Project No.4: Huffman Compression and Decompression

Name	Student No.	Email	Responsibility
李志容	14346009	978282388@qq.com	Parts of codes, parts of the reports
谭笑	14346022	296309711@qq.com	Parts of codes, parts of the reports

Data: 2016.4.24

1. Introduction

Huffman encoding is a way to assign binary codes to symbols that reduces the overall number of bits used to encode a typical string of those symbols. It is based on the frequency of occurrence of a data item. The principle is to use a lower number of bits to encode the data that occurs more frequently. Codes are stored in a code file which may be constructed for a series of symbols. In all cases the code file plus encoded data must be transmitted to enable decoding. In this project, we only consider the case that the input file only contains a series of alphabets.

2. Analysis and Design

1) Data structure

The Huffman_node struct is used to store the information of each letter, including its frequency, id(what's it), code(binary code) and its left, right and parent node. Use a fixed size array(in our project, its maximum size is 26) to store nodes' information since the types of symbols are limited so a fixed array is effective to store nodes. A priority queue is used to sort the nodes according to the frequency of the letters. Obviously the automated sorting function in priority queue is convenient in this project. Initially, open the input file, and count the frequency of occurrence of each symbol. To build the Huffman Tree, we use binary node list. Each node contains a left and right pointer pointing to its child. During coding, using nodes and their pointers is very effective, so we decide to use the binary node list.

2) Important algorithms

a) Create Huffman Tree

While there is more than one node in the queue, remove the node of highest priority twice to get two nodes; Then create a new internal node with these two nodes as children and with frequency equal to the sum of the two nodes' frequencies; After that add the new node to the queue.

Finally the only node left is the root node and the tree is complete.

Importantly, some special cases should be considered. If the queue is empty, there will be no operation; if the queue has only one node, the node will be the root; if the queue has more than one node, build the tree.

Here is the pseudocode:

Algorithm 1 Create A Huffman Tree

```
1: procedure create Huffman tree(())
2:     pq = priority queue
3:     if Size(pq) = 0 then
4:         null → root
5:         return
6:     end if
7:     if Size(pq) = 1 then
8:         Top(pq) → root
9:         return
10:    end if
11:    while Size(pq) > 1 do
12:        new nodes p1, p2, temp
13:        two tops in pq → p1, p2
14:        freq(p1 + p2) → freq(temp)
15:        p1, p2 parent ← temp
16:        push temp → pq
17:    end while
18:    root = Top(pq)
19: end procedure
```

Justification: This algorithm is essential in this project. It has successfully considered all cases that will happen, like blank input file or only one type of symbol in the file. During the procedure, two elements in the queue are popped and then a new node is pushed into the queue. Eventually there will be only one node in the queue, and that is the root. This is effective and less space consuming.

b) Calculate Huffman codes

After building the Huffman tree, we need to calculate the Huffman codes for each symbol. Our method is to traverse the constructed binary tree from root to leaves assigning and accumulating a '0' for the left branch and a '1' for the right branch at each node. The accumulated zeros and ones at each leaf constitute the Huffman codes for those symbols and weights.

Specially, we use recursive method to traverse the tree.

Here is the pseudocode:

Algorithm 2 Calculate Huffman Codes

```
1: procedure calculate huffman codes(&root)
2:     if root == NULL then
3:         return
4:     end if
5:     if root.left ≠ NULL and root.right ≠ NULL then
6:         root.left.code ← root.code + "0"
7:         root.right.code ← root.code + "1"
8:         calculate huffman codes(root.left)
9:         calculate huffman codes(root.right)
10:    else
11:        return
12:    end if
13: end procedure
```

Justification: This algorithm is not hard to understand since it is a traverse of a binary tree. In addition, using recursion brings the advantage that the codes are concise and easy to read.

c) Rebuild the Huffman Tree

When encoding, the program produces the code file, which saves the symbols as well as their corresponding Huffman codes. Now we are trying to decode the file that contains a series of Huffman codes, and we must rebuild the Huffman Tree using the code file.

First and foremost, read the code file. Read a symbol and its corresponding Huffman code. Create a new node and save it to the array and queue.

