山东大学　　计算机科学与技术　　学院

数据结构与算法　　课程实验报告

学号：201700130033	姓名：　武学伟	班级：　2017 级 2 班
实验题目：堆及其应用		
实验学时：4	实验日期：　2018.12.3	

实验目的：

1.掌握堆结构的定义、描述方法、操作定义及实现。
2.掌握堆结构的应用

软件环境：

Win10home，sublime

1. 实验内容（题目内容，输入要求，输出要求）

1）第 k 小元素

维护一个系统，支持查询数据库第 k 小元素（初始 k=0）

命令有以下三种：

令 M：向数据库中插入一个数，值为 x；

令 L：k++；

令 W：输出数据库中的第 k 小元素

输入描述：

第一行，一个正整数 n，表示命令的条数 。

接下来 n 行，每行一个字母 M、L 或 W，表示执行命令 M、L 或 W。

对于 M 命令，之后有一个正整数 x，表示向数据库中插入 x。

输出描述：

对于每条 W 命令，输出一个正整数。

2）Huffman 树与 Huffman 编码

输入描述：

一串小写字母组成的字符串，长度不超过 10^5。

输出描述：

输出这个字符通过 Huffman 编码的长度。

2. 数据结构与算法描述　（整体思路描述，所需要的数据结构与算法）

数据结构：

1) 实验一：小根堆、大根堆
2) 实验二：二叉树、小根堆

算法：

1) 实验一：
   a) 在小根堆中加入一个数据成员：容量为 n 的数组，n 为小根堆元素的数量，在每一次 W 命令时，进行一次堆排序，将结果保存在数组中，用另外一个数据成员 k 保存需要输出的第 k 小元素的索引，然后输出数组的第 k 个元素。
   b) 上述的时间复杂度较高，为了降低复杂度，采用小根堆与大根堆合用的方法。

      对于 L 命令，将小根堆的最小元素插入到大根堆中。

对于 M 命令，将元素插入到小根堆中，然后比较小根堆的最小元素和大根堆的最大元素，若是大根堆的最大元素大于小根堆中的最小元素，则将两个元素交换，保证大根堆中最大元素小于小根堆的最小元素。此时，大根堆的最大元素就是原本第 k 小的元素。

对于 W 命令，直接输出大根堆的最大元素。

2) 实验二：

a) 创建霍夫曼树类

数据成员

```
linkedBinaryTree<T>* tree;  //霍夫曼树
int stringLength;           //霍夫曼编码长度
int countNum;               //元素数量
int* weight;                //权重数组
T* theElement;              //元素数组
```

成员函数

```
int makeHuffman();              //构建霍夫曼树并获取编码长度
void getin(string& in);         //获取输入
```

b) 获取输入的信息

首先统计字符串中的每个字母出现的频率存储在数组中，然后统计出现的字母的数量（即遍历频率数组，当元素非零时，数量加一）。

若是字母的数量为 1，则直接输出字符长度，不需要进行后面的操作。然后将字母和权重赋值给元素数组以及权重数组。

c) 构建霍夫曼树

创建一个霍夫曼树的节点数组，每个节点是一个单节点的二叉树，然后建立一个小根堆，存储这些节点，每次去除小根堆的前两个最小的元素，然后合并为一棵树，根节点为对应的权值相加。

d) 计算编码长度，编码长度等于所有新合成的节点的权值之和，所以在每次合并的时候，计算出权值之和，最终的和就是编码长度。

3. 测试结果（测试输入，测试输出，结果分析）

测试一（实验一）：

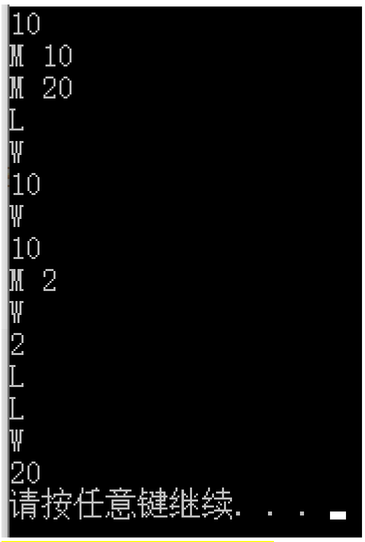
输入：

10
M 10
M 20
L
W
W
M 2
W
L
L
W

输出：

10
10

2
20


10
M 10
M 20
L
W
10
W
10
M 2
W
2
L
L
W
20
请按任意键继续. . .

测试二（实验二）：

输入：

abcdabcaba

输出：

19


abcdabcaba
19请按任意键继续. . .

