Προγραμματισμός Υπολογιστών C++

Διαχείριση Μνήμης και
Δομές Δεδομένων

Διαχείριση Μνήμης

- Η διαχείριση της μνήμης αφορά κυρίως τις διαδικασίες **δέσμευσης** της μνήμης και **αποδέσμευσης** αυτής (όταν πλέον δεν μας χρειάζεται)
- Η δέσμευση μνήμης μπορεί να γίνει με δύο τρόπους:
  - είτε **στατικά**
  - είτε **δυναμικά**
- Στα προγράμματα μέχρι τώρα, έχουμε χρησιμοποιήσει μόνο τον στατικό τρόπο δέσμευσης μνήμης. Ένα παράδειγμα στατικής δέσμευσης είναι ο πίνακας, όπου τα στοιχεία του αποθηκεύονται σε μνήμη που το μέγεθός της δεν μπορεί να μεταβληθεί
- Σε αυτή την ενότητα, θα δούμε με ποιον τρόπο μπορούμε να δεσμεύουμε δυναμικά μνήμη κατά την εκτέλεση του προγράμματος (run time) για τη δημιουργία δυναμικών δομών δεδομένων, όπως στοίβες, ουρές, συνδεδεμένες λίστες και δέντρα

Κατανομή Μνήμης

- Όταν ένα πρόγραμμα εκτελείται, ζητάει πόρους μνήμης από το λειτουργικό σύστημα του υπολογιστή και, συνήθως, η μνήμη που αποδίδεται χωρίζεται στα ακόλουθα τέσσερα τμήματα:

  1) Στο τμήμα κώδικα (*code segment*), το οποίο χρησιμοποιείται για την αποθήκευση του μεταγλωττισμένου κώδικα του προγράμματος

  2) Στο τμήμα δεδομένων (*data segment*), το οποίο χρησιμοποιείται για την αποθήκευση των καθολικών και `static` μεταβλητών του προγράμματος

  3) Στο τμήμα που ονομάζεται στοίβα (*stack segment*), το οποίο χρησιμοποιείται για την αποθήκευση των δεδομένων των συναρτήσεων (π.χ. τοπικές μεταβλητές)

  4) Στο τμήμα που ονομάζεται σωρός (*heap*), το οποίο χρησιμοποιείται για δυναμική δέσμευση μνήμης

Παρατηρήσεις

- **Σημειώστε ότι το παραπάνω μοντέλο κατανομής μνήμης αποτελεί μία συνήθης επιλογή, το κάθε σύστημα μπορεί να καθορίσει το δικό του μοντέλο**

- **Ο κάθε μεταγλωττιστής μπορεί να εφαρμόσει τη δική του πολιτική βελτιστοποίησης, π.χ., οι παράμετροι μίας συνάρτησης μπορεί να μην αποθηκεύονται στη στοίβα, αλλά σε καταχωρητές (registers) του συστήματος, για πιο γρήγορη πρόσβαση**

Στατική Δέσμευση Μνήμης (1)

- Με τη στατική δέσμευση, η μνήμη δεσμεύεται από τη στοίβα
- Το μέγεθος της μνήμης που θα δεσμευτεί πρέπει να είναι γνωστό κατά τη συγγραφή του προγράμματος και δεν μπορεί να αλλάξει κατά τη διάρκεια εκτέλεσής του
- Για παράδειγμα, με τη δήλωση:

```
float grades[1000];
```

  ο μεταγλωττιστής, όταν μεταγλωττίζει το πρόγραμμα, δεσμεύει στατικά μνήμη `1000×sizeof(float)` bytes για την αποθήκευση των βαθμών 1000 φοιτητών
- Το μέγεθος του πίνακα `grades` δεν μπορεί να αλλάξει κατά τη διάρκεια εκτέλεσης του προγράμματος, ακόμα και αν χρειαστεί να αποθηκευτούν βαθμοί για περισσότερους φοιτητές
- Επίσης, αν οι φοιτητές είναι λιγότεροι από 1000, τότε γίνεται σπατάλη μνήμης, αφού δεσμεύεται περισσότερη απ' την απαιτούμενη
- Ο μοναδικός τρόπος για να αλλάξει το μέγεθος του πίνακα είναι να ξαναγραφεί το πρόγραμμα και να γίνει νέα μεταγλώττιση

Στατική Δέσμευση Μνήμης (2)

Τι συμβαίνει όταν γίνεται κλήση μίας συνάρτησης;

- Όταν καλείται μία συνάρτηση, ο μεταγλωττιστής δεσμεύει μνήμη στη στοίβα για να αποθηκευτούν τα δεδομένα της συνάρτησης, θεωρώντας – για απλότητα – ότι δεν χρησιμοποιούνται και καταχωρητές του συστήματος, παρόλο που κάτι τέτοιο είναι δυνατό

- Π.χ., όταν καλείται μία συνάρτηση που επιστρέφει τιμή, δέχεται παραμέτρους και χρησιμοποιεί τοπικές μεταβλητές, ο μεταγλωττιστής δεσμεύει μνήμη για να αποθηκεύσει:
  - τις τιμές των παραμέτρων
  - τις τοπικές μεταβλητές
  - την τιμή επιστροφής
  - τη διεύθυνση μνήμης της εντολής που θα εκτελεστεί μετά τον τερματισμό της συνάρτησης

