

Κατακερματισμός (Hashing)

Ο Αφηρημένος Τύπος Δεδομένων: Πίνακας Συμβόλων (Υπενθύμιση)

- Ένας πίνακας συμβόλων T είναι μια αφηρημένη αποθήκη που περιέχει καταχωρήσεις πίνακα οι οποίες είναι είτε κενές είτε ζεύγη της μορφής (K, I) , όπου:
 - K είναι ένα κλειδί (key)
 - I είναι κάποια πληροφορία σχετιζόμενη με το κλειδί, την οποία ονομάζουμε τιμή (value)
 - Υποθέτουμε ότι κάθε διαφορετική καταχώρηση έχει διαφορετικό κλειδί.
 - Άρα, οι πίνακες συμβόλων που μελετώνται εδώ είναι ουσιαστικά **maps** (αντιστοιχίσεις).
-

Λειτουργίες για τον Πίνακα Συμβόλων (Symbol Table ADT)

- Αρχικοποίηση** του πίνακα T ως τον κενό πίνακα.
 - Ο κενός πίνακας περιέχει μόνο κενές καταχωρήσεις (K_0, I_0) , όπου K_0 είναι ειδικό κενό κλειδί, διακριτό από όλα τα άλλα μη κενά κλειδιά.
- Έλεγχος** αν ο πίνακας T είναι πλήρης.
- Εισαγωγή** νέας καταχώρησης (K, I) στον πίνακα, εφόσον δεν είναι ήδη πλήρης.
- Διαγραφή** της καταχώρησης (K, I) από τον πίνακα T .
- Ανάκτηση** της πληροφορίας I με βάση το κλειδί αναζήτησης K , από την καταχώρηση (K, I) στον πίνακα T .
- Ενημέρωση** της καταχώρησης (K, I) στον πίνακα T , αντικαθιστώντας την με μια νέα καταχώρηση (K, I') .
- Απαρίθμηση** των καταχωρήσεων (K, I) του πίνακα T με αύξουσα σειρά κλειδιών.

Πιθανές Αναπαραστάσεις για τον Πίνακα Συμβόλων (Symbol Table ADT)

- Έχουμε ήδη συζητήσει τις παρακάτω δομές δεδομένων για πίνακες συμβόλων:
 - Πίνακες από δομές (structs) ταξινομημένους σε αύξουσα σειρά με βάση τα κλειδιά
 - Συνδεδεμένες λίστες από δομές
 - Δυαδικά δέντρα αναζήτησης (Binary Search Trees)
 - Δέντρα AVL
 - Δέντρα (2,4)
 - Δέντρα red-black
 - Λίστες skip (Skip Lists)
-

Κατακερματισμός (Hashing)

- Θα εισαγάγουμε μια νέα μέθοδο για την υλοποίηση του πίνακα συμβόλων, που ονομάζεται **κατακερματισμός (hashing)**.
 - Ο κατακερματισμός διαφέρει από τις προηγούμενες αναπαραστάσεις που βασίζονται σε συγκρίσεις κλειδιών, επειδή προσπαθούμε να αναφερθούμε **άμεσα** σε στοιχεία του πίνακα, μετατρέποντας τα κλειδιά σε **διευθύνσεις** στον πίνακα.
-

Εισαγωγή στον Κατακερματισμό με Παράδειγμα

- Θα χρησιμοποιήσουμε ως κλειδιά γράμματα του αλφαβήτου με δείκτες που δείχνουν τη σειρά τους στο αλφάβητο.
 - Π.χ., A_1 , C_3 , R_{18}
- Θα χρησιμοποιήσουμε έναν μικρό πίνακα T με 7 θέσεις ως αποθηκευτικό χώρο.
 - Ο πίνακας αυτός ονομάζεται **πίνακας κατακερματισμού (hash table)**.
- Θα βρούμε τη θέση για να αποθηκεύσουμε ένα κλειδί L_n χρησιμοποιώντας την εξής **συνάρτηση κατακερματισμού**:

$$h(L_n) = n \% 7$$

Η έκφραση $x \% y$ υπολογίζει το υπόλοιπο της ακέραιας διαίρεσης του x με το y .

An Empty Hash Table

	Table T
0	
1	
2	
3	
4	
5	
6	

Αποθήκευση Κλειδιών στον Πίνακα Κατακερματισμού

- Τα **κλειδιά** αποθηκεύονται στις **διευθύνσεις κατακερματισμού** τους.
- Τα κελιά του πίνακα συχνά ονομάζονται **buckets (κάδοι)**.

Table T after Inserting keys $B_2, J_{10}, S_{19}, N_{14}$

	Table T
0	N_{14}
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

Insert X_{24}

	Table T
0	N_{14}
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

$$h(X_{24}) = 3$$

Σύγκρουση (Collision) στον Πίνακα Κατακερματισμού

- Τώρα έχουμε μια **σύγκρουση (collision)**.
- Θα χρησιμοποιήσουμε μια **πολιτική επίλυσης συγκρούσεων (collision resolution policy)**:
 - Θα αναζητήσουμε **χαμηλότερες θέσεις** στον πίνακα για να βρούμε μια διαθέσιμη θέση για το κλειδί.

Insert X_{24}

	Table T	
0	N_{14}	
1	X_{24}	← 3 rd probe
2	B_2	← 2 nd probe
3	J_{10}	← $h(X_{24}) = 3$ 1 st probe
4		
5	S_{19}	
6		

Insert W_{23}

	Table T	
0	N_{14}	← 3 rd probe
1	X_{24}	← 2 nd probe
2	B_2	← $h(w_{23}) = 2$ 1 st probe
3	J_{10}	
4		
5	S_{19}	
6	W_{23}	← 4 th probe

Ανοικτή Διευθυνσιοδότηση (Open Addressing)

- Η μέθοδος εισαγωγής κλειδιών που συγκρούονται σε κενές θέσεις του πίνακα ονομάζεται **ανοικτή διευθυνσιοδότηση (open addressing)**.
- Η εξέταση κάθε θέσης ονομάζεται **διερεύνηση (probe)**.
- Οι θέσεις που εξετάζουμε αποτελούν τη **σειρά διερεύνησης (probe sequence)**.
- Η διαδικασία διερεύνησης που ακολουθήσαμε ονομάζεται **γραμμική διερεύνηση (linear probing)**.
- Άρα, η τεχνική κατακερματισμού που χρησιμοποιούμε εδώ είναι: **ανοικτή διευθυνσιοδότηση με γραμμική διερεύνηση (open addressing with linear probing)**.

Διπλός Κατακερματισμός (Double Hashing)

- Ο **διπλός κατακερματισμός (double hashing)** είναι μια άλλη τεχνική ανοικτής διευθυνσιοδότησης.
- Χρησιμοποιεί **μη γραμμική διερεύνηση** υπολογίζοντας διαφορετικές τιμές μείωσης (decrements) στη διερεύνηση για διαφορετικά κλειδιά, με χρήση μιας **δεύτερης συνάρτησης κατακερματισμού** $p(L_n)$.

Ορισμός της Συνάρτησης Μείωσης Διερεύνησης:

$$p(L_n) = \max(1, \lfloor n / 7 \rfloor)$$

Στην παραπάνω εξίσωση:

Ο όρος $n / 7$ είναι το ηηλίκo της ακέραιας διαίρεσης του n με το 7.

Ο τελεστής $\max(a, b)$ επιστρέφει το μέγιστο μεταξύ των a και b .

Table T after Inserting keys
 $B_2, J_{10}, S_{19}, N_{14}$

	Table T
0	N_{14}
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

Insert X_{24}

	Table T	
0	N_{14}	← 2 nd probe
1		
2	B_2	
3	J_{10}	← $h(X_{24}) = 3$
4	X_{24}	← 1 st probe ← 3 rd probe
5	S_{19}	
6		

Όταν προκύψει σύγκρουση, και ξεκινάμε probing (αναζήτηση άλλης θέσης), μετακινούμαστε προς τα πίσω κατά 3 θέσεις κάθε φορά (ή modulo το μέγεθος του πίνακα).

Insert W_{23}

	Table T	
0	N_{14}	
1		
2	B_2	← $h(W_{23}) = 2$ ← 1 st probe
3	J_{10}	
4	X_{24}	
5	S_{19}	
6	W_{23}	← 2 nd probe

Παράδειγμα Διερεύνησης με Διπλό Κατακερματισμό

- Χρησιμοποιούμε μια **μείωση διερεύνησης** (probe decrement) της μορφής:

$$p(W_{23}) = 3$$

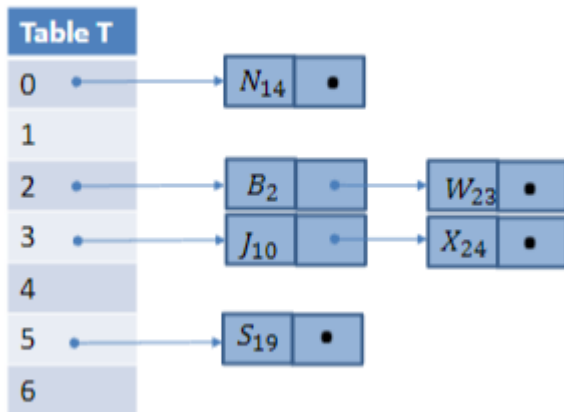
Γενικά, οι τιμές μείωσης διερεύνησης (probe decrements) θα είναι διαφορετικές για διαφορετικά κλειδιά.

Δοκίμασε να εισάγεις μόνος/η σου το κλειδί P_{16} .

