Τμήμα ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ

Διαδικασιακός Προγραμματισμός

Διάλεξη 14η
Διαχείριση Μνήμης και Δομές Δεδομένων

Οι διαλέξεις βασίζονται στο βιβλίο των Τσελίκη και Τσελίκα
C: Από τη Θεωρία στην Εφαρμογή

Σωτήρης Χριστοδούλου

Διαχείριση Μνήμης

- Η **διαχείριση της μνήμης** ενός υπολογιστή μπορεί να γίνει με δύο τρόπους: με τον **στατικό** τρόπο ή με τον **δυναμικό** τρόπο

- Σε όλα τα προγράμματα που έχουμε δει μέχρι αυτό το σημείο, η δέσμευση μνήμης γινόταν με τον στατικό τρόπο

- Ο μεταγλωττιστής χωρίζει τη διαθέσιμη μνήμη του υπολογιστή στα ακόλουθα τέσσερα τμήματα:

  1) Στο τμήμα που ονομάζεται **στοίβα (stack)** και χρησιμοποιείται για τη δέσμευση μνήμης με **στατικό τρόπο**

  2) Στο τμήμα που ονομάζεται **σωρός (heap)** και χρησιμοποιείται για τη δέσμευση μνήμης με **δυναμικό τρόπο**

  3) Στο τμήμα από το οποίο δεσμεύεται μνήμη **για τις καθολικές μεταβλητές** του προγράμματος

  4) Στο τμήμα από το οποίο δεσμεύεται μνήμη **για την αποθήκευση του κώδικα** (π.χ. εντολές, συναρτήσεις, ...) του προγράμματος

Στατική Δέσμευση Μνήμης (Ι)

- Με τον στατικό τρόπο δέσμευσης, ο προγραμματιστής πρέπει να γνωρίζει εκ των προτέρων το ακριβές μέγεθος της μνήμης που χρειάζεται να δεσμεύσει
- Το μέγεθος της μνήμης που δεσμεύεται με στατικό τρόπο δεν μπορεί να μεταβληθεί κατά τη διάρκεια του προγράμματος
- Π.χ. όταν δημιουργούμε ένα πρόγραμμα για την αποθήκευση των βαθμών 500 φοιτητών, τότε γνωρίζουμε ότι πρέπει να δεσμεύσουμε μνήμη για 500 φοιτητές και όχι για 300 ή 600 (δηλ. το πρόγραμμα καθορίζει ακριβώς το μέγεθος της απαιτούμενης μνήμης)
- Άρα, με την εντολή: `float grades[500];`
  δεσμεύουμε με στατικό τρόπο από τη μνήμη του υπολογιστή `500*4 = 2000 bytes`
- Το μέγεθος του πίνακα `grades` δεν μπορεί να αλλάξει κατά τη διάρκεια του προγράμματος

Στατική Δέσμευση Μνήμης (ΙΙ)

- Οι δηλώσεις μεταβλητών αποτελούν χαρακτηριστικά παραδείγματα στατικής δέσμευσης μνήμης, π.χ.

```
int a; /* Δέσμευση 4 bytes μνήμης. */
double b; /* Δέσμευση 8 bytes μνήμης. */
int* ptr; /* Δέσμευση 4 bytes μνήμης. */
struct student stud; /* Δέσμευση μνήμης ίση με το
                        μέγεθος της δομής student */
char str[200]; /* Δέσμευση 200 bytes μνήμης. */
```

- Για κάθε τύπο μεταβλητών που δηλώνεται σε ένα πρόγραμμα (π.χ. αριθμητικές μεταβλητές, πίνακες, δομές, συναρτήσεις...) ο μεταγλωττιστής δεσμεύει την αντίστοιχη μνήμη

- Αυτή η δέσμευση της μνήμης γίνεται σε ένα συγκεκριμένο τμήμα μνήμης που παρέχει το λειτουργικό σύστημα στο πρόγραμμα και ονομάζεται στοίβα (stack)

- Η αποδέσμευση αυτής της μνήμης γίνεται αυτόματα, όταν τερματιστεί η εκτέλεση της συνάρτησης στην οποία δηλώνονται

