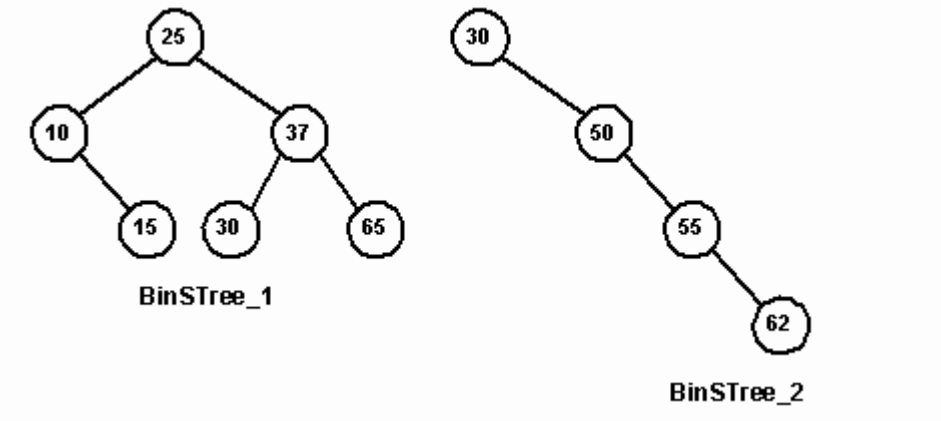
Двоичное дерево поиска

Двоичный поиск, описанный выше, применяется для поиска в сортированных линейных структурах. Однако, как вы уже знаете, структуру бывают нелинейными. Одна из таких структур - бинарные деревья. Интуитивно кажется, что двоичный поиск и бинарные деревья как-то можно совместить. И это абсолютно так. Комбинация этих двух подходов называется **двоичным деревом поиска**. Не всякое двоичное дерево является двоичным деревом поиска. Оно должно обладать некоторыми свойствами:

1. Оба поддерева каждого узла являются двоичными деревьями поиска
2. Для узла с ключом X все узлы левого поддерева должны быть строго меньше X
3. Аналогично, для узла с ключом X все узлы правого поддерева должны быть строго больше X



Источник: http://www.k-press.ru/cs/2000/3/trees/trees.asp

Такое представление данных имеет преимущество над остальными, если основная операция - поиск элемента, потому что сложность этой операции O(log(n)) - и это очень хорошо. Операции вставки и удаления требуют в среднем также O(log(n)) времени. Такая сложность объясняется тем, что при поиске или вставке мы всегда идем в глубину дерева - сравнивая искомое значение с ключом узла мы идем или в левое дерево, или в правое. На каждом этапе оставшийся размер дерева, который нужно просмотреть, сокращается (в среднем) в 2 раза.

Напишем класс BinarySearchTree, реализующий двоичное дерево поиска.

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

    def __str__(self): # печать с помощью обхода в ширину
        queue = [self] # создаем очередь
        values = [] # значения в порядке обхода в ширину
        while queue != []: # пока она не пустая
            last = queue.pop(0) # извлекаем из начала
```

```
            if last is not None: # если не None
                values.append("%d" % last.value) # добавляем значение
                queue.append(last.left_child) # добавляем левого потомка
                queue.append(last.right_child) # добавляем правого потомка
        return ''.join(values)
```

Какие методы должен реализовать этот класс?

1. Поиск элемента
2. Поиск минимума/максимума
3. Поиск следующего элемента
4. Вставка элемента
5. Удаление элемента
6. Построение дерева по массиву

Можно расширять этот функционал еще многими полезными методами (такие как повороты, балансировки и т.д.), но описанные методы являются базовыми.

Для начала реализуем поиск элемента в двоичном дереве поиска. Мы знаем, что благодаря его свойствам, мы можем осуществлять поиск только в левом или правом дереве, пока ключ узла не совпадет с искомым элементом.