Στατική Δέσμευση Μνήμης (3)

<u>Τι συμβαίνει όταν τερματίζει μία συνάρτηση;</u>

- Όταν τερματίσει η συνάρτηση (έστω η συνάρτηση του περιγράφηκε προηγουμένως) συμβαίνουν τα ακόλουθα:
  - Αν η τιμή επιστροφής της συνάρτησης εκχωρείται σε κάποια μεταβλητή, εξάγεται από τη στοίβα και αποθηκεύεται σε αυτήν
  - Η διεύθυνση μνήμης της επόμενης εντολής εξάγεται επίσης από τη στοίβα, ώστε το πρόγραμμα να συνεχίσει με την εκτέλεσή της
  - Η μνήμη που δεσμεύτηκε για την αποθήκευση των δεδομένων της συνάρτησης αποδεσμεύεται, εκτός από τη μνήμη για τις στατικές μεταβλητές

Παράδειγμα

```
#include <iostream>

void test(int i, int j);

int main()
{
    float a[500], b[10];
    test(10, 20);
    return 0;
}

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

- Όταν καλείται η `test()` ο μεταγλωττιστής δεσμεύει 202×`sizeof(int)` bytes στη στοίβα, για να αποθηκευτούν οι τιμές των παραμέτρων `i` και `j` και των στοιχείων του πίνακα `arr`
- Όταν τερματίζει η εκτέλεση της `test()`, η μνήμη αυτή αποδεσμεύεται
- Παρομοίως, η μνήμη των 510×`sizeof(float)` bytes που έχει δεσμευτεί για τις τοπικές μεταβλητές της `main()` αποδεσμεύεται, όταν τερματιστεί η εκτέλεση του προγράμματος

Παρατηρήσεις

- Αν δεν υπάρχει διαθέσιμος χώρος στη στοίβα για την αποθήκευση των δεδομένων μίας συνάρτησης, η εκτέλεση του προγράμματος **θα τερματιστεί ανώμαλα** και είναι πιθανό να εμφανιστεί το μήνυμα "Stack overflow" *(υπερχείλιση στοίβας)*

- Κάτι τέτοιο μπορεί να συμβεί αν μία συνάρτηση δεσμεύει αρκετή μνήμη και καλεί άλλες ένθετες συναρτήσεις, οι οποίες επίσης δεσμεύουν αρκετή μνήμη

- Για παράδειγμα, μία **αναδρομική συνάρτηση** που καλεί πολλές φορές τον εαυτό της μπορεί επίσης να οδηγήσει σε εξάντληση της διαθέσιμης μνήμης της στοίβας

Δυναμική Δέσμευση Μνήμης (1)

- Με τη δυναμική δέσμευση, η μνήμη δεσμεύεται από τον σωρό (heap) δυναμικά, δηλαδή κατά την εκτέλεση του προγράμματος
- Σε αντίθεση με τη στατική δέσμευση, το μέγεθος της μνήμης που θα δεσμευτεί δεν χρειάζεται να είναι γνωστό πριν την εκτέλεση του προγράμματος (δηλ. κατά τη μεταγλώττισή του), αλλά μπορεί να καθοριστεί δυναμικά κατά την εκτέλεσή του
- Επίσης, το μέγεθος της μνήμης που δεσμεύεται με δυναμικό τρόπο μπορεί να μεγαλώσει ή να μικρύνει δυναμικά, ανάλογα με τις απαιτήσεις του προγράμματος
- Για παράδειγμα, η κλάση `vector` δεσμεύει δυναμικά τη μνήμη που χρειάζεται για να αποθηκεύσει τα στοιχεία της
- Η δυναμική δέσμευση είναι πολύ συνηθισμένη σε εφαρμογές όπου ο χρήστης επικοινωνεί με το πρόγραμμα. Για παράδειγμα, όταν το πρόγραμμα δεν γνωρίζει τον αριθμό των φοιτητών, να ζητάει από τον χρήστη να τον εισάγει, και μετά να δεσμεύει μνήμη για έναν πίνακα δομών, στον οποίο να αποθηκεύονται τα στοιχεία τους

Δυναμική Δέσμευση Μνήμης (2)

- Όπως είπαμε, με την στατική δέσμευση η μνήμη δεσμεύεται από την στοίβα. Συνήθως, το προκαθορισμένο μέγεθος της στοίβας δεν είναι μεγάλο. Για παράδειγμα, το επόμενο πρόγραμμα μπορεί να μην εκτελεστεί λόγω έλλειψης διαθέσιμης μνήμης στη στοίβα

```
#include <iostream>
int main()
{
        int arr[10000000]; // Στατική δέσμευση μνήμης
        return 0;
}
```

- Αντιθέτως, το μέγεθος του σωρού είναι συνήθως αρκετά μεγαλύτερο από αυτό της στοίβας. Για παράδειγμα, αν στο προηγούμενο πρόγραμμα γινόταν δυναμική δέσμευση μνήμης (όπως παρακάτω) η απαιτούμενη μνήμη θα δεσμευτεί από τον σωρό και το πρόγραμμα θα εκτελεστεί χωρίς πρόβλημα

