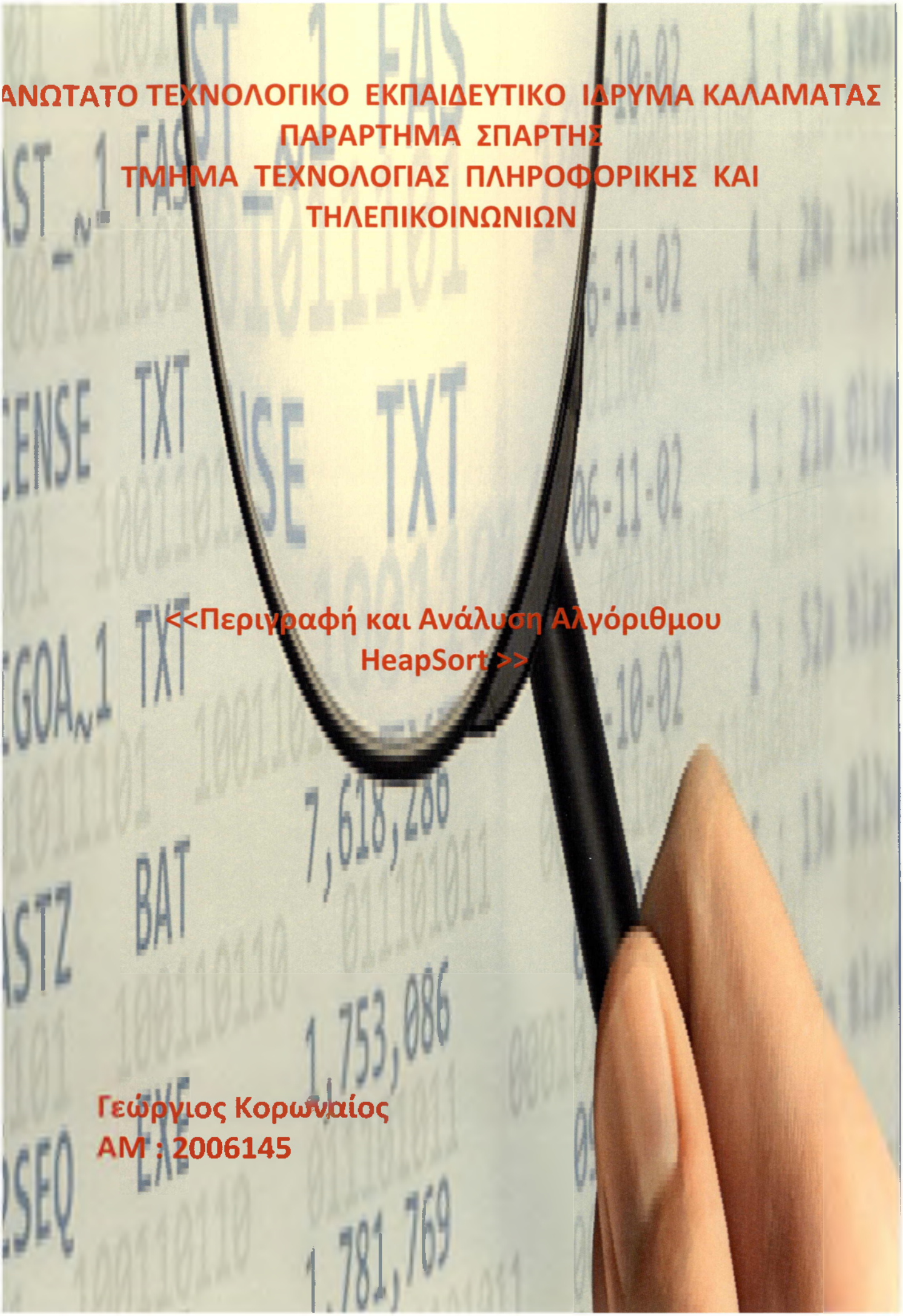


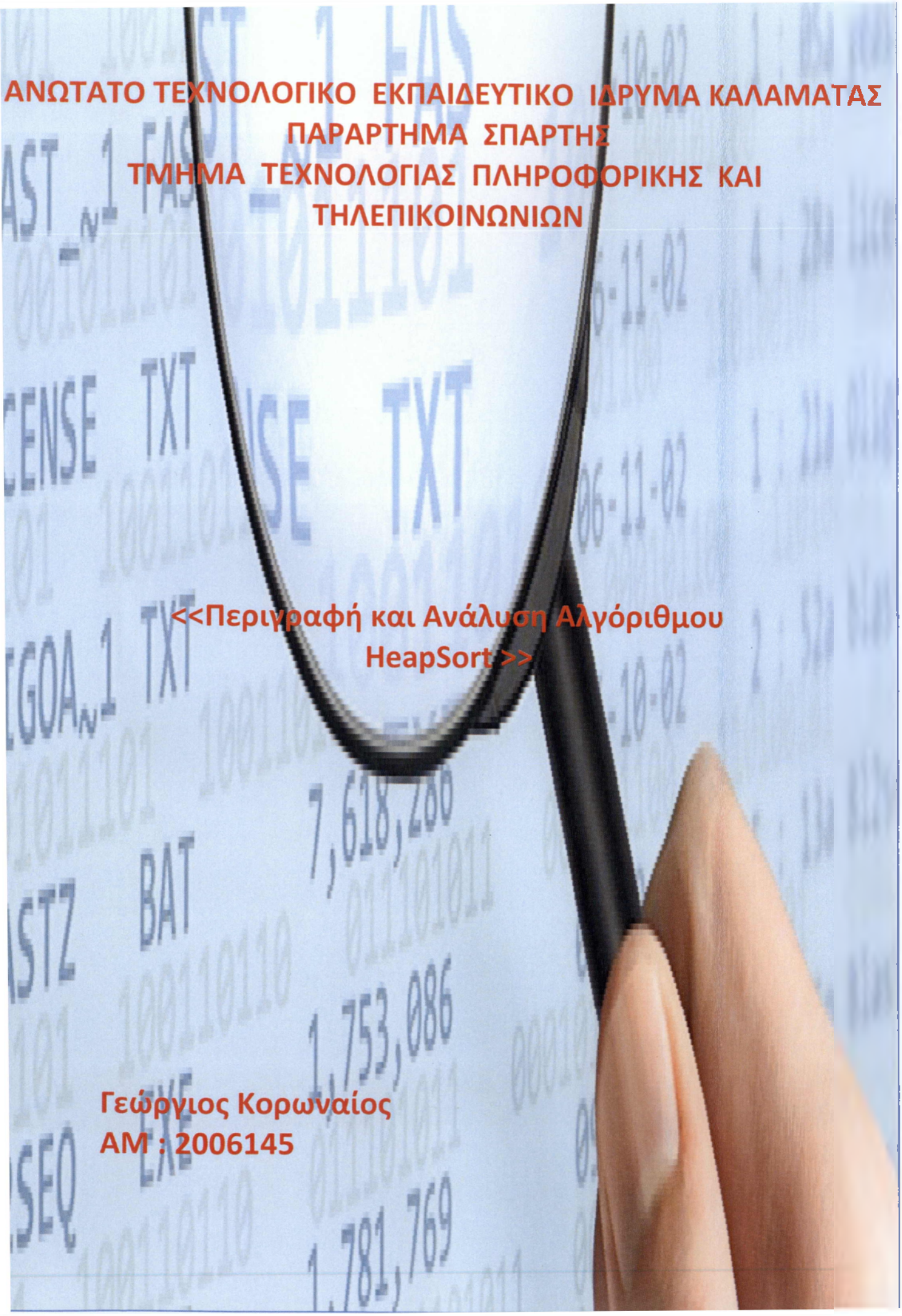
ΑΝΩΤΑΤΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΑΛΑΜΑΤΑΣ
ΠΑΡΑΡΤΗΜΑ ΣΠΑΡΤΗΣ
ΤΜΗΜΑ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

<<Περιγραφή και Ανάλυση Αλγόριθμου HeapSort >>

Γεώργιος Κορωναίος
ΑΜ : 2006145

ΑΝΩΤΑΤΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΚΑΛΑΜΑΤΑΣ
ΠΑΡΑΡΤΗΜΑ ΣΠΑΡΤΗΣ
ΤΜΗΜΑ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

<<Περιγραφή και Ανάλυση Αλγόριθμου HeapSort >>



Γεώργιος Κορωναίος
ΑΜ : 2006145

Περιγραφή και Ανάλυση Αλγόριθμου HeapSort

Πτυχιακή Εργασία: Γεώργιος Κορωναίος
ΑΜ : 2006145
Επιβλέπων Καθηγητής: Καραγιώργος Γρηγόρης

ΑΤΕΙ Καλαμάτας Παράρτημα Σπάρτης
Τμήμα Τεχνολογίας Πληροφορικής και Τηλεπικοινωνιών

Σπάρτη 2013

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή μου και καθηγητή του τμήματος Τεχνολογίας Πληροφορικής και Τηλεπικοινωνιών Καλαμάτας Παράρτημα Σπάρτης, κ. Καραγιώργο Γρηγόρη τόσο για την πολύτιμη βοήθειά του κατά τη διάρκεια εκπόνησης της εργασίας αυτής όσο και για τη γενικότερη καθοδήγησή του. Επίσης, θα ήθελα να ευχαριστήσω την οικογένεια και τους φίλους μου που με βοήθησαν σε όλη αυτή την προσπάθεια.

Περιεχόμενα

Περίληψη

Η ταξινόμηση είναι μία από τις πιο σημαντικές διαδικασίες που εκτελούνται σε ένα υπολογιστή και αυτό οφείλεται στο γεγονός ότι τα ταξινομημένα δεδομένα είναι πιο εύκολο να τα διαχειριστεί κανείς σε σύγκριση με τα τυχαίως διατεταγμένα δεδομένα.

Ένας πολύ γνωστός αλγόριθμος ταξινόμησης είναι ο HeapSort. Ο αλγόριθμος HeapSort χρησιμοποιεί μια δομή δεδομένων που ονομάζεται σωρός με σκοπό να ταξινομήσει ένα σύνολο από τιμές. Ο HeapSort είναι βέλτιστος αλγόριθμος ταξινόμησης που βασίζετε στην πράξη της σύγκρισης. Ο αλγόριθμος αυτός προτάθηκε από τον Williams το 1964 και εργάστηκε επίσης ο Floyd (1962,1964).

Στόχος αυτής της εργασίας είναι να παρουσιαστούν και περιγραφούν οι βασικοί αλγόριθμοι ταξινόμησης και να αναλυθεί σε βάθος ο αλγόριθμος HeapSort. Επίσης θα γίνει περιγραφή τις πρακτικής εφαρμογής του αλγορίθμου HeapSort σε ένα προγραμματιστικό περιβάλλον σε γλώσσα Java το οποίο δημιουργήθηκε ιδικά για την εκπόνηση της πτυχιακής εργασίας.

Εισαγωγή

Αντικείμενο της εργασίας

Οι αλγόριθμοι ταξινόμησης είναι αλγόριθμοι που τοποθετούν τα στοιχεία μίας λίστας με ταξινομημένη σειρά (αύξουσα η φθίνουσα). Η ταξινόμησή τους γίνεται με βάση διάφορα κριτήρια όπως την υπολογιστική πολυπλοκότητα των συγκρίσεων, και την υπολογιστική πολυπλοκότητα των ανταλλαγών. Στη συγκεκριμένη εργασία θα ασχοληθούμε με τους αλγορίθμους ταξινόμησης και ειδικά με τον αλγόριθμο HeapSort. Χρησιμοποιεί μια ειδική δομή δεδομένων που ονομάζεται σωρός (heap). Ο σωρός είναι ένας πίνακας αντικειμένων που μπορεί να αντιμετωπισθεί ως ένα σχεδόν πλήρες δυαδικό δέντρο.

Διάρθρωση Της Εργασίας

- Στο πρώτο κεφάλαιο θα γίνει περιγραφή της έννοιας του αλγορίθμου.
- Στο δεύτερο κεφάλαιο θα γίνει περιγραφή των ποιο γνωστών αλγορίθμων ταξινόμησης.
- Στο τρίτο κεφάλαιο θα γίνει περιγραφή και ανάλυση εις βάθος του αλγορίθμου HeapSort.
- Στο ένθετο θα παρουσιαστεί ο αλγόριθμος HeapSort σε μια πρακτική εφαρμογή του πάνω σε ένα πρόγραμμα σε γλώσσα Java (τόσο σε γραφικό περιβάλλον όσο και σε web περιβάλλον) το οποίο έχει σχεδιαστεί και υλοποιηθεί ειδικά για την εργασία αυτή και είναι εμπνευσμένο από την πρακτική άσκηση.

Κεφάλαιο 1: Η Έννοια Των Αλγορίθμων

1.1 Εισαγωγή

Η λέξη Αλγόριθμος προέρχεται από μία μελέτη του Πέρση μαθηματικού του 8ου αιώνα μ.Χ. Αλ Χουαρίζμι (Abu Ja'far Mohammed ibn Musa Al-Khowarismi), η οποία περιείχε συστηματικές τυποποιημένες λύσεις αλγεβρικών προβλημάτων και αποτελεί ίσως την πρώτη πλήρη πραγματεία άλγεβρας. Πέντε αιώνες αργότερα η μελέτη μεταφράστηκε στα Λατινικά και άρχιζε με τη φράση "Algorithmus dixit ..." (ο Αλγόριθμος είπε...). Έτσι η λέξη αλγόριθμος καθιερώθηκε αργά τα επόμενα χίλια χρόνια με την έννοια «συστηματική διαδικασία αριθμητικών χειρισμών». Τη σημερινή της σημασία την οφείλει στη γρήγορη ανάπτυξη των ηλεκτρονικών υπολογιστών στα μέσα του 20ου αιώνα.

Στην επιστήμη των υπολογιστών ως ενδιάμεσο βήμα σε πολλά προγράμματα χρησιμοποιείται η ταξινόμηση, για τον λόγο αυτό αναπτύχθηκαν ικανοποιητικοί αλγόριθμοι ταξινόμησης. Η καταλληλότητα ενός αλγορίθμου σε μια δεδομένη εφαρμογή εξαρτάται από τον αριθμό των ταξινομητέων στοιχείων, τον βαθμό της αρχικής τους ταξινόμησης, το είδος της συσκευής αποθήκευσης (κύρια μνήμη, δίσκος ή μαγνητική ταινία) και τους πιθανούς περιορισμούς στις τιμές των στοιχείων. [1, σελ. 5,6]

1.2 Τι είναι αλγόριθμος

Ως αλγόριθμος ορίζεται μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος. Πιο απλά αλγόριθμο ονομάζουμε μία σειρά από εντολές που έχουν αρχή και τέλος, είναι σαφείς και εκτελέσιμες που σκοπό έχουν την επίλυση κάποιου προβλήματος.

Ο όρος αλγόριθμος αναφέρεται σε οποιαδήποτε καλά ορισμένη υπολογιστική διαδικασία που δέχεται κάποια τιμή ή σύνολο τιμών ως είσοδο και δίνει κάποια τιμή ή κάποιο σύνολο τιμών ως έξοδο. [1, σελ. 5]

Ας υποθέσουμε ότι θέλουμε να ταξινομήσουμε μια ακολουθία αριθμών κατά αύξουσα σειρά.

Είσοδος: Μια ακολουθία n αριθμών $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$.

Έξοδος: Μια αναδιάταξη $\{\alpha'_1, \alpha'_2, \ldots, \alpha'_n\}$ της ακολουθίας εισόδου τέτοια ώστε $\alpha'_1 \leq \alpha'_2 \leq \ldots \leq \alpha'_n$. [1, σελ. 5]

π.χ. αν η ακολουθία εισόδου ήταν {67, 32, 54, 87, 2, 14} η έξοδος του αλγορίθμου ταξινόμησης θα είναι : {2, 14, 32, 54, 67, 87}.

Οι αλγόριθμοι θα πρέπει να πληρούν κάποια πρότυπα και να διατυπώνονται με συγκεκριμένο τρόπο.

