Μελέτη διεξαγωγής αποσβέσεων δίχως την χρήση επαναζυγιστικών πράξεων

Καλαμπούκας Αθανάσιος

Επιβλέποντες καθηγητές:

Μποζάνης Παναγιώτης
Κατσαρός Δημήτριος

Βόλος 2011

Ευχαριστίες

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

Στη συνέχεια, θα ήθελα να ευχαριστήσω τους γονείς μου, Κωνσταντίνο και Παναγιώτα, καθώς και την αδερφή μου, Ειρήνη 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 επιτύχω τους στόχους μου. Τους οφείλω πολλά και τους υπερευχαριστώ.

Τέλος, ένα πολύ μεγάλο ευχαριστώ στην γιαγιά και παππού μου, Βασιλική και Αθανάσιο, για τη συμπαράστασή τους, όλα αυτά τα χρόνια των σπουδών μου. Αυτήν την διπλωματική εργασία την αφιερώνω στην μνήμη της πολυαγαπημένης γιαγιάς μου.

Πίνακας Περιεχομένων

Περίληψη……………………………………………….…………………………….........4
1.Εισαγωγή………..…………………...…………………………………………………....5
1.1. Βασικές Έννοιες Αλγορίθμων.................……………………………….................5
1.2. Αντικείμενο Δομών Δεδομένων…………………….………………………….......5
1.3. Ανάλυση Αλγορίθμων…………………………………………………………....5
1.4. Ανάλυση Χειρότερης Περιπτώσεως…..…………………………………….........6
1.5. Ανάλυση Μέσης Περιπτώσεως.……………………………………………….......6
1.6. Ανάλυση Επιμερισμένης ή Κατανεμημένης Περιπτώσεως…………………...…6
1.6.1. Μέθοδος Λογαριασμού Τραπεζίτη…….……..…………......................7
1.6.2. Μέθοδος Δυναμικού…………………………………………...........7
2. Βασικές Δομές Δεδομένων………………..………………………………………...9
2.1 Στατικά Δένδρα..…………………………………………………………….9
2.2 Δυναμικά Δένδρα……………..………………………………………….......9
3. Δένδρα Αναζητήσεως………….……………………………………………..............10
3.1 Πρόβλημα Λεξικού……….…………………………………………………...10
3.2 Αζύγιστα Δυαδικά Δένδρα Αναζητήσεως.…………………………………….10
3.2.1 Περιγραφή πράξεων………………..……………………….……...11
3.2.1 Ανάλυση Πολυπλοκότητας……………………………………….....25
4. Δένδρα AVL……………………………………………………………………28
4.1 Ορισμός………………...…………………………………………………...28
4.2 Περιγραφεί των Βασικών Πράξεων…………………………………………..28
4.3 Ιδιότητες – Ανάλυση Πολυπλοκότητας……………………………..……...39
4.4 Ανάλυση Κώδικα AVL.……………………………………………………..40
5. Διαγραφή δίχως επαναζύγιση σε δυαδικά δένδρα αναζήτηση……………………………44
5.1 Εισαγωγή……………………………...………….........................44
5.2 Ορολογία δένδρου………………………………………………….............45
5.3 Χαλαρά(relaxed) Δένδρα AVL…….………………………………………….45
5.4 Από κάτω προς τα πάνω επαναζύγιση…………………………………..............49
5.5 Από πάνω προς τα κάτω επαναζύγιση…………………………………………...51
5.6 Επαναζύγιση ενός δέντρου……………………………………………………52

5.7 Καλές και κακές εναλλακτικές λύσεις…………………………………………...52
5.8 Επαναζύγιση μετά από διαγραφή ή όχι;…………………………………………53
5.9 Ανάλυση ΚώδικαRAVL…….…………………………………………………...53
6.Πειραματικά αποτελέσματα……….....…………………….……………………………55
6.1Συγκριτικά αποτελέσματα δένδρων AVL,RAVL…………….……………………..55
6.2 Παρατηρήσεις……..…………………………………………………….……59
7. Βιβλιογραφία…………………………………………………………………….....60
8.Παράρτημα…………………….…………………………………………………………61

Περίληψη

Στην παρούσα διπλωματική εργασία αντιμετωπίζουμε το ζήτημα των διαγραφών σε ισοζυγισμένα δέντρα. Η επαναζύγιση ***(rebalancing)*** μετά από μια διαγραφή είναι γενικά πιο περίπλοκη από ότι η επαναζύγιση ***(rebalancing)*** μετά από μια εισαγωγή. Περιγράφουμε μια χαλάρωση των δέντρων ***AVL*** στα οποία η επαναζύγιση ***(rebalancing)*** γίνεται μετά τις εισαγωγές και όχι μετά τις διαγραφές, με αποτέλεσμα ο χρόνος πρόσβασης να παραμένει λογαριθμικός στον αριθμό των εισαγωγών. Για πολλές εφαρμογές των ισοζυγισμένων δέντρων, η δομή μας προσφέρει ανταγωνιστικές επιδόσεις με αυτές των κλασσικών ισοζυγισμένων δέντρων. Η δομή μας χρειάζεται *O*(*log log* m) *bits of balance information* ανά κόμβο, όπου το *m* είναι ο αριθμός των εισαγωγών, ή *O*(*log log n*) με περιοδική ισοσκέλιση, όπου το *n* είναι ο αριθμός των κόμβων. Μια εισαγωγή παίρνει μέχρι δύο περιστροφές και *O(1)* υπολογιστικό χρόνο. Χρησιμοποιώντας μια ανάλυση που στηρίζεται σε μια εκθετική δυναμική συνάρτηση, δείχνουμε ότι λαμβάνουν χώρα τα βήματα για την επαναζύγιση ***(rebalancing)*** με μια συχνότητα που είναι εκθετικά μικρή στο ύψος του επηρεαζόμενου κόμβου.

1. Εισαγωγή

1.1 Βασικές Έννοιες Αλγορίθμων

Με τον όρο ***αλγόριθμος*** χαρακτηρίζεται μια πεπερασμένη σειρά ενεργειών, αυστηρά καθορισμένων και εκτελέσιμων σε πεπερασμένο χρόνο, που στοχεύουν στην επίλυση ενός προβλήματος.

Κάθε αλγόριθμος ικανοποιεί τα παρακάτω βήματα:

Είσοδος (input): Καμία, μία ή περισσότερες τιμές δεδομένων πρέπει να δίνονται ως είσοδοι στον αλγόριθμο. Η περίπτωση που δεν δίνονται τιμές δεδομένων εμφανίζεται, όταν ο αλγόριθμος δημιουργεί και επεξεργάζεται κάποιες πρωτογενείς τιμές με τη βοήθεια συναρτήσεων παραγωγής τυχαίων αριθμών ή με την βοήθεια άλλων απλών εντολών.

Έξοδος (output): Ο αλγόριθμος πρέπει να δημιουργεί τουλάχιστον μία τιμή δεδομένων ως αποτέλεσμα προς το χρήστη ή προς έναν άλλο αλγόριθμο.

Αποτελεσματικότητα(effectiveness): Κάθε μεμονωμένη εντολή του αλγορίθμου να είναι απλή. Αυτό σημαίνει ότι μία εντολή δεν αρκεί να έχει ορισθεί, αλλά πρέπει να είναι και εκτελέσιμη.

1.2 Αντικείμενο Δομών Δεδομένων

Αντικείμενο των ***Δομών Δεδομένων*** είναι η αναπαράσταση και η διαχείριση συνόλων αντικειμένων, τα οποία επιδέχονται πράξεις εξαγωγής πληροφορίας ή αλλαγής της συνθέσεως τους. Αυστηρότερα, μπορεί να ορίσει κανείς πως ασχολούνται με την επισταμένη μελέτη των υλοποιήσεων των συχνότερα εμφανιζομένων Αφηρημένων Τύπων Δεδομένων (**ΑΤΔ**). Ως αφηρημένος τύπος δεδομένων ορίζεται ένα σύνολο, με μία συλλογή πράξεων επί των στοιχείων του συνόλου.

1.3 Ανάλυση Αλγορίθμων

Ανάλυση αλγορίθμου αποκαλείται η εύρεση των ***πόρων (resources)*** που αυτός απαιτεί για να τρέξει. Με άλλα λόγια, ο ***χρόνος (time)*** περάτωσής του και ο αναγκαίος για τους υπολογισμούς ***χώρος (space)***, μετρούμενος σε ***αποθηκευτικές θέσεις (memory locations)***.

Οι δύο αυτοί δείκτες μετρήσεως της αποτελεσματικότητας του εκάστοτε αλγορίθμου συνιστούν την ***πολυπλοκότητα χρόνου*** και ***χώρου*** του *(**time and space complexity**)*.

1.4 Ανάλυση Χειρότερης Περιπτώσεως

Με τον όρο ***ανάλυση χειρότερης περιπτώσεως (worst case analysis)*** ονομάζουμε την μέγιστη τιμή που μπορεί να πάρει ο χρόνος τρεξίματος ή ο χώρος ενός αλγορίθμου για οποιαδήποτε είσοδο με συγκεκριμένο μέγεθος n.

Επομένως, η ανάλυση χειρότερης περίπτωσης βρίσκει το άνω όριο στην συμπεριφορά του αλγορίθμου, όταν του δοθεί μία νόμιμη είσοδος μεγέθους n.

1.5 Ανάλυση Μέσης Περιπτώσεως

Σε περίπτωση που είναι γνωστό το πιθανοτικό μοντέλο που διέπει τα δεδομένα, 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 μία νόμιμη είσοδος με συγκεκριμένο μέγεθος *n*.

1.6 Ανάλυση Επιμερισμένης ή Κατανεμημένης Περιπτώσεως

Υπάρχουν περιπτώσεις, στις οποίες μας ενδιαφέρει ο συνολικός χρόνος ακολουθίας πράξεων να είναι φραγμένος. Αυτό συνεπάγεται μεγαλύτερη ευελιξία στο θέμα σχεδιασμού της δομής επιτρέπεται, πλέον, ο χρόνος μίας πράξεως να μεταβάλλεται, αρκεί *ακριβές* πράξεις να ακολουθούνται από πολλές *φθηνές.*

Αυτός ο τρόπος αναλύσεως της επιδόσεως μίας δομής, ως μέσος όρος επιδόσεως επί ακολουθίας πράξεων, είναι γνωστός ως ***ανάλυση κατανεμημένης ή επιμερισμένης περιπτώσεως (amortized-case analysis)***.

Έστω *T(n)* ο μέγιστος χρόνος εκτελέσεως μίας οποιασδήποτε ακολουθίας *n* πράξεων επί μίας δομής. Για μία πράξη, ως επιμερισμένος ή κατανεμημένος χρόνος ορίζεται το πηλίκο *T(n)/n*. Αυτό σημαίνει πως, εάν η επιμερισμένη επίδοση μίας δομής είναι *f(n),* τότε μία οποιαδήποτε ακολουθία *n* πράξεων κοστίζει το πολύ *nf(n)*. Παρακάτω θα εξετάσουμε δύο ισοδύναμες τεχνικές επιμερισμένης αναλύσεως την

μέθοδο λογαριασμού τραπεζίτη (banker account method) και την ***μέθοδο συναρτήσεως δυναμικού (potential function method)***.

1.6.1 Μέθοδος Λογαριασμού Τραπεζίτη

Σύμφωνα με την ***μέθοδο λογαριασμού τραπεζίτη (banker account method)***, κάθε πράξη χρεώνεται ένα ***κατανεμημένο ή επιμερισμένο κόστος (amortized cost)***, το οποίο, ενδέχεται να είναι μικρότερο ή και μεγαλύτερο από το αντίστοιχο πραγματικό.

Κατάλληλα πρέπει να γίνει η επιλογή του κατανεμημένου κόστους έτσι ώστε: (α) να προσεγγίζεται το μέσο κόστος της πράξεως σε μία οποιαδήποτε ακολουθία πράξεων, και (β) το επιμέρους κατανεμημένο κόστος όλων των πράξεων, αθροιζόμενο, να φράσσει από πάνω το πραγματικό παρατηρούμενο χειρότερο κόστος της ακολουθίας.

Τις περισσότερες φορές, η διαφορά μεταξύ πραγματικού και κατανεμημένου κόστους χαρακτηρίζεται ως ***πίστωση (credit)*** και δηλώνει είτε το πλεόνασμα, που κατατίθεται προς μελλοντική χρήση, κατά την εξυπηρέτηση των επόμενων πράξεων, είτε το δάνειο, που λαμβάνεται από τα αποθεματικά, για την κάλυψη των τρεχούμενων αναγκών μίας πράξεως.

1.6.2 Μέθοδος Δυναμικού

Η ***μέθοδος δυναμικού*** *(potential method)* στηρίζεται στην ιδέα της απεικονίσεως της καταστάσεως μίας δομής ή ενός αλγορίθμου A μέσω μίας συναρτήσεως δυναμικού:

$$\mathbf{\Phi} : A \rightarrow R.$$

Αρχικά, αποδίδεται μία αρχική τιμή $\Phi(A_o)$. 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:

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

Το κατανεμημένο κόστος c_i' της o_i ορίζεται ως:

$$c_i' = c_i + \Delta\Phi i.$$

Αυτό έχει σαν αποτέλεσμα, το πραγματικό κόστος συν την μεταβολή που επήλθε στο δυναμικό εξ αιτίας της o_i.

Από την παραπάνω συζήτηση καθίσταται φανερό πως, κεντρικό ρόλο στην ανάλυση δυναμικού, παίζει η εκλογή της κατάλληλης συναρτήσεως δυναμικού **Φ.** Χάρις στην τελευταία, κάποιες πράξεις χρεώνονται περισσότερο (όταν $\mathbf{\Delta\Phi}_i > 0$) και κάποιες λιγότερο (όταν $\mathbf{\Delta\Phi}_i < 0$), συνολικά, όμως, επιτυγχάνεται ορθή ερμηνεία της πολυπλοκότητας της *A*.

2. Βασικές Δομές Δεδομένων

2.1 Δένδρα

Πρόκειται για τα γνωστά, από την Θεωρία Γραφημάτων, δένδρα με ρίζα. Από προγραμματιστικής, δε, απόψεως χαρακτηρίζονται από τις ακόλουθες πράξεις:

element(v) Επιστρέφει το στοιχείο που είναι αποθηκευμένο στον κόμβο v

father(v) Επιστρέφει δείκτη προς τον πατέρα του v

children(v) Επιστρέφει όλους τους δείκτες προς τα παιδιά του v

Παρατηρήστε πως οι παραπάνω πράξεις επιτρέπουν την επισκόπηση του δένδρου και καμία αλλαγή. Μερική «ευελιξία» παρέχουν οι ακόλουθες πράξεις:

setElement(v, e) Θέτει το e ως στοιχείο του v

setSon(v, p, i) Θέτει τον κόμβο p ως τον i-στο γιο του v

setFather(v, p) Θέτει τον κόμβο p ως τον πατέρα του v

οι οποίες αλλάζουν την μορφή του εκάστοτε δένδρου, χωρίς να αφαιρούν ή να προσθέτουν κόμβους. Οι όροι Πατέρας-Γονιός, και Παιδί-Γιος είναι ισοδύναμοι και χρησιμοποιούνται εναλλακτικά.

2.2 Δυναμικά Δένδρα

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

addLeaf (v, kindofson) Προσθέτει έναν νέο κόμβο ως *kindofson* (δεξί ή αριστερό) παιδί του *v*, εφ' όσον δεν διαθέτει τέτοιο

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

3. Δένδρα Αναζητήσεως

3.1 Πρόβλημα Λεξικού

Έστω ένα σύνολο *S = { (x,y) | x € U,y }*, όπου *U* το σύνολο σύμπαν, ήτοι έναν ολικώς διατεταγμένο σύνολο αντικειμένων – *κλειδιών*, και *y* η συσχετιζόμενη με το *x* πληροφορία.

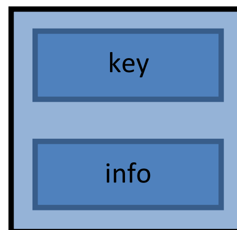
Μία δομή επί ενός συνόλου διατεταγμένων δυάδων καλείται ***δομή λεξικού (dictionary data structure)***, όταν ικανοποιεί τον ακόλουθο ΑΤΔ:

insertItem(key, info) Ένθεση ενός νέου στοιχείου που φέρει πληροφορία *info* και χαρακτηρίζεται από ένα κλειδί *key*

deleteItem(key) Απόσβεση, εάν υπάρχει, του στοιχείου με κλειδί *key*

findInfo(key) Εύρεση, εάν υπάρχει, του στοιχείου με κλειδί το *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:



Σχήμα 3.1: Γραφική αναπαράσταση στοιχείου λεξικού

3.2 Αζύγιστα Δυαδικά Δένδρα Αναζητήσεως

Ένα ***δέντρο δυαδικής αναζήτησης*** είναι ένα δυαδικό δέντρο όπου κάθε κόμβος είναι συσχετισμένος με ένα κλειδί και με την πρόσθετη ιδιότητα ότι το κλειδί οποιουδήποτε κόμβου είναι μεγαλύτερο (ή ίσο) από τα κλειδιά όλων των κόμβων του αριστερού υποδέντρου αυτού του κόμβου, και μικρότερο (ή ίσο) από τα κλειδιά όλων των κόμβων του δεξιού υποδέντρου αυτού του κόμβου.

3.2.1 Περιγραφή πράξεων

Βάσει τα παραπάνω, οι βασικές πράξεις σε ψευδογλώσσα, έχουν ως εξής:

Αναζήτηση Στοιχείου: Εφαρμόζουμε τον κάτωθι Αλγόριθμο:

```
Algorithm FINDNODE (BTNode u , Object key)
Input: Ένας κόμβος δένδρου u και ένα κλειδί key
Output: Ο κόμβος του υποδένδρου 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 { // πηγαίνουμε 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
```

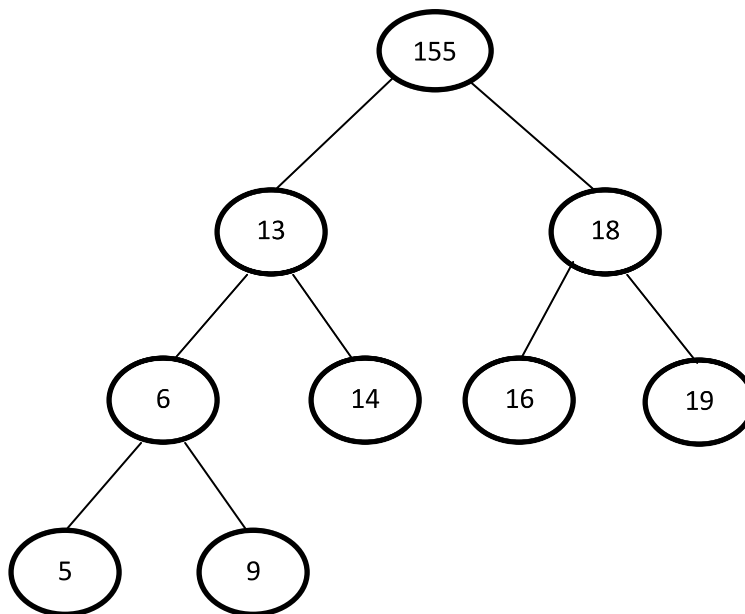
Η κλήση του παραπάνω αλγορίθμου γίνεται από τον κόμβο της ρίζας. Εκμεταλλευόμενος της σχετικής τοποθετήσεως των στοιχείων, συγκρίνει το κλειδί του τρέχοντος κόμβου με το *key*. Εάν είναι ίσα, τότε ο κόμβος έχει εντοπιστεί. Διαφορετικά, κινείται είτε αριστερά (το *key* είναι μικρότερο) είτε δεξιά (το *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:

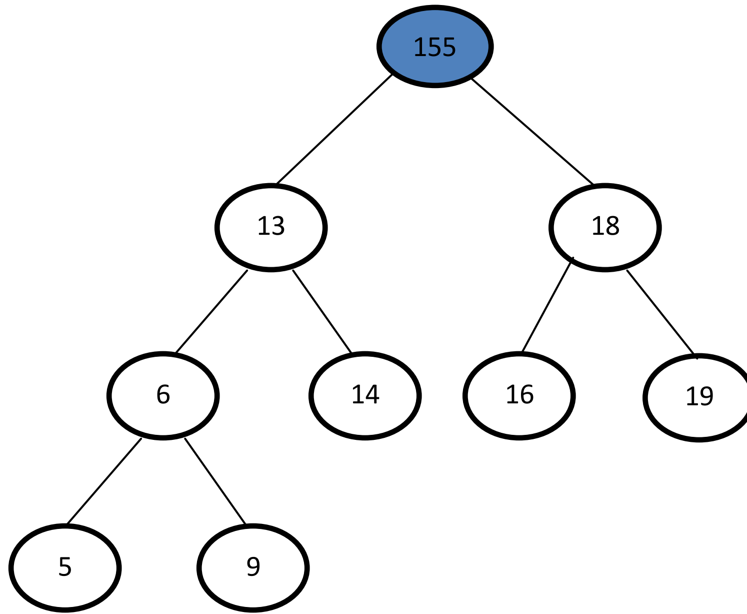
```
Algorithm FINDINFO(Object key)
Input: Ένα κλειδί key
Output:  Η πληροφορία που σχετίζεται με το κλειδί, εάν υπάρχει.
         Διαφορετικά null.
1. insNode = FindNode (key, ρίζα του δένδρου);
2. if (το κλειδί του insNode == key)
3.        return insNode.getElement ();
4. else
5.        return null;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

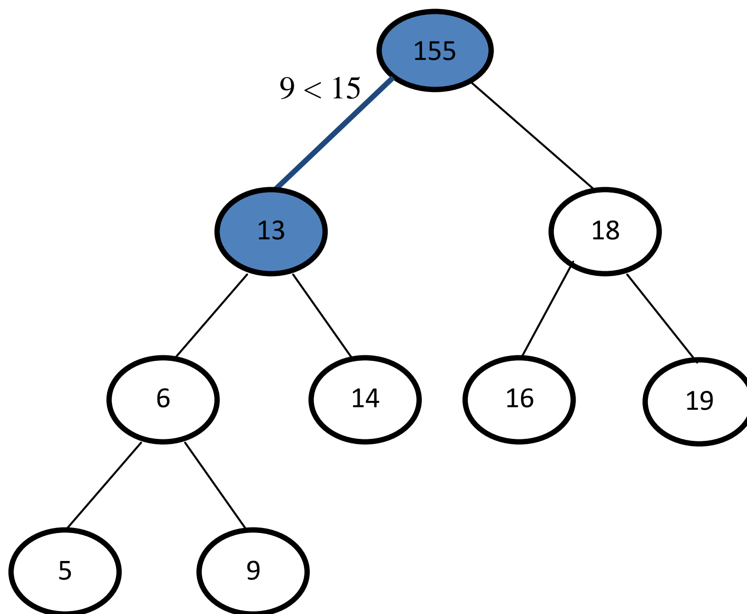
Στο σχήμα 3.2 εικονίζονται τα ακολουθούμενα ***μονοπάτια αναζητήσεως,*** για δύο περιπτώσεις αναζητήσεως: μίας επιτυχημένης, για το κλειδί 9, και μίας αποτυχημένης, για το κλειδί 17.

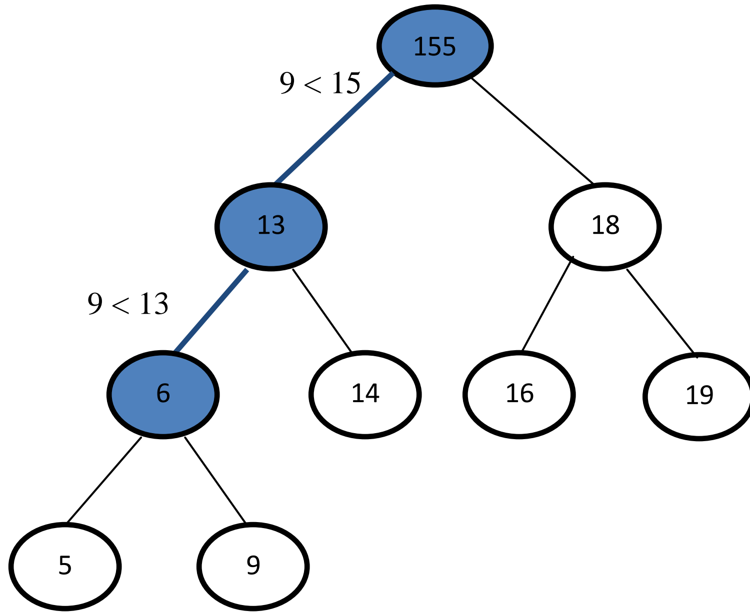


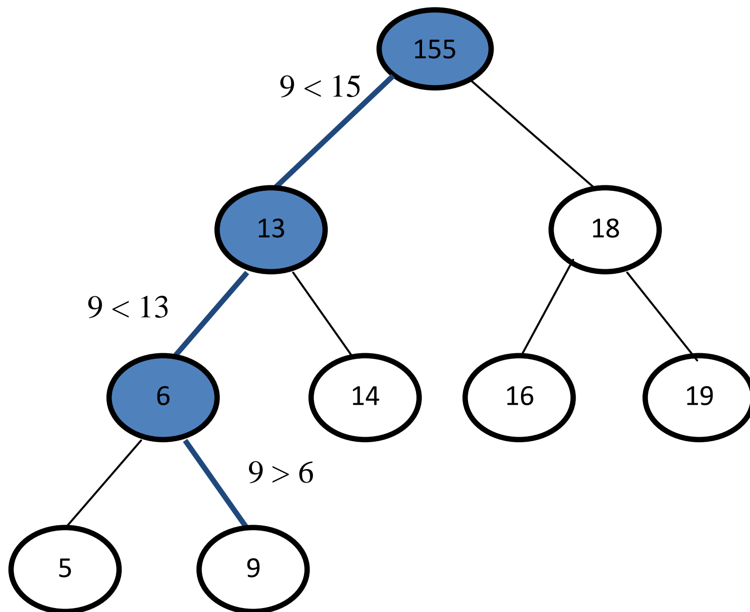
Σχήμα 3.2: Στιγμιότυπο: (α) επιτυχημένης αναζητήσεως του 9, (β) αποτυχημένης αναζητήσεως του 17

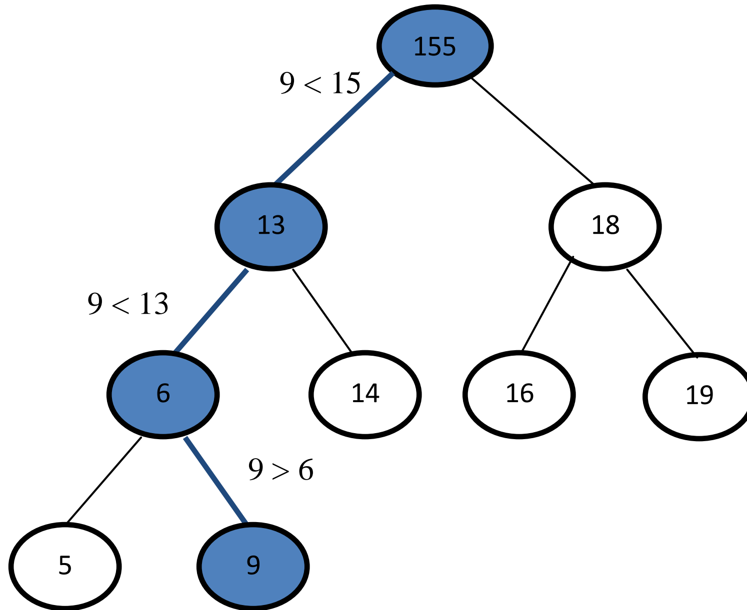
(α) Επιτυχημένη αναζήτηση του 9



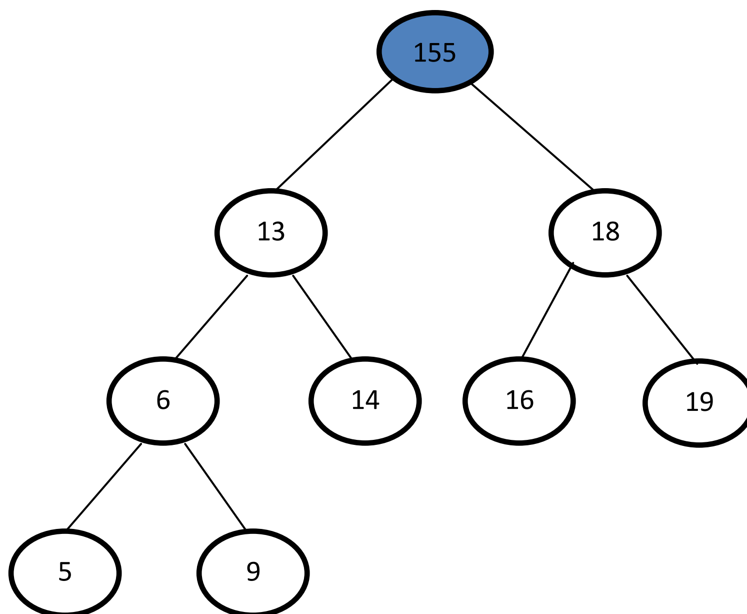


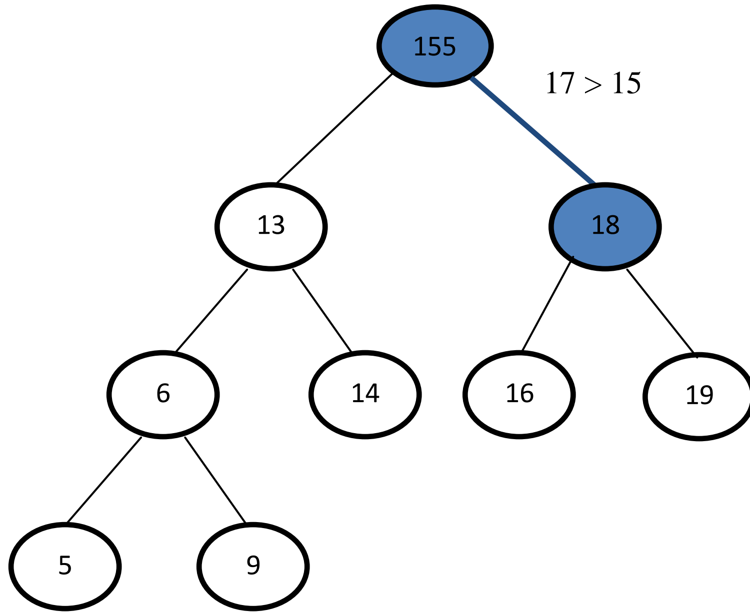

155
9 < 15
13
18
9 < 13
6
14
16
19
5
9

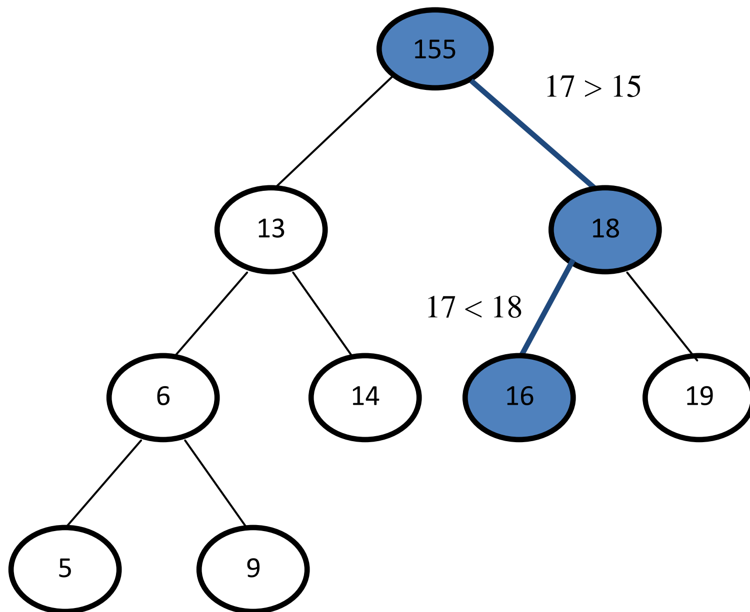

155
9 < 15
13
18
9 < 13
6
14
16
19
9 > 6
5
9

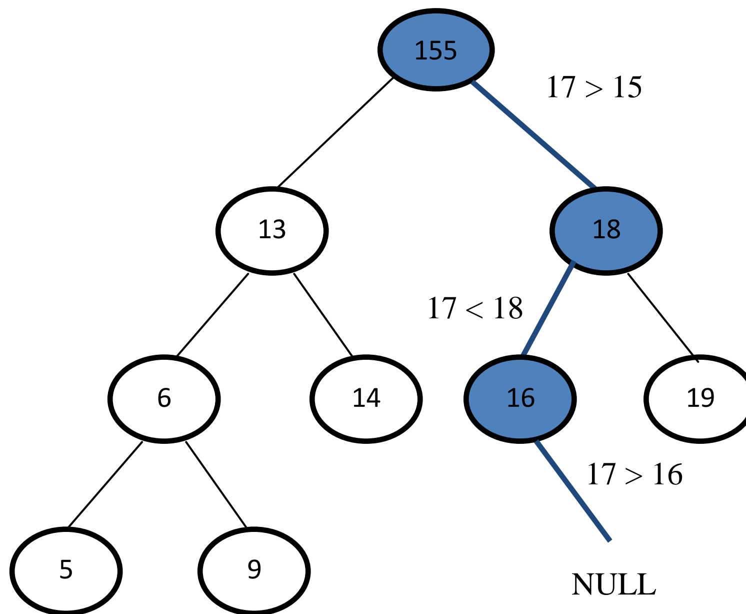


(β) Αποτυχημένη αναζήτηση του 17




155
17 > 15
13
18
6
14
16
19
5
9

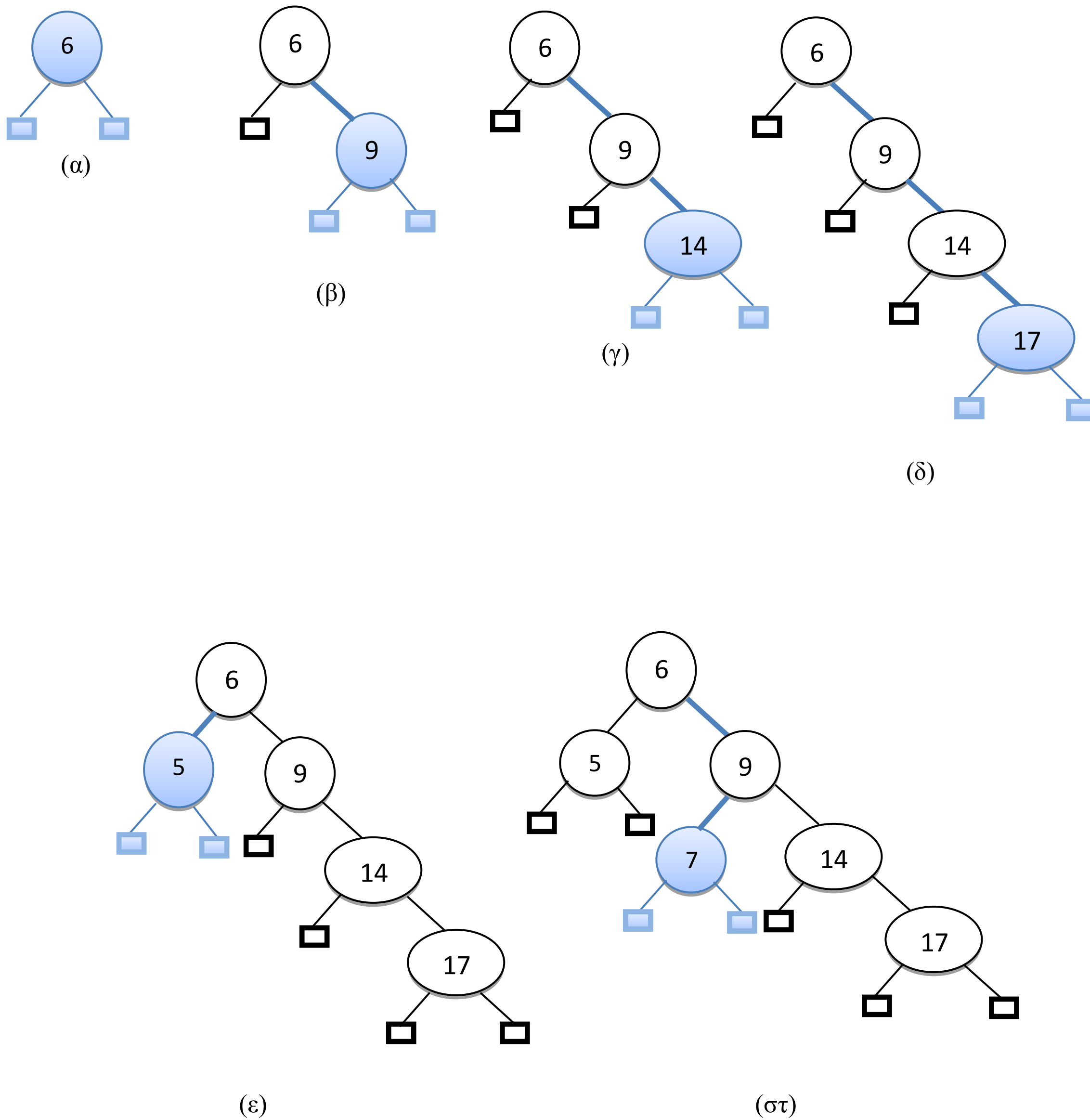

155
17 > 15
13
18
17 < 18
6
14
16
19
5
9

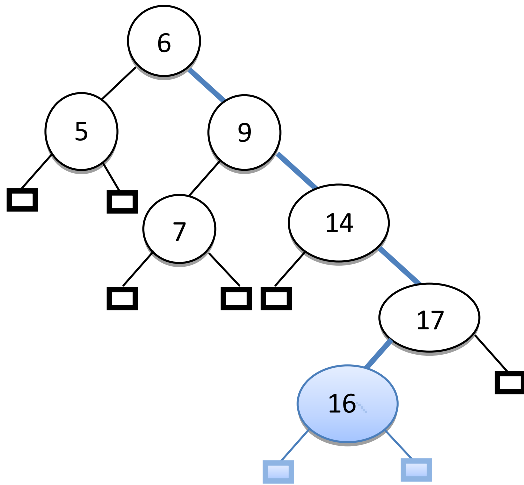


Εισαγωγή Στοιχείου: Η εισαγωγή ενός στοιχείου *i* περιλαμβάνει μία αναζήτηση βάσει του κλειδιού του *x*. Εάν το δένδρο διαθέτει ήδη αντικείμενο με κλειδί *x,* τότε η διαδικασία σταματά. Διαφορετικά, τοποθετείται στην θέση του κατάλληλου κενού φύλλου του κόμβου που επιστρέφει η διαδικασία ψαξίματος, έτσι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

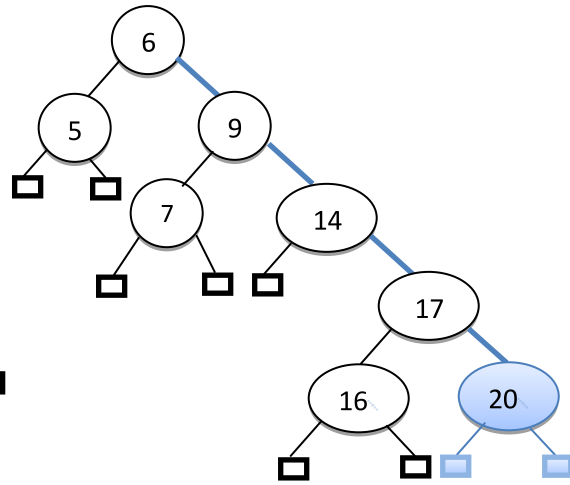
```
Algorithm INSERTITEM(item i)
Input: Ένα στοιχείο i
Output: Ο κόμβος όπου το i θα τοποθετηθεί, εάν δεν υπάρχει ήδη.
1. insNode = FindNode (( x = κλειδί του i), ρίζα του δένδρου);
2. if (το κλειδί του insNode == x)
3.        return null;
4. else if (το κλειδί του insNode > x){
5.        δημιούργησε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

Το σχήμα 3.3 παρουσιάζει παράδειγμα 12 διαδοχικών ενθέσεων σε ένα αρχικώς άδειο δένδρο αναζητήσεως. Με έντονο χρώμα εικονίζονται τα μονοπάτια αναζητήσεως, ήτοι οι δείκτες που ακολουθούν οι αντίστοιχες διαδικασίες αναζητήσεως προκειμένου να εντοπιστούν οι θέσεις τοποθετήσεως των νέων κόμβων.

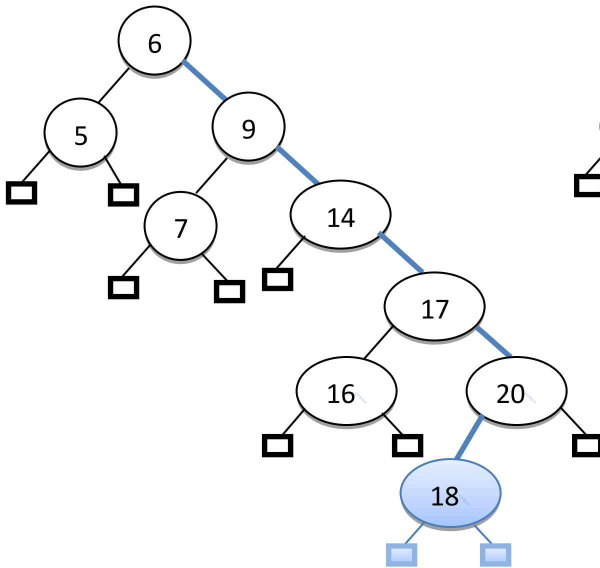




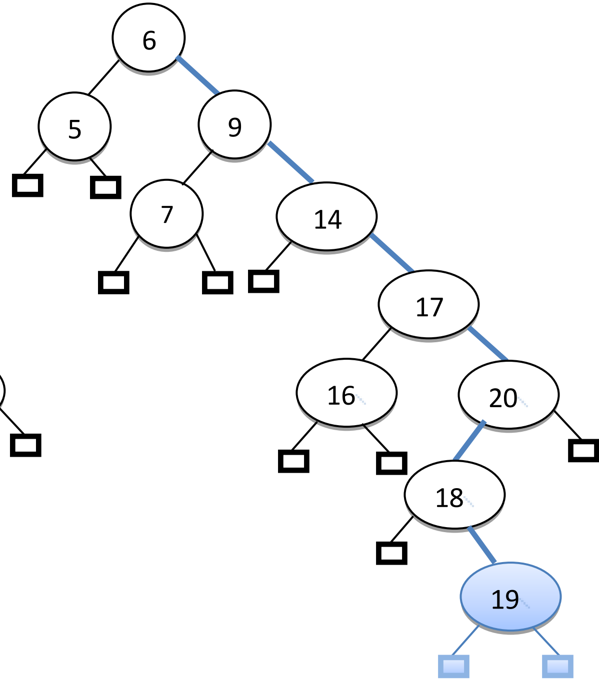
(ζ)



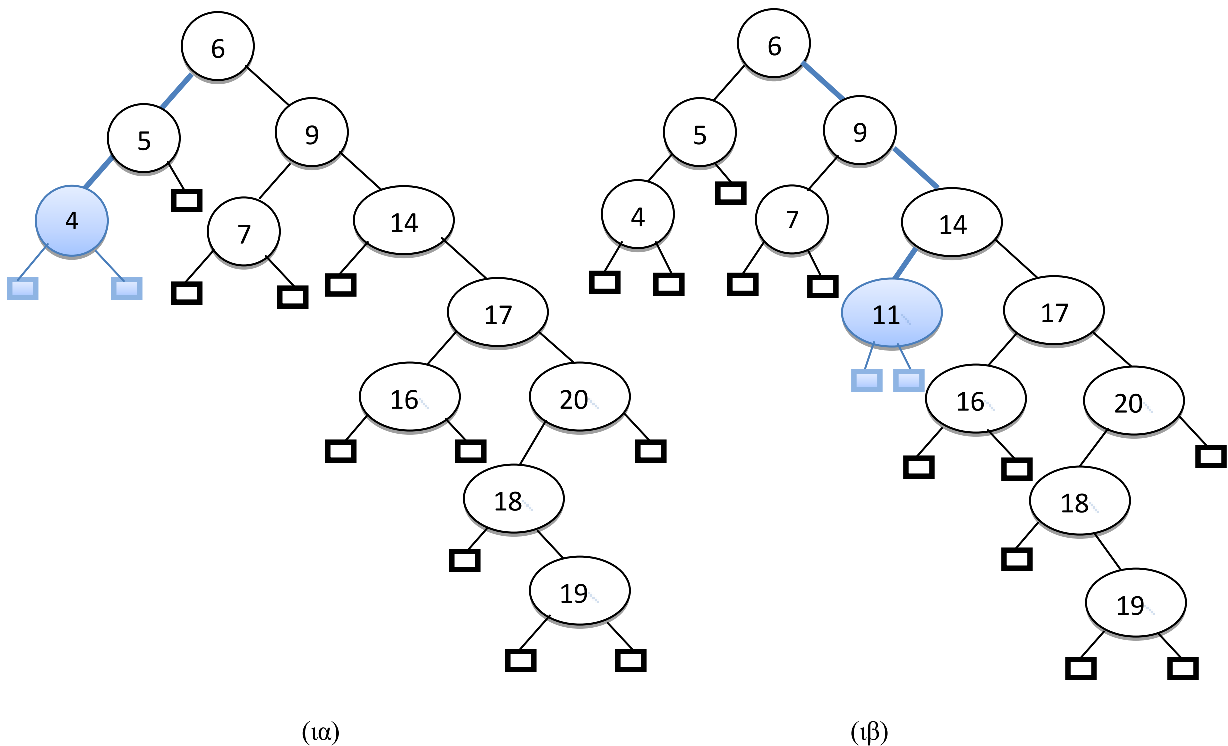
(η)



(θ)



(ι)

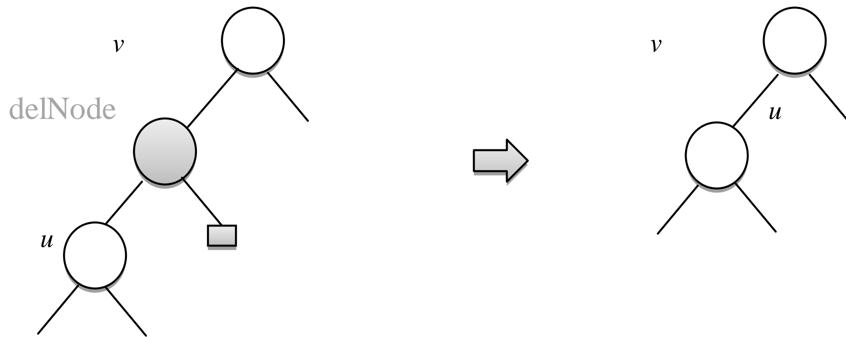


Σχήμα 3.3: (α) – (ιβ) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 16, 20, 18, 19, 4, 11.

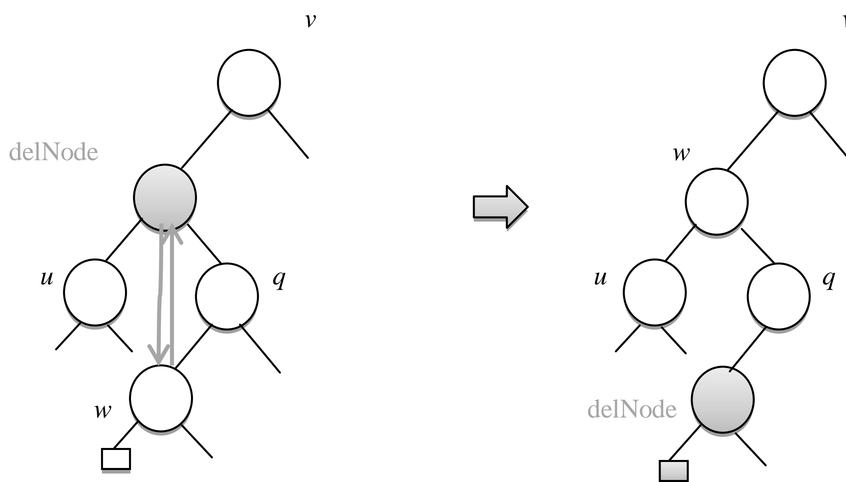
Διαγραφή Στοιχείου: Η διαγραφεί ενός στοιχείου *i* είναι ελάχιστα δυσκολότερη. Περιλαμβάνει και αυτή μία αναζήτηση βάσει του κλειδιού του *x*. Εάν το δένδρο δεν διαθέτει αντικείμενο με κλειδί *x*, τότε η διαδικασία σταματά. Διαφορετικά, έστω delNode ο κόμβος που περιέχει το *i*. Διακρίνουμε δύο περιπτώσεις:

(α) Εάν ο delNode διαθέτει τουλάχιστον ένα κενό φύλλο, προκειμένου να διατηρηθεί η αμετάβλητη συνθήκη που διέπει το δένδρο, καταργείται ο delNode και το δεύτερο, ενδεχομένως μη κενό, παιδί του *u* συνδέεται με τον πατέρα *u* του delNode (Σχήμα 7.6(α)).

