【第二十一课】手撕红黑树

red_black_tree

```
/*************************************************************************
  > File Name: 1.red_black_tree.cpp
  > Author: huguang
  > Mail: hug@haizeix.com
  > Created Time:
 ************************************************************************/

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <queue>
#include <stack>
#include <algorithm>
#include <string>
#include <map>
#include <set>
#include <vector>
using namespace std;

#define NIL &(node::__NIL)
struct node {
    node(int key = 0, int color = 0, node *lchild = NIL, node *rchild = NIL)
    : key(key), color(color), lchild(lchild), rchild(rchild) {}
    int key;
    int color; // 0 red, 1 black, 2 double black
    node *lchild, *rchild;
    static node __NIL;
};

node node::__NIL(0, 1);

node *getNewNode(int key) {
    return new node(key);
}

bool has_red_child(node *root) {
    return root->lchild->color == 0 || root->rchild->color == 0;
}

node *left_rotate(node *root) {
    node *temp = root->rchild;
    root->rchild = temp->lchild;
    temp->lchild = root;
    return temp;
}

node *right_rotate(node *root) {
    node *temp = root->lchild;
    root->lchild = temp->rchild;
    temp->rchild = root;
    return temp;
}
```

```
node *insert_maintain(node *root) {
    int flag = 0;
    if (root->lchild->color == 0 && has_red_child(root->lchild)) flag = 1;
    if (root->rchild->color == 0 && has_red_child(root->rchild)) flag = 2;
    if (flag == 0) return root;
    if (root->lchild->color == 0 && root->rchild->color == 0) {
        root->color = 0;
        root->lchild->color = root->rchild->color = 1;
        return root;
    }
    if (flag == 1) {
        if (root->lchild->rchild->color == 0) {
            root->lchild = left_rotate(root->lchild);
        }
        root = right_rotate(root);
    } else {
        if (root->rchild->lchild->color == 0) {
            root->rchild = right_rotate(root->rchild);
        }
        root = left_rotate(root);
    }
    root->color = 0;
    root->lchild->color = root->rchild->color = 1;
    return root;
}

node *__insert(node *root, int key) {
    if (root == NIL) return getNewNode(key);
    if (key == root->key) return root;
    if (key < root->key) {
        root->lchild = __insert(root->lchild, key);
    } else {
        root->rchild = __insert(root->rchild, key);
    }
    return insert_maintain(root);
}

node *insert(node *root, int key) {
    root = __insert(root, key);
    root->color = 1;
    return root;
}

void clear(node *root) {
    if (root == NIL) return ;
    clear(root->lchild);
    clear(root->rchild);
    delete root;
    return ;
}

void print(node *root) {
    printf("( %d | %d, %d, %d )\n",
        root->color, root->key,
        root->lchild->key, root->rchild->key
    );
    return ;
}
```

```
void output(node *root) {
    if (root == NIL) return ;
    print(root);
    output(root->lchild);
    output(root->rchild);
    return ;
}

int main() {
    int val;
    node *root = NIL;
    while (cin >> val) {
        root = insert(root, val);
        cout << endl << "==== rbtree print ====" << endl;
        output(root);
        cout << "==== rbtree print done ====" << endl;
    }
    return 0;
}
```

981. 基于时间的键值存储

设计一个基于时间的键值数据结构，该结构可以在不同时间戳存储对应同一个键的多个值，并针对特定时间戳检索键对应的值。

实现 `TimeMap` 类：

- `TimeMap()` 初始化数据结构对象
- `void set(String key, String value, int timestamp)` 存储键 `key`、值 `value`，以及给定的时间戳 `timestamp`。
- `String get(String key, int timestamp)`
  - 返回先前调用 `set(key, value, timestamp_prev)` 所存储的值，其中 `timestamp_prev <= timestamp` 。
  - 如果有多个这样的值，则返回对应最大的 `timestamp_prev` 的那个值。
  - 如果没有值，则返回空字符串（`""`）。