```
#include <iostream>
int main()
{
        int *arr;
        arr = new int[10000000]; // Δυναμική δέσμευση μνήμης
        delete[] arr; // Αποδέσμευση μνήμης
        return 0;
}
```

- Η διαχείριση δυναμικά δεσμευμένης μνήμης συχνά προκαλεί την εμφάνιση σοβαρών λαθών κατά την εκτέλεση του προγράμματος. Για παράδειγμα, όπως θα δούμε στη συνέχεια, η διαρροή μνήμης και η προσπέλαση μνήμης που έχει ήδη αποδεσμευτεί αποτελούν πολύ συνηθισμένα **λάθη**. Χρειάζεται λοιπόν ιδιαίτερη προσοχή στον τρόπο χρήσης της.

Ο Τελεστής new

- Με τον τελεστή new επιλέγουμε τον τύπο για τον οποίον θέλουμε να δεσμεύσουμε μνήμη, ο new βρίσκει την αντίστοιχη μνήμη, τη δεσμεύει, επιστρέφει τη διεύθυνση της, την οποία και εκχωρούμε σε αντίστοιχο δείκτη για να τη διαχειριστούμε

- Για παράδειγμα, ο παρακάτω κώδικας δεσμεύει μνήμη για έναν ακέραιο, διαβάζει μία τιμή και την αποθηκεύει στη μνήμη:

```
int *p;
p = new int;
cin >> *p; /* Η τιμή του *p θα γίνει ίση με την τιμή
που θα εισάγει ο χρήστης. */
```

- Γενικά, δεν υπάρχει λόγος να χρησιμοποιήσουμε τον τελεστή new για να δεσμεύσουμε ένα στοιχείο (π.χ. έναν ακέραιο) αφού μπορούμε να δηλώσουμε μία αντίστοιχη μεταβλητή (π.χ. int i) και να χρησιμοποιήσουμε τη μεταβλητή

Ο Τελεστής new[]

- Για τη δέσμευση μνήμης για πολλά στοιχεία χρησιμοποιούμε τον τελεστή new[], όπου ο αριθμός των στοιχείων δηλώνεται μετά τον τύπο τους μέσα σε αγκύλες
- Αυτός ο αριθμός μπορεί να καθοριστεί και κατά την εκτέλεση του προγράμματος. Π.χ:

```
int num;
cin >> num;
int *p = new int[num];
```

- Ο τελεστής new[] δεσμεύει ένα τμήμα μνήμης για num ακεραίους και, αν η δέσμευση είναι επιτυχημένη, ο p δείχνει στην αρχή αυτής της μνήμης. Ο αριθμός στις [] πρέπει να είναι ακέραιος
- Οι αρχικές τιμές των στοιχείων είναι τυχαίες. Αν θέλουμε να τις αρχικοποιήσουμε με 0 προσθέτουμε κενές παρενθέσεις ή άγκιστρα. Π.χ. `int *p = new int[100]();`
- Με τη C++11 μπορούμε να προσδιορίσουμε μία λίστα αρχικών τιμών. Π.χ: `int *p = new int[100]{10, 20, 30};` Τα τρία πρώτα στοιχεία αρχικοποιούνται με τις αντίστοιχες τιμές και τα υπόλοιπα με 0
- Παρόμοια, για να δεσμεύσουμε μνήμη για έναν πίνακα 100 δομών τύπου Student γράφουμε: `Student *p = new Student[100];`

Παρατηρήσεις (1)

- Σημειώστε ότι ο μεταγλωττιστής δεν μας υποχρεώνει να προσπελαύνουμε με τον δείκτη μόνο τα στοιχεία για τα οποία έχει δεσμευτεί μνήμη

- Π.χ., αν ο δείκτης **`p`** δείχνει σε μία μνήμη που δεσμεύσαμε για 100 ακεραίους ο μεταγλωττιστής μας επιτρέπει να γράψουμε:

  `p[200] = 10;` ή ακόμα και **`p[-10] = 20;`**

  και να προσπελάσουμε μνήμη εκτός του έγκυρου εύρους με καταστροφικές συνέπειες για τη λειτουργία του προγράμματος. Ο μεταγλωττιστής δεν θα απαγορεύσει αυτήν την ενέργεια. Μας εμπιστεύεται ότι ξέρουμε τι κάνουμε και θα μας αφήσει να προσπελάσουμε αυτή τη μνήμη

- Αυτό το είδος σφάλματος είναι ιδιαίτερα δύσκολο να βρεθεί και το χειρότερο είναι ότι, κάθε φορά που εκτελείται το πρόγραμμα, μπορεί να έχει διαφορετική λανθασμένη συμπεριφορά, γεγονός που καθιστά ακόμα πιο δύσκολο τον εντοπισμό του σφάλματος

Παρατηρήσεις (2)

- Μην ξεχνάμε, ότι όπως έχουμε δει σε παραδείγματα, μπορούμε αντί για `new[]` να χρησιμοποιήσουμε ένα `vector` αντικείμενο. Τότε, μπορούμε να προσπελάσουμε με ασφάλεια τα στοιχεία του διανύσματος και να μην ασχολούμαστε με την δέσμευση και αποδέσμευση μνήμης. Π.χ:

```
int num;
cin >> num;
vector<int> v(num);
```

Παρατηρήσεις (3)

- Είναι προτιμότερο να μην δεσμεύετε δυναμικά μνήμη για τη δημιουργία τοπικών μεταβλητών, εφόσον μπορείτε να το κάνετε με στατικό τρόπο. Π.χ. ο παρακάτω κώδικας είναι επιρρεπής σε σφάλματα:

```
void f()
{
        int *p = new int[1000];
        ...
        delete[] p;
}
```

- Π.χ., προσθέτοντας ο προγραμματιστής μία εντολή **return** και ξεχνώντας να καλέσει πρώτα την **delete[]** για να αποδεσμεύσει τη μνήμη, αν εκτελεστεί αυτή η **return**, θα προκληθεί διαρροή μνήμης
- Είναι πολύ πιο ασφαλές να χρησιμοποιήσετε μία συνηθισμένη μεταβλητή (π.χ. `int p[1000]`), για την οποία η μνήμη θα αποδεσμευτεί αυτόματα όταν τερματιστεί η συνάρτηση
- Γενικά, σε ένα πρόγραμμα που εκτελείται για ένα μεγάλο χρονικό διάστημα ή και για πάντα, οι επαναλαμβανόμενες διαρροές μνήμης μπορεί να προκαλέσουν σοβαρή υποβάθμιση της απόδοσης και πιθανώς την κατάρρευση του προγράμματος

Ο Τελεστής delete

- Για να αποδεσμεύσουμε τη μνήμη που έχει δεσμευτεί για ένα στοιχείο με τον τελεστή new χρησιμοποιούμε τον τελεστή delete. Π.χ:

```
int *p = new int;
...
delete p;
```

- Η απόπειρα αποδέσμευσης μνήμης που έχει ήδη αποδεσμευτεί ή η προσπέλαση μνήμης που έχει ήδη αποδεσμευτεί είναι λανθασμένη ενέργεια. Π.χ:

```
int *p = new int;
...
delete p;
*p = 10; // Λάθος
delete p; // Λάθος
```

- Αν ο δείκτης είναι μηδενικός, είναι ασφαλές να χρησιμοποιήσουμε τον delete. Αφού ο μηδενικός δείκτης δεν δείχνει κάπου, η εντολή δεν έχει καμία επίδραση, τίποτα δεν θα συμβεί. Π.χ:

```
int *p = nullptr;
delete p; // Σωστό
```

Ο Τελεστής delete[]

- Για να αποδεσμεύσουμε τη μνήμη που έχει δεσμευτεί για πολλά στοιχεία με τον τελεστή new[] χρησιμοποιούμε τον τελεστή delete[]. Π.χ:

```
Student *p = new Student[100];
...
delete[] p;
```

- Οι αγκύλες υποδεικνύουν στον μεταγλωττιστή ότι πρέπει να αποδεσμευτεί όλη η μνήμη και όχι μόνο η μνήμη για το στοιχείο που δείχνει ο δείκτης. Αν ξεχάσουμε τις αγκύλες η συμπεριφορά του προγράμματος είναι απροσδιόριστη
- Θυμόμαστε, χρησιμοποιούμε τον delete για αποδέσμευση μνήμης που έχει δεσμευτεί με τον new και τον delete[] για αποδέσμευση μνήμης που έχει δεσμευτεί με τον new[]

Παρατηρήσεις

- Προσέξτε ότι όταν αποδεσμεύετε την μνήμη δεν σημαίνει ότι ο δείκτης θα γίνει μηδενικός. Για παράδειγμα, η τιμή του μπορεί να είναι ακόμα ίση με τη διεύθυνση της δεσμευμένης μνήμης. Γενικά, ιδιαίτερα σε μεγάλες εφαρμογές, όπου η ίδια μνήμη μπορεί να χρησιμοποιείται σε διαφορετικά σημεία του προγράμματος, συστήνεται, για μεγαλύτερη ασφάλεια και καλύτερο έλεγχο του προγράμματος, να εκχωρείτε την τιμή **nullptr** στον αντίστοιχο δείκτη όταν αποδεσμεύετε την μνήμη. Για παράδειγμα:

```
delete[] p; /* Αν είναι μηδενικός δείκτης δεν υπάρχει
πρόβλημα. */
p = nullptr;
```

Επίσης, αυτή η πρακτική είναι πολύ βοηθητική για την αποσφαλμάτωση προγραμμάτων με προβλήματα στη διαχείριση μνήμης

Διαρροές Μνήμης (1)

- Όταν δεν χρειάζεστε άλλο τη δεσμευμένη μνήμη, μην ξεχνάτε να την απελευθερώσετε, ώστε να μη δημιουργούνται διαρροές μνήμης που μπορεί να οδηγήσουν στην εξάντληση της μνήμης
- Ας δούμε ένα παράδειγμα διαρροής μνήμης:

```
void f()
{
      int *ptr = new int[10000];
      ...
      ptr = new int[200];
      ...
      delete[] ptr;
}
```

Σε αυτό το παράδειγμα ξεχνάμε να αποδεσμεύσουμε το πρώτο τμήμα μνήμης πριν από τη νέα δέσμευση. Αυτή είναι μια διαρροή μνήμης, η οποία προκαλείται κάθε φορά που καλείται η `f()`

- Όταν δεν χρειάζεστε άλλο τη δεσμευμένη μνήμη, μην ξεχνάτε να την απελευθερώσετε, ώστε να μη δημιουργούνται διαρροές μνήμης που μπορεί να οδηγήσουν στην εξάντληση της μνήμης

Διαρροές Μνήμης (2)

- Μία ακόμα συνηθισμένη περίπτωση διαρροής μνήμης συμβαίνει όταν εκχωρούμε ένα δείκτη σε κάποιον άλλο, ξεχνώντας όμως να αποδεσμεύσουμε τη μνήμη στην οποία αρχικά έδειχνε ο δείκτης. Για παράδειγμα, βρείτε τα λάθη στο παρακάτω πρόγραμμα και να τα διορθώσετε:

```
#include <iostream>
int main()
{
        char *p1, *p2;

        p1 = new char[10];
        p2 = new char[10];
        p1 = p2;
        delete[] p1;

        p2[0] = 'a';
        delete[] p2;
        return 0;
}
```

Διαρροές Μνήμης (2)

- Ας δούμε τα προβλήματα:

  α) Με την εντολή `p1 = p2;` οι δείκτες `p1` και `p2` δείχνουν στην ίδια μνήμη. Η μνήμη αυτή αποδεσμεύεται με τη `delete[] p1;`, επομένως, η επόμενη εντολή θέτει τιμή σε μη δεσμευμένη μνήμη.

  β) Η εντολή `delete[] p2;` αποδεσμεύει μνήμη που έχει ήδη αποδεσμευτεί

  γ) Η αρχική μνήμη στην οποία έδειχνε ο δείκτης `p1` δεν αποδεσμεύτηκε ποτέ

  Αν βάλουμε την εντολή `delete[] p1;` πριν από την `p1 = p2;` το πρόγραμμα λειτουργεί κανονικά

Συναρτήσεις Διαχείρισης Μνήμης

- Στη συνέχεια, θα περιγράψουμε συνοπτικά τις συναρτήσεις βιβλιοθήκης `memcpy()`, `memmove()` και `memcmp()` που χρησιμοποιούνται αρκετά συχνά για τη διαχείριση μνήμης

Η Συνάρτηση memcpy()

- Η συνάρτηση memcpy() χρησιμοποιείται για την αντιγραφή οποιουδήποτε τύπου δεδομένων από μία περιοχή μνήμης σε μία άλλη
- Το πρωτότυπό της δηλώνεται στο αρχείο cstring ως εξής:

```
void *memcpy(void *dest, const void *src, size_t size);
```

- Η memcpy() αντιγράφει size bytes από την περιοχή μνήμης στην οποία δείχνει ο δείκτης src στην περιοχή μνήμης στην οποία δείχνει ο δείκτης dest
- Αν οι περιοχές μνημών επικαλύπτονται, η συμπεριφορά της συνάρτησης είναι ακαθόριστη

Παρατηρήσεις

- Όταν η `memcpy()` χρησιμοποιείται για την αντιγραφή αλφαριθμητικών μοιάζει με τη `strcpy()` με την εξής όμως διαφορά:
  - Με την `strcpy()` η αντιγραφή αλφαριθμητικών ολοκληρώνεται όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)
  - Αντίθετα, η `memcpy()` συνεχίζει μέχρι να αντιγραφεί ο αριθμός των καθορισμένων οκτάδων

Π.χ. αν έχουμε:

```
char str2[6] = {0}, str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
```

και γράψουμε: `memcpy(str2, str1, 6);`

το περιεχόμενο του πίνακα `str2` θα γίνει το ίδιο με του `str1`

Αντίθετα, αν γράψουμε: `strcpy(str2, str1);`

το περιεχόμενο του `str2` θα γίνει ίσο με

```
{'a', 'b', 'c', '\0', '\0', '\0'}
```

Η Συνάρτηση memmove()

- Η συνάρτηση memmove() είναι παρόμοια με την memcpy(), με τη διαφορά ότι η memmove() εξασφαλίζει τη σωστή αντιγραφή των δεδομένων, ακόμα και αν οι δύο περιοχές μνήμης επικαλύπτονται

- Επειδή ακριβώς η memcpy() δεν ελέγχει αν οι δύο περιοχές επικαλύπτονται, εκτελείται πιο γρήγορα από τη memmove()

Παρατηρήσεις

- Κατά τη χρήση της memcpy() ή της memmove(), για τη μνήμη προορισμού πρέπει να έχουν δεσμευτεί τουλάχιστον size bytes, σε διαφορετική περίπτωση, τα πλεονάζοντα bytes θα εγγραφούν σε μη δεσμευμένη μνήμη και τα δεδομένα εκεί θα υπερεγγραφούν

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

- Η memcpy() είναι πολύ χρήσιμη, γιατί συνήθως υλοποιείται με τέτοιο τρόπο ώστε η αντιγραφή μεγάλου όγκου δεδομένων από μία περιοχή μνήμης σε μία άλλη να ολοκληρώνεται πιο γρήγορα από ότι με έναν επαναληπτικό βρόχο

- Για παράδειγμα, ο επόμενος κώδικας χρησιμοποιεί τη memcpy() για την αντιγραφή ενός πίνακα σε έναν άλλο:

```
int arr1[SIZE], arr2[SIZE];
...
memcpy(arr2, arr1, sizeof(arr1)); /* Χρησιμοποιούμε τη
memcpy() αντί για έναν επαναληπτικό βρόχο, όπως:
for(int i = 0; i < SIZE; i++)
      arr2[i] = arr1[i]; */
```

Η Συνάρτηση memcmp()

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

- Το πρωτότυπό της δηλώνεται στο αρχείο cstring ως εξής:

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

- Η 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 <u>περιέχουν τα ίδια δεδομένα</u>, τότε η συνάρτηση memcmp() επιστρέφει 0, αλλιώς μια μη μηδενική τιμή, όπως κάνει και η strcmp()

Παρατηρήσεις

- Όταν η `memcmp()` χρησιμοποιείται για τη σύγκριση αλφαριθμητικών μοιάζει με τη `strcmp()` με την εξής όμως διαφορά:
  - Με την `strcmp()` η σύγκριση αλφαριθμητικών ολοκληρώνεται όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)
  - Αντίθετα, η `memcmp()` δεν σταματάει τη σύγκριση όταν συναντηθεί ο τερματικός χαρακτήρας (`'\0'`)

Παράδειγμα (1)

```
#include <iostream>
#include <cstring>
using std::cout;
int main()
{
    char str1[] = {'a', 'b', 'c', '\0', 'd', 'e'};
    char str2[] = {'a', 'b', 'c', '\0', 'd', 'f'};

    if(strcmp(str1, str2) == 0)
            cout << "Same\n";
    else
            cout << "Different\n";

    if(memcmp(str1, str2, sizeof(str1)) == 0)
            cout << "Same\n";
    else
            cout << "Different\n";
    return 0;
}
```

Επειδή η `strcmp()` σταματάει τη σύγκριση όταν συναντήσει τον τερματικό χαρακτήρα, το πρόγραμμα θα εμφανίσει `Same`. Αντίθετα, η `memcmp()` συγκρίνει όλες τις οκτάδες και το πρόγραμμα θα εμφανίσει `Different`

Παράδειγμα (2)

- Δημιουργήστε μία εκδοχή της συνάρτησης memcmp(). Να γραφεί ένα πρόγραμμα το οποίο να διαβάζει δύο αλφαριθμητικά μέχρι 100 χαρακτήρες, τον αριθμό των χαρακτήρων που θα συγκριθούν και να εμφανίζει το αποτέλεσμα της σύγκρισής τους με χρήση της συνάρτησης

Παράδειγμα (2)

```
#include <iostream>
#include <cstring>
using std::cout;
using std::cin;

int mem_cmp(const void *ptr1, const void *ptr2, size_t size);

int main()
{
	char str1[100], str2[100];
	int num;

	cout << "Enter first text: ";
	cin.getline(str1, sizeof(str1));

	cout << "Enter second text: ";
	cin.getline(str2, sizeof(str2));

	cout << "Enter characters to compare: ";
	cin >> num;

	cout << mem_cmp(str1, str2, num) << '\n';
	return 0;
}
int mem_cmp(const void *ptr1, const void *ptr2, size_t size)
{
	char *p1, *p2;

