

数据结构与算法 课程实验报告

学号：202000130143	姓名： 郑凯饶	班级： 计科 20.1
实验题目：堆及其应用		
实验学时：2	实验日期： 1201	
<p>实验目的：</p> <ol style="list-style-type: none"> 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。 		
<p>软件开发环境：</p> <p>Windows 10 家庭中文版 64 位 (10.0, 版本 18363)</p> <p>Dev-C++ IDE</p>		
<ol style="list-style-type: none"> 1. 实验内容 2. 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。 3. 哈夫曼编码 4. 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） <p>1></p> <pre> template<class T> class minHeap { public: minHeap(int ini = 10); ~minHeap() { delete [] heap; } bool empty() const { return Size == 0; } int size() const { return Size; } const T& top() { if (Size == 0) { } return heap[1]; } void pop(); void push(const T&); void initialize(T *, int); void deactivate() { heap = NULL; Length = Size = 0; } void output(ostream& out) const; private: int Size; int Length; T * heap; }; </pre> <p>Private:</p> <p>Heap: 由于堆是完全二叉树，数组存储可以很好地利用空间，并且访问简单</p> <p>Public:</p> <p>Top(): 返回堆顶元素，即 heap[1]</p> <p>Pop(): 弹出堆顶元素，具体实现如下：</p>		

```

template<class T>
void minHeap<T>::pop() {
    if (Size == 0) { return; }
    heap[1].~T();

    T last = heap[Size--];
    int cur = 1, child = 2;

    // 向下遍历
    while (child <= Size) {
        if (child < Size && heap[child] > heap[child + 1]) child++;

        if (last <= heap[child]) break;
        heap[cur] = heap[child];
        cur = child;
        child *= 2; // cur = child / 2
    }
    heap[cur] = last;
}

```

1

将下标为 Size 的元素取代堆顶元素，并“下沉”，选择当前节点及其左右孩子的最小者换至当前元素位置，由于是完全二叉树节点和孩子的关系为 $child_1 = curNode * 2$ 和 $child_2 = curNode * 2 + 1$ ，以上为“下沉”细节。

Push()：往小根堆中加入元素，将节点添至数组最后位置，然后执行“上浮”操作，与 pop() 操作相似。

Initialize()：O(n) 初始化小根堆，具体实现如下：

```

template<class T>
void minHeap<T>::initialize(T *theheap, int theSize) {
    delete [] heap;
    heap = theheap;
    Size = theSize;
    // How about Length ?
    // it assumes: theheap.Length == heap.Length

    // 堆化
    for (int root = Size / 2; root >= 1; root--) {
        T rEle = heap[root];

        int child = 2 * root;
        while (child <= Size) {
            if (child < Size && heap[child] > heap[child + 1]) child++;
            if (rEle <= heap[child]) break;

            heap[child / 2] = heap[child];
            child *= 2;
        }
        heap[child / 2] = rEle;
    }
}