(β) Διαφορετικά, ο delNode διαθέτει μη κενό δεξιό γιο *q*(Σχήμα 3.4(β)). Οπότε, ανταλλάσουμε το στοιχείο(Item) του delNode με αυτό του βαθύτερου, αριστερότερου μη κενού απογόνου *w* του δεξιού υποδένδρου T_q. Κατά αυτόν τον τρόπο, είναι σαν οι *w* και delNode να αλλάξουν θέσεις. Οπότε, δημιουργούνται οι συνθήκες της περιπτώσεως (α), η οποία και εφαρμόζεται, ώστε ο delNode να διαγραφεί, δίχως παραβίαση της αμετάβλητης συνθήκης.



(α)



(β)

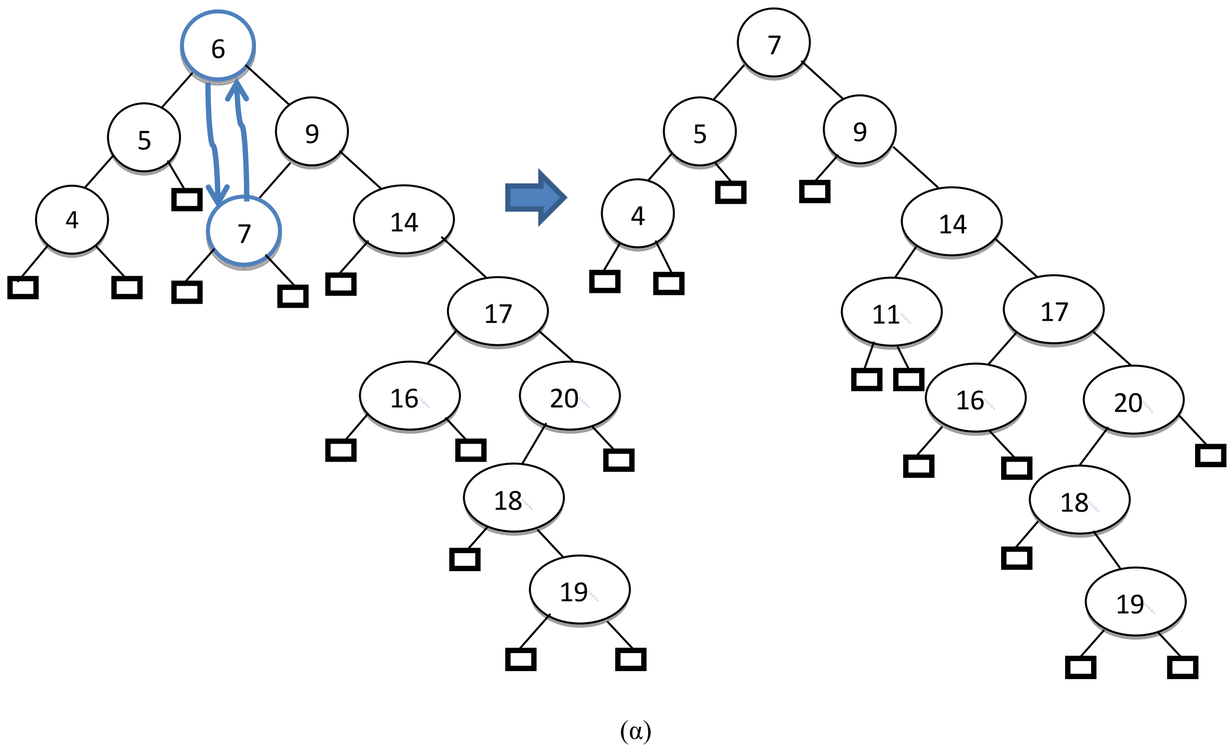
Σχήμα 3.4: Περιπτώσεις αποσβέσεως: (α) άμεση, και (β) ανταλλαγή με τον αριστερότερο απόγονο του δεξιού υποδένδρου.

Τυπικότερα, έχουμε:

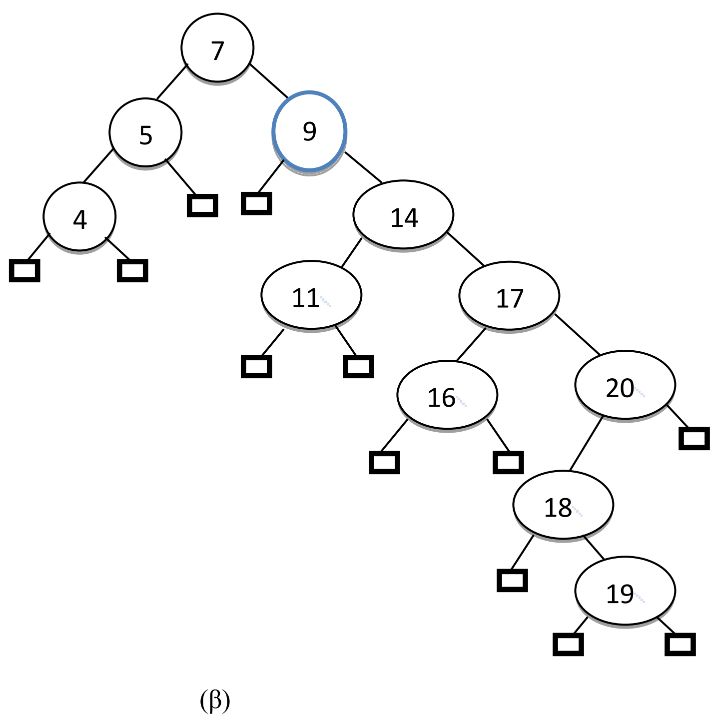
```
Algorithm DELETEITEM(Object k)
Input: Ένα κλειδί k
Output:  Η απόσβεση του κόμβου που περιέχει  item με κλειδί k,επιστρέφοντας τον
εναπομείναντα εμπλεκόμενο κόμβο.
1. delNode = FindNobe ( k, ρίζα του δένδρου);
2. if (το κλειδί του delNode != k)
3.        return null;
4. if(delNode έχει τουλάχιστον ένα παιδί){
5.        σβήσε τον delNode όπως στα LinkedBinaryTrees;
6. επέστρεψε τον άλλο γιο;
7. }
8. else {
9.        βρες τον κόμβο w με το μικρότερο κλειδί του δεξιού υποδένδρου του
          delNode;
10. αντάλλαξε τα στοιχεία τους;
11. σβήσε τον w όπως στην απλή περίπτωση;
12. }

end of DELETEITEM
```

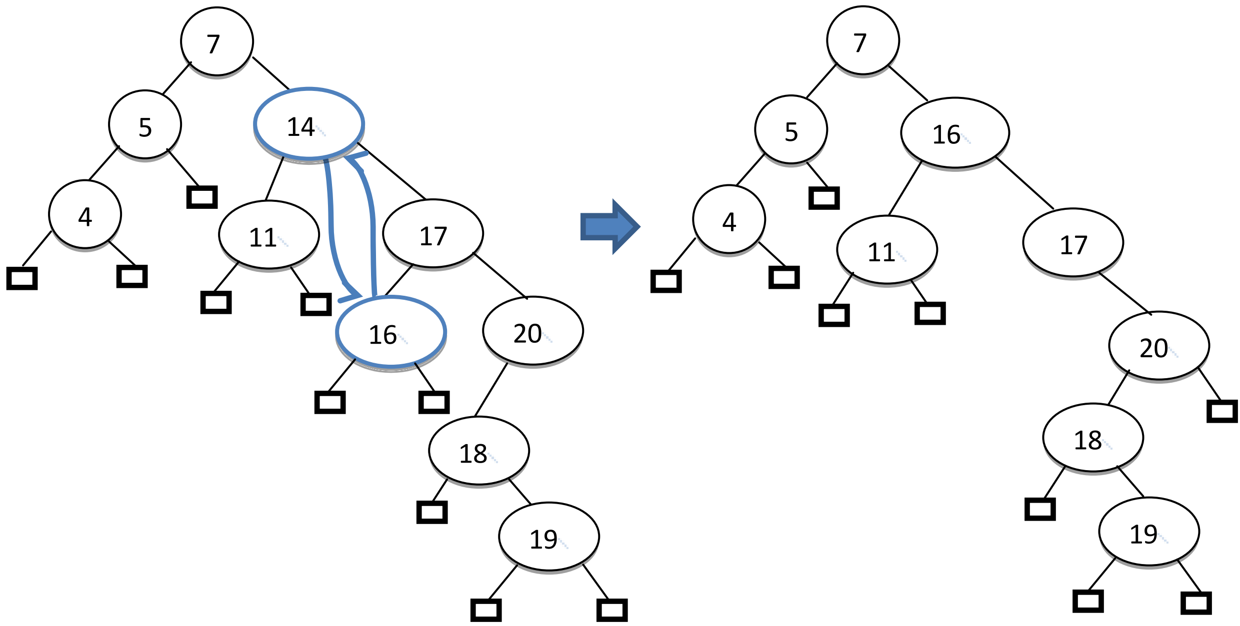
Το σχήμα 3.5 εμφαίνει παράδειγμα 6 διαδοχικών αποσβέσεων στο δένδρο του Σχήματος 3.3(ιβ). Παρατηρήστε πως αντιμετωπίζονται οι δύσκολες περιπτώσεις των 6 και 14 – Σχήματα (α) και (γ), αντιστοίχως.



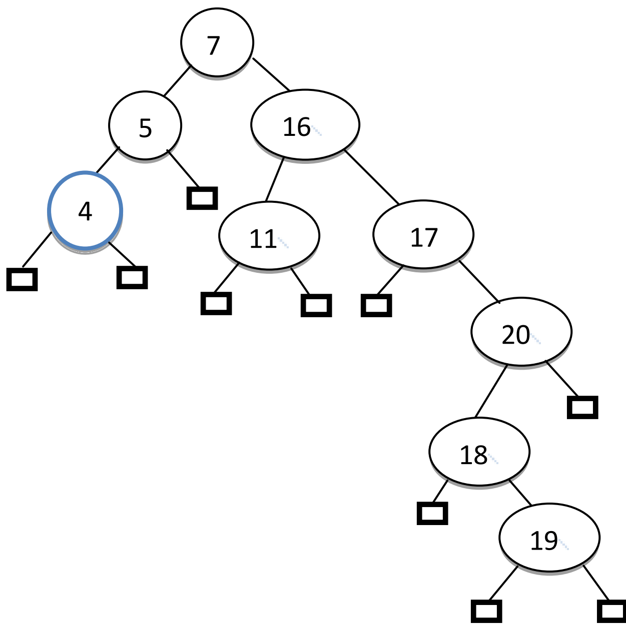
(α)

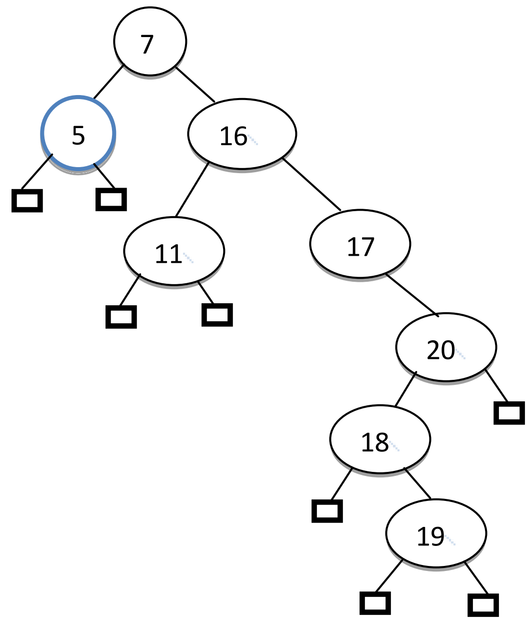


(β)

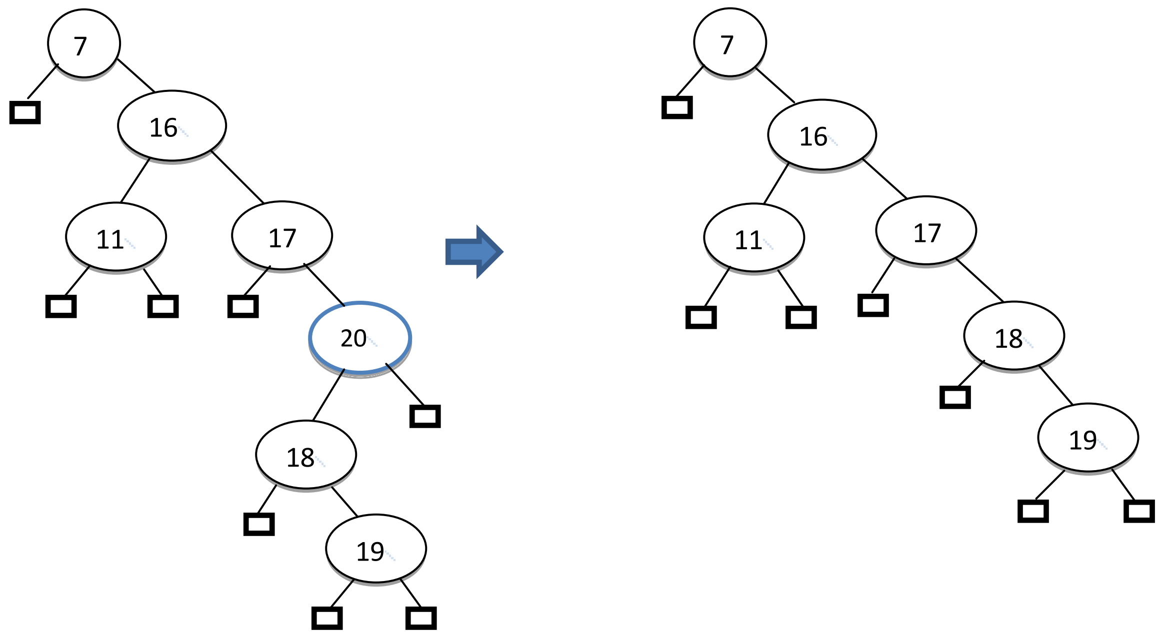


(γ)





(δ) (ε)



(στ)

Σχήμα 3.5: (α) – (στ) Διαδοχικές αποσβέσεις των 6, 9, 14, 4, 5, 20.

3.2.1 Ανάλυση Πολυπλοκότητας

***Χειρότερη Περίπτωση*:** Η πράξη της αναζητήσεως κυριαρχεί στην πολυπλοκότητα, καθώς από αυτή εξαρτώνται και η ένθεση και η απόσβεση. Όριο, δε, στην πολυπλοκότητά της αποτελεί το ύψος του εμπλεκόμενου δένδρου, επειδή η διαδικασία επεξεργάζεται ένα μονοπάτι αναζητήσεως, το οποίο, ενδεχομένως, να καταλήξει σε φύλλο. Από την άλλη, το ύψος του δένδρου μπορεί να είναι γραμμικό στο πλήθος των αποθηκευμένων στοιχείων. Οπότε,

Θεώρημα 3.2: Σε ένα δυναμικό, αζύγιστο, δυαδικό δένδρο αναζητήσεως οι πράξεις της αναζητήσεως, της ενθέσεως και της αποσβέσεως κοστίζουν, στην χειρότερη περίπτωση, γραμμικό, στο πλήθος του υποκείμενου συνόλου, χρόνο.

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

Λήμμα 3.1 Έστω $\wp i(Tv)$ και $\wp e(Tv)$, αντίστοιχα, το εσωτερικό και εξωτερικό μήκος μονοπατιού ενός δένδρου *Tv* ρίζα τον κόμβο *v,* το μέγεθος του οποίου, σε πλήθος κόμβων, είναι $|Tv|$, ενώ διαθέτει αριστερό και δεξιό υπόδενδρο, T_v^l και T_v^r αντίστοιχα. Τότε ισχύουν τα εξής:

- $\wp i(Tv) = \wp i(T_v^l) + \wp i(T_v^r) + |T_v| - 1$
- $\wp e(T_v) = \wp e(T_v^l) + \wp e(T_v^r) + |T_v| + 1$
- $\wp e(T_v) = \wp i(T_v) + 2|T_v|$

Στην συνέχεια θα ασχοληθούμε με το μέσο μήκος εσωτερικού μονοπατιού σε ένα αζύγιστο δένδρο αναζητήσεως *T*, μεγέθους $|T| = n$ στοιχείων. Θα θεωρήσουμε πως το *T* είναι ένα τυχαίο δένδρο αναζητήσεως. Έστω, λοιπόν,

$$\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$$

τα κλειδιά που μετέχουν στην δημιουργία του *T,* διατεταγμένα κατά αύξουσα τιμή. Η τυχαιότητα προκύπτει από την θεώρηση ότι το *T* χτίστηκε μέσω *n* διαδοχικών ενθέσεων και τα ν συμμετέχοντα κλειδιά είναι ανεξάρτητοι και ομοιόμορφοι 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 δένδρου, καθώς υπάρχουν $(n - 1)!$ ακολουθίες ενθέσεων που ξεκινούν με το α_i ως πρώτο εισαχθέν κλειδί.

Σημειώστε πως η τοποθέτηση του *i*-στού μικρότερου στοιχείου α_i στην ρίζα συνεπάγεται, εξορισμού των δυαδικών δένδρων αναζητήσεως, ότι το αριστερό υπόδενδρο T^l_{ai} θα περιέχει $i - 1$ κλειδιά, ενώ το δεξιό υπόδενδρο T^r_{ai} θα στεγάζει $n - i$ κλειδιά.

Λήμμα 3.2 Το μέσο εσωτερικό μήκος μονοπατιού σε ένα τυχαίο δυαδικό δένδρο αναζητήσεως *T, n* στοιχείων, είναι περίπου *1.39n logn,* ενώ το μέσο εξωτερικό μήκος του είναι περίπου *1.39n logn + 2n*

Ο ισχυρισμός για το μέσο εξωτερικό μήκος μονοπατιού προκύπτει άμεσα από την Σχέση (γ) του Λήμματος 3.1.

Επομένως, είμαστε σε θέση να διατυπώσουμε το εξής συμπέρασμα:

Λήμμα 3.3 Έστω Τα ένα τυχαίο δυαδικό δένδρο αναζητήσεως μεγέθους n. Το αναμενόμενο κόστος ενός επιτυχημένου και ενός αποτυχημένου ψαξίματος είναι λογαριθμικό στο πλήθος n των στοιχείων.

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

4. Δένδρα AVL

4.1 Ορισμός

Προκειμένου ένα δένδρο αναζητήσεως να επιδεικνύει καλή συμπεριφορά και στις τρεις βασικές πράξεις, θα πρέπει να τεθούν όρια στο ύψος του. Το πρώτο δένδρο που πέτυχε κάτι τέτοιο ήταν το ***AVL.*** Η πρωτοτυπία της προτάσεώς τους ήταν η επιβολή της κατώθι αμετάβλητης συνθήκης:

Ένα δυαδικό δένδρο T καλείται δένδρο AVL αν και μόνον αν τα ύψη των δύο υποδένδρων κάθε εσωτερικού κόμβου u διαφέρει το πολύ κατά ένα.

Όπως θα δούμε και σε παρακάτω ενότητα, η ανωτέρω ιδιότητα *εγγυάται την λογαριθμικότητα του ύψους*. Για την διατήρηση της συνθήκης αυτής, θα χρειαστεί να διατηρήσουμε σε κάθε κόμβο και το ύψος του.

4.2 Περιγραφεί των Βασικών Πράξεων

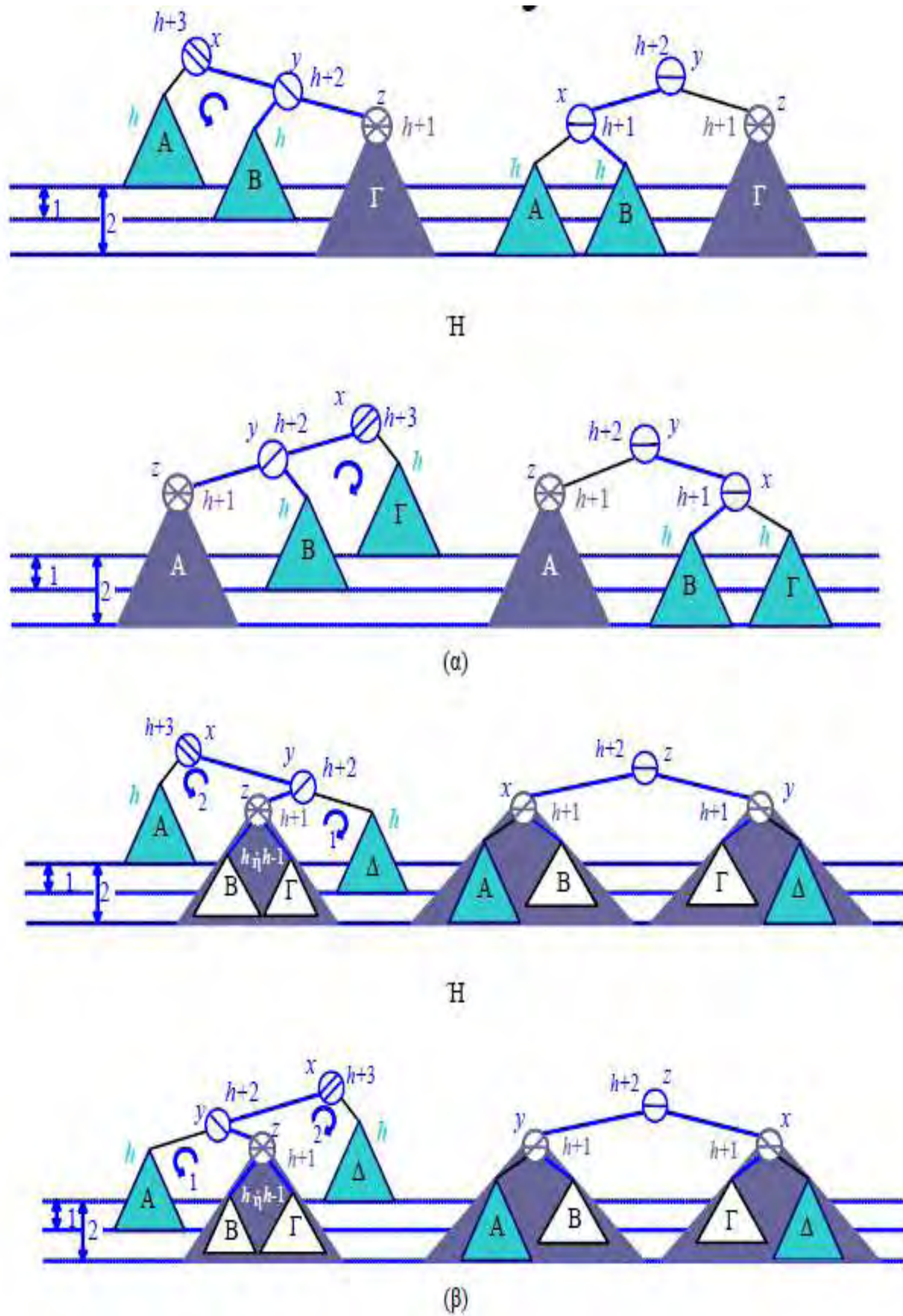
Εισαγωγή Στοιχείου: Είναι δυνατόν ο νέος κόμβος να προκαλέσει παραβίαση της συνθήκης AVL, η οποία πρέπει να θεραπευτεί βάσει κατάλληλων ενεργειών. Το σχεδιάγραμμα του αλγορίθμου της ενθέσεως έχει ως ακολούθως:

Algorithm INSERTITEM(Item i)
Input: Ένα στοιχείο i
Output: Τοποθέτηση του i σε έναν κόμβο, εάν δεν υπάρχει ήδη

1. Κλήση της αντίστοιχης μεθόδου εισαγωγής των απλών, αζύγιστων δυαδικών δένδρων;
2. Με αφετηρία τον κόμβο της ενθέσεως, αναρρίχηση προς τα επάνω, με επιδιόρθωση των υψών, μέχρι να εντοπιστεί ο πρώτος μη ισοζυγισμένος κόμβος *u*;
3. Ισοζύγιση του *u* βάσει της κατάλληλης επαναζυγιστικής πράξης (απλής ή διπλής περιστροφής) του Σχήματος 4.1;

end of INSERTITEM

Στο Σχήμα 4.1 εικονίζονται οι δύο περιπτώσεις που μπορεί να εμφανιστούν:



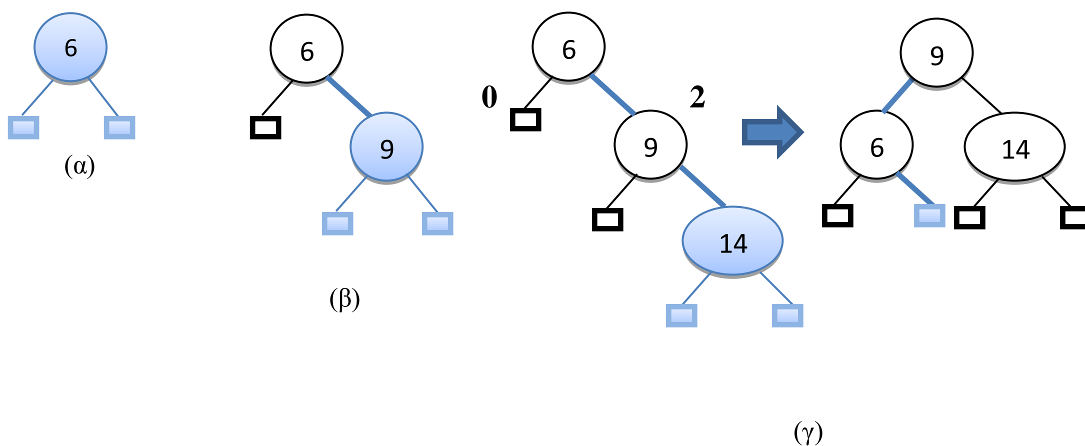
Σχήμα 4.1: Περιπτώσεις αζύγιστων κόμβων μετά από ένθεση και αντίστοιχης επαναζύγισης. Οι γραμμές εντός των κόμβων δηλώνουν την ισορροπία (-), την κατά μία μονάδα υψομετρική διαφορά (\ ή /), την διπλή διαφορά (\\ ή //) ή κάποια από τις τρεις περιπτώσεις: (α) αριστερή ή δεξιά απλή περιστροφή (rot), (β) αριστερή ή δεξιά διπλή περιστροφή (drot).

