

# 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      实验任课教师：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	242325157	姓名	梁玮麟
电话	18620062182	Email	3371676041@qq.com
开始日期	2025.10.22	结束日期	2025.10.25

## 第一题

### 1、实验题目

#### z9-二叉搜索树的遍历

**题目描述** 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。

**输入描述** 输入包括多组数据，每组数据包含两行：第一行为整数  $m(1 \leq m \leq 3000)$ ，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。

**输出描述** 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

#### 输入样例

```
9
10 4 16 9 8 15 21 3 12
6
20 19 16 15 45 48 0
```

#### 输出样例

```
3 4 8 9 10 12 15 16 21
10 4 3 9 8 16 15 12 21
15 16 19 20 45 48
20 19 16 15 45 48
```

### 2、实验目的

- 学会构建二叉搜索树
- 掌握中序遍历和前序遍历的概念，并成功实现

### 3、算法设计

设计思路：

- 首先整理清楚二叉搜索树的过程：
  - 判断根节点与当前插入的数的大小；
  - 如果小于当前节点，则与节点的左子节点比较；
  - 否则与当前节点的右子节点比较；
  - 如果当前节点为空，则将当前节点的数值设为要插入的数，同时将左右子节点设为空。

- 根据搜索结果过程，定义一个类 `node`，可以储存当前节点的值 `val`，同时有着两个指针，分别指向 `left` 和 `right` 子节点。
- 根据输入数据，**构建树**：
  - 首先，当 `root` 指向空，给 `root` 节点分配内存；
  - 否则：
    1. 设置一个 `parent` 指针指向 `nullptr`, `p` (代表当前指针) 指向 `root`。
    2. 用一个 `while` 循环，用 `parent` 指向 `p`, 然后根据数值和 `p` 的值的比较结果移动 `p` 指针，直到 `p` 指向 `nullptr`。
    3. 因为 `p` 指向空，所以比较当前数值和 `parent->val` 的大小关系，决定 `p` 是左节点还是右节点，并给 `parent->left/right` 分配内存。
- 对于中序遍历，可以用递归实现：
  1. 终止条件：
    - 果当前指针指向 `nullptr`, 则直接 `return`;
  2. 首先调用左节点；
  3. 然后输出当前节点的值
  4. 再调用右节点。
- 对于前序遍历，可以用递归实现：
  1. 终止条件：
    - 果当前指针指向 `nullptr`, 则直接 `return`;
  2. 首先输出当前节点的值
  3. 然后调用左节点；
  4. 再调用右节点。

### 流程图：

```
// Flow of z9-BST Traversal
start
|
|--> while (read m) and m != 0:
|     create node* root = nullptr
|     for i in [0..m-1]:
|         read x
|         if root == nullptr:
|             root = new node(x)
|         else:
|             node *parent = nullptr, *p = root
|             while (p != nullptr):
|                 parent = p
|                 if (x < p->val) p = p->left
|                 else                  p = p->right
|                 // attach new node under parent
|                 if (x < parent->val) parent->left = new node(x)
|                 else                      parent->right = new node(x)
|
|             // inorder traversal
|             call inorder(root)    // L - V - R
|             print '\n'
|
|             // preorder traversal
|             call preorder(root)   // V - L - R
|             print '\n'
|
|             delete root // 递归析构释放整棵树
|
```

end

### 复杂度分析:

- 构建 BST:

- 平均时间复杂度:  $O(m \log m)$ , 因为插入每个节点期望深度为  $\log m$ ;
- 最坏时间复杂度:  $O(m^2)$ , 当输入序列有序时退化为链。

- 遍历:

- 时间复杂度:  $O(m)$ , 每个节点访问一次;
- 空间复杂度:  $O(h)$ , 递归栈深度  $h$  为树高, 平均  $O(\log m)$ , 最坏  $O(m)$ 。

