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：202000130198	姓名： 隋春雨	班级： 20.4
实验题目：堆及其应用		
实验学时：2	实验日期： 2021-12-2	

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。

软件开发环境：

CLION2020

1. 实验内容

①题目描述：

创建最小堆类，使用数组作为存储结构，提供操作：插入、删除、初始化、排序。保证第一个操作是建堆操作，接下来是对堆的插入和删除操作，删除和插入都在建好的堆上进行操作。

输入输出格式：

输入：

第一行一个数 n（n<=5000)，代表堆的大小；第二行 n 个数，代表堆的各个元素；第三行一个数 m (m<=1000)，代表接下来共 m 个操作。接下来 m 行，分别代表各个操作。下面是各个操作的格式：

插入操作：1 num;

删除操作：2;

排序操作：第一行两个数 3 和 n，3 代表是排序操作，n 代表待排序的数的数目，接下来一行 n 个数是待排序数。

保证排序操作只出现一次且一定是最后一个操作。

输出：

第一行建堆操作输出建好堆后的堆顶的元素。接下来 m 个操作，若是插入和删除操作。每行输出执行操作后堆顶的元素的值；若是排序操作，输出一行按升序排序好的结果，每个元素间用空格分隔。

②题目描述：哈夫曼编码。

输入输出格式：

输入：

一串小写字母组成的字符串（不超过 1000000)。

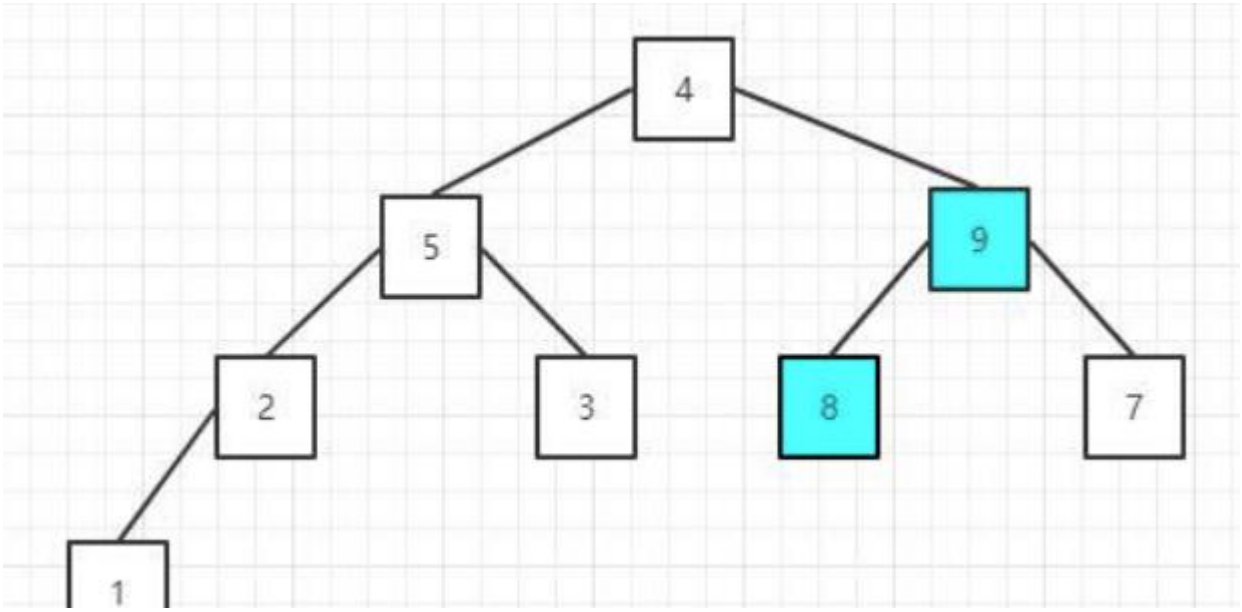
输出：

输出这个字符串通过 Huffman 编码后的长度。

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）

① A 题

1) 数据结构：选择堆。堆其实就是优先队列，大根树（小根树）是每个节点的值都大于（小于）或等于其子节点（如果有的话）值的树。大根树或 小根树节点的子节点个数可以大于 2。 如果既是大根树（小根树）又是完全二叉树，那么我们称为大根堆（小根堆）。



