树和二叉树的应用

哈夫曼树和哈夫曼编码

哈夫曼树

哈夫曼(Huffman)树，又称最优二叉树，是树结构应用之一。

基本概念和术语

1. 路径(Path)：从一个结点到另一个结点之间的分支序列，构成这两个结点之间的路径。
2. 路径长度(Path Length)：路径上的分支数目称为路径长度。例如，根结点到第L层结点路径长度为$L-1$。完全二叉树是路径长度最短的二叉树。
3. 结点的权值(Weight)：在实际应用中，人们常常给树的每个结点赋予一个具有实际意义的数，该数被称为该结点的权值。
4. 结点的带权路径长度：从根结点到某一结点的路径长度与该结点的权值的乘积，称为该结点的带权路径长度。
5. 树的带权路径长度(Weighted Path Length, WPL)：树中所有叶结点的带权路径长度之和。通常记为：$\mathrm{WLP}=\sum_{i=1}^{n}w_il_i$。其中$n$为叶结点数量，$w_i$为第$i$个叶结点的权值，$l_i$为根结点到第$i$个叶结点的路径长度。
6. 哈夫曼树(Huffman Tree)：由n个带权值的叶结点的构成的二叉树中，WPL最小的二叉树称为最优二叉树，也称为哈夫曼树。

哈夫曼树的特点

1. 有n个叶结点的哈夫曼树共有$2n-1$个结点。
2. 权值越大的叶结点，离根结点越近，权值越小的叶结点，离根结点越远。
3. 哈夫曼树是正则二叉树，只有度为0（叶结点）和度为2（分支）的结点，不存在度为1的结点。
4. 哈夫曼树的任意飞叶结点的左、右子树交换后仍然是哈夫曼树，哈夫曼树的形状不唯一，但是其WPL是相同的。

哈夫曼算法

构造最优二叉树的算法描述如下：

1. 根据给定的权值集合$\{w_1,w_2,\dots,w_n\}$，构造含有n棵二叉树的的集合（森林）$F=\{T_1,T_2,\dots,T_n\}$，其中每棵二叉树T_i只有根结点，其权值为w_i，左、右子树为空。
2. 在集合F中选取根结点的权值最小的两棵二叉树分别作为左、右子树构造一棵全新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和。
3. 从集合F中删除作为左、右子树的两棵二叉树，同时把新二叉树加入F中。
4. 重复上述两个步骤，直到集合F中只有一个二叉树为止，这棵二叉树即为哈夫曼树。

从哈夫曼算法可以看出，初始时，共有n棵二叉树，且均只有一个结点；在构造过程中选取两棵根结点权值最小的二叉树合并成一棵新的二叉树时，需要增加一个结点作为新二叉树的根结点。由于要进行$n-1$次合并才能使得初始的n棵二叉树合并为一棵二叉树，因此合并$n-1$次共产生$n-1$个结点，所以最终求得的哈夫曼树共有$2n-1$个结点。

哈夫曼树的类型定义及运算实现如下：

```
template <class T>
class huffmanTree{
    struct Node{
        T data; // 结点的数据域
        int weight; // 结点的权值
        int parent, left, right; // 双亲及左右孩子的下标
        Node(){
            weight = parent = left = right = 0;
        }
    };
    struct huffmanCode{
        T data;
        string code; // 保存data的哈夫曼编码
        huffmanCode(){
            code = "";
        }
    };
    Node* hfTree; // 顺序存储结构，保存哈夫曼树
    huffmanCode* hfCode; // 顺序存储结构，保存哈夫曼编码
    int size; // 叶结点个数
    void selectMin(int m, int& p); // 选出当前集合中的最小元素
public:
    huffmanTree(int initSize); // 构造函数
    ~huffmanTree(){delete [] hfTree; delete [] hfCode;}; // 析构函数
    void createHuffmanTree(const T* d, const double* w); // 创建哈夫曼树
    void huffmanEncoding(); // 获取哈夫曼编码
    void printHuffmanCode(); // 输出哈夫曼编码
};
```

