ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων

Διπλωματική εργασία

«Μελέτη δομών τυχαίων δένδρων αναζήτησης»

Όνομα: **ΔΗΜΗΤΡΑ ΚΑΤΑΝΟΥ**

Επιβλέπων καθηγητής: Παναγιώτης Μποζάνης

Βόλος, Φεβρουάριος 2011

Μελέτη δομών τυχαίων δένδρων αναζήτησης

Δήμητρα Κατάνου

……	……..
Παναγιώτης Μποζάνης	Δημήτριος Κατσαρός
Επίκουρος Καθηγητής	Λέκτορας
Παν. Θεσσαλίας	Παν. Θεσσαλίας

Βόλος, Φεβρουάριος 2011

Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου, κ. Παναγιώτη Μποζάνη για την αμέριστη βοήθεια που μου προσέφερε σε όλες τις φάσεις συγγραφής αυτής της εργασίας και για τις πολύτιμες συμβουλές του πάνω σε ερωτήματα που προέκυπταν κατά τη διάρκεια της εκπόνησης της διατριβής.

Στη συνέχεια, θα ήθελα να ευχαριστήσω τους γονείς μου, Γιάννη και Εύα, για τη στήριξη, που μου έδειξαν όλα αυτά τα χρόνια. Χωρίς την βοήθεια τους, την καθοδήγησή τους, αλλά κυρίως την αγάπη τους δε θα κατάφερνα να ολοκληρώσω τις σπουδές μου και να πραγματοποιήσω τα όνειρά μου. Τους οφείλω πολλά και τους υπερευχαριστώ.

Τέλος, ένα πολύ μεγάλο ευχαριστώ για τις δύο αγαπημένες μου φίλες, Κατερίνα και Κυριακή Γρατσία, όχι μόνο για τη συμπαράστασή τους, αλλά και για όλα αυτά τα υπέροχα χρόνια που πορευτήκαμε μαζί μέσα από δυσκολίες αλλά κυρίως μέσα από πολύ ευτυχισμένες στιγμές, δημιουργώντας μια αληθινή, αδελφική σχέση.

Περίληψη

Στην παρούσα διπλωματική εργασία έγινε μια μελέτη τυχαίων δομών και κυρίως, *τυχαίων δυαδικών δένδρων αναζήτησης*. Αφού παρουσιάστηκε μια θεωρητική προσέγγιση των δομών, έγινε μελέτη διαφόρων αλγορίθμων ενημέρωσης των δένδρων και περιγραφή τους σε ψευδογλώσσα, δηλαδή αλγόριθμοι ένθεσης και διαγραφής στοιχείων, πραγματοποιήθηκε η υλοποίηση τους σε γλώσσα προγραμματισμού Java και στη συνέχεια δόθηκε έμφαση στην ανάλυση της πολυπλοκότητας των τυχαίων δομών.

Η διάρθρωση της ύλης έγινε ως εξής:

Το *Κεφάλαιο 1* αποτελεί μια εισαγωγή σε βασικές δομές, όπως η δομή λεξικού και τα δένδρα αναζήτησης. Αυτές οι έννοιες είναι χρήσιμες για τα επόμενα κεφάλαια. Επίσης γίνεται μια σύντομη αναφορά και περιγραφή και άλλων δομών δένδρων αναζήτησης.

Στο *Κεφάλαιο 2* γίνεται περιγραφή διάφορων τυχαίων δομών. Γίνεται αναλυτική περιγραφή των τυχαίων δένδρων αναζήτησης και περιγράφονται τυχαιοποιημένοι αλγόριθμοι για διάφορα μοντέλα των τυχαίων δένδρων. Παρουσιάζονται στιγμιότυπα διαδοχικών ενθέσεων/αποσβέσεων από τα δένδρα, ώστε να γίνεται κατανοητό από τον αναγνώστη πως αναπτύσσονται τα δένδρα.

Στο *Κεφάλαιο 3* γίνεται μελέτη για την απόδοση των αλγορίθμων που παρουσιάστηκαν στο προηγούμενο κεφάλαιο. Παρουσιάζονται οι γραφικές παραστάσεις που προέκυψαν από μετρήσεις

Στο Κεφάλαιο 4 παραθέτουμε τις βασικές classes του κώδικα, από την υλοποίηση των αλγορίθμων.

Abstract

This thesis is a study of random data structures and most importantly about random binary search trees. Having presented a theoretical approach to the structures, we studied various update algorithms of trees and described then in pseudocode, like insertion and deletion of data, they were implemented in Java and then we emphasized the analysis of the complexity of data structures.

Chapter 1 is an introduction to basic structures such as the dictionary structure and the search trees. These concepts are useful in subsequent chapters. There is also quick reference and description of other search trees structures.

In *Chapter 2* there is a description of various random structures. There is a detailed description of random search trees and we describe randomized algorithms for various models of random trees. We present snapshots of subsequent insertions/deletions from the trees, in order to be understood by the reader, how the trees develop.

In *Chapter 3* there is a study of the performance of the algorithms presented in the previous chapters. We show the graphs obtained from measurements.

In *Chapter 4* we present the main classes of the code, from the implementation of the algorithms.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγικά

1.1 Δομή Λεξικού

Έστω ένα σύνολο από διατεταγμένες δυάδες, όπου η κάθε δυάδα αποτελείται από ένα κλειδί, από ένα σύνολο διατεταγμένων κλειδιών και από μια συσχετιζόμενη πληροφορία. Αυτά τα ζεύγη ονομάζονται στοιχεία (items). Μια δομή επί αυτού του συνόλου ονομάζεται ***δομή λεξικού (dictionary data structure)***[3]. Αυτή η δομή πρέπει να ικανοποίει τον αφηρημένο τύπο δεδομένων (ΑΤΔ), ο οποίος κάνει:

- **Εύρεση ενός στοιχείου.** Η αναζήτηση του στοιχείου γίνεται βάσει του κλειδιού του, κι επιστρέφεται η αντίστοιχη πληροφορία
- **Ένθεση ενός στοιχείου.** Γίνεται εισαγωγή ενός στοιχείου, το οποίο αποτελείται από μια πληροφορία info και ένα κλειδί key.
- **Απόσβεση ενός στοιχείου**. Η διαγραφή του στοιχείου γίνεται βάσει του κλειδιού.

Σχήμα 1.1: Γραφική αναπαράσταση στοιχείου λεξικού

1.2 Δένδρα αναζήτησης

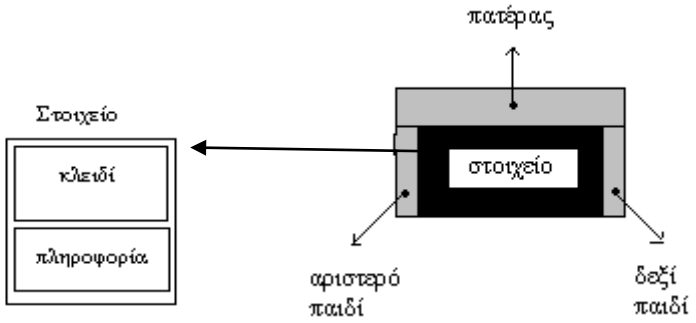
1.2.1 Ορισμός Δυναμικών Δένδρων

Τα ***δυναμικά δένδρα (dynamic trees)***[3] είναι δένδρα τα οποία προκύπτουν με την ένθεση και απόσβεση κόμβων, δηλαδή με δύο πράξεις όπου η πρώτη προσθέτει ένα κόμβο ως αριστερό ή δεξί παιδί και η δεύτερη αφαιρεί έναν κόμβο.

1.2.2 Δυαδικά Δένδρα Αναζητήσεως

Τα ***δυαδικά δένδρα αναζητήσεως (binary search trees)***[3] προκύπτουν από τα δυναμικά δυαδικά δένδρα βάσει της δομής Λεξικού. Για κάθε κόμβο u του δυαδικού δένδρου αναζήτησης ισχύει ότι το κλειδί του αριστερού παιδιού του u είναι μικρότερο από αυτό του u και το κλειδί του δεξιού παιδιού του u είναι μεγαλύτερο από του u. Οι τρεις βασικές πράξεις σε ένα δυαδικό δένδρο αναζήτησης είναι:

- **Αναζήτηση στοιχείου.** Η αναζήτηση γίνεται με βάση το κλειδί ενός κόμβου και σε περίπτωση που βρεθεί, επιστρέφεται η πληροφορία του.
- **Εισαγωγή στοιχείου.** Η εισαγωγή γίνεται μόνο όταν το στοιχείο δεν υπάρχει ήδη μέσα στο δένδρο.
- **Διαγραφή στοιχείου.** Η διαγραφή περιλαμβάνει μια αναζήτηση του στοιχείου βάσει του κλειδιού του και σε περίπτωση που βρεθεί διαγράφεται.



Σχήμα 1.2: Γραφική αναπαράσταση κόμβου δυαδικού δένδρου αναζήτησης

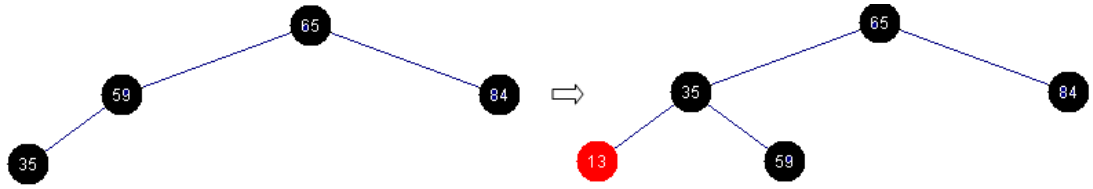
1.2.3 Δένδρα AVL

Το δένδρο ***AVL***[3] είναι ένα δυαδικό δένδρο στο οποίο μπαίνουν όρια στο ύψος του, δηλαδή τα ύψη των υποδένδρων κάθε εσωτερικού κόμβου του, διαφέρουν το πολύ κατά ένα. Το AVL χαρακτηρίζεται από τις *επαναζυγιστικές πράξεις (rebalancing operations)* , οι οποίες προσπαθούν να διατηρήσουν τη διαφορά ύψους στη μονάδα.

Οι βασικές πράξεις στα AVL δένδρα είναι:

- **Αναζήτηση στοιχείου.** Η αναζήτηση ενός στοιχείου γίνεται με βάση το κλειδί του, όπως και στα δυαδικά δένδρα αναζήτησης.
- **Εισαγωγή στοιχείου.** Σε περίπτωση παραβίασης της συνθήκης ισοζύγισης μετά την ένθεση ενός κόμβου, χρειάζεται να γίνει μια απλή περιστροφή ή μια διπλή περιστροφή ώστε να επανέλθει η ισορροπία στο δένδρο.
- **Διαγραφή στοιχείου.** Και εδώ, μόλις διαγραφεί ο κόμβος, χρειάζεται επαναζύγιση για τη διατήρηση της ισορροπίας.

Τα δένδρα AVL επιδεικνύουν λογαριθμική συμπεριφορά και για τις τρεις πράξεις.



Σχήμα 1.3: Παράδειγμα επαναζυγιστικής πράξης με την εισαγωγή του στοιχείου 13

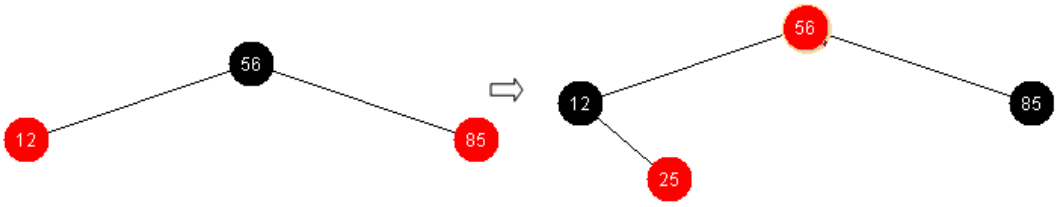
1.2.4 Ερυθρόμαυρα δένδρα

Το ***ερυθρόμαυρο red-black***[3] δένδρο είναι ένα δυαδικό δένδρο όπου όλοι οι κόμβοι του είναι χρωματισμένοι, κόκκινοι ή μαύροι ως εξής: η ρίζα και τα φύλλα είναι μαύρα, κάθε κόκκινος κόμβος έχει μαύρα παιδιά και όλα τα μονοπάτια από τη ρίζα προς τα φύλλα έχουν τον ίδιο αριθμό μαύρων κόμβων.

Οι βασικές πράξεις στα ερυθρόμαυρα δένδρα είναι:

- **Αναζήτηση στοιχείου.** Η αναζήτηση ενός στοιχείου γίνεται με βάση το κλειδί του, όπως και στα δυαδικά δένδρα αναζήτησης.
- **Εισαγωγή στοιχείου.** Η ένθεση του στοιχείου γίνεται όπως και στα απλά αζύγιστα δυαδικά δένδρα. Σε περίπτωση που παραβιαστεί κάποιος από τους κανόνες του ορισμού, αυτό λύνεται είτε με απλή/διπλή περιστροφή ή με αντιστροφή χρωμάτων.

- **Απόσβεση στοιχείου**. Η διαγραφή γίνεται κανονικά εάν ο κόμβος που στεγάζει το στοιχείο είναι κόκκινος. Σε περίπτωση που είναι μαύρος όμως, χρειάζονται να γίνουν κάποιες επιπλέον ενέργειες για την αποκατάσταση της ισορροπίας, όπως περιστροφές ή αλλαγή χρώματος.



Σχήμα 1.4: Παράδειγμα αντιστροφής χρωμάτων με την ένθεση του στοιχείου 25

1.2.5 Άλλοι τύποι ισοζυγισμένων δένδρων αναζήτησης

Κάνουμε μια σύντομη αναφορά σε άλλους τύπους δένδρων αναζήτησης.

Δένδρα-(a, b)

Τα ***δένδρα-(a, b)***[3] αποτελούν ισοζυγισμένα δένδρα τέτοια ώστε όλα τα φύλλα τους έχουν το ίδιο βάθος, η ρίζα έχει τουλάχιστον δύο παιδιά και το πολύ b , όπου $b \geq 2\alpha$, $\alpha \geq 2$ και το πλήθος των παιδιών των υπόλοιπων εσωτερικών κόμβων είναι μεταξύ α και b.

BB[a] δένδρα

Τα ***δένδρα BB[a]***[9] ή ***weight-balanced trees*** είναι δυαδικά δένδρα αναζήτησης που είναι κατασκευασμένα βάσει των πιθανοτήτων αναζήτησης του κάθε κόμβου. Σε κάθε υποδένδρο, ρίζα είναι ο κόμβος με το μεγαλύτερο βάρος, δηλαδή με την μεγαλύτερη πιθανότητα. Έχουν αρκετές ομοιότητες με τα treaps, που θα εξετάσουμε σε επόμενο κεφάλαιο.

Height-ratio balanced trees

Τα δένδρα ***height-ratio balanced***[10] είναι παρόμοια με τα weight balanced trees, όμως είναι κατασκευασμένα βάσει του ύψους.

Κεφάλαιο 2

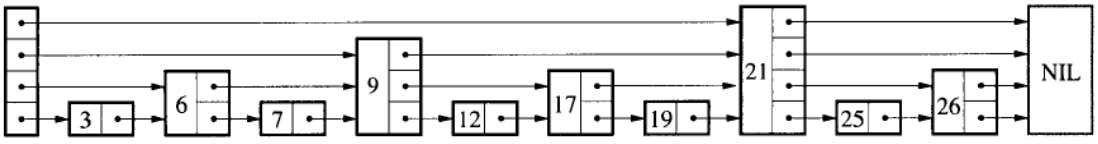
Τυχαίες Δομές και Τυχαιοποιημένοι Αλγόριθμοι

2.1 Τυχαίες Δομές

Οι ***τυχαίες δομές (randomized)*** αποτελούν κι αυτές δομές λεξικού, οι οποίες βασίζουν την προσαρμογή τους στις μεταβολές του υποκείμενου συνόλου, σε τυχαίες επαναζυγιστικές πράξεις, σε αντίθεση με τις δομές που εξετάσαμε πιο πάνω, οι οποίες λύνουν το πρόβλημα λεξικού, ωστόσο η αποτελεσματικότητα τους βασίζεται σε συμμετρικές επαναζυγιστικές πράξεις, γεγονός που αυξάνει την πολυπλοκότητα και την δυσκολία της υλοποίησης. Οι τυχαίες δομές έχουν απλούστερη υλοποίηση και χαμηλότερο βαθμό πολυπλοκότητας καθώς βασίζονται στην τυχαιότητα των δεδομένων.

2.1.1 Λίστες Αναπηδήσεως

Οι ***λίστες αναπηδήσεως (skip lists)***[8] είναι δομές που βασίζονται σε παράλληλες συνδεδεμένες λίστες. Αυτό που πετυχαίνουμε με τις λίστες αναπηδήσεως, είναι ότι σε μια αναζήτηση δε χρειάζεται να εξετάζουμε όλους τους κόμβους μιας λίστας αλλά με τη χρήση δεικτών μπορούμε να υλοποιήσουμε λίστες υψηλότερου επιπέδου, οι οποίες περιέχουν τα μισά ή και λιγότερα στοιχεία από την αρχική. Όλες οι πράξεις εγγυώνται λογαριθμικό κόστος.



Σχήμα 2.1: Λίστα Αναπηδήσεως

Στο σχήμα 2.1 βλέπουμε την αρχική διασυνδεδεμένη λίστα που περιλαμβάνει όλα τα στοιχεία ενώ στο δεύτερο επίπεδο μπορούμε να μειώσουμε τα στοιχεία που θα

εξετάσουμε σε μια αναζήτηση σε [n/2]+1, με τη χρήση δεικτών προς τους κόμβους που βρίσκονται δυο θέσεις πιο μπροστά. Στα επόμενα επίπεδα μειώνονται ακόμα περισσότερο οι κόμβοι, καθώς οι δείκτες τώρα δείχνουν σε στοιχεία που βρίσκονται τοποθετημένα 4 θέσεις πιο μπροστά και έτσι τα στοιχεία μειώνονται σε [n/4]+2. Έτσι το κόστος αναζητήσεως μειώνεται σε λογαριθμικό.

