Δομές Δεδομένων

16η Διάλεξη

Κατακερματισμός

Ε. Μαρκάκης

Περίληψη

- Συναρτήσεις κατακερματισμού
- Χωριστή αλυσίδωση
- Γραμμική διερεύνηση
- Διπλός κατακερματισμός
- Δυναμικός κατακερματισμός
- Προοπτική

Συναρτήσεις κατακερματισμού

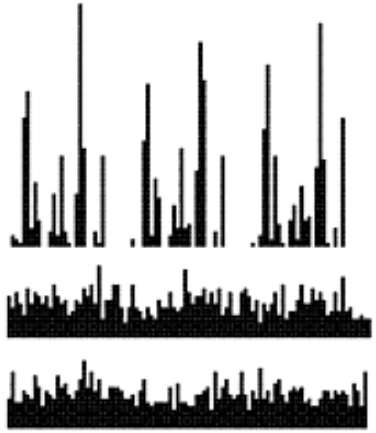
- Προηγούμενο μάθημα: h(k) = k mod M
- Εναλλακτική μέθοδος υπολοίπου
  - h(k) = floor(k*a) mod M
  - Με αυτή τη μέθοδο ο M δεν χρειάζεται πλέον να είναι πρώτος
  - Αρκεί να επιλέξουμε το κατάλληλο a!
    - Μία καλή επιλογή είναι φ = 0,618033
    - Ο φ είναι η χρυσή τομή (golden ratio)
- Παράδειγμα
  - Ακέραια κλειδιά των 16 bit σε δεκαδική μορφή
  - Με h(k) = k % 97 έχουμε καλή κατανομή
  - Με h(k) = k % 100 έχουμε κακή κατανομή
    - Ουσιαστικά χρησιμοποιούμε 2 δεκαδικά ψηφία
  - Με h(k) = (int) (φ * k) % 100 έχουμε καλή κατανομή
    - Το φ αναιρεί τα προβλήματα του 100

16838	57	38	6
5758	35	58	58
10113	25	13	50
17515	55	15	24
31051	11	51	90
5627	1	27	77
23010	21	10	20
7419	47	19	85
16212	13	12	19
4086	12	86	25
2749	33	49	98
12767	60	67	90
9084	63	84	14
12060	32	60	53
32225	21	25	16
17543	83	43	42
25089	63	89	5
21183	37	83	91
25137	14	37	35
25566	55	66	0
26966	0	66	65
4978	31	78	76
20495	28	95	66
10311	29	11	72
11367	18	67	25

Συναρτήσεις κατακερματισμού

- Σε πολλές περιπτώσεις, τα κλειδιά δεν είναι αριθμοί αλλά strings
- Αλφαριθμητικά κλειδιά μεγάλου μήκους
- Μπορούμε να τα γράψουμε σε δυαδική μορφή (7 bits για κάθε χαρακτήρα)
- Π.χ. now → $110\ (128)^2 + 111(128)^1 + 119\ (128)^0$
- Για μεγάλα κλειδιά όμως θα έχουμε υπερχείλιση
  - Αν $k = k_n k_{n-1} \ldots\ k_3\, k_2\, k_1\, k_0 \rightarrow 128^0 \cdot k_0 + 128^1 \cdot k_1 + 128^2 \cdot k_2 + 128^3 \cdot k_3 + \ldots$
  - Ξαναγράφουμε σύμφωνα με τον αλγόριθμο του Horner (δείτε Κεφ. 4)
    - $(\ldots(((k_n \cdot 128 + k_{n-1}) \cdot 128 + k_{n-2}) \cdot 128 + k_{n-3})\ldots) \cdot 128 + k_1)128 + k_0$

Συναρτήσεις κατακερματισμού



- Ιδέα: Υπολογίζουμε το υπόλοιπο σε κάθε βήμα (μετά τον πολλαπλασιασμό και την πρόσθεση)
  - Δεν αλλάζει το τελικό αποτέλεσμα

```
static int hash(String s, int M) {
  int h = 0, a = 127;
  for (int i = 0; i < s.length(); i++)
       h = (a*h + s.charAt(i)) % M;
  return h; }
```

- Προσοχή: στο πρόγραμμα του βιβλίου πολλαπλασιάζουμε με 127 αντί για 128
  - Έτσι μπορούμε να έχουμε $M = 2^N$ χωρίς προβλήματα

