ЛЕКЦИЯ 22
Двоичное дерево

1. Двоичное дерево. Класс дерево.

k-ричное дерево — дерево, в котором количество дочерних вершин у каждой вершины не больше k штук. При этом троичное дерево является одновременно и четверичным — просто четвертого ребра еще нет. Последовательность дочерних вершин может быть неупорядоченной. Нам же интересен случай, когда k-ричное дерево является упорядоченным.

Рассмотрим двоичное дерево, упорядоченное, в общем случае не сбалансированное (в отличии от кучи). Двоичное дерево можно нарисовать так, что мы будем называть дочерние вершины "левая" или "правая"(по аналогии с кучей).

Поддерево (правое/левое) — подграф дерева, корень которого является дочерней вершиной (правой/левой).

В прошлом семестре вводился такой способ хранения данных как односвязный список. Оказывается, можно реализовать такое звено, которое подходит для двоичного дерева поиска. Сделаем 2 указателя: на левое поддерево и правое поддерево, также указатель наверх (т.е. на родителя). Из таких звеньев можно собирать дерево.

Удобно считать пустой граф пустым деревом (хотя по определению дерева это неверно).

Программа №1.1. Класс Дерево

```
1    class Node:
2        def __init__(self, key, value): # Создаем звено ключ, значение
3            self.key = key
4            self.value = value
5            self.parent = None
6            self.left = None
7            self.right = None
8
9    class Tree:
10       def __init__(self):
11           self.root = None
12       def print(self, node):
13           if node is None: # Благодаря этому нам не важно, пустое наше поддерево
      или нет - крайний случай проверен
14               return
15           self.print(node.left)
16           print((node.key, node.value)) # Это не совсем обратный ход рекурсии
17           self.print(node.right)
```

2. Двоичное дерево поиска

Двоичное дерево поиска — это корневое двоичное упорядоченное дерево, построенное по следующему правилу для любого звена node: все ключи левого поддерева $key_i < key_{node}$, все ключи правого поддерева $key_j > key_{node}$.

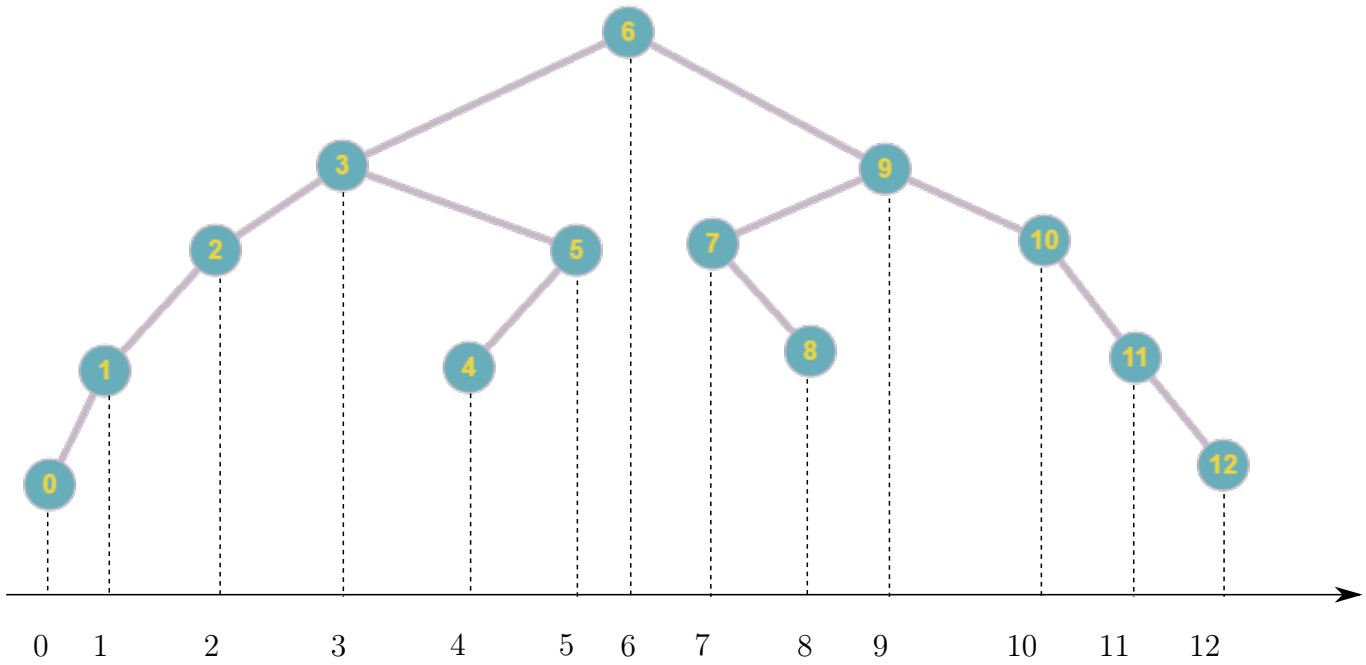

Рис. 1. Двоичное дерево поиска

Возьмем такие числа:

$$6\ 3\ 5\ 4\ 2\ 9\ 7\ 8\ 1\ 11\ 10\ 12\ 0$$

и изобразим для них двоичное дерево поиска (рис. 1).

Алгоритм построения: сначала дерево пустое и ни одного звена нет. Берем числа поочереди. 3 меньше 6, поэтому 6 становится главной. 3 становится левым поддеревом шестерки, т.к. $3 < 6$ (добавляем числа меньшие корня в левое поддерево, а числа большие корня в правое поддерево). Далее число 5 меньше 5, но больше 3. Поэтому 5 находится в левом поддереве 6, но в правом поддереве тройки. Дальнейшее построение аналогично. Красота такого метода в том, что если «спроектировать» числа на прямую, получается числовая ось, на которой числа расставлены в порядке возрастания.

Двоичное дерево поиска работает как бинарный поиск. Количество операций сравнения равно высоте дерева $O(\log_2 N)$ (если дерево сбалансировано). Чем оно лучше бинарного поиска в списке? Для добавления элемента требуется то же время ($O(\log_2 N)$). Но в списке после того, как мы нашли, куда вставить элемент, требуется сделать циклический сдвиг.

Если у элемента нет дочерних вершин, а его надо удалить, то это сделать просто. Но вот если у него есть одна дочерняя вершина, мы присоединяем оставшееся дерево к верхнему родителю (можно привести аналогию с подчиненными: если начальника подчиненных уволили, то эти подчиненные становятся подчиненными начальника рангом выше).

В случае когда нужно присоеденить 2 дерева (т.е. если мы удалили вершину, у которого было 2 поддерева, например третью) переходим к левому поддереву удаляемой вершины и к самому правому из него добавляем правое поддерево той вершины, которую мы удалили. Минусы: при удалении корневого элемента длина дерева удваивается.

Но есть более оптимальный вариант: после удаления вершины (например тройки) возьмем самый правый элемент левого поддерева этой вершины (т.е. тройки) и поставим его вместо элемента, которого мы удалили (т.е. вместо стройки), что предотвратит от удваивания длины всего дерева (т.е. если бы у двойки было бы правое поддерево, мы бы нашли самый большой элемент в этом поддереве и поставили бы его на место тройки).

Почему не воспользоваться хеш–таблицей? Хеш–таблица не упорядочена, в отличии от двоичного дерева поиска.