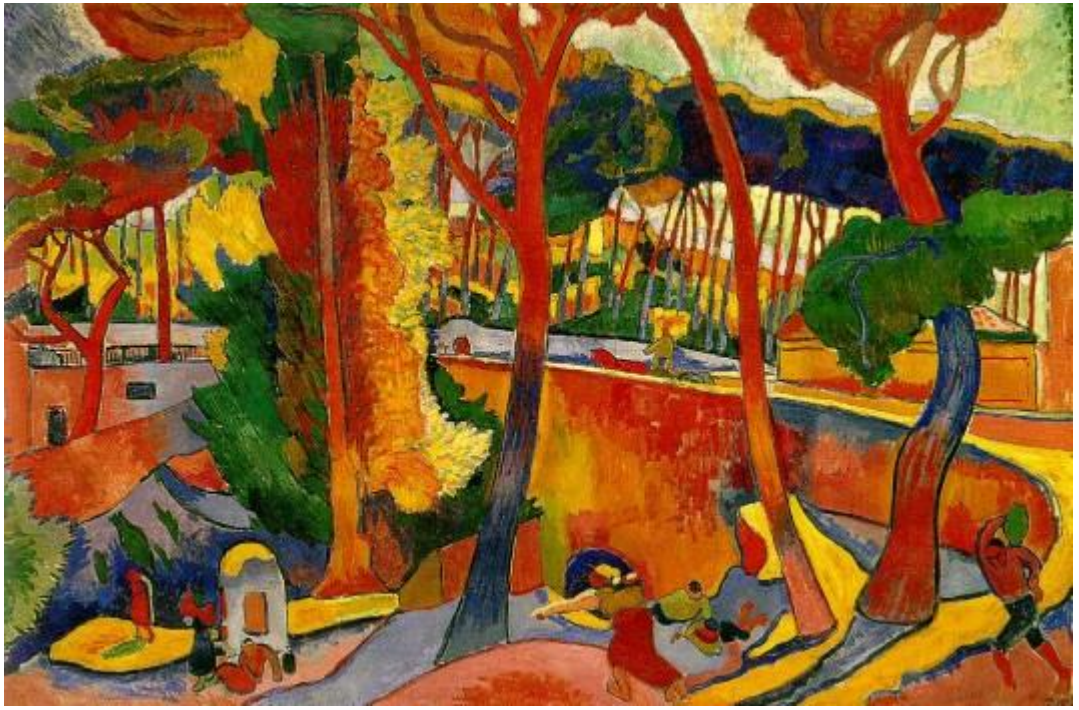
Διαχείριση Μνήμης



Andre Derain, *The turning road, Lestraque,* 1906

Κωστής Σαγώνας <kostis@cs.ntua.gr>
Νίκος Παπασπύρου <nickie@softlab.ntua.gr>

Δυναμική δέσμευση μνήμης

- Κατά την εκτέλεση του προγράμματος υπάρχει ανάγκη για δέσμευση μνήμης:
  - Εγγραφών δραστηριοποίησης
  - Αντικειμένων
  - Άμεσων κλήσεων δέσμευσης μνήμης: **`new`**, **`malloc`**, κ.λπ.
  - Εμμέσων κλήσεων δέσμευσης μνήμης: δημιουργία συμβολοσειρών, buffers για αρχεία, πινάκων με δυναμικά καθοριζόμενο μέγεθος, κ.λπ.
- Οι υλοποιήσεις των γλωσσών παρέχουν τη δυνατότητα διαχείρισης μνήμης κατά το χρόνο εκτέλεσης του προγράμματος

Περιεχόμενα

- Μοντέλο μνήμης με πίνακες στη Java
- Στοίβες (stacks)
- Σωροί (heaps)
- Τρέχοντες σύνδεσμοι στο σωρό
- Συλλογή σκουπιδιών (garbage collection)

Μοντέλο μνήμης

- Προς το παρόν, ας υποθέσουμε ότι το λειτουργικό σύστημα παρέχει στο πρόγραμμα μία ή περισσότερες περιοχές μνήμης κάποιου προκαθορισμένου μεγέθους για δυναμική δέσμευση
- Θα μοντελοποιήσουμε τις περιοχές αυτές ως πίνακες στη Java (Java arrays)
  - Για να δούμε παραδείγματα κώδικα διαχείρισης μνήμης, και
  - Για πρακτική εξάσκηση με τη Java

Δήλωση πινάκων στη Java

- Στη Java οι πίνακες ορίζονται ως:

```
int[] a = null;
```

- Οι τύποι των πινάκων είναι τύποι αναφοράς – ένας πίνακας στην πραγματικότητα είναι ένα αντικείμενο, το οποίο μπορούμε να διαχειριστούμε με κάποια ειδική σύνταξη
- Στο παραπάνω, η μεταβλητή **a** αρχικοποιείται σε **null**
- Μπορεί να της ανατεθεί μια αναφορά σε έναν πίνακα με ακέραιες τιμές, αλλά στο παραπάνω παράδειγμα κάποιος τέτοιος πίνακας δεν της έχει ανατεθεί μέχρι στιγμής

Δημιουργία πινάκων στη Java

- Νέα αντικείμενα πίνακες δημιουργούνται με τη **new**:

```
int[] a = null;
a = new int[666];
```

- Και μπορούμε να συνδυάσουμε τις δύο παραπάνω δηλώσεις σε μία ως εξής:

```
int[] a = new int[666];
```

Χρησιμοποίηση πινάκων στη Java

```
int i = 0;
while (i < a.length) {
  a[i] = 42;
  i++;
}
```

- Για να προσπελάσουμε ένα στοιχείο του πίνακα χρησιμοποιούμε την έκφραση **`a[i]`** (ως lvalue ή rvalue): όπου **`a`** είναι μια έκφραση αναφοράς σε πίνακα και **`i`** είναι μια ακέραια έκφραση
- Για το μέγεθος ενός πίνακα χρησιμοποιούμε **`a.length`**
- Οι δείκτες των πινάκων έχουν εύρος **`0`**..(**`a.length-1`**)

Διαχειριστές μνήμης στη Java

```
public class MemoryManager {
  private int[] memory;

  /**
   * MemoryManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public MemoryManager(int[] initialMemory) {
    memory = initialMemory;
  }
  …
}
```

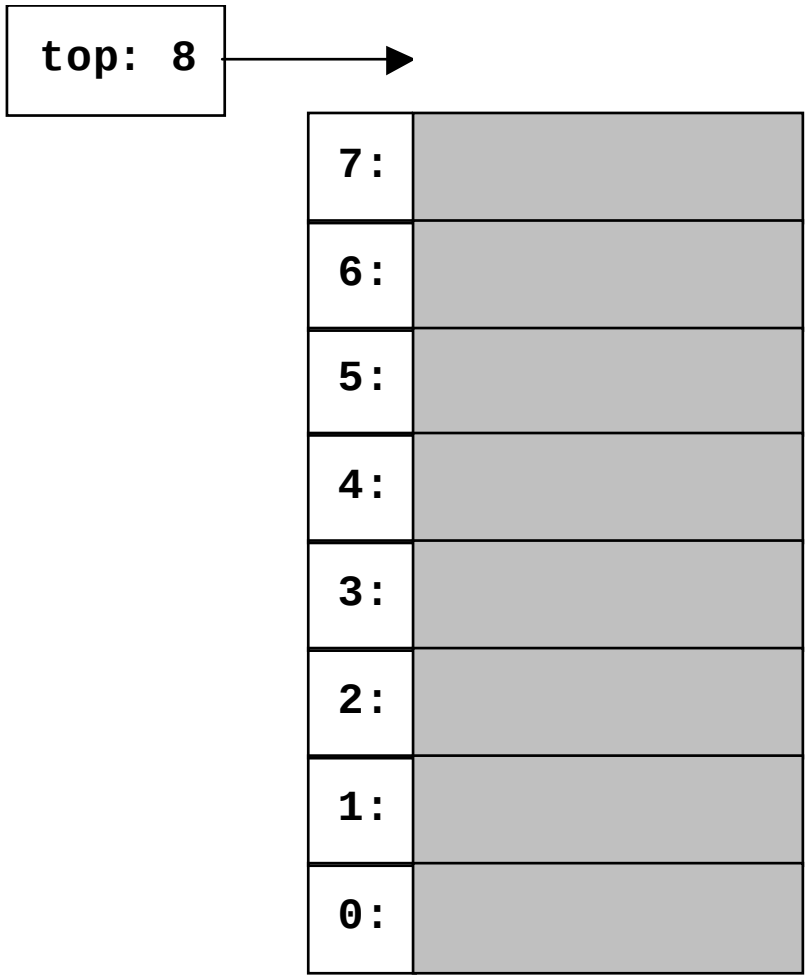
Ο πίνακας **`initialMemory`** είναι μια περιοχή μνήμης που μας δίνεται από το λειτουργικό σύστημα.

