# ΑΝΑΦΟΡΑ PROJECT DATA STRUCTURES 2018-2019

- **(Όλοι οι κώδικες είναι σε γλώσσα προγραμματισμού C)**
- **(Όλα τα ερωτήματα είναι υλοποιημένα και δουλεύουν σωστά όλα εκτός από κάποιους αλγόριθμους εύρεσης, αναφέρουμε αναλυτικά παρακάτω)**

## PART I: "Sorting and Searching Algorithms"

(1)

Χρησιμοποιώντας το ίδιο data set για τους αλγορίθμους **Bubble Sort**, **Insertion Sort** καθώς και **Selection Sort**, εξάγουμε το συμπέρασμα ότι και οι τρεις κάνουν ταξινόμηση σε χρόνο περίπου $T(n)=O(n^2)$ . Συγκεκριμένα στον Bubble Sort γίνονται πιο πολλές εναλλαγές μεταξύ των στοιχείων και τελικά ο πίνακας διαπερνάται περισσότερες φορές με αποτέλεσμα ο αλγόριθμος να είναι μη αποδοτικός. Μεταξύ των τριών, ο Insertion Sort είναι με σημαντική διαφορά ο πιο γρήγορος αφού χρειάζεται περίπου το μισό χρόνο από τον Selection Sort.

| n | Bubble Sort(sec) | Insertion Sort(sec) | Selection Sort(sec) |
|---|---|---|---|
| 1.000 | 0,005 | 0,004 | 0,005 |
| 10.000 | 0,395 | 0,248 | 0,234 |
| 100.000 | 38.672 | 13,9800 | 22,4336 |

Ο **Bubble Sort** ξεκινάει από την αρχή του πίνακα και κάνει bubbling up το μεγαλύτερο στοιχείο, μεταξύ των δυο που συγκρίθηκαν, εκτελώντας στην συνέχεια swap. Αυτό γίνεται σταδιακά μέχρι όλα τα στοιχεία να τοποθετηθούν στον πίνακα ταξινομημένα με το μεγαλύτερο να βρίσκεται στο τέλος. Χρειάζονται το πολύ n-1 περάσματα (worst case) με μέγιστες n συγκρίσεις.

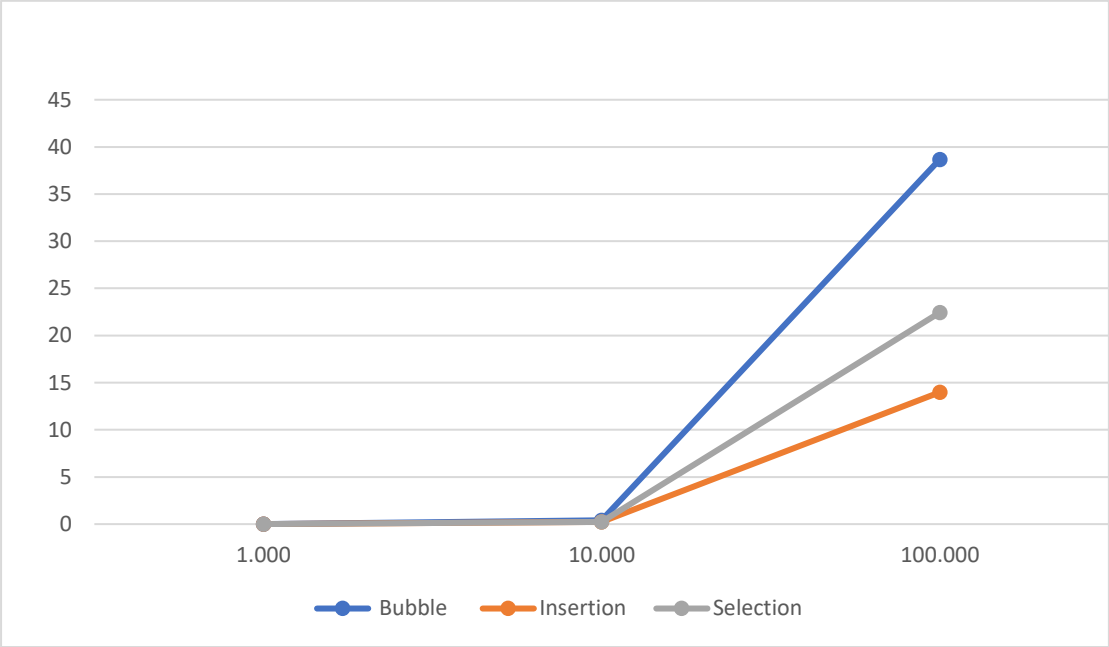
Ο **Insertion Sort** χρησιμοποιεί γραμμική αναζήτηση, θεωρώντας το πρώτο στοιχείο ως ταξινομημένο, το συγκρίνει κάθε φορά με τα δεξιότερα στοιχεία του πίνακα. Αν κάποιο από αυτά είναι μεγαλύτερο, τότε τοποθετείται στο τέλος του ταξινομημένου κομματιού του πίνακα ή αλλιώς στην σωστή θέση (αφού εκτελούνται shift των μεγαλύτερων στοιχείων). Οπότε θα χρειαστούν το πολύ n-1 συγκρίσεις (best case) ή n(n-1)/2 συγκρίσεις (worst case).

Ο **Selection Sort** ξεκινάει από την αρχή του πίνακα και επιλέγει το μικρότερο στοιχείο, το οποίο σε κάθε πέρασμα τοποθετείται στην αρχή του πίνακα, μέχρι να ταξινομηθεί πλήρως. Να σημειωθεί πως αυτός ο αλγόριθμος έχει $\Omega(n^2)$ στην καλύτερη περίπτωση.

