Конспект лекций
Алгоритмы и структуры данных

Кисляков Иван Дмитриевич

МФТИ

Москва, 2023 г.

Глава 1

Бинарные деревья поиска

1.1 Несбалансированные бинарные деревья поиска

В данном параграфе предлагается начать разговор о такой структуре данных, как бинарные деревья поиска. Эта структура данных использует те же понятия и идеи, которые мы рассматривали в параграфе про структуру данных список или дек, поэтому настоятельно рекомендуется сначала ознакомиться с предыдущими темами перед началом прочтения данной главы.

Для начала, давайте разберемся, что же такое дерево в математике. Если вводить данное понятие строго, мы получим большое количество рекурсивных определений, в которых читатель может запутаться, поэтому немного пренебрежем чистотой математического изложения: точные определения можно найти в книгах по теории графов.

1.1.1 Немного о математике

def *Дерево* - набор вершин (узлов) и ребер (связей), соединяющих эти вершины, такой что не существует циклического пути по вершинам. Если говорить на языке математики: связный граф без циклов.
def *Путь* - упорядоченный набор вершин, соединенных друг с другом ребрами.

Дерево можно предстваить в виде корневой системы или генеалогического древа. Где в самом верху находится «самый важный элемент», который мы будем называть **корнем дерева**. У корня есть вершины, которые с ним соедиенены - назовем их дочерними вершинами для корня. Дочерняя вершина вершины v - вершина, куда можно попасть, перейдя единожды по ребру из вершины v.

На рисунке спарва мы видим изображение дерева. Вершина (или, иначе, узел) под номером 1 является **корнем** дерева.

Вершины 4, 5, 6 являются **дочерними** для вершины 2.

Вершины, которые не имеют «детей» называются **листьями дерева**.

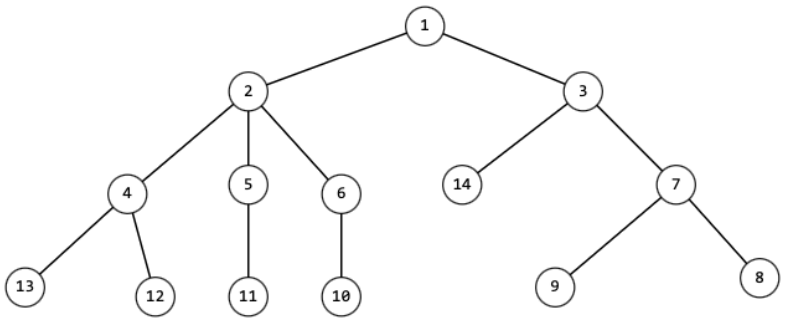


Рис. 1.1: Пример дерева

Также нельзя не заметить, что одно и то же дерево может выглядеть по-разному. Например, если «подвесить» дерево за вершину 2 (то есть переместить ее выше всех), будет ощущение, что корнем стала вершина 2 – но, как мы видим, от этого дерево ничуть не поменялось. В программировании мы не умеем «подвешивать» дерево за другие вершины, поэтому особенно не будем заострять на этом внимание.

def *Высотой дерева* назовем наибольшую длину пути из корня до листа. Чтобы понять, что же это такое, можно вспомнить изображение генеалогического древа и сообразить, что количество поколений родственникв, изображенных на древе и есть высота дерева. Для графа на изображении выше

высота дерева равна 4. Высоты листьев равны единицы. На втором уровне находятся вершины 4, 5, 6, 7. На третьем - 2, 3; на четвертом - 1.

Выведем формулу подсчета высоты дерева в вершине v: Высота вершины v равна наибольшей высоте ее дочерних вершин + 1.

1.1.2 Немного о структуре данных

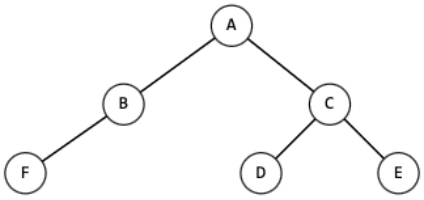
def *Бинарное дерево* - дерево, у каждой вершины которого имеется не более двух дочерних узлов.
def *Бинарное дерево поиска (Binary Search Tree - BST)* - структура данных (то есть алгоритм хранения данных), сохраняющая принципы бинарного дерева и имеющее набор свойств:

- Каждая вершина дерева имеет свое уникальное значение - данные, которые хранит структура BST.
- Для любой вершины v выполнено, что в левом поддереве вершины v хранятся значения, строго меньшие значения вершины v, а в правом поддереве хранятся вершины, имеющие значения строго большие значения вершины v.
  (Строго говоря: $\forall v$ - вершина, $v.left.value < v.value < v.right.value$)

По названию структуры данных можно понять набор тех методов, которые нам хотелось бы использовать в бинарном дереве поиска. Очевидно, это как минимум **Search(value)** - проверить наличие заданного значения в дереве. Далее становится понятно, что еще хочется добавлять (**Push(value)**) и удалять (**Remove(value)**) значения из дерева.