Παρατηρήσεις

- Η μνήμη που απαιτείται για τις τοπικές μεταβλητές μίας συνάρτησης **δεσμεύεται στατικά σε κάθε κλήση της συνάρτησης** αυτής και **αποδεσμεύεται όταν τερματίζεται η εκτέλεσή της**
- Π.χ. η μνήμη που απαιτείται για τις τοπικές μεταβλητές της συνάρτησης `test()` είναι 812 bytes

```
void test()
{
        int i,j,k,arr[200];
        ...
}
```

- Παρομοίως, η μνήμη που έχει δεσμευτεί για τις τοπικές μεταβλητές της συνάρτησης `main()` **αποδεσμεύεται όταν τερματίζεται η εκτέλεση του προγράμματος**
- Η μνήμη που δεσμεύεται με **στατικό** τρόπο για τις τοπικές μεταβλητές μίας συνάρτησης αποδεσμεύεται μετά τον τερματισμό της συνάρτησης και οι τιμές των τοπικών μεταβλητών **χάνονται**
- Άρα, να **μην χρησιμοποιείτε** στο πρόγραμμά σας διευθύνσεις μεταβλητών που δηλώνονται **τοπικά** μέσα σε συναρτήσεις

Δυναμική Δέσμευση Μνήμης (Ι)

- Η **δυναμική δέσμευση μνήμης** χρησιμοποιείται όταν ο προγραμματιστής **δεν γνωρίζει εκ των προτέρων** το μέγεθος της μνήμης που χρειάζεται να δεσμεύσει, αφού το μέγεθος της μνήμης που δεσμεύεται με δυναμικό τρόπο **μπορεί να μεταβληθεί κατά τη διάρκεια του προγράμματος**
- Π.χ. όταν θέλουμε να δημιουργήσουμε ένα πρόγραμμα, το οποίο πρέπει να αποθηκεύει τους βαθμούς φοιτητών, αλλά **δεν γνωρίζουμε τον αριθμό των φοιτητών** τότε **δεν πρέπει** να δεσμεύσουμε μνήμη **με στατικό τρόπο**
- Αν δηλαδή, δηλώσουμε: `float grades[500];`

  τότε δεσμεύουμε με στατικό τρόπο μνήμη για 500 βαθμούς φοιτητών
- Όμως, αν οι φοιτητές **είναι περισσότεροι από** 500, τότε **δεν δεσμεύεται μνήμη για τους βαθμούς όλων των φοιτητών** αλλά μόνο για τους 500 πρώτους
- Επίσης, αν οι φοιτητές **είναι τελικά λιγότεροι από** 500, τότε έχουμε **σπατάλη μνήμης**, αφού **δεσμεύεται περισσότερη από την απαιτούμενη μνήμη**

Δυναμική Δέσμευση Μνήμης (ΙΙ)

- Με τη **δυναμική δέσμευση** μνήμης ο προγραμματιστής μπορεί να δεσμεύσει τη μνήμη που χρειάζεται από το τμήμα μνήμης του υπολογιστή που ονομάζεται **σωρός** (**heap**)
- Τα όρια της μνήμης για τον σωρό και τη στοίβα είναι πεπερασμένα, αλλά **το μέγεθος του σωρού είναι σαφέστατα μεγαλύτερο από το προκαθορισμένο μέγεθος της στοίβας**
- Π.χ. με την παρακάτω δήλωση πίνακα:

```
int arr[1000000];
```

το πρόγραμμα μπορεί να μην βρει την απαιτούμενη μνήμη **στη στοίβα** και να μην εκτελεστεί, ενώ με τις εντολές:

```
int* arr;
arr = (int*)malloc(1000000 * sizeof(int));
```

το πρόγραμμα θα βρει την απαιτούμενη μνήμη **στον σωρό** και θα εκτελεστεί χωρίς πρόβλημα

- Η μνήμη που δεσμεύεται με δυναμικό τρόπο **πρέπει να αποδεσμεύεται**, όταν πλέον δεν χρειάζεται

Η συνάρτηση `malloc()`

- Η συνάρτηση `malloc()` χρησιμοποιείται για τη δυναμική δέσμευση ενός συγκεκριμένου μεγέθους μνήμης
- Το πρωτότυπό της δηλώνεται στο αρχείο `stdlib.h` και είναι το ακόλουθο:

```
void* malloc(unsigned int size);
```

