Теория по графам

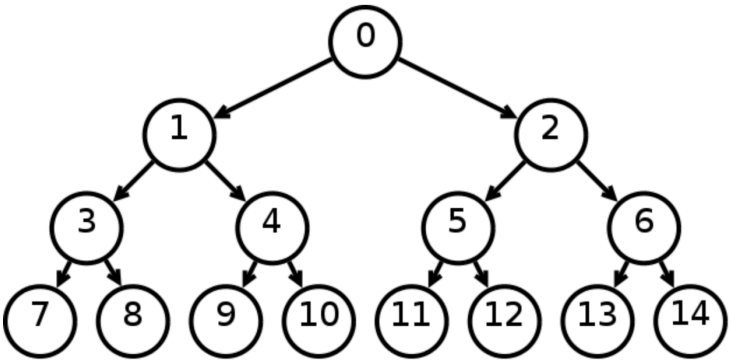
N-арное дерево
Это дерево (ориентированное или неориентированное), в котором степени вершин не превосходят N+1.

Сбалансированное дерево
Дерево называется сбалансированным тогда и только тогда, когда для каждого его узла высоты его левого и правого поддеревьев отличаются не более чем на единицу. Для каждого узла дерева можно определить показатель сбалансированности как разность между высотой правого и левого поддерева данного узла.

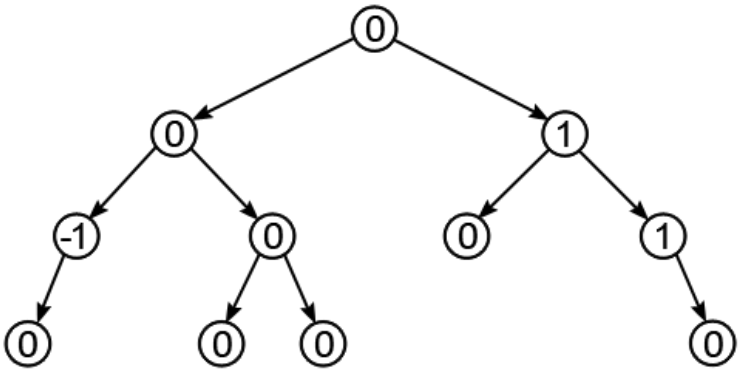
Бинарное дерево
Это иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Двоичное дерево является упорядоченным ориентированным деревом.

Бинарное дерево поиска
Двоичное дерево поиска (англ. binary search tree, BST) — двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- оба поддерева — левое и правое — являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше либо равны, нежели значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных больше, нежели значение ключа данных самого узла X.



Дерево AVL
АВЛ-дерево — сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.



Красно-черное дерево
Красно-черные деревья относятся к сбалансированным бинарным деревьям поиска.

Как бинарное дерево, красно-черное обладает свойствами:
1) Оба поддерева являются бинарными деревьями поиска.
2) Для каждого узла с ключом k выполняется критерий упорядочения:
ключи всех левых потомков <= k < ключи всех правых потомков

Свойства красно-черных деревьев:
1) Каждый узел окрашен либо в красный, либо в черный цвет (в структуре данных узла появляется дополнительное поле – бит цвета).
2) Корень окрашен в черный цвет.
3) Листья(так называемые NULL-узлы) окрашены в черный цвет.
4) Каждый красный узел должен иметь два черных дочерних узла. Нужно отметить, что у черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.
5) Пути от узла к его листьям должны содержать одинаковое количество черных узлов(это черная высота).

Ну и почему такое дерево является сбалансированным?

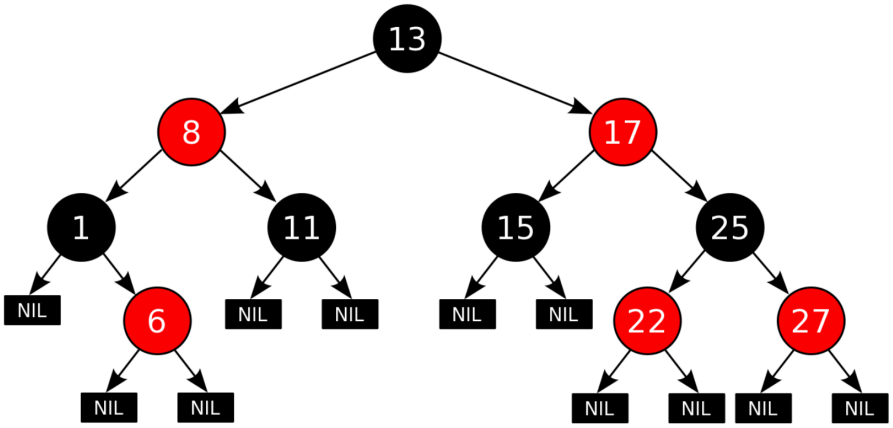
Действительно, красно-черные деревья не гарантируют строгой сбалансированности (разница высот двух поддеревьев любого узла не должна превышать 1), как в АВЛ-деревьях. Но соблюдение свойств красно-черного дерева позволяет обеспечить выполнение операций вставки, удаления и выборки за время $O(logN)$. И сейчас посмотрим, действительно ли это так.