На самом деле мы хотим реализовать еще несколько вспомогательных и полезных методов, такие как: *FindMax(), FindMin(), RemoveMax(), RemoveMin(), Find(value) - возвращающая указатель на элемент со значением value.*

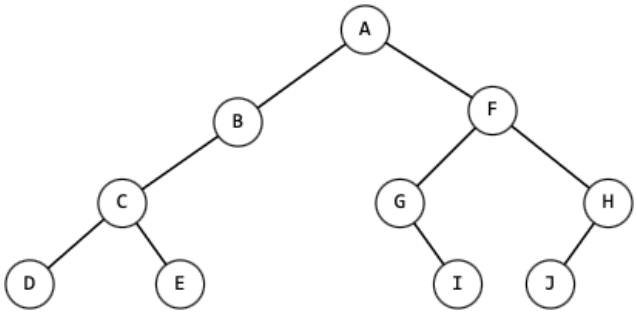
Для начала предлагается решить следующую задачу: **найти максимум** в бинарном дереве поиска.



Рассмотрим пример поддерева на изображении слева. У вершины A есть два дочерних узла - B и C. Мы видим, что вершина C находится в правом поддереве вершины A, а значит значение в вершине C больше значения в вершине A в силу свойств BST. Значит мы можем смело перейти в вершину C и искать максимум уже среди поддеревьев вершины C. Применив аналогичный алгоритм мы окажемся в правом поддереве вершины C, в вершине E. Но у этого узла нет правой дочерней вершины, из чего мы делаем вывод, что вершина E (крайняя правая вершина) является максимальной в нашем поддереве. *Очевидно также находится минимум в дереве.*

1.1.3 Обходы дерева

Для того, чтобы пройти по всем вершинам дерева и сделать с ними некоторые действия, необходимо придумать алгоритм обхода дерева. Существует две глобальные идеи, как можно обходить вершины дерева: вглубь (**DFS**) и вширину (**BFS**). В данном параграфе мы рассмотрим только **обход вглубину (DFS)**, а если быть точнее: три его возможных варианта.



1. pre-order *(обход сверху-вниз)* Посещает сначала текущую вершину, затем спускается в левое поддерево, а после прохождения всего левого поддерева, спускается в правое поддерево. Выполняется

рекурсивно для каждой вершины.
Для лучшего понимания, приведем псевдокод:

```
def PreOrder(node):
    if node is not nullptr:
        print(node.value)
        PreOrder(node.left)
        PreOrder(node.right)
```

Любой обход необходимо начинать с некоторой вершины. В стандартном случае мы начинаем обход дерева с корня (root). Например, для дерева, изображенного на схеме выше можно представить, как будет выглядеть обход дерева pre-order, запущенный из корневой вершины:

A, B, C, D, E, F, G, I, H, J

2. post-order *(обход снизу-вверх)*

```
def PostOrder(node):
    if node is not nullptr:
        PostOrder(node.left)
        PostOrder(node.right)
        print(node.value)
```

Сначала спускается в левое поддерево, после прохождения всего левого поддерева, спускается в правое поддерево, и только после этого посещает текущую вершину. Выполняется рекурсивно для каждой вершины. Для схемы выше, обход дерева post-order, запущенный из корневой вершины:
D, E, C, B, I, G, J, H, F, A

3. in-order *(обход слева-направо)*

```
def InOrder(node):
    if node is not nullptr:
        InOrder(node.left)
        print(node.value)
        InOrder(node.right)
```

Сначала спускается в левое поддерево, после прохождения всего левого поддерева посещаем текущую вершину, и только после этого спускается в правое поддерево. Выполняется рекурсивно для каждой вершины. Для схемы выше, обход дерева in-order, запущенный из корневой вершины:
D, C, E, B, A, G, I, F, J, H

1.1.4 Описание методов

В структуре данных Binary Search Tree выделяется набор основных и дополнительных методов. Основные методы, это:

- ***Search(value)*** - проверить наличие значения *value* в дереве и вернуть *true / false* в зависимости от ответа.
- ***Push(value)*** - добавить значение *value* в дерево, если такого элемента еще не было.
- ***Remove(value)*** - удалить значение *value* из дерева.
- ***Bypass()*** - пройти в порядке обхода по дереву и вывести его элементы.
- ***Clean()*** - очистить (удалить) структуру данных.

