



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	2025.11.26	结束日期	2025.11.30

第一题

1、实验题目

☆ z12-Huffman coding

题目描述

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:

$$B(T) = \sum f(c)d_T(c)$$

where $f(c)$ is the frequency of character c , and $d_T(c)$ is the depth of character c 's leaf in the tree T .

要求使用堆或优先队列来完成，会检查代码和报告（报告中请附上代码）。

输入描述

The first line is the number of characters n .

The following n lines, each line gives the character c and its frequency $f(c)$.

输出描述

Output a single number $B(T)$.

输入样例

```
5
0 5
1 4
2 6
3 2
4 3
```

输出样例

```
45
```

2、实验目的

- 理解 Huffman 编码的基本原理。
 - 掌握通过“贪心 + 最小堆”构造 Huffman 树的思想。
 - 熟悉优先队列（priority_queue）的使用方式。
 - 学会通过重复合并最小权值的方法计算 Huffman 编码总长度。
-

3、算法设计

设计思路

1. 读入所有字符与其频率，将频率值放入**最小堆**。
2. 重复执行以下操作直到堆中只剩一个元素：
 - 取出最小的两个数 a 、 b ；
 - 将它们合并为新结点 $c = a + b$ ；
 - 将 c 加入结果 $result$ ，同时将 c 再次放入堆中；
3. $result$ 即为最终的 Huffman 编码总长度 $B(T)$ 。

关键：因为对于一个叶子节点，它对 $B(T)$ 的贡献为**当前频率值 * 路径深度**。但是对于路径上的**非叶子节点**，每一个节点代表当前树下的**所有叶子节点的总和**。所以把这些非叶子节点的值求和，就能加多次底下的叶子节点，**实现 * 路径深度的效果**。

流程图

```
Start
|
|--> Read n
|
|--> For i = 1..n:
|       Read char, freq
|       Push freq into min-heap
|
|--> result = 0
|
|--> While heap.size > 1:
|       a = pop smallest
|       b = pop second smallest
|       c = a + b
|       result += c
```

```
|         push c into heap
|
|--> Print result
|
End
```

复杂度分析

- 入堆 n 次: $O(n \log n)$
- 合并 $n-1$ 次, 每次 pop 两次 + push 一次: $O(n \log n)$
- 总时间复杂度: $O(n \log n)$
- 空间复杂度: $O(n)$

细节注意

- 需要把所有非叶子节点的数值加起来。
- 优先队列中的 `greater<int>`: 如果 $a > b$, 返回 `true`. 但是在优先队列中, 如果返回的 `true`, 则把 a 放在 b 后面。所以实际上小的数值优先级更高。

具体实现

```
//z12-Huffman coding
#include<iostream>
#include<queue>
#include<vector>
using namespace std;
int main(){
    int n;
    cin>>n;
    priority_queue<int,vector<int>,greater<int>> que;
```

```
for(int i=0;i<n;i++){
    int frq;
    char chr;
    cin>>chr>>frq;
    que.push(frq);
}
int result=0;
while(que.size()>1){
    int a=que.top();
    que.pop();
    int b=que.top();
    que.pop();
    int c=a+b;
    result+=c;
    que.push(c);
}
cout<<result<<endl;
}
```

4、程序运行与测试

测试样例一

- 标准输入：

```
2
a 1
b 100
```

- 实际输出：

```
101
```

- 期望输出：

101

测试样例二

- 标准输入：

```
10
5 1294
9 1525
6 2260
4 2561
2 4442
3 5960
7 6878
8 8865
1 11610
0 70784
```

- 实际输出：

```
246790
```

- 期望输出：

```
246790
```

5、实验总结与心得

- 理解了 Huffman 树的本质是**不断合并最小权值**，这是典型的贪心结构。
- 使用**优先队列**能极大简化 Huffman 过程。
- 通过本题进一步熟悉了**最小堆**的用法与复杂度特性。

第二题

1、实验题目

☆ z12-判断一个图是否构成树

问题

给定一个无向图，判断该图是否为树。

输入

输入有若干个测试样例。第一行是测试样例个数。
每个测试样例的第一行是结点数 n (编号 $1..n$) 。
第二行为边数 m ，接下来有 m 个结点对。

输出

如果一个图是树，则打印 YES，否则打印 NO。

输入样例

```
3  
  
3  
2  
1 2  
2 3  
  
3  
3  
1 2  
2 3  
1 3
```

```
3
1
2 3
```