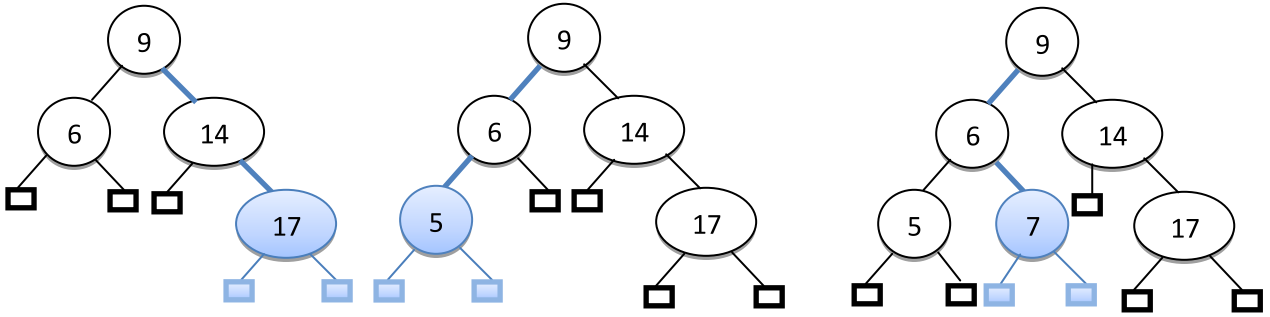
(α) Οι κόμβοι *x,y,z* σχηματίζουν δεξιό ή αριστερό ευθές μονοπάτι, το νέο στοιχείο έχει εισαχθεί στο υποδένδρο Γ και ο *x* παραβιάζει την συνθήκη ισοζύγισης λόγω της αυξήσεως του ύψους του από *h + 2 σε h + 3.* Τότε, αρκεί μία, αριστερή ή δεξιά, ***απλή περιστροφή (single rotation – rot)*** στον *x* για να επανέλθει το ύψος του εικονιζόμενου υποδένδρου στην προηγούμενη τιμή του *h + 2* και, άρα, να αποκατασταθεί η ισορροπία στο δένδρο.

(β) Οι κόμβοι *x,y,z* σχηματίζουν δεξιά ή αριστερή γωνία, το νέο στοιχείο έχει εισαχθεί στο υπόδενδρο Β ή Γ και ο *x* παραβιάζει την συνθήκη λόγω της αυτής έως, κατά μία μονάδα, του ύψους του σε *h + 3.* Τότε, αρκεί μία, αριστερή ή δεξιά, ***διπλή περιστροφή (double rotation - drot)*** για να επανέλθει το ύψος του *z* στην προηγούμενη τιμή του και άρα να αποκατασταθεί η ισορροπία στο δένδρο.

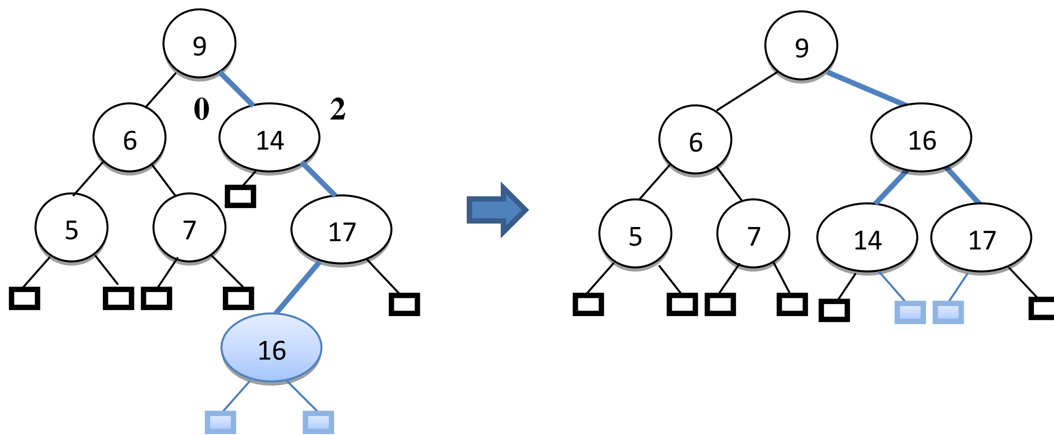
Οι περιστροφές μεταβάλλουν την μορφή του δένδρου, οπότε καλούνται και ***δομικές επαναζυγιστικές πράξεις (reconstruction rebalancing operations).*** Εν αντιθέσει, πράξεις που, απλώς, αλλάζουν βοηθητική πληροφορία, όπως στην περίπτωση μας το ύψος, χαρακτηρίζονται ως ***μη δομικές επαναζυγιστικές πράξεις.***

Τα Σχήματα 4.2 και 4.3 παρουσιάζουν ένα παράδειγμα 12 διαδοχικών ενθέσεων σε ένα αρχικώς άδειο δένδρο AVL. Με έντονο χρώμα εικονίζονται τα μονοπάτια αναζητήσεως, ήτοι οι δείκτες που ακολουθούν οι αντίστοιχες διαδικασίες αναζητήσεως προκειμένου να εντοπιστούν οι θέσεις τοποθετήσεως των νέων κόμβων.

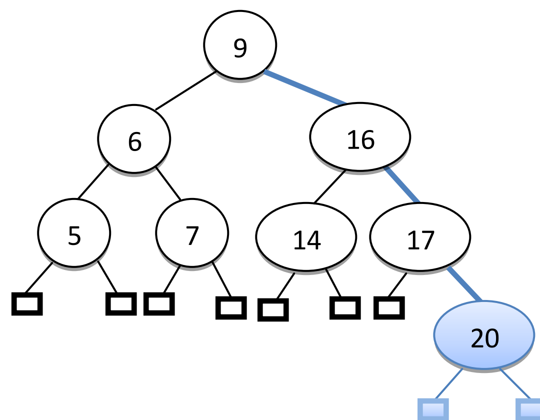




(δ) (ε) (στ)

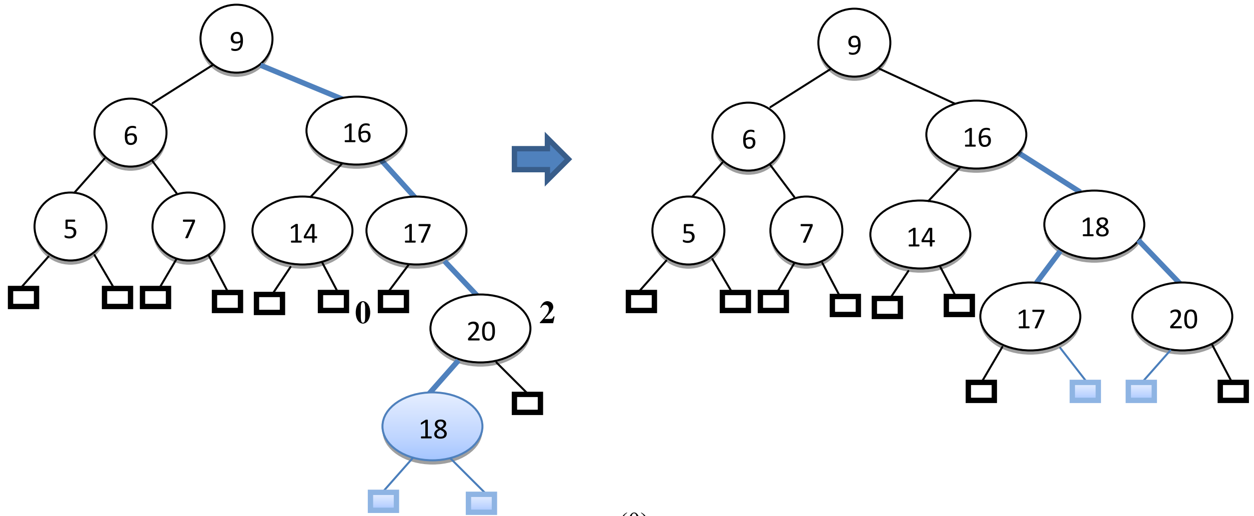


(ζ)

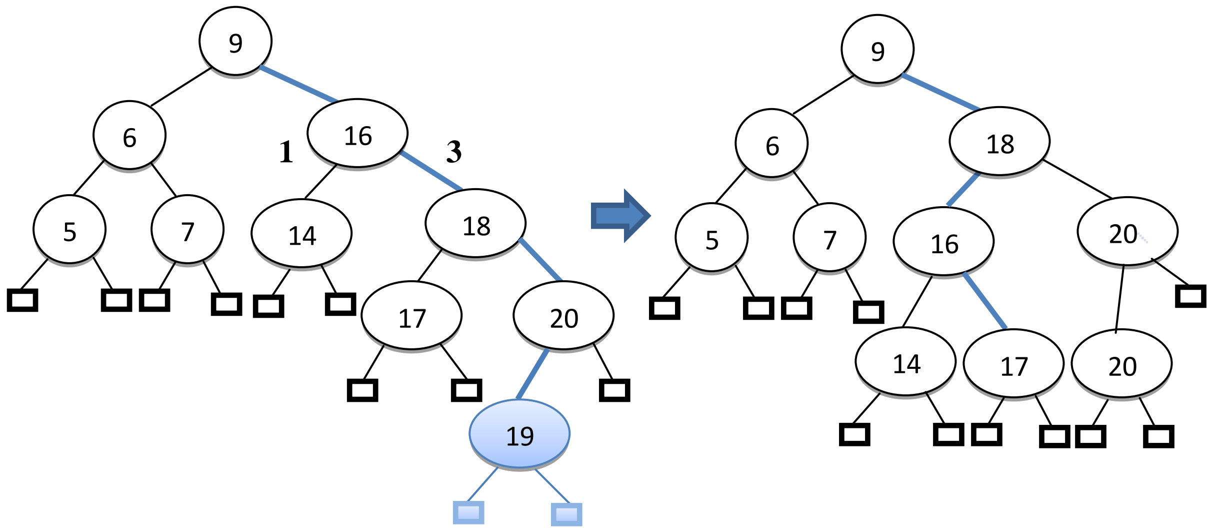


(η)

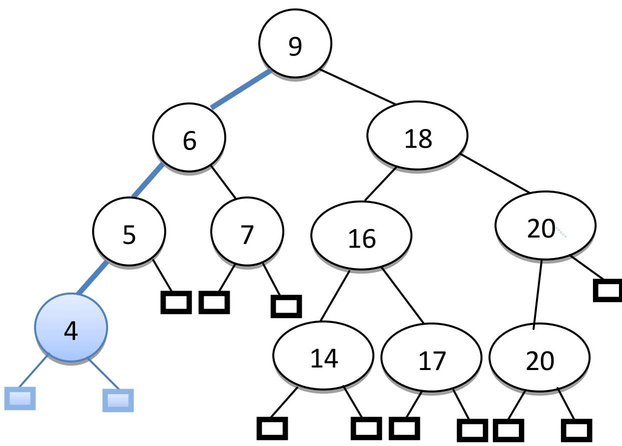
Σχήμα 4.2: (α) – (η) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 17, 20.



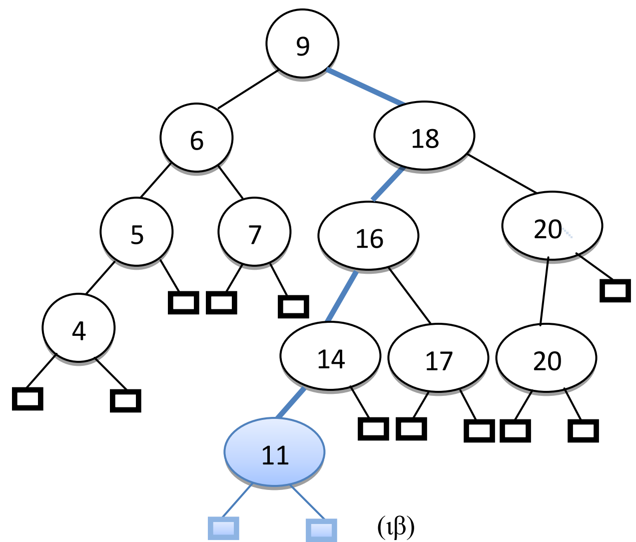
(θ)



(ι)



(ια)



(ιβ)

Σχήμα 4.3: Συνέχεια (θ) – (ιβ) 18, 19, 4, 11.

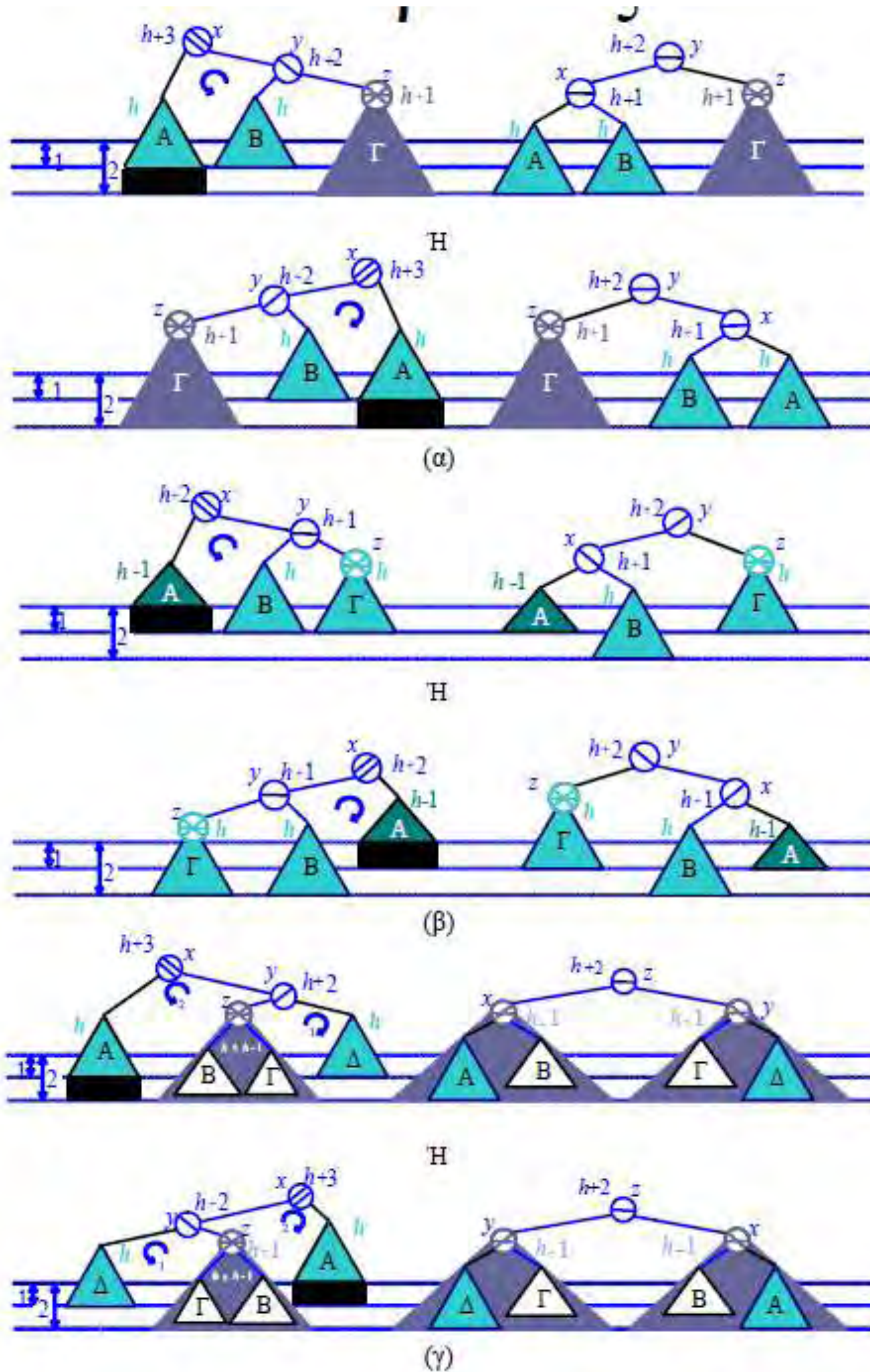
Διαγραφή Στοιχείου: Η περιγραφεί της πράξεως της αποσβέσεως, σε ψευδογλώσσα, έχει ως εξής:

Algorithm DELETEITEM(Item *k*)
Input: Ένα κλειδί *k*
Output: Η απόσβεση του κόμβου που περιέχει item με κλειδί *k*, επιστρέφοντας τον εναπομείναντα εμπλεκόμενο κόμβο

1. Κλήση της αντίστοιχης μεθόδου των απλών δυαδικών δένδρων;
2. Με αφετηρία τον κόμβο της αποσβέσεως, αναρρίχηση προς την ρίζα, με ταυτόχρονη επιδιόρθωση των υψών;
3. Σε κάθε κόμβο *u,* όπου σημειώνεται παραβίασει της ισοζύγισης, εκτέλεση της κατάλληλης ισοζυγιστικής πράξης του Σχήματος 4.4;

end of DELETEITEM

Στο Σχήμα 4.4 εικονίζονται οι τρεις περιπτώσεις που μπορεί να εμφανιστούν, μετά την διαγραφή ενός στοιχείου:



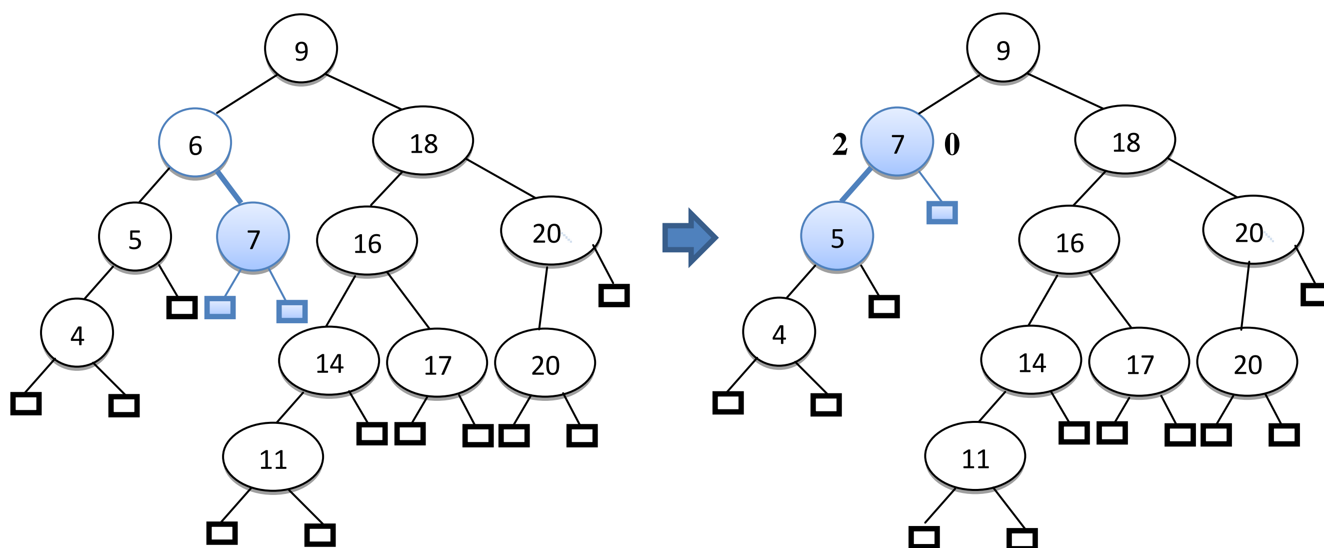
Σχήμα 4.4: Περιπτώσεις αζύγιστων κόμβων μετά από απόσβεση και αντίστοιχης επαναζύγισης: (α) αριστερή ή δεξιά, μη τερματική περιστροφή (rot 1), (β) αριστερή ή δεξιά, τερματική περιστροφή (rot 2), (γ) αριστερή ή δεξιά διπλή, μη τερματική περιστροφή (drot).

