SKILLFACTORY

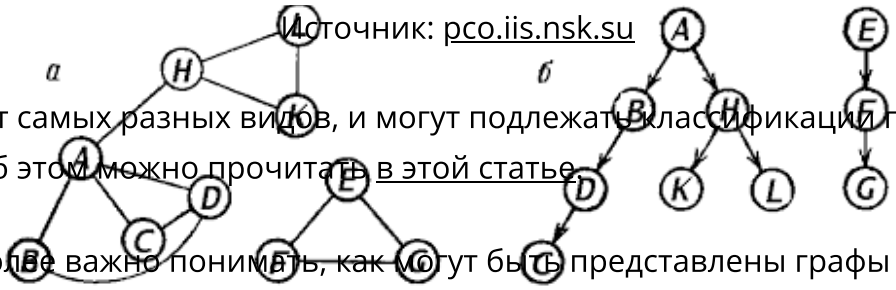
7.5 Нелинейные структуры данных: графы и деревья

Графы

Совершенно другой тип структур данных представляют собой графы.

> **Граф** — это структура, имеющая узлы (вершины графа) и связи между ними (ребра).

На этой картинке можно увидеть несколько примеров графов:

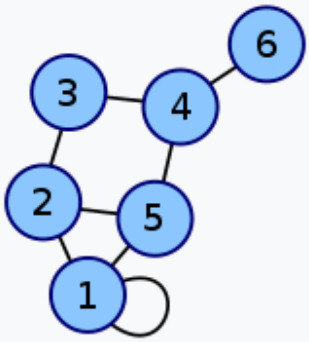


Источник: pco.iis.nsk.su

Графы бывают самых разных видов, и могут подлежать классификации по разным критериям. Об этом можно прочитать в этой статье.

Нам сейчас более важно понимать, как могут быть представлены графы в памяти компьютера.

Первый способ представления называется **матрица смежности**.

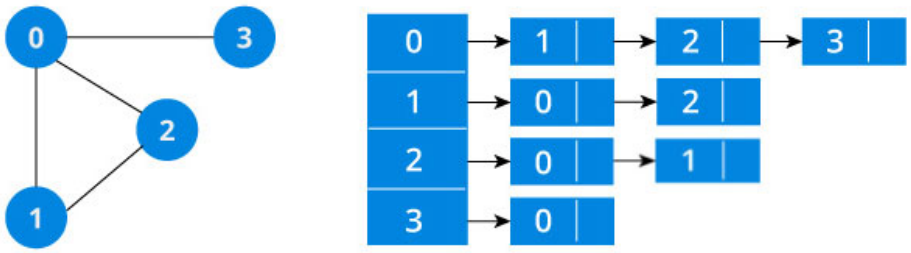
Граф	Матрица смежности
	$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$

Источник: teletype.in

В матрице смежности (двумерном массиве) каждая строка и каждый столбец соответствуют вершине. На пересечении столбца *x* и строки *y* стоит 1, если эти две вершины *x* и *y* соединены ребром, иначе стоит ноль. Такой способ представления является эффективным, если граф является плотным — в нём много рёбер. Однако он сильно теряет в эффективности, если в нём присутствует большое количество вершин, но мало рёбер.

Второй способ представления — **список смежности**.

Существует много реализаций графов списком смежности. Основная суть сводится к тому, чтобы хранить массив (список, словарь) связанных списков. Каждый внутренний список, в свою очередь, хранит вершины, с которыми имеет связь вершина, образующая список.



Источник: evileg.com

Список смежности легко реализуется с помощью двух встроенных типов *Python* — словарь (`dict`) и список (`list`).

Например, граф из картинки выше можно было бы представить следующим образом:

```
G = {0 : [1, 2, 3],
     1 : [0, 2],
     2 : [0, 1],
     3 : [0]}
```

Большинство необходимых операций с этими структурами имеют невысокую сложность по сравнению с двумерным массивом. Давайте посмотрим на них в сравнении:

Операция	Список смежности	Матрица смежности
Проверка наличия ребра (*x*,*y*)	*O*(|*E*|)	*O*(1)
Определение степени вершины	*O*(1)	*O*(|*V*|)
Использование памяти для разреженных графов	*O*(|*V*|+|*E*|)	*O*(|*V*|^2)
Вставка/удаление	*O*(1)	*O*(*d*)
Обход графа	*O*(|*V*|+|*E*|)	*O*(|*V*|^2)

В данной таблице $|V|$ — количество вершин, $|E|$ — количество рёбер, d — количество рёбер при вершине (степень вершины).

