



Π.Μ.Σ. ΤΜΗΜΑΤΟΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**Υπολογιστική σύγκριση των αλγορίθμων Heap Sort και Weak
Heap Sort.**

**Βασιλεία Φορμόζη
Α.Μ. 43/11**

Επιβλέπων Καθηγητής: Σαμαράς Νικόλαος, Επ. Καθηγητής

Τμήμα Εφαρμοσμένης Πληροφορικής
Πρόγραμμα Μεταπτυχιακών Σπουδών Ειδίκευσης
Πανεπιστήμιο Μακεδονίας
Θεσσαλονίκη

Ιούνιος, 2012

Περίληψη

Σκοπός της διπλωματικής αυτής εργασίας είναι η μελέτη δύο γνωστών αλγορίθμων ταξινόμησης του HeapSort και του WeakHeapSort οι οποίοι χρησιμοποιούν συγκεκριμένες δομές δεδομένων που ονομάζονται σωροί.

Αφότου γίνει μία επιγραμματική περιγραφή των γνωστότερων αλγορίθμων ταξινόμησης και μία ιστορική αναδρομή, θα προχωρήσουμε στην περιγραφή των σωρών και στον τρόπο εισαγωγής και διαγραφής στοιχείων στις δομές σωρών. Στη συνέχεια θα γίνει μία αναλυτικότερη περιγραφή των δύο γνωστών αλγορίθμων αρχίζοντας από τον HeapSort, με ένα παράδειγμα, την υλοποίησή του σε κώδικα και την επεξήγησή του καθώς και την ανάλυση πολυπλοκότητάς του. Το ίδιο θα γίνει και με τον αλγόριθμο Weak HeapSort ο οποίος αποτελεί εξέλιξη του HeapSort.

Τέλος θα γίνει μία υπολογιστική μελέτη των δύο αυτών αλγορίθμων η οποία θα αφορά των αριθμό επαναλήψεων, το χρόνο επεξεργασίας και τη μέτρηση των βασικών πράξεων που επιτελούνται. Αφότου έχουν υλοποιηθεί σε Java και οι δύο αλγόριθμοι, θα εισάγουμε στα προγράμματα αυτά στοιχεία με τη χρήση ψευδογεννήτριων συναρτήσεων και θα παρουσιάσουμε τα αποτελέσματα και τα συμπεράσματα της μελέτης.

Abstract

The main aim of this thesis is to study two well-known sorting algorithms, HeapSort and WeakHeapSort, which use specific data structures called heaps.

After a peer review of the most well-known sorting algorithms, this thesis describes the data structure heap and its basic operations, i.e. insert and delete of an element. Then, a detailed presentation of the two sorting algorithms will follow. Furthermore, these algorithms are implemented in code and then their complexity is analyzed.

Finally, this thesis presents a computational study of these algorithms. This computational study includes the number of comparisons, the execution time and the number of the basic operations. The algorithms are implemented in Java and data from pseudorandom functions are inserted in order to execute the algorithms. Finally, the conclusions of the thesis are presented.

Περιεχόμενα

ΠΙΝΑΚΑΣ ΕΙΚΟΝΩΝ	5
ΠΙΝΑΚΑΣ ΠΙΝΑΚΩΝ.....	6
1. Εισαγωγή	7
1.1 Αντικείμενο Εργασίας.....	7
1.2 Διάρθρωση της Εργασίας.....	9
1.3 Ευχαριστίες	10
2. Αλγόριθμοι Ταξινόμησης	11
2.1 Εισαγωγή	11
2.2 Bubble Sort.....	13
2.3 Selection Sort.....	15
2.4 Insertion Sort	17
2.5 Radix Sort	19
2.6 Count Sort.....	20
2.7 Bucket Sort	22
3. Σωροί και Δομή WeakHeap.....	24
3.1 Δένδρα.....	24
3.2 Σωροί.....	25
3.3 Δομή weakheap	31
4. Αλγόριθμος Heapsort.....	33
4.1 Ψευδοκώδικας Αλγορίθμου	33
4.1.1 Οι αλγόριθμοι buildheapup και buildheapdown	33
4.1.2 Ο αλγόριθμος Heap Sort	34
4.2 Παράδειγμα	36
4.3 Ανάλυση πολυπλοκότητας.....	36
5. Αλγόριθμος WeakHeapSort.....	38
5.1 Ψευδοκώδικας Αλγορίθμου	38
5.1.1 Ο αλγόριθμος Gparent.....	38
5.1.1 Ο αλγόριθμος Merge.....	38
5.1.3 Ο αλγόριθμος MergeForest.....	39
5.1.4 Ο αλγόριθμος Weak-heapsort	40
5.2 Παράδειγμα	41
5.3 Ανάλυση πολυπλοκότητας.....	51
6. Υπολογιστική Μελέτη	53
6.1 Εισαγωγή.....	53
6.2 Εκτέλεση πειραμάτων	53
7. Συμπεράσματα	60
Βιβλιογραφία	62
Παράρτημα: Κώδικας εφαρμογής.....	64
Π.1 Κλάση HeapSort.....	64
Π.2 Κλάση WeakHeapSort	67
Π.3 Κλάση HeapWeakHeap.....	72