Στοίβες (stacks)

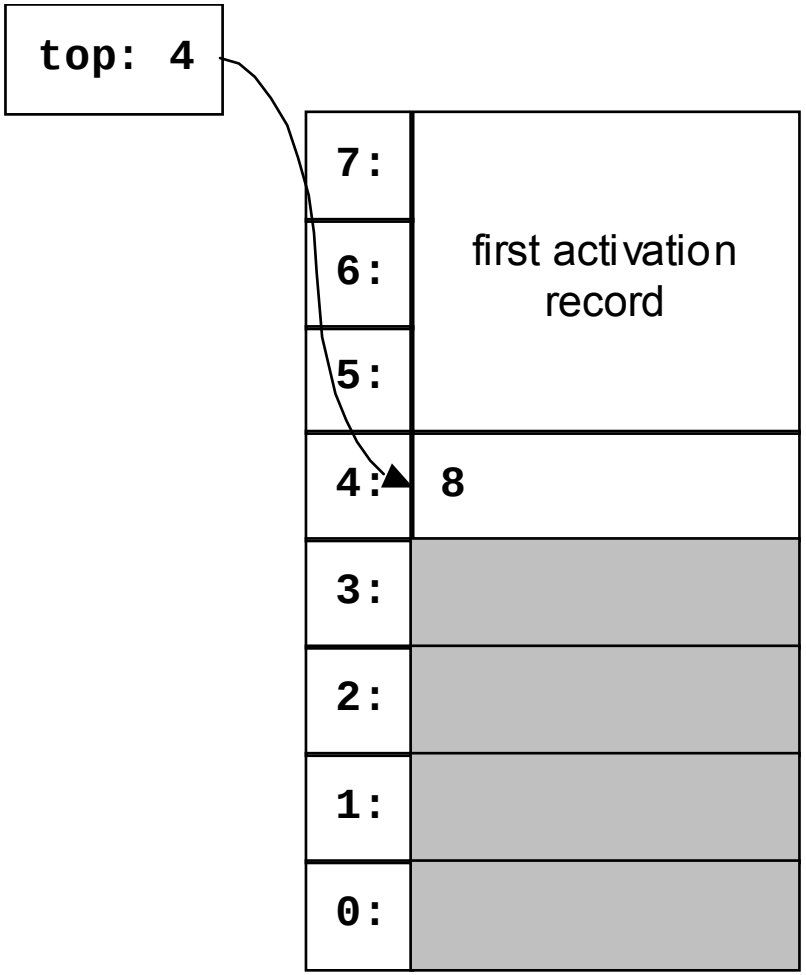
Στοίβες από εγγραφές δραστηριοποίησης

- Στις περισσότερες γλώσσες, οι εγγραφές δραστηριοποίησης δεσμεύονται δυναμικά
- Σε πολλές γλώσσες, αρκεί να δεσμεύσουμε μια εγγραφή κατά την κλήση μιας συνάρτησης η οποία αποδεσμεύεται κατά την επιστροφή της συνάρτησης
- Με αυτόν τον τρόπο παράγεται μια στοίβα από εγγραφές δραστηριοποίησης
- Μια στοίβα χρειάζεται ένα σχετικά απλό αλγόριθμο διαχείρισης μνήμης

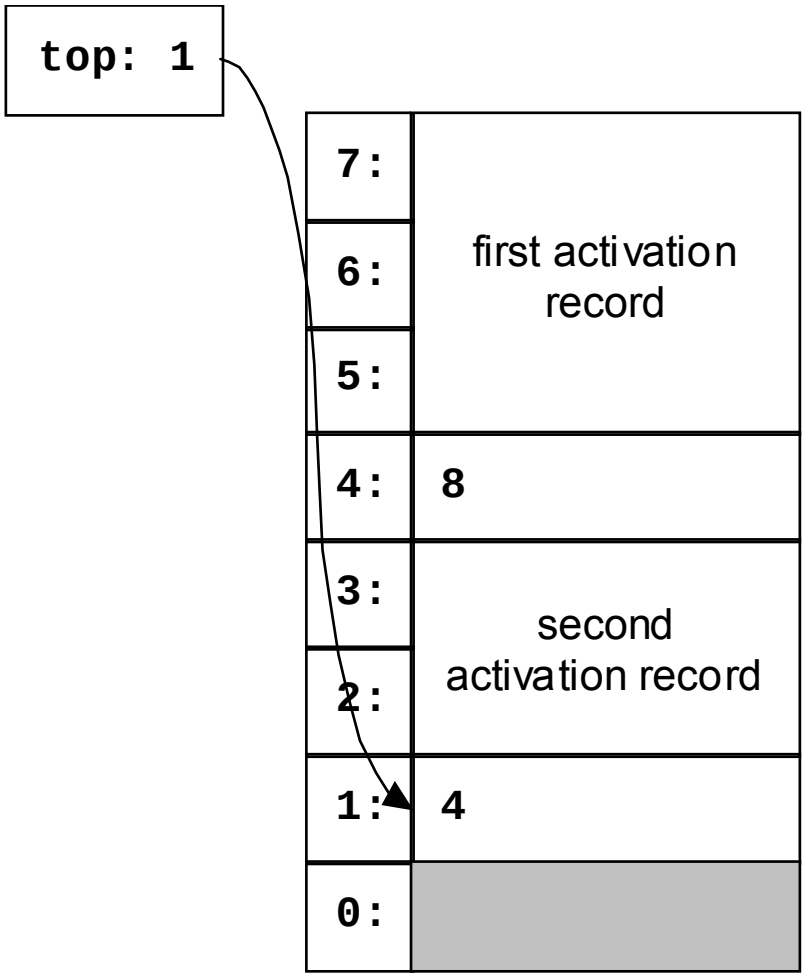
Εικονογράφηση μιας στοίβας



Μια κενή στοίβα με 8 λέξεις. Η στοίβα μεγαλώνει προς τα κάτω, από μεγάλες σε μικρές διευθύνσεις. Μια προκαθορισμένη θέση μνήμης (ή πιθανόν ένας καταχωρητής) αποθηκεύει τη διεύθυνση της κορυφής της στοίβας (της μικρότερης χρησιμοποιούμενης θέσης μνήμης).



Το πρόγραμμα καλεί την **`m.push(3)`**, η οποία επιστρέφει 5: τη διεύθυνση της πρώτης από τις 3 λέξεις που δεσμεύονται για την εγγραφή δραστηριοποίησης. Η διαχείριση της μνήμης χρησιμοποιεί μια επιπλέον λέξη για να καταγράψει την προηγούμενη τιμή της **`top`**.



Το πρόγραμμα τώρα καλεί την **`m.push(2)`**, η οποία γυρνάει 2: τη διεύθυνση της πρώτης από τις 2 λέξεις που δεσμεύονται για την εγγραφή δραστηριοποίησης. Η στοίβα τώρα γέμισε—δεν υπάρχει χώρος ούτε για μια κλήση **`m.push(1)`**.

Για μια κλήση **`m.pop()`**, αναθέτουμε

```
top = memory[top]
```

για να επιστρέψουμε στην προηγούμενη κατάσταση.

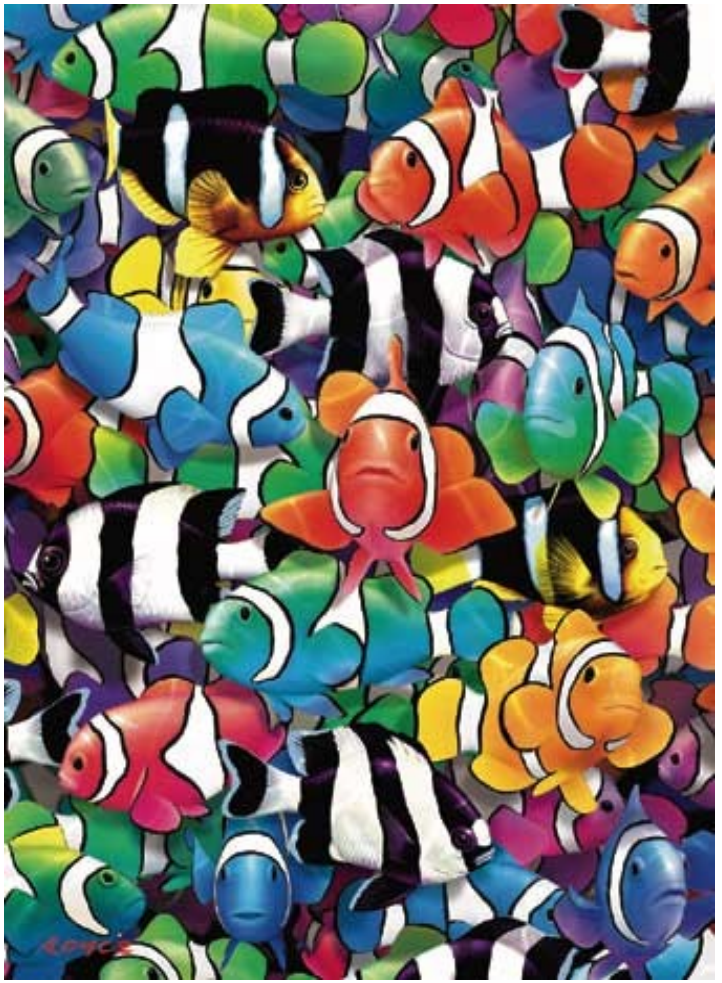
Υλοποίηση της μνήμης στοίβας στη Java

```
public class StackManager {
  private int[] memory; // the memory we manage
  private int top;      // index of top stack block

  /**
   * StackManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public StackManager(int[] initialMemory) {
    memory = initialMemory;
    top = memory.length;
  }
```

```
  /**
    * Allocate a block and return its address.
    * @param requestSize int size of block, > 0
    * @return start of block address
    * @throws StackOverflowError if out of stack space
    */
   public int push(int requestSize) {
     int oldtop = top;
     top -= (requestSize+1); // extra word for oldtop
     if (top < 0) throw new StackOverflowError();
     memory[top] = oldtop;
     return top+1;
   }

  /** Pop the top stack frame.
    * This works only if the stack is not empty.
    */
   public void pop() {
     top = memory[top];
   }
}
```

Θα δούμε την εντολή **throw** και το χειρισμό εξαιρέσεων σε επόμενο μάθημα.

