不同二叉树的生成问题

1.数列或者有序列的生成问题

1-a 有序列的生成问题

1-a-(1) 二叉搜索树的生成问题

对于二叉搜索树而言，其定义为root的value大于左边子树的value(左子树如果存在的话)，root的value小于右边子树的value(右子树如果存在的话)。

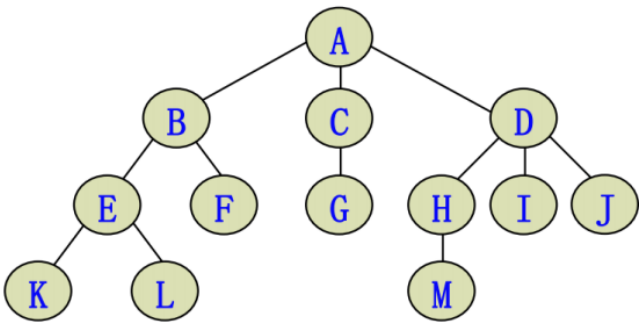
对于二叉搜索树而言，有序列的生成问题或者数列的生成问题可以化简为一种序列减少的有序列的生成问题，其核心思想在于二叉搜索树可以用带关键字的广义表表示(有序),而生成过程就是该表示有序列的广义表的扩大化。

所以，采用的方法可以是回溯。

补充：树的广义表的表示

- 树的表示：

（4）广义表表示法

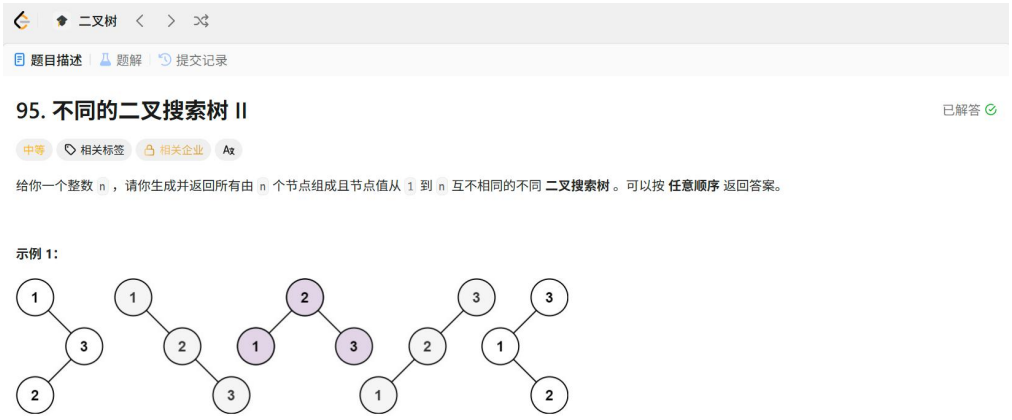


(A) 第一层

(A(B(E,F), C(G), D(H,I,J))) 第三层

(A(B(E(K,L),F), C(G), D(H(M),I,J))) 第四层

LEETCODE例题：



思路构造

1.首先构造一个有序序列，即从1到n的递增序列，即未经过关键字筛选的二叉树的单调递增序列。
2.构造回溯结构：对于已排序的序列来说，其中子问题高度重复，对于一个root节点来说，只需找到其左边的二叉搜索树和右边的二叉搜索树就可以构造出完整的二叉搜索树。
3.对于有序序列的子问题，只需缩短有序序列的排序范围即可。
4.由于是从大序列递减，故为回溯算法。

方法一：回溯

思路与算法

二叉搜索树关键的性质是根节点的值大于左子树所有节点的值，小于右子树所有节点的值，且左子树和右子树也同样为二叉搜索树。因此在生成所有可行的二叉搜索树的时候，假设当前序列长度为 n，如果我们枚举根节点的值为 i，那么根据二叉搜索树的性质我们可以知道左子树的节点值的集合为 [1⋯i−1]，右子树的节点值的集合为 [i+1⋯n]。而左子树和右子树的生成相较于原问题是一个序列长度缩小的子问题，因此我们可以想到用回溯的方法来解决这道题目。

我们定义 generateTrees(start, end) 函数表示当前值的集合为 [start,end]，返回序列 [start,end] 生成的所有可行的二叉搜索树。按照上文的思路，我们考虑枚举 [start,end] 中的值 i 为当前二叉搜索树的根，那么序列划分为了 [start,i−1] 和 [i+1,end] 两部分。我们递归调用这两部分，即 generateTrees(start, i - 1) 和 generateTrees(i + 1, end)，获得所有可行的左子树和可行的右子树，那么最后一步我们只要从可行左子树集合中选一棵，再从可行右子树集合中选一棵拼接到根节点上，并将生成的二叉搜索树放入答案数组即可。

递归的入口即为 generateTrees(1, n)，出口为当 start>end 的时候，当前二叉搜索树为空，返回空节点即可。

```
struct TreeNode** buildTree(int start, int end, int* returnSize) {
    if (start > end) {
        (*returnSize) = 1;
        struct TreeNode** ret = malloc(sizeof(struct TreeNode*));
        ret[0] = NULL;
        return ret;
    }
    *returnSize = 0;
    struct TreeNode** allTrees = malloc(0);
    // 枚举可行根节点
    for (int i = start; i <= end; i++) {
        // 获得所有可行的左子树集合
        int leftTreesSize;
        struct TreeNode** leftTrees = buildTree(start, i - 1, &leftTreesSize);

        // 获得所有可行的右子树集合
        int rightTreesSize;
        struct TreeNode** rightTrees = buildTree(i + 1, end, &rightTreesSize);

        // 从左子树集合中选出一棵左子树，从右子树集合中选出一棵右子树，拼接到根节点上
        for (int left = 0; left < leftTreesSize; left++) {
            for (int right = 0; right < rightTreesSize; right++) {
                struct TreeNode* currTree = malloc(sizeof(struct TreeNode));
                currTree->val = i;
                currTree->left = leftTrees[left];
                currTree->right = rightTrees[right];

                (*returnSize)++;
                allTrees = realloc(allTrees, sizeof(struct TreeNode*) * (*returnSize));
                allTrees[(*returnSize) - 1] = currTree;
            }
        }
        free(rightTrees);
        free(leftTrees);
    }
    return allTrees;
}

struct TreeNode** generateTrees(int n, int* returnSize) {
    if (!n) {
        (*returnSize) = 0;
        return NULL;
    }
    return buildTree(1, n, returnSize);
}
```

复杂度分析

- 时间复杂度：整个算法的时间复杂度取决于「可行二叉搜索树的个数」，而对于 n 个点生成的二叉搜索树数量等价于数学上第 n 个「卡特兰数」，用 G_n 表示。卡特兰数具体的细节请读者自行查询，这里不再赘述，只给出结论。生成一棵二叉搜索树需要 $O(n)$ 的时间复杂度，一共有 G_n 棵二叉搜索树，也就是 $O(nG_n)$。而卡特兰数以 $\frac{4^n}{n^{3/2}}$ 增长，因此总时间复杂度为 $O(\frac{4^n}{n^{1/2}})$。
- 空间复杂度：n 个点生成的二叉搜索树有 G_n 棵，每棵有 n 个节点，因此存储的空间需要 $O(nG_n) = O(\frac{4^n}{n^{1/2}})$，递归函数需要 $O(n)$ 的栈空间，因此总空间复杂度为 $O(\frac{4^n}{n^{1/2}})$。