МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА № 42

ОТЧЁТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

| старший преподаватель |  |  |  | С.Ю. Гуков |
| --- | --- | --- | --- | --- |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

| ОТЧЁТ О ЛАБОРАТОРНОЙ РАБОТЕ №8 |
| --- |
| Обходы двоичного дерева |
| по курсу: |
| АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ |
|  |

РАБОТУ ВЫПОЛНИЛ

| СТУДЕНТ гр. № | 4329 |  |  |  | Д.С. Шаповалова |
| --- | --- | --- | --- | --- | --- |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

ОГЛАВЛЕНИЕ

[Цель работы 3](#_30j0zll)

[Постановка задачи 3](#_1fob9te)

[Схема алгоритма решения 4](#_3znysh7)

[Полное описание реализованной функции 4](#_gqc1nhfmq022)

[Листинг программы 5](#_2et92p0)

[Результат выполнения программы. 5](#_tyjcwt)

[ВЫВОДЫ 6](#_3dy6vkm)

# Цель работы

Построить in-order, pre-order и post-order обходы данного двоичного дерева.

# Постановка задачи

Задание: Написать программу, строящую in-order, pre-order и post-order обходы данного двоичного дерева. Текст задания приведён в таблице 1.

Таблица 1. Индивидуальное задание

| № | Текст задания |
| --- | --- |
| 0 | **Формат входа:** Первая строка содержит число вершин n. Вершины дерева пронумерованы числами от 0 до n−1. Вершина 0 является корнем. Каждая из следующих n строк содержит информацию о вершинах 0, 1, ..., n−1: i-я строка задаёт числа key[i], left[i] и right[i], где key[i] – ключ вершины i, left[i] – индекс левого сына вершины i, а right[i] – индекс правого сына вершины i. Если у вершины i нет одного или обоих сыновей, соответствующее значение равно −1.  **Формат выхода:** Три строки: in-order, pre-order и post-order обходы.  **Ограничения:** 1 ≤ n ≤ 105; 0 ≤ keyi ≤ 109; −1 ≤ left[i], right[i] ≤ n−1.  Гарантируется, что вход задаёт корректное двоичное дерево: в частности, если left[i] != −1 и right[i] != −1, то left[i] != right[i]; никакая вершина не является сыном двух вершин; каждая вершина является потомком корня.  **Справка:**  *In-order обход (симметричный)* соответствует следующей рекурсивной процедуре, получающей на вход корень v текущего поддерева: произвести рекурсивный вызов для v.left, напечатать v.key, произвести рекурсивный вызов для v.right.  *Pre-order обход (прямой):* напечатать v.key, произвести рекурсивный вызов для v.left, произвести рекурсивный вызов для v.right.  *Post-order (обратный):* произвести рекурсивный вызов для v.left, произвести рекурсивный вызов для v.right, напечатать v.key. |

# Схема алгоритма решения

1. Ввод данных.
2. Запуск функций обхода
3. Вывод данных

# Полное описание реализованной функции

**Ввод данных:**

1 строка - число n - количество узлов

Со 2 по n+1 строку - 3 числа - значение узла, индекс левого потомка, индекс правого потомка + номер строки = индексу узла.

Создаём списки keys, lefts и rights, заполняя их данными из последующих строк ввода. Создаём три списка для хранения результатов обходов.

Вызывает функции in\_order, pre\_order и post\_order, передавая им корень дерева (индекс 0).

**Функция построения дерева:**

Выводим словарь из 3-х массивов: keys - значения узлов, lefts - индексы левых потомков, rights - индексы правых потомков.

**Функция in-order:**

Симметричный обход.

1. Проверяем есть ли у узла потомки (v != -1)
2. Рекурсивно вызываем для левого потомка(поддерева) и вызываем-вызываем пока не дойдём до узла, у которого нет потомков.
3. Когда доходим до самого левого нижнего (потомкистого) потомка, добавляем значение в результирующий массив.
4. Переходим к правому потомку, записываем его значение.
5. Переходим к узлу выше, записываем его значение.
6. Когда дошли до корня, записываем его значение и переходим к аналогичному обходу правого поддерева.

**Функция pre-order:**

Прямой обход.

1. Записываем значение узла (1-корень, дальше потомки).
2. Идём к левым потомкам, вниз, записывая каждое значение.
3. Потом записываем правых потомков, идя вверх.
4. Рассматриваем правое поддерево, аналогично.

**Функция post-order:**

Обратный обход.

1. Идём от корня к самому левому нижнему потомку
2. Когда до него дошли, смотрим на его левого потомка (его нет), на правого потомка (его нет), затем записываем значение текущего узла(самого нижнего левого)
3. Возвращаемся на шаг раньше в рекурсии. Смотрим на потомков правого потомка, если их нет, записываем его значение, иначе углубляемся и пока не дойдём до дна.
4. Когда вернулись снова до корня, идём к правому потомку корня и далее аналогично.