2.1.2 Treaps

Οι δομές ***treaps*** (δεν υπάρχει μετάφραση του όρου) προκύπτουν από τη συγχώνευση των δομών *tree* και *heap* (σωρός). Αυτό που τα διαφοροποιεί από τα δένδρα αναζητήσεως είναι ότι το κάθε στοιχείο, εκτός από το κλειδί που το χαρακτηρίζει, σχετίζεται και με μια **προτεραιότητα** (priority), δηλαδή έναν αριθμό από ένα ολικώς διατεταγμένο σύμπαν. Συνεπώς, το treap αποτελείται από ένα δυαδικό δένδρο αναζήτησης, όπου οι κόμβοι ακολουθούν, ως προς τα κλειδιά τη συμμετρική διάταξη, ενώ, ως προς τις προτεραιότητες ακολουθούν τη διάταξη σωρού[1]. Τελικά, το treap αποτελεί ένα δένδρο αναζητήσεως χωρίς τις προτεραιότητες, ενώ, ως προς αυτές ένας κόμβος διαθέτει στοιχείο με μεγαλύτερο αριθμό προτεραιότητας από του πατέρα του.

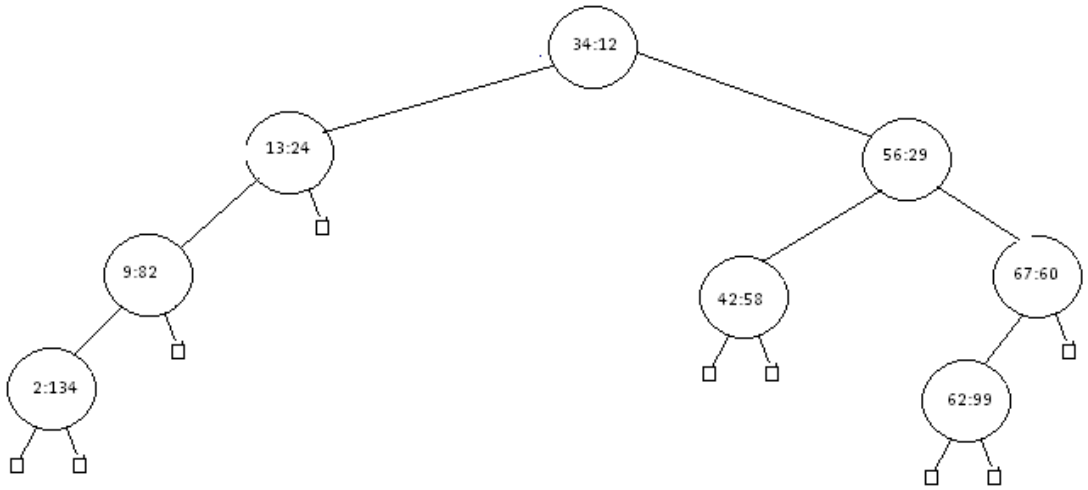
Οι βασικές πράξεις των treaps είναι:

- **Αναζήτηση στοιχείου**. Η αναζήτηση ενός στοιχείου γίνεται με βάση το κλειδί του, όπως και στα δυαδικά δένδρα αναζήτησης.
- **Εισαγωγή στοιχείου.** Το στοιχείο εισάγεται στο δένδρο βάσει του κλειδιού του σε έναν κόμβο-φύλλο και στη συνέχεια βάσει της προτεραιότητας του, όσο η συνθήκη σωρού παραβιάζεται προάγουμε τον κόμβο, με δεξιές ή αριστερές περιστροφές.
- **Απόσβεση στοιχείου.** Το στοιχείο που πρόκειται να διαγραφεί, εντοπίζεται και με διαδοχικές δεξιές ή αριστερές απλές περιστροφές υποβιβάζεται μέχρι να βρεθεί σε θέση φύλλου, όπου και διαγράφεται.

Και εδώ, όπως και στις skip lists, το κόστος και των τριών πράξεων είναι λογαριθμικό.

Στο επόμενο σχήμα παρουσιάζεται ένα treap, όπου σε κάθε κόμβο αναγράφονται οι δυο τιμές με τον εξής συμβολισμό: [κλειδί: προτεραιότητα].

[1] Στη διάταξη σωρού μέγιστου στοιχείου, το στοιχείο κάθε κόμβου είναι μεγαλύτερο από αυτά των παιδιών του ενώ στη διάταξη σωρού ελάχιστου στοιχείου, συμβαίνει το αντίθετο.



Σχήμα 2.2: Παράδειγμα ενός treap

2.1.3 Τυχαία δυαδικά δένδρα αναζήτησης

Τα ***τυχαία δυαδικά δένδρα αναζήτησης (random binary search trees)*** είναι δυαδικά δένδρα αναζήτησης που προκύπτουν κάνοντας μόνο τυχαίες ενθέσεις στοιχείων. Μια εισαγωγή σε ένα δυαδικό δένδρο αναζήτησης είναι ***τυχαία*** μόνο εάν για κάθε κλειδί που εισάγεται, η πιθανότητα να βρεθεί σε οποιοδήποτε από τα διαστήματα που καθορίζουν τα κλειδιά που βρίσκονται ήδη μέσα στο δένδρο, είναι ίδια. Η αναμενόμενη συμπεριφορά όλων των πράξεων στα τυχαία δυαδικά δένδρα αναζήτησης είναι O(log *n*).

2.2 Τυχαιοποιημένοι αλγόριθμοι

Εξετάζουμε τυχαιοποιημένους αλγορίθμους εισαγωγής και διαγραφής στοιχείων σε διάφορα ισοδύναμα μοντέλα που δυναμικά διατηρούν δομή λεξικού σε ένα δυαδικό δένδρο αναζήτησης. Δεν μας ενδιαφέρει η τυχαιότητα των στοιχείων που εισάγονται ή διαγράφονται αλλά μπορούμε να παράγουμε αυτήν την τυχαιότητα μέσα από τους αλγορίθμους ένθεσης- απόσβεσης. Όλοι οι αλγόριθμοι που παρουσιάζονται παρακάτω εξασφαλίζουν τυχαιότητα δηλαδή τα δένδρα που παράγονται μετά την εφαρμογή αυτών των αλγορίθμων παραμένουν τυχαία δυαδικά δένδρα αναζήτησης. Οι αλγόριθμοι που προϋπήρχαν[1] βασίζονταν στην τυχαία σειρά εισαγωγής και διαγραφής στοιχείων, δηλαδή οι ερευνητές εξέταζαν την τυχαιότητα στα δεδομένα και έτσι ενώ οι αλγόριθμοι εισαγωγής εξασφάλιζαν τυχαιότητα, δεν υπήρχε κάποιος αλγόριθμος διαγραφής που να εξασφαλίζει τυχαιότητα. Έτσι υιοθέτησαν την ιδέα ότι

η τυχαιότητα πρέπει να παραχθεί μέσα στους ίδιους τους αλγορίθμους. Έτσι οι παρακάτω αλγόριθμοι βασίζουν τις τυχαίες επιλογές τους είτε στα μεγέθη των υποδένδρων, στα priorities ή σε τελείως τυχαίες επιλογές.

2.2.1 Randomized Binary Search Trees

Τα ***Randomized Binary Search Trees (RBST)*** στην ουσία αποτελούν τυχαία δυαδικά δένδρα αναζήτησης, άσχετα από τη σειρά που τα στοιχεία εισάγονται ή διαγράφονται. Τα RBST έχουν κι αυτά εγγυημένα αναμενόμενη λογαριθμική συμπεριφορά.

Εκτός από τον παραπάνω ορισμό για τα τυχαία δυαδικά δένδρα αναζήτησης, υπάρχει και ο εξής:

Έστω *T* ένα δυαδικό δέντρο αναζήτησης μεγέθους n.

-- Αν n = 0, τότε *T* = □ και είναι ένα τυχαίο δυαδικό δέντρο αναζήτησης (το □ συμβολίζει το κενό δέντρο ή τον εξωτερικό κόμβο)

-- Αν n>0, το δέντρο *T* είναι ένα τυχαίο δυαδικό δέντρο αναζήτησης αν και μόνο αν και τα δύο υποδένδρα του, το αριστερό L και το δεξί R, είναι ανεξάρτητα τυχαία δυαδικά δένδρα αναζήτησης και

$$\Pr\{ \text{size}(L) = i \mid \text{size}(T) = n\} = \frac{1}{n}, \; i=0,\ldots, n-1, \; n>0 \qquad (1)$$

Οι αλγόριθμοι ένθεσης και διαγραφής των RBST διατηρούν την τυχαιότητα, βασίζοντας τις τυχαίες επιλογές τους σε δομικές πληροφορίες, δηλαδή στο μέγεθος του δένδρου, σε αντίθεση με τα treaps που βασίζονται σε τυχαίους αριθμούς, τα priorities.

Εισαγωγή στοιχείου

Εφαρμόζουμε τον κάτωθι αλγόριθμο

Αλγόριθμος 1 Εισαγωγή

```
Node insert ( int x, Node T ) {
        int n, r;

        n = T→size;
        r= random ( 0, n );
        if ( r ==n )
                return insert_at_root ( x, T );
        if ( x < T→key )
                T→left = insert( x, T→left );
```

```
        else
                T→right = insert( x, T→right );
        return T;
}
```

Υποθέτουμε ότι το κάθε στοιχείο του δένδρου αποτελείται από ένα κλειδί και όχι κάποια άλλη συσχετιζόμενη πληροφορία, και ότι όλα τα κλειδιά είναι μη αρνητικοί ακέραιοι. Η ιδιότητα που προκύπτει από την σχέση (1) είναι ότι κάθε κλειδί του δένδρου μεγέθους N, έχει την ίδια πιθανότητα να γίνει ρίζα του δένδρου, η οποία είναι 1/n. Με σκοπό να παράγουμε ένα τυχαίο δυαδικό δέντρο, το κλειδί , το οποίο δεν υπάρχει στο δένδρο από πριν, έχει πιθανότητα να γίνει η ρίζα του δένδρου ή να εισαχθεί σε κάποιο από τα δύο υποδένδρα της ρίζας. Το πεδίο size δεν θα αλλάζει, για λόγους απλότητας.

Αρχικά επιλέγουμε έναν τυχαίο αριθμό r μεταξύ (0…n) όπου n είναι το μέγεθος του δένδρου. Αν το δένδρο είναι αρχικά άδειο τότε θα δημιουργηθεί ένα δένδρο με μοναδικό κόμβο-ρίζα με κλειδί x και δύο άδεια υποδένδρα. Αν το δένδρο δεν είναι άδειο τότε το x μπορεί να γίνει ρίζα του δένδρου με πιθανότητα 1/(n+1) ή να εισαχθεί αναδρομικά στο αριστερό ή δεξί υποδένδρο του T, ανάλογα με τη σχέση του x με το κλειδί της ρίζας, με πιθανότητα $1 – 1/(n + 1) = n/(n + 1)$.

Αλγόριθμος 2 Εισαγωγή στη ρίζα

```
Node insert_at_root ( int x, Node T ) {
        Node S, G;

        split( x, T, &S, &G );
        T = new_node( );
        T→key = x;
        T→left =S;
        T→right = G;
        return T;
}
```

Στον αλγόριθμο 2 περιγράφεται πως εισάγουμε τον κόμβο με κλειδί x στο δένδρο, στη θέση της ρίζας και πως διασπάμε το δένδρο σε δύο υποδένδρα $T_<$ και $T_>$ με τη βοήθεια της split.

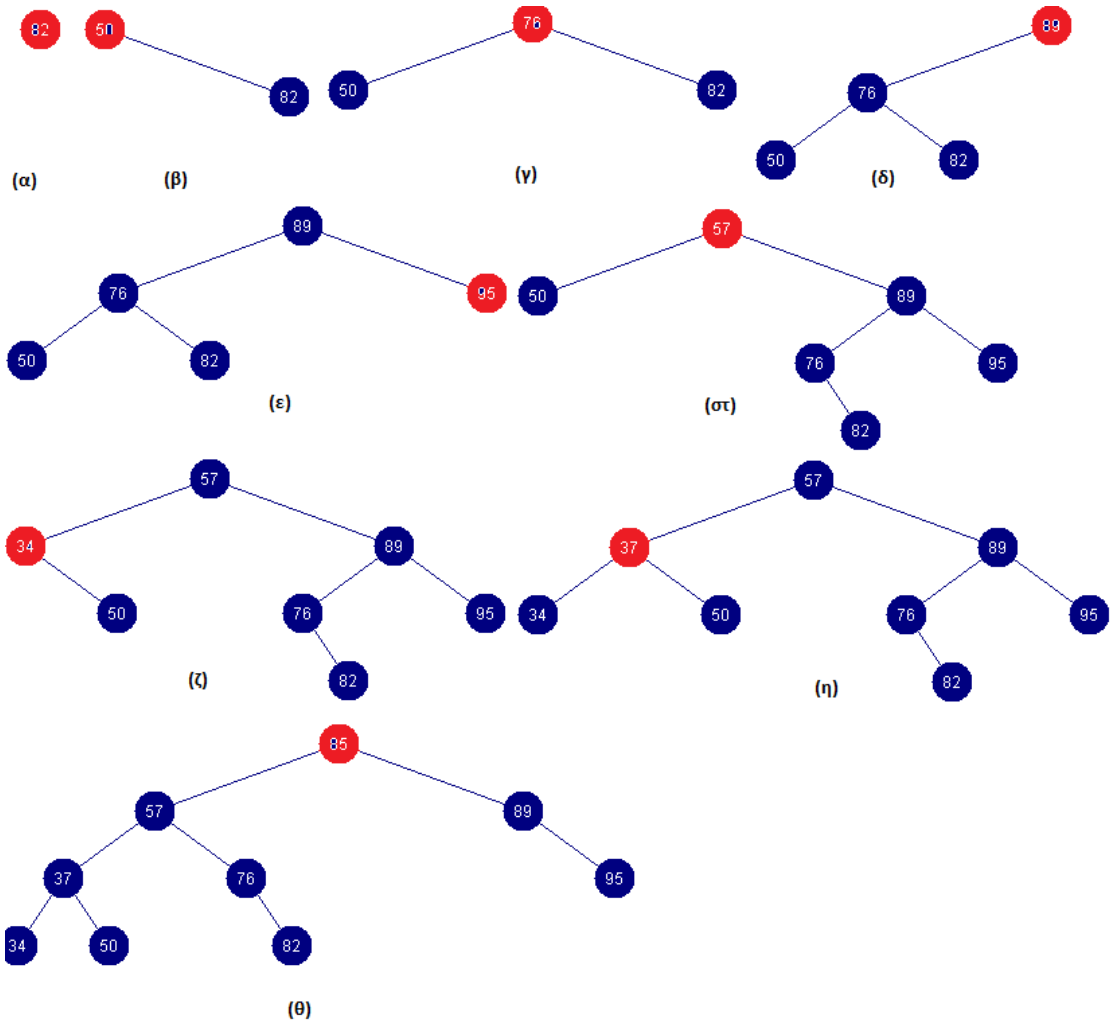
```
void split ( int x, Node T, Node *S, Node *G ) {
        if ( T ==  ) {
                *S = *G = ;
                return;
        }
        if ( x < T→key ) {
                *G = T;
                split ( x, T→left, S, &( *G→left ) );
```

```
        }
        else{
                *S = T;
                split ( x, T→right, &( *S→right ), G );
        }
        return;
}
```

Η split λειτουργεί ως εξής: αν το δένδρο είναι άδειο τα $T_<$ και $T_>$ θα είναι επίσης άδεια. Αλλιώς αν x < T→key τότε το δεξί υποδένδρο του T και το T ανήκουν στο $T_>$. Για να υπολογίσουμε το $T_<$ και το υπολειπόμενο κομμάτι του $T_>$, κάνουμε μια αναδρομική κλήση split(x, T→left). Αν x > T→key, πράττουμε ανάλογα.

Ο αλγόριθμος εισαγωγής πάντα διατηρεί τυχαιότητα. Αποδεικνύεται ότι μετά την εφαρμογή της split σε ένα random binary search tree τότε και τα δύο υποδένδρα που παράγονται αποτελούν random binary search trees. Άρα καταλήγουμε στο εξής: αν σε ένα random binary search tree εισάγουμε ένα στοιχείο, τότε παράγεται ένα νέο random binary search tree που περιέχει αυτό το στοιχείο[1].

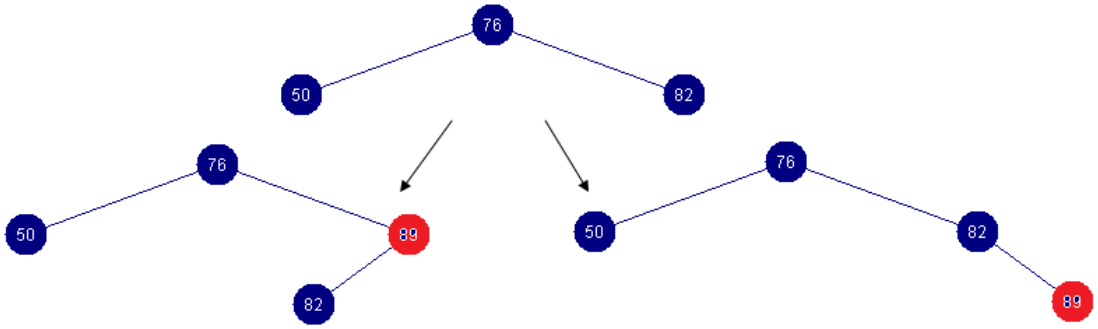
Στο σχήμα 2.3 βλέπουμε 9 διαδοχικές ενθέσεις στοιχείων σε ένα αρχικά άδειο δένδρο. Με κόκκινο χρώμα εικονίζεται ο κόμβος με το νέο στοιχείο. Όπως παρατηρούμε υπάρχει περίπτωση το στοιχείο που εισάγεται να γίνει ρίζα του δέντρου, όπως οι κόμβοι 50, 76, 89 57 και 85, να γίνει ρίζα κάποιου υποδένδρου, όπως οι κόμβοι 34 και 37 ή να εισαχθεί σε θέση φύλλου όπως ο κόμβος 95.



