Α.Τ.Ε.Ι ΚΑΛΑΜΑΤΑΣ ΠΑΡΑΡΤΗΜΑ ΣΠΑΡΤΗΣ
ΤΜΗΜΑ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

Αλγοριθμοι Συμπιεσης Δεδομενων

Πτυχιακη Εργασια

Κρικέλης Εμμανουήλ

Μάιος 2012

Επιβλέπων
Επ. Καθ. Καραγιώργος Γρηγόρης

Στην εργασία αυτή παρουσιάζεται για πρώτη φορά ο αλγόριθμος με την ονομασία ΔΚΜΜ ή VLCC coding. Ο αλγόριθμος αυτός είναι προϊόν μίας προσπάθειας που γίνεται εδώ και χρόνια. Δεν αποτελεί μέρος καμίας βιβλιογραφίας, δημοσίευσης, ή οτιδήποτε άλλο. Υπεύθυνος για την δημιουργία του αλγορίθμου είναι ο συγγραφέας αυτής της πτυχιακής εργασίας.

Ο συγγραφέας και δημιουργός,

Εμμανουήλ Π. Κρικέλης

Περιεχόμενα

Περίληψη

Στην παρούσα πτυχιακή θα μελετήσουμε αλγόριθμους για την συμπίεση δεδομένων και συγκεκριμένα ενός κειμένου. Θα δούμε τον αλγόριθμο που ανέπτυξε ο David A. Huffman, ο οποίος αποτελεί αν όχι τον καλύτερο, έναν από τους καλύτερους αλγορίθμους στον κόσμο, για την κωδικοποίηση σύμβολο προς σύμβολο. Στην συνέχεια θα δημιουργηθεί ένας αλγόριθμος με στόχο να ξεπεράσει, όπου αυτό είναι δυνατό, τον αντίστοιχο του Huffman.

Θα ξεκινήσουμε παρουσιάζοντας μερικά βασικά πράγματα που πρέπει να γνωρίζουμε για το πρόβλημα. Από εκεί και πέρα, θα επικεντρωθούμε καθαρά στον τρόπο λειτουργίας των αλγορίθμων. Οι αλγόριθμοι που χρησιμοποιούμε συνοδεύονται από ψευδοκώδικα και παραδείγματα για την καλύτερη κατανόηση του τρόπου λειτουργίας τους.

Στο τέλος θα υλοποιήσουμε τους αλγορίθμους στην γλώσσα C για να δούμε το πως αυτοί λειτουργούν στην πράξη.

Ευχαριστίες

Με την εργασία αυτή ολοκληρώνω τις σπουδές μου στο Α.Τ.Ε.Ι Καλαμάτας παράρτημα Σπάρτης, τμήμα τεχνολογίας πληροφορικής και τηλεπικοινωνιών. Θα ήθελα να ευχαριστήσω σχεδόν όλους τους καθηγητές που με στήριξαν χαρίζοντας μου πολύτιμη γνώση.

Θα ήθελα να ευχαριστήσω ιδιαίτερα τον επιβλέπων της πτυχιακής μου, Επ. Καθηγητή Γρηγόρη Καραγιώργο. Τον ευχαριστώ για την γνώση και στήριξη που μου πρόσφερε σε πολλά θέματα, όλα αυτά τα χρόνια που τον γνωρίζω. Τον ευχαριστώ που ήταν πάντα πρόθυμος να με βοηθήσει σε όλη την προσπάθεια που κατέβαλα για να δημιουργήσω τον αλγόριθμο που παρουσιάζω σε αυτή την πτυχιακή. Επίσης, τον ευχαριστώ για την τεράστια βοήθεια του, στο κομμάτι της υλοποίησης των αλγορίθμων στην γλώσσα C.

Θα ήθελα να ευχαριστήσω τους φίλους που έκανα κατά την διάρκεια των σπουδών μου, για όλα όσα μου πρόσφεραν.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου, που με στηρίζει σε οποιαδήποτε επιλογή μου.

1. Τι Πρέπει να Γνωρίζουμε

1.1. Το Πρόβλημα

Σε αυτή την πτυχιακή εργασία θα μελετήσουμε το πρόβλημα της συμπίεσης των δεδομένων. Με τον όρο συμπίεση δεδομένων αναφερόμαστε σε ένα μεγάλο και πολύ σημαντικό κεφάλαιο στην επιστήμη της πληροφορικής. Η συμπίεση είναι μία τεχνική που χρησιμοποιείται σήμερα σε όλους τους τύπους δεδομένων. Παρόλα αυτά, εμείς θα ασχοληθούμε με ένα πολύ μικρό κομμάτι, που αφορά την συμπίεση ενός κειμένου.

Ένα κείμενο αποτελείται από χαρακτήρες, αριθμούς και άλλα σύμβολα. Για την αναπαράσταση αυτών των συμβόλων σε ένα ηλεκτρονικό υπολογιστή χρησιμοποιούμε ένα δυαδικό κώδικα (π.χ. ASCII). Έτσι λοιπόν, αν είχαμε ένα κείμενο με τους χαρακτήρες a, b, c, d, e, μπορούμε να τους αναπαραστήσουμε σε κώδικα ASCII, κάτι που σημαίνει ότι θα χρησιμοποιούσαμε 8 δυφία για την αναπαράσταση του κάθε χαρακτήρα. Μπορούμε όμως να σκεφτούμε ακόμα πιο έξυπνα και να χρησιμοποιήσουμε ένα δικό μας κώδικα. Εφόσον έχουμε μόλις πέντε χαρακτήρες μπορούμε να χρησιμοποιήσουμε ένα κώδικα που να μπορεί να αναπαραστήσει τον κάθε χαρακτήρα με μόλις 3 δυφία.

Σε κάθε περίπτωση ο ελάχιστος αριθμός δυφίων x που χρειάζονται για να αναπαραστήσω ένα σύνολο από n χαρακτήρες πρέπει να ικανοποιούν την σχέση$2^x \geq n$. Έτσι, για $n = 5$ χρειάζομαι τουλάχιστον $2^3 \geq 5$,άρα $x = 3$ δυφία για να αναπαραστήσω τον κάθε χαρακτήρα.

000
001
010
011
100
101
110
111

Πίνακας 1.1.: Όλοι οι συνδυασμοί για 3 δυφία.

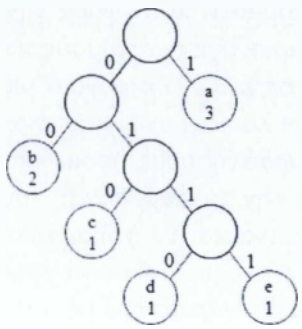
Σύμφωνα με τον Πίνακα 1.1, αν έχω τρία δυφία μπορώ να σχηματίσω οκτώ συμβολοσειρές. Στην συνέχεια μπορώ να επιλέξω πέντε από αυτές και να αναπαραστήσω τους χαρακτήρες μου.

Για την κωδικοποίηση του κάθε χαρακτήρα επιλέγω τις πρώτες πέντε συμβολοσειρές. Οπότε έχουμε, a:000, b:001, c:010, d:011, e:100. Αν για παράδειγμα είχα την λέξη aaab-

bcde τότε μπορώ να την κωδικοποιήσω ως 000000000001001010011100. Το σημαντικό εδώ είναι ότι μπορούμε να διαβάσουμε και να ξεχωρίσουμε τον κάθε χαρακτήρα.

Το επόμενο βήμα είναι να βρούμε ένα τρόπο, έτσι ώστε να χρειάζονται λιγότερα δυφία για την αναπαράσταση του κάθε χαρακτήρα. Αυτό σημαίνει ότι θα καταφέρναμε να συμπιέσουμε το κείμενο μας. Η ιδέα προέκυψε από μία δημοσίευση του C. E. Shannon με την ονομασία *A mathematical theory of communication*. Στην συνέχεια, ο καθηγητής του MIT R. M. Fano δημοσίευσε μια αναφορά *The transmission of information* κατά την όποια η κωδικοποίηση ενός χαρακτήρα θα ήταν ανάλογη με την συχνότητα της εμφάνισης του στο κείμενο.

Σύμφωνα με την ιδέα αυτή, θα χρησιμοποιούσαμε διαφορετικό αριθμό δυφίων για την αναπαράσταση του κάθε χαρακτήρα. Έτσι οι χαρακτήρες με την μεγαλύτερη συχνότητα εμφάνισης θα διαθέτουν λιγότερα δυφία από αυτούς με την μικρότερη συχνότητα. Η κωδικοποίηση του κάθε χαρακτήρα θα υπολογίζεται μέσω ενός δυαδικού δένδρου.



Σχήμα 1.1.: Δυαδικό δένδρο για τον υπολογισμό δυαδικού κώδικα μεταβλητού μήκους.

Στο Σχήμα 1.1, έχουμε ένα δυαδικό δένδρο. Επίσης, το παραπάνω δυαδικό δένδρο παράγει βέλτιστο δυαδικό κώδικα μεταβλητού μήκους, πράγμα που σημαίνει ότι κανένας μηχανισμός για την κωδικοποίηση σύμβολο προς σύμβολο δεν θα μπορέσει να κωδικοποιήσει το μήνυμα χρησιμοποιώντας μικρότερο αριθμό δυφύων. Οι αλγόριθμοι που θα μελετήσουμε σε αυτή την πτυχιακή, έχουν ως στόχο να δημιουργήσουν ένα τέτοιο δένδρο. Για να βρούμε την κωδικοποίηση ξεκινάμε από την ρίζα και καταλήγουμε στον κόμβο με τον χαρακτήρα. Για αριστερό θυγατρικό κόμβο σημειώνουμε 0 και για δεξιό 1.

a:1
b:00
c:010
d:0110
e:0111

Πίνακας 1.2.: Ο δυαδικός κώδικας για κάθε χαρακτήρα του Σχήματος 1.1.

Ο κώδικας του Πίνακα 1.2, είναι ένας κώδικας μεταβλητού μήκους. Το μήκος του

κάθε δυαδικού κώδικα είναι ανάλογο με την συχνότητα του κάθε χαρακτήρα. Όπως παρατηρούμε, για την κωδικοποίηση των χαρακτήρων με την μεγαλύτερη συχνότητα χρησιμοποιούμε μικρότερο αριθμό δυφίων απ' ότι στους χαρακτήρες με την μικρότερη συχνότητα.

Τώρα μπορούμε να κωδικοποιήσουμε την λέξη aaabbcde ως 111000001001100111. Ο αριθμός των δυφίων που απαιτούνται για την κωδικοποίηση του μηνύματος είναι 18 δυφία. Στο προηγούμενο παράδειγμα χρησιμοποιώντας ένα κώδικα σταθερού μήκους χρειαστήκαμε 21 δυφία. Να μην ξεχνάμε όμως και το γεγονός ότι μπορούμε να αποκωδικοποιήσουμε το μήνυμα εύκολα διαβάζοντας τα δυφία ένα-ένα καθώς η κάθε κωδικοποίηση μπορεί και ξεχωρίζει. Διαβάζοντας το πρώτο δυφίο 1 παρατηρούμε ότι προκύπτει μόνο ο χαρακτήρας a. Συνεχίζοντας μπορούμε να αποκωδικοποιήσουμε την δυαδική συμβολοσειρά χωρίς πρόβλημα.

Αν όμως χρησιμοποιήσουμε μία τυχαία κωδικοποίηση όπου a:0, b:00, c:10, d:11, e:100, τότε η κωδικοποίηση της λέξης μας θα ήταν 00000001011100 και θα χρειαζόμασταν ακόμα μικρότερο αριθμό δυφίων για την κωδικοποίηση της λέξης μας. Όμως στην συνέχεια, δεν θα μπορούσαμε να αποκωδικοποιήσουμε το μήνυμα καθώς το δύο πρώτα δυφία 00 συμβολίζουν δύο a ή ένα b την ίδια στιγμή. Συμπεραίνουμε λοιπόν ότι όποιος και να είναι ο κώδικας που θα χρησιμοποιούμε, θα πρέπει να έχει υπολογιστεί έτσι ώστε κατά την αποκωδικοποίηση να μπορούμε να βρούμε τον κάθε χαρακτήρα χωρίς προβλήματα.

1.2. Εντροπία

Στο πρόβλημά μας, η εντροπία είναι ένα μέτρο το οποίο ορίζει τον θεωρητικά βέλτιστο αριθμό δυφίων που πρέπει να χρησιμοποιήσουμε για την δυαδική αναπαράσταση του κάθε χαρακτήρα, ανάλογα με την συχνότητα f_i εμφάνισής του.

Σύμφωνα με το θεώρημα του Shannon, ο αριθμός των δυφίων h για κάθε σύμβολο s_i σε ένα κείμενο προκύπτει από την παρακάτω σχέση (1.1).

$$h(s_i)=log_2\frac{1}{f_i} \tag{1.1}$$

Στην συνέχεια η εντροπία H (σε δυφία) είναι το άθροισμα όλων των συμβόλων με συχνότητα $f_i > 0$ και μπορεί να υπολογιστεί από την σχέση (1.2).

$$H(S) = \sum_{f_i>0} f_i h(s_i) \tag{1.2}$$

Για να καταλάβουμε καλύτερα το πως χρησιμοποιούμε την θεωρία της εντροπίας θα δούμε ένα παράδειγμα.

Έστω οι χαρακτήρες a:0,17 - b:0,17 - c:0,2 - d:0,2 - e:0,26 με τις συχνότητες εμφάνισης σε ένα κείμενο.

Σύμφωνα με την σχέση 1.1 και την σχέση 1.2 προκύπτει ο παρακάτω πίνακας.

s_i	a	b	c	d	e
f_i	0.17	0.17	0.2	0.2	0.26
$h(s_i)$	2.56	2.56	2.32	2.32	1.94
$H(S)$ (για κάθε σύμβολο)	0.44	0.44	0.46	0.46	0.51

Αθροίζοντας τα αποτελέσματα της εντροπίας προκύπτει $H(S) \approx 2,31$.

Τώρα, οποιοσδήποτε αλγόριθμος μπορεί και παράγει ένα τέτοιο αποτέλεσμα, τότε θα επιτυγχάνει και θεωρητικά τον βέλτιστο αριθμό δυφίων για κάθε χαρακτήρα.

Αυτό που θα κάνουμε τώρα είναι να συγκρίνουμε τα αποτελέσματα των δύο αλγορίθμων Huffman και ΔΚΜΜ, που θα δούμε στα επόμενα κεφάλαια, σε σχέση με την εντροπία.