Επίλυση Συγκρούσεων με Χωριστή Αλυσίδωση (Separate Chaining)

- Η μέθοδος επίλυσης συγκρούσεων με χωριστή αλυσίδωση (separate chaining) χρησιμοποιεί μια συνδεδεμένη λίστα για να αποθηκεύσει τα κλειδιά σε κάθε θέση του πίνακα.
- Δηλαδή, κάθε θέση του πίνακα κατακερματισμού δείχνει σε μια λίστα με όλα τα κλειδιά που κατακερματίστηκαν σε αυτήν τη θέση.
- Αυτή η μέθοδος δεν είναι κατάλληλη όταν η μνήμη είναι περιορισμένη, π.χ. όταν υλοποιούμε έναν πίνακα κατακερματισμού για φορητές συσκευές.

Example



Καλές Συναρτήσεις Κατακερματισμού (Good Hash Functions)

- Έστω ότι T είναι ένας πίνακας κατακερματισμού με M θέσεις, με διευθύνσεις από 0 έως $M - 1$.
- Μια **ιδανική συνάρτηση κατακερματισμού** $h(K)$ χαρτογραφεί τα κλειδιά σε διευθύνσεις πίνακα **ομοιόμορφα και τυχαία**.
- Δηλαδή, για οποιοδήποτε κλειδί, κάθε πιθανή διεύθυνση του πίνακα είναι **εξίσου πιθανή** (πιθανότητα επιλογής $= 1 / M$).
- Επιπλέον, ο **υπολογισμός** της συνάρτησης κατακερματισμού θα πρέπει να είναι **πολύ γρήγορος**.

Συγκρούσεις (Collisions)

- Μια **σύγκρουση** ανάμεσα σε δύο κλειδιά K και K' συμβαίνει όταν και τα δύο δίνουν την **ίδια** διεύθυνση κατακερματισμού:

$$h(K) = h(K')$$

- Συγκρούσεις είναι σχετικά συχνές, ακόμα και σε αραιά γεμάτους πίνακες κατακερματισμού.
- Το παράδοξο von Mises: Αν υπάρχουν πάνω από 23 άτομα σε ένα δωμάτιο, υπάρχει πάνω από 50% πιθανότητα ότι δύο από αυτούς έχουν την ίδια ημέρα γενεθλίων ($M = 365$).

Δες την απόδειξη στο βιβλίο του Standish.

- Μια καλή συνάρτηση κατακερματισμού πρέπει να ελαχιστοποιεί τις συγκρούσεις.

Πρωταρχική Συσταδοποίηση (Primary Clustering)

- Η γραμμική διερεύνηση (linear probing) υποφέρει από το φαινόμενο της πρωταρχικής συσταδοποίησης (primary clustering).
- Μια συστάδα (cluster) είναι μια ακολουθία συνεχόμενων κατειλημμένων θέσεων στον πίνακα κατακερματισμού.
- Στην ανοικτή διευθυνσιοδότηση με γραμμική διερεύνηση, τέτοιες συστάδες σχηματίζονται και στη συνέχεια μεγαλώνουν.
- Αυτό συμβαίνει γιατί όλα τα κλειδιά που συγκρούονται στην ίδια αρχική θέση ακολουθούν παρόμοιες διαδρομές διερεύνησης για να βρουν κενή θέση.
- Ο διπλός κατακερματισμός (double hashing) δεν υποφέρει από πρωταρχική συσταδοποίηση, γιατί τα συγκρουόμενα κλειδιά ακολουθούν διαφορετικές διαδρομές διερεύνησης.

Εξασφάλιση ότι οι Σειρές Διερεύνησης Καλύπτουν τον Πίνακα

- Για να λειτουργούν σωστά οι **αλγόριθμοι εισαγωγής και αναζήτησης** σε πίνακα κατακερματισμού με **ανοικτή διευθυνσιοδότηση**, πρέπει να εξασφαλίζεται ότι **κάθε σειρά διερεύνησης** μπορεί να καλύψει **όλες τις θέσεις** του πίνακα.
- Αυτό είναι **προφανές** για τη **γραμμική διερεύνηση** (linear probing), αφού πηγαίνουμε απλά από τη μια θέση στην επόμενη κυκλικά.
- Ισχύει το ίδιο και για τον **διπλό κατακερματισμό (double hashing)**:
 - Για να είναι εγγυημένο ότι κάθε θέση μπορεί να ερευνηθεί, η δεύτερη συνάρτηση κατακερματισμού (δηλ. η μείωση $p(K)$) πρέπει να είναι **συνεπόμενη** (coprime) με το μέγεθος του πίνακα M .

🔴 Επιλογή Μεγέθους Πίνακα και Βημάτων Προσπέλασης (Probing)

Θεώρημα:

Αν επιλέξουμε:

- Το μέγεθος του πίνακα (M) να είναι πρώτος αριθμός, και
- Τα βήματα προσπέλασης (probe decrements) p_K να είναι θετικοί ακέραιοι στο διάστημα:

$$1 \leq p_K \leq M$$

Τότε η σειρά probing (οι θέσεις που εξετάζονται σε περίπτωση σύγκρουσης) καλύπτει όλες τις θέσεις του πίνακα, από 0 έως $M - 1$, ακριβώς μία φορά.

🔵 Απόδειξη

➤ Περιγραφή Σειράς Probing:

Για κάποιο κλειδί K που προκαλεί σύγκρουση, η σειρά probing είναι:

$$(h_K - i \cdot p_K) \bmod M$$

όπου $i \in [0, M - 1]$.

➡ Αυτή η σειρά έχει M θέσεις.

► Απόδειξη με Αντίφαση:

Ας υποθέσουμε ότι δύο διαφορετικές τιμές $j \neq k$ από το διάστημα $[0, M - 1]$ παράγουν την ίδια θέση:

$$(h_K - j \cdot p_K) \bmod M = (h_K - k \cdot p_K) \bmod M$$

Αυτό ισοδυναμεί (στη θεωρία αριθμών) με:

$$(h_K - j \cdot p_K) \equiv (h_K - k \cdot p_K) \bmod M$$

Αφαιρούμε το h_K και από τις δύο πλευρές και πολλαπλασιάζουμε με -1:

$$j \cdot p_K \equiv k \cdot p_K \bmod M$$

► Διαίρεση και Συμπέρασμα:

Εφόσον:

- p_K και M είναι σχετικά πρώτοι, μπορούμε να διαιρέσουμε και τις δύο πλευρές με το p_K :

$$j \equiv k \bmod M \Rightarrow j \% M = k \% M$$

Αλλά:

- Οι τιμές j και k βρίσκονται στο διάστημα $[0, M - 1]$,
- Άρα: $j = k$, κάτι που αντιφάσκει με την υπόθεση ότι $j \neq k$.

✓ Συμπέρασμα:

Η υπόθεση ότι δύο διαφορετικοί δείκτες οδηγούν στην ίδια θέση **οδηγεί σε αντίφαση**. Άρα:

Η σειρά probing καλύπτει όλες τις θέσεις του πίνακα ακριβώς μία φορά, χωρίς επαναλήψεις.

■ Ορολογία

► Ισοδυναμία Modulo (Congruence):

Δύο ακέραιοι αριθμοί a και b λέγονται **ισοδύναμοι modulo n** αν:

$$a \equiv b \pmod{n} \Leftrightarrow n \mid (a - b)$$

Δηλαδή, αν η διαφορά τους είναι πολλαπλάσιο του n .

Παραδείγματα:

- $38 \equiv 14 \pmod{12} \rightarrow$ γιατί: $38 - 14 = 24 = 2 \cdot 12$
- $8 \equiv -7 \pmod{5} \rightarrow$ γιατί: $8 - (-7) = 15 = 3 \cdot 5$

► Σχετικώς Πρώτοι:

Δύο ακέραιοι αριθμοί a και b είναι **σχετικώς πρώτοι** αν ο μέγιστος κοινός διαιρέτης τους (GCD) είναι το 1.

Παραδείγματα:

- ✓ 7 και 5 \rightarrow σχετικώς πρώτοι
- ✓ 8 και 3 \rightarrow σχετικώς πρώτοι
- ✗ 8 και 4 \rightarrow **όχι** σχετικώς πρώτοι (GCD = 4)

🔗 Σύνδεση με την Απόδειξη:

Επειδή το M είναι **πρώτος αριθμός**, έχει μόνο δύο διαιρέτες: το 1 και το M ίδιο. Άρα:

Κάθε ακέραιος p_K τέτοιος ώστε: $1 \leq p_K \leq M - 1$ είναι **σχετικώς πρώτος με το M**

➡ Και έτσι η διαίρεση που έγινε στην απόδειξη είναι έγκυρη.

Καλές Επιλογές για Διπλό Κατακερματισμό (Good Double Hashing Choices)

- **Επιλογή 1:** Ορίστε το μέγεθος του πίνακα M να είναι **πρώτος αριθμός**, και επιλέξτε **τιμές μείωσης διερεύνησης** (probe decrements) ως **οποιοδήποτε ακέραιο από 1 έως $M - 1$** .

- **Επιλογή 2:** Ορίστε το M ως **δύναμη του 2**, και επιλέξτε τις μειώσεις διερεύνησης ως **μονούς αριθμούς** από 1 έως $M - 1$.
- Με άλλα λόγια: είναι καλό οι τιμές των μειώσεων διερεύνησης να είναι **σχετικά πρώτοι (relatively prime)** με το M .