```
def search(self, x):
    if self.value == x: # если нашли элемент,
        return self # возвращаем ссылку на узел
    elif x < self.value: # или, если значение меньше ключа, продолжаем
        return self.left_child.search(x) # поиск в левом поддереве
    elif x > self.value: # иначе в правом
        return self.right_child.search(x)
    else: # если такое значение не нашлось,
        return False # возвращаем False
```

Здесь мы использовали рекурсивный подход, запуская функцию поиска каждый раз в левом или правом дереве до тех пор, пока не найдем нужный ключ или не столкнемся с тем, что такого ключа в дереве нет.

Похожим образом можно реализовать поиск минимума в двоичном дереве поиска. Ключевое отличие от поиска произвольного элемента заключается в том, что мы должны каждый раз “идти” только в левое поддерево (по свойствам именно в левых поддеревьях находятся элементы меньше ключа родительского узла).

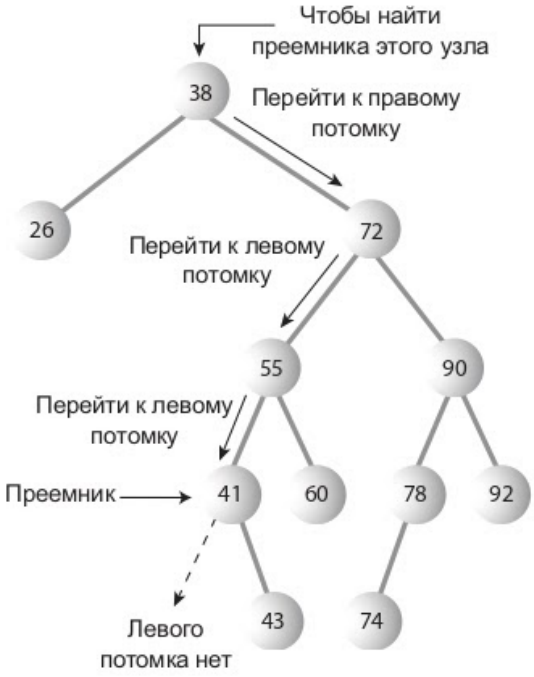
```
def minimum(self):
    if self.left_child is None:
        return self
    else:
        return self.left_child.minimum()
```

Задание 1. Реализуйте функцию поиска максимума в двоичном дереве поиска по аналогии с написанной функцией поиска минимума.

Реализуем теперь функцию поиска следующего элемента. На вход этой функции подается исходный ключ, по которому нужно найти следующий. Поиск начнем с корня дерева и будем спускаться вниз. Дополнительно будем хранить две переменные - *current* с текущей ячейкой и *successor*, в которой будем хранить последний элемент, удовлетворяющий условию *successor.value* > *x.* Рассмотрим текущий узел *current*. Если current.value <= x, то искомый элемент находится в правом поддереве. Иначе - в левом поддереве, но узел этого поддерева может быть искомым узлом *successor*, поэтому обновляем значение этой переменной. Таким образом получается, что мы шагаем то в правое поддерево, то в левое до тех пор, пока не дойдем до нулевого указателя на дочерний элемент.

