ΕΝΟΤΗΤΑ 4
ΣΥΝΟΛΑ - ΛΕΞΙΚΑ

Σύνολα (Sets)

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

Χρησιμότητα

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

Παραδείγματα

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

Λειτουργίες Συνόλων

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

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

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

Σύνολα – Λεξικά

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

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

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

Λεξικά

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

Υλοποίηση Λεξικών με Λίστες

Αποθήκευση των στοιχείων του συνόλου σε μια (μη-ταξινομημένη) λίστα:
- ❑ Στατική λίστα (χρήση πίνακα)
- ❑ Συνδεδεμένη Λίστα (απλά ή διπλά συνδεδεμένη).

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

Υλοποίηση Λειτουργιών

Συνδεδεμένη Λίστα

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

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

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

Στατική Λίστα

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

Υλοποίηση Λεξικών με Λίστες - Αναμενόμενο Κόστος

Υποθέτουμε πως η πιθανότητα p_j η LookUp() να ψάχνει για το j-οστό στοιχείο είναι η ίδια για κάθε j, $1 \le j \le n$. Έτσι, αν έχουμε n στοιχεία, η πιθανότητα είναι 1/n.
Έστω ότι c_j είναι το κόστος που πληρώνουμε για να βρούμε το j-οστό στοιχείο $\Rightarrow c_j = j$.
Το αναμενόμενο κόστος είναι το άθροισμα των $(c_j * p_j)$ για κάθε j:
Αναμενόμενο Κόστος =
$1*1/n+2*1/n+\ldots+n*1/n = (1+2+\ldots+n)/n = n(n+1)/(2n) = (n+1)/2 = \Theta(n)$.

Διαφορετικές Πιθανότητες για διαφορετικά κλειδιά

$K_1, \ldots, K_n$: τα κλειδιά στο σύνολο σε φθίνουσα διάταξη ως προς τη συχνότητα με την οποία αναζητούνται μέσω της LookUp().
$p_1 \ge p_2 \ge \ldots \ge p_n$: πιθανότητα μια LookUp() να ψάχνει για το $K_1, K_2, \ldots, K_n$, αντίστοιχα.
Ο αναμενόμενος χρόνος αναζήτησης ελαχιστοποιείται όταν τα στοιχεία έχουν τη διάταξη $K_1, K_2, \ldots, K_n$ στη λίστα:

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

Γιατί αυτό είναι βέλτιστο;

Αναμενόμενο Κόστος

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

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

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

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

Ευριστικά

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

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

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

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

Ευριστικό «Αλληλομετάθεσης» (Transpose)

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

Σύγκριση μεταξύ Ευριστικών

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

Ευριστικό «Μετακίνησης στην Αρχή» - Αναμενόμενο Κόστος

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

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

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

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

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

Άρα, $p(i,j) = p_i / (p_i+p_j)$.

Ευριστικό «Μετακίνησης στην Αρχή» - Αναμενόμενο Κόστος

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

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

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

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

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

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

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

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

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

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

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

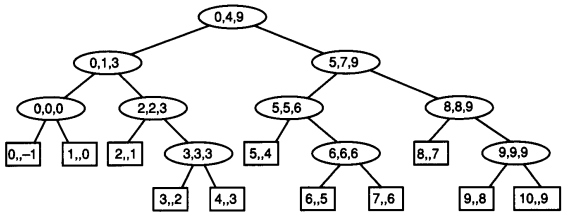
Ταξινομημένες Στατικές Λίστες

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

BinarySearch
Type *BinarySearchLookUp*(**key** K, **table** T[0..n-1])
/* Return information stored with key K in T, or nill if K is not in T */

```
left = 0;
right = n-1;
repeat forever
        if (right < left) then
                return nill;
        else
                middle = ⌊(left+right)/2⌋;
                if (K == T[middle]->Key) then
                        return T(middle]->data;
                else if (K < T[middle]->Key) then
                        right = middle-1;
                else left = middle+1;
```


