

---

# ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

---

## ΕΡΓΑΣΙΑ 2

ΣΟΦΡΑΣ ΑΝΤΩΝΗΣ - Π10119

ΔΗΜΑΚΟΓΙΑΝΝΗ ΜΑΡΙΑ - Π17025

# ΑΝΑΛΥΣΗ HEADER FILES - ΚΛΑΣΕΩΝ

---

## Αρχείο TreeNode.h:

Το αρχείο `TreeNode.h` περιλαμβάνει την templated κλάση `TreeNode` η οποία αναπαριστά έναν κόμβο του ΔΔΑ. Έχει τα **private** πεδία:

- **data:** Τύπου “T”, όπου T είναι ο τύπος δεδομένων που θα περιέχει το ΔΔΑ. Σε αυτό το πεδίο βρίσκονται τα δεδομένα του κάθε κόμβου.
- **leftChild/rightChild:** Τύπου “**TreeNode\***”, δηλαδή πρόκειται για δείκτες στο αριστερό (αντ. Δεξί) παιδί του κάθε `TreeNode`.

Οι **public** μέθοδοι είναι οι:

- **TreeNode(T dataIn):** Είναι ο constructor των αντικειμένων, δέχεται σαν όρισμα τα δεδομένα του κόμβου που θα δημιουργηθεί και αρχικοποιεί τους δείκτες να δείχνουν στο NULL.
- **getData():** Μία βοηθητική μέθοδος σε περίπτωση που θέλουμε να έχουμε πρόσβαση στο πεδίο `data`, δεν θα μας χρειαστεί (παραμόνο για δοκιμές από την `main`) αφού η κλάση `BSTree` είναι friend της `TreeNode`.

## Αρχείο BSTree.h:

Στο αρχείο `BSTree.h` υλοποιούμε την κλάση του συγκεκριμένου ΔΔΑ και όλες τις μεθόδους του. Είναι friend της κλάσης `TreeNode` οπότε έχει πρόσβαση στο private πεδίο `data`. Και αυτή η κλάση είναι templated τύπου T.

**Private** πεδία:

- **root:** Τύπου **TreeNode<T>\***, πρόκειται για έναν δείκτη σε κόμβο του δένδρου και πιο συγκεκριμένα θα μας δείχνει πάντα στην ρίζα του συγκεκριμένου δένδρου.
- **n:** Τύπου **int**, είναι το πλήθος των στοιχείων που περιέχει το ΔΔΑ.
- **d:** Τύπου **int**, είναι το πλήθος των διαγραφών που έχουν γίνει στο δένδρο.
- **c/b:** Τύπου **double**, είναι τα ορίσματα c και b που δείνει ο χρήστης κατά την εκτέλεση του προγράμματος.

## Public μέθοδοι:

- **BSTree(double, double):** Ο constructor της κλάσης. Αρχικοποιεί το πλήθος στοιχείων  $n$  και το πλήθος διαγραφών  $d$  σε 0. Τα  $c$  και  $b$  με τις τιμές που παίρνει σαν ορίσματα και τέλος τον δείκτη της ρίζας να δείχνει στο NULL (αφού είναι κενό).
- **bool isEmpty():** Επιστρέφει **TRUE** αν ο δείκτης της ρίζας  $root$  δείχνει στο NULL δηλαδή το δένδρο είναι άδειο. Αλλιώς επιστρέφει **FALSE**.
- **T getNode(const TreeNode<T>\*):** Επιστρέφει το περιεχόμενο  $data$  του κόμβου που δείχνει ο δείκτης που παίρνει σαν όρισμα.
- **int insert(const T&):** Εισάγει το στοιχείο τύπου  $T$  που δέχεται σαν όρισμα στο ΔΔΑ. Αρχικοποιούμε 2 δείκτες έναν στην ρίζα που θα βρεί την θέση που πρέπει να εισαχθεί το στοιχείο και έναν που θα δείχνει στον πατέρα του. Παράλληλα δημιουργούμε μία απλά συνδεδεμένη λίστα που θα δέχεται δείκτες σε κόμβους του δένδρου (η υλοποίηση της έχει γίνει στην προηγούμενη εργασία) στην οποία θα αποθηκεύσουμε το μονοπάτι των κόμβων μέχρι το σημείο που θα εισαχθεί το νέο στοιχείο.

Εφόσον βρεθεί θέση για το στοιχείο (δηλαδή δεν υπάρχει ήδη στο δένδρο) αυξάνουμε τον μετρητή  $n$  κατά ένα και δεσμεύουμε χώρο στην μνήμη για έναν καινούργιο κόμβο. Έπειτα αν το δένδρο δεν είναι άδειο το τοποθετούμε σαν αριστερό/δεξί παιδί του πατέρα του, αλλιώς το τοποθετούμε σαν ρίζα.

Αφού γίνει η εισαγωγή γίνεται ο έλεγχος και οι απαραίτητες ενέργειες για να διατηρήσουμε την ισορροπία του δένδρου. Ελέγχουμε δηλαδή την μεταβλητή  $depth$  η οποία μας κρατάει το βάθος του νεοεισερχόμενου κόμβου αν είναι μεγαλύτερη από  $\lceil \log(n+1+d) \rceil$ .

