Δομές Δεδομένων (Εργαστηριακή Άσκηση)

Ονοματεπώνυμο: Σπύρος Καφτάνης
Αριθμός Μητρώου: 23 5542
E-Mail: kaftanis@ceid.upatras.gr
Γλώσσα Προγραμματισμού: Java

Θέμα 1: Ταξινόμηση - Αναζήτηση

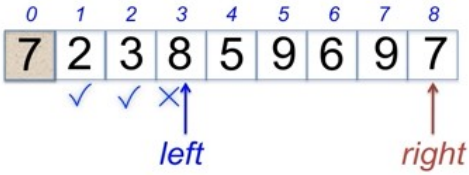
Στο θέμα αυτό γίνεται υλοποίηση κάποιων αλγορίθμων ταξινόμησης και αναζητητής. Η κλάση ελέγχου των αλγορίθμων αυτών βρίσκεται στο αρχείο MainClass.java. Εκεί εμφανίζεται το μενού με όλες τις δυνατές επιλογές για τη χρήση των πέντε αυτών αλγορίθμων.

Quick Sort περιγραφή και υλοποίηση

Ο πρώτος αλγόριθμος που υλοποιήθηκε είναι ο αλγόριθμος ταξινόμησης Quick Sort. Πρόκειται για έναν αλγόριθμο που χρησιμοποιεί τη τεχνική Διαίρει και Βασίλευε. Αρχικά επιλέγεται ένας τυχαίος οδηγός (pivot) (στη συγκεκριμένη υλοποίηση επιλέγεται πάντα το μεσαίο στοιχείο) το οποίο χωρίζει τον πίνακα σε δύο μέρη, ένα με τα μικρότερα του pivot και ένα με τα μεγαλύτερα. Η διαδικασία αυτή εκτελείται αναδρομικά για κάθε κομμάτι του πίνακα.

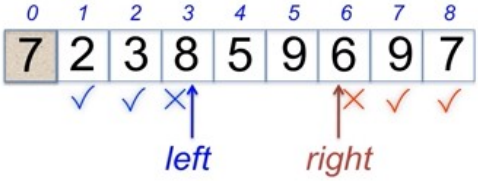
Η υλοποίηση του αλγορίθμου βρίσκεται στο αρχείο **QuickSort.java** και για την εκτέλεσή της αρκεί να καλέσετε τη μέθοδο sort, δίνοντας σαν όρισμα ένα πίνακα ακεραίων.

Όσον αφορά την **υλοποίηση** υπάρχει αρχικά η μέθοδος split. Αφού οριστεί σαν pivot το πρώτο στοιχείο του πίνακα, σαν right δείκτης το low+1 και σαν right το hight ξεκινάει η διαδικασία. Όσο ο δείκτης left είναι μικρότερος του right ο πρώτος αυξάνει κατά ένα μέχρι να βρει κάποιο στοιχείου που δεν είναι μικρότερο του pivot οπότε και σταματάει.




```
while ( left <= right ) {
    if (array[left] < pivot  )
        left ++;
    else
        break;
    compares++;
}
```

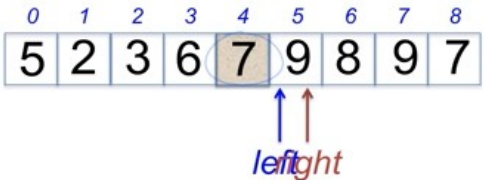
Τότε ο δείκτης right αρχίζει να μειώνεται μέχρι να βρεθεί κάποια τιμή μικρότερη του pivot.




```
while ( right > left ) {
    if ( array[right] < pivot )
        break;
    else
        right --;
    compares++;
}
```

Τα στοιχεία τώρα που βρίσκονται στις θέσεις που δείχνει το left και ο right αλλάζουν θέση και οι δείκτες προχωρούν άλλη μία θέση.

Η διαδικασία συνεχίζεται μέχρι ο δείκτης left να γίνει μεγαλύτερος ή ίσος του right οπότε η διαδικασία σταματά, το στοιχείου που βρίσκεται μία θέση πριν τον right αλλάζει θέση με το pivot και η μέθοδος επιστρέφει το splitpoint, το οποίο χωρίζει τα μεγαλύτερα από τα μικρότερα στοιχεία του στοιχείου που δείχνει (pivot).



Η private sort τώρα αφού πάρει σαν όρισμα τον πίνακα, το low και το hight καλεί την split ώστε να βρει το splitPoint αλλά και να κάνει τις κατάλληλες αλλαγές στον πίνακα και έπειτα τον εαυτό της για το κομμάτι που πίνακα μικρότερο του splitPoint και για το κομμάτι μεγαλύτερο αυτού.

```
int splitPoint = split (array,low,high);
sort(array, low, splitPoint-1);
sort (array, splitPoint+1, high);
```

Merge Sort περιγραφή και υλοποίηση

Ο αλγόριθμος αυτός βασίζεται επίσης στην τεχνική διαίρει και βασίλευε έχοντας μια πιο απλή λογική σε σχέση με τον QuickSort. Αρχικά ο πίνακας που δίνεται σαν είσοδος χωρίζεται σε δύο "ίσα" μέρη, τα οποία ταξινομούνται το καθένα ξεχωριστά. Η ταξινόμηση δε γίνεται με κάποιον άλλου τύπου αλγόριθμο, αλλά με αναδρομικές κλήσεις της ίδιας μεθόδου για τα τμήματα 0-n/2 -1 και n/2 -n.
Τέλος οι δύο ταξινομημένοι πίνακες που προκύπτουν ενώνονται σε έναν ταξινομημένο πίνακα.

Η υλοποίηση του αλγορίθμου βρίσκεται το αρχείο **MergeSort.java** και για την εκτέλεσή της αρκεί να καλέσετε τη μέθοδο sort, δίνοντας σαν όρισμα ένα πίνακα ακεραίων.

Η μέθοδος **merge** είναι η μέθοδος που ενώνει τους δύο μικρούς ταξινομημένους πίνακες σε έναν μεγάλο. Χρησιμοποιώντας ένα δείκτη για κάθε ένα από τους τρεις πίνακες γίνεται έλεγχος των αντίστοιχων στοιχείων των μικρών πινάκων

και το μικρότερο μπαίνει στον τελικό πίνακα.

