SHANGHAI JIAO TONG UNIVERSITY

内部算法排序

吕艺

517021910745

December 6, 2018

1 需求分析

1. 本演示文件的主要需求为判断各种排序方法的比较次数和移动次数，涉及到的排序方法分别为冒泡排序，直接插入排序，简单选择排序，快速排序，希尔排序和堆排序。首先先生成随机数 num, 代表随机数个数。根据每组随机数个数不超过 100 个的要求，本演示文件中选择了每组随机数的个数在 100 1000 个。根据随机数的组数不少于 5 的条件，本演示文件中选择的随机数组数为 8 组。

2. 演示文件无需用户输入，文件运行结束后即在计算机终端上打印运算结果，显示在屏幕上。

3. 程序执行的命令包括:

1. 根据系统时间生成随机数个数。

2. 根据系统时间和事先生成的随机数个数生成随机数数组

3. 利用上述六种排序方式对随机数数组进行从小到大的排序

4. 将每种排序方式的比较次数和移动次数打印在屏幕上

4. 测试数据

本演示文件中用到的测试数据均为随机生成的数组

2 概要设计

本演示文件中涉及到排序方式多样，例如插入排序中的直接插入排序和希尔排序，选择排序中的直接选择排序，堆排序，交换排序中的冒泡排序和快速排序。

1. 哈夫曼类 $hfTree$

数据对象：Node *elem;　　int length;

私有类：struct Node {
　　Type data;
　　int weight; };
　　struct hfCode {
　　Type data;
　　string Code;};

基本操作：hfTree(const Type *x, const int *w, int size);
　　操作条件：哈夫曼树还未被初始化
　　操作结果：利用传入参数初始化哈夫曼树
　　void getCode(hfCode result[]) const;
　　初始条件：哈夫曼树已被建好
　　操作结果：形成含有编码的 $Node$ 类型的数组
　　hfTree(const hfTree & myTree);
　　初始条件：哈夫曼树已经被初始化
　　操作条件：将传入的树 $MyTree$ 的各个数据对象复制给该树
　　hfTree() delete[] elem;
　　操作条件：哈夫曼树存在

操作结果：释放哈夫曼树中 *elem* 所占的内存

Node *GetElem() const return elem;

操作条件：哈夫曼树存在

操作结果：返回指针 *elem*

int GetLength() const return length;

