09/05/2021

[ΠΛΗ 201] ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΑΡΧΕΙΩΝ

2Η ΑΣΚΗΣΗ

ΚΑΤΣΙΜΠΑΣ ΠΕΤΡΟΣ

2016030038

Διδάσκων: **Ευριπίδης Πετράκης**
Υπεύθυνοι Εργαστηρίου: **Σπύρος Αργυρόπουλος, Στέφανος Καρασαββίδης**

Δυαδικό Δένδρο Έρευνας

(Η υλοποίηση και η Main του εμπεριέχεται στο bST.BinaryTree.java και bST.BinaryTree_ MAIN.java)

Στο πρώτο μέρος μας ζητήθηκε να δημιουργήσουμε ένα Δυαδικό Δέντρο Έρευνας(Binary Search Tree) με τη χρήση array μεγέθους Nx3 (όπου N = 10^5). Για την υλοποίηση αυτή είναι σημαντικό να κατανοήσουμε πως λειτουργούν οι μέθοδοι του βοηθητικού δέντρου δυναμικής μνήμης που μας δόθηκε. Επίσης, το πρώτο πράγμα για να είναι εφικτή η δημιουργία του Δένδρου-Πίνακα είναι η αρχικοποίηση του δένδρου με την τιμή -1 σε όλα τα πεδία ώστε να δημιουργήσουμε τα *Null* κελιά, με μόνη εξαίρεση την 3^{η} στήλη του πίνακα, η οποία θα περιέχει θετικούς όρους σε αύξουσα σειρά με σκοπό να χρησιμοποιείται παράλληλα για την αναπαράσταση των διαθέσιμων θέσεων.(***bst.Init()***)

Data[i][k]	**Info(k=0)**	**left(k=1)**	**right(k=2)**
i=0	-1	-1	0
i=1	-1	-1	1
i=2	-1	-1	2
i=3	-1	-1	3
i=4	-1	-1	4
i=5	-1	-1	5

Η εισαγωγή έγινε με τη συνάρτηση ***bst.callInsertNode(),*** η οποία έγινε με παρόμοια λογική της δοθείσας βοηθητικής συνάρτησης. Η λογική της είναι, αρχικά, να κοιτάει αν βρίσκεται σε κενή θέση πίνακα ώστε να εισαχθεί ο αριθμός στη θέση 0 αλλά και οι απαραίτητες αλλαγές στη θέση 2 με τιμή -1, ώστε να μην θεωρείται πλέον ¨κενό¨ κελί, και στο προηγούμενο κόμβο ώστε να δείχνει στον τωρινό μας με τη βοήθεια της μεταβλητής ***LeftOrRight(1-left, 2-right)***. Σε περίπτωση όπου είμαστε σε μη διαθέσιμη θέση πίνακα η συνάρτηση αποφασίζει με συγκρίσεις αν πρέπει να πάμε δεξιά ή αριστερά και να ξανακαλέσει τον εαυτό της.

Την ίδια λογική διάσχισης δένδρου ακολουθεί και η συνάρτηση ***bst.callSearchNode().*** Η διαφορά, ωστόσο, είναι ότι δεν χρειάζεται να κρατάμε περαιτέρω πληροφορίες αφού μας αρκεί να βρούμε τη θέση του πίνακα που εμπεριέχει την αναζητούμενη τιμή και να γυρίσουμε τον αριθμό συγκρίσεων.

Για την αναζήτηση εύρους τιμών υλοποιήσαμε, πάλι, συνάρτηση παρόμοιας λογικής αντί για root.left()/root.right χρησιμοποιήσαμε ***data[nodeIndex][1]*** για μετάβαση αριστερά και ***data[nodeIndex][2]*** για μετάβαση δεξιά.

