|  |  |  |
| --- | --- | --- |
|  |  |  |
| 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 группы **ИКБО-13-22** | Абанин Фёдор Дмитриевич |
| **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 1)**

Значение информационной части: **целое число**

Операции варианта:

* Определить высоту дерева;
* Определить длину пути дерева (количество ребер), используя алгоритм прямого обхода;
* Вычисляет среднее арифметическое всех чисел в дереве;
  + 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*: идеально сбалансированное из n узлов (не AVL).

* Реализовать операции общие для вариантов c 1 по 10:

Создать идеально сбалансированное бинарное дерево из n узлов. Структура узла дерева включает: информационная часть узла, указатель на левое и указатель на правое поддерево. Информационная часть узла определена вариантом.

* Реализовать операции варианта.

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).

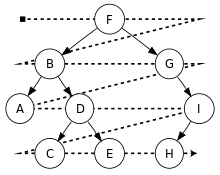


Рисунок 1 – пример обхода бинарного дерева в ширину

**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).

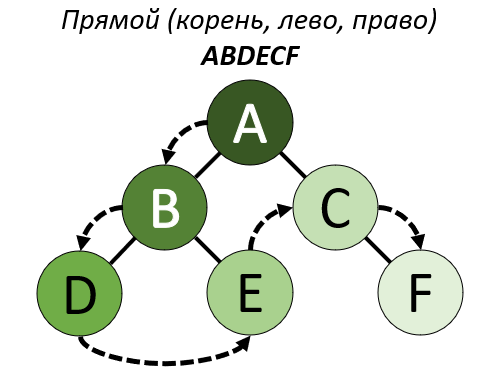


Рисунок 2 – Прямой обход дерева (корень, лево, право)

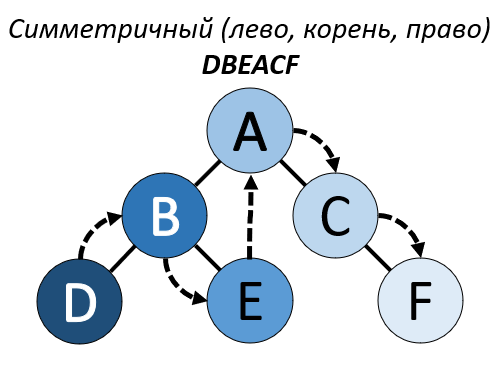


Рисунок 3 – Симметричный обход дерева (лево, корень, право)

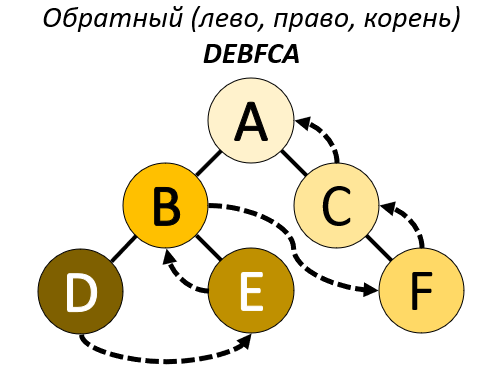


Рисунок 4 – Обратный обход дерева (лево, право, корень)

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**

|  |  |  |  |
| --- | --- | --- | --- |
| *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 |

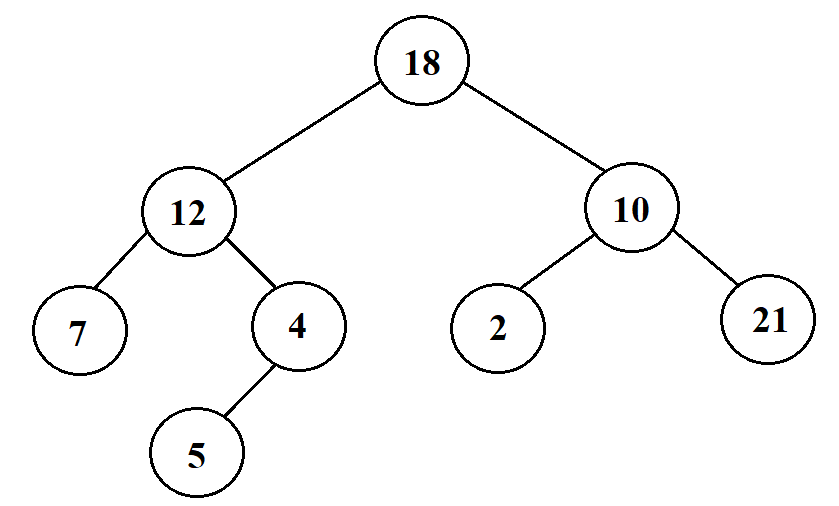


Рисунок 5 – бинарное дерево, представленное таблицей

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

Представление трех видов обхода дерева в глубину:

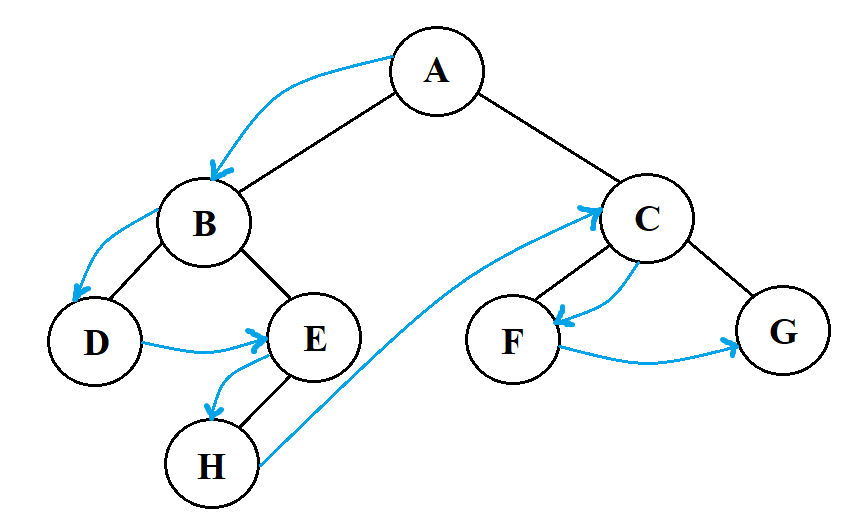
****

Рисунок 6 – Прямой обход дерева (**ABDEHCFG**)

