Εθνικό και Καποδιστριακό
Πανεπιστήμιο Αθηνών

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΦΥΣΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΣΧΕΔΙΑΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΔΕΝΤΡΑ ΔΙΑΤΑΚΤΙΚΗΣ ΣΤΑΤΙΣΤΙΚΗΣ

ΚΟΥΡΗΣ ΔΙΟΝΥΣΙΟΣ

ΕΠΙΒΛΕΠΩΝ: ΡΕΪΣΗΣ ΔΙΟΝΥΣΙΟΣ

ΑΘΗΝΑ
ΙΟΥΛΙΟΣ 2017

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

ΣΧΕΔΙΑΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΔΕΝΤΡΑ ΔΙΑΤΑΚΤΙΚΗΣ ΣΤΑΤΙΣΤΙΚΗΣ

ΚΟΥΡΗΣ ΔΙΟΝΥΣΙΟΣ
Α.Μ.: 200500326

ΕΠΙΒΛΕΠΩΝ: ΡΕΪΣΗΣ ΔΙΟΝΥΣΙΟΣ, ΑΝΑΠΛ. ΚΑΘΗΓΗΤΗΣ

ΠΕΡΙΛΗΨΗ

Τα δυαδικά δέντρα αναζήτησης είναι μοντέλα δομών δεδομένων, που αποτελούνται από κόμβους, στους οποίους περιέχονται τα δεδομένα που θέλουμε να αποθηκευτούν. Ικανοποιούν την απαίτηση, η τιμή ενός κόμβου να είναι μεγαλύτερη από την τιμή οποιουδήποτε κόμβου στο αριστερό υποδέντρο, και μικρότερη από την τιμή οποιουδήποτε κόμβου στο δεξί υποδέντρο. Υποστηρίζουν τις βασικές πράξεις της εισαγωγής, αναζήτησης και διαγραφής,, καθώς και την εύρεση του διαδόχου και του προκατόχου, καθώς και του κόμβου με τη μέγιστη ή την ελάχιστη τιμή κλειδιού. Στη χειρότερη περίπτωση, ο χρόνος εκτέλεσης των πράξεων είναι $O(n)$, όπου n είναι ο αριθμός των κόμβων του δέντρου.

Προκειμένου να αποφευχθεί η χειρότερη περίπτωση, υπάρχουν διάφορες εκδοχές δυαδικών δέντρων, όπως είναι τα μελανέρυθρα δέντρα, που εξασφαλίζουν χρόνο εκτέλεσης των πράξεων επί του δέντρου σε χρόνο $O(\log n)$.

Σε ορισμένες περιπτώσεις, όμως, δεν αρκούν οι τυποποιημένες δομές δεδομένων, αλλά χρειάζεται να επαυξηθούν. Μια τέτοια δομή δεδομένων είναι τα δέντρα διατακτικής στατιστικής, τα οποία έχουν δύο πρόσθετες πράξεις: α. Της εύρεσης του $\nu - o\sigma\tau o\acute{\upsilon}$ μικρότερου στοιχείου του δέντρου, και β. της εύρεσης της τάξης οποιοδήποτε στοιχείου του δέντρου.

Περιεχόμενα

Κεφάλαιο 1^{ο}

Εισαγωγή

Στο κεφάλαιο αυτό παρουσιάζεται το θέμα της παρούσας πτυχιακής εργασίας. Αρχικά δίνεται μια εισαγωγή του αντικειμένου της εργασίας, και στη συνέχεια αναφέρεται ο στόχος τον οποίο θέλουμε να επιτύχουμε, με την ανάπτυξη του αντίστοιχου προγράμματος. Τέλος, αναφέρεται η διάρθρωση της εργασίας

1.1 Αντικείμενο εργασίας

Το αντικείμενο της παρούσας πτυχιακής εργασίας είναι η σχεδίαση και η υλοποίηση ενός προγράμματος, που θα βασίζεται σε επαυξημένες δομές δεδομένων. Πιο συγκεκριμένα, θα υλοποιηθεί με τη χρήση ενός επαυξημένου μελανέρυθρου δέντρου (red-black tree), και θα έχει δυνατότητα εισαγωγής, διαγραφής, αναζήτησης κόμβων, καθώς και την εύρεση του $\nu - οστού$ μικρότερου κόμβου, και την τάξη κάποιου κόμβου του δέντρου.
Η υλοποίηση θα γίνει σε γλώσσα προγραμματισμού C.

1.2 Στόχος της εργασίας

Στόχος της εργασίας είναι η μελέτη και η αξιοποίηση των δομών δεδομένων, και η χρήση τους στην υλοποίηση προγραμμάτων.

1.3 Διάρθρωση εργασίας

Το κεφάλαιο 1 περιλαμβάνει μια μικρή εισαγωγή στο τι πραγματεύεται αυτή η εργασία.

Το κεφάλαιο 2 περιλαμβάνει μια σύντομη εισαγωγή στις δομές δεδομένων.

Το κεφάλαιο 3 αναλύει τα δυαδικά δέντρα αναζήτησης.

Το κεφάλαιο 4 περιλαμβάνει την περιγραφή των μελανέρυθρων δέντρων, και τις διαφορές τους με τα δυαδικά δέντρα.

Στο κεφάλαιο 5 έχουμε αναφορά στα δέντρα διατακτικής στατιστικής, και στην υλοποίηση του δέντρου.

Κεφάλαιο 2ο

Δομές δεδομένων

2.1 Τι είναι οι δομές δεδομένων

Στην Επιστήμη των Υπολογιστών, δομές δεδομένων καλούνται οι διάφορες μέθοδοι αποθήκευσης και οργάνωσης δεδομένων. Σκοπός των δομών δεδομένων είναι η διευκόλυνση της προσπέλασης και τροποποίησης των δεδομένων αυτών.

Ανάλογα με τις ανάγκες του προβλήματος χρησιμοποιούνται διαφορετικές δομές δεδομένων, με διαφορετικά χαρακτηριστικά, πλεονεκτήματα, μειονεκτήματα και περιορισμούς.

Ορισμένοι από τους πλέον γνωστούς και ευρέως χρησιμοποιούμενους τύπους δομών δεδομένων είναι:

- Πίνακες
- Λίστες
- Δέντρα
- Στοίβα
- Γράφοι

Στα πλαίσια αυτής της πτυχιακής θα ασχοληθούμε με τα δέντρα, και ειδικότερα μια υποκατηγορία αυτών, τα δέντρα διατακτικής στατιστικής.

2.2 Στοιχεία δομής δεδομένων

Σε μια τυπική υλοποίηση μιας δομής δεδομένων, τα στοιχεία αναπαρίστανται από αντικείμενα. Τα αντικείμενα αυτά, μπορούν να περιέχουν παρελκόμενα δεδομένα, καθώς κι ένα πεδίο κλειδί. Τα παρελκόμενα δεδομένα ακολουθούν τις μετακινήσεις του κλειδιού, αλλά πέραν τούτου δεν επηρεάζουν την υλοποίηση της δομής δεδομένων.

2.3 Πράξεις σε δομές δεδομένων

Οι διάφοροι αλγόριθμοι, οι οποίοι χρησιμοποιούν και αξιοποιούν τις δομές δεδομένων, απαιτούν την εκτέλεση διαφόρων πράξεων στις δομές αυτές.

- Τροποποιητικές, οι οποίες μεταβάλλουν τη δομή
- Ερωτηματικές, οι οποίες επιστρέφουν πληροφορίες σχετικά με τη δομή

Ανεξάρτητα από το εάν μιλάμε για λίστες, πίνακες, δέντρα ή οποιονδήποτε άλλο τύπο δομής δεδομένων, υπάρχουν 3 βασικές που απαντώνται πάντα:

- $Εισαγωγή\ (S, x)$, η οποία είναι μια τροποποιητική πράξη, η οποία επαυξάνει τη δομή S με το στοιχείο που υποδεικνύει ο δείκτης x.
- $Αναζήτηση\ (S, k)$. Η Αναζήτηση είναι μια ερωτηματική πράξη, η οποία επιστρέφει έναν δείκτη x, προς ένα στοιχείο της δομής S, ο οποίος εξαρτάται από ένα κλειδί k, $κλειδί[k] = x$, ή ΚΕΝΟ εάν η δομή δεν περιέχει τέτοιο στοιχείο.
- $Διαγραφή\ (S, x)$, η οποία είναι μια τροποποιητική πράξη, η οποία διαγράφει το στοιχείο x από μια δομή δεδομένων S.