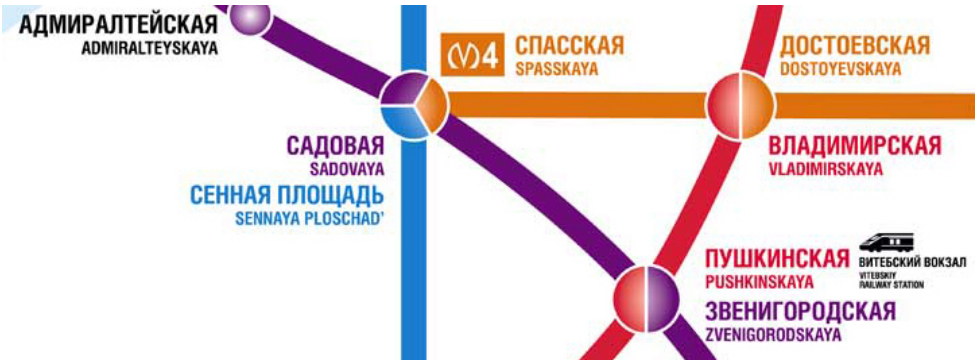
На рисунке изображена часть схемы метрополитена г. Санкт-Петербурга. Давайте попробуем представить её в виде списка смежности.


```
G = {"Лиговский проспект" :
          ["Площадь Александра Невского 2"],
     "Площадь Александра Невского 2" :
          ["Площадь Александра Невского 1",
           "Лиговский проспект",
           "Новочеркасская"],
     "Площадь Александра Невского 1" :
          ["Площадь Александра Невского 2",
           "Елизаровская"],
     "Новочеркасская" :
          ["Площадь Александра Невского 2",
           "Ладожская"],
     "Ладожская" :
          ["Новочеркасская",
           "Проспект Большевиков"],
     "Проспект Большевиков" :
          ["Ладожская",
           "Дыбенко"],
     "Дыбенко" :
          ["Проспект Большевиков"]}
```

Задание 17.5.1

Задание на самопроверку.

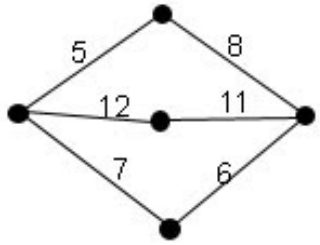
Представьте эту часть схемы в виде графа и создайте список смежности, используя словарь.



Ответ

```
G = {"Адмиралтейская" :
          ["Садовая"],
     "Садовая" :
          ["Сенная площадь",
           "Спасская",
           "Адмиралтейская",
           "Звенигородская"],
     "Сенная площадь" :
          ["Садовая",
           "Спасская"],
     "Спасская" :
          ["Садовая",
           "Сенная площадь",
           "Достоевская"],
     "Звенигородская" :
          ["Пушкинская",
           "Садовая"],
     "Пушкинская" :
          ["Звенигородская",
           "Владимирская"],
     "Владимирская" :
          ["Достоевская",
           "Пушкинская"],
     "Достоевская" :
          ["Владимирская",
           "Спасская"]}
```

Граф, помимо прочего, может быть взвешенным — каждое ребро может иметь свой вес в графе.



Источник: function-x.ru

Такой граф можно представить в виде словаря словарей, где значение ключа представляет собой вес ребра.

Любой невзвешенный граф имеет одинаковый вес рёбер, поэтому можем представить один из примеров выше в виде «словаря словарей», несмотря на то, что все «веса» рёбер одинаковы.

```
G = {0 : {1 : 1,
          2 : 1,
          3 : 1},
     1 : {0 : 1,
          2 : 1},
     2 : {0 : 1,
          1 : 1},
     3 : {0 : 1}}
```

Задание 7.5.2

Возьмите граф из предыдущего задания (с картой метро) и постройте из него взвешенный граф. В качестве весов используйте время, необходимое для того, чтобы доехать (или перейти) с одной станции на другую. Для этого можно воспользоваться сервисом Яндекс.Метро.

