Διάλεξη 35: Τεχνικές Κατακερματισμού (Hashing)

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Ανασκόπηση Προβλήματος και Προκαταρκτικών Λύσεων

Bit-Διανύσματα

- Τεχνικές Κατακερματισμού & Συναρτήσεις Κατακερματισμού
- Διαχείριση Συγκρούσεων με Αλυσίδωση & με ανοικτή διεύθυνση

Διδάσκων: Παναγιώτης Ανδρέου

Εισαγωγή – Το πρόβλημα

Το Πρόβλημα

- Έχουμε ένα στοιχείο **u**. Θέλουμε κάποια αποδοτική δομή η οποία θα μας επιτρέψει να εκτελέσουμε τις ακόλουθες δυο πράξεις σε σταθερό χρόνο O(1).
- **Εύρεση** στοιχείου **u** σε μια συλλογή **S (**δηλ. **Find(u,S)).**
- **Εισαγωγή** του **u** στην συλλογή S **(**δηλ. **Insert(u, S))**

Παράδειγμα

- Η συλλογή S περιέχει μια μεγάλη λίστα φοιτητών. Θέλουμε να βρούμε αν ο u="Νεόφυτος Χαραλάμπους" είναι μέρος αυτής της λίστας.
- Αν δεν είναι στην λίστα, τότε θέλουμε να τον εισάγουμε.

Εισαγωγή – Ανασκόπηση Λύσεων

Ακατάλληλες Υλοποιήσεις

1. ***<u>Συνδεδεμένη λίστα</u>***

   Insert: O(1), Find: O(n)

2. ***<u>Ισοζυγισμένο δένδρο αναζήτησης</u>***

   Insert: $O(\log_m n)$, Find: $O(\log_m n)$

   n: ο αριθμός των **κόμβων** και

   m: ο **βαθμός** (branching factor) κάθε κόμβου.

Εισαγωγή – Ανασκόπηση Λύσεων ΙΙ

- **Υπάρχει πιο αποδοτική μέθοδος από τη χρήση ισοζυγισμένων δένδρων;**

- **Ναι,** δεδομένου ότι υπάρχει μια συνάρτηση η οποία μας επιτρέπει για κάθε στοιχείο **u**, να βρούμε την ακριβή θέση του **u** στον πίνακα.

- Έχουμε χρησιμοποιήσει αντίστοιχη ιδέα στον αλγόριθμο ταξινόμησης BucketSort.

- Μια απλή λύση είναι να απεικονίσουμε το σύνολο $S \subseteq U$ (όπου U είναι το πεδίο ορισμού της S) με ένα **διάνυσμα διφίων (διάνυσμα δυαδικών ψηφίων, *bit-vector*).**

- Παρόμοια δομή χρησιμοποιήσαμε και στον αλγόριθμο ταξινόμησης bucketsort.

Bit vectors (Διανύσματα Διφύων)

- Ένα **bit-vector** είναι μονοδιάστατος πίνακας με n bits, Bits[1..n], τέτοιος ώστε:

  Bits[u] = 1, αν u $\in$ S και Bits[u] = 0, αν u $\notin$ S

- Για παράδειγμα αν U = {1,…,9} τότε το σύνολο S = {1 , 3 , 7} αναπαρίσταται ως το bit-vector:

  Bits = [1,0,1,0,0,0,1,0,0]

- Ο χρόνος εύρεσης και εισαγωγής / αναζήτησης κάποιου στοιχείου είναι σε χρόνο O(1)!

- **Πρόβλημα**: Αν το |U| είναι πολύ μεγαλύτερο από το |S| τότε σπαταλάμε πάρα πολύ χώρο!

