

**Instituto**

**Politécnico**

**Nacional**

**Escuela Superior de Cómputo**

**REPORTE DE LA SESIÓN 7: Algoritmos Ávidos**

**Materia:**

Análisis de algoritmos

**Grupo:**

3CM3

**Integrantes:**

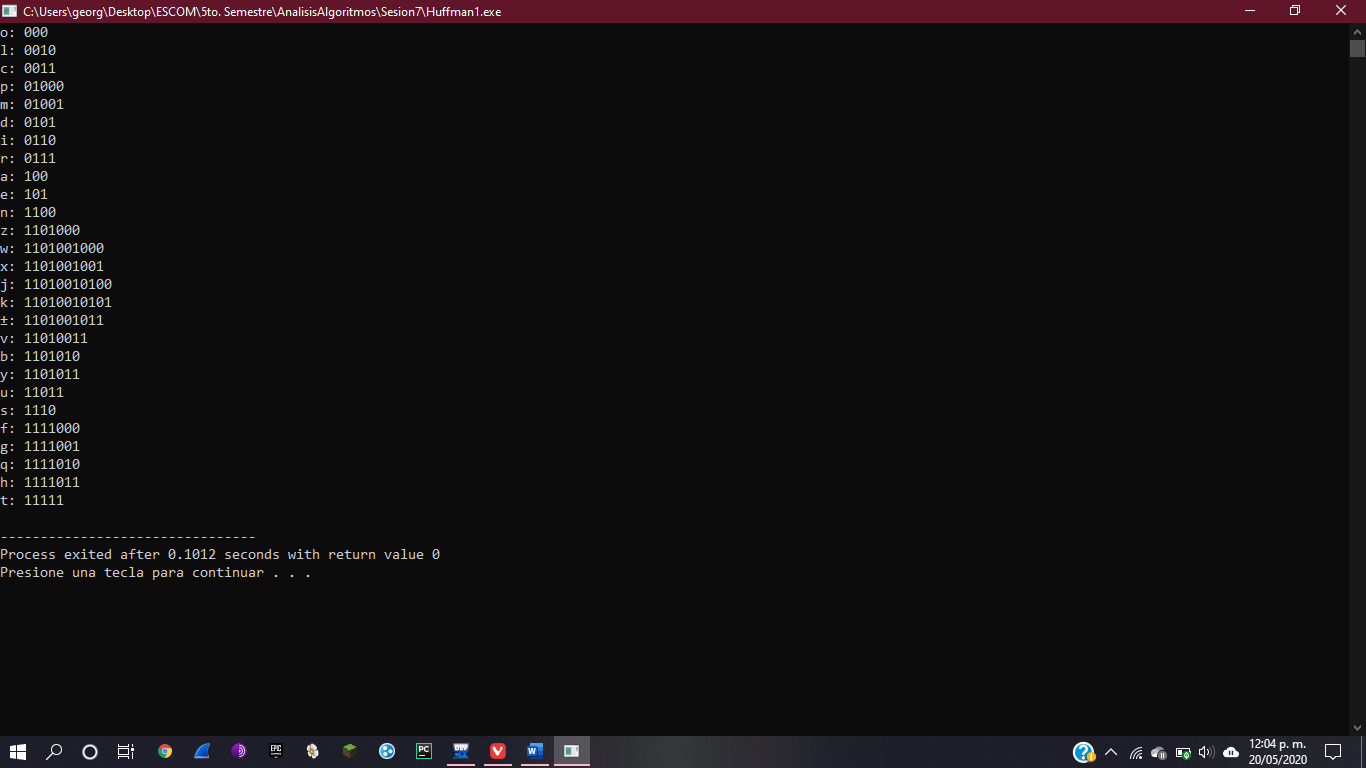
Castro Cruces Jorge Eduardo

**Fecha:**

Jueves, mayo 21, 2020**Ejercicio 2**

1. // C program for Huffman Coding
2. #include <stdio.h>
3. #include <stdlib.h>
5. // This constant can be avoided by explicitly calculating height of Huffman Tree
6. #define MAX\_TREE\_HT 100
8. // A Huffman tree node
9. **struct** MinHeapNode {
10. **char** data; // One of the input characters
11. unsigned freq; // Frequency of the character
12. **struct** MinHeapNode \*left, \*right; // Left and right child of this node
13. };
15. // A Min Heap:  Collection of min-heap (or Huffman tree) nodes
16. **struct** MinHeap {
17. unsigned size; // Current size of min heap
18. unsigned capacity; // capacity of min heap
19. **struct** MinHeapNode\*\* array; // Array of minheap node pointers
20. };
22. // A utility function allocate a new min heap node with given character and frequency of the character
23. **struct** MinHeapNode\* newNode(**char** data, unsigned freq) {
24. **struct** MinHeapNode\* temp = (**struct** MinHeapNode\*)malloc(**sizeof**(**struct** MinHeapNode));
25. temp->left = temp->right = NULL;
26. temp->data = data;
27. temp->freq = freq;
28. **return** temp;
29. }
31. // A utility function to create a min heap of given capacity
32. **struct** MinHeap\* createMinHeap(unsigned capacity) {
33. **struct** MinHeap\* minHeap = (**struct** MinHeap\*)malloc(**sizeof**(**struct** MinHeap));
34. minHeap->size = 0; // current size is 0
35. minHeap->capacity = capacity;
36. minHeap->array = (**struct** MinHeapNode\*\*)malloc(minHeap->capacity \* **sizeof**(**struct** MinHeapNode\*));
37. **return** minHeap;
38. }
40. // A utility function to swap two min heap nodes
41. **void** swapMinHeapNode(**struct** MinHeapNode\*\* a, **struct** MinHeapNode\*\* b) {
42. **struct** MinHeapNode\* t = \*a;
43. \*a = \*b;
44. \*b = t;
45. }
47. // The standard minHeapify function.
48. **void** minHeapify(**struct** MinHeap\* minHeap, **int** idx) {
49. **int** smallest = idx;
50. **int** left = 2 \* idx + 1;
51. **int** right = 2 \* idx + 2;
52. **if** (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq) smallest = left;
53. **if** (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq) smallest = right;
54. **if** (smallest != idx) {
55. swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);
