```
 1
 2 #ifndef HUFFMAN_YOUR_HUFFMAN_CODE_H
 3 #define HUFFMAN_YOUR_HUFFMAN_CODE_H
 4
 5 #include <map>
 6 #include "huffman_helper.h"
 7 using namespace std;
 8
 9 struct TreeNode {
10     char ch;
11     int weight;
12     TreeNode *left;
13     TreeNode *right;
14
15     TreeNode(char cha) {
16         ch = cha;
17         left == nullptr;
18         right == nullptr;
19     }
20     TreeNode() {
21         left == nullptr;
22         right == nullptr;
23     }
24 };
25 struct CompareTreeNode
26 {
27     bool operator()(const TreeNode* lhs, const TreeNode* rhs) const
28     {
29         return lhs->weight > rhs->weight;
30     }
31 };
32 /* You need the unusual CompareTreeNode struct above if you want to make
33  * a priority queue of TreeNodes.  (Hint: You do!)  This struct defines
   an
34  * operator for comparing TreeNode*, which makes it possible for the
35  * underlying heap for a priority queue to work correctly.  It's weird,
36  * but here's the syntax you'll need:
37        priority_queue<TreeNode*, vector<TreeNode*>, CompareTreeNode> pq;
38  * The first parameter describes what it is a priority queue of, the
39  * second parameter describes the underlying heap implementation ("I'm
40  * using a vector for this heap"), and the third parameter specifies a
41  * way to compare TreeNode*.  Phew...
42  */
43
44 // NOTE: The struct EncodedData is defined in the huffman_helper.h file
45 void destroyTree(TreeNode* tree) {
46     if (tree->left == nullptr && tree->right == nullptr)
47         return;
48     destroyTree(tree->left);
49     destroyTree(tree->right);
```

```
50      delete tree;
51  }
52  map<char, int> makeFreqMap(const string &text) {
53      map<char, int> result;
54      for(char ch : text) {
55          result[ch]++;
56      }
57
58      return result;
59  }
60  priority_queue<TreeNode*, vector<TreeNode*>, CompareTreeNode> makeQueue(
    map<char,int> freqMap) {
61      priority_queue<TreeNode*, vector<TreeNode*>, CompareTreeNode> result;
62
63      for(auto i : freqMap) {
64          TreeNode* tree = new TreeNode();
65          tree->ch = i.first;
66          tree->weight = i.second;
67          // cout << tree->ch << ": " << tree->weight << endl;
68          result.push(tree);
69      }
70
71      return result;
72  }
73
74  priority_queue<TreeNode*, vector<TreeNode*>, CompareTreeNode>
    huffmanEncode(priority_queue<TreeNode*, vector<TreeNode*>,
    CompareTreeNode> pq) {
75      while(pq.size() > 1) {
76          TreeNode* tree = new TreeNode();
77          tree->left = pq.top();
78          pq.pop();
79          tree->right = pq.top();
80          pq.pop();
81
82          tree->weight = tree->left->weight + tree->right->weight;
83          pq.push(tree);
84      }
85      return pq;
86  }
87
88  map<char, queue<Bit>> makeEncodingMap(TreeNode* tree, queue<Bit> &code,
    map<char, queue<Bit>> &result) {
89      if(tree->left == nullptr && tree->right == nullptr) {
90          result[tree->ch] = code;
91      }
92      else {
93          queue<Bit> qL = code;
94          qL.push(0);
95          makeEncodingMap(tree->left, qL, result);
```

```
 96         queue<Bit> qR = code;
 97         qR.push(1);
 98         makeEncodingMap(tree->right, qR, result);
 99     }
100     return result;
101 }
102
103 queue<Bit> encodeFile(map<char, queue<Bit>> encodingMap, string &text) {
104     queue<Bit> result;
105     for(char i : text) {
106         queue<Bit> code = encodingMap.at(i);
107         while(!code.empty()) {
108             result.push(code.front());
109             code.pop();
110         }
111     }
112     return result;
113 }
114 void flattenTree(queue<Bit> &encodedTreeShape, queue<char> &
    encodedTreeLeaves, TreeNode* tree){
115     if(tree->left == nullptr && tree->right == nullptr) {
116         encodedTreeShape.push(0);
117         encodedTreeLeaves.push(tree->ch);
118     }
119     else{
120         encodedTreeShape.push(1);
121         flattenTree(encodedTreeShape, encodedTreeLeaves, tree->left);
122         flattenTree(encodedTreeShape, encodedTreeLeaves, tree->right);
123     }
124
125 }
126
127 EncodedData compress(string text) {
128     EncodedData result;
129     // you've got a lot to write here
130     map<char, int> freqMap = makeFreqMap(text);
131
132     priority_queue<TreeNode*, vector<TreeNode*>, CompareTreeNode> pq =
    makeQueue(freqMap);
133     pq = huffmanEncode(pq);
134
135     TreeNode* tree = pq.top();
136     queue<Bit> q;
137     map<char, queue<Bit>> encodingMap;
138     encodingMap = makeEncodingMap(tree, q, encodingMap);
139
140     queue<Bit> encodedMessageBits = encodeFile(encodingMap, text);
141
142     queue<Bit> encodedTreeShape;
143     queue<char> encodedTreeLeaves;
```

```
144     flattenTree(encodedTreeShape, encodedTreeLeaves, tree);
145 
146     result.messageBits = encodedMessageBits;
147     result.treeLeaves = encodedTreeLeaves;
148     result.treeShape = encodedTreeShape;
149 
150     destroyTree(tree);
151 
152     return result;
153 }
154 
155 TreeNode* makeTree(queue<Bit> &treeShape, queue<char> &treeLeaves) {
156     Bit bit = treeShape.front();
157     treeShape.pop();
158 
159     if(bit == 0) {
160         TreeNode* tree = new TreeNode();
161         tree->ch = treeLeaves.front();
162         treeLeaves.pop();
163         return tree;
164     }
165     else {
166         TreeNode* tree = new TreeNode();
167         tree->left = makeTree(treeShape, treeLeaves);
168         tree->right = makeTree(treeShape, treeLeaves);
169         return tree;
170     }
171 }
172 
173 
174 string decodeCharacter(queue<Bit> &messageBits, TreeNode* head) {
175     string result = "";
176 
177     if(head->left == nullptr && head->right == nullptr) {
178         string a;
179         a.push_back(head->ch);
180         return a;
181     }
182     Bit bit = messageBits.front();
183     messageBits.pop();
184     if(bit == 0) {
185         return result + decodeCharacter(messageBits, head->left);
186     }
187     else {
188         return result + decodeCharacter(messageBits, head->right);
189 
190     }
191 
192 }
193 
```

```
194 string decompress(EncodedData& data) {
195     TreeNode* tree = makeTree(data.treeShape, data.treeLeaves);
196     string result;
197 
198     while(!data.messageBits.empty()) {
199         result += decodeCharacter(data.messageBits, tree);
200     }
201     destroyTree(tree);
202 
203     return result;
204 }
205 
206 #endif //HUFFMAN_YOUR_HUFFMAN_CODE_H
207 
```