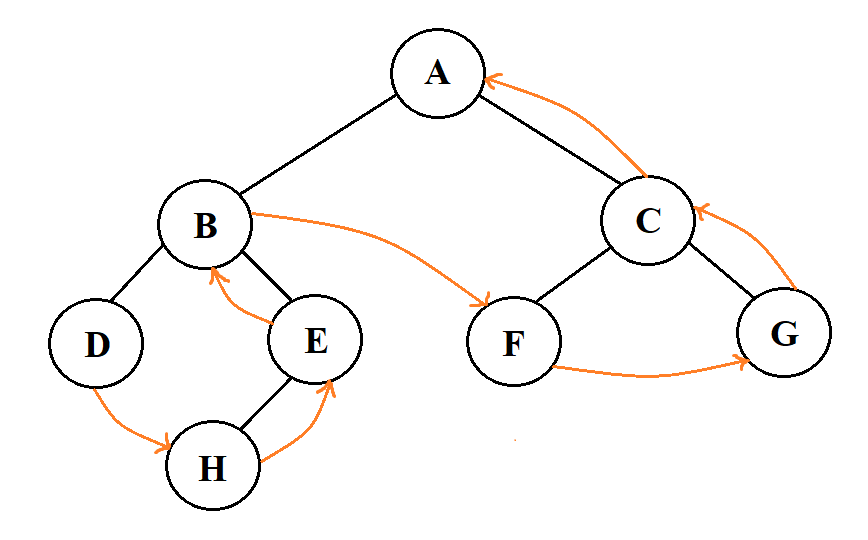
****

Рисунок 7 – Обратный обход дерева (**DHEBFGCA**)

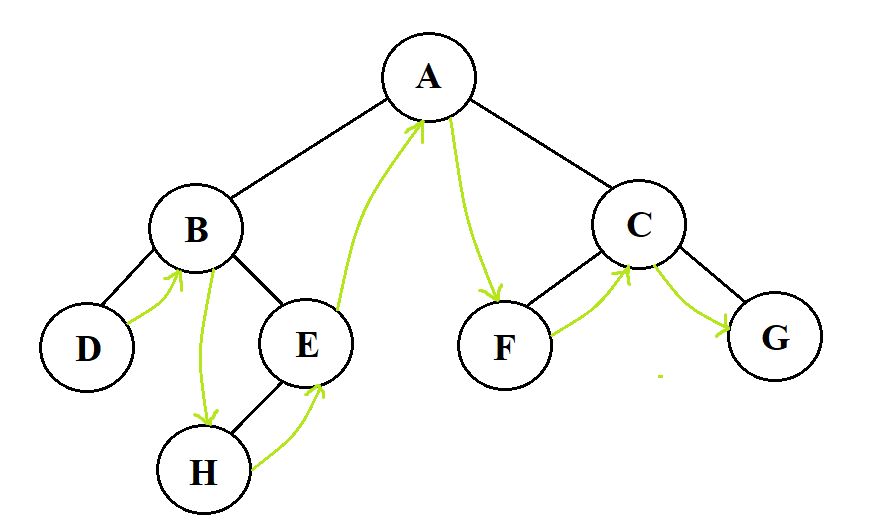
****

Рисунок 8 – Симметричный обход дерева (**DBHEAFCG**)

* + 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.**

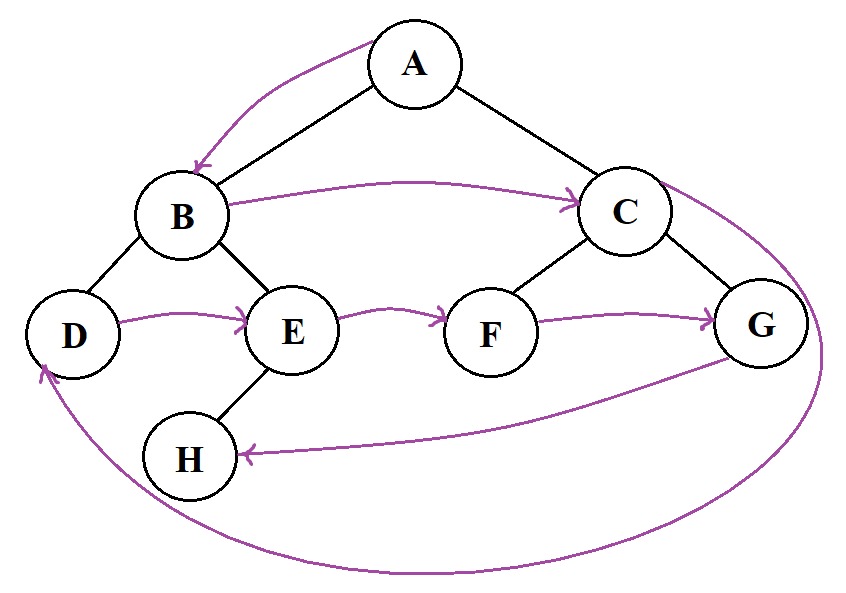
****

Рисунок 9 – обход дерева в ширину (**ABCDEFGH**)