- Λύση: ο **Κατακερματισμός (hashing)** που είναι μια οικογένεια μεθόδων που αντιστοιχεί ένα κλειδί σε μία θέση ενός πίνακα (*key-to-address transformation).*

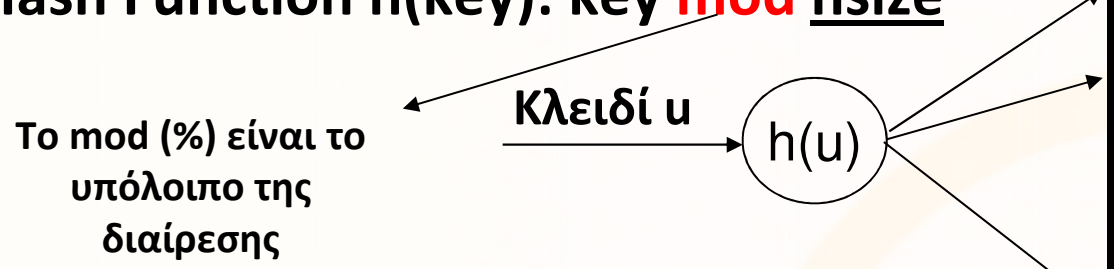
Βασική Ιδέα Κατακερματισμού

- Έστω το σύνολο ακεραίων S από το πεδίο ορισμού U

  S= {0, 1, 2, 3, 4, 5, 8, 30, 57} (U=[0..99])
- Θα φτιάξουμε ένα **Πίνακα Κατακερματισμού (Hash Table),** ο οποίος στο παράδειγμα μας έχει μέγεθος **hsize=5 (**το μέγεθος είναι συνήθως συναρτήσει του διαθέσιμου χώρου)
- Χρησιμοποιώντας κάποια **συνάρτηση κατακερματισμού (hashing function) h(key),** θα εισάγουμε τα στοιχεία του **S** στο hashtable.

Παράδειγμα (hsize: 5)

Hash Function h(key): key mod hsize



H	data
0	0, 5, 30
1	1,
2	2, 57
3	3, 8
4	4,

- Αν ψάχνουμε το key=21, τότε ξέρω ότι πρέπει να ψάξω στην θέση 1 (21 mod 5)
- Αν ψάχνω το key=4;

Κατακερματισμός – Ορισμοί & Ερωτήματα

- **Πίνακας κατακερματισμού (hash table)** είναι μια δομή δεδομένων που υποστηρίζει τις διαδικασίες **insert** και **find** σε (σχεδόν) σταθερό χρόνο O(1).

- Ένας πίνακας κατακερματισμού χαρακτηρίζεται από
  1. το μέγεθος του, **hsize,** και
  2. κάποια **συνάρτηση κατακερματισμού h** η οποία αντιστοιχεί κλειδιά στο σύνολο των ακεραίων [0,…, hsize –1] (εφόσον εφαρμοστεί το MOD)

- Τα δεδομένα αποθηκεύονται στον πίνακα H[0,…, hsize –1]:

  το κλειδί k αποθηκεύεται στον H στη θέση H[h(k)].

- Ωστόσο τα hashtable δεν είναι ιδανικό για να ανακτούμε τα στοιχεία **ταξινομημένα,** ή γενικότερα, για **αναζητήσεις σε κάποιο εύρος (range queries).**
  - π.χ. Αν ψάχνω κλειδιά μεταξύ 2-10 (range query); – θα πρέπει δυστυχώς να κοιτάξω σε όλες τις θέσεις του πίνακα.

- **Επίσης, δημιουργούνται δύο νέα σημαντικά ερωτήματα:**
  1. ποια είναι καλή επιλογή για τη συνάρτηση κατακερματισμού h;
  2. τι θα πρέπει να γίνεται αν πολλά κλειδιά είναι στο ίδιο bucket (κάδο). Τέτοιου είδους **συγκρούσεις (collisions),** είναι πολύ πιθανό να συμβούν.

     Δηλαδή για δύο κλειδιά k_1, k_2, με $k_1 \neq k_2$, το bucket να είναι ο ίδιος $h(k_1) = h(k_2)$;

1) Επιλογή Συνάρτησης Κατακερματισμού

- **Ιδιότητες μιας καλής συνάρτησης κατακερματισμού:**
  1. **Θα πρέπει να χρησιμοποιεί ολόκληρο τον πίνακα [0… hsize –1].**
  2. **Θα πρέπει να ΄σκορπίζει΄ ομοιόμορφα τα κλειδιά στον πίνακα Η.**
  3. **Θα πρέπει να υπολογίζεται εύκολα.**

- Η συνάρτηση **h** αρχικά αντιστοιχίζει το κλειδί σε κάποιο ακέραιο αριθμό a και στη συνέχεια παίρνει την τιμή «**a mod hsize**».

