10802EE 398000 Algorithms

Homework 9. Encoding ASCII Texts

● Introduction

The American Standard Code for Information Interchange (**ASCII**) is a standard encoding method for English letters and symbols. **Each letter takes a fixed number of bits (8 bits)** to represent it. Since there are some letters not used as often as others, this fixed length encoding scheme may not be the most efficient in information storage or exchange.

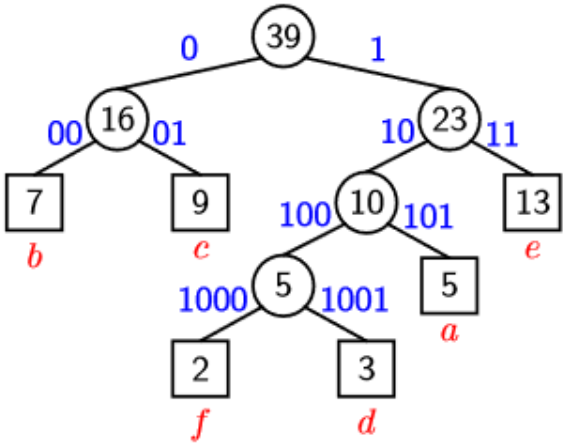
To increase the storage efficiency, **the variable length encoding scheme** can be adopted. That is, each letter may take different number of bits to represent it. The **Huffman encoding scheme** is a kind of it. It has been shown to achieve minimum storage requirement for any given set of letters.

The idea of Huffman encoding is that **symbols that are used more frequently should be shorter while symbols that appear more rarely can be longer**. This way, the number of bits it takes to encode a given message will be shorter, on average, than if a fixed length code was used.

We can use the **Binary Merge Tree algorithm (5.3.4, lec53, class note)** to construct Huffman codes. The algorithm views each letter as a leaf node initially (also called **external node**). For each iteration, we pick two nodes with the least frequency, remove them from the list, and merge them to make a new node with the frequency equal to the sum of the two. Then we insert it into the list. After using this algorithm, we can construct the binary tree (also called **Huffman tree** in this case) where **the node with less frequency has the larger depth and the node with higher frequency has the**

less depth. **The depth is corresponding to the number of bits of the Huffman code of the letter.** Thus, we can achieve our goal: symbols that are used more frequently should be shorter while symbols that appear more rarely can be longer.

Here is an example of Huffman encoding: suppose 6 letters, A = {a,b,c,d,e,f} are used to store some information and the corresponding frequencies are also known, F = {5,7,9,3,13,2}. Then the **Binary Merge Tree algorithm (5.3.4, lec53, class note)** constructs the following binary tree shown below (**Figure 1, left**) such that $\sum_{i=1}^{n} d_i f_i$ **is minimum, where d_i is the depth of the leave node and f_i is the frequency.** From the tree, we get the Huffman encoding table (**Figure 1, right**).



letter	code
a	101
b	00
c	01
d	1001
e	11
f	1000

Figure 1 the binary tree (Huffman tree) of the example and its Huffman encoding table.

The goal of this homework is to construct the Huffman code corresponding to each given text file and show the ratio between the Huffman encoded storage and the standard ASCII storage. The ratio is defined below (Equation (1)):

$$ratio = 100 * \frac{total\ Bytes\ of\ the\ input\ file\ using\ Huffman\ coding}{total\ Bytes\ of\ the\ input\ file\ using\ ASCII} \quad (1)$$

Where total bytes of the input file using ASCII is the amount of characters in the text, and total bytes

of the input file using Huffman coding is defined below (Equation (2)):

$$total\ Bytes\ of\ the\ input\ file\ using\ Huffman\ coding = \left\lceil \frac{\sum_{i=1}^{n} d_i f_i}{8} \right\rceil \quad (2)$$

Where $\sum_{i=1}^{n} \boldsymbol{d_i f_i}$ is the total bits using Huffman coding which has already explained.

In **Approach** part, I will explain the data structure I use in the code and show algorithms of how I deal with the input file, how I construct the data structure, how I make the Huffman tree, how I find Huffman codes of nodes from traveling the Huffman tree, and how I calculate the ratio. Also, I will analyze their space and time complexities.

In **Result** part, I will show the number of characters read, the number of bits and bytes needed for Huffman Coding, and the ratio of each input file. Last, I will show the CPU time of "making the Huffman tree part" and demonstrate some observations in this homework.

