数据结构与算法实验报告

哈夫曼树的实现

学号： 2017141051019

姓名： 王崇智

教师点评：

成绩：

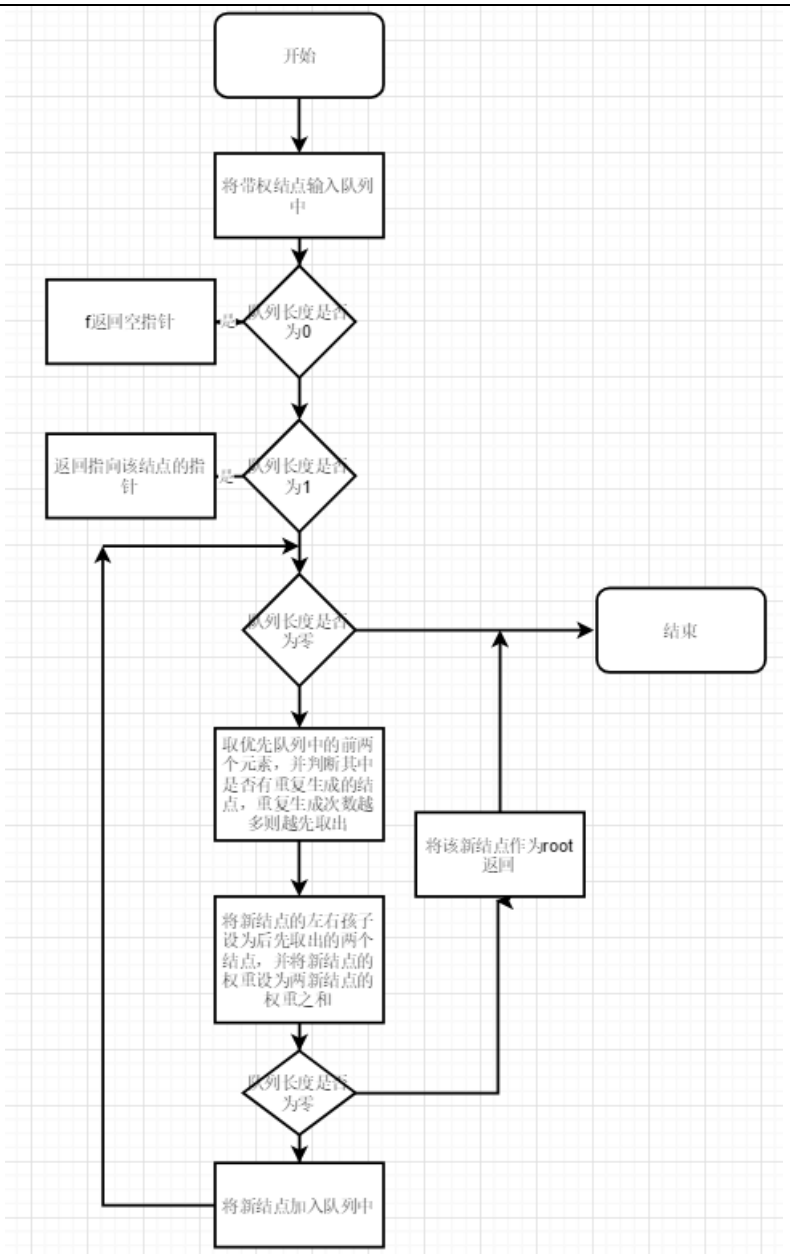
四 川 大 学 计 算 机 学 院、软 件 学 院

实 验 报 告

学号：2017141051019 姓名：王崇智 专业：计算机科学与技术 班级：173040104 第 12 周

课程名称	数据结构课程设计	实验课时	4
实验项目	哈夫曼树的实现	实验时间	第 11 周，第 12 周
实验目的	1.理解并掌握哈夫曼树的生成原理及实现算法 2.掌握优先队列的使用方法与多模板的应用 3.上机调试程序，通过差错排错使程序正常运行		
实验环境	Win10 + dev C++ 5.1.0		

实验内容（实验基本过程和内容）	一、 问题描述 哈夫曼树使一种带权路径最短的二叉树，也称最优二叉树。作为一种无损压缩方式其思想广泛应用于工业生产的各个领域。为了使一个二叉树排布更加有规律，更加方便于的人的数据操作，在次尝试使用哈夫曼的思想重构二叉树，使其权值于深度相关。 二、 基本要求 输入一系列相同类型的带权值结点，压入优先队列后调用生成哈夫曼树的函数后可以返回该新树的根节点，通过事先定义好的遍历方式可以显示出树的各个节点 三、 工具准备 Dev C++ 5.1.0 与 Gcc 编译器 四、 分析与实现 算法思想： 本实验需要利用带头节点的链表生成树，且同时需要利用优先队列的数据结构，将队列中权重最小的两个结点不断从队列中取出，将这两个作为新结点的子节点，且新节点的权重为两个子节点的权重 1 的之和，将其重新压入已规定为由小到大排列的 STL 优先队列中重新进行操作即可。 算法流程图如下：



注意事项：

1. 在生成树的函数中需要注意三种情况，一种是队列中没有数据，则直接返回空指针。一种是队列长度为一，直接返回指向该结点的指针即可。最后是队列长度大于一，则采用 while 循环的方式，直到队列为空，返回最后处理得到的 root 指针即可。
2. 每次重新压入处理后的结点需要对该几点的属性进行判断，例如出现权重为 1 2 3 的三个结点，1+2 得到权重为

3 的新结点后进入队列，但现在存在两个权重为 3 的结点，为了确定其位于新结点的左右相对位置以保证输出新树时候数据，权重与位置不产生矛盾。需要判断取出的结点的子节点是否存在，并进行相应的数据处理。

