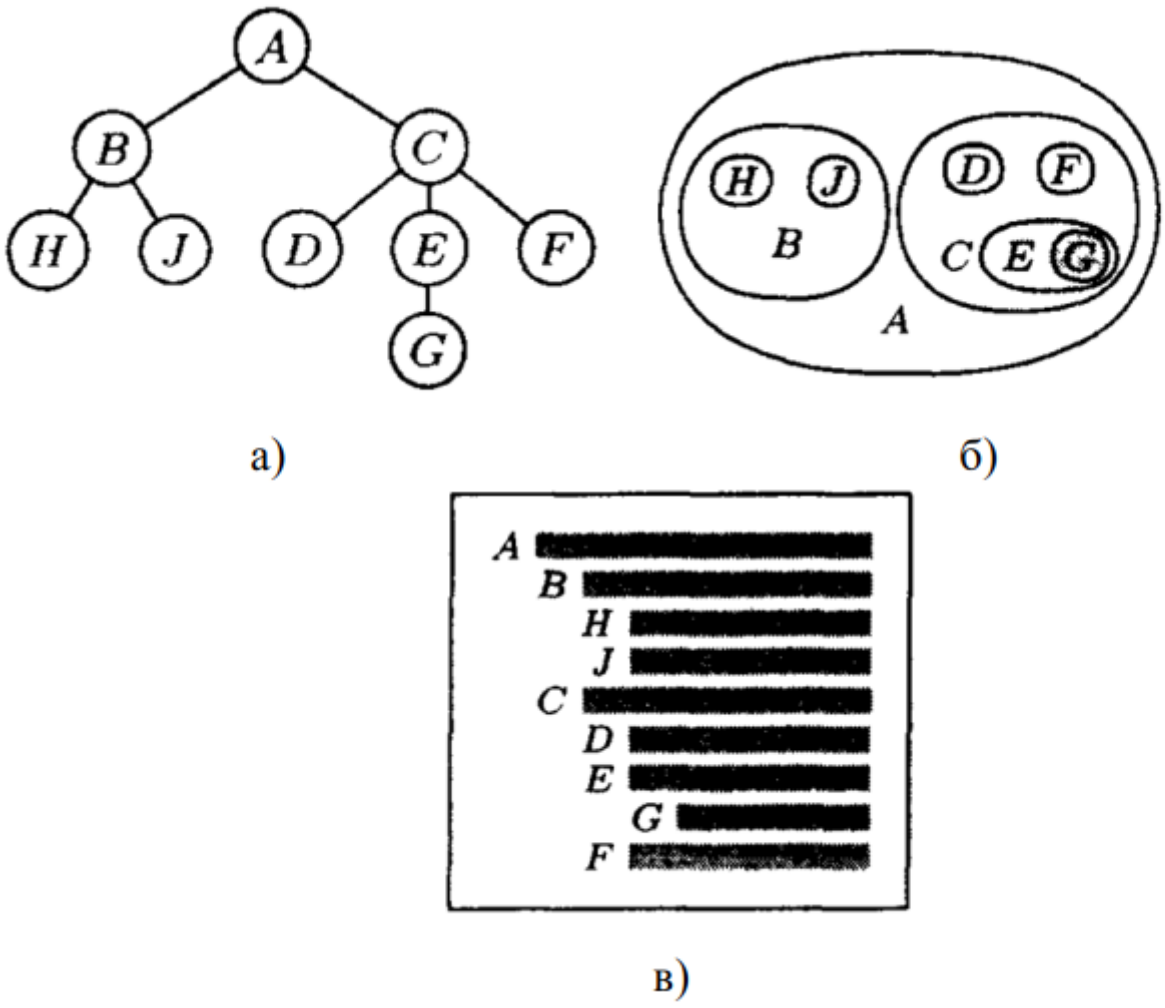
Деревья

Дерево - связный ациклический граф.

Представление деревьев: а – иерархическая структура, б – множества, в – линейное представление



Граф (или сеть) состоит из: вершин или узлов (vertice, node) и ребер или дуг или связей (edge, arc, link).

Путь - упорядоченный список узлов, связанных друг с другом.

Дочерние узлы (children) - все узлы имеющие входящие связи от некоторого узла, называются дочерними узлами данного узла.

Родитель (parent) - узел, исходящая связь которого соединена с данным узлом.

Сиблинги (sibling) - узлы, являющиеся дочерними узлами родителя данного узла.

Корень (root) — узел дерева, не имеющий родитетеля.

Лист (leaf node) (или терминальный узел) — узел, не имеющий дочерних узлов.

Внутренний узел — любой узел дерева, имеющий дочерние узлы, и таким образом, не являющийся листовым узлом.

Уровень (level) узла - длина пути от корня до данного узла. Уровень корня по определению равен 0.

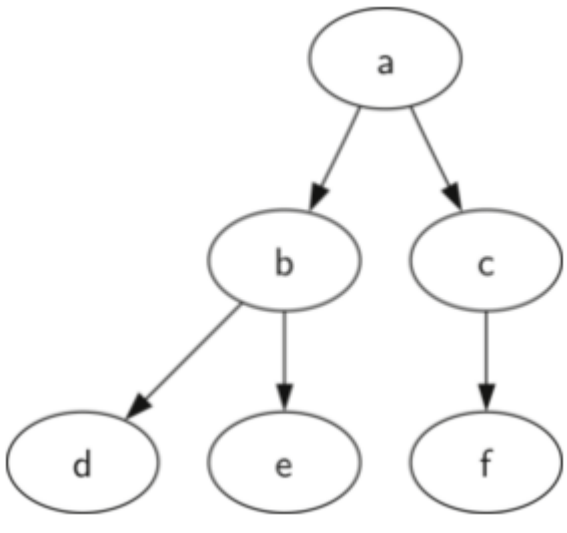
Высота дерева (height) - максимальный уровень среди узлов дерева.

Поддерво (subtree) - множество узлов и связей включающее родителя и всех его потомков.

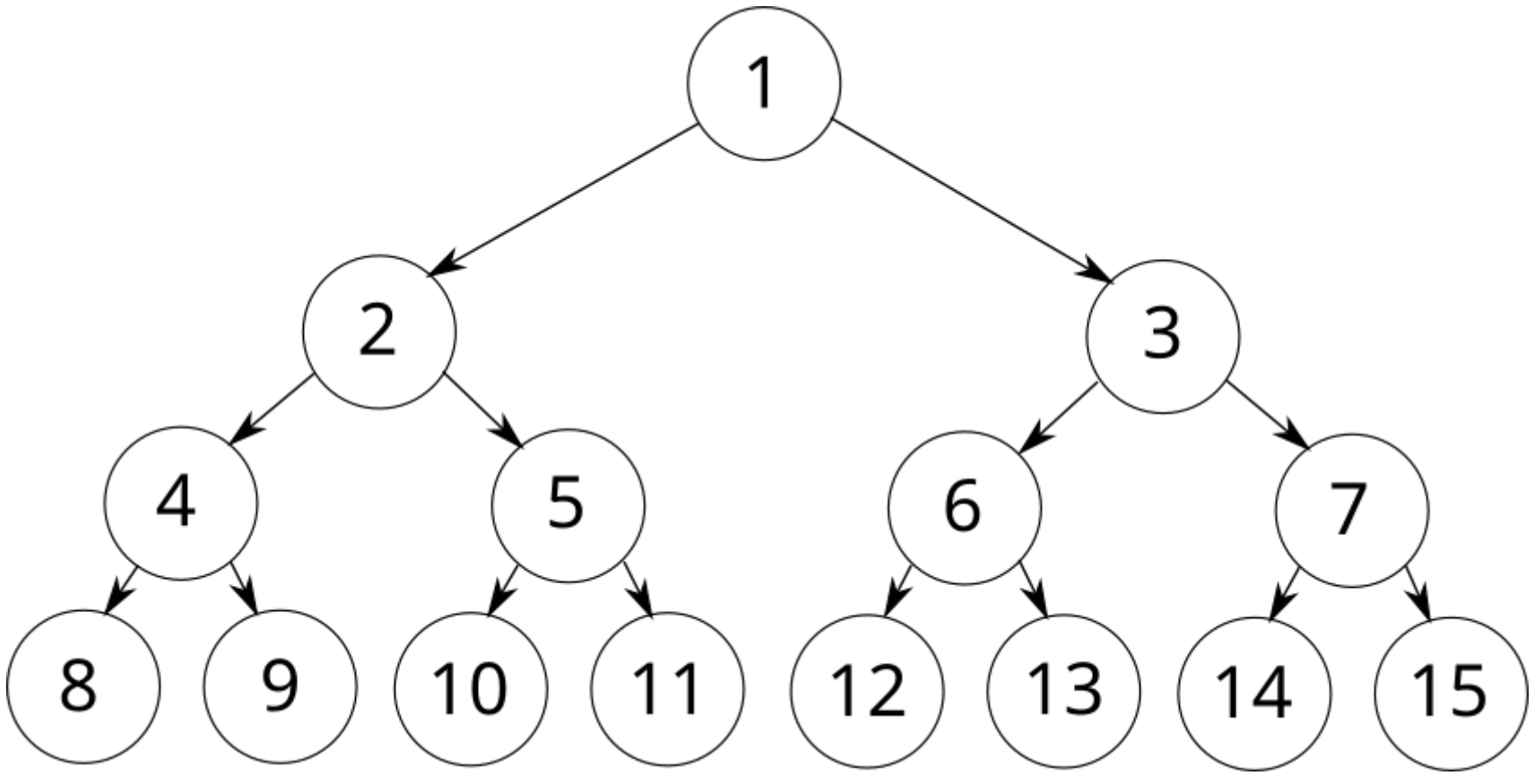
Знакомство с бинарными деревьями

Двоичное (бинарное) дерево (binary tree) — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Обычно, первый называется родительским узлом, а дети называются левым и правым наследниками.