**Вывод данных:**

Выводим строки порядка значений обхода in-order, pre-order, post-order.

# Листинг программы

def build\_tree(keys, lefts, rights):

return {

'keys': keys,

'lefts': lefts,

'rights': rights

}

# симметричный

def in\_order(tree, v, result):

if v != -1: # проверяем есть ли у узла потомки

# идём от левого потомка к корню

# смотрим индекс левого потомка в массиве левых потомков по индексу текущего узла.

in\_order(tree, tree['lefts'][v], result)

# записываем значение узла

result.append(tree['keys'][v])

# идём от правого потомка к корню

in\_order(tree, tree['rights'][v], result)

# прямой

def pre\_order(tree, v, result):

if v != -1:

# записываем значение

result.append(tree['keys'][v])

# идём к левым потомкам

pre\_order(tree, tree['lefts'][v], result)

# идём к правым потомкам

pre\_order(tree, tree['rights'][v], result)

# обратный

def post\_order(tree, v, result):

if v != -1: # проверяем что у узла есть потомки,

# если нет, то он самый нижний, возвращаемся на шаг раньше в рекурсии

# идём к левым потомкам

post\_order(tree, tree['lefts'][v], result)

# идём к правым потомкам

post\_order(tree, tree['rights'][v], result)

# если он самый нижний, записываем его и идём наверх.

result.append(tree['keys'][v])

def main():

import sys

input = sys.stdin.read # читаем данные до конца ввода

data = input().splitlines() # делим на строки

n = int(data[0])

keys = []

lefts = []

rights = []

for i in range(1, n + 1):

key, left, right = map(int, data[i].split())

keys.append(key)

lefts.append(left)

rights.append(right)

tree = build\_tree(keys, lefts, rights)

in\_order\_result = []

pre\_order\_result = []

post\_order\_result = []

in\_order(tree, 0, in\_order\_result)

pre\_order(tree, 0, pre\_order\_result)

post\_order(tree, 0, post\_order\_result)

print(" ".join(map(str, in\_order\_result)))

print(" ".join(map(str, pre\_order\_result)))

print(" ".join(map(str, post\_order\_result)))

if \_\_name\_\_ == "\_\_main\_\_":

main()

# 

# Результат выполнения программы.

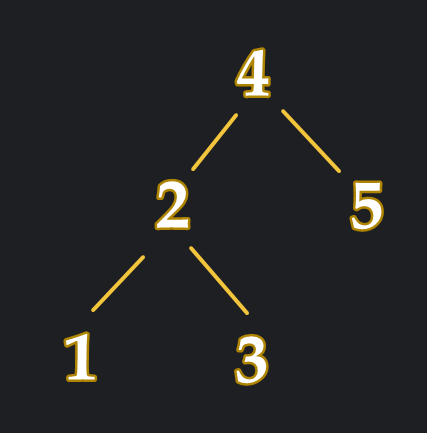
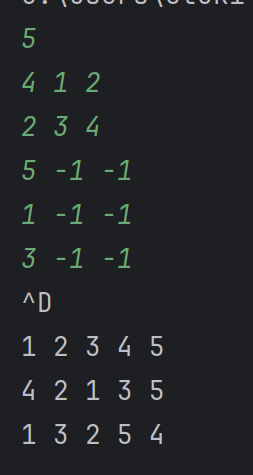


Рисунок 2.1 - 1-ый результат работы программы

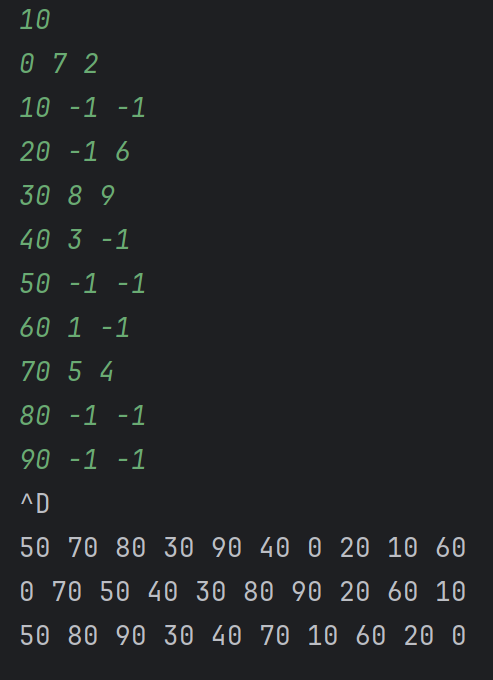


Рисунок 2.2. - 2-рой результат работы программы

# Выводы

В ходе выполнения лабораторной работы мной были освоены и изучены: понятие двоичное дерево; обход дерева (симметричный) in-order, (прямой) pre-order и (обратный) post-order. Написанная программа была протестирована, полученный результат соответствует значению в примере.