实验报告

数据结构 *Data Structures*

报告标题: 二叉树算法
学号: 19240212
姓名: 华博文
日期: 2025 年 11 月 1 日

一 实验环境

1 操作系统

Ubuntu 24.04.3 LTS x86_64，Linux 6.14.0-28-generic，AMD Ryzen 7 7700 (16) @ 5.3GHz。

2 编程工具

NeoVim v0.11.1，gcc 13.3.0，GNU Make 4.3，cmake 3.28.3。

3 其他工具

LaTeX (TeX*studio*)。

二 实验内容及其完成情况

1 实验目的

1. 掌握二叉树的基本概念和存储结构；
2. 理解二叉树的四种遍历方式及其应用：先序、中序、后序、层次遍历；
3. 掌握二叉树常用操作（创建、遍历、求高度、计数、查找、求父结点）的算法实现；
4. 提高使用 C++ 面向对象实现复杂数据结构的能力。

2 实验内容

实现一个完整的二叉树类，包含以下功能：

1. 根据带虚结点（如使用特殊符号表示空子结点）的先序序列创建二叉树；
2. 四种遍历：先序、（递归 / 非递归）中序、后序和层次遍历；
3. 求二叉树的高度和结点总数；
4. 查找指定结点并返回该结点的父结点指针或值（若存在）。

三 实现细节与过程

本节简要说明实现的关键点与算法契约（输入 / 输出 / 错误处理）。

1 设计契约

- 输入：使用带虚结点的先序序列（例如：AB##CD##E##），其中 # 表示空结点；
- 输出：构建出的二叉树，支持各种遍历输出、查询高度、结点数、查找结点及其父结点；
- 错误模式：非法输入（例如序列过短或不匹配）将通过异常或返回特殊值提示调用者。

2 数据结构定义

核心数据类型为二叉树结点（C++ 结构体或类内部私有类型）：

```
template <typename T>
class BinaryTree {
public:
    struct Node {
        T val;
        std::unique_ptr<Node> left;
        std::unique_ptr<Node> right;
        explicit Node(const T &v) : val(v), left(nullptr),
        ↪  right(nullptr) {}
    };

private:
    ...
};
```

外层提供 BinaryTree 类，封装创建与各类操作，并在析构时释放内存。

3 从带虚结点的先序序列创建二叉树

算法思路：使用迭代器或索引从序列头部递归构建，遇到虚结点符号（例如 #）返回空。
递归伪代码：

```
std::unique_ptr<Node> buildFromPreorderRec(const std::vector<T>
↪  &tokens, size_t &idx, const T &nullValue) {
    if (idx >= tokens.size())
        return nullptr;
    T cur = tokens[idx++];
    if (cur == nullValue)
        return nullptr;
    auto node = std::make_unique<Node>(cur);
    node->left = buildFromPreorderRec(tokens, idx, nullValue);
    node->right = buildFromPreorderRec(tokens, idx, nullValue);
    return node;
}
```

时间复杂度：$\mathcal{O}(n)$，空间复杂度：递归栈最坏 $\mathcal{O}(n)$。

4 遍历实现

先序 / 中序 / 后序遍历均提供递归实现，并在需要时提供基于栈的非递归版本（中序与先序常用）。层次遍历使用队列实现（BFS）。

```
std::vector<T> levelOrderRec(Node *node) const {
    std::vector<T> out;
    if (!node)
        return out;
    std::queue<Node *> q;
    q.push(node);
    while (!q.empty()) {
        Node *cur = q.front();
        q.pop();
        out.push_back(cur->val);
        if (cur->left)
            q.push(cur->left.get());
        if (cur->right)
            q.push(cur->right.get());
    }
    return out;
}
```

时间复杂度：每种遍历都访问每个结点一次，均为 $\mathcal{O}(n)$，空间复杂度：层次遍历 $\mathcal{O}(\texttt{width})$（最坏为 $\mathcal{O}(n)$），递归版本的额外空间为递归深度 $\mathcal{O}(h)$。

