Time Complexity of File Compression

Roll no. 1: 2018-CS-123

Roll no. 2: 2018-CS-121

Analysis Of Algorithm

Project: File Compression

Time Complexity of File Compression

Below is the pseudocode for building Huffman tree for file compression

```
Huffman Tree(ch, freq)
ch = [0...n]
freq = [0....n] //arranged in ascending order
sum = 0 ---------- (1)
node* s, tpt r, pptr  //pointer pointing to node
---------- (1)
pptr = s = node   //pointer pptr and s pointing to node---------- (1)
s.data = sum ---------- (1)
if freq[] = NULL
   return ---------- (1)
for i = 0 to n ---------- (n+1)
  create new_node
  tptr = new_node   ----------   (n)
  tptr.data = ch[i] ----------   (n)
tptr.freq = freq[i] ----------  (n)
```

```
 sum = sum + freq[i]
 if s.left = NULL or s.right = NULL
   if tptr.freq > s.freq and s.right = NULL
     s.freq = sum
     s.right = tptr

   else if  s.left = NULL
          s.freq = sum
           s.left = tptr
 else
  create new_node_2 ----------------------------------------------
---------------------------------------------------------------------- (n)
  new_node_2.data = sum ----------------------------------(n)

  s = new_node_2 ------------------------------------------------ (n)
  if tptr.freq < s.freq  ------------------------------------------------  (n)
    s.left = tptr  ------------------------------------------------  (n)
    s.right = pptr  ------------------------------------------------  (n)
    pptr = s  ------------------------------------------------ (n)


  else
    s.right = tptr
  return tree
```

Time Complexity for building Huffman tree:

=5*1+n+1+n+n+n+n+n+n+n+n+n+n+n+n+6

= 5 +8+ 13n

= 13 +13n

=O(n)

2) Below is the psuedocode that generates code from Huffman tree. Tree will be passed as the parameter to the function.

Huffman Code(Tree)

i = 0, array[], node* tptr, head ----------(1)

head = root node ---------- (1)

tptr = head=headnode ---------- (1)

if tptr.left!=Null ---------- (1)

tptr = tptr.left ---------- (1)

array[i] = 0 ---------- (1)

while tptr.left!=NULL and tptr.right!=NULL **----------(n+1)**

if(tptr.left != NULL) ----------(n)

i++ ---------- (n)

tptr= tptr.left ---------- (n)

array[i] = 0 ---------- (n)

else

i++ ---------- (n)

tptr = tptr.right ---------- (n)

array[i] = 1 ---------- (n)

write character freq and code to a file ----------(n)

i = 0, tptr = head ---------- (1)

if tptr.right !=NULL ---------- (1)

tptr= tptr.right ---------- (1)

```
array[i] = 1 ---------------------------------------------------- (1)
while tptr.left != NULL and tptr.right != NUL----------(n+1)
  if tptr.left.left == NULL and tptr.left.right == NULL
    i++

    array[i] = 0

   ** copy character, freq and array[] code in file**
    tptr = tptr.right

  else if tptr.right.left == NULL and tptr.right.right == NULL
    i++
    array[i] = 0
    **copy character, freq and array[] code in file**
    tptr = tptr.left
  if tptr.left != NULL

     tptr=tptr.right
     i++
     array[i] = 1

else
     tptr=tptr.left

     i++
     array[i] = 0
```

Time Taken to generate code from Huffman tree :

The while loop will execute till the length of the tree, hence, time complexity will be O (log n)

3) Below is the pseudocode for decoding compressed text into original file.

```
Decoder(tree, code_char)
  char = ''  ---------- (1)
  tptr = root_node ---------- (1)
        if tree=NULL
        Return
  while(!eof())          //for file1   ----------  (n+1)
  if code_char = '0'  ---------- (n)
 tptr= tptr.left  ---------- (n)
  else if code_char = '1'  ----------          ----------
----------(n)
            tptr = tptr.right  ---------- (n)


  if tptr.left = NULL and tptr.right = NULL ----------
---------- (n)
    write to new file      //file2  ---------- (n)
  else if tptr.left != NULL and tptr.right != NULL ----------
---------- (n)
    read new char from file     //file 1  ---------- (n)
```

Time taken for decoding compressed text:

=1+1+n+1+n+n+n+n+n+n+n+n

=2n+3

=O(n)