Εάν ισχύει θα πρέπει να βρούμε τον πρώτο κόμβο στο μονοπάτι που έχουμε σώσει στην λίστα που ικανοποιεί την συνθήκη:

$\text{ύψος κόμβου} > \lceil \log(nv+1) \rceil$  όπου  $nv$  είναι το πλήθος των κόμβων του υποδένδρου με ρίζα τον συγκεκριμένο κόμβο. Όταν τον βρούμε κάνουμε κάποιες εκτυπώσεις που βοήθησαν στην επαλήθευση του προγράμματος και καλούμε την  $treeReconstruction()$  που θα δούμε αργότερα για να ανακατασκευάσει το υποδένδρο. Έπειτα ελευθερώνουμε την μνήμη της λίστας και σαν επιπρόσθετη λειτουργία επιστρέφουμε τον ακέραιο που είναι και το βάθος που εισάγαμε το νέο στοιχείο (πριν την ανακατασκευή του δένδρου) αλλιώς -1 αν η εισαγωγή απέτυχε.

- **Void deleteNode(const T&) και TreeNode<T>\* deleteNode(const T&, TreeNode<T>\*, bool):** Καλούμε από την  $main$  την  $deleteNode$  με μοναδικό όρισμα την τιμή που θέλουμε να διαγράψουμε από το δένδρο η οποία με την σειρά της καλεί αυτή με τα περισσότερα ορίσματα για να ψάξει και να διαγράψει αναδρομικά το στοιχείο. Εφόσον το βρει και ο

κόμβος του έχει 2 παιδιά χρησιμοποιούμε την βοηθητική συνάρτηση `minNode()` που θα δούμε παρακάτω για να βρούμε το ελάχιστο στοιχείο του δεξιού υποδένδρου και να κάνουμε αναγωγή σε διαγραφή φύλλου.

Τέλος ελέγχουμε αν η διαγραφή πέτυχε ή όχι με την μεταβλητή `flag`. Και έπειτα ελέγχουμε αν ο μετρητής διαγραφών `d` είναι μεγαλύτερος ή ίσος με  $(2^{b/c}-1)(n+1)$ . Αν ναι προχωράμε σε ολική ανακατασκευή όλου του δένδρου.

- **Bool searchTree(const T& key, T& e):** Ψάχνει το δένδρο για το στοιχείο με τιμή `key`, εφόσον το βρεί το τοποθετεί στην μεταβλητή `e` και επιστρέφει `True`, αλλιώς επιστρέφει `false`.
- **printTree(char order) και printTree(char order, TreeNode<T>\* node):** Επισκέπτεται τους κόμβους του δένδρου αναδρομικά και εκτυπώνει το περιεχόμενό τους. Το όρισμα `order` καθορίζει τον τύπο της διάσχισης, `p` για `preorder`, `i` για `inorder` και `s` για `postorder`.
- **Void printTreeStats():** Η μέθοδος αυτή εκτυπώνει χρήσιμες πληροφορίες του δένδρου όπως το ύψος, τον αριθμό στοιχείων και διαγραφών καθώς και το μέγιστο ύψος που μπορεί να φτάσει σύμφωνα με τον τύπο:  $c \log(n+1)+b+1$
- **TreeNode<T>\* getRoot():** Επιστρέφει έναν δείκτη στην ρίζα του δένδρου. Κανονικά ο χρήστης δεν θα έπρεπε να είχε πρόσβαση σε αυτή την συνάρτηση αλλά βοήθησε στην διαδικασία της αποσφαλμάτωσης και ελέγχου.
- **void treeReconstruction(TreeNode<T>\* node, TreeNode<T>\* parentNode, char succession):** Αυτή η μέθοδος ανακατασκευάζει ένα δένδρο με ρίζα το `node` σε ένα ισορροπημένο ΔΔΑ. Το όρισμα `parentNode` και το όρισμα `succession` μας βοηθάνε στην σωστή επανασύνδεση του κόμβου (ρίζα του υποδένδρου) `node` με το υπόλοιπο δένδρο εάν πρόκειται για ανακατασκευή υποδένδρου.

Αρχικά δημιουργούμε μία λίστα που θα δέχεται τα στοιχεία του δένδρου. Έπειτα χρησιμοποιούμε την `bstToList()` για να την γεμίσουμε με τα στοιχεία του δένδρου. Εάν το δένδρο που ανακατασκευάζουμε είναι υποδένδρο τότε το ισορροπημένο ΔΔΑ που θα μας γυρίσει η `sortedListToBST()` το συνδέουμε κατάλληλα χρησιμοποιώντας τα ορίσματα `parentNode` και `succession`, αλλιώς αν ανακατασκευάζουμε όλο το δένδρο απλά το θέτουμε σαν ρίζα.