● Approach

The pseudocode of the main function is shown in the next page. There are five steps: first, I read the input file and construct the list (pointer array). Each element of it points to the node with a letter in the text (it's a leaf node of the Huffman tree). Second, I transform the list into a min heap. Third, I make the Huffman tree. Fourth, I make Huffman codes. Last, I print the results. I will explain these five steps more thoroughly in the following content.

```
// main function for this homework.
// Input: the text from stdin
// Output: list[1:N] points to leaf nodes initially, the root of the Huffman Tree, t is the CPU time
//         of line 6, totalCh is the total characters of the text, totalBits and totalBytes is the total
//         bits and bytes using Huffman coding, ratio = 100.0 * totalBytes / totalCh.
1 Algorithm main(void)
2 {
3       constructList(totalCh);
4       constructMinHeap(list, N);
5       t := GetTime();
6       root := Tree(N, list);
7       t := GetTime() - t;
8       makeHuffman(root, 0, code);
9       printAndCal(root, 0, totalBits);
10      totalBytes := ⌈totalBits ÷ 8⌉;
11      ratio := 100.0 * totalBytes / totalCh;
12      write (totalCh, totalBits, totalBytes, ratio, t);
13      freeTree(root);
14      free the memory of list;
15 }
```

Algorithm 1 main function for this homework.

I use the structure shown in **Algorithm 2** to store every different character which has appeared in the text. This structure can view as a binary tree node with a left child and a right child.

```
typedef struct _List {          // the structure of an input (it's a node)
    char ch;                    // character
    int frequency;              // the frequency of the letter
    struct _List* lchild;       // the left child
    struct _List* rchild;       // the right child
    char* huffman;              // the corresponding Huffman code
    int depth;                  // the depth of this node
} List;                         // the new data type List
List** list;    // pointer array points to leaf nodes initially: index from 1!
```

Algorithm 2 The structure to store the input data.

I also store the character which this node represents, the frequency of this character, its Huffman code, and its depth. Most importantly, I use a pointer array, **list[1 : N]**, to store pointers which point to these nodes initially. To transform the input text into tree nodes and generate **list[1 : N]**, constructList function is implemented and shown below:

```
// Make the new tree node.
// Input: input file from stdin.
// Output: the pointer array list[1 : N], total: total amounts of characters.
1 Algorithm constructList (total)
2 {
3       N := 0;                  // initialize the number of elements in the list
4       _total := 0;             // initialize the total amount of characters
5       for i := 0 to 255 do     // initialize f array
6             f[i] := 0;
7       c := get a character from stdin;
8       while (c is not EOF) do {
9             f[c] := f[c] + 1;          // add 1 to the frequency of a character whose ASCII is c
10            _total := _total + 1;      // add 1 to the total amount of characters
11            c := get a character from stdin;
12      }
13      for i := 0 to 255 do {    // calculate N: number of elements of the list
14            if f[i] is not 0 then
15                  N := N + 1;    // the character i is in the list
16      }
17      j := 1;
18      for i := 0 to 255 do {
19            if f[i] is not 0 then {
20                  list[j] := makeNode(i, f[i]); // make a node of character i and let list[j] points to it
21                  j := j + 1;
22            }
23      }
24      total := _total;
25 }
```

Algorithm 3 The pseudocode of constructList.

The most important part is the f array. It is used to calculate the frequency. **f[i] is the frequency of a character whose ASCII is i**. f[0 : 255] array is initialized to 0 at line 5 to line 6. Then we read the character from the input text iteratively, and add one to f[character]. Since a character is actually an integer from 0 to 255, we can directly use a character as an index.

makeNode is the function used to generate the new tree node with _ch (the character) and _frequency (the frequency of character _ch). The algorithm is shown below.

```
// Make the new tree node.
// Input: _ch: character of the node, _frequency: the frequency of the node.
// Output: the new node.
1 Algorithm makeNode (_ch, _frequency)
2 {
3       newNode := new List;
4       newNode->ch := _ch;
5       newNode->frequency := _frequency;
6       newNode->lchild := NULL;
7       newNode->rchild := NULL;
8       newNode->huffman := NULL;
9       newNode->depth := -1;
10      return newNode;
11 }
```

Algorithm 4 The pseudocode of makeNode.