Συναρτήσεις κατακερματισμού

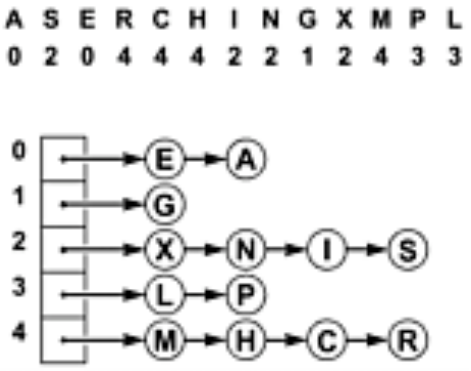
- Καθολικός κατακερματισμός
  - Βασίζεται στην παραπάνω μέθοδο
  - Η ιδέα είναι να ελαχιστοποιήσουμε την πιθανότητα σύγκρουσης κλειδιών
  - Χρήση τυχαίων παραγόντων για κάθε ψηφίο
    - Συνήθως οι αριθμοί είναι ψευδοτυχαίοι
    - Παράγονται από γεννήτρια τυχαίων αριθμών
  - Πιθανότητα σύγκρουσης 1/M για πίνακα μεγέθους M

```
static int hashU(String s, int M) {
  int h = 0, a = 31415, b = 27183;
  for (int i = 0; i < s.length(); i++) {
       h = (a*h + s.charAt(i)) % M;
       a = a*b % (M-1); }
  return h; }
```

    - Η γεννήτρια τυχαίων αριθμών είναι μέσα στον κώδικα

Χωριστή αλυσίδωση

- Πώς διαχειριζόμαστε τις συγκρούσεις?
- Διαχείριση συγκρούσεων με λίστες
  - Σε κάθε θέση του πίνακα ξεκινάει μία λίστα
  - Τα αντικείμενα που συγκρούονται τοποθετούνται στη λίστα
  - Διατεταγμένη ή μη διατεταγμένη λίστα
    - Συνήθως μη διατεταγμένη λόγω μικρού μήκους
  - Πιθανόν μεγάλη σπατάλη χώρου
    - Κάθε θέση απαιτεί έναν δείκτη σε λίστα
- Κόστος πράξεων
  - Εισαγωγή: σταθερό κόστος σε μη διατεταγμένη λίστα
    - Η εισαγωγή γίνεται στην αρχή
  - Αναζήτηση: N/M κατά μέσο όρο για N στοιχεία σε M λίστες
    - Το μέσο μήκος των λιστών είναι N/M
    - Μπορεί βέβαια μία να έχει N στοιχεία και όλες οι άλλες κανένα! (πολύ μικρή πιθανότητα για κάτι τέτοιο)



Χωριστή αλυσίδωση

- Πόσο μεγάλες είναι οι λίστες;
  - Είναι σχεδόν πάντα πολύ κοντά στο N/M! (αν είναι καλή η συνάρτηση κατακερματισμού)
- Υλοποίηση χωριστής αλυσίδωσης

```
private Node[] heads;
private int N, M;
ST(int maxN) {
  N = 0; M = maxN/5; heads = new Node[M]; }
ITEM search(KEY key) {
  return searchR(heads[hash(key, M)], key); }
//αναδρομική αναζήτηση σε αταξινόμητη λίστα (την είδαμε στο Κεφ. 12)
void insert(ITEM x) {
  int i = hash(x.key(), M);
  heads[i] = new Node(x, heads[i]); //εισαγωγή στην
  αρχή
  N++; }
```