- Πρέπει επίσης να μπορούμε να υπολογίζουμε την συνάρτηση κατακερματισμού για **συμβολοσειρές**!

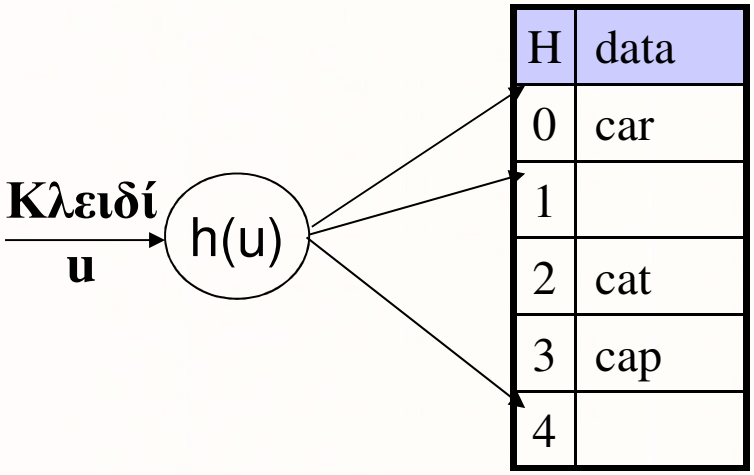
Παράδειγμα Συνάρτησης Κατακερματισμού

```
// Η συνάρτηση κατακερματισμού αθροίζει όλους τους χαρακτήρες του string
   s, και στην συνέχεια βρίσκει το υπόλοιπο από την διαίρεση με το hsize
int hash(char *s, int hsize) {
   int hash = 0;
   while ((*s) != '\0') {
        hash += (*s);
        s++;
   }
   return (hash % hsize);
}

int main() {
   char *name1 = "cat";
   char *name2 = "car";
   char *name3 = "cap";

   printf("%s hashes to bucket %d\n", name1, hash(name1, 5));
   printf("%s hashes to bucket %d\n", name2, hash(name2, 5));
   printf("%s hashes to bucket %d\n", name3, hash(name3, 5));

   return 0;
}
```



Output>

```
car hashes to bucket 2
cat hashes to bucket 0
cap hashes to bucket 3
```

1) Επιλογή Συνάρτησης Κατακερματισμού

- Έστω ότι το κλειδί είναι αλυσίδα από 3 χαρακτήρες s[0]..s[2].
- Παραδείγματα συνάρτησης κατακερματισμού είναι:

A) <u>Function 1 (Απλή συνάρτηση):</u>

Το άθροισμα των κωδικών ASCII των χαρακτήρων s[0] + s[1] + s[2]

Πρόβλημα: Λέξεις μπορούν να έχουν το ίδιο άθροισμα

π.χ. Οι λέξεις “cat” και “tac” θα έχουν το άθροισμα 99+97+116 == 116+97+99

B) <u>Function 2 (Βελτιωμένη Λύση): (Υποθέστε ότι οι χαρακ. είναι Ascii-7bit)</u>

(s[0]+127*0) + (s[1]+127*1) + (s[1]+127*2)

Τώρα κάθε επί-μέρους άθροισμα απέχει το πιο λίγο κατά 128 από το επόμενο.

Ωστόσο εάν έχουμε χαρακτήρες σε κάποια άλλη κωδικοποίηση π.χ. UNICODE-16bit τότε αυτό μας δίδει πολύ μεγάλους αριθμούς.

<u>C) Function 3 (Η συνάρτηση hashcode() στην γλώσσα JAVA)</u>

$s[n-1] + s[n-2] * 31^{1} + \ldots. + s[1]*31^{(n-2)} + s[0]*31^{(n-1)}$

Όπου **n είναι το μήκος του string**. Το hash κάποιου κενού string είναι 0.