▼ Ответ

```
G = {"Адмиралтейская" :
          {"Садовая" : 4},
     "Садовая" :
          {"Сенная площадь" : 3,
           "Спасская" : 3,
           "Адмиралтейская" : 4,
           "Звенигородская" : 5},
     "Сенная площадь" :
          {"Садовая" : 3,
           "Спасская" : 3},
     "Спасская" :
          {"Садовая" : 3,
           "Сенная площадь" : 3,
           "Достоевская" : 4},
     "Звенигородская" :
          {"Пушкинская" : 3,
           "Садовая" : 5},
     "Пушкинская" :
          {"Звенигородская" : 3,
           "Владимирская" : 4},
     "Владимирская" :
          {"Достоевская" : 3,
           "Пушкинская" : 4},
     "Достоевская" :
          {"Владимирская" : 3,
           "Спасская" : 4}}
```

При работе с графами одна из наиболее частых задач — поиск кратчайшего пути от одной вершины к другой. Сейчас мы с вами попытаемся реализовать алгоритм, позволяющий найти его.

Он носит название **«алгоритм Дейкстры»**.

Его суть заключается в том, чтобы последовательно перебирать вершины одну за другой в поисках кратчайшего пути до этой вершины. Вершину будем называть **предком** для другой, если она идет раньше по пути перемещения по ребрам в графе. Ближайший предок — это предок, имеющий прямую связь (ребро) с рассматриваемой. Рассмотрим этот алгоритм также на примере взвешенного графа станций метро из последней задачи.

Для начала нам потребуется дополнительная структура данных для хранения расстояний. Если вершины пронумерованы числами, то можно использовать массив, но т. к. мы имеем проименованные узлы, то удобнее пользоваться словарём.

```
D = {k : 100 for k in G.keys()}
```

Проинициализируем словарь расстояний числами, которые заведомо больше максимального расстояния в графе. Значения 100 в данной задаче нам будет более чем достаточно. Одну из вершин мы должны выбрать как стартовую. Поэтому стартовая вершина будет предком для всех остальных. Расстояние для неё (от неё же самой) будет равно нулю. Пусть это будет «Адмиралтейская».

```
D["Адмиралтейская"] = 0
```

Также нам потребуется хранить словарь с булевыми значениями, в котором *True* — если вершина просмотрена, иначе — *False*.

```
U = {k : False for k in G.keys()}
```

Далее мы должны пройти циклом из *n* итераций, выбирая вершину с наименьшим *D* среди непросмотренных. Очевидно, что на первой итерации будет выбрана стартовая вершина. Из неё мы должны проверить все вершины, в которые можем перейти, и в *D* записать наименьшее расстояние до них. Пока что мы можем идти только из стартовой вершины, поэтому запишутся именно эти расстояния. Стартовая вершина станет помеченной как уже просмотренная. После чего начнется поиск вершины с минимальным *D* из уже просмотренных (куда можно добраться из стартовой). От неё также будут строиться возможные ребра и проверяться минимум расстояний. И так далее, пока процесс не завершится. Утверждается, что достаточно числа итераций равного количеству вершин. При достижении этого алгоритм завершится корректно.

```
D = {k : 100 for k in G.keys()} # расстояния
start_k = 'Адмиралтейская' # стартовая вершина
D[start_k] = 0 # расстояние от неё до самой себя равно нулю
U = {k : False for k in G.keys()} # флаги просмотра вершин

for _ in range(len(D)):
    # выбираем среди непросмотренных наименьшее по расстоянию
    min_k = min([k for k in U.keys() if not U[k]], key = lambda x:
D[x])

    for v in G[min_k].keys(): # проходимся по всем смежным вершинам
        D[v] = min(D[v], D[min_k] + G[min_k][v]) # минимум
    U[min_k] = True # просмотренную вершину помечаем
```

Результат работы программы можно увидеть в словаре *D*.

```
{'Адмиралтейская': 0,
 'Садовая': 4,
 'Сенная площадь': 7,
 'Спасская': 7,
 'Звенигородская': 9,
 'Пушкинская': 12,
 'Владимирская': 14,
 'Достоевская': 11}
```

В нем отражено кратчайшее расстояние от «Адмиралтейской» до станции, которая задает ключ.

Таким образом, мы реализовали тот же самый алгоритм, который использует Яндекс для поиска кратчайшего пути между станциями!

Алгоритм Дейкстры можно модифицировать таким образом, что можно определить не только величину пути, но ещё и сами вершины минимального пути.

Для этого определим ещё один словарь *P*, в котором будем для каждой вершины хранить вершину-предок с минимальным расстоянием.

```
P = {k : None for k in G.keys()}
```

Задание 7.5.3

Модифицируйте алгоритм Дейкстры таким образом, что в массив P по соответствующему ключу будет записываться предок с минимальным расстоянием, если это необходимо.