Σωροί (heaps)



Διάταξη σε σωρό

- Η διάταξη σε στοίβα έχει σχετικά εύκολη υλοποίηση
- Αλλά δεν επαρκεί πάντα: τι συμβαίνει για παράδειγμα εάν οι δεσμεύσεις και οι αποδεσμεύσεις κομματιών μνήμης ακολουθούν τυχαία σειρά;
- Ένας σωρός (heap) είναι μία ακολουθία από μπλοκ μνήμης τα οποία μας δίνουν τη δυνατότητα να έχουμε μη ταξινομημένη δέσμευση και αποδέσμευση μνήμης
- Υπάρχουν πολλοί μηχανισμοί υλοποίησης μνήμης σωρού

Υλοποίηση μέσω του αλγόριθμου First Fit

- Ένας σωρός υλοποιείται ως μια συνδεδεμένη λίστα από ελεύθερα μπλοκ (**free list**)
- Αρχικά αποτελείται από μόνο ένα μεγάλο μπλοκ
- Για δέσμευση κάποιου κομματιού μνήμης:
  - Ψάχνουμε στη free list για το πρώτο μπλοκ με μέγεθος τέτοιο ώστε να ικανοποιούνται οι απαιτήσεις της ζήτησης
  - Εάν το μπλοκ που βρήκαμε είναι μεγαλύτερο από τη ζήτηση, τότε επιστρέφουμε το αχρησιμοποίητο κομμάτι του στη free list
  - Ικανοποιούμε την απαίτηση δέσμευσης μνήμης και επιστρέφουμε την αρχική διεύθυνση του δεσμευμένου μπλοκ
- Για αποδέσμευση, απλώς προσθέτουμε το ελευθερωθέν μπλοκ στην αρχή της free list

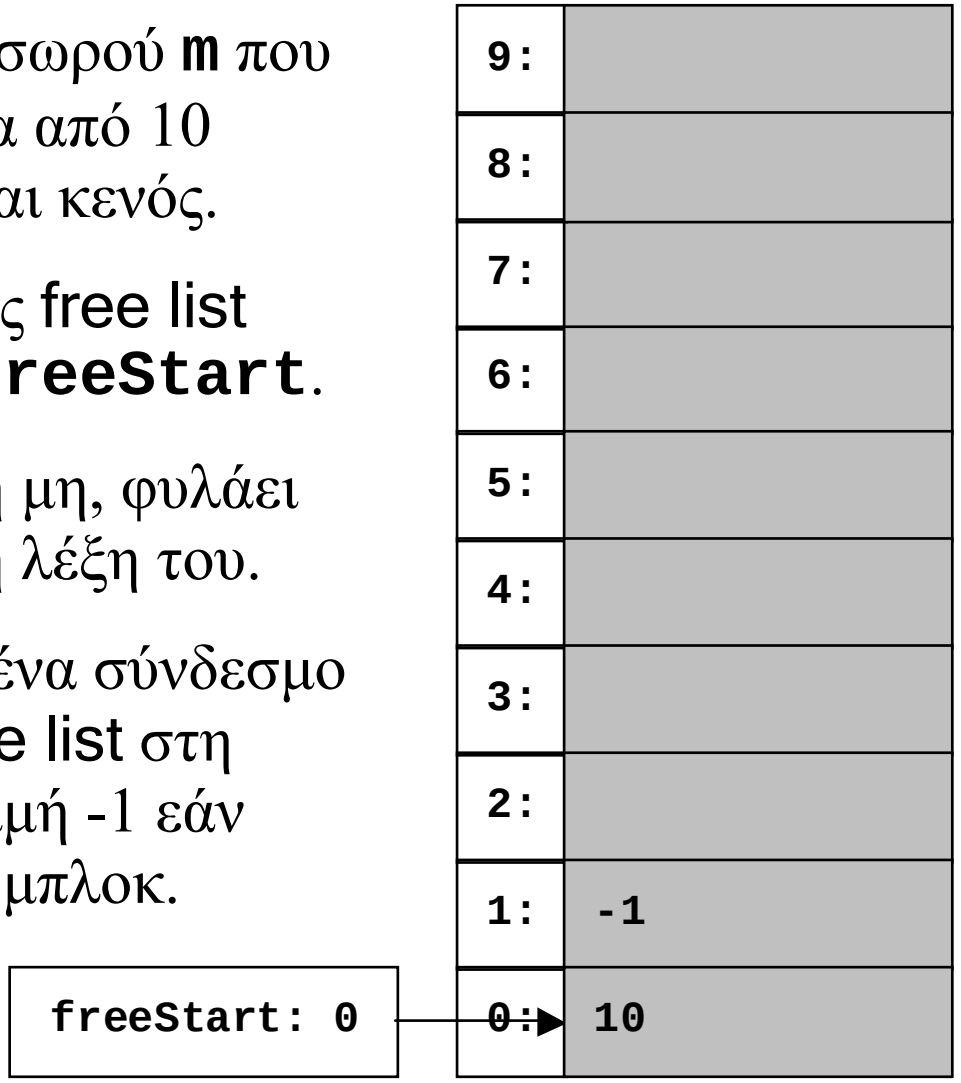
Εικονογράφηση της μνήμης σωρού

Ένας διαχειριστής μνήμης σωρού **m** που αποτελείται από ένα πίνακα από 10 λέξεις, ο οποίος αρχικά είναι κενός.

Ο σύνδεσμος στην αρχή της free list κρατιέται στη μεταβλητή **`freeStart`**.

Κάθε μπλοκ, δεσμευμένο ή μη, φυλάει το μέγεθός του στην πρώτη λέξη του.

Τα ελεύθερα μπλοκ έχουν ένα σύνδεσμο στο επόμενο μπλοκ της free list στη δεύτερη λέξη τους, ή την τιμή -1 εάν πρόκειται για το τελευταίο μπλοκ.

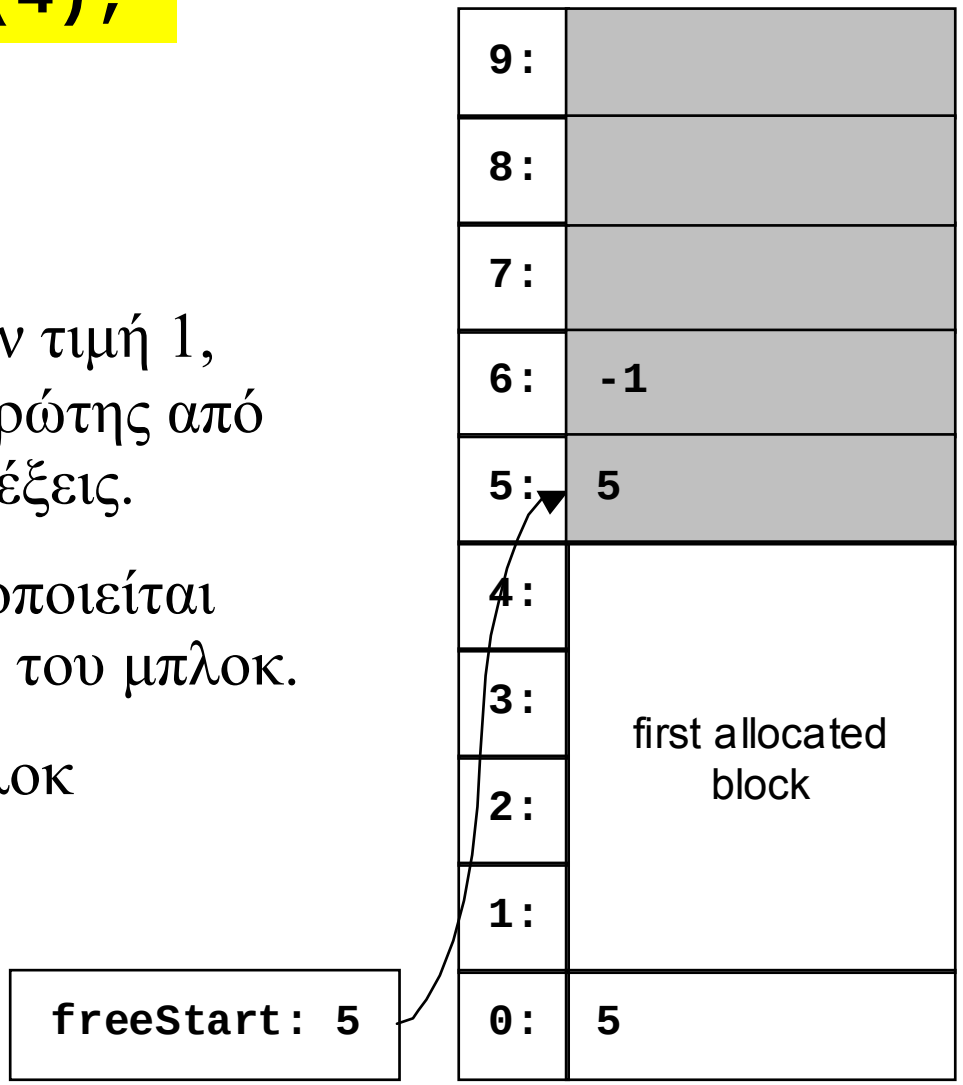


```
p1 = m.allocate(4);
```

Η αναφορά **p1** θα πάρει την τιμή 1, δηλαδή τη διεύθυνση της πρώτης από τις τέσσερις δεσμευμένες λέξεις.

