Coding Factory

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΑΛΓΟΡΙΘΜΟΙ

3.1 Δομές Δεδομένων

Δεδομένο (datum) ή **δομή δεδομένων (data structure)** είναι ένας ορισμένος τρόπος παράστασης μίας πληροφορίας. **Τύποι δεδομένων (data types)** είναι τα πεδία τιμών που μπορούν να λάβουν οι μεταβλητές μιας γλώσσας προγραμματισμού και εξακολουθητικά οι πράξεις που μπορούν να οριστούν πάνω στις μεταβλητές.

Οι τύποι δεδομένων διαχωρίζονται σε **πρωταρχικούς (primitive)** ή απλούς τύπους και σε **σύνθετους (complex)**. Πρωταρχικές δομές δεδομένων είναι οι ακέραιοι αριθμοί, οι δεκαδικοί, οι χαρακτήρες, οι τιμές αλήθειας (true, false), οι ημερομηνίες και γενικά οι δομές που δεν μπορούν να διασπαστούν σε απλούστερες μορφές.

Οι σύνθετες δομές δεδομένων αποτελούνται από πρωταρχικές δομές δεδομένων και είναι τριών τύπων:

α) **Γραμμικές δομές (linear structures)**, όπως πίνακες (arrays) , στοίβες (stacks), λίστες (lists), που περιλαμβάνουν ένα ή περισσότερα στοιχεία του ίδιου πρωταρχικού

τύπου καθώς και δομές (structs) που περιλαμβάνουν ένα ή περισσότερα στοιχεία διαφορετικού πρωταρχικού τύπου,

β) **δενδρικές δομές (trees)**, όπου υπάρχει η σχέση προγόνου-απογόνου στην ιεραρχία του δέντρου και

γ) **γραφήματα (graphs)**, όπου κάθε κόμβος του γραφήματος μπορεί να συνδέεται με οποιονδήποτε άλλο κόμβο.

3.2 Πίνακες (arrays)

Πίνακας (array) είναι μία διατεταγμένη ακολουθία τιμών του ίδιου τύπου δεδομένων. Η αναφορά στις θέσεις του πίνακα γίνεται με ακέραιους δείκτες, δηλαδή ακέραιες μεταβλητές που συμβολίζουν τη θέση του στοιχείου μέσα στην ακολουθία.

Το πλήθος των ακέραιων δεικτών ενός πίνακα ονομάζεται διάσταση (dimension) του πίνακα. Το πρώτο στοιχείο κάθε πίνακα μπορεί να ξεκινάει από τη θέση ένα ή από τη θέση μηδέν ανάλογα με την υλοποίηση κάθε γλώσσας προγραμματισμού. Η προσπέλαση ενός στοιχείου σε ένα πίνακα μπορεί να γίνει άμεσα με τη χρήση του ακέραιου δείκτη του.

Για παράδειγμα ένας πίνακας Α ακεραίων δέκα θέσεων μπορεί να οριστεί στη γλώσσα C ως εξής:

```
int A[10];
```

ενώ τα στοιχεία του καταλαμβάνουν τις θέσεις από `A[0]` έως `A[9]`.

Οι τιμές 0 έως 9 είναι τιμές δείκτη του πίνακα. Οι πίνακες μπορεί να είναι και περισσοτέρων από μίας διάστασης.

Για παράδειγμα η δήλωση:

```
int A[3][3];
```

ορίζει ένα δυσδιάστατο πίνακα ακεραίων 3x3.

3.3 Δομές (Structs)

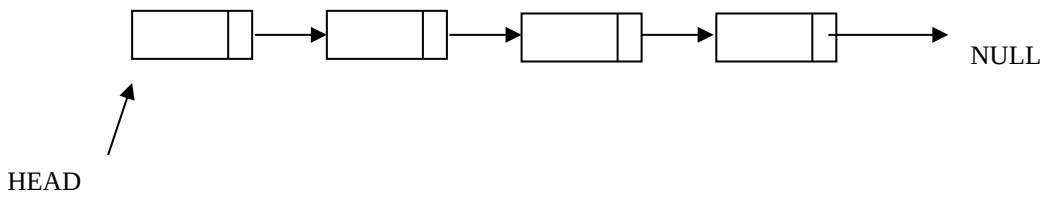
Οι **δομές (structs)** (ή εγγραφές) είναι δομές δεδομένων που ενθυλακώνουν (encapsulate) τιμές διαφορετικών τύπων δεδομένων. Για παράδειγμα, θα μπορούσε να χρησιμοποιηθεί ένας τύπος δομής για την υλοποίηση ενός τύπου δεδομένων διεύθυνσης ως εξής:

```
struct ADDRESS
{
     char street[50];     //   οδός
     int num;             //   αριθμός
     char city[20[]       //   πόλη
     char zipCode[5]      //   ταχ. κώδικας
};
```

3.4 Γραμμικές Λίστες

Γραμμική Λίστα (Linear List) είναι μια διατεταγμένη ακολουθία στοιχείων του ίδιου τύπου. Οι γραμμικές λίστες μπορεί να είναι στατικές ή δυναμικές ανάλογα με την δομή υλοποίησης.

Οι στατικές λίστες υλοποιούνται με πίνακες ενώ οι δυναμικές λίστες υλοποιούνται με δείκτες (pointers).



Σχήμα 3.1: Δυναμική Γραμμική Λίστα μιας διεύθυνσης

Κάθε κόμβος μιας λίστας μονής σύνδεσης ή μιας διεύθυνσης (one-way linked list) αποτελείται από τουλάχιστον ένα πεδίο που αποθηκεύει τα δεδομένα της λίστας και από ένα πεδίο δείκτη που δείχνει στο επόμενο κόμβο.

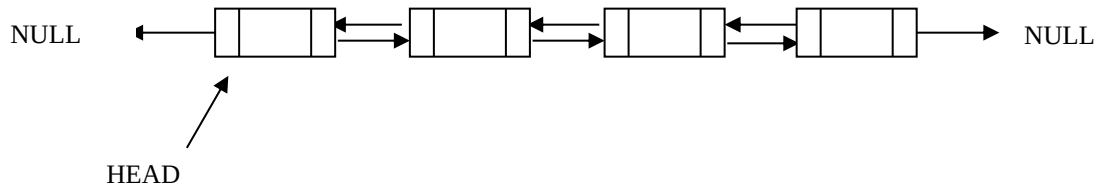
```
struct ListNode
{
   int data;               // Δεδομένα κόμβου (ακέραιοι)
   ListNode *next;         // Δείκτης στον επόμενο κόμβο
};
```

Ο δείκτης του τελευταίου κόμβου έχει την ειδική τιμή NULL (δείχνει σε null) που συμβολίζει το τέλος της λίστας, ενώ ένας ειδικός δείκτης έχει την διεύθυνση του πρώτου στοιχείου της λίστας και ονομάζεται HEAD (κεφαλή της λίστας).

Στις διπλά συνδεδεμένες λίστες ή λίστες δύο κατευθύνσεων (doubly linked lists) κάθε κόμβος αποτελείται από τουλάχιστον ένα πεδίο που αποθηκεύει τα δεδομένα της λίστας και δύο πεδία δείκτη, όπου το ένα πεδίο δείχνει στον επόμενο κόμβο και το άλλο στον προηγούμενο.

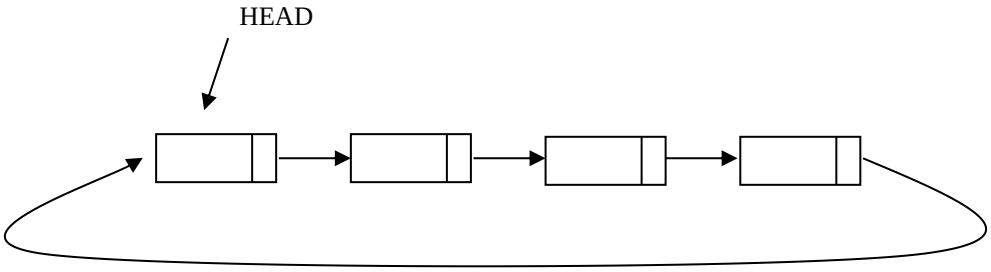
```
struct DListNode
{
   DListNode *pNext;       // Δείκτης στον επόμενο κόμβο
   DListNode *pPrev;       // Δείκτης στον προηγούμενο κόμβο
   int Data;               // Δεδομένα κόμβου (ακέραιοι)
};
```

Διαγραμματικά, μία διπλά συνδεδεμένη λίστα μπορεί να παρασταθεί ως εξής:



Σχήμα 3.2: Δυναμική Γραμμική Λίστα δύο διευθύνσεων

Στις κυκλικές γραμμικές λίστες το τελευταίο στοιχείο της λίστας δείχνει στην κεφαλή της λίστας:



Σχήμα 3.3: Κυκλική Γραμμική Λίστα

3.4.1 Στοίβα (Stack)

Η **Στοίβα (stack)** είναι μία λίστα (δυναμική ή στατική) με την ιδιότητα το τελευταίο στοιχείο που εισέρχεται στη στοίβα, εξέρχεται πρώτο (last In First Out – LIFO). Για να υλοποιηθεί μία στοίβα απαιτούνται:

– Μία στατική ή δυναμική γραμμική λίστα
– Ένας δείκτης που να δείχνει στην κορυφή (`top`) της στοίβας
– Ορισμός ενός συνόλου επιτρεπόμενων πράξεων που θα υλοποιούν τη δομή LIFO

```
#define MAXSTACK 50
int top;                  // Δείκτης στην κορυφή της στοίβας
char items[MAXSTACK];     // Στατική (πίνακας) Δομή στοίβας
```

Οι βασικές πράξεις μιας στοίβας είναι:

1. Δημιουργία στοίβας (create Stack)
2. Έλεγχος αν η στοίβα είναι άδεια (empty)
3. Έλεγχος αν η στοίβα είναι γεμάτη (full)
4. Εισαγωγή στοιχείου στη στοίβα (push)
5. Εξαγωγή στοιχείου από τη στοίβα (pop)

3.4.2 Ουρές (Queues)

Η ουρά (queue) είναι μία λίστα (δυναμική ή στατική) με την ιδιότητα το πρώτο στοιχείο που εισέρχεται στη ουρά, εξέρχεται και πρώτο (First In First Out – FIFO). Για να υλοποιηθεί μία ουρά απαιτούνται:

– Μία στατική ή δυναμική γραμμική λίστα
– Δύο δείκτες που να δείχνουν ο ένας στην αρχή της ουράς και ο άλλος στο τέλος της ουράς
– Ορισμός ενός συνόλου επιτρεπόμενων πράξεων που θα υλοποιούν τη δομή FIFO

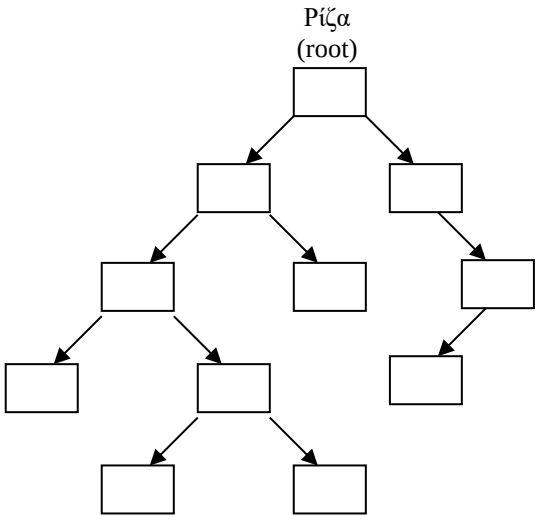
```
struct ListNode            /* Simply linked list */
{
   int data;
   struct ListNode* next;
};

struct queue {             // Τύπος Δεδομένων Ουρά (queue)
   ListNode* first;        // Δείκτης στην αρχή
   ListNode* last;         // Δείκτης στο τέλος
};
```