Έτσι ένας αλγόριθμος πρέπει να ικανοποιεί τα επόμενα κριτήρια:

- Να έχει είσοδο δεδομένων
- Έξοδο αποτελεσμάτων
- Καθοριστικότητα
- Περατότητα
- Αποτελεσματικότητα

Είσοδος Δεδομένων - Input

Κατά την εκκίνηση εκτέλεσης του αλγορίθμου καμία, μία ή περισσότερες τιμές δεδομένων πρέπει να δίνονται ως είσοδοι στον αλγόριθμο. Η περίπτωση που δε δίνονται τιμές δεδομένων εμφανίζεται όταν ο αλγόριθμος δημιουργεί και επεξεργάζεται κάποιες πρωτογενείς τιμές με τη βοήθεια συναρτήσεων παραγωγής τυχαίων αριθμών ή με τη βοήθεια άλλων απλών εντολών. Η είσοδος η οποία χρειάζεται για τον υπολογισμό της λύσης ενός προβλήματος δηλώνεται με τον όρο **στιγμιότυπο προβλήματος**.[2, σελ. 25]

Έξοδος Αποτελεσμάτων - Output

Δίδει τουλάχιστον ένα μέγεθος ως αποτέλεσμα που εξαρτάται κατά κάποιο τρόπο από τις αρχικές εισόδους. Ο αλγόριθμος πρέπει να δημιουργεί τουλάχιστον μία τιμή (δεδομένων) ως αποτέλεσμα προς το χρήστη ή προς ένα άλλο αλγόριθμο. [2, σελ. 25]

Καθοριστικότητα - Definiteness

Κάθε εντολή πρέπει να καθορίζεται χωρίς καμία αμφιβολία για τον τρόπο εκτέλεσής της. π.χ. Σε μία διαίρεση να λαμβάνεται υπόψη και η περίπτωση όπου ο διαιρετέος λαμβάνει μηδενική τιμή. Τυπικές περιπτώσεις: η διαίρεση με το μηδέν, υπόριζος να έχει ποσότητα αρνητική, κλπ. Προβλήματα καθοριστικότητας αντιμετωπίζονται συχνά με τη λογική της επιλογής, δηλ. Αν α>0 τότε ..... αλλιώς .....(if....else....).[2, σελ. 25]

Πεοατότητα - Finiteness

Κάθε εκτέλεση είναι πεπερασμένη, δηλαδή τελειώνει ύστερα από έναν πεπερασμένο αριθμό διεργασιών ή βημάτων. Μία διαδικασία που δεν τελειώνει μετά από συγκεκριμένο/πεπερασμένο αριθμό βημάτων λέγεται απλά υπολογιστική διαδικασία. [2, σελ. 26]

Αποτελεσματικότητα - Effectiveness

Όλες οι διαδικασίες που περιλαμβάνει μπορούν να πραγματοποιηθούν με ακρίβεια και σε πεπερασμένο χρόνο. Κάθε μεμονωμένη εντολή του αλγορίθμου να είναι απλή (και όχι σύνθετη). Δηλαδή μία εντολή δεν αρκεί να έχει ορισθεί αλλά πρέπει να είναι και εκτελέσιμη. [2, σελ. 26]

Ανάλογα με την τεχνική επίλυσης ενός προβλήματος οι αλγόριθμοι διακρίνονται σε:

- Αναδρομικοί (recursive): αλγόριθμοι που χρησιμοποιούν αναδρομικές λύσεις προβλημάτων, π.χ. πολυώνυμα Hermite, υπολογισμός παραγοντικού, κ.α.
- Διαίρει και Βασίλευε (divide and conquer): επιλύουν το πρόβλημα αναγάγοντάς το σε μικρότερα ανάλογα προβλήματα, π.χ. quick sort, merge sort, κ.α.
- Άπληστοι (greedy): επιλύουν προβλήματα επιλέγοντας κάθε φορά την τοπικά βέλτιστη λύση προσδοκώντας την συνολικά βέλτιστη, π.χ. πρόβλημα επιστροφής ρέστων, χρονικός προγραμματισμός, κ.α.
- Δυναμικού Προγραμματισμού (dynamic programming): συνήθως αναδρομικοί αλγόριθμοι οι οποίοι χρησιμοποιούν τις ενδιάμεσα παραγόμενες λύσεις, π.χ. πολλαπλασιασμός αλυσίδας πινάκων, κ.α.
- Παράλληλοι (parallel): εύρεση λύσης όπου το πρόβλημα δεν λύνεται σειριακά αλλά πολλές σχέσεις του εκτελούνται παράλληλα, π.χ. πολλά προβλήματα πινάκων, τεχνικές τύπου «διαίρει και βασίλευε», κ.α.

Επίσης, ανάλογα με την λύση που επιτυγχάνουν μπορούν να διακριθούν σε:

- Βέλτιστοι: εύρεση της βέλτιστης λύσης του προβλήματος, π.χ. επίλυσης μίας εξίσωσης δευτέρου βαθμού.
- Προσεγγιστικοί: εύρεση «καλών» λύσεων σε άλυτα ή πολύ δύσκολα προβλήματα, π.χ. χρονοπρογραμματισμός εργασιών, χρωματισμός χάρτη κ.α.

Δημιουργία Αλγορίθμου

Τα βήματα δημιουργίας αλγόριθμου είναι:

1. Διατύπωση του προβλήματος
2. Κατανόηση του προβλήματος
3. Λύση του προβλήματος
4. Διατύπωση του αλγόριθμου
5. Έλεγχος της λύσης

1.3 Περιγραφή και αναπαράσταση αλγορίθμων

Τέσσερις είναι οι βασικοί τρόποι αναπαράστασης ενός αλγορίθμου:

- με **ελεύθερο κείμενο**, που αποτελεί τον πιο ανεπεξέργαστο και αδόμητο τρόπο παρουσίασης αλγορίθμου. Έτσι εγκυμονεί τον κίνδυνο ότι μπορεί εύκολα να οδηγήσει σε μη εκτελέσιμη παρουσίαση παραβιάζοντας το τελευταίο χαρακτηριστικό των αλγορίθμων, δηλαδή την αποτελεσματικότητα.
- με **διαγραμματικές τεχνικές**, που συνιστούν ένα γραφικό τρόπο παρουσίασης του αλγορίθμου. Από τις διάφορες διαγραμματικές τεχνικές που έχουν επινοηθεί, η πιο παλιά και η πιο γνωστή ίσως, είναι το διάγραμμα ροής (flow chart). Ωστόσο η χρήση διαγραμμάτων ροής για την παρουσίαση αλγορίθμων δεν αποτελεί την καλύτερη λύση, γι'αυτό και εμφανίζονται όλο και σπανιότερα στη βιβλιογραφία και στην πράξη.
- με **φυσική γλώσσα** κατά βήματα. Στην περίπτωση αυτή χρειάζεται προσοχή, γιατί μπορεί να παραβιασθεί το τρίτο βασικό χαρακτηριστικό ενός αλγορίθμου.
- με **κωδικοποίηση**, δηλαδή με ένα πρόγραμμα γραμμένο σε κάποια γλώσσα προγραμματισμού που όταν εκτελεσθεί θα δώσει τα ίδια αποτελέσματα με τον αλγόριθμο.

1.5 Ασυμπτωτικός Συμβολισμός

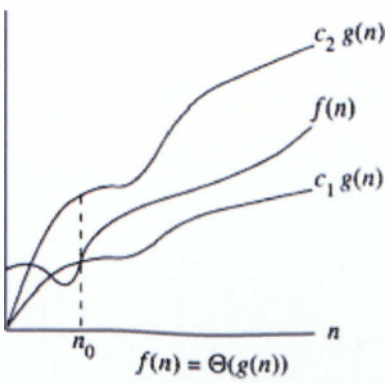
Οι διάφοροι τύποι συμβολισμού που χρησιμοποιούμε για την περιγραφή του ασυμπτωτικού χρόνου εκτέλεσης ενός αλγόριθμου ορίζονται με βάση συναρτήσεις των οποίων το πεδίο ορισμού είναι το σύνολο των φυσικών αριθμών N = {0, 1, 2, ...}. Οι συμβολισμοί αυτοί προσφέρονται για την περιγραφή της συνάρτησης του χρόνου εκτέλεσης χειρότερης περίπτωσης T(n), ο όποιος συνήθως ορίζεται μόνο για ακέραιο μέγεθος εισόδου. [1, σελ. 42]

1.5.1 Συμβολισμός Θ

Για κάποια δεδομένη συνάρτηση g(n), το σύμβολο Θ(g(n)) δηλώνει το σύνολο των συναρτήσεων

$$\Theta(g(n)) = \{f(n) : \text{υπάρχουν θετικές σταθερές } c_1, c_2, \text{ και } n_0 \text{ τέτοιες ώστε}$$
$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ για κάθε } n \ge n_0\}.$$

Μια συνάρτηση f(n) ανήκει στο σύνολο Θ(g(n)) αν υπάρχουν θετικές σταθερές c_1 και c_2 τέτοιες ώστε η f(n) να μπορεί να είναι μεταξύ των $c_1 g(n)$ και $c_2 g(n)$, από κάποια τιμή του n και πάνω. Δεδομένου ότι το Θ(g(n)) είναι ένα σύνολο, θα μπορούσαμε να δηλώσουμε ότι η f(n) είναι μέλος του Θ(g(n)) γράφοντας $f(n) \in \Theta(g(n))$. Αντί αυτού για να περιγράψουμε τη συγκεκριμένη έννοια θα χρησιμοποιήσουμε συνήθως την έκφραση f(n) = Θ(g(n)). Τέλος λέμε ότι η g(n) αποτελεί ένα συμπτωτικά αυστηρό φράγμα για την f(n). [1, σελ. 43]



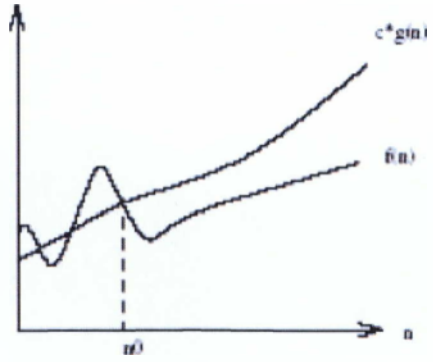
Εικόνα 1

1.5.2 Συμβολισμός O

Όταν έχουμε μόνο ένα ασυμπωτικό άνω φράγμα, χρησιμοποιούμε τον συμβολισμό O. Για μια δεδομένη συνάρτηση g(n), η έκφραση O(g(n)) δηλώνει το σύνολο των συναρτήσεων

$$O(g(n)) = \{f(n) : \text{υπάρχουν θετικές σταθερές } c \text{ και } n_o \text{ τέτοιες ώστε}$$
$$0 \leq f(n) \leq cg(n) \text{ για κάθε } n \geq n_o\}.$$

Για όλες τις τιμές του n πέραν του n_0, η τιμή της f(n) κείται κάτω από την cg(n) ή συμπίπτει με αυτή. [1, σελ. 45]



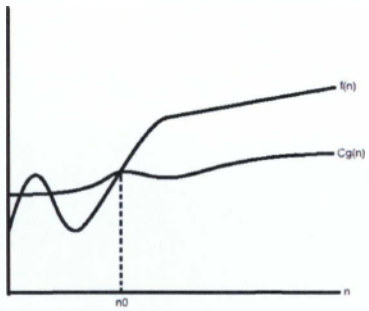
Εικόνα 2

1.5.3 Συμβολισμός Ω

Όπως ακριβώς ο συμβολισμός Ο ορίζει ένα ασυμπτωτικό άνω φράγμα για μια συνάρτηση, ο συμβολισμός **Ω** ορίζει αντίστοιχα ένα συμπωτικό κάτω φράγμα. Για μια δεδομένη συνάρτηση g(n), η έκφραση Ω(g(n)) δηλώνει το σύνολο των συναρτήσεων

$$\Omega(g(n)) = \{f(n) : \text{υπάρχουν θετικές σταθερές } c \text{ και } n_0 \text{ τέτοιες ώστε}$$
$$0 \leq cg(n) \leq f(n) \text{ για κάθε } n \geq n_0\}.$$

Για όλες τις τιμές του n πέραν του n_0, η τιμή της f(n) κείται επάνω από την cg(n) ή συμπίπτει με αυτή. [1, σελ. 47]



Εικόνα 3

1.6 Κατηγορίες Πολυπλοκότητας Συναρτήσεων

Η πολυπλοκότητα αυξάνεται καθώς κατεβαίνουμε.

Σταθερές	5, 1/2, 3, π, e
Λογαριθμικές	logn, $\log^2 n$, $\log^n n$
Πολυωνυμικές	$\sqrt[3]{n}$, $\sqrt{n}$, an+b,
Εκθετικές	2^n
Παραγοντικές	n!
Υπερεκθετικές	n^n
Ackerman	2^{2^n}

Πίνακας 1

Κεφάλαιο 2: Αλγόριθμοι Ταξινόμησης

2.1 Εισαγωγή

Οι αλγόριθμοι ταξινόμησης είναι αλγόριθμοι που τοποθετούν τα στοιχεία μίας λίστας με φθίνουσα ή αύξουσα σειρά. Οι αλγόριθμοι ταξινόμησης που χρησιμοποιούνται ταξινομούνται με βάση :

1. Την υπολογιστική πολυπλοκότητα (χειρότερης, μέσης και καλύτερης) των συγκρίσεων των στοιχείων από την άποψη του μεγέθους της λίστας(n). Για χαρακτηριστικούς αλγόριθμους ταξινόμησης καλύτερης περίπτωσης είναι $O(n \log n)$ και χειρότερης είναι $O(n^2)$. Η μέση περίπτωση για μία ταξινόμηση είναι $O(n)$.
2. Την υπολογιστική πολυπλοκότητα των ανταλλαγών.
3. Την χρήση μνήμης και άλλων υπολογιστών πόρων, έτσι ώστε μόνο O(1) ή O(log n) μνήμη απαιτείται πέρα από τα στοιχεία που ταξινομούνται, ενώ άλλοι πρέπει να δημιουργήσουν τις βοηθητικές θέσεις για τα στοιχεία που αποθηκεύονται προσωρινά.
4. Την επαναληπτικότητα. Μερικοί αλγόριθμοι είναι είτε επαναλαμβανόμενοι είτε μη επαναλαμβανόμενοι, ενώ άλλοι μπορούν να είναι και τα δύο.
5. Χρησιμοποιούν την: εισαγωγή (insertion), ανταλλαγή (exchange), επιλογή (selection), συγχώνευση (merging), κατανομή (*distrbution),* διαμερισμού *(partition),* των στοιχείων .

Στη συνέχεια περιγράφονται εν συντομία οι πιο γνωστοί αλγόριθμοι ταξινόμησης.

2.2 Βασικοί Αλγόριθμοι Ταξινόμησης

Αλγόριθμος φυσαλίδας (*bubble sort*)

Ένας από τους πιο ευρέως χρησιμοποιούμενους αλγόριθμους αναζήτησης είναι ο αλγόριθμος φυσαλίδας (*bubble sort*), ο οποίος είναι ένας απλός και ευθύς αλγόριθμος για την ταξινόμηση συλλογών από δεδομένα. Ο αλγόριθμος αυτός ξεκινά από την αρχή της λίστας και συγκρίνει τα πρώτα δύο στοιχεία. Αν το πρώτο στοιχείο είναι το μεγαλύτερο, τότε εναλλάσσει τη θέση των δύο στοιχείων. Έπειτα συνεχίζει να συγκρίνει τα υπόλοιπα ζεύγη, δηλαδή το δεύτερο στοιχείο με το τρίτο και ούτω καθεξής. Όταν φτάσει στο τέλος της λίστας, ξεκινά πάλι από την αρχή δουλεύοντας με τον ίδιο τρόπο και σταματά όταν σε ένα πέρασμα της λίστας δε γίνει καμία εναλλαγή στοιχείων.

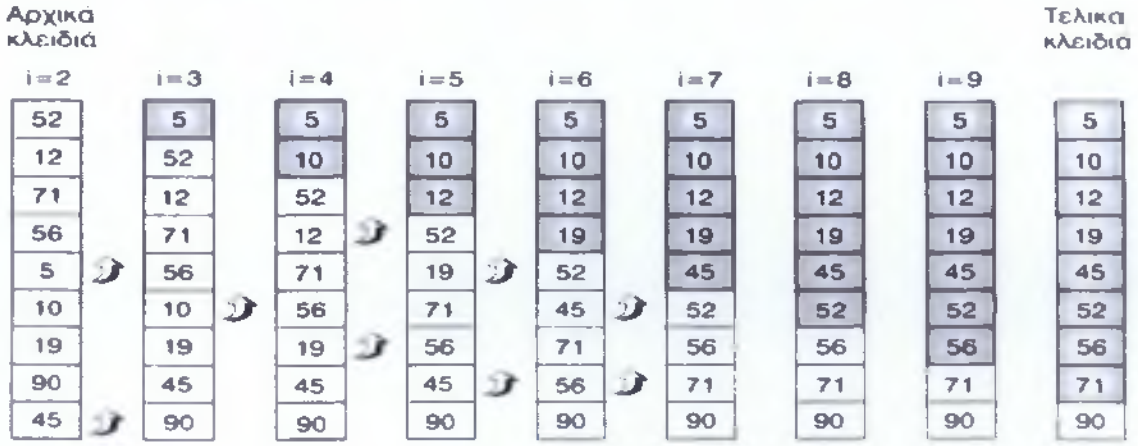
Η πολυπλοκότητα καλύτερης περίπτωσης του συγκεκριμένου αλγόριθμου είναι O(n) και της χειρότερης περίπτωσης $O(n^2)$, όπου n είναι πάλι το μέγεθος της λίστας.

Η ταξινόμηση φυσαλίδας έχει τα εξής χαρακτηριστικά:

- Απλή υλοποίηση
- Αποτελεσματικός για πολύ μικρά σύνολα δεδομένων
- Βελτιώνεται η απόδοσή του όταν τα στοιχεία είναι ήδη ταξινομημένα
- Χρησιμοποιεί την ανταλλαγή στοιχείων για την υλοποίηση του.

Παράδειγμα :

Έστω ότι ο αρχικός πίνακας αποτελείται από εννέα κλειδιά τα εξής: 52, 12, 71, 56, 5, 10, 19, 90 και 45. Η μέθοδος εφαρμοζόμενη σε αυτά τα εννέα κλειδιά εξελίσσεται όπως φαίνεται στον επόμενο πίνακα. Κάθε φορά το ταξινομημένο τμήμα του πίνακα εμφανίζεται με χρώμα, ενώ τα στοιχεία που σαν φυσαλίδες ανέρχονται μέσα στον πίνακα εντοπίζονται με το αντίστοιχο βέλος στα δεξιά τους. Κάθε φορά εμφανίζεται η τάξη της επανάληψης (i).