Υλοποίηση Ανοικτής Διευθυνσιοδότησης με Διπλό Κατακερματισμό σε C

```
#define M 997 // 997 είναι πρώτος αριθμός
#define EmptyKey 0 // Ειδική τιμή για κενή θέση

typedef int KeyType;

typedef struct {
    // Μέλη διαφόρων τύπων με πληροφορίες που σχετίζονται με τα κλειδιά
} InfoType;

typedef struct {
    KeyType Key;
    InfoType Info;
} TableEntry;

typedef TableEntry Table[M];

// Παγκόσμια μεταβλητή πίνακας κατακερματισμού
Table T;

// Αρχικοποίηση πίνακα
void Initialize(void) {
    int i;
    for (i = 0; i < M; i++)
        T[i].Key = EmptyKey;
}

// Συνάρτηση εισαγωγής
void HashInsert(KeyType K, InfoType I) {
    int i;
    int ProbeDecrement;

    i = h(K);
    ProbeDecrement = p(K);

    while (T[i].Key != EmptyKey) {
        i -= ProbeDecrement;
        if (i < 0)
            i += M;
    }

    T[i].Key = K;
    T[i].Info = I;
}
```

```
// Συνάρτηση αναζήτησης
int HashSearch(KeyType K) {
    int i;
    int ProbeDecrement;
    KeyType ProbeKey;

    i = h(K);
    ProbeDecrement = p(K);
    ProbeKey = T[i].Key;

    while ((K != ProbeKey) && (ProbeKey != EmptyKey)) {
        i -= ProbeDecrement;
        if (i < 0)
            i += M;
        ProbeKey = T[i].Key;
    }

    if (ProbeKey == EmptyKey)
        return -1;    // Αποτυχία
    else
        return i;     // Επιτυχία, επιστρέφει το index
}
```

Διαγραφή από Πίνακα Κατακερματισμού (Deletion)

- Η συνάρτηση για **διαγραφή** από έναν πίνακα κατακερματισμού **αφήνεται ως άσκηση**.
- Όμως, η διαγραφή **δημιουργεί προβλήματα**:
 - Αν διαγράψουμε μια καταχώρηση και **αφήσουμε κενή θέση (EmptyKey)**, τότε **καταστρέφεται η εγκυρότητα** της αναζήτησης, επειδή η αναζήτηση σταματά μόλις βρει κενό κλειδί.
- **Λύση**:
 - Αντί να αφήσουμε κενό, μπορούμε να **σημειώσουμε την καταχώρηση ως διαγραμμένη** (π.χ. με ειδική τιμή "διαθέσιμο" ή "deleted").
 - Έτσι, οι **συναρτήσεις αναζήτησης** θα συνεχίσουν να διερευνούν τις διαγραμμένες θέσεις.
 - Οι **συναρτήσεις εισαγωγής** μπορούν να τις επαναχρησιμοποιούν.
- **Πρόβλημα**:
 - Αν γίνουν πολλές διαγραφές, ο πίνακας γεμίζει με "deleted" και **μειώνεται η απόδοση**.

Κώδικας για Χωριστή Αλυσίδωση (Separate Chaining)

```
typedef struct STnode* link;

struct STnode {
    Item item;
    link next;
};

static link *heads, z;
```

```

static int N, M;

// Δημιουργία νέου κόμβου
static link NEW(Item item, link next) {
    link x = malloc(sizeof *x);
    x->item = item;
    x->next = next;
    return x;
}

// Αρχικοποίηση πίνακα κατακερματισμού
void STinit(int max) {
    int i;
    N = 0;
    M = max / 5; // Μείωση μεγέθους για λόγους κατακερματισμού
    heads = malloc(M * sizeof(link));
    z = NEW(NULLitem, NULL);
    for (i = 0; i < M; i++)
        heads[i] = z;
}

```

Σημειώσεις για Χωριστή Αλυσίδωση (Separate Chaining)

- **N**: Πλήθος των στοιχείων (κλειδιών) στον πίνακα κατακερματισμού.
- **M**: Μέγεθος του πίνακα (αριθμός κάδων).
- Διατηρούμε **M** λίστες με δείκτες κεφαλών αποθηκευμένους στον πίνακα **heads**.
- Χρησιμοποιούμε **συνάρτηση κατακερματισμού** για να επιλέξουμε σε ποια λίστα θα μπει το κάθε στοιχείο.
- Η συνάρτηση **STinit** ορίζει το **M** έτσι ώστε κάθε λίστα να έχει κατά μέσο όρο ~5 στοιχεία, άρα οι πράξεις απαιτούν λίγες διερευνήσεις.

Αναδρομική Αναζήτηση (Recursive Search)

```

Item searchR(link t, Key v) {
    if (t == z) return NULLitem;           // Τέλος λίστας,
    δεν βρέθηκε                             // Αντιστοιχία
    if (eq(key(t->item), v)) return t->item;
    κλειδιού
    return searchR(t->next, v);             // Συνέχισε στην
    επόμενη θέση
}

Item STsearch(Key v) {
    return searchR(heads[hash(v, M)], v);  // Ξεκίνημα από
    τη σωστή λίστα
}

void STinsert(Item item) {
    int i = hash(key(item), M);             // Καθορισμός

```

```
κάδου
    heads[i] = NEW(item, heads[i]);           // Νέα καταχώρηση
στο κεφάλι της λίστας
    N++;                                     // Ενημέρωση
πλήθους στοιχείων
}
void STdelete(Item item) {
    int i = hash(key(item), M);              // Εύρεση σωστής
λίστας
    heads[i] = deleteR(heads[i], item);      // Διαγραφή από
τη λίστα
}
```

⚠ Η deleteR δεν δίνεται, αλλά υποθέτουμε ότι είναι αναδρομική συνάρτηση που διαγράφει ένα στοιχείο από μια λίστα.

Συνάρτηση Κατακερματισμού (Hash Function)

- Ο προηγούμενος κώδικας **υποθέτει** ότι έχουμε ορίσει μια κατάλληλη **συνάρτηση κατακερματισμού**.
- Για παράδειγμα, για **ακέραια κλειδιά**, θα μπορούσαμε να έχουμε:

```
#define hash(v, M) (v % M)
```

✦ Συντελεστής Πλήρωσης (Load Factor)

► Ορισμός:

Ο συντελεστής πλήρωσης α ενός πίνακα κατακερματισμού (hash table) με μέγεθος M και N κατειλημμένες θέσεις ορίζεται ως:

$$\alpha = \frac{N}{M}$$

- Δηλαδή, εκφράζει το ποσοστό του πίνακα που είναι γεμάτο.
- Παίζει σημαντικό ρόλο στην αξιολόγηση της απόδοσης τεχνικών κατακερματισμού.

⚙️ Τύποι Απόδοσης (Performance Formulas)

Ας υποθέσουμε ότι έχουμε:

- Πίνακα κατακερματισμού T με μέγεθος M
- N κατειλημμένες θέσεις (δηλ. $\alpha = \frac{N}{M}$)

Ορίζουμε:

- C_N : Μέσος αριθμός προσπελάσεων (probes) κατά την επιτυχή αναζήτηση.
- C'_N : Μέσος αριθμός προσπελάσεων κατά την ανεπιτυχή αναζήτηση (ή εισαγωγή).

✦ Για τους παρακάτω τύπους υποθέτουμε ότι οι συναρτήσεις κατακερματισμού h και p είναι ομοιόμορφες και τυχαίες.



🚀 Αποδοτικότητα Γραμμικής Αναζήτησης (Linear Probing)

► Τύποι Απόδοσης:

Για ανοιχτή διευθυνσιοδότηση (open addressing) με γραμμική προσπέλαση (linear probing), ισχύουν οι εξής τύποι:

✅ Μέση επιτυχής αναζήτηση:

$$C_N = \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right)$$

❌ Μέση αποτυχημένη αναζήτηση ή εισαγωγή:

$$C'_N = \frac{1}{2} \left(1 + \left(\frac{1}{1 - \alpha} \right)^2 \right)$$

🔑 Σημειώσεις

- Οι παραπάνω τύποι ισχύουν με **αξιοπιστία** όταν ο πίνακας είναι γεμάτος **μέχρι 70%**, δηλαδή:

$$\alpha \leq 0.7$$

- Το αποτέλεσμα αυτό αποδείχθηκε από τον **Donald Knuth** το **1962**, και θεωρείται θεμελιώδες στην ανάλυση των τεχνικών κατακερματισμού.

🌀 Αποδοτικότητα Διπλού Κατακερματισμού (Double Hashing)

Για ανοιχτή διευθυνσιοδότηση (open addressing) με διπλό κατακερματισμό (double hashing), ισχύουν οι εξής τύποι απόδοσης:

➤ Επιτυχής αναζήτηση:

$$C_N = \frac{1}{\alpha} \cdot \ln \left(\frac{1}{1 - \alpha} \right)$$

➤ Ανεπιτυχής αναζήτηση:

$$C'_N = \frac{1}{1 - \alpha}$$

✅ Αυτό είναι ένα σημαντικό αποτέλεσμα που αποδείχθηκε από τους **Guibas και Szere**di το 1976.

✳️ Σημείωση: Το $\ln(x)$ είναι ο φυσικός λογάριθμος του x .

🔗 Αποδοτικότητα Κατακερματισμού με Συνδεδεμένες Λίστες (Separate Chaining)

Για κατακερματισμό με αλυσίδες (separate chaining) ισχύουν οι εξής τύποι:

➤ Επιτυχής αναζήτηση:

$$C_N = 1 + \frac{1}{2} \cdot \alpha$$

➤ Ανεπιτυχής αναζήτηση:

$$C'_N = \alpha$$

✳️ Σε έναν πίνακα με αλυσίδες μεγέθους M και N κλειδιά, η πιθανότητα να έχει κάθε λίστα αριθμό κλειδιών κοντά στον μέσο όρο $\frac{N}{M}$ είναι **πολύ κοντά στο 1** (δηλαδή πολύ πιθανό).

✓ Απόδειξη για $C'_N = \alpha$ (Κατακερματισμός με Συνδεδεμένες Λίστες)

► Ιδέα:

Θα αποδείξουμε ότι ο μέσος αριθμός συγκρίσεων σε μια ανεπιτυχή αναζήτηση σε έναν hash table με *separate chaining* είναι ίσος με τον συντελεστή πλήρωσης α .