Οι βασικές πράξεις μιας ουράς είναι:

1. Δημιουργία κενής ουράς (create queue)
2. Έλεγχος αν η ουρά είναι άδεια (empty)
3. Έλεγχος αν η ουρά είναι γεμάτη (full), όταν γίνεται στατική υλοποίηση γιατί στην δυναμική υλοποίηση μπορεί θεωρητικά να δημιουργούνται νέοι κόμβοι μέχρι το τέλος της φυσικής μνήμης
4. Εισαγωγή στοιχείου στη ουρά (enqueue)
5. Εξαγωγή στοιχείου από τη ουρά (dequeue)

3.5 Δένδρα (trees)

Ένα δένδρο (tree) είναι μία ιεραρχική δομή δεδομένων, που εκφράζει μη γραμμικές σχέσεις και ορίζει σχέσεις προγόνων-απογόνων. Πιο συγκεκριμένα, δένδρο T είναι ένα πεπερασμένο σύνολο κόμβων που αποτελείται από ένα ειδικό κόμβο ρίζα (root) που βρίσκεται στην υψηλότερη θέση της ιεραρχίας και από `n` ξένα μεταξύ τους σύνολα κόμβων (T_i, `i=1..n`) που καθένα είναι ένα δένδρο. Τα δένδρα T_i ονομάζονται υπόδενδρα της ρίζας.



Σχήμα 3.4: Δενδρική δομή

Βαθμός κόμβου (node degree) ή **παράγοντας διακλάδωσης (branching factor)** είναι ο αριθμός των υποδένδρων που ξεκινούν από ένα κόμβο ή αλλιώς το πλήθος των απογόνων ενός κόμβου.

Βαθμός δένδρου (tree degree) είναι ο μέγιστος βαθμός των κόμβων του, ενώ ένα δένδρο ονομάζεται **πλήρες (complete)** όταν όλοι οι κόμβοι του εκτός από τα φύλλα έχουν ως βαθμό το μέγιστο βαθμό του δένδρου. Ένα δένδρο ονομάζεται **βαθμού b**, όταν ξεκινούν το πολύ b υποδένδρα από κάθε κόμβο του.

Φύλλα (leaves) ενός δένδρου είναι οι τελικοί του κόμβοι, από όπου δεν ξεκινάει άλλο υποδένδρο, ενώ εσωτερικοί ή ενδιάμεσοι κόμβοι ονομάζονται όλοι οι υπόλοιποι κόμβοι εκτός από τα φύλλα.

Δυαδικό δένδρο (binary tree) ονομάζεται ένα δένδρο βαθμού δύο, ενώ δυαδικό δένδρο αναζήτησης είναι ένα δυαδικό δένδρο όπου όλοι οι κόμβοι του αριστερού υποδένδρου κάθε κόμβου έχουν τιμές μικρότερες από τον κόμβο και όλοι οι κόμβοι του δεξιού υποδένδρου έχουν τιμές μεγαλύτερες από τον κόμβο.

Οι σύνδεσμοι (links) ανάμεσα στους κόμβους του δένδρου ονομάζονται **κλαδιά (branches)** ή **ακμές (edges)**. Μήκος μονοπατιού ενός κόμβου ονομάζεται το πλήθος των κλαδιών από τη ρίζα μέχρι τον κόμβο.

Επίπεδο (level) σε μία δενδρική δομή ονομάζεται το βάθος στην ιεραρχία των κόμβων. Η **ρίζα (root)** βρίσκεται στο 1ο επίπεδο ιεραρχίας, οι απόγονοι της ρίζας στο 2ο επίπεδο, ενώ τα φύλλα στο τελευταίο επίπεδο.

΄Υψος (height) ή βάθος (depth) του δένδρου ονομάζεται το πλήθος των επιπέδων του.

Κάθε πλήρες δένδρο βαθμού b και ύψους h, έχει $(b^h - 1)/(b-1)$ κόμβους.

Επομένως ένα πλήρες δυαδικό δένδρο έχει n = $(2^h - 1)$ κόμβους και συνεπώς αν λογαριθμίσουμε και τους δύο όρους υπολογίζουμε ότι το ύψος ενός πλήρους δυαδικού δένδρου είναι h= $\log_2 n$.

Οι βασικές πράξεις σε μία δενδρική δομή είναι οι :

– Δημιουργία του δένδρου
– Εισαγωγή νέου κόμβου
– Διαγραφή κόμβου
– Διάσχιση του δένδρου

Η διάσχιση ενός δένδρου είναι η επίσκεψη (και εκτύπωση) κάθε κόμβου του δένδρου με μια συγκεκριμένη σειρά επίσκεψης.

Ανάλογα με τη σειρά επίσκεψης των κόμβων, η διάσχιση ενός δένδρου μπορεί να είναι ενδοδιατεταγμένη (inorder), προδιατεταγμένη (preorder) και μεταδιατεταγμένη (postorder).

Κατά την **ενδοδιατεταγμένη διάσχιση** η σειρά επίσκεψης των κόμβων είναι:

1. Αριστερό υποδένδρο
2. Ρίζα
3. Δεξιό Υποδένδρο

Μία αναδρομική συνάρτηση `inorder` στη γλώσσα C δίνεται παρακάτω:

```
void inorder(treeNode n)
{
      if (n!=NULL)
      {
            inorder(t→left);
            printf("%d", t→data);
            inorder(t→right);
      }
}
```

Κατά την **προδιατεταγμένη διάσχιση** η σειρά επίσκεψης των κόμβων είναι:

1. Ρίζα
2. Αριστερό υποδένδρο
3. Δεξιό Υποδένδρο

Μία αναδρομική συνάρτηση preorder στη γλώσσα C δίνεται παρακάτω:

```
void preorder(treeNode n)
{
      if (n!=NULL)
      {
            printf("%d", t→data);
            inorder(t→left);
            inorder(t→right);
      }
}
```

Κατά την **μεταδιατεταγμένη διάσχιση** η σειρά επίσκεψης των κόμβων είναι:

1. Αριστερό υποδένδρο
2. Δεξιό Υποδένδρο
3. Ρίζα

Μία αναδρομική συνάρτηση postorder στη γλώσσα C δίνεται παρακάτω:

```
void postorder(treeNode n)
{
      if (n!=NULL)
      {
            inorder(t→left);
            inorder(t→right);
            printf("%d", t→data);
      }
}
```

3.5.1 Δυαδικά Δένδρα Αναζήτησης

Τα **δυαδικά δένδρα αναζήτησης (binary search trees)**, όπως αναφέραμε ήδη, είναι δυαδικά δένδρα όπου όλοι οι κόμβοι του αριστερού υποδένδρου κάθε κόμβου έχουν τιμές μικρότερες από τον κόμβο και όλοι οι κόμβοι του δεξιού υποδένδρου έχουν τιμές μεγαλύτερες από τον κόμβο.

Η κύρια πράξη σε ένα δυαδικό δένδρο αναζήτησης είναι η αναζήτηση και εύρεση ενός κόμβου που περιέχει ένα συγκεκριμένο στοιχείο.

Μία επαναληπτική συνάρτηση αναζήτησης κόμβων που περιέχουν ακεραίους σε γλώσσα C δίνεται παρακάτω:

```
treeNode find (TreeNode n, int data)
{
      int found=0;
      while (n != NULL) and (!found)
      {
            if (n→data = data)
                  found=1
            else if (n→data < data)
                  n=n→right
            else n=n→left
      }
      return n;
}
```

Σε ένα πλήρες δυαδικό δένδρο αναζήτησης η πράξη της αναζήτησης έχει πολυπλοκότητα χρόνου $\log_n$, σε αντίθεση με τις γραμμικές λίστες όπου η αναζήτηση έχει πολυπλοκότητα χρόνου γραμμική O(n).

Για την εισαγωγή ενός στοιχείου σε δένδρο αναζήτησης απαιτείται πρώτα να βρεθεί η θέση που θα εισαχθεί ο νέος κόμβος. Μία αναδρομική συνάρτηση εισαγωγής σε γλώσσα C δίνεται παρακάτω:

```
void insert (treeNode n, int data)
{
      if (n=NULL)
      {
            n=new(treeNode); // Δημιουργεί νέο κόμβο
            n→data = data;
            n→left=NULL;
            n→right=NULL;
      }
      else if (data<n→data) insert (n→left, data);
      else if (data>n→data) insert (n→right, data);
      else printf("Το στοιχείο υπάρχει»");
}
```

Η πολυπλοκότητα χρόνου της εισαγωγής είναι επίσης $\log_n$, αφού χρειάζεται πρώτα η εύρεση της θέσης που θα εισαχθεί το νέο στοιχείο.

3.6 Ισοζυγισμένα Δένδρα

Ένα δένδρο ονομάζεται d-ισοζυγισμένο (d-balanced) όταν το αριστερό και δεξί υποδένδρο κάθε κόμβου του διαφέρουν το πολύ κατά d επίπεδα. Τέλεια ισοζυγισμένο (perfectly balanced) ή 1-ισοζυγισμένο (balanced 1-d) ονομάζεται όταν το αριστερό και δεξί υποδένδρο κάθε κόμβου του διαφέρουν το πολύ κατά ένα επίπεδο.

Ένα 1-ισοζυγισμένο δένδρο που είναι ταυτόχρονα και δένδρο αναζήτησης ονομάζεται AVL δένδρο.

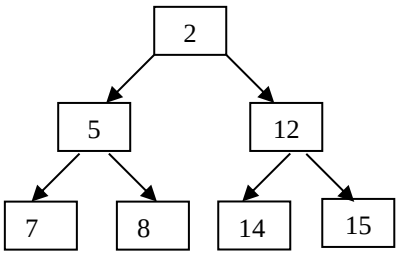
Ο λόγος που τα AVL δένδρα είναι χρήσιμα είναι ότι τα δυαδικά δένδρα αναζήτησης για να έχουν πολυπλοκότητα χρόνου αναζήτησης $\log_n$ θα πρέπει να παραμένουν πλήρη, αλλιώς αν εκφυλιστούν σε λίστες, τότε ο χρόνος αναζήτησης γίνεται γραμμικός.

Σε ένα AVL δένδρο οι πράξεις της εισαγωγής και διαγραφής κόμβου διατηρούν το δένδρο αναζήτησης τέλεια ισοζυγισμένο και επομένως η πράξη της αναζήτησης παραμένει $\log_n$.

3.7 Σωροί

Η **σωρός (heap)** είναι μια ακολουθία στοιχείων a(i) για τα οποία ισχύει: a(i)<=α(2i) και a(i)<=α(2i+1).

Η σωρός μπορεί να παρασταθεί με ένα πλήρες δυαδικό δένδρο, όπου το περιεχόμενο κάθε κόμβου είναι μικρότερο από το περιεχόμενο των παιδιών του.



