ΕΝΟΤΗΤΑ 5
ΣΥΝΟΛΑ - ΛΕΞΙΚΑ

Σύνολα (Sets)

- Τα μέλη ενός συνόλου προέρχονται από κάποιο χώρο U αντικειμένων/στοιχείων (π.χ., σύνολα αριθμών, λέξεων, ζευγών αποτελούμενα από έναν αριθμό και μια λέξη, κ.ο.κ.).
- Αν S είναι ένα σύνολο και x είναι ένα αντικείμενο του χώρου από όπου το S προέρχεται, είτε $x \in S$ ή $x \notin S$.
- Ένα σύνολο δεν μπορεί να περιέχει το ίδιο στοιχείο 2 ή περισσότερες φορές.
- Σύνολα που περιέχουν πολλαπλά στιγμιότυπα του ίδιου στοιχείου λέγονται πολυ-σύνολα (multi-sets).

Χρησιμότητα

Πολλές εφαρμογές χρησιμοποιούν σύνολα και απαιτούν να είναι δυνατή η απάντηση ερωτήσεων του στυλ «Είναι το στοιχείο $x \in S$»;

Παραδείγματα

- Υπάρχει αυτός ο εργαζόμενος στη βάση δεδομένων των εργαζομένων;
- Υπάρχει αυτό το τηλέφωνο στον ηλεκτρονικό τηλεφωνικό κατάλογο;
- Υπάρχει αυτή η κράτηση στη βάση δεδομένων μιας αεροπορικής ή ακτοπλοϊκής εταιρείας;

Λειτουργίες Συνόλων

- **MakeEmptySet()**: Επιστρέφει το κενό σύνολο ∅.
- **IsEmptySet(S)**: Επιστρέφει true αν S είναι το κενό σύνολο και false διαφορετικά.
- **Insert(x,S)**: Προσθέτει το στοιχείο x στο σύνολο S ή δεν πραγματοποιεί καμία ενέργεια αν $x \in S$.
- **Delete(x,S)**: Διαγράφει το στοιχείο x από το S. Δεν πραγματοποιεί καμία αν $x \notin S$.
- **Member(x,S)**: επιστρέφει true αν το x είναι μέλος του συνόλου S και false διαφορετικά.
- **Size(S)**: Επιστρέφει |S|, δηλαδή τον αριθμό των στοιχείων του S.
- **Union(S,T)**: επιστρέφει $S \cup T$, δηλαδή το σύνολο που αποτελείται από τα στοιχεία εκείνα που είναι μέλη είτε του S ή του T.
- **Intersection(S,T)**: επιστρέφει $S \cap T$, δηλαδή το σύνολο που αποτελείται από τα στοιχεία εκείνα που είναι μέλη και του S και του T.
- **Difference(S,T)**: Επιστρέφει $S \setminus T$, δηλαδή το σύνολο των στοιχείων που ανήκουν στο S αλλά δεν ανήκουν στο T.
- **Equal(S,T)**: Επιστρέφει true αν S=T και false διαφορετικά.
- **Iterate(S,F)**: Εφαρμόζει τη λειτουργία F σε κάθε στοιχείο του S.

Για χώρους στοιχείων στους οποίους ορίζεται η ιδιότητα της γραμμικής διάταξης:

- **Min(S) (Max(S))**: επιστρέφει το μικρότερο (μεγαλύτερο) στοιχείο του S.

Σύνολα – Λεξικά

Θεωρούμε ότι κάθε στοιχείο του συνόλου είναι ένα ζεύγος <K,I> όπου K είναι το κλειδί που χαρακτηρίζει μοναδικά στοιχείου, και I είναι πληροφορίες (δεδομένα, data) που συνοδεύουν το στοιχείο με κλειδί K.

Θα επικεντρωθούμε στην υλοποίηση του αφηρημένου τύπου δεδομένων που υποστηρίζει κύρια τις παρακάτω λειτουργίες σε σύνολα:

- MakeEmptySet()
- IsEmptySet()
- Insert(K,I,S)
- Delete(K,S)
- LookUp(K,S): Δεδομένου ενός κλειδιού K, επιστρέφει τα δεδομένα I, τέτοια ώστε <K,I> $\in$ S. Αν δεν υπάρχει στοιχείο με κλειδί K στο S, επιστρέφει nill.

Λεξικά

Ο αφηρημένος τύπος δεδομένων που υποστηρίζει μόνο τις παραπάνω λειτουργίες (MakeEmptySet, IsEmptySet, Insert, Delete, και LookUp) λέγεται **λεξικό**.

Υλοποίηση Λεξικών με Λίστες

Αποθήκευση των στοιχείων του συνόλου σε μια (μη-ταξινομημένη) λίστα:

- Στατική λίστα (χρήση πίνακα)
- Συνδεδεμένη Λίστα (απλά ή διπλά συνδεδεμένη).

Τα στοιχεία βρίσκονται στη λίστα διατεταγμένα βάσει της σειράς εισαγωγής τους.

Υλοποίηση Λειτουργιών

Συνδεδεμένη Λίστα

Χρονική Πολυπλοκότητα LookUp(): Θ(n)
Στη χειρότερη περίπτωση το στοιχείο είναι το τελευταίο στη λίστα!

Χρονική Πολυπλοκότητα Insert(): Θ(n)
Θα πρέπει να προηγηθεί αναζήτηση και αν το στοιχείο υπάρχει να μην εισαχθεί ξανά στη λίστα.

Χρονική Πολυπλοκότητα Delete(): Θ(n)
Αναζήτηση του στοιχείου και διαγραφή του -> Η αναζήτηση κοστίζει O(n).

Στατική Λίστα

Το μέγιστο πλήθος στοιχείων του συνόλου πρέπει να είναι γνωστό εξ αρχής.
Ποια είναι η χρονική πολυπλοκότητα των LookUp, Insert, Delete;

Υλοποίηση Λεξικών με Λίστες - Αναμενόμενο Κόστος

Υποθέτουμε πως η πιθανότητα p_j η LookUp() να ψάχνει για το στοιχείο x_j είναι η ίδια για κάθε j, $1 \leq j \leq n$. Έτσι, αν έχουμε n στοιχεία, η πιθανότητα είναι 1/n.

Έστω ότι c_j είναι το κόστος που πληρώνουμε για να βρούμε το στοιχείο $x_j \Rightarrow c_j = j$.

Το αναμενόμενο κόστος είναι το άθροισμα των $(c_j * p_j)$ για κάθε j:

Αναμενόμενο Κόστος =

$1*1/n+2*1/n+...+n*1/n = (1+2+...+n)/n = n(n+1)/(2n) = (n+1)/2 = \Theta(n)$!!

Διαφορετικές Πιθανότητες για διαφορετικά κλειδιά

$K_1, \ldots, K_n$: τα κλειδιά στο σύνολο σε φθίνουσα διάταξη ως προς τη συχνότητα με την οποία αναζητούνται μέσω της LookUp().

$p_1 \geq p_2 \geq \ldots \geq p_n$: πιθανότητα μια LookUp() να ψάχνει για το $K_1, K_2, \ldots, K_n$, αντίστοιχα.

Ο αναμενόμενος χρόνος αναζήτησης ελαχιστοποιείται όταν τα στοιχεία έχουν τη διάταξη $K_1, K_2, \ldots, K_n$ στη λίστα:

$$C_{opt} = \sum_{i=1}^{n} i p_i$$

Γιατί αυτό είναι βέλτιστο;

Αναμενόμενο Κόστος

Ας υποθέσουμε, για να καταλήξουμε σε άτοπο, ότι η διάταξη των κλειδιών που οδηγεί στο ελάχιστο αναμενόμενο πλήθος συγκρίσεων είναι $K_{m_1}, K_{m_2}, \ldots, K_{m_n}$, όπου η ακολουθία $m_1, \ldots, m_n$, αποτελεί μετάθεση της $1, \ldots, n$ και ότι $p_{m_i} < p_{m_j}$ για κάποιο $i < j$.

Τότε, ανταλλάσοντας τις θέσεις των K_{m_i} και K_{m_j} στην ακολουθία θα μείωνε το αναμενόμενο πλήθος συγκρίσεων κατά

$$ip_{m_i} + jp_{m_j} - ip_{m_j} - jp_{m_i} = (j-i)(p_{m_j}-p_{m_i}) > 0$$

Άρα, η διάταξη $K_{m_1}, K_{m_2}, \ldots, K_{m_n}$ δεν οδηγεί στο ελάχιστο αναμενόμενο πλήθος συγκρίσεων, το οποίο αντιτίθεται στην υπόθεση.

Ευριστικά

- Η πραγματική κατανομή πιθανότητας συνήθως δεν είναι γνωστή.
- Το λεξικό μπορεί να αλλάζει μέγεθος κατά τη χρήση του.
- Η μελέτη πιθανοτικών μοντέλων κάποιες φορές απέχει αρκετά από το τι γίνεται στην πράξη!

Ευριστικό "Move-To-Front" (Ευριστικό «Μετακίνησης στην Αρχή»)

«Μετά από κάθε επιτυχημένη αναζήτηση, το στοιχείο που βρέθηκε μετακινείται στην αρχή της λίστας.»

Χρονική Πολυπλοκότητα; (α) Συνδεδεμένη Λίστα; (β) Στατική Λίστα;

Ευριστικό «Αλληλομετάθεσης» (Transpose)

«Μετά από κάθε επιτυχημένη αναζήτηση, το στοιχείο που βρέθηκε μετακινείται μια θέση προς την αρχή της λίστας (δηλαδή ανταλλάσσεται με το προηγούμενό του στη λίστα)».

Σύγκριση μεταξύ Ευριστικών

- Το ευριστικό Transpose αποδεικνύεται ότι έχει καλύτερο αναμενόμενο κόστος από το MoveToFront.
- Ωστόσο, το Transpose σταθεροποιείται σε μια σταθερή "καλή" κατάσταση πιο αργά από το MoveToFront.

Ευριστικό «Μετακίνησης στην Αρχή» - Αναμενόμενο Κόστος

Ας υποθέσουμε ότι η διαδικασία αναζήτησης κλειδιών έχει πραγματοποιηθεί για αρκετά μεγάλο χρονικό διάστημα, ώστε όλα τα κλειδιά να έχουν αναζητηθεί αρκετές φορές και η λίστα να βρίσκεται σε κάποιου είδους «σταθερή» κατάσταση.

p(i,j): πιθανότητα ότι το κλειδί K_i προηγείται του K_j στη λίστα

Ποια είναι η τιμή της p(i,j) συναρτήσει των p_i και p_j;

Προκειμένου το K_i να βρίσκεται πριν το K_j στη λίστα θα πρέπει η τελευταία LookUp(K_i,S) να έχει συμβεί πιο πρόσφατα από την τελευταία LookUp(K_j,S).

Αν επικεντρωθούμε στην τελευταία LookUp που πραγματοποιείται για κάποιο από τα κλειδιά K_i και K_j και αγνοήσουμε τις υπόλοιπες, τότε p(i,j) είναι η πιθανότητα αυτή η LookUp να αναζητά το K_i.

Άρα, $p(i,j) = p_i / (p_i+p_j)$.

Ευριστικό «Μετακίνησης στην Αρχή» - Αναμενόμενο Κόστος

Το αναμενόμενο πλήθος κλειδιών που προηγούνται του K_j στη λίστα είναι επομένως $\sum_{i \neq j} p(i, j)$.

Άρα, το αναμενόμενο πλήθος συγκρίσεων για να βρεθεί το κλειδί K_j είναι

$1 + \sum_{i \neq j} p(i, j)$.

Καταλήγουμε πως το αναμενόμενο πλήθος συγκρίσεων που απαιτούνται για την εύρεση ενός κλειδιού είναι:

$$C_{MTF} = \sum_{j=1}^{n} p_j (1 + \sum_{i \neq j} p(i, j)) = \sum_{j=1}^{n} p_j + \sum_{j=1}^{n} p_j \sum_{i \neq j} p(i, j) = 1 + \sum_{i \neq j} p_j p(i, j)$$

$$= 1 + \sum_{i \neq j} \frac{p_i p_j}{p_i + p_j} = 1 + 2 \sum_{i < j} \frac{p_i p_j}{p_i + p_j}$$

Πως συγκρίνεται το C_{MFT} με το C_{OPT};

$$\sigma = \sum_{i<j} \frac{p_i p_j}{p_i + p_j} = \sum_{j=1}^{n} p_j \sum_{1 \leq i < j} \frac{p_i}{p_i + p_j} \leq \sum_{j=1}^{n} p_j (j - 1), \qquad since \frac{p_i}{p_i + p_j} \leq 1, \forall i, j$$

$$= C_{OPT} - 1, since \sum_{j=1}^{n} p_j = 1.$$

Το αναμενόμενο κόστος του ευριστικού MoveToFront είναι το πολύ 2 φορές χειρότερο από εκείνο του βέλτιστου αλγόριθμου!

Άρα, $C_{MFT}/C_{OPT} \leq (1+2\sigma)/(1+\sigma) = 2 - 1/(1+\sigma) < 2$.

Ταξινομημένες Στατικές Λίστες

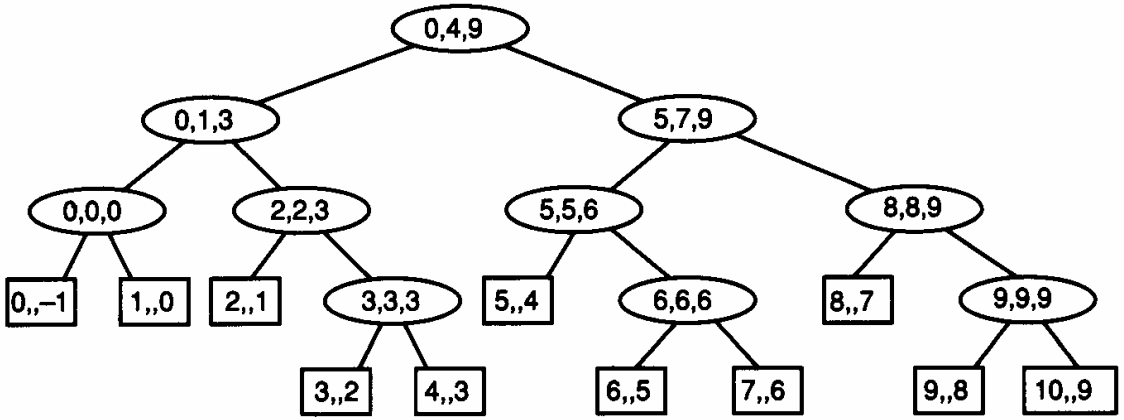
- Χρήση Πίνακα για αποθήκευση των στοιχείων του συνόλου.
- Κάθε στοιχείο του πίνακα είναι ένα struct με πεδία Key και data.
- Τα στοιχεία του πίνακα είναι ταξινομημένα βάσει του κλειδιού τους.
- Χρήση δυαδικής αναζήτησης για την υλοποίηση της LookUp().

BinarySearch

Type *BinarySearchLookUp*(**key** K, **table** T[0..n-1])

```
/* Return information stored with key K in T, or nill if K is not in T */
left = 0;
right = n-1;
repeat forever
        if (right < left) then
                return nill;
        else
                middle = ⌊(left+right)/2⌋;
                if (K == T[middle]->Key) then
                        return T(middle]->data;
                else if (K < T[middle]->Key) then
                        right = middle-1;
                else left = middle+1;
```

Δυνατές Εκτελέσεις BinarySearchLookUp

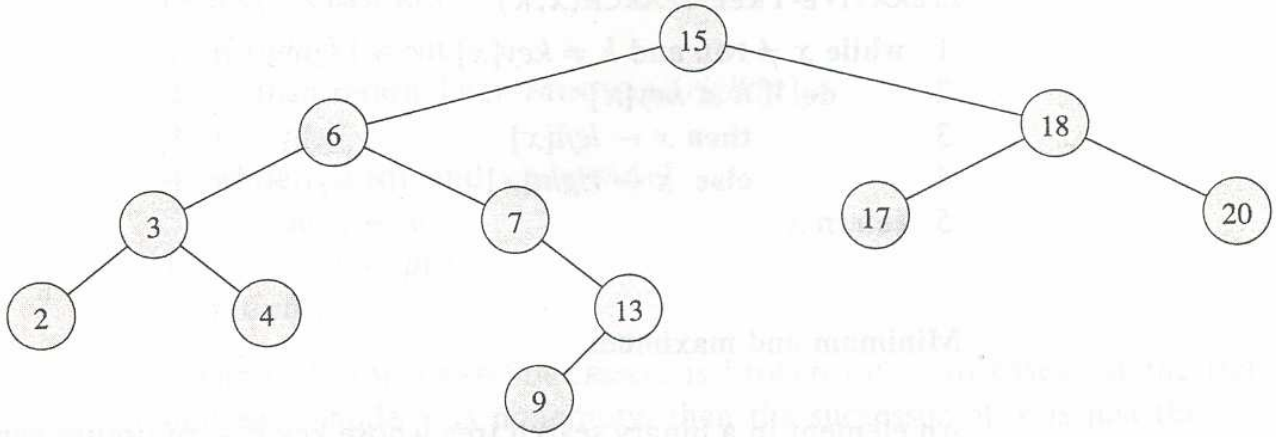


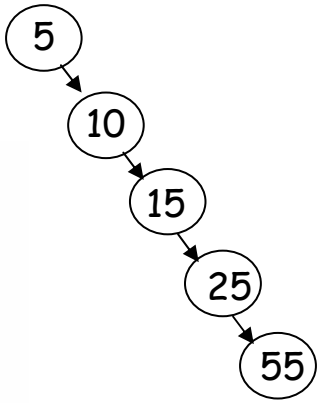
- **Κυκλικοί κόμβοι:** εσωτερικοί
- **Τετραγωνισμένοι κόμβοι:** εξωτερικοί

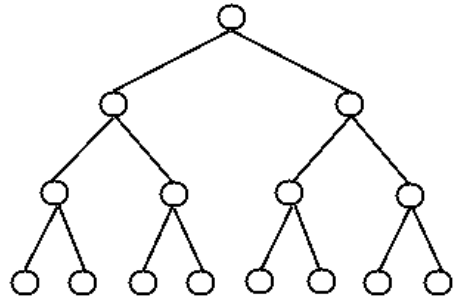
- Το δένδρο περιγράφει όλες τις δυνατές εκτελέσεις της BinarySearch() σε ένα πίνακα 10 στοιχείων.

Θεώρημα 1
Ο αλγόριθμος δυαδικής αναζήτησης εκτελεί O(logn) συγκρίσεις για κάθε αναζήτηση στοιχείου σε πίνακα με n στοιχεία.

Ταξινομημένα Δυαδικά Δένδρα (Δένδρα Δυαδικής Αναζήτησης)







Είναι δυαδικά δένδρα στα κλειδιά των οποίων είναι ορισμένη μια γραμμική διάταξη.
Για κάθε κόμβο η τιμή του κλειδιού του είναι μεγαλύτερη από όλες τις τιμές των κόμβων του αριστερού υποδένδρου του κόμβου και μικρότερη από όλες τις τιμές των κόμβων του δεξιού υποδένδρου του.

Κάθε κόμβος είναι ένα struct με πεδία key, data, LC, RC.

- Το μέγιστο ύψος δυαδικού δένδρου με n κόμβους είναι n-1. Γιατί;
- Το ελάχιστο ύψος δυαδικού δένδρου με n κόμβους είναι logn. Γιατί;

Ποια η σχέση ταξινομημένων δυαδικών δένδρων και ενδο-διατεταγμένης διάσχισης?