- **TreeNode<T>\* sortedListToBST(List<T> &li, TreeNode<T>\* node) και TreeNode<T>\* sortedListToBSTrecur(List<T> &li, int count):** Η ιδέα προσπαθώντας να επιτύχουμε πολυπλοκότητα  $O(n)$  είναι να προσθέτουμε στο δένδρο στοιχεία όπως μας έρχονται από μία ταξινομημένη λίστα με τον εξής τρόπο. Μετράμε τα στοιχεία της λίστας, έστω ότι είναι `count`. Θα πάρουμε τα πρώτα `count/2` στοιχεία και

θα κατασκευάσουμε το αριστερό υποδένδρο. Έπειτα θα δεσμεύσουμε χώρο για την ρίζα και τέλος θα πάρουμε τα υπόλοιπα  $\text{count} - \text{count}/2 - 1$  στοιχεία για το δεξιό υποδένδρο. Εκτελώντας αναδρομικά αυτή την διαδικασία καταφέρνουμε να κατασκευάσουμε ένα ισορροπημένο ΔΔΑ από μία ταξινομημένη λίστα με πολυπλοκότητα  $O(n)$ .

- **bstToList(const TreeNode<T>\* node, List<T>& li):** Αναδρομικά διασχίζουμε το δένδρο με ρίζα node με ενδοδιάταξη και εισάγουμε κάθε στοιχείο στο τέλος της λίστας. Έτσι η λίστα που προκύπτει είναι ταξινομημένη.
- **Int maxHeight(const TreeNode<T>\* node):** Υπολογίζει το ύψος του δένδρου με ρίζα node αναδρομικά, υπολογίζοντας το ύψος του κάθε αριστερού και δεξιού υποδένδρου και επιστρέφοντας το μεγαλύτερο (+1 για την ρίζα).
- **int countTreeNodes(TreeNode<T>\* node):** Επιστρέφει τον αριθμό των κόμβων ενός δένδρου με ρίζα το node. Αναδρομικά υπολογίζει τον αριθμό των αριστερών και δεξιών υποδένδρων και επιστρέφει το άθροισμα +1 για την ρίζα.

## COMPILATION ΚΑΙ ΕΚΤΕΛΕΣΗ ΤΟΥ MAIN.CPP

---

Στο αρχείο main.cpp υπάρχει μία ενδεικτική main συνάρτηση όπου αρχικά παίρνουμε τα ορίσματα c και b από τον χρήστη, αλλιώς εμφανίζουμε στην οθόνη ένα μήνυμα σφάλματος, και κατασκευάζουμε το δένδρο καλώντας τον constructor.

Έπειτα χρησιμοποιώντας ένα for loop γεμίζουμε το δένδρο με αριθμούς από το 1 μέχρι το 19. Η συνάρτηση insert του δένδρου εκτυπώνει τα χρήσιμα πεδία του δένδρου καθώς και το δένδρο σε προδιάταξη για να ελέγχουμε ότι διατηρεί την μορφή που θέλουμε, **θα πρέπει οι κλήσεις print μέσα στην insert() να γίνουν σχόλια αν θέλουμε να γεμίσουμε το δένδρο με πάρα πολλά στοιχεία.** Έπειτα διαγράφουμε με άλλο ένα for loop τα στοιχεία με τους αριθμούς από το 19 έως το 6. Και η deleteNode() εκτυπώνει το δένδρο σε προδιάταξη καθώς και τα χρήσιμα πεδία του όταν χρειαστεί να ανακατασκευάσει το δένδρο.