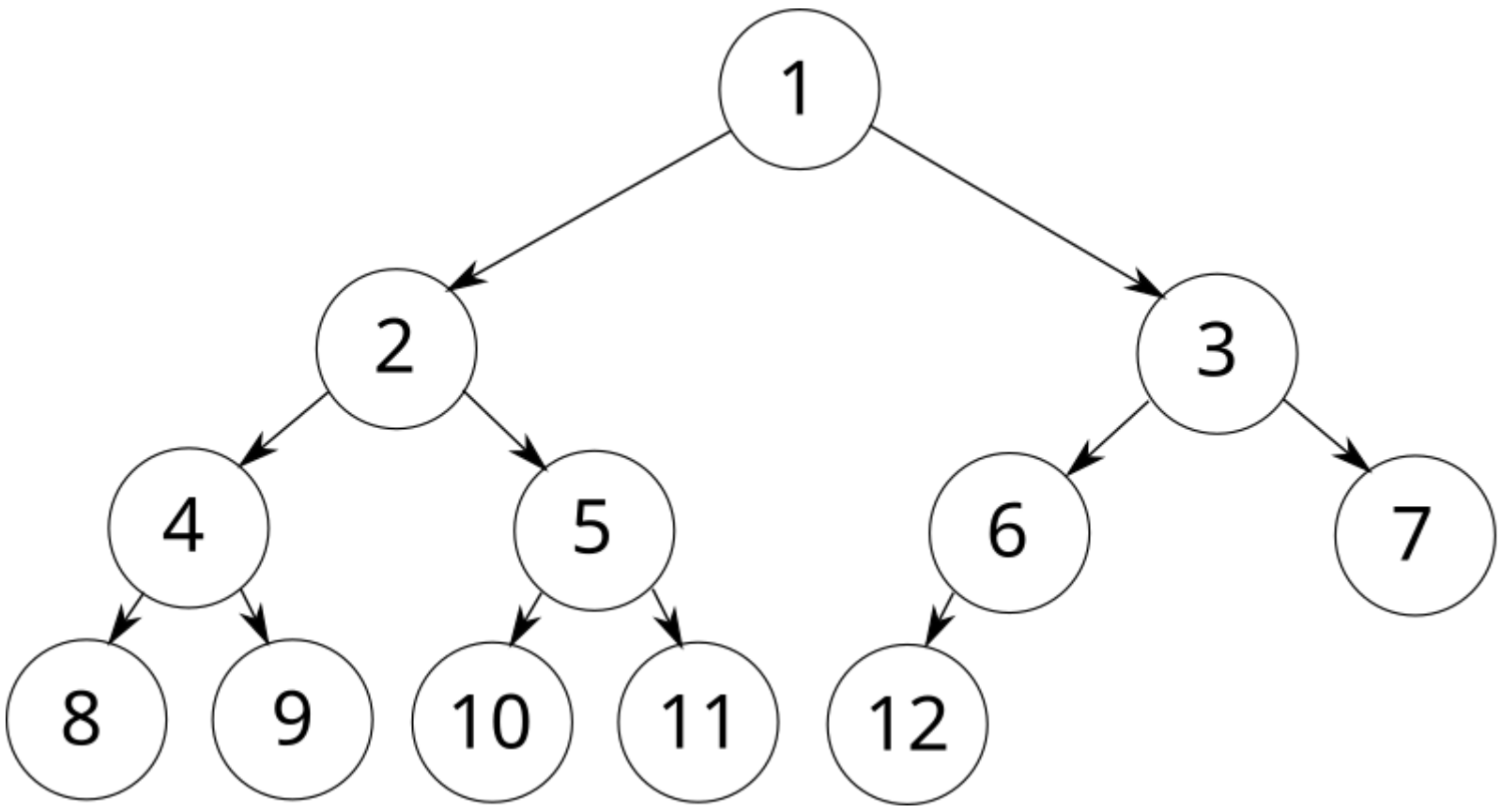
Пример небольшого бинарного дерева.



Среди бинарных деревьев отдельно выделяют полные бинарные деревья, все вершины которых имеют по две дочерних, кроме листьев, которые расположены на одинаковой глубине:



Также полными часто называют бинарные деревья, у которых полностью заполнены все уровни, кроме последнего:



In [2]:
```
# Представление бинарного дерева в виде списка списков:
my_tree = \
['a', #root
    ['b', #left subtree
        ['d', [], []],
        ['e', [], []] ],
    ['c', #right subtree
        ['f', [], []],
        [] ]
]
```

Пример API (application programming interface) для работы с двоичными деревьями:

- BinaryTree() - создание нового экземпляра бинарного дерева.
- get_left_child() - возвращает бинарное дерево связанное с левым дочерним узлом рассматриваемого узла.
- get_right_child() - возвращает бинарное дерево связанное с правым дочерним узлом рассматриваемого узла.
- get_root_val() - возвращает объект, хранящийся в данном узле.
- set_root_val(val) - сохраняет объект, хранящийся в данном узле.
- insert_left(val) - создает новое бинарное дерево связанное с левым дочерним узлом рассматриваемого узла.
- insert_right(val) - создает новое бинарное дерево связанное с правым дочерним узлом рассматриваемого узла.

In [3]:
```
print(my_tree)
print('left subtree = ', my_tree[1])
print('root = ', my_tree[0])
print('right subtree = ', my_tree[2])
```

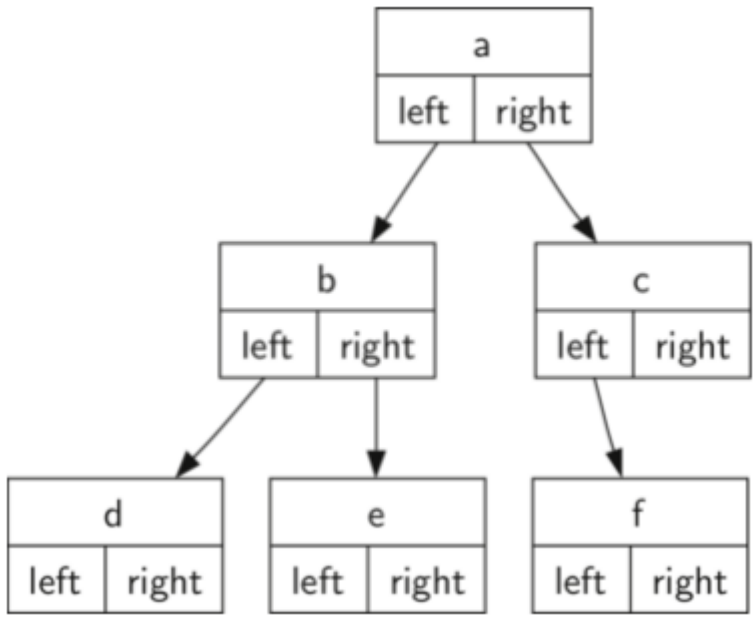
```
['a', ['b', ['d', [], []], ['e', [], []]], ['c', ['f', [], []], []]]
left subtree =  ['b', ['d', [], []], ['e', [], []]]
root =  a
right subtree =  ['c', ['f', [], []], []]
```

In [4]:
```
# Пример реализации функции insert_left для представления бинарного дерева в виде списка списков:
def insert_left(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])
    return root
```

In [4]:
```
insert_left(my_tree[2], 'H')
```

Out[4]:
```
['c', ['H', ['f', [], []], []], []]
```

Пример представления бинарного дерева в виде узлов и ссылок:



In [5]:
```
# Фрагмент реализации бинарного дерева с помощью представления в виде узлов и ссылок:
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None

    def insert_left(self,new_node):
        if self.left_child == None:
            self.left_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.left_child = self.left_child
            self.left_child = t

    def insert_right(self,new_node):
        if self.right_child == None:
            self.right_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.right_child = self.right_child
            self.right_child = t

    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_val(self,obj):
        self.key = obj

    def get_root_val(self):
        return self.key

    def __str__(self):
        return '{} ({}, {})'.format(self.get_root_val(), str(self.get_left_child()), str(self.get_right_
```

In [12]:
```
r = BinaryTree('a')
print(r)
```

```
a (None, None)
```

In [13]:
```
r.insert_left('b')
print(r)
print(r.get_left_child())
print(r.get_left_child().get_root_val())
```

```
a (b (None, None), None)
b (None, None)
b
```

In [14]:
```
r.insert_right('c')
print(r)
print(r.get_right_child())
print(r.get_right_child().get_root_val())
```

```
a (b (None, None), c (None, None))
c (None, None)
c
```

In [15]:
```
r.get_right_child().set_root_val('hello')
print(r.get_right_child().get_root_val())
```

```
hello
```

In [16]:
```
print(r)
```

```
a (b (None, None), hello (None, None))
```

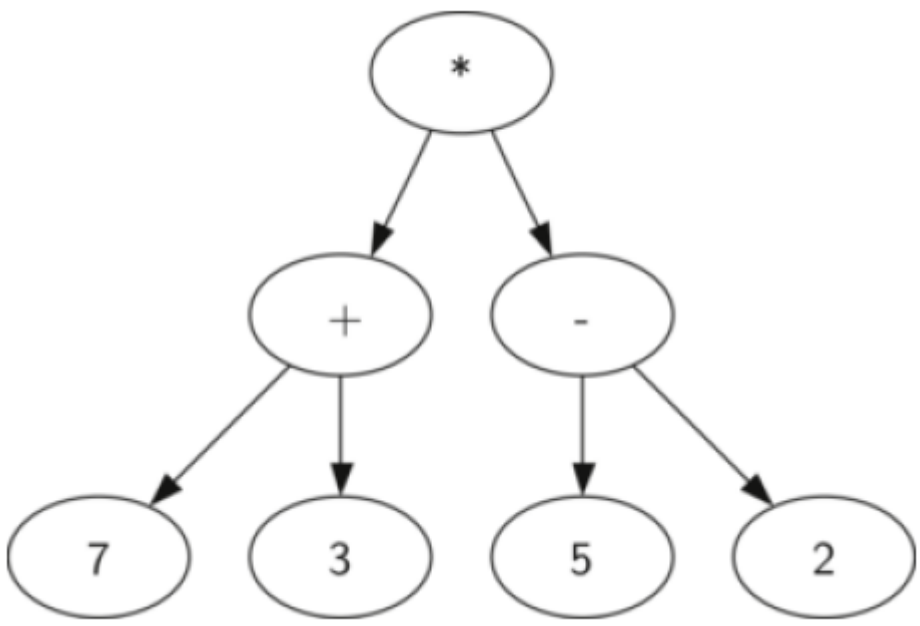
In [17]:
```
r.insert_right('f')
print(r)
```

```
a (b (None, None), f (None, hello (None, None)))
```

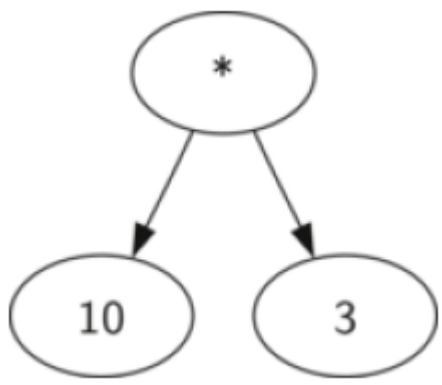
Использование бинарных деревьев в прикладных задачах

Двоичные деревья можно использовать для разбора (parsing) различных выражений. Рассмотрим, например, представления математического выражения ((7 + 3) * (5 – 2)) в виде двоичного дерева.