Ταξινομημένα Δένδρα

Αναδρομική Έκδοση

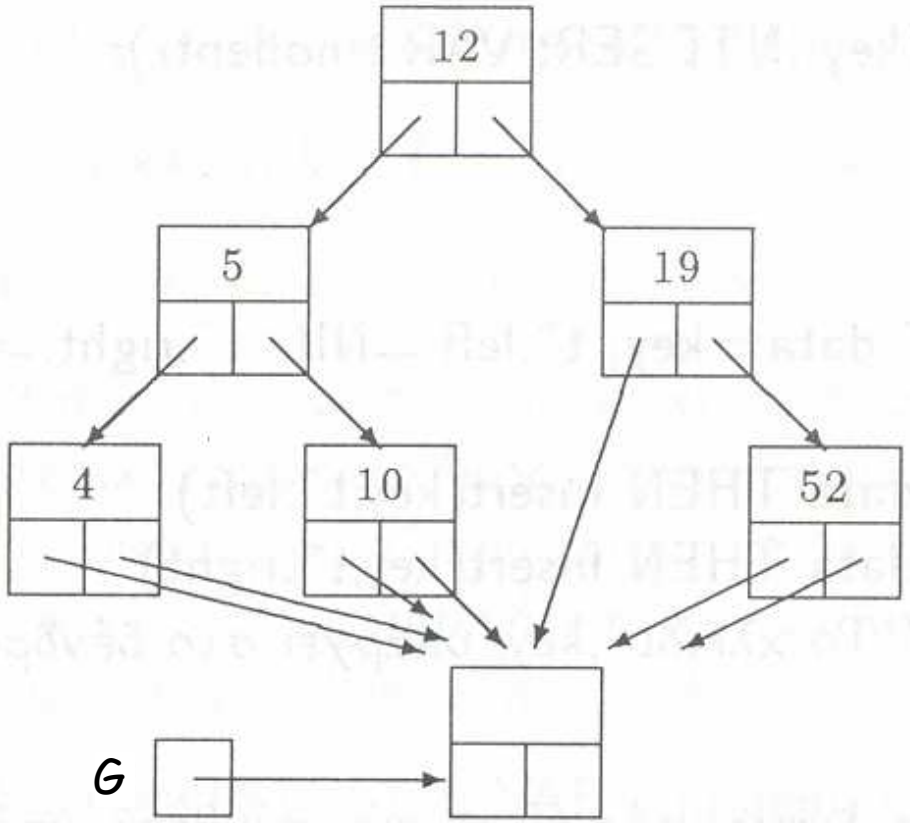
```
function BinarySearchTreeLookUp(
key K, pointer R):Type
/* Εύρεση του κλειδιού K στο δένδρο με ρίζα P
και επιστροφή του  πεδίου data ή nill αν το κλειδί
δεν υπάρχει στο δένδρο */
    if (R == NULL) return nill;
    else if (K == R->key) return R->data;
    else if (K < R->key)
        return(BinaryTreeLookUp(K, R->LC));
    else
        return(BinaryTreeLookUp(K, R->RC));
```

Χρονική πολυπλοκότητα;

Μη-Αναδρομική Έκδοση

```
function BinarySearchTreeLookUp (
key K, pointer R):Type
/* Εύρεση του κλειδιού K στο δένδρο με ρίζα P
και επιστροφή του  πεδίου data ή nill αν το
κλειδί δεν υπάρχει στο δένδρο */
    while (R != NULL && K != R->key) {
        if (K < R->key) R = R->LC;
        else R = R->RC;
    }
    if (R != NULL)
            return(R->data);
    else return nill;
}
```

Ταξινομημένα Δένδρα με Κόμβο Φρουρό



Ποια η χρησιμότητα του κόμβου φρουρού;

```
function BinarySearchTreeLookUp( key K,
pointer R):Type
    while (R != NULL && K != R->Key) {
        if (K < R->key) R = R->LC;
        else R = R->RC;
    }
    if (R != NULL) return(R->data);
    else return nill;
}
```

```
function BinarySearchTreeLookUp( key K,
pointer R):Type
    while (K != R->Key) {
        if (K < R->key) R = R->LC;
        else R = R->RC;
    }
    if (R != G) return(R->data);
    else return nill;
}
```

Ταξινομημένα Δένδρα - Ελάχιστο και Μέγιστο Στοιχείο

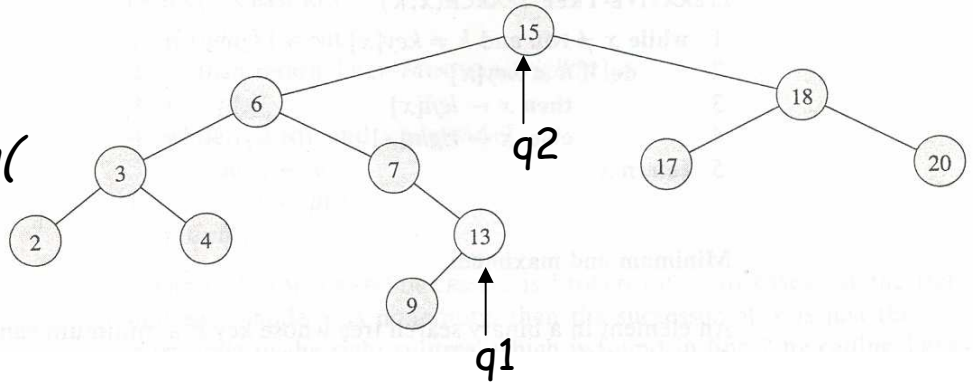
function *BinarySearchTreeMinimum(*
pointer R): **info**
/* ο R είναι δείκτης στη ρίζα του δένδρου */
if (R == NULL) return error;
while (R->LC != NULL) R = R->LC;
return(R->data);

function *BinarySearchTreeMaximum(*
pointer R): **info**
/* ο R είναι δείκτης στη ρίζα του δένδρου */
if (R == NULL) return error;
while (R->RC != NULL) R = R->RC;
return (R->data);

Πολυπλοκότητα;

Επόμενο και Προηγούμενο Στοιχείο

Το πρόβλημα είναι ίδιο με την εύρεση του επόμενου και προηγούμενου κόμβου στην ενδο-διατεταγμένη διάσχιση του δένδρου.



Πως θα βρούμε τον επόμενο του κόμβου στον οποίο δείχνει ο q1 στην ενδο-διατεταγμένη διάσχιση;

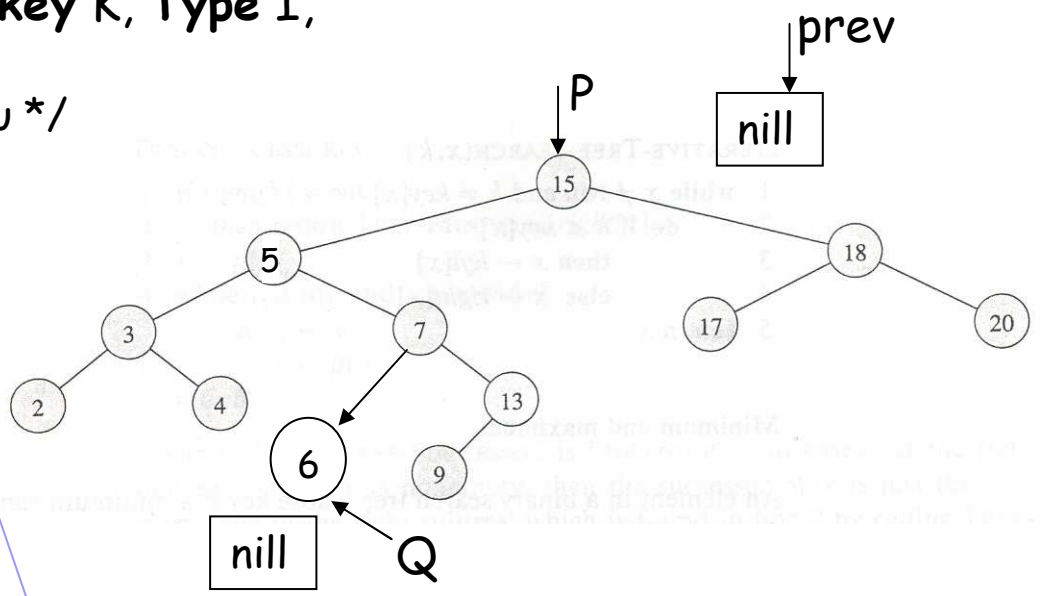
Πως θα βρούμε τον επόμενο του κόμβου στον οποίο δείχνει ο q2 στην ενδο-διατεταγμένη διάσχιση;

Ταξινομημένα Δένδρα - Εισαγωγή

```
function BinarySearchTreeInsert(key K, Type I,
pointer R): pointer
/* ο R είναι δείκτης στη ρίζα του δένδρου */
    pointer P, Q, Prev = NULL;

    P = R;
    while (P != NULL) {
        if (P->Key == K) {
                P->data = I;
                return R;
        }
        Prev = P;
        if (K < P->Key) P = P->LC;
        else P = P->RC;
    }

    /* Δημιουργία & προσθήκη νέου κόμβου */
    Q = NewCell(Node);
    Q->Key = K; Q->data = I; Q->LC = Q->RC = NULL;
    if (Prev == NULL) return Q;
    else if (K < Prev->Key) Prev->LC = Q;
    else Prev->RC = Q;
    return R;
```



Αναζήτηση κόμβου με κλειδί ίσο με K. Αν τέτοιος κόμβος δεν βρεθεί, ο δείκτης prev, μετά το πέρας της ανακύκλωσης, θα δείχνει στον κόμβο γονέα του προς εισαγωγή κόμβου!

Παράδειγμα
Εισαγωγή κόμβου με κλειδί 6

Ταξινομημένα Δένδρα - Διαγραφή

Όταν ένας κόμβος διαγράφεται, η ενδο-διατεταγμένη διάσχιση των υπόλοιπων κόμβων πρέπει να διασχίζει τους κόμβους σύμφωνα με τη διάταξη που είχαν πριν τη διαγραφή.

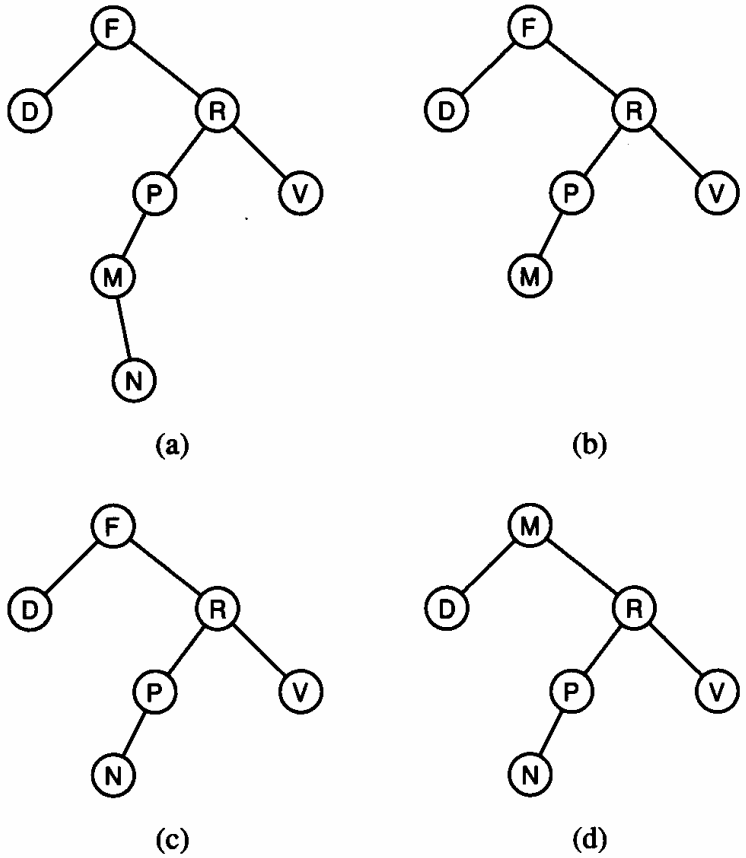
Περιπτώσεις

1) **Ο προς διαγραφή κόμβος είναι φύλλο.** Διαγράφουμε τον κόμβο χωρίς να εκτελέσουμε κάποια επιπρόσθετη ενέργεια.

2) **Ο προς διαγραφή κόμβος είναι εσωτερικός αλλά έχει μόνο ένα παιδί.** Αντικαθιστούμε τον κόμβο με το μοναδικό παιδί του.

3) **Ο προς διαγραφή κόμβος είναι εσωτερικός με δύο παιδιά.** Αντικαθιστούμε τον κόμβο με τον επόμενο ή τον προηγούμενό του στην ενδο-διατεταγμένη διάσχιση.

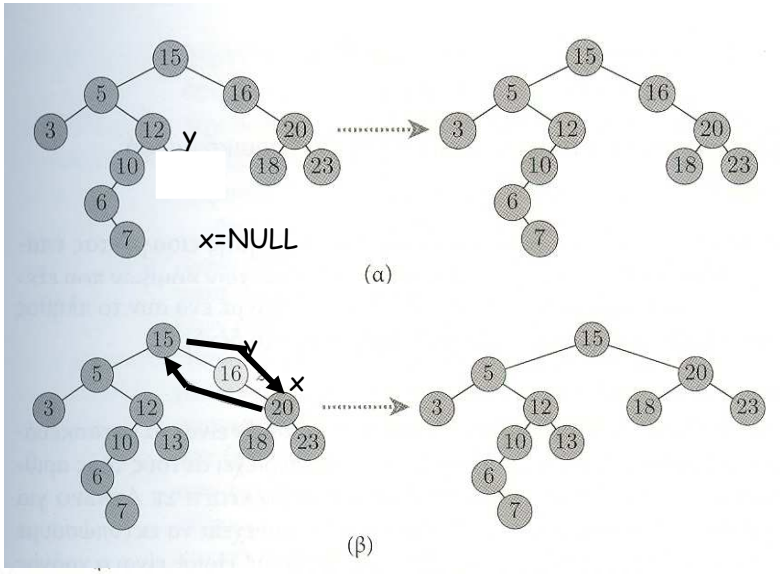
Αυτός είναι κόμβος με το πολύ ένα παιδί. Γιατί;



α) Αρχικό Δένδρο, β) Διαγραφή Ν από αρχικό δένδρο, γ) Διαγραφή Μ από αρχικό δένδρο, δ) Διαγραφή F από αρχικό δένδρο

Ταξινομημένα Δένδρα - Διαγραφή

Υποθέτουμε πως το δένδρο είναι διπλά συνδεδεμένο, δηλαδή κάθε κόμβος έχει ένα δείκτη p που δείχνει στο γονικό κόμβο.



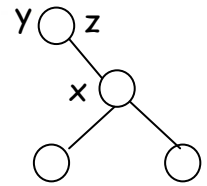

```
pointer BinaryTreeDelete(pointer R, Key K) {
/* Ο R είναι η διεύθυνση ενός δείκτη στη ρίζα του δένδρου */
/* Το K είναι το κλειδί του προς διαγραφή κόμβου */

   if (z->LC == NULL || z->RC == NULL)
       y = z;
   else y = TreeSuccessor(z);

   if (y->LC != NULL) x = y->LC;
   else x = y->RC;

   if (x != NULL) x->p = y->p;
   if (y->p == NULL) return x;
                                  // διαγραφή ρίζας
   else if (y == y->p->LC) y->p->LC = x;
   else y->p->RC = x;

   if (y != z) z->Key = y->Key;
   if y has other fields copy them two;
   return y;
}
```



Ταξινομημένα Δένδρα

Θεώρημα

Η χρονική πολυπλοκότητα των λειτουργιών Insert(), Delete() και LookUp() σε ταξινομημένα δυαδικά δένδρα είναι O(h), όπου h το ύψος του δένδρου.

Ποιο πρόβλημα μπορεί να προκύψει με συνεχή αντικατάσταση του κόμβου με τον επόμενό του στην ενδο-διατεταγμένη διάσχιση?

Στατικά Ταξινομημένα Δυαδικά Δένδρα

- Υπάρχουν κλειδιά που αναζητούνται πιο συχνά και άλλα που αναζητούνται πιο σπάνια.
- Σε μια λίστα συχνά αναζητούμενα κλειδιά κρατούνται όσο το δυνατόν πιο κοντά στην αρχή της λίστας.

Ποιο είναι το ανάλογο του ευριστικού "Μετακίνησης στην Αρχή» σε ένα ταξινομημένο δυαδικό δένδρο?

- Κάθε φορά που αναζητείται ένα κλειδί μεταφέρεται στη ρίζα.
- Ωστόσο, αυτό θα πρέπει να γίνει προσεκτικά ώστε να μην καταστρέφεται η ιδιότητα ταξινόμησης του δένδρου.

ΕΝΟΤΗΤΑ 6
ΥΛΟΠΟΙΗΣΗ ΛΕΞΙΚΩΝ ΜΕ ΙΣΟΖΥΓΙΣΜΕΝΑ ΔΕΝΔΡΑ

Ισοζυγισμένα Δένδρα

Χρονική Πολυπλοκότητα δομών που έχουν διδαχθεί μέχρι τώρα:

Στατική Μη-Ταξινομημένη Λίστα -> O(n), όπου n το πλήθος των κόμβων

Στατική Ταξινομημένη Λίστα -> O(n)

Δυναμική μη-ταξινομημένη λίστα -> O(n)

Δυναμική ταξινομημένη λίστα -> O(n)

Μη-ταξινομημένα Δένδρα -> O(n), όπου n το πλήθος των κόμβων

Γραμμική πολυπλοκότητα

Ταξινομημένα Δένδρα - > O(h), όπου h το ύψος του δένδρου

Τι τιμές μπορεί να πάρει το h;

Αναζητούνται δομές δεδομένων για την υλοποίηση δυναμικών λεξικών που να παρουσιάζουν καλύτερη χρονική πολυπλοκότητα από αυτή που παρέχουν οι παραπάνω δομές (π.χ., όλες οι λειτουργίες να έχουν χρονική πολυπλόκοτητα O(logn)).

Δένδρα AVL

Ένα ταξινομημένο δυαδικό δένδρο Τ ονομάζεται AVL (ή ισοζυγισμένο κατ' ύψος) αν και μόνο αν τα ύψη των δύο υποδένδρων κάθε κόμβου v διαφέρουν το πολύ κατά ένα.

Για κάθε κόμβο v ενός δυαδικού δένδρου Τ, ορίζουμε το αριστερό ύψος του v (LeftHeight(v)) ως εξής:

$$LeftHeight(v) = \begin{cases} 0 & \text{if v->LC == NULL} \\ 1 + Height(v->LC) & \text{otherwise} \end{cases}$$

Ομοίως, ορίζουμε το δεξιό ύψος του v (RightHeight(v)) ως εξής:

$$RightHeight(v) = \begin{cases} 0 & \text{if v->RC == NULL} \\ 1 + Height(v->RC) & \text{otherwise} \end{cases}$$

Το LeftHeight(T) (RightHeight(T)) του δένδρου ισούται με το LeftHeight(R) (RightHeight(R), αντίστοιχα), όπου R είναι ο κόμβος ρίζα του Τ.

Δένδρα AVL

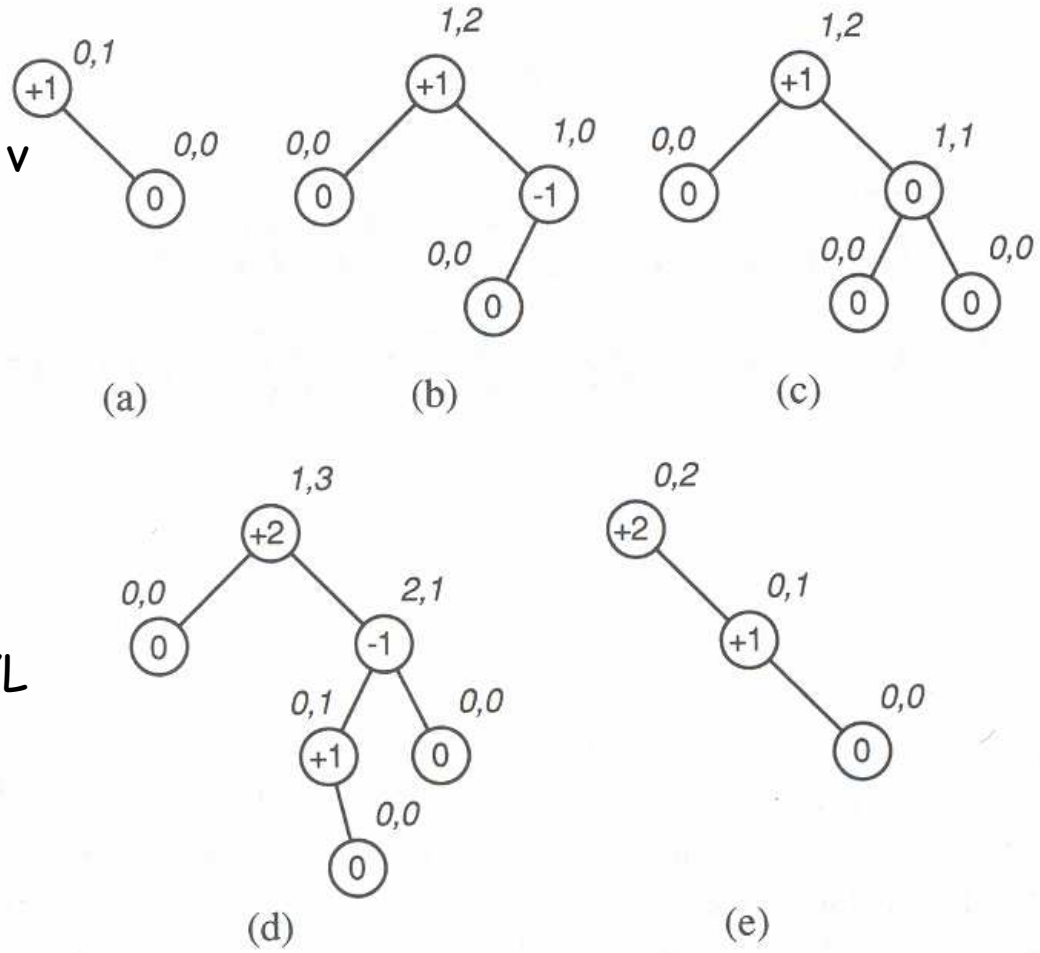
Το balance (ισοζύγιο) του κόμβου v ορίζεται ως εξής:
balance(v) = RightHeight(v) – LeftHeight(v).

Σε ένα δένδρο AVL, κάθε κόμβος πρέπει να έχει balance 0, +1 ή -1.

Πρόταση 1
Κάθε υποδένδρο ενός δένδρου AVL είναι επίσης δένδρο AVL.

Θεώρημα
Το ύψος h ενός δένδρου AVL, n στοιχείων, είναι O(logn).

Μπορούμε να συμπεράνουμε κάτι για την πολυπλοκότητα της AVLTreeLookUp()?



Τα δένδρα (a), (b) και (c) είναι δένδρα AVL, ενώ τα (d) και (e) δεν είναι.

Δένδρα AVL – Απόδειξη Θεωρήματος

- Αναζητούμε ένα πάνω φράγμα στο μήκος h του μακρύτερο μονοπατιού οποιουδήποτε δένδρου AVL με n κόμβους.
- Έστω ένας οποιοσδήποτε ακέραιος h. Θα εστιάσουμε στο ερώτημα «Ποιος είναι ο μικρότερος ακέραιος n τ.ω. να υπάρχει ένα δένδρο AVL ύψους h με n κόμβους;»
- W_h: σύνολο όλων των δένδρων AVL ύψους h με ελάχιστο πλήθος κόμβων.
- w_h: πλήθος κόμβων σε οποιοδήποτε από αυτά τα δένδρα.
- Προφανώς $w_0 = 1$ και $w_1 = 2$.
- Έστω ότι T είναι οποιοδήποτε δένδρο AVL στο W_h και έστω T_L και T_R το αριστερό και το δεξιό υποδένδρο του T, αντίστοιχα.
- Αφού το T έχει ύψος h, είτε το T_L ή το T_R έχει ύψος h-1 (ας υποθέσουμε χωρίς βλάβη της γενικότητας ότι το T_R έχει ύψος h-1).
- Το T_R είναι δένδρο AVL ύψους h-1 και μάλιστα θα πρέπει να έχει το μικρότερο πλήθος κόμβων μεταξύ των δένδρων AVL ύψους h-1 (αφού στην αντίθετη περίπτωση, θα μπορούσε να αντικατασταθεί από κάποιο δένδρο AVL ύψους h-1 με λιγότερους κόμβους και να οδηγηθούμε έτσι σε ένα δένδρο AVL ύψους h με λιγότερους κόμβους από ότι το T, το οποίο αντιτίθεται στον ορισμό του T).