Πίνακας 2

Αλνόριθμος: (Υλοποίηση σε java) : [1]

```
//java class

public class BubbleSort
{

 public void bubbleSort(int[] arr){
    for(int i=0; i<arr.length; i++){
      for(int j=1; j<arr.length; j++){
         if(arr[j]< arr[j-1] ){
            int temp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
         }
      }
    }

    for(int i=0; i<arr.length; i++)
    {
       System.out.print(arr[i] + " ");
    }
}

}
```

[1] Ο κώδικας που παραθέτω έχει χρησιμοποιηθεί από την πηγή http://en.wikipedia.org.

Αλγόριθμος επιλογής (*selection sort*)

Ένας άλλος απλός αλγόριθμος ταξινόμησης είναι ο αλγόριθμος επιλογής (*selection sort*) που έχει βελτιωμένη απόδοση σε σχέση με τον αλγόριθμο φυσαλίδας. Ο αλγόριθμος αυτός αρχικά βρίσκει το μικρότερο σημείο της λίστας και το εναλλάσσει με το πρώτο στοιχείο της λίστας. Στη συνέχεια βρίσκει το μικρότερο από τα άλλα στοιχεία, το εναλλάσσει με το δεύτερο στοιχείο της λίστας και ούτω καθεξής. Η επίδοση του αλγόριθμου επιλογής δεν επηρεάζεται από την προηγούμενη σειρά τοποθέτησης των στοιχείων της λίστας και εκτελεί τον ίδιο αριθμό επαναλήψεων εξαιτίας της απλής δομής του.

Η ταξινόμηση επιλογής έχει τα εξής χαρακτηριστικά:

- Απλή υλοποίηση
- Αποτελεσματικός για πολύ μικρά σύνολα δεδομένων
- Χρησιμοποιεί την επιλογή των στοιχείων για την υλοποίηση του.

Αλγόριθμος: (Υλοποίηση σε java) : [2]

```
public static void selectionSort1(int[] x) {
  for (int i=0; i<x.length-1; i++) {
    for (int j=i+1; j<x.length; j++) {
      if (x[i] > x[j]) {
        //... Exchange elements
        int temp = x[i];
        x[i] = x[j];
        x[j] = temp;
      }
    }
  }
}
```

[2] Ο κώδικας που παραθέτω έχει χρησιμοποιηθεί από την πηγή http://www.leepoint.net/notes-java/data/arrays/31arrayselectionsort.html

Αλγόριθμος εισαγωγής (*insertion sort*)

Ο αλγόριθμος εισαγωγής (*insertion sort*) είναι ένας απλός αλγόριθμος ταξινόμησης που είναι σχετικά αποτελεσματικός σε μικρές λίστες και σχεδόν ταξινομημένες λίστες και συνήθως χρησιμοποιείται ως μέρος άλλων πιο περίπλοκων αλγόριθμων. Ο αλγόριθμος παίρνει ένα ένα τα στοιχεία της λίστας και τα τοποθετεί σε μια νέα λίστα σε ταξινομημένη σειρά. Η λειτουργία της εισαγωγής είναι όμως, «ακριβή» γιατί απαιτεί τη μετατόπιση των υπόλοιπων στοιχείων μία θέση αριστερά. Ο εν λόγω αλγόριθμος έχει πολυπλοκότητα καλύτερης περίπτωσης O(n) και χειρότερης περίπτωσης $O(n^2)$.

Η ταξινόμηση εισαγωγής έχει τα εξής χαρακτηριστικά:

- Απλή υλοποίηση
- Αποτελεσματικός για πολύ μικρά σύνολα δεδομένων
- Χρησιμοποιεί την εισαγωγή των στοιχείων για την υλοποίηση του.

Παράδειγμα:

Στο παρακάτω παράδειγμα βλέπουμε πως λειτουργεί η ταξινόμηση εισαγωγής. Η εύρεση της κατάλληλης θέσης γίνεται εύκολα με διαδοχικές συγκρίσεις και μετακινήσεις. Δηλαδή, το στοιχείο χ συγκρίνεται με τα προηγούμενα στοιχεία table[i-1], table[i-2], κ.τ.λ. (προς τα αριστερά) μέχρι να βρεθεί κάποιο στοιχείο table[k] με τιμή μικρότερη από το χ. Κατόπιν τα κλειδιά από table[i-1] έως table[k+1] μετακινούνται προς τα δεξιά και το χ καταλαμβάνει τη σωστή θέση στην ταξινομημένη ακολουθία.

Αρχικά κλειδιά	52	12	71	56	5	10	19	90	45
$i=2$	12	52	71	56	5	10	19	90	45
$i=3$	12	52	71	56	5	10	19	90	45
$i=4$	12	52	56	71	5	10	19	90	45
$i=5$	5	12	52	56	71	10	19	90	45
$i=6$	5	10	12	52	56	71	19	90	45
$i=7$	5	10	12	19	52	56	71	90	45
$i=8$	5	10	12	19	52	56	71	90	45
$i=9$	5	10	12	19	45	52	56	71	90

Πίνακας 3

Αλγόριθμος (Υλοποίηση σε java) : [3]

```
void insertionSort(int[] arr) {
    int i, j, newValue;
    for (i = 1; i < arr.length; i++) {
        newValue = arr[i];
        j = i;
        while (j > 0 && arr[j - 1] > newValue) {
            arr[j] = arr[j - 1];
            j--;
        }
        arr[j] = newValue;
    }
}
```

[3] Ο κώδικας που παραθέτω έχει χρησιμοποιηθεί από την πηγή http://en.wikipedia.org

Αλνόριθμος συνένωσης (*merge sort)*

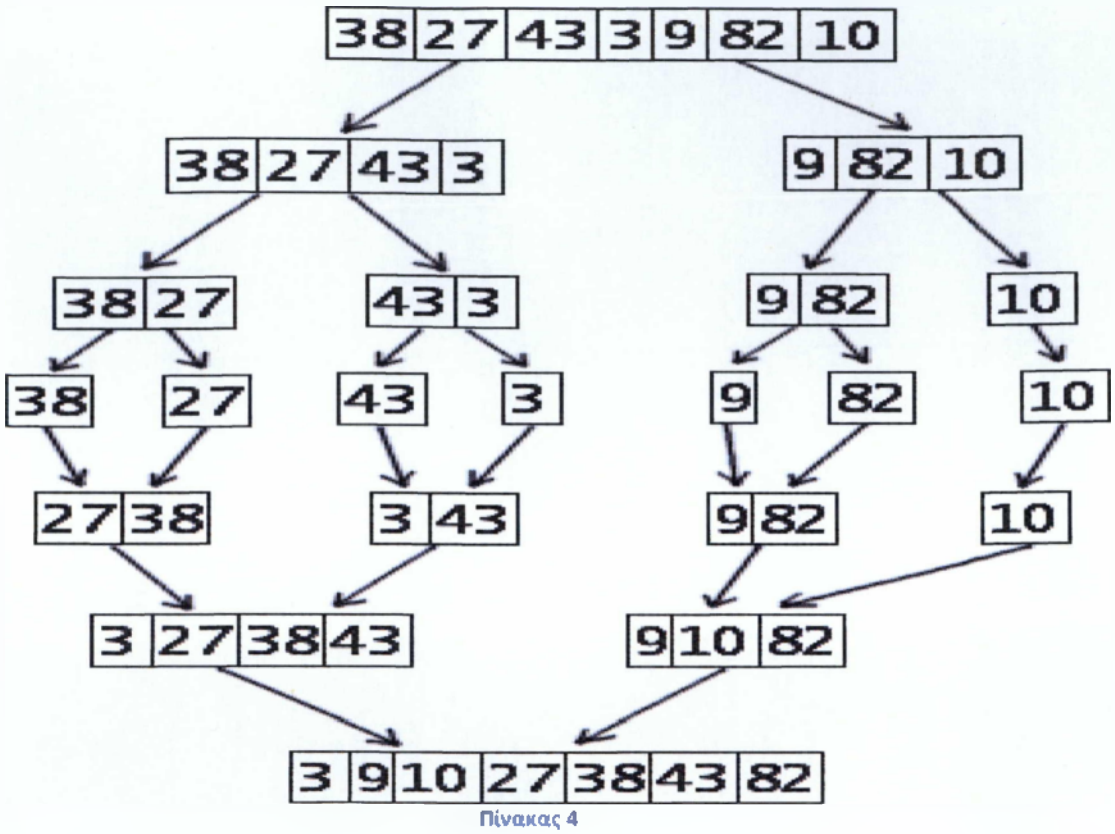
Ο αλγόριθμος συνένωσης (*διαίρι και βασίλευε*) έχει ως πλεονέκτημα την ευκολία που παρουσιάζει η συνένωση ήδη ταξινομημένων λιστών σε μια νέα ταξινομημένη λίστα. Αρχίζει συγκρίνοντας κάθε ζευγάρι στοιχείων της λίστας (δηλαδή το πρώτο με το δεύτερο, το τρίτο με το τέταρτο και ούτω καθεξής) και εναλλάσσει τα στοιχεία αν το πρώτο είναι μεγαλύτερο του δεύτερου. Στη συνέχεια συνενώνει τα ζευγάρια των ταξινομημένων λιστών των δύο στοιχείων σε ταξινομημένες λίστες των τεσσάρων στοιχείων, στη συνέχεια των οχτώ και ούτω καθεξής, μέχρι να συνενωθούν δύο λίστες στην τελική ταξινομημένη λίστα. Ο εν λόγω αλγόριθμος έχει θεωρητική πολυπλοκότητα O(n log n).

Ο αλγόριθμος ταξινόμησης συνένωσης έχει τα εξής χαρακτηριστικά:

- Είναι ο μόνος αλγόριθμος μεγάλης ταχύτητας
- Χρησιμοποιεί την συγχώνευση των στοιχείων για την υλοποίηση του.

Παράδειγμα:

Στον παρακάτω πίνακα βλέπουμε πώς λειτουργεί η ταξινόμηση συνένωσης. Αρχίζει συγκρίνοντας κάθε ζευγάρι στοιχείων της λίστας (δηλαδή το πρώτο με το δεύτερο, το τρίτο με το τέταρτο και ούτω καθεξής) και εναλλάσσει τα στοιχεία αν το πρώτο είναι μεγαλύτερο του δεύτερου. Στη συνέχεια συνενώνει τα ζευγάρια των ταξινομημένων λιστών των δύο στοιχείων σε ταξινομημένες λίστες των τεσσάρων στοιχείων, στη συνέχεια των οχτώ και ούτω καθεξής, μέχρι να συνενωθούν δύο λίστες στην τελική ταξινομημένη λίστα.



Πίνακας 4

Αλγόριθμος (Υλοποίηση σε java):[4]

```
static void mergeSort(int[] A) {
    if (A.length > 1) {
      int q = A.length/2;
      int[] leftArray = Arrays.copyOfRange(A, 0, q);
      int[] rightArray = Arrays.copyOfRange(A,q+1,A.length);
      mergeSort(leftArray);
      mergeSort(rightArray);
      A = merge(leftArray,rightArray);
    }
  }

  static int[] merge(int[] l, int[] r) {
    int totElem = l.length + r.length;
    int[] a = new int[totElem];
    int i,li,ri;
    i = li = ri = 0;
    while ( i < totElem) {
      if ((li < l.length) && (ri<r.length)) {
```

[4] Ο κώδικας που παραθέτω έχει χρησιμοποιηθεί από την πηγή
http://stackoverflow.com/questions/13727030/mergesort-in-java

```
            if (l[li] < r[ri]) {
               a[i] = l[li];
               i++;
               li++;
            }
            else {
               a[i] = r[ri];
               i++;
               ri++;
            }
         }
         else {
            if (li >= l.length) {
               while (ri < r.length) {
                  a[i] = r[ri];
                  i++;
                  ri++;
               }
            }
            if (ri >= r.length) {
               while (li < l.length) {
                  a[i] = l[li];
                  li++;
                  i++;
               }
            }
         }
      }
      return a;

   }

}
```

Αλγόριθμος γρήγορης ταξινόμησης *(quick sort)*

Ο αλγόριθμος γρήγορης ταξινόμησης_είναι αλγόριθμος διαίρει και βασίλευε που στηρίζεται σε μια λειτουργία που ονομάζεται Διαμέριση (partition). Για τη διαμέριση ενός πίνακα, επιλέγουμε ένα στοιχείο, επονομαζόμενο ως βήμα (pivot), και μετακινούμε όλα τα μικρότερα στοιχεία πριν το στοιχείο βήμα και τα μεγαλύτερα μετά από αυτό. Η λειτουργία αυτή μπορεί να γίνει αποτελεσματικά σε γραμμικό χρόνο. Στη συνέχεια επαναληπτικά ταξινομούμε τις μικρότερες και μεγαλύτερες υπολίστες.

Ο εν λόγω αλγόριθμος είναι ένας από τους ταχύτερους αλγόριθμους ταξινόμησης, το πιο περίπλοκο δε θέμα σε αυτόν είναι η επιλογή του στοιχείου – βήματος. Λανθασμένες επιλογές αυτού του στοιχείου μπορούν να κοστίσουν σε δραματική μείωση της επίδοσης σε $O(n^2)$. Αν όμως, σε κάθε βήμα διαλέγουμε το μεσαίο στοιχείο της λίστας τότε ο αλγόριθμος εκτελείται σε O(n log n).

Η γρήγορη ταξινόμηση έχει τα εξής χαρακτηριστικά:

- Πολυπλοκότητα χειρότερης περίπτωσης $O(n^2)$
- Χρησιμοποιεί τον διαμερισμό των στοιχείων για την υλοποίηση του.

Παράδειγμα:

Πίνακας 5

Αλγόριθμος(Υλοποίηση σε java)[5]

```
int partition(int arr[], int left, int right)
{
    int i = left, j = right;
    int tmp;
    int pivot = arr[(left + right) / 2];

    while (i <= j) {
        while (arr[i] < pivot)
            i++;
        while (arr[j] > pivot)
            j--;
        if (i <= j) {
            tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i++;
            j--;
        }
    };

    return i;
}

void quickSort(int arr[], int left, int right) {
    int index = partition(arr, left, right);
    if (left < index - 1)
        quickSort(arr, left, index - 1);
    if (index < right)
        quickSort(arr, index, right);
}
```

[5] Ο κώδικας που παραθέτω έχει χρησιμοποιηθεί από την πηγή http://en.wikipedia.org

Αλγόριθμος *heapsort*

Ένας άλλος γνωστός αλγόριθμος ταξινόμησης είναι ο αλγόριθμος heapsort. Όπως και ο αλγόριθμος επιλογής, έτσι και αυτός ο αλγόριθμος δουλεύει ψάχνοντας το μεγαλύτερο (ή μικρότερο) στοιχείο της λίστας, τοποθετώντας το στο τέλος (ή στην αρχή) και συνεχίζει με το υπόλοιπο της λίστας. Η διαφορά είναι όμως, ότι ο αλγόριθμος heapsort εκτελεί αυτή τη διαδικασία πιο αποτελεσματικά χρησιμοποιώντας ένα τύπο δεδομένων που ονομάζεται heap, που ουσιαστικά είναι ένας ειδικός τύπος δυαδικού δέντρου.

Μόλις τα στοιχεία της λίστας σχηματίσουν το heap, η ρίζα του δέντρου είναι το μεγαλύτερο στοιχείο. Τότε αφαιρείται και τοποθετείται στο τέλος της λίστας και σχηματίζεται ξανά το heap με αποτέλεσμα η ρίζα του δέντρου να είναι πάλι το μεγαλύτερο στοιχείο. Χρησιμοποιώντας το heap για να βρεθεί το μεγαλύτερο στοιχείο της λίστας απαιτείται O(log n) χρόνος αντί για O(n) που χρειάζεται για μια σειριακή σάρωση στον απλό αλγόριθμο επιλογής. Αυτό επιτρέπει στον heapsort να εκτελείται σε χρόνο O(n logn).

Η ταξινόμηση σωρού έχει τα εξής χαρακτηριστικά:

- πολυπλοκότητα χειρότερης περίπτωσης O(n logn)
- Χρησιμοποιεί την επιλογή των στοιχείων για την υλοποίηση του.