Άλλες πράξεις που απαντώνται συχνά είναι:

- $Ελάχιστο\ (S)$, η οποία είναι μια ερωτηματική πράξη, η οποία επιστρέφει το μικρότερο στοιχείο της δομής
- $Μέγιστο\ (S)$. Επίσης είναι ερωτηματική πράξη, και επιστρέφει το μέγιστο στοιχείο της δομής.
- $Διάδοχος\ (S, x)$. Ερωτηματική πράξη, η οποία, όταν δοθεί ένα κλειδί x, επιστρέφει το αμέσως μεγαλύτερο στοιχείο της δομής
- $Προκάτοχος\ (S, x)$. Ερωτηματική πράξη, η οποία, όταν δοθεί ένα κλειδί x, επιστρέφει το αμέσως μικρότερο στοιχείο της δομής
- $Ταξινόμηση\ (S)$. Τροποποιητική πράξη, η οποία ταξινομεί τα δεδομένα μιας δομής, από το μικρότερο προς το μεγαλύτερο ή το αντίθετο
- $Συγχώνευση\ (S_1, S_2)$. Τροποποιητική πράξη, η οποία συγχωνεύει δύο ταξινομημένες δομές, σε μία νέα, επίσης ταξινομημένη δομή.

Κεφάλαιο 3ο

Δυαδικά δέντρα αναζήτησης

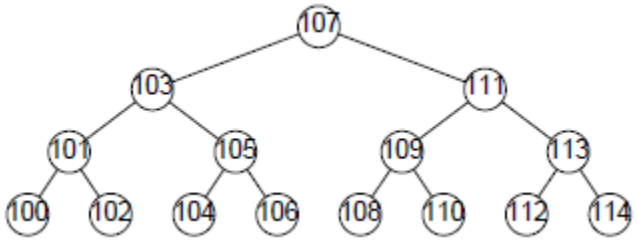
3.1 Τι είναι τα δυαδικά δέντρα αναζήτησης

Τα δυαδικά δέντρα αναζήτησης (Binary Search Trees – BSTs), όπως υποδηλώνει και το όνομά τους, έχουν δομή δυαδικού δέντρου, και μπορούν να αναπαρίστανται από μια συνδεδεμένη δομή δεδομένων, στην οποία κάθε κόμβος είναι ένα αντικείμενο. Σε κάθε αντικείμενο – κόμβο, εκτός από το πεδίο κλειδί και τα παρελκόμενα δεδομένα, υπάρχουν και 3 ακόμη πεδία:

- Γονιός, που υποδεικνύει τον πατρικό κόμβο
- Αριστερός, που υποδεικνύει τον αριστερό θυγατρικό κόμβο
- Δεξιός, που υποδεικνύει τον δεξιό θυγατρικό κόμβο

Εάν κάποιος εξ αυτών των κόμβων δεν υπάρχει, το αντίστοιχο πεδίο παίρνει την τιμή ΚΕΝΟ. Ο μόνος κόμβος που έχει την τιμή ΚΕΝΟ, είναι η ρίζα του δέντρου.

Ένα τυπικό παράδειγμα δυαδικού δέντρου αναζήτησης φαίνεται στην Εικόνα 2.1



Εικόνα 2.1 Δυαδικό Δέντρο Αναζήτησης

Σε αυτό το δέντρο, ο κόμβος στην κορυφή, με τιμή 107 είναι ο κόμβος από τον οποίον εκκινούν όλες οι αναζητήσεις στο δέντρο, και για αυτό το λόγο ονομάζεται ρίζα του δέντρου (root). Ο κόμβος που βρίσκεται κάτω αριστερά από τη ρίζα, με τιμή 103, είναι ο αριστερός θυγατρικός κόμβος της ρίζας, ενώ ο κόμβος κάτω δεξιά, με τιμή 111, είναι ο δεξιός θυγατρικός κόμβος.

Ορισμένοι κόμβοι, όπως αυτός με τιμή 104, δεν έχουν παιδιά. Αυτοί οι κόμβοι ονομάζονται φύλλα (leaves) ή τερματικοί κόμβοι (terminal nodes). Ένας κόμβος, επίσης, δύναται να έχει μόνο ένα θυγατρικό κόμβο, είτε δεξιό είτε αριστερό.

Κάθε κόμβος σε ένα δυαδικό δέντρο μπορεί να θεωρηθεί ως η ρίζα του δικού του δέντρου. Ένα τέτοιο δέντρο ονομάζεται υποδέντρο του δέντρου στο οποίο ανήκει. Ο αριστερός θυγατρικός κόμβος, και οι θυγατρικοί κόμβοι αυτού, ονομάζονται αριστερό υποδέντρο. Αντίστοιχα ορίζεται το δεξί υποδέντρο, για τη δεξιά πλευρά του δέντρου. Για το δέντρο της εικόνας 2.1, οι κόμβοι με τιμές 108, 109 και 110 είναι το αριστερό υποδέντρο του κόμβου 111, με τον κόμβο με τιμή 109 να είναι η ρίζα του υποδέντρου.

3.2 Ιδιότητα δυαδικού δέντρου αναζήτησης

Η τοποθέτηση των κόμβων σε ένα δυαδικό δέντρο αναζήτησης γίνεται με τρόπο τέτοιο ώστε να ικανοποιείται η ιδιότητα του δυαδικού δέντρου αναζήτησης:

> Έστω ένας κόμβος x σε ένα δυαδικό δέντρο αναζήτησης. Οποιοσδήποτε κόμβος y στο αριστερό υποδέντρο του x θα πρέπει να ικανοποιεί τη συνθήκη $κλειδί[x] \geq κλειδί[y]$. Αντίστοιχα, για οποιονδήποτε κόμβο στο δεξί υποδέντρο του x θα πρέπει να ικανοποιείται η σχέση $κλειδί[x] \leq κλειδί[y]$.

Δηλαδή, σε ένα δυαδικό δέντρο αναζήτησης (και σε όλες τις άλλες εξελίξεις τους, όπως στα μελανέρυθρα δέντρα), ο αριστερός θυγατρικός κόμβος, εφόσον υπάρχει, θα πρέπει να έχει τιμή μικρότερη ή ίση με τον πατρικό κόμβο, ενώ ο δεξιός θυγατρικός κόμβος θα πρέπει να έχει μεγαλύτερη ή ίση με τον πατρικό κόμβο. Ο κανόνας αυτός δεν ισχύει για τα δυαδικά δέντρα.

3.3 Πράξεις επί των δυαδικών δέντρων αναζήτησης

3.3.1 Συμμετρική Διάνυση Δέντρου

Η συμμετρική διάνυση δέντρου (inorder traversal) επιτρέπει την εκτύπωση των στοιχείων ενός δέντρο, με καθορισμένη ταξινόμηση, με τη βοήθεια ενός αναδρομικού αλγόριθμου, χάρη στην ιδιότητα του δυαδικού δέντρου αναζήτησης.

Inorder-Traversal *(x)*

```
1. if x ≠ NIL
2.     then Inorder-Traversal(left[x])
3.          print key[x]
4.          Inorder-Traversal(right[x])
```

Για να εκτυπωθούν όλα τα στοιχεία του δέντρου, αρκεί να καλέσουμε την Inorder-Traversal (root).

3.3.2. Αναζήτηση

Για την αναζήτηση ενός κόμβου, ενός δυαδικού δέντρου αναζήτησης, που να περιέχει ένα δεδομένο κλειδί, χρησιμοποιούμε τη διαδικασία $Search\ (x, k)$, όπου δέχεται ως ορίσματα ένα δείκτη προς τη ρίζα και ένα κλειδί k, και επιστρέφει έναν δείκτη προς κάποιον κόμβο με κλειδί k, εφόσον υπάρχει, ειδάλλως επιστρέφει την τιμή ΚΕΝΟ.

TREE-SEARCH*(x, k)*

```
1. if x = NIL or k = key[x]
2.     then return x
3. if k < key[x]
4.     then return TREE-SEARCH(left[x], k)
5.     else return TREE-SEARCH(right[x], k)
```

Αυτή η διαδικασία ξεκινάει από τη ρίζα του δέντρου, και προχωράει προς τα κάτω, έως ότου βρει τον κόμβο που να περιέχει το κλειδί. Σε κάθε κόμβο που συναντάει, κάνει σύγκριση της τιμής k με το $κλειδί[x]$. Εάν το k είναι μεγαλύτερο από την τιμή $κλειδί[x]$ κινείται προς τον δεξιό θυγατρικό κόμβο, ειδάλλως προς τον αριστερό θυγατρικό κόμβο. Αν $κλειδί[x] = k$, η αναζήτηση τερματίζεται.