# ΠΑΡΑΔΕΙΓΜΑ COMPILEATION ΚΑΙ ΕΚΤΕΛΕΣΗΣ:

```
anthonysof@KuPtop: ~/self-balancing-BST
anthonysof@KuPtop: ~/self-balancing-BST 104x55

~/self-balancing-BST on  master!  18:44:25
$ uname -a                                     <system>
Linux KuPtop 4.15.0-23-generic #25-Ubuntu SMP Wed May 23 18:02:16 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux

~/self-balancing-BST on  master!  18:44:27
$ g++ main.cpp -o main -Wall                  <system>

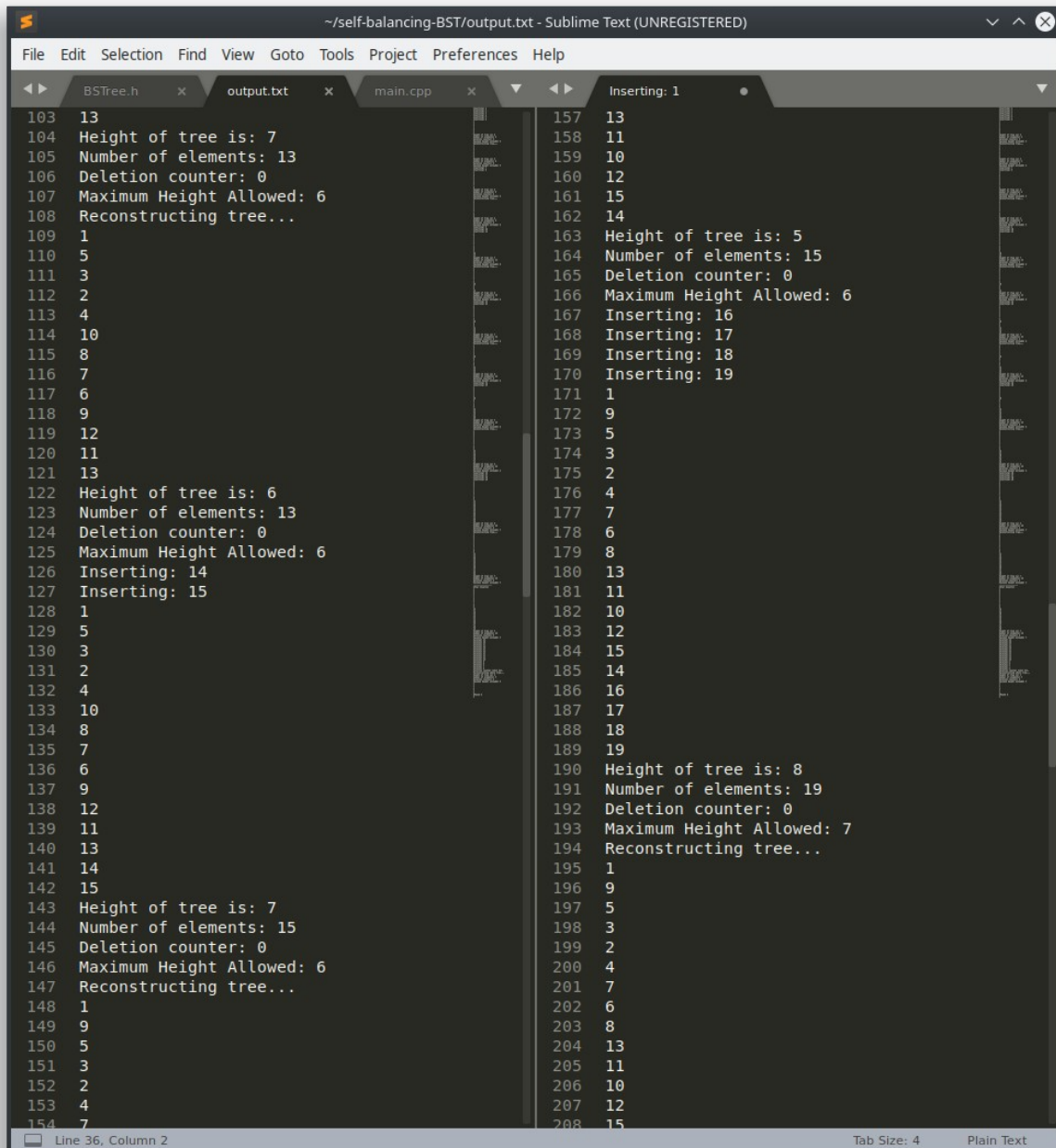
~/self-balancing-BST on  master!  18:44:31
$ ./main 1.2 2 > output.txt                  <system>

~/self-balancing-BST on  master!  18:44:35
$ cat output.txt                              <system>
Inserting: 1
Inserting: 2
Inserting: 3
Inserting: 4
Inserting: 5
Inserting: 6
1
2
```

## ΠΕΡΙΕΧΟΜΕΝΑ OUTPUT.TXT:

```
~/self-balancing-BST/output.txt - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help

BSTree.h x output.txt x main.cpp x Inserting: 1
1 Inserting: 1
2 Inserting: 2
3 Inserting: 3
4 Inserting: 4
5 Inserting: 5
6 Inserting: 6
7 1
8 2
9 3
10 4
11 5
12 6
13 Height of tree is: 6
14 Number of elements: 6
15 Deletion counter: 0
16 Maximum Height Allowed: 5
17 Reconstructing tree...
18 1
19 2
20 5
21 4
22 3
23 6
24 Height of tree is: 5
25 Number of elements: 6
26 Deletion counter: 0
27 Maximum Height Allowed: 5
28 Inserting: 7
29 Inserting: 8
30 1
31 2
32 5
33 4
34 3
35 6
36 7
37 8
38 Height of tree is: 6
39 Number of elements: 8
40 Deletion counter: 0
41 Maximum Height Allowed: 5
42 Reconstructing tree...
43 1
44 5
45 3
46 2
47 4
48 7
49 6
50 8
51 Height of tree is: 4
52 Number of elements: 8
53 Number of elements: 8
54 Deletion counter: 0
55 Maximum Height Allowed: 5
56 Inserting: 9
57 Inserting: 10
58 Inserting: 11
59 1
60 5
61 3
62 4
63 7
64 6
65 8
66 9
67 10
68 11
69 Height of tree is: 7
70 Number of elements: 11
71 Deletion counter: 0
72 Maximum Height Allowed: 6
73 Reconstructing tree...
74 1
75 5
76 3
77 2
78 4
79 7
80 6
81 10
82 9
83 8
84 11
85 Height of tree is: 6
86 Number of elements: 11
87 Deletion counter: 0
88 Maximum Height Allowed: 6
89 Inserting: 12
90 Inserting: 13
91 1
92 5
93 3
94 2
95 4
96 7
97 6
98 10
99 9
100 8
101 11
102 12
103 13
```



```
103 13
104 Height of tree is: 7
105 Number of elements: 13
106 Deletion counter: 0
107 Maximum Height Allowed: 6
108 Reconstructing tree...
109 1
110 5
111 3
112 2
113 4
114 10
115 8
116 7
117 6
118 9
119 12
120 11
121 13
122 Height of tree is: 6
123 Number of elements: 13
124 Deletion counter: 0
125 Maximum Height Allowed: 6
126 Inserting: 14
127 Inserting: 15
128 1
129 5
130 3
131 2
132 4
133 10
134 8
135 7
136 6
137 9
138 12
139 11
140 13
141 14
142 15
143 Height of tree is: 7
144 Number of elements: 15
145 Deletion counter: 0
146 Maximum Height Allowed: 6
147 Reconstructing tree...
148 1
149 9
150 5
151 3
152 2
153 4
154 7
157 13
158 11
159 10
160 12
161 15
162 14
163 Height of tree is: 5
164 Number of elements: 15
165 Deletion counter: 0
166 Maximum Height Allowed: 6
167 Inserting: 16
168 Inserting: 17
169 Inserting: 18
170 Inserting: 19
171 1
172 9
173 5
174 3
175 2
176 4
177 7
178 6
179 8
180 13
181 11
182 10
183 12
184 15
185 14
186 16
187 17
188 18
189 19
190 Height of tree is: 8
191 Number of elements: 19
192 Deletion counter: 0
193 Maximum Height Allowed: 7
194 Reconstructing tree...
195 1
196 9
197 5
198 3
199 2
200 4
201 7
202 6
203 8
204 13
205 11
206 10
207 12
208 15
```