示例：

```
输入：
["TimeMap", "set", "get", "get", "set", "get", "get"]
[[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo",
 5]]
输出：
[null, null, "bar", "bar", null, "bar2", "bar2"]

解释：
TimeMap timeMap = new TimeMap();
timeMap.set("foo", "bar", 1);  // 存储键 "foo" 和值 "bar" ，时间戳 timestamp = 1
timeMap.get("foo", 1);         // 返回 "bar"
timeMap.get("foo", 3);         // 返回 "bar", 因为在时间戳 3 和时间戳 2 处没有对应 "foo" 的值，所
```

```
以唯一的值位于时间戳 1 处（即 "bar"） 。
timeMap.set("foo", "bar2", 4); // 存储键 "foo" 和值 "bar2" ，时间戳 timestamp = 4
timeMap.get("foo", 4);         // 返回 "bar2"
timeMap.get("foo", 5);         // 返回 "bar2"
```

```
class TimeMap {
public:
    /** Initialize your data structure here. */
    TimeMap() {}
    unordered_map<string, map<int, string> > h;
    void set(string key, string value, int timestamp) {
        h[key][timestamp] = value;
        return ;
    }

    string get(string key, int timestamp) {
        if (h.find(key) == h.end()) return "";
        if (h[key].find(timestamp) != h[key].end()) return h[key][timestamp];
        h[key].insert(pair<int, string>(timestamp, ""));
        auto iter = h[key].find(timestamp);
        string ret = (--iter)->second;
        h[key].erase(h[key].find(timestamp));
        return ret;
    }
};

/**
 * Your TimeMap object will be instantiated and called as such:
 * TimeMap* obj = new TimeMap();
 * obj->set(key,value,timestamp);
 * string param_2 = obj->get(key,timestamp);
 */
```