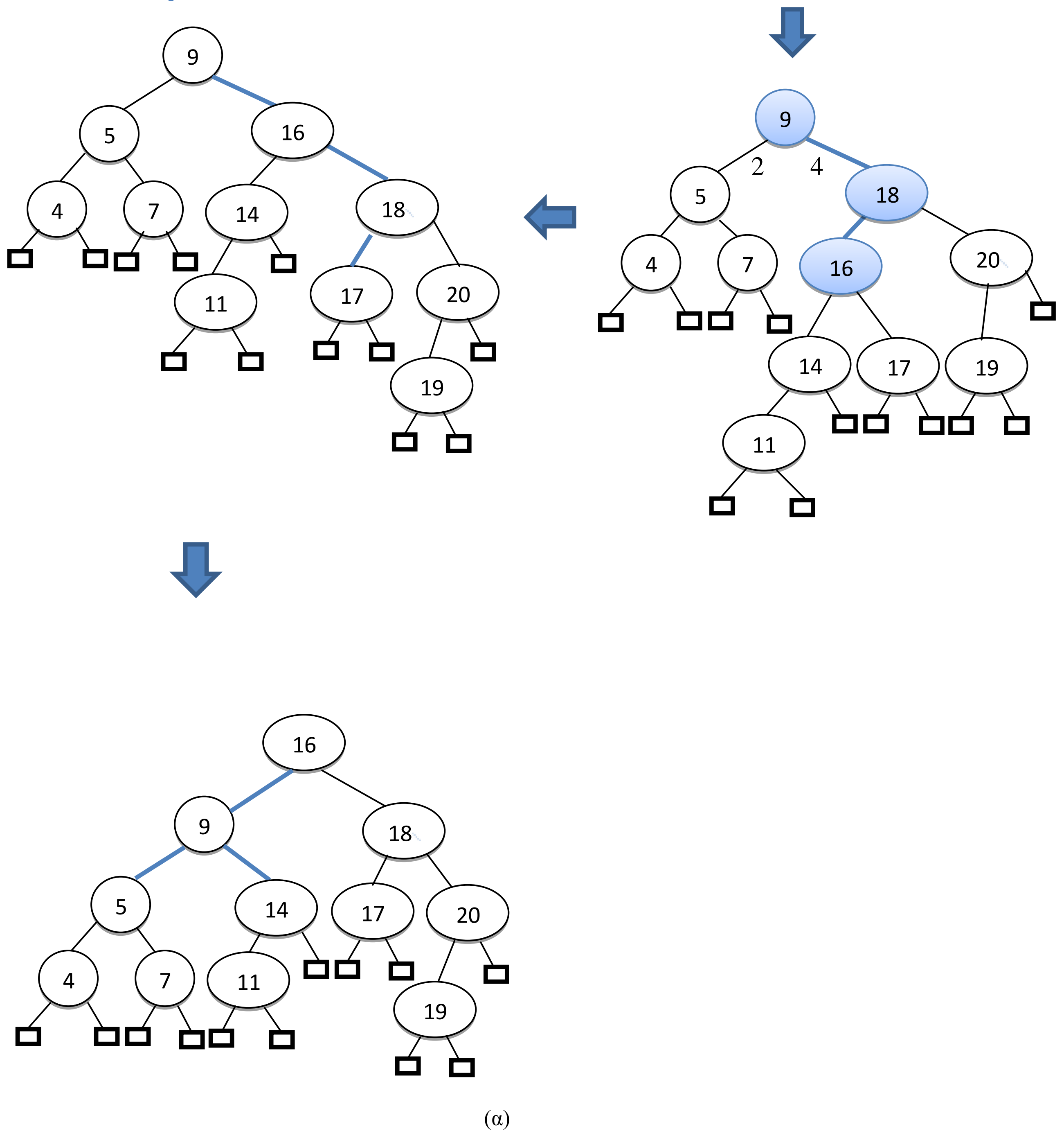
(α) Οι κόμβοι *x,y,z* σχηματίζουν δεξιό ή αριστερό ευθές μονοπάτι, το στοιχείο έχει διαγραφεί από το υποδένδρο Α, μειώνοντας το ύψος του, κατά μία μονάδα, σε *h* και ο *x* παραβιάζει την συνθήκη. Τότε, μία, αριστερή ή δεξιά, απλή περιστροφή (rot1) στον *x* ισοζυγίζει το υποδένδρο, μειώνοντας, όμως, το ύψος του κορυφαίου κόμβου κατά μία μονάδα. Γεγονός που καθιστά υποχρεωτική την εξέταση των προγόνων του *y* για τυχόν δημιουργία παραβιάσεως. Αυτού του είδους οι περιστροφές, οι οποίες θεραπεύουν τοπικά το πρόβλημα, αλλά δεν εγγυώνται την οριστική εξάλειψή του, καθώς, ενδεχομένως, το μεταθέτουν σε υψηλότερους κόμβους, καλούνται ***μη τερματικές.***

(β) Οι κόμβοι *x,y,z* σχηματίζουν δεξιό ή αριστερό ευθές μονοπάτι, το στοιχείο έχει διαγραφεί από το υποδένδρο Α, μειώνοντας το ύψος του, κατά μία μονάδα, σε $h-1$ και ο *x* παραβιάζει την συνθήκη. Τότε, μία, αριστερή ή δεξιά, απλή περιστροφή (rot2) στον *x,* ισοζυγίζει, κατά τρόπο τερματικό, το υποδένδρο, διατηρώντας το προηγούμενο ύψος.

(γ) Οι κόμβοι *x,y,z* σχηματίζουν δεξιό ή αριστερό γωνία, το στοιχείο έχει αποσβεστεί από το υποδένδρο Α, μειώνοντας το ύψος του, κατά μία μονάδα, σε *h* και ο *x* παραβιάζει την συνθήκη. Τότε, μία, αριστερή ή δεξιά, διπλή περιστροφή (drot) ισοζυγίζει το υποδένδρο, μειώνοντας, όμως, το ύψος του κατά μία μονάδα. Με συνέπεια, η επαναζυγιστική πράξη να είναι μη τερματική.

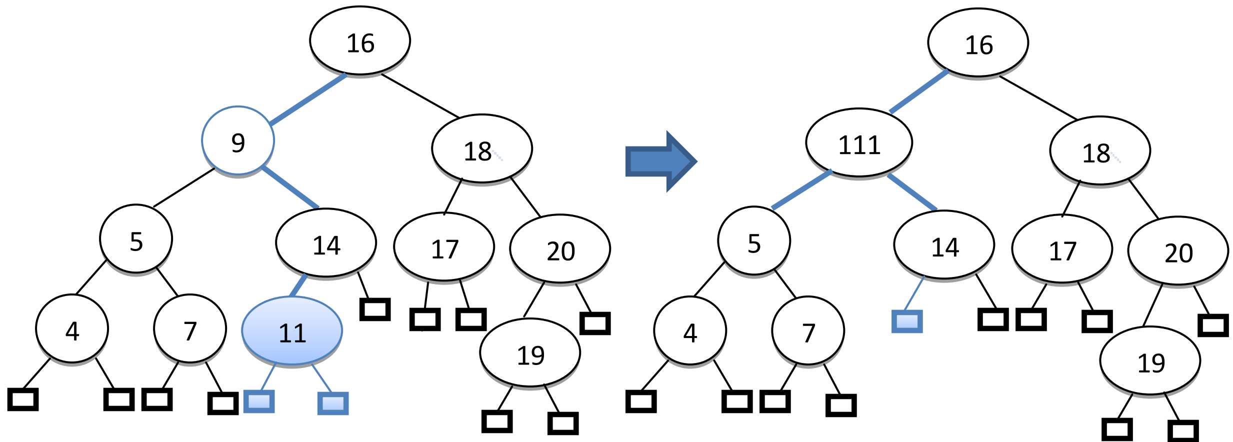
Στα Σχήματα 4.5, 4.6, 4.7 εικονίζονται οι αλλαγές που επιφέρουν 6 διαδοχικές αποσβέσεις στο δένδρο του Σχήματος 4.3(ιβ).



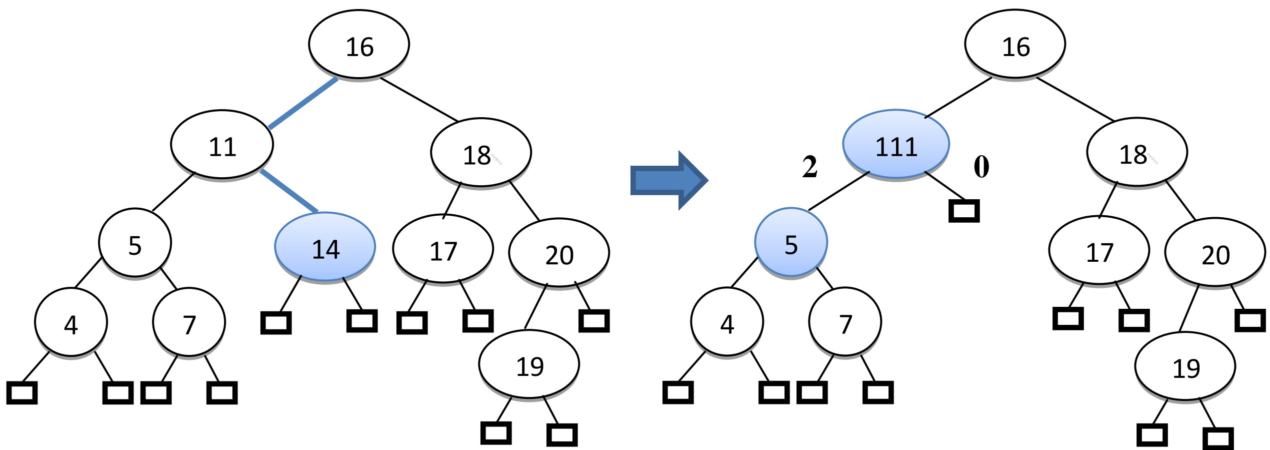


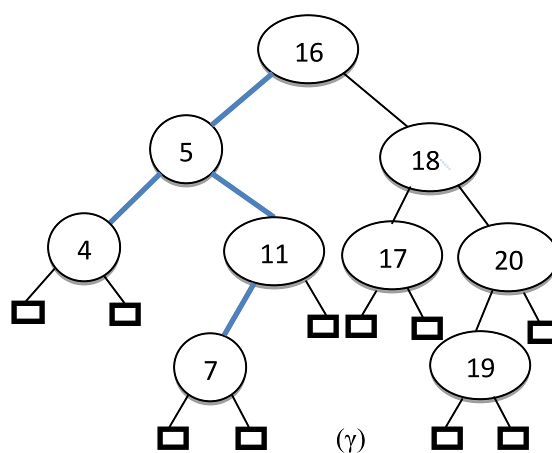
(α)

Σχήμα 4.5: Διαδοχικές αποσβέσεις (α) 6.



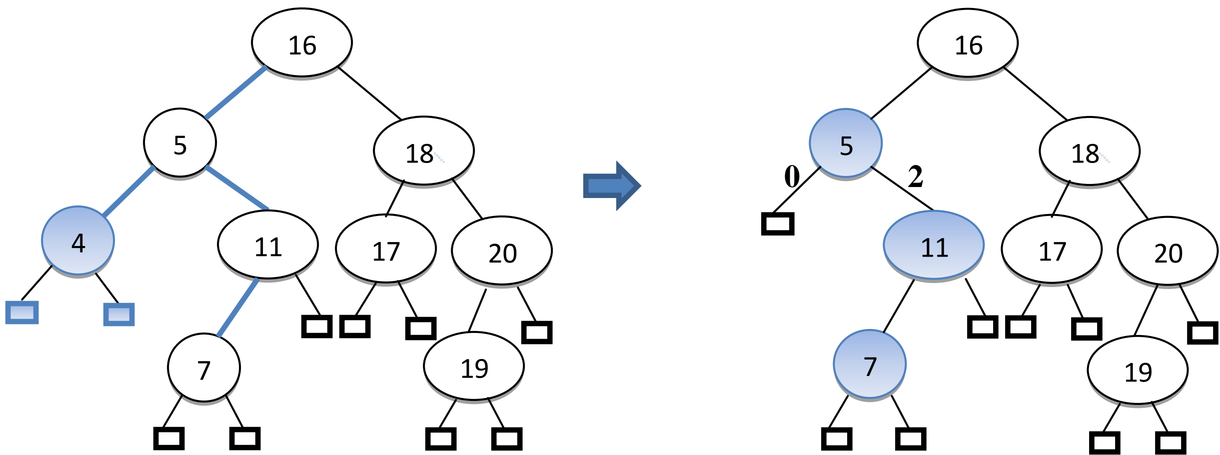
(β)

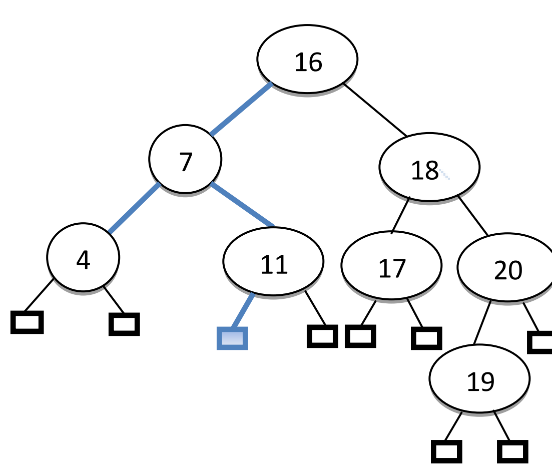




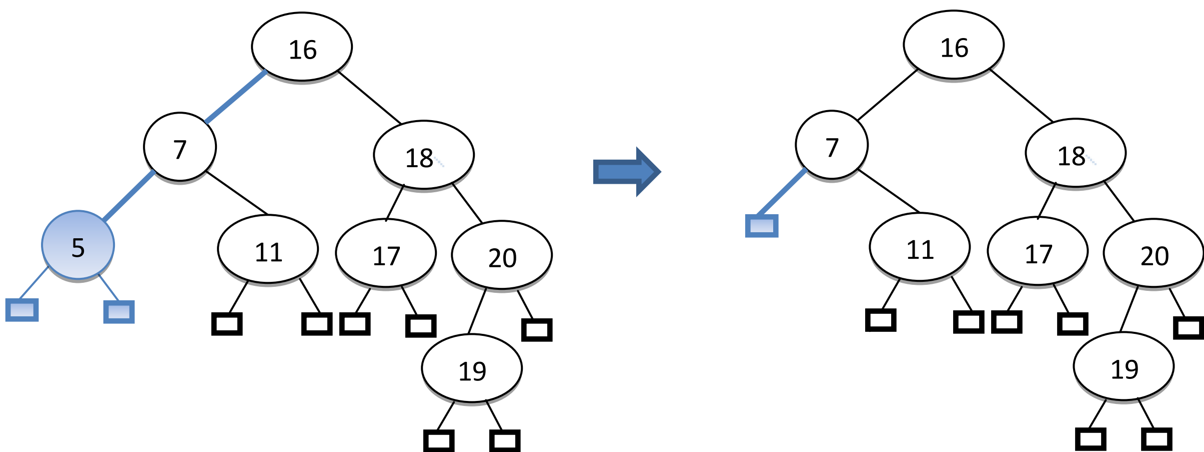
(γ)

Σχήμα 4.6: Συνέχεια (β) – (γ) 9, 14.

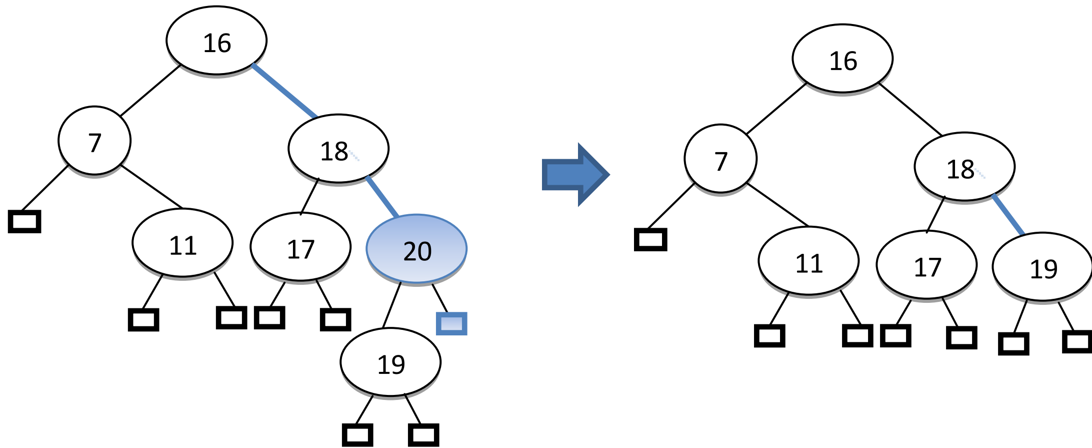




(δ)



(ε)



(στ)

Σχήμα 4.7: Συνέχεια (δ) – (στ)

4.3 Ιδιότητες – Ανάλυση Πολυπλοκότητας

Προκειμένου να δεσμεύσουμε τις πολυπλοκότητες των δένδρων AVL, πρέπει να βρεθεί η πολυπλοκότητα του ύψους τους. Έχουμε, λοιπόν, τα εξής:

Λήμμα 4.1: Κάθε υποδένδρο ενός δένδρου AVL είναι επίσης δένδρο AVL.

Λήμμα 4.2: Το ύψος h ενός δένδρου AVL T, n στοιχείων, είναι O(log n).

Παρατηρούμε ότι

(α) το ύψος ενός δένδρου F_h είναι h,

(β) τα δένδρα Fibonacci είναι και δένδρα AVL, και μάλιστα «χειρότερης» περιπτώσεως, και

(γ) το πλήθος των εσωτερικών κόμβων ικανοποιεί την αναδρομική σχέση:

$$F_h = F_{h-1} + F_{h-2} + 1,\ F_0 = 0,\ F_1 = 1$$

Οπότε, προσθέτοντας ένα και στα δύο μέλη προκύπτει η σχέση που διέπει το πλήθος των εξωτερικών κόμβων (κενών φύλλων):

$$(F_h + 1) = (F_{h-1} + 1) + (F_{h-2} + 1)$$

$$\Phi_h = \Phi_{h-1} + \Phi_{h-2},\ \Phi_0 = 1,\ \Phi_1 = 2$$

Όμως *Φh*= *Ah*+1, όπου *Ah* η ακολουθία Fibonacci!

Συνεπώς,

$$\Phi_h = n+1 = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - \left(\frac{1-\sqrt{5}}{2}\right)^{h+2}}{\sqrt{5}} \overset{\frac{1-\sqrt{5}}{2}<1}{\geq} \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{h+2} - 1}{\sqrt{5}}$$

και με λογαρίθμιση:

$$h \leq 1.44\log(n+1) - 0.33$$

Από την άλλη, γνωρίζουμε πως κάθε πλήρες δυαδικό δένδρο με *n* εσωτερικούς κόμβους έχει ύψος:

$$h = \log(n+1),$$

που είναι και η καλύτερη περίπτωση.

Θεώρημα 4.1: Ένα δένδρο AVL *n* στοιχείων, στην χειρότερη περίπτωση, επιδεικνύει λογαριθμική συμπεριφορά και για τις τρεις πράξεις. Επιπροσθέτως, μία ένθεση κοστίζει *O(1),* ενώ μία απόσβεση *O(log n)* επαναζυγιστικές πράξεις.

4.4 Ανάλυση Κώδικα *AVL*

Στον κώδικα του παραρτήματος υλοποιούμε τα ***avl trees.*** Παρακάτω γίνεται αναφορά στην βασική δομή και στις βασικές συναρτήσεις για την καλύτερη κατανόηση του αναγνώστη. Επιπλέον έχει προστεθεί ένα σχετικό τμήμα κώδικα για την επίδειξη λειτουργίας αυτών των συναρτήσεων.

Η βασική δομή είναι:

```
struct AVLnode
{
  int data;              //the tree's data. Can be changed at will
  avltree  left;          //left node (child)
  avltree  right;         //right node
```

```
  int height;            //the height of the tree
};
```

που περιλαμβάνει τα απαραίτητα στοιχεία για την υλοποίηση ενός δέντρου. Το αριστερό και δεξί κόμβο («παιδί») και τα δεδομένα του κάθε κόμβου.

Οι βασικές συναρτήσεις που περιλαμβάνονται είναι οι εξής:

```
avlt create_tree(avlt tree, int d)   /*Χρησιμοποιείται για τη δημιουργία ενός δέντρου.*/
avlt delete_tree(avlt tree)          /*Χρησιμοποιείται για τη διαγραφή ενός δέντρου.*/
avlt insert_data(avlt tree, int d)   /*Χρησιμοποιείται για την εισαγωγή δεδομένων σε ένα
                                     δέντρο. Εάν δεν υπάρχει τότε το δημιουργεί αυτόματα*/
avlt search_and_find(int d, avlt tree)   /*Χρησιμοποιείται για την αναζήτηση δεδομένων σε
                                         ένα δέντρο.*/
int max( int a, int b )              /*Επιστρέφει το μέγιστο ανάμεσα από δύο ακεραίους.*/
avltree rotateL( avltree t )         /*Απλή περιστροφή από αριστερά.*/
avltree rotateR( avltree t )         /*Απλή περιστροφή από δεξιά.*/
avltree  drotateL( avltree t )       /*Διπλή περιστροφή από αριστερά.*/
avltree drotateR( avltree t)         /*Διπλή περιστροφή από δεξιά.*/
avltree getmin( avltree t )          /* Βρίσκει το ελάχιστο στοιχείο στο δένδρο και το
                                     χρησιμοποιεί για την πράξη της διαγραφής.*/
avlt delete_node(avlt tree, int d)   /*Χρησιμοποιείται για την διαγραφή δεδομένων σε ένα
                                     δέντρο.*/
void twrite(avltree tree, FILE* f)    /*Ίδια λειτουργικότητα με την display με την διαφορά ότι
                                      γράφει τα αποτελέσματα σε ένα αρχείο.*/
void display(avlt tree)     /*Χρησιμοποιείται για την εκτύπωση των δεδομένων ενός δέντρου.*/
```

Η display προστέθηκε για λόγους επαλήθευσης και επίδειξης των αποτελεσμάτων.

Επιπλέον, στην main() έχει προστεθεί ένα σχετικό τμήμα κώδικα για την επίδειξη λειτουργίας των παραπάνω συναρτήσεων.

```
int main()
{
  avltree mytree=NULL;
  mytree = create_tree(mytree,1);

  start = clock();
  for (i=2;  i<1e6; i++)
          mytree=insert_data(mytree, i);

  fprintf(file,"\n=========================\n");
  fprintf(file,"Display initial tree\n");
  fprintf(file,"=========================\n");
  //twrite(mytree, file);

  delete_node(mytree,6);
  fprintf(file,"\n=========================\n");
  fprintf(file,"Display tree after deletion\n");
  fprintf(file,"=========================\n");
  //twrite(mytree, file);

   delete_node(mytree,11);
   fprintf(file,"\n===========================================\n");
   fprintf(file,"Test tree validity after deleting a non-existing element\n");
   fprintf(file,"============================================\n");
   //twrite(mytree, file);

  mytree=delete_tree(mytree);
```

```
    fprintf(file, "\n=========================================\n");
    fprintf(file,"Display tree after deleting all tree\n");
    fprintf(file,"===========================================\n");
    //twrite(mytree, file);


    end= clock();
    fprintf(file, "That took %f seconds", (end-start)/CLOCKS_PER_SEC);


    fprintf(file, "\ntotal rotations = %d", rotcount);


    system("PAUSE");
    return EXIT_SUCCESS;

}
```

5. Διαγραφή δίχως επαναζύγιση σε δυαδικά δένδρα αναζήτησης

5.1 Εισαγωγή

Πριν ξεκινήσουμε τίθεται το εξής ερώτημα: Το κατά πόσο έχει νόημα να προσπαθήσουμε να διατηρήσουμε την ισορροπία σε ένα δυαδικό δένδρο αναζήτησης μετά από μία εισαγωγή, και όχι μετά από μία διαγραφή;

Μία πιο ακριβής εκδοχή της ερώτησης μας είναι η εξής: Μπορεί να διατηρηθεί ο χρόνος αναζήτησης σε ένα δυαδικό δένδρο λογαριθμικός αλλά η αποκατάσταση της ισορροπίας να γίνεται μετά από μία εισαγωγή, και όχι μετά από μία διαγραφή; Πριν απαντήσουμε σε αυτή την ερώτηση θα πρέπει να αναρωτηθούμε: Λογαριθμικός σε ποια παράμετρο;

Εάν δεν υπάρξει καμία αποκατάσταση της ισορροπίας μετά από την πράξη της διαγραφής τότε το δέντρο μπορεί να έχει αυθαίρετη δομή, με αποτέλεσμα ο χρόνος αναζήτησης θα μπορούσε να γίνει $\Theta(n)$. Για μια τέτοια εξέλιξη μπορεί να χρειαστούν πολλές διαγραφές, και είναι πιθανόν το ύψος των δέντρων, και ως εκ τούτου ο χρόνος αναζήτησης, θα μπορούσε να παραμείνει λογαριθμικός στο m, όπου m είναι ο αριθμός των εισαγωγών. Σε αυτό το σημείο εισάγουμε ένα νέο είδος δυαδικού δέντρου, το δέντρο ***ravl***, το οποίο επαναζυγίζεται μετά από εισαγωγές , και όχι μετά από διαγραφές. Το ύψος ενός δένδρου ***ravl*** είναι στο μέγιστο $log_{\varphi}m$, όπου φ είναι η χρυσή αναλογία. Δίχως την πράξη της διαγραφής ένα δένδρο ***ravl*** είναι ένα δένδρο ***AVL***. Ο υπολογιστικός χρόνος αποκατάστασης της ισορροπίας ανά εισαγωγή είναι $O(1)$ και εμφανίζεται βαθιά μέσα στο δέντρο. Τα αποτελέσματα μας ισχύουν για την αποκατάσταση της ισορροπίας από κάτω προς τα πάνω τα οποία επεκτείνουμε σε αποκατάσταση της ισορροπίας από πάνω προς τα κάτω με πεπερασμένο *look-ahead.*

Το τίμημα που πληρώνουμε για τα αποτελέσματά μας είναι ότι κάθε κόμβος σε ένα δυαδικό δένδρο πρέπει να αποθηκεύει *lg lgm + 1 bit balance information*. (ή *lg lg n + O(1)* με περιοδική ισοσκέλιση).