Το δένδρο εκτυπώνεται συνεχώς με την διάσχιση της προδιάταξης. Παρατηρούμε ότι όποτε πάει να ξεπεράσει το επιτρεπτό μέγιστο ύψος η insert() ανακατασκευάζει το υποδένδρο που γίνεται η εισαγωγή.



Inserting: 1 • Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Project Preferences Help

BSTree.h x output.txt x main.cpp x Inserting: 1

```
208 15
209 14
210 18
211 17
212 16
213 19
214 Height of tree is: 7
215 Number of elements: 19
216 Deletion counter: 0
217 Maximum Height Allowed: 7
218 -----
219 After Inserts:
220 1
221 9
222 5
223 3
224 2
225 4
226 7
227 6
228 8
229 13
230 11
231 10
232 12
233 15
234 14
235 18
236 17
237 16
238 19
239 Height of tree is: 7
240 Number of elements: 19
241 Deletion counter: 0
242 Maximum Height Allowed: 7
243 Deleted: 19
244 Deleted: 18
245 Deleted: 17
246 Deleted: 16
247 Deleted: 15
248 Deleted: 14
249 Deleted: 13
250 Deleted: 12
251 Deleted: 11
252 Deleted: 10
253 Deleted: 9
254 Deleted: 8
255 Deleted: 7
256 Deleted: 6
257 Deletion counter limit hit.
    Reconstructing whole tree...
258 Height of tree is: 3
    Deleted: 6
    Deletion counter limit hit.
    Reconstructing whole tree...
    Height of tree is: 3
    Number of elements: 5
    Deletion counter: 0
    Maximum Height Allowed: 5
    3
    2
    1
    5
    4
    0Found 0
    5
    
```

Line 269, Column 1

Tab Size: 4 Plain Text

# ΑΝΑΛΥΤΙΚΟΣ ΚΩΔΙΚΑΣ ΑΡΧΕΙΩΝ

---

## BSTREE.H:

```
#pragma once
#include <iostream>
#include "TreeNode.h"
#include "List.h"
#include <math.h>

template <typename T>
class BSTree
{
public:
    BSTree(double, double);    //constructor
    //~BSTree();    //SEGFAULTS
    bool isEmpty();    //bool μέθοδος, επιστρέφει True αν το δένδρο είναι άδειο
    bool searchTree(const T& key, T& found);    //bool, επιστρέφει True αν βρει στο δένδρο το
    key και το τοποθετεί στο found
    int insert(const T& elem);    //εισάγει στοιχείο στο δένδρο, επιστρέφει 1 αν πέτυχε η
    εισαγωγή
    TreeNode<T>* deleteNode(const T& key, TreeNode<T>* root, bool &flag); //βοηθητική της
    παρακάτω για διαγραφή αναδρομικά
    void deleteNode(const T& key); //διαγράφει τον κόμβο με την τιμή key
    TreeNode<T>* minNode(TreeNode<T>* node); //βρίσκει το ελάχιστο στοιχείο ενός
    υποδένδρου με ρίζα node, χρησιμοποιείται στην διαγραφή
    void printTree(char order, TreeNode<T>* node); //βοηθητική μέθοδος για την παρακάτω
    για την εκτύπωση με αναδρομικό τρόπο
    void printTree(char order); //εκτυπώνει το δένδρο με ορίσματα 'p': preorder, 'i': inorder,
    's':postorder
    void printTreeStats(); //εκτυπώνει διάφορα βοηθητικά στατιστικά όπως αριθμο στοιχείων,
    διαγραφών, ύψος
    T getNode(const TreeNode<T>* node);    //επιστρέφει τα δεδομένα ενός κόμβου
    int maxHeight(const TreeNode<T>* node); //υπολογίζει το ύψος ενός υποδένδρου με ρίζα
    node
    TreeNode<T>* getRoot(); //επιστρέφει pointer στην ρίζα του δένδρου
    int bstToList(const TreeNode<T>* node, List<T>& li); //εισάγει ένα δυαδικό δένδρο
    αναζήτησης στην λίστα li ταξινομημένα
    TreeNode<T>* sortedListToBSTrecur(List<T> &li, int count); //βοηθητική μέθοδος για την
    παρακάτω
    TreeNode<T>* sortedListToBST(List<T>& li, TreeNode<T>* node); //κατασκευάζει με
    αναδρομικό τρόπο ένα δυαδικό δένδρο αναζήτησης από ταξινομημένη λίστα με O(n)
    void treeReconstruction(TreeNode<T>* node, TreeNode<T>* parentNode, char
    succession); //ανακατασκευάζει το υποδένδρο με ρίζα node
    int countTreeNodes(TreeNode<T>* node);
    //void prettyPrint() bonus challenge
private:
    TreeNode<T> *root;
    int n;    //πλήθος στοιχείων δένδρου
    int d = 0; //αριθμός διαγραφών
    double c; //σταθερά c
    double b; //σταθερά b
};

template<typename T>
int BSTree<T>::countTreeNodes(TreeNode<T>* node)
{
    int left,right;
    if(!node)
```

```

        return 0;
    else
    {
        left = countTreeNodes(node->leftChild);
        right = countTreeNodes(node->rightChild);

        return left+right+1;
    }
}