- Η παράμετρος `size` δηλώνει πόσα bytes μνήμης θα δεσμευτούν
  - Αν η δέσμευση της μνήμης είναι επιτυχημένη, τότε η συνάρτηση επιστρέφει έναν δείκτη προς το πρώτο byte που δεσμεύτηκε, δηλαδή στην αρχή της δεσμευμένης μνήμης
  - Αν η δέσμευση της μνήμης αποτύχει, τότε η συνάρτηση επιστρέφει την τιμή `NULL`
- Η συνάρτηση `malloc()` επιστρέφει έναν δείκτη σε τύπο `void`, που σημαίνει ότι στη δεσμευμένη μνήμη μπορεί να αποθηκευτεί οποιοσδήποτε τύπος δεδομένων
- Παρόλα αυτά, προτείνεται να δηλώνεται ο τύπος των δεδομένων που θα αποθηκευτεί στη μνήμη (με χρήση typecast, όπως θα δείτε και στα επόμενα παραδείγματα)

Παράδειγμα χρήσης της malloc() (1)

```
int* ptr;
ptr = (int*) malloc(100);
```

- Το (int*) πριν από την κλήση της malloc() δηλώνει ρητά ότι η μνήμη που θα δεσμευτεί θα χρησιμοποιηθεί για την αποθήκευση ακεραίων αριθμών
- Αν η δέσμευση της μνήμης είναι επιτυχημένη, τότε θα δεσμευτούν 100 bytes μνήμης και ο δείκτης ptr θα δείχνει στην αρχή αυτής της μνήμης
- Άρα, αφού κάθε ακέραιος απαιτεί 4 bytes, θα μπορούν να αποθηκευτούν συνολικά 25 ακέραιοι σε αυτή τη μνήμη
- Αντί για 100, θα μπορούσαμε (και μάλιστα συνηθίζεται) να γράψουμε 25*sizeof(int)
- Αν η δέσμευση αποτύχει, τότε η τιμή του ptr θα είναι ίση με NULL

Παράδειγμα χρήσης της malloc() (2)

```
char* ptr;
ptr = (char*) malloc(n+1);
```

- Με την παραπάνω δήλωση δεσμεύουμε δυναμικά μνήμη για την αποθήκευση n χαρακτήρων
- Η επιπλέον θέση (παρατηρείστε το n+1) δεσμεύεται για την αποθήκευση του τερματικού χαρακτήρα ('\0')

Παράδειγμα χρήσης της malloc() (3)

```
struct student* ptr;
ptr = (struct student*) malloc(100 * sizeof(student));
```

- Το (struct student*) πριν από την κλήση της malloc() δηλώνει ότι η μνήμη που θα δεσμευτεί θα χρησιμοποιηθεί για την αποθήκευση 100 δομών τύπου student
- Τα bytes μνήμης που θα δεσμευτούν θα είναι ίσα με το γινόμενο του 100 επί το μέγεθος της μνήμης που καταλαμβάνει μία τέτοια δομή (τύπου student)
- Ο δείκτης ptr θα δείχνει στην αρχή αυτής της μνήμης

Παρατηρήσεις (I)

- Τον δείκτη που επιστρέφει η συνάρτηση `malloc()` μπορούμε να τον χειριστούμε είτε σαν δείκτη είτε σαν πίνακα
- Δεδομένου ότι το όνομα ενός πίνακα είναι δείκτης στο πρώτο του στοιχείο, μπορούμε να θεωρήσουμε ότι η `malloc()` δημιουργεί έναν μονοδιάστατο πίνακα που έχει ως όνομα το όνομα του δείκτη
- Π.χ. αν δεσμεύσουμε μνήμη για 1000 ακέραιους

```
int* ptr;
ptr = (int*) malloc(1000*sizeof(int));
```

και θέλουμε να αποθηκεύσουμε στον πρώτο ακέραιο αυτής της μνήμης την τιμή 13, θα μπορούσαμε να γράψουμε:

- είτε `*ptr = 13;`
- είτε `ptr[0] = 13;`

Παρατηρήσεις (II)