Η public μέθοδος **sort**, βρίσκει το μέσο, χωρίζει τον πίνακα εισόδου σε δύο μικρούς πίνακες και καλεί αναδρομικά τον εαυτό της για αυτούς τους πίνακες. Τέλος καλεί την merge για τη συγχώνευση.

Insertion Sort περιγραφή και υλοποίηση

Ο αλγόριθμος αυτός, σε αντίθεση με τους προηγούμενους, δε χρησιμοποιεί αναδρομικές κλήσεις ούτε κάποια τεχνική διαίρει και βασίλευε. Η λειτουργία του είναι απλή: Σκανάρουμε τον πίνακα σειριακά κάνοντας την υπόθεση ότι τα προηγούμενα στοιχεία είναι ταξινομημένα. Έτσι ξεκινώντας από το δεύτερο στοιχείο έχουμε υποθέσει μια ταξινομημένη λίστα ενός στοιχείου (του πρώτου). Προχωρώντας στη λίστα ελέγχουμε αν το επιλεγμένο στοιχείο είναι μικρότερο από το προηγούμενο. Αν είναι γίνεται η αντιμετάθεση μέχρι να βρεθεί κάποιο μεγαλύτερο στοιχείο ή μέχρι να φτάσει στη πρώτη θέση της λίστας.

Η υλοποίηση του αλγορίθμου βρίσκεται το αρχείο **InsertionSort.java** και για την εκτέλεσή της αρκεί να καλέσετε τη μέθοδο sort, δίνοντας σαν όρισμα ένα πίνακα ακεραίων.

Quick Sort χειρότερη περίπτωση (ερώτημα Α)

Η χειρότερη περίπτωση είναι η περίπτωση κατά την οποία τα στοιχεία είναι ίδια.

Για μέγεθος εισόδου χειρότερης περίπτωσης από 1000 μέχρι 8000 καταγράφηκαν οι συκγρίσεις καθώς και ο χρόνος εκτέλεσης σε δευτερόλεπτα έτσι όπως φαίνονται στον παρακάτω πίνακα.

Είσοδος	Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα εκτέλεσης
1000	501499	501498	0.005050878
2000	2003999	2002998	0.00160817700
3000	4505999	4504498	0.003668511
4000	8005999	8005998	0.00556541
5000	12507499	12507498	0.00838328
6000	18008999	18008998	0.016407867

7000	24510499	24510498	0.020692024
8000	32011999	32011998	0.028301651
9000	40513499	40513498	0.031980025
10000	50014999	50014998	0.042881425

Πίνακας 1: Ανάλυση Χειρότερης Περίπτωσης QuickSort

Οι συγκρίσεις υπολογίστηκαν με τη βοήθεια μιας static μεταβλητής (compares), η οποία αυξανόταν κατά ένα σε κάθε σύγκριση που γινόταν.

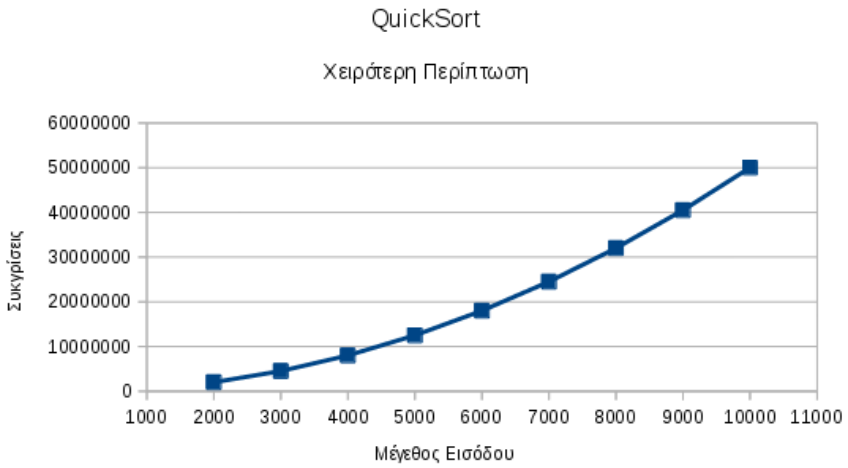
Θεωρητικά οι συγκρίσεις στη χειρότερη περίπτωση δίνονται από τον τύπο

$$\frac{(n+1)(n+2)}{2}-3$$

Οι τιμές αυτές διαφέρουν λίγο από τις τιμές που μετρήθηκαν λόγω της υλοποίησης.

Για τον υπολογισμό του χρόνου εκτέλεσης χρησιμοποιήθηκε η μέθοδος **nanoTime** της κλάσης System. Συγκεκριμένα υπολογίστηκε η ώρα εκτέλεσης πριν την κλήση της sort και μετά από αυτή και έγινε η αφαίρεση με την κατάλληλη μετατροπή από nanoseconds σε seconds. Με τον ίδιο τρόπο υπολογίστηκε ο χρόνος εκτέλεσης και στους υπόλοιπους αλγορίθμους.

Η γραφική παράσταση του μεγέθους της εισόδου σαν συνάρτηση του αριθμού των συγκρίσεων φαίνεται παρακάτω.



Είναι φανερό ότι ο αριθμός των συγκρίσεων αυξάνεται εκθετικά με το μέγεθος της εισόδου.

Merge Sort χειρότερη περίπτωση (ερώτημα Α)

Ο αντίστοιχος πίνακας για τις ίδιες μετρήσεις στην MergeSort φαίνεται παρακάτω.

