глубоких уровнях.

* + 1. **Выполните прямой, симметричный, обратный методы обхода дерева выражений**

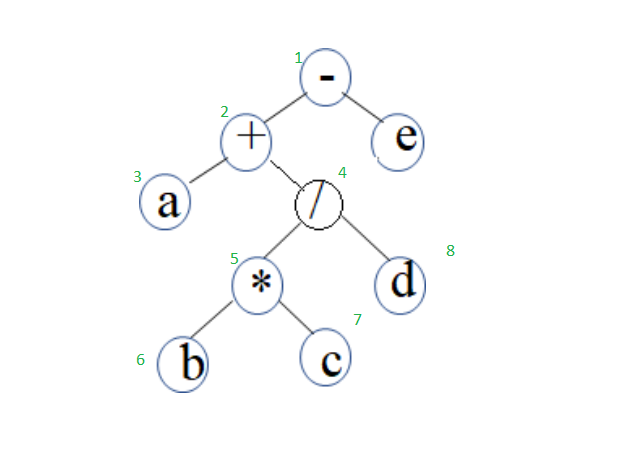
****

Рисунок 10 – Прямой обход дерева выражений (**-+a/bc\*de**)

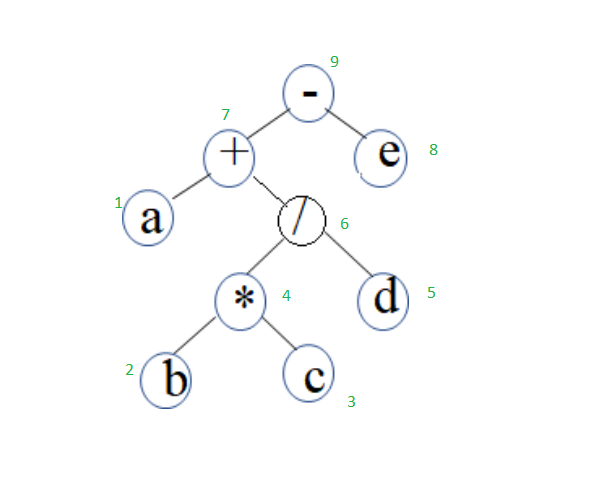
****

Рисунок 11 – Обратный обход дерева выражений

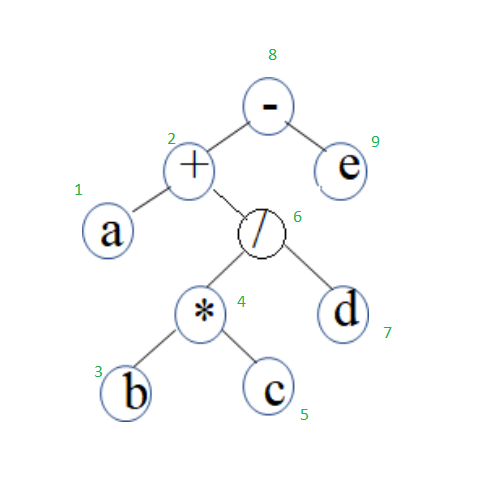
****

Рисунок 12 – Симметричный обход дерева выражений

* + 1. **Для каждого заданного арифметического выражения постройте бинарное дерево выражений**