```

/\* Επιστρέφει αναδρομικά δείκτη στον κόμβο του υποδένδρου με ρίζα node, με την μικρότερη τιμή, προφανώς σε ΔΔΑ είναι ο κάτω αριστερά κόμβος \*/

```

template<typename T>
TreeNode<T>* BSTree<T>::minNode(TreeNode<T>* node)
{
    if(node == NULL)
        return NULL;
    else if(node->leftChild == NULL)
        return node;
    else
        return minNode(node->leftChild);
}

```

/\*Κλήση αυτής της μεθόδου με μοναδικό όρισμα την τιμή του στοιχείου που θέλουμε να διαγράψουμε θα καλέσει την βοηθητική deleteNode με τα παραπάνω ορίσματα για να γίνει αναδρομικά η διαγραφή σε αυτό το ΔΔΑ έχει προστεθεί η έξτρα συνθήκη με τον αριθμο διαγραφών ώστε να γίνεται ανακατασκευή όλου του δένδρου όταν φτάσει το d στην συγκεκριμένη τιμή\*/

```

template<typename T>
void BSTree<T>::deleteNode(const T& key)
{
    bool flag = true;
    root = deleteNode(key,root,flag);
    if(flag)
    {
        std::cout<<"Deleted: "<<key<<std::endl;
        n--;
        d++;
        if(d >= ceil((pow(2,(b/c))-1)*(n+1)))
        {
            d = 0;
            std::cout<<"Deletion counter limit hit. Reconstructing whole tree..."<<std::endl;
            treeReconstruction(root,root,'l');
            printTreeStats();
        }
    }
    else
    {
        std::cout<<"Deletion failed."<<std::endl;
    }
}

```

```

template<typename T>
void BSTree<T>::printTreeStats()
{
    int heightUlt = c*log2(n+1)+b;
    std::cout<<"Height of tree is: "<<maxHeight(getRoot())<<std::endl;
    std::cout<<"Number of elements: "<<n<<std::endl;
    std::cout<<"Deletion counter: "<<d<<std::endl;
}

```

```

        std::cout<<"Maximum Height Allowed: "<<height<<std::endl;
    }

template<typename T>
void BSTree<T>::treeReconstruction(TreeNode<T>* node, TreeNode<T>* parentNode, char
succession)
{
    List<T> *li = new List<T>();
    bstToList(node, *li);
    if(node == root){
        root = sortedListToBST(*li, node);
    }
    else{
        if(succession == 'r')
            parentNode->rightChild = sortedListToBST(*li, node);
        else
            parentNode->leftChild = sortedListToBST(*li, node);
    }
    delete(li);
}

template<typename T>
TreeNode<T>* BSTree<T>::sortedListToBST(List<T> &li, TreeNode<T>* node)
{
    int count = li.length();
    node = sortedListToBSTrecur(li, count);
    return node;
}

template<typename T>
TreeNode<T>* BSTree<T>::sortedListToBSTrecur(List<T> &li, int count)
{
    //count = li.length();
    T deleted;
    if (count <= 0)
        return NULL;
    TreeNode<T> *left = sortedListToBSTrecur(li, count/2);
    TreeNode<T> *temp = new TreeNode<T>(li.getHead().getData() );
    temp->leftChild = left;
    li.deleteStart(deleted);
    temp->rightChild = sortedListToBSTrecur(li, count-count/2-1);
    return temp;
}

template<typename T>
TreeNode<T>* BSTree<T>::getRoot()
{
    return root;
}

/*Constructor του συγκεκριμένου BST
αρχικοποιεί το πλήθος των στοιχείων με 0,
τον αριθμό διαγραφών με 0 και τις τιμές b
και c σύμφωνα με αυτές που έδωσε ο χρήστης κατά
την εκτέλεση*/
template <typename T>
BSTree<T>::BSTree(double cusr, double busr)
{
    root = NULL;
    n = 0;
    d = 0;
    c = cusr;
    b = busr;
}