Χωριστή αλυσίδωση

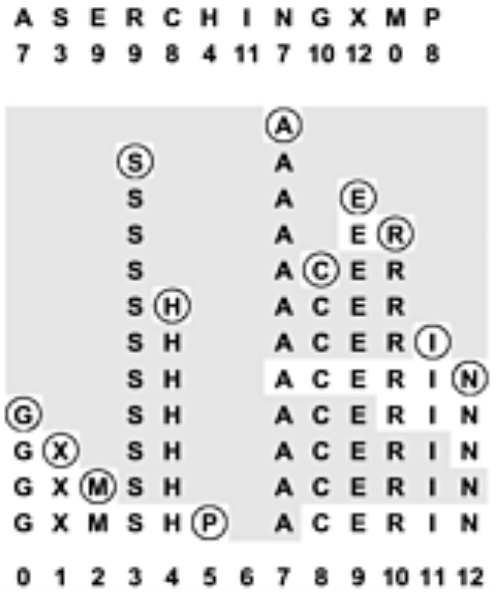
- Ιδιότητα 1: Η πιθανότητα μία λίστα να έχει μήκος πολύ κοντά στο N/M μετά την εισαγωγή N στοιχείων τείνει στο 1
  - Απόδειξη με χρήση Poisson approximation
  - Τέτοιου είδους προβλήματα αναφέρονται συνήθως ως occupancy problems
- Ιδιότητα 2: Κατά μέσο όρο η πρώτη σύγκρουση συμβαίνει όταν έχουμε εισάγει περίπου $1.25\sqrt{M}$ στοιχεία
  - Birthday paradox!
- Ιδιότητα 3: Οι λίστες ουσιαστικά συμπεριφέρονται σαν στοίβες (αφού κάνουμε εισαγωγή στην αρχή)
  - Πολύ χρήσιμη ιδιότητα για τον πίνακα συμβόλων ενός compiler

Χωριστή αλυσίδωση

- Ανταγωνισμός χώρου-χρόνου
  - Ο παράγοντας N/M παίζει κεντρικό ρόλο
  - Αυξάνοντας το M μειώνεται το μήκος των λιστών
    - Ταχύτερη αναζήτηση, αλλά περισσότερος χώρος για δείκτες
  - Στην πράξη διαλέγουμε M = N/5 ή N/10
    - Όταν το N μεταβάλλεται χρησιμοποιούμε δυναμικούς πίνακες
- Χειρισμός διπλών κλειδιών
  - Αν επιτρέπονται, η εισαγωγή δεν αλλάζει
    - Στην αναζήτηση πρέπει να ψάχνουμε όλη τη λίστα
  - Αν δεν επιτρέπονται, η αναζήτηση δεν αλλάζει
    - Στην εισαγωγή πρέπει να ψάχνουμε όλη τη λίστα
- Μειονεκτήματα κατακερματισμού
  - Ακατάλληλος για επιλογή k-οστού και ταξινόμηση

Γραμμική διερεύνηση

- Κατακερματισμός ανοιχτής διευθυνσιοδότησης
  - Αντίθετη προσέγγιση από τη χωριστή αλυσίδωση
  - Διαχείριση συγκρούσεων χωρίς λίστες
  - Τα στοιχεία τοποθετούνται μέσα στον πίνακα
    - Πρέπει να ισχύει πάντα ότι M >= N
  - Διάφορες μέθοδοι εντοπισμού μιας θέσης
- Γραμμική διερεύνηση
  - Χρήση της πρώτης κενής θέσης
    - Ξεκινάμε στο σημείο που μας λέει η συνάρτηση
    - Σαρώνουμε κυκλικά τον πίνακα (με φορά προς τα δεξιά)
  - Εισαγωγή: εισάγουμε το στοιχείο στο πρώτο κενό
    - Μπορεί να είναι η αρχική θέση αν δεν έχουμε σύγκρουση
  - Αναζήτηση: επιτυχία αν βρούμε το στοιχείο
    - Αποτυχία αν φτάσουμε σε κενή θέση



Γραμμική διερεύνηση

- Υλοποίηση γραμμικής διερεύνησης

```
private ITEM[] st; private int N, M;
ST(int maxN) { N = 0; M = 2*maxN; st = new ITEM[M]; }
void insert(ITEM x) {
  int i = hash(x.key(), M);
  while (st[i] != null) i = (i+1) % M;
  st[i] = x;
  N++; }
ITEM search(KEY key) {
  int i = hash(key, M);
  while (st[i] != null)
       if (equals(key, st[i].key())) return st[i];
       else i = (i+1) % M;
  return null; }
```

