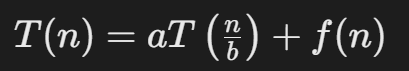
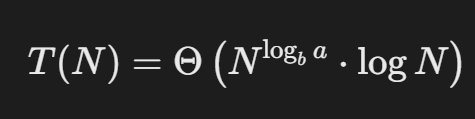
Homework1-4错题知识点整理：  
1，主定理：

如果时间复杂度满足以下式子：

  
（注意通常我们只计算f（n）=N的情况

那么其时间复杂度表达式为  


2，节点深度，距离根节点的距离，根节点的深度为0

**ancestors of a node：**

**到根节点的所有经过的节点**

**descendants of a node**

**子树中所有的节点**

**3，post in pre fix表达式与前中后序对应！！！！**

**4，二叉树的性质：N0=N2+1；**

Homework5 易错题：

1. BST同一行按照非降序排列

5，thread tree 只要箭头的行走方向与对应遍历的方向一样就是对应的thread方式

并查集（

Symentric 交换性

Reflexive 自反性

Transitive 传递性

Equivalence class 等价类，等价的元素组成的集合

Dynamic equvalance problem

将其中一个元素作为等价类的标志，其他元素作为叶子节点，并且用叶子节点指向标志节点，构成一个反树，来表示一个集合

并查集的基本操作：

1. 合并集合
2. 找到元素属于哪一个集合

直接使用树来进行操作，时间复杂度为n\*n

Union by size logN（以2为底）//根节点的内容是子树的元素个数的相反数

Union by height 改变高度，让其更加的扁平（N+MlogN）

Path compression 路径压缩：在第一次查找的时候，将所有经过的元素都指向根节点（M）！！！！！与N无关了

一般Union by height以及path compression一般不会一起用

树（tree）

树的遍历方式：

1. 层序遍历（level order traversal）  
   将一层遍历完再遍历下一层

辅助记忆

初始化一个辅助队列  
{

数组，front=0，rear=0；

F将root入队（front++）

}

主循环（当队列不为空的时候）

{

出队一个元素：