**Συμπερασματικά και για τους τρεις αλγορίθμους ισχύει ότι στην καλύτερη περίπτωση η πολυπλοκότητα θα είναι O(n), ενώ στην μέση και χειρότερη οποίες**

*Επιμέλεια:*
*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

**φράσσουν άνω τους αλγόριθμους, παρατηρούνται μεγάλες διαφορές στους χρόνους εκτέλεσης.**




(2)

Χρησιμοποιώντας το ίδιο data set για τους αλγορίθμους **Merge Sort** και **Quick Sort**, εξάγουμε το συμπέρασμα ότι και οι δύο κάνουν ταξινόμηση σε χρόνο περίπου T(n)=O(nlogn). Πιο συγκεκριμένα, ο Merge Sort είναι λιγότερο αποδοτικός από τον Quick Sort ο οποίος δεν εκτελεί μη αναγκαίες εναλλαγές στοιχείων. Ο Merge Sort ανεξάρτητα από το αν είναι ταξινομημένα ή όχι τα στοιχεία τα περνάει σε διαφορετικό πίνακα και τα γράφει ξανά στον αρχικό και άρα είναι λογικό να χρειάζεται περισσότερο χρόνο.

| N | Merge Sort(sec) | Quick Sort(sec) |
|---|---|---|
| 1.000 | 0,000 | 0,001 |
| 10.000 | 0,003 | 0,002 |
| 100.000 | 0,135 | 0,019 |

Ο **Merge Sort** χρησιμοποιεί την μέθοδο «διαίρει και βασίλευε», διότι χωρίζει τα στοιχεία στην μέση όσες φορές χρειάζεται και τα ταξινομεί ξεχωριστά, συγχωνεύοντας (merging) τις λύσεις στο τέλος. Χρειάζονται το πολύ n-1 συγκρίσεις (worst case) και για κάθε βήμα πληρώνουμε O(n) συγκρίσεις .

Ο **Quick Sort** χρησιμοποιεί την μέθοδο «διαίρει και βασίλευε», διότι χωρίζει τα στοιχεία σε δύο μέρη. Ορίζει έναν pivot στην μέση και εκτελεί συγκρίσεις με αυτόν

*Επιμέλεια:*
*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

βάζοντας τις μικρότερες τιμές αριστερά και τις μεγαλύτερες δεξιά. Δημιουργείται καινούργιο pivot αναδρομικά (όσες φορές χρειαστεί) για να προκύψουν ταξινομημένοι πίνακες. Χρειάζονται λοιπόν το πολύ n-1 συγκρίσεις (worst case) και για κάθε βήμα ισχύει ότι πληρώνουμε O(n) συγκρίσεις.

**Συμπερασματικά ισχύει και για τους δύο αλγορίθμους ότι στην καλύτερη και μέση περίπτωση η πολυπλοκότητα θα είναι O(nlogn) με τη σταθερά για τον Merge Sort να είναι πολύ μεγαλύτερη από την σταθερά του Quick Sort. Στη χειρότερη περίπτωση το άνω φράγμα για τον χρόνο είναι O(nlogn) για τον Merge Sort και $O(n^2)$ για τον Quick Sort. Θεωρητικά στην μέση και την καλύτερη περίπτωση ο Quick Sort αποδίδει πολύ καλύτερα από τους περισσότερους αλγορίθμους ενώ στην χειρότερη είναι πιο αργός από τον Merge Sort, κάτι που δεν επαληθεύεται πειραματικά αφού ο Quick Sort χρειάζεται λιγότερο χρόνο.**

(3)

Συγκρίσεις μεταξύ των αλγορίθμων ταξινόμησης έχουν γίνει και παραπάνω. Επιλέξαμε τους καλύτερους και τους συγκρίναμε με τον Heap Sort για να βρούμε τον ποιο αποδοτικό μεταξύ των 6.

Χρησιμοποιώντας το ίδιο data set για τους αλγορίθμους **Bubble Sort**, **Insertion Sort**, **Selection Sort, Merge Sort**, **Quick Sort** καθώς και **Heap Sort**, εξάγουμε το συμπέρασμα ότι ο τελευταίος έχει πολυπλοκότητα O(nlogn) ,που είναι γενικά καλύτερη από τον Quick Sort**.** Επιπλέον, ο Heap Sort δεν χρειάζεται όσο χώρο αποθήκευσης όσος ο Quick Sort. Γενικά σε time critical applications προτιμάται ο Quick sort αφού με τον Heap Sort ακόμα και αν τα δεδομένα είναι ταξινομημένα θα εκτελεστούν swap για όλα τα στοιχεία.

| N | Heap Sort(sec) |
|---|---|
| 1.000 | 0,000 |
| 10.000 | 0,003 |
| 100.000 | 0,038 |

