Πανεπιστήμιο Θεσσαλίας
Πολυτεχνική Σχολή
Τμήμα Μηχανικών Η/Υ, Τηλεπικοινωνιών & Δικτύων

Βαθμο-Εξαρτώμενες Ουρές Προτεραιοτήτων

Διπλωματική Εργασία

Καραμπέρης Στυλιανός
Επιβλέπων Καθηγητής: Παναγιώτης Μποζάνης

Περιεχόμενα

ΚΕΦΑΛΑΙΟ 1 - ΕΙΣΑΓΩΓΗ – ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ

1.1 Αντικείμενο Δομών Δεδομένων 4
1.1.1 Βασικές Λειτουργίες (πράξεις) επί των Δομών Δεδομένων 4
1.1.2 Στατικά Δέντρα 5
1.1.3 Δυναμικά Δέντρα 5
1.1.4 Εισαγωγή στο πρόβλημα Λεξικού 6
1.1.5 Περιγραφή αλγορίθμου 7
1.2 Ανάλυση Αλγορίθμων 7
1.2.1 Μοντέλα υπολογισμού – Μοντέλο Μηχανής Τυχαίας Προσπελάσεως 7
1.2.2 Ανάλυση Χειρότερης Περίπτωσης 8
1.2.3 Ανάλυση Μέσης Περίπτωσης 8
1.2.4 Ανάλυση Επιμερισμένης ή Κατανεμημένης Περίπτωσης 8
1.2.4.1 Μέθοδος Λογαριασμού Τραπεζίτη ή Λογιστή 9
1.2.4.2 Μέθοδος Δυναμικού 10

ΚΕΦΑΛΑΙΟ 2 - ΔΥΑΔΙΚΑ ΔΕΝΤΡΑ ΑΝΑΖΗΤΗΣΕΩΣ

2.1 Αζύγιστα Δυαδικά Δέντρα Αναζητήσεως 11
2.1.1 Περιγραφή Πράξεων 12
2.1.2 Ανάλυση Πολυπλοκότητας 18

ΚΕΦΑΛΑΙΟ 3 - ΤΥΧΑΙΕΣ ΔΟΜΕΣ ΛΕΞΙΚΟΥ

3.1 Treap 20

3.1.1 Περιγραφή των Βασικών Πράξεων 20

3.1.2 Ανάλυση Πολυπλοκότητας 24

ΚΕΦΑΛΑΙΟ 4 - ΒΑΘΜΟ-ΕΞΑΡΤΩΜΕΝΕΣ ΟΥΡΕΣ ΠΡΟΤΕΡΑΙΟΤΗΤΩΝ 26

4.1 Τυχαιοποιημένη προσέγγιση 26

4.1.1 Περιγραφή των Βασικών Πράξεων 26

4.1.2 Ανάλυση Πολυπλοκότητας 30

4.1.3 Πειραματική Αξιολόγηση 30

4.1.3.1 Υπολογισμός Κόστους Στοιχειωδών Πράξεων 30

4.1.3.2 Μετρήσεις Πραγματικού Χρόνου 34

4.2 Αναπόσβεστη Προσέγγιση 35

4.2.1 Περιγραφή των Βασικών Πράξεων 35

4.2.2 Ανάλυση Πολυπλοκότητας 40

4.2.3 Πειραματική Αξιολόγηση 40

4.2.3.1 Υπολογισμός Κόστους Στοιχειωδών Πράξεων 40

4.2.3.2 Μετρήσεις Πραγματικού Χρόνου 44

ΠΑΡΑΡΤΗΜΑ Α 45

ΠΑΡΑΡΤΗΜΑ Β 53

ΠΕΡΙΛΗΨΗ

Αντικείμενο της παρούσης διπλωματικής εργασίας είναι η υλοποίηση και αξιολόγηση δύο νέων δομών που σχετίζονται με βαθμο-εξαρτώμενες ουρές προτεραιοτήτων, μια δομή η οποία γνωρίζει πάντα το ελάχιστο στοιχείο που περιέχει για το οποίο η εισαγωγή και διαγραφή κοστίζει $O(\log n/r)$, n ο αριθμός των στοιχείων στην δομή και r ο βαθμός του στοιχείου που εισάγεται ή διαγράφεται ($r = 1$ για το μικρότερο στοιχείο, $r = n$ για το μέγιστο). Τέλος αξιολογούμε δυο προσεγγίσεις την «τυχαιοποιημένη» και την «αναπόσβεστη».

ΚΕΦΑΛΑΙΟ 1 - ΕΙΣΑΓΩΓΗ – ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ

1.1 Αντικείμενο Δομών Δεδομένων

Με τον όρο **«αλγόριθμος»** χαρακτηρίζεται κάθε καλώς ορισμένη υπολογιστική διαδικασία, η οποία καλείται να επιλύσει ένα συγκεκριμένο πρόβλημα, εντός πεπερασμένου χρόνου. Τυπικότερα, αποτελεί μια ακολουθία υπολογιστικών βημάτων, η οποία απεικονίζει την **είσοδο (input)** του προβλήματος –ήτοι, τα δεδομένα του- στην **έξοδο (output)** –δηλαδή, στην λύση του προβλήματος. Κάθε είσοδος που ικανοποιεί τις προδιαγραφές του προβλήματος καλείται **νόμιμη (legal)** και λέμε ότι ορίζει ένα συγκεκριμένο **στιγμιότυπο (instance)** του προβλήματος. Ένας αλγόριθμος επιλύει ένα πρόβλημα, όταν, για κάθε στιγμιότυπο (instance) του εν λόγω προβλήματος, τερματίζει μετά από πεπερασμένο χρόνο, παράγοντας σωστή έξοδο.

Από την άλλη, αντικείμενο των «**Δομών Δεδομένων**» είναι η αναπαράσταση και η διαχείριση συνόλων αντικειμένων, τα οποία επιδέχονται πράξεις εξαγωγής πληροφορίας ή αλλαγής της συνθέσεώς τους. Αυστηρότερα, μπορεί να ορίσει κανείς πως ασχολούνται με την επιστάμενη μελέτη των υλοποιήσεων των συχνότερα εμφανιζόμενων **Αφηρημένων Τύπων Δεδομένων (ΑΤΔ) (Abstract Data Types – ADT)**. Ως αφηρημένος τύπος δεδομένων ορίζεται ένα σύνολο, με μια συλλογή πράξεων επί των στοιχείων του συνόλου.

1.1.1 Βασικές Λειτουργίες (πράξεις) επί των Δομών Δεδομένων

Οι βασικές λειτουργίες (πράξεις) επί των Δομών Δεδομένων είναι οι ακόλουθες:

- ***Προσπέλαση (access)***, πρόσβαση σε ένα κόμβο με σκοπό να εξεταστεί ή να τροποποιηθεί το περιεχόμενο του.
- ***Εισαγωγή (insertion)***, προσθήκη νέων κόμβων σε μια υπάρχουσα δομή.
- ***Διαγραφή (deletion)***, αφαίρεση ενός κόμβου από μια δομή.
- ***Αναζήτηση (searching)***, προσπέλαση των κόμβων μιας δομής, προκειμένου να εντοπιστούν ένας ή περισσότεροι που έχουν μια δεδομένη ιδιότητα.

- ***Ταξινόμηση (sorting)***, όπου οι κόμβοι μιας δομής διατάσσονται κατά αύξουσα ή φθίνουσα σειρά.
- ***Αντιγραφή (copying)***, κατά την οποία όλοι οι κόμβοι ή μερικοί από κόμβους μιας δομής αντιγράφονται σε μια άλλη.
- ***Συγχώνευση (merging)***, δύο ή περισσότερες δομές συνενώνονται σε μια ενιαία δομή.
- ***Διαχωρισμός (separation)***, μια ενιαία δομή διασπάται σε δυο ή περισσότερες δομές.

1.1.2 Στατικά Δέντρα

Πρόκειται για τα γνωστά από τη Θεωρία Γραφημάτων, δέντρα με ρίζα. Από προγραμματιστικής απόψεως χαρακτηρίζονται από τις ακόλουθες πράξεις (ΑΤΔ):

- ***Element(u)*** επιστρέφει το στοιχείο που είναι αποθηκευμένο στον κόμβο u
- ***Father(u)*** επιστρέφει δείκτη προς τον πατέρα του u
- ***Children(u)*** επιστρέφει όλους τους δείκτες προς τα παιδιά του u

Παρατηρήστε πως οι παραπάνω πράξεις, ναι μεν επιτρέπουν την επισκόπηση του δέντρου, αλλά απαγορεύουν οποιαδήποτε τροποποίηση του. Μερική ευελιξία προς αυτήν την κατεύθυνση, παρέχουν οι ακόλουθες πράξεις:

- ***setElement(u,e)*** θέτει το e ως στοιχείο του u
- ***setSon(u,p,i)*** θέτει τον κόμβο p ως τον i-στο γιο του u
- ***setFather(u,p)*** θέτει τον κόμβο p ως πατέρα του u

οι οποίες αλλάζουν τη μορφή του εκάστοτε δέντρου, χωρίς να αφαιρούν ή να προσθέτουν κόμβους.

1.1.3 Δυναμικά Δέντρα

Τα **δυναμικά δέντρα** (dynamic trees) προκύπτουν με την προσθήκη πράξεων που ενθέτουν και αποσβένουν κόμβους στο εκάστοτε δέντρο που υφίσταται την αλλαγή. Ο αντίστοιχος ΑΤΔ για τα δυναμικά δέντρα έχει ως εξής:

- ***addLeaf(u,kindofson)*** Προσθέτει ένα νέο κόμβο ως kindofson (δεξί ή αριστερό)

παιδί του u, εφ' όσον δεν διαθέτει τέτοιο

- ***deleteNode(u)*** Αφαιρεί τον κόμβο u, εφ' όσον έχει το πολύ έναν μη κενό γιο

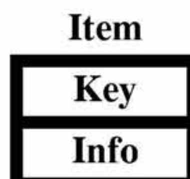
1.1.4 Εισαγωγή στο πρόβλημα Λεξικού

Έστω ένα σύνολο $S = \{(x, y) | x \in U, y\}$, όπου U το σύνολο σύμπαν, δηλαδή ένα ολικώς διατεταγμένο σύνολο αντικειμένων –«κλειδιών», και y η συσχετιζόμενη με το x πληροφορία. Παραδείγματος χάριν, στις Δ.Ο.Υ., κάθε φορολογούμενος, είτε φυσικό είτε νομικό πρόσωπο, χαρακτηρίζεται από ένα αριθμό μητρώου, (Α.Φ.Μ.), έναν εξαψήφιο ακέραιο, ο οποίος προσδιορίζει την ταυτότητα του. Οπότε πολύ σχηματικά, για κάθε φορολογούμενο ορίζεται η δυάδα (Α.Φ.Μ., φάκελος φορολογούμενου). Συνήθως το ζεύγος (x,y) ονομάζεται **στοιχείο (item)**.

Μια δομή επί ενός συνόλου διατεταγμένων δυάδων, καλείται δομή λεξικού (dictionary data structure), όταν ικανοποιεί τον ακόλουθο ΑΤΔ:

- ***insertItem(key, info)*** Ένθεση ενός στοιχείου ου φέρει πληροφορία info και χαρακτηρίζεται με ένα κλειδί key
- ***deleteItem(key)*** Απόσβεση, αν υπάρχει, του στοιχείου με κλειδί key
- ***findInfo(key)*** Εύρεση, αν υπάρχει, του στοιχείου με κλειδί key και επιστροφή της πληροφορίας του

Από απόψεως υλοποιήσεως, χρειαζόμαστε ένα νέο αντικείμενο Item, το οποίο θα φέρει δύο πεδία: ένα Object key για το κλειδί και ένα Object info για την συσχετιζόμενη με το key πληροφορία. Γραφικά έχουμε την εξής αναπαράσταση:



1.1.5 Περιγραφή αλγορίθμου

Η φυσιολογική κατάληξη κάθε αλγορίθμου είναι η τελική έκφραση του σε κάποια γλώσσα προγραμματισμού, ώστε, πλέον, να είναι δυνατή η χρήση ενός υπολογιστή για την επίλυση του αντίστοιχου προβλήματος. Συνήθως , όμως, κατά το στάδιο σχεδιασμού ενός αλγορίθμου, προτιμάται η περιγραφή του είτε σε **φυσική γλώσσα** είτε σε **ψευδογλώσσα προγραμματισμού (pseudo-programming language) ή ψευδοκώδικα (pseudocode)**. Η τελευταία αποτελείται από μια μίξη εντολών πραγματικών γλωσσών προγραμματισμού, με προτάσεις σε φυσική γλώσσα. Κατά τον τρόπο αυτό είμαστε σε θέση να εστιάσουμε την προσοχή μας στον ακριβέστερο προσδιορισμό του αλγορίθμου και στις ιδιαιτερότητες του προβλήματος.

1.2 Ανάλυση Αλγορίθμων

Ανάλυση Αλγορίθμου (algorithm analysis) αποκαλείται η εύρεση των **πόρων (resources)** που αυτός απαιτεί να τρέξει. Με άλλα λόγια, ο **χρόνος (time)** περάτωσης του και ο αναγκαίος –για τους υπολογισμούς- **χώρος (space)**, μετρούμενος σε **αποθηκευτικές θέσεις (memory locations)**. Οι δυο αυτοί δείκτες μετρήσεων της αποτελεσματικότητας του εκάστοτε αλγορίθμου συνιστούν την **πολυπλοκότητα χρόνου και χώρου** του **(time and space complexity)**.

1.2.1 Μοντέλα υπολογισμού – Μοντέλο Μηχανής Τυχαίας Προσπελάσεως

Το **Μοντέλο Μηχανής Τυχαίας Προσπελάσεως RAM (Random Access Machine Model)** αναφέρεται σε συστήματα του ενός επεξεργαστή γενικού σκοπού δίχως την δυνατότητα τελέσεως ταυτόχρονων ενεργειών (concurrent operations). Με απλά λόγια, σε ένα σύστημα που

(α) διαθέτει τους αναγκαίους καταχωρητές (registers), έναν συσσωρευτή (accumulator) και μια ακολουθία αποθηκευτικών θέσεων με διευθύνσεις 0, 1, 2, …. οι οποίες συνιστούν την κύρια μνήμη του (main memory), και

(β) είναι σε θέση να εκτελεί τις αριθμητικές πράξεις {+, -, *, /, %}, να παίρνει αποφάσεις διακλάδωσης (τύπου **if**) βάσει των τελεστών $\{=, <, >, \leq, \geq, \neq\}$ και να διαβάζει και να γράφει από και προς τις θέσεις μνήμης.

Οι **στοιχειώδεις πράξεις (primitive operations)** που αναφέρονται στο (β) πρέπει να χρεωθούν κάποιο χρόνο. Υπάρχουν δύο θεωρήσεις. Η πρώτη, η λεγόμενη και **μέτρηση μοναδιαίου κόστους (unit cost measure)**, χρεώνει κάθε πράξη σταθερό (πεπερασμένο) κόστος, ανεξαρτήτως του μήκους της δυαδικής αναπαραστάσεως των τελεστών (operands). Η δεύτερη, η αποκαλούμενη **μέτρηση λογαριθμικού κόστους (logarithmic cost measure)**, θεωρεί πως η πράξη παίρνει χρόνο ανάλογο με το μήκος της δυαδικής αναπαραστάσεως των τελεστών. Λόγου χάριν, η μετακίνηση του αριθμού n από την κύρια μνήμη προς έναν καταχωρητή χρεώνεται $\lfloor \log n \rfloor + 1$ μονάδες χρόνου. Συνήθως, χρησιμοποιείται το μοναδιαίο μέτρο, εκτός και εάν γίνεται εκτενής χρήση πράξεων επί συμβολοσειρών μπιτ.

1.2.2 Ανάλυση Χειρότερης Περίπτωσης

Με τον όρο **ανάλυση χειρότερης περίπτωσης (worst case analysis)** ονομάζουμε την μέγιστη τιμή που μπορεί να πάρει ο χρόνος τρεξίματος ή ο χώρος ενός αλγορίθμου για οποιοδήποτε είσοδο με συγκεκριμένο αριθμό n. Επομένως, η ανάλυση χειρότερης περίπτωσης βρίσκει το άνω όριο στην συμπεριφορά του αλγορίθμου, όταν του δοθεί μια νόμιμη είσοδος μεγέθους n.

1.2.3 Ανάλυση Μέσης Περίπτωσης

Σε περίπτωση που είναι γνωστή η κατανομή πιθανότητας επί του συνόλου των στιγμιότυπων του εν λόγω προβλήματος, τότε είναι δυνατή η **ανάλυση μέσης ή αναμενόμενης περίπτωσης (average/expected case analysis)**. Αυτή μας δίνει την μέση ή την αναμενόμενη συμπεριφορά του αλγορίθμου, όταν του δοθεί μια νόμιμη είσοδος με συγκεκριμένο μέγεθος n.

1.2.4 Ανάλυση Επιμερισμένης ή Κατανεμημένης Περίπτωσης

Η πολυπλοκότητα μιας δομής καθορίζει και την συμπεριφορά του αλγορίθμου που την μεταχειρίζεται. Υπάρχουν περιπτώσεις, στις οποίες μας

ενδιαφέρει ο συνολικός χρόνος *ακολουθίας* πράξεων ωα είναι φραγμένος. Αυτό συνεπάγεται μεγαλύτερη ευελιξία στο θέμα σχεδιασμού της δομής· επιτρέπεται, πλέον, ο χρόνος μιας πράξεως να μεταβάλλεται, αρκεί «ακριβές» πράξεις να ακολουθούνται από πολλές «φθηνές». Αυτός ο τρόπος αναλύσεως της επιδόσεως μιας δομής, ως μέσος όρος επιδόσεως επί ακολουθίας πράξεων, είναι γνωστός **ως ανάλυση κατανεμημένης ή επιμερισμένης περίπτωσης (amortized-case analysis)**.

Τυπικότερα, έστω $T(n)$ ο μέγιστος χρόνος εκτελέσεως μιας οποιασδήποτε ακολουθίας n πράξεων επί μιας δομής. Ως επιμερισμένος ή κατανεμημένος χρόνος για μια πράξη ορίζεται το πηλίκο $T(n)/n$. Μερικές φορές στην βιβλιογραφία χρησιμοποιείται και ο όρος **μέσος χειρότερης περίπτωσης (worst-case average)**. Αυτό σημαίνει πως, εάν η επιμερισμένη επίδοση μιας δομής είναι $f(n)$, τότε μια οποιαδήποτε ακολουθία n πράξεων κοστίζει το πολύ $nf(n)$. Στη συνέχεια, θα εξετάσουμε δύο ισοδύναμες τεχνικές επιμερισμένης αναλύσεως: την **μέθοδο λογαριασμού τραπεζίτη (banker account method)** και την **μέθοδο συναρτήσεως δυναμικού (potential function method)**.

1.2.4.1 Μέθοδος Λογαριασμού Τραπεζίτη ή Λογιστή

Κατά τη **μέθοδο λογαριασμού τραπεζίτη (banker account method)** ή την **λογιστική μέθοδο (accounting method)**, κάθε πράξη χρεώνεται ένα **κατανεμημένο** ή **επιμερισμένο κόστος (amortized cost)**, το οποίο, ενδεχομένως να είναι μικρότερο ή και μεγαλύτερο από το αντίστοιχο πραγματικό. Η επιλογή του κατανεμημένου κόστους πρέπει να γίνει κατάλληλα, ούτως ώστε: (α) να προσεγγίζεται το μέσο κόστος της πράξεως σε μια οποιαδήποτε ακολουθία πράξεων, και (β) το επιμέρους κατανεμημένο κόστος όλων των πράξεων, αθροιζόμενο, να φράσσει από πάνω το πραγματικά παρατηρούμενο χειρότερο της ακολουθίας.

Συνήθως, η διαφορά μεταξύ πραγματικού και κατανεμημένου κόστους χαρακτηρίζεται ως **πίστωση (credit)** και δηλώνει είτε *πλεόνασμα*, που κατατίθεται προς μελλοντική χρήση, κατά την εξυπηρέτηση των επόμενων πράξεων, είτε το *δάνειο*, που λαμβάνεται από τα αποθεματικά, για την κάλυψη των τρεχουσών αναγκών μιας πράξεως. Η πρακτική της πιστώσεως, με τραπεζικούς όρους, χρεώνει τις «φθηνές» πράξεις κάτι παραπάνω, ώστε να καλυφθεί το επιπλέον, από το μέσο παρατηρούμενο, κόστος των «ακριβών» πράξεων.

1.2.4.2 Μέθοδος Δυναμικού

Η μέθοδος δυναμικού (potential method) στηρίζεται στην ιδέα της απεικονίσεως της καταστάσεως μιας δομής ή ενός αλγορίθμου Α μέσω μιας συνάρτησης δυναμικού

$\mathbf{\Phi} : \mathrm{A} \rightarrow \mathbb{R}$

Αρχικά, αποδίδεται μια αρχική τιμή $\Phi(A_0)$. Μετά την i-στη πράξη o_i, πραγματικού κόστους c_i, έχουμε μετάβαση από την κατάσταση A_{i-1} στην A_i και μεταβολή του δυναμικού κατά:

$\mathbf{\Delta\Phi_i} = \mathbf{\Phi}(A_i) - \mathbf{\Phi}(A_{i-1})$

Το κατανεμημένο κόστος c_i' της o_i ορίζεται ως:

$$c_i' = c_i + \boldsymbol{\Delta\Phi_i}$$

Δηλαδή, το πραγματικό κόστος συν την μεταβολή που επήλθε στο δυναμικό εξ αιτίας της o_i.

Καθίσταται φανερό, πως κεντρικό ρόλο στην ανάλυση δυναμικού, παίζει η εκλογή της κατάλληλης συναρτήσεως δυναμικού **Φ**. Χάρις στην τελευταία, κάποιες πράξεις χρεώνονται περισσότερο (όταν $\boldsymbol{\Delta\Phi_i} > 0$) και κάποιες λιγότερο (όταν $\boldsymbol{\Delta\Phi_i} < 0$), συνολικά, όμως,επιτυγχάνεται ορθή ερμηνεία της πολυπλοκότητας της Α.

