数据结构-树

Frank Jagger
Programmer | Lifelong Learner

树的基本操作

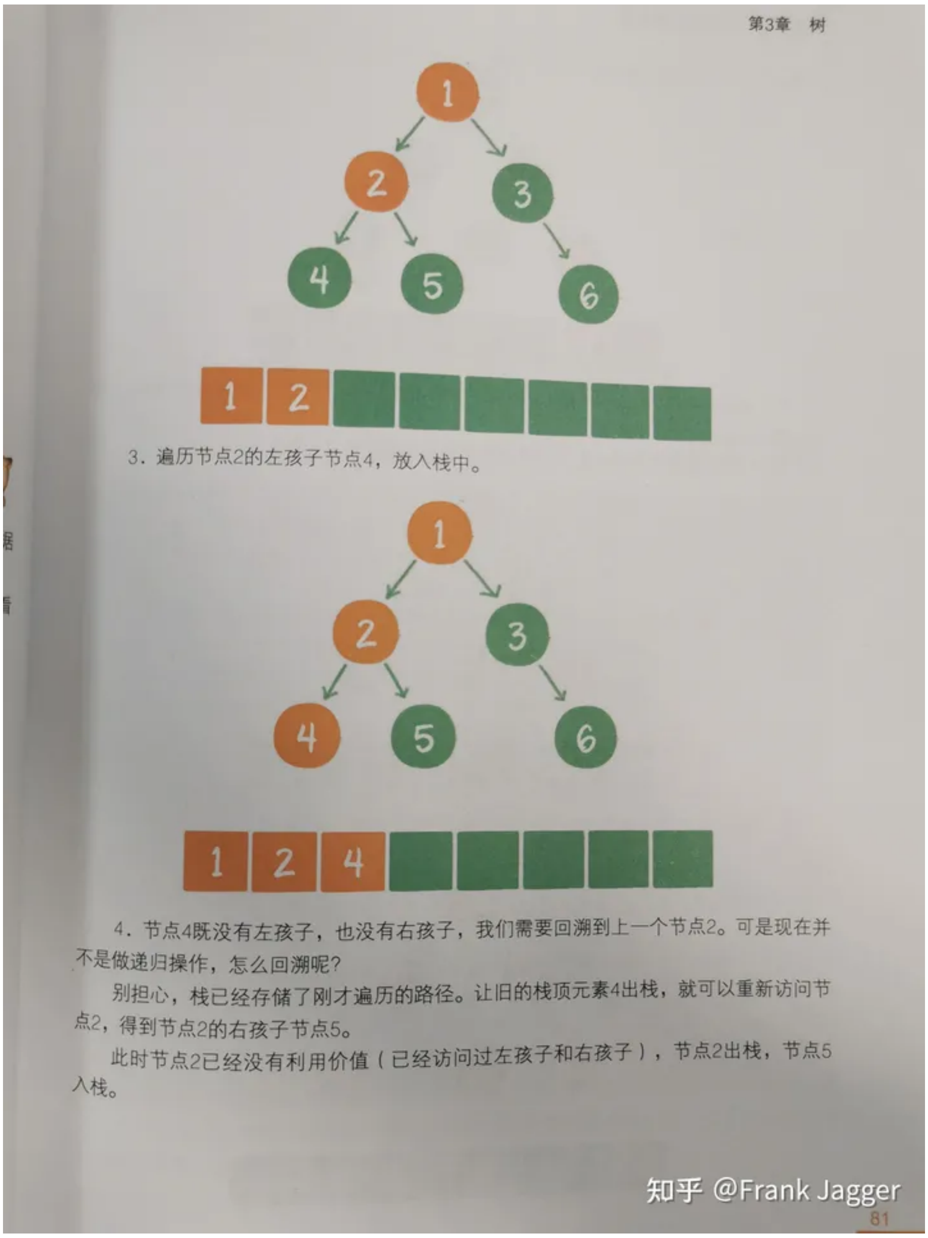
```
struct node
{
    int data;
    node* lchild;
    node* rchild;
};

void preorder(node* root) {
    if(root == NULL)
        return;
    printf(" %d", root->data);
    preorder(root->lchild);
    preorder(root->rchild);
}
void inorder(node* root) {
    if(root == NULL)
        return;
    inorder(root->lchild);
    printf(" %d", root->data);
    inorder(root->rchild);
}
void postorder(node* root) {
    if(root == NULL)
        return;
    postorder(root->lchild);
    postorder(root->rchild);
    printf(" %d", root->data);
}
void layerorder(node* root, vector<int> &layer) {
    if(root == NULL) return;
    queue<node*> q;
    q.push(root);
    while(!q.empty()) {
        node* now = q.front(); // 取队首元素
        q.pop();
        layer.push_back(now->data); // 访问队首元素 or "cout << now->data;"
        if(now->lchild) q.push(now->lchild);
        if(now->rchild) q.push(now->rchild);
    }
}

// 层序遍历递归实现
class Solution {
public:
    void order(node* cur, vector<vector<int>>& result, int depth)
    {
        if (cur == nullptr) return;
        if (result.size() == depth) result.push_back(vector<int>());
        result[depth].push_back(cur->val);
        order(cur->left, result, depth + 1);
        order(cur->right, result, depth + 1);
    }
    vector<vector<int>> levelOrder(node* root) {
        vector<vector<int>> result;
        int depth = 0;
        order(root, result, depth);
```

```
    }
};
```

