HASHING

***ΒΑΣΙΚΗ ΙΔΕΑ*:** Η συνάρτηση κατατεμαχισμού, γνωστή και ως συνάρτηση κατακερματισμού, είναι μια μαθηματική συνάρτηση που δέχεται ως είσοδο κάποιο δεδομένο τυχαίου μεγέθους και επιστρέφει ένα ακέραιο σταθερού μεγέθους αναπαράστασης. Οι τιμές που επιστρέφει η συνάρτηση κατατεμαχισμού ονομάζονται τιμές κατατεμαχισμού (hash values), κώδικες κατατεμαχισμού (hash codes), αθροίσματα κατατεμαχισμού (hash sums) ή απλά τιμές κατατεμαχισμού (hashes).

Οι τιμές αυτές θα πρέπει να είναι διαφορετικές για διαφορετική είσοδο, καθώς η κύρια χρησιμότητα αυτών των συναρτήσεων είναι να ταυτοποιούν τα δεδομένα.

Μια συνάρτηση κατατεμαχισμού μπορεί να αντιστοιχίζει δύο ή περισσότερες εισόδους στην ίδια τιμή κατατεμαχισμού (collision). Στις περισσότερες εφαρμογές είναι επιθυμητή η ελαχιστοποίηση αυτών των συγκρούσεων. Αυτό σημαίνει ότι η συνάρτηση κατατεμαχισμού θα πρέπει να αντιστοιχίζει κάθε είσοδο σε διαφορετική τιμή κατατεμαχισμού. Ανάλογα με την εφαρμογή χρήσης, η συνάρτηση κατατεμαχισμού σχεδιάζεται με διαφορετικές προδιαγραφές.

Οι συναρτήσεις κατατεμαχισμού κυρίως χρησιμοποιούνται σε πίνακες κατατεμαχισμού (hash tables), για γρήγορη εύρεση εγγραφών σε βάσεις δεδομένων. Για παράδειγμα σε ένα λεξικό έχουμε τις λέξεις-κλειδιά και τους αντίστοιχους ορισμούς-περιγραφές. Η συνάρτηση κατατεμαχισμού μπορεί να εξυπηρετήσει αντιστοιχώντας τις λέξεις-κλειδιά με τις αντίστοιχες τιμές κατατεμαχισμού.

Τεχνικές Διαχείρισης Συγκρούσεων:

1)*Ξεχωριστών Αλυσίδων (Separate Chaining):* Η θέση A[j] του πίνακα κατακερματισμού δεν περιέχει ένα στοιχείο αλλά ένα δείκτη σε μια δυναμική δομή (που υλοποιεί ένα λεξικό), (π.χ. μια λίστα, η οποία περιέχει κάθε στοιχείο με κλειδί K τέτοιο ώστε h(K) = j)

2) *Κατακερματισμός με Ανοικτή Διευθυνσιοδότηση (Open Addressing)*: Η μέθοδος στοχεύει στην ορθή διαχείριση του ελεύθερου χώρου του ίδιου του πίνακα κατακερματισμού.

Όλα τα κλειδιά αποθηκεύονται στον πίνακα χωρίς ωστόσο να σχηματίζονται αλυσίδες (δεν χρησιμοποιούνται δείκτες).

Για κάθε κλειδί ελέγχεται μία ακολουθία από θέσεις του πίνακα, που ονομάζεται ακολουθία εξέτασης. Η πρώτη διαθέσιμη θέση του πίνακα με τη σειρά που καθορίζεται από την ακολουθία αυτή θα στεγάσει το κλειδί.

Η ακολουθία καθορίζεται βάσει κάποιου κανόνα. Τέτοιοι κανόνες είναι:

Α) Γραμμική Αναζήτηση (linear probing): Με αυτή την μέθοδο, αν υπάρξει σύγκρουση σε μια θέση του πίνακα ελέγχουμε την επόμενη θέση μέχρι να μην υπάρχει σύγκρουση.

Β) Τετραγωνική Αναζήτηση (quadratic probing): Με αυτή την μέθοδο, αν υπάρξει σύγκρουση σε μια θέση του πίνακα ελέγχουμε θέση i+i^2 μέχρι να μην υπάρχει σύγκρουση (π.χ. αν υπάρξει σύγκρουση στη θέση i=1, ελέγχουμε την θέση 1+1^2=2. Μετά τη θέση 1+2^2=5 κλπ).

