数据结构 第二次作业报告

2022013014 黄泽文 未央-软件21

1. 范围查询

分析

先对数组进行排序，然后实现 `lower_bound` 与 `upper_bound` 函数，对于任意一个数字 n，返回数组中小于等于 n 的最大的数与大于等于 n 的最小的数。则题目所要的结果为 $ub(r)-lb(l)+1$。注意判断数组长度为 0 的特殊情况。

这可以通过二分的思想实现。以 `lower_bound(l, r, n)` 为例，设 $mid=[\frac{l+r}{2}]$，若 $a_{mid}<n$，则将 l 更新为 $mid+1$，否则将 r 更新为 mid，不断循环直至 $l\geq r$，此时 l 即为结果。

代码

`lower_bound` 函数的代码如下。

```
int *lower_bound(int *first, int *last, const int &value) {
      int *it;
      int count;
      count = last - first;
      while (count > 0) {
            it = first + count / 2;
            if (*it < value) {
                  first = ++it;
            } else {
                  last = it;
            }
            count = last - first;
      }
      return first;
}
```

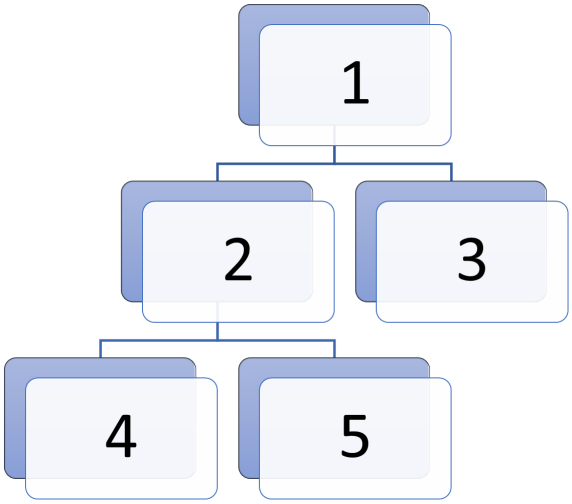
2. 真二叉树重构

分析

不妨通过样例输入理解建树的过程。

```
5
1 2 4 5 3
4 5 2 3 1
```

这个样例所代表的树形如：



我们通过递归建树，这个过程可以概括为：

第一步，获取当前根节点。先序遍历的第一个节点应该等于后序遍历的最后一个节点，即节点 1。

1 2 4 5 3
4 5 2 3 1

第二步，划分左右子树。去掉节点 1，考察剩下的节点，不论是在先序遍历还是后序遍历中，一定是**所有的左子树节点在前，所有右子树节点在后**，我们可以看出剩下四个节点对于 1 节点的从属关系是：

1 2 4 5 3
4 5 2 3 1

其中蓝色代表左子树，绿色代表右子树。这个划分必定是唯一的，因为我们可以考察左子树的根节点，先序遍历的第一个蓝色节点（下标为 1，即节点 2）必定等于后序遍历的最后一个蓝色节点；由于前者是已知的，因而我们可以直接在后序遍历中寻找这个节点（发现其下标为 2），在其之前（包括自身）的都属于左子树，之后的都属于右子树。完全二叉树保证了左右子树总是同时存在的。

第三步，递归建树。前序遍历和后序遍历序列都被划分为了代表左右子树的两个连续的子序列，对左右子序列分别递归建树（重复三个步骤），并接在根节点 1 下即可。

建树完成后，同样通过递归输出中序遍历即可。

代码

代码中将 `preIndex` 直接作为一个累加全局的变量，这里利用了一个性质，即在我们的建树代码中，将节点接到树上的顺序是“父节点-左子树递归-右子树递归”，这恰好是符合先序遍历的。另一种更直观的写法是将 `preStart`，`preEnd` 加入到 `buildTree` 函数的参数中，不影响建树结果。

```
int pre[4000005], post[4000005];
int n;
int preIndex = 0;

struct TreeNode {
      int val;
      TreeNode *left, *right;
      TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

TreeNode* buildTree(int postStart, int postEnd) {
      if (postStart > postEnd || preIndex >= n) {
            return nullptr;
      }
      TreeNode* root = new TreeNode(pre[preIndex++]); // 第一步，建立父节点
```

```
        // 第二步，查找左子树根节点在后序遍历中的位置
        int postIndex;
        for (postIndex = postStart; postIndex <= postEnd; ++postIndex) {
            if (post[postIndex] == pre[preIndex]) {
                break;
            }
        }
        if(postIndex > postEnd) return root;
        // 第三步，递归建树
        root->left = buildTree(postStart, postIndex);
        root->right = buildTree(postIndex + 1, postEnd - 1);
        return root;
    }

    void inorderPrint(TreeNode* root) {
        if (root) {
            inorderPrint(root->left);
            cout << root->val << " ";
            inorderPrint(root->right);
        }
    }
```

3. 游戏

分析

本题通过简单的模拟就可以解决。建立一个**存储还没有配对的 A 的位置的栈**，记变为 A 的 C 球个数为 C_A，变为 B 的 C 球个数为 C_B。

按顺序遍历各个球，按照以下的规则做处理：

1. 如果是 A 球，将其入栈；
2. 如果是 B 球，且栈不为空，取栈顶的球 A 取出进行配对；
3. 如果是 B 球，且栈为空，则用在其左侧且距离最远的（坐标最小的）、还没有使用过的 C 球进行配对，若找不到这样的 C 球，直接输出 `False` 结束程序。

若遍历成功完成，则所有 B 球都已经被配对。检查栈中是否还有未配对的 A，如果有，则每次取出栈顶的球 A，用在其右侧且距离最远的（坐标最大的）、还没有使用过的 C 球进行配对，若找不到这样的 C 球，直接输出 `False` 结束程序。

若栈成功清空，则实现了所有的配对，满足$Cnt_A + C_A = Cnt_B + C_B$，输出 `True`。

不加证明地，这个模拟使用了多处贪心的思想：①优先让已有的 AB 球配对，且尽可能用靠左的 B 球配对靠右的 A 球②如果要用 C 球配对 B，则尽可能用最左边的 C 球与最左边的 B 球配对，③如果要用 C 球配对 A，则尽可能用最右边的 C 球与最右边的 A 球配对。

代码

```
int main() {
    int n;
    cin >> n;
    char a[n + 1];
    bool vis[n + 1];
    int st[n + 1], stTop = 0, acnt = 0, bcnt = 0, cuseda = 0, cusedb = 0;

    for (int i = 0; i < n; i++) {
        cin >> a[i];
        vis[i] = 0;
```

```
    }
    for (int i = 0; i < n; i++) {
        if (a[i] == 'A') {
            st[stTop++] = i;
            acnt++;
        } else if (a[i] == 'B') {
            bcnt++;
            if (stTop) {
                stTop--;
            } else {
                bool found = false;
                for (int j = 0; j < i; j++) {
                    if (a[j] == 'C' && !vis[j]) {
                        vis[j] = 1;
                        cuseda++;
                        found = true;
                        break;
                    }
                }
                if (!found) {
                    cout << "False";
                    return 0;
                }
            }
        }
    }
    int cpos = n-1;
    while (stTop) {
        for(; cpos >= st[stTop - 1]; cpos--){
            if(a[cpos] == 'C' && !vis[cpos]){
                cusedb++;
                stTop--;
                cpos--;
                break;
            }
        }
        if(cpos < st[stTop - 1]){
            cout << "False";
            return 0;
        }
    }
    if(acnt + cuseda == bcnt + cusedb) {
        cout << "True";
    } else {
        cout << "False";
    }
}
```