### 细节注意:

- parent 指针是必要的, 否则 p 为空时, 直接给 p 分配内存, 会导致每一个节点都是孤立的。

### 具体实现:

```
//z9-二叉搜索树的遍历
#include <iostream>
using namespace std;
class node {
public:
    int val;
    node *left;
    node *right;
    node() = default;
    node(int val) : val(val), left(nullptr), right(nullptr) {}
    ~node() {
        if (right) {
            delete right;
        }
        if (left) {
            delete left;
        }
    }
};
void inorder(node *&p) {
    if (p == nullptr)
        return;
    if (p->left) {
        inorder(p->left);
    }
    cout << p->val << " ";
    if (p->right) {
        inorder(p->right);
    }
}
void preorder(node *&p) {
    if (p == nullptr)
        return;
    cout << p->val << " ";
    if (p->left) {
        preorder(p->left);
    }
    if (p->right) {
```

```

        preorder(p->right);
    }
}

int main() {
    int m;
    while (cin >> m && m != 0) {
        node *root = nullptr;
        for (int i = 0; i < m; i++) { // 构建树
            int num;
            cin >> num;
            if (root == nullptr) {
                root = new node(num);

            } else {
                node *parent = nullptr;
                node* p=root;
                while (p) {
                    parent = p;
                    if (num > p->val)
                        p = p->right;
                    else
                        p = p->left;
                }
                if(num<parent->val){
                    parent->left=new node(num);
                }
                else{
                    parent->right=new node(num);
                }
            }
        }
        inorder(root);
        cout << endl;
        preorder(root);
        cout << endl;
        if (root)
            delete root;
    }
    return 0;
}

```

#### 4、程序运行与测试

运行结果：

##### 测试样例一

- 标准输入：

```

12
25 45 42 27 8 30 18 43 32 41 11 12
10
15 42 35 1 45 24 49 5 34 18
15
36 27 52 24 17 8 4 37 49 5 26 13 40 21 55

```

0

- 实际输出:

```
8 11 12 18 25 27 30 32 41 42 43 45  
25 8 18 11 12 45 42 27 30 32 41 43  
1 5 15 18 24 34 35 42 45 49  
15 1 5 42 35 24 18 34 45 49  
4 5 8 13 17 21 24 26 27 36 37 40 49 52 55  
36 27 24 17 8 4 5 13 21 26 52 37 49 40 55
```

- 期望输出:

```
8 11 12 18 25 27 30 32 41 42 43 45  
25 8 18 11 12 45 42 27 30 32 41 43  
1 5 15 18 24 34 35 42 45 49  
15 1 5 42 35 24 18 34 45 49  
4 5 8 13 17 21 24 26 27 36 37 40 49 52 55  
36 27 24 17 8 4 5 13 21 26 52 37 49 40 55
```

## 测试样例二

- 标准输入:

```
20  
25 37 14 49 10 13 8 3 38 31 21 27 29 33 28 41 11 26 47 9  
30  
19 22 7 9 49 21 35 43 3 11 10 41 30 25 6 48 1 29 13 36 37 45 5 12 17 14 32 27 20 46  
0
```

- 实际输出:

```
3 8 9 10 11 13 14 21 25 26 27 28 29 31 33 37 38 41 47 49  
25 14 10 8 3 9 13 11 21 37 31 27 26 29 28 33 49 38 41 47  
1 3 5 6 7 9 10 11 12 13 14 17 19 20 21 22 25 27 29 30 32 35 36 37 41 43 45 46 48 49  
19 7 3 1 6 5 9 11 10 13 12 17 14 22 21 20 49 35 30 25 29 27 32 43 41 36 37 48 45 46
```

- 期望输出:

```
3 8 9 10 11 13 14 21 25 26 27 28 29 31 33 37 38 41 47 49  
25 14 10 8 3 9 13 11 21 37 31 27 26 29 28 33 49 38 41 47  
1 3 5 6 7 9 10 11 12 13 14 17 19 20 21 22 25 27 29 30 32 35 36 37 41 43 45 46 48 49  
19 7 3 1 6 5 9 11 10 13 12 17 14 22 21 20 49 35 30 25 29 27 32 43 41 36 37 48 45 46
```