Σχήμα 2.3: (α)-(θ) Διαδοχικές ενθέσεις των 82, 50, 76, 89, 95, 57, 34, 37, 85

Να τονίσουμε σε αυτό το σημείο ότι αυτό το δένδρο που προκύπτει είναι ένα από τα πολλά που μπορούν να προκύψουν, καθώς λόγω των τυχαίων επιλογών η θέση που μπορεί να καταλάβει ένα στοιχείο μετά την εισαγωγή του στο δένδρο δεν είναι πάντα η ίδια, δηλαδή ένα στοιχείο μπορεί να γίνει ρίζα του δένδρου, ρίζα ενός από τα υποδένδρα ή να μπει σε μια θέση-φύλλο.

Για παράδειγμα στο στάδιο όπου εισάγεται το στοιχείο με κλειδί 89, τα άλλα δύο πιθανά δένδρα που θα μπορούσαν να προκύψουν εκτός από αυτό του σχήματος (δ) είναι αυτά που φαίνονται στο σχήμα 2.4.



Σχήμα 2.4: Πιθανά δένδρα μετά την ένθεση του στοιχείου 89

Διαγραφή στοιχείου

Εφαρμόζουμε τον κάτωθι αλγόριθμο

Αλγόριθμος 3 Διαγραφή

```
Node delete ( int x, Node T ) {
        Node Aux;

        if ( T == □ )
                return □;
        if ( x < T→key )
                T→left = delete ( x, T→left );
        else if ( x > T→key )
                T→right = delete ( x, T→right );
        else {
                Aux = join ( T→left, T→right );
                free_node ( T );
                T = Aux;
        }
        return T;
}
```

Αλγόριθμος 4 Συνένωση των δύο RBSTs

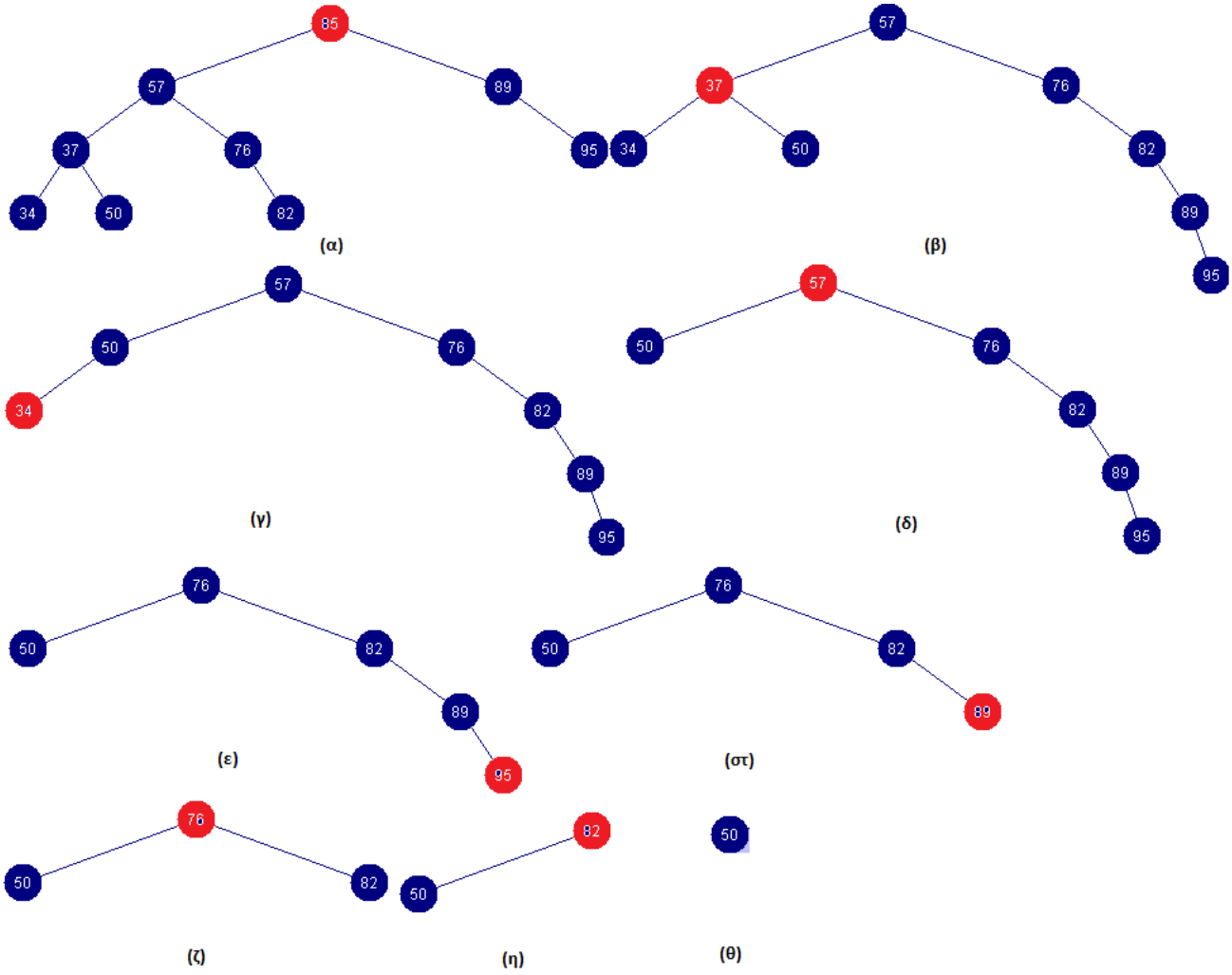
```
Node join ( Node L, Node R ) {
        int m, n, r, total;

        m = L→size;
        n = R→size;
        total = m + n;
```

```
        if ( total == 0 ) return □;
        r = random ( 0, total – 1 );
        if ( r < m ) {
                L→right = join ( L→right, R );
                return L;
        }
        else {
                R→left = join ( L, R→left );
                return R;
        }
}
```

Ο αλγόριθμος της διαγραφής χρησιμοποιεί την μέθοδο join, η οποία συνενώνει τα δύο υποδένδρα που προκύπτουν μετά τη διαγραφή της ρίζας, σε ένα καινούριο δέντρο θέτοντας ως ρίζα, τη ρίζα του δεξιού ή του αριστερού υποδένδρου.

Όπως και στο αλγόριθμο εισαγωγής, έτσι και στον αλγόριθμο διαγραφής διατηρείται η τυχαιότητα κι αυτό προέρχεται από την ιδιότητα της join. Η συνένωση δύο τυχαίων δυαδικών δένδρων αναζήτησης παράγει ένα τυχαίο δυαδικό δένδρο αναζήτησης. Άρα και εδώ αν σε ένα random binary search tree διαγραφεί ένα στοιχείο τότε παράγεται ένα νέο random binary search tree που περιέχει τα ίδια στοιχεία εκτός από αυτό που διαγράφηκε[1].

Στο σχήμα 2.5 βλέπουμε 8 διαδοχικές αποσβέσεις στοιχείων από ένα δένδρο. Με κόκκινο χρώμα εικονίζεται ο κόμβος που πρόκειται να διαγραφεί. Όπως παρατηρούμε υπάρχει περίπτωση το στοιχείο που διαγράφεται να είναι η ρίζα του δέντρου, όπως στην περίπτωση που διαγράφονται οι κόμβοι 85, 57, 76 και 82 όπου επιλέγεται ως ρίζα του νέου δέντρου η ρίζα του αριστερού ή δεξιού υποδένδρου, το στοιχείο που διαγράφεται να είναι κάποιος εσωτερικός κόμβος, όπως ο κόμβος οπότε πάλι έχουμε αλλαγές στο δέντρο και ο κόμβος που διαγράφεται να είναι φύλλο. όπως οι κόμβοι 34, 95 και 89 οπότε δεν γίνεται κλήση της join.



Σχήμα 2.5: (α)-(θ) Διαδοχικές αποσβέσεις των 85, 37, 34, 57, 95, 89, 76, 82

2.2.2 Random Root Model

Το ***random root model (μοντέλο τυχαίας ρίζας)*** αναφέρεται σε ένα τυχαίο δυαδικό δέντρο αναζήτησης όπου κάθε κόμβος του χαρακτηρίζεται από ένα κλειδί. Σε αυτό το μοντέλο τυχαία επιλέγεται ένα από τα n κλειδιά, ώστε να γίνει ρίζα του δέντρου. Έτσι τα δεξιά και αριστερά υποδένδρα θα δημιουργήσουν αναδρομικά τυχαία δένδρα αναζήτησης για τα υπόλοιπα κλειδιά. Αυτό το μοντέλο έχει αρκετές ομοιότητες με το randomized binary search tree, όπως για παράδειγμα ότι και σε αυτό οι αλγόριθμοι ένθεσης/διαγραφής βασίζουν τις τυχαίες επιλογές τους στο μέγεθος των δένδρων.

Εισαγωγή στοιχείου

Εφαρμόζουμε τον κάτωθι αλγόριθμο

Αλγόριθμος 5 Εισαγωγή

```
Node Generic-Insert ( int x, Node T ) {
        if ( Is-Null ( T ) )
                T←NewNode( x );
        else if ( newRoot (x, T) )
                Root-Insert ( x, T );
        else if ( x < T→key )
                Generic-Insert ( x, T→left );
        else
                Generic-Insert( x, T→right );
}
```

Αλγόριθμος 6 Εισαγωγή στη ρίζα

```
Node Root-Insert ( int x, Node T ) {
        if ( Is-Null ( T ) )
                T←NewNode ( x );
        else if ( x < T→key )
                Root-Insert ( x, T→ left );
                Rotate-Right ( T );
        else
                Root-Insert( x, T→right );
                Rotate-Left ( T );
}
```

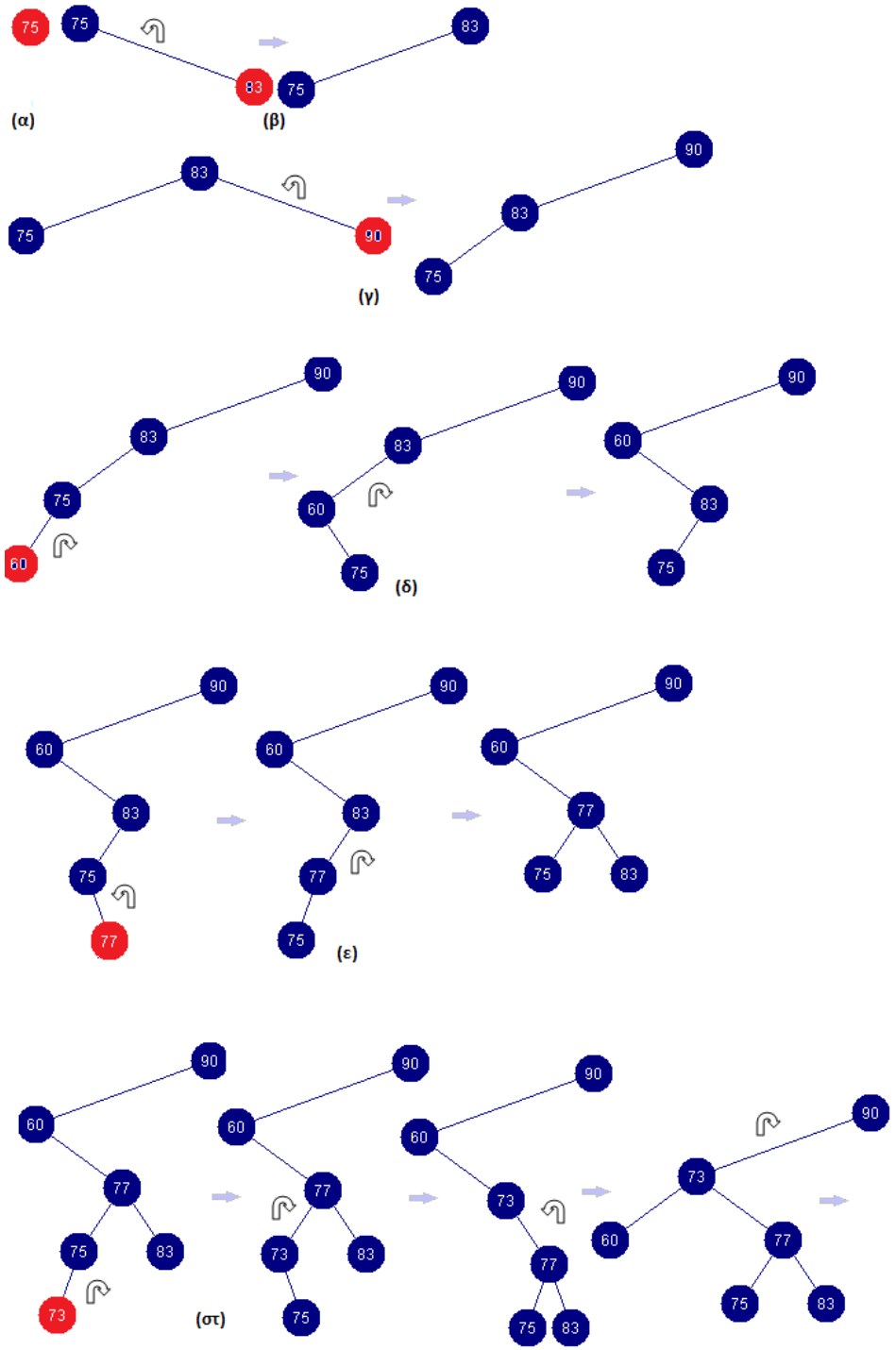
Η διαδικασία εισαγωγής ενός στοιχείου ξεκινάει ελέγχοντας αν το νέο στοιχείο πρόκειται να γίνει η ρίζα του δέντρου. Αυτός ο έλεγχος πραγματοποιείται με την κλήση της **newRoot** . Αν το αποτέλεσμα της κλήσης της newRoot είναι μηδέν τότε το αντικείμενο θα εισαχθεί αναδρομικά στο κατάλληλο υποδένδρο. Αν όμως το αποτέλεσμα είναι θετικό, που αυτό σημαίνει ότι το αντικείμενο πρόκειται να γίνει η νέα ρίζα του δένδρου, τότε το δένδρο πρέπει να διασπαστεί στο σημείο που θα εισαχθεί και τα δύο υποδένδρα που προκύπτουν θα είναι το αριστερό και δεξί υποδένδρο της νέας ρίζας. Αυτό υλοποιείται μέσω μιας απλής αναδρομής που περιλαμβάνει περιστροφές, δηλαδή το αντικείμενο εισάγεται κανονικά στην κατάλληλη θέση ως φύλλο και στη συνέχεια περιστρέφεται μέχρι να φτάσει στη θέση

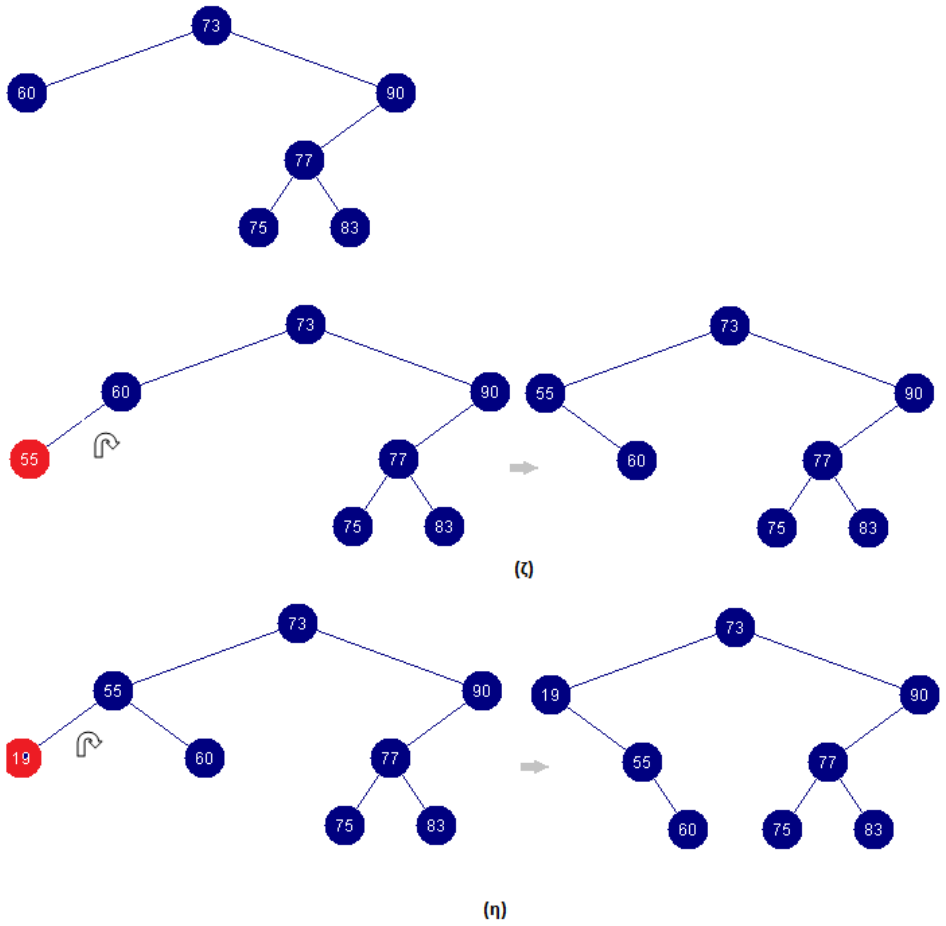
της ρίζας. Οι διαδικασίες Rotate-Right και Rotate-Left είναι οι κλασικές διαδικασίες περιστροφής δένδρου.