输出样例

```
YES
NO
NO
```

2、实验目的

- 理解树在无向图中的定义：**连通且无环、边数为 $n-1$** 。
- 学习使用并查集（Union-Find）检测环。
- 掌握对多测试样例的输入处理方式。

3、算法设计

设计思路

1. 对每个测试样例，读取 n 和 m 。
 2. 初始化并查集 $root[i] = i$ 。
 3. 对每条边 (u, v) ：
 - $find(u)$ 与 $find(v)$ ，若相等则说明出现了环。
 - 若不相等则合并两个集合。
 4. 最后判断：
 - 若无环且边数为 $n-1$ ，则是树。
 - 否则不是树。
-

流程图

```
Start
|
|--> Read T
|
|--> Loop T times:
|       Read n, m
|       初始化 root[i] = i
|       hascycle = 0
|
|       For each edge (u, v):
|           ru = find(u)
|           rv = find(v)
|           If ru == rv:
|               hascycle = 1
|           Else:
|               root[ru] = rv
|
|       If (not hascycle) and (m == n - 1):
|           Print YES
|       Else:
|           Print NO
|
End
```

复杂度分析

- 并查集的路径压缩使得 find 几乎为 $O(1)$
- **m 条边合并**: $O(m \alpha(n))$
- **整体时间复杂度**: $O(m \alpha(n))$
- **空间复杂度**: $O(n)$

细节注意

- 实际上也可以用搜索的思路：随机挑选节点当作根节点，**进行dfs或者bfs**。使用一个数组标记是否被查询过。
 - **dfs**: 如果当前节点的所有节点都被查询过，则查询当前节点的兄弟节点。否则继续往下。直到所有**与根节点联通的节点**都被探查过。如果还**存在某些节点未被探查，或者边数不是 $n-1$** ，就不是树。
 - **bfs**类似。
-

具体实现

```
//z12-判断一个图是否构成树
#include <iostream>
#include <vector>
using namespace std;
int find(vector<int> &root, int i)
{
    return root[i] == i ? i : root[i] = find(root, root[i]);
}
int main()
{
    int num;
    cin >> num;
    while (num-->0)
    {
        int n, m;
        cin >> n >> m;
        vector<int> root(n + 1);
        int hascycle = 0;
        for (int i = 0; i < n + 1; i++)
        {
            root[i] = i;
        }
    }
}
```

```
for (int i = 0; i < m; i++)
{
    int u, v;
    cin >> u >> v;
    int ru = find(root, u);
    int rv = find(root, v);
    if (ru == rv)
    {
        hascycle = 1;
    }
    else
    {
        root[ru] = rv;
    }
}
```

```
if (!hascycle && m == n - 1)
{
    // 根据树的性质，一共有n个节点，就会有n-1条边
    // 如果符合条件，且无环，则是树
    cout << "YES" << endl;
}
else
{
    cout << "NO" << endl;
}
}
return 0;
}
```

4、程序运行与测试

测试样例一

- 标准输入：

4

4

1

3 1

9

8

2 1

3 1

5 1

5 2

6 2

6 3

6 5

7 1

8

7

2 1

4 2

4 3

5 2

5 3

6 1

6 2

8

7

3 2

4 3

6 3

6 5

7 1

7 2

7 3

- 实际输出：

```
NO
NO
NO
NO
```

- 期望输出：

```
NO
NO
NO
NO
```

测试样例二

- 标准输入：

```
4

10
5
1 3
1 5
1 7
1 9
2 3

10
10
1 2
1 3
1 4
1 5
1 6
1 7
```

```
1 8
1 9
1 10
2 3

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

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

- 实际输出：

```
NO
NO
NO
YES
```

- 期望输出：

NO
NO
NO
YES

5、实验总结与心得

- 本题再次加深了对并查集的理解，学会用它判断**“是否成树”**。
 - 掌握了**无向图判断树结构的两个关键条件**：无环 + 边数为 $n-1$ 。
 - 进一步熟悉了 find 的路径压缩技巧。
-

第三题

1、实验题目

☆ z12-哈夫曼树编码

题目描述

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。

按后序遍历输出字符。

输入描述

第一行为数字 k ，表示有 k 个字符；
接下来 k 行，每行一个字符和它的频率。

样例输入

