Τμήμα Πληροφορικής

Πανεπιστήμιο Ιωαννίνων

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Παναγιώτα Φατούρου

faturu@cs.uoi.gr

ΥΠΟΥΡΓΕΙΟ ΕΘΝΙΚΗΣ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ ΕΠΕΑΕΚ

ΕΥΡΩΠΑΪΚΗ ΕΝΩΣΗ
ΣΥΓΧΡΗΜΑΤΟΔΟΤΗΣΗ
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

ΕΛΛΑΔΑ 2008 ΚΟΙΝΩΝΙΚΟ ΠΛΑΙΣΙΟ ΣΤΗΡΙΞΗΣ
Ανάπτυξη παντού. Ανάπτυξη για όλους.

Η ΠΑΙΔΕΙΑ ΣΤΗΝ ΚΟΡΥΦΗ
Επιχειρησιακό Πρόγραμμα
Εκπαίδευσης και Αρχικής
Επαγγελματικής Κατάρτισης

Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Τ.Θ. 1186, Γραφείο Α26,
Τηλ. +30 26510 98808, Fax: +30 26510 98890, URL: http://www.cs.uoi.gr/~faturu/

ΕΝΟΤΗΤΑ 1
ΕΙΣΑΓΩΓΗ

Δεδομένα

Σύνολο από πληροφορίες που πρέπει να αποθηκευτούν σε έναν υπολογιστή.

Αλγόριθμος

Αλγόριθμος είναι ένα πεπερασμένο σύνολο βημάτων/εντολών αυστηρά καθορισμένων (και εκτελέσιμων μέσα σε πεπερασμένο χρόνο), τα οποία αν ακολουθηθούν επιλύεται κάποιο πρόβλημα.

Είσοδος: Δεδομένα που παρέχονται εξ αρχής στον αλγόριθμο.

Έξοδος: δεδομένα που αποτελούν το αποτέλεσμα του αλγορίθμου.

Ευκρίνια/Αποτελεσματικότητα: Κάθε εντολή θα πρέπει να είναι απλή και ο τρόπος εκτέλεσής της να καθορίζεται χωρίς καμία αμφιβολία.

Περατότητα: Ο αλγόριθμος θα πρέπει να τερματίζει μετά από πεπερασμένα βήματα εκτέλεσης των εντολών του.

Πρόγραμμα

Υλοποίηση ενός αλγορίθμου σε κάποια γλώσσα προγραμματισμού.

Παράδειγμα

Εύρεση ενός στοιχείου Κ σε έναν ταξινομημένο πίνακα?

Αλγόριθμοι (περιγραφή σε φυσική γλώσσα)

Ξεκινώντας από το πρώτο στοιχείο του πίνακα προσπέλασε κάθε στοιχείο του πίνακα και εξέτασε αν είναι το Κ, μέχρι είτε να βρεθεί το Κ, ή να εξεταστούν όλα τα στοιχεία του πίνακα.

Ξεκινώντας από το πρώτο στοιχείο του πίνακα προσπέλασε κάθε στοιχείο του πίνακα και εξέτασε αν είναι το Κ, μέχρι είτε να βρεθεί το Κ, ή να βρεθεί κάποιο στοιχείο μεγαλύτερο από το Κ ή να εξεταστούν όλα τα στοιχεία του πίνακα.

Δυαδική Αναζήτηση

Ξεκίνησε από το μεσαίο στοιχείο του πίνακα. Αν αυτό είναι το Κ επέστρεψε. Διαφορετικά, σύγκρινε το μεσαίο στοιχείο με το Κ. Αν είναι μικρότερο, το Κ βρίσκεται στο δεξί μισό του πίνακα, αν όχι τότε βρίσκεται στο αριστερό μισό του πίνακα. Σε κάθε περίπτωση μόνο το ένα μισό του πίνακα πρέπει να εξεταστεί. Επανέλαβε την διαδικασία αυτή μέχρι είτε να βρεθεί το Κ ή το προς εξέταση μέγεθος του πίνακα να μηδενιστεί.

Παράδειγμα

Ύψωση ενός αριθμού x σε μια ακέραια δύναμη n.

Αλγόριθμος 1 (Περιγραφή με ψευδο-κώδικα)

```
double power1(double x, int n)
    int j = 0;
    double y = 1;

    while (j < n)
        y = y*x;
        j = j+1;
    return y;
```

Υλοποίηση του Αλγ. 1 στη γλώσσα C

```
double power_1(double x, int n) {
    int j;
    double y = 1;

    j = n;                  /*  j = 0 */
    while (j > 0 ) {        /*  while (j < n) { */
        y = y*x;
        j = j -1;           /* j = j +1 */
    }
    return y;
}
```

Κριτήρια Επιλογής Αλγορίθμων

- Ταχύτητα
- Απαιτούμενος χώρος μνήμης
- Ευκολία προγραμματισμού
- Γενικότητα

Μας ενδιαφέρει κυρίως η ταχύτητα και ο απαιτούμενος χώρος μνήμης. Οι δύο αυτοί παράμετροι καθορίζουν την αποδοτικότητα ενός αλγόριθμου.

Είναι η $Power_1$ ο πιο αποδοτικός αλγόριθμος για το πρόβλημα μας?

Αλγόριθμος $Power_2$: είσοδος & έξοδος ίδια με πριν

```
    double y
1.  if (n == 1) return x
2.  y = power2(x, ⌊n/2⌋)
3.  if n is even
4.          return y*y;
5.  else
6.          return x*y*y;
```

Μοντέλο Εργασίας

Ο υπολογιστής στον οποίο δουλεύουμε έχει έναν επεξεργαστή και ένα μεγάλο μπλοκ μνήμης.

Κάθε θέση μνήμης έχει μια αριθμητική διεύθυνση (διεύθυνση μνήμης).

Η μνήμη είναι τυχαίας προσπέλασης, το οποίο σημαίνει πως η πρόσβαση σε οποιαδήποτε θέση μνήμης (ανεξάρτητα από τη διεύθυνση στην οποία βρίσκεται) γίνεται με την ίδια ταχύτητα.

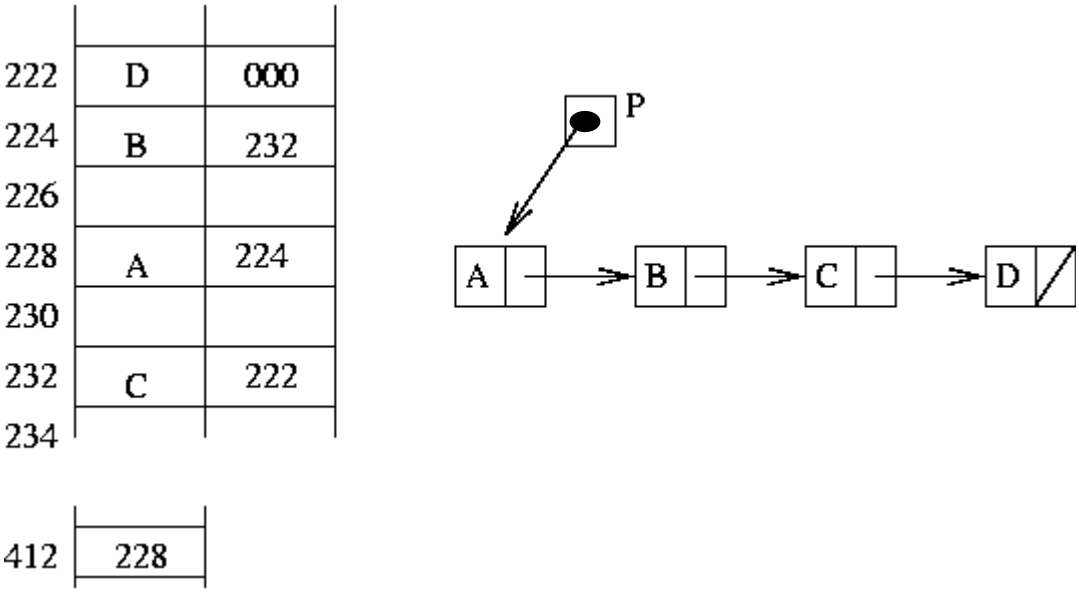
Συνεχόμενες θέσεις μνήμης μπορούν να αποθηκεύσουν τα πεδία μιας εγγραφής. Τα πεδία μιας εγγραφής μπορεί να μην είναι του ίδιου τύπου.

Συνεχόμενες θέσεις μνήμης μπορούν επίσης να αποθηκεύσουν ένα πίνακα (δηλαδή ένα σύνολο στοιχείων του ίδιου τύπου).

Μια μεταβλητή στην οποία είναι αποθηκευμένη μια διεύθυνση μνήμης ονομάζεται μεταβλητή δείκτη (ή δείκτης).

Γιατί οι δείκτες είναι πολύ χρήσιμοι;

Απλά Συνδεδεμένη Λίστα



Γιατί οι λίστες είναι πολύ χρήσιμες;

Σύγκριση με Πίνακες

Θετικά

☺ Εισαγωγή/διαγραφή νέων στοιχείων γίνεται πολύ εύκολα

☺ Ο συνολικός αριθμός στοιχείων δεν χρειάζεται να είναι γνωστός εξ αρχής

Αρνητικά

☹ Απαιτούν περισσότερη μνήμη (λόγω των δεικτών).

☹ Ανάκτηση στοιχείου για το οποίο είναι γνωστή η θέση του στη δομή (π.χ., ανάκτηση του $5^{ου}$ στοιχείου) δεν μπορεί να γίνει σε σταθερό χρόνο.

Αφηρημένος Τύπος Δεδομένων

Αποτελείται από δύο μέρη:

- Ένα ή περισσότερα πεδία (σύνολα αντικειμένων, classes of mathematical objects)
- Ένα σύνολο λειτουργιών (mathematical operations) στα στοιχεία αυτών των πεδίων.

Παράδειγμα (εύρεση στοιχείου σε πίνακα)

- Τα δεδομένα μας είναι κάποιου τύπου, έστω key, και υπάρχει μια γραμμική διάταξη ανάμεσά τους:

  $\forall$ *u,v τύπου key, είτε* $u < v$, *ή* $v < u$, *ή* $v = u$.

- Έχουμε ένα σύνολο S από στοιχεία τύπου key και θέλουμε να μπορούμε να απαντάμε το ερώτημα: u $\in$ S?

Πεδία:

- στοιχεία τύπου key (π.χ., u,v) και πεπερασμένα σύνολα αυτών (π.χ., S)

Σύνολο λειτουργιών:

- Παροχή true ή false απάντησης στην ερώτηση «u $\in$ S?”, όπου: u είναι στοιχείο τύπου key, S είναι πεπερασμένο σύνολο από στοιχεία τύπου key.

Δομή Δεδομένων

Μια δομή δεδομένων υλοποιεί έναν αφηρημένο τύπο δεδομένων.

Μια δομή δεδομένων επομένως συμπεριλαμβάνει:

- ένα σύνολο αποθηκευμένων δεδομένων τα οποία μπορούν να υποστούν επεξεργασία από ένα σύνολο λειτουργιών που σχετίζονται με τη συγκεκριμένη δομή
- μια δομή αποθήκευσης
- ένα σύνολο από ορισμούς συναρτήσεων, όπου η κάθε συνάρτηση εκτελεί μια λειτουργία στο περιεχόμενο της δομής, και
- ένα σύνολο από αλγόριθμους, έναν αλγόριθμο για κάθε συνάρτηση.

Οι βασικές λειτουργίες επί των δομών δεδομένων είναι οι ακόλουθες:

- Προσπέλαση
- Εισαγωγή
- Διαγραφή
- Αναζήτηση
- Ταξινόμηση
- Αντιγραφή
- Συγχώνευση
- Διαχωρισμός

Μαθηματικό Υπόβαθρο

- Μονοτονία Συναρτήσεων
- Ακέραια Μέρη πραγματικών αριθμών
  $\lfloor x \rfloor, \lceil x \rceil, [x]$
- Πολυώνυμα – Πολυωνυμικές Συναρτήσεις
  $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$
- Λογάριθμοι & Εκθέτες – Ιδιότητες
- Χρήσιμοι Συμβολισμοί:
  $\lg n = \log_2 n$
  $\ln n = \log_e n$
  $\lg^k n = (\lg n)^k$
  $\lg \lg n = \lg(\lg n)$
- Παραγοντικά
  $n!$
- Άθροισμα όρων γεωμετρικής ή αριθμητικής προόδου

Μαθηματική Επαγωγή

Παράδειγμα 1 – Βασική Μέθοδος

Ορθότητα Αλγόριθμου Power$_1$

Με επαγωγή στο j.

Συμβολίζουμε με y_j την τιμή της μεταβλητής y στην αρχή της j-οστής ανακύκλωσης, j = 1, …, n+1

Αρκεί να δείξουμε πως $y_j = x^{j-1}$.

Βάση επαγωγής

j = 1. Αρχικά, $y_1 = 1 = x^0 = x^{1-1} = x^{j-1}$.

Επαγωγική Υπόθεση

Έστω ότι για κάποιο k, $1 \leq k < n+1$, $y_k = x^{k-1}$.

Επαγωγικό Βήμα

Θα δείξουμε ότι ο ισχυρισμός ισχύει για (k+1): $y_{k+1} = x^k$.

Από αλγόριθμο: $y_{k+1} = y_k * x$.

Από επαγωγική υπόθεση: $y_k = x^{k-1}$.

Άρα, $y_{k+1} = x^k$, όπως απαιτείται.

Παράδειγμα 3: Ορθότητα Προγ/τος Power_1

Με επαγωγή στο j, j = n, …, 0.

Αρκεί να δείξω ότι $y_{n-j+1}=x^{n-j}$

Βάση Επαγωγής

j = n:

$y_{n-n+1} = y_1 = 1 = x^0 = x^{n-n}$, όπως απαιτείται.

Επαγωγική Υπόθεση

Έστω ότι για κάποιο k, $n \geq k > 0$, ισχύει ότι $y_{n-k+1} = x^{n-k}$.

Επαγωγικό Βήμα

Θα δείξουμε ότι ο ισχυρισμός ισχύει για (k-1): $y_{n-k+2} = x^{n-k+1}$.

Από αλγόριθμο: $y_{n-k+2} = y_{n-k+1} * x$.

Από επαγωγική υπόθεση: $y_{n-k+1} = x^{n-k}$.

Άρα, $y_{n-k+2} = x^{n-k+1}$, όπως απαιτείται.

Παράδειγμα 3 – Ισχυρή Επαγωγή
Αιγυπτιακός Πολλαπλασιασμός

Δίνεται η συνάρτηση:

$$m(x,y) = \begin{cases} 0, \text{ αν } y = 0 \\ m(x+x, y/2), \text{ αν } y \text{ ζυγός } \& \neq 0 \\ x + m(x, y-1), \text{ διαφορετικά} \end{cases}$$

Θα δείξω ότι $m(x,y) = x*y$, $\forall$ ακέραιο x,y

Απόδειξη

Με επαγωγή στο y.

Βάση της επαγωγής

Αν $y = 0$, $m(x,y) = 0$, αλλά και $x*y = 0$, οπότε ισχύει.

Επαγωγική Υπόθεση

Φιξάρουμε μια τιμή του $y > 0$. Υποθέτουμε ότι $m(x,z) = x*z$, για κάθε z, $0 \leq z < y$.

Επαγωγικό Βήμα

Αποδεικνύουμε τον ισχυρισμό για την τιμή y: $m(x*y) = x*y$.

- y περιττός: $m(x,y) = x + m(x,y-1)$. Επαγωγική υπόθεση ($z = y-1 < y$): $m(x,y-1) = x*(y-1)$. Άρα: $m(x,y) = x + x*(y-1) = x + x*y - x = x*y$
- y άρτιος: $m(x,y) = m(x+x, y/2)$. Επαγωγική υπόθεση ($z = y/2 < y$): $m(x,y) = (x+x)*y/2 = x*y$. Άρα: $m(x,y) = x*y$.

Ανάλυση Αλγορίθμων

Εμπειρικός τρόπος μέτρησης της επίδοσης

Θετικά: απλότητα

Αρνητικά:

- Δύσκολο να προβλεφθεί η συμπεριφορά για άλλα σύνολα δεδομένων.
- Ο χρόνος επεξεργασίας εξαρτάται από το υλικό, τη γλώσσα προγραμματισμού και το μεταφραστή, αλλά και από το πόσο δεινός είναι ο προγραμματιστής.

Θεωρητικός τρόπος μέτρησης της επίδοσης

Εισάγεται μια μεταβλητή n που εκφράζει το μέγεθος του προβλήματος.

Παραδείγματα

- Στο πρόβλημα ανυψώσεως σε δύναμη το μέγεθος του προβλήματος είναι το n: η δύναμη στην οποία πρέπει να υψωθεί ο δεδομένος αριθμός.
- Σε ένα πρόβλημα ταξινόμησης ενός πίνακα, το μέγεθος του προβλήματος είναι ο αριθμός στοιχείων του πίνακα.

Ανάλυση Αλγορίθμων

Πρόβλημα Ταξινόμησης

Είσοδος (input): Μια ακολουθία από n αριθμούς $<a_1, a_2, ..., a_n>$.

Έξοδος (output): Μια «αναδιάταξη» $< a'_1, a'_2, ..., a'_n>$ της ακολουθίας εισόδου έτσι ώστε:

$$a'_1 \leq a'_2 \leq ... \leq a'_n$$

InsertionSort

```
void InsertionSort(int A[n]) {
/* ταξινόμηση των στοιχείων του A */
    int key, i, j;
    for (j = 1; j < n; j++) {
        key = A[j];
        i = j-1;
        while (i >= 0 && A[i] > key) {
            A[i+1] = A[i];
            i = i-1;
        }
        A[i+1] = key;
    }
}
```

Πως λειτουργεί ο αλγόριθμος αν A = <5,2,4,6,1,3>?

Ανάλυση του Αλγορίθμου InsertionSort

Ο χρόνος που σπαταλιέται από την InsertionSort εξαρτάται από την είσοδο:

«Η ταξινόμηση 1000 αριθμών απαιτεί περισσότερο χρόνο από την ταξινόμηση 3 αριθμών»!

Αυτό ισχύει γενικότερα:

«Ο χρόνος που απαιτεί ένας αλγόριθμος για να εκτελεστεί εξαρτάται από το μέγεθος της εισόδου»!

Χρόνος Εκτέλεσης

Ο χρόνος εκτέλεσης (running time) ενός αλγορίθμου **για μια συγκεκριμένη είσοδο** είναι ο αριθμός των βασικών λειτουργιών/εντολών (ή βημάτων) που εκτελούνται κατά την εκτέλεση του αλγορίθμου με αυτή την είσοδο.

Ανάλυση Αλγορίθμων

```
void InsertionSort(int A[n]) {                          Κόστος   Χρόνος
    int key, i, j;
    for (j = 1; j < n; j++) {          ---------->        c1        n
        key = A[j];                    ---------->        c2        n-1
        i = j-1;                       ---------->        c3        n-1
        while (i >= 0 && A[i] > key) { ---------->        c4        Σ tj
            A[i+1] = A[i];             ---------->        c5        Σ (tj – 1)
            i = i-1;                   ---------->        c6        Σ (tj – 1)
        }
        A[i+1] = key;                  ---------->        c7        n-1
    }
}
```

t_j: ο αριθμός των φορών που ο έλεγχος του while loop εκτελείται για αυτή την τιμή του j.

Ανάλυση του Αλγορίθμου InsertionSort

Συνολικός Χρόνος Εκτέλεσης

$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4 \Sigma\ t_j + c_5 \Sigma\ (t_j-1) + c_6 \Sigma\ (t_j-1) + c_7(n-1)$

- *Ποιος είναι ο καλύτερος χρόνος εκτέλεσης που μπορεί να επιτευχθεί από την InsertionSort?*

Ο πίνακας είναι εξ αρχής ταξινομημένος σε αύξουσα διάταξη.

Τότε, $t_j = 1, \forall\ j = 1,2, \ldots, n-1$:

$$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$
$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

$\Rightarrow$ **γραμμική συνάρτηση του n!**

Ανάλυση του Αλγορίθμου InsertionSort

- *Ποιος είναι ο χειρότερος χρόνος εκτέλεσης κατά την εκτέλεση της InsertionSort?*

  Τα στοιχεία του πίνακα είναι σε φθίνουσα διάταξη. Τότε, $t_j = (j+1)$, $\forall$ j = 1, …, n-1 και:

  - $\Sigma\ t_j = \Sigma\ (j+1)$
    $= 2 + 3 + \ldots + n$
    $= (1+2+ \ldots + n) - 1$
    $= n(n+1)/2 - 1$

  - $\Sigma\ (t_j - 1) = \Sigma\ j$
    $= 1 + 2 + \ldots + (n-1)$
    $= n(n-1)/2$

  Επομένως:

  $$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n(n+1)/2 - 1) + c_5\, n(n-1)/2 + c_6\, n(n-1)/2 + c_7(n-1)$$
  $$= \ldots$$
  $$= (c_4/2 + c_5/2 + c_6/2)\, n^2 + (c_1+c_2+c_3+c_4/2 - c_5/2-c_6/2+c_7)\, n - (c_2+c_3+c_4+c_7)$$

  $\Rightarrow$ τετραγωνική συνάρτηση του n!

Χρονική Πολυπλοκότητα
Χείριστος Χρόνος Εκτέλεσης

Ο χείριστος χρόνος εκτέλεσης (ή χρονική πολυπλοκότητα) ενός αλγορίθμου ορίζεται να είναι ο μέγιστος χρόνος εκτέλεσης για κάθε είσοδο μεγέθους n (και είναι συνάρτηση του n).

- *Πως μπορούμε να βρούμε ένα πάνω όριο στη χρονική πολυπλοκότητα ενός αλγορίθμου;*

- *Πως μπορούμε να βρούμε ένα κάτω όριο στην χρονική πολυπλοκότητα ενός αλγορίθμου;*

Τάξη Χρονικής Πολυπλοκότητας

Η χρονική πολυπλοκότητα της InsertionSorn είναι an^2+bn+c, όπου $a = c_4/2 + c_5/2 + c_6/2$, $b = c_1+c_2+c_3+c_4/2-c_5/2-c_6/2+c_7$ και $c = c_2+c_3+c_4+c_7$.

Τα κόστη c_i δεν μας δίνουν χρήσιμη πληροφορία.

Πολλές φορές, κατά τη μελέτη της χρονικής πολυπλοκότητας, είναι χρήσιμο να θεωρούμε ακόμη πιο απλουστευτικές αφαιρέσεις από την an^2+bn+c.

Μας ενδιαφέρει μόνο ο ρυθμός μεταβολής της συνάρτησης της χρονικής πολυπλοκότητας:

- Από το άθροισμα an^2+bn+c μας ενδιαφέρει μόνο ο κυρίαρχος όρος an^2, αφού οι άλλοι δύο όροι είναι μη-σημαντικοί για μεγάλες τιμές του n.

- Αγνοούμε επίσης το συντελεστή a, αφού οι σταθεροί παράγοντες είναι λιγότεροι σημαντικοί για μεγάλες τιμές του n.

Η χρονική πολυπλοκότητα της InsertionSort είναι τετραγωνικής τάξης.

Τάξη Χρονικής Πολυπλοκότητας

Μερικές φορές μπορούμε να υπολογίσουμε την ακριβή πολυπλοκότητα ενός αλγορίθμου.

Όταν το n (είσοδος) είναι μεγάλο, οι σταθερές και οι χαμηλότερης τάξης όροι δεν είναι σημαντικοί (κυριαρχεί έναντι αυτών ο υψηλότερης τάξης όρος).

Η μελέτη της πολυπλοκότητας για πολύ μεγάλα n (όριο συνάρτησης πολυπλοκότητας όταν το n τείνει στο άπειρο), ονομάζεται ασυμπτωτική μελέτη.

Συνήθως, ένας αλγόριθμος που είναι ασυμπτωτικά ο πιο αποτελεσματικός, είναι η καλύτερη επιλογή για όλες εκτός από πολύ μικρές εισόδους.

Ανάλυση Αλγορίθμων

Συμβολισμός Ο

Έστω f(n) και g(n) δύο συναρτήσεις. Η f(n) είναι O(g(n)), f(n) = O(g(n)), αν υπάρχουν σταθερές $c \in R^+$ (c πραγματικός, c > 0) και ακέραιος $n_0 \geq 0$, έτσι ώστε για κάθε $n \geq n_0$ να ισχύει:

$$0 \leq f(n) \leq cg(n)$$

Παράδειγμα

Έστω $f(n) = an^2 + bn$, όπου a,b θετικές σταθερές.

? $f(n) = O(n^2)$.

Ψάχνουμε για c & n_0 τ.ω.

$$an^2 + bn \leq cn^2, \text{ για κάθε } n \geq n_0$$

$0 \leq (c-a)n^2 - bn \Rightarrow$

$0 \leq n[(c-a)n - b] \Rightarrow$

$(c-a)n - b \geq 0$

Αν επιλέξουμε c = a+1 και οποιοδήποτε $n_0 \geq b$, η ανισότητα $(c-a)n - b \geq 0$ ισχύει.

Συμβολισμός Ω

Έστω f(n) και g(n) δύο συναρτήσεις. Η f(n) είναι Ω(g(n)), f(n) = Ω(g(n)), αν υπάρχουν σταθερές $c \in R^+$ (c πραγματικός, c > 0) και ακέραιος $n_0 \geq 0$, έτσι ώστε για κάθε $n \geq n_0$ να ισχύει:

$$0 \leq cg(n) \leq f(n)$$

Παράδειγμα

Θα αποδείξω ότι f(n) = Ω(n).

$an^2 + bn \geq cn \Rightarrow$

$an^2 + (b-c)n \geq 0 \Rightarrow$

$an + b-c \geq 0.$

Αν επιλέξουμε c = b ισχύει ότι $an \geq 0$, για κάθε $n \geq 0$, άρα για c = b & $n_0 = 0$, ο ισχυρισμός αποδεικνύεται.

Συμβολισμός Θ

Έστω f(n) και g(n) δύο συναρτήσεις. Η f(n) είναι Θ(g(n)), f(n) = Θ(g(n)), αν f(n) = O(g(N)) και f(n) = Ω(g(n)).

Παράδειγμα

Αποδεικνύουμε πως f(n) = $\Omega(n^2)$, οπότε f(n) = $\Theta(n^2)$.

Ψάχνουμε για c > 0 & $n_0 \geq 0$, τ.ω.

$$an^2 + bn \geq cn^2, \text{ για κάθε } n \geq n_0$$

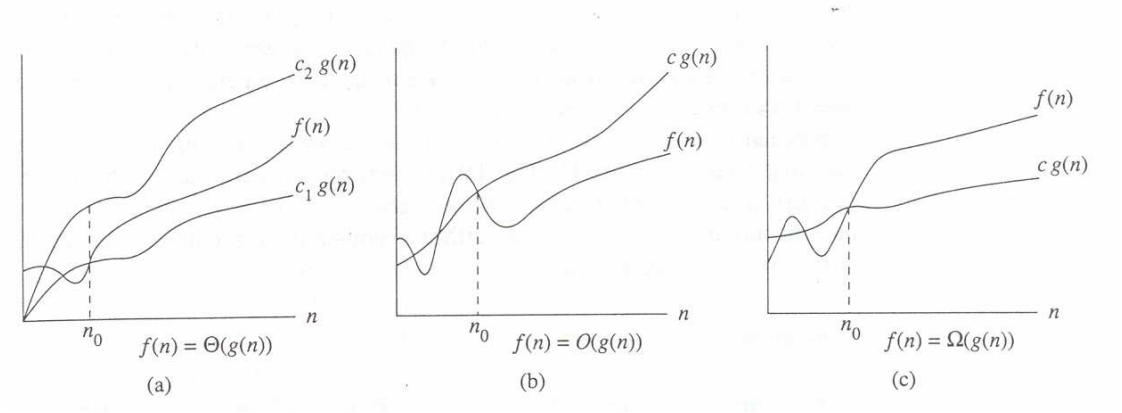
$0 \geq (c-a)n^2 - bn \Rightarrow$

$0 \geq n\,[(c-a)n - b] \Rightarrow$

$(c-a)n - b \leq 0.$

Αν επιλέξουμε c = a/2 > 0 ισχύει ότι (c-a)n-b = -an/2-b ≤ 0, για κάθε n ≥ 0, άρα για c = a/2 & n_0 = 0, ο ισχυρισμός αποδεικνύεται.

Τάξη Πολυπλοκότητας – Συμβολισμοί Ο, Ω, Θ Σχηματικά



Συνήθεις Κατηγορίες Χρονικής Πολυπλοκότητας

O(1): σταθερή πολυπλοκότητα

O(logn): λογαριθμική πολυπλοκότητα

O(n): γραμμική πολυπλοκότητα

O(n logn): πολυπλοκότητα O(n log n) (συνήθης πολυπλοκότητα βέλτιστων αλγόριθμων ταξινόμησης)

O(n^2): τετραγωνική πολυπλοκότητα

O(n^3): κυβική πολυπλοκότητα

O(n^k), για κάποιο προκαθορισμένο ακέραιο k: πολυωνυμική πολυπλοκότητα

O(2^n): εκθετική πολυπλοκότητα

Ιδιότητες Ο, Ω, Θ

Ανακλαστική Ιδιότητα

$f(n) = O(f(n))$

$f(n) = \Omega(f(n))$

$f(n) = \Theta(f(n))$

Μεταβατική Ιδιότητα

$f(n) = O(g(n)) \ \& \ g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

$f(n) = \Omega(g(n)) \ \& \ g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

$f(n) = \Theta(g(n)) \ \& \ g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

Συμμετρική Ιδιότητα

$f(n) = \Theta(g(n))$ αν και μόνο αν $g(n) = \Theta(f(n))$

Αντίστροφη Συμμετρική Ιδιότητα

$f(n) = O(g(n))$ αν και μόνο αν $g(n) = \Omega(f(n))$

Καλή άσκηση για το σπίτι

Αποδείξτε τις παραπάνω ιδιότητες.

Απλές Αναδρομικές Σχέσεις

Έστω ότι η f(n) αναπαριστά το άθροισμα των n μικρότερων ακεραίων. Τότε, η f(n) μπορεί να οριστεί αναδρομικά ως εξής:

$f(n) = 1$, *αν* $n = 1$ και $f(n) = f(n-1) + n$, *αν* $n > 1$.

Ο υπολογισμός μιας μη-αναδρομικής φόρμας για την f(n) είναι απλός:

$f(n) = f(n-1) + n$

$= f(n-2) + (n-1) + n$

$= ....$

$= f(1) + 2 + ... + (n-1) + n$

$= 1 + 2 + ... + (n-1) + n$

$= (n+1)\, n / 2$

Παράδειγμα 2

Έστω ότι $f(0) = 1$ και $f(n) = 2f(n-1)$.

Ο υπολογισμός μιας μη-αναδρομικής φόρμας για την f(n) είναι και πάλι απλός:

$f(n) = 2\, f(n-1)$

$= 2^2\, f(n-2)$

$= ...$

$= 2^n\, f(0)$

$= 2^n$.

Αναδρομές τύπου «Διαίρει και Βασίλευε»

Δυαδική Αναζήτηση

int *BinarySearch*(**table** T[a...b], **key** K)

/ επιστρέφει τη θέση του K μέσα στον ταξινομημένο πίνακα T, αν το K υπάρχει, και -1 διαφορετικά */*

```
  int middle, tmp;

  if (a > b) return -1;
  middle = κάτω ακέραιο μέρος του (a+b)/2;
  if (K == T[middle]) return middle;
  else if (K < T[middle]) {
        tmp = BinarySearch(T[a...middle-1], K);
        return tmp;
  }
  else {
         tmp = BinarySearch(T[middle+1...b],K);
         return tmp;
  }
}
```

- Κάθε μια από τις αναδρομικές κλήσεις εξετάζει περίπου το μισό πίνακα.

- Έστω ότι *n* είναι το μέγεθος του πίνακα (*n=b-a+1*). Ας υποθέσουμε για λόγους απλότητας ότι $n = 2^k - 1$.

Αναδρομές τύπου «Διαίρει και Βασίλευε»

Τότε:

$$2^k - 1 = b - a + 1 \Rightarrow b = 2^k - 2 + a$$

Οπότε, κατά την 1η κλήση της BinarySearch():

$$middle = [(a+b)/2] = [(a+2^k - 2 + a)/2] = a + 2^{k-1} - 1$$

Η 1η αναδρομική κλήση αναφέρεται σε πίνακα μήκους:

$$middle - 1 - a + 1 = middle - a$$
$$= a + 2^{k-1} - 1 - a$$
$$= 2^{k-1} - 1.$$

Η 2η αναδρομική κλήση αναφέρεται σε πίνακα μήκους:

$$b - (middle + 1) + 1 = b - middle$$
$$= 2^k - 2 + a - a - 2^{k-1} + 1$$
$$= 2^{k-1} - 1.$$

- *Ποιο είναι το χειρότερο σενάριο που μπορεί να συμβεί κατά την εκτέλεση;*

  Ο πίνακας δεν περιέχει το κλειδί και οι αναδρομικές κλήσεις συνεχίζονται μέχρι να γίνει κλήση σε άδειο πίνακα.

Αναδρομές τύπου «Διαίρει και Βασίλευε»

Αν T(n) : μέγιστος δυνατός χρόνος εκτέλεσης σε πίνακα με n στοιχεία, και c,d: σταθερές

Τότε:

$$T(2^k-1) = c, \text{ αν } k = 0$$

και

$$T(2^k-1) = d + T(2^{k-1}-1), \text{ αν } k > 0.$$

Έχουμε:

$$\begin{aligned} T(2^k-1) &= d + T(2^{k-1}-1) \\ &= 2d + T(2^{k-2}-1) \\ &= \ldots \\ &= kd + T(0) \\ &= kd + c. \end{aligned}$$

Αφού $n = 2^k-1$, είναι $k = \log(n+1)$ και

$$T(n) = d \log(n+1) + c.$$

ΕΝΟΤΗΤΑ 2
ΠΙΝΑΚΕΣ

Πίνακες

Δομή δεδομένων που αποθηκεύει μια ακολουθία από διαδοχικά αριθμημένα αντικείμενα.

Πιο φορμαλιστικά:

✓ l,u: ακέραιοι

➢ Το διάστημα l…u είναι το σύνολο των ακεραίων j τ.ω. $l \leq j \leq u$.

➢ Ένας πίνακας είναι μια συνάρτηση με πεδίο ορισμού ένα διάστημα, που λέγεται σύνολο δεικτών, και πεδίο τιμών ένα σύνολο από αντικείμενα (τα στοιχεία του πίνακα).

✓ X: πίνακας

✓ j: δείκτης στο σύνολο δεικτών του X

✓ X[j]: το j-οστό στοιχείο του X

Λειτουργίες σε πίνακες

- *Access(X,j):* επιστρέφει X[j]
- *Length(X):* Επιστρέφει u, τον αριθμό των στοιχείων του X
- *Assign(X,j,a):* εκτελεί τη λειτουργία X[j] = a
- *Initialize(X,a):* Αρχικοποιεί σε a κάθε στοιχείο του X
- *Iterate(X,f):* Εφαρμόζει την συνάρτηση f σε κάθε ένα στοιχείο του X, ξεκινώντας από το μικρότερο και καταλήγοντας στο μεγαλύτερο.

Πολυδιάστατοι Πίνακες

- Ένας διδιάστατος πίνακας ορίζεται ως ο μονοδιάστατος πίνακας του οποίου τα στοιχεία είναι μονοδιάστατοι πίνακες.
- Ένας d-διάστατος πίνακας ορίζεται ως ο μονοδιάστατος πίνακας του οποίου τα στοιχεία είναι (d-1)-διάστατοι πίνακες.
- Αν $<j_1, \ldots, j_d>$ είναι ένα διάνυσμα d ακεραίων που ανήκει στο σύνολο δεικτών ενός d-διάστατου πίνακα X, τότε $X[j_1,\ldots,j_d]$ είναι η τιμή του πίνακα για το συγκεκριμένο δείκτη.

Υλοποίηση Πινάκων σε Διαδοχικές Θέσεις Μνήμης

- Στη C το όνομα, π.χ. X, ενός πίνακα είναι και δείκτης στο 1ο στοιχείο του.
- Αν L είναι το μέγεθος κάθε στοιχείου του πίνακα, τότε το j-οστό στοιχείο του πίνακα βρίσκεται στη διεύθυνση X + L*(j-1).
- Κάποιες φορές το μήκος του πίνακα δεν είναι γνωστό και αντί αυτού χρησιμοποιείται ένα στοιχείο φρουρός το οποίο σηματοδοτεί το τέλος του πίνακα (αυτό π.χ. γίνεται στη C με τα strings).

Διδιάστατοι Πίνακες

Στη C οι πίνακες αποθηκεύονται κατά γραμμές:

X[0,0], X[0,1], ... , X[0, m], X[1,0], ... , X[1,m], ... , X[n,0], ... , X[n,m].

	0	**1**	**...**	**m**
0	[0,0]	[0,1]	…	[0,m]
1	[1,0]	[1,1]	…	[1,m]
...	…	…	…	…
n	[n,0]	[n,1]	…	[n,m]

Άρα το στοιχείο X[k,j] βρίσκεται στη θέση μνήμης X + L*(m*k + j). *Γιατί?*

Ειδικές Μορφές Πινάκων

Συμμετρικοί Πίνακες

Συμμετρικός λέγεται ο τετραγωνικός πίνακας με στοιχεία X[k,j] = X[j,k], για κάθε $0 \le j,k \le n$.

	0	**1**	**2**
0	5	34	31
1	34	6	87
2	31	87	45

Υπάρχουν 1 + 2 + ... + n διαφορετικά στοιχεία και άρα απαιτούνται (1+n)*n/ 2 θέσεις μνήμης για την αποθήκευση του πίνακα.

Τριγωνικοί Πίνακες

Για τα στοιχεία ενός πάνω (κάτω) τριγωνικού πίνακα ισχύει X[k,j] = 0, αν k < j (αντίστοιχα k > j), όπου $1 \le k,j \le n$.

1	0	0	0
21	43	0	0
34	7	18	0
5	86	23	43

Πως μπορούμε να υλοποιήσουμε τριδιαγώνιους πίνακες χωρίς μεγάλη σπατάλη μνήμης?

Αραιοί Πίνακες

Ένας πίνακας λέγεται αραιός, όταν ένα μεγάλο ποσοστό των στοιχείων του έχουν την τιμή 0.

Παράδειγμα

0	7	0	0	0
1	2	0	0	-3
0	0	4	0	0
12	0	0	0	0

Τρόπος Αποθήκευσης

Αποθήκευση του αντίστοιχου δυαδικού πίνακα:

0	1	0	0	0
1	1	0	0	1
0	0	1	0	0
1	0	0	0	0

ακολουθούμενο από ένα μονοδιάστατο πίνακα με τιμές:

(7 1 2 -3 4 12)

ΕΝΟΤΗΤΑ 3
ΓΡΑΜΜΙΚΕΣ ΛΙΣΤΕΣ

Γραμμικές Λίστες

Ορισμός

Γραμμική λίστα (linear list) είναι ένα σύνολο από $n \geq 0$ κόμβους $L_0, L_1, ..., L_{n-1}$, όπου το στοιχείο L_0 είναι το πρώτο στοιχείο (ή ο πρώτος κόμβος), ενώ το στοιχείο L_k προηγείται του στοιχείου L_{k+1} και έπεται του στοιχείου L_{k-1}, $0 < k < n-1$.

- L_0: κεφαλή (head)
- L_{n-1}: ουρά (tail)
- $|L|$: μήκος λίστας ($|L| = n$)
- $< >$: κενή λίστα

Λειτουργίες που συνήθως υποστηρίζονται από λίστες:

- *Access(L,j):* Επιστρέφει L_j ή ένα μήνυμα λάθους αν j είναι < 0 ή $> |L|-1$.
- *Length(L):* Επιστρέφει $|L|$.
- *Concat(L_1,L_2):* επιστρέφει μια λίστα: το αποτέλεσμα της συνένωσης των 2 λιστών σε 1.
- *MakeEmptyList():* επιστρέφει $< >$.
- *IsEmptyList(L):* επιστρέφει true αν $L = < >$, false διαφορετικά.

Είδη Γραμμικών Λιστών

Σειριακή Λίστα**:** καταλαμβάνει συνεχόμενες θέσεις κύριας μνήμης

Συνδεδεμένη Λίστα: οι κόμβοι βρίσκονται σε απομακρυσμένες θέσεις συνδεδεμένες όμως μεταξύ τους με δείκτες.

Στατικές Λίστες: ο μέγιστος αριθμός στοιχείων είναι εξ αρχής γνωστός (υλοποίηση με σειριακές λίστες).

Δυναμικές Λίστες: ο μέγιστος αριθμός στοιχείων δεν είναι γνωστός. Επιτρέπεται η επέκταση ή η συρρίκνωση της λίστας κατά την εκτέλεση του προγράμματος (υλοποίηση με συνδεδεμένες λίστες).

- Στοίβα
- Ουρά

Αφηρημένος τύπος δεδομένων Στοίβα (Stack)

Λίστα που μπορεί να τροποποιείται μόνο με την εισαγωγή και διαγραφή στοιχείων στο ένα της άκρο.

Λειτουργίες

Top(S): επιστρέφει το κορυφαίο στοιχείο της S (αντίστοιχο Access(S, |S|-1)).

Pop(S): (λειτουργία απώθησης) διαγραφή και επιστροφή του κορυφαίου στοιχείου της S

Push(x,S): (λειτουργία ώθησης) εισαγωγή του στοιχείου x στην κορυφή της στοίβας

MakeEmptyStack(): επιστρέφει την < >.

IsEmptyStack(S): επιστρέφει true αν η |S| = 0, διαφορετικά false.

Η μέθοδος επεξεργασίας των δεδομένων στοίβας λέγεται **«Τελευταίο Μέσα – Πρώτο Έξω» (Last In – First Out, LIFO).**

Αφηρημένος τύπος δεδομένων Ουρά (Queue)

Λίστα που μπορεί να τροποποιείται μόνο με την εισαγωγή στοιχείων στο ένα άκρο της και τη διαγραφή στοιχείων στο άλλο.

Λειτουργίες

Enqueue(x,Q): Εισαγωγή x στο τέλος της Q (αντίστοιχο Concat(Q, <x>)).

Dequeue(Q): Διαγραφή & επιστροφή του πρώτου στοιχείου της Q (Q = < Q_0, …, $Q_{|Q|-1}$> και το Q_0 επιστρέφεται).

Front(Q): επιστρέφει Q_0.

MakeEmptyQueue(): επιστρέφει < >.

IsEmptyQueue(Q): επιστρέφει true αν |Q| = 0, false διαφορετικά.

Η μέθοδος επεξεργασίας των δεδομένων ουράς λέγεται **«Πρώτο Μέσα – Πρώτο Έξω» (First In – First Out, FIFO).**

Σειριακές Γραμμικές Λίστες

Στατικές Στοίβες

Μια στατική στοίβα υλοποιείται με τη χρήση ενός μονοδιάστατου πίνακα.

- $S = <S_0, \ldots, S_{n-1}>$ η στοίβα & A[0 … N-1] ο πίνακας, $n \leq N$
- $A[j] = S_j$
- Η στοίβα καταλαμβάνει το κομμάτι A[0 … n-1].
- A[n-1]: κορυφαίο στοιχείο της στοίβας
- A[0]: βαθύτερο στοιχείο της στοίβας

Η στοίβα υλοποιείται ως μια δομή (struct στη C) με πεδία τον πίνακα Infos και τον ακέραιο Length (μέγεθος στοίβας).

S: δείκτης σε στοίβα

info: τύπος στοιχείων του πίνακα (Infos(S)).

S->Length == 0: άδεια στοίβα

S->Length == N: γεμάτη στοίβα

Πιο απλά (σε C), η στοίβα μπορεί να υλοποιηθεί από έναν ακέραιο length και από έναν πίνακα Infos (και όχι ως δομή που περιέχει αυτά τα 2 πεδία).

int Length; info Infos[0..N-1];

Στην απλούστερη έκδοση οι 2 αυτές μεταβλητές είναι global. Ωστόσο, ΔΕΝ ΣΥΝΙΣΤΑΤΑΙ!

Υλοποίηση Λειτουργιών Στοίβας: Απλά

void *MakeEmptyStack(void)*

```
Length = 0;
for (i=0; i < N; i++) Infos[i] = ∅;  /* initialize */
```

boolean *IsEmptyStack(void)*

```
/* return (Length == 0) */
if (Length == 0) return 1;
else return 0;
```

info *Top(void)*

```
if (IsEmptyStack()) then error;
else (return(Infos[Length – 1]));
```

Χρονική Πολυπλοκότητα

MakeEmptyStack(): Θ(N)

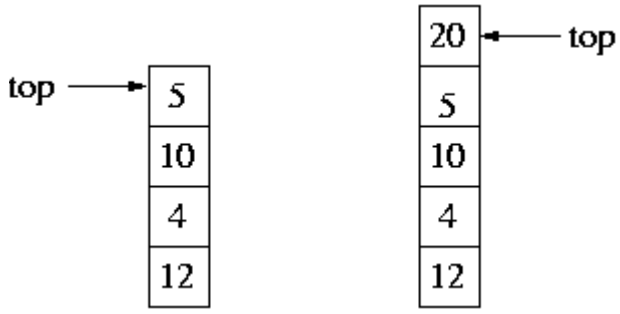
IsEmptyStack(): Θ(1)

Top(): Θ(1)

Συνολικός Απαιτούμενος Χώρος Μνήμης

Ανεξάρτητα από τον αριθμό στοιχείων: N

Υλοποίηση Λειτουργιών Στοίβας: Απλά



info *Pop(void)*

```
if (Length == 0) return error
else
    x = Top();
    Length = Length –1;
    return x;
```

void *Push(info x)*

```
if (Length == N) then error
else
    Length = Length + 1;
    Infos[Length-1] = x;
```

Χρονική Πολυπλοκότητα

Pop(): Θ(1)

Push(): Θ(1)

Υλοποίηση Λειτουργιών Στοίβας
Πιο δύσκολα

pointer *MakeEmptyStack(void)*

```
pointer S;                /* temporary pointer */
S = newcell(Stack);       /* malloc() */
S->Length = 0;
return S;
```

boolean *IsEmptyStack(pointer S)*

```
/* return (S->Length == 0) */
if (S->Length == 0) return 1;
else return 0;
```

info *Top(pointer S)*

```
if (IsEmptyStack(S)) then error;
else (return(S->Infos[S->Length – 1]));
```

Χρονική Πολυπλοκότητα

MakeEmptyStack(): Θ(1)

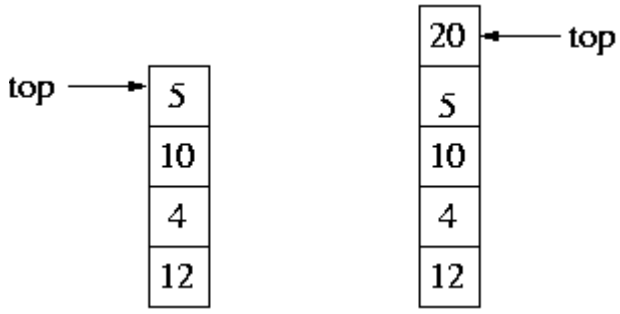
IsEmptyStack(): Θ(1)

Top(): Θ(1)

Συνολικός Απαιτούμενος Χώρος Μνήμης

Ανεξάρτητα από τον αριθμό στοιχείων: N

Υλοποίηση Λειτουργιών Στοίβας



info *Pop(pointer S)*

```
if (S->Length == 0) return error
else
    x = Top(S);
    S->Length = S->Length –1;
    return x;
```

void *Push(info x, pointer S)*

```
if (S->Length == N) then error
else
    S->Length = S->Length + 1;
    S->Infos[S->Length-1] = x;
```

Χρονική Πολυπλοκότητα

Pop(): Θ(1)

Push(): Θ(1)

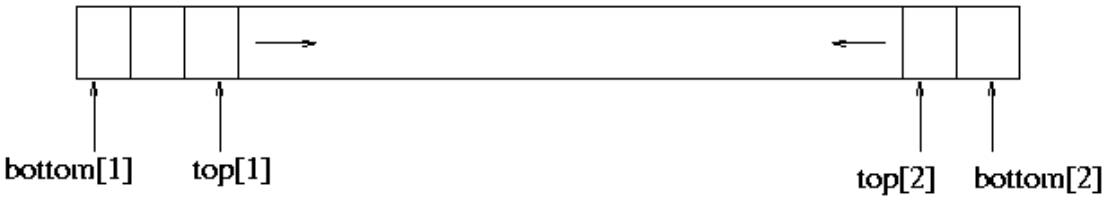
Πολλαπλή Στοίβα

Περισσότερες από μια στοίβες που υλοποιούνται σε συνεχόμενες θέσεις μνήμης.

Παράδειγμα 1: Δύο Στοίβες

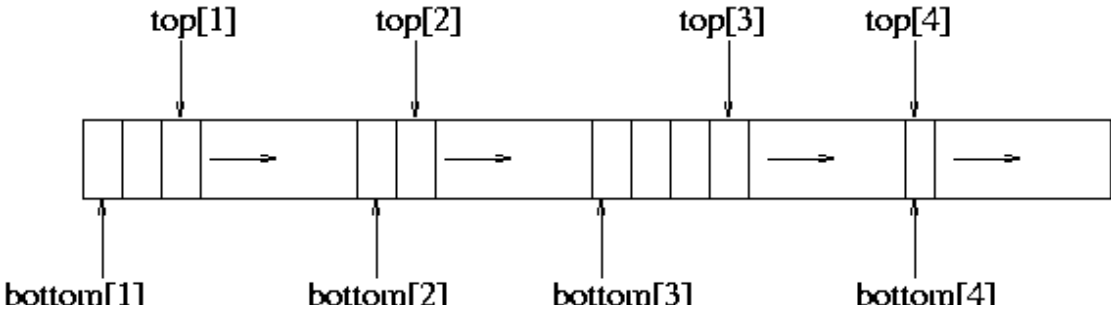
Έστω Stack[0…n-1] ο πίνακας που χρησιμοποιείται για την αποθήκευση των λιστών.

Η 1η στοίβα ξεκινάει από τη θέση Stack[0] και αναπτύσσεται προς τα δεξιά, ενώ η 2η ξεκινάει από τη θέση Stack[n-1] και αναπτύσσεται προς τα αριστερά.