π.χ. “cat” => n=3 => (t)**99 +** **(a)97*31^{1} + (c)116* 31^{2}** = 114,582

Η επιλογή του 31 δεν είναι τυχαία. Είναι Prime & επίσης υπολογίζεται αποδοτικά με ένα shift (a * 31 == (a << 5 - 1) (Άλλοι αριθμοί που έχουν τέτοιες ιδιότητες 17,31,127,…)

1. Συνάρτηση Κατακερματισμού (συνέχεια)

Συνάρτηση Κατακερματισμού

- Πολλές φορές οι συμβολοσειρές μπορεί να είναι πολύ μεγάλες «1304 Lincoln Ave, Alameda, 94501, CA, USA» , που θα κάνει ακριβό τον υπολογισμό της συνάρτησης κατακερματισμού.
- Για αυτό μπορεί να χρησιμοποιούνται επιλεκτικά κάποιοι χαρακτήρες (π.χ. κάθε 10ος)

Μέγεθος Πίνακα Κατακερματισμού

- Αν το μέγεθος του πίνακα κατακερματισμού είναι πολύ μεγάλο (π.χ. 10,000) και οι τιμές παράγονται όλες σε κάποιο μικρό-διάστημα τότε θα υπάρχουν πολλές συγκρούσεις.
- **Παράδειγμα:** Ένα σύνολο λέξεων όπου κάθε λέξη αποτελείται από 10 χαρακτήρες ASCII-7bit. Το συνολικό άθροισμα της κάθε λέξης είναι το πολύ 10x127=1,270.
- Άρα όλες οι πιθανές 127^10 γραμματοσειρές θα βρίσκονται στις πρώτες 1270%10000=1270 θέσεις του πίνακα και οι υπόλοιπες 10000-1270=8730 θα είναι άδειες παρόλο που χρησιμοποιήσαμε το MODulo.

Το θέμα της εύρεσης της πιο κατάλληλης συνάρτησης κατακερματισμού είναι δύσκολο.

Πάντοτε θα είναι πιθανή η ύπαρξη συγκρούσεων!

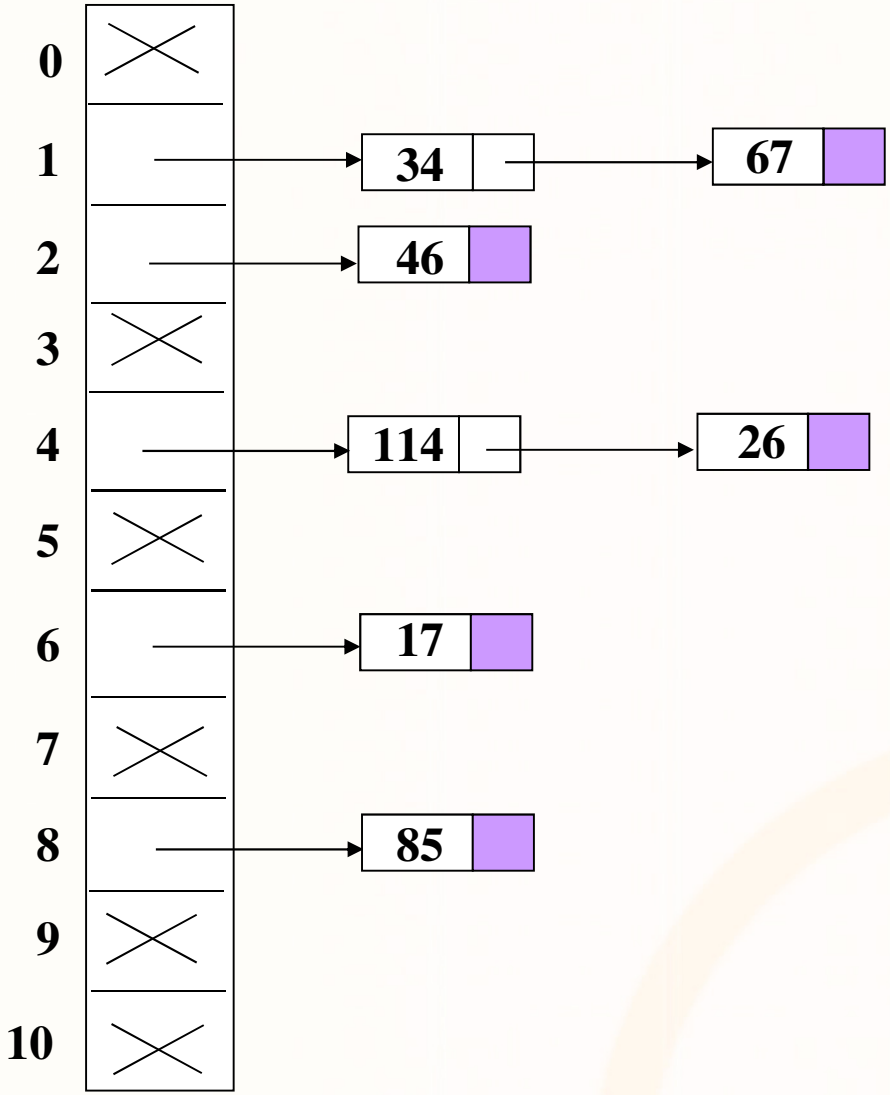
Πως γίνεται η διαχείριση συγκρουόμενων κλειδιών;