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**

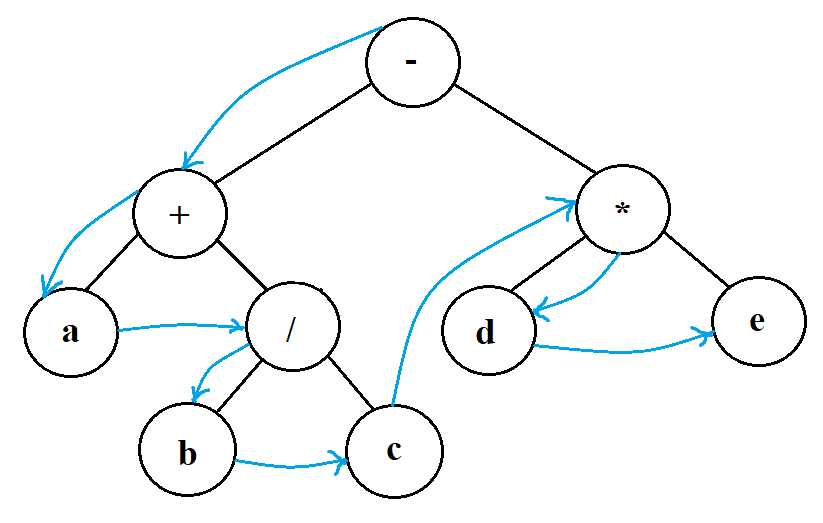
****

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

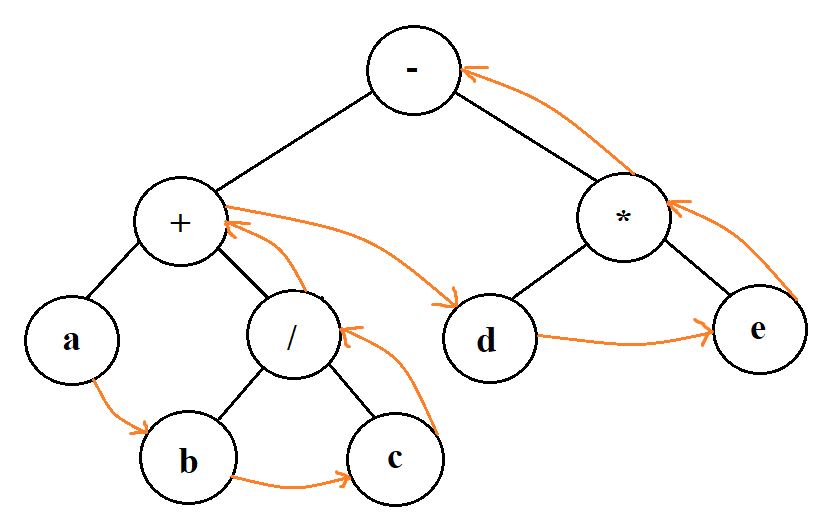
****

Рисунок 11 – Обратный обход дерева выражений (**abc/+de\*-**)

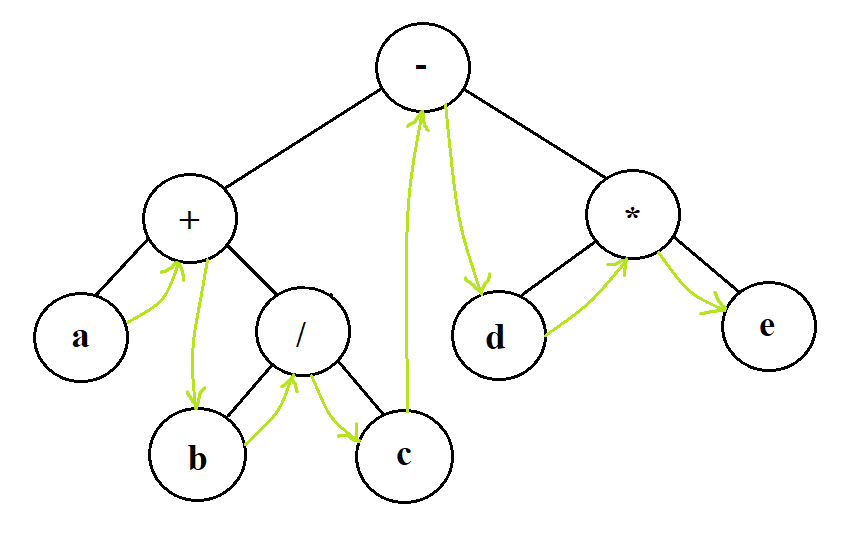
****

Рисунок 12 – Симметричный обход дерева выражений (**a+b/c-d\*e**)