Παράδειγμα 2: n Στοίβες

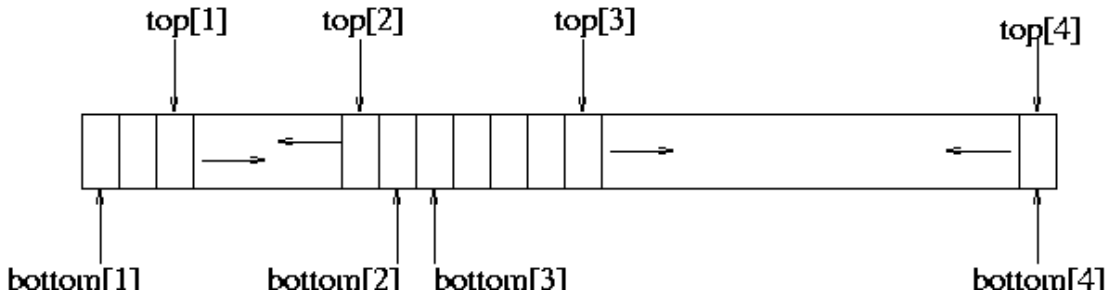
Η πολλαπλή στοίβα χωρίζεται σε n ίσα τμήματα.



Τι γίνεται σε περίπτωση υπερχείλισης?

Καλύτερες Υλοποιήσεις n-Πολλαπλής Στοίβας

Δικατευθυνόμενη Πολλαπλή Στοίβα



Εφαρμογές Σειριακών Γραμμικών Στοιβών

Αναδρομή

Υπολογισμός του n!

Αναδρομική Λύση

```
integer factorial(integer n)
    if (n == 0) then return 1;
    else return (n * factorial(n-1));
```

Μη Αναδρομική Λύση

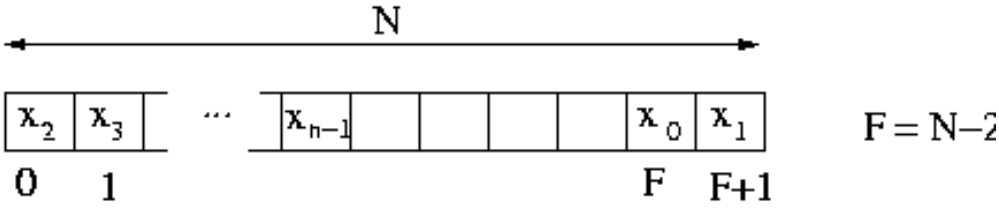
```
integer factorial(integer n)
    integer j, product;

    j = n;
    product = 1;
    while (j > 0)
        product = j * product;
        j = j-1;
    return product;
```

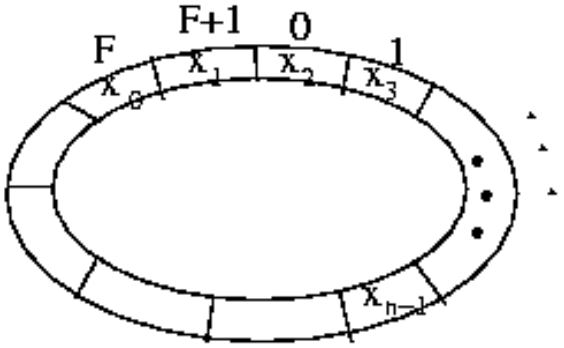
Στατική Ουρά

Q: **ουρά**, δομή με τρία πεδία:

- A=Q->Infos: πίνακας με στοιχεία
- F=Q->Front: θέση πρώτου στοιχείου
- n=Q->Length: συνολικός αριθμός στοιχείων.

Κυκλική Στατική Ουρά





- $x_0, \ldots, x_{n-1}$: στοιχεία ουράς
- A[F], A[(F+1) mod N], A[(F+2) mod N] , … , A[(F+n-1) mod N]: θέσεις στις οποίες είναι αποθηκευμένα τα $x_0, \ldots, x_{n-1}$.

Υλοποίηση Λειτουργιών Κυκλικής Ουράς

pointer *MakeEmptyQueue(void)*

```
pointer Q;            /* temporary pointer */
Q = NewCell(Queue);  /* malloc() */
Q->Front = 0;
Q->Length = 0;
return Q;
```

boolean *IsEmptyQueue(pointer Q)*

```
return (Q->Length == 0);
```

info *Front(pointer Q)*

```
If IsEmptyQueue(Q) then error;
else return (Q->Infos[Q->Front]);
```

Πολυπλοκότητα

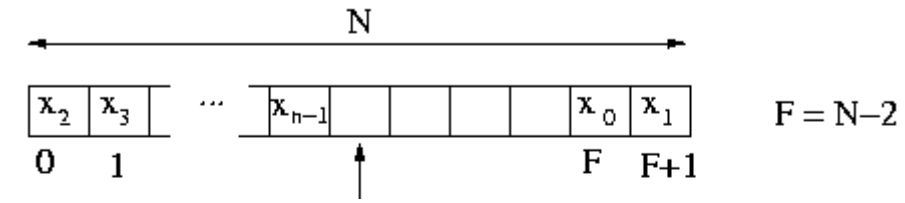
Ίδια με την υλοποίηση στοίβας

Χρόνος εκτέλεσης κάθε λειτουργίας: Θ(1)

Χρησιμοποιούμενος χώρος μνήμης: N

Υλοποίηση Λειτουργιών Κυκλικής Ουράς

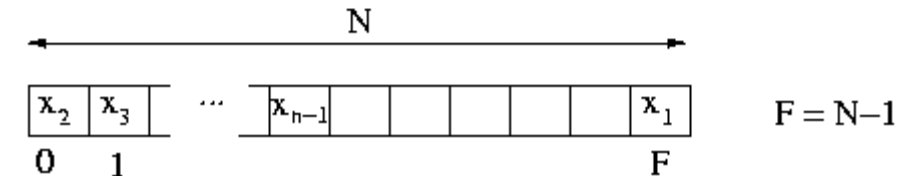
Εισαγωγη σε κυκλικη ουρα



(Q->Front + Q->Length –1) mod N

(F + n) mod N

Εξαγωγη απο κυκλικη ουρα



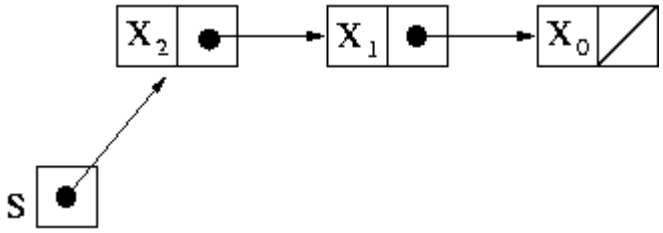
info *Dequeue(pointer L)*

```
if IsEmptyQueue(Q) then error;
else
    x = Q->Infos[Q->Front];
    Q->Front = (Q->Front+1) mod N;
    Q->Length = Q->Length –1;
    return x;
```

procedure *Enqueue(info x, pointer Q)*

```
if (Q->Length == N) then error;
else
   Q->Length = Q->Length+1
   Q->Infos[(Q->Front + Q->Length –1) mod N] = x
```

Συνδεδεμένες Γραμμικές Λίστες
Στοίβα ως Συνδεδεμένη Λίστα



S: δείκτης σε δομή (που ονομάζεται Node) με πεδία :

- next: δείκτης στο επομένο στοιχείο της στοίβας
- data: πληροφορία αποθηκευμένη στο στοιχείο

Υλοποίηση Λειτουργιών

pointer *MakeEmptyStack()*

```
return NULL;
```

boolean *IsEmptyStack(pointer S)*

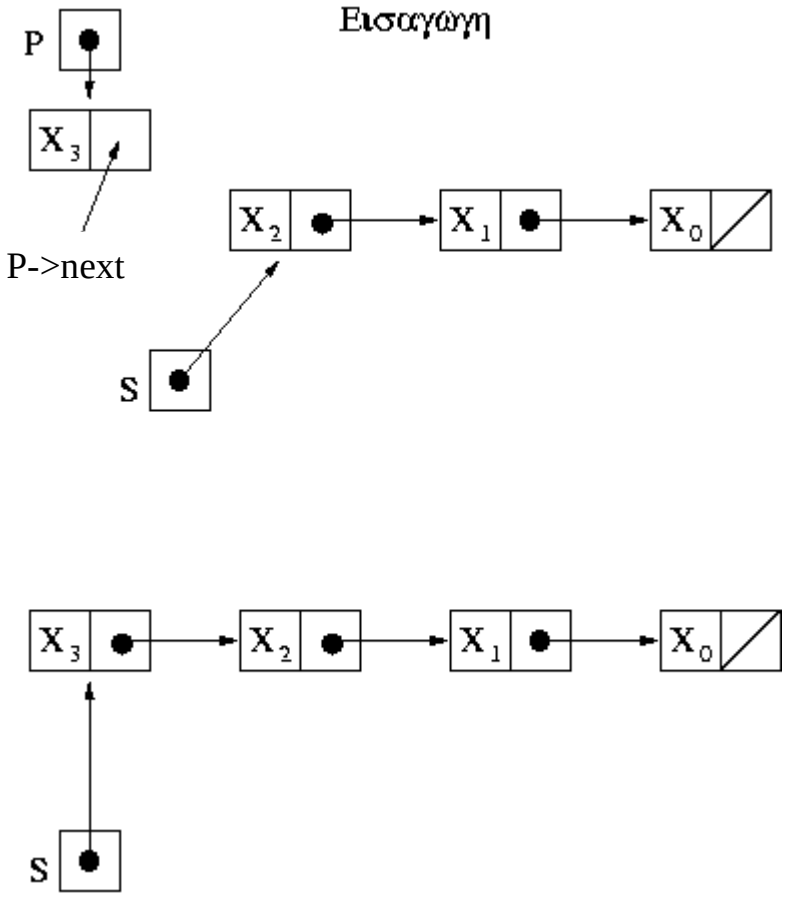
```
return (S == NULL);
```

info *Top(pointer S)*

```
if IsEmptyStack(S) then error;
else return S->data;
```

Χρόνος εκτέλεσης κάθε λειτουργίας: Θ(1)

Υλοποίηση Λειτουργιών Συνδεδεμένης Στοίβας



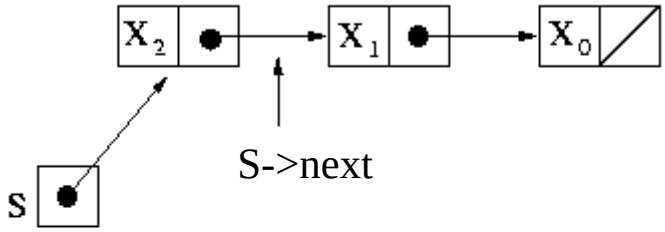
void *Push(info x, pointer S)*

```
pointer P;              /* temporary pointer */
P = NewCell(Node);      /* malloc() */
P->data = x;
P->next = S;
S = P;
```

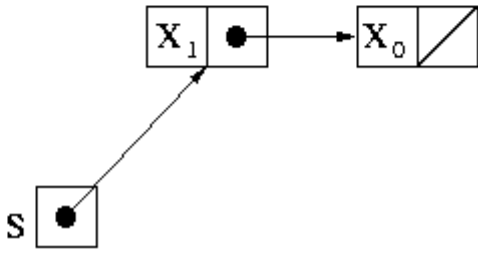
Απαιτούμενος χρόνος: Θ(1)

Υλοποίηση Λειτουργιών Συνδεδεμένης Στοίβας

Διαγραφη



S=S->Next



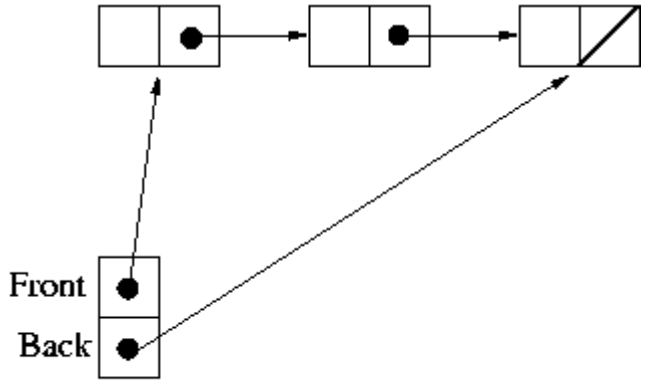
info *Pop(pointer S)*

```
if (IsEmptyStack(S)) then error;
else
    x = Top(S);
    S = S->Next;
    return x;
```

Απαιτούμενος Χρόνος: Θ(1)

Extra μνήμη (για δείκτες): n (όπου n: # στοιχείων)

Ουρά ως Συνδεδεμένη Λίστα: Απλά



Ουρά: 2 δείκτες (Front και Back) που δείχνουν σε δομή (με όνομα Node) 2 πεδίων:

- next: δείκτης στο επομένο στοιχείο της στοίβας
- data: πληροφορία αποθηκευμένη στο στοιχείο

Στην απλούστερη περίπτωση οι Front, Back είναι global (ΔΕΝ ΣΥΝΙΣΤΑΤΑΙ!).

Υλοποίηση Λειτουργιών Ουράς

void *MakeEmptyQueue(void)*

```
    Front = Back = NULL;
```

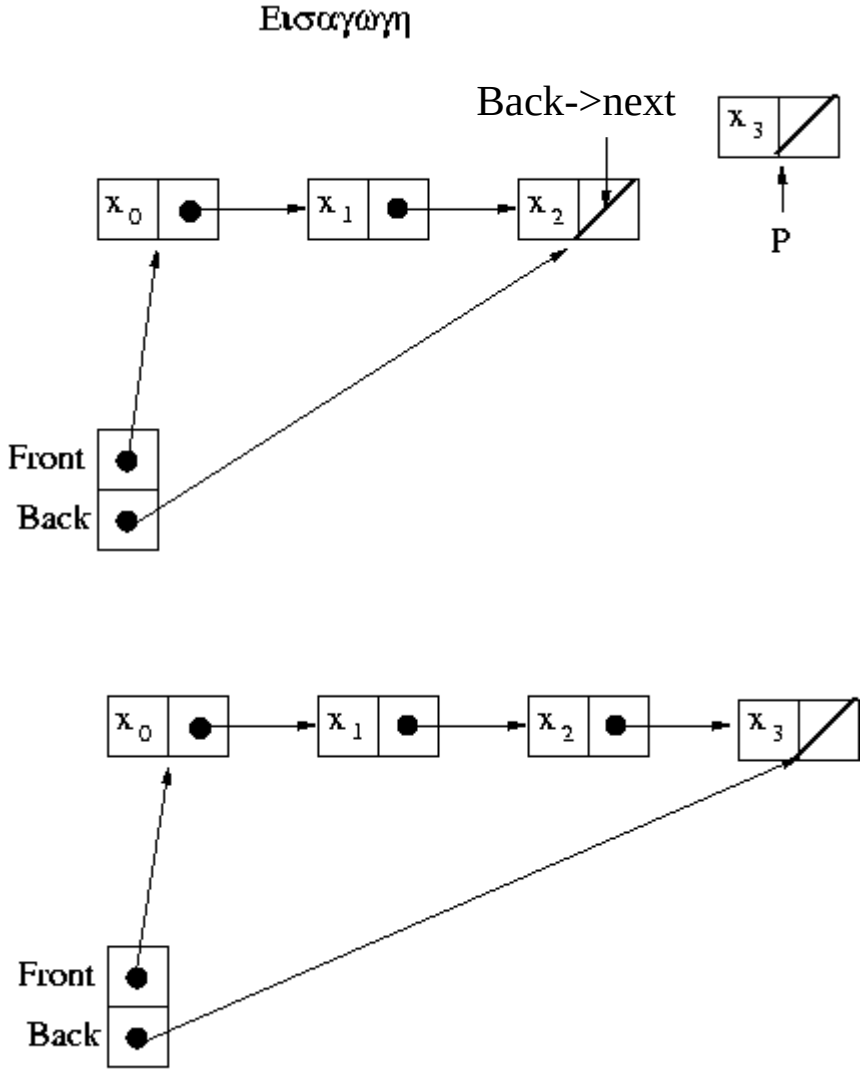
boolean *IsEmptyQueue(void)*

```
    return (Front == NULL);
```

info *Front(void)*

```
    if (IsEmptyQueue()) then error;
    else return (Front->data);
```

Υλοποίηση Λειτουργιών Ουράς: Απλά

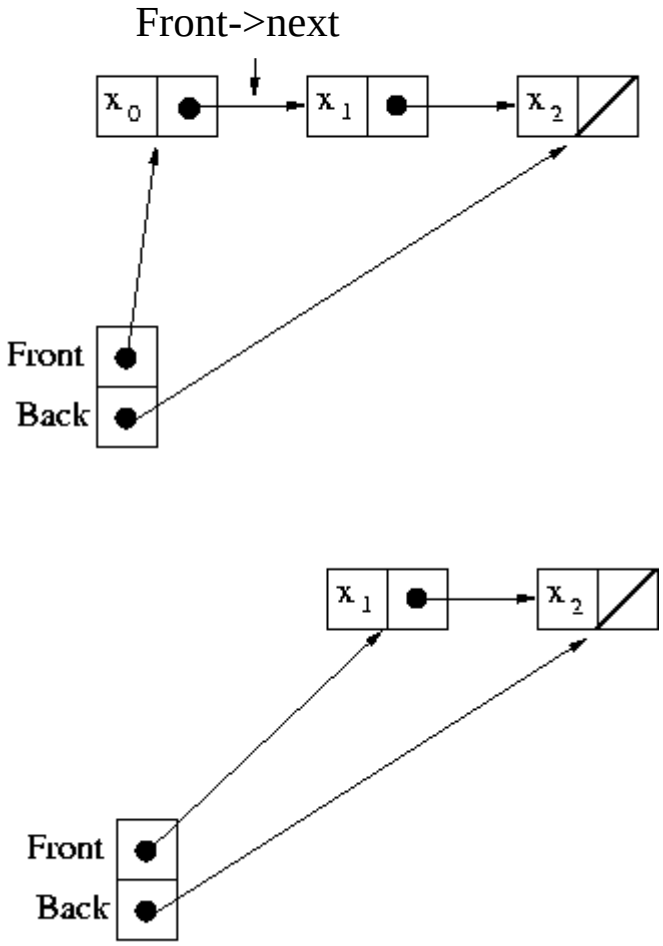


void *Enqueue(info x)*

```
pointer P;        /* temporary pointer */
P = NewCell(Node);
P->data = x;
P->next = NULL;
if (IsEmptyQueue()) then Front = P;
else Back->next = P;
Back = P;
```

Υλοποίηση Λειτουργιών Ουράς: Απλά

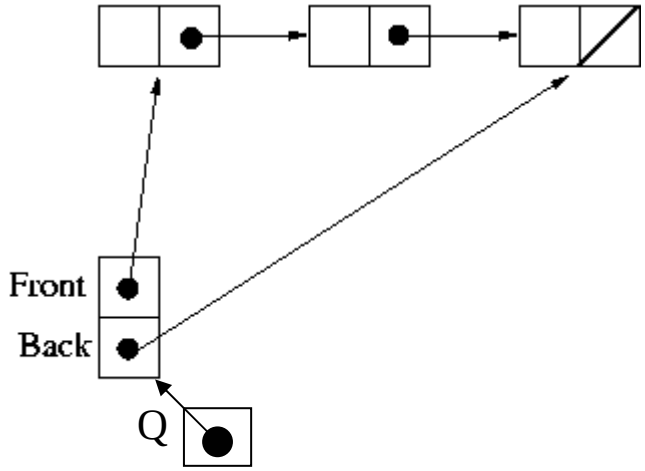
Διαγραφη



info *Dequeue(void)*

```
if (IsEmptyQueue()) then error;
else
    x = Front->data;
    Front = Front->next;
    if  (Front == NULL) then
        Back = NULL;
    return x;
```

Ουρά ως Συνδεδεμένη Λίστα: Πιο δύσκολα



Ουρά Q: δομή (που ονομάζεται Queue) 2 δεικτών (Q->Front και Q->Back) που κάθε ένας δείχνει σε δομή (που ονομάζεται Node) 2 πεδίων:

- next: δείκτης στο επόμενο στοιχείο της στοίβας
- data: πληροφορία αποθηκευμένη στο στοιχείο

Υλοποίηση Λειτουργιών Ουράς

pointer *MakeEmptyQueue(void)*

```
pointer Q;          /* temporary pointer */
Q = NewCell(Queue);     /* malloc() */
Q->Front = Q->Back = NULL;
return Q;
```

boolean *IsEmptyQueue(pointer Q)*

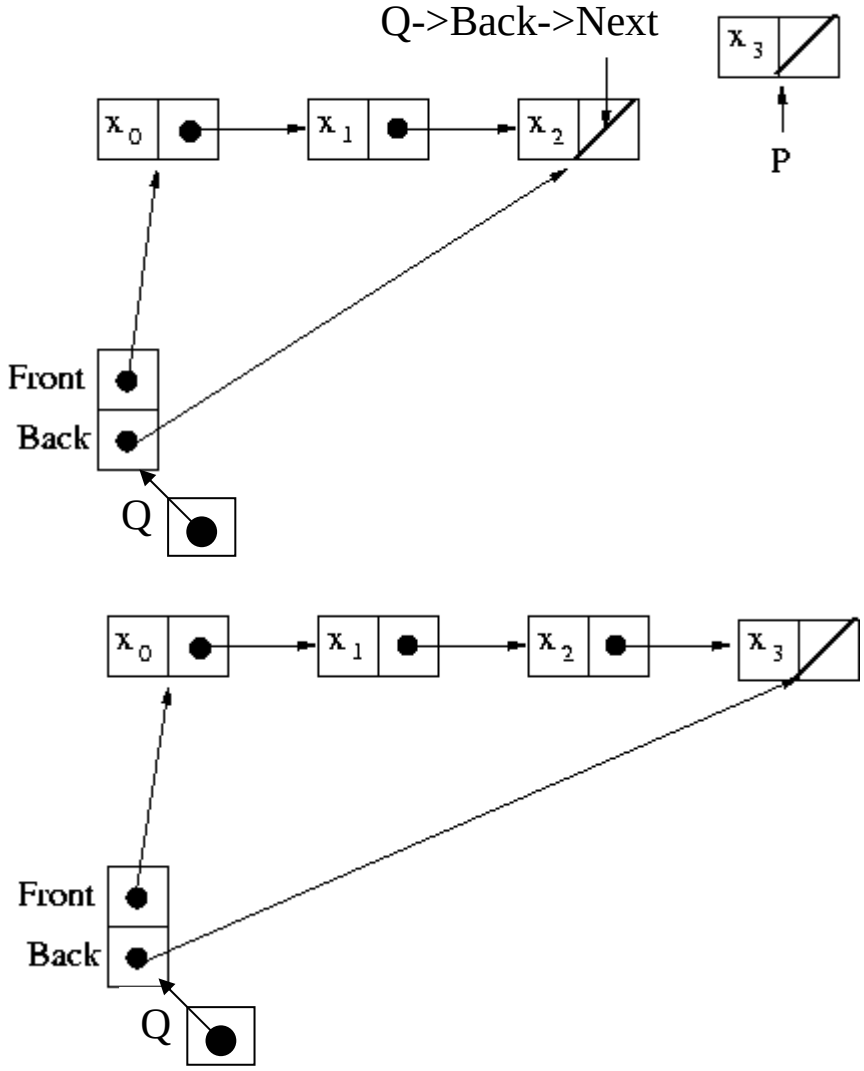
```
return (Q->Front == NULL);
```

info *Front(pointer Q)*

```
if (IsEmptyQueue(Q)) then error;
else return (Q->Front->data);
```

Υλοποίηση Λειτουργιών Ουράς
Πιο δύσκολα

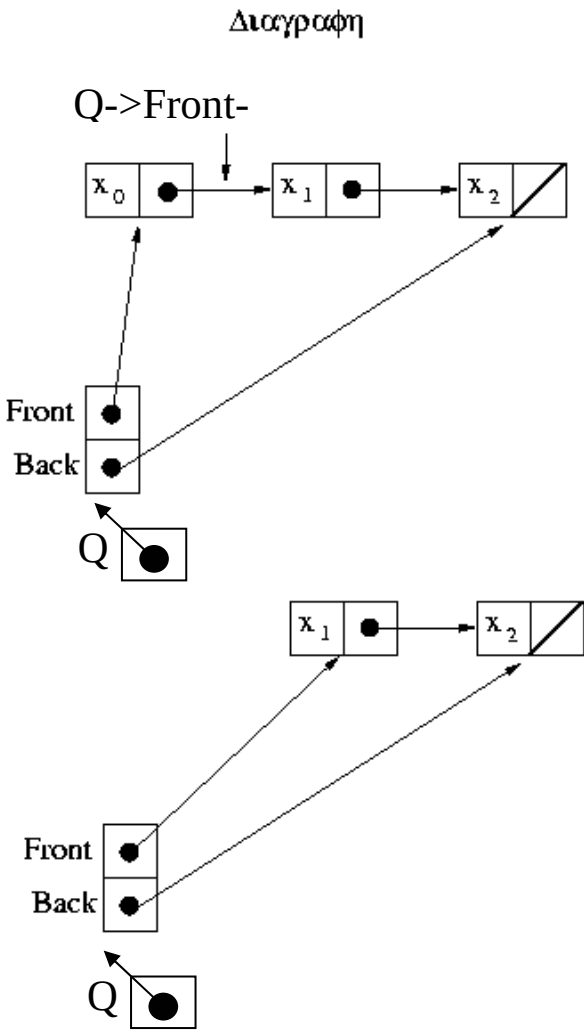
Εισαγωγη



void *Enqueue(info x, pointer Q)*

```
pointer P;        /* temporary pointer */
P = NewCell(Node);
P->data = x;
P->next = NULL;
if (IsEmptyQueue(Q)) then Q->Front = P;
else Q->Back->next = P;
Q->Back = P;
```

Υλοποίηση Λειτουργιών Ουράς
Πιο Δύσκολα



info *Dequeue(pointer Q)*

```
if (IsEmptyQueue(Q)) then error;
else
    x = Q->Front->data;
    Q->Front = Q->Front->next;
    if  (Q->Front == NULL) then
        Q->Back = NULL;
    return x;
```

Πολυπλοκότητα: Ίδια με εκείνη για στοίβες.

Παράδειγμα Εφαρμογής Συνδεδεμένων Λιστών

Παράσταση πολυωνύμων

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

Χρήση λίστας για την αποθήκευση των συντελεστών.

Κόμβος Φρουρός

Αναζήτηση κόμβου με συγκεκριμένη τιμή. Αν ο κόμβος δεν υπάρχει στη λίστα εισάγεται.

Εισάγεται ένας κόμβος (τελευταίος πάντα στη λίστα) που λέγεται κόμβος φρουρός.
Ένας δείκτης δείχνει μόνιμα σε αυτόν τον κόμβο.

Η προς αναζήτηση τιμή αρχικά αποθηκεύεται στον κόμβο αυτό.

Εκτελείται διάσχιση της λίστας και αναζήτηση της τιμής. Αν βρεθεί στον κόμβο φρουρό γίνεται η εισαγωγή. Διαφορετικά όχι.

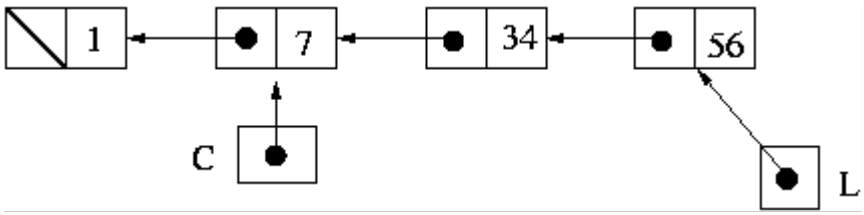
Σε τι μας βοηθάει ο κόμβος φρουρός?

2ο Παράδειγμα Εφαρμογής Συνδεδεμένων Λιστών

Αραιοί Πίνακες

- Αποθήκευση των μη-μηδενικών στοιχείων σε μια λίστα.
- Κάθε στοιχείο της λίστας αντιστοιχεί σε ένα μη-μηδενικό στοιχείο του πίνακα.
- Προσπέλαση ενός στοιχείου απαιτεί χρόνο ανάλογο του αριθμού των μη-μηδενικών στοιχείων στον πίνακα.
- Εισαγωγές στοιχείων είναι ωστόσο δυνατές (δηλαδή ενός μηδενικού στοιχείου του πίνακα σε μη-μηδενικό στοιχείο είναι δυνατές).

Εισαγωγή Στοιχείου σε Ταξινομημένη Λίστα



Ο κάθε κόμβος της λίστας περιέχει π.χ. έναν ακέραιο και ένα δείκτη στον επόμενο κόμβο:
num: αποθηκευμένος ακέραιος στον κόμβο
next: δείκτης στον επόμενο κόμβο της λίστας

L: δείκτης στο πρώτο στοιχείο της λίστας

Πρόβλημα προς επίλυση
Εισαγωγή νέου στοιχείου στη λίστα, έτσι ώστε η λίστα να εξακολουθήσει να είναι ταξινομημένη.

K: προς εισαγωγή ακέραιος

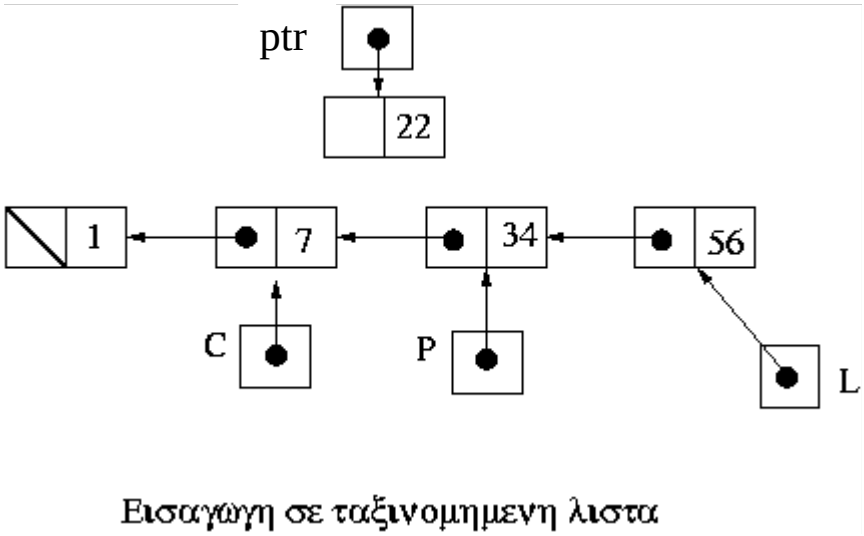
Πρόβλημα με την εισαγωγή στοιχείου σε ταξινομημένη λίστα:

*Είναι δυνατή η εισαγωγή ενός στοιχείου μόνο ως **επόμενου** κόμβου κάποιου δεδομένου κόμβου και όχι ως προηγούμενου.*

Εισαγωγή Στοιχείου σε Ταξινομημένη Λίστα

Λύση

Χρήση ενός βοηθητικού δείκτη P (που δείχνει πάντα στο προηγούμενο από το τρέχον στοιχείο).



Εισαγωγη σε ταξινομημενη λιστα

void *LLInsert(integer K, pointer L)*

```
pointer C, ptr;        /* temporary pointers */

C = L;
P = NULL;
while (C != NULL) and (C->Num > K) do
      P = C;
      C = C->next;
if (C != NULL) and (C->Num == K) then
      return;
ptr = NewCell(Node);          /* malloc */
ptr->num = K;
if (P == NULL) then L = ptr;
else  P->next = ptr;
ptr->next = C;
```

Διάσχιση Λίστας

Εκτέλεση επίσκεψης σε ένα ή σε κάποια προκαθορισμένα στοιχεία μιας λίστας με κάποια προκαθορισμένη σειρά.

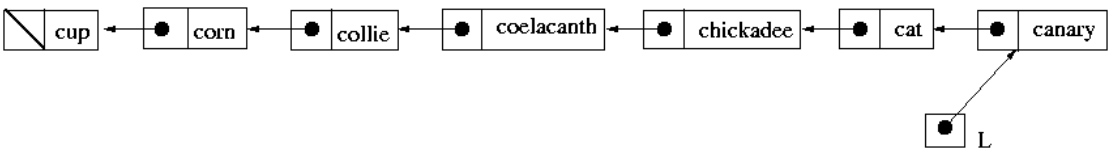
Θεωρούμε λίστα που περιέχει strings (λέξεις) & είναι λεξικογραφικά ταξινομημένη.

Πρόβλημα
Δεδομένης λέξης w, βρείτε την τελευταία λέξη στη λίστα που προηγείται αλφαβητικά της w και τελειώνει με το ίδιο γράμμα όπως η w.

Παράδειγμα

w = crabapple

L = <canary, cat, chickadee, coelacanth, collie, corn, cup>.



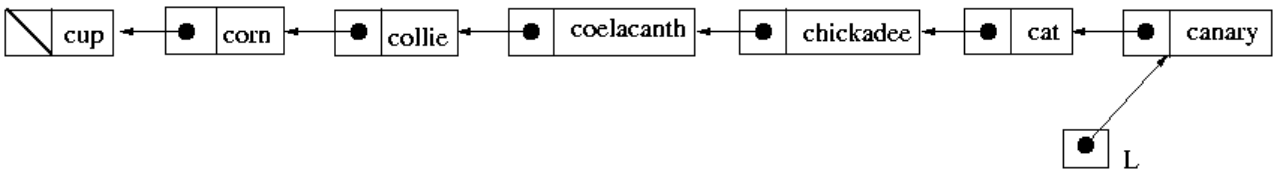
Η απάντηση θα πρέπει να είναι collie.

Πιθανοί Αλγόριθμοι

Αλγόριθμος 1

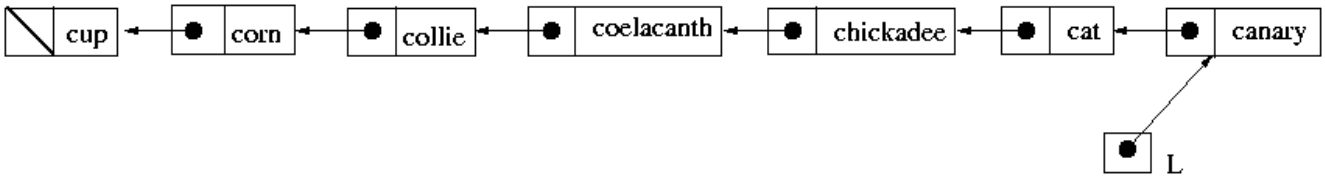
1) Διασχίζουμε τη λίστα προς τα εμπρός ως να βρούμε την πρώτη αλφαβητικά «μεγαλύτερη» λέξη από την crabapple, κρατώντας δείκτες προς τα πίσω (στο παράδειγμα την cup).

2) Ακολουθούμε τους δείκτες προς τα πίσω (που προσθέσαμε κατά τη διάσχιση προς τα εμπρός) μέχρι να βρούμε την πρώτη λέξη που τελειώνει σε e.



Αλγόριθμος 2

Διασχίζουμε τη λίστα προς τα εμπρός κρατώντας έναν δεύτερο (βοηθητικό) δείκτη στο τελευταίο στοιχείο που είδαμε να έχει τη ζητούμενη ιδιότητα.



Ψευδοκώδικας

```
function FindLast(pointer L, string w): string
/*find the last word in L ending with the same letter as w*/
/* return NULL if there is no such word */

   P = NULL;
   C = L;
   while (C != NULL) and (C->string < w) do
       if (C->string ends with the
                   same letter as w) then P = C;
       C = C->Next;
   If (P == NULL) then return NULL;
   else return P->string;
```

Πως θα συγκρίνατε την πολυπλοκότητα των δύο αλγορίθμων?

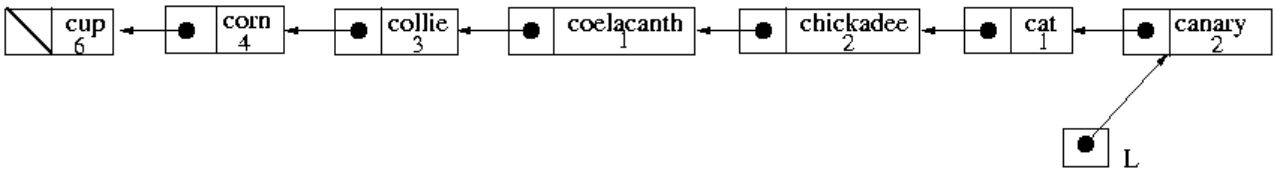
Διασχίσεις Zig-Zag

Παράδειγμα

Έστω ότι κάθε κόμβος έχει τα εξής πεδία:

- String: λέξη
- Num: ακέραιος
- Next: δείκτης στον επόμενο κόμβο

Δεδομένης λέξης w της λίστας η οποία σχετίζεται με τον αριθμό n, βρείτε τη λέξη που προηγείται της w κατά n θέσεις στη λίστα.



Αλγόριθμος

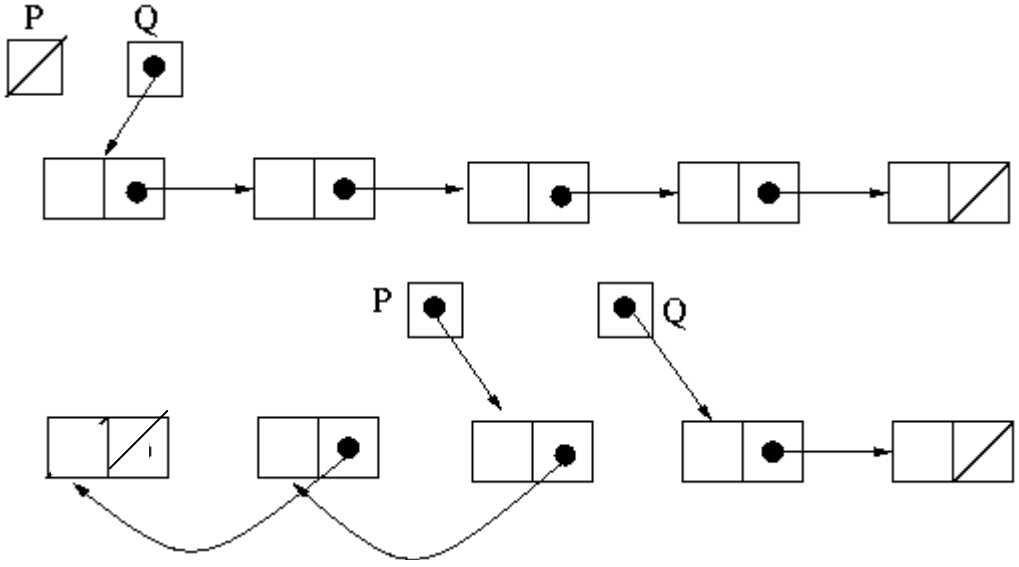
Αναζήτηση της w στη λίστα με ταυτόχρονη κράτηση δεικτών προς τα πίσω.

Οπισθοδρόμηση κατά n θέσεις στη λίστα.

Η λύση με χρήση δεικτών προς τα πίσω είναι ακριβή σε μνήμη!!!! Γιατί?

Υπάρχει λύση φθηνή σε μνήμη?

Μέθοδος Αναστροφής Δεικτών Λίστας



Λειτουργίες

StartTraversal(L):

$$\begin{pmatrix} P \\ Q \end{pmatrix} \leftarrow \begin{pmatrix} NULL \\ L \end{pmatrix}$$

Forward(P,Q):

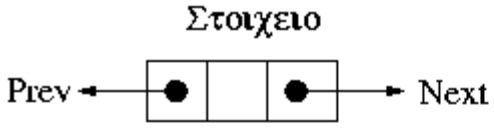
$$\begin{pmatrix} P \\ Q \\ Q->Next \end{pmatrix} \leftarrow \begin{pmatrix} Q \\ Q->Next \\ P \end{pmatrix}$$

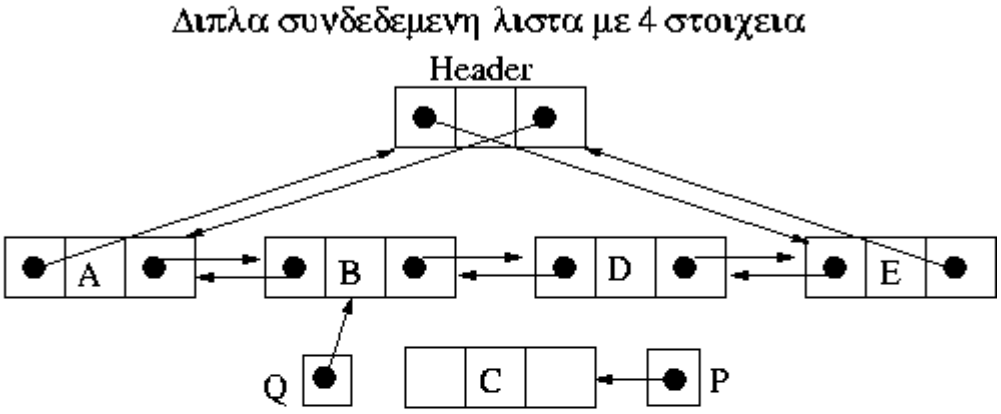
Back(P,Q):

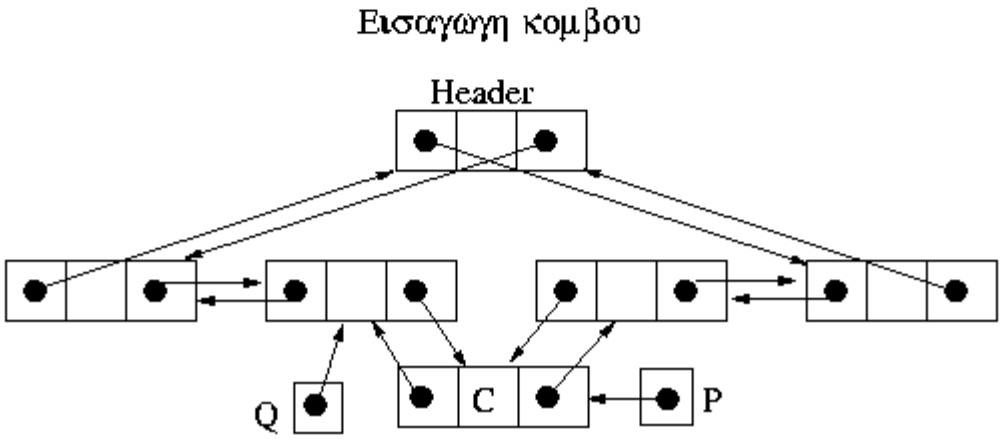
$$\begin{pmatrix} P \\ P->Next \\ Q \end{pmatrix} \leftarrow \begin{pmatrix} P->Next \\ Q \\ P \end{pmatrix}$$

Διπλά Συνδεδεμένες Λίστες

Οι κόμβοι μιας διπλά συνδεδεμένης λίστας περιέχουν δείκτες και προς τα εμπρός και προς τα πίσω και άρα διασχίσεις Zig-Zag είναι εύκολα υλοποιήσιμες.







Λειτουργίες Διπλά Συνδεδεμένης Λίστας

Εισαγωγή κόμβου στον οποίο δείχνει ο P μετά τον κόμβο στον οποίο δείχνει ο Q:

void DoublyLinkedInsert(pointer P,Q)
/* insert node pointed to by P just after node pointed to by Q */

$$\begin{pmatrix} P->Prev \\ P->Next \\ Q->Next \\ Q->Next->Prev \end{pmatrix} \leftarrow \begin{pmatrix} Q \\ Q->Next \\ P \\ P \end{pmatrix}$$

Διαγραφή κόμβου P από τη λίστα

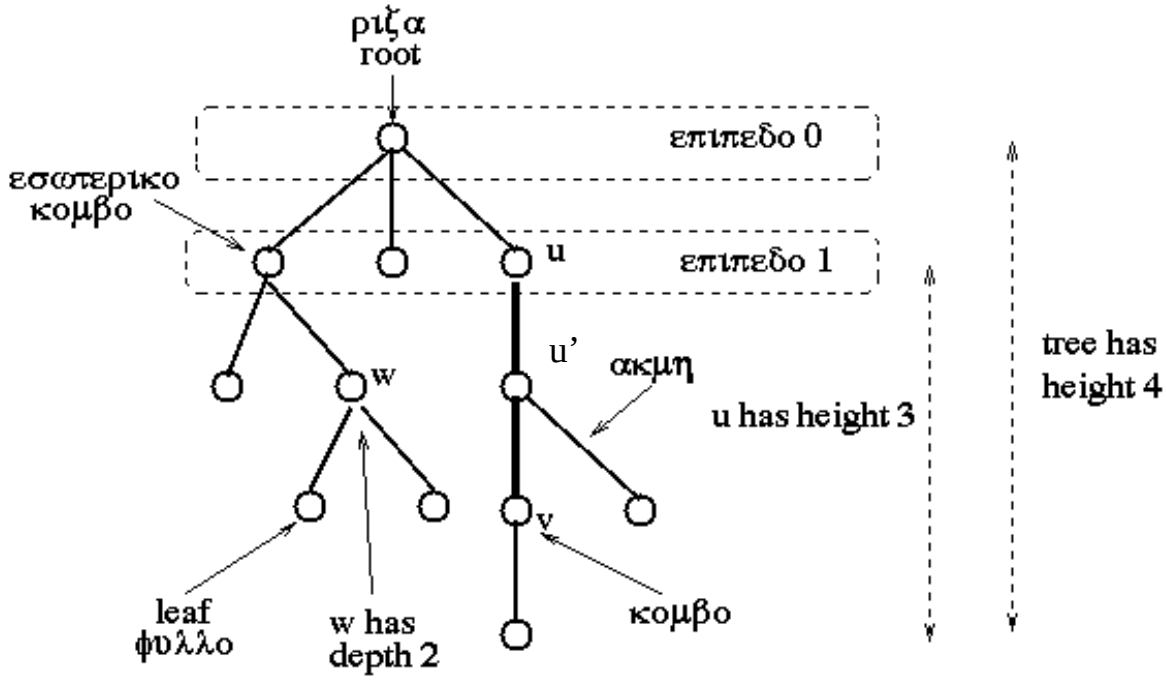
void DoublyLinkedDelete(pointer P)
/* delete node P from its doubly linked list */

$$\begin{pmatrix} P->Prev->Next \\ P->Next->Prev \end{pmatrix} \leftarrow \begin{pmatrix} P->Next \\ P->Prev \end{pmatrix}$$

Πολυπλοκότητα?

ΕΝΟΤΗΤΑ 4
ΔΕΝΔΡΑ

Δένδρα



- Κόμβοι (nodes)
- Ακμές (edges)
- Ουρά και κεφαλή ακμής (tail, head)
- Γονέας – Παιδί – Αδελφικός κόμβος (parent, child, sibling)
- Μονοπάτι (path)
- Πρόγονος – απόγονος (ancestor, descendant)
- Φύλλο – Εσωτερικός Κόμβος (leaf, non-leaf)

Βαθμός Κόμβου (node degree)

Ο αριθμός των υποδένδρων που αρχίζουν από ένα κόμβο.

Βαθμός Δένδρου (tree degree)

Μέγιστος βαθμός των κόμβων του δένδρου.

Επίπεδο (level)

Η ρίζα βρίσκεται στο επίπεδο 0. Ένας κόμβος βρίσκεται στο επίπεδο k αν η απόσταση του από τη ρίζα είναι k. Το επίπεδο είναι επομένως ένα σύνολο από κόμβους.

Ύψος Κόμβου (node height)

Μήκος μακρύτερου μονοπατιού από τον κόμβο σε οποιοδήποτε φύλλο.

Ύψος Δένδρου (tree height)

Μέγιστο ύψος μεταξύ των κόμβων του δένδρου.

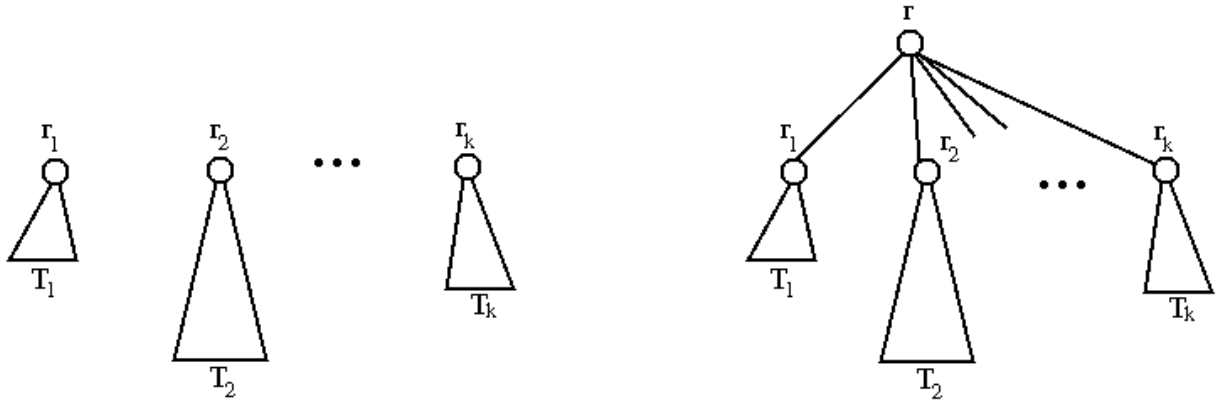
Βάθος Κόμβου (node depth)

Μήκος μονοπατιού από τη ρίζα στον κόμβο.

Βάθος Δένδρου (tree depth)

Μέγιστο βάθος μεταξύ των κόμβων του δένδρου.

Αναδρομικός Ορισμός



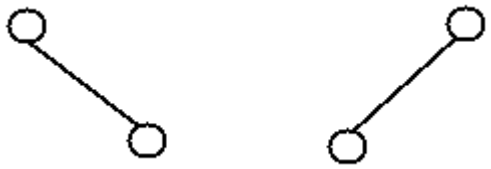
Ένα δένδρο Τ είναι ένα πεπερασμένο σύνολο από έναν ή περισσότερους κόμβους:

- Ένας μόνο κόμβος (χωρίς καμία ακμή) αποτελεί ένα δένδρο. Ο κόμβος αυτός είναι και ρίζα του δένδρου.
- Έστω ότι T_1, ..., T_κ ($\kappa > 0$) είναι δένδρα που δεν μοιράζονται κόμβους και έστω r_1, ..., r_k οι ρίζες τους. Έστω r ένας νέος κόμβος. Αν το Τ αποτελείται από τους κόμβους και τις ακμές των T_1, ..., T_κ, το νέο κόμβο r και τις νέες ακμές $<r,r_1>$, $<r,r_2>$, …, $<r,r_k>$, τότε το Τ είναι δένδρο. Η ρίζα του Τ είναι το r. Τα T_1, ..., T_κ είναι υποδένδρα του Τ.

