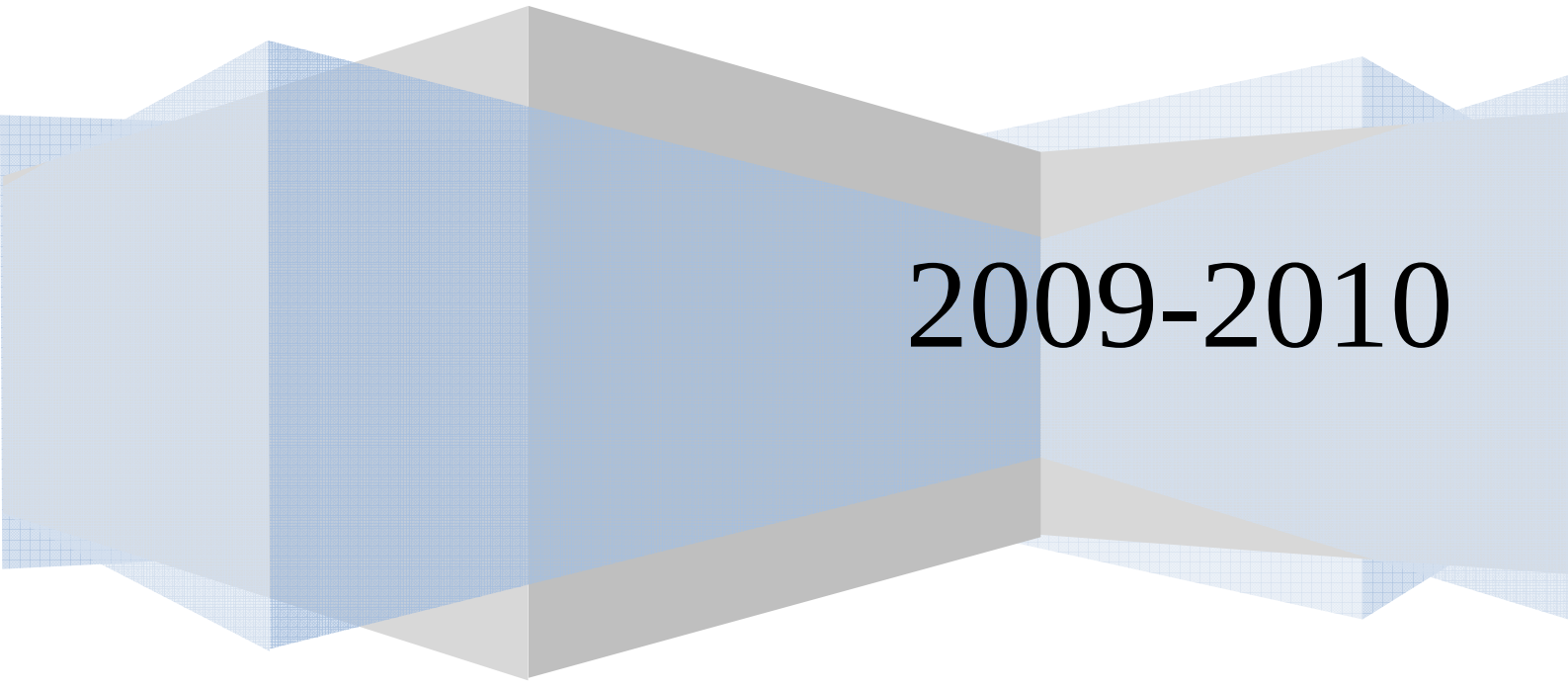
Πανεπιστήμιο Θεσσαλίας
Τμήμα Μηχανικών Η/Υ Τηλεπικοινωνιών & Δικτύων

Βαθμοζυγισμένα Δυαδικά Δέντρα Αναζήτησης

Επιβλέπων καθηγητής
Μποζάνης Παναγιώτης

Εμμανουηλίδης Κωνσταντίνος



2009-2010

Πίνακας περιεχομένων

Περίληψη

Από την ανακάλυψη των AVL δέντρων το 1962, έχουν προταθεί αρκετοί τρόποι για να ισορροπούμε δυαδικά δέντρα αναζήτησης. Αξιοσημείωτα είναι τα ερυθρόμαυρα(red-black) δέντρα στα οποία η bottom-up επαναζύγιση μετά από μία εισαγωγή ή διαγραφή κοστίζει στη χειρότερη περίπτωση O(1) κατανεμημένο χρόνο και O(1) περιστροφές. Αλλά ο χώρος σχεδίασης των ισορροπημένων δέντρων δεν έχει πλήρως εξερευνηθεί. Περιγράφουμε τα ***rank-balanced trees***, μια παραλλαγή των AVL trees, τα οποία μπορούν να επαναζυγιστούν bottom-up μετά από μία εισαγωγή ή διαγραφή στη χειρότερη περίπτωση σε O(1) κατανεμημένο χρόνο και το πολύ σε δύο περιστροφές, σε αντίθεση με τα ερυθρόμαυρα δέντρα τα οποία χρειάζονται μέχρι και τρεις περιστροφές ανά διαγραφή. Η επαναζύγιση μπορεί επίσης να γίνει top-down με σταθερό lookahead σε O(1) κατανεμημένο χρόνο. Χρησιμοποιώντας, λοιπόν μία νέα ανάλυση η οποία βασίζεται σε μια εκθετική συνάρτηση δυναμικού, δείχνουμε ότι τόσο η bottom-up όσο και η top-down επαναζυγίση τροποποιούν τους κόμβους εκθετικά σπάνια με βάση τα ύψη τους. Τέλος, υλοποιούμε στη γλώσσα προγραμματισμού C τα rank-balanced trees κάνοντας παράλληλα ορισμένες μετρήσεις αποδεικνύοντας την ορθότητα της θεωρητικής ανάλυσης.

1. ΕΙΣΑΓΩΓΗ – ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ

1.1. Αντικείμενο Δομών Δεδομένων

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

1.2. Βασικές λειτουργίες (πράξεις) επί των Δομών Δεδομένων

Οι βασικές λειτουργίες (ή αλλιώς πράξεις) επί των δομών δεδομένων είναι οι ακόλουθες:

Προσπέλαση (access), πρόσβαση σε ένα κόμβο με σκοπό να εξετασθεί ή να τροποποιηθεί το περιεχόμενό του.

Εισαγωγή (insertion), δηλαδή η προσθήκη νέων κόμβων σε μία υπάρχουσα δομή.

Διαγραφή (deletion), που αποτελεί το αντίστροφο της εισαγωγής, δηλαδή ένας κόμβος αφαιρείται από μία δομή.

Αναζήτηση (searching), κατά την οποία προσπελαύνονται οι κόμβοι μιας δομής, προκειμένου να εντοπιστούν ένας ή περισσότεροι που έχουν μια δεδομένη ιδιότητα.

Ταξινόμηση (sorting), όπου οι κόμβοι μιας δομής διατάσσονται κατά αύξουσα ή φθίνουσα σειρά.

Αντιγραφή (copying), κατά την οποία όλοι οι κόμβοι ή μερικοί από τους κόμβους μίας δομής αντιγράφονται σε μία άλλη δομή.

Συγχώνευση (merging), κατά την οποία δύο ή περισσότερες δομές συνενώνονται σε μία ενιαία δομή.

Διαχωρισμός (separation), που αποτελεί την αντίστροφη πράξη της συγχώνευσης.

1.3. Στατικά Δένδρα

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

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

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

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

οι οποίες αλλάζουν την μορφή του εκάστοτε δένδρου, χωρίς να αφαιρούν ή να προσθέτουν κόμβους.

1.4. 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 δυναμικά δένδρα έχει ως εξής:

addLeaf(u,kindofson)	προσθέτει ένα νέο κόμβο ως kinofson(δεξί ή αριστερό) παιδί του u, εφ' όσον δεν διαθέτει τέτοιο
deleeteNode(u)	αφαιρεί τον κόμβο u, εφ' όσον έχει το πολύ έναν μη κενό(non null) γιο

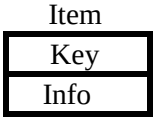
1.5. Εισαγωγή στο πρόβλημα Λεξικού

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

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

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

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



1.6. Ανάλυση Αλγορίθμων

Ανάλυση αλγορίθμου (algorithm analysis) 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 και αποτελεσματικότητας του εκάστοτε αλγορίθμου συνιστούν την ***πολυπλοκότητα χρόνου*** και **χώρου** του ***(time and space complexity)***.

1.7. Ανάλυση Χειρότερης Περιπτώσεως

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

1.8. Ανάλυση Μέσης Περιπτώσεως

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

1.9. Ανάλυση Επιμερισμένης ή Κατανεμημένης Περιπτώσεως

Η πολυπλοκότητα μια δομής καθορίζει και την συμπεριφορά του αλγορίθμου που την μεταχειρίζεται. 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 μιας πράξεως να μεταβάλλεται, αρκεί «ακριβές» πράξεις να ακολουθούνται από πολλές «φθηνές». Αυτός ο τρόπος αναλύσεως της επιδόσεως μιας δομής, ως μέσος όρος επιδόσεως επί ακολουθίας πράξεων, είναι γνωστός ως ***Ανάλυση Επιμερισμένης ή Κατανεμημένης Περιπτώσεως (amortized case analysis).***

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

1.9.1. Μέθοδος Λογαριασμού Τραπεζίτη ή Λογιστή

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

Συνήθως η διαφορά μεταξύ πραγματικού και κατανεμημένου κόστους χαρακτηρίζεται ως πίστωση (credit) και δηλώνει είτε το πλεόνασμα, που κατατίθεται προς μελλοντική χρήση, κατά τη εξυπηρέτηση των επόμενων πράξεων, είτε το δάνειο, που λαμβάνεται από τα αποθεματικά, για την κάλυψη των τρεχουσών αναγκών μιας πράξεως, ώστε να καλυφθεί το επιπλέον, από το παρατηρούμενο, κόστος των «ακριβών» πράξεων.

1.9.2. Μέθοδος Δυναμικού

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

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

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

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

Το κατανεμημένο κόστος c_i' της o_i ορίζεται ως:

$$c_i' = c_i + \Delta\Phi_i$$

Δηλαδή, το πραγματικό κόστος συν την μεταβλητή που επήλθε στο δυναμικό εξ αιτίας της $o_{i..}$

Καθίσταται λοιπόν φανερό πως, κεντρικό ρόλο στην ανάλυση δυναμικού, παίζει η επιλογή της κατάλληλης συναρτήσεως δυναμικού Φ. Χάρις στην τελευταία, κάποιες πράξεις χρεώνονται περισσότερο (όταν $\Phi_i > 0$) και κάποιες λιγότερο (όταν $\Phi_i < 0$), συνολικά, όμως, επιτυγχάνεται ορθή ερμηνεία της πολυπλοκότητας της A.

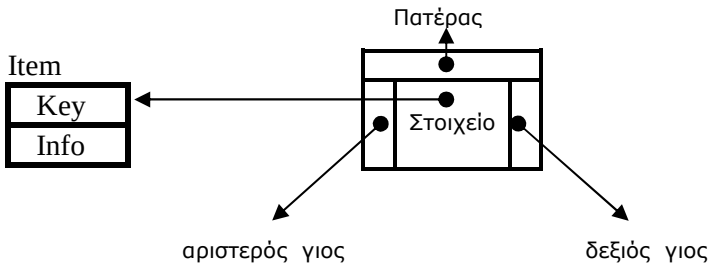
2. ΔΥΑΔΙΚΑ ΔΕΝΔΡΑ ΑΝΑΖΗΤΗΣΕΩΣ

2.1. Αζύγιστα Δυαδικά Δένδρα Αναζητήσεως

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

1. τα κλειδιά του αριστερού υποδένδρου του u είναι μικρότερα από το κλειδί του u
2. τα κλειδιά του δεξιού υποδένδρου του u είναι μεγαλύτερα (ή ίσα) από το κλειδί του u.

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



Σχήμα 2.1: Γραφική παράσταση κόμβου δυαδικού δένδρου αναζητήσεως έτσι ώστε, σε κάθε κόμβο υ της δομής να ισχύει η παρακάτω αμετάβλητη συνθήκη (invariant):

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

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

2.1.1. Περιγραφή Πράξεων

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

Αναζήτηση στοιχείου. Εφαρμόζουμε τον παρακάτω Αλγόριθμο:

Algorithm FINDNODE (BTNobe υ, Object key)
Input: Ένας κόμβος δένδρου υ και ένα κλειδί key
Output: Ο κόμβος του υποδένδρου T_υ όπου θα έπρεπε να υπάρχει στοιχείο με κλειδί key

1. If (key < key του υ) { // πηγαίνουμε αριστερά

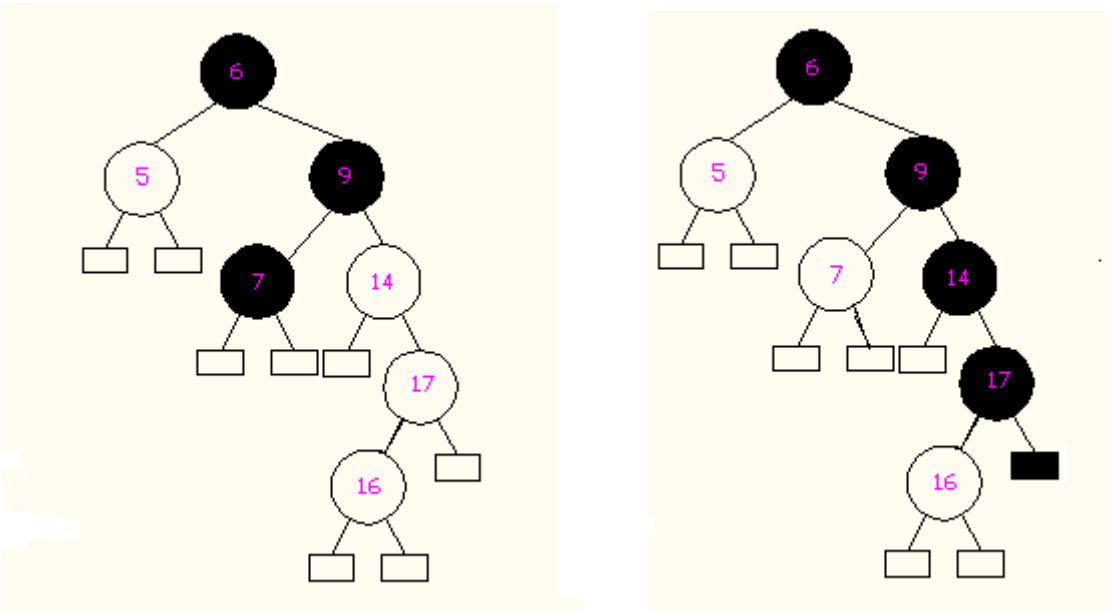
```
2.      if (ο υ δεν είναι έχει αριστερό παιδί)
3.          return υ;
4.   else
5.          return FindNobe(αριστερό παιδί του υ, key)
6.   }
7.   else if (key ==key του υ) // το βρήκαμε
8.          return υ;
9.   else { // πηγαίνουμε δεξιά
10.     if (ο υ δεν έχει δεξιό παιδί)
11.         return υ;
12. else
13.         return FindNode(δεξιό  παιδί του υ, key)
14. }
end of FINDNODE
```