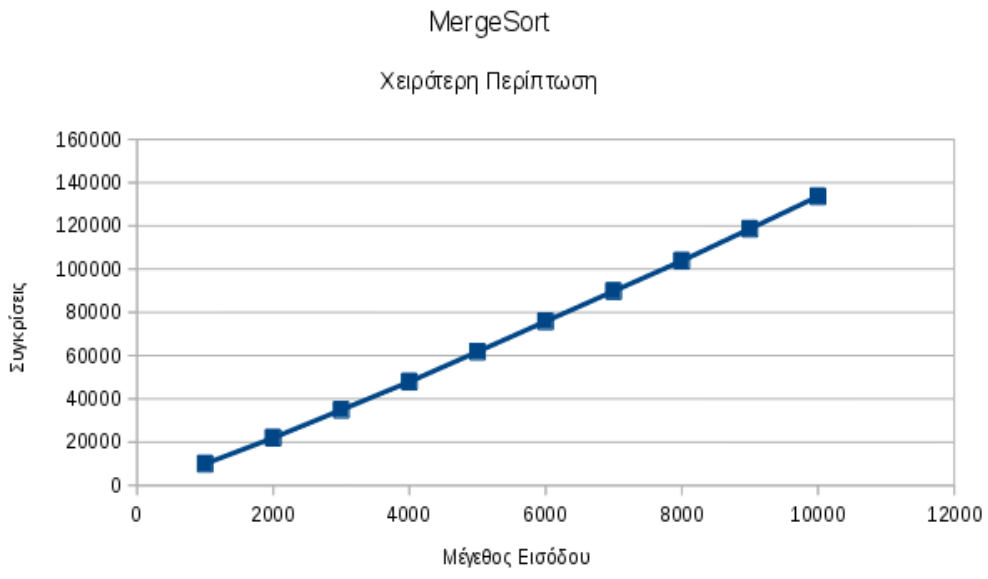
Θεωρητικά στη χειρότερη περίπτωση χρειάζονται

$$nlogn-2^{logn}+1$$

συγκρίσεις.

Είσοδος	Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα εκτέλεσης
1000	9976	8966	0.053644667
2000	21952	19932	0.083135915
3000	34904	31653	0.100113950
4000	47904	43864	0.10970716
5000	61808	56439	0.113738325
6000	75808	69305	0.13092824
7000	89808	82412	0.142750668
8000	103808	95727	0.147208848
9000	118616	109222	0.161939879
10000	133616	122878	0.17127545900

Οι συγκρίσεις είναι ίδιες με τη μέση περίπτωση για είσοδο έναν αντεστραμένα ταξινομημένο πίνακα. Υπάρχει βέβαια και είσοδος που δίνει ακριβώς nlogn-n+1 συγκρίσεις.
Ακολουθεί το διάγραμμα, το οποίο παρουσιάζει γραμμική συμπεριφορά!



Insertion Sort χειρότερη περίπτωση (ερώτημα Α)

Είσοδος	Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα εκτέλεσης
1000	999000	1000000	0.008197074
2000	3998000	4000000	0.009335787
3000	8997000	9000000	0.013134681
4000	15996000	16000000	0.0182982370
5000	24995000	25000000	0.0253816610
6000	35994000	36000000	0.0338938610
7000	48993000	49000000	0.0442985150
8000	63992000	64000000	0.0559338960
9000	80991000	81000000	0.0688565900
10000	99990000	100000000	0.0842534990

Στην Insertion Sort οι συγκρίσεις, σύμφωνα με τη θεωρία, είναι O(n^2), κάτι το οποίο επιβεβαιώνεται και από τις μετρήσεις.

Όσον αφορά τη σχέση του πλήθους της εισόδου με το χρόνο εκτέλεσης το παρακάτω διάγραμμα φανερώνει εκθετική αύξηση.

Σύγκριση Αλγορίθμων ως προς τον χρόνο

Παρατηρώντας τους χρόνους χειρότερης περίπτωσης βλέπουμε ότι ο **QuickSort** είναι εμφανών γρηγορότερος από τους άλλου δύο, κάτι που γνωρίζουμε αφού ο QuickSort πρακτικά είναι ο πιο γρήγορος O(nlogn) αλγόριθμος ταξινόμησης.

Quick Sort μέση περίπτωση (ερώτημα Β)

Τα 5 αρχεία αρχεία που χρησιμοποιούνται εδώ έχουν μέγεθος 1000,5000,10000,50000 και 100000 στοιχεία αντίστοιχα το ένα κάτω από το άλλο. Τα αρχεία αυτά είναι αποθηκευμένα στο φάκελο src με το όνομα datasetx.txt, όπου x ο αριθμός του κάθε αρχείου.

Χρησιμοποιώντας τη κατάλληλη επιλογή από το μενού έχουμε τις μετρήσεις που ακολουθούν.

Οι θεωρητικές συγκρίσεις της μέσης περίπτωσης σύμφωνα με την απόδειξη (σελίδα 54) είναι O(nlogn).

Data Set 1 (N=1000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
9591	9965	0.000782105

Data Set 2 (N=5000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
54029	61438	0.00917498000

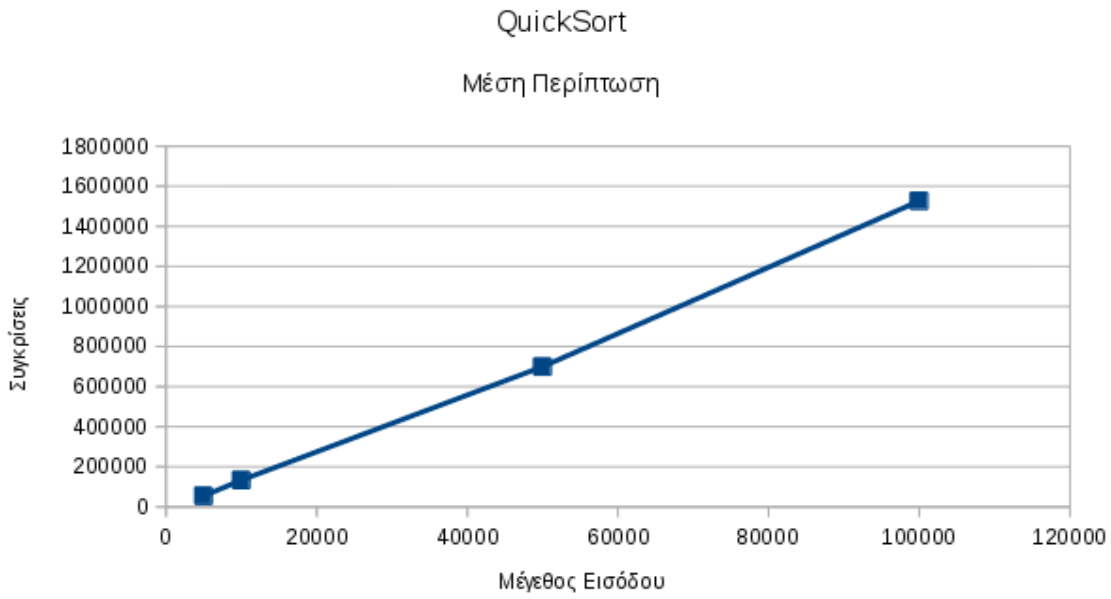
Data Set 3 (N=10000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
132771	132877	0.01236442300

Data Set 4 (N=50000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
699574	780482	0.026621379

Data Set 5 (N=100000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
1526556	1660964	0.025231658

Οι θεωρητικές συγκρίσεις όπως είναι εμφανές είναι ικανοποιητικά κοντά στις πραγματικές για τα 5 αυτά σετ αριθμών.

