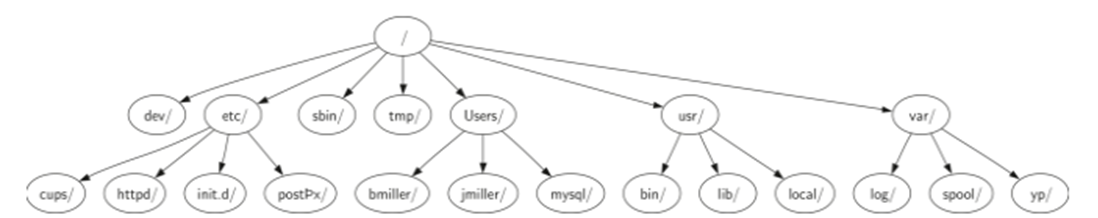
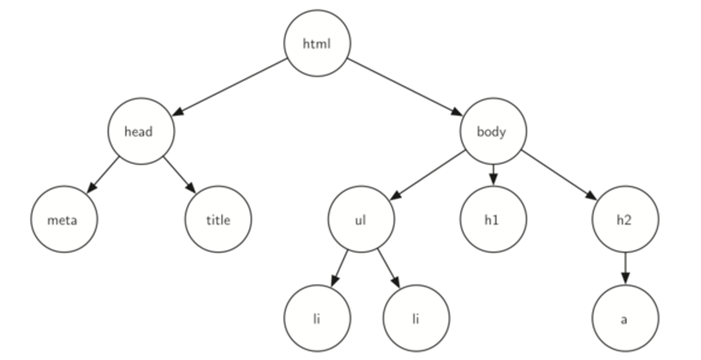
***Бинарные деревья***

Введение

В завершении курса мы поговорим о такой популярной структуре данных, как дерево, которую мы можем наблюдать в базах данных, графике, компьютерных сетях, операционных системах. Отличие от обычных деревьев в том, что деревья в информатике растут корнями вверх.

Примеры деревьев:





Терминология

**Узел.** Ключевая составляющая дерева. Может иметь название – ключ.

**Ветвь.** Еще одна фундаментальная часть дерева, соединяющая два узла и отражающая связь между ними. Любой узел (кроме корня) имеет только одну входящую ветвь, но может иметь несколько исходящих ветвей.

**Корень.** Узел дерева, не имеющий входящих ветвей.

**Путь.** Упорядоченная последовательность узлов, соединенных ветвями.

**Дети.** Совокупность узлов, имеющих входящие ветви от одного узла.

**Родитель.** Узел, связанный с другими узлами с помощью исходящих ветвей.

**Братья.** Узлы, являющиеся детьми одного родителя.

**Поддерево.** Совокупность узлов и ветвей, формирующих связь некоторого родителя и всех его потомков.

**Лист.** Узел, не имеющий детей.

**Уровень.** Уровень узла X определяется как число ветвей в пути от корня до самого узла X.

**Высота.** Это максимальный уровень любого узла дерева.

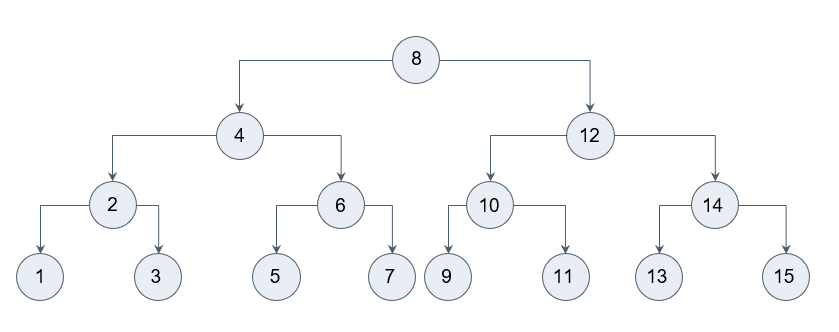
В нашем курсе сделаем упор на двоичные деревья.

Двоичное дерево

# Двоичным деревом поиска (ДДП) называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя. Вершины, не имеющие потомков, называются листами.