```

从最后一个有孩子的节点开始，“下沉”。

复杂度具体计算：

在树的第 i 层，最多有 2^{i-1} 个节点，它们的高度为 $h_i = h - j + 1$ 。

于是初始化的时间为

$$O\left(\sum_{j=1}^{h-1} 2^{j-1}(h-j+1)\right) = O(n)$$

2>

这里基本使用课程网站上的标程。

统计最终编码长度可以在霍夫曼树的建树过程中实现。

我们知道，最终编码长度等于 WEP，它的计算方式为

$$WEP = \sum_{i=1}^n L(i) * F(i)$$

实际对应叶子节点的权重乘深度。

建树过程中的每次合并使得被合并节点的深度+1，最终长度应为每次合并节点的权重之和求和。

5. 测试结果（测试输入，测试输出）

在 OJ 平台上成功提交。

6. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

这次实验比较简单，我尝试使用局部的静态变量避免了全局变量的使用 hhhhh 另一方面，尝试计算了一下时间复杂度。

7. 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）

```
8. #include<bits/stdc++.h>
9. using namespace std;
10.
11. template<class T>
12. class minHeap {
13. public:
14.     minHeap(int ini = 10);
15.     ~minHeap() { delete [] heap; }
16.     bool empty() const { return Size == 0; }
17.     int size() const { return Size; }
18.     const T& top() {
19.         if (Size == 0) { }
20.         return heap[1];
21.     }
22.
23.     void pop();
24.     void push(const T&);
25.     void initialize(T *, int);
26.
27.     void deactivate() {
28.         heap = NULL; Length = Size = 0;
29.     }
30.     void output(ostream& out) const;
31. private:
32.     int Size;
33.     int Length;
34.     T * heap;
35. };
36.
37. template<class T>
38. minHeap<T>::minHeap(int iniCap) {
39. // if
40. Length = iniCap + 1; // **
41. heap = new T[Length];
42. Size = 0;
43. }
```

```
44.
45. template<class T>
46. void changeLength1D(T*& a, int oldLength, int newLength)
47. {
48. //   if (newLength < 0)
49. //       throw illegalParameterValue("new Length must be >= 0");
50.
51.   T* temp = new T[newLength];           // new array
52.   int number = min(oldLength, newLength); // number to copy
53.   copy(a, a + number, temp);
54.   delete [] a;                          // deallocate old memory
55.   a = temp;
56. }
57.
58. template<class T>
59. void minHeap<T>::push(const T& ele) {
60.   if (Size == Length - 1) {
61.     changeLength1D(heap, Length, 2 * Length);
62.     Length <<= 1;
63.   }
64.
65.   int cur = ++Size;
66.
67.   // 向上起泡
68.   while (cur != 1 && heap[cur / 2] > ele) {
69.     heap[cur] = heap[cur / 2];
70.     cur /= 2;
71.   }
72.   heap[cur] = ele;
73. }
74.
75. template<class T>
76. void minHeap<T>::pop() {
77.   if (Size == 0) { return; }
78.   heap[1].~T();
79.
80.   T last = heap[Size--];
81.   int cur = 1, child = 2;
82.
83.   // 向下遍历
84.   while (child <= Size) {
85.     if (child < Size && heap[child] > heap[child + 1]) child++;
86.
87.     if (last <= heap[child]) break;
88.     heap[cur] = heap[child];
89.     cur = child;
```

```

90.  child *= 2; // cur = child / 2
91. }
92. heap[cur] = last;
93.
94.}
95.
96.template<class T>
97.void minHeap<T>::initialize(T *theheap, int theSize) {
98.  delete [] heap;
99.  heap = theheap;
100.   Size = theSize;
101.   // How about Length ?
102.   // it assumes: theheap.Length == heap.Length
103.
104.   // 堆化
105.   for (int root = Size / 2; root >= 1; root--) {
106.       T rEle = heap[root];
107.
108.       int child = 2 * root ;
109.       while (child <= Size) {
110.           if (child < Size && heap[child] > heap[child + 1]) child++;
111.           if (rEle <= heap[child]) break;
112.
113.           heap[child / 2] = heap[child];
114.           child *= 2;
115.       }
116.       heap[child / 2] = rEle;
117.   }
118. }
119.
120. template<class T>
121. void heapSort(T a[], int n) { // ** 原地重排 **
122.     minHeap<T> heap(n);
123.     heap.initialize(a, n);
124.
125.     for (int i = n - 1; i >= 1; i--) {
126.         T x = heap.top(); heap.pop(); // heap.Size = i then we can put the min ele
            on the (i + 1) th.
127.         a[i + 1] = x;
128.     }
129.     heap.deactivate();
130. }
131.
132. void solve() {
133.     int n; cin >> n;
134.     int *a = new int[n + 1];

```

```

135.     for (int i = 1; i <= n; i++) {
136.         cin >> a[i];
137.     }
138.
139.     minHeap<int> hp(n);
140.     hp.initialize(a, n);
141.     cout << hp.top() << endl;
142.
143.     int m; cin >> m;
144.     for (int i = 0, opt, num; i < m; i++) {
145.         cin >> opt;
146.         if (opt == 1) {
147.             cin >> num;
148.             hp.push(num);
149.             cout << hp.top() << endl;
150.         }
151.         else if (opt == 2) {
152.             hp.pop();
153.             cout << hp.top() << '\n';
154.         }
155.         else if (opt == 3) {
156.             int sz; cin >> sz;
157.             int *b = new int[sz + 1];
158.             for (int j = 1; j <= sz; j++) cin >> b[j];
159.             heapSort(b, sz);
160.             for (int j = sz; j >= 1; j--) cout << b[j] << " ";
161.             cout << endl;
162.         }
163.     }
164. }
165.
166. int main() {
167.     solve();
168.     return 0;
169. }
170.
171.
172.
173.
174.
175. #include<bits/stdc++.h>
176. using namespace std;
177.
178. // huffman 树本质上是二叉树，其构造是特殊的二叉树建树过程
179. template <class T>
180. struct binaryTreeNode

