ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΕΦΑΡΜΟΣΜΕΝΩΝ ΜΑΘΗΜΑΤΙΚΩΝ ΚΑΙ ΦΥΣΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΟΜΕΑΣ ΜΑΘΗΜΑΤΙΚΩΝ

Γραπτή άσκηση #3

Μάθημα: Δομές Δεδομένων

Διδάσκων: Αντώνιος Συμβώνης

Φοιτήτρια: Ελένη Στυλιανού, ge21708

Email: elenistylianou03@live.com

Άσκηση 1

Ο συγκεκριμένο αλγόριθμος αποτελεί ένα παράδειγμα αναδρομικού αλγορίθμου που είναι γνωστός ως αλγόριθμος μείωσης δυαδικού δένδρου. Σκοπός του αλγορίθμου είναι να προσθέσει τα στοιχεία του διανύσματος διαιρώντας επαναληπτικά το διάνυσμα στο μισό και προσθέτοντας τα διπλανά στοιχεία. Λαμβάνοντας υπόψη τον αριθμό των αναδρομικών βημάτων που απαιτούνται μπορούμε να αναλύσουμε την χρονική πολυπλοκότητα του αλγορίθμου. Το μέγεθος του διανύσματος διαιρείται στο μισό, σε κάθε βήμα, μέχρι να φτάσει το μέγεθος 1. Ο αριθμός των αναδρομικών βημάτων που απαιτούνται είναι $\log_2(n)$, εφόσον το αρχικό μέγεθος του διανύσματος είναι n και το n είναι κάποια δύναμη του 2. Σε κάθε βήμα πραγματοποιούνται n/2 προσθέσεις για να δημιουργηθεί το νέο διάνυσμα B. Επομένως, ο συνολικός αριθμός των προσθέσεων που πραγματοποιεί ο αλγόριθμος μπορεί να υπολογιστεί αθροίζοντας τον αριθμό των προσθέσεων σε κάθε βήμα. Συμβολίζουμε με A(n) τον συνολικό αριθμό των προσθέσεων, όπου n το αρχικό μέγεθος του διανύσματος. Ο αριθμός των προσθέσεων σε κάθε βήμα είναι n/2 και $n=2^m$ για κάποιο m. Συνεπώς, $A(n) = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 4 + 2 + 1 = 2^0 + 2^1 + 2^2 + \cdots + 2^{m-3} + 2^{m-2} + 2^{m-1} = \frac{2^m - 1}{2-1} = 2^m - 1 = n - 1.$

- ⇨ Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(\log_2(n))$ και ο ακριβής αριθμός των προσθέσεων που πραγματοποιούνται είναι n-1.

Άσκηση 2

Για την υλοποίηση του ΑΤΔ θα χρησιμοποιήσουμε ένα διάνυσμα για να αποθηκεύει τα n στοιχεία. Η κόκκινη στοίβα θα αυξάνεται από την αρχή του διανύσματος και η πράσινη από το τέλος του. Με αυτό τον τρόπο όταν προσθέτουμε ένα στοιχείο στην κόκκινη στοίβα πηγαίνει προς την αρχή του διανύσματος και όταν προσθέτουμε ένα στοιχείο στην πράσινη πηγαίνει στο τέλος του διανύσματος.

Ψευδοκώδικας για την υλοποίηση του αλγορίθμου:

```
array: array of elements
capacity: maximum capacity of the array
redTop: index of the top element in the red stack
greenTop: index of the top element in the green stack

procedure initialize(capacity)
   array = new array of elements with size capacity
   this.capacity = capacity
   redTop = -1
   greenTop = capacity

procedure redPush(value)
   if redTop + 1 < greenTop then
     redTop = redTop + 1
     array[redTop] = value
   else
     throw StackOverflowError
```

```
procedure greenPush(value)
    if greenTop - 1 > redTop then
      greenTop = greenTop - 1
      array[greenTop] = value
   else
     throw StackOverflowError

function redPop
    if redTop >= 0 then
      value = array[redTop]
      redTop = redTop - 1
      return value
   else
     throw EmptyStackException

function greenPop
    if greenTop < capacity then
      value = array[greenTop]
      greenTop = greenTop + 1
      return value
   else
     throw EmptyStackException

function redTop
   if redTop >= 0 then
     return array[redTop]
  else
    throw EmptyStackException

function greenTop
  if greenTop < capacity then
    return array[greenTop]
  else
    throw EmptyStackException

function isRedEmpty
   return redTop == -1

function isGreenEmpty
   return greenTop == capacity

function isFull
   return redTop + 1 >= greenTop
```

Οι λειτουργίες redPush και greenPush ελέγχουν αν υπάρχει αρκετός χώρος στις αντίστοιχες στοίβες πριν να προσθέσουν ένα στοιχείο. Αν μια στοίβα είναι γεμάτη σημαίνει πως δεν μπορούμε να προσθέσουμε άλλα στοιχεία στο διάνυσμα.