3.3.3 Ελάχιστο και μέγιστο

Σε ένα δυαδικό δέντρο αναζήτησης υπάρχει η δυνατότητα εύρεσης του κόμβου του οποίου το κλειδί είναι ελάχιστο. Για να επιτευχθεί αυτό, αξιοποιείται η ιδιότητα του δυαδικού δέντρου αναζήτησης, κι έτσι, εκκινώντας από τη ρίζα του δέντρου, ακολουθούνται οι διαδοχικοί δείκτες Αριστερός, προς τον εκάστοτε αριστερό θυγατρικό κόμβο, μέχρις ότου να απαντηθεί η τιμή ΚΕΝΟ, όπως δείχνει ο ψευδοκώδικας:

MINIMUM*(x)*

```
1. while left[x] ≠ NIL
2.     do x ← left[x]
3. return x
```

Αντίστοιχη θα είναι η διαδικασία για να βρεθεί ο μέγιστος, με τη διαφορά ότι θα ακολουθούνται, τώρα, διαδοχικοί δείκτες προς τα δεξιά:

MAXIMUM*(x)*

```
1. while right[x] ≠ NIL
2.     do x ← right[x]
3. return x
```

3.3.4 Διάδοχος και προκάτοχος

Άλλες χρήσιμες πράξεις είναι η εύρεση του διαδόχου ή του προκατόχου ενός κόμβου. Διάδοχος ενός κόμβου είναι ο κόμβος που έχει το μικρότερο κλειδί από το σύνολο των κόμβων με κλειδί μεγαλύτερο του $κλειδί[x]$. Αντίστοιχα, προκάτοχος είναι ο κόμβος που έχει το μεγαλύτερο κλειδί από το σύνολο των κόμβων με κλειδί μικρότερο του $κλειδί[x]$.

Για την εύρεση του διαδόχου, εξετάζονται 2 περιπτώσεις:

- Εάν ο κόμβος x έχει δεξιό υποδέντρο, τότε ο διάδοχος του χ είναι απλώς ο ακραίος αριστερός κόμβος αυτού του δεξιού υποδέντρου.
- Αν ο κόμβος x δεν έχει δεξιό υπόδεντρο, τότε ο διάδοχος είναι ο κατώτερος πρόγονος του x του οποίου ο αριστερός θυγατρικός κόμβος είναι επίσης πρόγονος του x

SUCCESSOR*(x)*

```
1. if right[x] ≠ NIL
2.     then return MINIMUM(right[x])
```

```
3. y ← p[x]
4. while y ≠ NIL and x = right[y]
5.     do x ← y
6.         y ← p[y]
7. return y
```

Αντίστοιχα, για τον προκάτοχο, ο ψευδοκώδικας θα είναι:

TREEPREDECESSOR*(x)*

```
1. if left[x] ≠ NIL
2.     then return MAXIMUM(left[x])
3. y ← p[x]
4. while y ≠ NIL and x = left[y]
5.     do x ← y
6.         y ← p[y]
7. return y
```

3.3.5 Εισαγωγή

Για την εισαγωγή μιας νέας τιμής υ σε ένα δυαδικό δέντρο αναζήτησης Τ, χρησιμοποιείται η διαδικασία Εισαγωγή, η οποία δέχεται ως είσοδο έναν κόμβο z με $\kappa\lambda\epsilon\iota\delta\acute{\iota}\ [z] = \upsilon$, $\alpha\rho\iota\sigma\tau\epsilon\rho\acute{o}\varsigma[z] = KENO$ και $\delta\epsilon\xi\iota\acute{o}\varsigma[z] = KENO$. Η διαδικασία τροποποιεί το δέντρο Τ και ορισμένα από τα πεδία του z με τέτοιο τρόπο ώστε το z να εισαχθεί στην κατάλληλη θέση του δέντρου.

INSERT*(T, z)*

```
1. y ← NIL
2. x ← root[T ]
3. while x ≠ NIL
4.     do y ← x
5.         if key[z] < key[x]
6.             then x ← left[x]
```

```
7.            else x ← right[x]
8. p[z]← y
9. if y = NIL
10.         then root[T]← z
11.         else if key[z] < key[y]
12.             then left[y]← z
13.             else right[y] ← z
```

3.3.6 Διαγραφή

Για τη διαγραφή ενός κόμβου z από ένα δυαδικό δέντρο αναζήτησης, χρησιμοποιείται μια διαδικασία που έχει ως παράμετρο έναν δείκτη προς τον κόμβο z. Για τη διαγραφή ενός κόμβου ελέγχονται τρεις διαφορετικές περιπτώσεις:

- Εάν ο z δεν έχει θυγατρικούς κόμβους, απλώς αφαιρείται
- Εάν ο z έχει μόνο έναν θυγατρικό κόμβο, αποκόπτεται ο z.
- Εάν ο z έχει δύο θυγατρικούς κόμβους, αποκόπτεται ο διάδοχός του, y, ο οποίος δεν έχει αριστερό θυγατρικό κόμβο και αντικαθίσταται το κλειδί και τα παρελκόμενα δεδομένα του z με τις αντίστοιχες τιμές του y

DELETE*(T, z)*

```
1. if left[z] = NIL or right[z] = NIL
2.     then y ← z
3.     else y ← TREE-SUCCESSOR(z)
4. if left[y] ≠ NIL
5.     then x ← left[y]
6.     else x ← right[y]
7. if x ≠ NIL
8.     then p[x] ← p[y]
9. if p[y] = NIL
10.         then root[T ] ← x
11.         else if y = left[p[y]]
12.             then left[p[y]] ← x
13.             else right[p[y]] ← x
14.     if y ≠ z
```

```
15.         then key[z]← key[y]
16.             copy y’s satellite data into z
17.     return y
```

Κεφάλαιο 4ο

Μελανέρυθρα δέντρα (Red-Black Trees)

4.1 Τι είναι τα μελανέρυθρα δέντρα

Τα δυαδικά δέντρα αναζήτησης έχουν χρόνο εκτέλεσης των βασικών πράξεων που αναλύθηκαν στο προηγούμενο κεφάλαιο $O(h)$. Είναι αποδοτικά, έχουν δηλαδή μικρό χρόνο εκτέλεσης για τις πράξεις που αναλύθηκαν, εφόσον το ύψος τους είναι μικρό. Εάν, όμως, το ύψος τους είναι μεγάλο, η επίδοσή τους καθίσταται συγκρίσιμη με την αντίστοιχη επίδοση σε αλυσίδα. Καθώς η κατασκευή του δυαδικού δέντρου αναζήτησης είναι τυχαία, δεν είναι δυνατή η πρόβλεψη της μορφής του και του ύψους του. Αυτό το πρόβλημα έρχονται να λύσουν τα μελανέρυθρα δέντρα, τα οποία είναι ισοσταθμισμένα, με σκοπό να εξασφαλιστεί ότι οι βασικές πράξεις δυναμικού συνόλου απαιτούν, στη χειρότερη περίπτωση, χρόνο $O(\log n)$.

Τα μελανέρυθρα δέντρα είναι δυαδικά δέντρα αναζήτησης, τα οποία σε κάθε κόμβο, πέρα από το κλειδί και τα παρελκόμενα δεδομένα, έχουν ένα επιπλέον στοιχείο πληροφορίας: το χρώμα του, το οποίο, όπως προδίδει το όνομά τους, μπορεί να είναι είτε μαύρο (μελανό) είτε κόκκινο (ερυθρό). Μέσω των ιδιοτήτων τους, που επιβάλλουν περιορισμούς στο πως θα χρωματιστούν οι κόμβοι του δέντρου, εξασφαλίζεται ότι καμία διαδρομή από καταληκτικό κόμβο έως τη ρίζα, δε γίνεται να έχει μήκος μεγαλύτερο από το διπλάσιο οποιασδήποτε άλλης διαδρομής. Επομένως το δέντρο είναι κατά προσέγγιση ισοσταθμισμένο.

