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 30 日	完成日期	2025 年 11 月 30 日

第一题：实验题目

给定 K 个不同的字符和它们的频率，建立一棵哈夫曼树对字符进行编码，合并节点时频率小的在右边，频率相同时字符小的在右边。

按照后序遍历的顺序输出字符。

输入包含两部分，第一部分一个数字 K 表示有 K 个字符，第二部分有 K 行，每行两个数表示字符和它的频率

实验目的

1. 掌握哈夫曼树的构建原理和实现方法
2. 理解贪心算法在哈夫曼编码中的应用
3. 学习使用优先队列（最小堆）来高效构建哈夫曼树
4. 掌握二叉树的后序遍历算法

算法设计

0.1 数据结构设计

- **节点结构体 leave**：包含频率 (fre)、标志位 (flag)、字符 (c)、左右子节点指针 (l, r)
- **优先队列**：使用 STL 的 priority_queue，自定义比较函数 cmp 实现最小堆

0.2 算法流程

1. **初始化**：读取 K 个字符及其频率，为每个字符创建叶子节点并入队
2. **建树过程**：
   - 当队列不为空时，弹出两个频率最小的节点
   - 创建新节点，频率为两节点频率之和，字符设为'Z'
   - 将新节点作为父节点，频率较小的节点作为右子树，较大的作为左子树
   - 将新节点重新入队
3. **后序遍历**：递归遍历哈夫曼树，按照左-右-根的顺序输出叶子节点字符

0.3 复杂度分析

- **时间复杂度**：$O(K \log K)$
  - 建堆操作：$O(K)$
  - 每次出队入队操作：$O(\log K)$
  - 共需 K-1 次合并操作：$O(K \log K)$
  - 后序遍历：$O(K)$
  - 总复杂度：$O(K \log K)$
- **空间复杂度**：$O(K)$
  - 存储 K 个节点：$O(K)$
  - 优先队列空间：$O(K)$

程序运行与测试

Listing 1: 哈夫曼树编码实现代码

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

struct leave {
    int fre;
    int flag = 0;
    char c;
    leave* l;
    leave* r;
};

struct cmp {
    bool operator()(leave* l1, leave* l2) {
        if (l1 -> fre < l2 -> fre) return false;
        else if (l1 -> fre > l2 -> fre) return true;
```

```
        else {
            return l1 -> c > l2 -> c;
        }
    }
};

void post(leave* root) {
    if (root == nullptr) return;
    post(root -> l);
    post(root -> r);
    if (root -> flag == 1) cout << root -> c << endl;
}

int N;
priority_queue<leave*, vector<leave*>, cmp> q;

int main () {
    cin >> N;
    int fre;
    char c;
    for (int i = 0; i < N; i++) {
        cin >> c >> fre;
        leave* tmp = new leave{fre, 1, c, nullptr, nullptr};
        q.push(tmp);
    }

    leave* root = nullptr;
    while (!q.empty()) {
        leave* l1 = q.top();
        q.pop();
        if (q.empty()) {
            root = l1;
        } else {
            leave* l2 = q.top();
            q.pop();
            leave* n = new leave{l1 -> fre + l2 -> fre, 0, 'Z', l2, l1};
            root = n;
            q.push(n);
        }
    }

    post(root);
    return 0;
}
```

实验总结

- 成功实现了哈夫曼树的构建和编码功能

- 掌握了使用优先队列优化哈夫曼树构建的方法
- 理解了贪心算法在最优编码问题中的应用
- 通过后序遍历验证了树的正确构建
- 算法的时间复杂度为 $O(K \log K)$，空间复杂度为 $O(K)$，具有较高的效率

第二题：实验题目

给定一个无向图，判断该图是否构成树。

实验目的

1. 掌握树的基本性质和判定条件
2. 学习使用并查集（Union-Find）算法检测图中是否存在环
3. 理解图的连通性判断方法
4. 掌握多测试样例的处理技巧

算法设计

0.4 树的性质

一个无向图是树当且仅当满足以下两个条件：

1. 图中不存在环
2. 图是连通的（只有一个连通分量）

0.5 算法流程

1. **初始化**：读取测试样例个数，对每个测试样例：
   - 读取结点数 n 和边数 m
   - 初始化并查集数组，所有结点父节点设为-1
2. **处理边**：对每条边 (a,b)：
   - 如果 a 和 b 已经在同一集合中，说明存在环，设置标志位 flag=1
   - 否则，将 a 和 b 所在集合合并
3. **连通性检查**：统计连通分量个数（根节点个数）
4. **结果判断**：
   - 如果存在环（flag=1），输出”NO”
   - 如果连通分量个数大于 1，输出”NO”
   - 否则，输出”YES”

0.6 并查集操作

- **find 操作**：带路径压缩的查找根节点
- **unite 操作**：合并两个集合

0.7 复杂度分析

- **时间复杂度**：$O(m \cdot \alpha(n))$
  - 其中 $\alpha(n)$ 是反阿克曼函数，增长极其缓慢
  - 对于每个测试样例，处理 m 条边的时间为 $O(m \cdot \alpha(n))$
  - 统计连通分量：$O(n \cdot \alpha(n))$
  - 总复杂度：$O((m+n) \cdot \alpha(n))$
- **空间复杂度**：$O(n)$
  - 存储并查集数组：$O(n)$

程序运行与测试

Listing 2: 判断无向图是否为树实现代码

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

int find(int* arr, int x) {
    if (arr[x] == -1) return x;
    arr[x] = find(arr, arr[x]);
    return arr[x];
}

void unite(int* arr, int x, int y) {
    x = find(arr, x);
    y = find(arr, y);
    if (x == y) return;
    arr[y] = x;
}

int main () {
    int N;
    cin >> N;
    for (int j = 0; j < N; j++) {
        int n, m;
        cin >> n >> m;

        int* arr = new int[n + 1];
        for (int i = 0; i < n + 1; i++) {
            arr[i] = -1;
```

```
        }

        int flag = 0;
        for (int i = 0; i < m; i++) {
            int a, b;
            cin >> a >> b;
            if (find(arr, a) == find(arr, b)) {
                flag = 1;
            }
            unite(arr, a, b);
        }

        if (flag == 1) {
            cout << "NO" << endl;
            delete[] arr;
            continue;
        }

        int rootcount = 0;
        for (int i = 1; i <= n; i++) {
            if (find(arr, i) == i) {
                rootcount++;
            }
        }

        if (rootcount > 1) {
            cout << "NO" << endl;
        } else {
            cout << "YES" << endl;
        }

        delete[] arr;
    }
    return 0;
}
```