Η κλήση του παραπάνω αλγόριθμου 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 διαθέτοντα ένα κενό (null) δείκτη στη 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 = FindNobe (key, ρίζα του δένδρου);
2.  if (το κλειδί του insNode == key)
3.          return insNobe.getElement ();
4.  else
5.          return null;
end of FINDONFO
```

Στο σχήμα 2.2. εικονίζονται τα ακόλουθα ***μονοπάτια αναζητήσεως***, για δυο περιπτώσεις αναζητήσεως: μια επιτυχημένης, για το κλειδί 10, και μιας αποτυχημένης, για το κλειδί 16. Παρατηρούμε ότι η τελευταία μας οδηγεί σε κενό παιδί. Είναι ανάγκη, σε αυτό το σημείο να επισημάνουμε ότι οι κενοί δείκτες εικονίζονται ως ακμές σε μικρούς τετράγωνους κόμβους □. Θα ακολουθήσουμε αυτή τη σύμβαση, σε όλες τις δενδρικές δομές που θα παρουσιάσουμε στη συνέχεια, θεωρώντας τους □ ως κενά φύλλα. Κάτι τέτοιο όπως θα φανεί διευκολύνει την πλήρη καταγραφή και κατανόηση των ενεργειών των πράξεων. Στο παράδειγμα μας το κενό φύλλο δείχνει την θέση όπου θα έπρεπε να υπάρχει κόμβος με το κλειδί 16.



(α) (β)

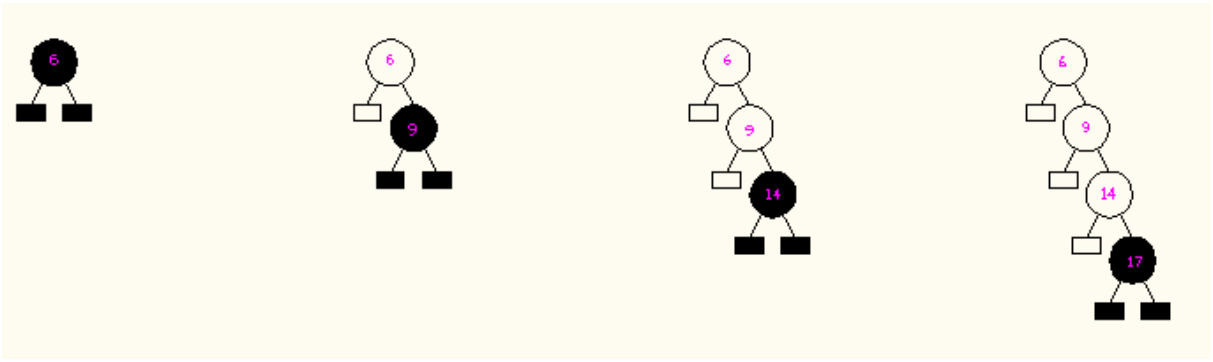
Σχήμα 2.2: Στιγμιότυπα: (α) μια επιτυχημένης αναζητήσεως του 7, (β) μιας αποτυχημένης του 20.

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

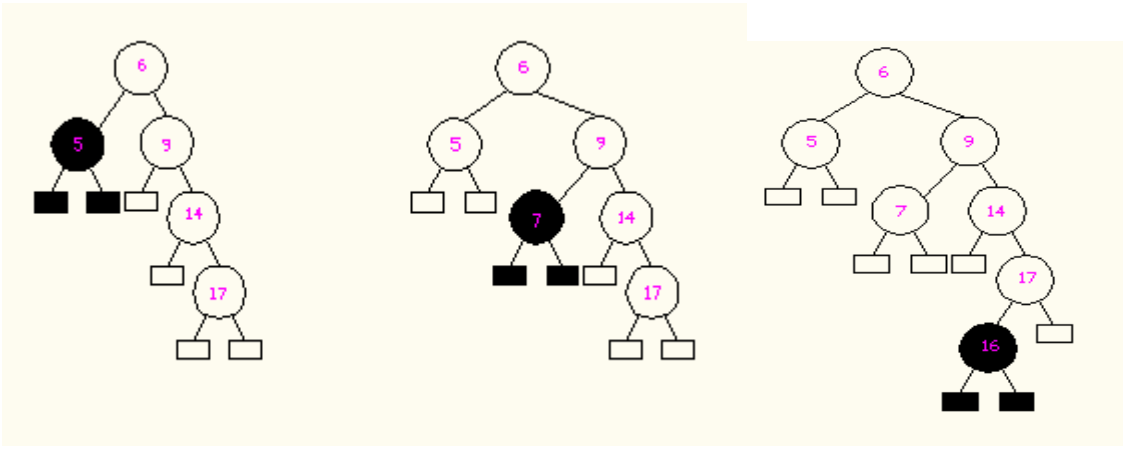
```
Algorithm INSERTITEM(item i)
Input:      Ένα στοιχείο i
Output:     Ο κόμβος όπου το i θα τοποθετηθεί, εάν δεν υπάρχει ήδη
1.  insNode = FindNobe ((  x = κλειδί του i), ρίζα του δένδρου);
2.  if (το κλειδί του insNode == x)
3.        return null;
4.  else if (το κλειδί του insNode > x){
5.  δημιούργησε ένα νέο κόμβο w ως αριστερό παιδί του insNode και τοποθέτησε το i
6.        return w;
7.  }
8.  else {
9.   δημιούργησε ένα νέο κόμβο w ως δεξιό παιδί  του insNode και τοποθέτησε το i
10.     return w;
11. }
end of INSERTITEM
```

Το σχήμα 2.3. παρουσιάζει παράδειγμα 12 διαδοχικών ενθέσεων σε ένα αρχικά άδειο δένδρο αναζήτησης. Με έντονο χρώμα εικονίζονται τα μονοπάτια αναζήτησης, δηλαδή οι δείκτες που ακολουθούν οι αντίστοιχες διαδικασίες αναζήτησης προκειμένου να εντοπιστούν οι θέσεις τοποθετήσεως των νέων κόμβων. Λόγου χάριν η ένθεση του 18 (σχ. 2.3 (θ)) προκαλεί αναζήτηση που

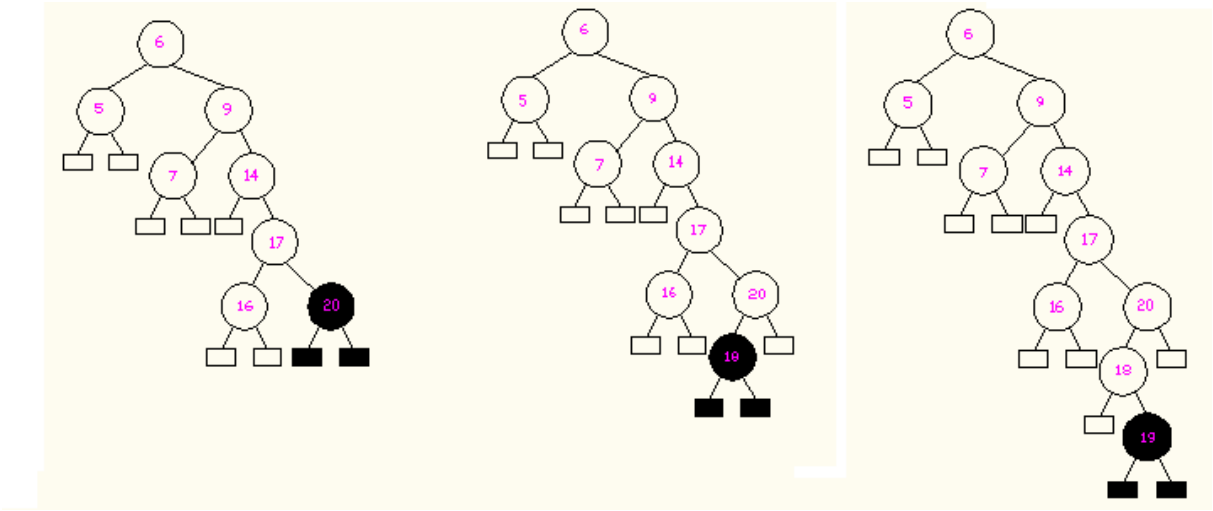
καταλήγει στον άκρα δεξιά κόμβο με το κλειδί 20. Έπειτα δημιουργείται ένας νέος κόμβος, αριστερό παιδί του 20, ώστε να στεγάσει το στοιχείο με τιμή 18.



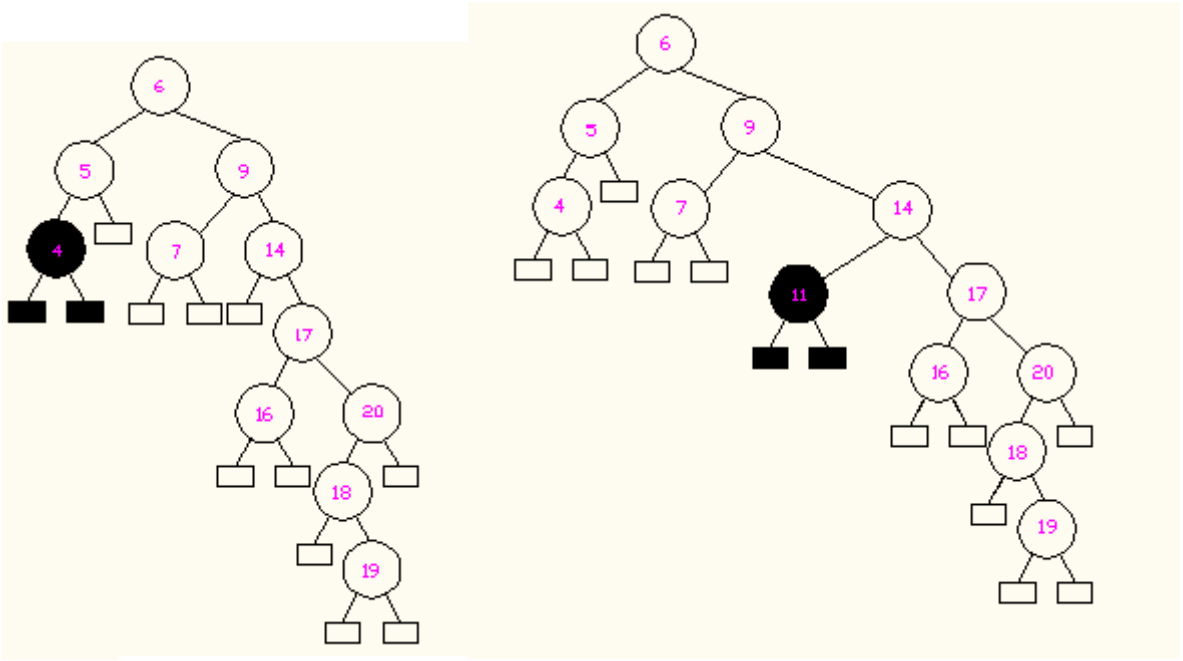
(α) (β) (γ) (δ)



(ε) (στ) (ζ)



(η) (θ) (ι)



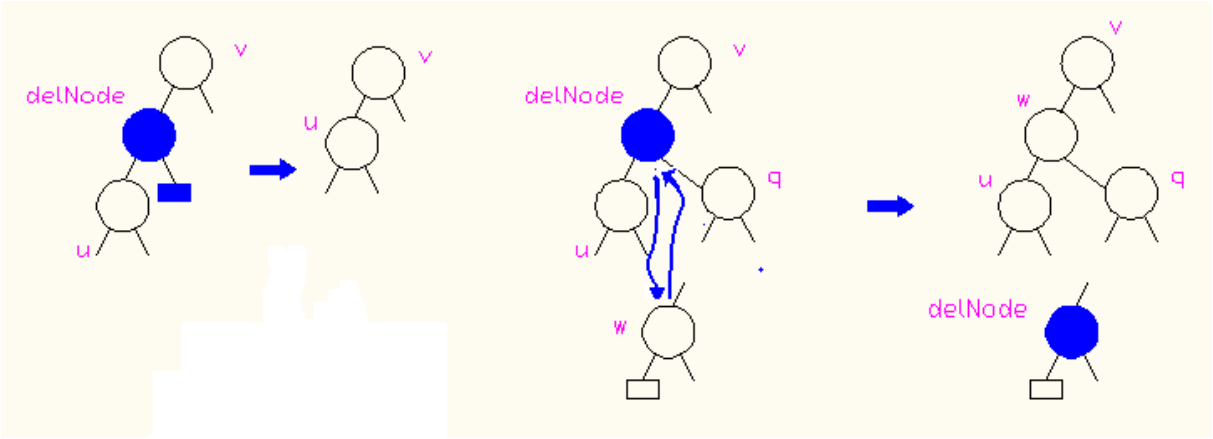
(ια) (ιβ)

Σχήμα 2.3: (α) –(ιβ) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 16, 20, 18, 19, 4, 11.

Διαγραφή στοιχείου. Η διαγραφή ενός στοιχείου i είναι ελάχιστα δυσκολότερη. Περιλαμβάνει και αυτή μια 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 δυο περιπτώσεις:

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

(β) Διαφορετικά ο delNode διαθέτει μη δεξιό κενό γιο q (Σχ. 2.4 (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 άλλαξαν θέσεις! Οπότε δημιουργούνται οι συνθήκες της περιπτώσεως (α) η οποία και εφαρμόζεται ώστε delNode να διαγράφει δίχως της αμετάβλητης συνθήκης.



(α) (β)

Σχήμα 2.4: Περιπτώσεις απόσβεσης: (α) άμεση, και (β) ανταλλαγή με τον αριστερότερου απόγονο του δεξιού υποδένδρου.

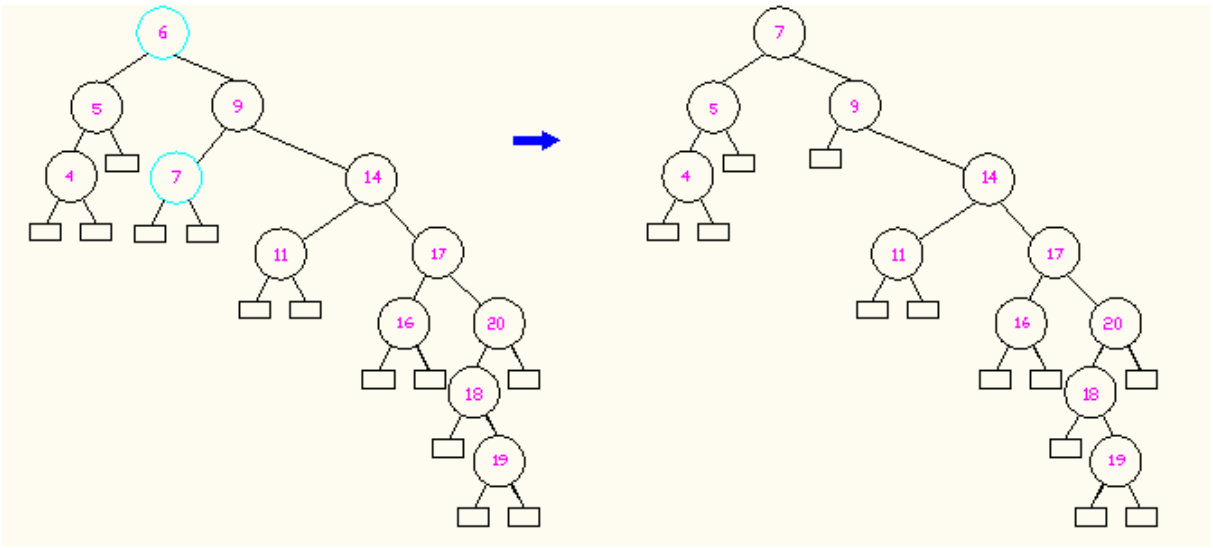
Τυπικότερα έχουμε:

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

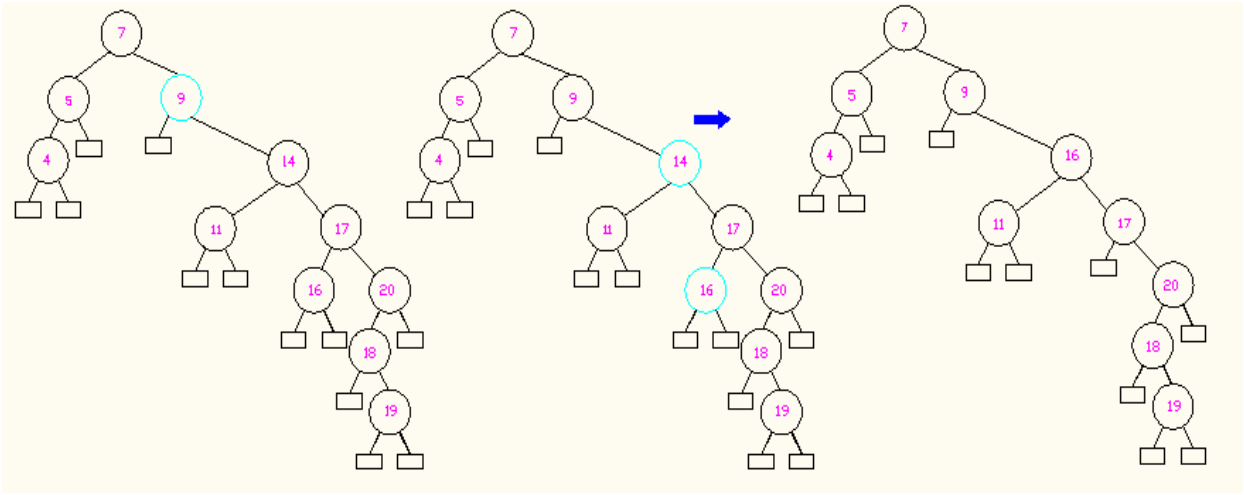
1. delNode = FindNobe (k, ρίζα του δένδρου);
2. if (το κλειδί του delNode != k)
3. return null;
4. if(delNode έχει τουλάχιστον ένα κενό παιδί){
5. σβήσε τον delNode όπως στα LinkedBinaryTrees;
6. επέστρεψε τον άλλο γιο;
7. }
8. else {
9. βρες τον κόμβο w με το μικρότερο κλειδί του δεξιού υποδένδρου του delNode;
10. αντάλαξε τα στοιχελια τους ;
11. σβήσε τον w όπως στην απλή περίπτωση
12. }

end of DELETEITEM

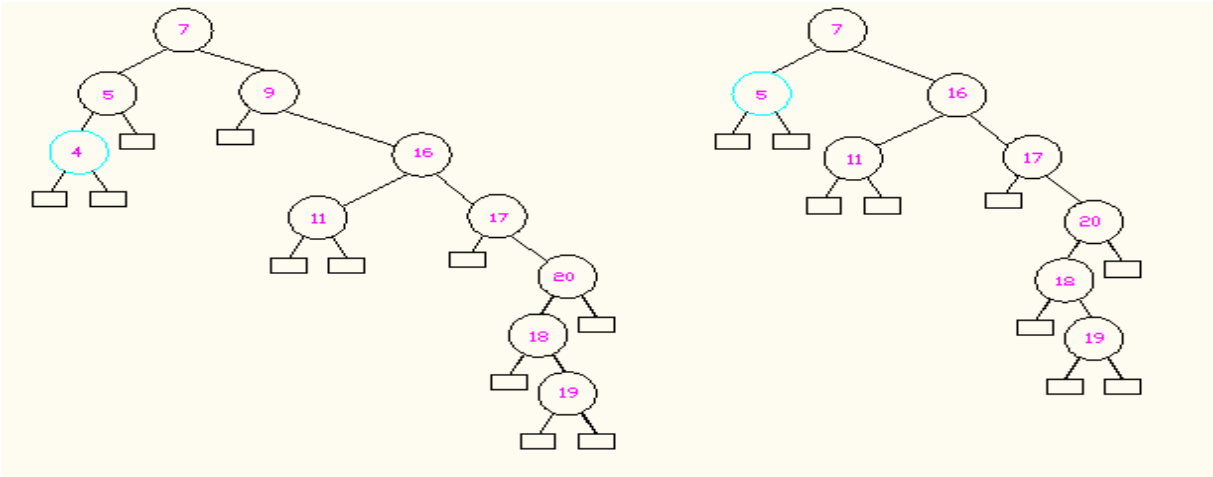
Το Σχήμα 2.5 δείχνει παράδειγμα 6 διαδοχικών αποσβέσεων στο δένδρο του Σχήματος 2.3 (ιβ). Παρατηρήστε πως αντιμετωπίζονται οι «δύσκολες» περιπτώσεις των 6 και 14 – Σχήματα (α) και (γ) , αντιστοίχως.


6
5
9
4
7
14
11
17
16
20
18
19
7
5
9
4
14
11
17
16
20
18
19

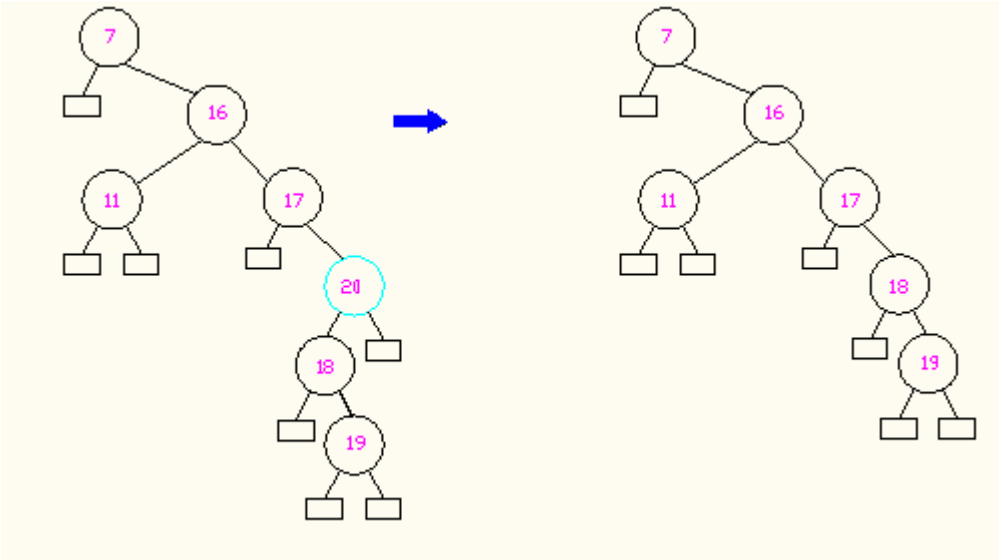
(α)


7
5
9
4
14
11
17
16
20
18
19
7
5
9
4
14
11
17
16
20
18
19
7
5
9
4
16
11
17
20
18
19

(β) (γ)


7
5
9
4
16
11
17
20
18
19
7
5
16
11
17
20
18
19

(δ) (ε)



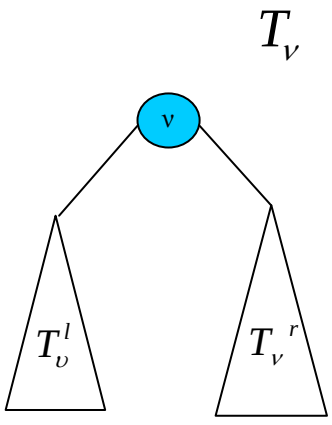
(στ)

Σχήμα 2.5: (α)-(στ) Διαδοχικές αποσβέσεις των 6, 9, 14, 5, 20.

2.1.2. Ανάλυση Πολυπλοκότητας

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

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



Σχήμα 2.6: Τα δυο υποδένδρα του $T_{υ.}$

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

Λήμμα 2.1 Έστω Qi(Tu) και Qe(Tu), αντίστοιχα το εσωτερικό και το εξωτερικό μήκος μονοπατιού ενός δένδρου Tu με ρίζα τον κόμβο u το μέγεθος του οποίου σε πλήθος κόμβων είναι |Tu|, ενώ διαθέτει αριστερό και δεξιό υποδένδρο T_v^r και T_υ^l αντίστοιχα. Τότε ισχύουν τα εξής:

(α) $\wp_i(T_\upsilon) = \wp_i(T_\upsilon^l) + \wp_i(T_\upsilon^r) + |T_\upsilon| - 1$

(β) $\wp_e(T_\upsilon) = \wp_e(T_\upsilon^l) + \wp_e(T_\upsilon^r) + |T_\upsilon| - 1$

(γ) $\wp_e(T_\upsilon) = \wp_i(T_\upsilon^l) + 2|T_\upsilon|$

Στη συνέχεια θα ασχοληθούμε με το μέσο μήκος εσωτερικού μονοπατιού σε ένα αζύγιστο δένδρο αναζήτησης Τ, μεγέθους $|T| = n$ στοιχείων. Θα θεωρήσουμε πως το Τ είναι ένα τυχαίο δένδρο αναζήτησης. Έστω λοιπόν, $\sqrt{n}$

$$a_1, a_2, a_3, \ldots\ldots a_n$$

τα κλειδιά που συμμετέχουν στην δημιουργία του Τ, διατεταγμένα κατά αύξουσα τιμή. Η «τυχαιότητα» προκύπτει από τη θεώρηση ότι το Τ χτίστηκε μέσω n διαδοχικών ενθέσεων και τα n συμμετέχοντα κλειδιά είναι ανεξάρτητοι και ομοιόμορφα κατανεμημένοι τυχαίο 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 στη ρίζα του δένδρου, καθώς υπάρχουν (n-1)! ακολουθίες ενθέσεων που ξεκινούν από το a_i ως πρώτο εισαχθέν κλειδί. Σημειώστε πως η τοποθέτηση του i-ιστου μικρότερου στοιχείου a_i στη ρίζα συνεπάγεται εξ ορισμού των δυαδικών δένδρων αναζήτησης, ότι το αριστερό υποδένδρο $T_{a_i}^l$ περιέχει i-1 κλειδιά ενώ το δεξιό υποδένδρο $T_{a_i}^r$ θα στεγάσει n-i κλειδιά.

Λήμμα 2.2 Το μέσο εσωτερικό μήκος μονοπατιού σε ένα τυχαίο δυαδικό δένδρο αναζήτησης Τ, n στοιχείων, είναι $\approx 1.39n\log n$, ενώ το μέσο εξωτερικό μήκος του είναι $\approx 1.39n\log n+2n$.

Λήμμα 2.3 Έστω Τ ένα τυχαίο δυαδικό δένδρο αναζήτησης μεγέθους n. Το αναμενόμενο κόστος ενός επιτυχημένου και ενός αποτυχημένου ψαξίματος είναι λογαριθμικό στο πλήθος n των στοιχείων.

Τέλος 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 ενθέσεων και αποσβέσεων. Το τελευταίο οφείλεται στην υπόθεση του χτισίματος του τυχαίου δένδρου μέσω διαδοχικών ενθέσεων. Εάν παρεμβάλλονται και αποσβέσεις, τότε η ανάλυση είναι δύσκολή, ενώ είναι δυνατόν να παραχθούν δένδρα με μέσο ύψος $\sqrt{n}$.

2.2. Δένδρα AVL

2.2.1. Ορισμός

Είναι φανερό πλέον πως ένα δένδρο αναζήτησης για 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 ονομασία του αποτελεί ακρωνύμιο από τα ονομάτων των ερευνητών που το εισήγαγαν το 1962, Adel'son, Vel'skii και Landis. Η πρωτοπορία της πρότασής τους ήταν η επιβολή της παρακάτω αμετάβλητης συνθήκης:

> **Ένα δυαδικό δένδρο Τ καλείται δένδρο AVL αν και μόνο αν τα ύψη των δυο υποδένδρων κάθε εσωτερικού κόμβου διαφέρουν το πολύ κατά ένα.**

Η ανωτέρω ιδιότητα εγγυάται λογαριθμικότητα του ύψους. Για την διατήρηση της συνθήκης αυτής, θα χρειαστεί να διατηρήσουμε σε ένα κόμβο και το ύψος του.

2.2.2. Περιγραφή των βασικών Πράξεων

Οι πράξεις FindNode(BTNode υ, Object key) και FindInfo(Object key) είναι ίδιες με αυτές των απλών αζύγιστων δυναμικών δυαδικών δένδρων αναζήτησης. Μεγαλύτερη δυσκολία παρουσιάζουν οι πράξεις της ενθέσεως και της αποσβέσεως. Κεντρικό σημείο σε αυτές αποτελούν οι λεγόμενες ***επαναζυγιστικές πράξεις (rebalancing operation)***, οι οποίες προσπαθούν να διατηρήσουν τη διαφορά ύψους αδελφών κόμβων φραγμένη από την μονάδα.

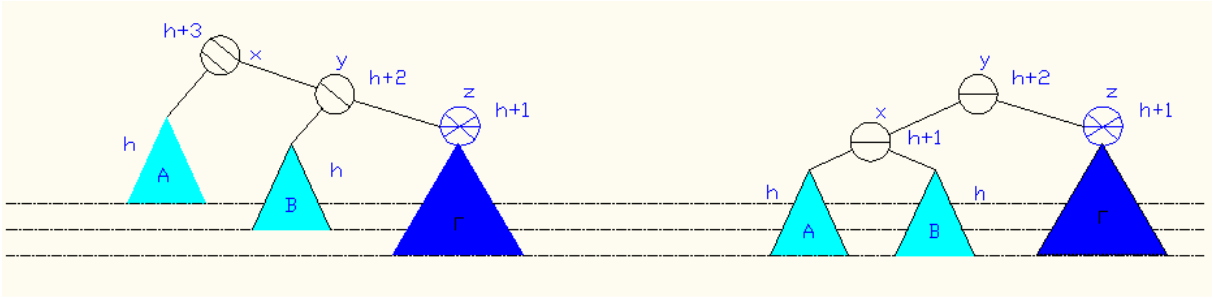
Εισαγωγή Στοιχείου. Είναι δυνατόν ένας νέος κόμβος να προκαλέσει παραβίαση της συνθήκης AVL, η οποία πρέπει να διατηρηθεί βάσει κατάλληλων ενεργειών. Το σχεδιάγραμμα του αλγορίθμου της ενθέσεως έχει ως ακολούθως:

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

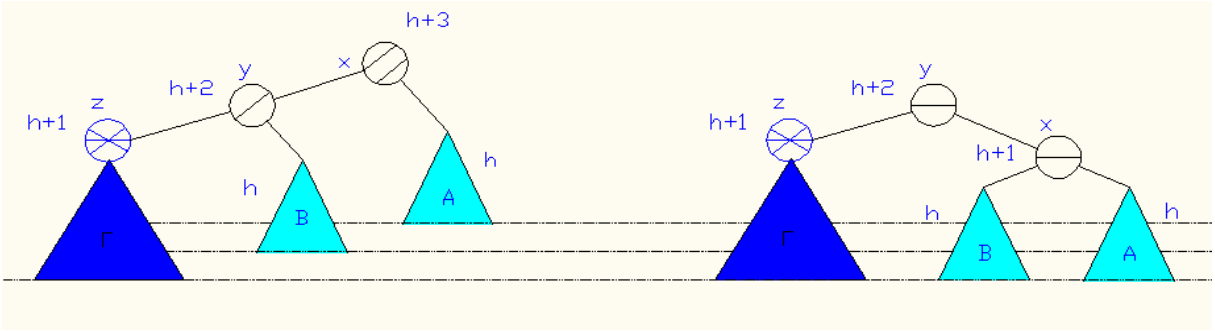
1. Κλήση της αντίστοιχης μεθόδου εισαγωγής των απλών αζύγιστων δυαδικών δένδρων;
2. Με αφετηρία τον κόμβο της ενθέσεως αναρρίχηση προς τα επάνω, με επιδιόρθωση των υψών, μέχρι να εντοπιστεί ο πρώτος μη ισοζυγισμένος κόμβος υ;
3. Ισοζύγιση του υ βάσει της κατάλληλης επαναζυγιστικής πράξης (απλής ή διπλής περιστροφής) του Σχήματος 3.7;

end of INSERTITEM

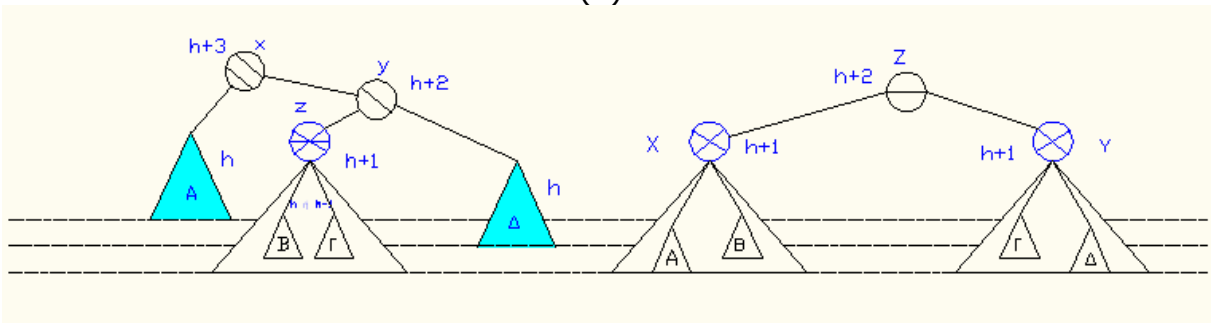
Στο Σχήμα 2.7 εικονίζονται οι δυο περιπτώσεις που μπορεί να εμφανιστούν:



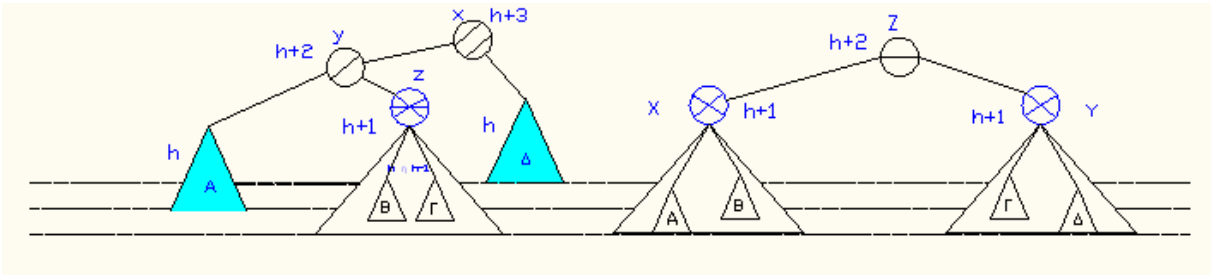
Ή



(α)



Ή



(β)

Σχήμα 2.7: Περιπτώσεις αζύγιστων κόμβων μετά από ένθεση και αντίστοιχης επαναζύγισης. Οι γραμμές εντός των κόμβων δηλώνουν την ισορροπία (-), την κατά μία μονάδα υψομετρικής διαφορά(\ ή /), την διπλή (\\ ή //) ή κάποια από τις 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 τη 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 στη προηγούμενη τιμή του h=2 και άρα να αποκατασταθεί η ισορροπία στο δένδρο.

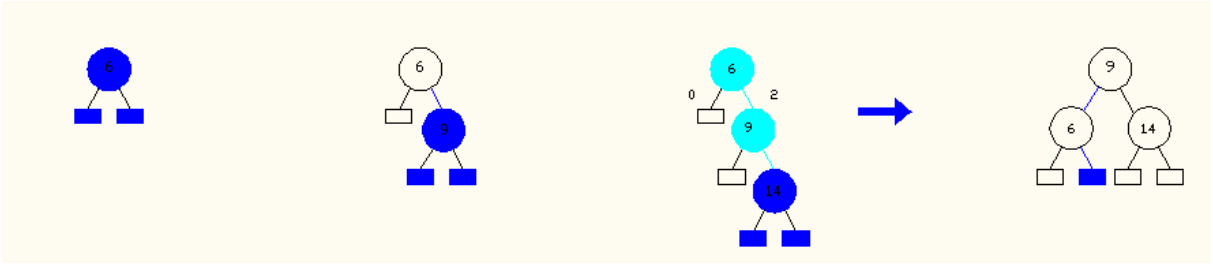
Με τον όρο ***περιστροφή*** εννοούμε την ανταλλαγή ρόλων των δυο εμπλεκόμενων κόμβων. Λ.χ. στην περίπτωσή μας ο x γίνεται γιος του y και αποκτά ως υποδένδρο το Β και ο y μετατρέπεται σε πατέρα του x. Η φορά της περιστροφής είναι αντίθετη προς της κατεύθυνση όπου «γέρνει» το δένδρο ώστε να εξισορροπηθεί η συμμετρία.

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

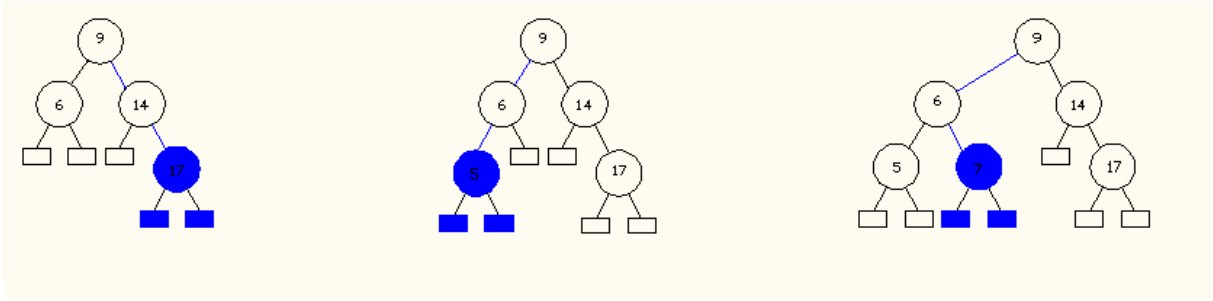
Η διπλή περιστροφή συνίσταται από δυο απλές περιστροφές αντίθετης φοράς. Για παράδειγμα μια αριστερή διπλή περιστροφή αποτελείται από μια δεξιά απλή περιστροφή στον ενδιάμεσο κόμβο y (με επακόλουθη την υποβάθμισή του), ακολουθούμενη από μια απλή περιστροφή στον υψηλότερο κόμβο x. Εδώ πρέπει να σημειωθεί πως περιστροφές απλές ή διπλές που αποκαθιστούν την ισοζύγιση στο δένδρο καλούνται ***τερματικές.***

Οι περιστροφές μεταβάλουν 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***.

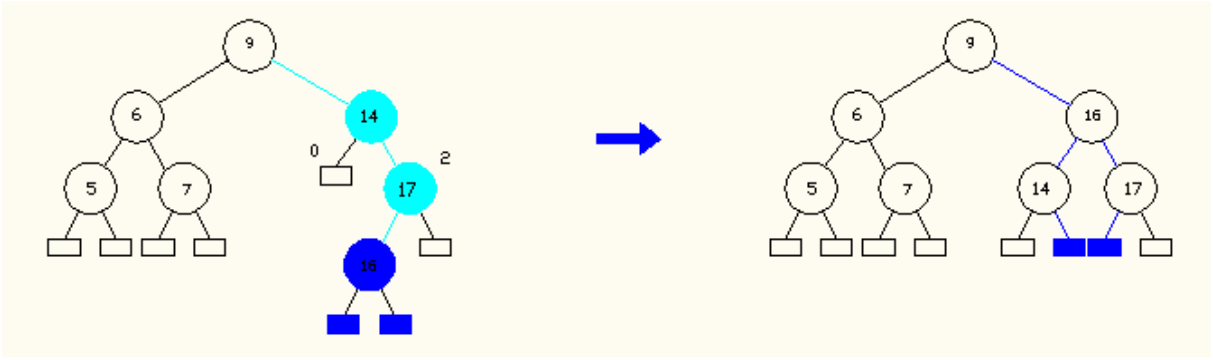
Τα σχήματα 2.8 και 2.9 παρουσιάζουν ένα παράδειγμα 12 διαδοχικών ενθέσεων σε ένα αρχικά άδειο δένδρο AVL. Με έντονο χρώμα εικονίζονται τα μονοπάτια αναζήτησης, δηλαδή οι δείκτες που ακολουθούν οι αντίστοιχες διαδικασίες αναζήτησης προκειμένου να εντοπιστούν οι θέσεις τοποθέτησης των νέων κόμβων. Για παράδειγμα μετά την εισαγωγή του 16 (περίπτωση (ζ)) αρχίζουμε να ανεβαίνουμε το μονοπάτι αναζήτησης. Διορθώνουμε την τοπική πληροφορία ύψους του 17, από 1 σε 2 και παρατηρούμε ότι δεν χρειάζεται να εκτελέσουμε κάποια δομική πράξη, καθώς η διαφορά των υψών των παιδιών του είναι $|1-0|=1$. Κατόπιν ανερχόμαστε στον κόμβο με κλειδί 14, προσαρμόζουμε το ύψος του, από 2 σε 3 και παρατηρούμε ότι η ένθεση έχει προκαλέσει διαφορά 2 στα ύψη των υποδένδρων του. Η παράβαση αυτή επιδιορθώνεται με αριστερή διπλή περιστροφή η οποία επαναφέρει το ύψος του κόμβου που πλέον στη θέση 14 στεγάζει το κλειδί 1 στην αρχική τιμή 2. Πρέπει να τονιστεί ότι μια παραβίαση μπορεί να εμφανιστεί σε οποιοδήποτε ύψος, δείτε λ.χ. την περίπτωση (ι) της εισαγωγής του 19.



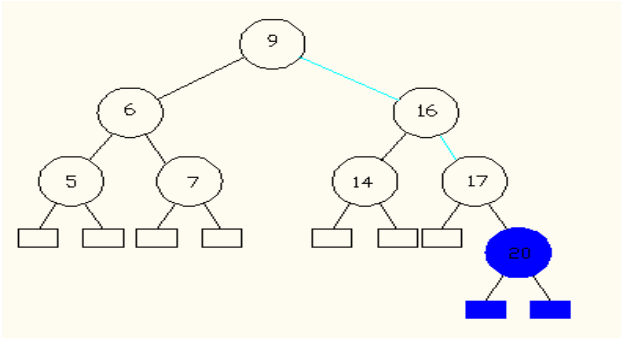
(α) (β) (γ)



(δ) (ε) (στ)

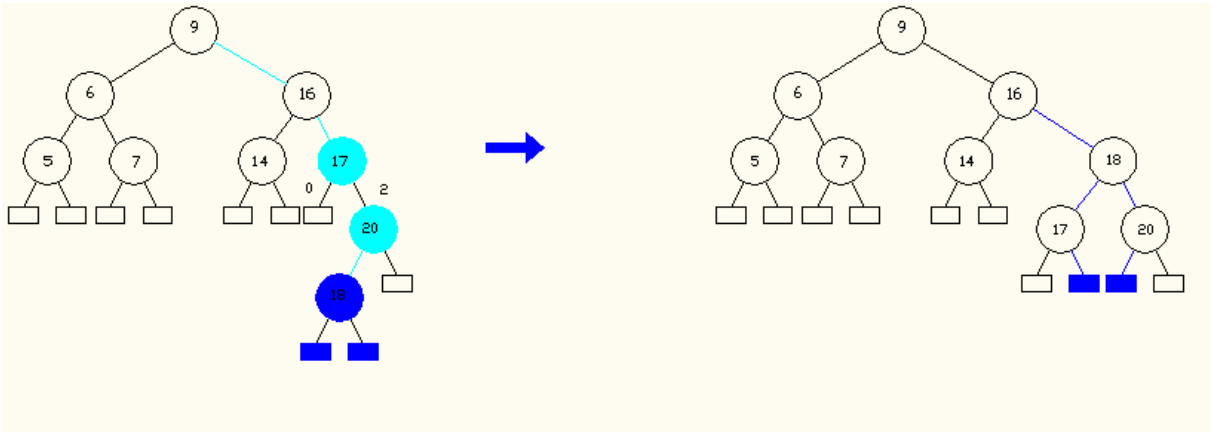


(ζ)

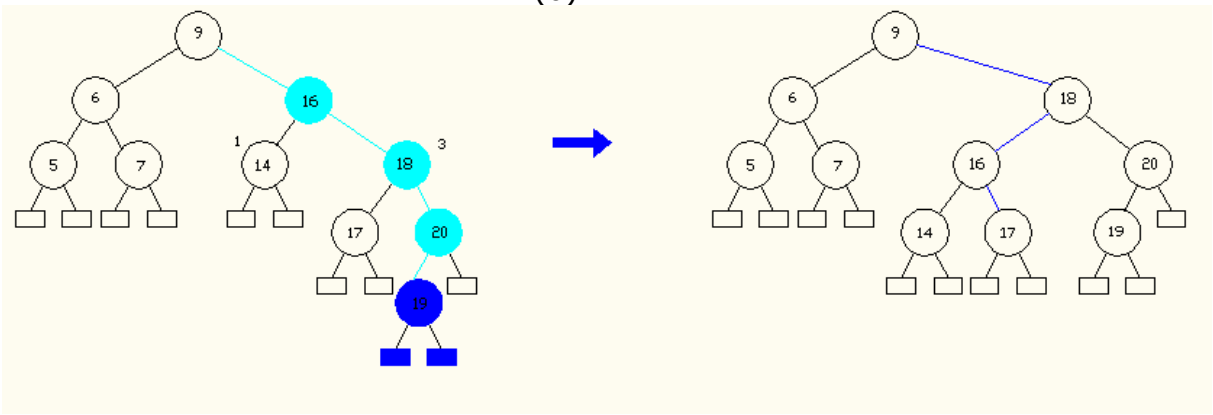


(η)

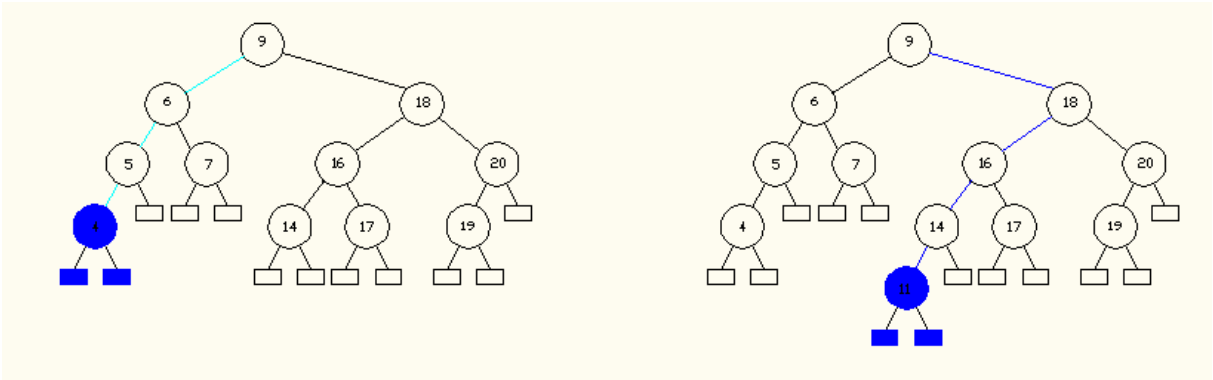
Σχήμα 2.8: (α)-(η) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 16, 20



(θ)



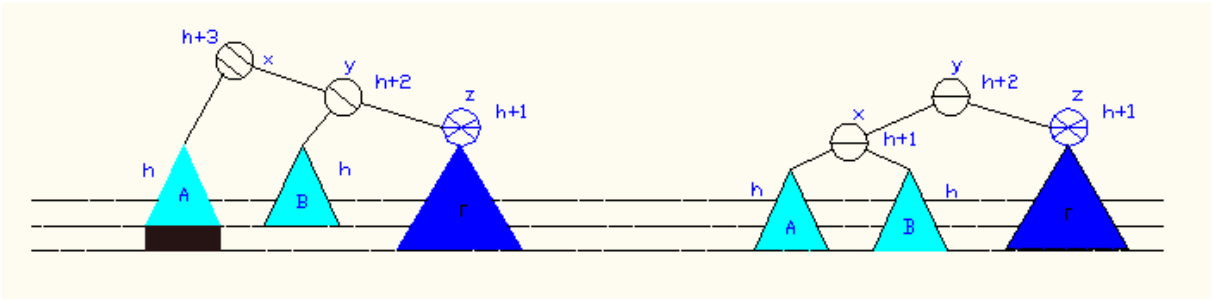
(ι)



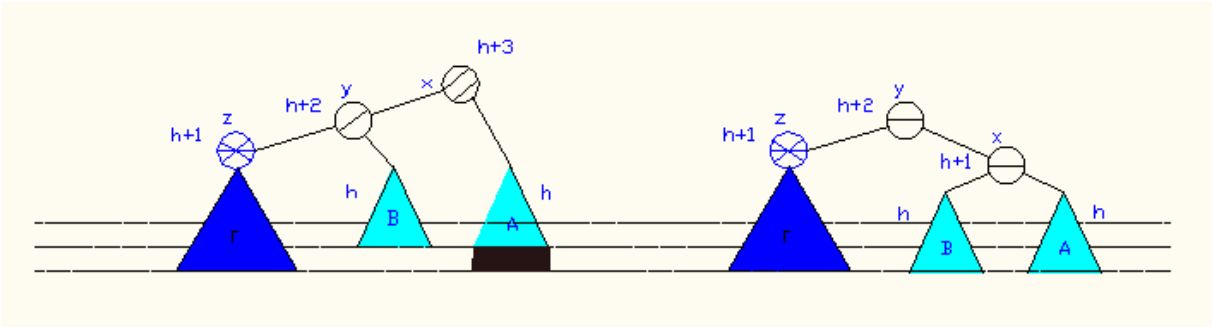
(ια) (ιβ)

Σχήμα 2.9: Συνέχεια (θ)-(ιβ) 18, 19, 4, 11

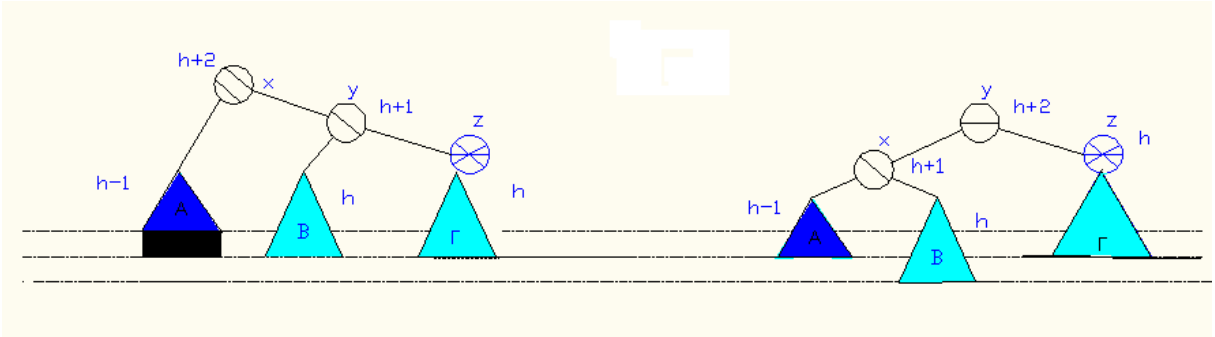
Διαγραφή στοιχείου. Όσον αφορά στην απόσβεση, στο Σχήμα 2.10 εικονίζονται οι τρεις περιπτώσεις που μπορεί να εμφανιστούν μετά την διαγραφή ενός στοιχείου:


h+3
x
y
h+2
z
h+1
h
A
h
B
Γ
y
h+2
z
h+1
x
h+1
h
A
h
B
Γ

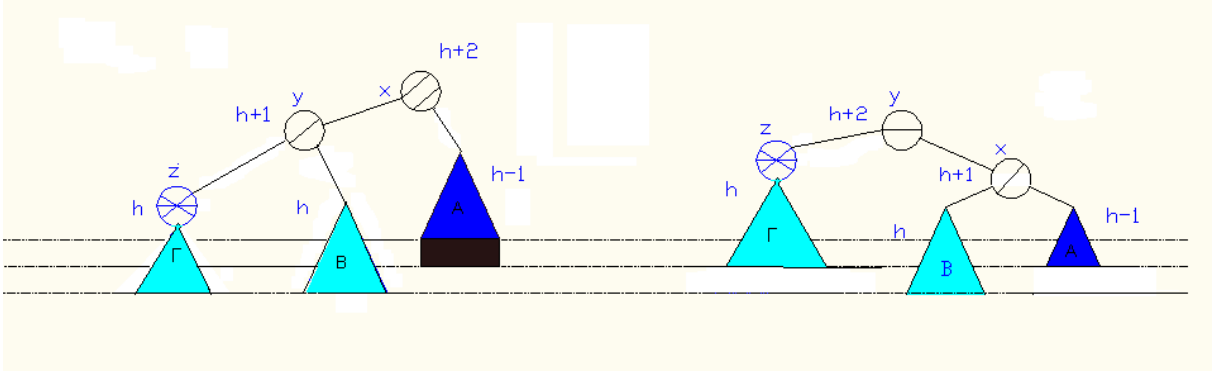
Ή


h+3
x
y
h+2
z
h+1
Γ
h
B
h
A
z
h+1
y
h+2
x
h+1
Γ
h
B
h
A

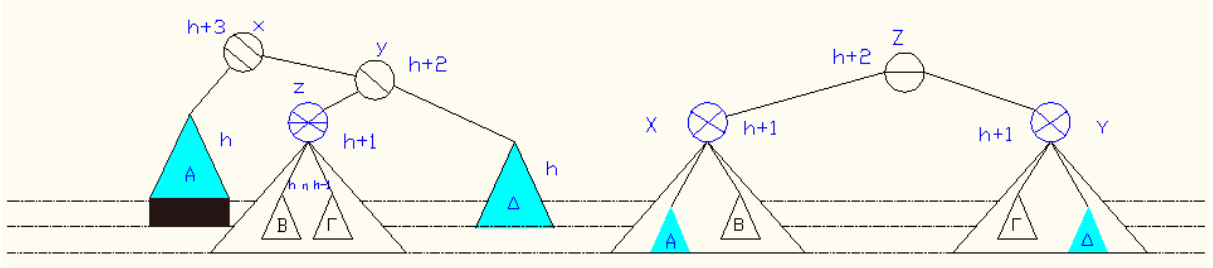
(α)


h+2
x
y
h+1
z
h−1
A
h
B
h
Γ
y
h+2
z
h
x
h+1
h−1
A
h
B
Γ

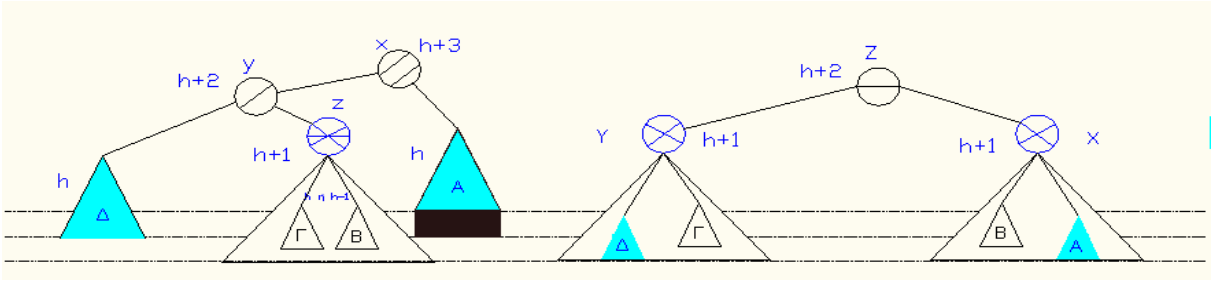
Ή


h+2
x
y
h+1
z
h−1
h
A
h
B
Γ
z
h+2
y
x
h+1
h
Γ
h−1
h
B
A

(β)


h+3
x
y
h+2
z
h
h+1
A
B
Γ
h
Δ
Z
h+2
X
h+1
h+1
Y
A
B
Γ
Δ

Ή



(γ)

Σχήμα 2.10: Περιπτώσεις αζύγιστων κόμβων μετά από απόσβεση και αντίστοιχης επαναζύγισης: (α) αριστερή ή δεξιά μη τερματική περιστροφή (rot1), (β) αριστερή ή δεξιά τερματική περιστροφή (rot2), (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 μια μονάδα, σε h και ο x παραβιάζει τη συνθήκη. Τότε μια αριστερή ή δεξιά απλή περιστροφή (rot1) στον x ισοζυγίζει το υποδένδρο μειώνοντας όμως το ύψος του κορυφαίου κόμβου κατά μια 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 εξάλειψη του καθώς ενδεχομένως το μεταθέτουν σε ψηλότερους 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 μια μονάδα, σε h-1 και ο x παραβιάζει τη συνθήκη. Τότε μια 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 δεξιά ή αριστερή «γωνία», το στοιχείο έχει διαγραφεί από το υποδένδρο Α, μειώνοντας το ύψος του, κατά μια μονάδα, σε h και ο x παραβιάζει τη συνθήκη. Τότε μια αριστερή ή δεξιά ***διπλή περιστροφή*** (drot) ισοζυγίζει το υποδένδρο, μειώνοντας όμως το ύψος κατά μία μονάδα. Με συνέπεια η επαναζυγιστική πράξη να είναι μη τερματική.

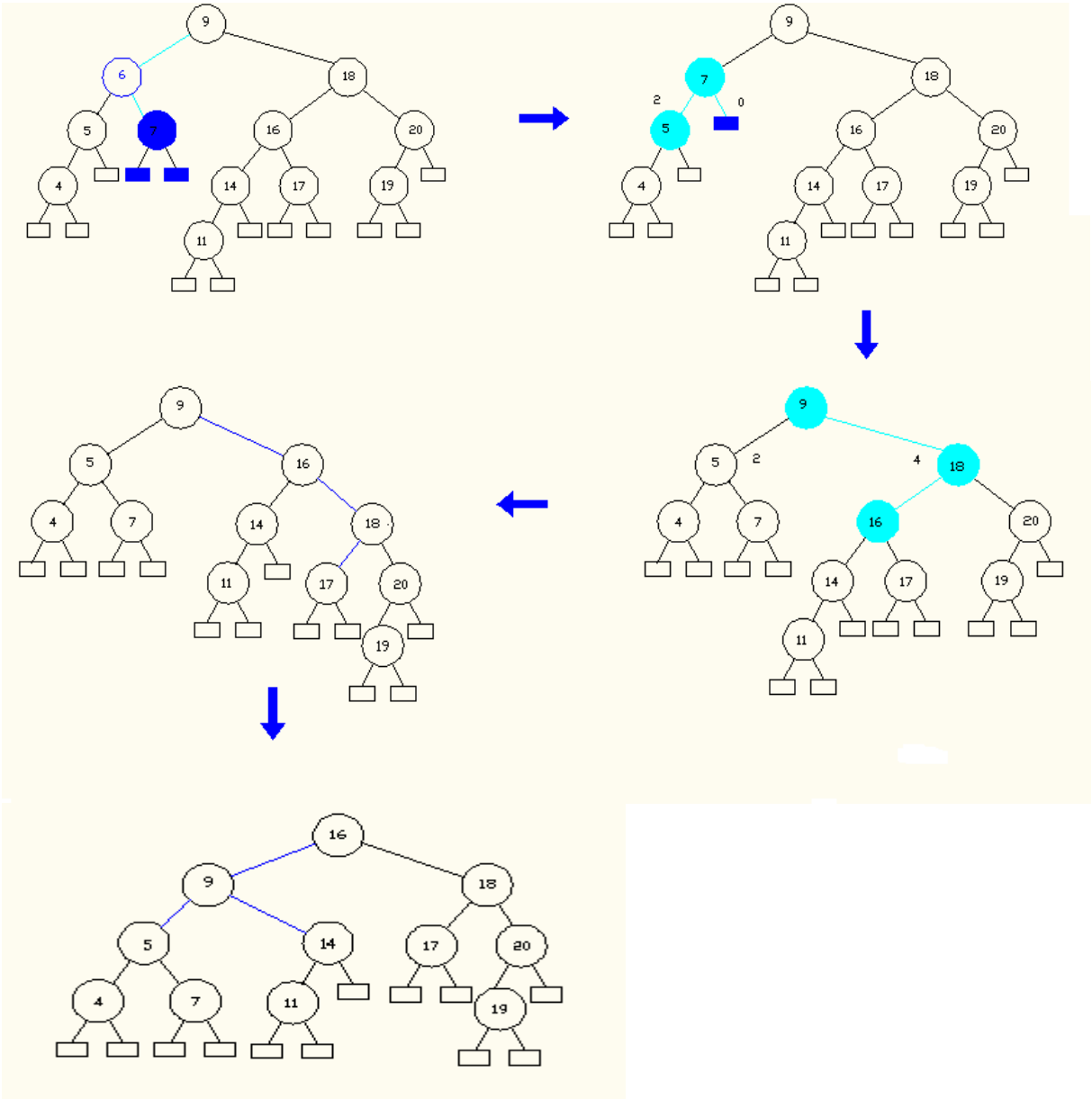
Η περιγραφή της πράξης της απόσβεσης σε ψευδογλώσσα έχε ως εξής:

Algorithm DELETEITEM(Object k)
Input: Ένα κλειδί k
Output: Η απόσβεση του κόμβου που περιέχει item με κλειδί k, επιτρεποντας τον εναπμείναντα κεμπλεκόμενο κόμβο

4. Κλήση της αντίστοιχης μεθόδου των απλών δυαδικών δένδρων;
5. Με αφετηρία τον κόμβο της απόσβεσης αναρρίχηση προς τη ρίζα, ταυτόχρονή επιδιόρθωση των υψών;
6. Σε κάθε κόμβο υ όπου σημειώνεται παραβίαση τη ισοζύγισης, εκτέλεση της κατάλληλης ισοζυγιστικής πράξης του Σχήματος 3.10;

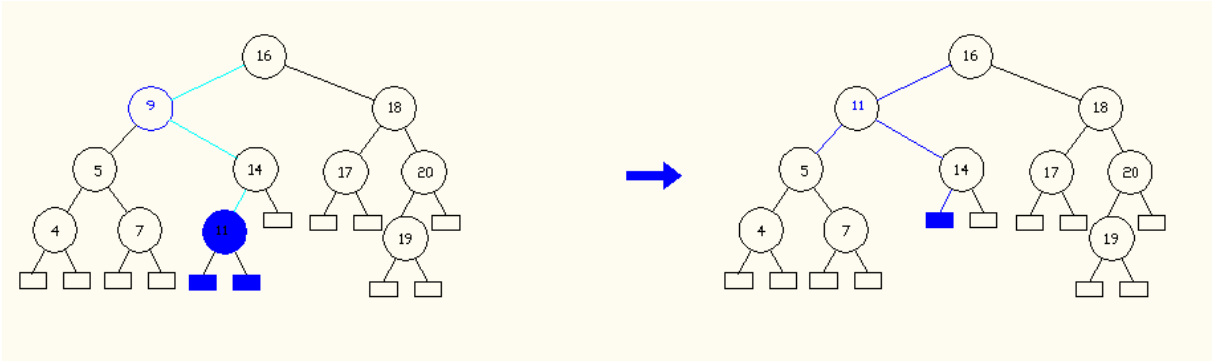
end of DELETEITEM

Στα Σχήματα 2.11, 2.12 και 2.13 εικονίζονται οι αλλαγές που επιφέρουν 6 διαδοχικές αποσβέσεις στο δένδρο του Σχήματος 2.9(ιβ). Αξίζει να επισημανθεί ή δύσκολη απόσβεση του 6 (περίπτωση (α)) η οποία προκαλεί διαδοχικές επαναζυγίσεις μέχρι την ρίζα. Αφού ανταλλάξει θέσεις με τον 7, ώστε να αποκτήσει κενά παιδιά διαγράφεται. Η απόσβεσή του παρ' ότι δεν αλλάζει το ύψος του 7 προκαλεί παραβίαση της συνθήκης ισοζύγισεώς του ($|2-0|=2$), η οποία βάσει της περίπτωσης (α), επιδιορθώνεται με μια δεξιά μη τερματική απλή περιστροφή, καθώς μειώνεται το ύψος κατά μια μονάδα. Ως εκ τούτου ανερχόμαστε στη ρίζα με το 9 όπου πάλι έχουμε παραβίαση συνθήκης, εφόσον $|2-4|=2$. Η ισοζύγιση επανέρχεται με μια αριστερή διπλή περιστροφή (περίπτωση (γ)) η οποία εμπλέκει τους 18 και 16, μειώνοντας το ύψος της ρίζας κατά μία μονάδα. Και η διαδικασία σταματά αφού δεν υπάρχουν άλλοι προγονικοί κόμβοι προς θεώρηση.

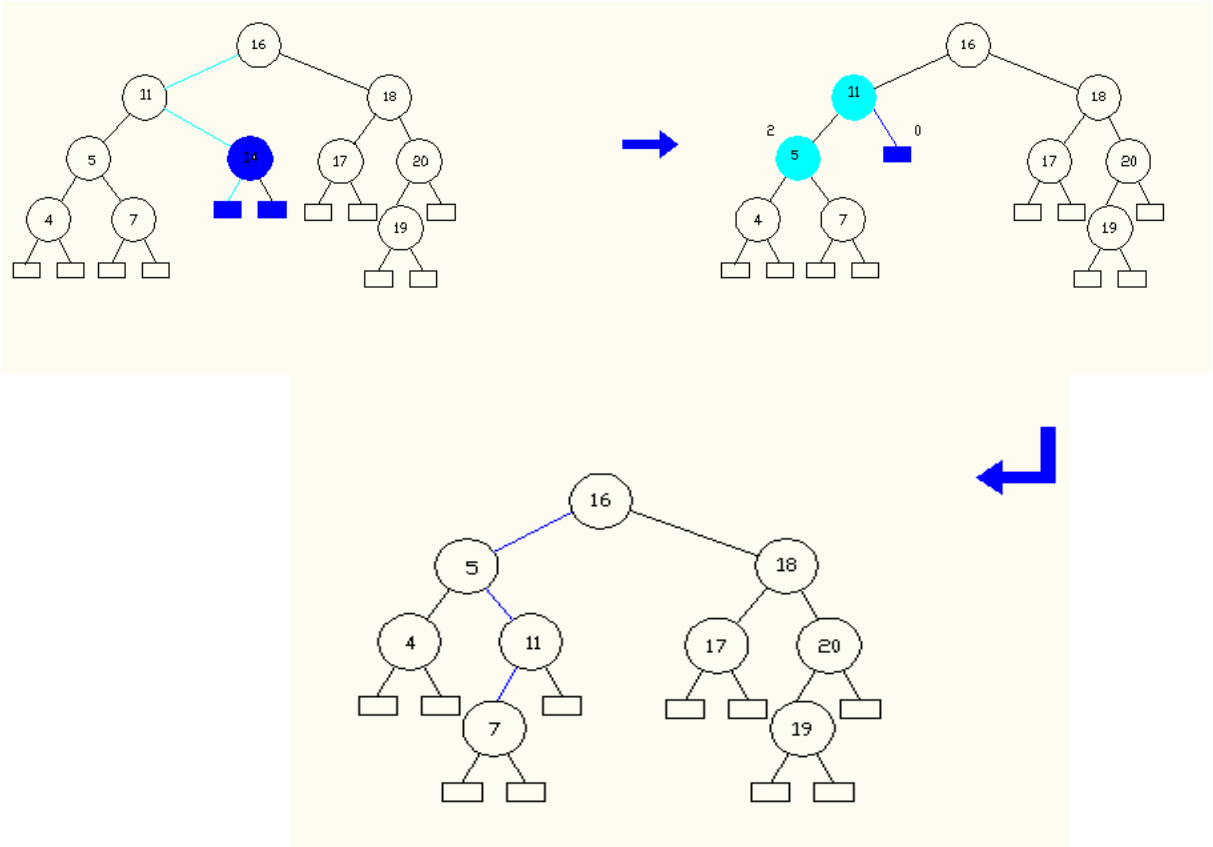


(α)

Σχήμα 2.11: Διαδοχικές αποσβέσεις: (α) 6.

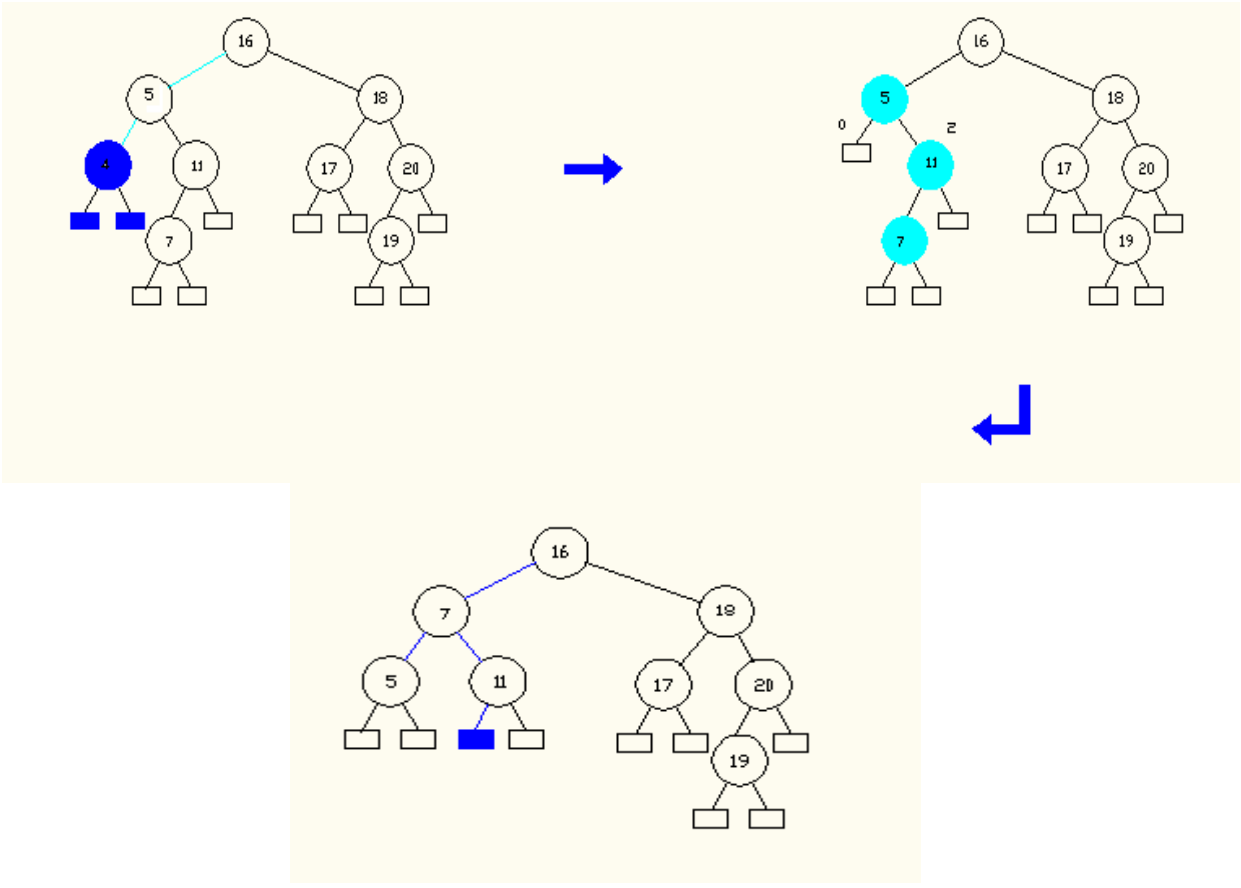


(β)

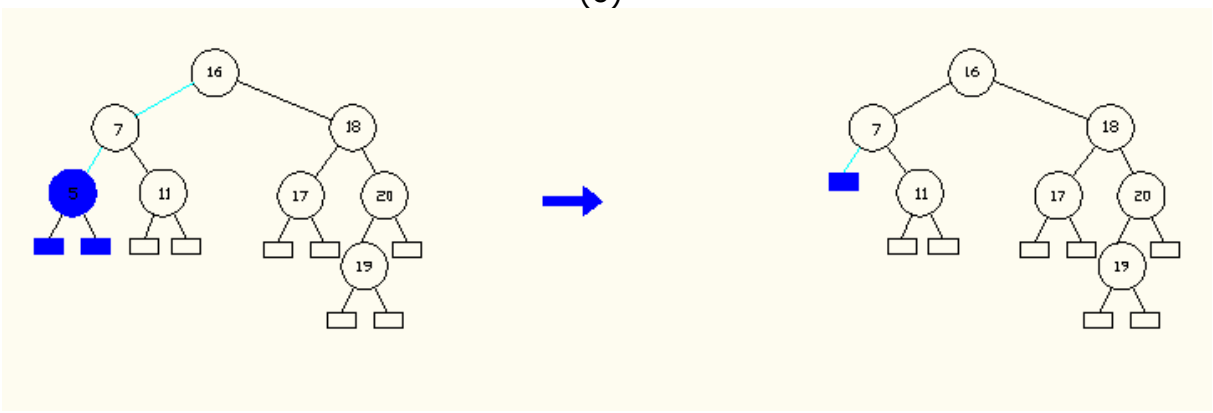


(γ)

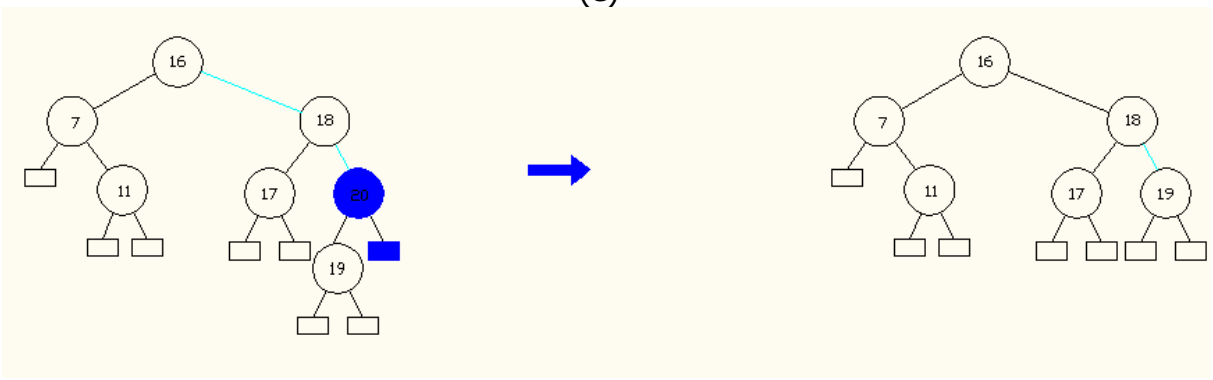
Σχήμα 2.12: (Συνέχεια) (β)-(γ) 9, 14.



(δ)



(ε)



(στ)

Σχήμα 2.13: (Συνέχεια)(δ)-(στ) 4, 5, 20.

2.2.3 Ιδιότητες – Ανάλυση Πολυπλοκότητας

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

Λήμμα 2.4 Κάθε υποδένδρο ενός δένδρου AVL είναι επίσης δένδρο AVL

Λήμμα 2.5 Το ύψος h ενός δένδρου AVL T n στοιχείων είναι O(logn)

Θεώρημα 2.2 Ένα δένδρο AVL n στοιχείων στη χειρότερη περίπτωση επιδεικνύει λογαριθμική συμπεριφορά και για τις τρεις πράξεις . Επιπροσθέτως μια ένθεση κοστίζει O(1), ενώ μια απόσβεση O(log n) επαναζυγιστικές (δομικές) πράξεις.

2.3. *Ερυθρόμαυρα Δένδρα*

2.3.1 Ορισμός

Ένα δυαδικό δένδρο Τ καλείται ***ερυθρόμαυρο red-black***, όταν όλοι οι κόμβοι του είναι χρωματισμένοι κόκκινοι ή μαύροι βάσει των ακολούθων κανόνων:

(α) η ρίζα του έχει χρώμα μαύρο,

(β) όλα τα (κενά) φύλλα είναι μαύρα,

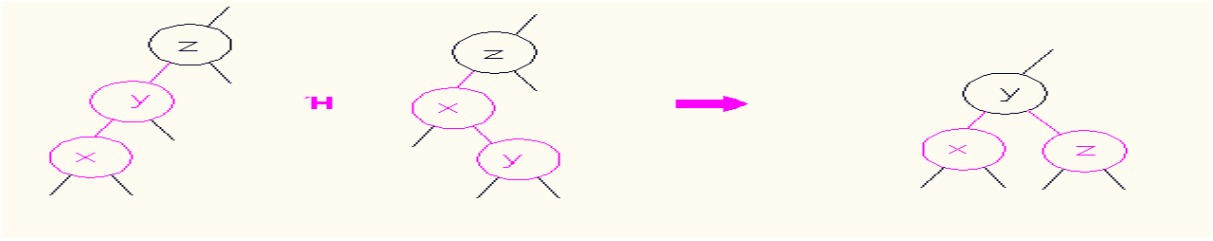
(γ) δεν υπάρχουν δυο διαδοχικοί κόκκινοι κόμβοι (ή ισοδύναμα, κάθε κόκκινος κόμβος έχει μαύρα παιδία), και

(δ) όλα τα μονοπάτια από τη ρίζα προς τα φύλλα διαθέτουν το ίδιο μαύρο βάθος ήτοι τον ίδιο αριθμό μαύρων κόμβων.

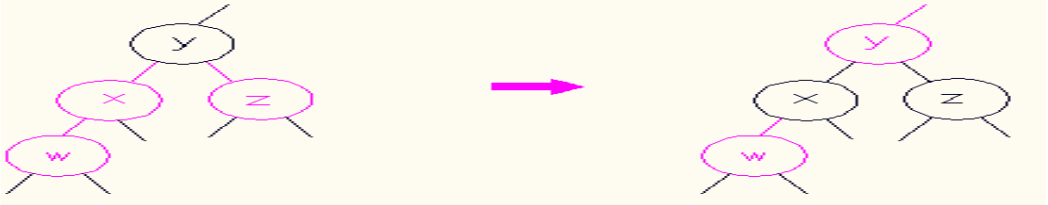
2.3.2 Περιγραφή των Βασικών Πράξεων

Οι πράξεις FindMode(BTNode u, Object key) και FindInfo(Object key) είναι ίδιες με αυτές των απλών αζύγιστων δυναμικών δυαδικών δένδρων αναζήτησης.

Εισαγωγή Στοιχείου. Η ένθεση ενός νέου στοιχείου i εκτελείται αρχικά όπως και στα απλά αζύγιστα δυαδικά δένδρα. Ο κόμβος u που θα στεγάσει το i χρωματίζεται κόκκινος και διαθέτει δυο μαύρα φύλλα. Κατά αυτόν τον τρόπο δεν παραβιάζεται ο κανόνας (δ) του ορισμού που αφορά το μαύρο βάθος, καθώς ο u αντικαθιστά ένα μαύρο (κενό) φύλλο. Η εισαγωγή του νέου κόκκινου κόμβου ίσως όμως προκαλέσει παραβίαση του κανόνα (γ) εάν ο πατέρας το u είναι κόκκινος. Διακρίνουμε δυο κύριες τέτοιες περιπτώσεις όπως εικονίζονται στο Σχήμα 2.14 παραλείποντας τις «συμμετρικές» που λαμβάνουμε εάν ο υψηλότερος από τους δυο διαδοχικούς κόκκινους κόμβους είναι δεξιό παιδί ή απλούστερα φανταστούμε πως βλέπουμε το εν λόγω σχήμα σε καθρέφτη.



(α)



(β)

Σχήμα 2.14: Επαναζυγιστικές πράξεις ενθέσεως: (α) C1 – απλή ή διπλή περιστροφή (rot ή drot) C1, (β) C2 – αντιστροφή χρωμάτων (color flip).

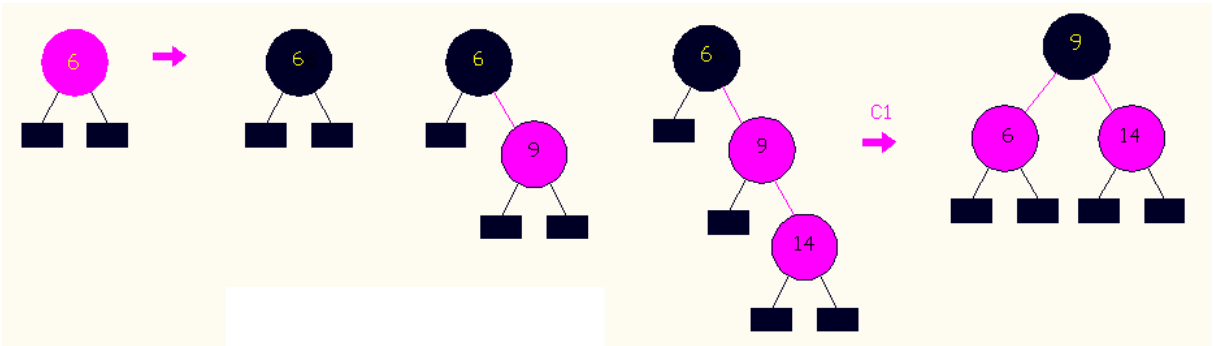
(α) Ο αδελφός του υψηλότερου κόκκινου κόμβου είναι μαύρος, οπότε το πρόβλημα λύνεται με απλή η διπλή περιστροφή. Η πράξη αυτή καλείται C1, είναι δομική και τερματική, καθώς αλλάζει την «τοπολογία» του δένδρου και επιλύει οριστικά το πρόβλημα, μετακινώντας τον υψηλότερο πλεονάζοντα κόκκινο κόμβο στο «αδελφό» υποδένδρο του μαύρου πατέρα.

(β) Ο υψηλότερος κόκκινος κόμβος διαθέτει κόκκινο αδελφό οπότε αντιστρέφουμε τα χρώματά τους με τον μαύρο πατέρα τους. Η πράξη αυτή καλείται C2 – ***αντίστροφή χρωμάτων (color flip)*****.** Παρατηρείστε δε πως δεν αλλάζει την μορφολογία του δένδρου ούτε επιλύει το πρόβλημα, απλώς το μεταθέτει στον (πρώην μαύρο) πατέρα όπου πρέπει να διεξαχθεί εκ νέου έλεγχος.

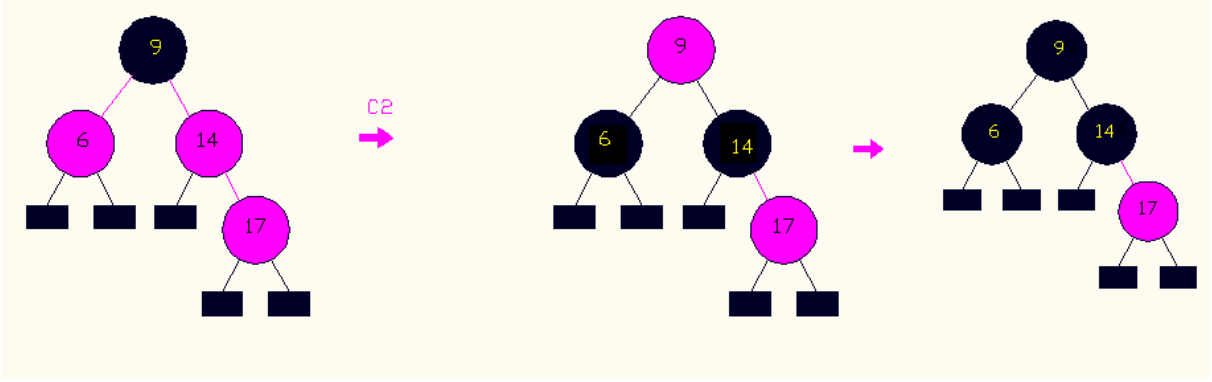
Ακολουθεί η περιγραφή σε ψευδογλώσσα:

```
Algorithm INSERTITEM(Item i)
Input:    Ένα στοιχείο i
Output:   Ο κόμβος που το i τοποθετείται εάν δεν υπάρχει ήδη
1.  Κλήση της αντίστοιχης μεθόδου των απλών δυαδικών δένδρων ώστε να εισαχθεί το  i  σε
    νέο κόκκινο κόμβο με μαύρα φύλλα;
2.  if (δεν δημιουργήθηκαν δυο διαδοχικοί κόκκινοι κόμβοι)
3.     return;
4.  while(υπάρχουν δυο διαδοχικοί κόκκινοι κόμβοι) {
5.     ανεβαίνουμε προς τα επάνω εκτελώντας όσο είναι δυνατόν πράξεις C2;
6.     ενδεχομένως εκτελούμε και μία C1;
7.  }
8.  if(η ρίζα r χρωματίστηκε κόκκινη)
9.    την βάφουμε μαύρη;
end of INSERTITEM
```

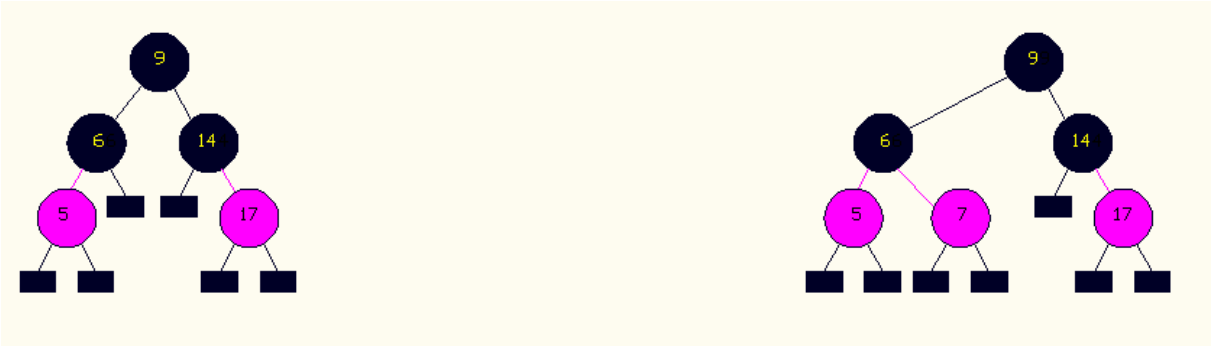
Τα Σχήματα 2.15–2.16 αποτελούν παράδειγμα 12 διαδοχικών ενθέσεων σε ένα αρχικά άδειο ερυθρόμαυρο δένδρο αναζήτησης. Αξίζει να παρατηρήσει κανείς πως οι επαναζυγιστικές πράξεις θεραπεύουν το πρόβλημα στις ενθέσεις (δ), (ι) και (ια):



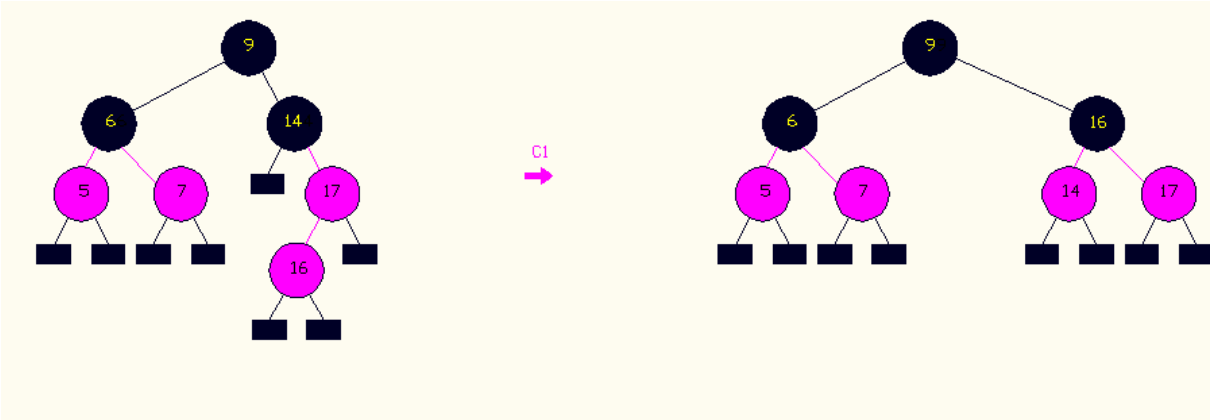
(α) (β) (γ)



(δ)



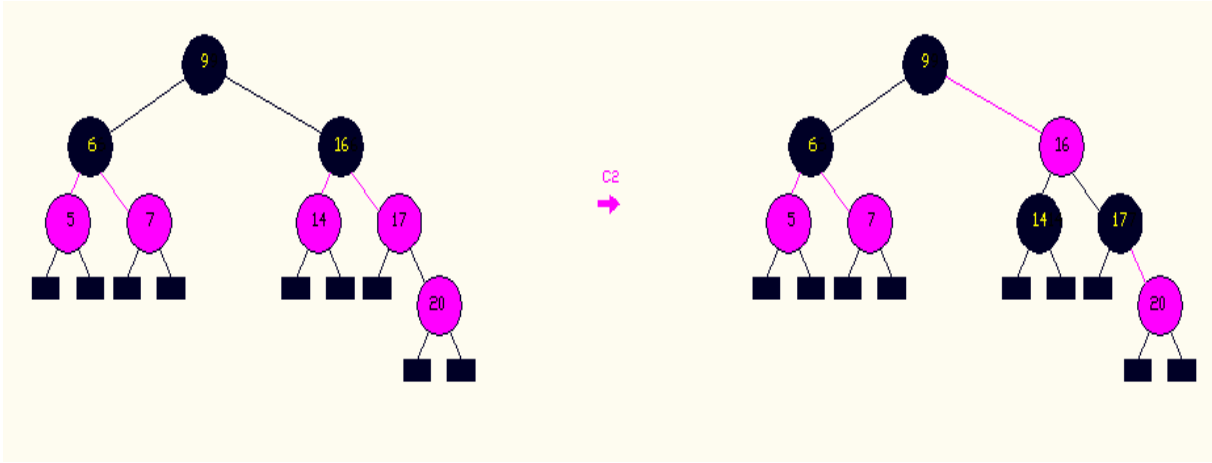
(ε) (στ)



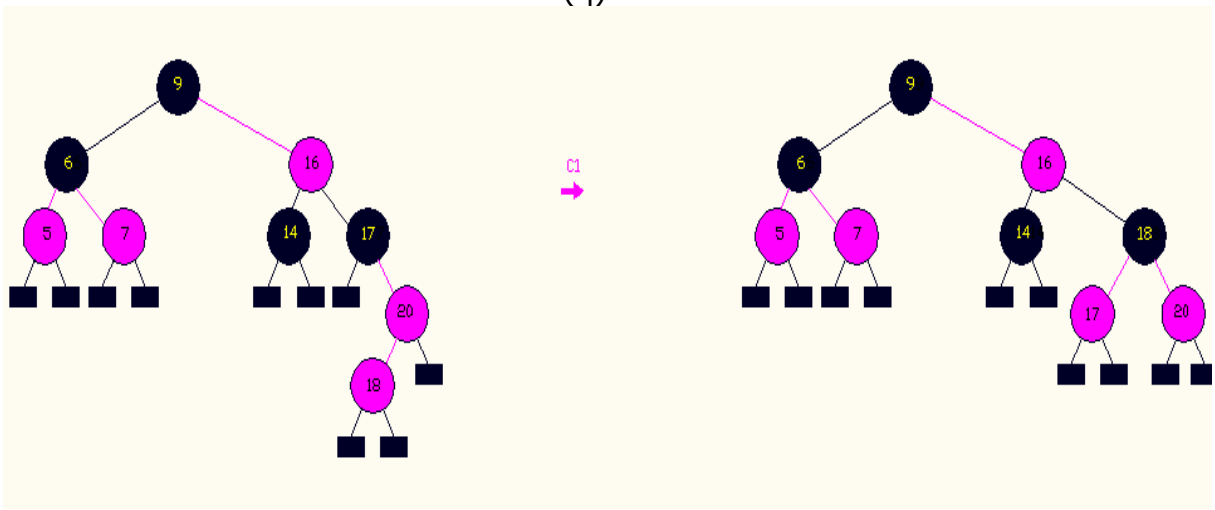
(ζ)

Σχήμα 2.15: (α)-(ζ) Διαδοχικές ενθέσεις των 6, 9, 14, 17, 5, 7, 16 20.

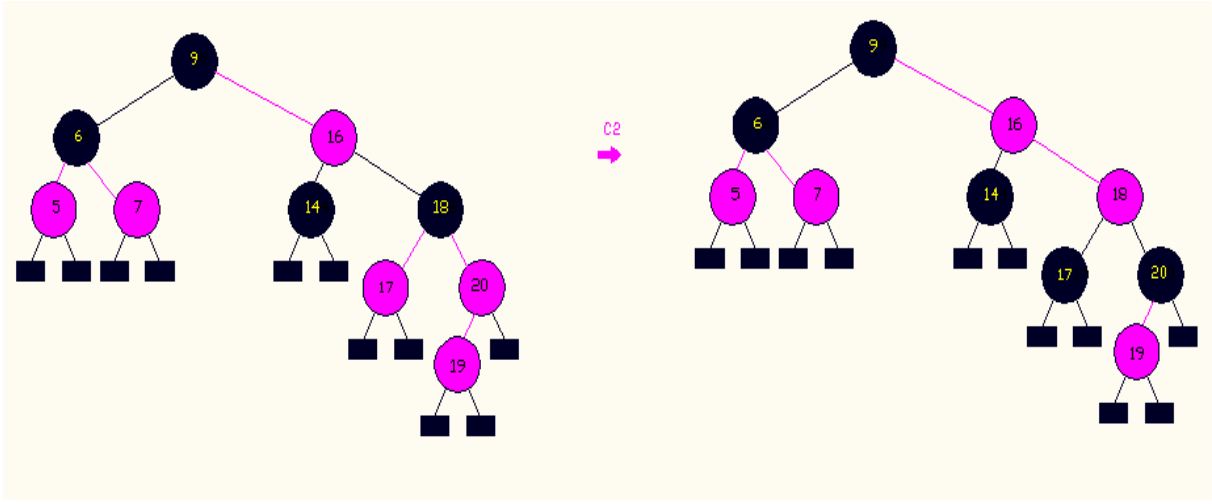
Στην πρώτη περίπτωση αφού εισαχθεί ο νέος κόκκινος κόμβος, που φέρει το 17 και δυο μαύρα φύλλα, εμφανίζονταν δυο διαδοχικοί κόκκινοι κόμβοι. Καθώς ο δεξιός αδελφός του 14 είναι κόκκινος εκτελείται αναχωματισμός (C2), που κοκκινίζει τη ρίζα την οποία βάφουμε αμέσως μαύρη.

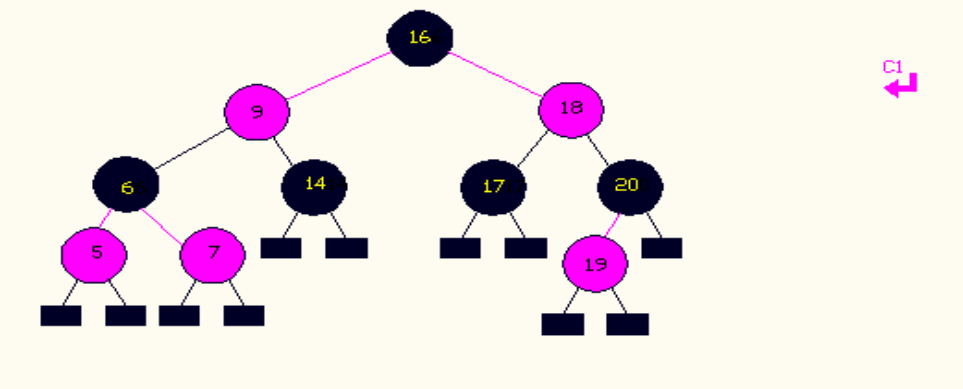

9
6
16
5
7
14
17
20
C2
9
6
16
5
7
14
17
20

(η)


9
6
16
5
7
14
17
20
18
C1
9
6
16
5
7
14
18
17
20

(θ)


9
6
16
5
7
14
18
17
20
19
C2
9
6
16
5
7
14
18
17
20
19

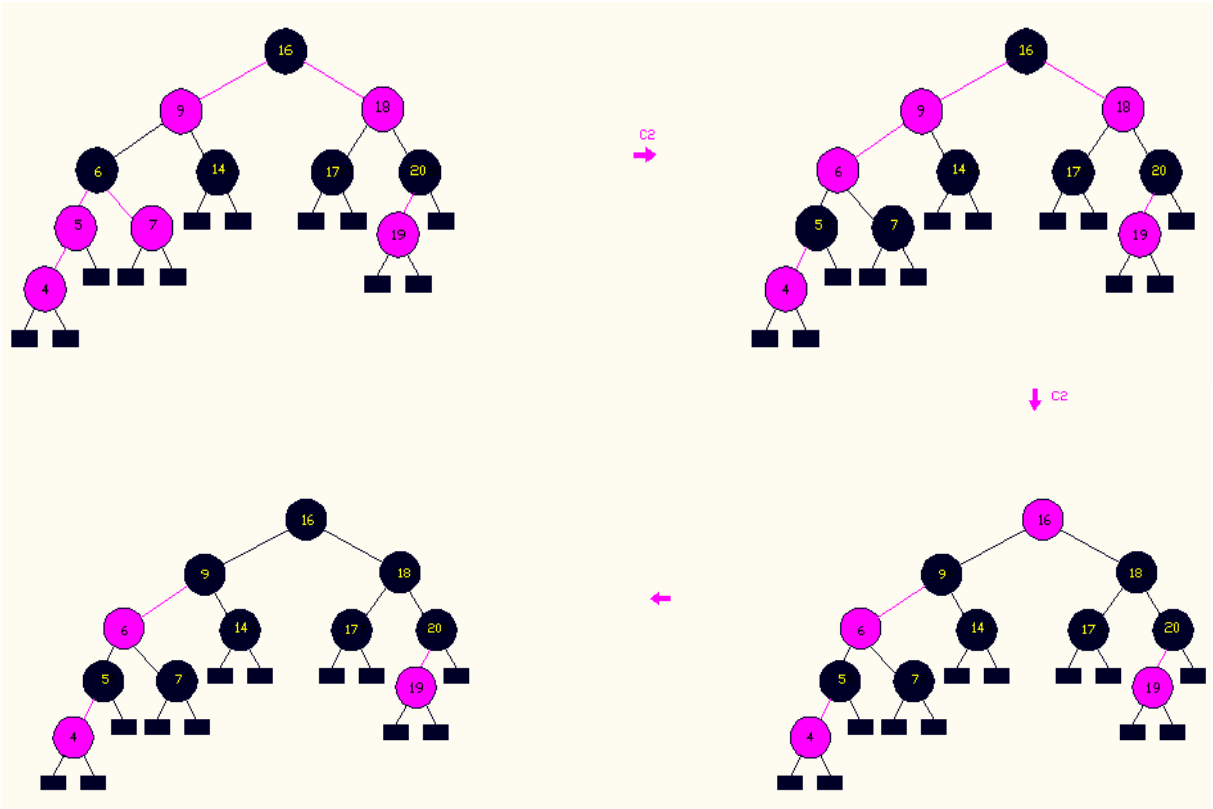


(ι)

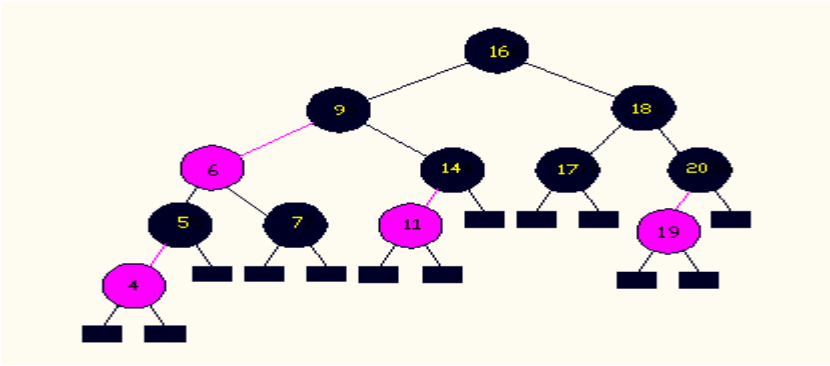
Σχήμα 2.16: (Συνέχεια) (η)-(ι)20, 18, 19

Κατά την ένθεση του 19 (περίπτωση(ι)), δημιουργούνται οι συνθήκες του αναχρωματισμού του κανόνα C2, που, όμως, μεταθέτει το πρόβλημα στους κόμβους 16 και 18. Ο 16 έχει μαύρο αδελφό, οπότε εμφανίζουμε αριστερή δεξιά περιστροφή στην ρίζα επιλύοντας το πρόβλημα με μεταφορά του πλεονάζοντος κόκκινου κόμβου στο αριστερό υποδένδρο.

Τέλος η εισαγωγή του 4 (στιγμιότυπο (ια)), ως ακραίου αριστερού κόμβου, προκαλεί την εμφάνιση δυο κόκκινων κόμβων. Ο 5 έχει κόκκινο δεξιό αδελφό (τον 7). Κατά συνέπεια, λαμβάνει χώρα ανταλλαγή χρωμάτων που, απλώς ανάγει το πρόβλημα στους 6 και 9. Καθώς ο 18 είναι κόκκινος εμφανίζουμε ξανά την πράξη C2 η οποία κοκκινίζει την ρίζα, ως εκ τούτου, την μαυρίζουμε και η ισοζύγιση αποκαθίσταται.



(ια)

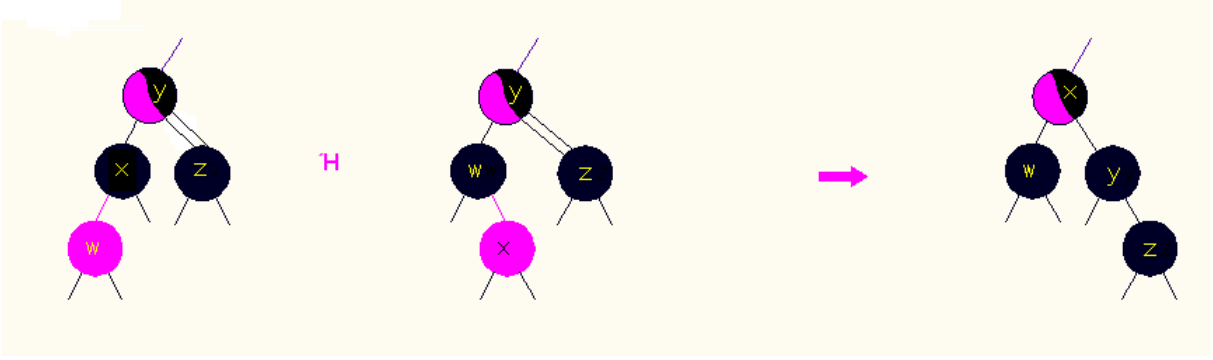


(ιβ)

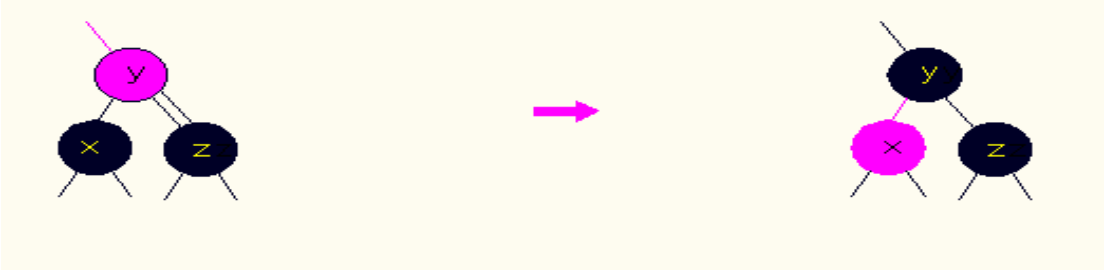
Σχήμα 2.17: (Συνέχεια) (ια)-(ιβ) 4, 11.

Απόσβεση Στοιχείου. Η διαδικασία της απόσβεσης, αφού καλέσει την αντίστοιχη πράξη των αζύγιστων δυαδικών δένδρων, εξετάζει το είδος του αφαιρεθέντος κόμβου. Εάν είναι κόκκινος τότε δεν χρειάζεται να πράξομε τίποτε. Εάν όμως είναι μαύρος τότε έχουμε παραβίαση της ιδιότητας (δ) των ερυθρόμαυρων δένδρων, καθώς απωλέσθη ένας μαύρος κόμβος επί του μονοπατιού απόσβεσης. Εάν επιπροσθέτως ως αποτέλεσμα της διαγραφής, εμφανίστηκαν δυο διαδοχικοί κόκκινοι κόμβοι, τότε το (διπλό) πρόβλημα επιλύεται με το μαύρισμα του χαμηλότερου από αυτούς.

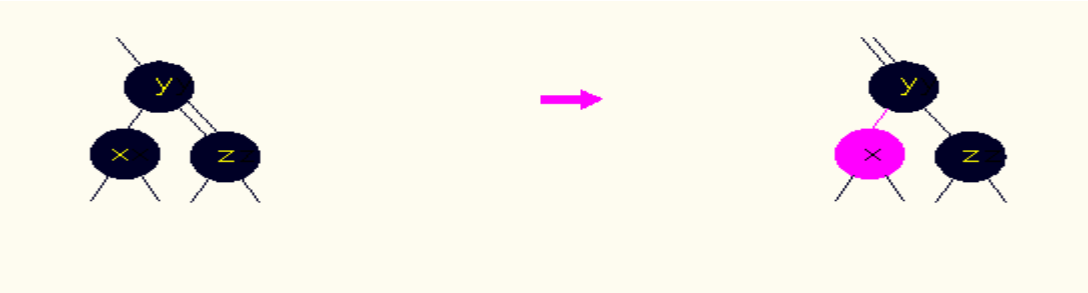
Διαφορετικά εφαρμόζουμε τις επαναζυγιστικές πράξεις του σχήματος 2.19 όπου με διπλή γραμμή \\ συμβολίζομε την «ελλειμματική» μαύρη ακμή e, η οποία προκύπτει από την απώλεια του μαύρου κόμβου που υπήρχε μεταξύ των κόμβων που συνδέει η e. Στο σχήμα αυτό δε εικονίζονται οι συμμετρικές περιπτώσεις.



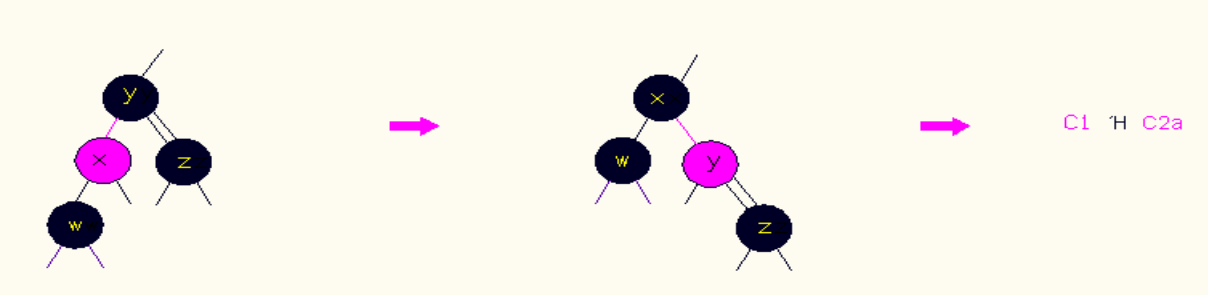
(α)



(β)



(γ)



(δ)

Σχήμα 2.18: Επαναζυγιστικές πράξεις απόσβεσης ερυθρόμαυρου δένδρου. Η περιστροφή C1, (β) τερματικός αναχρωματισμός C2a, (γ) μη τερματικός αναχρωματισμός C2b, και (δ) περιστροφή C3 που οδηγεί σε τερματική περίπτωση C1 ή C2a.

Από αυτές,

Η C2b (γ) μεταθέτει το πρόβλημα στον πατέρα, μειώνοντας τον αριθμό των μαύρων κόμβων στο αδελφό μονοπάτι, δίχως δομικές αλλαγές. Οι υπόλοιπες πράξεις είναι τερματικές. Και συγκεκριμένα:

Η C1 (α) επαναφέρει την ισορροπία, μετακινώντας με απλή διπλή περιστροφή τον κόκκινο – και άρα, με έχοντα επίδραση επί του μαύρου βάθους – κόμβο στο ελλειμματικό υποδένδρο, ως μαύρο πλέον, διατηρώντας, παράλληλα, το όποιο χρώμα του παλαιού κορυφαίου κόμβου (ο y στο Σχήμα) και στον νέο κορυφαίο κόμβο που προκύπτει μετά την περιστροφή (ο x στο Σχήμα).

Η C2a (β) αποκαθιστά το πλήθος των μαύρων κόμβων στον ελλειμματικό μονοπάτι χρωματίζοντας μαύρο τον κόκκινο πατέρα, αφαιρώντας, όμως , έναν μαύρο κόμβο από το αδελφό μονοπάτι, μέσω του κοκκινίσματος του μαύρου αδελφού, ώστε έτσι να μην διαταραχθεί το μαύρο βάθος των άλλων φύλλων – απογόνων του.

Η C3 (δ), διενεργώντας απλή περιστροφή δημιουργεί τις συνθήκες για την εφαρμογή είτε της C1 είτε της C2a, αναλόγως του χρώμματος του νέου αδελφού του ελλειμματικού κόμβου και επομένως αποτελεί τερματική πράξη.

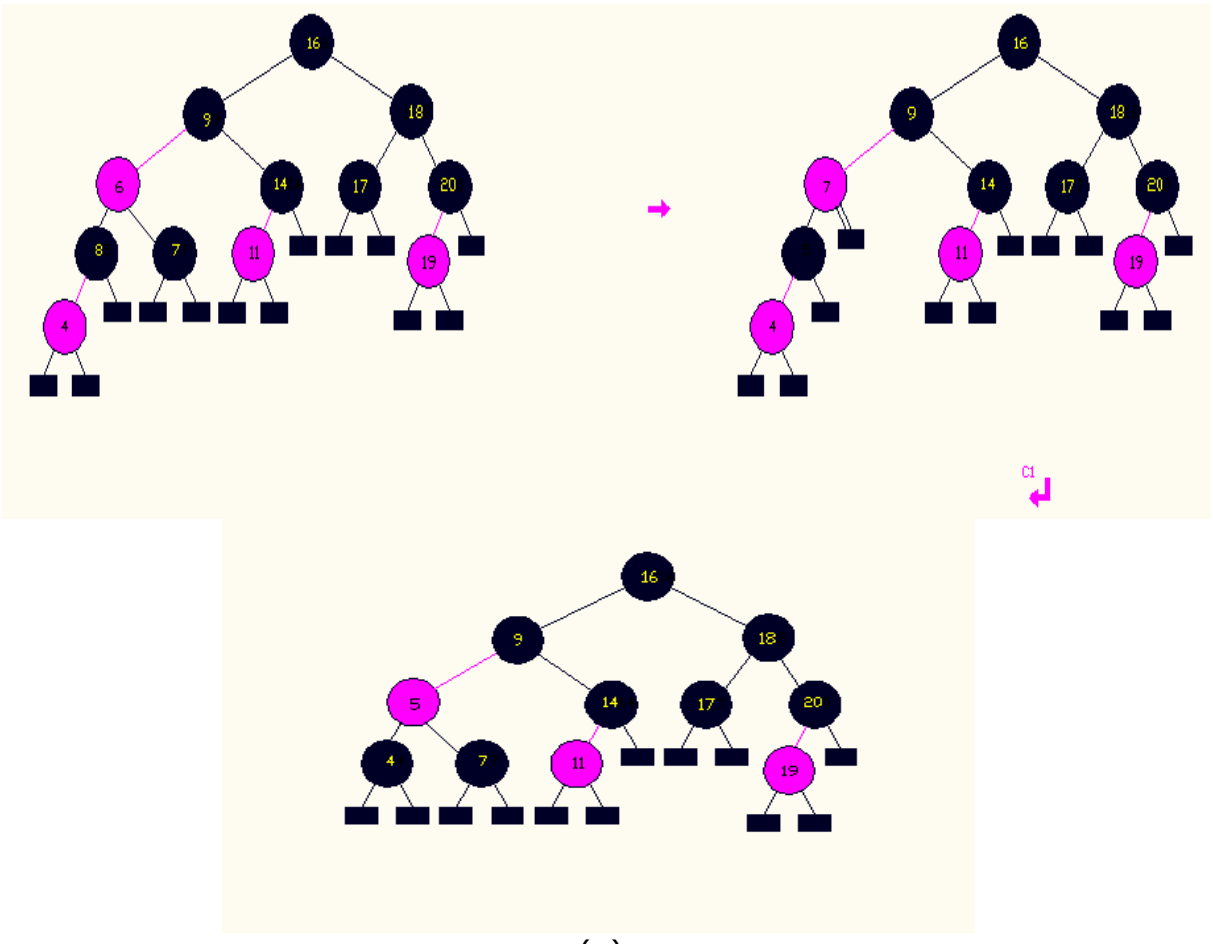
Τυπικότερα έχουμε:

Algorithm DELETEITEM(Object k)
Input: Ένα κλειδί k
Output: Η απόσβεση του κόμβου που περιέχει item με κλειδί k, επιστρέφοντας τον

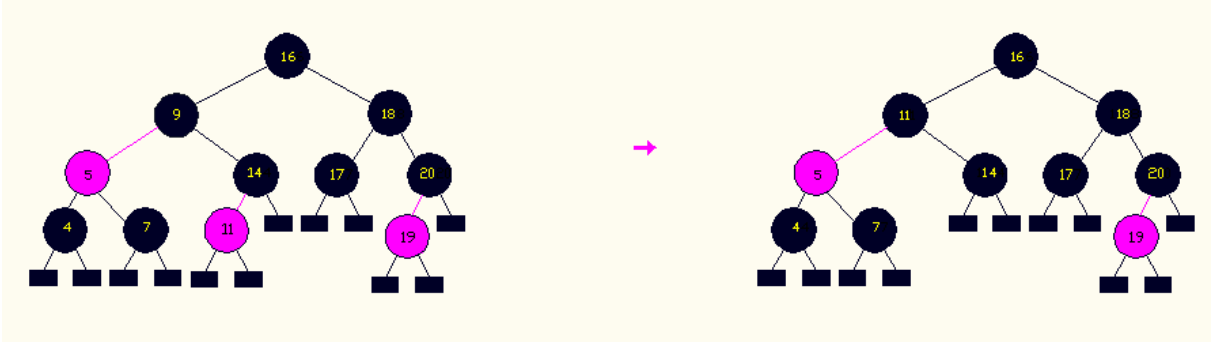
εναπομείναντα εμπλεκόμενο κόμβο.
1. κλήση της αντίστοιχής μεθόδου των απλών δυαδικών δένδρων;
2. if (σβήστηκε κόκκινος κόμβος)
3. return;
4. else if(προέκυψαν δυο διαδοχικοί κόκκινοι)
5. μαυρίζουμε τον χαμηλότερο; // απορρόφηση απώλειας μαύρου κόμβου
6. else // έχουμε απώλεια μαύρου κόμβου εν εκκρεμότητα
7. while(δεν φθάσαμε στην ρίζα και δεν αναπληρώθηκε η απώλεια)
8. ανεβαίνουμε το δένδρο εκτελώντας κατά περίπτωση τις επαναζυγιστικές πράξεις του Σχήματος 2.19;

end of DELETEITEM

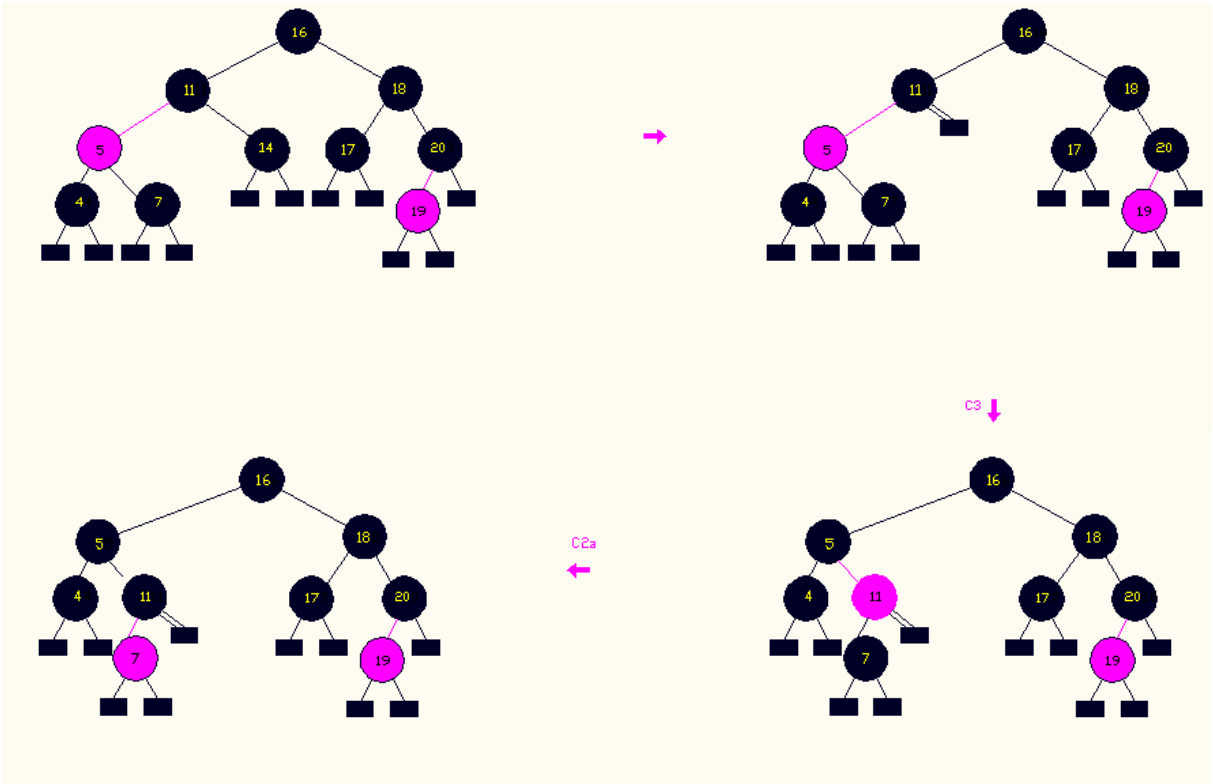
Στο σχήμα 2.19 παρουσιάζονται χαρακτηριστικές περιπτώσεις διαγραφής στο δέντρο του σχήματος 2.17(ιβ). Λόγου χάριν αυτή του 14 στιγμιότυπο (γ) προκαλεί εύρημα μαύρου κόμβου. Καθώς ο πατέρας του ελλειμματικού κόμβου είναι μαύρος και ο αριστερός αδελφός κόκκινος εφαρμόζεται η πράξη C3. Αυτό έχει ως αποτέλεσμα την απόκτηση κόκκινου πατέρα και μαύρου αριστερού αδελφού, γεγονός που επιτρέπει την διενέργεια τερματικής πράξης ανταλλαγής χρωμάτων C2a. Αξίζει επίσης να επισημανθεί η τελευταία περίπτωση (στ) της αποσβέσεως του 20. Εδώ η απόσβεση του μαύρου κόμβου προκαλεί την εμφάνιση δύο διαδοχικών κόμβων (18 και 19), ασσυμετρία που απορροφάται με μαύρισμα του χαμηλότερου 19.



(α)

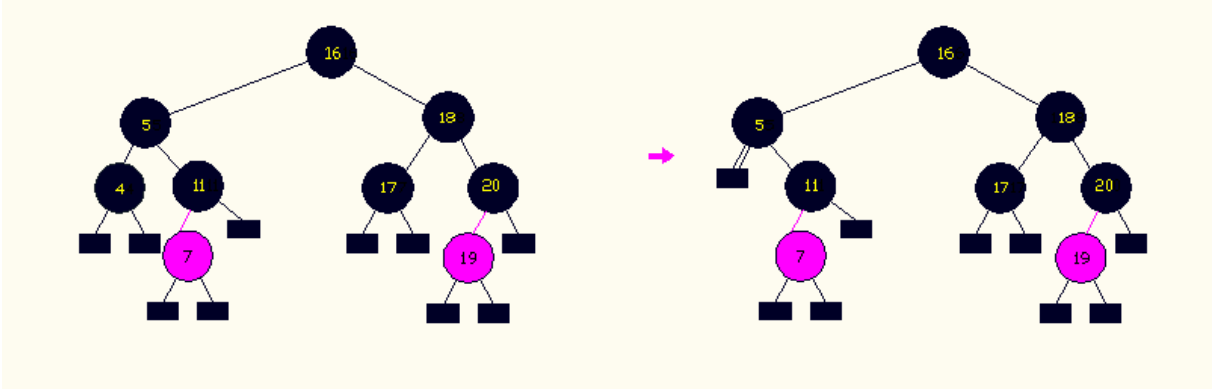


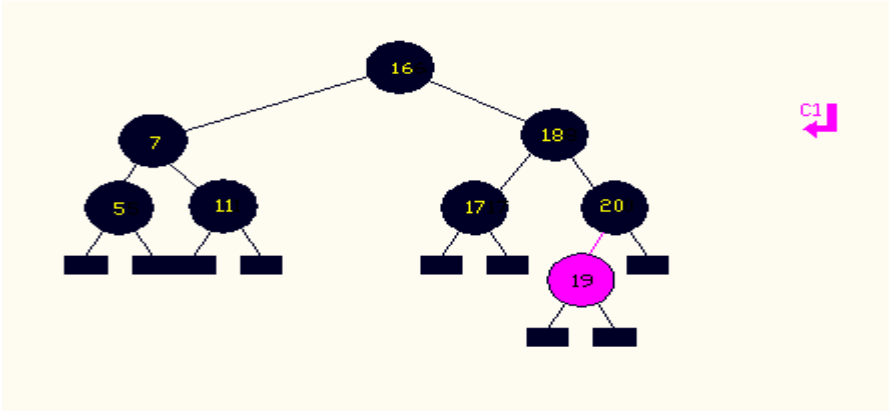
(β)



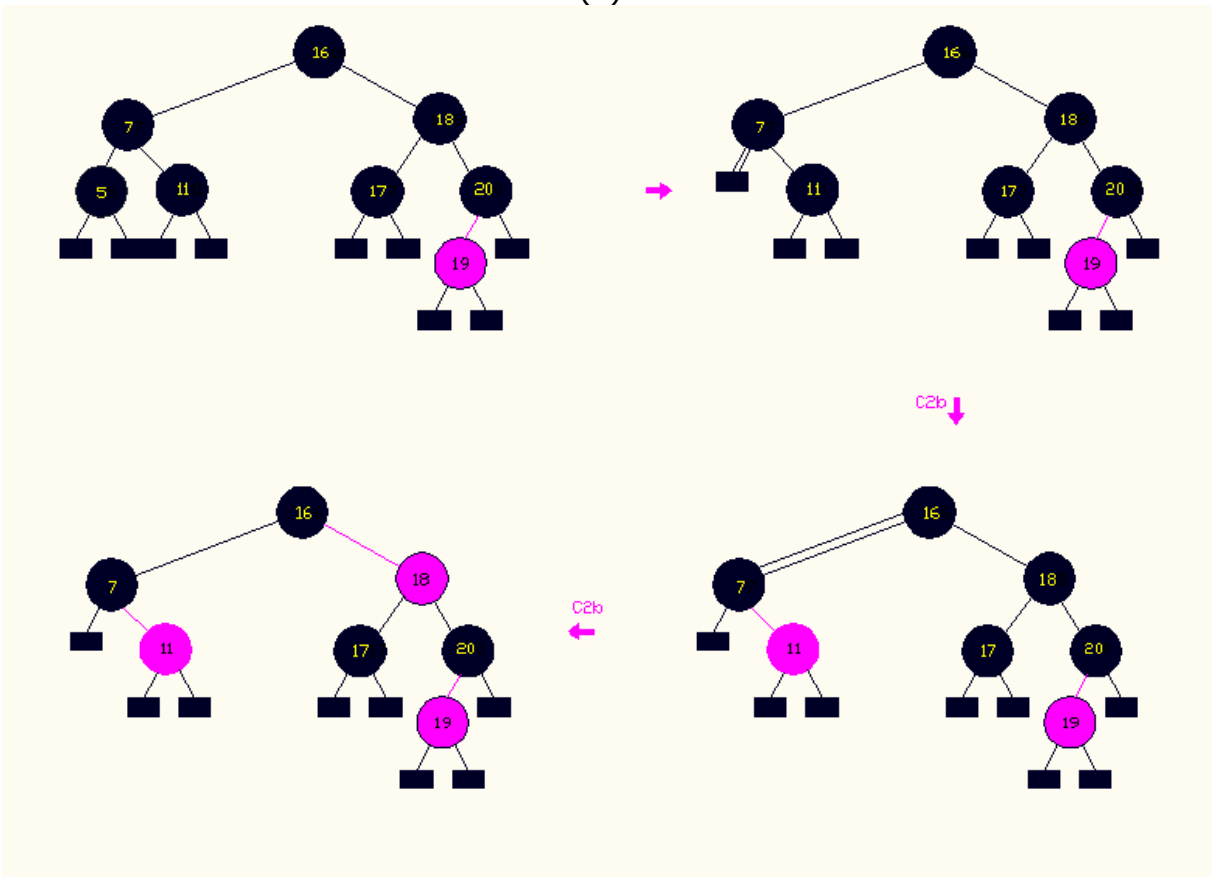
(γ)

Σχήμα 2.19:(α)-(γ) Διαδοχικές αποσβέσεις των 6, 9, 14



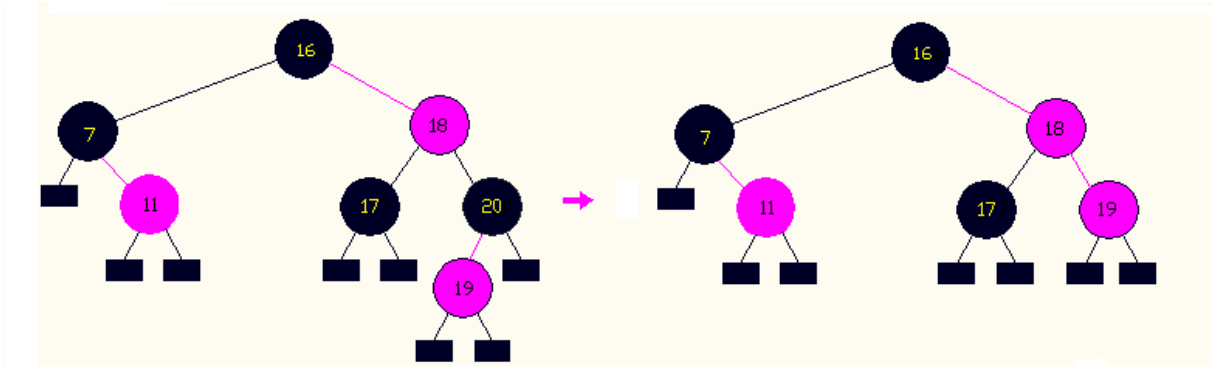


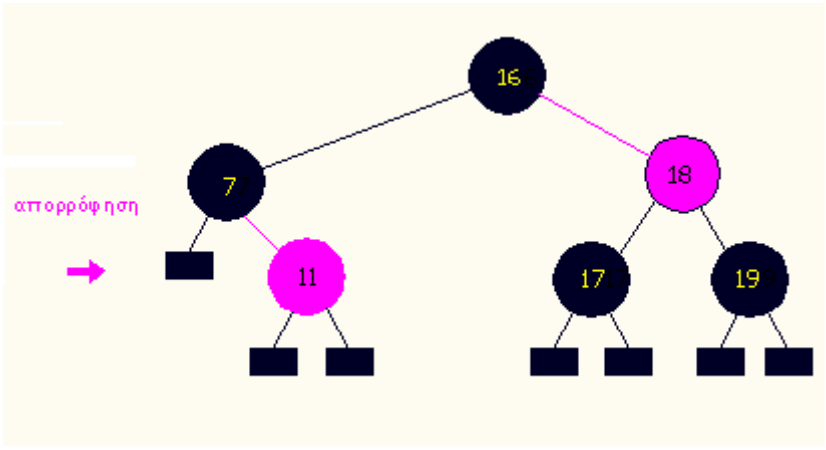
(δ)



(ε)

Σχήμα 2.19:(Συνέχεια)(δ)-(ε) Διαδοχικές αποσβέσεις των 4, 5.





(στ)

Σχήμα 2.21: (στ) απόσβεση του 20.