2) 算法：

(一)插入：类似与我们实现中的二分法，我们每一步都进行这个元素和 heap 中的元素的大小比较，如果是小于，那么我们就继续向下找，同时注意这里需要防止短路问题，要防止数组下标越界。

```
43      <T>
44      void minHeap<T>::insert(T& val) {
45          if (size + 1 == length) //要扩容
46          {
47              T* new_array = new T[2 * length];
48              copy(array, last: array + length, new_array);
49              length *= 2;
50              delete[]array;//删除原来的数组
51              array = new_array;
52          }
53          array[++size] = val;
54          int pos = size;
55          while (pos != 1 && array[pos] < array[pos / 2])//我们应该先判断pos是否为root,防止短路
56          {
57              std::swap( &: array[pos / 2], &: array[pos]);
58              pos /= 2;
59          }
60      }
```

(二)排序：使用堆排序，首先进行数组的初始化。进行一个时间复杂度 0（n）的初始化，初始化的步骤是，将每一个子树都调整成为一个堆，这样我们就需要一个 down 的操作，实现如下：

```cpp
void minHeap<T>::down(int id) {
    int pos = id;//记录id
    while (2 * pos <= size)//没有越界
    {
        int min_id = 2 * pos;//孩子ID
        if (2 * pos + 1 <= size)
        {
            if (array[2 * pos + 1] < array[2 * pos])//找到最小的孩子
            {
                min_id = 2 * pos + 1;
            }
        }
        if (array[pos] > array[min_id])//判断大小，如果大于继续进行
        {
            std::swap( &: array[pos], &: array[min_id]);
            pos = min_id;
        }
        else//否则退出
        {
            break;
        }
    }
```

我们需要每一步都进行一个取出堆顶的操作，然后保存到数组中，实现如下：

```cpp
void minHeap<T>::sort(T* _array, int n) {
    delete[]array;
    array = new int[n + 1];
    size = n;
    length = n + 1;
    for (int i = 1; i <= n; i++)
    {
        array[i] = _array[i - 1];//记录
    }
    for (int i = size / 2; i != 0; i--)
    {
        down(i);//调整位置
    }
    queue<T>q;//用队列来进行保存
    while (!empty())//非空
    {
        q.push( x: top());
        this->pop();
    }
    while (!q.empty())
    {
        cout << q.front() << " ";//输出
        q.pop();
```