Είδη Δένδρων

Διατεταγμένο Δένδρο

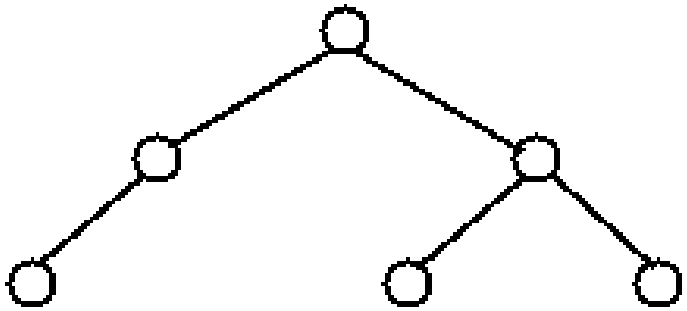
Δένδρο στο οποίο έχει οριστεί μια διάταξη στα παιδιά κάθε κόμβου.



Δυαδικό δένδρο

Διατεταγμένο δένδρο του οποίου κάθε κόμβος έχει το πολύ δύο παιδιά (ένα αριστερό και ένα δεξί).

Λ (nill ή NULL): άδειο δυαδικό δένδρο (που δεν περιέχει κανένα κόμβο και άρα και καμία ακμή)



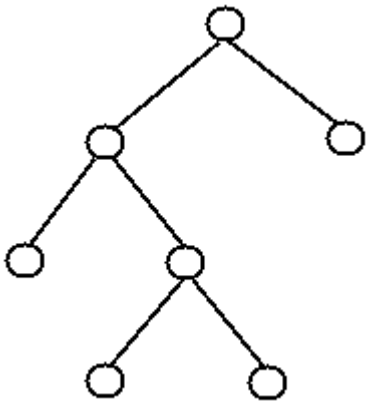
Δάσος

Πεπερασμένο σύνολο από δένδρα.

Είδη και Ιδιότητες Δυαδικών Δένδρων

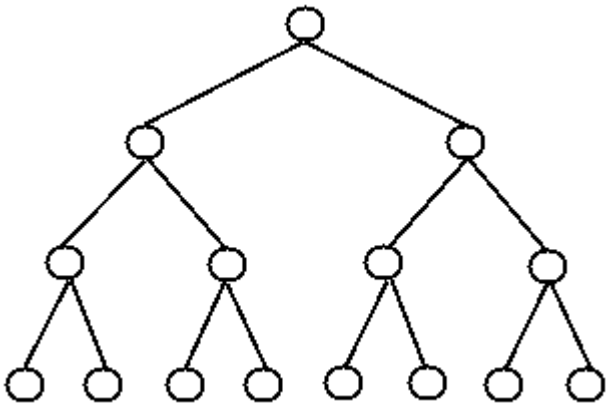
Γεμάτο Δυαδικό Δένδρο (full binary tree)

Δεν υπάρχει κόμβος με μόνο 1 παιδί στο δένδρο.



Τέλειο Δυαδικό Δένδρο (perfect binary tree)

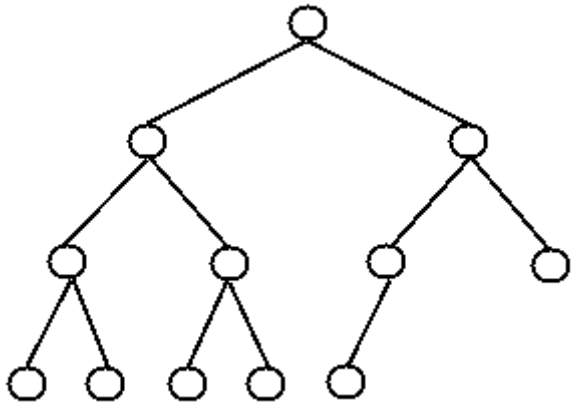
Γεμάτο δυαδικό δένδρο στο οποίο όλα τα φύλλα έχουν το ίδιο βάθος.



Είδη και Ιδιότητες Δυαδικών Δένδρων

Πλήρες Δυαδικό Δένδρο Ύψους h (complete binary tree of height h)

Αποτελείται από ένα τέλειο δυαδικό δένδρο ύψους h-1 στο οποίο έχουν προστεθεί ένα ή περισσότερα φύλλα με ύψος h. Τα φύλλα αυτά έχουν τοποθετηθεί στις αριστερότερες θέσεις του δένδρου.



Αναδρομικός Ορισμός

- Ένα πλήρες δυαδικό δένδρο ύψους 0 αποτελείται από ένα μόνο κόμβο.
- Ένα πλήρες δυαδικό δένδρο ύψους 1 είναι ένα δένδρο ύψους 1 στο οποίο η ρίζα έχει είτε 2 παιδιά ή ένα μόνο **αριστερό** παιδί.
- Ένα πλήρες δυαδικό δένδρο ύψους h>1, αποτελείται από μια ρίζα και 2 υποδένδρα τ.ω:
  - είτε το αριστερό υποδένδρο είναι τέλειο ύψους h-1 και το δεξιό είναι πλήρες ύψους h-1, ή
  - το αριστερό υποδένδρο είναι πλήρες ύψους h-1 και το δεξιό είναι τέλειο ύψους h-2.

Ιδιότητες Δυαδικών Δένδρων

Πρόταση

Ένα τέλειο δυαδικό δένδρο ύψους h έχει $2^{h+1} - 1$ κόμβους, εκ των οποίων 2^h είναι φύλλα και $2^h - 1$ είναι εσωτερικοί κόμβοι.

Απόδειξη

Με επαγωγή στο h.

Βάση επαγωγής, h = 0

Το τέλειο δυαδικό δένδρο ύψους 0 αποτελείται μόνο από ένα κόμβο-ρίζα και άρα έχει 1 κόμβο που είναι φύλλο και 0 εσωτερικούς κόμβους. Πράγματι:

$2^{h+1} - 1 = 2^{0+1} - 1 = 2 - 1 = 1$ κόμβος

$2^h = 2^0 = 1$ φύλλο

$2^h - 1 = 0$ εσωτερικοί κόμβοι

Επαγωγική Υπόθεση

Έστω ότι ένα τέλειο δυαδικό δένδρο ύψους κ έχει $2^{\kappa+1} - 1$ κόμβους, εκ των οποίων 2^κ είναι φύλλα και $2^\kappa - 1$ είναι εσωτερικοί κόμβοι, $\kappa >= 0$.

Απόδειξη Πρότασης (Συνέχεια)

Επαγωγικό βήμα

Θα δείξουμε ότι ο ισχυρισμός είναι σωστός για δένδρα ύψους k+1.

Ένα τέλειο δένδρο Τ ύψους k+1 αποτελείται από 2 τέλεια δένδρα ύψους k (έστω T_1, T_2) και τη ρίζα του. Από επαγωγική υπόθεση καθένα από τα T_1, T_2, έχει $2^{\kappa+1} - 1$ κόμβους, εκ των οποίων 2^{κ} είναι φύλλα και $2^{\kappa} - 1$ είναι εσωτερικοί κόμβοι. Άρα το Τ έχει

- $2*(2^{\kappa+1} - 1) + 1$ κόμβους = $2^{\kappa+2} - 1$ κόμβους (όπως απαιτείται),

εκ των οποίων

- $2^{\kappa} + 2^{\kappa} = 2^{\kappa+1}$ είναι φύλλα (όπως απαιτείται), και
- $2*(2^{\kappa} - 1) + 1 = 2^{\kappa+1} - 1$ είναι εσωτερικοί κόμβοι (όπως απαιτείται).

Λειτουργίες σε Δένδρα

Parent(v): επιστρέφει τον κόμβο γονέα του v ή nill αν ο v είναι η ρίζα

Children(v): επιστρέφει το σύνολο των παιδιών του v ή το άδειο σύνολο αν ο v είναι φύλλο

FirstChild(v): επιστρέφει το πρώτο παιδί του v ή nill αν ο v είναι φύλλο

RightSibling(v): επιστρέφει το δεξιό αδελφικό κόμβο του v ή nill αν ο v είναι η ρίζα ή το δεξιότερο παιδί του γονικού του κόμβου

LeftSibling(v): επιστρέφει τον αριστερό αδελφικό κόμβο του v ή nill αν ο v είναι η ρίζα ή το αριστερότερο παιδί του γονικού του κόμβου

LeftChild(v), RightChild(v): επιστρέφει το αριστερό/δεξιό παιδί του v (ή nill)

IsLeaf(v): επιστρέφει true αν ο v είναι φύλλο, false διαφορετικά

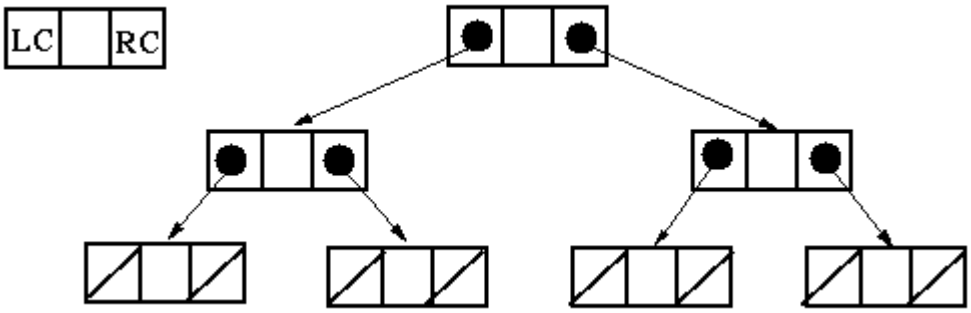
Depth(v): επιστρέφει το βάθος του v στο δένδρο

Height(v): επιστρέφει το ύψος του v στο δένδρο

Υλοποίηση Δένδρων

Υλοποίηση Δυαδικών Δένδρων

Κάθε κόμβος έχει ένα πεδίο Info και 2 δείκτες LC (Left Child) και RC (Right Child) που δείχνουν στο αριστερό και στο δεξιό παιδί του κόμβου αντίστοιχα.



Οι λειτουργίες LeftChild() και RightChild() υλοποιούνται πολύ εύκολα σε Θ(1) χρόνο.

Είναι το ίδιο αλήθεια για τη λειτουργία Parent()?

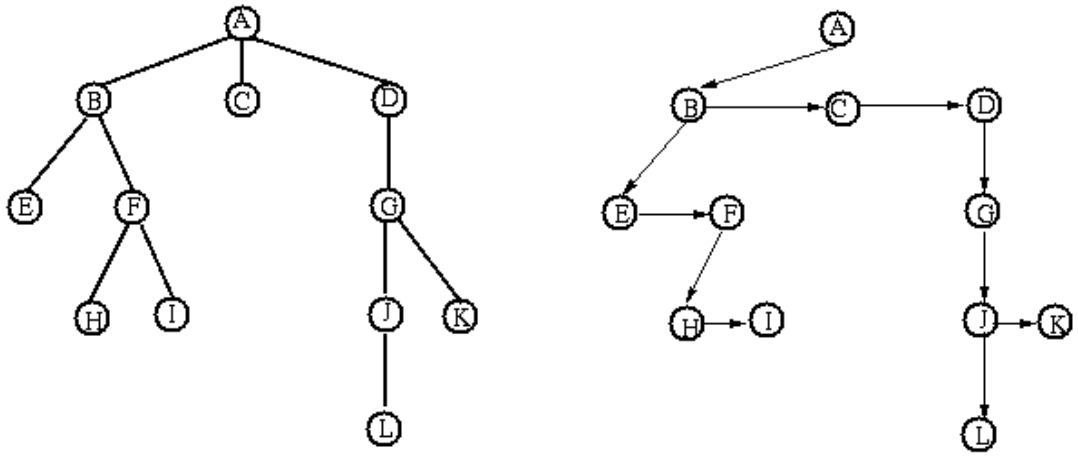
Αποδοτική Υλοποίηση της Parent()

Κρατάμε και ένα τρίτο δείκτη σε κάθε κόμβο που δείχνει στον πατρικό του κόμβο («Διπλά Συνδεδεμένο» δένδρο).

Υλοποίηση Διατεταγμένων Δένδρων

Τι γίνεται αν δεν γνωρίζουμε τον αριθμό των παιδιών που μπορεί να έχει κάποιος κόμβος?

Απεικόνιση Διατεταγμένου σαν Δυαδικό Δένδρο



Έχουμε χάσιμο πληροφορίας?

Μπορούμε να ξαναχτίσουμε το αρχικό δένδρο από το δυαδικό δένδρο?

Ένα δυαδικό δένδρο μπορεί να απεικονίσει και ένα δάσος από διατεταγμένα δένδρα (δεξιός αδελφικός κόμβος της ρίζας είναι σε αυτή την περίπτωση η ρίζα του επόμενου δένδρου στο δάσος)

Ύψος δυαδικού σε σχέση με το αρχικό δένδρο?

Πολυπλοκότητα:

FirstChild(), RightSibling(): Θ(1) χρόνο

K^{th}-child(k, v): εύρεση του κ-οστού παιδιού του v σε Θ(k) χρόνο.

Η Parent() δεν υποστηρίζεται αποδοτικά.

Απεικόνιση Πλήρων Δυαδικών Δένδρων

Υπάρχει μόνο ένα πλήρες δυαδικό δένδρο με n κόμβους και το υλοποιούμε με ένα πίνακα n στοιχείων.

Αριθμούμε τους κόμβους 1...n και αποθηκεύουμε τον κόμβο i στο στοιχείο T[i] του πίνακα.

Θέλουμε να κάνουμε την αρίθμηση με τέτοιο τρόπο ώστε να πετύχουμε την εκτέλεση χρήσιμων λειτουργιών στο δένδρο σε σταθερό χρόνο.

Αρίθμηση:

- Η ρίζα είναι ο κόμβος 0.
- Το αριστερό παιδί του κόμβου i αριθμείται ως κόμβος 2i+1, ενώ το δεξί παιδί του ως κόμβος 2i+2.

Υλοποίηση Λειτουργιών

IsLeaf(i): return (2i+1>n);

LeftChild(i): if (2i+1 < n); return (2i+1) else return nill;

RightChild(i): if (2i+2<n) return(2i+2); else return nill;

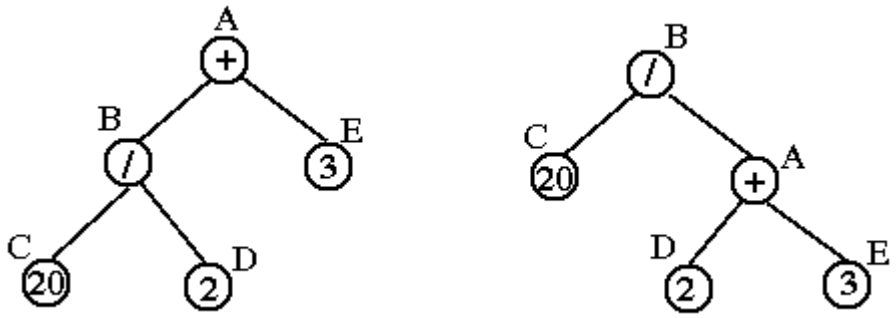
LeftSibling(i): if (i != 0 and i not odd) return (i-1);

RightSibling(i): if (i != n-1 and i not even) return(i+1);

Parent(i): if (i != 0) return($\lfloor (i-1)/2 \rfloor$);

Χρονική πολυπλοκότητα κάθε λειτουργίας: Θ(1)

Δένδρα Αριθμητικών Εκφράσεων



Υπολογισμός Αριθμητικής Έκφρασης

Label(v): ο αριθμός ή η πράξη που αναγράφεται στον v

ApplyOp(op: operation, x,y: numbers): υπολογίζει την έκφραση x <op> y, ανάλογα με το τι είναι το op.

function *Evaluate(pointer P):* **integer**

```
/* Return value of the expression represented by
   the tree with root P */

   if IsLeaf(P) then return Label(P)
   else
        x_l = Evaluate(LeftChild(P))
        x_r = Evaluate(RightChild(P))
        op = Label(P)
        return  ApplyOp(op, x_l, x_r)
```

Διάσχιση Δένδρων

Visit(P): αυθαίρετη λειτουργία που εφαρμόζεται στον κόμβο στον οποίο δείχνει ο δείκτης P

Προδιατεταγμένη Διάσχιση:

Επίσκεψη της ρίζας

Επίσκεψη του αριστερού υποδένδρου

Επίσκεψη του δεξιού υποδένδρου

```
Procedure Preorder(pointer P):
/* P is a pointer to the root of a binary tree */

    Visit(P)
    foreach child Q of P, in order, do
        Preorder(Q)
```

Μεταδιατεταγμένη διάσχιση:

Επίσκεψη του αριστερού υποδένδρου

Επίσκεψη του δεξιού υποδένδρου

Επίσκεψη της ρίζας

```
Procedure Postorder(pointer P):
/* P is a pointer to the root of a binary tree */

    foreach child Q of P, in order, do
        Postorder(Q)
    Visit(P)
```

Διάσχιση Δένδρων

Ενδοδιατεταγμένη διάσχιση:

Επίσκεψη του αριστερού υποδένδρου

Επίσκεψη της ρίζας

Επίσκεψη του δεξιού υποδένδρου

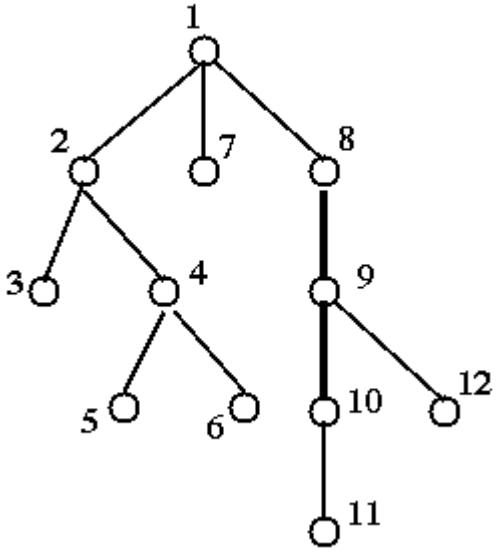
```
Procedure Inorder(pointer P):
/* P is a pointer to the root of a binary tree */

    if P = NULL then return
    else
        Inorder(P->LC)
        Visit(P)
        Inorder(P->RC)
```

Μνημονικός Κανόνας

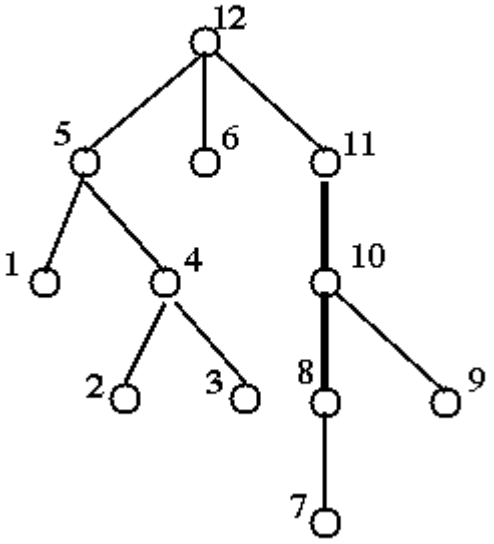
Στην προδιατεταγμένη μορφή επισκεπτόμαστε τον κόμβο πριν από τα παιδιά του, στη μεταδιατεταγμένη μορφή επισκεπτόμαστε τον κόμβο μετά από τα παιδιά του και στην ενδοδιατεταγμένη μορφή επισκεπτόμαστε τον κόμβο μεταξύ των παιδιών του.

Διάσχιση Δένδρων

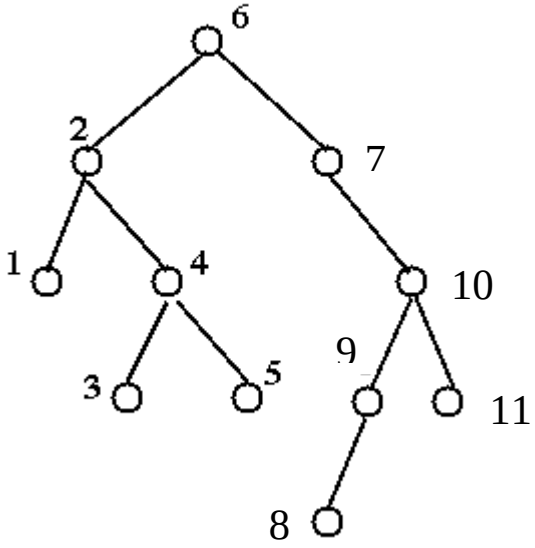
Προδιατεταγμένη



Μεταδιατεταγμένη



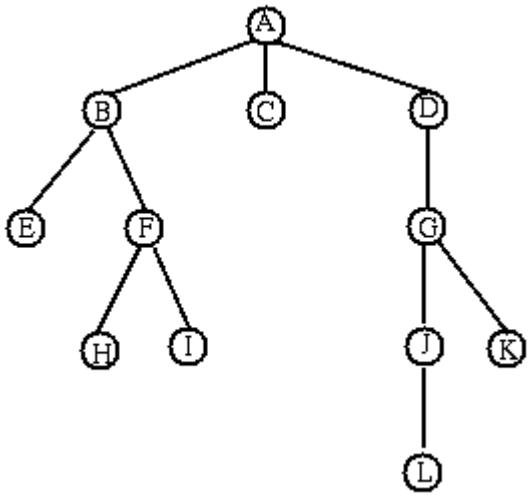
Ενδοδιατεταγμένη



Διάσχιση Δένδρων

Διάσχιση Δένδρου κατά Επίπεδα (κατά πλάτος)

Επισκέπτεται τους κόμβους κατά αύξον βάθος και τους κόμβους του ίδιου επιπέδου από τα αριστερά προς τα δεξιά.



Υλοποιήσεις Διάσχισης Δένδρων

Με Χρήση Στοίβας

Αναδρομικές λύσεις έχουν ήδη συζητηθεί.

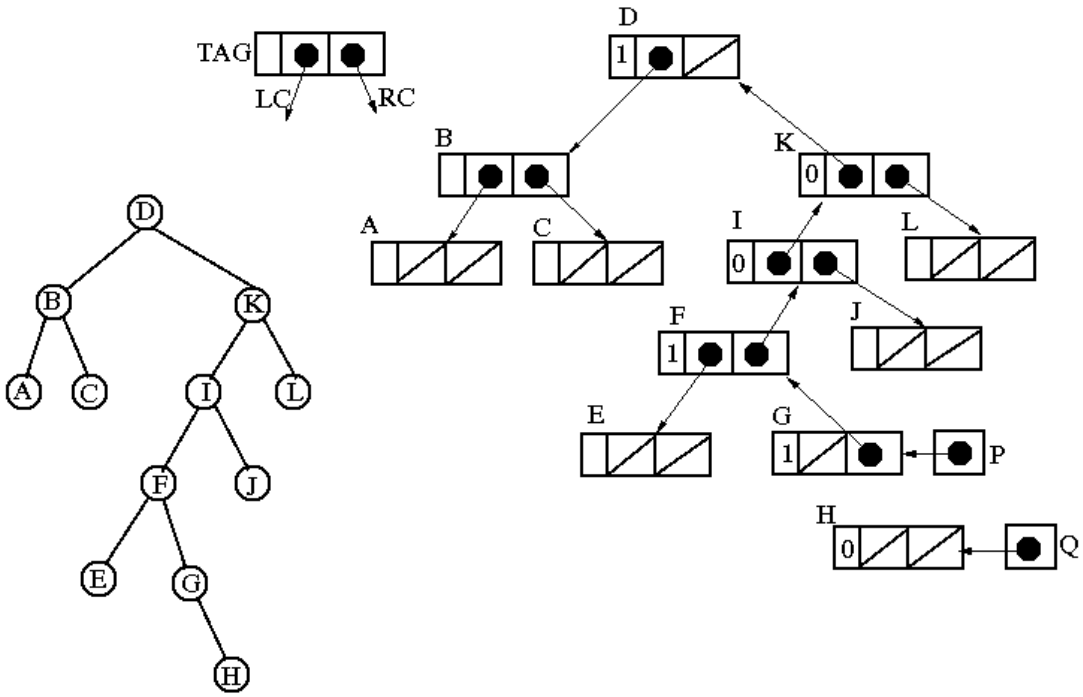
Χρόνος Διάσχισης: O(n), n: αριθμός κόμβων

Μνήμη?

Μέγεθος στοίβας ανάλογο του ύψους του δένδρου.

Άρα, οι αναδρομικοί αλγόριθμοι είναι πολύ απαιτητικοί σε μνήμη.

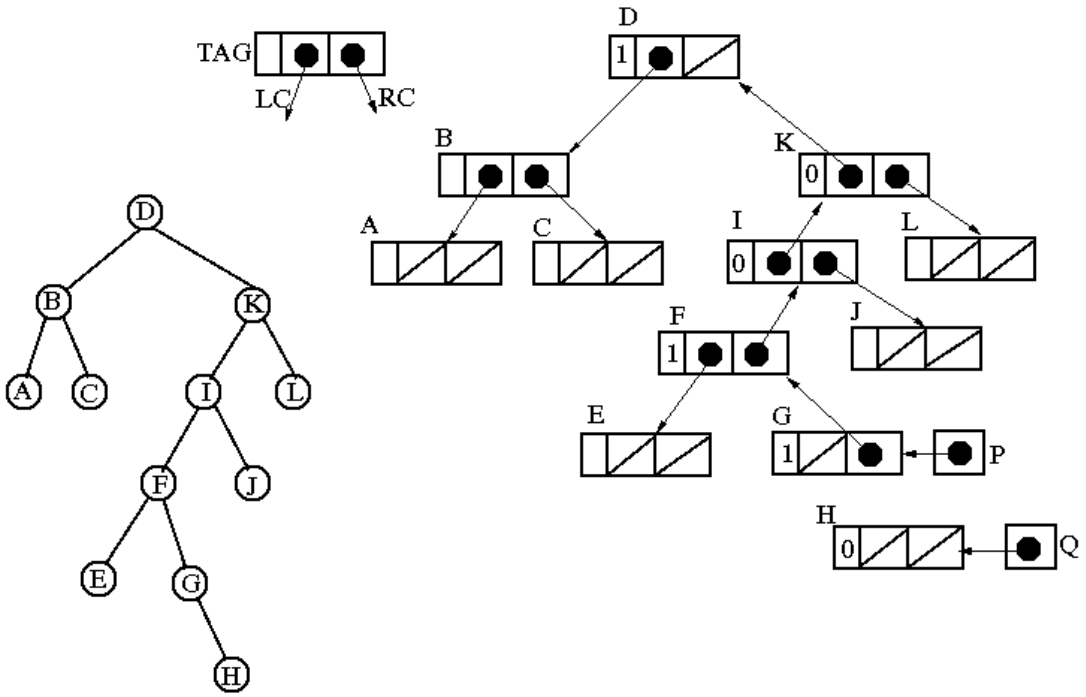
Με Αναστροφή Δεικτών



Procedure *LinkInversionTraversal(pointer Q):*
/* Initially Q points to the root of the tree to be traversed */

```
P = NULL;
while (1)
        while (Q != NULL) do
                Visit(Q)
                Tag(Q) = 0
                descend to left
        while (P != NULL and Tag(P) = 1) do
                ascend from right
                Visit(Q)
        if (P == NULL) then return
        else
                ascend from left
                Visit(Q)
                Tag(Q) = 1
                descend to right
```

Με Αναστροφή Δεικτών



descend to left: descend to right:

$$\begin{pmatrix} P \\ Q \\ Q->LC \end{pmatrix} \leftarrow \begin{pmatrix} Q \\ Q->LC \\ P \end{pmatrix} \qquad \begin{pmatrix} P \\ Q \\ Q->RC \end{pmatrix} \leftarrow \begin{pmatrix} Q \\ Q->RC \\ P \end{pmatrix}$$

ascend from left: ascend from right:

$$\begin{pmatrix} Q \\ P \\ P->LC \end{pmatrix} \leftarrow \begin{pmatrix} P \\ P->LC \\ Q \end{pmatrix} \qquad \begin{pmatrix} Q \\ P \\ P->RC \end{pmatrix} \leftarrow \begin{pmatrix} P \\ P->RC \\ Q \end{pmatrix}$$

Έχουμε βελτίωση στην απαιτούμενη μνήμη?

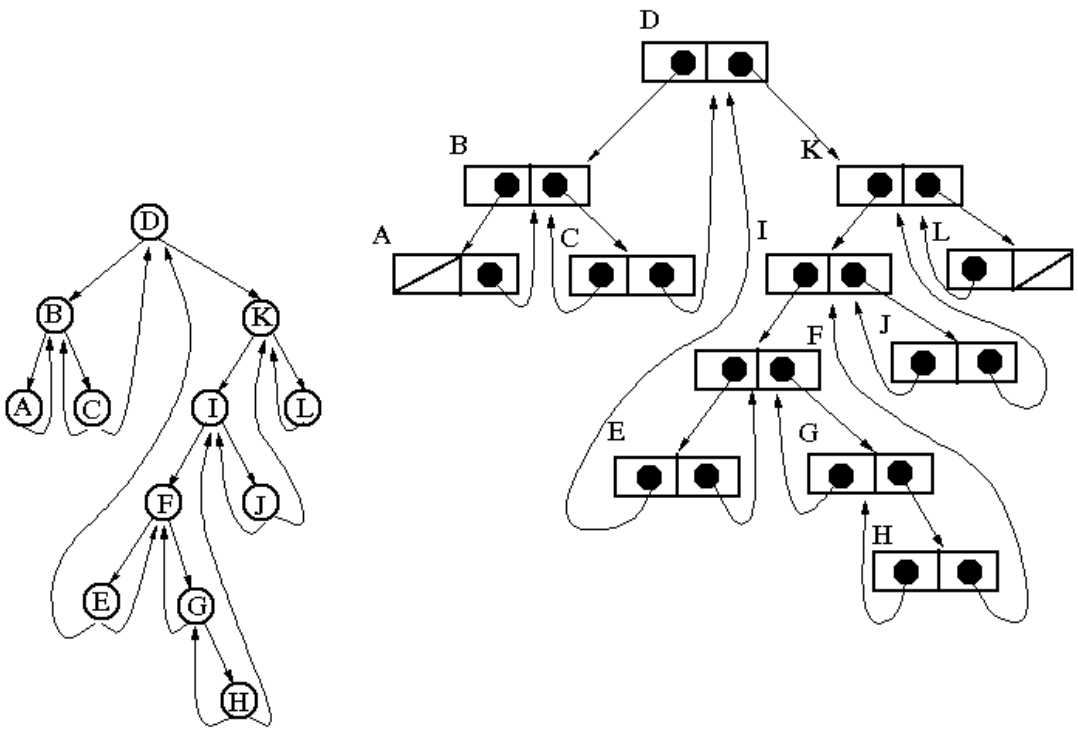
Νηματικά Δυαδικά Δένδρα

- ✓ Σε ένα δυαδικό δένδρο οι μισοί περίπου δείκτες έχουν την τιμή NULL.
- ✓ Η αναδρομική διάσχιση είναι ακριβή σε μνήμη.

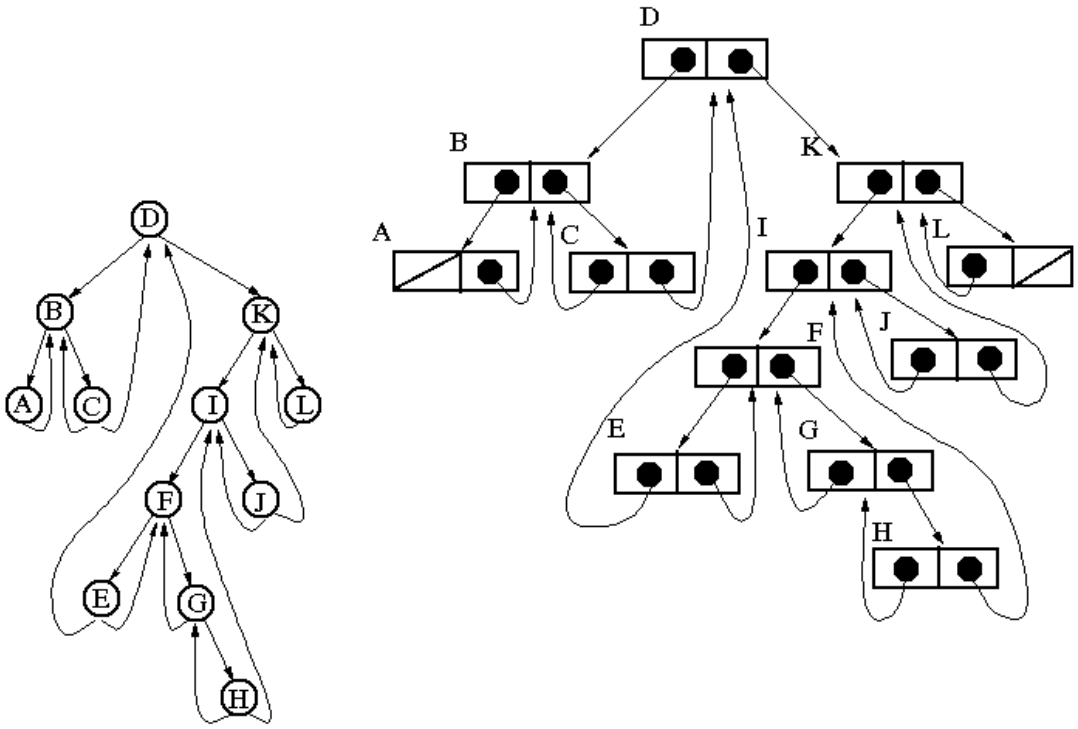
Συχνά οι δείκτες NULL χρησιμοποιούνται για να δείχνουν σε άλλους κόμβους. Τα δένδρα που υλοποιούνται με αυτόν τον τρόπο λέγονται **νηματικά**.

Ο RC δείκτης ενός κόμβου χωρίς δεξιό παιδί δείχνει στον επόμενό του κόμβο στην ενδοδιατεταγμένη διάταξη, ενώ ο LC δείχνει στον προηγούμενό του κόμβο στην ενδοδιατεταγμένη διάταξη.

Χρειαζόμαστε ένα ακόμη bit ανά δείκτη για να ξεχωρίζουμε τους νηματικούς από τους κανονικούς δείκτες.



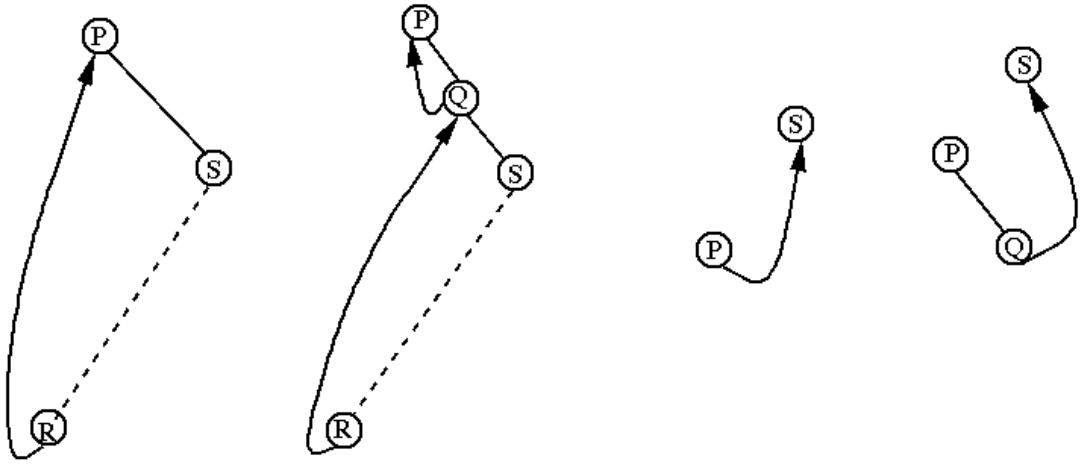
Νηματικά Δυαδικά Δένδρα



function *InorderSuccessor(pointer N):* **pointer**
/* returns the inorder successor of node N, or Λ if N has none */

```
P = N->RC
if (P = NULL) then return NULL
else if (P is not a thread) then
        while (P->LC is not a thread or NULL) do
              P = P->LC
return P
```

Εισαγωγή Κόμβου σε Νηματικό Δυαδικό Δένδρο



procedure *ThreadedInsert(pointer P,Q):*

/* Make node Q the inorder successor of node P */

$$\begin{pmatrix} \boldsymbol{P->RC} \\ \boldsymbol{Q->LC} \\ \boldsymbol{Q->RC} \end{pmatrix} \leftarrow \begin{pmatrix} \boldsymbol{(child)Q} \\ \boldsymbol{(thread)P} \\ \boldsymbol{P->RC} \end{pmatrix}$$

if Q->RC is not a thread then

R = InorderSuccessor(Q);

R->LC = (thread) Q

Υλοποίηση Διάσχισης κατά Επίπεδα

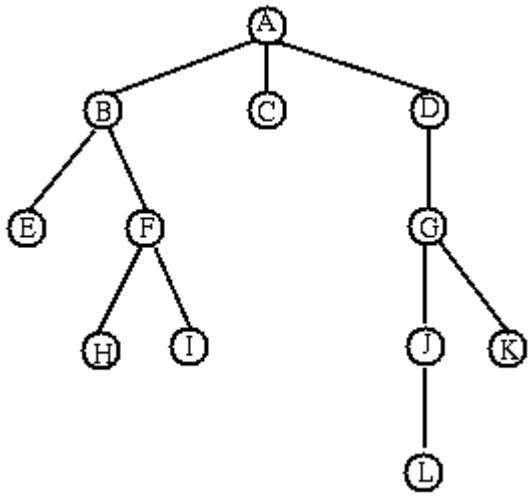
Χρήση Ουράς

- Αρχικά η ουρά περιέχει μόνο τη ρίζα.
- Στη συνέχεια επαναληπτικά: κάνουμε Deque ένα στοιχείο της ουράς και προσθέτουμε τα παιδιά από αριστερά προς τα δεξιά του στοιχείου αυτού.

Παράδειγμα

Περιεχόμενα Ουράς

A
B, C, D
C, D, E, F
D, E, F
E, F, G
F, G
G, H, I
H, I, J, K
I, J, K
J, K
K, L
L
<empty>



ΕΝΟΤΗΤΑ 5
ΣΥΝΟΛΑ & ΛΕΞΙΚΑ

Σύνολα (Sets)

- ✓ Τα μέλη ενός συνόλου προέρχονται από κάποιο χώρο αντικειμένων/στοιχείων (π.χ., σύνολα αριθμών, λέξεων, ζευγών αποτελούμενα από έναν αριθμό και μια λέξη, κοκ).

- ✓ 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 επιτρέπουν 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 $x \in S$»;

- Υπάρχει αυτό το κλειδί στον πίνακα συμβόλων του μεταγλωτιστή;
- Υπάρχει αυτός ο εργαζόμενος στη βάση δεδομένων των εργαζομένων;
- Υπάρχει αυτό το τηλέφωνο στον ηλεκτρονικό τηλεφωνικό κατάλογο;

Χρήσιμες Λειτουργίες στα Σύνολα

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

Χρήσιμες Λειτουργίες στα Σύνολα - Λεξικά

Για χώρους στοιχείων με την ιδιότητα της γραμμικής διάταξης:

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

Θεωρούμε ότι κάθε στοιχείο του συνόλου είναι ένα ζεύγος <K,I> όπου K είναι ένα κλειδί για το στοιχείο, και I είναι η πληροφορία που συνοδεύει το στοιχείο με κλειδί K.

Η τιμή του κλειδιού ενός στοιχείου είναι μοναδική.

- *LookUp(K,S):* Δεδομένου ενός κλειδιού K, επιστρέφει μια πληροφορία I, τέτοια ώστε <K,I> $\in$ S. Αν δεν υπάρχει στοιχείο με κλειδί K στο S, επιστρέφει Λ.

- *Insert(K,I,S)*
- *Delete(K,S)*

Λεξικά

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

Κατάλληλες Δομές Δεδομένων για Λεξικά

Λίστες

Αποθήκευση των στοιχείων του λεξικού σε μια λίστα:

- Σειριακή λίστα (δηλαδή πίνακα)
- Δυναμική Λίστα (απλά ή διπλά συνδεδεμένη).

Τα στοιχεία βρίσκονται στη λίστα διατεταγμένα κατά τον τρόπο εισαγωγής τους.

Υλοποίηση Λειτουργιών

Συνδεδεμένη Λίστα

- LookUp(): Θ(n)
- Insert(): Θ(n)
- Delete(): Θ(n)

Πίνακας

- Το μέγιστο μέγεθος του λεξικού πρέπει να είναι γνωστό εξ αρχής
- LookUp, Insert, Delete: *Χρονική Πολυπλοκότητα;*

Αναμενόμενο Κόστος

- Η πιθανότητα p_j η LookUp() να ψάχνει για το στοιχείο x_j είναι η ίδια για κάθε x, δηλαδή αν έχουμε n στοιχεία είναι 1/n. Έστω ότι c_j είναι το κόστος που πληρώνουμε για να βρούμε το στοιχείο x_j, δηλαδή $c_j = j$.
- Το αναμενόμενο κόστος είναι το άθροισμα των $(c_j * p_j)$ για κάθε j.

$$(\sum_{i=1}^{n} i) / n = (n+1) / 2$$

Άρα το αναμενόμενο κόστος είναι επίσης Θ(n).

Διαφορετικές Πιθανότητες για κλειδιά

- $K_1, \ldots, K_n$: τα κλειδιά στο λεξικό σε φθίνουσα διάταξη αναζήτησης από την 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;

Ευριστικά

- Η πραγματική κατανομή πιθανότητας συνήθως δεν είναι γνωστή.
- Το λεξικό μπορεί να αλλάζει μέγεθος κατά τη χρήση του.

➢ Η μελέτη πιθανοτικών μοντέλων απέχει αρκετά από το τι γίνεται στην πράξη!

Heuristic "Move-To-Front" (Ευριστικό «Μετακίνησε στην Αρχή»): Μετά από κάθε επιτυχημένη αναζήτηση, μετακίνησε το στοιχείο που βρέθηκε στην αρχή της λίστας.

Πόσο ακριβό είναι αυτό;
Συνδεδεμένη Λίστα – Σειριακή Λίστα.

Το αναμενόμενο κόστος του ευριστικού Move-To-Front είναι το πολύ 2 φορές χειρότερο από εκείνο του βέλτιστου αλγόριθμου.

Heuristic Transpose (Αλληλομετάθεσης):
Αν το στοιχείο που αναζητήθηκε δεν είναι το πρώτο της λίστας, μετακινείται μια θέση προς τα εμπρός και ανταλλάσσεται με το προηγούμενό του στη λίστα.

- Καλύτερο αναμενόμενο κόστος από MoveToFront
- Σταθεροποιείται σε μια σταθερή "καλή" κατάσταση πιο αργά από το MoveToFront.

Διατεταγμένες Λίστες

Υλοποίηση LookUp() ως Δυαδική Αναζήτηση

- Χρήση Πίνακα για αποθήκευση στοιχείων + BinarySearch() για LookUp().

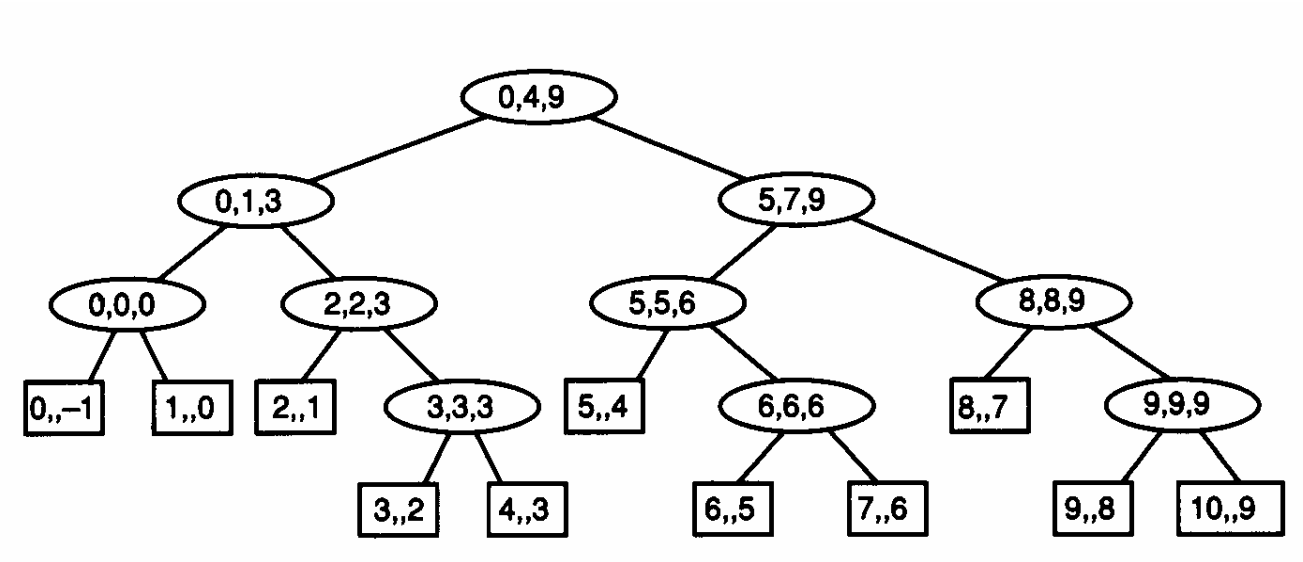
- Τα στοιχεία του πίνακα είναι ταξινομημένα με βάση το Key τους.

BinarySearch - Μη Αναδρομική Έκδοση

function *BinarySearchLookUp*(**key** K, **table** T[0..n-1]):**info**
/* Return information stored with key K in T, or NULL if K is not in T */

```
left = 0;
right = n-1;

repeat forever
    if (right < left) then
        return NULL
    else
        middle = ⌊(left+right)/2⌋;
        if (K == T[middle]->Key) then
            return T(middle]->Info;
        else if (K < T[middle]->Key) then
            right = middle-1;
        else left = middle+1;
```

Δυνατές Εκτελέσεις BinarySearchLookUp



Κυκλικοί κόμβοι: εσωτερικοί
Τετραγωνισμένοι κόμβοι: εξωτερικοί

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

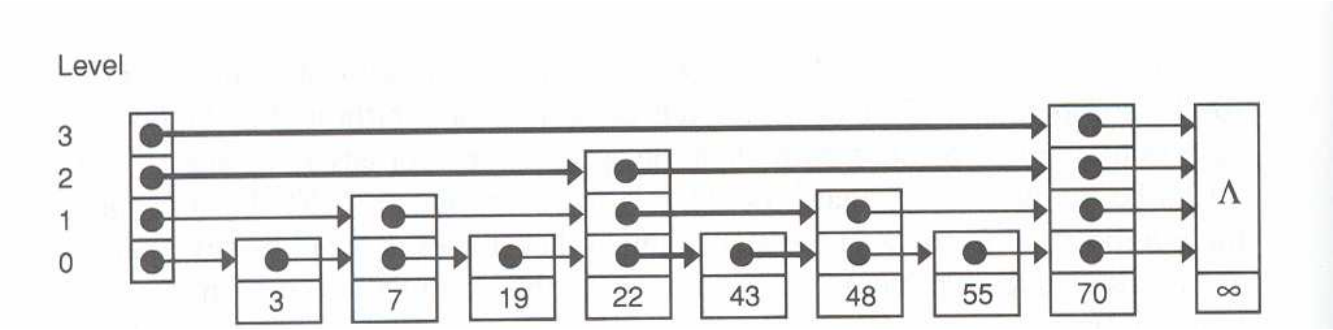
Απόδειξη: Αν h το ύψος του δένδρου, οι εσωτερικοί κόμβοι αποδεικνύεται (επαγωγικά) ότι ανήκουν σε ένα τέλειο δυαδικό δένδρο με ύψος h-1. Αν n είναι οι κόμβοι του τέλειου δένδρου έχουμε δείξει πως $n = 2^h - 1$. Άρα,

μέγιστο βάθος εσωτερικού κόμβου = logn.

Εξωτερικοί κόμβοι;

Skip Lists (Λίστες Παράλειψης)

Είναι δυνατό να εφαρμόσει κάποιος (αποδοτικά) δυαδική αναζήτηση σε μια συνδεδεμένη λίστα;



- Περίπου logn δείκτες ανά στοιχείο χρειάζονται για μια πλήρως συνδεδεμένη λίστα με n στοιχεία.
- Τα περισσότερα στοιχεία δεν χρειάζεται να κρατούν τόσους πολλούς δείκτες:

  Μόνο οι μισοί χρειάζονται έναν ακόμη δείκτη, και από αυτούς μόνο οι μισοί έναν ακόμη, κοκ.
- Οι standard δείκτες της λίστας ονομάζονται δείκτες επιπέδου 0.
- Οι δείκτες που δείχνουν στον 2^i–οστό επόμενο κόμβο λέγονται δείκτες επιπέδου i.
- Ένας κόμβος κεφαλή δείχνει στους αρχικούς κόμβους κάθε επιπέδου.

Skip Lists (Λίστες Παράλειψης)

Αλγόριθμος Αναζήτησης

Ξεκινώντας με τους δείκτες του υψηλότερου επιπέδου:

Ακολούθησε δείκτες μέχρι να βρεθεί ένα στοιχείο με κλειδί > ή ίσο Κ.
Αν ίσο επέστρεψε με επιτυχία.
Αν μεγαλύτερο, οπισθοδρομούμε κατά ένα δείκτη και επαναλαμβάνουμε ακολουθώντας δείκτες του επόμενου χαμηλότερου επιπέδου.
Αν ένας δείκτης επιπέδου 0 οδηγεί σε κλειδί > Κ, επέστρεψε ανεπιτυχώς.