Γ)Διπλός Κατακερματισμός (double hashing): Σε αυτή την μέθοδο χρησιμοποιούμε και δεύτερη συνάρτηση κατακερματισμού. Αν υπάρξει σύγκρουση σε μια θέση του πίνακα ελέγχουμε θέση i+i^hash2 μέχρι να μην υπάρχει σύγκρουση (π.χ. αν υπάρξει σύγκρουση στη θέση i=5, ελέγχουμε την θέση 5+1^hash2. Μετά τη θέση 5+2^ hash2 κλπ).

ΚΩΔΙΚΑΣ ΑΣΚΗΣΗΣ:

Στην συγκεκριμένη άσκηση, για να φτιάξω το Hash Table δημιούργησα μια κλάση (HashTable) η οποία αποτελείται από 13 συναρτήσεις.

Σύμφωνα με την άσκηση το μέγεθος του πίνακα κατακερματισμού διπλασιάζεται όταν γεμίζει περίπου το ήμισυ αυτού. Αυτό επιτυγχάνεται μέσω της συνάρτησης def resize(). Σε αυτή τη συνάρτηση διπλασιάζω την χωρητικότητα του πίνακα INITIAL_CAPACITY, μηδενίζω τη μεταβλητή size με την οποία μετράω της θέσης που έχουν καταληφθεί στον πίνακα και ουσιαστικά δημιουργώ νέο πίνακα στον οποίο αντιγράφω όλα τα tuples που υπάρχουν στον παλιό old_table.

Παράλληλα η συνάρτηση def hash(key) είναι η συνάρτηση κατακερματισμού που επιστρέφει την θέση την οποία πρέπει να πάρουν τα δεδομένα με ετικέτα key.

Με την συνάρτηση def insert(key,value1,value2) εισάγω τα δεδομένα που βρίσκω στον Η όπου value1=cost και value2 μια λίστα 6 ακεραίων (πέντε μηδενικών και ενός άσου η θέση του οποίου προσδιορίζει την μέρα). Για κάθε εισαγωγή αυξάνεται ο μετρητής size που όπως είπαμε υποδηλώνει τον αριθμό των θέσεων που καταλαμβάνονται από δεδομένα. Στη συνέχεια ελέγχει αν η θέση που προκύπτει από def hash(key) είναι κατειλημμένη. Αν είναι άδεια τοποθετεί στην συγκεκριμένη θέση το tuple με (key,value1,value2). Σε αντίθετη περίπτωση υπάρχουν δύο περιπτώσεις. Πρώτα ελέγχει αν το key που δίνεται είναι ίδιο με αυτό που καταλαμβάνει την συγκεκριμένη θέση. Αν ναι, το tuple που θα μπει σε αυτή τη θέση είναι το (key,tc,totaldays) όπου το key είναι το ίδιο με πριν, το tc (total cost το άθροισμα του προηγούμενου με το value1 της εισόδου) αντικαθιστά το προηγούμενο και το totaldays είναι μια ανανεωμένη λίστα που προκύπτει από την πρόσθεση των δεδομένων της υπάρχουσας με αυτής που εισάγεται. Προσοχή απαιτείται σε αυτή την περίπτωση με τον μετρητή που μετράει αριθμό των θέσεων που καταλαμβάνονται από δεδομένα. Πρέπει να μειωθεί κατά 1 αφού στην αρχή της συνάρτησης τον έχουμε αυξήσει χωρίς στην συγκεκριμένη περίπτωση να πρέπει αφού δεν θα καλυφθεί καινούργια θέση αλλά μια υπάρχουσα. Στην περίπτωση τώρα που δεν είναι ίδιο με αυτό της θέσης έχουμε collision άρα αυξάνουμε τον μετρητή που μετράει της συγκρούσεις. Για να το αποφύγουμε χρησιμοποιούμε τον αλγόριθμο γραμμικής αναζήτησης καλώντας την συνάρτηση def increment_key(key) η οποία θα κοιτάξει την διαθεσιμότητα της επόμενης θέσης μέχρι να βρεθεί διαθέσιμη. Να σημειώσουμε ότι καθώς ο πίνακας δεν θα είναι ποτέ γεμάτος, ο αλγόριθμος της γραμμικής αναζήτησης δεν θα έχει πρόβλημα. Τέλος γίνεται ένας τελευταίος έλεγχος για το εάν ο πίνακας έχει γεμίσει μέχρι την μέση και συνεπώς θέλει επέκταση. Αν όντως θέλει επέκταση δεν ξεχνάμε να μηδενίσουμε τον μετρητή που μετράει τις συγκρούσεις καθώς η άσκηση ζητάει της συγκρούσεις που υπάρχουν **ΜΟΝΟ** στον τελικό πίνακα.