借助栈实现非递归遍历（以二叉树的前序遍历为例）：



第3章 树

3．遍历节点2的左孩子节点4，放入栈中。

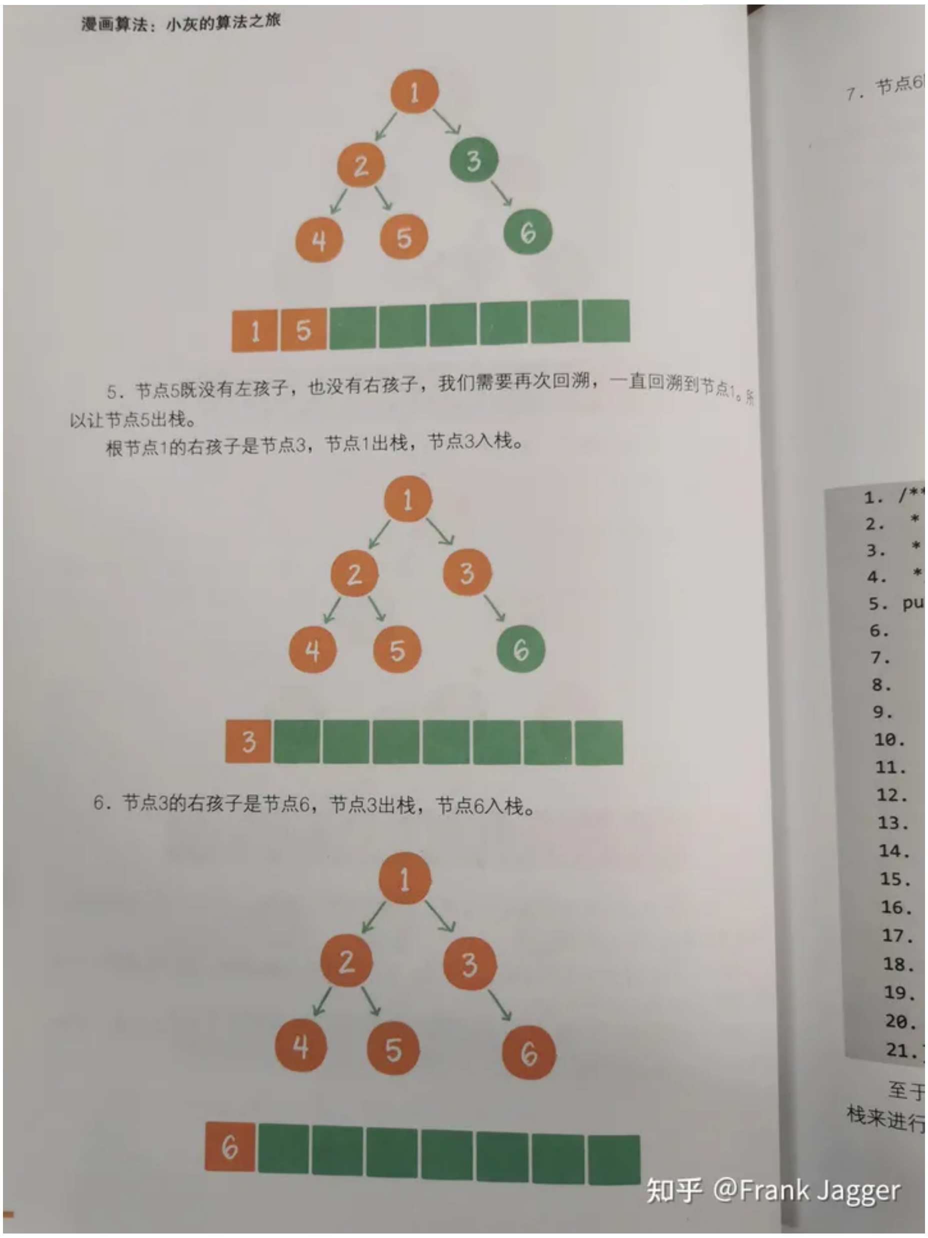
4．节点4既没有左孩子，也没有右孩子，我们需要回溯到上一个节点2。可是现在并不是做递归操作，怎么回溯呢？