- Οι τεχνικές λύσεις διακρίνονται σε 2 κατηγορίες: μέθοδοι με **Αλυσίδωση (Chaining)** και μέθοδοι **Ανοικτής Διεύθυνσης (Open Addressing)**

Διαχείριση συγκρούσεων με Αλυσίδωση (Chaining)

- Αφού περισσότερα από ένα κλειδιά μπορούν να πάρουν την ίδια τιμή από τη συνάρτηση κατακερματισμού, μπορούμε να θεωρήσουμε ότι κάθε θέση του πίνακα **'δείχνει' σε μια ευθύγραμμη απλά συνδεδεμένη λίστα.**
- **Για κάθε i, στη θέση H[i] του πίνακα βρίσκουμε λίστα** που περιέχει όλα τα κλειδιά που απεικονίζονται από τη συνάρτηση h στη θέση αυτή.
- Για να βρούμε κάποιο κλειδί k, πρέπει να ψάξουμε στη λίστα που δείχνεται στη θέση H[h(k)].
- Εισαγωγές και εξαγωγές στοιχείων μπορούν να γίνουν εύκολα με βάση τις διαδικασίες συνδεδεμένων λιστών.

Παράδειγμα Διαχείρισης με Αλυσίδωση



hsize = 11

Ανάλυση της Τεχνικής Αλυσίδωσης

- **Εισαγωγή Στοιχείου:** απαιτεί χρόνο O(1). (προσθήκη στην αρχή)
- **Εύρεση/διαγραφή Κλειδιού k**
  - συνεπάγεται τη διέλευση της λίστας H[h(k)].
  - Για να αναλύσουμε τον χρόνο εκτέλεσης των πιο πάνω διαδικασιών ορίζουμε τον συντελεστή φορτίου **(load factor) λ** του πίνακα **H** ως τον λόγο

    λ = (αριθμός των στοιχείων που αποθηκεύει ο πίνακας) / hsize

    π.χ. 100 τιμές σε 5 buckets => λ=20.0

  - Δηλαδή, κατά μέσο όρο, κάθε λίστα του πίνακα έχει μήκος λ.
  - Το στοιχείο δεν υπάρχει (χειρίστη περίπτωση): **O(1+λ)** → **Εύρεση bucket**
  - Το στοιχείο υπάρχει (μέση περίπτωση): **O(1+λ/2)** → **Αναζήτηση Λίστας**

    Όπου O(1) κόστος για εύρεση του bucket και O(λ) & O(λ/2) αντίστοιχα για ανάλυση των στοιχείων της λίστας.