	p1 = (char*)ptr1;
	p2 = (char*)ptr2;
	while(size != 0)
	{
		if(*p1 != *p2)
			return *p1 - *p2;
		p1++;
		p2++;
		size--;
	}
	return 0;
```

Παράδειγμα (2)

- Σχόλια: Επειδή συγκρίνουμε χαρακτήρες, προσαρμόζουμε τον τύπο void* σε char*. Σε κάθε επανάληψη του βρόχου, η mem_cmp() συγκρίνει τους χαρακτήρες στους οποίους δείχνουν οι ptr1 και ptr2. Αν όλοι οι χαρακτήρες είναι ίδιοι, η mem_cmp() επιστρέφει 0, αλλιώς τη διαφορά των δύο πρώτων διαφορετικών χαρακτήρων

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 είναι **στατικές**, με την έννοια ότι η μνήμη που έχει δεσμευτεί για αυτές είναι στατική και **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

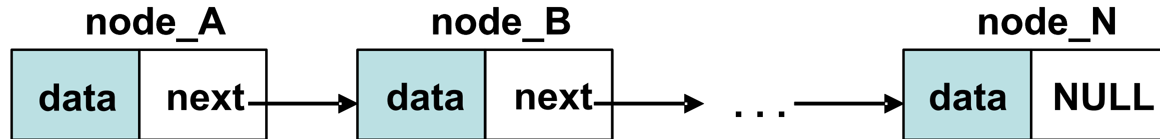
Δυναμικές Δομές Δεδομένων

- 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
- Μία δυναμική δομή δεδομένων αποτελείται από ένα ή περισσότερα συνδεδεμένα στοιχεία, τα οποία συνήθως ονομάζονται κόμβοι
- Κάθε κόμβος συνήθως αντιπροσωπεύεται από μία δομή, η οποία περιέχει τα δεδομένα του κόμβου κι ένα πεδίο που είναι δείκτης στον επόμενο κόμβο
- Στη συνέχεια θα περιγράψουμε την πιο συνηθισμένη δυναμική δομή δεδομένων, την απλά συνδεδεμένη λίστα καθώς και δύο συγκεκριμένες υποπεριπτώσεις αυτής, τη στοίβα και την ουρά

Απλά Συνδεδεμένη Λίστα

- Μία συνηθισμένη δυναμική δομή δεδομένων είναι η **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 `nullptr`, 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 (1)

- Για <u>να εισάγουμε έναν νέο κόμβο στη λίστα</u>, εξετάζουμε τις ακόλουθες περιπτώσεις:

  1) *Αν* η λίστα είναι <u>**κενή**</u> (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 `nullptr`, αφού δεν υπάρχει επόμενος κόμβος

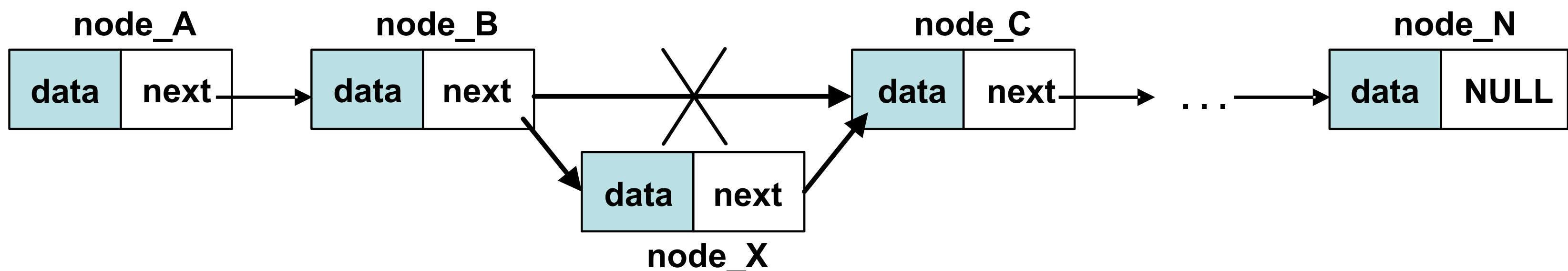
  2) *Αν* η λίστα <u>**δεν είναι κενή**</u> τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

    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 (2)

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 `nullptr`, 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 `nullptr` και να δείχνει στον νέο κόμβο

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



Διαγραφή Κόμβου από Απλά Συνδεδεμένη Λίστα (1)

- Για <u>να διαγράψουμε έναν κόμβο από μία λίστα</u>, 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**

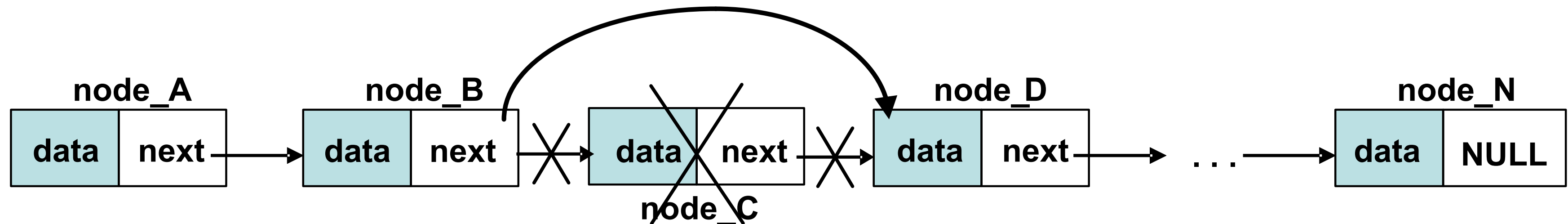
  2) Αν επιθυμούμε να διαγράψουμε τον **τελευταίο** κόμβο της λίστας, τότε εξετάζουμε τις ακόλουθες υπο-περιπτώσεις:

  2α) Αν υπάρχει προηγούμενος κόμβος, τότε αυτός ο κόμβος γίνεται η **νέα ουρά** της λίστας και η τιμή του δείκτη του γίνεται `nullptr`

  2β) Αν δεν υπάρχει προηγούμενος κόμβος, τότε η λίστα γίνεται **κενή**

Διαγραφή Κόμβου από Απλά Συνδεδεμένη Λίστα (2)

3) Για να διαγράψουμε ένα κόμβο που βρίσκεται ανάμεσα σε δύο κόμβους μίας λίστας, τότε κάνουμε τον δείκτη του προηγούμενου κόμβου 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

- Σε αντίθεση με τους πίνακες, που το μέγεθός τους δεν μπορεί να αλλάξει, μία συνδεδεμένη λίστα είναι πιο ευέλικτη γιατί το μέγεθός της μπορεί να μεταβληθεί δυναμικά, ανάλογα με τις ανάγκες του προγράμματος
- Επίσης, επειδή μπορούμε εύκολα να προσθέτουμε και να αφαιρούμε κόμβους οπουδήποτε στη λίστα, είναι πολύ πιο εύκολη η δημιουργία και διατήρηση μίας ταξινομημένης διάταξης