Τα αποτελέσματα της εκτέλεσης των δύο αλγορίθμων για το παράδειγμα μας εμφανίζεται στον παρακάτω πίνακα.

s_i	a	b	c	d	e
$f_i - P(i)$	0.17	0.17	0.2	0.2	0.26
ΔΚΜΜ (αριθμός δυφίων) - $L(i)$	3	3	2	2	2
Huffman (αριθμός δυφίων) - $L(i)$	3	3	2	2	2
$P(i)L(i)$	0.51	0.51	0.4	0.4	0.52

Υπολογίζουμε τον μέσο όρο πολλαπλασιάζοντας την συχνότητα $P(i)$ του κάθε χαρακτήρα με αυτή του αριθμού των δυφίων (μήκος δυφίων) $L(i)$. Στο τέλος αθροίζουμε τα αποτελέσματα.

$$L_{av} = \sum_{i=1}^{|c|} P(i)L(i)$$

Ακολουθώντας την σχέση 1.3, προκύπτει ότι ο μέσος όρος των δυφίων που χρησιμοποιούν ο καθένας από τους δύο αλγόριθμους για την αναπαράσταση του κάθε χαρακτήρα είναι $L_{av} = 2,34$.

Όπως παρατηρούμε, ο αριθμός L_{av} είναι πολύ κοντά στο θεωρητικό μέγιστο $H(S) \approx 2,31$.

2. Αλγόριθμος του Huffman

2.1. Εισαγωγή

Ο αλγόριθμος του Huffman είναι ένας από τους σημαντικότερους αλγορίθμους σε ότι αφορά την συμπίεση των δεδομένων. Τόσο σημαντικός, που οι δυαδικές συμβολοσειρές μεταβλητού μήκους ονομάζονται και κώδικες Huffman. Επίσης, πολλοί από τους αλγόριθμους που χρησιμοποιούνται για το ίδιο πρόβλημα ακολουθούν παρόμοιες στρατηγικές.

Όλα ξεκίνησαν από μία δημοσίευση του C. E. Shannon για την εντροπία του καναλιού. Στην συνέχεια ο τότε καθηγητής του MIT R. M. Fano βασιζόμενος στις παρατηρήσεις του Shannon, δημιούργησε τον πρώτο αλγόριθμο για την κωδικοποίηση του κάθε χαρακτήρα χρησιμοποιώντας συμβολοσειρές μεταβλητού μήκους. Ο αλγόριθμος αυτός σχημάτιζε ένα δυαδικό δένδρο από πάνω προς τα κάτω κάτι που δεν οδηγούσε πάντοτε στο βέλτιστο αποτέλεσμα. Ο αλγόριθμος αυτός ονομάστηκε ως αλγόριθμος των Shannon-Fano ή ισοδύναμα κωδικοποίηση Shannon-Fano.

Ο David A. Huffman, που έκανε τότε το διδακτορικό του στο MIT, ήταν μαθητής του Fano. Ο Fano, σαν καθηγητής, έδωσε στους φοιτητές του δύο επιλογές για να καταφέρουν να περάσουν το μάθημά του. Η πρώτη ήταν η τυπική, δηλαδή να δώσουν εξετάσεις. Η δεύτερη ήταν να δημιουργήσουν ένα αλγόριθμο, για το πρόβλημα της κωδικοποίησης του κάθε χαρακτήρα χρησιμοποιώντας συμβολοσειρές μεταβλητού μήκους. Σε αντίθεση με τον δικό του αλγόριθμο, ο νέος θα έπρεπε να δίνει πάντοτε το βέλτιστο αποτέλεσμα.

Μετά από πολύ προσπάθεια, ο Huffman ήταν αυτός που κατάφερε να δημιουργήσει ένα τέτοιο αλγόριθμο. Ο αλγόριθμος του Huffman σε αντίθεση με αυτόν των Shannon-Fano σχημάτιζε ένα δυαδικό δένδρο από κάτω προς τα πάνω.

2.2. Περιγραφή Αλγορίθμου

Ο αλγόριθμος που θα περιγράψουμε στις επόμενες παραγράφους χρησιμοποιεί ένα σωρό ελαχίστου και ακλουθεί μία άπληστη στρατηγική προκειμένου να παράγει πάντοτε το βέλτιστο αποτέλεσμα.

Η είσοδος του προβλήματος είναι ένα σύνολο από χαρακτήρες μαζί με την συχνότητα εμφάνισης τους σε ένα κείμενο. Ο αλγόριθμος επιστρέφει στην έξοδο την βέλτιστη δυαδική αναπαράσταση μεταβλητού μήκους του κάθε χαρακτήρα.

Θα ξεκινήσουμε παρουσιάζοντας τον ψευδοκώδικα και στην συνέχεια θα δούμε την εκτέλεση του σε ένα παράδειγμα.

Αλγόριθμος 2.1 Ο αλγόριθμος του Huffman υπό την μορφή ψευδοκώδικα.

HUFFMAN (C)

1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. **για** $i \leftarrow 1$ **έως** $n - 1$
4. δεσμεύουμε ένα νέο κόμβο z
5. αριστερός $[z] \leftarrow x \leftarrow$ ΕΞΑΓΩΓΗ ΕΛΑΧΙΣΤΟΥ (Q)
6. δεξιός $[z] \leftarrow y \leftarrow$ ΕΞΑΓΩΓΗ ΕΛΑΧΙΣΤΟΥ (Q)
7. $f[z] \leftarrow f[x] + f[y]$
8. ΕΙΣΑΓΩΓΗ (Q, z)
9. **επιστροφή** ΕΞΑΓΩΓΗ ΕΛΑΧΙΣΤΟΥ (Q)

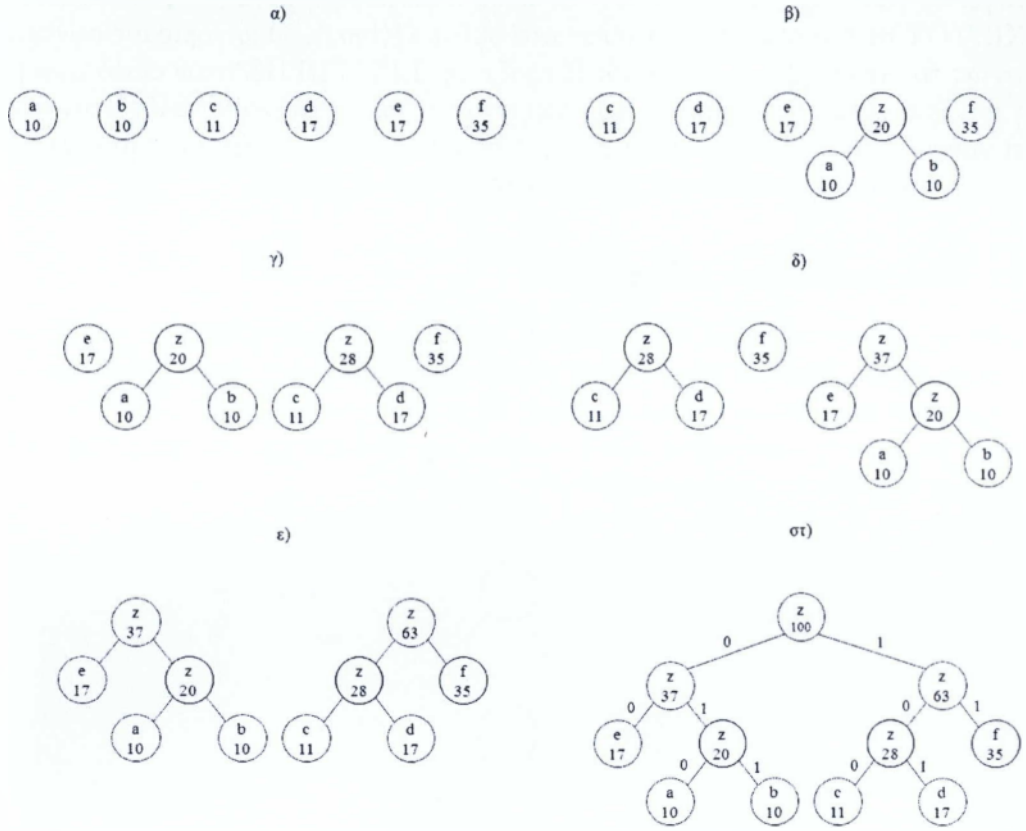
Σύμφωνα με τον Αλγόριθμο 2.1, το n (γραμμή 1) ισούται με το πλήθος τον χαρακτήρων ενός αλφαβήτου C που εμφανίζονται σε ένα κείμενο. Στην συνέχεια, (γραμμή 2) ανάλογα με τις συχνότητες του κάθε χαρακτήρα, δημιουργούμε μία ουρά προτεραιότητας Q ή οποία και υλοποιείται ως δυαδικός σωρός ελαχίστου. Στην συνέχεια (γραμμή 3) για $n - 1$ βήματα εκτελούμε μια συγκεκριμένη διαδικασία. Στην αρχή (γραμμή 4) δημιουργούμε ένα νέο κόμβο z. Συνεχίζουμε (γραμμή 5-6) ορίζοντας ως παιδιά του z τα δύο στοιχεία της Q με την μικρότερη συχνότητα. Η συχνότητα του νέου κόμβου z (γραμμή 7) ισούται με το άθροισμα των συχνοτήτων των παιδιών του. Στην συνέχεια εισάγουμε τον κόμβο z (γραμμή 8) στην κατάλληλη θέση της Q. Εφόσον επαναλάβουμε την ίδια διαδικασία για $n - 1$ βήματα επιστρέφεται (γραμμή 9) ή ρίζα του δυαδικού δένδρου, από το οποίο μπορούμε πλέον να βρούμε την κωδικοποίηση του κάθε χαρακτήρα.

Για να καταλάβουμε καλύτερα την λειτουργία του αλγορίθμου θα την περιγράψουμε σχηματικά σε ένα παράδειγμα. Για το παράδειγμα που ακολουθεί υποθέτουμε ότι μετά από την μέτρηση των χαρακτήρων σε ένα κείμενο προκύπτουν τα ακόλουθα αποτελέσματα.

a:10	b:10	c:11	d:17	e:17	f:35

Πίνακας 2.1.: Συχνότητες εμφάνισης χαρακτήρων σε ένα κείμενο.

Στον Πίνακα 2.1, έχουμε μερικούς χαρακτήρες μαζί με τις συχνότητες εμφάνισης τους σε ένα κείμενο. Οι χαρακτήρες εμφανίζονται ταξινομημένοι κατά αύξουσα σειρά ως προς την συχνότητα.



Σχήμα 2.1.: Η πορεία του αλγορίθμου του Huffman για n=6.

Στο Σχήμα 2.1, έχουμε την εκτέλεση του αλγορίθμου του Huffman για τους χαρακτήρες και τις συχνότητες που είδαμε στο παράδειγμα του πίνακα 2.1. Στην αρχή (δένδρο α) έχουμε τον σχηματισμό της ουράς προτεραιότητας Q. Οι χαρακτήρες μετατρέπονται σε κόμβους και ταξινομούνται ως προς την συχνότητα. Στην συνέχεια (δένδρο β) έχουμε την δημιουργία ενός νέου κόμβου z. Τα παιδιά του z είναι οι δύο κόμβοι με την μικρότερη συχνότητα. Το z εισάγεται στην κατάλληλη θέση της Q και είναι πλέον ένας νέος κόμβος με συχνότητα το άθροισμα των παιδιών του. Η διαδικασία επαναλαμβάνεται (δένδρο γ-ε) για $n-1$ βήματα έως ότου σχηματιστεί (δένδρο στ) το τελικό δυαδικό δένδρο από το οποίο και μπορούμε εύκολα να βρούμε την κωδικοποίηση του κάθε χαρακτήρα.

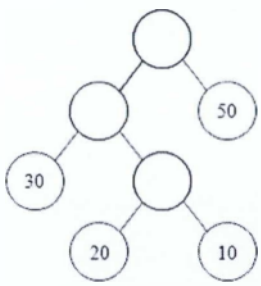
2.3. Ανάλυση Αλγορίθμου

2.3.1. Χρόνος Εκτέλεσης

Για να αναλύσουμε τον χρόνο εκτέλεσης του αλγορίθμου θεωρούμε ότι η ουρά προτεραιότητας Q υλοποιείται ως δυαδικός σωρός ελαχίστου. Το κόστος μίας τέτοιας διαδικασίας είναι $O(n)$. Στην συνέχεια ο αλγόριθμος δημιουργεί ένα νέο κόμβο z και εξάγει τα δύο μικρότερα στοιχεία. Σε κάθε περίπτωση η πράξη της ΕΞΑΓΩΓΗΣ ΕΛΑΧΙΣΤΟΥ σε ένα σωρό ελαχίστου απαιτεί χρόνο $O(lgn)$[1]. Ο αλγόριθμος συνεχίζει εισάγοντας τον νέο κόμβο z στο σωρό. Η πράξη της ΕΙΣΑΓΩΓΗΣ στον σωρό είναι μία πράξη που απαιτεί επίσης χρόνο $O(lgn)$. Λαμβάνοντας υπόψη ότι κάθε πράξη στον σωρό απαιτεί χρόνο $O(lgn)$ και ότι ο αλγόριθμος εκτελείται για $n-1$ βήματα συμπεραίνουμε ότι ο συνολικός χρόνος εκτέλεσης του αλγορίθμου είναι $O(nlgn)$.

2.3.2. Βέλτιστο Αποτέλεσμα

Όπως θα δούμε στην συνέχεια, ο αλγόριθμος του Huffman παράγει πάντοτε το βέλτιστο αποτέλεσμα καθώς διαθέτει την ιδιότητα της άπληστης επιλογής. Αυτό σημαίνει ότι σε κάθε βήμα, από ένα σύνολο επιλογών, αυτός κάνει εκείνη με το μικρότερο κόστος. Θα περιγράψουμε γιατί ο αλγόριθμος αυτός διαθέτει την ιδιότητα της άπληστης επιλογής, χωρίς να μας ενδιαφέρει η μαθηματική αυστηρότητα.



Σχήμα 2.2.: Ένα δυαδικό δένδρο με 4 καταληκτικούς κόμβους.

