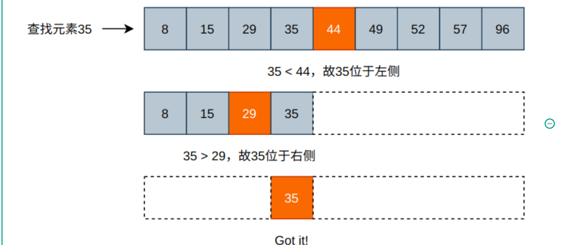
二分搜索树

二分查找

二分查找又称为折半查找，应用于一个有序序列中。在查找时，首先将序列从中间值一分为二，判断元素在前半部分还是后半部分，若在前半部分，则再将前半部分一分为二，再次判断。循环往复，最终找到需要的元素位置



每次查找时都会将原有有序序列元素个数减半，所以整体的查询平均时间复杂度为O(logn)，是一种非常高效的查找方法

基于数组实现的二分查找

```python
def binary_search(storage, value):
    # storage必须是有序序列，并且元素可比较

    left, right = 0, len(storage) - 1

    while left <= right:
        middle = left + (right - left) // 2

        if storage[middle] == value:
            return middle

        if storage[middle] < value:
            left = middle + 1
        else:
            right = middle - 1

    return None
```

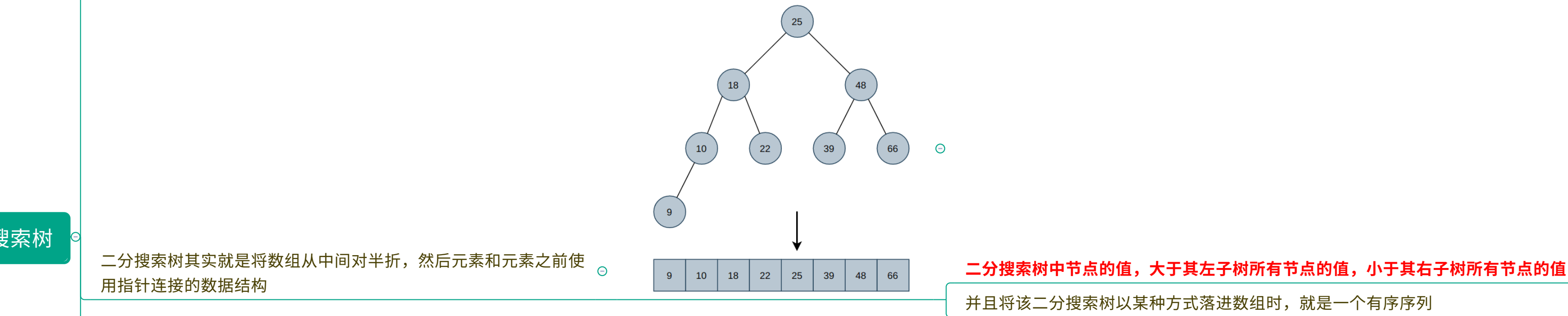
- 唯一需要注意的地方在于获取中间索引的方式，谨防整数越界
- Python虽说没有越界一说，但是数值越大的计算越慢

