Estructures de Dades

Dict_Encode

Estructura de dades on cada posició representarà una seqüencia de bytes b0,b1,...,bn-1,bn. Per entendre com es genera i s'obté aquesta seqüencia es recomanable mirar la descripció de la classe *Node* donat que és l'estructura de dades de cada posició del nostre Array.

Les propietats principals son les següents.

reset_dictionary()

S'encarrega de reinicialitzar el diccionari, el qual passarà a tenir 256 posicions (codi ASCII). Només el cridarem quan el diccionari sigui ple (de manera automàtica) o en la construcció del qual.

Ascii_value(byte c)

Retorna el valor númeric corresponent al byte c.

search_and_insert_BST(Integer i, byte c)

És la funció principal de la classe i s'encarrega de les següents funcions:

> En primer cas mira si cal reiniciar el diccionari, en aquest cas ho fa directament sense haver de cancel·lar tot el procés, etc.
>
> Cada cop que es crida es mira si ja tenim la cadena al diccionari indicada per (i,c).
> -En cas afirmatiu retornem la seva posició.
> -Si teniem el prefix però no la cadena, busquem exactament on s'insereix.
> -En cas contrari, l'afegim al diccionari.

Dict_Decode

Estructura de dades on cada posició indicarà una seqüencia de bytes b0,b1,...,bn-1,bn. El primer valor de l'Array indica la posició del següent byte de la subcadena b0,...,bn-1 mentre que el segon valor és bn.

reset_dictionary(Boolean b)

S'encarrega de reinicialitzar el diccionari, el qual passarà a tenir 256 posicions (codi ASCII) només en cas de que b sigui true. Només el cridarem quan el diccionari sigui ple (de manera automàtica) o en la construcció del qual.

getWord(Integer i)

S'encarrega de recorrer de manera iterativa l'Array de la classe fins que el valor de i sigui -1. Amb això el que aconseguim és retornar la paraula que correspon a la seqüència encadenada de bytes a partir de i.

Pair

És un template de la classe Pair de C++ el qual ha sigut creat per no haver d'utilitzar llibreries externes de Java.

Node

Representa un node emprat per la representació de cadenes de caràcters, utilitzat en la classe LZW.

Consta de 4 atributs de classe: - First -> Serà l'index al primer node que utilitzi la cadena de bytes que representi aquest. - c -> Valor del node - Left -> Index dels següents nodes(x) que utilitzin el mateix prefix que aquest i on x <= c - Right -> Index dels següents nodes(x) que utilitzin el mateix prefix que aquest i on x > c

ArrayCircular

Té la mateixa idea que una Array, però no hi ha cap remove. I, si s'excedeix el size del array, s'anira sobreescrivint els valors del inici i així seqüencialment amb un punter (anomenat start). És a dir, utilitza una política circular. Té diferents funcions:

setValue(byte value)

Que afegeix al array circular el valor value

getValue(int index)

Que retorna el valor que es troba a la posició index de l'array circular. En el cas que es trobi fora de rang, es crea una excepció.

getValueAmbDespl(int despl)

Utilitzat pel descompressor. El qual va despl vegades enrere des de la posició del punter, retornant el valor en aquella posició.

isIn(byte value)

Retorna un boolean indicant si value es troba o no a l'array circular. També hi ha operacions simples com retornar la posició del punter a la primera posició o el punter a la última posició, el nombre de valors afegits, que, com es obvi, té un màxim del valor que se li ha pasat per parametre al constructor, el qual serà la mida de la array.

IntorByte

En aquesta classe es tracta de construir un objecte per obtenir una unitat mínima per al descompressor. El constructor admetrà com a paràmetre un boolean, que indicarà si és un Byte (true) o la tupla despl-mida (false). Per tant, té les operacions òbvies de set i get de els atributs byte, despl i mida. Per últim, també té la consultora de si és un byte o dos Int's.

Trie

Aquesta classe és una implementació general de l'estructura de dades Trie. Cada node representa una seqüencia de Bytes. Cada connexió entre nodes (pare-fill), representa un caràcter. I un node representa la seqüència de caràcters des d'ell fins a l'arrel. A més, cada node té un enter que identifica la seva seqüència. (això serà útil per al compressor). Tot i que aquesta classe està implementada de forma general (), parlarem com si fos un Trie, ja que és com l'utilitzarem sempre en aquest projecte.

insert(ArrayList list, Integer índex)

L'algoritme lz78 només fa **insert**, si en el diccionari ja existeix la frase *list* - (últim Byte de *list*). O sigui que mai es donarà el cas que la funció inserti més d'un node al Trie. Dit això, aquesta funció afegeix un Node (que serà fill del node que representi *list* - (últim Byte de *list*)), i li assigna el *index*.

indexNode(ArrayList list)

Retorna l'índex del node que representa la frase *list*. Retorna -1 si no existeix.