Πως γίνεται η οπισθοδρόμηση;

Σε μια τέλεια οργανωμένη λίστα παράλειψης, ποιά θα ήταν η πολυπλοκότητα του αλγορίθμου;

Είναι οι τέλεια οργανωμένες λίστες παράλειψης πρακτικές στην περίπτωση που έχουμε εισαγωγές και εξαγωγές στοιχείων;

Skip Lists (Λίστες Παράλειψης)

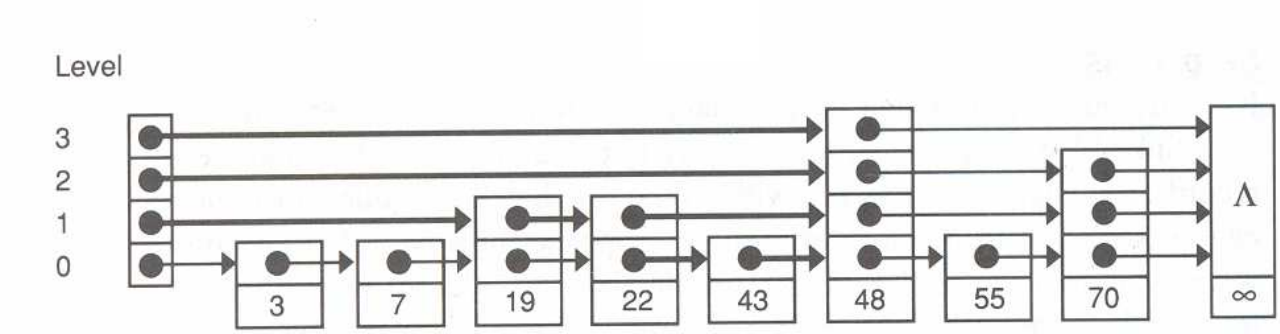
- ✓ Αντί τέλεια οργανωμένων λιστών παράλειψης χρησιμοποιούμε μη-τέλεια οργανωμένες λίστες παράλειψης που τις φτιάχνουμε εισάγοντας κάποια τυχαιότητα στο σύστημα:

- ➢ Ένας κόμβος με (j+1) δείκτες, ένα για κάθε επίπεδο 0, 1, ..., j, ονομάζεται κόμβος επιπέδου j.

Βασική Ιδέα

Οι κόμβοι των διαφόρων επιπέδων εξακολουθούν να υπάρχουν στη λίστα σε «περίπου» ίδια αναλογία, αλλά είναι διασκορπισμένη τυχαία μέσα στη λίστα.

Κόμβοι μεγαλύτερου επιπέδου πρέπει να συναντούνται λιγότερο συχνά.



Skip Lists (Λίστες Παράλειψης)

Με τα νέα μας δεδομένα η εισαγωγή γίνεται αρκετά πιο εύκολη:

- Βρες την κατάλληλη θέση εισαγωγής στη λίστα;
- Επέλεξε το επίπεδου του κόμβου με τυχαίο τρόπο, αλλά με βάση τον κανόνα:

  «Για κάθε επίπεδο j, η πιθανότητα να επιλεγεί το j ως επίπεδου του κόμβου είναι διπλάσια από την πιθανότητα να επιλεγεί το j+1».

✓ N: ο μέγιστος δυνατός αριθμός στοιχείων της λίστας

✓ MaxLevel = $\lfloor \log N \rfloor$ -1: μέγιστο επίπεδο κάθε κόμβου

Κάθε κόμβος μιας λίστας παράλειψης έχει τρία πεδία:

- Key
- Info
- ένα πίνακα Forward από δείκτες.

Η **λίστα παράλειψης** είναι struct με δύο πεδία:

- Header: δείκτης σε dummy node (που περιέχει πίνακα Forward από MaxLevel δείκτες). Ο δείκτης Header->Forward[j] δείχνει στον πρώτο κόμβο του επιπέδου j.
- Level: ακέραιος

Αρχικά, Level = 0, και όλοι οι δείκτες NULL.

Skip Lists (Λίστες Παράλειψης): Εισαγωγή

Αλγόριθμος Αναζήτησης

function *SkipListLookUp*(**key** K, **pointer** L): info
/* επιστρέφει το Info του στοιχείου με κλειδί K στη λίστα παράλειψης αν υπάρχει, διαφορετικά Λ */

```
P = L->Header;
for j from L->Level downto 0 do
      while ((P->Forward[j])->Key < K) do
            P = P->Forward[j];
P = P->Forward[0];
if (P->Key == K) return P->Info;
else return Λ;
```

function *RandomLevel*():**integer**
/* παράγει ένα τυχαίο επίπεδο μεταξύ 0 και MaxLevel */

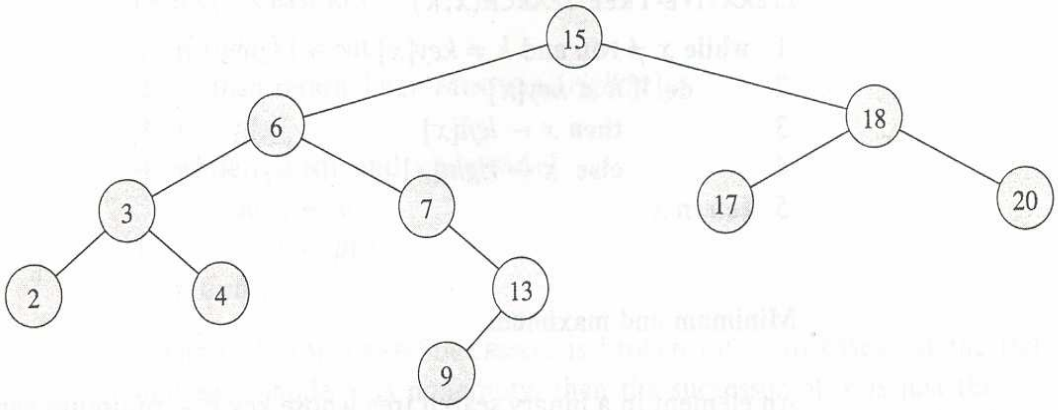
```
v = 0;
while (Random() < ½ && v < MaxLevel) do
      v = v+1;
return v;
```

Skip Lists (Λίστες Παράλειψης): Εισαγωγή

procedure *SkipListInsert*(**key** K, **info** I, **pointer** L):**pointer**

```
/* Εισάγει την πληροφορία I με κλειδί K στην λίστα παράλειψης
                                    L */

P = L->Header;
for j from L->Level downto 0 {
        while ((P->Forward[j])->Key < K) do
                P = P->Forward[j];
        Update[j] = P;
}

P = P->Forward[0];
if (P->Key == K) P->Info = I;
else {
        NewLevel = RandomLevel();
        if (NewLevel > L->Level) {
                for j from L->Level+1 to NewLevel do
                        Update[j] = L->Header;
                L->Level = NewLevel;
        }
        P = NewCell(Node);
        P->Key = K;
        P->Info = I;
        for i from 0 to NewLevel {
             P->Forward[i] = ((Update[i])->Forward)[i]
             ((Update[i])->Forward)[i] = P;
        }
}
```

Ταξινομημένα Δυαδικά Δένδρα (ή Δυαδικά Δένδρα Αναζήτησης)



Είναι δυαδικά δένδρα που με κάθε κόμβο τους έχει συσχετιστεί μια τιμή από ένα χώρο τιμών στον οποίο είναι ορισμένη μια γραμμική διάταξη. Σε κάθε κόμβο η τιμή είναι μεγαλύτερη από όλες τις τιμές των κόμβων του αριστερού υποδένδρου και μικρότερη από όλες τις τιμές των κόμβων του δεξιού υποδένδρου.

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

Το μέγιστο ύψος δυαδικού δένδρου με n κόμβους είναι n-1.

Το ελάχιστο ύψος δυαδικού δένδρου με n κόμβους είναι logn.

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

Υλοποίηση λειτουργίας LookUp σε Ταξινομημένα Δυαδικά Δένδρα

Αναδρομική Έκδοση

function *BinaryTreeLookUp*(**key** K, **pointer** P):**info**

/* Εύρεση του κλειδιού Κ στο δένδρο Ρ, με αναδρομική αναζήτηση και επιστροφή του Info πεδίου του ή Λ αν το κλειδί δεν υπάρχει στο δένδρο */

```
if (P == NULL) return Λ;
else if (K == P->Key) return P->data;
else if (K < P->Key)
        return(BinaryTreeLookUp(K, P->LC));
else return(BinaryTreeLookUp(K, P->RC));
```

Μη-Αναδρομική Έκδοση

function *BinaryTreeLookUp*(**key** K, **pointer** P):**info**

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

- *Χρονική πολυπλοκότητα?*
- *Κόμβος Φρουρός?*

Ελάχιστο και Μέγιστο Στοιχείο

function *TreeMinimum*(**pointer** P): **info**
/* P είναι δείκτης στη ρίζα του δένδρου */

```
if (P == NULL) return error;
while (P->LC != NULL) P = P->LC;

return(P->data);
```

function *TreeMaximum*(**pointer** P): **info**
/* P είναι δείκτης στη ρίζα του δένδρου */

```
if (P == NULL) return error;
while (P->RC != NULL) P = P->RC;

return (P->data);
```

Πολυπλοκότητα?

Επόμενο και Προηγούμενο Στοιχείο

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

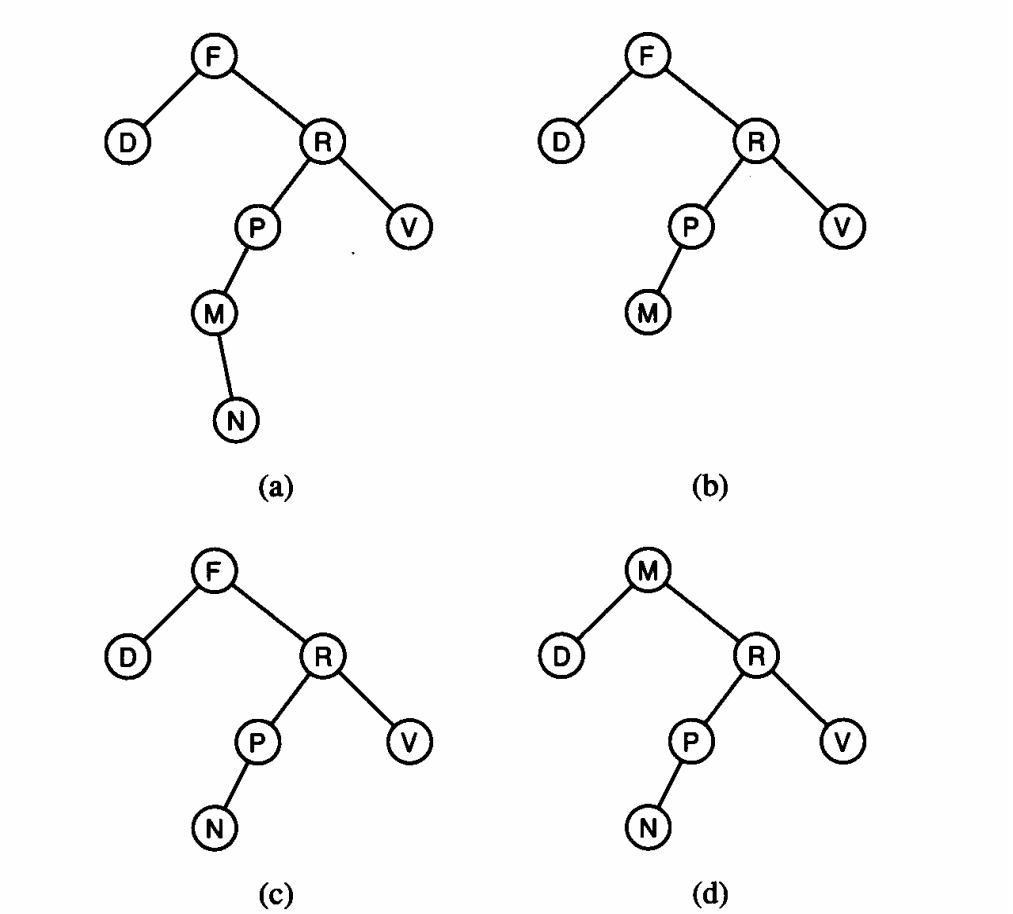
Εισαγωγές σε Ταξινομημένο Δυαδικό Δένδρο

```
function BinSearchTreeInsert(key K, info I,
                             pointer R): pointer
        /* 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;
```

Οι νέοι κόμβοι εισάγονται πάντα σαν παιδιά φύλλων. *Είναι αυτό καλό?*

Διαγραφές: Ταξινομημένο Δυαδικό Δένδρο

Όταν ένας κόμβος διαγράφεται, η ενδο-διατεταγμένη διάσχιση των υπόλοιπων κόμβων πρέπει να «δίνει» τα κλειδιά στη διάταξη που είχαν πριν τη διαγραφή.



Περιπτώσεις:

1) **Ο προς διαγραφή κόμβος είναι φύλλο.** *Απλά τον διαγράφουμε.*

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 2 παιδιά.** *Αντικαθιστούμε τον κόμβο με τον επόμενο του στην ενδο-διατεταγμένη διάσχιση.*

Διαγραφή: Ταξινομημένο Δυαδικό Δένδρο

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

function *BinaryTreeDelete*(**pointer** *R, **pointer** z): **pointer**

```
/* Ο R είναι η διεύθυνση ενός δείκτη στη ρίζα του δένδρου */
/* Ο z είναι δείκτης στον προς διαγραφή κόμβο */

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;
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;
```

Διαγραφές σε Ταξινομημένο Δυαδικό Δένδρο

Γιατί ο επόμενος και όχι ο προηγούμενος στην ενδο-διατεταγμένη διάσχιση?

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

Στατικά Ταξινομημένα Δυαδικά Δένδρα

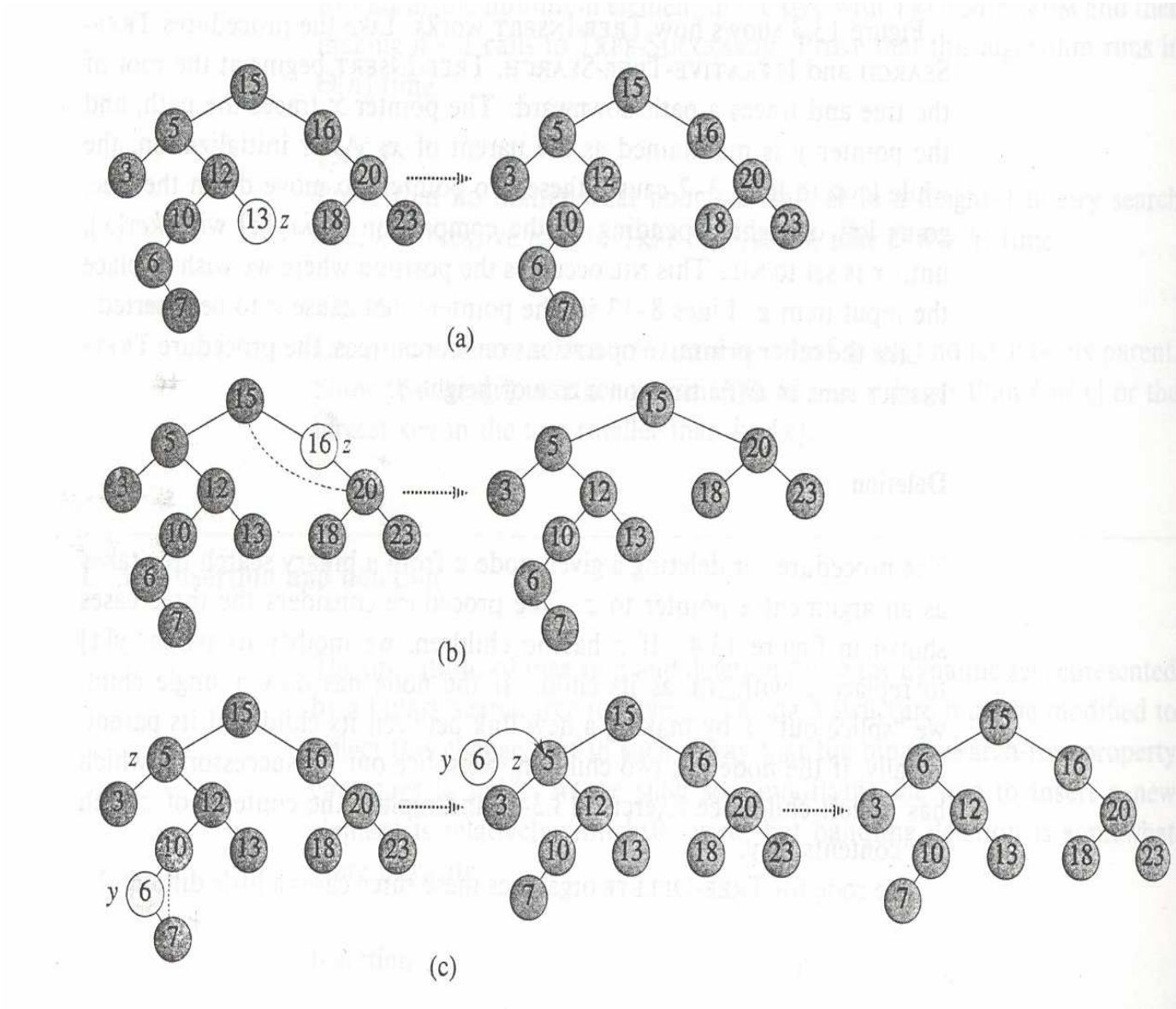
Υπάρχουν κλειδιά που αναζητούνται πιο συχνά και άλλα που αναζητούνται πιο σπάνια.

Σε μια λίστα συχνά ζητούμενα κλειδιά κρατούνται όσο το δυνατόν πιο κοντά στην αρχή της λίστας.

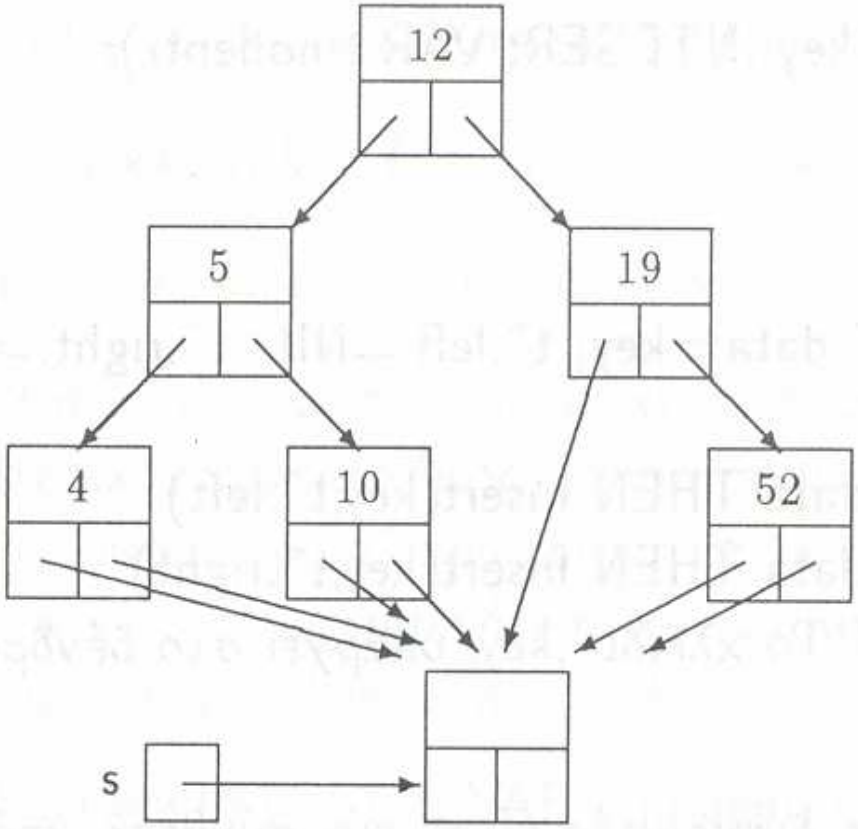
Ποιο είναι το ανάλογο σε ένα ταξινομημένο δυαδικό δένδρο?

Διαγραφή σε Ταξινομημένο Δυαδικό Δένδρο

Παράδειγμα



Ταξινομημένο Δυαδικό Δένδρο Αναζήτησης με Κόμβο Φρουρό



ΕΝΟΤΗΤΑ 6
ΔΥΝΑΜΙΚΑ ΛΕΞΙΚΑ ΙΣΟΖΥΓΙΣΜΕΝΑ ΔΕΝΔΡΑ

Δενδρικές Δομές για Υλοποίηση Δυναμικών Λεξικών

? *Αναζητάμε δένδρα για την υλοποίηση δυναμικών λεξικών που να υποστηρίζουν τις λειτουργίες LookUp(), Insert() και Delete() σε χρόνο O(logn).*

✓ Για κάθε κόμβο v ενός δυαδικού δένδρου T, ορίζουμε

LeftHeight(v) = 0, αν v->LC = NULL,
LeftHeight(v) = 1 + Height(v->LC), διαφορετικά.

✓ Το **RightHeight(v)** ορίζεται αντίστοιχα.

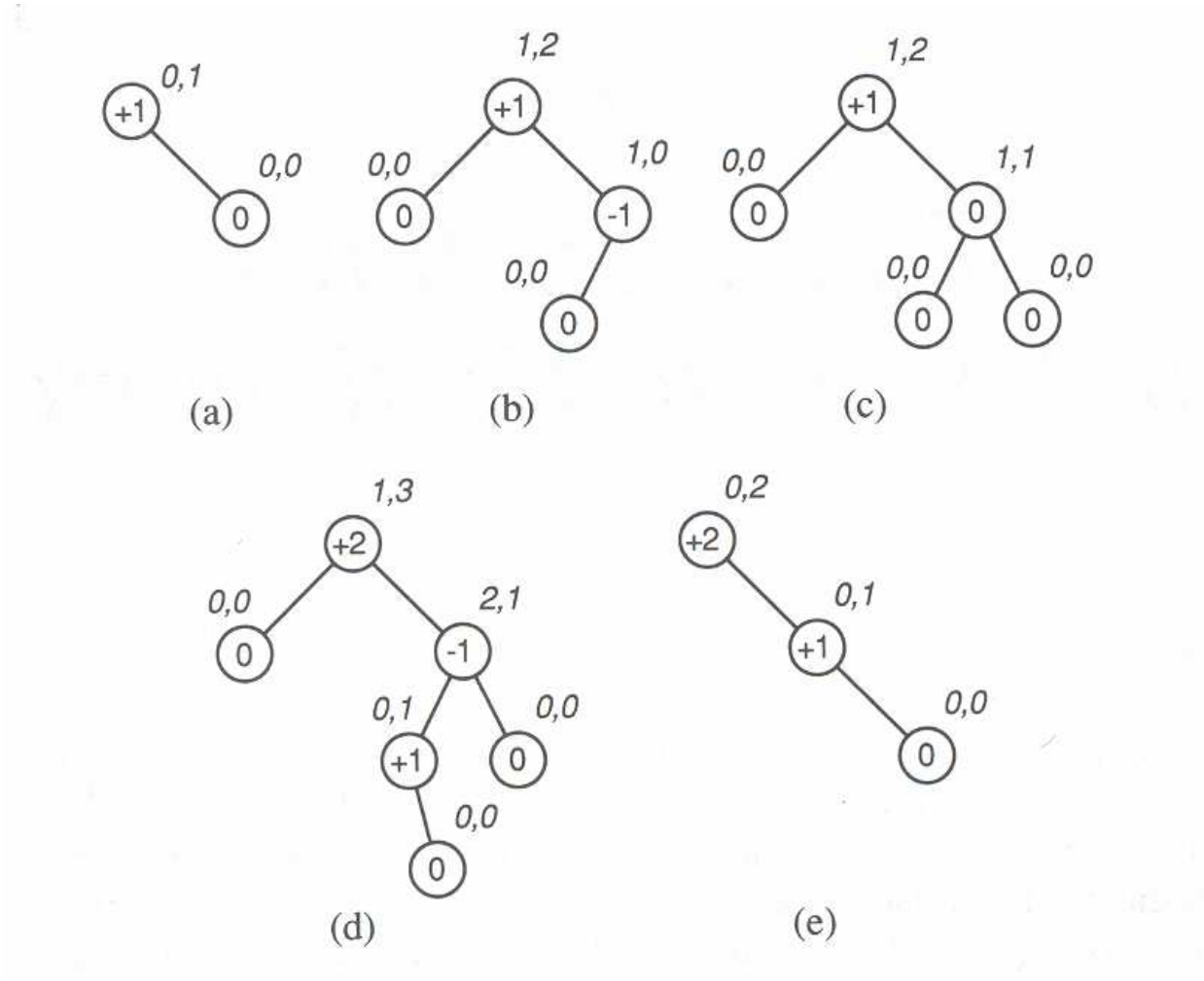
✓ Το **LeftHeight(T) (RightHeight(T))** του δένδρου ισούται με το **LeftHeight(r) (RightHeight(r)**, αντίστοιχα), όπου r είναι ο κόμβος ρίζα του T.

✓ Το **balance** (**ισοζύγιο**) του κόμβου v είναι

balance(v) = RightHeight(v) – LeftHeight(v).

Ισοζυγισμένα κατά ύψος δένδρα
AVL Δένδρα

✓Ένα *δυαδικό δένδρο αναζήτησης* Τ λέγεται **ισοζυγισμένο κατά ύψος** αν κάθε κόμβος ν του Τ έχει balance ή 0, ή –1 ή +1.



Χαρακτηριστικά AVL Δένδρων

✓ Κάθε AVL δένδρο έχει ύψος O(logn).

Τι σημαίνει αυτό για την πολυπλοκότητα της LookUp()?

✓ Ένας κόμβος μπορεί να προστεθεί ή να αφαιρεθεί από ένα AVL δένδρο χωρίς να καταστραφεί η AVL ιδιότητα (ιδιότητα ισοζυγισμού κατά ύψος) σε χρόνο O(logn).

Αναπαράσταση

Κάθε κόμβος είναι ένα struct με πεδία Key, Info, LC, RC, balance.

Πόσο χώρο στη μνήμη χρειαζόμαστε για να αποθηκεύσουμε το balance?

Εισαγωγές κόμβων σε AVL δένδρο

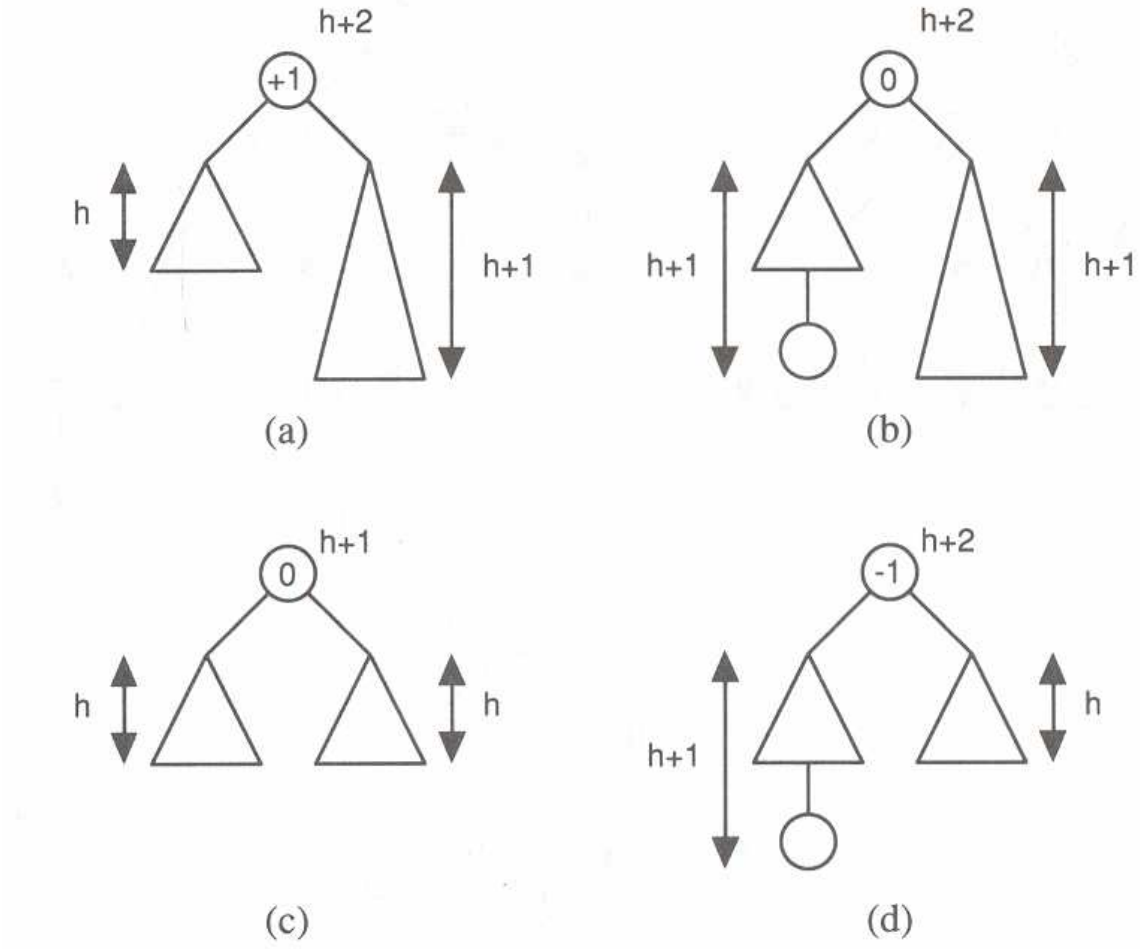
Πως μπορούμε να υλοποιήσουμε την Insert()?Που διαφέρει από την Insert() σε δυαδικό δένδρο αναζήτησης?

- Ακολουθώντας τη γνωστή μέθοδο εισαγωγής σε δυαδικό δένδρο αναζήτησης, βρες το μονοπάτι από τη ρίζα στο κατάλληλο φύλλο στο οποίο θα γίνει η εισαγωγή. *Αποθήκευσε αυτό το μονοπάτι (ανεστραμμένο).*

- Ακολούθησε προς τα πίσω αυτό το μονοπάτι και υπολόγισε τα νέα balances για τους κόμβους του μονοπατιού αυτού.

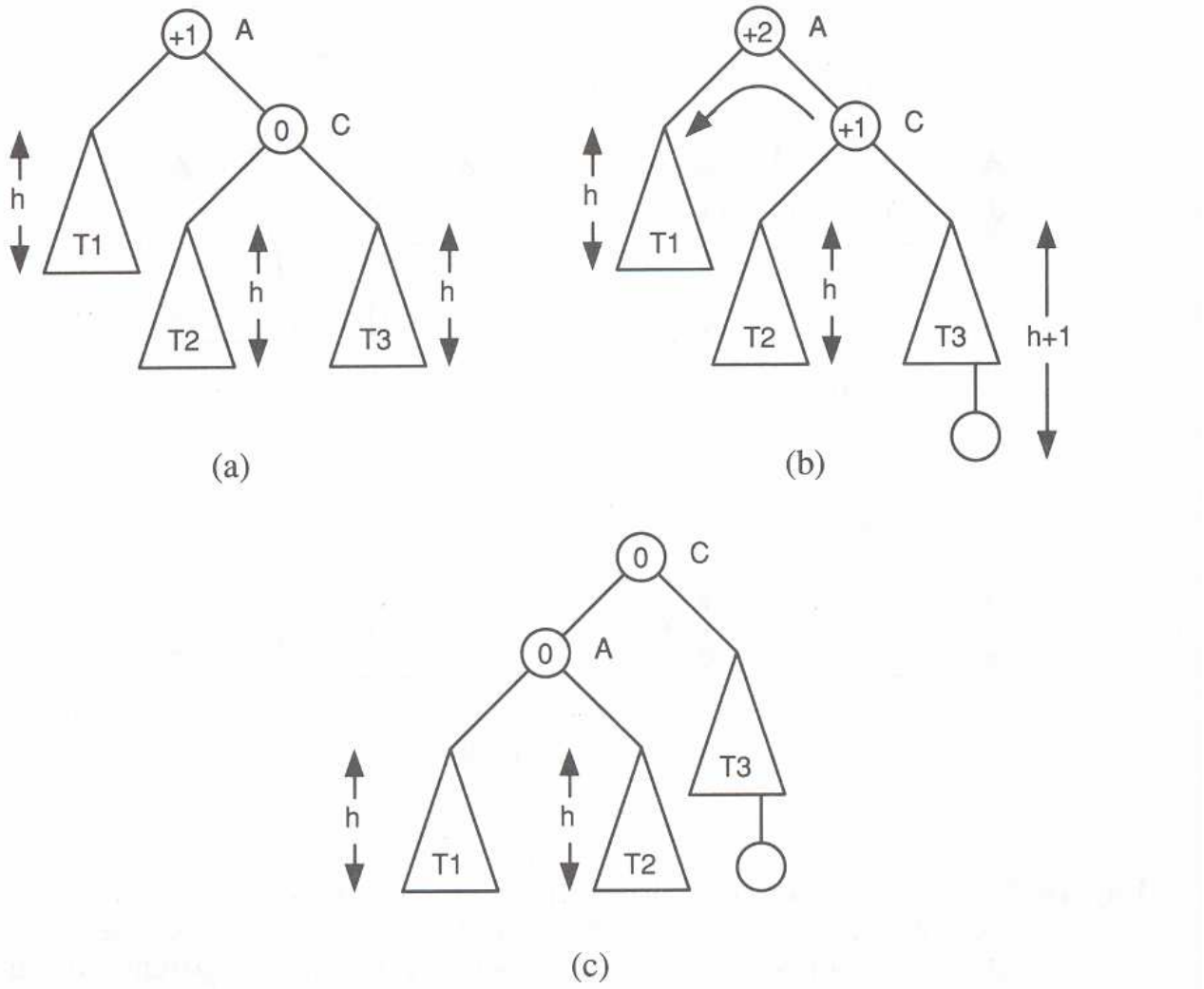
- Αν το balance κάποιου κόμβου αλλάζει σε +2 ή σε –2, ακολούθησε διαδικασία προσαρμογής του balance στο κατάλληλο εύρος: Η διαδικασία αυτή έχει σαν αποτέλεσμα το δένδρο να εξακολουθεί να είναι δυαδικό δένδρο αναζήτησης με τους ίδιους κόμβους και κλειδιά αλλά με όλους τους κόμβους να έχουν balance 0, 1, ή -1.

Εισαγωγές κόμβων σε AVL δένδρο
Παράδειγμα



Εισαγωγές κόμβων σε AVL δένδρο

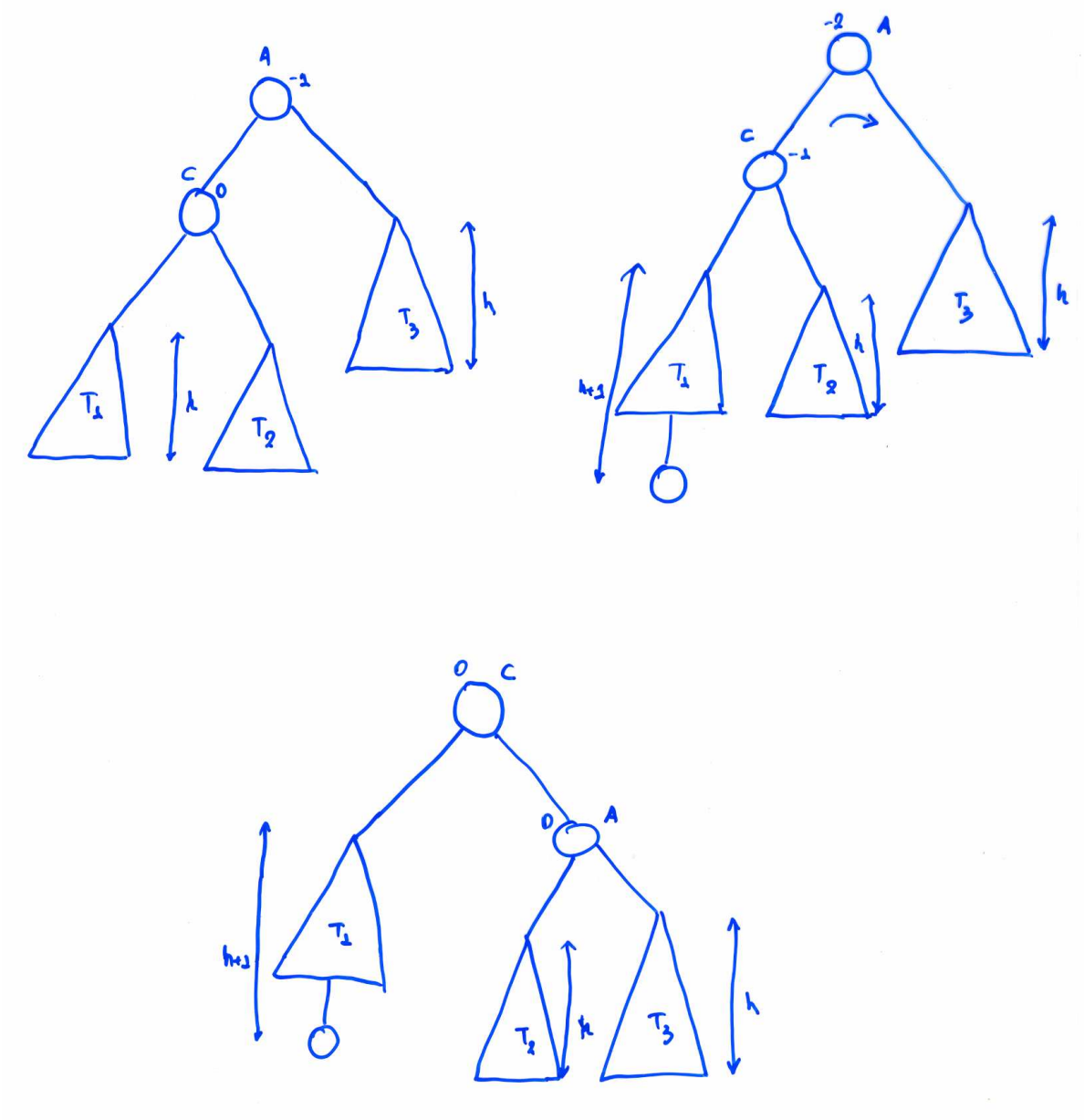
Single Left Rotation, RR (Απλή Αριστερή Περιστροφή)



- *Γιατί να μην ανταλλάξουμε το υποδένδρο Τ1 με το υποδένδρο Τ3 για να επιλύσουμε το πρόβλημα?*
- *Ποια είναι η συμμετρική περίπτωση (Single Right Rotation, LL)?*
- *Χρονική Πολυπλοκότητα?*

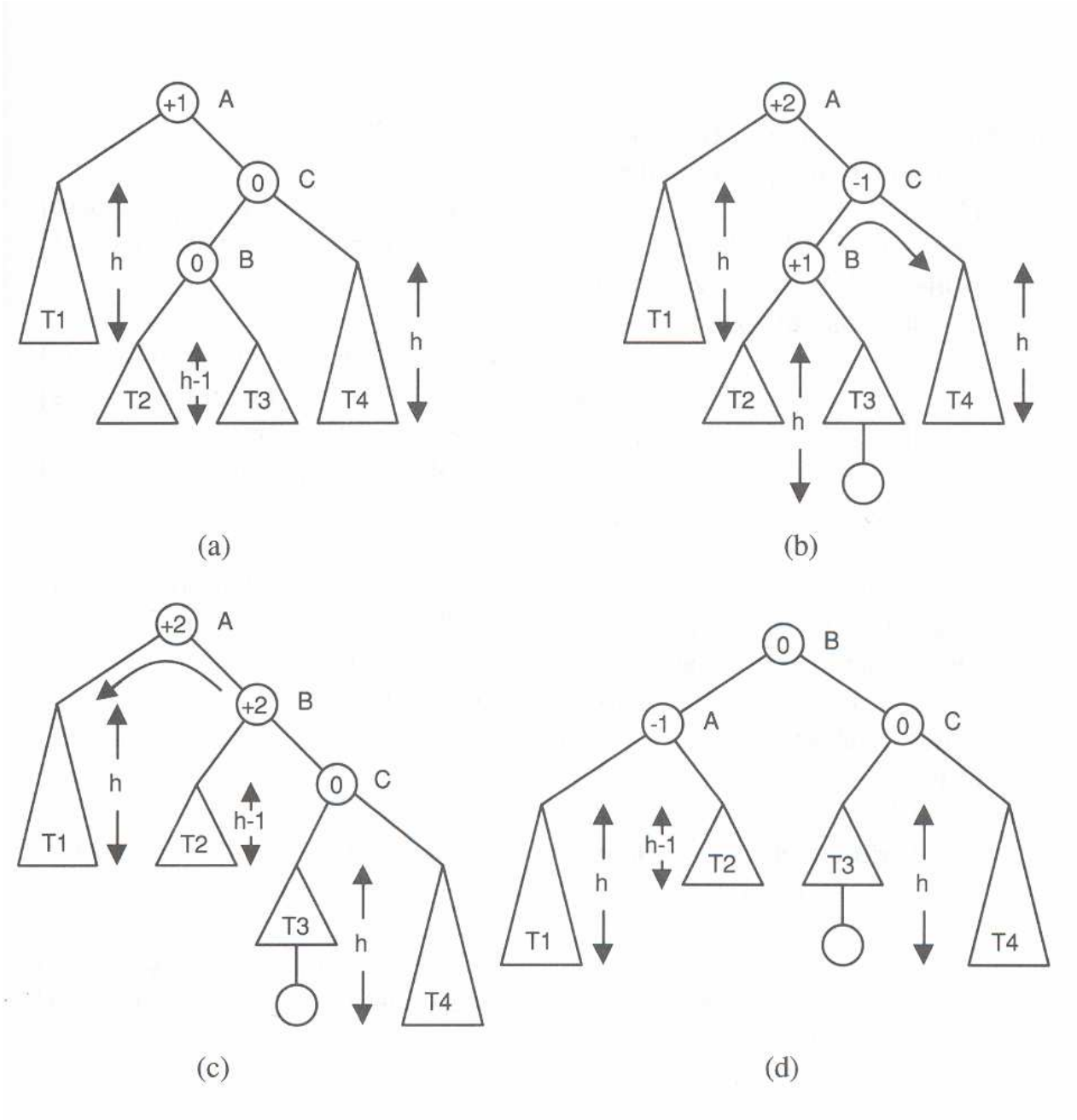
Εισαγωγές κόμβων σε AVL δένδρο

Single Right Rotation, LL (Απλή Δεξιά Περιστροφή)



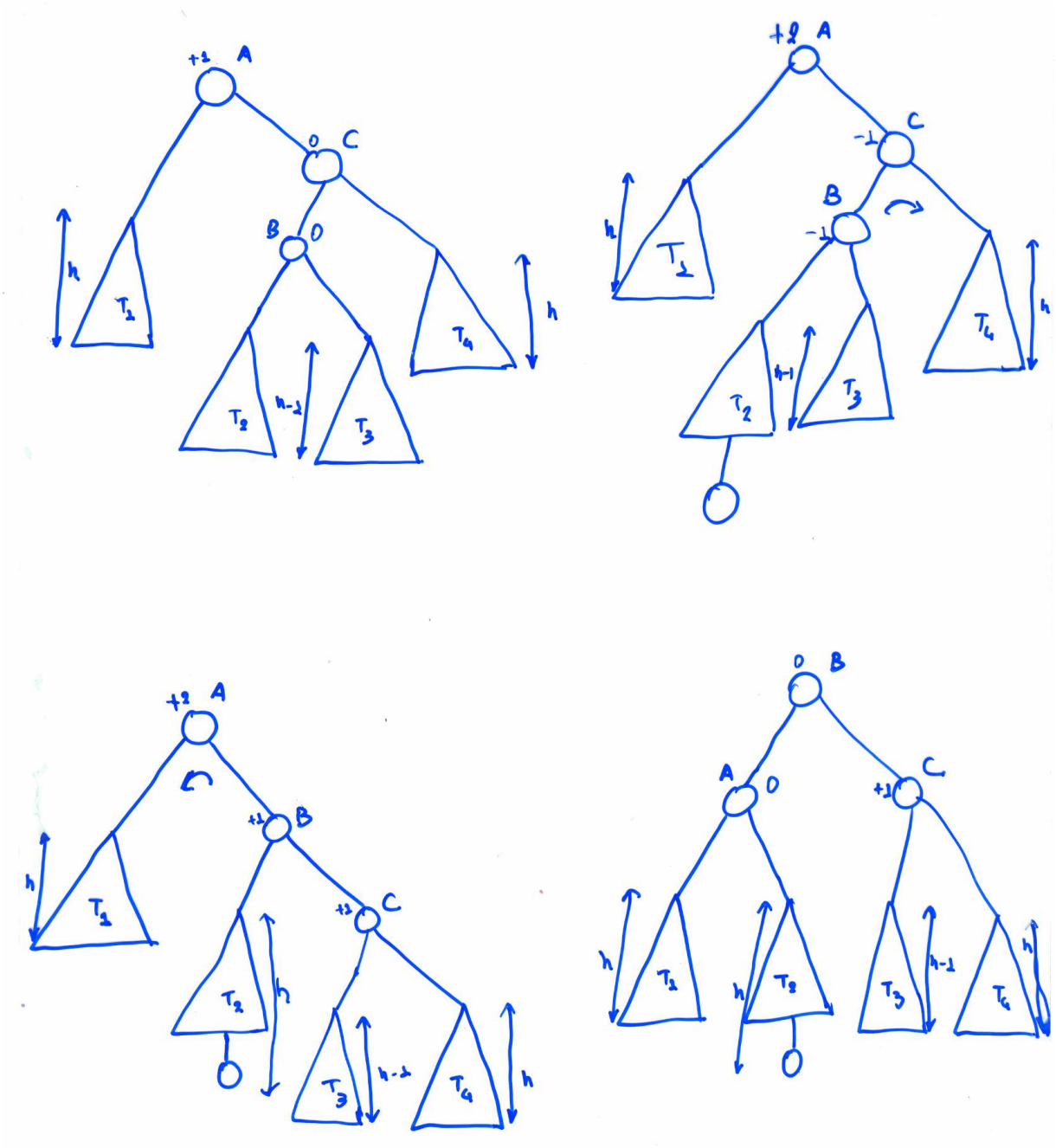
Εισαγωγές κόμβων σε AVL δένδρο

Double Rotation, RL (Right-Left): Case A



Εισαγωγές κόμβων σε AVL δένδρο

Double Rotation, RL (Right-Left): Case B

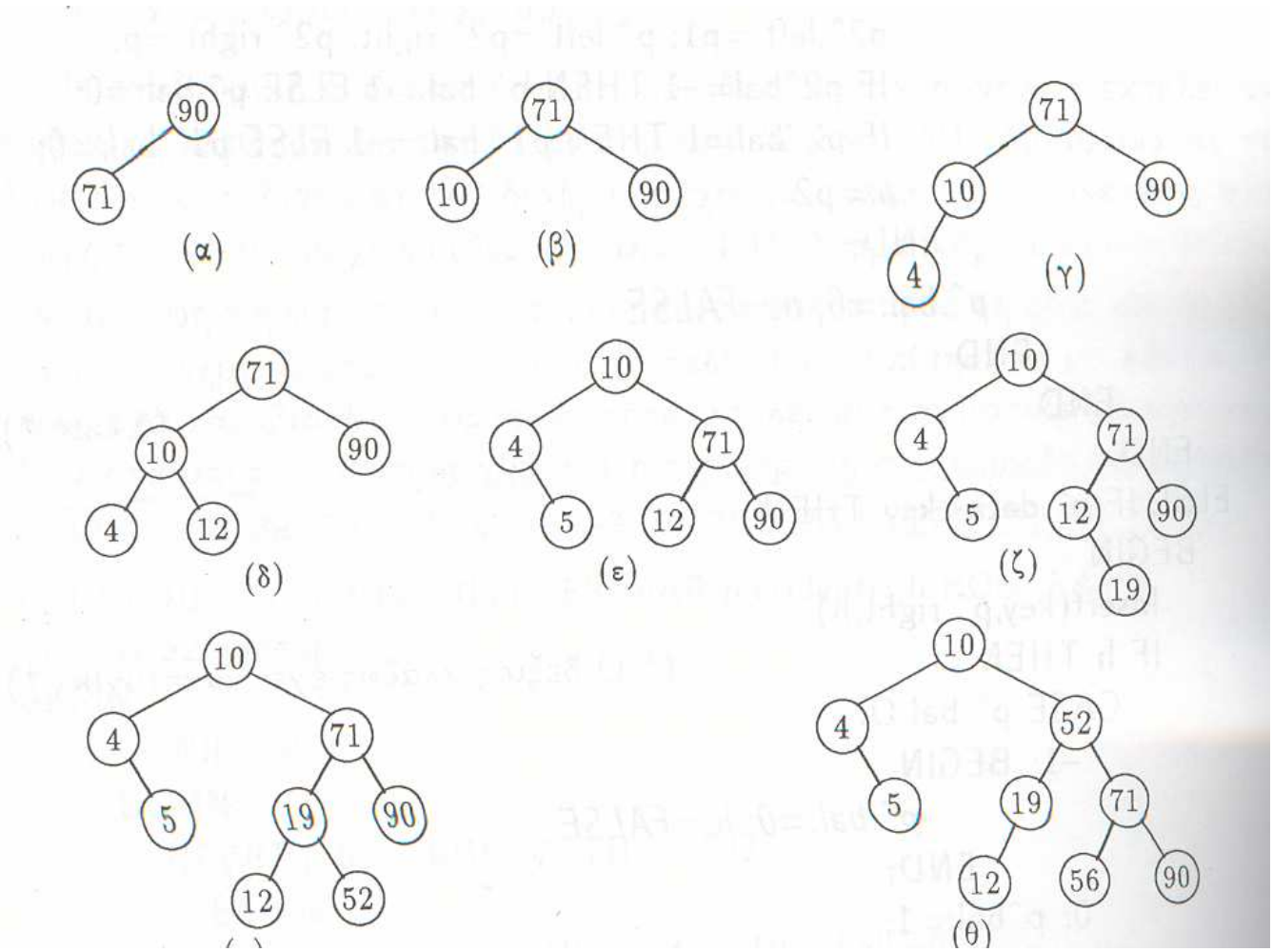


Υπάρχουν άλλες δύο περιπτώσεις (Left-Right: Case A & Case B). Να τις σχεδιάσετε.

Εισαγωγές κόμβων σε AVL δένδρο

Παράδειγμα

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



Κρίσιμοι Κόμβοι

Ο πρώτος κόμβος με balance +1 ή –1 στο μονοπάτι από το φύλλο στο οποίο θα γίνει η εισαγωγή του νέου κόμβου στη ρίζα, λέγεται κρίσιμος κόμβος (critical node).

Κάθε κόμβος από το φύλλο μέχρι τον κρίσιμο κόμβο είχε balance 0 οπότε θα έχει balance +1 ή –1 μετά την εισαγωγή.

Τι καθορίζει το αν κάποιος τέτοιος κόμβος θα έχει balance +1 ή –1?