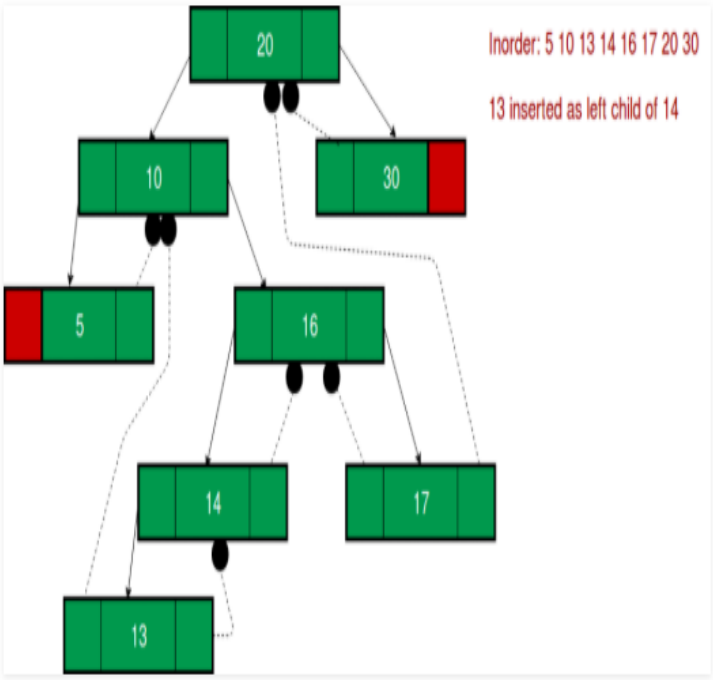
Νηματοειδές Δυαδικό Δένδρο Έρευνας

(Η υλοποίηση και η Main του εμπεριέχεται στο bST.ThreadBST.java και bST. ThreadBST _ MAIN.java)

Για το 2ο μέρος της άσκησης ζητήθηκε να υλοποιήσουμε ένα δυαδικό δένδρο έρευνας με νήματα. Στην ουσία, η υλοποίηση ήταν παρόμοια με το δένδρο του προηγούμενου μέρους με διαφορά, όμως, την αλλαγή του πίνακα Nx5. Ο λόγος που προστέθηκαν δύο νέες στήλες είναι ότι σε αυτό το δένδρο έχουμε δύο νέες μεταβλητές την ***LeftThread*** και την ***RightThread***(boolean variables, βέβαια εμείς τις αναθέσαμε σαν 1 για true και -1 για false). Η ύπαρξη των μεταβλητών αυτών είναι αναγκαία καθώς σε αυτό το δένδρο, κάθε κόμβος που δεν έχει υποκλάδια, αντί να έχει τις τιμές -1 στη θέση του ***left/right*** , οφείλει να δείχνει στην επόμενη τιμή του δένδρου είτε στην αμέσως μικρότερη είτε στην αμέσως μεγαλύτερη. Σε περίπτωση που έχουμε δηλαδή αυτά τα νήματα οι μεταβλητές μας το φανερώνουν έχοντας ανεβασμένη την τιμή σε 1. (***tBST.Init()***)

Για να καταφέρουμε να υλοποιήσουμε σωστά το δένδρο, ήταν αρκετά σημαντικό να διευκρινίσουμε πως θα κρατάμε τις απαραίτητες πληροφορίες. Η παρατήρηση κλειδί για αυτό ήταν ότι σε κάθε εισαγωγή είτε διασχίσουμε αριστερά είτε διασχίσουμε δεξιά, ο αμέσως μικρότερο/μεγαλύτερος θα είναι ο κόμβος που μόλις προσπελάσαμε και έτσι αποθηκεύουμε την θέση του στον πίνακα στον ***LeftThread/ RightThread(εδώ παίζουν τον ρόλο του Integer).*** Έτσι, στην περίπτωση του κενού κελιού έχουμε όλες τις απαραίτητες πληροφορίες. (***tBST.callInsertNode ()***)

Name	Value
[1]	(id=25)
[0]	30
[1]	0
[2]	-1
[3]	1
[4]	-1
[2]	(id=26)
[0]	10
[1]	3
[2]	4
[3]	-1
[4]	-1
[3]	(id=27)
[0]	5
[1]	-1
[2]	2
[3]	-1
[4]	1
[4]	(id=28)
[0]	16
[1]	5
[2]	6
[3]	-1
[4]	-1
[5]	(id=29)
[0]	14
[1]	7
[2]	4
[3]	-1
[4]	1
[6]	(id=30)
[0]	17
[1]	4
[2]	0
[3]	1
[4]	1
[7]	(id=31)
[0]	13
[1]	2
[2]	5
[3]	1



Η υλοποίηση της ***tBST.searchNode()*** για την αναζήτηση τυχαίας μεταβλητής αποτελεί την ίδια συνάρτηση με το προηγούμενο δένδρο αφού η διάσχιση γίνεται με τον ίδιο τρόπο.