****BONUS TASK:****

Merge Files (string)

count1 = 0, count0=0 ---------- (1)

if string = “”

return ---------- (1)

n = string.length ---------- (1)

for i = 0 to n ---------- (n+1)

if string[i] = “1” and string[i+1] != “0”

increment count1----------

---------- (n)

if string[i+1] = “0” ---------- (n)

increment count1 ----------(n)

if count1=>1 ---------- (n)

write to file count1 and “1” ----------(n)

count1 = 0 ---------- (n)

else if string[i] = 0 and string[i+1] != “1”----------

---------- (n)

increment count0 ---------- (n)

if string[i+1] = “1” ---------- (n)

increment count0 ---------- (n)

if(count0>1) ---------- (n)

write to file count0 and “0”

count0 = 0 ---------- (n)

write to file string[i]

*also copy coded scheme file data to this file after above code completion**---------- (n)

Total time taken to merging files : 1+n+1+n+n+n+n+n+n+n+n+n+n+n+n+n_

=3n+2= O(N)

To copy the coded scheme file, while loop is used. **n** times statements are executed so complexity of loop will also be **n**

Total time complexity is calculated

O(n)+O(log n)+O(n)

=O(N)

CORRECTNESS

Huffman Tree (ch, freq)

```
1    sum = 0, node* s, tpt r, pptr  //pointer pointing to node
2    create node
3    pptr = s = node   //pointer pptr and s pointing to node
4    s.data = sum
5    if freq [] = NULL
6        return
7    for i = 0 to n
8      create new_node      // left, right, character, frequency = NULL
9      tptr = new_node
10       tptr.data = ch[i]
11       tptr.freq = freq[i]
12        sum = sum + freq[i]
13       if s.left = NULL or s.right = NULL
```

```
14                  if tptr.freq > s.freq and s.right = NULL
15                      s.freq = sum
16                      s.right = tptr
17                  else if  s.left = NULL
18                          s.freq = sum
19                          s.left = tptr
20          else
21              create new_node_2
22              new_node_2.data = sum
23               s = new_node_2
24          if tptr.freq < s.freq
25               s.left = tptr
26               s.right = pptr
27               pptr = s
28           else
29              s.right = tptr
30      return tree
```

Suppose we have an array of characters C = [A …..Z] and we have a frequency array of each character represented by F = [0 …n]. the frequency is arranged in increasing order and so the corresponding character is stored in character array C. character array also involves blank spaces, commas, and many other special characters(symbols).

INITIALIZATION:

Before the loop we will have a single node with data 0 in it and left and right of the node will be NULL. If there are no elements in frequency array F = [], the function will perform no operation and will return.

MAINTENANCE:

Suppose we have elements in an array F = [1…n]. now, in his case, the array is not empty and loop will execute. Initially, there is single node present in the tree with data zero. After entering

the loop, a node will be created with all null attributes. Tptr(pointer to node) will be pointing to the newly created node. The data from the frequency and character array will be stored in node's attributes respectively. The frequency from array F[i] will also be added in the pervious sum.

At line 13, it will check if the node's (to which s is pointing) left or right is null. If the condition is true , it will further check if the right node is null or left and will connect the new node to the NULL side.

If the condition is not true, it will go to the else on line 20 and create new node. The sum of current frequency F[i] and previous sum will be stored in this new node. then, it will again check for smaller data in node. if the node with pointer tptr is smaller than node pointed by s, it will place at the left side of node pointed by s, else right. The other null side of node will be connected by the previous sum node pointed by pptr.

TERMINATION:

After, the termination of the loop we will have a tree with characters at its leaf nodes. The character with the most frequency will be at the root node and will be assigned the smallest code. The character with least frequency will be at the lowest level of the tree, so it will be assigned the longest code.

Huffman Code(Tree)

```
1     i = 0, array[], node* tptr, head
2    if tree = NULL
         return
3     tptr= head = head node
4     if tptr.left!=Null
5         tptr = tptr.left
6          array[i] = 0
7      while tptr.left!=NULL and tptr.right!=NULL
8           if tptr.left != NULL
9                  i++
10               tptr= tptr.left
11                array[i] = 0
```

```
12          else
13              i++
14              tptr = tptr.right
15              array[i] = 1
**write character freq and  code to a file**
16     i = 0, tptr = head
17     if tptr.right !=NULL
18     tptr= tptr.right
19    array[i] = 1
 20  while tptr.left != NULL and tptr.right != NULL
21        if tptr.left.left == NULL and tptr.left.right == NULL
22                i++
23              array[i] = 0
                ** copy character, freq and array[] code in file**
24             tptr = tptr.right
25        else if tptr.right.left == NULL and tptr.right.right == NULL
26            i++
27            array[i] = 0
              **copy character, freq and array[] code in file**
28          tptr = tptr.left

29   if tptr.left != NULL
30            tptr=tptr.right
31            i++
32            array[i] = 1
33   else
34                tptr=tptr.left
```

35 i++

36 array[i] = 0

INITIALIZATION:

Before the starting of the loop, if the tree is null, no operation will be performed and the function will return. If a tree exists, then the loop of the function will execute. There is an initially empty array in function. The code digits will be stored in this array as the loop terminates.

