树——二叉树的构造

如果是标准的二叉树遍历结果，只要知道中序遍历的结果，搭配任意一个遍历的结果都可以唯一确定的构造一棵二叉树。

根据二叉树的前序和中序遍历结果构造二叉树

- [x] Leetcode 105.从前序与中序遍历序列构造二叉树

1. 前序的第一个为根，在中序中找到根的位置。
2. 中序中根的左右两边即为左右子树的中序遍历。同时可知左子树的大小size-left。
3. 前序中根接下来的size-left个是左子树的前序遍历。
4. 由此可以递归处理左右子树。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        std::ios_base::sync_with_stdio(false);
      cin.tie(NULL);
      cout.tie(NULL);

        int m = preorder.size(), n = inorder.size();
```

```
        return build(0, m, 0, n, preorder, inorder);
    }

    TreeNode *build(int pre_start, int pre_end, int in_start, int in_end, vector<int>& preorder, vector<int>& inorder)
    {
        if (pre_start == pre_end || in_start == in_end) return NULL;

        TreeNode *root = new TreeNode(preorder[pre_start]);
        int pos = in_start;
        for (int i = in_start; i < in_end; ++i) {
            if (inorder[i] == preorder[pre_start]) {
                pos = i; break;
            }
        }
        int leftSize = pos - in_start;

        root -> left = build(pre_start + 1, pre_start + 1 + leftSize, in_start, pos, preorder, inorder);
        root -> right = build(pre_start + 1 + leftSize, pre_end, pos + 1, in_end, preorder, inorder);
        return root;
    }
};
```

- [x] 一本通-1366：二叉树输出(btout)

题目的意思是要根据每个节点的长度然后前序遍历输出，关键点在于如何确定节点长度。在二叉树的每个节点额外保存一个频率信息`freq`，首先是根据前序和中序遍历结果建树，然后递归的去计算节点的频率信息，最后递归实现前序遍历即可。

```
#include <bits/stdc++.h>

using namespace std;
```

```
4
5  string preorder, inorder;
6
7  struct TreeNode
8  {
9    char ch;
10   int freq;
11   TreeNode *left, *right;
12   TreeNode(char x): ch(x), freq(0), left(NULL), right(NULL) {}
13 };
14
15
16 TreeNode *build(int pre_start, int pre_end, int in_start, int
   in_end)
17 {
18   if (pre_start == pre_end || in_start == in_end) return NULL;
19
20   TreeNode *root = new TreeNode(preorder[pre_start]);
21   int pos = inorder.find(preorder[pre_start]);
22   int leftSize = pos - in_start;
23
24   root -> left = build(pre_start + 1, pre_start + 1 + leftSize,
   in_start, pos);
25   root -> right = build(pre_start + 1 + leftSize, pre_end, pos
   + 1, in_end);
26
27   return root;
28 }
29
30
31 void makeEmpty(TreeNode *&root)
32 {
33   if (root) {
34     makeEmpty(root -> left);
35     makeEmpty(root -> right);
```

```
    delete root;
    root = NULL;
  }
}

void calculateFreq(TreeNode *root)
{
  if (!root) return;
  if (!root -> left && !root -> right) {
    root -> freq = 1;
    return;
  }

  calculateFreq(root -> left);
  calculateFreq(root -> right);
  root -> freq = (root -> left ? root -> left -> freq : 0)
    + (root -> right ? root -> right -> freq : 0);
}

void preorderTraversal(TreeNode *root)
{
  if (!root) return;

  for (int i = 0; i < root -> freq; ++i) {
    cout << root -> ch;
  }
  cout << endl;
  preorderTraversal(root -> left);
  preorderTraversal(root -> right);
}

int main()
{
  std::ios_base::sync_with_stdio(false);
  cin.tie(NULL);
```

```
  cout.tie(NULL);

  cin >> preorder >> inorder;