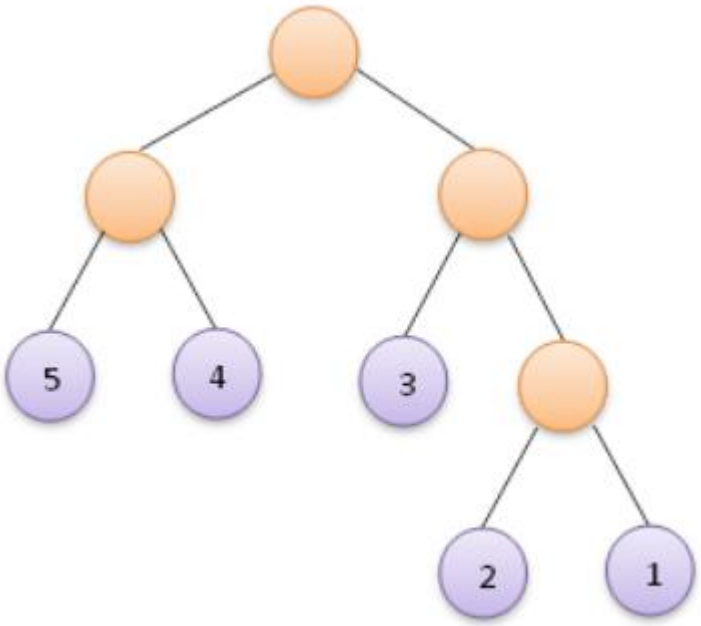
minHeap<T>::sort

(三)删除操作：首先我们需要明确如何进行删除操作，我们将数组最后的一个元素放置到堆顶，然后调整这个堆的位置。同时我们知道，需要特判堆元素的个数，如果堆此时只有一个元素，那么我们就不能用最后一个元素进行替换，此时会 RE，实现代码如下：

```
63  void minHeap<T>::pop() {
64      if (size == 0)
65      {
66          throw "the Heap is empty";
67      }
68      if (size > 1) //一般情况
69      {
70          array[1] = array[size--];
71          down( id: 1);//调整Heap
72      }
73      else //特殊情况
74      {
75          size--;//更新私有变量
76      }
77
78  }
```

②霍夫曼编码：

数据结构：霍夫曼编码使用扩充二叉树（外部节点对应于字符串中被编码的字符）进行编码。对于每一个次的合并，我们都需要找出来这个数组中的最小元素，故使用堆。我们只需要重载一下比较运算符就可以了。



算法：每一次都进行 pop 操作合并，然后对于合并出来的结点的元素值都是两个弹出来结点的权重之和。合并之后 push 进去。等只有一棵树的时候，弹出来。此时再进行一次遍历操作即可。Main 函数中实现代码如下：

```
while (Tree.size() != 1)//循环条件
{
    int leftChild = Tree.top();
    Tree.pop();
    int rightChild = Tree.top();
    Tree.pop();
    answer += leftChild;
    answer += rightChild;
    Tree.push( theElement: leftChild + rightChild);
}
cout << answer << endl;
```

3. 测试结果（测试输入，测试输出）

(1) a 题

输入：

输入

```
10
-225580 113195 -257251 384948 -83524 331745 179545 293165 125998 376875
10
1 -232502
1 -359833
1 95123
2
2
2
1 223971
1 -118735
1 -278843
3 10
-96567 37188 -142422 166589 -169599 245575 -369710 423015 -243107 -108789
```

输出：

```
-96567 37188 -142422 166589 -169599 245575 -369710 423015 -243107 -108789-257251
-257251
-359833
-359833
-257251
-232502
-225580
-225580
-225580
-278843

-369710 -243107 -169599 -142422 -108789 -96567 37188 166589 245575 423015

进程已结束，退出代码为 0
```

提交 OJ 的结果：

✓Accepted

#	Result	Score	Time	Memory
1	✓Accepted	10	0 ms	3496 KiB
2	✓Accepted	10	0 ms	3444 KiB
3	✓Accepted	10	0 ms	3404 KiB
4	✓Accepted	10	1 ms	3368 KiB
5	✓Accepted	10	1 ms	3428 KiB
6	✓Accepted	10	2 ms	3488 KiB
7	✓Accepted	10	2 ms	3564 KiB
8	✓Accepted	10	3 ms	3508 KiB
9	✓Accepted	10	4 ms	3564 KiB
10	✓Accepted	10	6 ms	3788 KiB

(2)b 题

输入：

```
abcdabcaba
```

输出：

```
C:\Users\4399\untitled115\cmake-build-debug\unti
abcdabcaba
19
```

提交 OJ 最后的结果：

✓Accepted

#	Result	Score	Time	Memory
1	✓Accepted	10	1 ms	3748 KiB
2	✓Accepted	10	1 ms	4144 KiB
3	✓Accepted	10	2 ms	4428 KiB
4	✓Accepted	10	2 ms	4656 KiB
5	✓Accepted	10	3 ms	5472 KiB
6	✓Accepted	10	3 ms	5684 KiB
7	✓Accepted	10	4 ms	6092 KiB
8	✓Accepted	10	4 ms	6556 KiB
9	✓Accepted	10	2 ms	6788 KiB
10	✓Accepted	10	5 ms	7328 KiB