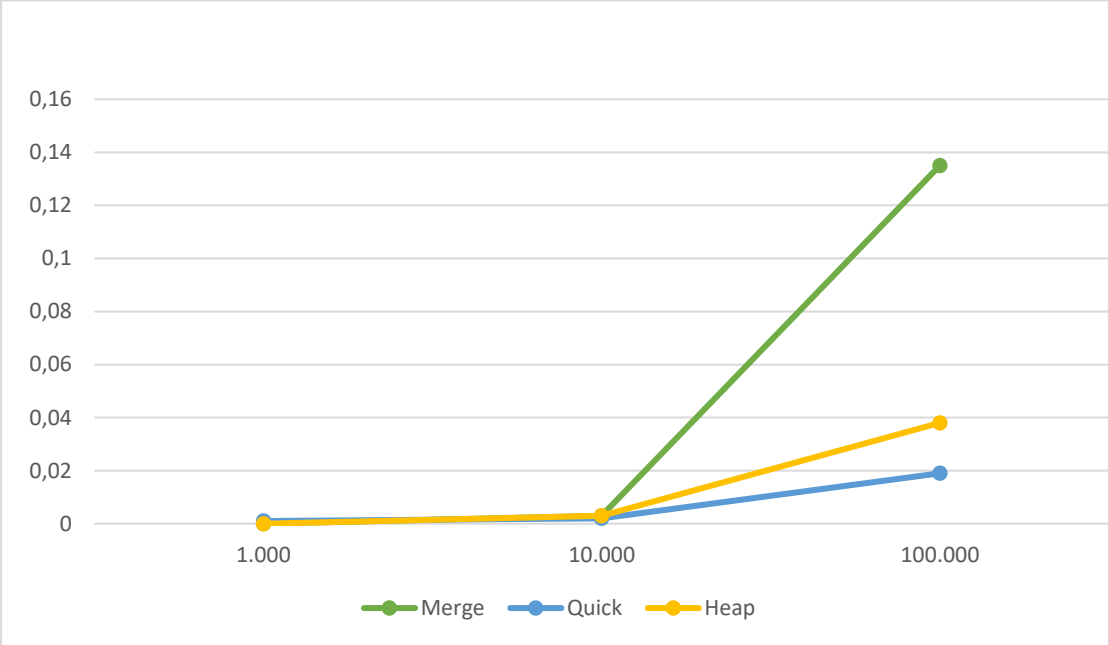
Ο **Heap Sort** ξεκινάει χρησιμοποιώντας δυναμικά μεγάλο χώρο μνήμης για να δημιουργηθεί ένα balanced, left -justified δυαδικό δέντρο, που κανένας κόμβος δεν έχει μεγαλύτερη τιμή από τον πατέρα του. Στην συνέχεια για την ταξινόμηση εκτελείται διαγραφή της ρίζας και μεταφορά της στο τέλος της λίστας. Στον πλέον άδειο κόμβο τοποθετείται το τελευταίο στοιχείο(δηλαδή το πιο δεξί φύλλο) και

***Επιμέλεια:***

*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

μέσω διαδικασιών(heapify up/down) αποκαθίσταται η δομή του δέντρου. Η παραπάνω διαδικασία εκτελείται μέχρι να δημιουργηθεί μια ταξινομημένη λίστα. Γίνονται O(n) συγκρίσεις στη φάση δόμησης. Στην φάση διαλογής το μέγιστο στοιχείο απομακρύνεται από την ρίζα το πολύ n φορές με αποτέλεσμα το μονοπάτι απομάκρυνσης να έχει μήκος O(logn) και σε κάθε κόμβο του θα γίνονται O(1) συγκρίσεις. Ως αποτέλεσμα κάθε επανάληψη στοιχίζει O(logn) συγκρίσεις και άρα το συνολικό κόστος είναι O(nlogn).

**Συμπερασματικά ισχύει για τον Heap Sort ότι σε κάθε περίπτωση η πολυπλοκότητα του θα είναι O(nlogn). Συνεπώς ισχύει ότι στην καλύτερη περίπτωση αποδοτικότεροι αλγόριθμοι είναι οι Merge Sort, Quick Sort και Heap Sort με πολυπλοκότητα O(nlogn) και από αυτούς υπερισχύει ο Quick Sort. Στην μέση περίπτωση αποδοτικότεροι είναι οι Merge Sort, Quick Sort, Heap Sort με πολυπλοκότητα O(nlogn) και από αυτούς υπερισχύει ο Quick Sort. Ενώ στην χειρότερη περίπτωση αποδοτικότεροι είναι οι Merge Sort, Heap Sort με πολυπλοκότητα O(nlogn) και από αυτούς υπερισχύει ο Heap Sort.**




(4)

Χρησιμοποιώντας το ίδιο data set για τη **Γραμμική Αναζήτηση, Δυαδική Αναζήτηση και Αναζήτηση με Παρεμβολή,** εξάγουμε το συμπέρασμα ότι και οι τρεις κάνουν αναζήτηση στα n ταξινομημένα πλέον στοιχεία. Πιο συγκεκριμένα η Γραμμική Αναζήτηση είναι η λιγότερο αποδοτική εκ των τριών, ενώ η Αναζήτηση Παρεμβολής η πιο γρήγορη με την Δυαδική Αναζήτηση να βρίσκεται ενδιάμεσα των παραπάνω.

***Επιμέλεια:***

*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

# ΑΝΑΦΟΡΑ PROJECT DATA STRUCTURES 2018-2019

Στον παρακάτω πίνακα γίνεται αναζήτηση για το στοιχείο 1289 του data set μας. Παραθέτουμε τον αριθμό των συγκρίσεων για 1 από τα στοιχεία που δοκιμάσαμε.

| | Γραμμική Αναζήτηση | Δυαδική Αναζήτηση | Αναζήτηση Παρεμβολής |
|---|---|---|---|
| συγκρίσεις | 268 | 16 | 0 |
| χρόνος | 0 | 0 | 0 |

Η **Γραμμική Αναζήτηση** ξεκινάει από την αρχή του πίνακα και ψάχνει/σαρώνει κάθε στοιχείο με τη σειρά από το μικρότερο προς το μεγαλύτερο, μέχρι να γίνει κάποιο match. Συνεπώς παρατηρούμε ότι εκτελεί n/2 συγκρίσεις στην μέση περίπτωση και ότι ο βρόχος while εκτελείται (n+1)/2 φορές.

Η **Δυαδική Αναζήτηση** ξεκινάει με elimination of the impossible region και συγκρίνει το στοιχείο με το μεσαίο για να βρει σε ποιο κομμάτι βρίσκεται και μικραίνει συνεχώς το διάστημα μέχρι να το εντοπίσει. Αν δεν το βρει τότε δεν θα υπάρχει στην λίστα. Άρα, ο βρόχος while θα εκτελεστεί logn φορές στην χειρότερη περίπτωση.