Σχήμα 3.5: Σωρός

Για την κατασκευή μιας σωρού θα πρέπει όλοι οι κόμβοι ενός πλήρους δυαδικού δένδρου να πληρούν τη συνθήκη της σωρού. Επομένως, θα πρέπει να αναπροσαρμοστούν όλοι οι κόμβοι που δεν είναι φύλλα, γιατί τα φύλλα δεν έχουν παιδιά και επομένως πληρούν τη συνθήκη της σωρού.

Το πλήθος των εσωτερικών κόμβων ενός πλήρους δυαδικού και ισοζυγισμένου δένδρου αναζήτησης είναι n/2, ενώ για κάθε εσωτερικό κόμβο απαιτείται πολυπλοκότητα χρόνου αναπροσαρμογής το πολύ $\log_n$, όσο δηλαδή και η δυαδική αναζήτηση. Άρα τελικά η πολυπλοκότητα χρόνου αναπροσαρμογής μιας σωρού είναι $n\log_n$. Η σωρός χρησιμοποιείται για την υλοποίηση ουρών προτεραιότητας γιατί έχει καλύτερο χρόνο εισαγωγής και διαγραφής στοιχείων ($\log_n$) από ότι η υλοποίηση της ουράς με λίστες (γραμμική πολυπλοκότητα).

3.8 Μ-κατευθυνόμενα δένδρα αναζήτησης

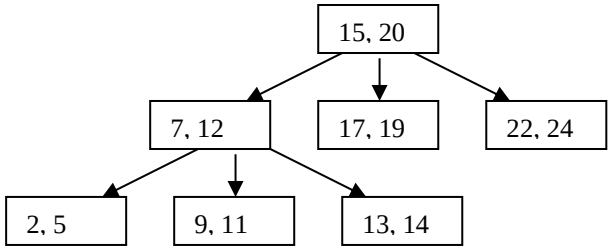
Ένα δένδρο ονομάζεται **Μ-κατευθυνόμενο δένδρο αναζήτησης (M-directed search tree)** όταν πληρεί τις παρακάτω ιδιότητες:

– όλοι οι κόμβοι του, N_i , $0<=I<=M$, έχουν βαθμό μικρότερο ή ίσο με Μ

– κάθε κόμβος του δένδρου έχει το πολύ $M-1$ κλειδιά, K_i , $0<=i<M$, με τιμές $K_i < K_{i+1}$

– κάθε υπόδενδρο T_i του κόμβου-προγόνου N_i έχει κλειδιά με τιμές μικρότερες από το κλειδί K_i του N_i , ενώ κάθε υπόδενδρο T_{i+1} έχει κλειδιά μεγαλύτερα από το κλειδί K_i του N_i

– κάθε υποδένδρο T_i είναι Μ-κατευθυνόμενο δένδρο αναζήτησης



Σχήμα 3.6: 3-κατευθυνόμενο δένδρο αναζήτησης

3.9 Β-δένδρα

Κόμβος αποτυχίας είναι ένας ειδικός κόμβος, S_i που δεν έχει απογόνους και θεωρούμε ότι φτάνουμε σε αυτόν κατά τη διαδικασία αναζήτησης ενός στοιχείου που δεν υπάρχει.

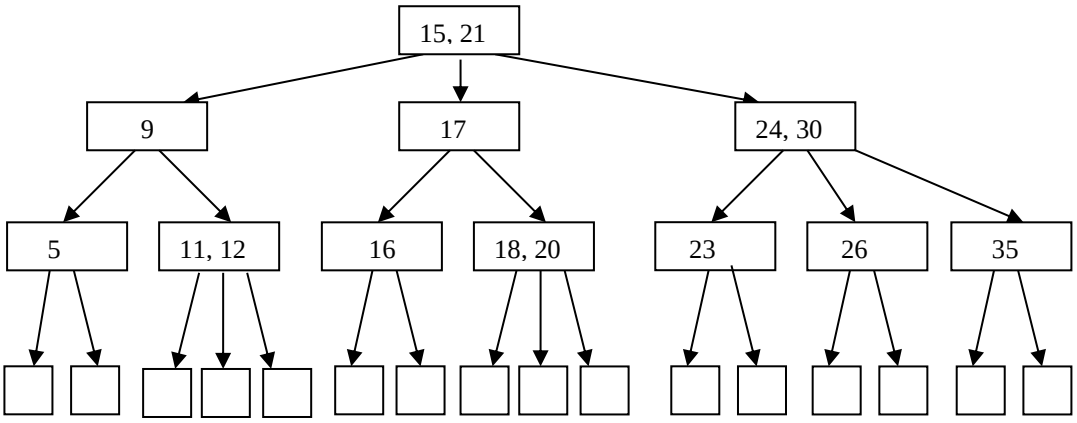
Στην υλοποίηση ενός δένδρου δεν υπάρχουν πραγματικοί κόμβοι αποτυχίας, αλλά υπονοούνται θέτοντας την τιμή NULL στον σύνδεσμο που δείχνει σε κόμβο αποτυχίας.

Ένα Β-δένδρο είναι ένα Μ-κατευθυνόμενο δένδρο αναζήτησης με τις παρακάτω ιδιότητες:

– Η ρίζα έχει τουλάχιστον δύο απογόνους

– Όλοι οι κόμβοι (εκτός από τη ρίζα) έχουν τουλάχιστον M/2 απογόνους

– Όλοι οι κόμβοι αποτυχίας βρίσκονται στο ίδιο επίπεδο



Σχήμα 3.7 : B-δένδρο

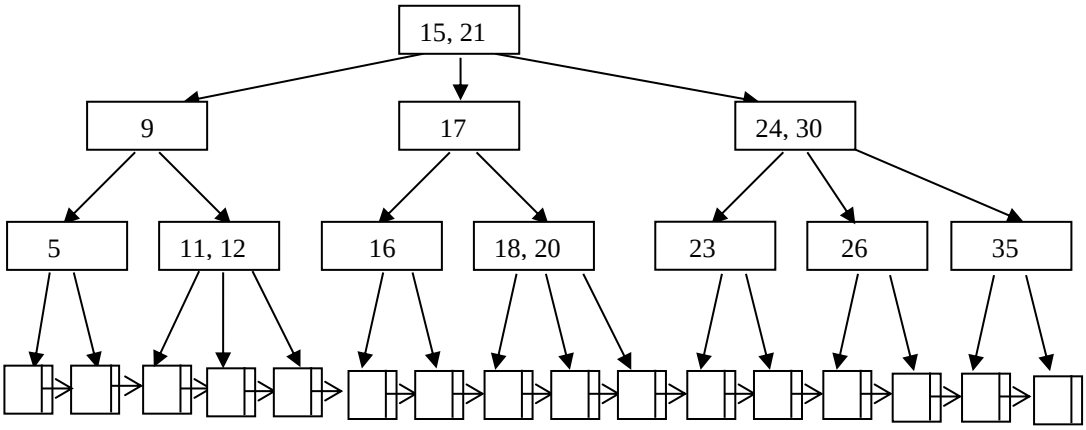
Το πλήθος των κλειδιών κάθε κόμβου ενός B-δένδρου είναι κατά ένα μικρότερος από το πλήθος των απογόνων του κόμβου. Το ύψος, h, ενός B-δένδρου με N κλειδιά είναι: $h \le \log_{M/2}\left(\frac{N+1}{2}+1\right)$ και επομένως η πολυπλοκότητα αναζήτησης ενός κόμβου είναι $\log_{M/2} N$

Αν το B-δένδρο είναι πλήρες τότε το ύψος του είναι $h \le \log_{M}\left(\frac{N+1}{2}+1\right)$ και επομένως η πολυπλοκότητα αναζήτησης ενός κόμβου είναι $\log_{M} N$

3.10 B^+ Δένδρα

Η διάσχιση σε ένα B–δένδρο απαιτεί τη χρήση στοίβας λόγω αναδρομής. Μία πιο γρήγορη και αποτελεσματική δομή B-δένδρων θα ήταν αν αντί για τους κόμβους αποτυχίας υπήρχαν κανονικοί κόμβοι (φύλλα) που αποθήκευαν όλα τα στοιχεία μιας δομής δεδομένων (π.χ. εγγραφής) ενώ οι ενδιάμεσοι κόμβοι αποθήκευαν μόνο τα κλειδιά των εγγραφών.

Αν τα φύλλα συνδεθούν σε μια γραμμική λίστα τότε μπορεί ένα σύνολο εγγραφών να προσπελαστεί ακολουθιακά σε γραμμικό χρόνο.



Σχήμα 3.8: B^+-δένδρο

Επίσης, επειδή οι ενδιάμεσοι κόμβοι περιέχουν μόνο τα κλειδιά μπορούν να αποθηκευτούν περισσότερα στοιχεία μειώνοντας το ύψος του δένδρου και τον χρόνο αναζήτησης.

3.11 Γραφήματα

Σε δομές δεδομένων όπως η λίστα υπάρχει σχέση προτεραιότητας (προηγούμενος-επόμενος) και στις δενδρικές δομές η σχέση ιεραρχίας (πατέρας-παιδί). Τα γραφήματα

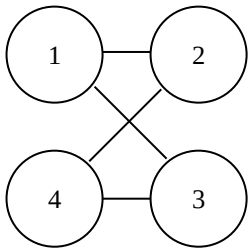
είναι γενικευμένες δομές όπου κάθε στοιχείο-κόμβος μπορεί να συνδέεται με οποιοδήποτε άλλο στοιχείο.

Πιο συγκεκριμένα, γράφημα είναι μια δομή που αποτελείται από δύο σύνολα:

– ένα πεπερασμένο σύνολο, V, των κόμβων ή κορυφών (Vertices) του γραφήματος
– και το σύνολο Ε των ακμών (Edges) του γραφήματος με στοιχεία όλα τα διατεταγμένα ζεύγη (α,β) όπου α, β οι κόμβοι που συνδέονται μεταξύ τους.

Οι κόμβοι α, β που συνδέονται με την ακμή (α,β) ονομάζονται γειτονικοί ενώ η ακμή ονομάζεται προσκείμενη. Δύο ακμές που πρόσκεινται στον ίδιο κόμβο ονομάζονται γειτονικές, ενώ ο βαθμός του κόμβου είναι το πλήθος των ακμών που πρόσκεινται στον κόμβο αυτό.

Έστω ένα γράφημα G με σύνολο κορυφών V(G)={1, 2, 3, 4} και σύνολο ακμών E(G) = {(1,2), (1,3), (2,3), (2,4), (3,4)}. Το παραπάνω γράφημα μπορεί να παρασταθεί διαγραμματικά ως εξής:



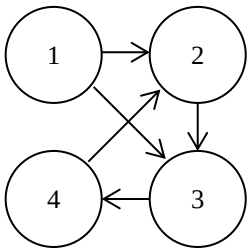
Σχήμα 3.9: Μη κατευθυνόμενο γράφημα

Μονοπάτι είναι μία διατεταγμένη ακολουθία κόμβων από τον ένα κόμβο σε ένα άλλο. Για παράδειγμα η ακολουθία (1, 2. 4, 3) είναι ένα μονοπάτι από τον κόμβο 1 προς τον κόμβο 3. Το μονοπάτι (1, 2, 4, 3, 1) που έχει τον ίδιο αρχικό και τελικό κόμβο ονομάζεται **κύκλος**, ενώ ένα γράφημα δίχως κύκλους ονομάζεται **ακυκλικό (acyclic)**.