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

(1) 堆排序为什么要把 a 和 heap 给分开?也就是解耦合？

我们知道大根堆退出的时候会调用析构函数，将堆中数组 heap 删除，执行 heap.initialize(a,n)时，把数组 heap 初始化为数组 a。 为避免调用堆的析构函数时将数组 a 删除，在 maxHeap 类中增加 deactiveArray()函数

(2) 对于指针的使用，一定要谨慎！比如以下这个错误，查了好几个小时。我在输入测试用例的时候，发现输出不对，在经过了二分等 debug 技巧后确定了 bug 所在的位置。

```
template<class T>
T& minHeap<T>::top() {
    if (empty())
    {
        throw "the heap is empty";
    }
    return array[1];
}
```

我发现这行代码在 return array[1]之后，它结点的值自动的改变了！按理来说它是绝对不应该改变的，后来就逐行查找，最终发现


```
array[++size] = val;
```

在 insert 的时候，对于重载的赋值操作，他是直接进行了浅层赋值。就是说，犯错的 array[3]中的左孩子指向的是 top1,而 top1 是一个局部变量！它在本轮循环结束后会自动析构，而 array[3]中的左孩子指针的值却不会更新，在下一次执行的时候，对于 top1 的新的值，地址却没有变，array[3]依然指向了它，这就导致了错误。

解决：使用深层 copy,或者使用动态分配

```
while (heap._size() > 1)
{
    treeNode<char>*top1 = new treeNode<char>(heap.top().num,heap.top().leftChild,heap.top(
    heap.pop();
    treeNode<char>*top2 = new treeNode<char>(heap.top().num,heap.top().leftChild,heap.top(
    heap.pop();
    treeNode<char>new_tree( num: top1->num + top2->num, top1, top2);
    heap.insert( &: new_tree);
}
```

(3) 在提交 oj 的时候要把自己的测试删除，比如多输出了一个换行可能就会导致全部的判错。

(4) 对于 BOOL 数组的初始化，绝对不能想当然，每一次的定义都要对其进行初始化。在本次的实验中，因为 bool 数组没有初始化导致 debug 了很久。

```
bool* jud_push = new bool[_size + 1]();
```

(5) 对于下标从 1 开始的数组，要多动态分配一块内存。因为索引为 0 的地方我们是没有访问的。

```
int* height = new int[_size + 1];
```

(6) 对于一个只有两个私有成员的 struct，我们可以直接使用 pair 来存储。简化了我们的代码量。同时对于返回多个值的函数，我们只能使用引用来返回。这也是为什么兴起了 Go 语言的一个原因。

(7) 对于 height 等操作的计算，一定要特判是否合法。因为对于 root 结点来说，它的父节点是自身，不能直接增加。

```
if(node.first.element!=node.second.element)//非根结点
{
    height[node.first.element] = max(height[node.first.element], height[node.secor
}
s.pop();
```

(8) 我们在写循环条件判断的时候，对于短路情况的判断一定要慎重，我们对于指针的使用一定要先判断是否为空，在进行取值操作，如下：

```
40      while (currentNode != NULL &&//值不等于输入，且不越界，对于NULL值的判断要放到前边
41             currentNode->element.first != theKey)
42          currentNode = currentNode->next;
```

(9) 对于维护私有变量的时候，要特殊情况特殊判：比如说删除结点的时候，要考虑这个是不是头结点，

如果是，那么更新私有变量。如下：

```
112     if (p != NULL && p->element.first == theKey)
113     {
114         if (tp == NULL) firstNode = p->next;//头结点特殊处理
115         else tp->next = p->next;
116         delete p;//删除
117         dSize--;
118     }
119 }
```

(10)要注意私有成员的更新，public 函数知道自己所处的状态都是靠着私有成员才知道的，如果没有及时更新，那数据就成了垃圾数据，没有任何意义。我在写实验的时候，也经历过没更新导致的 Bug，最终 debug 查出来，就是下面这个：

```
    // switch to newQueue and set theFront and theBack
    theFront = 2 * arrayLength - 1;//更新私有成员
    theBack = arrayLength - 2;    // queue size arrayLength - 1
    arrayLength *= 2;
    queue = newQueue;//指针赋值
}

// put theElement at the theBack of the queue
theBack = (theBack + 1) % arrayLength;
queue[theBack] = theElement;
```