Η **Αναζήτηση Παρεμβολής** ξεκινάει ορίζοντας μεταβλητές που δείχνουν την αρχή και το τέλος του πίνακα(αρχικά) και με βήματα μήκους √n τον σαρώνουν όλον μέχρι να βρεθεί το κομμάτι του περιέχει το ζητούμενο στοιχείο. Αν δεν βρεθεί τότε δεν θα υπάρχει στην λίστα.

Η κατανομή του data set επηρεάζει την απόδοση όλων των αλγορίθμων, αφού το στοιχείο που θα ψάχνουμε κάθε φορά θα βρίσκεται σε διαφορετική θέση άρα θα χρειαζόμαστε διαφορετικό χρόνο εύρεσης. Πιο συγκεκριμένα για την Γραμμική Αναζήτηση ισχύει ότι αν το στοιχείο μας βρίσκεται προς το τέλος θα χρειαστεί πολύ περισσότερος χρόνος μέχρι να διαπεραστούν όλα τα προηγούμενα στοιχεία, και άρα και πολύ περισσότερες συγκρίσεις. Στη συνέχεια για την Δυαδική Αναζήτηση ισχύει ότι θα χρειαστεί πολλές παραπάνω διαμερίσεις του διαστήματος, δηλαδή πολύ παραπάνω χρόνο και κατά επέκταση συγκρίσεις, αν το στοιχείο που ψάχνουμε βρίσκεται ανάμεσα σε πολύ κοντινούς αριθμούς αφού η εύρεση ενός στοιχείου είναι πιο δύσκολη σε πυκνό διάστημα. Τέλος για την Αναζήτηση Παρεμβολής ισχύει ότι και για την Δυαδική Αναζήτηση με την μόνη διαφορά ότι το ζητούμενο στοιχείο εντοπίζεται πιο εύκολα αφού κάνουμε μεγαλύτερο βήμα αποκλείοντας περισσότερη γκάμα άσχετων στοιχείων, δηλαδή δεν μας επηρεάζει τόσο το πόσο πυκνό είναι το διάστημά μας.

**Συμπερασματικά η Γραμμική Αναζήτηση έχει πολυπλοκότητα O(n), η Δυαδική Αναζήτηση έχει O(logn) και η Αναζήτηση με Παρεμβολή έχει O(loglogn) ενώ στην χειρότερη περίπτωση η πολυπλοκότητα της είναι O(n) . Συνεπώς παρατηρούμε ότι**

*Επιμέλεια:*
*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

**στην καλύτερη και μέση περίπτωση αποδοτικότερη είναι η Αναζήτηση με Παρεμβολή με πολυπλοκότητα O(loglogn). Για την χειρότερη περίπτωση ωστόσο αποδοτικότερη είναι η Δυαδική Αναζήτηση με πολυπλοκότητα O(logn).**

Να τονίσουμε πως και οι 3 παραπάνω αλγόριθμοι βρίσκουν το στοιχείο που δίνουμε στην είσοδο αν αυτό φυσικά υπάρχει. Αυτό που δεν δουλεύει σωστά είναι ο υπολογισμός του χρόνου εκτέλεσης που χρειάζεται ο κάθε αλγόριθμος. Ωστόσο από τον αριθμό των συγκρίσεων που γίνονται μπορούμε να συμπεράνουμε πως ο **Binary** είναι ο μόνος αποτελεσματικός αλγόριθμος και ότι ο **Interpolation** για ένα μη ομαλό δείγμα(σαν το δικό μας στην προκειμένη περίπτωση) δίνει περίπου ίδιες συγκρίσεις με τον **Linear**. Τέλος, οι counter που μετράνε τις συγκρίσεις για τον Binary και τον Linear δουλεύουν κανονικά.

(5)

Χρησιμοποιώντας το ίδιο dataset για την **Δυική Αναζήτηση Παρεμβολής (BIS)** και την **Βελτιωμένη Παραλλαγής του BIS,** εξάγουμε το συμπέρασμα ότι και οι δύο κάνουν αναζήτηση στα n ταξινομημένα στοιχεία. Πιο συγκεκριμένα η Δυική Αναζήτηση Παρεμβολής (BIS) είναι λιγότερο αποδοτική στην χειρότερη περίπτωση, βρίσκοντας το στοιχείο σε λιγότερο χρόνο από τον αλγόριθμο Βελτιωμένου BIS. Επίσης στη μέση και καλύτερη περίπτωση εκτελούν την αναζήτηση στον ίδιο χρόνο.

Η **Δυική Αναζήτηση Παρεμβολής** ξεκινάει και συγκρίνει το ζητούμενο στοιχείο με το μεσαίο για να βρει σε ποιο κομμάτι του πίνακα βρίσκεται αλλάζοντας κάθε φορά το μέγεθος του πίνακα. Αν δεν το βρει τότε δεν θα υπάρχει στην λίστα. Επίσης θεωρούμε ότι για σχετικά λίγο αριθμό στοιχείων εκτελούμε γραμμική αναζήτηση, χωρίς αυτό να επηρεάζει την ασυμπτωτική συμπεριφορά του αλγορίθμου. Η μέση περίπτωση έχει χρόνο 2,4*loglogn+O(1), δηλαδή O(loglogn), αφού όταν η λίστα μας γίνει αρκετά μικρή εκτελούμε γραμμική αναζήτηση με άλματα μεγέθους √n για να προσδιοριστεί το υποδιάστημα που περιέχει το ζητούμενο στοιχείο. Ενώ στην χειρότερη O(√n) αφού εκτελούνται το πολύ √n συγκρίσεις.

Η **Βελτιωμένη Παραλλαγή του BIS** εκτελεί ακριβώς την ίδια διαδικασία με την μόνη διαφορά ότι το i αντί να αυξάνεται γραμμικά (i=i+1 όπως στην Δυική Αναζήτηση Παρεμβολής) αυξάνεται εκθετικά (i=i*2), εκτελώντας άλματα μεγέθους (2^i)*√n.