```

```

}

template<typename T>
bool BSTree<T>::isEmpty()
{
    if(root == NULL)
        return 1;
    else
        return 0;
}

template<typename T>
T BSTree<T>::getNode(const TreeNode<T>* node)
{
    return node->data;
}
/*Εκτύπωση του ΔΔΑ αναδρομικά με την χρήση μίας ακόμη
βοηθητικής μεθόδου με παραπάνω ορίσματα,
σύμφωνα με τον χαρακτήρα για την διάσχιση που επιθυμεί ο χρήστης
p: προδιάταξη, i: ενδοδιάταξη, s: μεταδιάταξη*/
template<typename T>
void BSTree<T>::printTree(char order)
{
    TreeNode<T> *node = root;
    //preorder
    if (order == 'p')
    {
        if(node)
        {
            std::cout<<getNode(node)<<std::endl;
            printTree('p',node->leftChild);
            printTree('p',node->rightChild);
        }
    }
    else if (order == 'i')
    //inorder
    {
        if(node)
        {
            printTree('i',node->leftChild);
            std::cout<<getNode(node)<<std::endl;
            printTree('i',node->rightChild);
        }
    }
    else if (order == 's')
    //postorder
    {
        if(node)
        {
            printTree('s',node->leftChild);
            printTree('s',node->rightChild);
            std::cout<<getNode(node)<<std::endl;
        }
    }
}

template<typename T>
void BSTree<T>::printTree(char order, TreeNode<T>* node)
{
    if (order == 'p')
    {
        if(node)

```

```

        {
            std::cout<<getNode(node)<<std::endl;
            printTree('p',node->leftChild);
            printTree('p',node->rightChild);
        }
    }
else if (order == 'i')
{
    if(node)
    {
        printTree('i',node->leftChild);
        std::cout<<getNode(node)<<std::endl;
        printTree('i',node->rightChild);
    }
}
else if (order == 's')
{
    if(node)
    {
        printTree('s',node->leftChild);
        printTree('s',node->rightChild);
        std::cout<<getNode(node)<<std::endl;
    }
}
}

```

/\*αναζήτηση στο ΔΔΑ για το στοιχείο με την τιμή key  
και τοποθέτηση του εφόσον το βρίσκει στην μεταβλήτη e \*/

```

template<typename T>
bool BSTree<T>::searchTree(const T& key, T& e)
{

```

```

    TreeNode<T> *current = root;
    while(current)
    {
        if (key < current->data)
        {
            current = current->leftChild;
        }
        else if (key > current->data)
        {
            current = current->rightChild;
        }
        else
        {
            e = current->data;
            return true;
        }
    }
    return false;
}

```

/\*εισαγωγή του στοιχείου elem στο ΔΔΑ  
έχει προστεθεί η έξτρα συνθήκη στο συγκεκριμένο ΔΔΑ  
για να διατηρεί την ισορροπία του\*/

```

template<typename T>
int BSTree<T>::insert(const T& elem)
{

```

```

    TreeNode<T> *current = root;
    TreeNode<T> *parent = 0;
    /*θα χρησιμοποιηθεί μία δομή απλά συνδεδεμένης λίστας
    θα μπορούσε να χρησιμοποιηθεί μία απλή στοίβα αλλά αφού είχαμε
    υλοποιήσει συναρτήσεις push και pop (insertStart, deleteStart) για την 1η εργασία
    πρακτικά η

```

```

λίστα μπορεί να χρησιμοποιηθεί και ως στοίβα*/
List<TreeNode<T>*> *li = new List<TreeNode<T>*>();
int depth = 0;
while (current)
{
    parent = current;
    li->insertStart(parent);
    if(elem < current->data)
    {
        current = current->leftChild;
        depth++;
    }
    else if (elem > current->data)
    {
        current = current->rightChild;
        depth++;
    }
    else
        return -1;
}
n++;
TreeNode<T> *newnode = new TreeNode<T>(elem);
if (!isEmpty())
{
    if (elem < parent->data)
        parent->leftChild = newnode;
    else
        parent->rightChild = newnode;
}
else
    root = newnode;
std::cout<<"Inserting: "<<elem<<std::endl;
//Έλεγχος βάθους εισαγόμενου κόμβου και ανακατασκευή αν χρειάζεται
if (depth > ceil(c*log2(n+1+d)))
{
    TreeNode<T> *temp;
    TreeNode<T> *papatemp;
    int lisz = li->length();
    for(int i = 0; i < lisz; i++)
    {
        li->deleteStart(temp);\
        char dir;
        if(li->deleteStart(papatemp) == 0)
        {
            dir = 'r';
            papatemp = root;
        }
        if(temp->data > papatemp->data)
        {
            dir = 'r';
        }
        else
        {
            dir = 'l';
        }
        if (maxHeight(temp) > ceil(c*log2(countTreeNode(temp)+1)))
        {
            printTree('p');
            printTreeStats();
            /* το όρισμα dir μας δείχνει αν ήταν
            αριστερό ή δεξί παιδί ο κόμβος ρίζα του υποδένδρου
            που θα ανακατασκευαστεί για την σωστή σύνδεσή του

