中山大学计算机学院本科生实验报告

(2025 学年第 1 学期)

课程名称：数据结构与算法实验　　任课老师：张子臻

年级	2024 级	专业（方向）	计算机科学与技术 (人工智能与大数据)
学号	24325155	姓名	梁桂铭
电话	15817681625	Email	lianggm8@mail2.sysu.edu.cn
开始日期	2025 年 11 月 5 日	完成日期	2025 年 11 月 5 日

第一题：实验题目

给定一组无序整数，以第一个元素为根节点，按顺序插入结点生成一棵二叉搜索树，并对其进行中序遍历和先序遍历。

输入包括多组数据，每组数据包含两行：第一行为整数 m(1<=m<=3000)，表示该组数据中整数的数目，第二行给出 m 个整数，相邻整数间用一个空格间隔。最后一组数据后紧跟着包含 0 的一行输入，标识输入的结束。

每组输入产生两行输出，第一行是中序遍历结果，第二行是先序遍历结果，每个整数后面带一个空格，每行中第一个整数前无空格。

实验目的

1. 掌握二**叉搜索树**的基本概念和性质
2. 理解二叉搜索树的构建过程
3. 掌握二叉树的**中序遍历和先序遍历算法**
4. 熟练运用**递归思想**实现树的相关操作

算法设计

1. 数据结构定义

定义二叉树结点结构体，包含三个成员：

- `val`: 存储结点的整数值
- `left`: 指向左子树的指针
- `right`: 指向右子树的指针

2. 主要算法设计

(1) insert 函数

采用递归方式实现结点插入：

- 若待插入值大于等于当前结点值，则插入右子树
- 若待插入值小于当前结点值，则插入左子树
- 递归查找合适的插入位置，直到找到空位置

(2) 中序遍历算法

递归实现中序遍历（左子树 → 根节点 → 右子树）：

- 递归遍历并打印左子树
- 打印根节点的 val
- 递归遍历并打印右子树

(3) 先序遍历算法

递归实现先序遍历（根节点 → 左子树 → 右子树）：

- 打印根节点的 val
- 递归遍历并打印左子树
- 递归遍历并打印右子树

3. 复杂度分析

整个程序的时间复杂度主要由构建过程决定：

(1) 时间复杂度

$$\text{总时间复杂度} = T_{\text{build}} + T_{\text{traverse}}$$

- **最坏情况时间复杂度**：$O(n^2) + O(n) = \mathbf{O(}n^2\mathbf{)}$
- **平均情况时间复杂度**：$O(n \log n) + O(n) = \mathbf{O(}n \log n\mathbf{)}$

(2) 空间复杂度

$$\text{总空间复杂度} = S_{\text{heap}} + S_{\text{stack}}$$

- **最坏情况空间复杂度**：$O(n) + O(n) = \mathbf{O}(n)$
- **平均情况空间复杂度**：$O(n) + O(\log n) = \mathbf{O}(n)$

程序运行与测试

测试样例 1

输入：

```
9
10 4 16 9 8 15 21 3 12
0
```

输出：

```
3 4 8 9 10 12 15 16 21
10 4 3 9 8 16 15 12 21
```

测试样例 2

输入：

```
6
20 19 16 15 45 48
0
```

输出：

```
15 16 19 20 45 48
20 19 16 15 45 48
```

程序代码

```cpp
#include <iostream>
#include <stack>
using namespace std;
struct tree {
    int val;
    tree* left;
    tree* right;
};
void insert(tree* root, int val) {
    if (root == nullptr) return;
    tree* p = new tree{val, nullptr, nullptr};
    if (val >= root -> val) {
```

```
        if (root -> right == nullptr) {
            root -> right = p;
        }
        else insert(root -> right, val);
    }else {
        if (root -> left == nullptr) {
            root -> left = p;
        }
        else insert(root -> left, val);
    }
}
void in(tree* root) {
    if (root == nullptr) return;
    in(root -> left);
    cout << root -> val << " ";
    in(root -> right);
}
void pre(tree* root) {
    if (root == nullptr) return;
    cout << root -> val << " ";
    pre(root -> left);
    pre(root -> right);
}
int main () {
    int m;
    while (cin >> m && m != 0) {
        tree* root = new tree{0, nullptr, nullptr};
        int tmp;
        for (int i = 0; i < m; i++) {
            cin >> tmp;
            if (i == 0) root -> val = tmp;
            else {
                insert(root, tmp);
            }
        }
        in(root);
        cout << endl;
        pre(root);
        cout << endl;
    }
    return 0;
}
```

实验总结

通过本次实验，我深入理解了二叉搜索树的特性和构建方法。递归方法在树的操作中非常有效，代码简洁易懂。实验中需要注意内存管理，避免内存泄漏。同时，处理多组数据输入时需要正确控制循环条件。本次实验巩固了我对二叉树遍历算法的掌握，提高了递归编程能力。