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）

(1) A 题

```
1.  #include<iostream>
2.  using namespace std;
3.
4.  template<class T>
5.  class minHeap //小根堆类
6.  {
7.  public:
8.   minHeap() :heapSize(0), arrayLength(10) { heap = new T[10]; } //构造函数
9.   minHeap(T* theData, int theSize):heap(NULL){ initialize(theData, theSize); } //构造并初始化
10.  ~minHeap() { delete[] heap; } //析构函数
11.  int size() { return heapSize; } //堆元素个数
12.  bool empty() const {return heapSize == 0;} //是否为空
13.  void initialize(T* theHeap, int theSize); //初始化
```

```
14.   void push(const T& theElement); //插入元素
15.   void pop(); //弹出元素
16.   T& top() { return heap[1]; }//首元素
17.   void deactivateArray() { heap = NULL; }
18.  private:
19.   T* heap;//元素数组，一个类型为T的一维数组
20.     int arrayLength;//数组的容量
21.   int heapSize;//堆的元素个数
22.
23.  };
24.  template<class T>
25.  void minHeap<T>::initialize(T* theHeap, int theSize) //初始化一个非空小根堆
26.  {//在数组theHeap[1:theSize]中建小根堆
27.   delete[] heap;
28.   heap = theHeap;
29.   heapSize = theSize;
30.   arrayLength = theSize + 1;
31.
32.   //堆化
33.   for (int root = heapSize / 2; root >= 1; root--)
34.   {
35.    //为元素rootElement 寻找位置
36.    T rootElement = heap[root];//子树的根
37.    int child = root * 2;//child 的父节点是 rootElement 的位置
38.    while (child <= heapSize)
39.    {
40.     //heap[child]应是兄弟中较小者
41.     if (child < heapSize && heap[child] > heap[child + 1])
42.      child++;
43.     //能把rootElement 放入heap[child/2]吗
44.     if (rootElement <= heap[child])//小于两个儿子
45.      break;//可以
46.     //不可以
47.     heap[child/2] = heap[child];//把孩子上移
48.     child *= 2;//下移一层
49.    }
50.    heap[child / 2] = rootElement;
51.   }
52.  }
53.
54.  template<class T>
55.  void minHeap<T>::push(const T& theElement)
56.  {
57.   //必要时增加数组长度
58.   if (heapSize == arrayLength-1) //没有足够空间
59.   {//数组长度加倍，以下为changeLength1D 内容
```

```
60.     arrayLength = 2 * heapSize + 1;//数组长度倍增
61.     T* temp = new T[arrayLength];
62.     copy(heap+1, heap+heapSize+1, temp+1);
63.     delete [] heap;
64.     heap = temp;
65.    }
66.    heapSize++;//heapSize+1 （新叶子）
67.    //为元素 theElement 寻找插入位置
68.    int currentNode = heapSize;//currentNode 从新叶子向上移动
69.    while (currentNode > 1 && heap[currentNode/ 2] > theElement)