The second step is to construct the min heap using list[1 : N]. Please note that the initial index of list[1 : N] is actually 1 in the code. The constructMinHeap and Heapify are implemented and shown in the next page. The most important part is that since A is a pointer array, we have to use "A[i]->frequency" in Heapify part. The overall space complexity of constructMinHeap is O(n) and the time complexity is O(n lg n) because the time complexity of Heapify is O(lg n).

```
// Make A[1 : n] into Min Heap.
// Input: pointer Array A with n elements
// Output: pointer Array A with the frequencies of the nodes of it obey Min Heap properties.
1 Algorithm constructMinHeap (A, n)
2 {
3       for i := ⌊n/2⌋ to 1 step −1 do // Initialize A[1 : n] to be a min heap.
4           Heapify(A, i, n);
5 }
```

Algorithm 5 The pseudocode of constructMinHeap.

```
// To enforce min heap property for n-element heap A with root i.
// Input: size n min heap pointer array A, root i
// Output: updated A.
1 Algorithm Heapify(A,i,n)
2 {
3       j := 2×i;           // A[j] is the lchild.
4       item := A[i];
5       done := false ;
6       while ((j ≤ n) and ( not done )) do { // A[j + 1] is the rchild.
7           if ((j < n) and (A[j]->frequency > A[j + 1]->frequency)) then
8               j := j + 1; // A[j] is the smaller child.
9           if (item->frequency <= A[j]->frequency) then // If less than children, done.
10              done := true ;
11          else { // Otherwise, continue.
12              A[⌊j / 2⌋] := A[j];
13              j := 2×j;
14          }
15      }
16      A[⌊j / 2⌋] := item;
17 }
```

Algorithm 6 Heapify psedocode

The third step is the most important step. I have already explained in **Introduction** that **Binary Merge Tree algorithm (5.3.4, lec53, class note)** can be used to generate a Huffman tree. All we

have to do now is to make it fit in the data structure I have created (list[1 : n] where list is a pointer array points to the leaf nodes initially (also called **external nodes**), and it obeys the min heap properties). The corresponding Algorithm is shown below:

```
// Generate binary merge tree from list of n leaf nodes initially.
// Input: int n, A[1:n]: an array of pointers
// Output: the root of Huffman tree
1 Algorithm Tree(n, A)
2 {
3       total := n;                              // the size of Min Heap
4       for i := 1 to (n−1) do {
5             pt := makeNode(EOF, 0);       // EOF must not the character of external nodes
6             pt→ rchild := HeapRmMin (A, total--);       // Find and remove min from A
7             pt→ lchild := HeapRmMin (A, total--);
8             pt→ frequency := (pt→ lchild) → frequency+ (pt→ rchild) → frequency;
9             HeapInsert(A, ++total, pt);
10      }
11      return pt;      // return the root of Huffman Tree
12 }
```

Algorithm 7 The pseudocode of Tree.

The merging process will repeat n – 1 times. For each iteration, we make the new node (the node created after merging, also called an **internal node**) whose character is EOF which must not in the final word list (this can be used to identify whether it is an internal node or an external node). Then we remove the node with the least frequency twice from A array and make them be the left child and the right child of the new node. Last, we let the frequency of the new node equal to the sum of the frequency of its children and insert it into the A array.

The overall space complexity of Tree is O(n) and the time complexity is O(n lg n). Because A is represented by a **min heap, both HeapRmMin and HeapInsert can be done in O(lg n) time**.

The pseudocodes of HeapRmMin and HeapInsert are shown below:

```
// Remove and return the minimum of the heap pointer array A[1 : n].
// Input: min heap pointer array A, int n
// Output: the node with min frequency and updated A.
1 Algorithm HeapRmMin(A, n)
2 {
3       if (n = 0) then error (" heap is empty! ");
4       x := A[1];
5       A[1] := A[n];
6       Heapify(A,1,n−1);
7       return x ;
8 }
```

Algorithm 8 The pseudocode of HeapRmMin.

```
// Insert the n-th element, item, to the min heap, A.
// Input: heap array A, int n, item
// Output: updated A.
1 Algorithm HeapInsert(A, n, item)
2 {
3       i := n;   // initialization
4       A[n] := item;
5       while ((i > 1) and (A[⌊i/2⌋]->frequency > item->frequency)) do {
6             A[i] := A[⌊i/2⌋]; // parent should be smaller.
7             i := ⌊i/2⌋;
8       }
9       A[i] := item;
10 }
```

Algorithm 9 The pseudocode of HeapInsert.