Τα πεδία κάθε κόμβου σε ένα τέτοιο δέντρο είναι

- Κλειδί
- Γονιός, που υποδεικνύει τον πατρικό κόμβο
- Αριστερός, που υποδεικνύει τον αριστερό θυγατρικό κόμβο
- Δεξιός, που υποδεικνύει το δεξιό θυγατρικό κόμβο
- Χρώμα, που δηλώνει το χρώμα του κόμβου

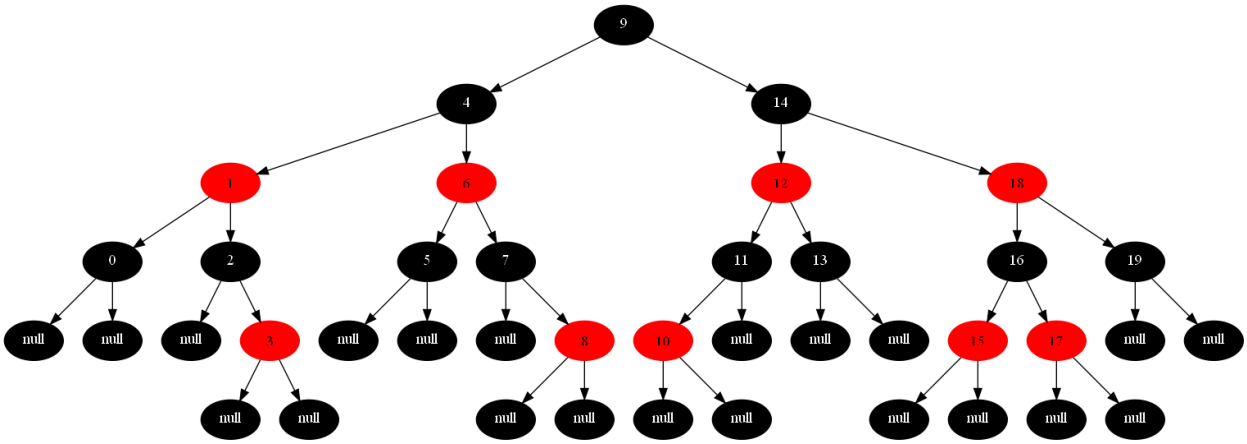
Εάν κάποιος θυγατρικός ή ο πατρικός κόμβος δεν υπάρχει, τότε το αντίστοιχο πεδίο θα περιέχει την τιμή ΚΕΝΟ.

4.2 Ιδιότητες Μελανέρυθρων δέντρων

Ένα δυαδικό δέντρο αναζήτησης συνιστά μελανέρυθρο δέντρο εάν ικανοποιεί τις εξής ιδιότητες:

- Κάθε κόμβος είναι είτε μελανός είτε ερυθρός
- Ο κόμβος της ρίζας είναι μελανός
- Κάθε καταληκτικός κόμβος είναι μελανός
- Εάν ένας κόμβος είναι ερυθρός, τότε και οι δύο θυγατρικοί κόμβοι του είναι μελανοί
- Για κάθε κόμβο, όλες οι διαδρομές από αυτόν προς τους καταληκτικούς κόμβους-απογόνους του, περιέχουν ισάριθμους μελανούς κόμβους.

Στην εικόνα που ακολουθεί βλέπουμε ένα παράδειγμα μελανέρυθρου δέντρου.



Εικόνα 3.1 Μελανέρυθρο δέντρο

4.3 Πράξεις επί των μελανέρυθρων δέντρων

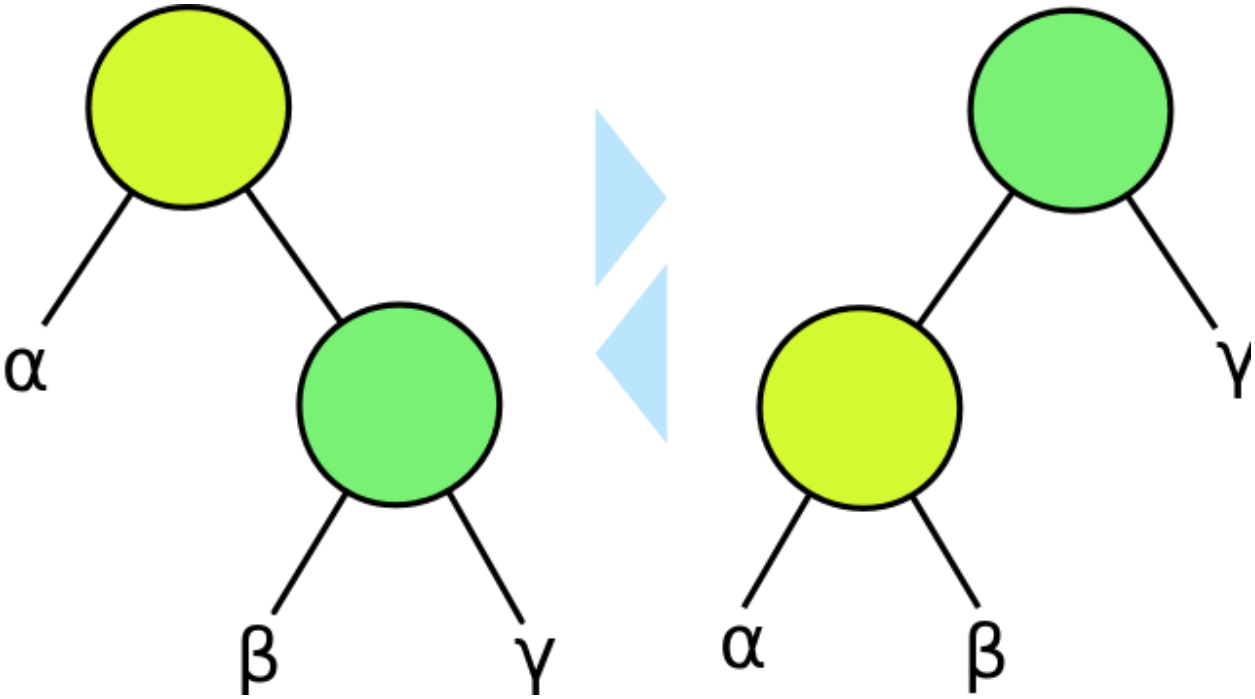
Καθώς τα μελανέρυθρα δέντρα είναι παράγωγα των δυαδικών δέντρων αναζήτησης, οι μη τροποποιητικές πράξεις (Αναζήτηση, Ελάχιστο, Μέγιστο,

Διάδοχος, Προκάτοχος) είναι οι ίδιες. Οι τροποποιητικές πράξεις, όμως, η Εισασγωγή και η Διαγραφή, δηλαδή, ενδέχεται να επηρρεάσουν το δέντρο και οι ιδιότητες του μελανέρυθρου δέντρου να μην ικανοποιούνται μετά την εκτέλεσή τους. Για αυτό το λόγο, μετά την εκτέλεση των πράξεων αυτών, θα πρέπει να μεταβάλλουμε το χρώμα ορισμένων κόμβων του δέντρου, καθώς και τη δομή των δεικτών.

Η δομή των δεικτών μεταβάλλεται με περιστροφή, αριστερή ή δεξιά. Η περιστροφή είναι μια τοπική πράξη σε ένα δέντρο αναζήτησης η οποία διατηρεί την ιδιότητα του δυαδικού δέντρου αναζήτησης.

4.3.1 Περιστροφές

Στην εικόνα 3.2 βλέπουμε τα δύο είδη περιστροφών, και στη συνέχεια παρατίθεται ο ψευδοκώδικας για την αριστερή και στη συνέχεια τη δεξιά περιστροφή:



Εικόνα 3.2 Περιστροφές

LEFT-ROTATE*(T, x)*

```
1.  y ← right[x]
2.  right[x] ← left[y]
3.  if left[y] ≠ nil[T ]
```

```
4.      then p[left[y]] ← x
5.  p[y] ← p[x]
6.  if p[x] = nil[T ]
7.      then root[T ] ← y
8.      else if x = left[p[x]]
9.          then left[p[x]] ← y
10.         else right[p[x]] ← y
11. left[y] ← x
12. p[x] ← y
```

RIGHT-ROTATE*(T, x)*

```
1.  y ← left[x]
2.  left[x] ← right[y]
3.  if right[y] ≠ nil[T]
4.      then p[right[y]] ← x
5.  p[y] ← p[x]
6.  if p[x] = nil[T ]
7.      then root[T ] ← y
8.      else if x = right[p[x]]
9.          then right[p[x]] ← y
10.         else left[p[x]] ← y
11. right[y] ← x
12. p[x] ← y
```

Με τις περιστροφές ορισμένες, μπορούμε να δούμε πως αλλάζουν οι πράξεις της Εισαγωγής και της Διαγραφής

4.3.2 Εισαγωγή

Η εισαγωγή κόμβου σε μελανέρυθρο δέντρο γίνεται με μια ελαφρά τροποποιημένη εκδοχή της διαδικασίας που χρησιμοποιείται για το δυαδικό δέντρο αναζήτησης. Συγκεκριμένα, ο νέος κόμβος εισάγεται σαν να επρόκειτο για κοινό δυαδικό δέντρο αναζήτησης, και στη συνέχεια χρωματίζεται ερυθρός. Κατόπιν καλείται μια διαδικασία επιδιόρθωσης, η οποία εκτελεί κατάλληλες περιστροφές και επαναχρωματίζει κόμβους ώστε να εξασφαλιστεί η διατήρηση των ιδιοτήτων του μελανέρυθρου δέντρου.