Στο Σχήμα 2.2, έχουμε ένα βέλτιστο δένδρο από το οποίο μπορούμε να παράγουμε δυαδικό κώδικα μεταβλητού μήκους. Κάθε καταληκτικός κόμβος εμπεριέχει και μία συχνότητα. Υποθέτουμε ότι η συχνότητες αυτές ανήκουν σε κάποιους τυχαίους χαρακτήρες. Από το δένδρο προκύπτει ότι ο αριθμός των δυφίων που απαιτούνται για την κωδικοποίηση αυτών των χαρακτήρων είναι $1*50+2*30+3*20+3*10 = 200$ δυφία. Αν τώρα εναλλάξουμε τον κόμβο με την συχνότητα 50 και αυτόν με την συχνότητα 10 τότε ο αριθμός των δυφίων που απαιτούνται για την κωδικοποίηση των χαρακτήρων θα αυξηθεί. Το ίδιο θα συμβεί για οποιαδήποτε εναλλαγή. Όπως παρατηρούμε στο δένδρο μας οι κόμβοι μεγίστου βάθους εμπεριέχουν τους χαρακτήρες με την μικρότερη συχνότητα και

[1] Ο λογάριθμος, όπου εμφανίζεται, έχει ως βάση το δύο.

οι κόμβοι ελαχίστου βάθους τους χαρακτήρες με την μεγαλύτερη συχνότητα. Ο αλγόριθμος του **Huffman** μας εξασφαλίζει κάτι τέτοιο λόγω της στρατηγικής που ακολουθεί κατά την συγχώνευση των κόμβων. Άρα, η άπληστη επιλογή που κάνει ο Huffman σε κάθε βήμα, είναι να συγχωνεύει πάντοτε τους χαρακτήρες με την μικρότερη συχνότητα.

Ο αλγόριθμός του **Huffman** εκτός από την ιδιότητα της άπληστης επιλογής διαθέτει επίσης και αυτή της βέλτιστης υποδομής. Ένα πρόβλημα διαθέτει βέλτιστη υποδομή αν μια βέλτιστη λύση του περιέχει βέλτιστες λύσεις υποπροβλημάτων. Αυτό σημαίνει ότι, η άπληστη επιλογή του αλγορίθμου οδηγεί σε μία καθολικά βέλτιστη λύση.

3. Αλγόριθμος ΔΚΜΜ

3.1. Εισαγωγή

Σε αυτό το κεφάλαιο θα προσπαθήσουμε να δημιουργήσουμε ένα αλγόριθμο ο οποίος θα μπορεί να ανταγωνιστεί τον αντίστοιχο αλγόριθμο του Huffman, τον οποίο και αναλύσαμε στο προηγούμενο κεφάλαιο. Στην συνέχεια θα επιχειρήσουμε να υλοποιήσουμε τον αλγόριθμο μας, στην πιο απλή του μορφή, στην γλώσσα C. Ο αλγόριθμος που θα παρουσιάσουμε στο κεφάλαιο αυτό ονομάζεται αλγόριθμος Δημιουργίας Κώδικα Μεταβλητού Μήκους (ΔΚΜΜ). Στόχος του είναι να δημιουργεί βέλτιστους δυαδικούς κώδικες μεταβλητού μήκους.

Όσον αφορά το πρόβλημα, παραμένει το ίδιο με αυτό που επιλύει ο αλγόριθμος του Huffman. Έχοντας ένα κείμενο, στην αρχή μετράμε την συχνότητα εμφάνισης του κάθε χαρακτήρα σε αυτό. Στην συνέχεια προσπαθούμε να κωδικοποιήσουμε τον κάθε χαρακτήρα χρησιμοποιώντας «βέλτιστες» δυαδικές συμβολοσειρές μεταβλητού μήκους.

Εκτός από την διαδικασία Ομαδοποίηση, η οποία μπορεί να θεωρηθεί όμοια με αυτή του αλγόριθμου του Huffman, ο αλγόριθμος ΔΚΜΜ δεν αποτελεί αντιγραφή και είναι προϊόν μίας προσπάθειας που γίνεται εδώ και δύο χρόνια. Επίσης, από τους πολλούς αλγορίθμους που έχουν επινοηθεί κατά καιρούς για το πρόβλημα αυτό, επιλέχθηκε ένας από τους καλύτερους, αν όχι ο καλύτερος, για να συγκριθεί με τον αλγόριθμο ΔΚΜΜ.

Τέλος, η δημιουργία ενός αλγορίθμου που να μπορεί να ανταγωνιστεί τους καλύτερους στον κόσμο είναι μία διαδικασία ιδιαίτερα δύσκολη και απαιτεί πολύ χρόνο και κόπο. Συνεπώς, σκοπός του κεφαλαίου αυτού δεν είναι να παρουσιάσει τον καλύτερο αλγόριθμο στον κόσμο αλλά κάτι το ανταγωνιστικό.

3.2. Περιγραφή Αλγορίθμου

3.2.1. Διαδικασία Ομαδοποίηση

Η διαδικασία ομαδοποίηση αποτελεί το πρώτο από τα δύο κύρια βήματα του αλγορίθμου ΔΚΜΜ. Σκοπός της διαδικασίας αυτής είναι να μετασχηματίσει, αν αυτό είναι απαραίτητο, οποιαδήποτε είσοδο του προβλήματος σε συγκεκριμένη μορφή.

Έτσι λοιπόν, ο αλγόριθμος Ομαδοποίηση είναι υπεύθυνος για την μετατροπή όλων των συχνοτήτων σε μορφή κατάλληλη, τέτοια ώστε να θεωρούνται ίσες μεταξύ τους. Ο αλγόριθμος θα εκτελείται έως ότου ικανοποιηθεί η παρακάτω σχέση.

$$P(1) + P(2) \geq P(N)$$

Σύμφωνα με την παραπάνω σχέση, είμαστε ικανοποιημένοι, αν το άθροισμα των δύο μικρότερων συχνοτήτων είναι μεγαλύτερο ή ίσο από την συχνότητα με την μεγαλύτερη τιμή. Στην περίπτωση που αυτή η σχέση δεν ικανοποιείται ο αλγόριθμος θα σταματά να εκτελείται μετά από $n-3$ βήματα.

Εκτέλεση της ομαδοποίησης

Για να καταλάβουμε το πως λειτουργεί η Ομαδοποίηση θα παρουσιάσουμε πρώτα τον αλγόριθμο υπο την μορφή ψευδοκώδικα και μετά θα δούμε την εκτέλεση του σε ένα παράδειγμα.

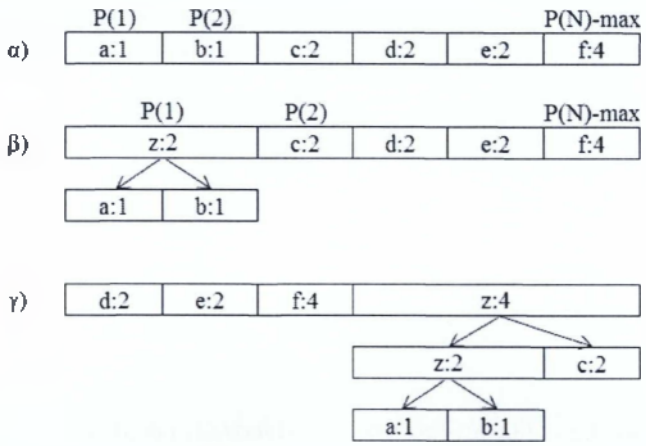
Αλγόριθμος 3.1 Ο αλγόριθμος Ομαδοποίηση υπό την μορφή ψευδοκώδικα.

ΟΜΑΔΟΠΟΙΗΣΗ (C)

1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. $max \leftarrow$ μέγιστη συχνότητα Q
4. **για** $i \leftarrow 1$ **έως** $n-3$
5. δεσμεύουμε ένα νέο κόμβο z
6. αριστερός $[z] \leftarrow x \leftarrow$ ΕΞΑΓΩΓΗ ΕΛΑΧΙΣΤΟΥ (Q)
7. δεξιός $[z] \leftarrow y \leftarrow$ ΕΞΑΓΩΓΗ ΕΛΑΧΙΣΤΟΥ (Q)
8. $f[z] \leftarrow f[x] + f[y]$
9. **Αν** $(f[z] < max)$
10. ΕΙΣΑΓΩΓΗ (Q, z)
11. **Άλλως**
12. τοποθέτηση του κόμβου z στο τέλος της Q
13. έξοδος απο τον βρόχο
14. Η ομαδοποίηση ολοκληρώθηκε

Σύμφωνα με τον παραπάνω αλγόριθμο 3.1, ο οποίος δεν διαφέρει και πολύ από τον αντίστοιχο του Huffman, το n (γραμμή 1) ισούται με το πλήθος τον χαρακτήρων ενός αλφαβήτου C που εμφανίζονται σε ένα κείμενο. Ορίζουμε (γραμμή 2) ως αρχικό περιεχόμενο της Q το σύνολο C των χαρακτήρων. Επίσης (γραμμή 3) ορίζουμε μία μεταβλητή

max η τιμή της οποίας είναι αυτή του χαρακτήρα με τη μεγαλύτερη συχνότητα. Στην συνέχεια (γραμμές 5-8) δημιουργούμε ένα νέο κόμβο z παιδία του οποίου είναι οι δύο κόμβοι της Q με τη μικρότερη συχνότητα. Η συχνότητα του νέου κόμβου z είναι το άθροισμα των συχνοτήτων των παιδιών του. Έπειτα (γραμμή 9) ελέγχεται αν η συχνότητα του νέου κόμβου z είναι μικρότερη από αυτή της μεταβλητής max. Αν η συχνότητα του κόμβου z είναι μικρότερη από την τιμή max τότε ο κόμβος z εισάγεται (γραμμή 10) στην κατάλληλη θέση της ουράς Q. Η διαδικασία ολοκληρώνεται (γραμμή 11-14) αν η τιμή του z είναι μεγαλύτερη ή ίση από αυτή της μεταβλητής max ή αν $n = 3$.



Σχήμα 3.1.: Η εκτέλεση του αλγορίθμου Ομαδοποίηση για $n = 6$.

Όπως παρατηρούμε στο Σχήμα 3.1, η διαδικασία ομαδοποίηση εκτελείται για 2 βήματα προκειμένου να μετατρέψει τις συχνότητες στην επιθυμητή για μας μορφή. Τα περιεχόμενα (πίνακας α) της Q ταξινομημένα κατά αύξουσα σειρά. Στην συνέχεια (πίνακας β) δημιουργείται ένας νέος κόμβος z με παιδιά τους κόμβους με τους χαρακτήρες a, b και με συχνότητα το άθροισμα των συχνοτήτων των παιδιών του. Η συχνότητα του z είναι μικρότερη από αυτή της μεταβλητής max και κατά συνέπεια ο νέος κόμβος τοποθετείται στην κατάλληλη θέση, ώστε να διατηρούνται τα στοιχεία της Q ταξινομημένα. Στο τελευταίο βήμα (πίνακας γ) έχουμε ξανά την δημιουργία ενός κόμβου z αυτή την φορά όμως η συχνότητα του είναι μεγαλύτερη ή ίση από αυτή της μεταβλητής max. Έτσι, ο κόμβος z τοποθετείται στην τελευταία θέση της Q και ο αλγόριθμος τερματίζει.

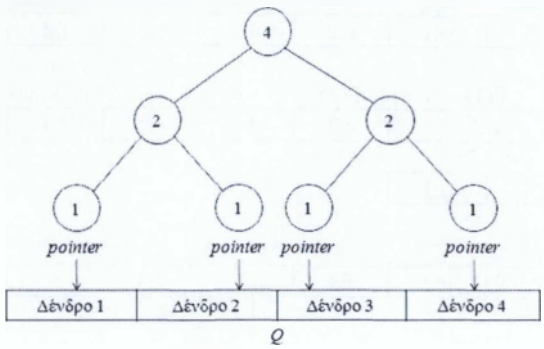
Η εκτέλεση της διαδικασίας της Ομαδοποίησης έχει πολύ μεγάλη σημασία και εφόσον ολοκληρωθεί, έχουμε πλέον ένα σύνολο δυαδικών δένδρων[1] τα οποία μπορεί να διαθέτουν ένα ή περισσότερους κόμβους. Το σημαντικό είναι ότι μέσω του αλγορίθμου αυτού, οποιαδήποτε είσοδος του προβλήματος μετασχηματίζεται στην μορφή που εμείς επιθυμούμε.

[1]Θεωρούμε όλους του κόμβους ως δυαδικά δέντρα ακόμα και αν δεν διαθέτουν παιδιά.

3.2.2. Διαδικασία Διαίρεση

Η διαδικασία Διαίρεση αποτελεί το δεύτερο κύριο κομμάτι του αλγορίθμου ΔΚΜΜ. Στόχος του αλγορίθμου αυτού είναι να ενώσει τα δυαδικά δένδρα που δημιουργήθηκαν από την προηγούμενη διαδικασία «Ομαδοποίηση». Το πλεονέκτημα της «Διαίρεσης» είναι ότι χρειάζεται να δημιουργήσει μόνο ένα πλήρες δυαδικό δένδρο.

Με τον όρο «πλήρες» ονομάζουμε ένα δυαδικό δένδρο του οποίου κάθε πατρικός κόμβος διαθέτει πάντοτε δύο παιδία και τα καταληκτικά του επίπεδα έχουν το πολύ διαφορά ύψους 1. Χρησιμοποιώντας ένα τέτοιο δένδρο έχουμε ένα πρόσθετο πλεονέκτημα. Δεν χρειάζεται πλέον να κάνουμε πράξεις με τις συχνότητες των χαρακτήρων. Η μόνη απαραίτητη πληροφορία, είναι το πλήθος των στοιχείων της ουράς προτεραιότητας Q.



Σχήμα 3.2.: Το δένδρο της «Διαίρεσης» για $n = 4$.

Στο Σχήμα 3.2, παρατηρούμε ένα πλήρες δυαδικό δένδρο για $n = 4$. Ο αλγόριθμος ξεκινά δημιουργώντας ένα ριζικό κόμβο η τιμή του οποίου είναι αυτή του πλήθους των στοιχείων της Q. Σε κάθε βήμα ο αλγόριθμος δημιουργεί δύο παιδία για κάθε κόμβο του οποίου η τιμή είναι διαφορετική του 1. Το αριστερό παιδί του κόμβου πατέρα προκύπτει από το άνω ακέραιο μέρος της διαίρεσης της τιμής του πατρικού κόμβου με το 2. Το δεξιό παιδί προκύπτει από το κάτω ακέραιο μέρος της ίδιας διαίρεσης. Κάθε φορά που ένας κόμβος αποκτά την τιμή 1 του τοποθετούμε έναν δείκτη ο οποίος και δείχνει προς ένα δένδρο της Q. Όταν αποκτήσουν όλοι οι κόμβοι την τιμή 1 το δένδρο της διαδικασίας Διαίρεση έχει πλέον ενωθεί με αυτά της διαδικασίας Ομαδοποίηση. Έπειτα μπορούμε να διαβάσουμε το τελικό δένδρο όπως και ένα δένδρο της κωδικοποίησης Huffman.