Представление математического выражения ((7 + 3) * (5 – 2)) в виде двоичного дерева:



Представление выражения (предложения) в виде дерева позволяет удобно работать с частями выражения как с поддервеьями или родительским и дочерним узлов (пример на рис.):



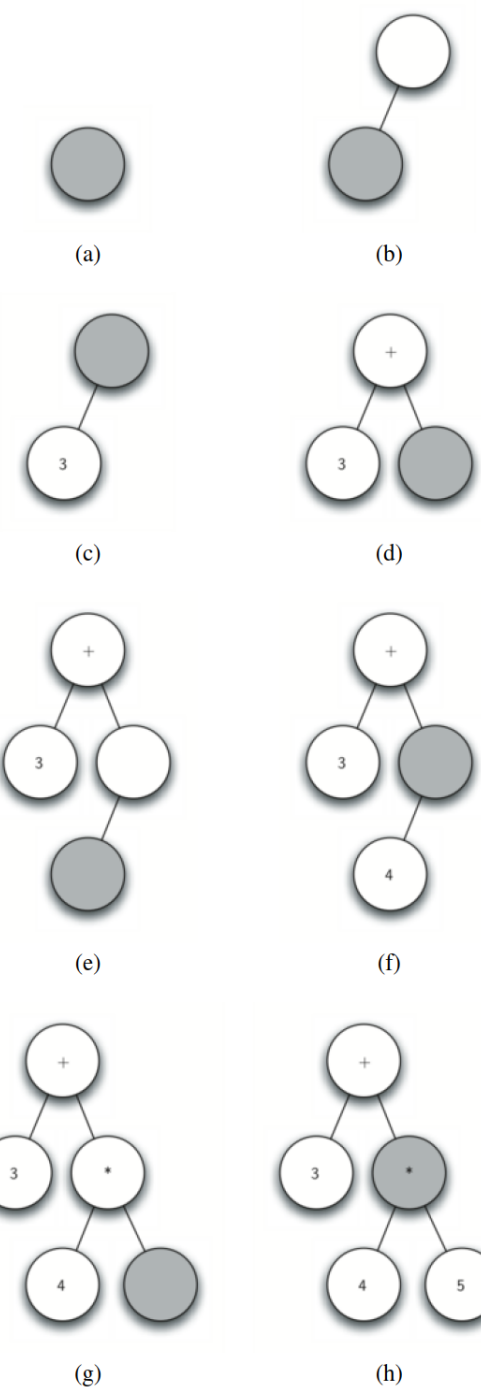
Для испльзования древовидного представления выражения необходимо решить следующие задачи:

- построить древововидное представление выражения в результате разбора математического выражения.
- вычислить математиченское выражение представленное в виде двоичного дерева.
- восстановить исходное математическое выражение из дрвеовидного представления.

Алгоритм разбора математического выражения для получения его древовидного представления:

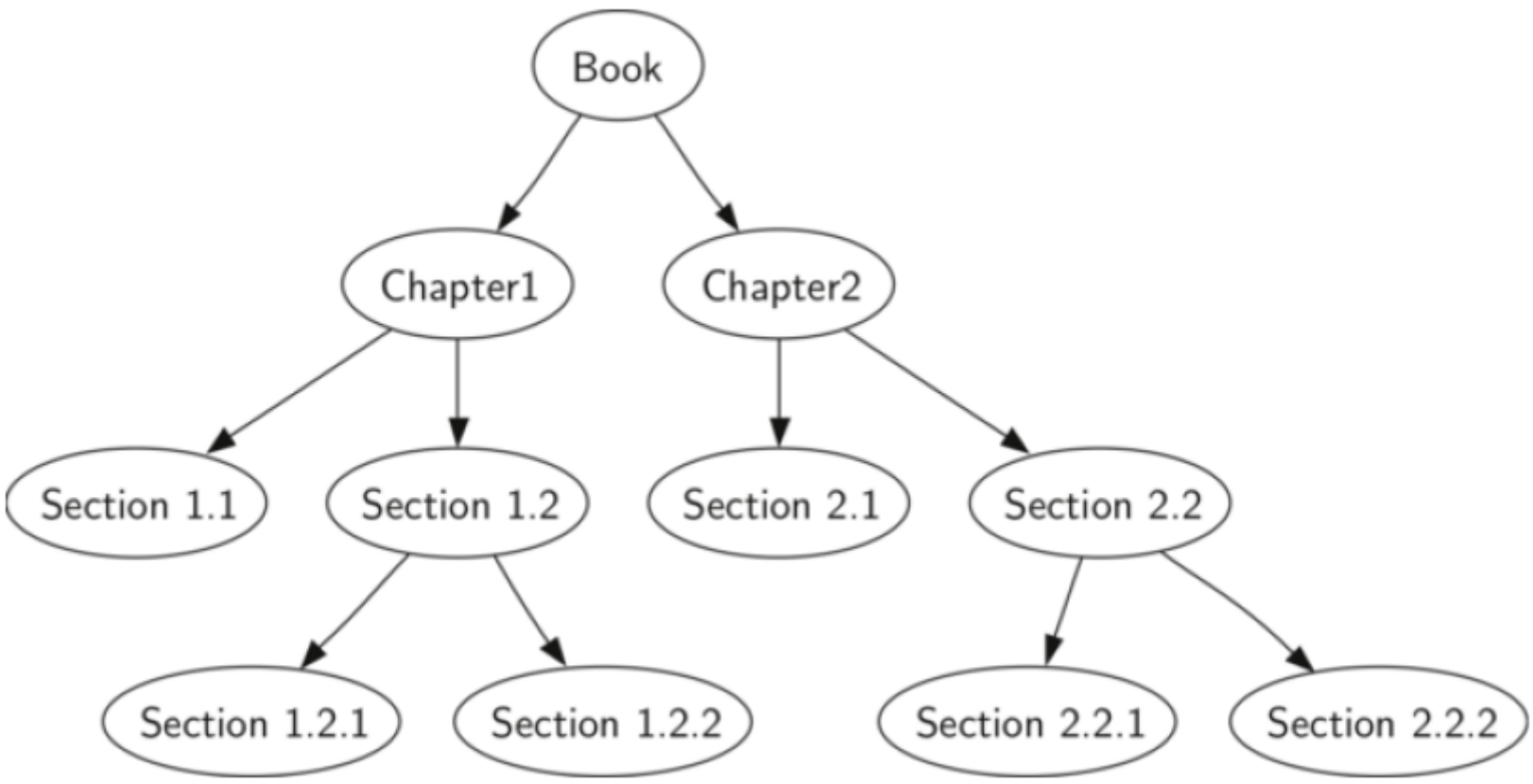
1. Если текущий токен ‘(’, то добавляем новый узел в качестве левого дочернего узла текущего узла и спускаемся в новый узел.
2. Если текущий токен содержится в списке [‘+’,‘–’,‘/’,‘*’], то устанавливаем значение в текущем узле, соответствующее оператору, представленному в токене. Добавляем новый узел в качестве правого дочернего узла текущего узла и спускаемся в него.
3. Если текущий токен является числом, то устанавливаем значение в текущем узле соответствующее числу в токене и переходим к родительскому узлу.
4. Если текущий токен ‘)’, то переходим к родителю текущего узла.

Разбор математического выражения (3 + (4 * 5)) или [‘(’, ‘3’, ‘+’, ‘(’, ‘4’, ‘*’, ‘5’ ,‘)’, ‘)’] и построение на его основе двоичного дерева:



Для вычисления значения выражения имеющего древовидное представление достаточно реализовать простую рекурсивную функцию.

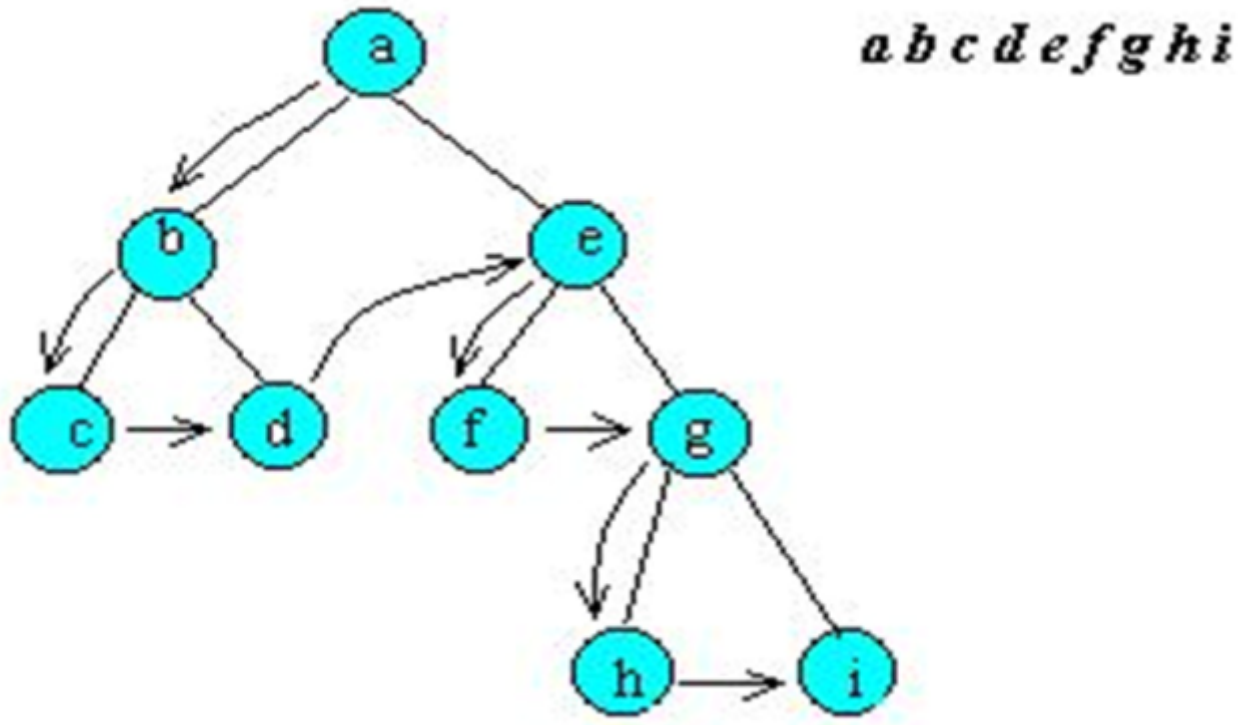
Представление документа (книги) как дерева:



Обход (traversing/walk) бинарных деревьев

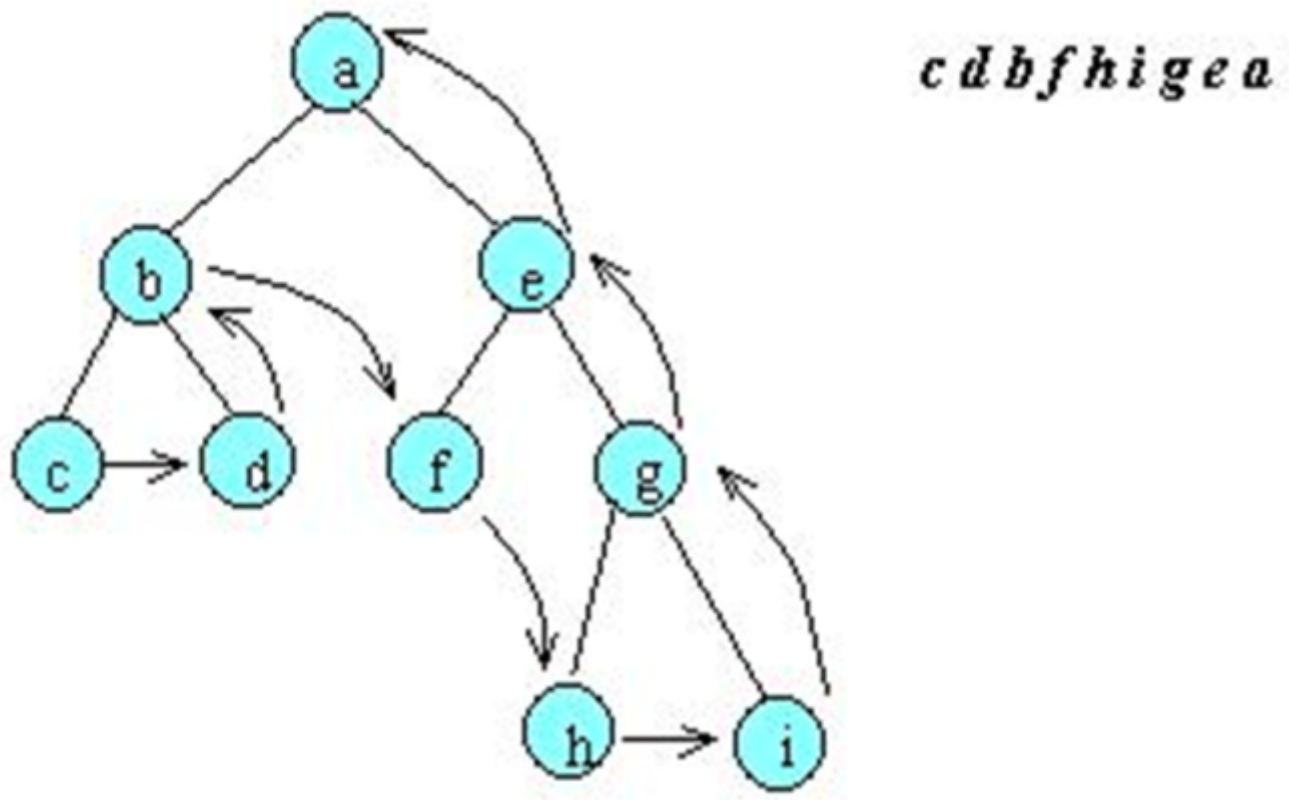
Прямой (preorder) порядок обхода дерева:

1. Первым просматривается корневой узел
2. Затем производится рекурсивный прямой обход левого поддерева.
3. Затем производится рекурсивный прямой обход правого поддерева.



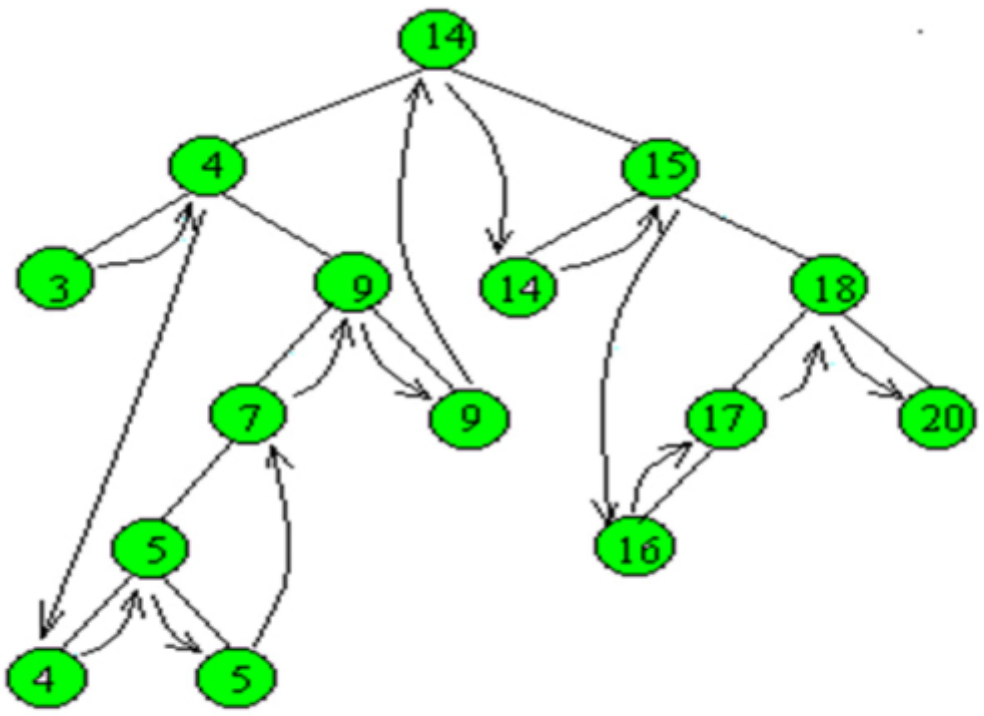
Обратный (postorder) порядок обхода дерева:

1. Первым производится рекурсивный обратный обход левого поддерева.
2. Затем производится рекурсивный обратный обход правого поддерева.
3. Затем просматривается корневой узел



Симметричный (inorder) порядок обхода дерева:

1. Первым производится рекурсивный симметричный обход левого поддерева.
2. Затем просматривается корневой узел
3. Затем производится рекурсивный симметричный обход правого поддерева.



Прохождение в симметричном порядке:
3, 4, 4, 5, 5, 7, 9, 9, 14, 14, 15, 16, 17, 18, 20

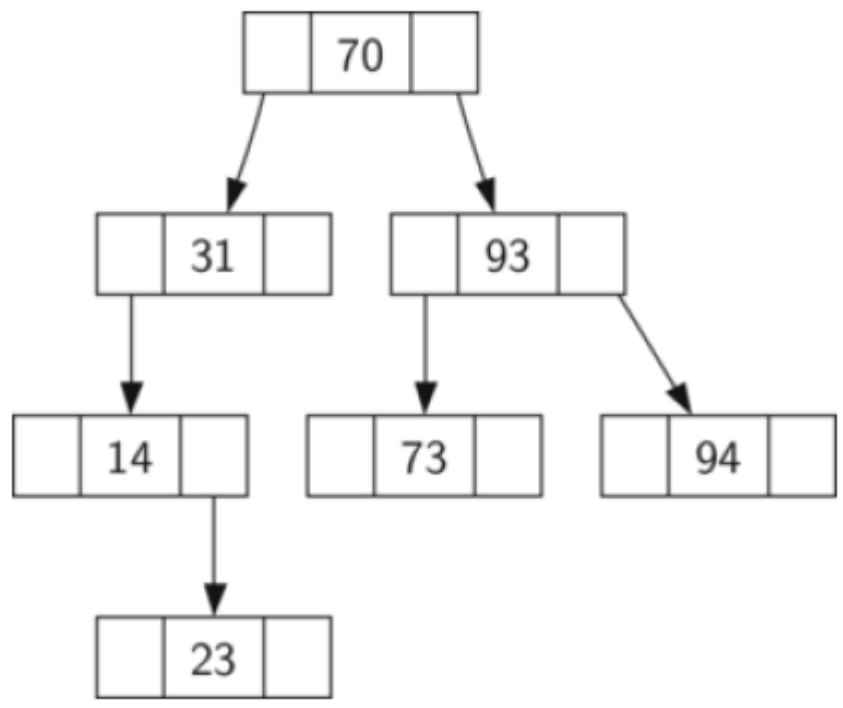
Двоичное дерево поиска

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

1. Оба поддерева — левое и правое — являются двоичными деревьями поиска.
2. У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
3. У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равно, нежели значение ключа данных самого узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения (например, операция меньше).

Пример двоичного дерва поиска:



Основные операции в бинарном дереве поиска выполняются за время, пропорциональное его высоте. Для полного бинарного дерева с n узлами эти операции выполняются за время O(lg n) в наихудшем случае. Математическое ожидание высоты построенного случайным образом бинарного дерева равно O (lg n), так что все основные операции динамическим множеством в таком дереве выполняются в среднем за время Θ(lg n).