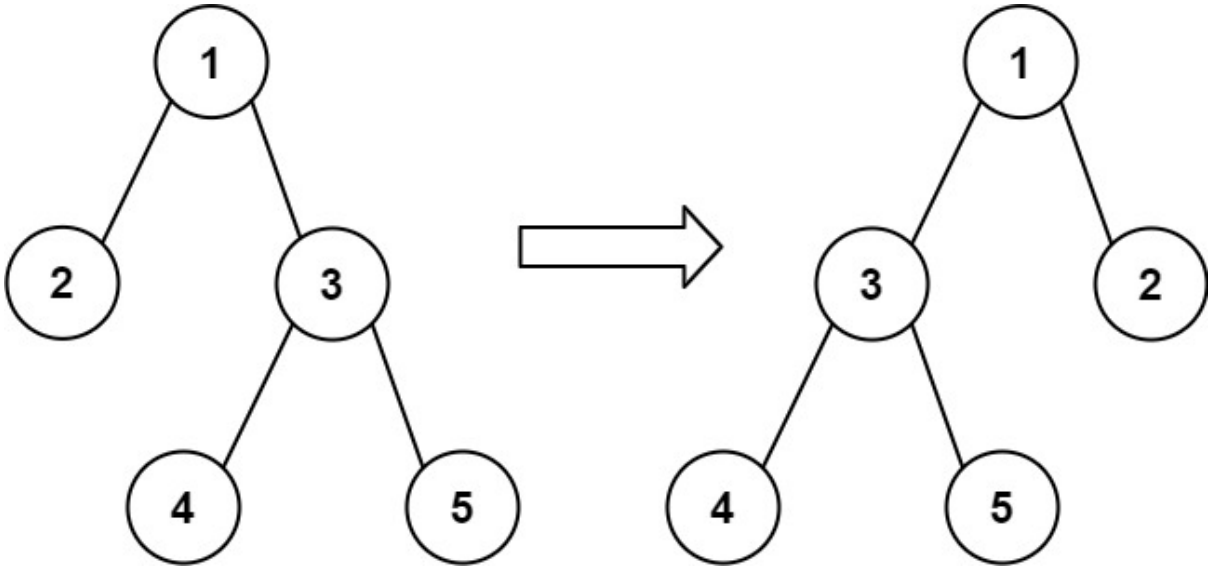
971. 翻转二叉树以匹配先序遍历

难度中等70

给你一棵二叉树的根节点 `root` ，树中有 `n` 个节点，每个节点都有一个不同于其他节点且处于 `1` 到 `n` 之间的值。

另给你一个由 `n` 个值组成的行程序列 `voyage` ，表示 **预期** 的二叉树 **先序遍历** 结果。

通过交换节点的左右子树，可以 **翻转** 该二叉树中的任意节点。例，翻转节点 1 的效果如下：



请翻转 **最少** 的树中节点，使二叉树的 **先序遍历** 与预期的遍历行程 `voyage` **相匹配** 。

如果可以，则返回 **翻转的** 所有节点的值的列表。你可以按任何顺序返回答案。如果不能，则返回列表 `[-1]`。

示例 1：


```
输入：root = [1,2], voyage = [2,1]
输出：[-1]
```

解释：翻转节点无法令先序遍历匹配预期行程。

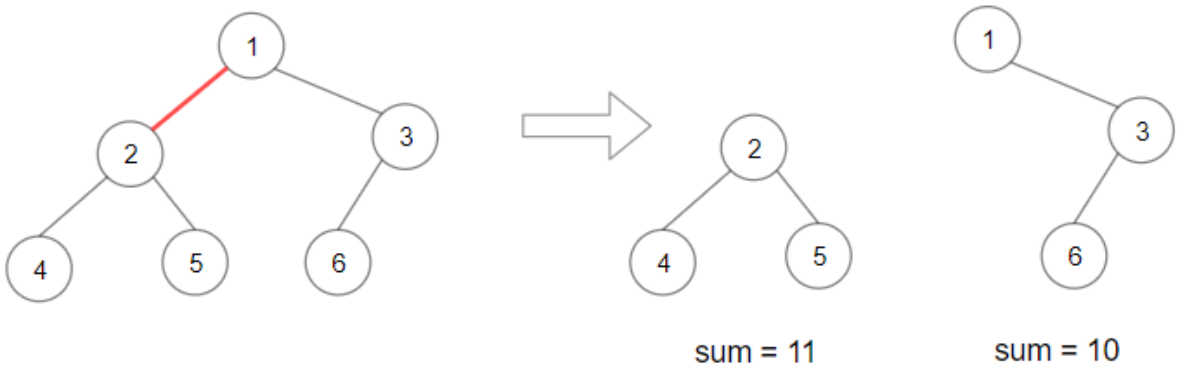
```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int ind;
    bool preorder(TreeNode *root, vector<int> &voyage, vector<int> &ret) {
        if (root == nullptr) return true;
        if (voyage[ind] != root->val) {
            ret.clear();
            ret.push_back(-1);
            return false;
        }
        ind += 1;
        if (ind + 1 == voyage.size()) return true;
        if (root->left && root->left->val != voyage[ind]) {
            swap(root->left, root->right);
            ret.push_back(root->val);
        }
        if (!preorder(root->left, voyage, ret)) return false;
        if (!preorder(root->right, voyage, ret)) return false;
        return true;
    }
    vector<int> flipMatchVoyage(TreeNode* root, vector<int>& voyage) {
        vector<int> ret;
        ind = 0;
        preorder(root, voyage, ret);
        return ret;
    }
};
```

1339. 分裂二叉树的最大乘积

给你一棵二叉树，它的根为 `root` 。请你删除 1 条边，使二叉树分裂成两棵子树，且它们子树和的乘积尽可能大。

由于答案可能会很大，请你将结果对 10^9 + 7 取模后再返回。

示例 1：




```
输入：root = [1,2,3,4,5,6]
输出：110
解释：删除红色的边，得到 2 棵子树，和分别为 11 和 10 。它们的乘积是 110 （11*10）
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int avg, ans = 0;
    int getTotal(TreeNode *root) {
        if (root == nullptr) return 0;
        int val = root->val + getTotal(root->left) + getTotal(root->right);
        if (abs(val - avg) < abs(ans - avg)) ans = val;
        return val;
    }
    int maxProduct(TreeNode* root) {
        int total = getTotal(root);
        avg = total / 2;
        ans = total;
        getTotal(root);
        return (long long)(ans) * (total - ans) % (long long)(1e9+7);
    }
};
```

449. 序列化和反序列化二叉搜索树

序列化是将数据结构或对象转换为一系列位的过程，以便它可以存储在文件或内存缓冲区中，或通过网络连接链路传输，以便稍后在同一个或另一个计算机环境中重建。

设计一个算法来序列化和反序列化 **二叉搜索树** 。 对序列化/反序列化算法的工作方式没有限制。 您只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。