RB-INSERT*(T, z)*

```
1.  y ← nil[T ]
2.  x ← root[T ]
3.  while x ≠ nil[T ]
4.      do y ← x
5.      if key[z] < key[x]
6.          then x ← left[x]
7.          else x ← right[x]
```

```
8.  p[z] ← y
9.  if y = nil[T ]
10.     then root[T ] ← z
11.     else if key[z] < key[y]
12.         then left[y] ← z
13.         else right[y]← z
14. left[z]← nil[T ]
15. right[z]← nil[T ]
16. color[z]← RED
17. RB-INSERT-FIXUP(T, z)
```

RB-INSERT-FIXUP*(T, z)*

```
1.  while color[p[z]] = RED
2.      do if p[z] = left[p[p[z]]]
3.          then y ← right[p[p[z]]]
4.              if color[y] = RED
5.                  then color[p[z]] ← BLACK
6.                      color[y] ← BLACK
7.                      color[p[p[z]]] ← RED
8.                      z ← p[p[z]]
9.                  else if z = right[p[z]]
10.                     then z ← p[z]
11.                         LEFT-ROTATE(T, z)
12.                     color[p[z]] ← BLACK
13.                     color[p[p[z]]] ← RED
14.                     RIGHT-ROTATE(T, p[p[z]])
15.         else y ← left[p[p[z]]]
16.             if color[y] = RED
17.                 then color[p[z]] ← BLACK
18.                     color[y] ← BLACK
19.                     color[p[p[z]]] ← RED
20.                     z ← p[p[z]]
21.                 else if z = left[p[z]]
22.                     then z ← p[z]
23.                         RIGHT-ROTATE(T, z)
24.                     color[p[z]] ← BLACK
25.                     color[p[p[z]]] ← RED
26.                     LEFT-ROTATE(T, p[p[z]])
27. color[root[T ]]← BLACK
```

4.3.3 Διαγραφή

Η διαγραφή κόμβου από ένα μελανέρυθρο δέντρο είναι πιο περίπλοκη από την εισαγωγή. Για την ακρίβεια, αφού η διαδικασία της διαγραφής αποκόψει από το δέντρο τον κόμβο προς διαγραφή, καλεί κι αυτή μια βοηθητική διαδικασία επιδιόρθωσης η οποία εκτελεί περιστροφές και επαναχρωματίζει κόμβους προκειμένου να διατηρηθούν οι ιδιότητες του μελανέρυθρου δέντρου.

RB-DELETE*(T, z)*

```
1.  if left[z] = nil[T] or right[z] = nil[T ]
2.      then y ← z
3.      else y ← TREE-SUCCESSOR(z)
4.  if left[y] ≠ nil[T ]
5.      then x ← left[y]
6.      else x ← right[y]
7.  p[x] ← p[y]
8.  if p[y] = nil[T ]
9.      then root[T ] ← x
10.     else if y = left[p[y]]
11.         then left[p[y]] ← x
12.         else right[p[y]] ← x
13. if y ≠ z
14.     then key[z]← key[y]
15.         copy y's satellite data into z
16. if color[y] = BLACK
17.     then RB-DELETE-FIXUP(T, x)
18. return y
```

και

RB-DELETE-FIXUP*(T, x)*

```
1.  while x ≠ root[T] and color[x] = BLACK
2.      do if x = left[p[x]]
3.          then w ← right[p[x]]
4.              if color[w] = RED
5.                  then color[w] ← BLACK
6.                      color[p[x]] ← RED
7.                      LEFT-ROTATE(T, p[x])
8.                      w ← right[p[x]]
9.              if color[left[w]] = BLACK and color[right[w]] = BLACK
10.                 then color[w] ← RED
11.                     x ← p[x]
12.                 else if color[right[w]] = BLACK
13.                     then color[left[w]] ← BLACK
14.                         color[w]← RED
15.                         RIGHT-ROTATE(T,w)
16.                         w ← right[p[x]]
17.                     color[w] ← color[p[x]]
18.                     color[p[x]] ← BLACK
19.                     color[right[w]] ← BLACK
20.                     LEFT-ROTATE(T, p[x])
21.                     x ← root[T ]
22.         else w ← left[p[x]]
23.             if color[w] = RED
24.                 then color[w] ← BLACK
25.                     color[p[x]] ← RED
26.                     RIGHT-ROTATE(T, p[x])
27.                     w ← left[p[x]]
28.             if color[right[w]] = BLACK and color[left[w]] = BLACK
29.                 then color[w] ← RED
30.                     x ← p[x]
31.                 else if color[left[w]] = BLACK
32.                     then color[right[w]] ← BLACK
```

```
                        color[w]← RED
                        LEFT-ROTATE(T,w)
                        w ← left[p[x]]
                    color[w] ← color[p[x]]
                    color[p[x]] ← BLACK
                    color[left[w]] ← BLACK
                    RIGHT-ROTATE(T, p[x])
                    x ← root[T]
color[x] ← BLACK
```

Κεφάλαιο 5ο

Δέντρα Διατακτικών Στατιστικών

5.1 Τι είναι οι διατακτικές στατιστικές

Η $i-οστή$ διατακτική στατιστική ενός συνόλου n στοιχείων είναι το $i-οστό$ μικρότερο στοιχείο του συνόλου αυτού. Για παράδειγμα, το ελάχιστο σε ένα σύνολο στοιχείων είναι η πρώτη διατακτική στατιστική $(i=1)$ και το μέγιστο είναι η τελευταία διατακτική στατιστική $(i=n)$.

5.2 Τι είναι τα δέντρα διατακτικών στατιστικών

Ένα δέντρο διατακτικών στατιστικών είναι ένα επαυξημένο μελανέρυθρο δέντρο. Ο κάθε κόμβος του δέντρου αυτού, πέρα από τα συνηθισμένα πεδία (κλειδί, παρελκόμενα δεδομένα, γονιός, αριστερός θυγατρικός κόμβος, δεξιός θυγατρικός κόμβος, και χρώμα), διαθέτει κι ένα επιπλέον πεδίο, $μέγεθος[x]$, το οποίο περιέχει το πλήθος των κόμβων του υποδέντρου που εκφύεται από τον κόμβο x, δηλαδή δίνει το μέγεθος του υποδέντρου.

Στα δέντρα διατακτικών στατιστικών, τα κλειδιά των κόμβων δε χρειάζεται να είναι διαφορετικά μεταξύ τους.

5.3 Πράξεις

5.3.1 Ανάκτηση στοιχείου δεδομένης τάξης

Η ανάκτηση στοιχείου δεδομένης τάξης από ένα δέντρο διατακτικών στατιστικών γίνεται με τη διαδικασία $Επιλογή(x,i)$, η οποία επιστρέφει έναν δείκτη προς τον κόμβο που περιέχει το $i-οστό$ μικρότερο κλειδί, στο υποδέντρο που ορίζει το x.

OS-SELECT*(x, i)*

```
1. r ← size[left[x]]+1
2. if i = r
3.    then return x
4. elseif i < r
5.    then return OS-SELECT(left[x], i )
6. else return OS-SELECT(right[x], i − r)
```

5.3.2 Προσδιορισμός τάξης ενός κόμβου

Αυτή η πράξη δέχεται ως όρισμα έναν δείκτη προς κάποιον κόμβο x ενός δέντρου, και επιστρέφει τη θέση του x στη γραμμική διάταξη που καθορίζει η συμμετρική διάνυση του δέντρου T.

