实验4-哈夫曼编码

问题分析

- 哈夫曼编码本质是个贪心的算法。将每个字符视作一个带权结点（子树）。
- 每次优先选择权值最小的两个子树，将二者合并，权值相加，权值小的作为右子结点，权值大的作左子结点，形成一个新的子树。再将这棵子树放回优先队列中，重复以上的操作。
- 定义一个结构体，便于在优先队列中排序，按照题目要求进行双关键字排序
  - 如果权值相等，则按它们加入的顺序排序（id小的充当左子树）
  - 如果权值不等，权值大的充当左子树

```
struct node{
    int val,id;//节点权值 节点编号
    vector<int> leaves;//子树中所有得叶子结点
    bool operator < (const node & b) const{//双关键字排序
        if(val==b.val) return id < b.id;
        return val > b.val;
    }
}tr[N];
```

- 合并的时候从优先队列头部取出两个节点进行合并，然后加入到优先队列中去，如果队列中只剩下一个节点则代表已经合并完成

```
while(!heap.empty()){
    node r = heap.top();heap.pop();
    if(heap.empty())
        break;
    node l = heap.top();heap.pop(); //取出权值最小的两个子树
    node t;
    t.val = l.val + r.val;//合并权值
    t.id = idx++;
    //...
    heap.push(t);
}
```

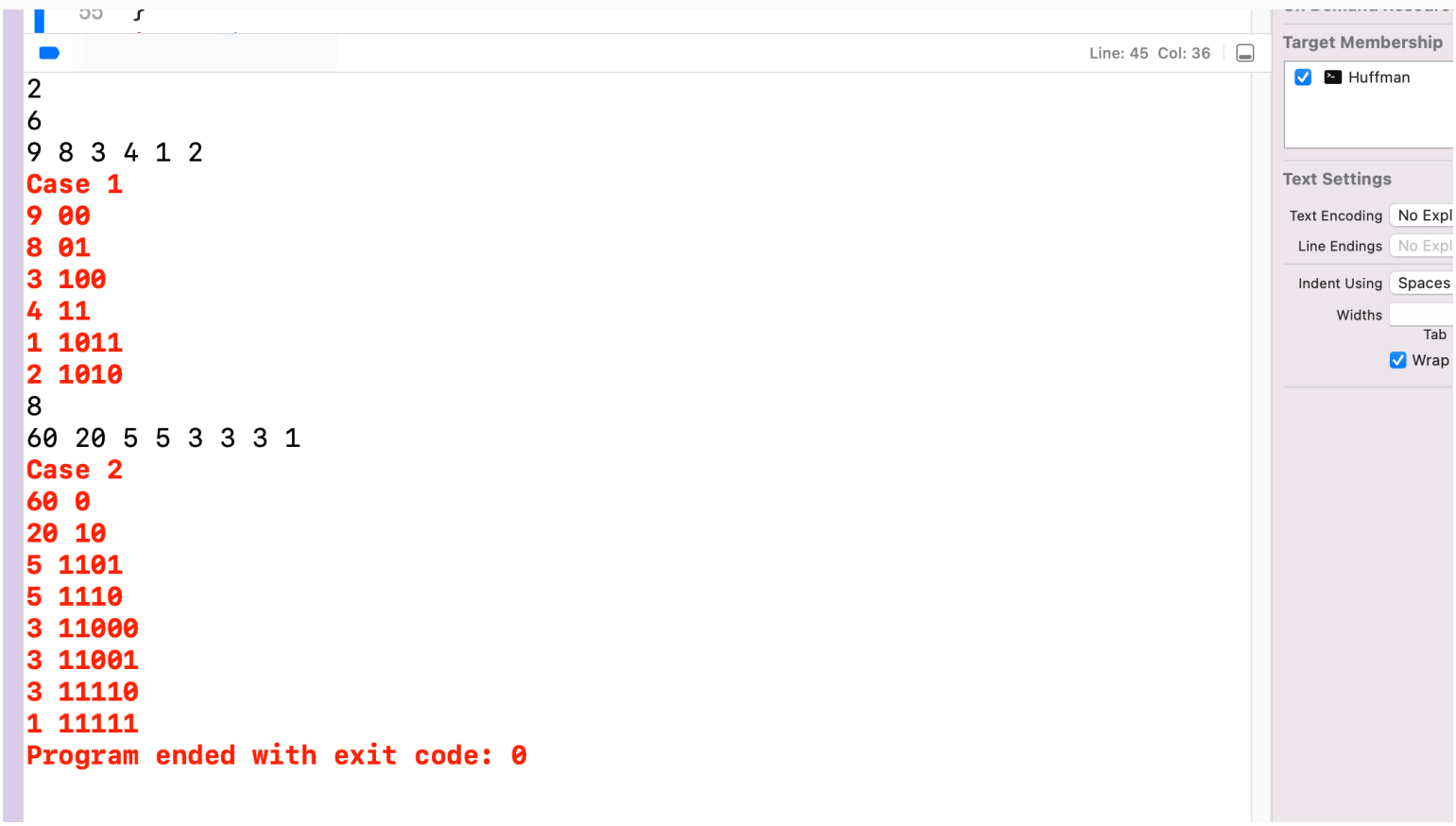
- 对于每个节点，我们存下该节点子树中所有的叶子结点以便合并节点的时候给它们增加编码
- 合并的时候，遍历子树的叶子结点
  - 对于右子树把它们的答案字符串尾部加1
  - 对于左子树把它们的答案字符串尾部加0

```
for(int i=0;i<(int)r.leaves.size();++i){//右子树
    int id = r.leaves[i];
    res[id].push_back('1');
    t.leaves.push_back(id);//把叶子结点加到新合成的节点中
}
for(int i=0;i < (int)l.leaves.size(); ++i){//左子树
    int id = l.leaves[i];
    res[id].push_back('0');
    t.leaves.push_back(id);
}
```

