Huffman Algorithm

Generated by Doxygen 1.9.1

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 bits Struct Reference

Almacena el byte correspondiente, su tamaño y sus bits de la codificacion de Huffman.

`#include <HuffmanStructures.h>`

Data Fields

- unsigned char byte
- int sizeBits
- int bits

3.1.1 Detailed Description

Almacena el byte correspondiente, su tamaño y sus bits de la codificacion de Huffman.

3.1.2 Field Documentation

3.1.2.1 bits

`bits::bits`

'bits' contiene el valor en entero de los bits correspondientes a la codifiacion

Referenced by getBits(), and writeBinaryCode().

3.1.2.2 byte

`bits::byte`

'byte' contiene el valor decimal del byte leido

Referenced by getBits().

3.1.2.3 sizeBits

`bits::sizeBits`

'sizeBits' contiene el tamaño de los bits correspondientes a la codifcacion

Referenced by getBits(), and writeBinaryCode().

The documentation for this struct was generated from the following file:

- HuffmanStructures.h

3.2 data Struct Reference

Almacena un byte y las veces que aparecio en el archivo.

`#include <HuffmanStructures.h>`

Data Fields

- int **frequency**
- int byte

3.2.1 Detailed Description

Almacena un byte y las veces que aparecio en el archivo.

3.2.2 Field Documentation

3.2.2.1 byte

`data::byte`

'byte' contiene el valor decimal del byte leido

Referenced by getBits(), getCharacters(), mergeNodes(), pushTree(), readFile(), and readFrecuencyTable().

The documentation for this struct was generated from the following file:

- HuffmanStructures.h

3.3 Heap Struct Reference

Almacena un arreglo de nodos.

`#include <HuffmanStructures.h>`

Data Fields

- struct node ** **arrayOfNodes**
- int count
- int capacity

3.3.1 Detailed Description

Almacena un arreglo de nodos.

3.3.2 Field Documentation

3.3.2.1 capacity

`Heap::capacity`

'capacity' contiene la capacidad maxima del arreglo

Referenced by CreateHeap(), and insert().

3.3.2.2 count

`Heap::count`

'count' contiene los nodos existentes en el arreglo

Referenced by CreateHeap(), heapify_top_bottom(), insert(), mergeTrees(), and PopMin().

The documentation for this struct was generated from the following file:

- HuffmanStructures.h

3.4 node Struct Reference

Estructura del arbol de Huffman.

`#include <HuffmanStructures.h>`

Data Fields

- struct node * left
- struct node * right
- struct data data

3.4.1 Detailed Description

Estructura del arbol de Huffman.

3.4.2 Field Documentation

3.4.2.1 data

`node::data`

'data' contiene el byte correspondiente y su frecuencia

Referenced by getBits(), getCharacters(), heapify_bottom_top(), heapify_top_bottom(), mergeNodes(), and pushTree().

3.4.2.2 left

`node::left`

'left' contiene el nodo hijo izquierdo

Referenced by getBits(), heapify_top_bottom(), isLeaf(), mergeNodes(), and pushTree().

3.4.2.3 right

`node::right`

'right' contiene el nodo hijo derecho

Referenced by getBits(), heapify_top_bottom(), isLeaf(), mergeNodes(), and pushTree().

The documentation for this struct was generated from the following file:

- HuffmanStructures.h

Chapter 4

File Documentation

4.1 CompressHuffman.h File Reference

Este programa comprime un archivo usando el algoritmo de Huffman.

Functions

- unsigned char * readFile (const char *fileToOpen, struct data bytesFrecuency[ ], int *fileSize)
- void writeFrecuenyTable (struct data bytesFrecuency[ ], int fileSize)
- void writeBinaryCode (unsigned char *bytesRead, struct node *mainTree, struct bits *bytesCode, int fileSize, int *compressedFileSize)

4.1.1 Detailed Description

Este programa comprime un archivo usando el algoritmo de Huffman.