Пусть у нас есть красно-черное дерево. Черная высота равна bh(black height).

Если путь от корневого узла до листового содержит минимальное количество красных узлов (т.е. ноль), значит этот путь равен bh.

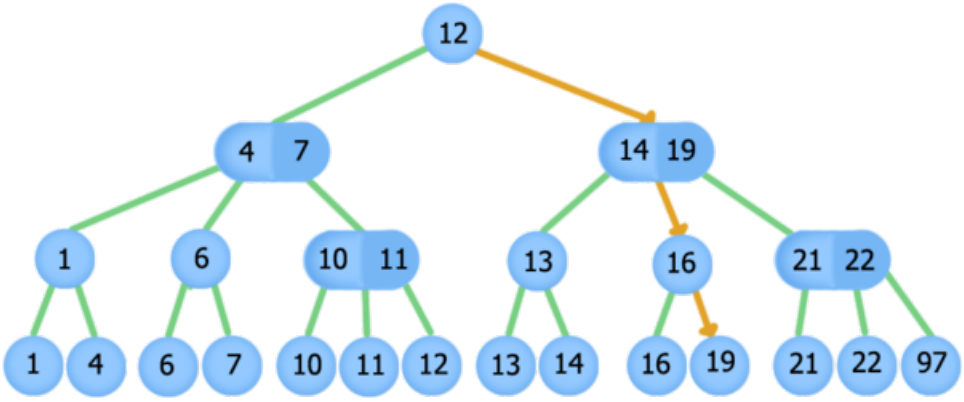
Если же путь содержит максимальное количество красных узлов (bh в соответствии со свойством 4), то этот путь будет равен $2bh$.

То есть, пути из корня к листьям могут различаться не более, чем вдвое, этого достаточно, чтобы время выполнения операций в таком дереве было $O(logN)$



2-3 дерево
2-3 дерево — сбалансированное дерево поиска, обладающее следующими свойствами:
- нелистовые вершины имеют либо 2 , либо 3 сына

- нелистовая вершина, имеющая двух сыновей, хранит максимум левого поддерева. Нелистовая вершина, имеющая трех сыновей, хранит два значения. Первое значение хранит максимум левого поддерева, второе максимум центрального поддерева,
- сыновья упорядочены по значению максимума поддерева сына
- все листья лежат на одной глубине,
- высота 2-3 дерева $O(\log n)$, где n — количество элементов в дереве.



Пояснения:
Дерево — связный ациклический граф. Связность означает наличие маршрута между любой парой вершин, ацикличность — отсутствие циклов. Отсюда, в частности, следует, что число рёбер в дереве на единицу меньше числа вершин, а между любыми парами вершин имеется один и только один путь.

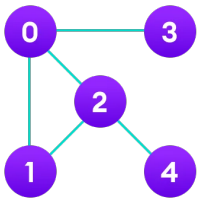
Обходы графа

Обход в глубину
Принцип его работы совпадает с его названием, данный алгоритм идет "внутрь" графа, до того момента как ему становится некуда идти, затем возвращается в предыдущую вершину и снова идет от нее до тех пор, пока есть куда идти. И так далее.

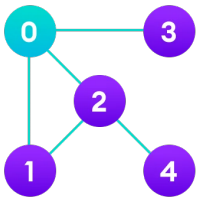
Алгоритм первого поиска Depth
Стандартная реализация DFS помещает каждую вершину графика в одну из двух категорий:
1. Посетил
2. Не Посетил

Цель алгоритма - пометить каждую вершину как посещенный, избегая при этом циклов.
Алгоритм DFS работает следующим образом:
1. Начните с того, что поместите любую из вершин графика поверх стека.
2. Возьмите верхний элемент стека и добавьте его в список посещенных.
3. Создайте список смежных узлов этой вершины. Добавьте те, которых нет в списке посещенных, в верхнюю часть стека.
4. Продолжайте повторять шаги 2 и 3, пока стек не опустеет.