5.2 Ορολογία δένδρου

Ένα δυαδικό δένδρο αναζήτησης είναι ένα διατεταγμένο δυναμικό δένδρο στο οποίο κάθε κόμβος ***x*** έχει ένα αριστερό παιδί ***left(x)*** και ένα δεξί παιδί ***right(x)*** καθένα από τα οποία ή και τα δύο μπορεί να λείπουν *(missing nodes).* Κάθε κόμβος είναι ο πατέρας των παιδιών του. Δείχνουμε τον πατέρα ενός κόμβου ***x*** ως ***p(x).*** Η ρίζα είναι ο μόνος κόμβος χωρίς πατέρα. Φύλλο είναι ένας κόμβος με τα δύο παιδιά του να λείπουν. Η σχέση προγόνου είναι το ανακλαστικό μεταβατικό σύνολο της σχέσης γονέα. Αντίστοιχα ισχύει για την σχέση απογόνου και παιδιού.

Το ενδιαφέρον μας στρέφεται στα δυαδικά δένδρα αναζητήσεως. Ένα δυαδικό δέντρο αναζήτησης αποθηκεύει ένα *σύνολο στοιχείων*, κάθε ένα από το οποίο επιλέγει ένα κλειδί από ένα συνολικά διαταγμένο σύνολο. Σε ένα ***εσωτερικό(internal)*** δυαδικό δέντρο αναζήτησης, κάθε κόμβος είναι ένα στοιχείο και τα στοιχεία τακτοποιούνται σε συμμετρική τάξη. Το κλειδί ενός κόμβου ***x*** είναι μεγαλύτερο, αντίστοιχα λιγότερο από τα στοιχεία του αριστερού, δεξιού υπόδενδρου, αντίστοιχα. Κάθε σύγκριση κλειδιού είναι ένα *βήμα* της αναζήτησης. Ο *τρέχων κόμβος* είναι αυτός του οποίου το κλειδί συγκρίνεται με το κλειδί αναζήτησης. Η αναζήτηση είτε εντοπίζει το επιθυμητό στοιχείο είτε φθάνει σε έναν *null(missing node)* κόμβο.

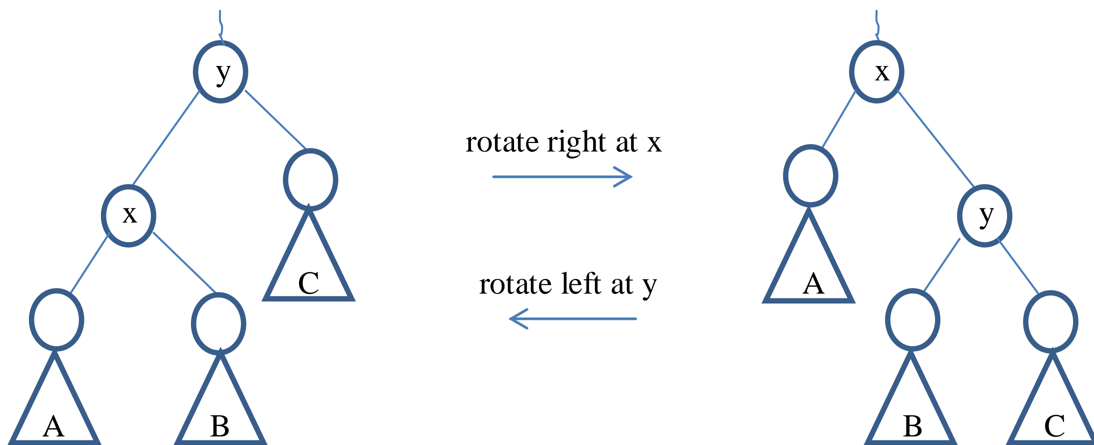
Ένα εναλλακτικό είδος δέντρου αναζήτησης είναι ένα ***εξωτερικό(external)*** δυαδικό δέντρο αναζήτησης*.* Οι εξωτερικοί κόμβοι είναι τα στοιχεία, οι εσωτερικοί κόμβοι περιέχουν το κλειδί δίχως κανένα στοιχείο, και όλα τα κλειδιά βρίσκονται σε συμμετρική τάξη. Με το όρο δυαδικό δέντρο εννοούμε ένα εσωτερικό δυαδικό δέντρο αναζήτησης. Τα αποτελέσματά μας επεκτείνονται στα εξωτερικά δυαδικά δέντρα αναζήτησης και σε άλλες δυαδικές δομές δεδομένων δέντρων.

5.3 Χαλαρά(relaxed) Δένδρα AVL

Ένα *ταξινομημένο δυαδικό δέντρο* είναι ένα δυαδικό δέντρο στο οποίο κάθε κόμβος ***x*** έχει έναν ακέραιο αριθμό *τάξης(rank)* ***r(χ)***. Οι *null* κόμβοι έχουν τάξη - 1. Η *διαφορά τάξης(rank)* ενός κόμβου ***x*** με το γονέα **p(χ)** είναι ***r(p(χ)) - r(χ).*** Ένα *i-child* είναι ένας κόμβος με διαφορά τάξης *i.* Ένας *i,j-node* είναι ένας κόμβος του οποίου τα παιδιά έχουν διαφορά τάξης *i* και *j.* Ένα *δέντρο **AVL*** είναι ένα ταξινομημένο δυαδικό δέντρο στο οποίο κάθε κόμβος είναι ένας *1,1-node* ή ένας *1,2- node.*

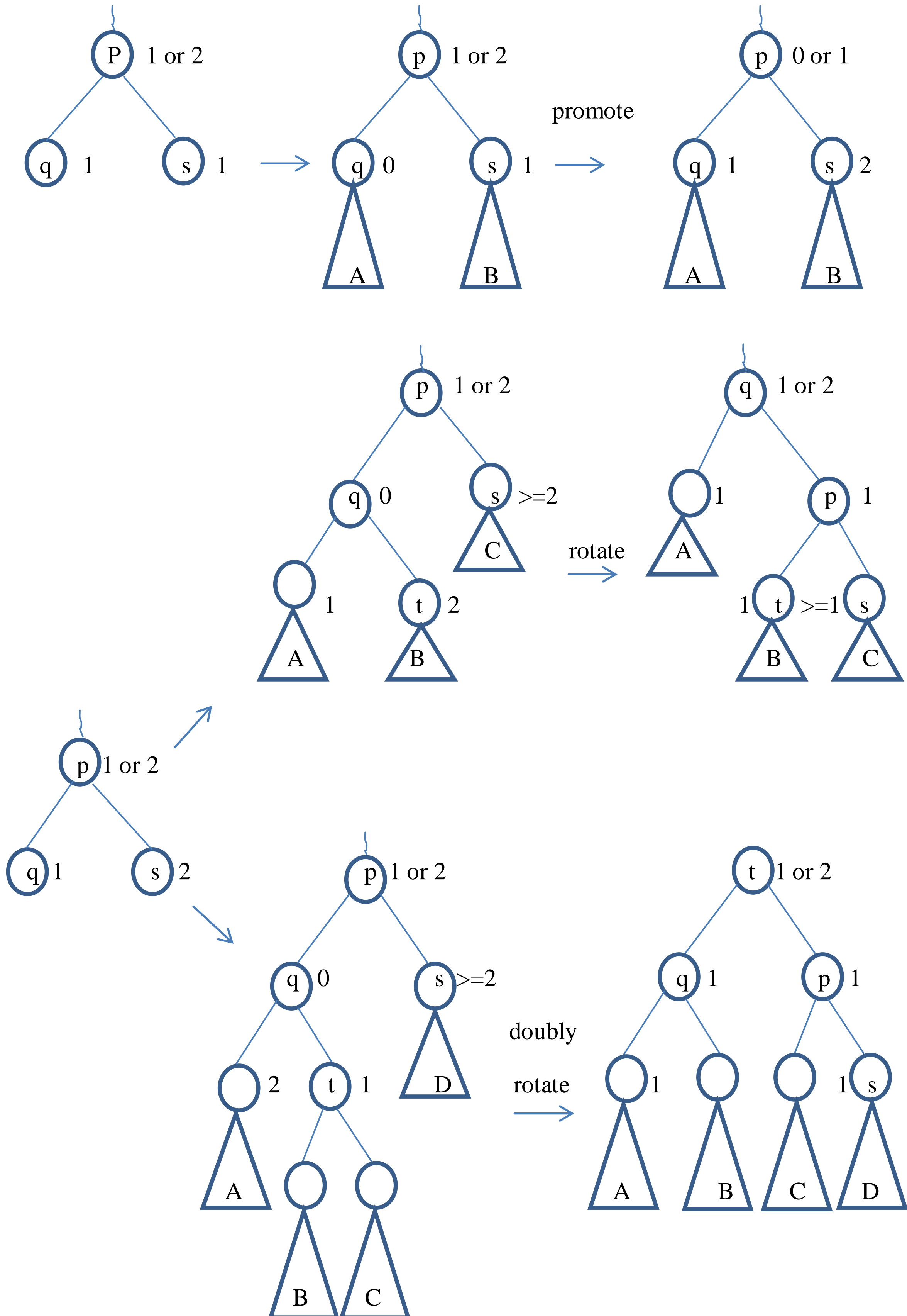
Τα φύλλα ενός δέντρου AVL είναι *1,1-node* τάξης μηδέν. Ένα *χαλαρό δέντρο AVL*, ή δέντρο ***ravl***, είναι ένα ταξινομημένο δυαδικό δέντρο που υπακούει στον ακόλουθο *κανόνα*: κάθε διαφορά τάξης*(rank)* είναι θετική.

Θεωρούμε δέντρα ***ravl***, τα δέντρα που χτίζονται από ένα κενό δέντρο μέσω ακολουθίας ανακατεμένων εισαγωγών φύλλων και διαγραφών αυθαίρετων κόμβων. Ένα νέο φύλλο ***q*** αντικαθιστά έναν null κόμβο και έχει τάξη μηδέν. Εάν ο γονέας ***p*** του νέου φύλλου ήταν ο ίδιος ένα φύλλο πριν από την εισαγωγή, το νέο φύλλο είναι *0-child* και παραβιάζει τον κανόνα τάξης*(rank)*. Αποκαθιστάμε τον κανόνα τάξης*(rank)* με την ***προώθηση(promoting)*** και τον ***υποβιβασμό(demoting)*** των κόμβων και κάνοντας περιστροφές. Μια ***προώθηση(promotion)*** αυξάνει την τάξη*(rank)* ενός κόμβου κατά ένα, ένας ***υποβιβασμός(demotion)*** την μειώνει κατά ένα. Μια ***περιστροφή(rotation)*** σε ένα αριστερό παιδί ***x*** με το γονέα **y** κάνει **y** το δεξί παιδί του ***x*** διατηρώντας τη συμμετρική τάξη. Μια ***περιστροφή(rotation)*** σε ένα δεξί παιδί είναι συμμετρική. (Βλέπε σχήμα 5.1)



Σχήμα 5.1: Περιστροφή. Τα τρίγωνα υποδουλώνουν υπόδεντρα.

Για να αποκαταστήσουμε τον κανόνα τάξης, αφήνουμε το ***q*** και το ***p*** να είναι το νέο φύλλο και ο γονέας του, αντίστοιχα. Το ***p*** είναι *null* εάν το ***q*** είναι η ρίζα. Επαναλαμβάνουμε το ακόλουθο βήμα έως ότου εμφανίζεται μια περίπτωση εκτός από μια προώθηση. (Βλέπε Σχήμα 5.2)



Σχήμα 5.2: Επαναζύγιση μετά από εισαγωγή. Οι αριθμοί έχουν διαφορά τάξης. Η πρώτη περίπτωση είναι μη τερματική.

Επαναζυγιστικά βήματα στον κόμβο q

Stop: ο κόμβος p είναι null ή ο q δεν είναι 0-child. **Stop**

Στις υπόλοιπες περιπτώσεις ο q είναι 0-child. Έστω ότι ο κόμβος s είναι αδελφός του q , ο οποίος μπορεί να είναι *missing node.*

Promote: Ο κόμβος s είναι ένα 1-child. Κάνουμε promote στον p. Πλέον ο p είναι 1,2-node, αλλά θα μπορεί να είναι και 0-child.

Στις υπόλοιπες περιπτώσεις ο s δεν είναι ένα 1-child και ο q είναι ένα 1,2-node. Υποθέτουμε ότι ο q είναι το αριστερό παιδί του p. Έστω ο t είναι το αριστερό παιδί του q, ο οποίος μπορεί να είναι ελλείπων

Rotate: Ο κόμβος t είναι ένα 2-child. Κάνουμε rotate στον q και demote στον p. Αυτό επιδιορθώνει την παραβίαση του rank rule. **Stop**

Double Rotate: Ο κόμβος t είναι ένα 1-child. Κάνουμε δύο φορές rotate στον t, κάνοντας τον q αριστερό του παιδί και τον p δεξί παιδί του. Κάνουμε promote στον t και demote στον p και q. Αυτό επιδιορθώνει την παραβίασει του rank rule. **Stop**

Για να διαγράψουμε ένα φύλλο σε ένα δέντρο ***ravl***, το αντικαθιστάμε από έναν null κόμβο. Για να διαγράψουμε έναν κόμβο με ένα παιδί, το αφαιρούμε και το αντικαθιστάμε από το παιδί του. Για να διαγράψουμε έναν κόμβο με δύο παιδιά, το ανταλλάσσουμε με τον προκάτοχο ή το διάδοχο της συμμετρικής τάξης του, κάνοντας το ένα φύλλο ή έναν κόμβο με ένα παιδί. Σε μια διαγραφή δεν εμφανίζονται περιστροφές, και καμία τάξη δεν αλλάζει εκτός των ανταλλαγμένων κόμβων που ανταλλάσσει η τάξη. Εφ' όσον δεν υπάρξει καμία διαγραφή, όλοι οι κόμβοι παραμένουν 1,1- ή 1,2- nodes, με αποτέλεσμα το δέντρο να παραμένει ένα δέντρο ***AVL***. Πράγματι, ο αλγόριθμος αποκατάστασης της ισορροπίας είναι ακριβώς ο αλγόριθμος αποκατάστασης της ισορροπίας από κάτω προς τα πάνω για τα δέντρα ***AVL***. Οι διαγραφές μπορούν να δημιουργήσουν τους κόμβους αυθαίρετης θετικής διαφοράς τάξης, και έτσι μπορούν να δημιουργήσουν δέντρα που δεν είναι δέντρα ***AVL***.

Αναπαριστάμε ένα δέντρο ***ravl*** αποθηκεύοντας με κάθε κόμβο την τάξη και τους δείκτες του στα αριστερά και δεξιά παιδιά του. Αλλά σε ένα δέντρο ***ravl***, οι διαφορές τάξεις μπορούν να γίνουν αυθαίρετα μεγάλες, και η αποθήκευση των

τάξεων με μορφή διαφοράς δεν προσφέρει κανένα πλεονέκτημα και τουλάχιστον ένα μειονέκτημα. Κατά συνέπεια αποθηκεύουμε τις τάξεις εμφανώς.

Η επαναζύγιση απαιτεί ***traversal*** της πορείας από το νέο φύλλο προς τη ρίζα. Υπάρχουν διάφοροι τρόποι να υποστηριχθεί αυτό:

- Ένας είναι να προστεθούν οι δείκτες γονέων, ή να χρησιμοποιηθεί μια άλλη αναπαράσταση που υποστηρίζει τη γονική πρόσβαση
- Ένας άλλος είναι να αποθηκευτεί η πορεία πρόσβασης κατά τη διάρκεια της αναζήτησης από τη ρίζα για τη θέση της εισαγωγής, είτε σε μια βοηθητική στήλη είτε αντιστρέφοντας τους δείκτες των παιδιών κατά μήκος της πορείας αναζήτησης.
- Ένας τρίτος είναι να διατηρηθεί ένας τελικός κενός κόμβος κατά τη διάρκεια της αναζήτησης της θέσης εισαγωγής.

Αποκαθιστούμε τον τελικό κενό κόμβο ώστε να είναι η ρίζα και το αλλάζουμε στον γονέα του τρέχοντος κόμβου της αναζήτησης κάθε φορά που ο τρέχων κόμβος δεν είναι ούτε 1-child ούτε ένας 1,1-node. Μόλις φθάσει η αναζήτηση στο κατώτατο σημείο του δέντρου, αποκαθιστούμε την ισορροπία από πάνω προς τα κάτω από τον τελικό κενό κόμβο στο νέο φύλλο.

5.4 Από κάτω προς τα πάνω επαναζύγιση

Για να επιτύχουμε καλύτερα όρια για την επαναζύγιση, και για να περιοριστεί το ύψος του δέντρου, χρησιμοποιούμε την δυναμική μέθοδο της *υπολογιστικής ανάλυσης*. Σε κάθε κατάσταση της δομής δεδομένων ορίζουμε μια μη αρνητική δυναμική, μηδέν για μια κενή δομή.

Καθορίζουμε την υπολογιστική τιμή μιας λειτουργίας να είναι η πραγματική τιμή της συν την καθαρή αύξηση στη δυναμική που προκαλεί. Κατόπιν για οποιαδήποτε ακολουθία λειτουργιών σε μια αρχικά κενή δομή, η συνολική υπολογιστική τιμή των λειτουργιών είναι ένα ανώτερο όριο στη συνολική πραγματική τιμή τους.

Ορίζουμε ως δυναμική του δέντρου το σύνολο των δυναμικών των κόμβων. Ορίζουμε την τιμή της επαναζύγισης ίσο με ένα. Μια διαγραφή δεν μπορεί να αυξήσει τη δυναμική δεδομένου ωσότου να μπορέσει να δημιουργήσει ένα *0-child* ή έναν *1,1-node*. Παρακάτω εξετάζουμε μία εισαγωγή. Η προσθήκη ενός νέου φύλλου αυξάνει τη δυναμική το πολύ κατά ένα και έχει θετική τάξη*(rank)*.

Ένα βήμα προώθησης(*promotion*) που ακολουθείται από μια διακοπή δεν αυξάνει τη δυναμική αλλά μειώνει τη δυναμική του προαχθέντος κόμβου κατά ένα και μπορεί να κάνει το γονέα του έναν *1,1-node*, που αυξάνει τη δυναμική κατά ένα. Μια διακοπή δεν έχει καμία επίδραση στη δυναμική. Μια περιστροφή ή μια διπλή περιστροφή μπορεί να αυξήσει τη δυναμική το πολύ κατά δύο, με τη δημιουργία δύο νέων *1,1-node* θετικής τάξης*(rank)*. Καταλήγουμε στο συμπέρασμα ότι η υπολογιστική τιμή μιας εισαγωγής είναι το πολύ τέσσερα. Η προσθήκη του νέου φύλλου αυξάνει τη δυναμική κατά ένα. Εάν το τελευταίο βήμα είναι μια διακοπή(*stop*) αυτή και το επόμενο-τελευταίο βήμα έχουν μια υπολογιστική τιμή το πολύ δύο. Εάν το τελευταίο βήμα είναι μια περιστροφή(*rotation*) ή διπλή περιστροφή(*double rotation*) έχουν μια υπολογιστική τιμή το πολύ τρία. Κάθε βήμα προώθησης(*promotion*) έχει μια υπολογιστική τιμή μηδενική εκτός αν ακολουθείται από μια διακοπή(*stop*).

Θεώρημα 4.1: Αρχίζοντας από ένα κενό δέντρο ***ravl***, μια ακολουθία *m* εισαγωγών με την από κάτω προς τα πάνω επαναζύγιση, που αναμιγνύεται με τις αυθαίρετες διαγραφές απαιτεί το πολύ *4m* βήματα επαναζύγισης.

Θεώρημα 4.2: Αρχίζοντας από ένα αυθαίρετο δέντρο ***ravl*** που περιέχει *g* 1,1-κόμβους θετικής τάξης, μια ακολουθία *m* εισαγωγών με την από κάτω προς τα πάνω επαναζύγιση που αναμιγνύεται με τις αυθαίρετες διαγραφές απαιτεί το πολύ *4m+g* βήματα επαναζύγισης.

Ένα δέντρο ***ravl*** που χτίζεται από ένα κενό δέντρο έχει ύψος λογαριθμικό στον αριθμό των εισαγωγών, ακόμα κι αν οι διαγραφές είναι ανακατεμένες αυθαίρετα.

Θεώρημα 4.3: Εάν ένα δέντρο ***ravl*** χτίζεται από ένα κενό δέντρο μέσω μιας ακολουθίας *m* εισαγωγών με την από κάτω προς τα πάνω επαναζύγιση αναμιγμένη με αυθαίρετες διαγραφές, $m \geq F_{h+3} - 1 \geq \varphi^h$. Κατά συνέπεια $h \leq \log_{\varphi} m$.

Θεώρημα 4.4: Αρχίζοντας από ένα αρχικά κενό δέντρο, μια ακολουθία εισαγωγών με την από κάτω προς τα πάνω επαναζύγιση αναμεμιγμένη με αυθαίρετες διαγραφές απαιτείται το πολύ $(m-1)/F_k \leq (m-1)/\varphi^{k-2}$ βήματα επαναζύγισης της τάξης *k*, για οποιοδήποτε $k > 0$.

5.5 Από πάνω προς τα κάτω επαναζύγιση

Αντί να επαναζυγίσουμε από κάτω προς τα πάνω αφότου προστεθεί ένα νέο φύλλο, μπορούμε να επαναζυγίσουμε από πάνω προς τα κάτω προτού προστεθεί το νέο φύλλο.

Με αυτόν τον τρόπο μπορούμε να κρατήσουμε τον τελικό κόμβο μέσα στους κόμβους *O(1)* του τρέχοντος κόμβου αναζήτησης. Η ιδέα είναι να αναγκαστεί ένα *reset* του τελικού κόμβου μετά από έναν επαρκώς μεγάλο αριθμό βημάτων αναζήτησης που δεν κάνουν κανένα *reset*. Μια αναστοιχειοθέτηση(*reset*) εμφανίζεται στο επόμενο βήμα αναζήτησης εκτός αν ο τρέχων κόμβος είναι ένας *1,1-node*. Εάν ο τρέχων κόμβος και ο γονέας του είναι *1,1-node*, μπορούμε να αναγκάσουμε μια αναστοιχειοθέτηση με την *προώθηση(promoting)* του τρέχοντος κόμβου και την επαναζύγιση του *trailing node* από πάνω προς τα κάτω.

Οι διαγραφές δεν αυξάνουν τη δυναμική. Η προσθήκη ενός φύλλου αυξάνει τη δυνατότητα το πολύ κατά τέσσερα. Η επαναζύγιση αφότου προστεθεί ένα φύλλο αποτελείται από μια διακοπή, ή από μια, δύο, ή τρεις *προωθήσεις(promotions)* που ακολουθούνται από ένα τελικό βήμα. Η αύξηση δυναμικής παίρνει σε κάθε περίπτωση τιμές από -4 έως 4.

Θεώρημα 5.1: Αρχίζοντας από ένα αυθαίρετο δέντρο που περιέχει *g* 1,1- κόμβους θετικής τάξης, μια ακολουθία *m* εισαγωγών με την από πάνω προς τα κάτω επαναζύγιση αναμιγμένη με αυθαίρετες διαγραφές απαιτεί το πολύ *10m+4g* βήματα επαναζύγισης.

Θεώρημα 5.2: Ένα δέντρο **ravl** που χτίζεται από ένα κενό δέντρο μέσω μιας ακολουθίας *m* εισαγωγών με την από πάνω προς τα κάτω επαναζύγιση αναμιγμένη με αυθαίρετες διαγραφές έχει ύψος το πολύ $log_b m + O(1)$, όπου b = 1.325+ είναι η μεγαλύτερη πραγματική ρίζα του b^3 - b -1.

Θεώρημα 5.3: Αρχίζοντας από ένα αρχικά κενό δέντρο, μια ακολουθία *m* εισαγωγών με την από πάνω προς τα κάτω επαναζύγιση αναμιγμένη με αυθαίρετες διαγραφές κάνει $O(m/b^k)$ βήματα επαναζύγισης της τάξης *k* για οποιοδήποτε $k \geq 0$, όπου το *b* έχει την ίδια αξία όπως στο θεώρημα 5.2.

Τελικά βγάζουμε το συμπέρασμα ότι οι διαγραφές δεν αυξάνουν τη δυναμική. Η προσθήκη ενός νέου φύλλου αυξάνει τη δυναμική κατά *O(1).* Ούτε η αναστοιχειοθέτηση ούτε η επαναζύγιση αφότου προστεθεί ένα νέο φύλλο δεν μπορεί να αυξήσει τη δυναμική.