Αντίστοιχα, για σχετικά λίγο αριθμό στοιχείων εκτελούμε γραμμική αναζήτηση, χωρίς αυτό να επηρεάζει την ασυμπτωτική συμπεριφορά του αλγορίθμου. Ο βελτιωμένος BIS χρειάζεται για κάθε επανάληψη της while χρόνο O(logn) στην

*Επιμέλεια:*
*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

χειρότερη περίπτωση λόγω συγκρίσεων με τα βήματα να είναι εκθετικά. Ο χρόνος μέσης περίπτωσης δεν αλλάζει σε σχέση με την Δυική Αναζήτηση Παρεμβολής (BIS) αφού παραμένει O(loglogn), υπό την συνθήκη ότι όλα τα στοιχεία επιλέχθηκαν ισοπίθανα.

**Συμπερασματικά ισχύει για την Δυική Αναζήτηση Παρεμβολής (BIS) ότι στη καλύτερη και μέση περίπτωση η πολυπλοκότητα της είναι O(loglogn), ενώ στην χειρότερη περίπτωση η πολυπλοκότητα της είναι O(√n). Για την Βελτιωμένη Παραλλαγή του BIS ισχύει ότι στην καλύτερη και μέση περίπτωση η πολυπλοκότητα της είναι O(loglogn) ενώ στην χειρότερη περίπτωση η πολυπλοκότητα της είναι O(logn). Θεωρητικά στην καλύτερη και μέση περίπτωση οι αναζητήσεις είναι εξίσου αποδοτικές. Για την χειρότερη περίπτωση ωστόσο αποδοτικότερη είναι η Βελτιωμένη Παραλλαγή του BIS, με πολυπλοκότητα O(logn) κάτι το οποίο επιβεβαιώνεται και από τα πειραματικά μας αποτελέσματα.**

Να τονίσουμε ξανά πως το απλό BIS δουλεύει κανονικά για όποιο στοιχείο δώσει ο χρήστης αρκεί αυτό να ανήκει στο αρχείο αναζήτησης. Ο βελτιωμένος BIS δουλεύει για τους περισσότερους αριθμούς που δώσαμε ως είσοδο και εκτελώντας κάποιες μετρήσεις παρατηρήσαμε πως η βελτιωμένη εκδοχή του αλγόριθμου χρειάζεται περίπου τον μισό χρόνο εκτέλεσης κάτι που επιβεβαιώνεται και από την θεωρία. Οι μετρητές δουλεύουν κανονικά και για τους δυο αλγόριθμους.

***Επιμέλεια:***
*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

# ΑΝΑΦΟΡΑ PROJECT DATA STRUCTURES 2018-2019

| ΥΛΟΠΟΙΗΣΗ | ΚΑΛΥΤΕΡΗ | ΜΕΣΗ | ΧΕΙΡΟΤΕΡΗ |
|---|---|---|---|
| **Bubble Sort** | O(n) | $O(n^2)$ | $O(n^2)$ |
| **Insertion Sort** | O(n) | $O(n^2)$ | $O(n^2)$ |
| **Selection Sort** | O(n) | $O(n^2)$ | $O(n^2)$ |
| **Merge Sort** | O(nlogn) | O(nlogn) | O(nlogn) |
| **Quick Sort** | O(nlogn) | O(nlogn) | $O(n^2)$ |
| **Heap Sort** | O(nlogn) | O(nlogn) | O(nlogn) |
| **Γραμμικής Αναζήτησης** | O(n) | O(n) | O(n) |
| **Δυαδικής Αναζήτησης** | O(logn) | O(logn) | O(logn) |
| **Αναζήτησης με Παρεμβολή** | O(loglogn) | O(loglogn) | O(n) |
| **Δυικής Αναζήτησης Παρεμβολής (BIS)** | O(loglogn) | O(loglogn) | O(√n) |
| **Βελτιωμένη Παραλλαγής του BIS** | O(loglogn) | O(loglogn) | O(logn) |

***Επιμέλεια:***

*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*

*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*

*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

# PART II: BSTs && HASHING

(A)

Ένα δυαδικό δέντρο αποτελείται από ένα πλήθος κόμβων με μοναδικό αριθμό ο κάθε ένας και με ίδια ομάδα περιεχομένων, οι οποίοι πρέπει να έχουν το πολύ δύο παιδιά με αριστερά το μικρότερο και δεξιά το μεγαλύτερο από αυτούς.

Για έναν αλγόριθμο υλοποιημένο με **Δυαδικό Δένδρο Αναζήτησης (ΔΔΑ)** ισχύει ότι αποτελείται από ένα δυαδικό δέντρο στο οποίο γίνεται αναζήτηση ενός στοιχείου η πλήθος δεδομένων όπως στην περίπτωση μας, αφού κάθε κόμβος αποτελείται από μία δομή δεδομένων (struct). Η συγκεκριμένη υλοποίηση είναι αρκετά αποδοτική, διότι μειώνει την περιοχή αναζήτησης στο μισό, αφού μετά από κάθε αναζήτηση ο αλγόριθμος ψάχνει σε κάποιο υποδένδρο. Ακόμα η πολυπλοκότητα του αλγορίθμου είναι στην καλύτερη περίπτωση **O(logn) για balanced δένδρα, ενώ για not balanced O(h) (με h το ύψος του δένδρου).**

- 1.
  Για την απεικόνιση του ΔΔΑ με ενδο-διατεταγμένη διάσχιση ισχύει ότι η «επίσκεψη» κάθε κόμβου είναι προφανής (με γνωστό ορισμό και τρόπο). Πιο συγκεκριμένα αν το δέντρο δεν είναι άδειο πρώτα εκτελείται «επίσκεψη» του αριστερού υποδένδρου, μετά της ρίζας και τέλος του δεξιού υποδένδρου.