编码的字符串应尽可能紧凑。

示例 1：

```
输入：root = [2,1,3]
输出：[2,1,3]
```

```
/**
 * 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) {
        if (root == nullptr) return "";
        string ret = "";
        stringstream ss;
        ss << root->val;
        ss >> ret;
        if (root->left == nullptr && root->right == nullptr) return ret;
        ret += "(";
        ret += serialize(root->left);
        if (root->right) {
            ret += ",";
            ret += serialize(root->right);
        }
        ret += ")";
        return ret;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        int scode = 0, ind = 0, k = 0;
        stack<TreeNode *> s;
        TreeNode *p, *root = nullptr;
        while (ind < data.size()) {
            switch (scode) {
                case 0: {
                    if (data[ind] <= '9' && data[ind] >= '0') scode = 1;
                    else if (data[ind] == '(') scode = 2;
                    else if (data[ind] == ',') scode = 3;
                    else if (data[ind] == ')') scode = 4;
                } break;
                case 1: {
```

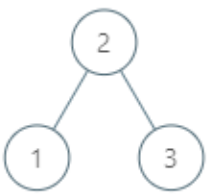
```
                    int num = 0;
                    while (ind < data.size() && data[ind] <= '9' && data[ind] >= '0') {
                        num = num * 10 + (data[ind] - '0');
                        ind += 1;
                    }
                    p = new TreeNode(num);
                    if (root == nullptr) root = p;
                    if (k == 1) s.top()->left = p;
                    else if (k == 2) s.top()->right = p;
                    scode = 0;
                } break;
                case 2: {
                    s.push(p);
                    ind += 1;
                    k = 1;
                    scode = 0;
                } break;
                case 3: {
                    k = 2;
                    ind += 1;
                    scode = 0;
                } break;
                case 4: {
                    s.pop();
                    ind += 1;
                    scode = 0;
                } break;
            }
        }
        return root;
    }
};

// Your Codec object will be instantiated and called as such:
// Codec* ser = new Codec();
// Codec* deser = new Codec();
// string tree = ser->serialize(root);
// TreeNode* ans = deser->deserialize(tree);
// return ans;
```

剑指 Offer II 053. 二叉搜索树中的中序后继

给定一棵二叉搜索树和其中的一个节点 `p` ，找到该节点在树中的中序后继。如果节点没有中序后继，请返回 `null` 。

节点 `p` 的后继是值比 `p.val` 大的节点中键值最小的节点，即按中序遍历的顺序节点 `p` 的下一个节点。

示例 1：



```
输入：root = [2,1,3], p = 1
输出：2
解释：这里 1 的中序后继是 2。请注意 p 和返回值都应是 TreeNode 类型。
```

```
/**
 * 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 *pre, *ans;
    bool inorder(TreeNode *root, TreeNode *p) {
        if (root == nullptr) return false;
        if (inorder(root->left, p)) return true;
        if (pre == p) {
            ans = root;
            return true;
        }
        pre = root;
        if (inorder(root->right, p)) return true;
        return false;
    }
    TreeNode* inorderSuccessor(TreeNode* root, TreeNode* p) {
        pre = ans = nullptr;
        inorder(root, p);
        return ans;
    }
};
```

117. 填充每个节点的下一个右侧节点指针 II

给定一个二叉树

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 `NULL`。

初始状态下，所有 next 指针都被设置为 `NULL`。

进阶：

- 你只能使用常量级额外空间。
- 使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

示例：

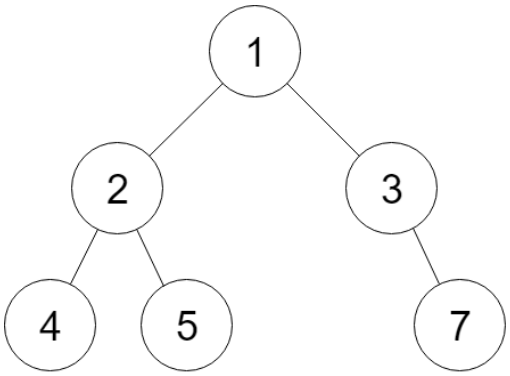


Figure A

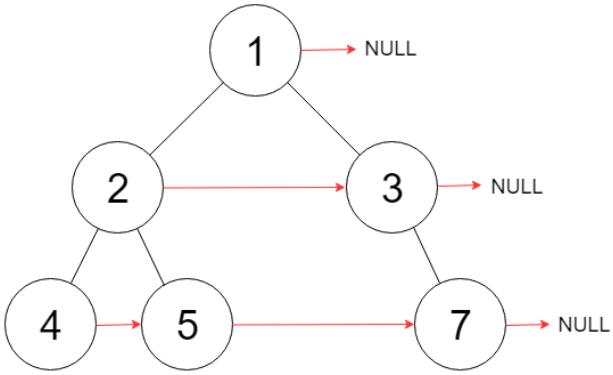


Figure B

```
输入：root = [1,2,3,4,5,null,7]
输出：[1,#,2,3,#,4,5,7,#]
解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。序列化输出按层序遍历顺序（由 next 指针连接），'#' 表示每层的末尾。
```