Η υλοποίηση της newRoot σχετίζεται με την πληροφορία για το μέγεθος του δένδρου ή του υποδένδρου που εξετάζουμε κάθε φορά, η οποία αποθηκεύεται στον εκάστοτε κόμβο. Υλοποιείται ως εξής σε αυτό το μοντέλο:

```
int newRoot ( Node T ){
        r = random( 0, T→size );
        if ( r == T→size )
                return 1;
        else
                return 0;
}
```

Στο σχήμα 2.6 βλέπουμε 8 διαδοχικές ενθέσεις στοιχείων σε ένα αρχικά άδειο δένδρο. Με κόκκινο χρώμα εικονίζεται ο κόμβος με το νέο στοιχείο. Στο σχήμα φαίνονται επίσης και οι διαδοχικές περιστροφές που γίνονται σε περίπτωση που το στοιχείο θα γίνει ρίζα του δένδρου ή κάποιου υποδένδρου. Για παράδειγμα τα στοιχεία 83, 90 και 73 πρόκειται να μπουν στη θέση της ρίζας του δένδρου όταν εισάγονται, οπότε αρχικά μπαίνουν σε θέση-φύλλο και περιστρέφονται μέχρι να φτάσουν στη θέση-ρίζα.


75
75
83
83
75
(α)
(β)
83
75
90
83
75
90
(γ)
90
83
75
60
90
83
60
75
(δ)
90
60
83
75
77
90
60
83
77
75
(ε)
90
60
77
75
83
73
(στ)



Σχήμα 2.6(α)-(η): Διαδοχικές ενθέσεις των 75, 83, 90, 60, 77, 73, 55, 19

Διαγραφή στοιχείου

Εφαρμόζουμε τον κάτωθι αλγόριθμο

Αλγόριθμος 7 Διαγραφή

```
Node Generic-Delete ( int x, Node T ) {
        if ( T = null )
                return;
        if ( x < T→key )
                Generic-Delete( x, T→left );
        else if ( x > T→key )
                Generic-Delete( x, T→right );
        else
                Root-Delete( T );
}
```

Αλγόριθμος 8 Διαγραφή της ρίζας

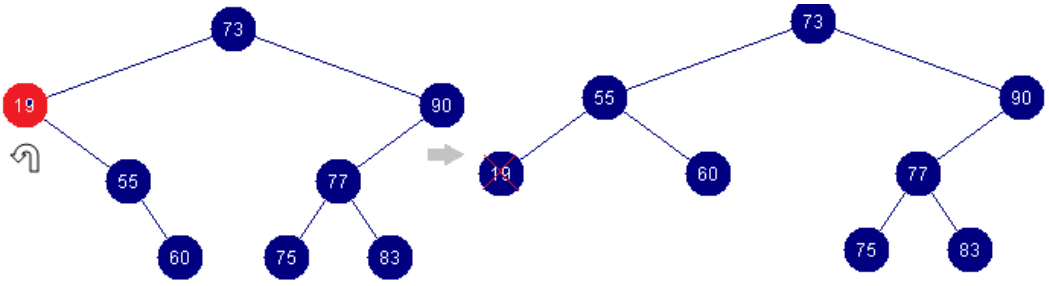
```
Node Root-Delete ( int x, Node T  ) {
        if (  IsLeaf ( T )  )
                T←null;
        else if ( Lchild-Wins ( T ) )
                Rotate-Right ( T );
                Root-Delete ( T→right );
        else
                Rotate-Left ( T );
                Root-Delete ( T→left );
}
```

Η διαδικασία της διαγραφής ενός στοιχείου με κλειδί x, ξεκινάει με την αναζήτηση του κόμβου που στεγάζει αυτό το κλειδί. Αυτός ο κόμβος υπάρχει περίπτωση να είναι η ρίζα του δένδρου ή η ρίζα κάποιου υποδένδρου. Σε αυτήν την περίπτωση πρέπει τα δύο υποδένδρα που προκύπτουν μετά την διαγραφή, να συνενωθούν σε ένα. Αυτό επιτυγχάνεται με τη χρήση της Root-Delete η οποία περιστρέφει αναδρομικά τον κόμβο μέχρι να φτάσει σε θέση φύλλου και τον διαγράφει. Το αν η περιστροφή θα είναι δεξιά ή αριστερή καθορίζεται από το αποτέλεσμα της κλήσης της Lchild-Wins. Αν το αποτέλεσμα της είναι 0, τότε θα γίνει αριστερή περιστροφή, αλλιώς θα γίνει δεξιά.

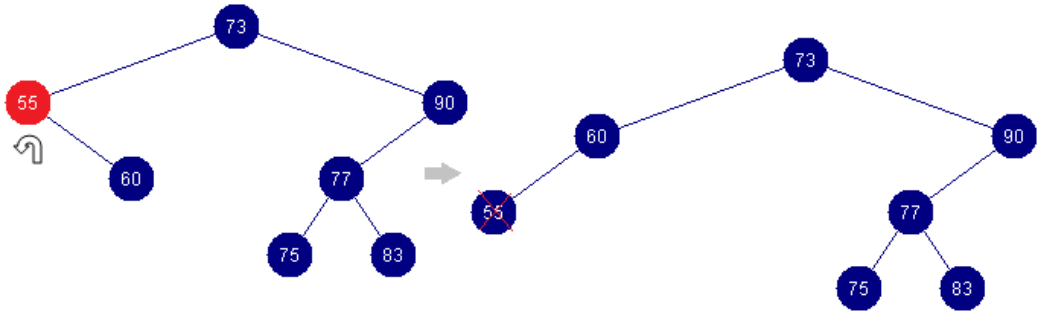
Η υλοποίηση της Lchild-Wins σχετίζεται με την πληροφορία για το μέγεθος του δένδρου ή του υποδένδρου που εξετάζουμε κάθε φορά, η οποία αποθηκεύεται στον εκάστοτε κόμβο. Υλοποιείται ως εξής σε αυτό το μοντέλο:

```
int Lchild-Wins ( Node T ) {
        if ( T→left→size > T→right→size)
                return 1;
        else
                return 0;
}
```

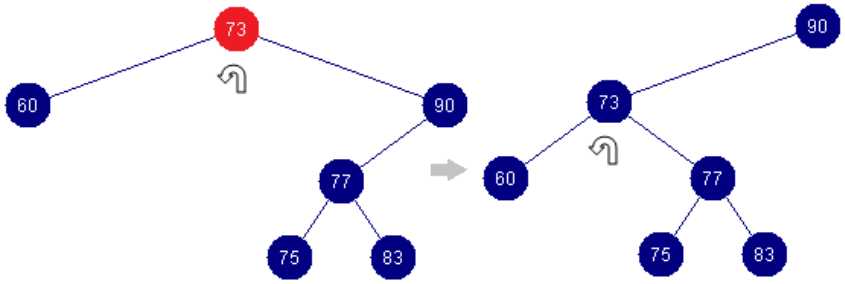
Στο σχήμα 2.7 βλέπουμε 8 διαδοχικές αποσβέσεις στοιχείων. Με κόκκινο χρώμα εικονίζεται ο κόμβος με το στοιχείο που πρόκειται να διαγραφεί. Το στοιχείο αν δε βρίσκεται σε θέση φύλλο, με διαδοχικές περιστροφές το «κατεβάζουμε» σε θέση φύλλο και το διαγράφουμε. Όταν το στοιχείο αφήσει την παλιά του θέση, την καταλαμβάνει η ρίζα του υποδένδρου του με το μεγαλύτερο μέγεθος. Για παράδειγμα, όταν πρόκειται να διαγραφεί το στοιχείο 73, επιλέγεται το στοιχείο 90 καθώς στο υποδένδρο που είναι ρίζα υπάρχουν ακόμη τρία στοιχεία ενώ στο αριστερό υποδένδρο του 73, υπάρχει μόνο ένα. Έτσι το 73 υποβιβάζεται σε θέση-φύλλο και διαγράφεται.


73
19
90
55
77
60
75
83
73
55
90
19
60
77
75
83

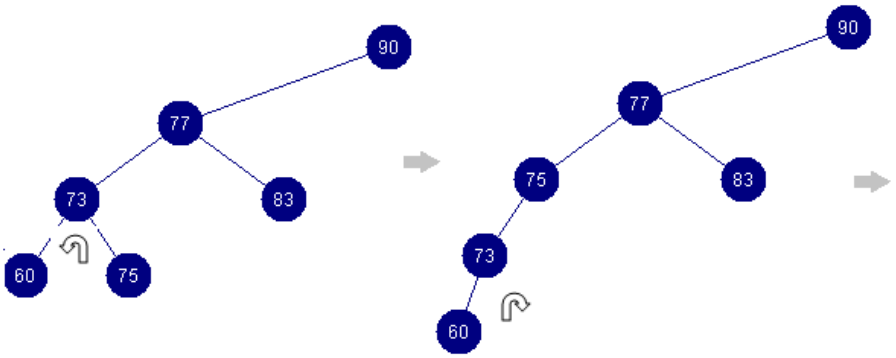
(α)

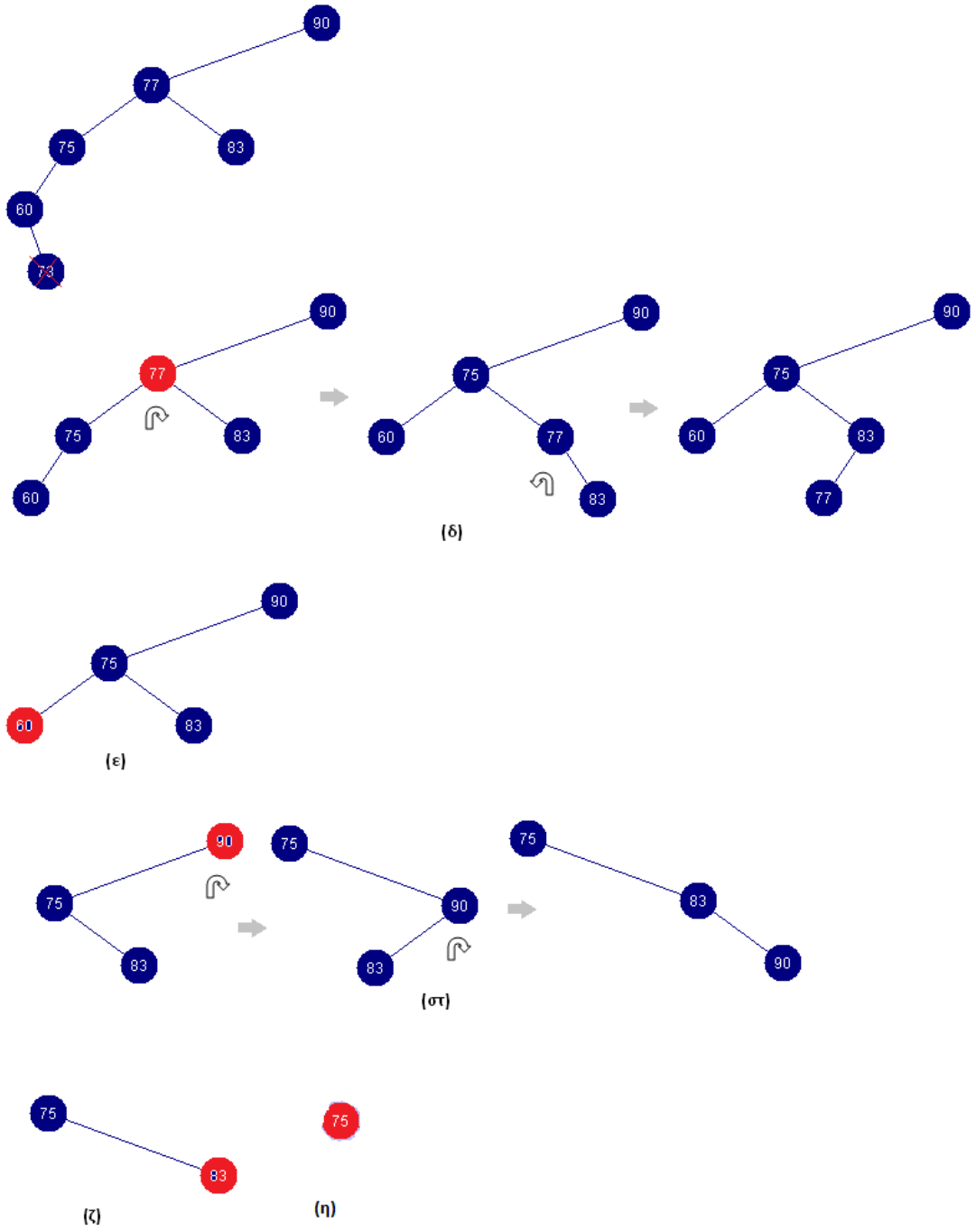

73
55
90
60
77
75
83
73
60
90
55
77
75
83

(β)


73
60
90
77
75
83
90
73
60
77
75
83

(γ)


90
77
73
83
60
75
90
77
75
83
73
60



Σχήμα 2.7(α)-(η): Διαδοχικές αποσβέσεις των 19, 55, 73, 77, 60, 90, 83, 75

2.2.3 Priority Model

Το ***priority model (μοντέλο προτεραιότητας)*** αποτελεί ένα δυαδικό δέντρο αναζήτησης που ουσιαστικά λειτουργεί ως treap, καθώς τα στοιχεία που το αποτελούν, χαρακτηρίζονται εκτός από το κλειδί, και από μια προτεραιότητα. Οι

προτεραιότητες p_1, …, p_n είναι τυχαίοι αριθμοί, έστω επιλεγμένοι από την ομοιόμορφη κατανομή πιθανότητας στο διάστημα [0, 1]. Έτσι κατασκευάζεται το treap που αποτελείται από τα ζευγάρια { (a_1, p_1),…, (a_n, p_n) }. Σε κάθε κόμβο αποθηκεύεται ένα ζευγάρι (α_i, p_i) το οποίο ακολουθεί τη συμμετρική διάταξη ως προς τα κλειδιά α_i και σέβεται τη διάταξη σωρού μέγιστου στοιχείου ως προς τις προτεραιότητες p_i. Οι αλγόριθμοι εισαγωγής/διαγραφής βασίζουν τις τυχαίες επιλογές τους στα priorities.

Εισαγωγή στοιχείου

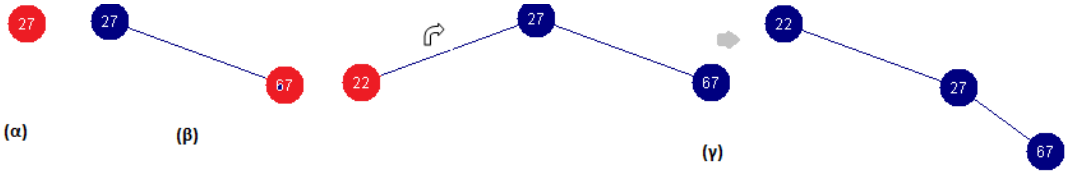
Ο αλγόριθμος της εισαγωγής είναι ίδιος με αυτόν του random root model. Η μοναδική διαφοροποίηση που έχουν είναι στην υλοποίηση της ρουτίνας newRoot, η οποία σχετίζεται με την προτεραιότητα του κάθε στοιχείου, κάτι που δεν υπάρχει στο προηγούμενο μοντέλο. Έτσι, σε αυτό το μοντέλο ο έλεγχος για το αν το νέο στοιχείο που εισάγεται θα γίνει ρίζα του δένδρου γίνεται ως εξής:

```
int newRoot (Item x, Node T) {
        if (x→priority > T→priority)
                return 1;
        else
                return 0;
}
```

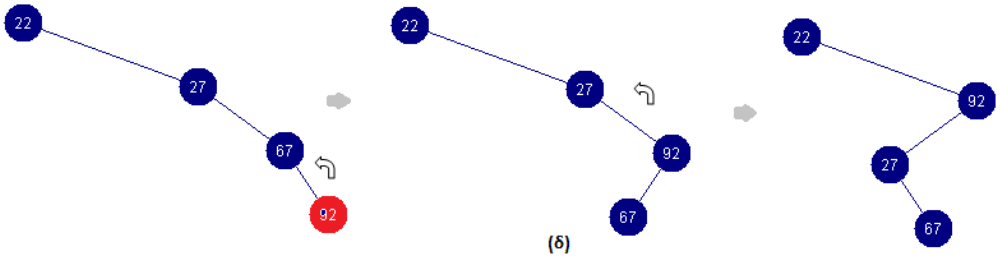
Στο σχήμα 2.8 φαίνονται οι διαδοχικές ενθέσεις 8 στοιχείων με κλειδιά και προτεραιότητες, που φαίνονται στον πίνακα

Keys	27	67	22	92	58	57	21	82
Priorities	0.13	0.05	0.63	0.18	0.25	0.50	0.75	0.78

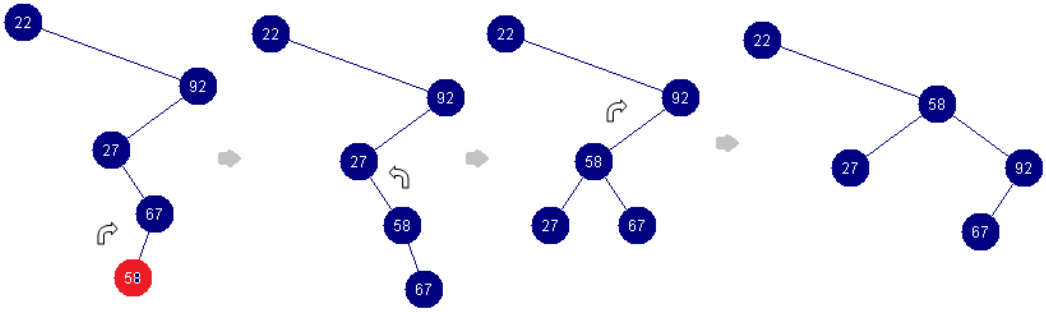
Με κόκκινο χρώμα εικονίζεται ο κόμβος με το νέο στοιχείο. Στο σχήμα φαίνονται επίσης και οι διαδοχικές περιστροφές που γίνονται σε περίπτωση που το στοιχείο θα γίνει ρίζα του δένδρου ή κάποιου υποδένδρου. Για παράδειγμα το στοιχείο 82 πρόκειται να γίνει η νέα ρίζα του δένδρου όταν εισάγεται, οπότε αρχικά μπαίνει σε θέση-φύλλο και περιστρέφεται μέχρι να φτάσει στη θέση-ρίζα.