ΚΕΦΑΛΑΙΟ 2 - ΔΥΑΔΙΚΑ ΔΕΝΤΡΑ ΑΝΑΖΗΤΗΣΕΩΣ

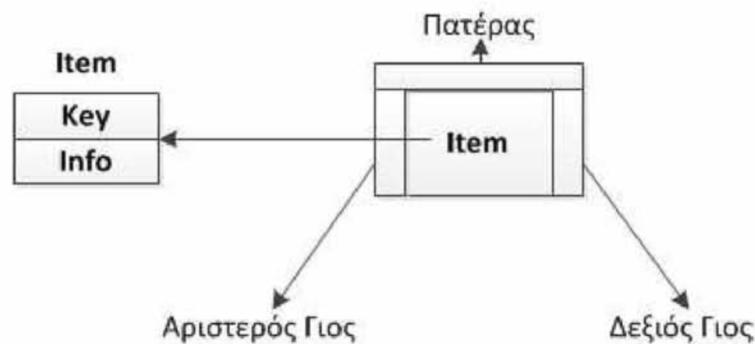
2.1 Αζύγιστα Δυαδικά Δέντρα Αναζητήσεως

Ένα **δυαδικό δέντρο αναζήτησης (binary search tree)** είναι ένα δυαδικό δέντρο κάθε κόμβος u του οποίου ικανοποιεί τα εξής:

i) Τα κλειδιά του αριστερού υποδέντρου του u είναι μικρότερα από το κλειδί του u

ii) Τα κλειδιά του δεξιού υποδέντρου του u είναι μεγαλύτερα (ή ίσα) από το κλειδί του u.

Από απόψεως υλοποιήσεως, ως element, αποθηκεύουμε πια ένα αντικείμενο, που φέρει δύο πεδία: ένα Object key, για το κλειδί και ένα Object info, για την συσχετιζόμενη πληροφορία:



Σχήμα 2.1: Γραφική παράσταση κόμβου δυαδικού δέντρου αναζητήσεως.

Έτσι ώστε, σε κάθε κόμβο u της δομής, να ισχύει η κάτωθι αμετάβλητη συνθήκη (invariant):

Το κλειδί του αριστερού γιου του u είναι μικρότερο από αυτό του u, ενώ ο δεξιός γιος του u διαθέτει μεγαλύτερο κλειδί από εκείνο του u.

Τα δέντρα που προκύπτουν από αυτή τη συνθήκη χαρακτηρίζονται και ως **κομβοπροσανατολισμένα δυαδικά δέντρα αναζητήσεως (node-oriented binary search trees).**

2.1.1 Περιγραφή Πράξεων

Βάσει της ανωτέρω θεωρήσεως, οι βασικές πράξεις, σε ψευδογλώσσα, έχουν ως εξής:

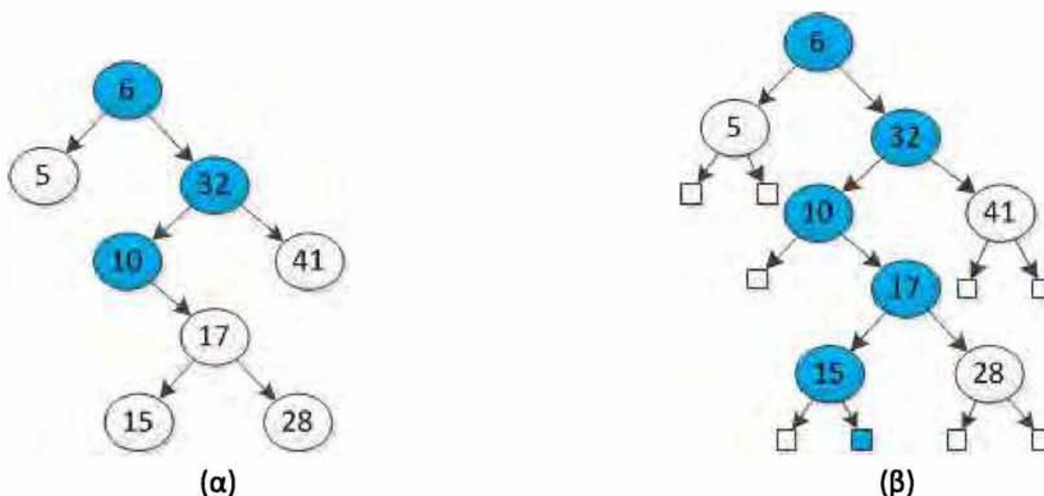
Αναζήτηση Στοιχείου. Εφαρμόζουμε τον κάτωθι αλγόριθμο:

```
Algorithm FINDNODE (BTNode u, Object key)
Input:    Ένας κόμβος δέντρου u με ένα κλειδί key
Output:   Ο κόμβος του υποδέντρου T_u όπου θα έπρεπε να υπάρχει στοιχείο με
κλειδί key
1.    If (key < key του u { //πηγαίνουμε αριστερά
2.          if (ο u δεν έχει αριστερό παιδί)
3.                return u;
4.          else
5.                return FindNode (αριστερό παιδί του u, key);
6.    }
7.    else if (key == key του u) //το βρήκαμε
8.          return u;
9.    else { //πήγαινε δεξιά
10.         if (ο u δεν έχει δεξιό παιδί)
11.               return u;
12.         else
13.               return FindNode (δεξιό παιδί του u, key);
14.   }
end of FINDNODE
```

Η κλήση του παραπάνω αλγορίθμου γίνεται από τον κόμβο της ρίζας. Εκμεταλλευόμενος της σχετικής τοποθετήσεως των στοιχείων, συγκρίνει το κλειδί του τρέχοντος κόμβου με το key. Εάν είναι ίσα, τότε ο κόμβος έχει εντοπιστεί. Διαφορετικά, κινείται είτε αριστερά (το key είναι μικρότερο) είτε δεξιά (το key είναι μεγαλύτερο). Εάν το key δεν υπάρχει, τότε η αναζήτηση θα καταλήξει σε ένα κόμβο διαθέτοντας ένα κενό (null) δείκτη, στην θέση του οποίου, θα έπρεπε να υπάρχει δείκτης προς κόμβο με το εν λόγω κλειδί.

Οπότε η αναζήτηση πληροφορίας βάσει κλειδιού key είναι άμεση:

```
Algorithm FINDINFO (Object key)
Input:    Ένα κλειδί key
Output:   Η πληροφορία που σχετίζεται με το κλειδί, εάν υπάρχει
1.   insNode = FindNode(key, ρίζα δέντρου);
2.   if (το κλειδί του insNode == key)
3.        return insNode.getElement;
4.   else
5.        return null;
end of FINDINFO
```

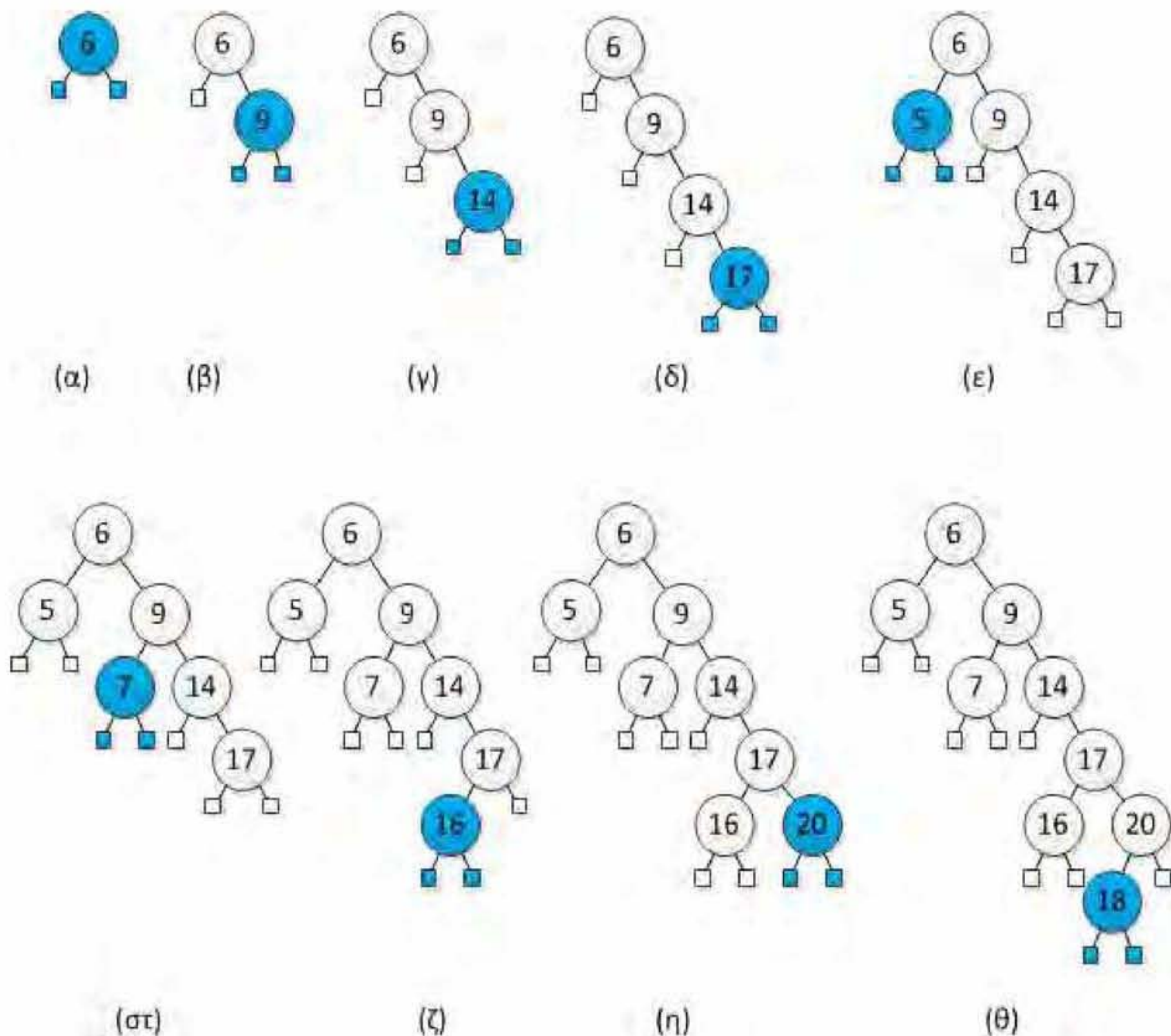


Σχήμα 2.2: Στιγμιότυπα: (α) επιτυχημένη αναζήτηση του 10, (β) αποτυχημένης αναζήτηση του 16.

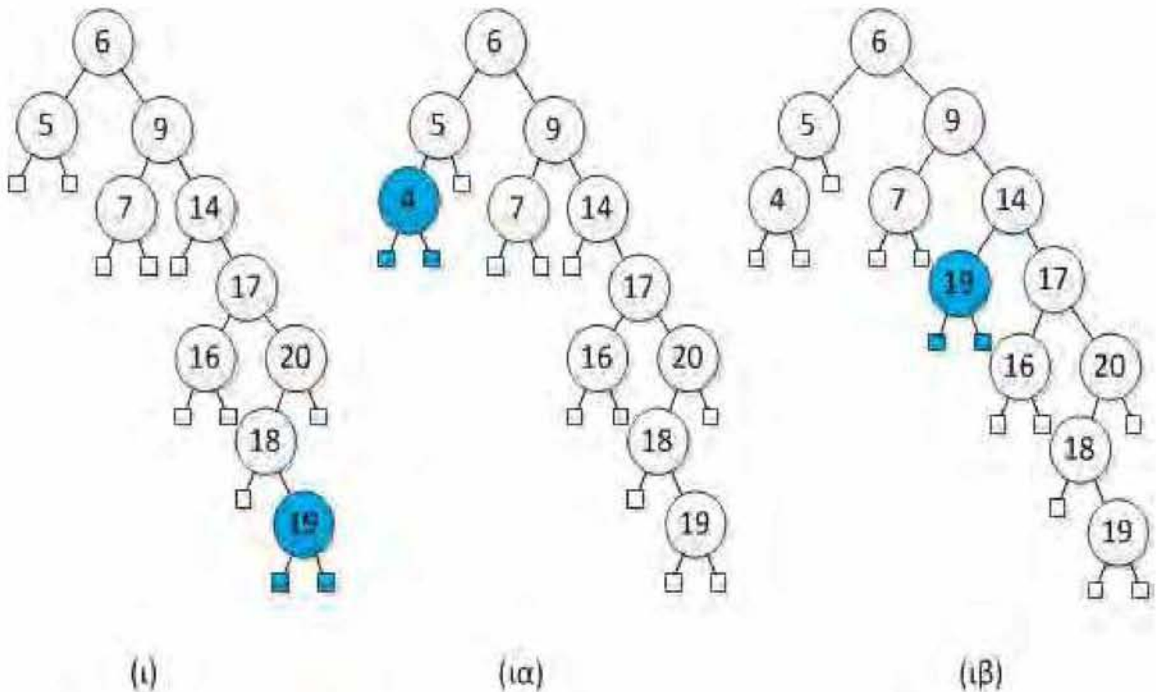
Εισαγωγή στοιχείου. Η εισαγωγή ενός στοιχείου i περιλαμβάνει μια αναζήτηση βάσει του κλειδιού του x. Εάν το δέντρο διαθέτει ήδη αντικείμενο με κλειδί x, τότε η διαδικασία σταματά. Διαφορετικά, τοποθετείται στην θέση του κατάλληλου κενού φύλλου του κόμβου που επιστρέφει η διαδικασία ψαξίματος, έτσι ώστε να ισχύει η αμετάβλητη συνθήκη των δυαδικών δέντρων αναζητήσεως:

```
Algorithm INSERTITEM (Item i)
Input:    Ένα στοιχείο i
Output:   Ο κόμβος όπου το I θα τοποθετηθεί, εάν δεν υπάρχει ήδη
1.   insNode = FindNode((x = κλειδί του i), ρίζα του δέντρου);
2.   if (το κλειδί του insNode == x)
3.        return null;
4.   else if (το κλειδί του insNode > x) {
```

```
5.          δημιούργησε έναν νέο κόμβο w ως αριστερό παιδί του insNode και
            τοποθέτησε το i;
6.          return w;
7.     }
8.     else {
9.          δημιούργησε έναν νέο κόμβο w ως δεξιό παιδί του insNode και
            τοποθέτησε το I;
10.         return w;
11.    }
end of      INSERTITEM
```



Σχήμα 2.3: (α)-(θ) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 16, 20, 18

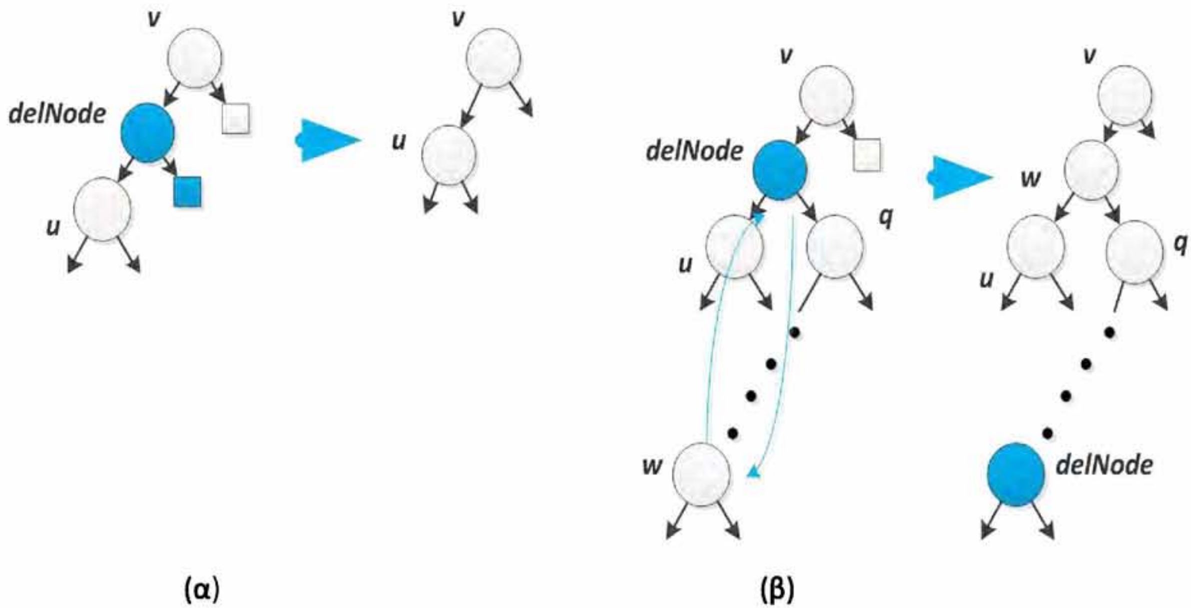


Σχήμα 2.4: (Συνέχεια) (ι)-(ιβ) 19, 4, 11.

Διαγραφή στοιχείου. Η διαγραφή στοιχείου i είναι ελάχιστα δυσκολότερη. Περιλαμβάνει και αυτή μια αναζήτηση βάσει του κλειδιού x. εάν το δέντρο δεν διαθέτει αντικείμενο με κλειδί x, τότε η διαδικασία σταματά. Διαφορετικά, έστω delNode ο κόμβος που περιέχει το i. Διακρίνουμε δυο περιπτώσεις:

a) Εάν ο delNode διαθέτει τουλάχιστον ένα κενό φύλλο, προκειμένου να διατηρηθεί η αμετάβλητη συνθήκη που διέπει το δέντρο, καταργείται ο delNode και το δεύτερο, ενδεχομένως μη κενό, παιδί του u συνδέεται με τον πατέρα y του delNode (Σχ. 2.4(α))

b) Διαφορετικά, ο delNode διαθέτει μη κενό δεξιό γιο q (Σχ. 2.4(β)). Οπότε, ανταλλάσσουμε το στοιχείο (Item) του delNode με αυτό του βαθύτερου, αριστερότερου μη κενού απογόνου w του δεξιού υποδέντρου T_q. Κατά αυτόν τον τρόπο, είναι σας οι w και delNode να άλλαξαν θέσεις! Οπότε, δημιουργούνται οι συνθήκες της περιπτώσεως (a), η οποία και εφαρμόζεται, ώστε ο delNode να διαγραφεί, δίχως παραβίαση της αμετάβλητης συνθήκης.



Σχήμα 2.4: Περιπτώσεις αποσβέσεως: (α) άμεση, και (β) ανταλλαγή με τον αριστερότερο απόγονο του δεξιού υποδέντρου

Τυπικότερα έχουμε:

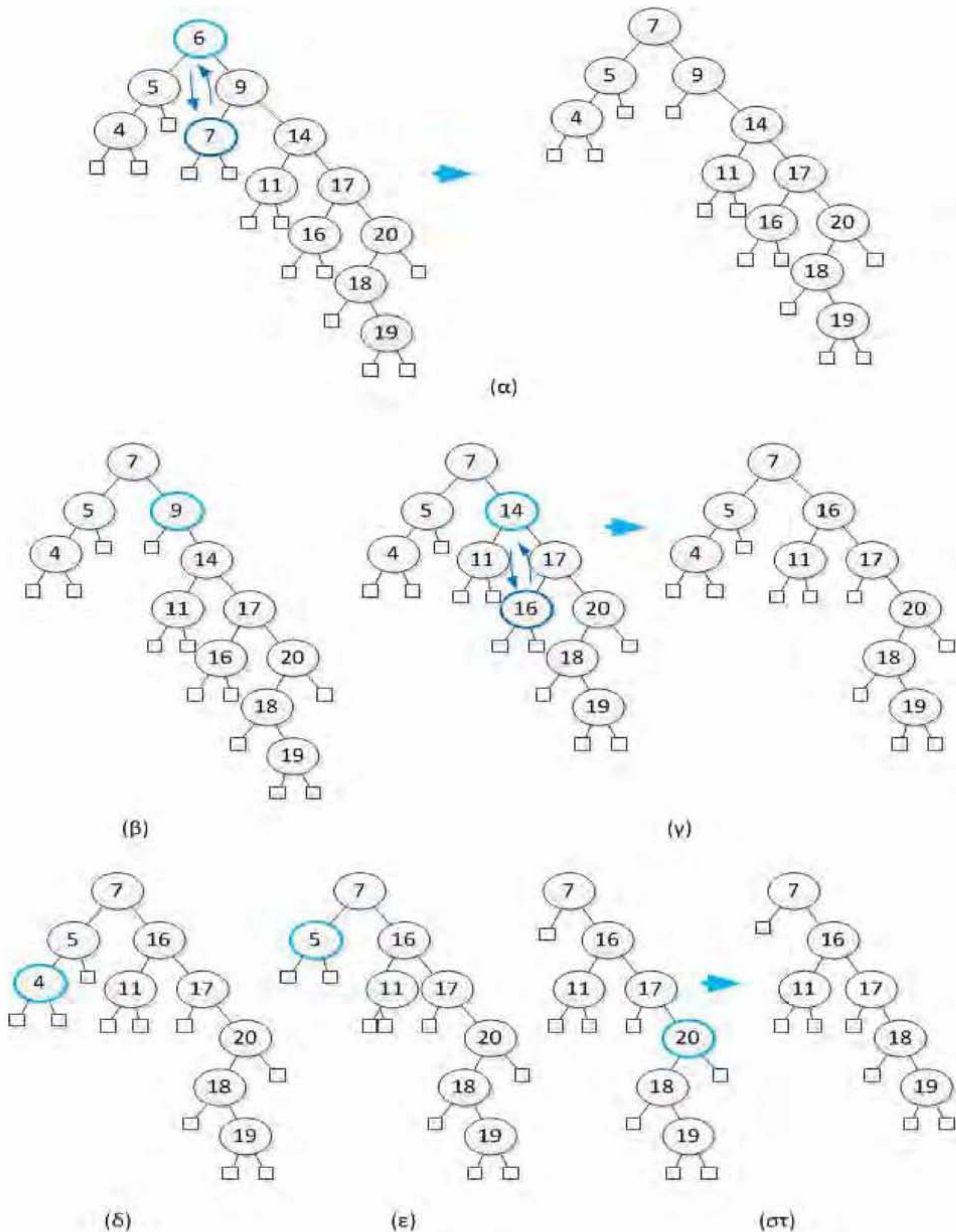
Algorithm DELETEITEM (Object k)

Input: Ένα κλειδί k

Output: Η απόσβεση του κόμβου που περιέχει item με κλειδί k, επιστρέφοντας τον εναπομείναντα εμπλεκόμενο κόμβο

```
1.  delNode = FindNode (k, ρίζα του δέντρου);
2.  if (το κλειδί του delNode != k)
3.      return null;
4.  if (delNode έχει τουλάχιστον ένα κενό παιδί) {
5.      σβήσε το delNode;
6.      επέστρεψε τον άλλο γιο;
7.  }
8.  else {
9.      βρες τον κόμβο w με το μικρότερο κλειδί του δεξιού υποδέντρου του
        delNode;
10.     αντάλλαξε τα στοιχεία τους;
11.     σβήσε τον w;
12. }
end of  DELETEITEM
```

Το Σχήμα 2.5 εμφαίνει παραδείγματα 6 διαδοχικών αποσβέσεων στο δέντρο του Σχήματος 2.3(ιβ). Παρατηρήστε πως αντιμετωπίζονται οι «δύσκολες» περιπτώσεις των 6 και 14 –Σχήματα (α) και (γ), αντιστοίχως.