Αλγόριθμος (Υλοποίηση σε java)

Ο παρακάτω κώδικας είναι ο κώδικας ο όποιος έχει χρησιμοποιηθεί στο πρόγραμμα το όποιο έχει υλοποιηθεί για την πρακτική εφαρμογή του HeapSort η οποία περιγράφεται στο Ένθετο.

```
public class HeapSort {

  public static void heapsort(int[] data, int n) {
    // sort first n elements of data array via heapsort; 0 <= n < data.length
    // first make heap
    for (int r = (n - 2) / 2; r >= 0; r -= 1) // work backwards from last parent node = (last index
- 1)/2
    {
      heapify(data, r, n);
    }
    // sort heap by swapping biggest element to end of list,
    //   and make heap again of the rest
    while (n > 1) {
      n -= 1;
      int temp = data[n]; // swap last unsorted element with top
      data[n] = data[0];
      data[0] = temp;
```

```
        heapify(data, 0, n);
      }
    }

    private static void heapify(int[] data, int root, int n) // let out-of-place element at index
root
    //   settle down through tree to make a legal heap
    // start assuming any children of root are at tops of legal heaps
    // all indices of heap elements have index < n
    {
      int left = 2 * root + 1;
      while (left < n) {  // while root is not a leaf
        int right = left + 1;
        int rootdata = data[root]; // need a copy for swaps
        if (right == n) { // right past end, only one leaf
          if (data[left] > rootdata) { // swap if smaller at root
            data[root] = data[left];
            data[left] = rootdata;
          }
          return;  // already fixed child, which is the final leaf
        } else { // two leaves
          int bigger = right; // find bigger of two children
          if (data[left] > data[bigger]) {
            bigger = left;
          }
          if (rootdata >= data[bigger]) {
            return;  // OK, now is legal heap below
          }            // else root out of place
          data[root] = data[bigger]; //   swap root with bigger child
          data[bigger] = rootdata;
          root = bigger;  // descend in heap for next loop
          left = 2 * root + 1;
        }
      }
    }
}
```

Κεφάλαιο 3: Ανάλυση Αλγορίθμου HeapSort

Για να μπορέσουμε να αναλύσουμε τον αλγόριθμο HeapSort θα πρέπει πρώτα να αναφέρουμε και να αναλύσουμε το σωρό (heap), τις λειτουργίες και τα είδη του.

3.1 Σωροί (Heap)

3.1.1 Εισαγωγή

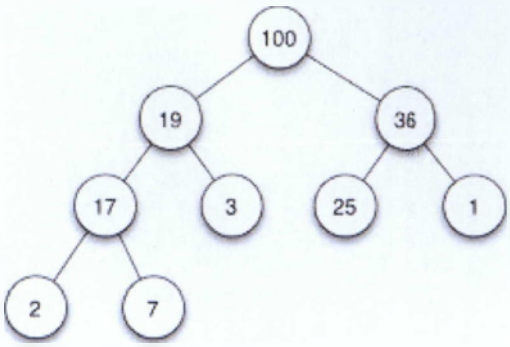
Ο σωρός είναι μια δενδρική δομή και χρησιμοποιείται για την δημιουργία ουρών προτεραιότητας (priority queues). Η ρίζα του δέντρου περιλαμβάνει το μικρότερο ή μεγαλύτερο στοιχείο του, αναλόγως, αν έχουμε σωρό ελαχίστων ή μεγίστων. Τα επόμενα δύο στοιχεία του δένδρου είναι τα παιδιά του. Γενικότερα αν ο πατέρας είναι στη θέση i τα παιδιά του θα είναι στην θέση 2i (αριστερό παιδί) και 2i+1 (δεξί παιδί) αντίστοιχα. Αν i η θέση ενός παιδιού i/2 είναι η θέση του πατέρα του. Κάθε σωρός με n στοιχεία έχει ύψος $\log_2 n$. Ο σωρός, όπως και τα δέντρα γενικότερα, μπορεί να υλοποιηθεί με πίνακα, στον οποίο εισάγονται τα κλειδιά του σωρού από αριστερά προς τα δεξιά και από πάνω προς τα κάτω.

Τα είδη των σωρών είναι:

- Οι σωροί μεγίστων (max heap)
- Οι σωροί ελαχίστων (min heap)
- Διπλός σωρός (double-ended heap,deap)
- Αριστερίστικος σωρός (leftist heap)
- Δυωνυμικοί σωροί (binomial heap)
- Σωροί Fibonacci

Ο διπλός σωρός, αριστερίστικος σωρός, δυωνυμικός σωρός και ο σωρός Fibonacci περιγράφονται στο **ένθετο**.

Στην εικόνα φαίνεται ένας σωρός και η αντίστοιχη υλοποίησή του σε πίνακα



Εικόνα 4

Οι βασικές λειτουργίες ενός σωρού είναι:

- Η εισαγωγή(Insert) ενός στοιχείου στον σωρό
- Η διαγραφή(Delete) ενός στοιχείου από τον σωρό

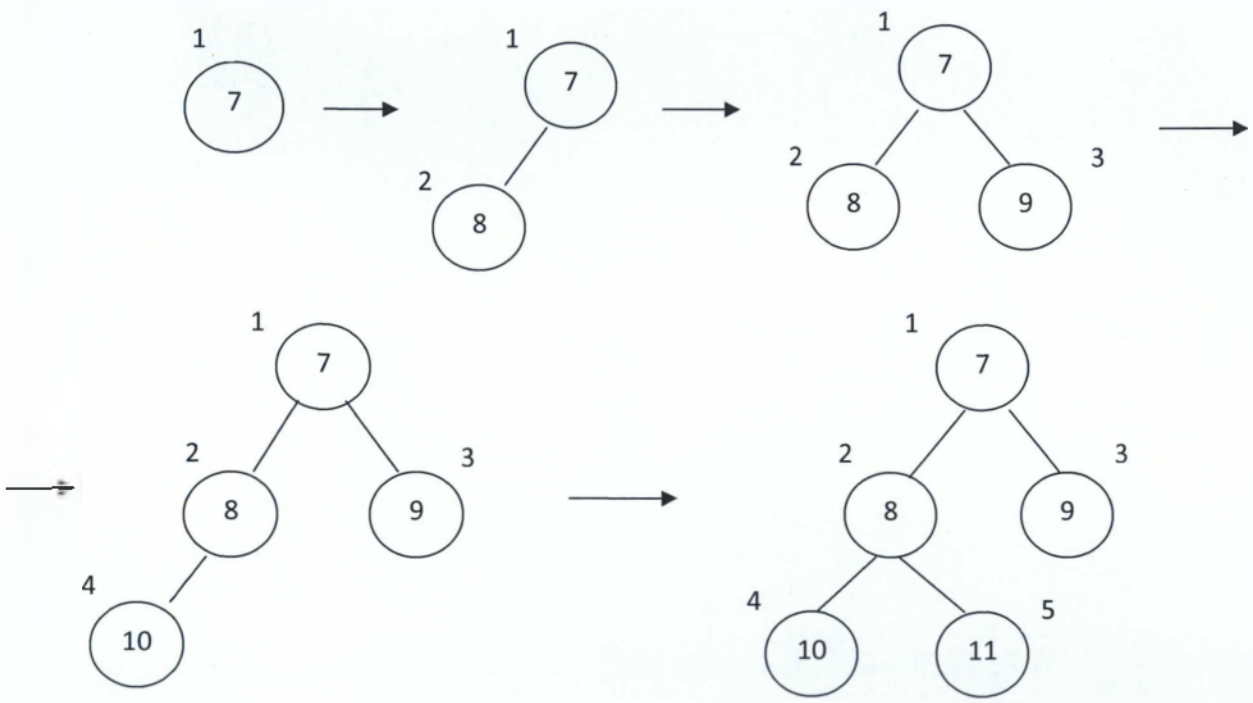
3.1.2 Κατασκευή Σωρού

Ο σωρός μπορεί να κατασκευαστεί χρησιμοποιώντας τον παρακάτω αλγόριθμο, ο οποίος καλεί τον αλγόριθμο εισαγωγής ενός στοιχείου. Θεωρούμε ότι τα στοιχεία έρχονται με μια σειρά και δεν είναι γνωστά εκ των προτέρων.

```
BUILDHEAP (A(n))
{
for i = 1 to n do
Insert (A(n),i])
end for
}
```

Παρακάτω φαίνεται ένα παράδειγμα κατασκευής σωρού. Ας υποθέσουμε ότι έχουμε την ακολουθία στοιχείων 1,2,3,4. Η διαδικασία κατασκευής του σωρού φαίνεται στο παρακάτω σχήμα:

1	2	3	4	5
7	8	9	10	11



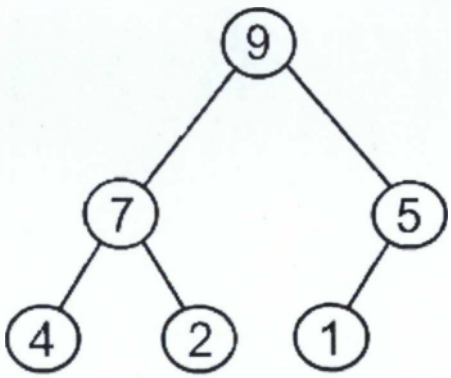
Στην χείριστη περίπτωση, κάθε στοιχείο που εισάγεται θα ξεκινήσει από φύλλο του σωρού και θα καταλήξει στην ρίζα, άρα αν εισάγεται στο βήμα i, τότε θα χρειαστεί O(log i) βήματα.

Ωστόσο μπορούμε να επιτύχουμε καλύτερο χρόνο για την κατασκευή του σωρού αν γνωρίζουμε από την αρχή όλη την ακολουθία εισόδου και την έχουμε αποθηκευμένη σε έναν πίνακα. Τότε θα θεωρήσουμε ότι ο αρχικός πίνακας εισόδου A είναι η αναπαράσταση ενός αταξινόμητου σωρού και εμείς θα επιβάλλουμε την ιδιότητα του σωρού στον πίνακα αυτόν, ξεκινώντας από τους χαμηλότερους εσωτερικούς κόμβους και πηγαίνοντας προς την ρίζα. Για τον σκοπό αυτό χρησιμοποιούμε την Heapify η οποία αναλύεται στο επόμενο κεφάλαιο (**3.2.2 Ανάλυση της HeapSort**).

3.1.3 Δομή σωρού μεγίστουν (max heap)

Σωρός μεγίστουν είναι ένα δένδρο, το οποίο ικανοποιεί δύο συνθήκες:

- Η τιμή του κλειδιού κάθε κόμβου είναι μεγαλύτερη ή ίση από τις τιμές των κλειδιών των παιδιών του.
- Είναι ένα συμπληρωμένο δέντρο, που σημαίνει ότι μπορεί να προκύψει από ένα πλήρες δέντρο αφαιρώντας έναν αριθμό στοιχείων από το τέλος.



Εικόνα 5

Οι διαδικασίες της εισαγωγής και της διαγραφής υλοποιούνται με τους ακόλουθους τρόπους:

Εισαγωγή (Insert)

Όταν εισάγουμε ένα καινούργιο στοιχείο στο σωρό αυτό τοποθετείται στην επόμενη διαθέσιμη θέση στο τελευταίο επίπεδο. Αν η τιμή του σωρού δεν παραβιάζει την ιδιότητα του σωρού (δηλαδή η τιμή του είναι μικρότερη η ίση από αυτή του γονέα του) τότε το στοιχείο μένει στη θέση του. Σε διαφορετική περίπτωση παιδί και γονέας ανταλλάσσουν θέσεις. Αυτή η διαδικασία συνεχίζεται μέχρι το καινούργιο στοιχείο να βρει μια θέση τέτοια ώστε να ικανοποιείται η ιδιότητα του σωρού. Έτσι κάθε στοιχείο που εισάγεται στο σωρό θα πρέπει να διασχίσει ένα μονοπάτι από το φύλλο στο οποίο βρισκόταν αρχικά μέχρι κάποιον κατάλληλο εσωτερικό κόμβο. Στη χειρότερη περίπτωση το μονοπάτι θα είναι ως τη ρίζα του δέντρου. Ο χρόνος εισαγωγής ενός στοιχείου στο δέντρο είναι O(logn) όπου n το πλήθος των στοιχείων του σωρού. Στη χειρότερη περίπτωση θα χρειαστούν τόσα βήματα όσο και το ύψος του δέντρου.

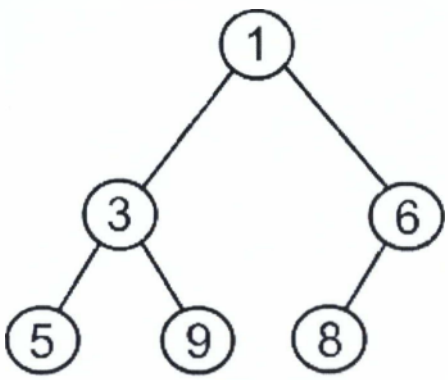
Διαγραφή (Delete)

Το μεγαλύτερο στοιχείο βρίσκεται πάντα στη ρίζα αλλά όταν το απομακρύνουμε δημιουργείται ένα κενό. Το κενό πρέπει να καλυφθεί με τέτοιο τρόπο ώστε ο σωρός να τηρεί την ιδιότητά του. Μετά την απομάκρυνση του μεγαλύτερου στοιχείου τοποθετείται στη ρίζα το τελευταίο στοιχείο του σωρού και το τρέχον μέγεθος μειώνεται κατά ένα. Αν το στοιχείο που περιέχει τώρα η ρίζα είναι μεγαλύτερο ή ίσο με τα παιδιά του μένει ως έχει. Σε διαφορετική περίπτωση το μεγαλύτερο παιδί παίρνει τη θέση του γονέα και αυτό συνεχίζεται μέχρι το στοιχείο που ήταν στη ρίζα να βρεθεί σε μία θέση στο σωρό που οι τιμές των παιδιών του να μην είναι μεγαλύτερες από τη δικιά του. Και σε αυτήν τη περίπτωση ο χρόνος διαγραφής είναι O(logn) όπου n το πλήθος των στοιχείων του σωρού.

3.1.4 Δομή σωρού ελαχίστουν (min heap)

Σωρός ελαχίστουν είναι ένα δένδρο, το οποίο ικανοποιεί δύο συνθήκες:

- ✓ Η τιμή του κλειδιού κάθε κόμβου είναι μικρότερη ή ίση από τις τιμές των κλειδιών των παιδιών του.
- ✓ Είναι ένα συμπληρωμένο δέντρο, που σημαίνει ότι μπορεί να προκύψει από ένα πλήρες δέντρο αφαιρώντας έναν αριθμό στοιχείων από το τέλος.

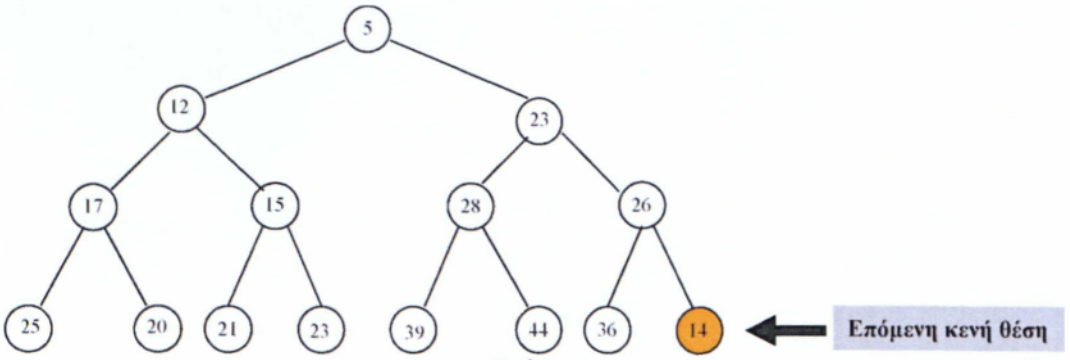


Εικόνα 6

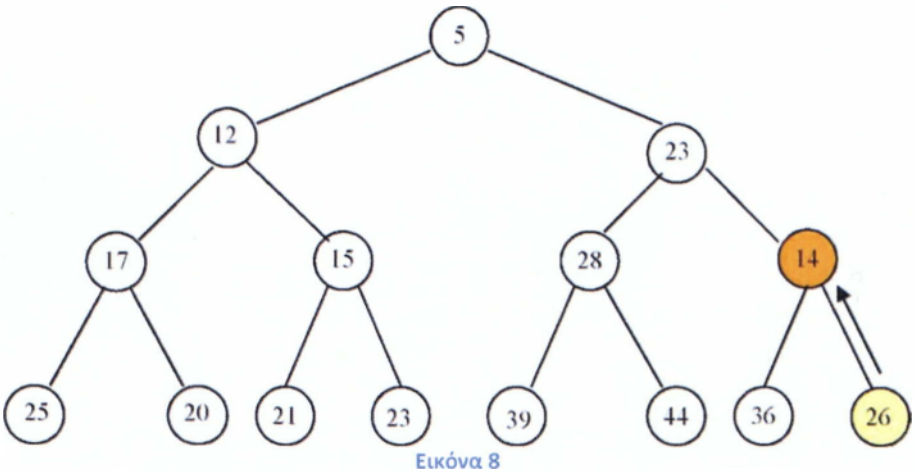
Οι διαδικασίες της εισαγωγής και της διαγραφής υλοποιούνται με τους ακόλουθους τρόπους:

Εισαγωγή στοιχείου (Insert)

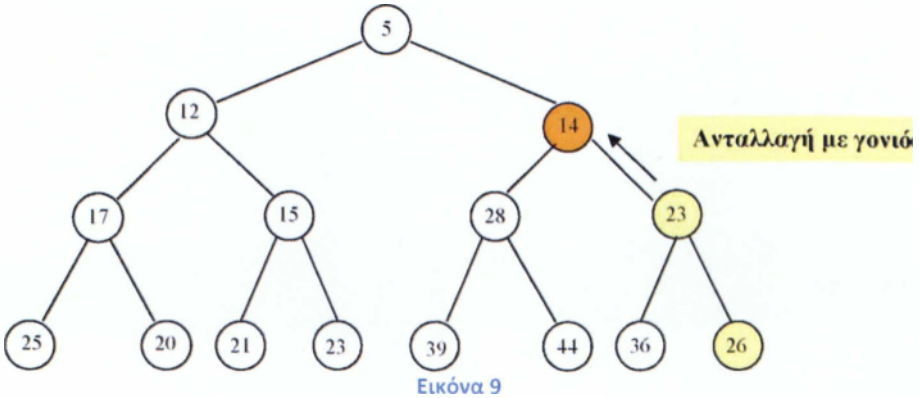
Για να εισάγουμε ένα στοιχείο στο σωρό, αρχικά το προσθέτουμε στο τέλος του. Αν το στοιχείο που εισάγουμε είναι μικρότερο από τον γονιό του, τότε αλλάζουν θέση. Επαναλαμβάνουμε αυτή τη διαδικασία μέχρι να μην παραβιάζεται ο κανόνας αυτός. Παρακάτω φαίνεται ένα παράδειγμα εισαγωγής στοιχείου στον σωρό.