27
27
67
22
27
67
22
27
67

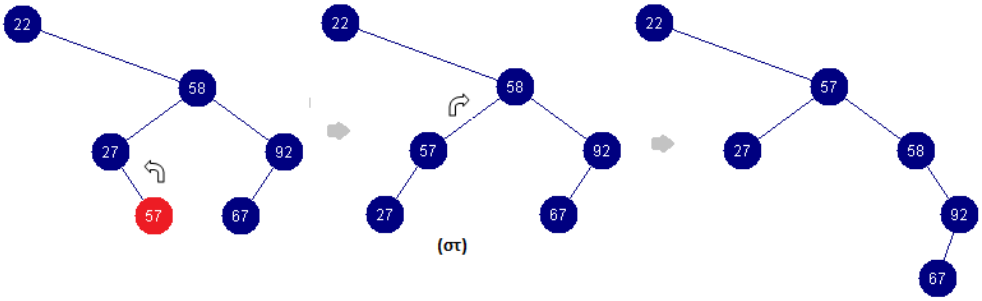
(α) (β) (γ)


22
27
67
92
22
27
92
67
22
92
27
67

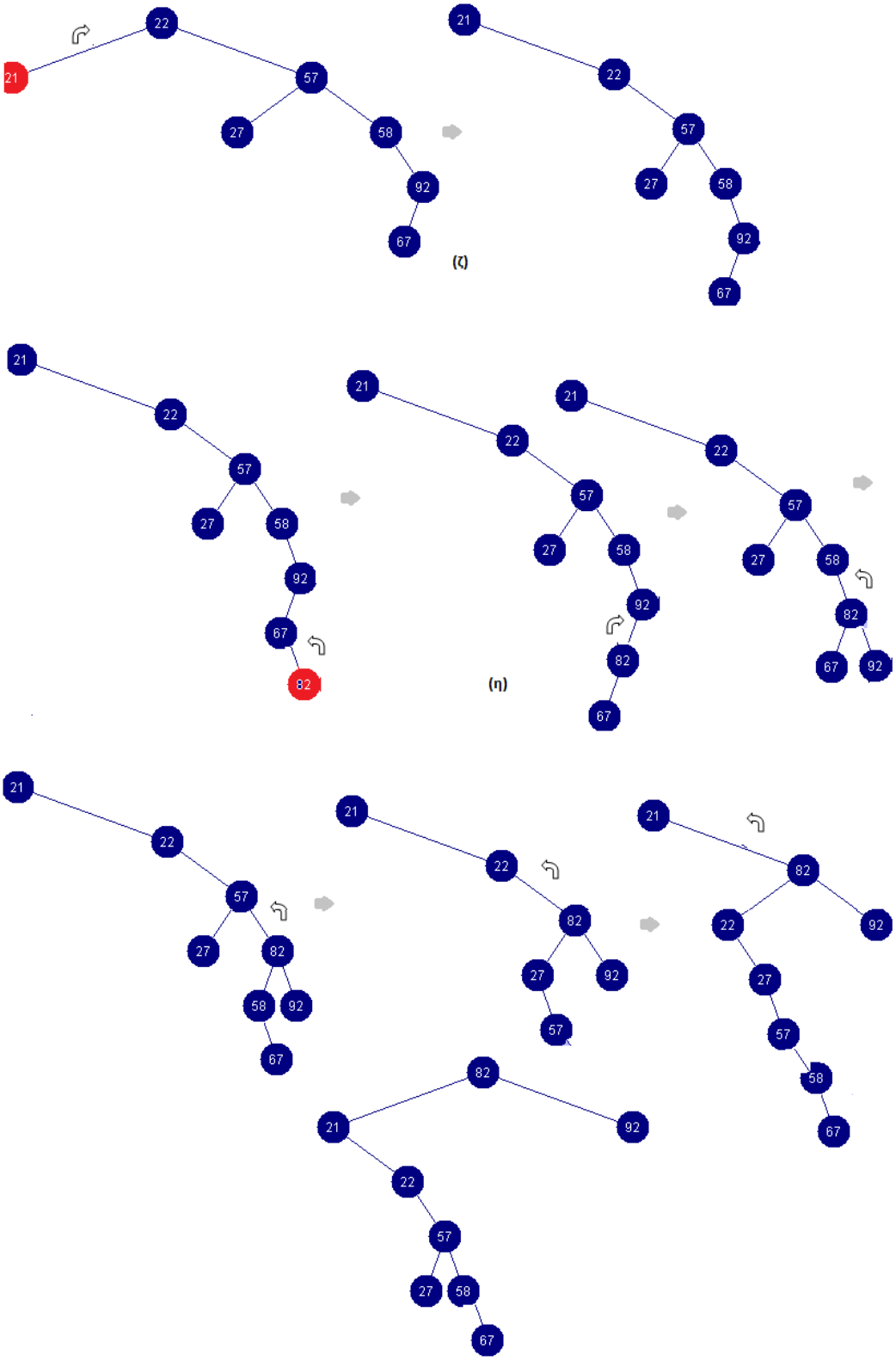
(δ)


22
92
27
67
58
22
92
27
58
67
22
92
58
27
67
22
58
27
92
67

(ε)


22
58
27
92
57
67
22
58
57
92
27
67
22
57
27
58
92
67

(στ)



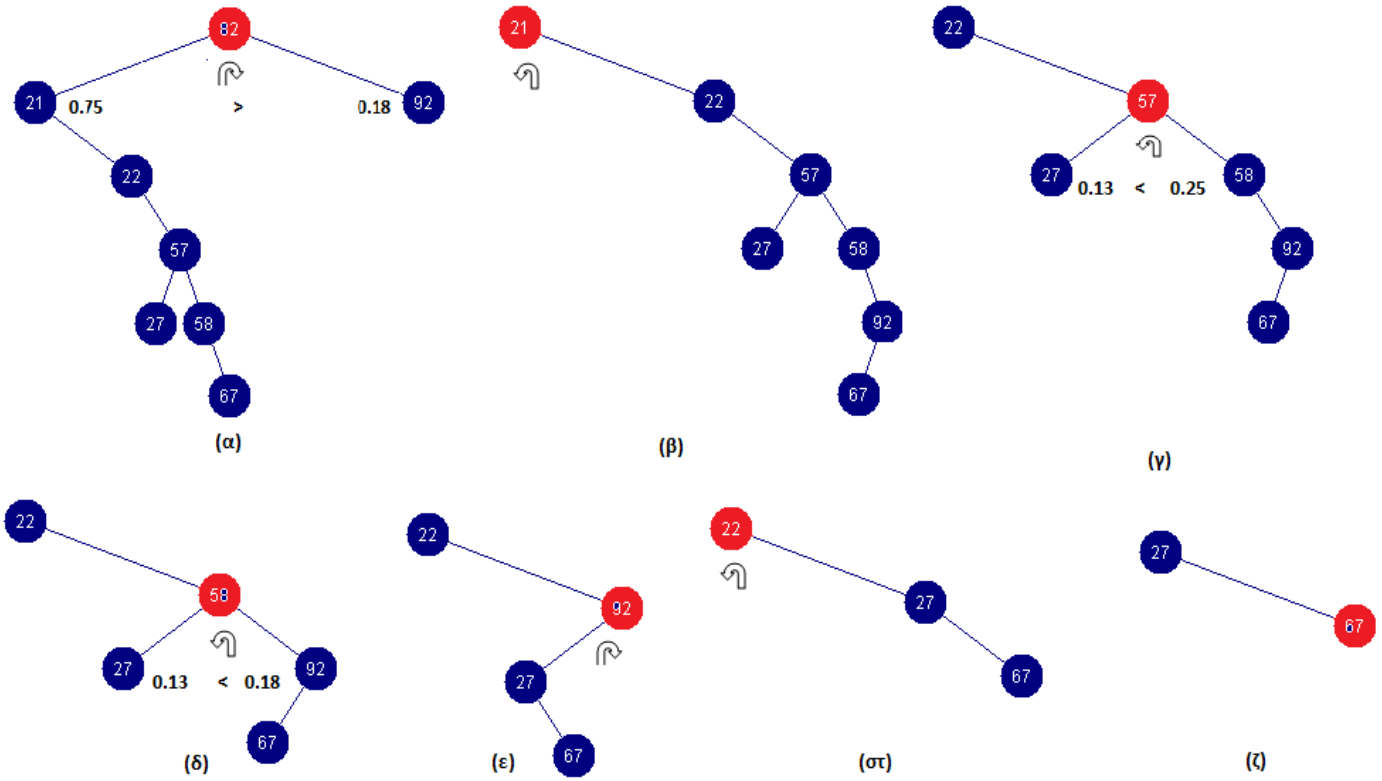
Σχήμα 2.8(α)-(η): Διαδοχικές ενθέσεις των 27, 67, 22, 92, 58, 57, 21, 82.

Διαγραφή στοιχείου

Ο αλγόριθμος της διαγραφής είναι ίδιος με αυτόν του random root model. Η μοναδική διαφοροποίηση που έχουν βρίσκεται στην υλοποίηση της ρουτίνας Lchild-Wins, η οποία σχετίζεται με την προτεραιότητα του κάθε στοιχείου, κάτι που δεν υπάρχει στο προηγούμενο μοντέλο. Έτσι, σε αυτό το μοντέλο ο έλεγχος για το αν το στοιχείο που θα διαγραφεί, θα περιστραφεί δεξιά ή αριστερά με σκοπό να αποκοπεί, γίνεται ως εξής:

```
int Lchild-Wins ( Node T ) {
        if ( T→left→priority > T→right→priority)
                return 1;
        else
                return 0;
}
```

Στο σχήμα 2.9 βλέπουμε την διαγραφή 7 στοιχείων από ένα δένδρο. Παρατηρούμε ότι ένα στοιχείο που πρόκειται να διαγραφεί, αντικαθίσταται από τη ρίζα του υποδένδρου της με το μεγαλύτερο priority. Στη συνέχεια περιστρέφεται διαδοχικά σε θέση-φύλλο, όπου και διαγράφεται.



Σχήμα 2.9(α)-(ζ): Διαδοχικές αποσβέσεις των 82, 21, 57, 58, 92, 22, 67, 27.

2.2.4 No Extra Space Model

Το μοντέλο αυτό είναι ισοδύναμο με τα προηγούμενα αλλά η διαφορά του με αυτά είναι ότι στους κόμβους του δένδρου δεν αποθηκεύεται κάποια επιπλέον πληροφορία, δηλαδή δεν αποθηκεύεται ούτε το μέγεθος του υποδένδρου που αντιστοιχεί στον κάθε κόμβο, ούτε η προτεραιότητα.

Εισαγωγή στοιχείου

Ο αλγόριθμος της εισαγωγής είναι ίδιος με αυτόν του random root model. Η μοναδική διαφοροποίηση που έχουν είναι στην υλοποίηση της ρουτίνας newRoot. Στα δύο προηγούμενα μοντέλα, η απόφαση για το αν το στοιχείο που εισαγόταν θα γινόταν ή όχι ρίζα του δένδρου, βασιζόταν είτε στο μέγεθος του δένδρου ή στην προτεραιότητα των κόμβων. Σε αυτό το μοντέλο όμως που δεν υπάρχει καμία επιπλέον πληροφορία αποθηκευμένη στους κόμβους, ας δούμε πως λαμβάνεται η απόφαση. Για το στοιχείο που πρόκειται να εισαχθεί επιλέγουμε τυχαία έναν αριθμό *p* από μια συνεχή κατανομή πιθανότητας, έστω D. Για κάθε στοιχείο που υπάρχει ήδη στο δένδρο, στο σύνολο M, επιλέγουμε από έναν τυχαίο αριθμό από την D. Διατρέχουμε το δένδρο και αν έστω και ένας από αυτούς τους αριθμούς είναι μεγαλύτερος από τον *p* τότε σταματάμε τη διέλευση του δένδρου και η κλήση της newRoot επιστρέφει 0. Αν όμως αυτό δεν ισχύει για κανένα στοιχείο του δένδρου, τότε επιστρέφει 1 και το στοιχείο εισάγεται στη ρίζα του δένδρου.

```
int newRoot ( Node T ){
        p = random();
        foreach k belongs to M
                if ( p < p.k )
                        return 1;
                else
                        return 0;
}
```

Κεφάλαιο 3

Απόδοση

3.1 Μετρήσεις

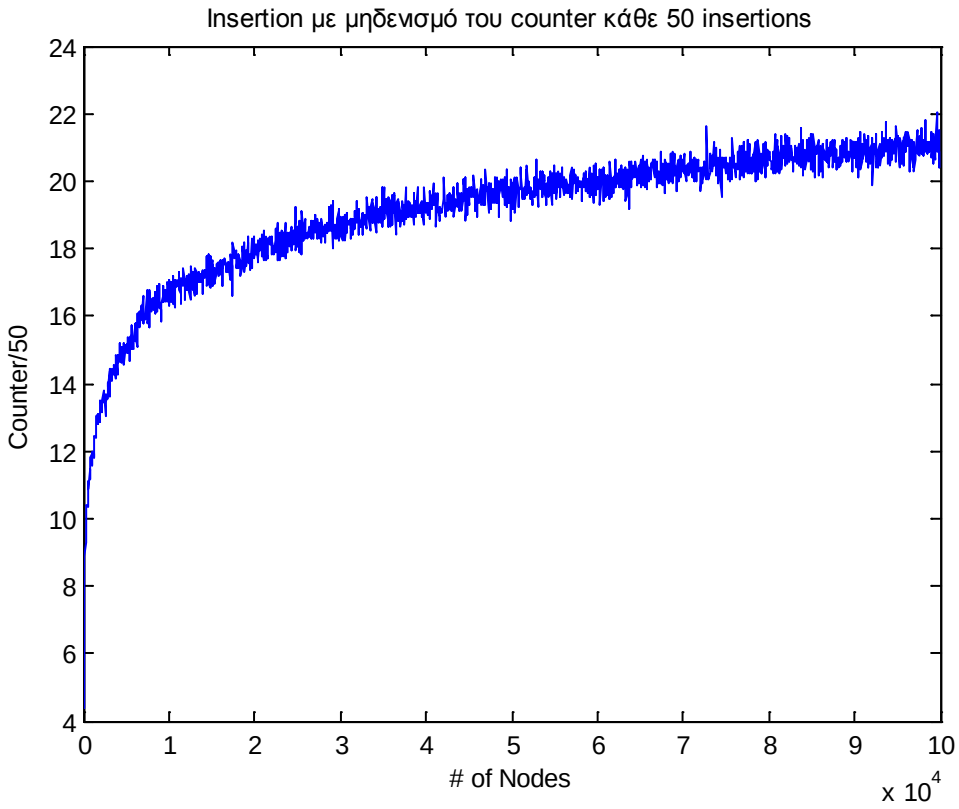
Στα απλά αζύγιστα δυαδικά δένδρα το κόστος οποιασδήποτε πράξης είναι γραμμικό σε σχέση με το ύψος τους. Αυτό που θέλουμε να αποδείξουμε είναι ότι στα τυχαία δυαδικά δένδρα αναζήτησης το κόστος όλων των πράξεων που εξετάσαμε είναι λογαριθμικό. Έτσι, προκειμένου να το αποδείξουμε, ακολουθήσαμε την εξής διαδικασία για όλους τους αλγορίθμους που περιγράψαμε:

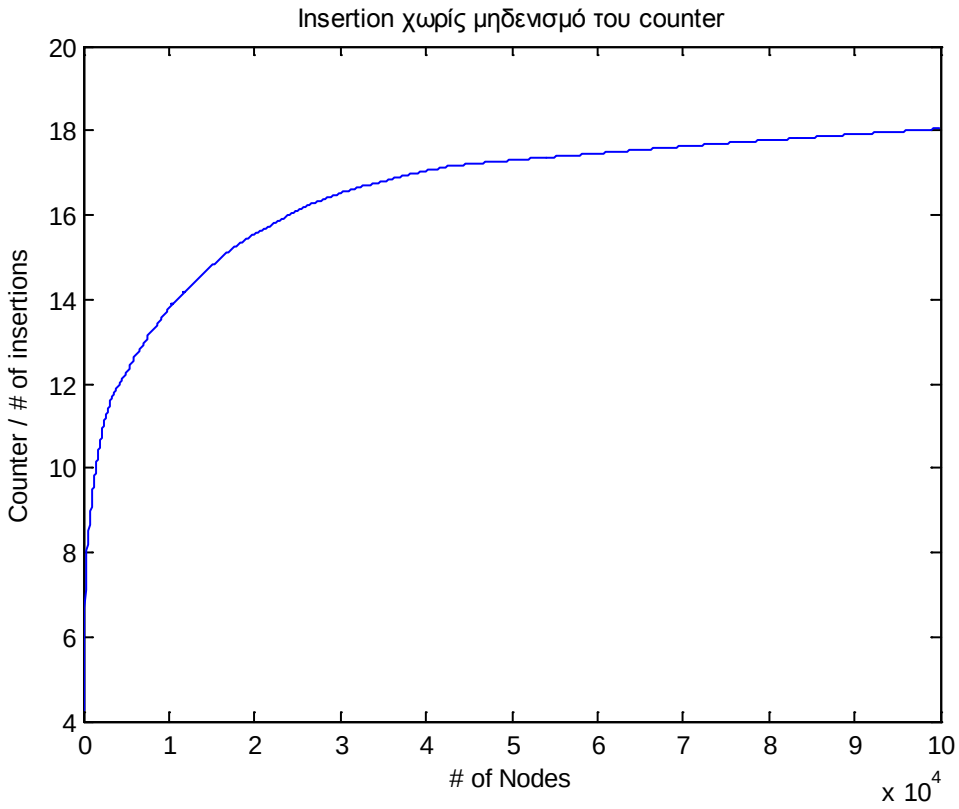
- Αρχικά, ξεκινήσαμε με τη διαδικασία της ένθεσης. Εισαγάγαμε σε ένα αρχικά άδειο δένδρο εκατό χιλιάδες στοιχεία.
- Με έναν μετρητή, μετρούσαμε το κόστος της ένθεσης κάθε πενήντα στοιχεία, δηλαδή μετρούσαμε τους κόμβους που συναντούσε το στοιχείο που εισαγόταν, σε δύο περιπτώσεις.
- Στην πρώτη περίπτωση μηδενίζαμε τον μετρητή κάθε πενήντα στοιχεία και βρίσκαμε το κόστος διαιρώντας με το πενήντα.
- Στη δεύτερη περίπτωση δε μηδενίζαμε τον μετρητή αλλά διαιρούσαμε με τον συνολικό αριθμό των πράξεων που προηγήθηκαν.
- Στη συνέχεια, προχωρήσαμε με τη διαδικασία της διαγραφής και διαγράψαμε τα εκατό χιλιάδες στοιχεία που ενθέσαμε προηγουμένως.
- Με έναν άλλο μετρητή μετρούσαμε το κόστος της διαγραφής κάθε πενήντα στοιχεία.
- Τα κόστη προέκυψαν όπως και στη διαδικασία της εισαγωγής.
- Με τη χρήση του προγράμματος Matlab, προέκυψαν οι παρακάτω γραφικές παραστάσεις καθώς εισαγάγαμε τα κόστη που προέκυψαν στα προηγούμενα βήματα.