  TreeNode *root = build(0, preorder.size(), 0,
inorder.size());
  calculateFreq(root);

  preorderTraversal(root);

  makeEmpty(root);

  return 0;
}
```

- [x] 一本通-1339：【例3-4】求后序遍历

```
#include <bits/stdc++.h>

using namespace std;

string preorder, inorder;

struct TreeNode
{
  char ch;
  TreeNode *left, *right;
  TreeNode(char x): ch(x), left(NULL), right(NULL) {}
};

TreeNode *treeBuild(int in_start, int in_end, int pre_start,
int pre_end)
{
  if (in_start == in_end || pre_start == pre_end) return NULL;

```

```
  TreeNode *root = new TreeNode(preorder[pre_start]); //建立根节点

  int inRootPos = inorder.find(preorder[pre_start]);
  int leftSize = inRootPos - in_start;

  root -> left = treeBuild(in_start, inRootPos, pre_start + 1, pre_start + 1 + leftSize);
  root -> right = treeBuild(inRootPos + 1, in_end, pre_start + 1 + leftSize, pre_end);

  return root;
}

void postorderTraversal(TreeNode *root)
{
  if (!root) return;
  postorderTraversal(root -> left);
  postorderTraversal(root -> right);
  cout << root -> ch;
}

void makeEmpty(TreeNode *&root)
{
  if (root) {
    makeEmpty(root -> left);
    makeEmpty(root -> right);
    delete root;
    root = NULL;
  }
}


int main()
{
```

```
  std::ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);

  cin >> preorder >> inorder;

  TreeNode *root = treeBuild(0, inorder.size(), 0,
preorder.size());
  postorderTraversal(root);
  makeEmpty(root);

  return 0;
}
```

根据二叉树的中序和后序遍历结果构造二叉树

- [x] LeetCode 106.从中序与后序遍历序列构造二叉树

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    using Iter = vector<int>::iterator;
    TreeNode* buildTreeFromInAndPost(Iter in_start, Iter
in_end, Iter post_start, Iter post_end){
        if(in_start == in_end) return nullptr;
        if(post_start == post_end) return nullptr;
```

```
16
17          auto root = new TreeNode(*(post_end - 1));
18          auto inRootPos = find(in_start, in_end, *(post_end-1));
19          auto rightSzie = distance(inRootPos+1, in_end);
20
21          root -> right = buildTreeFromInAndPost(inRootPos+1,
    in_end, post_end-1-rightSzie, post_end-1);
22          root -> left = buildTreeFromInAndPost(in_start,
    inRootPos, post_start, post_end - 1 - rightSzie);
23
24          return root;
25      }
26
27
28      TreeNode* buildTree(vector<int>& inorder, vector<int>&
    postorder) {
29          return buildTreeFromInAndPost(inorder.begin(),
    inorder.end(), postorder.begin(), postorder.end());
30      }
31  };
```

根据二叉树的中序和层序遍历结果构造二叉树

- [x] 一本通-1364：二叉树遍历(flist)

层序遍历的每一层的节点都是下一层的根节点，以测试用例为例：

首先根据层序遍历的第一个元素可知，A是根节点，然后在中序遍历里找到A，那么A的左边的元素就是左子树的元素，右边的元素是右子树的元素。这里我们判断左右两边是否存在元素，因为如果不存在元素的话，层序遍历的结果里是不会出现的。更确切的讲，比如A的右边无元素，那么意味着层序遍历的结果里第二层只有B一个元素。

用变量`pos`记录上一个根节点的位置，只需要从根节点往后开始寻找即可，比如到了B，位于左子树，只需要从DBE中查找，因为层序遍历是按照每一层元素出现的先后顺序显示的，那么最先匹配的就是先序遍历的根，所以直接输出即可。