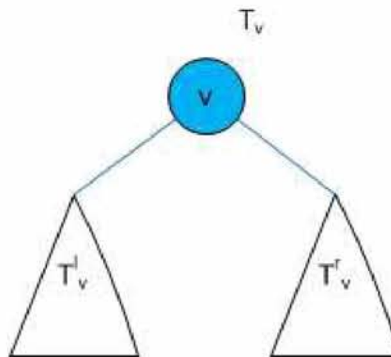


Σχήμα 2.6: (β) - (στ) Διαδοχικές αποσβέσεις των 9, 14, 4, 5, 20

2.1.2 Ανάλυση Πολυπλοκότητας

Χειρότερη Περίπτωση. Καθίσταται φανερό πως η πράξη της αναζητήσεως κυριαρχεί στην πολυπλοκότητα, καθώς από αυτή εξαρτώνται και η ένθεση και η απόσβεση. Όριο δε στην πολυπλοκότητα της αποτελεί το ύψος του εμπλεκόμενου δέντρου, επειδή η διαδικασία επεξεργάζεται ένα μονοπάτι αναζητήσεως, το οποίο ενδεχομένως να καταλήξει σε φύλλο. Από την άλλη το ύψος του δέντρου μπορεί να είναι γραμμικό στο πλήθος των αποθηκευτικών στοιχείων –π.χ. δοκιμάστε διαδοχικές ενθέσεις στοιχείων, ώστε τα κλειδιά να σχηματίζουν γνησίως αύξουσα ακολουθία –καθώς δεν παίρνουμε κανένα μέτρο να διορθώσουμε την τυχούσα ασυμμετρία. Οπότε ,

Θεώρημα 2.1: Σε ένα δυναμικό, αζύγιστο, δυαδικό δέντρο αναζητήσεως οι πράξεις της αναζητήσεως, της ενθέσεως και της αποσβέσεως κοστίζουν, στην χειρότερη περίπτωση, γραμμικό, στο πλήθος του υποκειμένου συνόλου, χρόνου.



Σχήμα 2.7: Τα δύο υποδέντρα του T_v

Μέση Περίπτωση. Προκειμένου να αναλύσουμε την μέση συμπεριφορά των αζύγιστων δέντρων αναζητήσεως, θα χρειαστούμε δυο χαρακτηριστικά μεγέθη του: το **εσωτερικό μήκος μονοπατιού (internal path length)**, το οποίο ορίζεται ως το άθροισμα των βαθών των κόμβων του, και το **εξωτερικό μήκος μονοπατιού (external path length)**, το οποίο σχηματίζεται με το άθροισμα των βαθών όλων των κενών (null) φύλλων.

Λήμμα 2.1 Έστω $Q_i(T_u)$ και $Q_e(T_u)$, αντίστοιχα το εσωτερικό και το εξωτερικό μήκος μονοπατιού ενός δέντρου T_u με ρίζα τον κόμβο u το μέγεθος του οποίου σε πλήθος κόμβων είναι $|T_u|$, ενώ διαθέτει αριστερό και δεξιό υποδέντρο T_u^r και T_u^l αντίστοιχα. Τότε ισχύουν τα εξής:

i. $Q_i(T_u) = Q_i(T_u^l) + Q_i(T_u^r) + |T_u| - 1$

ii. $Q_e(T_u) = Q_e(T_u^l) + Q_e(T_u^r) + |T_u| - 1$

iii. $Q_e(T_u) = Q_i(T_u^l) + 2|T_u|$ [σελ. 62 -2]

Στη συνέχεια θα ασχοληθούμε με το μέσο μήκος εσωτερικού μονοπατιού σε αζύγιστο δέντρο αναζήτησης T, μεγέθους $|T| = n$ στοιχείων. Θα θεωρήσουμε πως το Tα είναι ένα τυχαίο δέντρο αναζήτησης. Έστω λοιπόν, «$a_1, a_2, a_3, \ldots, a_n$» τα κλειδιά που μετέχουν στην δημιουργία του Tα, διατεταγμένα κατά αύξουσα τιμή. Η «τυχαιότητα» προκύπτει από την θεώρηση ότι το Tα χτίστηκε μέσω n διαδοχικών ενθέσεων και τα n συμμετέχοντα κλειδιά είναι ανεξάρτητοι και ομοιόμορφα κατανεμημένοι τυχαίοι αριθμοί.

Με άλλα λόγια, και οι n! μεταθέσεις τους έχουν την ίδια πιθανότητα να αποτελέσουν ακολουθία εισόδου, ή, ισοδύναμα, κάθε στοιχείο a_i έχει πιθανότητα $1/n$ να αποθηκευτεί στην ρίζα του δέντρου, καθώς υπάρχουν $(n-1)!$ Ακολουθίες ενθέσεων που ξεκινούν με το a_i ως πρώτο εισαχθέν κλειδί. Σημειώστε πως η τοποθέτηση του i-στου μικρότερου στοιχείου a_i στην ρίζα συνεπάγεται, εξ ορισμού των δυαδικών δέντρων αναζητήσεως, ότι το αριστερό υποδέντρο $T_{a_i}^l$ θα περιέχει $i-1$ κλειδιά, ενώ το δεξιό υποδέντρο $T_{a_i}^r$ θα στεγάζει $n-i$ κλειδιά.

Λήμμα 2.2 Το μέσο εσωτερικό μήκος ενός μονοπατιού σε ένα τυχαίο δυαδικό δέντρο αναζητήσεως T, n στοιχείων, είναι $\approx 1.39n\log n$, ενώ το μέσο εξωτερικό μήκος του είναι $\approx 1.39n\log n + 2n$. [σελ. 62 - 2]

Λήμμα 2.3 Έστω T ένα τυχαίο δυαδικό δέντρο αναζητήσεως μεγέθους n. Το αναμενόμενο κόστος ενός επιτυχημένου και ενός αποτυχημένου ψαξίματος είναι λογαριθμικό στο πλήθος n των στοιχείων. [σελ. 62 - 2]

Κλείνοντας, πρέπει να επισημαίνουμε πως (α) αποδεικνύεται ότι και το μέσο ύψος των τυχαίων αυτών δέντρων είναι λογαριθμικό, ενώ (β) αποφύγαμε να αναφερθούμε στο μέσο κόστος ενθέσεως και αποσβέσεως. Το τελευταίο οφείλεται στην υπόθεση του χτισίματος του τυχαίου δέντρου μέσω *διαδοχικών ενθέσεων*. Εάν παρεμβάλλονται και αποσβέσεις, τότε η ανάλυση είναι πολύ δύσκολη, ενώ είναι δυνατόν να παραχθούν δέντρα με μέσο ύψος $\sqrt{n}$.

ΚΕΦΑΛΑΙΟ 3 - ΤΥΧΑΙΕΣ ΔΟΜΕΣ ΛΕΞΙΚΟΥ

3.1 Treap

Ένα **treap** (**tr**ee και h**eap**), εν σχέση με τα δέντρα αναζητήσεως, είναι πώς κάθε στοιχείο *item*, πέραν του κλειδιού *key*, σχετίζεται και με μια προτεραιότητα *priority*, ήτοι έναν αριθμό από ένα ολικώς διατεταγμένο σύμπαν, ίσως διακριτό από το αντίστοιχο σύμπαν των κλειδιών. Οπότε, έχουμε τον ακόλουθο ορισμό:

Έστω ένα σύνολο στοιχείων S, συσχετιζόμενα το καθένα με ένα κλειδί και μια προτεραιότητα. Τότε, ένα treap T επί του S αποτελεί ένα δυαδικό δέντρο αναζήτησης, όπου οι κόμβοι, ως προς τα κλειδιά, ακολουθούν την συμμετρική διάταξη, ενώ, ως προς τις προτεραιότητες, σέβονται την διάταξη σωρού.

Ο ανωτέρω ορισμός σημαίνει, πως εάν ξεχάσουμε τις προτεραιότητες, το T αποτελεί ένα δέντρο αναζητήσεως βάσει κλειδιού, ενώ, ως προς τις προτεραιότητες, ένας κόμβος x, που αποτελεί πατέρα ενός κόμβου y, διαθέτει στοιχείο με μικρότερο αριθμό προτεραιότητας από αυτόν του y.

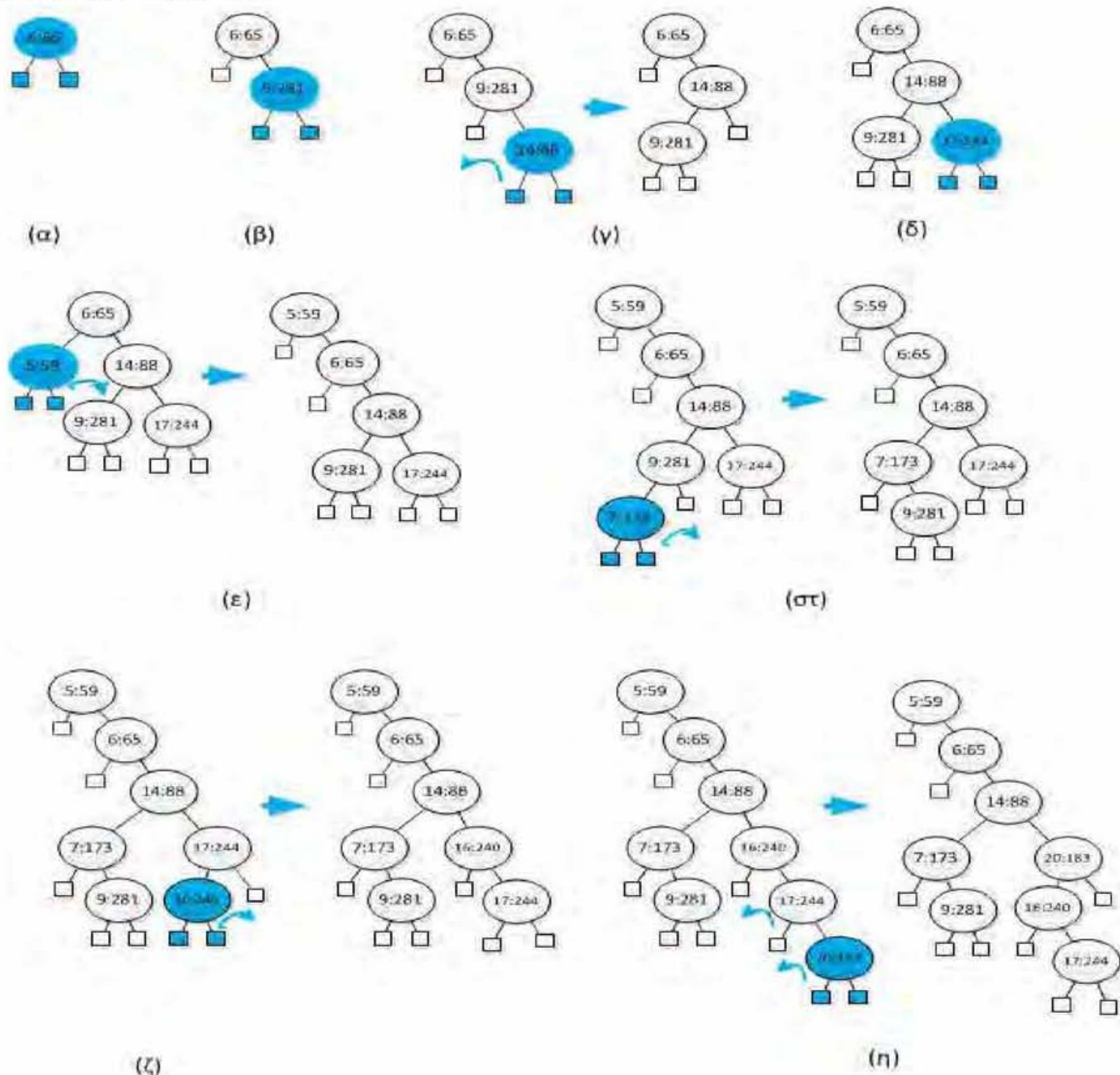
Ισοδύναμα, για διακριτές προτεραιότητες, ο ορισμός σημαίνει πως το στοιχείο s_{min}, με το μικρότερο αριθμό προτεραιότητας, τίθεται στην κορυφή του δέντρου, ενώ το αριστερό υποδέντρο $T_{s_<}$ και το δεξιό υποδέντρο $T_{s_>}$, αποτελούν δομές treap για το σύνολο στοιχείων $S_<$, με μικρότερατου s_{min} κλειδιά, και το σύνολο $S_>$, με μεγαλύτερα του s_{min} κλειδιά, αντιστοίχως. Ο αναγνώστης καλείται να επιβεβαιώσει ότι, ουσιαστικώς, πρόκειται για ένα δέντρο αναζητήσεως βάσει κλειδιού, όπου οι εισαγωγές των στοιχείων έχουν γίνει κατά σειρά προτεραιότητας!

3.1.1 Περιγραφή των Βασικών Πράξεων

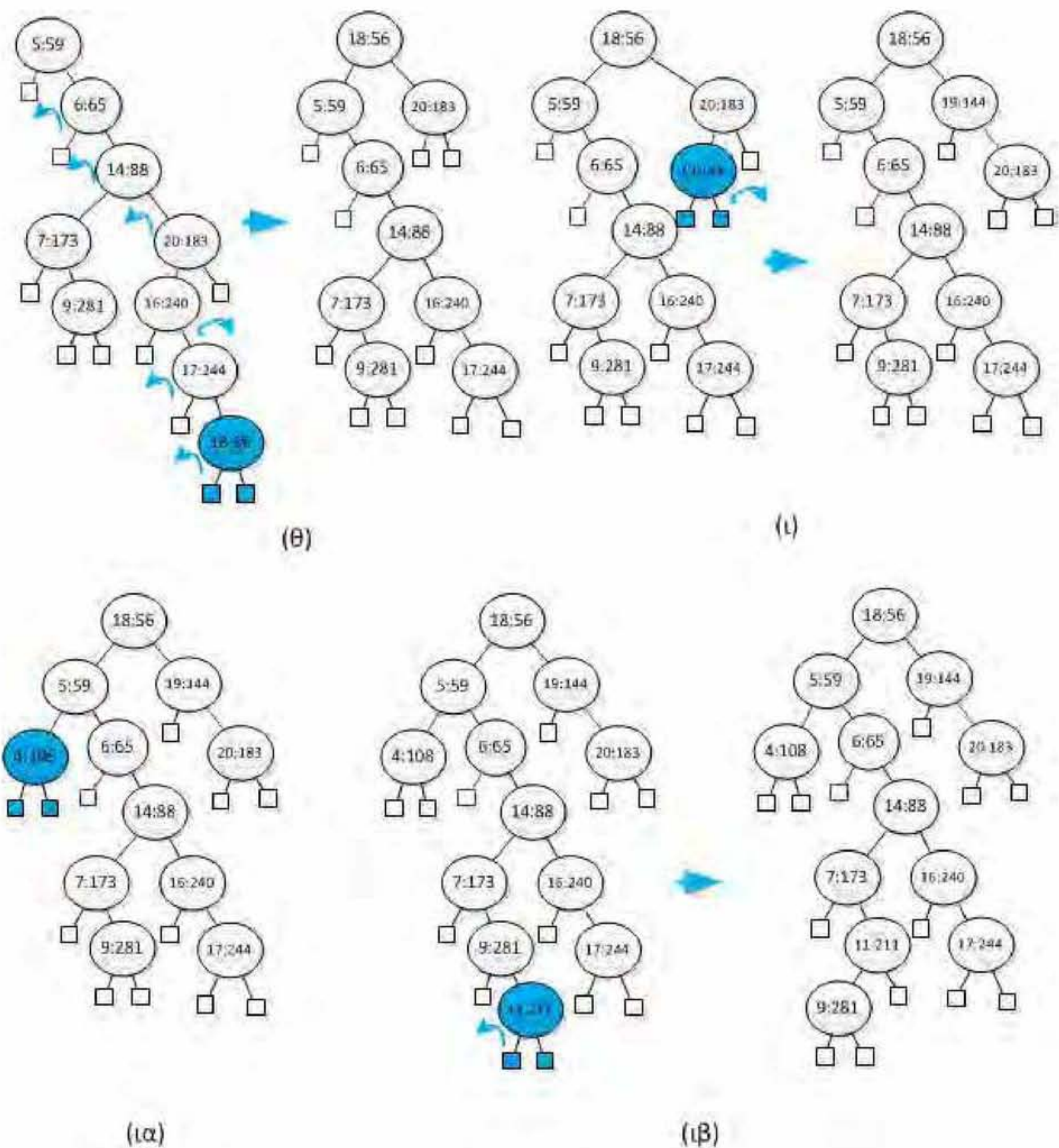
Αναζήτηση Στοιχείου. Η αναζήτηση ενός στοιχείου βάσει κλειδιού *key* σε ένα *treap,* λόγω ορισμού, είναι ίδια με αυτή των αζύγιστων δυαδικών δέντρων αναζητήσεως: Εκκινούμε από την ρίζα και, αγνοώντας τις προτεραιότητες, κατερχόμαστε του δέντρου, κάνοντας συγκρίσεις με τα κλειδιά, μέχρι είτε να εντοπίσουμε τον κόμβο είτε να καταλήξουμε σε κενό φύλλο.

Εισαγωγή Στοιχείου. Η ένθεση ενός στοιχείου *x*, με κλειδί *key*, είναι πολύ απλή στην σύλληψή της: Ενθέτουμε το *x* βάσει του *key,* όπως και στα απλά δέντρα αναζητήσεως, σε ένα κόμβο-φύλλο *insNode,* δίνοντας του μια προτεραιότητα. Κατόπιν, όσο η συνθήκη διατάξεως σωρού παραβιάζεται και δεν ξεπερνούμε την ρίζα, προάγουμε τον *insNode* με αριστερές ή δεξιές απλές περιστροφές, αναλόγως εάν είναι δεξιό ή αριστερό παιδί αντίστοιχα.

Η απλότητα της μεθόδου φαίνεται και στα Σχήματα 3.1 και 3.2, τα οποία αφορούν 12 διαδοχικές ενθέσεις σε ένα αρχικώς άδειο treap. Χαρακτηριστικα, αναφέρουμε την περίπτωση (θ) του 18, του οποίου η μικρή προτεραιότητα το οδηγεί μέχρι την ρίζα.