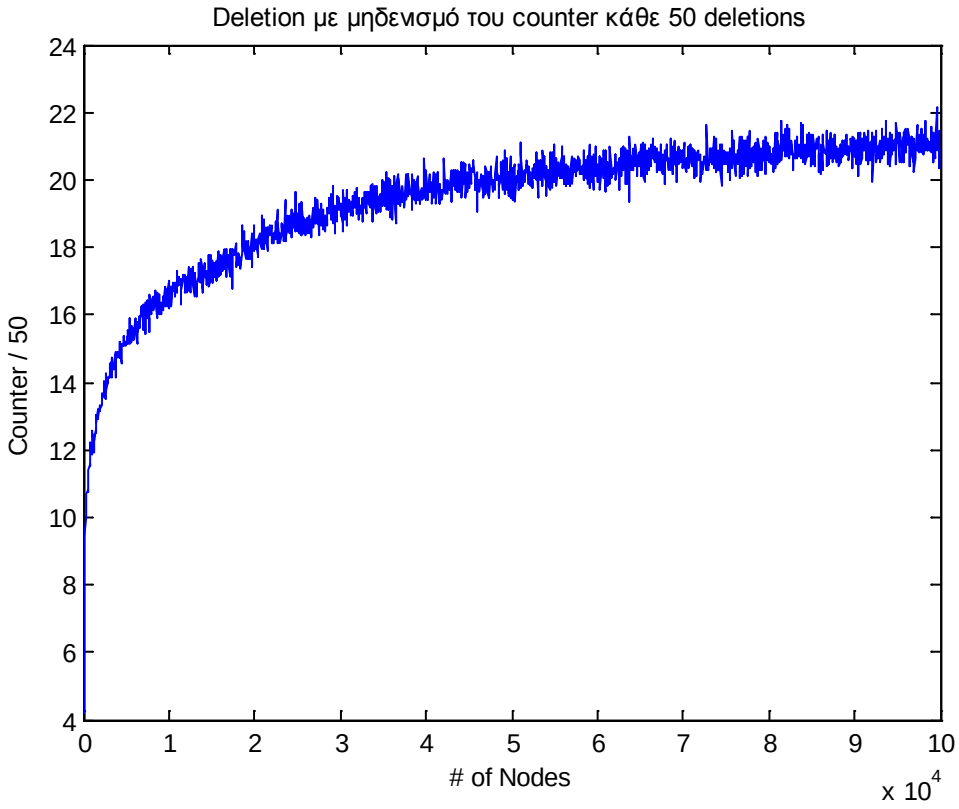
3.2 Γραφικές Παραστάσεις

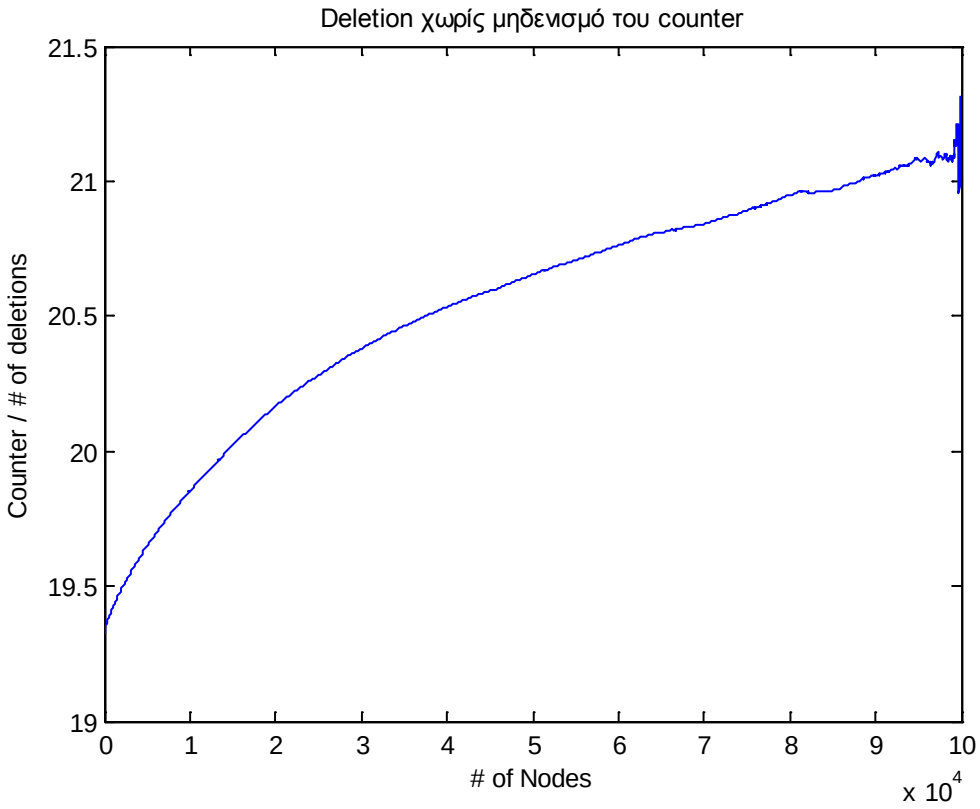
Παρατηρώντας τις γραφικές παραστάσεις που προέκυψαν, καταλήγουμε στο συμπέρασμα ότι το κόστος όλων των πράξεων στα τυχαία δένδρα αναζήτησης είναι λογαριθμικό, δηλαδή O(log *n*), όπου *n* είναι οι κόμβοι του δένδρου.

3.2.1 Randomized Binary Search Trees

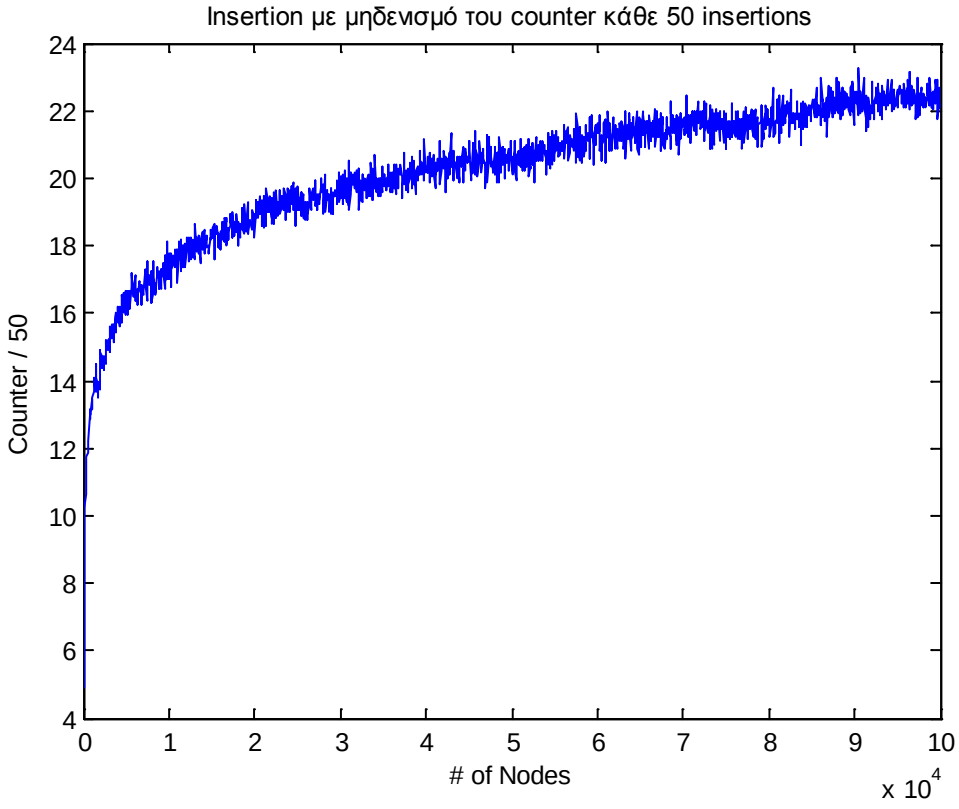


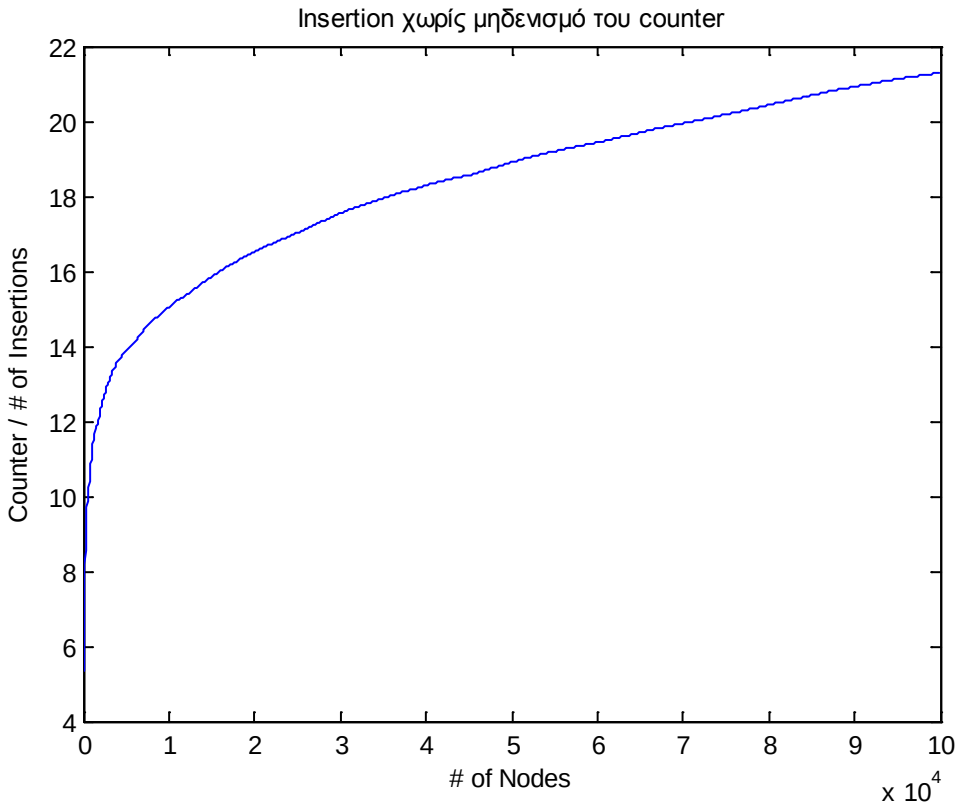

Insertion χωρίς μηδενισμό του counter
Counter / # of insertions
# of Nodes
x 10^4
20
18
16
14
12
10
8
6
4
0
1
2
3
4
5
6
7
8
9
10

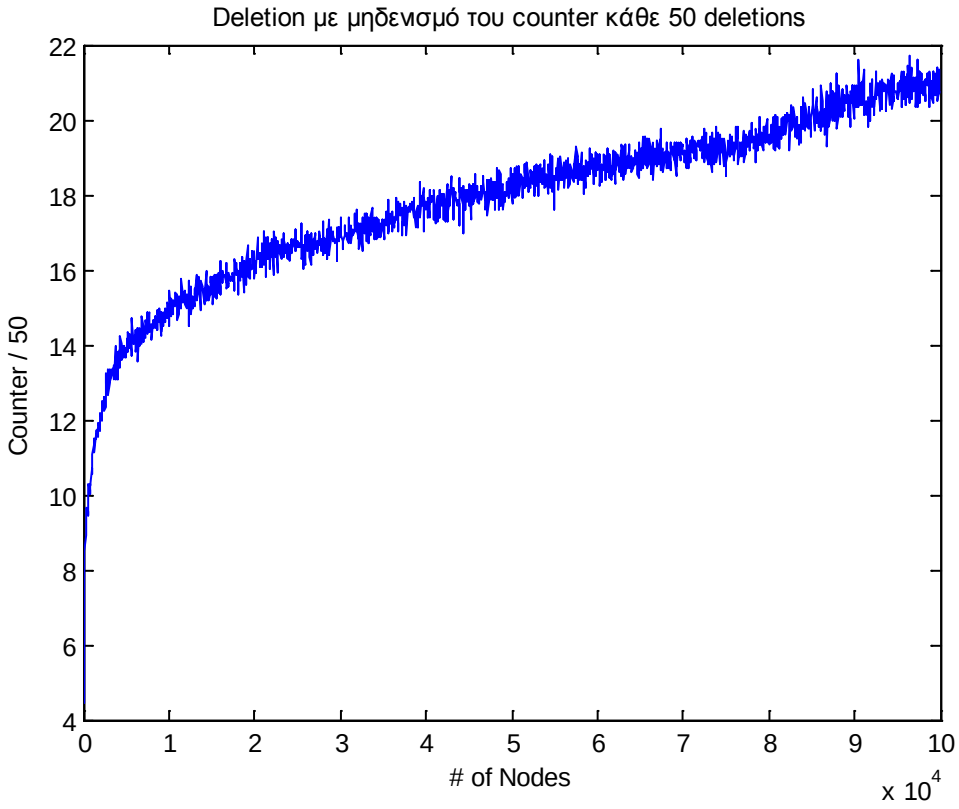

Deletion με μηδενισμό του counter κάθε 50 deletions
Counter / 50
# of Nodes
x 10^4
24
22
20
18
16
14
12
10
8
6
4
0
1
2
3
4
5
6
7
8
9
10