```

```

181.  {
182.      T element;
183.      binaryTreeNode<T> *leftChild,    // Left subtree
184.                          *rightChild; // right subtree
185.
186.      binaryTreeNode() {leftChild = rightChild = NULL;}
187.      binaryTreeNode(const T& theElement):element(theElement)
188.      {
189.          leftChild = rightChild = NULL;
190.      }
191.      binaryTreeNode(const T& theElement,
192.                      binaryTreeNode *theLeftChild,
193.                      binaryTreeNode *theRightChild)
194.          :element(theElement)
195.      {
196.          leftChild = theLeftChild;
197.          rightChild = theRightChild;
198.      }
199.  };
200.
201.
202.  template<class E>
203.  class linkedBinaryTree // : public binaryTree<binaryTreeNode<E> >
204.  {
205.      public:
206.          linkedBinaryTree() {root = NULL; treeSize = 0;}
207.          ~linkedBinaryTree(){erase();};
208.          bool empty() const {return treeSize == 0;}
209.          int size() const {return treeSize;}
210.          E* rootElement() const;
211.          void makeTree(const E& element,
212.                        linkedBinaryTree<E>&, linkedBinaryTree<E>&);
213.          linkedBinaryTree<E>& removeLeftSubtree();
214.          linkedBinaryTree<E>& removeRightSubtree();
215.          void preOrder(void(*theVisit)(binaryTreeNode<E>*))
216.              {visit = theVisit; preOrder(root);}
217.          void inOrder(void(*theVisit)(binaryTreeNode<E>*))
218.              {visit = theVisit; inOrder(root);}
219.          void postOrder(void(*theVisit)(binaryTreeNode<E>*))
220.              {visit = theVisit; postOrder(root);}
221.          void levelOrder(void(*)(binaryTreeNode<E> *));
222.          void preOrderOutput() {preOrder(output); cout << endl;}
223.          void inOrderOutput() {inOrder(output); cout << endl;}
224.          void postOrderOutput() {postOrder(output); cout << endl;}
225.          void levelOrderOutput() {levelOrder(output); cout << endl;}
226.          void erase()

```

```

227.         {
228.             postOrder(dispose);
229.             root = NULL;
230.             treeSize = 0;
231.         }
232.         int height() const {return height(root);}
233.     protected:
234.         binaryTreeNode<E> *root;           // pointer to root
235.         int treeSize;                     // number of nodes in tree
236.         static void (*visit)(binaryTreeNode<E> *); // visit function
237.         static int count;                 // used to count nodes in a subtree
238.         static void preOrder(binaryTreeNode<E> *t);
239.         static void inOrder(binaryTreeNode<E> *t);
240.         static void postOrder(binaryTreeNode<E> *t);
241.         static void countNodes(binaryTreeNode<E> *t)
242.         {
243.             visit = addToCount;
244.             count = 0;
245.             preOrder(t);
246.         }
247.         static void dispose(binaryTreeNode<E> *t) {delete t;}
248.         static void output(binaryTreeNode<E> *t)
249.             {cout << t->element << ' ';}
250.         static void addToCount(binaryTreeNode<E> *t)
251.             {count++;}
252.         static int height(binaryTreeNode<E> *t);
253.     };
254.     // the following should work but gives an internal compiler error
255.     template <class E> void (*linkedBinaryTree<E>::visit)(binaryTreeNode<E> *);
256.     // so the explicit declarations that follow are used for our purpose instead
257.     //void (*linkedBinaryTree<int>::visit)(binaryTreeNode<int> *);
258.     //void (*linkedBinaryTree<booster>::visit)(binaryTreeNode<booster> *);
259.     //void (*linkedBinaryTree<pair<int,int> >::visit)(binaryTreeNode<pair<int,int> > *);
260.     //void (*linkedBinaryTree<pair<const int,char> >::visit)(binaryTreeNode<pair<const int,char> > *);
261.     //void (*linkedBinaryTree<pair<const int,int> >::visit)(binaryTreeNode<pair<const int,int> > *);
262.
263.     template<class E>
264.     E* linkedBinaryTree<E>::rootElement() const
265.     {// Return NULL if no root. Otherwise, return pointer to root element.
266.         if (treeSize == 0)
267.             return NULL; // no root
268.         else
269.             return &root->element;