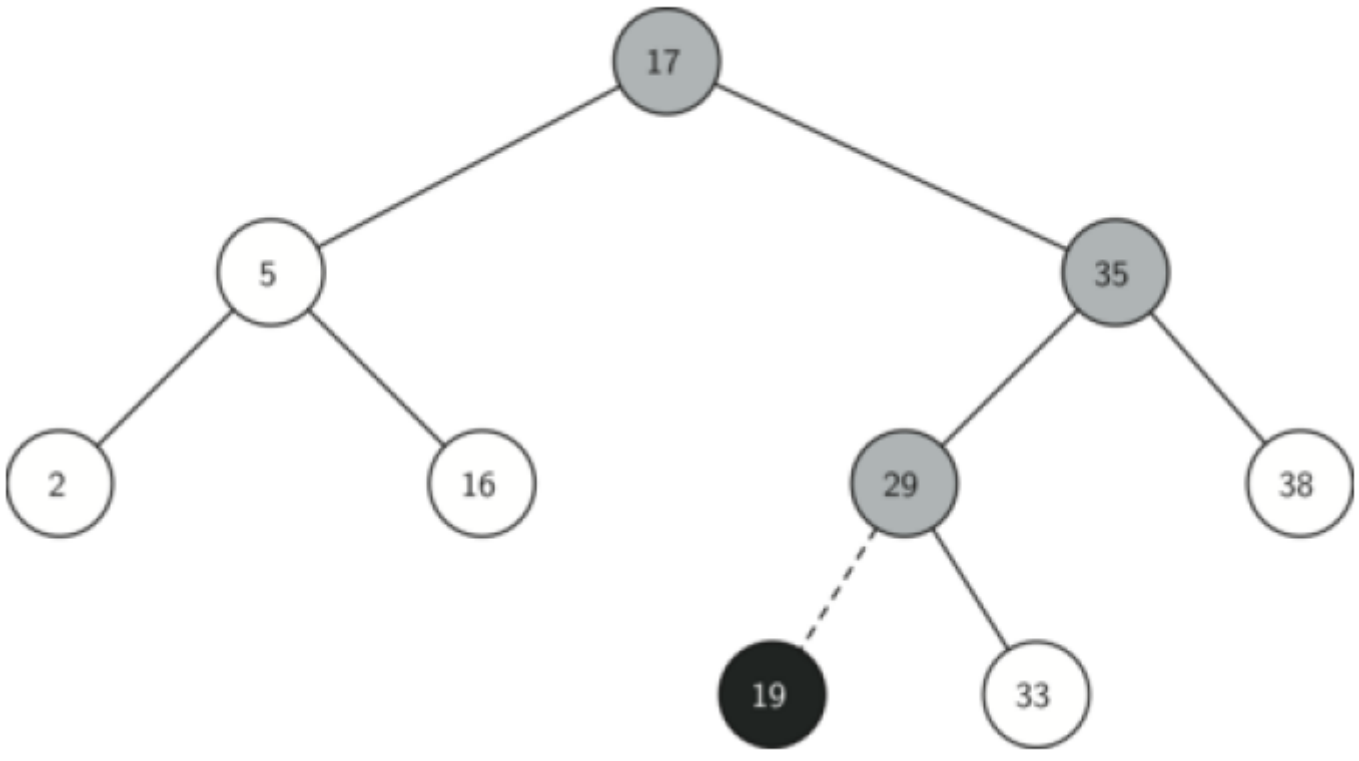
На практике мы не всегда можем гарантировать случайность построения бинарного дерева поиска, однако имеются версии деревьев, в которых гарантируется хорошее время работы в наихудшем случае. Речь идет о деревьях сбалансированных по высоте по определенным критериям, это АВЛ –деревья, 2-3, 3-4 деревья, красно-черные деревья, высота которых определяется как O (lg n).

Алгоритм вставки нового узла в двоичное дерево поиска:

1. Начинаем просмотр с корня деерва (первый текущий узел - корень).
2. Сравниваем значение нового узла со значением в текущем узле. Если значение в новом узле меньше, то продолжаем поиск в левом поддереве (текущим узлом становется левый дочерений узел предыдущего текущего узла). Если значение в новом узле больше чем в текущем узле, то продолжаем поиск в правом поддереве.

3. Если левого или правого поддерва не существует, то мы обнаружили место для вставки нвого элемента. На это место вставляется новый узел дерева.

Пример вставки узла в двоичное дерево поиска:



In [18]:
```python
# Приер реализации поика элемента в двоичном дереве поиска:
def get(self,key):
    if self.root:
        res = self._get(key, self.root)
        if res:
            return res.payload
        else:
            return None
    else:
        return None

def _get(self, key, current_node):
    if not current_node:
        return None
    elif current_node.key == key:
        return current_node
    elif key < current_node.key:
        return self._get(key, current_node.left_child)
    else:
        return self._get(key, current_node.right_child)
```

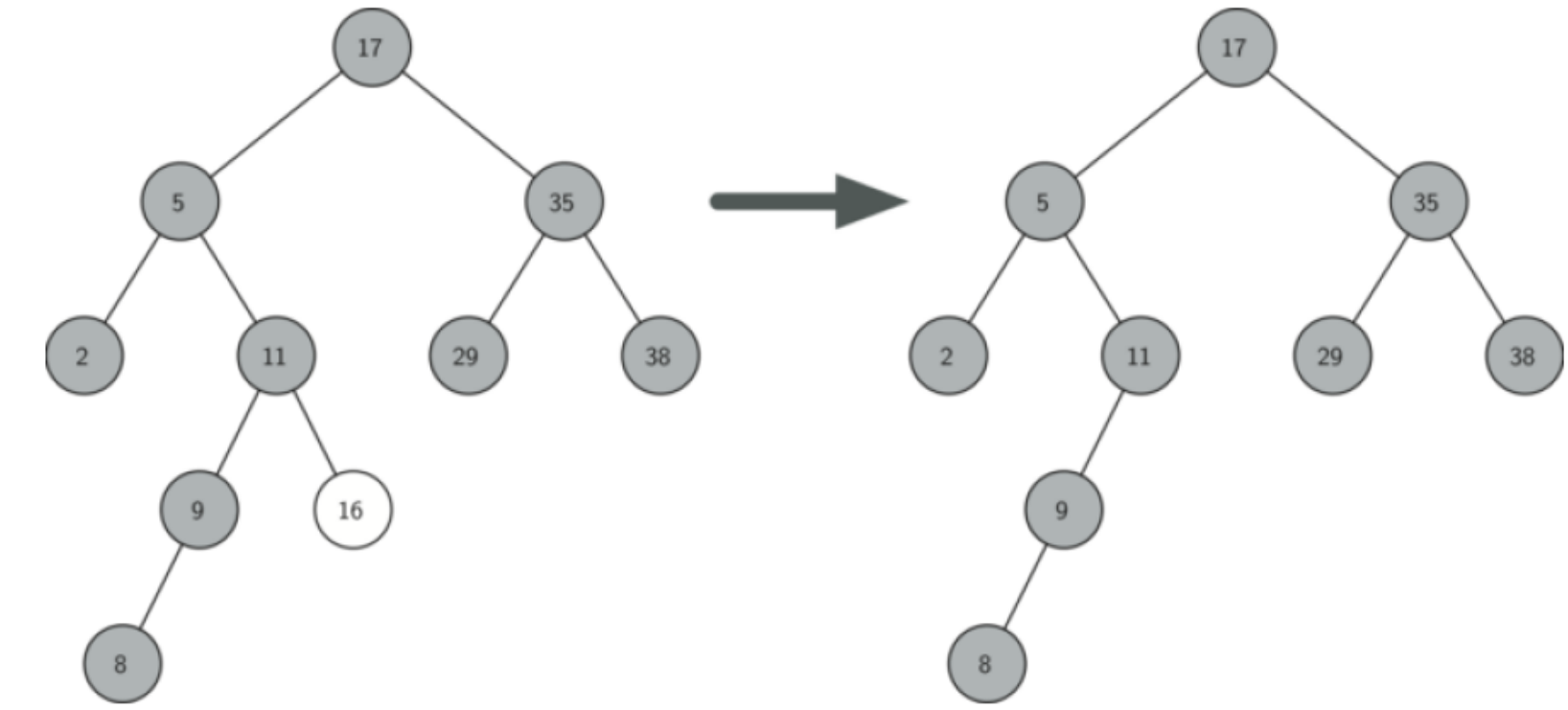
In [1]:
```python
def __getitem__(self, key):
    return self.get(key)

def __contains__(self, key):
    if self._get(key, self.root):
        return True
    else:
        return False
```

Наиболее сложной операцией является удаление узла из дерева:

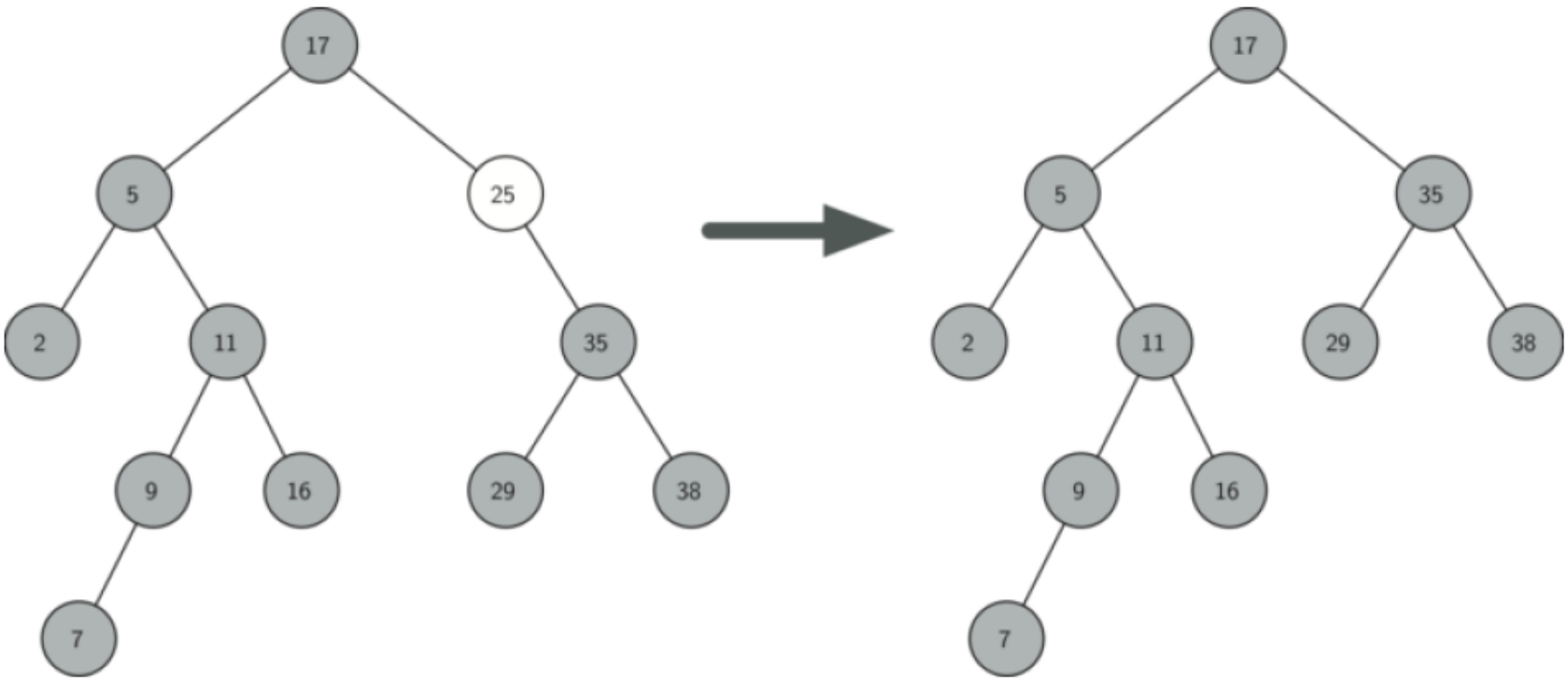
Шаг 1. При помощи поиска по дереву найти узел который нужно удалить.

