题目地址(109. 有序链表转换二叉搜索树)

https://leetcode-cn.com/problems/convert-sorted-list-to-binary-search-tree/

入选理由

1. 和二叉搜索树联动，大家可以提前预习一下。
2. 链表的题目，我们核心思路就是链表的基本操作，别想别的。比如链表上的快排，如果你不会快排，那么肯定做不出来，但不表示你不会链表，因此大家学习的时候一定要分清这些。

难度

- 中等

标签

- 链表
- 二叉搜索树

题目描述

给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

示例:

给定的有序链表： [-10, -3, 0, 5, 9],

一个可能的答案是：[0, -3, 9, -10, null, 5], 它可以表示下面这个高度平衡二叉搜索树:

```
      0
     / \
   -3   9
   /   /
 -10  5
```

前置知识

- 递归
- 二叉搜索树的任意一个节点，当前节点的值必然大于所有左子树节点的值。同理,当前节点的值必然小于所有右子树节点的值

双指针法

思路

使用快慢双指针可定位中间元素，具体可参考双指针的讲义。这里我简单描述一下算法流程：

1. 获取当前链表的中点
2. 以链表中点为根
3. 中点左边的值都小于它,可以构造左子树
4. 同理构造右子树
5. 循环第一步

具体算法：

1. 定义一个快指针每步前进两个节点，一个慢指针每步前进一个节点
2. 当快指针到达尾部的时候，正好慢指针所到的点为中点

代码

代码支持：JS,Java,Python,C++

JS Code

```
var sortedListToBST = function (head) {
  if (!head) return null;
  return dfs(head, null);
};

function dfs(head, tail) {
  if (head == tail) return null;
  let fast = head;
  let slow = head;
  while (fast != tail && fast.next != tail) {
    fast = fast.next.next;
    slow = slow.next;
  }
```

```
  let root = new TreeNode(slow.val);
  root.left = dfs(head, slow);
  root.right = dfs(slow.next, tail);
  return root;
}
```

Java Code:

```
class Solution {
  public TreeNode sortedListToBST(ListNode head) {
      if(head == null) return null;
      return dfs(head,null);
  }
  private TreeNode dfs(ListNode head, ListNode tail){
      if(head == tail) return null;
      ListNode fast = head, slow = head;
      while(fast != tail && fast.next != tail){
          fast = fast.next.next;
          slow = slow.next;
      }
      TreeNode root = new TreeNode(slow.val);
      root.left = dfs(head, slow);
      root.right = dfs(slow.next, tail);
      return root;
  }
}
```

Python Code:

```
class Solution:
    def sortedListToBST(self, head: ListNode) -> TreeNode:
        if not head:
            return head
        pre, slow, fast = None, head, head

        while fast and fast.next:
            fast = fast.next.next
            pre = slow
            slow = slow.next
        if pre:
            pre.next = None
        node = TreeNode(slow.val)
        if slow == fast:
            return node
        node.left = self.sortedListToBST(head)
        node.right = self.sortedListToBST(slow.next)
        return node
```

C++ Code:

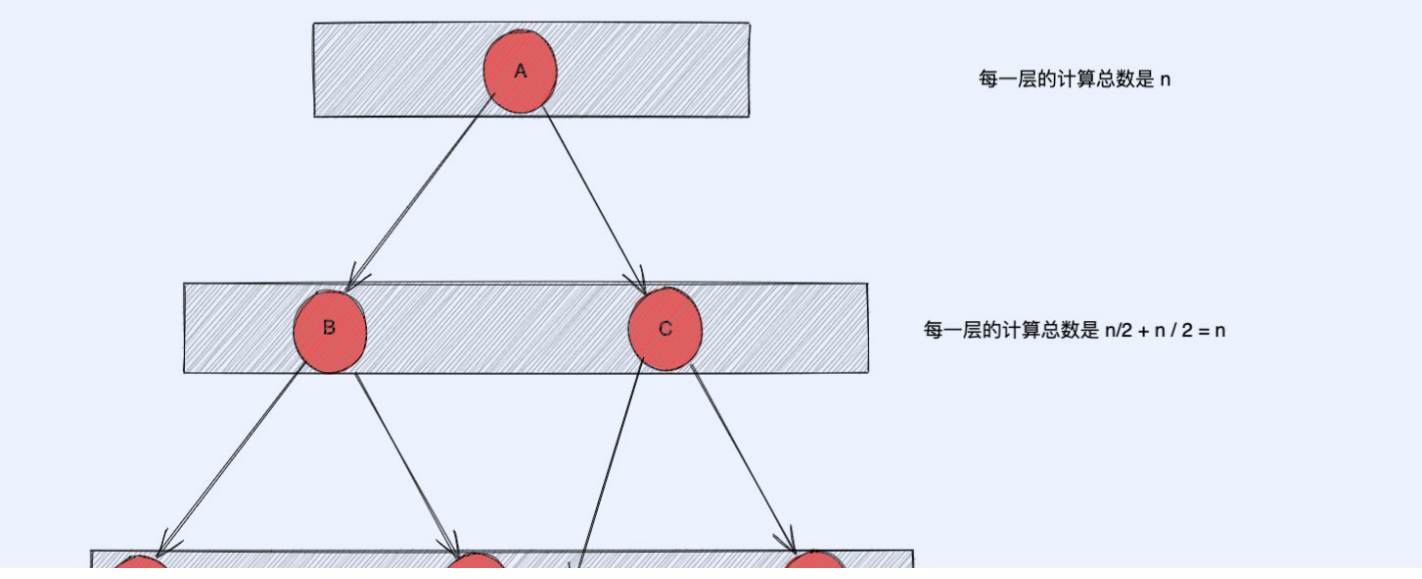
```
class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        if (head == nullptr) return nullptr;
        return sortedListToBST(head, nullptr);
    }
    TreeNode* sortedListToBST(ListNode* head, ListNode* tail) {
        if (head == tail) return nullptr;

        ListNode* slow = head;
        ListNode* fast = head;

        while (fast != tail && fast->next != tail) {
            slow = slow->next;
            fast = fast->next->next;
        }

        TreeNode* root = new TreeNode(slow->val);
        root->left = sortedListToBST(head, slow);
        root->right = sortedListToBST(slow->next, tail);
        return root;
    }
};
```