- Να ελέγχετε πάντα αν η τιμή επιστροφής της `malloc()` είναι διαφορετική από `NULL`
- Για τον καθορισμό του μεγέθους της μνήμης να χρησιμοποιείτε τον τελεστή `sizeof()`, ώστε το πρόγραμμα να μπορεί να εκτελείται σε διαφορετικούς υπολογιστές
- Π.χ. ο τύπος `short int` σε έναν υπολογιστή μπορεί να δεσμεύει 2 bytes και σε κάποιον άλλο 4 bytes
- Όταν δεν χρειάζεστε άλλο τη δεσμευμένη μνήμη, πρέπει να την απελευθερώνετε, ώστε να μπορεί το λειτουργικό σύστημα να τη διαθέσει σε άλλα προγράμματα
- Η αποδέσμευση της μνήμης γίνεται με τη συνάρτηση `free()`
- Ο γενικός κανόνας είναι ότι η μνήμη που δεσμεύεται με κάθε κλήση της συνάρτησης `malloc()` πρέπει να αποδεσμεύεται με αντίστοιχη κλήση της συνάρτησης `free()`

Παρατηρήσεις (III)

- Η προσαρμογή (typecast) του τύπου επιστροφής της `malloc()` δεν είναι υποχρεωτική
- Δηλαδή, π.χ αντί για :

```
int* ptr;
ptr = (int*) malloc(100);
```

Θα μπορούσαμε να γράψουμε:

```
int* ptr;
ptr = malloc(100);
```

- Αν και η πρώτη σύνταξη είναι πιο δυσνόητη, προτείνεται γιατί:

α) φαίνεται ξεκάθαρα ο τύπος των δεδομένων που θα περιέχει η μνήμη και δεν χρειάζεται ο αναγνώστης του κώδικα να ανατρέξει στη δήλωση της μεταβλητής `ptr` για να το θυμηθεί

β) Θα μπορεί το πρόγραμμά σας να μεταγλωττιστεί σε κάποιον άλλο μεταγλωττιστή που μπορεί να απαιτεί typecast (π.χ. σε C++ μεταγλωττιστή)

Η συνάρτηση free()

- Η συνάρτηση free() χρησιμοποιείται για την απελευθέρωση μνήμης που δεσμεύτηκε δυναμικά
- Το πρωτότυπό της δηλώνεται στο αρχείο stdlib.h και είναι το ακόλουθο:

```
void free(void* ptr);
```

- Η παράμετρος ptr είναι δείκτης στην αρχή της δεσμευμένης μνήμης, την οποία θα αποδεσμεύσει η free()
- ΠΡΟΣΟΧΗ: Αν ο δείκτης-παράμετρος της free() δεν δείχνει σε μία δεσμευμένη μνήμη, τότε το πρόγραμμα δεν θα λειτουργήσει
- Π.χ. το διπλανό πρόγραμμα δεν θα λειτουργήσει, γιατί ο δείκτης ptr έχει απλώς οριστεί και δεν δείχνει σε κάποια δεσμευμένη μνήμη...

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int* ptr;
        free(ptr);
        return 0;
}
```

Οι συναρτήσεις memcpy() και memmove()

- Η συνάρτηση memcpy() χρησιμοποιείται για την αντιγραφή οποιουδήποτε τύπου δεδομένων από μία περιοχή μνήμης σε μία άλλη
- Το πρωτότυπό της δηλώνεται στο αρχείο string.h και είναι το ακόλουθο:

```
void* memcpy(void* dest,void* src,unsigned int size);
```

- Η συνάρτηση memcpy() αντιγράφει size bytes από την περιοχή μνήμης στην οποία δείχνει ο δείκτης src στην περιοχή μνήμης στην οποία δείχνει ο δείκτης dest
- Το πρωτότυπο της συνάρτησης memmove() είναι ίδιο με αυτό της memcpy()
- Η κύρια διαφορά μεταξύ των δύο συναρτήσεων είναι ότι η memmove() εξασφαλίζει τη σωστή αντιγραφή των δεδομένων, ακόμα και αν οι δύο περιοχές μνήμης επικαλύπτονται, ενώ η memcpy() δεν εξασφαλίζει κάτι τέτοιο και, επειδή δεν ελέγχει αν οι δύο περιοχές μνήμης επικαλύπτονται, εκτελείται πιο γρήγορα από τη memmove()

Παρατηρήσεις

- Για τη μνήμη προορισμού **πρέπει** να έχουν δεσμευτεί τουλάχιστον `size` bytes, σε διαφορετική περίπτωση, τα πλεονάζοντα bytes θα εγγραφούν σε **μη δεσμευμένη μνήμη**, η οποία μπορεί να χρησιμοποιείται για άλλους σκοπούς
- Π.χ. η επόμενη αντιγραφή **δεν είναι σωστή**, γιατί το μέγεθος της μνήμης προορισμού είναι 3 bytes, ενώ τα bytes που θα αντιγραφούν είναι 6