5.6 Επαναζύγιση ενός δέντρου

Για να επαναζυγίσουμε το δέντρο, εισάγουμε ένα νέο δέντρο σε κενό. Κατόπιν διερχόμαστε το παλιό δέντρο με συμμετρική σειρά, διαγράφοντας κάθε επισκεπτόμενο κόμβο και εισάγοντας τον στο νέο δέντρο. Η διέλευση του παλιού δέντρου απαιτεί χρόνο *O(n).* Στο τέλος αποθηκεύουμε τους κόμβους στη δεξιά στήλη του σε έναν σωρό, με τον κατώτατο κόμβο στην κορυφή.

Για να αποφασίσουμε πότε να επαναζυγίσουμε το δέντρο, παρακολουθούμε το n και την τάξη r της ρίζας. Εάν χρησιμοποιούμε την από κάτω προς τα πάνω επαναζύγιση, ανασυγκροτούμε το δέντρο όταν $r > \log_{\varphi} n + c$, όπου το c είναι μια μικρή θετική σταθερά. Εάν χρησιμοποιούμε την από πάνω προς τα κάτω επαναζύγιση, ανασυγκροτούμε το δέντρο όταν $r > \log_{\varphi} n + c,$ όπου το b είναι η βάση στο θεώρημα 5.2 και το c είναι μεγαλύτερο από την πρόσθετη σταθερά στο θεώρημα 5.2.

Μπορούμε επίσης να καταστήσουμε την επαναζύγιση *incremental.* Παραδείγματος χάριν, μπορούμε να αρχίσουμε την επαναζύγιση εφόσον παραβιαστεί το όριο του ύψους του και να μετακινήσουμε δύο κόμβους από το παλαιό προς το νέο δέντρο μετά από κάθε εισαγωγή ή διαγραφή.

Εάν το δέντρο επαναζυγιστή αυξητικά ή μονομιάς, το ύψος του δέντρου είναι το πολύ $\log_{\varphi} n + O(1)$ με την από κάτω προς τα πάνω επαναζύγιση ή $\log_{\varphi} n + O(1)$ με την από πάνω προς τα κάτω επαναζύγιση.

5.7 Καλές και κακές εναλλακτικές λύσεις

Όλα τα αποτελέσματα που έχουμε παρουσιάσει στηρίζονται σε τρεις ιδιότητες:

- Εάν ένα νέο φύλλο πριν την επαναζύγιση είναι *0-child*, έχει τάξη μηδέν.
- Εάν ο γονέας ενός νέου φύλλου είναι ένας 1,1-κόμβος, έχει τάξη 1, και
- Οι διαγραφές δεν αυξάνουν οποιαδήποτε τάξη.

Τα παραδείγματα υποδεικνύουν ότι η επίτευξη ενός λογαρίθμου συνδεδεμένο με το ύψος στον αριθμό των εισαγωγών, χρησιμοποιώντας ένα *bit* ανά *non-root* κόμβο, απαιτεί μια μέθοδο διαγραφής που δεν αυξάνει οποιαδήποτε τάξη.

5.8 Επαναζύγιση μετά από διαγραφή ή όχι;

Η διαγραφή σε ένα δένδρο ***ravl*** είναι πολύ πιο απλούστερη σε αντίθεση με τα άλλα δένδρα. Τα δέντρα ***ravl*** απαιτούν *O(log log m) balance bits* ανά κόμβο αντί του ενός *bit* ανά κόμβο που απαιτείται από ένα ***AVL***, *rank-balanced,* και *red-black* δέντρα. Το ύψος τους είναι το πολύ $log_{\varphi}\, m$, έναντι $log_{\varphi}\, n$ για τα δέντρα AVL, *2lg n* για τα *red-black* δέντρα, και $min\{log_{\varphi}m,\ 2lg\ n\}$ για τα *rank-balanced* δέντρα. Με την περιοδική επαναζύγιση, με τιμή *O(1)* ανά *update*, μπορεί να διατηρηθεί ένα ύψος ορίου $log_{\varphi}n + O(1)$, και απαιτούνται μόνο *O(log log n) balance bits* ανά κόμβο. Η επαναζύγιση μετά από εισαγωγή απαιτεί στην χειρότερη περίπτωση *O(1)* χρόνο περιστροφών και *O(1)* υπολογιστικό χρόνο, και οι κόμβοι επηρεάζονται από την επαναζύγιση με μια συχνότητα εκθετικά μικρή στο ύψος τους.

5.9 Ανάλυση Κώδικα *RAVL*

Στον κώδικα του παραρτήματος υλοποιούμε τα ***ravl trees.*** Παρακάτω γίνεται αναφορά στην βασική δομή και στις βασικές συναρτήσεις για την καλύτερη κατανόηση του αναγνώστη.

Η βασική δομή είναι:

```
class ravl{
public:
        int countSingleLeft;
        int countSingleRight;
        int countDoubleL_R;
        int countDoubleR_L;
        int countNoRebalance;
        bool verbose;
```

```
        node *root;
        ravl(bool verbose);
        ~ravl();
        void printTree(bool inFile);
        int countNodes();
        int insertNode(int newNode);
        int deleteNode(int oldNode);
        double diffclock(clock_t clock1,clock_t clock2);
        stack<node *> travPath;
        ofstream treeLog;

};
```

Οι βασικές συναρτήσεις που περιλαμβάνονται είναι οι εξής:

ravl :: printTree(bool inFile) /*Χρησιμοποιείται για την εκτύπωση του δένδρου.*/

ravl :: insertNode(int newData) /* Χρησιμοποιείται για την εισαγωγή δεδομένων σε ένα δέντρο. Εάν επιτύχει επιστρέφει τιμή 1 διαφορετικά 0.*/

ravl :: deleteNode(int delData) /* Χρησιμοποιείται για τη διαγραφή ενός δέντρου . Εάν επιτύχει επιστρέφει τιμή 1 διαφορετικά 0.*/

ravl :: countNodes /*Επιστρέφει των αριθμό των δεδομένων σε ένα δένδρο.*/

Επιπλέον ορίστηκαν τρεις *global* μεταβλητές:

MAX_INIT_POW /*Ορίζει τον αρχικό αριθμό των στοιχείων που θα περιέχει το δέντρο πριν ξεκινήσει το πείραμα. */

MAX_INS_POW /*Αναφέρεται στο πλήθος των στοιχείων προς εισαγωγή και διαγραφή.*/

MAX_DEL_POW /*Αναφέρεται στο πλήθος των στοιχείων προς εισαγωγή και διαγραφή.*/

Λόγω του μεγάλου πλήθους στοιχείων που μελετώνται οι τιμές αυτών των μεταβλητών είναι εκθετικές. Δηλαδή, θέτοντας παραδείγματος χάριν MAX_INS_POW = 6 συνεπάγεται ότι θα εισαχθούν 10^6 = 1.000.0000 στοιχεία.

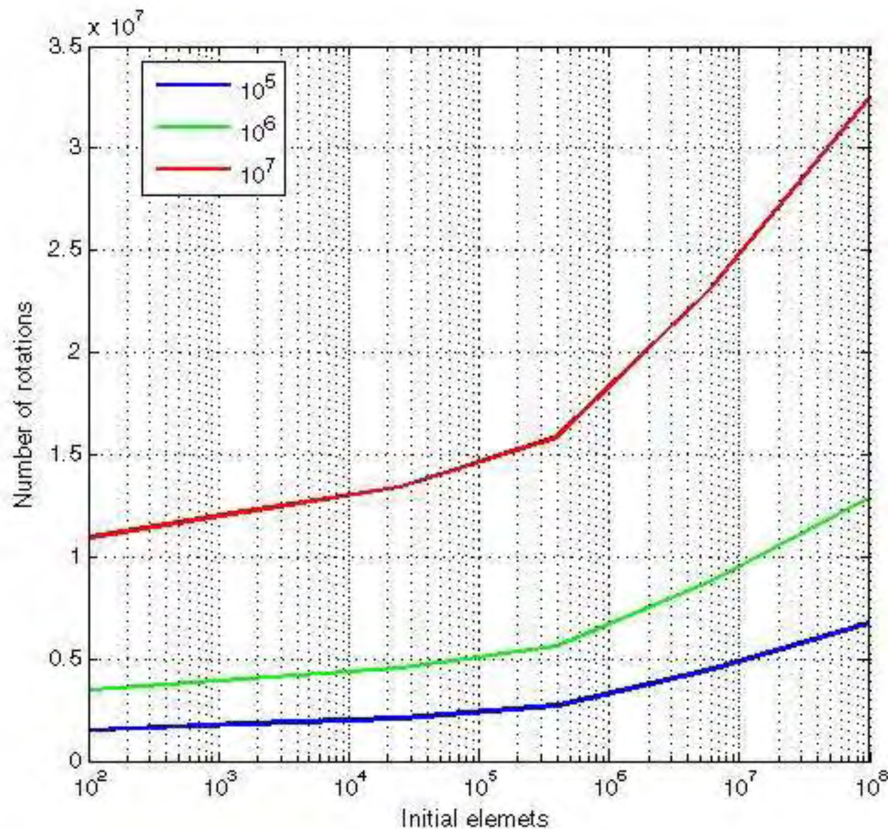
6. Πειραματικά αποτελέσματα

6.1 Συγκριτικά αποτελέσματα δένδρων AVL, RAVL

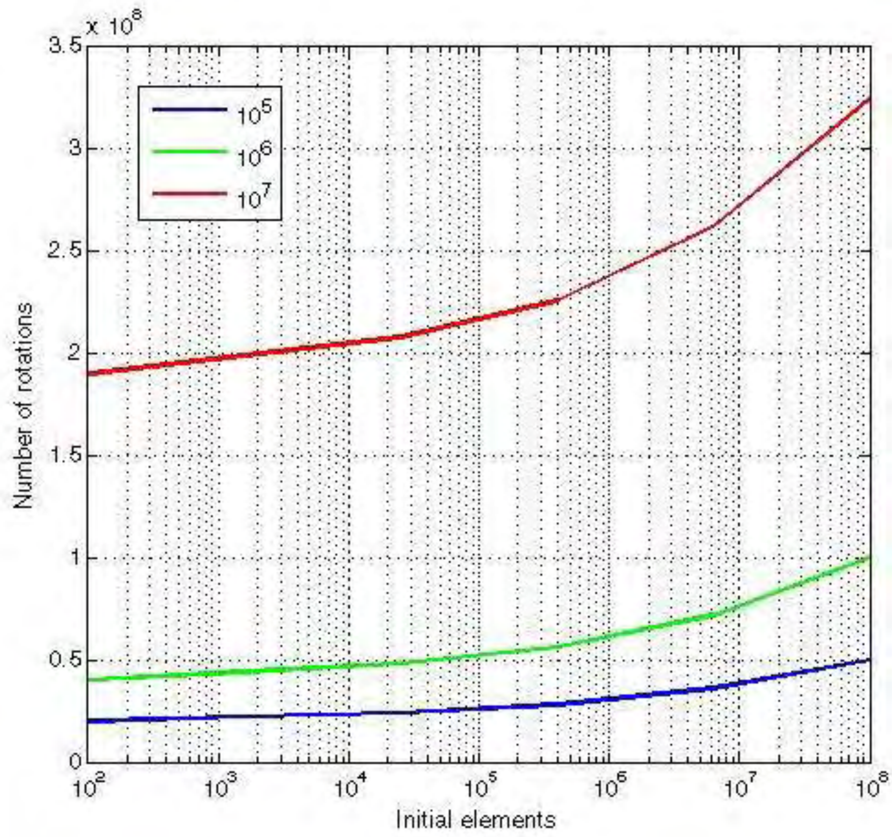
Αφού κάνουμε *compile and run* τους κώδικες του παραρτήματος δημιουργούνται τυχαίοι αριθμοί σε αρχείο τόσο στον κώδικα ***avl,*** όσο και στον κώδικα ***ravl*** αντίστοιχα. Στην συνέχεια εισάγοντας αυτούς τους αριθμούς στο εργαλείο **MATLAB** παράγονται τα παρακάτω αποτελέσματα:

- **Εισαγωγή**

Όσον αφορά τις εισαγωγές, η θεωρία (paper) προβλέπει ότι πρέπει να είναι ίδιες και στους δύο τύπους δένδρων. Δηλαδή, εάν δεν υπάρξουν διαγραφές οι δύο τύποι είναι όμοιοι. Πράγματι, αυτό αντικατοπτρίζεται πλήρως στα αποτελέσματα των προσομοιώσεων μας Σχήμα 6.1 και Σχήμα 6.2, όπου παρατηρούμε ότι το πλήθος των εισαγωγών είναι ίδιο τόσο στο RotationInsertion AVL όσο και στο RotationInsertion RAVL, στα πλαίσια πάντα του στατικού σφάλματος.



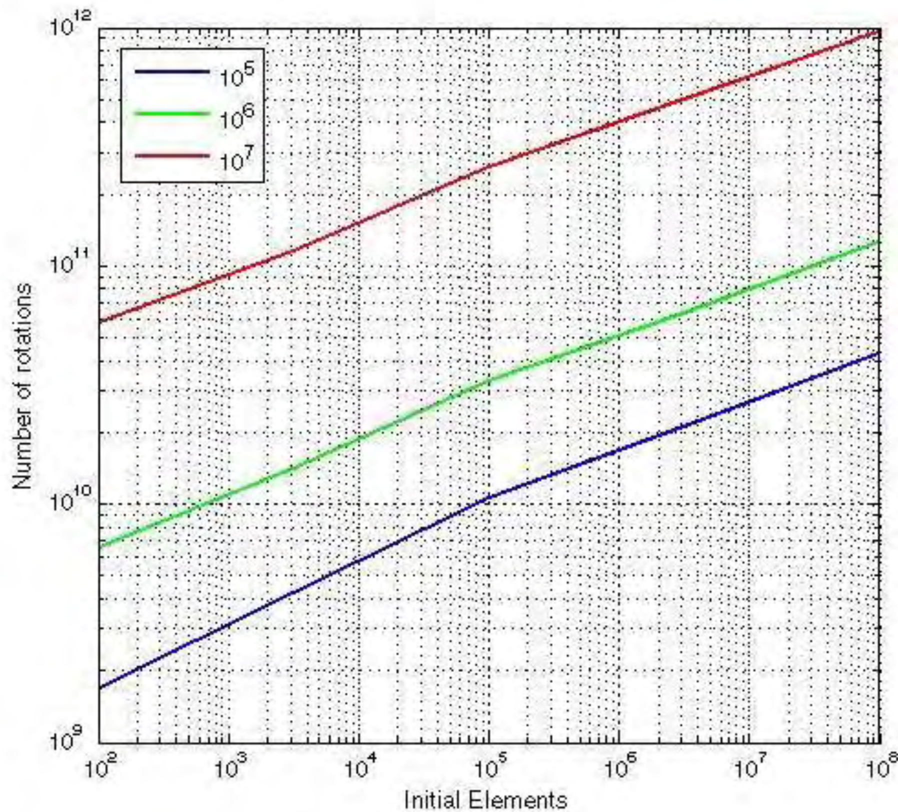
Σχήμα 6.1: RotationInsertion AVL



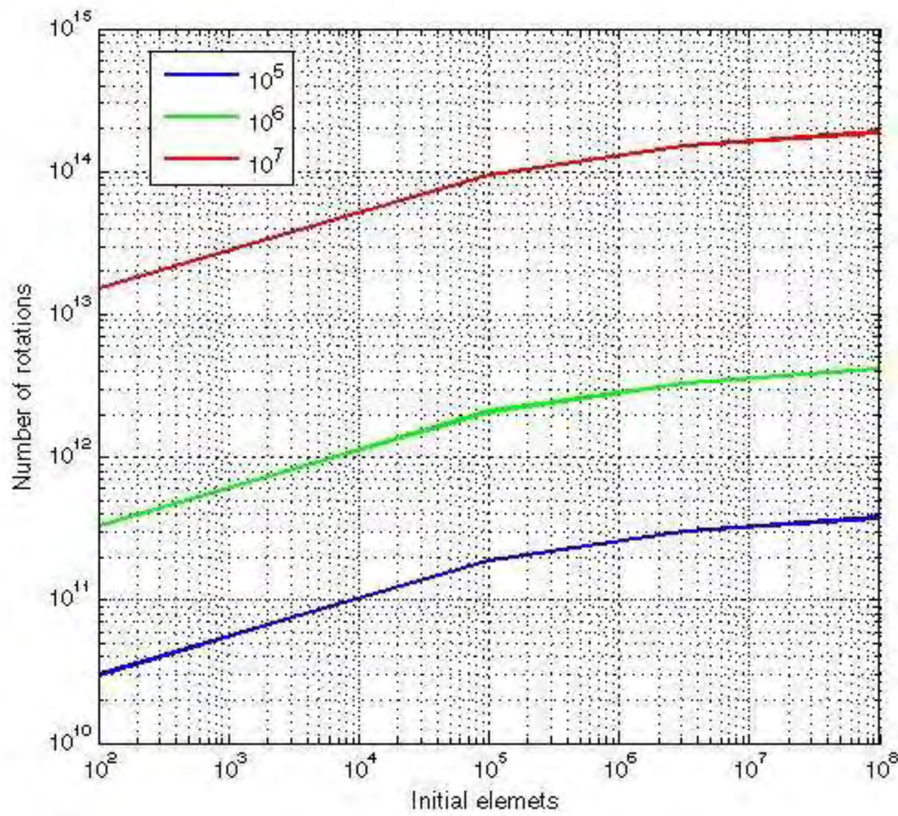
Σχήμα 6.2: RotationInsertion RAVL

- **Διαγραφή**

Όσον αφορά τις διαγραφές , η θεωρία (paper) προβλέπει ότι τα δένδρα ***ravl*** έχουν σημαντικά χαμηλότερο αριθμό περιστροφών σε κάθε περίπτωση. Πράγματι το ίδιο αποτέλεσμα λαμβάνουμε και από τις προσομοιώσεις μας Σχήμα 6.3 και Σχήμα 6.4, όπως είναι άμεσα εμφανές.



Σχήμα 6.3: RotationDeletion AVL



Σχήμα 6.4: RotationDeletion RAVL

Τέλος από τις προσομοιώσεις μας προέκυψε ότι τα κοινά δένδρα παρουσιάζουν κορεσμό, δηλαδή η κλίση της ευθείας μηδενίζεται για αρκετά μεγάλο αριθμό δεδομένων. Το σημείο κορεσμού είναι λίγο πολύ σταθερό Σχήμα 6.4 στα 10^5 στοιχεία, αλλά το ύψος της καμπύλης αυξάνεται μη γραμμικά με τον αριθμό των προς διαγραφεί στοιχείων. Αντίθετα η τάση των δένδρων ***ravl*** είναι αυξητική βλέπε Σχήμα 6.3 και δεν δείχνει σημεία κορεσμού.

Η εκτίμηση μου είναι ότι αυτό οφείλεται στη διαφορετική δομή των δένδρων ***avl***. Ωστόσο πρέπει να σημειωθεί, ότι παρά τον κορεσμό τα δένδρα ***avl*** συνεχίζουν να έχουν σημαντικά χαμηλότερο αριθμό περιστροφών.

6.2 Παρατηρήσεις

Έχουμε αποδείξει ότι ένα δυαδικό δένδρο αναζήτησης μπορεί στην χειρότερη περίπτωση να έχει λογαριθμικό χρόνο αναζήτησης δίχως την πράξη της επαναζύγισης μετά από μία διαγραφή, αλλά αυτό φαίνεται να απαιτεί αποθήκευση *Ω(log log n) balance bits* ανά κόμβο.

7. Βιβλιογραφία

1. L. A. Adamic and B. A. Huberman. Zipf's law and the Internet. Glottometrics, 3:143{150, 2002.
2. G. M. Adel'son-Vel'skii and E. M. Landis. An algo-rithm for the organization of information. Sov. Math.Dokl., 3:1259{1262, 1962.
3. A. Andersson. Balanced search trees made simple. In
WADS, volume 709, pages 60{71, 1993.
4. R. Bayer. Binary B-trees for virtual memory. In SIGFIDET, pages 219{235, 1971.
5. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Inf., 1:290{306,1972.
6. J. B. Estoup. Gammes stenographiques., 1916.
7. C. C. Foster. A study of AVL trees. Technical Report GER-12158, Goodyear Aerospace Corp., 1965.
8. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. Algorithmica, 1(1):111{129, 1986.
9. L. J. Guibas and R. Sedgewick. A dichromatic frame-work for balanced trees. In FOCS, pages 8{21, 1978.

8.Παράρτημα

Στο παράρτημα παρατίθενται οι κώδικες σε γλώσσα προγραμματισμού C για τα δένδρα ***avl,*** και C++ για τα δένδρα ***ravl.***

Κώδικας *avl*:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>


long int rotcount=0;


typedef struct AVLnode* avltree;


struct AVLnode

{

   int data;          //the tree's data. Can be changed at will

   avltree  left;     //left node (child)

   avltree  right;  //right node

   int height;        //the height of the tree


};


static int height( avltree t )

{

        if( t == NULL )

              return -1;

           else

              return t->height;

        }
```

```
/* return maximum of two integers */
static int max( int a, int b )
{
        return a > b ? a : b;
}

/* Rotate with left*/
static avltree rotateL( avltree t )
{
      rotcount++;

      avltree tmp;

      tmp = t->left;
      t->left = tmp->right;
      tmp->right = t;

      t->height = max( height( t->left ), height( t->right ) ) + 1;
      tmp->height = max( height( tmp->left ), t->height ) + 1;

      return tmp;
}

/* Rotate with right */
static avltree  rotateR( avltree t )
{
      rotcount++;
```

```
        avltree tmp;
        tmp = t->right;
        t->right = tmp->left;
        tmp->left = t;

        t->height = max( height( t->left ), height( t->right ) ) + 1;
        tmp->height = max( height( tmp->right ), t->height ) + 1;

        return tmp;
}

/* Double rotate with left */
static avltree  drotateL( avltree t )
{
         rotcount++;

        t->left = rotateR( t->left );

        return rotateL( t );
}

/* Double rotate with right */
static avltree drotateR( avltree t)
{
            rotcount++;

        t->right = rotateL( t->right );

        return rotateR( t );
}
```

```
/**
* Deletes the entire tree and frees up the memory
* Returns a NULL pointer
*/
avltree delete_tree(avltree tree)
{
  if( tree != NULL )
  {
      delete_tree( tree->left );
      delete_tree( tree->right );
      tree->left=NULL;
      tree->right=NULL;
      free(tree);
  }

  return NULL;
}

/**
* Creates a new AVL tree and it returns it
*/
avltree create_tree(avltree tree, int d)
{
  if( tree == NULL )
  {
        tree= malloc( sizeof( struct AVLnode ) );
        if( tree==NULL )
        {
              printf( "ERROR: NOT ENOUGH MEMORY" );
               return NULL;
```

```
        }
        else
        {
            tree-> data = d;
            tree-> left = NULL;
            tree-> right = NULL;
            tree-> height=0;

            return tree;
        }
    }
    else
    {
        printf("Tree already exists!");
        return tree;
    }
}

/**
* Inserts data to the tree
* If tree is empty (doesn't exist) it creates it
* Otherwise it inserts a child at the corresponding location
*/
avltree insert_data(avltree tree, int d)
{
    if( tree == NULL )
    {
        tree=create_tree(tree, d);
    }
```

```
  else if( d < tree->data )
  {
     tree->left = insert_data( tree->left, d );
     if( height( tree->left ) - height( tree->right ) == 2 )
                  if( d < tree->left->data )
                        tree = rotateL( tree );
                     else
                        tree = drotateL( tree );


  }
  else if( d > tree->data )
  {
     tree->right = insert_data( tree->right, d);
      if( height( tree->right ) - height( tree->left ) == 2 )
                        if( d > tree->right->data )
                           tree = rotateR( tree );
                        else
                           tree = drotateR( tree );
  }
  else
     printf("Data already exists in tree");



     tree->height = max( height( tree->left ), height( tree->right ) ) + 1;
     return tree;
}
```

```
/**
* Searches the tree for the data d.
* Returns the node (the location) where the data were found.
* If not found returns NULL
*/
avltree search_and_find(int d, avltree tree)
{
  if( tree== NULL )
  {
     printf("Data not found! Please retry with different data");
     return NULL;
  }

  if( d < tree->data )
     return search_and_find(d, tree->left );

  else if( d > tree->data )
        return search_and_find( d, tree->right );

  else
     return tree;
}

/* Get the minimum of the tree to be used for the delete function */
avltree getmin( avltree t )
{
        if( t == NULL )
                return NULL;
        else
```

```
                if( t->left == NULL )
                return t;
        else
                return getmin( t->left );
}