**ИТАК**

Двоичное дерево поиска строится по определенным правилам:

* У каждого узла не более двух детей.
* Любое значение меньше значения узла становится левым ребенком или ребенком левого ребенка.
* **Любое значение больше или равное значению узла становится правым ребенком или ребенком правого ребенка.

Как сделать программную реализацию бинарного дерева на Python? Воспользоваться знаниями в области ООП.

**Листинг 1. task\_1.py**

|  |
| --- |
| **class** BinaryTree:  **def** \_\_init\_\_(self, root\_obj):  *# корень* self.root = root\_obj  *# левый потомок* self.left\_child = **None** *# правый потомок* self.right\_child = **None** *# добавить левого потомка* **def** insert\_left(self, new\_node):  *# если у узла нет левого потомка* **if** self.left\_child == **None**:  *# тогда узел просто вставляется в дерево  # формируется новое поддерево* self.left\_child = BinaryTree(new\_node)  *# если у узла есть левый потомок* **else**:  *# тогда вставляем новый узел* tree\_obj = BinaryTree(new\_node)  *# и спускаем имеющегося потомка на один уровень ниже* tree\_obj.left\_child = self.left\_child  self.left\_child = tree\_obj   *# добавить правого потомка* **def** insert\_right(self, new\_node):  *# если у узла нет правого потомка* **if** self.right\_child == **None**:  *# тогда узел просто вставляется в дерево  # формируется новое поддерево* self.right\_child = BinaryTree(new\_node)  *# если у узла есть правый потомок* **else**:  *# тогда вставляем новый узел* tree\_obj = BinaryTree(new\_node)  *# и спускаем имеющегося потомка на один уровень ниже* tree\_obj.right\_child = self.right\_child  self.right\_child = tree\_obj   *# метод доступа к правому потомку* **def** get\_right\_child(self):  **return** self.right\_child   *# метод доступа к левому потомку* **def** get\_left\_child(self):  **return** self.left\_child   *# метод установки корня* **def** set\_root\_val(self, obj):  self.root = obj   *# метод доступа к корню* **def** get\_root\_val(self):  **return** self.root   r = BinaryTree(8) print(r.get\_root\_val()) print(r.get\_left\_child()) r.insert\_left(25) print(r.get\_left\_child()) print(r.get\_left\_child().get\_root\_val()) r.insert\_right(12) print(r.get\_right\_child()) print(r.get\_right\_child().get\_root\_val()) r.get\_right\_child().set\_root\_val(16) print(r.get\_right\_child().get\_root\_val()) |

В приведенном примере мы создали простое дерево с узлом 8 в качестве корня и узлами 4 и 12 в качестве левого и правого потомков, соответственно. Обратите внимание, что левый и правый потомки корня являются сущностями класса BinaryTree. Таким образом, мы видим рекурсивное определение дерева. Это позволяет нам работать с любым потомком двоичного дерева, как с самим деревом.

Еще пример построения дерева.