Το αντίστοιχα διάγραμμα μεγέθους εισόδου και συγκρίσεων δείχνει σχεδόν γραμμική σχέση.



Merge Sort μέση περίπτωση (ερώτημα Β)

Στη μέση περίπτωση η πολυπλοκότητα της Merge Sort είναι O(nlogn), όπως ακριβώς είναι και στην QuickSort. Χρησιμοποιώντας τα ίδια αρχεία έχουμε τις παρακάτω μετρήσεις και το διάγραμμα μεγέθους εισόδου – συγκρίσεων.

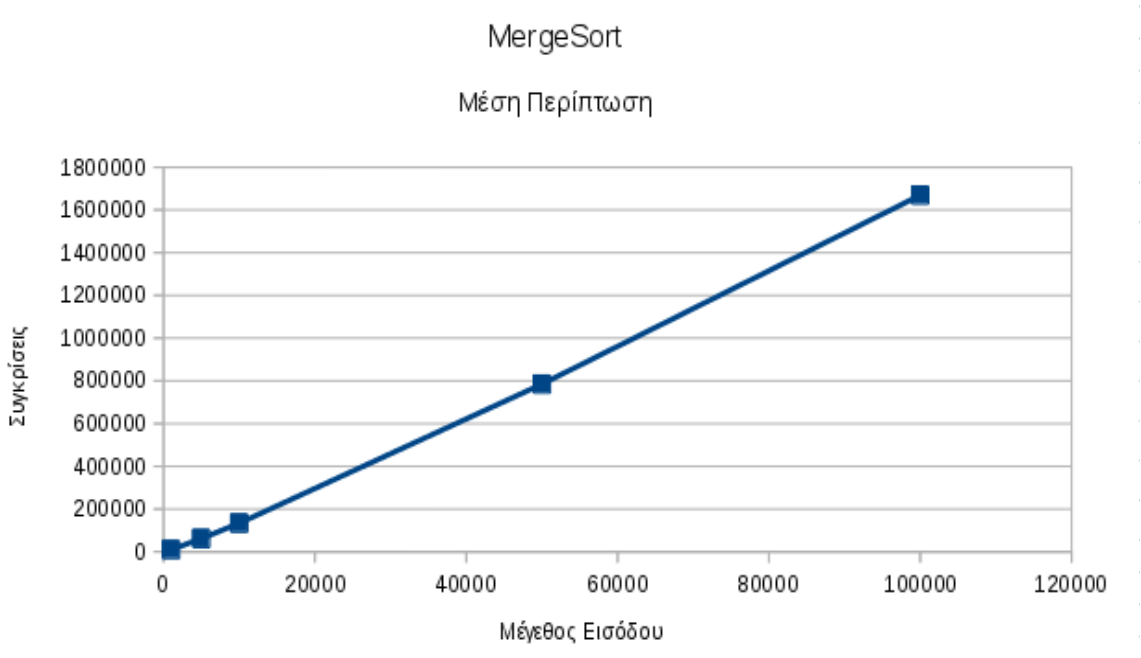
Data Set 1 (N=1000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
9976	9965	0.055951987

Data Set 2 (N=5000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
61808	61438	0.0607510020

Data Set 3 (N=10000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
133616	132877	0.11560242500

Data Set 4 (N=50000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
784464	780482	0.351275756

Data Set 5 (N=100000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
1668928	1660964	0.572891359



Παρατηρούμε ότι ο αριθμός των συγκρίσεων είναι ίδιος με αυτόν της

χειρότερης περίπτωσης, μιας και στη MergeSort εκτελείται πάντα ο ίδιος αρθμός βημάτων ανεξαρτήτως του είδους της εισόδου. Σημειώνεται ότι μετρήθηκαν όλες οι συγκρίσεις, συμπεριλαμβανομένου και των συγκρίσεων που γίνονται με το μέγεθος του πίνακα για την εισαγωγή των στοιχείων που απέμειναν στην διαδικασία της συγχώνεσης.

Insertion Sort μέση περίπτωση (ερώτημα Β)

Στη μέση περίπτωση στον Insertion Sort γίνονται $O(n^2)$ συγκρίσεις. Στις συγκεκριμένες τιμές παρατηρούμε ότι έχουμε αρκετά λιγότερες συγκρίσεις από n στο τετράγωνο.

Data Set 1 (N=1000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
502886	1000000	0.0173774380

Data Set 2 (N=5000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
12663302	25000000	0.032561245

Data Set 3 (N=10000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
49897310	100000000	0.067618793

Data Set 4 (N=50000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
1254204854	2500000000	0.994647033

Data Set 5 (N=100000)		
Συγκρίσεις	Συγκρίσεις Θεωρητικά	Δευτερόλεπτα Εκτέλεσης
713981770	10000000000	3.927803792

Στο παρακάτω διάγραμμα βλέπουμε ότι ο αλγόριθμος αυτό δυσκολεύεται αρκετά στο DataSet 4. Αυτό συμβαίνει γιατί οι μικρές τιμές δεν βρίσκονται στην αρχή του αρχείου. Έτσι έχουν περισσότερη "απόσταση" να διανύσουν, άρα και περισσότερες συγκρίσεις να κάνουν.

Συνοπτικά

Συνοψίζοντας γρηγορότερος αλγόριθμος και στη χειρότερη περίπτωση, αλλά και στη μέση είναι ο QuickSort. Ακολουθεί ο InsertionSort, ο οποίος είναι αρκετά αργός για πολύ μεγάλη είσοδο.
Ο παρακάτω πίνακας ίσως διαλευκάνει την υπόθεση:

	Χειρότερη Περίπτωση										Μέση Περίπτωση				
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000	DS1	DS2	DS3	DS3	DS5
QuickSort	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
MergeSort	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2
InsertionSort	2	2	2	2	2	2	2	2	2	2	2	2	2	3	3

Πράγματι, η μόνη περίπτωση όπου ο InserctionSort είναι πιο αργός από τον MergeSort είναι η περίπτωση με N=50000 και N=100000. Ο InsertionSort δεν προτιμάται λοιπόν για μεγάλες εισόδους. Μάλιστα σε περίπτωση χειρότερης

περίπτωσης με N=100000 ο InsertionSort χρειάστηκε 7.99 δευτερόλεπτα για να ολοκληρώσει την ταξινόμηση.