- **Ιδανικά** θα θέλαμε ο λόγος λ να έχει σταθερή τιμή (συνήθως κοντά στο 1), ώστε οι διαδικασίες να εκτελούνται σε σταθερό χρόνο.
- Στην συνέχεια θα δούμε τεχνικές **δυναμικής αύξησης / μείωσης** του πίνακα σε συνδυασμό με διαδικασίες **επανακερματισμού**, έτσι ώστε το μέγεθος του πίνακα να είναι πάντα ανάλογο του αριθμού των στοιχείων.

Διαχείριση Συγκρούσεων με ανοικτή διεύθυνση

- Η αντιμετώπιση **συγκρούσεων με αλυσίδωση** περιλαμβάνει επεξεργασία δεικτών και δυναμική χορήγηση μνήμης. Επίσης δημιουργούνται overflow chains, τα οποία θα κάνουν τις αναζητήσεις ακριβότερες στην συνέχεια

- Η στρατηγική **ανοικτής διεύθυνσης** επιτυγχάνει την αντιμετώπιση συγκρούσεων χωρίς τη χρήση δεικτών. Τα στοιχεία αποθηκεύονται κατ' ευθείαν στον πίνακα κατακερματισμού ως εξής:

- Για να **εισαγάγουμε** το κλειδί k στον πίνακα:
  1. υπολογίζουμε την τιμή i=h(k), και
  2. αν η θέση H[i] είναι κενή τότε αποθηκεύουμε εκεί το k,
  3. διαφορετικά, δοκιμάζουμε τις θέσεις f(i), f(f(i)),…, για κάποια συνάρτηση f, μέχρις ότου βρεθεί κάποια κενή θέση όπου και τοποθετούμε το k.

- Για την **αναζήτηση** κάποιου κλειδιού k μέσα στον πίνακα:
  1. υπολογίζουμε την τιμή i=h(k), και
  2. κάνουμε διερεύνηση της ακολουθίας, i, f(i), f(f(i)),…, μέχρι, είτε να βρούμε το κλειδί, είτε να βρούμε κενή θέση, ή να περάσουμε από όλες τις θέσεις του πίνακα.

Γραμμική Αναζήτηση Ανοικτής Διεύθυνσης (Linear Probing)

- Η αρχική συνάρτηση κατακερματισμού είναι

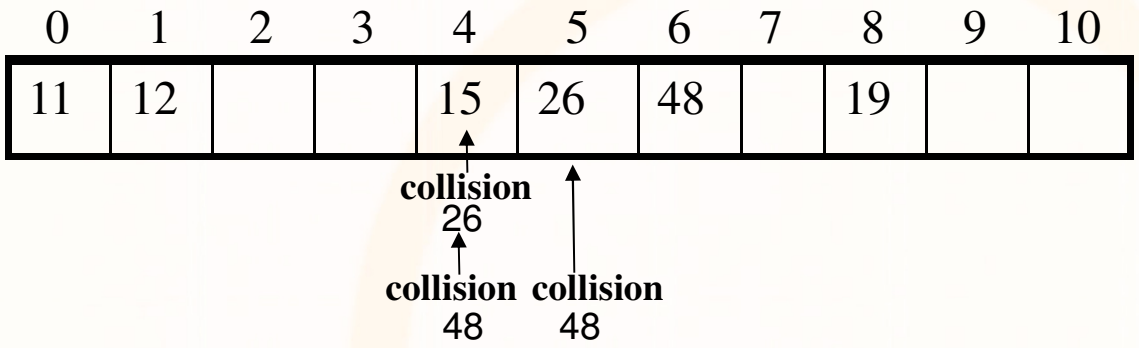
  f(x)'= x mod hsize

- Όταν υπάρξει σύγκρουση (collision) δοκιμάζουμε αναδρομικά την επόμενη συνάρτηση μέχρι να βρεθεί κενή θέση:

  f(x)= (f(x)'+i) mod hsize (i=1,2,3,…)

- Δηλαδή η αναζήτηση κενής θέσης γίνεται σειριακά, και η αναζήτηση ονομάζεται **γραμμική (linear probing).**