MAINTENANCE:

Before, the while condition in line 7, it will check for the left side of the node. If the left node of the head exist. If there is a left node, the node pointer tptr will point to the left node of the head and store 0 in tha first index of array. It will again check for for left and right of tptr node. if left exist it will again store 0 in next index of array, otherwise , it will go to right and store 1 in next index of array.

After traversing the left side of the tree, the code stored in array index will be copied into the file, as the tptr reaches the leaf nodes.

Now the index i of array will be initialized with 0 and tptr will be again pointing to the head of the tree in line 16. Next, it will check for the right node of head. If it si not null it will move the tptr to right, 1 will be stored in the array first index and will enter in while loop. The loop at line 17 will execute until the node's left and right will be null. If the tptr's left node exist and if it is a leaf node, 0 will be stored in next index of array and the code will be copied into the file and the pointer tptr will be moves to its right node. when the tptr will move to its right, the code 1 will overwrite the current index.

the while loop continues until the tptr reaches the lowest position of the tree where its left and right node will be null.

After while loop, the if condition will check if there exist left aur right node and will moeve the tptr to that position. The code will be written to array.

TERMINATION:

In the end of an array, the code for the last leaf node character will be stored in array indexes. It will write the code into the file. The function will be completely execute and terminate.

Decoder(tree, code_char)

```
1  tptr = root_node
2  if tree = Null
       return
3  while(!eof())                    //for file1
4      if code_char = '0'
5          tptr= tptr.left
6      else if code_char = '1'
7                  tptr = tptr. rightS
8      if tptr. left = NULL and tptr. right = NULL
                   write to new file      //file2
9      else if tptr. left != NULL and tptr. right != NULL
                   read new char from file      //file 1
```

INITIALIZATION:

The tree and a single character from file will be passed to the function. Initially, if the tree is null, the function will not execute.

MAINTENANCE:

While condition in line 3 will check for the end of the compressed file. when the while loop starts, it compares the code bit from file to 0 and 1. If the condition is 0 tptr will be moved to left else right. If after moving, tptr reaches the the leaf node, it will write the character of the node to new file. else, it will stop pointer at current position and read next code bit from file. the loop continues until the while condition is true and the function terminates.

TERMINATION:

At the termination of the loop, the coded bits will be replaced by character in new file.

BONUS TASK:

Merge Files (string)

```
1  count1 = 0, count0 = 0
2  if string = “”
```

```
3        return
4     n = string. Length
5   for i = 0 to n
6         if string[i] = “1” and string[i+1] != “0”
7                  increment count1
8         if string[i+1] = “0”
9                  increment count1
10                 if count1=>1
                           write to file count1 and “1”
11                         count1 = 0
12       else if string[i] = 0 and string[i+1] != “1”
13                 increment count0
14        if string[i+1] = “1”
15                 increment count0
16                 if count0 => 1
                       write to file count0 and “0”
17                      count0 = 0
         write to file string[i]
```

also copy coded scheme file data to this file after above code completion

INITIALIZATION:

Initially, a string variable containing the data of compressed file is passed into the function. There are two variables count1 and count0 are used in the function and are initially 0. If the string is empty, the function will return.

MAINTENANCE:

The for loop will continue until the string ends. If the string at index i is 1, it will increment the count1. If there is 0 in the next index of string, it will write the character as it is in new file. else, if the next character is not 0, it will again increment the count1. Count1 will be incremented until the next character of string is 0. Then the count1 value concatenated with

character ‘1’ will be written to the file. Similarly, it will check the condition for 0 and follow the same steps and increment count0 variable by one.

TERMINATION:

After the loop ends, the file will be more compressed and the data from coded scheme file will also be copied into the same file.

CONCLUSION:

The correctness for all the functions are proved. Hence, our pseudocode will work correct.