ΠΙΝΑΚΑΣ ΕΙΚΟΝΩΝ

Εικόνα 1: Ψευδοκώδικας αλγόριθμου Bubble Sort	14
Εικόνα 2: Παράδειγμα ταξινόμησης με τον αλγόριθμο Bubble Sort	15
Εικόνα 3: Ψευδοκώδικας αλγόριθμου Selection Sort.....	16
Εικόνα 4: Παράδειγμα ταξινόμησης με τον αλγόριθμο Selection Sort	17
Εικόνα 5: Ψευδοκώδικας αλγόριθμου Insertion Sort	18
Εικόνα 6: Παράδειγμα ταξινόμησης με τον αλγόριθμο Insertion Sort.....	19
Εικόνα 7: Ψευδοκώδικας αλγόριθμου Radix Sort	20
Εικόνα 8: Ψευδοκώδικας αλγόριθμου Count Sort.....	21
Εικόνα 9: Παράδειγμα ταξινόμησης με τον αλγόριθμο Count Sort	22
Εικόνα 10: Ψευδοκώδικας αλγόριθμου Bucket Sort	23
Εικόνα 11: Σχεδόν πλήρες δυαδικό δέντρο	25
Εικόνα 12: Υλοποίηση σωρού με πίνακα	26
Εικόνα 13: Ένας σωρός και η υλοποίησή του με πίνακα	27
Εικόνα 14: Παράδειγμα της πράξης της ανόδου.	29
Εικόνα 15: Παράδειγμα της πράξεως της καθόδου.	31
Εικόνα 16: Δομή weakheap	32
Εικόνα 17: Παράδειγμα αλγόριθμου κατασκευής αρχικής σωρού με καθόδους.	34
Εικόνα 18: Παράδειγμα ταξινόμησης με τον αλγόριθμο Heap Sort.....	36
Εικόνα 19: Αρχικό δέντρο παραδείγματος	42
Εικόνα 20: Χρόνος εκτέλεσης αλγόριθμου HeapSort	54
Εικόνα 21: Αριθμός συγκρίσεων αλγόριθμου HeapSort	55
Εικόνα 22: Αριθμός αναθέσεων αλγόριθμου HeapSort	55
Εικόνα 23: Χρόνος εκτέλεσης αλγόριθμου WeakHeapSort	56
Εικόνα 24: Αριθμός συγκρίσεων αλγόριθμου WeakHeapSort.....	57
Εικόνα 25: Αριθμός αναθέσεων αλγόριθμου WeakHeapSort	57
Εικόνα 26: Σύγκριση χρόνου εκτέλεσης αλγόριθμων HeapSort και WeakHeapSort	58
Εικόνα 27: Σύγκριση αριθμού συγκρίσεων αλγόριθμων HeapSort και WeakHeapSort ..	59
Εικόνα 28: Σύγκριση αριθμού αναθέσεων αλγόριθμων HeapSort και WeakHeapSort ...	59