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

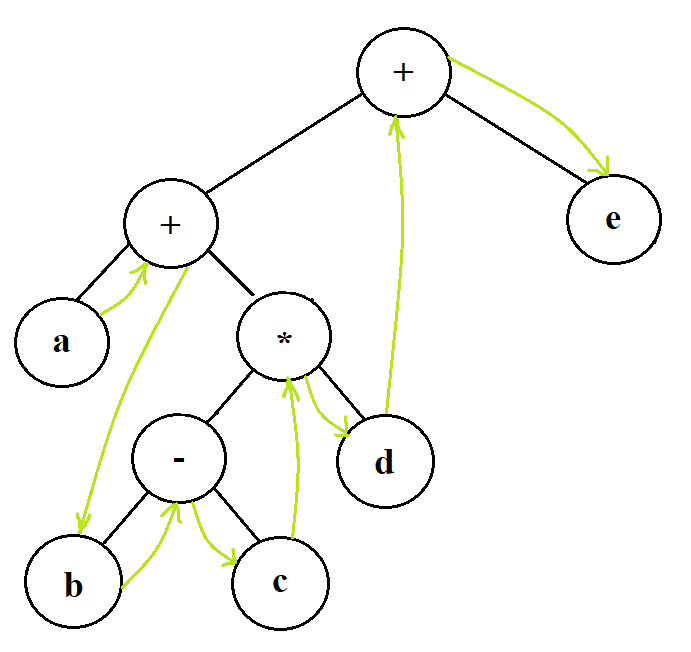
****

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

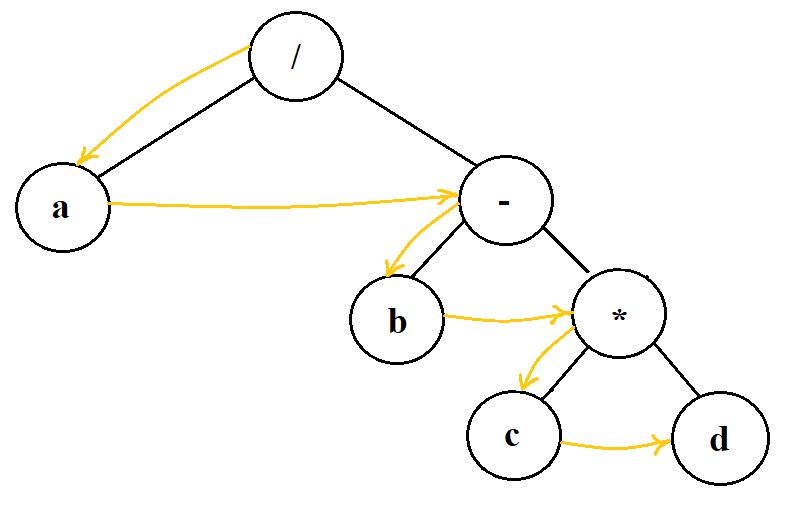
****

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

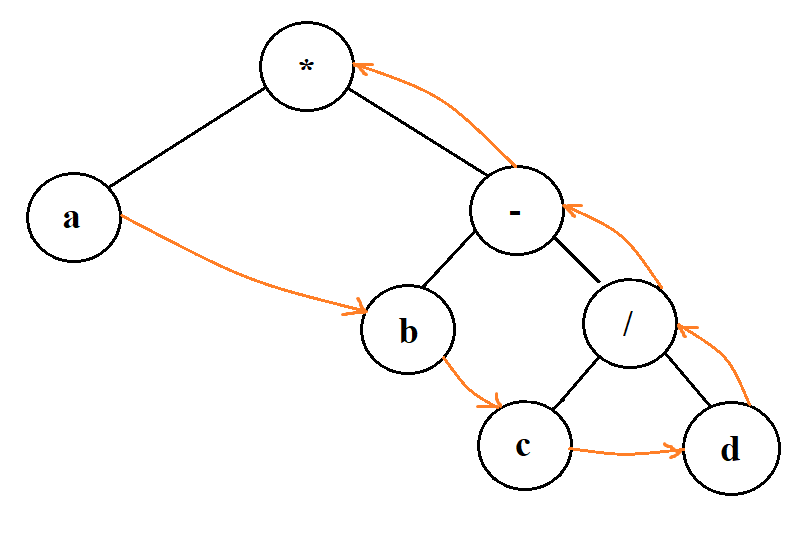