Author

Victor Torres
Leilani Sotelo
Guillermo Sanchez

Version

1.0

4.1.2 Function Documentation

4.1.2.1 readFile()

```
unsigned char* readFile (
            const char * fileToOpen,
            struct data bytesFrecuency[],
            int * fileSize )
```

Lee los bytes del archivo de entrada y lo guarda en una cadena. Esta funcion lee un archivo de entrada, calcula el tamaño del archivo, lee los bytes del archivo y los guarda en una cadena ademas registra la frecuencia de aparicion de cada byte aplicando el concepto de Hash, donde la llave es el valor en decimal del byte que leimos

Parameters

fileToOpen	nombre del archivo a abrir
bytesFrecuency	arreglo donde esta la frecuencia de repeticion de bits
fileSize	direccion de memoria de la variable del tamaño del archivo

Returns

el apuntador de la cadena de los bytes leidos

```
9 {
10     int i;
11     FILE *file;
12     unsigned char c;
13 
14     //Abrimos y verificamos que si se abrio correctamente
15     file = fopen(fileToOpen, "rb");
16     if (file == NULL)
17     {
18         puts("The file could not be opened.\n");
19         exit(1);
20     }
21 
22     //Obtenemos el tamaño del archivo.
23     fseek(file, 0L, SEEK_END);
24     (*fileSize) = ftell(file);
25     rewind(file);
26 
27     //Reservamos memoria para la cadena de bytes leidos
28     unsigned char *bytesRead = malloc((*fileSize) * sizeof(unsigned char));
29 
30     //Guardamos los bytes leidos y su frecuencia con su valor en decimal
31     for (i = 0; i < (*fileSize); i++)
32     {
33         fread(&c, sizeof(unsigned char), 1, file);
34         bytesRead[i] = c;
35         bytesFrecuency[c].byte = c;    // Enmascaramiento
36         bytesFrecuency[c].frequency++; // frecuencia de apariciones
37     }
38     fclose(file);
39     return bytesRead;
40 }
```

References data::byte.

4.1.2.2 writeBinaryCode()

```
void writeBinaryCode (
             unsigned char * bytesRead,
             struct node * mainTree,
             struct bits * bytesCode,
             int fileSize,
             int * compressedFileSize )
```

Escribe la codificacion de Huffman de cada byte correspondiente. En un archivo llamado byteCode.dat, escribe el equivalente de la codificacion de Huffman de los bytes leidos del archivo original, esto lo hace mediante corrimientos de bits y con una bandera llamada sizeByteToWrite que nos indica si tenemos los 8 bits para escribir en el .dat, si no la guarda y mediante corrimeintos la suma con la codificacion del siguiente byte, en el ultimo byte del .dat, escribe cuantos bits extra escribimos que no corresponden a la codificacion de Huffman

Parameters

bytesRead	el apuntador al inicio de la cadena de los bytes leidos
mainTree	el arbol de Huffman donde unimos todos los bytes y su frecuencia
frecuencyTable	un arreglo donde temenos la codificacion de Huffman de los bytes
fileSize	el tamaño del archivo original

```
65 {
66     int i, j, bitsExtraWritten = 0;
67     FILE *binaryCode;
68 
69     //Abrimos y verificamos que si se abrio correctamente
70     binaryCode = fopen("byteCode.dat", "wb+");
71     if (binaryCode == NULL)
72     {
73         puts("The file could not be opened.\n");
74         exit(1);
75     }
76 
77     //Variables para poder escribir los bits
78     int bitsSize, sizeByteToWrite, tempSize = 0;
79     int bits, tempBits, tempBits2, tempBitsAux;
80     unsigned char byteToWrite;
81 
82     for (i = 0; i < fileSize; i++)
83     {
84         bitsSize = bytesCode[bytesRead[i]].sizeBits;
85         bits = bytesCode[bytesRead[i]].bits;
86 
87         //Checamos si hay bits que no escribimos
88         if (tempSize != 0)
89         {
90             bits = (tempBits « bitsSize) + bits;
91             bitsSize += tempSize;
92             tempBits = 0;
93             tempSize = 0;
94         }
95 
96         //Podemos escribir o no el byte?
97         if (bitsSize >= 8)
98         {
99             if (bitsSize > 8)
100             {
101                 //Tenemos mas bits de los que podemos escribir, hay que reducir
102                 //Guardamos lo que no podemos escribir en una variable temporal
103                 tempSize = bitsSize - 8;
104                 for (j = 0; j < tempSize; j++)
105                 {
106                     tempBits2 = (CONSULTARBIT(bits, j)) « j;
107                     tempBits = tempBits2 + tempBits;
108                 }
109                 //ajustamos los bits para poder escribir los primeros 8
110                 bits = bits » tempSize;
111             }
112             byteToWrite = bits;
113             sizeByteToWrite = 8; // Podemos escribir en el .dat
114         }
115         else
116         {
117             tempBits = bits;     // Guardamos en una variable temporal
118             tempSize = bitsSize; // lo que no pudimos escribir
119         }
120 
121         // Hay bytes por escribir?
122         if (sizeByteToWrite == 8)
123         {
124             sizeByteToWrite = 0; //Escribimos el byte
125             fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);
126         }
127 
128         //Checamos que no se desborde tempBits
129         while (tempSize >= 8)
130         {
131             //Ajustamos tempsize para poder escribir los primeros 8
132             tempSize = tempSize - 8;
133             tempBitsAux = tempBits;
134             bits = tempBits » tempSize;
135             tempBits = 0;
136 
137             //Guardamos el exceso en una variable temporal
138             for (j = 0; j < tempSize; j++)
139             {
140                 tempBits2 = ((CONSULTARBIT(tempBitsAux, j)) « j);
141                 tempBits = tempBits2 + tempBits;
142             }
143             byteToWrite = bits; // Escribimos el byte
144             fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);
145         }
146     }
147 
148     //Faltaron bits por escribir?
149     if (tempBits != 0)
150     {
151         tempBits = (tempBits « (8 - tempSize));
```

```
152         byteToWrite = tempBits;
153         fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);
154         bitsExtraWritten = 8 - tempSize; //Bits extra escritos
155     }
156 
157     //En el ultimo byte, indica cuantos bits extra escribimos
158     byteToWrite = bitsExtraWritten;
159     fwrite(&byteToWrite, sizeof(unsigned char), 1, binaryCode);
160 
161     //Obtenemos el tamaño del archivo comprimido
162     fseek(binaryCode, 0L, SEEK_END);
163     (*compressedFileSize) = ftell(binaryCode);
164     rewind(binaryCode);
165     //Cerrar el archivo
166     fclose(binaryCode);
167 }
```