操作条件：哈夫曼树存在

操作结果：返回哈夫曼叶子结点的个数

2. 本程序包括两个模块：

1) int main() {
while(Command != 'E'){
Process Command;
input Command;} }

2) 哈夫曼树单元模块--实现哈夫曼树的构建和编码

3 详细设计

1）哈夫曼树单元模块

```
//定义模板类
template<class Type>
class hfTree {
public:
    //定义节点类
    struct Node {
        Type data;                  //节点数据类型
        int weight;                 //节点所对应的字符出现次数
        int parent, left, right;    //节点的父亲，左儿子右儿子在数组中的位置
    };
private:
    Node *elem;                     //Node类型的数组指针
    int length;                     //字符个数

public:
    //定义哈夫曼编码类
    struct hfCode {
        Type data;                  //字符
        string Code;                //对应的哈夫曼编码
    };

    hfTree(const Type *x, const int *w, int size);  //根据给定数值初始化哈弗曼树

    void getCode(hfCode result[]) const;            //返回哈弗曼编码数组的数组名指针

    hfTree(const hfTree &myTree);                   //哈弗曼树的拷贝构造函数
```

```
    ~hfTree() { delete[] elem; }                    //析构函数

    Node *GetElem() const { return elem; }          //返回哈弗曼树的节点数组名指针

    int GetLength() const { return length; }        //返回哈弗曼树的字符个数
};

//定义哈夫曼树的拷贝构造函数
template <class Type>
hfTree<Type>::hfTree(const hfTree<Type> & myTree)
{
    delete [] elem;                             //释放指针内存
    elem = new Node[myTree.GetLength()];        //申请新的空间
    for(int i=0;i<myTree.GetLength();++i)
    {
        elem[i] = myTree.GetElem()[i];          //对应元素赋值
    }
    length = myTree.length;                     //长度赋值
}

//定义哈夫曼树的初始化函数
template <class Type>
hfTree<Type>::hfTree(const Type *v,const int *w,int size)
{
    const int MAX_INT = 32767;                  //出现次数的上界
    int min1,min2;                              //出现次数最少的两个
    int x,y;                                    //x对应出现次数第二少的，y对应最少的

    length = 2*size;
    elem = new Node[length];

    for(int i=size;i<length;++i)                //叶节点的字符，出现次数和父母孩子所在位置初始化
    {
        elem[i].weight = w[i-size];
        elem[i].data = v[i-size];
        elem[i].parent = elem[i].left=elem[i].right=0;
    }
    for(int i=0;i<size;++i)                 //将非叶节点的数值初始化为'\0'(为了之后打印方便)
    { elem[i].data = '\0'; }

    //合并树
    for(int i=size-1;i>0;--i)
    {
        min1=min2=MAX_INT; x=y=0;               //通过遍历挑选出出现次数最少的两个节点
        for(int j=i+1;j<length;++j) {
            if (elem[j].parent == 0) {
                if (elem[j].weight < min1) {
                    min2 = min1;
                    min1 = elem[j].weight;
```

```
                    x = y;
                    y = j;
                }
                else if (elem[j].weight < min2) {
                    min2 = elem[j].weight;
                    x = j;
                }
            }
        }
        elem[i].weight = min1 + min2;                //进行对赋值处理
        elem[i].left = x;
        elem[i].right = y;
        elem[i].parent = 0;
        elem[x].parent = i;
        elem[y].parent = i;
    }
}



//生成哈夫曼编码
template<class Type>
void hfTree<Type>::getCode (hfCode result[]) const
{
    int size = length/2;
    int p,s;                                    //s为目前处理的节点   p是s的父亲节点
    for(int i=size;i<length;++i)
    {
        result[i-size].data = elem[i].data;
        result[i-size].Code ="";
        p = elem[i].parent; s=i;
        while(p)
        {
            if(elem[p].left==s)                 //若s为父亲节点的左儿子，则在编码最前面加'0'
                result[i-size].Code = '0'+result[i-size].Code;
            else                                //若s为父亲节点的右儿子，则在编码最前面加'1'
                result[i-size].Code = '1' + result[i-size].Code;
            s = p;
            p = elem[p].parent;
        }
    }
}
```

2）主函数模块

```
int main() {
    cout<<"Welcome␣to␣Huffman␣System!"<<endl;                  //输出提示信息
    cout<<"I(initialization)␣E(encoding)␣D(Decoding)␣P(Code␣Printing)␣T(Tree␣printing)"<<endl;
    char Command;                                               //输入命令
    in>>Command;
```

```
    hfTree<char>* MyTree;                                   //利用默认构造函数定义一棵树

    hfTree<char>* (*I)()  = initial;                        //利用函数指针实现主菜单的功能
    void (*E)(const hfTree<char> & MyTree) = Encoding;
    void (*D)(const hfTree<char> & MyTree) = Decoding;
    void (*P)(const hfTree<char> &MyTree) = CodePrinting;
    void (*T)(const hfTree<char> & MyTree) = Treeprinting;

    int flag = false;
    while(Command != 'Q')                                   //直到输入'Q'命令后再退出循环
    {   if(Command != 'I' && flag == false)
        //未初始化时无法进行其他操作，打印错误信息并提示用户重新输入
        cout<<"Not Initialized yet!!"<<endl;
        if(Command=='I' )
        {
            MyTree = initial();                             //初始化哈弗曼树
            flag = true;
            cout<<"Initialization ended!"<<endl;
        }

        if(Command == 'E'&&flag != 0 ) {                    //根据已建好的哈弗曼树进行编码并将结果输出到文件中
            Encoding(*MyTree);
            cout<<"Encoding ended!"<<endl;
        }


        if(Command == 'D'&&flag != 0 )                      //读入文件进行解码，并将解码的结果输出到文件中
        {Decoding(*MyTree);
         cout<<"Decoding ended!"<<endl;}

        if(Command == 'P'&&flag != 0)                       //将编码按要求打印在屏幕上并输出到文件中
        {CodePrinting(*MyTree);
         cout<<"Code Printing ended!"<<endl;}

        if(Command == 'T'&&flag != 0)                       //将已建好的哈夫曼树以凹入表的形式输出到文件中
        {Treeprinting(*MyTree);
        cout<<"Tree Printing Ended!"<<endl;}

        in>>Command;
    }
    return 0;
}
```