Υπογράφημα ενός γραφήματος G ονομάζεται ένα γράφημα του οποίου όλοι οι κόμβοι και οι ακμές ανήκουν στο G. Ένα γράφημα ονομάζεται **συνεκτικό (connected)** αν υπάρχει μονοπάτι για κάθε ζεύγος κόμβων. **Συνεκτική συνιστώσα (connected**

component) ονομάζεται ένα γράφημα που είναι μέγιστο συνεκτικό υπογράφημα. Ένα γράφημα μπορεί να είναι δέντρο όταν δεν έχει κύκλους και είναι συνεκτικό.

Γράφημα με κατεύθυνση (directed graph) είναι ένα γράφημα του οποίου όλες οι ακμές είναι διατεταγμένα ζεύγη από κόμβους. Σχηματικά κάθε ακμή παριστάνεται με βέλος.



Σχήμα 3.10: Κατευθυνόμενο γράφημα

Για παράδειγμα στο παραπάνω γράφημα όλες οι ακμές έχουν κατεύθυνση. Η ακμή (1, 2) ξεκινά από τον κόμβο 1 και προσπίπτει στον κόμβο 2. **Προς τα έσω βαθμός (in-degree)** ενός κόμβου είναι το πλήθος των ακμών που προσπίπτουν στον κόμβο, ενώ **προς τα έξω βαθμός (out-degree)** είναι το πλήθος των ακμών που ξεκινούν από τον κόμβο.

Πηγή (source) ονομάζεται ένας κόμβος με στον οποίο δεν προσπίπτει καμία ακμή (in-degree = 0) και **καταβόθρα (sink)** ένας κόμβος από τον οποίο δεν ξεκινά καμία ακμή (out-degree = 0).

Γράφημα με βάρη (weighted graph) είναι ένα γράφημα με ακμές που φέρουν βάρη. Βάρος ακμής είναι ένας ακέραιος αριθμός που μπορεί να συμβολίζει ένα φυσικό μέγεθος, όπως το κόστος διάσχισης της ακμής, την απόσταση ενός κόμβου από έναν άλλο, την χωρητικότητα μιας σύνδεσης, κλπ.

Ένα γράφημα μπορούμε να το παραστήσουμε με μία από τις παρακάτω δομές δεδομένων:

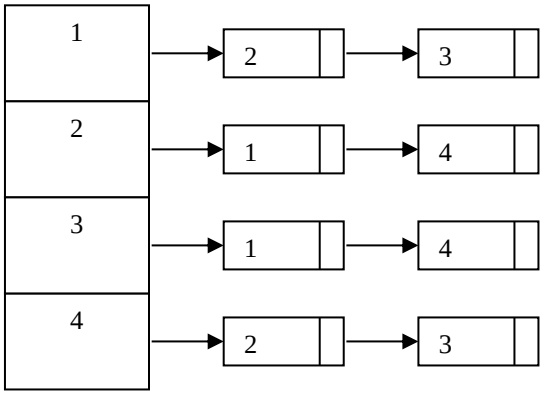
– Μήτρα γειτνίασης (adjacency matrix)
– Λίστα γειτνίασης (adjacency list)

Η δομή δεδομένων που χρησιμοποιείται σε μια **μήτρα γειτνίασης** είναι ένας πίνακας `nxn` όπου n είναι το πλήθος των κόμβων του γραφήματος. Κάθε στοιχείο `a(i,j)` του πίνακα γίνεται ένα αν υπάρχει ακμή ανάμεσα στους κόμβους `i,j` και μηδέν αν δεν υπάρχει. Για παράδειγμα, το γράφημα 3.8 μπορεί να παρασταθεί από τον παρακάτω πίνακα γειτνίασης.

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Σχήμα 3.11: Μήτρα Γειτνίασης

Η **λίστα γειτνίασης** υλοποιείται με ένα μονοδιάστατο πίνακα κάθε στοιχείο του οποίου αντιστοιχεί σε ένα κόμβο του γραφήματος. Κάθε στοιχείο του πίνακα είναι μία γραμμική λίστα που αποτελείται από όλους τους γειτονικούς κόμβους κάθε κόμβου-στοιχείου του πίνακα.



Σχήμα 3.12: Λίστα Γειτνίασης

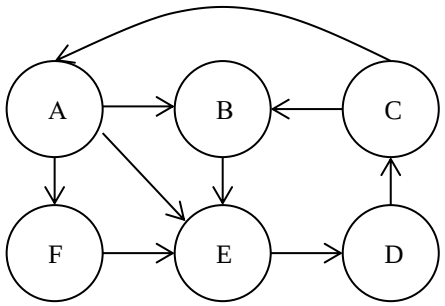
Τα γραφήματα με βάρη μπορούν να παρασταθούν, αν αντί για 0 και 1 στον πίνακα γειτνίασης τοποθετούμε τα βάρη, ενώ στην λίστα γειτνίασης μπορούμε να συμπεριλαμβάνουμε και το βάρος της ακμής σε κάθε κόμβο της λίστας.

3.11.1 Αλγόριθμοι διάσχισης DFS-BFS

Δύο βασικοί αλγόριθμοι διάσχισης όλων των κόμβων και ακμών ενός γραφήματος είναι οι:

- DFS (Depth First Search), που διασχίσει ένα γράφημα κατά βάθος
- BFS (Breadth First Search), που διασχίζει ένα γράφημα κατά πλάτος

Έστω το παρακάτω κατευθυνόμενο γράφημα:



Σχήμα 3.13: Κατευθυνόμενο Γράφημα

Μία απλή εκδοχή του αλγόριθμου DFS είναι η εξής:

```
dfs (graph  G(V, E))
{
      Για κάθε κόμβο v do
      {
            visited(v)=false; //αρχικοποίηση     (1)
      }

      Για κάθε κόμβο v do
      {
            if not visited(v) then explore(v);   (2)
      }
}

explore (vertex v)
{
      visited(v)=true;                           (3)
      Για κάθε γειτονικό κόμβο u του v do
      {
            // για κάθε ακμή (v,u)
            if not visited(u) then explore(u);   (4)

      }
}
```

Στο βήμα (1) του DFS όλοι οι κόμβοι μαρκάρονται/αρχικοποιούνται ως `false`, που σημαίνει ότι δεν τους έχει επισκεφθεί η DFS.

Στο βήμα (2) για κάθε κόμβο, ξεκινώντας από την πηγή, αρχίζει η εξερεύνηση DFS του κόμβου.

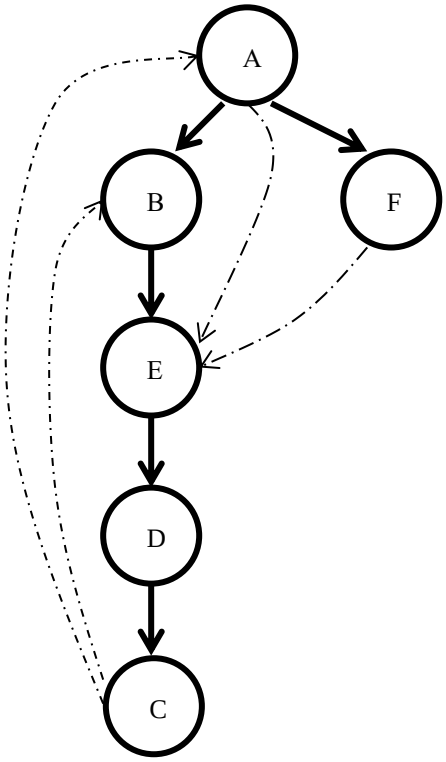
Η εξερεύνηση με τη μέθοδο του DFS όπως φαίνεται και στο βήμα (4) είναι αναδρομική. Πρώτα μαρκάρει τον κάθε κόμβο που επισκέπτεται (3) και στη συνέχεια εξερευνά όλες τις ακμές του κάθε κόμβου αναδρομικά.

Την μέθοδο αναζήτησης κατά βάθος (DFS) μπορούμε να την χρησιμοποιήσουμε και για να χαρακτηρίσουμε τις ακμές του γραφήματος ως ακμή εμπρός, ακμή πίσω ή ακμή πάνω, προσθέτοντας απλά μεταβλητές μετρητών, που αυξάνονται πριν και μετά την αναδρομική κλήση της explore().

Επίσης με την DFS, μπορούμε να βρούμε τις συνεκτικές συνιστώσες του γραφήματος αν προσθέσουμε στον αλγόριθμο μια μεταβλητή που να αυξάνεται κατά ένα στο βήμα (3).

Η εκτέλεση του αλγορίθμου DFS παράγει ένα επικαλυπτικό δέντρο DFS (DFS spanning tree) δηλαδή ένα δένδρο με τη σειρά διάσχισης των κόμβων του γραφήματος ή ένα δάσος DFS αν πρόκειται για περισσότερα από ένα δένδρα.

Για παράδειγμα η εκτέλεση του DFS στο γράφημα του σχήματος 3.13 παράγει το παρακάτω DFS δένδρο:



Σχήμα 3.14: Δένδρο DFS

Στην εξερεύνηση κατά πλάτος (BFS) πρώτα επισκεπτόμαστε τους γειτονικούς κόμβους ενός κόμβου v και στη συνέχεια τους γειτονικούς των γειτονικών, κλπ.

```
BFS (node n, graph G)
{
      // Χρησιμοποιούμε μία δομή ουράς (FIFO Queue)
      //Εισήγαγε τον κόμβο n στην ουρά
      insertQueue(n);                                    (1)

      // Η απόσταση του κόμβου από τη ρίζα
      d[n]=0;                                            (2)

      Όσο η ουρά έχει κόμβους do                         (3)
      {
            Αφαίρεσε από την ουρά τον πρώτο κόμβο v;     (4)

            visited(v)=true;                             (5)

            Για κάθε γειτονικό κόμβο u του v do          (6)
                  if not visited(u)
                  {
                        d[u]=d[v]+1;                     (7)
                        insertQueue(u);                  (8)
                  }
      }
}
```

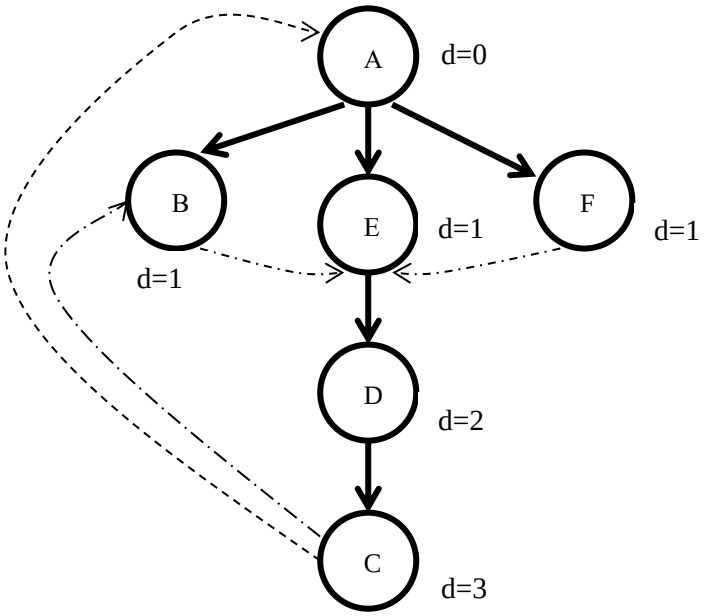
Στον BFS εισάγουμε πρώτα στην ουρά τον κόμβο-πηγή-ρίζα (βήμα(1)) ενώ στη συνέχεια αρχικοποιούμε τη απόσταση από τη ρίζα, που για την ίδια την ρίζα είναι μηδέν (βήμα (2)).