56. minHeapify(minHeap, smallest);
57. }
58. }
60. // A utility function to check if size of heap is 1 or not
61. **int** isSizeOne(**struct** MinHeap\* minHeap) {
62. **return** (minHeap->size == 1);
63. }
65. // A standard function to extract minimum value node from heap
66. **struct** MinHeapNode\* extractMin(**struct** MinHeap\* minHeap) {
67. **struct** MinHeapNode\* temp = minHeap->array[0];
68. minHeap->array[0] = minHeap->array[minHeap->size - 1];
69. --minHeap->size;
70. minHeapify(minHeap, 0);
71. **return** temp;
72. }
74. // A utility function to insert a new node to Min Heap
75. **void** insertMinHeap(**struct** MinHeap\* minHeap, **struct** MinHeapNode\* minHeapNode) {
76. ++minHeap->size;
77. **int** i = minHeap->size - 1;
78. **while** (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
79. minHeap->array[i] = minHeap->array[(i - 1) / 2];
80. i = (i - 1) / 2;
81. }
82. minHeap->array[i] = minHeapNode;
83. }
85. // A standard function to build min heap
86. **void** buildMinHeap(**struct** MinHeap\* minHeap) {
87. **int** n = minHeap->size - 1;
88. **int** i;
89. **for** (i = (n - 1) / 2; i >= 0; --i) minHeapify(minHeap, i);
90. }
92. // A utility function to print an array of size n
93. **void** printArr(**int** arr[], **int** n) {
94. **int** i;
95. **for** (i = 0; i < n; ++i) printf("%d", arr[i]);
96. printf("\n");
97. }
99. // Utility function to check if this node is leaf
100. **int** isLeaf(**struct** MinHeapNode\* root) {
101. **return** !(root->left) && !(root->right);
102. }
104. // Creates a min heap of capacity equal to size and inserts all character of data[] in min heap.
105. // Initially size of min heap is equal to capacity
106. **struct** MinHeap\* createAndBuildMinHeap(**char** data[], **int** freq[], **int** size) {
107. **struct** MinHeap\* minHeap = createMinHeap(size);
108. **for** (**int** i = 0; i < size; ++i) minHeap->array[i] = newNode(data[i], freq[i]);
109. minHeap->size = size;
110. buildMinHeap(minHeap);
111. **return** minHeap;
112. }
114. // The main function that builds Huffman tree
115. **struct** MinHeapNode\* buildHuffmanTree(**char** data[], **int** freq[], **int** size) {
116. **struct** MinHeapNode \*left, \*right, \*top;
117. // Step 1: Create a min heap of capacity equal to size.  Initially, there are modes equal to size.
118. **struct** MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);
119. // Iterate while size of heap doesn't become 1
120. **while** (!isSizeOne(minHeap)) {
121. // Step 2: Extract the two minimum freq items from min heap
