Διάλεξη 19: Τεχνικές Κατακερματισμού ΙΙ (Hashing)

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Διαχείριση Συγκρούσεων με Ανοικτή Διεύθυνση

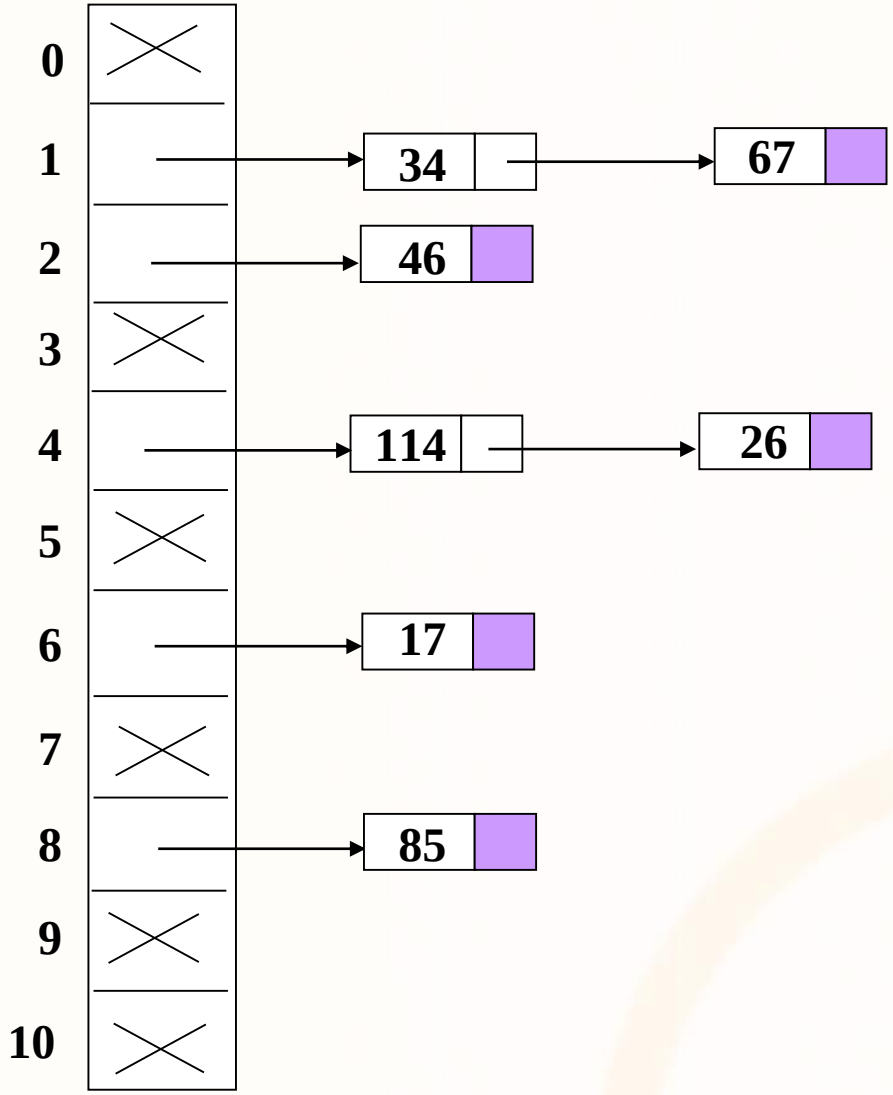
  a) Linear Probing, b) Quadratic Probing c) Double Hashing

- Διατεταγμένος Κατακερματισμός (Ordered Hashing)

- Επανακατακερματισμός (Rehashing)

