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 Ефремова Полина Александровна

Проверил: ассистент Муравьёва Е.А.

МОСКВА 2023 г.

1. ВВЕДЕНИЕ

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

1.1.Задания (Вариант 9)

Значение информационной части: **символьное значение**

Операции варианта:

- Найти максимальное значение среди значений листьев дерева;
- Определить уровень, на котором находится заданное значение;
- Определить количество цифр в правом поддереве исходного дерева.

1.1.1. Формулировка задания 1

Ответьте на вопросы и выполните упражнения.

1.1.2. 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 с 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.

2. ХОД РАБОТЫ

2.1.Ответы на вопросы

2.1.1. 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.

2.1.2. 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.

2.1.3. 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.

2.1.4. 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 (то есть 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.

2.1.5. 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.

2.1.6. 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 быть 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.

2.1.7. 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:

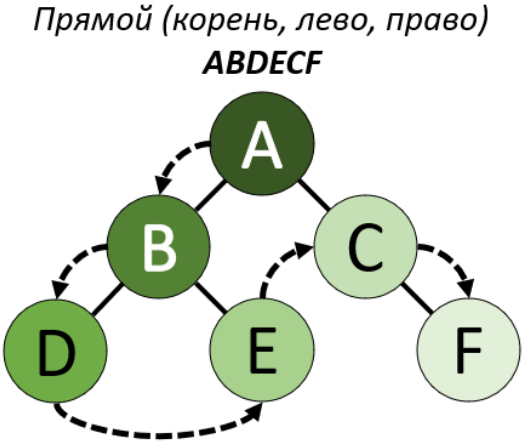
- *Корневая вершина (Root)*: Это вершина, которая находится в самом верхнем уровне бинарного дерева. Все остальные вершины происходят от корневой вершины.
- *Левый и правый потомки*: Каждая вершина может иметь не более двух потомков: левого и правого. Левый потомок находится слева от родительской вершины, 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.

2.1.8. Дайте определение алгоритму обхода.

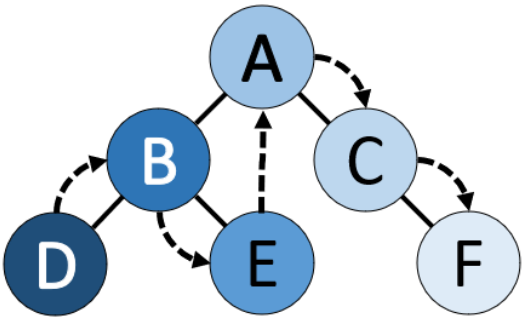
Алгоритм обхода (или алгоритм прохода) — это специфическая процедура, которая определяет порядок посещения элементов (например, узлов или вершин) в структуре данных, такой как дерево, граф, массив и другие. Обход используется для систематического и последовательного доступа ко всем элементам структуры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

Алгоритмы обхода могут быть применены к различным задачам, таким как поиск, вывод данных, анализ и обработка информации внутри структуры. Они могут также предоставлять разные способы доступа к элементам в зависимости от порядка обхода, такие как в ***прямой, симметричный и обратный***.



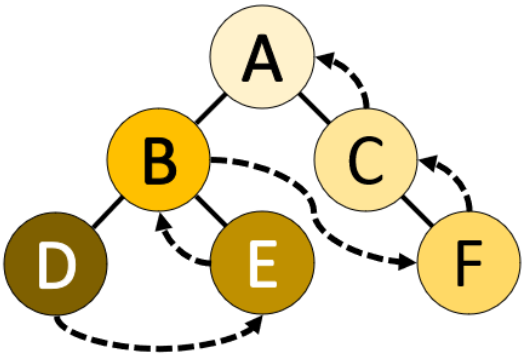
Симметричный (лево, корень, право)

DBEACF



Обратный (лево, право, корень)

DEBFCA



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

2.1.9. 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:

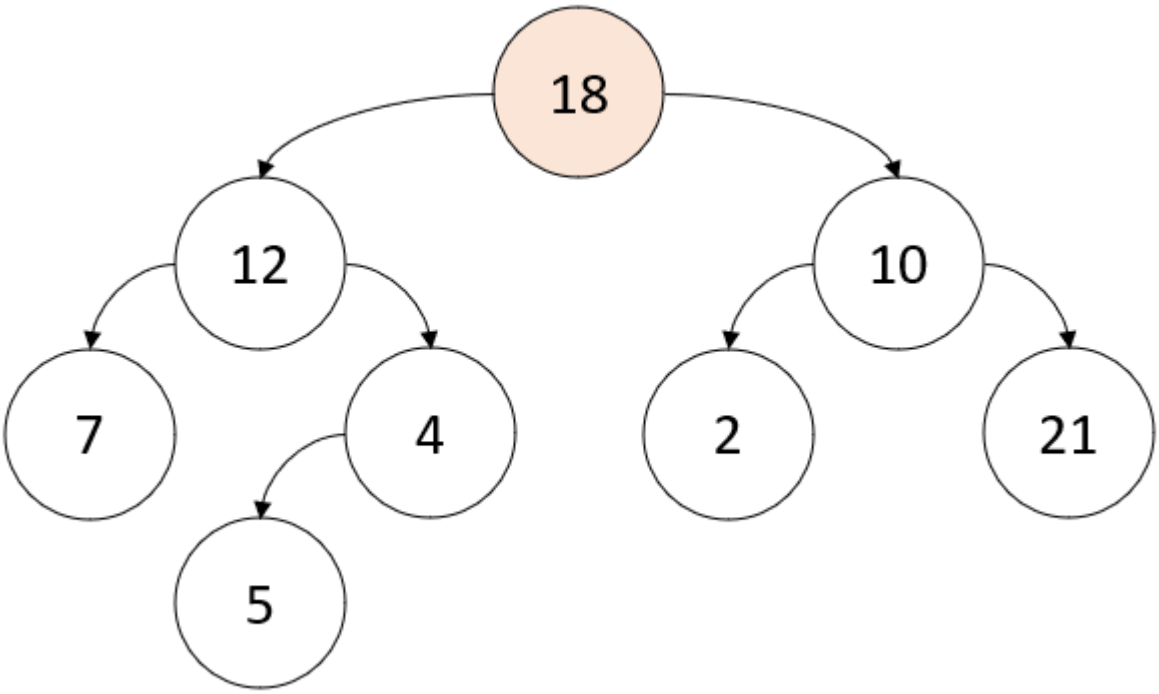
$$\mathbf{h}\boldsymbol{eight}(\boldsymbol{root}) = \mathbf{max}\big(\boldsymbol{height}(\boldsymbol{root.left}), \boldsymbol{height}(\boldsymbol{root.right})\big) + \mathbf{1}$$

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.

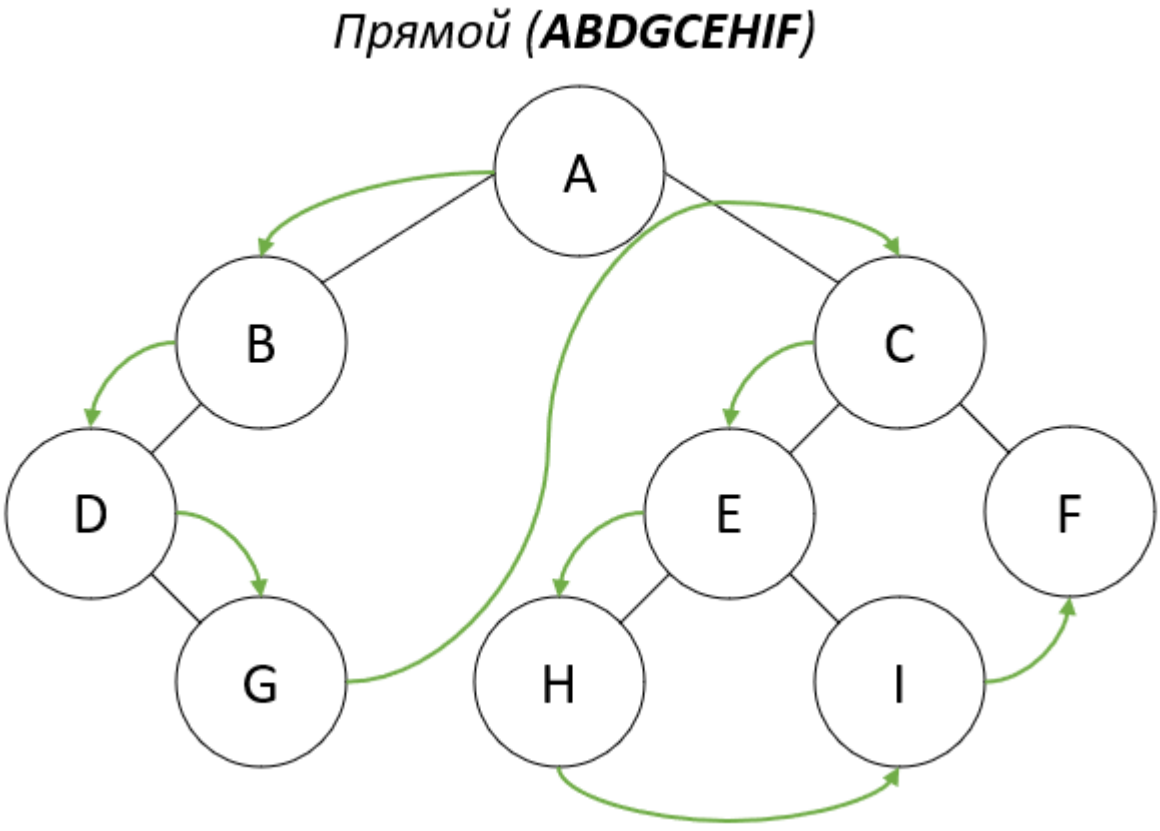
Эта рекуррентная зависимость рекурсивно вычисляет высоту дерева, начиная с корневой вершины и продолжая до листовых вершин. Результатом будет высота всего дерева.