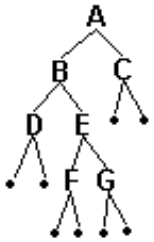
```cpp
#include <bits/stdc++.h>

using namespace std;

string inorder, levelorder;
int n;

void solve(int pos, int start, int end)
{
  bool flag = false;
  int j;
  for (int i = pos; i < n; ++i) {
    for (j = start; j <= end; ++j) {
      if (inorder[j] == levelorder[i]) {
        cout << levelorder[i];
        flag = true; break;
      }
    }
    if (flag) break;
  }

  if (start < j) solve(pos + 1, start, j - 1);
  if (j < end) solve(pos + 1, j + 1, end);
}

int main()
{
  std::ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);

  cin >> inorder >> levelorder;
  n = inorder.size();
  solve(0, 0, n - 1);
```

```
35
36    return 0;
37  }
```

扩展二叉树

- [x] 一本通-1340：【例3-5】扩展二叉树

由于先序、中序和后序序列中的任一个都不能唯一确定一棵二叉树，所以对二叉树做如下处理，将二叉树的空结点用·补齐，如图所示。我们把这样处理后的二叉树称为原二叉树的扩展二叉树，扩展二叉树的先序和后序序列能唯一确定其二叉树。



现给出扩展二叉树的先序序列，要求输出其中序和后序序列。

```cpp
#include <bits/stdc++.h>

using namespace std;

class Tree
{
  struct TreeNode {
    char ch;
    TreeNode *left, *right;
    TreeNode(char x): ch(x), left(NULL), right(NULL) {}
  };
  TreeNode *root;
  int pos;

  void build(const string & s, TreeNode *& root)
  {
```

```
16  {
17    if (s[++pos] != '.') {
18      root = new TreeNode(s[pos]);
19      build(s, root -> left);
20      build(s, root -> right);
21    }
22    else root = NULL;
23  }
24
25  void inorderTraversal(TreeNode *root)
26  {
27    if (root) {
28      inorderTraversal(root -> left);
29      cout << root -> ch;
30      inorderTraversal(root -> right);
31    }
32  }
33
34  void postorderTraversal(TreeNode *root)
35  {
36    if (root) {
37      postorderTraversal(root -> left);
38      postorderTraversal(root -> right);
39      cout << root -> ch;
40    }
41  }
42
43  void makeEmpty(TreeNode *& root)
44  {
45    if (root) {
46      makeEmpty(root -> left);
47      makeEmpty(root -> right);
48      delete root;
49      root = NULL;
50    }
51  }
```

```
  }

public:
  Tree(): root(NULL), pos(-1) {}

  ~Tree() { makeEmpty(root); }

  void build(const string & s){ build(s, root); }

  void inorderTraversal() { inorderTraversal(root); }

  void postorderTraversal() { postorderTraversal(root); }
};


int main()
{
  std::ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);

  string s;
  cin >> s;
  Tree obj;
  obj.build(s);

  obj.inorderTraversal();
  cout << endl;
  obj.postorderTraversal();
  cout << endl;

  return 0;
}
```

本通1268：对称二叉树(tree.c)

一本通-1368：对称二叉树(tree_c)

核心是通过层序遍历结果构建二叉树，然后递归检查，也可以非递归的方法检查。但是也可以不建树的方法直接求得结果。

但是这道题还是可以练习如何通过扩展的层序遍历结果来构建二叉树（略微改编了题目）

与这道题相关联的是LeetCode 101.Symmetric Tree，注意题目背景区别还是很大的。

回到本题，我们可以发现其实给出的层序遍历恰好是一棵完全二叉树，那么我们就可以利用下标关系来进行检验了。相应的建树也变得很容易了。

```
//不建树，利用下标关系检查
#include <bits/stdc++.h>

using namespace std;

vector<char> s(1005);
int n = 0;

void solve()
{
  bool flag = true;

  for (int i = 1; i <= n; ++i) {
    int leftPos = 2 * i;
    int rightpos = leftPos + 1;
    if (leftPos > n && rightpos > n) break; //到了最后一层
    if (leftPos <= n && rightpos > n) { flag = false; break; }
    if ((s[leftPos] == '#' && s[rightpos] != '#')
      || (s[leftPos] != '#' && s[rightpos] == '#')) {
      flag = false; break;
    }

  }
```

```
    }

    if (flag) cout << "Yes" << endl;
    else cout << "No" << endl;
}


int main()
{
    std::ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

    while (cin >> s[++n]) {}
    --n;

    solve();

    return 0;
}
```