****

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

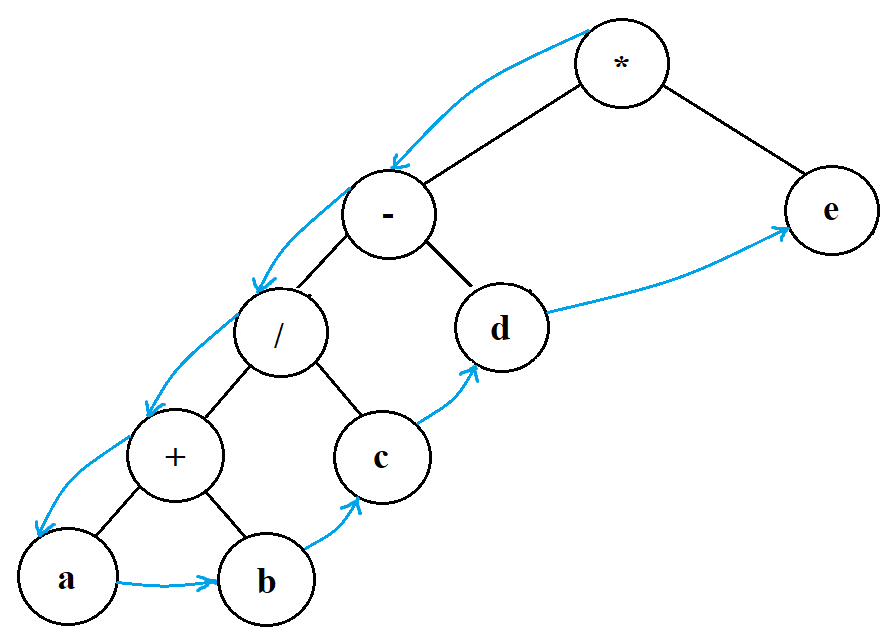
****

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**

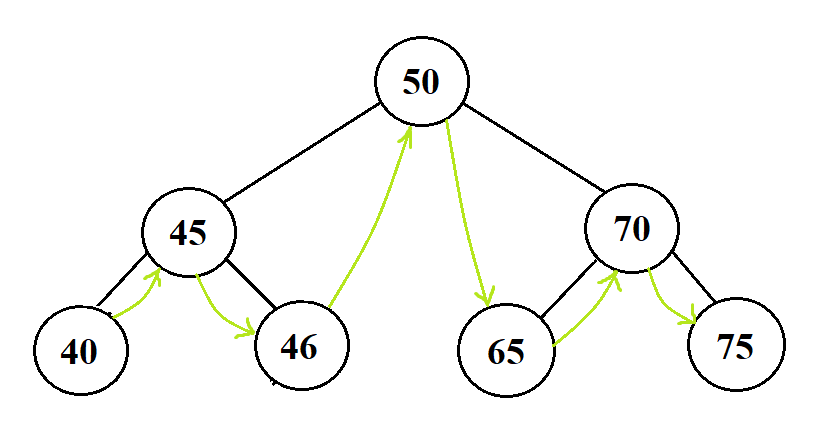
****

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**

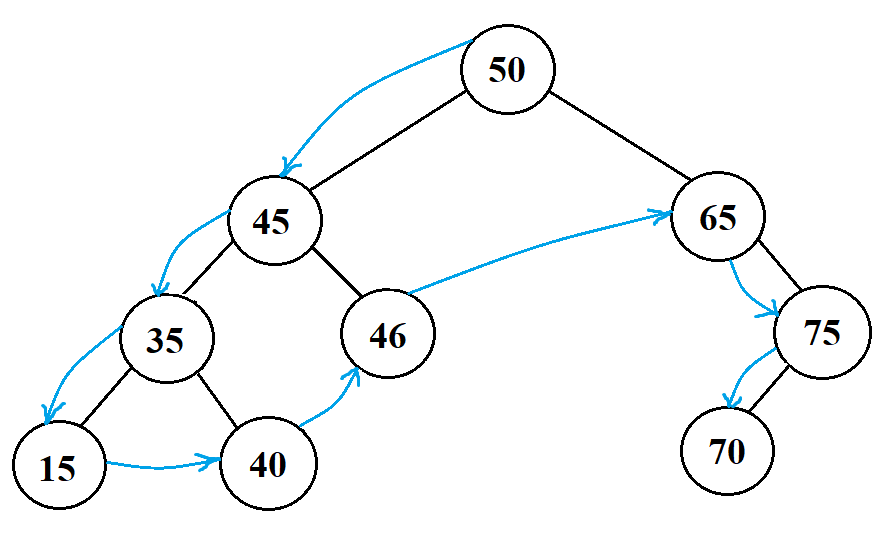
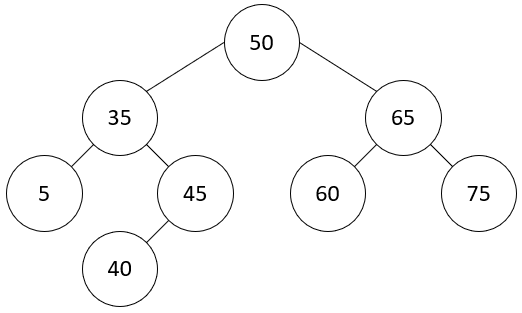
****