- 2.
  Για την αναζήτηση ενός φοιτητή βάση του ΑΜ του ισχύει ότι ξεκινάει από την ρίζα και συγκρίνεται κάθε φορά το ζητούμενο στοιχείο (ΑΜ φοιτητή) με το αριστερό ή δεξιό παιδί για να βρει σε ποιο σημείο βρίσκεται και ανάλογα με το αν είναι μικρότερο η μεγαλύτερο αντίστοιχα επιλέγεται το αντίστοιχο παιδί, κοκ μέχρι να βρεθεί το στοιχείο(κόμβος) ή το NULL που σημαίνει πως ο μαθητής δεν βρίσκεται στην δομή μας.

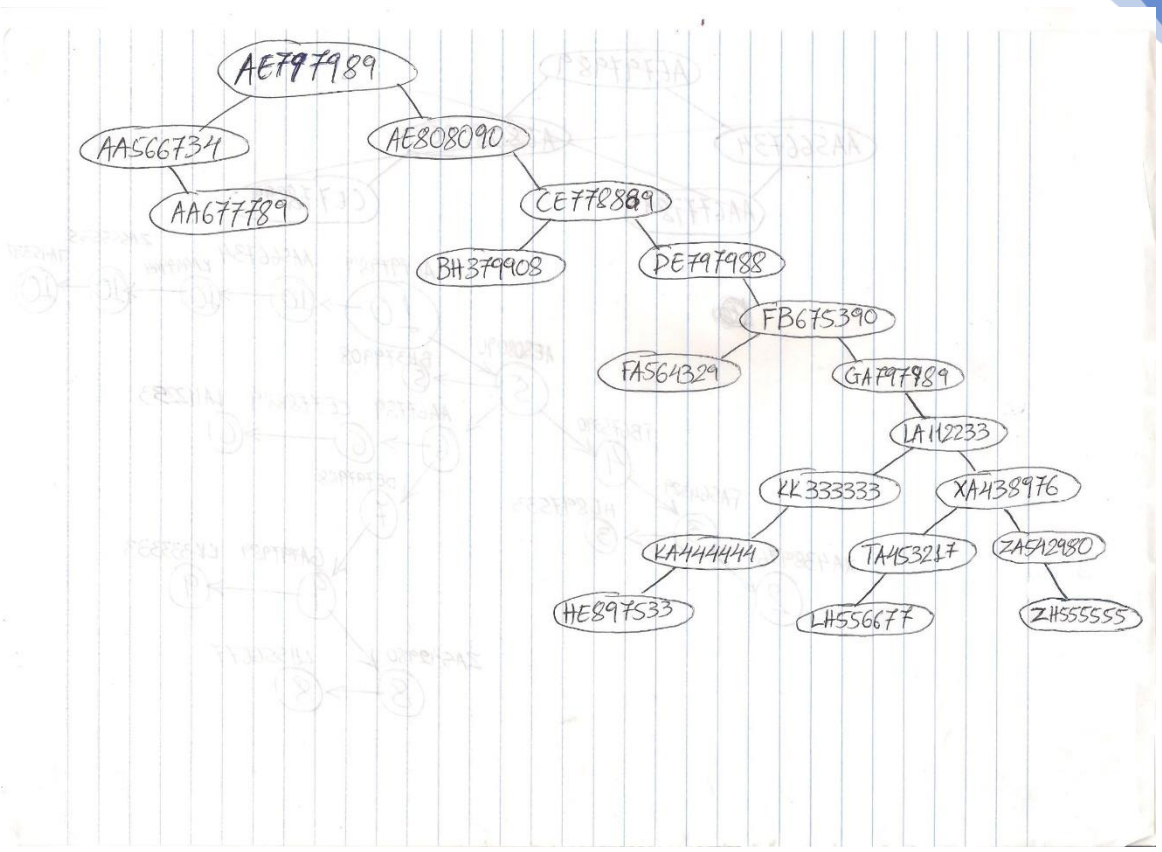
- 3.
  Για την τροποποίηση των καταχωρημένων στοιχείων ισχύει ότι πάλι γίνεται αναζήτηση του φοιτητή(κόμβου) με βάση το ΑΜ του και στη συνέχεια τροποποιείται κάποιο δεδομένο του που είναι αποθηκευμένο ως bucket μέσα στον κόμβο και αποθηκεύεται η αλλαγή μέχρι την έξοδο από το πρόγραμμα.

***Επιμέλεια:***
*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

- 4.
  Για την διαγραφή μιας εγγραφής του φοιτητή βάση του ΑΜ του ισχύει ότι διαγράφουμε την εγγραφή(ένα στοιχείο του bucket) που επιθυμούμε, αφού πρώτα έχουμε βρει τον κόμβο(bucket) μέσω της αναζήτησης βάση του ΑΜ . Η διαγραφή γίνεται μόνο αν το δέντρο δεν είναι άδειο και ταυτόχρονα αν ο φοιτητής και η εγγραφή υπάρχουν. Κατά την διαγραφή ενός κόμβου παρατηρήσαμε πως υπάρχουν 3 περιπτώσεις. Αρχικά, αν ο κόμβος που θέλουμε να σβηστεί δεν έχει παιδιά τότε απλά τον ελευθερώνουμε και ορίζουμε πως ο προηγούμενος του δείχνει στο NULL. Δεύτερον, αν ο κόμβος έχει 1 παιδί τότε το τοποθετούμε στην θέση του πατέρα και διαγράφουμε τον κόμβο. Τέλος, αν ο κόμβος έχει 2 παιδιά βρίσκουμε αυτό με την μικρότερη αξία και τον τοποθετούμε στην θέση του κόμβου που διαγράφουμε. Τέλος, ελευθερώνουμε τον δείκτη του στοιχείου που επιθυμούμε να διαγράψουμε.