70.    {//当插入元素小于父节点，不能把 theElement 插在 heap[currentNode]
71.     heap[currentNode] = heap[currentNode / 2]; //元素向下移动
72.     currentNode /= 2; //移向父节点
73.    }
74.    heap[currentNode] = theElement;//插入
75.  }
76.
77.  template<class T>
78.  void minHeap<T>::pop()
79.  {
80.      //删除最小元素
81.    heap[1].~T();
82.    //删除最后一个元素，然后重新建堆
83.    T lastElement = heap[heapSize--];
84.    //从根开始，为最后一个元素寻找位置
85.    int currentNode = 1;//最后一个元素当前在根
86.    int child = 2;//尾节点的孩子节点
87.    while (child <= heapSize)
88.    {
89.    //heap[child]应该是 currentNode 的更小孩子
90.    if (heap[child + 1] < heap[child])
91.     child++;
92.    //判断可以把 theElement 放在 heap[currentNode]吗
93.    if (lastElement <= heap[child])//lastElement 小于儿子
94.     break;//可以，插入
95.    //不可以
96.    heap[currentNode] = heap[child]; //孩子上移
97.    currentNode = child;  //向下一层寻找位置
98.    child *= 2;
99.    }
100.  heap[currentNode] = lastElement;
101. }
102.
103. template<class T>
104. void heapSort(T a[], int n)
105. {//利用堆排序方法给数组 a[1:n]排序
```

```
106.    //在数组上建立小根堆
107.  minHeap<T> heap(a, n);
108.
109.  //逐个从小根堆中提取元素
110.  for (int i = n-1; i >= 1; i--)
111.  {
112.   T x = heap.top();
113.   heap.pop();
114.   a[i + 1] = x;
115.  }
116.  //从堆的析构函数中保存数组 a
117.  heap.deactivateArray();//把 a 和 heap 给分开！原来他俩绑定的（解绑）
118. }
119.
120. int main()
121. {
122.  minHeap<int> heap;
123.  int n;//n 代表堆的大小。第二行 n 个数，代表堆的各个元素
124.  int m;//m 为行数
125.  int * a;
126.  cin >> n;
127.  a = new int[n+1];
128.  for (int i = 1; i <= n; i++)
129.   cin >> a[i];//输入 n 个数
130.  heap.initialize(a, n);
131.  cout << heap.top() << endl;//建堆操作后输出栈顶元素
132.  cin >> m;//行数
133.  for(int j= 1; j<=m; j++)
134.  {
135.   int opt;
136.   int num;
137.   cin >> opt;
138.    switch(opt)
139.   {
140.        case 1:
141.            cin >> num;
142.    heap.push(num);
143.    cout << heap.top()<<endl;
144.            break;
145.        case 2:
146.       heap.pop();
147.    cout << heap.top()<<endl;
148.            break;
149.        case 3:
150.         cin >> num;
151.    int *  a=new int[num+1];
```

```
152.    for (int i = 1; i <= num; i++)
153.     cin >> a[i];
154.    heapSort(a, num);
155.    for (int i = n; i >= 1; i--)
156.     cout << a[i]<<" ";
157.         }
158.
159.  }
160.
161. }
```

