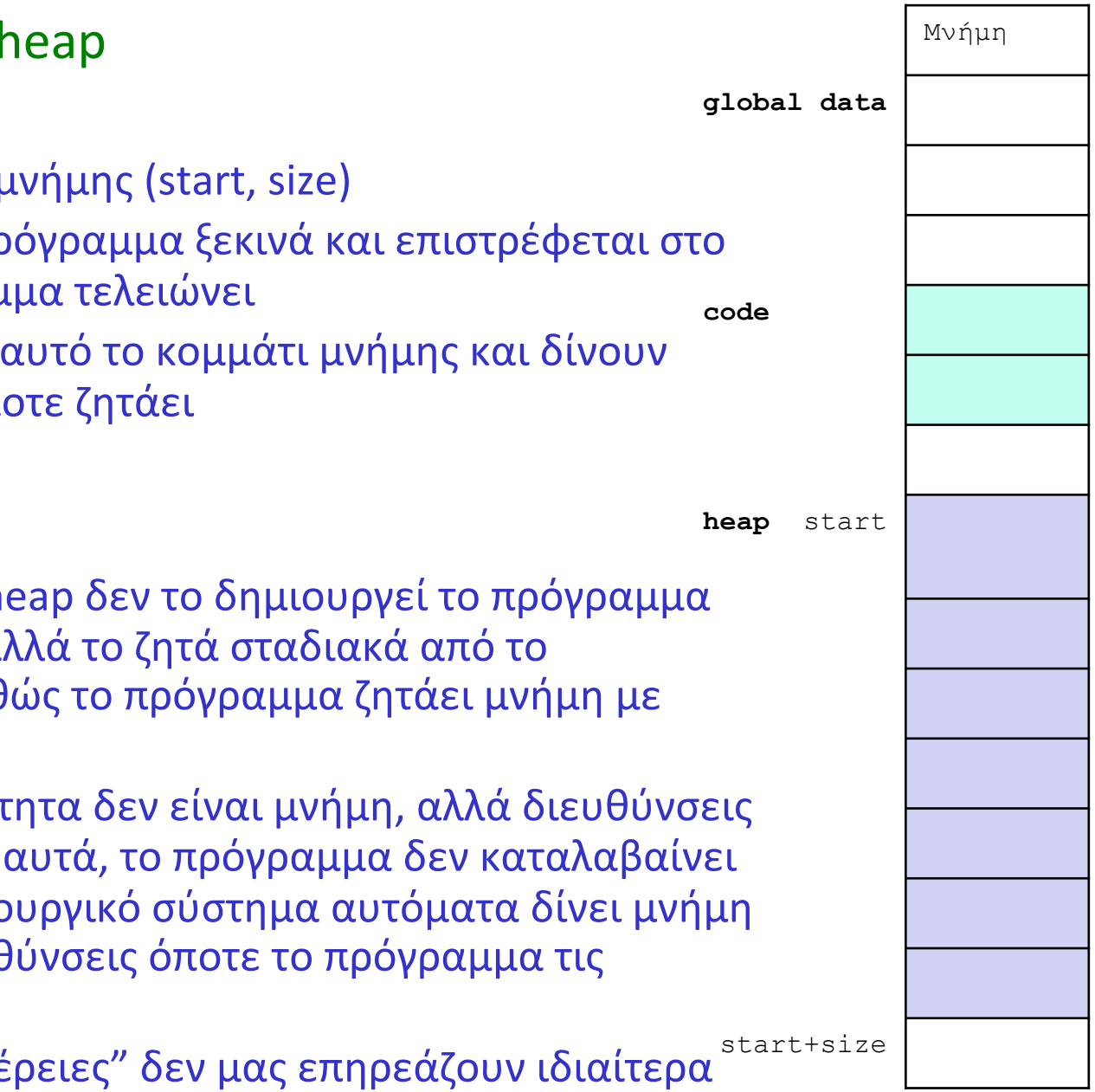
# HY255 Εργαστήριο Λογισμικού – L11

## How malloc/free works (Memory allocators)

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

# Memory layout of programs

- Μέχρι τώρα: data, code, heap
- Heap
  - Ένα συνεχόμενο κομμάτι μνήμης (start, size)
  - Γίνεται allocate όταν το πρόγραμμα ξεκινά και επιστρέφεται στο σύστημα όταν το πρόγραμμα τελειώνει
  - Η malloc/free χειρίζονται αυτό το κομμάτι μνήμης και δίνουν μνήμη στο πρόγραμμα όποτε ζητάει
  - Πως το κάνουν αυτό;
- Σημείωση
  - Στην πραγματικότητα το heap δεν το δημιουργεί το πρόγραμμα ολόκληρο στο ξεκίνημα, αλλά το ζητά σταδιακά από το λειτουργικό σύστημα, καθώς το πρόγραμμα ζητάει μνήμη με malloc/free
  - Το heap στην πραγματικότητα δεν είναι μνήμη, αλλά διευθύνσεις (address space). Παρ΄όλα αυτά, το πρόγραμμα δεν καταλαβαίνει την διαφορά γιατί το λειτουργικό σύστημα αυτόματα δίνει μνήμη (DRAM) σε αυτές τις διευθύνσεις όποτε το πρόγραμμα τις χρησιμοποιεί
  - Και οι δύο αυτές “λεπτομέρειες” δεν μας επηρεάζουν ιδιαίτερα