复杂度分析

令 n 为链表长度。

- 时间复杂度：递归树的深度为 $\log n$，每一层的基本操作数为 n，因此总的时间复杂度为 $O(n \log n)$
- 空间复杂度：空间复杂度为 $O(\log n)$

有的同学不太会分析递归的时间复杂度和空间复杂度，我们在这里给大家再次介绍一下。



首先我们尝试画出如上的递归树。由于递归树的深度为 $\log n$ 因此空间复杂度就是 $\log n$ * 递归函数内部的空间复杂度，由于递归函数内空间复杂度为 $O(1)$，因此总的空间复杂度为 $O(\log n)$。

时间复杂度稍微困难一点点。之前西法在先导篇给大家说过：**如果有递归那就是：递归树的节点数 * 递归函数内部的基础操作数**。而这句话的前提是所有递归函数内部的基本操作数是一样的，这样才能直接乘。而这里递归函数的基本操作数不一样。

不过我们发现递归树内部每一层的基本操作数都是固定的， 为啥固定已经在图上给大家算出来了。因此总的空间复杂度其实可以通过**递归深度 * 每一层基础操作数**计算得出，也就是 $n\log n$。 类似的技巧可以用于归并排序的复杂度分析中。

另外大家也直接可以通过公式推导得出。对于这道题来说，设基本操作数 T(n)，那么就有 T(n) = T(n/2) * 2 + n/2，推导出来 T(n) 大概是 nlogn。这应该高中的知识。 具体推导过程如下：

$$T(n) = T(n/2) * 2 + n/2 = \frac{n}{2} + 2 * (\frac{n}{2})^2 + 2^2_(\frac{n}{2})^3 + ... = \log n * \frac{n}{2}$$

类似地，如果递推公式为 T(n) = T(n/2) * 2 + 1 ， 那么 T(n) 大概就是 logn。

缓存法

思路

因为访问链表中点的时间复杂度为$O(n)$,所以可以使用数组将链表的值存储,以空间换时间。

代码

代码支持：JS, C++

JS Code:

```
var sortedListToBST = function (head) {
  let res = [];
  while (head) {
    res.push(head.val);
    head = head.next;
  }
  return dfs(res, 0, res.length - 1);
};

function dfs(res, l, r) {
  if (l > r) return null;
  let mid = parseInt((l - r) / 2 + r);
  let root = new TreeNode(res[mid]);
  root.left = dfs(res, l, mid - 1);
```

```
    root.right = dfs(res, mid + 1, r);
    return root;
}
```

C++ Code:

```
class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        vector<int> nodes;
        while (head != nullptr) {
            nodes.push_back(head->val);
            head = head->next;
        }
        return sortedListToBST(nodes, 0, nodes.size());
    }
    TreeNode* sortedListToBST(vector<int>& nodes, int start, int end) {
        if (start >= end) return nullptr;

        int mid = (end - start) / 2 + start;
        TreeNode* root = new TreeNode(nodes[mid]);
        root->left = sortedListToBST(nodes, start, mid);
        root->right = sortedListToBST(nodes, mid + 1, end);
        return root;
    }
};
```

复杂度分析

令 n 为链表长度。

- 时间复杂度：递归树每个节点的时间复杂度为 $O(1)$，每次处理一个节点，因此总的节点数就是 n， 也就是说总的时间复杂度为$O(n)$。
- 空间复杂度：使用了数组对链表的值进行缓存，空间复杂度为$O(n)$

上一页 下一页