References bits::bits, and bits::sizeBits.

4.1.2.3 writeFrecuenyTable()

```
void writeFrecuenyTable (
            struct data bytesFrecuency[],
            int fileSize )
```

Escribe la tabla de frecuencias y el tamaño del archivo leido. Escribe en un txt, el tamaño original del archivo ademas de el valor en decimal de cada bit que aparecio seguida de las veces que aparecio solo si la frecuencia de aparicion es mayor a 0, esto es, que el byte esta en el archivo original

Parameters

symbolFrecuency	arreglo donde esta la frecuencia de repeticion de bits
fileSize	direccion de memoria de la variable del tamaño del archivo

```
43 {
44     int i;
45     FILE *frecuencyTable;
46 
47     //Abrimos y verificamos que si se abrio correctamente
48     frecuencyTable = fopen("frecuencyTable.txt", "wt+");
49     if (frecuencyTable == NULL)
50     {
51         puts("The file could not be opened.\n");
52         exit(1);
53     }
54 
55     //Escribimos el tamaño original al inicio del txt
56     fprintf(frecuencyTable, "%d\n", fileSize);
57     for (i = 0; i < 256; i++)
58         if (bytesFrecuency[i].frequency > 0) //Si el simbolo aparecio en el archivo, escribelo
59             fprintf(frecuencyTable, "%d %d\n", bytesFrecuency[i].byte, bytesFrecuency[i].frequency);
60 
61     fclose(frecuencyTable);
62 }
```

4.2 DecompressHuffman.h File Reference

Este programa descomprime un archivo usando el algoritmo de Huffman.

Functions

- void readFrecuencyTable (struct data bytesFrecuency[ ], int *fileSize)

- unsigned char * readByteCode (int *fileSize, int *byteFileSize)
- void writeFile (unsigned char *bytesRead, struct node *mainTree, const char *file, int byteFileSize, int *file↩ Size)

4.2.1 Detailed Description

Este programa descomprime un archivo usando el algoritmo de Huffman.

Author

Victor Torres

Leilani Sotelo

Guillermo Sanchez

Version

1.0

4.2.2 Function Documentation

4.2.2.1 readByteCode()

```
unsigned char* readByteCode (
            int * fileSize,
            int * byteFileSize )
```

Lee el archivo comprimido. Obtenemos el tamaño del archivo comprimido y los bytes que lo componen, esto lo guardamos en una cadena.

