上机报告-5

数算B 谢胡睿 2400014151

题目

1.题目背景

在日常生活中，我们经常会遇到流式媒体，即不能提前得知媒体的完整内容。此时，常规的 $Huffman$ 编码算法无能为力。我们需要采用动态 $Huffman$ 编码（自适应 $Huffman$ 编码）算法。

2.题目描述

对于给定的流式输入（即逐渐增长的输入）和在其中穿插的查询，输出截止到查询为止的 $Huffman$ 编码总长度。在初始时刻，流式输入包含一个表示输入结束的特殊字符，该字符不包括在新增的字符种类中，不会被输入，但需要对其编码。随后每次新增字符，都是在该特殊字符前插入。即可以认为不管什么时候该特殊字符出现且仅出现一次。$Huffman$ 编码指最小化总长的无二义可变长编码。在本题中，采用二叉 $Huffman$ 树生成，以二进制表示编码。不需要将 $Huffman$ 树嵌入输出的编码中，即不计入编码长度。

3.输入格式

第1行为2个正整数N和M，表示有N个不同的输入字符。接下来为M行，每行有两种可能，分别为：

1. 一个小于等于N的正整数，表示流式输入新增了一个由该正整数标识的字符。
2. 一个字符0，表示查询截止到该查询为止的 $Huffman$ 编码总长度。

4.输出格式

L行，其中L为输入中查询的数量。每行一个非负整数，表示截止到对应查询为止的流式输入的 $Huffman$ 编码的二进制总位数。

输入输出样例

输入

```
3 10
1
1
2
2
2
0
1
3
3
0
```

输出

```
9
18
```

数据范围和提示
对于80%的数据，$N \times log_2 N \times M \leq 10000000$
对于100%的数据，$log_2 N \times M \leq 10000000$
评测限时$1s$,无存储限制

Solution

总体描述

本次实验实现了两种哈夫曼编码方案：传统哈夫曼编码和自适应哈夫曼编码（FGK算法）。传统哈夫曼编码基于已知的符号频率静态构建编码树，而FGK算法则能够动态调整树结构以适应数据流的变化。

方案一：传统哈夫曼编码

设计思路

传统哈夫曼编码基本步骤如下：

1. 收集所有符号的频率
2. 使用优先队列，每次合并频率最小的两个节点
3. 构建哈夫曼树，并计算总编码长度

核心代码：

```
long long HT(const vector<long long> &st, const set<long long> &idx) {
    long long total = 0;
    priority_queue<long long, vector<long long>, greater<long long>> pq;
    for(auto i : idx) {
        pq.push(st[i]); // 压入优先队列
    }
    // 构建哈夫曼树
    while(pq.size() > 1) {
        long long a = pq.toop();
        pq.pop();
        long long b = pq.top();
        pq.pop();
        pq.push(a + b);
        total += a + b; // 统计编码总长度
    }
    return total;
}
```

优缺点

优点：
实现简单，时间复杂度为$O(nlogn)$
缺点：
需要预先知道所有符号的频率分布
无法适应符号频率的动态变化

方案二：自适应哈夫曼编码（FGK算法）

设计思路

FGK的关键步骤包括：

1. 维护一棵FGK树，初始只有一个NYT节点
2. 每处理一个符号：

- 若符号首次出现，分裂NYT节点并添加新符号节点
- 若符号已存在，增加该符号节点的权重并调整树结构

3. 维护"兄弟属性"：相同权重的节点位于树的同一深度

关键代码：

```
void update(int s) {
    if(nodes[s]) { // 已存在符号
        updateExistSym(nodes[s]);
    } else { // 新符号
        addNewSym(s);
    }
}

void addNewSym(int s) {
    FGKNode* oldNYT = NYT;
    FGKNode* newNYT = new FGKNode(0, nextnumber--, -2);
    FGKNode* symNode = new FGKNode(1, nextnumber--, s);
    nodes[s] = symNode;
    // 连接新节点
    oldNYT->left = newNYT;
    oldNYT->right = symNode;
    newNYT->parent = symNode->parent = oldNYT;
    // 更新原NYT节点
    oldNYT->sym = -1;
    oldNYT->weight = 1;
    NYT = newNYT;
    updatePath(oldNYT->parent);
}

void updateExistSym(FGKNode* node) {
    node->weight++;
    while(node != root) {
        // 维护兄弟属性
        FGKNode* swapNode = node2swap(node);
        if(swapNode && swapNode != node->parent) {
            swapNodes(node, swapNode);
        }
        node = node->parent;
        node->weight++;
    }
}
```

优缺点

优点：
无需预先知道符号频率，可以处理在线数据流
缺点：
实现复杂度高，需要处理多种特殊情况
对于大量数据，维护树结构的开销较大

问题与挑战

1. 计算编码长度
   问题：如何准确计算动态变化的FGK树的编码总长度。
   解决方案：实现深度优先搜索遍历FGK树，根据节点权重和深度计算编码总长度。
2. 动态维护树结构
   问题：在FGK算法中，每次更新符号后都需要调整树结构，维持"兄弟属性"。
   解决方案：实现节点交换，通过节点编号标识创建顺序，确保相同权重下编号大的节点位于更高层。

总结

通过实现传统哈夫曼编码和自适应哈夫曼编码（FGK算法），我深入理解了两种编码策略的异同点。传统哈夫曼编码在符号频率已知的情况下能快速构建最优编码树，而FGK算法则在不预先知道频率分布的情况下，通过动态调整树结构实现自适应编码。
从理论角度看，FGK算法的核心在于保持"兄弟属性"，通过节点交换维护树结构。虽然实现复杂度高于传统哈夫曼编码，但其能够处理在线数据流的特性使其在某些实际场景中更具优势。

或者说，FGK树构建的虽然并非真正的 $huffman$ 树，但是因为它满足了"兄弟属性",所以在面对相同问题时，面对 $huffman$ 树有着相同的输出与特性