Γραμμική διερεύνηση

- Ανταγωνισμός χώρου-χρόνου
  - Ο παράγοντας α = N/M παίζει πάλι κεντρικό ρόλο
  - Εδώ όμως α < 1 ενώ στην αλυσίδωση α > 1
    - Το α εδώ ονομάζεται συντελεστής πλήρωσης (load factor)
  - Όσο το α προσεγγίζει το 1 τόσο επιβραδύνονται οι πράξεις
    - Τόσο όμως μειώνεται η επιβάρυνση σε χώρο
  - Το κόστος εξαρτάται από τα clusters στον πίνακα
    - Cluster (συστοιχία) είναι μία ακολουθία συνεχόμενων μη κενών θέσεων
  - Μέσο κόστος πράξεων (Knuth 1962)
    - Επιτυχής αναζήτηση: $(1+1/(1-\alpha))/2$
    - Ανεπιτυχής αναζήτηση: $(1+1/(1-\alpha)^2)/2$
    - Πολύ καλή προσέγγιση όσο το α είναι μακριά από το 1
  - Στην πράξη προσπαθούμε να κρατάμε το α κοντά στο ½
    - Τα παραπάνω κόστη γίνονται 1,5 και 2,5

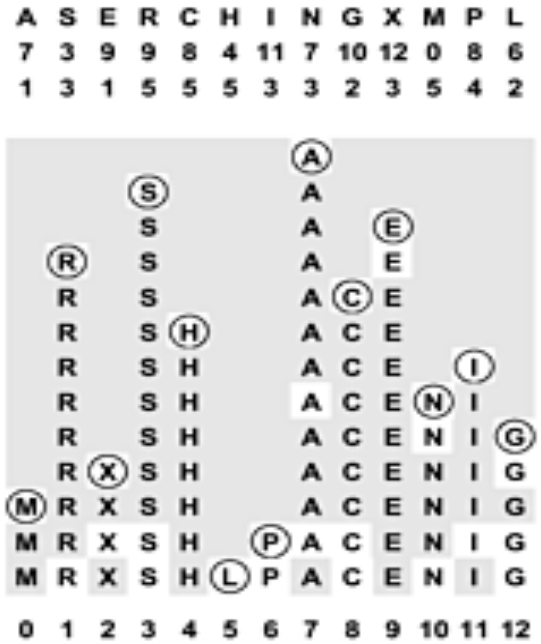
Γραμμική διερεύνηση

- Χειρισμός διπλών κλειδιών
  - Παρόμοιος με αυτόν της χωριστής αλυσίδωσης
- Αφαίρεση κλειδιών: αφήνει κενό στη συστοιχία!
  - Μπορεί να προκαλέσει λάθη σε μελλοντικές αναζητήσεις
  - Επανεισαγωγή των επόμενων κλειδιών της συστοιχίας

```
public void remove(ITEM x) {
  int i = hash(x.key(), M);
  while (st[i] != null)
       if (equals(x.key(), st[i].key())) break;
       else i = (i+1) % M;
  if (st[i] == null) return;
  st[i] = null; N--;
  for (int j = i+1; st[j] != null; j = (j+1) % M) {
       x = st[j]; st[j] = null;
       insert(x); N--; } }
```

Διπλός κατακερματισμός

- Παραλλαγή της γραμμικής διερεύνησης
  - Η γραμμική διερεύνηση δημιουργεί εύκολα μεγάλα clusters
  - Χρήση δεύτερης συνάρτησης κατακερματισμού g
  - Εξετάζουμε τα στοιχεία με βήμα αναζήτησης το g(k)
    - Στο γραμμικό κατακερματισμό το βήμα είναι 1
- Επιλογή της δεύτερης συνάρτησης
  - g(k) > 0 για να μην έχουμε άπειρο βρόχο
  - g(k) πρώτο σε σχέση με το M
    - Με αυτό τον τρόπο εξετάζουμε όλες τις θέσεις
  - Παράδειγμα: (k % 97) +1
- Μέσο κόστος πράξεων (Guibas & Szemeredi)
  - Επιτυχής αναζήτηση: ln(1/(1-α))/α
  - Ανεπιτυχής αναζήτηση: 1/(1-α)



Διπλός κατακερματισμός

- Υλοποίηση διπλού κατακερματισμού

```
void insert(ITEM x) {
  KEY key = x.key();
  int i = hash(key, M);
  int k = hashtwo(key);
  while (st[i] != null) i = (i+k) % M;
  st[i] = x; N++; }
ITEM search(KEY key) {
  int i = hash(key, M);
  int k = hashtwo(key);
  while (st[i] != null)
       if (equals(key, st[i].key())) return st[i];
       else i = (i+k) % M;
  return null; }
```