The fourth step is to travel the Huffman tree and generate the Huffman code of each node. From the example illustrated in **Introduction** part (Figure 1), the edge from a parent to a right child is labeled 1, and the edge from a parent to a left child is labeled 0. The code for a character is its path to

the node, starting from the root. This encoding process can be done by makeHuffman which is shown below:

```
// Make the Huffman code of each node in the Huffman tree
// Input: root of the Huffman tree, the last character index of code string, the Huffman code string
// Output: the tree with updated nodes
1 Algorithm makeHuffman(root, back, code)
2 {
3       if (root = NULL) then return;    // termination condition
4       root->huffman := code;
5       code[back] := '0';      // go to the left child
6       back := back + 1;
7       code[back] := '\0';      // the end of code
8       makeHuffman(root->lchild, back, code);
9       back := back – 1;       // undo
10      code[back] := '1';      // go to the right child
11      back := back + 1;
12      code[back] := '\0';      // the end of code
13      makeHuffman(root->rchild, back, code);
14      back := back – 1;       // undo
15 }
```

Algorithm 10 The pseudocode of makeHuffman.

makeHuffman is a preorder traverse. Before going to the left child, we push '0' to the back of the code. Before going to the right child, we push '1' to the back of the code. It is really important to do back = back – 1 (line 9 and line 14) after returning from the previous call.

The final step is to print the Huffman codes (printAndCal shown in the next page). I use the inorder traverse. If the root->ch is not EOF, then root->ch is the character in the input text. Thus, print its Huffman code and update total bits.

```
// Print Huffman codes of leaf nodes of the Huffman tree and calculate the total bits (∑_{i=1}^{n} d_i f_i)
// Input: root of the Huffman tree, the depth of the current stage
// Output: print Huffman code of leaf nodes, total bits
1 Algorithm printAndCal(root, _depth, _totalBits)
2 {
3       if (root = NULL) then return;    // termination condition
4       printAndCal (root->lchild, _depth + 1, _totalBits);
5       if (root->ch is not EOF) then { // the leaf node (ch = EOF must not a leaf node)
6             write (root->huffman);
7             root->depth := _depth;
8             _totalBits := _totalBits + (_depth * root->frequency);
9       }
10      printAndCal (root->rchild, _depth + 1, _totalBits);
11 }
```

Algorithm 11 The pseudocode of printAndCal.

After outputting results, we have to free allocated memory. The freeTree is used to free the memory of the Huffman tree. The most important part is that we must use the **postorder traverse** to avoid the segmentation fault.

```
// free the Huffman tree
// Input: root of the Huffman tree
// Output: none
1 Algorithm freeTree(root)
2 {
3       if (root = NULL) then return;    // termination condition
4       freeTree(root->lchild);
5       freeTree(root->rchild);
6       free the memory of root->huffman;
7       free the memory of root;
8 }
```

Algorithm 12 The pseudocode of freeTree.

The overall space complexities of makeHuffman, printAndCal, and freeTree are all max{O(lg V), O(V)} = O(V) where V is the number of nodes of the Huffman tree and lg V is the

recursion depth. The time complexity are all O(V) because we have to visit all nodes.

Algorithm	Overall Space Complexity	Time Complexity
constructMinHeap	O(n)	O(n lg n)
Tree	O(n)	O(n lg n)
makeHuffman	O(V)	O(V)
printAndCal	O(V)	O(V)
freeTree	O(V)	O(V)

Table 1 space and time complexities. Please note that n is the number of different characters in the text, V is the number of nodes in the Huffman tree.

- **Results**

	talk1.txt	talk2.txt	talk3.txt	talk4.txt	talk5.txt
Number of Chars read	11949	17654	15944	11014	11802
Huffman Coding needs	53830 bits 6729 bytes	79601 bits 9951 bytes	70812 bits 8852 bytes	50144 bits 6268 bytes	53813 bits 6727 bytes
Ratio	56.3143%	56.3668%	55.5193%	56.9094%	56.9988%
CPU Time of Tree function	2.78950e-05 seconds	2.81334e-05 seconds	2.71797e-05 seconds	2.88486e-05 seconds	2.69413e-05 seconds

Table 2 results table.

The interesting point in this homework is that the Huffman code is not unique. It depends on the order of the leaf nodes in the list[1 : n] initially. The order will be different if there are some nodes with the same frequency. Thus, how the person construct list will affect. Using stable or unstable sort will cause the different results, too. Although the Huffman code of each character is not unique, the total bits and bytes of Huffman coding needs will be the same.