```
char str1[3];
char str2[] = "abcde";
memcpy(str1,str2,sizeof(str2));
```

- Οι συναρτήσεις `memcpy()` και `memmove()` είναι πολύ **χρήσιμες** γιατί **μπορούμε να αντιγράψουμε άμεσα μεγάλο όγκο δεδομένων** από μία περιοχή μνήμης σε μία άλλη
- Για παράδειγμα, αν θέλετε να αντιγράψετε τα περιεχόμενα ενός πίνακα `100000` ακεραίων σε έναν άλλον πίνακα, τότε να χρησιμοποιήσετε τη `memcpy()` και όχι `for` βρόχο, γιατί **η αντιγραφή θα γίνει πιο γρήγορα**

Παράδειγμα

- Γράψτε ένα πρόγραμμα το οποίο να ορίζει δύο πίνακες 100000 ακεραίων, να θέτει τις τιμές από 1 έως 100000 στα στοιχεία του πρώτου πίνακα και να αντιγράφει τις τιμές των στοιχείων του στον δεύτερο πίνακα με χρήση της συνάρτησης **memcpy()**.

  Το πρόγραμμα να εμφανίζει τα περιεχόμενα του δεύτερου πίνακα και να τερματίζει.

```
#include <stdio.h>
#include <string.h>
int main()
{
        int i,arr1[100000],arr2[100000];

        for(i = 0; i < 100000; i++)
                arr1[i] = i+1;

        memcpy(arr2,arr1,sizeof(arr1));

        for(i = 0; i < 100000; i++)
                printf("%d\n",arr2[i]);
        return 0;
}
```

Εναλλακτικά:

```
for(i = 0; i < 100000; i++)
        arr2[i] = arr1[i];
```

Η συνάρτηση memcmp()

- Η συνάρτηση memcmp() χρησιμοποιείται για τη σύγκριση οποιουδήποτε τύπου δεδομένων που περιέχονται σε μία περιοχή μνήμης με τα δεδομένα που περιέχονται σε μία άλλη περιοχή μνήμης

- Το πρωτότυπό της δηλώνεται στο αρχείο string.h και είναι το ακόλουθο:

```
int memcmp(void* ptr1,void* ptr2,unsigned int size);
```

- Η συνάρτηση memcmp() συγκρίνει size bytes από την περιοχή μνήμης στην οποία δείχνει ο δείκτης ptr1 με τα αντίστοιχα bytes που περιέχονται στην περιοχή μνήμης στην οποία δείχνει ο δείκτης ptr2

- Αν οι δύο περιοχές μνήμης περιέχουν τα ίδια δεδομένα, τότε η συνάρτηση memcmp() επιστρέφει 0

Παρατηρήσεις

- Η συνάρτηση `memcmp()` είναι πολύ χρήσιμη γιατί μπορούμε να συγκρίνουμε άμεσα μεγάλο όγκο δεδομένων που περιέχονται σε δύο περιοχές μνήμης

- Για παράδειγμα, αν θέλετε να συγκρίνετε τα περιεχόμενα ενός πίνακα `100000` ακεραίων με τα περιεχόμενα ενός άλλου πίνακα, τότε να μην χρησιμοποιήσετε `for` βρόχο, αλλά να χρησιμοποιήσετε τη `memcmp()`, γιατί η σύγκριση θα γίνει πολύ πιο γρήγορα

Παράδειγμα

- Γράψτε ένα πρόγραμμα το οποίο να ορίζει δύο πίνακες 100000 ακεραίων, να θέτει τις τιμές από 1 έως 100000 στα στοιχεία τους και να συγκρίνει τις τιμές των στοιχείων τους με χρήση της συνάρτησης **memcmp()**.

  Το πρόγραμμα να εμφανίζει ένα διαγνωστικό μήνυμα για το αποτέλεσμα της σύγκρισης.