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

结果分析：

数据正确。

实验一原本采用堆排序来获得排序结果，但是每次 W 命令进行一次堆排序，一次的时间复杂度为 O(nlogn)，多次操作时间复杂度较高，并且没有必要。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
/*Ex11_1.cpp*/
#include <iostream>
#include <string>
using namespace std;

/*改变数组的长度*/
template <class T>
void changeLength1D(T*& a, int oldLength, int newLength)
{
    if (newLength < 0)      //新数组长度小于 0，抛出异常
    {
        throw "New length must be >= 0";
        cout << "New length must be >= 0" << endl;
```

```
    }
    T* temp = new T[newLength];  //新数组
    int number;                 //需要复制的元素数
    if (oldLength<newLength)
        number = oldLength;
    else
        number = newLength;
    for (int i=0; i<number; i++)
        temp[i] = a[i];
    delete []a;     //删除旧数组
    a = temp;     //复制新的数组
}

/*小根堆*/
template <class T>
class minHeap
{
   public:
       minHeap (int initialCapacity = 10);
       ~minHeap () {delete []heap;}
       bool empty() const {return heapSize == 0;}
       int size() const {return heapSize;}
       T& top()
       {
           if (heapSize == 0)
           {
               string error = "empty heap";
               throw error;
           }
           return heap[1];
       }
       void pop();                     //删除最小元素
       void push(const T& theElement); //插入元素
       void output();                  //输出
       void initialize(T*, int);       //构建小根堆
       void deactivateArray()
       {

           heap = NULL;
           arrayLength = heapSize = 0;
       }
   private:
       int heapSize;                //元素数量
       int arrayLength;             //队列元素加 1
```

```
        T* heap;                        //元素数组
};

/*构造函数*/
template <class T>
minHeap<T>::minHeap (int initialCapacity)
{
    if (initialCapacity < 1)
    {
        cout << "wrong capacity" << endl;
        return;
    }
    arrayLength = initialCapacity + 1;
    heap = new T[arrayLength];
    heapSize = 0;
}

/*删除最小值*/
template <class T>
void minHeap<T>::pop()
{
    if (heapSize == 0)
    {
        string error = "empty heap";
        throw error;
    }
    heap[1].~T();
    T lastElement = heap[heapSize--];       //将尾节点保存并在树中删
除
    int currentNode = 1;                //从根节点起，为其寻找位置
    int child = 2;
    while (child <= heapSize)               //child 应该是较小的孩子
    {
        if (child < heapSize && heap[child] > heap[child+1])
            child++;                    //找到较小的孩子
        if (lastElement <= heap[child])     //如果根节点较小，则是正
确的位置
            break;
        heap[currentNode] = heap[child];    //否则孩子上移一层
        currentNode = child;
        child *= 2;
    }
    heap[currentNode] = lastElement;        //正确放置尾节点
}
```

```
/*插入新值*/
template <class T>
void minHeap<T>::push (const T& theElement)
{
    if (heapSize == arrayLength-1)          //堆已满，扩容
    {
        changeLength1D (heap, arrayLength, 2*arrayLength);
        arrayLength *= 2;
    }
    int currentNode = ++heapSize;           //新插入的元素在数组最后
    while (currentNode != 1 && heap[currentNode/2] > theElement)
    {
        heap[currentNode] = heap[currentNode/2];  //新插入值较大，移
动双亲
        currentNode /= 2;                   //新值移向双亲
    }
    heap[currentNode] = theElement;         //正确放置新值
}

/*输出*/
template <class T>
void minHeap<T>::output ()
{
    for (int i=1; i<heapSize; i++)
        cout << heap[i] << " ";
    cout << endl;
}

template <class T>
void minHeap<T>::initialize(T* theHeap, int theSize)
{
    delete []heap;
    heap = theHeap;
    heapSize = theSize;
    for (int root = heapSize/2; root >= 1; root--)
    {
        T rootElement = heap[root];
        int child = 2*root;
        while (child <= heapSize)
        {
            if (child < heapSize && heap[child] > heap[child+1])
                child++;
            if (rootElement <= heap[child])
```