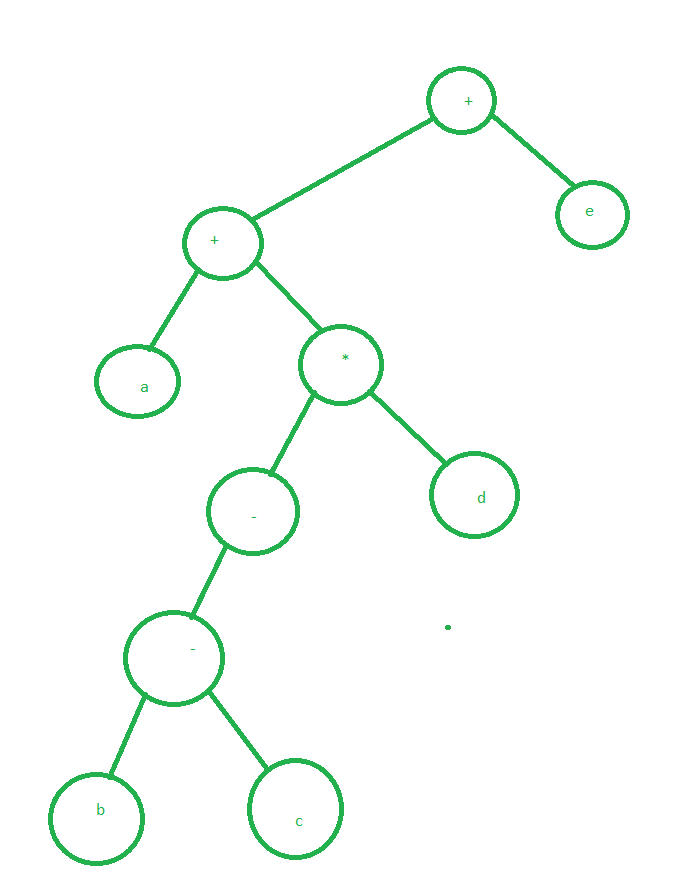
****

Рисунок 13 – Инфиксное выражение (a+b-c\*d+e)

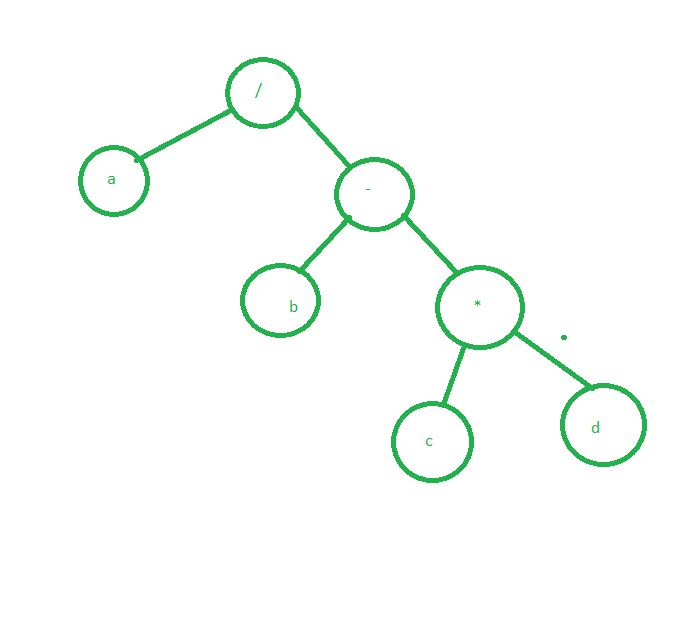
****

Рисунок 14 – Префиксное выражение (**/a-b\*cd**)

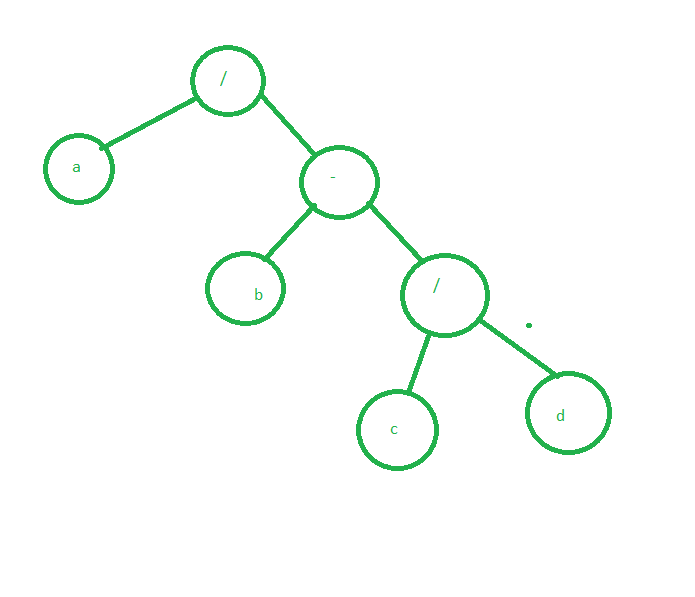
****

Рисунок 15 – Постфиксное выражение (abcd/-\*)