Εικόνα 7



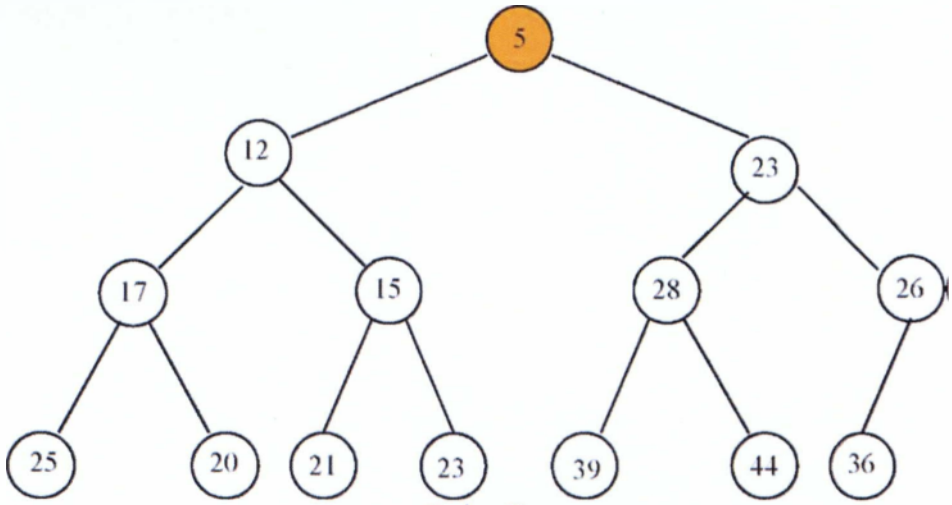
Εικόνα 8



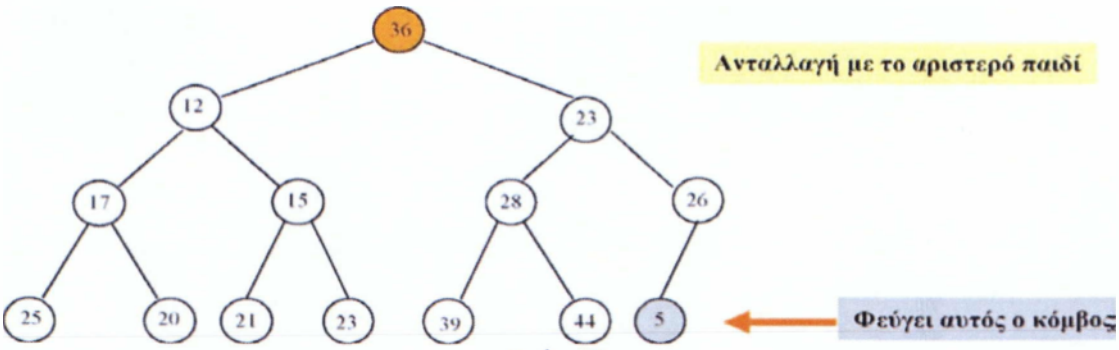
Εικόνα 9

Διαγραφή στοιχείου (delete min)

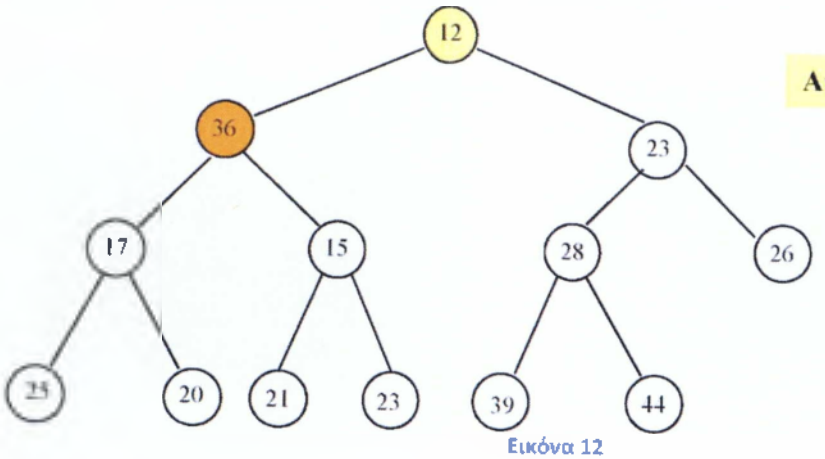
Για να διαγράψουμε το στοιχείο του σωρού ακολουθούμε την εξής μέθοδο. Πρώτα διαγράφουμε το στοιχείο της κορυφής και βάζουμε στη θέση του το τελευταίο στοιχείο του σωρού. Στη συνέχεια κάνουμε τις κατάλληλες αλλαγές, εφόσον αυτές απαιτούνται για να μην παραβιάζεται ο 1ος κανόνας του σωρού ελαχίστων. Παρακάτω φαίνεται ένα παράδειγμα διαγραφής στοιχείου από τον σωρό.



Εικόνα 10

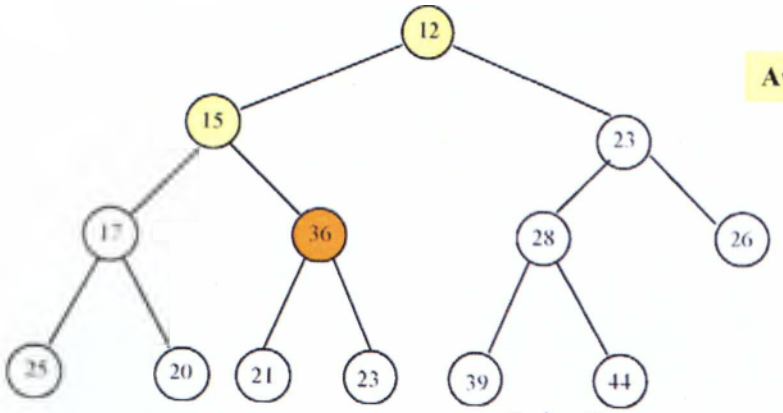


Εικόνα 11



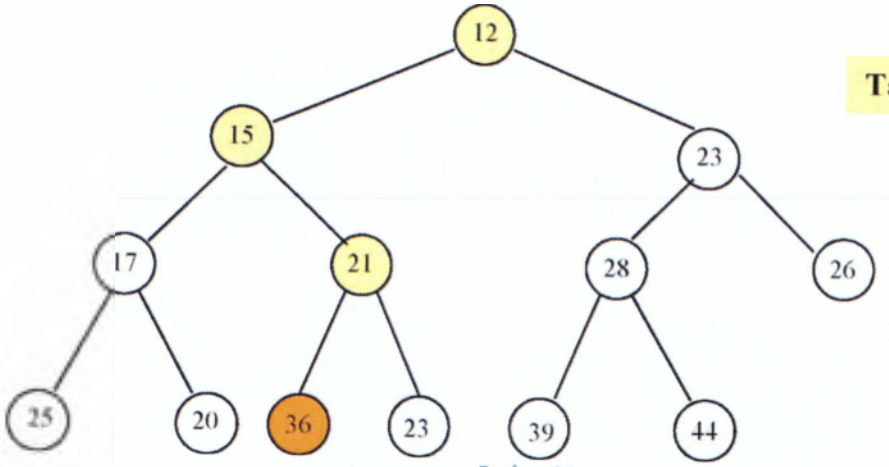
Ανταλλαγή με δεξί παιδί

Εικόνα 12



Ανταλλαγή με αριστερό παιδί

Εικόνα 13



Τέλος: σωρός

Εικόνα 14

3.2 Ανάλυση της HeapSort

3.2.1 Εισαγωγή

Ο αλγόριθμος Heapsort είναι ένας αλγόριθμος ταξινόμησης και ανήκει στην οικογένεια αλγορίθμων ταξινόμησης επιλογής με συγκρίσεις. Η μέθοδος αυτή προτάθηκε από τον Williams το 1964. Στη μέθοδο αυτή εργάστηκε επίσης και ο Floyd (1962,1964). Έχουν προταθεί κάποιες παραλλαγές του αλγορίθμου Heapsort.

Ο BOTTOM-UP-HEAPSORT είναι μια παραλλαγή του Heapsort που εκτελεί 1.5n log n+O(n) βασικές συγκρίσεις στη χειρότερη περίπτωση. Η ιδέα είναι να αναζητηθεί η πορεία προς το φύλλο ανεξαρτήτως της θέσης του στοιχείου της ρίζας. Δεδομένου ότι το αναμενόμενο βάθος είναι υψηλό αυτή η πορεία διαπερνιέται bottom-up. Ο Fleischer (1991) όπως και ο Schaffer και ο Sedgewick (1993) θεωρούν ότι ο αλγόριθμος αυτός απαιτεί τουλάχιστον 1.5n log$n$ — $o(n$ log n) συγκρίσεις. Κάποιος μπορεί να συμπεράνει ότι ο μέσος αριθμός συγκρίσεων σε αυτήν την παραλλαγή είναι οριακός κοντά στο n log n + O(n).

Ο MDR-HEAPSORT προτάθηκε από τον McDiarmid και τον Reed (1989). Ο αλγόριθμος αυτός εκτελεί λιγότερες από $n$log$n$ + cn συγκρίσεις στη χειρότερη περίπτωση και επεκτείνεται τον BOTTOM-UP-HEAPSORT με τη χρησιμοποίηση ενός bit για να κωδικοποιήσει σε ποιο κλάδο μπορεί να βρεθεί το μικρότερο στοιχείο και ένα άλλο για να χαρακτηρίσει εάν αυτή η πληροφορία είναι άγνωστη.

Ο αλγόριθμος WEAK-HEAPSORT είναι πιο κομψός και πιο γρήγορος. Αντί για δύο bits ανά στοιχείο ο *WEAK-HEAPSORT* χρησιμοποιεί μόνο ένα.

Ο *ULTIMATE-HEAPSORT* προτάθηκε από τον Katajainen το 1998. Ο αλγόριθμος αυτός αποφεύγει τα χειρότερα παραδείγματα περίπτωσης του *BOTTOM-UP-HEAPSORT* με τον περιορισμό του συνόλου σωρών σε σωρούς δύο στρωμάτων. Είναι δυσκολότερο να εγγυηθεί αυτή η περιορισμένη μορφή. Ο αλγόριθμος αυτός πετυχαίνει μικρότερο αριθμό συγκρίσεων χειρότερης περίπτωσης και ο αριθμός των συγκρίσεων στη μέση περίπτωση είναι μεγαλύτερος του BOTTOM-UP-HEAPSORT.

3.2.2 Αλγόριθμος HeapSort

Ο αλγόριθμος του HeapSort αποτελείται από 2 φάσεις:

1. Τη φάση της δημιουργίας του σωρού (Buildheap) και
2. Τη φάση της επεξεργασίας του σωρού.

Ο αλγόριθμος HeapSort κατασκευάζει ως πρώτο βήμα μέσω του Buildheap έναν σωρό μεγίστου από μια συστοιχία εισόδου A[1..n] όπου n = μήκος [A]. Σε κάθε κόμβο που διεξέρχεται καλεί την Heapify η οποία αποκαθιστά την ιδιότητα του σωρού.

Κατά τη φάση της επεξεργασίας του σωρού το κλειδί της ρίζας εναλλάσσεται με το τελευταίο κλειδί του σωρού και στο εξής δεν λαμβάνεται υπ όψη. Όμως με την αλλαγή αυτή τα εναπομείναντα κλειδιά δεν αποτελούν πλέον σωρό μεγίστου. Έτσι ακολουθεί η διαδικασία Heapify για να αποκαταστήσει την ιδιότητα του σωρού. Πιο συγκεκριμένα, το νέο κλειδί που βρέθηκε στη ρίζα του σωρού συγκρίνεται με το μεγαλύτερο από τα κλειδιά των κόμβων που είναι παιδιά της ρίζας, Αν το νέο κλειδί είναι μικρότερο, τότε τα κλειδιά ανταλλάσσονται. Η διαδικασία της καθόδου του νέου κλειδιού από τη ρίζα προς κατώτερα επίπεδα συνεχίζεται μέχρι να βρεθεί μικρότερο κλειδί ή να φτάσει και πάλι στο επίπεδο των φύλλων. Κατόπιν και πάλι το κλειδί της νέας ρίζας εναλλάσσεται με το κλειδί του (προ-) τελευταίου κόμβου του πίνακα.

Κώδικας του αλγορίθμου Heapsort

```
Heapify(A, i)
{ l <- left(i) r <- right(i)
if l <= heapsize[A] and A[l] > A[i]
        then largest <- l
        else largest <- i
 if r <= heapsize[A] and A[r] > A[largest]
         then largest <- r
 if largest != i
        then swap A[i] <-> A[largest]
            Heapify(A, largest)
}
Buildheap(A)
{ heapsize[A] <- length[A]
for i <- |length[A]/2| down to 1
        do Heapify(A, i)
}
Heapsort(A)
{ Buildheap(A)
for i <- length[A] down to 2
        do swap A[1] <-> A[i]
        heapsize[A] <- heapsize[A] - 1
        Heapify(A, 1)
}
```

Ανάλυση πολυπλοκότητας

Έστω *T(n)* ο χρόνος που χρειάζεται ο αλγόριθμος Heapsort για να εφαρμοστεί σε έναν πίνακα μεγέθους n.

Ο χρόνος που χρειάζεται ο αλγόριθμος για να ολοκληρωθεί είναι

$$T(n)= T_{buildheap}(n) + T_{Heapify}(n) +\Theta(n-1)$$

HEAPIFY [6]

Η διαδικασία Heapify (αποκατάσταση σωρού μεγίστου) είναι ένα σημαντικό υποπρόγραμμα για τη διαχείριση σωρών μεγίστου. Δέχεται ως είσοδο μια συστοιχία Α και η θέση i είναι κάποιο στοιχείο της συστοιχίας. Η λειτουργία της διαδικασίας προϋποθέτει ότι τα δυαδικά δένδρα που εκφύονται από τους κόμβους Αριστερός(i)[7] και Δεξιός(i)[8] είναι σωροί μεγίστου, αλλά το Α[i][9] ενδέχεται να είναι μικρότερο από τους θυγατρικούς του κόμβους, δηλαδή να παραβιάζει την ιδιότητα του σωρού μεγίστου. Η Heapify μετακυλύει την τιμή του Α[i] προς τα κάτω στον σωρό μεγίστου έτσι ώστε το υποδένδρο που εκφύεται από το στοιχείο στη θέση i να καταστεί σωρός μεγίστου.[1, σελ.128]

Σε κάθε βήμα, προσδιορίζεται το μέγιστο των στοιχείων Α[i], Α[Αριστερός(i)] και Α[Δεξιός(i)] και αποθηκεύεται στην μεταβλητή *μεγίστου (largest)* η θέση του *[βλέπε Κώδικας του αλγορίθμου Heapsort].* Εάν το μέγιστο είναι το Α[i] , τότε το υποδένδρο που εκφύεται από τον κόμβο i αποτελεί σωρό μεγίστου, οπότε η διαδικασία τερματίζεται. Διαφορετικά, το μέγιστο στοιχείο βρίσκεται σε έναν από τους δυο θυγατρικούς κόμβους, οπότε η διαδικασία εναλλάσσει τα στοιχεία Α[i] και Α[*μεγίστου*]. Με τον τρόπο αυτό ικανοποιείται η ιδιότητα του σωρού μεγίστου. [1, σελ.128]

[6] Αποκατάσταση σωρού μεγίστου
[7] Αριστερός ή left ή αριστερός θυγατρικού (i) = επιστροφή 2i
[8] Δεξιός ή right ή δεξιός θυγατρικού (i) = επιστροφή 2i+1
[9] Ρίζα δένδρου

Ο χρόνος εκτέλεσης της Heapify σε ένα υποδένδρο μεγέθους n είναι:

$$T_{heapify}(n) = \Theta(1) + T_{heapify}(\text{μέγεθος υποδένδρου})$$

Όταν δεν γίνονται ανταλλαγές η καλύτερη περίπτωση της Heapify είναι:

$$T_{best}(n) = \Theta(1)$$

Έτσι πρέπει να ξέρουμε πόσο μεγάλα μπορεί να είναι τα υποδένδρα του σωρού με n στοιχεία. Μια πρώτη προσέγγιση θα έδειχνε ότι ένα υποδένδρο θα μπορούσε να είναι το μισό του δένδρου μιας και ο σωρός είναι δυαδικό δένδρο.