Το balance του κρίσιμου κόμβου γίνεται είτε 0 ή +2 ή –2. *Γιατί? Πότε το balance γίνεται 0, πότε +2 και πότε –2?*

Παρατήρηση

Το ύψος του κρίσιμου κόμβου δεν αλλάζει, οπότε μετά τις περιστροφές, ο αντίστοιχος κόμβος έχει το ίδιο ύψος. Άρα, μόνο οι κόμβοι στο μονοπάτι από το φύλλο ως τον κρίσιμο κόμβο χρειάζονται να ελεγχθούν για αλλαγές στο balance τους. *Γιατί?*

Η περιστροφή γίνεται μόνο σε ένα σημείο που σχετίζεται με τον κρίσιμο κόμβο.

Σε τι μας βοηθάει η παρατήρηση αυτή?

Κρίσιμοι Κόμβοι

Καθώς ο αλγόριθμος κατεβαίνει προς το ζητούμενο φύλλο, αρκεί να θυμάται μόνο τον τελευταίο κρίσιμο κόμβο.

Μετά την εισαγωγή, ξεκινώντας από τον κρίσιμο κόμβο ακολούθησε και πάλι το ίδιο μονοπάτι προς τον εισερχόμενο κόμβο και διόρθωσε τα balances.

Τέλος, αν χρειάζεται κάνε τις περιστροφές.

Γιατί μπορούμε να βρούμε και πάλι αυτό το μονοπάτι?

Πως διορθώνουμε τα balances?

Ποια είναι η πολυπλοκότητα της Insert()?

Διαγραφές κόμβων σε AVL δένδρο

Αρχικά ακολουθούμε το γνωστό αλγόριθμο διαγραφής σε δυαδικό δένδρο:

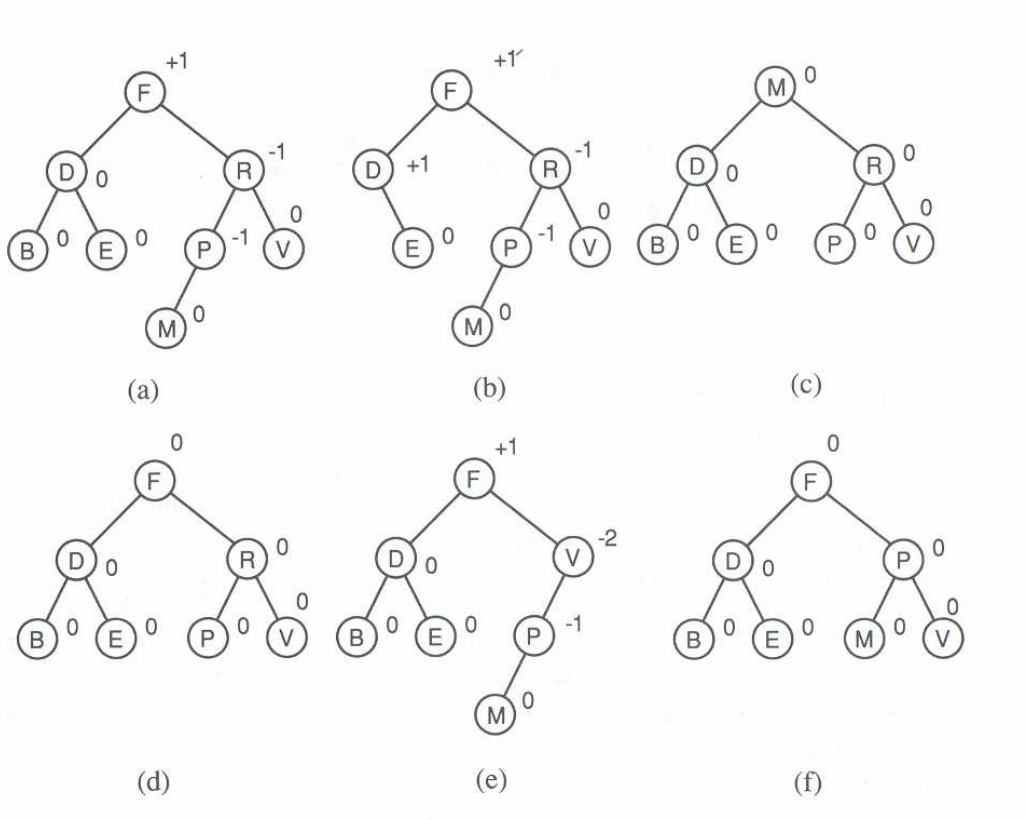
1) Διαγραφή του ίδιου του κόμβου v αν είναι φύλλο;

2) Αντικατάσταση του από το παιδί του αν έχει μόνο ένα παιδί;

3) Αντικατάσταση του από τον επόμενό του στην ενδοδιατεταγμένη διάταξη αν έχει 2 παιδιά.

Balance

Αν 1) ή 2), το balance του γονικού κόμβου του v αλλάζει.
Αν 3), το balance του επομένου του v στην ενδοδιατεταγμένη διάσχιση αλλάζει.



Διαγραφές κόμβων σε AVL δένδρο

- ✓ Αν το balance αλλάζει από 0 σε +1 ή σε –1, τότε ο αλγόριθμος τερματίζει.
- ✓ Αν το balance αλλάζει από +1 ή –1 σε 0 το ύψος του πατέρα του v μειώνεται και άρα και το balance άλλων προγόνων του v ενδεχομένως αλλάζει.
- ✓ Αν το Balance αλλάζει από +1 ή –1 σε +2 ή –2 τότε γίνεται μία ή περισσότερες περιστροφές. Μετά την περιστροφή το ύψος του δένδρου έχει μειωθεί και το balance προγόνων του κόμβου αλλάζει. Είναι δυνατόν να χρειαστεί να γίνουν περιστροφές σε όλους τους κόμβους που βρίσκονται στο μονοπάτι από τον κόμβο στη ρίζα (στη χειρότερη περίπτωση).

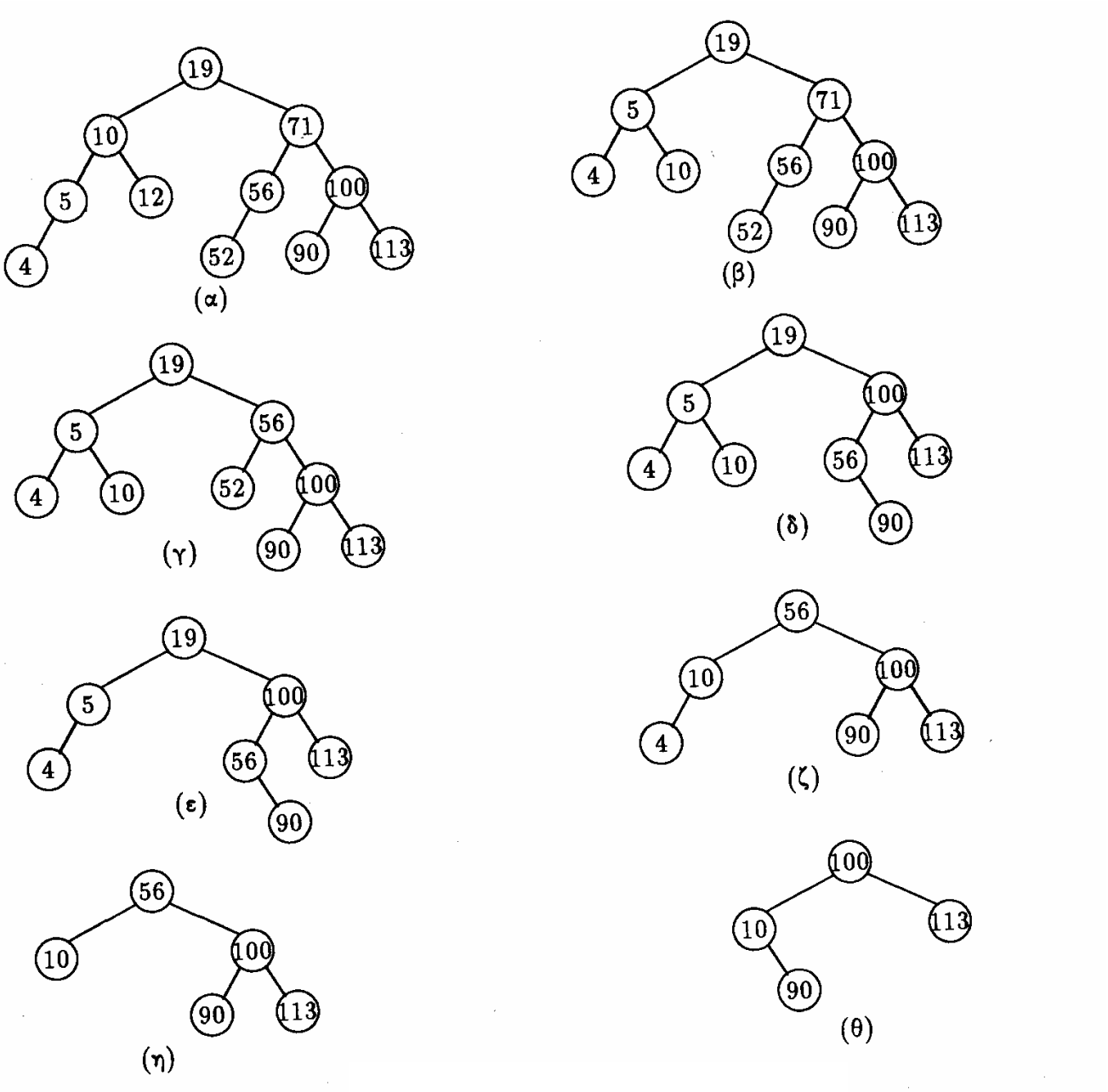
Ποια είναι η πολυπλοκότητα υλοποίησης της Delete()?

- ➢ Ολόκληρο το μονοπάτι αναζήτησης πρέπει να αποθηκευτεί, και να ακολουθηθεί ξανά μέχρι κάποιος κόμβος με balance 0 να βρεθεί. Το balance του κόμβου αυτού γίνεται +1 ή –1, αλλά το ύψος του δεν αλλάζει και από εκεί και στο εξής δεν χρειάζεται να γίνουν περαιτέρω περιστροφές.

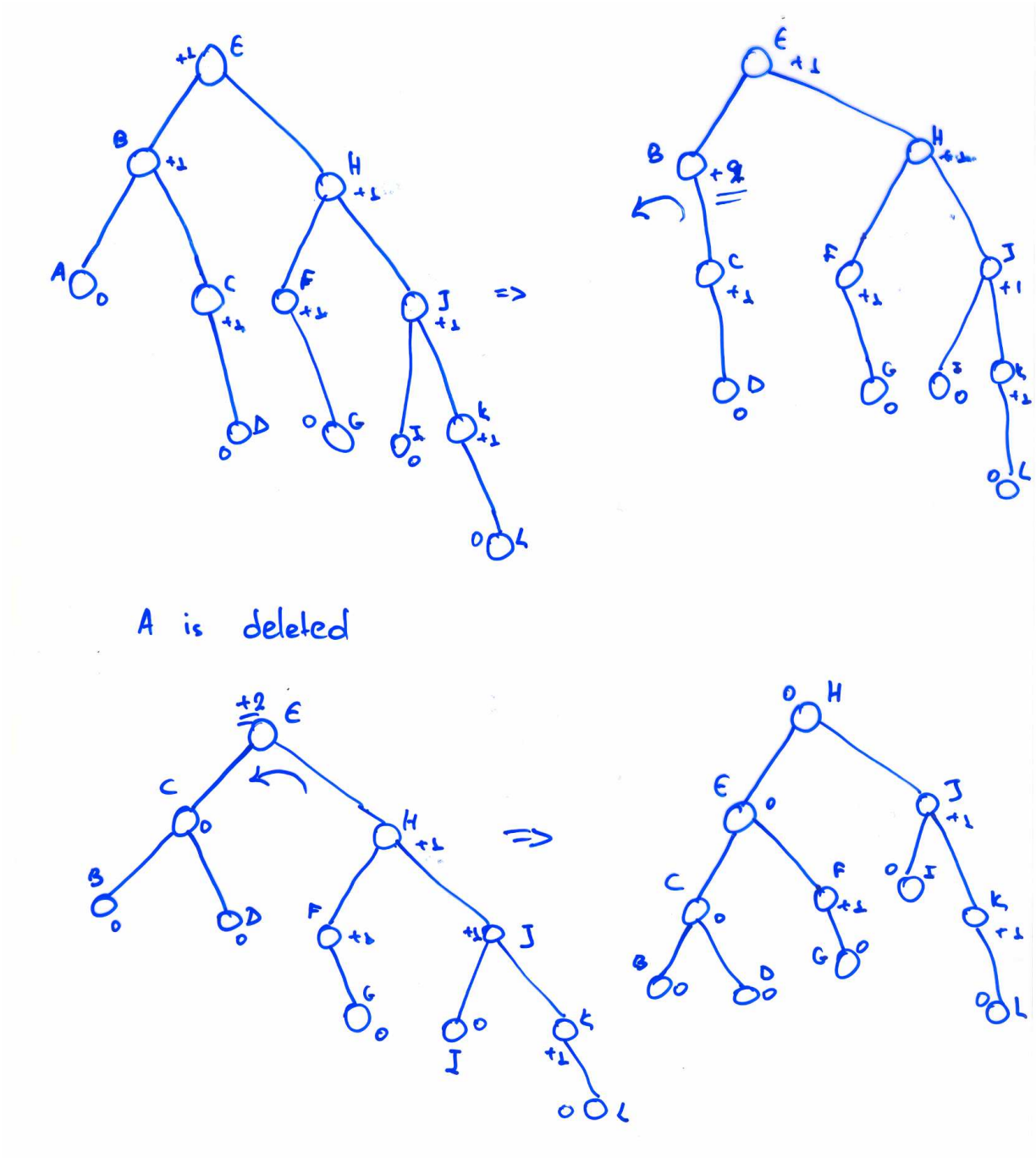
Διαγραφές κόμβων σε AVL δένδρο

Παράδειγμα

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



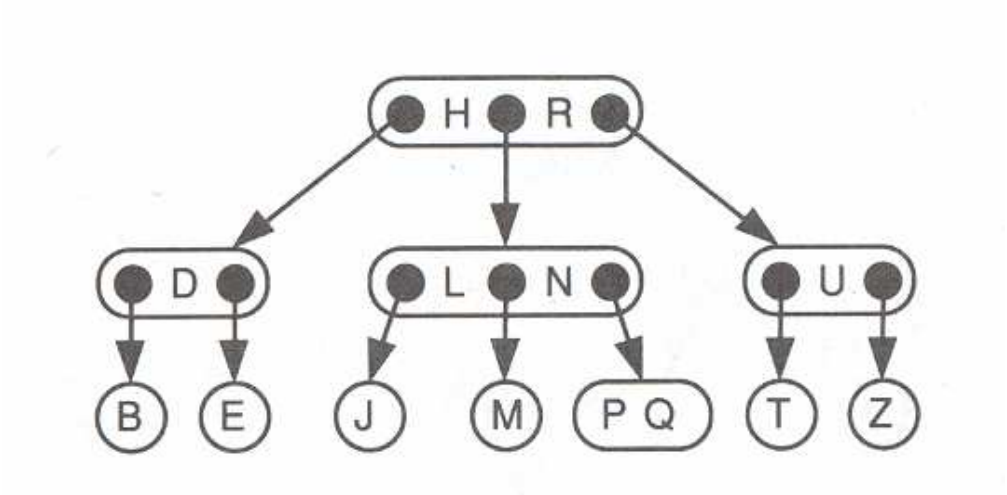
Παράδειγμα Διαγραφής που Οδηγεί σε Πολλαπλές Περιστροφές



2-3 Δένδρα

Ιδέα

Γιατί το δένδρο να μην είναι τέλεια εξισορροπημένο?



Σε ένα 2-3 δένδρο ένας κόμβος που δεν είναι φύλλο, έχει είτε δύο παιδιά (2-κόμβος), ή τρία παιδιά (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 περιέχει οποιονδήποτε επιθυμητό αριθμό φύλλων.

Σε ένα 2-3 δένδρο:

- Όλα τα φύλλα έχουν το ίδιο βάθος και περιέχουν 1 ή 2 κλειδιά.
- Κάθε εσωτερικός κόμβος:
  - είτε περιέχει ένα κλειδί και έχει δύο παιδιά,
  - ή περιέχει δύο κλειδιά και έχει τρία παιδιά.

Το δένδρο είναι δένδρο αναζήτησης.

2-3 Δένδρα

Μεταξύ όλων των 2-3 δένδρων ύψους h, ποιο είναι εκείνο με τους λιγότερους κόμβους?

Ποιο είναι το ύψος αυτού του δένδρου?

Ποιο είναι το μεγαλύτερο 2-3 δένδρο ύψους h?

Ποιο είναι το ύψος αυτού του δένδρου?

Το ύψος ενός 2-3 δένδρου με n κόμβους είναι Θ(logn).

Πως θα υλοποιήσουμε την LookUp() σε ένα 2-3 δένδρο?

Ποια θα είναι η πολυπλοκότητά της?

Εισαγωγή σε 2-3 Δένδρο

Ιδέα

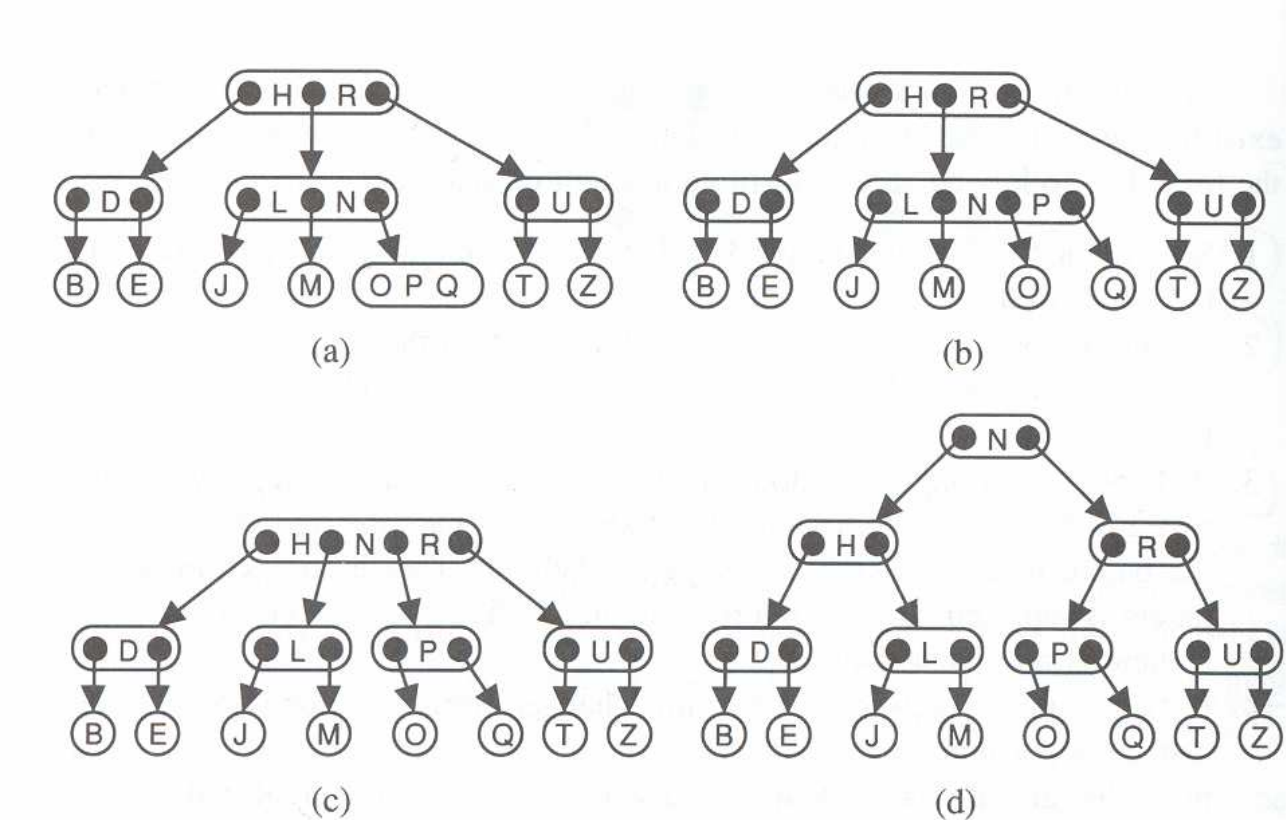
Προσπάθεια εκμετάλευσης του extra χώρου που υπάρχει στα φύλλα με μόνο ένα κλειδί.

Αλγόριθμος

1. Εύρεση του φύλλου στο οποίο «ανήκει» το κλειδί. Διατήρηση του μονοπατιού που ακολουθήθηκε.
2. Αν υπάρχει χώρος στον κόμβο (είναι 2-κόμβος, δηλαδή έχει μόνο ένα κλειδί), πρόσθεσε το κλειδί και τερμάτισε.
3. Αν δεν υπάρχει χώρος (ο κόμβος είναι ήδη 3-κόμβος), χώρισε τον κόμβο σε δύο 2-κόμβους, έναν με το πρώτο και έναν με το τρίτο κλειδί και πέρασε το μεσαίο κλειδί στον πατέρα του αρχικού κόμβου για να αποθηκευτεί εκεί (το παιδί ενός κόμβου αντικαθίσταται από 2 παιδιά και ένα κλειδί). Αν δεν υπάρχει πατρικός κόμβος πήγαινε στο βήμα 5.
4. Αν ο πατέρας είναι 2-κόμβος, μετατρέπεται σε 3-κόμβο και ο αλγόριθμος τερματίζει. Διαφορετικά, επιστρέφουμε στο βήμα 3 για να χωρίσουμε τον πατέρα 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, οπότε και η ρίζα χωρίζεται σε 2 κόμβους και το ύψος του δένδρου αυξάνει.

Εισαγωγή σε 2-3 δένδρο: Παράδειγμα

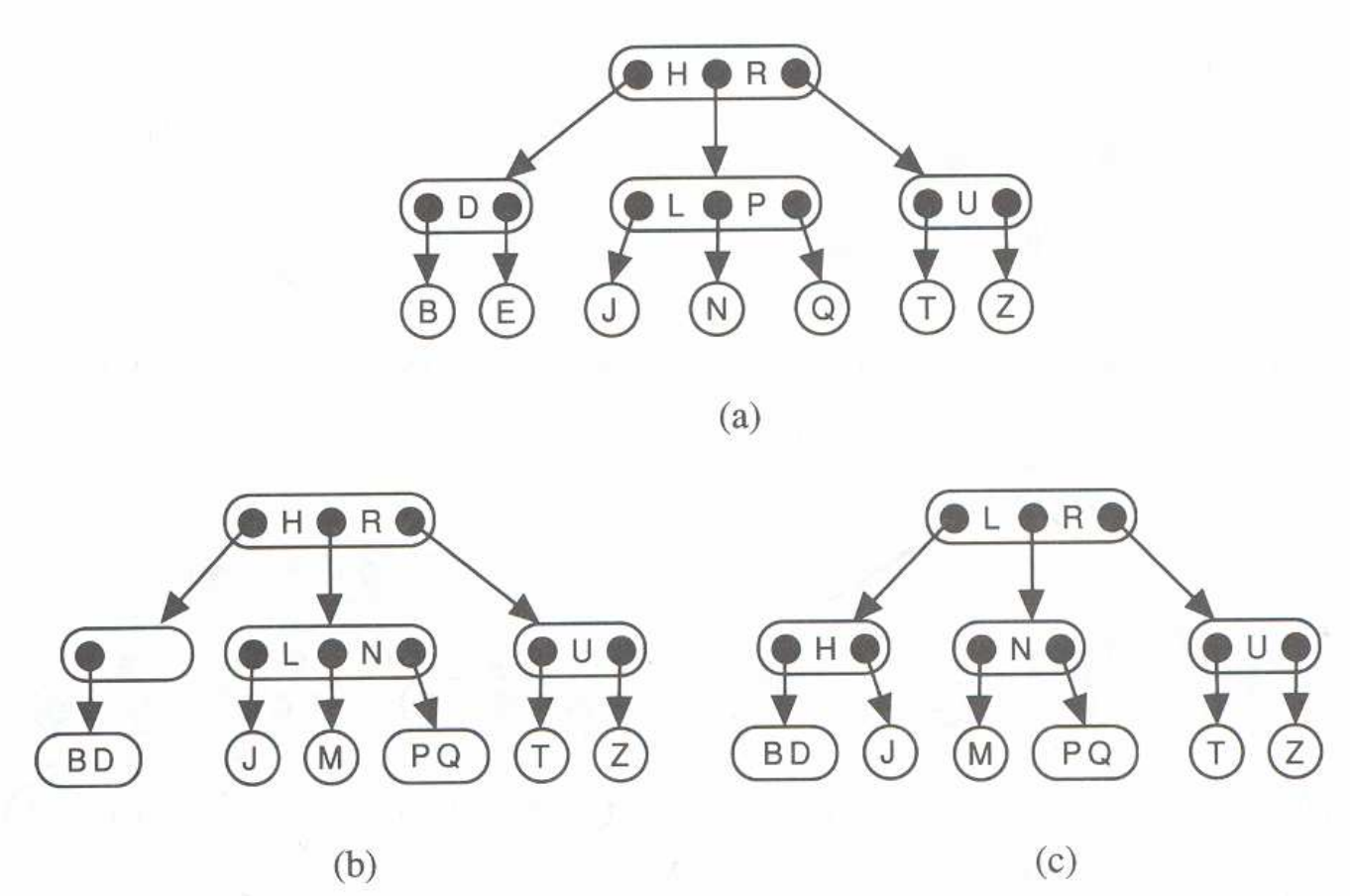
Εισαγωγή του κλειδιού Ο στο δένδρο.



Διαγραφή σε 2-3 δένδρο

Αντίστροφο πρόβλημα: κάποιοι κόμβοι μπορεί να μένουν χωρίς καθόλου κλειδιά.

Όταν αυτό συμβαίνει, αν ο κόμβος έχει κάποιον αδελφικό κόμβο με δύο κλειδιά, μπορούμε να μεταφέρουμε ένα κλειδί από τον αδελφικό κόμβο για να επιλύσουμε το πρόβλημα.



Διαγραφή σε 2-3 δένδρο

Αλγόριθμος

1. Αν το προς διαγραφή κλειδί περιέχεται σε κόμβο φύλλο, διέγραψέ το. Αν όχι, τότε το επόμενο του κλειδιού στην ενδο-διατεταγμένη διάσχιση περιέχεται σε κόμβο φύλλο, οπότε αντικατέστησε το κλειδί με το επόμενο και διέγραψε το επόμενο.
2. Έστω N ο κόμβος από τον οποίο διαγράφεται το κλειδί. Αν ο N εξακολουθεί να έχει ένα κλειδί, ο αλγόριθμος τερματίζει. Διαφορετικά, ο N δεν έχει κανένα κλειδί:
   a. Αν ο N είναι η ρίζα, διέγραψέ τον. Ο N μπορεί να έχει ή κανένα ή 1 παιδί. Αν δεν έχει παιδί, το δένδρο γίνεται άδειο. Διαφορετικά, το παιδί του N γίνεται ρίζα.
   b. (Ο N έχει τουλάχιστον έναν αδελφικό κόμβο. *Γιατί*?) Αν ο N έχει έναν αδελφικό 3-κόμβο N' ακριβώς στα αριστερά του ή ακριβώς στα δεξιά του, έστω S το κλειδί του πατρικού κόμβου των N, N' που χωρίζει τα δύο υποδένδρα. Μετακινούμε το S στο N, και το αντικαθιστούμε στον πατέρα του με το N'. Αν N και N' είναι εσωτερικοί κόμβοι, αλλάζουμε και ένα κατάλληλο παιδί του N' σε παιδί του N. Τα N, N' έχουν από ένα κλειδί το καθένα, αντί 0 και 2, και ο αλγόριθμος τερματίζει.
   c. (Ο N έχει έναν αδελφικό κόμβο N', αλλά με ένα μόνο κλειδί.) Έστω P ο πατέρας των N, N' και S το κλειδί που χωρίζει τους N, N' στον P. Συνενώνουμε το S και το κλειδί του N' σε έναν νέο 3-κόμβο, οποίος αντικαθιστά τα N, N'. Θέτουμε N = P και επαναλαμβάνουμε το βήμα 2.

Διαγραφή σε 2-3 δένδρο: Παράδειγμα

Κόκκινα-Μαύρα Δένδρα (Red-Black Trees)

Ένα κόκκινο-μαύρο δένδρο είναι ένα δυαδικό δένδρο αναζήτησης στο οποίο οι κόμβοι μπορούν να χαρακτηρίζονται από ένα εκ των δύο χρωμάτων: μαύρο-κόκκινο.

Το χρώμα της ρίζας είναι πάντα μαύρο. Αν κάποιο παιδί κάποιου κόμβου δεν υπάρχει, ο αντίστοιχος δείκτης είναι NIL. Θα θεωρήσουμε αυτούς τους δείκτες NIL ως δείκτες σε εξωτερικούς NIL κόμβους (φύλλα) του δυαδικού δένδρου και τους κανονικούς κόμβους που αποθηκεύουν κλειδιά ως εσωτερικούς κόμβους του δένδρου.

Ένα κόκκινο-μαύρο δένδρο πληροί τις ακόλουθες ιδιότητες:

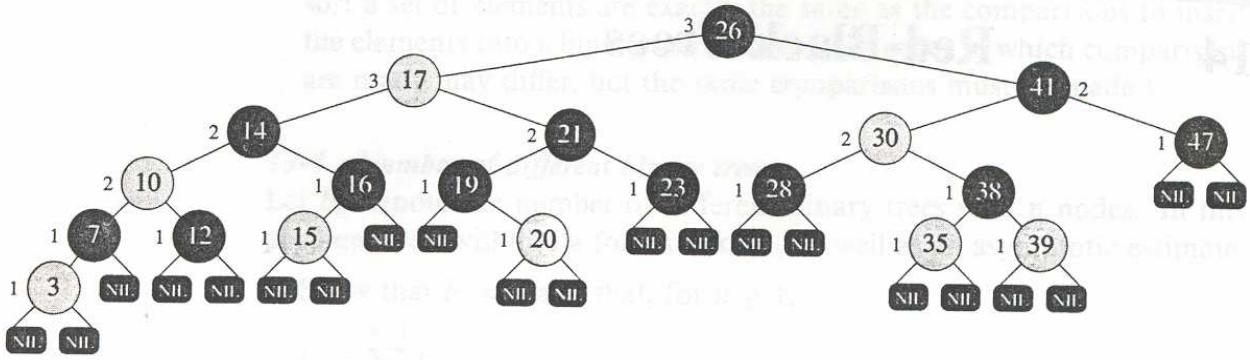
- Κάθε κόμβος έχει είτε κόκκινο είτε μαύρο χρώμα.
- Κάθε NIL κόμβος έχει μαύρο χρώμα.
- Κάθε απλή διαδρομή από έναν κόμβο σε έναν NIL κόμβο που είναι απόγονός του περιέχει το ίδιο πλήθος μαύρων κόμβων.
- Αν ένας κόμβος έχει κόκκινο χρώμα, τότε και τα δύο παιδιά του έχουν μαύρο χρώμα.

Μέγεθος Κόκκινων-Μαύρων Δένδρων

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, bh(v), να είναι ο αριθμός των μαύρων κόμβων σε κάθε μονοπάτι από τον κόμβο v σε οποιοδήποτε φύλλο, χωρίς να συμπεριλαμβάνουμε τον v.
- ✓ Αποδεικνύουμε ότι το υπο-δένδρο με ρίζα κάποιο κόμβο v περιέχει τουλάχιστον $2^{bh(v)}-1$ εσωτερικούς κόμβους με επαγωγή ως προς το ύψος του v. Στη βάση επαγωγής, αν ο v έχει ύψος 0 τότε είναι φύλλο (ο κόμβος φρουρός), οπότε ο αριθμός των εσωτερικών κόμβων στο υποδένδρο που εκφύεται από τον v είναι 0 όπως απαιτείται. Δοθέντος ενός κόμβου v, υποθέτουμε ότι ο ισχυρισμός ισχύει για όλους τους απογόνους του v και τον αποδεικνύουμε για τον v.
- ✓ Έστω h το ύψος του δένδρου και r η ρίζα. Είναι $bh(r) \geq h/2$. Άρα, $n \geq 2^{h/2} - 1 \Rightarrow h \leq 2\log(n+1)$.

Πως υλοποιούμε τη LookUp()? Πολυπλοκότητα?

Εισαγωγή σε Κόκκινο-Μαύρο Δένδρο

Πως κάνουμε εισαγωγή ενός κόμβου x σε κόκκινο-μαύρο δένδρο? Τι πρόβλημα δημιουργείται?

Αλγόριθμος

- Εισαγωγή όπως σε δυαδικό δένδρο αναζήτησης.
- Το μονοπάτι αποθηκεύεται σε στοίβα.
- Ο νέος κόμβος που εισάγεται χαρακτηρίζεται κόκκινος.
- Εξετάζουμε αν οι ιδιότητες χρωματισμού εξακολουθούν να ισχύουν. Αν ναι τερματίζουμε, διαφορετικά διορθώνουμε.

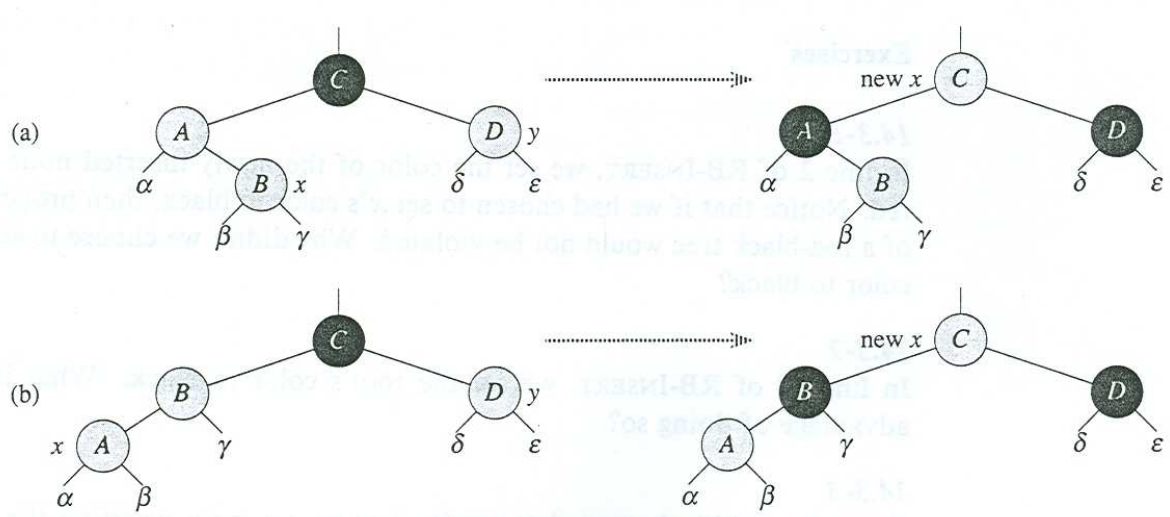
- *Ο πατρικός κόμβος πρέπει να είναι κόκκινος. Γιατί; Ο κόμβος δεν μπορεί να είναι η ρίζα.*
- *Ο παππούς πρέπει να είναι μαύρος. Γιατί?*

Εισαγωγή σε Κόκκινο-Μαύρο Δένδρο

Αλγόριθμος (συνέχεια)

- Υποθέτουμε ότι ο πατέρας του x είναι αριστερό παιδί. Διακρίνουμε περιπτώσεις:

1. **Περίπτωση 1:** *Αν ο θείος (αδελφικός κόμβος πατέρα) είναι κόκκινος*, αλλάζουμε το χρώμα του πατέρα και του θείου σε μαύρο και το χρώμα του παππού σε κόκκινο. Επαναλαμβάνουμε.

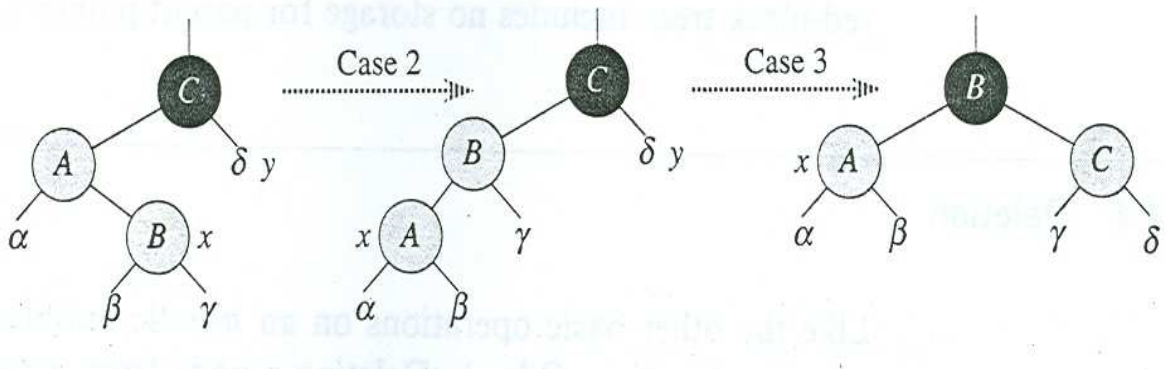


Εισαγωγή σε Κόκκινο-Μαύρο Δένδρο

Αλγόριθμος (συνέχεια)

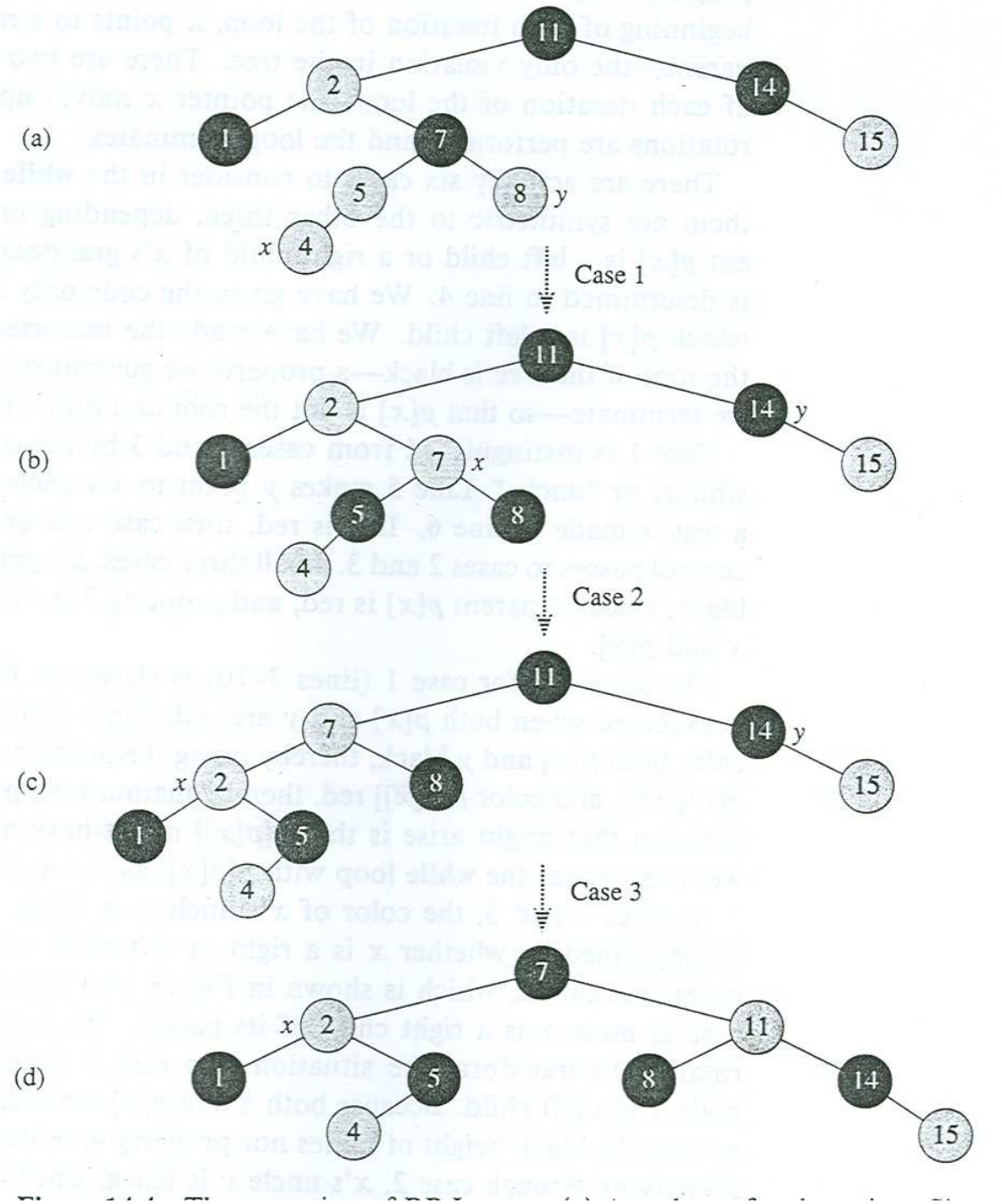
2. **Περίπτωση 2:** Ο θείος είναι μαύρος και ο x είναι δεξί παιδί του πατέρα του. Ανάγουμε την περίπτωση αυτή στην περίπτωση 3 με την εκτέλεση μιας αριστερής περιστροφής.

3. **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.



- *Ποια είναι η πολυπλοκότητα της RB-Insert()?*

Εισαγωγή σε Κόκκινο-Μαύρο Δένδρο: Παράδειγμα



Διαγραφή από κόκκινο-μαύρο δένδρο

Η διαγραφή γίνεται με παρόμοιο τρόπο όπως σε δυαδικό δένδρο αναζήτησης και στη συνέχεια ελέγχεται αν οι ιδιότητες χρωματισμού ισχύουν ή όχι και γίνονται κατάλληλες ενέργειες.

Έστω y ο κόμβος που θα διαγραφεί από το δένδρο.

Αν y κόκκινος, δεν υπάρχει πρόβλημα. *Γιατί*?

Αν y μαύρος, η διαγραφή του δημιουργεί (τουλάχιστον) ένα μονοπάτι με μαύρο ύψος μικρότερο κατά ένα.

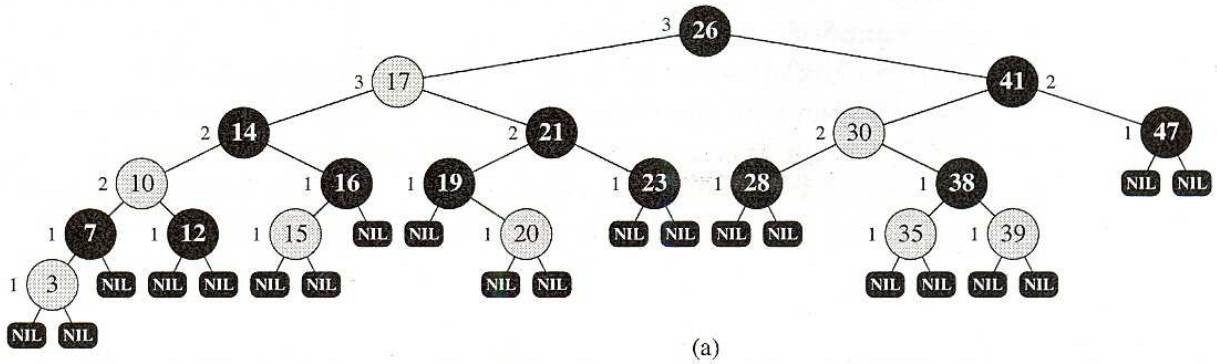
Υποθέτουμε ότι η μαύρη ιδιότητα του v μεταφέρεται στο παιδί του, το οποίο τώρα γίνεται διπλά μαύρο (που είναι μη επιτρεπτό).

Πρέπει να μεταφέρουμε το extra μαύρο προς τα πάνω στο δένδρο μέχρι είτε:

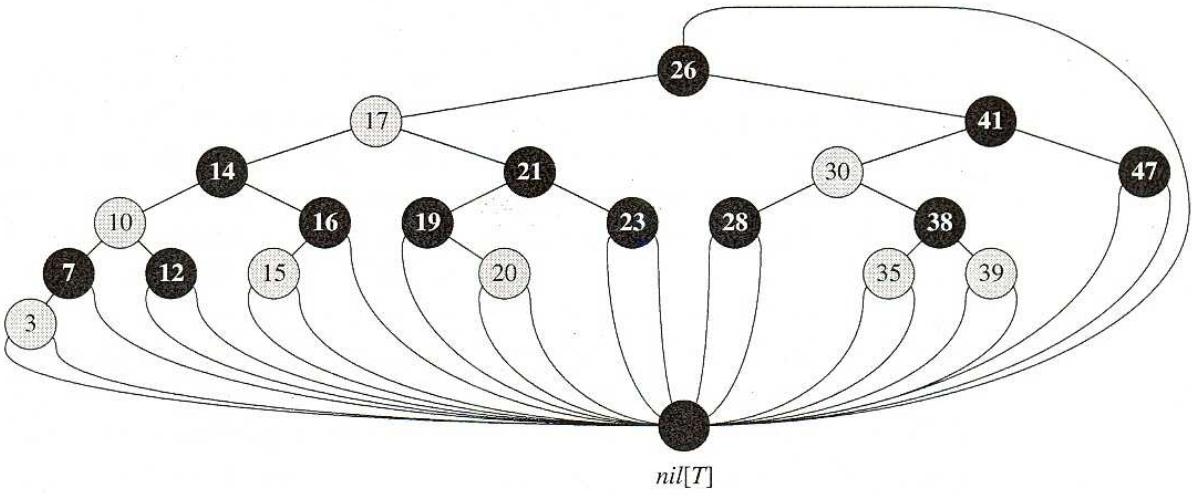
- να φθάσουμε στη ρίζα, ή
- να βρούμε έναν κόκκινο κόμβο που τον μετονομάζουμε σε μαύρο και τερματίζουμε, ή
- να μπορούν να εκτελεστούν κατάλληλες περιστροφές και επαναχρωματισμοί κάποιων κόμβων ώστε να λυθεί το πρόβλημα.

Διαγραφή από κόκκινο-μαύρο δένδρο - Περιπτώσεις

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

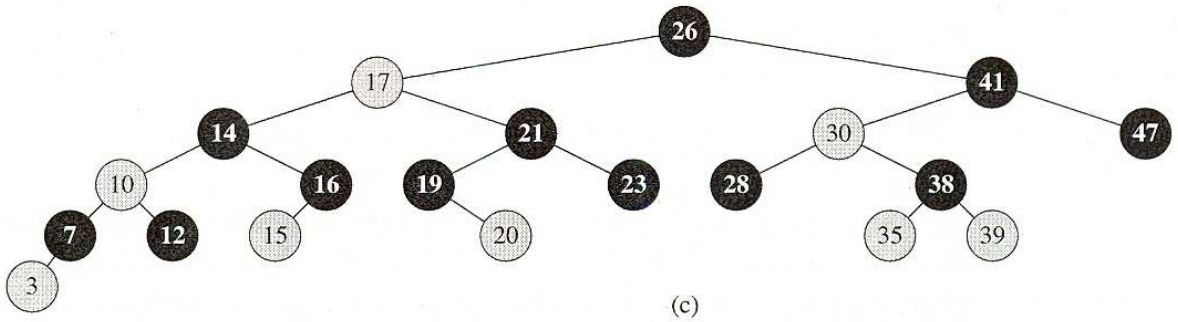


(a)



nil[*T*]

(b)



(c)

- Έστω x το παιδί του κόμβου που διαγράφεται.

Διαγραφή από κόκκινο-μαύρο δένδρο - Περιπτώσεις

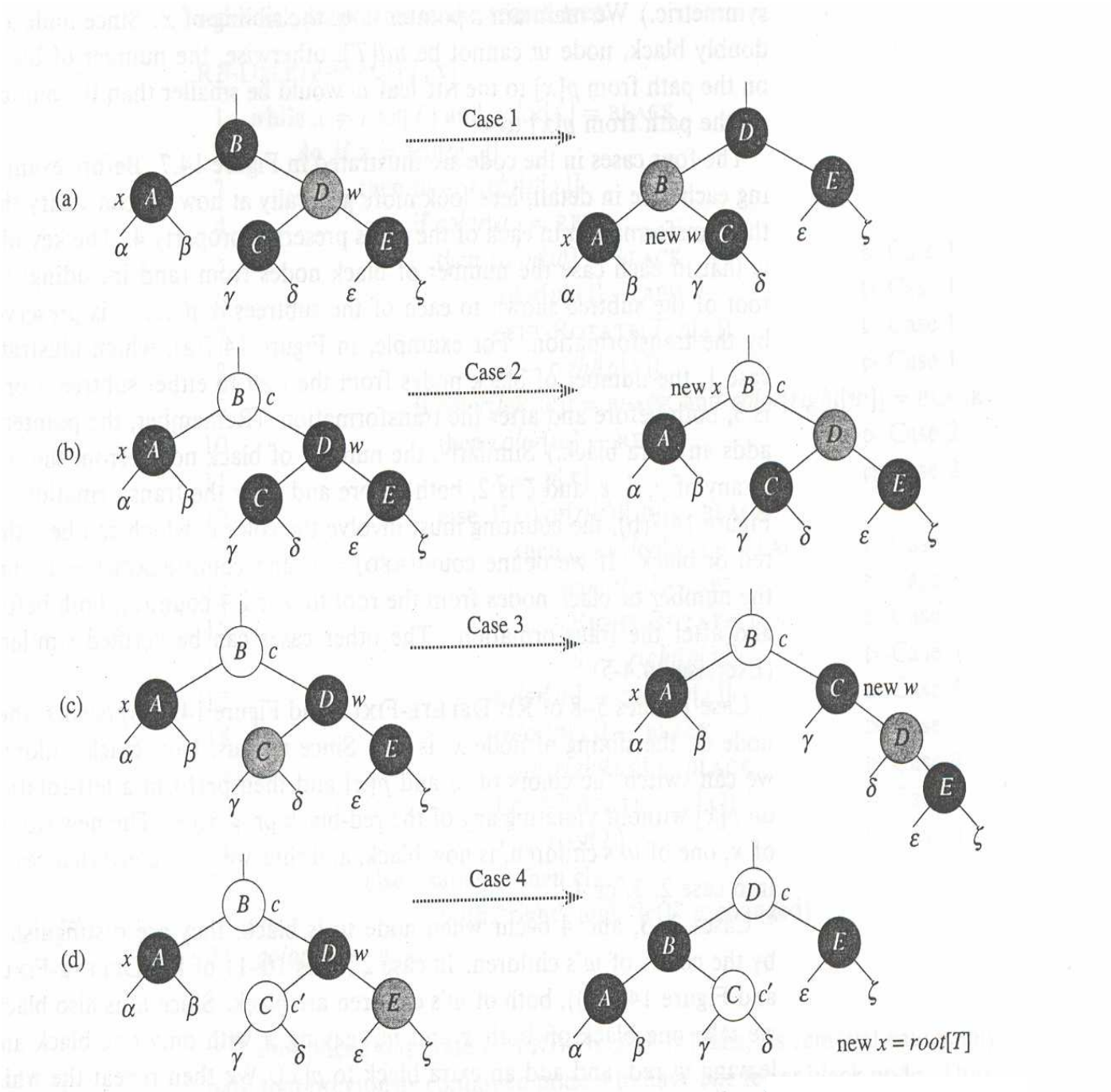
Έστω w ο αδελφικός κόμβος & p ο πατέρας του x. Ο w δεν μπορεί να είναι ο κόμβος φρουρός. *Γιατί;*

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

Διακρίνουμε περιπτώσεις ως προς το χρώμα του w.