2.3.3 Ανάλυση Πολυπλοκότητας.

Λήμμα 2.6 Το ύψος h ενός ερυθρόμαυρου δένδρου Τ επί ενός συνόλου n στοιχείων είναι Θ(log n).

Λήμμα 2.7 Μια ένθεση (απόσβεση) σε ένα ερυθρόμαυρο δένδρο Τ, n στοιχείων απαιτεί Ο(1) δομικές πράξεις και Ο(log n) αναχρωματισμούς.

Θεώρημα 2.3 Μια ακολουθία n αναμεμειγμένων ενθέσεων και αποσβέσεων, σε ένα αρχικά άδειο ερυθρόμαυρο δένδρο Τ απαιτεί Ο(n) επαναζυγιστικές πράξεις. Ένας κόμβος υ δε, ύψους l θα δεχθεί $O(n/c^l)$ πράξεις.

3. RANK-BALANCED TREES

3.1 Εισαγωγή

Τα ισορροπημένα δένδρα αναζήτησης είναι θεμελιώδη και πανταχού παρόντα στην επιστήμη των υπολογιστών. Από την ανακάλυψη των AVL trees το 1962 έχουν προταθεί πολλές εναλλακτικές λύσεις με σκοπό είτε την απλούστερη υλοποίηση τους, είτε την ταχύτερη απόδοση ή και τα δυο. Απλούστερες υλοποιήσεις ισορροπημένων δένδρων αποτελούν η υλοποίηση του Andersson, των δυαδικών B-trees του Bayer, καθώς και η υλοποίηση του Sedgewick των related left–leaning ερυθρόμαυρων δέντρων. Αυτές οι δομές δεδομένων είναι μη συμμετρικές, οι οποίες απλοποιούν την επαναζύγιση εξαλείφοντας κατά προσέγγιση τις μισές περιπτώσεις. Ο Andersson απλοποίησε περαιτέρω την υλοποίηση παράγοντας την επαναζύγιση σε δυο διαδικασίες, "skew and split" αλλά προσθέτοντας και άλλες έξυπνες ιδέες. Τα τυποποιημένα ερυθρόμαυρα δένδρα, από την άλλη πλευρά, έχουν αλγορίθμους ενημέρωσης που εγγυώνται την αποδοτικότητα: η επαναζύγιση μετά από μία εισαγωγή ή διαγραφή κοστίζει στη χειρότερη περίπτωση O(1) περιστροφές και O(1) κατανεμημένο χρόνο. Ως αποτέλεσμα αυτών των αναλύσεων, ένας συγγραφέας είπε: «AVL …trees are now passé».

Η σχεδίαση και ανάλυση των ισορροπημένων δένδρων είναι ένα γνωστικό αντικείμενο το οποία δεν έχει πλήρως εξερευνηθεί. Περιγράφουμε μια νέα σχεδίαση και ανάλυση και προτείνουμε ότι τα AVL δένδρα δεν είναι κάτι άλλο αλλά passé (ξεπερασμένα). Η νέα μας σχεδίαση είναι τα ***rank-balanced trees***, μια παραλλαγή των AVL δένδρων τα οποία έχουν παρόμοιες ιδιότητες με τα ερυθρόμαυρα δένδρα, αλλά είναι καλύτερα σε ορισμένους τρόπους. Ένα rank-balanced tree χωρίς την πράξη της διαγραφής είναι ακριβώς όπως ένα AVL δένδρο, ενώ με διαγραφές το ύψος του είναι το πολύ ίσο με το ύψος ενός AVL δένδρου με τον ίδιο αριθμό εισαγωγών, αλλά χωρίς διαγραφές. Τα rank-balanced trees είναι ένα κανονικό υποσύνολο των ερυθρόμαυρων δένδρων με ένα διαφορετικό κανόνα ισορρόπησης και διαφορετικούς αλγόριθμους επαναζύγισης. Η εισαγωγή και η διαγραφή κοστίζουν το πολύ δυο περιστροφές στη χειρότερη περίπτωση και O(1) κατανεμημένο χρόνο. Τα ερυθρόμαυρα δένδρα χρειάζονται τρεις περιστροφές στη χειρότερη περίπτωση για μια διαγραφή. Η εισαγωγή και η διαγραφή μπορεί να γίνουν top-down με σταθερό lookahead σε O(1) κατανεμημένο χρόνο ισορρόπησης για κάθε ενημέρωση.

Η νέα μας ανάλυση χρησιμοποιεί μια εκθετική συνάρτηση δυναμικού για να μετρήσουμε την κατανεμημένη αποδοτικότητα των λειτουργιών σε ένα ισορροπημένο δένδρο σε συνάρτηση με το ύψος των κόμβων του. Χρησιμοποιούμε αυτή τη μέθοδο για να δείξουμε ότι η επαναζύγιση στα rank-balanced trees επηρεάζει τους κόμβους εκθετικά σπάνια στα ύψη τους. Αυτό συμβαίνει τόσο για τη top-down όσο και για την bottom-up επαναζύγιση.

Ορολογία των δένδρων. Ένα δυαδικό δένδρο 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 ποία 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 μοναδικός κόμβος χωρίς πατέρα. Φύλλο είναι ένας κόμβος με τα δυο παιδιά του να λείπουν. Η σχέση προγονού είναι το ανακλαστικό μεταβατικό σύνολο της σχέσης γονέα. Αντίστοιχα ισχύει και για τη σχέση απογόνου και παιδιού. Εάν ο κόμβος x είναι ένας πρόγονος ενός κόμβου y και y ≠ x, ο x είναι ένα κατάλληλος πρόγονος του y και ο y είναι ένας κατάλληλος απόγονος του x. Εάν x είναι ένας κόμβος το αριστερό, δεξιό υποδένδρο είναι το δυαδικό δένδρο που περιέχει όλους τους απογόνους του left(x), του right(x), αντίστοιχα. Το ύψος ***h(x)*** ενός κόμβου x ορίζεται αναδρομικά ως h(x) = 0 εάν το x είναι φύλλο $h(x) = \max\{h(left(x), h(rigth(x))\} + 1$ διαφορετικά. Τα ύψος h ενός δένδρου είναι το ύψος της ρίζας τους.

Τα δυαδικά δένδρα εμφανίζουν ιδιαίτερο ενδιαφέρον ως δένδρα αναζήτησης. Ένα δυαδικό δένδρο αναζήτησης αποθηκεύει ένα σύνολο στοιχείων καθένα από τα οποία έχει ένα κλειδί, το οποίο έχει επιλεγεί από ένα ολικά διατεταγμένο σύνολο. Θα υποθέσουμε ότι κάθε στοιχείο έχει ένα διαφορετικό κλειδί, διαφορετικά το επιλέγουμε με βάση τον προσδιοριστή στοιχείων. Σε ένα ***εσωτερικό δυαδικό δένδρο αναζήτησης*** κάθε κόμβος είναι ένα στοιχειό και τα στοιχεία είναι τακτοποιημένα σε συμμετρική σειρά. Το κλειδί ενός κόμβου x είναι μεγαλύτερο, μικρότερο από όλα τα κλειδιά των στοιχείων του αριστερού, δεξιού υποδένδρου, αντίστοιχα. Δοθέντος ενός τέτοιου δένδρου και ενός κλειδιού, μπορούμε να αναζητήσουμε το στοιχείο έχοντας αυτό το κλειδί και συγκρίνοντας το με αυτό της ρίζας. Εάν είναι ίσα έχουμε βρει το επιθυμητό στοιχείο. Εάν το κλειδί αναζήτησης είναι μικρότερο ή μεγαλύτερο ψάχνουμε αναδρομικά στο αριστερό ή στο δεξί υποδένδρο της ρίζας, αντίστοιχα. Κάθε κλειδί σύγκρισης είναι και ένα βήμα της αναζήτησης. Ο τωρινός κόμβος είναι αυτός, του οποίου το κλειδί συγκρίνεται με το κλειδί της αναζήτησης. Τελικά η αναζήτηση είτε εντοπίζει το επιθυμητό στοιχειό είτε καταλήγει σε ένα ελλείπων κόμβο, καταλήγοντας στο αριστερό ή στο δεξί παιδί του τελευταίου κόμβου που συναντήσαμε από την αναζήτηση στο δένδρο.

Για την εισαγωγή ενός νέου στοιχείου σε ένα τέτοιο δένδρο κάνουμε πρώτα αναζήτηση με βάση το κλειδί του. Μόλις η αναζήτηση φτάσει σε ένα ελλειπόν κόμβο(missing node) αντικαθιστούμε αυτόν τον κόμβο με το νέο στοιχείο. Η διαγραφή είναι λίγο δυσκολότερη. Πρώτα βρίσκουμε το στοιχείο που πρόκειται να διαγραφεί κάνοντας μια αναζήτηση με βάση το κλειδί του. Εάν δεν λείπει κανένα παιδί από το στοιχείο που πρόκειται να διαγραφεί, βρίσκουμε είτε το επόμενο στοιχείο είτε το προηγούμενο στοιχείο διατρέχοντας τα αριστερά ,τα δεξιά παιδιά του δεξιού αντίστοιχα αριστερού παιδιού του στοιχείου μέχρι να φθάσουμε ένα κόμβο με ελλείπων αριστερό αντίστοιχα δεξί παιδί. Ανταλλάσσουμε το στοιχείο με το στοιχείο του βρίσκουμε. Τώρα το στοιχείο που πρόκειται να διαγραφεί είναι φύλλο ή έχει ένα ελλείπον παιδί. Στην πρώτη περίπτωση το αντικαθιστούμε από ένα ελλείπον κόμβο, ενώ στην δεύτερη περίπτωση το αντικαθιστούμε από το μη–ελλείπον παιδί του(non-missing node). Μια πρόσβαση, εισαγωγή ή διαγραφή κοστίζει O(h+1) χρόνο στην χειρότερη περίπτωση, όπου h είναι το ύψος του δένδρου.

3.2 Rank-Balanced Trees

Για να είναι η πράξη της αναζήτησης, της εισαγωγής και της διαγραφής αποδοτική περιορίζουμε το ύψος του δέντρου επιβάλλοντας ένα ***rank rule*** στο