```
                break;
            heap[child/2] = heap[child];
            child *= 2;
        }
        heap[child/2] = rootElement;
    }
}

/*大根堆*/
template <class T>
class maxHeap
{
    public:
        maxHeap (int initialCapacity = 10);
        ~maxHeap () {delete []heap;}
        bool empty() const {return heapSize == 0;}
        int size() const {return heapSize;}
        T& top()
        {
            if (heapSize == 0)
            {
                string error = "empty heap";
                throw error;
            }
            return heap[1];
        }
        void pop();                          //删除最大元素
        void push(const T& theElement);      //插入元素
        void output();                       //输出
        void initialize(T*, int);            //构建小根堆
        void deactivateArray()
        {

            heap = NULL;
            arrayLength = heapSize = 0;
        }
    private:
        int heapSize;                    //元素数量
        int arrayLength;                 //队列元素加 1
        T* heap;                         //元素数组
};

/*构造函数*/
template <class T>
```

```
maxHeap<T>::maxHeap (int initialCapacity)
{
    if (initialCapacity < 1)
    {
        cout << "wrong capacity" << endl;
        return;
    }
    arrayLength = initialCapacity + 1;
    heap = new T[arrayLength];
    heapSize = 0;
}

/*删除最大值*/
template <class T>
void maxHeap<T>::pop()
{
    if (heapSize == 0)
    {
        string error = "empty heap";
        throw error;
    }
    heap[1].~T();
    T lastElement = heap[heapSize--];
    int currentNode = 1;
    int child = 2;
    while (child <= heapSize)
    {
        if (child < heapSize && heap[child] < heap[child+1])
            child++;
        if (lastElement >= heap[child])
            break;
        heap[currentNode] = heap[child];
        currentNode = child;
        child *= 2;
    }
    heap[currentNode] = lastElement;
}

/*插入新值*/
template <class T>
void maxHeap<T>::push (const T& theElement)
{
    if (heapSize == arrayLength-1)
    {
```

```
        changeLength1D (heap,arrayLength, 2*arrayLength);
        arrayLength *= 2;
    }
    int currentNode = ++heapSize;
    while (currentNode != 1 && heap[currentNode/2] < theElement)
    {
        heap[currentNode] = heap[currentNode/2];
        currentNode /= 2;
    }
    heap[currentNode] = theElement;
}

/*输出*/
template <class T>
void maxHeap<T>::output ()
{
    for (int i=1; i<heapSize; i++)
        cout << heap[i] << " ";
    cout << endl;
}

template <class T>
void maxHeap<T>::initialize(T* theHeap, int theSize)
{
    delete []heap;
    heap = theHeap;
    heapSize = theSize;
    for (int root = heapSize/2; root >= 1; root--)
    {
        T rootElement = heap[root];
        int child = 2*rootElement;
        while (child <= heapSize)
        {
            if (child < heapSize && heap[child] < heap[child+1])
                child++;
            if (rootElement >= heap[child])
                break;
            heap[child/2] = heap[child];
            child *= 2;
        }
        heap[child/2] = rootElement;
    }
}
```

```cpp
template <class T>
class minNok
{
    public:
        minNok() {;}
        ~minNok() {;}
        void insert(const T& theElement);
        void remove();
        void out();
    private:
        maxHeap<T> max;        //大根堆
        minHeap<T> min;        //小根堆
};

template <class T>
void minNok<T>::insert(const T& theElement)
{
    min.push(theElement);
    if (!max.empty())
    {
        if (max.top() > min.top())
        {
            T temp = min.top();
            min.pop();
            min.push(max.top());
            max.pop();
            max.push(temp);
        }
    }
}

template <class T>
void minNok<T>::remove()
{
    max.push(min.top());
    min.pop();
}

template <class T>
void minNok<T>::out()
{
    cout << max.top() << endl;
}
```

```
int main()
{
    minNok<int> test;
    int num;
    cin >> num;
    char choice;
    for (int i=0; i<num; i++)
    {
        cin >> choice;
        switch(choice)
        {
            case 'M':
                int n;
                cin >> n;
                test.insert(n);
                break;
            case 'L':
                test.remove();
                break;
            case 'W':
                test.out();
            default:
                break;
        }
    }
    return 0;
}
/*Ex11_2.cpp*/
#include <iostream>
#include <string>
using namespace std;