Parameters

fileSize	direccion de memoria de la variable del tamaño del archivo
byteFileSize	direccion de memoria de la variable del tamaño del archivo comprimido

Returns

el arreglo donde estan los bytes leidos

4.2.2.2 readFrecuencyTable()

```
void readFrecuencyTable (
            struct data bytesFrecuency[],
            int * fileSize )
```

Lee y guarda la tabla de frecuencias. A partir de la tabla de frecuencia generada al comprimir el archivo, leemos las veces que se se repitio cada bit, ademas al principio del archivo leemos el tamaño original del archivo, las frecuencias las guardamos en un arreglo, con su respectivo byte aplicando hash, donde la llave es el byte.

Parameters

bytesFrecuency	arreglo donde esta la frecuencia de repeticion de bits
fileSize	direccion de memoria de la variable del tamaño del archivo

```
8                                                                   {
9       int byte, frecuency, i = 0;
10      FILE* frecuencyTable;
11
12      //Abrimos y verificamos que si se abrio correctamente
13      frecuencyTable = fopen("frecuencyTable.txt", "r");
14      if(frecuencyTable == NULL){
15          puts("The file could not be opened.\n");
16          exit(1);
17      }
18
19      //Leemos el tamaño de archivo y las veces que se repite cada byte
20      while(!feof(frecuencyTable)){
21      if(i == 0)
22          fscanf(frecuencyTable, "%d", fileSize);
23      fscanf(frecuencyTable, "%d", &byte);
24      fscanf(frecuencyTable, "%d", &frecuency);
25      bytesFrecuency[byte].byte =  byte;
26      bytesFrecuency[byte].frequency = frecuency;
27      i++;
28      }
29      fclose(frecuencyTable);
30  }
```

References data::byte.

4.2.2.3 writeFile()

```
void writeFile (
              unsigned char * bytesRead,
              struct node * mainTree,
              const char * file,
              int byteFileSize,
              int * fileSize )
```

Escribimos el archivo original a partir de la decodificacion de los bytes. Vamos recorriendo bit a bit de los bytes leidos en el archivo comprimido, y recorremos el arbol de Huffman, en base a esto nos devuelve tanto la la posicion en bits de 0 a 7 en la que se quedo, como en la variable byteToWrite, el byte correspondiente a cada cadena de bits, una ves que tenemos esto, escribimos en el archivo el byte correspondiente.

Parameters

bytesRead	el apuntador al inicio de la cadena de los bytes leidos
mainTree	el arbol de Huffman donde unimos todos los bytes y su frecuencia
file	el nombre del archivo en el que escribiremos
byteFileSize	el tamaño del archivo comprimido
fileSize	apuuntador de la variable del tamaño del archivo que escribiremos

```
61
              {
62      FILE* finalFile;
63
64      //Abrimos y verificamos que si se abrio correctamente
```

```
65      finalFile = fopen(file, "wb+");
66      if(finalFile == NULL){
67          puts("The file could not be opened.\n");
68          exit(1);
69      }
70
71      //Variables para escribir en el archivo
72      int i, posInBits = 7, bytesWritten = 0;
73      unsigned char byteToWrite;
74
75      //Leemos los bits extra que escribimos
76      int bitsExtraWritten = bytesRead[byteFileSize-1];
77
78      for(i = 0; i < byteFileSize - 1 && bytesWritten < (*fileSize);){
79      posInBits = getCharacters(mainTree, bytesRead , &i, posInBits, &byteToWrite);
80      //escribimos el byte correspondiente a los bits
81      fwrite(&byteToWrite, sizeof(unsigned char), 1, finalFile);
82      bytesWritten++;//conteo para no escribir de mas
83
84      //Cuidamos no escribir los bits extra que escribimos al comprimir
85      if(i == (*fileSize) - 2 && 7 - posInBits == bitsExtraWritten)
86          bytesWritten++;
87
88      //Checamos que no nos desbordamos de posicion de bits
89      if(posInBits < 0){
90          posInBits = 7;
91          i++;
92      }
93      }
94      (*fileSize) = bytesWritten;
95      fclose(finalFile);
96  }
```

References getCharacters().

4.3 HuffmanStructures.h File Reference

Las esctructuras y funciones usadas en el Algoritmo de Huffman.

Data Structures

- struct bits

  Almacena el byte correspondiente, su tamaño y sus bits de la codificacion de Huffman.
- struct data

  Almacena un byte y las veces que aparecio en el archivo.
- struct node

  Estructura del arbol de Huffman.
- struct Heap

  Almacena un arreglo de nodos.