δέντρο. Ένα ***ranked binary tree*** είναι ένα δυαδικό δέντρο όπου κάθε ένας από τους κόμβους του, x, έχει έναν ακέραιο ***rank r(x)***. Υιοθετούμε τη σύμβαση ότι οι ***missing nodes(ελλειπόντες κόμβοι)*** έχουν rank **-1**. Το rank ενός ranked binary tree είναι το rank της ρίζας του. Εάν x είναι ένας κόμβος με πατέρα p(x), το ***rank difference*** του x είναι ***r(p(x)) - r(x)***. Ονομάζουμε έναν κόμβο ως ***i-child***, εάν το rank difference αυτού του κόμβου είναι i και ως ***i,j-node*** εάν τα παιδιά του έχουν rank difference i και j. Ο τελευταίος ορισμός δεν διακρίνει μεταξύ αριστερών και δεξιών παιδιών και επιτρέπει τα παιδιά να είναι missing nodes.

Ο αρχικός μας rank rule είναι ότι κάθε κόμβος πρέπει να είναι 1,1-node ή 1,2-node. Αυτός ο κανόνας μας δίνει ακριβώς τα AVL trees: κάθε φύλλο είναι ένας 1,1-node με rank μηδέν, το rank κάθε κόμβου είναι το ύψος του και το αριστερό και το δεξί υποδέντρο έχουν ύψη που διαφέρουν το πολύ κατά ένα. Για να κωδικοποιήσουμε το rank αποθηκεύουμε με κάθε non-root κόμβο ένα bit που δείχνει εάν είναι 1-child ή 2-child. Ο rank rule εγγυάται ένα λογαριθμικό όριο ύψους. Ειδικότερα ο ελάχιστος αριθμός από κόμβους n_k σε ένα AVL tree με rank κ ικανοποιεί την επανάληψη $n_0=1$, $n_1=2$, $n_k=1+n_{k+1}+n_{k-2}$ για $k>1$. Αυτή η επανάληψη δίνει $n_k=F_{k+3}-1$, όπου F_k είναι ο k^{th} Fibonacci number. Από $F_{k+2}\geq\varphi^k$ όπου $\varphi=(1+\sqrt{5})/2$ είναι η χρυσή αναλογία, $k\leq\log_\varphi n\leq 1.4404\lg n$.

Τα AVL trees υποστηρίζουν την αναζήτηση σε O(logn) χρόνο, αλλά μία εισαγωγή ή διαγραφή μπορούν να προκαλέσουν παραβίαση του rank rule. Για να διατηρήσουμε τον κανόνα, αλλάζουμε τα ranks συγκεκριμένων κόμβων και κάνουμε περιστροφές για να επαναζυγίσουμε το δέντρο. Το ***promotion*** , ***demotion*** ενός κόμβου x αυξάνει ή αντίστοιχα μειώνει το rank του κατά ένα. Μία περιστροφή σε ένα αριστερό παιδί x με γονέα y κάνει τον y δεξί παιδί του x ενώ διατηρείται η συμμετρική διάταξη. Μια περιστροφή κοστίζει O(1) χρόνο.

Στην περίπτωση μιας εισαγωγής, εάν ο γονέας του πρόσφατα εισερχόμενου κόμβου ήταν φύλλο ο νέος κόμβος θα έχει rank difference μηδέν και ως εκ τούτο παραβιάζει τον rank rule. Έστω λοιπόν ότι ο q είναι ο πρόσφατα εισερχόμενος κόμβος και έστω p είναι ο πατέρας του εάν υπάρχει, διαφορετικά έστω ότι είναι κενός. Αφού εισάγουμε τον κόμβο q επαναζυγίζουμε το δέντρο επαναλαμβάνοντας τα ακόλουθα βήματα μέχρι να συμβεί μία περίπτωση εκτός από promotion.

Επαναζυγιστικά βήματα μετά από εισαγωγή στον κόμβο p:

Stop: ο κόμβος p είναι κενός ή ο q δεν είναι 0-child. **Stop** (τερματική συνθήκη)

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

Promotion: Ο κόμβος s είναι 1-child. Κάνουμε promote(αύξηση του rank του κατά ένα) στον p. Αυτό επιδιορθώνει την παραβίαση στον q αλλά μπορεί να δημιουργήσει νέα παραβίαση του rank rule στον p. Ο κόμβος p έχει τώρα ακριβώς ένα παιδί με rank difference ένα, δηλαδή τον q. Αντικαθιστούμε τον q με

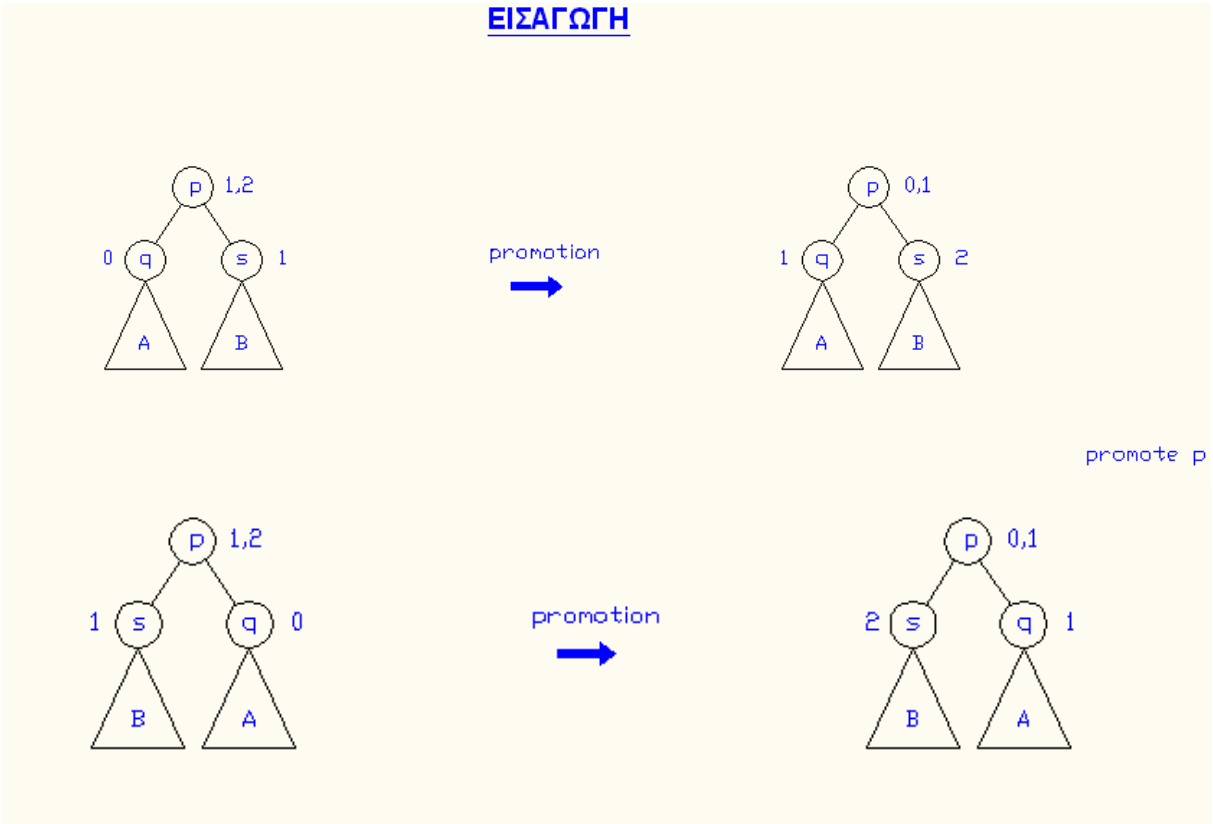
τον p. Έστω ότι ο p είναι ο πατέρας του q εάν αυτός υπάρχει, διαφορετικά είναι κενός.

Στις υπόλοιπες περιπτώσεις ο κόμβος s είναι 2-child. Υποθέτουμε ότι ο q είναι το αριστερό παιδί του p, ενώ η άλλη περίπτωση είναι συμμετρική. Έστω ο t είναι το δεξί παιδί του q, ο οποίος μπορεί να είναι ελλείπον.

Rotation: ο κόμβος t είναι 2-child. Κάνουμε rotate στον q και demote στον p. Αυτό επιδιορθώνει την παραβίαση του rank rule χωρίς να δημιουργεί κάποια καινούρια.**Stop**

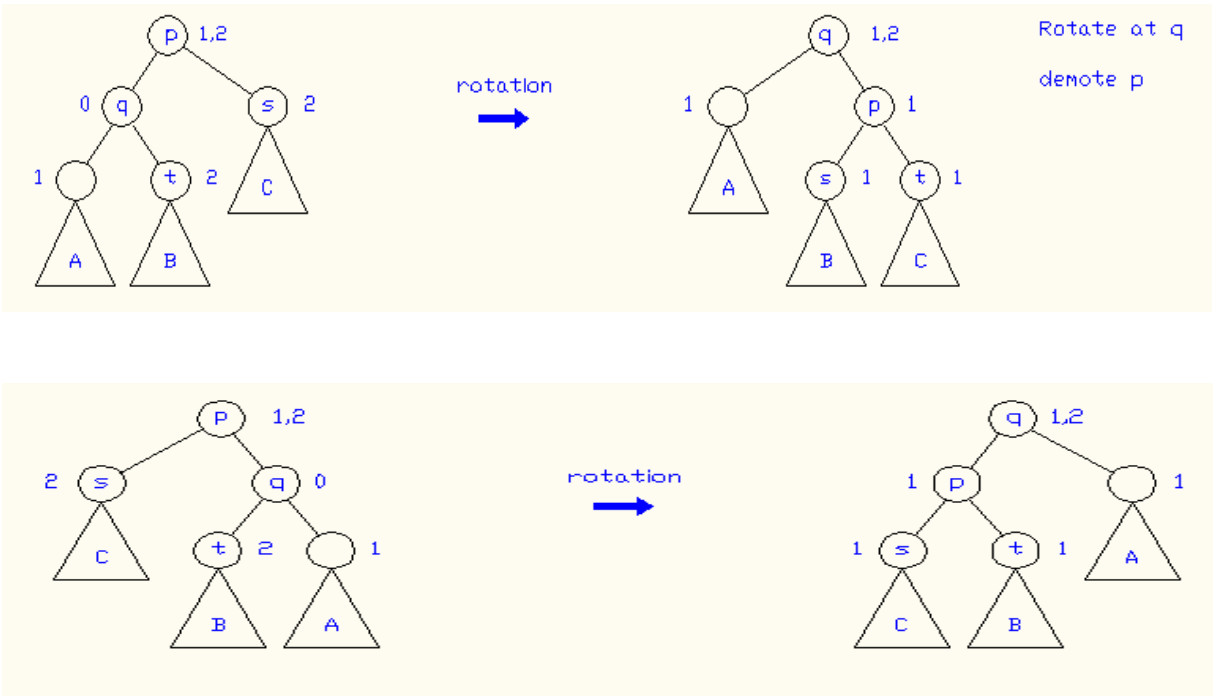
Double rotation: ο κόμβος t είναι ένας 1-child. Κάνουμε δύο φορές rotate στον t κάνοντας τον q αριστερό του παιδί και τον p δεξί παιδί του. Κάνουμε promote στον t και demote στον p και στον q. Αυτό επιδιορθώνει την παραβίαση χωρίς να δημιουργεί κάποια καινούρια.**Stop**

Κατά την διάρκεια της επαναζύγισης υπάρχει το πολύ μία παραβίαση του rank rule: ο κόμβος q μπορεί να είναι 0-child. Η επαναζύγιση γίνεται από το μονοπάτι του πρόσφατα εισερχόμενο κόμβο προς την ρίζα κάνοντας μηδέν ή περισσότερα βήματα promotion ακολουθούμενα από non-promotion βήματα. Το πρώτο βήμα είναι είτε stop είτε promotion. Μετά από ένα βήμα promotion ο κόμβος q είναι πάντα ένας 1,2-node.

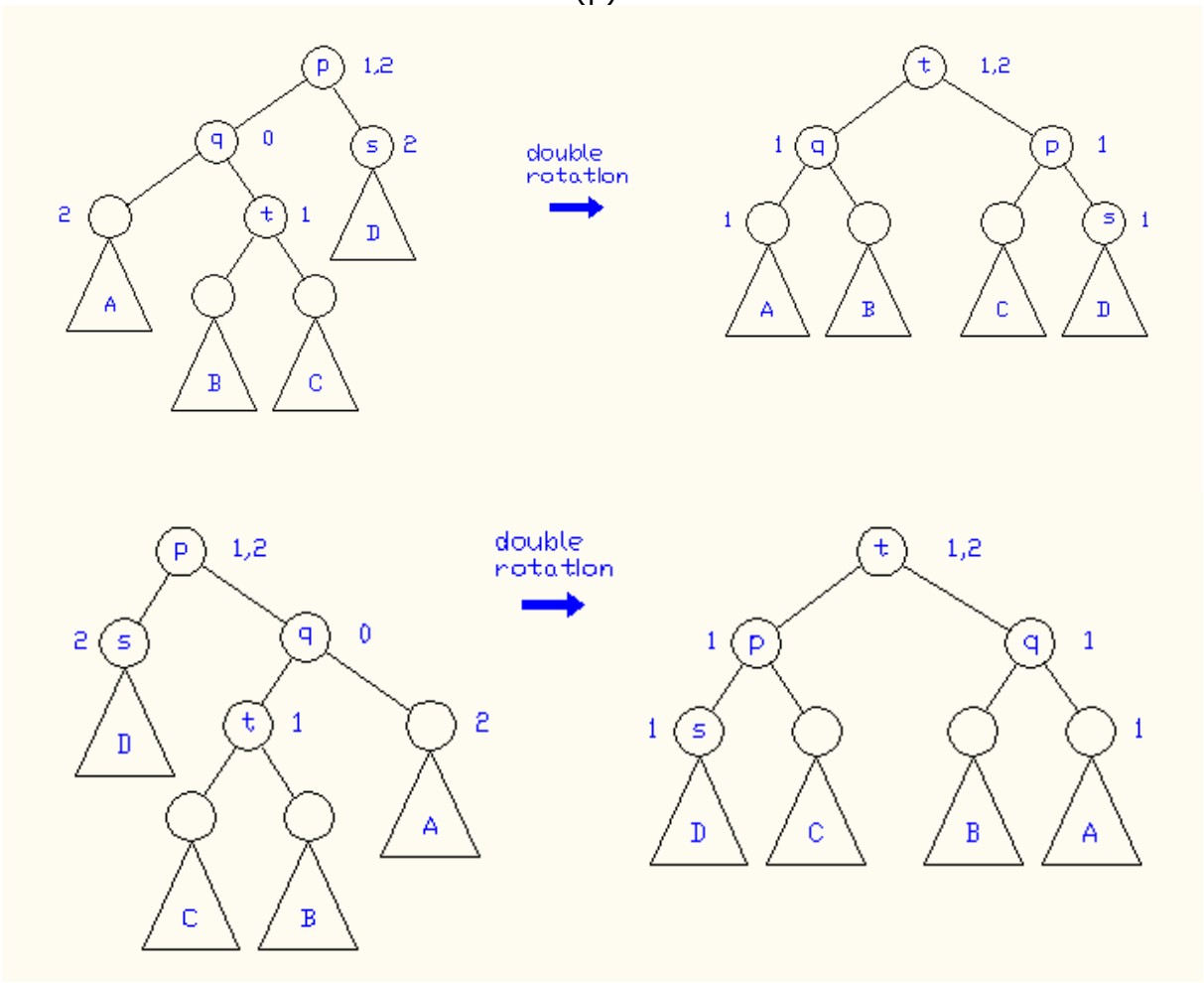


(α)

Σχήμα 3.1: Επαναζύγιση μετά από μία εισαγωγή. Οι αριθμοί είναι τα rank differences. Η περίπτωση (α) είναι μη τερματική.



(β)



(γ)

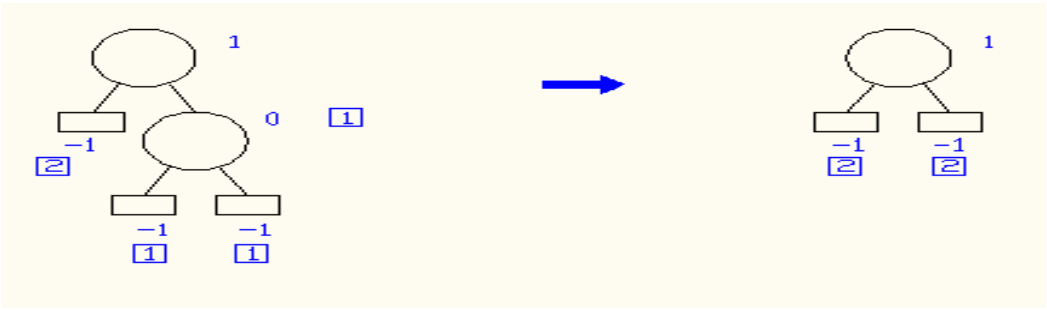
Σχήμα 3.2:(Συνέχεια) επαναζύγιση μετά από μία εισαγωγή. Οι αριθμοί είναι τα rank differences.

Η διαγραφή μπορεί να γίνει σε ένα AVL δέντρο με τον ίδιο τρόπο, αλλά η επαναζύγιση μπορεί να κοστίσει λογαριθμικό αριθμό περιστροφών, αντί για μία ή για δύο που χρειάζεται η εισαγωγή. Για να μειώσουμε αυτόν τον αριθμό αλλάζουμε τον rank rule και επιτρέπουμε non-leaf κόμβοι να είναι και 2-2 nodes όπως και 1,1- και 1,2-nodes. Τα φύλλα όμως πρέπει να είναι μόνο 1,1-nodes. Ονομάζουμε τα δέντρα που προκύπτουν ***rank-balanced trees*** ή ***rb-trees***. Ένα bit για κάθε non-root κόμβο είναι πάλι αρκετό για να κωδικοποιήσουμε το rank difference. Ένα AVL δέντρο λοιπόν είναι όπως ένα rb-tree χωρίς 2,2-nodes. Το rank ενός δέντρου είναι τουλάχιστον ίσο με το ύψος και το πολύ το διπλάσιο του.

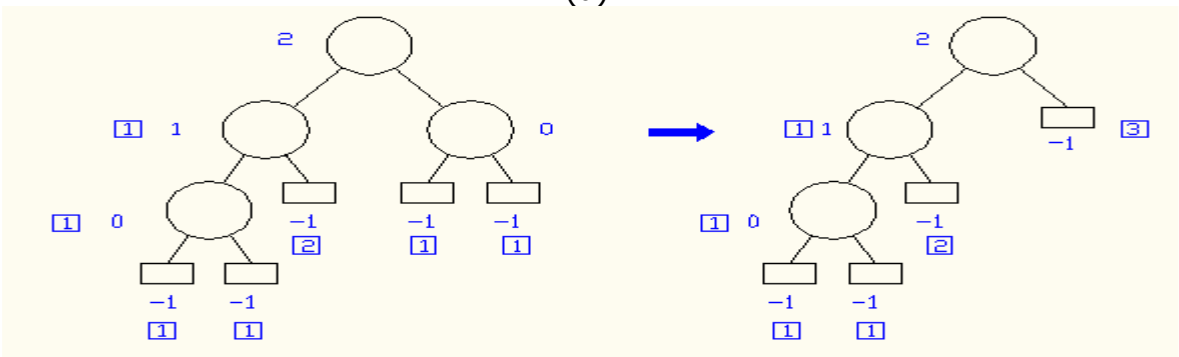
Θεώρημα 3.1 Το rank και ως εκ τούτο το ύψος ενός rb-tree είναι το πολύ 2logn.

Απόδειξη: ο ελάχιστος αριθμός κόμβων n_k σε ένα rb-tree με rank k ικανοποιεί την επανάληψη $n_0=1$, $n_1=2$, $n_k=1+2n_{k-2}$ για $k\geq 2$. Από την επαγωγή $n_k \geq 2^{k/2}$

Η εισαγωγή είναι ίδια στα rb-trees όπως και στα AVL trees: τα επαναζυγιστικά βήματα μετά από μία εισαγωγή δεν δημιουργούν 2,2-nodes(αλλά τους καταστρέφουν). Μια διαγραφή σε ένα rb-tree μπορεί να παραβιάσει τον rank rule δημιουργώντας έναν κόμβο με rank ένα με δύο missing παιδιά ή ένα κόμβο με rank δύο με ένα missing 3-child. Έστω q είναι ο κόμβος που αντικαθιστά το διαγραμμένο κόμβο(ο οποίος μπορεί να είναι missing node) και έστω p είναι ο γονέας του αν υπάρχει, διαφορετικά null. Επιδιορθώνουμε την παραβίαση διανύοντας το μονοπάτι προς την ρίζα, επαναλαμβάνοντας τα ακόλουθα βήματα μέχρι να συμβεί ένα βήμα εκτός από demotion ή double demotion.



(α)



(β)

Σχήμα 3.3: Μια διαγραφή σε ένα rb-tree μπορεί να παραβιάσει τον rank rule δημιουργώντας έναν κόμβο με rank ένα με δύο missing παιδιά (α) ή ένα κόμβο με rank δύο με ένα missing 3-child(β).

Επαναζυγιστικά μετά από διαγραφή βήματα στον p:

Stop: Ο κόμβος p είναι κενός, ο q δεν είναι 3-child ή ο p δεν είναι 2,2-node με rank 1. **Stop**

Στις υπόλοιπες περιπτώσεις ο κόμβος q είναι 2- ή 3-child. Έστω s είναι ο αδελφός του q, ο οποίος μπορεί να είναι ελλείπον.

Demotion: ο κόμβος s είναι 2-child. Κάνουμε demote στον p. Αυτό επιδιορθώνει τη παραβίαση του rank rule στον q αλλά μπορεί να δημιουργήσει μία νέα παραβίαση στον p. Αντικαθιστούμε τον q με τον p. Έστω p είναι ο πατέρας του q εάν υπάρχει, διαφορετικά κενός.