Αν και στο Σχήμα 3.2 περιγράψαμε την αρχική ιδέα για την υλοποίηση της διαδικασίας Διαίρεση σε αυτή την εργασία θα ακολουθήσουμε μία διαφορετική στρατηγική στην πρακτική εφαρμογή του αλγόριθμου. Ο λόγος για τον οποίο θα αλλάξουμε την «Διαίρεση» είναι για να μπορέσουμε να την υλοποιήσουμε ευκολότερα στην γλώσσα C.

Εκτέλεση της διαίρεσης

Ο αλγόριθμος που θα χρησιμοποιήσουμε για την διαδικασία Διαίρεση, είναι εύκολα υλοποιήσιμος στην γλώσσα C και δεν διαφέρει και πολύ από αυτόν που θα ακολουθήσουμε για να υλοποιήσουμε την κωδικοποίηση Huffman.

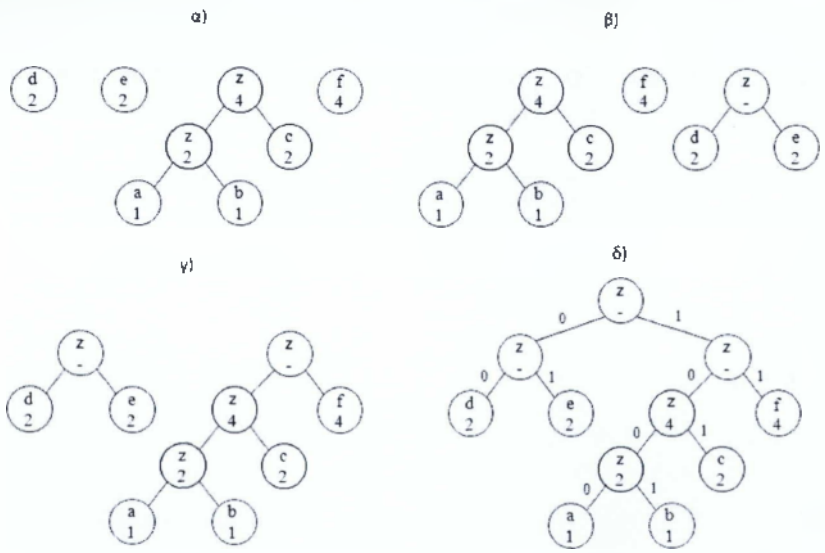
Αλγόριθμος 3.2 Βασικό μέρος του αλγορίθμου Διαίρεση σε ψευδοκώδικα.

ΔΙΑΙΡΕΣΗ (Q)

1. **για** $i \leftarrow 1$ **έως** $n-1$
2. δεσμεύουμε ένα νέο κόμβο z
3. αριστερός $[z] \leftarrow x \leftarrow$ ΕΞΑΓΩΓΗ ΠΡΩΤΟΥ ΣΤΟΙΧΕΙΟΥ (Q)
4. δεξιός $[z] \leftarrow x \leftarrow$ ΕΞΑΓΩΓΗ ΠΡΩΤΟΥ ΣΤΟΙΧΕΙΟΥ (Q)
5. ΕΙΣΑΓΩΓΗ z ΣΤΟ ΤΕΛΟΣ(Q)
6. **επιστροφή** Q

Σύμφωνα με τον Αλγόριθμο 3.2, για n-1 (γραμμή 1) βήματα, ξεκινάμε (γραμμή 2) δημιουργώντας ένα νέο κόμβο z. Τα παιδιά του κόμβου z (γραμμή 3-4) είναι τα δύο πρώτα στοιχεία της Q. Αυτό που έχει μεγάλη σημασία είναι ότι δεν μας ενδιαφέρει αν οι συχνότητες που εξάγουμε είναι οι μικρότερες. Επίσης, δεν μας ενδιαφέρει η συχνότητα του νέου κόμβου z να είναι το άθροισμα των συχνοτήτων των παιδιών του. Στην συνέχεια τοποθετούμε τον κόμβο z (γραμμή 5) απ' ευθείας στο τέλος της ουράς. Η διαδικασία επαναλαμβάνεται έως ότου ένας κόμβος απομείνει στην Q. Ο κόμβος που απέμεινε αποτελεί την ρίζα του δένδρου από το οποίο μπορούμε να βρούμε τον βέλτιστο δυαδικό κώδικα μεταβλητού μήκους για κάθε χαρακτήρα.

Για το παράδειγμα που ακολουθεί επιλέξαμε να συνεχίσουμε με ένα παράδειγμα παρόμοιο με αυτό στο Σχήμα 3.1 της διαδικασίας Ομαδοποίηση.



Σχήμα 3.3.: Η εκτέλεση του αλγορίθμου Διαίρεση για $n = 4$.

Στο Σχήμα 3.3, (δένδρο α) έχουμε την αρχική είσοδο του προβλήματος μετά την ολοκλήρωση του αλγορίθμου Ομαδοποίηση.(δένδρο β-γ) Ακολουθώντας τον αλγόριθμο, δημιουργούμε ένα νέο κόμβο z και αφαιρούμε τα δύο πρώτα στοιχεία της Q. Τα δύο πρώτα στοιχεία της Q γίνονται παιδιά του z. Στην συνέχεια τοποθετούμε τον κόμβο z και τους κόμβους παιδιά του στο τέλος της ουράς προτεραιότητας.(δένδρο δ) Ο αλγόριθμος συνεχίζει να εκτελείται συγχωνεύοντας τα δένδρα που απέμειναν και τερματίζει επιστρέφοντας ολοκληρωμένο το δυαδικό δένδρο.

Τώρα, όπως και στην κωδικοποίηση **Huffman**, μπορούμε να βρούμε τον κώδικα του κάθε χαρακτήρα ξεκινώντας από την ρίζα του δένδρου. Από τον ριζικό κόμβο έως τον κάθε χαρακτήρα τοποθετούμε «0» αν πρόκειται για αριστερό θυγατρικό κόμβο και «1» αν πρόκειται για δεξιό.

3.2.3. Υπολογισμός Συντελεστή Ταξινόμησης

Όπως είδαμε στις προηγούμενες παραγράφους ο αλγόριθμος ΔΚΜΜ χρησιμοποιεί δύο διαδικασίες. Στο πρώτο βήμα έχουμε την διαδικασία Ομαδοποίηση και στο δεύτερο την διαδικασία Διαίρεση. Ο αλγόριθμος δουλεύει καλά αν πρώτα εξασφαλίσουμε ότι οι χαρακτήρες είναι ταξινομημένοι κατά αύξουσα σειρά ως προς την συχνότητα. Κάτι τέτοιο όμως δεν απαιτείται στον αλγόριθμο του Huffman καθώς η δημιουργία ενός σωρού ελαχίστου είναι το μόνο που χρειάζεται. Συνεπώς εμείς πρέπει να ταξινομήσουμε τα στοιχεία, μία πράξη που απαιτεί χρόνο $O(nlgn)$, ενώ στην κωδικοποίηση Huffman πρέπει να δημιουργηθεί μόνο ένας σωρός ελαχίστου, πράξη που απαιτεί χρόνο $O(n)$. Αυτό αποτελεί για τον αλγόριθμο ΔΚΜΜ ένα σοβαρό μειονέκτημα.

Για να ξεπεράσουμε αυτό το πρόβλημα θα υπολογίζουμε μία σταθερά k σύμφωνα με την οποία θα εκτελείται η διαδικασία Διαίρεση. Ο υπολογισμός της σταθεράς k είναι μία διαδικασία υπολογιστικά απλή. Η διαδικασία Ομαδοποίηση παραμένει αμετάβλητη. Αυτή την φορά όμως μετά την εκτέλεση της, ο αριθμός των στοιχείων της Q είναι ιδιαίτερα σημαντικός.

Ο αλγόριθμος που υπολογίζει την σταθερά k είναι μία διαδικασία που ονομάζεται υπολογισμός συντελεστή ταξινόμησης.

Αλγόριθμος 3.3 Υπολογισμός συντελεστή ταξινόμησης.

---

1. $n \leftarrow |Q|$
2. $k = 2$
3. **ενόσω** $k \leq n$
4. $\quad k \leftarrow k * 2$
5. $k \leftarrow k/2$
6. **επιστροφή** k

Σύμφωνα με τον Αλγόριθμο 3.3, αν το πλήθος των στοιχείων της Q είναι 10 τότε το k θα πολλαπλασιαστεί 4 φορές (γραμμή 3-4) δίνοντας μας το αποτέλεσμα 16. Στην συνέχεια (γραμμή 5) το k διαιρείται με το 2. Ο αλγόριθμος (γραμμή 6) τερματίζει επιστρέφοντας την τιμή του k που στην περίπτωσή μας είναι το 8.

Η σταθερά k είναι το πλήθος των στοιχείων της Q για το οποίο θα αρχίσει να εκτελείται η Διαίρεση. Αν το πλήθος των στοιχείων είναι μεγαλύτερο από αυτόν τον αριθμό τότε θα εκτελείται η Ομαδοποίηση.

Έτσι λοιπόν, στο παράδειγμα μας για $n = 10$ τότε $k = 8$ και έτσι η Ομαδοποίηση θα εκτελεστεί για $n - k = 2$ βήματα επιπλέον προτού ξεκινήσει η διαδικασία της Διαίρεσης.

3.2.4. Ψευδοκώδικας Αλγορίθμου ΔΚΜΜ

Ξεκινάμε περιγράφοντας ολοκληρωμένα τον αλγόριθμο της Διαίρεσης. Στην συνέχεια, ο αλγόριθμος ΔΚΜΜ είναι το αποτέλεσμα της Ομαδοποίησης και της Διαίρεσης.

Αλγόριθμος 3.4 Ψευδοκώδικας για την Διαίρεση.

ΔΙΑΙΡΕΣΗ (Q)

1. $n \leftarrow |Q|$
2. Υπολογισμός συντελεστή ταξινόμησης (k)
3. **ενόσω** $n > k$
4. δεσμεύουμε ένα νέο κόμβο z
5. αριστερός $[z] \leftarrow x \leftarrow$ ΕΞΑΓΩΓΗ ΕΛΑΧΙΣΤΟΥ (Q)
6. δεξιός $[z] \leftarrow y \leftarrow$ ΕΞΑΓΩΓΗ ΕΛΑΧΙΣΤΟΥ (Q)
7. $f[z] \leftarrow f[x] + f[y]$
8. ΕΙΣΑΓΩΓΗ z ΣΤΟ ΤΕΛΟΣ(Q)
9. $n \leftarrow n - 1$
10. **για** $i \leftarrow 1$ **έως** $k - 1$
11. δεσμεύουμε ένα νέο κόμβο z
12. αριστερός $[z] \leftarrow x \leftarrow$ ΕΞΑΓΩΓΗ ΠΡΩΤΟΥ ΣΤΟΙΧΕΙΟΥ (Q)
13. δεξιός $[z] \leftarrow x \leftarrow$ ΕΞΑΓΩΓΗ ΠΡΩΤΟΥ ΣΤΟΙΧΕΙΟΥ (Q)
14. ΕΙΣΑΓΩΓΗ z ΣΤΟ ΤΕΛΟΣ(Q)
15. **επιστροφή** Q

Αλγόριθμος 3.5 Ο αλγόριθμος ΔΚΜΜ υπό την μορφή ψευδοκώδικα.

ΔΚΜΜ (C)

1. ΟΜΑΔΟΠΟΙΗΣΗ (C)
2. ΔΙΑΙΡΕΣΗ (Q)

3.3. Ανάλυση Αλγορίθμου

3.3.1. Χρόνος Εκτέλεσης

Στο σημείο αυτό θα αναλύσουμε την πολυπλοκότητα του αλγορίθμου ΔΚΜΜ. Ο χρόνος εκτέλεσης του αλγόριθμου ΔΚΜΜ είναι ανάλογος με την είσοδο του προβλήματος. Ανάλογα το στιγμιότυπο[2] του προβλήματος, μετράμε την επίδοση του αλγορίθμου στην καλύτερη και την χειρότερη περίπτωση. Θα υπολογίσουμε επίσης και την περίπτωση εκτέλεσης μεταξύ αυτών των δύο.

Αυτό που παραμένει σταθερό για οποιαδήποτε περίπτωση είναι η μέτρηση της συχνότητας εμφάνισης του κάθε χαρακτήρα σε ένα κείμενο. Έχοντας τις συχνότητες εμφάνισης των χαρακτήρων του κειμένου που θέλουμε να κωδικοποιήσουμε, ξεκινάμε δημιουργώντας μία ουρά προτεραιότητας την οποία και υλοποιούμε ως ένα δυαδικό σωρό ελαχίστου. Η διαδικασία κατασκευής ενός σωρού ελαχίστου για n στοιχεία απαιτεί χρόνο $O(n)$.

Καλύτερη περίπτωση

Έστω ότι οι συχνότητες είναι ταξινομημένες. Όταν το άθροισμα των δύο μικρότερων συχνοτήτων είναι μικρότερο ή ίσο από την μεγαλύτερη συχνότητα, άρα ισχύει ότι $Pr(min1) + Pr(min2) \geq Pr(max)$, τότε όπως είδαμε και στις προηγούμενες παραγράφους έχουμε τον τερματισμό της διαδικασίας Ομαδοποίηση.

Η διαδικασία Ομαδοποίηση θα εκτελεστεί μόνο για ένα βήμα. Το κόστος της είναι να δημιουργήσει ένα κόμβο z, να εξάγει τα δύο μικρότερα στοιχεία της Q, στην συνέχεια να τα προσθέσει και να εισάγει τον νέο κόμβο στο τέλος του σωρού. Η διαδικασία της εξαγωγής ενός στοιχείου από τον σωρό ελαχίστου έχει κόστος $O(lgn)$. Η εισαγωγή του νέου κόμβου στο τέλος της Q απαιτεί σταθερό χρόνο $O(1)$.