第二题：实验题目

Given a binary tree, your task is to get the height and size(number of nodes) of the tree. The Node is defined as follows.

```
struct Node {
Node *lc, *rc;
char data;
};
void query(const Node *root, int &size, int &height)
{
// put your code here
}
```

实验目的

1. 掌握二叉树的基本遍历方法
2. 理解递归在树结构计算中的应用
3. 学会计算二叉树的深度（高度）和规模（节点数）
4. 培养**递归思维**和**分治策略**的编程能力

算法设计

1. 数据结构定义

题目已给出二叉树结点的定义：

```
struct Node {
    Node *lc, *rc;
    char data;
};
```

2. 递归算法设计

采用**后序遍历**（左子树 → 右子树 → 根节点）的递归策略：

(1) 基本情况（递归终止条件）

- 如果当前节点为空（`root == nullptr`），则：
  - 节点数 `size += 0`
  - 树高度 `height += 0`

(2) 递归步骤

1. 递归计算左子树的节点数和高度
2. 递归计算右子树的节点数和高度
3. 计算当前子树的节点数：`size = leftSize + rightSize + 1`
4. 计算当前子树的高度：`height = max(leftHeight, rightHeight) + 1`

3. 算法正确性分析

- **节点数计算**：二叉树的节点总数等于左子树节点数加上右子树节点数加上根节点
- **高度计算**：二叉树的高度等于左右子树中较高的那个子树的高度加 1（根节点所在层）
- **边界处理**：空树的高度为 0，节点数为 0，符合题目定义

程序运行与测试

测试用例设计

程序代码

```cpp
#include <iostream>
#include "play.h"
#include <queue>
using namespace std;

void query(const Node *root, int &size, int &height) {
    if (root == nullptr) {
        size = 0;
        height = 0;
        return;
    }

    int leftSize, leftHeight, rightSize, rightHeight;

    // 递归计算左子树
    query(root->lc, leftSize, leftHeight);
    // 递归计算右子树
    query(root->rc, rightSize, rightHeight);

    // 计算当前树的节点数 = 左子树节点数 + 右子树节点数 + 1(根节点)
    size = leftSize + rightSize + 1;

    // 计算当前树的高度 = max(左子树高度, 右子树高度) + 1
    height = max(leftHeight, rightHeight) + 1;
}
```

复杂度分析

时间复杂度

- 算法需要访问二叉树中的每个节点 exactly once
- 每个节点的处理时间为常数时间 $O(1)$
- **总时间复杂度**：$\mathbf{O(n)}$，其中 n 为二叉树节点数

空间复杂度

- **堆空间**：不需要额外分配动态内存，$O(1)$
- **栈空间**：递归调用栈的深度等于树的高度
  - **最好情况**（平衡二叉树）：$O(\log n)$
  - **最坏情况**（斜树）：$O(n)$
- **总空间复杂度**：$\mathbf{O(h)}$，其中 h 为二叉树高度

实验总结

通过本次实验，我深入理解了二叉树的递归遍历方法，并掌握了以下重点：

1. **递归思维**：将复杂问题分解为相似的子问题，通过解决子问题来构建原问题的解
2. **后序遍历应用**：先处理左右子树，再处理根节点的策略非常适合计算树的聚合属性
3. **边界条件处理**：正确处理空树情况是算法健壮性的重要保证
4. **效率分析**：该算法的时间复杂度为最优的 $O(n)$，每个节点只访问一次

第三题：实验题目

给定二叉树的**前序遍历**序列和**中序遍历**序列，求该二叉树的**后序遍历**序列。

实验目的

1. 掌握由前序和中序遍历序列重构二叉树的递归算法
2. 学会通过遍历序列推导二叉树的后序遍历结果
3. 培养递归分治思想和数组索引处理能力

算法设计

1. 核心思想

利用二叉树遍历序列的特性：

- **前序遍历**：根节点 → 左子树 → 右子树
- **中序遍历**：左子树 → 根节点 → 右子树
- **后序遍历**：左子树 → 右子树 → 根节点

2. 递归算法设计

(1) 递归函数参数

```
void post(int s1, int e1, int s2, int e2, int* A, int* B)
```

- `s1, e1`: 前序序列 A 的当前处理区间 $[s1, e1)$
- `s2, e2`: 中序序列 B 的当前处理区间 $[s2, e2)$
- `A`: 前序遍历序列数组
- `B`: 中序遍历序列数组

(2) 递归步骤

1. **确定根节点**：前序序列的第一个元素 `A[s1]` 就是当前子树的根节点
2. **定位根节点**：在中序序列中找到根节点的位置 `pos`
3. **计算左右子树大小**：左子树节点数 `l = pos - s2`
4. **递归处理左子树**：
   - 前序序列：`[s1+1, s1+1+l)`
   - 中序序列：`[s2, pos)`
5. **递归处理右子树**：
   - 前序序列：`[s1+1+l, e1)`
   - 中序序列：`[pos+1, e2)`
6. **输出根节点**：按照后序遍历顺序，最后输出根节点