- Εφαρμογές Κατακερματισμού

Παράδειγμα Διαχείρισης με Αλυσίδωση



hsize = 11

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

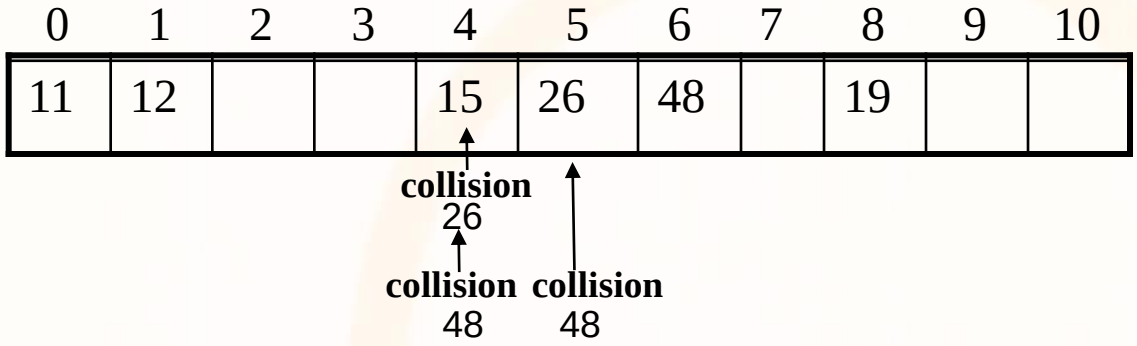
  f(x)'= x mod hsize

- 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).**

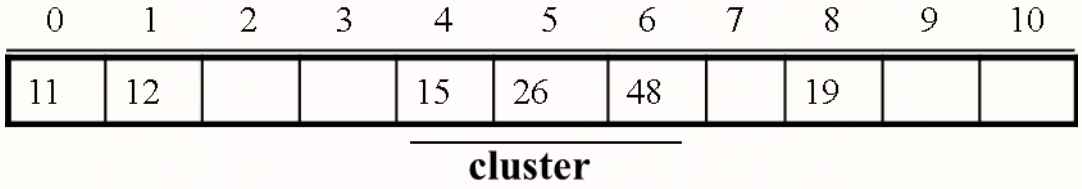
- Παράδειγμα: **hsize = 11**, εισαγωγή 11, 12, 15, 19, 26, 48.



Σχόλια για το Linear Probing

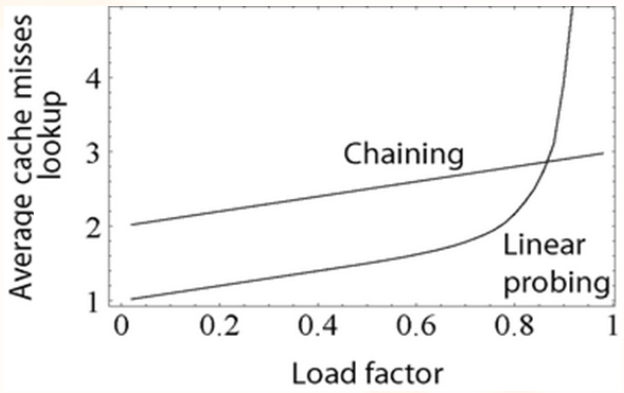
Εισαγωγή

- 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

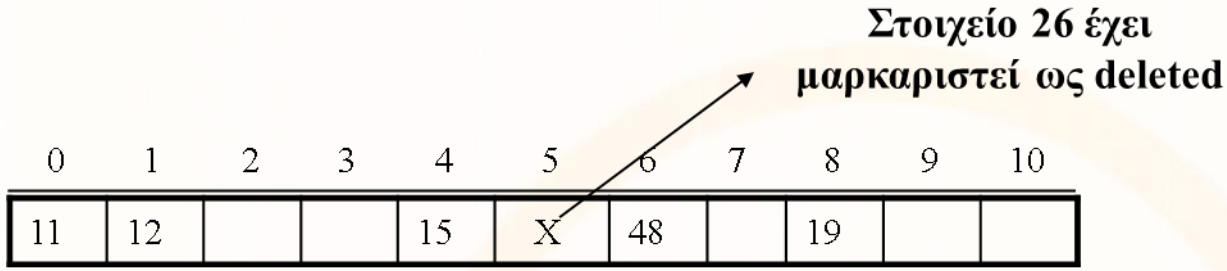
- Η αναζήτηση γίνεται όπως και την εισαγωγή (σταματάμε όταν βρούμε κενή θέση).
- Μπορεί να αποδειχθεί ότι για ένα πίνακα μισογεμάτο (δηλαδή λ = 0.5) και μια ομοιόμορφη κατανομή τότε:
  1. **Ανεπιτυχή Διερεύνηση:** Ο αριθμός βημάτων είναι ~2.5
  2. **Επιτυχή Διερεύνηση:** Ο αριθμός βημάτων είναι ~1.5.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