Η διαδικασία Διαίρεση δημιουργεί ένα κόμβο z, εξάγει τα δύο πρώτα στοιχεία, όποια και να είναι αυτά και τα ορίζει ως παιδιά του z. Στην συνέχεια τοποθετεί τον κόμβο z στο τέλος της Q. Όλες αυτές οι διαδικασίες απαιτούν σταθερό χρόνο $O(1)$.

Τώρα, έχουμε τον υπολογισμό της σταθεράς k. Η σταθερά k προκύπτει από μια σειρά πολλαπλασιασμών και μία διαίρεση. Σε κάθε περίπτωση επειδή διαιρέσεις και πολλαπλασιασμοί με το 2 είναι πράξεις που μπορούν να γίνουν πολύ γρήγορα σε ένα επεξεργαστή, θεωρούμε το κόστος τους αμελητέα ποσότητα. Ενδεικτικό είναι το παράδειγμα ότι για $n = 128$ θα πραγματοποιηθούν μόλις 8 πολλαπλασιασμοί και μία διαίρεση. Αν

[2]Με τον όρο αυτό ορίζουμε μία συγκεκριμένη είσοδο του προβλήματος.

λοιπόν κατά τον υπολογισμό του συντελεστή ταξινόμησης το k που προκύπτει ισούται με μία περίπτωση της σχέσης 3.1, τότε εκτελείται η Διαίρεση, το κόστος της οποίας, σε κάθε περίπτωση είναι $O(n)$ για n στοιχεία.

$$n = k = 2^x \quad \text{ή} \quad n = k = 2^x - 1 \quad \text{όπου} \quad x \quad \text{ακέραιος αριθμός} \quad \geq 1 \tag{3.1}$$

Έκτος από το πρώτο βήμα της διαδικασίας Ομαδοποίηση, η οποία εκτελεί 2 μόνο πράξεις με κόστος $O(lgn)$ η καθεμία, όλες οι υπόλοιπες εργασίες εκτελούνται σε σταθερό χρόνο $O(1)$.

Συμφώνα με τον χρόνο εκτέλεσης όλων των παραπάνω διαδικασιών η πολυπλοκότητα του αλγόριθμου ΔΚΜΜ στην καλύτερη περίπτωση είναι $O(n)$.

a:0.13	b:0.14	c:0.23	d:0.24	e:0.26

Πίνακας 3.1.: Παράδειγμα καλύτερης περίπτωσης.

Στον Πίνακα 3.1, έχουμε ένα πίνακα που περιέχει $n = 5$ χαρακτήρες μαζί με τις συχνότητες εμφάνισης τους σε ένα κείμενο. Οι χαρακτήρες a, b είναι οι χαρακτήρες με την μικρότερη συχνότητα και ο χαρακτήρας e είναι αυτός με την μεγαλύτερη συχνότητα. Μετά την πρώτη συγχώνευση ισχύει ότι το άθροισμα των συχνοτήτων a, b είναι ίσο ή μεγαλύτερο από την συχνότητα του e. Το n είναι πλέον 4 και η σταθερά k επίσης 4. Η διαδικασία Διαίρεση μπορεί να συνεχίσει επιλύοντας το πρόβλημα σε γραμμικό χρόνο $O(n)$.

Χειρότερη περίπτωση

Η χειρότερη περίπτωση εκτέλεσης του αλγορίθμου ΔΚΜΜ προκύπτει όταν ισχύει $Pr(min1) + Pr(min2) < Pr(max)$ για $n - 3$ βήματα. Η διαδικασία Ομαδοποίηση εκτελείται για $n - 3$ βήματα, Η διαδικασία Διαίρεση για $n > k$ εκτελείται για 1 βήμα.

Η διαδικασία Διαίρεση για $n \leq k$ που απαιτεί τον μικρότερο χρόνο εκτέλεσης θα εκτελεστεί μόνο για ένα βήμα.

Έτσι μπορούμε να θεωρήσουμε ότι στην χειρότερη περίπτωση, η πολυπλοκότητα του αλγορίθμου ΔΚΜΜ είναι $O(nlgn)$.

a:0.14	b:0.15	c:0.17	d:0.18	e:0.36

Πίνακας 3.2.: Παράδειγμα χειρότερης περίπτωσης.

Στον Πίνακα 3.2, έχουμε την εκτέλεση της χειρότερης περίπτωσης όπου η Ομαδοποίηση θα εκτελεστεί για $n-3$ βήματα. Για $n=3$ έχουμε $k=2$ οπότε η Διαίρεση για $n \leq k$ θα εκτελεστεί μόνο μία φορά.

Μεταξύ καλύτερης και χειρότερης περίπτωσης

Η ενδιάμεση περίπτωση είναι όταν ή διαδικασία Ομαδοποίηση και η εκτέλεση της Διαίρεσης για $n > k$ εκτελούνται για $n/2$ βήματα όπου n ισούται με το αρχικό πλήθος των στοιχείων στην ουρά προτεραιότητας.

Η διαδικασία Ομαδοποίηση έχει κόστος εκτέλεσης για κάθε βήμα $O(lgn)$, της Διαίρεσης για $n > k$, $O(lgn)$ επίσης.

Η διαδικασία Διαίρεση για $n \leq k$ έχει γραμμικό χρόνο εκτέλεσης $O(n)$.

Η πολυπλοκότητα του αλγορίθμου σε αυτή την περίπτωση είναι μεταξύ του $O(n)$ και $O(nlgn)$.

Συνεπώς, μπορούμε να θεωρήσουμε ότι η πολυπλοκότητα του αλγορίθμου για την περίπτωση αυτή είναι $O(nlgn)$.

a:0.10	b:0.11	c:0.13	d:0.13	e:0.13	f:0.14	g:0.26

Πίνακας 3.3.: Παράδειγμα μεταξύ καλύτερης και χειρότερης περίπτωσης.

Στον Πίνακα 3.3, έχουμε ένα παράδειγμα μεταξύ καλύτερης και χειρότερης περίπτωσης εκτέλεσης του αλγορίθμου, όπου $n=7$. Η διαδικασία Ομαδοποίηση θα εκτελεστεί για 2 βήματα. Το μέρος της διαίρεσης που είναι ίδιο με αυτό της Ομαδοποίησης θα εκτελεστεί για 1 βήμα. Το n είναι πλέον ίσο με την σταθερά $k=4$ και έτσι η Διαίρεση επιλύει το πρόβλημα σε γραμμικό χρόνο $O(n)$.

3.3.2. Βέλτιστο Αποτέλεσμα

Σε αυτό το κεφάλαιο περιγράψαμε ένα αλγόριθμο για τη δημιουργία κώδικα μεταβλητού μήκους. Τώρα θα αποδείξουμε ότι ο αλγόριθμος ΔΚΜΜ παράγει πάντοτε βέλτιστο δυαδικό κώδικα μεταβλητού μήκους για οποιαδήποτε είσοδο.

Ο αλγόριθμος χωρίζεται σε δύο κύριες διαδικασίες. Η πρώτη, η Ομαδοποίηση, μπορεί να υλοποιηθεί στην πράξη με τον ίδιο ακριβώς τρόπο με τον αλγόριθμο του Huffman και έτσι είναι εύκολο να αποδείξουμε ότι παράγει το βέλτιστο δυνατό αποτέλεσμα για όσα βήματα εκτελεστεί (βλ. Κεφ. 2.3.2). Το ίδιο ισχύει και για την εκτέλεση της διαδικασίας Διαίρεση αν $n > k$. Για να αποδείξουμε τώρα ότι η διαδικασία Διαίρεση μας δίνει τα καλύτερα δυνατά αποτελέσματα θα αποδείξουμε ότι από την στιγμή που τερματίζει η Ομαδοποίηση χρειάζεται απλά να δημιουργήσουμε ένα πλήρες δυαδικό δένδρο.

Έστω x και y οι χαρακτήρες με την μικρότερη συχνότητα και m ο χαρακτήρας με την μεγαλύτερη συχνότητα. Μετά τον τερματισμό της Ομαδοποίησης ισχύει ότι το άθροισμα των συχνοτήτων των x, y είναι μεγαλύτερο ή ίσο του m.

$$\text{Αρχικά ισχύει η συνθήκη } f[x] + f[y] \geq f[m]$$

Εφόσον οι χαρακτήρες είναι ταξινομημένοι κατά αύξουσα σειρά ως προς την συχνότητα τότε, λόγο ταξινόμησης, η συχνότητα οποιουδήποτε άλλου χαρακτήρα $f[char]$ στο πρόβλημα είναι $(f[x] + f[y]) \geq f[char] \geq f[m]$. Επίσης οι χαρακτήρες x και y καταλαμβάνουν πάντοτε τις δύο πρώτες θέσεις της Q.

$$(f[x] + f[y]) \geq f[char] \geq f[m] = f[char] \geq f[m] \geq (f[x] + f[y]) \tag{3.2}$$

Σύμφωνα με την σχέση 3.2, η τοποθέτηση του αθροίσματος των δύο μικρότερων συχνοτήτων στο τέλος δεν αλλοιώνει την αρχική συνθήκη. Επαναλαμβάνοντας την διαδικασία για $n-1$ δεν αλλοιώνουμε πότε αυτή την συνθήκη, καθώς $f[char]$ ένα σύνολο χαρακτήρων που είναι ταξινομημένοι κατά αύξουσα σειρά ως προς την συχνότητα. Οι δύο μικρότερες συχνότητες καταλαμβάνουν πάντα τις δύο πρώτες θέσεις και ο χαρακτήρας με την μεγαλύτερη συχνότητα την τελευταία. Κατά συνέπεια το αποτέλεσμα παραμένει βέλτιστο.

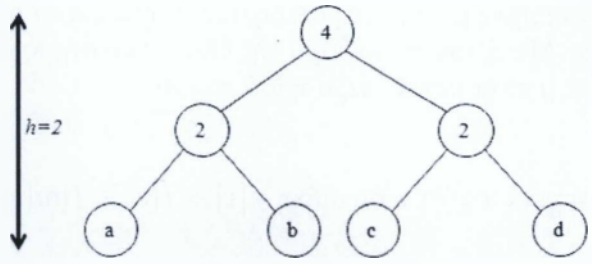
Επίσης αν αναπαραστήσουμε σχηματικά την διαδικασία αυτή θα προκύψει ένα πλήρες δυαδικό δένδρο.

Δυστυχώς, η ταξινόμηση των στοιχείων είναι μία διαδικασία που έχει επιπλέον κόστος σε σχέση την δημιουργία ενός σωρού ελαχίστου. Με την δημιουργία ενός σωρού ελαχίστου τα στοιχεία δεν ταξινομούνται. Για να καταφέρουμε να δημιουργήσουμε βέλτιστο κώδικα από μη ταξινομημένους χαρακτήρες πρέπει να γνωρίζουμε (βλ. παράρτημα Α'.1.) μερικά πράγματα για τα δυαδικά δένδρα.

Γνωρίζουμε ότι σε ένα πλήρες δυαδικό δένδρο, αν ο αριθμός των καταληκτικών κόμβων είναι κάποια δύναμη του 2 τότε σε αυτό το δένδρο όλοι οι καταληκτικοί κόμβοι βρίσκονται στο ίδιο επίπεδο. Αυτό σημαίνει ότι όποια θέση και αν επιλέξουμε στο δένδρο θα έχει το ίδιο βάρος. Άρα αν,

$$n = 2^x \text{ όπου } x \text{ακέραιος αριθμός } \geq 1 \text{ τότε ισχύει ότι } h = x \tag{3.3}$$

Σύμφωνα με την σχέση 3.3, αν το πλήθος n των καταληκτικών κόμβων είναι κάποια δύναμη του 2 τότε αυτή η δύναμη x είναι ίση με το ύψος h του δένδρου. Εφόσον το ύψος h του δένδρου είναι x για όλους τους καταληκτικούς κόμβους τότε η θέση τους στο δένδρο δεν έχει καμία σημασία.



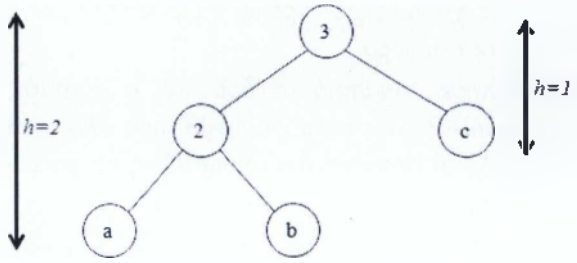
Σχήμα 3.4.: Δυαδικό δένδρο όπου n κάποια δύναμη του 2.

Στο Σχήμα 3.4, παρατηρούμε ένα δυαδικό δένδρο όπου το πλήθος των καταληκτικών του κόμβων είναι ίσο με μία δύναμη του 2. Άρα ισχύει ότι $n = 2^x$ όπου $n = 4$ και $x = 2$. Το ύψος του δένδρου είναι $h = x$. Όποια-και αν είναι η θέση των a, b, c, d στο δένδρο το αποτέλεσμα παραμένει βέλτιστο.

Το ίδιο μπορούμε να υποθέσουμε και για αν $n = 2^x - 1$ γιατί όταν τερματίζει η διαδικασία Ομαδοποίηση τοποθετεί τον νέο κόμβο z στο τέλος της Q.

$$z \geq max(Q) \tag{3.4}$$

Σύμφωνα με την σχέση 3.4, ο κόμβος z μπορεί να θεωρηθεί το μέγιστο στοιχείο της Q. Δεν πρέπει να ξεχνάμε ότι ο z είναι ένας κόμβος ο οποίος προκύπτει από την πρόσθεση των δύο μικρότερων συχνοτήτων. Άρα γνωρίζουμε ότι ο κόμβος με την μεγαλύτερη συχνότητα βρίσκεται στο τέλος της Q.



Σχήμα 3.5.: Δυαδικό δένδρο για $n = 2^x - 1$

Στο Σχήμα 3.5, παρατηρούμε ότι το ύψος h του δυαδικού δένδρου δεν είναι το ίδιο για όλους τους καταληκτικούς κόμβους. Για να έχουμε το βέλτιστο αποτέλεσμα πρέπει ο κόμβος c να είναι ο κόμβος με την μεγαλύτερη συχνότητα. Η Ομαδοποίηση, στο

τελευταίο βήμα της εκτέλεσής της, τοποθετεί τον κόμβο c στο τέλος της Q. Η διαίρεση στο πρώτο βήμα εξάγει τα δύο πρώτα a, b τα οποία και ορίζει ως παιδία του z. Τα μόνα στοιχεία που απέμειναν είναι ο z και ο c. Εκτελώντας το τελευταίο βήμα της Διαίρεσης διασφαλίζουμε ότι ο c θα καταλάβει την θέση με το μικρότερο ύψος.