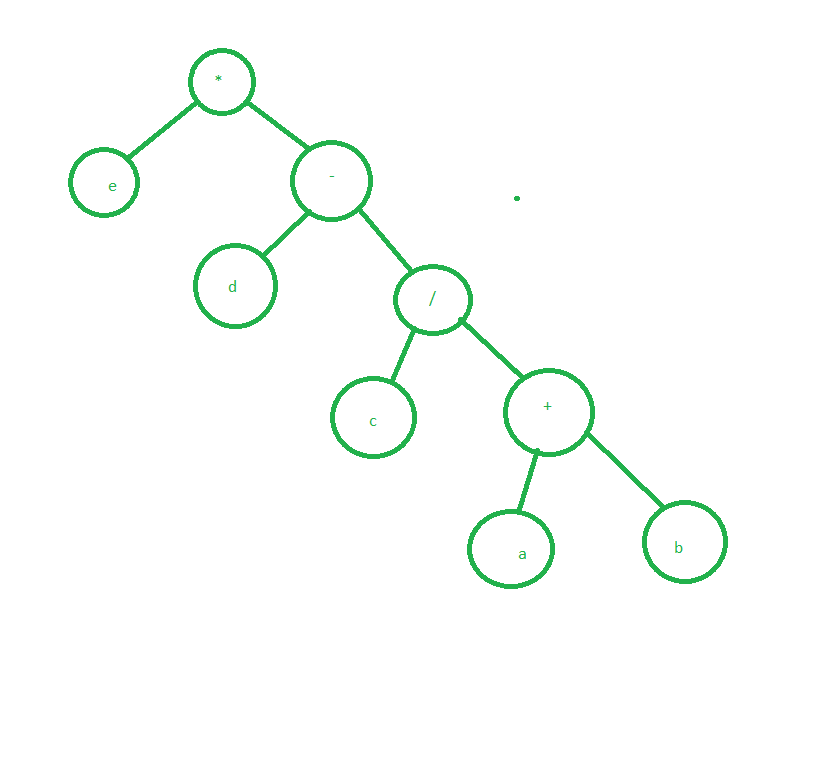
****

Рисунок 16 – Префиксное выражение (**\*-/+abcde**)

* + 1. **В каком порядке будет проходиться бинарное дерево, если алгоритм обхода в ширину будет запоминать узлы не в очереди, а в стеке?**

Если алгоритм обхода в ширину будет запоминать узлы не в очереди, а в стеке, то порядок обхода дерева изменится. Вместо классического порядка по уровням, при котором ближайшие узлы к корню обрабатываются раньше, узлы будут обрабатываться в обратном порядке. Это произойдет из-за того, что стек работает по принципу "последним пришел - первым ушел" (LIFO - Last-In-First-Out).

Порядок обхода в ширину с использованием стека будет следующим:

* Начнем с корневой вершины и добавим ее в стек.
* Затем извлечем вершину из вершины стека, обработаем ее и добавим в стек ее потомков (сначала правого, затем левого).
* Повторяем шаг 2 для вершины, находящейся на вершине стека.
* Продолжаем извлекать вершины из вершины стека и добавлять их потомков до тех пор, пока стек не опустеет.

Итак, в этом случае обход будет происходить в порядке, обратном обходу в ширину. Первыми будут обработаны узлы на самом нижнем уровне, затем узлы на более высоких уровнях.

* + 1. **Постройте бинарное дерево поиска, которое в результате симметричного обхода дало бы следующую последовательность узлов: 40 45 46 50 65 70 75**

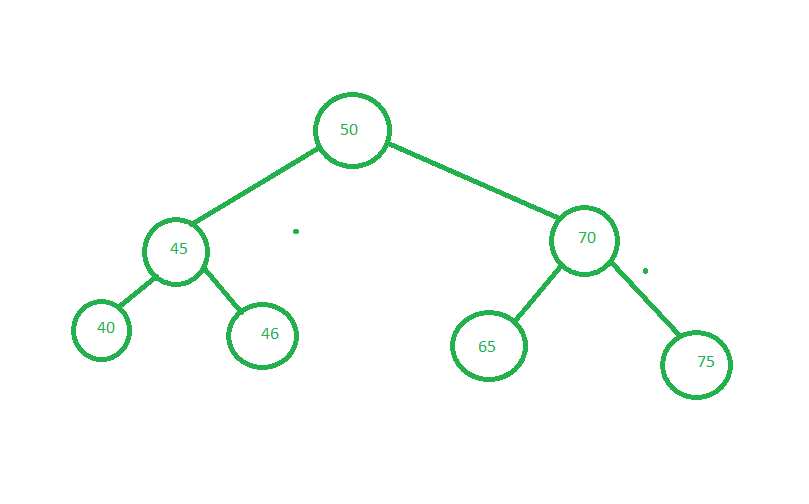
****

Рисунок 17 – ответ на задание 18

* + 1. **Приведите ниже последовательность получена путем прямого обхода бинарного дерева поиска. Постройте это дерево: 50 45 35 15 40 46 65 75 70**