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

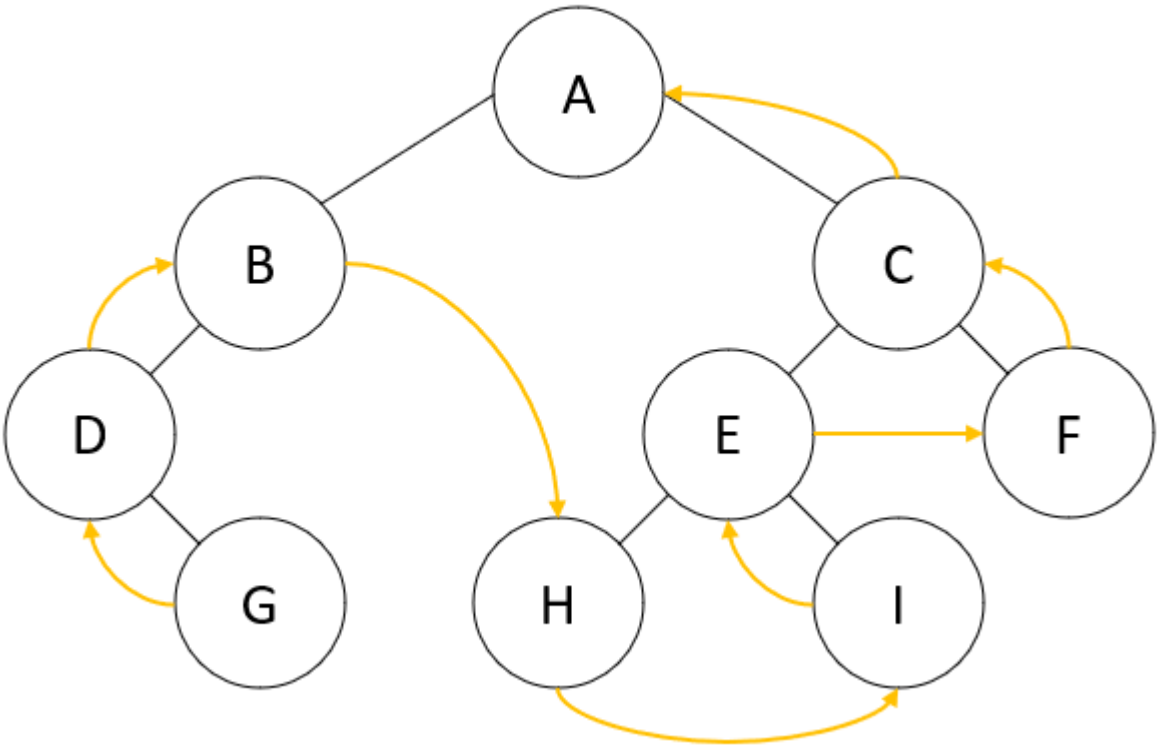
Индекс	*key*	*left*	*right*
1	12	7	3
2	15	8	NULL
3	4	10	NULL
4	10	5	9
5	2	NULL	NULL
6	**18**	**1**	**4**
7	7	NULL	NULL
8	14	6	2
9	21	NULL	NULL
10	5	NULL	NULL



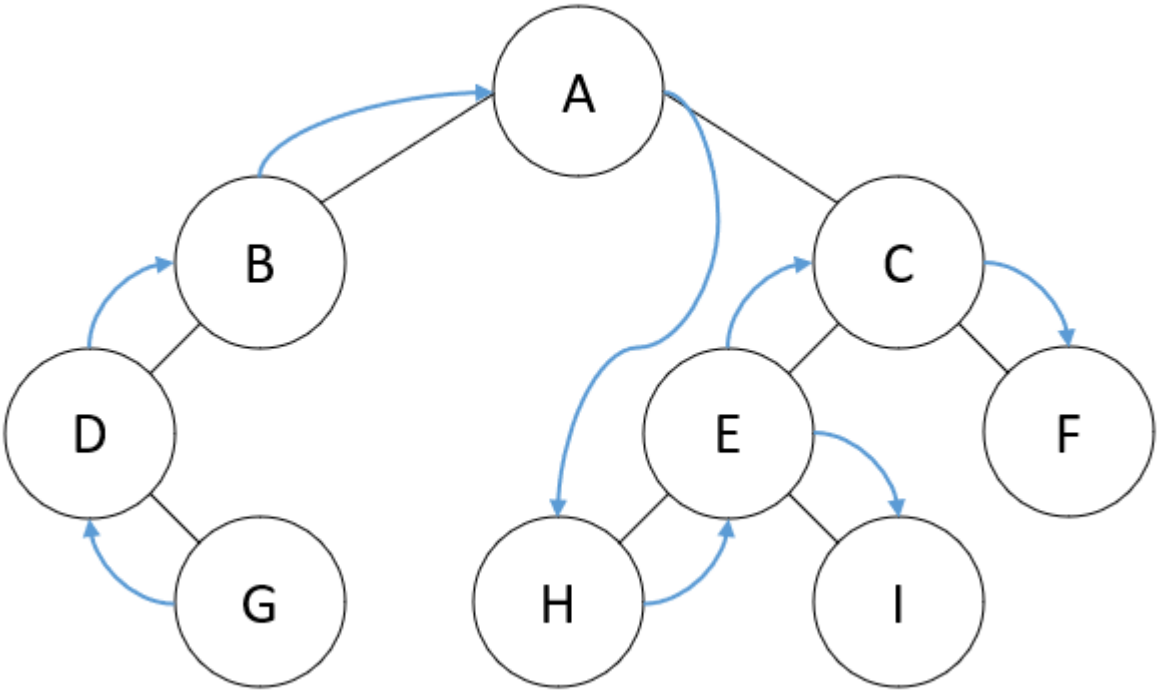
2.1.11. Укажите путь обхода дерева по алгоритмам: прямой, обратный, симметричный



*Обратный (**GDBHIEFCA**)*



*Симметричный (**GDBAHEICF**)*



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

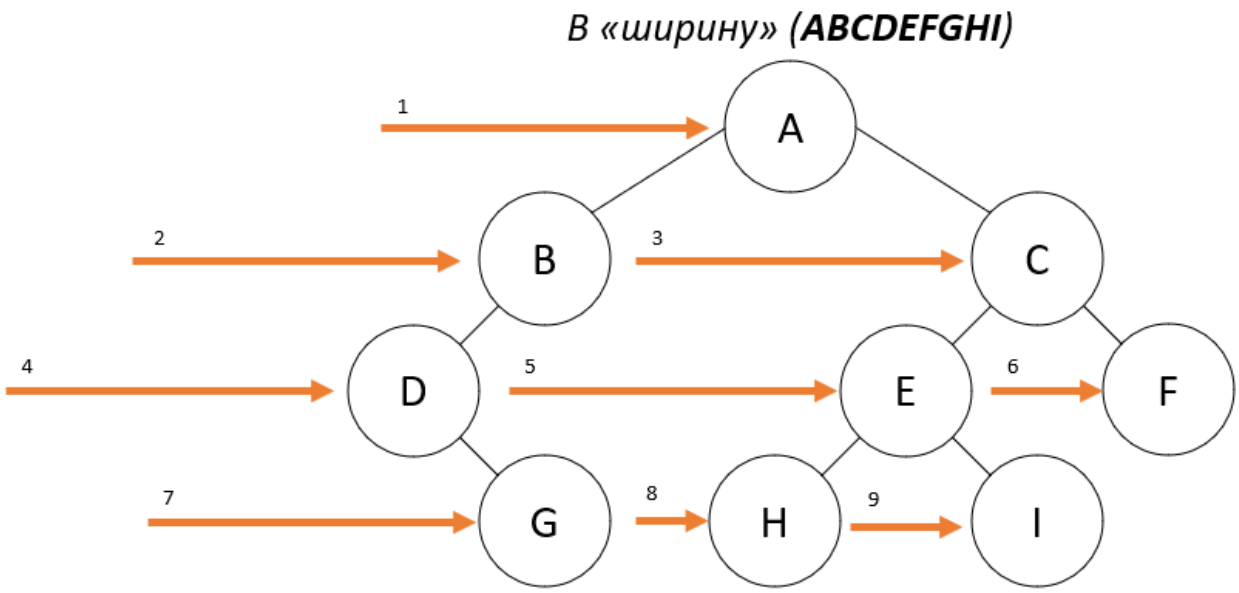
В алгоритме обхода дерева методом в "ширину" (Breadth-First Search, BFS) для хранения вершин, которые ожидают обработки, обычно используется структура данных, называемая ***очередью*** (queue). Очередь работает по принципу "первым пришел - первым ушел" (FIFO - First-In-First-Out). Это означает, что вершины 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.