访问这个元素

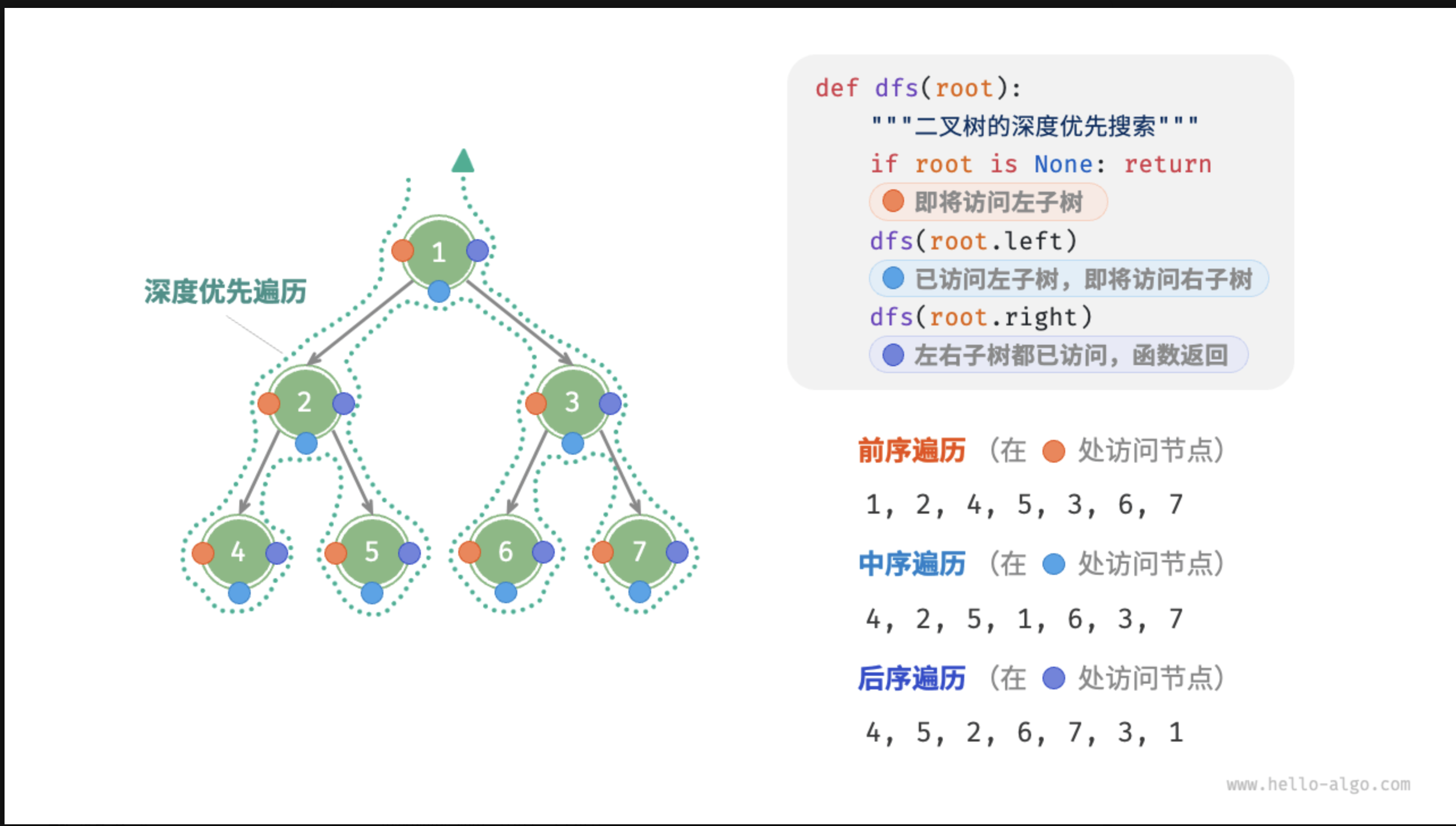
如果左子树不为空，那么就将左子树入队；

如果右子树不为空那么就让右子树入队；

}

1. 深度遍历：  
   前，中，后序遍历

辅助记忆



只需在对应的递归两条递归语句的合适位置加上访问语句就可以！！！！

1. BST（二进制搜索树）

查找元素：

辅助记忆

初始需要分配一个节点用于存储新数字，将新节点的左右子指针都赋值为NULL！！

只需要在初始设置的时候，设置一个指针指向前一次访问的指针，并且将指针的值在初始状态赋值为NULL；（cur赋值为root）

然后再每次更新cur之前将cur的值赋值给pre就行

最后跳出循环之后将判断数字与pre的大小，根据大小决定接哪边；

1. BST删除节点

辅助记忆：  
  
找到要删除的节点cur，以及父节点pre  
  
如果此节点有1或0个子树；  
则判断左子树是否为空，如果为空就对child变量赋值为右子树，反之则赋值为左子树；  
判断pre下的cur是左还是右子树，直接将pre->child;

如果有两个子节点：  
则寻找下一个中序遍历的元素：  
{

Ptr=cur；

Ptr=ptr->right

While(ptr->left!=0)

{

Ptr=ptr->left;

}

然后ptr就是大于num的最小数字节点

}  
  
递归调用删除函数，删除ptr位置的元素；  
最后将ptr位置变量覆盖cur位置的数值；  
  
【指引】可以用中序遍历来找到右子树的最小元素  
 当然如果要找最大的左子树元素，只需要修改中序遍历先遍历又子树再遍历左子树便可！！！！

5，决策树只要保证一边的节点树始终比另一边少就可以！！！

以下是一些优缺点的分析，还是很有趣的

二叉树的数组表示主要有以下优点。

* 数组存储在连续的内存空间中，对缓存友好，访问与遍历速度较快。
* 不需要存储指针，比较节省空间。
* 允许随机访问节点。

然而，数组表示也存在一些局限性。

* 数组存储需要连续内存空间，因此不适合存储数据量过大的树。
* 增删节点需要通过数组插入与删除操作实现，效率较低。
* 当二叉树中存在大量 None 时，数组中包含的节点数据比重较低，空间利用率较低。