别担心，栈已经存储了刚才遍历的路径。让旧的栈顶元素4出栈，就可以重新访问节点2，得到节点2的右孩子节点5。

此时节点2已经没有利用价值（已经访问过左孩子和右孩子），节点2出栈，节点5入栈。

81

来源：《漫画算法：小灰的算法之旅》



来源：《漫画算法：小灰的算法之旅》

```
void preorderWithStack(node* root) {
    stack<node*> s;
    vector<int> res;
    if (root == NULL) return res;
    s.push(root);
    while(!s.empty()) {
        node* now = s.top();
        s.pop();
        res.push_back(now->data);
        if (now->right) s.push(now->right); // 空节点不入栈，先右后左（因为栈 LIFO）
        if (now->left) s.push(now->left); // 空节点不入栈
    }
    return res;
}

// preorder 是中-左-右，需要调整左右的入栈顺序，得到中-右-左，再反转 res 数组即可得到 postord
void postorderWithStack(node* root) {
    stack<node*> s;
    vector<int> res;
    if (root == NULL) return res;
    s.push(root);
    while(!s.empty()) {
        node* now = s.top();
```

```
        if (now->left) s.push(now->left); // 空节点不入栈，先左后右
        if (now->right) s.push(now->right); // 空节点不入栈
    }
    reverse(res.begin(), res.end()); // 反转 res 数组即得到左-右-中顺序
    return res;
}

/*

Another writing

*/

void preorderWithStack(node* root) {
    stack<node*> s;
    node* p = root;
    while(p != NULL || !s.empty()) {
        while(p != NULL) {
            cout << p->data << " ";
            s.push(p);
            p = p->lchild;
        }
        if(!s.empty()) {
            p = s.top();
            s.pop();
            p = p->rchild;
        }
    }
}

void inorderWithStack(node* root) {
    stack<node*> s;
    node* p = root;
    while(p != NULL || !s.empty()) {
        while(p != NULL) {
            s.push(p);
            p = p->lchild;
        }
        if(!s.empty()) {
            p = s.top();
            s.pop();
            cout << p->data << " ";
            p = p->rchild;
        }
    }
}

// 要保证根结点在左孩子和右孩子访问之后才能访问，因此对于任一结点P，先将其入栈。
// 如果P不存在左孩子和右孩子，则可以直接访问它；
// 如果P存在左孩子或者右孩子，但其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。
// 若非上述两种情况，则将P的右孩子和左孩子依次入栈
void postorderWithStack(node* root) {
    stack<node*> s;
    node* cur;
    node* pre = NULL;
    s.push(root);
    while(!s.empty()) {
        cur = s.top();
        if((!cur->lchild && !cur->rchild) || (pre && (pre==cur->lchild || pre==cur->rc
            cout << cur->data << " ";
            s.pop();
            pre = cur;
        } else {
            if(cur->rchild) s.push(cur->rchild);
            if(cur->lchild) s.push(cur->lchild);
        }
```

```
    }
}
```