实验总结

- 成功实现了判断无向图是否为树的算法
- 掌握了并查集数据结构的应用，包括 find 和 unite 操作
- 理解了树的两个关键性质：无环性和连通性
- 学习了处理多测试样例时的注意事项，特别是内存管理
- 算法的时间复杂度为 $O(m \cdot \alpha(n))$，空间复杂度为 $O(n)$，效率较高
- 通过路径压缩优化，提高了并查集的查找效率

第三题：实验题目

In computer science and information theory, a Huffman code is an optimal prefix code algorithm.

In this exercise, please use Huffman coding to encode a given data. You should output the number of bits, denoted as B(T), to encode the data: B(T)= f(c)dT(c), where f(c) is the frequency of character c, and dT(c) is be the depth of character c's leaf in the tree T.

实验目的

1. 理解哈夫曼编码的原理和最优前缀码的性质
2. 掌握使用优先队列构建哈夫曼树的方法
3. 实现哈夫曼编码并计算总编码位数 $B(T) = \sum f(c)d_T(c)$
4. 学习二叉树深度计算和遍历算法

算法设计

0.8 哈夫曼编码原理

哈夫曼编码是一种最优前缀码，通过构建哈夫曼树来实现数据压缩。编码长度与字符频率成反比，高频字符使用短编码，低频字符使用长编码。

0.9 数据结构设计

- **节点结构体 node**：包含频率 (fre)、深度 (depth)、左右子节点指针 (l, r)
- **优先队列**：使用 STL 的 priority_queue，自定义比较函数 cmp 实现最小堆

0.10 算法流程

1. **初始化**：读取 K 个字符及其频率，为每个字符创建叶子节点
2. **建树过程**：
    - 使用最小堆存储所有节点
    - 每次弹出两个频率最小的节点
    - 创建新节点，频率为两节点频率之和
    - 将新节点作为父节点，重新入队
    - 重复直到堆中只剩一个节点（根节点）
3. **计算深度**：递归遍历哈夫曼树，计算每个叶子节点的深度
4. **计算总位数**：$B(T) = \sum_{i=1}^{N} f(c_i) \times d_T(c_i)$

0.11 复杂度分析

- **时间复杂度**：$O(N \log N)$
  - 建堆操作：$O(N)$
  - 每次出队入队操作：$O(\log N)$
  - 共需 N-1 次合并操作：$O(N \log N)$
  - 深度计算遍历：$O(N)$
  - 总复杂度：$O(N \log N)$
- **空间复杂度**：$O(N)$
  - 存储 N 个节点：$O(N)$
  - 优先队列空间：$O(N)$
  - 节点指针数组：$O(N)$

程序运行与测试

Listing 3: 哈夫曼编码数据压缩实现代码

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

struct node {
    int fre;
    int depth = 0;
    node* l;
    node* r;
};

struct cmp {
    bool operator()(node* l1, node* l2) const {
        return l1 -> fre > l2 -> fre;
    }
};

void caldepth(node* root) {
    if (root == nullptr) return;
    if (root -> l != nullptr) root -> l -> depth = root -> depth + 1;
    if (root -> r != nullptr) root -> r -> depth = root -> depth + 1;
    caldepth(root -> l);
    caldepth(root -> r);
}

priority_queue<node*, vector<node*>, cmp> q;

int main () {
```

```
    int N, fre;
    char val;
    cin >> N;
    node** arr = new node*[N];
    for (int i = 0; i < N; i++) {
        cin >> val >> fre;
        node* p = new node{fre, 0, nullptr, nullptr};
        arr[i] = p;
        q.push(p);
    }