Στις υπόλοιπες περιπτώσεις ο κόμβος q είναι 3-child και ο s είναι non-missing 1-child. Υποθέτουμε ότι ο q είναι το δεξί παιδί του p. Η άλλη περίπτωση είναι συμμετρική. Έστω t και u είναι το δεξί και αριστερό παιδί του s, από τα οποία το καθένα ή και τα δύο μπορεί να λείπουν.

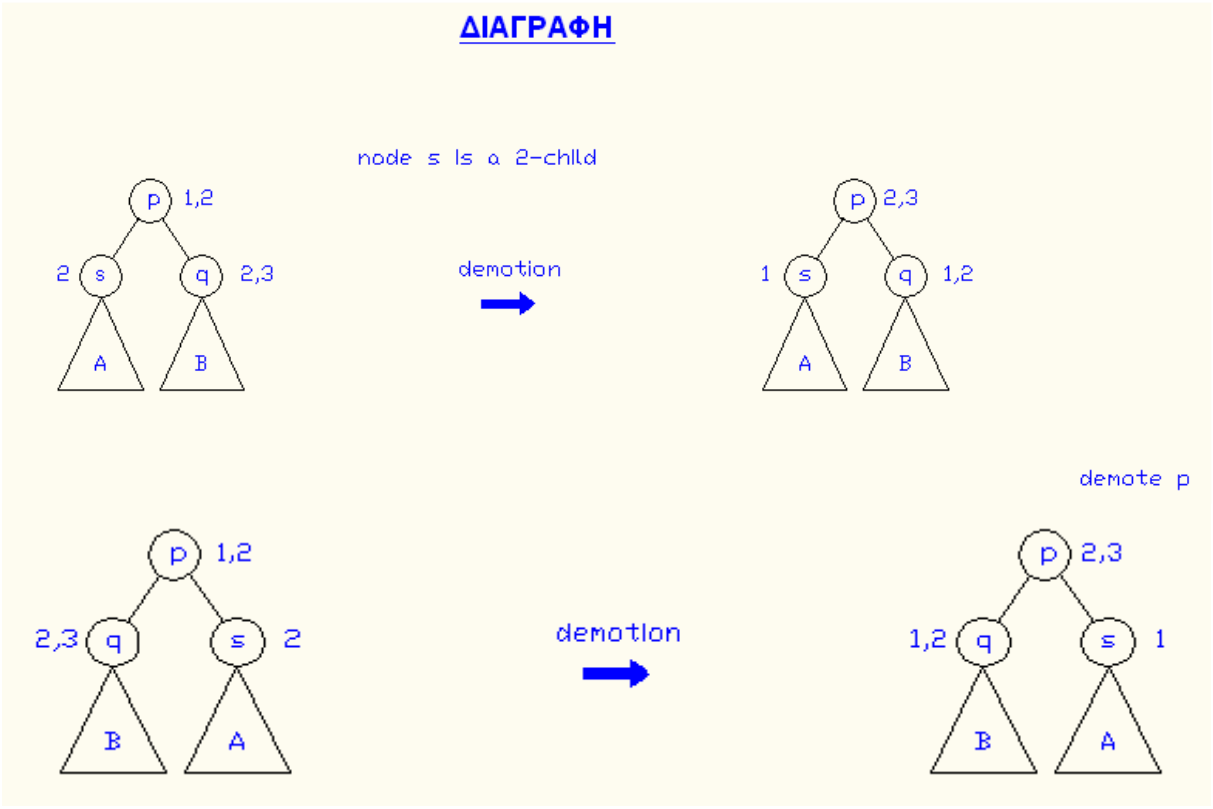
Double demotion: οι κόμβοι t και u είναι 2-children. Κάνουμε demote στον p και στον s. Αυτό επιδιορθώνει την παραβίαση στον q αλλά μπορεί να δημιουργήσει μία νέα παραβίαση στον p. Αντικαθιστούμε τον q με τον p. Έστω p είναι ο πατέρας του q εάν υπάρχει, διαφορετικά κενός.

Rotation: ο κόμβος u είναι 1-child. Κάνουμε rotate στον s, promote στον s και demote στον p. Εάν ο t λείπει κάνουμε demote στον p ξανά.(Σε αυτή την περίπτωση ο q λείπει επίσης και ο p είναι τώρα leaf του οποίου το rank πρέπει να είναι μηδέν). Αυτό επιδιορθώνει την παραβίαση χωρίς να δημιουργεί νέα. **Stop**

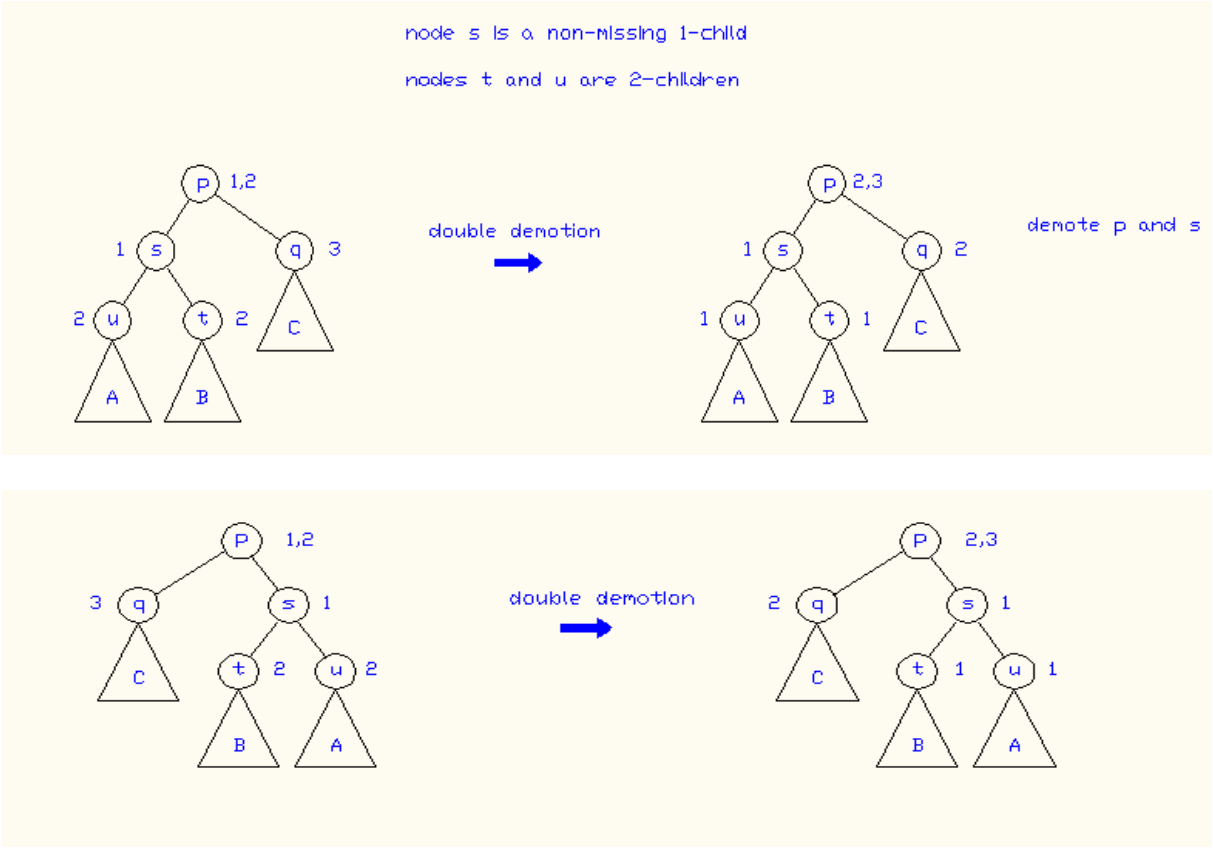
Double rotation: ο κόμβος t είναι 1-child και ο u είναι 2-child. Κάνουμε rotate στον t δύο φορές κάνοντας τον s αριστερό του παιδί και τον p δεξί του παιδί. Κάνουμε promote τον t δύο φορές, demote τον s και demote τον p δύο φορές. Αυτό επιδιορθώνει την παραβίαση χωρίς να δημιουργεί νέα.**Stop**

Κατά τη διάρκεια της επαναζύγισης μετά από διαγραφή, υπάρχει το πολύ μία παραβίαση του rank rule. Ο κόμβος p είναι 2,2-node με rank ένα ή ο κόμβος q είναι 3-child. Μετά το πρώτο βήμα ο q πρέπει να είναι 3-child. Η επαναζύγιση γίνεται από το μονοπάτι του κόμβο που αντικαθιστά τον διαγραφέντα κόμβο προς την ρίζα κάνοντας μηδέν ή περισσότερα βήματα demotion ή double demotion ακολουθούμενα από stop, rotation ή double rotation.

Η διαγραφή στα rb-trees είναι λίγο πιο πολύπλοκη από την εισαγωγή με δύο μη τερματικές περιπτώσεις αντί για μία. Η διαγραφή κοστίζει το πολύ δύο περιστροφές, όπως και η εισαγωγή.



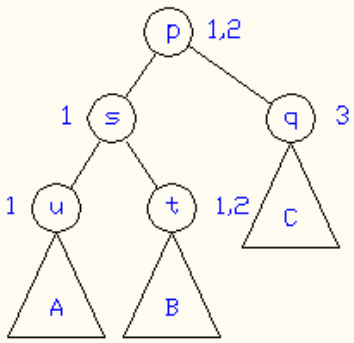
(α)



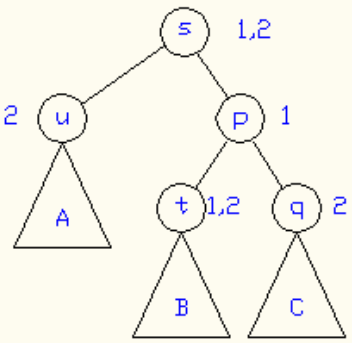
(β)

Σχήμα 3.4: Επαναζύγιση μετά από μία διαγραφή. Οι αριθμοί είναι τα rank differences. Οι περιπτώσεις (α) και (β)είναι μη τερματικές. Εάν ο q είναι 2-child, η πρώτη (α) περίπτωση χρησιμοποιείται εάν ο p είναι φύλλο.

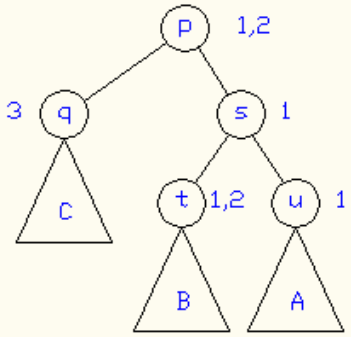
node s is a non-missing 1-child
node u is a 1-child



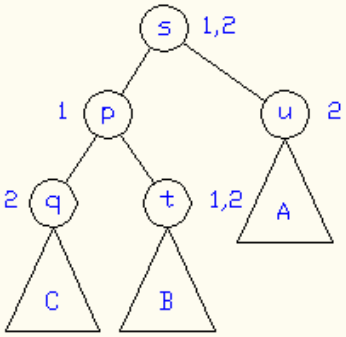
rotation



rotate at s
promote s
demote p (again)

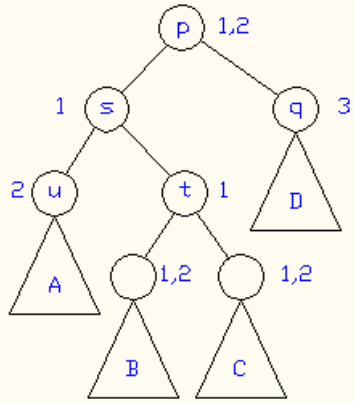


rotation

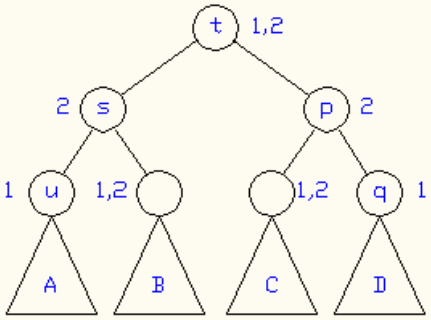


(γ)

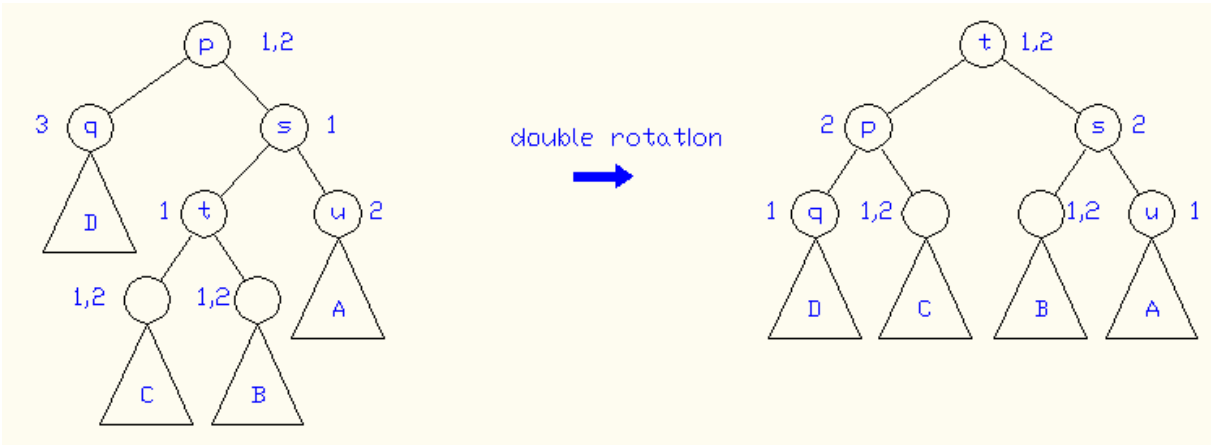
node s is a non-missing 1-child
node t is a 1-child
node u is a 2-child



double rotation



rotate at t twice
promote t twice
demote s
demote p twice

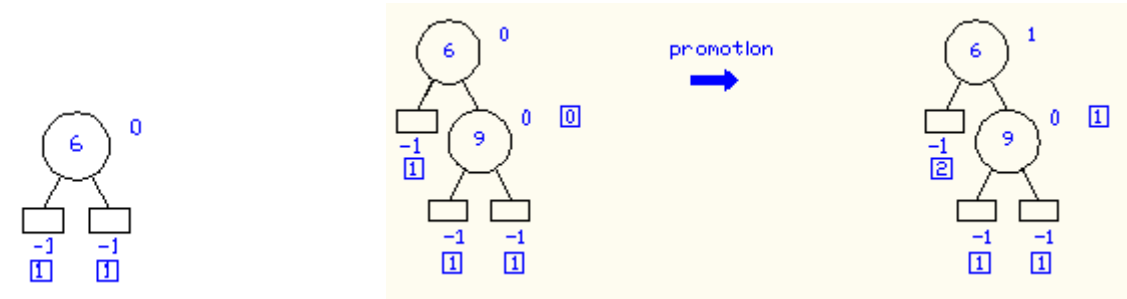


(δ)

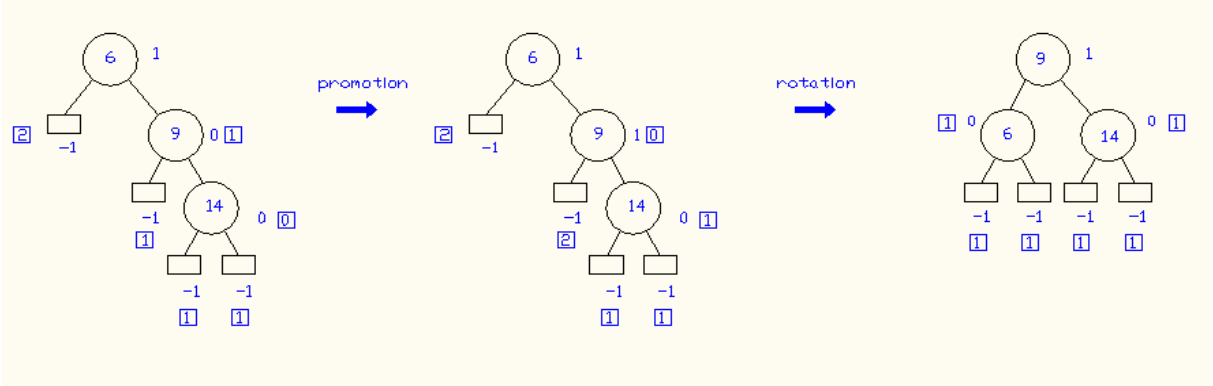
Σχήμα 3.5:(Συνέχεια) επαναζύγιση μετά από διαγραφή. Οι αριθμοί είναι τα rank differences. Στην περίπτωση (γ) υποθέτουμε ότι ο t δεν είναι ελλείπον. Εάν είναι, ο p είναι φύλλο και τον κάνουμε demote.

3.3. Παραδείγματα εισαγωγής και διαγραφής στοιχείων στα rb-trees

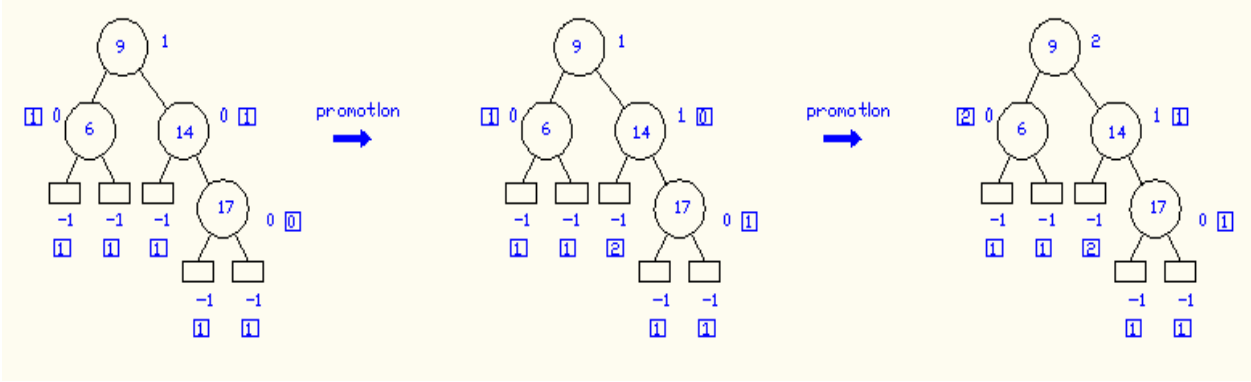
Τα σχήματα 3.6, 3.7, 3.8, 3.9 αποτελούν παράδειγμα 12 διαδοχικών ενθέσεων σε ένα αρχικά άδειο rank-balance tree. Οι αριθμοί που είναι μέσα στα «κουτάκια» δείχνουν το rank difference του κόμβου ενώ οι αριθμοί που είναι εκτός δείχνουν το rank του.



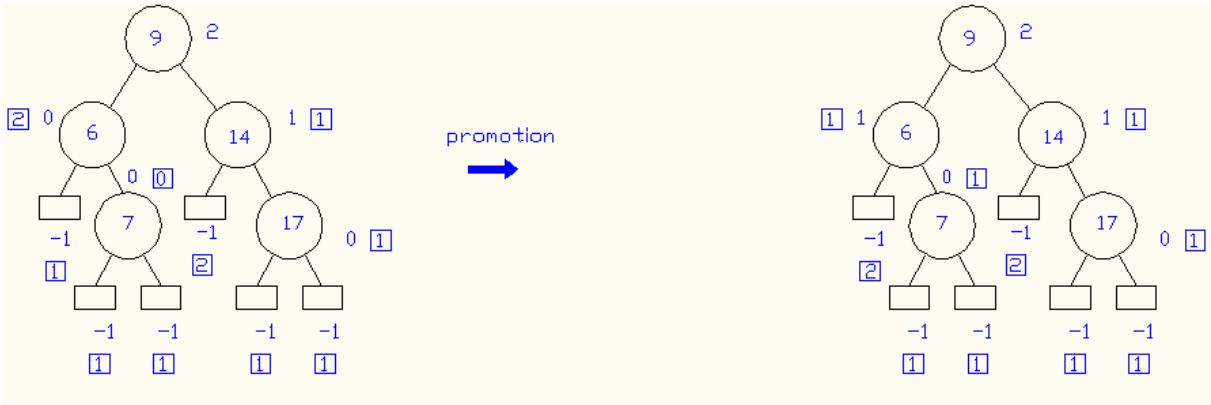
(α) (β)



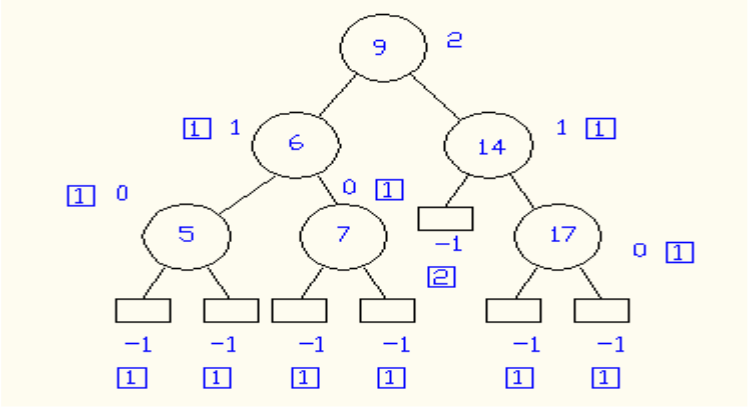
(γ)



(δ)

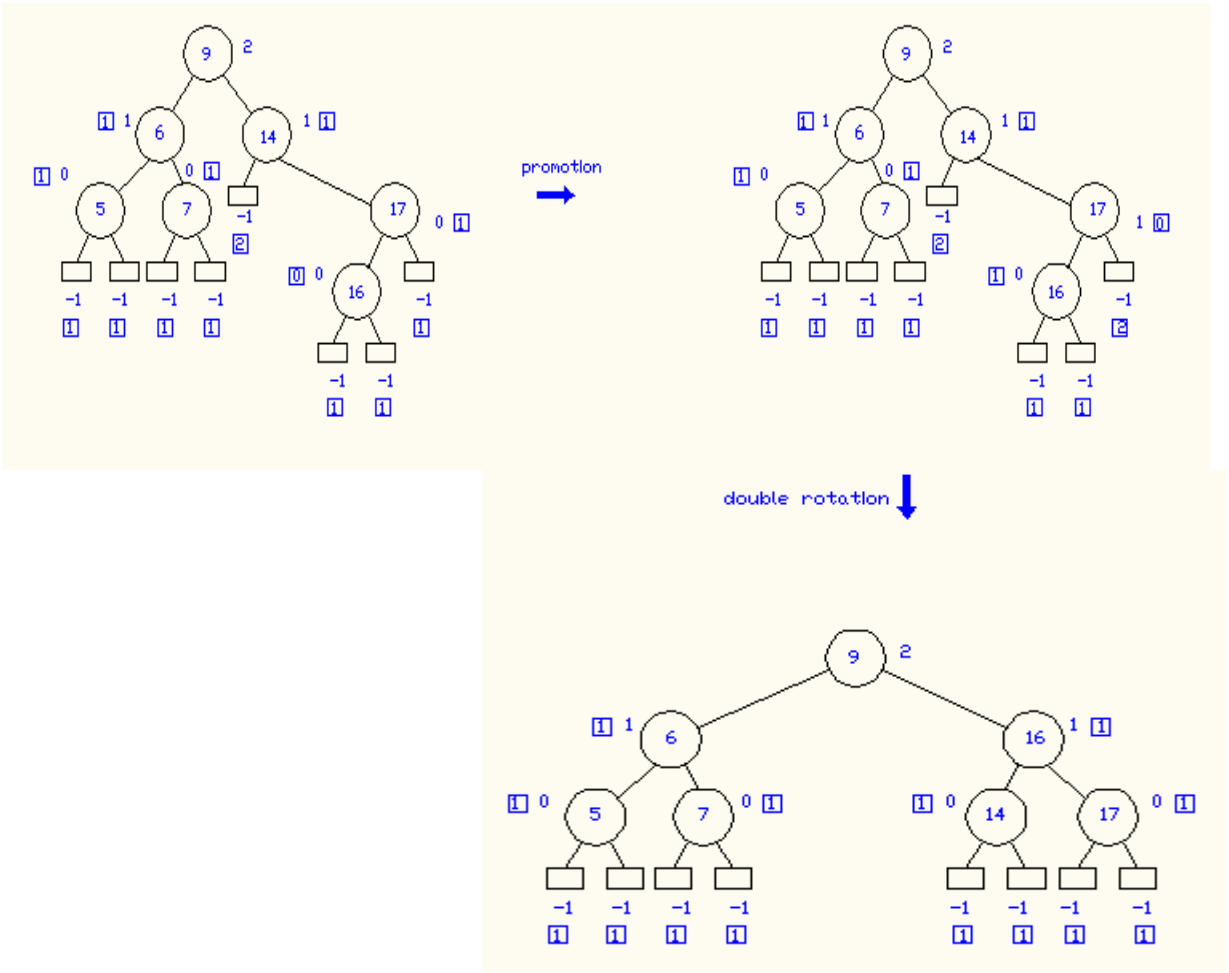


(ε)



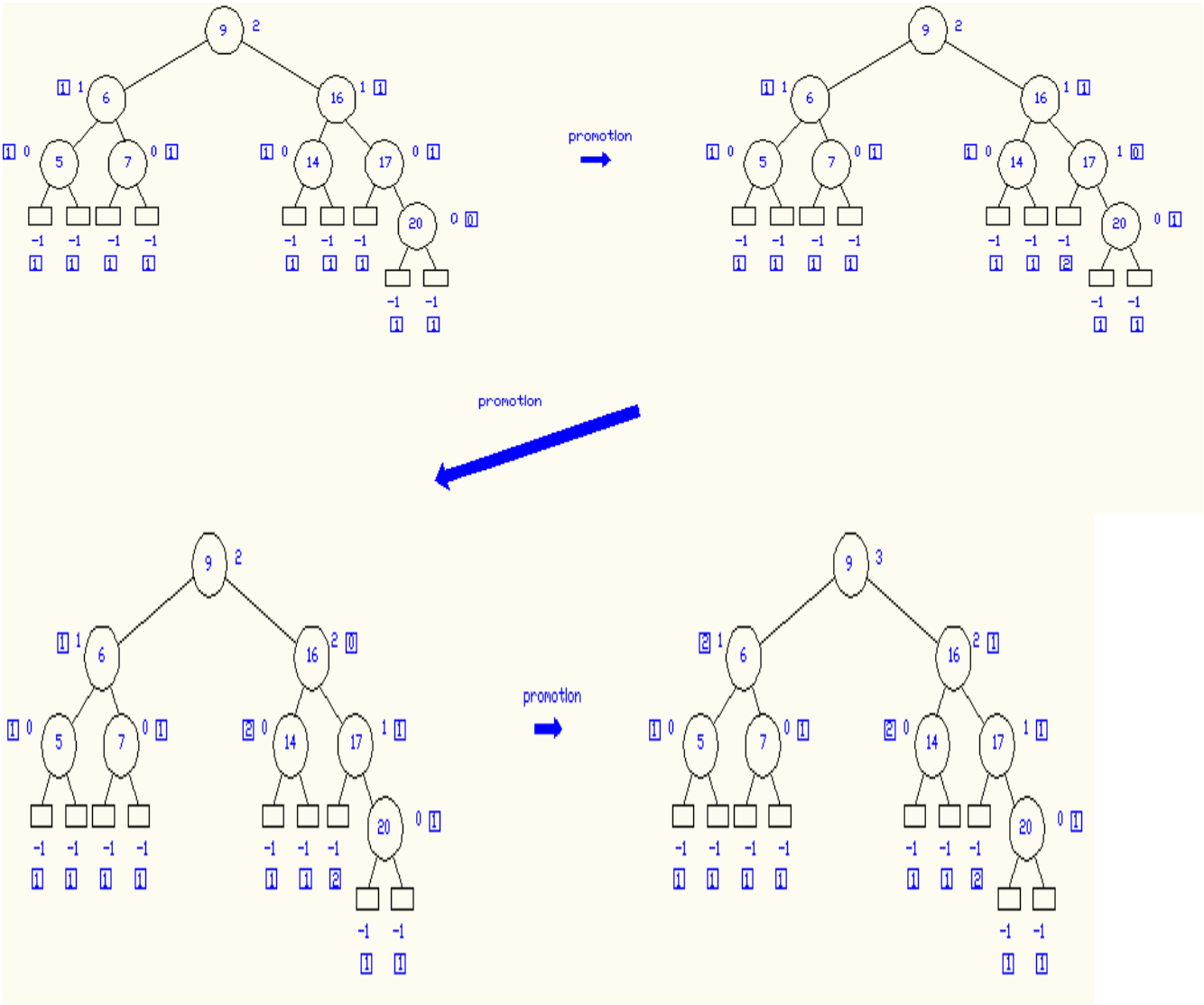
(στ)

Σχήμα 3.3: (α)-(στ) Διαδοχικές ενθέσεις των 6, 9, 14, 7, 5.



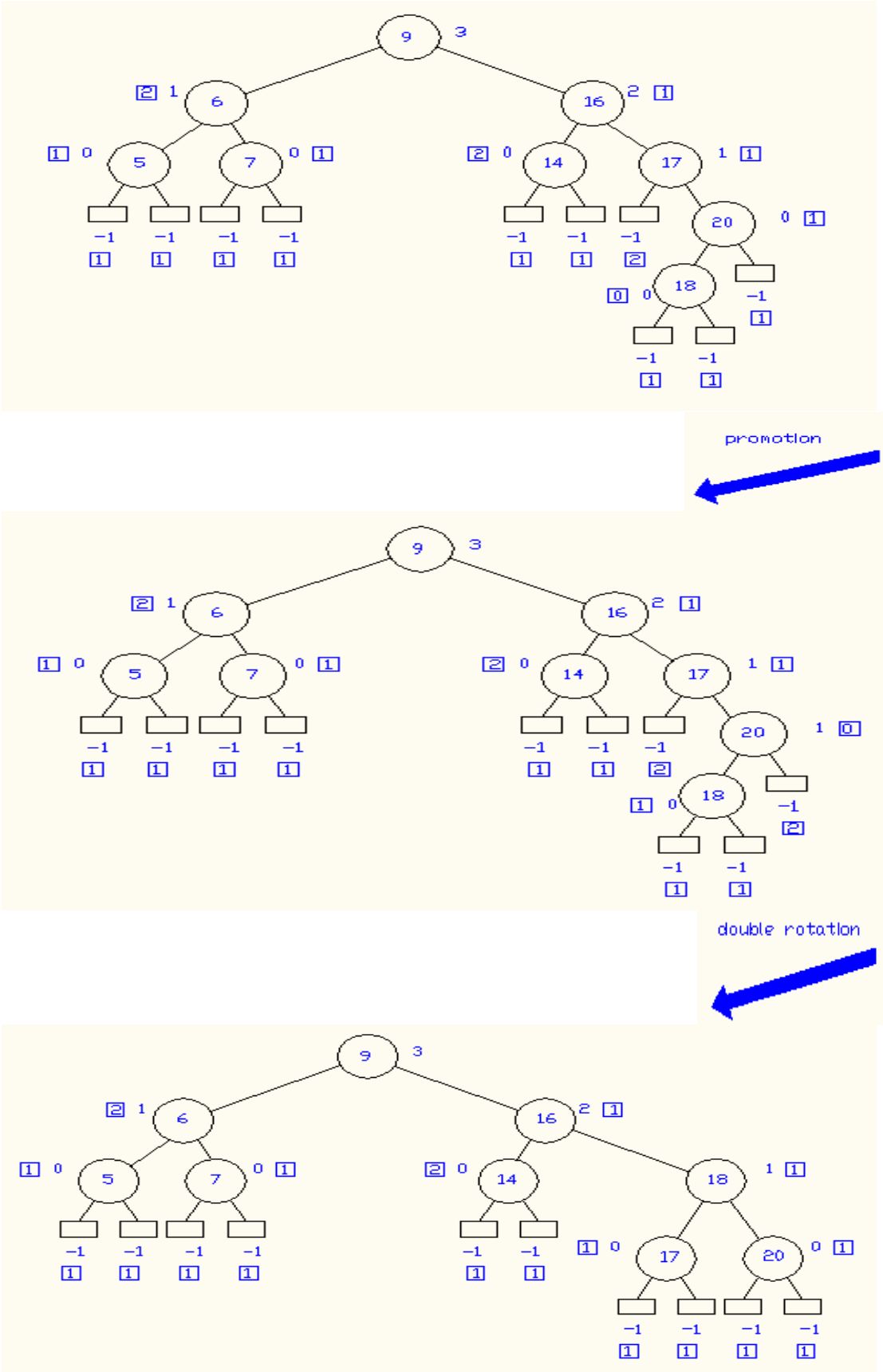
(ζ)

Σχήμα 3.4: (Συνέχεια) (ζ) ένθεση του 16.



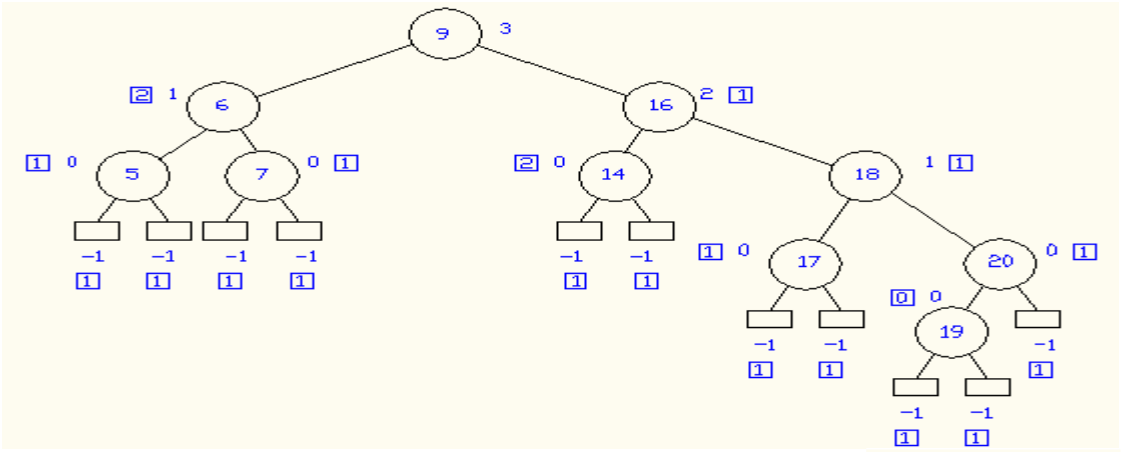
(η)

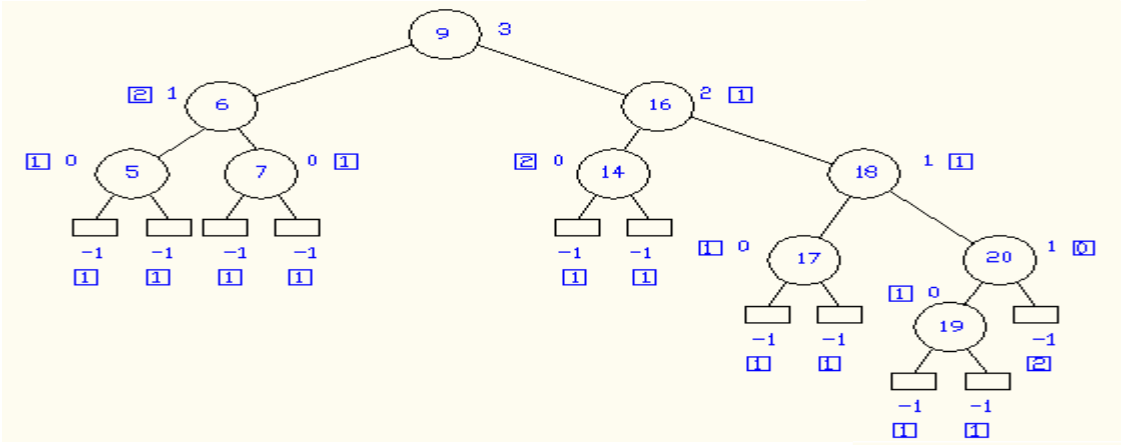
Σχήμα 3.5: (Συνέχεια) (η) ένθεση του 20.

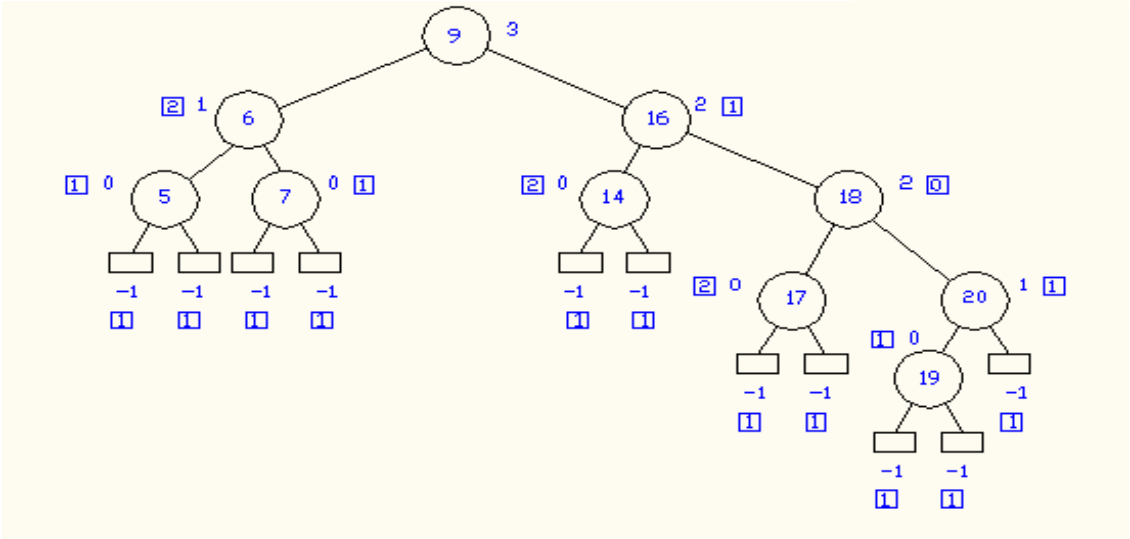


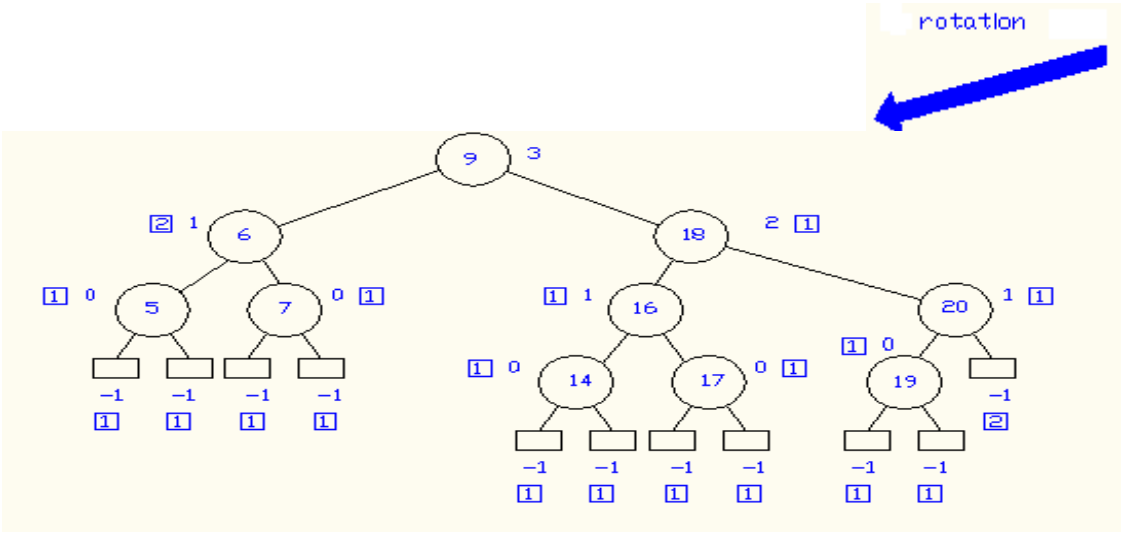
(θ)

Σχήμα 3.6: (Συνέχεια) (θ) ένθεση του 18.


9
3
6
16
5
7
14
18
17
20
19
promotion

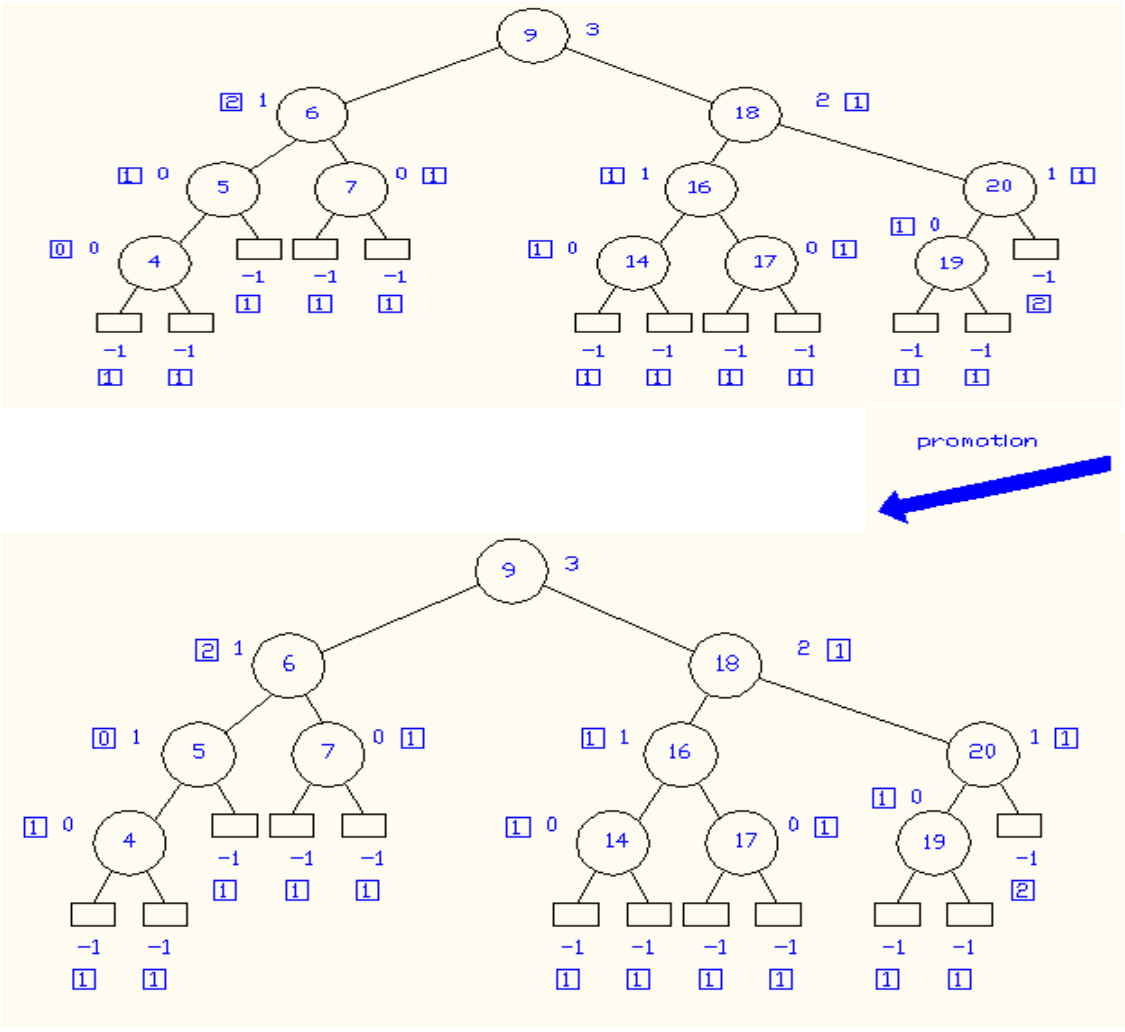

9
3
6
16
5
7
14
18
17
20
19
promotion

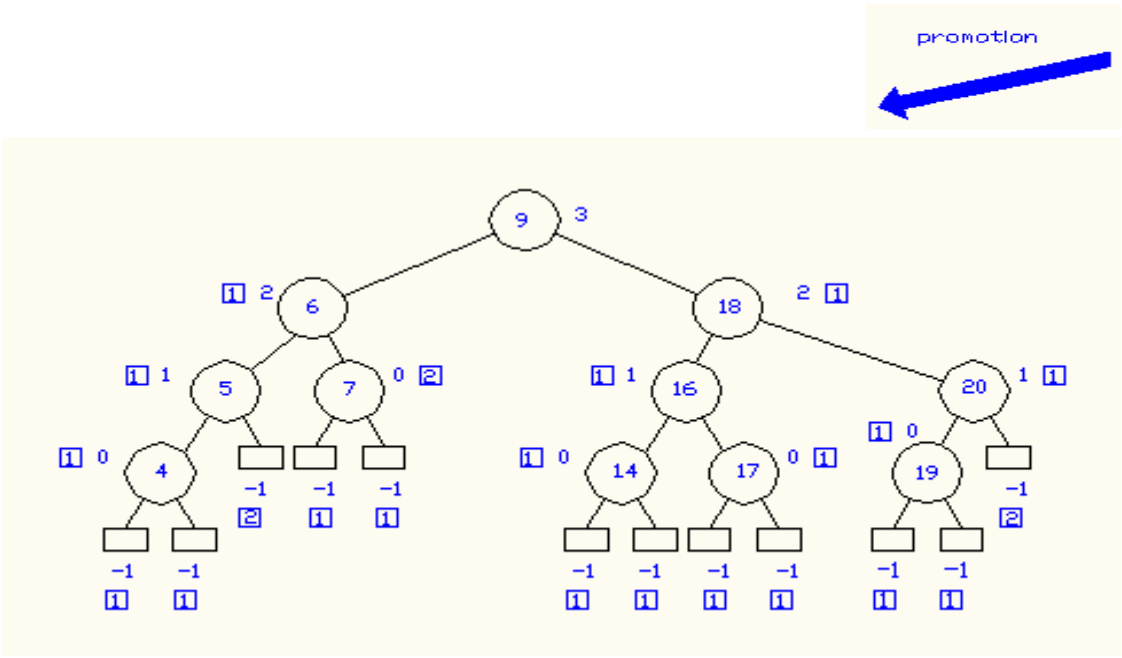

9
3
6
16
5
7
14
18
17
20
19



(ι)

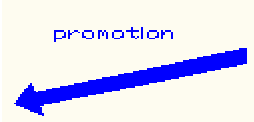
Σχήμα 3.7: (Συνέχεια) (ι) ένθεση του 19.

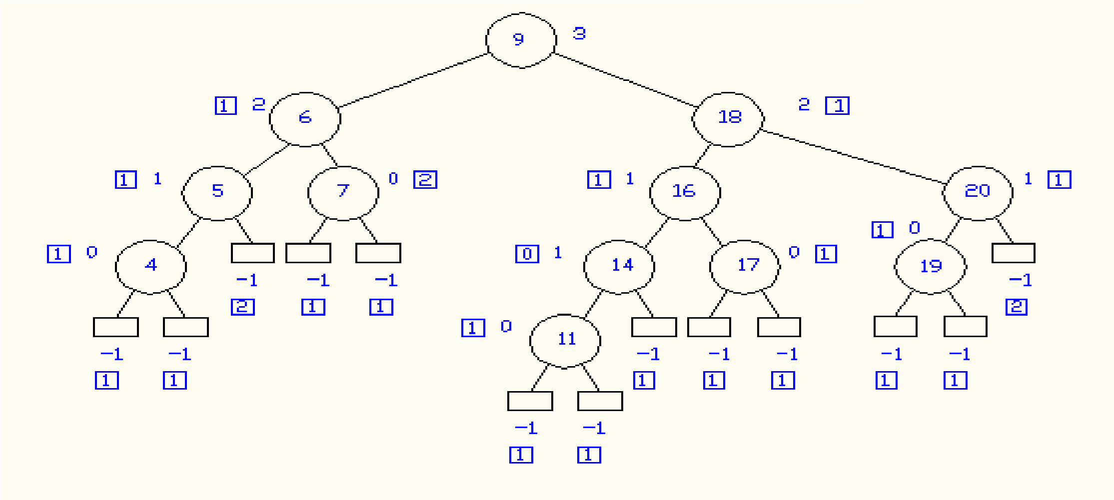


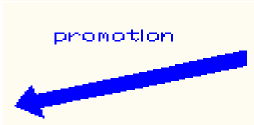


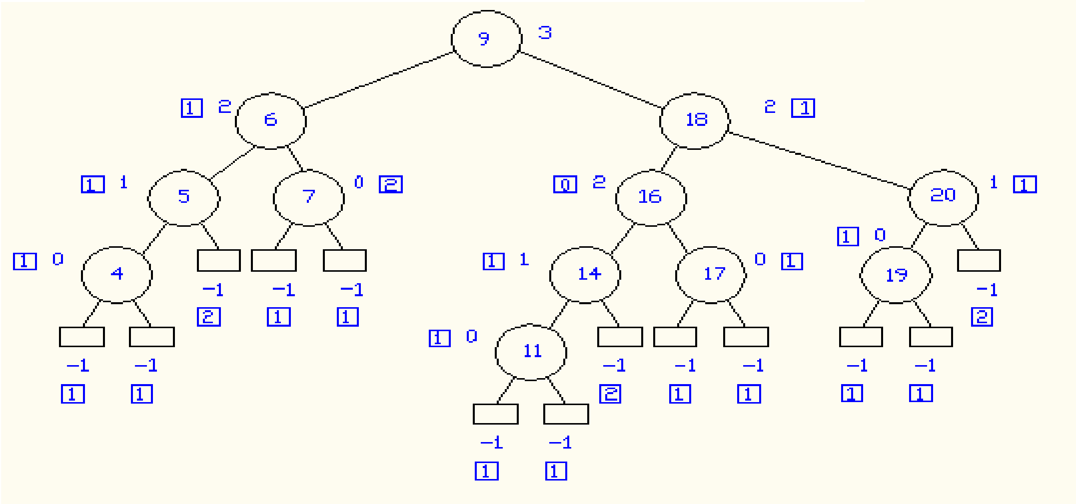
(ια)

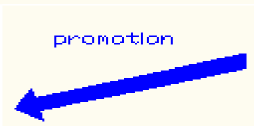
Σχήμα 3.8: (Συνέχεια) (ια) ένθεση του 4.

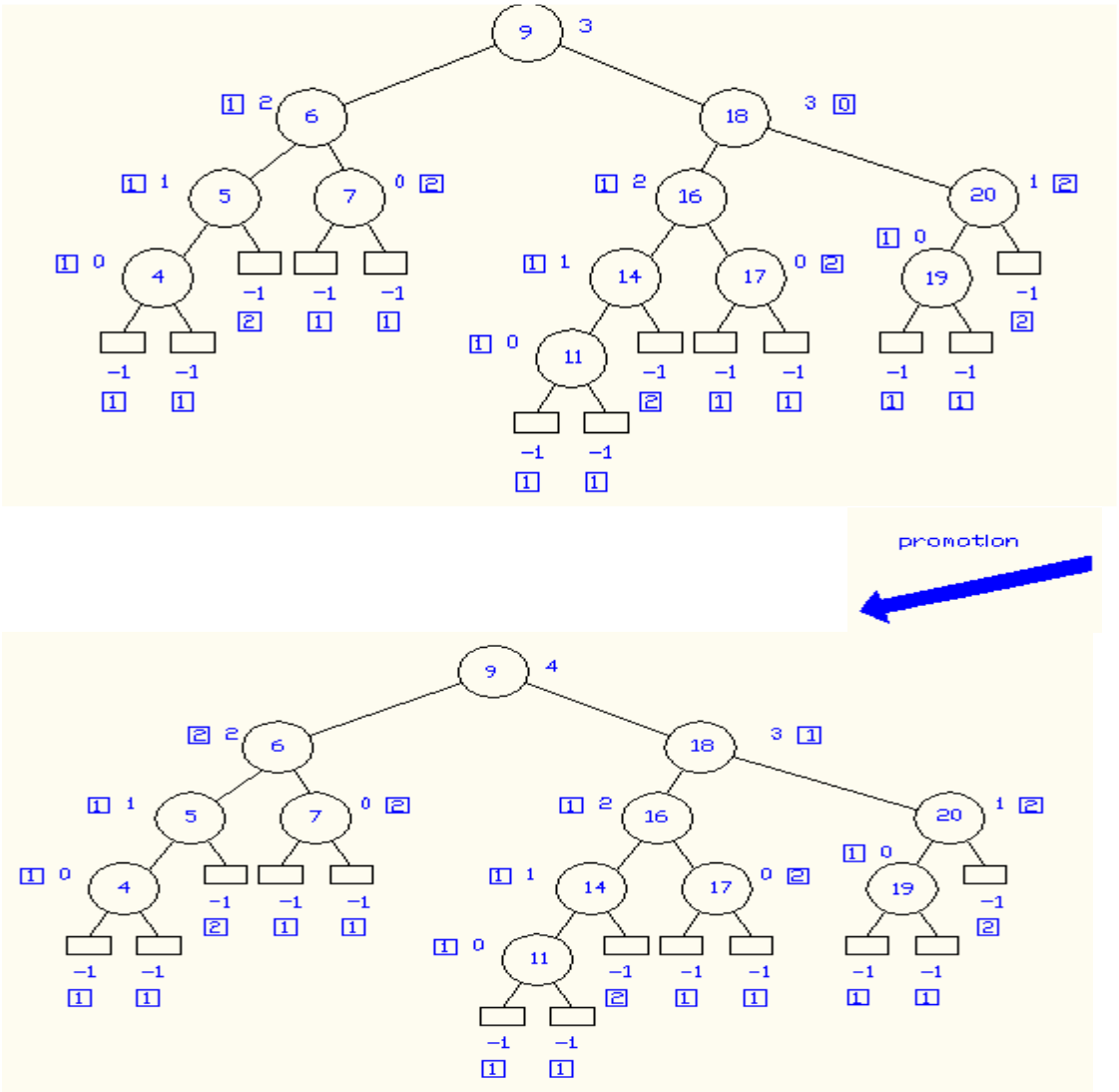

promotion


9 3
1 2 6
18 2 1
1 1 5
7 0 2
1 1 16
20 1 1
1 0 4
0 1 14
17 0 1
1 0 19
1 0 11
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
2 1 1 2
1 1 1 1 1 1 1 1 1 1


promotion


9 3
1 2 6
18 2 1
1 1 5
7 0 2
0 2 16
20 1 1
1 0 4
1 1 14
17 0 1
1 0 19
1 0 11
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
2 1 1 2
1 1 2 1 1 1 1 1 1 1

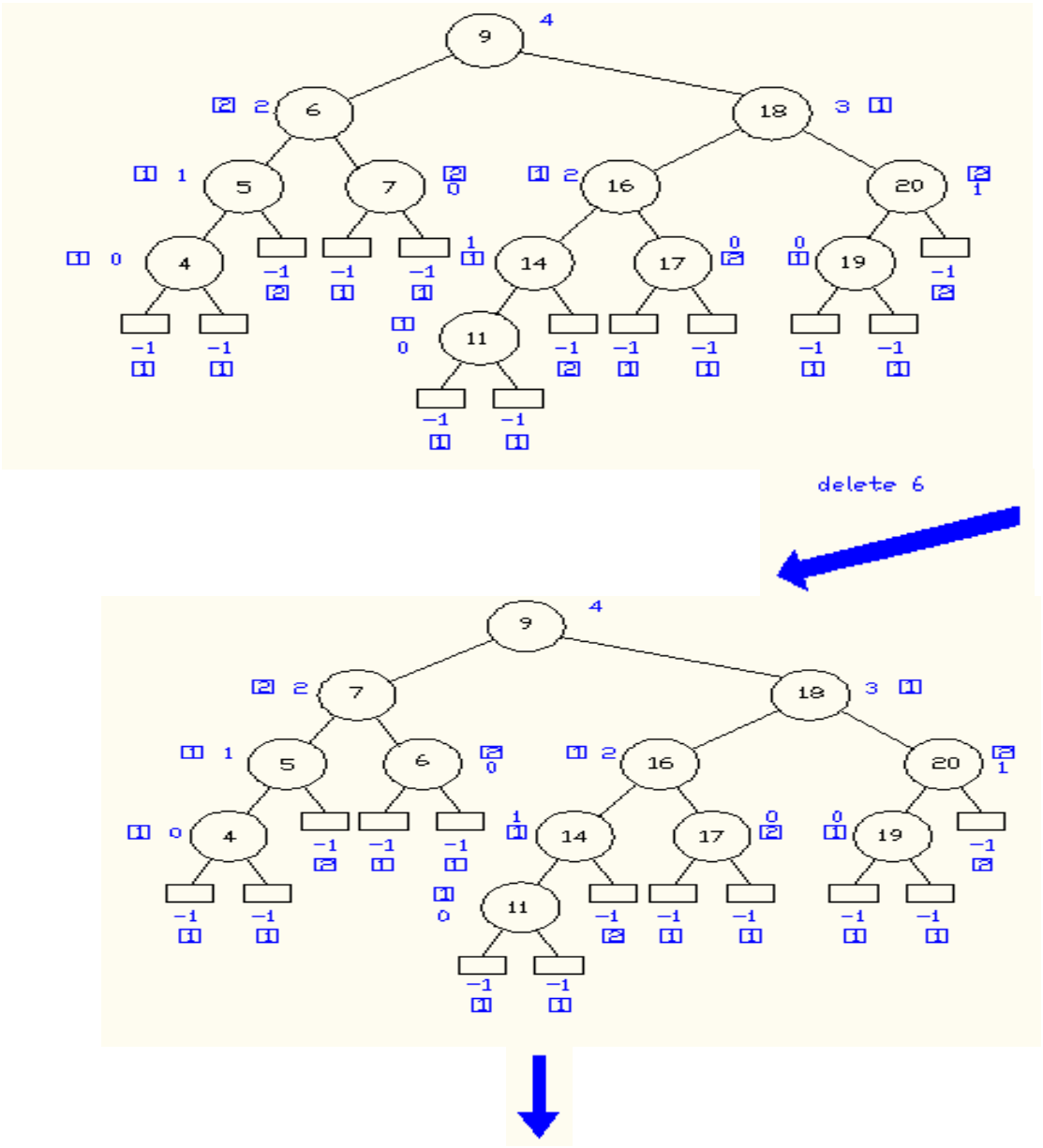

promotion

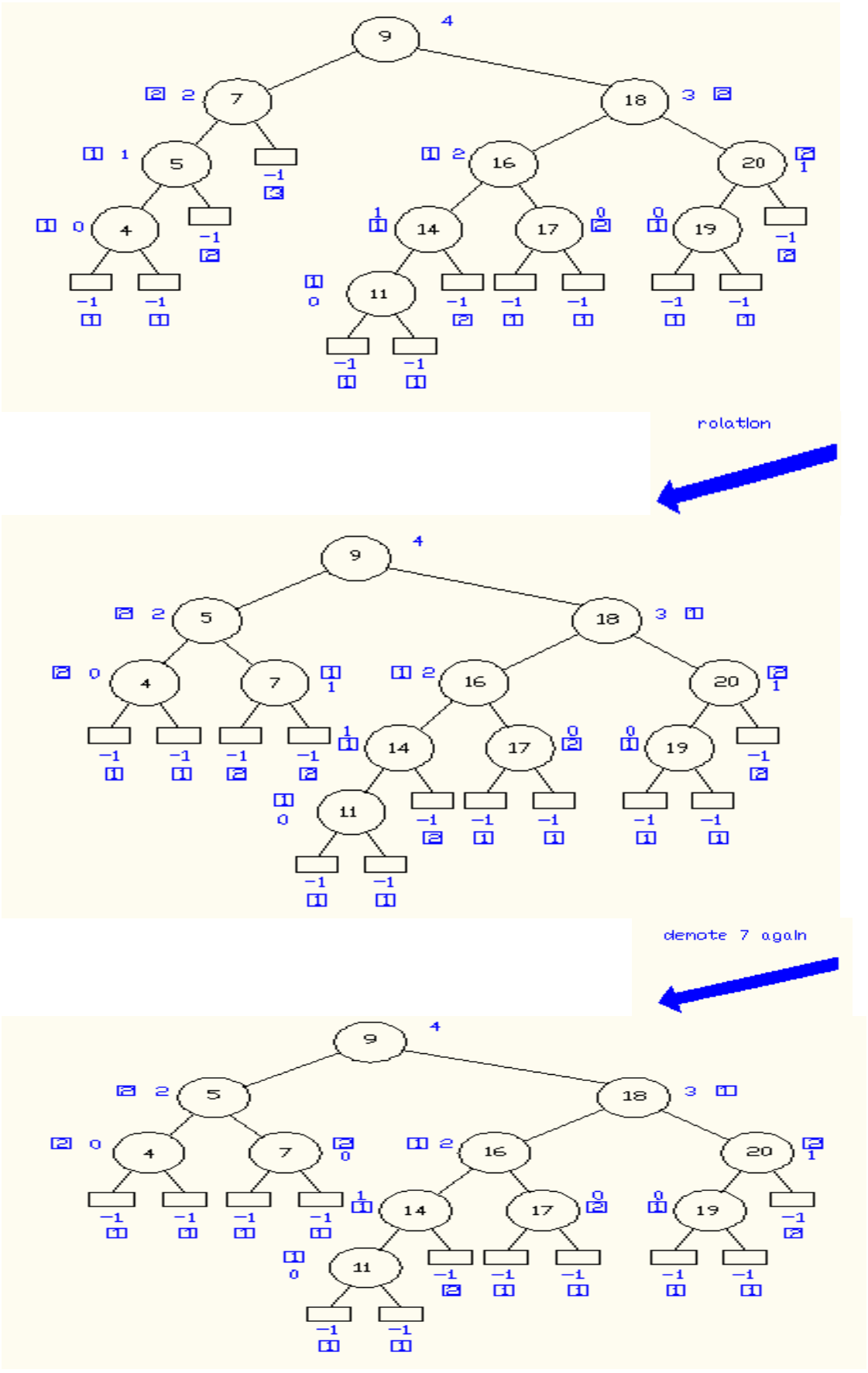


(ιβ)

Σχήμα 3.9: (Συνέχεια) (ιβ) ένθεση του 11.

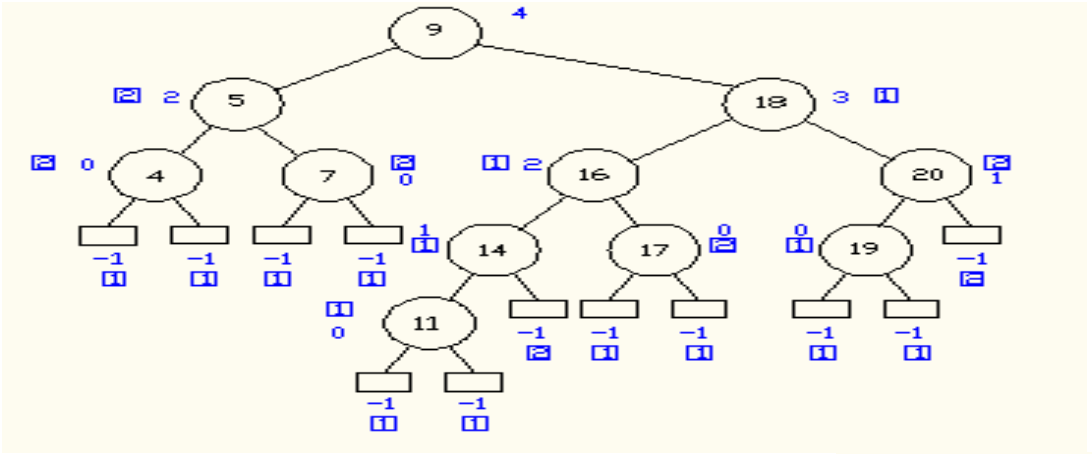
Τα σχήματα 3.10, 3.11, 3.12, 3.13, αποτελούν παράδειγμα 7 διαδοχικών αποσβέσεων.



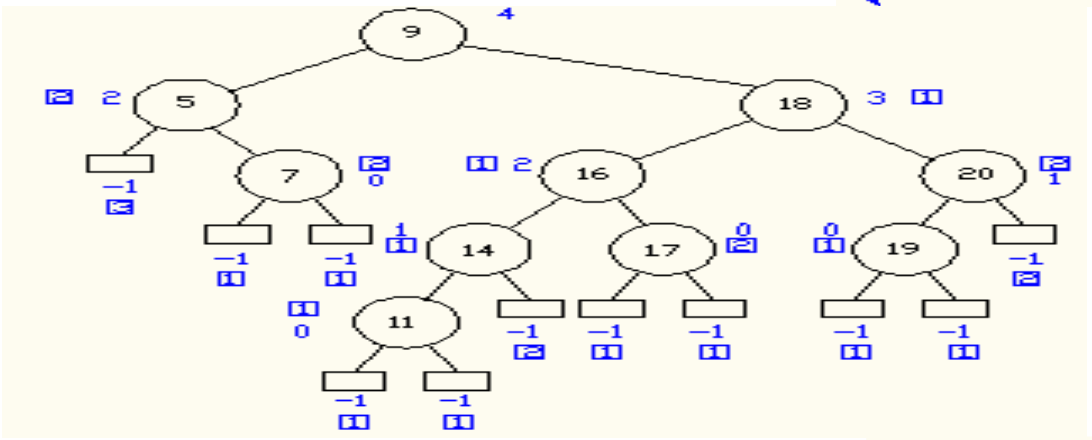


(α)

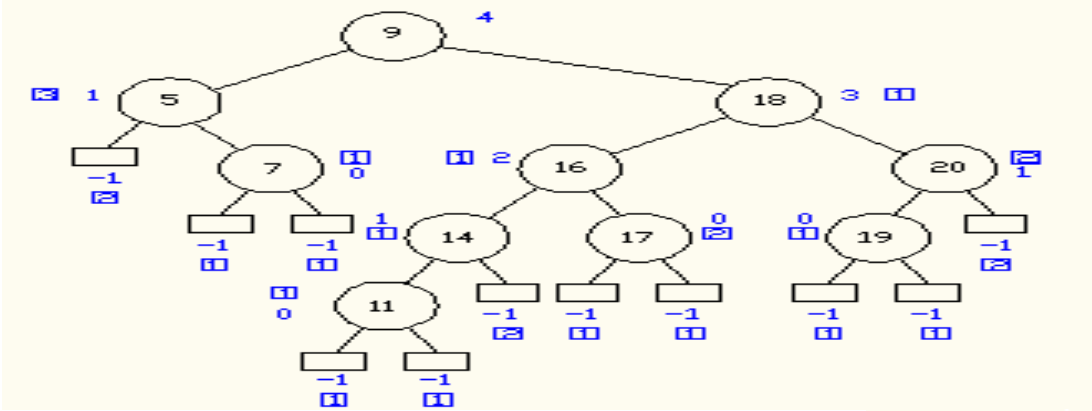
Σχήμα 3.10: (α) Απόσβεση του 6



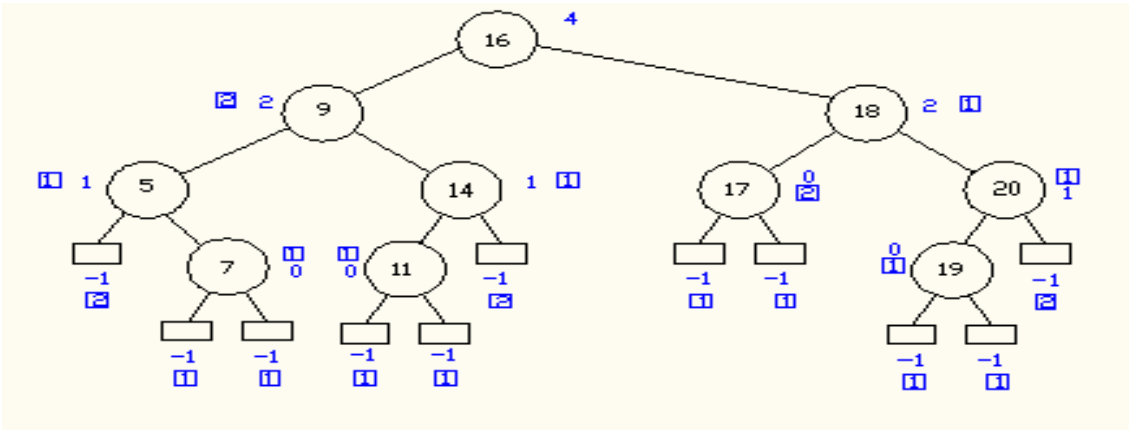
delete 4



delete 5

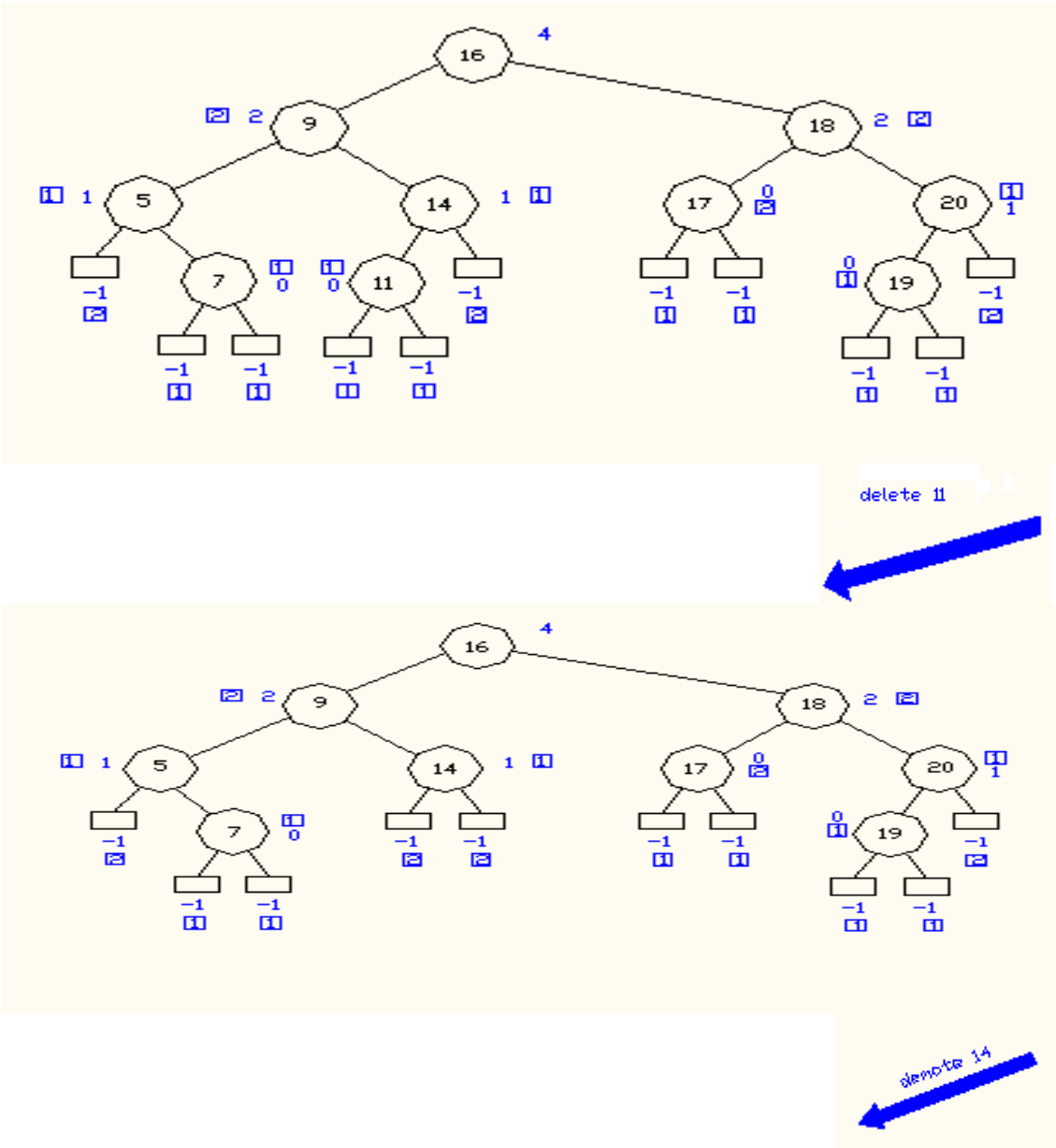


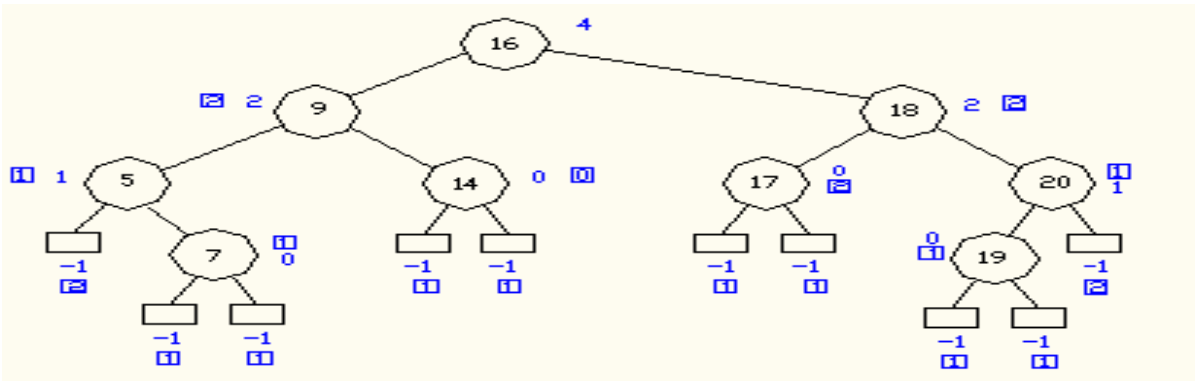
double rotate



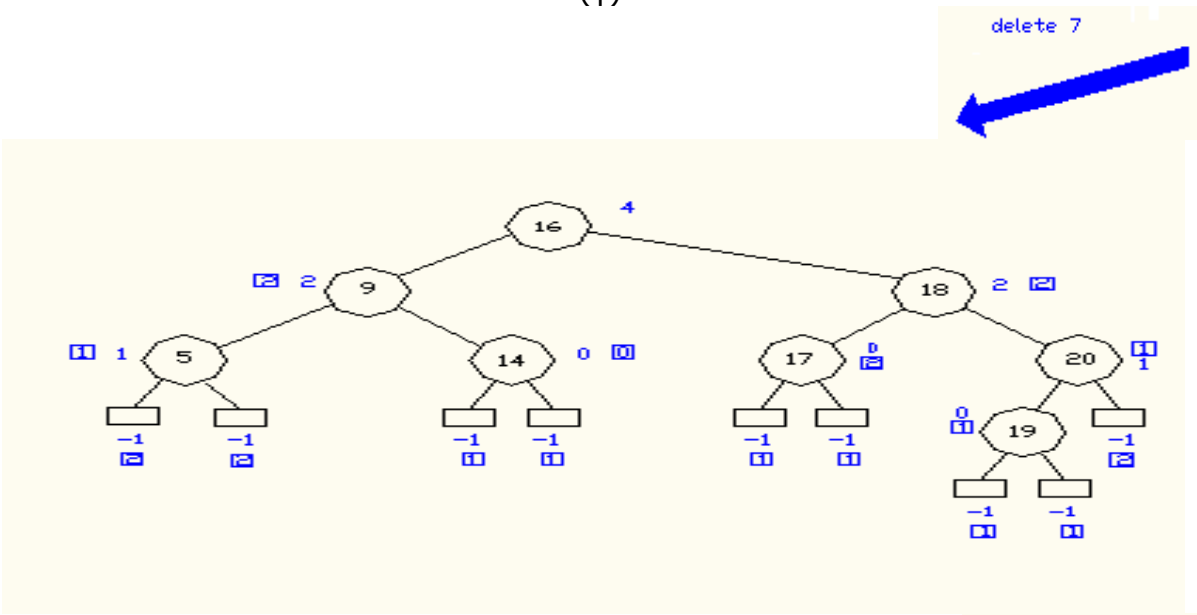
(β)

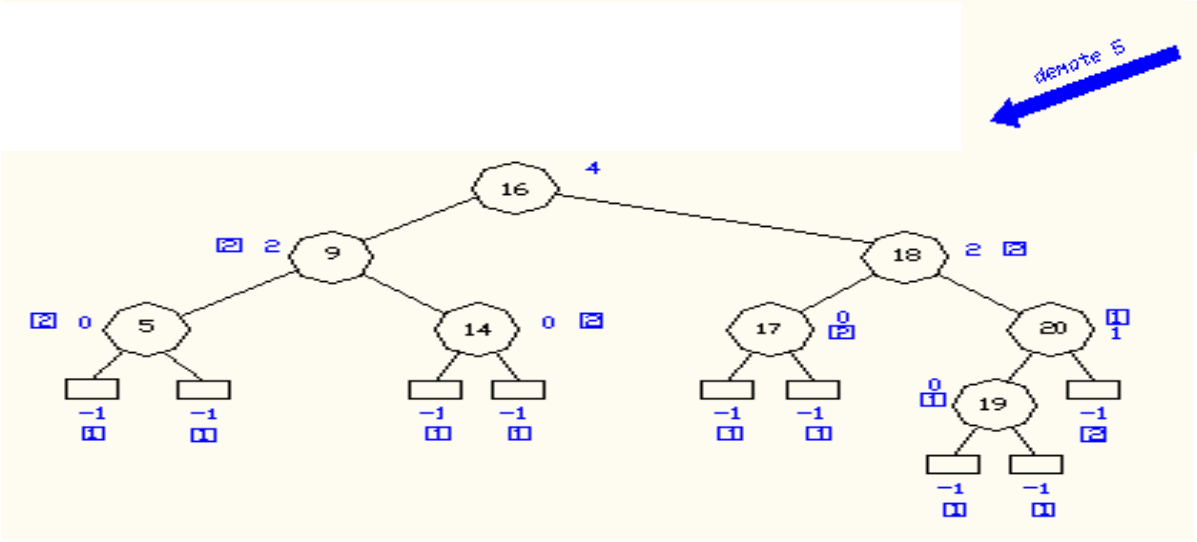
Σχήμα 3.11: (Συνέχεια) (β) απόσβεση του 4




16
9
18
5
14
17
20
7
19

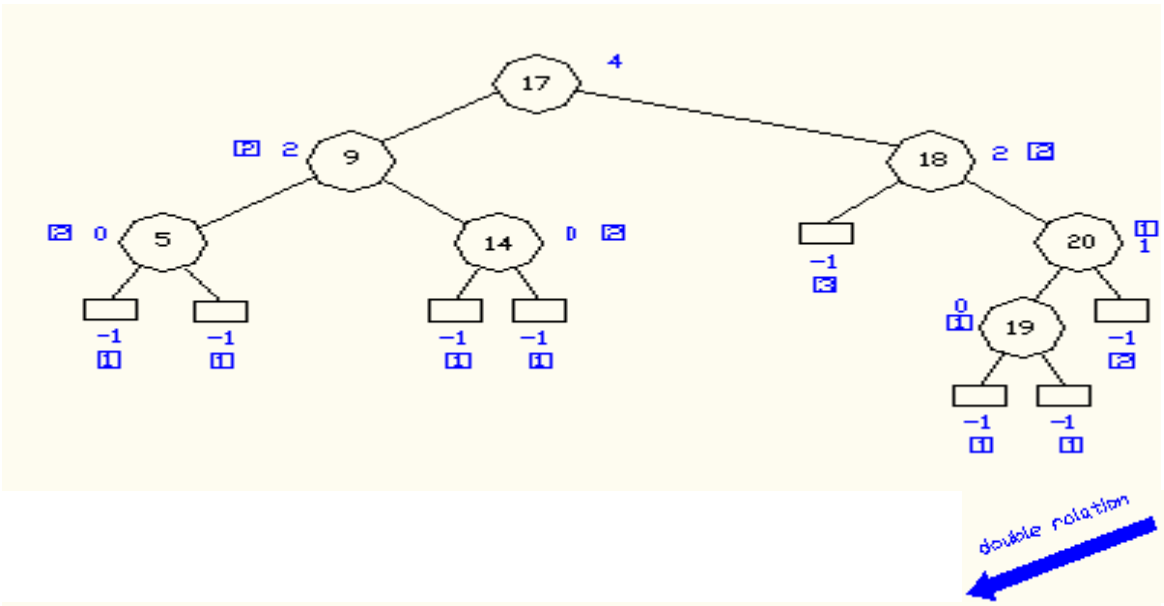
(γ)

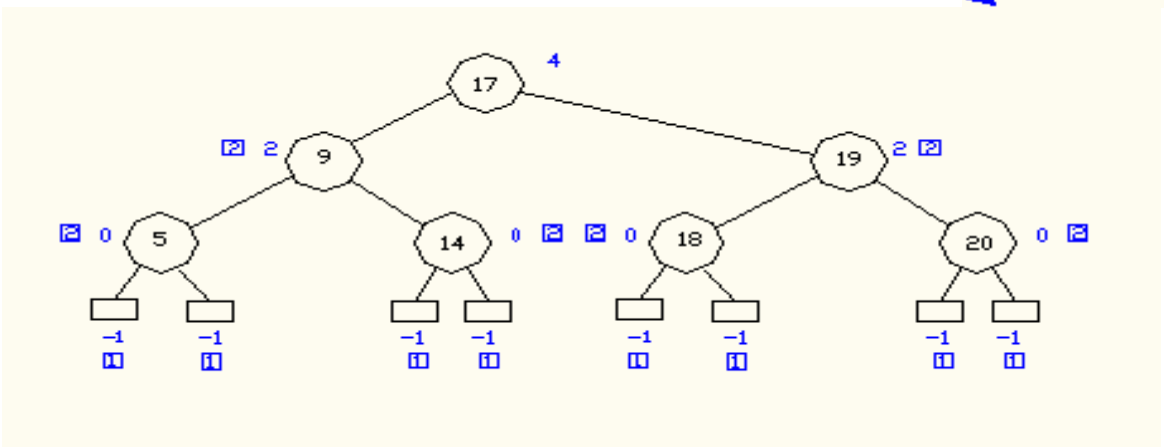

delete 7
16
9
18
5
14
17
20
19


demote 5
16
9
18
5
14
17
20
19

(δ)

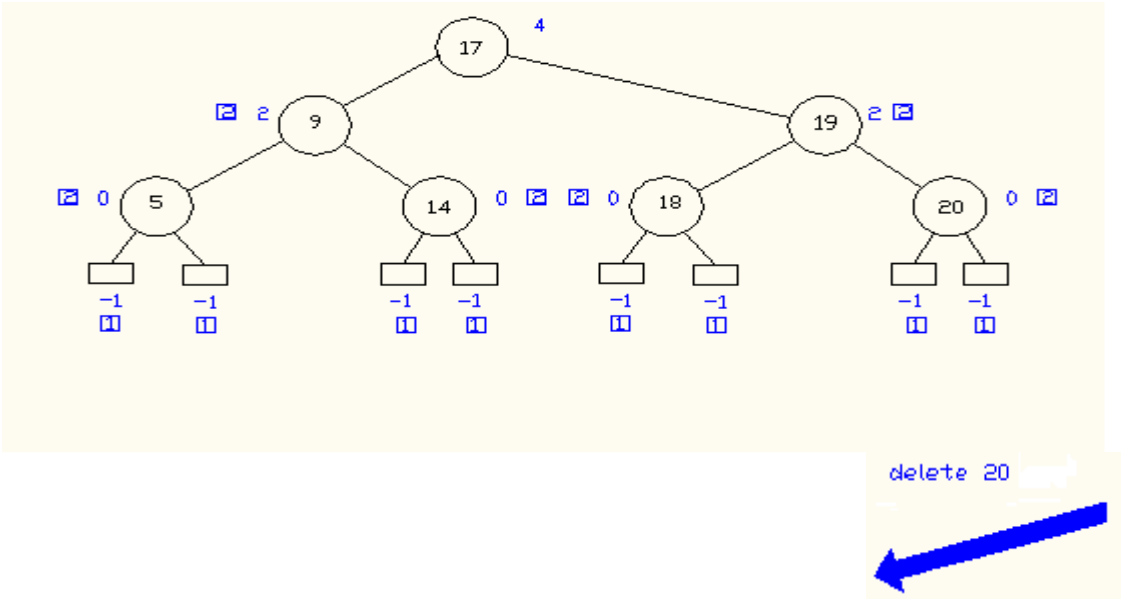

delete 16

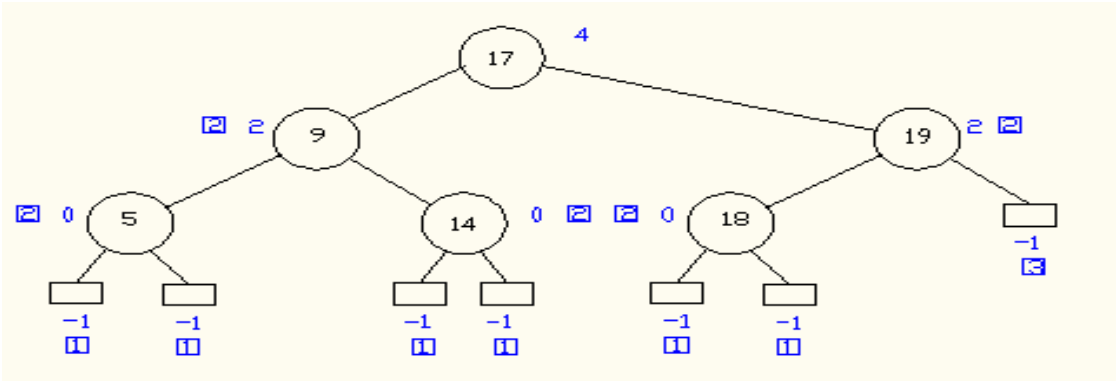


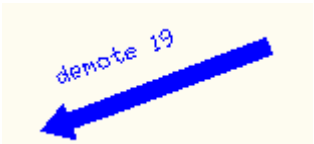


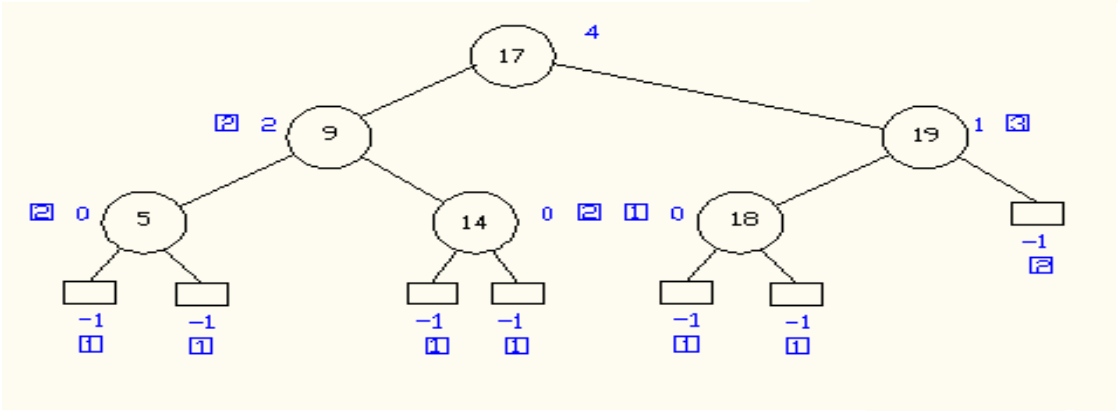
(ε)

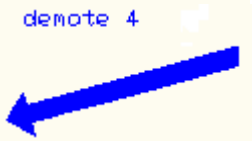
Σχήμα 3.12: (Συνέχεια) (γ)-(ε) διαδοχικές αποσβέσεις των 11, 7, 16

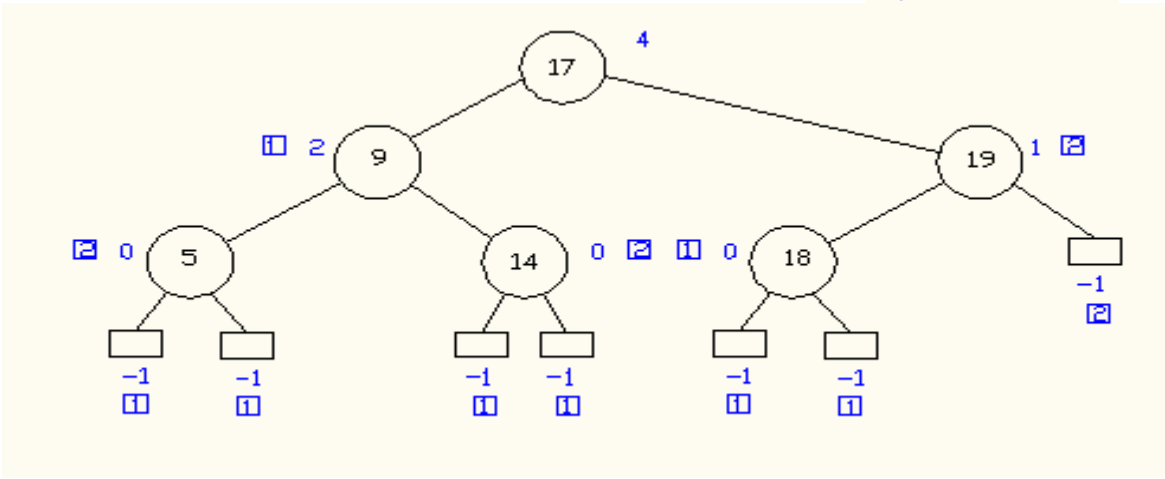



17
4
9
2
19
2
5
0
14
0
18
0
-1
-1
-1
-1
-1
-1
-1


demote 19

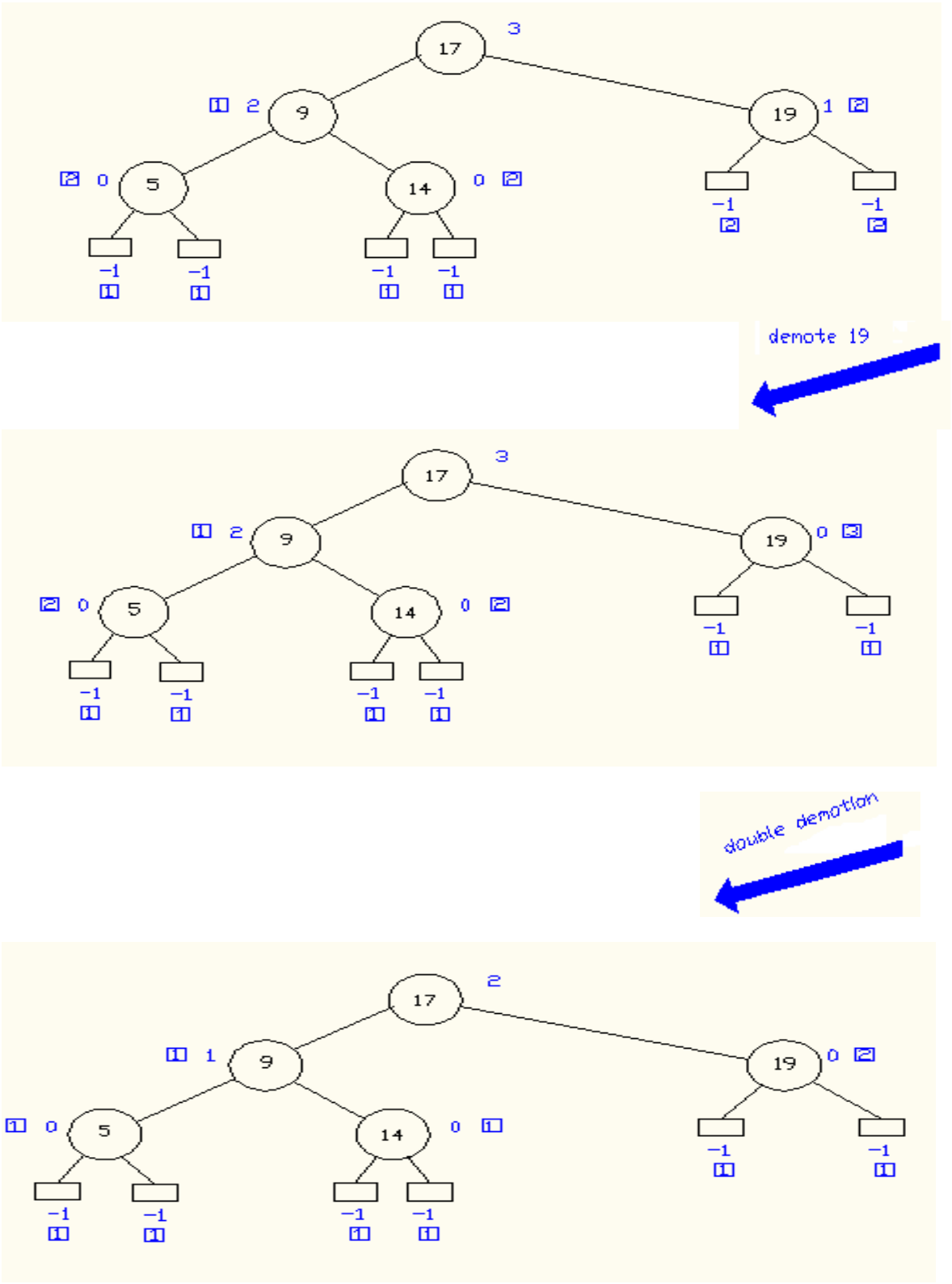

17
4
9
2
19
1
5
0
14
0
18
0
-1
-1
-1
-1
-1
-1
-1


demote 4


17
4
9
2
19
1
5
0
14
0
18
0
-1
-1
-1
-1
-1
-1
-1

(στ)


delete 18



(ζ)

Σχήμα 3.13: (Συνέχεια) (στ)-(ζ) διαδοχικές αποσβέσεις των 20, 18

3.4. Πειραματική αξιολόγηση

Στον κώδικα του παραρτήματος υλοποιούμε τα rank-balanced trees. Παράλληλα εισάγουμε ορισμένες μεταβλητές και κάνουμε κάποιες μετρήσεις για να αποδείξουμε την ορθότητα της θεωρητικής ανάλυσης. Οι μεταβλητές αυτές είναι οι εξής:

```
//κόστος αναζήτησης
int access_cost=0;

//συνολικό κόστος εισαγωγής
int Total_insertion_cost=0;

//συνολικό κόστος διαγραφής
int Total_deletion_cost=0;

//επαναζυγιστικό κόστος
int rebalancing_cost=0;
```

Η μεταβλητή ***access_cost*** αυξάνεται κατά ένα σε κάθε βήμα της αναζήτησης. Το γεγονός αυτό είναι φανερό στο παρακάτω κομμάτι κώδικα.

……..

```
// αναζήτηση ενός στοιχείου με βάση το κλειδί του, αν δεν βρεθεί το
// στοιχείο επιστρέφεται το τελευταίο στοιχείο κατά μήκος του μονοπατιού
// αναζήτησης

Tree * access(int key) {
        assert(root != NULL);
        Tree * x = root, * prev_x=NULL;
        while ((x != NULL) && (x->key != key)) {
                prev_x = x;
                if (compare_key(key, x) < 0) {
                        x = x->left;
                }
                else {
                        x = x->right;
                }
                // αύξηση του access_cost κατά ένα
                access_cost++;
        }
```

……..

Η μεταβλητή ***rebalancing_cost*** αυξάνεται κατά ένα έπειτα από την εκτέλεση οποιασδήποτε επαναζυγιστικής πράξης. Αυτό συμβαίνει τόσο στη διαγραφή όσο και στην εισαγωγή. Ορισμένα κομμάτια του κώδικα που συμβαίνει αυτό είναι τα εξής:

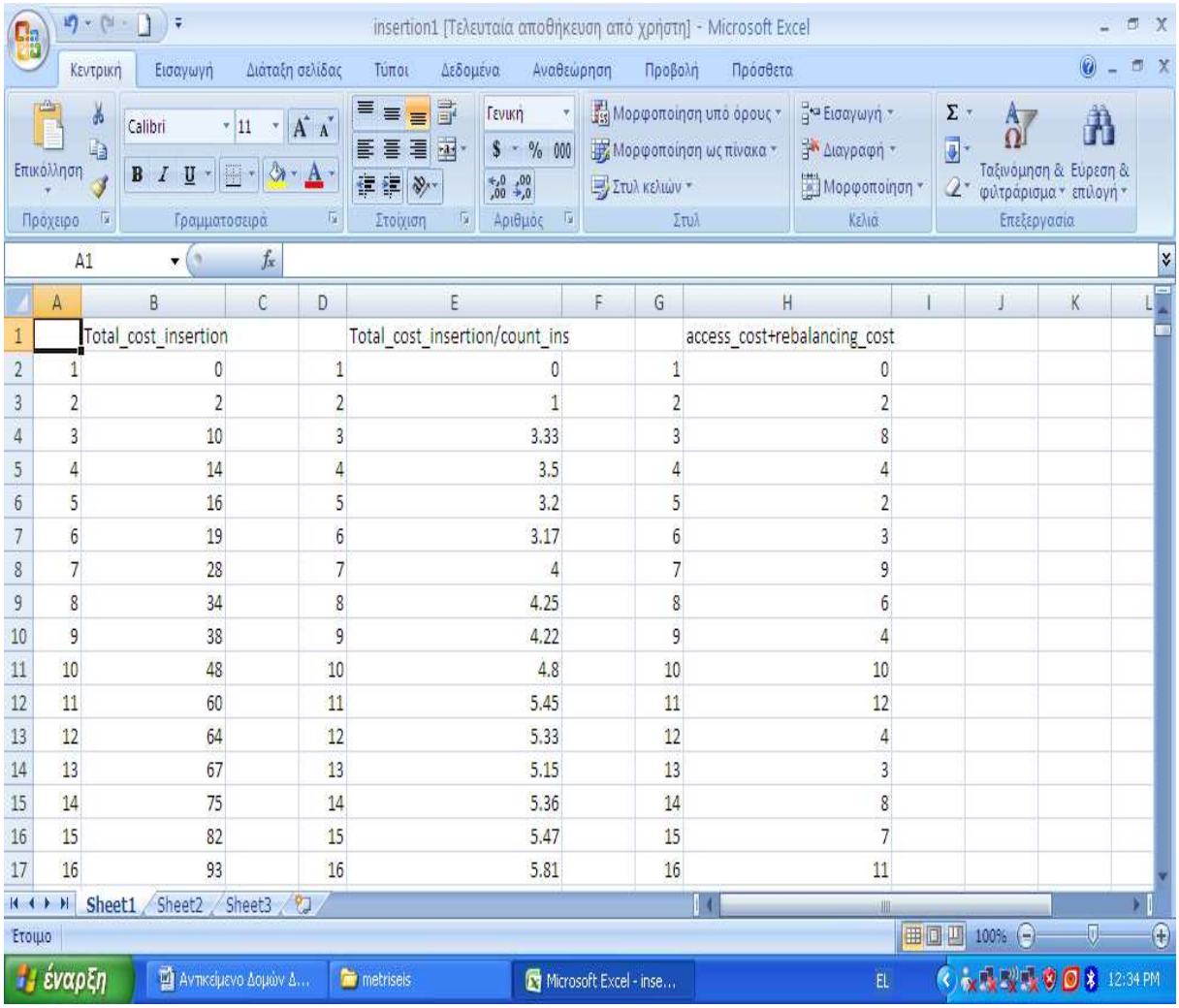
```
……..
update_rank(p, 1);
//αύξηση του rebalancing_cost κατά ένα
rebalancing_cost++;
// p is now a 1,2-node.
assert(check_ij_node(p, 1, 2));
q = p;
p = q->parent;
// αυτή η περίπτωση είναι μη-τερματικη
…….