TrieNode

Aquesta estructura de dades són els nodes del Trie. Cada node representa la seqüencia de bytes des d'ell fins a l'arrel. L'atribut *index* representa l'índex del node (que és l'identificador d'aquest). Després tenim *children*, que es un HashMap que cada entrada té com a key un byte i com a value un TrieNode. Després tenim dos consultores, una que retorna el HasMap que conté *children*. La segona constructora retorna l'índex d'aquest mateix TrieNode. Per últim, el constructor, que se li passa com a paràmetre l'índex.

BinTree

BinTree representa un arbre binari, on només les fulles poden tenir valor. Per tant, permet representar un arbre de Huffman.

El primer que se'm va acudir per representar un arbre binari és que contingués tres atributs: el valor de l'arrel, si en té; el subarbre fill esquerra, si en té; i el subarbre fill dret. Tanmateix em preocupava l'eficiència espacial de recórrer un arbre, ja que l'única manera que se m'acudia d'obtenir informació d'un node x és obtenir el subarbre del qual x és arrel i, per tant, haver de replicar tots els descendents de x.

Per tant vaig buscar una alternativa: assignaria un enter que identifiqués cada node i l'accés a un node amb al seu identificador seria directe: es podria obtenir en cost constant sabre si és fulla, saber el seu valor (en cas de ser fulla) i sabre l'identificador dels seus fills.

L'atribut principal de BinTree és arr, que funciona de la següent manera: - sigui x un node - si arr[2x] = -2, x és una fulla i arr[2x+1] és el valor de x - altrament x+arr[2x] i x+arr[2x+1] són els fills de x (-1 significa indefinit) - Un node no està inicialitzat <-> els dos fills son indefinits - El node arrel és 0

Per afegit un fill a un node identificat per x passem per paràmetre el subarbre que inclou el node fill a afegir amb tots els seus descendents. Com que la informació dels fills s'emmagatzema a l'atribut arr com a

distància relativa entre l'índex del pare i l'índex del fill, no cal modificar l'arr del subarbre i només cal afegir-lo al final de l'arr de l'arbre al qual afegim i assignar a arr[2x] (si és fill esquerre) o a arr[2x+1] (si és fill dret) la distància relativa entre pare i fill.

Algoritmes

LZW (Lempel–Ziv–Welch, 1984)

El mètode LZW crea sobre la marxa, de manera automàtica i en una única passada, un diccionari de cadenes que es trobin dins del text a comprimir mentre al mateix temps es procedeix a la seva codificació. Aquest diccionari no és transmès amb el text comprimit, ja que el descompressor pot reconstruir-lo usant la mateixa lògica amb què el fa el compressor i, si està codificat correctament, tindrà exactament les mateixes cadenes que el diccionari del compressor tenia.

Compressió

En la compressió amb LZW s'ha emprat l'estructura de dades*Dict_Encode* (la qual ha estat definida a l'apartat de **Estructures de Dades**). En un principi es va utilitzar un*Map\<Integer,String>* i això feia que la compressió fos massa lenta. En aquest moment i tenint en compte els dos factors següents, es va prendre la decisió d'eliminar els Strings i simplificar la representació de les cadenes: - No necessitem emmagatzemar la cadena sencera de Strings. - Qualsevol nou String que tinguem es pot representar com un append d'un byte al String anterior. Ara mateix ens quedariem doncs amb un *Map\<Integer,Byte>* on cada entrada representa l'index a la seqüència previa i el byte actual, inicialitzat amb (-1,*) on * són els primers 256 valors ASCII .

En aquest moment el compressor funcionava relativament bé però (inspirat en l'article de Juha Nieminen), es va decidir substituir el map per crear una estructura de dades propia que millores el funcionament, *Dict_Encode*. La definició de la qual, com bé s'ha dit al començament, ja està explicada més amunt, però la idea per voler utilitzar-la es que d'aquesta manera podiem buscar si una paraula ja estava al diccionari o en cas contrari afegir-la en una mateixa crida, sense haver de recorrer el contingut dos cops. A més, amb la representació d'arbre aconseguim agilitzar el procés donat que tindrem un byte ordenat amb tots els altres que accedeixin a un mateix prefixe.