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 (περίπτωση α. σχήματος). Έτσι, μεταπίπτουμε στην περίπτωση 2.
2. Ο w είναι μαύρος κόμβος.
    a. 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. 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. 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 το p. Ο αλγόριθμος τερματίζει.

Διαγραφή από κόκκινο-μαύρο δένδρο - Περιπτώσεις



Ποια είναι η πολυπλοκότητα της RB-Delete()?

Splay Δένδρα

Είναι απλά δυαδικά δένδρα αναζήτησης. Κάθε κόμβος έχει τα εξής πεδία: Key, Info, LC, RC.

Διαφορά από δυαδικά δένδρα αναζήτησης
Τρόπος υλοποίησης LookUp(), Insert() και Delete().

Αν το λεξικό έχει n στοιχεία, δεν είναι εγγυημένο ότι οι αλγόριθμοι αυτοί έχουν πολυπλοκότητα O(log n).

Υπάρχουν μόνο εγγυήσεις για το amortized κόστος κάθε λειτουργίας:
Κάθε ακολουθία από m λειτουργίες, ξεκινώντας από ένα άδειο δένδρο, χρειάζεται O(m logn) χρόνο για την εκτέλεσή της.

Άρα:
Το amortized κόστος κάθε λειτουργίας είναι O(logn).

Όμως:
Μπορεί να υπάρχουν λειτουργίες που το κόστος τους είναι πολύ υψηλό, π.χ., Ω(n), αλλά αυτό συμβαίνει μόνο αν αυτές έπονται πολλών λειτουργιών με κόστος πολύ μικρό.

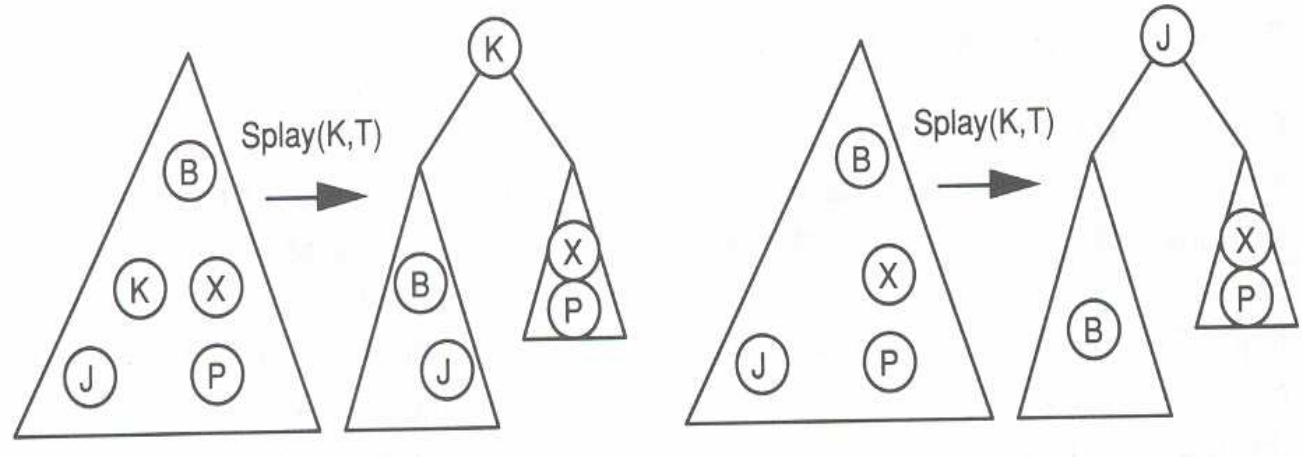
Splay Δένδρα

Ιδέα (κοντινή σε ευριστικό Move-To-Front)

Κάθε φορά που ένα κλειδί είναι το αποτέλεσμα μιας επιτυχημένης αναζήτησης στο δένδρο, ο κόμβος του μετακινείται στη ρίζα.

Κρίσιμη Λειτουργία

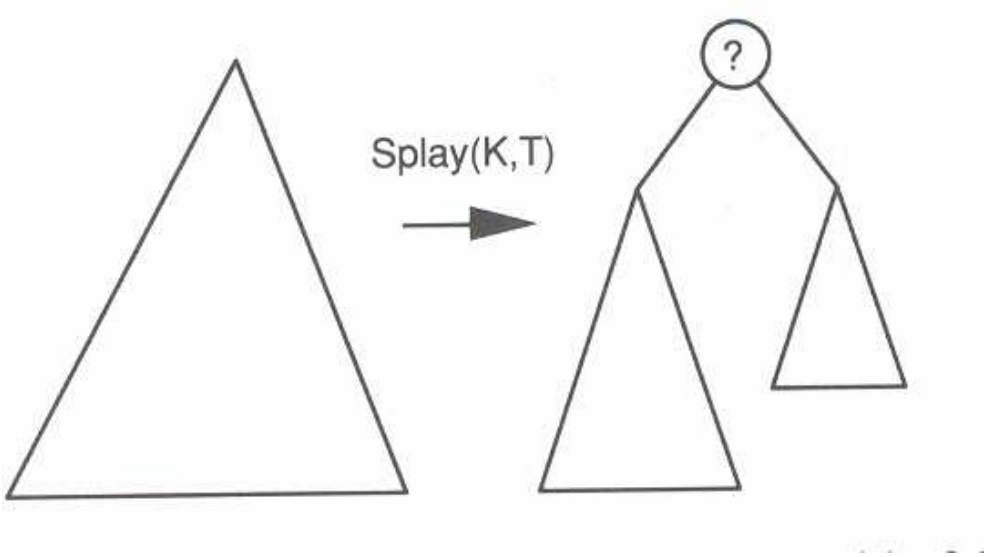
Splay(K,T), K -> κλειδί, T -> δένδρο: τροποποιεί το T έτσι ώστε το προκύπτον δένδρο (1) είναι επίσης δυαδικό δένδρο αναζήτησης, και (2) έχει το κλειδί K στη ρίζα, αν το K υπάρχει στο δένδρο. Αν όχι, η ρίζα περιέχει το κλειδί που θα ήταν ο επόμενος ή ο προηγούμενος του κλειδιού στην ενδοδιατεταγμένη διάσχιση.



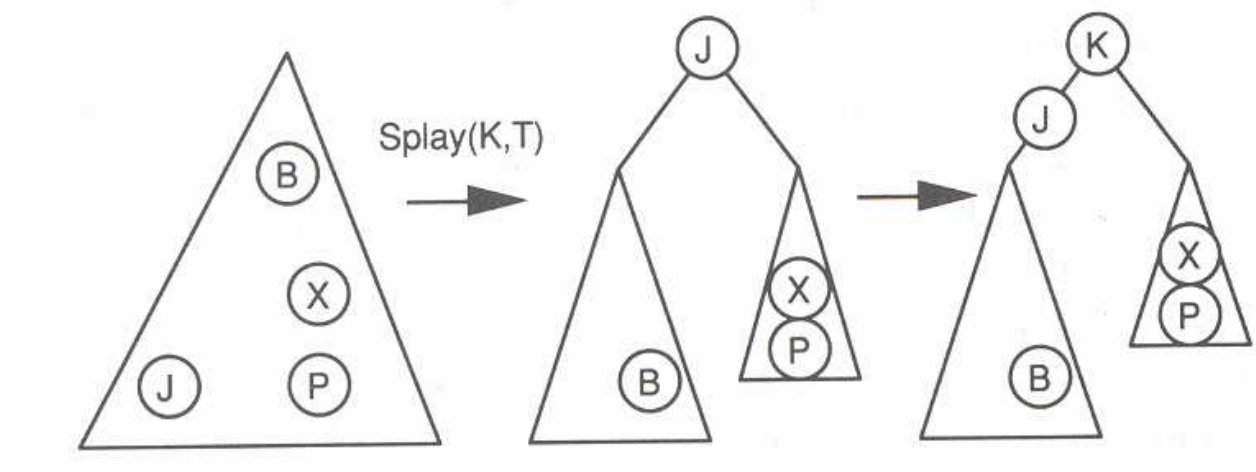
Τα δένδρα των οποίων η διαχείριση γίνεται βάσει της Splay λειτουργίας λέγονται **Splay δένδρα**.

Υλοποίηση Λειτουργιών Splay Δένδρων

LookUp(K,T): Εκτελούμε τη λειτουργία Splay(K,T) και εξετάζουμε το κλειδί της ρίζας.

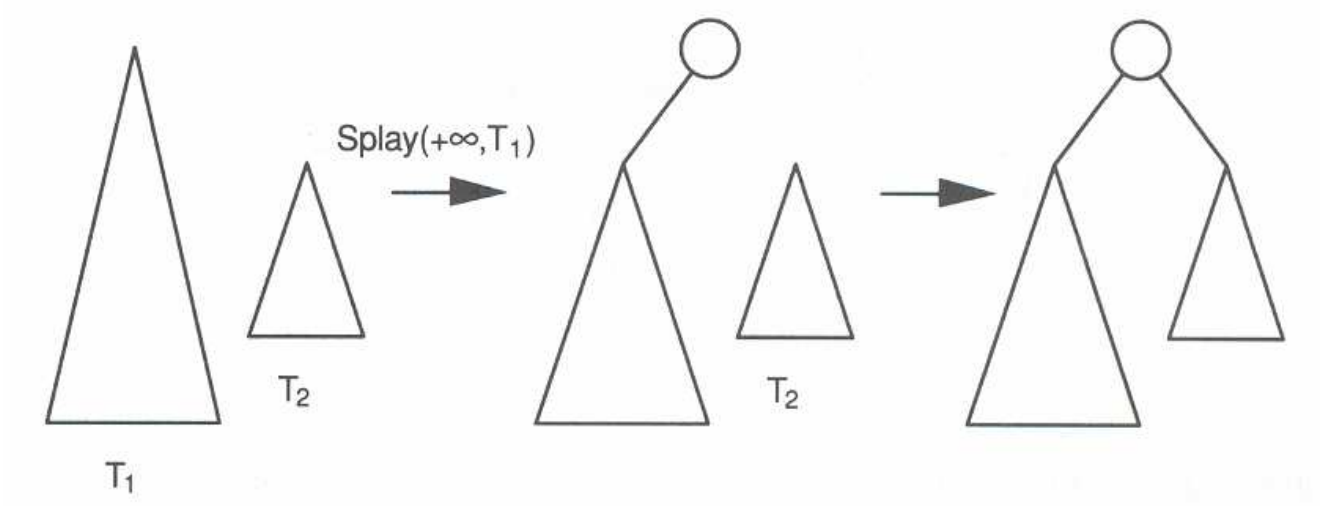


Insert(K, I, T): Εκτελούμε τη λειτουργία Splay(K,T). Αν το K είναι στη ρίζα, αλλαγή του Info του σε I. Διαφορετικά, δημιουργούμε νέο κόμβο που να περιέχει τα K, I, και τοποθετούμε τον κόμβο αυτό σαν ρίζα.

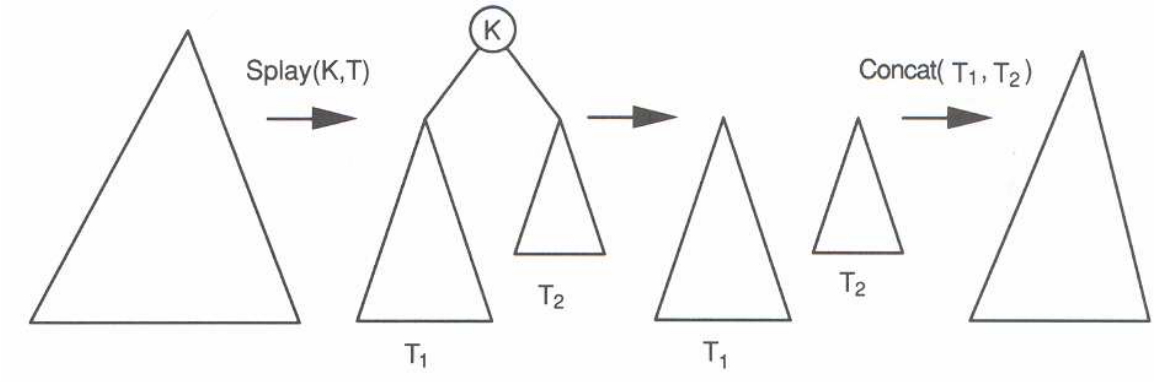


Delete(K,T)

Concat(T1,T2): Εκτελούμε πρώτα Splay(+∞, T1), όπου +∞ είναι ένα κλειδί μεγαλύτερο από κάθε κλειδί που μπορεί να περιέχεται στο δένδρο (μετά από αυτή τη λειτουργία, το T1 δεν έχει δεξιό υποδένδρο). Τοποθετούμε το T2 σαν δεξιό υποδένδρο της ρίζας του T1.



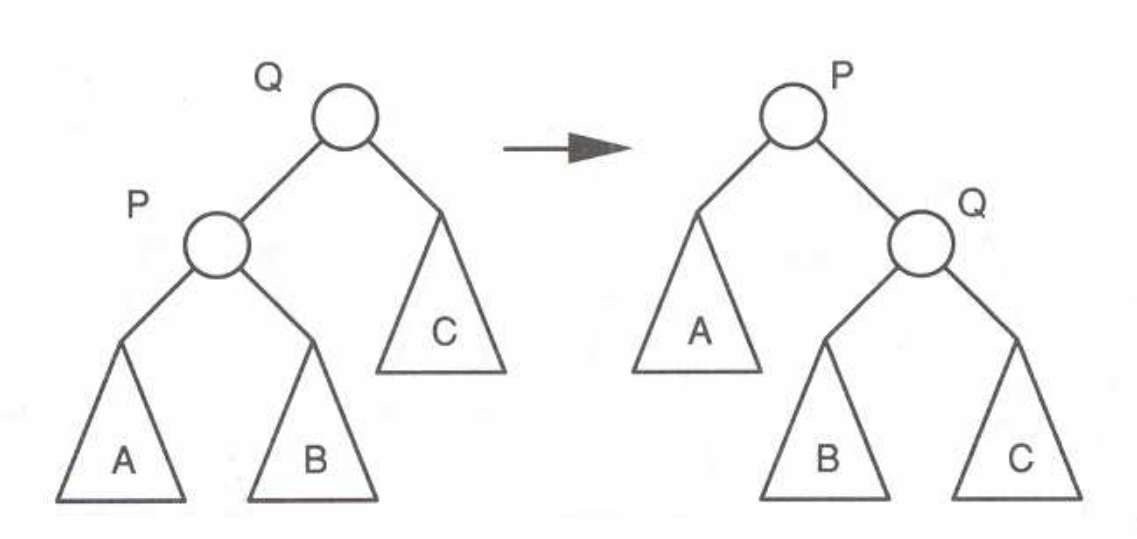
Delete(K,T): Εκτελούμε Splay(K,T). Αν η ρίζα δεν περιέχει το K τότε ο αλγόριθμος τερματίζει. Διαφορετικά, εκτελούμε την Concat στα δύο υποδένδρα της ρίζας.



Υλοποίηση της Λειτουργίας Splay

- Για να εκτελέσουμε splay γύρω από ένα κλειδί Κ, πρώτα αναζητούμε το Κ με το γνωστό τρόπο, και αποθηκεύουμε το μονοπάτι που ακολουθήσαμε σε στοίβα.
- Έστω Ρ ο τελευταίος κόμβος σε αυτό το μονοπάτι. Αν το Κ υπάρχει στο δένδρο, θα βρίσκεται στον Ρ.
- Διαφορετικά ο κόμβος με κλειδί Κ θα πρέπει να εισαχθεί σαν ένα από τα παιδιά του Ρ.
- Μετά την Splay, ο Ρ θα βρίσκεται στη ρίζα. Διακρίνουμε περιπτώσεις:

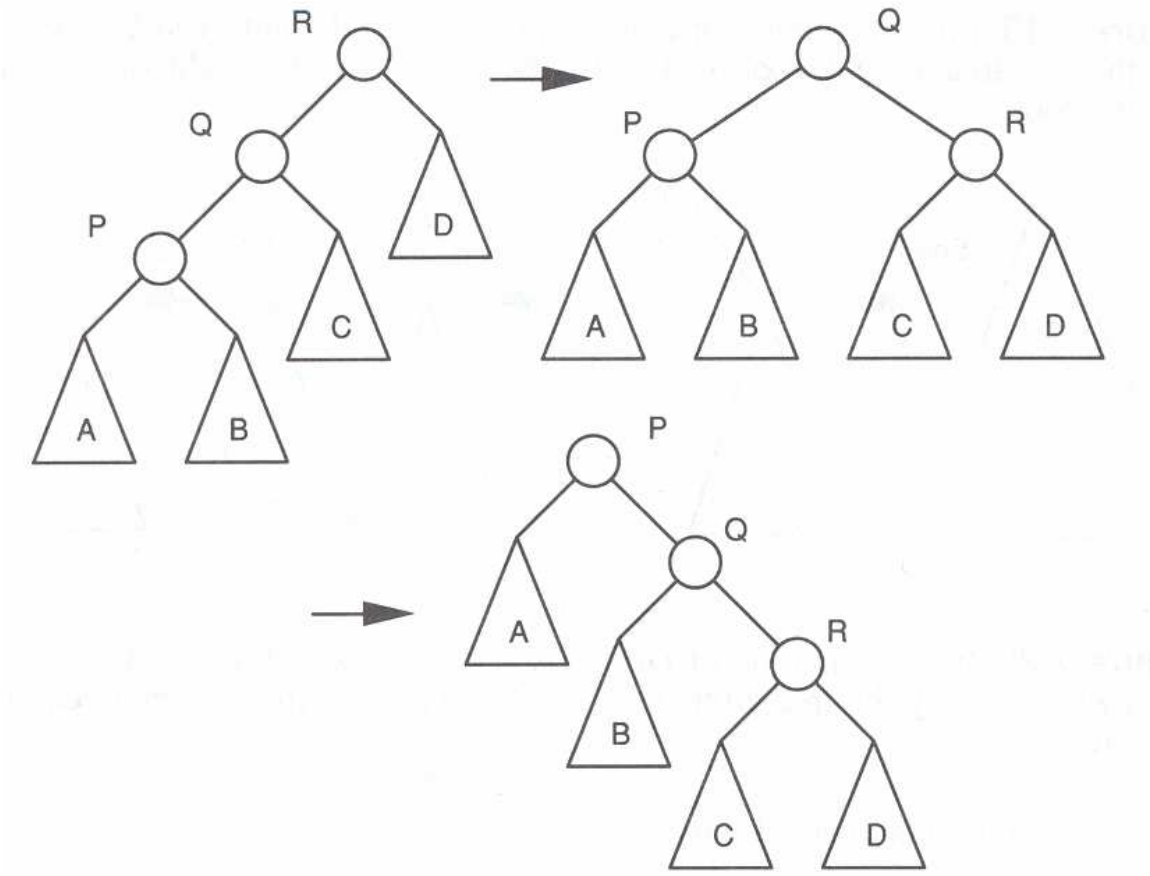
1. Ο Ρ δεν έχει παππού, δηλαδή το Parent(P) είναι η ρίζα.



Εκτελούμε μια απλή περιστροφή.

Υλοποίηση της Λειτουργίας Splay: Συνέχεια

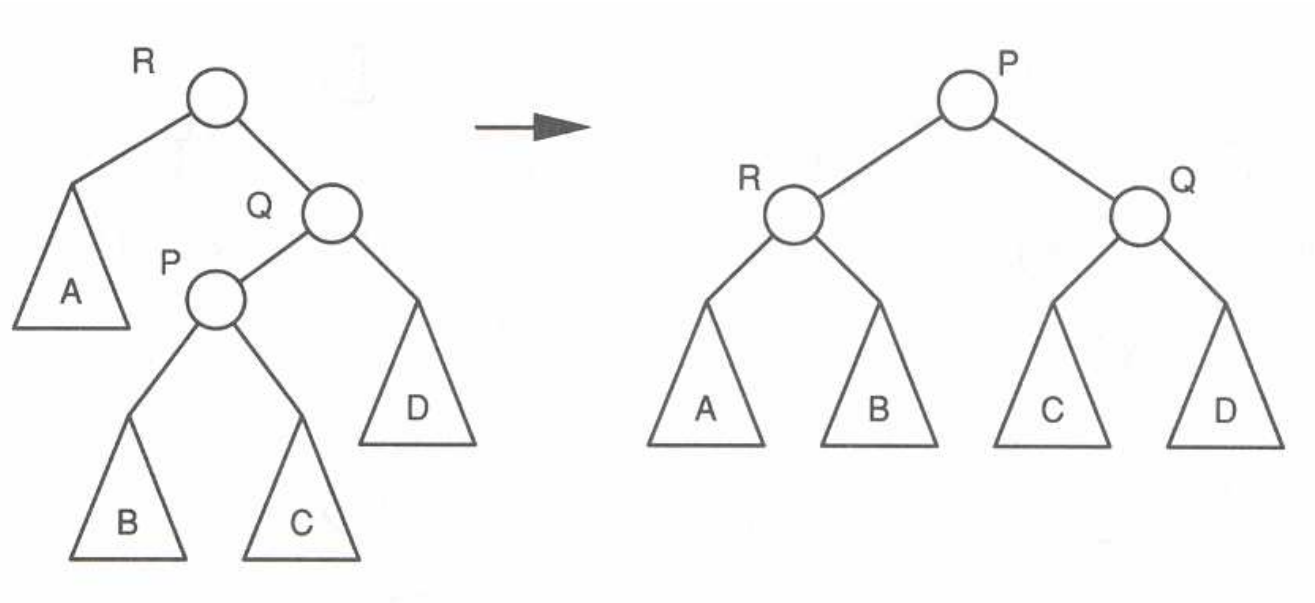
2. Ο P και ο Parent(P) είναι και οι δύο είτε αριστερά παιδιά, είτε δεξιά παιδιά.



Εκτελούμε δύο απλές περιστροφές προς την ίδια κατεύθυνση, την 1η γύρω από τον παππού του P και την 2η γύρω από τον πατέρα του.

Υλοποίηση της Λειτουργίας Splay: Συνέχεια

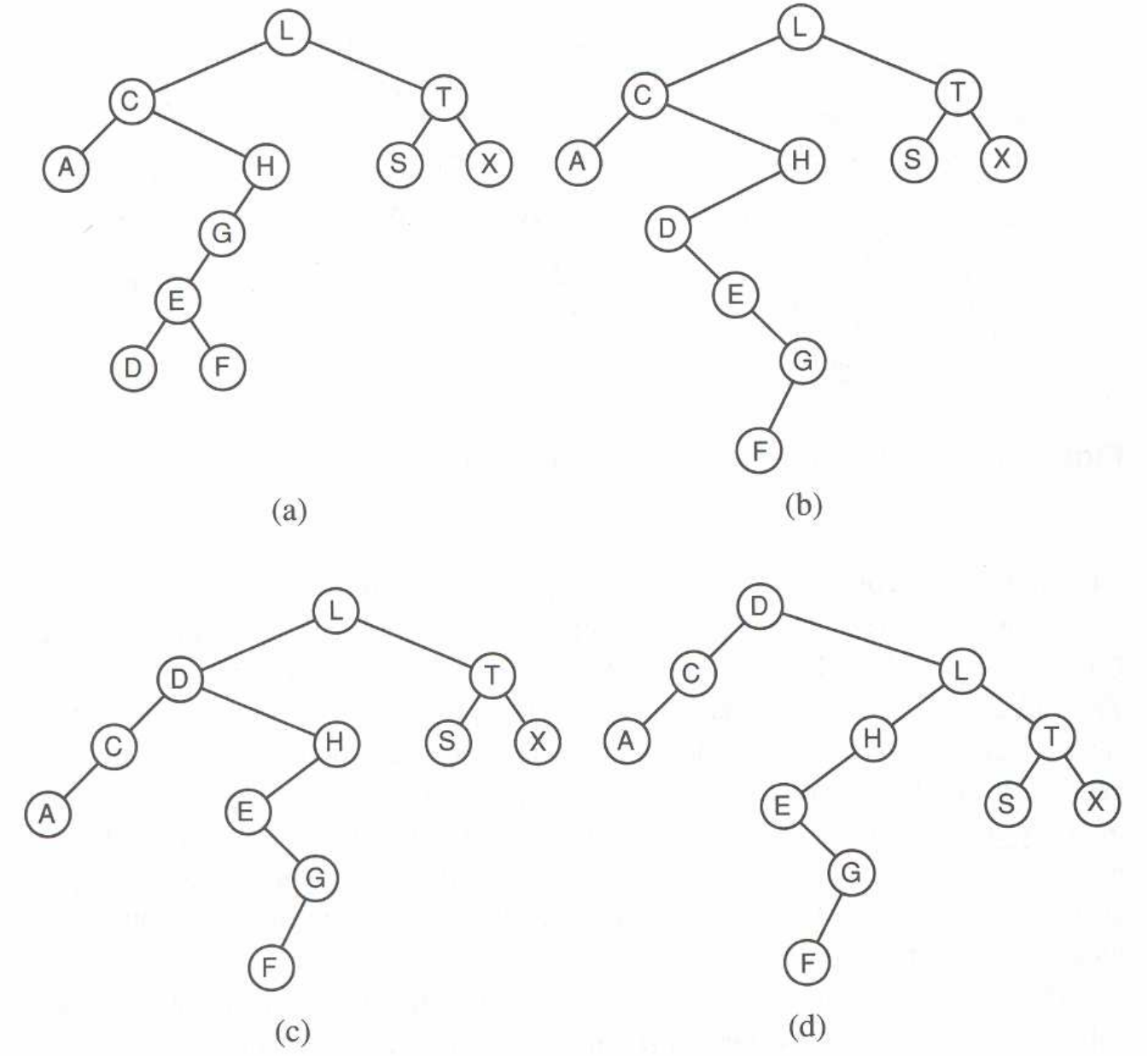
3. Ένας από τους P, Parent(P) είναι αριστερό παιδί και ο άλλος δεξιό παιδί.



Εκτελούμε δύο περιστροφές αλλά σε αντίθετες κατευθύνσεις, την $1^{η}$ γύρω από τον πατέρα του P και την $2^{η}$ γύρω από τον παππού του.

Παράδειγμα

Splaying γύρω από το D



ΕΝΟΤΗΤΑ 7
ΚΑΤΑΚΕΡΜΑΤΙΣΜΟΣ

Εισαγωγή

Σκοπός

Υλοποίηση συνόλων που λαμβάνουν υπ' όψιν τους τη δομή των κλειδιών. Οι υλοποιήσεις αυτές δεν εκτελούν μόνο συγκρίσεις πάνω στα κλειδιά, αλλά αντιμετωπίζουν τα κλειδιά σαν μια αριθμητική ποσότητα πάνω στην οποία μπορούν να εκτελεστούν αυθαίρετοι αριθμητικοί υπολογισμοί.

Πρόβλημα

Έχουμε ένα σύνολο από κλειδιά $\{K_0, \ldots, K_{n-1}\}$ και θέλουμε να υλοποιήσουμε Insert() και LookUp() (ίσως και Delete()).

Ιδέα

Αποθηκεύουμε τα στοιχεία αυτά σε ένα πίνακα T[0..m-1], που ονομάζεται **hash table (πίνακας κατακερματισμού)**, με τη βοήθεια μιας **hash function (συνάρτησης κατακερματισμού)** h: **K**→ {0, …, m-1}, όπου **K** είναι ο χώρος των κλειδιών.

Για κάθε j, το κλειδί K_j αποθηκεύεται στη θέση του πίνακα $h(K_j)$.

Αν η h μπορεί να υπολογιστεί γρήγορα, σε πόσο χρόνο μπορούμε να προσπελάσουμε το κλειδί?

Ποιο πρόβλημα μπορεί να δημιουργηθεί?

Συγκρούσεις (collisions)

Σύγκρουση συμβαίνει όταν για δύο κλειδιά K_j, K_i που είναι διαφορετικά μεταξύ τους ισχύει ότι $h(K_j) = h(K_i)$.

Όταν συμβαίνουν συγκρούσεις, θα πρέπει να γίνει ανακατανομή κλειδιών, ώστε να επιλυθεί η σύγκρουση και τα κλειδιά να μπορούν να βρεθούν (με LookUp()) σε λογικό χρόνο μετά την ανακατανομή.

Καλές Συναρτήσεις Κατακερματισμού

Κάνουν καλή διασκόρπιση των κλειδιών στον πίνακα:

Αν ένα κλειδί Κ επιλέγεται τυχαία από το χώρο κλειδιών, η πιθανότητα να ισχύει h(K) = j, θα πρέπει να είναι 1/m, ίδια για όλα τα j (δηλαδή για όλες τις θέσεις του πίνακα).

Παράδειγμα

h(k) = k mod m

Το m δεν πρέπει να είναι δύναμη του 2.

Η hash function modulo είναι καλή μόνο αν ο m είναι πρώτος αριθμός.

Μέθοδοι Διαχείρισης Συγκρούσεων

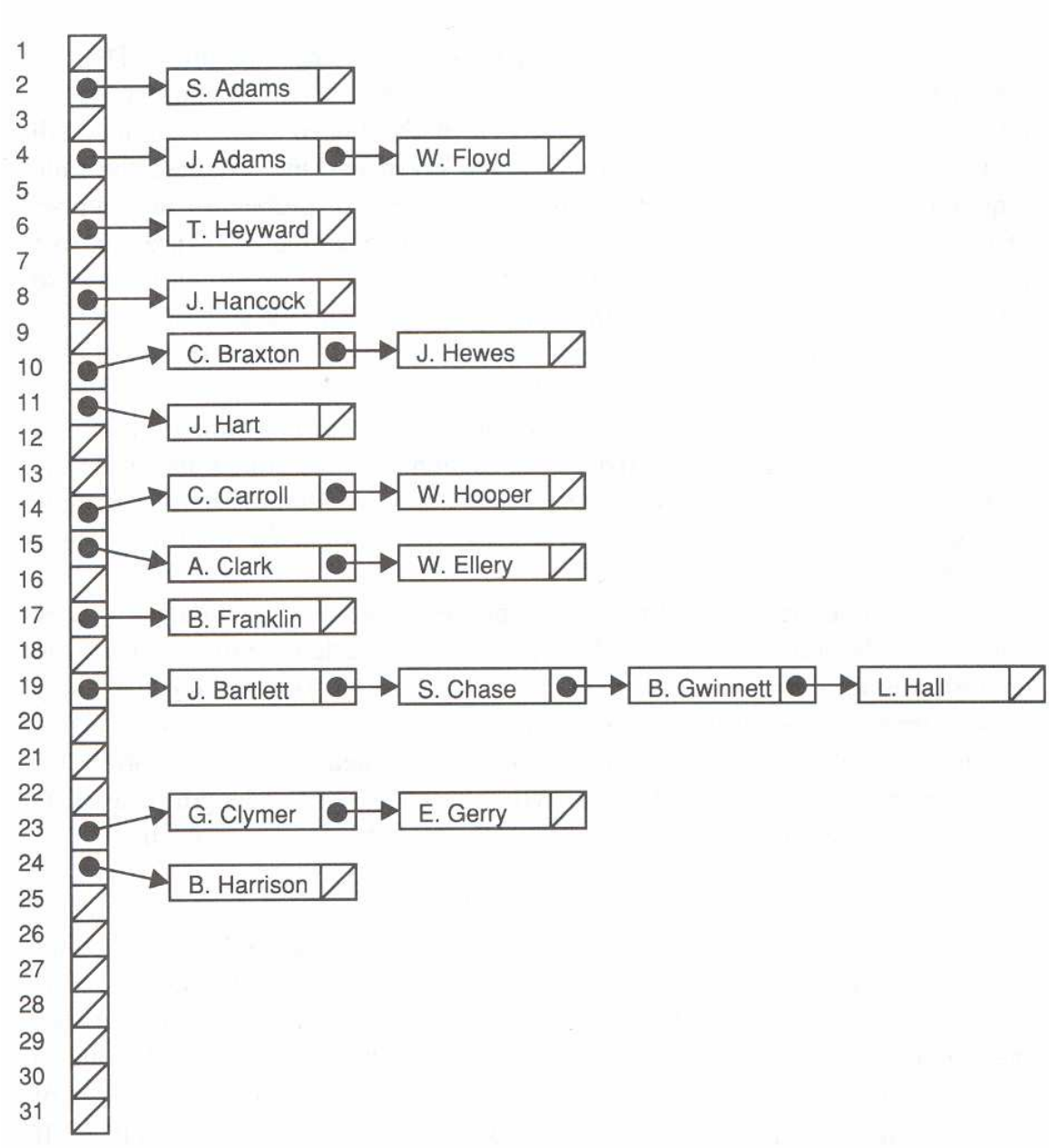
Μέθοδος αλυσίδας

T[j]: δεν περιέχει ένα στοιχείο αλλά ένα δείκτη σε μια δυναμική δομή η οποία περιέχει κάθε στοιχείο με κλειδί K τέτοιο ώστε h(K) = j.

Name	Date of Death	$h(K)$	$h_2(K)$
J. Adams	July 4, 1826	4	7
S. Adams	October 2, 1803	2	10
J. Bartlett	May 19, 1795	19	5
C. Braxton	October 10, 1797	10	10
C. Carroll	November 14, 1832	14	11
S. Chase	June 19, 1811	19	6
A. Clark	September 15, 1794	15	9
G. Clymer	January 23, 1813	23	1
W. Ellery	February 15, 1820	15	2
W. Floyd	August 4, 1821	4	8
B. Franklin	April 17, 1790	17	4
E. Gerry	November 23, 1814	23	11
B. Gwinnett	May 19, 1777	19	5
L. Hall	October 19, 1790	19	10
J. Hancock	October 8, 1793	8	10
B. Harrison	April 24, 1791	24	4
J. Hart	May 11, 1779	11	5
J. Hewes	November 10, 1779	10	11
T. Heyward	March 6, 1809	6	3
W. Hooper	October 14, 1790	14	10

Μέθοδοι Διαχείρισης Συγκρούσεων

Μέθοδος αλυσίδας



Probes

1 Probe: 1 πρόσβαση σε κάποιο μέρος της δομής.

Παράδειγμα

1 probe για την πρόσβαση στον πίνακα, προκειμένου να βρεθεί η θέση του πρώτου στοιχείου της αλυσίδας, 2ο probe απαιτείται για την ίδια την πρόσβαση στο 1ο στοιχείο της αλυσίδας, κλπ.

n: μέγεθος λεξικού

m: μέγεθος πίνακα

α = n/m, load factor: ο μέσος αριθμός κλειδιών σε κάθε αλυσίδα είναι α.

S(α): αναμενόμενος αριθμός probes για την εκτέλεση LookUp σε κλειδί που υπάρχει στην δομή.

U(α): αναμενόμενος αριθμός probes για την εκτέλεση LookUp σε κλειδί που δεν υπάρχει στην δομή.

U(α) = 1 + α.

Probes

S(α)

Ποιος είναι ο μέσος αριθμός probes για επιτυχημένη αναζήτηση, αν

Το μέγεθος της αλυσίδας είναι 1?

Το μέγεθος της αλυσίδας είναι 2?

Το μέγεθος της αλυσίδας είναι κ?

Αν όλες οι αλυσίδες ήταν μη-άδειες, το αναμενόμενο μήκος κάθε αλυσίδας θα ήταν α, οπότε $S(\alpha) = 1 + (1+\alpha)/2 = 3/2 + \alpha/2$.

Μια επιτυχημένη LookUp ποτέ δεν εξετάζει άδειες αλυσίδες. Γιατί?

Ωστόσο, το μήκος της αλυσίδας μπορεί να είναι λίγο μεγαλύτερο από α:

$$S(\alpha) \approx 2 + \alpha/2$$

Χειρότερη περίπτωση?

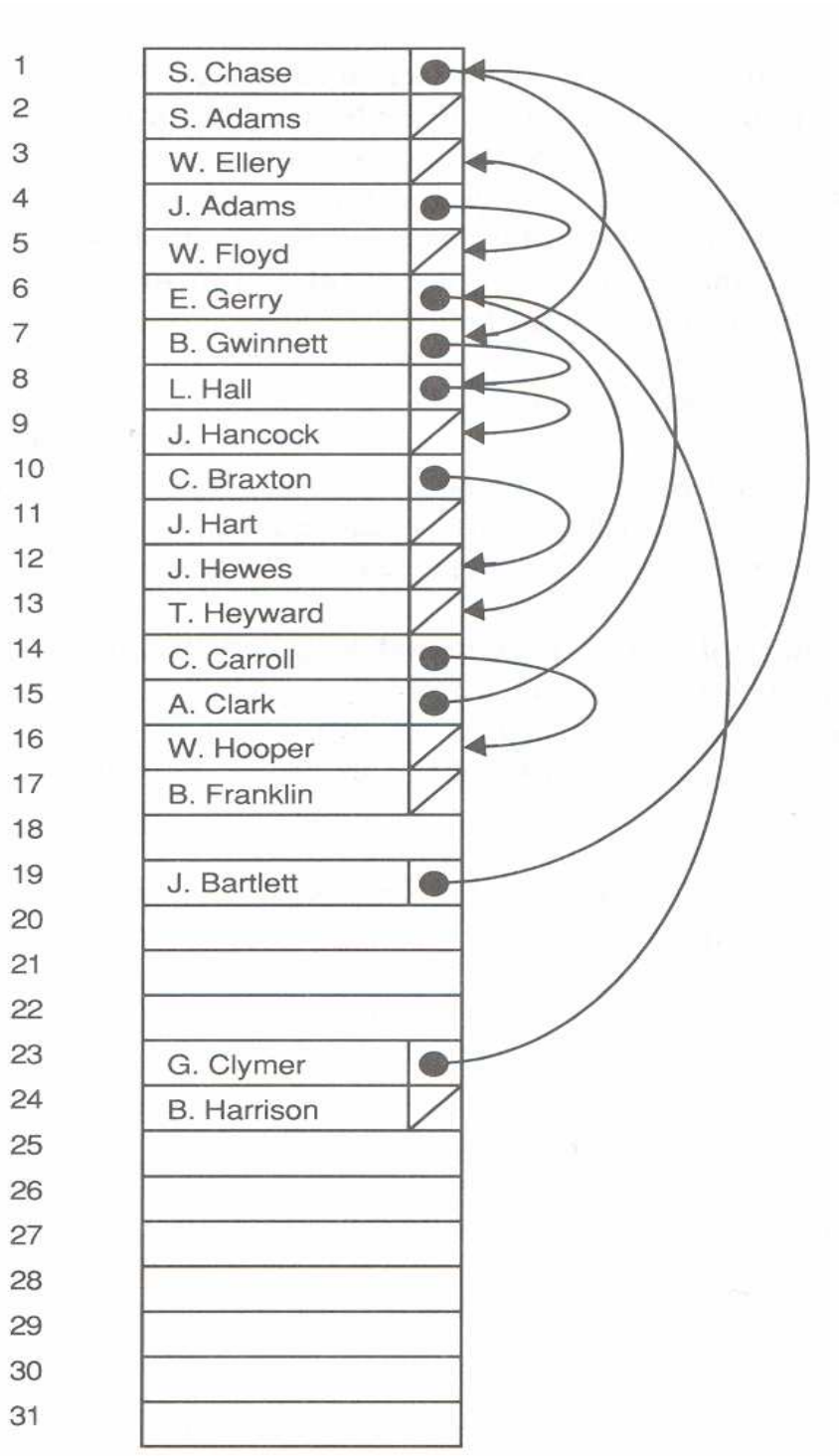
Ποια είναι η κατάλληλη επιλογή για το m?

Πόσο εύκολα μπορούμε να υλοποιήσουμε διαγραφή?

Μέθοδοι Διαχείρισης Συγκρούσεων

Μέθοδος Μικτών Αλυσίδων

Ιδέα: Ολόκληρη η αλυσίδα αποθηκεύεται μέσα στον πίνακα.



Μέθοδοι Διαχείρισης Συγκρούσεων

Πρόβλημα

Μια θέση στην οποία έχει τοποθετηθεί ένα κλειδί για το οποίο η συνάρτηση κατακερματισμού καθορίζει άλλη θέση εισαγωγής, μπορεί να πρέπει να αποθηκεύσει αργότερα το πρώτο κλειδί μιας νέας αλυσίδας.

Λύση

Το κλειδί τοποθετείται στην πρώτη διαθέσιμη θέση και συνδέεται στην αλυσίδα του κλειδιού που κατέχει την αρχική θέση.

LookUp: Ίδια όπως στη μέθοδο με αλυσίδες, αλλά η κάθε αλυσίδα ίσως περιέχει κλειδιά για τα οποία η συνάρτηση κατακερματισμού δίνει διαφορετικές τιμές.

Insert: Όπως στη μέθοδο με αλυσίδες, αλλά τα νέα στοιχεία τοποθετούνται στον ίδιο τον πίνακα και ο πίνακας μπορεί να γεμίσει.

Cellar – Κελάρι

Κρατάμε τις πρώτες θέσεις του πίνακα μόνο για την επίλυση συγκρούσεων.

Πειραματική και θεωρητική δουλειά έχει αποδείξει ότι αν το κελάρι είναι το 14% του συνολικού πίνακα, η απόδοση είναι καλή (για επιτυχής και μη αναζήτηση).

Μέθοδοι Διαχείρισης Συγκρούσεων

Μέθοδος Ανοικτής Διεύθυνσης

Τα κλειδιά αποθηκεύονται στον πίνακα κατακερματισμού, αλλά δεν χρησιμοποιούνται δείκτες (δεν σχηματίζονται αλυσίδες).

Για κάθε κλειδί ελέγχεται μια ακολουθία από θέσεις (ακολουθία αναζήτησης/εξέτασης, probe sequence). Η ακολουθία καθορίζεται βάσει κάποιου κανόνα και μπορεί να εξαρτάται και από το ίδιο το κλειδί.

H(K,p): Η p-οστή θέση που ελέγχεται για κάποιο κλειδί, p = 0, 1 …

LookUp: Ψάξε τις διαδοχικές θέσεις στην ακολουθία αναζήτησης, μέχρι είτε να βρεθεί το κλειδί, ή να βρεθεί μια κενή θέση στον πίνακα. Στη δεύτερη περίπτωση, το κλειδί δεν υπάρχει στον πίνακα.

Insert: Κάλεσε LookUp. Αν το κλειδί βρεθεί, ο αλγόριθμος τερματίζει, διαφορετικά, το νέο κλειδί εισάγεται στην κενή θέση (μπορεί να απαιτείται ανακατανομή των κλειδιών στον πίνακα, προκειμένου να μειωθεί ο χρόνος αναζήτησης).

Μέθοδοι Καθορισμού Ακολουθιών Αναζήτησης

Γραμμική Αναζήτηση

Αλγόριθμος:

H(K,0) = h(K);

H(K,p+1) = (H(K,p)+1) mod m.

Τι γίνεται αν ο πίνακας είναι γεμάτος? Πώς μπορούμε να συμπεράνουμε κάτι τέτοιο?

1		
2	S. Adams	1
3		
4	J. Adams	1
5	W. Floyd	2
6	T. Heyward	1
7		
8	J. Hancock	1
9		
10	C. Braxton	1
11	J. Hart	1
12	J. Hewes	3
13		
14	C. Carroll	1
15	A. Clark	1
16	W. Ellery	2
17	B. Franklin	1
18	W. Hooper	5
19	J. Bartlett	1
20	S. Chase	2
21	B. Gwinnett	3
22	L. Hall	4
23	G. Clymer	1
24	E. Gerry	2
25	B. Harrison	2
26		
27		
28		
29		
30		
31		

Μέθοδοι Καθορισμού Ακολουθιών Αναζήτησης

Γραμμική Αναζήτηση

Η απόδοση της μεθόδου είναι ικανοποιητική όταν ο πίνακας δεν είναι πολύ γεμάτος.

Σημαντικότερο Πρόβλημα

Φαινόμενο Συγκέντρωσης (primary clustering)

Όταν ένα μπλοκ (cluster) με συνεχόμενες κατειλημμένες θέσεις δημιουργηθεί, αποτελεί προορισμό για περαιτέρω συγκρούσεις, ενώ νέες τέτοιες συγκρούσεις οδηγούν στην αύξηση του μεγέθους του μπλοκ.

Αριθμός απαιτούμενων probes για μη επιτυχημένη αναζήτηση,

- *αν η συνάρτηση κατακερματισμού επιστρέφει μια κενή θέση του πίνακα?*
- *αν η συνάρτηση κατακερματισμού επιστρέφει μια κατειλημμένη θέση του πίνακα?*

Προσπάθεια βελτίωσης:

Πρόσθεσε μια σταθερά c > 1, στον δείκτη για να βρεις τον επόμενο δείκτη στην ακολουθία.

Τι αποτέλεσμα θα έχει αυτή η προσπάθεια? Βοηθάει?

Μέθοδοι Καθορισμού Ακολουθιών Αναζήτησης

Διπλός κατακερματισμός

Αλγόριθμος:

$H(K,0) = h(K);$

$H(K, p+1) = (H(K, p) + h_2(K)) \bmod m$

Η γραμμική αναζήτηση ισοδυναμεί με διπλό κατακερματισμό, αν $h_2(K) = 1$, $\forall$ κλειδί K.

Η ακολουθία αναζήτησης πρέπει (τελικά) να περιέχει όλες τις θέσεις του πίνακα.

$h_2(K) > 0$

$h_2(K)$ και m δεν πρέπει να έχουν κοινούς διαιρέτες

Γιατί?

Έστω d διαιρεί το $h_2(K)$ και το m:

$[(m/d)h_2(K)] \bmod m = [m(h_2(K)/d)] \bmod m = 0,$

άρα η (m/d)-οστή θέση στην ακολουθία θα είναι η ίδια όπως η $1^{η}$.

Διαλέγουμε τον m να είναι πρώτος.

Παράδειγμα Διπλού Κατακερματισμού

Βασική συνάρτηση κατακερματισμού: η ημέρα θανάτου

Δευτερεύουσα συνάρτηση κατακερματισμού: ο μήνας θανάτου (Ιανουάριος = 1, Φεβρουάριος = 2, κλπ.)

1	J. Hewes	3
2	S. Adams	1
3	E. Gerry	2
4	J. Adams	1
5		
6	T. Heyward	1
7		
8	J. Hancock	1
9		
10	C. Braxton	1
11	J. Hart	1
12	W. Floyd	2
13	W. Hooper	4
14	C. Carroll	1
15	A. Clark	1
16		
17	W. Ellery	2
18		
19	J. Bartlett	1
20		
21	B. Franklin	2
22		
23	G. Clymer	1
24	B. Gwinnett	2
25	S. Chase	2
26		
27		
28	B. Harrison	2
29	L. Hall	2
30		
31		

Τι συμβαίνει κατά την εισαγωγή του τελευταίου ονόματος, W. Hooper?

Απόδοση Μεθόδου Διπλού Κατακερματισμού

Υπόθεση

Κάθε θέση που εξετάζεται στον πίνακα κατακερματισμού είναι ανεξάρτητη από τις υπόλοιπες θέσεις του πίνακα, και η πιθανότητα να επιλεγεί μια κατειλημμένη θέση είναι ίση με το load factor.

Είναι αυτή η υπόθεση σωστή?

Όχι, αφού:

1. Διαδοχικές θέσεις μπορεί να εξαρτώνται με κάποιο τρόπο.
2. Είναι αδύνατο να εξεταστεί η ίδια θέση 2 φορές.

Εισαγωγή n κλειδιών σε πίνακα κατακερματισμού μεγέθους m, δεδομένου ότι η υπόθεση είναι σωστή

$\alpha_i = i/m$, $i \leq n$: η πιθανότητα σύγκρουσης σε κάθε βήμα είναι α_i μετά την εισαγωγή i κλειδιών.

Απόδοση Μεθόδου Διπλού Κατακερματισμού - Συνέχεια

Αναμενόμενος αριθμός probes σε μη επιτυχημένη αναζήτηση, αν n-1 κλειδιά έχουν ήδη εισαχθεί είναι:

$U_{n-1} = 1*(1 - a_{n-1}) + 2*a_{n-1}*(1-a_{n-1})+3a_{n-1}^2(1-a_{n-1}) + \ldots$
$= 1 + a_{n-1} + a_{n-1}^2 + \ldots$
$= 1/(1 - a_{n-1})$

Αριθμός probes σε επιτυχημένη αναζήτηση

Μέσος αριθμός από probes για την εισαγωγή κάθε ενός από τα n κλειδιά.

Ο αναμενόμενος αριθμός probes για την εισαγωγή του i-οστού κλειδιού = αναμενόμενο αριθμό probes σε μη επιτυχημένη αναζήτηση. Επομένως:

$$S_n = (1/n) \sum_{i=1}^{n} U_{i-1}$$
$$= (1/n) \sum_{i=1}^{n} 1/(1 - a_{i-1})$$
$$= (m/n) \sum_{i=1}^{n} 1/(m - i + 1)$$
$$= (m/n) (H_m - H_{m-n}),$$

όπου $H_i = 1 + ½ + \ldots + 1/i \approx \ln i$.

$S_n \approx (m/n) (\ln m - \ln (m-n))$
$= (m/n) \ln(m/m-n)$
$= (1/a_n) \ln (1/(1-a_n))$.

Ταξινομημένος Κατακερματισμός
Ordered Hashing

Αν τα κλειδιά ήταν αλφαβητικά ταξινομημένα θα είχαμε μείωση του χρόνου για αποτυχημένες αναζητήσεις:

Αν ένα μεγαλύτερο κλειδί από το ζητούμενο Κ προσπελαστεί τερματίζει η αναζήτηση.

Μέθοδος με αλυσίδες

Διατηρούμε τα κλειδιά στις αλυσίδες ταξινομημένα αλφαβητικά.