二分搜索树

既然有了基于链表的二分查找，为什么还需要有二分搜索树？

原因在于数组插入元素的低效性，除了定位元素的成本以外，为了保持数组元素的有序性，必须挪动响应的元素，当数组较大时，每一次的插入操作都将带来不菲的额外代价

二分搜索树其实就是将数组从中间对半拆，然后元素和元素之前使用指针连接的数据结构

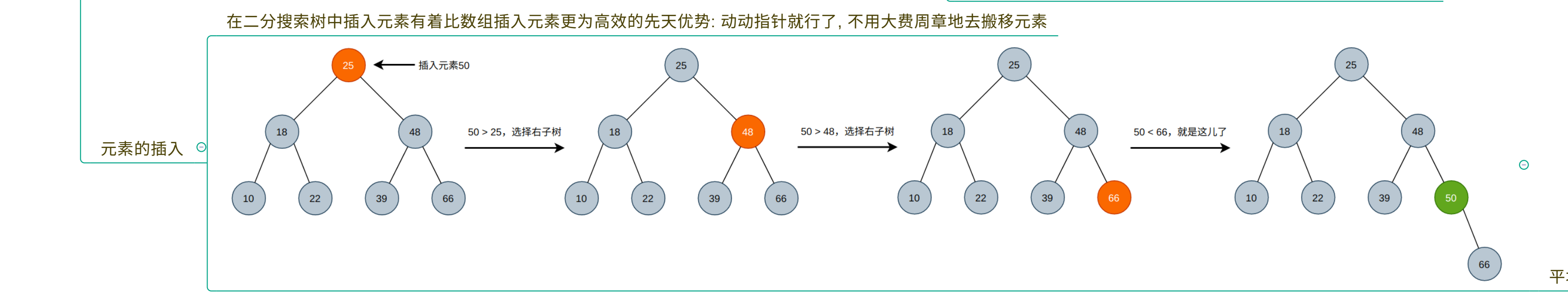


二分搜索树中节点的值，大于其左子树所有节点的值，小于其右子树所有节点的值

并且将该二分搜索树以某种方式落进数组时，就是一个有序序列

元素的插入

在二分搜索树中插入元素有着比数组插入元素更为高效的先天优势：动动指针就行了，不用大费周章地去搬移元素



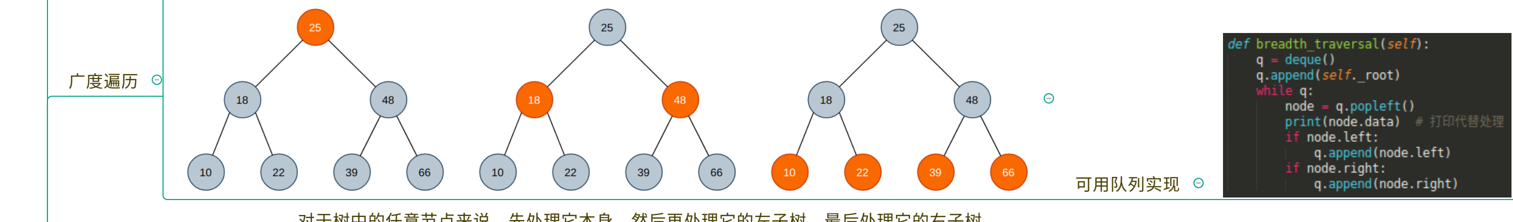
平均时间复杂度为O(n)

二叉树的遍历

在描述二分搜索树的其它操作之前，首先描述下二叉树的遍历，这对后续二分搜索树的操作理解会有帮助作用

广度遍历

广度遍历是指将二分搜索树逐层地进行遍历，首先从第一层开始，然后到第二层，再到第三层……直到没有下一层为止



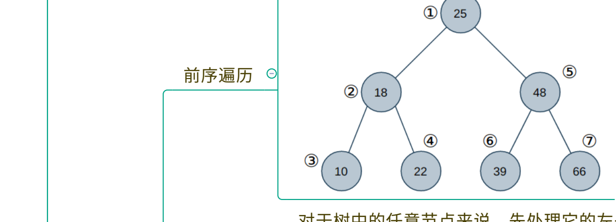
可用队列实现

```python
def breadth_traversal(self):
    q = deque()
    q.append(self._root)
    while q:
        node = q.popleft()
        print(node.data)  # 打印代替处理
        if node.left:
            q.append(node.left)
        if node.right:
            q.append(node.right)
```

深度遍历

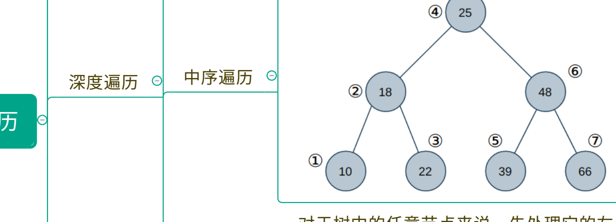
前序遍历

对于树中的任意节点来说，先处理它本身，然后再处理它的左子树，最后再处理它的右子树



中序遍历

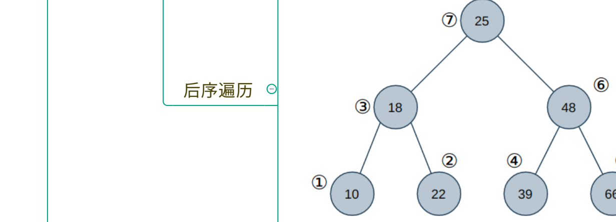
对于树中的任意节点来说，先处理它的左子树，然后再处理它本身，最后再处理它的右子树



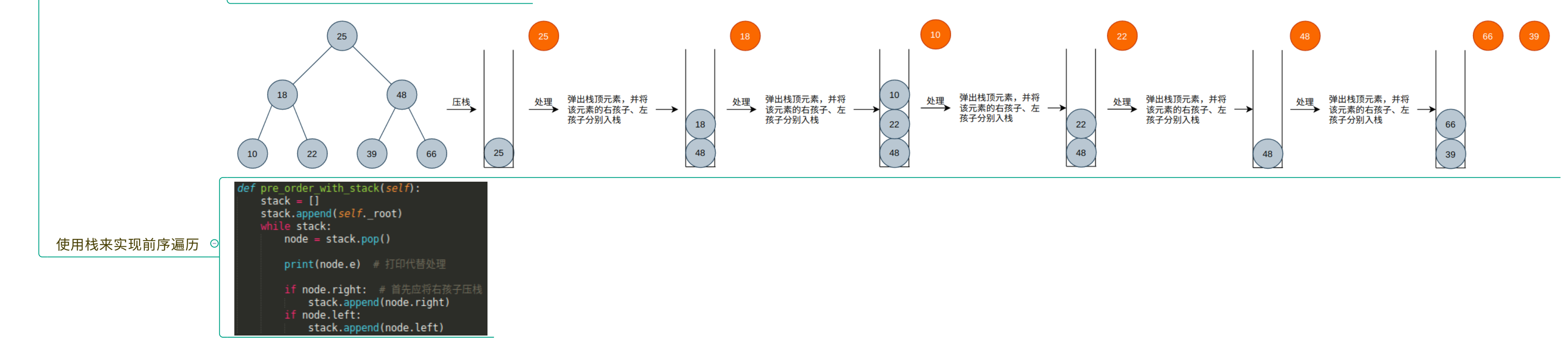
可以看到，二分搜索树的中序遍历结果就是有序数组的原有结果

后序遍历

对于树中的任意节点来说，先处理它的左子树，然后再处理它的右子树，最后再处理它本身



使用栈来实现前序遍历




```python
def pre_order_with_stack(self):
    stack = []
    stack.append(self._root)
    while stack:
        node = stack.pop()

        print(node.e)  # 打印代替处理

        if node.right:  # 首先应将右孩子压栈
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
```