- 5.
  Για την έξοδο από την εφαρμογή ισχύει ότι όλες οι αλλαγές που υλοποιούμε ισχύουν καθ' όλη την διάρκεια που τρέχουμε το πρόγραμμα, μόλις σταματήσουμε να το τρέχουμε επανέρχεται στην προηγούμενη του κατάσταση.

***Επιμέλεια:***
*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*



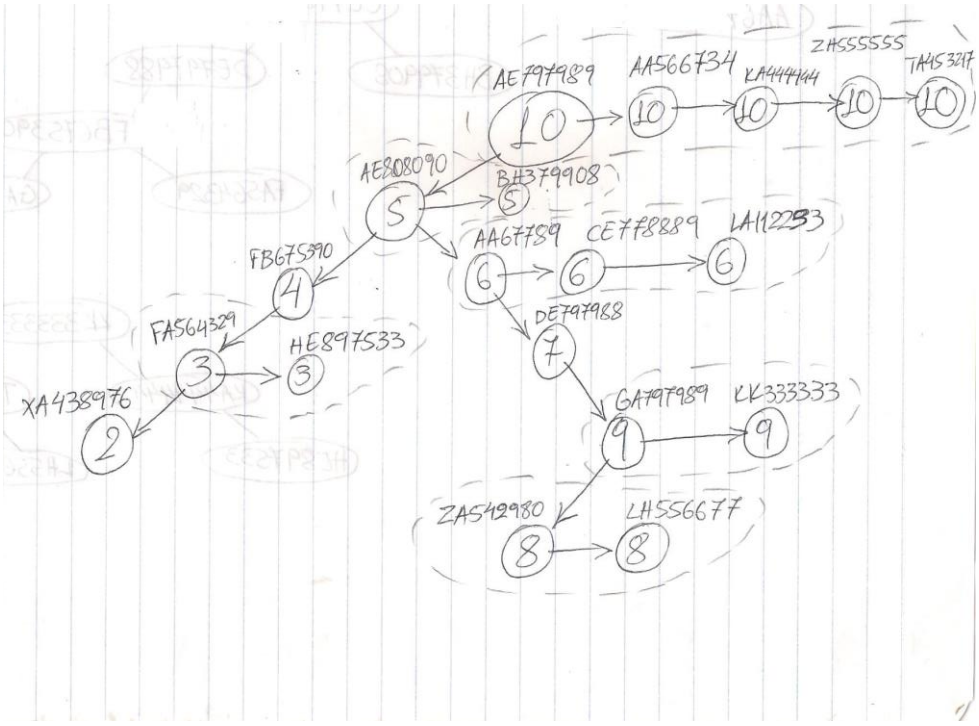

(B)

Όταν το ΔΔΑ διατάσσεται ως προς τη βαθμολογία της εγγραφής του φοιτητή στο μάθημα δομές δεδομένων, ισχύει ότι για την εισαγωγή των buckets των φοιτητών σε κόμβους με βάση τον βαθμό τους το δέντρο αρχικά πρέπει να είναι κενό οπότε ο πρώτος φοιτητής γίνεται ρίζα. Στη συνέχεια και εφόσον το δέντρο δεν είναι κενό ο επόμενος φοιτητής τοποθετείται στον πρώτο ελεύθερο κόμβο, με βάση τον βαθμό του έτσι ώστε να ισχύει η διάταξη. Ο κώδικας που υλοποιήσαμε αποτελεί έναν συνδυασμό διασυνδεδεμένης λίστας και δυαδικού δέντρου. Ουσιαστικά διατηρούμε την δομή του ΔΔΑ του 1ου ερωτήματος και προσθέτουμε έναν δείκτη σε κάθε κόμβο που κρατάει την βαθμολογία του κάθε μαθητή και δείχνει σε όλους τους υπόλοιπους μαθητές με την ίδια βαθμολογία. Σκεφτήκαμε την παραπάνω υλοποίηση ως λύση για την δυσκολία που αντιμετωπίσαμε όταν είδαμε πως υπάρχουν μαθητές με ίδιο βαθμό. Φανταστήκαμε λοιπόν πως αυτός ο δείκτης θα δημιουργεί έναν μεγαλύτερο κόμβο που κρατάει τα στοιχεία όλων των μαθητών με κοινούς βαθμούς.

***Επιμέλεια:***

*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

- 1. , 2.
  Για την εύρεση των φοιτητών με την ελάχιστη ή την μέγιστη βαθμολογία ισχύει ότι γίνεται αναζήτηση του φοιτητή-φοιτητών (κόμβου-κόμβων) με βάση τον βαθμό του-τους και στη συνέχεια εμφανίζεται το αποτέλεσμα στην οθόνη.




(Γ)

Ένα πλήθος από δεδομένα(buckets) μπορούμε να το εισάγουμε σε πίνακα με την τεχνική του Hashing, δηλαδή μιας λίστας αρχείων με κάθε θέση να περιέχει ένα μοναδικό κλειδί. Για να χρησιμοποιούμε όσο χώρο θέλουμε χωρίς να χρειάζεται να γίνεται αλλαγή μεγέθους του αρχικού Hash Table χρησιμοποιούμε **HASHING με αλυσίδες,** όπου κάθε κελί περιέχει έναν δείκτη που δείχνει σε μία λίστα με αυξομειωμένο μέγεθος μετατρέποντας την δομή σε δυναμική. Επομένως γίνεται αντιληπτό ότι η απόδοση του Hashing με αλυσίδες είναι καλύτερη τόσο από το απλό Hashing όσο και από το διπλό Hashing, με την πολυπλοκότητα στην χειρότερη περίπτωση να είναι **O(n) στην περίπτωση διαγραφής ή εύρεσης, αλλά O(1) στην εισαγωγή**.