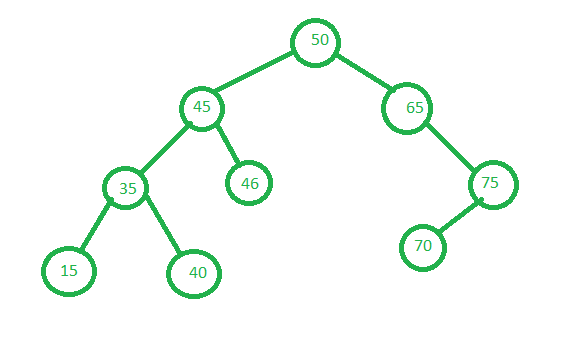
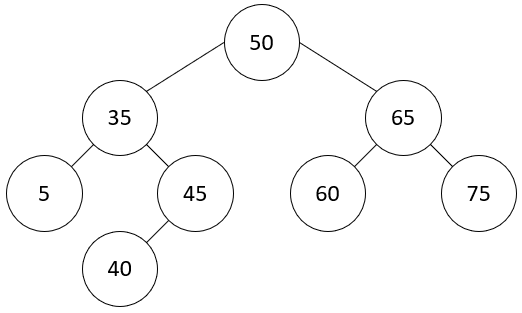
****

Рисунок 18 – Ответ на задание 19

* + 1. **Дано следующее бинарное дерево поиска**



80

Рисунок 19 – Бинарное дерево поиска

Покажите дерево:

* после включения узлов 1, 48, 75, 100
* после удаления узлов 5, 35
* после удаления узла 45
* после удаления узла 50
* после удаления узла 65 и вставки его снова