BinarySearch

Η δυική αναζήτηση είναι μια αρκετά γρήγορη αναζήτηση με μέσο χρόνο logn+1, η οποία μπορεί να βρεις ένα στοιχείο σε μία ταξινομημένη λίστα.

Αρχικά ο αλγόριθμος βρίσκει το μεσαίο στοιχείου της λίστας και ελέγχει αν αυτό είναι μεγαλύτερο του στοιχείου που ψάχνουμε. Αν είναι τότε η αναζήτηση περιορίζεται στο αριστερό μέρος της λίστας, διαφορετικά στο δεξί.
Στο τέλος, αν το στοιχείο τελικά υπάρχει, θα είναι αποθηκευμένο στο τελευταίο στοιχείου που θα έχει μείνει.

Ο αλγόριθμος αυτός βρίσκεται στο αρχείο BinarySearch.java. Ακολουθεί ένα παράδειγμα εκτέλεσης.

```
Enter the list of sorted integers. (Seperate them with comma): 1,2,3,4,5,6,7,8,9,10,11,12,13,14,123,34243,432424
Enter the target number:5
Found in 4 steps
FOUND in position 4
```

BinaryInterpolationSearch

Η δυική αναζήτηση παρεμβολής διαφέρει από τη δυική αναζήτηση στο ότι το στοιχείο το οποίο χωρίζουμε την λίστα δεν είναι το μεσαίο, αλλά κάτι πιο έξυπνο. Υπολογίζεται από τον τύπο:

$$mid = \frac{target - A[low] * (high - low)}{A[high] - A[low]}$$

όπου target είναι ο αριθμός που ψάχνουμε, low ο δείκτης που αρχικά δείχνει στη πρώτη θέση (μηδέν) της λίστας και low αυτός που δείχνει στο τέλος της (length-1).
Στην ουσία, με τον τρόπο αυτό παίρνουμε μια εκτίμηση της θέσης του target. Αφού υπολογιστεί η τιμή mid συγκρίνεται η τιμή της λίστας στη θέση αυτή με το target. Αν είναι μεγαλύτερη του target, τότε το στοιχείο (εφόσον η λίστα είναι ταξινομημένη) θα βρίσκεται στο αριστερό κομμάτι, οπότε το low γίνεται mid – 1 και σε αντίθετη περίπτωση mid + 1. Όταν τώρα το στοιχείο που δείχνει ο mid ισούται με το target, τότε βρήκαμε το στοιχείο.

Στην ακραία περίπτωση, μετά από αυτό το loop το στοχείο θα έχει μείνει εκεί που δείχνει ο low, οπότε κάνουμε τον κατάλληλο έλεγχο εκτός αυτού.

Η πολυπλοκότητα του αλγορίθμου αυτού είναι O(loglogn).

Θέμα 2: Υλοποίηση 2-3-4 δέντρου.

Εισαγωγή

Τα 2-3-4 δέντρα είναι μια δομή δεδομένων που χρησιμοποιείται συχνά για την υλοποίηση λεξικών, δηλαδή τύπων δεδομένων που μπορούν να κάνουν τις πράξεις Access(x), Insert(x) και Delete(x).

Τα δέντρα αυτά είναι υψοζυγισμένα, δηλαδή το ύψος του αριστερού υπόδεντρου μπορεί να διαφέρει μέχρι και ένα από του δεξιού. Επιπλέον το αριστερό και το δεξί υπόδεντρο είναι επίσης υψοζυγισμένα.

Η ονομασία προέρχεται από τα εξής χαρακτηριστικά του:

- Οι κόμβοι με ένα στοιχείο πρέπει να έχουν δύο παιδιά (εκτός και αν είναι στο τέλος (φύλλα).
- Οι κόμβοι με δύο στοιχεία πρέπει να έχουν τρία παιδιά.
- Οι κόμβοι με τρία στοιχεία πρέπει να έχουν τέσσερα παιδιά.

Επιπλέον στοιχείο είναι ότι ένας κόμβος με ένα στοιχείο πρέπει να έχει ένα παιδί μεγαλύτερο από αυτόν και ένα στοιχείο μικρότερο από αυτόν. Αντίστοιχα ένας κόμβος με δύο στοιχεία πρέπει να έχει ένα παιδί μικρότερο από το πρώτο στοιχείο, ένα ανάμεσα στο πρώτο και το δεύτερο και ένα μεγαλύτερο από το δεύτερο κοκ.

Τέλος τα στοιχεία είναι πάντα σε αύξουσα σειρά από τα αριστερά προς τα δεξιά.

Βασική Δομή

Το πρόγραμμα περιλαμβάνει τέσσερεις κλάσεις, τη MainClass, τη DataItem, τη Node, και τη Tree234.

Η κλάση **DataItem** χρησιμοποιείται έτσι ώστε κάθε στοιχείο στο Node να αναφέρεται από ένα αντικείμενο (αντί από έναν απλό ακέραιο). Η κλάση αυτό

περιλαμβάνει έναν ακέραιο dData, μία συνάρτηση displayItem, η οποία εμφανίζει το item και κατά την δημιουργία ενός νέου τέτοιου αντικειμένου (στον constructor), το ένα όρισμα που έρχεται σαν είσοδος τίθεται στο dData.

Η κλάση Node περιγράφει έναν κόμβο με όλες τις απαραίτητες σε αυτόν λειτουργείες, η κλάση Tree234 περιγράφει ένας 234 δέντρο, ενώ η MainClass περιέχει το μενού με όλες τις λειτουργίες και κατασκευάζει το δέντρο theTree πάνω στο οποίο τρέχει το πρόγραμμα.