ΠΙΝΑΚΑΣ ΠΙΝΑΚΩΝ

Πίνακας 1: Συγκεντρωτικός Πίνακας Αλγόριθμων Ταξινόμησης.....	12
Πίνακας 2: Ψευδοκώδικας μεθόδου heapifyup	28
Πίνακας 3: Ψευδοκώδικας μεθόδου heapifyup	30
Πίνακας 4: Ψευδοκώδικας αλγόριθμου buildheapup	33
Πίνακας 5: Ψευδοκώδικας αλγόριθμου buildheapdown	33
Πίνακας 6: Ψευδοκώδικας αλγόριθμου Heap Sort	35
Πίνακας 7: Αλγόριθμος Gparent.....	38
Πίνακας 8: Αλγόριθμος Merge	39
Πίνακας 9: Αλγόριθμος MergeForest	40
Πίνακας 10: Αλγόριθμος MergeForest	41
Πίνακας 11: Αποτελέσματα αλγορίθμου HeapSort	54
Πίνακας 12: Αποτελέσματα αλγορίθμου WeakHeapSort.....	56

1. Εισαγωγή

1.1 Αντικείμενο Εργασίας

Η εύρεση της πολυπλοκότητας των αλγορίθμων και η υπολογιστική τους συμπεριφορά αποτελεί ένα από τα πιο θεμελιώδη ζητήματα στην επιστήμη της πληροφορικής. Υπάρχουν τρία είδη θεωρητικής πολυπλοκότητας: (1) η ανάλυση χειρότερης περίπτωσης, (2) η ανάλυση καλύτερης περίπτωσης και (3) η ανάλυση μέσης περίπτωσης (Sedgewick and et al., 1996). Αν και η σημασία της θεωρητικής πολυπλοκότητας είναι αδιαμφισβήτητη, εξίσου σημαντική είναι και η μελέτη της υπολογιστικής συμπεριφοράς των αλγορίθμων. Αλγόριθμοι, όπως αυτοί της ταξινόμησης είναι από τους πιο βασικούς και διδάσκονται σε όλα σχεδόν τα τμήματα Πληροφορικής. Η επιστημονική έρευνα στο γνωστικό αντικείμενο των αλγορίθμων έχει να κάνει με την ανάπτυξη νέων αλγορίθμων ή με τη βελτίωση ήδη υπαρχόντων.

Κάθε φορά, όμως, που αναπτύσσεται ένας νέος αλγόριθμος, για να συγκριθεί ως προς την υπολογιστική αποτελεσματικότητά του με τους ήδη υπάρχοντες απαιτείται μια χρονοβόρα και επίπονη διαδικασία. Η διαδικασία αυτή αναφέρεται ως υπολογιστική μελέτη. Πολλές εργασίες έχουν γραφτεί για το πως πρέπει να σχεδιάζονται τα πειράματα και πως πρέπει να διενεργούνται οι υπολογιστικές μελέτες (Hooker, 1994; Rardin et al., 2001). Για την υπολογιστική σύγκριση διαφόρων αλγορίθμων, απαραίτητη είναι η πολλαπλή εκτέλεση όλων των αλγορίθμων σε συλλογές από δεδομένα. Για την εξαγωγή ασφαλών συμπερασμάτων σχετικά με τη συμπεριφορά των αλγορίθμων, τα δεδομένα αυτά πρέπει να πληρούν ορισμένες προϋποθέσεις. Συγκεκριμένα, πρέπει να είναι μεγάλης διάστασης, να δημιουργούνται από γεννήτριες τυχαίων αριθμών και για κάθε διάσταση να λύνονται από τους αλγορίθμους 10 ή και περισσότερα στιγμιότυπα. Περισσότερες πληροφορίες σχετικά με τη δημιουργία τυχαίων συλλογών από δεδομένα μπορούν να βρεθούν στην αναφορά (Hall, 2001).

Η επιλογή των κατάλληλων δεικτών απόδοσης είναι ένας κρίσιμος παράγοντας σε μια υπολογιστική μελέτη (Crowder, 1979). Οι δείκτες απόδοσης πρέπει να είναι, όσο

το δυνατόν, πιο ανεξάρτητοι από την υπολογιστική μελέτη. Δείκτες απόδοσης που έχουν χρησιμοποιηθεί περισσότερο στις υπολογιστικές μελέτες είναι:

- Ο χρόνος επεξεργασίας
- Ο αριθμός των επαναλήψεων
- Η μέτρηση των βασικών πράξεων

Αν χρησιμοποιηθεί σε μια υπολογιστική μελέτη ο χρόνος επεξεργασίας ως δείκτης απόδοσης, πρέπει να περιλαμβάνονται μέθοδοι για την όμοια τυποποίηση των αποτελεσμάτων.

