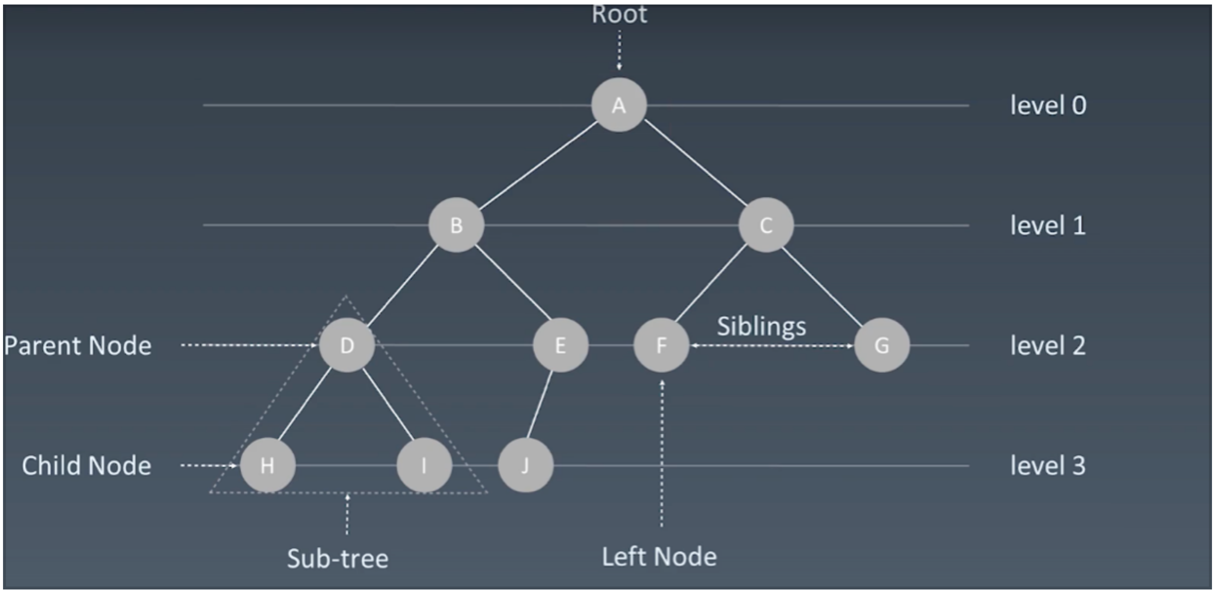
5. 树 （ Tree ）、二叉树（ Binary Tree ）、二叉搜索树（ Binary Search Tree ）的实现和特性

树 Tree

> 二维数据结构.

树的概念

1. 根节点 Root；
2. 父亲节点；
3. 兄弟节点;
4. 左右节点;
5. 左右子树；
6. 层级从0开始，根节点为第0层.



树产生的原因

出现树本身并不是说一维的结构不好用，而是我们本身生活在一个三维的世界里面，很多工程问题需要在二维的基础上去解决。
树这种二维的数据结构，就是人经常会碰到的一种情况，类似于人在看到鸟后发明飞机一样，都是从生活实际问题，抽象发明出来解决问题的。