Δένδρα AVL – Απόδειξη Θεωρήματος

- Άρα, $T_R \in W_{h-1}$.
- Αφού το T είναι δένδρο AVL, το T_L έχει ύψος είτε h-1 ή h-2. Εφόσον το T είναι δένδρο με ελάχιστο πλήθος κόμβων μεταξύ εκείνων με ύψος h, θα πρέπει να ισχύει $T_L \in W_{h-2}$.
- Επομένως:

$$w_h = 1 + w_{h-2} + w_{h-1}$$

Μπορεί να αποδειχθεί (επαγωγικά) ότι $w_h = F_{h+3} - 1$, όπου F_i είναι ο i-οστός όρος της ακολουθίας Fibonacci, για τον οποίο ισχύει ότι $F_i > \varphi^i/\sqrt{5} - 1$, όπου $\varphi = (1+\sqrt{5})/2$.

Επομένως, $n \geq w_h > \varphi^{h+3}/\sqrt{5} - 2$.

Από την παραπάνω ανίσωση προκύπτει ότι $h = O(\log n)$.

Δένδρα AVL – Εισαγωγή

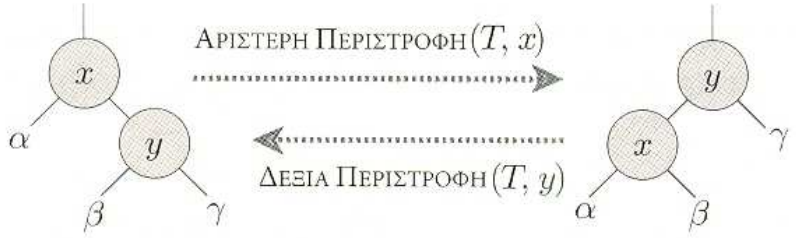
Πως μπορούμε να υλοποιήσουμε την Insert()? Που διαφέρει από την Insert() σε δυαδικό δένδρο αναζήτησης?

1η Προσπάθεια Σχεδίασης Αλγορίθμου Εισαγωγής

- Ακολουθώντας το γνωστό αλγόριθμο εισαγωγής σε ταξινομημένο δένδρο (δένδρο δυαδικής αναζήτησης), βρίσκουμε το μονοπάτι από τη ρίζα προς τον κατάλληλο κόμβο (έστω v) στον οποίο θα γίνει η εισαγωγή. Το μονοπάτι αυτό αποθηκεύεται σε στοίβα.
- Ακολουθώντας το μονοπάτι από τον v προς τη ρίζα (εκτελώντας επαναληπτικά pop() στη στοίβα), υπολογίζουμε τα νέα balances για κάθε κόμβο του μονοπατιού αυτού.
- Αν το balance όλων των κόμβων του μονοπατιού εξακολουθεί να είναι 0, +1 ή -1, η διαδικασία εισαγωγής τερματίζει.
- Αν, ωστόσο, το balance κάποιου κόμβου αλλάζει σε +2 ή σε –2, θα πρέπει να ακολουθηθεί διαδικασία προσαρμογής του balance στο κατάλληλο εύρος. Η διαδικασία αυτή θα έχει ως αποτέλεσμα το δένδρο να εξακολουθεί να είναι ταξινομημένο δυαδικό δένδρο με τους ίδιους κόμβους και κλειδιά, αλλά με όλους τους κόμβους να έχουν balance 0, 1, ή -1.

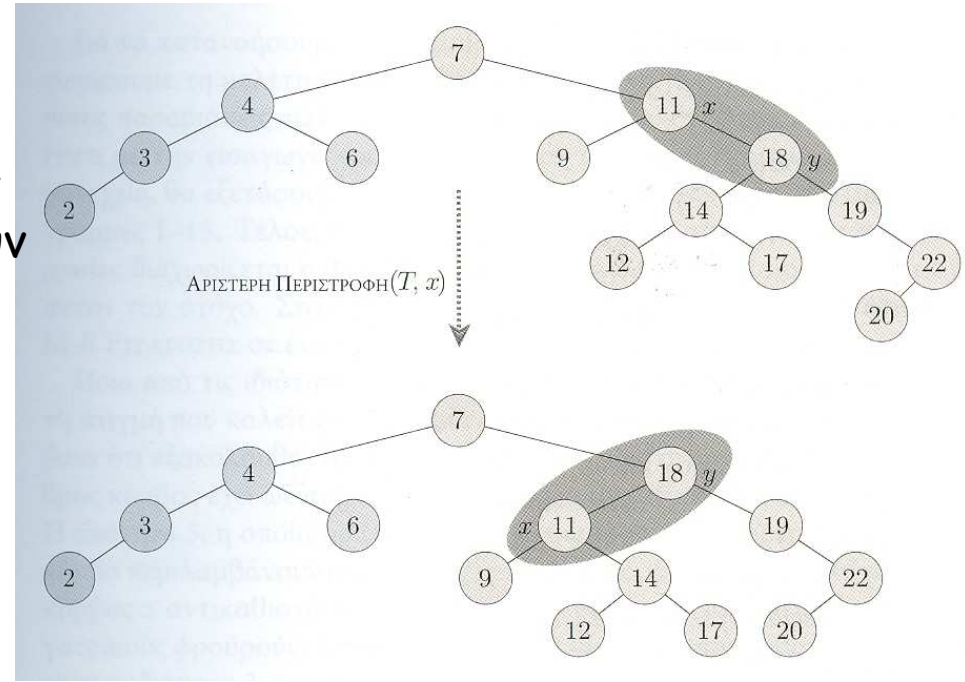
Δένδρα AVL – Περιστροφές

Μια περιστροφή είναι μια τοπική λειτουργία σε ένα ταξινομημένο δένδρο η οποία διατηρεί την ιδιότητα της ταξινόμησης.



Αριστερή Περιστροφή γύρω από έναν κόμβο x (ο οποίος έχει δεξιό θυγατρικό κόμβο y != nill)

- Η αριστερή περιστροφή στρέφει τους x και y κατά αντίστροφη φορά αυτής των δεικτών του ρολογιού.
- Καθιστά τον y νέα ρίζα του υποδένδρου, μετατρέποντας τον x σε αριστερό παιδί του y και το αριστερό παιδί του y σε δεξιό παιδί του x.

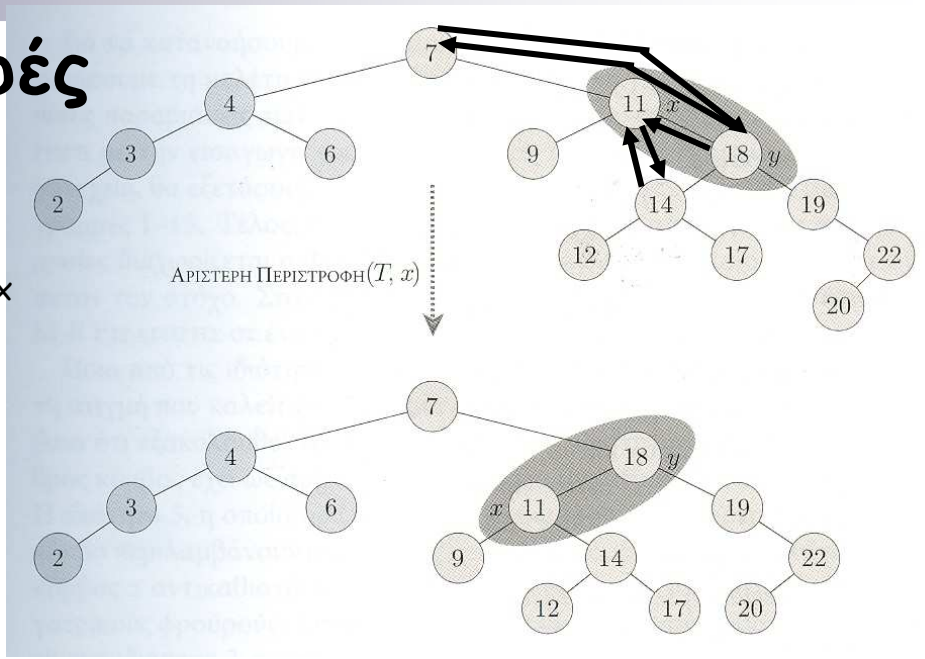


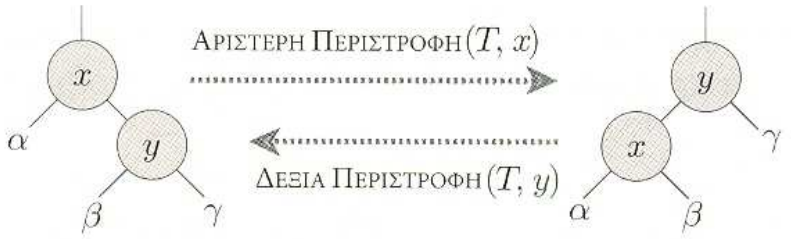
Δένδρα AVL – Περιστροφές

```
Procedure LeftRotate(pointer R, pointer x) {
    y = x->RC;              // ορισμός του y

    // μετατροπή αριστερού υποδένδρου y σε δεξιό υποδένδρο x
    x->RC = y->LC;
    if (y->LC != NULL) y->LC->p = x;

/* νέος πατρικός κόμβος του y γίνεται ο πρώην
πατρικός κόμβος του x και αν δεν υπάρχει
τέτοιος κόμβος, νέα ρίζα του δένδρου γίνεται το y */
    y->p = x->p;
    if (x->p == NULL) return y;
    else if (x == x->p->LC) // αν ο x είναι αριστερό παιδί του πατέρα του
            x->p->LC = y;
    else x->p->RC = y;
    y->LC = x;    // αριστερό παιδί του y γίνεται το x
    x->p = y;  // πατέρας του x γίνεται το y
}
```





Μια δεξιά περιστροφή γίνεται με συμμετρικό τρόπο και ο κώδικας της είναι απολύτως συμμετρικός με τον παραπάνω.

Χρονική Πολυπλοκότητα μιας περιστροφής; $O(1)$

Δένδρα AVL – Εισαγωγή

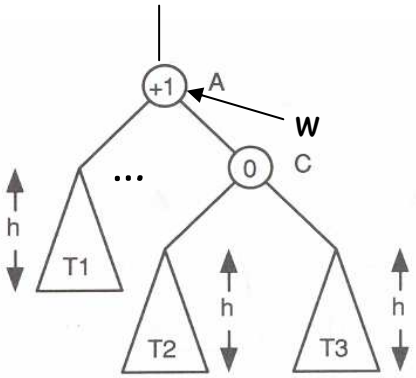
Ο πρώτος κόμβος στο μονοπάτι από τον κόμβο v (που εισήχθη στο δένδρο) προς τη ρίζα, του οποίου το balance ήταν +1 ή -1 (πριν την εισαγωγή) ονομάζεται κρίσιμος κόμβος.

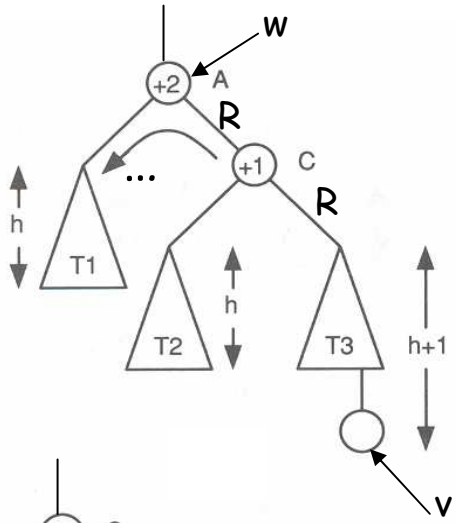
Αν ο κόμβος αυτός αποκτά balance +2 ή -2 μετά την εισαγωγή, τότε είναι ο πρώτος κόμβος στο μονοπάτι από τον v στη ρίζα για τον οποίο θα πρέπει να γίνουν κατάλληλες ενέργειες ώστε να διορθωθεί το balance του.

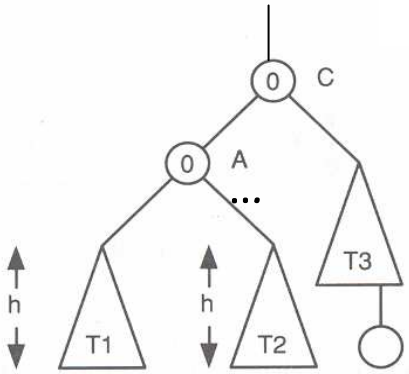
Διακρίνουμε περιπτώσεις ανάλογα με το είδος των δύο πρώτων ακμών του μονοπατιού από τον κρίσιμο κόμβο w προς τον εισαχθέντα κόμβο v.

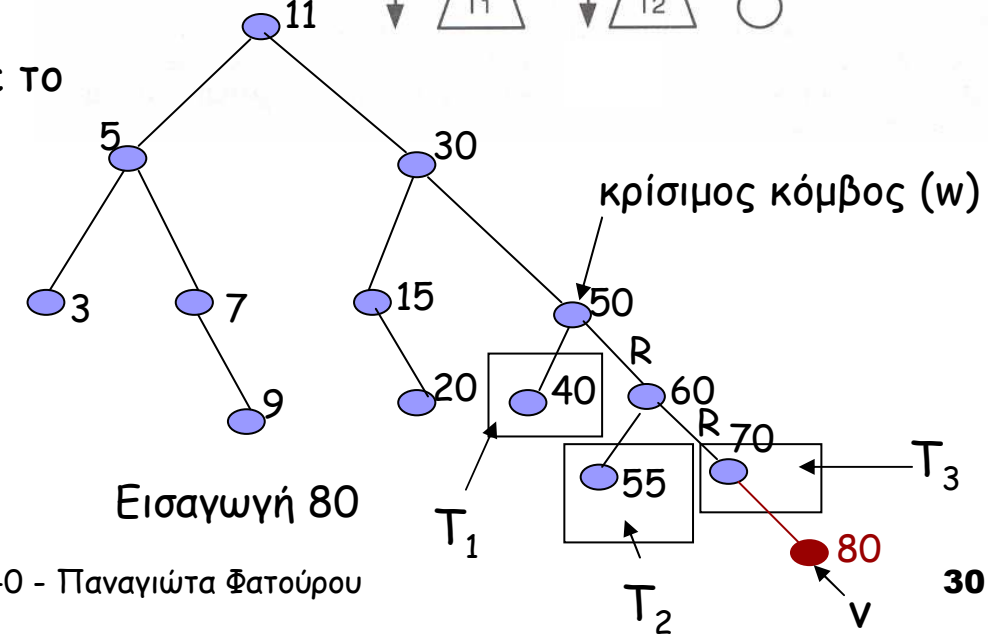
Περίπτωση RR (Right-Right)
Και οι δύο ακμές οδηγούν δεξιά.

Εκτελούμε μια αριστερή περιστροφή γύρω από τον κρίσιμο κόμβο.



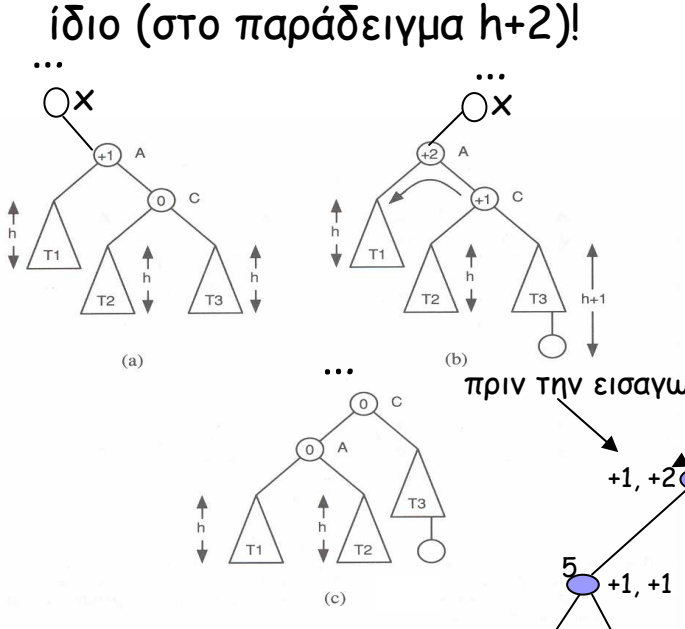


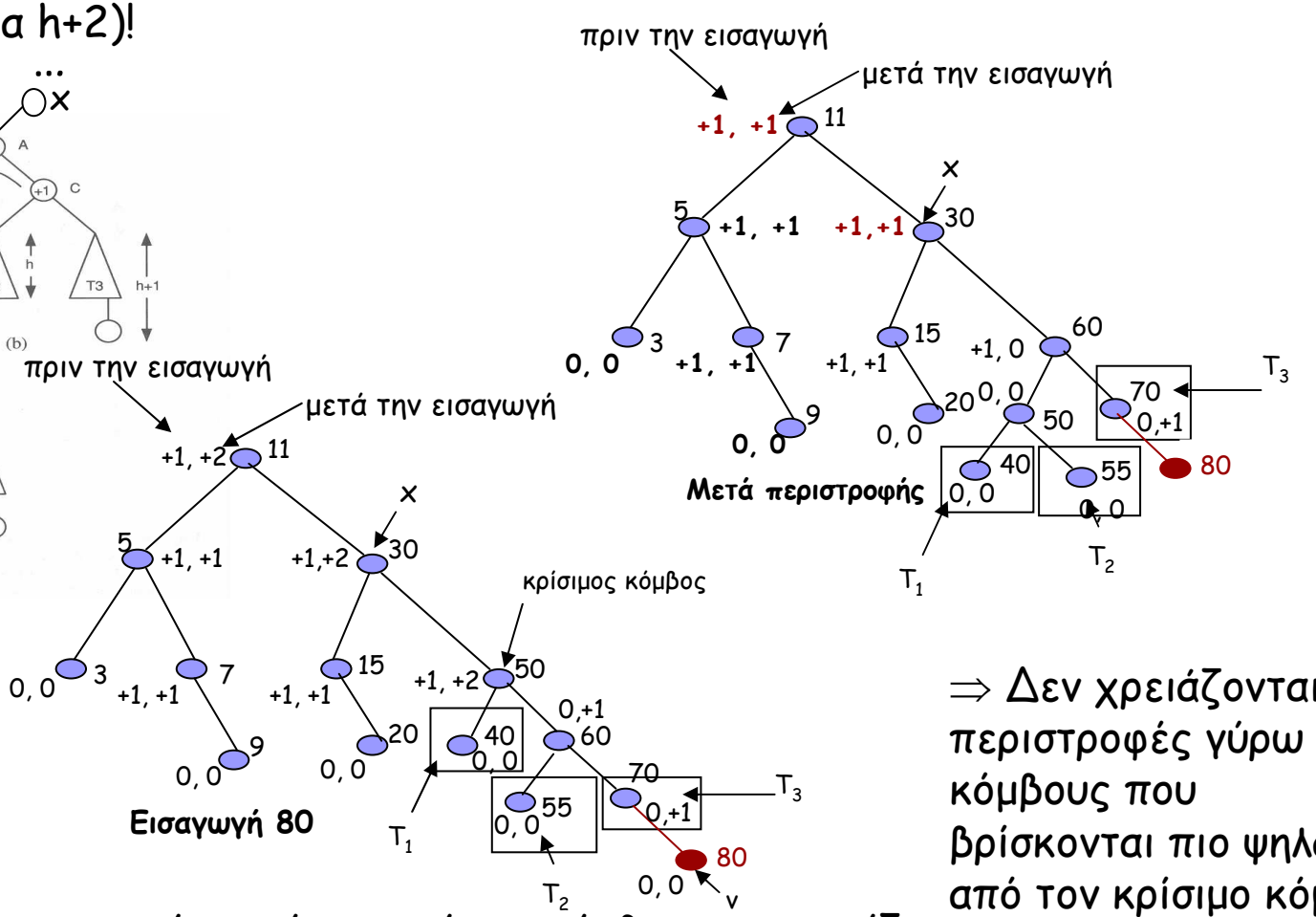




Δένδρα AVL – Εισαγωγή – Περίπτωση RR

Παρατήρηση: Έστω x ο πατρικός κόμβος του w και έστω ότι ο w είναι δεξιό παιδί του x. Το ύψος του δεξιού παιδιού του x πριν την εισαγωγή και μετά την περιστροφή είναι το ίδιο (στο παράδειγμα h+2)!





⇒ Μία περιστροφή γύρω από τον κρίσιμο κόμβο αρκεί!!!!

⇒ Δεν χρειάζονται περιστροφές γύρω από κόμβους που βρίσκονται πιο ψηλά από τον κρίσιμο κόμβο στο μονοπάτι από τον v προς τη ρίζα!!!!!