/*改变数组的长度*/
template <class T>
void changeLength1D(T*& a, int oldLength, int newLength)
{
     if (newLength < 0)     //新数组长度小于 0，抛出异常
     {
         throw "New length must be >= 0";
         cout << "New length must be >= 0" << endl;
     }
     T* temp = new T[newLength];  //新数组
     int number;                  //需要复制的元素数
     if (oldLength<newLength)
```

```
        number = oldLength;
    else
        number = newLength;
    for (int i=0; i<number; i++)
        temp[i] = a[i];
    delete []a;    //删除旧数组
    a = temp;     //复制新的数组
}

template <class T>
class minHeap
{
   public:
      minHeap (int initialCapacity = 10);
      ~minHeap () {delete []heap;}
      bool empty() const {return heapSize == 0;}
      int size() const {return heapSize;}
      T& top()
      {
         if (heapSize == 0)
         {
            string error = "empty heap";
            throw error;
         }
         return heap[1];
      }
      void pop();                       //删除最小元素
      void push(const T& theElement);   //插入元素
      void output();                    //输出
      void initialize(T*, int);         //构建小根堆
      void deactivateArray()
      {

         heap = NULL;
         arrayLength = heapSize = 0;
      }
   private:
      int heapSize;              //元素数量
      int arrayLength;           //队列元素加 1
      T* heap;                   //元素数组
};

/*构造函数*/
template <class T>
```

```
minHeap<T>::minHeap (int initialCapacity)
{
    if (initialCapacity <1)
    {
        cout << "wrong capacity" << endl;
        return;
    }
    arrayLength = initialCapacity + 1;
    heap = new T[arrayLength];
    heapSize = 0;
}

/*删除最小值*/
template <class T>
void minHeap<T>::pop()
{
    if (heapSize == 0)
    {
        string error = "empty heap";
        throw error;
    }
    heap[1].~T();
    T lastElement = heap[heapSize--];      //将尾节点保存并在树中删
除
    int currentNode = 1;                  //从根节点起，为其寻找位置
    int child = 2;
    while (child <= heapSize)             //child 应该是较小的孩子
    {
        if (child < heapSize && heap[child] > heap[child+1])
            child++;                      //找到较小的孩子
        if (lastElement <= heap[child])    //如果根节点较小，则是正
确的位置
            break;
        heap[currentNode] = heap[child];   //否则孩子上移一层
        currentNode = child;
        child *= 2;
    }
    heap[currentNode] = lastElement;       //正确放置尾节点
}

/*插入新值*/
template <class T>
void minHeap<T>::push (const T& theElement)
{
```

```
    if (heapSize == arrayLength-1)          //堆已满，扩容
    {
        changeLength1D (heap, arrayLength, 2*arrayLength);
        arrayLength *= 2;
    }
    int currentNode = ++heapSize;           //新插入的元素在数组最后
    while (currentNode != 1 && heap[currentNode/2] > theElement)
    {
        heap[currentNode] = heap[currentNode/2];  //新插入值较大，移
动双亲
        currentNode /= 2;                          //新值移向双亲
    }
    heap[currentNode] = theElement;                //正确放置新值
}

/*输出*/
template <class T>
void minHeap<T>::output ()
{
    for (int i=1; i<heapSize; i++)
        cout << heap[i] << ″ ″;
    cout << endl;
}

template <class T>
void minHeap<T>::initialize(T* theHeap, int theSize)
{
    delete []heap;
    heap = theHeap;
    heapSize = theSize;
    for (int root = heapSize/2; root >= 1; root--)
    {
        T rootElement = heap[root];
        int child = 2*root;
        while (child <= heapSize)
        {
            if (child < heapSize && heap[child] > heap[child+1])
                child++;
            if (rootElement <= heap[child])
                break;
            heap[child/2] = heap[child];
            child *= 2;
        }
        heap[child/2] = rootElement;
```

```
    }
}

/*二叉树结点*/
template <class E>
struct binaryTreeNode
{
    E element;        //节点元素
    binaryTreeNode<E>* leftChild;    //左儿子
    binaryTreeNode<E>* rightChild;  //右儿子
    /*构造函数*/
    binaryTreeNode()
    {rightChild = leftChild = NULL;}
    binaryTreeNode(const E& theElement):element(theElement)
    {rightChild = leftChild = NULL;}
    binaryTreeNode(const E& theElement, binaryTreeNode<E>*
theLeftChild, binaryTreeNode<E>* theRightChild):element(theElement)
    {
        rightChild = theRightChild;
        leftChild = theLeftChild;
    }
};