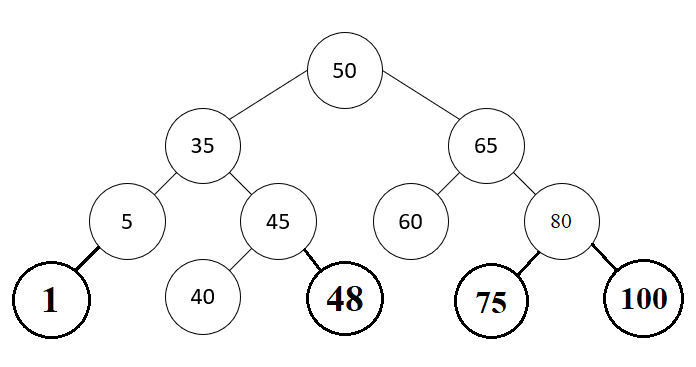


Рисунок 20 – Дерево поиска после добавления **1**, **48**, **75**, **100**

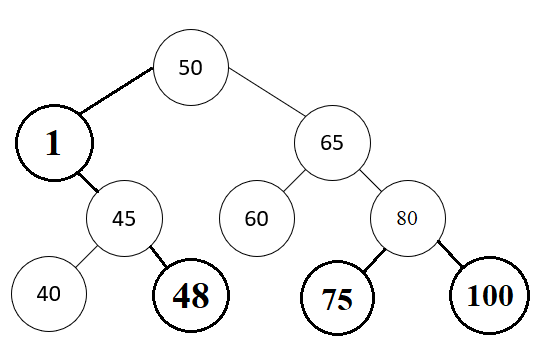


Рисунок 21 – Дерево поиска после удаления **5**, **35**

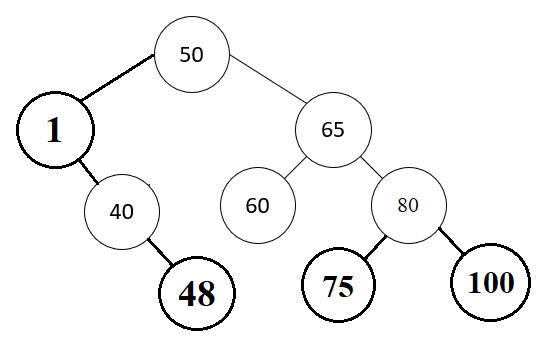


Рисунок 22 – Дерево поиска после удаления **45**

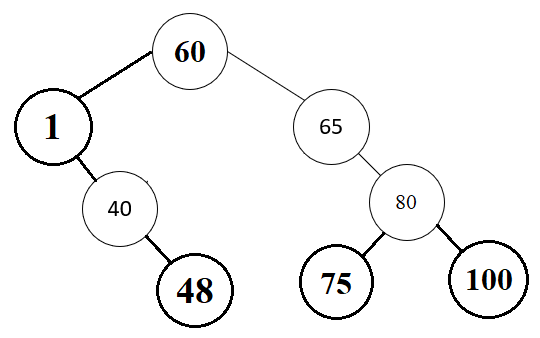


Рисунок 23 – Дерево поиска после удаления 50

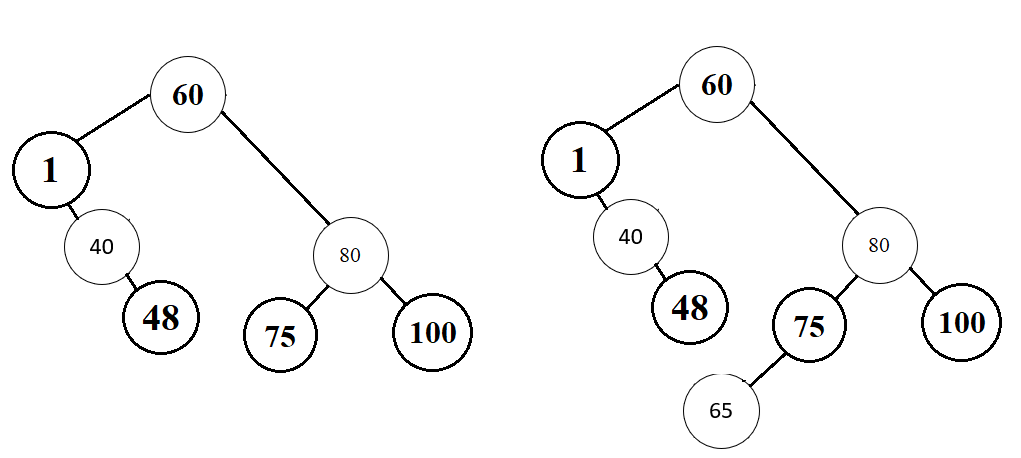


Рисунок 24 – Дерево поиска после удаления 65 и вставки его снова

* 1. **Алгоритмы функций**

Листинг 1. Структура узла в дереве выражений

|  |
| --- |
| struct node  {  char data;  node\* leftNodePtr;  node\* rightNodePtr;  node(char userData)  {  this->data = userData;  this->leftNodePtr = nullptr;  this->rightNodePtr = nullptr;  }  }; |

Данный код определяет структуру узла для дерева выражения. Узел состоит из следующих частей:

* Информационная часть узла, которая содержит **целое число**.
* Это указатель на левое поддерево, т.е., на другой узел, являющийся левым потомком данного узла.
* Это указатель на правое поддерево, т.е., на другой узел, являющийся правым потомком данного узла.

Таким образом, эта структура представляет базовый элемент для построения дерева выражения, где каждый узел содержит целое число и указатели на его левое и правое поддеревья.

Листинг 2. Функция для построения дерева выражения по постфиксной записи выражения

|  |
| --- |
| node\* buildExpressionTree(char userInput[])  {  std::string postfixString = userInput;  std::stack<node\*> s;  for (char c : postfixString)  {  if (isOperator(c))  {  node\* x = s.top();  s.pop();  node\* y = s.top();  s.pop();  node\* a = new node(c);  a->leftNodePtr = y;  a->rightNodePtr = x;  s.push(a);  }  else {  s.push(new node(c));  }  }  return s.top();  } |

Данный код представляет собой функцию, которая используется для построения дерева выражения по заданному постфиксному выражению.