Για οποιαδήποτε άλλη περίπτωση, δηλαδή αν $n \neq 2^x$ ή $n \neq 2^x - 1$ πρέπει να γνωρίζουμε την ακριβή θέση των στοιχείων με την μεγαλύτερη τιμή. Επειδή κάτι τέτοιο απαιτεί πολλές πράξεις, αυτό που κάνουμε είναι να χρησιμοποιούμε την ιδιότητα της διαδικασίας Ομαδοποίησης και να μετατρέπουμε οποιοδήποτε n σε $n = 2^x$ ή $n = 2 - 1$.

Όσον αφορά την διαδικασία του υπολογισμού της σταθεράς k αυτό κάνει είναι να βρίσκει την δύναμη x του 2 ώστε 2^x ο πιο κοντινός αριθμός στο n. Πρέπει όμως πάντα $k \leq n$. Για παράδειγμα για $n = 10$ τότε $k = 8$,όπου $8 = 2^3$ άρα $x = 3$, και όχι $k = 4$ ή $k = 16$. Αν $n = 16$ τότε $k = 16$,όπου $16 = 2^4$ άρα $x = 4$, και όχι $k = 8$ ή $k = 32$.

Μέχρι το n να αποκτήσει την τιμή του k χρησιμοποιούμε την σίγουρη Ομαδοποίηση και εφόσον το n αποκτήσει την τιμή του k τότε η Διαίρεση αναλαμβάνει να επιλύσει το πρόβλημα ακολουθώντας μία πολύ πιο γρήγορη στρατηγική.

3.4. Συμπεράσματα

Ο αλγόριθμος ΔΚΜΜ παράγει πάντοτε το βέλτιστο αποτέλεσμα σε 3 βήματα. Η διαδικασία Ομαδοποίηση είναι μία διαδικασία ισοδύναμη με αυτή του αλγορίθμου του Huffman. Το ίδιο ισχύει και για την Διαίρεση όπου το $n > k$. Η Διαίρεση για $n \leq k$ εκτελείται πολύ ταχύτερα από αυτή του αλγορίθμου του Huffman. Σε ένα βήμα του, ο αλγόριθμος του Huffman εκτελεί 3 πράξεις με κόστος $O(lgn)$ η καθεμία ενώ η Διαίρεση εκτελεί 3 πράξεις με κόστος $O(1)$.

Αυτό που χρειάζεται επιπλέον ο αλγόριθμος ΔΚΜΜ είναι η γνώση της αρχικής μέγιστης συχνότητας, και ο υπολογισμός της σταθεράς k. Ο υπολογισμός της σταθεράς k μπορεί να γίνει εύκολα και με πολύ μικρό υπολογιστικό κόστος επειδή χρησιμοποιεί πολλαπλασιασμούς και μία διαίρεση με το 2. Επίσης στην χειρότερη περίπτωση εκτέλεσης του ΔΚΚΜ, όπου ο αλγόριθμος γίνεται συγκρίσιμος με τον αντίστοιχο του Huffman, θα έχουμε μόλις μία πράξη πολλαπλασιασμού και μία διαίρεσης.

Τέλος, αυτό που έχει μεγάλη σημασία είναι ότι δεν μπορούμε να ισχυριστούμε ότι ο αλγόριθμος ΔΚΜΜ είναι πιο αργός ή πιο γρήγορος από αυτόν του Huffman, αν πρώτα δεν τους υλοποιήσουμε και τους συγκρίνουμε στην πράξη.

4. Υλοποίηση Αλγορίθμων στην Γλώσσα C

4.1. Εισαγωγή

Σε αυτό το κεφάλαιο θα υλοποιήσουμε τους αλγορίθμους που είδαμε στα προηγούμενα κεφάλαια στην γλώσσα προγραμματισμού C. Πρώτα θα υλοποιήσουμε τον αλγόριθμο του Huffman. Η υλοποίηση του θα γίνει με τέτοιο τρόπο, ώστε να μπορέσουμε να τον μετασχηματίσουμε εύκολα στον αλγόριθμο ΔΚΜΜ. Το πρόγραμμα που θα παρουσιάζει την κωδικοποίηση Huffman θα ονομάζεται *Huffman coding* (κωδικοποίηση Huffman). Αυτό που θα παρουσιάζει την κωδικοποίηση ΔΚΜΜ θα ονομάζεται *VLCC coding* (Variable Length Code Creator) που σημαίνει δημιουργός κώδικα μεταβλητού μήκους.

4.2. Κωδικοποίση Huffman

4.2.1. Το Πρόγραμμα

Το πρόγραμμά μας βασίζεται στην θεωρία του Κεφαλαίου 2. Θα ξεκινήσουμε μετρώντας την συχνότητα εμφάνισης των χαρακτήρων σε ένα κείμενο. Στην συνέχεια θα μετατρέψουμε σε κόμβους τον κάθε χαρακτήρα με την συχνότητα του. Έπειτα, θα εισάγουμε τους κόμβους σε μία ουρά προτεραιότητας Q, η οποία και θα υλοποιείται ως διπλά συνδεδεμένη λίστα. Οι κόμβοι στην λίστα μας θα είναι ταξινομημένοι κατά αύξουσα σειρά ως προς την συχνότητα τους. Ακολουθώντας τον αλγόριθμο του Huffman θα δημιουργούμε για $n-1$ βήματα ένα νέο κόμβο z, θα εξάγουμε τα δύο πρώτα στοιχεία της ουράς Q και θα τα ορίζουμε ως παιδία του z. Η συχνότητα του z θα προκύπτει από το άθροισμα των συχνοτήτων των παιδιών του. Ο νέος κόμβος θα εισάγεται ανάλογα με την συχνότητα του στην κατάλληλη θέση ώστε τα στοιχεία να διατηρούνται ταξινομημένα. Μέτα από $n-1$ βήματα θα έχουμε σχηματίσει το δένδρο μας και θα μπορούμε να βρούμε την κωδικοποίηση του κάθε χαρακτήρα.

Για να μπορεί ο καθένας να αντιλαμβάνεται τα στάδια της εκτέλεσης του προγράμματος θα χωρίσουμε σε πίνακες την κάθε λειτουργία του. Στην συνέχεια θα αναλύσουμε τον κάθε πίνακα.

Πίνακας 1

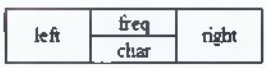
Το πρόγραμμα δέχεται ως είσοδο ένα αρχείο τύπου *.txt. Το πρόγραμμα έχει την δυνατότητα να διαβάσει οποιονδήποτε από τους 128 χαρακτήρες του κώδικα ASCII.

Ανοίγουμε λοιπόν το αρχείο κειμένου και μετράμε τους χαρακτήρες που αυτό περιέχει. Το αποτέλεσμα της μέτρησης θα εμφανίζεται στην οθόνη του χρήστη.

Στην συνέχεια ορίζουμε το πλήθος των ενεργών χαρακτήρων που εμφανίζονται στο κείμενο. Για παράδειγμα, η λέξη programm διαθέτει 6 διαφορετικούς χαρακτήρες.

Πίνακας 2

Ο κάθε χαρακτήρας με την συχνότητα του μετατρέπεται σε έναν κόμβο. Το νέο αντικείμενο ονομάζεται node και εμπεριέχει τον χαρακτήρα, την συχνότητα και δύο δείκτες *left και *right.



Σχήμα 4.1.: Σχηματική αναπαράσταση του κόμβου node.

Στον χρήστη εμφανίζεται ο αρχικός σωρός με τους κόμβους.

Πίνακας 3

Οι κόμβοι του αρχικού σωρού ταξινομούνται ως προς την συχνότητα. Για την ταξινόμηση των στοιχείων χρησιμοποιούμε την ταξινόμηση σωρού (HeapSort).

Πίνακας 4

Τα στοιχεία εισάγονται σε μία διπλά συνδεδεμένη λίστα. Το κάθε στοιχείο node αποτελεί πλέον μέλος ενός άλλου στοιχείου με το όνομα snode. Το snode εκτός από το node έχει και δύο δείκτες *next και *prev. Όλα τα στοιχεία snode συνδέονται μεταξύ τους με τους δείκτες *next και *prev .

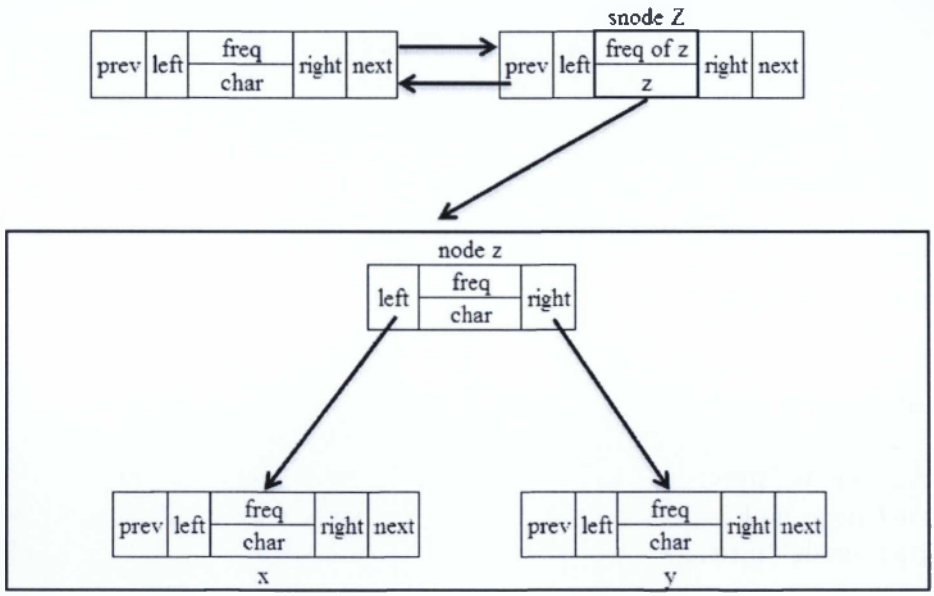
Σχήμα 4.2.: Σχηματική αναπαράσταση του snode.

Οι κόμβοι σχηματίζουν πλέον μία διπλά συνδεδεμένη λίστα.

Σχήμα 4.3.: Παράδειγμα διπλά συνδεδεμένης λίστας.

Πίνακας 5

Στον πίνακα αυτό έχουμε την εκτέλεση του αλγορίθμου του Huffman. Ο χρήστης μπορεί να παρακολουθεί το κάθε βήμα της εκτέλεσης του αλγορίθμου. Σε κάθε βήμα δημιουργείται ένας κόμβος z. Ο κόμβος z είναι ένας κόμβος node. Στην συνέχεια εξάγονται οι δύο snode με την μικρότερη συχνότητα από την λίστα. Ο πρώτος κόμβος που εξάγεται ονομάζεται x και ο δεύτερος y. Έπειτα, ο δείκτης *left του κόμβου z δείχνει στον κόμβο x και ό δείκτης *right στον κόμβο y. Η συχνότητα του z είναι το άθροισμα των συχνοτήτων των κόμβων x, y. Ο node z αλλάζει ξανά σε snode Z. Τέλος εισάγουμε τον νέο κόμβο snode Z στην κατάλληλη θέση στην λίστα.



Σχήμα 4.4.: Η δημιουργία του κόμβου z.

Η διαδικασία εκτελείται για n-1 βήματα. Στο τελευταίο βήμα δημιουργούμε ένα ακόμη snode ο οποίος ονομάζεται root και ο οποίος εμπεριέχει όλους κόμβους.

Πίνακας 6

Στον τελευταίο πίνακα υπολογίζουμε την κωδικοποίηση του κάθε χαρακτήρα. Ξεκινώντας από τον μόνο κόμβο snode root ακολουθούμε τους δείκτες *left και *right. Για τον δείκτη *left σημειώνουμε 0 και για τον δείκτη *right 1. Η διαδικασία ολοκληρώνεται όταν ο δείκτης *left ή ο δείκτης *right έχουν την τιμή κενό, NULL. Στον χρήστη εμφανίζεται ο πίνακας με τον χαρακτήρα, την συχνότητα, και την κωδικοποίηση του.