Αναλυτική εξέταση των βασικών κλάσεων

▶ Node.java

Η κλάση **Node** υπάρχουν οι μεταβλητές numItems, η οποία δείχνει πόσα στοιχεία περιέχονται στον συγκεκριμένο κόμβο, η parent που είναι τύπου Node και έχει τον γονέα, ο πίνακας κόμβων childArray[4] που περιέχει όλα τα παιδιά του συγκεκριμένου κόμβου και ο πίνακας itemArray[3] που περιέχει όλα τα στοιχεία του.
Επιπλέον η κλάση αυτή περιλαμβάνει τις παρακάτω μεθόδους:

```
void connectChild(int childNum, Node child)
```

Η μέθοδος αυτή προσθέτει τον κόμβο child στον πίνακα των παιδιών childArray στη θέση childNum.

```
Node disconnectChild (int childNum)
```

Αφαιρεί το παιδί που βρίσκεται στη θέση chilNum θέτοντάς το σαν null.

```
Node getChild (int childNum)
```

Επιστρέφει το παιδί που βρίσκεται στη θέση childNum του πίνακα childArray.

```
int getChildNum ( Node child)
```

Επιστρέφει τη θέση του παιδιού στον πίνακα σε σχέση με τον πατέρα. Αν δηλαδή είναι το πρώτο παιδί του πατέρα (από αριστερά προς τα δεξιά) επιστρέφει 0, αν είναι το δεύτερο 1 κτλ.

```
int getNoofClids ()
```

Επιστρέφει τον αριθμό των παιδιών. Προφανώς δε μετράει αυτά που έχουν τεθεί ως null.

```
Node getParent()
```

Επιστρέφει τη μεταβλητή κλάσης parent.

```
boolean isLeaf()
```

Επιστρέφει true αν το συγκεκριμένος κόμβος είναι φύλλο.

```
int getNumItems()
```

Επιστρέφει τον αριθμό των αντικειμένων (μεταβλητή κλάσης numItems)

```
DataItem getItem (int index)
```

Επιστρέφει το DataItem που βρίσκεται στη θέση index.

```
void setItem (int index, int number)
```

Θέτει το αντικείμενο που βρίσκεται στη θέση index με τον αριθμό number (απευθείας ανάθεση στην μεταβλητή dData της DataItem).

```
boolean replace (DataItem old , DataItem with )
```

Αντικαθιστά την πρώτη εμφάνιση του DataItem old με το DataItem with στον κόμβο.

```
boolean isFull()
```

Επιστρέφει true αν ο κόμβος είναι γεμάτος (περιέχει 3 στοιχεία).

```
int findItem (int key)
```

Ψάχνει τον πίνακα αντικειμένων του κόμβου και επιστρέφει τη θέση στην οποία βρέθηκε το key, ή -1 αν δεν βρέθηκε. Η αναζήτηση υλοποιείται με μια απλή for η οποία ψάχνει όλο τον πίνακα των αντικειμένων και συγκρίνει κάθε στοιχείο με το key.

```
DataItem removeItem()
```

Αφαιρεί (θέτει ίσο με null) το μεγαλύτερο στοιχείο του πίνακα αντικειμένων itemArray και επιστρέφει τον ανανεωμένο πίνακα.

```
void deleteItem (int number)
```

Διαγράφει το στοιχείο Number από τον πίνακα των αντικειμένων. Αυτό γίνεται ως εξής: Αρχικά δημιουργείται μια λίστα. Με ένα βρόγχο for ελέγχεται να το κάθε ένα στοιχείου του πίνακα είναι διάφορο του number. Αν είναι τότε εισέρχεται στη λίστα. Τέλος η λίστα αυτή μετατρέπεται σε πίνακα και το numItems μειώνεται κατά ένα.

```
void displayNode()
```

Εμφανίζει όλα τα στοιχεία του κόμβου με το κατάλληλο διαχωριστικό.

```
int insertItem (DataItem newItem)
```

Προσθέτει ένα νέο στοιχείο στον κόμβο. Το στοιχείο newItem μπαίνει στη κατάλληλη θέση έτσι ώστε τα στοιχεία στον κόμβο να είναι πάντα σε αύξουσα σειρά.

```
ArrayList  getSortedElements ()
```

Η μέθοδος αυτή επιστρέφει ταξινομημένα τα στοιχεία της μαζί με τα στοιχεία όλων των παιδιών της. Χρησιμοποιείται στην αναδρομική μέθοδο toIntArray, η οποία υπάρχει στη κλάση Tree234. Γνωρίζουμε ότι κάθε κόμβος έχει ταξινομημένα αύξουσα τα στοιχεία του και ότι τη σχέση των στοιχείων των παδιών με τα στοιχεία του πατέρα (βλ Εισαγωγή).
Αρχικοποιούμε δύο ArrayList, τη sortedElements και τη tempList. Ξεκινάμε ένα loop όσο το πλήθος των στοιχείου του κόμβου και σε έναν προσωρινό Node (tempNode) βάζουμε το i-οστό παιδί (getChild(i)). Έτσι γεμίζουμε την tempList με τα στοιχεία αυτού του παιδιού. Με αυτό το τρόπο η tempList θα περιέχει το επιθυμητό αποτέλεσμα, το οποίο θέτουμε στο sortedList και επιτρέφουμε.

▶ Tree234.java

Αρχικά ένα αντικείμενο Tree234 περιλαμβάνει μια μεταβλητή που δείχνει το πλήθος των κόμβων (noOfElements) και έναν κόμβο root.
Οι μέθοδοι σε αυτή τη κλάση φαίνονται παρακάτω:

```
Node getNextChild (Node theNode, int theValue)
```

Επιστρέφει το επόμενο παιδί του theNode που δεν περιέχει το theValue. Σε έναν βρόγχο ελέγχεται πότε το στοιχείου του theNode θα είναι μεγαλύτερο από την τιμή theValue, έτσι ώστε να επιστρέψει το κατάλληλο παιδί του theNode

```
(*) boolean contains (int key)
```

Επιστρέφει true αν και μόνο αν το δέντρο περιέχει την τιμή. Η αναζήτηση ξεκινάει από τον κόμβο root (ρίζα) οπότε ένας κόμβος με το όνομα curNode αρχικοποιείται με τη ρίζα. Μέσα σε έναν while(true) βρόγχο τρέχει αρχικά η findItem(key) για τον curNode. Αν βρεθεί εκεί τότε η contains επιστρέφει true. Αν τώρα το curNode είναι φύλλο επιστρέφει false, αφού δεν υπάρχουν άλλα παιδιά για να ψάξει και σε οποιαδήποτε άλλη περίπτωση το curNode τίθεται ίσο με το επόμενο παιδί με τη μέθοδο getNextChild(curNode,key).

```
Node find (int key)
```