/**
* Deletes data from tree.
* It deletes the node which holds the input data.
*/
avltree delete_node(avltree tree, int d)
{
  avltree temp;

  // Check if empty
  if( tree== NULL )
  {
     return NULL;
  }

   else if( d < tree->data ){
      tree->left = delete_node( tree->left, d );
      if( height( tree->left ) - height( tree->right ) == -2 )
         if( height(tree->right->right) > height(tree->right->left) )
            tree = rotateR( tree );
         else
            tree = drotateR( tree );
   }
   else if( d > tree->data ){
      tree->right = delete_node( tree->right, d );
```

```
    if( height( tree->right ) - height( tree->left ) == -2 )
        if( height(tree->left->left) > height(tree->left->right) )
            tree = rotateL( tree );
        else
            tree = drotateL( tree );
}
else if( tree->left && tree->right ){ // both children exist
  temp=getmin( tree->right );
  tree->data = temp->data;
  tree->right = delete_node( tree->right, tree->data );
  if( height( tree->right ) - height( tree->left ) == -2 )
      if( height(tree->left->left) > height(tree->left->right) )
          tree = rotateL( tree );
      else
          tree = drotateL( tree );
}
else {
   temp=tree;
       //replace with existing
       if (tree->left )
                   tree = tree->left;
       else
                   tree = tree->right;

       //free memory
   free( temp );
}

if( tree!= NULL ) //rebalance
       tree->height=max(height(tree->left), height(tree->right))+1;
```

```
    return tree;
}

/**
* Displays the data in the tree
*/
void display(avltree tree)
{
    if( tree== NULL )
    {
        return;
    }
    else
    {
        printf("%d ",tree->data);

        display(tree->left );

        display(tree->right );
    }
}

/**
* Like display() but writes data to a file
*/
void twrite(avltree tree, FILE* f)
{
    if( tree== NULL )
    {
```

```
        return;
    }
    else
    {
        fprintf(f,"%d ",tree->data);

        twrite(tree->left, f);

        twrite(tree->right, f);
    }

return;
}

int main()
{
    int i;
    clock_t start, end;

    FILE *file;

     file = fopen("file.txt","a+"); /* apend file (add text to a file or create a file if it does not exist.*/

    avltree mytree=NULL;
    mytree = create_tree(mytree,1);

    start = clock();
    for (i=2;  i<1e6; i++)
        mytree=insert_data(mytree, i);
```

```
    fprintf(file,"\n==========================\n");
    fprintf(file,"Display initial tree\n");
    fprintf(file,"==========================\n");
    //twrite(mytree, file);

    delete_node(mytree,6);
    fprintf(file,"\n==========================\n");
    fprintf(file,"Display tree after deletion\n");
    fprintf(file,"==========================\n");
    //twrite(mytree, file);

    delete_node(mytree,11);
    fprintf(file,"\n================================================\n");
    fprintf(file,"Test tree validity after deleting a non-existing element\n");
    fprintf(file,"=================================================\n");
    //twrite(mytree, file);

    mytree=delete_tree(mytree);
    fprintf(file, "\n============================================\n");
    fprintf(file,"Display tree after deleting all tree\n");
    fprintf(file,"==============================================\n");
    //twrite(mytree, file);

    end= clock();
    fprintf(file, "That took %f seconds", (end-start)/CLOCKS_PER_SEC);
    fprintf(file, "\ntotal rotations = %d", rotcount);

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Κώδικας *ravl*:

Αρχείο ravl.cpp

```
#include <stack.h>

#include <limits.h>

#include <iostream.h>

#include <fstream.h>

#include <math.h>

#include <sys/time.h>


class ravl{

public:

        int countSingleLeft;

        int countSingleRight;

        int countDoubleL_R;

        int countDoubleR_L;

        int countNoRebalance;

        bool verbose;


       node *root;

       ravl(bool verbose);

       ~ravl();

       void printTree(bool inFile);

       int countNodes();

       int insertNode(int newNode);

       int deleteNode(int oldNode);

       double diffclock(clock_t clock1,clock_t clock2);

       stack<node *> travPath;

       ofstream treeLog;
```

```
};

/***

	if verbose == true, log files will be created.

	treeLog.txt will contain tree building logs.

*/

ravl :: ravl(bool verbose){

	this->root = 0;

	countSingleLeft = 0;

	countSingleRight = 0;

	countDoubleL_R = 0;

	countDoubleR_L = 0;

	countNoRebalance = 0;

	if(verbose){

		this->verbose = verbose;

		treeLog.open("treeLog.txt");

		treeLog<<"Tree building log file."<<endl;

		treeLog<<"***********************"<<endl;

	}

}

ravl :: ~ravl(){

	if(verbose)

		treeLog.close();

}

/***

	function printTree uses inorder traversal of the tree to pront it's contents

	inFile parameter informs whether you want the function to print the tree

	inside a file rather than stdout.inFile == 1 will print inside a file, inFile ==0

	will print on screen.

	ravlTree.txt will contain the inorder printout of the tree.

	results.txt will contain tree's results like max rank, tree numel etc.
```

```
*/

void ravl :: printTree(bool inFile){

        int max=INT_MIN;

        if(!inFile){

                cout << "\n******************\nTree traversal configuration : Inorder" <<endl;

                stack<node *> nodeStack;

                node *curr = this->root;

                int count = 0;


                for (;;) {

                        if (curr != NULL) {

                                nodeStack.push(curr);

                                curr = curr->leftChild;

                                continue;

                        }

                        if (nodeStack.size() == 0){

                                cout<< "MAXIMUM RANK = " << max<<endl;

                                cout<<"TREE NUMEL = " <<count<<endl;

                                return;

                        }

                curr = nodeStack.top();

                        nodeStack.pop();

        cout << "Node : [data,rank]: " << "["<<curr->data << " , "<< curr->rank <<"]"<<endl;

                        count++;

                        if(curr->rank > max)

                                max = curr->rank;


                        curr = curr->rightChild;


                }
```

```
}else{
        ofstream fout, results;
        fout.open("ravlTree.txt");
        results.open("results.txt");
        fout << "\n********************\nTree traversal configuration : Inorder" <<endl;
        stack<node *> nodeStack;
        node *curr = this->root;
        int count = 0;

        for (;;) {
                if (curr != NULL) {
                        nodeStack.push(curr);
                        curr = curr->leftChild;
                        continue;
                }
                if (nodeStack.size() == 0){

results<<"*********************************************************"<<endl;
results<<"MAXIMUM RANK = " << max<<endl;
results<<"TREE NUMEL = " <<count<<endl;
results<<"NUMBER OF SINGLE ROTATIONS = "<< countSingleLeft + countSingleRight<<endl;
results<<"OF WHICH "<<countSingleLeft<<" WERE LEFT AND " <<countSingleRight<<" WERE RIGHT ROTATIONS."<<endl;
results<<"NUMBER OF DOUBLE ROTATIONS = "<< countDoubleL_R + countDoubleR_L<<endl;
results<<"OF WHICH "<<countDoubleL_R<<" WERE L-R AND " <<countDoubleR_L<<" WERE R-L ROTATIONS."<<endl;
results<<"AT "<<countNoRebalance<<" CASES NO REBALANCE WAS NECESSARY"<<endl;
```

```
	results<<"*********************************************************"<<endl;
				return;
			}
		curr = nodeStack.top();
			nodeStack.pop();
			fout << "Node "<<count++<<": "<< "[data,rank]= " << "["<<curr->data
<< " , "<< curr->rank <<"]"<<endl;
			if(curr->rank > max)
				max = curr->rank;

			curr = curr->rightChild;

		}

		fout.close();
	}

}
/***
	function insertNode places new nodes to the tree. If this is done succesfully,
	value of one will be returned, else zero. We only care about demonstration purposes
	so only integer intake is considered.
	function configuration : non recursive.
*/
int ravl :: insertNode(int newData){
	if(verbose)
		treeLog<<"[+]Inserting node with value: "<<newData<<endl;
	if(!this->root){ // tree is empty, so root is null
		if(verbose)
```

```
                treeLog<<"   - Tree was empty, new node is now the root."<<endl;

            this->root = new node(newData);
            return 1;
        }else{ // traverse the tree looking for the father of the new leaf
            node *tempNode = this->root;
            node *newFather;
            if(verbose)
                treeLog<<"   - Traversing the tree..."<<endl;

            while(1){

                if(!tempNode){
                    break; // newFather points to the new father
                }else{

                    if(tempNode->data == newData){
                        if(verbose)
treeLog << "   - Duplicate value found, no duplicates are allowed. Aborting inserion. "<<endl;
                        while(travPath.size())
                            travPath.pop();
                        return 1;
                    }// abort insertion, no duplicates are allowed
                    if(tempNode->data > newData){

                        newFather = tempNode;
                        tempNode = tempNode->leftChild;
                    }else{

                        newFather = tempNode;
```

```
                        tempNode = tempNode->rightChild;
                    }
            }
            travPath.push(newFather);

    }

    tempNode = new node(newData);
    travPath.push(tempNode);
    if(newFather->data > newData){
            newFather->leftChild = tempNode;
    }else{
            newFather->rightChild = tempNode;
    }
    if(verbose)
    treeLog<<"- Insertion was successful. Proceeding to rebalance."<<endl;

    // after insertion comes balancing
    node *q,*p,*r; // p is the father, q is the case node and r is q's sibling
    while(travPath.size() != 0){
            q = travPath.top(); // node to be examined
            travPath.pop();

            if(travPath.size()){
                    p = travPath.top();
                    r = q==p->rightChild? p->leftChild : p->rightChild;
            }else{ // q is the root, it has no father
                    continue; // end it
            }
```

```
int Dq = q->getRankDifference(p);
int Dr;
if(!r){
    Dr = p->rank+1;
}else{
    Dr= r->getRankDifference(p);
}

if(Dq){
    //if(verbose)
//      treeLog<<"   - No rebalace needed."<<endl; // very talkative if you un-comment this...
    countNoRebalance++;//can finish.
    continue;
}else{
    if(Dr == 1){
    if(verbose)
        treeLog <<"   - Dr == 1 case."<< endl;

        p->rank++;
    }else{
    if(verbose)
        treeLog<< "   - Dr > 1 case." << endl;
    node *t = q==p->rightChild? q->leftChild : q->rightChild;
        int Dt;
        if(!t){
            Dt = q->rank+1;
        }else{
            Dt = t->getRankDifference(q);
        }
        if(Dt == 1){
```

```
if(verbose)
    treeLog<< "   - Dt == 1 case." << endl;


    if(!t){
    if(verbose)
treeLog << "   - Aborting, t child is null pointer."<<endl;
        exit(-1);
    }
    if(t->data > q->data){ // left-right rotation
    if(verbose)
treeLog<< "   - Double rotation at t, left-right." << endl;
        countDoubleL_R++;
        if(travPath.size()==1){ // p is the root
            q->rightChild = t->leftChild;
            p->leftChild = t->rightChild;
            t->rightChild = p;
            t->leftChild = q;
            t->rightChild->rank--;
            t->leftChild->rank--;
            t->rank++;
            this->root = t;


        }else{
            q->rightChild = t->leftChild;
            p->leftChild = t->rightChild;
            t->rightChild = p;
            t->leftChild = q;
            t->rightChild->rank--;
            t->leftChild->rank--;
            t->rank++;
```

```
					travPath.pop();
				node *grandfather = travPath.top();
				if(grandfather->rightChild == p){
					grandfather->rightChild = t;
					}else{
					grandfather->leftChild = t;
					}
					travPath.push(p);


				}


			}else{ // right-left rotation
			if(verbose)
		treeLog<< "   - Double rotation at t, right-left." << endl;
				countDoubleR_L++;
				if(travPath.size()==1){ // p is the root
					p->rightChild = t->leftChild;
					q->leftChild = t->rightChild;
					t->rightChild = q;
					t->leftChild = p;
					t->rightChild->rank--;
					t->leftChild->rank--;
					t->rank++;
					this->root = t;
				}else{
					p->rightChild = t->leftChild;
					q->leftChild = t->rightChild;
					t->rightChild = q;
					t->leftChild = p;
					t->rightChild->rank--;
```

```
                    t->leftChild->rank--;
                    t->rank++;
                    travPath.pop();
                node *grandfather = travPath.top();
                if(grandfather->rightChild == p){
                    grandfather->rightChild = t;
                    }else{
                    grandfather->leftChild = t;
                    }
                    travPath.push(p);
                }

            }
        }else{ // Dt == 2...
        if(verbose)
            treeLog << "   - Dt = 2 case." <<endl;

            if(q->data < p->data){
            if(verbose)
        treeLog << "   - Single right rotation at q."<<endl;
                countSingleRight++;
                if(travPath.size() == 1){
                node *tempA = q->rightChild;
                node *tempB = p->rightChild;
    q->rightChild = new node(p->data, --p->rank,tempA, tempB );
                    this->root = q;
                }else{
                node *tempA = q->rightChild;
                node *tempB = p->rightChild;
    q->rightChild = new node(p->data, --p->rank, tempA, tempB);
```

```
                travPath.pop();
            node *grandfather = travPath.top();
            if(grandfather->rightChild == p)
                grandfather->rightChild = q;
                else
                grandfather->leftChild = q;
                travPath.push(p);
            }
        }else{
        if(verbose)
    treeLog << "   - Single left rotation at q."<< endl;
            countSingleLeft++;
            if(travPath.size() == 1){
                node *tempA =  q->leftChild;
                node *tempB = p->leftChild;
q->leftChild = new node(p->data, --p->rank, tempB, tempA);
                this->root = q;
                }else{
                node *tempA = q->leftChild;
                node *tempB = p->leftChild;
q->leftChild = new node(p->data, --p->rank, tempB, tempA);
                travPath.pop();
            node *grandfather = travPath.top();
            if(grandfather->rightChild == p)
                grandfather->rightChild = q;
                else
                grandfather->leftChild = q;
                travPath.push(p);


            }
```

```
                                                }
                                        }

                                }

                        }
                }

        }
        return 1;
}

/***
        function deleteNode deletes nodes from the tree in the relaxed manner shown in the
        "Deletion without rebalancing in balanced binary trees" paper. If the deletion is
        succesful, value of one will be returned, else zero.
*/
int ravl :: deleteNode(int delData){
if(verbose)
        treeLog<< "[+] Atempting to delete node with value: "<<delData<<endl;
        node *tempNode = this->root;
        node *lastNode = 0;
        if(verbose)
                treeLog<<"   - Traversing the tree..."<<endl;
        while(1){
                if(!tempNode){
                                if(verbose)
                                treeLog<< "   - Aborting deletion, cannot find data."<<endl;
                                return 0;
                }
```

```
if(tempNode->data == delData){
        if(verbose)
                treeLog<< "   - Data found, proceeding to deletion."<<endl;

        // is a leaf
        if(!tempNode->rightChild && !tempNode->leftChild){
                if(verbose)
                        treeLog<< "   - Deleting a leaf."<<endl;
                if(lastNode){
                        if(lastNode->rightChild == tempNode){
                                lastNode->rightChild = 0;
                        }else{
                                lastNode->leftChild =0;
                        }
                }
                if(this->root == tempNode){
                        this->root = 0;
                }

                free(tempNode);
                return 1;
        }
        // is a father of one
        if(tempNode->rightChild && !tempNode->leftChild){
                if(verbose)
        treeLog<< "   - Deleting a father of one, left child missing."<<endl;
                if(lastNode){
                        if(lastNode->rightChild == tempNode){
                        lastNode->rightChild = tempNode->rightChild;
                        }else{
```

```
                    lastNode->leftChild = tempNode->rightChild;
                }
            }
            if(this->root == tempNode)
                    this->root = tempNode->rightChild;
            free(tempNode);
            return 1;
        }
        if(!tempNode->rightChild && tempNode->leftChild){
            if(verbose)
        treeLog<< "   - Deleting a father of one, right child missing."<<endl;
            if(lastNode){
                if(lastNode->rightChild == tempNode)
                    lastNode->rightChild = tempNode->leftChild;
                else
                    lastNode->leftChild = tempNode->leftChild;
            }
            if(this->root == tempNode)
                    this->root = tempNode->leftChild;
            free(tempNode);
            return 1;

        }
        // is a father of two
        if(tempNode->rightChild && tempNode->leftChild){
            if(verbose)
                treeLog<< "   - Deleting a father of two."<<endl;

            node *successor = tempNode->rightChild;
            node *newFather;
```

```
while(1){
        if(!successor){
                break; // newFather points to the new father
        }else{


                if(successor->data == tempNode->data){
                //cout << "duplicate found "<<endl;
                        while(travPath.size())
                                travPath.pop();
                        return 1;
                }// abort insertion, no duplicates are allowed
                if(successor->data > tempNode->data){
                        newFather = successor;
                        successor = successor->leftChild;
                }else{
                        newFather = successor;
                        successor = successor->rightChild;
                }
        }
        travPath.push(newFather);


}


travPath.pop();


if(!newFather->rightChild){
        successor = newFather->leftChild;
}else{
        successor = newFather->rightChild;
}
```

```
if(tempNode->rightChild == newFather){
    newFather->leftChild = tempNode->leftChild;
    newFather->rank = tempNode->rank;
    if(lastNode){
        if(lastNode->rightChild == tempNode){
            lastNode->rightChild = newFather;
        }else{
            lastNode->leftChild = newFather;
        }
    }
    if(this->root == tempNode)
        this->root = newFather;

    free(tempNode);
    while(travPath.size())//empty stack
        travPath.pop();
    return 1;
}else{

    if(!successor){
        if(travPath.top()->rightChild == newFather)
            travPath.top()->rightChild = 0;
        else
            travPath.top()->leftChild = 0;

    }else{
        if(travPath.top()->rightChild == newFather){
        travPath.top()->rightChild = successor;
        }else{
```

```
                                travPath.top()->leftChild = successor;
                                }


                        }
                        newFather->rightChild = tempNode->rightChild;
                        newFather->leftChild = tempNode->leftChild;
                        newFather->rank = tempNode->rank;
                }
                if(this->root == tempNode)
                        this->root = newFather;
                if(lastNode){


                        if(lastNode->rightChild == tempNode){
                                lastNode->rightChild = newFather;
                        }else{
                                lastNode->leftChild = newFather;
                        }
                }
                free(tempNode);
                while(travPath.size())//empty stack
                        travPath.pop();
                return 1;
        }



}else{
        if(tempNode->data < delData){
                lastNode = tempNode;
                tempNode = tempNode->rightChild;
        }else{
```

```
                        lastNode = tempNode;

                        tempNode = tempNode->leftChild;

                    }


                }

            }

}


double ravl :: diffclock(clock_t clock1,clock_t clock2)

{

        double diffticks=clock1-clock2;

        double diffms=(diffticks*10)/CLOCKS_PER_SEC;

        return diffms;

}


/***

        function countNodes returns the number of tree elements.

*/

int ravl :: countNodes(){

                stack<node *> nodeStack;

                node *curr = this->root;

                int count = 0;


                for (;;) { //inorder traverse the tree counting nodes

                    if (curr != NULL) {

                        nodeStack.push(curr);

                        curr = curr->leftChild;

                        continue;

                    }

                    if (nodeStack.size() == 0){
```

```
                    return count;
                }
            curr = nodeStack.top();
                nodeStack.pop();
                count++;
                curr = curr->rightChild;

            }
}
```

<u>Αρχείο node.cpp</u>

```
class node{
    public:
    node *leftChild;
    node *rightChild;
    int data;
    int rank;
    node(int newData);
    node(int newData,int rank, node *leftChild, node *rightChild);
    int getRankDifference(node *father);
};

node :: node(int newData){
    this->rank = 0; // new leaf
    this->data = newData;
    this->leftChild = 0; //null offspring
    this->rightChild = 0;
}
```

```
node :: node(int newData,int rank, node *leftChild, node *rightChild){
	this->rank = rank;
	this->data = newData;
	this->leftChild = leftChild;
	this->rightChild = rightChild;
}




int node :: getRankDifference(node *father){
	return father->rank- this->rank;
}
```

<u>Αρχείο timers.cpp</u>

```
#include <sys/time.h>
int getMillis(struct timeval first, struct timeval second){
	struct timeval lapsed;
	if  (first.tv_usec  >  second.tv_usec)  {
		second.tv_usec  +=  1000000;
		second.tv_sec--;
	}
	lapsed.tv_usec  =  second.tv_usec  -  first.tv_usec;
	lapsed.tv_sec  =  second.tv_sec  -  first.tv_sec;
	return int(lapsed.tv_sec*1000+lapsed.tv_usec/1000);
}
double getMicro(struct timeval first, struct timeval second){
	struct timeval lapsed;
	if  (first.tv_usec  >  second.tv_usec)  {
		second.tv_usec  +=  1000000;
```

```
            second.tv_sec--;
        }
        lapsed.tv_usec = second.tv_usec - first.tv_usec;
        lapsed.tv_sec = second.tv_sec - first.tv_sec;
        return double(lapsed.tv_sec*1000000+lapsed.tv_usec);
}
```

Αρχείο delTest.cpp

```
#include <iostream.h>
#include "node.cpp"
#include "ravl.cpp"
#include "timers.cpp"
#include <stdlib.h>
#include <sys/time.h>
#define MAX_POW 8
#define MAX_INIT_POW 8
#define MAX_DEL_POW 5

int main(){
        ravl *tree;
        int *data = new int[pow(10, MAX_POW)];
        struct timeval first, second;
        struct timezone tzp;
        ofstream fout;
        fout.open("timeResults.txt");

        // generate random insertion data
        srand(time(NULL));
        for(int i =0;i<pow(10,MAX_POW);i++) // 10 million random elements
```

```
        data[i] = rand();


for(int delPow =4;delPow<MAX_DEL_POW;delPow++){

        for(int power = 7;power<MAX_INIT_POW;power++){

                if(power<delPow)continue;

                tree = 0;

                tree = new ravl(false);

                int INIT_MAX = (int)pow(10, power);


                // filling up the tree to an initial state

                srand(1); // restart the seed to 1

                gettimeofday(&first,  &tzp);

                        for(int i = 0;i<INIT_MAX;i++)

tree->insertNode(data[rand()%(int)pow(10,MAX_POW)]); // insert pseudorandomly

                gettimeofday  (&second,  &tzp);

                cout<<"Filling up the tree with "<<INIT_MAX<<" elements, took ";

                cout<<getMillis(first, second)<<" ms."<<endl;


                // do the deletions

                int DEL_MAX = (int)pow(10,delPow);

                srand(1); // restart the seed as before the insertions


                // timing

                gettimeofday(&first,  &tzp);

                        for(int i = 0;i<DEL_MAX;i++)

                                tree->deleteNode(data[i]);

                gettimeofday  (&second,  &tzp);

                fout<<getMillis(first, second)<<" ";

        }

        fout<<endl;
```

```
        }
        free(tree);
        fout.close();
}
```

<u>Αρχείο insTest.cpp</u>

```
#include <iostream.h>
#include "node.cpp"
#include "ravl.cpp"
#include "timers.cpp"
#include <stdlib.h>
#include <sys/time.h>
#define MAX_POW 8
#define MAX_INIT_POW 7
#define MAX_INS_POW 7

int main(){
        ravl *tree;
        int *data = new int[pow(10, MAX_POW)];
        struct  timeval  first,   second;
        struct  timezone  tzp;
        ofstream fout;
        fout.open("timeResults.txt");

        // generate random insertion data
        srand(time(NULL));
        for(int i =0;i<pow(10,MAX_POW);i++) // 10 million random elements
                data[i] = rand();
        for(int insPow = 1;insPow<MAX_INS_POW;insPow++){
```

```
		for(int power = 2;power<MAX_INIT_POW;power++){
			tree = 0;
			tree = new ravl(false);
			int INIT_MAX = (int)pow(10, power);

			// filling up the tree to an initial state
			gettimeofday(&first,  &tzp);
				for(int i = 0;i<INIT_MAX;i++)
	tree->insertNode(data[rand()%(int)pow(10,MAX_POW)]); // insert pseudorandomly
			gettimeofday  (&second,  &tzp);
			cout<<"Filling up the tree with "<<INIT_MAX<<" elements, took ";
			cout<<getMillis(first, second)<<" ms."<<endl;

			// do the insertions
			int INS_MAX = (int)pow(10,insPow);
			srand(1); // restart the seed

			// timing
			gettimeofday(&first,  &tzp);
				for(int i = 0;i<INS_MAX;i++)
					tree->insertNode(rand());
			gettimeofday  (&second,  &tzp);
			fout<<getMillis(first, second)<<" ";
		}
		fout<<endl;
	}
	free(tree);
	fout.close();
}
```