*Στα δεξιά φαίνεται ο κώδικας της **tBST.printRange()**. Αναζητάει κόμβους με τιμές ανάμεσα από ένα εύρος τιμών **low , high(οπου high = low + K).** Η λειτουργία είναι παρόμοια με την βοηθητική του εργαστηρίου αλλά έχουν προστεθεί κάποιες συγκρίσεις ώστε να αποφεύγουμε κάποια loop-conditions και να λειτουργεί σωστά. Στην προσπάθεια μας να πάρουμε όλο το εύρος του πίνακα παρατηρήσαμε ότι λειτουργεί σαν την συνάρτηση **InOrder()** και μας εκτυπώνει τους κόμβους σε αύξουσα σειρά.*

```
176    public int printRange(int nodeIndex, int low, int high, int counter) {
177        if (nodeIndex == -1) {
178            counter++;
179            return counter;
180        }
181        counter++;
182        if (high < data[nodeIndex][0]) {
183            counter++;
184            if (data[nodeIndex][3] == -1) {
185                counter = printRange(data[nodeIndex][1], low, high, counter);
186            } else {
187                return counter;
188            }
189         counter++;
190        } else if (low > data[nodeIndex][0]) {
191            counter++;
192            if (data[nodeIndex][4] == -1) {
193                counter = printRange(data[nodeIndex][2], low, high, counter);
194            } else {
195                return counter;
196            }
197        } else {
198
199            if (data[nodeIndex][3] == -1) {
200                counter++;
201                counter = printRange(data[nodeIndex][1], low, high, counter);
202            }
203
204            //System.out.print(" " + data[nodeIndex][0]);
205
206            //// To prevent second print of a threaded Node
207            if (data[nodeIndex][4] != -1) {
208                return counter;
209            }
210            counter = printRange(data[nodeIndex][2], low, high, counter);
211
212
213        }
214        return counter;
215    }
216
```

```
counter = tBST.callPrintRange(counter, 10000); //K=1000;
    int rkey = rand.nextInt(19) + 1;
    int low = helping_buf4Keys.get(rkey);
    int high = low + k;
    counter = printRange(nodeIndex, low, high, counter);
```

ΕΚΤΥΠΩΣΗ ΤΙΜΩΝ:

```
44581 46431 48434 49075
```

Ο πίνακας περιείχε τις τιμές:

```
1080 5752 13384 19164 31148 44581 46431 48434 49075 59564 59782 62131 62139 82527 83021 92708 97644 97873 98256 98477 99222
```

Απόδοση και Τεμκηρίωση Αποτελεσμάτων

Για να εξετάσουμε την απόδοση των δένδρων που υλοποιήσαμε κάναμε εισαγωγή 10^5 κλειδιών (τα οποία δεν επαναλαμβάνονται) και μετρήσαμε τον μέσο όρο συγκρίσεων σε εισαγωγή κόμβου, αναζήτηση κόμβου και αναζήτηση εύρους τιμών. Τα αποτελέσματα του *BinaryTree*, *Threaded_BST* και *SortedArray(runnable)* φαίνονται στο παρακάτω πίνακα:

Μέθοδος	Μέσος αριθμός συγκρίσεων / εισαγωγή	Μέσος αριθμός συγκρίσεων / τυχαία αναζήτηση	Μέσος αριθμός συγκρίσεων / αναζήτηση εύρους (K=100)	Μέσος αριθμός συγκρίσεων / αναζήτηση εύρους (K=1000)
ΔΔΕ Α	524	54	98	455
Νηματοειδές ΔΔΕ Β	232	45	76	254
Ταξινομημένο πεδίο		29	26	53

Βλέπουμε ότι γενικά καλύτερο στην απόδοση αποτελεί ο ταξινομημένος πίνακας σε όλες τις αναζητήσεις. Είναι αρκετά λογικό να βγαίνει αυτό το αποτέλεσμα αφού όλοι οι κόμβοι είναι σε αύξουσα σειρά ώστε μια δυαδική διάσχιση να εξάγει γρήγορα τα αποτελέσματα και χωρίς πολλές συγκρίσεις. Το ελάττωμα, όμως, του ταξινομημένου πίνακα είναι ότι καθυστερεί πάρα πολύ συγκριτικά με τα άλλα δύο δένδρα στις μεθόδους του *Initialization/InsertionOfNewNode*(κάτι όμως που εμείς δεν συγκρίναμε). Όσο αναφορά, λοιπόν, τα άλλα δύο δένδρα παρατηρούμε μία καλύτερη απόδοση στο νηματοειδές. Αυτό συμβαίνει καθώς το νηματοειδές δένδρο έρευνας εκμεταλλεύεται τα “null” στοιχεία των Left/Right στους τελευταίους κόμβους του δένδρου, δηλαδή αντί για -1 υπάρχει μία τιμή που υποδεικνύει στην αμέσως μικρότερη/μεγαλύτερη τιμή και γλυτώνει έτσι τις παραπανίσιες συγκρίσεις του κανονικού δυαδικού δένδρου.