Αν ένας σωρός *A* έχει μέγεθος *n*, τα υποδένδρα του έχουν μέγεθος μικρότερο ή ίσο με 2*n*/3. Άρα :

$$T_{heapify}(n) \leq T_{heapify}(2n/3) + \Theta(1)$$

Σύμφωνα με *το κεντρικό θεώρημα*[10], η αναδρομική αυτή σχέση έχει λύση :

$$T_{heapify}(n) = O(\log n)$$

[10] **Κεντρικό θεώρημα:**
T(n) = aT(n/b) + f(n)
1. Αν $f(n) = O(n^{\log_b a - e})$ για κάποια σταθερά e>0, τότε $T(n) = \Theta(n^{\log_b a})$.
2. Αν $f(n) = \Theta(n^{\log_b a})$, τότε $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. Αν $f(n) = \Omega(n^{\log_b a + e})$ για κάποια σταθερά e>0, και αν $af(n/b) \leq cf(n)$ για κάποια σταθερά $c < 1$ και για όλα τα n από κάποια τιμή και πάνω, τότε $T(n) = \Theta(f(n))$.

BUILDHEAP[11]

Η Buildheap διεξέρχεται από τους κόμβους του δένδρου και εκτελεί σε κάθε ένα από αυτούς την Heapify. Μπορούμε να υπολογίσουμε ένα απλό άνω φράγμα για τον χρόνο εκτέλεσης της Buildheap ως εξής. Η κάθε κλήση της Heapify απαιτεί χρόνο O(logn), και εκτελούνται O(n) τέτοιες κλήσεις. Επομένως ο χρόνος εκτέλεσης είναι O(n logn). Αυτό το άνω φράγμα, αν και σωστό δεν είναι ασυμπωτικά αυστηρό.

Μπορούμε να προσδιορίσουμε ένα αυστηρότερο φράγμα παρατηρώντας ότι ο χρόνος εκτέλεσης της Heapify για έναν συγκεκριμένο κόμβο εξαρτάται από το ύψος του κόμβου στο δένδρο. Η αυστηρότερη αυτή ανάλυση βασίζεται στις ιδιότητες ότι ένας σωρός n στοιχείων έχει ύψος logn και το πολύ $n/2^{h+1}$ κόμβους ύψους h.

Ο χρόνος που απαιτείται από την συνάρτηση MAX-HEAPIFY όταν καλείται σε έναν κόμβο του ύψους h είναι O(h). Έτσι μπορούμε εκφράσουμε το συνολικό κόστος της συνάρτησης BUILD-MAX-HEAP ως [1, σελ. 133]

$$\sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor lgn \rfloor} \left\lceil \frac{h}{2^h} \right\rceil \right)$$

$$\sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^k} = \frac{1/2}{(1-1/2)^2} = 2$$

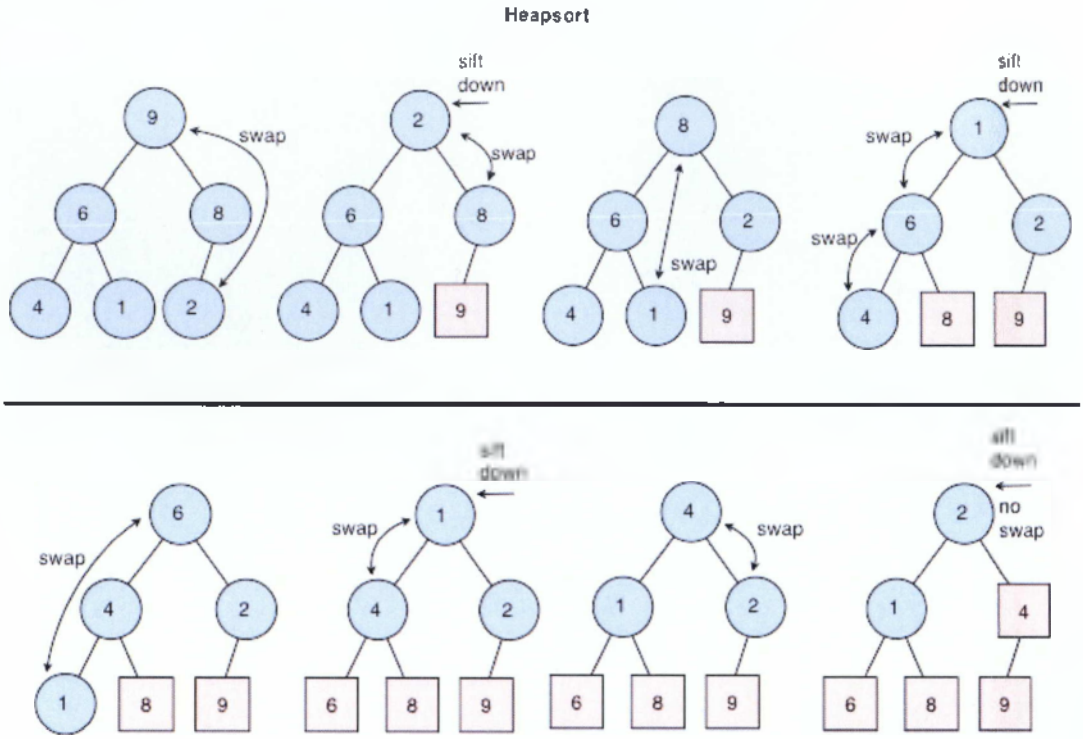
Κατά συνέπεια, ο χρόνος υλοποίησης του BUILD-MAX-HEAP μπορεί να υπολογιστεί ως [1, σελ. 133]

$$O\left(n \sum_{h=0}^{\lfloor lgn \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

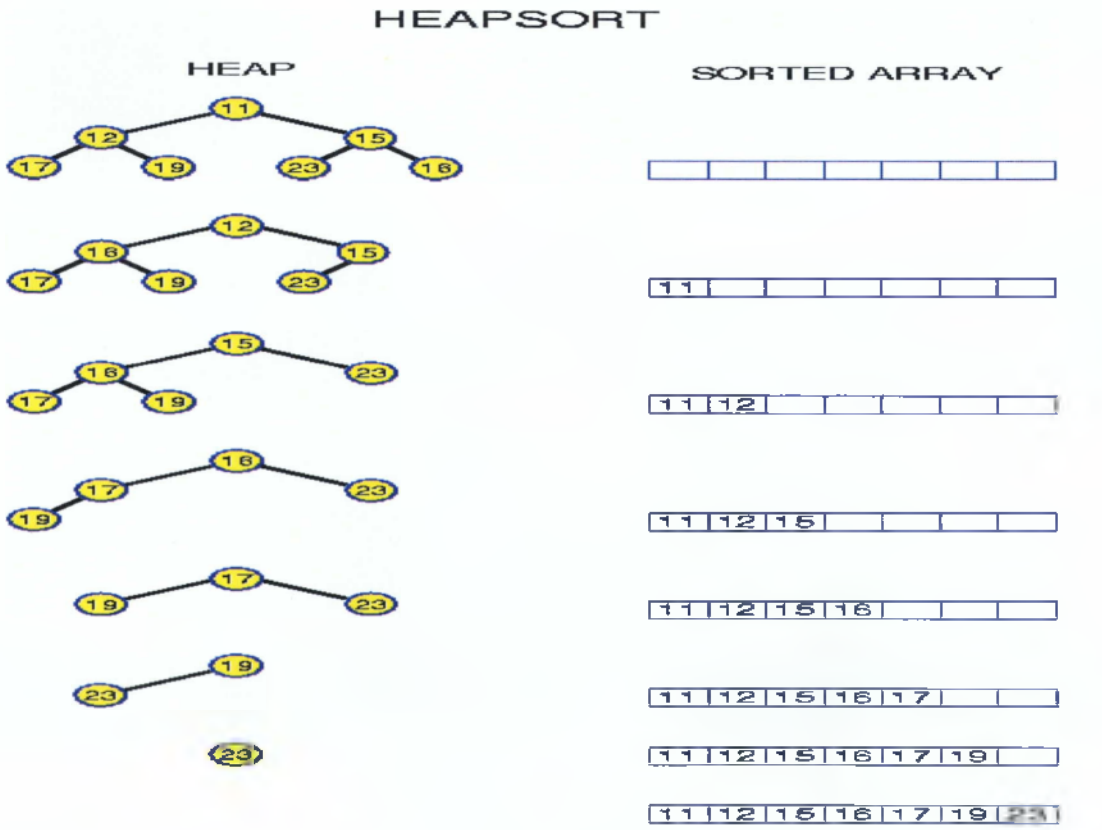
Άρα η πολυπλοκότητα της κατασκευής του σωρού φράσσεται από O(n).

Άρα τελικά η διαδικασία του αλγορίθμου HEAPSORT απαιτεί χρόνο **O(n logn)**, δεδομένου ότι η κλήση στην Buildheap απαιτεί χρόνο O(n) και κάθε μια από τις n-1 κλήσεις στην Heapify απαιτεί χρόνο O(logn).

[11] Κατασκευή σωρού μεγίστου



Εικόνα 15



Εικόνα 16

Υλοποίηση σε Java

Ο παρακάτω κώδικας είναι ο κώδικας ο όποιος έχει χρησιμοποιηθεί στο πρόγραμμα το όποιο έχει υλοποιηθεί για την πρακτική εφαρμογή τις HeapSort η οποία περιγράφεται στο Ένθετο.

```
public class HeapSort {

   public static void heapsort(int[] data, int n) {
      // sort first n elements of data array via heapsort; 0 <= n < data.length
      // first make heap
      for (int r = (n - 2) / 2; r >= 0; r -= 1) // work backwards from last parent node = (last index
- 1)/2
      {
         heapify(data, r, n);
      }
      // sort heap by swapping biggest element to end of list,
      //   and make heap again of the rest
      while (n > 1) {
         n -= 1;
         int temp = data[n]; // swap last unsorted element with top
         data[n] = data[0];
         data[0] = temp;
         heapify(data, 0, n);
      }
   }

   private static void heapify(int[] data, int root, int n) // let out-of-place element at index
root
   //   settle down through tree to make a legal heap
   // start assuming any children of root are at tops of legal heaps
   // all indices of heap elements have index < n
   {
      int left = 2 * root + 1;
      while (left < n) {  // while root is not a leaf
         int right = left + 1;
         int rootdata = data[root]; // need a copy for swaps
         if (right == n) { // right past end, only one leaf
            if (data[left] > rootdata) { // swap if smaller at root
               data[root] = data[left];
               data[left] = rootdata;
            }
            return;  // already fixed child, which is the final leaf
         } else { // two leaves
            int bigger = right; // find bigger of two children
            if (data[left] > data[bigger]) {
               bigger = left;
            }
            if (rootdata >= data[bigger]) {
               return;  // OK, now is legal heap below
```

```
            }           // else root out of place
            data[root] = data[bigger]; //  swap root with bigger child
            data[bigger] = rootdata;
            root = bigger;  // descend in heap for next loop
            left = 2 * root + 1;
        }
    }
  }
}
```

3.3 Σύγκριση Θετικά Αρνητικά HeapSort

Θετικά :

1. Ο Αλγόριθμος είναι βέλτιστος και έχει εγγυημένο χρόνο διεκπεραίωσης O(n logn).
2. Ο Αλγόριθμος είναι ένα από τα λίγα μη-αναδρομικά είδη O(n logn).
3. Αρκετά εύκολος για τον εντοπισμό σφαλμάτων (debug), διότι είναι μη-επαναληπτικός και αποτελείται από δυο μικρότερα και μη-σχετικά τμήματα.
4. Ο Αλγόριθμος είναι αποτελεσματικός όταν τα δεδομένα εισάγονται ταξινομημένα κατά την αντίστροφη φορά.
5. Η χειρότερη περίπτωση έχει την ίδια πολυπλοκότητα με τη μέση περίπτωση, ενώ όπως είναι γνωστό δεν συμβαίνει το ίδιο με τον QuickSort.

Αρνητικά:

1. Ο αλγόριθμος δεν είναι τόσο γρήγορος κατά μέσο όρο, όσο ο αλγόριθμος QuickSort.
2. Στη χειρότερη περίπτωση εκτελεί περίπου (2n logn) συγκρίσεις, ενώ ο QuickSort εκτελεί στη μέση περίπτωση περίπου (1,38n logn) συγκρίσεις.
3. Πολυπληθείς κώδικας για την υλοποίηση των εργασιών του σωρού.

Οι επιδόσεις του αλγορίθμου HeapSort (ταξινόμηση με σωρό) σε σύγκριση με άλλους αλγορίθμους

	Συγκρίσεις
Insertion sort	O(n) $O(n^2)$ $O(n^2)$
Bubble sort	$O(n^2)$ $O(n^2)$ $O(n^2)$
Selection sort	$O(n^2)$ $O(n^2)$ $O(n^2)$
Quick sort	O(n logn) O(n logn) $O(n^2)$
Heap sort	**O(n logn)** **O(n logn)**
Merge sort	O(n logn) O(n logn) O(n logn)

Πίνακας 6

Επίλογος – Συμπεράσματα

Η εργασία αυτή πραγματεύτηκε τις γνώσεις και τα στοιχεία που απαιτούνται για την κατανόηση του αλγορίθμου HeapSort. Αναλύθηκαν έννοιες που αφορούσαν τις κατηγορίες αλγορίθμων ταξινόμησης και του σωρού (Heap). Αφιερώθηκε χρόνος για την δημιουργία προγράμματος, σε γλώσσα προγραμματισμού JAVA και PHP, στο οποίο υλοποιήθηκε η HeapSort για την ταξινόμηση της βάσης δεδομένων του προγράμματος καταγραφής βλαβών.

Στόχος μου ήταν να αναλύσω και να παρουσιάσω τον αλγόριθμο HeapSort και θα ήθελα να επισημάνω τα εξής:

- Ο αλγόριθμος HeapSort είναι βέλτιστος και έχει εγγυημένο χρόνο διεκπεραίωσης O(n logn).
- Αρκετά εύκολος για τον εντοπισμό σφαλμάτων (debug).

Από την παραπάνω εργασία κατανοούμε την αναγκαιότητα των αλγορίθμων ταξινόμησης και ειδικά του αλγόριθμου HeapSort όπου μπορεί να έχει πρακτική εφαρμογή σχεδόν παντού και βεβαίως σε πολύπλοκους και σοβαρότατους τομείς όπως αυτός της βιολογίας, του διαδικτύου, του ηλεκτρονικού εμπορίου και της βιομηχανικής παραγωγής. Η χρήση του αλγορίθμου HeapSort οδηγεί σε εξοικονόμηση χρόνου και χρήματος καθώς είναι σε θέση να διαχωριστεί μεγάλο όγκο δεδομένων και να παράγει μια πληθώρα πληροφοριών.

Βιβλιογραφία

1. **Thomas H. Cormen, Charles E. Ceiserson, Ronald C. Rivest, Clifford Stein.** *Εισαγωγή Στους Αλγορίθμους Τομος Ι.* s.l. : Πανεπιστημιακές Εκδόσεις Κρήτης, 2006.
2. **Μανωλοπουλος, Ιωάννης.** *Δομές Δεδομένων.* s.l. : Art Of Text.
3. **Κοίλιας, Χρήστος.** *Δομές Δεδομένων & Οργανώσεις Αρχείων.* s.l. : Νέων Τεχνολογίων, 2004.
4. **Dutton, Ronald D.** *Weak-heap sort.* s.l. : BIT, 1993.
5. **Παπουτσής, Ιωάννης.** *Εισαγωγή στις Δομές Δεδομένων και στους αλγόριθμους.*
6. **Sedgewick., R. Schaffer and R.** *The analysis of heapsort.* s.l. : Journal of Algorithms , 1993. Vol.15, No.1 .
7. **Μποζάνης, Π.** *Δομές Δοδομένων: Ταξινόμηση και Αναζήτηση με Java.* s.l. : Τζιόλα, 2003.
8. **T.H. Cormen. C.E. Leiserson, R.L. Rivest, and C. Stein.** *An Introduction to Algorithms.* s.l. : The MIT Press, 2003. 2 Edition.
9. **Φ. Αφράτη, και Γ. Παπαγεωργίου.** *Αλγόριθμοι: Μέθοδοι Σχεδίασης Και Ανάλυση Πολυπλοκότητας.* Αθήνα : Συμμετρία, 1993.
10. **Μποζάνης, Π.** *Αλγόριθμοι: Σχεδιασμός και Ανάλυση.* s.l. : Τζιόλα, 2003.
11. **Δουκάκης, Μ.** *Δομές Δεδομένων, Αλγόριθμοι.* s.l. : Ζυγός, 1998.
12. **Χατζηλυγερούδης, Ι.** *Δομές Δεδομένων, Εισαγωγή Στην Πληροφορική.* s.l. : Ελληνικό Ανοικτό Πανεπιστήμιο, 2000.
13. **A. V. Aho, J. E. Hopcroft, and J. D. Ullman.** *Data Structures And Algorithms.* s.l. : Addison – Wesley Publishing Company, 1985.
14. **Sedgewick, R.** *Algorithms.* s.l. : Addison – Wesley Publishing Company, 1984.
15. **Wilf, H. S.** *Algorithms And Complexity.* s.l. : Prentice Hall Inc, 1986.
16. **Mount, D.** *Algorithms.* s.l. : Lecture Notes, Dept. of Computer Science, University, 1998.