上述哈夫曼树类型的定义如下：

1. 每个Node类型的元素保存的信息有：结点的数据域data，权值weight，双亲和左、右孩子的下标parent,left,right。因为size个叶结点的哈夫曼树共有2*size-1个结点，所以哈夫曼树可以用一个大小为2*size的数组hfTree来存储。数组下标为0的单元不用，根结点存放在下标为1的单元中，叶结点依次存放在下标为size到2*size-1的单元中。
2. parent域在构造哈夫曼树的过程中有两个作用。第一，在建立哈夫曼树的过程中，用于区分结点是否被使用过。parent=0表示该结点没有双亲，还没有被使用过。一旦结点被使用，就有了双亲，parent域的值就是指向双亲的结点（这里的指针实际上是数组的下标）。第二，在构造好哈夫曼树之后求哈夫曼编码时，需要从叶结点出发走一条从叶结点到根结点的路径，因此需要知道结点的双亲信息。
3. 每个huffmanCode类型的元素保存的信息有：结点的数据域data以及data对应的哈夫曼编码code。因为哈夫曼树有size个叶结点，所以哈夫曼编码可以用一个大小为size的huffmanCode类型的数组hfCode来存储。编码前，code初始化为空串。

构造函数：

```
template <class T>
huffmanTree<T>::huffmanTree(int initSize){
    size = initSize;
    hfTree = new Node[2 * size]; // 哈夫曼树的结点数为2*size-1
    hfCode = new huffmanCode[size]; // 哈夫曼编码的个数为size
}
```

根据叶结点数据数组v及其权值数组w创建哈夫曼树：

```
template <class T>
void huffmanTree<T>::createHuffmanTree(const T* d, const double* w){
    int i, min1, min2; // 最小树、次最小树的下标
    for(i = size; i < 2 * size; ++i){ // 给size个叶结点赋值
        hfTree[i].data = d[i - size];
        hfTree[i].weight = w[i - size];
    }
    for (i = size - 1; i > 0; --i){ // 合并产生size-1个新结点
        // 选出parent的值为0且权值最小的两棵子树min1、min2作为结点i的左右孩子
        selectMin(i + 1, min1); hfTree[min1].parent = i;
        selectMin(i + 1, min2); hfTree[min2].parent = i;
        hfTree[i].weight = hfTree[min1].weight + hfTree[min2].weight;
        hfTree[i].left = min1;
        hfTree[i].right = min2;
    }
}
```

选出parent值为0且权值最小的子树的根结点，并记录其下标。

```
template <class T>
void huffmanTree<T>::selectMin(int m, int& p){
    int j = m;
    while(hfTree[j].parent != 0) j++; // 跳过已有双亲的结点
    for(p = j, j+=1; j < 2 * size; ++j){ // 向后扫描剩余元素
        if((hfTree[j].weight < hfTree[p].weight) && 0 == hfTree[j].parent)
            p = j; // 发现更小的记录，记录它的下标
    }
}
```

哈夫曼编码

哈夫曼树被广泛应用在各种技术中，其中最典型的就是在编码技术上的应用。利用哈夫曼树，可以得到平均长度最短的编码。基本概念和术语说明如下：

1. 字符编码，狭义的字符编码是指给一组对象中的每个对象标记一个二进制位串，方便文本在计算机中存储和通过通信网络的传递。
2. 等长编码，表示一组对象的二进制位串的长度相等，如ASCII编码。
3. 不等长编码，表示一组对象的二进制位串的长度不相等。
4. 前缀码，任何一个字符的编码都不是另一个字符的编码的前缀。

通信中要讲待传字符转换成二进制位串，下面以数据通信的二进制编码的优化问题为例来分析说明。例如，有一段报文：”GOOGLE GOOSE GOOD”，在报文中出现的字符集是Data = {’G’,’O’,’L’,’E’,’S’,’D’,’ ’}，每个字符出现的频率（次数）是$W = \{4, 6, 1, 2, 1, 1, 2\}$。若每个字符用一个等长的三位二进制位串表示，则所发报文长度为$17 \times 3 = 51$。

Data	‘G’	‘O’	‘L’	‘E’	‘S’	‘D’	‘ ’
频率	4	6	1	2	1	1	2
code	000	001	010	011	100	101	110

如果按照字符出现频率的不同进行不等长编码，出现频率较多的字符采用位数较少的编码，出现频率较少的字符采用位数较多的编码，可以使得报文中的码数减少。但是这显然是不可行的，例如，“good”的二进制位串变成了“011100”，其中：

“00”可以识别为‘E’，还可以识别为“GG”

“01”可以识别为‘ ’，还可以识别为“GO”

“11”可以识别为‘S’，还可以识别为“OO”

“10”可以识别为‘L’，还可以识别为“OG”

“100”可以识别为‘D’，还可以识别为“LG”或“OE”或“OGG”