Είναι η ίδια μέθοδος με την contains, με τη διαφορά ότι επιστρέφει όλο τον κόμβο στον οποίο βρέθηκε το key. Χρειάζεται σε πατακάτω μεθοδο.

```
Node getNextChild (Node theNode, int theValue)
```

Επιστρέφει το επόμενο παιδί.

```
(*) void insert (int dValue)
```

Η μέθοδος εισόδου ενός νέου στοιχείου στον κόμβο. Αρχικά αυξάνουμε τον αριθμό των στοιχείων κατά ένα, θέτουμε τον κόβμο curNode ίσο με το root και δημιουργούμε ένα DataItem με όρισμα το dValue (συνεπώς η μεταβλητή dData της κλάσης DataItem για το αντικείμενο αυτό θα πάρει τη τιμή dValue).
Σε έναν while(true) βρόγχο ο οποίος τερματίζεται με break όταν η κόμβος στον οποίο αναφερόμαστε είναι φύλλο, αρχικά ελέγχουμε αν είναι γεμάτος με την μέθοδο isFull. Αν είναι, τότε κάνουμε split τον κόμβο αυτό το curNode παίρνει τη τιμή του επόμενο παιδιού. Στο τέλος αυτής της διαδικασίας στο curNode θα περιέχεται ο κόμβος στον οποίο πρέπει να μπει το στοιχείο dvalue κάτι που γίνεται καλώντας την insertItem.

```
void recDisplayTree (Node thisNode, int level, int childNumber)
```

Καλείται από τη displayTree με ορίσματα root, 0, 0. Εμφανίζει κάθε κόμβο κάνοντας αναδρομικές κλήσεις για κάθε νέο κόμβο.

```
(*) boolean empty ()
```

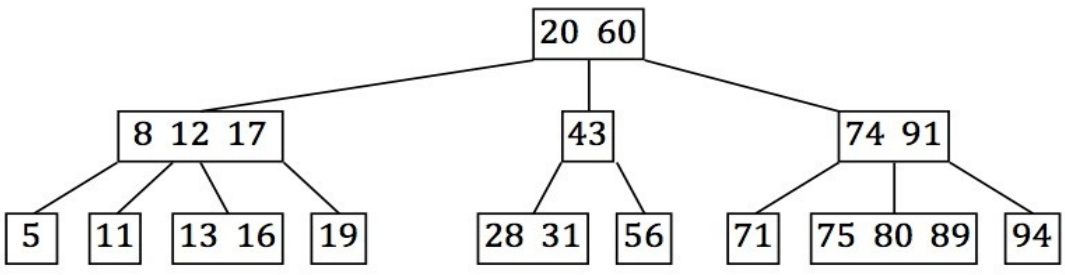
Επιστρέφει true αν το δέντρο είναι άδειο. Η συνθήκη ελέγχου είναι η εξής:
Αν η root είναι φύλλο και όλα τα στοιχεία του είναι null τότε το δέντρο είναι άδειο.

```
List<Integer> merge ( List<Integer> First , List<Integer> Second)
```

Η διαδικασία ένωσης δύο λιστών σε μία που χρησιμοποιήθηκε στον MergeSort. Χρησημεύει για την υλοποίηση της toIntArray.

```
(*) List<Integer> toIntArray (Node curNode)
```

Αρχικά δημιουργείται μια λίστα στην οποία αποθηκεύεται η επιστρεφόμενη τιμή του getSortedElements για τον curNode. Αφού βρούμε τον αριθμό των παιδιών για το curNode ξεκινάμε ένα for loop μέχρι τον αριθμό αυτό. Στον κόμβο next αποθηκεύουμε το i-οστό παιδί. Αν αυτό το παιδί δεν είναι φύλλο, τότε στη λίστα εκχωρείται το αποτέλεσμα της merge με ορίσματα τη λίστα και την αναδρομική κλήση της toIntArray με όρισμα τον Next. Η toIntArray επιστρέφει τη λίστα αυτή.
Για να γίνει πιο σαφής η λειτουργία αυτής της αναδρομικής μεθόδου έστω ότι έχουμε το παρακάτω δέντρο:



Η συνάρτηση αρχικά θα εκτελεστεί για το root, οπότε η getSortedElements θα επιστρέψει ταξινομημένα τα στοιχεία της root και των παιδιών της (8,12,,17,20,60,74,91). Αφού μπούμε στη for loop και θα γίνει αναδρομική κλήση του ίδιου πράγματος για το πρώτο, το δεύτερο και το τρίτο παιδί. Έτσι όταν η αναδρομή επιστρέψει στην αρχική κλήση η επιστρεφόμενη τιμή που θα έχει προκύψει θα έχει προκύψει θα είναι ένας ταξινομημένος πίνακας, ο οποίος όμως θα περιέχει και διπλά στοιχεία, καθώς για παράδειγμα τα στοιχεία του πρώτου παιδιού της ρίζας μπαίνουν στη λίστα και στη κλήση της toIntArray για το root και στην αναδρομική κλήση της για το πρώτο παιδί. Οπότε με τη βοήθεια της merge αφαιρούμε τα κοινά στοιχεία και έχουμε στην έξοδο τη ζητούμενη λίστα.

Η υλοποίηση έγινε με λίστα για λόγους ευκολίας. Μπορεί εύκολα να μετατραπεί σε πίνακα.

```
Node SwapWithInorderSuccessor(int number)
```

Χρησιμοποιείται για τη delete και αντιμεταθέτει τον αριθμό number με τον αμέσως επόμενο μεγαλύτερο στο δέντρο. Αρχικά βρίσκουμε τον κόμβο όπου βρίσκεται το number (με την find) αν βέβαια υπάρχει στο δέντρο (ο έλεγχος αυτός γίνεται με την contain). Χρησιμοποιώντας το αποτέλεσμα του toIntArray, βρίσκουμε το επόμενο μεγαλύτερο στοιχείο. Στη συνέχεια βρίσκουμε με τον ίδιο τρόπο τον κόμβο όπου περιέχεται το στοιχείο αυτό, δημιουργούμε δύο DataItems (ένα για την παλιά τιμή και ένα και την καινούρια) και κάνουμε την αντιμετάθεση χρησιμοποιώντας 2 φορές την replace μία για κάθε στοιχείο.

```
(*) int NumberOfNodes( Node curNode)
```