Ένθετο

Υπόλοιπα είδη σωρών

Διπλός Σωρός (double-ended heap, deap) [2, σελ. 273]

Ο διπλός σωρός (double-ended heap, deap) προτάθηκε από τον Carlsson (1987) και είναι μια δομή που εκτελεί τις ίδιες λειτουργιές που εκτελεί και ο σωρός ελαχίστων και μεγίστων και μάλιστα με την ίδια ποιοτική επίδοση. Η διαφορά τους είναι ότι η διαχείριση του διπλού σωρού είναι απλούστερη και επομένως η επίδοση είναι οριακά καλύτερη.

Ο διπλός σωρός είναι ένα σχεδόν πλήρες δυαδικό δένδρο με τα εξής χαρακτηριστικά:

- Η ρίζα είναι κενή
- Το αριστερό υποδένδρο είναι σωρός ελαχίστων
- Το δεξιό υποδένδρο είναι σωρός μεγίστων, και
- Αν το δεξιό υποδένδρο δεν είναι κενό, i είναι ένας κόμβος του αριστερού υποδένδρου και j είναι ο συμμετρικός κόμβος του στο δεξιό υποδένδρο, τότε το κλειδί του κόμβου i είναι μικρότερο ή ίσο από το κλειδί του κόμβου j. Αν ο κόμβος i δεν έχει συμμετρικό κόμβο στο δεξιό υποδένδρο, τότε ως j λαμβάνεται ο συμμετρικός του πατέρα του κόμβου i.

Αριστερίστικος σωρός (leftist heap) [2, σελ. 276]

Οι αριστερίστικοι σωροί είναι μια δυαδική δομή που είναι σχεδιασμένη ώστε να επιτυγχάνει τη συγχώνευση σε χρόνο O(logn) στη χειρότερη περίπτωση, ενώ οι πράξεις της εισαγωγής και της εξαγωγής στηρίζονται στη λειτουργία της συγχώνευσης, και επομένως διακρίνονται από την ίδια χρονική επίδοση στη χειρότερη περίπτωση. Ένας αριστερίστικος σωρός ελαχίστων (μεγίστων) είναι ένα αριστερίστικο δένδρο, όπου η τιμή κάθε κόμβου δεν είναι μεγαλύτερη (αντίστοιχα, μικρότερη) από τις τιμές των αντίστοιχων κόμβων παιδιών.

Δυωνυμικοί σωροί (binomial heaps) [2, σελ. 281]

Ένας δυωνυμικός σωρός αποτελείται από ένα σύνολο δυωνυμικών δένδρων όπου σε κάθε δυωνυμικό σωρό μπορεί να περιέχεται μόνο ένα δυωνυμικό δένδρο συγκεκριμένου ύψους. Έτσι το δυωνυμικό δένδρο ύψους 1,B_1, περιέχει έναν μόνο κόμβο, ενώ το δυωνυμικό δένδρο B_k ύψους k σχηματίζεται θέτοντας ένα δυωνυμικό δένδρο B_{k-1} ως δεξιό υποδένδρο ενός άλλου δυωνυμικού δένδρου B_{k-1}.

Οι δυωνυμικοί σωροί:

- Εκτελούν τις λειτουργιές της εισαγωγής, εξαγωγής και της συγχώνευσης σε χρόνο O(logn) στη χειρότερη περίπτωση, και
- Υλοποιούνται δυναμικά με τη βοήθεια δεικτών και όχι με πινάκες

Ωστόσο

- Δεν αναπαριστώνται ως μια ενιαία δομή, αλλά ως ένα σύνολο δομών, δηλαδή είναι ένα δάσος δένδρων, και
- Εκτελούν την εισαγωγή σε O(1) κατά μέσο όρο.

Σωροί Fibonacci [2, σελ. 285]

Οι σωροί Fibonacci είναι μια παραλλαγή των δυωνυμικών σωρών και χρησιμοποιούνται για τις γνωστές λειτουργιές της εισαγωγής, της εξαγωγής και της συγχώνευσης. Ωστόσο είναι σχεδιασμένοι για να παρέχουν αποτελεσματικά την δυνατότητα υλοποίησης δυο ακόμα πράξεων:

- Τη διαγραφή τυχόντος κόμβου, και
- Τη μείωση τυχόντος κλειδιού κατά μια ποσότητα

Όσον αφορά στην επίδοση των σωρών Fibonacci, αναφέρεται ότι ασυμπτωτικά έχουν καλύτερη επίδοση από τους δυωνυμικούς σωρούς.

Όπως οι δυωνυμικοί σωροί, έτσι και οι σωροί Fibonacci είναι μια συλλογή δένδρων που προσομοιάζουν με τα δυωνυμικά δένδρα. Ωστόσο, παρουσιάζουν τις εξής διαφορές:

- Τα δυωνυμικά δένδρα είναι διατεταγμένα, ενώ τα αντίστοιχα δένδρα Fibonacci δεν είναι διατεταγμένα
- Απαιτείται ένας επιπλέον δείκτης προς τη ρίζα του δένδρου Fibonacci που περιέχει τη μικρότερη τιμή
- Απαιτείται μια επιπλέον μεταβλητή, που δηλώνει το σύνολο των κόμβων όλων των δένδρων του σωρού Fibonacci

Με απλά λόγια μπορεί να λεχθεί ότι δομικά ένα δυωνυμικό δένδρο είναι και δένδρο Fibonacci, αλλά το αντίθετο δεν ισχύει. Τέλος από την άποψη των λειτουργιών (εισαγωγή, μείωση, διαγραφή) οι γνωστές διαδικασίες στους σωρούς Fibonacci υλοποιούνται με τεμπέλικο (lazy) τρόπο.

Υλοποίηση HeapSort

Εισαγωγή

Εδώ θα παρουσιαστεί ο αλγόριθμος HeapSort σε μια πρακτική εφαρμογή του πάνω σε ένα πρόγραμμα σε γλώσσα Java (τόσο σε γραφικό περιβάλλον όσο και σε web περιβάλλον) το οποίο έχει σχεδιαστεί ειδικά για την εργασία αυτή και είναι εμπνευσμένο από την πρακτική άσκηση. Επίσης θα παρουσιαστούν τα προγράμματα τα όποια χρησιμοποιηθήκαν, τα κυριότερα προβλήματα τα οποία παρουσιάστηκαν κατά την δημιουργία και εκτέλεση του κώδικα και οι παρατηρήσεις οι οποίες έγιναν.

Παρουσίαση Προγράμματος

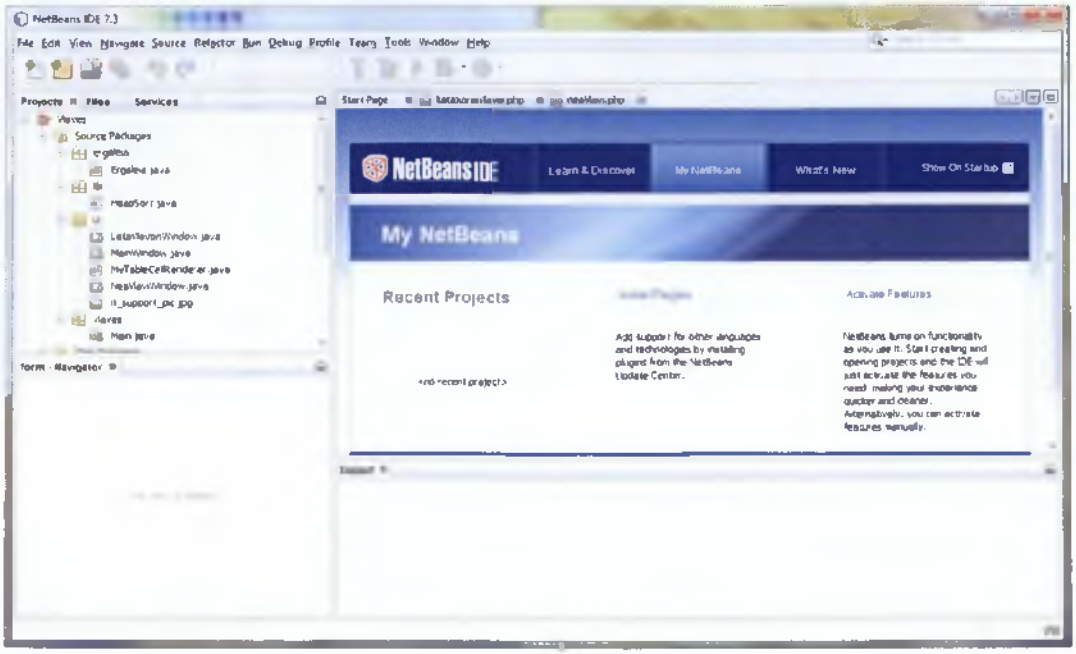
Προγράμματα

Τα προγράμματα τα οποία χρησιμοποιήσαμε για την υλοποίηση του προγράμματος ονομαστικά είναι:

1. Netbeans
2. Xampp Control Panel
3. MySQL
4. Apache HTTP

NetBeans

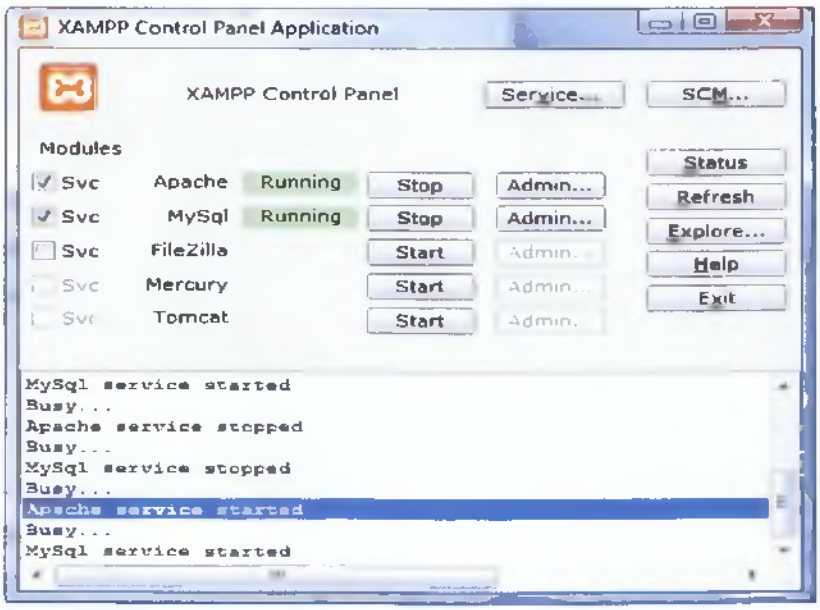
Το πρόγραμμα αυτό μπορούμε να το βρούμε δωρεάν στο Internet και μας είναι χρήσιμο για την δημιουργία, απασφαλμάτωση (Debug) και εκτέλεση του κώδικα Java για το γραφικό όσο και για το Web κομμάτι της εφαρμογής.



Εικόνα 17

Xampp Control Panel

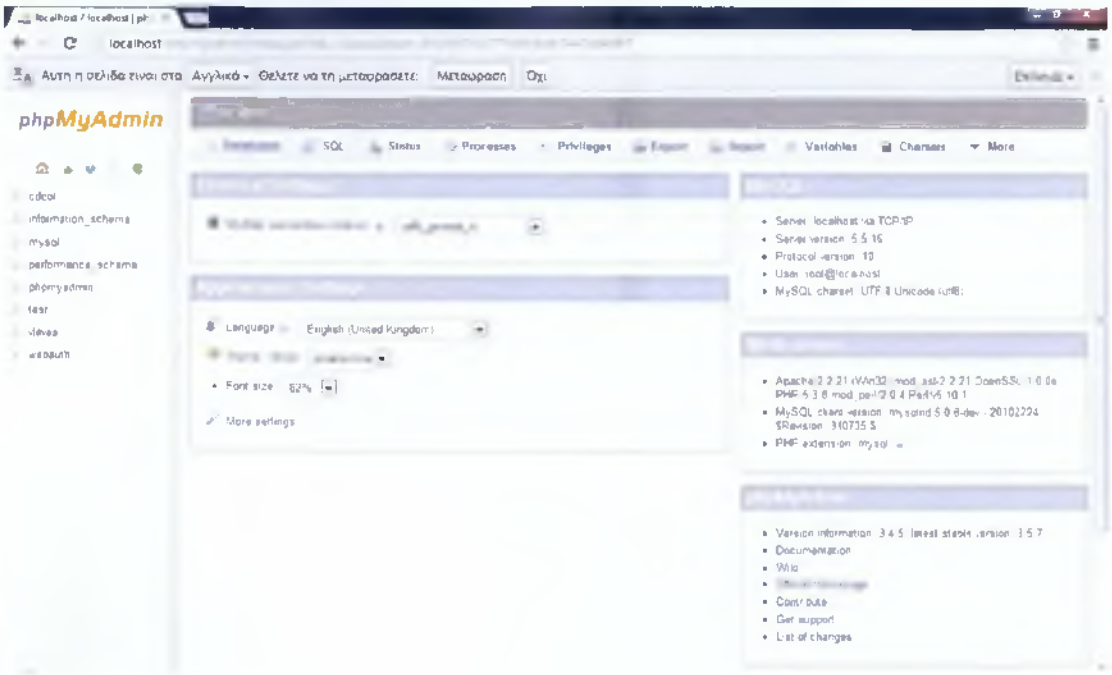
Το Xampp είναι ένα δωρεάν και open source cross-platform πακέτο web server, αποτελούμενο κυρίως από το Apache HTTP Server, MySQL βάση δεδομένων. Το εργαλείο αυτό σας επιτρέπει να ξεκινήσετε και να σταματήσετε τους servers που έχουν εγκατασταθεί ως μέρος του XAMPP.



Εικόνα 18

MySQL

Η MySQL είναι ένα σύστημα διαχείρισης σχεσιακών βάσεων δεδομένων που μετρά περισσότερες από 11 εκατομμύρια εγκαταστάσεις. Έλαβε το όνομά της από την κόρη του Μόντυ Βιντένιους, τη Μάι (αγγλ. My). Το πρόγραμμα τρέχει έναν εξυπηρετητή (server) παρέχοντας πρόσβαση πολλών χρηστών σε ένα σύνολο βάσεων δεδομένων. Μπορούμε να την βρούμε στο Internet δωρεάν.



Εικόνα 19

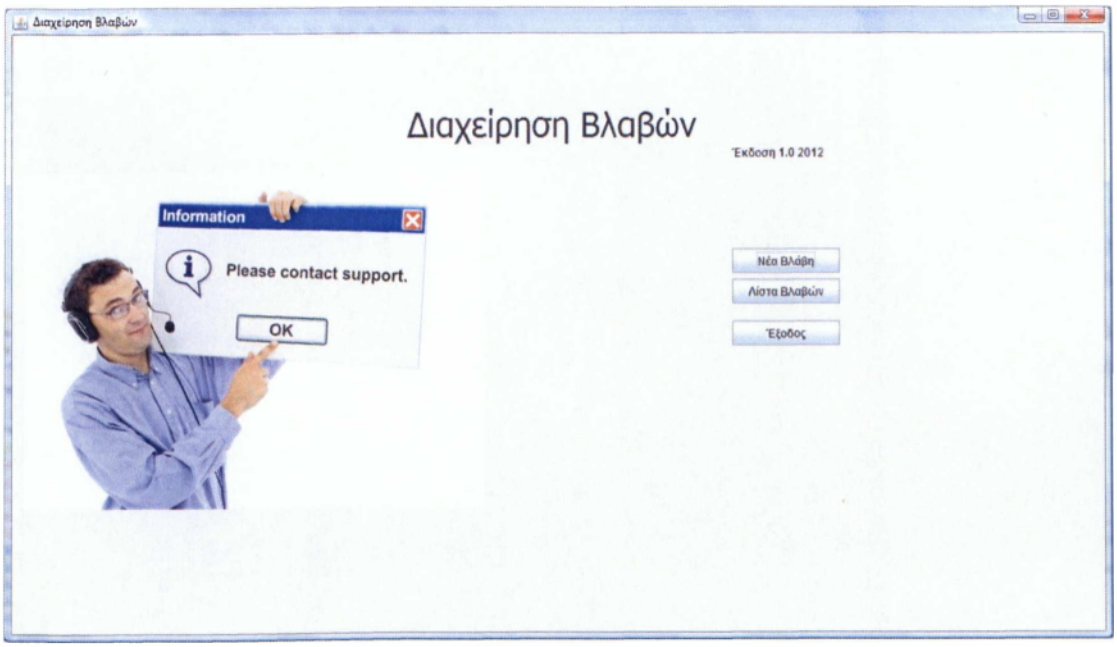
Apache HTTP

Η πρώτη του έκδοση, γνωστή ως NCSA HTTPd, δημιουργήθηκε από τον Robert McCool και κυκλοφόρησε το 1993. Θεωρείται ότι έπαιξε σημαντικό ρόλο στην αρχική επέκταση του παγκόσμιου ιστού. Ο Apache HTTP γνωστός και απλά σαν Apache είναι ένας εξυπηρετητής του παγκόσμιου ιστού (web). Όποτε ένας χρήστης επισκέπτεται ένα ιστότοπο το πρόγραμμα πλοήγησης (browser) επικοινωνεί με έναν διακομιστή (server) μέσω του πρωτοκόλλου HTTP, ο οποίος παράγει τις ιστοσελίδες και τις αποστέλλει στο πρόγραμμα πλοήγησης. Ο Apache είναι ένας από τους δημοφιλέστερους, εν μέρει γιατί λειτουργεί σε διάφορες πλατφόρμες όπως τα Windows, το Linux, το Unix και το Mac OS X. Συντηρείται τώρα από μια κοινότητα ανοικτού κώδικα με επιτήρηση από το Ίδρυμα Λογισμικού Apache (Apache Software Foundation). Ο Apache χρησιμοποιείται και σε τοπικά δίκτυα σαν διακομιστής συνεργαζόμενος με συστήματα διαχείρισης Βάσης Δεδομένων π.χ. Oracle, MySQL.