122. left = extractMin(minHeap);
123. right = extractMin(minHeap);
124. // Step 3:  Create a new internal node with frequency equal to the sum of the two nodes frequencies.
125. // Make the two extracted node as left and right children of this new node.
126. // Add this node to the min heap .
127. // '$' is a special value for internal nodes, not used
128. top = newNode('$', left->freq + right->freq);
129. top->left = left;
130. top->right = right;
131. insertMinHeap(minHeap, top);
132. }
133. // Step 4: The remaining node is the root node and the tree is complete.
134. **return** extractMin(minHeap);
135. }
137. // Prints huffman codes from the root of Huffman Tree.
138. // It uses arr[] to store codes
139. **void** printCodes(**struct** MinHeapNode\* root, **int** arr[], **int** top) {
140. // Assign 0 to left edge and recur
141. **if** (root->left) {
142. arr[top] = 0;
143. printCodes(root->left, arr, top + 1);
144. }
145. // Assign 1 to right edge and recur
146. **if** (root->right) {
147. arr[top] = 1;
148. printCodes(root->right, arr, top + 1);
149. }
150. // If this is a leaf node, then it contains one of the input characters, print the character and its code from arr[]
151. **if** (isLeaf(root)) {
152. printf("%c: ", root->data);
153. printArr(arr, top);
154. }
155. }
157. // The main function that builds a Huffman Tree and print codes by traversing the built Huffman Tree
158. **void** HuffmanCodes(**char** data[], **int** freq[], **int** size) {
159. // Construct Huffman Tree
160. **struct** MinHeapNode\* root = buildHuffmanTree(data, freq, size);
161. // Print Huffman codes using the Huffman tree built above
162. **int** arr[MAX\_TREE\_HT], top = 0;
163. printCodes(root, arr, top);
164. }
166. // Driver program to test above functions
167. **int** main() {
168. **char** arr[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'ñ', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z' };
169. **int** freq[] = { 13, 1, 5, 6, 14, 1, 1, 1, 6, 0, 0, 5, 3, 7, 0, 9, 3, 1, 7, 8, 5, 4, 1, 0, 0, 1, 1 };
170. **int** size = **sizeof**(arr) / **sizeof**(arr[0]);
171. HuffmanCodes(arr, freq, size);
172. **return** 0;
173. }

**Capturas de pantalla**



**Complejidad**

O (nlogn) donde n es el número de caracteres únicos. Si hay n nodos, extractMin () se llama 2 \* (n - 1) veces. extractMin () toma el tiempo O (logn) ya que se llama minHeapify (). Entonces, la complejidad general es O (nlogn). Si la matriz de entrada está ordenada, existe un algoritmo de tiempo lineal. Pronto discutiremos en nuestro próximo post.