Στο βήμα (3) εκτελείται μία επανάληψη' όσο η ουρά έχει στοιχεία, αφαιρείται το πρώτα στοιχείο της ουράς (βήμα (4)), σημαδεύεται ο κόμβος ως visited (βήμα (5)) και στη

συνέχεια κάθε γειτονικός κόμβος (βήμα (6)) εισάγεται στην ουρά και αυξάνεται η απόστασή του από τη ρίζα κατά ένα (βήμα (7) και (8)).

Η μεταβλητή απόστασης d[v] είναι η ελάχιστη απόσταση (ελάχιστο μονοπάτι) από τη ρίζα.

Η εκτέλεση του αλγόριθμου BFS στο γράφημα του σχήματος 3.13 παράγει το παρακάτω BFS δένδρο:



Σχήμα 3.15: Δένδρο BFS

Στο δένδρο BFS οι αποστάσεις των κόμβων από τη ρίζα είναι ελάχιστες και τα μονοπάτια μέχρι τους κόμβους είναι τα ελάχιστα μονοπάτια.

3.12 Αλγόριθμοι και Πολυπλοκότητα

Αλγόριθμος είναι ένα πεπερασμένο σύνολο από καλά ορισμένα βήματα για την πραγματοποίηση μιας λειτουργίας ή τη λύση ενός προβλήματος, τα οποία -δοσμένης μιας αρχικής κατάστασης- θα τερματίζουν σε μια ορισμένη τελική κατάσταση.

Πιο απλά, αλγόριθμος είναι μια πεπερασμένη ακολουθία πράξεων (βημάτων) για την επίλυση ενός προβλήματος.

Η ιδέα του αλγορίθμου σχετίζεται ιδιαίτερα με μαθηματικά προβλήματα, όπως η εύρεση του μέγιστου κοινού διαιρέτη δύο αριθμών, η την εύρεση του μικρότερου στοιχείου μιας αταξινόμητης ή ταξινομημένης ακολουθίας χαρακτήρων, κλπ..

Οι περισσότεροι αλγόριθμοι μπορούν να υλοποιηθούν με τον προγραμματισμό υπολογιστών, ενώ κάποιοι άλλοι μπορούν απλά να προσομοιωθούν.

Κάθε αλγόριθμος πρέπει να έχει τα παρακάτω χαρακτηριστικά:

– Να είναι αυστηρά ορισμένος, τυπικά σε μια μαθηματική γλώσσα ή γλώσσα προγραμματισμού,

– να έχει κανένα, ένα ή περισσότερα δεδομένα εισόδου,

– να παράγει αποτελέσματα (δεδομένα εξόδου) και

– να είναι ορθός και όσο το δυνατό πιο αποδοτικός σε όρους πολυπλοκότητας χρόνου.

3.13 Πολυπλοκότητα

Από όλους τους αλγόριθμους που επιλύουν ένα πρόβλημα, ενδιαφέρουν οι “καλοί“ αλγόριθμοι, δηλαδή οι πιο αποδοτικοί σε όρους χρόνου και χώρου.

Ιδιαίτερα η πολυπλοκότητα χρόνου (time complexity) αποτελεί το βασικότερο κριτήριο αξιολόγησης των αλγορίθμων.

Ένας αλγόριθμος είναι καλύτερος από κάποιον άλλο αν η μέση συμπεριφορά του σε όρους χρόνου είναι πιο αποδοτική, δηλαδή έχει μικρότερο χρόνο εκτέλεσης.

Λέγοντας πολυπλοκότητα χρόνου (time complexity) δεν εννοείται ο απόλυτος χρόνος σε λεπτά, δευτερόλεπτα κλπ., γιατί τότε η συμπεριφορά του αλγόριθμου θα εξαρτιόταν από τις επιδόσεις του υπολογιστή που τον εκτελεί. Άλλωστε, στην αξιολόγηση των αλγορίθμων δεν ενδιαφέρει ο απόλυτος χρόνος εκτέλεσης, γιατί η έννοια του "καλύτερου" είναι σχετική.

Πολυπλοκότητα χρόνου ενός αλγορίθμου είναι το πλήθος των βασικών πράξεων σε σχέση με τα δεδομένα εισόδου.

Για παράδειγμα, έστω η συνάρτηση παραγοντικό:

$$f(n)=\begin{cases} n!, & n \geq 1 \\ 1, & n = 0 \end{cases}$$

και ένας επαναληπτικός αλγόριθμος για τον υπολογισμό της:

```
int factorial (int n)
{
      int f=1;
      for (int i=1; i<=n; i++) f=f*i;
      return f;
}
```

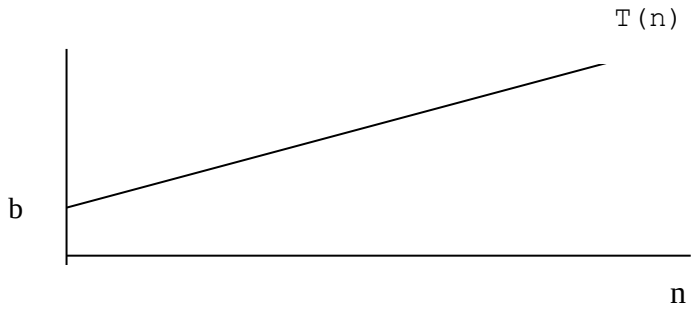
Χρόνος	Επαναλήψεις
t1	1
t2	n

Όπως παρατηρούμε για την εκτέλεση του παραπάνω αλγορίθμου απαιτούνται μία εκχώρηση με χρόνο `t1` και `n` επαναλήψεις –κάθε μία με χρόνο `t2`. Άρα η πολυπλοκότητα χρόνου είναι:

```
T(n) = t1 + n*t2
```

και γενικότερα αν αντικαταστήσουμε τα `t1, t2` με τις μεταβλητές `b και a` αντίστοιχα που είναι ανεξάρτητες από τον απόλυτο χρόνο, τότε

`T(n) = an + b`, δηλαδή η πολυπλοκότητα χρόνο είναι γραμμική ως προς το n (δεδομένα εισόδου).



Σχήμα: 3.16: Γραμμική πολυπλοκότητα χρόνου

Για μεγάλες τιμές του n, η μεταβλητή `b` δεν επηρεάζει την χρονική πολυπλοκότητα και επομένως η συμπεριφορά του αλγορίθμου βασικά προσδιορίζεται από τον όρο `an`.

Επίσης, όπως έχει ειπωθεί, αυτό που ενδιαφέρει δεν είναι ο απόλυτος χρόνος εκτέλεσης του αλγορίθμου αλλά το πως μεταβάλλεται η πολυπλοκότητα, όταν μεταβάλλονται τα δεδομένα εισόδου. Αν για παράδειγμα έχουμε `2n` δεδομένα εισόδου, τότε ο βασικός όρος γίνεται `a(2n)`, ενώ η σχέση $\frac{T(n)}{T(2n)} = \frac{an}{2an} = \frac{n}{2n} = \frac{1}{2}$

Παρατηρούμε πως η σχέση δεν προσδιορίζεται από τον όρο `a`, ο οποίος απαλείφεται αλλά μόνο από τη σχέση των δεδομένων εισόδου, που είναι ανάλογη του n, δηλαδή γραμμική.

Συμπερασματικά μπορούμε να πούμε πως η πολυπλοκότητα χρόνου του επαναληπτικού αλγόριθμου υπολογισμού του παραγοντικού είναι γραμμική, ανάλογη του n.

Για παράδειγμα αν ο χρόνος εκτέλεσης ενός αλγορίθμου είναι T(n)=15n^3, τότε η πολυπλοκότητα χρόνου προσδιορίζεται μόνο από τον όρο n^3, είναι δηλαδή ανάλογη του κύβου του n.

Την πολυπλοκότητα χρόνου μπορούμε να την μετρήσουμε σε τρεις περιπτώσεις:

– στην καλύτερη περίπτωση (best case), όπου T(n) = min T(a_1, a_2, …, a_n)

– την χειρότερη περίπτωση (worst case), όπου T(n) = max T(a_1, a_2, …, a_n)

– την μέση περίπτωση (average case), όπου T(n) = $\frac{1}{|P|}\sum$T(a$_1$, a$_2$, …, a$_n$)

όπου P είναι το πλήθος των δυνατών συνδυασμών από ακολουθίες δεδομένων (a_1, a_2, …, a_n) και |P| είναι ο πληθάριθμος του P.

Για παράδειγμα στην δυαδική αναζήτηση (binary search), όπου αναζητούμε ένα στοιχείο σε ένα ταξινομημένο πίνακα, στην καλύτερη μπορούμε να το βρούμε (αν είμαστε τυχεροί) με μία αναζήτηση, στην χειρότερη περίπτωση με $\log_n$ αναζητήσεις και στη μέση περίπτωση με $\frac{\log n+1}{2}$ αναζητήσεις, αν υποθέσουμε ότι έχουμε μόνο δύο ενδεχόμενα.

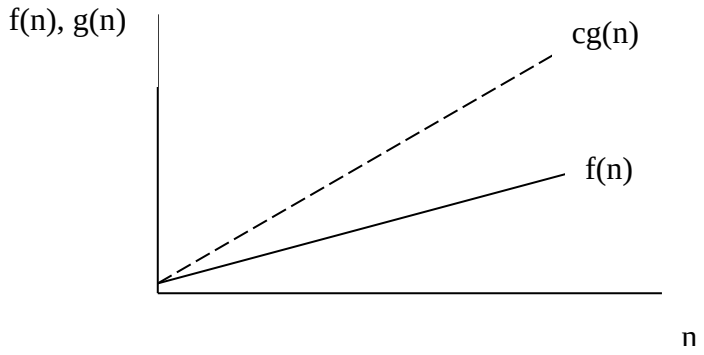
Για τον ορισμό της πολυπλοκότητας χρόνου ενός αλγορίθμου χρησιμοποιούνται κυρίως τρεις συμβολισμοί, που είναι γνωστοί ως υπεροπτικοί (hyperoptic) συμβολισμοί:

– ο συμβολισμός O (Big O notation),
– ο συμβολισμός Θ (Theta notation) και
– ο συμβολισμός Ω (Big Omega notation).

3.14 Συμβολισμός O (Big O Notation)

Ο συμβολισμός O (Big O Notation) δίνει ένα άνω φράγμα της πολυπλοκότητα χρόνου ενός αλγορίθμου στην χειρότερη περίπτωση. Πιο συγκεκριμένα, λέμε ότι `f(n)=O(g(n))` `όταν υπάρχουν c θετικό, n0, τέτοια ώστε f(n)<=c*g(n), για κάθε n>=n0.` Σε μαθηματικούς όρους:

$$f(n) = O(g(n)), \qquad \exists c > 0, n_0: \qquad \forall n \geq n_0, \qquad f(n) \leq cg(n)$$



Σχήμα 3.17: Συμβολισμός Big O

Για παράδειγμα αν ένας αλγόριθμος έχει πολυπλοκότητα `O(n)`, τότε σημαίνει πως το πάνω όριο είναι `c*n`.

Θα μπορούσε όμως να ειπωθεί ότι ο ίδιος αλγόριθμος είναι $O(n^2)$ ή $O(n^3)$ αφού οτιδήποτε μεγαλύτερο του O(n) αποτελεί επίσης άνω φράγμα.