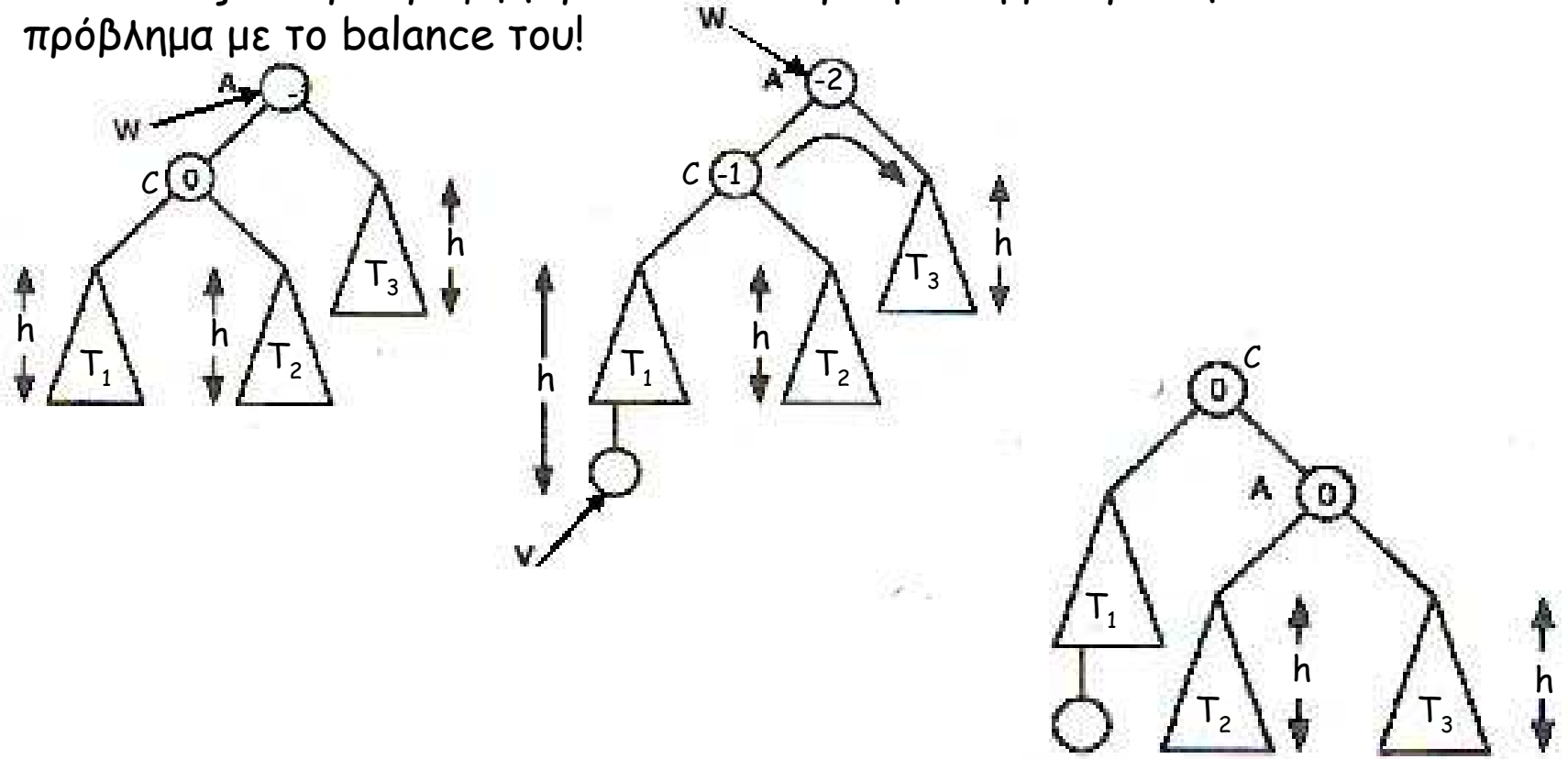
- Όλοι οι κόμβοι στο μονοπάτι από τον κρίσιμο κόμβο προς τη ρίζα έχουν το ίδιο balance μετά την περιστροφή με αυτό που είχαν πριν την εισαγωγή του v στο δένδρο!

Δένδρα AVL – Εισαγωγή – Περίπτωση LL

Περίπτωση LL

Και οι δύο ακμές οδηγούν αριστερά.

- Είναι συμμετρική της περίπτωσης RR!
- Μία δεξιά περιστροφή γύρω από τον κρίσιμο κόμβο αρκεί για να επιλυθεί το πρόβλημα με το balance του!

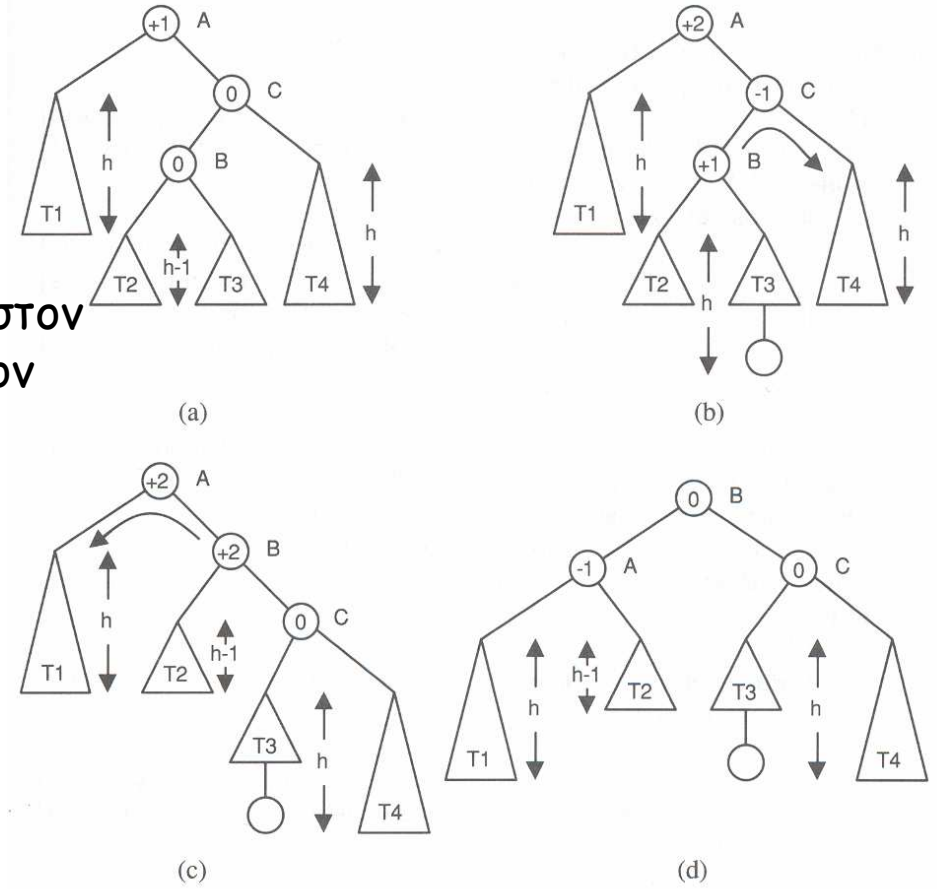


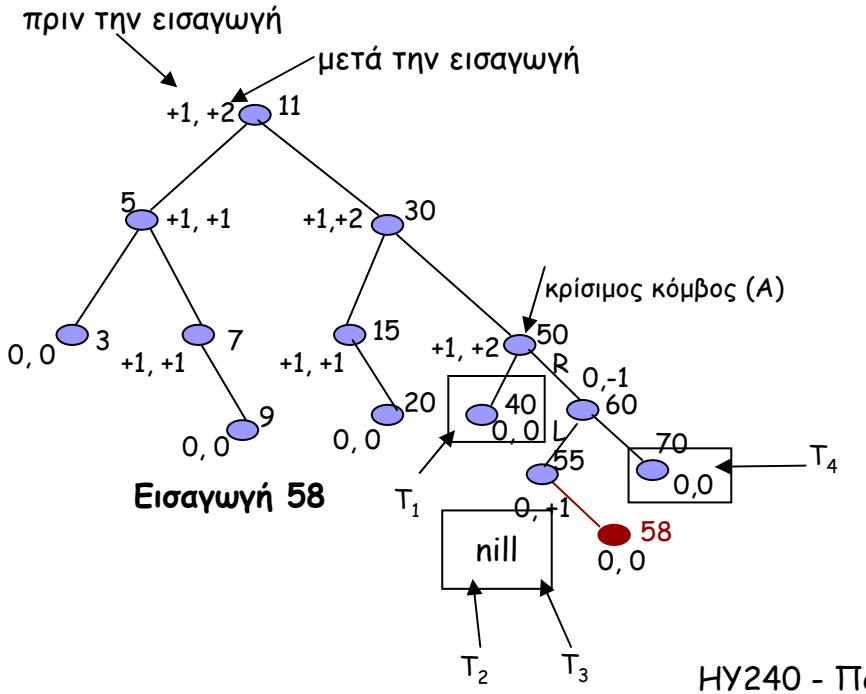
Δένδρα AVL – Εισαγωγή – Περίπτωση RL

Περίπτωση RL

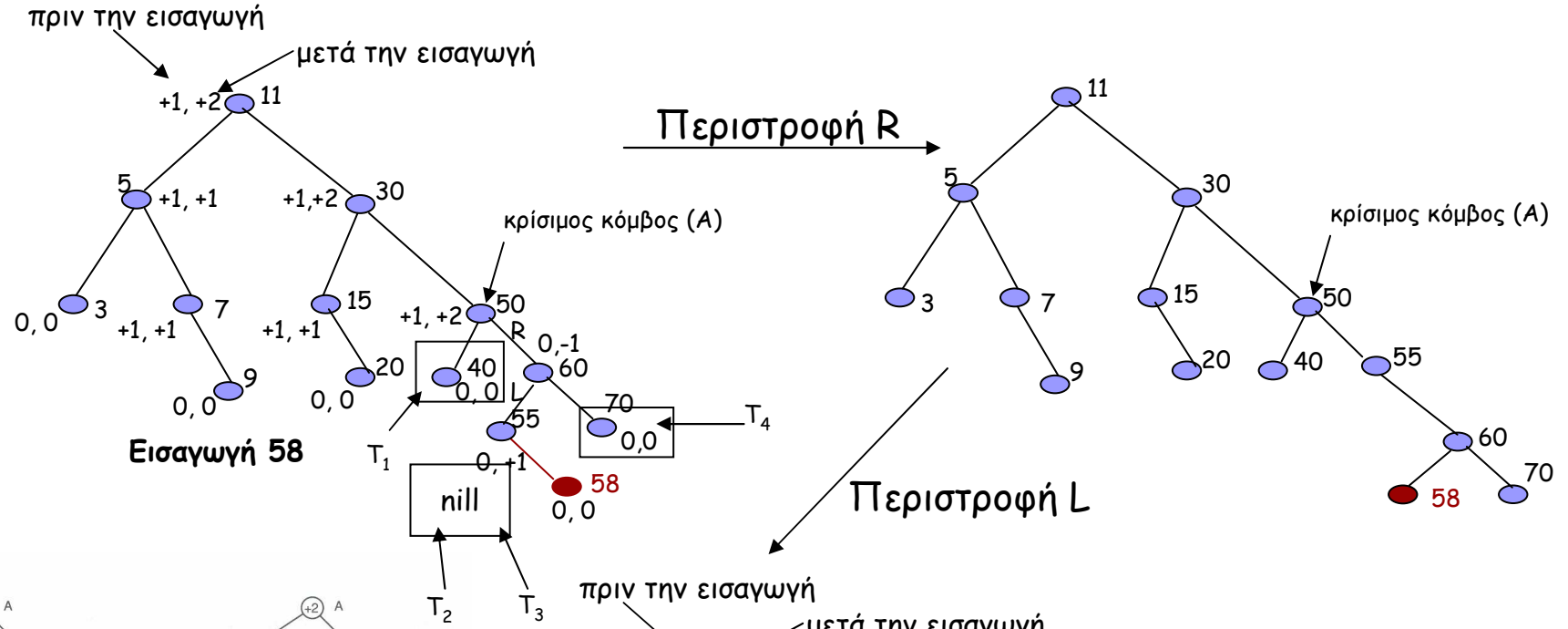
Η πρώτη ακμή οδηγεί δεξιά και η δεύτερη αριστερά.

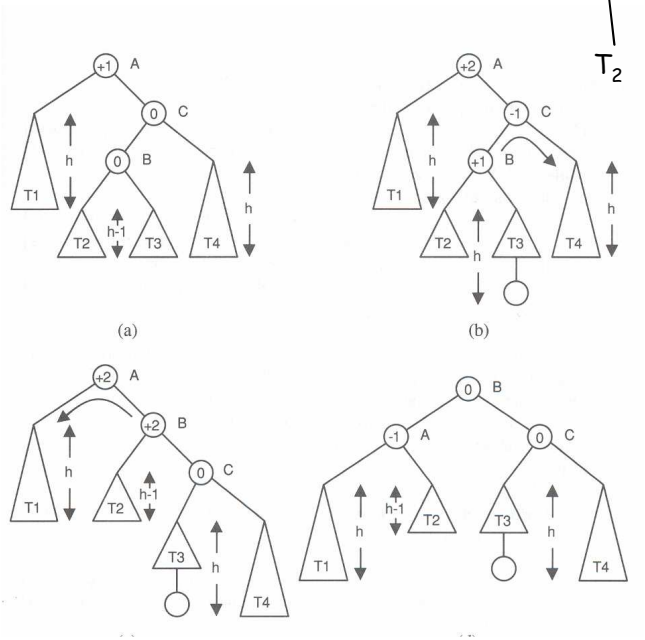
- Απαιτούνται δύο περιστροφές, μια δεξιά περιστροφή γύρω από τον επόμενο του κρίσιμου κόμβου στο μονοπάτι που οδηγεί στον ν και μια αριστερή περιστροφή γύρω από τον κρίσιμο κόμβο.

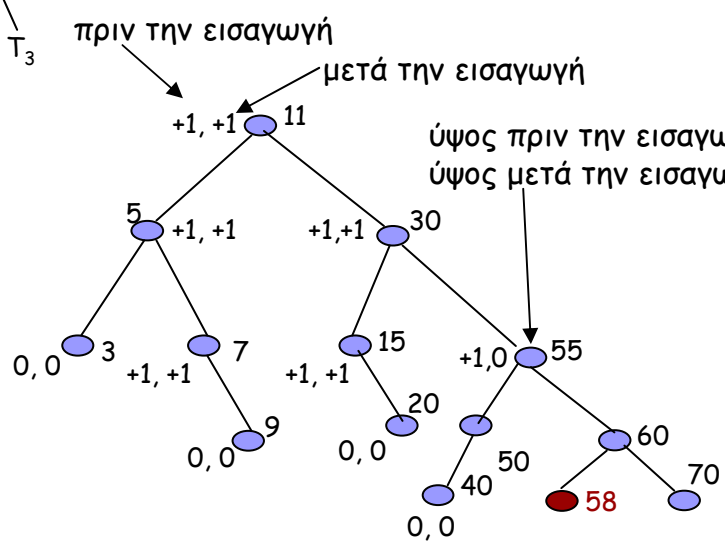




Δένδρα AVL – Εισαγωγή – Περίπτωση RL







Ίδιες περιστροφές θα πραγματοποιούνταν αν είχαμε εισαγωγή του 53 στο αρχικό δένδρο (το οποίο θα ήταν αριστερό παιδί του 55).

Δένδρα AVL – Εισαγωγή – Περίπτωση LR

Περίπτωση LR

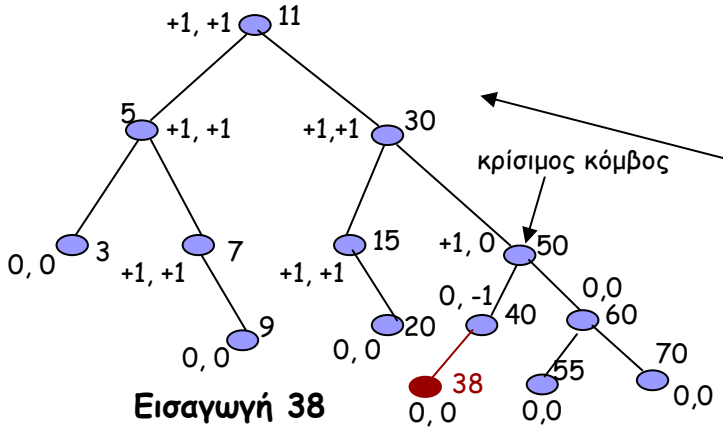
Η πρώτη ακμή οδηγεί αριστερά και η δεύτερη δεξιά.

- Είναι συμμετρική της περιπτώσεως RL.
- Απαιτούνται δύο περιστροφές, μια αριστερή περιστροφή γύρω από τον επόμενο του κρίσιμου κόμβου στο μονοπάτι που οδηγεί στον v και μια δεξιά περιστροφή γύρω από τον κρίσιμο κόμβο.

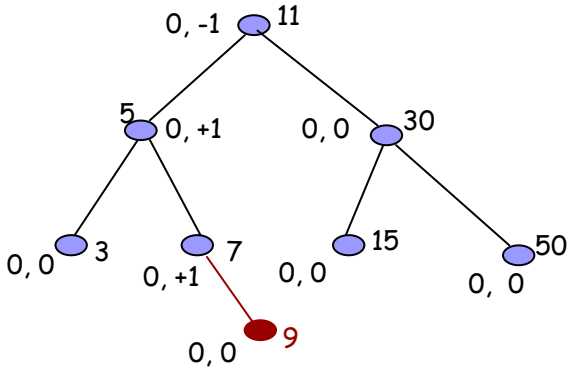
Άσκηση για το σπίτι

Φτιάξτε τα αντίστοιχα συμμετρικά σχήματα εκείνων που παρουσιάστηκαν στις δύο τελευταίες διαφάνειες και μελετήστε τις ενέργειες που πρέπει να πραγματοποιηθούν στην περίπτωση LR.

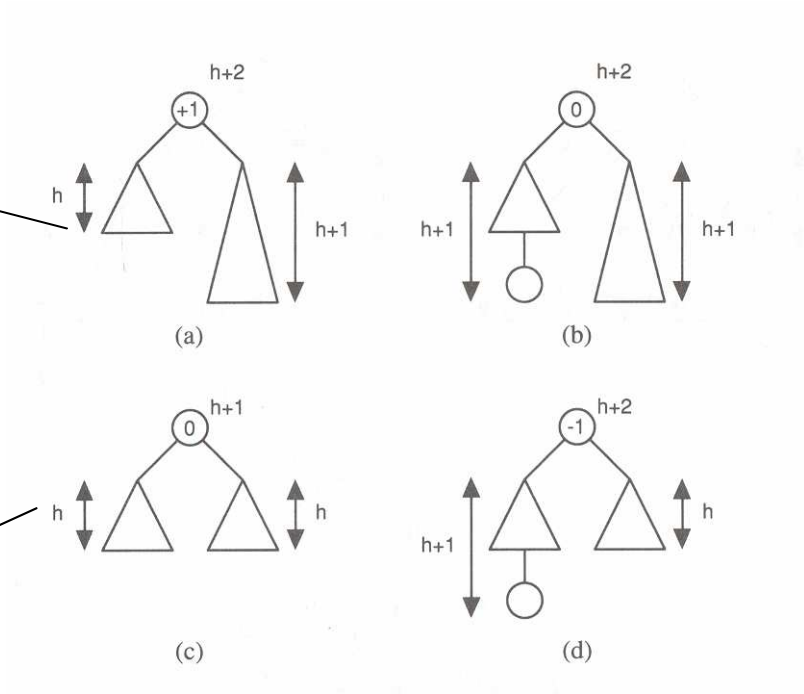
Δένδρα AVL – Εισαγωγή – Περιπτώσεις που δεν απαιτούν περιστροφές