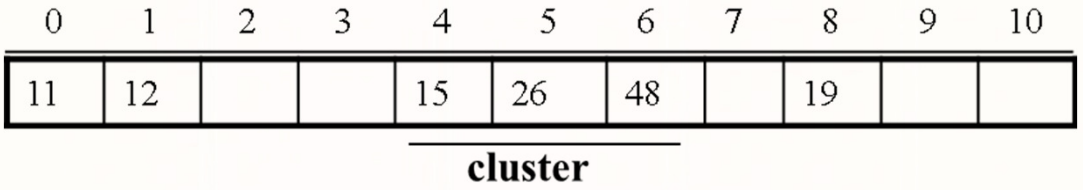
- Παράδειγμα: **hsize = 11**, εισαγωγή 11, 12, 15, 19, 26, 48.



Σχόλια για το Linear Probing

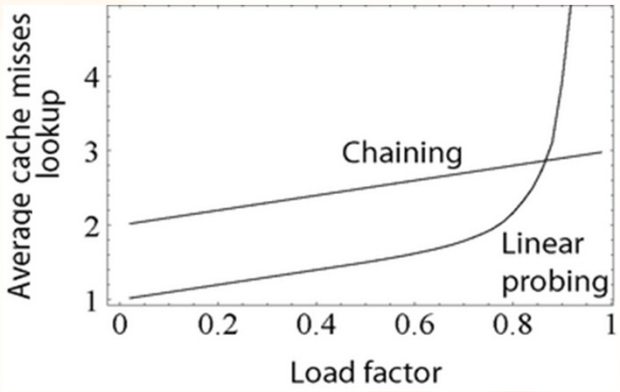
Εισαγωγή

- Εφόσον ο πίνακας κατακερματισμού δεν είναι γεμάτος, είναι πάντα δυνατό να εισάγουμε κάποιο καινούριο κλειδί. Αν γεμίσει θα κάνουμε rehash τον πίνακα (θα το δούμε στην συνέχεια)
- Αν οι γεμάτες θέσεις του πίνακα είναι **μαζεμένες (clustered)** τότε ακόμα και αν ο πίνακας είναι σχετικά άδειος, πιθανόν να χρειαστούν πολλές δοκιμές για εύρεση κενής θέσης (κατά την εκτέλεση διαδικασίας insert), ή για εύρεση στοιχείου.



Αναζήτηση

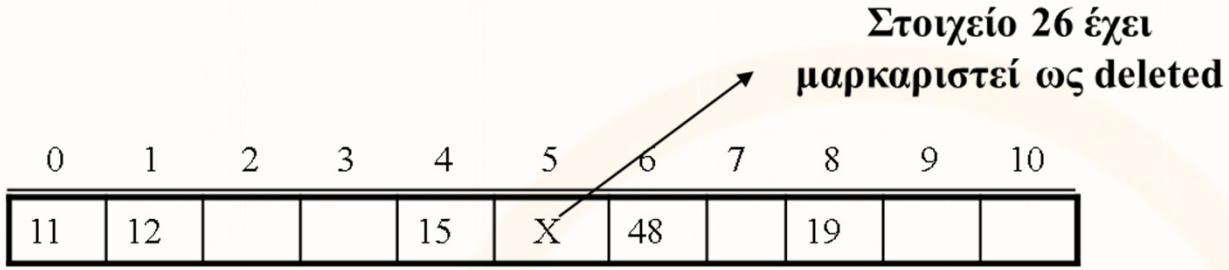
- Η αναζήτηση γίνεται όπως και την εισαγωγή (σταματάμε όταν βρούμε κενή θέση).
- Μπορεί να αποδειχθεί ότι για ένα πίνακα μισογεμάτο (δηλαδή λ = 0.5) και μια ομοιόμορφη κατανομή τότε:
  1. **Ανεπιτυχή Διερεύνηση:** Ο αριθμός βημάτων είναι ~2.5
  2. **Επιτυχή Διερεύνηση:** Ο αριθμός βημάτων είναι ~1.5.
- Αν το λ πλησιάζει το 1, τότε οι πιο πάνω αναμενόμενοι αριθμοί βημάτων αυξάνονται εκθετικά.