Secondly, fetch a node from the queue and read its Huffman code one by one. If a '0' is read, create a left child node for current sub root if it does not have one; if a '1' is read, create a right child node for current sub root if it does not have one. When you create a child node for the root, the child node will be the current sub root. Then repeat reading the codes and create corresponding child nodes until the end. Then pop the element and fetch next node from the queue if it is not NULL. Again, repeat the operations until the queue is empty.

Eventually, the Huffman Tree is rebuilt.

Here is the pseudocode for the second procedure above:

Algorithm 3 Rebuild the Huffman Tree

```
1:  procedure rebuild the Huffman tree
2:      pq = priority queue
3:      while pq not empty do
4:          new node subroot ← root
5:          new node p ← Top(pq)
6:          code ← p.code
7:          for i = 0 to code.length - 1 do
8:              new node temp
9:              if code[i] == '0' then
10:                 if subroot.left == NULL then
11:                     subroot.left ← temp
12:                 end if
13:                 subroot ← subroot.left
14:             else
15:                 if subroot.right == NULL then
16:                     subroot.right ← temp
17:                 end if
18:                 subroot ← subroot.right
19:             end if
20:         end for
21:     end while
22:     p → subroot
23: end procedure
```

Justification: This algorithm satisfies the requirement that the code of each node corresponds to the root of the tree to the right leaf. Any two leaves of the tree will not be the same because after using the code of the node to build its path in the tree, it will be popped from the queue.

d) Huffman decompression

After rebuilding the Huffman tree, now we start to decode. First input the file, and read the char one by one. Then turn the ASCAII char to 8 bits binary strings. Next, traverse the binary string. Start from the root, When a '0' is read, traverse to the left path and when a '1' is read, traverse to the right path. When a leaf is reached, output the id of it, which is the symbol corresponding to the previous codes. Then starting from the root again and read the codes. Repeat the operation until the end is read.

Here is the pseudocode:

Algorithm 4 Huffman Decoding

```
1: procedure Huffman decoding( )
2:     open file
3:     char c
4:     node sub
5:     while file.read ≠ file.end do
6:         file.input c
7:         string s = toBinary(c,8)
8:         for  i = 0 to s.length - 1 do
9:             if s[i] == '0' then
10:                if sub.left ≠ NULL then
11:                    sub = sub.left
12:                end if
13:            else
14:                if sub.right ≠ NULL then
15:                    sub = sub.right
16:                end if
17:            end if
18:            if sub.id ≠ NULL then
19:                output sub.id
20:                sub ← root
21:            end if
22:        end for
23:    end while
24: end procedure
```

Justification: This algorithm can decode the char bits accurately since each judgment brings the right path for the symbol. However, what if the last char didn't contain 8 bits, that is, when we compress the binary codes to char, total number of binary bits is not divided evenly by 8? In order to solve this problem, we set the first two char of the decode file to be specific symbols. The first char means how many bits the last char of the decode file actually represent. The second char means what the value of the last char is. (Please see it dentally in our source code)

3) Time complexity

Encode:

Suppose there are n symbols in the input file. We have known that the average height of a binary tree is h = log n. That means, for average cases, when encoding a symbol, it needs to go through average path of length (log (n)). Therefore, for all of n symbols, total time complexity can be approximately calculated by T (n) = O (n log n).

Decode:

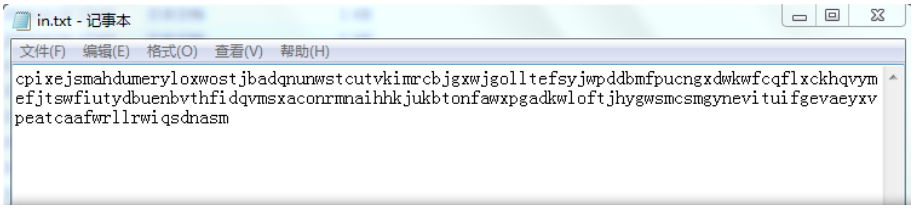
Suppose there are n chars to be decoded. For every char, when it is turned to binary bits, it at most traverses 8 nodes. It is easy to get the estimation that T(n) = 8n = O(n), linearly.