► Περιγραφή:

- Έστω ότι έχουμε έναν hash πίνακα T .
- Όταν γίνεται κατακερματισμός ενός κλειδιού K , τοποθετείται σε μια αλυσίδα στη θέση $h(K)$.
- Έστω ότι η αλυσίδα αυτή περιέχει j κλειδιά. Είναι πιθανό να είναι και άδεια, δηλαδή $j = 0$.

► Υπόθεση:

Υποθέτουμε ότι η συνάρτηση κατακερματισμού είναι ομοιόμορφη και τυχαία (uniform and random). Δηλαδή:

- Τα N κλειδιά έχουν ομοιόμορφη κατανομή σε όλες τις M θέσεις του πίνακα.
- Έτσι, δημιουργούνται M ανεξάρτητες αλυσίδες.
- Κάποιες είναι άδειες, άλλες έχουν 1 ή περισσότερα κλειδιά.

► Συμπέρασμα:

- Ο μέσος όρος μήκους αλυσίδας είναι:

$$\frac{N}{M} = \alpha$$

- Άρα, κατά μέσο όρο, μια ανεπιτυχής αναζήτηση θα εξετάσει όλα τα στοιχεία της αλυσίδας που βρίσκεται στη θέση $h(K)$.
- Επομένως:

$$C'_N = \alpha$$

✓ Αυτή είναι η απλή απόδειξη ότι ο μέσος αριθμός συγκρίσεων σε αποτυχία για κατακερματισμό με αλυσίδες είναι ίσος με τον συντελεστή πλήρωσης α .

- Σημείωσε ότι οι προηγούμενοι τύποι δείχνουν πως η **απόδοση** ενός πίνακα κατακερματισμού **εξαρτάται μόνο από τον συντελεστή φόρτωσης (load factor)** και όχι από άλλες παραμέτρους.

III Θεωρητικά Αποτελέσματα: Εφαρμογή Τύπων

- Ας συγκρίνουμε τώρα την **απόδοση** των τεχνικών που μελετήσαμε για διάφορους **συντελεστές φόρτωσης**, χρησιμοποιώντας τους τύπους που παρουσιάστηκαν.
- Οι τύποι εφαρμόζονται σε πίνακα μεγέθους **997**.
- Τα **πειραματικά αποτελέσματα**, που προέκυψαν μετά την υλοποίηση των αλγορίθμων, είναι **παρόμοια** με τα θεωρητικά.

🔑 **Συντελεστής φόρτωσης (Load Factor):** Ορίζεται ως:

$$\alpha = N / M$$

Όπου:

α : Ο συντελεστής πλήρωσης (load factor).

N : Ο αριθμός των στοιχείων (κελιών) που είναι κατειλημμένα στον πίνακα.

M : Το συνολικό μέγεθος του πίνακα (πόσες θέσεις έχει συνολικά ο πίνακας).

Successful Search

Load Factors

	0.10	0.25	0.50	0.75	0.90	0.99
Separate chaining	1.05	1.12	1.25	1.37	1.45	1.49
Open/linear probing	1.06	1.17	1.50	2.50	5.50	50.5
Open/double hashing	1.05	1.15	1.39	1.85	2.56	4.65

📈 Μέσος Αριθμός Διευθύνσεων Ελέγχου (Probes)

- Τα κελιά του πίνακα δίνουν την τιμή C_h , δηλαδή τον **μέσο αριθμό διερευνήσεων (probe addresses)** για κάθε μία από τις τεχνικές κατακερματισμού.

🔑 Παρατηρήσεις

- Η τιμή του C_h **αυξάνεται** καθώς αυξάνεται ο **συντελεστής φόρτωσης (load factor) α** .
- Η **ανοιχτή διευθυνσιοδότηση με γραμμική διερεύνηση (linear probing)** παρουσιάζει **χειρότερη απόδοση** από τις άλλες δύο τεχνικές όταν ο α είναι μεγάλος.

- Η τεχνική με την **καλύτερη απόδοση** είναι η **χωριστή αλυσίδωση (separate chaining)**.

❏ Συμπέρασμα

Για εφαρμογές με μεγάλο αριθμό στοιχείων και υψηλό φορτίο στον πίνακα:

- **+ Προτίμηση** χωριστή αλυσίδωση για καλύτερη επίδοση.
- **–** Απόφυγε γραμμική διερεύνηση όταν ο πίνακας είναι σχεδόν γεμάτος.

Unsuccessful Search

Load Factors

	0.10	0.25	0.50	0.75	0.90	0.99
Separate chaining	0.10	0.25	0.50	0.75	0.90	0.99
Open/linear probing	1.12	1.39	2.50	8.50	50.5	5000.5
Open/double hashing	1.11	1.33	2.50	4.00	10.0	100.0

III Μέσος Αριθμός Διευθύνσεων Ελέγχου (C_n)

- Τα κελιά του πίνακα παρουσιάζουν την τιμή C_n , δηλαδή τον **μέσο αριθμό διευθύνσεων που ελέγχονται (probe addresses)** για κάθε τεχνική κατακερματισμού.

🔑 Παρατηρήσεις

- Η τιμή του C_n **αυξάνεται** καθώς αυξάνεται ο **συντελεστής φόρτωσης (load factor) α** .
- Η **ανοιχτή διευθυνσιοδότηση με γραμμική διερεύνηση (open addressing with linear probing)** έχει **χειρότερη επίδοση** από τις άλλες τεχνικές όταν ο α είναι μεγάλος.
- Η τεχνική με την **καλύτερη απόδοση** είναι η **χωριστή αλυσίδωση (separate chaining)**.

✓ Συμπέρασμα

- Για υψηλές τιμές α , η **χωριστή αλυσίδωση** προσφέρει σταθερά καλύτερη απόδοση.
- Η **γραμμική διερεύνηση** οδηγεί σε συμφόρηση (clustering) και αυξημένο αριθμό ελέγχων.

❏ Πολυπλοκότητα του Κατακερματισμού (Hashing)

💡 Βασικές Παραδοχές

- Υποθέτουμε ότι χρησιμοποιούμε πίνακα κατακερματισμού που **δεν γεμίζει ποτέ πάνω από 50%** ($\alpha \leq 0.50$).
- Όταν ο πίνακας γίνει πιο γεμάτος, τον **επεκτείνουμε** διπλασιάζοντας το μέγεθός του και **επανακατακερματίζουμε (rehash)** τα στοιχεία.

🔍 Αναζήτηση

- Σύμφωνα με τους πίνακες που παρουσιάστηκαν:
 - **Επιτυχής αναζήτηση**: το πολύ **1.50** συγκρίσεις κλειδιών.
 - **Ανεπιτυχής αναζήτηση**: το πολύ **2.50** συγκρίσεις κλειδιών.
 - Άρα η **αναζήτηση** έχει **χρονική πολυπλοκότητα**:
 $O(1)$
-

+ Εισαγωγή, ↻ Ενημέρωση & – Διαγραφή

- **Εισαγωγή** στοιχείου απαιτεί ίδιο αριθμό συγκρίσεων με ανεπιτυχή αναζήτηση $\Rightarrow O(1)$.
 - **Ανάκτηση** και **ενημέρωση** στοιχείων έχουν επίσης $O(1)$.
 - **Διαγραφή** στοιχείου έχει επίσης $O(1)$.
-

📋 Καταγραφή Όλων των Στοιχείων (Enumeration)

- Για να καταγράψουμε όλα τα στοιχεία σε **αύξουσα σειρά κλειδιών**, απαιτείται **ταξινόμηση**.
 - Καλή ταξινόμηση (π.χ. QuickSort) έχει χρόνο: $[\mathcal{O}(n \log n)]$
-

📈 Συντελεστής Φόρτωσης & Rehashing

- Σε **όλες** τις τεχνικές κατακερματισμού, πρέπει να διατηρούμε τον **συντελεστή φόρτωσης (α)**:
 - ≤ 0.5 για **open addressing**
 - ≤ 0.9 για **separate chaining**
 - Με **open addressing**, όταν το $\alpha > 0.5$:
 - Οι **συστάδες (clusters)** μεγαλώνουν.
 - Όταν $\alpha \approx 1$, οι λειτουργίες έχουν **γραμμική πολυπλοκότητα $O(n)$** .
-

✓ **Συμπέρασμα**: Ο κατακερματισμός προσφέρει **σταθερή πολυπλοκότητα** για βασικές λειτουργίες όταν εφαρμόζεται σωστά, αλλά απαιτεί προσοχή στον έλεγχο του α και χρήση **rehashing** όταν χρειάζεται.

📊 Συντελεστής Φόρτωσης & Rehashing (συνέχεια)

- Αν ο **συντελεστής φόρτωσης** (load factor) ενός πίνακα κατακερματισμού υπερβεί σημαντικά ένα προκαθορισμένο όριο, τότε **είναι συνηθισμένο να απαιτείται αλλαγή μεγέθους του πίνακα** ώστε να επανέλθει σε αποδεκτό επίπεδο.
 - Αυτή η διαδικασία ονομάζεται **rehashing (ανακατακερματισμός)** ή **dynamic hashing (δυναμικός κατακερματισμός)**.
 - Κατά τον rehashing:
 - Καλή πρακτική είναι το νέο μέγεθος του πίνακα να είναι τουλάχιστον **διπλάσιο** από το προηγούμενο.
-

? Open Addressing ή Separate Chaining;

◆ Open Addressing

- **Πλεονέκτημα:** Εξοικονόμηση μνήμης.
- **Μειονέκτημα:** Δεν είναι απαραίτητα ταχύτερο.

◆ Separate Chaining

- Σύμφωνα με θεωρητικά και πειραματικά αποτελέσματα:
 - Είναι **ανταγωνιστικό** ή ακόμα και **ταχύτερο** από άλλες μεθόδους, ειδικά όταν ο πίνακας είναι φορτωμένος.
- **Συμπέρασμα:** Αν η μνήμη **δεν αποτελεί περιοριστικό παράγοντα**, τότε η καλύτερη πολιτική επίλυσης συγκρούσεων είναι η **χωριστή αλυσίδωση (separate chaining)**.