OS-RANK*(T, x)*

```
1. r ← size[left[x]] + 1
2. y ← x
3. while y = root[T ]
4.    do if y = right[p[y]]
5.       then r ← r + size[left[p[y]]] + 1
6.    y ← p[y]
7. return r
```

5.4 Υλοποίηση

Για την υλοποίηση του δέντρου, στηριχθήκαμε σε ένα Μελανέρυθρο δέντρο. Η υλοποίηση έγινε με σκοπό να έχουμε ένα ΑΤΔ (Αφαιρετικό Τύπο Δεδομένων) που να επιτρέπει χρήση με διαφορετικούς τύπους δεδομένων στο ίδιο πρόγραμμα.

Η πρώτη αλλαγή που κάναμε ήταν να χωρίσουμε το interface από την υλοποίηση. Δημιουργήσαμε 2 κεφαλίδες στις οποίες περιέχονται οι δηλώσεις των συναρτήσεων και της δομής που μπορεί να χρησιμοποιήσει ο χρήστης. Ο ορισμός των συναρτήσεων και της δομής γίνεται στο αντίστοιχο .c αρχείο όπου ο χρήστης δεν έχει πρόσβαση (Πλήρης Απόκρυψη). Με την πλήρη απόκρυψη δεν

επιτρέπουμε στο χρήστη να αλλάξει τον τρόπο με τον οποίο λειτουργούν οι συναρτήσεις. Ακόμα πετυχαίνουμε ότι μπορούμε απλώς να αλλάξουμε τον τρόπο υλοποίησης των συναρτήσεων χωρίς να επηρεαστεί ο χρήστης από αυτή την αλλαγή (λόγω του ότι δε θα αλλάξει τίποτα στο interface του).

Η πρώτη κεφαλίδα “OSTree.h” περιέχει τις δηλώσεις των τριών συναρτήσεων που χρειάζονται στον χρήστη για τη λειτουργία του δέντρου (insert, delete, print). Το αντίστοιχο αρχείο “OSTree.c” περιέχει έναν μεγαλύτερο αριθμό συναρτήσεων που βοηθούν στην υλοποίηση των τριών λειτουργιών στις οποίες έχει πρόσβαση ο χρήστης. Οι βοηθητικές συναρτήσεις δεν είναι ορατές από τον χρήστη.

Η δεύτερη κεφαλίδα “inputcheck.h” περιέχει τις δηλώσεις 2 συναρτήσεων ελέγχου λαθών εισαγωγής στοιχείων (τύπου integer και character). Η υλοποίηση τους υπάρχει στο αντίστοιχο αρχείο “inputcheck.c” στο οποίο και πάλι δεν έχει πρόσβαση ο χρήστης. Ο έλεγχος για τα δεδομένα τύπου float και double έγινε με ενσωμάτωση του κώδικα μέσα στο block όπου παίρνονται τα δεδομένα. Επιλέχθηκε αυτή η «ανορθόδοξη» προσέγγιση (δύο διαφορετικές μέθοδοι για την ίδια λειτουργία) για να δειχθεί η διαφορά ανάμεσα σ’ αυτές τις δύο μεθόδους. Παρατηρούμε δύο πλεονεκτήματα στην περίπτωση που χρησιμοποιούμε τη συνάρτηση για τον έλεγχο. Πρώτον ο κώδικας γίνεται αρκετά πιο ευανάγνωστος και δεύτερον, στην περίπτωση που ζητάμε σε αρκετά διαφορετικά σημεία input από τον χρήστη, δε χρειάζεται να γράφουμε ξανά και ξανά σε αυτά τα διάφορα σημεία του προγράμματος τον ίδιο κώδικα που πραγματοποιεί τον έλεγχο.

Με τη χρήση των δύο παραπάνω βιβλιοθηκών στο αρχείο “OrderStatisticTree.c” καταφέραμε να έχουμε μόνο την main η οποία ουσιαστικά περιέχει μόνο το μενού επιλογών.

Τα δύο modules που φτιάξαμε είναι συνεπή ως προς τα ονόματα των συναρτήσεων (κάθε συνάρτηση αρχίζει με “OST_” για το ένα module και με “input” για το άλλο). Παρουσιάζουν ισχυρή συνάφεια μιας και όλες οι συναρτήσεις σχετίζονται άμεσα μεταξύ τους και με τα encapsulated data.

Ακόμα με τη χρήση των “Super-insert, Super-delete” πετυχαίνουμε μια ασθενή σύζευξη (Weak Coupling) μιας και ο χρήστης μπορεί να καλέσει πολύ λίγες συναρτήσεις από το module μου οπότε κατ’ επέκταση θα κάνει και λίγες κλήσεις στο module μου. (αυτό είναι άλλο ένα μεγάλο πλεονέκτημα της πλήρους απόκρυψης).

Η δεύτερη αλλαγή που κάναμε ήταν η τροποποίηση του κώδικα έτσι ώστε να δέχεται multiple types of data. Ο χρήστης έχει την επιλογή του να κατασκευάσει ένα δέντρο με διάφορους τύπους στοιχείων. όπως int, float, double, char. Αυτό το πετύχαμε με τη χρήση void pointers μιας και μπορούμε να κάνουμε assign οποιονδήποτε pointer σε void pointer και αντιστρόφως. Το πρόβλημα που προκύπτει είναι ότι ο χρήστης θα πρέπει να ξέρει σε ποιον τύπο δεδομένων δείχνει ο void pointer. Στη δική μας εργασία δεν υπάρχει τέτοιο θέμα μιας και ο χρήστης επιλέγει στην αρχή του προγράμματος τα δεδομένα εισόδου και εμείς κρατώντας την επιλογή του αποφασίζουμε σε τι τύπο δεδομένων δείχνει ο void pointer.

Άλλο ένα πρόβλημα που προκύπτει με αυτή την αλλαγή είναι ότι οι τελεστές σύγκρισης, αν μείνουν όπως είναι, δε θα δουλέψουν. Οπότε θα πρέπει να αντικατασταθούν από κάποιες συναρτήσεις οι οποίες θα επιστρέφουν μία τιμή που θα δείχνει αν είναι αληθείς ή ψευδείς.

Κατά τη συγγραφή του προγράμματος γινόταν ταυτόχρονα και ο έλεγχος για κάθε τμήμα που ετοιμαζόταν. Με τον τρόπο αυτό η διόρθωση των εκάστοτε λαθών ήταν πολύ πιο εύκολη απ’ ότι να ετοιμαζόταν όλο το πρόγραμμα και ελεγχόταν στο τέλος. Για τον έλεγχο χρησιμοποιήθηκαν διάφορες printf, για να διαπιστώνεται το που μπήκε σε κάποια συνάρτηση το πρόγραμμα όταν έτρεχε και από πού βγήκε. Ακόμα χρησιμοποιήθηκαν κατάλληλα δεδομένα εισόδου για να ελεγχθεί η ορθή λειτουργία της κάθε συνάρτησης.

ΠΑΡΑΡΤΗΜΑ

Εδώ παρατίθεται ο κώδικας

OrderStatisticTree.c

```c
#include"OSTree.h"
#include"inputcheck.h"
#include<stdio.h>
#include<stdlib.h>


int main(void)
{
    int datatype = 0;
    char c = '*';

    while (datatype < 1 || datatype > 4)
    {
        printf("\n(1)Integer\n(2)Float\n(3)Double\n(4)Character");
        printf("\nSelect data type for the tree: ");
        scanf("%d", &datatype);
        getchar();        //flush buffer
    }

    while (c != 'q')
    {
        printf("\n\nPrevious choice was:%c\n\n", c);
        printf("Your options are: (i)nsert, (d)elete, (p)rint, (q)uit\nCHOICE: ");
        c = getchar();
        getchar();            //flush buffer
        switch (c)
        {
        case 'i':
            OST_Super_Insert(datatype);
            break;
        case 'd':
            OST_Super_Delete(datatype);
            break;
        case 'p':
            OST_Super_Print_InOrder(datatype);
            break;
        case 'r':
            OST_Super_kth (int datatype);
            break;
        case 'o':
            OST_cnt(int x);
            break;
        case 'q':
            return 0;
        default:
            printf("\n\nINVALID CHOICE!!!");
        }
    }
}
```

Inputcheck.h

```h
#ifndef check_h
#define check_h

int inputint();
char inputchar();

#endif
```