Οι λειτουργίες redPop και greenPop αφαιρούν και επιστρέφουν τα επάνω στοιχεία από τις αντίστοιχες στοίβες. Όταν μία στοίβα είναι άδεια σημαίνει ότι δεν υπάρχουν στοιχεία να αφαιρεθούν.
Οι λειτουργίες isRedEmpty και isGreenEmpty ελέγχουν εάν η κόκκινη ή η πράσινη στοίβα είναι άδεια αντίστοιχα.
Η λειτουργία isFull ελέγχει εάν και οι δύο στοίβες είναι γεμάτες.

Άσκηση 3

Για τον υπολογισμό του path-length ενός δένδρου T σε γραμμικό χρόνο μπορούμε να εκτελέσουμε μια διάσχιση πρώτα κατά βάθος(DFT) του δέντρου και να κρατάμε το βάθος κάθε κόμβου.
Αλγόριθμος:
Αρχικά, εκτελούμε μια διάσχιση με τη μέθοδο Depth-First Search(DFS) του δένδρου ξεκινώντας από έναν αυθαίρετο κόμβο ως ρίζα. Κατά τη DFT υπολογίζουμε τη συνεισφορά της στο μήκος της διαδρομής πολλαπλασιάζοντας το μέγεθος του υποδένδρου στη μία πλευρά της ακμής με το μέγεθος του υποδένδρου στην άλλη πλευρά της ακμής και προσθέτουμε τη συνεισφορά στο συνολικό μήκος της διαδρομής. Έπειτα συνεχίζουμε τη DFT μέχρι να επισκεφθούμε όλους τους κόμβους του δένδρου. Τέλος, υπολογίζουμε το συνολικό μήκος της διαδρομής στο τέλος της διάσχισης.
Ψευδοκώδικας:

```
function CalculatePathLength(tree)
    totalPathLenght=0
    function DFS(node)
        size=1
        for each child in node.children do
            childSize=DFS(child)
            totalPathLength+=childSize*(tree.size-childSize)
            size+=childSize
        return size
    DFS(tree.root)
    Return totalPathLength
```

Για την λειτουργία του παραπάνω αλγορίθμου δίνουμε το αντικείμενο tree (δένδρο) ως είσοδο. Η tree.root αναπαριστά τον κόμβο στη ρίζα του δένδρου και η tree.size τον συνολικό αριθμό των κόμβων στο δένδρο
Η συνάρτηση CalculatePathLength αρχικοποιεί τη μεταβλητή totalPathLenght σε 0 και ορίζει μια εσωτερική συνάρτηση DFS για την εκτέλεση της διάσχισης πρώτα κατά βάθος. Η DFS παίρνει ως παράμετρο ένα κόμβο και υπολογίζει αναδρομικά το μέγεθος του υπόδενδρου με ρίζα σε αυτόν τον κόμβο. Διατρέχει κάθε παιδί του τρέχοντος κόμβου και καλεί αναδρομικά την συνάρτηση DFS για κάθε παιδί. Ακολούθως, υπολογίζει τη συνεισφορά της ακμής που συνδέει τον τρέχοντα κόμβο με το παιδί του στο μήκος της διαδρομής πολλαπλασιάζοντας το μέγεθος του υπόδενδρου του παιδιού με αυτό του συμπληρωματικού υπόδενδρου, που είναι το συνολικό μείον αυτό του παιδιού. Επίσης, ενημερώνει την μεταβλητή size προσθέτοντας το μέγεθος του υπόδενδρου του παιδιού.
Με την ολοκλήρωση της διάσχισης DFS επιστρέφεται από τη συνάρτηση CalculatePathLength η μεταβλητή totalPathLength που περιέχει το υπολογισμένο path-length του δένδρου.