Μία επιπλέον λέξη χρησιμοποιείται για να κρατήσει το μέγεθος του μπλοκ.

Το υπόλοιπο μέρος του μπλοκ επεστράφη στη free list.

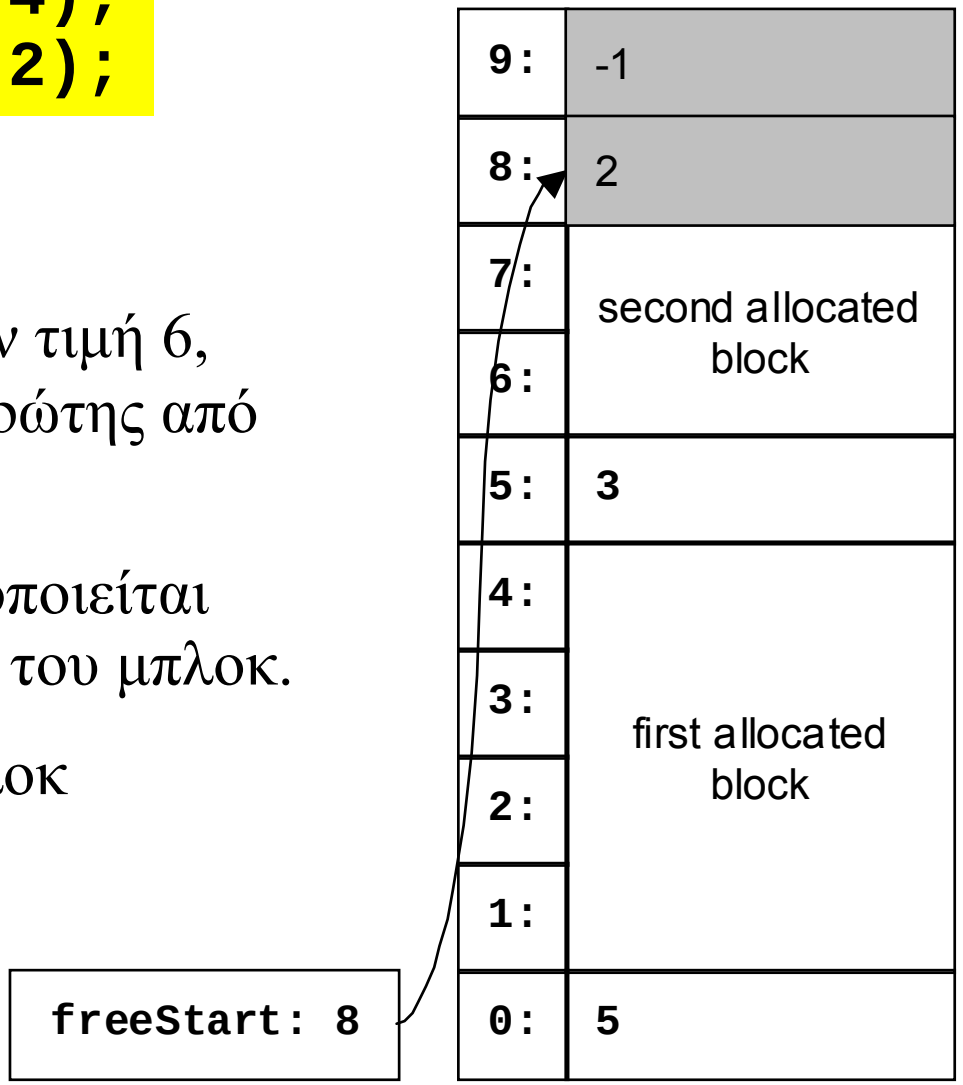


```
p1 = m.allocate(4);
p2 = m.allocate(2);
```

Η αναφορά **p2** θα πάρει την τιμή 6, δηλαδή τη διεύθυνση της πρώτης από τις δύο δεσμευμένες λέξεις.

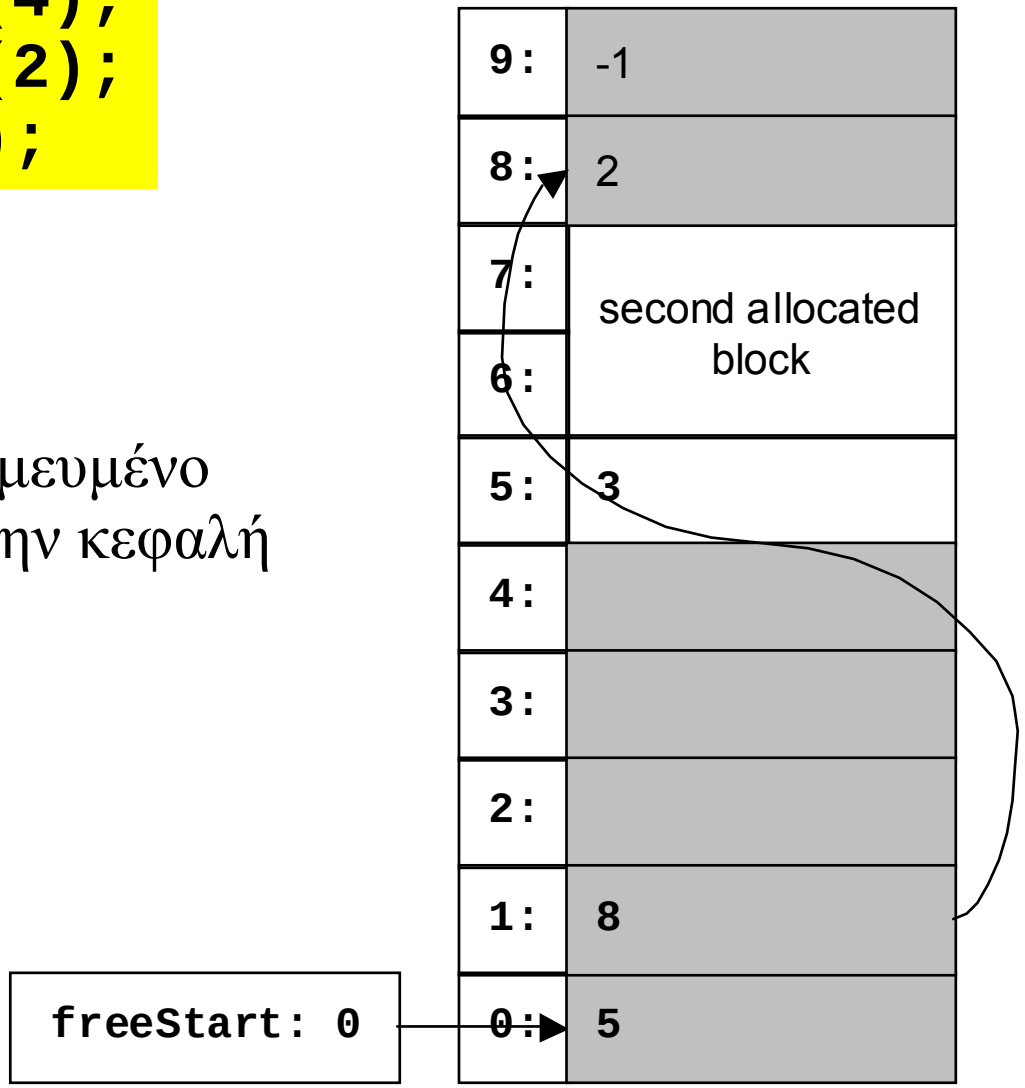
Μία επιπλέον λέξη χρησιμοποιείται για να κρατήσει το μέγεθος του μπλοκ.