注意36行的`--n`，因为当最后一个空输入的时候，`n`的数值还是增加了。

那么我们来练习一下根据层序遍历的结果来建树。

```
//练习根据层序遍历结果建树
#include <bits/stdc++.h>

using namespace std;

vector<char> s(1005);
int n = 0;

struct TreeNode
{
    char val;
```

```
  char val;
  TreeNode *left, *right;
  TreeNode(char x): val(x), left(NULL), right(NULL) {}
};

TreeNode *build(int pos)
{
  if (pos > n || s[pos] == '#') return NULL;

  TreeNode *root = new TreeNode(s[pos]);
  root -> left = build(2 * pos);
  root -> right = build(2 * pos + 1);

  return root;
}

void makeEmpty(TreeNode *&root)
{
  if (root) {
    makeEmpty(root -> left);
    makeEmpty(root -> right);
    delete root;
    root = NULL;
  }
}

bool isSymmetric(TreeNode *root)
{
  if (!root) return true;
  if ((root -> left && !root -> right)
    || (!root -> left && root -> right)) return false;

  return isSymmetric(root -> left) && isSymmetric(root ->
right);
}
```

```
void solve()
{
  TreeNode *root = build(1);
  bool flag = isSymmetric(root);
  makeEmpty(root);

  if (flag) cout << "Yes" << endl;
  else cout << "No" << endl;
}


int main()
{
  std::ios_base::sync_with_stdio(false);
  cin.tie(NULL);
  cout.tie(NULL);

  while (cin >> s[++n]) {}
  --n;
  solve();

  return 0;
}
```

- [x] LeetCode 297.二叉树的序列化与反序列化

这道题其实就是一本通-1340：【例3-5】扩展二叉树的一个翻版，只不过现在需要自己手动实现扩展二叉树的输出，并且有一个很重要的不同点，相比于一本通1340里的单个字母，二叉树里的节点存储的数值可能不止一位，所以需要用一个特殊的标记符号`#`来对数据进行分隔，用`.`代表空节点。

序列化部分采用前序遍历的递归实现，注意需要去掉末尾的`#`符号，用`pos`记录处理到序列的哪个位置，每次指向第一个数字，然后递归构建左右子树。

```
/**
```

```
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        std::ios_base::sync_with_stdio(false);
      cin.tie(NULL);
      cout.tie(NULL);

      string res;
      if (!root) return res;

      preorderTraversal(root, res);

      return res.substr(0, res.size() - 1);
    }

    void preorderTraversal(TreeNode *root, string & res)
    {
      if (!root) {
        res += ".#"; return;
      }

      res += to_string(root -> val);
      res.push_back('#');
      preorderTraversal(root -> left, res);
      preorderTraversal(root -> right, res);
```

```
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
      if (data.empty()) return NULL;

        int pos = 0;
        TreeNode *root;
        build(data, root, pos);
        return root;
    }

    void build(string & data, TreeNode *&root, int &pos)
    {
      int nextPos = data.find("#", pos);
      if (nextPos == string::npos) {
        string tmp = data.substr(pos);
        if (tmp.size() == 1 && tmp[0] == '.') root = NULL;
        else root = new TreeNode(stoi(tmp));
        return;
      }

      string tmp = data.substr(pos, nextPos - pos);
      pos = nextPos + 1;
      if (tmp[0] == '.') root = NULL;
      else {
        root = new TreeNode(stoi(tmp));
        build(data, root -> left, pos);
        build(data, root -> right, pos);
      }
    }
};

// Your Codec object will be instantiated and called as such:
// Codec codec;
```

```
// codec.deserialize(codec.serialize(root));
```