Η συνάρτηση def findmostcost() είναι υπεύθυνη στο να βρει την θέση της κάρτας που έχει ξοδέψει το μεγαλύτερο ποσό. Για να το πετύχει αυτό διατρέχει ολόκληρο τον πίνακα συγκρίνοντας κάθε ποσό με το maximum_cost. Αν το ποσό είναι μεγαλύτερο από το maximum_cost

τοτε το maximum_cost αντικαθίσταται από το ποσό που βρέθηκε και στην μεταβλητή position_cost εκχωρείται η θέση στην οποία βρέθηκε.

Ακριβώς την ίδια λογική ακολουθεί η def findmostvisits() για να βρει την θέση της κάρτας με τις περισσότερες επισκέψεις. Πρώτα προσθέτει και τα 6 στοιχεία της λίστας value2 και στην συνέχεια όλα τα αθροίσματα με τον ίδιο τρόπο.

Εκτός της κλάσης μέσω της οποίας φτιάχνεται ο Hash Table πρέπει να δημιουργήσω και τα δεδομένα με τα εκάστοτε κλειδιά.

Αρχικά χρησιμοποιώντας μια επαναληπτική δομή while τοποθετώ σε μία λίστα (theseis) 4 τυχαίους αριθμούς από 0 ως 15 που απεικονίζουν τις θέσεις του αλφαριθμητικού card=«1234567890123456» που θα τοποθετηθούν τα γράμματα της λίστας l=["A","B","C","D"]. Αυτό γίνεται με την χρήση της βιβλιοθήκης random. Στη συνέχεια δημιουργώ λεξικό που να αντιστοιχίζει τα στοιχεία των δύο λιστών και αρχικοποιώ των πίνακα που είχαν μπει οι τυχαίες θέσεις (theseis=[]). Παράλληλα με ένα for loop 16 επαναλήψεων αντικαθιστώ με βάση το λεξικό τους αντίστοιχους χαρακτήρες σχηματίζοντας ένα αλφαριθμητικό της μορφής «123A56B8901C34D6». Με αυτό τον τρόπο έχω φτιάξει τις κάρτες (key).

Μέσω της εντολής choice της βιβλιοθήκης random διαλέγω μια τυχαία μέρα από την λίστα meres=["ΔΕΥΤΕΡΑ","ΤΡΙΤΗ","ΤΕΤΑΡΤΗ","ΠΕΜΠΤΗ","ΠΑΡΑΣΚΕΥΗ","ΣΑΒΒΑΤΟ"]. Παράλληλα με μια πολλαπλή if αντιστοιχίζω κάθε μέρα σε μια λίστα week (6 ακεραίων πέντε μηδενικών και ενός άσου η θέση του οποίου προσδιορίζει την μέρα ,π.χ. για ΔΕΥΤΕΡΑ θα έχουμε [1,0,0,0,0,0]) που αποτελεί το value2. Εν τέλει ανάλογα με το ποια μέρα προκύπτει αυξάνεται ο αντίστοιχος μετρητής με σκοπό να μετρήσω συνολικά τις μέρες για όλες τις επισκέψεις.

Τέλος μέσω πάλι της random.randint βρίσκω ένα τυχαίο cost μεταξύ 10 και 100 που αποτελεί το value1.

Μέσω της def insert(key,value1,value2) εισάγω στο πίνακα H:
(key = card , value1 = Cost , value2 = week)

Για την απάντηση του ερωτήματος (α) καλω τη συνάρτηση def findmostcost() και τυπώνω την ανάλογη θέση του πίνακα. Ομοίως για το (β) την def findmostvisits().

Για την απάντηση του (γ) δημιουργώ έναν πίνακα total_visits_per_day με 6 υποπίνακες 2 στοιχείων ο καθένας [day,visits]. Το πρώτο δείχνει την μέρα το δεύτερο το σύνολο των επισκέψεων. Μέσω της sort() ταξινομώ τον total_visits_per_day και βρίσκω τη μέρα με τις περισσότερες επισκέψεις.

Τέλος για την απάντηση του (δ) καλώ την def numcollissions() που επιστρέφει το συνολικό αριθμό των συγκρούσεων στον τελικό πίνακα. Τρέχοντας τον κώδικα για τα 3 διαφορετικά load factors (0.4 , 0.5 , 0.6) διαπιστώνουμε ότι όσο αυξάνεται το load factor αυξάνονται τα collisions.