5 高度与结点计数

使用递归方式计算高度与结点总数：

```
int heightRec(Node *node) const {
    if (!node)
        return 0;
    return 1 + std::max(heightRec(node->left.get()),
     ↪  heightRec(node->right.get()));
}

int countRec(Node *node) const {
    if (!node)
        return 0;
```

```
    return 1 + countRec(node->left.get()) +
     ↪  countRec(node->right.get());
}
```

时间复杂度均为 $\mathcal{O}(n)$，空间复杂度最坏 $\mathcal{O}(n)$（退化链）。

6 查找指定结点与求父结点

查找结点：可使用任一遍历（递归或非递归）在 $\mathcal{O}(n)$ 时间找到第一个匹配值的结点指针。

查找父结点：在遍历过程中记录父指针，或在 BinaryTree 中维护父指针（在创建时设定）。下面给出递归查找父结点的示例：

```
Node *findParentRec(Node *node, Node *parent, const T &value) const {
    if (!node)
        return nullptr;
    if (node->val == value)
        return parent;
    Node *l = findParentRec(node->left.get(), node, value);
    if (l)
        return l;
    return findParentRec(node->right.get(), node, value);
}
```

时间复杂度：$\mathcal{O}(n)$。

7 测试用例与运行结果

设计了若干用例覆盖常见场景：空树、单结点、完全二叉树、非完全二叉树以及包含虚结点的序列。示例输入与期望输出：

1. 输入先序序列：ABD##E##C##（对应的树为：A 的左子树为 B，B 的左子树为 D，右子树为 E；A 的右子树为 C）
2. 先序输出：A B D E C
3. 中序输出：D B E A C
4. 后序输出：D E B C A
5. 层次输出：A B C D E
6. 结点总数：5，高度：3
7. 查找结点 E 返回存在；查找父结点返回 B。

在我的本地测试中，上述用例均输出与预期一致，边界用例（全为 # 的序列表示空树）亦能正确处理。

四　复杂度分析与讨论

对主要操作给出时间/空间复杂度总结：

- 构建二叉树（先序带虚结点）：时间 $\mathcal{O}(n)$，空间 $\mathcal{O}(n)$（用于递归栈或临时序列）；
- 各种遍历（先/中/后/层次）：时间均为 $\mathcal{O}(n)$，空间最坏 $\mathcal{O}(n)$（递归深度或队列/栈）；
- 计算高度/计数：时间 $\mathcal{O}(n)$，空间 $\mathcal{O}(h)$（递归栈）；
- 查找结点 / 查找父结点：时间 $\mathcal{O}(n)$，空间 $\mathcal{O}(h)$（递归）或 $\mathcal{O}(1)$（迭代并记录）。

常见优化与注意点：

- 若二叉树是平衡的，递归栈深度为 $\mathcal{O}(\log n)$；退化链情况退化到 $\mathcal{O}(n)$。
- 若频繁查找父结点，可在结点结构中维护父指针，查找父结点复杂度可降为 $\mathcal{O}(1)$（若已有结点指针）；
- 若结点值不是字符而是复杂对象，建议使用模板类并传入比较函数或重载比较操作符以支持泛型。

五　实验总结

本次实验实现了一个功能完整的二叉树类，并验证了：

- 能根据带虚结点的先序序列正确构建二叉树；
- 提供递归与非递归的遍历实现，层次遍历使用队列实现；
- 能正确计算结点总数和树的高度；
- 能查找指定结点并返回其父结点（递归实现），并对复杂度给出分析。

代码结构清晰，注释完整；测试用例覆盖了常见与边界场景。该实现为进一步扩展（例如添加删除结点、子树复制、序列化/反序列化等）提供了良好基础。本报告按要求完成了二叉树的设计、实现与测试，并对复杂度进行了分析，满足实验要求。