(3) 递归终止条件

当处理区间为空时（`s1 >= e1`），直接返回。

程序运行与测试

测试样例

输入：

```
10
7 2 0 5 8 4 9 6 3 1
7 5 8 0 4 2 6 3 9 1
```

输出：

```
8 5 4 0 3 6 1 9 2 7
```

程序代码

```cpp
#include <iostream>
using namespace std;

void post(int s1, int e1, int s2, int e2, int* A, int* B) {
    if (s1 >= e1) return;
    int key = A[s1];
    int pos = s2;
    for (; pos < e2; pos++) {
        if (B[pos] == key) {
            break;
        }
    }
    int l = pos - s2;
    post(s1 + 1, s1 + 1 + l, s2, pos, A, B);
    post(s1 + 1 + l, e1, pos + 1, e2, A, B);
    cout << key << " ";
}

int main () {
    int N;
    cin >> N;
    int* A = new int[N];
    int* B = new int[N];
    for (int i = 0; i < N; i++) cin >> A[i];
    for (int i = 0; i < N; i++) cin >> B[i];
    post(0, N, 0, N, A, B);
    delete[] A;
    delete[] B;
    return 0;
}
```

复杂度分析

时间复杂度

- **最坏情况**：每次都需要线性扫描中序序列寻找根节点位置，时间复杂度为 $O(n^2)$
- **平均情况**：如果树比较平衡，时间复杂度为 $O(n \log n)$

空间复杂度

- **递归栈空间**：取决于树的高度
  - 最好情况（平衡二叉树）：$O(\log n)$
  - 最坏情况（斜树）：$O(n)$
- **数组空间**：存储两个遍历序列，$O(n)$
- **总空间复杂度**：$O(n)$

实验总结

通过本次实验，我深入理解了二叉树三种遍历序列之间的关系，并掌握了以下重要知识点：

1. **遍历序列特性**：前序序列首元素为根，中序序列根元素划分左右子树
2. **递归分治策略**：将大问题分解为相似的子问题，分别处理左右子树
3. **索引边界处理**：精确计算左右子树在序列中的位置范围是关键
4. **内存管理**：正确使用动态内存分配和释放，避免内存泄漏

附录、提交文件清单

第一题

```cpp
#include <iostream>
#include <stack>
using namespace std;
struct tree {
    int val;
    tree* left;
    tree* right;
};
void insert(tree* root, int val) {
    if (root == nullptr) return;
    tree* p = new tree{val, nullptr, nullptr};
    if (val >= root -> val) {
        if (root -> right == nullptr) {
            root -> right = p;
```

```
        }
        else insert(root -> right, val);
    }else {
        if (root -> left == nullptr) {
            root -> left = p;
        }
        else insert(root -> left, val);
    }
}
void in(tree* root) {
    if (root == nullptr) return;
    in(root -> left);
    cout << root -> val << " ";
    in(root -> right);
}
void pre(tree* root) {
    if (root == nullptr) return;
    cout << root -> val << " ";
    pre(root -> left);
    pre(root -> right);
}
int main () {
    int m;
    while (cin >> m && m != 0) {
        tree* root = new tree{0, nullptr, nullptr};
        int tmp;
        for (int i = 0; i < m; i++) {
            cin >> tmp;
            if (i == 0) root -> val = tmp;
            else {
                insert(root, tmp);
            }
        }
        in(root);
        cout << endl;
        pre(root);
        cout << endl;
    }
    return 0;
}
```

第二题

```
#include <iostream>
#include "play.h"
#include <queue>
using namespace std;

void query(const Node *root, int &size, int &height) {
    if (root == nullptr) {
```

```
        size = 0;
        height = 0;
        return;
    }

    int leftSize, leftHeight, rightSize, rightHeight;

    // 递归计算左子树
    query(root->lc, leftSize, leftHeight);
    // 递归计算右子树
    query(root->rc, rightSize, rightHeight);

    // 计算当前树的节点数 = 左子树节点数 + 右子树节点数 + 1(根节点)
    size = leftSize + rightSize + 1;

    // 计算当前树的高度 = max(左子树高度, 右子树高度) + 1
    height = max(leftHeight, rightHeight) + 1;
}
```

第三题

```
#include <iostream>
using namespace std;

void post(int s1, int e1, int s2, int e2, int* A, int* B) {
    if (s1 >= e1) return;
    int key = A[s1];
    int pos = s2;
    for (; pos < e2; pos++) {
        if (B[pos] == key) {
            break;
        }
    }
    int l = pos - s2;
    post(s1 + 1, s1 + 1 + l, s2, pos, A, B);
    post(s1 + 1 + l, e1, pos + 1, e2, A, B);
    cout << key << " ";
}

int main () {
    int N;
    cin >> N;
    int* A = new int[N];
    int* B = new int[N];
    for (int i = 0; i < N; i++) cin >> A[i];
    for (int i = 0; i < N; i++) cin >> B[i];
    post(0, N, 0, N, A, B);
    delete[] A;
    delete[] B;
    return 0;
```

```
30 }
```