Το υπόλοιπο μέρος του μπλοκ επεστράφη στη free list.



```
p1 = m.allocate(4);
p2 = m.allocate(2);
m.deallocate(p1);
```

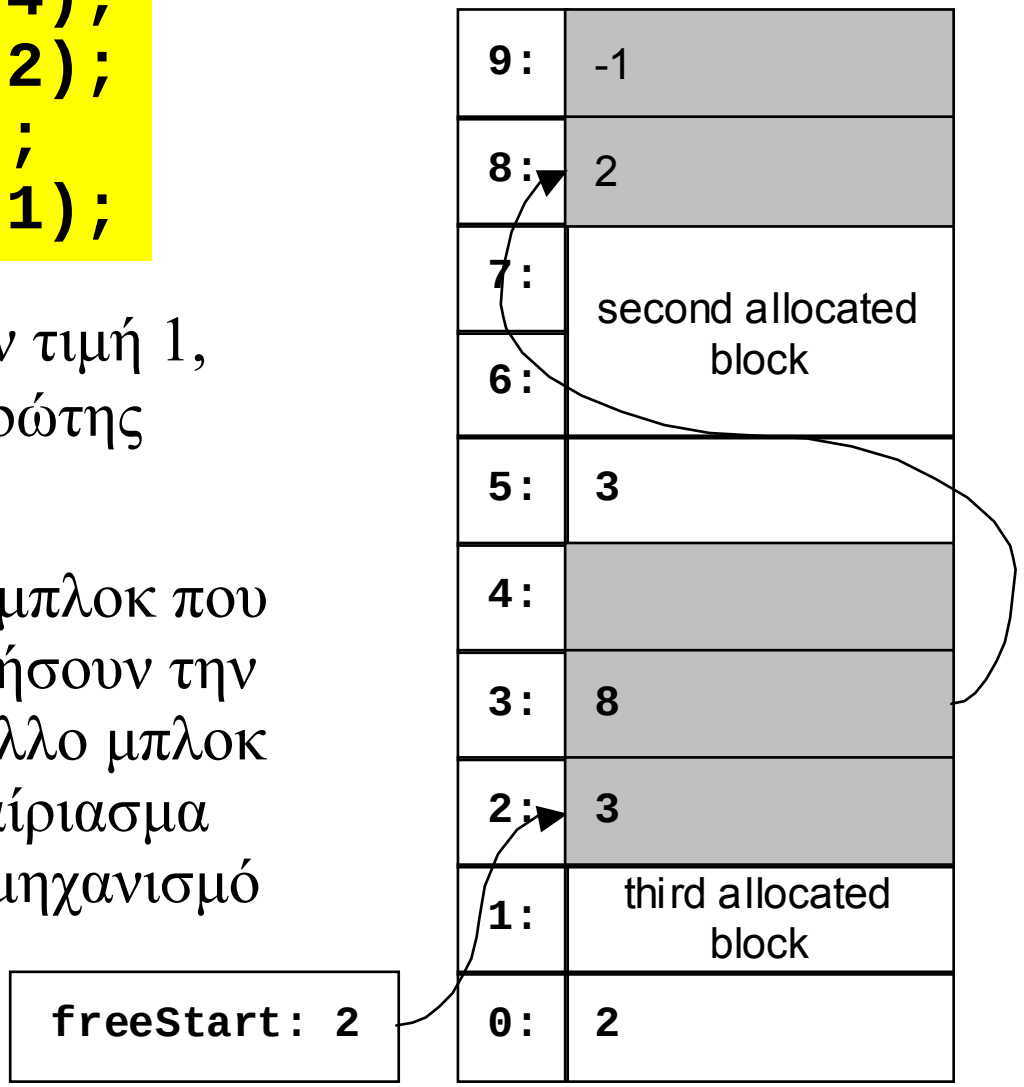
Αποδεσμεύει το πρώτο δεσμευμένο μπλοκ και το επιστρέφει στην κεφαλή της free list.



```
p1 = m.allocate(4);
p2 = m.allocate(2);
m.deallocate(p1);
p3 = m.allocate(1);
```

Η αναφορά **p3** θα πάρει την τιμή 1, δηλαδή τη διεύθυνση της πρώτης δεσμευμένης λέξης.

Προσέξτε ότι υπήρχαν δύο μπλοκ που θα μπορούσαν να εξυπηρετήσουν την απαίτηση δέσμευσης. Το άλλο μπλοκ θα προσέφερε ένα τέλειο ταίριασμα και θα επιλέγονταν από το μηχανισμό δέσμευσης μνήμης Best Fit.



Υλοποίηση της μνήμης σωρού στη Java

```
public class HeapManager {
  static private final int NULL = -1; // null link
  public int[] memory;   // the memory we manage
  private int freeStart; // start of the free list

  /**
   * HeapManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public HeapManager(int[] initialMemory) {
    memory = initialMemory;
    memory[0] = memory.length; // one big free block
    memory[1] = NULL; // free list ends with it
    freeStart = 0;    // free list starts with it
  }
```

```
/**
 * Allocate a block and return its address.
 * @param requestSize int size of block, > 0
 * @return block address
 * @throws OutOfMemoryError if no block big enough
 */
public int allocate(int requestSize) {
  int size = requestSize + 1; // size with header

  // Do first-fit search: linear search of the free
  // list for the first block of sufficient size.
  int p = freeStart; // head of free list
  int lag = NULL;
  while (p != NULL && memory[p] < size) {
    lag = p; // lag is previous p
    p = memory[p+1]; // link to next block
  }
  if (p == NULL) // no block large enough
    throw new OutOfMemoryError();
  int nextFree = memory[p+1]; // block after p
```

```
    // Now p is the index of a block of sufficient size,
    // and lag is the index of p's predecessor in the
    // free list, or NULL, and nextFree is the index of
    // p's successor in the free list, or NULL.
    // If the block has more space than we need, carve
    // out what we need from the front and return the
    // unused end part to the free list.
    int unused = memory[p]-size; // extra space
    if (unused > 1) { // if more than a header's worth
      nextFree = p+size; // index of the unused piece
      memory[nextFree] = unused; // fill in size
      memory[nextFree+1] = memory[p+1]; // fill in link
      memory[p] = size; // reduce p's size accordingly
    }

    // Link out the block we are allocating and done.
    if (lag == NULL) freeStart = nextFree;
    else memory[lag+1] = nextFree;
    return p+1; // index of useable word (after header)
  }
```

```
   /**
    * Deallocate an allocated block.  This works only
    * if the block address is one that was returned
    * by allocate and has not yet been deallocated.
    * @param address int address of the block
    */
   public void deallocate(int address) {
     int addr = address-1;
     memory[addr+1] = freeStart;
     freeStart = addr;
   }
}
```

Ένα πρόβλημα

- Έστω η παρακάτω ακολουθία εντολών:

```
p1 = m.allocate(4);
p2 = m.allocate(4);
m.deallocate(p1);
m.deallocate(p2);
p3 = m.allocate(7);
```

- Η τελευταία κλήση της **`allocate`** θα αποτύχει διότι τα ελεύθερα μπλοκ σπάνε σε μικρά κομμάτια αλλά πουθενά δεν ξαναενώνονται
- Χρειαζόμαστε ένα μηχανισμό ο οποίος συνασπίζει συνεχόμενα γειτονικά μπλοκ που είναι ελεύθερα

Μια από τις λύσεις του προβλήματος

- Υλοποιούμε μια πιο έξυπνη μέθοδο **`deallocate`**:
  - Διατηρούμε τη λίστα των ελεύθερων μπλοκ ταξινομημένη με βάση τις διευθύνσεις των μπλοκ
  - Όταν ελευθερώνουμε ένα μπλοκ, κοιτάμε το προηγούμενο και το επόμενό του μπλοκ που είναι ελεύθερα
  - Εάν το μπλοκ που ελευθερώνουμε είναι συνεχόμενο με κάποιο από αυτά, τα συνασπίζουμε
- Η μέθοδος αυτή είναι πιο αποτελεσματική από την απλή τοποθέτηση του μπλοκ στην κεφαλή της free list αλλά όμως είναι και πιο δαπανηρή σε χρόνο εκτέλεσης

```
/**
 * Deallocate an allocated block.  This works only
 * if the block address is one that was returned
 * by allocate and has not yet been deallocated.
 * @param address int address of the block
 */
public void deallocate(int address) {
  int addr = address-1; // real start of the block

  // Find the insertion point in the sorted
  // free list for this block.

  int p = freeStart;
  int lag = NULL;
  while (p != NULL && p < addr) {
    lag = p;
    p = memory[p+1];
  }
```

```
// Now p is the index of the block to come after
// ours in the free list, or NULL, and lag is the
// index of the block to come before ours in the
// free list, or NULL.

// If the one to come after ours is adjacent to it,
// merge it into ours and restore the property
// described above.

if (addr+memory[addr] == p) {
  memory[addr] += memory[p]; // add its size to ours
  p = memory[p+1];
}
```

```
    if (lag == NULL) { // ours will be first free
      freeStart = addr;
      memory[addr+1] = p;
    }
    else if (lag+memory[lag]==addr) { // block before is
                                      // adjacent to ours
      memory[lag] += memory[addr]; // merge ours into it
      memory[lag+1] = p;
    }
    else { // neither: just a simple insertion
      memory[lag+1] = addr;
      memory[addr+1] = p;
    }
  }
```