# Free list

- Ας πούμε ότι για να γνωρίζουμε ποιος χώρος είναι ελεύθερος στο heap η malloc/free θα διατηρεί μια λίστα που περιέχει έναν κόμβο για κάθε συνεχόμενο κομμάτι μνήμης που είναι ελεύθερο
- Αρχικά η FreeL θα περιέχει έναν κόμβο (start,size) που περιγράφει όλο το heap
- Κάθε φορά που καλείται η malloc για n bytes, διασχίζει τη FreeL, βρίσκει έναν κατάλληλο block (που να έχει αρκετό χώρο), το σπάζει σε δύο κομμάτια, ένα μεγέθους n και ένα το υπόλοιπο, επιστρέφει το πρώτο στον χρήστη και βάζει το υπόλοιπο στη FreeL
- Κάθε φορά που καλείται η free(p) παίρνουμε το block p και το επιστρέφουμε στην free list
- Αυτή η απλή σχετικά διαδικασία, τελικά προκύπτει ότι έχει σημαντική πολυπλοκότητα για το heap και είναι η δουλειά των memory allocators

# Μετά από μερικά malloc/free...

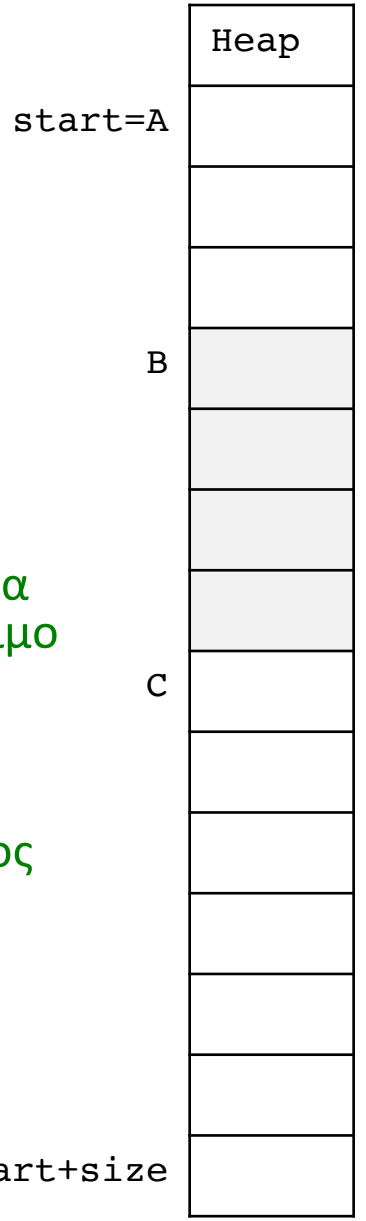
Αρχική FreeL

```
p1 = malloc(12);

p2 = malloc(16);

free(p1);
```

| FreeL | A, size | | | |
|---|---|---|---|---|
| FreeL | B, size-12 | | | |
| FreeL | C, size-28 | | | |