Дополнительными методами можно назвать нижеследующий набор. Важно понимать, что большинство методов из них не являются пользовательскими - они приватные. Тем не менее, некоторые допустимо располагать в пользовательском интерфейсе структуры.

- ***Find(value)*** - проверить наличие значения *value* в дереве и вернуть *nullptr* или *ссылку на искомый узел* в зависимости от ответа.
- ***FindMax()*** - вернуть *value* наибольшего элемента в дереве.
- ***RemoveMax()*** - удалить максимальное значение из дерева и вернуть его *value*.
- ***FindMin()*** - вернуть *value* наименьшего элемента в дереве.
- ***RemoveMin()*** - удалить минимальное значение из дерева и вернуть его *value*.

1.1.5 Хранение данных

Перед тем, как приступить непосредственно к реализации различного рода методов бинарного дерева поиска, необходимо описать, как будут храниться данные в структуре данных.

```
struct Node {
    auto value;
    Node* left = nullptr;
    Node* right = nullptr;
    // Node* parent = nullptr;
};
```

Поле структуры *value* отвечает за хранение значения узла дерева. Именно в этом поле мы сохраняем данные, переданные в структуру BST.
Поля *left* и *right* отвечают за хранение указателей на левую и правую дочернюю вершину соответственно.
Опционально можно еще хранить ссылку на родителя узла (узел, для кого мы являемся дочерним) - такая реализация немного проще. Именно ее мы и собираемся выполнять, ведь доработка программы так, чтобы не использовать третью ссылку является не слишком сложным упражнением.

Как мы видим, реализация хранения узла структуды данных очень похожа на АТД список.

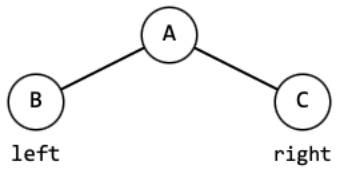
Помимо хранения узлов, необходимо предусмотреть хранение бинарного дерева поиска, а именно: всех методов и, самое главное, корня **root**. Ведь по сути корень символизирует все дерево и без него невозможно найти дргие узлы дерева.
При этом мы разделяем объявление публичных от приватных методов, чтобы никакой пользователь не смог бы обратиться ко внутренним данным структуры.

```
class BST {
private:
    Node* root;
    Find(target_value, current_node);
    RemoveMin(current_node);
public:
    Search(target_value);
    Push(target_value);
    Remove(target_value);

};
```

1.1.6 Реализация поисковых методов

Самый простой и один из самых важных методов в создаваемой структуре данных - метод *Search(value)*. Но для его реализации предлагается ввести вспомогательный приватный метод *Find(value)*, который будет возвращать указатель на искомый элемент или nullptr, если такого нет в дереве.

Алгоритм поиска будет рекурсивный. Рассмотрим следующую ситуацию, когда мы находимся в вершине A и у нее есть две дочерние вершины B и C. Представим, что мы хотим найти узел со значением *target_value* в дереве. Тогда приется рассмотерть три случая: $target\_value < A.value$, $target\_value = A.value$ и $target\_value > A.value$. В первом и третьем случае нам необходимо перейти в поддерево B или поддерево C соответственно. Во втором случае надо просто вернуть из функции сообщение, что искомый узел найден. Необходимо не забыть случай, когда узел A изначально сам равен nullptr - это означает, что в процессе рекурсии мы достигли листа дерева и не нашли элемента.




```
def Find(target_value, current_node):
    if current_node == nullptr:
        return nullptr
    if target_value < current_node.value:
        return Find(target_value, current_node->left)
    if target_value > current_node.value:
        return Find(target_value, current_node->right)
```

После реализации вспомогательного метода Find, можно перейти к реализации метода Search, который будет вызывать приватный метод Find с требуемыми параметрами и обрабатывать результат: либо возвращать true, либо false.

```
def Search(target_value):
    auto target_node = Find(target_value, root)
    if target_node == nullptr:
        return false
    else:
        return true
```

1.1.7 Реализация метода добавления

Теперь предлагается перейти к реализации добавления элементов в структуру данных. И правда: какой это тип данных, если в него нельзя добавлять элемнеты. Заметим, что добавление узла всегда будет происходить в лист дерева, так что особыз проблем быть не должно.
Алгоритм представляет собой набор действий: найти нужное место, создать новый узел, поставить его на найденную позицию. Мы приведем нерекурсивный алгоритм, хотя, безусловно, рекурсивный также имеет место существовать.

```
def Push(target_value):
    runner = root;
    new_node = new Node;
    new_node->value = target_value;
    if root is nullptr:
        root = new_node
        return;

    while runner is not nullptr:
        if target_value < runner->value:
            if runner->left is not nullptr:
                runner = runner->left;
            else:
                runner->left = new_node;
                // new_node->parent = runner;
                break;
        else if target_value > runner->value:
            if runner->right is not nullptr:
                runner = runner->right;
            else:
                runner->right = new_node;
                // new_node->parent = runner;
                break;
        else:
            return;
```