D'aquesta manera el que farem es per cada byte que llegim del fitxer d'entrada, buscarem si la cadena que representa ja existeix, en aquest cas llegirem el següent amb la intenció de generar una cadena més gran, en cas contrari afegim aquesta cadena al diccionari i passem al byte següent.

Descompressió

En la descompressió amb LZW hem emprat l'estructura de dades*Dict_Decode* (la qual ha estat definida a l'apartat de **Estructures de Dades**). En un primer moment es va utilitzar un*ArrayList* però això no era eficient

per unes raons molt semblants a les de la compressió: - No necessitem emmagatzemar tota l'estona el text descomprimit. - Podem anar ajuntant els bytes que anem descomprimint amb els que ja estaven descomprimits.

Per això es van eliminar els Strings pel que es va quedar l'estructura de dades*ArrayList\< Pair\<Integer,Byte> >* amb la qual només necessitariem anar unint els bytes segons l'index al que apunta cada qual.

D'aquesta manera el que farem es anar generant de nou un diccionari a mesura que anem descomprimint cada byte i retornant la cadena que representa cada decodificació.

LZSS (Lempel–Ziv–Storer–Szymansk, 1982)

En aquesta classe es tracta la compressió mitjançant l'algorisme LZSS i la descompressió d'un input el qual ha estat comprimit amb aquest mateix algoritme. Per la part de la compressió es té una finestra corredissa de 2048 posicions (11 bits), ja que s'han fet proves estimant un equilibri eficiència/temps. Després la mida de 5 bits, per tant, es buscarà una compressió mínima de 3B. Per la part de la descompressió, s'agafarà la unitat mínima representada per IntorByte, on, com bé especifica aquella classe, seran dos Int's o un Byte.

Compressió

Es crean dos estructures de dades: ArrayCircular, per la finestra corredissa i ArrayList, per la subparaula que es voldrà trobar a la finestra.
L'ArrayList és de Bytes per la qüestió que s'ha de tractar utf-8, ja que si es posa de char's, java farà una interpretació ASCII.
En el cas de l'ArrayCircular s'ha utilitzat per reduir el cost a nivell de temps del programa, ja que, anteriorment, s'utilitzava un TreeMap\<Integer, Byte> el qual, quan arribava al size màxim de la finestra, s'anava fent remove de les primeres posicions afegides (molt costós). Per tant, amb un array amb política circular, no cal fer cap remove i reestructurar (molt menys costós).
A continuació, es fa un while, on, a cada iteració, s'agafa un byte del input. Amb aquest byte nou, s'afegeix primerament a l'estructura de dades ArrayList (subparaula). Aquesta, es buscarà si està o no a la finestra, mitjançant l'algorisme Knuth-Morris-Pratt, amb cost lineal. Per tant, es tenen dos casos:

> S'ha trobat a la finestra:
>
> > Llavors es continuarà la busqueda d'una subparaula més gran a la finestra afegint bytes a aquesta (amb un màxim de mida de 34, ja que, com s'utilitzen 5 bits de mida, 2^5-1 == 31 (rang), llavors, se sap que 0, 1 i 2 no s'utilitzarà, per una qüestió que no es comprimiran bytes de mida més petita a 3, es pot afegir +3 a la mida per comprimir).
>
> No s'ha trobat a la finestra:
>
> > Llavors, si la subparaula, es compleix que mida-1 >= 3 (-1 ja que al inici s'afegeix un byte a aquesta subparaula que no s'ha trobat), s'haurà d'afegir al Output el substring de la subparaula (de la posició 0 a size-2) i continuar buscant "match's" per l'últim byte afegit a la subparaula.
> > En el cas que no es compleixi la condició anterior, s'haurà d'afegir el byte sense comprimir al Output.

Descompressió

Es crea una estructura de dades ArrayCircular, per representar la finestra corredissa ja introduïda, la qual, s'anirà omplint a mesura que estem al while agafant byte's del input. També es crea un IntorByte, ja introduït a apartats anteriors. Dins del while, s'aniran agafant instàncies IntorByte, les quals, representaran, o bé un byte i per tant, s'afegeix a la finestra i al output, o bé dos Int's, els quals seran una tupla punter/tamany, que s'aniran afegint al Output i a la vegada a la finestra.