Δυνατές Εκτελέσεις BinarySearchLookUp

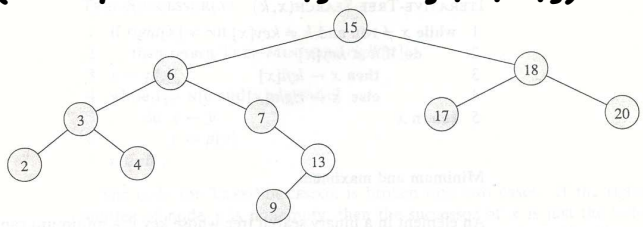


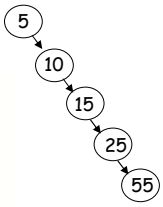
- **Κυκλικοί κόμβοι:** εσωτερικοί
- **Τετραγωνισμένοι κόμβοι:** εξωτερικοί

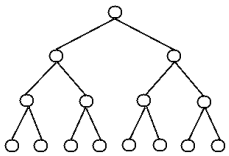
- Το δένδρο περιγράφει όλες τις δυνατές εκτελέσεις της BinarySearch() σε ένα πίνακα 10 στοιχείων.

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

Ταξινομημένα Δυαδικά Δένδρα (Δένδρα Δυαδικής Αναζήτησης)







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

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

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

Ποια η σχέση των ταξινομημένων δυαδικών δένδρων και της ενδοδιατεταγμένης διάσχισης?

Ταξινομημένα Δένδρα

Αναδρομική Έκδοση

function *BinarySearchTreeLookUp*(**key** K, **pointer** R):**Type**

/* Εύρεση του κλειδιού K στο δένδρο με ρίζα P και επιστροφή του πεδίου data ή nill αν το κλειδί δεν υπάρχει στο δένδρο */

```
if (R == NULL) return nill;
else if (K == R->key) return R->data;
else if (K < R->key)
   return(BinaryTreeLookUp(K, R->LC));
else
   return(BinaryTreeLookUp(K, R->RC));
```

Χρονική πολυπλοκότητα;

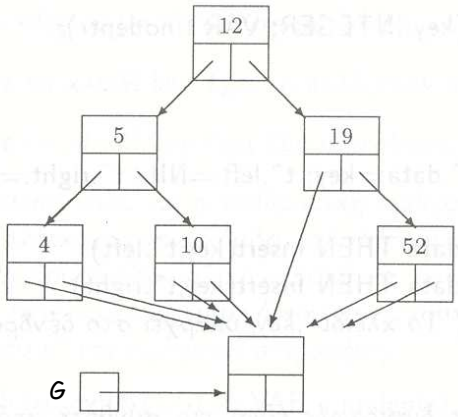
Μη-Αναδρομική Έκδοση

function *BinarySearchTreeLookUp* (**key** K, **pointer** R):**Type**

/* Εύρεση του κλειδιού K στο δένδρο με ρίζα P και επιστροφή του πεδίου data ή nill αν το κλειδί δεν υπάρχει στο δένδρο */

```
P = R;
while (P != NULL && K != P->key) {
   if (K < P->key) P = P->LC;
   else P = P->RC;
}
if (P != NULL)
      return(P->data);
else return nill;
}
```

Ταξινομημένα Δένδρα με Κόμβο Φρουρό




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

Ποια η χρησιμότητα του κόμβου φρουρού;

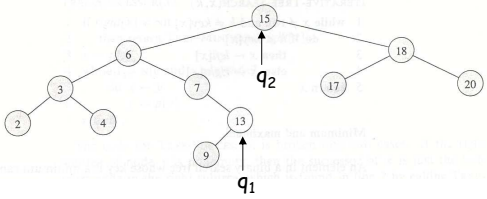
Ταξινομημένα Δένδρα - Ελάχιστο και Μέγιστο Στοιχείο

```
function BinarySearchTreeMinimum(
                    pointer R): info
/* ο R είναι δείκτης στη ρίζα του δένδρου */
   pointer P;

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

```
function BinarySearchTreeMaximum(
                    pointer R): info
/* ο R είναι δείκτης στη ρίζα του δένδρου */
   pointer P;

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

Στοιχείο με αμέσως μεγαλύτερο και αμέσως μικρότερο κλειδί από το κλειδί ενός κόμβου v

Το πρόβλημα είναι ίδιο με την εύρεση του επόμενου και προηγούμενου κόμβου του v στην ενδοδιατεταγμένη διάσχιση του δένδρου.