▼ Ответ

```
D = {k : 100 for k in G.keys()} # расстояния
start_k = 'Адмиралтейская' # стартовая вершина
D[start_k] = 0 # расстояние от нее до самой себя равно нулю
U = {k : False for k in G.keys()} # флаги просмотра вершин
P = {k : None for k in G.keys()} # предки

for _ in range(len(D)):
    # выбираем среди непросмотренных наименьшее по расстоянию
    min_k = min([k for k in U.keys() if not U[k]], key = lambda x:
D[x])

    for v in G[min_k].keys(): # проходимся по всем смежным вершинам
        if D[v] > D[min_k] + G[min_k][v]: # если расстояние от текущей
вершины меньше
            D[v] = D[min_k] + G[min_k][v] # то фиксируем его
            P[v] = min_k # и записываем как предок
    U[min_k] = True # просмотренную вершину помечаем
```

Теперь проходом цикла `while` по вершинам в словаре *P* можно найти вершины кратчайшего пути, правда, в обратном порядке:

```
pointer = some_station # куда должны прийти
while pointer is not None: # перемещаемся, пока не придём в стартовую точку
    print(pointer)
    pointer = P[pointer]
```

Задание 7.5.4

0/1 point (graded)

Запишите станции в порядке вывода через запятую и пробел (в именительном падеже с заглавной буквы), если нужно найти кратчайший путь от Адмиралтейской до Владимирской.

Адмиралтейская, Садо ✖

Отправить

✖ Incorrect (0/1 point)

Деревья

Один из наиболее часто используемых подвидов графов — это **деревья**.

Чтобы граф считался деревом, необходимо выполнение нескольких условий:

1. Граф должен быть связным — все вершины должны быть с рёбрами.
2. Должны отсутствовать циклы.
3. Часто под деревьями подразумеваются только неориентированные и невзвешенные графы.

При соблюдении этих условий граф можно визуально представить в виде структуры, напоминающей обычное дерево. Только, как правило, оно «растёт» вниз.

В дереве можно выделить:

- корневой узел;
- потомки;
- лист или терминальный узел — узел, не имеющий потомков;
- внутренние узлы — некорневые узлы, имеющие хотя бы одного потомка.

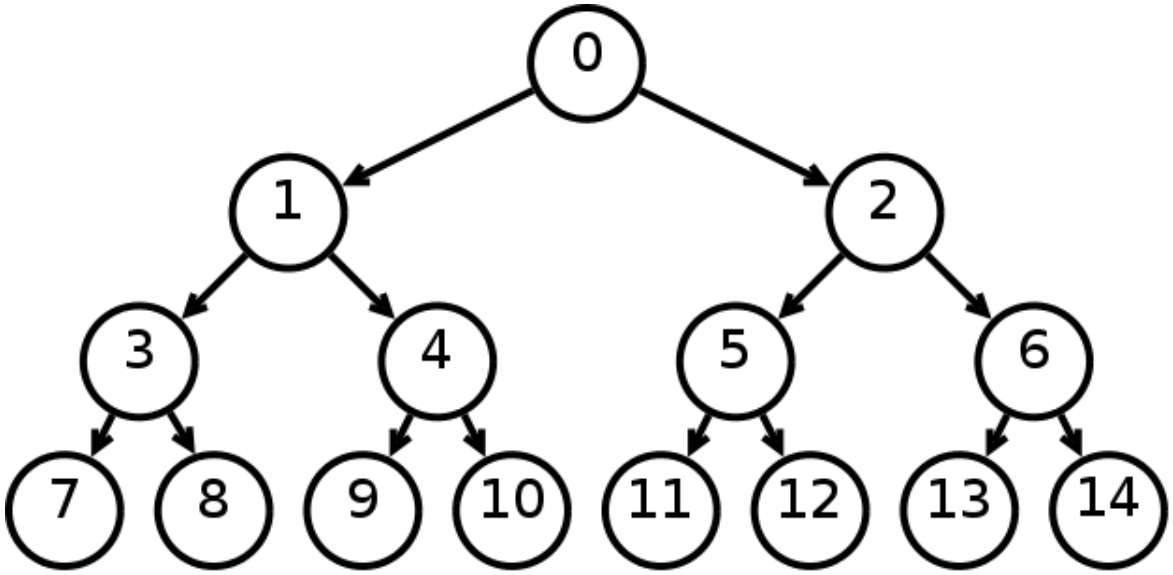
Древовидные структуры мы видим повсеместно: фамильное древо, структура организаций, которая тоже чаще всего иерархическая. Или же, например, объектная модель документов (*DOM*) в языке *HTML*, с которой мы познакомимся в следующих модулях.