LZ78 (Lempel-Ziv, 1978)

LZ78 és un algoritme *lossless*, i que en temps lineal comprimeix un arxiu qualsevol. L'estructura de dades que utilitza és un diccionari, el qual és una col·lecció potencialment il·limitada de frases vistes anteriorment. Cada frase en el diccionari té associada un índex que codificarà aquesta frase.

Compressió

La compressió té el següent funcionament: A mesura que va llegint, va introduïnt frases a un diccionari, i després, quan es troba una repetició d'alguna frase que estigui al diccionari, *imprimeix* l'índex de l'entrada del diccionari en lloc de la frase. En la meva implementació de l'algoritme, vaig començar utilitzant un HashMap<String,Integer>;. La clau era la frase, i el valor era l'índex que representava aquella frase en concret. L'algoritme funcionava bé, a una bona velocitat (seguia sent lineal). Però utilitzava molt espai de memòria (RAM). Vaig decidir canviar el HashMap per un altre tipus de diccionari: El Trie. L'aventatge que té el Trie respecte al HashMap és que, tot i mantenir la funció *find* a cost temporal constant, utilitza molta menys memòria. Ja que cada frase emmagatzemada al Trie ocupa sempre 1 byte, mentre que el HashMap ocupava tants bytes com la mida de la frase. Això es deu a l'estructura que segueix el Trie, que cada node representa la seqüència de bytes que va des d'ells fins a l'arrel.

Descompressió

El text descomprimit, per la natura de la compressió, és sempre una seqüència de Pair<Integer,Byte>;. Tenint això en compte, la descompressió segueix aquest funcionament: Anem generant un nou diccionari a mesura que llegim l'entrada. A més, els integers que llegim són entrades al diccionari que existeixen (a no ser que sigui 0, que aleshores no està al diccionari i hem d'afegir-lo). Quan llegim un integer, *imprimim* la frase apuntada per aquest. Els bytes els *imprimim* tal com els llegim.

L'estructura de dades emperada a la descompressió ha estat*Dict_Decode* (la qual ha estat definida a l'apartat de **Estructures de Dades**).

D'aquesta manera el que farem es anar generant de nou un diccionari a mesura que anem descomprimint cada byte i retornant la cadena que representa cada decodificació.

JPEG (Joint Photographic Experts Group)

Explicaré simultàniament la compressió i la descompressió ja que els passos d’un son els inversos dels passos de l’altre en ordre invers.

Block splitting

El fet d'haver de treballar en blocs 8x8 fa que el compressor processa els píxels en el mateix ordre en què es llegeixen i el descompressor no processa els píxels en el mateix ordre en què s'escriuen. Això és així perquè en una imatge PPM entre una fila de píxels d'un bloc i la següent hi ha n files d'altres blocs, on n és l'enter superior a (amplada de la imatge)/8. Per això la unitat mínima de lectura i escriptura d'una imatge PPM per a l'algoritme JPEG és el conjunt dels n blocs que comparteixen files entre ells, concretament una matriu nx8x8x3, diguem-ne mat, on mat[b][i][j][k] és la quantitat de color #k (color #0 és red; color #1 és green; color #2 és blue) del píxel (i, j) del bloc b.

Si l'amplada o alçada son múltiples de 8 l'últim bloc de cada fila i l'última fila de blocs contindrà valors que no representen a cap píxel de la imatg però son necessaris per tal que l'algoritme JPEG pugui processar els valors d'aquests blos que sí que representen píxels de la imatge.

En el compressor, un cop llegida aquesta matriu, ja es pot processar cadascun dels n blocs per separat. En el descompressor un cop processats cadascun dels n blocs per separat ja es pot escriure aquesta matriu. Les classes Ctrl_Input_Img i Ctrl_Output_Img s'encarreguen de llegir i escriure, respectivament, en el format d'aquesta matriu.

Color space transformation

Sigui (r, g, b) la tripleta que representa un píxel en format RGB Sigui (y, cb, cr) la tripleta que representa un píxel en format YCbCr.