Κάτι τέτοιο όμως θα ήταν λάθος γιατί ο συμβολισμός O χρησιμοποιείται για αυστηρά ασυμπτωτικά φράγματα ώστε να δίνει το μέτρο της πολυπλοκότητας στην χειρότερη περίπτωση.

Για παράδειγμα αν θέλουμε να δείξουμε ότι $f(n) = 3n^2 + 5n = O(n^2)$ εργαζόμαστε ως εξής:

$$f(n) = 3n^2 + 5n \le 3n^2 + 5n^2 \le 8n^2, \qquad \forall n \ge n_0 = 1$$

Άρα, αν θέσουμε $g(n)=n^2$, $c=8$ και $n_0=1$, τότε

$\exists c, n_0: \quad \forall n \ge n_0 : \quad 3n^2 + 5n \le 8n^2$

Αν θέλουμε να δείξουμε ότι $f(n) = \frac{1}{1+n^2} = O(1) = O(n^0)$ εργαζόμαστε ως εξής:

$$f(n) = \frac{1}{1+n^2} \le \frac{1}{1+n^0} \le 1 \ , \qquad \forall n \ge n_0 = 1$$

Άρα, αν θέσουμε $g(n)=n^0$, $c=1$ και $n_0=1$, τότε

$\exists c, n_0: \quad \forall n \ge n_0 : \quad \frac{1}{1+n^2} \le 1n^0 = 1$

Έστω επίσης, δύο αλγόριθμοι A, B ,με $A \equiv O(3n\log_n)$ και $B \equiv O(24n)$. Είναι προφανές πως καλύτερος αλγόριθμος είναι ο Β γιατί είναι γραμμικός, ενώ ο Α έχει επιπλέον τον όρο $\log_n$.

Όμως για μικρές τιμές του n (για ποιες τιμές;) ο Α είναι καλύτερος (γιατί;). Η απάντηση είναι πως θα προτιμηθεί ο A, όταν $3n\log_n < 24n$. Έχουμε:

$$3n\log_n < 24n \Rightarrow 3\log_n < 24 \Rightarrow \log_n < 8 \Rightarrow 2^{\log_n} < 2^8 \Rightarrow n < 256$$

Άρα για τιμές του $n<256$ ο αλγόριθμος Α έχει καλύτερη πολυπλοκότητα, αλλά για μεγαλύτερες τιμές υπερτερεί ο Β.

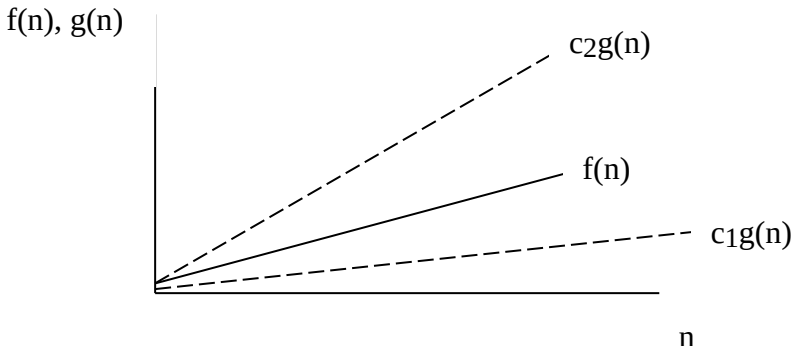
3.15 Συμβολισμός Θ (Big Theta Notation)

Ο συμβολισμός Θ δίνει την πολυπλοκότητα χρόνου ενός αλγορίθμου στη μέση περίπτωση.

Πιο συγκεκριμένα, λέμε ότι f(n)=Θ(g(n)) όταν υπάρχουν c1, c2 θετικά, n0, τέτοια ώστε $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$, για κάθε n>=n0.

Σε μαθηματικούς όρους:

$$f(n) = \Theta(g(n)), \qquad \exists c_1, c_2 > 0, n_0: \qquad \forall n \ge n_0, \qquad 0 \le c_1 g(n) \le f(n) \le c_2 g(n)$$



Σχήμα 3.18: Συμβολισμός Big Θ

Για παράδειγμα αν θέλουμε να δείξουμε ότι $f(n) = (n+1)^2 = \Theta(3n^2)$ εργαζόμαστε ως εξής:

$$f(n) = n^2 + 2n + 1 \le n^2 + 2n^2 + n^2 \le 4n^2, \qquad \forall n \ge n_0 = 1$$

$$f(n) = n^2 + 2n + 1 \ge n^2, \qquad \forall n \ge n_0 = 1$$

Άρα, αν θέσουμε g(n)=3n^2, c$_1$=1/3, c$_2$=4/3 και n$_0$=1, τότε

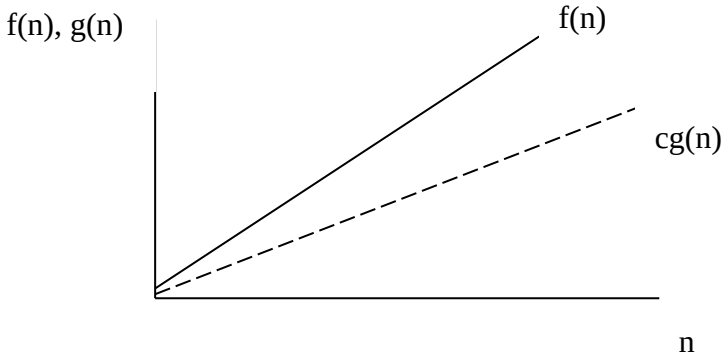
$\exists c, n_0$: $\quad \forall n \geq n_0 : \quad c_1 g(n) \leq f(n) \leq c_2 g(n)$

γιατί $\frac{1}{3}3n^2 \leq f(n) \leq \frac{4}{3}3n^2$ δηλαδή $n^2 \leq f(n) \leq 4n^2$ όπως δείξαμε παραπάνω.

3.16 Συμβολισμός Ω (Big Omega Notation)

Ο συμβολισμός Ω δίνει το κάτω όριο στην πολυπλοκότητα ενός αλγορίθμου άρα δίνει την πολυπλοκότητα χρόνου ενός αλγορίθμου στη καλύτερη περίπτωση.

Πιο συγκεκριμένα, λέμε ότι `f(n)=Ω(g(n))` όταν υπάρχει `c` θετικό, τέτοιο ώστε $f(n) \geq cg(n)$, για απείρως πολλά `n`.



Σχήμα 3.19: Συμβολισμός Big Ω

Αν για παράδειγμα `f(n)=5n`^{5}`-100`, όταν ο `n` είναι περιττός, τότε:

`5n`^{5}`-100 > n`^{5} για `c=1` και άπειρα `n` (οι περιττοί). Άρα `f(n)=Ω(n`^{5}`)`.

Στις περιπτώσεις που η πολυπλοκότητα ενός αλγορίθμου προσδιορίζεται από κάποιο άθροισμα, τότε μπορούμε χρησιμοποιούμε τη μέθοδο της επαγωγής ή τη μέθοδο της αντικατάστασης.

Για παράδειγμα αν θέλουμε να βρούμε ένα άνω φράγμα της συνάρτησης $f(n) = \sum_{i=0}^{n} i$ με τη μέθοδο της αντικατάστασης θα μπορούσε να εργαστούμε ως εξής:

Να αντικαταστήσουμε κάθε όρο i, με τον μέγιστο όρο της ακολουθίας, n. Τότε θα έχουμε: $\sum_{i=0}^{n} i \le \sum_{i=0}^{n} n = n^2$. Άρα f(n)=O(n^2). Η μέθοδος αυτή όμως μπορεί να δώσει πολύ κακό άνω φράγμα.

Αν θέλαμε να δείξουμε ότι ισχύει $f(n) = \sum_{i=0}^{n} i$ =O(n^2), (δηλαδή γνωρίζαμε από την αρχή το ασυμπτωτικό άνω φράγμα που θέλουμε να δείξουμε ότι ισχύει) τότε μπορούμε να χρησιμοποιήσουμε την μέθοδο της επαγωγής:

1. Για n=0 ισχύει $\sum_{i=0}^{n} i$ =O(n^2), γιατί 0<=0

2. Έστω ότι ισχύει για n, δηλαδή έστω ότι ισχύει $\sum_{i=0}^{n} n \le cn^2$

3. Θα δείξουμε ότι ισχύει για (n+1), δηλαδή ότι $\sum_{i=0}^{n} (n+1) \le c(n+1)^2$

Είναι: $\sum_{i=0}^{n} n \le cn^2 \Rightarrow \sum_{i=0}^{n} n + (n+1) \le cn^2 + (n+1) \overset{\text{έστω } c \text{ θετικό}}{\le} cn^2 + 2cn + c)$

Άρα: $\sum_{i=0}^{n} (n+1) \le c(n^2 + 2n + 1) = c(n+1)^2$, ΟΕΔ

3.17 Αναδρομικά Προβλήματα – Μέθοδος Διαίρει και Βασίλευε

Αναδρομικός ονομάζεται ένας αλγόριθμος όταν μέσα στο σώμα των εντολών του καλεί τον ίδιο τον εαυτό του. Για παράδειγμα η συνάρτηση παραγοντικό θα μπορούσε να επιλυθεί με αναδρομή ως εξής:

$$n! = \begin{cases} n*(n-1)! \\ 0! = 1 \end{cases}$$

```
int factorial (int n)
{
      int f;
      if (n=0) f=1;
      else
            f=n*factorial(n-1);
      return f;
}
```

Η αλγοριθμική στρατηγική που ακολουθήθηκε ονομάζεται **"διαίρει και βασίλευε" (divide and conquer)** η οποία επιλύει ένα πρόβλημα επιλέγοντας υποπροβλήματα που λύνει αναδρομικά και μετά συνθέτει τις τμηματικές λύσεις.

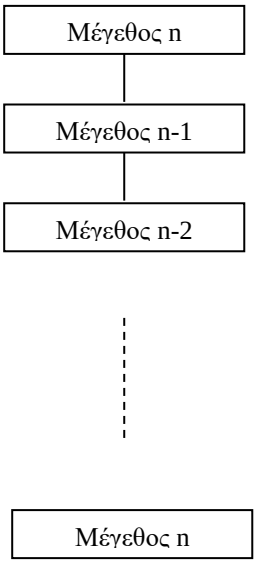
Στη γενική της μορφή η μέθοδος "διαίρει και βασίλευε" περιλαμβάνει τα παρακάτω βήματα:

1. Διαίρεσε το πρόβλημα σε υποπροβλήματα μικρότερου μήκους δεδομένων
2. Επίλυσε τα υποπροβλήματα αναδρομικά
3. Συγχώνευσε τις λύσεις για την επίλυση του προβλήματος

Ο χρόνος επίλυσης των αλγορίθμων μας προσδιορίζεται από το σχήμα των αναδρομικών κλήσεων, που μπορεί να αναπαρασταθεί με ένα δένδρο. Πιο συγκεκριμένα, ο χρόνος επίλυσης προσδιορίζεται από τον χρόνο για την επιλογή των περαιτέρω υποπροβλημάτων και από τον χρόνο σύνθεσης των αποτελεσμάτων.

3.17.1 Παραγοντικό (n!)

Στην απλή περίπτωση επίλυσης του παραγοντικού με αναδρομή το δένδρο των αναδρομικών κλήσεων είναι:



Σχήμα 3.20: Δένδρο αναδρομικών κλήσεων n!