**Листинг 2. task\_2.py**

|  |
| --- |
| **class** Node:  **def** \_\_init\_\_(self, data=**None**, left=**None**, right=**None**):  self.data = data  self.left = left  self.right = right   **def** \_\_str\_\_(self):  **return f'Node [{**str(self.value)**}]'   class** Tree:  **def** \_\_init\_\_(self):  self.root = **None** *# функция для добавления узла в дерево* **def** new\_node(self, data):  temp = Node(0, **None**, **None**)  temp.data = data  **return** temp   *# функция для вычисления высоты дерева* **def** height(self, node):  **if** node == **None**:  **return** 0  **else**:  lheight = self.height(node.left)  rheight = self.height(node.right)   **if** lheight > rheight:  **return** (lheight + 1)  **else**:  **return** (rheight + 1)   *# функция для распечатки элементов на определенном уровне дерева* **def** print\_given\_level(self, root, level, ltr):  **if** root == **None**:  **return  if** level == 1:  print(**"%d "** % root.data)  **elif** level > 1:  **if** ltr:  self.print\_given\_level(root.left, level - 1, ltr);  self.print\_given\_level(root.right, level - 1, ltr);  **else**:  self.print\_given\_level(root.right, level - 1, ltr);  self.print\_given\_level(root.left, level - 1, ltr);   *# функция для распечатки дерева* **def** print\_level\_order(self, root):  h = self.height(self.root)  i = 1  ltr = 1  **while** i <= h:  self.print\_given\_level(self.root, i, ltr)  i += 1   *# функция для вычисления ширины дерева* **def** get\_max\_width(self, root):  max\_wdth = 0  i = 1  width = 0  h = self.height(root)  **while** i < h:  width = self.get\_width(root, i)  **if** width > max\_wdth:  max\_wdth = width  i += 1   **return** max\_wdth   **def** get\_width(self, root, level):  **if** root == **None**:  **return** 0  **if** level == 1:  **return** 1  **elif** level > 1:  **return** self.get\_width(root.left, level - 1) + self.get\_width(root.right, level - 1)  self.get\_width(root.right, level - 1)   t = Tree() t.root = t.new\_node(8)  t.root.left = t.new\_node(4) t.root.right = t.new\_node(12) t.root.left.left = t.new\_node(2) t.root.left.right = t.new\_node(6) t.root.right.left = t.new\_node(10) t.root.right.right = t.new\_node(14) t.root.left.left.left = t.new\_node(1) t.root.right.right.right = t.new\_node(25) t.root.left.left.left.left = t.new\_node(0) t.root.left.left.right = t.new\_node(3) t.root.right.right.right.right = t.new\_node(100) t.root.right.right.right.left = t.new\_node(20)  t.print\_level\_order(t.root) print(**f'Высота: {**t.height(t.root)**}'**) print(**f'Ширина: {**t.get\_max\_width(t.root)**}'**) |

Зачем нужны бинарные деревья и где применяются

Основное назначение двоичных деревьев заключается в повышении эффективности поиска. При поиске выполняются такие операции, как нахождение заданного элемента из набора различных элементов, определение наибольшего или наименьшего элемента в наборе, фиксация факта, что набор содержит заданный элемент.

На практике двоичные деревья поиска очень удобны во многих приложениях, где входные данные случайны или есть другие причины надеяться на сохранение деревом хорошей формы.

Что же касается в целом деревьев, то в различных задачах применяются различные *виды деревьев*, так например:

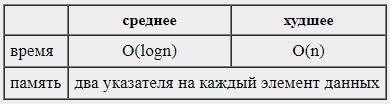
* при разработке парсеров или трансляторов полезным может оказаться *дерево синтаксического разбора* (*синтаксическое дерево*);
* при работе со строками удобными могут оказаться *суффиксные деревья*;
* при разработке оптимальных алгоритмов на графах полезным может оказаться структура данных в виде *кучи*;
* *двоичные деревья поиска* используются при реализациях словаря, они являются достаточно простым и распространенным видом деревьев, поэтому их мы рассмотрим более подробно;
* перечислять дальше виды деревьев не имеет смысла, думаю, уже понятно, что разновидностей деревьев много, и у каждой есть своя *сфера применения*.

В большинстве реляционных СУБД в качестве структуры данных для индексов (та или иная их реализация) используются именно деревья. И не просто деревья, а сбалансированные деревья поиска. В этой статье как раз о них.

В большинстве реляционных СУБД в качестве структуры данных для индексов (та или иная их реализация) используются именно деревья. И не просто деревья, а сбалансированные деревья поиска. В этой статье как раз о них.

Сами по себе деревья бесполезны, если мы не можем выполнять с ними определенные операции. К таким операциям обычно относят: поиск элемента, поиск минимального (максимального) элемента, вставка, удаление.

Эффективность



Применение бинарного дерева для реализации алгоритма кодирования по Хаффману

В основе идеи кодировании Хаффмана – частота появления символа в последовательности. Символ, который встречается чаще всего, получает новый, очень маленький код. Наиболее редко фигурирующий символ, напротив, получает очень длинный код.

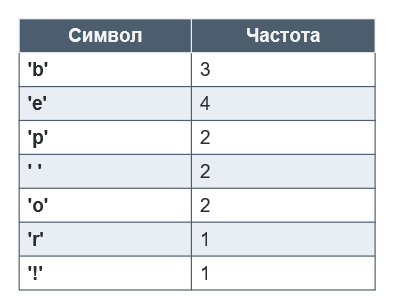
В следующем примере символ будет иметь длину 8 бит.