```



```

270.     }
271.
272.     template<class E>
273.     void linkedBinaryTree<E>::makeTree(const E& element,
274.         linkedBinaryTree<E>& left, linkedBinaryTree<E>& right)
275.     {// Combine left, right, and element to make new tree.
276.         // left, right, and this must be different trees.
277.         // create combined tree
278.         root = new binaryTreeNode<E> (element, left.root, right.root);
279.         treeSize = left.treeSize + right.treeSize + 1;
280.
281.         // deny access from trees left and right
282.         left.root = right.root = NULL;
283.         left.treeSize = right.treeSize = 0;
284.     }
285.
286.     template<class E>
287.     linkedBinaryTree<E>& linkedBinaryTree<E>::removeLeftSubtree()
288.     {// Remove and return the left subtree.
289.         // check if empty
290.         // if (treeSize == 0)
291.         // throw emptyTree();
292.
293.         // detach left subtree and save in leftSubtree
294.         linkedBinaryTree<E> leftSubtree;
295.         leftSubtree.root = root->leftChild;
296.         count = 0;
297.         leftSubtree.treeSize = countNodes(leftSubtree.root);
298.         root->leftChild = NULL;
299.         treeSize -= leftSubtree.treeSize;
300.
301.         return leftSubtree;
302.     }
303.
304.     template<class E>
305.     linkedBinaryTree<E>& linkedBinaryTree<E>::removeRightSubtree()
306.     {// Remove and return the right subtree.
307.         // check if empty
308.         // if (treeSize == 0)
309.         // throw emptyTree();
310.
311.         // detach right subtree and save in rightSubtree
312.         linkedBinaryTree<E> rightSubtree;
313.         rightSubtree.root = root->rightChild;
314.         count = 0;
315.         rightSubtree.treeSize = countNodes(rightSubtree.root);

```

```

316.     root->rightChild = NULL;
317.     treeSize -= rightSubtree.treeSize;
318.
319.     return rightSubtree;
320. }
321.
322. template<class E>
323. void linkedBinaryTree<E>::preOrder(binaryTreeNode<E> *t)
324. { // Preorder traversal.
325.     if (t != NULL)
326.     {
327.         linkedBinaryTree<E>::visit(t);
328.         preOrder(t->leftChild);
329.         preOrder(t->rightChild);
330.     }
331. }
332.
333. template<class E>
334. void linkedBinaryTree<E>::inOrder(binaryTreeNode<E> *t)
335. { // Inorder traversal.
336.     if (t != NULL)
337.     {
338.         inOrder(t->leftChild);
339.         linkedBinaryTree<E>::visit(t);
340.         inOrder(t->rightChild);
341.     }
342. }
343.
344. template<class E>
345. void linkedBinaryTree<E>::postOrder(binaryTreeNode<E> *t)
346. { // Postorder traversal.
347.     if (t != NULL)
348.     {
349.         postOrder(t->leftChild);
350.         postOrder(t->rightChild);
351.         linkedBinaryTree<E>::visit(t);
352.     }
353. }
354.
355. template <class E>
356. void linkedBinaryTree<E>::levelOrder(void(*theVisit)(binaryTreeNode<E> *))
357. { // Level-order traversal.
358.     queue<binaryTreeNode<E>*> q;
359.     binaryTreeNode<E> *t = root;
360.     while (t != NULL)
361.     {

