2 树

1.1 12.1-4: 对于一棵有 n 个结点的树，请设计在 O(n)时间内完成的先序遍历和后序遍历算法（注意：这里不一定是二叉树）

```
1:  function PreorderTraversal(root)
2:      ▷ Perform pre-order traversal of the tree
3:      if root is null then
4:          return
5:      Visit (root)
6:      for child in root.children do
7:          PreorderTraversal(child)
8:
9:  function PostorderTraversal(root)
10:     ▷ Perform post-order traversal of the tree
11:     if root is null then
12:         return
13:     for child in root.children do
14:         PostorderTraversal(child)
15:     Visit (root)
```

1.2 12.2-2 写出 BST 树查找最小值和最大值的伪代码（或是 Python/C 代码），并证明给出的算法复杂度在 O(n)内。

```
1:  function FindMin(root)
2:      ▷ Find the minimum value in a BST
3:      if root is null then
4:          return null
5:      while root.left is not null do
6:          root ← root.left
7:      return "root.value"
8:
9:  function FindMax(root)
10:     ▷ Find the maximum value in a BST
11:     if root is null then
12:         return null
13:     while root.right is not null do
14:         root ← root.right
15:     return root.value
```

在最坏情况下，BST 可能退化成一条链，此时树的高度为 n，其中 n 是节点的数量。因此，在最坏情况下，算法的时间复杂度是 $O(n)$。

因此算法时间复杂度在 $O(n)$ 内。

1.3 12.2-3 写出 BST 里查找前驱的伪代码，并证明给出的算法复杂度在 O(n)内。这里前驱是指给定一个 key，BST 比 key 小的最大值。

```
1:  function FindPredecessor(root, key)
```

```
2:      ▷ Find the predecessor (maximum value smaller than key) in a BST
3:
4:      predecessor ← null
5:      while root is not null do
6:          if root.value < key then
7:              predecessor ← root
8:              root ← root.right
9:          else
10:             root ← root.left
11:     return predecessor.value
```

在最坏情况下，BST 可能退化成一条链，此时树的高度为 n。在每一次迭代中，算法都会沿着树的高度移动一个节点。

因此，在最坏情况下，算法的时间复杂度是 $O(n)$。

1.4 针对前述数据库系统和 range index（指单选题和多选题中的系统），构建一个操作 LIST(l, h)，即给定 l 和 h，l < h，返回 range index 中所有在 l 和 h 之间的 key。显然这个操作无法在低于 O(N)的复杂度内完成（思考这是为什么）。我们将 LIST 的复杂度定义为 T(N) + O(L)，这里的 L=l-h。我们要求给出一种算法，使得 T(N)是低于线性复杂度的，伪代码如下

…

1.4.1 说明为何 LIST 无法低于 O(N)的时间复杂度完成。

LIST 操作需要返回范围索引中在给定范围 [l, h] 内的所有键。由于我们无法事先知道树中哪些节点的键位于指定的范围内，因此必须检查所有节点以确定它们是否在范围内。这就意味着算法需要访问树中的每个节点，因此其复杂度将是树中节点数量的线性函数，即 O(N)。

1.4.2 说明上述的 LCA 的用处，即 LCA 返回的是什么？

LCA（Lowest Common Ancestor）的返回值是范围 [l, h] 中的最近公共祖先节点。最近公共祖先是指在给定的范围内，键值既大于等于 l 又小于等于 h 的最深的节点。

在 LIST 操作中，LCA 的作用是找到范围内的一个起始点，以便从该点开始进行进一步的节点检查。

1.4.3 证明 LCA 算法的正确性，从而进一步证明 LIST 算法的正确性

1. LCA 返回的节点 lca 一定满足 $l \leq \text{lca.key} \leq h$。
2. LCA 返回的节点 lca 是范围 [l, h] 中的最近公共祖先。
3. LCA 算法的终止条件是 `node = NIL` 或 `(l ⩽ node.key and h ⩾ node.key)`，确保 LCA 返回的节点一定在范围内

1.5 考虑平衡二叉树的概念。我们称高度为$O(\lg(N))$的二叉树为平衡二叉树。请问，平衡二叉树每一个结点不是叶子就是拥有两个子结点的论断是否正确？若正确请证明，不正确请举反例。

不正确。

这是一颗平衡二叉树，但 C 节点不是叶子节点，也不拥有两个子结点。

1.6 考虑平衡二叉树的概念。我们称高度$O(\lg(N))$的二叉树为平衡二叉树。请问：若一棵树中的每一棵子树都包含2^k-1个结点，这里 k 是整数且对应不同的子树是不同的，那么这是否是一棵平衡二叉树？若是请证明，若不是，请举反例。（提示：使用数学归纳法证明）

1. 对于高度为 1 的树，只有一个根节点，显然 $2^1-1=1$，基础情况成立。
2. 假设对于任意高度为 h 的子树，包含的节点数为 2^h-1。考虑高度为 $h+1$ 的树的根节点有两个子节点，它们都是高度为 h 的子树。根据归纳假设，每个子树包含 2^h-1 个节点，因此整个高度为 $h+1$ 的子树包含的节点数为 $2\times\left(2^h-1\right)+1=2^{h+1}-2+1=2^{h+1}-1$。
3. 由归纳法可知，对于任意高度 h，包含 2^h-1 个节点的子树都符合条件。因此，如果一棵树中的每一棵子树都包含 2^k-1 个结点，其中 k 是整数，那么这棵树是一棵平衡二叉树。

1.7 考虑平衡二叉树的概念。我们称高度$O(\lg(N))$的二叉树为平衡二叉树。请问：对于一棵树，若存在一个常量c，对于这棵树中的任一结点，其左右子树的高度差至多为c，那么这是否是一棵平衡二叉树？若是请证明，若不是，请举反例。（提示：使用数学归纳法证明，通过证明满足题干条件的树，有$n(h)\geq\left(1+\alpha\right)^h-1$，从而得到 h 的高度不超过$O(\lg N)$。注意高度差$c$在这里的应用。）

1. 对于高度为 1 的树，最小节点数为 1，而 $\left(1+\alpha\right)^1-1=\alpha$显然成立。
2. 假设对于任意高度为 h 的树，最小节点数为 $n(h)\geq\left(1+\alpha\right)^h-1$。
3. 考虑高度为 $h+1$ 的树。根据题意，该树的任一结点的左右子树高度差至多为 c，因此可以将树拆分成一个根节点和两个子树，每个子树的高度至多为 h 或 $h-1$。

   最小节点数 $n(h+1)$ 为根节点的节点数加上两个子树的节点数之和。根据归纳假设，我们有：

   $$n(h+1)\geq n(h)+\left(1+\alpha\right)^h-1+\left(1+\alpha\right)^{h-1}-1$$

   …