Размер символа выбирается соответственно ожидаемой строке ввода. Важно помнить, что размер кода символа прямо пропорционален выбранному размеру символа.

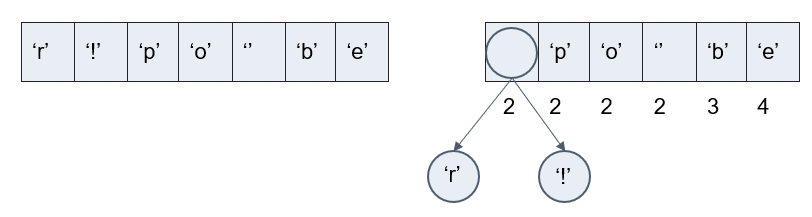
Рассмотрим применение алгоритма Хаффмана. Допустим, у нас есть строка «beep boop beer!», для которой на каждый знак тратится по одному байту. Это означает, что вся строка занимает 15\*8 = 120 бит памяти. После кодирования – 40 бит.

Чтобы получить код для каждого символа, нужно построить бинарное дерево, где каждый лист будет содержать символ. Символы с меньшей частотой будут дальше от корня, чем символы с большей.

Посчитаем частоты всех символов.

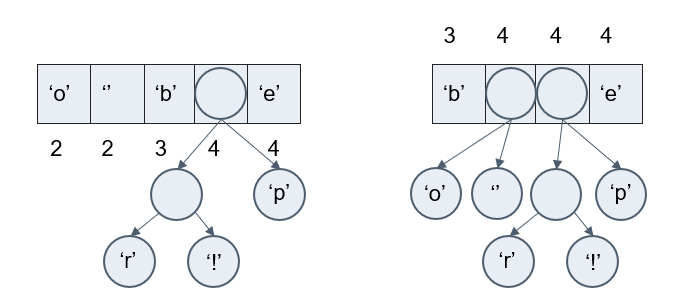


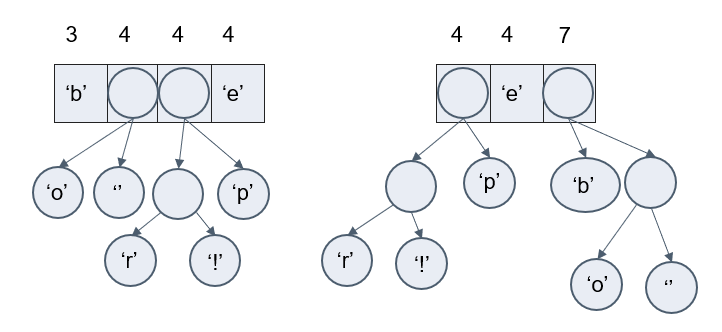
После вычисления частот создадим узлы бинарного дерева для каждого знака и добавим их в очередь, используя частоту в качестве приоритета. В начале очереди окажутся символы с наименьшей частотой, с наибольшей – в конце.

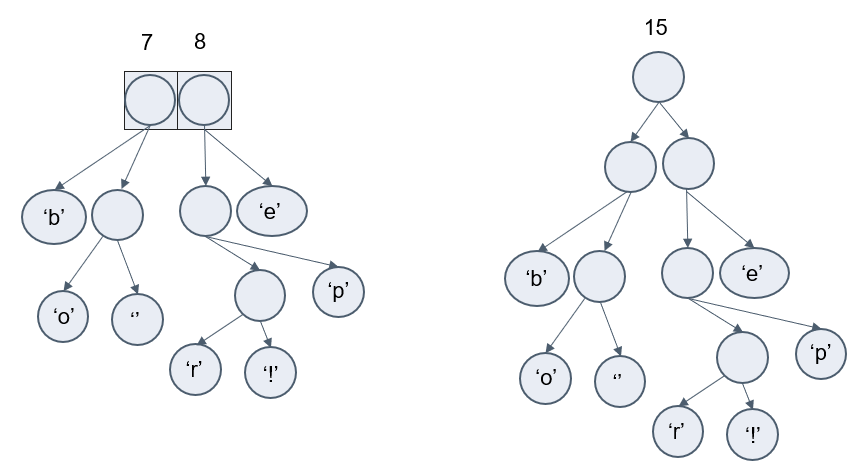


Берем два первых элемента из очереди и связываем их, создавая новый узел дерева. В нем они оба будут потомками, а приоритет нового узла будет равен сумме их приоритетов. Добавим получившийся узел обратно в очередь.