Παρατηρούμε πως επειδή ο παράγοντας διακλάδωσης (branching factor) του δένδρου είναι ένα (1), το δένδρο έχει εκφυλιστεί σε λίστα. Για κάθε υποπρόβλημα απαιτείται σταθερός χρόνος για την επιλογή (μία σύγκριση) και σταθερός χρόνος για την σύνθεση (ένας πολλαπλασιασμός και μία εκχώρηση). Επομένως, ο συνολικός χρόνος στο επίπεδο `k` της λίστας είναι: $1 * O(n^0) = O(1)$

Στο τελευταίο επίπεδο της αναδρομής επίσης ο χρόνος είναι σταθερός (μία εκχώρηση).

Επομένως, στα `n` επίπεδα της αναδρομής ο χρόνος είναι `O(n)`, ο ίδιος όπως και στην επαναληπτική λύση.

3.18 Επαναληπτική Μέθοδος

Η παραπάνω μέθοδος εκτίμησης της πολυπλοκότητας χρόνου ήταν εμπειρική. Η εκτίμηση της πολυπλοκότητας ενός αλγορίθμου γίνεται κυρίως με την επαναληπτική μέθοδο που βασίζεται στο ανάπτυγμα του επαναληπτικού σχήματος, που αποτελείται από το πλήθος και το μέγεθος των υποπροβλημάτων καθώς και τον χρόνο συγχώνευσης των αποτελεσμάτων.

Σύμφωνα με την επαναληπτική μέθοδο η πολυπλοκότητα στο παραπάνω πρόβλημα είναι:

$T(n) = 1*T(n-1) + O(n^0)$,

όπου $1*T(n-1)$ σημαίνει ότι σε κάθε επίπεδο της αναδρομής έχουμε ένα (1) υποπρόβλημα μεγέθους $n-1$ ενώ απαιτείται και σταθερός χρόνος $(O(n^0))$ για την συγχώνευση σε κάθε επίπεδο.

Άρα θα έχουμε:

$T(n)= T(n-1) + O(1) = T(n-2)+O(1)+O(1)=... = T(0)+n*O(1) = O(1) + n*O(1) = O(n+1)=O(n)$

3.18.1 Ακολουθία Fibonacci

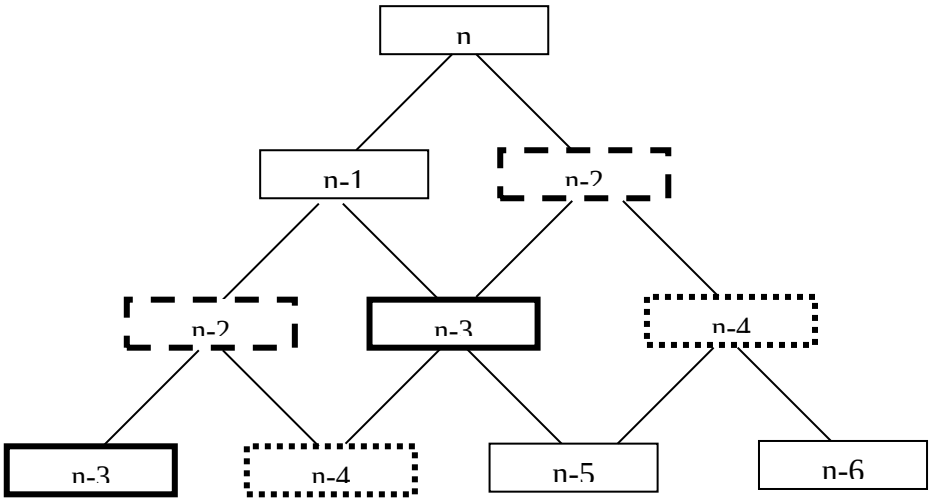
Ένα άλλο παράδειγμα αναδρομικού προβλήματος είναι η ακολουθία **Fibonacci**, από τον γνωστό Ιταλό μαθηματικό του 150υ αιώνα, Leonardo Fibonacci. Η ακολουθία αναπαρίσταται ως $F_0, F_1, F_2, ….,$ όπου $F_0=0, F_1=1$ και για κάθε $n>=2$ ορίζεται από την αναδρομική σχέση $F_n=F_{n-1}+F_{n-2}$.

Επομένως, η πολυπλοκότητα χρόνου της αναδρομικής λύσης είναι:

$T(n) = T(n-1) + T(n-2) = T(n-2) + T(n-3) + T(n-2) > 2T(n-2) > 2(2T(n-2-2)) = 4T(n-4) > 4(2T(n-4-2)) = 8T(n-6) > \ldots > 2^kT(n-2k)$.

Αν κ είναι άρτιος τότε n-2k = 0 και k=n/2, ενώ αν k περιττός τότε n-2k=1 και k=(n-1)/2. Άρα η πολυπλοκότητα χρόνου είναι σε κάθε περίπτωση $2^{n/2}$ ή $2^{(n-1)/2}$ δηλαδή **εκθετική**.

Το βασικό πρόβλημα στον αναδρομικό υπολογισμό της ακολουθίας Fibonacci είναι ότι σε κάθε βήμα της αναδρομής η αναδρομική συνάρτηση καλείται ξανά και ξανά με το ίδιο όρισμα.



Σχήμα 3.21: Δένδρο κλήσεων αναδρομικού Fibonacci

Υπάρχει καλύτερος αλγόριθμος από τον εκθετικό; Ναι, αν χρησιμοποιούσαμε μία δομή πίνακα που να αποθηκεύουμε τις τιμές fibonacci που υπολογίζουμε, τότε με ένα απλό επαναληπτικό σχήμα θα μπορούσαμε να βρούμε τον ν-οστό όρο σε χρόνο γραμμικό.

```
int linearFibonacci (int n)
```

```
{
      int A[n];  // πίνακας από 0 έως n
      A[0]=0; A[1]=1;
      for (int i=2; i<=n; i++)
            A[i]=A[i-1]+A[i-2]
      return A[n];
}
```

Μπορούμε να έχουμε καλύτερο χρόνο από τον γραμμικό;

Η απάντηση είναι πάλι ναι! Αν χρησιμοποιήσουμε πολλαπλασιασμό πινάκων $\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$, τότε μπορεί να λυθεί το πρόβλημα σε χρόνο $\log_n$.

3.19 Master Theorem

Ένας μεγάλος αριθμός αναδρομικών προβλημάτων ακολουθεί το εξής γενικό σχήμα:

1. πρόβλημα μεγέθους n
2. Αναδρομική επίλυση a υποπροβλημάτων μεγέθους n/b
3. Σύνθεση των επιμέρους λύσεων σε χρόνο $O(n^d)$

Ο συνολικός χρόνος είναι $T(n)=aT(n/b) + O(n^d)$.

Σύμφωνα με το Master Theorem (CLR) ισχύει ότι:

$$T(n) = \begin{cases} O(n^d) & a < b^d \\ O(n^d \log n) & a = b^d \\ O(n^{\log_b a}) & a > b^d \end{cases}$$

3.20 Πολλαπλασιασμός Πινάκων

Ο πολλαπλασιασμός πινάκων μεγέθους $n \times n$ εύκολα διαιρείται σε υποπροβλήματα, αν επιλέξουμε υποπίνακες μεγέθους $n/2 \times n/2$.

Αν θεωρήσουμε τους υποπίνακες ως απλά στοιχεία, τότε έχουμε:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Η πολυπλοκότητα χρόνου είναι `T(n) = 8T(n/2) + O(n^2)` γιατί όπως παρατηρούμε έχουμε 8 πολλαπλασιασμούς υποπινάκων μεγέθους `n/2` και `4` προσθέσεις `(n^2)`.

Σύμφωνα με το Master Theorem είναι `a=8, b=2, d=2,` άρα `a>b^d` και επομένως η πολυπλοκότητα είναι $O(n^{\log_b a}) = O(n^{\log_2 8}) = O(n^3)$.

Ο πολλαπλασιασμός πινάκων έχει δειχθεί πως μπορεί να έχει καλύτερη πολυπλοκότητα χρόνου γιατί με ένα μικρό τρικ μπορούμε να μειώσουμε τους πολλαπλασιασμούς που χρειάζονται από οκτώ σε επτά μειώνοντας έτσι την πολυπλοκότητα σε $O(n^{\log_2 7}) = O(n^{2.81})$.

3.21 Δυαδική Αναζήτηση

Η δυαδική αναζήτηση, δηλαδή η αναζήτηση ενός στοιχείου σε ένα ταξινομημένο πίνακα, υλοποιείται με την επιλογή του μεσαίου στοιχείου του πίνακα και την σύγκριση με το στοιχείο που αναζητούμε.

Αν το στοιχείο βρεθεί ο αλγόριθμος σταματά διαφορετικά ξανακαλείται αναδρομικά, δηλαδή αν α αριθμός που ψάχνουμε είναι μικρότερος, τότε κάνουμε πάλι δυαδική αναζήτηση στο πάνω-μισό του πίνακα, αν είναι μεγαλύτερος κάνουμε αναζήτηση στο κάτω-μισό, κλπ., μέχρι να βρούμε το στοιχείο με διαδοχικές κατατμήσεις του πίνακα και συγκρίσεις του μεσαίου κάθε φορά στοιχείου.

```
int binarySearch(int value, int low, int high)
{
       int high=n, mid, low=0;
       if high < low return not found;
       mid = (high+low)/2;
       if a[mid] == value     return mid;
       if value < a[mid]
              return binarySearch(value, low, mid-1);
       else
              return binarySearch(value, mid+1, high);
}
```

Ο αλγόριθμος αυτός είναι αναδρομικός γιατί καλεί τον εαυτό του -δηλαδή εκτελεί την ίδια πράξη (δυαδική αναζήτηση)- αναδρομικά.

Η πολυπλοκότητά του είναι: $T(n)= T(n/2) + O(n^0)$, όπου $T(n/2)$ είναι ο χρόνος του υποπροβλήματος με μέγεθος n/2 και $O(n^0)$ είναι ο χρόνος για την σύνθεση της λύσης που είναι σταθερός (μία σύγκριση και ένα return: `if a[mid] == value return mid;`)

Είναι λοιπόν: $T(n)=O(1) + T(n/2) = 2*O(1) + T(n/2^2) = \ldots = k*O(1) + T(n/2^k)$.

Στο τελευταίο επίπεδο της αναδρομής έχουμε επιλύσει k υποπροβλήματα, όπου $n=2^k$ και είναι επομένως: $k=\log_n$. Τότε είναι $T(n) = \log_n + O(1)$

Επομένως η πολυπλοκότητα χρόνου της δυαδικής αναζήτησης είναι $O(\log_n)$.

Με το Master Theorem είναι: $a=1, b=2, d=0$ και είναι $a=b^d$ και επομένως είναι $O(n^d \log_n) = O(\log_n)$

3.22 Μέγιστο/Ελάχιστο Πίνακα

Ο υπολογισμός του ελάχιστου ή μέγιστου ενός αταξινόμητου πίνακα μπορεί να υλοποιηθεί με ένα απλό αλγόριθμο ως εξής:

```
int min (int low, int high)
{
      int tmpMin=A[low];
      if (low<high)
      {
            tmp = min(low+1, high);
            if (tmpMin >  tmp) tmpMin = tmp;
      }
      return tmpMin;
}
```

Και εδώ η αναδρομή εκτελείται n φορές με κόστος σταθερό, άρα η συνολική πολυπλοκότητα είναι γραμμική O(n).