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

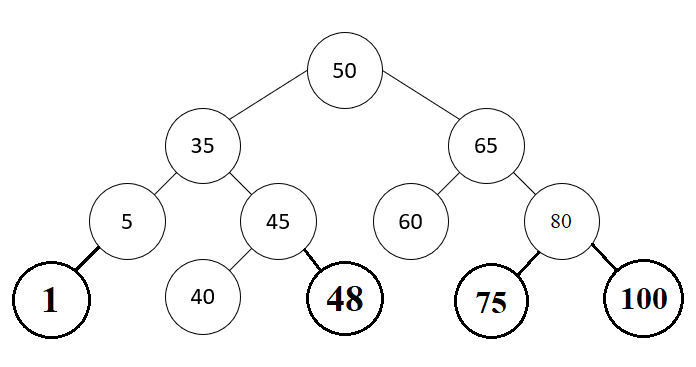


Рисунок 20 – Дерево поиска после добавления **1**, **48**, **75**, **100**

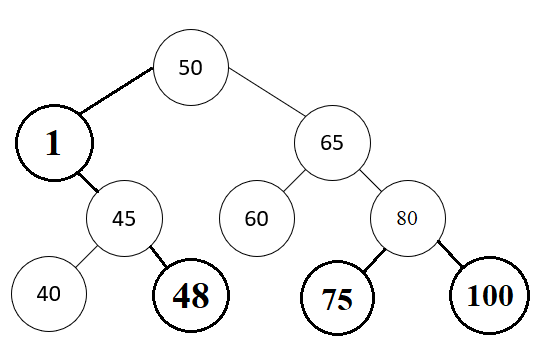


Рисунок 21 – Дерево поиска после удаления **5**, **35**

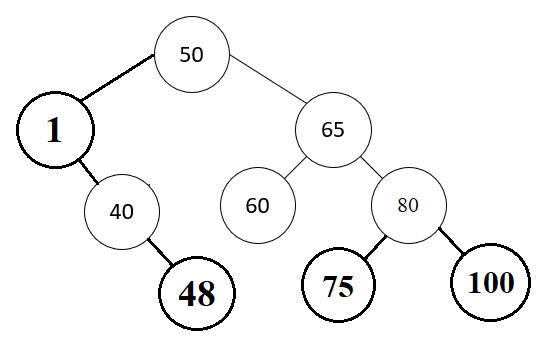


Рисунок 22 – Дерево поиска после удаления **45**

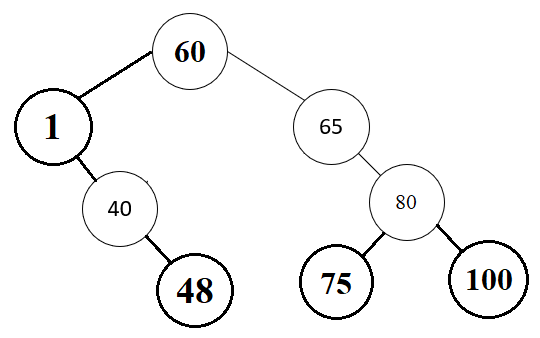


Рисунок 23 – Дерево поиска после удаления 50

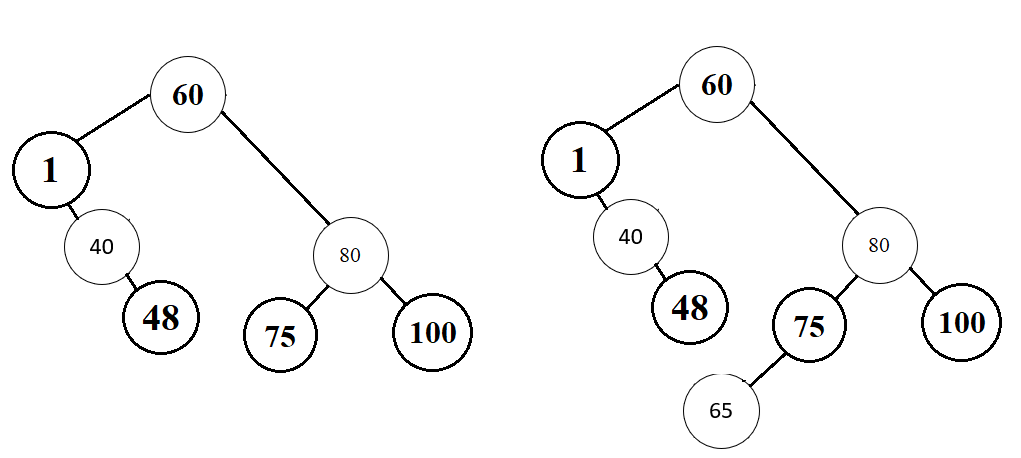


Рисунок 24 – Дерево поиска после удаления 65 и вставки его снова

* 1. **Алгоритмы функций**

Листинг 1. Структура узла в бинарном дереве

|  |
| --- |
| struct Node {  int data; //информационная часть узла  Node \*left; //левое поддерево  Node \*right; //правое поддерево   Node(int value) : data(value), left(nullptr), right(nullptr) {} //параметризированный конструктор узла }; |

Данный код определяет структуру узла для бинарного дерева. 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. Если левого поддерева нет, указатель устанавливается в ***nullptr***.
* Это указатель на правое поддерево, т.е., на другой узел, являющийся правым потомком данного узла. Если правого поддерева нет, указатель устанавливается в ***nullptr***.

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

Листинг 2. Функция для вставки элементов в сбалансированное бинарное дерево

|  |
| --- |
| Node \*insertBalanced(Node \*root, int data, int n) {  if (n <= 0) {  return nullptr;  }   int mid = n / 2;  root = new Node(data);  root->left = insertBalanced(root->left, data - mid, mid); //рекурсивное создание левого поддерева  root->right = insertBalanced(root->right, data + mid, n - mid - 1); //рекурсивное создание правого поддерева   return root; //возвращение корневого узла(дерева) } |

Данный код представляет собой функцию, которая используется для построения идеально сбалансированного бинарного дерева из ***n*** узлов. На вход функция принимает значение корневого узла, информационную часть корневого узла ***data*** и количество элементов в дереве ***n***.

Алгоритм работает следующим образом:

* Проверка на случай, если ***n*** меньше или равно нулю. Если ***n*** не положительное число, функция возвращает ***nullptr***, что означает пустое дерево.
* Вычисляется средний элемент ***mid*** для разделения ***n*** на две части. Это необходимо для создания идеально сбалансированного дерева.
* Создается новый узел с информационной частью, переданной функции, и устанавливается как корень дерева.
* Рекурсивно вызывается функция, чтобы построить левое поддерево. Функция принимает значение левого поддерева, в качестве информационной части передаётся значение ***data*** – ***mid***, в качестве количества элементов в дереве передаётся ***mid***.
* Рекурсивно вызывается функция, чтобы построить правое поддерево.

Функция принимает значение правого поддерева, в качестве информационной части передаётся значение ***data*** + ***mid***, в качестве количества элементов в дереве передаётся ***n*** - ***mid*** - ***1***.

Таким образом данный алгоритм создаёт *идеально сбалансированное дерево*, ***в котором все его уровни, за исключением, возможно, последнего, полностью заполнены.***

Листинг 3. Функция для подсчёта среднего арифметического всех элементов

|  |
| --- |
| double calculateAverage(Node \*root) {  if (!root) {  return 0.0;  }   double sum = root->data; // сумма равна значению текущего узла  double count = 1.0; // начальное количество узлов равно 1   if (root->left) {  sum += calculateAverage(root->left); // добавляем сумму и количество из левого поддерева  count += countNodes(root->left); // увеличиваем количество узлов из левого поддерева  }   if (root->right) {  sum += calculateAverage(root->right); // добавляем сумму и количество из правого поддерева  count += countNodes(root->right); // увеличиваем количество узлов из правого поддерева  }   return sum / count; // возвращаем среднее арифметическое } |

Этот код реализует алгоритм для вычисления среднего арифметического всех узлов в дереве, используя алгоритм обхода в "ширину". Для этого алгоритма используется очередь, которая работает по принципу FIFO (первый пришел, первый вышел).

Алгоритм работает следующим образом:

* Сначала происходит проверка, пустое ли дерево, путем проверки указателя корневого узла **root**. Если дерево пустое, алгоритм завершает выполнение.
* Если дерево не пустое, то алгоритм создает пустую очередь q и помещает в нее корневой узел **root**. Это позволяет начать обход с корневого узла.
* Затем инициализируются две переменные: **sum** (сумма) и **count** (количество), для хранения общей суммы всех информационных частей узлов и количества узлов дерева соответственно.
* Алгоритм начинает цикл, который продолжается до тех пор, пока очередь не станет пустой. Внутри цикла извлекается текущий узел **current** из начала очереди.
* Значение информационной части узла **current** добавляется к **sum**, а счетчик **count** увеличивается.
* Если у текущего узла есть левое поддерево, оно также помещается в очередь.
* Если у текущего узла есть правое поддерево, оно также помещается в очередь.
* После завершения цикла, алгоритм вычисляет среднее арифметическое всех узлов, разделяя общую сумму **sum** на количество узлов **count**.
* Наконец, алгоритм возвращает среднее арифметическое.