- Από την άλλη, η πρόσβαση σε ένα στοιχείο του πίνακα είναι γρήγορη και εύκολη, αφού ένας πίνακας υποστηρίζει τυχαία πρόσβαση χρησιμοποιώντας τη θέση του σαν δείκτη. Αντίθετα, για την πρόσβαση σε έναν κόμβο πρέπει να ξεκινήσουμε από την κεφαλή και να διασχίσουμε τη λίστα σειριακά
- Κατά συνέπεια, ο χρόνος πρόσβασης ενός κόμβου εξαρτάται από τη θέση του στη λίστα. Αν είναι στην αρχή της λίστας είναι μικρός, μεγαλύτερος αν είναι στο τέλος της

Στοίβα (stack)

- Η **στοίβα** (**stack**) αποτελεί μία ειδική περίπτωση λίστας, με τους ακόλουθους περιορισμούς:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται **μόνο** στην **αρχή** της στοίβας, δηλαδή, κάθε νέος κόμβος στη στοίβα 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**

- Μπορείτε να σκεφτείτε την υλοποίηση μίας στοίβας ως μία στοίβα από άπλυτα πιάτα, όπου το κάθε νέο (άπλυτο) πιάτο το τοποθετούμε στην κορυφή της στοίβας και το πιάτο που θέλουμε να πλύνουμε το αφαιρούμε από την κορυφή της

- Μπορείτε να δείτε ένα παράδειγμα υλοποίησης στοίβας στην αντίστοιχη ενότητα του βιβλίου

Ουρά (queue)

- Η ουρά (queue) αποτελεί μία ειδική περίπτωση λίστας, με τους ακόλουθους περιορισμούς:

  1) Η εισαγωγή ενός νέου κόμβου γίνεται μόνο στο τέλος της ουράς, δηλαδή, κάθε νέος κόμβος γίνεται η νέα «ουρά» της ουράς

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

- Μία τέτοια ουρά ονομάζεται FIFO (First In First Out), με την έννοια ότι ο κόμβος που εισάγεται πρώτος εξάγεται και πρώτος

- Μπορείτε να δείτε ένα παράδειγμα υλοποίησης ουράς στην αντίστοιχη ενότητα του βιβλίου

Δυαδικά Δέντρα Αναζήτησης (1)

- Για την οργάνωση των δεδομένων σε ιεραρχική διάταξη συνήθως χρησιμοποιείται μία ευέλικτη δομή δεδομένων που ονομάζεται δένδρο
- Η βασική έννοια του δένδρου μας είναι γνωστή από διάφορα παραδείγματα, όπως το οικογενειακό δένδρο, τα οργανογράμματα οργανισμών ή το σύστημα οργάνωσης φακέλων σε έναν υπολογιστή
- Υπάρχουν πολλοί τύποι δένδρων, εμείς θα παρουσιάσουμε τις βασικές λειτουργίες στο δυαδικό δένδρο αναζήτησης (binary search tree)
- Ένα δένδρο (με ρίζα) είναι μία συλλογή από κόμβους που συνδέονται με ακμές
- Υπάρχει ένας μόνο κόμβος στον οποίο δεν καταλήγει καμία ακμή και ονομάζεται ρίζα
- Στους υπόλοιπους κόμβους καταλήγει μία και μόνο ακμή
- Κάθε κόμβος, εκτός της ρίζας, συνδέεται με ακριβώς έναν κόμβο πάνω από αυτόν που ονομάζεται γονικός
- Οι κόμβοι με τους οποίους συνδέεται ακριβώς κάτω από αυτόν ονομάζονται παιδιά του

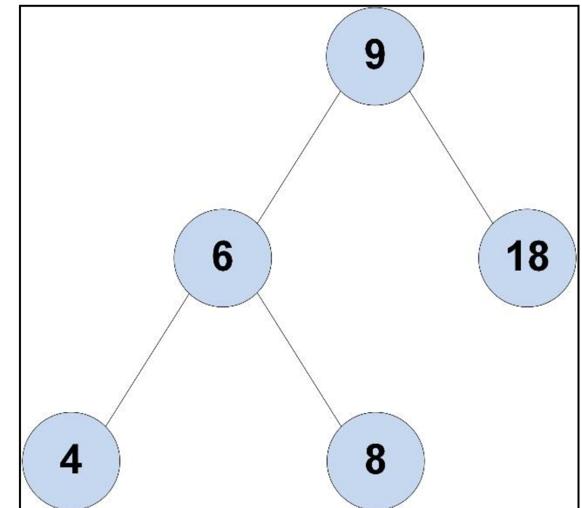
Δυαδικά Δέντρα Αναζήτησης (2)

- Ένα δένδρο του οποίου κάθε κόμβος έχει το πολύ δύο παιδιά ονομάζεται δυαδικό δέντρο

- Στα δύο τμήματα του κάθε κόμβου αναφερόμαστε ως αριστερό και δεξιό υποδένδρο, αντίστοιχα

- Δυαδικό δέντρο αναζήτησης είναι ένα δυαδικό δένδρο, του οποίου κάθε κόμβος σχετίζεται με ένα κλειδί και ισχύει ότι:
  - α. το κλειδί του είναι μεγαλύτερο (ή ίσο) από τα κλειδιά όλων των κόμβων στο αριστερό του υποδένδρο και
  - β. το κλειδί του είναι μικρότερο (ή ίσο) από τα κλειδιά όλων των κόμβων στο δεξιό του υποδένδρο

- Ο σημαντικότερος λόγος δημιουργίας ενός τέτοιου δένδρου είναι ότι επιτρέπει την ανάπτυξη αλγορίθμων με υψηλές επιδόσεις κατά την εκτέλεση κρίσιμων λειτουργιών, όπως της εισαγωγής και αναζήτησης ενός στοιχείου



- Μπορείτε να δείτε ένα παράδειγμα υλοποίησης κάποιων βασικών λειτουργιών σε ένα δυαδικό δένδρο αναζήτησης στην αντίστοιχη ενότητα του βιβλίου