Εισαγωγή 38



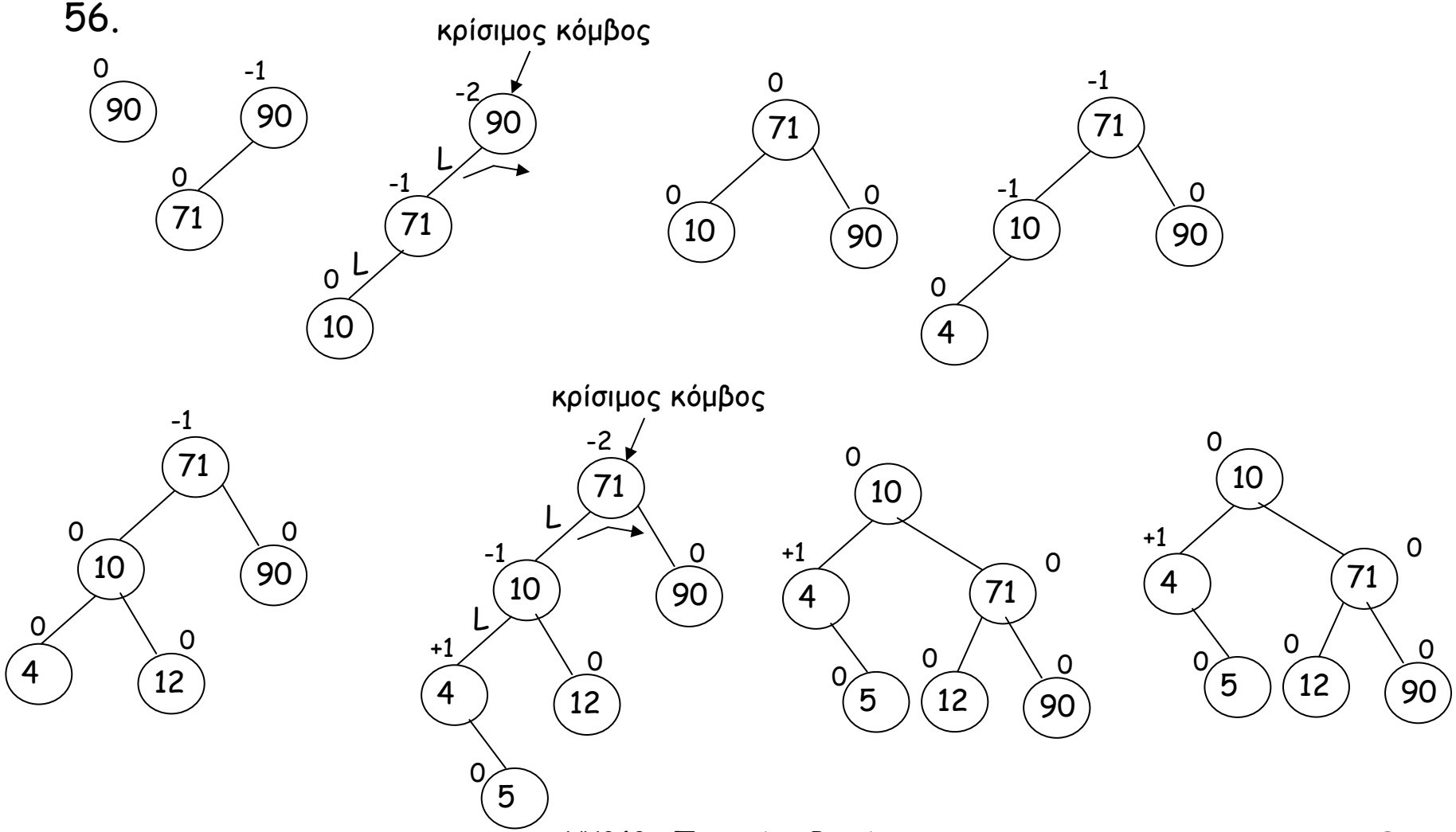
Εισαγωγή 9



Εισαγωγές κόμβων σε AVL δένδρο

Παράδειγμα

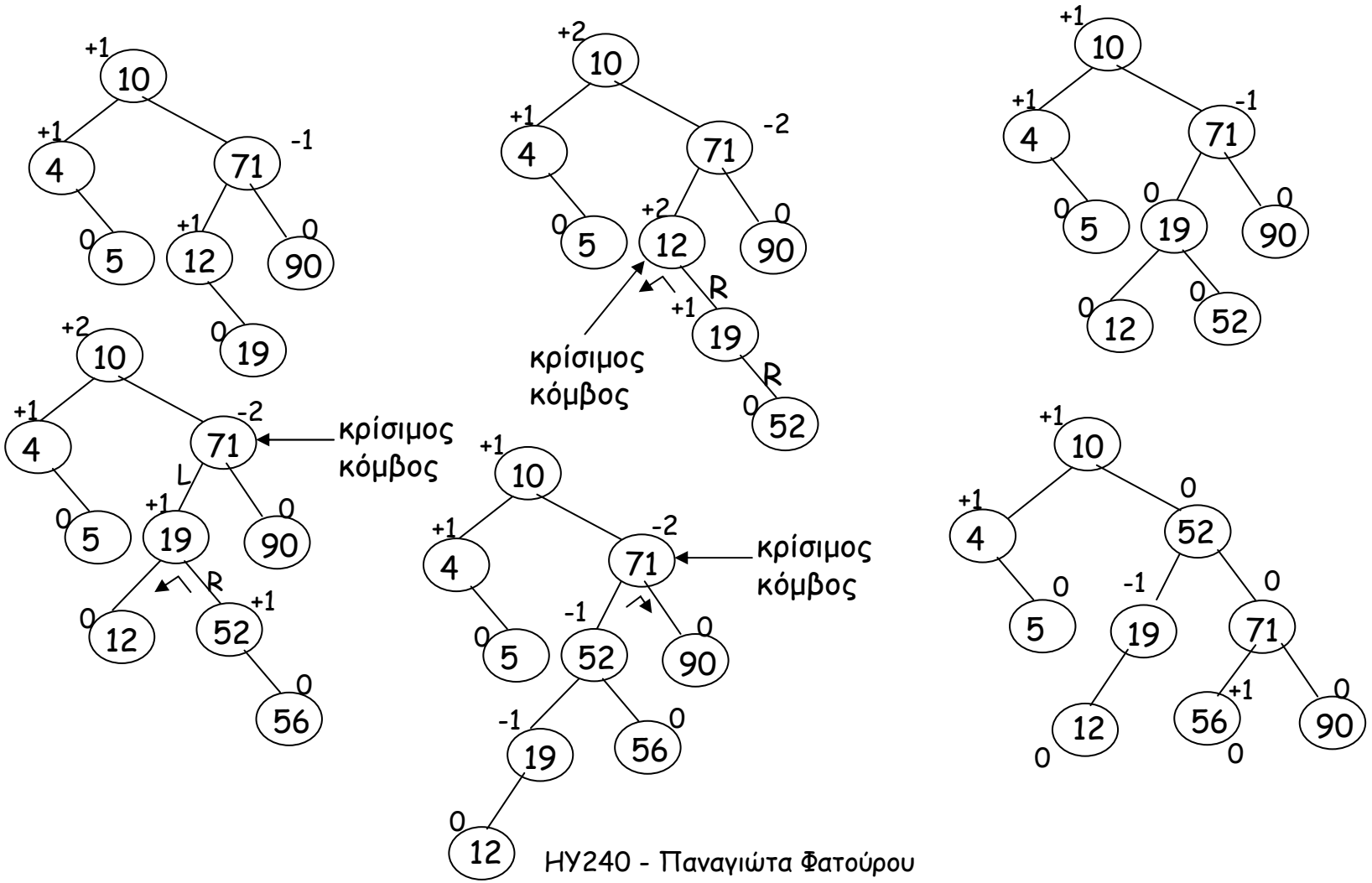
Διαδοχικές εισαγωγές των κλειδιών με τιμές 90, 71, 10, 4, 12, 5, 19, 52 και 56.



Εισαγωγές κόμβων σε AVL δένδρο

Παράδειγμα

Διαδοχικές εισαγωγές των κλειδιών με τιμές 90, 71, 10, 4, 12, 5, 19, 52 και 56.



Δένδρα AVL – Εισαγωγή - Παρατηρήσεις

- Κάθε κόμβος στο μονοπάτι από τον εισαχθέντα κόμβο v μέχρι τον κρίσιμο κόμβο w είχε balance 0 πριν την εισαγωγή. Άρα, θα έχει balance +1 ή –1 μετά την εισαγωγή.

Τι καθορίζει το αν κάποιος τέτοιος κόμβος θα έχει balance +1 ή –1;
Το αν η πρώτη ακμή στο μονοπάτι από τον κόμβο αυτό προς τον εισαχθέντα κόμβο οδηγεί αριστερά ή δεξιά.

- Το balance του κρίσιμου κόμβου w γίνεται είτε 0 ή +2 (ή –2, αντίστοιχα) μετά την τοποθέτηση του νέου κόμβου στην κατάλληλη θέση στο δένδρο.

Γιατί; Πότε το balance γίνεται 0 και πότε +2 (ή –2);
Αν το balance ήταν +1 (-1) και η πρώτη ακμή στο μονοπάτι από τον w στον v οδηγεί δεξιά (αριστερά, αντίστοιχα), ο w θα έχει balance +2 (-2, αντίστοιχα) μετά την τοποθέτηση του v, ενώ αν το μονοπάτι οδηγεί αριστερά (δεξιά, αντίστοιχα), τότε ο w θα έχει balance 0 μετά την τοποθέτηση του v.

Παρατήρηση
Οι περιστροφές γίνονται γύρω από το πολύ δύο κόμβους στο δένδρο ανάλογα με την περίπτωση που πρέπει να εφαρμοστεί (μία περιστροφή γύρω από τον κρίσιμο κόμβο για τις περιπτώσεις LL, RR και δύο περιστροφές, η πρώτη γύρω από τον επόμενο του κρίσιμου κόμβου στο μονοπάτι που ακολουθήθηκε κατά την τοποθέτηση του εισαχθέντα κόμβου και η δεύτερη γύρω από τον κρίσιμο κόμβο για τις περιπτώσεις RL, LR).

Δένδρα AVL – Εισαγωγή - Παρατηρήσεις

Καθώς η αναζήτηση του πατρικού κόμβου του v εξελίσσεται, αρκεί να αποθηκεύεται ο τελευταίος κόμβος με balance +1 ή -1 στο μονοπάτι που ακολουθείται.

Μετά την τοποθέτηση του v στο δένδρο (και πριν τις περιστροφές), ξεκινώντας από τον κρίσιμο κόμβο w μπορεί να ακολουθηθεί και πάλι το ίδιο μονοπάτι προς τον εισερχόμενο κόμβο ώστε να διορθωθούν τα balances.

Τέλος, αν χρειάζεται πραγματοποιούνται περιστροφές.

Γιατί μπορούμε να βρούμε και πάλι αυτό το μονοπάτι?

Ακολουθούμε την ίδια διαδικασία αναζήτησης (ξεκινώντας από τον w), δηλαδή αν το κλειδί του προς εισαγωγή κόμβου είναι μικρότερο από εκείνο του τρέχοντος ο επόμενος κόμβος στο μονοπάτι είναι στα αριστερά του τρέχοντος, ενώ διαφορετικά στα δεξιά.

Πως διορθώνουμε τα balances?

Έστω ένας οποιοσδήποτε κόμβος x στο μονοπάτι από τον w στον v. Ο κόμβος αυτός είχε balance 0. Αν η πρώτη ακμή στο μονοπάτι από τον x στον v οδηγεί αριστερά του x, τότε το balance του x γίνεται -1, διαφορετικά η νέα τιμή του balance θα είναι +1.

Ποια είναι η πολυπλοκότητα της AVLInsert()?

O(ύψος δένδρου). Θεώρημα $\Rightarrow$ ύψος δένδρου AVL = $O(\log n)$

$\Rightarrow$ Χρονική Πολυπλοκότητα = $O(\log n)$

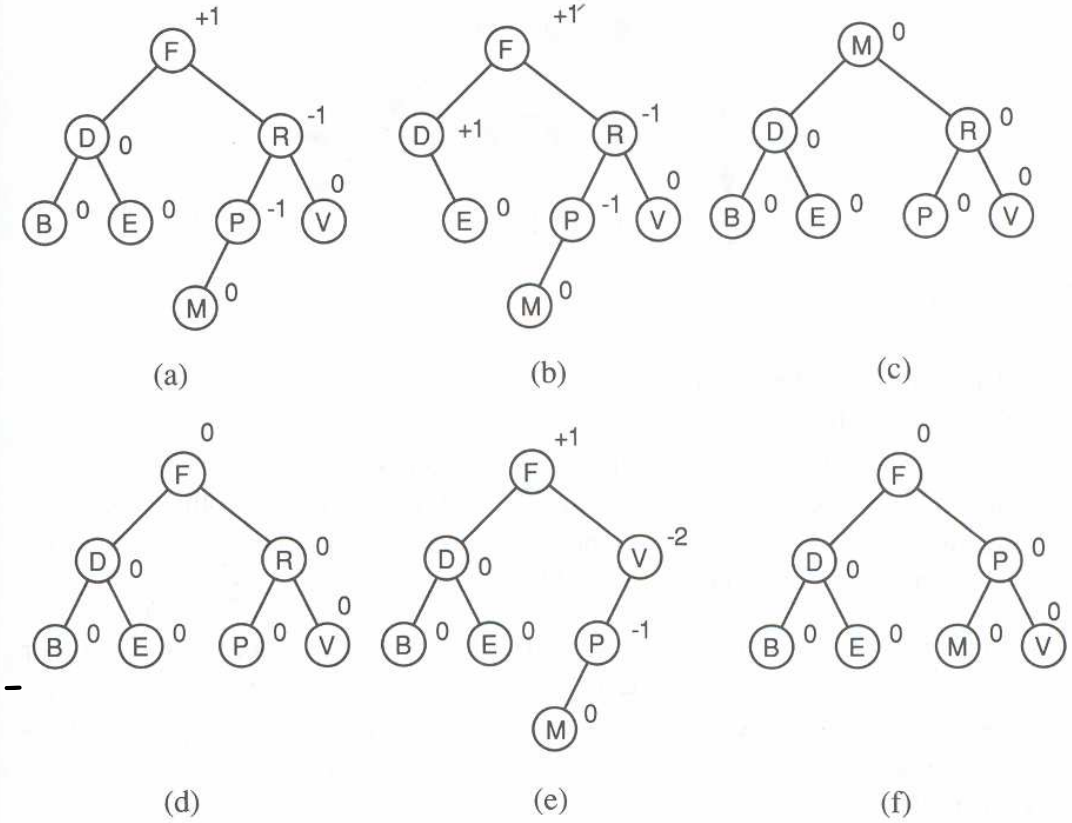
Δένδρα AVL – Διαγραφή

Αρχικά ακολουθούμε το γνωστό αλγόριθμο διαγραφής σε δυαδικό δένδρο:

1) Διαγραφή του ίδιου του κόμβου v αν είναι φύλλο.

2) Αντικατάστασή του από το παιδί του αν έχει μόνο ένα παιδί.

3) Αντικατάστασή του από τον επόμενό του στην ενδο-διατεταγμένη διάταξη αν έχει δύο παιδιά.



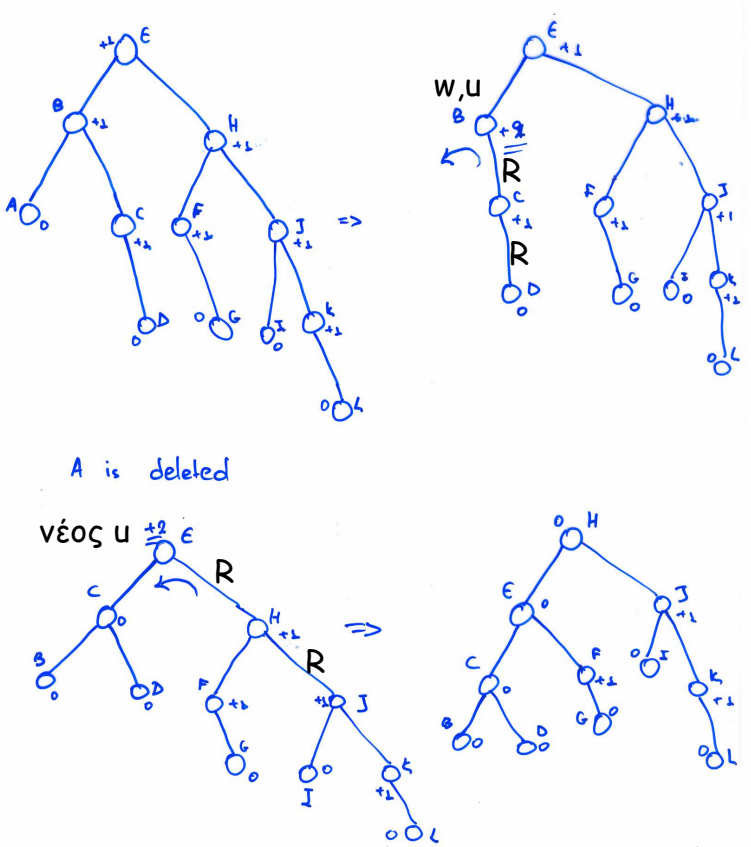
(a) Αρχικό δένδρο, (b) Διαγραφή του B, (c) Διαγραφή του F, (d) Διαγραφή του M, (e) Διαγραφή του R

Balance

Αν 1) ή 2), το balance του γονικού κόμβου w του v αλλάζει.

Αν 3), το balance του γονικού κόμβου w του επομένου του v στην ενδο-διατεταγμένη διάσχιση αλλάζει.

Δένδρα AVL – Διαγραφή

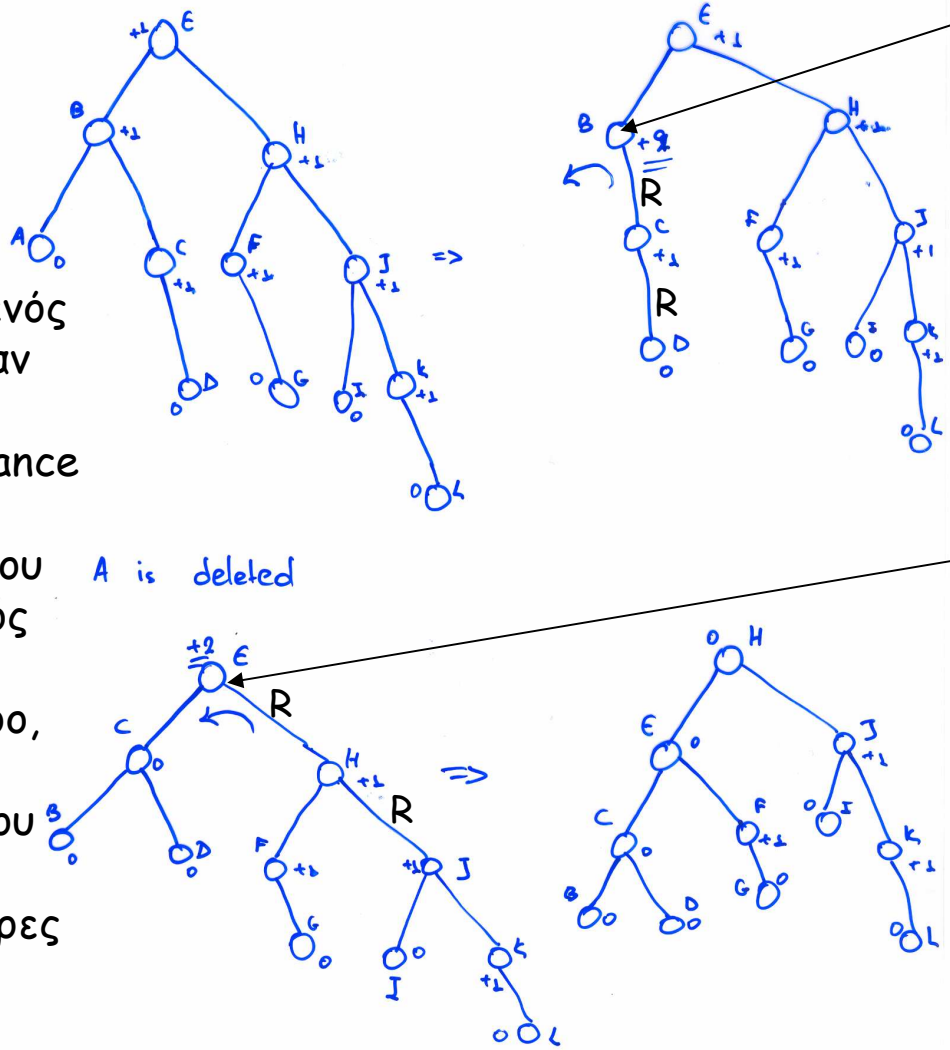


Αλγόριθμος

1. Εξετάζουμε όλο το μονοπάτι (ας το συμβολίσουμε με P1) από τον w ως τη ρίζα και για κάθε κόμβο του μονοπατιού αυτού που έχει balance +2 ή -2 μετά τη διαγραφή, ίσως χρειαστεί να γίνουν περιστροφές ως ακολούθως.
2. Έστω u ο πρώτος τέτοιος κόμβος στο P1.
3. Καθορίζουμε το μονοπάτι P2 με το μεγαλύτερο μήκος από τον u προς οποιοδήποτε φύλλο του υποδένδρου του και εξετάζουμε τις δύο πρώτες ακμές σε αυτό το μονοπάτι.
   - Αν αυτές είναι τύπου LL (ή RR) πραγματοποιούμε μια περιστροφή R (ή L, αντίστοιχα) γύρω από τον w.
   - Αν αυτές είναι τύπου RL (ή LR) πραγματοποιούμε δύο περιστροφές, την πρώτη τύπου R (L, αντίστοιχα) γύρω από τον επόμενο του w στο μονοπάτι P2 και τη δεύτερη τύπου L (R, αντίστοιχα) γύρω από τον w.
4. Επαναπροσδιορίζουμε τα balances των κόμβων στο μονοπάτι P1 και αν εξακολουθούν να υπάρχουν κόμβοι με μη-επιτρεπτό balance σε αυτό:
   - Θέτουμε u = πρώτος κόμβος στο μονοπάτι από τον u στη ρίζα με μη επιτρεπτό balance;
   - Πηγαίνουμε στο Βήμα 3;

Παράδειγμα Διαγραφής που Οδηγεί σε Πολλαπλές Περιστροφές

Αν υπάρχουν περισσότερα του ενός μονοπάτια από έναν κόμβο με προβληματικό balance προς φύλλα του υποδένδρου του που έχουν το ίδιο μήκος και το μήκος αυτό είναι το μεγαλύτερο, μπορεί να γίνει επιλογή εκείνου του μονοπατιού που οδηγεί στις λιγότερες και πιο απλές περιστροφές!