Per transformar de RGB a YCbCr en un rang de 0 a 255 he utilitzat les següents assignacions:

> y = 16 + (65.738r + 129.057g + 25.064b)/256;
>
> cb = 128 + (-37.945r - 74.494g + 112.439b)/256;
>
> cr = 128 + (112.439r - 94.154g - 18.285b)/256;

Per transformar de YCbCr a RGB en un rang de 0 a 255 he utilitzat les següents assignacions:

> r = 255/219*(y-16) + 255/224 1.402 (cr-128);
>
> g = 255/219*(y-16) + 255/224 (1.772 * 0.114/0.587 * (cb-128) - 1.402 * 0.299/0.587 (cr-128));
>
> b = 255/219*(y-16) + 255/224 * 1.772 (cb-128);

Vaig voler comprovar el bon funcionament de la transformació fent el següent: per diversos colors RGB, transformar-los a YCbCr, tornar-los a transformar a RGB i comparant els valors inicials de RGB amb els finals. Vaig veure que no eren exactament les mateixes però que les peites diferències eren imperceptibles a l'ull humà a l'hora de visualitzat el color. Tanmateix era preocupant que alguns valors sortissin del rang (0, 255) ja que això no permetria representar-los bé en el fitxer ppm. Per això en després d'aplicar la transformació de YCbCr si algun valor era negatiu forçava que fos 0 i si algun valor sobrepassava 255 forçava que fos 255.

Discrete cosine transformation

Per cada canal YCbCr vaig aplicar DCT en la compressió i DCT inversa en la descompressió.

Quantization

La matriu de quantització dependrà de l'atribut de classe anomenat quality, que és un enter entre 0 i 100.

Si quality == 0 cada valor de la matriu de quantització serà infinit, per tant, en dividir tots els valors resultants d'aquest procés seran 0, per tant, no s'estarà guardant cap informació més enllà de les dimensions de la imatge.

Si quality == 50 la matriu de quantització serà la següent:

> { {16, 11, 10, 16, 24, 40, 51, 61}, {12, 12, 14, 19, 26, 58, 60, 55}, {14, 13, 16, 24, 40, 57, 69, 56}, {14, 17, 22, 29, 51, 87, 80, 62}, {18, 22, 37, 56, 68, 109, 103, 77}, {24, 35, 55, 64, 81, 104, 113, 92}, {49, 64, 78, 87, 103, 121, 120, 101}, {72, 92, 95, 98, 112, 100, 103, 99} };

Aquesta és la que anomeno "mariu base". Totes les matrius de quantització es calculen a partir d'aquesta matriu i del paràmetre quality.

Si quality == 100 cada valor de la matriu de quantitació son 1, per tant no hi haurà cap pèrdua d'informació en el procés de quantització. Per tant la compressió serà gairebé loseless. Dic gairebé perquè altres processos de l'algorisme poden generar petites pèrdues d'informació que son quasi imperceptibles a l'ull humà.

El procés de calcular la matriu de quantització a partir d'un paràmetre de qualitat no és universal. Jo he utlitzat el que he trobat a la següent url: https://stackoverflow.com/questions/29215879/how-can-i-generalize-the-quantization-matrix-in-jpeg-compression

El valor de quality i la consegüent matriu de quantització es fixen quan es crida el mètode públic anomenat resetQuality(int q). Per tant la responsabilitat d'escollir aquest valor no és de l'algoritme JPEG.

- Si l'usuari decideix comprimir en mode automàtic quality serà 50.
- Si l'usuari decideix comprimir amb un factor de compressió baix quality serà 100
- Si l'usuari decideix comprimir amb un factor de compressió mitjà quality serà 70
- Si l'usuari decideix comprimir amb un factor de compressió alt quality serà 40

Aquests valors s'han decidit després de comprimir diverses imatges de diverses mides per diferens valors de qualitats i observant la qualitat visual dels descomprimits i el factor de compressió.

El factor de compressió depèn de cada imatge indivisual però oscil·la entre 20 (per una qualitat de 40%) i 2,3 (per una qualitat del 100%), aproximadament.

Entropy coding

La codificació de Huffman se n'encarrega una classe específica, que genera un codi de Huffman per cada possible valor que pugui prendre el que jo anomeno "símbol 1 de entropy coding", proporciona un codi de Huffman d'una mida determinada. També fa la transformació inversa: per cada codi de Huffman proporciona el un valor pel símbol 1 de entropy coding. El que jo anomeno "símbol 1 de entropy coding" és un byte que conté la runlength als 4 bits de més pes i la size (mida del símbol 2 de entropy coding) als 4 bits de menys pes, tal com explica https://en.wikipedia.org/wiki/JPEG#Entropy_coding .