Data	‘G’	‘O’	‘L’	‘E’	‘S’	‘D’	‘ ’
频率	4	6	1	2	1	1	2
code	0	1	10	00	11	100	01

这样一来，二进制位串‘011100’就有很多种翻译方法。因此，要使编码总长最小，所设计的不等长编码必须满足一个条件，：任意一个字符的编码不能称为其他字符的编码的前缀，即必须是“前缀码”。利用二叉树可以构造出前缀码，而利用哈夫曼算法可以设计出最优的前缀码，这种编码就称为哈夫曼编码。构造哈夫曼编码的方式是：将需要传送的信息中各个字符出现的频率作为权值来构造一棵哈夫曼树，每个带权叶结点都对应一个字符，根结点到叶结点都有一条路径，我们约定路径上指向左子树的分支用0表示，指向右子树的分支用1表示，则根结点到每个叶结点路径上的0、1码序列即位相应字符的哈夫曼编码。

例如，前面报文的字符集$\text{Data}=\{\text{'G','O','L','E','S','D',' '}\}$，各字符对应的使用频率（权值）$W=\{4,6,1,2,1,1,2\}$。利用权值$W$构造哈夫曼树，然后按照左孩子为0，右孩子为1的规则构造哈夫曼编码，由于哈夫曼树不唯一，因此如无特殊约定，通常哈夫曼编码也不唯一。

构造了哈夫曼树以后，求哈夫曼编码的方式是，依次从叶结点出发，向上回溯，直至根结点，在回溯的过程中生成哈夫曼编码。即从哈夫曼树的叶结点出发，利用其双亲指针`parent`找到其双亲，然后再利用其双亲的指针域`left`和`right`来判断该结点是双亲的左孩子还是右孩子：若是左孩子，则在该叶结点的编码前添加‘0’；若是右孩子，则在该叶结点的编码前添加‘1’。

根据哈夫曼树为每个叶结点生成哈夫曼编码：

```
template <class T>
void huffmanTree<T>::huffmanEncoding(){
    int f, p; // p是当前正在处理的结点，f是p的双亲的下标
    for(int i = 0; i < 2 * size; ++i){
        hfCode[i - size].data = hfTree[i].data;
        p = i;
        f = hfTree[p].parent;
        while (f){
            if(hfTree[f].left == p) // p是其双亲f的孩子，编码+'0'
                hfCode[i - size].code = '0' + hfCode[i - size].code;
            else // p是其双亲f的右孩子，编码+'1'
                hfCode[i - size].code = '1' + hfCode[i - size].code;
            p = f;
            f = hfTree[p].parent; // 继续向上追溯
        }
    }
}
```

输出叶结点及其哈夫曼编码：

```
template <class T>
void huffmanTree<T>::printHuffmanCode(){
    for(int i = 0; i < size; i++)
        cout << hfCode[i].data << ' ' << hfCode[i].code << endl;
}
```

主函数：

```
int main(){
    char d[] = "GOLESD";
    double w[] = {4,6,1,2,1,1,2};
    huffmanTree<char> tree(7);
    tree.createHuffmanTree(d, w);
    tree.huffmanEncoding();
    tree.printHuffmanCode();
    return 0;
}
```

堆和优先级队列

堆

堆（二叉堆）是满足下列性质的序列$\{K_1, K_2, \dots, K_n\}$：

$$\begin{cases} K_i \le K_{2i} \\ K_i \le K_{2i+1} \end{cases} \quad 或 \quad \begin{cases} K_i \ge K_{2i} \\ K_i \ge K_{2i+1} \end{cases} \qquad 其中：\ i = 1, 2, \dots, \left\lfloor \frac{n}{2} \right\rfloor$$

若将此序列看成是一棵完全二叉树，则堆或者是空树，或者是满足下列特性的完全二叉树：其左、右子树分别是堆，任何一个结点的键值不大于（或不小于）其左、右孩子（若存在）的键值。编号i即为二叉树按层次遍历的次序。

最大堆，也称大根堆或大堆：结点（双亲）的键值总是大于或等于任何一个孩子的键值，根结点K_1是序列中的最大值。

最小堆，也称小根堆或小堆：结点（双亲）的键值总是小于或等于任何一个孩子的键值，根结点K_1是序列中的最小值。

完全二叉树适合用顺序存储结构表示和实现，因此堆可以利用一维数组实现。顺序结构存储的优点是：元素排列紧凑，空间利用率高；元素间的逻辑关系通过下标就可以判定，不需要借助指针，不产生结构性存储开销。

有n个结点的堆，对结点从1到n编号，数组0号单元不适用，对任意一个结点i $(1 \le i \le n)$有：若$i = 1$，则结点i为根结点，无双亲；若$i > 1$，则结点i的双亲编号是$\left\lfloor \frac{i}{2} \right\rfloor$。若$2i \le n$，则$i$的左孩子编号是$2i$，否则$i$无左孩子；若$2i + 1 \le n$，则$i$的右孩子的编号是$2i + 1$，否则$i$无右孩子。