Ελεύθερες λίστες διαχωρισμένες βάσει μεγέθους

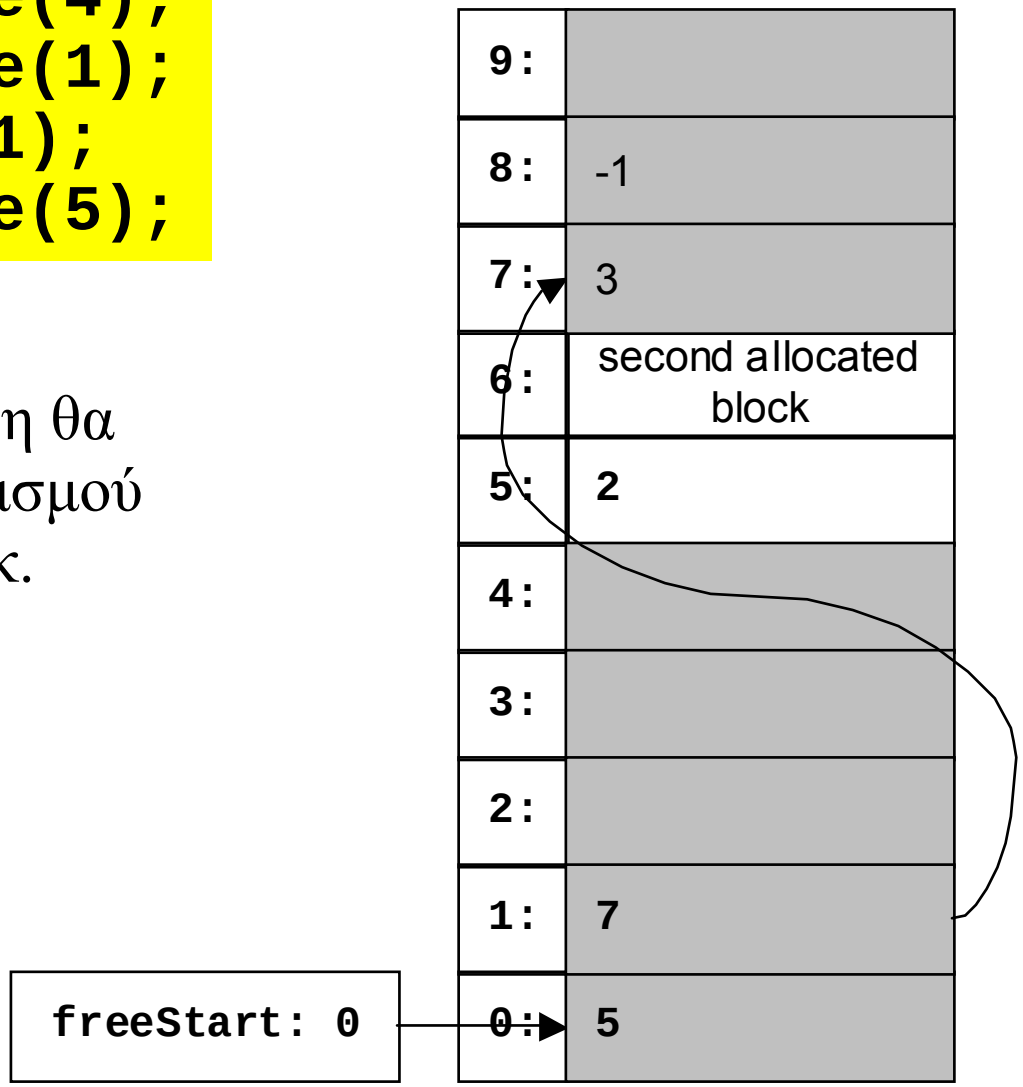
- Τα μικρού (και συνήθως προκαθορισμένου) μεγέθους μπλοκ συνήθως δεσμεύονται και αποδεσμεύονται πολύ πιο συχνά από τα μεγάλου ή τυχαίου μεγέθους μπλοκ
- Μια συνήθης βελτιστοποίηση: διατηρούμε ξεχωριστές λίστες ελεύθερων μπλοκ για (μικρού) μεγέθους μπλοκ τα οποία είναι δημοφιλή
- Με άλλα λόγια διαχωρίζουμε τις λίστες με βάση το μέγεθος των μπλοκ που αποθηκεύουν και αναφερόμαστε σε **ελεύθερες λίστες διαχωρισμένες βάσει μεγέθους (size-segregated free lists)**

Κατακερματισμός (fragmentation)

- Λέμε ότι η μνήμη είναι **κατακερματισμένη** όταν οι ελεύθερες περιοχές διαχωρίζονται από δεσμευμένα μπλοκ, και κατά συνέπεια δεν είναι δυνατό να δεσμεύσουμε όλη την ελεύθερη μνήμη ως ένα μπλοκ
- Γενικότερα, η μνήμη είναι κατακερματισμένη όταν ο διαχειριστής της μνήμης σωρού δεν είναι σε θέση να εξυπηρετήσει τις αιτήσεις δέσμευσης μνήμης παρόλο που υπάρχει αρκετή ελεύθερη μνήμη συνολικά

```
p1 = m.allocate(4);
p2 = m.allocate(1);
m.deallocate(p1);
p3 = m.allocate(5);
```

Η τελευταία δέσμευση θα αποτύχει λόγω τεμαχισμού των ελεύθερων μπλοκ.



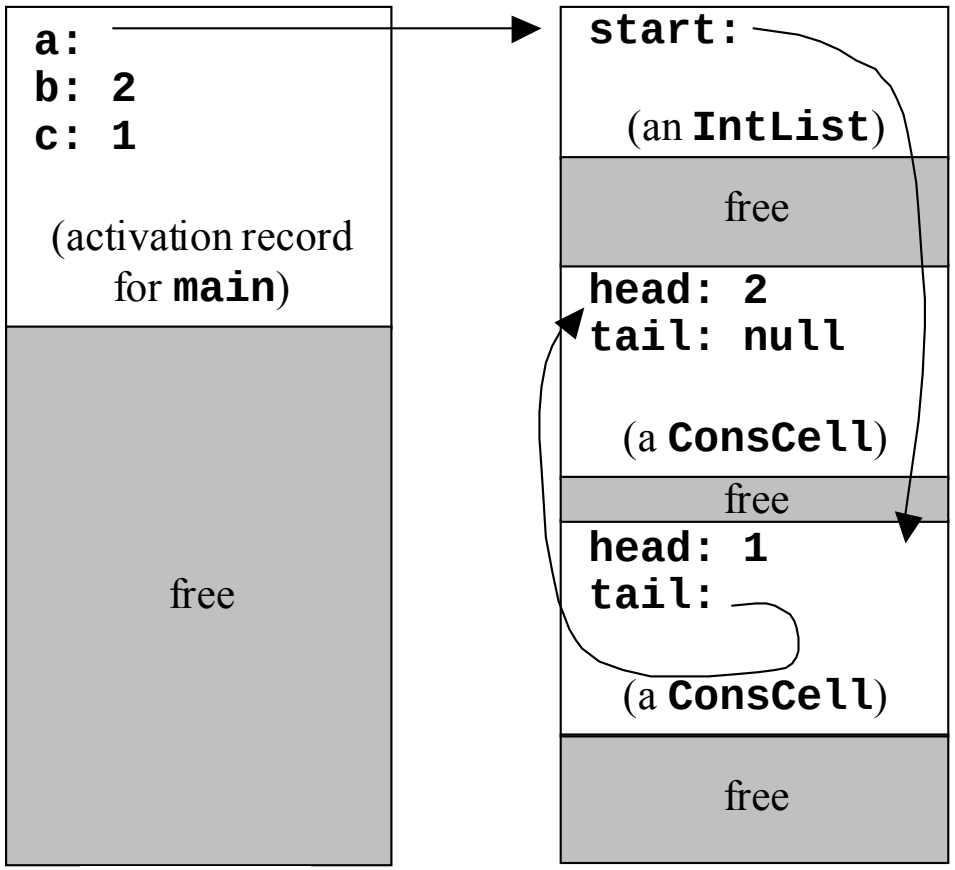
Τρέχοντες Σύνδεσμοι στο Σωρό

Τρέχοντες σύνδεσμοι στο σωρό

- Μέχρι στιγμής, βλέπαμε το υπό εκτέλεση πρόγραμμα ως ένα μαύρο κουτί που απλώς απαιτεί δεσμεύσεις και αποδεσμεύσεις μνήμης
- Τι κάνει το πρόγραμμα με τις διευθύνσεις μνήμης που του παραχωρούνται;
- Τις αποθηκεύει σε κάποια ονόματα (μεταβλητές)

- Ένας **τρέχων σύνδεσμος στο σωρό (current heap link)** είναι μια θέση μνήμης που περιέχει μια τιμή η οποία θα χρησιμοποιηθεί από το πρόγραμμα ως δείκτης στο σωρό

Τρέχοντες σύνδεσμοι στο σωρό




```
IntList a =
  new IntList(null);
int b = 2;
int c = 1;
a = a.cons(b);
a = a.cons(c);
```

Πού είναι οι τρέχοντες σύνδεσμοι στο σωρό στη διπλανή εικόνα;

Εύρεση των τρεχόντων συνδέσμων στο σωρό

- Αρχίζουμε με ένα σύνολο ριζών (root set): θέσεις μνήμης εκτός του σωρού που περιέχουν συνδέσμους σε θέσεις μνήμης στο σωρό, π.χ. σε
  - ενεργές εγγραφές δραστηριοποίησης (στη στοίβα)
  - καθολικές ή στατικές μεταβλητές
- Για κάθε στοιχείο του συνόλου, κοιτάμε το δεσμευμένο μπλοκ στο οποίο δείχνει ο σύνδεσμος και προσθέτουμε όλες τις θέσεις μνήμης του συγκεκριμένου μπλοκ στο σύνολό μας
- Επαναλαμβάνουμε την παραπάνω διαδικασία μέχρις του σημείου όπου δε βρίσκουμε άλλες νέες θέσεις μνήμης