Σχόλια για το Linear Probing

Εξαγωγή

- Πρέπει να είμαστε προσεκτικοί με τις εξαγωγές στοιχείων
  1. μια θέση από την οποία έχει αφαιρεθεί στοιχείο δεν μπορεί να θεωρηθεί ως άδεια (γιατί;) *διότι στην find δεν θα ξέρουμε που να σταματήσουμε*
  2. έτσι μαρκάρουμε τη θέση ως deleted, και
  3. κατά τη διαδικασία find, αγνοούμε θέσεις deleted, και προχωρούμε μέχρις ότου είτε να βρούμε το κλειδί που ψάχνουμε, είτε να βρούμε (πραγματικά) μια άδεια θέση είτε να σαρώσουμε ολόκληρο τον πίνακα).



Επανακατακερματισμός (Rehashing)

- Αν ο hash πίνακας αρχίσει να γεμίζει, παρατηρείται μεγάλος **αριθμός συγκρούσεων (collisions)** με αποτέλεσμα τη μειωμένη επίδοση.
- Η μειωμένη επίδοση παρατηρείται και σε πράξεις εισαγωγής αλλά στις πράξεις αναζήτησης.

- Σε τέτοιες περιπτώσεις, όταν η τιμή λ υπερβεί κάποιο όριο, πολλές υλοποιήσεις hash-πινάκων, αυτόματα εφαρμόζουν **επανά-κατακερματισμό**.
- Αυτό το όριο σε τυπικές υλοποιήσεις είναι συνήθως λ=0.7 (π.χ. Java)

- Επανακατακερματισμός (rehashing)
  - Δημιούργησε ένα καινούριο πίνακα μεγαλύτερου (διπλάσιου) μεγέθους.
  - Εισήγαγε όλα τα στοιχεία του παλιού πίνακα στον καινούριο.
  - Επέστρεψε τη μνήμη του παλιού πίνακα.

- Ακριβή διαδικασία, αλλά καλείται σπάνια.
- Σε συστήματα **πραγματικού χρόνου (real-time systems)** το rehashing μπορεί να πάρει περισσότερο χρόνο από ότι υπάρχει!
- Εκεί το rehashing γίνεται σταδιακά (δηλαδή κρατούμε το παλιό και νέο HashTable), και σε κάθε εισαγωγή μετακινούμε Κ στοιχεία στο νέο table μέχρι να μετακινηθούν όλα τα στοιχεία (οπόταν διαγράφεται το παλιό table)

Μερικές Εφαρμογές του Κατακερματισμού

- Εφαρμογές Κατακερματισμού (Μνήμης &Μαγνητ. Δίσκου)
  - **Unique:** Έχετε ένα αρχείο από strings και θέλετε με ένα πέρασμα (χρόνος O(n)), να βρείτε όλες τις μοναδικές λέξεις σε αυτό.
  - **Ευρετήρια Λέξεων σε Μηχανές Αναζήτησης**: Ψάχνουμε σε μια μηχανή αναζήτησης την λέξη “car + rental”. Η μηχανή μας επιστρέφει την τομή των αποτελεσμάτων (συνόλων) car και rental σε χρόνο O(1).
  - **Find Function:** Σε εργαλεία επεξεργασίας κειμένου (text editors, word, κτλ) το πρόγραμμα προσφέρει την δυνατότητα εύρεσης λέξεων. Πολλές φορές η πρώτη εκτέλεση του find είναι αργή (πχ. Microsoft Help) διότι χρειάζεται χρόνος για την δημιουργία του hash table).
  - **Σε μεταγλωττιστές,** πίνακες κατακερματισμού που ονομάζονται Symbol Tables αποθηκεύουν πληροφορίες για όλες τις μεταβλητές.
  - **Διερεύνηση γράφων** που δεν είναι εξ’αρχής γνωστοί.