优先级队列

优先级队列(Priority Queue)：是零个或多个元素的集合，优先级队列中的每个元素都有一个优先级，元素出队的先后次序由优先级的高低决定，而不是由入队的先后次序决定。优先级高的先出队，优先级低的后出队。优先级队列的主要特点是：支持从一个集合中快速地查找和删除具有最大值或最小值的元素。最小优先级队列适合查找和删除最小元素，最大优先级队列适合查找和删除最大元素。

优先级队列的实现方法很多，可以利用普通队列实现，还可以利用堆实现。

- **利用普通队列实现优先级队列**：入队时，按照优先级在队列中寻找合适位置插入元素；出队操作不变，仍然在队首出队。入队和出队的时间复杂度分别是$O(n)$和$O(1)$。

- **利用普通队列实现优先级队列**：入队操作不变，仍然在队尾入队；出队时，在整个队列中查找优先级最高的元素，并删除它。入队和出队的时间复杂度分别是$O(1)$和$O(n)$。
- **利用堆实现优先级队列**：例如，最小优先级队列，键值越小优先级越高，可以用一个小根堆实现。在小根堆中，存储在数组下标为1处的根结点是最小元素。入队操作就是在数组的末尾添加一个元素，然后调整元素的位置，以保持小根堆的特性；出队操作就是删除下标为1的根结点，然后调整元素的位置，以保持小根堆的特性；获取队首元素的操作就是返回下标为1的根结点。

队列的抽象数据类型定义如下：

```
template <class T>
class Queue{
public:
    virtual bool empty() const = 0; // 判空
    virtual void size() const = 0; // 队列大小
    virtual void enqueue(const T& x) = 0; // 入队
    virtual T dequeue() = 0; // 出队
    virtual T getHead() const = 0; // 获取队头元素
    virtual ~Queue() {} // 虚析构函数
};
```

自定义异常处理类：

```
class outOfRange : public exception{
public:
    const char* what() const throw(){
        return "Error! Out of range!";
    }
};
class badSize : public exception{
public:
    const char* what() const throw(){
        return "Error! Bad size!";
    }
};
```

基于小根堆的最小优先级队列的定义如下：

```
template <class elemType>
class priorityQueue : public Queue<elemType>{
    int curLength; // 当前队列长度
    elemType* data; // 指向存放元素的数组
    int maxSize; // 队列的大小
    void resize(); // 扩大队列空间
    void siftDown(int parent); // 从parent位置向下调整优先级队列
    void siftUp(int postion); // 从position位置向上调整优先级队列
public:
    priorityQueue(int initSize = 100);
    priorityQueue(const elemType data[], int size);
    ~priorityQueue() { delete[] data; }
    bool empty() const {return curLength == 0;} // 判空
    int size() const {return curLength;} // 队列大小
    void buildHeap(); // 建堆
    void enQueue(const elemType& x); // 入队
    elemType deQueue(); // 出队
```

```
    elemType getHead() const{
        if(empty()) throw outOfRange();
        return data[1]; // 返回队头元素
    }
};
```

下面讨论基于小根堆的最小优先级队列的基本操作：

入队（插入）

算法思想：若小根堆的结点个数为n，则插入一个新元素时，为了保持完全二叉树的性质，新增结点放在数组末尾，其编号应为$i=n+1$。为了保持小根堆的性质，还需要比较结点i及其双亲的键值，则将结点i中的元素与其双亲的元素进行交换，令结点i的键值不小于其双亲的键值或i到达根结点位置。为提高效率，算法设计可以采用向下移动较大的双亲数据的方式来替代交换数据。

入队：

```
template <class elemType>
void priorityQueue<elemType>::enQueue(const elemType& x){
    if(curLength == maxSize - 1) resize(); // 队列已满，扩大空间
    data[++curLength] = x; // 将新元素放在队尾
    siftUp(curLength); // 调整堆
}
```

向上调整堆，为提高效率，当双亲的键值大时，采用向下移动双亲数据的策略，而不是交换数据：

```
template <class elemType>
void priorityQueue<elemType>::siftUp(int position){
    elemType temp = data[position]; // 保存当前元素
    for(;position > 1 && temp < data[position / 2]; position /= 2)
        data[position] = data[position / 2]; // position位置元素比双亲小，双亲下移
    data[position] = temp; // 将当前元素放到合适的位置
}
```