Δευτεροβάθμια Αναζήτηση Ανοικτής Διεύθυνσης

- **Δευτεροβάθμια Αναζήτηση Ανοικτής Διεύθυνσης (Quadratic Probing)**

- Η αρχική συνάρτηση κατακερματισμού είναι και πάλι:

  f(x)'= x mod hsize

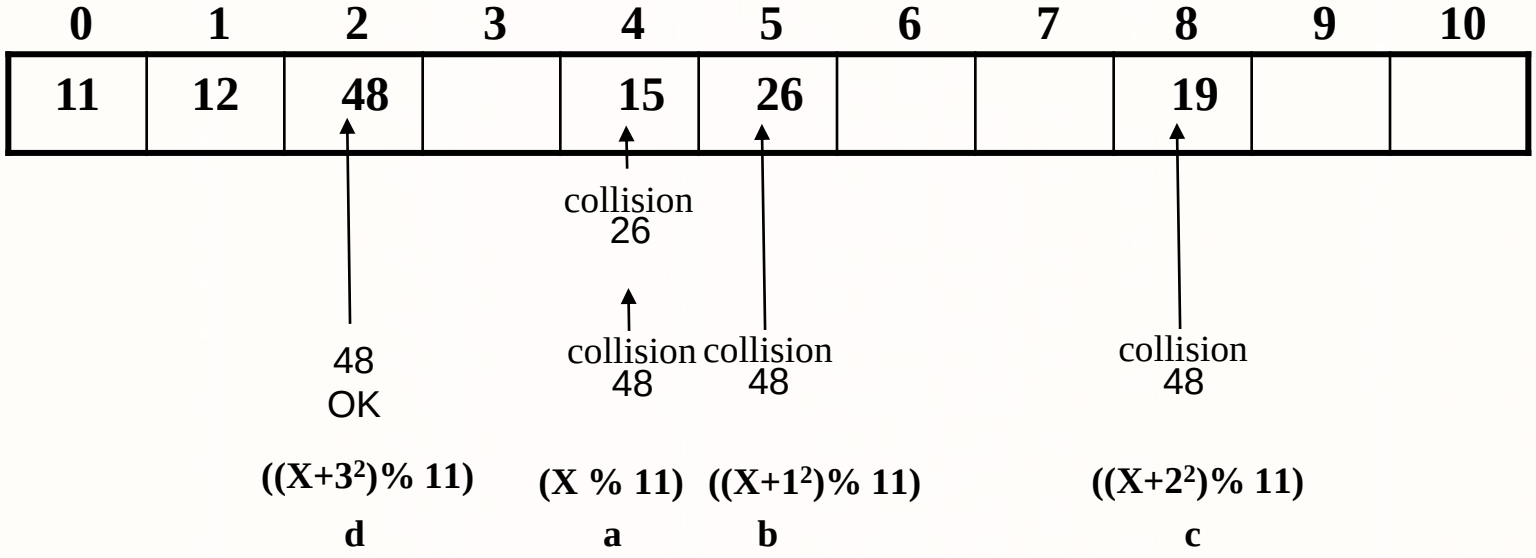
- Όταν υπάρξει σύγκρουση (**collision**) δοκιμάζουμε αναδρομικά την επόμενη συνάρτηση μέχρι να βρεθεί κενή θέση:

  $f(x)= (f(x)'+ i^2) \bmod hsize$ (i=1,2,3,…)

- **Στόχος:** αποφυγή των μαζεμένων κλειδιών (clusters)

Παράδειγμα Quadratic Probing

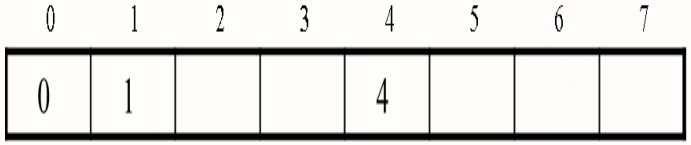
- Παράδειγμα: **hsize = 11**, εισαγωγή **11, 12, 15, 19, 26, 48**



- Στο linear probing ήταν εγγυημένη η εισαγωγή (εφόσον ο πίνακας δεν έχει γεμίσει)
- Και εδώ μπορεί να αποδειχθεί ότι :
- Θεώρημα: Αν το μέγεθος hsize είναι πρώτος (prime) αριθμός (>3) τότε οποιοδήποτε καινούριο κλειδί μπορεί να εισαχθεί στον πίνακα εφόσον ο πίνακας έχει $\lambda \leq 0.5$.