Macros

- #define **PESOBIT**(bpos) 1 << bpos
- #define **CONSULTARBIT**(var, bpos) (*(unsigned *)&var & PESOBIT(bpos)) ? 1 : 0

Typedefs

- typedef struct Heap **Heap**

Functions

- int isEmpty (struct node *root)
- int isLeaf (struct node *root)
- void pushTree (struct node *root, unsigned char byte, int frecuency)
- struct node * mergeNodes (struct node *node1, struct node *node2)
- void getBits (struct node *HuffmanTree, struct bits bytesCode[ ], int bits, int bitsSize)
- int getCharacters (struct node *HuffmanTree, unsigned char *cadena, int *posInString, int posInBits, unsigned char *byteToWrite)
- Heap * CreateHeap (int capacity)
- void insert (Heap *heap, struct node *node)
- void heapify_bottom_top (Heap *heap, int index)
- void heapify_top_bottom (Heap *heap, int parent_node)
- struct node * PopMin (Heap *heap)
- void insertTree (struct data bytesFrecuency[ ], struct node roots[ ], Heap *heap)
- struct node * mergeTrees (Heap *heap)

4.3.1 Detailed Description

Las esctructuras y funciones usadas en el Algoritmo de Huffman.

Author

Victor Torres

Leilani Sotelo

Guillermo Sanchez

Version

1.0

4.3.2 Function Documentation

4.3.2.1 CreateHeap()

```
Heap* CreateHeap (
            int capacity )
```

Crea una estructura Heap en memoria dinamica dada su capacidad, la cual es un arreglo de nodos.

Parameters

capacity	es la capacidad que queremos que tenga heap

Returns

el apuntador a la estructura heap creada

```
73 {
74     Heap *heap = (Heap *)malloc(sizeof(Heap));
75     heap->count = 0;
76     heap->capacity = capacity;
77     heap->arrayOfNodes = (struct node **)malloc(capacity * sizeof(struct node));
78     return heap;
79 }
```

References Heap::capacity, and Heap::count.

4.3.2.2 getBits()

```
void getBits (
              struct node * HuffmanTree,
              struct bits bytesCode[],
              int bits,
              int bitsSize )
```

Obtiene la codificaciond de Huffman correspondiente a los bytes. La guarda en un arreglo de struct junto con el tamaño de bits correspondiente. La codificacion se obtiene recorriendo el arbol inorder y cada vez registramos un 0 o un 1 si nos vamos a la izquierda o derecha, al final si es nodo hoja, guardamos lo que llevamos

Parameters

HuffmanTree	el arbol de Huffman
bytesCode	el arreglo donde se guarda el byte, su codificacion y tamaño
bits	los bits con los que inicia el nodo que por defecto es 0
bitsSize	el tamaño de los bits que por defecto es 0

```
33 {
34     if (!isEmpty(HuffmanTree))
35     {                                                                    //if notempty
36         getBits(HuffmanTree->left, bytesCode, (bits « 1), sizeBits + 1); // irte al nodo izq, sumandole
       un 0
37         if (isLeaf(HuffmanTree))
38         {
39             int hashKey = HuffmanTree->data.byte;
40             bytesCode[hashKey].bits = bits;
41             bytesCode[hashKey].byte = HuffmanTree->data.byte;
42             bytesCode[hashKey].sizeBits = sizeBits;
43         }
44         getBits(HuffmanTree->right, bytesCode, (bits « 1) + 1, sizeBits + 1); //irte al nodo der
       sumandole un 1
45     }
46 }
```

References bits::bits, bits::byte, data::byte, node::data, isEmpty(), isLeaf(), node::left, node::right, and bits::sizeBits.

4.3.2.3 getCharacters()

```
int getCharacters (
              struct node * HuffmanTree,
              unsigned char * cadena,
              int * posInString,
```

```
            int posInBits,
            unsigned char * byteToWrite )
```

Obtiene el byte correspondiente a la codificacion de Huffman. Recorre el arbol analizando si es 0 o 1 el bit en el que esta para irse a la izquierda o a la derecha, cuando detecta que ya ha llegado a un nodo hoja, regresa en byteToWrite el valor del byte dado la cadena de bits

Parameters

HuffmanTree	el arbol de Huffman
cadena	la cadena de bytes leidos del archivo comprimido
posInString	la posicion del arreglo de la cadena
posInBits	la posicion de bits en la que vamos en relacion a un byte
byteToWrite	la variable donde guardaremos el byte correspondiente