Алгоритм работает следующим образом:

* Инициализируется стэк для хранения элементов указателей на узлы дерева
* Начинается цикл для пробега по всей
* Если из строки мы встретили оператор, то из стека выталкиваем два операнда соответствующих этому операнду. Строим поддерево в корне которого оператор, а в левом/правом потомке операнды из стека
* Иначе просто вталкиваем наше поддерево в стек по корневому элементу

Листинг 3. Функция для подсчёта значения левого поддерева

|  |
| --- |
| int calculateLeftSubtree(node\* root)  {  if (isOperator(root->data))  {  int leftValue = calculateLeftSubtree(root->leftNodePtr);  int rightValue = calculateLeftSubtree(root->rightNodePtr);  switch (root->data) {  case '+':  return leftValue + rightValue;  case '-':  return leftValue - rightValue;  case '\*':  return leftValue \* rightValue;  default:  return 0;  }  }  else  {  return root->data - '0';  }  } |

Этот алгоритм предназначен для вычисления значения выражения в левом поддереве

Алгоритм работает следующим образом:

* Сначала он проверяет, пусто ли дерево, путем проверки указателя корневого узла ***root***. Если да, то это означает, что дерево пустое, и алгоритм завершает выполнение.
* Если дерево не пусто, алгоритм создает пустую очередь ***q***, в которую помещает корневой узел ***root***. Это начальное действие позволяет начать обход с корневого узла.
* Затем алгоритм инициализирует две переменные ***sum*** и ***count***, для общей суммы всех информационных частей узлов и количества узлов дерева соответственно.
* Алгоритм начинает цикл, который продолжается до тех пор, пока очередь не станет пустой. Это будет означать что все узлы дерева были посещены.
* Внутри цикла извлекается текущий узел ***current*** из начала очереди. Значение информационной части узла ***current*** добавляется к ***sum***, а счётчик ***count*** инкрементируется.
* Если у текущего узла есть левое поддерево – помещение его в очередь.
* Если у текущего узла есть правое поддерево – помещение его в очередь.
* После завершения цикла, алгоритм вычисляет среднее арифметическое всех узлов, разделяя общую сумму ***sum*** на количество узлов ***count***.

Таким образом этот алгоритм реализует подсчёт среднего арифметического всех узлов дерева, обходя его «в ширину». Очередь позволяет обойти каждый уровень дерева последовательно, начиная с ***0-го*** и заканчивая ***последним***, что и является обходом дерева в ширину.

Листинг 4. Функция для подсчета количества узлов в дереве

|  |
| --- |
| int countNodes(Node\* root) {  if (!root) {  return 0;  }  return 1 + countNodes(root->left) + countNodes(root->right); //рекурснивный подсчёт количества узлов  } |

Алгоритм работает рекурсивно и имеет следующую логику:

* Начальное условие проверка на то, является ли текущий узел оператором
* Если да, то рекурсивно высчитываем значения операндов. Также меняется и сам оператор
* Далее оператор ветвления по операции, которая содержится в текущем узле дерева
* Иначе возвращаем целочисленное значение в узле. Случай когда мы пришли в лист

Этот алгоритм эффективно подсчитывает количество узлов в дереве, используя рекурсию для обхода всех узлов и суммирует их.

Листинг 5. Функция для подсчета узлов на заданном уровне

|  |
| --- |
| int countOnLayer(node\* root, int targetLayer, int curLayer)  {  if (curLayer == targetLayer)  {  if (root == nullptr)  {  return 0;  }  return 1;  }  return countOnLayer(root->leftNodePtr, targetLayer, curLayer + 1) + countOnLayer(root->rightNodePtr, targetLayer, curLayer + 1);  } |

Данный алгоритм предназначен для вычисления количества узлов на заданном уровне. Алгоритм работает рекурсивно и имеет следующую логику:

* Условие выхода. Условие гласит, что если наш корень или узел равен нулевому указателю, то выходим со значением 0. Нулевой уровень.
* Второе условие, если мы пришли в нулевой уровень но не с пустым дерево. Есть только корень а значит эдемент 1

родительского узла, сохраняя сбалансированную структуру дерева после удаления максимального элемента (если это возможно).