```
def next_value(self, x):
    current = self
    successor = None
    while current is not None:
        if current.value > x:
            successor = current
            current = current.left_child
        else:
            current = current.right_child
    return successor
```

Графически это можно изобразить следующим образом:



Источник: https://sohabr.net/habr/post/442352/

Задание 2. Реализуйте функцию поиска предыдущего элемента.

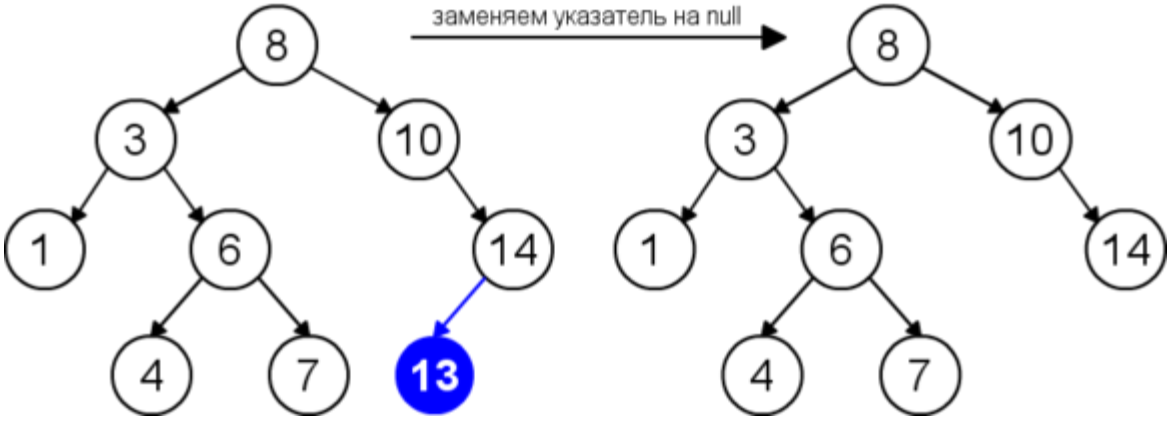
А теперь начинаем самое интересное - вставка и удаление элементов. Обе эти операции могут поменять структуру дерева, поэтому при их совершении необходимо помнить о свойствах двоичного дерева поиска.

Реализуем вставку нового узла рекурсивным способом. Условие выхода из рекурсии - это отсутствие одного или обоих потомков. Если же потомки присутствуют, то нужно установить в каком поддереве должно находиться вставляемое значение.

```
def insert(self, x):
    if x > self.value: # идем в правое поддерево
        if self.right_child is not None: # если оно существует,
            self.right_child.insert(x) # делаем рекурсивный вызов
        else: # иначе создаем правого потомка