Шаг 2. После обнаружения узла существует три случая, которые требуют специфической реализации операции удаления узла.

Случай 1. Удаляемый узел не имеет потомков:

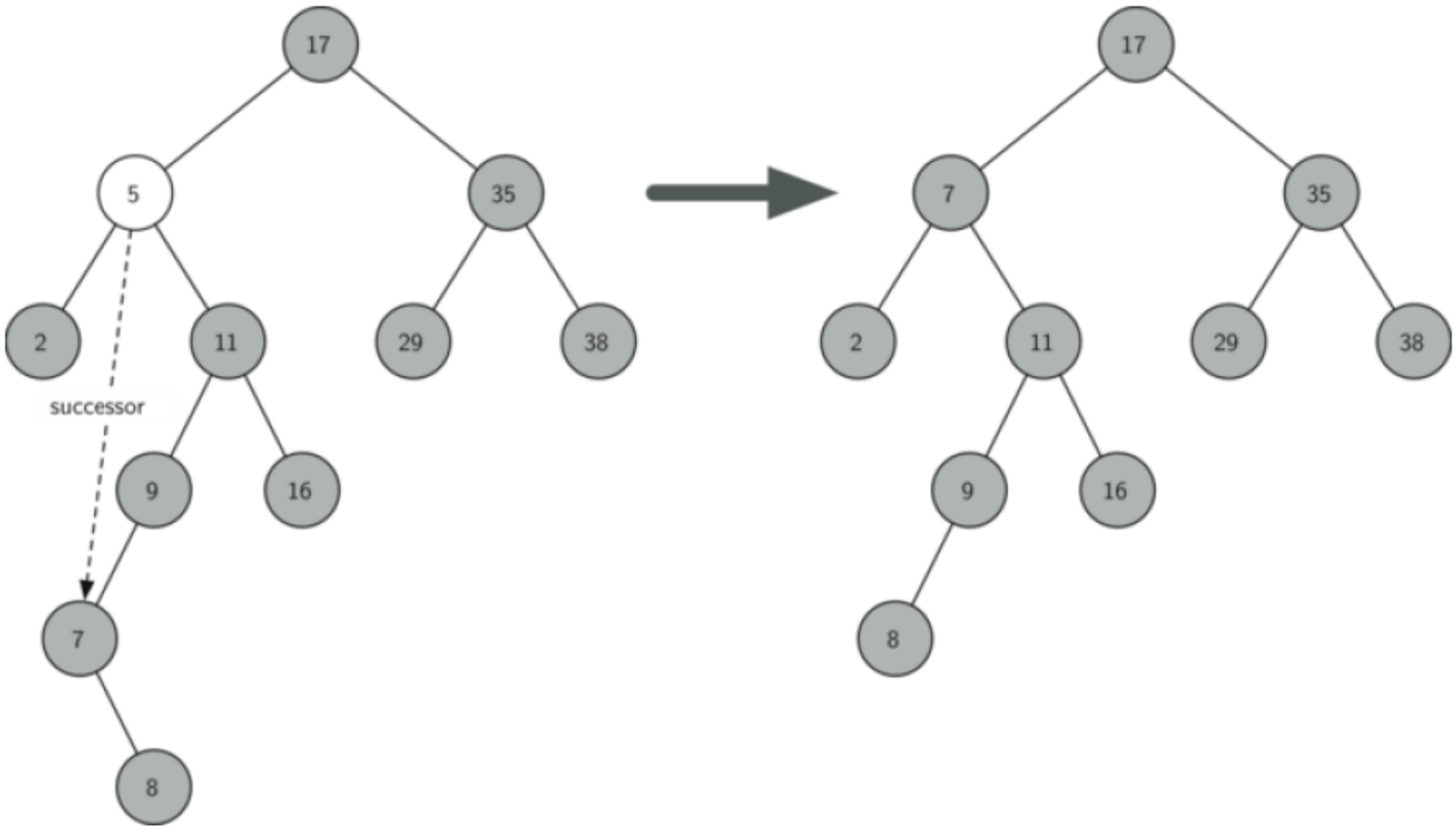


Случай 2. Удаляемый узел имеет только одного потомка:



1. Если текущий узел является левым дочерним узлом, то у его дочернего узла нужно обновить ссылку на родителя, так, чтобы она вела на родителя текущего узла. Связь родителя текущего узла с левым дочерним узлом изменится так, что он будет ссылался на дочерний узел текущего узла.
2. Если текущий узел является правым дочерним узлом, то у его дочернего узла нужно обновить ссылку на родителя, так, чтобы она вела на родителя текущего узла. Связь родителя текущего узла с правым дочерним узлом изменится так, что он будет ссылался на дочерний узел текущего узла.
3. Если у текущего узла нет родительского узла (т.е. он является корневым узлом), то его единственный потомок станет новым корневым узлом.

Случай 3. Удаляемый узел имеет двух потомков:

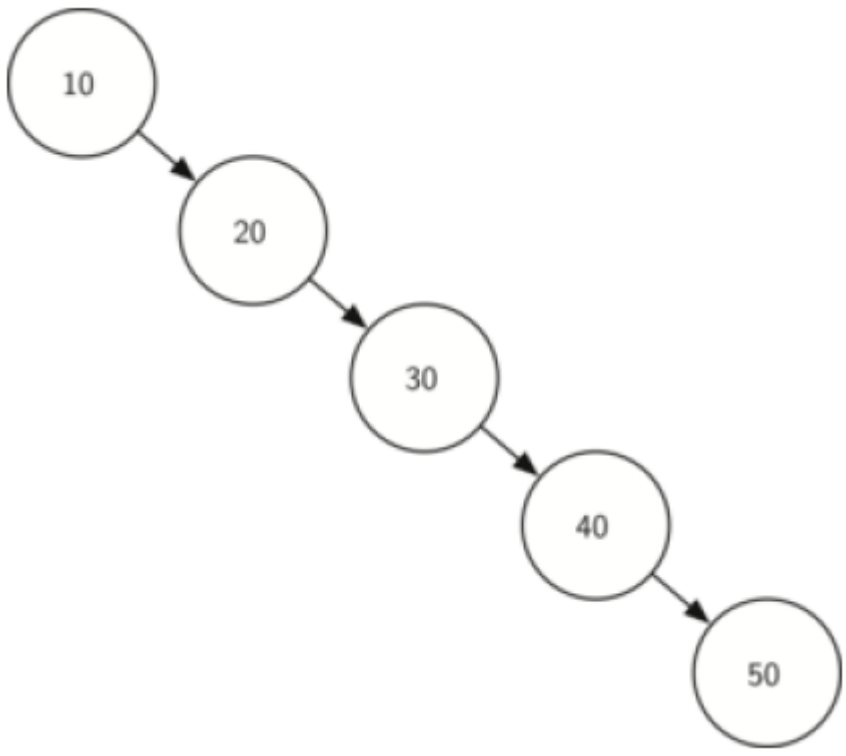


Если у удаляемого узла Z два дочерних узла, то мы находим следующий за ним по величине узел Y, у которого не более одного дочернего узла и убираем его из позиции, где он находился ранее, путем создания новой связи между его родителем и потомком, и заменяем им узел Z.

Поиск Y осуществляется следующим образом:

1. Переходим из Z в его правое поддерево.
2. Двигаемся вниз по левому поддерву, пока не встретим узел Y, у котрого нет левого дочернего узла (у Y может быть только один правый дочерний элемент, либо может не быть дочерних элементов вовсе). Y - искомый узел.

Пример неэффективного двоичного дерева поиска:



Алгоритм вставки в бинарное дерево, который мы только что рассмотрели, дает хорошие результаты при использовании случайных входных данных, но все же существует неприятная возможность того, что при этом будет построено вырожденное дерево. Можно было бы разработать алгоритм, поддерживающий дерево в оптимальном состоянии все время, где под оптимальностью мы в данном случае понимаем сбалансированность дерева.

Идеально сбалансированным называется дерево, у которого для каждой вершины выполняется требование: число вершин в левом и правом поддеревьях различается не более чем на 1.

Поддержка идеальной сбалансированности, к сожалению, очень сложная задача. Другая идея заключается в том, чтобы ввести менее жесткие критерии сбалансированности и на их основе предложить достаточно простые алгоритмы обеспечения этих критериев.