***Επιμέλεια:***

*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

- 1.
  Για την αναζήτηση ενός φοιτητή βάσει του ΑΜ του ισχύει ότι αρχικά γίνεται υπολογισμός του κελιού του μέσω της Hash Function που μας δίνεται και στη συνέχεια εύρεσή του στη λίστα μέσω της υπόδειξης του δείκτη. Αφού υπολογίσουμε το κλειδί μέσω της συνάρτησης κατακερματισμού διατρέχουμε την αλυσίδα για να εντοπίσουμε το ζητούμενο στοιχείο.

- 2.
  Για την τροποποίηση των στοιχείων εγγραφής φοιτητή βάσει του ΑΜ του ισχύει ότι πάλι γίνεται αναζήτηση του φοιτητή(κελιού) με βάση της δοσμένης συνάρτησης κατακερματισμού του και στη συνέχεια τροποποιείται κάποιο δεδομένο που είναι αποθηκευμένο ως bucket μέσα στην αντίστοιχη διασυνδεδεμένη λίστα που έχουμε υλοποιήσει. Στην συνέχεια η τροποποίηση αποθηκεύεται.

- 3.
  Για την διαγραφή μιας εγγραφής φοιτητή από τον πίνακα κατακερματισμού βάσει του ΑΜ του ισχύει ότι διαγράφουμε την εγγραφή(ένα στοιχείο του bucket) που επιθυμούμε, αφού πρώτα έχουμε βρει το κελί (bucket) μέσω της αναζήτησης. Η διαγραφή γίνεται μέσω της αλλαγής υπόδειξης των δεικτών μέσα στην δομή μας. Απαραίτητη προϋπόθεση για την διαγραφή είναι να υπάρχει η εγγραφή. Υπάρχουν δυο περιπτώσεις. Πρώτη είναι ο ζητούμενος κόμβος να μην είναι ο πρώτος της διασυνδεδεμένης λίστας. Τότε, η διαγραφή εκτελείται με τον δείκτη του προηγούμενου (από αυτόν που θέλουμε να διαγράψουμε) κόμβου να δείχνει στον επόμενο. Στην συνέχεια γίνεται ελευθέρωση του δείκτη του κόμβου που διαγράψαμε. Αν ωστόσο είναι ο πρώτος κόμβος τότε αλλάζουμε τον δείκτη του hTable[key] ώστε αυτός να δείχνει στο επόμενο στοιχείο από αυτό που θέλουμε να σβηστεί από την δομή μας.

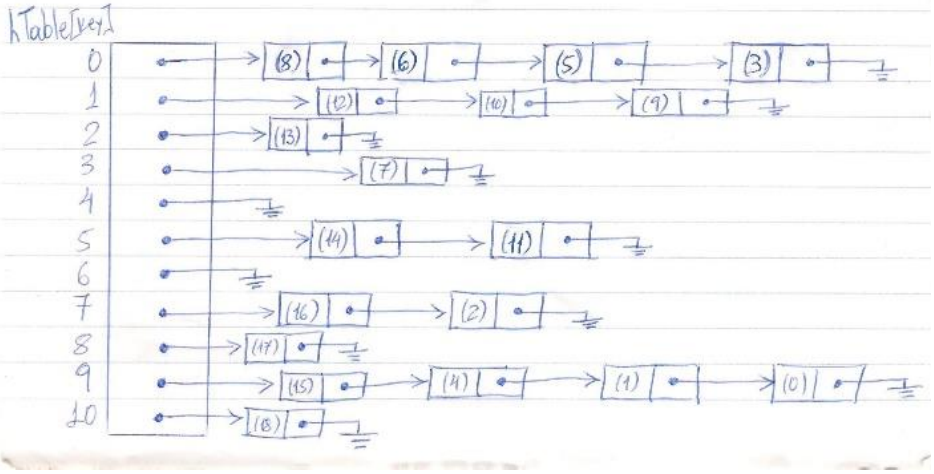
- 4.
  Για την έξοδο από την εφαρμογή ισχύει ότι όλες οι αλλαγές που υλοποιούμε ισχύουν καθ' όλη την διάρκεια που τρέχουμε το πρόγραμμα, μόλις σταματήσουμε να το τρέχουμε επανέρχεται στην προηγούμενη του κατάσταση.

***Επιμέλεια:***

*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*
*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*
*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*

# ΑΝΑΦΟΡΑ PROJECT DATA STRUCTURES 2018-2019

| | AM | total ascii | key |
|---|---|---|---|
| 0) | AE797986 | 471 | 9 |
| 1) | AA566734 | 449 | 9 |
| 2) | AE808090 | 447 | 7 |
| 3) | AA677789 | 462 | 0 |
| 4) | CE778899 | 471 | 9 |
| 5) | DE797988 | 473 | 0 |
| 6) | BH379908 | 462 | 0 |
| 7) | FB675390 | 454 | 3 |
| 8) | GA797989 | 473 | 0 |
| 9) | FA564329 | 452 | 1 |
| 10) | LA112233 | 441 | 1 |
| 11) | KK333333 | 456 | 5 |
| 12) | KA444444 | 452 | 1 |
| 13) | HE897533 | 464 | 2 |
| 14) | XA438976 | 478 | 5 |
| 15) | ZA542980 | 471 | 9 |
| 16) | ZH555555 | 480 | 7 |
| 17) | TA453217 | 459 | 8 |
| 18) | LH556677 | 472 | 10 |




***Επιμέλεια:***

*Κουκουβέλα Ασπασία ΑΜ: 1059617 e-mail: st1059617@ceid.upatras.gr*

*Ηλιοπούλου Σταυρούλα ΑΜ: 1059626 e-mail: st1059626@ceid.upatras.gr*

*Μαχιά Αριάδνη ΑΜ: 1059556 e-mail: st1059556@ceid.upatras.gr*