3）函数指针模块

```
hfTree<char>* initial()                                     //初始化哈弗曼树
{
    int n;
    cout<<"Please input the number of characters:"<<endl;
    in>>n;                                                  //输入叶节点个数
    in.get();
```

```
    cout<<"Please␣input␣the␣characters:(no␣spaces)"<<endl;
    char * chars;
    chars = new char [n + 1];                                   //输入叶节点的值
    in.getline(chars,n + 1);


    int * weights;
    weights = new int [n];
    cout<<"Please␣input␣the␣weights:(separated␣by␣spaces)"<<endl;
    for(int i=0;i<n;++i)                                        //输入节点的出现的次数
    {
        in>>weights[i];
    }
    ofstream out2;                                              //重载输出流
    out2.open("hfmTree.txt");
    hfTree<char>* MyTree = new hfTree<char>(chars,weights,n);
    out2<<"characters:"<<endl;
    for(int i=0;i<n;i++)
    {
        out2<<chars[i]<<"␣";                                    //输出字符
    }
    out2<<endl;

    out2<<"weights:"<<endl;
    for(int i=0;i<n;i++)
    {out2<<weights[i]<<"␣";}                                    //输出出现次数
    out2<<endl;

    out2<<"parents:"<<endl;
    for(int i=n;i<2*n;i++)
    {out2<<MyTree -> GetElem()[i].parent<<"␣";}        //输出父亲节点在数组中所在位置的下标

    out2<<endl;
    return MyTree;
}


//编码给定文件
void Encoding(const hfTree<char> & MyTree)
{   ifstream in3;
    in3.open("plainFile.txt");                  //重载输入流
    ofstream out3;
    out3.open("codeFile.txt");                  //重载输出流
    int num = MyTree.GetLength()/2;
    hfTree<char>::hfCode result[num];
    MyTree.getCode(result);

    char now= in3.get();
    while(now!=EOF)                             //读入文件中的字符
    {
```

```
        for(int i = 0;i<num;i++)
        {
            if(now == result[i].data)
            {out3<<result[i].Code<<"␣";          //输出字符对应编码
            break;}
        }
        now = in3.get();
    }
    out3.close();
    in3.close();
}


//解码给定文件
void Decoding(const hfTree<char> &MyTree)
{
    ifstream in4;                               //重载输入流
    ofstream out4;                              //重载输出流
    in4.open("codeFile.txt");
    out4.open("textFile.txt");
    string cur;
    int num = MyTree.GetLength()/2;
    hfTree<char>::hfCode result[num];
    MyTree.getCode(result);

    for(int i = 0;i < MyTree.GetLength(); ++i)          //文件输入编码
    {
        getline(in4,cur,'␣');
        for(int i = 0;i < MyTree.GetLength(); ++i)
        {
            if(cur == result[i].Code)
            {out4<<result[i].data; break;}              //文件输出编码对应字符
        }
    }
    in4.close();
    out4.close();
}


//打印编码
void CodePrinting(const hfTree<char> &MyTree)
{
    ifstream in5;                               //重载文件输入流
    in5.open("codeFile.txt");
    ofstream out5;                              //重载文件输出流
    out5.open("codePrint.txt");
    int sum = 0;
    char cur = in5.get();
    while(cur != EOF)                           //读入文件中的编码
    {
```

```
        if(cur == '␣')                          //跳过用作分割的空格
        {
            cur = in5.get();
            continue;
        }
        if(sum < 50)                            //每行打印不超过50个字符
        {
            cout<<cur;
            sum++;
        }
        else
        {
            sum = 1;
            cout<<endl;
            cout<<cur;
        }
        cur = in5.get();
    }
    in5.close();
    out5.close();
    cout<<endl;
}

void PrintHelp (hfTree<char>::Node p,const hfTree<char> & MyTree, ofstream& out6,string ss);


//利用递归以凹入表的形式打印哈弗曼树
void Treeprinting(const hfTree<char> & MyTree)
{
   ofstream out6;                               //重载文件输出流
   out6.open("treePrint.txt");
   string ss = "";
   PrintHelp(MyTree.GetElem()[1],MyTree, out6,ss);       //递归打印哈弗曼树
}

void PrintHelp (hfTree<char>::Node p,const hfTree<char> & MyTree, ofstream& out6,string ss)
{
    if(p.parent != 0)
        ss += "\t";                                         //利用空格表示层次关系
    if (p.left == 0)                                        //叶节点执行完则输出后返回
    {if (p.data != '\0')
        out6 << ss<<p.weight << "(" << p.data << ")"<<endl;     //如果为叶节点则输出节点值出现次数和字符
    else
        out6 << ss<<p.weight<<endl;                         //如果为非叶节点则仅输出节点值出现次数
    return;
    }
    PrintHelp(MyTree.GetElem()[p.left], MyTree, out6,ss);
    if (p.data != '\0')
        out6 <<ss<< p.weight << "(" << p.data << ")"<<endl;     //如果为叶节点则输出节点值出现次数和字符
    else
```

```
        out6 << ss<<p.weight<<endl;                    //如果为非叶节点则仅输出节点值出现次数
    PrintHelp(MyTree.GetElem()[p.right], MyTree, out6,ss);
}
```