4.2.2. Κωδικας C για Huffman coding

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct A0
{
    int f;
    int c;
};

struct node
{
    struct node *left;
    int f;
    int c;
    struct node *right;
};

struct snode
{
    struct snode *prev;
    struct node node;
    struct snode *next;
};

struct A0 A0[128], HEAP[128];

int A[2][128];
char k;
int tablecounter=0;

int Patrikos (int i);
int Aristeros (int i);
int Deksios (int i);

int Ceil (int a);
void Swap (int *a, int *b);

void MinHeapfy (struct A0 A[ ], int i);
void BuildMinHeapfy (struct A0 A[ ]);

void COPYSTR (struct A0 *a, struct A0 *b);
```

```
void SWAPSTR (struct A0 *a, struct A0 *b);
void DisplayHEAP ();
void HeapSort (struct A0 A[ ]);
void CodeInit ();

int A1[10] = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1};
int A2[10] = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7};

int N;
int N1;

struct snode *GetSnode ();
void CREATELIST ();
void DisplayLIST ();
void DisplayLIST2 ();

struct node *GetNext ();
struct node *GetNode ();

void DisplayKAD (struct node *root,char *copybincode);

void HUFFMAN ();

int InsNode (struct node *z);

struct snode *first, *current, *tmp, *root;

main ()
{
    int i;

    printf("\n\nHuffman coding\n");
    printf("\nPress Enter to start\n");
    getchar();

    printf("_____\n");
    printf("Table 1. Text file contents and relative frequencies\n");
    printf("_____\n");
    CodeInit ();
    printf(_____\n\n\n\n");

    getchar();

    N=128-tablecounter;
    N1=N;

    printf("Number of effective elements %d.\n\n",N);
```

```
getchar();

printf("_____\n");
printf("Table 2. Initial Heap -a node created for each character
and frequency-\n");
printf("_____\n");
DisplayHEAP ();
printf("_____\n\n\n\n");

getchar();

HeapSort (HEAP);
printf("_____\n");
printf("Table 3. Elements sorted by frequency\n");
printf("_____\n");
DisplayHEAP ();
printf("_____\n\n\n\n");

getchar();

CREATELIST ();
printf("_____\n");
printf("Table 4. Elements inserted into linked list\n");
printf("_____\n");
DisplayLIST ();
printf("_____\n\n\n\n");

getchar();

printf("_____\n");
printf("Table 5. Huffman coding process\n");
printf("_____\n");
DisplayLIST ();
getchar();
HUFFMAN();
DisplayLIST ();
printf("%3d \n %3c \n", root->node.f, root->node.c);
printf("_____\n\n\n\n");

getchar();
char copybincode[80]="";

printf("_____\n");
printf("Table 6. Huffman coding table\n");
printf("_____\n\n");
```

```
    printf("Symbol Frequency Code\n");
    DisplayKAD (&root->node,copybincode);
    printf("_____\n\n\n\n");
}

void DisplayKAD (struct node *root, char *copybincode)
{
    char bincode[2]="1";

    if (root->right)
    {
        strcat(copybincode,bincode);
        DisplayKAD(root->left,copybincode);
        copybincode[strlen(copybincode)-1]=0;
    }

    if (root->left)
    {
        bincode[0]='0';
        strcat(copybincode,bincode);
        DisplayKAD(root->right,copybincode);
        copybincode[strlen(copybincode)-1]=0;
    }

    else if (!root->left || !root->right)
        printf("%3c %13d %s\n",root->c,root->f,copybincode);
}

void HUFFMAN ()
{
    struct node *z, *x, *y;

    while (1)
    {
        z = GetNode ();
        x = GetNext ();

        if (x == NULL)
        {
            printf ("Error -> GetNext->x\n");
            exit (0);
        }

        y = GetNext ();

        if (y == NULL)
```

```
            {
                printf ("Error -> GetNext->y\n");
                exit (0);
            }

            z->f = x->f + y->f;
            z->c = 'z';

            z->left = x;
            z->right = y;

            if (first == NULL)
            {
                root = GetSnode ();
                root->prev = NULL;
                root->next = NULL;
                root->node.f = z->f;
                root->node.c = z->c;
                root->node.left = z->left;
                root->node.right = z->right;
                break;
            }

            else
                InsNode (z);

    DisplayLIST ();

        getchar();
        }
    }

    int InsNode (struct node *z)
    {

        struct snode *tmp, *t, *Z, *cur;

        Z = GetSnode ();
        Z->node.left = z->left;
        Z->node.f = z->f;
        Z->node.c = z->c;
        Z->node.right = z->right;

        for (tmp = first; tmp != NULL; tmp = tmp->next)
        {
```

```
            if (tmp->node.f >= Z->node.f)
                break;
            cur = tmp;
        }

        if (tmp == NULL)
        {
            cur->next = Z;
            Z->prev = cur;
        }

        else if (first == tmp)
        {
            tmp->prev = Z;
            Z->next = tmp;
            first = Z;
        }

        else
        {
            t = tmp->prev;
            Z->prev = t;
            Z->next = tmp;
            t->next = Z;
            tmp->prev = Z;
        }

        return (1);

}

struct node *GetNode ()
{

        struct node *tmp;

        tmp = (struct node *) malloc (sizeof (struct node));
        tmp->left = NULL;
        tmp->right = NULL;
        return (tmp);
}

struct node *GetNext ()
{
        struct snode *tmp;
```

```
    if (first != NULL)
    {
        tmp = first;
        first = first->next;
        return (&tmp->node);
    }
    return (NULL);
}

void DisplayLIST ()
{

    for (tmp = first; tmp != NULL; tmp = tmp->next)
    {
        printf ("%3d ", tmp->node.f);
    }

    printf ("\n");

    for (tmp = first; tmp != NULL; tmp = tmp->next)
    {
        printf ("%3c ", tmp->node.c);
    }

    printf ("\n");
}

void DisplayLIST2 ()
{

    for (tmp = current; tmp != NULL; tmp = tmp->prev)
    {
        printf ("%3d ", tmp->node.f);
    }

    printf ("\n");

    for (tmp = current; tmp != NULL; tmp = tmp->prev)
    {
        printf ("%3c ", tmp->node.c);
    }

    printf ("\n");
}

void CREATELIST ()
```

```
{
    int i;

    for (i = 0; i < N1; i++)
    {
        tmp = GetSnode ();
        tmp->prev = NULL;
        tmp->node.left = NULL;
        tmp->node.f = HEAP[i].f;
        tmp->node.c = HEAP[i].c;
        tmp->node.right = NULL;
        tmp->next = NULL;
        if (first == NULL)
        {
            first = tmp;
            current = first;
        }

        else
        {
            current->next = tmp;
            tmp->prev = current;
            current = tmp;
        }
    }
}

struct snode *GetSnode()
{
    tmp = (struct snode *) malloc (sizeof (struct snode));
    return (tmp);
}

void HeapSort (struct A0 A[ ])
{
    int i;

    BuildMinHeapfy (A);

    for (i = N - 1; i >= 1; i- -)
    {
        SWAPSTR (&A[0], &A[i]);
        N = N -1;
        MinHeapfy (A, 1);
    }
```

```
}

void DisplayHEAP ()
{
    int i;

    for (i = 0; i < N1; i++)
        printf ("%3d ", HEAP[i].f); printf ("\n");

    for (i = 0; i < N1; i++)
        printf ("%3c ", HEAP[i].c);

    printf ("\n");
}

void BuildMinHeapfy (struct A0 A[ ])
{
    int i;

    for (i = Ceil(N)-1; i>=0; i- -)
        MinHeapfy (A, i+1);
}

void MinHeapfy (struct A0 A[ ], int i)
{
    int l, r, posmax;

    l = Aristeros (i);
    r = Deksios (i);
    i = i - 1;

    if (l < N && A[l].f > A[i].f)
        posmax = l;

    else
        posmax = i;

    if (r < N && A[r].f > A[posmax].f)
        posmax = r;

    if (posmax != i)
    {
        SWAPSTR (&A[i], &A[posmax]);
        MinHeapfy (A, posmax+1);
    }

}
```

```
void Swap (int *a, int *b)
{
    int k;

    k = *a;
    *a = *b;
    *b = k;
}

int Ceil (int a)
{
    return ((a+1)/2);
}

int Patrikos (int i)
{
    return ((i/2)-1);
}

int Aristeros (int i)
{
    return ((2 * i)-1);
}

int Deksios (int i)
{
    return ((2 * i));
}

void CodeInit ()
{
    int i, j;

    FILE *fp;

    fp = fopen ("TEXT", 'r');

    if (fp == NULL)
    {
        printf ('Error file %s \n', 'TEXT');
        exit (0);
    }

    for (i = 0; i < 128; i++)
        A0[i].c = i;
```

```
        while (!feof (fp))
        {
            fscanf (fp, "%c", &k);
            A0[k - '\0' ].f++;
            printf("%c",k);
        }

        fclose (fp);

        for (j = i = 0; i < 128; i++)
        {

            if (A0[i].f != 0 && A0[i].c != '\n')
            {
                printf ("%c - %d\n", A0[i].c, A0[i].f);
                COPYSTR (&HEAP[j], &A0[i]);
                j++;
            }

            else
                tablecounter++;

        }

    }

    void COPYSTR (struct A0 *a, struct A0 *b)
    {
        a->f = b->f;
        a->c = b->c;
    }

    void SWAPSTR (struct A0 *a, struct A0 *b)
    {
        struct A0 tmp;

        tmp.f = a->f;
        tmp.c = a->c;

        a->f = b->f;
        a->c = b->c;