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



Шаг	*Из очереди*	*В очередь*	*Вывод*
1		A	
2	A	B C	A
3	B	C D	B
4	C	D E F	C
5	D	E F G	D
6	E	F G H I	E
7	F	G H I	F
8	G	H I	G
9	H	I	H
10	I		I

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

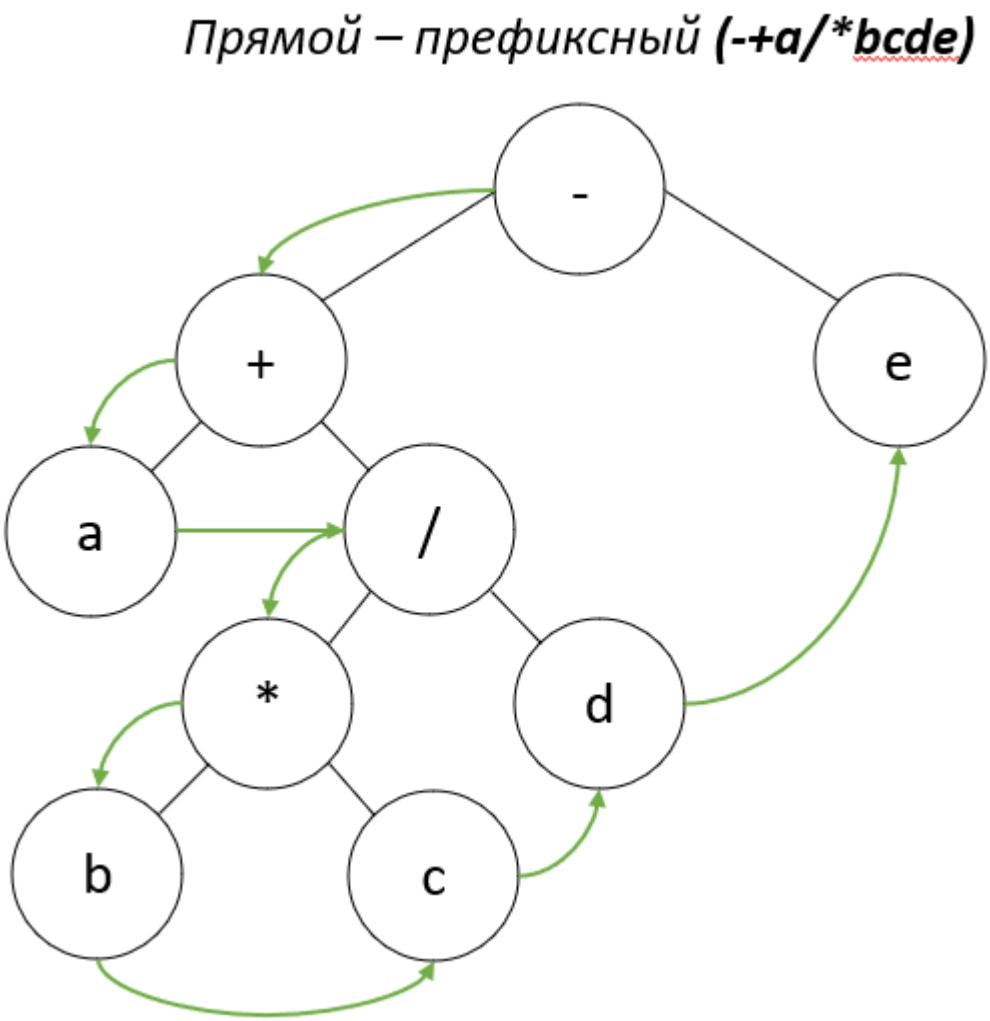
В не-рекурсивном обходе дерева методом в "глубину" (Depth-First Search, DFS), часто используется структура данных, называемая ***стеком*** (stack), для выполнения обхода вершин. Стек работает по принципу "последним пришел - первым ушел" (LIFO - Last-In-First-Out), что означает, что вершины добавляются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

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

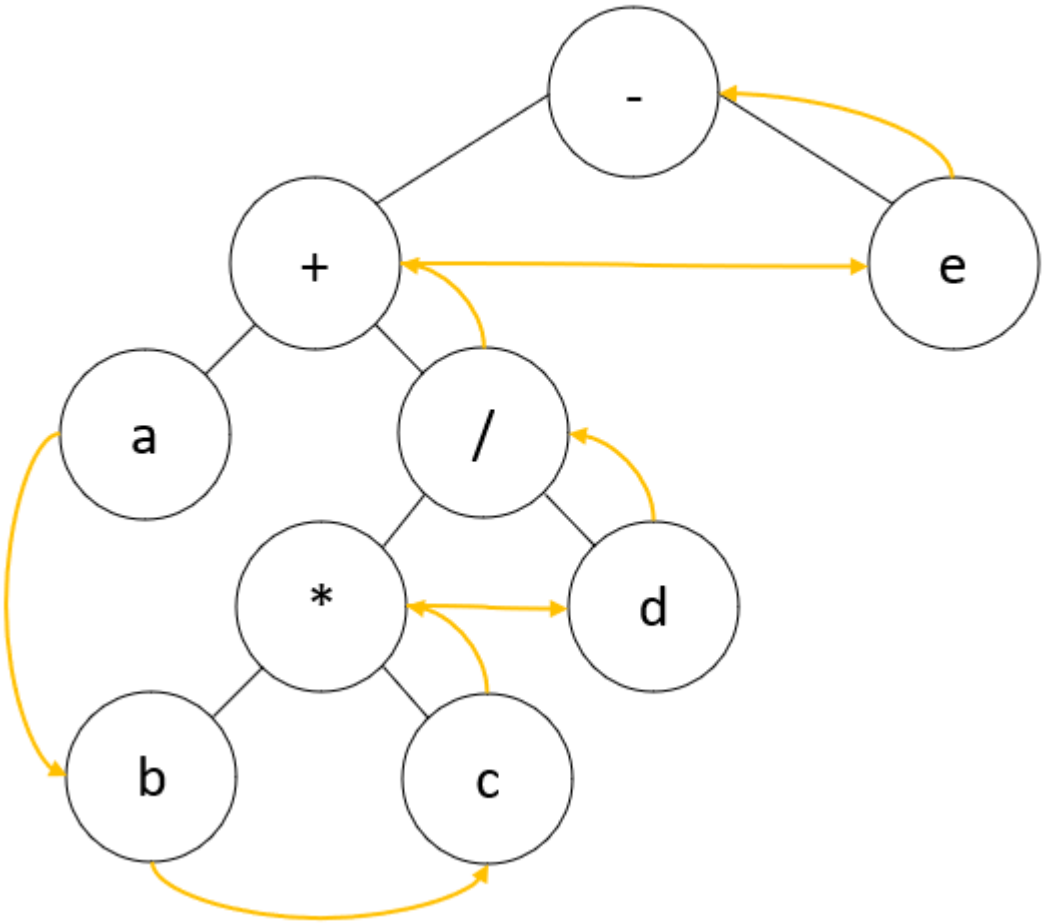
- Начинаем с корневой вершины дерева и помещаем ее в стек.
- Затем выполняем следующие шаги в цикле, пока стек не опустеет:
    - Извлекаем вершину из вершины стека.
    - Обрабатываем эту вершину (например, выводим ее значение или выполняем другие операции).
    - Помещаем в стек всех непосещенных потомков 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.

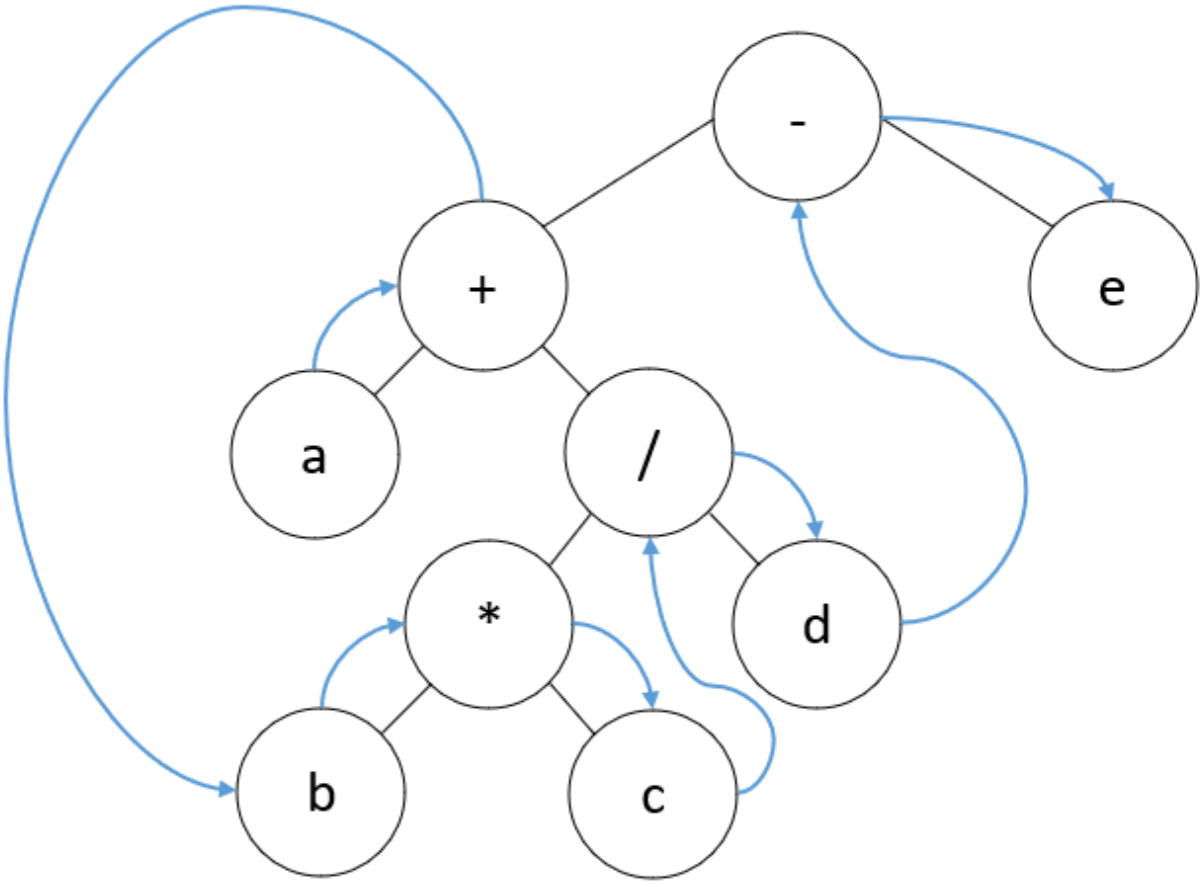
2.1.15. Выполните прямой, симметричный, обратный методы обхода дерева выражений