Υλοποίηση

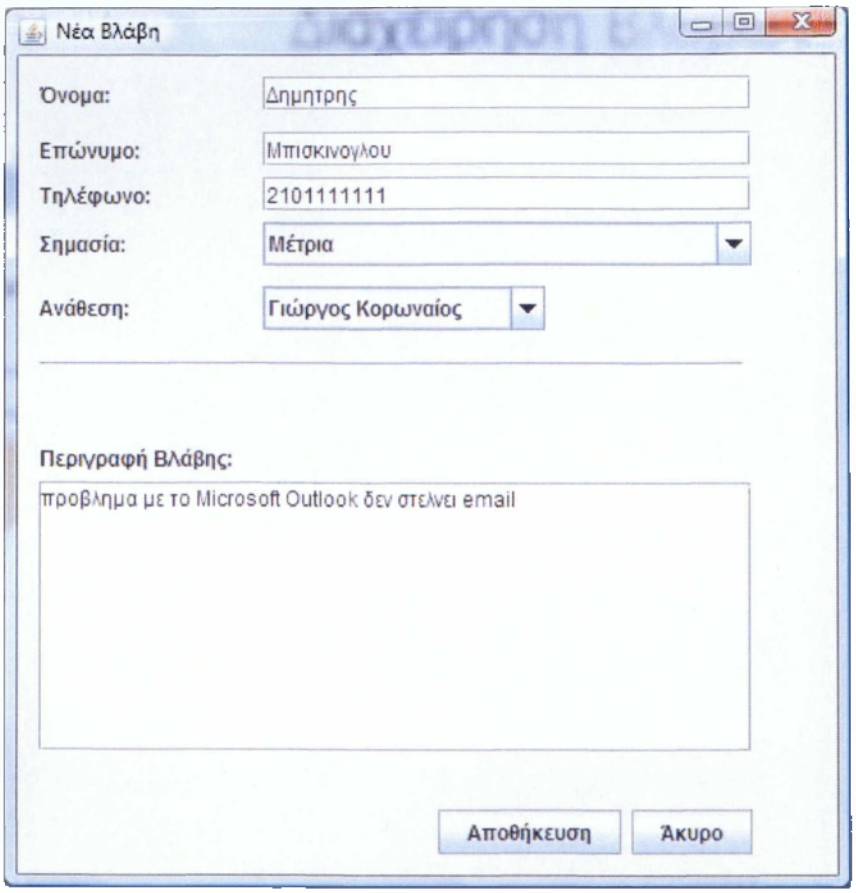
Η υλοποίηση του προγράμματος έγινε στο περιβάλλον του NetBeans και αποτελείται από κώδικα java και PHP. Το πρόγραμμα αφορά ένα πρόγραμμα καταγραφής βλαβών σε μια βάση δεδομένων. Οι βλάβες μπορούν να καταχωρηθούν από τον αντίστοιχο Administrator (γραφικό περιβάλλον Java) ή από περιβάλλον Web (PHP) που μπορεί να τις καταχωρίσει ο ίδιος ο χρήστης. Οι βλάβες μπορούν να ανατεθούν στον ανάλογο μηχανικό και υπάρχει η καταγραφή, χρόνου και ημερομηνίας (συστήματος PC), κατά την καταχώριση και διεκπεραίωση των βλαβών. Τέλος οι βλάβες αρχειοθετούνται ανάλογα με την σοβαρότητα τους κατά φθίνουσα σειρά με τον αλγόριθμο HeapSort.

Στην παρακάτω εικόνα μπορούμε να δούμε το κεντρικό παράθυρο του προγράμματος από το οποίο οι διαχειριστές του συστήματος καταχωρούν και διαχωρίζονται τις βλάβες. Από εδώ μπορούν να καταχωρίσουν μια καινούρια βλάβη ή να δουν τον πίνακα βλαβών και να κλείσουν κάποια βλάβη αφού θα έχει ολοκληρωθεί.



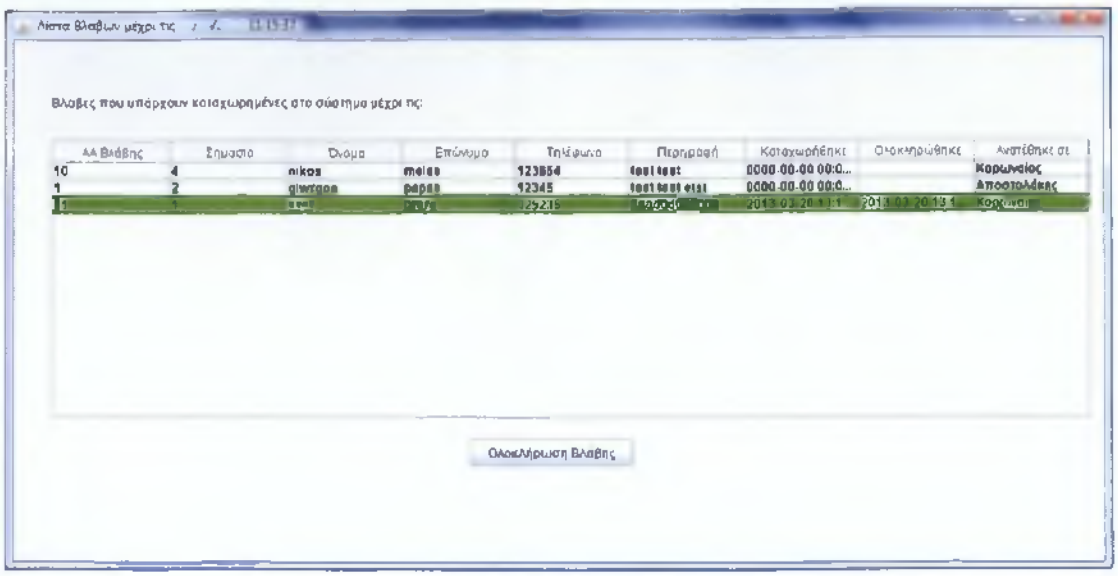
Εικόνα 20

Στην παρακάτω εικόνα είναι το παράθυρο από το οποίο καταχωρούνται οι βλάβες από τον αρμόδιο διαχειριστεί. Δηλώνει το όνομα, το επώνυμο αυτού που έχει την βλάβη, ένα τηλέφωνο επικοινωνίας, την σοβαρότητα που έχει η βλάβη και μια σύντομη περιγραφή της. Τέλος την αναθέτει στον ανάλογο μηχανικό για την επίλυση της



Εικόνα 21

Σε αυτή την εικόνα βλέπουμε την λίστα βλαβών ταξινομημένη από τον αλγόριθμο HeapSort. Επίσης βλέπουμε την ημερομηνία και ώρα καταχώρισής της. Τέλος ο διαχειριστής μπορεί να δηλώσει μια βλάβη ολοκληρωμένη και αυτή αμέσως θα χρωματιστεί και θα καταχωρηθεί στη βάση δεδομένων με την ημερομηνία του συστήματος.



Εικόνα 22

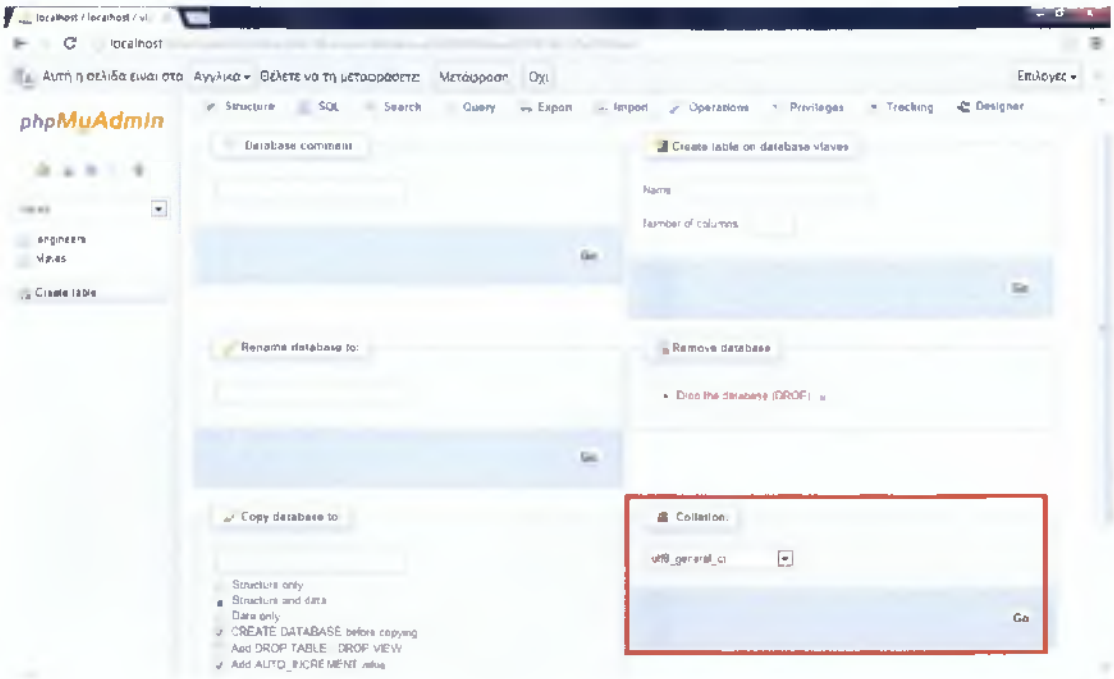
Τέλος στην επόμενη εικόνα βλέπουμε το web περιβάλλον του προγράμματος όπου μπορεί ο χρήστης να καταχωρήσει μια βλάβη ή να την διαγράψει, αν έχει γίνει κάποιο λάθος, με το ID της βλάβης που θα λάβει.

Εικόνα 23

Προβλήματα

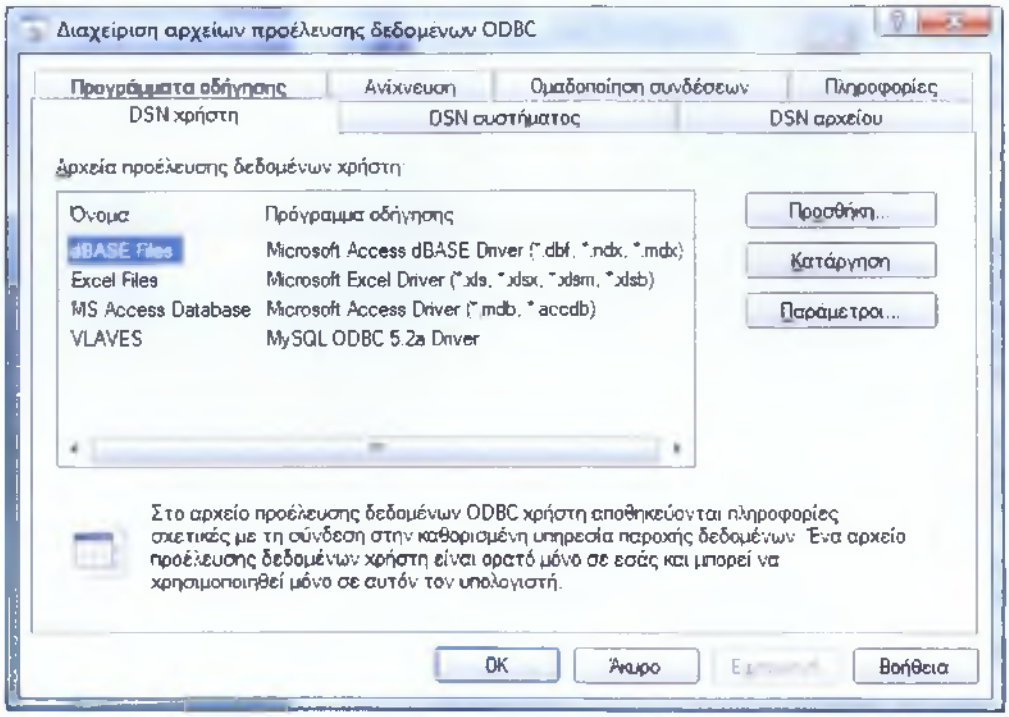
Κατά την υλοποίηση ολοκλήρου του προγράμματος (κώδικας και βάση δεδομένων) παρουσιάστηκαν κάποια προβλήματα. Τα ποιο κύρια είναι τα εξής:

- Παρουσιάστηκε πρόβλημα με την αναγνώριση των ελληνικών χαρακτήρων μεταξύ της βάσης δεδομένων και του Web περιβάλλον του προγράμματος.
    1. Αυτό λύθηκε με την διόρθωση κομμάτι κώδικα και βάζοντας την κωδικοποίηση UTF8 και
    2. Κατά την δημιουργία της βάσης στην Βάση δεδομένων ορίσαμε την κωδικοποίηση UTF8_general.

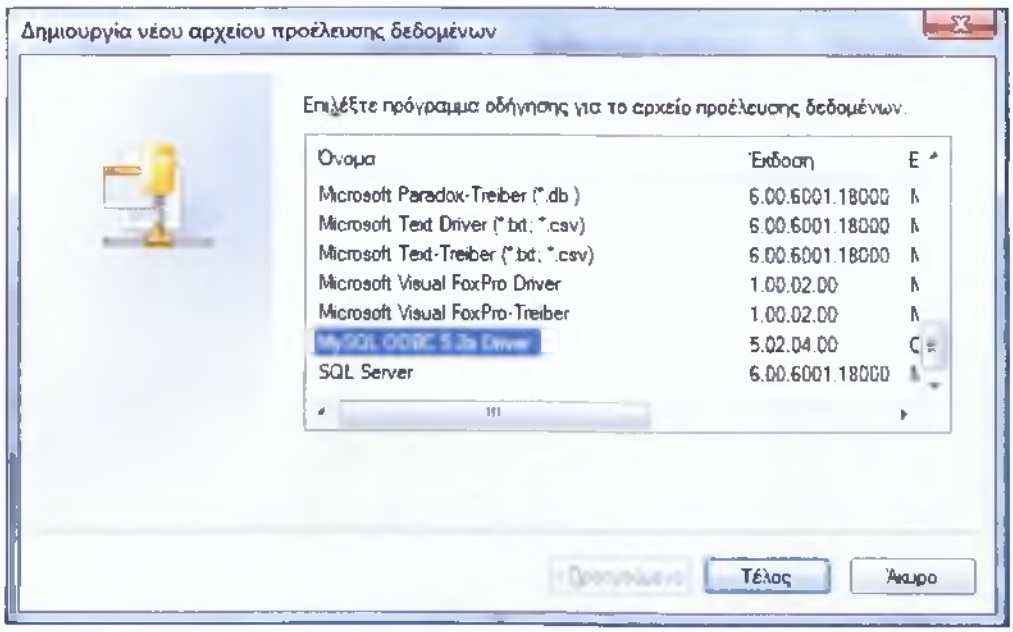


Εικόνα 24

- Παρουσιάστηκε πρόβλημα στην επικοινωνία της Βάσης Δεδομένων με το πρόγραμμα. Αυτό λύθηκε με την εγκατάσταση του Odbc connector driver ο ποιος είναι υπεύθυνος για την επικοινωνία της βάσης με το λειτουργικό σύστημα και κατ' επέκταση με το πρόγραμμα μας. Η ρύθμιση του Odbc Driver μπορεί να φανεί από τις παρακάτω εικόνες.



Εικόνα 25



Εικόνα 26

Τέλος ρυθμίζουμε και το κομμάτι του κώδικα για να βλέπει και να επιδεινώνει με τον Odbc driver.

```
try {
        Connection con =
DriverManager.getConnection("jdbc:odbc:VLAVES;USER=root;charset=utf8");
        //Initialize the statement to be used, specify if rows are scrollable
        Statement stmt = con.createStatement();
        //ResultSet will hold the data retrieved
        ResultSet rs = stmt.executeQuery("SELECT onoma, eponimo FROM engineers");
        //Display the results
        while (rs.next()) {
          String onoma = rs.getString("onoma");
          String eponimo = rs.getString("eponimo");

        }
        con.close();
      } catch (Exception e) {
        System.out.println(e);
      }
```

- Παρατηρήθηκαν προβλήματα φορητότητας του προγράμματος. Δηλαδή παρατηρήθηκαν διαφορές στην παρουσίαση του Web προγράμματος με διαφορετικούς browsers και προβλήματα επικοινωνίας της βάσης δεδομένων με τον κώδικα σε διάφορα λειτουργικά συστήματα (Windows 7 <-> Windows Vista). Όλα τα παραπάνω διορθώθηκαν με τις ανάλογες παρεμβάσεις στον κώδικα για την ορθή του λειτουργία.
- Επίσης ο Apache HTTP Server λειτουργεί στην πόρτα 80 του τοπικού συστήματος και ως αποτέλεσμα θα πρέπει η πόρτα αυτή να είναι ελεύθερη και να μην λειτουργούν άλλα προγράμματα εκεί π.χ. Skype γιατί έχει ως συνέπια να μην μπορεί να ξεκινήσει ο Apache HTTP Server.