- Άραγε που να τοποθετήσουμε στη FreeL το ελεύθερο block? Επιλογές:
- Αν το τοποθετήσουμε στην αρχή θα είναι γρήγορο το free αλλά η λίστα θα καταλήξει να είναι unsorted (με βάση το μέγεθος) και θα χρειάζεται ψάξιμο για να βρούμε αργότερα ένα block κατάλληλου μεγέθους
- Αν το τοποθετήσουμε με sorted order (μέγεθος) τότε το free θα είναι πιο αργό, αλλά το malloc πιο γρήγορο
- Επίσης, θα μπορούσε να είναι sorted με αυξανόμενο ή μειούμενο μέγεθος
- Δυστυχώς στους allocators δεν υπάρχει ιδανική λύση για όλες τις περιπτώσεις
- Ας πούμε εμείς ότι τοποθετούμε το block που ελευθερώνεται με βάση το μέγεθος (αυξανόμενο)
- Οπότε καταλήγουμε με:

| FreeL | A, 12 | C, size-28 | | |
|---|---|---|---|---|

# Fragmentation

- Παρατηρούμε ότι καθώς εκτελούνται malloc/free operations στη μνήμη δημιουργούνται δεσμευμένα και ελεύθερα blocks
- Καθώς συνεχίζεται αυτή η διαδικασία, επειδή το μόνο που κάνουμε στη λίστα είναι να ”σπάζουμε” blocks σε μικρότερα, τα διαθέσιμα blocks θα γίνονται όλο και μικρότερα
- Αυτό ονομάζεται fragmentation της μνήμης (heap)
- Τι μπορούμε να κάνουμε;
  - (a) Θα μπορούσαμε, αν υπάρχουν blocks που είναι συνεχόμενα στη μνήμη, να τα συνενώσουμε και στην free list σε ένα μεγαλύτερο block
    - Αυτό, μπορούμε να το κάνουμε εύκολα πχ όταν ελευθερώνουμε ένα block, αν κρατάμε τη free list διατεταγμένη με βάση το μέγεθος των blcoks
  - (b) Τι γίνεται όμως αν δεν έχουμε συνεχόμενα ελεύθερα blocks; Μπορούμε να μετακινήσουμε δεσμευμένα blocks σε άλλες θέσεις στο heap και να δημιουργήσουμε συνεχόμενα ελεύθερα blocks και μετά να τα συνενώσουμε, ώστε να μειώσουμε και άλλο το fragmentation?
    - Όχι. Οι διευθύνσεις που έχει επιστρέψει η malloc χρησιμοποιούνται από το πρόγραμμα και δεν μπορούμε να μετακινήσουμε την δεσμευμένη μνήμη επειδή δεν μπορούμε να αλλάξουμε αυτούς τους pointers (δεν γνωρίζουμε που βρίσκονται).
    - Αυτό γίνεται μόνο σε γλώσσες που υποστηρίζουν garbage collection, με το όποιο κόστος
- Το fragmentation μπορεί να είναι internal ή external
  - Internal: Όταν η malloc για λόγους ταχύτητας επιστρέφει μεγαλύτερα blocks από αυτό που ζήτησε ο χρήστης
  - External: Όταν στο heap δημιουργούνται μικρά blocks που δεν μπορούν να χρησιμοποιηθούν λόγω του μικρού μεγέθους τους
- Άρα στη free list συνενώνουμε πάντα συνεχόμενα ελεύθερα blocks μνήμης