* Иначе мы не на нужном уровне и тогда двигаемся дальше по детям от заданного узла меняя корень на его правого и левого потомка

Листинг 6. Функция вывода дерева виде постфиксного выражения

|  |
| --- |
| void postfixPrint(node\* root)  {  if (root == nullptr)  {  return;  }  postfixPrint(root->leftNodePtr);  postfixPrint(root->rightNodePtr);  std::cout << root->data;  } |

Данный алгоритм выводит выражение и древа выражения. Алгоритм работает рекурсивно и имеет следующую логику:

* Условие выхода. Если мы пришли в последнего потомка, то выходим из функции
* Иначе обход по потомкам в обратном порядке.
  1. **Код программы**

Листинг 7. Полный листинг задания.

|  |
| --- |
| #include <iostream>  #include <stack>  //----Func to classified digit----//  bool isFormula(char c)  {  return((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z'));  }  //----Func to classified operator----//  bool isOperator(char c)  {  return(c == '+' || c == '-' || c == '\*' || c == '/');  }  //----Func to get operation priority----//  int getPriority(char oper)  {  switch (oper)  {  case '/':  case '\*':  {  return(3);  break;  }  case '+':  case '-':  {  return(2);  break;  }  default:  {  return(0);  break;  }  }  }  //----Stack component realisation----//  struct stackComponent  {  char stackCompData;  stackComponent\* nextStackComponent;  stackComponent(char userData)  {  this->stackCompData = userData;  this->nextStackComponent = nullptr;  }  };  //----Stack realisation by myself----//  struct stack  {  stackComponent\* top;  stack()  {  top = nullptr;  }  char topElement()  {  return(top->stackCompData);  }  void push(char userData)  {  stackComponent\* newComp = new stackComponent(userData);  if (top == nullptr) { top = newComp; }  else  {  newComp->nextStackComponent = top;  top = newComp;  }  }  void showStack()  {  if (top == nullptr) return;  else  {  stackComponent\* iter = top;  while (iter != nullptr)  {  std::cout << iter->stackCompData << ' ';  iter = iter->nextStackComponent;  }  std::cout << '\n';  }  }  void pop()  {  stackComponent\* deleteMark = top;  top = top->nextStackComponent;  delete deleteMark;  }  void clear()  {  while (top != nullptr)  {  pop();  }  }  bool isEmpty()  {  if (top == nullptr) return(true);  return(false);  }  };  //----Node of tree structure----//  struct node  {  char data;  node\* leftNodePtr;  node\* rightNodePtr;  node(char userData)  {  this->data = userData;  this->leftNodePtr = nullptr;  this->rightNodePtr = nullptr;  }  };  //----Func to print tree in postfix form----//  void postfixPrint(node\* root)  {  if (root == nullptr)  {  return;  }  postfixPrint(root->leftNodePtr);  postfixPrint(root->rightNodePtr);  std::cout << root->data;  }  //----Func to print postfix expressionTree----//  node\* buildExpressionTree(char userInput[])  {  std::string postfixString = userInput;  std::stack<node\*> s;  for (char c : postfixString)  {  if (isOperator(c))  {  node\* x = s.top();  s.pop();  node\* y = s.top();  s.pop();  node\* a = new node(c);  a->leftNodePtr = y;  a->rightNodePtr = x;  s.push(a);  }  else {  s.push(new node(c));  }  }  return s.top();  }  //----Func to convert infix to postfix----//  void convertToPostfix(std::string infixString, char postfixString [])  {  unsigned int formulaCounter = 0;  unsigned int postCounter = 0;  char formulaElement;  stack postfixHelp;  while (formulaCounter < infixString.length())  {  formulaElement = infixString[formulaCounter];  if (formulaElement == '(')  {  postfixHelp.push(formulaElement);  formulaCounter++;  continue;  }  if (formulaElement == ')')  {  while (!postfixHelp.isEmpty() && postfixHelp.topElement() != '(')  {  postfixString[postCounter++] = postfixHelp.topElement();  postfixHelp.pop();  }  if (!postfixHelp.isEmpty())  {  postfixHelp.pop();  }  formulaCounter++;  continue;  }  if (getPriority(formulaElement) == 0)  {  postfixString[postCounter++] = formulaElement;  }  else  {  if (postfixHelp.isEmpty())  {  postfixHelp.push(formulaElement);  }  else  {  while (!postfixHelp.isEmpty() && postfixHelp.topElement() != '(' && getPriority(formulaElement) <= getPriority(postfixHelp.topElement()))  {  postfixString[postCounter++] = postfixHelp.topElement();  postfixHelp.pop();  }  postfixHelp.push(formulaElement);  }  }  formulaCounter++;  }  while (!postfixHelp.isEmpty())  {  postfixString[postCounter++] = postfixHelp.topElement();  postfixHelp.pop();  }  postfixString[postCounter] = '\0';  }  //----Function to calculate lefttree expression----//  int calculateLeftSubtree(node\* root)  {  if (isOperator(root->data))  {  int leftValue = calculateLeftSubtree(root->leftNodePtr);  int rightValue = calculateLeftSubtree(root->rightNodePtr);  switch (root->data) {  case '+':  return leftValue + rightValue;  case '-':  return leftValue - rightValue;  case '\*':  return leftValue \* rightValue;  default:  return 0;  }  }  else  {  return root->data - '0';  }  }  //----Func to count amount of modes in level----//  int countOnLayer(node\* root, int targetLayer, int curLayer)  {  if (curLayer == targetLayer)  {  if (root == nullptr)  {  return 0;  }  return 1;  }  return countOnLayer(root->leftNodePtr, targetLayer, curLayer + 1) + countOnLayer(root->rightNodePtr, targetLayer, curLayer + 1);  }  //----Func to show prefix form of left subtree----//  std::string prefixPrintLeftSubtree(node\* root, std::string ans)  {  if (root->leftNodePtr == nullptr)return(root->data + ans);  return root-> data + prefixPrintLeftSubtree(root->leftNodePtr, ans) + prefixPrintLeftSubtree(root->rightNodePtr, ans);  }  int main()  {  std::cout << "//---------EXPRESSION TREE TASKVAR NUMBER 17---------\\ \n";  std::cout << "//----------------------------------------------\\ \n";  std::cout << "Enter the formula. Enter symbols: (), +-\*/, a,bc..., 123..., but no ' ' \n";  std::cout << "--->Your formula: ";  std::string formula;  node\* expTree = nullptr;  std::cin >> formula;  char\* postfixFormula = (char\*)malloc(sizeof(char) \* formula.length());  convertToPostfix(formula, postfixFormula);  expTree = buildExpressionTree(postfixFormula);  std::cout << "Expression tree in postfixForm ";  postfixPrint(expTree);  std::cout << '\n';  std::cout << "//----------------------------------------------\\ \n";  std::cout << "Value of left subtree expression " << calculateLeftSubtree(expTree->leftNodePtr) << '\n';  std::cout << "//----------------------------------------------\\ \n";  std::cout << "Amount of nodes on 2 level " << countOnLayer(expTree, 2, 0);  std::cout << "//----------------------------------------------\\ \n";  std::cout << "Left tree in prefix form " << prefixPrintLeftSubtree(expTree->leftNodePtr, "");  std::cout << "//----------------------------------------------\\ \n";  std::cout << "//---------EXIT OF THE PROGRAMM---------\\ \n";  std::cout << "//----------------------------------------------\\ \n";  std::cout << "//----------------------------------------------\\ \n";  std::cout << "//----------------------------------------------\\ \n";  } |

* 1. **Тестирование**

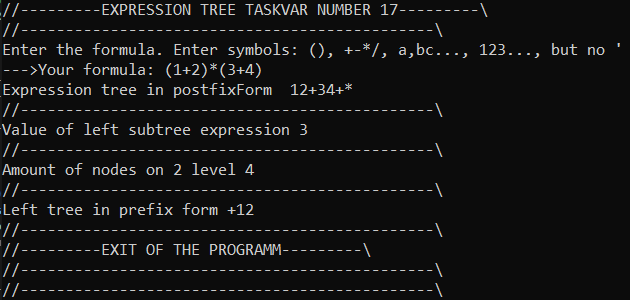


Рисунок 25 – Тестирование программы

1. **ВЫВОД**

Цель данной работы заключается в приобретении навыков и умений в области разработки и реализации операций, связанных с бинарными деревьями. Эти навыки включают в себя умение создавать, обходить и модифицировать бинарные деревья, что является важной частью структур данных и алгоритмов. Работа с бинарными деревьями может оказаться полезной при решении различных задач, таких как поиск, сортировка, и многие другие операции. В итоге, целью является повышение навыков программирования и алгоритмической компетенции в области бинарных деревьев.