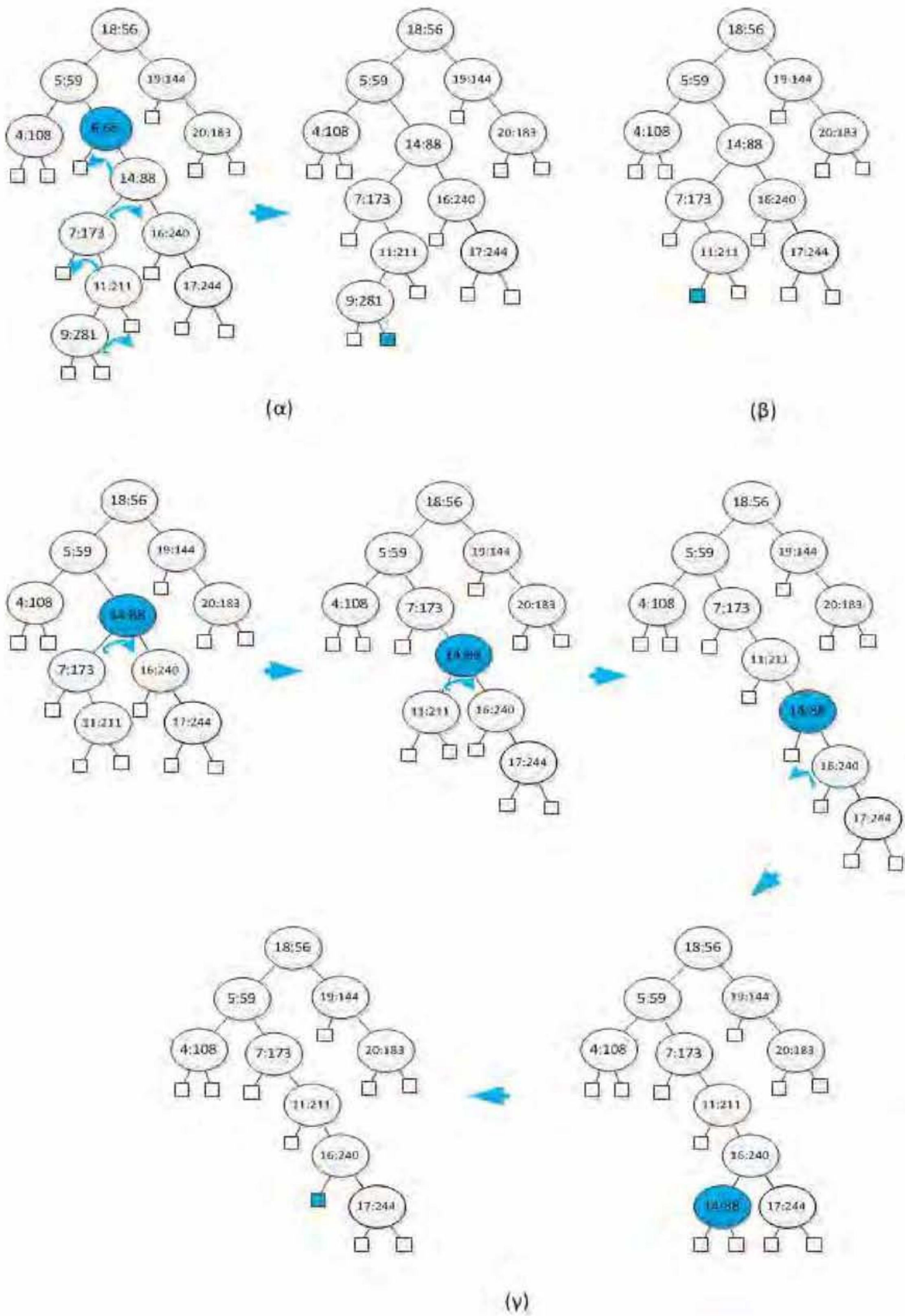
Σχήμα 3.1: (α)-(η) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 16, 20



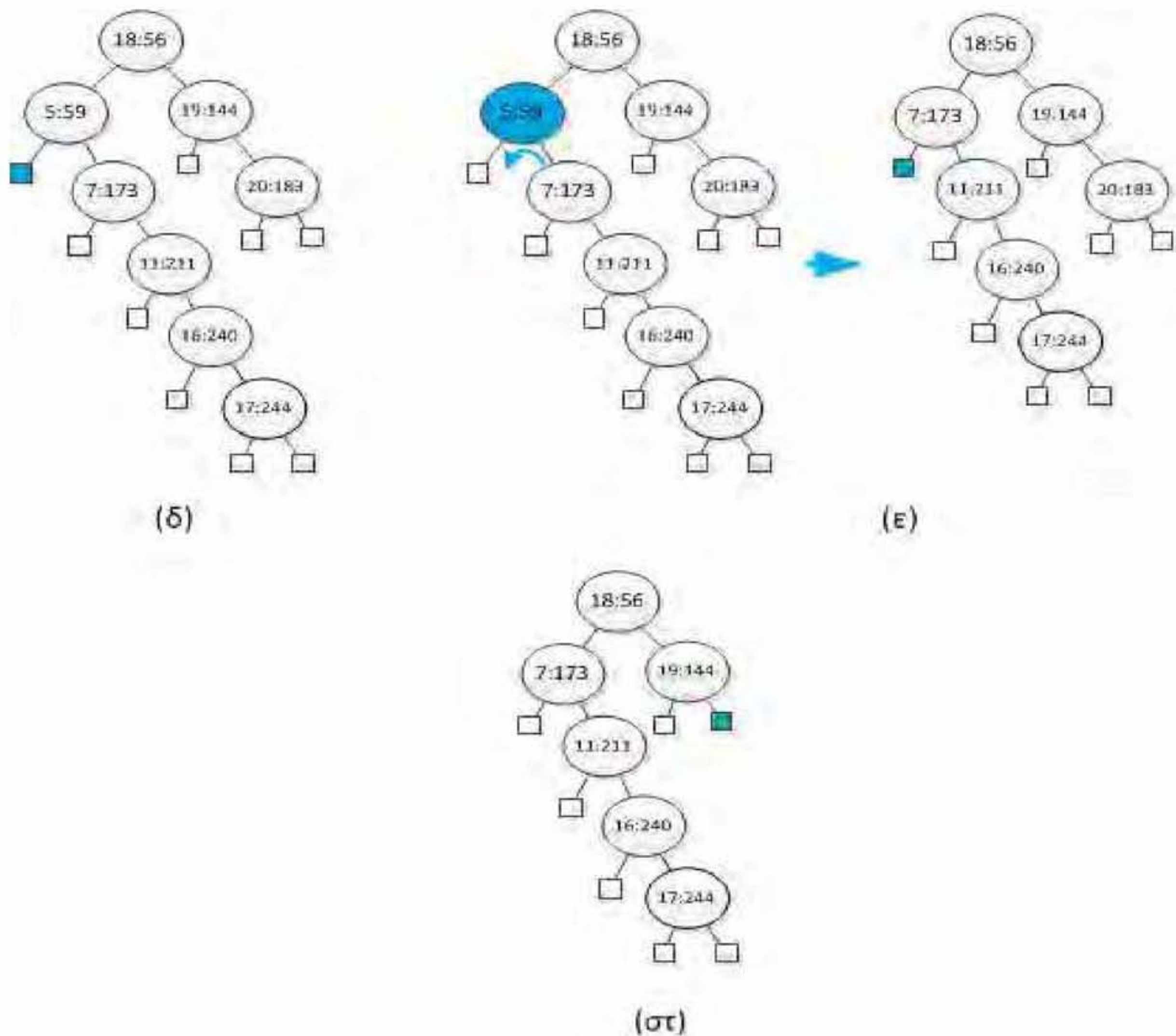
Σχήμα 3.2: (Συνέχεια) (θ)-(ιβ) 18, 19, 4, 11

Απόσβεση Στοιχείου. Η διαδικασία αποσβέσεως είναι συμμετρική αυτής της ενθέσεως· αφού εντοπιστεί ο κόμβος delNode που στεγάζει το εμπλεκόμενο στοιχείο, υποβιβάζεται με διαδοχικές, αριστερές ή δεξιές απλές περιστροφές, μέχρι ότου καταστεί παιδί-φύλλο, οπότε και διαγράφεται. Η «κατεύθυνση» της περιστροφής, σε κάθε περίπτωση, προσδιορίζεται, έτσι ώστε να προαχθεί το παιδί με τον μικρότερο αριθμό προτεραιότητας. Στα Σχήματα 3.3 και 3.4 εικονίζονται τα αποτελέσματα πέντε διαδοχικών αποσβέσεων στην δομή του Σχήματος 3.2(ιβ). Η περίπτωση (γ) του 14 παρουσιάζεται αναλυτικά. Αξίζει να παρατηρηθεί πως η φορά

των περιστροφών είναι αντίθετη προς το είδος (αριστερό ή δεξιό) του παιδιού με την μικρότερη προτεραιότητα.



Σχήμα 3.3: (α)-(γ) Διαδοχικές αποσβέσεις των 6, 9 και 14



Σχήμα 3.4: (Συνέχεια) (δ)-(στ) 4, 5 και 20.

3.1.2 Ανάλυση Πολυπλοκότητας

Προφανώς, μία δομή treap, επί του συνόλου S n στοιχείων, απαιτεί γραμμικό, στο πλήθος των στοιχείων, χώρο, ενώ το βάθος (ισοδύναμα, ύψος) της δομής καθορίζει και το κόστος μιας αναζήτησης και μίας ενημερώσεως (ενθέσεως ή αποσβέσεως).

Λήμμα 3.1 Το αναμενόμενο βάθος ενός treap n στοιχείων είναι λογαριθμικό. [σελ. 62 -2]

Μάλιστα, είναι δυνατόν να αποδειχθεί ότι η λογαριθμικότητα του βάθους ισχύει με μεγάλη πιθανότητα.

Λήμμα 3.2 Το αναμενόμενο κόστος και των τριών πράξεων σε μια δομή treap, n στοιχείων, είναι λογαριθμικό. [σελ. 62 -2]

Κλείνοντας, αξίζει να αναφερθεί πως μία δομή treap είναι εξίσου απλή με μία δομή λίστας αναπηδήσεως, ενώ παρουσιάζει τα ίδια χαρακτηριστικά απόδοσης. Ένα μικρό πλεονέκτημά της είναι πως η υλοποίηση της απαιτεί χώρο ακριβώς n, αλλά τούτο δεν αποτελεί καίριο χαρακτηριστικό. Ίσως, πάλι, η δημιουργία και αποθήκευσή της τυχαίας προτεραιότητας για κάθε στοιχείο, να αποτελεί ένα μειονέκτημα της δομής.

ΚΕΦΑΛΑΙΟ 4 - ΒΑΘΜΟ-ΕΞΑΡΤΩΜΕΝΕΣ ΟΥΡΕΣ ΠΡΟΤΕΡΑΙΟΤΗΤΩΝ

Οι **Βαθμο-Εξαρτώμενες Ουρές Προτεραιοτήτων (**Rank-Sensitive Priority **queues)** είναι οι δομές δεδομένων οι οποίες πάντα γνωρίζουν το **ελάχιστο στοιχείο** από τα δεδομένα που εμπεριέχουν. Στα επόμενα κεφάλαια θα αναλύσουμε δύο προσεγγίσεις –την τυχαιοποιημένη και την αναπόσβεστη προσέγγιση.

4.1 Τυχαιοποιημένη προσέγγιση

Η πρώτη ιδέα που έρχεται στο νου όταν προσπαθούμε να χτίσουμε μια βαθμο-εξαρτώμενη ουρά προτεραιοτήτων είναι εάν μια απλή δομή σωρού είναι επαρκής ή όχι. Η εισαγωγή και η διαγραφή ενός στοιχείου x σε μια δομή σωρού μπορεί εύκολα να υλοποιηθεί σε χρόνο ανάλογο με το ύψος, το οποίο είναι $O(\log n)$ για το ελάχιστο στοιχείο και $O(1)$ για τα περισσότερα υψηλόβαθμα στοιχεία στη δομή. Ωστόσο, εάν το μέγιστο στοιχείο στο αριστερό υποδέντρο της ρίζας είναι μικρότερο από το μικρότερο στοιχείο στο δεξί υποδέντρο της ρίζας, είναι πιθανό να καταλήξουμε με, το μέσο στοιχείο που είναι και δεξί παιδί της ρίζας, για το οποίο ο χρόνο που απαιτείται για την διαγραφή του είναι $O(\log n)$ αντί για $O(1)$ που απαιτείται για μια βαθμο-εξαρτώμενη δομή. Η τυχαιότητα μας δίνει ένα τρόπο να διορθώσουμε αυτό το πρόβλημα, δίνοντας ένα απλή και «κομψή» υλοποίηση μιας βαθμο-εξαρτώμενης ουράς προτεραιοτήτων όπου η εισαγωγή και η διαγραφή «τρέχουν» σε χρόνο $O\big(log(n/r)\big)$.

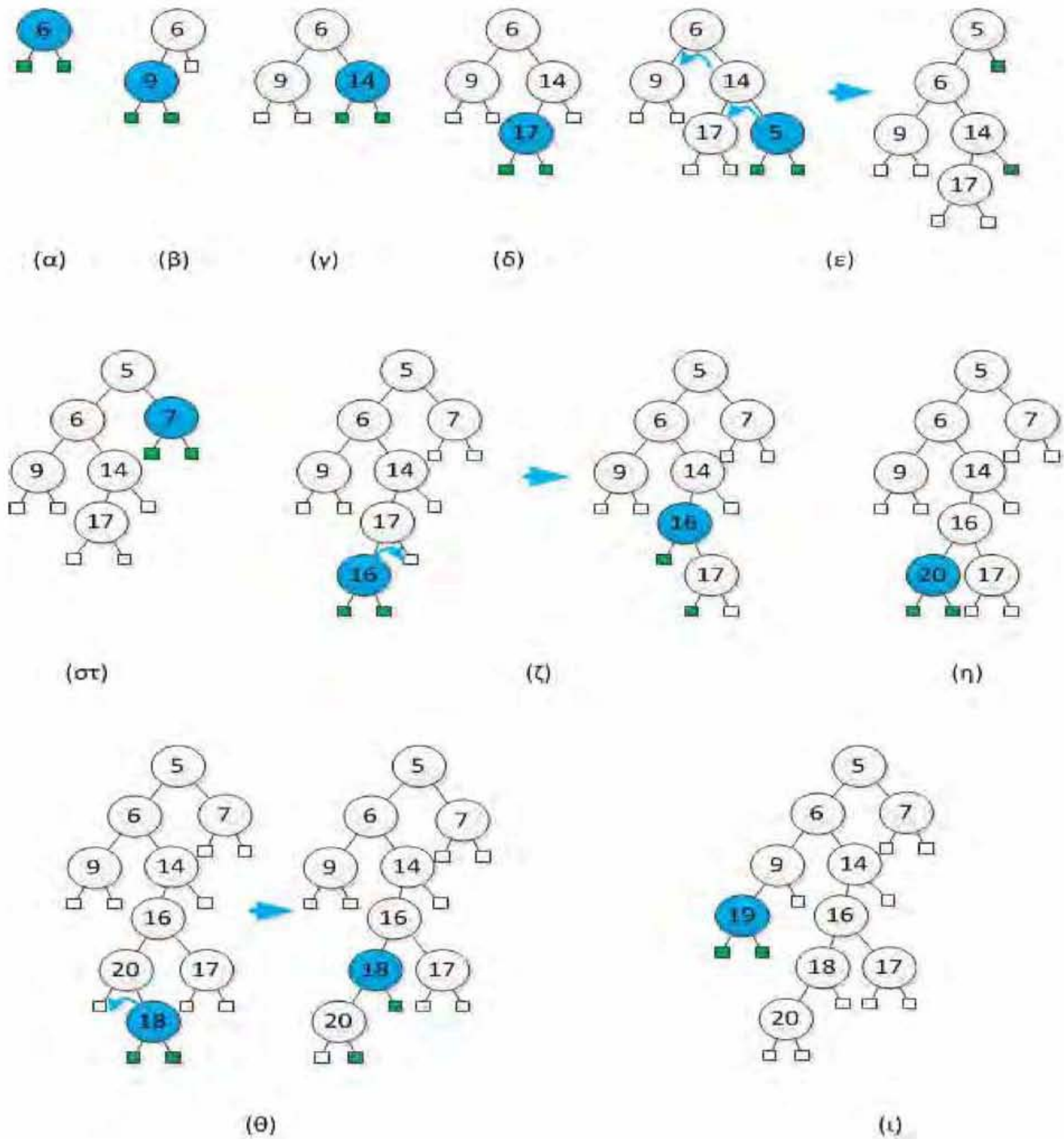
4.1.1 Περιγραφή των Βασικών Πράξεων

Αναζήτηση Στοιχείου. Για την αναζήτηση ενός στοιχείου χρησιμοποιούμε μια δομή hash και υποθέτουμε ότι ο χρόνος αναζήτησης είναι $O(1)$.

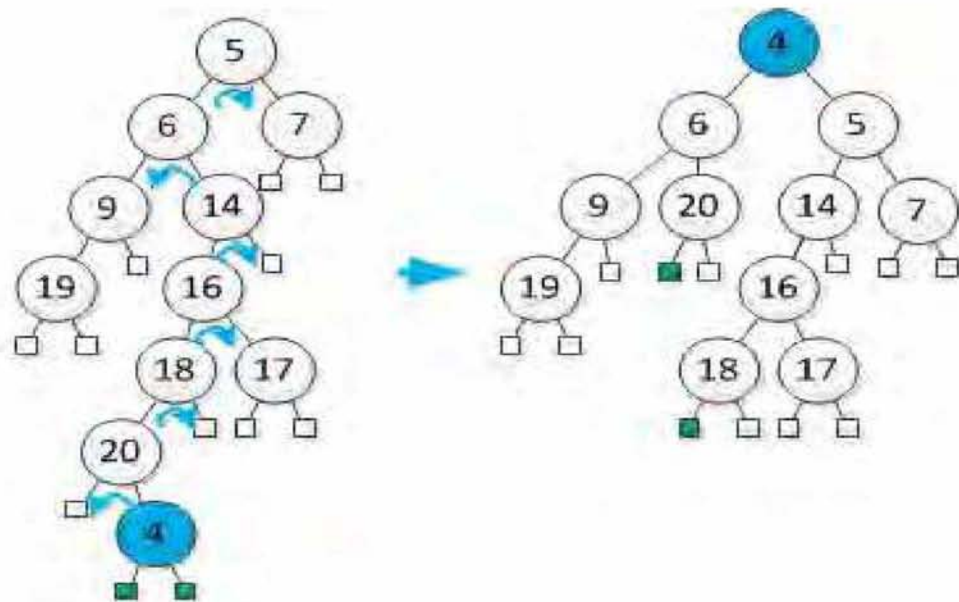
Εισαγωγή Στοιχείου. Για να εισάγουμε ένα στοιχείο *n*, με κλειδί *key*, σε ένα $(n-1)$-στοιχείων δέντρο, το τοποθετούμε σε μια από τις n κενές θέσεις στο κάτω μέρος του δέντρου, η επιλογή γίνεται τυχαία σε $O(1)$ χρόνο, και έπειτα το μετακινούμε προς τα πάνω έως ότου η δομή σωρού (heap) να αποκατασταθεί. Η

μετακίνηση γίνεται με δεξιές ή αριστερές απλές περιστροφές αναλόγως αν είναι αριστερό ή δεξιό παιδί αντίστοιχα.

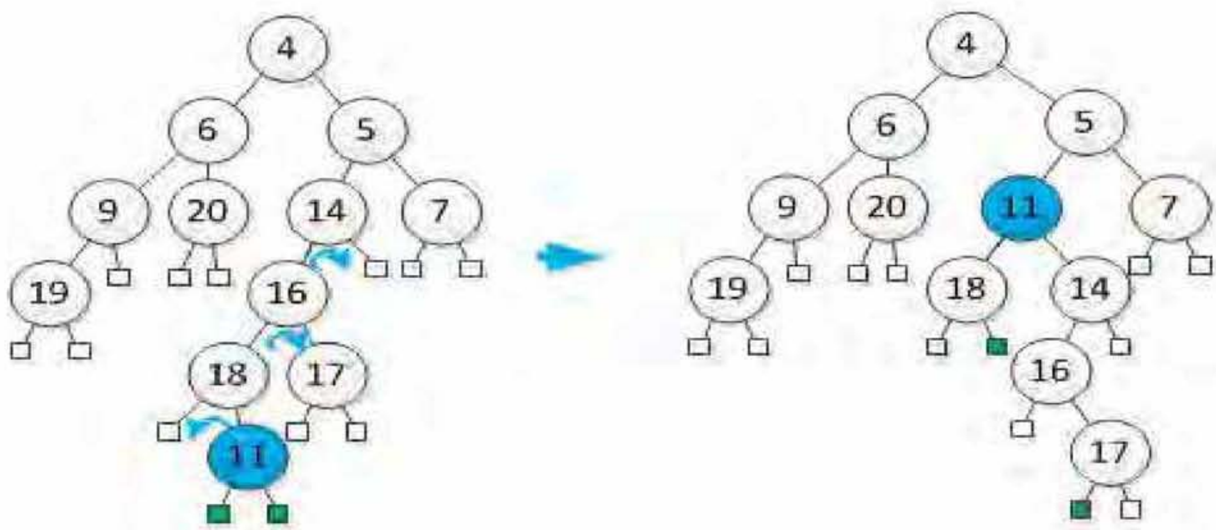
Στο παρακάτω σχήμα φαίνεται η εισαγωγή δώδεκα στοιχείων. Αξίζει να σημειωθεί ότι το στιγμιότυπο που παρουσιάζεται στο σχήμα είναι μοναδικό καθώς για δώδεκα εισαγωγές έχουμε 12! διαφορετικά δέντρα. Σημειώνεται επίσης ότι με τα τετράγωνα με έντονο χρώμα είναι τα οι νέοι κόμβοι που προστίθενται στη λίστα με τα παιδία των κόμβων.



Σχήμα 4.1: (α)-(ι) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 16, 20, 18, 19



(ια)



(ιβ)

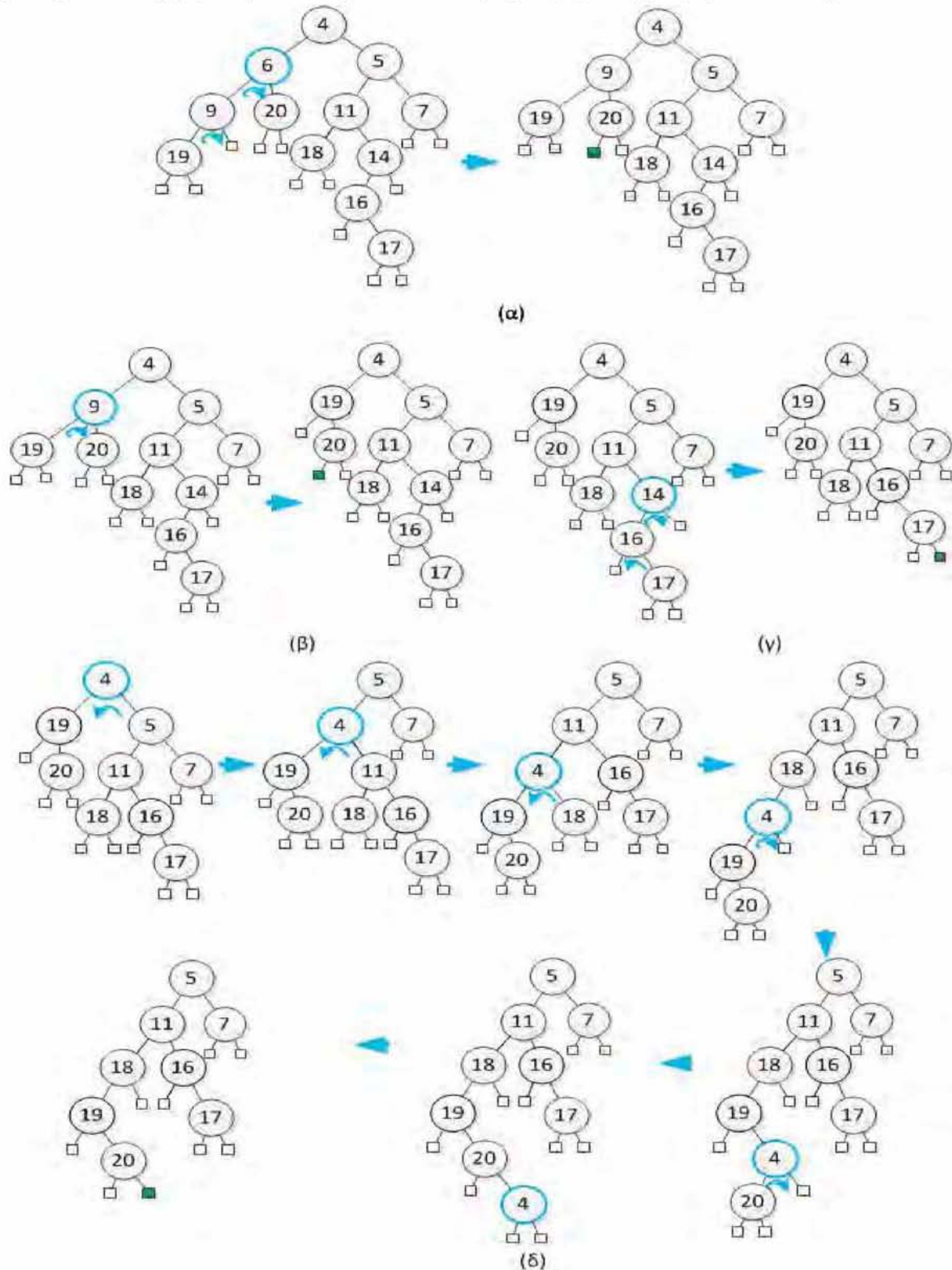
Σχήμα 4.2: (Συνέχεια) (ια)-(ιβ) 4, 11

Απόσβεση Στοιχείου. Για να αποσβέσουμε ένα στοιχείο, αφού πρώτα το αναζητήσουμε και το βρούμε επιτυχώς, θέτουμε το κλειδί του key $+\infty$ και το μετακινούμε προς τα κάτω –και σε αυτή την περίπτωση η μετακίνηση γίνεται με απλές (δεξιές ή αριστερές) περιστροφές.