- 输出答案的时候，因为我们是从底向上给边编码，所以对答案reverse然后输出即可

```
for(int i=1;i<=n;++i){
    reverse(res[i].begin(),res[i].end());
    cout<<tr[i].val<<" "<<res[i]<<endl;
}
```

运行结果截图



设计调试中的问题

对于结构体内的小于号的重载，要加上const修饰，否则将会语法错误

```
bool operator < (const node & b) const{
    //...
}
```

相关函数库里的声明

```
    _LIBCPP_CONSTEXPR_AFTER_CXX11 _LIBCPP_INLINE_VISIBILITY
    bool operator()(const _Tp& __x, const _Tp& __y) const
        {return __x < __y;}
};
```

实验体会

本次实验比较简单，是对一个序列按照权值进行Huffman编码。我个人认为，Huffman编码非常的优雅，相较与等长的编码方式，Huffman编码能极大地提高效率。这次实验用代码去实现了其中的编码过程，再一次让我感受到了算法的魅力。Huffman编码的应用非常广泛，可用于数据压缩等领域。本次实验，让我对优先队列的使用更加熟练，温习了树结构的存储与应用，也让我认识到啦对于一些小的语法问题也要加以重视。

程序源码

```
#include<bits/stdc++.h>
using namespace std;
const int N = 110;
int cas=1;
struct node{
    int val,id;
    vector<int> leaves;
    bool operator < (const node & b) const{        //双关键字排序
        if(val==b.val) return id < b.id;
        return val > b.val;
    }
}tr[N];
void solve(){
    int n;
    cin>>n;
    vector<string> res(N);
    priority_queue<node> heap;
    for(int i=1,w;i<=n;++i){
        node tmp;
        cin>>w;
        tr[i].val = w;
        tmp.val = w;
        tmp.id = i;
        tmp.leaves.push_back(i);
        heap.push(tmp);                //初始将每个结点视作子树
    }
    int idx = n+1;
    while(!heap.empty()){
        node r = heap.top();
        heap.pop();
        if(heap.empty())
            break;
        node l = heap.top();
        heap.pop();       //取出权值最小的两个子树
        node t;
        t.val = r.val + l.val;           //合并权值
        t.id = idx++;
        for(int i=0;i<(int)r.leaves.size();++i){//右子树
```

```
            int id = r.leaves[i];
            res[id].push_back('1');
            t.leaves.push_back(id);//把叶子结点加到新合成的节点中
        }
        for(int i=0;i < (int)l.leaves.size(); ++i){//左子树
            int id = l.leaves[i];
            res[id].push_back('0');
            t.leaves.push_back(id);
        }
        heap.push(t);
    }
    cout<<"Case "<<cas++<<'\n';
    for(int i=1;i<=n;++i){
        reverse(res[i].begin(),res[i].end());
        cout<<tr[i].val<<" "<<res[i]<<'\n';
    }
}
int main()
{
    int T;
    cin>>T;
    while(T--){
        solve();
    }
    return 0;
}
```