3. Test

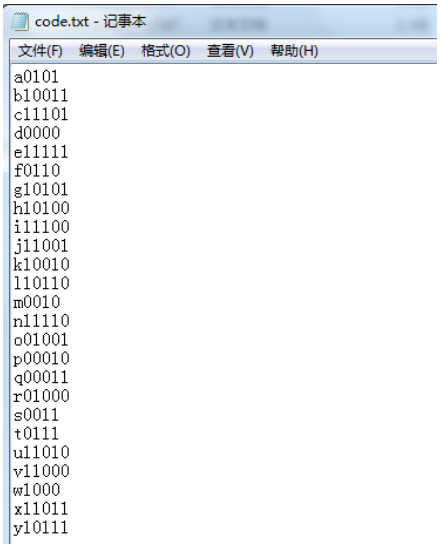
1) huffmanCoding: input the in_file_name and out_file_name.

```
D:\My Documents\Visual Studio 2012\Projects\huffman1\Debug>huffman1 in out
```

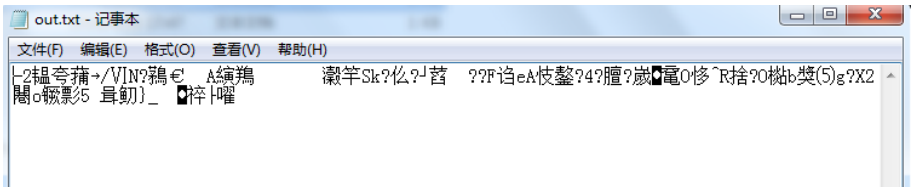
in.txt generated:



code.txt generated:



out.huf generated:

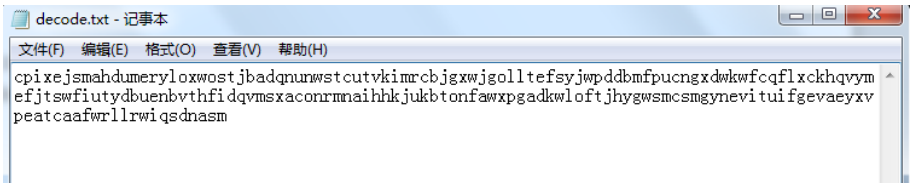


We can check out.huf corresponding to in.txt through code.txt. They are coherent.

2) huffmanDecoding: input out_file_name and decode_name.

```
D:\My Documents\Visual Studio 2012\Projects\huffman1\Debug>huffman1 out decode
```

decode.txt generated:



out.huf and code.txt are the same as the above huffmanCoding files generated. We can check this decode.txt is exactly the same as the above in.txt.

4. Conclusion and Discussion

Achieved: We have achieved the implementation of huffman class. We have achieved Huffman compression and Huffman decompression by Huffman Tree and generated in_file, out_file, code_file and decode_file.

Not achieved: We only achieve that compress and decompress the symbols containing alphabets.

Highlights: In huffmanCoding.cpp, we generate the out.txt by producing 200 alphabets randomly.

5. Appendices

Refer to the attachment “codes-huffman.docx”.

6. References

1) 《数据结构与算法实验实践教程》——乔海燕、蒋爱军、高集荣、刘晓铭

2) 《Introduction to Programming with C++》——Y. Daniel Liang

3) http://baike.baidu.com/link?url=Cop2D05waL6II6F2ejUojdj9nfmNqNB7-Qr5fXbIpUDp628vM3felg6pXtPAbM6SBEqKGuwRRwX2lu4KZSsRApiiCYliz3WWuJRM0686T-nx252DbGxRGsBL4yMje7G0EWaZZu33oCgokIEpakIHE8eN8yBAl15fpZwGxVnFRw7

4) http://www.cplusplus.com/reference/queue/priority_queue/

5) http://www.cplusplus.com/reference/queue/queue/

6) http://www.cplusplus.com/reference/string/basic_string/basic_string/

7) http://baike.baidu.com/link?url=76R0AnqqKSngZWYTeHy1YLG18JjfqQPP-Lsdqum6Puj9ERrtDmAtKePxJszhOYn9ZM7B5AN7mtkcVBFoxc32Sq