Morris中序遍历伪代码

```
1. Initialize current as root
2. While current is not NULL
   If current does not have left child
      a) Print current’s data
      b) Go to the right, i.e., current = current->right
   Else
      a) Make current as right child of the rightmost node in current's left subtree
      b) Go to this left child, i.e., current = current->left
```

Morris中序遍历C++实现

```
// 不论递归还是非递归，其额外的空间复杂度都为O(h)即O(logn)
// 一种不借助额外空间的方法来实现树的遍历：线索二叉树
void inorderMorris(node* root) {
    node* cur = root;
    while(cur != NULL) {
        if(cur->lchild == NULL) {
            cout << cur->data << " ";
            cur = cur->rchild;
        } else {
            node* prev = cur->lchild;
            while(prev->rchild != NULL && prev->rchild != cur) {
                prev = prev->rchild;
            }
            if(prev->rchild == NULL) {
                prev->rchild = cur;
                cur = cur->lchild;
            } else if(prev->rchild == cur) {
                cout << cur->data;
                cur = cur->rchild;
                prev->rchild = NULL;
            }
        }
    }
}
```

根据先序遍历和中序遍历构建树

```
node* create(int preL, int preR, int* pre, int inL, int inR, int* in) {
    if(preL > preR) {
        return NULL; //先序序列长度小于等于0，递归边界
    }
    node* root = new node; //申请一个node型变量的地址空间
    root->data = pre[preL];
    int k;
    for(k = inL; k <= inR; k++) {
        if(in[k] == pre[preL])
            break;
    }
    int numLeft = k - inL; //左子树的结点个数
    //左子树的先序区间为[preL + 1, preL + numLeft]，中序区间为[inL, k - 1]
    root->lchild = create(preL + 1, preL + numLeft, pre, inL, k - 1, in);
    //右子树的先序区间为[preL + numLeft + 1, preR]，中序区间为[k + 1, inR]
    root->rchild = create(preL + numLeft + 1, preR, pre, k + 1, inR, in);

    return root;
}
```

根据后序遍历和中序遍历构建树

```
node* create(int postL, int postR, int* post, int inL, int inR, int* in) {
    if(postL > postR) {
        return NULL; //后序序列长度小于等于0，递归边界
    }
    node* root = new node; //申请一个node型变量的地址空间
    root->data = post[postR];
    int k;
    for(k = inL; k <= inR; k++) {
        if(in[k] == post[postR])
            break;
    }
    int numLeft = k - inL; //左子树的结点个数
    //左子树的后序区间为[postL, postL + numLeft - 1]，中序区间为[inL, k - 1]
    root->lchild = create(postL, postL + numLeft - 1, post, inL, k - 1, in);
    //右子树的后序区间为[postL + numLeft, postR - 1]，中序区间为[k + 1, inR]
    root->rchild = create(postL + numLeft, postR - 1, post, k + 1, inR, in);

    return root;
}
```

n叉树的DFS遍历和BFS遍历

```
struct node
{
    int data;
    vector<node*> children; //结点个数不固定
    node* parent = NULL;
};

void dfs(node* root, int level) {
    num_of_nodes[level]++; // 记录每层的结点个数
    if(root->children.empty()) { // 到达叶结点（递归边界）
        return;
    }
    for(vector<node*>::iterator it = root->children.begin(); it != root->children.end(
        dfs(*it, level + 1);
    }
}
void bfs(node* root) {
    queue<node*> q;
    q.push(root);
    while(!q.empty()) {
        node* now = q.front(); //取队首元素
        q.pop();
        printf("%d ", now->data); //访问队首元素
        for(vector<node*>::iterator it = root->children.begin(); it != root->children.
            q.push(*it);
        }
    }
}
```

BFS层序遍历记录每层结点个数

```
void bfs(node* root) {
    if(root == NULL) return;
    queue<node*> q;
    q.push(root);
    int level = 1; // 记录层数
    while(!q.empty()) {
        int size = q.size(); // 获取当前层的叶结点个数
```

```
                q.pop();
                if(now->left) q.push(now->left);
                if(now->right) q.push(now->right);
            }
            level++;
        }
    }
```

二叉搜索树的最近公共祖先 (LCA)