Давайте подробно рассмотрим алгоритм выше. До начала цикла while мы должны создать новый узел, проверить, что дерево непусто. Именно это мы и делаем.
Внутри цикла мы определяем, в какое из двух поддеревьев необходимо спуститься. Если дочернего ребенка, куда мы собираемся спуститься нет, мы ставим созданный узел на эту позицию.
Если же дочерний узел есть, спускаемся в него.
Отдельно в else проверяем, что в дереве нет узла, совпадающего с добавляемым значением.

1.1.8 Реализация метода удаления

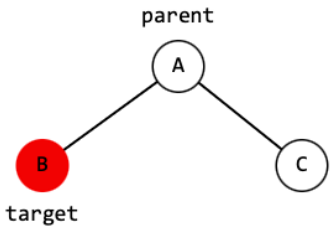
Плавно переходим к самому сложному методу изучаемой структуры данных - метод *Remove(target_value)*. Решение задачи сводится к решению нескольких этапов: поиск элемента, который мы хотим удалить (с помощью метода *Find*), непосредственное удаление элемента. Если с поиском элементов все понятно, то вот этап непосредственного удаления необходимо рассмотреть подробнее, там есть три различных варианта.

Непосредственное удаление элемента
Задача: имеется ссылка на узел, который мы хотим удалить. Требуется исключить такой узел из

структуры данных, сохранив корректность бинарного дерева поиска.
Разберем три случая:

1. Вершина является листом (нет дочерних узлов)
2. У вершины один дочерний узел.
3. У вершины два дочерних узла.

1. Если узел является листом, требуется удалить узел и у родителя поменять ссылку на *nullptr*.



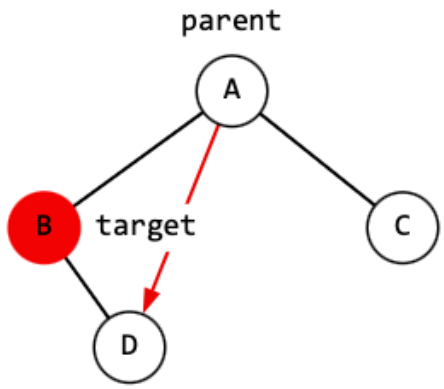
(a) Изображение первого случая

```
if target_node->parent->left == target_node:
    target_node->parent->left = nullptr;
else if target_node->parent->right == target_node:
    target_node->parent->right = nullptr;
delete target_node
```

(b) Алгоритм первого случая

Пусть *target_node* - вершина B является узлом, который мы хотим удалить. У нее нет дочерних листов, поэтому вершина подходит под требования. Алгоритм немного запутан, так как мы изначально не знаем, является ли *target_node* левым или правым «ребенком» parent.

2. Если у узла одна дочерняя вершина. Необходимо добавить ссылку с родителя на ребенка.



(a) Изображение второго случая

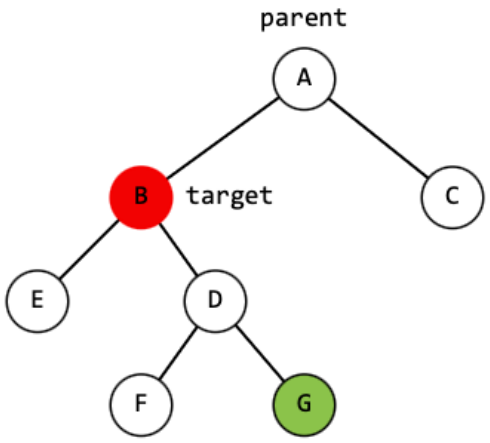
```
parent = target_node->parent
if parent->left == target_node:
    if target_node->left is not nullptr:
        parent->left = target_node->left;
    else if target_node->right is not nullptr:
        parent->left = target_node->right;

else if parent->right == target_node:
    if target_node->left is not nullptr:
        parent->right = target_node->left;
    else if target_node->right is not nullptr:
        parent->right = target_node->right;
delete target_node
```