Διπλός κατακερματισμός

- Κέρδος χρόνου για σταθερή επιβάρυνση μνήμης
  - Για α = 0,5 τα παραπάνω κόστη είναι 1,4 και 1,5
  - Στη γραμμική διερεύνηση ήταν 1,5 και 2,5
- Κέρδος μνήμης για σταθερό κόστος χρόνου
  - Έστω ότι θέλουμε το πολύ t διερευνήσεις ανά πράξη
  - Οι ανεπιτυχείς αναζητήσεις είναι πάντα πιο αργές
  - Στη γραμμική διερεύνηση πρέπει α < 1-1/sqrt(t)
  - Στο διπλό κατακερματισμό πρέπει α < 1-1/t
  - Για t=10 οι τιμές είναι α = 68% και α = 90%
- Αφαίρεση στοιχείων
  - Δεν λειτουργεί η μέθοδος της γραμμικής διερεύνησης
    - Δεν ξέρουμε ποια στοιχεία πέρασαν από το αφαιρούμενο!
  - Χρήση ειδικών στοιχείων φρουρών αντί για κενά

Δυναμικός κατακερματισμός

- Χειρισμός δυναμικού πλήθους κλειδιών
  - Προσπαθούμε να διατηρήσουμε ένα κατάλληλο α
  - Αλλαγή μεγέθους πίνακα ανάλογα με τα κλειδιά
  - Παράδειγμα με γραμμική διερεύνηση
    - Όταν το α περάσει το 50% διπλασιάζουμε τον πίνακα
    - Όταν το α πέσει κάτω από 12,5% υποδιπλασιάζουμε τον πίνακα
    - Διατήρηση του α ανάμεσα στο 25% και το 50%
    - Χρησιμοποιούμε διαφορετικά όρια για να αποφύγουμε προβλήματα
  - Τα κλειδιά επανεισάγονται σε κάθε αλλαγή
    - Κάποιες πράξεις κοστίζουν πολύ περισσότερο από τις άλλες
    - Το μέσο κόστος όμως είναι σταθερό!
    - Μετά από αλλαγή μεγέθους ο πίνακας έχει N/4 κλειδιά
    - Θα διπλασιαστεί μετά από N/4 τουλάχιστον εισαγωγές
    - Θα υποδιπλασιαστεί μετά από N/8 τουλάχιστον διαγραφές

Δυναμικός κατακερματισμός

- Υλοποίηση δυναμικού κατακερματισμού
  - Ο υποδιπλασιασμός είναι παρόμοιος με τον διπλασιασμό

```
private ITEM[] st;
private int N, M;
ST(int maxN) {
  N = 0; M = 4; st = new ITEM[M]; }
private void expand() {
  ITEM[] t = st; N = 0; M = M+M; st = new ITEM[M];
  for (int i = 0; i < M/2; i++)
       if (t[i] != null) insert(t[i]); }
void insert(ITEM x) {
  int i = hash(x.key(), M);
  while (st[i] != null) i = (i+1) % M;
  st[i] = x;
  if (N++ >= M/2) expand(); }
```

Προοπτική

- Γραμμική διερεύνηση ή διπλός κατακερματισμός;
  - Η γραμμική διερεύνηση είναι ταχύτερη αλλά ο διπλός κατακερματισμός είναι πιο οικονομικός σε μνήμη
  - Απαιτεί όμως τον υπολογισμό μιας δεύτερης συνάρτησης
- Χωριστή αλυσίδωση ή ανοιχτή διευθυνσιοδότηση;
  - Η χωριστή αλυσίδωση παρέχει απλές υλοποιήσεις
  - Εξαρτάται όμως και από το κόστος των συνδέσμων
  - Έστω αλυσίδωση με M λίστες των 4 στοιχείων
    - Έστω ότι κάθε σύνδεσμος απαιτεί μία λέξη μηχανης
    - Το κόστος μνήμης είναι 9M λέξεις (4M στοιχεία + 5M δείκτες)
    - Μέσος χρόνος επιτυχούς αναζήτησης: 2 διερευνήσεις (μισή λίστα)
  - Έστω γραμμική διερεύνηση σε πίνακα 9M θέσεων (στοιχεία)
    - Έχουμε α = 4M στοιχεία / 9M θέσεις = 4/9
    - Μέσος χρόνος επιτυχούς αναζήτησης: (1+1/(1-4/9))/2 = 1,4
    - Για 2 διερευνήσεις αρκεί να έχουμε πίνακα 6M