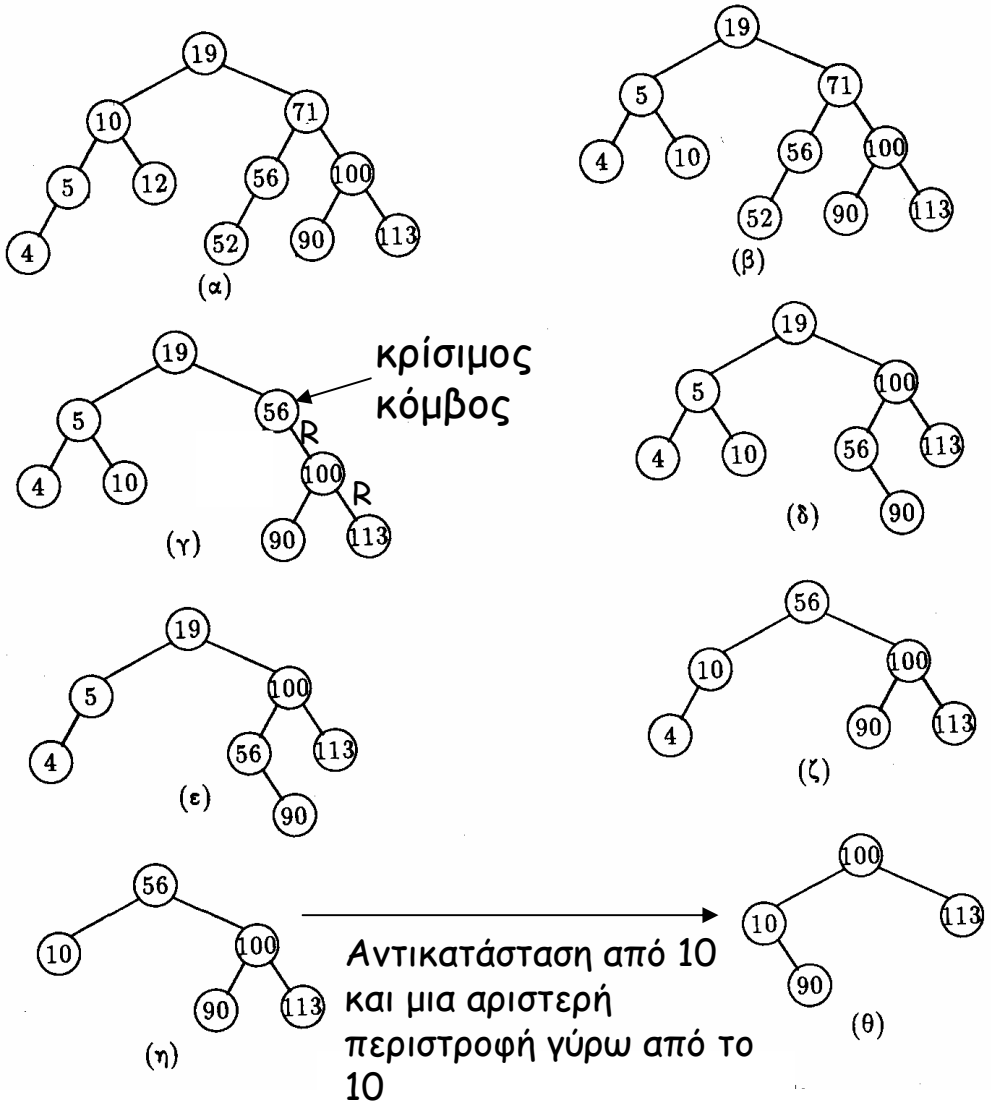
Κόμβος με μη-επιτρεπτό balance

Μακρύτερο μονοπάτι από αυτόν προς οποιοδήποτε φύλλο τύπου RR ⇒ απαιτείται μία περιστροφή τύπου L γύρω από τον κόμβο

Κόμβος με μη-επιτρεπτό balance

Μακρύτερο μονοπάτι από αυτόν προς οποιοδήποτε φύλλο τύπου RR ⇒ απαιτείται μία περιστροφή τύπου L γύρω από τον κόμβο

Δένδρα AVL – Διαγραφή



Παράδειγμα

Διαδοχική διαγραφή των κλειδιών 12, 71, 52, 10, 19, 4, και 56.

Ποια είναι η πολυπλοκότητα της AVLDelete()?

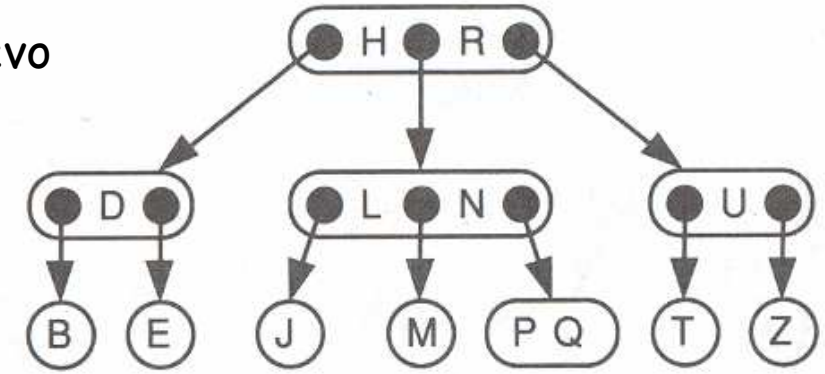
O(ύψος δένδρου).

Θεώρημα ⇒ ύψος δένδρου AVL = O(logn) ⇒

Χρονική Πολυπλοκότητα = O(logn)

2-3 Δένδρα

Ιδέα: Το δένδρο είναι τέλεια εξισορροπημένο (όλα τα φύλλα έχουν το ίδιο βάθος) αλλά δεν είναι δυαδικό.



❑ Σε ένα 2-3 δένδρο ένας κόμβος που δεν είναι φύλλο έχει δύο παιδιά και σε αυτόν αποθηκεύεται ένα στοιχείο του συνόλου (ένας τέτοιος κόμβος λέγεται 2-κόμβος), ή έχει τρία παιδιά και σε αυτόν αποθηκεύονται δύο στοιχεία του συνόλου (τότε ο κόμβος λέγεται 3-κόμβος).

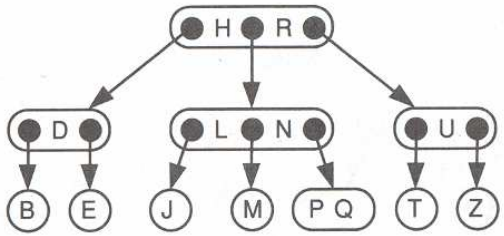
❑ Κάθε φύλλο που αποθηκεύει ένα στοιχείο του συνόλου είναι 2-κόμβος, ενώ κάθε φύλλο που αποθηκεύει δύο στοιχεία του συνόλου είναι 3-κόμβος.

❑ Με κατάλληλη διευθέτηση κόμβων και των δύο ειδών, μπορούμε να φτιάξουμε δένδρο στο οποίο όλα τα φύλλα έχουν το ίδιο βάθος και το οποίο αποθηκεύει οποιοδήποτε επιθυμητό πλήθος κλειδιών.

Ιδιότητες 2-3 Δένδρων

1. Όλα τα φύλλα έχουν το ίδιο βάθος και αποθηκεύουν 1 ή 2 στοιχεία.
2. Κάθε εσωτερικός κόμβος:
   - ❑ είτε αποθηκεύει ένα στοιχείο και έχει δύο παιδιά,
   - ❑ ή αποθηκεύει δύο στοιχεία και έχει τρία παιδιά.
3. Το δένδρο είναι ταξινομημένο.

2-3 Δένδρα



Ένα 2-3 δένδρο είναι ταξινομημένο αν για κάθε κόμβο v ισχύουν τα εξής.

- Αν ο v είναι 2-κόμβος, τα κλειδιά όλων των στοιχείων που βρίσκονται στο αριστερό υποδένδρο του v πρέπει να είναι μικρότερα από το κλειδί του v, ενώ εκείνα των στοιχείων που βρίσκονται στο δεξί υποδένδρο του v πρέπει να είναι μεγαλύτερα από εκείνα του v.
- Αν ο v είναι 3-κόμβος, το κλειδί (έστω K1) του πρώτου στοιχείου του v πρέπει να είναι μικρότερο από το κλειδί (έστω K2) του δέυτερου στοιχείου του v. Επίσης, τα κλειδιά όλων των στοιχείων στο πρώτο υποδένδρο του v πρέπει να είναι μικρότερα από K1, τα κλειδιά όλων των στοιχείων στο δεύτερο υποδένδρο του v πρέπει να είναι μεταξύ του K1 και του K2, ενώ τα κλειδιά του τρίτου υποδένδρου του v πρέπει να είναι μεγαλύτερα από K2.

Μεταξύ όλων των 2-3 δένδρων ύψους h, ποιο είναι εκείνο με τους λιγότερους κόμβους;
Εκείνο του οποίου όλοι οι εσωτερικοί κόμβοι είναι 2-κόμβοι.
Ποιο είναι το ύψος αυτού του δένδρου? $\log_2 n$

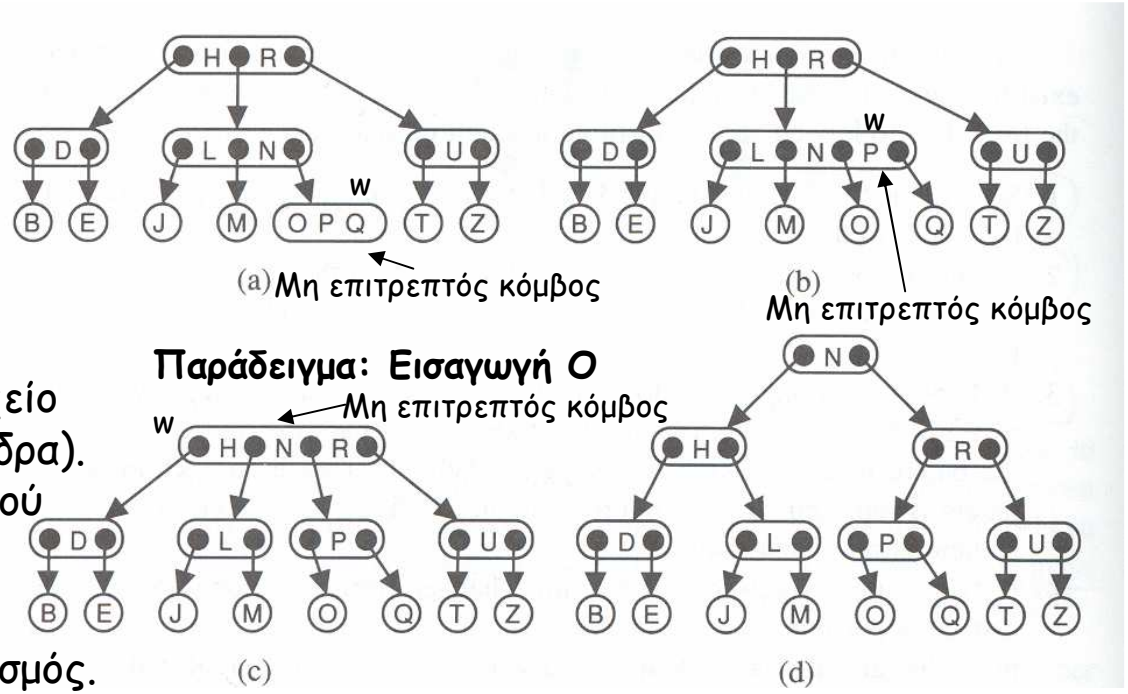
Μεταξύ όλων των 2-3 δένδρων ύψους h, ποιο είναι εκείνο με τους περισσότερους κόμβους;
Εκείνο του οποίου όλοι οι εσωτερικοί κόμβοι είναι 3-κόμβοι.
Ποιο είναι το ύψος αυτού του δένδρου? $\log_3 n$

Το ύψος ενός 2-3 δένδρου με n κόμβους είναι $\Theta(\log n)$!

Πως θα υλοποιήσουμε την LookUp() σε ένα 2-3 δένδρο?
Το δένδρο είναι ταξινομημένο!!!

Πολυπλοκότητα LookUp; $O(\log n)$

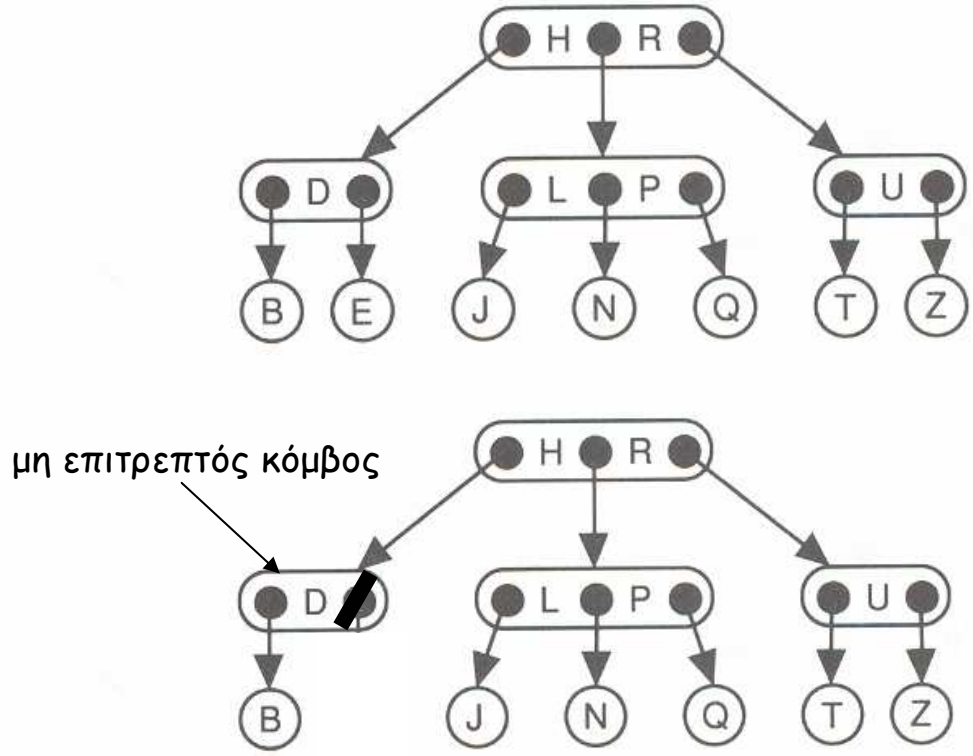
2-3 Δένδρα
Εισαγωγή



1. Εύρεση του φύλλου w στο οποίο θα έπρεπε σύμφωνα με την ιδιότητα της ταξινόμησης να εισαχθεί το νέο στοιχείο (όπως στα ταξινομημένα δυαδικά δένδρα).
2. Διατήρηση των κόμβων του μονοπατιού που ακολουθήθηκε σε στοίβα
3. Αν το w είναι 2-κόμβος, προσθήκη του νέου στοιχείου στο w και τερματισμός.
3. Αν το w είναι 3-κόμβος, μετατρέπεται σε κόμβο που πρέπει να αποθηκεύσει 3 στοιχεία (σε αύξουσα διάταξη) το οποίο είναι μη επιτρεπτό. Αντικαθιστούμε το w με δύο 2-κόμβους, έναν που αποθηκεύει το πρώτο και έναν που αποθηκεύει το τρίτο στοιχείο του w και το μεσαίο στοιχείο του w μεταφέρεται στον πατρικό του κόμβο. Αν δεν υπάρχει πατρικός κόμβος πηγαίνουμε στο βήμα 5.
4. Αν ο πατρικός κόμβος είναι 2-κόμβος, μετατρέπεται σε 3-κόμβο και ο αλγόριθμος τερματίζει. Διαφορετικά, θέτουμε w = <πατρικός κόμβος του w> και επιστρέφουμε στο βήμα 3 για να αντικαταστήσουμε τον πατρικό κόμβο του w με τον ίδιο τρόπο.
5. Η διαδικασία αυτή επαναλαμβάνεται μέχρι να βρούμε χώρο για το κλειδί σε κάποιο κόμβο στο μονοπάτι προς τη ρίζα ή να φτάσουμε στη ρίζα. Αν συμβεί το δεύτερο, η ρίζα χωρίζεται σε 2 κόμβους και ένας νέος κόμβος που αποτελεί τον πατρικό κόμβο αυτών των δύο κόμβων αποθηκεύει το μεσαίο στοιχείο που θα έπρεπε να αποθηκευθεί στη ρίζα. Έτσι, το ύψος του δένδρου αυξάνεται κατά 1.

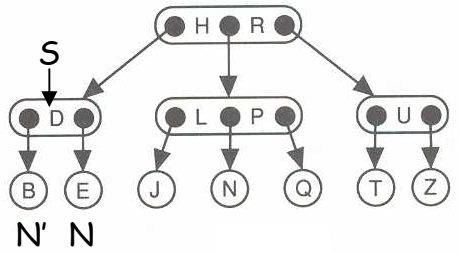
2-3 Δένδρα - Διαγραφή

Μπορεί να προκύψει το αντίστροφο πρόβλημα: μετά τη διαγραφή, κάποιος κόμβος μπορεί να έχει ένα μόνο παιδί.

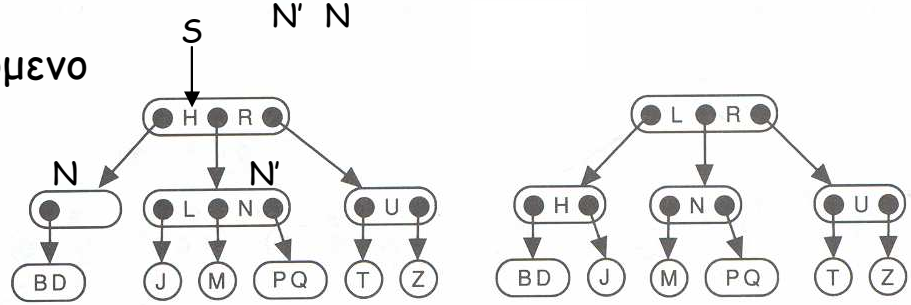


Παράδειγμα
Διαγραφή E

2-3 Δένδρα - Διαγραφή



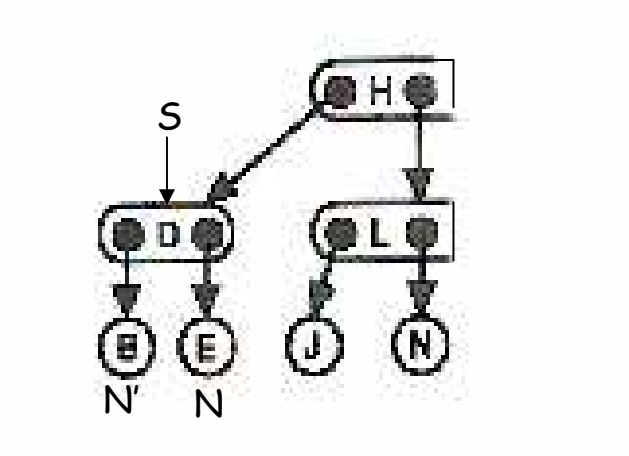
1. Αν το προς διαγραφή κλειδί (έστω K) περιέχεται σε κόμβο φύλλο, το διαγράφουμε. Διαφορετικά, το κλειδί (έστω K') που είναι επόμενο του K στην ενδο-διατεταγμένη διάσχιση περιέχεται σε φύλλο, οπότε αντικαθιστούμε το K με το K' και διαγράφουμε το K'.

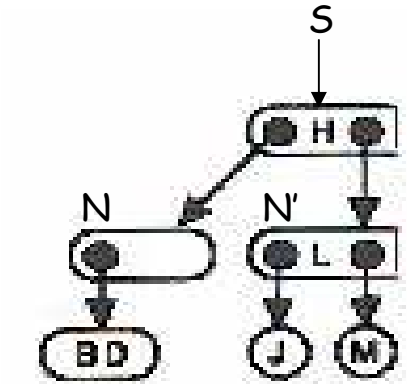


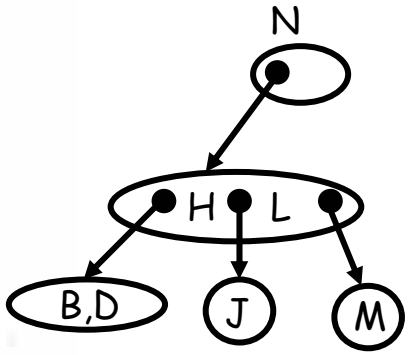
2. Έστω N ο κόμβος από τον οποίο διαγράφεται το κλειδί. Αν ο N εξακολουθεί να έχει ένα κλειδί, ο αλγόριθμος τερματίζει.
3. Αν ο N είναι η ρίζα, τον διαγράφουμε. **Σε αυτή την περίπτωση, ο N μπορεί να έχει ένα ή κανένα παιδί.** Αν δεν έχει κανένα παιδί, το δένδρο μετά τη διαγραφή είναι άδειο. Διαφορετικά, το παιδί του N είναι η νέα ρίζα του δένδρου.
4. Διαφορετικά, ο N έχει τουλάχιστον έναν αδελφικό κόμβο N'. Έστω P ο πατέρας των N, N' και έστω ότι S είναι το κλειδί του P που χωρίζει τα δύο υπο-δένδρα (που οδηγούν από τον P στους N και N').
   a. Ο N' είναι 3-κόμβος ακριβώς στα αριστερά (ακριβώς στα δεξιά) του N. Μετακινούμε το S στο N και αντικαθιστούμε το S στον πατέρα του με το δεξιότερο (αριστερότερο) κλειδί του N'. Αν οι N και N' είναι εσωτερικοί κόμβοι, μετατρέπουμε το δεξιότερο (αριστερότερο) παιδί του N' σε αριστερότερο (δεξιότερο) παιδί του N. Οι N, N' έχουν πλέον ένα κλειδί ο καθένας και ο αλγόριθμος τερματίζει.
   b. Ο N' είναι 2-κόμβος. Συνενώνουμε το S και το κλειδί του N' σε έναν νέο 3-κόμβο, ο οποίος αντικαθιστά τους N, N' και αποτελεί το μοναδικό παιδί του P (ο οποίος απομένει τώρα χωρίς κλειδί). Θέτουμε N = P και επαναλαμβάνουμε το βήμα 2.