# Allocation policy

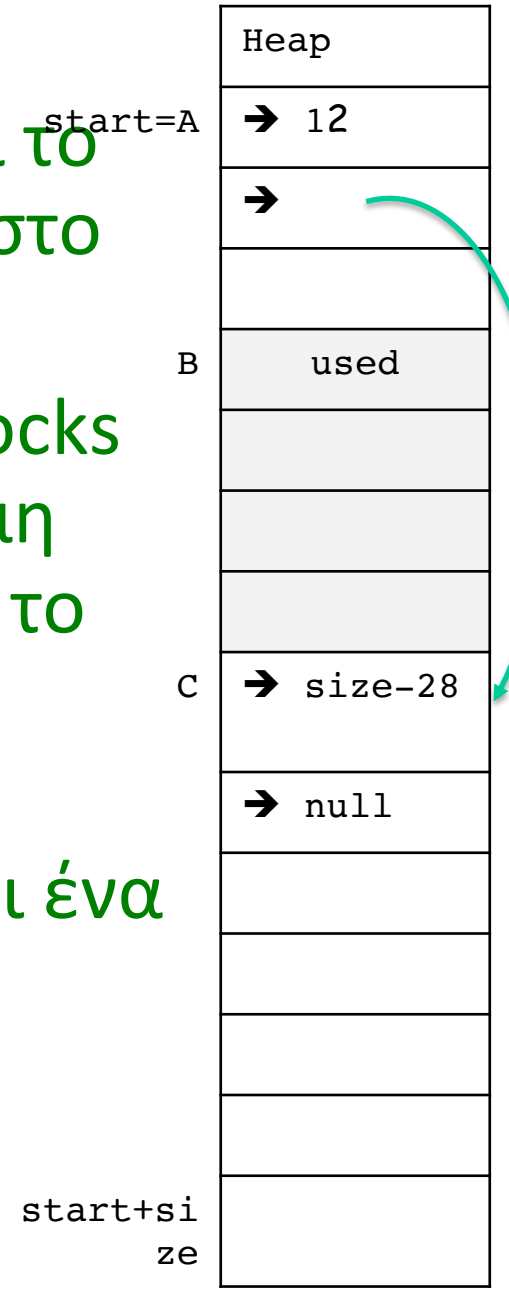
- Πως διαλέγουμε το block που θα επιστρέψουμε ή θα σπάσουμε όταν γίνεται ένα malloc; Μπορούμε να δώσουμε:
- Το πρώτο block που θα βρούμε στη free list και που έχει αρκετό χώρο ➔ first fit
  - Με το σκεπτικό να είμαστε όσο πιο γρήγοροι γίνεται
- Το block που έχει το μικρότερο μέγεθος και αρκεί για την malloc ➔ best fit
  - Με το σκεπτικό να μειώσουμε το fragmentation
- Το μεγαλύτερο block που έχουμε ➔ worst fit
  - Με το σκεπτικό να μειώσουμε πάλι το fragmentation αφήνοντας πίσω μας μεγάλα κομμάτια του heap ελεύθερα
- Το καλύτερο block, αλλά ξεκινώντας από κει που σταματήσαμε ➔ next fit
- Και πάλι δεν υπάρχει ιδανική πολιτική για όλες τις περιπτώσεις
  - Γενικά το best fit θεωρείται πιο αποτελεσματικό και πιο ασφαλής επιλογή
- Αυτή η ισορροπία ανάμεσα σε ταχύτητα και χαμένο χώρο είναι ο κύριος στόχος όλων των allocators

# Το μυστήριο της FreeL

- Η FreeL είναι ας πούμε μια σχετικά απλή δομή γιατί πρέπει να είναι γρήγορη και να μην καταναλώνει χώρο
- Που βρίσκεται όμως;
- 1. Για να δημιουργήσουμε μια λίστα, πρέπει να έχουμε στη διάθεσή μας την malloc/free ώστε να παίρνουμε δυναμικά χώρο -- Ωστόσο, εμείς τώρα προσπαθούμε να υλοποιήσουμε την malloc/free
- 2. Που θα την τοποθετήσουμε; Δεν μπορεί να είναι στο global data segment γιατί θα πρέπει να έχει σταθερό μέγεθος, άρα πρέπει και η free list να βρίσκεται μέσα στο heap
- Πως λοιπόν χειριζόμαστε την free list μέσα στο heap χωρίς να έχουμε ακόμη τα malloc/free calls;