    node* root = nullptr;
    while (q.size() > 1) {
        node* l1 = q.top();
        q.pop();
        node* l2 = q.top();
        q.pop();
        node* n = new node{l1 -> fre + l2 -> fre, 0};
        n -> l = l1;
        n -> r = l2;
        root = n;
        q.push(n);
    }

    if (!q.empty()) root = q.top();
    caldepth(root);

    int result = 0;
    for (int i = 0; i < N; i++) {
        result += arr[i] -> fre * arr[i] -> depth;
    }
    cout << result;

    delete[] arr;
    return 0;
}
```

实验总结

- 成功实现了哈夫曼编码算法，能够计算数据压缩后的总编码位数
- 掌握了使用优先队列构建哈夫曼树的高效方法
- 理解了哈夫曼编码的最优前缀码性质及其在数据压缩中的应用
- 通过深度计算实现了编码位数的准确统计
- 算法的时间复杂度为 $O(N \log N)$，空间复杂度为 $O(N)$，具有较好的效率
- 实验加深了对贪心算法和二叉树遍历的理解

附录、提交文件清单

第一题

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

struct leave {
    int fre;
    int flag = 0;
    char c;
    leave* l;
    leave* r;
};

struct cmp {
    bool operator()(leave* l1, leave* l2) {
        if (l1 -> fre < l2 -> fre) return false;
        else if (l1 -> fre > l2 -> fre) return true;
        else {
            return l1 -> c > l2 -> c;
        }
    }
};

void post(leave* root) {
    if (root == nullptr) return;
    post(root -> l);
    post(root -> r);
    if (root -> flag == 1) cout << root -> c << endl;
}

int N;
priority_queue<leave*, vector<leave*>, cmp> q;

int main () {
    cin >> N;
    int fre;
    char c;
    for (int i = 0; i < N; i++) {
        cin >> c >> fre;
        leave* tmp = new leave{fre, 1, c, nullptr, nullptr};
        q.push(tmp);
    }

    leave* root = nullptr;
```

```
    while (!q.empty()) {
        leave* l1 = q.top();
        q.pop();
        if (q.empty()) {
            root = l1;
        } else {
            leave* l2 = q.top();
            q.pop();
            leave* n = new leave{l1 -> fre + l2 -> fre, 0, 'Z', l2, l1};
            root = n;
            q.push(n);
        }
    }

    post(root);
    return 0;
}
```

第二题

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

int find(int* arr, int x) {
    if (arr[x] == -1) return x;
    arr[x] = find(arr, arr[x]);
    return arr[x];
}

void unite(int* arr, int x, int y) {
    x = find(arr, x);
    y = find(arr, y);
    if (x == y) return;
    arr[y] = x;
}

int main () {
    int N;
    cin >> N;
    for (int j = 0; j < N; j++) {
        int n, m;
        cin >> n >> m;

        int* arr = new int[n + 1];
        for (int i = 0; i < n + 1; i++) {
            arr[i] = -1;
        }

        int flag = 0;
```

```
        for (int i = 0; i < m; i++) {
            int a, b;
            cin >> a >> b;
            if (find(arr, a) == find(arr, b)) {
                flag = 1;
            }
            unite(arr, a, b);
        }

        if (flag == 1) {
            cout << "NO" << endl;
            delete[] arr;
            continue;
        }

        int rootcount = 0;
        for (int i = 1; i <= n; i++) {
            if (find(arr, i) == i) {
                rootcount++;
            }
        }

        if (rootcount > 1) {
            cout << "NO" << endl;
        } else {
            cout << "YES" << endl;
        }

        delete[] arr;
    }
    return 0;
}
```

第三題

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

struct node {
    int fre;
    int depth = 0;
    node* l;
    node* r;
};

struct cmp {
    bool operator()(node* l1, node* l2) const {
        return l1 -> fre > l2 -> fre;
```

```
    }
};

void caldepth(node* root) {
    if (root == nullptr) return;
    if (root -> l != nullptr) root -> l -> depth = root -> depth + 1;
    if (root -> r != nullptr) root -> r -> depth = root -> depth + 1;
    caldepth(root -> l);
    caldepth(root -> r);
}

priority_queue<node*, vector<node*>, cmp> q;

int main () {
    int N, fre;
    char val;
    cin >> N;
    node** arr = new node*[N];
    for (int i = 0; i < N; i++) {
        cin >> val >> fre;
        node* p = new node{fre, 0, nullptr, nullptr};
        arr[i] = p;
        q.push(p);
    }

    node* root = nullptr;
    while (q.size() > 1) {
        node* l1 = q.top();
        q.pop();
        node* l2 = q.top();
        q.pop();
        node* n = new node{l1 -> fre + l2 -> fre, 0};
        n -> l = l1;
        n -> r = l2;
        root = n;
        q.push(n);
    }

    if (!q.empty()) root = q.top();
    caldepth(root);

    int result = 0;
    for (int i = 0; i < N; i++) {
        result += arr[i] -> fre * arr[i] -> depth;
    }
    cout << result;

    delete[] arr;
    return 0;
}
```