(2)B 题

```
1.    #include<iostream>
2.    using namespace std;
3.    template<class T>
4.    class minHeap
5.    {
6.    public:
7.
8.     minHeap() :heapSize(0), arrayLength(10) { heap = new T[10]; } //构造函数
9.     minHeap(T* theData, int theSize):heap(NULL){ initialize(theData, theSize); } //构造并初始化
10.    minHeap(int theSize)//构造已知元素个数的堆
11.    {
12.          arrayLength = theSize + 1;
13.          heap = new T[arrayLength];
14.          heapSize = 0;
15.      }
16.    ~minHeap() { delete[] heap; } //析构函数
17.    int size() { return heapSize; } //堆元素个数
18.    bool empty() const {return heapSize == 0;} //是否为空
19.    void initialize(T* theHeap, int theSize); //初始化
20.    void push(const T& theElement); //插入元素
21.    void pop(); //弹出元素
22.    T& top() { return heap[1]; }//首元素
23.    void deactivateArray() { heap = NULL; }
24.   private:
25.      T* heap;//元素数组，一个类型为T 的一维数组
26.      int arrayLength;//数组的容量
27.    int heapSize;//堆的元素个数
28.   };
29.
30.
```

```
template<class T>
void minHeap<T>::initialize(T* theHeap, int theSize) //初始化一个非空小根堆
{//在数组 theHeap[1:theSize] 中建小根堆
 delete[] heap;
 heap = theHeap;
 heapSize = theSize;
 arrayLength = theSize + 1;

 //堆化
 for (int root = heapSize / 2; root >= 1; root--)
 {
  //为元素 rootElement 寻找位置
  T rootElement = heap[root];//子树的根
  int child = root * 2;//child 的父节点是 rootElement 的位置
  while (child <= heapSize)
  {
   //heap[child] 应是兄弟中较小者
   if (child < heapSize && heap[child] > heap[child + 1])
    child++;
   //能把 rootElement 放入 heap[child/2] 吗
   if (rootElement <= heap[child])//小于两个儿子
    break;//可以
   //不可以
   heap[child/2] = heap[child];//把孩子上移
   child *= 2;//下移一层
  }
  heap[child / 2] = rootElement;
 }
}

template<class T>
void minHeap<T>::push(const T& theElement)
{
 //必要时增加数组长度
 if (heapSize == arrayLength-1) //没有足够空间
 {//数组长度加倍，以下为 changeLength1D 内容
  arrayLength = 2 * heapSize + 1;//数组长度倍增
  T* temp = new T[arrayLength];
  copy(heap+1, heap+heapSize+1, temp+1);
  delete [] heap;
  heap = temp;
 }
 heapSize++;//heapSize+1 （新叶子）
 //为元素 theElement 寻找插入位置
 int currentNode = heapSize;//currentNode 从新叶子向上移动
 while (currentNode > 1 && heap[currentNode/ 2] > theElement)
```

```
77.  {//当插入元素大于父节点，不能把 theElement 插在 heap[currentNode]
78.   heap[currentNode] = heap[currentNode / 2]; //元素行下移动
79.   currentNode /= 2; //移向父节点
80.  }
81.  heap[currentNode] = theElement;//插入
82. }
83.
84. template<class T>
85. void minHeap<T>::pop()
86. {
87.    //删除最小元素
88.  heap[1].~T();
89.  //删除最后一个元素，然后重新建堆
90.  T lastElement = heap[heapSize--];
91.  //从根开始，为最后一个元素寻找位置
92.  int currentNode = 1;//最后一个元素当前在根
93.  int child = 2;//尾节点的孩子节点
94.  while (child <= heapSize)
95.  {
96.  //heap[child]应该是 currentNode 的更小孩子
97.  if (heap[child + 1] < heap[child])
98.   child++;
99.  //判断可以把 theElement 放在 heap[currentNode]吗
100.  if (lastElement <= heap[child])//lastElement 小于儿子
101.   break;//可以，插入
102.  //不可以
103.  heap[currentNode] = heap[child]; //孩子上移
104.  currentNode = child;  //向下一层寻找位置
105.  child *= 2;
106. }
107. heap[currentNode] = lastElement;
108. }
109.
110.
111. int main(){
112.    string theString;//要输入的字符串
113.    cin >> theString;
114.    int occurrenceOfLetters[26] = {0};
115.    int stringLength = theString.length();//获取字符串长度
116.    minHeap<int> huffmanTreeElements(stringLength); //建立大小为字符串长度的小根堆
117.    for(int i = 0; i < stringLength; i ++) //统计 26 个字幕出现的次数，并存入数组 occurrenceOfLetters
118.  {
119.        int j = theString[i] - 'a';
120.   occurrenceOfLetters[j] ++;
121.    }
122.    for(int i = 0; i < 26; i ++) //如果字母出现次数不是 0，把它的出现次数放入小根堆
```

```
{
        if(occurrenceOfLetters[i] != 0)
 {
 huffmanTreeElements.push(occurrenceOfLetters[i]);
 }
  }
  //“建树”并进行计算
  int answer = 0; //要输出的答案
  while(huffmanTreeElements.size() != 1) //除非只剩一个，否则还处于“建树”过程中
{
 //每次弹出两个最小的合并
      int leftChild = huffmanTreeElements.top();
      huffmanTreeElements.pop();
      int rightChild = huffmanTreeElements.top();
      huffmanTreeElements.pop();
      answer += leftChild;
      answer += rightChild;
      //为何直接加：每合并了一次就相当于增加了从外部节点到跟的一条线，也就增加了一次字符串长度
      //生成的新节点再去合并
      huffmanTreeElements.push(leftChild + rightChild);
  }
  cout << answer << endl;//输出答案

  return 0;
}
```