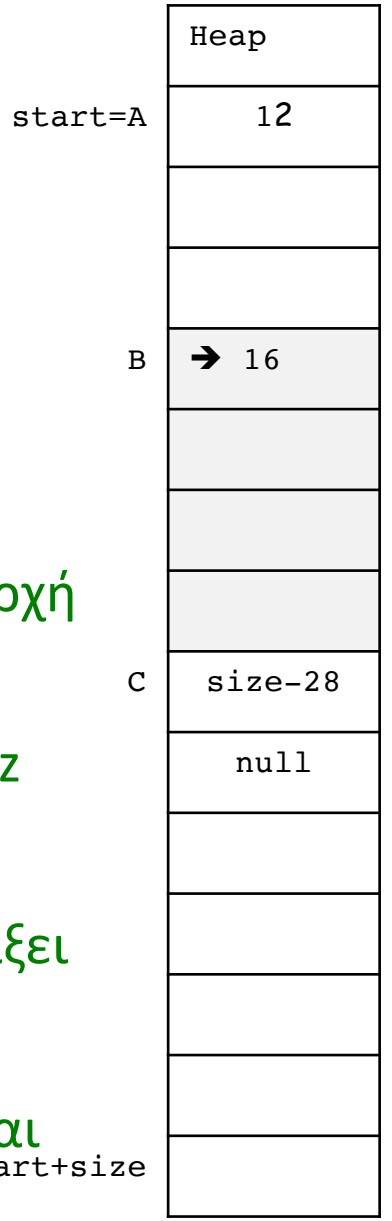
# Free block headers

- Σε κάθε free block μέσα στο heap δημιουργούμε έναν header που περιέχει το μέγεθος του free block και έναν pointer στο επόμενο block
- Δεν μας πειράζει αυτό μια και τα free blocks είναι εγγυημένο ότι είναι ελεύθερη μνήμη και απαγορεύεται να τα χρησιμοποιήσει το πρόγραμμα
- Οπότε, στο διπλανό σχήμα έχουμε δύο ελεύθερα blocks (A,12) και (C,size-28) και ένα δεσμευμένο (B,16)
- Οπότε η free list θα έχει την μορφή που δείνχει το σχήμε με 2 κόμβους



# Το μυστήριο του free operation

- Όταν κάνουμε free(p), πως γνωρίζουμε το μέγεθος sz του used block που δείχνει το p ώστε να το προσθέσουμε στην free list σαν (p, sz)
- Η free δεν μας δίνει αυτή την πληροφορία
- Η free list έχει μόνο τα ελεύθερα blocks και από αυτά δεν μπορούμε να βρούμε το μέγεθος των used blocks, πχ στην περίπτωση που έχουμε δύο διαδοχικά used blocks και ελευθερώνεται το πρώτο από αυτά
- Λύση: Προσθέτουμε headers και στα used blocks
- Σε κάθε block b που επιστρέφει η malloc δεσμεύει λίγο περισσότερο χώρο (ένα επιπλέον word) και προσθέτει στην αρχή του block μια λέξη με το μέγεθός του. Μετά επιστρέφει στον χρήστη τον pointer p=b+4 που δείχνει στην πρώτη ελεύθερη θέση του block, και που δείχνει σε ένα block με τουλάχιστον sz χώρο
- Το πρόγραμμα μετά από μια p=malloc(sz) επιτρέπεται να χρησιμοποιήσει μόνο τα bytes [p, p+sz-1], οπότε δεν θα πειράξει ποτέ τον header του used block
- Όταν το πρόγραμμα καλέσει την free(p), η free μπορεί να κοιτάξει στη θέση b=p-4 για να βρει το μέγεθος sz του block και να προσθέσει στη free list τον κόμβο (b, sz)

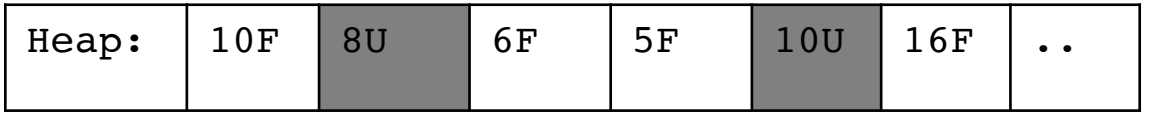


# Malloc/free

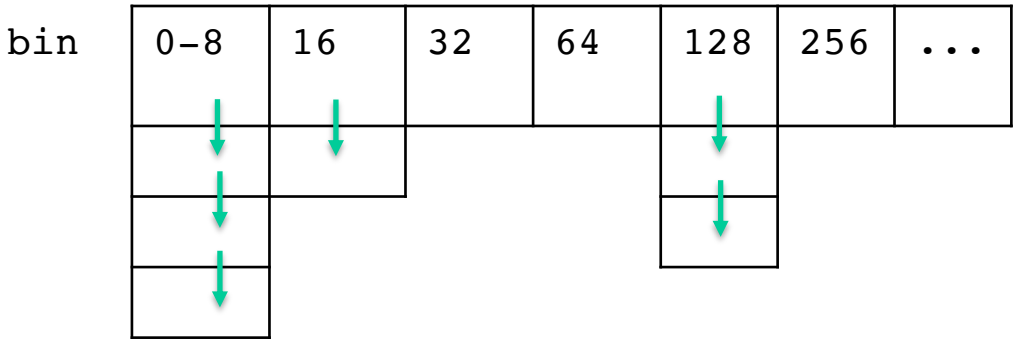
- Βλέπουμε τώρα πόσο “ευαίσθητη” είναι η εσωτερική δομή και χρήση του heap
- Χρήση unallocated μνήμης
  - Αν γράψουμε μια θέση που δεν την έχουμε κάνει allocate σωστά ➔ μπορεί να καταστρέψουμε την free list και φυσικά μετά θα ακολουθήσουν πολλά προβλήματα
  - Αν διαβάσουμε μια θέση που δεν την έχουμε κάνει allocate σωστά, μπορεί να διαβάσουμε οτιδήποτε
- Χρήση μνήμης που έχουμε κάνει free
  - Εξίσου κακό με την προηγούμενη περίπτωση
- Free unallocated διεύθυνσης
  - Αν κάνουμε free ένα p που δεν το επέστρεψε η malloc, δεν θα έχει τον σωστό header ➔ θα ακολουθήσουν πολλά προβλήματα
- κ.ο.κ.
- Τέτοια προβήματα είναι πολύ δύσκολο να τα βρούμε σε μεγάλα συστήματα/προγράμματα και δεν υπάρχουν αυτόματα εργαλεία
  - Ο gdb, variants της malloc που κρατούν πληροφορίες, και το valgrind βοηθούν, αλλά δεν μπορούν να βρουν όλα τα σχετικά λάθη (ούτε καν πολλά σε ρεαλιστικά προγράμματα)
- Οπότε χρησιμοποιούμε την δυναμική μνήμη πάντα με εξαιρετική προσοχή και επιμέλεια!