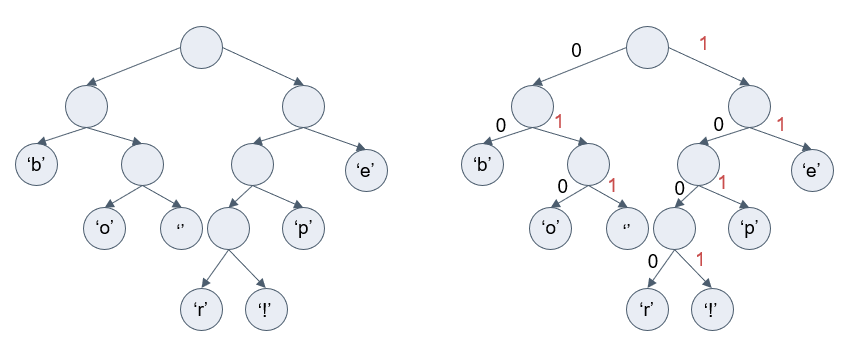
Повторим те же шаги необходимое количество раз.







В конце свяжем два последних элемента и получится итоговое дерево.



Чтобы получить код для каждого символа, надо просто пройтись по дереву и для каждого перехода добавлять 0, если мы идем влево, и 1 – если направо.

Получим следующие коды для символов:



Чтобы расшифровать закодированную строку, надо идти по дереву, сворачивая в соответствующую каждому биту сторону – до тех пор, пока мы не достигнем листа.

Важно иметь в виду, что каждый код не является префиксом для кода другого символа. В нашем примере, если 00 – это код для 'b', то 000 не может оказаться чьим-либо кодом – иначе мы получим конфликт.

На практике при реализации данного алгоритма сразу после создания дерева строится таблица Хаффмана. Она содержит каждый символ и его код, потому что это делает кодирование более эффективным (по сути, это односвязный список).

Входная строка: «beep boop beer!»

Входная строка в бинарном виде: «0110 0010 0110 0101 0110 0101 0111 0000 0010 0000 0110 0010 0110 1111 0110 1111 0111 0000 0010 0000 0110 0010 0110 0101 0110 0101 0111 0010 0010 000»

Закодированная строка: «0011 1110 1011 0001 0010 1010 1100 1111 1000 1001»

Как видим, ASCII-версия строки и закодированная версия существенно различаются.

Рассмотрим варианты программной реализации алгоритма Хаффмана.