Η συνολική πολυπλοκότητα είναι O(n), όπου n ο αριθμός κόμβων στο δένδρο, εφόσον κάθε κόμβος επισκέπτεται μόνο μία φορά κατά τη διάσχιση και οι υπολογισμοί για κάθε ακμή μπορούν να γίνουν σε σταθερό χρόνο.

Άσκηση 4

Για την υλοποίηση του ζητούμενου θα δημιουργήσουμε ένα αλγόριθμο που θα αρχίζει με τον υπολογισμό των βαθών των κόμβων p και q από τη ρίζα χρησιμοποιώντας μια συνάρτηση έστω CalculateDepth. Ακολούθως, θα προσαρμόζει την θέση των p και q ώστε να έχουν το ίδιο βάθος, μετακινώντας τον πιο βαθύ κόμβο προς τη ρίζα μέχρι να έχουν και οι δύο το ίδιο βάθος. Έπειτα, ο αλγόριθμος θα μετακινεί και τους δύο κόμβους προς τη ρίζα με ένα βήμα τη φορά έως ότου συναντηθούν σε έναν κόμβο. Εφόσον αυτός ο κόμβος θα είναι ο βαθύτερος κόμβος, όπου μπορούν να συναντηθούν οι δύο κόμβοι, τότε είναι και ο κοινός πρόγονος LCA των απογόνων p και q.

Ψευδοκώδικας:
```
procedure CalculateDepth(root, node)
        if root = NULL or root = node then
                return 0
        for each child in root.children do
                if child = node then
                        return 1 + CalculateDepth(child, node)
        return 1 + CalculateDepth(root.parent, node)
procedure ComputeLCA(root, p, q)
        depthP←CalculateDepth(root, p)
        depthQ←CalculateDepth(root, q)
        while depthP > depthQ do
                p←p.parent
                depthP CalculateDepth(root, p)
        while depthQ > depthP do
                q←q.parent
                depthQ←CalculateDepth(root, q)
        while p≠q do
                p←p.parent
                q←q.parent
        return p
```

Η χρονική πολυπλοκότητα του αλγορίθμου εξαρτάται από τη συνάρτηση CalculateDepth και το βάθος του δένδρου. Η εκτέλεση της προεπεξεργασίας κοστίζει O(n). Στη συνέχεια, ο αλγόριθμος εκτελείται με πολυπλοκότητα O(h), όπου h το ύψος του δένδρου. Αυτή η πολυπλοκότητα μπορεί να φτάσει έως O(n) στην περίπτωση ενός σχεδόν γραμμικού δένδρου. Επομένως, η συνολική πολυπλοκότητα του αλγορίθμου είναι O(n+ h), αλλά σε περιπτώσεις με ισορροπημένα δένδρα μπορεί να προσεγγιστεί ως O(n).

Άσκηση 5

Έστω δύο δυαδικά δένδρα T1 και T2, τα στοιχεία των οποίων ικανοποιούν την ιδιότητα διάταξης του σωρού (αλλά όχι απαραίτητα και την δομική ιδιότητα του πλήρους προς τα