/*二叉树*/
template <class E>
class linkedBinaryTree
{
    public:
        linkedBinaryTree()
        {
            root = NULL;
            treeSize = 0;
        }
        ~linkedBinaryTree(){;}
        int size() const {return treeSize;}
        void makeTree(const E&, linkedBinaryTree<E>&,
linkedBinaryTree<E>&);      //建树
    protected:
        binaryTreeNode<E>* root;    //二叉树的根节点
        int treeSize;           //树的节点的个数
};

/*二叉树建树*/
template <class E>
```

```
void linkedBinaryTree<E>::makeTree (const E& theElement, linkedBinaryTree<E>& lT, linkedBinaryTree<E>& rT)
{
    root = new binaryTreeNode<E>(theElement, lT.root, rT.root);
    treeSize = lT.treeSize + rT.treeSize + 1;
    //避免从左树和右树访问
    lT.root = rT.root = NULL;
    lT.treeSize = rT.treeSize = 0;
}

/*霍夫曼树的节点*/
template <class T,class E>
struct huffmanNode
{
    linkedBinaryTree<T>* tree;
    E weight;
    operator E() const {return weight;}            //调用 T 时，返回权重
};

/*霍夫曼树*/
template <class T>
class huffmanTree
{
    public:
        huffmanTree()
        {
            stringLength = 0;
            countNum = 0;
        }
        ~huffmanTree()
        {
            tree->~linkedBinaryTree();
            delete []weight;
            delete []theElement;
        }
        int makeHuffman();              //构建霍夫曼树并获取编码长度
        void getin(string& in);         //获取输入
    private:
        linkedBinaryTree<T>* tree;      //霍夫曼树
        int stringLength;             //霍夫曼编码长度
        int countNum;                 //元素数量
        int* weight;                  //权重数组
        T* theElement;                  //元素数组
```

```
};

/*获取输入*/
template <class T>
void huffmanTree<T>::getin(string& in)
{
    int countNeeded[26] = {0};          //统计每个字母出现的次数
    int len = in.length();
    for (int i=0; i<len; i++)
        countNeeded[in[i]-97]++;     //统计到的字母的对应元素值加 1
    for (int i=0; i<26; i++)
    {
        if (countNeeded[i] != 0)
            countNum++;                 //统计出现的字母的数量
    }
    if (countNum == 1)                  //所有元素均相同
        stringLength = len;
    theElement = new char[countNum+1];
    weight = new int[countNum+1];
    int index = 0;
    for (int i=0; i<26; i++)
    {
        if (countNeeded[i] != 0)
        {
            theElement[index+1] = i+97;
            weight[index+1] = countNeeded[i];
            index++;
        }
    }
}

/*构建霍夫曼树*/
template <class T>
int huffmanTree<T>::makeHuffman ()
{
    if (countNum == 1)
        return stringLength;
    huffmanNode<T,int>* hNode = new huffmanNode<T,int>[countNum+1];
    linkedBinaryTree<T> emptyTree;          //创建 n-1 个单节点的二叉
树
    for (int i=1; i<=countNum; i++)
    {
        hNode[i].weight = weight[i];              //权重
        hNode[i].tree = new linkedBinaryTree<T>;
```

```
        hNode[i].tree->makeTree(theElement[i],emptyTree,emptyTree);
    //建树
    }
    minHeap<huffmanNode<T,int> > heap(1);
    heap.initialize(hNode,countNum);
    huffmanNode<T,int> w, x, y;
    stringLength = 0;
    for (int i=1; i<countNum; i++)
    {
        x = heap.top();
        heap.pop();
        y = heap.top();
        heap.pop();
        tree = new linkedBinaryTree<T>;
        tree->makeTree('0', *x.tree, *y.tree);
        w.weight = x.weight + y.weight;
        stringLength += w.weight;
        w.tree = tree;
        heap.push(w);
        delete x.tree;
        delete y.tree;
    }

    return stringLength;
}

int main()
{
    string in;
    cin >> in;
    huffmanTree<char> huffman;
    huffman.getin(in);
    cout << huffman.makeHuffman();
    return 0;
}
```