Un atribut de la classe JPEG és una instància de la classe Huffman.

El símbol 2 de la entropy coding depèn del valor de la posició en qüestiò de la matriu que resulta de la quantització i té mida variable. He fet aquesta conversió de la manera que explica la següent web: https://www.impulseadventure.com/photo/jpeg-huffman-coding.html .

Per la descompressió, com que el símbol 2 té mida variable cal anar llegint del fitxer .jpeg bit a bit i anar preguntant a la instància de la classe Huffman si pels bits llegits fins al moment s'ha trobat un símbol pel codi format pels bits llegits fins al moment i seguir llegint bits fins a trobar-lo.

Entropy coding utilitza una taula de huffman predeterminada, és a dir, la instància huffman és amb mode automàtic. Per la segona entrega em vaig plantejar que la taula de huffman fos generada a partir de les freqüències de cada símbol. Això provocaria dos inconvenients:

- Com que cal guardar les freqüències dels símbols caldria recórrer tota la imatge guardar-se la informació de tots els blocs i després generar la taula de Huffman. Això implicaria que la compressió tindrà un cosst espacial lineal respecte la mida de la imatge comprimida. Si la imatge és gran l'increment de cost espacial podria fer incrementar el cost temporal.
- Caldria escriure la taula de Huffman a al imatge comprimida perquè el decompressor la pugués interpretarà bé. Aquest increment de la mida del comprimit no està clar que sigui compensat pel fer que la codificació de Huffman fos òptima.

Vaig implementar JPEG amb taules de Huffman generades a partir de les freqüències i els resultats no van ser satisfactoris per tant, vaig daixar-lo amb taules de Huffman automàtiques.

Huffman

Aquesta classe se'n carrega de de proporcionar la relació de codis de Huffman per una sèrie de símbols, que son enters.

Segons com s'inicialitzi la instància esterem en mode manual o automàtic: -**Mode automàtic**: Si la inicialitzem sense passar cap paràmetre a la constructora estarem en mode automàtic. Aquest mode utilitza una taula de Huffman predeterminada especialment pensada per l'algoritme JPEG. És la que vaig trobar en el següent link: https://www.ece.ucdavis.edu/cerl/reliablejpeg/coding/ . - **Mode manual**: Cal passar per paràmetre de la constuctora un map, on les claus son els símbols a codificar i els valors son les freqüències da cada símbol. Genera una codificació òptama mitjançant la tècnica que vaig trobar el el següent link: https://en.wikipedia.org/wiki/Huffman_coding#Basic_technique . Tot i que al final JPEG utilitza Huffman en mode automàtic m'ha semblat interessant deixar la implementació de Huffman pel mode manual.

La classe Huffman té un atribut anomenat auto_codes (long[]) i un altre anomentat man_codes (HashMap<Integer, Long>). Donat un símbol x: - En mode automàtic code(auto_codes[x]) és la codificació de x, que té una mida de size(auto_codes[x]). - En mode manual code(man_codes.get(x)) és la codificació de x, que té una mida de size(man_codes.get(x))

size(long) i code(long) són funcions estàtiques i privades de la classe.

Un altre atribut és tree, que és l'arbre de Huffman i és de tipus BinTree, una classe especialment creada per ser utilitzada per Huffman.

Per obtenir un codi de Huffman a partir d'un símbol, es fa amb les funcions getCode(int symbol) i getSize(int

symbol) i és tan fàcil com consultat la posició symbol del vector auto_codes.

Si tenim una tira se bits amb un codi de Huffman i volem obtenir el símbol que representen podem utilitzar getSymbol(int code) però com que possiblement no sabrem on acaba el codi i no volem llegir més bits dels necessaris podem seguir el següent procés:

> initSearchSymbol()
>
> mentre no haguem trobat el símbol i no tinguem la certesa que el codi no és vàlid:
>
> > llegim un bit
> >
> > executem searchSymbol(int bit), que ens indicarà si s'ha trobat el símbol, cal seguir buscant o el codi no és vàlid
>
> si s'ha trobat un símbol l'obtenim amb la funció getFoundSymbol()

Aquest procés funciona gràcies a un atribut privat de la classe que identifica un node de l'arbre de Huffman i fuciona com a punter.