- [x] LeetCode 1028.从先序遍历还原二叉树

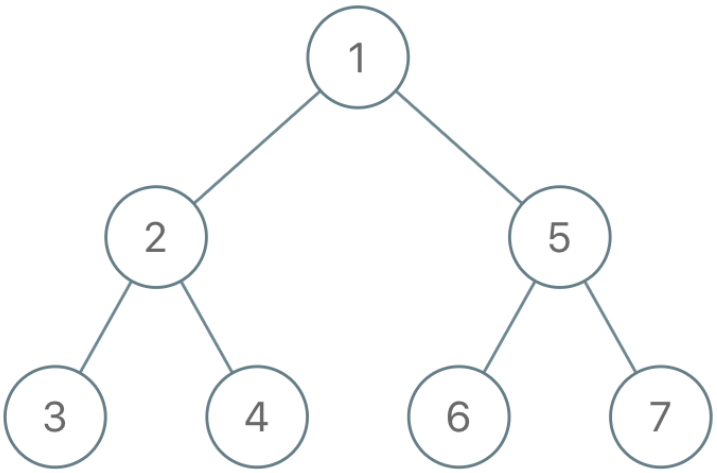
我们从二叉树的根节点 root 开始进行深度优先搜索。

在遍历中的每个节点处，我们输出 D 条短划线（其中 D 是该节点的深度），然后输出该节点的值。（如果节点的深度为 D，则其直接子节点的深度为 D + 1。根节点的深度为 0）。

如果节点只有一个子节点，那么保证该子节点为左子节点。

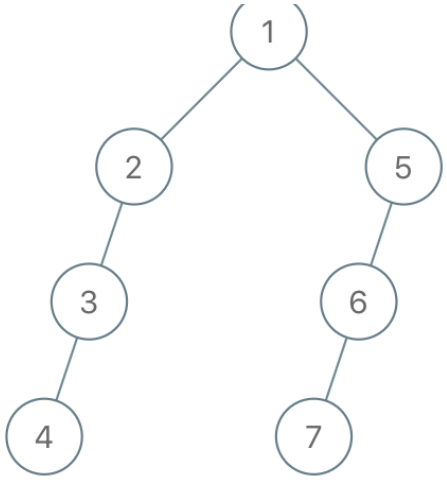
给出遍历输出 S，还原树并返回其根节点 root。

示例 1：



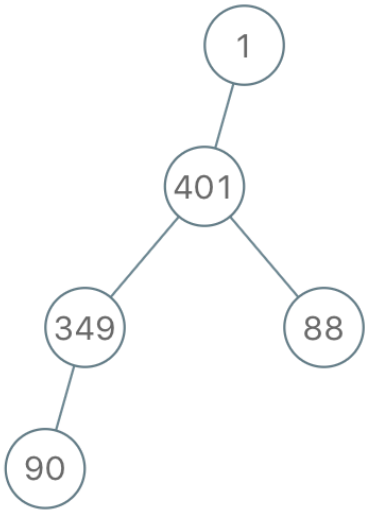

```
输入："1-2--3--4-5--6--7"
输出：[1,2,5,3,4,6,7]
```

示例 2：




```
输入："1-2--3---4-5--6---7"
输出：[1,2,5,3,null,6,null,4,null,7]
```

示例 3：




```
输入："1-401--349---90--88"
输出：[1,401,null,349,88,90]
```

构建二叉树一般采用递归的方法去构建，和LeetCode 297.二叉树的序列化与反序列化思路上有共同点，可以认为本题是二叉树序列化的一种方式，现在题目给出了序列化的结果，我们需要进行反序列化的操作。