Таким образом, этот алгоритм позволяет подсчитать среднее арифметическое всех узлов в дереве. Очередь позволяет обойти каждый уровень дерева последовательно, начиная с 0-го уровня и заканчивая последним.

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 } |

Данный алгоритм предназначен для подсчёта количества узлов в бинарном дереве. Алгоритм работает рекурсивно и имеет следующую логику:

* Начальный вызов функции происходит с указателем на корневой узел ***root*** бинарного дерева в качестве аргумента.
* В начале функция выполняет проверку, является ли переданный корневой узел ***nullptr*** (то есть, дерево пусто). Если это так, функция возвращает 0, так как пустое дерево не содержит узлов.
* Если корневой узел ***root*** не является ***nullptr*** (то есть, дерево не пусто), то функция увеличивает счетчик на **1** (1 означает текущий узел), затем рекурсивно вызывает **countNodes** для левого и правого поддеревьев и суммирует все эти три значения. Рекурсия выполняется до тех пор, пока не достигнуты *листовые узлы* (узлы без потомков). Каждый узел учитывается в подсчете как **1**, и к результату добавляется количество узлов в его левом и правом поддереве. Это позволяет рекурсивно пройти **все** узлы дерева.

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

Листинг 5. Функция для подсчёта высоты дерева

|  |
| --- |
| int getHeight(Node \*root) {  if (!root) {  return 0;  }   int leftHeight = getHeight(root->left); // рекурсивное получение высоты левого поддерева  int rightHeight = getHeight(root->right); // рекурсивное получение высоты правого поддерева   return 1 + max(leftHeight, rightHeight); // возвращение максимальной высоты из двух поддеревьев плюс 1 } |

Данный код представляет собой рекурсивную функцию для вычисления высоты дерева. Высота дерева определяется как наибольшее количество уровней, которые содержат узлы в дереве.

Процесс работы алгоритма, следующий:

* Алгоритм принимает указатель на корневой узел дерева **root**.
* В начале проверяется, является ли переданный указатель пустым (**nullptr**). Если да, то возвращается 0, что указывает на то, что высота данного поддерева равна 0.
* Если переданный указатель не является пустым, алгоритм рекурсивно вызывает себя для левого и правого поддеревьев.
* Вызов функции **getHeight**(**root**->**left**) рекурсивно вычисляет высоту левого поддерева.
* Вызов функции **getHeight**(**root**->**right**) рекурсивно вычисляет высоту правого поддерева.
* Затем возвращается максимальное значение из высоты левого и правого поддерева, увеличенное на 1. Это значение представляет высоту текущего поддерева.
* Полученное значение высоты используется, полагаясь на состояния высот левого и правого поддерева для определения полной высоты дерева.

Таким образом, данный код позволяет определить высоту дерева, обходя его рекурсивно и вычисляя максимальную высоту из всех поддеревьев плюс один уровень.ы

Листинг 6. Функция вывода дерева в виде структуры

|  |
| --- |
| void printTreeStructure(Node \*root, int level = 0) {  if (!root) {  return;  }   printTreeStructure(root->right, level + 1); //идём до крайнего правого узла  for (int i = 0; i < level; i++) { //как дошли - вывод отступов пропорционально уровню узла  cout << " ";  }  cout << root->data << endl; //вывод информационной части узла  printTreeStructure(root->left, level + 1); //если правых поддеревьев у узла не осталось - вывод левых } |

Данный алгоритм предназначен для вывода структуры бинарного дерева на экране с отступами, чтобы легче визуализировать структуру дерева. Начальный вызов функции происходит с указателем на корневой узел ***root*** и начальным уровнем ***level***, равным нулю. Алгоритм работает рекурсивно и имеет следующую логику:

* Сначала функция выполняет проверку, является ли переданный корневой узел ***nullptr*** (то есть, дерево пусто). Если дерево пусто, функция ничего не делает и просто завершается.
* Если корневой узел не является нулевым указателем, алгоритм рекурсивно вызывает **printTreeStructure** для правого поддерева с увеличение уровня ***level*** на **1** . Это позволяет алгоритму двигаться вглубь дерева до крайнего правого узла.
* Затем алгоритм использует цикл, чтобы добавить отступы перед выводом информационной части текущего узла. Количество отступов определяется текущим уровнем ***level***. Каждый уровень добавляет дополнительные пробелы перед выводом информации о текущем узле. Таким образом, узлы на разных уровнях будут отделены пропорциональными отступами.
* Выводит информационную часть текущего узла с переходом на новую строку, чтобы каждый узел выводился на новой строке.
* Завершает рекурсивный вызов, вызывая **printTreeStructure** для левого поддерева с увеличением уровня ***level*** + **1**. Это позволяет алгоритму продолжать обход дерева после вывода всех узлов правого поддерева.
* Рекурсия продолжается, пока все узлы дерева не будут посещены. Каждый узел выводится с соответствующими отступами, отображая иерархию дерева.

В результате выполнения этой функции, структура бинарного дерева будет выведена на экран с учетом уровней и отступов, что поможет визуализировать его структуру. Этот вид вывода часто используется для отладки и визуального анализа бинарных деревьев.

Листинг 7. Функция нахождения длины пути дерева

|  |
| --- |
| int getPathLength(Node \*root) {  if (!root) {  return 0;  }   int leftLength = getPathLength(root->left); // рекурсивный вызов для левого поддерева  int rightLength = getPathLength(root->right); // рекурсивный вызов для правого поддерева   return 1 + leftLength +  rightLength; // возвращение суммы длин путей в левом и правом поддеревьях, плюс 1 для учета текущего узла } |

Данный код представляет собой рекурсивную функцию для вычисления

суммарной длины всех путей в дереве. Длина пути в данном контексте определяется как количество ребер между узлами.