            self.right_child = BinarySearchTree(x)
    else: # иначе в левое поддерево и делаем аналогичные действия
        if self.left_child is not None:
            self.left_child.insert(x)
        else:
            self.left_child = BinarySearchTree(x)
    return self # возвращаем корень
```

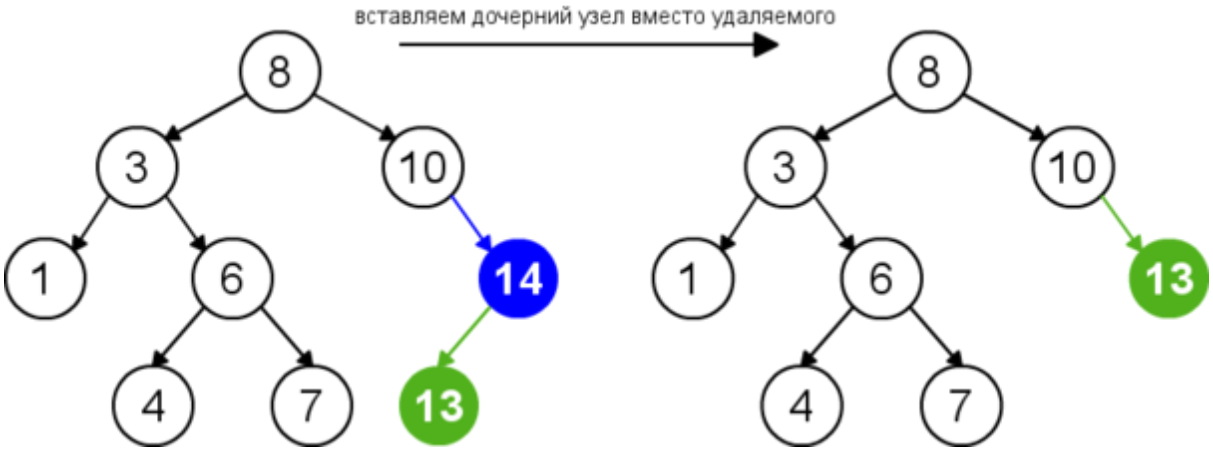
Для удаления мы должны рассмотреть несколько случаев:

1. Удаляемый узел не имеет потомков (является листом).



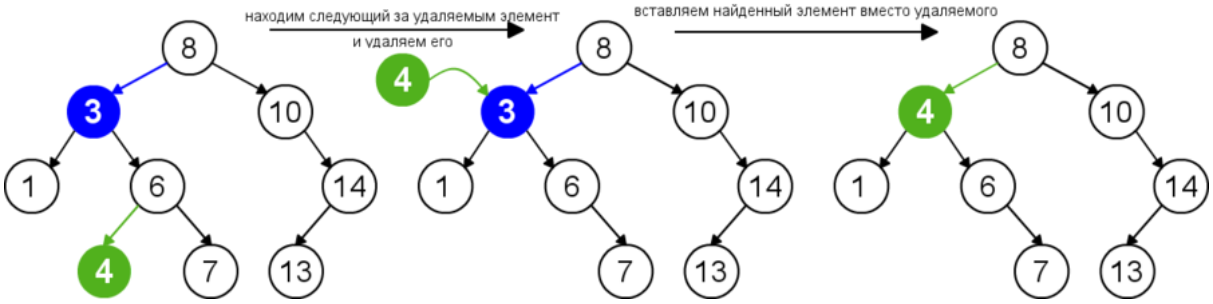
Источник: Дерево поиска, наивная реализация

2. Удаляемый узел имеет одного потомка. Тогда мы должны восстановить структуру дерева, создав связь от предка удаляемого узла к его потомку



Источник: [Дерево поиска, наивная реализация]

3. Удаляемый узел имеет двух потомков. Тогда мы должны найти следующий элемент за удаляемым, чтобы поставить его на место удаляемого. Затем рекурсивно вызвать функцию удаления для уже этого узла.



Источник: [Дерево поиска, наивная реализация]

Данный алгоритм, пожалуй, является одним из самых сложных. Очень рекомендуется представить все возможные случаи самостоятельно и опробовать их с помощью написанной функции.