用一个全局变量`pos`记录读取到字符串`s`中的位置，构建根节点时，用`end`指向`pos`

后面的第一个`-`字符，或者指向字符串末尾，构建完根节点，`pos`移动到根节点数字的最后一个数字，比如说根节点数字为`401`，最开始`pos`指向4，构建完根节点后`pos`指向`1`。

然后构建左右子树，用`cnt`去记录`-`字符的数量，参数`depth`代表上一层的深度，如果`cnt = depth + 1`，那么意味着一定存在左子节点（题目保证如果节点只有一个子节点，一定是左子节点），所以此时需要移动`pos`的位置，让其继续指向数字。如果`cnt != depth + 1`，那么意味着根节点的左子节点不存在。

然后构建右子节点，这时候`pos`的位置可能更新了，也可能并没有变化，所以仍然需要继续用`end`指向`pos`后面的第一个`-`字符，或者指向字符串末尾，继续用`cnt`去记录`-`的数量，完成右子树的构建。

只需要遍历依次字符串`s`即可完成构建，时间复杂度$O(n)$。

最开始写了一个版本，写完后发现构建左右子树的代码其实是一样的，可以进一步进行优化：

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
    int pos;
public:
    TreeNode* recoverFromPreorder(string s) {
        std::ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    cout.tie(NULL);
```

```
17
18          if (s.size() == 0) return NULL;
19
20          pos = 0;
21          TreeNode *root = build(s, 0);
22          return root;
23      }
24
25      TreeNode *build(const string & s, int depth)
26      {
27          int end = (s.find("-", pos) == string::npos) ?
    (int)s.size() : s.find("-", pos);
28          TreeNode *root = new TreeNode(stoi(s.substr(pos, end -
    pos)));
29          pos += end - pos - 1;
30          int cnt = 0;
31          for (int i = end; i < s.size(); ++i) {
32              if (s[i] == '-') ++cnt;
33              else break;
34          }
35
36          pos += (cnt == depth + 1) ? cnt + 1 : 0;
37          root -> left = (cnt == depth + 1) ? build(s, depth + 1)
    : NULL;
38
39          end = (s.find("-", pos) == string::npos) ?
    (int)s.size() : s.find("-", pos);
40          cnt = 0;
41          for (int i = end; i < s.size(); ++i) {
42              if (s[i] == '-') ++cnt;
43              else break;
44          }
45          pos += (cnt == depth + 1) ? cnt + 1 : 0;

46          root -> right = (cnt == depth + 1) ? build(s, depth +
```

```
    1) : NULL;

        return root;
    }
};
```

优化的递归版本，此时让`pos`每次指向数字后的第一个`-`，用`cnt`去记录`-`的个数，优先计算`-`，代表马上要建立的节点的深度，`depth`代表当前层的深度，如果`cnt != depth`，意味着接下来的数字并不属于这一层，所以直接返回即可。

如果相等，那么让`pos`移动`cnt`个位置，指向`-`后的第一个数字，然后用`end`指向数字后的`-`，提取出数字，构建根节点，此时`pos`移动到`end`的位置，于是`pos`又指向了数字后的第一个`-`，于是就可以递归的去构建左右子树了。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
    int pos;
public:
    TreeNode* recoverFromPreorder(string s) {
        std::ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    cout.tie(NULL);

        if (s.size() == 0) return NULL;


        pos = 0;
```

```
        TreeNode *root;
        build(s, root, 0);
        return root;
    }


    void build(const string & s, TreeNode *&root, int depth)
    {
      int cnt = 0;
      for (int i = pos; i < s.size(); ++i) {
        if (s[i] == '-') ++cnt;
            else break;
      }

      if (cnt != depth) { root = NULL; return; }

      pos += cnt;
    int end = (s.find("-", pos) == string::npos) ?
(int)s.size() : s.find("-", pos);
    root = new TreeNode(stoi(s.substr(pos, end - pos)));
      pos += end - pos;

      build(s, root -> left, depth + 1);
      build(s, root -> right, depth + 1);
    }
};
```