```

```

362.         theVisit(t); // visit t
363.
364.         // put t's children on queue
365.         if (t->leftChild != NULL)
366.             q.push(t->leftChild);
367.         if (t->rightChild != NULL)
368.             q.push(t->rightChild);
369.
370.         // get next node to visit
371.         // try {t = q.front();}
372.         // catch (queueEmpty) {return;}
373.
374.         if (!q.empty()) {
375.             t = q.front();
376.         }
377.         else return;
378.
379.         q.pop();
380.     }
381. }
382.
383. template <class E>
384. int linkedBinaryTree<E>::height(binaryTreeNode<E> *t)
385. { // Return height of tree rooted at *t.
386.     if (t == NULL)
387.         return 0; // empty tree
388.     int hl = height(t->leftChild); // height of left
389.     int hr = height(t->rightChild); // height of right
390.     if (hl > hr)
391.         return ++hl;
392.     else
393.         return ++hr;
394. }
395.
396.
397. // 最小堆
398. template<class T>
399. void changeLength1D(T*& a, int oldLength, int newLength)
400. {
401.     // if (newLength < 0)
402.     //     throw illegalParameterValue("new length must be >= 0");
403.
404.     T* temp = new T[newLength]; // new array
405.     int number = min(oldLength, newLength); // number to copy
406.     copy(a, a + number, temp);
407.     delete [] a; // deallocate old memory

```

```

408.     a = temp;
409. }
410.
411. template<class T>
412. class minHeap // : public minPriorityQueue<T>
413. {
414.     public:
415.         minHeap(int initialCapacity = 10);
416.         ~minHeap() {delete [] heap;}
417.         bool empty() const {return heapSize == 0;}
418.         int size() const
419.             {return heapSize;}
420.         const T& top()
421.             {// return min element
422.              //         if (heapSize == 0)
423.              //         throw queueEmpty();
424.              return heap[1];
425.             }
426.         void pop();
427.         void push(const T&);
428.         void initialize(T *, int);
429.         void deactivateArray()
430.             {heap = NULL; arrayLength = heapSize = 0;}
431.         void output(ostream& out) const;
432.     private:
433.         int heapSize;           // number of elements in queue
434.         int arrayLength;       // queue capacity + 1
435.         T *heap;               // element array
436. };
437.
438. template<class T>
439. minHeap<T>::minHeap(int initialCapacity)
440. {// Constructor.
441.  // if (initialCapacity < 1)
442.  // {ostream s;
443.  //   s << "Initial capacity = " << initialCapacity << " Must be > 0";
444.  //   throw illegalParameterValue(s.str());
445.  // }
446.  arrayLength = initialCapacity + 1;
447.  heap = new T[arrayLength];
448.  heapSize = 0;
449. }
450.
451. template<class T>
452. void minHeap<T>::push(const T& theElement)
453. {// Add theElement to heap.

```

```

454.
455.     // increase array length if necessary
456.     if (heapSize == arrayLength - 1)
457.     { // double array length
458.         changeLength1D(heap, arrayLength, 2 * arrayLength);
459.         arrayLength *= 2;
460.     }
461.
462.     // find place for theElement
463.     // currentNode starts at new leaf and moves up tree
464.     int currentNode = ++heapSize;
465.     while (currentNode != 1 && heap[currentNode / 2] > theElement)
466.     {
467.         // cannot put theElement in heap[currentNode]
468.         heap[currentNode] = heap[currentNode / 2]; // move element down
469.         currentNode /= 2;                          // move to parent
470.     }
471.
472.     heap[currentNode] = theElement;
473. }
474.
475. template<class T>
476. void minHeap<T>::pop()
477. { // Remove max element.
478.     // if heap is empty return null
479.     // if (heapSize == 0) // heap empty
480.     //     throw queueEmpty();
481.
482.     // Delete min element
483.     heap[1].~T();
484.
485.     // Remove Last element and reheapify
486.     T lastElement = heap[heapSize--];
487.
488.     // find place for lastElement starting at root
489.     int currentNode = 1,
490.         child = 2; // child of currentNode
491.     while (child <= heapSize)
492.     {
493.         // heap[child] should be smaller child of currentNode
494.         if (child < heapSize && heap[child] > heap[child + 1])
495.             child++;
496.
497.         // can we put lastElement in heap[currentNode]?
498.         if (lastElement <= heap[child])
499.             break; // yes