Σημαντικότητα Μεγέθους Πίνακα

- Αν το μέγεθος του πίνακα **δεν είναι prime** τότε μπορεί να δημιουργηθεί το φαινόμενο του **Funneling**
- Ας υποθέσουμε ότι θέλουμε να εισάγουμε το **{0,1,4,8}** σε ένα πίνακα μεγέθους **hsize = 8**.



- Η εισαγωγή του 0,1,4 γίνεται κανονικά.
- Το 8 ωστόσο δεν μπορεί να εισαχθεί χρησιμοποιώντας το quadratic probing. Συγκεκριμένα έχουμε αλλεπάλληλες συγκρούσεις (collisions) :

1) 8%8=0 (X) **2)** (8+1)%8=1 (X) **3)** (8+4)%8=4 (X) **4)** (8+9)%8=1 (X)
4) (8+16)%8=0 (X) **5)** (8+25)%8=1 (X) **6)** (8+36)%8=4 (X) **7)** (8+49)%8=1 (X)
……..

- Τώρα ας υποθέσουμε ότι θέλουμε να εισάγουμε το **{0,1,4,8}** σε ένα πίνακα μεγέθους **hsize = 7 (PRIME)**. Πάλι η εισαγωγή του 0,1,4 γίνεται κανονικά
- Επίσης το 8 μετά από ένα collision τοποθετείται στον πίνακα

1) 8%7=1 (X) **2)** (8+1)%7=2 (OK!)

Διπλός Κατακερματισμός Ανοικτής Διεύθυνσης

- **Διπλός Κατακερματισμός Ανοικτής Διεύθυνσης (Double Hashing)**

- Ο τελευταίος τρόπος αποφυγής συγκρούσεων χρησιμοποιεί δυο συναρτήσεις κατακερματισμού.
- Δηλαδή σε περίπτωση αρχικής αποτυχίας εισαγωγής / εύρεσης στοιχείου οι θέσεις που επιλέγουμε για να διερευνήσουμε στη συνέχεια (probe sequence) είναι **ανεξάρτητες** από την πρώτη.

  $f(x,0) = h_1(x)$ ***// η αρχική συνάρτηση κατακερματισμού***
- Αυτό επιτυγχάνεται με τη χρήση μιας δεύτερης συνάρτησης κατακερματισμού, h_2, ως εξής:

  $f(x,n) = (h_1(x) + n \cdot h_2(x)) \bmod hsize$

- Στην πράξη δουλεύει αποδοτικά ωστόσο είναι πιο ακριβό να υπολογίζουμε δυο συναρτήσεις κάθε φορά

Άλλες Τεχνικές – Ordered Hashing

- **Διατεταγμένος Κατακερματισμός (Ordered Hashing)**

- Η μέθοδος αυτή χρησιμοποιείται σε συνδυασμό με οποιαδήποτε από τις άλλες τεχνικές με στόχο την ελάττωση του χρόνου εκτέλεσης της διερεύνησης.
- Η βασική ιδέα είναι να εξασφαλίζεται ότι τα κλειδιά που συναντούμε κατά τη διερεύνηση μιας probing sequence είναι σε αύξουσα σειρά.
- Έτσι, αν συναντήσουμε κλειδί που είναι μεγαλύτερο από αυτό που ψάχνουμε, τότε συμπεραίνουμε πως δεν υπάρχει στον πίνακα.

Μέθοδος υλοποίησης

- Μέθοδος υλοποίησης: κατά την εισαγωγή κλειδιού k σε ένα πίνακα, αν βρούμε κλειδί k΄> k, τότε εισάγουμε το k στη θέση του k΄ και αναλαμβάνουμε να εισάγουμε το k΄ σε κάποια μετέπειτα θέση.
- Σαν αποτέλεσμα έχουμε βελτιωμένη διαδικασία **ανεπιτυχούς αναζήτησης**.

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.

Επανακατακερματισμός (Rehashing)

- 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)

- Σε **βάσεις δεδομένων (databases)**, ο όγκος των δεδομένων είναι πολύ μεγάλος και τα δεδομένα είναι αποθηκευμένα στον δίσκο.
- Άρα το re-hashing, θα έπαιρνε παρά πολύ χρόνο μέχρι να ολοκληρωθεί.
- Για αυτό χρησιμοποιούνται dynamic hashing techniques (π.χ. Linear and extendible hashing)
- Σε αυτές τις τεχνικές, μόνο ένα πολύ μικρό ποσοστό δεδομένων χρειάζεται να γίνει rehashed.