```
7
A 5
F 5
C 7
G 13
E 34
B 24
D 17
```

样例输出

```
E
D
G
B
F
A
C
```

2、实验目的

- 掌握哈夫曼树的构建过程。
- 学习自定义比较器（cmp）控制结点合并的优先级。
- 熟悉二叉树的后序遍历。
- 理解实际构造一棵 Huffman 树与计算编码长度之间的差异。

3、算法设计

设计思路

1. 将所有字符与频率封装为 node 对象，并放入优先队列。
2. 比较器 cmp 按以下规则排序：

- 频率小的优先
 - 若频率相等，字符 ASCII 小的放右边，因此队列中“字符大的优先弹出”
3. 不断从队列中弹出两个结点 left、right:
 - 建立新结点 root，频率为二者之和，**字符取较大者**
 - root->left = left
 - root->right = right
 - 再将 root 压回队列
 4. 队列中剩下的结点即为根结点
 5. 最后对树进行**后序遍历**输出所有叶子字符
-

流程图

```
Start
|
|--> Read n
|--> For i in 1..n:
|   Read chr, freq
|   new node and push to heap
|
|--> While heap.size > 1:
|   right = pop()
|   left  = pop()
|   root = new node(freq=left+right, chr=max)
|   root.left = left
|   root.right = right
|   push root
|
|--> root = heap.top()
|
|--> Postorder traverse:
|   if leaf: print char
|   else: recurse left then right
|
End
```

复杂度分析

- 建堆并插入 k 个节点: $O(k \log k)$
 - 构建哈夫曼树: 进行 $k-1$ 次合并, 每次操作 $\log k \rightarrow O(k \log k)$
 - 后序遍历: $O(k)$
 - 总时间复杂度: $O(k \log k)$
 - 总空间复杂度: $O(k)$
-

细节注意

- 字符取最大的原因是多次尝试后确定的。没有什么特殊原因。
-

具体实现

```
//z12-哈夫曼树编码
#include <iostream>
#include <vector>
#include <queue>
#include<algorithm>
using namespace std;
class node {
public:
    char chr;
    int frq;
    node *left;
    node *right;
    node() = default;
    node(int frq, char chr = 0)
        : frq(frq), chr(chr), left(nullptr), right(nullptr) {}
};
```

```

~node() {
    delete left;
    delete right;
}
int leaf() { return (!left) && (!right); }
};

struct cmp {
public:
    bool operator()(node *a, node *b) {
        if (a->frq == b->frq) {
            return a->chr > b->chr;
        }
        return a->frq > b->frq;
    }
};

node *build(priority_queue<node *, vector<node *>, cmp> &que) {
    while (que.size() > 1) {
        node *right = que.top();
        que.pop();
        node *left = que.top();
        que.pop();
        node *root = new node(left->frq + right->frq, max(left->chr, right->chr));

        root->left = left;
        root->right = right;
        que.push(root);
    }
    node *root = que.top();
    que.pop();
    return root;
}

void display(node *root) {
    if (root->leaf()) {
        cout << root->chr << endl;
        return;
    }
    display(root->left);
    display(root->right);
}

```

```
int main() {
    int n;
    cin >> n;
    priority_queue<node *, vector<node *>, cmp> que;
    for (int i = 0; i < n; i++) {
        char chr;
        int frq;
        cin >> chr >> frq;
        node *root = new node(frq, chr);
        que.push(root);
    }
    node *root = build(que);
    display(root);
    delete root;
    return 0;
}
```

4、程序运行与测试

测试样例

- 标准输入：

```
7
A 5
F 5
C 7
G 13
E 34
B 24
D 17
```

- 实际输出：

E
D
G
B
F
A
C

- 期望输出：

E
D
G
B
F
A
C

5、实验总结与心得

- 本题让我真正构造了一棵哈夫曼树，并体会到比较器对于构造结构的关键影响。
 - 明白了为何合并时“频率小的在右边”需要**反向定义比较器**。
 - **后序遍历的方式也更加熟练。**
 - 这题比起普通 Huffman 编码更贴近真实树结构，收获非常大。
-