αριστερά στοιχισμένου διαδικού δένδρου). Θα κατασκευάσουμε αλγόριθμο που ενώνει τα δύο δένδρα T1 και T2 σε ένα δυαδικό δένδρο T οι κόμβοι του οποίο είναι η ένωση των κόμβων των T1 και T2 και που ικανοποιεί την ιδιότητα διάταξης του σωρού. Το δέντρο T δεν θα ικανοποιεί απαραίτητα τη δομική ιδιότητα του πλήρους προς τα αριστερά στοιχισμένου διαδικού δέντρου. Ο αλγόριθμος θα λειτουργεί ως εξής:
Αρχικά, θα εξετάζει ποιο από τα 2 δέντρα έχει τη μικρότερη ρίζα. Χωρίς βλάβη της γενικότητας υποθέτουμε ότι το δέντρο T1 θα έχει τη μικρότερη ρίζα. Στη συνέχεια, ξεκινώντας από τη ρίζα του T2, θα επιλέγουμε συνεχώς το μεγαλύτερο παιδί του κάθε κόμβου μέχρι να φτάσουμε σε ένα φύλλο, έστω το p (προφανώς βάθος(p)≤h2). Η μέχρι στιγμής πολυπλοκότητα είναι O(h2), εφόσον διασχίσαμε ένα μονοπάτι του T2 από τη ρίζα μέχρι κάποιο φύλλο. Έπειτα, ο αλγόριθμος θα αφαιρεί από το T2 το φύλλο p και θα θέτει τα T1 και T2 ως υποδέντρα του p, δηλαδή οι ρίζες των T1 και T2 θα γίνουν παιδιά του p. Πλέον θα έχουμε ένα νέο δέντρο T με ρίζα p, του οποίου το αριστερό υποδέντρο θα είναι το T1 και το δεξί υποδέντρο θα είναι το T2. Τα T1 και T2 εξακολουθούν να ικανοποιούν την ιδιότητα διάταξης σωρού. Ο μόνος "προβληματικός" κόμβος είναι η ρίζα, γι' αυτό θα επαναφέρουμε σε ισχύ την ιδιότητα διάταξης σωρού (downheap στη ρίζα). Εξαιτίας του γεγονότος ότι η ρίζα του T1 είναι μικρότερη από τη ρίζα του T2, η πρώτη εναλλαγή θα γίνει ανάμεσα στη ρίζα p και στη ρίζα r1 του υποδέντρου T1. Έπειτα ο κόμβος p θα μετακινηθεί προς τα κάτω μέχρι να είναι μεγαλύτερος από τα παιδιά του ή μέχρι να γίνει φύλλο του υποδέντρου T1. Η διαδικασία downheap θα χρειαστεί O(h1) χρόνο, αφού ο κόμβος p θα διασχίσει κάποιο μονοπάτι από τη ρίζα του T1 μέχρι το πολύ κάποιο φύλλο του. Εποεμένως, η συνολική πολυπλοκότητα του αλγορίθμου είναι O(h1+h2).

Άσκηση 6

Για την κατασκευή του αντεστραμμένου αρχείου L που αντιστοιχεί στο κείμενο D, θα γίνει χρήση του ΑΤΔ Διατεταγμένος Χάρτης, αφού θέλουμε η συλλογή από λέξεις L να είναι ταξινομημένη (αλφαριθμητικά). Για την υλοποίηση του διατεταγμένου χάρτη θα χρησιμοποιήσουμε ένα AVL δέντρο. Οι κόμβοι του δέντρου θα αποτελούνται από το κλειδί και τα δεδομένα. Το κλειδί κάθε κόμβου θα είναι μια λέξη w του κειμένου D (ένας κόμβος θα αντιστοιχεί σε κάθε διακριτή λέξη του κειμένου), ενώ τα δεδομένα του κόμβου θα είναι μια λίστα στην οποία θα αποθηκεύουμε τους δείκτες των θέσεων όπου εμφανίζεται η λέξη w στο κείμενο D. Ο αλγόριθμος θα λειτουργεί ως εξής:
Αρχικά, ο χάρτης θα είναι κενός. Ξεκινώντας από την πρώτη λέξη του κειμένου D, και για κάθε επόμενη λέξη, θα εξετάζουμε αν η λέξη περιέχεται στον χάρτη (get(w)).

- Αν ήδη περιέχεται, τότε στη λίστα με τις εμφανίσεις της λέξης στο κείμενο θα προσθέτουμε τη θέση της τρέχουσας εμφάνισης της λέξης στο κείμενο (για το σκοπό αυτό θα χρησιμοποιήσουμε ένα μετρητή καθώς διαβάζουμε το κείμενο D).
- Αν η λέξη δεν περιέχεται ήδη στο κείμενο (δηλαδή η μέθοδος get(w) επέστρεψε null), προσθέτουμε στο χάρτη μια νέα εγγραφή με κλειδί τη λέξη w και με δεδομένα μία λίστα μεγέθους 1 που περιέχει τη θέση της 1ης εμφάνισης της λέξης στο κείμενο.