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}

    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}

    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};
*/

class Solution {
public:
    Node *layer_connect(Node *head) {
        Node *p = head, *pre = nullptr, *new_head = nullptr;
        while (p) {
            if (p->left) {
                if (pre) pre->next = p->left;
                pre = p->left;
            }
            if (new_head == nullptr) new_head = pre;
            if (p->right) {
                if (pre) pre->next = p->right;
```

```
                pre = p->right;
            }
            if (new_head == nullptr) new_head = pre;
            p = p->next;
        }
        return new_head;
    }
    Node* connect(Node* root) {
        Node *p = root;
        while (p = layer_connect(p)) ;
        return root;
    }
};
```

78. 子集

给你一个整数数组 `nums` ，数组中的元素 **互不相同** 。返回该数组所有可能的子集（幂集）。

解集 **不能** 包含重复的子集。你可以按 **任意顺序** 返回解集。

示例 1：

```
输入：nums = [1,2,3]
输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {
        int n = nums.size();
        unordered_map<int, int> mark;
        for (int i = 0, j = 1; i < 10; i++, j <<= 1) mark[j] = i;
        vector<vector<int>> ret;
        for (int i = 0, I = (1 << n); i < I; i++) {
            vector<int> arr;
            int val = i;
            while (val) {
                arr.push_back(nums[mark[val & (-val)]]);
                val &= (val - 1);
            }
            ret.push_back(arr);
        }
        return ret;
    }
};
```

220. 存在重复元素 III

给你一个整数数组 `nums` 和两个整数 `k` 和 `t` 。请你判断是否存在 **两个不同下标** `i` 和 `j`，使得 `abs(nums[i] - nums[j]) <= t` ，同时又满足 `abs(i - j) <= k` 。

如果存在则返回 `true`，不存在返回 `false`。

示例 1：

```
输入：nums = [1,2,3,1], k= 3, t = 0
输出：true
```

```
class Solution {
public:
    void delNum(map<long long, int> &h, long long x) {
        h[x] -= 1;
        if (h[x] == 0) h.erase(h.find(x));
        return ;
    }
    bool containsNearbyAlmostDuplicate(vector<int>& nums, int k, int t) {
        map<long long, int> h;
        for (int i = 0; i < nums.size(); i++) {
            if (i > k) {
                delNum(h, nums[i - k - 1]);
            }
            h[(long long)(nums[i]) - t - 1] += 1;
            h[(long long)(nums[i]) + t + 1] += 1;
            auto iter = h.find((long long)(nums[i]) - t - 1);
            iter++;
            if (iter->first != (long long)(nums[i]) + t + 1) return true;
            delNum(h, (long long)(nums[i]) - t - 1);
            delNum(h, (long long)(nums[i]) + t + 1);
            h[nums[i]] += 1;
        }
        return false;
    }
};
```

47. 全排列 II

给定一个可包含重复数字的序列 `nums` ，**按任意顺序** 返回所有不重复的全排列。

示例 1：

```
输入：nums = [1,1,2]
输出：
[[1,1,2],
 [1,2,1],
 [2,1,1]]
```

```
class Solution {
public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> ret;
        do {
            ret.push_back(nums);
        } while (next_permutation(nums.begin(), nums.end()));
```

```
        return ret;
    }
};
```

41. 缺失的第一个正数

给你一个未排序的整数数组 `nums` ，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为

```
O(n)
```

并且只使用常数级别额外空间的解决方案。

示例 1：

```
输入：nums = [1,2,0]
输出：3
```

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        for (int i = 0; i < nums.size(); i++) {
            while (nums[i] != i + 1) {
                if (nums[i] <= 0 || nums[i] > nums.size()) break;
                int ind = nums[i] - 1;
                if (nums[i] == nums[ind]) break;
                swap(nums[i], nums[ind]);
            }
        }
        int ind = 0;
        while (ind < nums.size() && nums[ind] == ind + 1) ++ind;
        return ind + 1;
    }
};
```