Μέθοδος Ανοικτής Διεύθυνσης

Τα κλειδιά πρέπει να εισαχθούν έτσι ώστε:

Τα κλειδιά που προηγούνται στην ακολουθία αναζήτησης από το Κ, θα πρέπει να είναι μικρότερα από το Κ.

Ιδέα

Αν στην ακολουθία αναζήτησης για το κλειδί Κ δούμε κλειδί Κ’ > Κ, τότε αντικαθιστούμε το Κ’ με το Κ και συνεχίζουμε με την εισαγωγή του Κ’ βάσει της ακολουθίας αναζήτησης του Κ’.

Ordered Hashing

```
procedure OrderedHashInsert(key K, info I, pointer P):
    if (P->size == m-1) then error;
    T = P->Table;
    pos = h(K);
    while (T[pos] != NULL) do
        if (T[pos]->Key > K) then
            swap(K, T[pos]->Key);
            swap(I, T[pos]->Info);
        else if (K == T[pos]->Key) then
            T[pos]->Info = I;
            return;
        pos = (pos + h2(K)) mod m;
    T[pos]->Key = K;
    T[pos]->Info = I;
    P->Size++;
```

```
function OrderedHashingLookUp(key K, pointer P):
                                                info
    T = P->Table;
    pos = h(K);
    while (T[pos] != NULL && T[pos]->Key < K) do
        pos = (pos + h2(K)) mod m;
    if (T[pos] != NULL && T[pos]->Key == K)
        return T[pos]->Info;
    else return NIL;
```

Ordered Hashing

Η τελική μορφή του πίνακα κατακερματισμού, μετά την εισαγωγή σε αυτόν (βάσει της τεχνικής ordered hashing) ενός συνόλου από κλειδιά, θα είναι η ίδια ανεξάρτητα από τη σειρά με την οποία τα κλειδιά αυτά εισάγονται στον πίνακα.

Υπόθεση

Η πιθανότητα να επιλεγεί ένα συγκεκριμένο κλειδί από το χώρο κλειδιών είναι η ίδια για όλα τα κλειδιά του χώρου.

Τότε:

- Ο αναμενόμενος χρόνος S_n για επιτυχημένη αναζήτηση δεν αλλάζει.

- Ο αναμενόμενος χρόνος Un για μη επιτυχημένη αναζήτηση μειώνεται. Γίνεται περίπου ίδιος με S_n.

Άρα:

$$S_n \approx U_n \approx (1/a_n) \ln(1/(1 - a_n)).$$

Ordered Hashing – Διαγραφές

Μέθοδος με αλυσίδες

Γίνεται εύκολα. Πως?

Μέθοδος Ανοικτής Διεύθυνσης

Ένα κλειδί δεν μπορεί να διαγραφεί αφήνοντας απλά τη θέση που κατείχε άδεια. *Γιατί?*

Παράδειγμα

Έστω ότι χρησιμοποιούμε τη μέθοδο γραμμικής αναζήτησης.

Έστω ότι 2 κλειδιά με την ίδια βασική/πρωταρχική τιμή κατακερματισμού εισάγονται στις θέσεις j και j+1 ενός πίνακα κατακερματισμού και στη συνέχεια αυτό στη θέση j διαγράφεται.

Το άλλο θα μείνει στη θέση j+1, αλλά η LookUp() θα σταματήσει όταν βρει τη θέση j κενή. Λάθος!

Ιδέα

Για κάθε θέση του πίνακα υπάρχει ένα bit, που ονομάζεται Deleted και μπορεί να είναι είτε 0 ή 1. Αρχικά όλα αυτά τα bits είναι 0. Αν διαγράψουμε κάτι από μια θέση του πίνακα, θέτουμε το bit της θέσης αυτής σε 1. Η LookUp() δεν τερματίζει σε άδειες θέσεις για τις οποίες το Deleted bit είναι 1.

Επεκτάσιμος Κατακερματισμός
Extendible Hashing

- ✓ Είναι μέθοδος που επιτρέπει την επαύξηση ή τη συρρίκνωση ενός πίνακα κατακερματισμού, διατηρώντας παράλληλα τους χρόνους πρόσβασης στη δομή χαμηλούς.

- ✓ Χρήσιμο για την αποθήκευση δεδομένων στη δευτερεύουσα μνήμη.

- ✓ Μπορεί να χρησιμοποιηθεί εναλλακτικά αντί ενός Β-δένδρου.

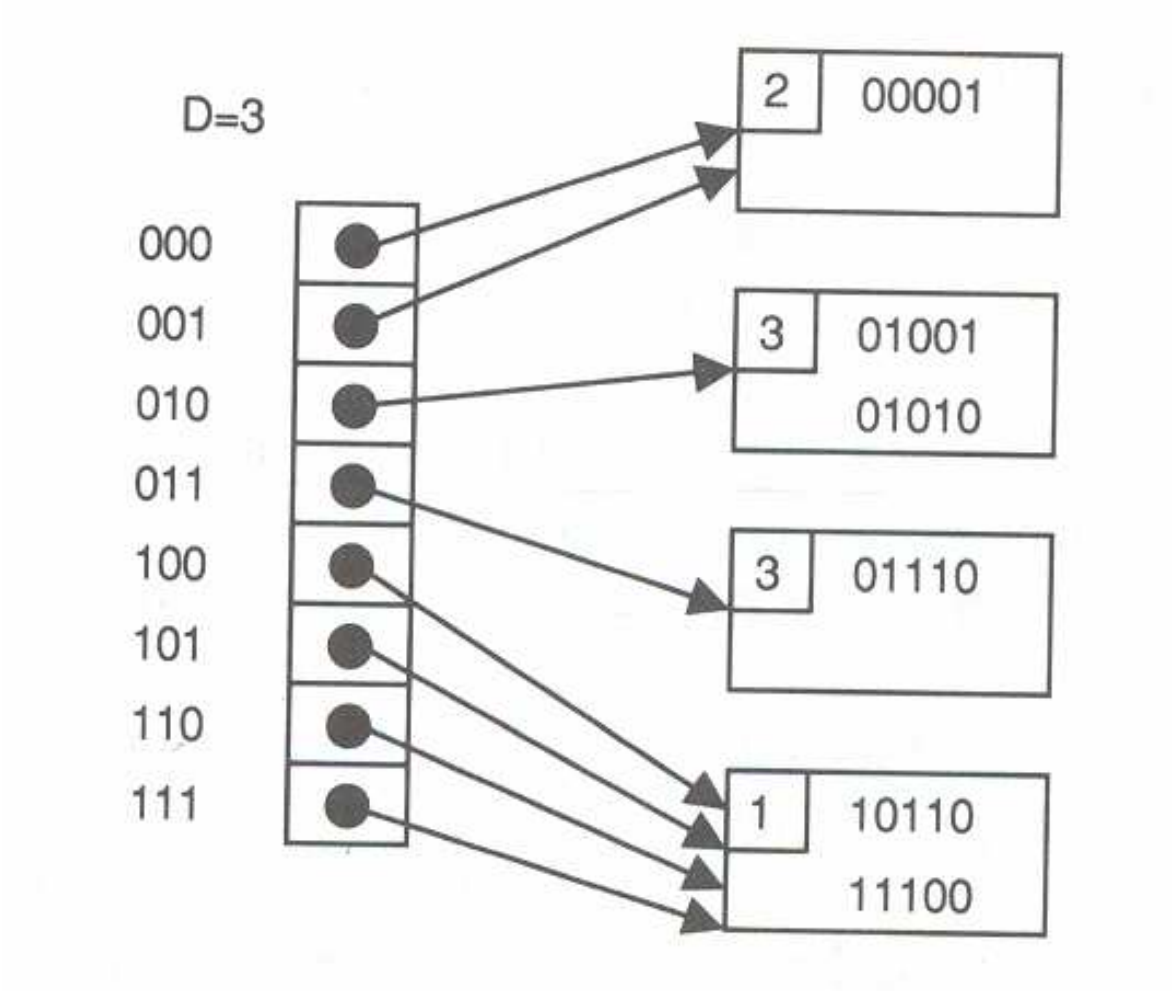
Δομή Δεδομένων 2 επιπέδων:

Ένας κατάλογος (directory) που αποτελεί τη δομή υψηλού επιπέδου και ένα σύνολο από σελίδες φύλλα (leaf pages) στις οποίες αποθηκεύονται δεδομένα.

Ο κατάλογος είναι ένας πίνακας από δείκτες στις σελίδες.

Οι σελίδες φύλλα είναι σταθερού μεγέθους, π.χ., b bytes η κάθε μια.

Επεκτάσιμος Κατακερματισμός



Υπάρχει μια συνάρτηση κατακερματισμού που απεικονίζει κλειδιά σε bit strings μήκους L.

L bits: 2^L τιμές κατακερματισμού

$h_d(K)$: τα πρώτα d bits του h(K), d <= L

Επεκτάσιμος Κατακερματισμός

Μια σελίδα περιέχει όλα εκείνα τα κλειδιά των οποίων η τιμή κατακερματισμού έχει ένα συγκεκριμένο πρόθεμα από bits.

Το μήκος αυτού του προθέματος ονομάζεται *βάθος* (depth) της σελίδας.

Το μέγιστο βάθος κάθε σελίδας ονομάζεται *βάθος* του πίνακα κατακερματισμού D.

Το directory είναι ένας πίνακας με 2^D δείκτες σε σελίδες.

Εύρεση σελίδας που περιέχει το κλειδί K

Υπολογίζουμε το $h_D(K)$;

Ακολουθούμε το δείκτη που περιέχεται στο $T[h_D(K)]$;

Αν για μια σελίδα ισχύει ότι d < D, τότε 2^{D-d} δείκτες σε συνεχόμενες θέσεις του πίνακα θα δείχνουν στη σελίδα αυτή.

Επεκτάσιμος Κατακερματισμός

Εισαγωγές

Μια σελίδα χωράει μόνο b δεδομένα.

Αν συμβεί υπερχείλιση μιας σελίδας με βάθος d, η σελίδα θα πρέπει να χωριστεί σε δύο σελίδες. Ο χωρισμός γίνεται με αύξηση του βάθους της σελίδας σε d+1 και με τη δημιουργία μιας νέας σελίδας, που ονομάζεται φιλική σελίδα (buddy page).

Τι αλλαγές προκαλεί η δημιουργία της νέας σελίδας στο directory?

Περιπτώσεις

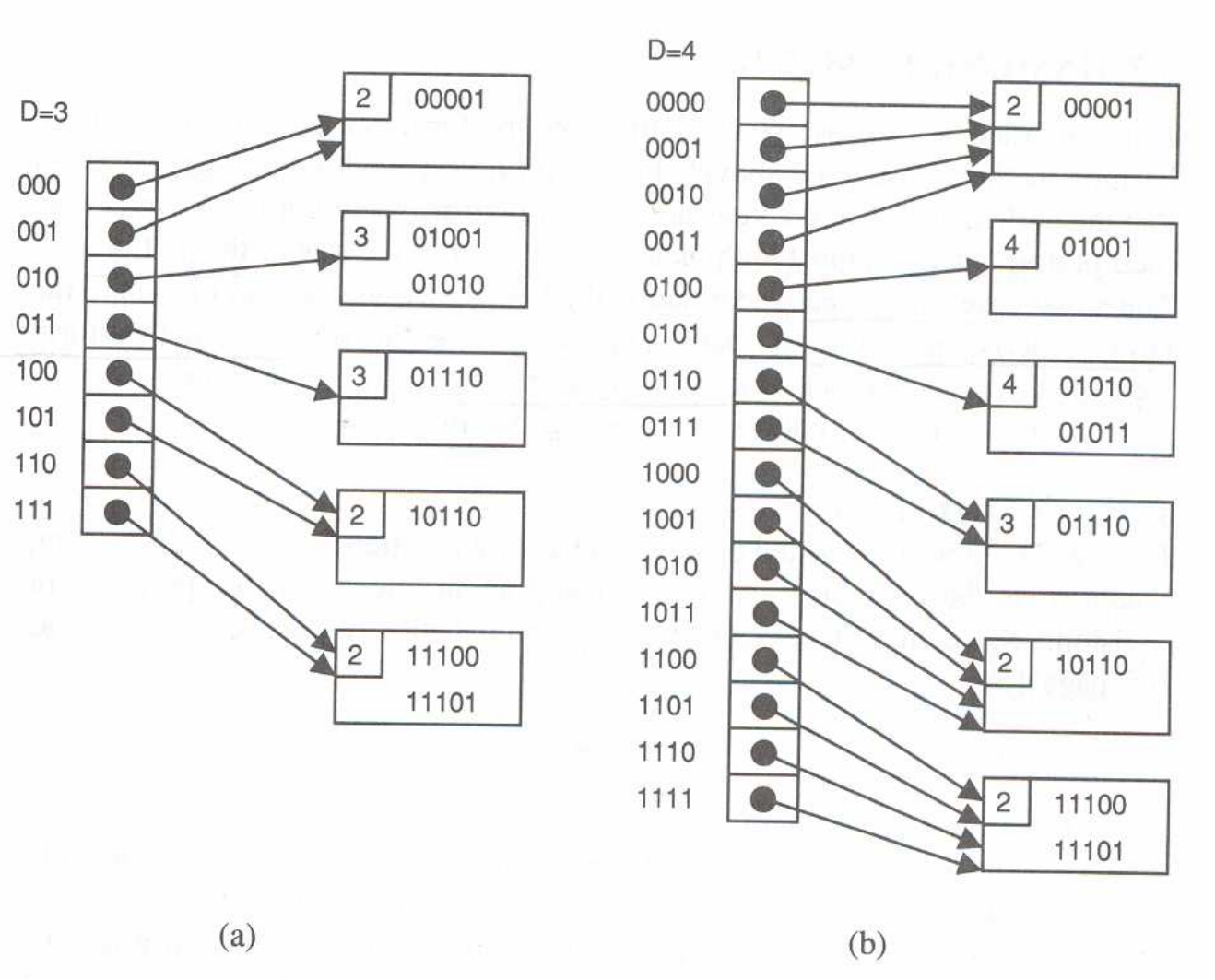
1. d < D

   Αλλαγή απλά μερικών δεικτών στο directory, ώστε αυτοί να δείχνουν στη νέα σελίδα.

2. d = D

   Διπλασιασμός του μεγέθους του directory και κατάλληλη αρχικοποίηση των δεικτών.

Επεκτάσιμος Κατακερματισμός
Εισαγωγή



Υποθέτουμε ότι b = 2.

(α) Εισαγωγή κλειδιού με τιμή κατακερματισμού 11101.

(β) Εισαγωγή κλειδιού με τιμή κατακερματισμού 01011.

Επεκτάσιμος Κατακερματισμός
Διαγραφή – Αποδοτικότητα

Διαγραφή

Αν η διαγραφή κάποιων κλειδιών από μια σελίδα 2i, για κάποιο i, έχει σαν αποτέλεσμα αυτή μαζί με την (2i+1) να έχει συνολικά b κλειδιά, θα μπορούσε να γίνει συνένωση των 2 σελίδων σε 1.

Γενικά αυτή η λειτουργία είναι ακριβή και δεν είναι συνετό να γίνεται συχνά.

Αποδοτικότητα

Η μέθοδος:

☺ είναι αποδοτική σε χρόνο προσπέλασης, αφού απαιτείται μόνο μια πρόσβαση στην κύρια μνήμη και μια στη δευτερεύουσα για να επιτευχθεί η προσπέλαση του ζητούμενου κλειδιού (και δεδομένου).

☹ δεν είναι αρκετά αποδοτική σε μνήμη, αφού πολλές από τις σελίδες μπορεί να είναι σχεδόν άδειες και αφού το directory συνήθως περιέχει παραπάνω από ένα δείκτες που δείχνουν στην ίδια σελίδα.

Συναρτήσεις Κατακερματισμού

Οι μέθοδοι κατακερματισμού που συζητήθηκαν έχουν πολύ καλή «μέση» απόδοση μόνο αν οι τιμές κατακερματισμού των κλειδιών είναι ομοιόμορφα κατανεμημένες (στα όρια του πίνακα κατακερματισμού).

Στη χειρότερη περίπτωση τα πράγματα μπορούν να είναι πολύ άσχημα (γραμμική πολυπλοκότητα).

Ποια είναι η χειρότερη περίπτωση?

- η συνάρτηση κατακερματισμού δεν επιτυγχάνει καλή κατανομή των κλειδιών γενικά, ή
- η συνάρτηση κατακερματισμού δεν επιτυγχάνει καλή κατανομή για συγκεκριμένα σύνολα κλειδιών.

Συναρτήσεις Κατακερματισμού

Κατακερματισμός με Διαίρεση

h(K) = K mod m

m: μέγεθος πίνακα κατακερματισμού

K: κλειδί (θεωρείται ακέραιος)

Αν K αλφαριθμητικό, υπολογίζουμε το $\sum_{i=0}^{p-1} c_i r^i$, όπου

- p: μήκος του αλφαριθμητικού
- r: ο αριθμός χαρακτήρων στον κώδικα (συνήθως 128 ή 256)
- c_j: ο κωδικός (ASCII) κάθε χαρακτήρα

Σκοπός Συνάρτησης Κατακερματισμού

Αποφυγή συστηματικών συγκρούσεων σε περιπτώσεις που τα κλειδιά επιλέγονται με ένα συστηματικά μη τυχαίο τρόπο.

Συναρτήσεις Κατακερματισμού: Κατακερματισμός με Διαίρεση

Παράδειγμα

- Αν το m είναι r ή r^2, το αποτέλεσμα της διαίρεσης είναι ο κωδικός του ενός ή των δύο τελευταίων χαρακτήρων.

Όμως:

Από τα γράμματα του αλφάβητου είναι πολύ λίγα αυτά που συνήθως συναντούνται ως τελευταίοι χαρακτήρες λέξεων.

- Αν το m είναι άρτιος, η τιμή h(K) θα είναι άρτια ή περιττή ανάλογα με το αν ο τελευταίος χαρακτήρας στο αλφαριθμητικό έχει περιττό ή άρτιο κωδικό.

Συναρτήσεις Κατακερματισμού: Κατακερματισμός με Διαίρεση

Λύση

Επιλογή του m να είναι πρώτος. Επίσης, είναι καλύτερο το m να μην διαιρεί τους r^k+a, r^k-1 για μικρές σταθερές k και a.

Παράδειγμα

Έστω m = r-1 και έστω ότι r-1 είναι πρώτος:

$$\sum_{i=0}^{p-1} c_i r^i \bmod (r-1) = \sum_{i=0}^{p-1} (c_i r^I \bmod (r-1)) \bmod (r-1)$$
$$= (\sum_{i=0}^{p-1} c_i) \bmod (r-1)$$

Μπορεί να αποδειχθεί επαγωγικά πως, για κάθε i,
$r^i \bmod (r-1) = (1 + (r-1) \sum_{j=0}^{i-1} r^j) \bmod (r-1) = 1$.

Άρα, αν m = r-1, όλες οι μεταθέσεις του ίδιου συνόλου χαρακτήρων (π.χ., ABC, BCA, CBA, κλπ.) έχουν την ίδια τιμή κατακερματισμού.

Συναρτήσεις Κατακερματισμού

Τέλειος Κατακερματισμός Στατικών Δεδομένων

Αν γνωρίζαμε εξ αρχής τα κλειδιά που θέλουμε να αποθηκευτούν, ίσως να μπορούσαμε να σχεδιάσουμε μια συνάρτηση κατακερματισμού που να αποφεύγει εντελώς τις συγκρούσεις.

Έχουν αναπτυχθεί διάφορες τεχνικές για την εύρεση τέλειων συναρτήσεων κατακερματισμού για δεδομένα σύνολα κλειδιών.

Η μέθοδος έχει περιορισμένη εφαρμογή, αφού συνήθως το σύνολο των κλειδιών δεν είναι γνωστό και τα δεδομένα αλλάζουν δυναμικά.

Καθολικές Κλάσεις Συναρτήσεων Κατακερματισμού

Ιδέα

Η συνάρτηση κατακερματισμού επιλέγεται τυχαία από ένα σύνολο συναρτήσεων κατακερματισμού (at run time).

Θετικά

- Η πιθανότητα επιλογής μιας «κακής» συνάρτησης κατακερματισμού είναι μικρή.

- Σε επόμενη εκτέλεση του προγράμματος, η πιθανότητα τα πράγματα να πάνε και πάλι άσχημα είναι μικρή.

- Ακόμη και αν τα κλειδιά που παρέχονται στο σύστημα είναι πολύ προσεκτικά επιλεγμένα από έναν αντίπαλο (ώστε να αποτελούν άσχημη είσοδο), η μέθοδος αυτή δουλεύει αποδοτικά.

Καθολικές Κλάσεις Συναρτήσεων Κατακερματισμού

K: χώρος κλειδιών

m: μέγεθος πίνακα κατακερματισμού

H: σύνολο συναρτήσεων από το **K** στο {0,...,m-1}

Ορισμός

Η κλάση συναρτήσεων H λέγεται καθολική αν για κάθε ζεύγος κλειδιών x,y, όπου $x \neq y$, ισχύει ότι:

$$|\{h \in H: h(x) = h(y)\}| / |H| \leq 1/m.$$

Για κάθε ζεύγος διαφορετικών κλειδιών, μόνο ένα ποσοστό 1/m (το πολύ) των συναρτήσεων της κλάσης μπορεί να οδηγεί σε σύγκρουση κατά την αποθήκευση του ζεύγους.

Διαλέγοντας μια συνάρτηση από την κλάση τυχαία, η πιθανότητα ένα ζεύγος κλειδιών να οδηγήσει σε σύγκρουση είναι 1/m.

Καθολικές Κλάσεις Συναρτήσεων Κατακερματισμού

Θεώρημα

Έστω ότι $|\mathbf{K}| = N$ είναι πρώτος αριθμός, και έστω ότι το **K** περιέχει (ως κλειδιά) τους ακεραίους 0,...,N-1. Για κάθε αριθμό $a \in \{1,\ldots,N-1\}$ και $b \in \{0,\ldots,N-1\}$ έστω:

$$h_{a,b}(x) = ((ax+b) \bmod N) \bmod m$$

Τότε, η

$$H = \{h_{a,b}: 1 \le a < N \text{ και } 0 \le b < N\}$$

είναι καθολική κλάση συναρτήσεων.

Παρατηρήσεις

- ✓ Το μέγεθος m του πίνακα μπορεί να είναι οποιοσδήποτε ακέραιος και όχι απαραίτητα πρώτος, ούτε καν περιττός.

- ✓ Το m μπορεί να είναι ακόμη και δύναμη του 2.

- ✓ Η μέθοδος μπορεί να χρησιμοποιηθεί σε συνδυασμό με τη μέθοδο του επεκτάσιμου κατακερματισμού.

ΕΝΟΤΗΤΑ 8

ΣΥΝΟΛΑ ΜΕ ΕΙΔΙΚΕΣ ΛΕΙΤΟΥΡΓΙΕΣ

(SETS WITH SPECIAL OPERATIONS)

ΟΥΡΕΣ ΠΡΟΤΕΡΑΙΟΤΗΤΑΣ

ΔΟΜΕΣ UNION-FIND

Ουρές Προτεραιότητας

- Έχει οριστεί μια διάταξη στα προς αποθήκευση στοιχεία και η προσπέλαση τους γίνεται βάσει αυτής της διάταξης.

- ✓ <K,I>: προς αποθήκευση στοιχεία
- ✓ K: είναι το κλειδί, τύπου key
- ✓ I: πληροφορία που συσχετίζεται με αυτό το κλειδί, τύπου info.

Ορισμός

Μια **ουρά προτεραιότητας** είναι ένας αφηρημένος τύπος δεδομένων για ένα σύνολο με στοιχεία ζεύγη <K,I>, που υποστηρίζει τις ακόλουθες λειτουργίες:

- *MakeEmptySet():* επιστρέφει το κενό σύνολο ∅.
- *IsEmptySet(S):* Επιστρέφει true αν S = ∅, false διαφορετικά
- *Insert(K,I,S):* Εισάγει το ζεύγος <K,I> στο S.
- *FindMin(S):* επιστρέφει το info πεδίο I του ζεύγους <K,I>, όπου K είναι το μικρότερο κλειδί στο σύνολο.
- *DeleteMin(S):* Διαγράφει το ζεύγος <K,I>, όπου K είναι το μικρότερο κλειδί στο σύνολο, και επιστρέφει I.

Ουρές Προτεραιότητας

Ποια είναι η διαφορά μιας ουράς από μια ουρά προτεραιότητας?

Ταξινόμηση

Δεδομένου ότι έχουμε υλοποιήσει μια ουρά προτεραιότητας, περιγράψτε αλγόριθμο που να ταξινομεί n στοιχεία.

Ποια είναι η χρονική πολυπλοκότητα του αλγορίθμου αυτού?

Αλγόριθμος

```
MakeEmptySet();
for (j = 0; j < n; j++)
    Εισαγωγή στην ουρά του j-οστού στοιχείου;
for (j = 0; j < n; j++)
    Print(DeleteMin(S));
```

Η πολυπλοκότητα εξαρτάται από την πολυπλοκότητα των λειτουργιών Insert() και DeleteMin() της ουράς προτεραιότητας.

Υλοποιήσεις Ουρών Προτεραιοτήτων

Υλοποίηση με Ισοζυγισμένα Δένδρα

Μπορούμε να υλοποιήσουμε μια ουρά προτεραιότητας με ισοζυγισμένα δένδρα?

AVL δένδρα, 2-3 δένδρα, Red-black δένδρα, B-δένδρα: όλα παρέχουν υλοποιήσεις ουρών προτεραιοτήτων.

Δεδομένου ενός ισοζυγισμένου δένδρου, πως θα μπορούσαμε να υλοποιήσουμε μια ουρά προτεραιότητας?

Ποια είναι η χρονική πολυπλοκότητα για τις λειτουργίες Insert(), FindMin() και DeleteMin()?

Υπάρχουν άλλες λειτουργίες που να υποστηρίζονται αποδοτικά?

(1) LookUp? (2) Delete? (3) FindMax – DeleteMax?

Μια ουρά προτεραιότητας που υποστηρίζει και τις λειτουργίες FindMax() και DeleteMax() ονομάζεται διπλή ουρά προτεραιότητας (ή ουρά προτεραιότητας με 2 άκρα).

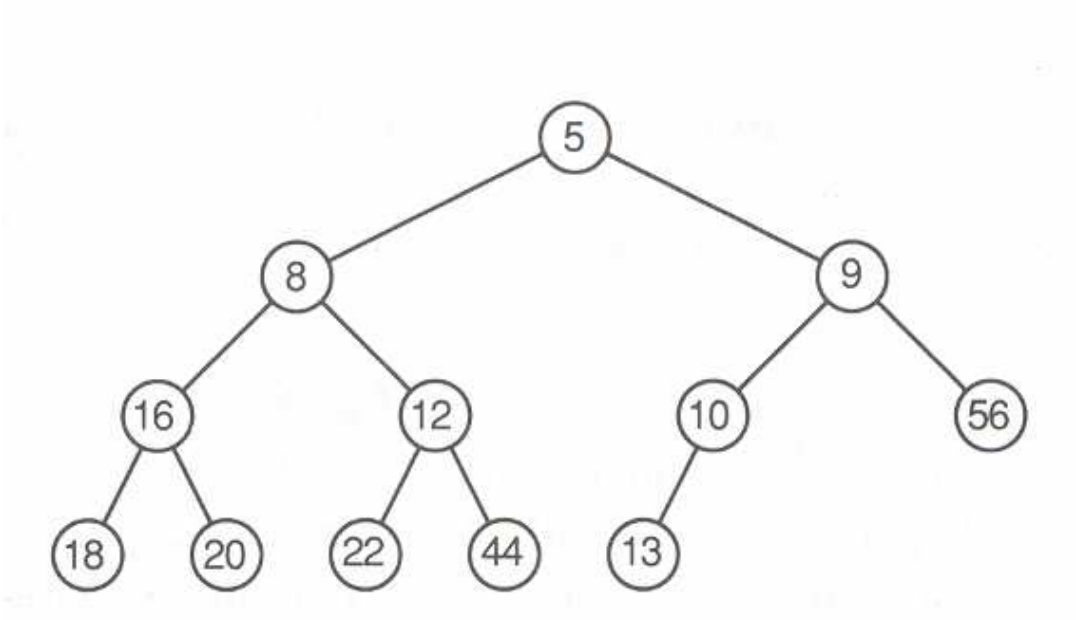
Υλοποιήσεις Ουρών Προτεραιοτήτων

Σωροί (Heaps)

Ένα μερικώς διατεταγμένο δένδρο είναι ένα δυαδικό δένδρο του οποίου τα στοιχεία έχουν την εξής ιδιότητα: Η προτεραιότητα κάθε κόμβου (δηλαδή το κλειδί του) είναι μικρότερη ή ίση εκείνης των παιδιών του κόμβου.

Ποιος είναι ο κόμβος με μικρότερη προτεραιότητα σε ένα μερικώς διατεταγμένο δένδρο?

Σε κάθε μονοπάτι από τη ρίζα προς οποιοδήποτε κόμβο, οι κόμβοι που διατρέχουμε είναι αύξουσας προτεραιότητας.



Πως υλοποιούμε την FindMin() σε μερικώς διατεταγμένο δένδρο?

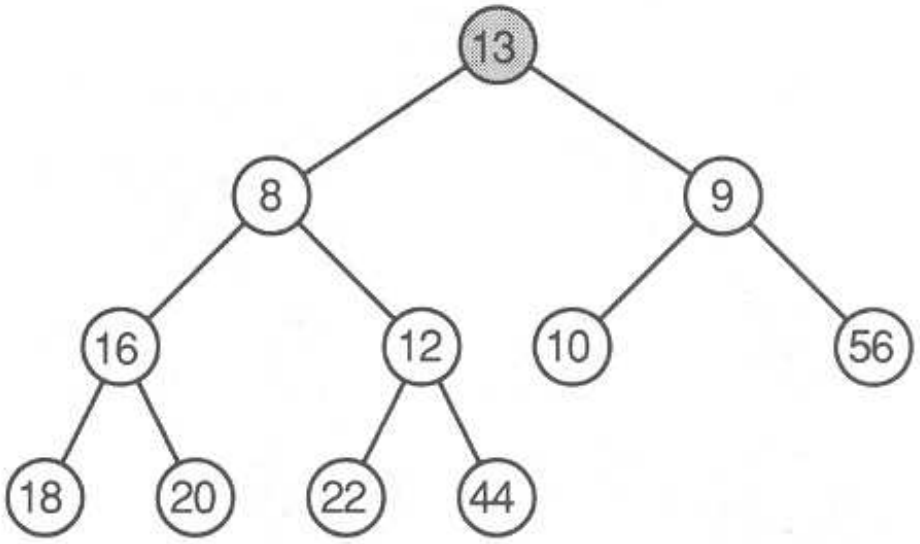
Υλοποίηση Insert() & DeleteMin()

Προκειμένου αυτές οι λειτουργίες να υλοποιηθούν αποδοτικά, θα πρέπει να είμαστε σίγουροι ότι κάθε στιγμή το ύψος του δένδρου είναι O(log n).

DeleteMin()

Δεν διαγράφουμε τη ρίζα, αλλά ένα φύλλο, αφού πρώτα αντιγράψουμε τα δεδομένα του στη ρίζα.

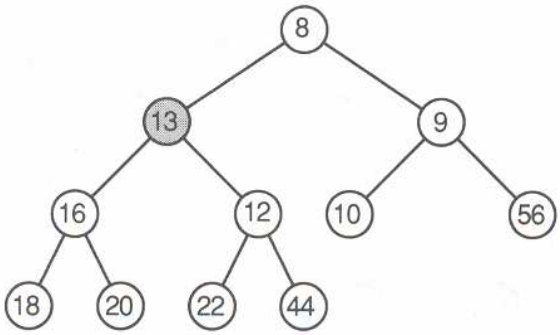
Πρόβλημα: Το προκύπτον δένδρο μπορεί να μην είναι μερικώς διατεταγμένο δένδρο.

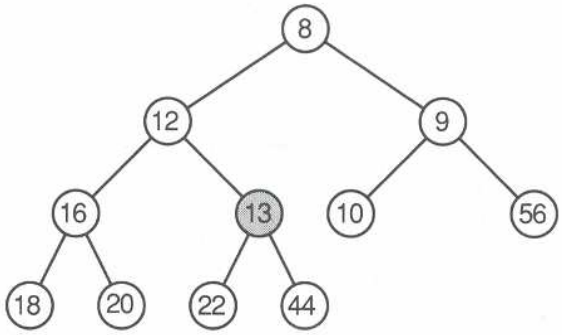


Υλοποίηση DeleteMin()

Παρατήρηση: Η μερική διάταξη καταστρέφεται μόνο στη ρίζα.

Επανόρθωση διάταξης: Ανταλλάσσουμε (swapping) τα δεδομένα της ρίζας με τα δεδομένα ενός από τα παιδιά της (εκείνου με μικρότερη προτεραιότητα). Επαναλαμβάνουμε την ίδια διαδικασία για το παιδί, μέχρι είτε να φθάσουμε σε κόμβο που η ανταλλαγή δεν επιφέρει πρόβλημα στη μερική διάταξη του παιδιού ή μέχρι να φθάσουμε σε κόμβο φύλλο.





Υλοποίηση DeleteMin() - Σωροί

Αν το δένδρο είχε λογαριθμικό ύψος, ποια θα ήταν η πολυπλοκότητα της DeleteMin()?

Πως επιλέγουμε το φύλλο που πρέπει να διαγραφεί?

Αν το δένδρο είναι ένα πλήρες δυαδικό δένδρο, μπορούμε να το αποθηκεύσουμε σε ένα πίνακα (βλέπε διαφάνεια 12, Ενότητα 4).

Το δεξιότερο φύλλο με μέγιστο βάθος στο δένδρο είναι ο τελευταίος αποθηκευμένος κόμβος στον πίνακα. Η θέση του πίνακα που περιέχει αυτό τον κόμβο μπορεί να καθοριστεί αν γνωρίζουμε τον αριθμό των κόμβων και την διεύθυνση του $1^{ου}$ στοιχείου του πίνακα.

Η διαγραφή του φύλλου αυτού δεν επηρεάζει την ιδιότητα ισοζυγισμού του δένδρου.

Ένα πλήρες μερικώς διατεταγμένο δένδρο υλοποιημένο με στατικό τρόπο (δηλαδή με πίνακα), ονομάζεται σωρός.

Ο σωρός είναι εξαιρετικά αποδοτική δομή για την υλοποίηση ουρών προτεραιότητας.

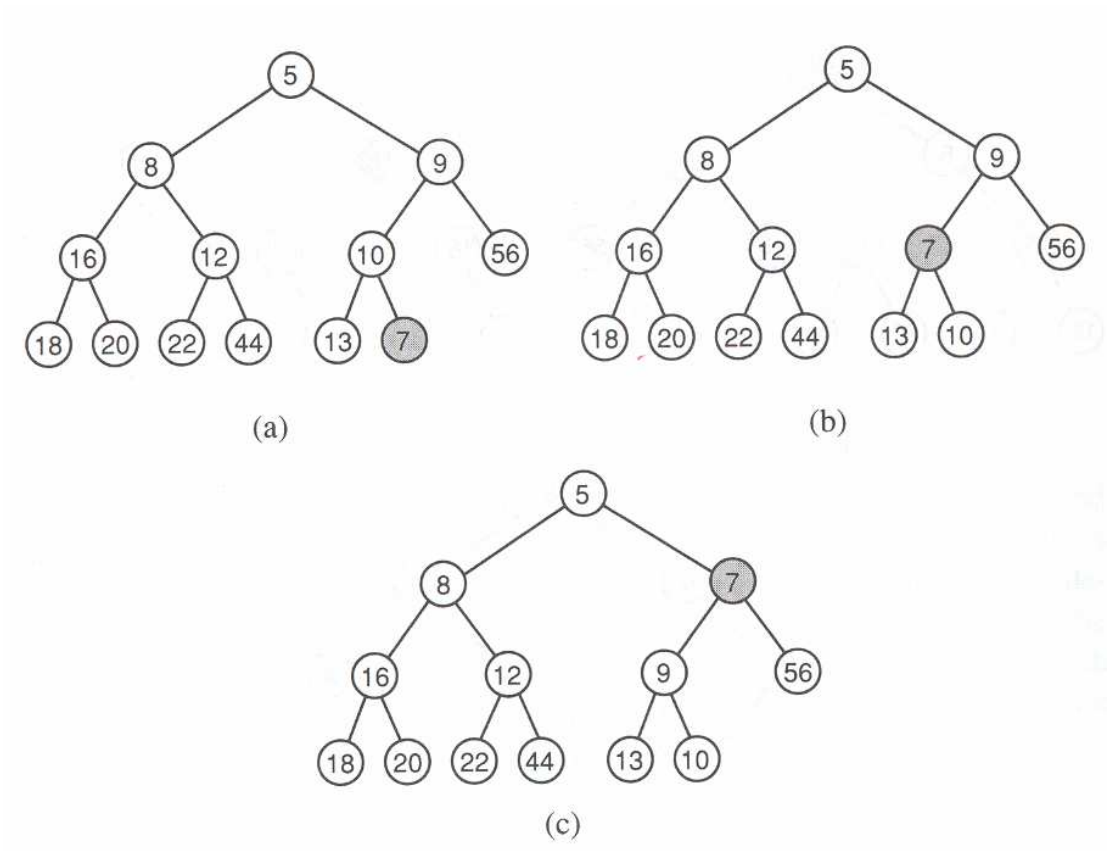
Υλοποίηση Insert()

Εισάγουμε το νέο στοιχείο σαν δεξιότερο φύλλο.

Η ιδιότητα της μερικής διάταξης ίσως καταστρέφεται άλλα μόνο για τον πατέρα του φύλλου αυτού.

Αν αυτό συμβαίνει, ανταλλάσσουμε τα δεδομένα του παιδιού με εκείνα του πατέρα και επαναλαμβάνουμε μέχρι είτε το νέο στοιχείο να φθάσει σε κάποιο επίπεδο στο οποίο ο πατέρας του έχει μικρότερη προτεραιότητα, ή να φθάσει στη ρίζα.

Ποια είναι η χρονική πολυπλοκότητα της Insert()?



Εισαγωγή σε Σωρό

```
#define N 1000

struct element {
                key Key;
                info data;
                };
typedef struct element ELEMENT;
typedef  struct element *ELEMENT_PTR;

struct heap {
             ELEMENT_PTR Table[N];
            int size;
            };
typedef struct heap *HEAP_PTR;
```

procedure HeapInsert(**key** K, **info** I, **heap_ptr** h):
/ Insert the pair <K,I> into heap h */*

```
    H = h->Table;
    n = h->size;

    if (n == N) then error;   /* Heap is full */

    m = n;   /* m is an integer "pointer" that  moves up a path in the tree */
    while (m > 0 and K < H[⌊(m-1)/2⌋]->Key) do
        H[m]->key  = H[⌊(m-1)/2⌋]->Key;
        H[m]->data = H[⌊(m-1)/2⌋]->data;
        m = ⌊(m-1)/2⌋;

    H[m]->Key = K;
    H[m]->data = I;
    h->Size = n+1;
```

Διαγραφή από Σωρό

```
function HeapDeleteMin(heap h): info
/* Delete an item of smallest priority from heap h, and return it */

H = h->Table;
n = h->Size;

if (n == 0) then error;      /* Heap is empty */

I = H[0]->data;              /* The item to be returned */
h->size = n-1;               /* The new size of the heap */

if (n == 1) then return;     /* heap is now empty */

K = H[n-1]->Key;  /* priority value of the item to be moved */
m = 0;      /* m is an integer "pointer" that moves down the tree */
while ((2m+1 < n and K > H[2m+1]->Key) || (2m+2 < n
                                and K > H[2m+2]->Key)) do
      if (2m +2 < n) then
            if (H[2m+1]->Key < H[2m+2]->Key) then
                  p = 2m+1;
            else p = 2m+2;
      else p = n-1;

      H[m]->Key = H[p]->Key;
      H[m]->data = H[p]->data;
      m = p;

H[m]->Key = H[n-1]->Key;
H[m]->data = H[n-1]->data;

return I;
```

Ένωση Ξένων Συνόλων
(Disjoint Sets with Union)

$S_1, \ldots, S_k$: ξένα υποσύνολα ενός συνόλου **U**, δηλαδή $S_i \cap S_j = \varnothing$, αν $i \neq j$, και $S_1 \cup \ldots \cup S_k = \mathbf{U}$.

Λειτουργίες που υποστηρίζονται:

- *MakeSet(X):* επιστρέφει ένα νέο σύνολο που περιέχει μόνο το στοιχείο X.
- *Union(S,T):* επιστρέφει το σύνολο $S \cup T$, το οποίο αντικαθιστά τα S, T.
- *Find(X):* επιστρέφει το σύνολο S στο οποίο ανήκει το στοιχείο X.

Υλοποιήσεις του Αφηρημένου Τύπου Δεδομένων «Ένωση Ξένων Συνόλων»

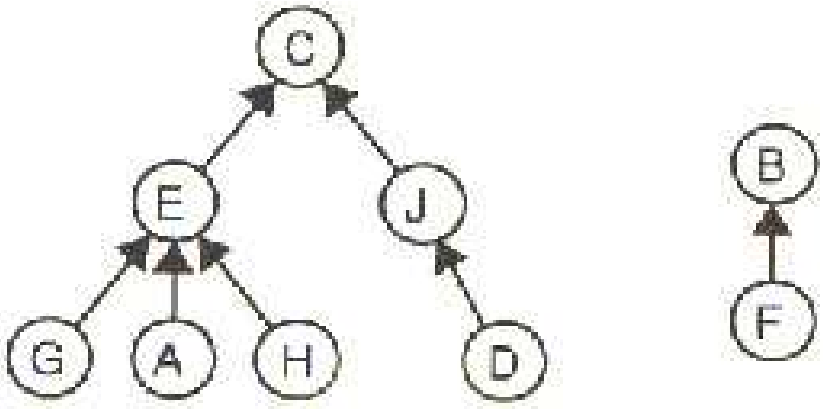
Up-Tree

Είναι δένδρο στο οποίο κάθε κόμβος διατηρεί μόνο ένα δείκτη στον πατέρα του (έτσι όλοι οι δείκτες δείχνουν προς τα πάνω).

Ένας κόμβος μπορεί να έχει οποιονδήποτε αριθμό παιδιών.

Το σύνολο U είναι ένα δάσος από Up-trees.

Κάθε τέτοιο δένδρο περιέχει τα στοιχεία ενός από τα ξένα σύνολα. Το στοιχείο της ρίζας παίζει και το ρόλο identifier για το σύνολο.



Up-Trees

Υλοποίηση Find(X):

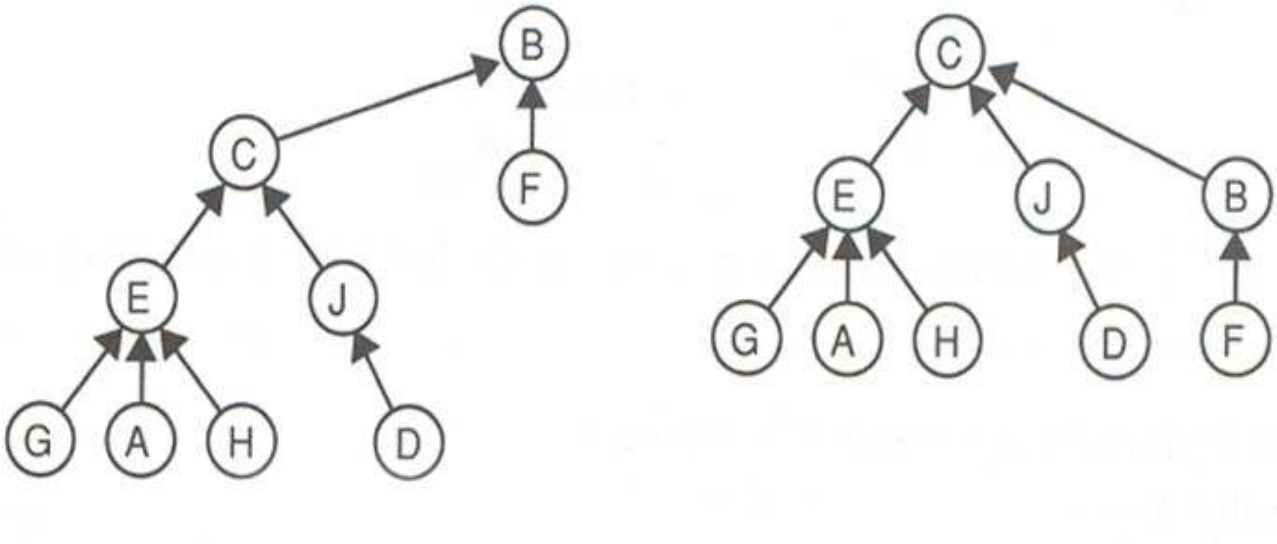
Ακολούθησε τους δείκτες από τον κόμβο προς τα πάνω ως τη ρίζα.

Έλεγχος αν ένα στοιχείο X ανήκει στο σύνολο S:

Ελέγχουμε αν η Find(X) επιστρέφει S.

Υλοποίηση Union(S,T):

Κάνε τη ρίζα του ενός δένδρου να δείχνει στη ρίζα του άλλου.



Up-Trees

Πόσο αποτελεσματικά είναι τα Up-trees για την υλοποίηση ένωσης ξένων συνόλων?

Ποια είναι η πολυπλοκότητα της Union()?

Ποια είναι η πολυπλοκότητα της Find()?

Και πάλι θέλουμε να κρατήσουμε το δένδρο ισοζυγισμένο, και για αυτό θα πρέπει να είμαστε πολύ προσεκτικοί με το πως υλοποιούμε την Merge().

Παράδειγμα

Έστω ότι έχουμε n σύνολα με ένα στοιχείο το καθένα.

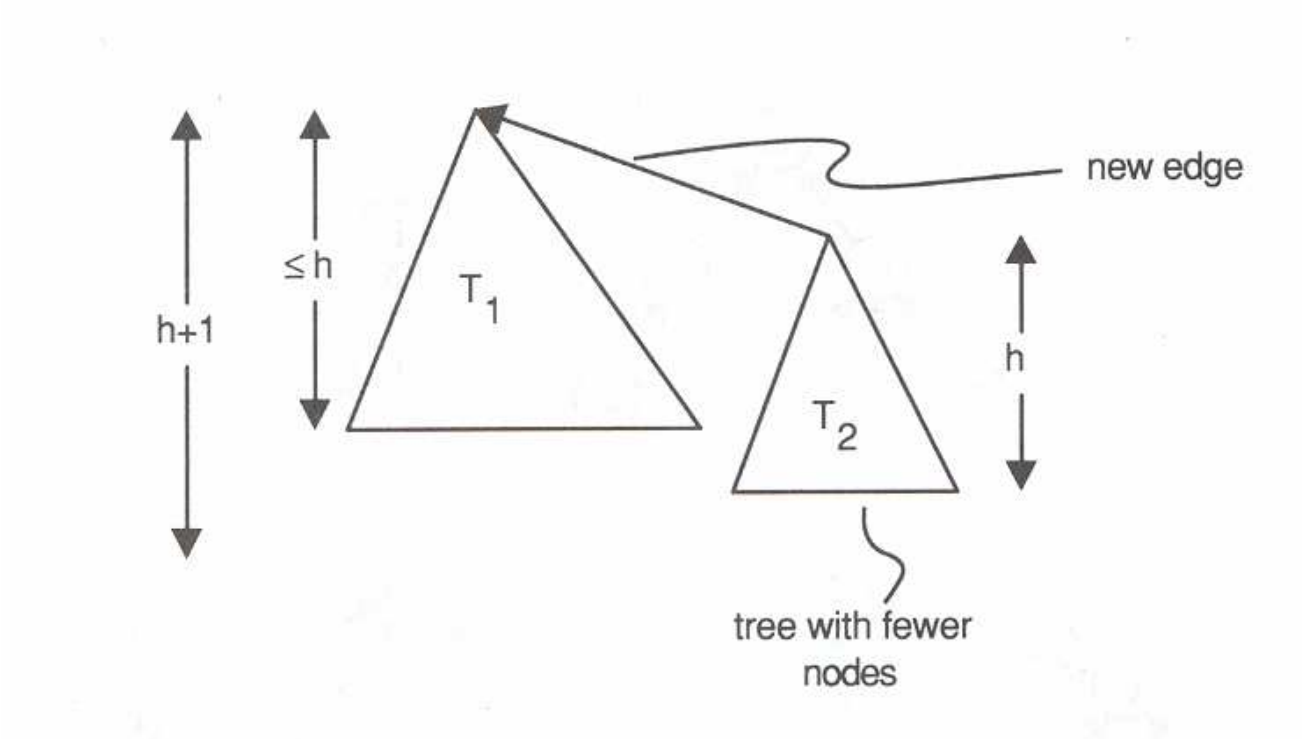
Ποιο είναι το χειρότερο ύψος δένδρου που μπορούμε να πάρουμε και πως πρέπει να εκτελέσουμε τη Merge() για να γίνει αυτό?

Up-Trees

Στρατηγική για μείωση ύψους δένδρου

«Πάντα συνενώνουμε το μικρό δένδρο στο μεγάλο, δηλαδή κάνουμε τη ρίζα του μικρού δένδρου να δείχνει στη ρίζα του μεγάλου δένδρου και όχι αντίστροφα».

Ένα δένδρο είναι μεγαλύτερο από ένα άλλο αν έχει περισσότερους κόμβους.



Up-Trees

Κάθε κόμβος είναι ένα struct με πεδία: κάποια πληροφορία για το στοιχείο, το δείκτη **parent** στο γονικό κόμβο, και ένα μετρητή **count**, που χρησιμοποιείται μόνο αν ο κόμβος είναι η ρίζα και περιέχει τον αριθμό των κόμβων στο up-tree.

```
function UpTreeFind(pointer P): pointer
/* return the root of the tree containing P */

    if (p == NULL) error;
    r = p;
    while (r->parent != NULL) do
        r = r->parent
    return r;
```

```
function UpTreeUnion(pointer S,T):pointer
/* S and T are roots of up-trees */
/* returns result of merging smaller into larger */

    if (S == NULL || P == NULL) return;
    if (S->count >= T->count) {
        S->count = S->count + T->count;
        T->parent = S;
        return S;
    }
    else {
        T->count = T->count + S->count;
        S->parent = T;
        return T;
    }
```

Up-Trees

Λήμμα

Έστω ότι Τ είναι ένα up-tree που αναπαριστά ένα σύνολο μεγέθους n, το οποίο δημιουργήθηκε με τη συνένωση n συνόλων μεγέθους 1 χρησιμοποιώντας τον παραπάνω αλγόριθμο. Τότε, το ύψος του Τ είναι το πολύ log n.