3. 本次实验定义了很多模板类，也又很多模板函数，需要明确各个函数和类公用的模板与单独使用的模板。且传递参数时要注意时传引用还是传指针。

源程序如下：

```
#include"map"
#include"queue"
#include"iostream"
#include"algorithm"
#include"utility"
using namespace std;

template<class T>
class TreeNode
{//基本的结点类
    private:
        T element;

        template<class C>
        friend void PreOrder(TreeNode<C> node);
    public:
        TreeNode<T> *lchild;
        TreeNode<T> *rchild;
        TreeNode(T
elem):element(elem),lchild(NULL),rchild(NULL){}
        bool AddL(TreeNode<T> &lnew);
        bool AddR(TreeNode<T> &rnew);
        template<class C>
        friend void preorder(TreeNode<C> *t);
};

template<class T>
void preorder(TreeNode<T> *t){
    if(t){
        cout<<t->element<<endl;
```

```
        preorder(t->lchild);
        preorder(t->rchild);
    }

}

template<class T>
class wei_in_pair;
//之后需要用到的一个对，用来存储结点和其权重

template<class T>
class TreeNode_Weight:public TreeNode<T>
{
    friend class wei_in_pair<T>;
    private:
        int weight;
    public:
        TreeNode_Weight(T elem,int
wei):TreeNode<T>(elem),weight(wei){}
        TreeNode_Weight* operator+(TreeNode_Weight<T>
&elem){
            return new
TreeNode_Weight('\0',this->weight+elem.weight);
        }
        int GetWeight() const
        {
            return this->weight;
        }
};

template<class T>
class wei_in_pair
{
    template<class C>
    friend bool operator >(wei_in_pair<C> a,wei_in_pair<C> b);
    private:
        pair<int,TreeNode_Weight<T>*> elem;
    public:
        wei_in_pair(TreeNode_Weight<T> &obj){
            elem.first=obj.weight;
```

```
            elem.second=&obj;
        }
        TreeNode_Weight<T> * second() const
        {
            return elem.second;
        }
};

template<class T>
bool operator>(wei_in_pair<T> a,wei_in_pair<T> b)
{
    return a.elem.first>b.elem.first;
}
/*
template<class T>
priority_queue<wei_in_pair<T>, vector<wei_in_pair<T> >,
greater<wei_in_pair<T> > >  pq;
*/

template<class T>
TreeNode_Weight<T>*
buildHuff(priority_queue<wei_in_pair<T>,
vector<wei_in_pair<T> >, greater<wei_in_pair<T> > > pq);

template<class T>
bool TreeNode<T>::AddL(TreeNode<T> &lnew){
    if(this->lchild==NULL)
    {
        this->lchild=&lnew;
        return true;
    }
    else
    {
        return false;
    }
}
//为新结点进行赋值
template<class T>
bool TreeNode<T>::AddR(TreeNode<T> &rnew){
    if(this->rchild==NULL)
```

```
    {
        this->rchild=&rnew;
        return true;
    }
    else
    {
        return false;
    }
}

template<class T>
TreeNode_Weight<T>*
buildHuff(priority_queue<wei_in_pair<T>,
vector<wei_in_pair<T>>, greater<wei_in_pair<T>> > pq)
{
    TreeNode_Weight<T> *root,*lchild,*rchild;
    if(pq.empty())
    {
        return NULL;
    }
    if(pq.size()==1)
    {
        root=pq.top().second();
        pq.pop();
        return root;
    }

    while(true)
    {
        if(pq.top().second()->lchild)
        {
            rchild=pq.top().second();
            pq.pop();
            lchild=pq.top().second();
            pq.pop();
        }
        else
        {

        lchild=pq.top().second();
```

```
        pq.pop();
        rchild=pq.top().second();
        pq.pop();
        }
        root= *rchild + *lchild;
        root->AddL(*lchild);
        root->AddR(*rchild);
        if(pq.empty()){
            break;
        }
        pq.push(wei_in_pair<T>(*root));
    }
    return root;
}

int main()
{
    priority_queue<wei_in_pair<int>,
vector<wei_in_pair<int> >, greater<wei_in_pair<int> >> pq;

    TreeNode_Weight<int> a1(1, 1);
    TreeNode_Weight<int> a2(2, 2);
    TreeNode_Weight<int> a3(4, 3);
    TreeNode_Weight<int> a4(9, 9);



    wei_in_pair<int> a5(a1),a6(a2),a7(a3),a8(a4);
    pq.push(a5);
    pq.push(a6);
    pq.push(a7);
    pq.push(a8);
    TreeNode_Weight<int> *tt = buildHuff(pq);
    preorder(tt);
    return 0;
}
```

数据记录和计算	五、 测试与结论 例 1. ``` TreeNode_Weight<int> a1(1, 1); TreeNode_Weight<int> a2(2, 2); TreeNode_Weight<int> a3(4, 3); TreeNode_Weight<int> a4(9, 9); ``` 输出结果采用前序遍历为：  ，生成哈夫曼树的程序得到验证，实验成功（其中 0 均为队列操作中的新结点）。
实验总结	哈夫曼树生成的思想很巧妙而且实现过程借助 STL 十分便捷，且即使自己写数据结构也可以更快地实现。这次实验中利用了现成地 STL 优先队列，也掌握了一些对队列地基本操作。又重新回顾了引用，传地址，传指针地相关知识。 重要的是深入了解了哈夫曼树的应用领域与基本实现过程，很有现实意义。本次实验还未完成的是完成编码，但其思想就是在树的生成之后进行某种遍历，在遍历过程中存储一些动态数组，在数组中存入其位于左孩子还是右孩子的位置，不断递归，到每一位之处将现有编码赋给该结点即可完成编码工作。
指导老师评 议	成绩评定： 指导教师签名：