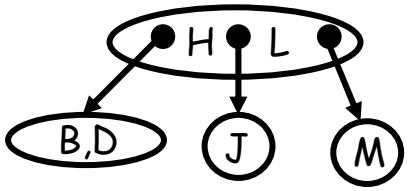
2-3 Δένδρα - Διαγραφή

Ένα ακόμη παράδειγμα







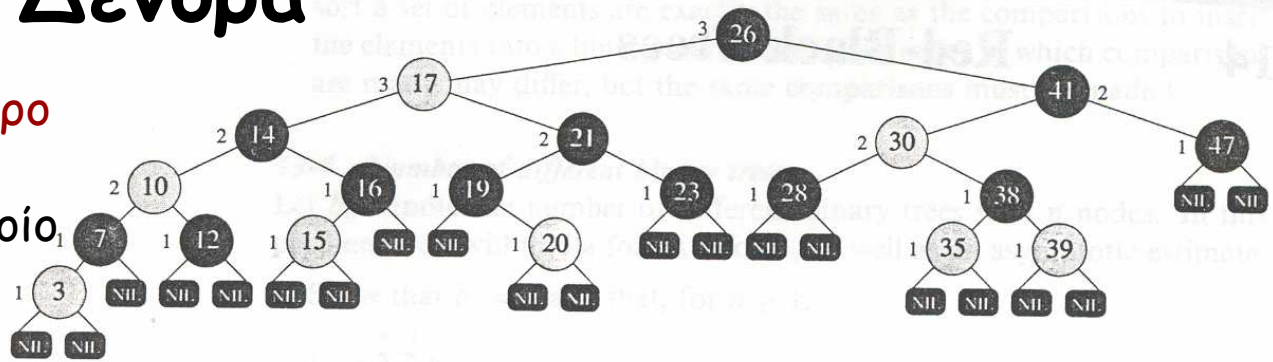


Πολυπλοκότητα Insert(); O(logn)

Πολυπλοκότητα Delete(); O(logn)

Πολυπλοκότητα LookUp(); O(logn)

Κοκκινόμαυρα Δένδρα



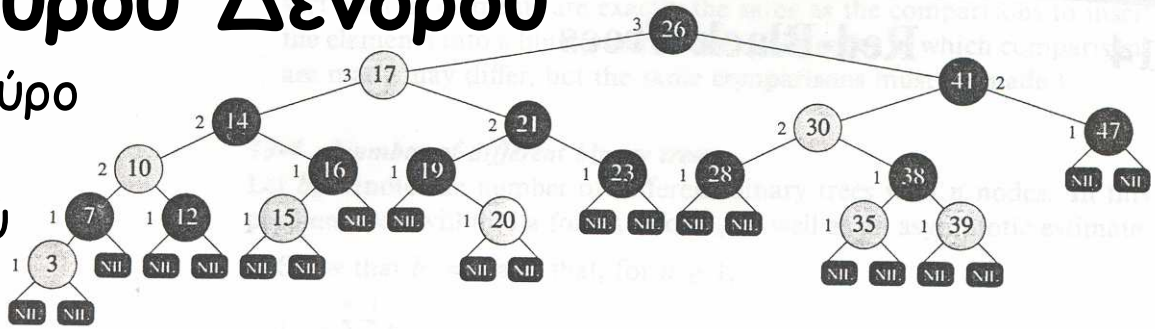
- Ένα κοκκινόμαυρο δένδρο είναι ένα ταξινομημένο δυαδικό δένδρο, στο οποίο κάθε κόμβος περιέχει ένα επιπλέον στοιχείο πληροφορίας, το χρώμα του, το οποίο μπορεί να είναι είτε μαύρο ή κόκκινο.
- Αν κάποιο παιδί κάποιου κόμβου δεν υφίσταται, ο αντίστοιχος δείκτης είναι nill. Θα θεωρούμε ότι δείκτες με τιμή nill είναι δείκτες προς εξωτερικούς κόμβους (φύλλα) του δυαδικού δένδρου και τους συνήθεις κόμβους που αποθηκεύουν κλειδιά ως εσωτερικούς κόμβους του δένδρου.

Ιδιότητες Κοκκινόμαυρων Δένδρων

1. Κάθε κόμβος έχει είτε κόκκινο είτε μαύρο χρώμα.
2. Το χρώμα της ρίζας είναι πάντα μαύρο.
3. Κάθε NIL κόμβος έχει μαύρο χρώμα.
4. Αν ένας κόμβος έχει κόκκινο χρώμα, τότε και τα δύο παιδιά του έχουν μαύρο χρώμα.
5. Κάθε μονοπάτι από οποιονδήποτε κόμβο προς οποιοδήποτε εξωτερικό κόμβο φύλλο που είναι απόγονός του περιέχει το ίδιο πλήθος μαύρων κόμβων.

Ύψος Κοκκινόμαυρου Δένδρου

Πρόταση: Ένα κόκκινο μαύρο δένδρο με n εσωτερικούς κόμβους έχει ύψος το πολύ 2log(n+1).



Διαίσθηση: Τουλάχιστον οι μισοί κόμβοι σε κάθε μονοπάτι από τη ρίζα σε φύλλο είναι μαύροι. Γιατί;

Απόδειξη

- Ορίζουμε το μαύρο ύψος ενός κόμβου v, το οποίο συμβολίζουμε με bh(v), να είναι το πλήθος των μαύρων κόμβων σε κάθε μονοπάτι από τον κόμβο v σε οποιοδήποτε φύλλο (εξωτερικό κόμβο nill), χωρίς να συμπεριλαμβάνουμε τον v.
- Αποδεικνύουμε επαγωγικά ότι το υπο-δένδρο με ρίζα κάποιο κόμβο v περιέχει τουλάχιστον $2^{bh(v)}-1$ εσωτερικούς κόμβους (η επαγωγή ως προς το ύψος του v).

Βάση επαγωγής: αν ο v έχει ύψος 0 τότε είναι φύλλο (δηλαδή εξωτερικός κόμβος), οπότε ο αριθμός των εσωτερικών κόμβων στο υποδένδρο που εκφύεται από τον v είναι 0 όπως απαιτείται.

Επαγωγική Υπόθεση: Δοθέντος ενός οποιουδήποτε κόμβου v, υποθέτουμε ότι ο ισχυρισμός ισχύει για τα παιδιά του v.

Ύψος Κοκκινόμαυρου Δένδρου

Απόδειξη (συνέχεια)

Επαγωγικό Βήμα: Αποδεικνύουμε τον ισχυρισμό για τον κόμβο v. Έστω w ένα οποιοδήποτε παιδί του v. Αν ο w είναι κόκκινος, τότε bh(w) = bh(v), ενώ αν ο w είναι μαύρος τότε bh(w) = bh(v) -1. Σε κάθε περίπτωση, bh(w) ≤ bh(v) -1. Αφού το ύψος του w είναι πιο μικρό από εκείνο του v, από επαγωγική υπόθεση, το υποδένδρο που εκφύεται από τον w έχει $2^{bh(w)}-1 \geq 2^{bh(v)-1} -1$ εσωτερικούς κόμβους. Άρα, το υποδένδρο που εκφύεται από τον v έχει τουλάχιστον
$2^* (2^{bh(v)-1} -1) + 1 = 2^{bh(v)} -1$ εσωτερικούς κόμβους, όπως απαιτείται.

Άρα, το υπο-δένδρο που εκφύεται από οποιονδήποτε κόμβο v περιέχει τουλάχιστον $2^{bh(v)}-1$ εσωτερικούς κόμβους. (1)

- Έστω h το ύψος του δένδρου και r η ρίζα. Είναι bh(r) ≥ h/2, αφού βάσει της ιδιότητας 4, σε οποιοδήποτε μονοπάτι από τη ρίζα προς οποιονδήποτε εξωτερικό κόμβο, τουλάχιστον οι μισοί κόμβο (εξαιρουμένης της ίδιας της ρίζας) θα πρέπει να είναι μαύροι.
- Άρα, από (1), $n \geq 2^{h/2} -1 \Rightarrow h \leq 2\log(n+1)$.

Πως υλοποιείται η LookUp(); Ποια είναι η πολυπλοκότητα της;

Κοκκινόμαυρα Δένδρα - Εισαγωγή

Αλγόριθμος

- Εισαγωγή του νέου στοιχείου με τον ίδιο τρόπο όπως σε ταξινομημένο δυαδικό δένδρο. Το μονοπάτι που ακολουθείται για την εισαγωγή του νέου κόμβου αποθηκεύεται σε στοίβα.
- Ο νέος κόμβος χαρακτηρίζεται κόκκινος.
- Αν οι ιδιότητες χρωματισμού εξακολουθούν να ισχύουν, ο αλγόριθμος τερματίζεται. Διαφορετικά, εκτελούνται ενέργειες που αποσκοπούν στην αποκατάσταση των ιδιοτήτων του δένδρου (οι οποίες περιγράφονται στη συνέχεια).

Για να απαιτείται η εκτέλεση ενεργειών αποκατάστασης του δένδρου θα πρέπει:

Ο πατρικός κόμβος του νέο-εισαχθέντα κόμβου να είναι κόκκινος. Γιατί;

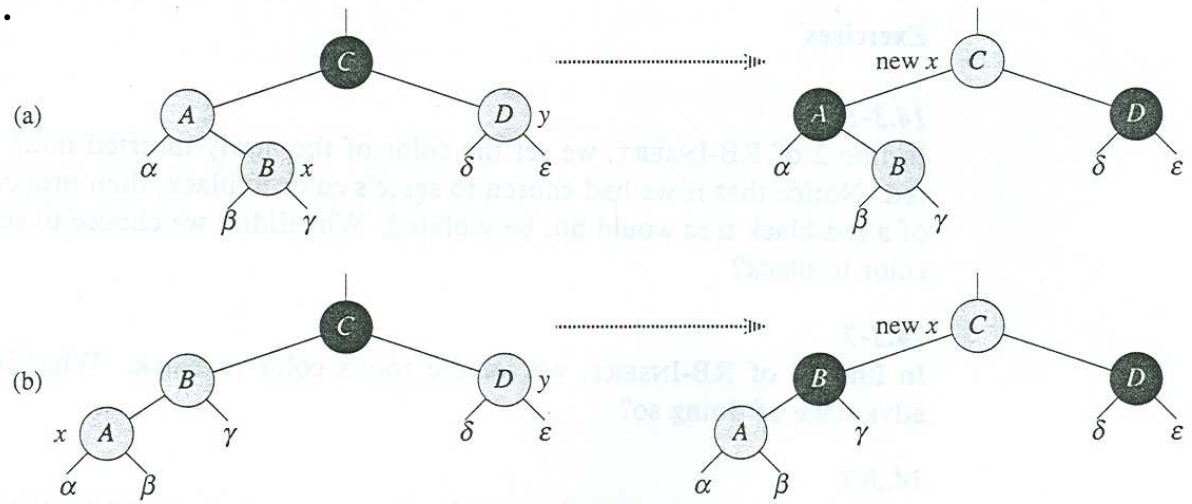
$\Rightarrow$ Ο πατρικός αυτός κόμβος δεν μπορεί να είναι η ρίζα.

Ο παππούς του νέο-εισαχθέντα κόμβου να είναι μαύρος. Γιατί;

Κοκκινόμαυρα Δένδρα - Εισαγωγή

Περιγραφή Ενεργειών Αποκατάστασης

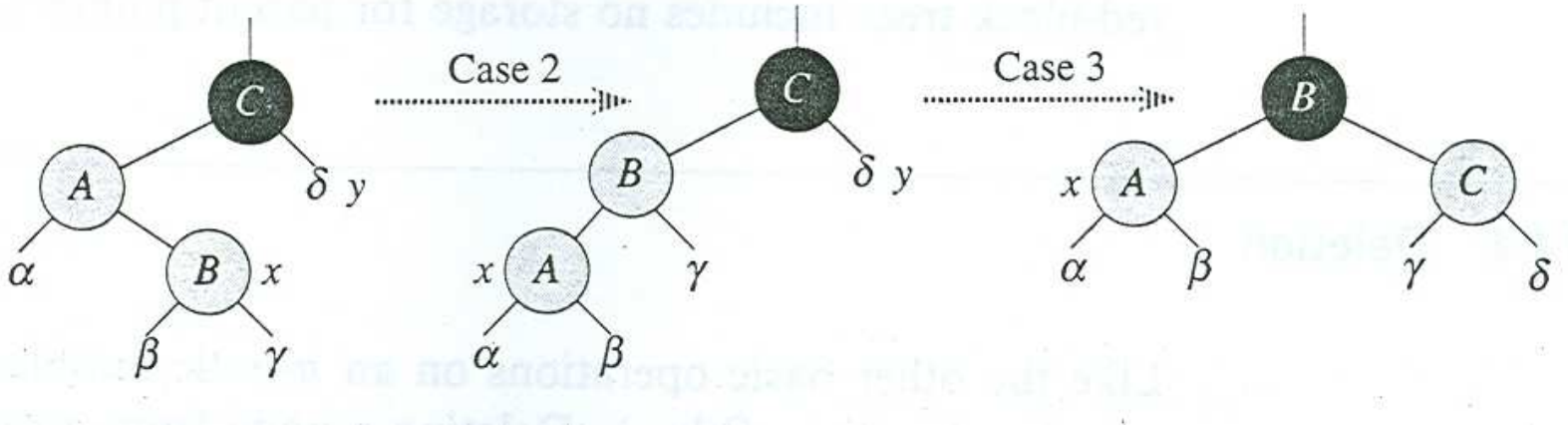
1. Υποθέτουμε ότι ο πατέρας του x είναι αριστερό παιδί του παππού του x (η περίπτωση που ο πατέρας του x είναι δεξιό παιδί του παππού του x είναι συμμετρική). Αν ο x και ο πατέρας του x είναι και οι δύο κόκκινοι κόμβοι, απαιτούνται ενέργειες αποκατάστασης των ιδιοτήτων του δένδρου. Διακρίνουμε περιπτώσεις:
2. **Περίπτωση 1:** Ο αδελφικός κόμβος y του πατέρα του x είναι κόκκινος. Τότε, αλλάζουμε το χρώμα του πατέρα του x και του αδελφικού του κόμβου y σε μαύρο και το χρώμα του παππού του x σε κόκκινο. Εκτελούμε την εντολή "x = παππούς του x". Επαναλαμβάνουμε ξεκινώντας από το Βήμα 1.



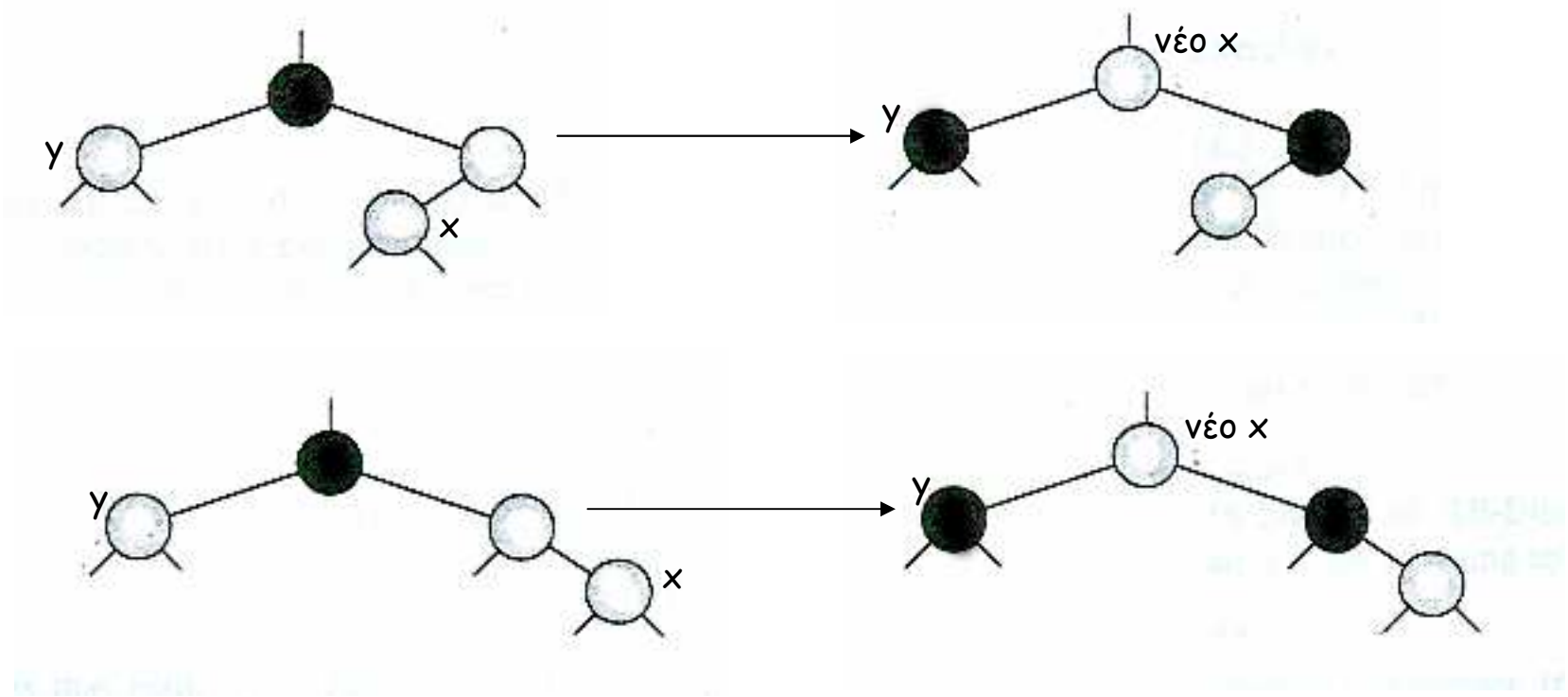
Κοκκινόμαυρα Δένδρα - Εισαγωγή

Περιγραφή Ενεργειών Αποκατάστασης (συνέχεια)

3. **Περίπτωση 2:** Ο y (αδελφικός κόμβος του πατέρα του x) είναι μαύρος και ο x είναι δεξί παιδί του πατέρα του. Ανάγουμε την περίπτωση αυτή στην περίπτωση 3 με την εκτέλεση μιας αριστερής περιστροφής γύρω από τον πατέρα του x. (Η περίπτωση που ο x είναι αριστερό παιδί του πατέρα του είναι συμμετρική).
4. **Περίπτωση 3:** Ο x είναι αριστερό παιδί του πατέρα του. Το χρώμα του πατέρα του x αλλάζει σε μαύρο και του παππού σε κόκκινο. Εκτελείται μια δεξιά περιστροφή γύρω από τον παππού του x.

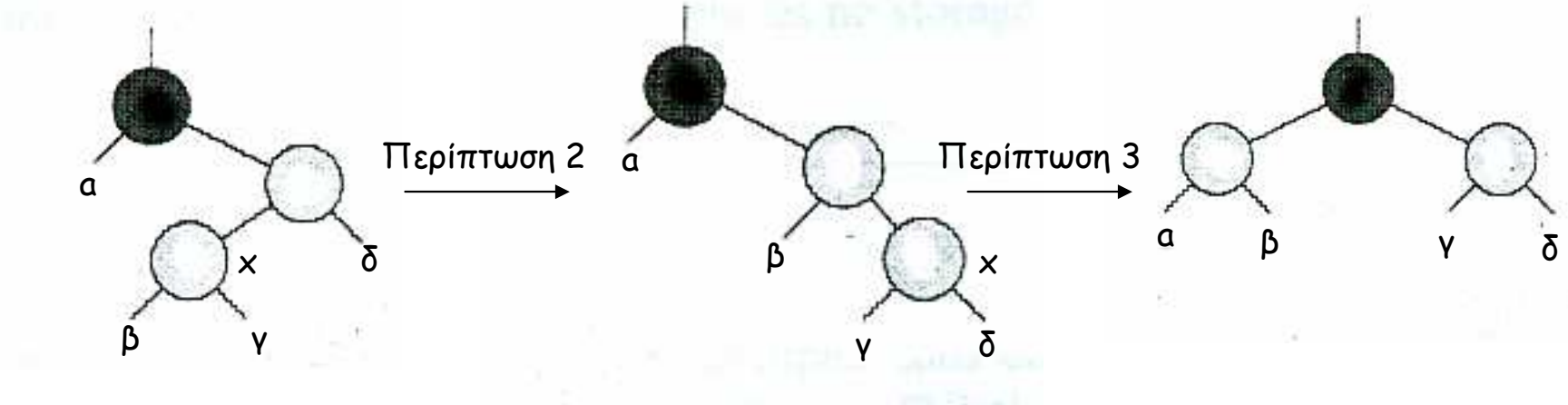


Κοκκινόμαυρα Δένδρα – Εισαγωγή – Κάποιες από τις Συμμετρικές Περιπτώσεις



Περίπτωση 1: Ο αδελφικός κόμβος y του πατρικού κόμβου του x είναι κόκκινος. (Συμμετρική περίπτωση που ο πατρικός κόμβος του x είναι δεξιό παιδί του παππού του x.)

Κοκκινόμαυρα Δένδρα – Εισαγωγή – Κάποιες από τις Συμμετρικές Περιπτώσεις



Περίπτωση 2: Ο γ (αδελφικός κόμβος του πατέρα του x) είναι μαύρος και ο x είναι αριστερό παιδί του πατέρα του. Ανάγουμε την περίπτωση αυτή στην περίπτωση 3 με την εκτέλεση μιας δεξιάς περιστροφής.

Περίπτωση 3: Ο x είναι δεξιό παιδί του πατέρα του. Το χρώμα του πατέρα του x αλλάζει σε μαύρο και του παππού σε κόκκινο. Εκτελείται μια αριστερή περιστροφή.