```
#include <stdio.h>
#include <string.h>
int main()
{
      int i,arr1[100000],arr2[100000];

      for(i = 0; i < 100000; i++)
            arr1[i] = arr2[i] = i+1;

      if(memcmp(arr1,arr2,sizeof(arr1)) == 0)
            printf("The same content\n");
      else
            printf("Different content\n");

      return 0;
}
```

Εναλλακτικά:

```
for(i = 0; i < 100000; i++)
      if(arr2[i] != arr1[i])
      {
        printf("Different content\n");
        break;
      }
```

Στατικές Δομές Δεδομένων

- Οι **δομές δεδομένων** χρησιμοποιούνται για την αποθήκευση και επεξεργασία πληθώρας δεδομένων με εύκολο και γρήγορο τρόπο
- Π.χ. ο **πίνακας** είναι μία **δομή δεδομένων**, ο οποίος χρησιμοποιείται για την αποθήκευση δεδομένων ίδιου τύπου
- Παρομοίως, οι **δομές** (**structs**) και οι **ενώσεις** (**unions**) είναι **δομές δεδομένων**, οι οποίες χρησιμοποιούνται για την αποθήκευση δεδομένων οποιουδήποτε τύπου
- **Αυτές οι δομές δεδομένων ονομάζονται στατικές, γιατί το μέγεθος της μνήμης που έχει δεσμευτεί για αυτές δεν μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος**
- Π.χ. όταν δηλώνεται έναν πίνακας με μία συγκεκριμένη διάσταση (π.χ. `int arr[100]`), η διάστασή του, δηλαδή το 100, δεν μπορεί να αλλάξει κατά την εκτέλεση του προγράμματος

Δυναμικές Δομές Δεδομένων

- Υπάρχουν όμως περιπτώσεις που αντί να χρησιμοποιήσουμε μία **στατική** δομή δεδομένων, όπως είναι ο πίνακας, να είναι πιο αποδοτικό να χρησιμοποιήσουμε μία **δυναμική δομή δεδομένων**

- Αντίθετα με τη στατική δομή, **το μέγεθος** μίας δυναμικής δομής δεδομένων **μπορεί να αυξομειώνεται κατά την εκτέλεση του προγράμματος** με τη δέσμευση και την αποδέσμευση αντίστοιχης μνήμης

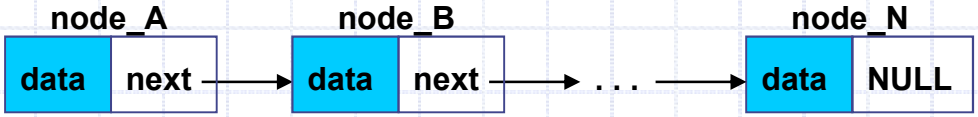
- Τα παραδείγματα των δυναμικών δομών δεδομένων που θα περιγράψουμε είναι η **απλά συνδεδεμένη λίστα**, η **ουρά** και η **στοίβα**

- Μία **δυναμική δομή δεδομένων** αποτελείται από ένα ή περισσότερα **συνδεδεμένα στοιχεία**, τα οποία ονομάζονται **κόμβοι**

- Κάθε **κόμβος** είναι μία **δομή**, η οποία περιέχει **τα δεδομένα για το συγκεκριμένο στιγμιότυπο της δομής** και **δείκτες** που χρησιμοποιούνται για τη σύνδεση με τον επόμενο ή τον προηγούμενο κόμβο

Απλά συνδεδεμένη λίστα

- Η πιο συνηθισμένη δυναμική δομή δεδομένων είναι μία απλά συνδεδεμένη λίστα