Returns

la posicion en bits en donde nos quedamos en relacion a un byte

```
49 {
50     if (isLeaf(HuffmanTree))
51     {
52         *byteToWrite = HuffmanTree->data.byte;
53         return posInBits;
54     }
55     else
56     {
57         if (posInBits < 0)
58         {
59             (*posInString)++;
60             posInBits = 7;
61         }
62         if (((int)CONSULTARBIT(cadena[(*posInString)], (posInBits))) == 0)
63             return getCharacters(HuffmanTree->left, cadena, posInString, (posInBits)-1, byteToWrite);
64         else
65             return getCharacters(HuffmanTree->right, cadena, posInString, posInBits - 1, byteToWrite);
66     }
67 }
```

References data::byte, node::data, and isLeaf().

Referenced by writeFile().

4.3.2.4 heapify_bottom_top()

```
void heapify_bottom_top (
            Heap * heap,
            int index )
```

Ordena los elementos del arreglo de nodos, haciendo que el elemento mas pequeño sea la rama principal de nuestro arbol.

Parameters

Heap	es el monticulo al que vamos a ordenar su arreglo de nodos
index	es el indice en este caso el numero de elementos que hay que ordenar

```
92 {
93     struct node *temp;
```

```
94      int parent_node = (index - 1) / 2;
95      if (heap->arrayOfNodes[parent_node]->data.frequency > heap->arrayOfNodes[index]->data.frequency)
96      {
97          temp = heap->arrayOfNodes[parent_node];
98          heap->arrayOfNodes[parent_node] = heap->arrayOfNodes[index];
99          heap->arrayOfNodes[index] = temp;
100          heapify_bottom_top(heap, parent_node);
101      }
102 }
```

References node::data.

Referenced by insert().

4.3.2.5 heapify_top_bottom()

```
void heapify_top_bottom (
              Heap * heap,
              int parent_node )
```

Ordena los elementos del arreglo de nodos, haciendo que el elemento mas pequeño sea la rama principal de nuestro arbol,este lo usamos al momento de retirar el elemento principal de nuestro Heap, debido a que el ordenamiento lo realiza a partir de nuestra raiz hacia los elementos hijos del arbol.

Parameters

Heap	es el monticulo al que vamos a ordenar su arreglo de nodos
parent_node	es el indice del elemento padre de nuestro nodo para realizar

```
105 {
106      int left = parent_node * 2 + 1;
107      int right = parent_node * 2 + 2;
108      int min;
109      struct node *temp;
110
111      if (left >= heap->count || left < 0)
112          left = -1;
113      if (right >= heap->count || right < 0)
114          right = -1;
115
116      if (left != -1 && heap->arrayOfNodes[left]->data.frequency <
       heap->arrayOfNodes[parent_node]->data.frequency)
117          min = left;
118      else
119          min = parent_node;
120      if (right != -1 && heap->arrayOfNodes[right]->data.frequency <
       heap->arrayOfNodes[min]->data.frequency)
121          min = right;
122
123      if (min != parent_node)
124      {
125          temp = heap->arrayOfNodes[min];
126          heap->arrayOfNodes[min] = heap->arrayOfNodes[parent_node];
127          heap->arrayOfNodes[parent_node] = temp;
128
129          heapify_top_bottom(heap, min);
130      }
131 }
```

References Heap::count, node::data, node::left, and node::right.

Referenced by PopMin().

4.3.2.6 insert()

```
void insert (
              Heap * heap,
              struct node * node )
```

Inserta Nodos en la estructura de datos, indicando el Heap donde se desea hacer la incersion, seguido del nodo que se desea insertar, simempre y cuando la capacidad de Heap no haya sido exedida.

Parameters

heap	es el monticulo donde vamos a insertar el nodo
node	es el nodo a insertar dentro del monticulo

```
82 {
83     if (heap->count < heap->capacity)
84     {
85         heap->arrayOfNodes[heap->count] = node;
86         heapify_bottom_top(heap, heap->count);
87         heap->count++;
88     }
89 }
```

References Heap::capacity, Heap::count, and heapify_bottom_top().

Referenced by insertTree(), and mergeTrees().