Κοκκινόμαυρα Δένδρα – Εισαγωγή - Ψευδοκώδικας

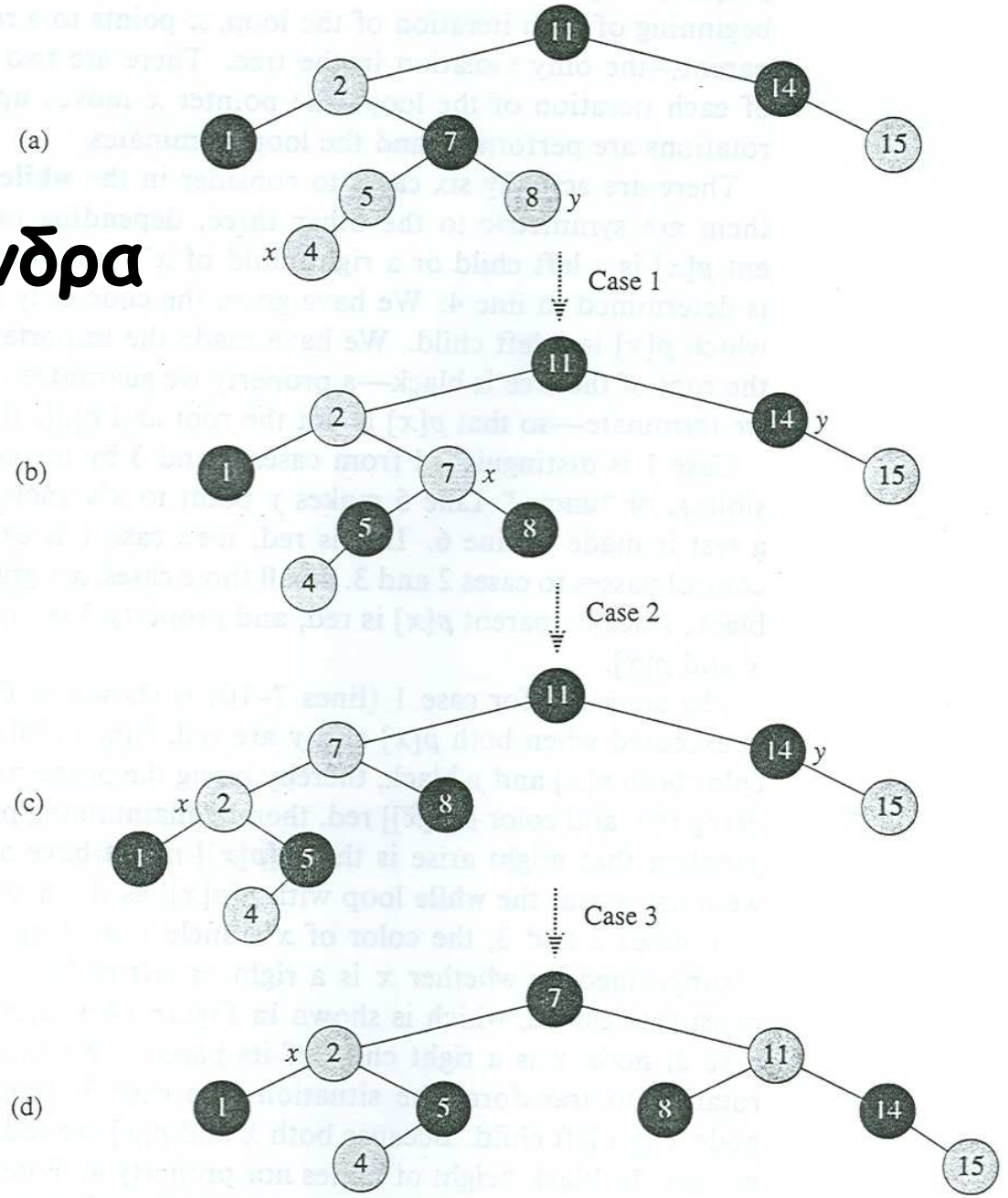
```
RedBlackTree-Insert(pointer R, Key k) {
/* Ο R είναι δείκτης στη ρίζα του δένδρου και το K είναι το
   προς-εισαγωγή κλειδί. Θεωρώ ότι το δένδρο είναι διπλά συνδεδεμένο */

   y = nill; z = R;
   while (z != null) {   // αγνοώ ελέγχους για το αν το κλειδί K βρίσκεται ήδη στο δένδρο
        y = z;
        if (K < z->Key) z = z->LC;
        else z = z->RC;
   }
   x = newcell(); x->Key = K;          x->LC = x->RC = null;          x->color = RED;
   x->p = y;
   if (y == null) (κάνε το x ρίζα του δένδρου);
   else if (x->key < y->key) y->lc = x;      // τοποθέτηση του x ως
   else y->rc = x;                           // κατάλληλου παιδιού του y
   ColorPropertiesInsert(R,x)
}
```

Κοκκινόμαυρα Δένδρα – Εισαγωγή

```
ColorPropertiesInsert(pointer R, pointer x) {
    while (x->p->color == RED) {     //αν το χρώμα του πατέρα του x είναι επίσης κόκκινο
        if (x->p == x->p->p->LC) {     // αν ο πατέρας του z είναι αριστερό παιδί του παππού του z
            y = x->p->p->RC;         // ο y είναι ο αδελφικός κόμβος του πατέρα του z
            if (y->color == RED) {
                x->p->color = BLACK;   // πατέρας και αδελφικός κόμβος πατέρα
                y->color = BLACK;        // γίνονται και οι δυο μαύροι
                x->p->p->color = RED;  // ο παππούς γίνεται κόκκινος
                x = x->p->p;                // (νέο x = παππούς x); και επαναλαμβάνουμε
            }
            else {
(1)             if (x == x->p->RC) {     // ο x είναι δεξιό παιδί του πατέρα του ⇒ Περίπτωση 2
                    x = x->p             // ο πατέρας του x θα παίξει το ρόλο του x (δείτε σχήματα)
                    LeftRotate(R, x);   // αριστερή περιστροφή γύρω από τον πατέρα του x
                }
                x->p->color = BLACK;
                x->p->p->color = RED;
                RightRotate(R, x->p->p);  // δεξιά περιστροφή γύρω από τον παππού του x
            }
        }
        else (ίδια με (1) με εναλλαγή του LC με RC και το αντίστροφο, καθώς και
              του LeftRotate με RightRotate και το αντίστροφο);
    } }
```

Κοκκινόμαυρα Δένδρα
Εισαγωγή
Παράδειγμα



Κοκκινόμαυρα Δένδρα – Διαγραφή

- Η διαγραφή γίνεται με παρόμοιο τρόπο όπως σε ταξινομημένο δυαδικό δένδρο και στη συνέχεια ελέγχεται αν οι ιδιότητες των κοκκινόμαυρων δένδρων ισχύουν ή όχι και γίνονται κατάλληλες ενέργειες.
- Έστω y ο κόμβος που θα διαγραφεί από το δένδρο.
- Αν το χρώμα του y είναι κόκκινο, ο y διαγράφεται και ο αλγόριθμος τερματίζει.
- Γιατί δεν προκύπτει πρόβλημα με τις ιδιότητες χρωματισμού του δένδρου σε αυτή την περίπτωση?
- Αν το χρώμα του y είναι μαύρο, η διαγραφή του δημιουργεί (τουλάχιστον) ένα μονοπάτι με μαύρο ύψος μικρότερο κατά ένα.
- Υποθέτουμε ότι η μαύρη ιδιότητα του y μεταφέρεται στο παιδί του, το οποίο αν είναι μαύρο γίνεται διπλά μαύρο (που είναι μη επιτρεπτό). Φανταζόμαστε πως κάθε κόμβος έχει ένα και μοναδικό κουπόνι που καθορίζει το χρώμα του (μαύρο ή κόκκινο). Τότε, ένας διπλά-μαύρος κόμβος είναι σαν να έχει δύο κουπόνια με μαύρο χρώμα. Αυτό είναι μη επιτρεπτό και ο κόμβος πρέπει να απαλλαγεί από το extra μαύρο κουπόνι.
- Προκειμένου να επιτευχθεί αυτό, το extra μαύρο κουπόνι μεταφέρεται προς τα πάνω (ακολουθώντας το μονοπάτι από τον κόμβο προς τη ρίζα) μέχρι είτε:
  1. να φθάσουμε στη ρίζα, ή
  2. να βρούμε έναν κόκκινο κόμβο που τον επαναχρωματίζουμε μαύρο και ο αλγόριθμος τερματίζει, ή
  3. να μπορούν να εκτελεστούν κατάλληλες περιστροφές και επαναχρωματισμοί κάποιων κόμβων ώστε να επιλυθεί το πρόβλημα.

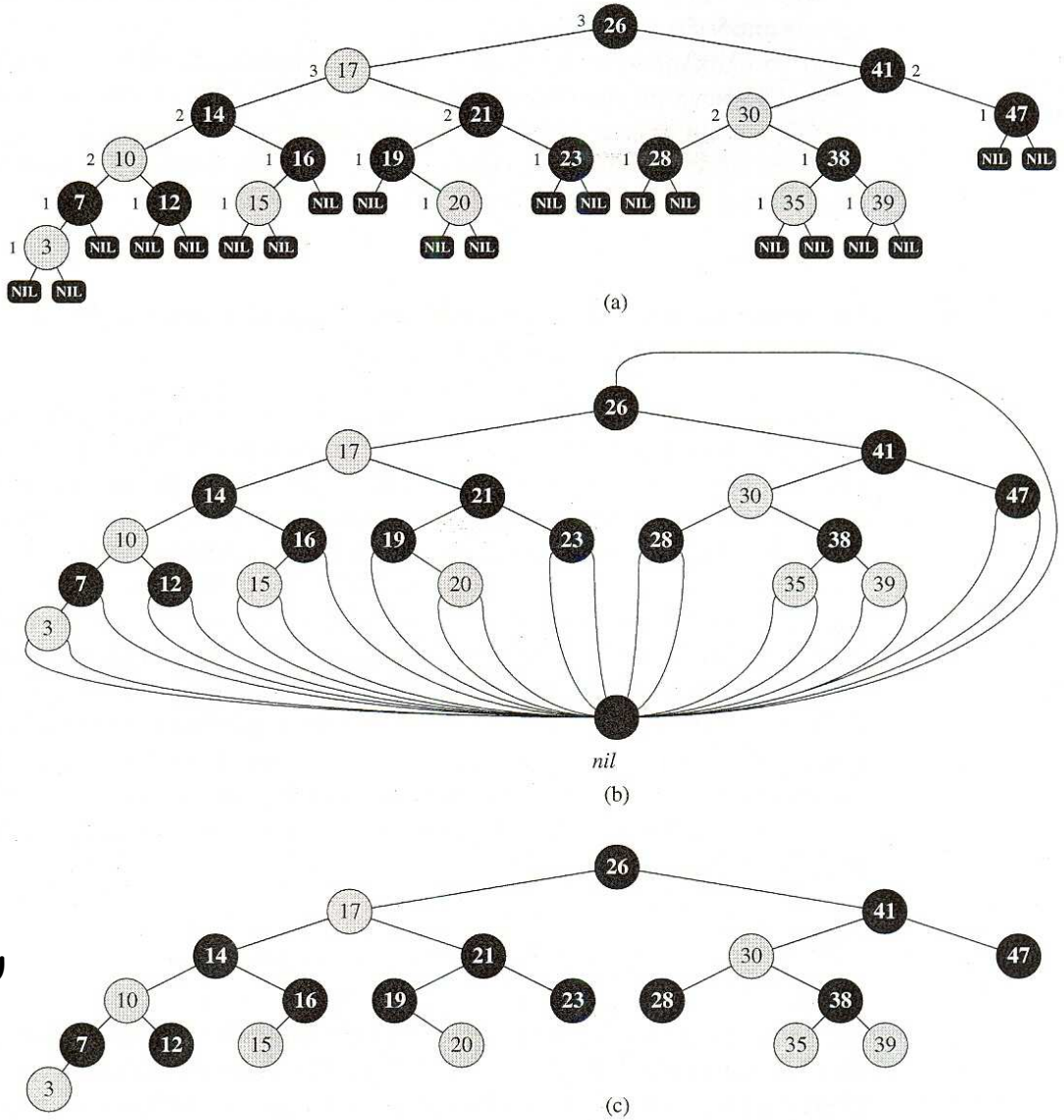
Κοκκινόμαυρα Δένδρα – Διαγραφή

Θεωρώ ότι το δένδρο υλοποιείται με κόμβο φρουρό. Έτσι, όλοι οι nill δείκτες δείχνουν στον κόμβο φρουρό.

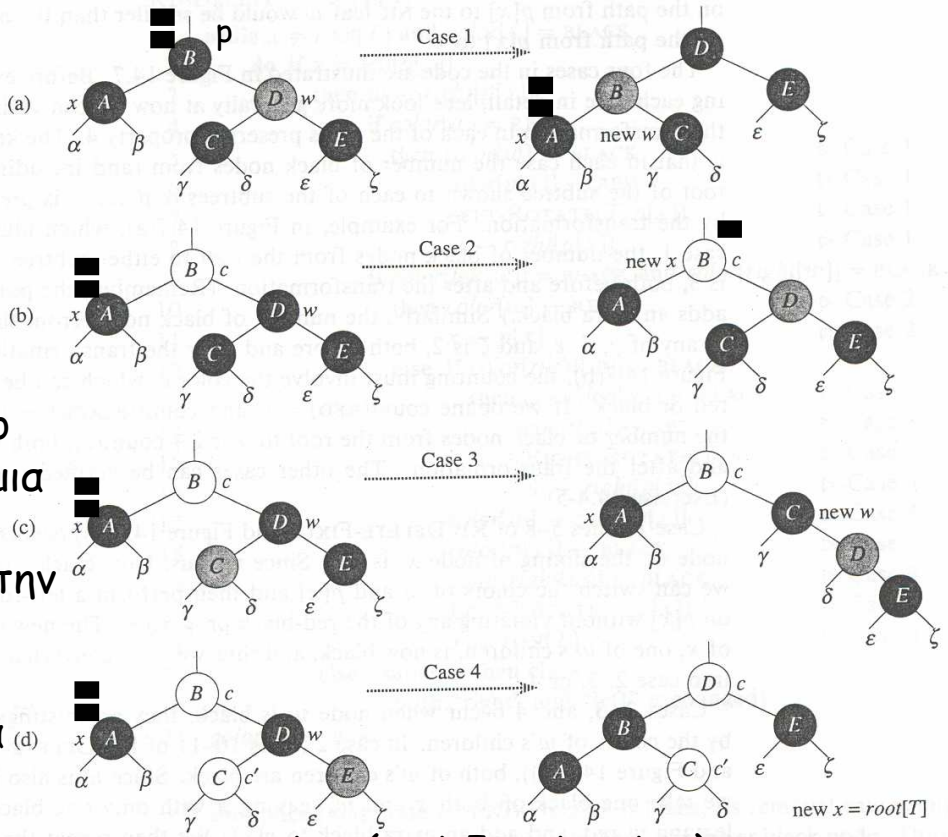
Έστω x το παιδί του κόμβου που διαγράφεται.

Έστω w ο αδελφικός κόμβος και p ο πατέρας του x.
Ο w δεν μπορεί να είναι ο κόμβος φρουρός. Γιατί;

Υποθέτουμε ότι ο x είναι αριστερό παιδί του πατρικού του κόμβου. (Η περίπτωση που ο x είναι δεξιό παιδί του πατρικού του κόμβου είναι συμμετρική.)



Κοκκινόμαυρα Δένδρα Διαγραφή



Διακρίνουμε περιπτώσεις ως προς το χρώμα του w.

1. **Περίπτωση 1:** Ο w είναι κόκκινος. Αλλάζουμε το χρώμα του w σε μαύρο και του p σε κόκκινο και εκτελούμε μια αριστερή περιστροφή γύρω από τον πατέρα του x. Έτσι, μεταπίπτουμε στην περίπτωση 2.
2. Ο w είναι μαύρος κόμβος.
   a. Περίπτωση 2: Και τα δύο παιδιά του w είναι μαύρα.

   Αλλάζουμε το χρώμα του w σε κόκκινο, του x σε μαύρο (από διπλά μαύρο) και μεταφέρουμε το μαύρο που αφαιρέσαμε από τους w, x στον p. Αν ο p ήταν κόκκινος γίνεται μαύρος και ο αλγόριθμος τερματίζει. Διαφορετικά, ο p γίνεται διπλά μαύρος και ο αλγόριθμος επαναλαμβάνεται με x = p.

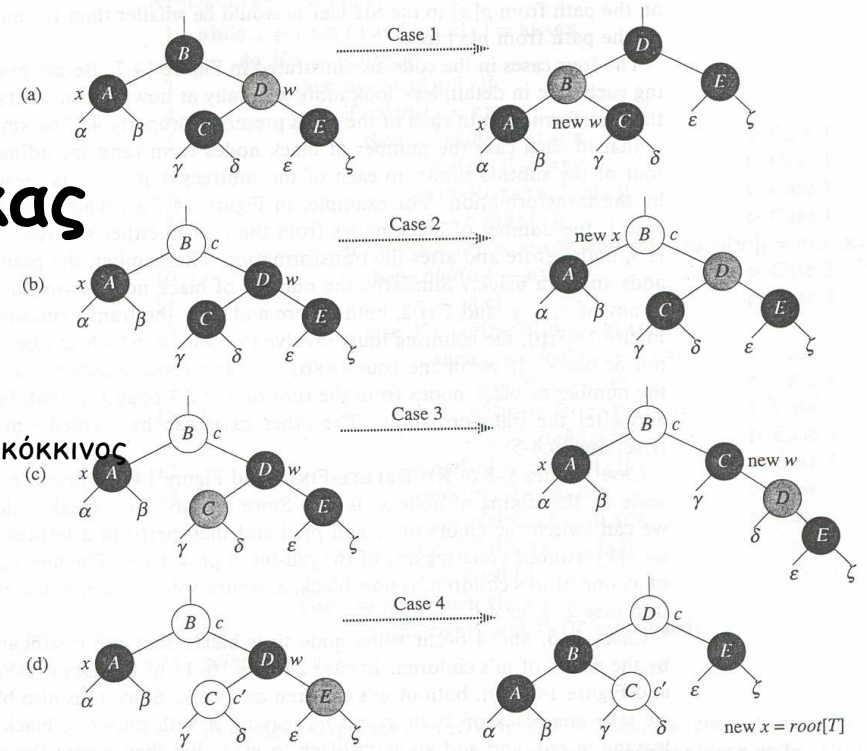
   b. Περίπτωση 3: Το w->lc είναι κόκκινο και το w->rc μαύρο. Αλλάζω το χρώμα του w σε κόκκινο και του w->lc σε μαύρο και εκτελώ μια περιστροφή γύρω από το w $\Rightarrow$ Μεταπίπτουμε στην περίπτωση 2c.

   c. Περίπτωση 4: Το w->rc είναι κόκκινο. Αλλάζω το χρώμα του w->rc σε μαύρο, του w σε ότι ήταν το χρώμα του p και του p σε μαύρο και εκτελώ μια περιστροφή γύρω από τον p. Ο αλγόριθμος τερματίζει.

Κοκκινόμαυρα Δένδρα – Διαγραφή - Ψευδοκώδικας

```
RedBlack-Delete(pointer R, pointer z) {
   if (z->RC == NULL OR z->LC == NULL)
      y = z;                                  // αν z έχει ένα ή κανένα παιδί, διαγράφεται το ίδιο το z
   else y = TreeSuccessor(R,z);               // διαφορετικά, διαγράφεται ο επόμενος κόμβος στην ενδ/γμένη διάσχιση
   if (y->LC != NULL) x = y->LC;              // ο x δείχνει στο μοναδικό παιδί του y
   else x = y->RC;
   x->p = y->p;
   if (y->p == NULL) (κάνε το x ρίζα του δένδρου);
   else if (y == y->p->LC) y->p->LC = x;      // αν ο y είναι αριστερό παιδί του πατέρα του
   else y->p->RC = x;
   if (y != z) {                              // αν ο προς διαγραφή κόμβος είναι ο επόμενος στην ενδ/γμένη διάσχιση
      z->Key = y->key;                        // αντιγραφή όλων των πεδίων δεδομένων
      (αντιγραφή των υπολοίπων πεδίων δεδομένων του y στο z);
   }
    if (y->color == BLACK) ColorPropertiesDelete(R,x);
                  // αν η διαγραφή αφορούσε κόμβο με χρώμα μαύρο, εκτέλεση κατάλληλων
                  //   ενεργειών για την επαναφορά των ιδιοτήτων χρωματισμού του δένδρου
    return y;
}
```

Κοκκινόμαυρα Δένδρα – Διαγραφή – Ψευδοκώδικας




```
ColorPropertiesDelete(pointer R, pointer x) {
  while (x != R AND x->color == BLACK) {
    if (x == x->p->LC) {  // αν ο x είναι αριστερό παιδί
      w = x->p->RC;
      if (w->color == RED) {      // αν αδελφικός του x κόκκινος
        w->color = BLACK;         // Περίπτωση 1
        x->p->color = RED;        // Περίπτωση 1
        LeftRotation(R, x->p);    // Περίπτωση 1
        w = x->p->RC;             // Περίπτωση 1
      }
      if (w->LC->color == BLACK AND
          w->RC->color == BLACK) {
                  // αν και τα δύο παιδιά του w είναι μαύρα
        w->color = RED;           // Περίπτωση 2
        x = x->p;                 // Περίπτωση 2
      }
      else {
        if (w->RC->color == BLACK) {
                                  // αν δεξιό παιδί του x μαύρο
          w->LC->color = BLACK;   // Περίπτωση 3
          w->color = RED;         // Περίπτωση 3
          RightRotation(R,w);     // Περίπτωση 3
          w = x->p;               // Περίπτωση 3
        }
        w->color = x->p->color;   // Περίπτωση 4
        x->p->color = RED;        // Περίπτωση 4
        w->RC->color = BLACK;     // Περίπτωση 4
        LeftRotation(R, x->p);    // Περίπτωση 4
        x = R;                    // Περίπτωση 4
      }  /* else */
    }  /* if */
    else (ίδια με (2) με εναλλαγή του LC με RC
          και το αντίστροφο, καθώς και του
          LeftRotate με RightRotate και το
          αντίστροφο);
  }
  x->color = BLACK;
}
```

(2)