# Free list variations

- (1) Η μέθοδος που περιγράψαμε είναι μια explicit λίστα με pointers στο επόμενο στοιχείο. Συνήθως είναι μια διπλή λίστα που μπορούμε να την ψάξουμε και προς τις δύο κατευθύνσεις και περιέχει μόνο τα free blocks ώστε να ψάχνουμε λιγότερα blocks για πιο γρήγορο το allocation
- (2) Μια άλλη μέθοδος είναι μια implicit λίστα που τη χειριζόμαστε μόνο μέσω των μεγεθών των blocks ως εξής:




  - F, U = ένα bit στο μέγεθος που δηλώνει αν το block είναι free ή used
  - Για να διατρέξουμε τη λίστα βρίσκουμε το επόμενο block από το μέγεθος του τρέχοντος block. Σε αυτή την περίπτωση πρέπει να διατρέχουμε και τα free και τα used blocks
  - Πολλές φορές το μέγεθος το γράφουμε στην αρχή και στο τέλος κάθε block (header + footer) ώστε να μπορούμε να διατρέξουμε την λίστα και προς τις δύο κατευθύνσεις (ο footer ονομάζεται και boundary tag).
  - Η implicit λίστα κάνει πολύ γρήγορο το coalescing, αφού κατά το free εξετάζουμε απλά τα γειτονικά blocks.
- (3) Μια Τρίτη λύση είναι μια explicit λίστα που συνδυάζεται με μια τεχνική δοχείων “binning”
  - Σε αυτή την τεχνική ορίζουμε δοχεία/bins που περιέχουν blocks με συγκεκριμένο εύρος μεγέθους, π.χ. bins για μεγέθη: π.χ. [1-8], [9-16], [17-32], ... με εύρος που διπλασιάζεται




  - Το binning βοηθάει πολύ στο να βρίσκουμε γρήγορα ένα block κατά το malloc
- Η malloc της libc (που προέρχεται από τον allocator του Doug Lea) και ο SLAB allocator του linux kernel (και η παραλλαγή του, ο SLUB) χρησιμοποιούν συνδιασμούς αυτών των τεχνικών

# Σχόλια

- Οι memory allocators είναι εξαιρετικά σημαντικοί στην απόδοση των προγραμμάτων και συστημάτων
- Σε γλώσσες που υποστηρίζουν garbage collection, η πολυπλοκότητα είναι τάξεις μεγέθους μεγαλύτερη (και δυστυχώς και τα overheads είναι σημαντικά)
- Το δυναμικό memory management γενικά παρ' ότι γενικά μια απλή έννοια, έχει εξαιρετικά υψηλή πολυπλοκότητα και βρίσκεται στην καρδιά των περισσότερων προβλημάτων που σχετίζονται με απόδοση
- Καθώς οι μνήμες των συστημάτων αλλάζουν από τεχνολογίες DRAM σε NVM (non-volatile memories) που έχουν πολύ μεγαλύτερη χωρητικότητα, πολλά πράγματα φαίνεται ότι θα αλλάξουν στο μέλλον στο memory management!

# Reading

- Bryant Ch. 9.{9, 11}