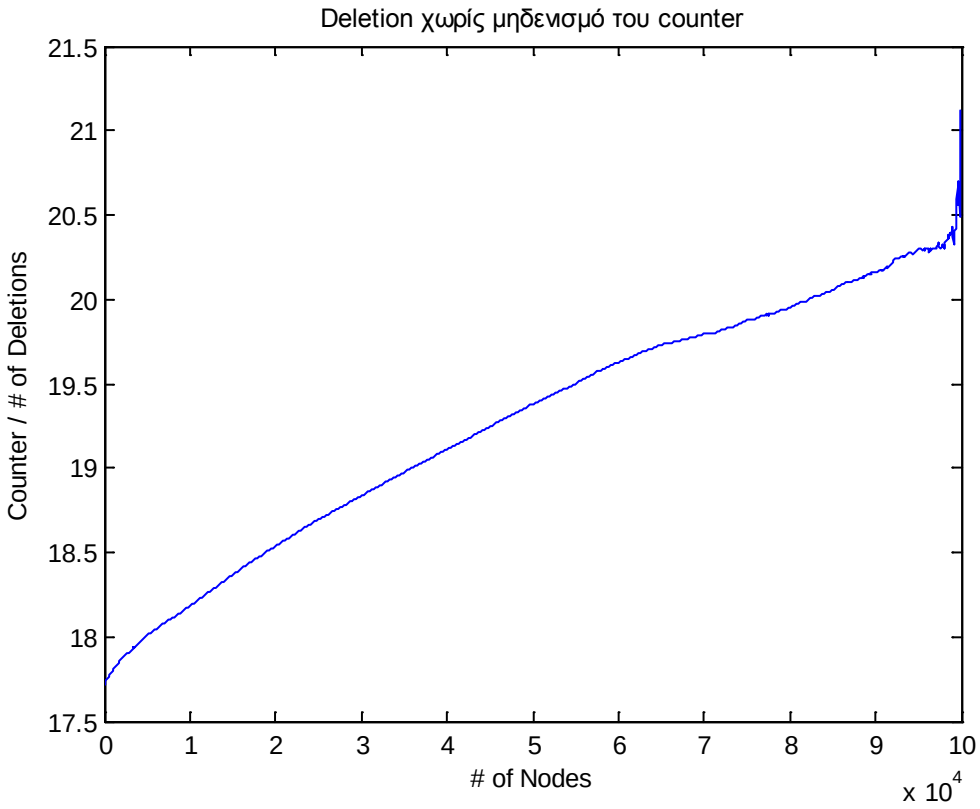


3.2.2 Random Root Model

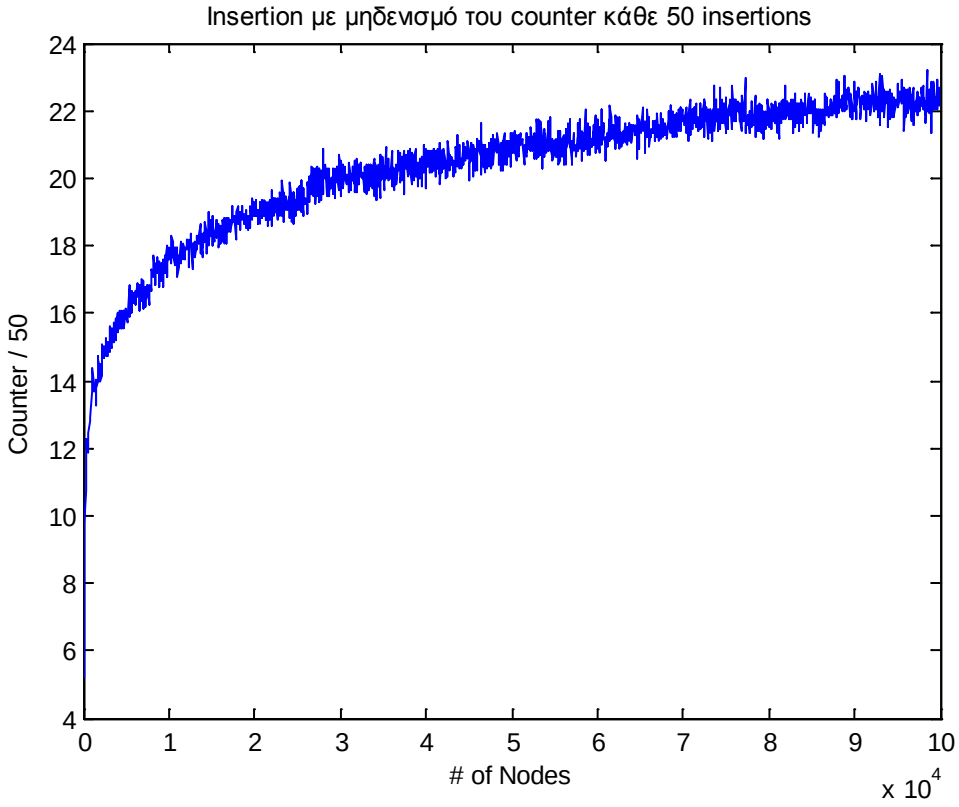


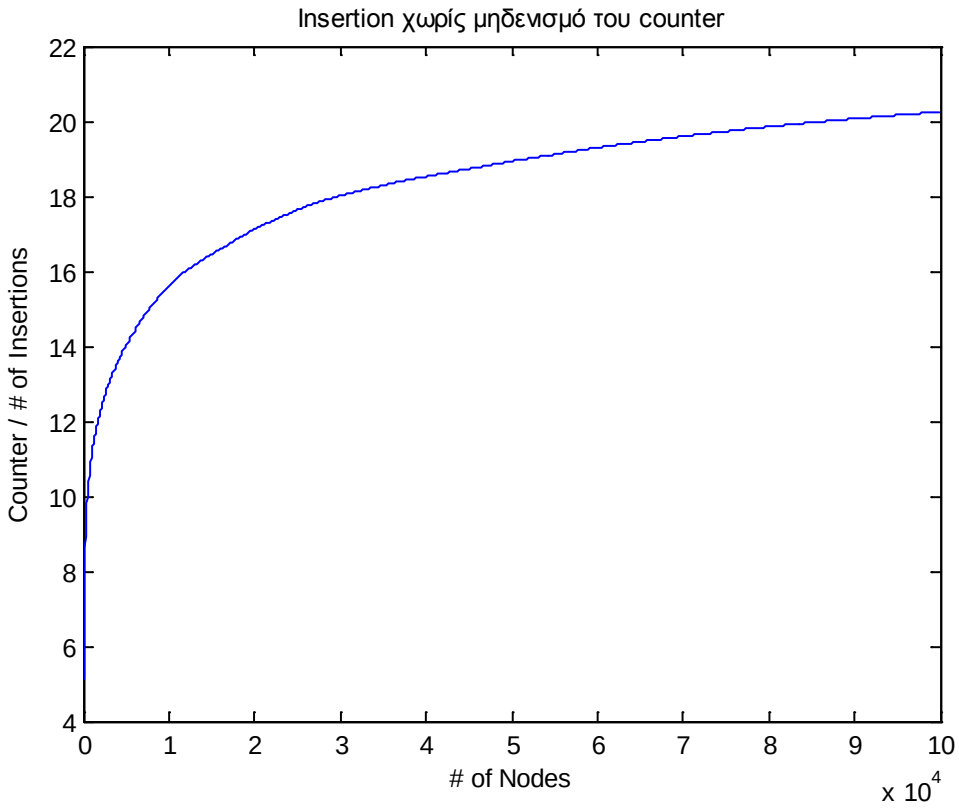

Insertion χωρίς μηδενισμό του counter
Counter / # of Insertions
# of Nodes
x 10^4
22
20
18
16
14
12
10
8
6
4
0
1
2
3
4
5
6
7
8
9
10

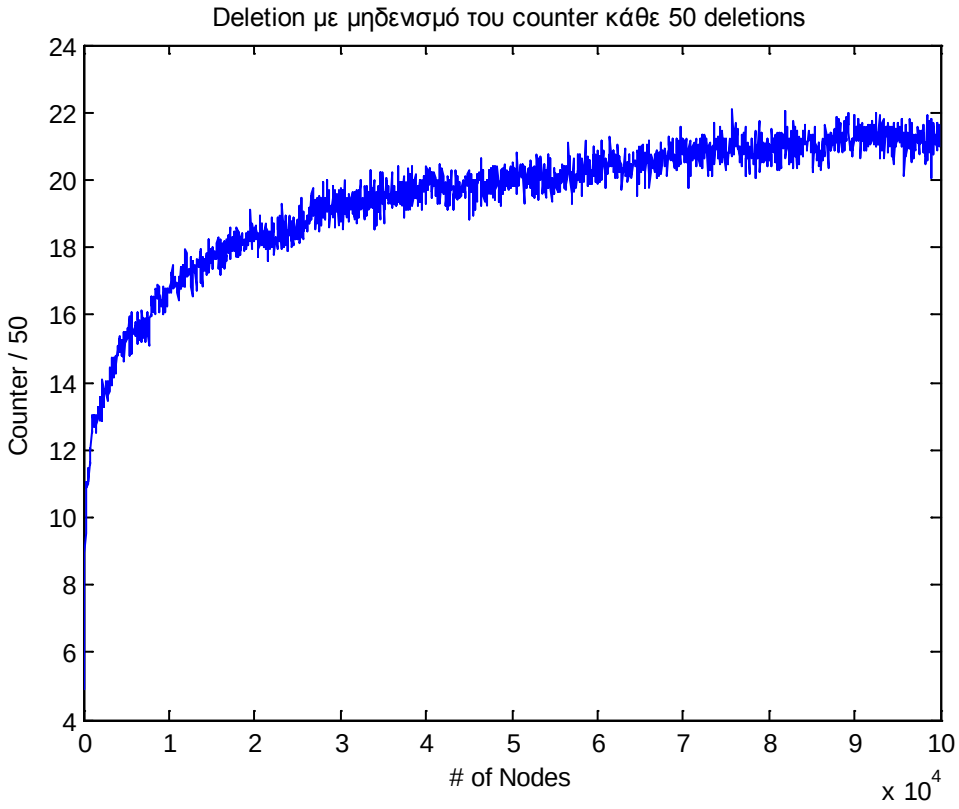

Deletion με μηδενισμό του counter κάθε 50 deletions
Counter / 50
# of Nodes
x 10^4
22
20
18
16
14
12
10
8
6
4
0
1
2
3
4
5
6
7
8
9
10

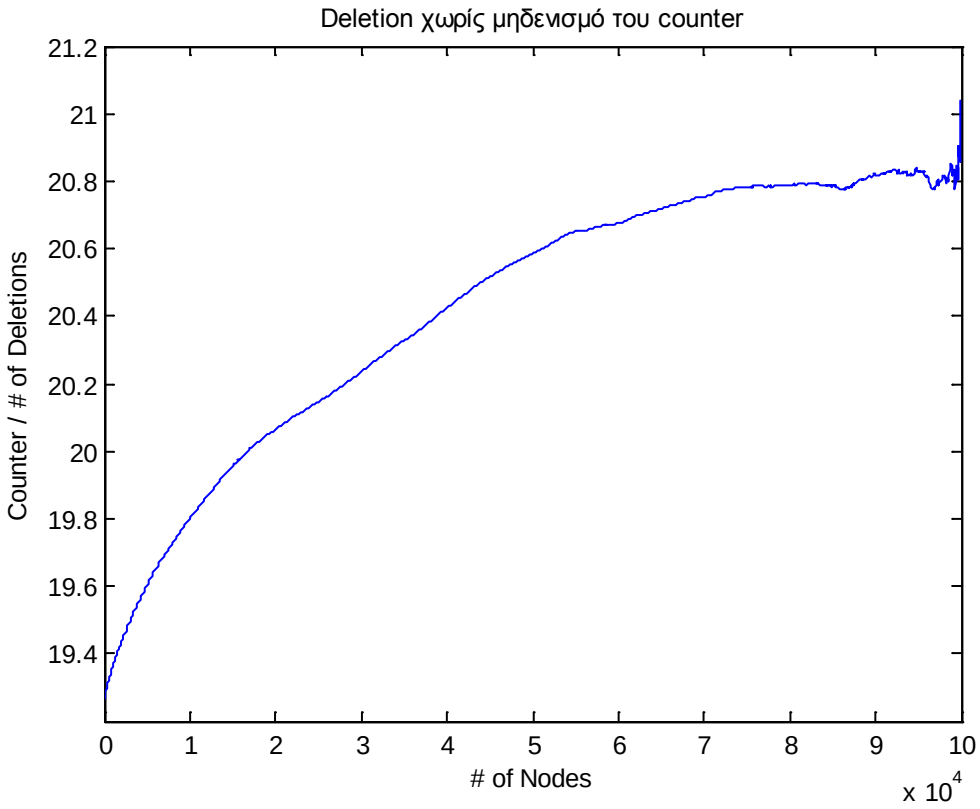


3.2.3 Priority Model

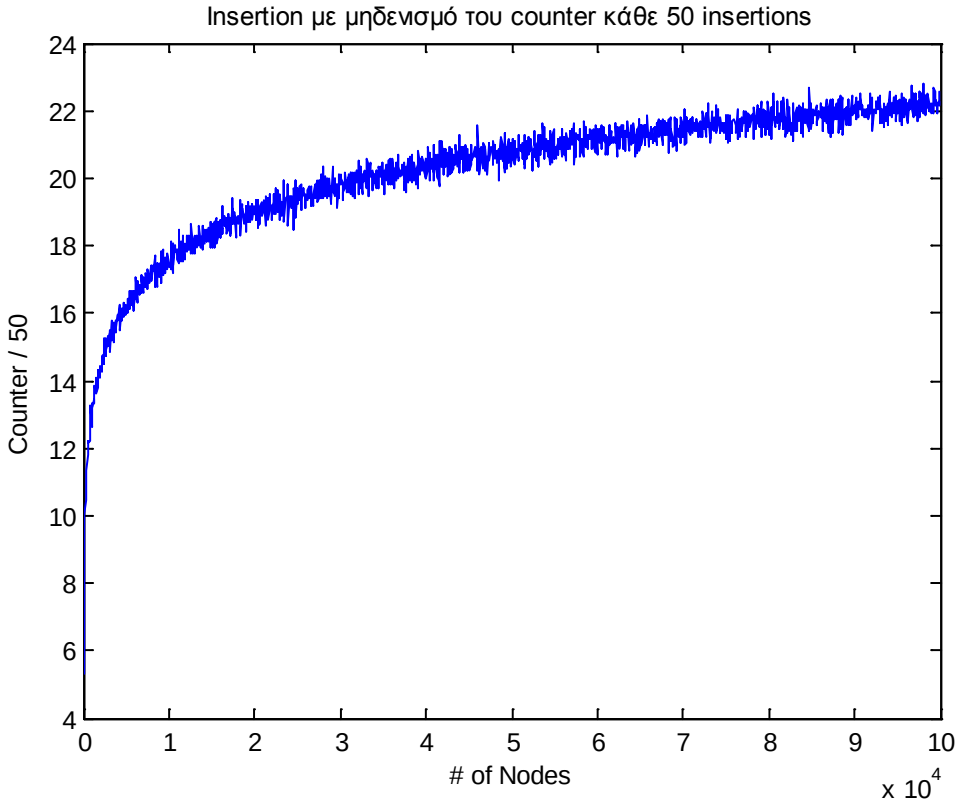


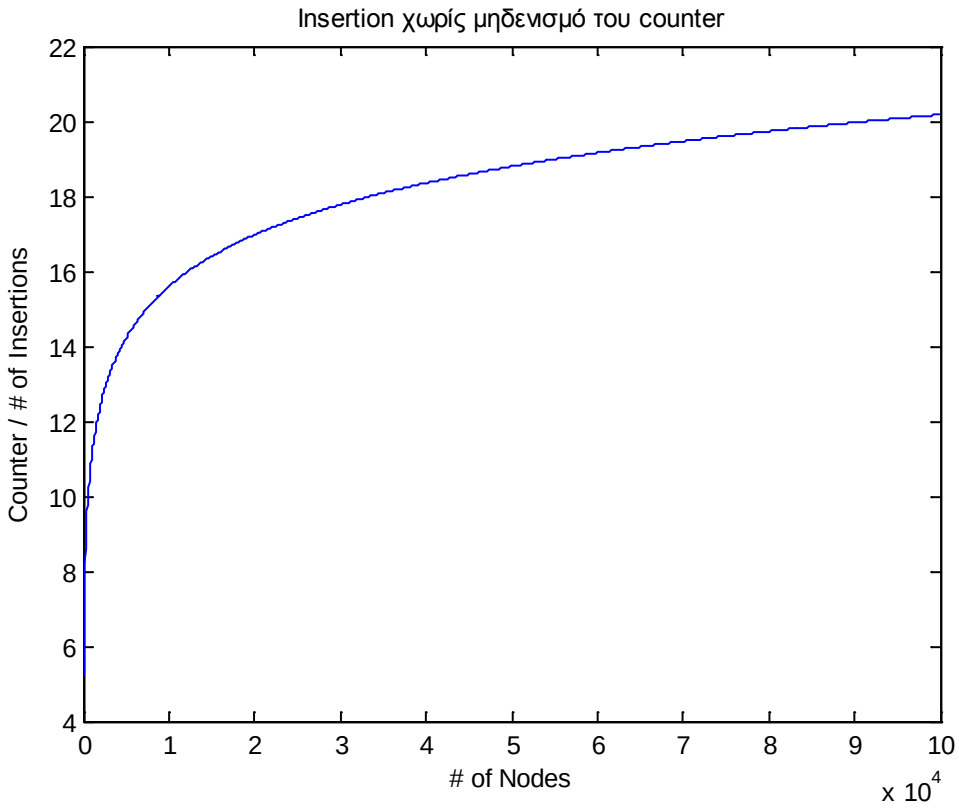

Insertion χωρίς μηδενισμό του counter
Counter / # of Insertions
# of Nodes
x 10^4
22
20
18
16
14
12
10
8
6
4
0
1
2
3
4
5
6
7
8
9
10


Deletion με μηδενισμό του counter κάθε 50 deletions
Counter / 50
# of Nodes
x 10^4
24
22
20
18
16
14
12
10
8
6
4
0
1
2
3
4
5
6
7
8
9
10



3.2.4 No Extra Space Model




Insertion χωρίς μηδενισμό του counter
Counter / # of Insertions
# of Nodes
x 10^4
22
20
18
16
14
12
10
8
6
4
0
1
2
3
4
5
6
7
8
9
10

Κεφάλαιο 4

Κώδικας

Σε αυτό το κεφάλαιο παραθέτουμε τον κώδικα της υλοποίησης των αλγορίθμων που παρουσιάσαμε στο δεύτερο κεφάλαιο. Η υλοποίηση έγινε σε γλώσσα προγραμματισμού Java. Το πρώτο μέρος αποτελεί την υλοποίηση των randomized binary search trees. Στο δεύτερο μέρος έχουμε την υλοποίηση των υπόλοιπων μοντέλων, η οποία είναι ενιαία για όλα, εκτός από την υλοποίηση των μεθόδων newroot και LchildWins.