В зависимости от максимального количества потомков в одной вершине различа

- бинарные;
- тернарные деревья;
- *n*-арные деревья (с максимальным количеством потомков *n*);
- 2-3 деревья, 2-3-4, в которых помимо увеличенного количества потомков в самом узле может храниться больше данных.

О таких деревьях в курсе мы говорить не будем, но после знакомства с основной информацией о бинарных деревьях, можно обратиться к статье.

Итак, рассмотрим бинарное дерево. Основное его свойство заключается в том, что у каждого узла может быть **не более** 2 потомков — соответственно, левый и/или правый.



Источник: habr.com

Линейные структуры данных (массивы, списки, очереди, стеки) более понятны интуитивно. Но как же хранить в памяти деревья? Здесь нам поможет знание объектно-ориентированного программирования, а воспользуемся мы принципом, по которому хранятся списки в памяти. Напомним, что каждый элемент списка хранит собственное значение и указатель на следующий элемент.

В нашей структуре данных, в каждом узле бинарного дерева мы будем хранить указатель на левого и правого потомка.

```
class BinaryTree:
    def __init__(self, value):
        self.value = value
        self.left_child = None
        self.right_child = None
```

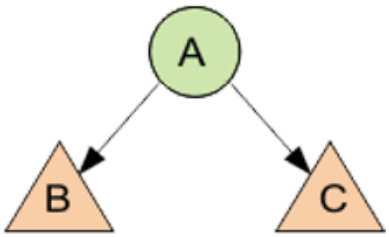
Мы создали класс узла, а в конструкторе записали значение, которое должно храниться в нём. Также инициализировали левого и правого потомка. Пока что в них ничего не хранится — нужно иметь процедуру вставки новых элементов. Напишем разные методы для вставки на место левого потомка и на место правого потомка.

```
def insert_left(self, next_value):
    if self.left_child is None:
        self.left_child = BinaryTree(next_value)
    else:
        new_child = BinaryTree(next_value)
        new_child.left_child = self.left_child
        self.left_child = new_child
    return self
```

Поясним, что здесь произошло. Если в текущем узле нет левого потомка, то новый узел вставляем на его место. Если левый потомок уже существует — он становится таким же левым потомком, но уже нового узла. Иными словами, он остается левым, но его глубина увеличивается. Аналогично поступим с правым.

```
def insert_right(self, next_value):
    if self.right_child is None:
        self.right_child = BinaryTree(next_value)
    else:
        new_child = BinaryTree(next_value)
        new_child.right_child = self.right_child
        self.right_child = new_child
    return self
```

В обоих случаях мы возвращаем ссылку на текущий узел. Это нам необходимо, чтобы создавать цепочки действий. Рассмотрим на примере:



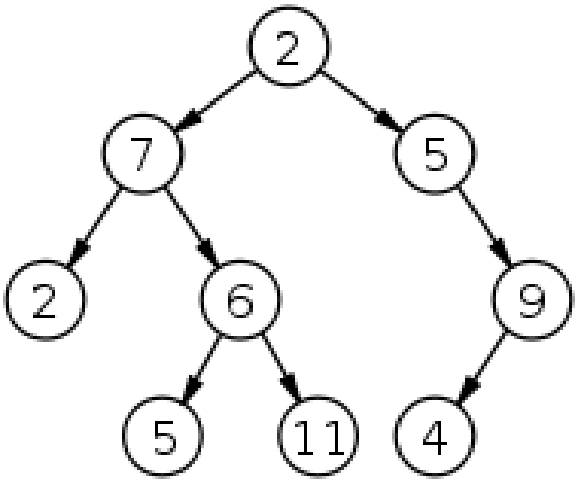

```
A_node = BinaryTree('A').insert_left('B').insert_right('C')
```

В одной строчке мы создали корневой узел дерева, вставили левого потомка и затем — сразу правого. Получая ссылки на потомков через атрибуты `left_child` и `right_child`, можно проделать ту же самую цепочку действий, чтобы расширить дерево.

Задание 7.5.5

Задание на самопроверку.

Реализуйте структуру дерева при помощи класса `BinaryTree`.