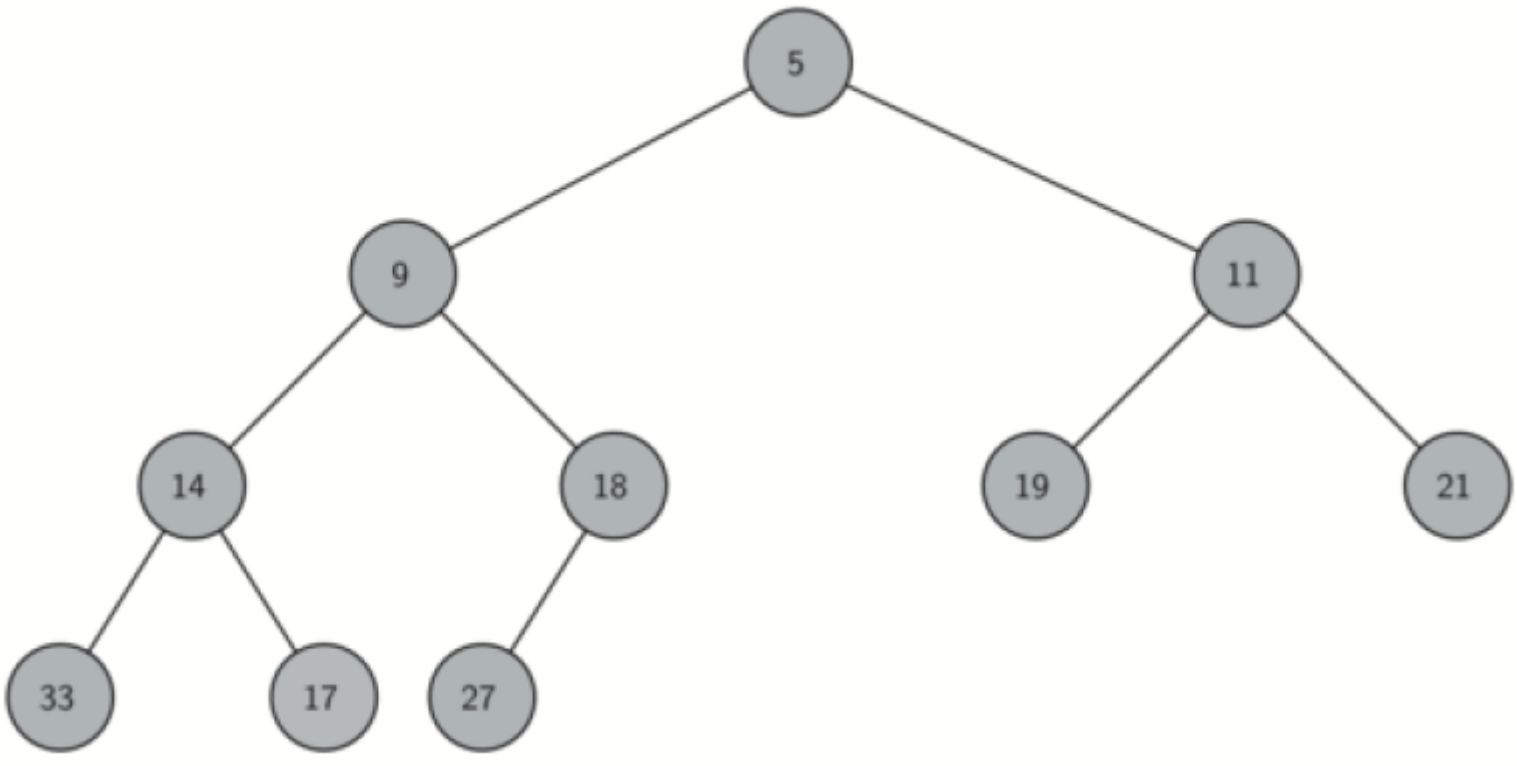
Двоичные кучи и очереди с приоритетом

Двоичная куча (пирамида) (binary heap) — такое двоичное дерево, для которого выполнены три условия:

1. Значение в любой вершине не больше, чем значения её потомков.
2. Уровень всех листьев (расстояние до корня) отличается не более чем на 1.

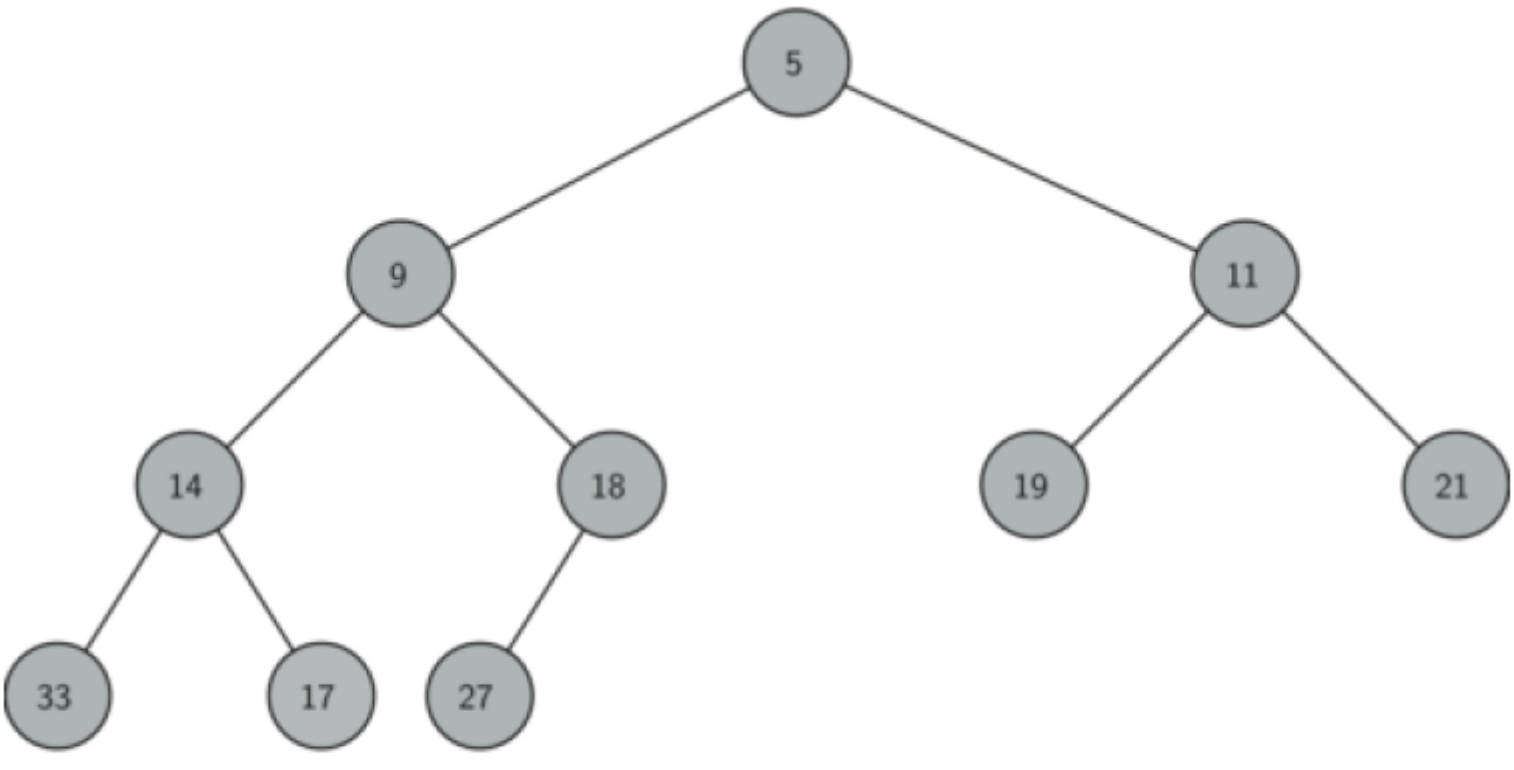
3. Последний уровень заполняется слева направо без «дырок».

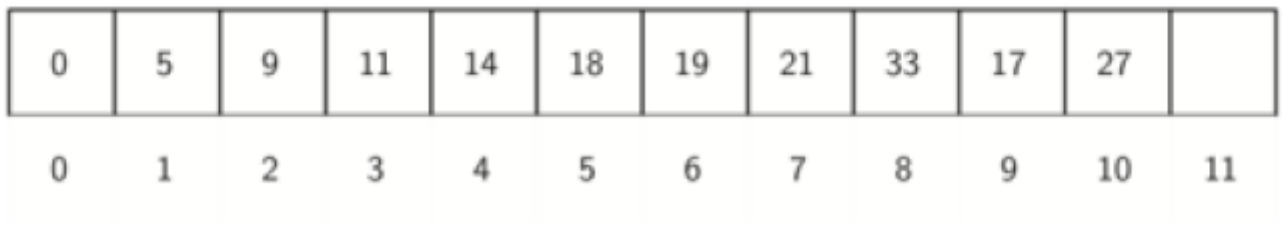
Пример двоичной кучи:



Для гарантированной логарифмической производительности мы дложны поддержвать двоичную кучу в виде сбалансированного дерева. Для этого мы будем строить двоичную кучу в виде полного бинарного дерева - дерева в котором каждый уровень кроме последнего содержит все возможные узлы, а последний уровень заполняется с лева на право без пропусков.

Пример двоичной кучи и ее представления в виде списка:





Важным свойством полного бинарного дерева является то, что такое дерево может быть представленно в виде одного списка. Левый потомок родителя (имеющего индекс p) является элеменотом списка с индексом 2p. Аналогично, правый потомок является элеменотом списка с индексом 2p + 1. Для того, чтобы найти индекс родительского узла нужно взять целую часть от индекса элемента, разделенного на 2.

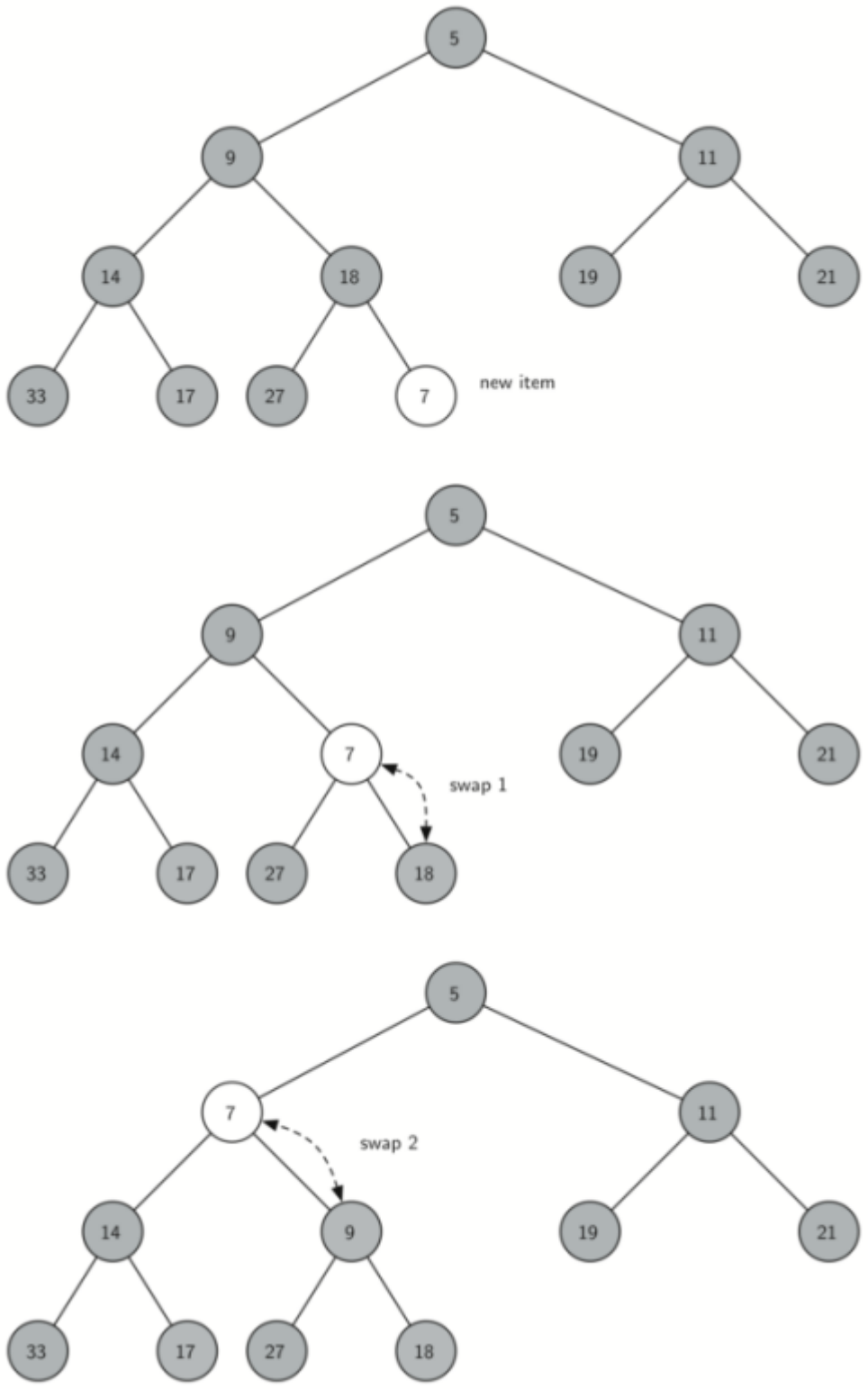
Базовые операции для двоичной кучи:

- BinaryHeap() # создает новую пустую бинарную кучу
- insert(k) # добавить элемент в кучу, сложность $O(\log n)$
- find_min() # возввратить минимальный элемент в куче, сложность $O(1)$
- del_min() # возввратить минимальный элемент и исключить его из кучи, сложность $O(\log n)$
- is_empty() # возвращает True если куча пуста и False в обратном случае
- size() # количество элементов в куче
- build_heap(list) # создает новую кучу на основе произвольного (не упроядоченного) массива, сложность $O(n)$

Отсортировать массив путём превращения его в кучу, а кучи в отсортированный массив. Время работы $O(n \log n)$

In [ ]:
```
class BinHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0
```

Вставка нового узла и просачивание его наверх:



In [23]:
```
class BinHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size = self.current_size + 1
        perc_up(self, self.current_size)
```

In [24]:
```
def perc_up(self, i):
    while i // 2 > 0:
        if self.heap_list[i] < self.heap_list[i // 2]:
            tmp = self.heap_list[i // 2]
            self.heap_list[i // 2] = self.heap_list[i]
            self.heap_list[i] = tmp
        else:
            break
        i = i // 2
```

Реализация метода del_min:

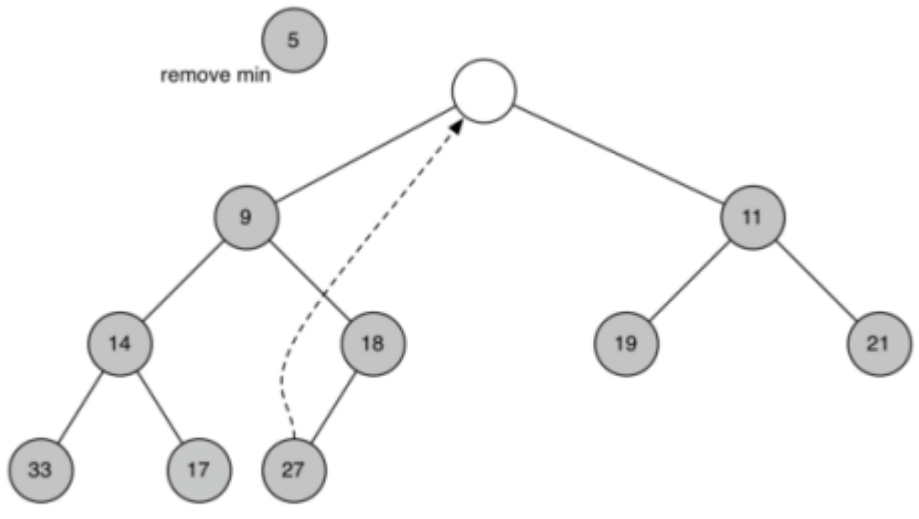
The heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy.

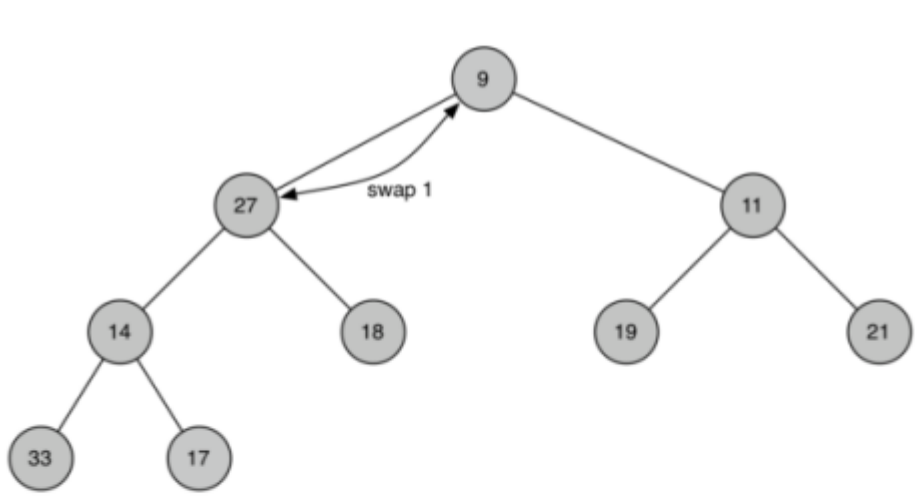
Свойства кучи требуют, чтобы корень дерева был наименьшим элементом в дереве. Таким образом найти наименьший элемент просто.

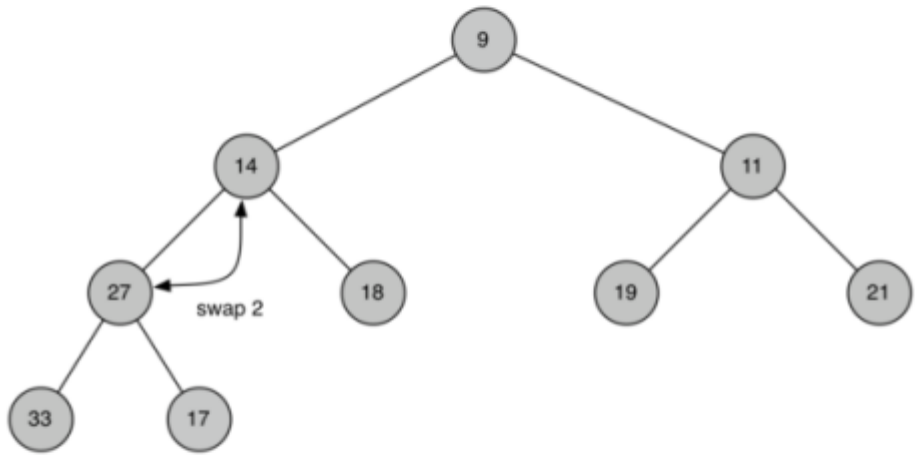
Сложной частью операции del_min является восстановление корректной структуры кучи и ее свойств после удаления корневого элемента.

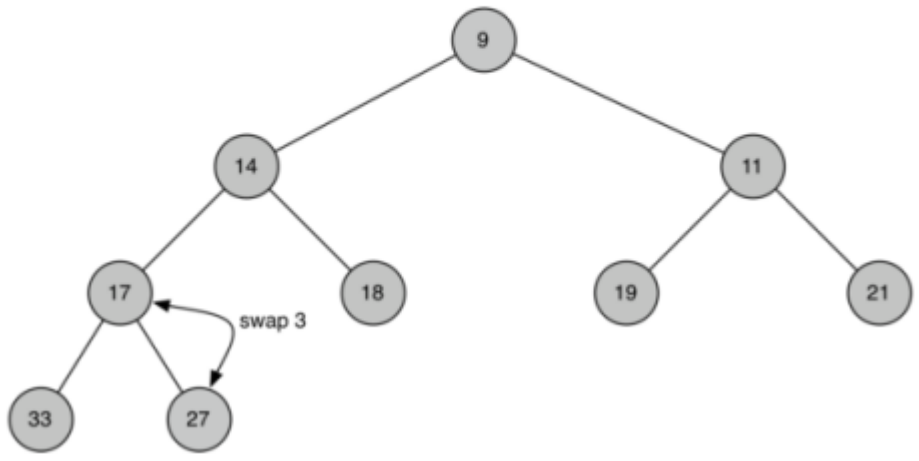
1. Мы восстановим корневой элемент установив на его место элемент, изъятый из последней позиции кучи (последний элемент в представлении кучи в виде списка). Таким образом будет получена корректная структура дерева для кучи, но при этом может быть нарушено правило порядка расположения элементов в куче.
2. Восстановление порядка элементов в куче будет выполнено за счет "проталкивания" нового корневого элемента на корректную позицию. При "проталкивании" элемент при необходимости будет опускаться на место своего наименьшего потомка.

Удаление корневого элемента и просачивание нового значения вниз:









In [33]:
```python
def perc_down(self, i):
    while (i * 2) <= self.current_size:
        mc = min_child(self, i)
        if self.heap_list[i] > self.heap_list[mc]:
            tmp = self.heap_list[i]
            self.heap_list[i] = self.heap_list[mc]
            self.heap_list[mc] = tmp
        else:
            break
        i = mc
```

In [34]:
```
def min_child(self, i):
    if i * 2 + 1 > self.current_size:
        return i * 2
    else:
        if self.heap_list[i * 2] < self.heap_list[i * 2 + 1]:
            return i * 2
        else:
            return i * 2 + 1
```

In [38]:
```
class BinHeap:
    def __init__(self):
        self.heap_list = [0]
        self.current_size = 0

    def insert(self, k):
        self.heap_list.append(k)
        self.current_size = self.current_size + 1
        perc_up(self, self.current_size)

    def del_min(self):
        ret_val = self.heap_list[1]
        self.heap_list[1] = self.heap_list[self.current_size]
        self.current_size = self.current_size - 1
        self.heap_list.pop()
        perc_down(self, 1)
        return ret_val

    def build_heap(self, a_list):
        i = len(a_list) // 2
        self.current_size = len(a_list)
        self.heap_list = [0] + a_list[:]
        while (i > 0):
            self.perc_down(i)
            i = i - 1
```