Inputcheck.c

```c
#include"inputcheck.h"
#include<stdio.h>


int inputint() //Διάβασμα και επαλήθευση ενός ακεραίου από τον χρήστη
{
    int a1, a2, r;

    a2 = scanf("%d", &a1); getchar();
    while (a2 != 1)
    {
        printf("prepei na einai ari8mos\n");
        a2 = scanf(" %d", &a1);
        while ((r = getchar()) != '\n');
    }
    return a1;
}


char inputchar() //Διάβασμα και επαλήθευση ενός character από τον χρήστη
{
    int a2, r; char a1;

    a2 = scanf("%c", &a1); getchar();
    while (a2 != 1)
    {
        printf("prepei na einai character\n");
        a2 = scanf(" %c", &a1);
        while ((r = getchar()) != '\n');
    }
    return a1;
}
```

OSTree.h

```h
#ifndef OSTree_h
#define OSTree_h

void OST_Super_Insert(int type);
void OST_Super_Delete(int type);
```

```
6.  void OST_Super_Print_InOrder(int type);
7.  void OST_cnt(int x);
8.  void OST_Super_kth (int K);
9.  #endif
```

OSTree.c

```
1.  #include "OSTree.h"
2.  #include "inputcheck.h"
3.  #include <stdio.h>
4.  #include <stdlib.h>
5.
6.  struct OSTnode
7.  {
8.   void *key;
9.   int color;//0=red 1=black
10.  int count=1;
11.  struct OSTnode *lchild,*rchild,*parent;
12. }*root;
13.
14.
15. int compare (void *x,void *y,int type)
16. {
17.     if (type==1)
18.     {
19.         if ((*(int*)x)<(*(int*)y))
20.          return 1;
21.       if ((*(int*)x)>(*(int*)y))
22.          return 0;
23.      }
24.      if (type==2)
25.      {
26.         if ((*(float*)x)<(*(float*)y))
27.          return 1;
28.       if ((*(float*)x)>(*(float*)y))
29.          return 0;
30.      }
31.      if (type==3)
32.      {
33.         if ((*(double*)x)<(*(double*)y))
34.          return 1;
35.       if ((*(double*)x)>(*(double*)y))
36.          return 0;
37.      }
38.      if (type==4)
39.      {
40.         if ((*(char*)x)<(*(char*)y))
41.          return 1;
42.       if ((*(char*)x)>(*(char*)y))
43.          return 0;
44.      }
45.  }
46.
47.
48. int equality (void *x,void *y,int type)
```

```
{
    if (type==1)
    {
        if ((*(int*)x)==(*(int*)y))
         return 1;
      else
         return 0;
     }
     if (type==2)
     {
        if ((*(float*)x)==(*(float*)y))
         return 1;
      else
         return 0;
     }
     if (type==3)
     {
        if ((*(double*)x)==(*(double*)y))
         return 1;
      else
         return 0;
     }
     if (type==4)
     {
        if ((*(char*)x)==(*(char*)y))
         return 1;
      else
         return 0;
     }
 }


struct OSTnode *OST_Search (void *i, struct OSTnode *r,int type)
{
    if (r == NULL || equality(r->key, i, type) == 1)
        return r;
    if (compare(r->key, i, type) == 1)
        r = r->rchild;
    else
        r = r->lchild;
    return OST_Search(i, r, type);
}



void OST_SetParent(struct OSTnode *r, struct OSTnode *r1)
{
    if (r != NULL) r->parent = r1;
}


void OST_Right_Rotate(struct OSTnode *x )
{
    struct OSTnode *y;
    y = x->lchild;
    x->lchild = y->rchild;
    OST_SetParent(y->rchild, x);
    OST_SetParent(y, x->parent);
    if (x->parent == NULL)
        root = y;
```

```
110.    else
111.    {
112.        if (x == x->parent->rchild)
113.            x->parent->rchild = y;
114.        else
115.            x->parent->lchild = y;
116.    }
117.    y->rchild = x;
118.    OST_SetParent(x, y);
119.    y->count = x->count;
120.    x->count = x->left->count + x->right->count +1;
121.}
122.
123.
124.void OST_Left_Rotate(struct OSTnode *x )
125.{
126.    struct OSTnode *y;
127.    y = x->rchild;
128.    x->rchild = y->lchild;
129.    OST_SetParent(y->lchild, x);
130.    OST_SetParent(y, x->parent);
131.    if (x->parent == NULL)
132.        root = y;
133.    else
134.    {
135.        if (x == x->parent->lchild)
136.            x->parent->lchild = y;
137.        else
138.            x->parent->rchild = y;
139.    }
140.    y->lchild = x;
141.    OST_SetParent(x, y);
142.    y->count = x->count;
143.    x->count = x->left->count + x->right->count +1;
144.}
145.
146.
147.
148.struct OSTnode *OST_Minimum(struct OSTnode *r)
149.{
150.    while (r->lchild != NULL)
151.        r = r->lchild;
152.    return r;
153.}
154.
155.
156.struct OSTnode *OST_Successor(struct OSTnode *r)
157.{
158.    struct OSTnode *y = r->parent;
159.
160.    if (r->rchild != NULL)
161.        return(OST_Minimum(r->rchild));
162.
163.    while (y != NULL && r == y->rchild)
164.    {
165.        r = y;
166.        y = y->parent;
167.    }
168.    if (y == NULL)
169.        return r;
170.    else
```

```
171.         return y;
172.}
173.
174.
175.void OST_Print(struct OSTnode *r,int type )
176. {
177.     struct OSTnode *z;
178.     if (type == 1)
179.     {
180.         if (r->lchild != NULL)
181.             OST_Print(r->lchild, type);
182.         printf("\n%d    ", *(int*)(r->key));
183.         if (r->color == 0)
184.             printf("Red");
185.         else
186.             printf("Black");
187.         if (r->parent != NULL)
188.             printf("   Parent: %d", *(int*)(r->parent->key));
189.         if (r->rchild != NULL)
190.             OST_Print(r->rchild, type);
191.        }
192.     if (type == 2)
193.     {
194.         if (r->lchild != NULL)
195.             OST_Print(r->lchild, type);
196.         printf("\n%f    ", *(float*)(r->key));
197.         if (r->color == 0)
198.             printf("Red");
199.         else
200.             printf("Black");
201.         if (r->parent != NULL)
202.             printf("   Parent: %f", *(float*)(r->parent->key));
203.         if (r->rchild != NULL)
204.             OST_Print(r->rchild, type);
205.     }
206.     if (type == 3)
207.     {
208.         if (r->lchild != NULL)
209.             OST_Print(r->lchild, type);
210.         printf("\n%lf    ", *(double*)(r->key));
211.         if (r->color == 0)
212.             printf("Red");
213.         else
214.             printf("Black");
215.         if (r->parent != NULL)
216.             printf("   Parent: %lf", *(double*)(r->parent->key));
217.         if (r->rchild != NULL)
218.             OST_Print(r->rchild, type);
219.     }
220.
221.     if (type == 4)
222.     {
223.         if (r->lchild != NULL)
224.             OST_Print(r->lchild, type);
225.         printf("\n%c    ", *(char*)(r->key));
226.         if (r->color == 0)
227.             printf("Red");
228.         else
229.             printf("Black");
230.         if (r->parent != NULL)
231.             printf("   Parent: %c", *(char*)(r->parent->key));
```

```
232.         if (r->rchild != NULL)
233.             OST_Print(r->rchild, type);
234.     }
235.
236.}
237.
238.
239.void OST_Insert(struct OSTnode *z,int type)
240.{
241.    struct OSTnode *x, *y;
242.
243.    y = NULL;
244.    x = root;
245.
246.    while (x != NULL)
247.    {
248.        y = x;
249.        y->count++;
250.        if (compare(z->key, x->key, type) == 1)
251.            x = x->lchild;
252.        else
253.            x = x->rchild;
254.    }
255.    z->parent = y;
256.    if (y == NULL)
257.        root = z;
258.    else
259.    {
260.        z->parent = y;
261.        if (compare(z->key, y->key, type) == 1)
262.            y->lchild = z;
263.        else
264.            y->rchild = z;
265.    }
266.}
267.
268.
269.void OST_Insert_Fix(struct OSTnode *x,int type)
270.{
271.     struct OSTnode *y;
272.
273.     OST_Insert(x,type);
274.
275.     while (x->parent != NULL && x->parent->color == 0)
276.     {
277.         if (x->parent == x->parent->parent->lchild)
278.         {
279.             y = x->parent->parent->rchild;
280.             if (y != NULL && y->color == 0)
281.             {
282.                 x->parent->color = 1;
283.                 y->color = 1;
284.                 x->parent->parent->color = 0;
285.                 x = x->parent->parent;
286.                 inorder(x,1)
287.             }
288.             else
289.             {
290.                 if (x == x->parent->rchild)
291.                 {
292.                     x = x->parent;
```

```
                    OST_Left_Rotate(x);
                }
            x->parent->color = 1;
            x->parent->parent->color = 0;
            OST_Right_Rotate(x->parent->parent);
            }
        }
        else
        {