```

```

500.
501.     // no
502.     heap[currentNode] = heap[child]; // move child up
503.     currentNode = child;             // move down a level
504.     child *= 2;
505. }
506.     heap[currentNode] = lastElement;
507. }
508.
509. template<class T>
510. void minHeap<T>::initialize(T *theHeap, int theSize)
511. { // Initialize max heap to element array theHeap[1:theSize].
512.     delete [] heap;
513.     heap = theHeap;
514.     heapSize = theSize;
515.
516.     // heapify
517.     for (int root = heapSize / 2; root >= 1; root--)
518.     {
519.         T rootElement = heap[root];
520.
521.         // find place to put rootElement
522.         int child = 2 * root; // parent of child is target
523.                                // location for rootElement
524.         while (child <= heapSize)
525.         {
526.             // heap[child] should be smaller sibling
527.             if (child < heapSize && heap[child] > heap[child + 1])
528.                 child++;
529.
530.             // can we put rootElement in heap[child/2]?
531.             if (rootElement <= heap[child])
532.                 break; // yes
533.
534.             // no
535.             heap[child / 2] = heap[child]; // move child up
536.             child *= 2;                     // move down a level
537.         }
538.         heap[child / 2] = rootElement;
539.     }
540. }
541.
542. template<class T>
543. void minHeap<T>::output(ostream& out) const
544. { // Put the array into the stream out.
545.     copy(heap + 1, heap + heapSize + 1, ostream_iterator<T>(cout, " "));

```

```

546.     }
547.
548.     // overload <<
549.     template <class T>
550.     ostream& operator<<(ostream& out, const minHeap<T>& x)
551.         {x.output(out); return out;}
552.
553.
554.     template<class T>
555.     struct huffmanNode {
556.         linkedBinaryTree<int> *tree;
557.         T weight;
558.
559.         operator T () const { return weight; }
560.     };
561.
562.     //int Ans = 0;
563.
564.     template <class T>
565.     linkedBinaryTree<int>* huffmanTree(T weight[], int n, bool op)
566.     {
567.         static int Ans = 0;
568.         if (op) {
569.             cout << Ans << '\n';
570.             return NULL;
571.         }
572.
573.         // Generate Huffman tree with weights weight[1:n], n >= 1.
574.         // create an array of single node trees
575.         huffmanNode<T> *hNode = new huffmanNode<T> [n + 1];
576.         linkedBinaryTree<int> emptyTree;
577.         for (int i = 1; i <= n; i++)
578.         {
579.             hNode[i].weight = weight[i];
580.             hNode[i].tree = new linkedBinaryTree<int>;
581.             hNode[i].tree->makeTree(i, emptyTree, emptyTree);
582.         }
583.
584.         // make node array into a min heap
585.         minHeap<huffmanNode<T> > heap(1);
586.         heap.initialize(hNode, n);
587.
588.         // repeatedly combine trees from min heap
589.         // until only one tree remains
590.         huffmanNode<T> w, x, y;
591.         linkedBinaryTree<int> *z;

```

```

592.     for (int i = 1; i < n; i++)
593.     {
594.         // remove two lightest trees from the min heap
595.         x = heap.top(); heap.pop();
596.         y = heap.top(); heap.pop();
597.
598.         // combine into a single tree
599.         z = new linkedBinaryTree<int>;
600.         z->makeTree(0, *x.tree, *y.tree);
601.         w.weight = x.weight + y.weight;
602.         Ans += w.weight;
603.         //     cout << Ans << '\n';
604.         w.tree = z;
605.         heap.push(w);
606.         delete x.tree;
607.         delete y.tree;
608.     }
609.
610.     // destructor for min heap deletes hNode
611.     return heap.top().tree;
612. }
613.
614.
615. void solve() {
616.     string s; cin >> s;
617.     int len = s.length();
618.
619.     // if (len == 1) {
620.     //     cout << "1\n";
621.     //     return;
622.     // }
623.
624.     int *cnt = new int[27];
625.     int *w = new int[27];
626.
627.     // memset(cnt, 0, cnt + 27); // 1 - 26
628.     for (int i = 1; i < 27; i++) cnt[i] = 0;
629.
630.     for (int i = 0; i < len; i++) {
631.         cnt[s[i] - 'a' + 1] ++;
632.     }
633.
634.     // 离散化
635.     int one = 1;
636.     for (int i = 1; i < 27; i++) {
637.         if (cnt[i] != 0) {

```



```
638.     w[one] = cnt[i];
639.     one ++;
640. }
641. }
642.
643. // 不发生合并
644. if (one == 2) {
645.     cout << len << '\n';
646.     return;
647. }
648.
649. // 建立含 26 个小写字母节点的霍夫曼树 kill
650.
651. huffmanTree(w, one - 1, 0);
652.
653. huffmanTree(w, one - 1, 1);
654. }
655.
656.
657. int main(){
658.     solve();
659.     return 0;
660. }
```