4.1 Randomized Binary Search Tree

Κώδικας 1 Ορισμός κόμβου του RBST

```
//Αρχικά ορίζουμε την class Node, δηλαδή τον κόμβο του RBST
public class Node {

   private int key;
   private Node left,right,parent;                    //3 δείκτες προς τους γειτονικούς κόμβους

   public Node(){}                                    //απλός constructor

   public Node(int k, Node r, Node l,Node p){      //constructor όπου καθορίζονται τα περιεχόμενα

      key=k;                                                                    //του κάθε κόμβου
      right=r;
      left=l;
      parent=p;
   }
   //Μέθοδοι που καθορίζουν κι επιστρέφουν τα στοιχεία του κόμβου και μια που μετράει τους
κόμβους
   public int getKey(){
      return key;
   }
   public void setKey(int k){
      key=k;
   }
   public Node getParent(){
      return parent;
```

```
    }
    public void setParent(Node v){
        parent=v;
    }
    public Node getRight(){
        return right;
    }
    public void setRight(Node v){
        right=v;
    }
    public Node getLeft(){
        return left;
    }
    public void setLeft(Node v){
        left=v;
    }
    public int count(Node v) {
            Node r= getRight();
            Node l= getLeft();
            int c;
            if(v==null){c=0;}
            else{
            c = 1;
            if ( r!=null ) c += r.count(r);          //δεξιά subtrees
            if ( l!=null ) c += left.count(l);       //αριστερά subtrees
            }
            return c;
     }
}
```

Κώδικας 2 Ορισμός Binary Tree

```
//Η class BinaryTree ορίζει ένα απλό διασυνδεδεμένο δένδρο
public class BinaryTree {

    private Node Root;

    public BinaryTree(){       //Απλός constructor που καθορίζει ένα άδειο δένδρο
        Root=null;
    }
    public Node getRoot(){
        return Root;
    }
    public void setRoot(Node v){
        Root=v;
    }
   public boolean isRoot(Node v){
            return(v==Root);
    }
    public boolean isLeft(Node v){
            if(v==Root){
                    return false;
            }
```

```
        return(v==v.getParent().getLeft());
    }
    public boolean isRight(Node v){
        if(v==Root){
            return false;
        }
        return(v==v.getParent().getRight());
    }
}
```

Κώδικας 3 Randomized Binary Search Trees

```
//Η βασική μας class, η TotalRandomBST η οποία καθορίζει ένα τυχαίο δυαδικό δένδρο αναζήτησης
//και σε αυτήν υλοποιούμε τις πράξεις ένθεσης και διαγραφής στοιχείων στο δένδρο
public class TotalRandomBST extends BinaryTree {

    Node S,G;

    private static Random ran;
    public static int counter1=0;       //Μετρητές για τους κόμβους που συναντάμε σε μια ένθεση
    public static int counter2=0;               //και σε μια απόσβεση

    TotalRandomBST(){                           //Constructor
        super();
    }

            /* Ενθέτει ένα νέο στοιχείο, με κλειδί x στο RBST με ρίζα τον κόμβο T */
public Node insert(int x, Node T){

        if(T==null){                                    //Αν το δένδρο είναι άδειο
            return insert_at_root(x,T);                 //εισάγει το στοιχείο στη ρίζα
        }

      int n;
      n=T.count(T);

      ran=new Random();                 //Επιστρέφει έναν τυχαίο αριθμό
      int r=ran.nextInt(n)+1;           //βασισμένο στο μέγεθος του δένδρου

      if(r==n){
          counter1++;
          return insert_at_root(x,T);           //Αν r=n, εισάγει το στοιχείο στη ρίζα
          }
      Node N;

      if(x<T.getKey()){
            counter1++;
            if(T.getLeft()==null){                  //Όταν ο T δεν έχει αριστερό παιδί
                N=new Node(x,null,null,null);       //εισάγουμε το κλειδί x, δημιουργώντας
                T.setLeft(N);               //ένα νέο κόμβο ως αριστερό παιδί του T
                N.setParent(T);
            }
            else{
```

```
                T.setLeft(insert(x,T.getLeft()));//Αναδρομικά εισάγουμε το x αριστερά του T
            }
        }
        else{
            counter1++;
                if(T.getRight()==null){          //Όταν ο T δεν έχει δεξί παιδί
                N=new Node(x,null,null,null);    //εισάγουμε το κλειδί x, δημιουργώντας
                T.setRight(N);                   //ένα νέο κόμβο ως δεξί παιδί του T
                N.setParent(T);
            }
            else{
                T.setRight(insert(x,T.getRight()));  //Αναδρομικά εισάγουμε το x στα δεξιά του T
            }
        }
        return T;
    }
```

/* Ενθέτει ένα νέο στοιχείο, με κλειδί x, στη ρίζα του δένδρου, αντικαθιστώντας την παλιά T*/

```
public Node insert_at_root(int x, Node T){

        if(T==null){                             //Όταν το δένδρο είναι άδειο, δημιουργούμε
                T=new Node(x,null,null,null);            //ένα νέο κόμβο, ως ρίζα του δένδρου
                setRoot(T);
                T.setKey(x);
                }
        else{
                split(x,T);                      //Καλούμε την split, που δημιουργεί 2 υποδένδρα
                Node p=T.getParent();
                Node temp=T;
                Node big=temp.getRight();
                Node small=temp.getLeft();

                T=new Node();
                T.setKey(x);
                T.setRight(G);
                T.setLeft(S);

                if(p==null){                 //Αν ο νέος κόμβος δεν έχει πατέρα, τον θέτουμε ως ρίζα
                        setRoot(T);
                }
                else{                        //Αλλιώς τον συνδέουμε με τον T
                        T.setParent(p);
                }

                temp.setParent(T);           //Θέτουμε ως πατέρα του προηγούμενου κόμβου, το νέο

                if(big!=null){
                        if(big.getKey()>T.getKey())
                        {
                                big.setParent(T);
```

```
                }
            }
            if(small!=null){
                if(small.getKey()<T.getKey())
                {
                    small.setParent(T);
                }
            }
    }
    return T;
}
```

/* Το δένδρο με ρίζα T διασπάται σε 2 υποδένδρα S(T<) και G(T>), που περιέχουν τα keys που είναι μικρότερα από το x και μεγαλύτερα από το x, αντίστοιχα*/

```
public void split(int x, Node T){

    Node Temp;

    if(T==null){                    //Αν ο κόμβος είναι άδειος, τότε και τα S και G είναι άδεια
        S=G=null;
        return;
    }

    if(x<T.getKey()){
        G=T;
        Temp=G;
        split(x,T.getLeft());    //Αναδρομικά διασπάμε το αριστερό υποδένδρο
        Temp.setLeft(G);
        G=Temp;
    }

    else{
        S=T;
        Temp=S;
        split(x,T.getRight());  //Αναδρομικά διασπάμε το δεξί υποδένδρο
        Temp.setRight(S);
        S=Temp;
    }
    return;
}
```

/* Η delete διαγράφει από το δένδρο με ρίζα T τον κόμβο με κλειδί x, αν υπάρχει */

```
public Node delete(int x, Node T){

        Node Aux;
        if(T==null){                            //Αν το δένδρο είναι άδειο, επιστρέφει null
            return null;
```

```
        }

        if(x<T.getKey()){
                counter2++;
                T.setLeft(delete(x,T.getLeft()));//Αναδρομικά αναζητούμε το x στα αριστερά του T
        }

        else if(x>T.getKey()){
                counter2++;
                T.setRight(delete(x,T.getRight())); //Αναδρομικά αναζητούμε το x στα δεξιά του T
        }

        else{
                counter2++;
                Node temp=T.getParent();
                Aux=join(T.getLeft(),T.getRight());                    //Καλούμε τη join
                T=Aux;
                if(temp==null){
                        setRoot(Aux);
                }
                if(Aux!=null){
                Aux.setParent(temp);
                }
        }
        return T;
}
```

/* Η join συνενώνει τα αριστερά και δεξιά υποδένδρα του κόμβου που διαγράφεται */

```
public Node join(Node L, Node R){

        int m,n, r, total;

        if(L==null){                    //Στο m αποθηκεύουμε το μέγεθος του αριστερού υποδένδρου
                m=0;
        }
        else{
                m=L.count(L);
        }

        if(R==null){                            //Στο n αποθηκεύουμε το μέγεθος του δεξιού υποδένδρου
                n=0;
        }
        else{
                n=R.count(R);
        }

        total=m+n;
        if(total==0){
                return null;
        }
```

```
            ran=new Random();
            r=ran.nextInt(total);  //Επιστρέφει έναν τυχαίο αριθμό βασισμένο στα μεγέθη των
υποδένδρων
                        if(r<m){
                                    Node Rt=L.getRight();
                                    L.setRight(join(Rt,R));
                                    return L;
                        }

                        else{
                                    Node Lt=R.getLeft();
                                    R.setLeft(join(L,Lt));
                                    return R;
                        }
            }
}
```

4.2 Random Root Model-Priority-No extra Space

Κώδικας 4 Ορισμός κόμβου

```
//Αρχικά ορίζουμε την class Node, δηλαδή τον κόμβο των διαφόρων μοντέλων τυχαίων δένδρων
public class Node{
            private int key;
            private double priority;                    //Προστίθεται και μια προτεραιότητα
            private Node left, right, parent;
            public int size=0;

            public Node(){}

            public Node(int k, double pr, Node r, Node l, Node p){
                        key=k;
                        priority=pr;
                        right=r;
                        left=l;
                        parent=p;
            }
   public int getKey(){
      return key;
   }
   public void setKey(int k){
      key=k;
   }
   public double getPriority(){
      return priority;
   }
   public void setPriority(double pr){
      priority=pr;
   }
```

```
    public Node getParent(){
        return parent;
    }
    public void setParent(Node v){
        parent=v;
    }
    public Node getRight(){
        return right;
    }
    public void setRight(Node v){
        right=v;
    }
    public Node getLeft(){
        return left;
    }
    public void setLeft(Node v){
        left=v;
    }

    public int getSize(){                                   //Ορίζουμε το μέγεθος ενός κόμβου
            Node r= getRight();
                    Node l= getLeft();
                    int c;
                    if((getRight()==null)&&(getLeft()==null)&&(getKey()==0)){c=0;}
                    else{
                    c = 1;
                    if ( r!=null ) c += r.getSize();            // δεξιά subtrees
                    if ( l!=null ) c += l.getSize();          //αριστερά subtrees
                    }
                    return c;
            }
}
```

Η class BinaryTree είναι ίδια με την BinaryTree του RBST

Κώδικας 5 Ορισμός Item

```
public class Item{

            private int key;
            private double priority;

            public Item(int k, double pr){
                    key=k;
                    priority=pr;
            }
            public int getKey(){
                    return key;
            }
            public void setKey(int k){
                    key=k;
            }
```

```
	public double getPriority(){
		return priority;
	}
	public void setPriority(double pr){
		priority=pr;
	}
}
```

Κώδικας 6 Ισοδύναμος και για τα 3 μοντέλα

```
//Η class Total καθορίζει τα 3 μοντέλα που περιγράψαμε, το Random Root, το μοντέλο με τα
//priorities και το μοντέλο χωρίς επιπλέον αποθηκευτικό χώρο
public class Total extends BinaryTree{

	public static int counter1=0;			//Μετρητές
	public static int counter2=0;

	public Total(){
		super();
		}
			/* Ενθέτει ένα νέο Item x στο δένδρο με ρίζα T */
	public void GInsert(Item x, Node T){

		Node Tnew;
		if(T==null){	//Αν το δένδρο είναι άδειο, δημιουργούμε έναν κόμβο, και τον
	θέτουμε
			T=new Node(0,0,null,null,null);		//ως ρίζα
			setRoot(T);
			T.setKey(x.getKey());
			T.setPriority(x.getPriority());
			}
		else{
		 if(newRoot(x,T)>0){//Αν το αποτέλεσμα της κλήσης της newroot είναι θετικό
			counter1++;	//εισάγουμε το στοιχείο στη ρίζα του δένδρου
			RootInsert(x,T);
			}
		    else{
			if(x.getKey()<T.getKey()){	//Αλλιώς το εισάγουμε σε κάποιο
								//από τα 2 υποδένδρα
			counter1++;
				if(T.getLeft()==null){
					Tnew=new Node(0,0,null,null,null);
					Tnew.setKey(x.getKey());
					Tnew.setPriority(x.getPriority());
					T.setLeft(Tnew);
					Tnew.setParent(T);
					}

				else{
					GInsert(x,T.getLeft());
					}
			}

			else{
```

```
                                counter1++;
                                if(T.getRight()==null){
                                        Tnew=new Node(0,0,null,null,null);
                                        Tnew.setKey(x.getKey());
                                        Tnew.setPriority(x.getPriority());
                                        T.setRight(Tnew);
                                        Tnew.setParent(T);
                                                }

                                else{
                                        GInsert(x,T.getRight());
                                        }
                                 }
                        }
                }
}

        /* Ενθέτει το Item x στο δένδρο/υποδένδρο αντικαθιστώντας τη ρίζα του*/
public void RootInsert(Item x, Node T){
                Node Told;
                if(T==null){                                    //Αν το δένδρο είναι άδειο
                        T=new Node(0,0,null,null,null);
                        setRoot(T);
                        T.setKey(x.getKey());
                        T.setPriority(x.getPriority());
                        }

                else{
                        if(x.getKey()<T.getKey()){
                                if(T.getLeft()==null){
                                        Told=T;
                                        T=new Node(0,0,null,null,null);
                                        T.setKey(x.getKey());
                                        T.setPriority(x.getPriority());
                                        Told.setLeft(T);
                                        T.setParent(Told);

                                        if(isLeft(T)){
                                        RotateRight(T,T.getParent());   //Δεξιά   περιστροφή
                                        return;
                                        }
                                        else{
                                        RotateLeft(T,T.getParent());   //Αριστερή περιστροφή
                                        return;
                                        }
                                }
                                else{
                                        counter1++;
                                        RootInsert(x,T.getLeft());
                                        T=T.getLeft();
                                        RotateRight(T,T.getParent());
                                        return;
```

```
                        }
                    }

                else{
                    if(T.getRight()==null){
                        Told=T;
                        T=new Node(0,0,null,null,null);
                        T.setKey(x.getKey());
                        T.setPriority(x.getPriority());
                        Told.setRight(T);
                        T.setParent(Told);

                        if(isLeft(T)){
                            RotateRight(T,T.getParent());
                            return;
                            }
                        else{
                            RotateLeft(T,T.getParent());
                            return;
                            }
                        }
                    else{
                        counter1++;
                        RootInsert(x,T.getRight());
                        T=T.getRight();

                        RotateLeft(T,T.getParent());
                        return;
                        }
                    }
                }
}
```

/* Διαγράφει ένα στοιχείο βάσει του κλειδιού του, a, από το δένδρο με ρίζα T */

```
public void GDelete(int a, Node T){

    if(T==null){
        return;
        }
    else{
        Node delNode=findNode(a,T);          //Αναζήτηση του κόμβου
            if(delNode==null){
            return;
            }

        if(a<T.getKey()){                    //Αναδρομικά στα αριστερά
            counter2++;
            GDelete(a,T.getLeft());
            }

        else if(a>T.getKey()){               //Αναδρομικά στα δεξιά
            counter2++;
```

```
                    GDelete(a,T.getRight());
                    }

                else{                                           //Διαγραφή της ρίζας
                    counter2++;
                    RootDelete(T);
                    }
                }
}

                        /* Διαγραφή της ρίζας του δένδρου, με ρίζα T */
public void RootDelete(Node T){

        if((T.getLeft()==null)&&(T.getRight()==null)){

                if(isRoot(T)){
                    setRoot(null);
                    }
                else{
                    if(isLeft(T)){
                            T.getParent().setLeft(null);
                            }
                    else{
                            T.getParent().setRight(null);
                            }
                }
                T.setParent(null);
                T=null;
        }

        else{
                if((T.getLeft()!=null)&&(T.getRight()!=null)){
                    if(LchildWins(T)==1){
                            RotateRight(T.getLeft(),T);     //Αναδρομικά το
                                                            περιστρέφουμε σε //θέση-
                                                            φύλλο
                            RootDelete(T);                  //και το διαγράφουμε
                            return;
                            }
                    else{
                            RotateLeft(T.getRight(),T);
                            RootDelete(T);
                            return;
                            }
                    }
                else if((T.getLeft()!=null)&&(T.getRight()==null)){
                            RotateRight(T.getLeft(),T);
                            RootDelete(T);
                            return;
                            }
                else if((T.getLeft()==null)&&(T.getRight()!=null)){
                            RotateLeft(T.getRight(),T);
```

```
                                        RootDelete(T);
                                        return;
                                        }
                                }
                        }
}
```

NewRoot:

για το μοντέλο RandomRoot:

```
public int newRoot(Item x, Node T){

                Random rans=new Random();
                int r=rans.nextInt(T.getSize())+1;   //Επιλέγεται ένας τυχαίος αριθμός, βασισμένος
                                                     //στο μέγεθος του δένδρου
                if(r==T.getSize()){
                        return 1;
                }
                else{
                        return 0;
                }
        }
```

για το μοντέλο Priority:

```
public int newRoot(Item x, Node T){
        if(x.getPriority()>T.getPriority()){        //Συγκρίνει τα priorities
                return 1;
                }
        else{
                return 0;
        }
}
```

για το μοντέλο χωρίς Extra Space:

```
public int newRoot(Item x, Node T){
        Random ran=new Random();
        double p=ran.nextDouble();                  //Επιλέγεται ένας τυχαίος αριθμός
                                                    //από μια κατανομή
        if(preOrder(T,p)==0){                       //Αν είναι μικρότερος από όλους τους
                return 0;                           //τυχαίους αριθμούς που έχουν δοθεί
        }                                           //στα υπόλοιπα στοιχεία, τότε return 0
        else{
                return 1;                           //Αλλιώς return 1
        }
}
public int preOrder(Node t,double p){               //Η pre-order διατρέχει αναδρομικά
        Random ran=new Random();                    // όλα τα στοιχεία του δένδρου και συγκρίνει
        double p2=ran.nextDouble();                 //τους τυχαίους αριθμούς που έχουν
                if(p2>p){
```

```
                return 0;
        }
        else{
                if(t.getLeft()!=null){
                        return(preOrder(t.getLeft(),p));
                }
                if(t.getRight()!=null){
                        return(preOrder(t.getRight(),p));
                }
                else{
                return 1;
                }
                }
    }
```

LchildWins

για το μοντέλο RandomRoot:

```
public int LchildWins(Node T){
    if(T.getLeft().getSize()>T.getRight().getSize()){       //Συγκρίνει τα μεγέθη των υποδένδρων
        return 1;
    }
    else{
        return 0;
    }
}
```

για το μοντέλο Priority:

```
public int LchildWins(Node T){
    if(T.getLeft().getPriority()>T.getRight().getPriority()){       //Συγκρίνει τα priorities
        return 1;
    }
    else{
        return 0;
    }
}
```

Βιβλιογραφία

[1] Martinez, C., Roura, S.: Randomized Binary Search Trees. J. ACM 45(2), 288-323 (1998).

[2] Seidel, R.: Maintaining Ideally Distributed Random Search Trees without Extra Space. Festschrift Mehlhorn, LNCS 5760, pp. 134-142, 2009.

[3] Π.Δ. Μποζάνης, Δομές Δεδομένων: Ταξινόμηση και αναζήτηση με Java, Θεσσαλονίκη, Εκδόσεις Τζιόλα, 2003.

[4] Roura, S., Martinez, C. 1996. Randomization of search trees by subtree size. In Proceedings of the 4th European Symposium on Algorithms (ESA), J. Dıaz and M. Serna, eds. Lecture Notes in Computer Science, vol. 1136. Springer-Verlag, New York, pp. 91–106.

[5] Seidel, R., Aragon, C.R.: Randomized Search Trees. Algorithmica 16(4/5), 464-497 (1996).

[6] Mehlhorn, K.: Data structures and Algorithms 3. Springer, Heidelberg (1984).

[7] Pugh, W. 1990. Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33, 6 (June), 668-676.

[8] Nievergelt, J., Reingold, E. 1973. Binary search trees of bounded balance. Siam J. Comput. 2, 1, 33-43.

[9] Gonnet, G., Olivie, H., and Wood, D. 1983. Height-ratio-balanced trees. Comput. J. 26, 2 106-108.