Comparing the Performance of Some Table ADT Representations

	Initialize	Determine if full	Search Retrieve Update	Insert	Delete	Enumerate
Sorted array of structs	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
AVL tree of structs (or (2,4) tree or red-black tree)	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Hashing	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n \log n)$

□ Επιλογή Καλής Συνάρτησης Κατακερματισμού (Hash Function)

- Ιδανικά, μια **συνάρτηση κατακερματισμού (hash function)** θα πρέπει να **κατανέμει ομοιόμορφα και τυχαία** τα κλειδιά σε όλο το εύρος των θέσεων του πίνακα κατακερματισμού.
- Κάθε θέση στον πίνακα πρέπει να έχει **ίση πιθανότητα** να είναι ο "προορισμός" της συνάρτησης, όταν επιλέγεται τυχαία ένα κλειδί.

● Παράδειγμα Κακής Επιλογής Συνάρτησης Κατακερματισμού

💡 Σενάριο

- Έστω ότι τα κλειδιά μας είναι μεταβλητές μήκους έως 3 χαρακτήρων σε μια γλώσσα assembly.
- Οι χαρακτήρες είναι ASCII 8-bit.
- Άρα, κάθε κλειδί μπορεί να αναπαρασταθεί ως ένας **24-bit ακέραιος** (3 χαρακτήρες \times 8 bits).
- Χρησιμοποιούμε **open addressing με double hashing**.
- Επιλέγουμε το μέγεθος πίνακα $M = 2^8 = 256$.
- Ορίζουμε τη συνάρτηση κατακερματισμού ως:

$$h(K) = K \bmod 256$$

✗ Γιατί αυτή η συνάρτηση είναι προβληματική

- Το κλειδί έχει τη μορφή $C_3C_2C_1$ (3 χαρακτήρες).
- Η αριθμητική τιμή του είναι:
$$K = C_3 \cdot 256^2 + C_2 \cdot 256 + C_1$$
- Όταν υπολογίζουμε $h(K) = K \bmod 256$, το αποτέλεσμα είναι **μόνο ο χαρακτήρας C_1** (χαμηλότερος χαρακτήρας).
- Δηλαδή η συνάρτηση αγνοεί εντελώς τους χαρακτήρες C_2 και C_3 !

📦 Παράδειγμα

Αν έχουμε τα εξής κλειδιά:

- X1, X2, X3
- Y1, Y2, Y3

Αν υποθέσουμε ότι το τελικό ψηφίο (χαρακτήρας) τους είναι το ίδιο (π.χ. '1', '2', '3'):

Παράδειγμα

Αν έχουμε τα εξής κλειδιά:

- X1, X2, X3
- Y1, Y2, Y3

Αν υποθέσουμε ότι το τελικό ψηφίο (χαρακτήρας) τους είναι το ίδιο (π.χ. '1', '2', '3'):

Κλειδί	h(Κλειδί)
X1	1
Y1	1
X2	2
Y2	2
X3	3
Y3	3

👉 Όλα τα κλειδιά με το ίδιο τελευταίο γράμμα (π.χ. '1') χαρτογραφούνται στην ίδια διεύθυνση!

Αποτέλεσμα

- Δημιουργούνται **συστάδες (clusters)** σε παρακείμενες θέσεις του πίνακα.
- Αντί να **διασπείρονται ομοιόμορφα**, τα κλειδιά **συγκεντρώνονται**.
- Η απόδοση της μεθόδου κατακερματισμού υποβαθμίζεται σημαντικά.

Συμπέρασμα

Οι καλές συναρτήσεις κατακερματισμού:

- Πρέπει να λαμβάνουν υπόψη **όλα τα bits** του κλειδιού.
- Όχι μόνο ένα μέρος του (π.χ. το τελευταίο byte).
- Πρέπει να προσφέρουν **ομοιόμορφη κατανομή** για την αποφυγή συγκρούσεων και συστάδων.

Σχεδίαση Συναρτήσεων Κατακερματισμού (Hash Functions)

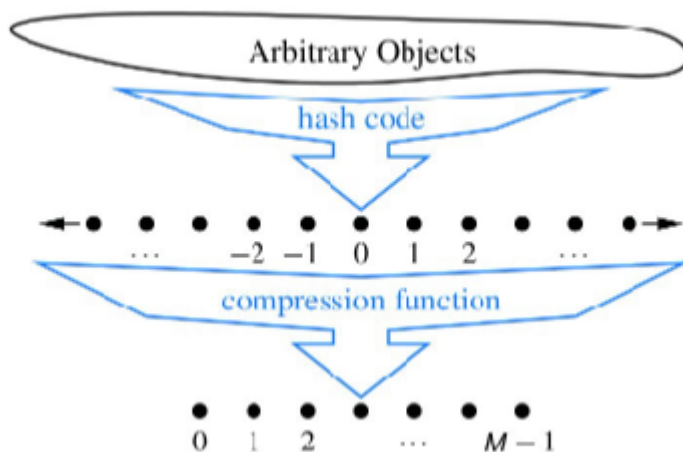
- Έστω M το μέγεθος του πίνακα κατακερματισμού (hash table).
- Η **αξιολόγηση μιας συνάρτησης κατακερματισμού $h(K)$** μπορεί να ιδωθεί ως αποτελούμενη από δύο βήματα:

1 Μετατροπή του κλειδιού K σε έναν ακέραιο

- Ο αριθμός αυτός λέγεται **κωδικός κατακερματισμού (hash code)**.

2 Συμπύεση του hash code σε τιμή εντός του εύρους $[0, M-1]$

- Αυτή η διαδικασία ονομάζεται **συνάρτηση συμπίεσης (compression function)**.



1.2 Κωδικοί Κατακερματισμού (Hash Codes)

□ Τι είναι ένας Hash Code;

- Η **πρώτη ενέργεια** που εκτελεί μια συνάρτηση κατακερματισμού είναι να μετατρέψει ένα **αυθαίρετο κλειδί K** σε έναν **ακέραιο αριθμό**.
- Αυτός ο αριθμός λέγεται **κωδικός κατακερματισμού (hash code)**.

ι Ιδιότητες του Hash Code

- Ο hash code **δεν χρειάζεται** να βρίσκεται στο εύρος $[0, M-1]$ και μπορεί ακόμα και να είναι **αρνητικός**.
- Ωστόσο, επιθυμούμε οι hash codes:
 - Να είναι **μοναδικοί** για διαφορετικά κλειδιά, ώστε να **αποφεύγονται συγκρούσεις (collisions)**.
 - Αν υπάρχουν συγκρούσεις σε αυτό το στάδιο, **καμία συνάρτηση συμπίεσης (compression function)** δεν μπορεί να τις διορθώσει.

□ Hash Codes για Τύπους Δεδομένων της C

- Οι hash codes για **στοιχεία τύπων της γλώσσας C** βασίζονται στην υπόθεση ότι **ο αριθμός bit κάθε τύπου δεδομένων είναι γνωστός**.

🔗 Αυτό είναι κρίσιμο για να σχεδιάσουμε αποδοτικούς και συμβατούς hash functions για struct, int, char, κ.λπ.

➡ Μετατροπή σε Ακέραιο (Converting to an Integer)

🔧 Βασική Ιδέα

- Για κάθε τύπο δεδομένων *D* που αναπαρίσταται με **λιγότερα ή ίσα bits** από όσα έχει ένας ακέραιος (int), μπορούμε:
 - Να θεωρήσουμε **την δυαδική του αναπαράσταση** ως έναν ακέραιο.
 - Να χρησιμοποιήσουμε απλά ένα cast σε `int` ως **hash code**.

✓ Εφαρμογή σε βασικούς τύπους C

- Για τύπους όπως `char`, `short int` και `int`, ένας καλός hash code μπορεί να προκύψει με:

```
(int)x
```

🧩 Long Int και Μεγαλύτερες Αναπαραστάσεις

⚠ Το Πρόβλημα με τον Long Int

- Συνήθως το `long int` έχει **διπλάσια bits** σε σχέση με το `int`.
- Αν απλά κάνουμε cast σε `int`, **χάνουμε τα πιο σημαντικά bits** (high-order bits), δηλαδή:
 - Χρησιμοποιείται μόνο το χαμηλό μισό της πληροφορίας → συγκρούσεις!

💡 Καλύτερη Προσέγγιση

- Χρησιμοποιούμε όλα τα bits, προσθέτοντας:
 - Τη δεκαδική αναπαράσταση των high-order bits +
 - Τη δεκαδική αναπαράσταση των low-order bits

📦 Γενική Μέθοδος για Σύνθετα Αντικείμενα

- Αν ένα αντικείμενο x μπορεί να ειπωθεί ως μια σειρά από ακέραιους:

$$x = (x_0, x_1, \dots, x_{k-1})$$

τότε μπορούμε να φτιάξουμε hash code με:

$$\text{hash}(x) = \sum_{i=0}^{k-1} x_i \quad (\text{αγνοώντας overflow})$$

💡 Παράδειγμα: Floating-point αριθμοί

- Δεκαδικός αριθμός μορφής `mE^e`:
 - Hash code μπορεί να είναι: `m + e` (ως long int)
 - Εναλλακτικά: XOR των components $\rightarrow m \wedge e$

+ Hash Codes με Άθροισμα (Summation Hash Codes)

⚠️ Όχι Καλή Επιλογή για Strings

- Ο hash κώδικας που βασίζεται στο **άθροισμα** των στοιχείων (π.χ. χαρακτήρες) **δεν είναι κατάλληλος** για μεταβλητού μήκους αντικείμενα, όπως οι συμβολοσειρές.