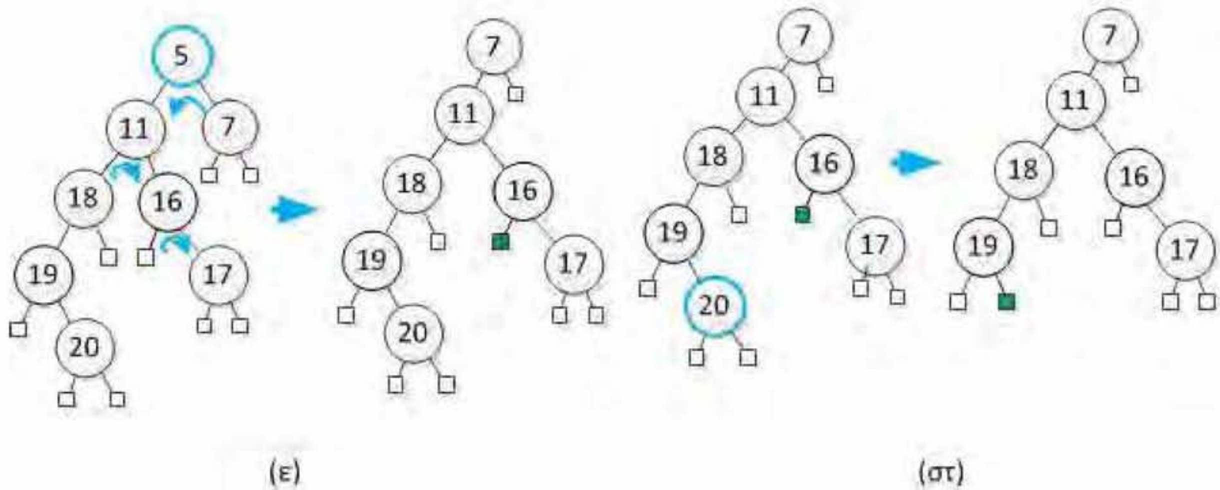
Η φορά της περιστροφής επιλέγεται με τέτοιο τρόπο ώστε να στην θέση του στοιχείου προς διαγραφή να έρθει το στοιχείο-παιδί με το μικρότερο κλειδί. Η

μετακίνηση θα τερματιστεί όταν το στοιχείο θα γίνει φύλλο (δεν θα έχει κανένα παιδί).

Στο παρακάτω σχήμα παρουσιάζεται η διαδοχική διαγραφή έξι στοιχείων. Στην περίπτωση (δ) παρουσιάζεται λεπτομερώς η διαγραφή του στοιχείου 4.



Σχήμα 4.3: (α)-(δ) Διαδοχικές αποσβέσεις των 6, 9, 14 και 4



Σχήμα 4.4: (Συνέχεια) (ε)-(στ) 5 και 20.

4.1.2 Ανάλυση Πολυπλοκότητας

Το κόστος εισαγωγής και διαγραφής σε μια δομή h-treap μπορεί εύκολα να αποδεδειχθεί ότι είναι λογαριθμικό. Έστω n το πλήθος των στοιχείων που βρίσκονται στη δομή και r ο βαθμός της δομής. Τότε μπορεί να αποδειχθεί το εξής θεώρημα:

Θεώρημα 4.1 Η διαδικασία της εισαγωγής και της διαγραφής σε μία δομή h-treap «τρέχει» σε χρόνο $O\big(log(n/r)\big)$. [σελ. 62 - 1]

Οι χρόνοι εισαγωγής και διαγραφής είναι ίσοι λόγω συμμετρίας, αφού ο χρόνος που απαιτείται για την εισαγωγή ενός στοιχείου βαθμού r είναι ακριβώς ίσος με τον χρόνο που απαιτείται για την διαγραφή του (η ακολουθία των περιστροφών της διαγραφής θα είναι αντίστροφη από τις αντίστοιχες της εισαγωγής).

4.1.3 Πειραματική Αξιολόγηση

4.1.3.1 Υπολογισμός Κόστους Στοιχειωδών Πράξεων

Στον κώδικα του παραρτήματος Α υλοποιούμε το h-treap. Παράλληλα εισάγουμε ορισμένες μεταβλητές και κάνουμε τις απαραίτητες μετρήσεις με σκοπό να αποδείξουμε την ορθότητα της θεωρητικής ανάλυσης. Οι μεταβλητές που εισάγαμε είναι οι εξής: access, rebalance, insertion_cost, deletion_cost.

Αναλυτικότερα έχουμε:

access: χρησιμοποιούμε την μεταβλητή αυτή για να υπολογίσουμε το κόστος της εκάστοτε ενθέσεως. Η μεταβλητή αυξάνεται κατά ένα σε κάθε βήμα της αναζήτησης. Υπενθυμίζουμε ότι στην εισαγωγή το κόστος εύρεσης είναι $O(1)$.

rebalance: χρησιμοποιείται για να υπολογίσουμε το κόστος επαναζύγισης. Η μεταβλητή αυξάνεται κατά ένα κάθε φορά που έχουμε περιστροφή στο δέντρο.
Ενδεικτικά παρατίθεται ο κώδικας της αριστερής περιστροφής με την χρήση της rebalance:

```
void rotate_up_left(struct Randomized *pointer){
        rebalance++;
        pointer->left->parent = pointer->parent;
        pointer->parent->right = pointer->left;
        pointer->left = pointer->parent;
        if(Rroot == pointer) pointer->parent = Rroot;
        else pointer->parent = pointer->parent->parent;
        pointer->left->parent = pointer;
}
```

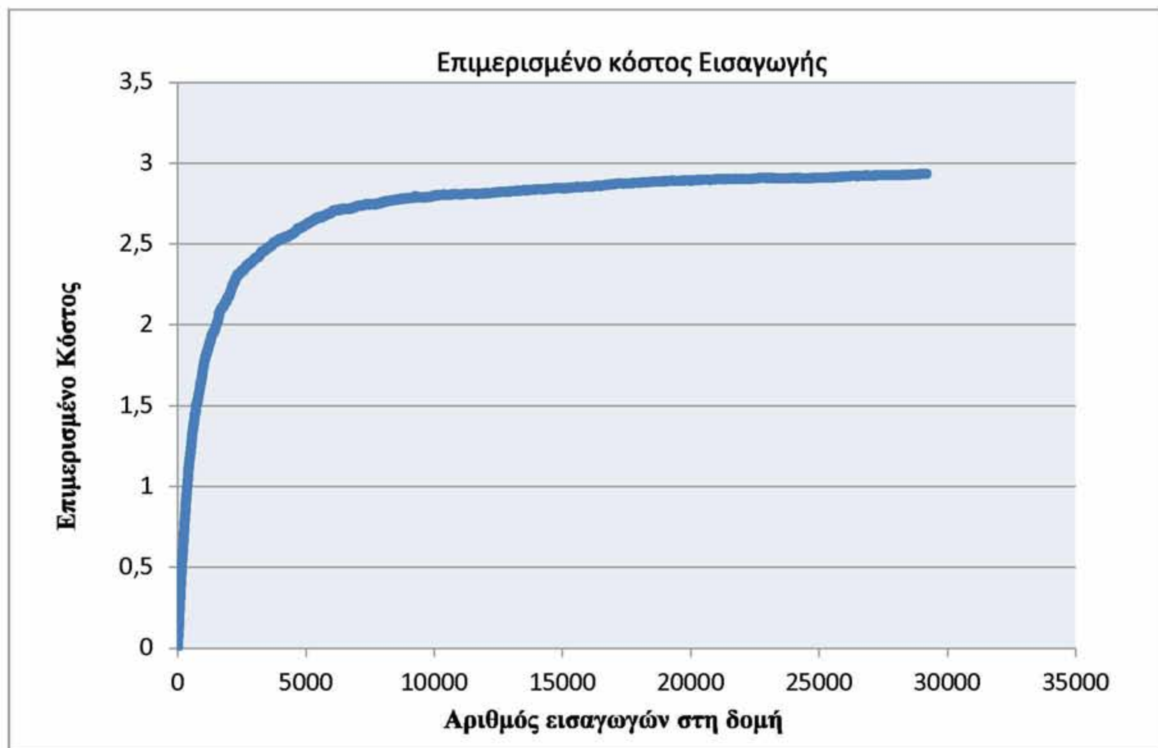
insertion_cost: υπολογίζει το συνολικό κόστος εισαγωγής στην δομή. Συγκεκριμένα, υπολογίζεται ως το άθροισμα $insertion_cost_{n-1} + rebalance_n + access_n$. Ο κώδικας του υπολογισμού της εισαγωγής είναι:

```
for(int i=0; i<nodes; i++){
        count_ins++;
        std::cout<<"Insert Node Data: ";
        std::cin>>val;
        Randomized_insert(val);
        Total_cost += (access_cost + reb_cost);
        fprintf(w,"%f\n",(double)Total_cost/count_ins);
        access_cost = 0;
        reb_cost = 0;
}
```

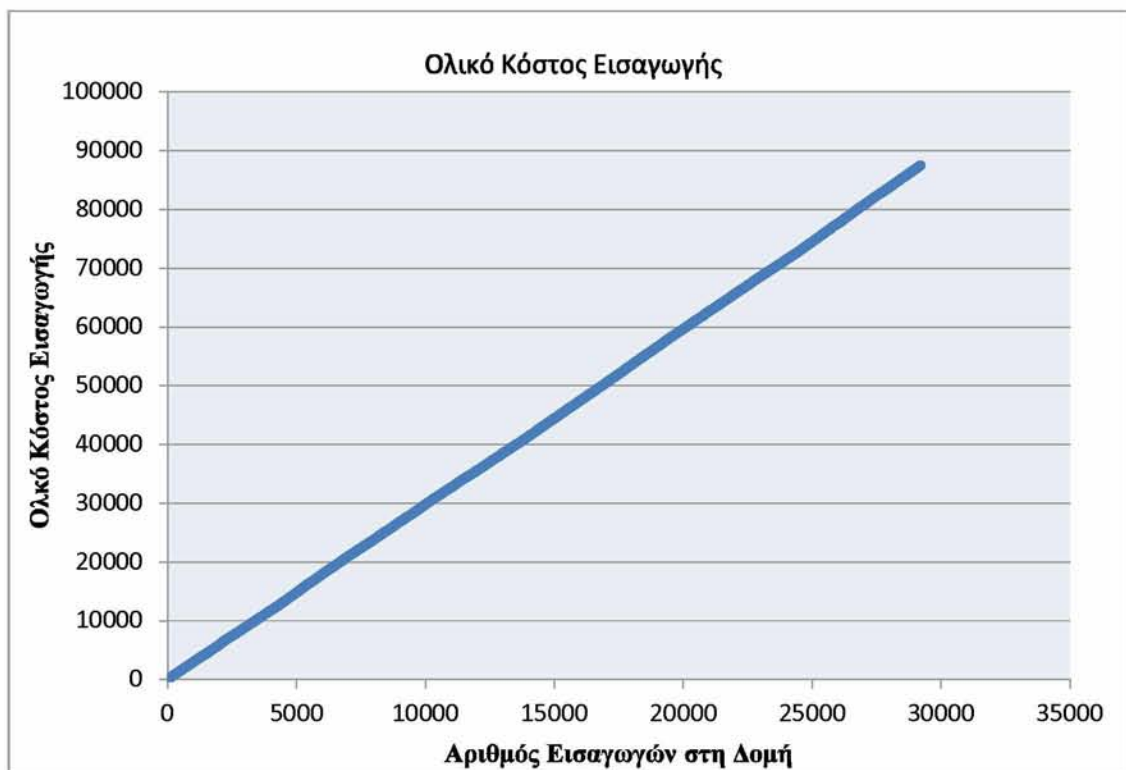
Σημείωση: count_ins θεωρούμε τον αριθμό των στοιχείων που βρίσκονται στη δομή αφού εισάγαμε το τελευταίο στοιχείο.

deletion_cost: αντίστοιχη μέθοδος με αυτή που χρησιμοποιήθηκε στο insertion_cost. Η μόνη διαφορά βρίσκεται στη διαίρεση του κόστους με τα στοιχεία, στην περίπτωση της διαγραφής ο αριθμός των στοιχείων μειώνεται.

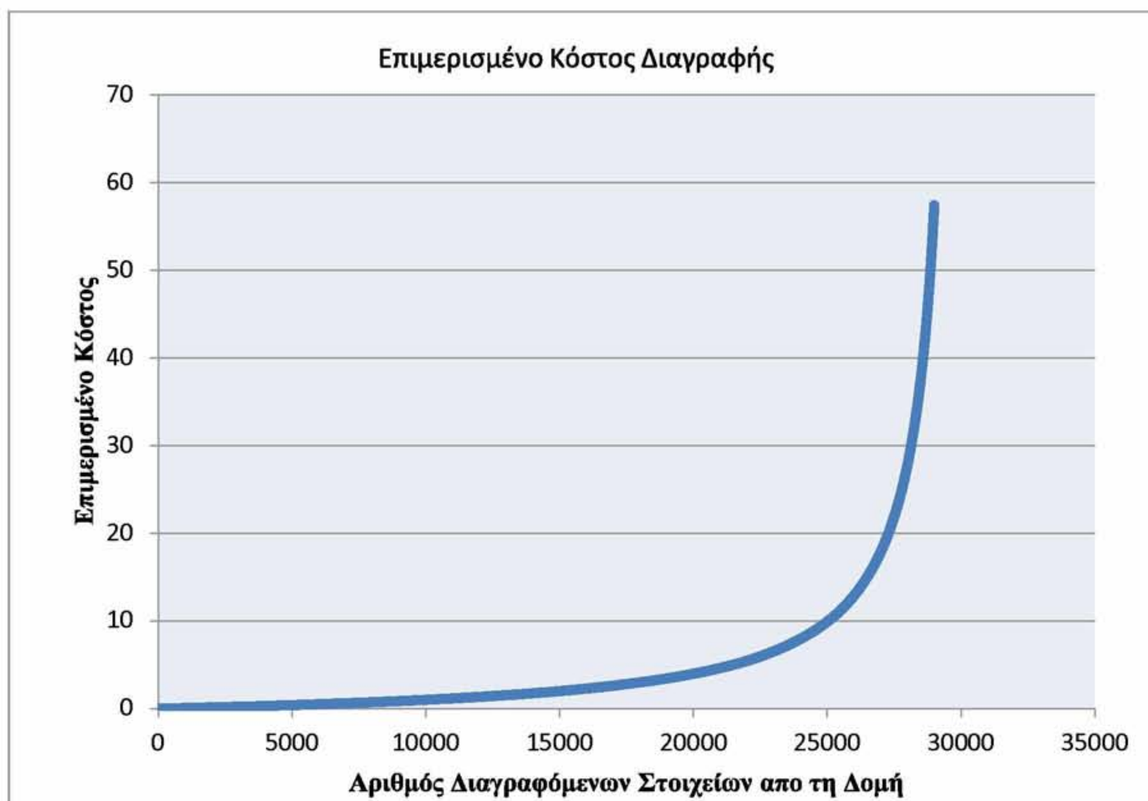
Τέλος για να υπολογίσουμε με αξιοπιστία την θεωρητική ανάλυση δημιουργήσαμε ένα αρχείο με 30.000 αριθμούς, το οποίο χρησιμοποιούμε για εισαγωγή και διαγραφή στοιχείων σε μια δομή. Από τα αποτελέσματα που εξάγαμε δημιουργήσαμε τις παρακάτω γραφικές παραστάσεις.



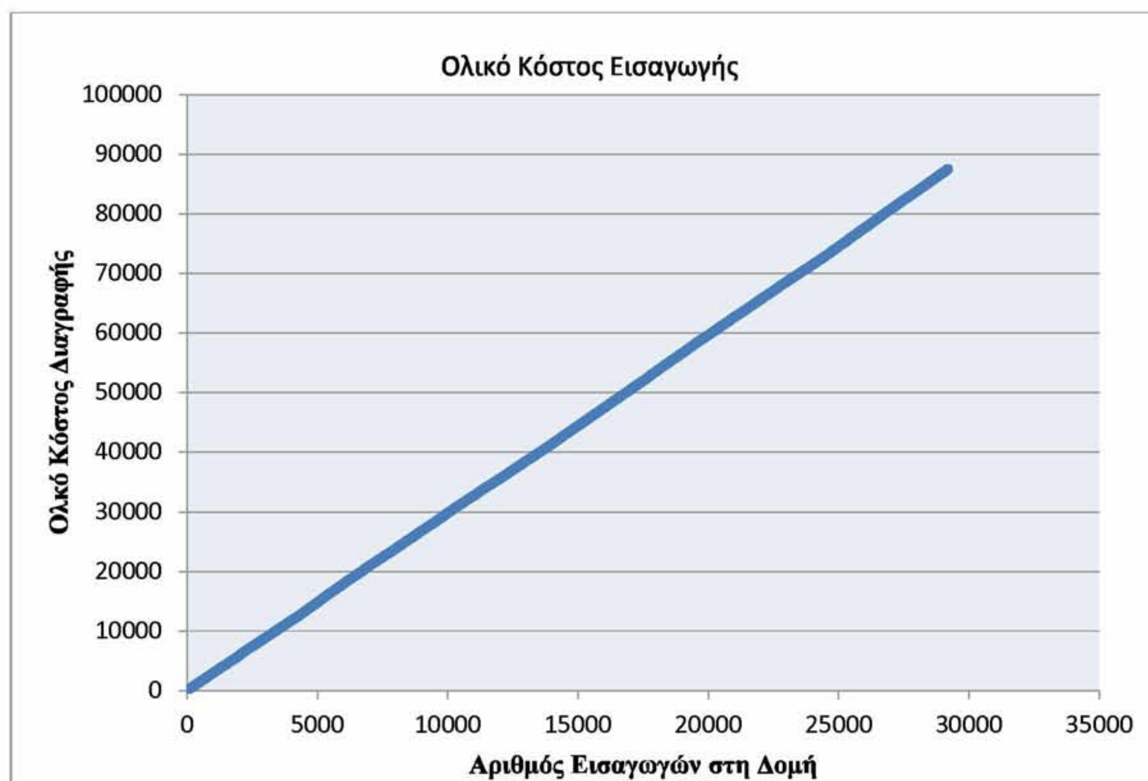
Σχήμα 4.5: Επιμερισμένο Κόστος Εισαγωγής



Σχήμα 4.6: Ολικό Κόστος Εισαγωγής



Σχήμα 4.7: Επιμερισμένο Κόστος Διαγραφής



Σχήμα 4.8: Ολικό Κόστος Διαγραφής

4.1.3.2 Μετρήσεις Πραγματικού Χρόνου

Ενσωματώνοντας στη δομή h-treap τη βιβλιοθήκη <time.h> μπορούμε να υπολογίσουμε τον πραγματικό χρόνο που χρειάστηκε η υλοποίηση για να ολοκληρώσει την διαδικασία. Οι μετρήσεις πραγματικού χρόνου έγιναν σε ένα σύστημα με τα εξής χαρακτηριστικά:

CPU: Intel Quad-Core Q8200

RAM: 4096MB

OS: MS Windows 7(64bit) **και** Ubuntu 11.04(32bit)

Στον παρακάτω πίνακες παρουσιάζονται ενδεικτικά μετρήσεις για εισαγωγές και διαγραφές:

	MS Windows 7		Ubuntu 11.04	
	Εισαγωγή	Διαγραφή	Εισαγωγή	Διαγραφή
10.000 στοιχεία	0.155 sec	1.468 sec	0.065 sec	0.591 sec
20.000 στοιχεία	0.347 sec	1.952 sec	0.157 sec	1.094 sec
30.000 στοιχεία	0.576 sec	2.254 sec	0.267 sec	1.493 sec
40.000 στοιχεία	0.837 sec	2.517 sec	0.470 sec	1.768 sec
50.000 στοιχεία	1.152 sec	2.613 sec	0.738 sec	1.928 sec