- Στο σχήμα φαίνεται ότι κάθε κόμβος μίας τέτοιας λίστας περιέχει τα δεδομένα του κόμβου (π.χ. το πεδίο `data`) και έναν δείκτη (π.χ. το πεδίο `next`) που «δείχνει» στον επόμενο κόμβο



- Ο πρώτος κόμβος της λίστας ονομάζεται κεφαλή (head) της λίστας και ο τελευταίος κόμβος ονομάζεται ουρά (tail)
- Το πεδίο-δείκτης του τελευταίου κόμβου πρέπει να έχει την τιμή `NULL`, ώστε να προσδιορίζεται το τέλος της λίστας
- Ο χειρισμός μίας απλά συνδεδεμένης λίστας γίνεται συνήθως με τη χρήση δύο δεικτών, με τον πρώτο να δείχνει στη διεύθυνση μνήμης της κεφαλής της λίστας και τον δεύτερο στη διεύθυνση μνήμης της ουράς της λίστας

Εισαγωγή κόμβου σε απλά συνδεδεμένη λίστα (Ι)

- Για να εισάγουμε έναν νέο κόμβο στη λίστα, εξετάζουμε τις ακόλουθες περιπτώσεις:

  1) Αν η λίστα είναι κενή (δηλ. δεν περιέχει κανέναν κόμβο) τότε ο κόμβος εισάγεται στη λίστα και αποτελεί ταυτόχρονα την κεφαλή και την ουρά της λίστας, ενώ η τιμή του δείκτη του γίνεται NULL

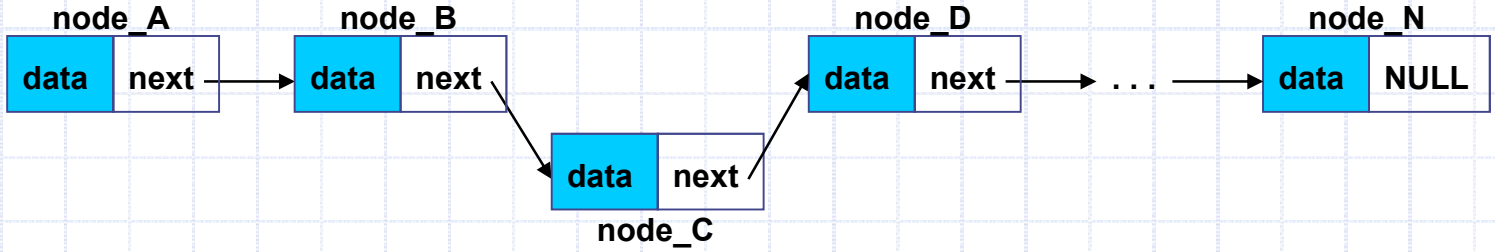
  2) Αν η λίστα δεν είναι κενή τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

     2α) Αν επιθυμούμε ο νέος κόμβος να τοποθετηθεί στην αρχή της λίστας, τότε ο νέος κόμβος γίνεται η νέα κεφαλή της λίστας και ο δείκτης του δείχνει στην παλιά κεφαλή, που τώρα γίνεται ο δεύτερος κόμβος της λίστας

Εισαγωγή κόμβου σε απλά συνδεδεμένη λίστα (ΙΙ)

2β) Αν επιθυμούμε ο νέος κόμβος να τοποθετηθεί στο **τέλος** της λίστας, τότε ο νέος κόμβος γίνεται η **νέα ουρά** της λίστας και η τιμή του δείκτη του γίνεται **NULL**, ενώ ο κόμβος που ήταν προηγουμένως η ουρά της λίστας γίνεται ο προτελευταίος κόμβος της λίστας με την τιμή του δείκτη του να αλλάζει από **NULL** και να δείχνει στον νέο κόμβο

2γ) Για να εισάγουμε έναν νέο κόμβο **μετά** από κάποιον τυχαίο κόμβο μίας λίστας, τότε κάνουμε τον δείκτη αυτού του τυχαίου κόμβου να δείχνει στον νέο κόμβο και τον δείκτη του νέου κόμβου να δείχνει στον κόμβο που έδειχνε ο τυχαίος κόμβος (όπως φαίνεται στο σχήμα, με τον κόμβο C να εισάγεται μεταξύ των κόμβων B και D)



Διαγραφή κόμβου από απλά συνδεδεμένη λίστα (Ι)

- Για να διαγράψουμε έναν κόμβο από μία λίστα, εξετάζουμε τις ακόλουθες περιπτώσεις:

1) Αν επιθυμούμε να διαγράψουμε τον κόμβο που είναι η **αρχή** της λίστας, τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

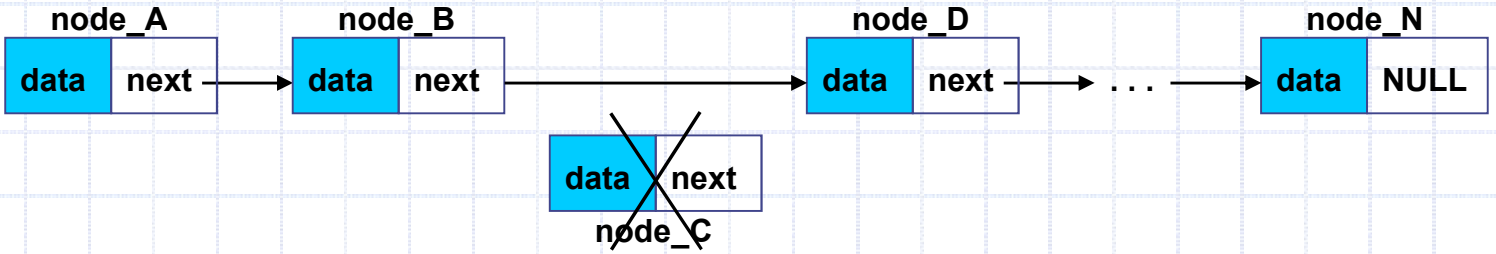
1α) Αν υπάρχει επόμενος κόμβος, τότε αυτός ο κόμβος γίνεται η **νέα κεφαλή** της λίστας