4 调试分析

1. 一开始本演示文件对于哈夫曼树仅设计了含有形参的初始化函数，因此哈夫曼树仅能在参数给定后被初始化 (在本题中只能在初始化函数作用域中被使用，函数调用结束后就会被析构)，这导致初始化后必须保存参数，并在每次执行其他操作时新建一棵树，大大降低的时间效率，提高了复杂度。为了解决这一问题，本演示文件中重载了初始化函数，并在主函数内定义了一棵默认哈弗曼树。调用初始化函数后利用重载的拷贝构造函数保存下初始化的树，在主函数作用域中都可用。

2. 一开始由于对输入情况考虑的欠缺，本演示文件一开始把哈弗曼树的节点值设为了字符型。为了加强类的泛化能力，本演示文件采用模板类以及模板函数，从而使文件的适用范围更广。

3. 算法的复杂度分析

1）时间复杂度由于采用数组的形式存储哈弗曼树，各种操作的算法复杂度比较合理。initial 函数和 Decoding 函数的时间复杂度是 $O(n^2)$, 函数 Codeprinting 和函数 Treeprinting 的时间复杂度均为 $O(n)$，而 Encoding 函数的时间复杂度是 $M(max\{O(m^2),O(n^2\})$ (设文件读入规模为 m, 哈弗曼树叶节点个数为 n)。

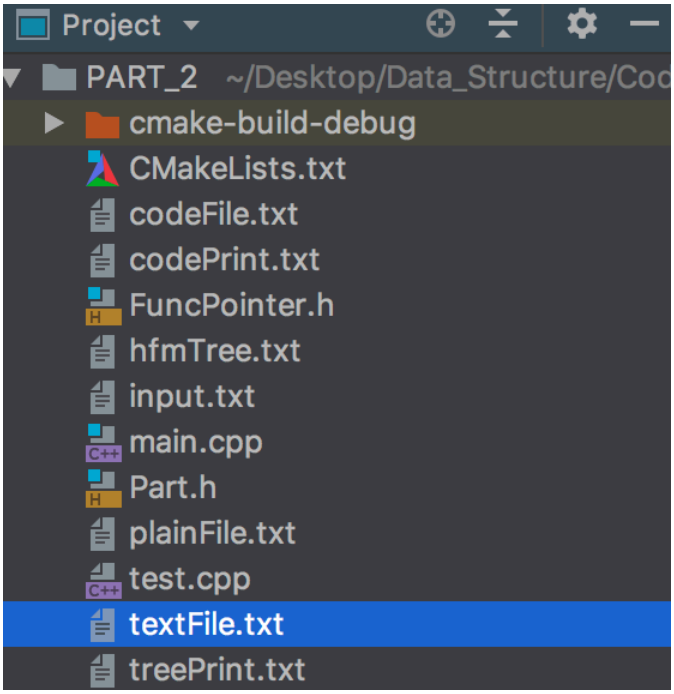
initial 函数在调用初始化函数时利用双层循环遍历进行树的合并，而 Decoding 函数中也利用双层循环读入文件的编码并寻找到对应字符的编码输出到文件中，因此这两个函数的时间复杂度为 $O(n^2)$。Encoding 函数从文件中读入 m 个字符并在 n 个字符中找到对应字符的编码输出，所以时间复杂度为 $M(max\{O(m^2),O(n^2\})$。Codeprinting 函数和 Treeprinting 函数的时间复杂度主要决定于编码长度和哈弗曼树节点个数，因此时间复杂度与元素个数成正比，因此时间复杂度为 $O(n)$。