Πίνακας 4.1: Μετρήσεις Πραγματικού Χρόνου στη δομή h-treap

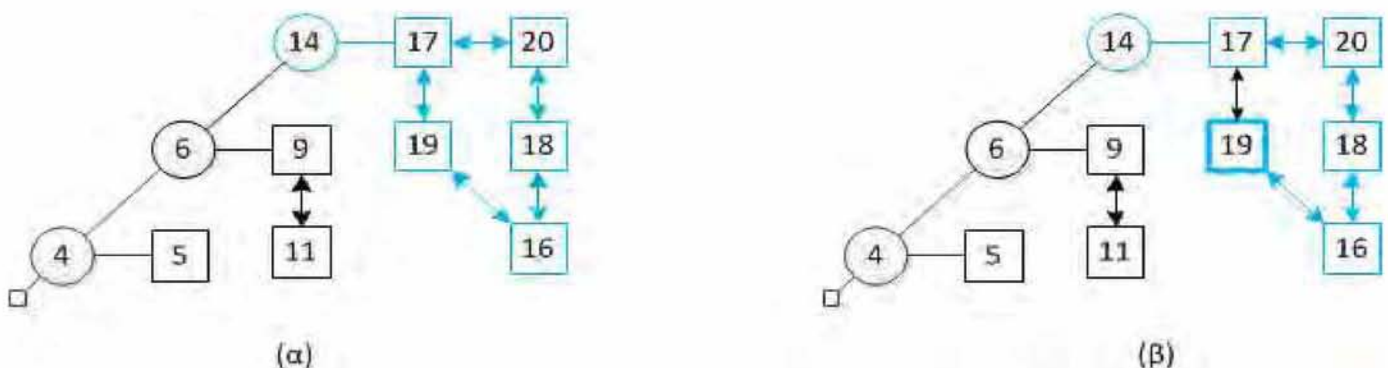
Σημείωση: οι διαγραφές πραγματοποιήθηκαν στην δομή αφού είχε εισαχθεί 50.000 στοιχεία

4.2 Αναπόσβεστη Προσέγγιση

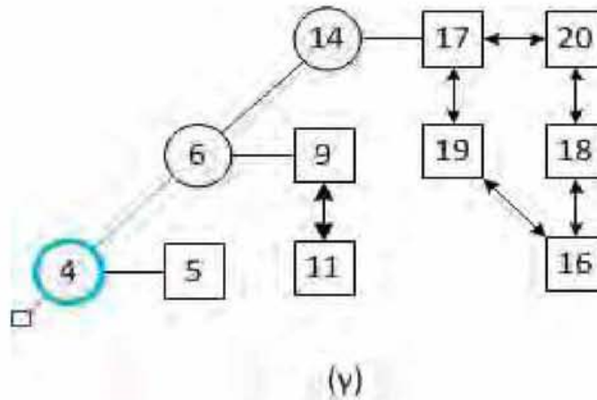
Για την προσέγγιση αύτη θα βασιστούμε εξολοκλήρου δυαδικά δέντρα αναζητήσεως (ΔΔΑ). Ως γνωστόν οι πράξεις της εισαγωγής και της διαγραφής σε ένα ΔΔΑ κοστίζει $O(\log n)$. Χρησιμοποιώντας τα συγκεκριμένα δέντρα θα «χτίσουμε» μια βαθμο-εξαρτώμενη ουρά προτεραιοτήτων με κόστος $O(log\, n/r)$. Έστω $s(x)$ το μέγεθος του υποδέντρου του x. Για κάθε $a \in [1/2, 1)$, λέμε ότι το x είναι α-ζυγισμένο εάν $s(left(x)) \leq as(x)$ και $s(right(x)) \leq as(x)$. Ένα δέντρο Τα καλείται α-ζυγισμένο εάν όλα τα στοιχεία του είναι α-ζυγισμένα. Μπορούμε να «χτίσουμε» μια αποτελεσματική βαθμο-εξαρτώμενη προτεραιότητα εάν «χαλαρώσουμε» τη δομή και αφήσουμε τα δεξιά υποδέντρα όλων των στοιχείων «άχτιστα», αποθηκεύοντας τα στοιχεία των «άχτιστων» υποδέντρων σε μια διπλή κυκλική λίστα.

4.2.1 Περιγραφή των Βασικών Πράξεων

Αναζήτηση Στοιχείου. Για την αναζήτηση ενός στοιχείου διατρέχουμε το δέντρο από την ρίζα προς τα κάτω έως ότου βρούμε το στοιχείο που ψάχνουμε. Η αναζήτηση είναι αντίστοιχη με την αναζήτηση ενός αζύγιστου δυαδικού δέντρου (κεφ. 2). Η διαφορά έγκειται στο γεγονός ότι τα δεξιά παιδία μένουν «άχτιστα», έτσι όταν χρειαστεί να πάμε σε δεξί παιδί η αναζήτηση αλλάζει και μετατρέπεται σε αναζήτηση διπλής κυκλικής λίστας. Προσοχή πρέπει να δοθεί εάν το στοιχείο που ψάχνουμε είναι το ελάχιστο της δομής, τότε μεταβαίνουμε απευθείας στο στοιχείο σε χρόνο $O(1)$.



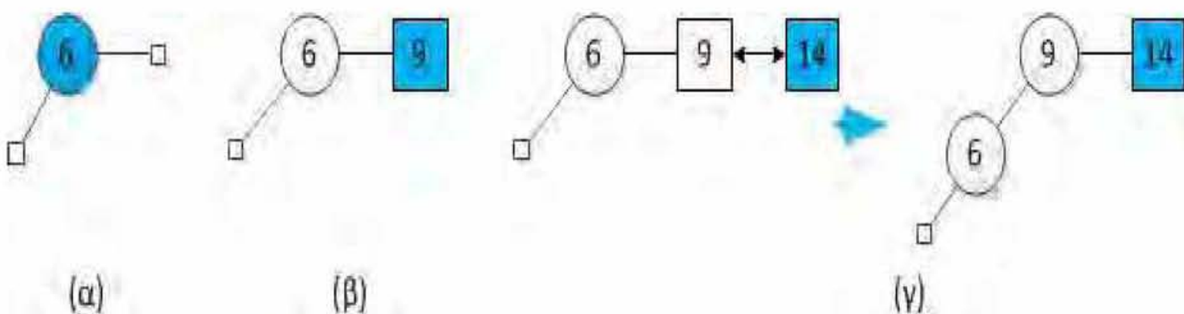
Σχήμα 4.9: Στιγμιότυπα: (α) αποτυχημένη αναζήτηση του 21, (β) επιτυχημένη αναζήτηση του 19.



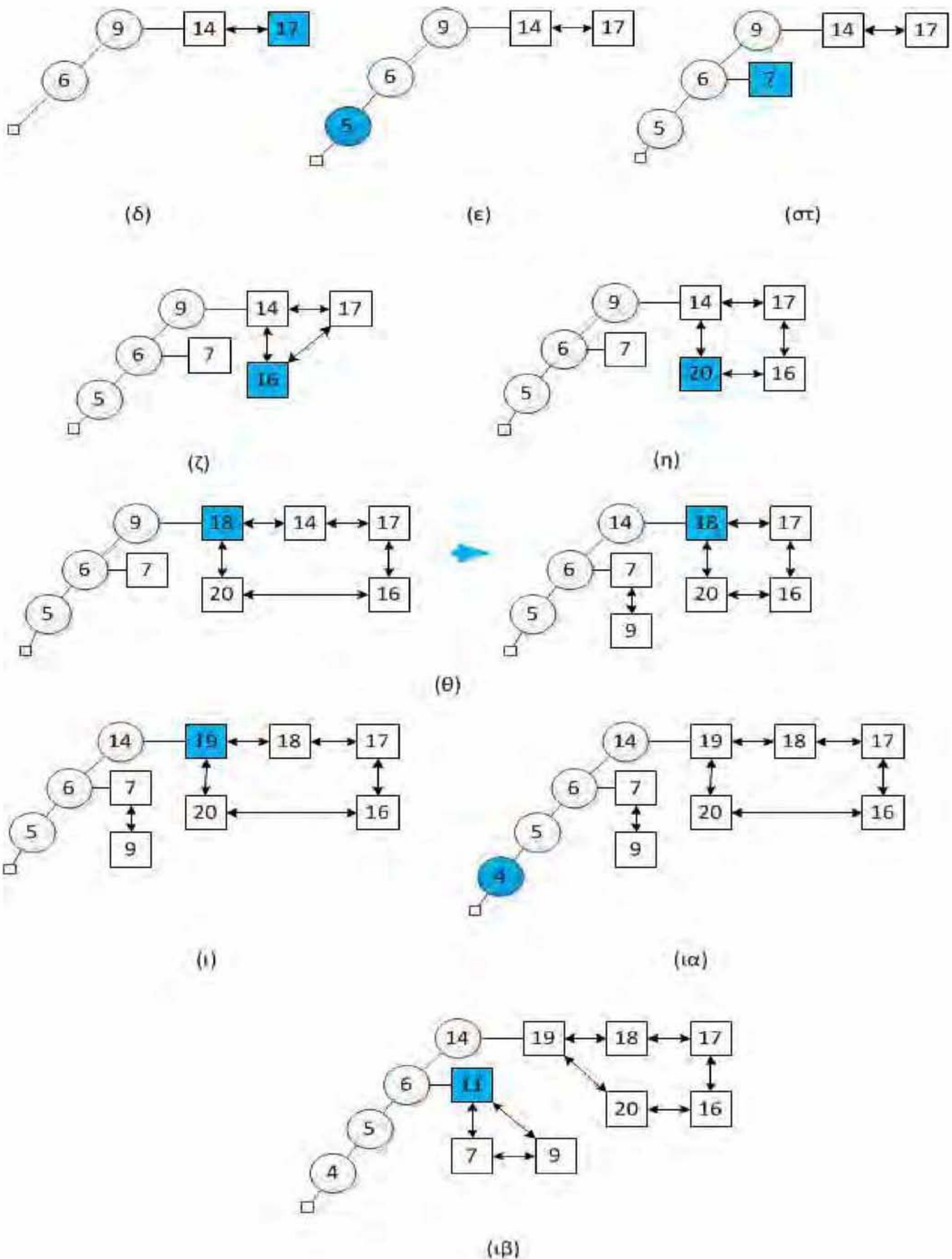
Σχήμα 4.10: Στιγμιότυπο: επιτυχημένη αναζήτηση ελαχίστου

Εισαγωγή Στοιχείου. Για να εισάγουμε ένα στοιχείο x, με κλειδί key, ελέγχουμε εάν το νέο στοιχείο είναι το νέο min. Στην περίπτωση που δεν είναι το νέο ελάχιστο στοιχείο της δομής διατρέχουμε το δέντρο έως ότου βρούμε στοιχείο y με κλειδί μικρότερο από το κλειδί του στοιχείου x. Όταν το στοιχείο βρεθεί τότε εισάγουμε το στοιχείο x στην αρχή της κυκλικής λίστας δεξιά του στοιχείου y. Εάν, όμως, το στοιχείο x είναι το νέο min τότε εισάγεται ως τελευταίο στοιχείο της δομής (αριστερά του min) σε χρόνο $O(1)$. Όταν εισαχθεί το στοιχείο ξεκινάμε από το σημείο εισαγωγής και διατρέχουμε το δέντρο προς τα πάνω ανανεώνοντας το μέγεθος του υποδέντρου και ελέγχουμε εάν πρέπει να έχουμε εξισορρόπηση. Σε περίπτωση που έχουμε «εξισορρόπηση» σε περισσότερα από ένα στοιχεία τότε επιλέγουμε το υψηλότερο.

Λίγη προσοχή πρέπει να δοθεί στην εξισορρόπηση. Όταν έχουμε εξισορρόπηση τότε καταστρέφουμε το δέντρο από το σημείο εξισορρόπησης και κάτω, κρατώντας τα στοιχεία του ταξινομημένα σε ένα πίνακα. Έπειτα επιλεγούμε το μέσο των στοιχείων και το τοποθετούμε στο σημείο εξισορρόπησης, στη συνέχεια τοποθετούμε τα μεγαλύτερα στοιχεία του ως δεξιά παιδία στην κυκλική λίστα. Συνεχίζοντας η ίδια διαδικασία ακολουθείται για τα στοιχεία που είναι μικρότερα του μέσου.

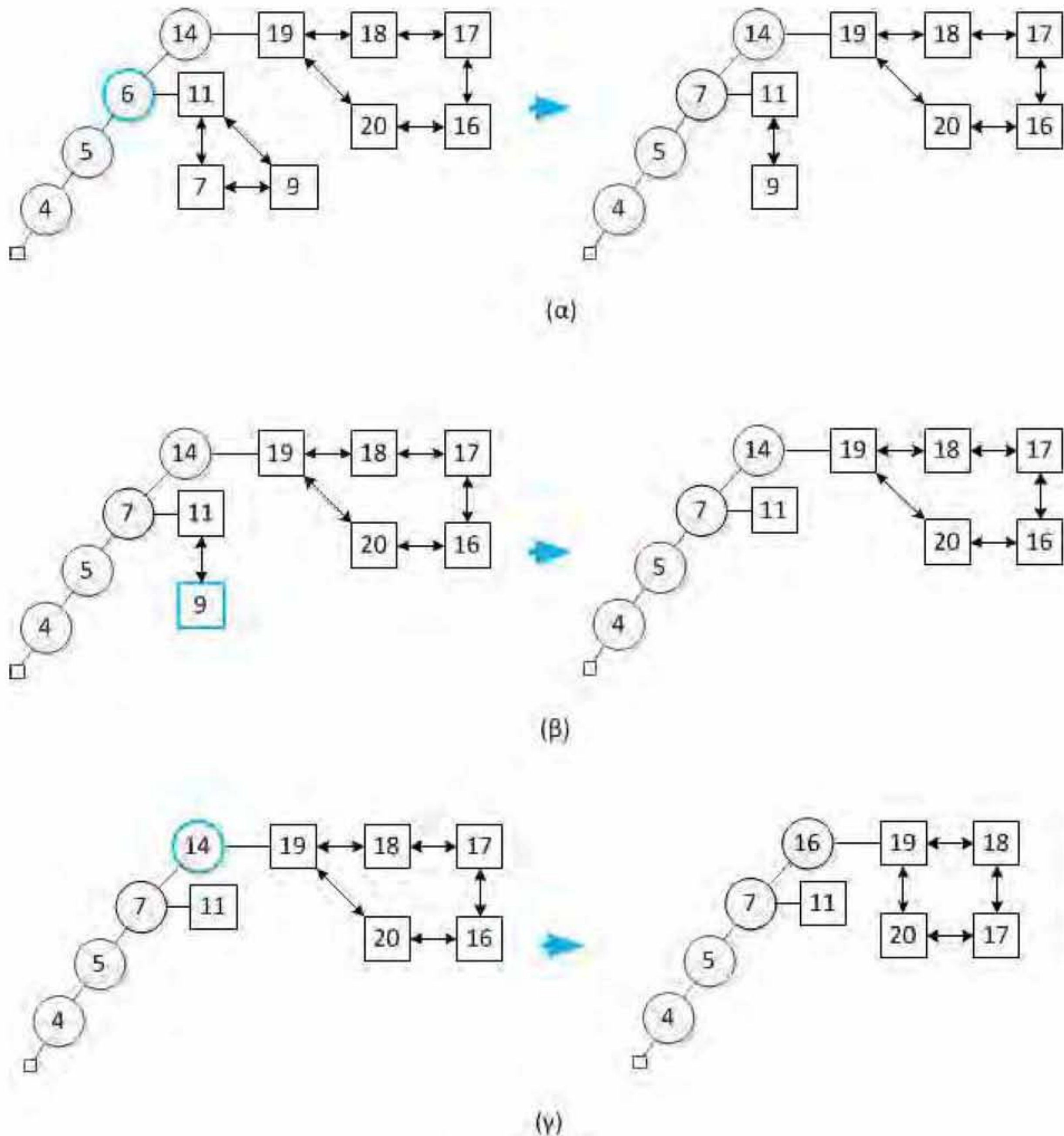


Σχήμα 4.11: (α)-(γ) Διαδοχικές ενθέσεις των 6, 9, 14

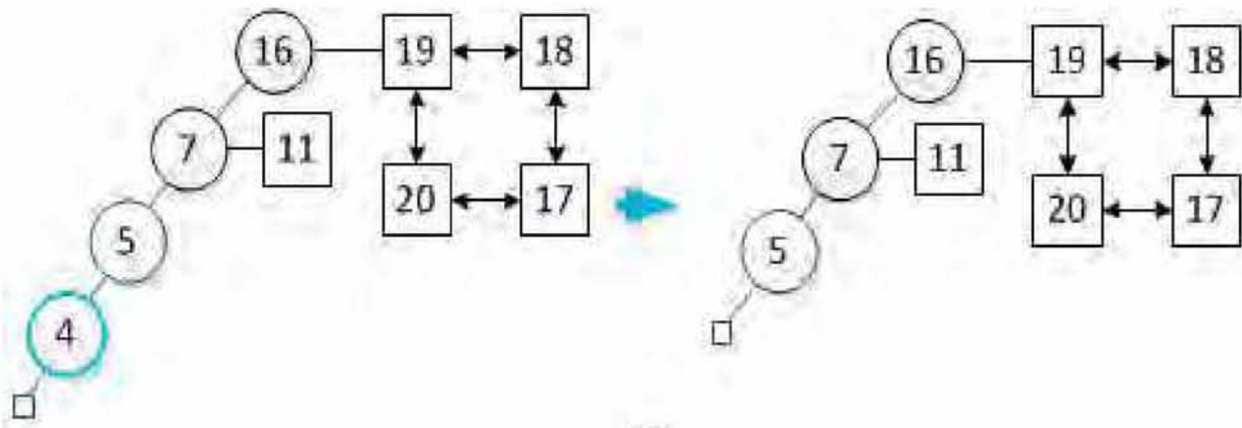


Σχήμα 4.12: (Συνέχεια) (δ)-(ιβ) 17, 5, 7, 16, 20, 18, 19, 4, 11

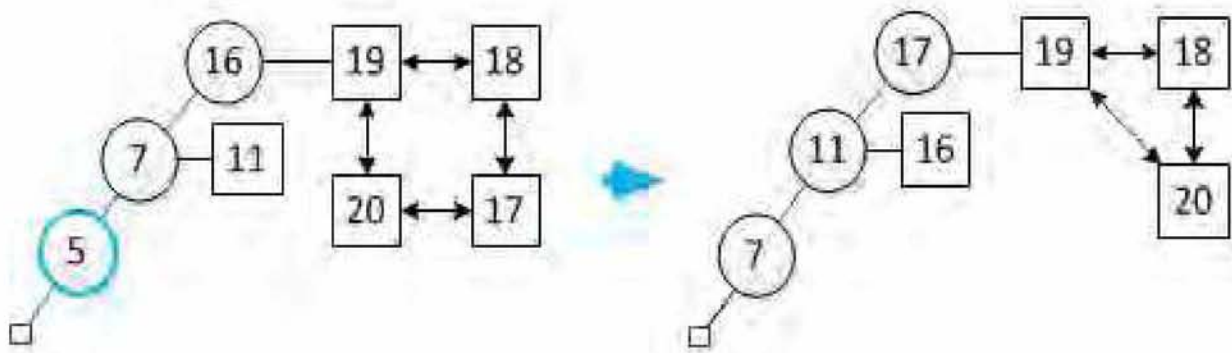
Απόσβεση Στοιχείου. Έστω ότι θέλουμε να διαγράψουμε στοιχείο x, με κλειδί key, τότε κάνουμε αναζήτηση του στοιχείου. Όταν βρεθεί τότε το στοιχείο χαρακτηρίζεται ως «ανενεργό», παράλληλα κρατάμε έναν μετρητή (inactive) ο οποίος αυξάνεται κάθε φορά που χαρακτηρίζεται ένα στοιχείο. Όταν ισχύσει $inactive \geq (2a-1)s(x)$ τότε ακολουθούμε την μέθοδο «εξισορρόπησης» που περιγράψαμε παραπάνω.



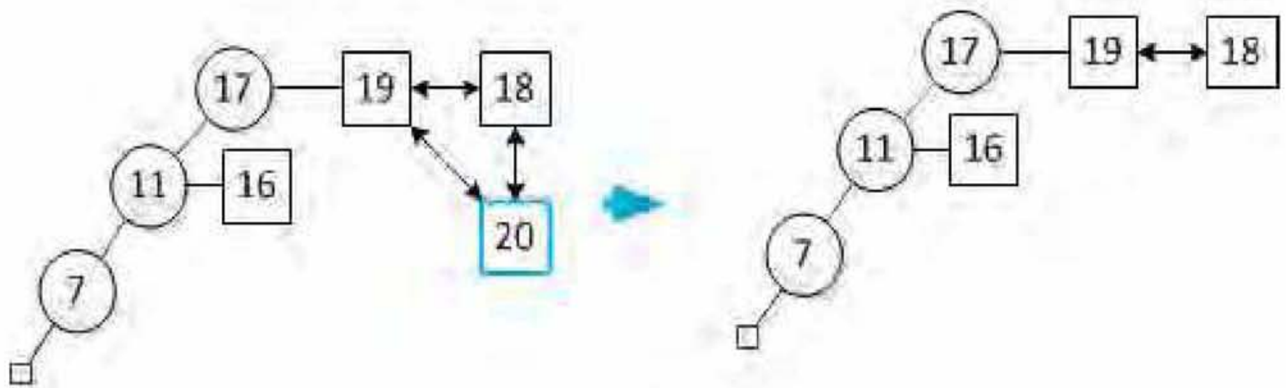
Σχήμα 4.13: (α)-(γ) Διαδοχικές αποσβέσεις των 6, 9, 14



(δ)



(ε)



(στ)

Σχήμα 4.14: (Συνέχεια) (δ)-(στ) 4, 5, 20

4.2.2 Ανάλυση Πολυπλοκότητας

Η πολυπλοκότητα της δομής που αναλύθηκε αποδεικνύεται ότι είναι λογαριθμική. Θα δώσουμε λίγη προσοχή στη «εξισορρόπηση» διότι η πρότυπη μέθοδος της «εξισορρόπησης» ενός ΔΔΑ σε γραμμικό χρόνο εκμεταλλεύεται το γεγονός ότι μπορούμε να αποκτήσουμε μια ταξινομημένη σειρά των στοιχείων της μέσω της inorder traversal. Αφού η δομή μας είναι «χαλαρή», δεν χρειαζόμαστε να γνωρίζουμε την συνολική ταξινομημένη σειρά των στοιχείων μας για να «εξισορροπήσουμε» τη δομή. Η ακόλουθη μέθοδος μετατρέπει ένα k-στοιχείων υποδέντρο $1/2$-ζυγισμένο σε χρόνο $O(k)$:

- Επιλέγουμε το μέσο στοιχείο m του υποδέντρου σε χρόνο $O(k)$.
- Χωρίζουμε τα $k-1$ εναπομείναντα στοιχεία σε δύο ομάδες που από τελούνται από $S_<$και $S_>$ έτσι ώστε $|S_<|$ και $|S_>|$ να διαφέρουν το πολύ κατά ένα.
- Κάνουμε το m ρίζα του «εξισορροπημένου» υποδέντρου, θέτοντας το δεξιό υποδέντρο να είναι μία διπλή κυκλική λίστα που θα περιέχει $S_>$.
- Αναδρομικά χτίζουμε ένα ½-γυρισμένο υποδέντρο από τα $S_<$ και το τοποθετούμε ως αριστερό υποδέντρο του m.