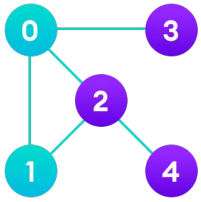
Применения алгоритма
1. Для поиска пути.
2. Для проверки двудольности графа.
3. Для поиска сильно связанных компонентов графа.
4. Для обнаружения циклов в графе.



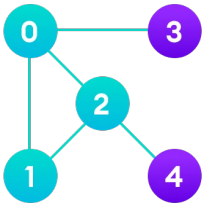
					Visited
					Stack



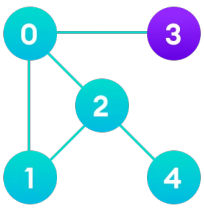
0					Visited
1	2	3			Stack



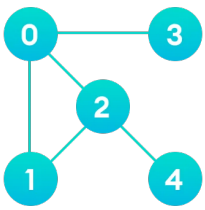
0	1				Visited
2	3				Stack



0	1	2			Visited
4	3				Stack



0	1	2	4		Visited
3					Stack



0	1	2	4	3	Visited
					Stack

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    print(start)

    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited


graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}

dfs(graph, '0')
```

Обход в ширину

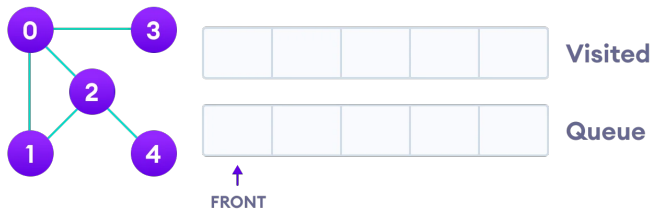
Стандартная реализация BFS помещает каждую вершину графика в одну из двух категорий:

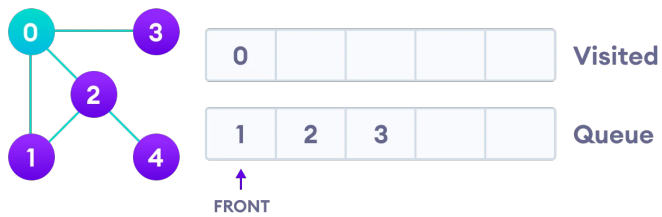
1. Посетил
2. Не Посетил

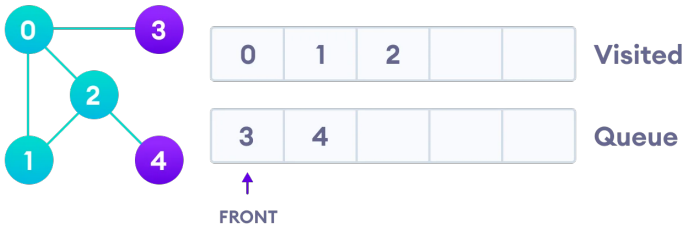
Цель алгоритма - пометить каждую вершину как посещенный, избегая при этом циклов.
Алгоритм работает следующим образом:

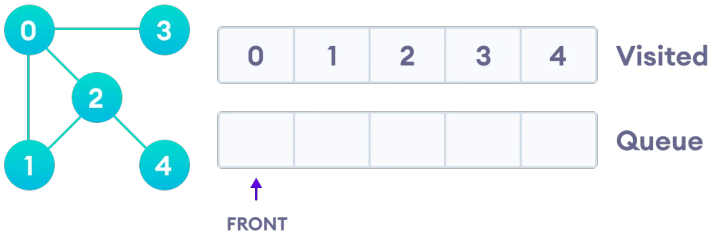
1. Начните с того, что поместите любую из вершин графа в задней части очереди.
2. Возьмите передний элемент очереди и добавьте его в список посещения.
3. Создайте список смежных узлов этой вершины. Добавьте те, которых нет в списке посещенных, в заднюю часть очереди.
4. Продолжайте повторять шаги 2 и 3, пока очередь не опустеет.
5. График может иметь две разные разрозненные части, поэтому, чтобы убедиться, что мы покрываем каждую вершину, мы также можем запустить алгоритм BFS на каждом узле.

Пример BFS










```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex)
            queue.extend(graph[vertex] - visited)

graph = {
    'A': set(['B', 'C']),
    'B': set(['A', 'D', 'E']),
    'C': set(['A', 'F']),
    'D': set(['B']),
    'E': set(['B', 'F']),
    'F': set(['C', 'E'])
}

bfs(graph, 'A')
```

Применение:

1) Вычисление расстояния до вершин
2) Поиск кратчайшего пути
3) Поиск компонент связности
4) Поиск взвешенных кратчайших путей

В последовательном случае алгоритмы имеют алгоритмическую сложность $O(|V|+|E|)$, где $|V|$ - число вершин в графе, $|E|$ - число ребер в графе