2）空间复杂度

哈弗曼树模块的空间复杂度与叶节点的个数成正比，即 $O(n)$。主函数模块的复杂度取决于定义主函数作用域中的哈弗曼树，故空间复杂度也为 $O(n)$。

5 用户手册

1. 本程序以 Jetbrains Clion 2018.2.5, 采用 C++ 11 标准，程序以项目方式组织 (project)，如图 1 所示：



2. 依次点击菜单"Run"->build, 再点击"Run", 显示文本方式的用户界面, 如图 2 所示:

3. 键入操作命令符后按“回车键”，程序就执行相应命令 (用户需在初始化时输入对应参数)。有效的命令符为 I,E,D,P,T 或 Q。若输入的命令符无效，提示并要求重新输入命令符。

6 测试结果

执行命令'I' 后: 输入测试数据 2 中的参数和权重, 建立哈弗曼树执行命令'E' 后: 在文件 PlainFile 中输入"THIS PROGRAM IS MY FAVORITE", 将编码存储到 codeFile 中执行命令'D' 后: 新建文件 textFile, 将文件 codeFile 中的编码翻译成"THIS PROGRAM IS MY FAVORITE" 执行命令'P' 后: 在屏幕上打印 codeFile 中的编码，并保存到 codePrint 中执行命令'T' 后: 将已建好的哈弗曼树以凹入表的形式存储到 treePrint 文件中

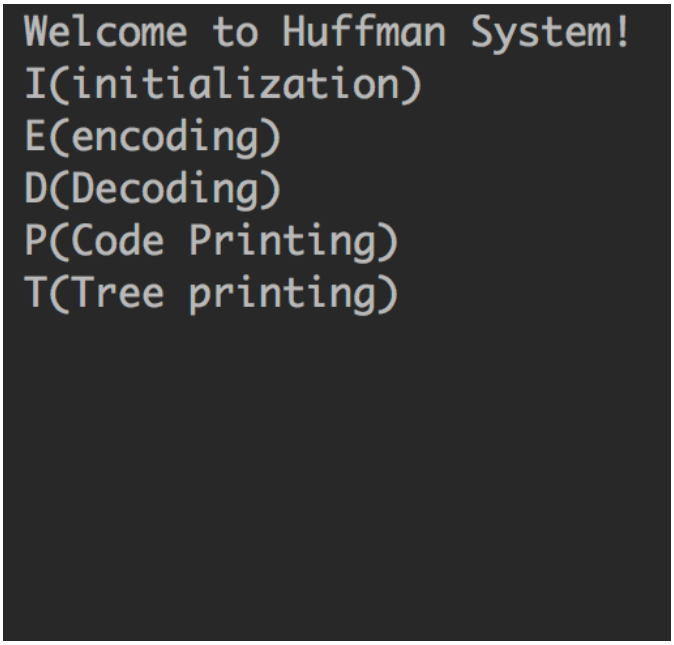
7 附录

源程序文件名清单

main.cpp //主函数

Part.h //哈夫曼树单元模块

FuncPointer.h //函数指针模块


Welcome to Huffman System!
I(initialization)
E(encoding)
D(Decoding)
P(Code Printing)
T(Tree printing)