根据二叉树前序和后序遍历构造二叉树

之前结论已经指明，必须有中序遍历结果才能唯一确定一棵二叉树，现在回到它的逆问题，也就是给出前序和后序遍历结果，问能否构造出一棵这样的二叉树。

- [x] LeetCode 889.根据前序和后序遍历构造二叉树

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* constructFromPrePost(vector<int>& pre, vector<int>& post) {
        std::ios_base::sync_with_stdio(false);
      cin.tie(NULL);
      cout.tie(NULL);

        int n = pre.size();
        return build(0, n - 1, 0, n - 1, pre, post);
    }

    TreeNode *build(int pre_start, int pre_end, int post_start, int post_end, vector<int>& pre, vector<int>& post)
    {
        if (pre_start > pre_end || post_start > post_end) return NULL;

        TreeNode *root = new TreeNode(pre[pre_start]);
        TreeNode *l = NULL, *r = NULL;

        int pos = -1, leftSize = 0;
        if (pre_start < pre_end) {

            pos = find(post.begin(), post.end(), pre[pre_start
```

```
+ 1]) - post.begin();
31              leftSize = pos - post_start + 1;
32              l = build(pre_start + 1, pre_start + leftSize,
post_start, post_start + leftSize - 1, pre, post);
33          }
34
35          if (pre_start + leftSize < pre_end) {
36              r = build(pre_start + 1 + leftSize, pre_end,
post_start + leftSize, post_end - 1, pre, post);
37          }
38
39          root -> left = l;
40          root -> right = r;
41          return root;
42      }
43  };
```

以题目中的数据为例：

```
1  pre:  1 2 4 5 3 6 7
2  post: 4 5 2 6 7 3 1
3
4  根据前面利用中序和前序来构建二叉树的思路，我们希望得到如下结构
5  pre: [root][leftSize][rightSize]
6  post:[leftSize][rightSize][root]
7
8  于是思路是首先根据pre的第一个节点构建根节点，然后去判断leftSize是否不为0，
   如果不为0，那么就可以递归的去构建左子树。
9
10 在post里寻找leftSize的第一个元素，那么从post_start到这个元素之间就是左子
   树的元素
11 注意每次都要去检查leftSize和rightSize是否存在，不存在则节点为NULL
```

根据二叉树的前序遍历构造二叉搜索树

根据二叉树前序遍历构造二叉搜索树

- [x] LeetCode 1008.先序遍历构造二叉树

二叉树没有限定子节点和根节点元素的大小关系，但是二叉搜索树因为限定了大小关系，所以仅仅给出其一个遍历结果，也能唯一确定的构造一个二叉树，只不过是二叉搜索树。

比如题目里的`[8,5,1,7,10,12]`，会发现10，12肯定在右子树，那么就很类似将链表转为二叉树的操作，找到第一个大于起始节点的值，然后递归的去实现，只需要注意一下边界条件即可。

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* bstFromPreorder(vector<int>& preorder) {
        std::ios_base::sync_with_stdio(false);
        cin.tie(NULL);
        cout.tie(NULL);

        int n = preorder.size();
        if (!n) return NULL;
        return build(preorder, 0, n - 1);
    }

    TreeNode *build(vector<int> & preorder, int start, int end)
    {
        TreeNode *root = new TreeNode(preorder[start]);
```

```
25          int pos = start;
26          while (pos <= end) {
27              if (preorder[pos] > preorder[start]) break;
28              else ++pos;
29          }
30
31          TreeNode *left = NULL, *right = NULL;
32          if (pos - 1 - start > 0) left = build(preorder, start +
    1, pos - 1);
33          if (end - pos + 1 > 0) right = build(preorder, pos,
    end);
34          root -> left = left;
35          root -> right = right;
36
37          return root;
38      }
39  };
```