算法分析：入队（插入）过程从完全二叉树的叶结点开始，并向上调整，最坏情况需要向上调整到根结点位为止，n个结点的完全二叉树的高度为$\lceil \log(n+1) \rceil$，入队算法`enQueue`调用向上调整算法`siftUp`中的循环最多迭代$O(\log n)$次，因此入队操作的时间复杂度为$O(\log n)$。

出队（删除）

算法思想：从最小优先级队列出队一个元素，即在小根堆中删除一个元素时，该元素必定在数组下标为1的根结点中；删除根结点后，小根堆的元素个数变为$n-1$，为了保持完全二叉树的性质，将下标为n的叶结点暂时存放在下标为1的根结点中。为了保持小根堆的性质，比较结点i和其较小孩子的键值，若结点i的键值大于其较小孩子的键值，则将结点i中的元素与其较小孩子的元素进行交换，令结点i的较小孩子成为新的结点i，继续向下比较，直到结点i的键值不大于其教小孩子的键值或i到达叶结点为止。为了提高效率，算法设计时可以采用向上移动较小孩子数据的方式来替代交换数据。

出队：

```
template <class elemType>
elemType priorityQueue<elemType>::deQueue(){
    if(empty()) throw outOfRange();
    elemType min;
    min = data[1];
    data[1] = data[curLength--];
    siftDown(1);
    return min;
}
```

向下调整堆，为提高效率，当孩子的键值较小时，采用向上移动较少的孩子数据的策略，而不是交换数据：

```
template <class elemType>
void priorityQueue<elemType>::siftDown(int parent){
    int child;
    elemType tmp = data[parent]; // 保存parent处结点
    for(;2 * parent <= curLength; parent = child){
        child = parent * 2; // child用于记录较小的子结点
        if(child != curLength && data[child + 1] < data[child])
            child++; // 右孩子更小
        if(data[child] < tmp) data[parent] = data[child];
        else break;
    }
    data[parent] = tmp;
}
```

算法分析：出队（删除）过程从完全二叉树的根结点开始，并向下调整，最坏情况一直向下调整到叶结点为止，n个结点的完全二叉树的高度为$\lceil \log(n+1) \rceil$，出队算法deQueue调用向下调整算法siftDown中的循环最多迭代$O(\log n)$次，因此出队操作的时间复杂度时$O(\log n)$。

建堆

1. 采用自上而下的建堆方法。首先初始化一个空的优先级列表，然后连续进行n次入队（插入）操作。
2. 采用自下而上的建堆方法。将给定的初始序列看成一棵完全二叉树，该完全二叉树暂时还不满足堆的性质，需要从最后一个分支结点一直到根结点，调用$\lfloor \frac{n}{2} \rfloor$次向下调整算法，把它调整成堆。可以证明，该方法的时间复杂度为$O(n)$。具有n个结点的完全二叉树，其叶结点被认为符合堆的定义，其最后一个分支结点的编号是$\lfloor \frac{n}{2} \rfloor$，从该结点开始，直到根结点，一次使用向下调整堆算法siftDown，使堆的序列从$[\frac{n}{2}, \ldots, n]$一直扩大到$[1, \ldots, n]$，就完成了初始堆的建立。

建堆方法2的实现：

```
template <class elemType>
void priorityQueue<elemType>::buildHeap(){
    for(int i = curLength / 2; i > 0; i--){
        siftDown(i); // [curLength/2..1]从下标最大的分支结点开始调整
    }
}
```

初始化堆

创建空堆，只有初始大小，没有初始序列，建堆时需要调用入队操作：

```
template <class elemType>
priorityQueue<elemType>::priorityQueue(int initSize = 100){
    if(initSize <= 0) throw badSize();
    data = new elemType[initSize];
    maxSize = initSize;
    curLength = 0;
}
```

构造无序堆，有初始大小和初始序列，使用该堆之前需要调用`buildHeap()`建堆：

```
template <class elemType>
priorityQueue<elemType>::priorityQueue(const elemType *items, int size):
maxSize(size + 10), curLength(size){
    data = new elemType[maxSize];
    for(int i = 0; i < size; i++)
        data[i + 1] = items[i]; // 复制元素
}
```

扩大堆空间

```
template <class elemType>
void priorityQueue<elemType>::resize(){
    elemType* tmp = data; // tmp指向原堆空间
    maxSize *= 2; // 扩大堆空间
    data = new elemType[maxSize]; // 申请新的堆空间
    for (int i = 0; i < curLength; ++i)
        data[i] = tmp[i]; // 复制元素
    delete [] tmp;
}
```