Обратный – постфиксный ***(abc*d/+e-)***

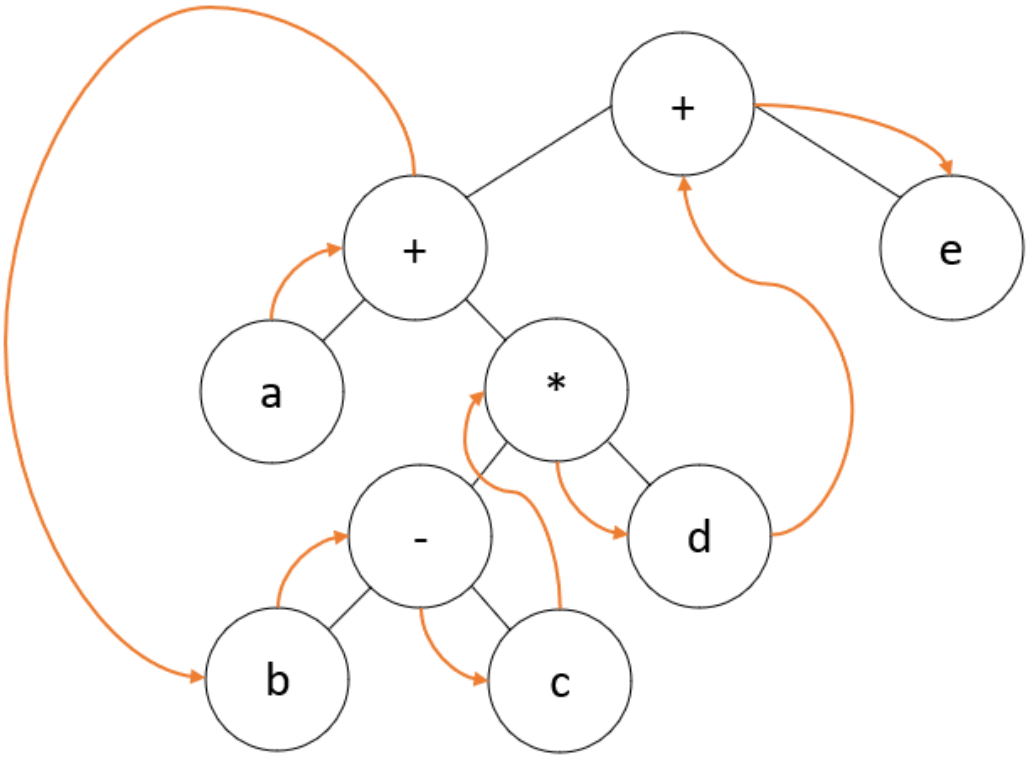


Обратный – инфиксный ***(a+b*c/d-e)***

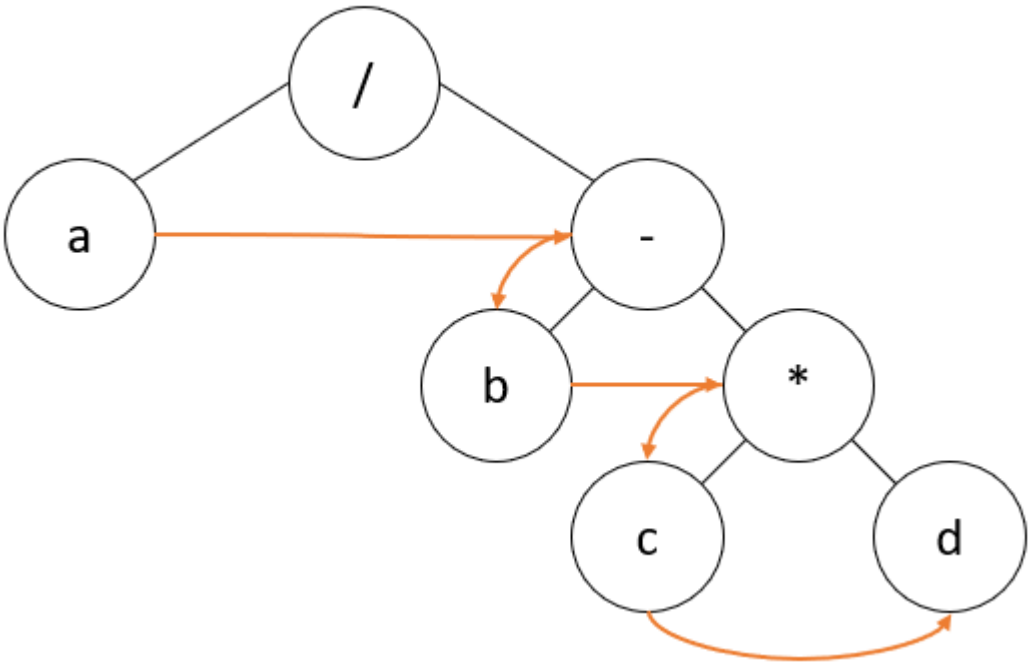


2.1.16. Для каждого заданного арифметического выражения постройте бинарное дерево выражений

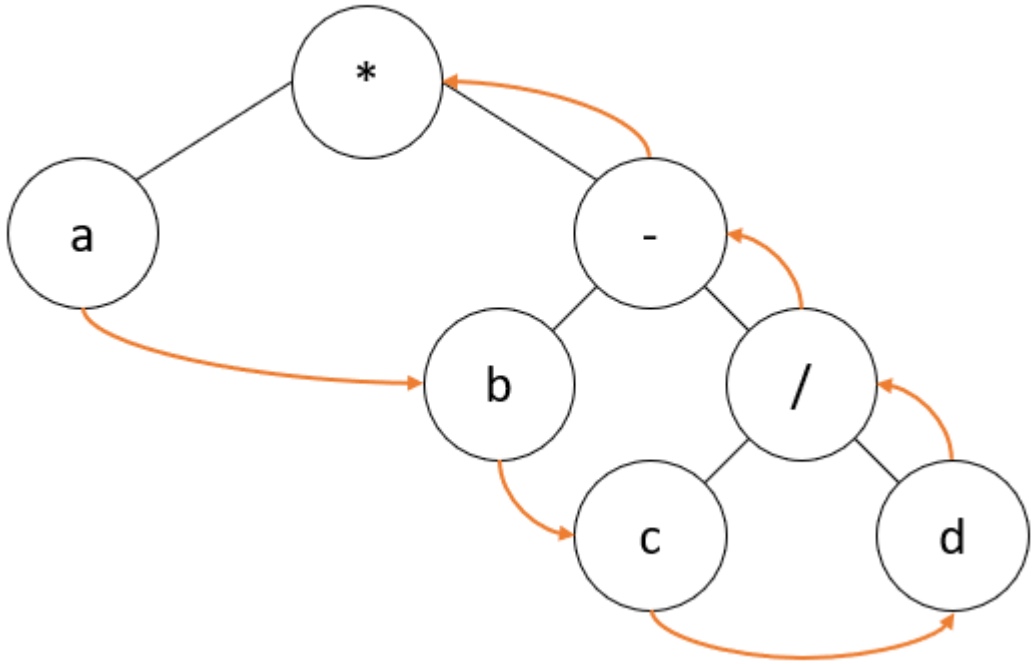
Инфиксное выражение (a+b-c*d+e)



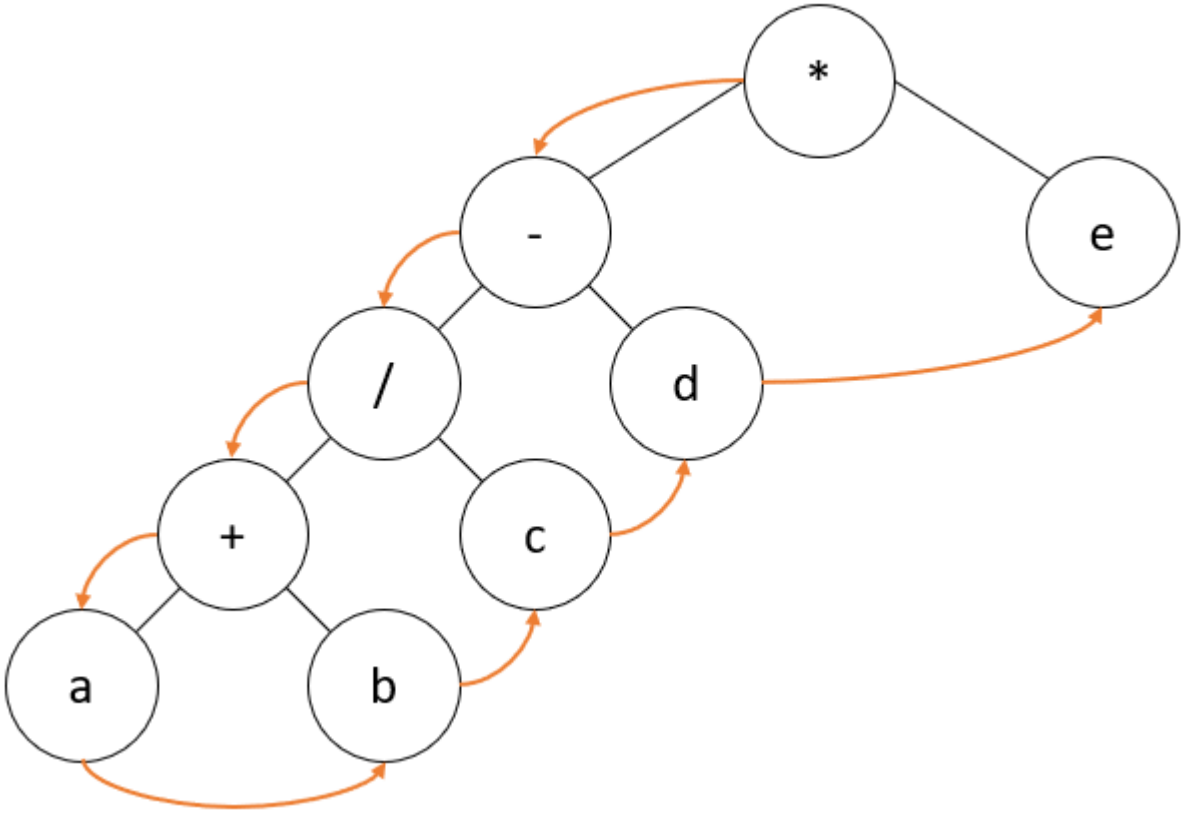
Префиксное выражение (/a-b*cd)