Источник: dic.academic.ru

▼ Ответ

```
# создаём корень и его потомков /7|2|5\
node_root = BinaryTree(2).insert_left(7).insert_right(5)
# левое поддерево корня /2|7|6\
node_7 = node_root.left_child.insert_left(2).insert_right(6)
# правое поддерево предыдущего узла /5|6|11\
node_6 = node_7.right_child.insert_left(5).insert_right(11)
# правое поддерево корня /|5|9\
node_5 = node_root.right_child.insert_right(9)
# левое поддерево предыдущего узла корня /4|9|\
node_9 = node_5.right_child.insert_left(4)
```

Обход дерева

В случае линейных структур данных итерация по элементам происходила достаточно интуитивно — в массивах мы делали это по индексам, в списках — по указателям. В стеках и очередях, как правило, нет необходимости в итерации, потому что мы всегда смотрим на крайние элементы структуры. В случае дерева нелинейная структура требует развития особых подходов в обходе по нему.

Различают два основных способа обхода:

1. **Поиск в глубину (*depth-first search*, *DFS*)**. Его основная суть заключается в том, что, проходя по каждому узлу, мы сначала идём в его потомка, а потом возвращаемся обратно — это обход с возвратом. Такой поиск бывает трёх видов:
   - префиксный (`pre-order`);
   - постфиксный (`post-order`);
   - инфиксный (`in-order`).
2. **Поиск в ширину (*breadth-first search*, *BFS*)**. Такой обход осуществляется в обходе уровня за уровнем.

Начнём с рассмотрения поиска в глубину.

В обходе в глубину мы всегда используем рекурсивный подход: префиксный, постфиксный или инфиксный, подходы отличаются лишь порядком выполнения процедуры обработки узла и вызовов этой же функции на потомках.

Рассмотрим префиксный подход. Сначала мы должны обработать значение самого узла (поэтому он и *префик*сный), а затем рекурсивно проделать то же самое с левым потомком, и затем — с правым. В качестве процедуры обработки узла возьмём самое простое — печать его значения.

```
def pre_order(self):
    print(self.value) # процедура обработки

    if self.left_child is not None: # если левый потомок существует
        self.left_child.pre_order() # рекурсивно вызываем функцию

    if self.right_child is not None: # если правый потомок существует
        self.right_child.pre_order() # рекурсивно вызываем функцию
```

Давайте посмотрим, в каком порядке будет производиться префиксный обход дерева в глубину на примере созданного нами дерева.

```
node_root.pre_order()
# 2
# 7
# 2
# 6
# 5
# 11
# 5
# 9
# 4
```

Сначала мы записали значение корневого узла (2), после чего — его левого потомка (7). У него также есть левый потомок (2), который является листом, поэтому происходит возврат на предыдущий уровень — печатается значение правого потомка (6). Дальше просматриваются его потомки и происходит возврат до того уровня, который не просмотрен — правое поддерево корневого узла (/|2|5\).

Постфиксный обход дерева отличается порядком вызова для **обоих** потомков и процедуры обработки узла.

Задание 7.5.6

Задание на самопроверку.

Напишите метод постфиксного обхода в глубину.

▼ Ответ

```
def post_order(self):
    if self.left_child is not None: # если левый потомок существует
        self.left_child.post_order() # рекурсивно вызываем функцию

    if self.right_child is not None: # если правый потомок существует
        self.right_child.post_order() # рекурсивно вызываем функцию

    print(self.value) # процедура обработки
```

Задание 7.5.7

1 point possible (graded)

Для рассматриваемого примера напишите значения узлов (через запятую и пробел) в порядке постфиксного обхода.

Отправить

Инфиксный подход заключается в том, что порядок обработки узла и его потомков смешивается: сначала шагаем в левое поддерево, потом обрабатываем сам узел, затем — правое поддерево. В итоге получается, что мы как будто «читаем» дерево слева направо.

Метод инфиксного обхода в глубину:

```
def in_order(self):
    if self.left_child is not None: # если левый потомок существует
        self.left_child.in_order() # рекурсивно вызываем функцию

    print(self.value) # процедура обработки

    if self.right_child is not None: # если правый потомок существует
        self.right_child.in_order() # рекурсивно вызываем функцию
```

В результате инфиксного обхода получаем следующий порядок узлов.

```
node_root.in_order()
# 2
# 7
# 5
# 6
# 11
# 2
# 5
# 4
# 9
```

По этой ссылке можно найти ещё немного информации про алгоритмы обхода в деревьях, а также не рассмотренный нами алгоритм обхода в ширину.