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    // 如果两个节点值都小于根节点，说明他们都在根节点的左子树上，我们往左子树上找
    // 如果两个节点值都大于根节点，说明他们都在根节点的右子树上，我们往右子树上找
    // 如果一个节点值大于根节点，一个节点值小于根节点，说明他们他们一个在根节点的左子树上一个在朴
    while (1) {
        if (p->val < root->val && q->val < root->val) root = root->left;
        else if (p->val > root->val && q->val > root->val) root = root->right;
        else break;
    }
    // while ((p->val - root->val) * (q->val - root->val) > 0)
    //     root = p->val < root->val ? root->left : root->right;
    return root;
}
```

构建哈夫曼树及哈夫曼编码

```
#include <iostream>
#include <deque>
#include <string>
#include <algorithm>

using namespace std;
struct node
{
    char key;
    double data; // 出现频率，即权重
    node* lchild;
    node* rchild;
};
deque<node*> forest;
bool cmp(node* a, node* b) { // 升序排列
    return a->data < b->data;
}
void preorder(node* root) {
    if(root == NULL) return;
    printf("%.2lf ", root->data);
    preorder(root->lchild);
    preorder(root->rchild);
}
void inorder(node* root) {
    if(root == NULL) return;
    inorder(root->lchild);
    printf("%.2lf ", root->data);
    inorder(root->rchild);
}
void printCode(node* root, string str) { // 哈夫曼编码
    if(root == NULL) return;
    if(root->lchild) str += '0';
    printCode(root->lchild, str);
    if(!root->lchild && !root->rchild) { // 叶子结点
        printf("%c's code is: ", root->key);
        cout << str << endl;
    }
```

```
        if(root->rchild) str += '1';
        printCode(root->rchild, str);
    }
    int main()
    {
        int n = 5; // num of the leaves
        char dict[n] = {'A', 'B', 'C', 'D', 'E'};
        double weight[n] = {0.33, 0.27, 0.2, 0.13, 0.07};
        for(int i = 0; i < n; i++) {
            node* ptr = new node;
            ptr->key = dict[i];
            ptr->data = weight[i];
            ptr->lchild = NULL;
            ptr->rchild = NULL;
            forest.push_back(ptr); // 形成森林，森林中的每一棵樹都是一個節點
        }
        for(int i = 0; i < n - 1; i++) { // 從森林構建霍夫曼樹(n棵树合并n-1次成为1棵树)
            sort(forest.begin(), forest.end(), cmp);
            node* ptr = new node;
            ptr->data = forest[0]->data + forest[1]->data;
            ptr->lchild = forest[0];
            ptr->rchild = forest[1];
            forest.pop_front();
            forest.pop_front();
            forest.push_back(ptr);
        }
        // preorder(forest.front());
        // printf("\n");
        // inorder(forest.front());
        string huffmancode = "";
        printCode(forest.front(), huffmancode);
        return 0;
    }
```

You may also enjoy:

Frank Jagger：数据结构-树

Frank Jagger：[PAT]A1064完全二叉搜索树

Frank Jagger：[PAT]A1066平衡二叉树的根

Frank Jagger：[PAT]A1110 Complete Binary Tree

Frank Jagger：判断一棵二叉树是否是二叉查找树(BST)

Thanks:

Inorder Tree Traversal without recursion and without stack! - GeeksforGeeks
www.geeksforgeeks.org/inorder-tree-trav...

木鸟杂记：数据结构与算法（一）：二叉树的非递归遍历
25 赞同 · 2 评论 文章

二叉树的非递归遍历 - Matrix海子 - 博客园
www.cnblogs.com/dolphin0520/archive/2011/08/25/2...

编辑于 2023-03-14 19:33・IP 属地天津

评论千万条，友善第一条

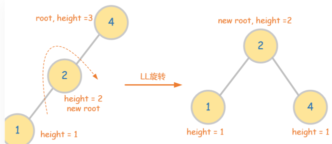
还没有评论，发表第一个评论吧

推荐阅读

数据结构——树

一、树的概念：树其实是区别于线性表，线性结构的另一种数据结构，本质是结点的有限集。树的定义有两点： 1)有且仅有一个特定的称为根（root）的结点；2）当结点数量>1时，其余结点可分...

David任耀坤



数据结构之：树

丫丫ust