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.

Bloom Filter

- Πολύ διαδεδομένη δομή δεδομένων που βασίζεται στον κατακερματισμό
- Μπορεί αποδοτικά να **ελέγχει αν ένα στοιχείο ΔΕΝ αποτελεί μέλος κάποιου συνόλου** (όχι το αντίθετο)
  - Πάντα επιστρέφει false αν το στοιχείο δεν ανήκει στο σύνολο
    ➔ **NO false negatives**
  - Μπορεί να επιστρέψει true για ένα στοιχείο που δεν ανήκει στο σύνολο
    ➔ **false positives**
- Υλοποιείται με:
  - ένα **BitArray**

0	1	0	1	1	1	0	0	0	0	0	1	0	1	0	0	1	0

  - **k hash functions** (ένα ή πολλά)
- Καινούρια στοιχεία μπορούν να προστεθούν
- Η Εξαγωγή στοιχείων δεν επιτρέπεται

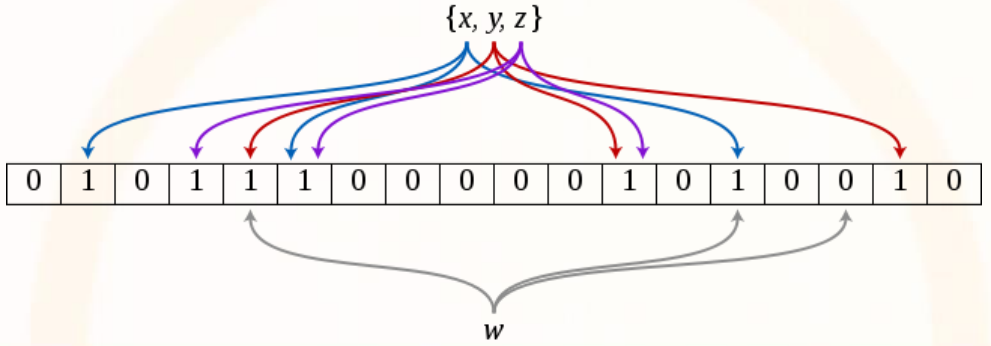
Bloom Filter: Υλοποίηση

- Έστω ότι το BitArray έχει μέγεθος **m**.
- Αρχικά, όλες οι θέσεις (bits) έχουν την τιμή 0

- Η **εισαγωγή στοιχείου *σ*** γίνεται ως εξής:
  - Υπολόγισε το hash value του σ για κάθε ένα από τα k hash functions
  - Αυτό δημιουργεί k'≤k θέσεις στον πίνακα (k αν τα functions είναι τέλεια!)
  - Θέσε τις θέσεις k' ίσες με 1

- Η **επερώτηση για κάποιο στοιχείου q** γίνεται ως εξής:
  - Υπολόγισε το hash value του q για κάθε ένα από τα k hash functions
  - Έλεγξε αν τουλάχιστον μία θέση =0➔επέστρεψε true ή false αντίστροφα

- Παράδειγμα m=18, k=3



Bloom Filter: Παρατηρήσεις

- Το Bloom Filter παρουσιάζει πολλά πλεονεκτ. και κάποια μειονεκτ.
  - + Πολύ αποδοτική: O(k), (k≤5 ➔ ~O(1)), άσχετο με τον αριθμό των αντικείμενων που υπάρχουν στο σύνολο
  - + Πολύ μικρή χρήση χώρου μνήμης σε σχέση με πίνακες κατακερματισμού, λίστες και δέντρα
    - Για 1,048,576 (1MB) στοιχεία: BloomFilter-132KB, HasTable≥1MB, Λίστα ≥8MB, Δέντρα≥12MB
  - + Αν το ποσοστό των false positives δεν είναι ικανοποιητικό τότε μπορούμε απλά να προσθέσουμε extra bits,
    π.χ., n bits=1% false positives, n+5 bits = 0.1% false positives!
  - - Τα false positives αυξάνονται ραγδαία με την είσοδο καινούριων στοιχείων
  - - Υπάρχει πιθανότητα μεγάλος χώρος του πίνακα να είναι συνεχώς αχρησιμοποίητος