Постфиксное выражение (abcd/-*)



Префиксное выражение (*-/+abcde)



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

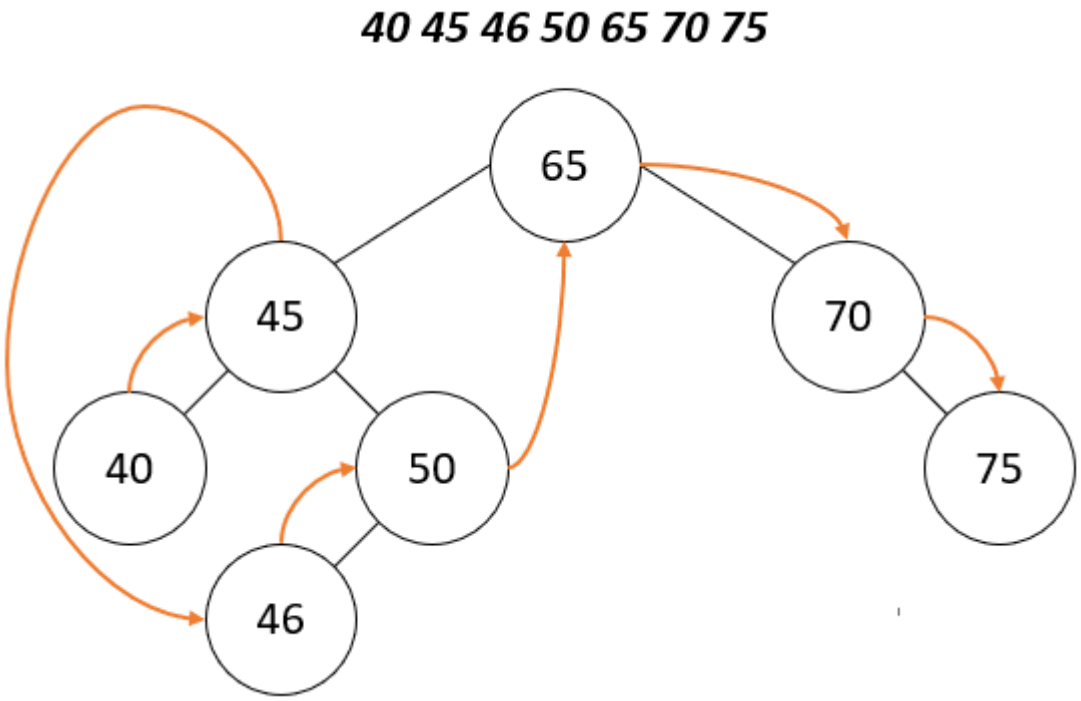
Если алгоритм обхода в ширину будет запоминать узлы не в очереди, а в стеке, то порядок обхода дерева изменится. Вместо классического порядка по уровням, при котором ближайшие узлы к корню обрабатываются раньше, узлы будут обрабатываться в обратном порядке. Это произойдет из-за того, что стек работает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

Порядок обхода в ширину с использованием стека будет следующим:

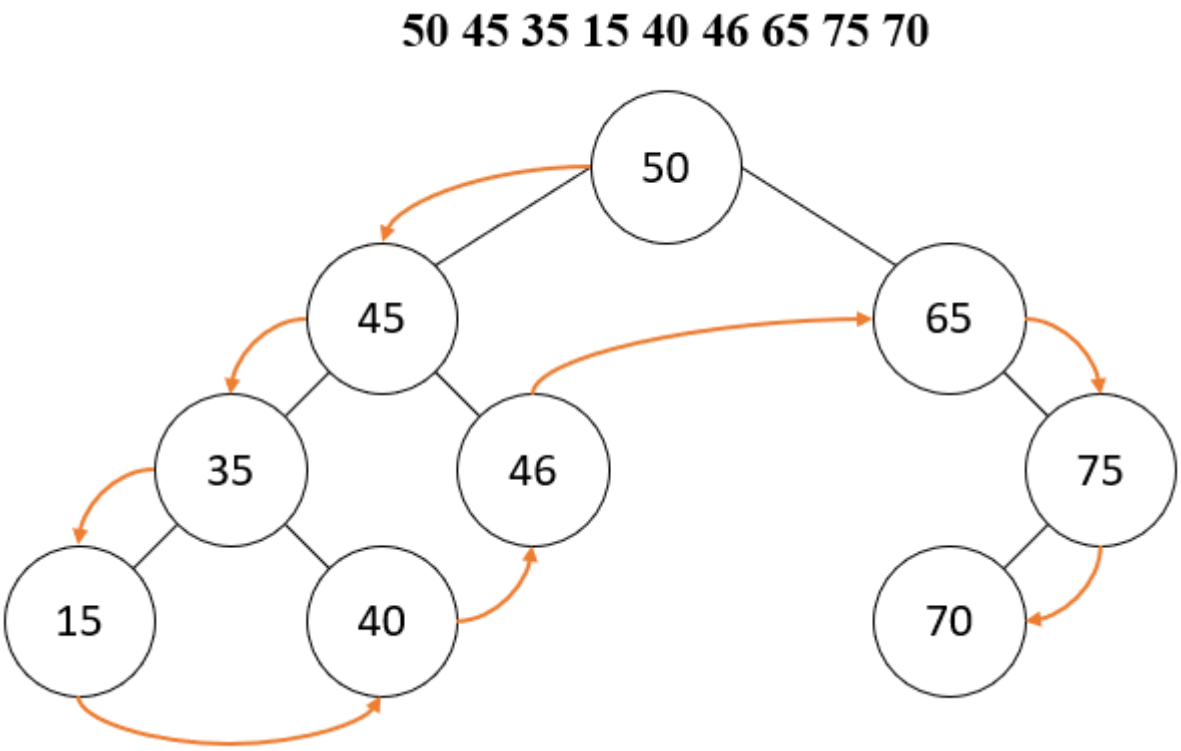
- Начнем с корневой вершины и добавим ее в стек.
- Затем извлечем вершину из вершины стека, обработаем ее и добавим в стек ее потомков (сначала правого, затем левого).
- Повторяем шаг 2 для вершины, находящейся на вершине стека.
- Продолжаем извлекать вершины из вершины стека и добавлять их потомков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

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

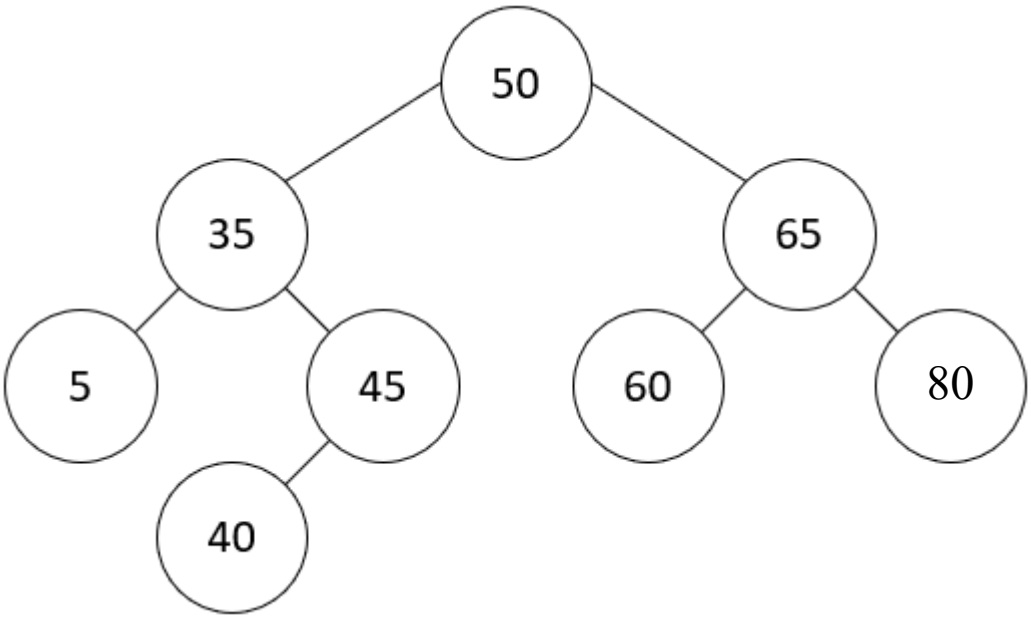
2.1.18. Постройте бинарное дерево поиска, которое в результате симметричного обхода дало бы следующую последовательность узлов: 40 45 46 50 65 70 75