**Листинг 3. task\_3.py**

|  |
| --- |
| *"""Хаффман через коллекции"""* **from** collections **import** Counter, deque   **def** haffman\_tree(s):  *# Считаем уникальные символы.  # Counter({'e': 4, 'b': 3, 'p': 2, ' ': 2, 'o': 2, 'r': 1, '!': 1})* count = Counter(s)  *# Сортируем по возрастанию количества повторений.  # deque([('r', 1), ('!', 1), ('p', 2), (' ', 2), ('o', 2), ('b', 3), ('e', 4)])* sorted\_elements = deque(sorted(count.items(), key=**lambda** item: item[1]))  *# Проверка, если строка состоит из одного повторяющего символа.* **if** len(sorted\_elements) != 1:  *# Цикл для построения дерева* **while** len(sorted\_elements) > 1:  *# далее цикл объединяет два крайних левых элемента  # Вес объединенного элемента (накопленная частота)  # веса - 2, 4, 4, 7, 8, 15* weight = sorted\_elements[0][1] + sorted\_elements[1][1]  *# Словарь из 2 крайних левых элементов, попутно вырезаем их  # из "sorted\_elements" (из очереди).  # comb - объединенный элемент* **'''  {0: 'r', 1: '!'}  {0: {0: 'r', 1: '!'}, 1: 'p'}  {0: ' ', 1: 'o'}  {0: 'b', 1: {0: ' ', 1: 'o'}}  {0: {0: {0: 'r', 1: '!'}, 1: 'p'}, 1: 'e'}  {0: {0: 'b', 1: {0: ' ', 1: 'o'}}, 1: {0: {0: {0: 'r', 1: '!'}, 1: 'p'}, 1: 'e'}}  '''** comb = {0: sorted\_elements.popleft()[0],  1: sorted\_elements.popleft()[0]}   *# Ищем место для ставки объединенного элемента* **for** i, \_count **in** enumerate(sorted\_elements):  **if** weight > \_count[1]:  **continue  else**:  *# Вставляем объединенный элемент* sorted\_elements.insert(i, (comb, weight))  **break  else**:  *# Добавляем объединенный корневой элемент после  # завершения работы цикла* sorted\_elements.append((comb, weight))  **'''  deque([({0: 'r', 1: '!'}, 2), ('p', 2), (' ', 2), ('o', 2), ('b', 3), ('e', 4)])  deque([(' ', 2), ('o', 2), ('b', 3), ({0: {0: 'r', 1: '!'}, 1: 'p'}, 4), ('e', 4)])  deque([('b', 3), ({0: ' ', 1: 'o'}, 4), ({0: {0: 'r', 1: '!'}, 1: 'p'}, 4), ('e', 4)])  deque([({0: {0: 'r', 1: '!'}, 1: 'p'}, 4), ('e', 4), ({0: 'b', 1: {0: ' ', 1: 'o'}}, 7)])  deque([({0: 'b', 1: {0: ' ', 1: 'o'}}, 7), ({0: {0: {0: 'r', 1: '!'}, 1: 'p'}, 1: 'e'}, 8)])  deque([({0: {0: 'b', 1: {0: ' ', 1: 'o'}}, 1: {0: {0: {0: 'r', 1: '!'}, 1: 'p'}, 1: 'e'}}, 15)])  '''  else**:  *# приравниваемыем значение 0 к одному повторяющемуся символу* weight = sorted\_elements[0][1]  comb = {0: sorted\_elements.popleft()[0], 1: **None**}  sorted\_elements.append((comb, weight))  *# sorted\_elements - deque([({0: {0: 'b', 1: {0: ' ', 1: 'o'}}, 1: {0: {0: {0: 'r', 1: '!'}, 1: 'p'}, 1: 'e'}}, 15)])  # {0: {0: 'b', 1: {0: ' ', 1: 'o'}}, 1: {0: {0: {0: 'r', 1: '!'}, 1: 'p'}, 1: 'e'}}  # словарь - дерево* **return** sorted\_elements[0][0]   code\_table = dict()  *# tree - {0: {0: 'b', 1: {0: ' ', 1: 'o'}}, 1: {0: {0: {0: 'r', 1: '!'}, 1: 'p'}, 1: 'e'}}* **def** haffman\_code(tree, path=**''**):  *# Если элемент не словарь, значит мы достигли самого символа  # и заносим его, а так же его код в словарь (кодовую таблицу).* **if not** isinstance(tree, dict):  code\_table[tree] = path  *# Если элемент словарь, рекурсивно спускаемся вниз  # по первому и второму значению (левая и правая ветви).* **else**:  haffman\_code(tree[0], path=**f'{**path**}0'**)  haffman\_code(tree[1], path=**f'{**path**}1'**)   *# строка для кодирования* s = **"beep boop beer!"** *# функция заполняет кодовую таблицу (символ-его код) # {'b': '00', ' ': '010', 'o': '011', 'r': '1000', '!': '1001', 'p': '101', 'e': '11'}* haffman\_code(haffman\_tree(s))  *# code\_table - {'b': '00', ' ': '010', 'o': '011', 'r': '1000', '!': '1001', 'p': '101', 'e': '11'}  # выводим коды для каждого символа* **for** i **in** s:  print(code\_table[i], end=**' '**) print() |

В этом примере для кодирования используется таблица Хаффмана (набор кодов и их соответствий в двоичном виде), а для декодирования — дерево Хаффмана. Для декодирования сообщения получателю необходимо сперва построить дерево, после чего он будет знать, как декодировать сообщение, чтобы прочесть оригинальное сообщение. Существует мнение, что кодирование по Хоффану не очень эффективно для большого количества редких элементов в огромном объеме текста. Однако эффективно для работы с документами, изображениями и т.д.