3.23 Ταξινόμηση

Έστω ένας πίνακας Α με τυχαία στοιχεία τα οποία θέλουμε να διατάξουμε με μια σχέση διάταξης είτε από το μικρότερο προς το μεγαλύτερο, είτε το αντίθετο. Η διαδικασία διάταξης των στοιχείων ονομάζεται ταξινόμηση.

Για να ταξινομήσουμε τα στοιχεία ενός πίνακα μπορεί να χρησιμοποιηθεί η μέθοδος "διαίρει και βασίλευε" ως εξής:

1. να χωρίσουμε τον πίνακα σε μικρότερους υποπίνακες,
2. να διατάξουμε τα στοιχεία του κάθε υποπίνακα και
3. να συγχωνεύσουμε τα αποτελέσματα.

Στη διαδικασία (1) ο χωρισμός (split) του πίνακα γίνεται με την επιλογή (selection) ενός στοιχείου του πίνακα σύμφωνα με κάποια μέθοδο. Για παράδειγμα ένας τρόπος είναι να επιλέξουμε το μεσαίο στοιχείο του πίνακα, δηλαδή αν ένας πίνακας έχει 10 στοιχεία να επιλέξουμε το $5^{ο}$ στοιχείο και να χωρίσουμε τον πίνακα σε δύο ίσους περίπου υποπίνακες.

Άλλος τρόπος είναι να επιλέξουμε τον μέσο (median) του πίνακα ώστε κάθε στοιχείο στον ένα υποπίνακα να είναι μικρότερο του μέσου και κάθε στοιχείο του δεύτερου υποπίνακα να είναι μεγαλύτερο του μέσου.

Ο πρώτος τρόπος είναι ένας εύκολος τρόπος διαχωρισμού του πίνακα (easy split) γιατί απλά επιλέγουμε το μεσαίο στοιχείο, ενώ ο δεύτερος τρόπος είναι δύσκολος (hard split) γιατί πρέπει να βρούμε και να επιλέξουμε τον μέσο όλων των στοιχείων.

Στη διαδικασία (3) η συγχώνευση μπορεί επίσης να είναι εύκολη (easy join) αν τα στοιχεία των υποπινάκων έχουν προκύψει με την δύσκολη μέθοδο διαχωρισμού, γιατί τότε απλά συνενώνουμε τους πίνακες (αφού στον πρώτο υποπίνακα όλα τα στοιχεία είναι μικρότερα από τα στοιχεία του δεύτερου υποπίνακα) ή δύσκολη (hard join) αν τα στοιχεία των υποπινάκων έχουν προκύψει με την εύκολη μέθοδο διαχωρισμού, γιατί τότε πρέπει να ταξινομήσουμε τα στοιχεία των ταξινομημένων πινάκων και όχι απλά να τα συνενώσουμε.

Επομένως μπορούμε να έχουμε δύο κατηγορίες αλγορίθμων ταξινόμησης:

1. Easy split και hard join
2. Hard split και easy join

3.23.1 MergeSort

Στη μέθοδο ταξινόμησης MergeSort τέμνουμε διαδοχικά τον πίνακα με βάση το μεσαίο στοιχείο κάθε πίνακα/υποπίνακα και αφού ταξινομήσουμε κάθε υποπίνακα στη συνέχεια συγχωνεύουμε διαδοχικά κάθε δυάδα υποπινάκων.

Η μέθοδος Mergesort ανήκει στην κατηγορία easy split-hard join και δίνεται παρακάτω:

```
int* mergesort(int* A, int s, int n)
{
       if length(A) ≤ 1 return A
       else
       {
              int middle = length(A) / 2
              int* left = mergesort(A, s, middle)
              int* right = mergesort(A, middle+1, n)
              return merge(left, right)
       }
}


int* merge(int* left, int* right)
{
       int* result;
       while (length(left) > 0 OR length(right)) > 0
       if ((length(left)>0) AND (first(left)≥first(right)))
       {
                     insert first(left) to result;
                     left = rest(left);
       }
       else
       {
                     append first(right) to result;
                     right = rest(right);
```

```
        }
        return result;
}
```

Η πολυπλοκότητα χρόνου είναι:

$T(n) = 2\ T(n/2) + O(n) = 2^2 T(n/2^2) + 2\ O(n) = \ldots = 2^k T(n/2^k) + k\ O(n)$

γιατί για την ταξινόμηση κάθε ενός από τα δύο τμήματα θέλουμε χρόνο $T(n/2)$ για το καθένα και για την συγχώνευση θέλουμε το πολύ n συγκρίσεις. Το δένδρο της αναδρομής έχει συντελεστή διακλάδωσης δύο και επομένως σε κάθε επίπεδο k έχει 2^k υποπίνακες, ενώ $k=\log n$ (ύψος δυαδικού δένδρου).

Τελικά έχουμε με διαδοχικές διαιρέσεις ότι:

$T(n) = 2^{\log n} * T(n/2^{\log n}) + \log_n * O(n) = n + n\log n$, και επομένως ο αλγόριθμος mergesort είναι $O(n \log_n)$.

Με το Master Theorem μπορούσαμε πιο εύκολα να βρούμε ότι:

$a=2,\ b=2,\ d=1$ και είναι $a=b^d$ και επομένως είναι $O(n^d \log_n) = O(n \log_n)$

3.23.2 QuickSort

Η ταξινόμηση quicksort ανήκει στην κατηγορία hard split-easy join γιατί ο αρχικός πίνακας θα πρέπει σε κάθε φάση της αναδρομής να χωρίζεται σε δύο υποπίνακες με βάση τον μέσο (median) του πίνακα. Ο μέσος (median) είναι το στοιχείο ενός πίνακα, όπου τα μισά στοιχεία του πίνακα είναι μικρότερα από τον μέσο και τα άλλα μισά μεγαλύτερα.

Αν έστω m ο μέσος κάθε υποπίνακα, τότε κάθε στοιχείο, $a_{left\,i}$ του αριστερού υποπίνακα θα είναι $a_{left\,i} < m$, ενώ κάθε στοιχείο $a_{right\,i}$ του δεξιού υποπίνακα θα είναι $a_{right\,i} > m$.

Τελικά, αφού τα στοιχεία των πινάκων ταξινομηθούν η διαδικασία της συγχώνευσης είναι απλά μία συνένωση των υποπινάκων.

```
int* quicksort(int* A)
{
     int* less, medianList, greater
     if length(A) ≤ 1        return A
     Επέλεξε τον median από τον A
     for each x in A except median
         if x < median then add x to less
         if x ≥ median then add x to greater
     add median to medianList
     return concatenate(quicksort(less), medianList,
                                          quicksort(greater));
}
```

Στην περίπτωση που σε κάθε στάδιο της split επιλέγουμε τον median, τότε η πολυπλοκότητα θα είναι O(nlogn), γιατί θα έχουμε logn υποπροβλήματα στο δένδρο της αναδρομής με χρόνο O(n) (για την επιλογή του median) για το κάθε υποπρόβλημα.

Αν όμως δεν επιλέγουμε τον median και χωρίζουμε σε υποπίνακες, όπου για παράδειγμα ο ένας υποπίνακας έχει μήκος $ένα$ και ο άλλος υποπίνακας μήκος $n-1$, τότε το δένδρο της αναδρομής εκφυλίζεται σε λίστα και έχουμε χρόνο $O(n^2)$.

3.23.3 Selection Sort

Η ταξινόμηση Selection sort αποτελεί αυτό που παραπάνω χαρακτηρίσαμε ως εκφυλισμό της quicksort, δηλαδή σε κάθε βήμα του διαχωρισμού του πίνακα επιλέγεται ως οδηγός (pivot) όχι ο median αλλά το ελάχιστο στοιχείο του πίνακα, ώστε ο αρχικός

πίνακας να χωρίζεται σε δύο υποπίνακες, όπου ο αριστερός υποπίνακας να έχει ένα στοιχείο (το ελάχιστο) και ο άλλος υποπίνακας να έχει n-1 στοιχεία.

```
int selectionSort(int data[], int count)
{
        int i, j;
        int tmp;
        int minimum;
        for (i = 0; i < count - 1; i++)
       {
                minimum = i; /* current minimum */
                /* find the global minimum */
                for (j = i + 1; j < count; j++)
                    {
                        if (data[minimum] > data[j])
                           {
                                /* new minimum */
                                minimum = j;

                           }
                    }
                /* swap data[i] and data[minimum] */
                tmp = data[i];
                data[i] = data[minimum];
                data[minimum] = tmp;
        }
        return 0;
}
```

Η πολυπλοκότητα του Selection sort είναι $O(n^2)$.

3.23.4 Heap Sort

Η ταξινόμηση με τη μέθοδο της σωρού χρησιμοποιεί ως δομή δεδομένων μία σωρό, όπως αναφέραμε προηγούμενα στις δομές δεομένων. Ο χρόνος κατασκευής της σωρού είναι γραμμικός O(n) ενώ ο τελικός χρόνος για την ταξινόμηση της σωρού είναι O(nlogn).

Κατά τη διαδικασία του αλγορίθμου σε κάθε βήμα γίνεται αμοιβαία ανταλλαγή του τελευταίου στοιχείου της σωρού με το πρώτο (άρα κατεβαίνει κάτω το μικρότερο στοιχείο) και στη συνέχεια με μια διαδικασία shiftdown (A[1], 1, n-1), αναπροσαρμόζουμε τη σωρό ώστε το στοιχείο A[1] να τοποθετηθεί στο σωστό κόμβο της τρέχουσας σωρού (από το στοιχείο 1 έως το στοιχείο n-1). Η διαδικασία shifdown έχει πολυπλοκότητα logn ενώ κατά τη διαδικασία ταξινόμησης εκτελείται n φορές, δίνοντας συνολικό χρόνο O(nlogn).

3.23.5 Bubble Sort

Στην ταξινόμηση με τη διαδικασία bubble sort γίνεται με αμοιβαία ανταλλαγή του μικρότερου (ή μεγαλύτερου) στοιχείου με το πρώτο κάθε φορά στοιχείο κάθε υποπίνακα.

```
function bubblesort (A : array[1..n])
{
	int i, j;
	for i from n downto 1
	{
		for j from 1 to i-1
		{
			if (A[j] > A[j+1]);
			swap(A[j], A[j+1]);
		}
	}
}
```

Η διαδικασία εύρεσης του μικρότερου ή μεγαλύτερου στοιχείου έχει γραμμική πολυπλοκότητα και επομένως αφού εκτελείται για όλα τα `n` στοιχεία, τελικά ο συνολικός χρόνος είναι $O(n^2)$.

3.24 Πολυπλοκότητα ταξινόμησης

Οι μέθοδοι ταξινόμησης που αναλύθηκαν παραπάνω βασίζονται σε διαδοχικές συγκρίσεις (comparison sort) των n στοιχείων. Αποδεικνύεται πως κάθε αλγόριθμος που ταξινομεί n στοιχεία μόνο με συγκρίσεις έχει κάτω-όριο (lower bound) `O(nlogn)`.

Υπάρχουν και άλλοι αλγόριθμοι ταξινόμησης που δεν βασίζονται σε συγκρίσεις αλλά σε μεθόδους κατάταξης των στοιχείων σύμφωνα με το περιεχόμενό τους, όπως οι αλγόριθμοι Bin Sort και Radix Sort που μπορούν να δώσουν πολυπλοκότητες γραμμικές O(n).