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



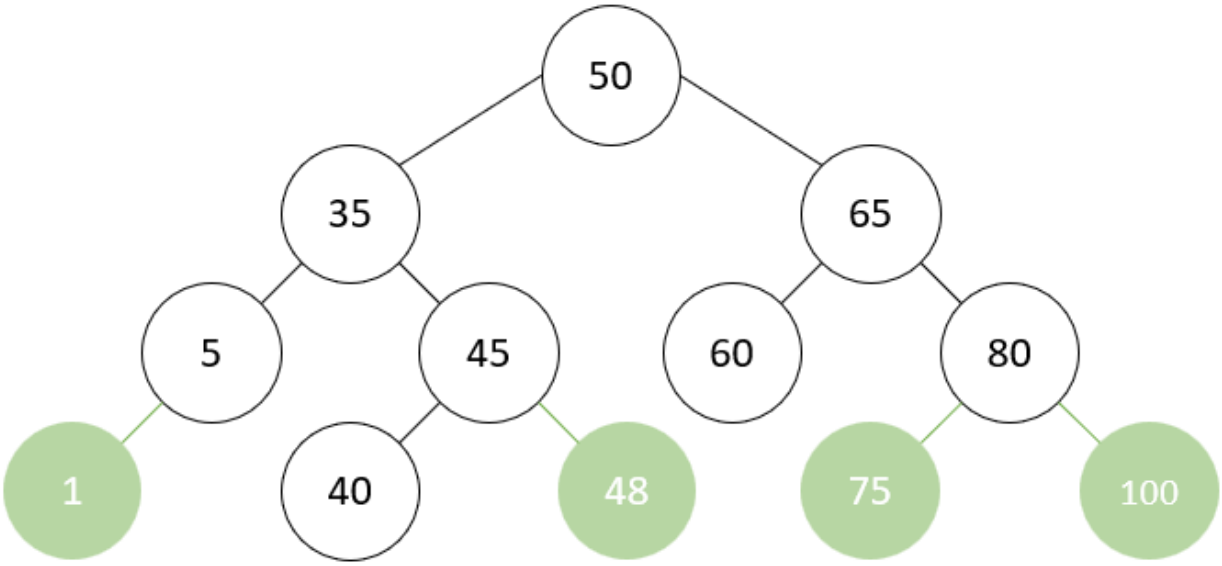
2.1.20. Дано следующее бинарное дерево поиска



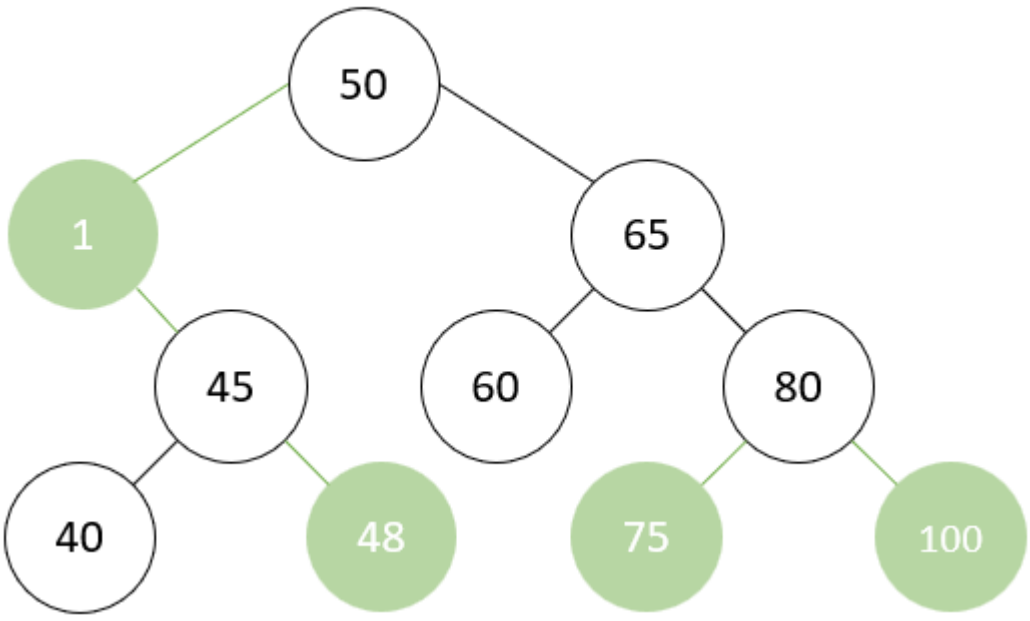
Покажите дерево:

- 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

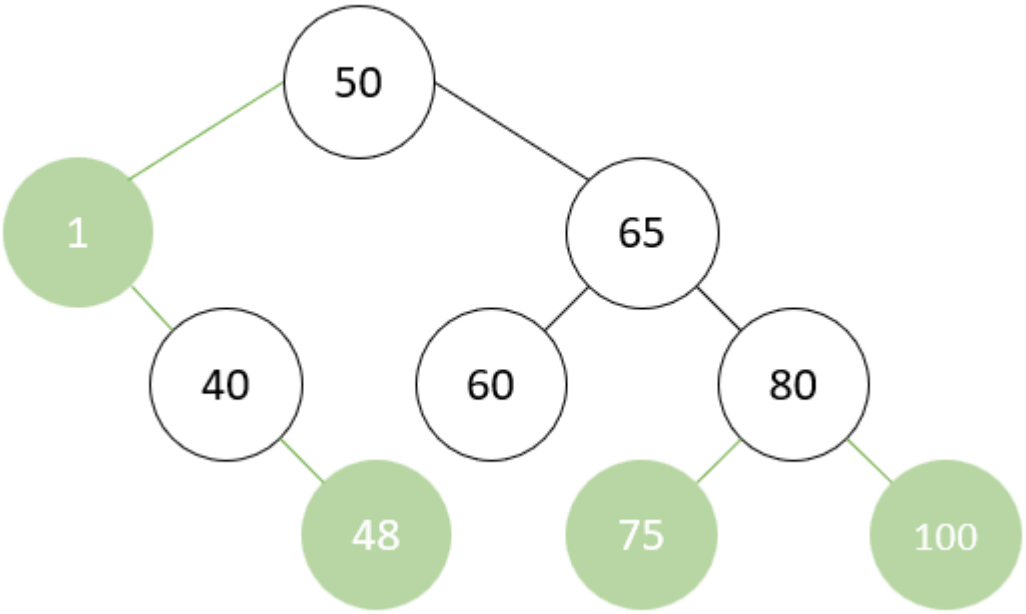
после включения узлов 1, 48, 75, 100



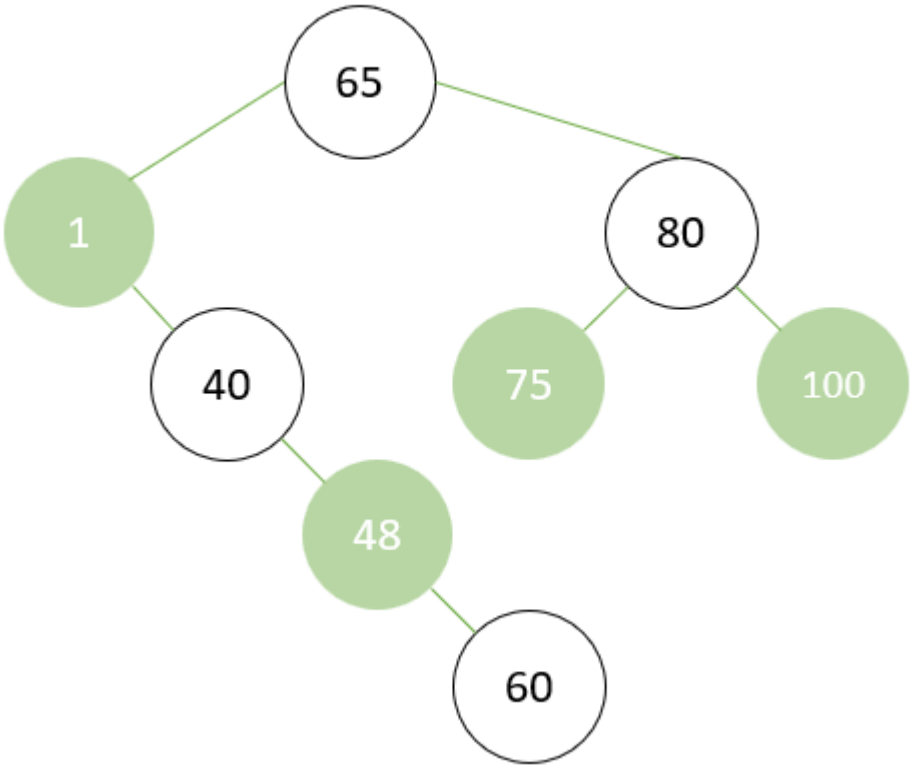
после удаления узлов 5, 35



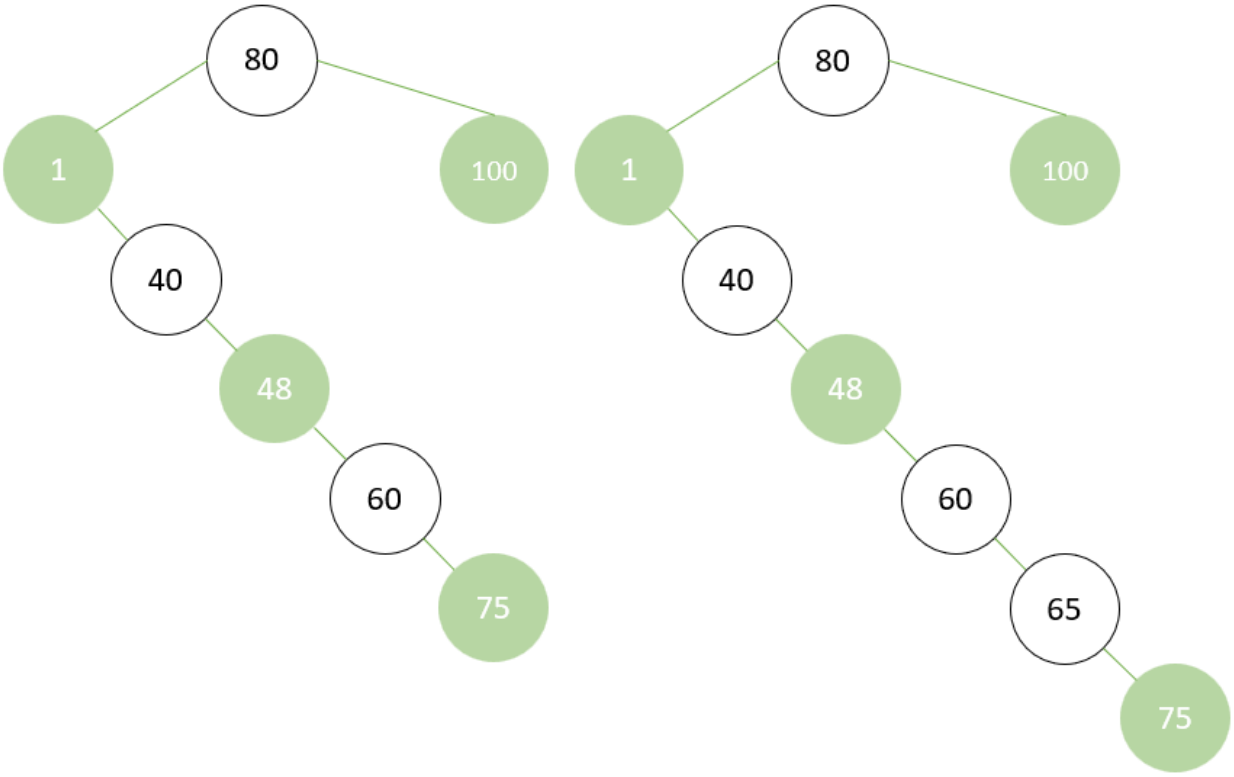
после удаления узла 45



после удаления узла 50



после удаления узла 65 и вставки его снова



2.2.Алгоритмы функций

Листинг 1. Структура для узла дерева

```
struct Node
{
    char data;          //----> Информационная часть узла (символьное значение)
    Node* left;         //----> Указатель на левое поддерево
    Node* right;        //----> Указатель на правое поддерево

    Node(char value) : data(value), left(nullptr), right(nullptr) {}
};
```

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

- Это информационная часть узла, которая содержит символьное значение (в данном случае, символ типа ***char***).
- 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 символьное значение и указатели на его потомков (левого и правого).

Листинг 2. Функция для построения идеально сбалансированного бинарного дерева

```
Node* CreateTree(int n)
{
    if (n <= 0)
    {
        return nullptr;
    }

    int middle = n / 2; //----> Поиск среднего элемента
    char data;
    cout << "   Введите символьное значение для узла " << middle + 1 << ": ";
    cin >> data;

    Node* root = new Node(data);                    //----> Создание корневого
узла
    root->left = CreateTree(middle);                //----> Рекурсивное построение
левого поддерева
```

```
    root->right = CreateTree(n - middle - 1);       //----> Рекурсивное построение
правого поддерева

    return root;
}
```

Данный код представляет собой функцию, которая используется для построения идеально сбалансированного бинарного дерева из ***n*** узлов. 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 ***middle*** для разделения ***n*** на две равные части. Это поможет создать идеально сбалансированное дерево.
- Объявление переменной для хранения символьного значения узла.
- Создается новый узел с информационной частью, равной введенному символу, и устанавливается как корень дерева.
- Рекурсивно вызывается функция, чтобы построить левое поддерево. Средний элемент передается в качестве аргумента, чтобы создать идеально сбалансированное дерево.
- Рекурсивно вызывается функция, чтобы построить правое поддерево.

Эта функция позволяет создать идеально сбалансированное бинарное дерево из ***n*** узлов, где пользователь вводит символьные значения для каждого узла, начиная с корня, и дерево автоматически разбивается на левое и правое поддерево согласно среднему элементу.

Листинг 3. Функция для вывода дерева

```
void PrintTree(Node* root, const string& prefix = "", bool left = true)
{
    if (root == nullptr)
    {
        return;
    }

    if (root->right)
    {
        PrintTree(root->right, prefix + (left ? "|   " : "    "), false);
    }

    cout << prefix;
```

```
    cout << (left ? "|-->" : "|-->");
    cout << root->data << endl;

    if (root->left)
    {
        PrintTree(root->left, prefix + (left ? "    " : "|   "), true);
    }
}
```

Этот алгоритм предназначен для печати бинарного дерева на экране в виде древовидной структуры. Он использует рекурсивный подход для обхода дерева и вывода узлов дерева, учитывая их отношения к родительским узлам (левый или правый). Алгоритм работает следующим образом:

- Сначала он проверяет, является ли корневой узел равным ***nullptr***. Если да, то это означает, что дерево пустое, и алгоритм завершает выполнение.
- Если у корневого узла есть правый потомок, алгоритм вызывает себя рекурсивно для правого поддерева, при этом реализуется форматированный вывод.
- Затем алгоритм выводит текущий узел с префиксом. Это позволяет отобразить отношения узлов в дереве.
- Если у корневого узла есть левый потомок, алгоритм вызывает себя рекурсивно для левого поддерева.

Этот алгоритм обеспечивает красивое и информативное представление бинарного дерева на экране, где узлы и их отношения видны в виде древовидной структуры.

Листинг 4. Функция удаления дерева после использования

```
void DeleteTree(Node* root)
{
    if (root == nullptr)
    {
        return;
    }

    //----> Рекурсивное удаление левого и правого поддерева
    DeleteTree(root->left);
    DeleteTree(root->right);

    //----> Удаление текущего узла
    delete root;
}
```

Этот алгоритм предназначен для удаления всех узлов в бинарном дереве и освобождения памяти, выделенной для каждого узла дерева. Алгоритм работает рекурсивно и имеет следующую логику:

- Сначала алгоритм проверяет, существует ли корневой узел. Если корневой узел является нулевым указателем, это означает, что дерево пустое, и алгоритм завершает выполнение.
- Если корневой узел не является нулевым указателем, алгоритм вызывает себя рекурсивно для левого поддерева и для правого поддерева. Это приводит к удалению всех узлов в левом и правом поддеревьях.
- После удаления всех узлов в поддеревьях, алгоритм освобождает память, выделенную для текущего узла, используя оператор ***delete***. Это приводит к удалению текущего узла.
- Этот процесс рекурсивно повторяется для каждого узла дерева, начиная с корневого узла и двигаясь вглубь дерева. В результате всех вызовов, все узлы дерева удаляются, и память, выделенная под них, освобождается, что предотвращает утечки памяти.

Этот алгоритм полезен, когда вам нужно удалить бинарное дерево после его использования, чтобы избежать утечек памяти и корректно освободить ресурсы.

Листинг 5. Функция нахождения максимального значения среди значений листьев дерева

```
char FindMax(Node* root)
{
    if (root == nullptr)
    {
        //----> Дерево пустое
        return '\0';
    }

    if (root->left == nullptr && root->right == nullptr)
    {
        //----> Листовой узел
        if (isdigit(root->data))
        {
            return root->data;
        }
        return '\0';    //----> Если это не цифра
    }

    //----> Рекурсивный поиск максимального значения среди листьев в левом и
правом поддеревьях
```

```
    char leftMax = '\0';
    char rightMax = '\0';

    if (root->left)
    {
        leftMax = FindMax(root->left);
    }

    if (root->right)
    {
        rightMax = FindMax(root->right);
    }

    //----> Сравнение максимального значения в левом и правом поддеревьях
    return max(leftMax, rightMax);
}
```

Этот алгоритм предназначен для поиска максимального значения среди листьев в бинарном дереве. Листья — это узлы дерева, которые не имеют ни левого, ни правого потомка. Алгоритм работает рекурсивно и имеет следующую логику:

- Сначала алгоритм проверяет, существует ли корневой узел. Если корневой узел является нулевым указателем, это означает, что дерево пустое, и алгоритм возвращает символ 0, чтобы указать, что в пустом дереве нет максимального значения.
- Если корневой узел не является нулевым указателем, алгоритм проверяет, является ли этот узел листовым узлом, проверяя, есть ли у него и левый потомок и правый потомок. Если узел является листом и его данные являются цифрой, то этот узел считается потенциальным максимальным значением.
- Если текущий узел не является листовым, алгоритм рекурсивно вызывает себя для левого поддерева и правого поддерева, и сохраняет результаты поиска максимального значения в переменных.
- Затем алгоритм сравнивает максимальные значения, найденные в левом и правом поддеревьях с использованием функции ***max***. Это позволяет выбрать более большее из двух значений.
- В итоге, алгоритм возвращает максимальное значение, найденное среди листьев во всем дереве.

Этот алгоритм полезен, когда вам нужно найти максимальное значение среди листьев в бинарном дереве, например, когда дерево представляет числовые данные, и вы хотите найти наибольшее число в нем.

Листинг 6. Функция определения уровня, на котором находится заданное значение

```
int FindLevel(Node* root, char value, int level = 1)
{
    if (root == nullptr)
    {
        return 0;   //----> Значение не найдено
    }

    if (root->data == value)
    {
        return level;   //----> Значение найдено на текущем уровне
    }

    int leftLevel = FindLevel(root->left, value, level + 1);
    if (leftLevel != 0)
    {
        return leftLevel;   //----> Значение найдено в левом поддереве
    }

    int rightLevel = FindLevel(root->right, value, level + 1);
    return rightLevel;  //----> Значение найдено в правом поддереве (или не
найдено вообще)
}
```

Этот алгоритм предназначен для поиска уровня (глубины) узла с определенным значением в бинарном дереве. Уровень узла — это количество родительских узлов, которые нужно пройти от корневого узла, чтобы достичь данного узла. Алгоритм работает рекурсивно и имеет следующую логику:

- Сначала алгоритм проверяет, существует ли корневой узел. Если корневой узел является нулевым указателем, это означает, что значение не найдено в дереве, и алгоритм возвращает 0, чтобы указать, что значение не найдено.
- Затем алгоритм проверяет, равно ли проверяемое значение значению элемента дерева. Если равно, это означает, что значение найдено на текущем уровне, и алгоритм возвращает текущий уровень.
- Если значение не равно, то алгоритм рекурсивно вызывает себя для левого поддерева и увеличивает уровень на 1.