        b->f = tmp.f;
        b->c = tmp.c;
    }
```

4.3. Κωδικοποίηση VLCC

4.3.1. Το Πρόγραμμα

Για να υλοποιήσουμε τον αλγόριθμο ΔΚΜΜ θα βασιστούμε σε αυτά που έχουμε ήδη κάνει για τον αλγόριθμο του Huffman. Από την στιγμή που ταξινομούμε τα στοιχεία δεν χρειάζεται να υπολογίσουμε τον συντελεστή ταξινόμησης k. Η εκτέλεση του Huffman μπορεί να θεωρηθεί ισοδύναμη με αυτή της Ομαδοποίησης, οπότε το μόνο που έχουμε να κάνουμε είναι να υλοποιήσουμε την διαδικασία Διαίρεση.

Το πρόγραμμα αυτό χρησιμοποιεί επίσης Πίνακες κατά την εκτέλεση του για την εύκολη κατανόηση της λειτουργίας του από τον χρήστη. Ο Πίνακας 1, Πίνακας 2,Πίνακας 3, Πίνακας 4, Πίνακας 6 εκτελούνται το ίδιο ακριβώς με την κωδικοποίηση Huffman. Το μόνο που αλλάζει είναι η διαδικασία του Πίνακα 5. Όπως αναφέραμε στο Κεφ. 3 αν τα στοιχεία είναι ταξινομημένα, ο αλγόριθμος ΔΚΜΜ χωρίζεται σε δύο διαδικασίες. Η πρώτη, η Ομαδοποίηση, μπορεί να υλοποιηθεί το ίδιο με τον αλγόριθμο του Huffman.

Στην αρχή κρατάμε την τιμή του μεγαλύτερου κόμβου στην λίστα μας. Ο τελευταίος κόμβος στην λίστα μας, είναι αυτός με την μεγαλύτερη συχνότητα. Συνεχίζουμε δημιουργώντας ένα κόμβο z παιδία του οποίου ορίζονται οι δύο κόμβοι με την μικρότερη συχνότητα. Η συχνότητα του z είναι το άθροισμα των συχνοτήτων των παιδιών του. Αν η συχνότητα του z είναι μικρότερη από την συχνότητα του τελευταίου κόμβου τότε εισάγουμε τον κόμβο z στην κατάλληλη θέση της Q ώστε να διατηρείται η ταξινόμηση. Η διαδικασία που περιγράψαμε μέχρι τώρα είναι ακριβώς η ίδια με αυτή της κωδικοποίησης Huffman.

Τώρα, η Διαίρεση ξεκινά αν, κάποια στιγμή ή εξ' αρχής, η συχνότητα του z είναι ίση ή μεγαλύτερη από αυτή του τελευταίου στοιχείου. Το πρόγραμμα εμφανίζει το μήνυμα *Normalization completed* (Η κανονικοποίηση ολοκληρώθηκε) και τότε ο κόμβος z τοποθετείται απ' ευθείας στο τέλος της λίστας μας. Ο όρος κανονικοποίηση και Ομαδοποίηση χρησιμοποιούνται για να περιγράψουν το ίδιο ακριβώς πράγμα. Αυτή είναι και η μόνη διαφορά σε σχέση με την υλοποίηση του Huffman.

4.3.2. Κώδικας C για VLCC coding

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct A0
{
    int f;
    int c;
};

struct node
```

```
{
    struct node *left;
    int f;
    int c;
    struct node *right;
};

struct snode
{
    struct snode *prev;
    struct node node;
    struct snode *next;
};

struct A0 A0[128], HEAP[128];

int A[2][128];
char k;
int tablecounter=0;

int Patrikos (int i);
int Aristeros (int i);
int Deksios (int i);

int Ceil (int a);
void Swap (int *a, int *b);

void MinHeapfy (struct A0 A[ ], int i);
void BuildMinHeapfy (struct A0 A[ ]);

void COPYSTR (struct A0 *a, struct A0 *b);
void SWAPSTR (struct A0 *a, struct A0 *b);
void DisplayHEAP ();

void HeapSort (struct A0 A[ ]);
void CodeInit ();

int A1[10] = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1};
int A2[10] = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7};

int N;
int N1;

struct snode *GetSnode ();

void CREATELIST ();
```

```
void DisplayLIST ();
void DisplayLIST2 ();

struct node *GetNext ();
struct node *GetNode ();

void DisplayKAD (struct node *root,char *copybincode);

void VLCC ();

int InsNode (struct node *z);
int InsNodedkmm (struct node *z);

struct snode *first, *current, *tmp, *root;

main ()
{
    int i;

    printf("\n\nVLCC coding\n");
    printf("\nPress Enter to start\n");
    getchar();

    printf("_____\n");
    printf("Table 1. Text file contents and relative frequencies\n");
    printf("_____\n");
    CodeInit ();
    printf(_____\n\n\n\n");

    getchar();

    N=128-tablecounter;
    N1=N;

    printf("Number of effective elements %d.\n\n",N);
    getchar();

    printf("_____\n");
    printf("Table 2. Initial Heap -a node created for each character
    and frequency-\n");
    printf("_____\n");
    DisplayHEAP ();
    printf("_____\n\n\n\n");

    getchar();

    HeapSort (HEAP);
```

```
    printf("_____\n");
    printf("Table 3. Elements sorted by frequency\n");
    printf("_____\n");
    DisplayHEAP ();
    printf("_____\n\n\n\n");

    getchar();

    CREATELIST ();
    printf("_____\n");
    printf("Table 4. Elements inserted into linked list\n");
    printf("_____\n");
    DisplayLIST ();
    printf("_____\n\n\n\n");

    getchar();

    printf("Symbol '%c' has the hightest frequency '%d'.\n\n",
    current->node.c, current->node.f);

    printf("Normalize frequencies until nodes x+y >= of the
    node with hightest frequency.\n\n");

    printf("_____\n");
    printf("Table 5. VLCC coding process\n");
    printf("_____\n");
    DisplayLIST ();
    getchar();
    VLCC();
    DisplayLIST ();
    printf("%3d \n %3c \n", root->node.f, root->node.c);
    printf("_____\n\n\n\n");

    getchar();
    char copybincode[80]="";

    printf("_____\n");
    printf("Table 6. VLCC coding table\n");
    printf("_____\n\n");
    printf("Symbol Frequency Code\n");
    DisplayKAD (&root->node,copybincode);
    printf("_____\n\n\n\n");
}

void DisplayKAD (struct node *root, char *copybincode)
```

```
{
    char bincode[2]="1";

    if (root->right)
    {
        strcat(copybincode,bincode);
        DisplayKAD(root->left,copybincode);
        copybincode[strlen(copybincode)-1]=0;
    }

    if (root->left)
    {
        bincode[0]='0';
        strcat(copybincode,bincode);
        DisplayKAD(root->right,copybincode);
        copybincode[strlen(copybincode)-1]=0;
    }

    else if (!root->left || !root->right)
        printf("%3c %13d %s\n",root->c,root->f,copybincode);
}

void VLCC ()
{
    struct node *z, *x, *y;

    while (1)
    {

        z = GetNode ();
        x = GetNext ();

        if (x == NULL)
        {
            printf ("Error -> GetNext->x\n");
            exit (0);
        }

        y = GetNext ();

        if (y == NULL)
        {
            printf ("Error -> GetNext->y\n");
            exit (0);
        }
```

```
z->f = x->f + y->f;

z->c = 'z';

z->left = x;
z->right = y;

if (z->f >= current->node.f)
{

    printf("normalization completed\n\n");

    if (first == NULL)
    {
        root = GetSnode ();
        root->prev = NULL;
        root->next = NULL;
        root->node.f = z->f;
        root->node.c = z->c;
        root->node.left = z->left;
        root->node.right = z->right;
        break;
    }

    else
        InsNodedkmm (z);
        DisplayLIST ();

}

else
{

    if (first == NULL)
    {
        root = GetSnode ();
        root->prev = NULL;
        root->next = NULL;
        root->node.f = z->f;
        root->node.c = z->c;
        root->node.left = z->left;
        root->node.right = z->right;
        break;
    }

    else
```

```
                    InsNode (z);
                    DisplayLIST ();

    }

getchar();
    }
}

int InsNode (struct node *z)
{

    struct snode *tmp, *t, *Z, *cur;

    Z = GetSnode ();
    Z->node.left = z->left;
    Z->node.f = z->f;
    Z->node.c = z->c;
    Z->node.right = z->right;

    for (tmp = first; tmp != NULL; tmp = tmp->next)
    {

        if (tmp->node.f >= Z->node.f)
            break;
        cur = tmp;
    }

    if (tmp == NULL)
    {
        cur->next = Z;
        Z->prev = cur;
    }

    else if (first == tmp)
    {
        tmp->prev = Z;
        Z->next = tmp;
        first = Z;
    }

    else
    {
        t = tmp->prev;
        Z->prev = t;
        Z->next = tmp;
```

```
            t->next = Z;
            tmp->prev = Z;
        }

        return (1);

    }

    int InsNodedkmm (struct node *z)
    {
        struct snode *tmp, *t, *Z, *cur;

        Z = GetSnode ();

        Z->node.left = z->left;
        Z->node.f = z->f;
        Z->node.c = z->c;
        Z->node.right = z->right;

        tmp=Z;

        if (tmp != NULL)
        {
            current->next=Z;
            Z->prev=current;
            Z->next=NULL;
            current=Z;
        }

        else
        {
            InsNode(z);
        }

        return (1);
    }

    struct node *GetNode ()
    {

        struct node *tmp;

        tmp = (struct node *) malloc (sizeof (struct node));
        tmp->left = NULL;
        tmp->right = NULL;

        return (tmp);
```

```
}

struct node *GetNext ()
{
    struct snode *tmp;

    if (first != NULL)
    {
        tmp = first;
        first = first->next;
        return (&tmp->node);
    }

    return (NULL);
}

void DisplayLIST ()
{

    for (tmp = first; tmp != NULL; tmp = tmp->next)
    {
        printf ("%3d ", tmp->node.f);
    }

    printf ("\n");

    for (tmp = first; tmp != NULL; tmp = tmp->next)
    {
        printf ("%3c ", tmp->node.c);
    }

    printf ("\n");
}

void DisplayLIST2 ()
{

    for (tmp = current; tmp != NULL; tmp = tmp->prev)
    {
        printf ("%3d ", tmp->node.f);
    }

    printf ("\n");

    for (tmp = current; tmp != NULL; tmp = tmp->prev)
    {
```

```
            printf ("%3c ", tmp->node.c);
        }

        printf ("\n");
    }

    void CREATELIST ()
    {

        int i;

        for (i = 0; i < N1; i++)
        {
            tmp = GetSnode ();

            tmp->prev = NULL;
            tmp->node.left = NULL;
            tmp->node.f = HEAP[i].f;
            tmp->node.c = HEAP[i].c;
            tmp->node.right = NULL;
            tmp->next = NULL;

            if (first == NULL)
            {
                first = tmp;
                current = first;
            }

            else
            {
                current->next = tmp;
                tmp->prev = current;
                current = tmp;
            }

        }
    }

    struct snode *GetSnode()
    {
        tmp = (struct snode *) malloc (sizeof (struct snode));
        return (tmp);
    }

    void HeapSort (struct A0 A[ ])
    {
```

```
    int i;

    BuildMinHeapfy (A);

    for (i = N - 1; i >= 1; i- -)
    {
        SWAPSTR (&A[0], &A[i]);
        N = N -1;
        MinHeapfy (A, 1);
    }

}

void DisplayHEAP ()
{
    int i;

    for (i = 0; i < N1; i++)
        printf ("%3d ", HEAP[i].f); printf ("\n");

    for (i = 0; i < N1; i++)
        printf ("%3c ", HEAP[i].c);

    printf ("\n");
}

void BuildMinHeapfy (struct A0 A[ ])
{
    int i;

    for (i = Ceil(N)-1; i>=0; i- -)
        MinHeapfy (A, i+1);
}

void MinHeapfy (struct A0 A[ ], int i)
{
    int l, r, posmax;

    l = Aristeros (i);
    r = Deksios (i);
    i = i - 1;

    if (l < N && A[l].f > A[i].f)
        posmax = l;

    else
```

```
            posmax = i;

        if (r < N && A[r].f > A[posmax].f)
            posmax = r;

        if (posmax != i)
        {
            SWAPSTR (&A[i], &A[posmax]);
            MinHeapfy (A, posmax+1);
        }

}

void Swap (int *a, int *b)
{
    int k;

    k = *a;
    *a = *b;
    *b = k;
}

int Ceil (int a)
{
    return ((a+1)/2);
}

int Patrikos (int i)
{
    return ((i/2)-1);
}

int Aristeros (int i)
{
    return ((2 * i)-1);
}

int Deksios (int i)
{
    return ((2 * i));
}

void CodeInit ()
```

```
{
    int i, j;
    FILE *fp;

    fp = fopen ("TEXT", 'r");

    if (fp == NULL)
    {
        printf ('Error file %s \n', "TEXT");
        exit (0);
    }

    for (i = 0; i < 128; i++)
        A0[i].c = i;

    while (!feof (fp))
    {
        fscanf (fp, "%c", &k);
        A0[k - '\0' ].f++;
        printf("%c",k);
    }

    fclose (fp);

    for (j = i = 0; i < 128; i++)
    {

        if (A0[i].f != 0 && A0[i].c != '\n')
        {
            printf ("%c - %d\n", A0[i].c, A0[i].f);
            COPYSTR (&HEAP[j], &A0[i]);
            j++;
        }

        else
            tablecounter++;

    }

}


void COPYSTR (struct A0 *a, struct A0 *b)
{
    a->f = b->f;
```

```
    a->c = b->c;
}

void SWAPSTR (struct A0 *a, struct A0 *b)
{
    struct A0 tmp;

    tmp.f = a->f;
    tmp.c = a->c;

    a->f = b->f;
    a->c = b->c;

    b->f = tmp.f;
    b->c = tmp.c;
}
```

Α΄. Παράρτημα - Επιπλέον Υλικό

Α΄.1. Ανάλυση Δυαδικών Δένδρων

Τα δυαδικά δένδρα που θα μελετήσουμε εδώ έχουν συγκεκριμένες ιδιότητες. Αυτό σημαίνει ότι αναφερόμαστε σε ένα πλήρες δυαδικό δένδρο όπου το πλήθος n συμβολίζει το σύνολο των καταληκτικών κόμβων. Έτσι, από το πλήθος n των κόμβων θα είμαστε σε θέση να υπολογίσουμε, με την χρήση απλών μαθηματικών, αν το δένδρο που σχηματίζουν αποτελείται από ένα καταληκτικό επίπεδο ή από δύο. Επίσης, θα είμαστε σε θέση να υπολογίσουμε και τον αριθμό τον καταληκτικών κόμβων του κάθε επιπέδου.

Έτσι για n που είναι κάποια δύναμη του 2 θα έχουμε ένα καταληκτικό επίπεδο στο δένδρο. Ισχύει δηλαδή ότι $n = 2^x$ και το ύψος h του δένδρου είναι x.

Για παράδειγμα αν $n = 32$, τότε $x = 5$ και $h = x$=5.

Αν δημιουργήσω ένα πλήρες δυαδικό δένδρο για $n = 32$ τότε το δένδρο αυτό θα διαθέτει ένα καταληκτικό επίπεδο και το ύψος του θα είναι ίσο με 5. Άρα οποιοσδήποτε καταληκτικός κόμβος απέχει 5 βήματα από την ρίζα του δένδρου.

Τώρα, για n που δεν είναι κάποια δύναμη του δύο έχουμε δύο επίπεδα στο δένδρο μας. Στην αρχή υπολογίζουμε τον συντελεστή k ο οποίος βρίσκει την δύναμη του 2 που αναπαριστά καλύτερα το n. Πρέπει επίσης να ισχύει πάντα $k \leq n$. Εφόσον βρούμε αυτόν τον αριθμό, τότε τον αφαιρούμε από το n. Πολλαπλασιάζοντας το αποτέλεσμα με το 2 προκύπτει ένας αριθμός l ο οποίος μας περιγράφει τον αριθμό τον καταληκτικών κόμβων στο τελευταίο επίπεδο. Τώρα η διαφορά του n από το l μας δίνει τον αριθμό τον καταληκτικών κόμβων f στο προηγούμενο επίπεδο. Οπότε,

$$l = (n - k) * 2 \text{ και } f = n - l$$

Για παράδειγμα αν $n = 10$, τότε $x = 3$ και $k = 8$. Οπότε, το τελευταίο επίπεδο θα έχει $l = (10 - 8) * 2 = 4$ κόμβους και το προτελευταίο επίπεδο θα έχει $f = 10 - 4 = 6$ κόμβους.

Α΄.2. Υπολογισμός Συντελεστή Ταξινόμησης

Στο παράρτημα αυτό υλοποιούμε τον αλγόριθμο για τον υπολογισμό του συντελεστή ταξινόμησης k στην γλώσσα C. Το παρακάτω πρόγραμμα δέχεται σαν είσοδο ένα ακέραιο αριθμό, που συμβολίζει το πλήθος των στοιχείων της Q. Στην συνέχεια υπολογίζει τον συντελεστή ταξινόμησης k καθώς και το πόσες φορές θα εκτελεστεί η διαδικασία Ομαδοποίηση (στο πρόγραμμα αναφέρεται ως Normalization).

Όταν στο πρόγραμμα μας αναφέρεται η εκτέλεση του VLCC, αυτό σημαίνει ότι θα έχουμε την εκτέλεση μόνο της διαδικασίας Διαίρεση.

Επίσης, στο πρόγραμμα, για τον συντελεστή k χρησιμοποιούμε την μεταβλητή j και για το n την μεταβλητή k.

Η υλοποίηση του έγινε σύμφωνα με την θεωρία του Κεφ. 3.2.3 και του παραρτήματος Α'.1.

```
#include <stdio.h>
#include <math.h>

main ()
{

    int k=0, j=2, f=0, l=0, k1=0;

    printf('Enter number of Q elements: ');
    scanf('%d',&k);

    printf('\n\n\n');
    k1=k;

    if (k==0 || k==1 || k==2)
    {
        printf('Invalid number of Q elements...\n\n');
        printf('Q must have at least 3 elements.\n\n\n\n\n\n');
        return -1;
    }

    else
    {
        while (j<=k)
        {
            j=j<<1;
        }
        j=j>>1;
        f=(k-j)<<1;
        l=k-f;
    }

    if (l==1)
    {
        printf('You needn't to normalize elements.\n\n');
```

```
            printf('Element <max> must swap with the last
            element of the heap.\n\n');

            printf('VLCC will execute for %d-1 = %d.\n\n\n
            \n\n\n',k,k-1);

            return -1;
        }

        if (l!=k1)
        {
            printf('Tree has two levels with %d child nodes.\n\n',k1);
            printf('First level of the tree has %d elements.\n',l);
            printf('Second level of the tree has %d elements.\n\n',f);
            printf('Normalization is necessary for %d times.\n',f>>1);

            printf('VLCC will execute for %d-1 = %d.\n\n\n\n\n\n',
            k-(f>>1),(k-(f>>1))-1);

        }

        else
        {
            printf('Tree has one level with %d child nodes.\n\n',l);
            printf('You needn't to normalize elements.\n\n');
            printf('VLCC will execute for %d-1 = %d.\n\n\n\n\n\n',l,l-1);
        }

}
```

Βιβλιογραφία

[1] T. H Cormen, C.E Leiserson, R. L. Rivest, C. Stein. *Εισαγωγή στους Αλγορίθμους Τόμος Ι*. Εκδόσεις Κρήτης, 2006.

[2] T.H Cormen, C.E Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, 2nd Edition. MIT Press, 2001.

[3] S. Dasgupta, C. H Papadimitriou, U. V Vazirani. *Αλγόριθμοι*. Κλειδάριθμος, 2009.

[4] C. E. Shannon. *A mathematical theory of communication*. Bell Sys. Tech. Jour., vol. 27, pp. 398-403, 1948.

[5] R. M. Fano. *The Transmission of information*. Technical Report No. 65, Research Laboratory of Electronics, M.I.T, 1949.

[6] D. A. Huffman. *A method for the Construction of Minimum - Redundancy Codes*. Proceedings of the I.R.E, 40(9):1098-1101, 1952.

[7] H. Schildt. *Οδηγός της C*. Μ. Γκιούρδας, 2006.

[8] B. W. Kernighan, D. E Ritchie. *Η γλώσσα προγραμματισμού C*. Κλειδάριθμος, 2008.

[9] D. E. Knuth. *The art of Computer Programming, Vol. 1, Fundamental Algorithms*. Addison-Wesley,1973.

[10] D. E. Knuth. *Optimum binary search trees*. Acta Informatica, 1:14 - 25, 1971.

[11] Φ. Αφράτη, Γ. Παπαγεωργίου. *Αλγόριθμοι: Μέθοδοι Σχεδίασης και Ανάλυση Πολυπλοκότητας*. Συμμετρία, 1993.

[12] Χ. Κοίλιας. *Δομές Δεδομένων και Οργανώσεις Αρχείων*. Εκδόσεις Νέων Τεχνολογιών, 2004.