(b) Алгоритм второго случая

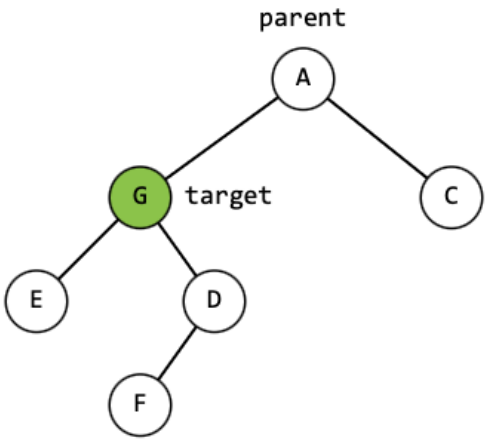
Пусть *target_node* - вершина B является узлом, который мы хотим удалить. У нее есть одна дочерняя вершина D, после удаления вершины B (аналогичное первому случаю) необходимо поставить вершине A в соответствие ребенка вершины B – а именно вершину D.
Сложность опять заключается в том, чтобы поставить вершине *parent* в соответствие нужного (левого или правого) «ребенка».

3. Если у узла присутствует обе дочерние вершины, алгоритм приобретает некую сложность. Мы не можем просто взять и удалить вершину, так как родителю не сможем поставить обе дочерние вершины *target_node*. Необходимо поставить на место удаляемой вершины либо узел со значением чуть больше вершины *target_node*, либо узел со значением чуть меньше вершины *target_node*. Такие узлы являются соответственно минимумом правого поддерева, либо максимумом левого поддерева. Для решения задачи необходимо выбрать стратегию, откуда мы берем этот узел: можно брать справа, можно слева – от этого ничего не поменяется. Более умным будет выбирать узел из того поддерева, где высота поддерева выше, но на таком «плохом» дереве это будет не критично.

Чтобы реализовать данный метод, предлагается использовать функцию *RemoveMin(node)*, удаляющий и возвращающий минимум в поддереве с корнем *node*.
Реализовать данный метод предлагается в качестве упражнения. Хочется лишь сказать, что максимум (и минимум) может быть только листом, либо узлом с одной дочерней вершиной, ибо если у максимума есть правый ребенок - максимум найден некорректно, так как правый ребенок больше текущего узла. Аналогично с левым ребенком минимума.



(a) До удаление target



(b) После удаление target

После написания кода *RemoveMin()*, исследуемый метод становится крайне простым:

```
min_right = RemoveMin(target_node->right)
target_node->value = min_right
```

Алгоритм удаления узла
Напишем алгоритм метода Remove, основываясь на изученной информации выше.

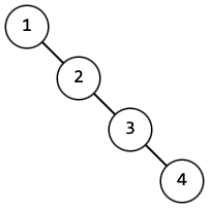
```
def Remove(target_value):
    target_node = Find(target_value, root)
    left = target_node->left
    right = target_node->right
    if left is not nullptr and right is not nullptr:
        RemoveNodeTwoChildren(target_node)
    else if left is nullptr and right is nullptr:
        RemoveLeaf(target_node)
    else:
        RemoveNodeOneChild(target_node)
```

Алгоритм стирания дерева
В разговоре про удаление данных из дерева нельзя не упоминуть про алгоритм стирания дерева - то есть освобождение памяти, требуемой для хранения структуры данных BST.
Алгоритм крайне прост: Просто проходим по узлам дерева в порядке *PostOrder* и вместо вывода данных на экран удалять вершину втроенной функцией *delete*.

1.1.9 Асимптотика выполнения методов

Можно заметить, что каждый из методов использует некий «спуск» к листам дерева: то есть рекурсивно спускаемся либо в правое поддерево, либо в левое.

Таким образом, в худшем случае, выполнение каждого из основных методов будет занимать время, прямопропорциональное высоте, а значит асимптотика будет **O(H)**, где H - высота дерева.



Проблема балансировки
Как мы видим, асимптотика крайне чувствительна к высоте дерева, а значит перед нами стоит задача уменьшения высоты дерева.
На данный момент наше дерево периодически может быть очень плохим: вырождаться в линию (список), как показано на изображении справа.

Для уменьшения количества таких случаев, необходимо сделать так, чтобы у каждого узла было как можно больше дочерних: то есть на всех высотах, больше единицы (листов) было по два ребенка. Если мы сможем придумать некий алгоритм, способный изменять дерево по правилу так, чтобы у почти всех вершин было по два ребенка, высота дерева, как и в случае с АТД пирамидой, пройденной раннее, будет **O(logN)** - что звучит крайне привлекательно. Такие алгоритмы называются балансировкой, а дерево, сохраняющее такие свойство - самобалансирующимся. В следующих главах мы подробнее поговорим о таких замечательных структурах данных.