- Затем алгоритм проверяет значение текущего уровня. Если не равно 0, это означает, что значение было найдено в левом поддереве, и алгоритм возвращает его.
- Если значение не было найдено в левом поддереве, алгоритм рекурсивно вызывает себя для правого поддерева с увеличенным уровнем и результат сохраняется в переменной.
- В конечном итоге, алгоритм возвращает значение текущего уровня. В противном случае алгоритм возвращает 0, указывая, что значение не найдено в дереве.

Этот алгоритм полезен, когда вам нужно найти уровень (глубину) узла с определенным значением в бинарном дереве, чтобы узнать, на какой глубине находится этот узел от корня дерева.

Листинг 7. Функция определения количества цифр в правом поддереве исходного дерева

```
int CountRight(Node* root)
{
    if (root == nullptr)
    {
        return 0;
    }

    int count = 0;

    //----> Если это цифра
    if (isdigit(root->data))
    {
        count++;
    }

    //----> Рекурсивный счет цифр в правом и левом поддеревьях
    count += CountRight(root->right);
    count += CountRight(root->left);

    return count;
}
```

Этот алгоритм предназначен для подсчета количества цифр, находящихся в правом и левом поддеревьях бинарного дерева, начиная с заданного корневого узла. Алгоритм работает рекурсивно и имеет следующую логику:

- Сначала алгоритм проверяет, существует ли корневой узел. Если корневой узел является нулевым указателем, это означает, что дерево

пустое, и алгоритм возвращает 0, чтобы указать, что в пустом дереве нет цифр.

- Затем алгоритм инициализирует счетчик равным 0, который будет использоваться для подсчета цифр в дереве.
- Если значение является цифрой, алгоритм увеличивает счетчик на 1. Таким образом, если корневой узел содержит цифру, она учитывается в общем подсчете.
- Алгоритм рекурсивно вызывает себя для правого поддерева. Результаты этих вызовов добавляются к счетчику. Таким образом, алгоритм проходит через каждый узел дерева и подсчитывает количество цифр.
- В конечном итоге, алгоритм возвращает общее количество цифр, найденных в дереве.

Этот алгоритм полезен, когда вам нужно подсчитать количество цифр в бинарном дереве, и вы хотите выполнить этот подсчет рекурсивно, начиная с корневого узла дерева.

2.3.Код программы

Листинг 8

```
#include <iostream>
#include <iomanip>

using namespace std;




//---------------- Общая часть ----------------//
//
//----> Структура для узла дерева
struct Node
{
    char data;          //----> Информационная часть узла (символьное значение)
    Node* left;         //----> Указатель на левое поддерево
    Node* right;        //----> Указатель на правое поддерево

    Node(char value) : data(value), left(nullptr), right(nullptr) {}
};


//----> Функция для построения идеально сбалансированного бинарного дерева
Node* CreateTree(int n)
{
    if (n <= 0)
    {
        return nullptr;
    }

    int middle = n / 2; //----> Поиск среднего элемента
    char data;
    cout << "   Введите символьное значение для узла " << middle + 1 << ": ";
    cin >> data;

    Node* root = new Node(data);                    //----> Создание корневого
узла
    root->left = CreateTree(middle);                //----> Рекурсивное построение
левого поддерева
    root->right = CreateTree(n - middle - 1);       //----> Рекурсивное построение
правого поддерева

    return root;
}


//----> Функция для вывода дерева
void PrintTree(Node* root, const string& prefix = "", bool left = true)
{
    if (root == nullptr)
    {
        return;
    }

    if (root->right)
    {
        PrintTree(root->right, prefix + (left ? "|    " : "     "), false);
    }

    cout << prefix;
    cout << (left ? "|-->" : "|-->");
```

```
    cout << root->data << endl;

    if (root->left)
    {
        PrintTree(root->left, prefix + (left ? "    " : "|   "), true);
    }
}


//----> Функция удаления дерева после использования
void DeleteTree(Node* root)
{
    if (root == nullptr)
    {
        return;
    }

    //----> Рекурсивное удаление левого и правого поддерева
    DeleteTree(root->left);
    DeleteTree(root->right);

    //----> Удаление текущего узла
    delete root;
}
//
//---------------- Общая часть ----------------//




//---------------- Часть варианта ----------------//
//
//----> Функция нахождения максимального значения среди значений листьев дерева
char FindMax(Node* root)
{
    if (root == nullptr)
    {
        //----> Дерево пустое
        return '\0';
    }

    if (root->left == nullptr && root->right == nullptr)
    {
        //----> Листовой узел
        if (isdigit(root->data))
        {
            return root->data;
        }
        return '\0';    //----> Если это не цифра
    }

    //----> Рекурсивный поиск максимального значения среди листьев в левом и
правом поддеревьях
    char leftMax = '\0';
    char rightMax = '\0';

    if (root->left)
    {
        leftMax = FindMax(root->left);
    }

    if (root->right)
    {
        rightMax = FindMax(root->right);
    }
```

```
    //----> Сравнение максимального значения в левом и правом поддеревьях
    return max(leftMax, rightMax);
}



//----> Функция определения уровня, на котором находится заданное значение
int FindLevel(Node* root, char value, int level = 1)
{
    if (root == nullptr)
    {
        return 0;   //----> Значение не найдено
    }

    if (root->data == value)
    {
        return level;   //----> Значение найдено на текущем уровне
    }

    int leftLevel = FindLevel(root->left, value, level + 1);
    if (leftLevel != 0)
    {
        return leftLevel;   //----> Значение найдено в левом поддереве
    }

    int rightLevel = FindLevel(root->right, value, level + 1);
    return rightLevel;  //----> Значение найдено в правом поддереве (или не
найдено вообще)
}


//----> Функция определения количества цифр в правом поддереве исходного дерева
int CountRight(Node* root)
{
    if (root == nullptr)
    {
        return 0;
    }

    int count = 0;

    //----> Если это цифра
    if (isdigit(root->data))
    {
        count++;
    }

    //----> Рекурсивный счет цифр в правом и левом поддеревьях
    count += CountRight(root->right);
    count += CountRight(root->left);

    return count;
}
//
//---------------- Часть варианта ----------------//



int main()
{
    setlocale(LC_ALL, "ru");

    int n;
```

```
    cout << endl << "Введите количество узлов в дереве: ";
    cin >> n;
    cout << endl;

    Node* root = CreateTree(n);

    cout << endl << "Идеально сбалансированное бинарное дерево ~_~" << endl;
    PrintTree(root);


    bool flag = true;

    int level = 0;
    char maxNode = '0';
    int digitCount = 0;
    while (flag)
    {
        int choice;
        cout << endl << "Выберите действие: " << endl << "   1 - найти
максимальное значение среди значений листьев дерева" << endl << "   2 - определить
уровень, на котором находится заданное значение" << endl << "   3 - определить
количества цифр в правом поддереве исходного дерева" << endl << "   4 - выйти из
системы" << endl;
        cin >> choice;
        switch (choice)
        {
        case 1:
            maxNode = FindMax(root);
            if (maxNode == '\0')
            {
                cout << endl << "Цифр нет для сравнения" << endl;
            }
            else
            {
                cout << endl << "Максимальное значение среди листьев дерева: " <<
maxNode << endl;
            }
            break;
        case 2:
            char target;
            cout << endl << "Введите значение для поиска уровня: ";
            cin >> target;

            level = FindLevel(root, target);
            if (level > 0)
            {
                cout << "   Уровень, на котором находится значение " << target <<
": " << level << endl;
            }
            else
            {
                cout << "   Значение " << target << " не найдено в дереве U_U" <<
endl;
            }
            break;
        case 3:
            digitCount = CountRight(root->right);
            cout << endl << "Количество цифр в правом поддереве: " << digitCount
<< endl;
            break;
        case 4:
            flag = false;
            break;
        default:
            return 0;
```

```
            break;
        }
    }

    DeleteTree(root);

    return 0;
}
```

2.4. Тестирование

```
Введите количество узлов в дереве: 9

   Введите символьное значение для узла 5: h
   Введите символьное значение для узла 3: 7
   Введите символьное значение для узла 2: 3
   Введите символьное значение для узла 1: k
   Введите символьное значение для узла 1: 9
   Введите символьное значение для узла 3: w
   Введите символьное значение для узла 2: 3
   Введите символьное значение для узла 1: 5
   Введите символьное значение для узла 1: 1

Идеально сбалансированное бинарное дерево ~_~
|        |-->1
|    |-->w
|    |    |-->3
|    |        |-->5
|-->h
     |    |-->9
     |-->7
          |-->3
              |-->k
Выберите действие:
   1 - найти максимальное значение среди значений листьев дерева
   2 - определить уровень, на котором находится заданное значение
   3 - определить количества цифр в правом поддереве исходного дерева
   4 - выйти из системы
1

Максимальное значение среди листьев дерева: 9
Выберите действие:
   1 - найти максимальное значение среди значений листьев дерева
   2 - определить уровень, на котором находится заданное значение
   3 - определить количества цифр в правом поддереве исходного дерева
   4 - выйти из системы
2

Введите значение для поиска уровня: k
   Уровень, на котором находится значение k: 4
```

```
Выберите действие:
   1 - найти максимальное значение среди значений листьев дерева
   2 - определить уровень, на котором находится заданное значение
   3 - определить количества цифр в правом поддереве исходного дерева
   4 - выйти из системы
3

Количество цифр в правом поддереве: 3
```

3. 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.