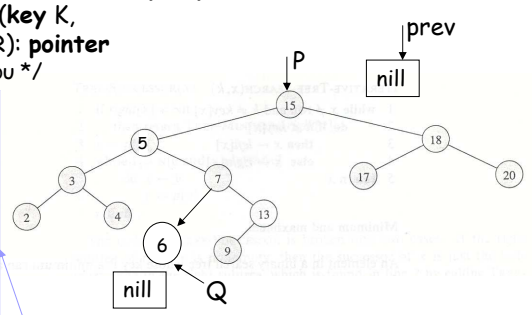
Πως θα βρούμε τον επόμενο του κόμβου στον οποίο δείχνει ο q_1, στην ενδοδιατεταγμένη διάσχιση;

Πως θα βρούμε τον επόμενο του κόμβου στον οποίο δείχνει ο q_2, στην ενδοδιατεταγμένη διάσχιση;

Πολυπλοκότητα;

Ταξινομημένα Δένδρα - Εισαγωγή

```
function BinarySearchTreeInsert(key K,
                    Type I, pointer R): pointer
/* ο R είναι δείκτης στη ρίζα του δένδρου */
   pointer P, Q, Prev = NULL;
   P = R;
   while (P != NULL) {
       if (P->Key == K) {
              P->data = I;
              return R;
       }
       Prev = P;
       if (K < P->Key) P = P->LC;
       else P = P->RC;
   }
   /* Δημιουργία & προσθήκη νέου κόμβου */
   Q = NewCell(Node);
   Q->Key = K; Q->data = I; Q->LC = Q->RC = NULL;
   if (Prev == NULL) return Q;
   else if (K < Prev->Key) Prev->LC = Q;
   else Prev->RC = Q;
   return R;
```



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

Παράδειγμα
Εισαγωγή κόμβου με κλειδί 6

Ταξινομημένα Δένδρα - Διαγραφή

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

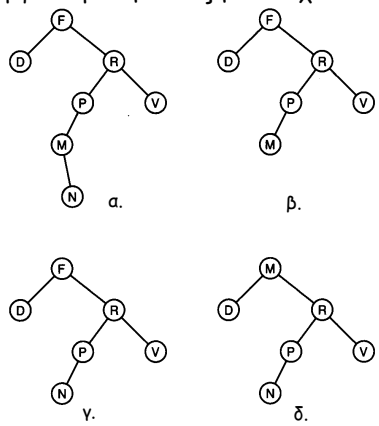
Περιπτώσεις

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

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

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

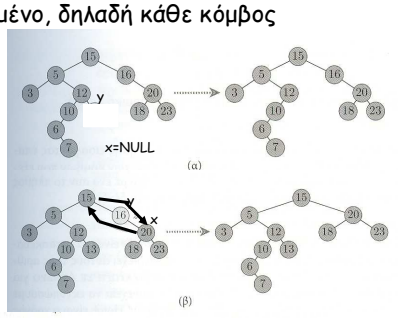
Αυτός είναι κόμβος με το πολύ ένα παιδί. Γιατί;



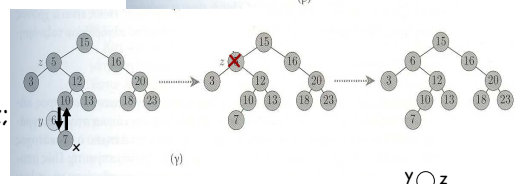
α. Αρχικό Δένδρο, β. Διαγραφή N από αρχικό δένδρο, γ. Διαγραφή M από αρχικό δένδρο, δ. Διαγραφή F από αρχικό δένδρο

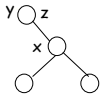
Ταξινομημένα Δένδρα - Διαγραφή

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




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

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

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

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

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





Ταξινομημένα Δένδρα

Θεώρημα

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

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

Στατικά Ταξινομημένα Δυαδικά Δένδρα

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

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

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