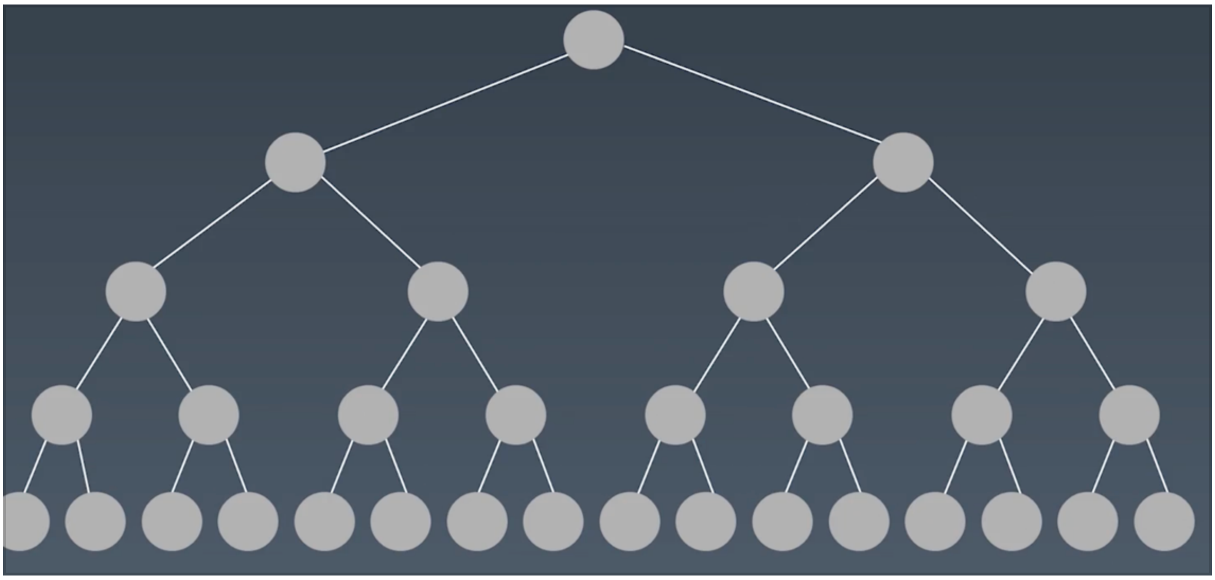
注意点

树与图的差别在于有没有环，如果存在环，则为图；不存在环，则为树.

Linked List 是特殊化的 Tree, Tree 是特殊化的图.

二叉树 Binary Tree

> 二叉树：子节点只有两个，左子节点和右子节点.
>
> 满二叉树：左右子节点都存在.



二叉树 示例代码

Python

```
class TreeNode:
 def __init__(self, val):
     self.val = val
     self.left,self.right = None,None
```

Java

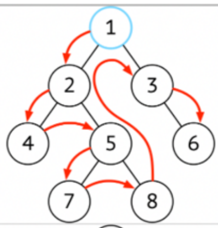
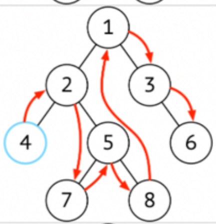
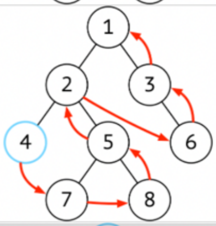
```java
public class TreeNode{
  public int val;
  public TreeNode left,right;
  public TreeNode(int val){
      this.val = val;
      this.left = null;
      this.right = null;
  }
}
```

C++

```cpp
struct TreeNode{
  int val;
  TreeNode * left;
  TreeNode * right;
  TreeNode(int x) : val(x),left(NULL),right(NULL){}
}
```

二叉树遍历

遍历算法总览

前序	**根 → 左 → 右**		想在节点上直接执行操作（或输出结果）使用先序
中序	**左 → 根 → 右**		在**二分搜索树**中，中序遍历的顺序符合从小到大（或从大到小）顺序的 要输出排序好的结果使用中序
后序	**左 → 右 → 根**		后续遍历的特点是在执行操作时，肯定**已经遍历过该节点的左右子节点** 适用于进行破坏性操作 比如删除所有节点，比如判断树中是否存在相同子树

遍历算法示例代码

> 树为什么基于递归?
> 因为树无法有效地进行循环，也可一强制性地循环，入广度优先搜索，但是比较麻烦，而写递归调用相对简单些.

Python

```
# 前序遍历 : 根 - 左 - 右
def preorder(self, root):

    if root:
        self.traverse_path.append(root.val)
        self.preorder(root.left)
        self.preorder(root.right)



# 中序遍历 : 左 - 根 - 右
def inorder(self, root):

    if root:
        self.inorder(root.left)
        self.traverse_path.append(root.val)
        self.inorder(root.right)
```

```python
# 后序遍历 : 左 - 右 - 根
def postorder(self, root):

    if root:
        self.postorder(root.left)
        self.postorder(root.right)
        self.traverse_path.append(root.val)
```

二叉搜索树 Binary Search Tree

定义：

> 二叉搜索树，也称为二叉排序树、有序二叉树（ Ordered Binary Tree ）、排序二叉树（ Sorted Binary Tree ），是指一棵空树或者具有下列性质的二叉树：
>
> 1. 左子树上 所有节点 的值均小于它的根节点的值；
> 2. 右子树上 所有节点 的值均大于它的根节点的值；
> 3. 以此类推,左、右子树也分别为二叉查找树 (这就是重复性) ；
> 4. 中序遍历是 升序遍历 .

二叉搜索树常见操作

Demo 演示

1. 查询 : O(long(n))

- 左子节点小于父节点，右子节点大于父节点，类似于二分查找.

2. 插入新节点（创建）: O(long(n))

- 插入的新节点已存在，count + 1;
- 插入的新节点不存在，在最后查找的位置处将其插入.

3. 删除 ：O(long(n))

- 如果是叶子节点，直接删掉即可；
- 如果不是叶子节点，删除后需要将第一个大于或小于其的子节点放在此位置代替它（一般选择第一个大于的它的子节点）.

注意：最坏情况下，二叉搜索树退化成链表，各个操作的时间复杂度均退化为 O(n).

二叉搜索树复杂度分析

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

思考题

树的面试题解法为什么一般都是 递归 操作？(见下节递归)

#Algorithm/Part II : Theory/Data Structure#