Λάθη κατά την εύρεση των τρεχόντων συνδέσμων

- **Παραλείψεις**: ξεχνάμε να περιβάλουμε μια θέση μνήμης που έχει έναν τρέχοντα σύνδεσμο στο σωρό
- **Άχρηστες τιμές**: συμπεριλαμβάνουμε θέσεις μνήμης για τις οποίες το πρόγραμμα ποτέ δε θα χρησιμοποιήσει τις τιμές που θα αποθηκευθούν
- **Διευθύνσεις που δεν είναι θέσεις μνήμης**: συμπεριλαμβάνουμε κάποιες θέσεις μνήμης στο σωρό μόνο και μόνο επειδή δε μπορούμε να ξεχωρίσουμε τιμές που χρησιμοποιούνται π.χ. ως δείκτες στο σωρό ή ως ακέραιοι (με τιμή που φαίνεται ως μια διεύθυνση μνήμης στο σωρό)

Δε μπορούμε πάντα να αποφύγουμε τα λάθη

- Για τη σωστή διαχείριση του σωρού, τα λάθη παράλειψης είναι μη αποδεκτά και δε συγχωρούνται
- Άρα είμαστε υποχρεωμένοι να συμπεριλάβουμε όλες τις θέσεις μνήμης για τις οποίες υπάρχει πιθανότητα η τιμή που αποθηκεύεται στη συγκεκριμένη θέση μνήμης να χρησιμοποιηθεί από το πρόγραμμα
- Κατά συνέπεια, το να συμπεριλάβουμε άχρηστες θέσεις μνήμης στον υπολογισμό είναι αναπόφευκτο
- Ανάλογα με τη γλώσσα, μπορεί να είναι αδύνατο να αποφύγουμε να συμπεριλάβουμε και τιμές που απλώς δείχνουν σαν θέσεις μνήμης στο σωρό

Τιμές που δείχνουν σαν διευθύνσεις μνήμης

- Για κάποιες μεταβλητές μπορεί να μην είμαστε σε θέση να καταλάβουμε πώς χρησιμοποιούνται οι τιμές τους
- Για παράδειγμα, η παρακάτω μεταβλητή **x** μπορεί να χρησιμοποιηθεί είτε ως δείκτης είτε ως πίνακας από τέσσερις χαρακτήρες

```
union {
  char *p;
  char tag[4];
} x;
```

Σημείωση: το παραπάνω πρόβλημα είναι ακόμα χειρότερο στη C, διότι οι C compilers δεν κρατούν πληροφορία για τους τύπους των μεταβλητών κατά τη διάρκεια εκτέλεσης του προγράμματος

Συμπίεση του σωρού (heap compaction)

- Μια λειτουργία που χρειάζεται πληροφορία για το σύνολο των τρεχόντων συνδέσμων στο σωρό
- Ο διαχειριστής της μνήμης μετακινεί δεσμευμένα μπλοκ:
  - Αντιγράφει το μπλοκ σε μια νέα θέση, και
  - Επικαιροποιεί όλους τους συνδέσμους στο (ή κάπου μέσα στο) μπλοκ
- Κατά συνέπεια συμπιέζει το σωρό, μετακινώντας όλα τα δεσμευμένα μπλοκ στο ένα άκρο του και αφήνοντας ένα μεγάλο ελεύθερο μπλοκ χωρίς κατακερματισμό

Συλλογή Σκουπιδιών

Κάποια συνηθισμένα προβλήματα με δείκτες

```
type
  p: ^Integer;
begin
  new(p);
  p^ := 42;
  dispose(p);
  p^ := p^ + 1
end
```

Ξεκρέμαστος δείκτης: το διπλανό πρόγραμμα Pascal χρησιμοποιεί ένα δείκτη σε μπλοκ μνήμης και μετά την αποδέσμευση του συγκεκριμένου μπλοκ

```
procedure Leak;
  type
    p: ^Integer;
  begin
    new(p)
  end;
```

Διαρροή μνήμης: το διπλανό πρόγραμμα Pascal δεσμεύει ένα μπλοκ μνήμης αλλά ξεχνάει να το αποδεσμεύσει

Συλλογή σκουπιδιών (garbage collection)

- Αφού τόσα σφάλματα λογισμικού συμβαίνουν λόγω λαθών αποδέσμευσης μνήμης…
- …και επειδή δεν είναι βολικό για τον προγραμματιστή να σκέφτεται για το πώς θα γίνει σωστά η αποδέσμευση μνήμης…
- …για ποιο λόγο να μην είναι ευθύνη της υλοποίησης της γλώσσας η αυτόματη ανακύκλωση μνήμης;

Τρεις βασικοί μηχανισμοί ανακύκλωσης μνήμης

1. Μαρκάρισμα και σκούπισμα (mark and sweep)
2. Αντιγραφή (copying)
3. Μέτρημα αναφορών (reference counting)

Μαρκάρισμα και σκούπισμα

- Ένας συλλέκτης μαρκαρίσματος και σκουπίσματος (mark-and-sweep collector) χρησιμοποιεί τους τρέχοντες συνδέσμους στο σωρό σε μια διαδικασία που έχει δύο φάσεις:
  - Μαρκάρισμα: βρίσκουμε όλους τους τρέχοντες συνδέσμους στο σωρό και μαρκάρουμε όλα τα μπλοκ του σωρού τα οποία δείχνονται από κάποιον από τους συνδέσμους
  - Σκούπισμα: κάνουμε ένα πέρασμα στο σωρό και επιστρέφουμε τα αμαρκάριστα μπλοκ στη λίστα με τα ελεύθερα μπλοκ
- Τα μπλοκ που μαρκάρονται στην πρώτη φάση του αλγόριθμου δε μετακινούνται (non-moving collector)

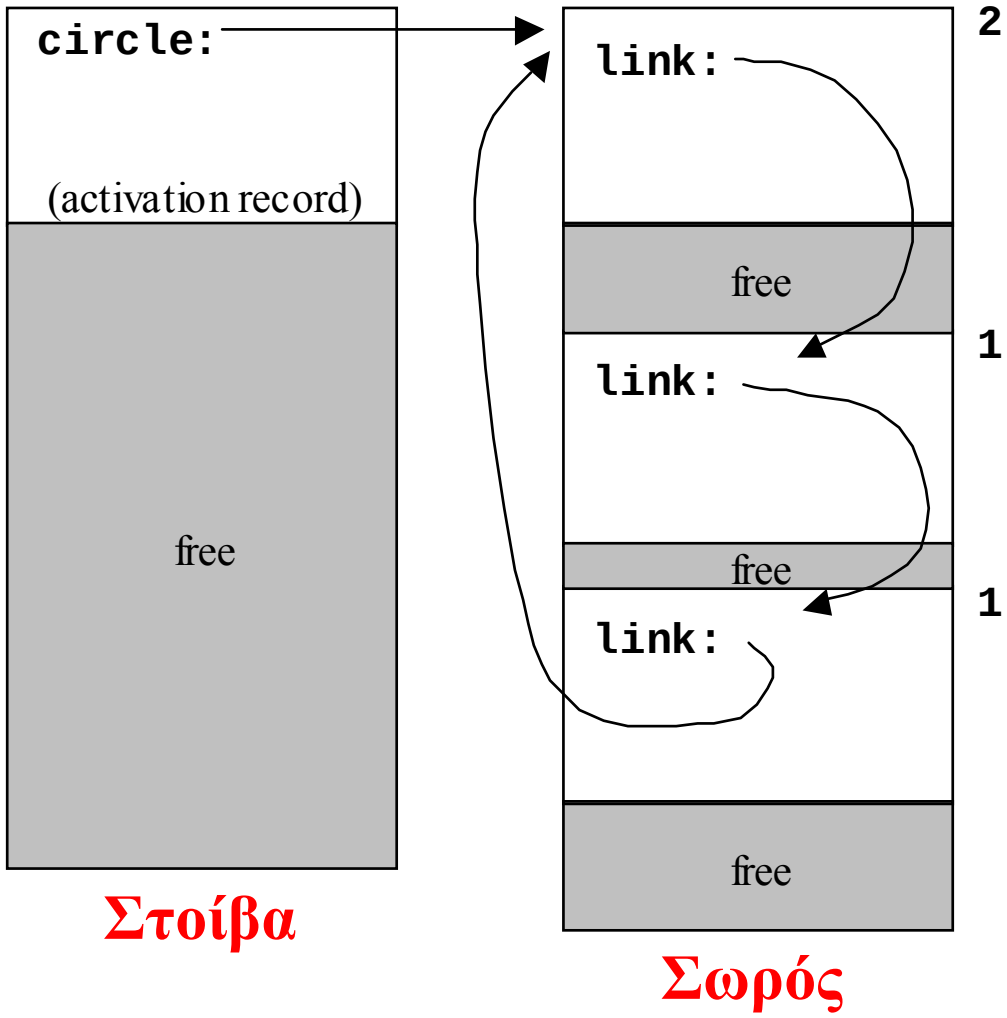
Συλλογή σκουπιδιών μέσω αντιγραφής

- Ένας συλλέκτης αντιγραφής (copying collector) διαιρεί τη μνήμη σε δύο κομμάτια και χρησιμοποιεί τη μισή μόνο μνήμη για να ικανοποιήσει τις αιτήσεις δέσμευσης
- Όταν το μισό αυτό μέρος γεμίσει, βρίσκουμε συνδέσμους που δείχνουν σε μπλοκ στο πρώτο μισό και αντιγράφουμε αυτά τα μπλοκ στο δεύτερο μισό
- Η διαδικασία αυτή συμπιέζει τα χρησιμοποιούμενα μπλοκ, με αποτέλεσμα να εξαφανίζει τον κατακερματισμό
- Συλλέκτης μετακίνησης (moving collector) μπλοκ