Процесс работы алгоритма, следующий:

* Алгоритм принимает указатель на корневой узел дерева **root**.
* В начале проверяется, является ли переданный указатель пустым (**nullptr**). Если да, то возвращается 0, что указывает на то, что длина пути в данном поддереве равна 0.
* Если переданный указатель не является пустым, алгоритм рекурсивно вызывает себя для левого и правого поддеревьев.
* Вызов функции **getPathLength**(**root**->**left**) рекурсивно вычисляет суммарную длину путей в левом поддереве.
* Вызов функции **getPathLength**(**root**->**right**) рекурсивно вычисляет суммарную длину путей в правом поддереве.
* Затем возвращается сумма длины пути в текущем поддереве, которая равна 1 плюс сумма длин в левом и правом поддеревьях. Это значение представляет суммарную длину всех путей в дереве.
* Полученное значение используется для определения суммарной длины путей в дереве.

Таким образом, данный код позволяет вычислить суммарную длину всех путей в дереве, обходя его рекурсивно и суммируя длины путей в каждом поддереве.

* 1. **Код программы**

Листинг 7. Полный листинг задания.

|  |
| --- |
| #include <iostream> #include <queue> //для обхода в ширину  using namespace std;  //---Структура узла в бинарном дереве---// struct Node {  int data; //информационная часть узла  Node \*left; //левое поддерево  Node \*right; //правое поддерево   Node(int value) : data(value), left(nullptr), right(nullptr) {} //параметризированный конструктор узла };  //---Функция для вставки элементов в сбалансированное бинарное дерево---// Node \*insertBalanced(Node \*root, int data, int n) {  if (n <= 0) {  return nullptr;  }   int mid = n / 2;  root = new Node(data);  root->left = insertBalanced(root->left, data - mid, mid); //рекурсивное создание левого поддерева  root->right = insertBalanced(root->right, data + mid, n - mid - 1); //рекурсивное создание правого поддерева   return root; //возвращение корневого узла(дерева) }   //---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 }   //---Функция для вывода дерева в виде структуры---// void printTreeStructure(Node \*root, int level = 0) {  if (!root) {  return;  }   printTreeStructure(root->right, level + 1); //идём до крайнего правого узла  for (int i = 0; i < level; i++) { //как дошли - вывод отступов пропорционально уровню узла  cout << " ";  }  cout << root->data << endl; //вывод информационной части узла  printTreeStructure(root->left, level + 1); //если правых поддеревьев у узла не осталось - вывод левых }  int getHeight(Node \*root) {  if (!root) {  return 0;  }   int leftHeight = getHeight(root->left); // рекурсивное получение высоты левого поддерева  int rightHeight = getHeight(root->right); // рекурсивное получение высоты правого поддерева   return 1 + max(leftHeight, rightHeight); // возвращение максимальной высоты из двух поддеревьев плюс 1 }  int getPathLength(Node \*root) {  if (!root) {  return 0;  }   int leftLength = getPathLength(root->left); // рекурсивный вызов для левого поддерева  int rightLength = getPathLength(root->right); // рекурсивный вызов для правого поддерева   return 1 + leftLength +  rightLength; // возвращение суммы длин путей в левом и правом поддеревьях, плюс 1 для учета текущего узла }  double calculateAverage(Node \*root) {  if (!root) {  return 0.0;  }   double sum = root->data; // сумма равна значению текущего узла  double count = 1.0; // начальное количество узлов равно 1   if (root->left) {  sum += calculateAverage(root->left); // добавляем сумму и количество из левого поддерева  count += countNodes(root->left); // увеличиваем количество узлов из левого поддерева  }   if (root->right) {  sum += calculateAverage(root->right); // добавляем сумму и количество из правого поддерева  count += countNodes(root->right); // увеличиваем количество узлов из правого поддерева  }   return sum / count; // возвращаем среднее арифметическое }   int main() {  setlocale(LC\_ALL, "Rus");  int n;  cout << "Введите количество узлов (n): ";  cin >> n;   Node \*root = nullptr; //корневой узел   if (n > 0) {  root = insertBalanced(root, n / 2, n); //генерация идеально сбалансированного дерева из n узлов  }   cout << "Дерево после вставки: ";  cout << endl << endl;  printTreeStructure(root); //вывод дерева  int height = getHeight(root); // получение высоты дерева  int pathLength = getPathLength(root); // получение длины пути дерева   cout << "Высота дерева: " << height << endl;  cout << "Длина пути дерева: " << pathLength << endl;   double average = calculateAverage(root); // вычисление среднего арифметического для дерева   cout << "Среднее арифметическое всех чисел в дереве: " << average << endl;     return 0; } |

* 1. **Тестирование**

Рисунок 25 – Тестирование программы с 7 элементами

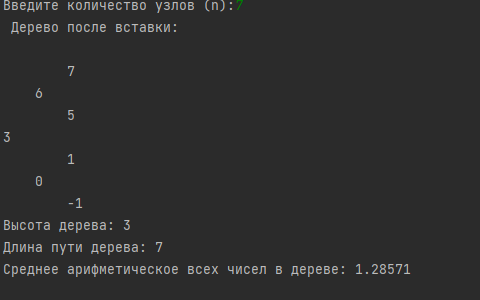


Рисунок 26 – Тестирование программы с 12 элементами

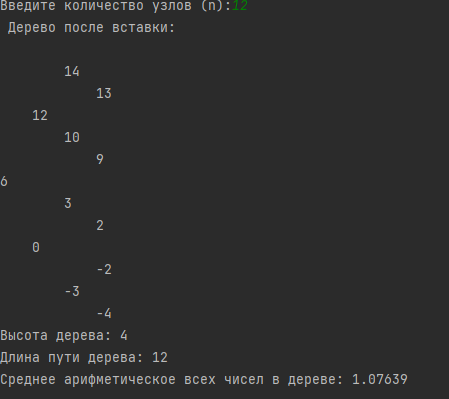


Рисунок 27 - Тестирование программы с 25 элементами

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.