# 数据结构实验三说明

## 1 概览

本次实验包含两个独立的编程任务，分别重点考察**二叉树**和**图**这两种非线性数据结构的应用。学生需要完成两个核心程序：

1. **二叉树应用**：实现基本二叉树操作、表达式二叉树和哈夫曼树，涵盖二叉树的遍历、构建和特殊应用。
2. **图的遍历与最短路径**：实现图的邻接表存储、深度优先遍历（DFS）以及Dijkstra最短路径算法。

## 2 预备知识

### 2.1 二叉树及其应用

二叉树是一种重要的非线性数据结构，每个节点最多有两个子节点。本次实验将涉及：

- **二叉树遍历**：前序、中序、后序和层次遍历的实现。
- **二叉树重建**：根据不同遍历序列重建二叉树。
- **表达式二叉树**：用二叉树表示和计算数学表达式。
- **哈夫曼树**：构建最优二叉树用于数据压缩编码。

### 2.2 图的存储与遍历

图是由顶点和边组成的数据结构。关键知识点包括：

- **邻接表存储**：使用链表存储图的邻接关系，适合稀疏图。
- **深度优先遍历（DFS）**：采用递归或栈的方式遍历图的所有顶点。
- **最短路径算法**：Dijkstra算法用于求解单源最短路径问题。

### 2.3 C++ 高级特性

本实验将使用以下C++特性：

- **继承与多态**：利用虚函数实现不同类型二叉树的统一接口。
- **STL容器**：使用vector、queue、stack、priority_queue等标准容器。
- **智能指针管理**：合理管理动态内存，避免内存泄漏。

# 3 实验框架

实验框架提供了完整的项目结构和基本的类定义。学生需要在指定位置补全核心算法实现。

项目主要目录结构如下：

```
Lab3/
├── BinaryTree/          // 二叉树相关实现
│   ├── include/         // 头文件目录
│   │   ├── BinaryTree.h          // 基础二叉树类声明
│   │   ├── ExpressionBinaryTree.h// 表达式二叉树类声明
│   │   └── HuffmanTree.h         // 哈夫曼树类声明
│   ├── src/             // 源代码目录
│   │   ├── BinaryTree.cpp        // 基础二叉树实现 (需实现)
│   │   ├── ExpressionBinaryTree.cpp // 表达式树实现 (需实现)
│   │   ├── HuffmanTree.cpp       // 哈夫曼树实现 (需实现)
│   │   └── main.cpp      // 主程序（已实现）
│   └── CMakeLists.txt    // 构建配置文件
├── graph/               // 图相关实现
│   ├── include/         // 头文件目录
│   │   ├── graph.h      // 图类声明
│   │   └── dijkstra.h   // Dijkstra算法类声明
│   ├── src/             // 源代码目录
│   │   ├── graph.cpp    // 图实现 (需实现)
│   │   ├── dijkstra.cpp // Dijkstra算法实现 (需实现)
│   │   └── main.cpp     // 主程序（已实现）
│   └── CMakeLists.txt    // 构建配置文件
└── CMakeLists.txt       // 主项目构建配置文件
```

# 4 实验任务

## 4.1 二叉树基础操作 ( `BinaryTree/src/BinaryTree.cpp` )

本部分要求完成基础二叉树的各种遍历和重建算法。

1. **遍历算法实现**
   - `void PreOrderTraverse(BinaryTreeNode* root)`：前序遍历实现
   - `void InOrderTraverse(BinaryTreeNode* root)`：中序遍历实现
   - `void PostOrderTraverse(BinaryTreeNode* root)`：后序遍历实现
   - `void LevelOrderTraverse(BinaryTreeNode* root)`：层次遍历实现（需使用队列）

2. **二叉树重建算法**
   - `buildTreeFromInorderPreorder()`：根据中序和前序序列重建二叉树
   - `buildTreeFromInorderPostorder()`：根据中序和后序序列重建二叉树
   - 需要实现对应的辅助函数，使用递归分治的思想