🤖 Γιατί;

- Τα strings μπορούν να θεωρηθούν ως πλειάδες: $x = (x_0, x_1, \dots, x_{\{k-1\}})$ όπου η **σειρά των στοιχείων** έχει σημασία.

🔑 Παράδειγμα Προβλήματος

Έστω hash function:

```
int hash = sum(ASCII(s[i]));
```

τότε:

Τα strings temp01 και temp10 θα έχουν τον ίδιο hash code, παρόλο που είναι διαφορετικά.

✓ Συμπέρασμα Χρειαζόμαστε καλύτερες τεχνικές για strings και παρόμοια αντικείμενα που λαμβάνουν υπόψη τη σειρά των στοιχείων.

Οι απλές αθροίσεις προκαλούν συχνές συγκρούσεις (collisions) σε κοινές κατηγορίες strings.

Πολυωνυμικοί Κωδικοί Κατακερματισμού (Polynomial Hash Codes)

Ορισμός

- Έστω ένας ακέραιος σταθερός αριθμός a , όπου $a \neq 1$.
- Για μια ακολουθία χαρακτήρων ή αριθμών x_0, x_1, \dots, x_{k-1} , μπορούμε να δημιουργήσουμε έναν κωδικό κατακερματισμού με το εξής πολυώνυμο:


$$h(x) = x_0 a^{k-1} + x_1 a^{k-2} + \dots + x_{k-2} a + x_{k-1}$$

- Αυτή η τεχνική ονομάζεται **πολυωνυμικός κατακερματισμός** (*polynomial hashing*).

Υπολογισμός με Αποδοτικό Τρόπο — Μέθοδος του Horner

Για αποδοτικό υπολογισμό της παραπάνω τιμής, χρησιμοποιούμε τη **μέθοδο του Horner**, που βασίζεται στην εξής ταυτότητα:

$$h(x) = x_{k-1} + a \cdot (x_{k-2} + a \cdot (x_{k-3} + \dots + a \cdot (x_1 + a \cdot x_0) \dots))$$

 Αντί να κάνουμε k υψώσεις σε δύναμη και πολλαπλασιασμούς, μπορούμε να εκτελέσουμε:

- $k - 1$ προσθέσεις
- $k - 1$ πολλαπλασιασμούς

Πλεονεκτήματα

- Αποδοτικός υπολογισμός σε χρόνο $O(k)$.
- Καλύτερη κατανομή τιμών κατακερματισμού.
- Χρησιμοποιείται σε πολλές εφαρμογές όπως:
 - Έλεγχος ισότητας υποσυμβολοσειρών (π.χ. Rabin-Karp αλγόριθμος).
 - Συστήματα ανίχνευσης διπλότυπων (e.g. string deduplication).

Παράδειγμα (προαιρετικό)

Αν θες, μπορώ να σου δείξω ένα αριθμητικό παράδειγμα με συγκεκριμένα x_i και a , για να δεις πώς λειτουργεί βήμα προς βήμα η μέθοδος Horner στην πράξη. Θες να το κάνουμε αυτό;

□ Πολυωνυμικοί Hash Κώδικες (Polynomial Hash Codes)

 Ιδέα

- Ένας **πολυωνυμικός hash κώδικας** χρησιμοποιεί **πολλαπλασιασμό με διαφορετικές δυνάμεις** ώστε κάθε στοιχείο να επηρεάζει με μοναδικό τρόπο το τελικό αποτέλεσμα.
- Βοηθά να **κατανεμηθούν τα κλειδιά πιο ομοιόμορφα** στον πίνακα.

⚙ Υλοποίηση και Υπερχείλιση

- Ο υπολογισμός γίνεται με ακέραιους, οπότε φυσικά θα προκύψουν **υπερχειλίσσεις (overflow)**.
- Αυτό **δεν μας απασχολεί πολύ**, γιατί θέλουμε απλώς **καλή κατανομή (spread)**, όχι αριθμητική ακρίβεια.
- Παρόλα αυτά, πρέπει να επιλέξουμε τη σταθερά **a** με τρόπο ώστε να έχει **μη μηδενικά χαμηλόβαθμα bits**, για να διατηρεί όσο το δυνατόν περισσότερη πληροφορία ακόμη και μετά από overflow.

📦 Πειραματικά Αποτελέσματα

- Σε ένα λεξικό με **πάνω από 50.000 αγγλικές λέξεις**, χρησιμοποιώντας τις τιμές:
 - **a = 33, 37, 39, 41**
 - Παρατηρήθηκαν **λιγότερες από 7 συγκρούσεις** σε κάθε περίπτωση!

⚡ Επιτάχυνση

- Για πολύ μεγάλες συμβολοσειρές, μπορεί να υπολογίζεται ο hash μόνο σε **υποσύνολο χαρακτήρων** για μεγαλύτερη ταχύτητα χωρίς σημαντική απώλεια απόδοσης.

💡 Παράδειγμα (σε ψευδοκώδικα)

```
int hash(char *s) {  
    int h = 0;  
    int a = 33; // ή 37, 39, 41  
    for (int i = 0; s[i] != '\0'; i++)  
        h = a * h + s[i]; // overflow OK  
    return h;  
}
```

✓ Συμπέρασμα Οι πολυωνυμικοί hash κώδικες είναι απλοί, αποδοτικοί και πρακτικά πολύ αποτελεσματικοί, ειδικά για strings.

🌀 Πλήρης Συνάρτηση Κατακερματισμού (Polynomial Hash Function)

Η παρακάτω συνάρτηση είναι μια **πλήρης hash function**, όχι απλώς ένας hash κώδικας. Δηλαδή, **επιστρέφει τη θέση** στον πίνακα hash (0 έως **M-1**), όχι απλώς έναν αυθαίρετο ακέραιο.

📄 Κώδικας σε C

```
int hash(char *K)
{
    int h = 0, a = 33;
    for (; *K != '\0'; K++)
        h = (a * h + *K) % M; // M: μέγεθος πίνακα
    return h;
}
```

🔍 Επεξήγηση

- h: η μεταβλητή που συσσωρεύει την τιμή του hash.
- a = 33: η βάση του πολυωνύμου, επιλεγμένη επειδή δίνει καλό "σπάσιμο" τιμών για αγγλικές λέξεις.
- K: κάθε χαρακτήρας της συμβολοσειράς.
- M: το μέγεθος του πίνακα hash (π.χ., 997).

🔗 Παρατηρήσεις Η χρήση του % M στο τέλος κάθε βήματος εξασφαλίζει ότι η τιμή παραμένει μέσα στα όρια του πίνακα.

Το a = 33 είναι μια κοινή και αποδεδειγμένα αποδοτική τιμή για strings.

Η συνάρτηση λειτουργεί καλά για αλφαριθμητικά κλειδιά (π.χ., ονόματα, usernames, κτλ).

💬 Σχόλια για την Πολυωνυμική Συνάρτηση Κατακερματισμού

- Η προηγούμενη συνάρτηση παίρνει ως είσοδο έναν **δείκτη σε πίνακα χαρακτήρων**, δηλαδή μια **null-τερματισμένη συμβολοσειρά** K, και υπολογίζει τη συνάρτηση κατακερματισμού για αυτή.
- Η εντολή:

```
h = (a * h + *K) % M;
```

ενημερώνει την τιμή του h χρησιμοποιώντας:

την προηγούμενη τιμή του h,

τη βάση a (συνήθως 33), και

την ASCII τιμή του τρέχοντος χαρακτήρα *K.

💬 Σχόλια για τη Συνάρτηση Κατακερματισμού

- Η παραπάνω συνάρτηση παίρνει ως είσοδο έναν **δείκτη σε πίνακα χαρακτήρων**.
- Ο δείκτης αυτός δείχνει σε μια **null-τερματισμένη συμβολοσειρά** K.
- Η συνάρτηση υπολογίζει μια **συνάρτηση κατακερματισμού (hash function)** για τη συμβολοσειρά αυτή.

🔑 Κεντρική Εντολή:

```
h = (a * h + *K) % M;
```

Ενημερώνει την τιμή του h με βάση:

την προηγούμενη τιμή του h ,

τη βάση a (συνήθως 33 ή 37),

και την ASCII τιμή του τρέχοντος χαρακτήρα $*K$.

➡ Αυτό γίνεται για κάθε χαρακτήρα της συμβολοσειράς, δημιουργώντας έναν πολυωνυμικό κώδικα κατακερματισμού με βάση τις τιμές των χαρακτήρων και τη σειρά τους.

✓ Η χρήση του $\% M$ (modulo) περιορίζει την τελική τιμή του hash μέσα στο εύρος $[0, M-1]$, που αντιστοιχεί στα έγκυρα indices του πίνακα hash.

12/34 Πολυωνυμικοί Κώδικες Κατακερματισμού (συνέχεια)

🧠 Θεωρητικά:

- Πρώτα υπολογίζουμε έναν **πολυωνυμικό κώδικα κατακερματισμού**.
- Έπειτα εφαρμόζουμε τη **συνάρτηση συμπίεσης** μέσω του **modulo M** (δηλαδή το υπόλοιπο της ακέραιας διαίρεσης με το M).

🧑💻 Πρακτικά (στην υλοποίηση που είδαμε):

- Το **$\% M$** εφαρμόζεται **σε κάθε βήμα** του υπολογισμού, **όχι μόνο στο τέλος**.

□ Γιατί είναι το ίδιο;

Για όλους τους μη αρνητικούς ακεραίους a, b, x, M ισχύει:

$$(a * x \% M + b \% M) \% M = (a * x + b) \% M$$

➡ Δηλαδή, μπορούμε να εφαρμόζουμε **mod M** σε κάθε ενδιάμεσο βήμα **χωρίς να αλλάζει το τελικό αποτέλεσμα**.