Επιστρέφει τον αριθμό των κόμβων (-1, επειδή υπήρχε κάποιο bug) , αν εκτελεστεί με όρισμα τη ρίζα. Αρχικά υπολογίζεται το πλήθος των παιδιών και έπειτα για κάθε παδί καλείται αναδρομική οι NumberOfNodes, με το πλήθος των παιδιών της κάθε μίας να αποθηκεύεται στην μεταβλητή children, η οποία και επιστρέφεται. Κατά τη κλήση της μεθόδου αυτής προστίθεται το 1 έτσι ώστε να εμφανιστεί το σωστό αποτέλεσμα.

```
Node getRoot ()
```

Απλώς επιστρέφει τη ρίζα.

```
(*) Tree234 delete ( int number)
```

Η διαγραφή ενός στοιχείου από τον πίκανα ακολουθεί τον παρακάτω

αλγόριθμο:
-Αρχικά εντοπίζεται ο κόμβος n, ο οποίος περιέχει τον αριθμό που πρέπει να διαγραφεί (με συνδιασμό της contain και της find).
-Αν ο κόμβος n δεν είναι φύλλο τότε γίνεται αντιμετάθεση με το αμέσως επόμενο μεγαλύτερο αριθμό (με τη χρήση της SwapWithInnorderSuccesor).
-Εάν ο κόμβος που είμαστε τώρα περιέχει περισσότερα από ένα στοιχεία απλά διαγράφουμε το στοιχείο που είναι διαγραφή.
-Διαφορετικά, εφόσον η διαγραφεί προκαλέσει κενό κόμβο αναδιανέμουμε (δεξιά ή αριστερά) ή ενώνουμε κόμβους.

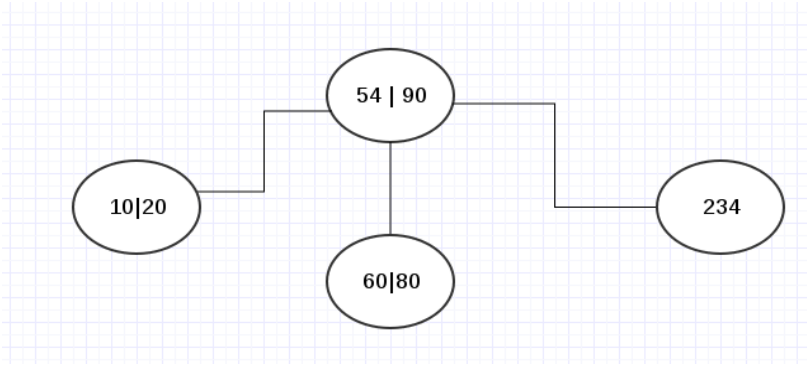
*μέθοδοι που ζητούνται στην εκφώνηση. Υπάρχει επιλογή στο μενού για την εκτέλεση κάθε μίας από αυτές.

Παράδειγμα Εκτέλεσης

Εισάγουμε με τη σειρά του αριθμούς: 234, 54, 20, 10, 60, 90, 80 οπότε παίρνουμε το παρακάτω δέντρο:

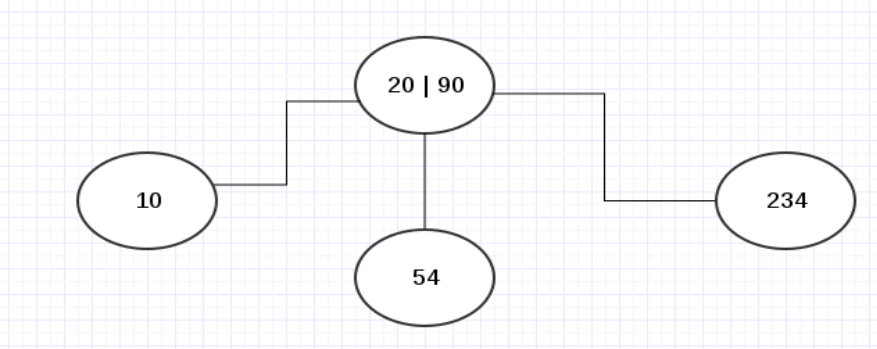
```
Select an option
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 0
level= 0 child= 0 /54/90/
level= 1 child= 0 /10/20/
level= 1 child= 1 /60/80/
level= 1 child= 2 /234/
```

Σχηματικά έχει τη παρακάτω μορφή:



Αν αφαιρέσουμε με τη σειρά τους αριθμού 60 και 80 τότε θα γίνει δεξιά αναδιανομή συνεπώς το δέντρο θα έχει τη παρακάτω μορφή:

```
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 0
level= 0 child= 0 /20/90/
level= 1 child= 0 /10/
level= 1 child= 1 /54/
level= 1 child= 2 /234/
```



Εάν τώρα σε αυτό το δέντρο αναζητήσουμε το 10 θα δούμε ότι θα βρεθεί, ενώ το 100 όχι.

```
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 2
Enter value to find 10
*Found
Select an option
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 2
Enter value to find 100
*Could not found
```

Χρησιμοποιώντας την επιλογή 4 μπορούμε να επιβεβαιώσουμε ότι το δέντρο δεν είναι άδειο.

```
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 4
*Is Not Empty
```

Μπορούμε ακόμα να δούμε τον αριθμό των κόμβων, χάρη στην αναδρομική μέθοδο που δημιουργήθηκε.

```
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 0
level= 0 child= 0 /20/90/
level= 1 child= 0 /10/
level= 1 child= 1 /54/
level= 1 child= 2 /234/
Select an option
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 5
*Number Of Nodes 4
```

Με την έκτη και τελευταία επιλογή έχουμε έναν ταξινομημένο πίνακα όλων των

στοιχείων που βρίσκονται στο δέντρο.

```
0.Display | 1.Insert | 2.Find | 3.Delete | 4.Is Empty | 5.Number Of Nodes | 6. To Int Array 6
|10|20|54|90|234|
```

Βιβλιογραφία

-Δομές Δεδομένων, Αθανάσιος Κ. Τσακαλίδης
-Sesh Venugopal, youtube channel
-234 tree instuctions
-Interpolation search