堆（heaps）or 优先队列（priority queue）

完全二叉树的结构性质：

1. 完全二叉树，只有最后一行没有被填满
2. 最后一行的节点全部都靠左
3. （如果索引0被使用）已知一个节点其索引为i，则其左子节点为2\*i+1，右子节点的索引为2\*i+2，其父节点为（i-1）/2；  
   （如果索引0不被使用）已知一个节点的索引为i，则其左子节点索引为2i，右子节点的索引为2i+1，父节点的索引为i/2  
   （以上所有的除法默认采用下取整）
4. 注意对于树的高度定义往往是以根节点高度为0

堆/优先队列的基本操作

1. 访问堆顶O（1）
2. 插入（O（logN）：  
   先放入堆底部，然后采用自顶向下的堆化
3. 堆化：O（logN）  
   1），自底向上：自底向上的交换元素，直到重新又满足堆的性质为止  
   2），自顶向下：与子节点中最大的元素交换，直至子节点都比当前访问节点要小
4. 堆顶元素出堆（OlogN）   
   交换堆顶元素与堆底元素，将堆底元素删除，然后采用自顶向下的堆化  
   之所以这个操作是因为，直接删除堆顶元素需要shift整个数组，复杂度为O（N）
5. 堆化数组：  
   1，利用自顶向下的堆化操作来实现，没输入一个元素就进行一次堆化操作O（NlogN）  
   2，将所有元素添加入堆，然后从最后一个非叶节点开始倒叙遍历堆，每遍历一个节点就对其使用自顶向下的堆化（O（N））  
   （这样就能保证再每次堆化元素的时候，所有的子树都是已经堆化好的，所以只需要对堆顶这一个元素进行自顶向下的堆化就可以了）  
   【如何估算这个复杂度】  
   假设为一个满二叉树：  
   1，对于节点高度为h-k（0<=k<=h）的一层，节点数有2^k，自顶向下堆化需要的操作数为，h-k  
   2，对每一层的所有操作数做累加，2^0\*h+2^1\*(h-1)+2^2\*(h-2)+……+2^(n-1)\*1=2^h; h=logn;  
   3，所以总的操作数为n；

典型案例：

返回一个数组中最大的k个数：（

1. 先将数组中的k个数字建堆
2. 后对于每一个数组中的数字，如果其大于堆顶元素  
   1，将堆顶元素弹出  
   2，将此元素入堆
3. 最后堆中剩下的元素即为需要的最大的k个元素

【复杂度分析】  
O（K）+（n-k）\*O（logK）<O(n\*log(k))

完全二叉树（complete tree）的索引的性质：

1. 若一个给定的节点的索引为i，则他的左子节点的索引为2\*i+1，右子节点的索引为2\*i+2，其父节点的索引为（i-1）/2（向下取整），如果索引越界，那么节点空或者节点不存在
2. 叶节点从数组的中间开始，即索引为n/2到n-1的节点；
3. 最后一个非叶子节点的索引：第一个叶节点的索引为n/2，故而最后一个·非叶子节点的索引为你n/2-1；

拓展d-heap

概念：一个节点最多连接d个子节点的树；

其性质：如果给定一个节点索引为i，则其第一个子节点索引为（i-1）d+2，最后一个子节点为id+1，父节点为（i+d-2）/d（向下取整）

（i-1）\*d+2；i\*d+1；（i+d-2）/d

代码模板：（辅助记忆）

下滤：

存储节点数值信息；

设置child变量为2\*p；

主循环（左子节点在树的范围内）

{

如果右子节点存在，并且右子节点的数值小于左子节点的数值，则设置子节点为右子节点；

如果节点值小于子节点数值，则break；

将节点的值由子节点的数值覆盖；

}

将初始节点值覆盖最后留下的节点位置；

上滤：

将初始节点的值保存；

主循环（没有遍历到root并且父节点大于此节点）

{

用父节点值覆盖此节点数值；

将p更新为parents；

将parents更新为p\*2；

}

最后将初始值覆盖最终的结束索引元素；

利用中序遍历将有序数组存为完全二叉树：

Inoder（索引，树的大小）

{

如果当前索引值超过了树的大小，那就停止递归；

Inoder（左子索引（2i），树的大小）

将有序数组的数值存入树的数组；

Inoder（右子索引（2i+1），树的大小）

}

【指引】中序遍历既可以让BST按序打印出有序的数列，也可以让有序的数列转换为BST！！！！！！（完全树）

【指引】一定要把完全二叉树的性质刻入脑子里！！！！！，很重要很重要