✓ Πλεονεκτήματα της ενδιάμεσης χρήσης **$\% M$**

- Αποφεύγονται **προβλήματα υπερχείλισης** (overflow), ιδιαίτερα όταν έχουμε **μεγάλες συμβολοσειρές**.
- Αν υπολογίσουμε πρώτα το πολυώνυμο και κάνουμε **$\% M$ μόνο στο τέλος**, μπορεί το αποτέλεσμα να υπερβεί τα όρια της αναπαράστασης ενός ακεραίου στη C.

□ **Δοκίμασε** να χρησιμοποιήσεις τη δεύτερη προσέγγιση με μια πολύ μεγάλη συμβολοσειρά — θα δεις ότι μπορεί να προκύψουν λάθη λόγω υπερχείλισης!

🔄 Κώδικες Κατακερματισμού με Κυκλική Ολίσθηση (Cyclic Shift Hash Codes)

□ Τι είναι;

Μια **παραλλαγή** του πολυωνυμικού κώδικα κατακερματισμού, όπου **αντικαθιστούμε** τον πολλαπλασιασμό με μια **κυκλική ολίσθηση** (cyclic shift) ενός ενδιάμεσου αθροίσματος κατά συγκεκριμένο αριθμό bit.

🔍 Σημεία-κλειδιά:

- Δεν έχει ιδιαίτερη **αριθμητική σημασία**, δηλαδή δεν αντιστοιχεί σε γνωστή μαθηματική πράξη.
- **Σκοπός** της είναι να **ποικίλει τα bits** του υπολογισμού του κώδικα κατακερματισμού.

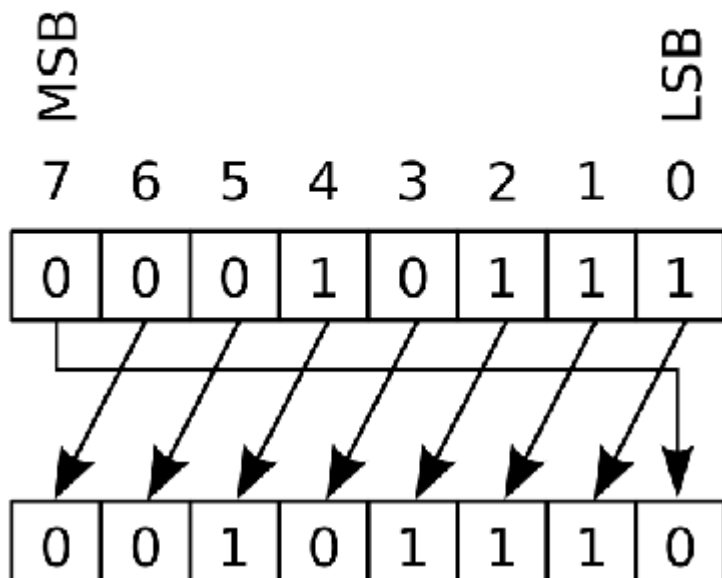
🤔 Γιατί το κάνουμε;

Η κυκλική ολίσθηση είναι:

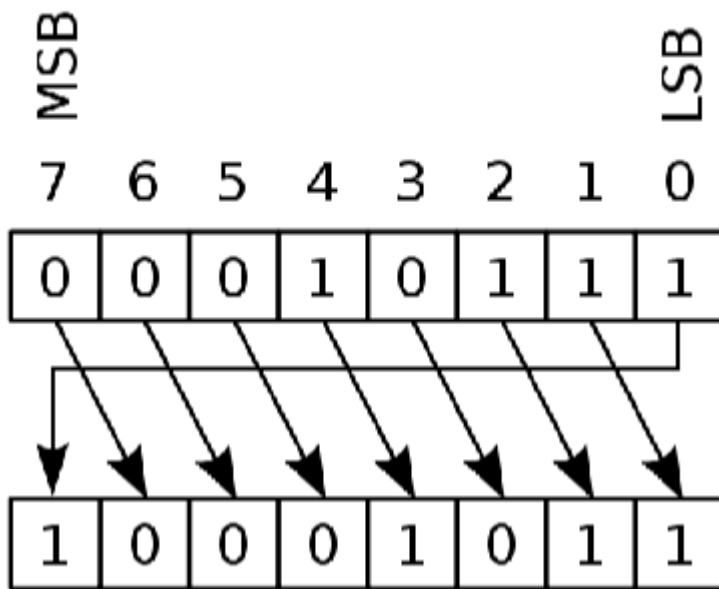
- **Γρήγορη** (bitwise operation)
- **Απλή** στην υλοποίηση
- Πολύ **αποτελεσματική** στο να δημιουργεί μια καλή κατανομή (spread) των hash values

✓ Με άλλα λόγια, χρησιμοποιείται για να αυξήσει τη **μοναδικότητα** των κωδικών κατακερματισμού — ακόμα και μικρές αλλαγές στην είσοδο να οδηγούν σε σημαντικά διαφορετικά hash codes.

1-Bit Left Cyclic Shift



1-Bit Right Cyclic Shift



🔗 Κώδικες Κατακερματισμού με Κυκλική Ολίσθηση (συνέχεια)

✂ Πώς γίνεται στην C;

Στην C, μια **κυκλική ολίσθηση bit** (cyclic bit shift) μπορεί να υλοποιηθεί με έξυπνη χρήση των τελεστών bitwise `<<` (αριστερή ολίσθηση) και `>>` (δεξιά ολίσθηση).

🔑 Παράδειγμα (5-bit κυκλική ολίσθηση):

```
unsigned int cyclicShift(unsigned int h) {
    return (h << 5) | (h >> (32 - 5));
}
```

🧠 Τι κάνει αυτός ο κώδικας;

Το `h << 5` μετακινεί τα bits 5 θέσεις αριστερά.

Το `h >> (32 - 5)` μετακινεί τα bits 27 θέσεις δεξιά.

Το bitwise OR (`|`) ενώνει αυτά τα δύο αποτελέσματα για να προσομοιώσει μια κυκλική μετατόπιση 5-bit.

ℹ Σημειώσεις: Λειτουργεί σωστά για 32-bit ακεραίους.

Είναι ένας γρήγορος τρόπος για να διασπείρει τα bits και να παράγει καλύτερους hash codes. ✓

Χρησιμοποιείται συχνά σε συνδυασμό με επαναληπτική επεξεργασία χαρακτήρων για την κατασκευή hash functions που διαχειρίζονται strings και άλλα σύνθετα αντικείμενα.

🔗 Κώδικες Κατακερματισμού με Κυκλική Ολίσθηση (συνέχεια)

✓ Παράδειγμα σε C:

```
int hashCode(const char *p, int len) {  
    unsigned int h = 0;  
    int i;  
    for (i = 0; i < len; i++) {  
        h = (h << 5) | (h >> 27); // Κυκλική ολίσθηση 5-bit  
        h += (unsigned int) p[i]; // Προσθήκη επόμενου χαρακτήρα  
    }  
    return h;  
}
```

🔍 Τι κάνει αυτή η συνάρτηση;

- **Είσοδος:** Δέχεται έναν δείκτη `p` προς συμβολοσειρά χαρακτήρων (π.χ. string) και το μήκος της.
- **Αρχικοποίηση:** Το `h` ξεκινάει από 0. Θα αποθηκεύσει το τελικό hash code.
- **Κυκλική Ολίσθηση:** `h = (h << 5) | (h >> 27);`
 - Ολισθαίνει τα bits του `h` 5 θέσεις αριστερά.
 - Ταυτόχρονα, φέρνει τα 5 πιο αριστερά bits πίσω στις χαμηλότερες θέσεις (27 δεξιά).
- **Συσσώρευση χαρακτήρων:** Κάθε χαρακτήρας `p[i]` προστίθεται στο `h`.
- **Επιστροφή:** Επιστρέφει τον τελικό ακέραιο hash code.

💡 Πλεονεκτήματα:

- Κατάλληλη για strings μεταβλητού μήκους.
- Η σειρά χαρακτήρων παίζει ρόλο (σε αντίθεση με απλούς αθροιστικούς hash codes).
- Η κυκλική ολίσθηση βοηθάει στην καλύτερη κατανομή τιμών (λιγότερες συγκρούσεις).

📝 Σχόλια για την Κυκλική Ολίσθηση στον Hash Code

Η γραμμή:

```
h = (h << 5) | (h >> 27);
```

εκτελεί έναν 5-bit κυκλικό μετασχηματισμό (cyclic shift) του ενδιάμεσου αποτελέσματος h.

🔗 Αναλυτικά:

`h << 5:`

Ολισθαίνει το `h` αριστερά κατά 5 bits.

Αυτό μετακινεί τα υψηλότερα bits προς τα πιο σημαντικά bits και "ανοίγει χώρο" στα δεξιά.

`h >> 27:`

Ολισθαίνει το `h` δεξιά κατά 27 bits (32 - 5, καθώς unsigned int είναι 32 bits).

Αυτό μετακινεί τα υψηλότερα bits πίσω στο χαμηλότερο άκρο.

| (bitwise OR):

Συνδυάζει τα αποτελέσματα της αριστερής και δεξιάς ολίσθησης.

Εξασφαλίζει ότι τα bits που "χάθηκαν" με την αριστερή ολίσθηση επανεισάγονται από την δεξιά πλευρά — κυκλική ροή των bits.

✓ Τελικό αποτέλεσμα:

Έχουμε μια κυκλική ολίσθηση 5 θέσεων, η οποία:

- Συμβάλλει στην καλύτερη κατανομή των bits του hash code.
- Ελαχιστοποιεί πιθανότητες συγκρούσεων (collisions).
- Προστατεύει την πληροφορία από απώλειες λόγω απλών ολισθήσεων.

🔗 Σημείωση: Αυτή η τεχνική είναι πολύ χρήσιμη όταν δουλεύουμε με συμβολοσειρές και θέλουμε να δίνουμε βάρος στη σειρά των χαρακτήρων αλλά και στη θέση κάθε χαρακτήρα.