            y = x->parent->parent->lchild;
            if (y != NULL && y->color == 0)
            {
                x->parent->color = 1;
                y->color = 1;
                x->parent->parent->color = 0;
                x = x->parent->parent;
            }
            else
            {
                if (x == x->parent->lchild)
                {
                    x = x->parent;
                    OST_Right_Rotate(x);
                }
                x->parent->color = 1;
                x->parent->parent->color = 0;
                OST_Left_Rotate(x->parent->parent);
            }
        }
    }
    root->color = 1;
    root->order = 1;
}


void OST_Delete_Fix(struct OSTnode *x)
{
    struct OSTnode *w;

    while (x != NULL && x->color == 1)
    {
        if (x == x->parent->lchild)
        {
            w = x->parent->rchild;
            if (w != NULL);
            {
                if (w->color == 0)
                {
                    w->color = 1; x->parent->color = 0;
                    OST_Left_Rotate(x->parent);
                    w = x->parent->rchild;
                }
                if (w->lchild->color == 1 && w->rchild->color == 1)
                {
                    w->color = 0; x = x->parent;
                }
                else if (w->rchild->color == 1)
                {
                    w->lchild->color = 1;
                    w->color = 0;
                    OST_Right_Rotate(w);
```

```
                    w = x->parent->rchild;
                }
                w->color = x->parent->color;
                x->parent->color = 1;
                w->rchild->color = 1;
            }
            OST_Left_Rotate(x->parent);
            x = root;
        }
        else
        {
            w = x->parent->lchild;
            if (w != NULL)
            {
                if (w->color == 0)
                {
                    w->color = 1;
                    x->parent->color = 0;
                    OST_Right_Rotate((x->parent));
                    w = x->parent->lchild;
                }
                if (w->rchild->color == 1 && w->lchild->color == 1)
                {
                    w->color = 0; x = x->parent;
                }
                else if (w->lchild->color == 1)
                {
                    w->rchild->color = 1;
                    w->color = 0;
                    OST_Left_Rotate(w);
                    w = x->parent->lchild;
                }
                w->color = x->parent->color;
                x->parent->color = 1;
                w->lchild->color = 1;
            }
            OST_Right_Rotate(x->parent);
            x = root;
        }
        x->color = 1;
    }
}


void OST_Delete (struct OSTnode *z)
{
    struct OSTnode *y, *x;

    y = z;

    if (root == NULL)
    {
        printf("\n Red-Black Tree is empty \n");
        return;
    }
    if (z == NULL)
        printf("\n\n\tThere is no Node with the key you gave to Delete.\n");
    else
    {
        if (z->lchild == NULL || z->rchild == NULL)
            y = z;
```

```
415.        else
416.            y = OST_Successor(z);
417.        if (y->lchild != NULL)
418.        {
419.            x = y->lchild;
420.        }
421.        else
422.        {
423.            x = y->rchild;
424.        }
425.        OST_SetParent(x, y->parent);
426.        if (y->parent == NULL)
427.            root = x;
428.        else if (y == y->parent->lchild)
429.            y->parent->lchild = x;
430.        else
431.        {
432.            y->parent->rchild = x;
433.        }
434.        if (y != z)
435.            z->key = y->key;
436.        if (y->color == 1)
437.            OST_Delete_Fix(x);
438.        printf("\n\nThe Node was deleted\n");
439.    }
440.}
441.
442.
443.
444.void OST_Super_Insert(int type)
445.{
446.    struct OSTnode *temp;
447.    int r, da1;
448.    float da2;
449.    double da3;
450.
451.    temp = (struct OSTnode *)malloc(sizeof(struct OSTnode));
452.    temp->color = 0;
453.    temp->lchild = NULL;
454.    temp->rchild = NULL;
455.    temp->parent = NULL;
456.    temp->count = 1;
457.    printf("\nGive key: ");
458.
459.    if (type == 1)
460.    {
461.        temp->key = (int*)malloc(sizeof(int));
462.        *(int*)(temp->key) = inputint();
463.    }
464.
465.    if (type == 2)
466.    {
467.        da1 = scanf("%f", &da2);
468.        getchar();
469.        while (da1 != 1)
470.        {
471.            printf("prepei na einai ari8mos\n");
472.            da1 = scanf(" %f", &da2);
473.            while((r = getchar()) != '\n');
474.        }
475.        temp->key = (float*)malloc(sizeof(float));
```

```
476.         *(float*)(temp->key) = da2;
477.     }
478.
479.     if (type == 3)
480. {
481.     da1 = scanf("%lf", &da3);
482.     getchar();
483.     while (da1 != 1)
484.     {
485.         printf("prepei na einai ari8mos\n");
486.         da1 = scanf(" %lf", &da3);
487.         while ((r = getchar()) != '\n');
488.     }
489.     temp->key = (double*)malloc(sizeof(double));
490.     *(double*)(temp->key) = da3;
491.     }
492.
493.     if (type == 4)
494.    {
495.        temp->key = (char*)malloc(sizeof(char));
496.        *(char*)(temp->key) = inputchar();
497.     }
498.
499.     OST_Insert_Fix(temp,type);
500. }
501.
502.
503.
504. void OST_Super_Delete(int type)
505. {
506.     int r, da1;
507.     float da2;
508.     double da3;
509.     char da4;
510.     void *temp = NULL;
511.
512.     printf("\n Give the key you want to delete: ");
513.
514.
515.     if (type == 1)
516.     {
517.         temp = (int*)malloc(sizeof(int));
518.         *(int*)(temp) = inputint();
519.     }
520.
521.     if (type == 2)
522.     {
523.         da1 = scanf("%f", &da2);
524.         getchar();
525.         while (da1 != 1)
526.         {
527.             printf("prepei na einai ari8mos\n");
528.             da1 = scanf(" %f", &da2);
529.             while ((r = getchar()) != '\n');
530.         }
531.         temp = (float*)malloc(sizeof(float));
532.         *(float*)(temp) = da2;
533.     }
534.
535.     if (type == 3)
536.     {
```

```
537.         da1 = scanf("%lf", &da3); getchar();
538.         while (da1 != 1)
539.         {
540.             printf("prepei na einai ari8mos\n");
541.             da1 = scanf(" %lf", &da3);
542.             while ((r = getchar()) != '\n');
543.         }
544.         temp = (double*)malloc(sizeof(double));
545.         *(double*)(temp) = da3;
546.     }
547.
548.     if (type == 3)
549.     {
550.         temp = (char*)malloc(sizeof(char));
551.         *(char*)(temp) = inputchar();
552.     }
553.
554.     OST_Delete(OST_Search(temp, root, type));
555. }
556.
557. void OST_Super_Print_InOrder(int type)
558. {
559.     if (root == NULL)
560.     {
561.         printf("\n Red-Black Tree is empty\n\n");
562.     }
563.     else
564.         OST_Print(root, type);
565. }
566.
567. void OST_Super_kth (int K)
568. {
569.         int crank = root->lchild->count + 1;
570.         struct OSTnode y = root;
571.     while (y != NULL && crank != K) {
572.         if (K < crank)
573.             y = y->lchild;
574.         else {
575.             K = K - crank;
576.             y = y->rchild;
577.         }
578.         if (y == NULL)
579.             return 0;
580.         crank = y->lchild->count + 1;
581.     }
582.     return y;
583. }
584.
585. void OST_cnt(int x) {
586.     int ans = 0;
587.     struct OSTnode y = root;
588.     while (y != NULL) {
589.         if (y->key > x)
590.             y = y->lchild;
591.         else if (y->key < x) {
592.             ans += y->lchild->count + 1;
593.             y = y->rchild;
594.         }
595.         else
596.             return ans + y->left->count;
597.     }
```

```
598.     return ans;
599. }
```

Αναφορές

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, (2009). Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill

[2] Robert Sedgewick, Kevin Wayne (2011). Algorithms (4th ed.). Pearson Education

[3] Sedgewick, Robert (1998). Algorithms in C++. Addison-Wesley Professional

[4] Charles Leiserson, and Erik Demaine. *6.046J Introduction to Algorithms (SMA 5503).* Fall 2005. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.