Μετά τη διαπέραση όλου του κειμένου D, στο διατεταγμένο χάρτη θα έχουν αποθηκευτεί όλες οι διακριτές λέξεις του κειμένου. Στα δεδομένα (data) της κάθε λέξης μπορούμε να δούμε τις εμφανίσεις της λέξης στο αρχικό κείμενο. Εφόσον η υλοποίηση του διατεταγμένου χάρτη θα γίνει με AVL δέντρο, η αναζήτηση κάθε λέξης στο χάρτη και η εισαγωγή μιας νέας εγγραφής κοστίζουν το πολύ O(logm) χρόνο η καθεμία, όπου m το πλήθος των διατεταγμένων λέξεων στο αρχικό κείμενο. Η προσθήκη της θέσης της λέξης στη λίστα (δεδομένα) της εγγραφής κοστίζει σταθερό χρόνο. Συνεπώς, για κάθε λέξη του κειμένου D απαιτείται O(logm) χρόνος. Αν το κείμενο περιέχει n συνολικά λέξεις, η χρονική πολυπλοκότητα του αλγορίθμου θα είναι O(nlogm).

Άσκηση 7

Στην συγκεκριμένη άσκηση θα τροποποιήσουμε την δομή του δυαδικού δένδρου αναζήτησης που υλοποιεί έναν διατεταγμένο χάρτη ώστε να υλοποιήσουμε την μέθοδο countRange σε χρόνο O(h). Για να το πετύχουμε αυτό θα προσθέσουμε ένα νέο πεδίο με όνομα leftCount σε κάθε κόμβο του δέντρου. Στο πεδίο αυτό, για κάθε κόμβο του δέντρου θα αποθηκεύουμε το μέγεθος του αριστερού του υποδέντρου (δηλαδή το πλήθος των απογόνων του κόμβου με μικρότερο κλειδί).
Ισχυρισμός: Με την πληροφορία αυτή, μπορούμε σε χρόνο O(h) να βρούμε πόσα κλειδιά του δέντρου είναι μικρότερα του k.
Απόδειξη:
Έστω x το πλήθος αυτών των κλειδιών. Για να υπολογίσουμε το x, εργαζόμαστε ως εξής: Αρχικά, βρίσκουμε τον κόμβο e με το μέγιστο κλειδί που είναι μικρότερο ή ίσο του k (floorEntry(k)). Το πλήθος των απογόνων του e με κλειδιά μικρότερα του k είναι η τιμή του πεδίου leftCount του e, οπότε αρχικά x=leftCount(k). Αν key(e)<k, αυξάνουμε το x κατά 1. Ακολούθως, για να βρούμε το πλήθος των κόμβων που δεν είναι απόγονοι του e αλλά έχουν κλειδί μικρότερο του k, θα κινηθούμε από τον e μέχρι τη ρίζα ως εξής:

- Αν ο τρέχων κόμβος είναι δεξί παιδί του γονέα του, τότε όλοι οι αριστεροί απόγονοι του γονέα και ο γονέας έχουν κλειδί μικρότερο του k, οπότε μεταβαίνουμε στον γονέα και αυξάνουμε το x κατά την τιμή leftCount+1 του γονέα.
- Αν ο κόμβος είναι αριστερό παιδί του γονέα του, μεταβαίνουμε στο γονέα χωρίς να αυξήσουμε το x, αφού ο γονέας δεν έχει άλλους απογόνους με μικρότερο κλειδί, ούτε ο ίδιος έχει μικρότερο κλειδί.

Όταν φτάσουμε στη ρίζα, η τιμή του x θα αντιστοιχεί στο πλήθος των κλειδιών του δέντρου που είναι μικρότερα του x.
Η αναζήτηση του κόμβου e στο δέντρο απαιτεί O(h) χρόνο και ο υπολογισμός του x απαιτεί επίσης O(h) χρόνο, επομένως μπορούμε να βρούμε το πλήθος των κλειδιών που είναι μικρότερα του k σε O(h) χρόνο.
Απομένει να δούμε πώς θα λαμβάνει τιμές το πεδίο leftCount για κάθε κόμβο χωρίς να μεταβάλλεται η πολυπλοκότητα των μεθόδων του διατεταγμένου χάρτη.