```
def delete(self, x):
    # первым этапом мы должны найти удаляемый узел и его предка
    parent = self
    node = self
    if not self.search(x):
        return self
    while node.value != x:
        parent = node
        if parent.left_child is not None and x < parent.value:
            node = parent.left_child
        elif parent.right_child is not None and x > parent.value:
            node = parent.right_child
    # по завершении в node хранится искомый узел

    # первый случай - если лист
    if node.left_child is None and node.right_child is None:
```

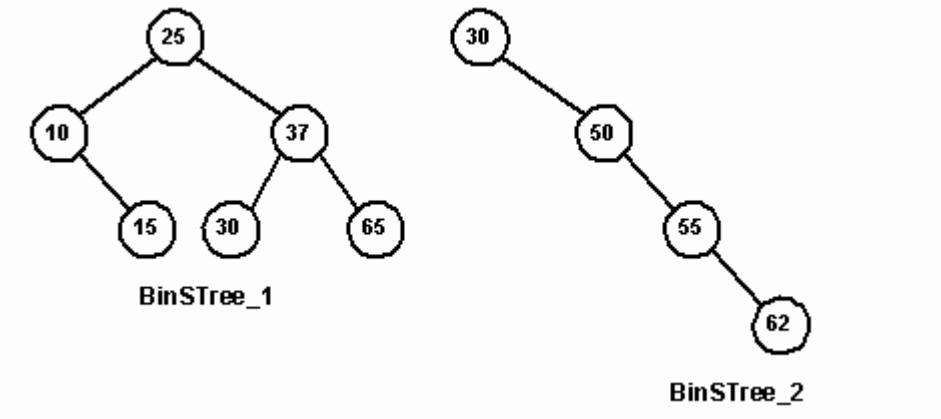
```
        if parent.left_child is node:
            parent.left_child = None
        if parent.right_child is node:
            parent.right_child = None
        if parent.value == x:
            # если нет листов и parent==node до сих пор,
            # значит, нужно вернуть None для корректной работы рекурсии
            return None
    # второй случай - имеет одного потомка
    elif node.left_child is None or node.right_child is None:
        if node.left_child is not None:
            if parent.left_child is node:
                parent.left_child = node.left_child
            elif parent.right_child is node:
                parent.right_child = node.left_child
        if node.right_child is not None:
            if parent.left_child is node:
                parent.left_child = node.right_child
            elif parent.right_child is node:
                parent.right_child = node.right_child
    else: # третий случай - имеет двух потомков
        next_ = node.next_value(x).value # ищем следующее значение
        node.value = next_ # меняем на него
        # делаем рекурсивный вызов
        node.right_child = node.right_child.delete(next_)
    return self
```

Алгоритм действительно непростой, поэтому разберем его еще раз по шагам:

1. С помощью цикла while ищем узел *node*, подлежающий удалению, а также его предка *parent*
2. Если найденный узел является листом, то удаляем его, присваивая значение None соотствующему левому (правому) поддереву предка.
3. Если найденный узел имеет одного потомка, то устанавливаем связь между ним и предком удаляемого узла. Этот единственный потомок становится на место удаленного узла.
4. Если найденный узел имеет сразу обоих потомков, то находим следующее значение за удаляемым и сохраняем его. Это значение становится на место удаленного. После чего, во избежание дублирования, нужно рекурсивно удалить новое значение. Элемент, стоящий на месте удаленного узла, является следующим, т.е. больше исходного, поэтому всегда находится в правом дереве. И именно для правого дерева мы запускаем эту же самую функцию.

Задание 3. Создайте двоичное дерево поиска BinSTree_1 согласно изображению. Затем последовательно удалите узлы со значением 37 и 25. В ответ введите вывод с консоли

```
print(BinSTree_1)
```



Источник: http://www.k-press.ru/cs/2000/3/trees/trees.asp

Нами остался незатронутым один очень важный вопрос - балансировка дерева. В среднем, высота дерева дерева равна примерно log(n). Такие деревья называются идеально сбалансированными - их высота является минимальной. Это оказывает огромное влияние на работу всех алгоритмов, связанных с двоичными деревьями поиска, так как их сложность в среднем равна O(log(n)). Однако деревья могут оказаться и несбалансированными, иногда очень сильно. Вырожденный случай достигается, когда на каждом уровне находится по одному элементу как в дереве BinSTree_2 с изображения выше. Эффективность работы с такими деревьями сильно снижается.

В связи с этим существует несколько подходов балансировки деревьев для достижения оптимальной высоты. Для ознакомления рекомендуются следующие статьи:

Красно-черное дерево:
https://habr.com/ru/post/330644/

АВЛ-дерево:
https://habr.com/ru/post/150732/

Декартово дерево:
Декартово дерево

Рандомизированное двоичное дерево поиска:
https://habr.com/ru/post/145388/

Ответы к заданиям в конце файла :)

Ответы к заданиям

Задание 1. Реализуйте функцию поиска максимума в двоичном дереве поиска по аналогии с написанной функцией поиска минимума.

```
def maximum(self):
    if self.right_child is None:
        return self
    else:
        return self.right_child.maximum()
```

Задание 2. Реализуйте функцию поиска предыдущего элемента.

```
def prev_value(self, x):
    current = self
    successor = None
    while current is not None:
        if current.value < x:
            successor = current
            current = current.right_child
        else:
            current = current.left_child
    return successor
```

Задание 3. Создайте двоичное дерево поиска BinSTree_1 согласно изображению. Затем последовательно удалите узлы со значением 37 и 25. В ответ введите вывод с консоли

Ответ: 30 10 65 15