4.3.2.7 insertTree()

```
void insertTree (
              struct data bytesFrecuency[],
              struct node roots[],
              Heap * heap )
```

Inserta en la cola de prioridad los nodos. EL numero de nodos que inserta son los bytes distintos que aparecieron en el archivo

Parameters

bytesFrecuency	el arreglo de struct donde viene el byte y su frecuencia
roots	el arreglo de nodos que trae todos los bytes y su frecuencia
heap	la cola de prioridad

```
152 {
153     int i;
154     for (i = 0; i < 256; i++)
155     {
156         if (bytesFrecuency[i].frequency > 0)
157         {
158             pushTree(&roots[i], bytesFrecuency[i].byte, bytesFrecuency[i].frequency);
159             insert(heap, &roots[i]);
160         }
161     }
162 }
```

References insert(), and pushTree().

4.3.2.8 isEmpty()

```
int isEmpty (
            struct node * root )
```

Nos dice si nodo en el arbol de Huffman es vacio o no.

Parameters

root	el nodo a comparar

Returns

1 si el arbol es vacio

0 si el arbol no es vacio

```
9 { return root == NULL; }
```

Referenced by getBits().

4.3.2.9 isLeaf()

```
int isLeaf (
            struct node * root )
```

Nos dice si nodo en el que estamos es un nodo hoja.

Parameters

root	el nodo a analizar

Returns

1 si el nodo es hoja

0 si el nodo no es hoja

```
11 { return root->left == NULL && root->right == NULL; }
```

References node::left, and node::right.

Referenced by getBits(), and getCharacters().

4.3.2.10 mergeNodes()

```
struct node* mergeNodes (
            struct node * node1,
            struct node * node2 )
```

Une dos nodos en un nodo ancestro comun

Parameters

node1	el primer nodo a unir que estara del lado izquierdo
node2	el segundo nodo a unir que estara del lado derecho

Returns

el nodo comun que tiene como hijos a ambos nodos

```
23 {
24     struct node *new_node = malloc(sizeof(struct node));
25     new_node->data.byte = 0;
26     new_node->left = node1;
27     new_node->right = node2;
28     new_node->data.frequency = node1->data.frequency + node2->data.frequency;
29     return new_node;
30 }
```

References data::byte, node::data, node::left, and node::right.

Referenced by mergeTrees().

4.3.2.11 mergeTrees()

```
struct node* mergeTrees (
             Heap * heap )
```

Une todos los nodos hoja en un solo, llamado Arbol de Huffman. Usando la cola de prioridad saca los dos nodos de menor frecuencia suma sus dos frecuencia y los une en un nodo padre, haciendo esto hasta que solo quede un nodo padre el cual contiene a todos los nodos de los bytes leidos

Parameters

heap	la cola de prioridad con los nodos de los bytes leidos

Returns

el Arbol de Huffman

```
165 {
166
167     while (heap->count > 1)
168     {
169         struct node *node1 = PopMin(heap);
170         struct node *node2 = PopMin(heap);
171         insert(heap, mergeNodes(node1, node2));
172     }
173     return PopMin(heap);
174 }
```

References Heap::count, insert(), mergeNodes(), and PopMin().

4.3.2.12 PopMin()

```
struct node* PopMin (
             Heap * heap )
```

Elmina el elemento mas pequeño de una cola de prioridad

Parameters

Heap	Es la estructura heap

Returns

nodo eliminado

```
134 {
135     struct node *pop;
136     if (heap->count == 0)
137     {
138         printf("Heap is Empty.\n");
139         return NULL;
140     }
141     pop = heap->arrayOfNodes[0];
142     heap->arrayOfNodes[0] = heap->arrayOfNodes[heap->count - 1];
143     heap->count--;
144     heapify_top_bottom(heap, 0);
145     return pop;
146 }
```

References Heap::count, and heapify_top_bottom().

Referenced by mergeTrees().

4.3.2.13 pushTree()

```
void pushTree (
              struct node * root,
              unsigned char byte,
              int frecuency )
```

Inserta dado un byte y una frecuencia, las inserta en el nodo del arbol

Parameters

root	el arbol de Huffman donde se insertara
byte	el byte correspondiente de su frecuencia
frecuency	la frecuencia correspondiente del byte

```
15 {
16     root->data.frequency = frecuency;
17     root->data.byte = byte;
18     root->left = NULL;
19     root->right = NULL;
20 }
```

References data::byte, node::data, node::left, and node::right.

Referenced by insertTree().

Index