```

```

        αν αυτός ο κόμβος είναι η ρίζα του αρχικού δένδρου
        και έχουμε ολική ανακατασκευή το όρισμα αυτό είναι
        irrelevant */
        treeReconstruction(temp,papatemp, dir);
        std::cout<<"Reconstructing tree..."<<std::endl;
        printTree('p');
        printTreeStats();

        delete(li);
        break;
    }
    li->insertStart(papatemp);
}
return depth;
}

/*διαγραφή κόμβου με αναδρομή*/
template<typename T>
TreeNode<T>* BSTree<T>::deleteNode(const T& key, TreeNode<T>* node, bool &flag)
{
    TreeNode<T>* temp;
    if(node == NULL)
    {
        flag = false;
        std::cout<<key<<" not found"<<std::endl;
        return NULL;
    }
    else if(key < node->data)
    {
        node->leftChild = deleteNode(key, node->leftChild,flag);
    }
    else if(key > node->data)
    {
        node->rightChild = deleteNode(key, node->rightChild,flag);
    }
    else if(node->leftChild and node->rightChild)
    {
        temp = minNode(node->rightChild);
        node->data = temp->data;
        node->rightChild = deleteNode(node->data, node->rightChild,flag);
    }
    else
    {
        temp = node;
        if(node->leftChild == NULL)
            node = node->rightChild;
        else if(node->rightChild == NULL)
            node = node->leftChild;
        delete temp;
    }
    return node;
}

/*αναδρομικός υπολογισμός ύψους υποδένδρου με ρίζα
node */
template<typename T>
int BSTree<T>::maxHeight(const TreeNode<T>* node)
{
    if(!node)
        return 0;
    else

```



```

    {
        int lDepth = maxHeight(node->leftChild);
        int rDepth = maxHeight(node->rightChild);
        if(lDepth > rDepth)
            return (lDepth+1);
        else
            return (rDepth+1);
    }
}

```

/\*Γέμισμα λίστας li με τα στοιχεία του δένδρου node  
ταξινομημένα, διασχίζοντας το με ενδοδιάταξη\*/

```

template<typename T>
int BSTree<T>::bstToList(const TreeNode<T>* node, List<T>& li)
{
    if(node)
    {
        bstToList(node->leftChild, li);
        li.insertEnd(getNode(node));
        bstToList(node->rightChild, li);
    }
    return 1;
}

```

# TREENODE.H:

```
#pragma once
template <typename T>
class BSTree;

template <typename T>
class TreeNode
{
    friend class BSTree<T>;
private:
    T data;
    TreeNode* leftChild;
    TreeNode* rightChild;

public:
    TreeNode(T dataIn);
    TreeNode()
    {};
    T getData();

};

template <typename T>
TreeNode<T>::TreeNode(T dataIn)
{
    data = dataIn;
    leftChild = 0;
    rightChild = 0;
}

template <typename T>
T TreeNode<T>::getData()
{
    return data;
}
```

# MAIN.CPP:

```
#include "BSTree.h"
#include "TreeNode.h"
#include <iostream>
#include "List.h"
#include <math.h>

int main(int argc, char* argv[])
{
    if(argc < 3)
    {
        std::cerr << "Usage: "<<argv[0]<<" c b "<<std::endl;
        return 1;
    }
    double c = atof(argv[1]);
    double b = atof(argv[2]);
    if (c <= 1)
    {
        std::cerr << "C must be bigger than 1"<<std::endl;
        return 1;
    }
    if(b <= 0)
    {
        std::cerr << "B must be bigger than 0"<<std::endl;
        return 1;
    }
    BSTree<int> *d1 = new BSTree<int>(c,b);

    // d1->insert(8);
    // d1->insert(5);
    // d1->insert(16);
    // d1->insert(1);
    // d1->insert(7);
    // d1->insert(14);
    // d1->insert(19);
    // d1->insert(4);
    // d1->insert(6);
    // d1->insert(9);
    // d1->insert(15);
    // d1->insert(18);
    // d1->insert(3);
    // d1->insert(12);
    // d1->insert(17);
    // d1->insert(2);
    // d1->insert(11);
    // d1->insert(13);
    // d1->insert(10);
    // d1->insert(40);
    // d1->insert(20);
    // d1->insert(22);
    // d1->insert(30);

    for(int i = 1; i<20; i++)
    {
        d1->insert(i);
    }
    //d1->printTree('p');
    //std::cout<<"Height of treess is: "<<d1->maxHeight(d1->getRoot())<<std::endl;
    std::cout<<"-----\nAfter Inserts:\n";
    d1->printTree('p');
    d1->printTreeStats();
    int temp = 0;
    // d1->deleteNode(19);
    // d1->printTree('p');
    // d1->deleteNode(18);
    // d1->printTree('p');
    // d1->deleteNode(17);
    // d1->printTree('p');
```

```

// d1->deleteNode(16);
// d1->printTree('p');
// d1->deleteNode(2229);
// d1->printTree('p');
for(int i = 19; i>5; i--)
{
    d1->deleteNode(i);
}
d1->printTree('p');
std::cout<<d1->searchTree(666,temp)<<"Found "<<temp<<std::endl;
temp = d1->countTreeNodes(d1->getRoot());
std::cout<<temp<<std::endl;

return 0;
}

```

## GITHUB REPOSITORY:

<https://github.com/anthonysof/self-balancing-BST>

Η απλα συνδεδεμένη λίστα έχει υλοποιηθεί στην προηγούμενη εργασία.