…….
// Rotate: t is a 2-child, rotate at q and demote p.
if (rank_diff_t == 2) {
        rotate(q, !left_child);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
        update_rank(p, -1);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;

…….
```

Οι μεταβλητές ***Total_insertion_cost*** και ***Total_deletion_cost*** ορίζονται από το άθροισμα του access_cost και rebalancıng_cost, δηλαδή και για τις δύο μεταβλητές ισχύουν Total_insertion_cost += (access_cost +rebalancıng_cost); και Total_deletion_cost += (access_cost +rebalancıng_cost);. Διευκρινίζουμε ότι έπειτα από κάθε εισαγωγή και διαγραφή οι μεταβλητές access_cost και rebalancıng_cost μηδενίζονται. Ενδεικτικά παρατίθεται το κομμάτι του κώδικα για την εισαγωγή.

```
…….

else if (a1==2){
                printf("η επιλογή είναι ΕΙΣΑΓΩΓΗ");

                //δημιουργώ τυχαίους αριθμούς
                for(i=0;i<50000;i++){
                        b[i]=rand();
                        //ελέγχω να μην είναι ίδιοι
                        for(j=0;j<i;j++){
                                if(b[i]==b[j]){i--;}
                        }
                }
                for(i=0;i<50000;i++){
                count_ins++;
                Tree *t1=insert(b[i]);
                //fprintf(f2,"node\t%d\tme\trank\t\t%d\n",t1->key,
                //t1->rank);
                //fprintf(f1,"%d\n",access_cost+rebalancing_cost);
              Total_insertion_cost +=(access_cost+rebalancing_cost);
```

```
                    //fprintf(f2,"%d\n",Total_insertion_cost);
                    //fprintf(f3,"%.2f\n",(float)Total_insertion_cost/count_ins);

                    access_cost=0;
                    rebalancing_cost=0;
                    }
            }
```
……

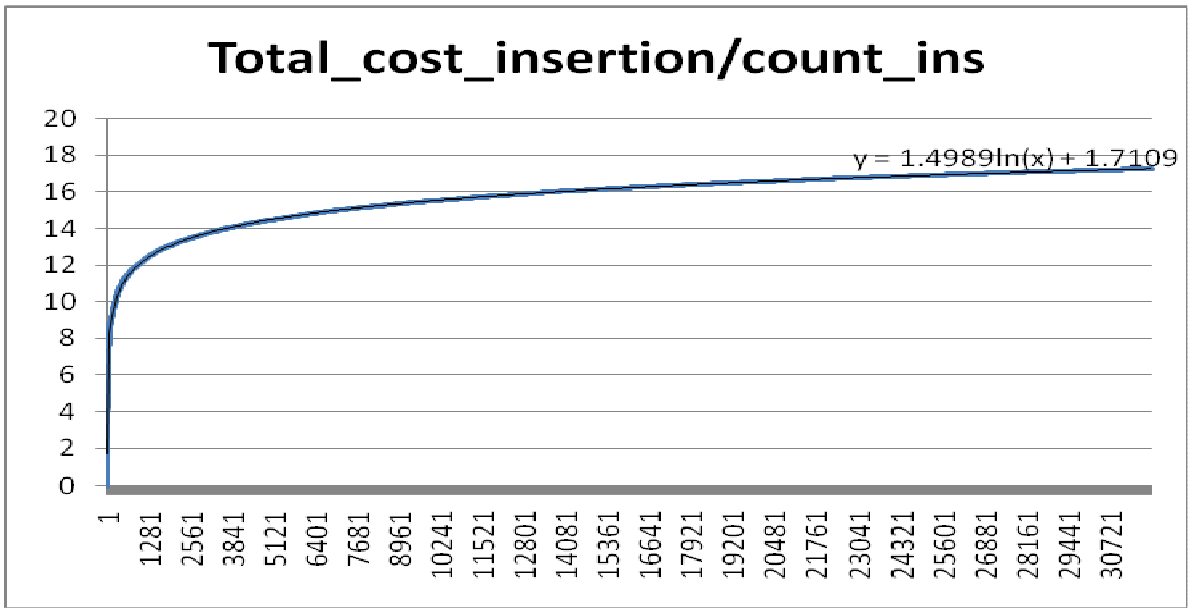
Τέλος αφού δημιουργήσουμε τυχαίους αριθμούς και τους κάνουμε τόσο εισαγωγή όσο και διαγραφή εξάγουμε τα αποτελέσματα των μεταβλητών και κάνουμε τις μετρήσεις όπως φαίνεται στα παρακάτω σχήματα.



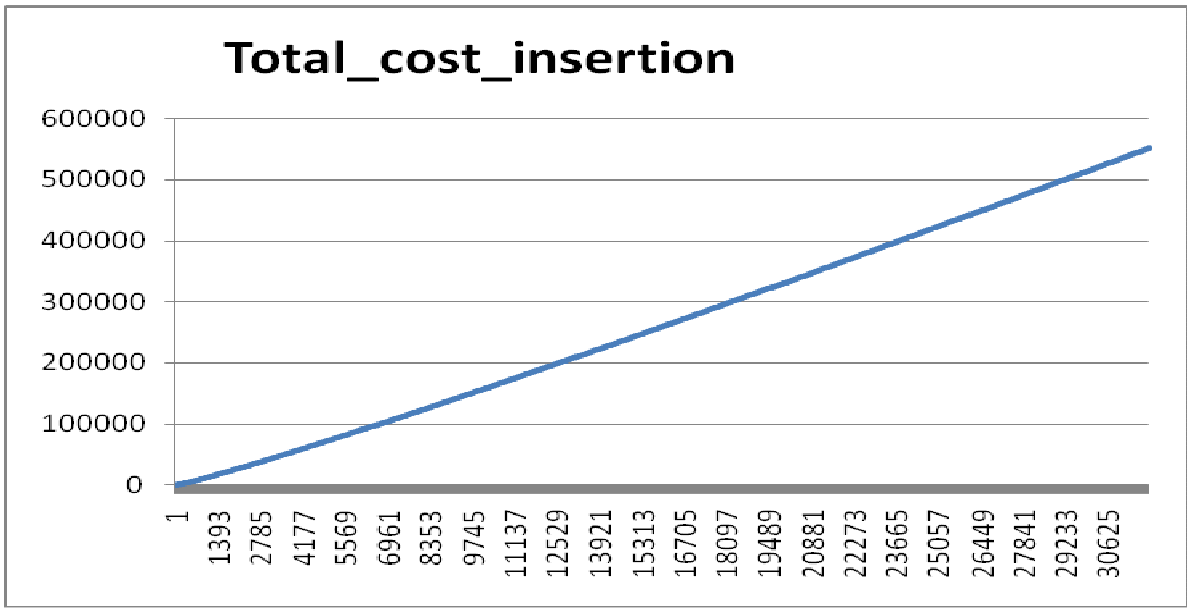
	A	B	C	D	E	F	G	H	I	J	K	L
1		Total_cost_insertion			Total_cost_insertion/count_ins			access_cost+rebalancing_cost				
2	1	0		1	0		1	0				
3	2	2		2	1		2	2				
4	3	10		3	3.33		3	8				
5	4	14		4	3.5		4	4				
6	5	16		5	3.2		5	2				
7	6	19		6	3.17		6	3				
8	7	28		7	4		7	9				
9	8	34		8	4.25		8	6				
10	9	38		9	4.22		9	4				
11	10	48		10	4.8		10	10				
12	11	60		11	5.45		11	12				
13	12	64		12	5.33		12	4				
14	13	67		13	5.15		13	3				
15	14	75		14	5.36		14	8				
16	15	82		15	5.47		15	7				
17	16	93		16	5.81		16	11				

Σχήμα 3.14: Εισάγουμε τα αποτελέσματα των μεταβλητών στο EXCEL για να τα επεξεργαστούμε.

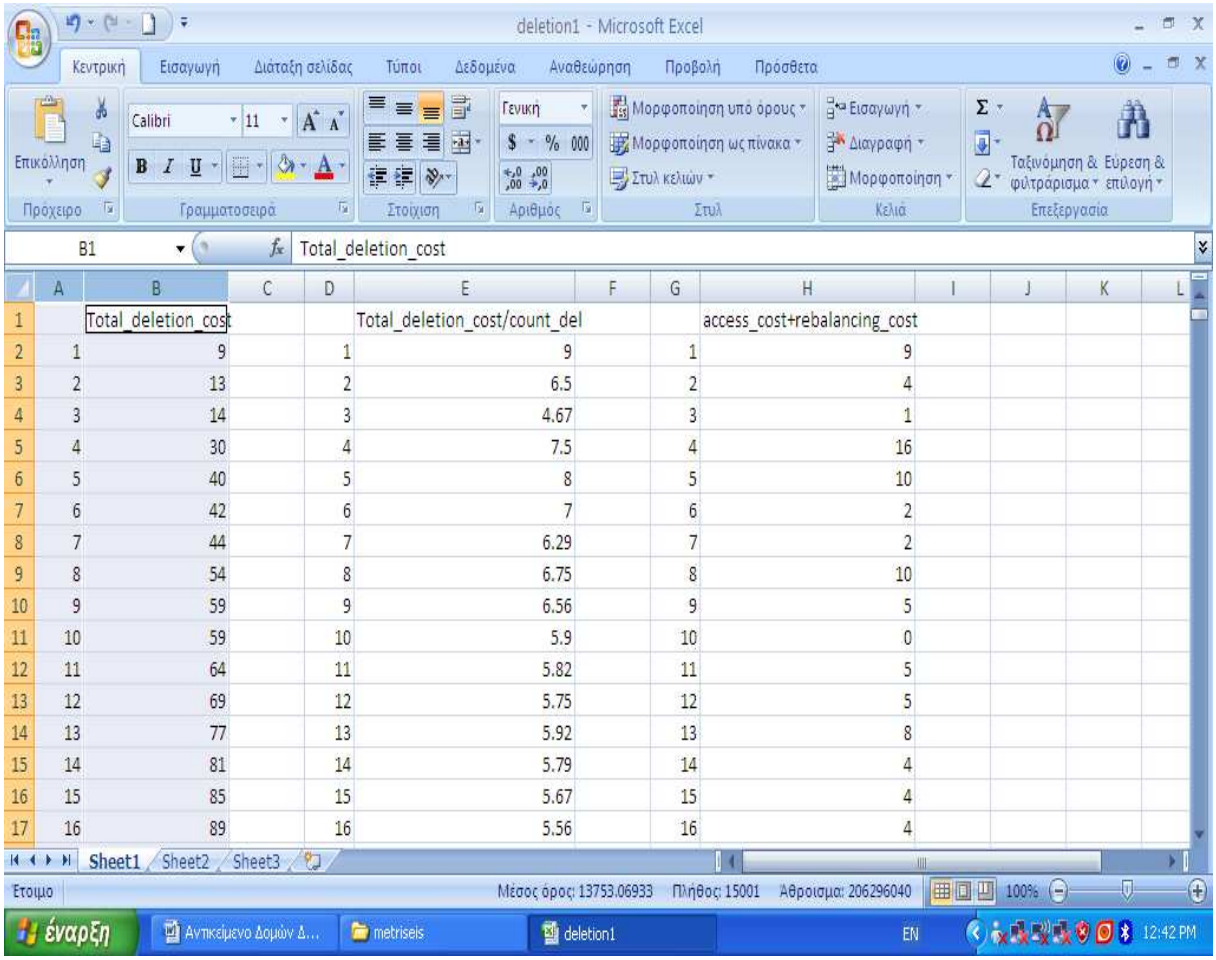
Συνεπώς προκύπτουν οι παρακάτω γραφικές παραστάσεις.



Σχήμα 3.15: Παρατηρούμε ότι η γραφική παράσταση του συνολικού κόστους εισαγωγής προς τον αριθμό των εισαγωγών σε σχέση με τον αριθμό των εισαγωγών Total_cost_insertion/count_ins είναι της μορφής y=1.4989ln(x)+1.7109, πράγμα που επιβεβαιώνει ότι τα rank-balanced trees εγγυώνται ένα λογαριθμικό όριο ύψους στην πράξη της εισαγωγής.

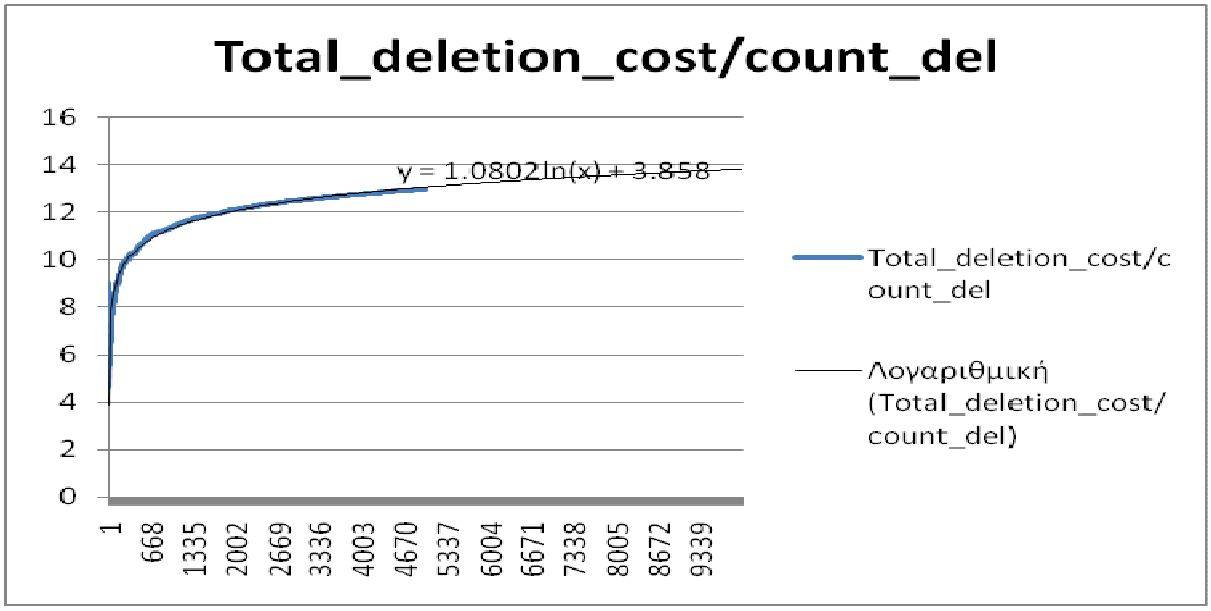


Σχήμα 3.16: Παρατηρούμε ότι η γραφική παράσταση του συνολικού κόστους εισαγωγής Total_cost_insertion σε σχέση με τον αριθμό των εισαγωγών είναι της μορφής nlogn.



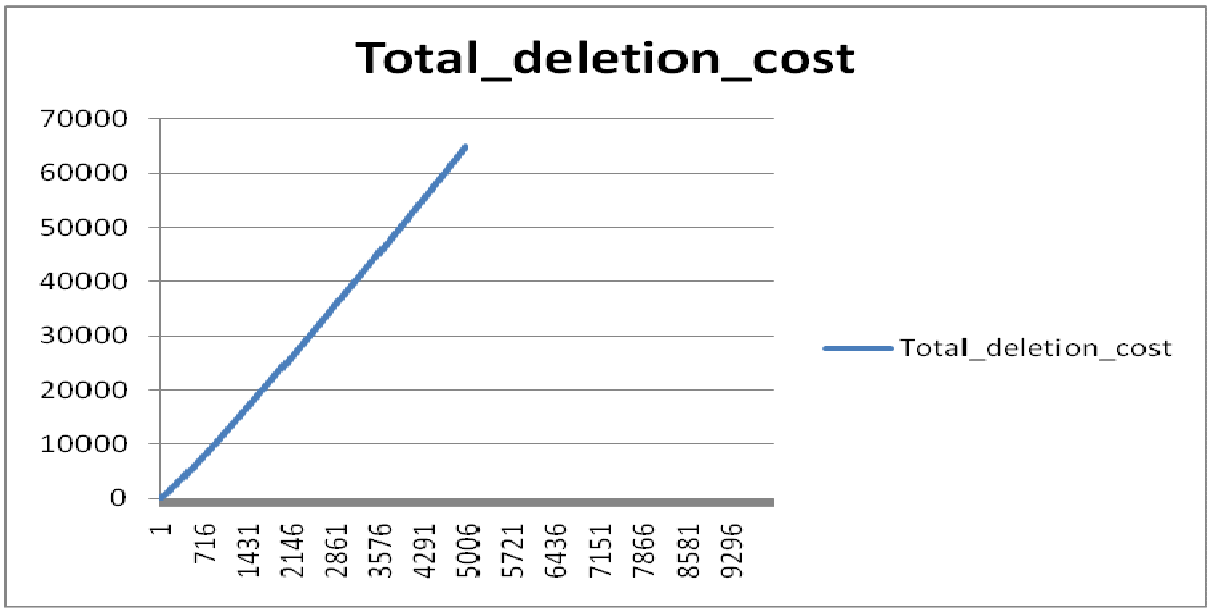
	A	B	C	D	E	F	G	H
1		Total_deletion_cost			Total_deletion_cost/count_del			access_cost+rebalancing_cost
2	1	9		1	9		1	9
3	2	13		2	6.5		2	4
4	3	14		3	4.67		3	1
5	4	30		4	7.5		4	16
6	5	40		5	8		5	10
7	6	42		6	7		6	2
8	7	44		7	6.29		7	2
9	8	54		8	6.75		8	10
10	9	59		9	6.56		9	5
11	10	59		10	5.9		10	0
12	11	64		11	5.82		11	5
13	12	69		12	5.75		12	5
14	13	77		13	5.92		13	8
15	14	81		14	5.79		14	4
16	15	85		15	5.67		15	4
17	16	89		16	5.56		16	4

Σχήμα 3.17: Εισάγουμε τα αποτελέσματα των μεταβλητών στο EXCEL για να τα επεξεργαστούμε.



Σχήμα 3.18: Παρατηρούμε ότι η γραφική παράσταση του συνολικού κόστους διαγραφής προς τον αριθμό των διαγραφών σε σχέση με τον αριθμό των διαγραφών Total_cost_deletion/count_ins είναι της μορφής

y=1.0802ln(x)+3.858, πράγμα που επιβεβαιώνει ότι τα rank-balanced trees εγγυώνται ένα λογαριθμικό όριο ύψους στην πράξη της διαγραφής.



Σχήμα 3.19 Παρατηρούμε ότι η γραφική παράσταση του συνολικού κόστους διαγραφής Total_cost_deletion σε σχέση με τον αριθμό των διαγραφών είναι της μορφής nlogn.

4. ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Adel'son-Vel'skii, G.M., Landis, E.M.: An algorithm for the organization of information. Sov. Math. Dokl. 3, 1259-1262 (1962)
2. Brown, M.R.: A storage scheme for height-balanced trees. Inf. Proc. Lett., 231-232(1978)
3. Guidas, L.J., Sedgewick, R.: A dichromatic framework for balanced tree. In: FOCS, pp. 8-21(1978)
4. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. SIAM J. on Comput., 33-43(1973)
5. Tarjan, R.E.: Amortized computational complexity. SIAM J. Algebraic and Disc. Methods 6, 306-318(1985)
6. Bernhard Haeupler, Siddhartha Sen and Robert E. Tarjan: Rank- Balanced Trees

5. ΠΑΡΑΡΤΗΜΑ

Στο παράρτημα παρατίθεται η υλοποίηση στην γλώσσα προγραμματισμού C των rank-balanced trees . Θα θέλαμε να ευχαριστήσουμε τον Siddhartha Sen για την βοήθεια που μας προσέφερε κατά την υλοποίηση και την διενέργεια των πειραμάτων.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>


#define MISSING_RANK                -1
#define IN_TREE_ID                  582
//συγκρίνουμε τα κλειδιά δύο κόμβων του δέντρου
#define compare(t1,t2)              ((t1)->key - (t2)->key)
//συγκρίνουμε το κλειδί με το κλειδί του κόμβου
#define compare_key(key,t1)          ((key) - (t1)->key)
#define set_rank(x,r)               { (x)->rank = r;  }
#define update_rank(x,d)             { (x)->rank += d; }

typedef struct tree_node Tree;

struct tree_node {
        Tree * left, * right;
        Tree * parent;
        int key;
        int rank;
        int check;
};

Tree *root=NULL;

//κόστος αναζήτησης
int access_cost=0;

//συνολικό κόστος εισαγωγής
int Total_insertion_cost=0;

//συνολικό κόστος διαγραφής
int Total_deletion_cost=0;

//επαναζυγιστικό κόστος
int rebalancing_cost=0;


// ελέγχουμε αν ο κόμβος x είναι  i,j-node.

int check_ij_node(Tree * x, int i, int j) {
        assert(x != NULL);
```

```
        int rank_diff_l = (x->left == NULL) ? x->rank - MISSING_RANK :
        x->rank - x->left->rank;
        int rank_diff_r = (x->right == NULL) ? x->rank - MISSING_RANK :
        x->rank - x->right->rank;
        return (((rank_diff_l == i) && (rank_diff_r == j)) || ((rank_diff_l == j) &&
        (rank_diff_r == i)));
}



// ελέγχουμε αν ο κόμβος x είναι φύλλο ή όχι

int is_leaf(Tree * x) {
        assert(x != NULL);
        return ((x->left == NULL) && (x->right == NULL));
}


// ανταλλάσσουμε τις τιμές των κόμβων x και y
void swap(Tree * x, Tree * y) {
        assertυ((x != NULL) && (y != NULL));
        int temp_key = x->key;
        x->key = y->key;
        y->key = temp_key;
}


// εκτελούμε rotation στην κατεύθυνση που του δίνουμε

void rotate(Tree * x, int left) {
        Tree * p = x->parent;
        assert(p != NULL);
        assert((left && p->right == x) || (!left && p->left == x));
        // Left rotation.
        if (left) {
                p->right = x->left;
                if (x->left != NULL) {
                        x->left->parent = p;
                }
                x->left = p;
        }
        // Right rotation.
        else {
                p->left = x->right;
                if (x->right != NULL) {
                        x->right->parent = p;
                }
                x->right = p;
        }
        Tree * pp = p->parent;
        x->parent = pp;
```

```
	if (pp != NULL) {
		if (pp->left == p) {
			pp->left = x;
		}
		else {
			assert(pp->right == p);
			pp->right = x;
		}
	}
	p->parent = x;
}



// επαναζυγίζουμε το δέντρο μετά από εισαγωγή

void rebalance_ins(Tree * q) {
	assert(q != NULL);
	Tree * p = q->parent;
	// If p is null or q is not a 0-child, stop.
	while ((p != NULL) && (p->rank - q->rank == 0)) {
		Tree * s = (p->left == q)? p->right : p->left;
		int rank_diff_s = (s == NULL) ? p->rank - MISSING_RANK :
		p->rank - s->rank;
		// Promote: if s is a 1-child, promote p; ο p μπορεί να γίνει //0-child
		if (rank_diff_s == 1) {
			update_rank(p, 1);
			//αύξηση του rebalancing_cost κατά ένα
			rebalancing_cost++;
			// p is now a 1,2-node.
			assert(check_ij_node(p, 1, 2));
			q = p;
			p = q->parent;
			// αυτή η περίπτωση είναι μη-τερματικη
		}
		else {
			Tree * t;
			// ελέγχουμε εάν ή όχι ο q είναι left
			 //child(χρησιμοποιούμε για να δείξουμε την κατεύθυνση του
			//rotation)
			int left_child = 0;
			if (p->left == q) {
				t = q->right;
				left_child = 1;
			}
			else {
				t = q->left;
			}
			int rank_diff_t = (t == NULL) ? q->rank - MISSING_RANK :
			q->rank - t->rank;
			// Rotate: t is a 2-child, rotate at q and demote p.
			if (rank_diff_t == 2) {
```

```
                        rotate(q, !left_child);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        update_rank(p, -1);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        // p and q are now 1,1-nodes.
                    assert(check_ij_node(p, 1, 1) && check_ij_node(q, 1, 1));
                }
                // Double rotate: t is a 1-child, rotate twice at t,
                //promote t and demote p and q.
                else {
                        assert(t != NULL);
                        assert(rank_diff_t == 1);
                        rotate(t, left_child);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        rotate(t, !left_child);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        update_rank(t, 1);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        update_rank(p, -1);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        update_rank(q, -1);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        // t is now a 1,1-node.
                        assert(check_ij_node(t, 1, 1));
                        // ο κόμβος p ή q είναι 1,1-node αλλιώς είναι και // τα
                        δύο φύλλα
                        assert(!(check_ij_node(p, 1, 1) && check_ij_node(q, 1,
                        1)) || (is_leaf(p) && is_leaf(q)));
                        // ούτε ο κόμβος p ούτε ο κόμβος q είναι 2,2-node.
                        assert(!check_ij_node(p, 2, 2) && !check_ij_node(q, 2,
                        2));
                }
                break;
            }
        }
}


//επαναζυγίζουμε το δέντρο μετά από διαγραφή

void rebalance_del(Tree * q, Tree * p) {
        assert(p != NULL);
        int rank_diff_q = (q == NULL) ? p->rank - MISSING_RANK : p->rank –
        q->rank;
```

```
// If p is null, or q is not a 3-child and p is not a 2,2-node of //rank 1,
stop.
while ((p != NULL) && ((rank_diff_q == 3) || (check_ij_node(p, 2, 2) &&
(p->rank == 1)))) {
	Tree * s = (p->left == q) ? p->right : p->left;
	int rank_diff_s = (s == NULL) ? p->rank - MISSING_RANK :
	p->rank - s->rank;
	// Demote: if s is a 2-child, demote p; ο κόμβος p μπορεί να //γίνει
	3-child
	if (rank_diff_s == 2) {
		update_rank(p, -1);
		// αύξηση του rebalancing_cost κατά ένα
		rebalancing_cost++;
		// p is either a 1,2-node or a leaf of rank 0.
		assert(check_ij_node(p, 1, 2) || (is_leaf(p) && (p->rank ==
		0)));
		q = p;
		p = q->parent;
		// If p is null, q has no rank difference (it is the //root);
		// θα τερματίσει στο επόμενο loop
		if (p != NULL) {
			rank_diff_q = p->rank - q->rank;
		}
		//  αυτή η περίπτωση είναι μη-τερματική
	}
	else {
		// ελέγχουμε αν ο q είναι left_child( το χρησιμοποιούμε // για
		να δείξουμε την κατεύθυνση του rotation
		int left_child = (p->left == q) ? 1 : 0;
		// q is a 3-child and s is a non-missing 1-child
		assert((rank_diff_q == 3) && (s != NULL) && (rank_diff_s
		== 1));
		Tree * t = left_child ? s->left : s->right;
		Tree * u = left_child ? s->right : s->left;
		int rank_diff_t = (t == NULL) ? s->rank - MISSING_RANK :
		s->rank - t->rank;
		int rank_diff_u = (u == NULL) ? s->rank - MISSING_RANK :
		s->rank - u->rank;
		// Double demote: t and u are 2-children, demote p and s.
		if ((rank_diff_t == 2) && (rank_diff_u == 2)) {
			update_rank(p, -1);
			// αύξηση του rebalancing_cost κατά ένα
			rebalancing_cost++;
			update_rank(s, -1);
			// αύξηση του rebalancing_cost κατά ένα
			rebalancing_cost++;
			// p is a 1,2-node and s is a 1,1-node.
			assert(check_ij_node(p, 1, 2) && check_ij_node(s, 1,
			1));
			q = p;
			p = q->parent;
			// If p is null, q has no rank difference (it is //the root);
```

```
        // θα τερματίσει στο επόμενο loop
        if (p != NULL) {
                rank_diff_q = p->rank - q->rank;
        }
        //αυτή η περίπτωση είναι μη-τερματική
}
// Rotate: u is a 1-child, rotate at s, promote s and //demote
p.  If t is
// missing, demote p again.
else if (rank_diff_u == 1) {
        rotate(s, left_child);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
        update_rank(s, 1);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
        update_rank(p, -1);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
        if (t == NULL) {
                update_rank(p, -1);
                // αύξηση του rebalancing_cost κατά ένα
                rebalancing_cost++;
        }
        // ο p δεν είναι 1,1-node εκτός αν είναι φύλλο με
        //rank 0
        assert(!check_ij_node(p, 1, 1) || (is_leaf(p) && (p-
        >rank == 0)));
        // ο κόμβος s είναι 1,2-node, ή ο p είναι φύλλο και //ο
        s είναι 2,2-node.
        assert(check_ij_node(s, 1, 2) || (is_leaf(p) &&
        check_ij_node(s, 2, 2)));
        break;
}
// Double rotate: t is a 1-child and u is a 2-child, //rotate at t
twice,
// promote t twice, demote s, and demote p twice.
else {
        assert(t != NULL);
        assert((rank_diff_t == 1) && (rank_diff_u == 2));
        rotate(t, !left_child);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
        rotate(t, left_child);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
        update_rank(t, 2);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
        update_rank(s, -1);
        // αύξηση του rebalancing_cost κατά ένα
        rebalancing_cost++;
```

```
                        update_rank(p, -2);
                        // αύξηση του rebalancing_cost κατά ένα
                        rebalancing_cost++;
                        // t is a 2,2 node.
                        assert(check_ij_node(t, 2, 2));
                        // p and s are not 2,2-nodes.
                        assert(!check_ij_node(p, 2, 2) && !check_ij_node(s, 2,
                        2));
                        break;
                }
        }
    }
}


// αναζήτηση ενός στοιχείου με βάση το κλειδί του, αν δεν βρεθεί το
// στοιχείο επιστρέφεται το τελευταίο στοιχείο κατά μήκος του μονοπατιού  //
αναζήτησης

Tree * access(int key) {
    assert(root != NULL);
    Tree * x = root, * prev_x=NULL;
    while ((x != NULL) && (x->key != key)) {
        prev_x = x;
        if (compare_key(key, x) < 0) {
            x = x->left;
        }
        else {
            x = x->right;
        }
        // αύξηση του access_cost κατά ένα
        access_cost++;
    }

    // το στοιχείο δεν βρέθηκε και επιστρέφουμε το τελευταίο στοιχείο
    // κατά μήκος του μονοπατιού αναζήτησης
    if (x == NULL) {
        assert(prev_x != NULL);
        x = prev_x;
    }
    return x;
}


// εισαγωγή ενός στοιχείου με βάση το κλειδί του.(αν δεν έχει δημιουργηθεί
 // δέντρο είναι το πρώτο στοιχείο που εισέρχεται)
// και επιστρέφεται το νέο εισερχόμενο στοιχείο

Tree * insert(int key) {
    Tree * p;
    // αν υπάρχει δέντρο ελέγχω αν το στοιχείο υπάρχει ήδη
    if (root != NULL) {
```

```
		p = access(key);
		if (p->key == key) {
			printf("Warning: an item with key %d already exists in tree
			with root key %d\n", key, root->key);
			return NULL;
		}
	}
	Tree * new = (Tree *)malloc(sizeof(Tree));
	if (new == NULL) {
		printf("Error: ran out of space\n"); exit(1);
	}
	// δημιουργούμε ένα νέο στοιχείο με το συγκεκριμένο κλειδί
	//memset(new, (int)NULL, sizeof(*new));
	new->key = key;
	set_rank(new, 0);
	new->check = IN_TREE_ID;
	// εισάγουμε το στοιχείο μέσα στο δέντρο αν υπάρχει δέντρο
	if (root != NULL) {
		// ο p πρέπει να δείχνει στον πατέρα του νέου στοιχείου στο
		//δέντρο
		assert((p != NULL) && (p->key != key));
		if (compare(new, p) < 0) {
			assert(p->left == NULL);
			p->left = new;
		}
		else {
			assert(p->right == NULL);
			p->right = new;
		}
		new->parent = p;
		rebalance_ins(new);
		// ενημερώνουμε το δείκτη στη ρίζα και αν έχει αλλάξει η  ρίζα //
		πρέπει να αλλάξουμε και το δείκτη προς τον πατέρα της ρίζας
		//
		if (root->parent != NULL) {
			root = root->parent;
		}
		assert(root->parent == NULL);
	}
	else {
		root = new;
	}
	return new;
}

//διαγράφουμε το στοιχείο από το δέντρο, επιστρέφουμε ένα δείκτη προς το
//διαγραμμένο κόμβο ο οποίος μπορεί να μην είναι ίδιος με αυτόν που δώσαμε
//σαν παράμετρο στην συνάρτηση
Tree * delete(Tree * x) {
	assert((x != NULL) && (root != NULL));
	// αποτυγχάνει η αναζήτηση αν αυτός είναι ήδη διαγραμμένος
	if (x->check != IN_TREE_ID) {
```

```
        printf("Warning: Attempting to delete a previously deleted
        item!\n");
        return NULL;
}
// εάν ο x έχει δύο (non-missing)παιδιά τον αλλάζουμε με τον //απόγονο
του
if ((x->left != NULL) && (x->right != NULL)) {
        Tree * s = x->right;
        while (s->left != NULL) {
                s = s->left;
        }
        swap(x,s);
        // το στοιχείο προς διαγραφή είναι το s
        x = s;
}
// σε αυτό το σημείο ο x δεν έχει δύο (non-miising) παιδιά
assert((x->left == NULL) || (x->right == NULL));
// εάν ο x είναι φύλλοαντικαθίσταται από άνα missing node(NULL)
// αν ο x έχει μόνο ένα παιδί αντικαθίσταται από το παιδί του
Tree * p = x->parent;
Tree * replace = NULL;
if ((x->left == NULL) ^ (x->right == NULL)) {
        replace = (x->left == NULL) ? x->right : x->left;
        // The replacement node's parent is x's parent.
        replace->parent = p;
}
if (p != NULL) {
        if (p->left == x) {
                p->left = replace;
        }
        else {
                p->right = replace;
        }
        rebalance_del(replace, p);
        // ενημερώνουμε το δείκτη στην ρίζα και αν έχει αλλάξει η ρίζα
        //πρέπει να αλλάξουμε και το δείκτη προς την ρίζα

        if (root->parent != NULL) {
                root = root->parent;
        }
        assert(root->parent == NULL);
}
// εάν ο πατέρας είναι κενός τότε ο root node διαγράφηκε και
// πρέπει να ενημερώσουμε τον root pointer

else {
        root = replace;
}
// διαγράφουμε το IN_TREE_ID πριν επιστρέψουμε το διαγραμμένο
//στοιχείο
x->check = 0;
return x;
```

```
}
int main (int argc, char *argv[]){

int a1, count_ins=0, count_del=0;
int i, j, b[50000];
FILE *f1 ,*f2, *f3;
f1=fopen(argv[1],"w");if(f1==NULL)exit(1);
f2=fopen(argv[2],"w");if(f2==NULL)exit(1);
f3=fopen(argv[3],"w");if(f3==NULL)exit(1);
        while(1) {
                printf("διάλεξε επιλογή\ηγια ΑΝΑΖΗΤΗΣΗ  1\ηγια ΕΙΣΑΓΩΓΗ
                        2\ηγια ΔΙΑΓΡΑΦΗ   3\ηγια ΚΛΕΙΣΙΜΟ αλλο\n");
        scanf("%d",&a1);
        if (a1==1){
                        printf("η επιλογή είναι ΑΝΑΖΗΤΗΣΗ\n");
                        int k,l,a[10];
                        //δημιουργώ τυχαίους αριθμούς
                        for(k=0;k<10;k++){
                                a[k]=rand()%10;
                                //ελέγχω να μην είναι οι ίδιοι
                                for(l=0;l<k;l++){
                                        if(a[k]==a[l]){k--;}
                                }
                        }
                        for(k=0;k<10;k++){
                        Tree *t0=access(a[k]);
                        //εκτύπωση μονοπατιού που οδηγεί στον κόμβο αναζήτησης
                        Tree *t2 = root;
                        fprintf(f1,"root\t%d\tme\trank\t%d\n",root->key,
                        root->rank);
                                while ((t2 != NULL) && (t2->key != a[k])) {
                                if (t2->key > a[k]) { t2 = t2->left; }
                                else /* t->key < a[k] */ { t2 = t2->right; }

                                //fprintf(f1,"node\t%d\tme\trank\t%d\n",t2->key,
                                t2->rank);
                                }
                       fprintf(f1,"το κόστος αναζήτησης είναι\t%d\n\n",access_cost);
                        access_cost=0;
                        }
                }
                else if (a1==2){
                        printf("η επιλογή είναι ΕΙΣΑΓΩΓΗ");

                        //δημιουργώ τυχαίους αριθμούς
                        for(i=0;i<50000;i++){
                                b[i]=rand();
                                //ελέγχω να μην είναι ίδιοι
                                for(j=0;j<i;j++){
                                        if(b[i]==b[j]){i--;}
                                }
                        }
```

```
		for(i=0;i<50000;i++){
		count_ins++;
		Tree *t1=insert(b[i]);
		//fprintf(f2,"node\t%d\tme\trank\t\t%d\n",t1->key,
		//t1->rank);
		//fprintf(f1,"%d\n",access_cost+rebalancing_cost);
		Total_insertion_cost +=(access_cost+rebalancing_cost);
		//fprintf(f2,"%d\n",Total_insertion_cost);
		//fprintf(f3,"%.2f\n",(float)Total_insertion_cost/count_ins);

		access_cost=0;
		rebalancing_cost=0;
		}

	}

	else if (a1==3){

		printf("η επιλογή είναι ΔΙΑΓΡΑΦΗ\n”);
	//i=0,j=0;
		//δημιουργώ τυχαίους αριθμούς
		//for(i=0;i<5000;i++){
			//b[i]=rand();
			//ελέγχω να μην είναι ίδιοι
			//	for(j=0;j<i;j++){
			//		if(b[i]==b[j]){i--;}
			//	}
			//}

		//αναζήτηση του item που διαγράψαμε
		for(i=0;i<5000;i++){
		count_del++;
		Tree *t3=root;
		while ((t3 != NULL) && (t3->key != b[i])) {
			if (t3->key > b[i]) { t3 = t3->left; }
			else /* t->key < z[x] */ { t3 = t3->right; }
			access_cost++;
		}
		Tree *t4=delete(t3);
		//fprintf(f3,"o node που διαγράφηκε είναι o\t\t //%d",z[x]);
		fprintf(f1,"%d\n",access_cost+rebalancing_cost);
		Total_deletion_cost +=	(access_cost+rebalancing_cost);

		fprintf(f2,"%d\n",Total_deletion_cost);
		fprintf(f3,"%.2f\n",(float)Total_deletion_cost/count_del);

		access_cost=0;
		rebalancing_cost=0;
		}
	}
	else	{ return 0;}
}
```

```
}
```