Μέτρημα αναφορών

- Κάθε μπλοκ έχει ένα μετρητή που κρατάει τον αριθμό των συνδέσμων σωρού που δείχνουν στο μπλοκ
- Ο μετρητής αυτός αυξάνεται όταν κάποιος σύνδεσμος προς το μπλοκ δημιουργηθεί (π.χ. μέσω ανάθεσης), και μειώνεται όταν κάποιος σύνδεσμος χαθεί
- Όταν η τιμή του μετρητή γίνει μηδέν, το μπλοκ είναι άχρηστο και μπορεί να ελευθερωθεί
- Δηλαδή στη διαχείριση μνήμης με μέτρημα αναφορών δε χρειάζεται να βρούμε δυναμικά τους τρέχοντες συνδέσμους στο σωρό

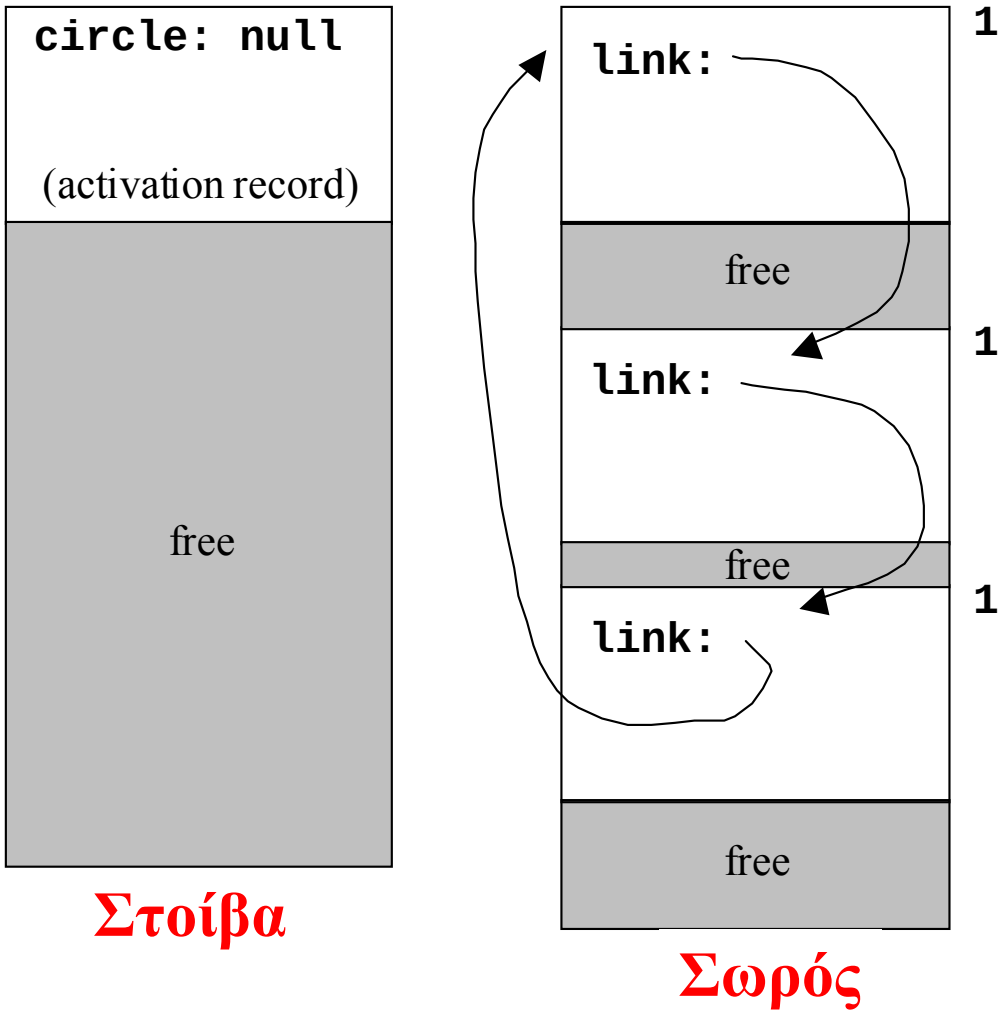
Πρόβλημα μετρήματος αναφορών



Ένα πρόβλημα μετρήματος αναφορών: δε μπορούμε να ανακαλύψουμε κύκλους από άχρηστα μπλοκ.

Στην εικόνα, μια κυκλικά συνδεδεμένη λίστα που δείχνεται από τη μεταβλητή **`circle`**.

Πρόβλημα μετρήματος αναφορών



Αν αναθέσουμε π.χ. την τιμή null στη μεταβλητή **`circle`**, η τιμή του μετρητή θα μειωθεί.

Κανένας μετρητής αναφορών δε γίνεται μηδέν, παρόλο που όλα τα μπλοκ είναι άχρηστα (σκουπίδια).

Μέτρημα αναφορών

- Δε μπορεί να συλλέξει κύκλους σκουπιδιών
- Επιβάλλει κάποιο μη αμελητέο κόστος στην εκτέλεση του προγράμματος, λόγω της ανάγκης να ενημερώσουμε τους μετρητές αναφορών σε κάθε ανάθεση
- Ένα πλεονέκτημα: ο αλγόριθμος είναι αυξητικός (incremental), δηλαδή εκτελείται από τη φύση του σε μικρά βήματα και δεν επιβάλλει μεγάλες παύσεις κατά την εκτέλεση του προγράμματος

Εκλεπτυσμένοι συλλέκτες σκουπιδιών

- Συλλέκτες γενεών (Generational collectors)
  - Η προς συλλογή μνήμη χωρίζεται σε *γενιές (generations)* με βάση την ηλικία των αντικειμένων στο μπλοκ
  - Συλλέγουμε τα σκουπίδια στις νέες γενιές πιο συχνά (χρησιμοποιώντας κάποια από τις προηγούμενες μεθόδους)
- Αυξητικοί συλλέκτες (Incremental collectors)
  - Συλλέγουν σκουπίδια σε μικρά χρονικά διαστήματα τα οποία παρεμβάλλονται με την εκτέλεση του προγράμματος
  - Κατά συνέπεια αποφεύγουν το σταμάτημα της εφαρμογής για μεγάλο χρονικό διάστημα και οι παύσεις λόγω αυτόματης ανακύκλωσης μνήμης έχουν πολύ μικρή διάρκεια

Γλώσσες με αυτόματη διαχείριση μνήμης

- Σε κάποιες γλώσσες η αυτόματη διαχείριση μνήμης είναι απαίτηση: Lisp, Scheme, ML, Haskell, Erlang, Clean, Prolog, Java, C#,...
- Κάποιες άλλες, όπως η Ada, απλώς την ενθαρρύνουν
- Τέλος, κάποιες άλλες γλώσσες όπως η C και η C++ την καθιστούν πολύ δύσκολη
  - Όμως ακόμα και για αυτές είναι δυνατή σε κάποιο βαθμό
  - Υπάρχουν βιβλιοθήκες που αντικαθιστούν τη συνηθισμένη υλοποίηση των **`malloc`**/**`free`** με ένα διαχειριστή μνήμης που χρησιμοποιεί συντηρητική συλλογή σκουπιδιών (conservative garbage collection) για την αυτόματη ανακύκλωση μνήμης

Τάσεις στην αυτόματη διαχείριση μνήμης

- Μια ιδέα από τις αρχές του 1960, η δημοτικότητα της οποίας έχει αυξηθεί σημαντικά τα τελευταία χρόνια
- Οι καλές υλοποιήσεις των συλλεκτών σκουπιδιών έχουν επίδοση παρόμοια με, ή μόνο κατά λίγο χειρότερη από, αυτήν η οποία είναι δυνατή με διαχείριση μνήμης υπό τον έλεγχο του προγραμματιστή
- Όλο και περισσότεροι προγραμματιστές συνειδητοποιούν ότι η αυτόματη ανακύκλωση μνήμης αυξάνει την παραγωγικότητά τους, οδηγεί σε λιγότερα σφάλματα, και κατά συνέπεια αξίζει τον έξτρα χρόνο εκτέλεσης

Συμπερασματικά

- Η διαχείριση μνήμης είναι ένα σημαντικό συστατικό των αποφάσεων σχεδιασμού καθώς και των συστημάτων υλοποίησης των γλωσσών προγραμματισμού
- Τόσο η επίδοση όσο και η αξιοπιστία των προγραμμάτων είναι σημαντικοί παράγοντες της ανάπτυξης λογισμικού
- Η αυτόματη διαχείριση μνήμης είναι μια από τις πιο ενεργές περιοχές ερευνητικής δραστηριότητας και πειραματισμού στην περιοχή της υλοποίησης γλωσσών προγραμματισμού