1β) Αν δεν υπάρχει επόμενος κόμβος, τότε η λίστα γίνεται **κενή**

2) Αν επιθυμούμε να διαγράψουμε τον **τελευταίο** κόμβο της λίστας, τότε **αποδεσμεύουμε τη μνήμη** που καταλαμβάνει, ενώ ο προτελευταίος κόμβος γίνεται η **νέα ουρά** της λίστας και η τιμή του δείκτη του γίνεται **NULL**

Διαγραφή κόμβου από απλά συνδεδεμένη λίστα (ΙΙ)

3) Για να διαγράψουμε ένα κόμβο που βρίσκεται ανάμεσα σε δύο κόμβους μίας λίστας, τότε κάνουμε τον δείκτη του προηγούμενου κόμβου από αυτόν που θέλουμε να διαγράψουμε να δείχνει στον επόμενο κόμβο από αυτόν που θέλουμε να διαγράψουμε και αποδεσμεύουμε τη μνήμη που καταλαμβάνει (όπως φαίνεται στο σχήμα, όπου διαγράφεται ο κόμβος C)



Παρατηρήσεις

- Το **βασικό μειονέκτημα** μίας λίστας, σε σχέση με τους πίνακες, είναι ότι για να βρούμε κάποιο στοιχείο της λίστας **πρέπει να διατρέξουμε όλη τη λίστα** ξεκινώντας από την αρχή της, ενώ με τους πίνακες μπορούμε να έχουμε **άμεση πρόσβαση** στο επιθυμητό στοιχείο

- Η **διαχείριση** μίας λίστας μέσα σε ένα πρόγραμμα γίνεται με τη **χρήση μίας μεταβλητής-δείκτη**

  - Ο δείκτης αυτός **δείχνει πάντα στον πρώτο** κόμβο της λίστας

  - Επίσης, μπορεί να χρησιμοποιηθεί και μία **δεύτερη μεταβλητή-δείκτης** που να **δείχνει πάντα στο τέλος** της λίστας

Στοίβα (stack)

- Η **στοίβα** (**stack**) είναι μία ειδική περίπτωση λίστας, γιατί ισχύουν οι ακόλουθοι περιορισμοί:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται **μόνο** στην **αρχή** της στοίβας, δηλαδή, ο νέος αυτός κόμβος γίνεται η νέα κεφαλή της στοίβας

  2) Ο **μόνος** κόμβος που επιτρέπεται να διαγράψουμε από τη στοίβα είναι η **κεφαλή** της στοίβας

- Μία τέτοια στοίβα ονομάζεται **LIFO** (**L**ast **I**n **F**irst **O**ut), με την έννοια ότι ο κόμβος που **εισάγεται τελευταίος εξάγεται πρώτος**

Ουρά (queue)

- Η **ουρά** (**queue**) είναι μία ειδική περίπτωση λίστας, γιατί ισχύουν οι ακόλουθοι περιορισμοί:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται **μόνο** στο **τέλος** της ουράς, δηλαδή, ο νέος αυτός κόμβος γίνεται ο τελευταίος κόμβος της ουράς

  2) Ο **μόνος** κόμβος που επιτρέπεται να διαγράψουμε από την ουρά είναι η **κεφαλή** της ουράς

- Μία τέτοια ουρά ονομάζεται **FIFO** (**F**irst **I**n **F**irst **O**ut), με την έννοια ότι ο κόμβος που **εισάγεται πρώτος εξάγεται και πρώτος**