## 4.2 表达式二叉树（`BinaryTree/src/ExpressionBinaryTree.cpp`）

本部分要求实现表达式二叉树的构建和求值功能。

1. **`ExpressionBinaryTreeNode* buildFromPostfix(const string& postfix)` 函数**
   从后缀表达式构建表达式二叉树。算法思路：
   - 使用栈存储操作数和子树
   - 遇到操作数直接入栈
   - 遇到操作符时，弹出两个操作数作为子树，构建新的子树根节点
2. **`ExpressionBinaryTreeNode* buildFromPrefix(const string& prefix)` 函数**
   从前缀表达式构建表达式二叉树。算法思路：
   - 从右向左扫描前缀表达式
   - 使用类似后缀表达式的栈操作方法
3. **`int evaluate()` 函数**
   计算表达式二叉树的值，使用递归的后序遍历思想。

## 4.3 哈夫曼树（`BinaryTree/src/HuffmanTree.cpp`）

本部分要求实现哈夫曼树的构建和编码输出。

1. **`HuffmanTreeNode* buildHuffmanTree(const unordered_map<char, int>& freqMap)` 函数**
   构建哈夫曼树的核心算法：
   - 使用优先队列（最小堆）存储节点
   - 每次取出频率最小的两个节点合并
   - 重复直到队列中只剩一个节点（根节点）
2. **`void printHuffmanCodes(HuffmanTreeNode* root, string code)` 函数**
   输出哈夫曼编码，使用前序遍历的思想递归生成编码。

## 4.4 图的基础操作（`graph/src/graph.cpp`）

本部分要求实现图的邻接表存储和深度优先遍历。

1. **`void init(int u[], int v[], int w[])` 函数**
   初始化图的邻接表存储结构：
   - 根据边的信息构建邻接表

- 处理边的权重和邻接关系

2. **`void dfs(int s)` 函数**
   实现图的深度优先遍历：
   - 使用递归或显式栈实现
   - 标记已访问的顶点避免重复访问
   - 按访问顺序输出顶点编号

## 4.5 Dijkstra最短路径算法（`graph/src/dijkstra.cpp`）

本部分要求实现Dijkstra单源最短路径算法。

1. **`void ssp(int s)` 函数**
   实现Dijkstra算法的核心逻辑：
   - 初始化距离数组和访问标记
   - 重复选择未访问的最近顶点
   - 更新相邻顶点的最短距离
   - 输出最终的最短路径结果
2. **`void print(int u)` 函数**
   输出路径的辅助函数，递归输出从源点到目标点的路径。

# 5 实验运行

使用 VSCode 打开到 `Lab3` 的目录（应包含主 `CMakeLists.txt` 文件）：

1. 使用 VSCode 的 CMake 插件进行配置和构建
2. 可以选择构建整个项目或单独的子项目：
   - `BinaryTreeDemo`：二叉树演示程序
   - `GraphDemo`：图算法演示程序

# 6 实验分数安排与检查

## 6.1 分数安排

- 完成 `BinaryTree/src/BinaryTree.cpp` 的实现：**2分**
- 完成 `BinaryTree/src/ExpressionBinaryTree.cpp` 的实现：**2分**
- 完成 `BinaryTree/src/HuffmanTree.cpp` 的实现：**2分**
- 完成 `graph/src/graph.cpp` 的实现：**2分**
- 完成 `graph/src/dijkstra.cpp` 的实现：**1分**

- 现场检查提问：**1分**
- **总分：10分**

## 6.2 现场检查

- 实验检查截止日期：2025年11月29日
- 检查要求：
  - 现场演示程序运行结果
  - 解释关键算法的实现思路
  - 回答算法复杂度和优化相关问题
- 逾期未检查者，本次实验成绩记为0分

# 7 附录

## 7.1 算法复杂度参考

- **二叉树遍历**：时间复杂度 O(n)，空间复杂度 O(h)，其中 n 为节点数，h 为树高
- **哈夫曼树构建**：时间复杂度 O(n log n)，其中 n 为字符种类数
- **图的 DFS 遍历**：时间复杂度 O(V + E)，其中 V 为顶点数，E 为边数
- **Dijkstra 算法**：时间复杂度 O($V^2$) 或 O((V + E) log V)（使用优先队列优化）

## 7.2 实现注意事项

1. **内存管理**：注意动态分配内存的释放，避免内存泄漏
2. **边界处理**：处理空树、空图等特殊情况
3. **输入验证**：对输入数据进行合法性检查
4. **代码风格**：保持良好的代码风格和注释习惯