Λήμμα 4.2.1 Σε ένα α-ζυγισμένο δέντρο με n στοιχεία, το βάθος ενός στοιχείου βαθμού r είναι το πολύ $\log_{1/a}(n/r)$. [σελ. 62 -1]

Θεώρημα 4.2.2 Η εισαγωγή και η διαγραφή στην παραπάνω δομή απαιτεί $O(\log(n/r))$ αναπόσβεστο χρόνο. [σελ. 62 - 1]

4.2.3 Πειραματική Αξιολόγηση

4.2.3.1 Υπολογισμός Κόστους Στοιχειωδών Πράξεων

Για την πειραματική αξιολόγηση της εν λόγω δομής χρησιμοποιούμε παρόμοια μεθοδολογία με αυτήν του κεφαλαίου 4.1.3, εισάγουμε και πάλι τους μετρητές access, rebalance, insertion_cost, deletion_cost σε μέρη του κώδικα όπου έχουμε μετακίνηση για εύρεση, «εξισορρόπηση», συνολικό κόστος εισαγωγής και διαγραφής αντίστοιχα. Ενδεικτικά παρατίθενται κάποια κομμάτια κώδικα που υποδεικνύουν τι χρήση των μετρητών:

access:

```
void deactivate(int key){
	struct tree *point, *check;
	struct dlist *pointer;

	for(point=Troot;(point->key>key)&&(point!=NULL);point=point->left){
		access++;
		if(point->left == NULL) break;
	}
	if(point->key == key){ point->active = false; point->inactive++; }
	else{
		for(pointer=point->right;pointer->key!=key;pointer=pointer->next)
			{ access++; }
		pointer->active = false;
		point->inactive++;
	}
	check = NULL;
	{………}
	reassembly(Troot, 1);
```

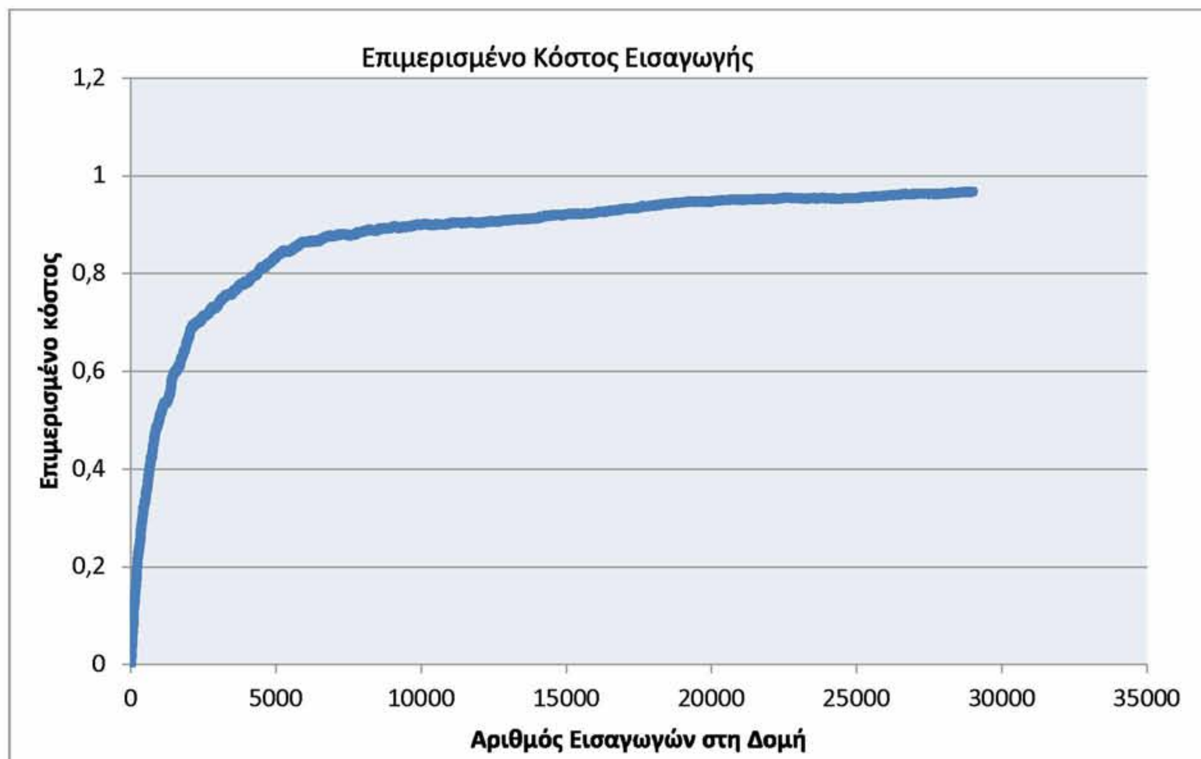
rebalance:

```
		//κομμάτι κώδικα από την εισαγωγή στοιχείου
if(msg == 0){
	new_point = NULL;
	while(point != Troot){
	if((point->lnodes>(int)(a*point->tnodes))||(point->rnodes>(int)(a*point->tnodes)))
		{ new_point = point; }
	point = point->parent;
	}
	if((point->lnodes>(int)(a*point->tnodes))||(point->rnodes>(int)(a*point->tnodes)))
		{ new_point = point; }
	if(new_point != NULL){ rebalance++ reassembly(new_point,0); }
}
```

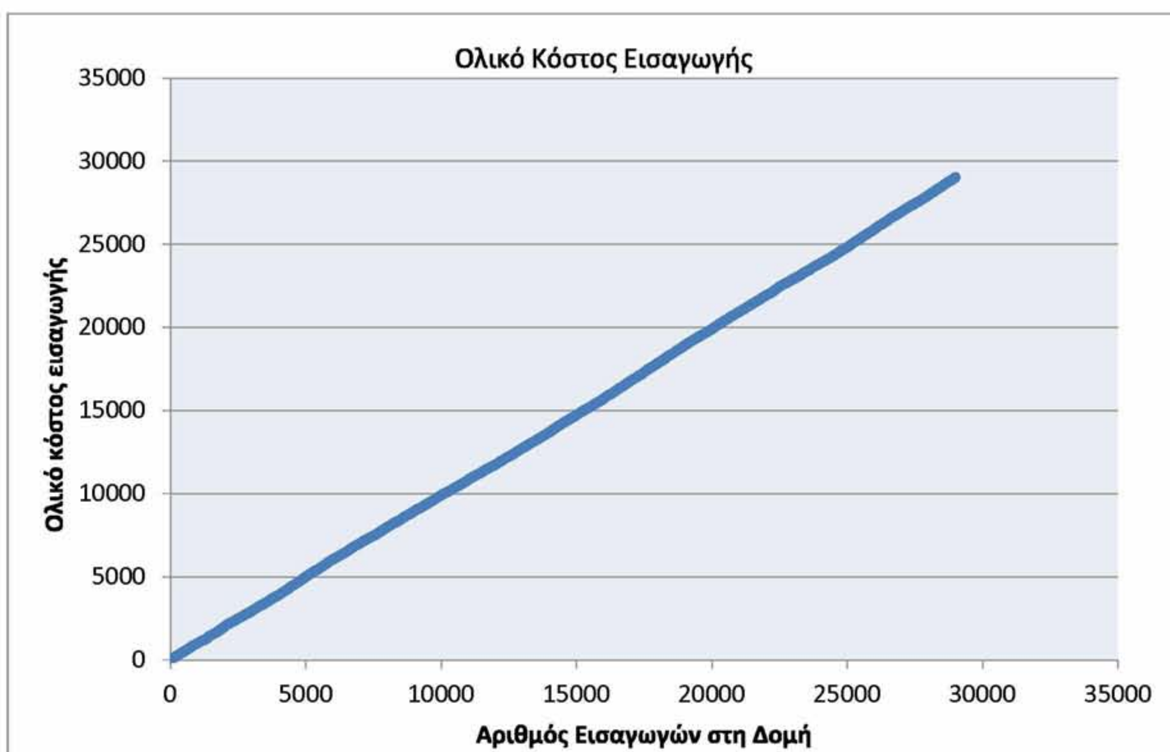
insertion_cost:

```
	//κομμάτι κώδικα από την main για την εισαγωγή
	std::cout<<"Insert Node Data: ";
	std::cin>>val;
	insert_tree(val, 0, NULL);
	insertion_cost += (access + rebalance);
	fprintf(w,"%.3f\n",(double)insertion_cost/count_ins);
	access = 0;
	rebalance = 0;
```

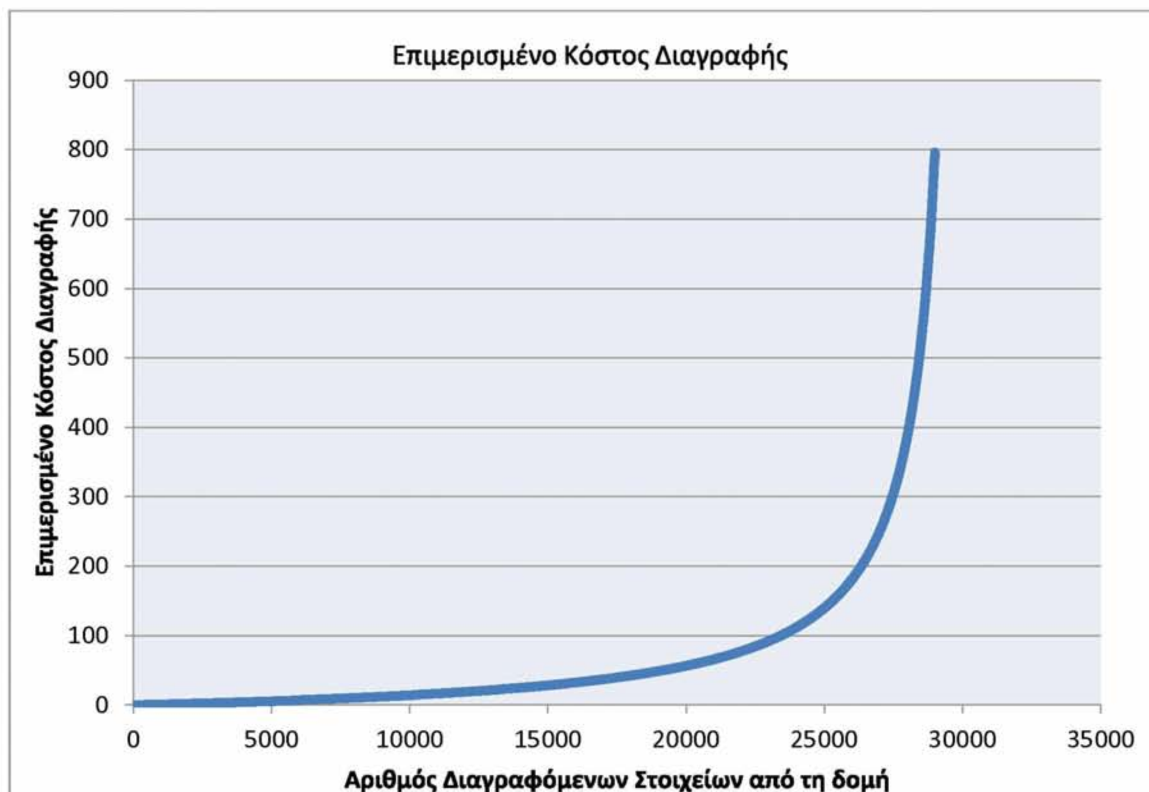
Με το πέρας της εισαγωγής 30000 και της διαγραφής 25000 στοιχείων σε/από μία δομή έχουμε τις παρακάτω γραφικές παραστάσεις:



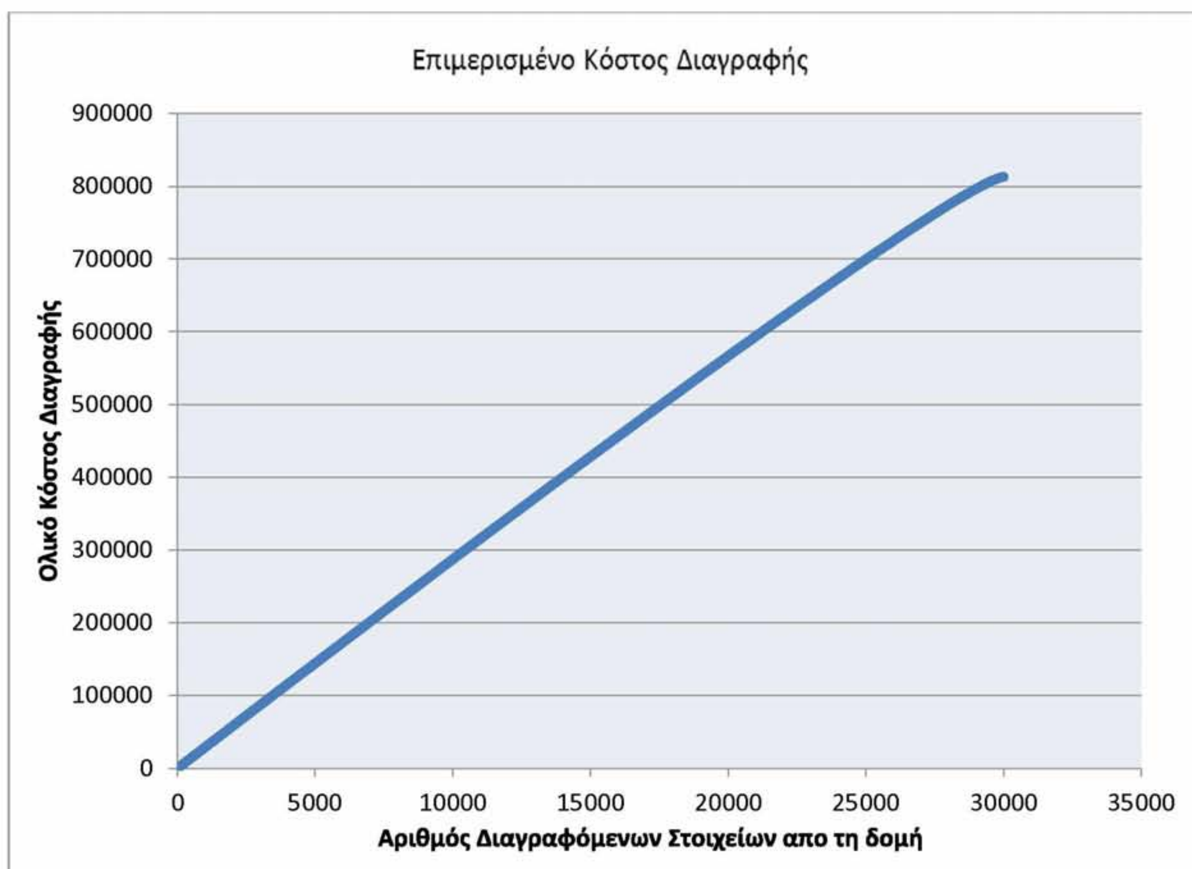
Σχήμα 4.15: Επιμερισμένο Κόστος Εισαγωγής



Σχήμα 4.16: Ολικό Κόστος Εισαγωγής



Σχήμα 4.17: Επιμερισμένο Κόστος Διαγραφής



Σχήμα 4.18: Ολικό Κόστος Διαγραφής

4.2.3.2 Μετρήσεις Πραγματικού Χρόνου

Όπως και στην προηγούμενη προσέγγιση χρησιμοποιώντας τη βιβλιοθήκη <time.h> υπολογίσαμε τους εκάστους πραγματικούς χρόνους μέχρι το πέρας της λειτουργίας της δομής για τις εισαγωγές και τις διαγραφές. Τα αποτελέσματα παρατίθενται στον παρακάτω πίνακα. Οι μετρήσεις έγιναν στο ίδιο σύστημα όπως προηγουμένως.

	MS Windows 7		Ubuntu 11.04	
	Εισαγωγή	Διαγραφή	Εισαγωγή	Διαγραφή
10.000 στοιχεία	0.876 sec	3.114 sec	0.564 sec	2.485 sec
20.000 στοιχεία	1.126 sec	3.754sec	0.856 sec	4.145 sec
30.000 στοιχεία	1.624 sec	4.244 sec	1.019 sec	4.545 sec
40.000 στοιχεία	2.023 sec	4.581 sec	1.672 sec	4.874 sec
50.000 στοιχεία	2.865 sec	4.697 sec	2.069 sec	5.944 sec

Πίνακας 4.2: Μετρήσεις Πραγματικού Χρόνου στη δομή h-treap

Σημείωση: οι διαγραφές πραγματοποιήθηκαν στην δομή αφού είχε εισαχθεί 50.000 στοιχεία

ΠΑΡΑΡΤΗΜΑ Α

Υλοποίηση Τυχαιοποιημένης προσέγγισης:

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<map>

struct Randomized{
	int heapkey;
	struct Randomized *right;
	struct Randomized *left;
	struct Randomized *parent;
};

std::map<int, struct Randomized *> hash;
std::vector<struct Randomized *> empty_space;
void Randomized_delete(int delobj);
struct Randomized *Rroot;

//απλή αριστερή περιστροφή προς άνω
void rotate_up_left(struct Randomized *pointer){
	pointer->left->parent = pointer->parent;
	pointer->parent->right = pointer->left;
	pointer->left = pointer->parent;
	if(Rroot == pointer) pointer->parent = Rroot;
	else pointer->parent = pointer->parent->parent;
	pointer->left->parent = pointer;
}

//απλή δεξιά περιστροφή προς άνω
void rotate_up_right(struct Randomized *pointer){
	pointer->right->parent = pointer->parent;
	pointer->parent->left = pointer->right;
```

```
    pointer->right = pointer->parent;
    if(Rroot == pointer) pointer->parent = Rroot;
    else pointer->parent = pointer->parent->parent;
    pointer->right->parent = pointer;
}

//δημιουργία παιδιών και προσθήκη τους στην empty_space
void create(struct Randomized *pointer, int heapdata){
    std::vector<struct Randomized >::iterator iter1;
    std::vector<struct Randomized >::iterator iter2;
    pointer->left = (struct Randomized *)malloc(sizeof(struct Randomized));
    pointer->right = (struct Randomized *)malloc(sizeof(struct Randomized));
    pointer->heapkey = heapdata; pointer->left->heapkey = 0;
    pointer->right->heapkey = 0; pointer->left->parent = pointer;
    pointer->right->parent = pointer;

    empty_space.push_back(pointer->left);
    empty_space.push_back(pointer->right);
}

//απλή αριστερή περιστροφή προς τα κάτω
void rotate_down_left(struct Randomized *pointer, struct Randomized *parentP){

    if(parentP->left == pointer){
        parentP->left = pointer->right; pointer->right = parentP->left->left;
        parentP->left->left = pointer; pointer->right->parent = pointer;

        parentP->left->parent=parentP; pointer->parent=pointer->parent->left;
    }
    else if(parentP->right == pointer) {
        parentP->right=pointer->right; pointer->right = parentP->right->left;
        parentP->right->left = pointer; pointer->right->parent = pointer;

        parentP->right->parent=parentP; pointer->parent=pointer->parent->right;
    }
    else if(pointer->parent == pointer){
```

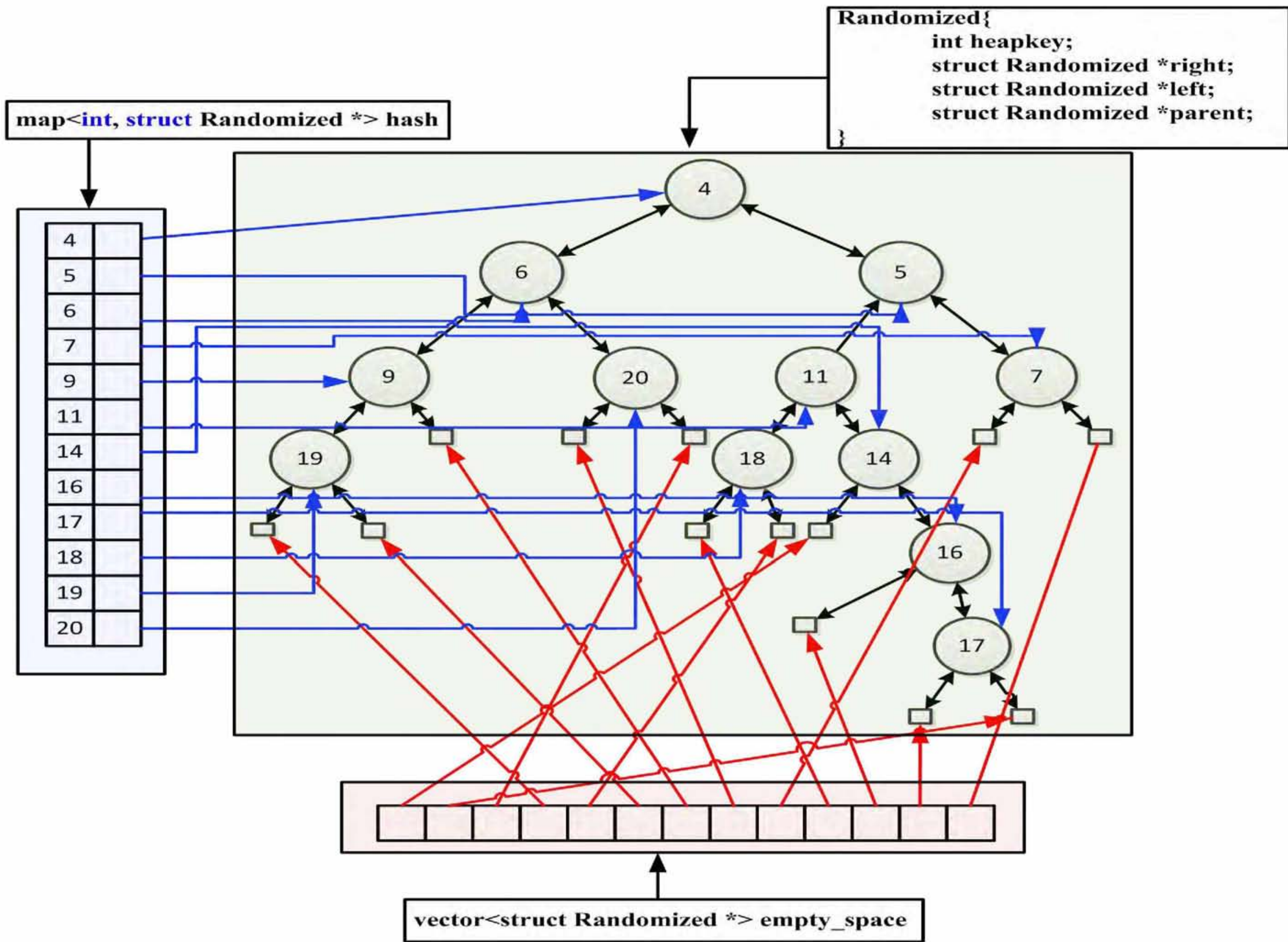
```
            Rroot = pointer->right; pointer->right = Rroot->left;
            Rroot->left = pointer; pointer->right->parent = pointer;
            Rroot->parent = Rroot; pointer->parent = Rroot;
      }
}