Ενημέρωση του πεδίου leftCount κατά την εισαγωγή στοιχείου στο δέντρο:
Με την προσθήκη της πρώτης εγγραφής, το δέντρο θα αποτελείται από έναν μόνο κόμβο (τη ρίζα). Οπότε το πεδίο leftCount της ρίζας θα παίρνει την τιμή 0.
Κάθε νέα εγγραφή σε ένα διαδικό δέντρο αναζήτητης προστίθεται ως ένα φύλλο στο δέντρο (αρχίζοντας από τη ρίζα και κατεβαίνοντας προς τα κάτω, κινούμενη δεξιά ή αριστερά ανάλογα με το αν το κλειδί της είναι μικρότερο ή μεγαλύτερο από το τρέχον κλειδί).
Επομένως, για κάθε νέα εγγραφή με κλειδί k που εισάγεται στο δέντρο, ορίζουμε την τιμή του πεδίου leftCount να είναι ίση με 0 και ακολουθούμε την εξής διαδικασία:
Αρχικά, ελέγχουμε αν υπάρχει άλλη εγγραφή με το ίδιο κλειδί. Αν ναι, τότε αποθηκεύουμε τη νέα εγγραφή στον ίδιο κόμβο με τις εγγραφές που έχουν ως κλειδί το k. Αν όχι, τότε

αρχίζοντας από τη ρίζα, και για κάθε κόμβο e με κλειδί m στον οποίο μεταβαίνουμε (μέχρι η νέα εγγραφή να γίνει φύλλο), εξετάζουμε:

- Αν k<m, μεταβαίνουμε στο αριστερό παιδί του e και το πεδίο leftCount του e αυξάνεται κατά 1(αφού το αριστερό υπόδενδρο του e μεγάλωσε κατά 1 κόμβο)
- Αν k>m, μεταβαίνουμε στο δεξί παιδί του e χωρίς να μεταβάλλεται το πεδίο leftCount.

Με την πιο πάνω διαδικασία επιτύχαμε την ενημέρωση του πεδίου leftCount όλων των εγγραφών που χρειάζονταν αλλαγή. Αυτό επιτεύχθηκε κατά την κάθοδο του στοιχείου από τη ρίζα προς τα κάτω, οπότε η εισαγωγή στοιχείου στο δέντρο και η ταυτόχρονη ενημέρωση των πεδίων leftCount δεν απαιτεί επιπλέον χρόνο. Η χρονική πολυπλοκότητα της εισαγωγής παραμένει O(h).

Ενημέρωση του πεδίου leftCount κατά την διαγραφή στοιχείου στο δέντρο:
Όμοια με την εισαγωγή, για να διαγράψουμε ένα κλειδί από το δέντρο χρειάζεται να ενημερώσουμε το πεδίο leftCount των προγόνων του.
Αν ο κόμβος με το συγκεκριμένο κλειδί είναι φύλλο, αρχίζουμε από αυτόν και για όσο ο τρέχων κόμβος δεν είναι ρίζα κινούμαστε προς τα πάνω ως εξής:

- Αν ο κόμβος είναι αριστερό παιδί του γονέα του, μειώνουμε την τιμή του πεδίου leftCount του γονέα κατά 1.
- Μεταβαίνουμε στον γονέα.

Με το πέρας αυτής της διαδικασίας έχουν ενημερωθεί τα πεδία leftCount των προγόνων και έτσι μπορούμε να διαγράψουμε την εγγραφή. Με παρόμοιο τρόπο ανανεώουμε το πεδίο leftCount των προγόνων αν ο κόμβος είναι εσωτερικός σε O(h) χρόνο.

Εύρεση του [k1,k2]:
Από τις 2 πιο πάνω παραγράφους (ενημέρωση του πεδίου leftCount κατά την εισαγωγή και διαγραφή στοιχείου από το δέντρο) γίνεται σαφές ότι η ενημέρωση του πεδίου leftCount δεν επιβαρύνει χρονικά τις λειτουργίες του διατεταγμένου χάρτη, άρα τα πεδία leftCount των εγγραφών μπορούν να είναι ενημερωμένα μετά από κάθε δομική μεταβολή του χάρτη. Επίσης, από την απόδειξη του ισχυρισμού, γνωρίζουμε ότι μπορούμε να βρούμε το πλήθος των κλειδιών που είναι μικρότερα από μια τιμή k σε O(h) χρόνο. Έστω find(k) η μέθοδος που υπολογίζει αυτό το πλήθος για το κλειδί k. Για να βρούμε το πλήθος των κλειδιών στο διάστημα [k1,k2], αρκεί να βρούμε τη διαφορά find(k2)-find(k1), η οποία θα υπολογίζει το πλήθος των κλειδιών στο διάστημα [k1,k2). Ακολούθως θα εξετάζουμε αν το κλειδί k2 περιέχεται στο χάρτη. Αν ναι, αυξάνουμε τη διαφορά αυτή κατά 1.