Experiment

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

Πείραμα (συνέχεια)

- Ο προηγούμενος πίνακας παρουσιάζει μια σύγκριση της συμπεριφοράς συγκρούσεων του hash με κυκλική ολίσθηση (cyclic-shift hash code), όπως εφαρμόστηκε σε μια λίστα με 230.000 αγγλικές λέξεις.
- Η στήλη "Total" καταγράφει το συνολικό πλήθος λέξεων που συγκρούονται με τουλάχιστον μία άλλη.
- Η στήλη "Max" καταγράφει το μέγιστο πλήθος λέξεων που έχουν το ίδιο hash code.
- Το πείραμα αυτό οδήγησε στην επιλογή της τιμής 5 για την κυκλική ολίσθηση στον κώδικά μας, καθώς σε αυτήν την περίπτωση έχουμε τις λιγότερες συνολικές συγκρούσεις.
- Σημείωση: Αν η κυκλική ολίσθηση είναι 0, τότε ο αλγόριθμος απλώς αθροίζει τους χαρακτήρες, όπως το απλό summation hash code.

Hash Codes για Αριθμούς Κινητής Υποδιαστολής (Floating Point)

- Μπορούμε να πετύχουμε καλύτερο hash code για floating-point αριθμούς απ' ό,τι με απλή μετατροπή σε `int` ως εξής:
- Αν υποθέσουμε ότι ένας `char` αποθηκεύεται ως 8-bit byte, τότε μπορούμε να θεωρήσουμε έναν 32-bit float ως πίνακα 4 χαρακτήρων.
- Έπειτα μπορούμε να χρησιμοποιήσουμε τις συναρτήσεις hash που συζητήσαμε για strings.

Συναρτήσεις Συμπίεσης (Compression Functions)

- Αφού υπολογίσουμε έναν ακέραιο hash code για το κλειδί K , παραμένει το ζήτημα της μετατροπής του σε έγκυρο δείκτη πίνακα στο εύρος 0 έως $M - 1$.
- Αυτό μπορεί να επιτευχθεί με τις εξής μεθόδους:

(εδώ μπορείς να προσθέσεις τις μεθόδους όπως modulo, MAD method, κ.λπ.)

□ Compression Functions: The Division Method

Η μέθοδος διαίρεσης (division method) είναι μια διαδεδομένη τεχνική για την κατασκευή συναρτήσεων συμπίεσης σε πίνακες κατακερματισμού.

Division Method

Μια καλή επιλογή συμπίεσης είναι η εξής:

- Επιλέγουμε το μέγεθος του πίνακα κατακερματισμού M ως **πρώτο αριθμό** (πολύ σημαντικό!)

- Ορίζουμε:

$$h(K) = K \% M$$

όπου $h(K)$ είναι η συμπιεσμένη τιμή του hash code του K .

- Στο πλαίσιο του **double hashing**, χρησιμοποιούμε:

$$p(K) = \max(1, \lfloor K/M \rfloor)$$

? Γιατί το M πρέπει να είναι ΠΡΩΤΟΣ αριθμός;

- Όταν το M είναι **πρώτος αριθμός**, η modulo συνάρτηση κατακερματισμού βοηθά στο να **κατανεμηθούν ομοιόμορφα** οι τιμές του hash στον πίνακα.
- Αν M **δεν είναι πρώτος**, υπάρχει **αυξημένος κίνδυνος σύγκρουσης** λόγω επαναλαμβανόμενων μοτίβων στην κατανομή των τιμών.

⚠ Παράδειγμα:

- Αν εισάγουμε τα hash codes $\{200, 205, 210, 215, \dots, 600\}$ σε πίνακα με $M = 100$, κάθε hash code θα συγκρουστεί με άλλες **τρεις** τιμές.
- Αν το $M = 101$ (πρώτος αριθμός), τότε **δεν υπάρχει καμία σύγκρουση**.

? Γιατί το M πρέπει να είναι ΠΡΩΤΟΣ αριθμός;

- Όταν το M είναι **πρώτος αριθμός**, η modulo συνάρτηση κατακερματισμού βοηθά στο να κατανεμηθούν ομοιόμορφα οι τιμές του hash στον πίνακα.
- Αν M δεν είναι πρώτος, υπάρχει **αυξημένος κίνδυνος σύγκρουσης** λόγω επαναλαμβανόμενων μοτίβων στην κατανομή των τιμών.

⚠ Παράδειγμα:

- Αν εισάγουμε τα hash codes `{200, 205, 210, 215, ..., 600}` σε πίνακα με $M = 100$, κάθε hash code θα συγκρουστεί με άλλες **τρεις** τιμές.
- Αν το $M = 101$ (πρώτος αριθμός), τότε **δεν υπάρχει καμία σύγκρουση**.

🛡 Προφυλάξεις

- Αν οι τιμές των κλειδιών ακολουθούν κάποιο **επαναλαμβανόμενο μοτίβο** της μορφής:

$$iM + j$$

για πολλά διαφορετικά i , τότε και πάλι μπορεί να προκύψουν συγκρούσεις — ακόμα και αν το M είναι πρώτος αριθμός.

⚠ Προφυλάξεις (συνέχεια)

- Αν r είναι η **βάση (radix)** του χαρακτήρα για τα κλειδιά (π.χ., 256 για ASCII),
- και k, a είναι **μικροί ακέραιοι αριθμοί**,

Τότε:

Δεν θα πρέπει να επιλέξουμε έναν πρώτο αριθμό M της μορφής:

$$M = r^k \pm a$$

Γιατί;

Οι τιμές hash που παράγονται από τέτοια M μπορεί να συγχρονιστούν με μοτίβα στα ίδια τα κλειδιά (ιδιαίτερα όταν αυτά προέρχονται από αλφαριθμητικά), οδηγώντας σε συχνές συγκρούσεις.

🧠 Η επιλογή του M είναι κρίσιμη για την αποτελεσματικότητα του hash table. Αποφύγετε τιμές που σχετίζονται στενά με τη δομή των ίδιων των κλειδιών.

🔴 Παράδειγμα — Κακή Σχεδίαση Συνάρτησης Κατακερματισμού

📊 Παράδειγμα

- Θεωρούμε ξανά κλειδιά αποτελούμενα από 3 χαρακτήρες: $C_3C_2C_1$, όπου κάθε χαρακτήρας είναι 8-bit ASCII, άρα ανήκει σε ένα σύνολο ακέραιων τιμών με βάση (radix) $r = 256$.
- Επιλέγουμε μέγεθος πίνακα κατακερματισμού $M = 2^{16} + 1 = 65537$.
- Ορίζουμε τη συνάρτηση κατακερματισμού:

$$h(C_3C_2C_1) = (C_2C_1 - C_3) \cdot 256$$

⚠️ Πρόβλημα

- Για τέτοιο M και συγκεκριμένο τύπο συνάρτησης, οι τιμές του $h(K)$ καταλήγουν να είναι απλά αθροίσματα ή διαφορές γινομένων των χαρακτήρων.
- Αυτό δεν εξασφαλίζει ομοιόμορφη κατανομή των κλειδιών στον πίνακα, με αποτέλεσμα:
 - Συστάδες (clusters) κλειδιών.
 - Μικρότερη αποδοτικότητα στις αναζητήσεις.

🔄 Διπλός Κατακερματισμός (Double Hashing)

✅ Καλή Επιλογή για Δεύτερη Συνάρτηση $p(K)$

- Σε τεχνικές διπλού κατακερματισμού, μια καλή επιλογή για την $p(K)$ είναι:

$$p(K) = Q - (K \bmod Q)$$

Όπου:

- Q είναι ένας πρώτος αριθμός και
- $Q < M$

📊 Ιδιότητες

- Οι τιμές του $p(K)$ κυμαίνονται από 1 έως Q .
- Εξασφαλίζει ότι το βήμα προώθησης στον πίνακα δεν είναι πολλαπλάσιο του μεγέθους του πίνακα, άρα αποφεύγουμε κυκλικές συγκρούσεις.
- Βοηθά στη μεγαλύτερη διασπορά των κλειδιών στον πίνακα.

Η Μέθοδος MAD (Multiply And Divide)

Περιγραφή

Η μέθοδος MAD είναι μια πιο εξελιγμένη συνάρτηση συμπίεσης (compression function) που χρησιμοποιείται σε hashing για να μειώσει τα επαναλαμβανόμενα μοτίβα σε ένα σύνολο ακέραιων κωδικών.

Συνάρτηση

Η μορφή της είναι:

$$h(i) = ((a \cdot i + b) \bmod p) \bmod M$$

Όπου:

- M = μέγεθος του πίνακα κατακερματισμού (hash table)
- p = πρώτος αριθμός μεγαλύτερος από το M
- $a, b \in [0, p - 1]$, τυχαία μη αρνητικά ακέραια
- $a > 0$

Πλεονεκτήματα

- Εξομαλύνει την κατανομή των hash codes.
- Είναι κατάλληλη για περιπτώσεις όπου τα αρχικά integer keys έχουν κάποιο επαναλαμβανόμενο μοτίβο.
- Πολύ χρήσιμη όταν η απλή $\bmod M$ συνάρτηση δεν είναι αρκετά καλή.

Εφαρμογές των Hash Tables

Οι πίνακες κατακερματισμού χρησιμοποιούνται σε πολλούς τομείς της Πληροφορικής:

Παραδείγματα:

- Βάσεις Δεδομένων (συμπεριλαμβανομένου του εξωτερικού hashing για μεγάλα αρχεία)
- Κρυπτογραφία
- Πίνακες Συμβόλων (Symbol Tables) στους μεταγλωττιστές (compilers)
- Caches σε προγράμματα περιήγησης (browser caches)
- Συστήματα peer-to-peer και torrents (με χρήση κατανεμημένων hash tables - DHTs)