//απλή δεξιά περιστροφή προς τα κάτω
void rotate_down_right(struct Randomized *pointer, struct Randomized *parentP){
      if(parentP->left == pointer){
            parentP->left = pointer->left; pointer->left = parentP->left->right;
            parentP->left->right = pointer; pointer->left->parent = pointer;

            parentP->left->parent=parentP; pointer->parent=pointer->parent->left;
      }
      else if(parentP->right == pointer){
            parentP->right =pointer->left; pointer->left = parentP->right->right;
            parentP->right->right = pointer; pointer->left->parent = pointer;

            parentP->right->parent=parentP; pointer->parent=pointer->parent->right;
            }
      else if(pointer->parent == pointer){
            Rroot = pointer->left; pointer->left = Rroot->right;
            Rroot->right = pointer; pointer->left->parent = pointer;
            Rroot->parent = Rroot; pointer->parent = Rroot;
      }
}

//εισαγωγή στοιχείου στη δομή
int Randomized_insert(int heapdata){
      struct Randomized *pointer;
      static int null_length = 2;

      if(Rroot == NULL){
            pointer=(struct Randomized *)malloc(sizeof(struct Randomized));
            create(pointer,heapdata); pointer->parent=pointer; Rroot=pointer;
      }
```

```
    else{
        int random = rand()%null_length; pointer = empty_space[random];
        create(pointer, heapdata);
        empty_space.erase(empty_space.begin() + random);

        while(pointer->parent->heapkey > heapdata){
            if(pointer->parent == Rroot) Rroot = pointer;
            else if(pointer->parent->parent->left == pointer->parent)
                { pointer->parent->parent->left = pointer; }
            else if(pointer->parent->parent->right == pointer->parent)
                { pointer->parent->parent->right = pointer; }

            if(pointer == pointer->parent->left) rotate_up_right(pointer);
            else if(pointer == pointer->parent->right) rotate_up_left(pointer);
        }
        null_length ++;
    }
    hash.insert(std::pair<int, struct Randomized *>(heapdata, pointer));
    return(1);
}

//διαγραφή στοιχείου από τη δομή
void Randomized_delete(int delobj){
    struct Randomized *pointer, *parentP;
    std::map<int, struct Randomized *>::iterator it = hash.find(delobj);

    pointer = it->second; parentP= pointer->parent;

    while((pointer->left->heapkey) || (pointer->right->heapkey)){
        if((pointer->left->heapkey) && (pointer->right->heapkey)){
            if(pointer->left->heapkey > pointer->right->heapkey)
                { rotate_down_left(pointer, parentP); }
            else if(pointer->left->heapkey < pointer->right->heapkey)
                { rotate_down_right(pointer, parentP); }
        }
```

```
		else{
			if(pointer->right->heapkey) rotate_down_left(pointer, parentP);
			else if(pointer->left->heapkey)
				{ rotate_down_right(pointer, parentP); }
		}
		parentP = pointer->parent;
	}
	if((!pointer->left->heapkey)&&(!pointer->right->heapkey)&&(pointer->parent==pointer))
		{ Rroot = NULL; }

std::vector<struct Randomized *>::iterator it1 = std::find(empty_space.begin(),
empty_space.end(), pointer->left);
	empty_space.erase(it1);
	std::vector<struct Randomized *>::iterator it2 =
	std::find(empty_space.begin(),empty_space.end(), pointer->right);
	empty_space.erase(it2);


	if(parentP->left == pointer){
		parentP->left = NULL; free(parentP->left);
		parentP->left = (struct Randomized *)malloc(sizeof(struct Randomized));
		parentP->left->heapkey = 0; parentP->left->parent = parentP;
		empty_space.push_back(parentP->left);
	}
	else if(parentP->right == pointer){
		parentP->right = NULL; free(parentP->right);
		parentP->right = (struct Randomized *)malloc(sizeof(struct Randomized));
		parentP->right->heapkey = 0; parentP->right->parent = parentP;
		empty_space.push_back(parentP->right);
	}
	hash.erase(it);
}

//εκτύπωση δέντρου
void printtree(struct Randomized* a,int m){
	int n=m;
```

```
    if(a==0) return;
    if(a->right->heapkey) printtree(a->right,m+1);
    while(n--) printf("   ");
    printf("%d\n",a->heapkey);
    if(a->left->heapkey) printtree(a->left,m+1);
}

int main(int argc, char *argv[]){
    int val, count_ins = 0, selection = 1;
    Rroot = NULL;
    srand(time(NULL));
    while(selection != 0){
        std::cout<<"Select action:\nInsert: 1\tDelete: 2\tExit 0"<<std::endl;
        std::cout<<"Selection: ";
        std::cin>>selection;
        switch(selection) {
            case 1:
                std::cout<<"Number of nodes: ";
                std::cin>>nodes;
                for(int i=0; i<5; i++){
                    std::cout<<"Insert Node Data: ";
                    std::cin>>val;
                    Randomized_insert(val);
                }
                printtree(Rroot, 0);
            break;

            case 2:
                std::cout<<"Number of nodes: ";
                std::cin>>nodes;
                for(int i=2; i<30000; i++){
                    std::cout<<"Insert Node Data: ";
                    std::cin>>val;
                    Randomized_delete(val);
                }
            break;
```

```
                case 0:printf("Closing Program...\n");
                return(0);

                default: printf("select one from below:\n");
                break;
            }
        }
        return(0);
        system("PAUSE");
}
```



Σχήμα Α: Σχηματική Αναπαράσταση της δομής H-Treap

ΠΑΡΑΡΤΗΜΑ Β

Υλοποίηση Αναπόσβεστης προσέγγισης:

```
#include<iostream>
struct tree{
	int key;
	int tnodes;
	int lnodes;
	int rnodes;
	int inactive;
	bool active;
	struct tree *parent;
	struct tree *left;
	struct dlist *right;
};

struct dlist{
	int key;
	bool active;
	struct dlist *next;
	struct dlist *previous;
};

struct tree *Troot, *parentP, *min_ptr;
static int min_key = 3000000, max_key = 0;
double a = 0.5;
int *nums, total, *nodes_count;
bool min = true, mid = true;

void insert_tree(int, struct tree *);
void deleteTree(struct tree *, int);
void reassembly(struct tree *, int);
void deactivate(int);
void printlist(struct dlist *);
```

```c
//δημιουργία ελαχίστου κόμβου
tree *create_min(int key){
	struct tree *new_ptr;
	new_ptr = (struct tree *)malloc(sizeof(struct tree));
	new_ptr->key = key; new_ptr->left = NULL;
	new_ptr->right = NULL; new_ptr->parent = NULL;
	new_ptr->active = true; new_ptr->tnodes = 1;
	new_ptr->lnodes = 0; new_ptr->rnodes = 0;
	new_ptr->inactive = 0;

	return new_ptr;
}

//δημιουργία κόμβου για τη διπλή κυκλική λίστα
dlist *create_ptr(int key){
	struct dlist *new_ptr;
	new_ptr = (struct dlist *)malloc(sizeof(struct dlist));
	new_ptr->key = key; new_ptr->active = true;
	new_ptr->next = NULL; new_ptr->previous = NULL;

	return new_ptr;
}

//προσθήκη κόμβου στην κυκλική λίστα
void add_right(struct tree *t_ptr, struct dlist *l_ptr){

	if(t_ptr->right == NULL){ l_ptr->next = l_ptr; l_ptr->previous = l_ptr; }
	else{
		l_ptr->next = t_ptr->right; l_ptr->previous = t_ptr->right->previous;
		t_ptr->right->previous->next = l_ptr; t_ptr->right->previous = l_ptr;
	}
	t_ptr->right = l_ptr;
}
```

```
//υπολογισμός αριθμού κόμβων
void calc_nodes(){
	struct tree *temp;

	for(temp = min_ptr->parent; temp!=Troot; temp = temp->parent){
		temp->lnodes++; temp->tnodes++;
	}
	temp->lnodes++; temp->tnodes++;

}

//διαγραφή δέντρου και αποθήκευση στοιχείων σε πίνακα
void deleteTree(struct tree *point, int func){
	struct dlist *pointer, *pointer1;
	struct tree *tmp, *temp;
	int pos;

	if(point->parent != point) temp = point->parent;
	else temp = NULL;
	point= min_ptr;
	while(point && point!=temp){
		if(point->active) pos = total;
		else pos = total - 1;
		if(point->right){
			for(pointer=point->right->next; pointer!=point->right;
			pointer=pointer->next){
				pointer->next->previous = pointer->previous;
				pointer->previous->next = pointer->next;
				pointer1 = pointer; pointer = pointer->previous;
				if(pointer1->active){ pos++; nums[pos] = pointer1->key;}
				free(pointer1);
			}
			if(pointer->active){ pos++; nums[pos] = pointer->key;}
			free(pointer);
		}
		if(point->parent == point){ Troot = NULL; parentP = Troot; }
```

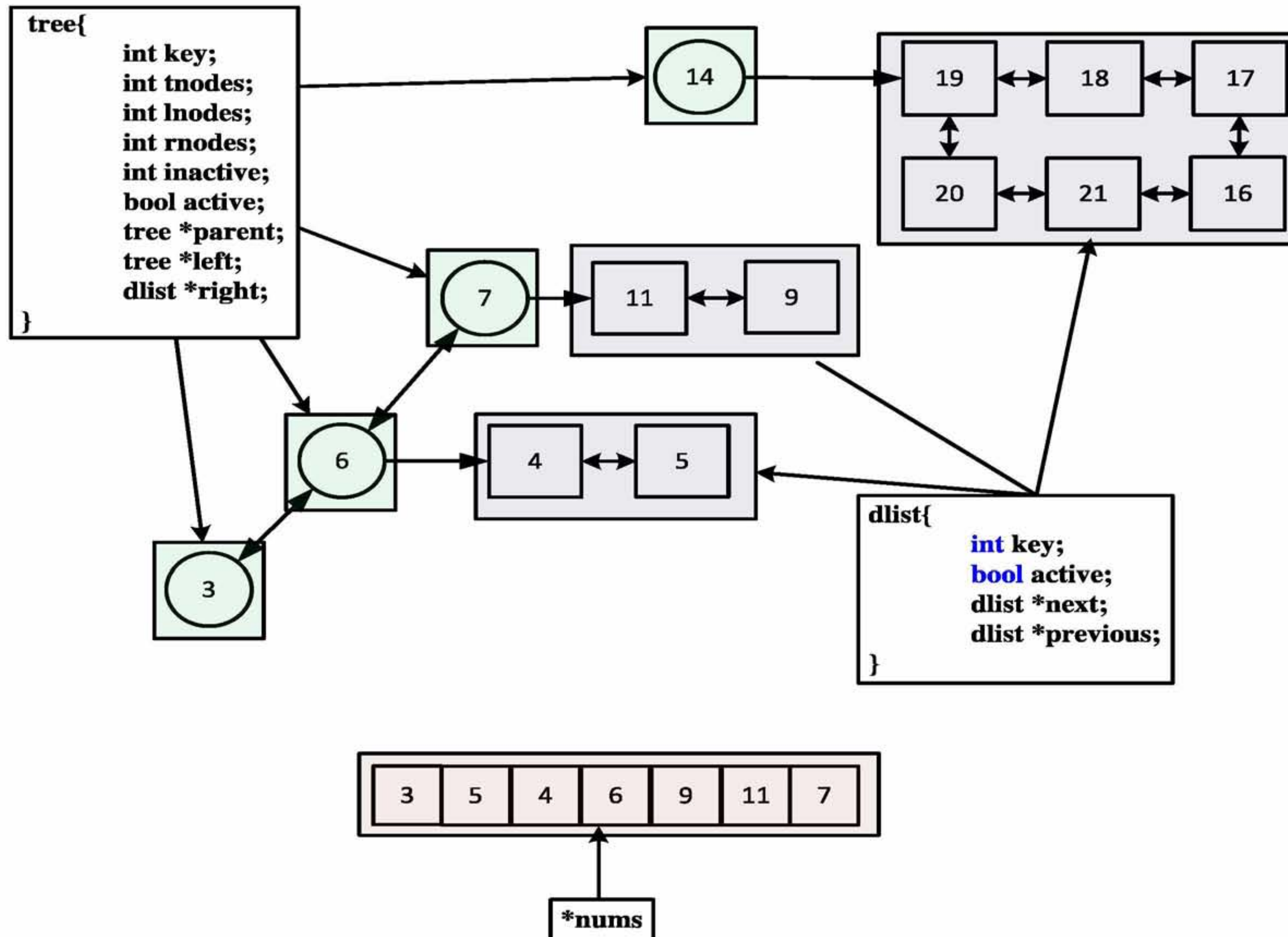
```
            if(point->active){ pos = total; nums[pos] = point->key;}
            if(!func) total = pos + point->rnodes + 1;
            else{
                  if(min && mid) total = pos + point->rnodes - point->inactive +1;
                  else if(!min && point->inactive == 1 && mid)
                        { total = pos + point->rnodes - point->inactive + 1; }
                  else if(!min && point->inactive != 1 && mid)
                        { total = pos + point->rnodes + 1; }
                  else if(!mid && point->inactive != 1 && min)
                        { total = pos + point->rnodes + 1; }
                  else if(!mid && point->inactive == 1 && min)
                        { total = pos + 1; }
            }
            point = point->parent;
            if(point->parent == point && !point->left){point= NULL; free(point); }
            else{ tmp= point->left; tmp = NULL; free(tmp); point->left = NULL;}
      }
}

//ταξινόμηση του πίνακα των στοιχείων και ένθεση τους στη δομή
void reassembly(struct tree *point, int func){
      struct tree *new_ptr, *temp_ptr; struct dlist *ptr;
      int count2, temp=0, count1 = 0, start, end, num, del = 0;

      if(point == Troot){ min_key=3000000; }
      else{ min_key = point->parent->key; }

      if(!func){ if((point->lnodes <= (int)(a*point->tnodes))&&(point->rnodes <=
      (int)(a*point->tnodes)))
            { point = point->left; } }

      if(!func){ count1 = point->tnodes - 1; }
      else count1 = point->tnodes - 2;

      total = 0; nums = (int *)malloc(count1*sizeof(int));
      temp_ptr = point;
```

```
    if(temp_ptr->parent == temp_ptr) temp_ptr = NULL;
    else { temp_ptr = temp_ptr->parent; }

    deleteTree(point,func);
    for(start=0;start<=count1-1;start++){
          end=start;
          num=nums[start];
          while( end>0 && nums[end-1]>num){ nums[end]=nums[end-1]; end=end-1; }
          nums[end]=num;
    }

    point = temp_ptr;
    do{
          count2 = count1;
          if(!(count1%2) ) count1 = (int)(count1/2);
          else if(count1%2 && count1 != 1 ) count1 = (int)(count1/2) + 1;
          else if(count1 == 1) count1--;
                new_ptr = create_min(nums[count1]);
                if(!temp_ptr){ Troot = new_ptr; new_ptr->parent = Troot;
                temp_ptr = Troot; temp_ptr->tnodes = count2+1;
                temp_ptr->lnodes=count1; }
          else { new_ptr->parent = temp_ptr; temp_ptr->left = new_ptr;
                      temp_ptr = new_ptr; temp_ptr->tnodes = count2+1;
                      temp_ptr->lnodes=count1; }
                       min_ptr = new_ptr; min_key = nums[count1];

          for(;count2>count1;count2--){ ptr = create_ptr(nums[count2]);
                add_right(new_ptr, ptr); temp_ptr->rnodes++; }
          temp = count1; count1 = count2-1;
          if((count1 == 1)&&(nums[temp] == nums[1])) count1--;
    }while(count1>=0);

    nums = NULL; free(nums);
    min = true; mid = true;
}
```

```
//εισαγωγή στη δομή
void insert_tree(int key, struct tree *p){
	struct tree *point, *temp, *new_min;
	struct dlist *new_ptr;

	if(key <= min_key) new_min = create_min(key);
	else new_ptr = create_ptr(key);

	if(!Troot){ Troot= new_min; new_min->parent = Troot;
		point = Troot; min_ptr = Troot; temp = point; }
	else if(min_ptr->key > key){ min_ptr->left = new_min;
		new_min->parent = min_ptr; min_ptr = new_min;
		temp = min_ptr; calc_nodes(); }
	else{
		for(point=Troot; (point->key > key)&&(point!=NULL); point = point->left){
			point->lnodes++; point->tnodes++; if(!point->left) break;
		}
		temp= point; point->rnodes++; point->tnodes++;
		add_right(point, new_ptr);
	}

	if(min_key>key) min_key = key;
	new_min = NULL;
	while(temp != Troot){
		if((temp->lnodes>(int)(a*temp->tnodes))||(temp->rnodes>(int)(a*temp-
		>tnodes))) new_min = temp;
		temp = temp->parent;
	}
	if((temp->lnodes>(int)(a*temp->tnodes))||(temp->rnodes>(int)(a*temp->tnodes)))
		{ new_min = temp; }
	if(new_min) reassembly(new_min,0);
}

//διαγραφή από τη δομή
void deactivate(int key){
	struct tree *point, *check;
```

```
    struct dlist *pointer;

    for(point = Troot; (point->key>key)&&(point!=NULL); point=point->left)
          { if(point->left == NULL) break; }
    if(point->key == key){ point->active = false; point->inactive++; }
    else{
          for(pointer = point->right; pointer->key!=key; pointer = pointer->next);
          pointer->active = false; point->inactive++;
    }
    check = NULL;
    if(point->key ==min_key && !point->active && point->right != NULL) min = false;
    else if(point->key==min_key && !point->active && point->right==NULL) mid=false;
    else if(point->key==key && !point->active && point->key !=min_key) mid = false;

    reassembly(Troot, 1);
}

int main(int argc, char *argv[]){
    int val, selection = 1, counts=0, nodes;
    Troot = NULL;
    while(selection != 0){
          std::cout<<"Select action:\tInsert: 1\tDelete: 2\tExit 0"<<std::endl;
          std::cout<<"Selection: ";
          std::cin>>selection;
          switch(selection) {
                case 1:
                      std::cout<<"Number of Nodes: ";
                      std::cin>>nodes;
                      for(int i=0; i<nodes; i++){
                            std::cout<<"Insert Node Data: ";
                            std::cin>>val;
                            insert_tree(val, NULL);
                      }
                break;
```

```
            case 2:
                  std::cout<<"Number of Nodes: ";
                  std::cin>>nodes;
                  for(int i=0; i<nodes; i++){
                        std::cout<<"Insert Node Data: ";
                        std::cin>>val;
                        deactivate(val);
                  }
            break;

            case 0: printf("Closing Program...\n");
            return(0);

            default: printf("select one from below:\n");
            break;
      }
   }
   return(0);
}
```



Σχήμα Α: Σχηματική Αναπαράσταση της Αναπόσβεστης Προσέγγισης τη στιγμή που αποκόπτεται η δομή στο ανώτερο σημείο που συναντήθηκε το πρόβλημα και έχουν περαστεί τα στοιχεία στον πίνακα nums, επόμενη ενέργεια είναι η ταξινόμηση του nums και επανεισαγωγή

Βιβλιογραφία

[1] B.C. Dean και Z. H. Jones, Rank-Sensitive Priority Queues, Clemson, 2009.

[2] Π.Δ. Μποζάνης, Δομές Δεδομένων, Βόλος, 2006.

[3] Π.Δ. Μποζάνης, Αλγόριθμοι, Βόλος, 2005

[4] R.K. Ahuja, K. Melhorn, J.B. Orlin and R.E. Tarzan, Faster algorithms for the shortest path problem, Journal of the ACM, 37:213–22, 1990.

[5] C.R. Aragon και R. Seidel, Randomized search trees, Algorithmica, 16:464-497, 1996

[6] J. Iacono and S. Langerman. Queaps. Algorithmica, 42(1):49–56, 2005.