Ποια είναι η πολυπλοκότητα της Find(X)?

Υπάρχει όμως ένα λεπτό σημείο: Πως βρίσκουμε τη θέση του X στο up-tree?

Η Find(X) εμπεριέχει μια LookUp(). Πως θα υλοποιήσουμε αυτή τη LookUp?

Περιπτώσεις

1. Ο χώρος των κλειδιών είναι μικρός (π.χ. έχω 100 κλειδιά συνολικά):

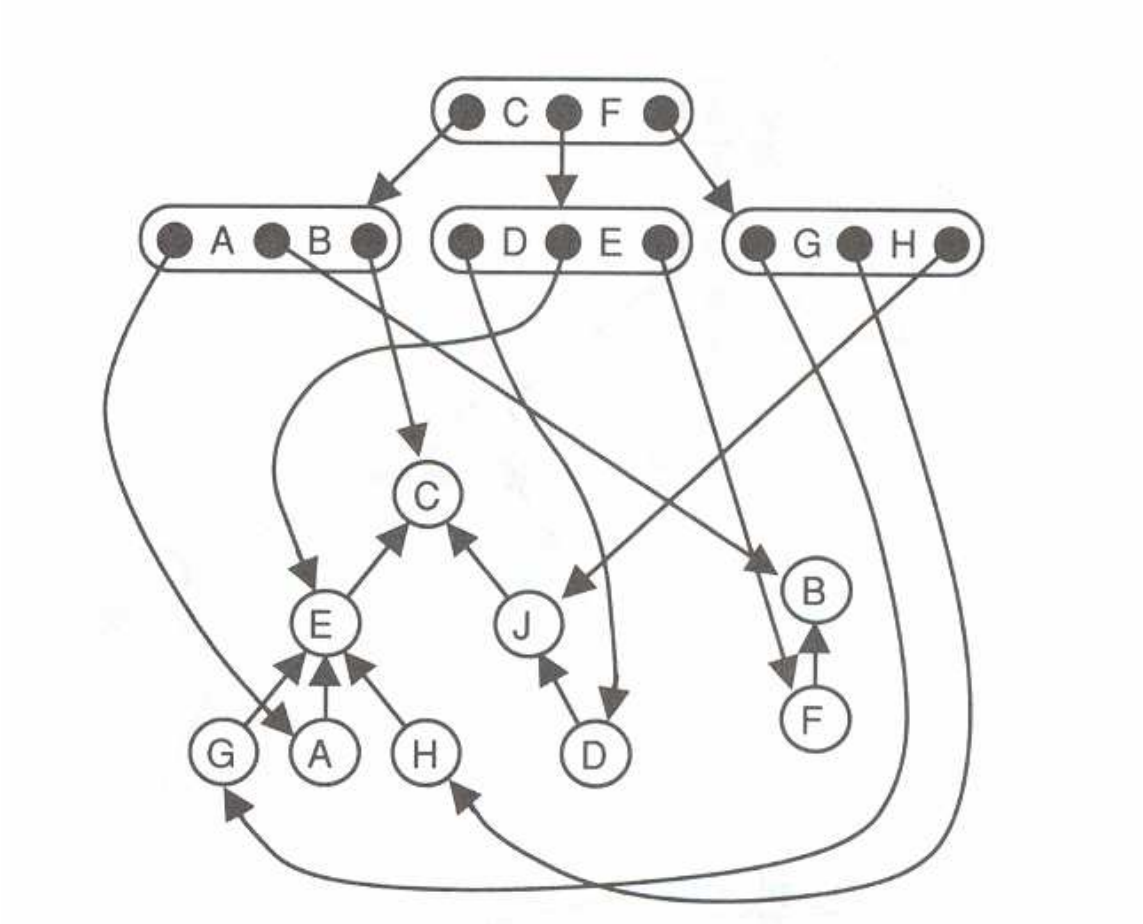
   Διατηρούμε πίνακα μεγέθους 100 που περιέχει δείκτες σε κάθε ένα από αυτά τα κλειδιά, οπότε η LookUp υλοποιείται σε σταθερό χρόνο.

Up-Trees

Πως βρίσκουμε τη θέση του X στο up-tree? (συνέχεια)

2. Ο χώρος είναι μεγάλος:

   Χρησιμοποιούμε μια βοηθητική δενδρική δομή (από αυτές που υλοποιούν λεξικά) που κάθε ένας κόμβος της δείχνει σε ένα από τα κλειδιά.



Ποια είναι η πολυπλοκότητα της Find(X) με τα νέα δεδομένα?

Up-Trees

Στρατηγική Συμπίεσης Μονοπατιού

«Κατά τη διάρκεια εκτέλεσης μιας Find(X) κάνε το parent πεδίο κάθε κόμβου στο μονοπάτι που διατρέχεις από τον κόμβο με κλειδί X στη ρίζα να δείχνει στη ρίζα».

Τι αλλαγές επιφέρει η στρατηγική συμπίεσης μονοπατιού στην απόδοση?

- Οι MakeEmptySet() και Union() εξακολουθούν να χρειάζονται σταθερό χρόνο.
- Η Find() αρχικά εκτελείται μέσα στον ίδιο ακριβώς χρόνο όπως χωρίς να εφαρμόζεται η στρατηγική, αλλά μετά την εκτέλεση μερικών Find(), η πολυπλοκότητά της γίνεται σχεδόν σταθερή.

Up-Trees

Στρατηγική Συμπίεσης Μονοπατιού (συνέχεια)

Τι θα πει σχεδόν σταθερή?

Για κάθε j, έστω F(j) η αναδρομική συνάρτηση που ορίζεται ως εξής: $F(0) = 1$ και $F(j+1) = 2^{F(j)}$, $j \geq 0$.

Οι τιμές της F(j) αυξάνουν τρομερά γρήγορα με το j, π.χ., για $j = 5$, $\Phi(5) = 2^{65536} \approx 10^{19728}$. Ο αριθμός αυτός είναι τρομερά μεγάλος (η διάμετρος του σύμπαντος είναι $\approx 10^{40}$)!!!

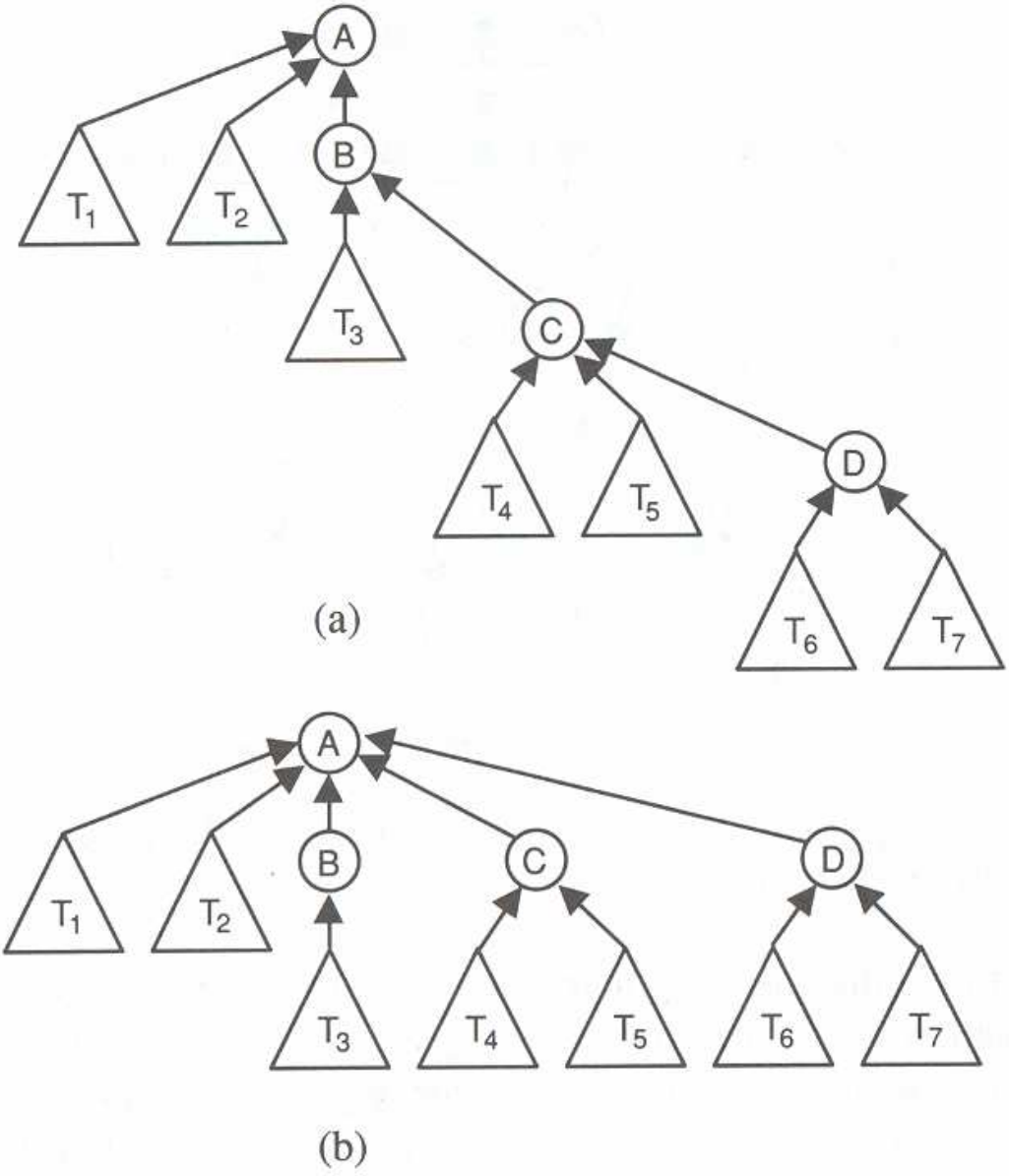
Ορίζουμε την log*n να είναι η αντίστροφη της F:

$\log^* n = \underbrace{\log \log \log \ldots \log}_{j \text{ φορές}} n$

Οι τιμές της log*n μειώνονται τρομερά γρήγορα με το n, π.χ., log*n είναι ≤ 5 για οποιαδήποτε «χρήσιμη» τιμή του n.

Αν η Find() εκτελείται σε χρόνο O(log*n) είναι επομένως σαν να είναι σταθερή! Αυτό συμβαίνει μετά από πολλαπλές εκτελέσεις της Find().

Συμπίεση Μονοπατιού



Αποτέλεσμα Find(D) όταν χρησιμοποιείται η στρατηγική συμπίεσης μονοπατιού.

ΕΝΟΤΗΤΑ 10
Αλφαριθμητικά (Strings)

Strings

Ένα *αλφαριθμητικό* είναι μια ακολουθία χαρακτήρων.

Ένα string μπορεί να είναι πολύ μεγάλο:

- αρχείο, ή ακόμη και
- σύνολο από αρχεία (π.χ. μια εγκυκλοπαίδεια)

Είναι πολύ σημαντικό να βρίσκουμε τρόπους να αναπαριστούμε strings που να ελαχιστοποιούν το χώρο που απαιτείται για την αποθήκευση τους.

Σ: αλφάβητο (σύνολο χαρακτήρων από το οποίο έχει δημιουργηθεί το string).

Τρόπος αποθήκευσης strings

Κωδικοποιούμε κάθε χαρακτήρα του Σ με μια ακολουθία από bits, που ονομάζεται κωδικοποίηση (encoding) και στη συνέχεια αποθηκεύουμε τις ακολουθίες που αντιστοιχούν στους διαδοχικούς χαρακτήρες τη μια μετά την άλλη.

Πόσα bits χρειάζονται για να αναπαραστήσω $|\Sigma|$ διαφορετικούς χαρακτήρες?

Τρόπος αποθήκευσης strings

Αν η ακολουθία των χαρακτήρων ήταν τυχαία, δεν θα υπήρχαν πολλοί τρόποι για να βελτιώσουμε τον αριθμό των bits που απαιτούνται για την αναπαράσταση ενός κειμένου.

Όμως συνήθως στην πράξη μερικοί χαρακτήρες συναντώνται πολύ πιο συχνά από ότι άλλοι (π.χ., το e συναντιέται πολύ πιο συχνά από το w και αυτό επίσης πολύ πιο συχνά από το @).

Πρόβλημα

Δεδομένου ενός string w με χαρακτήρες από ένα αλφάβητο Σ, πως μπορούμε να το αποθηκεύσουμε έτσι ώστε:

- να χρησιμοποιήσουμε όσο το δυνατόν μικρότερη ακολουθία από bits,
- να μπορούμε να ανακτήσουμε το string.

Το string w που θέλουμε να κωδικοποιήσουμε είναι το **text** μας (κείμενο), ενώ η διαδικασία ονομάζεται **κωδικοποίηση** (encoding) ή **συμπίεση** (compressing).

Τρόπος Αποθήκευσης String

Ιδέα

Οι χαρακτήρες που συναντώνται συχνά θα πρέπει να κωδικοποιούνται με μικρή ακολουθία από bits, ενώ εκείνοι που συναντώνται σπάνια θα πρέπει να έχουν μακρύτερες ακολουθίες bits ως κωδικοποιήσεις.

Κατά την υλοποίηση θα πρέπει κάπως να αποθηκεύσουμε για κάθε χαρακτήρα την ακολουθία από bits που αντιστοιχεί σε κάθε χαρακτήρα, καθώς και την ακολουθία από bits που αντιστοιχεί στην κωδικοποίηση του κειμένου μας.

Υπάρχει ένα αρνητικό στη μέθοδο αυτή:

Πως γνωρίζουμε με τι συχνότητα εμφανίζονται οι χαρακτήρες μέσα στο κείμενο μας?

Θα έπρεπε να διαβάζουμε το κείμενο μας 2 φορές, 1 για να υπολογίσουμε τις συχνότητες και 1 για να κάνουμε την κωδικοποίηση.

Η ανάγνωση ενός κειμένου 2 φορές στην καλύτερη περίπτωση δημιουργεί προβλήματα στην απόδοση του αλγορίθμου και στη χειρότερη είναι αδύνατη (π.χ., το κείμενο μπορεί να λαμβάνεται μέσω δικτύου ή να παράγεται από άλλο πρόγραμμα).

Τρόπος Αποθήκευσης String

Η χρήση μεταβλητών ακολουθιών από bits για την αναπαράσταση διαφορετικών χαρακτήρων οδηγεί στο εξής πρόβλημα:

Έστω:

Ε αναπαρίσταται με 101,

Τ αναπαρίσταται με 110

Q αναπαρίσταται με 101110

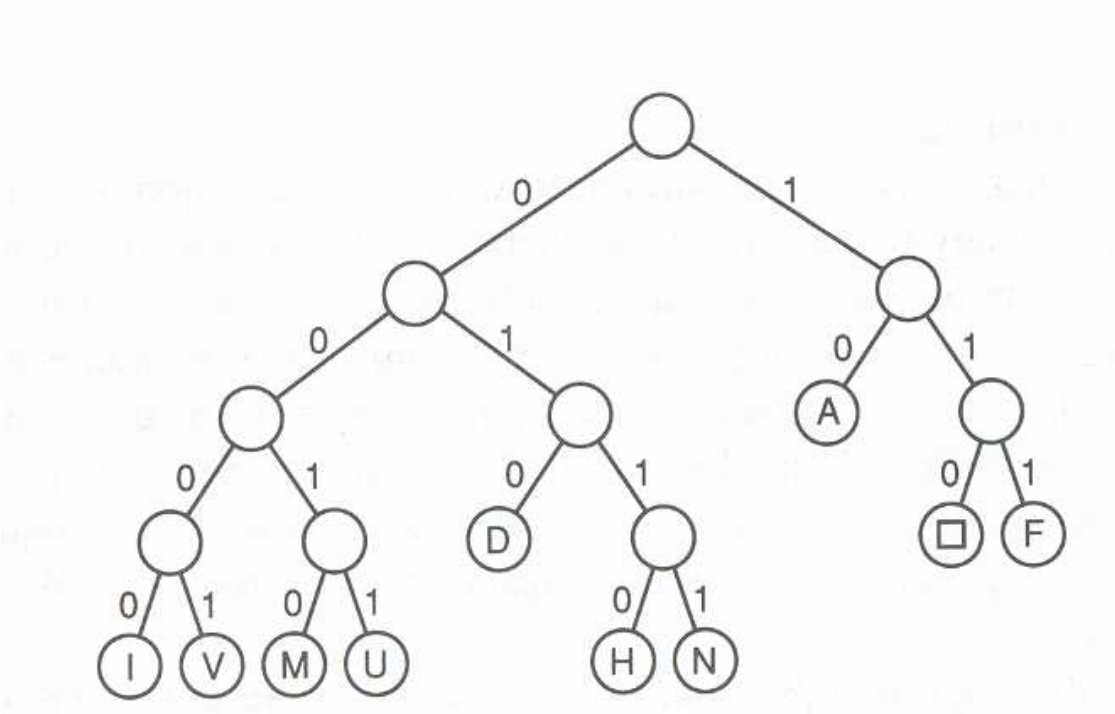
Πως θα μπορούσαμε να ξεχωρίσουμε την κωδικοποίηση του Q από εκείνη του string ΕΤ?

Το πρόβλημα δημιουργείται επειδή η κωδικοποίηση του Q ξεκινά με την ακολουθία από bits που κωδικοποιεί το Ε!

Κανόνας:

Αν δεν υπάρχουν χαρακτήρες c_1, c_2 τ.ω. η κωδικοποίηση του ενός εμπεριέχει την κωδικοποίηση του άλλου σαν πρόθεμα, τότε δεν υπάρχουν και strings w_1, w_2 τ.ω. η κωδικοποίηση του w_1 είναι ίδια με εκείνη του w_2.

Δένδρα Κωδικοποίησης



Κάθε φύλλο έχει ένα πεδίο τύπου char που περιέχει έναν από τους χαρακτήρες του Σ.

Κωδικοποίηση χαρακτήρα

Ακολουθούμε το μονοπάτι από τη ρίζα στο φύλλο που αντιστοιχεί σε αυτό τον χαρακτήρα, προσθέτοντας το 0 στην ακολουθία κάθε φορά που κινούμαστε προς τα αριστερά και το 1 κάθε φορά που κινούμαστε προς τα δεξιά μέσα στο δένδρο.

Τι μήκος έχει η ακολουθία κωδικοποίησης κάθε χαρακτήρα?

Το δένδρο ονομάζεται **δένδρο κωδικοποίησης**.

Δένδρα Κωδικοποίησης

Πως κωδικοποιούμε το string AIDA FAN δεδομένου του δένδρου κωδικοποίησης της προηγούμενης διαφάνειας?

Πως αποκωδικοποιούμε την ακολουθία bits 001010011110? Σε ποιο string αντιστοιχεί η ακολουθία αυτή?

Αλγόριθμος

Ξεκινώντας από τη ρίζα προχώρησε προς τα φύλλα του δένδρου χρησιμοποιώντας ως οδηγό τα bits της ακολουθίας: Αν το τρέχον bit της ακολουθίας είναι 1 πήγαινε δεξιά, διαφορετικά αριστερά.

Αν φθάσεις σε φύλλο τύπωσε τον χαρακτήρα που αντιστοιχεί στο φύλλο και ξεκίνα πάλι από τη ρίζα επαναλαμβάνοντας τα παραπάνω βήματα, μέχρι όλη η ακολουθία από bits να εξαντληθεί.

Υπάρχει κάποιο αρνητικό σε αυτή τη μέθοδο?

Είναι δυνατόν να ξεκινήσουμε αποκωδικοποίηση από οποιοδήποτε σημείο της ακολουθίας από bits?

Κωδικοποίηση Huffman

Το βασικό πρόβλημα παραμένει ακόμη ανεπίλυτο:

Πως θα κατασκευάσουμε το δένδρο έτσι ώστε να έχουμε την καλύτερη δυνατή κωδικοποίηση?

Έστω ότι για κάθε χαρακτήρα c_j γνωρίζουμε τον αριθμό των φορών f_j που ο c_j συναντάται στο w.

Κατασκευή του Τ:

Δημιούργησε έναν κόμβο-φύλλο για κάθε χαρακτήρα;

Κάθε κόμβος πρέπει να περιέχει έναν ακέραιο weight (βάρος του κόμβου), το οποίο για τον κόμβο που αντιστοιχεί στο χαρακτήρα c_j το έχουμε αρχικοποιήσει σε f_j;

Εκτέλεσε επαναληπτικά το εξής βήμα:

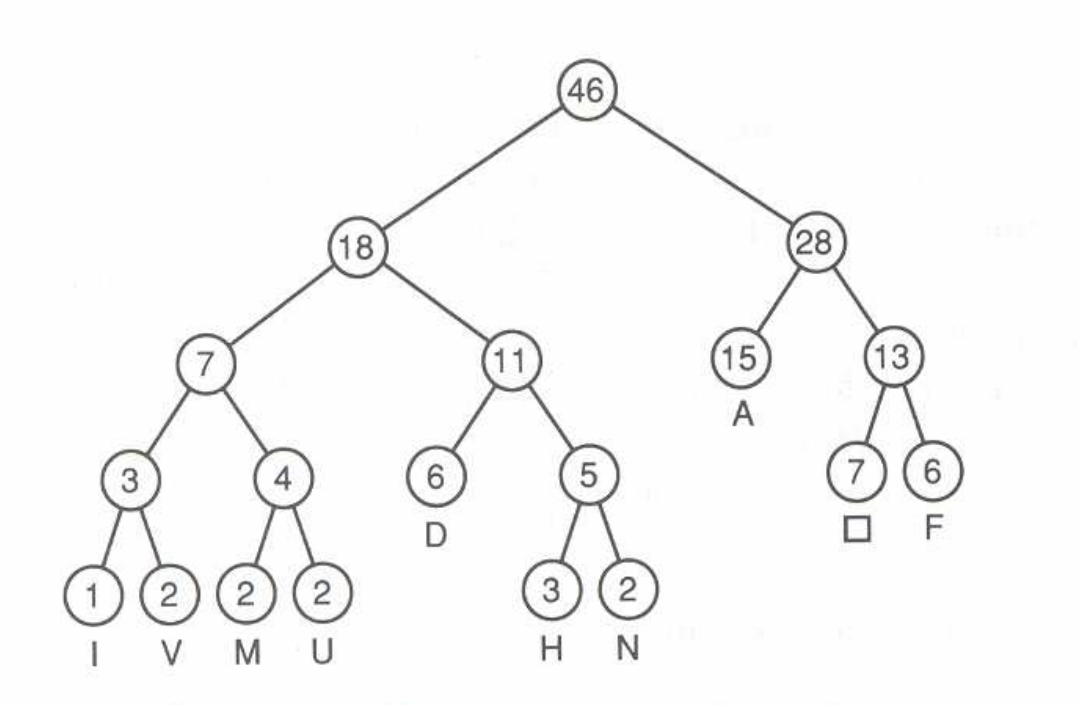
> Διάλεξε 2 κόμβους v1 και v2, με ελάχιστο βάρος και αντικατέστησε τους με έναν κόμβο που (1) έχει παιδιά τα v1 και v2, και (2) έχει βάρος το άθροισμα των βαρών των v1 και v2.

μέχρι να απομείνει μόνο ένας κόμβος, που θα αποτελέσει τη ρίζα του δένδρου.

Το δένδρο που κατασκευάζεται με αυτό τον αλγόριθμο ονομάζεται **δένδρο Huffman**.

Κωδικοποίηση Huffman

Παράδειγμα



Μέσα στους κόμβους εμφανίζεται το βάρος τους.

Παρατηρήσεις

Χαρακτήρες που συναντώνται με μεγαλύτερη συχνότητα βρίσκονται κοντά στη ρίζα και για αυτό έχουν μικρότερη ακολουθία κωδικοποίησης.

Η κωδικοποίηση που παρέχει το δένδρο Huffman είναι βέλτιστη (κανένα άλλο δένδρο κωδικοποίησης δεν οδηγεί σε μικρότερη αναπαράσταση του w).

Προβλήματα Εφαρμογής Μεθόδου Huffman

Ποιο είναι το σημαντικότερο πρόβλημα?

Και πάλι χρειάζεται μια δεύτερη ανάγνωση του κειμένου!!!

Τρόποι Αποφυγής 2ης Ανάγνωσης

Στατική Κωδικοποίηση Huffman

Φιξάρουμε το δένδρο κωδικοποίησης αρχικά και το χρησιμοποιούμε για όλα τα κείμενα.

- ✓ Αυτό θα δούλευε καλά π.χ., με κείμενα της αγγλικής γλώσσας (ενδεχομένως που αναφέρονται σε παραπλήσια θέματα)
- ✓ Δεν χρειάζεται να αποθηκεύονται πληροφορίες για την κωδικοποίηση με κάθε ένα από τα κείμενα. Αυτό γίνεται μια μόνο φορά για όλα τα κείμενα.

Κωδικοποίηση Huffman με προσαρμογή

Αρχικά θεωρούμε ότι όλοι οι χαρακτήρες έχουν βάρος 0 και σιγά-σιγά προσαρμόζουμε τα βάρη και το δένδρο (καθώς διαβάζουμε το κείμενο).

Κωδικοποίηση κατά Lempel-Ziv

Υπάρχουν ακολουθίες χαρακτήρων που συναντώνται με μεγάλη συχνότητα.

Παράδειγμα: the, ion, ing

Ιδέα

Παρότι δεν μπορούμε να τα πάμε καλύτερα από ότι η κωδικοποίηση κατά Huffman, όταν συσχετίζουμε κωδικούς με χαρακτήρες, μπορούμε ωστόσο να πετύχουμε καλύτερη συμπίεση αν αποδώσουμε κωδικούς σε ακολουθίες χαρακτήρων.

Περιγραφή Αλγόριθμου

Χρησιμοποιείται λεξικό που αποθηκεύει ακολουθίες χαρακτήρων (και τους κωδικούς που τους αναλογούν) που επιλέγονται δυναμικά κατά την ανάγνωση του κειμένου.

s: ακολουθία χαρακτήρων,

#(s): κωδικός της s

Όλοι οι κωδικοί έχουν το ίδιο μήκος σε bits (συνήθως 12 bits ο καθένας).

Αρχικά έχουμε αποδώσει ένα κωδικό σε κάθε χαρακτήρα του αλφάβητου και έχουμε τοποθετήσει όλους αυτούς τους χαρακτήρες στο λεξικό.

Κωδικοποίηση κατά Lempel-Ziv

Αλγόριθμος Κωδικοποίησης

w: string που θέλουμε να κωδικοποιήσουμε

Σε κάθε βήμα:

(1) βρες το μεγαλύτερο πρόθεμα p του w που υπάρχει στο λεξικό,

(2) αντικατέστησε το πρόθεμα με τον κωδικό του (που είναι αποθηκευμένος στο λεξικό),

(3) διέγραψε το p από το w.

Το πρόθεμα p ονομάζεται **current match (τρέχον επιλεγμένο)**.

Μετά από κάθε βήμα τροποποιούμε επίσης το λεξικό:

- Προσθέτουμε ένα νέο string στο λεξικό και του αποδίδουμε τον επόμενο διαθέσιμο ακέραιο του λεξικού ως κωδικό του.

- Το string που επιλέγεται αποτελείται από το προηγούμενο match (current match προηγούμενου βήματος) & ένα ακόμη χαρακτήρα (τον πρώτο) από το current match.

Κωδικοποίηση κατά Lempel-Ziv

Παράδειγμα

COCOA AND BANANAS

Step	Output	Add to Dictionary	Step	Output	Add to Dictionary
1	#(C)	—	8	#(D)	ND
2	#(O)	CO	9	#(□)	D□
3	#(CO)	OC	10	#(B)	□B
4	#(A)	COA	11	#(AN)	BA
5	#(□)	A□	12	#(AN)	ANA
6	#(A)	□A	13	#(A)	ANA
7	#(N)	AN	14	#(S)	AS

Κωδικοποίηση κατά Lempel-Ziv

Αλγόριθμος Αποκωδικοποίησης

- ✓ Το κωδικοποιημένο string αποτελείται απλά από μια ακολουθία από κωδικούς αριθμούς.
- ✓ Κάθε τέτοιος αριθμός αναπαρίσταται από μια ακολουθία από bits σταθερού μήκους: μπορούμε να διαβάσουμε αυτούς τους κωδικούς έναν-έναν.

Παρατήρηση

- o Το λεξικό δεν χρειάζεται να αποθηκεύεται κάπου. Δημιουργείται δυναμικά κατά την αποκωδικοποίηση, όπως και κατά την κωδικοποίηση.
- o Το λεξικό που προκύπτει είναι ίδιο με αυτό που προκύπτει κατά την κωδικοποίηση και έτσι η αποκωδικοποίηση γίνεται σωστά.

Περιγραφή

Αρχικοποίησε το λεξικό όπως και κατά την κωδικοποίηση.

Διάβασε έναν κωδικό

Βρες τον κωδικό στο λεξικό και αποκωδικοποίησε τον κωδικό (έστω s το string στο λεξικό για τον κωδικό)

Πρόσθεσε στο λεξικό την ακολουθία χαρακτήρων που αποτελείται από την προηγούμενη ακολουθία χαρακτήρων και τον πρώτο χαρακτήρα του s.

Κωδικοποίηση κατά Lempel-Ziv

Μπορεί το λεξικό να γεμίσει?

Με κωδικούς των 12 bits, πόσους διαφορετικούς κωδικούς μπορούμε να έχουμε?

Πόσες εγγραφές μπορούμε να έχουμε στο λεξικό?

Δυνατές επιλογές όταν το λεξικό γεμίσει:

- Σταματάμε την προσθήκη νέων strings στο λεξικό και κωδικοποιούμε το υπόλοιπο κείμενο με το ήδη υπάρχον λεξικό.
- Διαγράφουμε το παλιό λεξικό και ξεκινάμε να το ξαναγεμίσουμε εξ αρχής.
- Διαγράφουμε μη συχνά χρησιμοποιούμενες ακολουθίες χαρακτήρων από το λεξικό και έτσι μένουν κάποιοι κωδικοί για επαναχρησιμοποίηση. Σε αυτή την περίπτωση θα πρέπει να κρατάμε στατιστικά.
- Διπλασιάζουμε το μέγεθος της ακολουθίας bits που αντιστοιχεί σε κάθε κωδικό έτσι ώστε να διπλασιαστεί και ο αριθμός εγγραφών που χωρούν στο λεξικό.

Σε κάθε περίπτωση:

Ο αλγόριθμος αποκωδικοποίησης θα πρέπει να εφαρμόσει ακριβώς τον ίδιο κανόνα!!!

ΕΝΟΤΗΤΑ 9
Γράφοι (Graphs)

Από τι αποτελείται ένας γράφος?

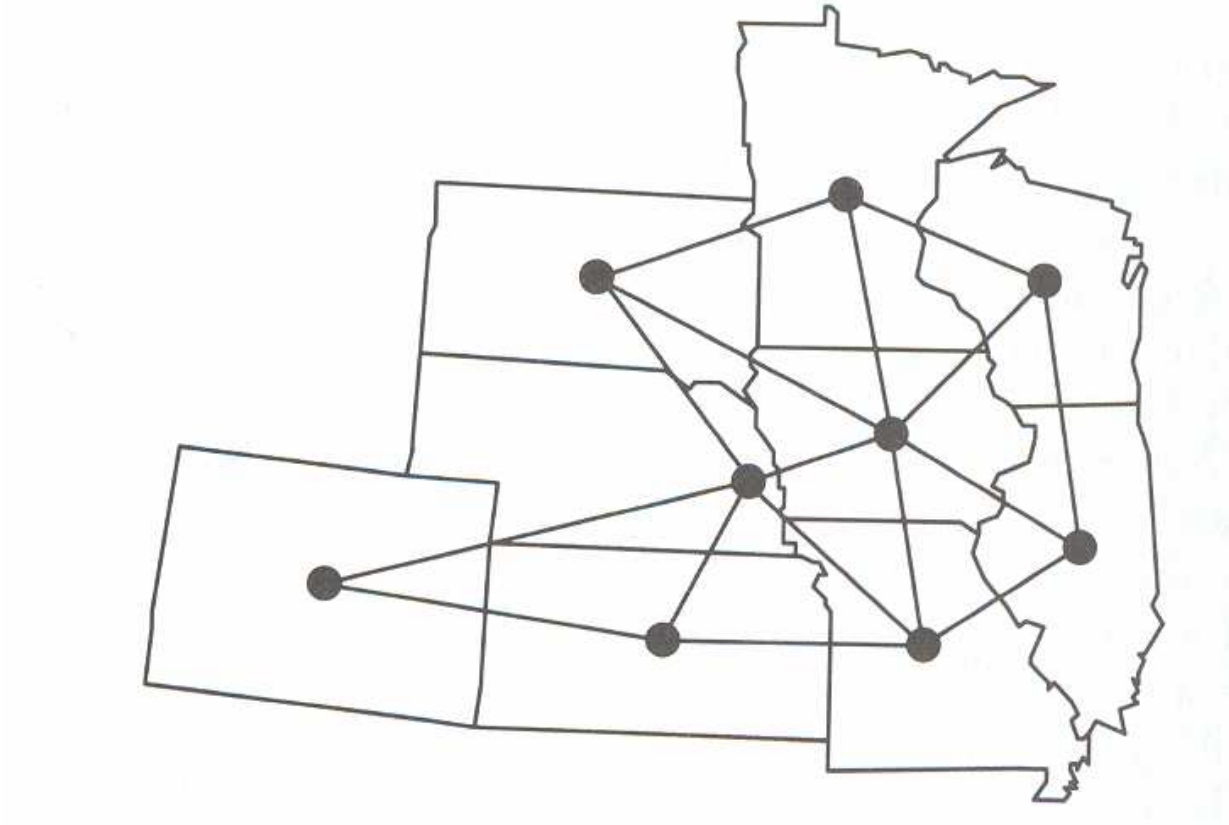
Ένας γράφος αποτελείται από ένα σύνολο από σημεία και ένα σύνολο από γραμμές που συνδέουν ζεύγη των σημείων αυτών.

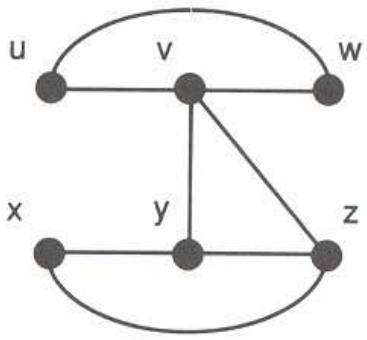
Γιατί είναι χρήσιμοι?

Γιατί μπορούν να μοντελοποιήσουν πολλά προβλήματα:

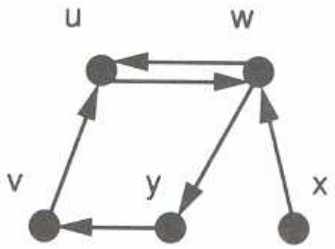
- Αεροπορικές πτήσεις μεταξύ κάποιων πόλεων.
- Συνηθίζεται στη δημιουργία χαρτών να ζωγραφίζονται γειτονικές χώρες (νομοί) με διαφορετικό χρώμα. Το πρόβλημα αυτό μπορεί να μοντελοποιηθεί σαν ένα πρόβλημα γράφων.
- Πολλά παιχνίδια μπορούν να μοντελοποιηθούν με χρήση γράφων.
- Traveling Salesman Problem (Πρόβλημα περιπλανώμενου πωλητή): Δεδομένου ενός συνόλου από πόλεις και της απόστασης μεταξύ κάθε ζεύγους πόλεων, βρείτε τη βέλτιστη διαδρομή που επισκέπτεται κάθε πόλη (δηλαδή εκείνη στην οποία διανύεται η μικρότερη απόσταση).

Παραδείγματα Γράφων

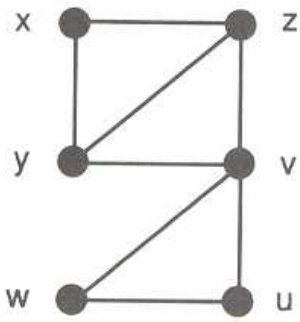




(a)



(b)



(c)

Ορισμοί – Ορολογία

Ένας **γράφος** G χαρακτηρίζεται από δύο σύνολα V και E. Το σύνολο V είναι ένα πεπερασμένο διάφορο του κενού σύνολο, που περιέχει ως στοιχεία τις **κορυφές** (vertices) ή **κόμβους** (nodes) ή **σημεία** (points) του γράφου. Το σύνολο E έχει ως στοιχεία τα ζεύγη κορυφών του γράφου, τα οποία ορίζουν τις **ακμές** (edges) ή **τόξα** (arcs) ή **συνδέσμους** (links).

V(G): σύνολο κόμβων γράφου G

E(G): σύνολο ακμών γράφου G

G(V,E): γράφος με σύνολο κορυφών V και σύνολο ακμών E.

Οι κορυφές ή οι ακμές ενός γράφου χαρακτηρίζονται από ένα μοναδικό όνομα που ονομάζεται **ετικέτα** ή **επιγραφή** (label)

Ζυγισμένος γράφος (weighted graph) ή **δίκτυο** (network) λέγεται ο γράφος, όπου κάθε ακμή χαρακτηρίζεται από έναν αριθμό που ονομάζεται βάρος (weight).

Ορισμοί – Ορολογία

Ένας γράφος είναι **μη κατευθυνόμενος** αν τα ζεύγη των κορυφών που ορίζουν τις ακμές του στερούνται διάταξης, π.χ., (υ_1, υ_2) και (υ_2, υ_1) αναφέρονται στην ίδια ακμή.

Στους **κατευθυνόμενους γράφους** κάθε ακμή συμβολίζεται με το κατευθυνόμενο ζεύγος $<\upsilon_1, \upsilon_2>$, όπου υ_1 είναι η **ουρά** (tail) και υ_2 είναι η **κεφαλή** (head) της ακμής (δηλαδή οι ακμές $<\upsilon_1, \upsilon_2>$ και $<\upsilon_2, \upsilon_1>$ είναι 2 διαφορετικές ακμές).

Ένας μη κατευθυνόμενος γράφος μπορεί να θεωρηθεί σαν ένας συμμετρικός κατευθυνόμενος γράφος.

Ορισμοί – Ορολογία

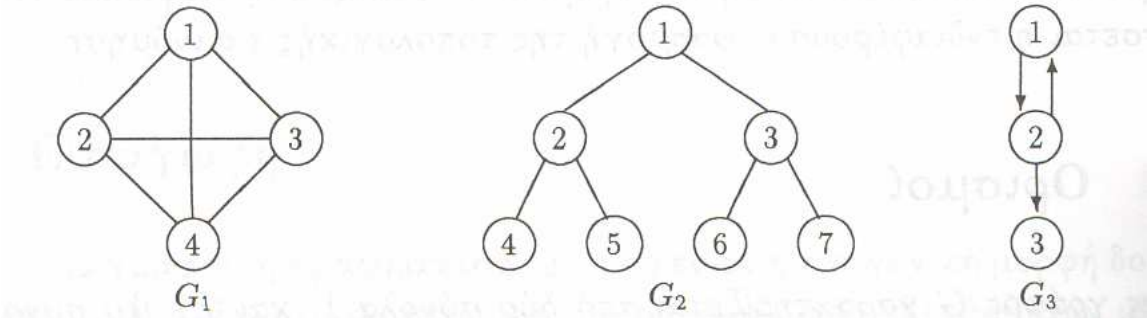
Αν (υ_1, υ_2) είναι μια ακμή του E(G), τότε οι κορυφές υ_1 και υ_2 λέγονται **διπλανές** (adjacent) ή **γειτονικές** (neighboring) και η ακμή (υ_1, υ_2) ονομάζεται **προσκείμενη** στις κορυφές υ_1 και υ_2.

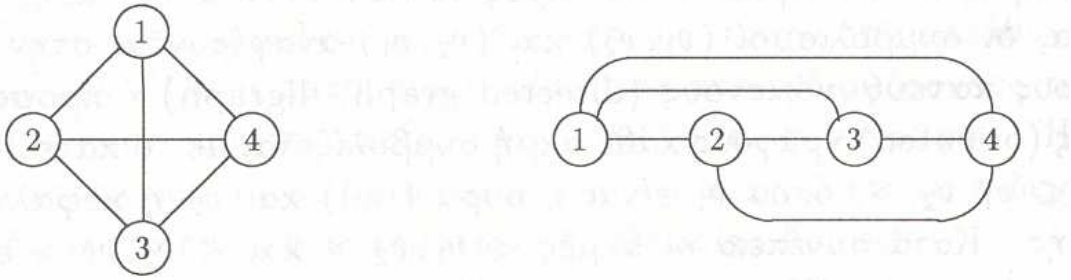
Αν δύο κορυφές υ_1 και υ_2 δεν συνδέονται μεταξύ τους με ακμή λέγονται **ανεξάρτητες** (independent).

Αν (υ_1, υ_2) είναι μια ακμή τότε η κορυφή υ_1 λέγεται **διπλανή** (adjacent) της υ_2. Επίσης, οι κορυφές υ_1, υ_2 λέγονται και **γειτονικές**.

Αν $<\upsilon_1,\upsilon_2>$ είναι μια ακμή ενός κατευθυνόμενου γράφου, τότε ο κόμβος υ_1 είναι **γειτονικός** του κόμβου υ_2, αλλά το αντίστροφο ισχύει μόνο αν και η ακμή $<\upsilon_2, \upsilon_1>$ υπάρχει επίσης στον κατευθυνόμενο γράφο.

Παράδειγμα





- $V(G_1) = \{1,2,3,4\}$,
- $E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$

- $V(G_2) = \{1,2,3,4,5,6,7\}$,
- $E(G_2) = \{(1,2), (1,3), (2,4), (2,5), (3,6), (3,7)\}$

- $V(G_3) = \{1,2,3\}$,
- $E(G_3) = \{<1,2>, <2,1>, <2,3>\}$.

Συνήθεις Λειτουργίες σε Γράφους

MakeGraph(V): επιστρέφει έναν γράφο που περιέχει όλες τις κορυφές του G και καμία ακμή.

Vertices(G): επιστρέφει V(G), το σύνολο των κόμβων του G.

Edges(G): επιστρέφει E(G), το σύνολο ακμών του G.

Neighbors(v,G): επιστρέφει το σύνολο κορυφών που είναι γειτονικές του κόμβου v στον G

AddVertex(v,G): Προσθέτει ένα νέο κόμβο με ετικέτα v στον G

AddDirectedEdge(u,v,G): προσθέτει μια νέα (κατευθυνόμενη) ακμή <u,v> που συνδέει τους κόμβους u και v στον G.

AddUndirectedEdge(u,v,G): προσθέτει τη νέα ακμή (u,v) που συνδέει τους κόμβους u,v στον G.

DeleteVertex(v,G): διαγράφει τον κόμβο v από τον G, μαζί με όλες τις ακμές που πρόσκεινται σε αυτόν.

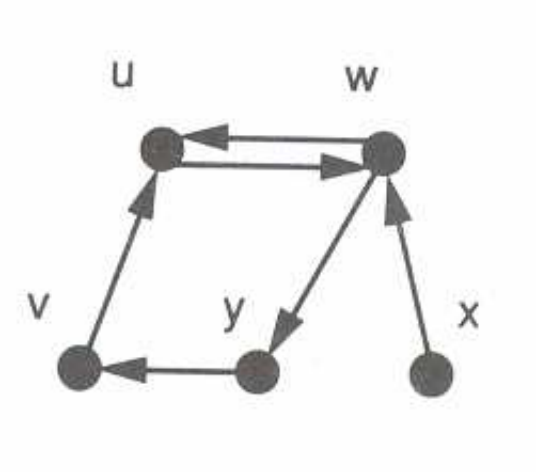
DeleteEdge(u,v,G): διαγράφει την ακμή που συνδέει τους κόμβους u,v στον G.

Αναπαράσταση Γράφων

G με κόμβους $\upsilon_1, \upsilon_2, \ldots, \upsilon_n$.

M_G: διδιάστατος πίνακας όπου $M_G[i,j] = 1$ αν ο υ_j είναι γειτονικός κόμβος του υ_i και $M_G[i,j] = 0$, διαφορετικά.

Παράδειγμα



$$\begin{array}{c} \\ u \\ v \\ w \\ x \\ y \end{array} \begin{array}{c} \begin{array}{ccccc} u & v & w & x & y \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Ο πίνακας M_G ονομάζεται **πίνακας γειτνίασης** (ή πίνακας διπλανών κορυφών) του G.

Αν ο G είναι μη κατευθυνόμενος, ο M_G είναι συμμετρικός.

$M_G[i,i] = 0$, για κάθε i.

Θετικά – Αρνητικά Αναπαράστασης Γράφων με Πίνακα Γειτνίασης

Αν δεν χρειάζεται να αποθηκευτεί κάποια πληροφορία για κάθε κόμβο, η μέθοδος είναι πολύ ελκυστική. Κάθε κόμβος ονοματίζεται από έναν ακέραιο και οι ακέραιοι αυτοί (δηλαδή τα ονόματα των κόμβων) χρησιμοποιούνται για διευθυνσιοδότηση του πίνακα.

Οι περισσότερες λειτουργίες υλοποιούνται πολύ απλά (και κάποιες πολύ αποτελεσματικά).

Αρνητικά

Διαγραφή ή εισαγωγή κόμβου στο γράφο: Το μέγεθος του πίνακα πρέπει να αλλάξει.

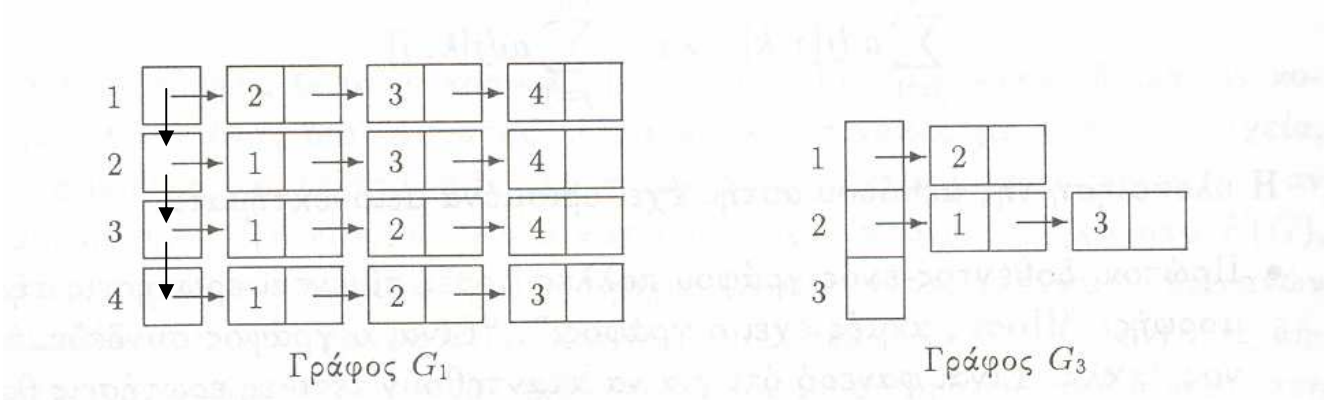
Η μέθοδος πίνακα γειτνίασης είναι απλή, αλλά δεν υποστηρίζει αποτελεσματικά όλες τις λειτουργίες.

Ποια η πολυπλοκότητα της λειτουργίας εύρεσης των γειτονικών κόμβων ενός κόμβου υ?

Θα μπορούσαμε να υλοποιήσουμε τη λειτουργία αυτή πιο αποτελεσματικά αν γνωρίζαμε πως ο κόμβος δεν έχει καθόλου γειτονικούς κόμβους?

Λίστες Διπλανών Κορυφών (ή Γειτνίασης)

Οι κόμβοι αποθηκεύονται σε μια λίστα. Κάθε κόμβος περιέχει δείκτη σε λίστα με τους γειτονικούς του κόμβους.



Θετικά

- Ένας κόμβος μπορεί να εισαχθεί ή να διαγραφεί με την ίδια ευκολία όπως μια ακμή.

Αρνητικά

- Η λειτουργία «Βρες αν δύο κόμβοι είναι γειτονικοί» αποκτά μεγαλύτερη πολυπλοκότητα από όταν χρησιμοποιείται πίνακας γειτνίασης.
- Περισσότερη μνήμη απαιτείται για την αναπαράσταση.

Πολυπλοκότητα Αλγορίθμων Γράφων

Τι σημαίνει ότι ένας αλγόριθμος γράφων έχει γραμμική ή πολυωνυμική πολυπλοκότητα?

Μέγεθος γράφου:

Αριθμός κόμβων, n ή |G|

Αριθμός ακμών, m ή |E|

Πόσες το πολύ ακμές έχει ένας γράφος με n κόμβους?

Πρόταση

Ο μέγιστος αριθμός ακμών για κάθε μη κατευθυνόμενο γράφο με n κορυφές είναι $E_{max} = n(n-1)/2$.

Τι ισχύει αν ο γράφος είναι κατευθυνόμενος?

Τι θα ήταν καλύτερο, ένας αλγόριθμος να τρέχει σε χρόνο $\Theta(n^2)$ ή σε $\Theta(m)$?

Αλγόριθμοι με πολυπλοκότητα Θ(m) συνήθως δεν μπορούν να σχεδιαστούν, αφού αν το |E| είναι μικρό, δεν αρκεί ο χρόνος ούτε για να εξεταστεί κάθε κόμβος.

Πολυπλοκότητα Αλγορίθμων Γράφων

Ένας γράφος με πολλές ακμές λέγεται **πυκνός** (συνήθως με Θ(nlogn) και πάνω ακμές).

Ένας γράφος με λίγες ακμές (συνήθως λιγότερες από O(n)) λέγεται **αραιός**.

Πολλές φορές η γνώση του αν ο γράφος είναι πυκνός ή αραιός βοηθάει στο σχεδιασμό αποτελεσματικών αλγόριθμων.

Η χρονική πολυπλοκότητα γράφων είναι συνήθως συνάρτηση τόσο του αριθμού των κόμβων, όσο και του αριθμού των ακμών, π.χ., Θ(n+m).

Άλλοι παράγοντες που επηρεάζουν σημαντικά την πολυπλοκότητα είναι η υλοποίηση (δηλαδή η μέθοδος αναπαράστασης που χρησιμοποιείται).

Παράδειγμα

Ποια η πολυπλοκότητα της λειτουργίας "foreach edge e in G" αν ο γράφος αναπαριστάται με:

- *Πίνακα γειτνίασης?*
- *Λίστες γειτνίασης?*