## 5、实验总结与心得

- 这道题对我来说，难点是理解**二叉搜索树的原理以及构建树的过程**。通过这个实验，让我掌握了二叉搜索树的具体实现，并提高了我的代码能力。

## 第二题

### 1、实验题目

#### z9-Play with Tree

**题目描述** 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;  
};
```

Implement the following function.

```
void query(const Node *root, int &size, int &height)  
{  
    // put your code here  
}
```

**提示** In this problem, we assume that the height of the tree with only one node is 1.

## 2、实验目的

- 掌握二叉树的具体结果；
- 对于二叉树的各种算法，想到可以把整个树划分成左子树和右子树，然后递归计算。

## 3、算法设计

设计思路：

- 把当前节点分成左子树和右子树；因为 size 和 height 是引用，可以通过这两个变量返回结果。
- 递归：
  - 终止条件：如果当前指向 nullptr，则 size 和 height 都是 0；
  - 因为 size 和 height 是作为返回结果的接口，所以不能直接用这两个变量存储递归结果。所以需要新开 lsize, lheight, rsize, rheight。
  - 递归调用左子节点。
  - 递归调用右子节点。
  - 因为 size 统计的是总数，所以直接令 size = lsize + rsize + 1；
  - height 并非统计总数，而需要比较 lheight 和 rheight 的大小，取最大值，然后 +=1（因为需要加上当前高度）。

流程图：

```
// Flow of query(root, size, height)  
start  
|  
|--> function query(root, ref size, ref height):  
|     if (root == nullptr):  
|         size = 0  
|         height = 0  
|         return  
|  
|         int lsize=0, rsize=0, lheight=0, rheight=0  
|         query(root->lc, lsize, lheight)  
|         query(root->rc, rsize, rheight)  
|  
|         size = lsize + rsize + 1  
|         height = max(lheight, rheight) + 1  
|         return  
|  
end
```

复杂度分析：

- 时间复杂度：O(n)，每个节点被访问一次；

- **空间复杂度:**  $O(h)$ , 递归栈深度为树高, 最坏  $O(n)$ , 平均  $O(\log n)$ 。
- **额外空间:** 常数级辅助变量。

### 细节注意:

- 递归思想: 把整个问题**分解成若干相似子问题**。在这一题就是对于任意一个节点, 分成左子树和右子树。
- 无论是 size 还是 height, 都需要加上自己。

### 具体实现:

```
//z9-Play with Tree
#include <iostream>
#include "play.h"
using namespace std;
void query(const Node *root, int &size, int &height) {
    if (root == nullptr) {
        size = 0;
        height = 0;
        return;
    } else {
        int lsize = 0, rsize = 0;
        int lheight = 0, rheight = 0;
        query(root->lc, lsize, lheight);
        query(root->rc, rsize, rheight);
        size = lsize + rsize + 1;
        height = lheight > rheight ? lheight : rheight;
        height += 1;//算上自己的高度
    }
}
```

## 4、程序运行与测试

### 运行结果:

#### 测试样例一

- 标准输入:

- 实际输出:

```
YES
```

YES  
YES  
YES  
YES  
YES  
YES

- 期望输出：

(同上)

## 5、实验总结与心得

- 这题卡了我一会。主要是一开始没想到可以通过左右子树这种递归思路做，而只是简单地想通过迭代完成。
- 这道题目启发我，以后碰到这种二叉树的题目，应该大概都可以把它分成左右去想。

## 第三题

### 1、实验题目

z9-postorder

**题目描述** Given the pre-order traversal sequence A[] and in-order traversal sequence B[] of a tree, find the post-order traversal sequence.

**输入描述** Input contains 3 lines; The first line contains an integer n( $n \leq 100000$ ), the number of node in the tree. The second line contains n distinct integers, which is the sequence A. The third line contains n distinct integers, which is the sequence B.

**输出描述** The postorder traversal sequence of the tree.

### 输入样例

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

### 输出样例

8 5 4 0 3 6 1 9 2 7

### 2、实验目的

- 深入理解前序表达式和中序表达式的结构特点
- 学会通过前序表达式和中序表达式构建二叉树。

### 3、算法设计

#### 设计思路：

- 采用递归设计。
- 思路：中序表达式，可以理解为输出左子树，然后输出当前节点，再输出右子树。而前序表达式，则先输出当前节点，再按顺序输出左子树和右子树。根据前序表达式，可以获得当前的 root 节点对应的数值，然后在中序表达式中对应的这个值将未被划分的整个部分分成左子树，root 节点，右子树。随后，根据前序表达式的顺序找到左子树的 root 节点和右子树的 root 节点，并在中序表达式中继续递归调用。

- 实现过程：

- 为了构建二叉树，先自定义一个 `node` 类，存储当前的值，`left` 指针指向左子节点，`right` 指向右子节点。
- 定义递归函数 `node* buildtree()`。因为需要在函数中查找 `pre 序列和 in 序列`，所以这两个必须传入。为了划分当前尚未划分的范围，需要 `left` 和 `right` 设定在中序表达式中划分的左边界和右边界。当前树中的数值也要传入，因为前序表达式中容易获取这个值，所以传入在前序表达式中的索引 `idxPre`，并传入一个 **哈希表**（映射从中序到前序的索引值，为了降低时间复杂度，否则每次递归都需要用  $O(n)$  的时间复杂度再找一次）。
- 递归函数的具体定义：
  1. 终止条件：如果 `left > right`，说明此时不存在未被放入树中的元素，返回 `nullptr`。
  2. 通过哈希表，找到 `idxPre` 在中序表达式中的索引 `idxIn`，并给当前节点分配内存。
  3. 找到左子树的范围与根节点：`[left:idxIn-1]`, `idxPre+1`。
  4. 找到右子树的范围与根节点：`[idxIn+1,right]`, `idxPre + 左子树节点数量 (idxIn-1-left+1) + 1`。
  5. 继续递归调用 `root->left=buildtree(...)`。
  6. 继续递归调用 `root->right=buildtree(...)`。
  7. 当前子树的根节点、左子树、右子树都已构建好，直接返回根节点。
- 构建后序表达式：
  1. 终止条件：根节点为 `nullptr`；
  2. 先调用左子树，再调用右子树，再输出当前节点值。

**流程图：**

```
// Flow of z9-postorder (build from preorder & inorder, then postorder-output)
start
|
|--> read n
|   read array pre[0..n-1]
|   read array in[0..n-1]
|
|--> // build value -> index map for inorder to achieve O(1) lookup
|--> create vector<int> idx(n)
|   for i in [0..n-1]: idx[in[i]] = i
|
|--> function build(pre, in, preIdx, L, R, idx) -> node*:
|   if (L > R): return nullptr
|   int rootIn = idx[ pre[preIdx] ]
|   node* root = new node(pre[preIdx])
|
|   int leftL = L
|   int leftR = rootIn - 1
|   int rightL = rootIn + 1
|   int rightR = R
|
|   int leftSize = leftR - leftL + 1
|
|   root->left = build(pre, in, preIdx + 1, leftL, leftR, idx)
|   root->right = build(pre, in, preIdx + leftSize + 1, rightL, rightR, idx)
|   return root
|
|--> node* root = build(pre, in, 0, 0, n-1, idx)
|
|--> function post(root):
|   if (root == nullptr) return
```

```

|     post(root->left)
|     post(root->right)
|     print(root->val, ' ')
|
|--> post(root)
|--> (optional) delete root
|
end

```

### 复杂度分析:

- **预处理哈希表:** 时间  $O(n)$ , 空间  $O(n)$ 。
- **构建二叉树:**
  - 每个结点常数时间处理一次, 总  $O(n)$ ;
  - 递归栈深度  $O(h)$ , 最坏  $O(n)$ , 平均  $O(\log n)$ 。
- **后序输出:**  $O(n)$ 。
- **总体复杂度:**
  - 时间复杂度:  $O(n)$
  - 空间复杂度:  $O(n)$  (映射表 + 递归栈)

### 细节注意:

- 需要在 main 函数中**提前用一次循环构建好哈希表**, 大大提升时间复杂度。
- 注意在前序表达式中找右子树根节点时的计算, 需要加上左子树节点数量后, **再加一**指向右子树的根节点。
- 本质其实是不断地对中序表达式进行**二分**。需要处理好递归函数在中序表达式中的处理范围以及终止条件。

### 具体实现:

```

#include <iostream>
#include <vector>
using namespace std;
class node {
public:
    int val;
    node *left;
    node *right;
    node() = default;
    node(int val) : val(val), left(nullptr), right(nullptr) {}
    ~node() = default;
};
node *biuldtree(const vector<int> &pre, const vector<int> &in, int idxPre,
                int left, int right, const vector<int>& Findididx) {
    if (left > right) {
        return nullptr;
    }
    // 在 in 中找到 pre[idxPre] 的位置
    int idxIn = Findididx[pre[idxPre]];
    node *root = new node(pre[idxPre]);
    int Lright = idxIn - 1;           // 左子树可以取到的右边界索引值
    int Lleft = left;                 // 左子树可以取到的左边界索引值
    int Right = right;               // 右子树可以取到的右边界索引值
    int Rleft = idxIn + 1;            // 右子树可以取到的左边界索引值
    int Lsize = Lright - Lleft + 1;   // 左子树的节点数量

```

```

root->left = biuldtree(pre, in, idxPre + 1, Lleft, Lright,Findidx);
// 默认前缀表达式下，根节点下一个就是左子树的节点。如果没有左子树，会自动返回 nullptr
root->right = biuldtree(pre, in, idxPre + Lsize + 1, Rleft, Rright,Findidx);
// 需要跨过左子树的节点数量，才是右子树的根节点
return root;
}
void post(node *root) {
    if (root == nullptr)
        return;
    else {
        post(root->left);
        post(root->right);
        cout << root->val << " ";
    }
}
int main() {
    int n;
    cin >> n;
    vector<int> pre(n);
    vector<int> in(n);
    vector<int> Findidx(n);
    for (int i = 0; i < n; i++) {
        cin >> pre[i];
    }
    for (int i = 0; i < n; i++) {
        cin >> in[i];
    }
    for (int i=0;i<n;i++){
        Findidx[in[i]]=i;
    }
    node *root = biuldtree(pre, in, 0, 0, n - 1,Findidx);
    post(root);
    return 0;
}

```

#### 4、程序运行与测试

**运行结果：**

##### 测试样例一

- 标准输入：

68877

67239 20038 21619 33452 20932 66131 1021 32222 63405 9834 64593 34031 43218 28305 19075 23644 15808 445

(数据太长，仅放部分数据)

- 实际输出：

15808 44541 23644 49998 6777 22077 19075 42674 1246 11638 47138 13825 4471 19201 28305 66494 67699 2350

(数据太长，仅放部分数据)

- 期望输出：

15808 44541 23644 49998 6777 22077 19075 42674 1246 11638 47138 13825 4471 19201 28305 66494 67699 2350

(数据太长，仅放部分数据)

## 测试样例二

- 标准输入:

58100

11074 41655 35671 7353 48672 10911 11017 26129 55903 33844 20134 12314 13733 33300 15555 3744 2165 3901  
(数据太长, 仅放部分数据)

- 实际输出:

3744 2165 15555 6222 844 39018 33300 24725 57603 15412 26294 19602 18059 40550 13733 35294 35884 16894  
(数据太长, 仅放部分数据)

- 期望输出:

3744 2165 15555 6222 844 39018 33300 24725 57603 15412 26294 19602 18059 40550 13733 35294 35884 16894  
(数据太长, 仅放部分数据)

## 5、实验总结与心得

- 本题我思考了很久, 最后才想到了实际上就是对中序表达式进行二分处理。但是要维护好**左边界和右边界**, 并通过在前序表达式中得到的索引反推出中缀表达式中的索引, 并进行划分。