Αναλυτικότερες οδηγίες για τον τρόπο διενέργειας υπολογιστικών μελετών καθώς και για τα συχνότερα εμφανιζόμενα λάθη, μπορεί κανείς να βρει στην εργασία (Coffin et al., 2000).

Οι αλγόριθμοι ταξινόμησης είναι αλγόριθμοι που τοποθετούν τα στοιχεία μίας λίστας με συγκεκριμένη σειρά, από τις οποίες οι πιο γνωστές είναι η αριθμητική και η λεξικογραφική σειρά. Όσον αφορά την ταξινομημένη λίστα που προκύπτει πρέπει να τηρούνται δύο κανόνες:

- Τα στοιχεία της λίστας πρέπει να είναι τοποθετημένα σε αύξουσα σειρά.
- Το αποτέλεσμα να περιέχει όλα τα στοιχεία της αρχικής λίστας, μόνο που είναι σε διαφορετική σειρά.

Οι αλγόριθμοι ταξινόμησης που χρησιμοποιούνται στην πληροφορική ταξινομούνται με βάση:

- Την υπολογιστική πολυπλοκότητα (worst, average and best behaviour) των συγκρίσεων των στοιχείων από την άποψη του μεγέθους της λίστας (n). Για χαρακτηριστικούς αλγόριθμους ταξινόμησης η καλή συμπεριφορά είναι $O(n \log n)$ και η κακή συμπεριφορά είναι $\Omega(n^2)$. Η ιδανική συμπεριφορά για μία ταξινόμηση είναι $O(n)$. Αλγόριθμοι ταξινόμησης οι οποίοι χρησιμοποιούν μόνο ένα αφηρημένο κλειδί για λειτουργίες σύγκρισης χρειάζονται πάντα τουλάχιστον $\Omega(n \log n)$ συγκρίσεις κατά μέσον όρο.

- Την υπολογιστική πολυπλοκότητα των ανταλλαγών (για "in place" αλγορίθμους).
- Τη χρήση μνήμης και άλλων υπολογιστών πόρων. Ειδικότερα, μερικοί αλγόριθμοι ταξινόμησης είναι "in place", έτσι ώστε να έχουν χωρική πολυπλοκότητα μόνο $O(1)$ ή $O(\log n)$ πέρα από τα στοιχεία που ταξινομούνται, ενώ άλλοι πρέπει να δημιουργήσουν τις βοηθητικές θέσεις για τα στοιχεία που αποθηκεύονται προσωρινά.
- Την επαναληπτικότητα. Μερικοί αλγόριθμοι είναι είτε επαναλαμβανόμενοι είτε μη επαναλαμβανόμενοι, ενώ άλλοι μπορούν να είναι και τα δύο.

Στη συγκεκριμένη εργασία θα ασχοληθούμε με τους αλγορίθμους HeapSort και WeakHeapSort. Η μέθοδος WeakHeapSort προτάθηκε από τον Dutton το 1993 (Dutton, 1993). Ο αλγόριθμος αυτός χρησιμοποιεί μία δομή δεδομένων, που ονομάζονται WeakHeaps (Dutton, 1992). Οι σωροί αυτοί μπορούν να υλοποιηθούν με $n - 1$ συγκρίσεις. Ο αριθμός συγκρίσεων χειρότερης περίπτωσης του αλγορίθμου είναι $n \log n - 2 \log n + n - \log n < n \log n + 0.1n$ (Edelkamp et al., 2000). Στόχος αυτής της εργασίας είναι να αναλύσει σε βάθος το θεωρητικό υπόβαθρο του αλγορίθμου WeakHeapSort.

1.2 Διάρθρωση της Εργασίας

Η εργασία αυτή έχει οργανωθεί ως εξής: στο κεφάλαιο 2 γίνεται μια ιστορική αναδρομή των αλγορίθμων ταξινόμησης και παρουσιάζονται οι πιο ευρέως διαδεδομένοι από αυτούς. Στο κεφάλαιο 3 παρουσιάζονται οι δομές δεδομένων σωρός και weak heap που χρησιμοποιούνται από τους αλγόριθμους heapsort και weakheapsort, που παρουσιάζονται αναλυτικά στα κεφάλαια 4 και 5. Για κάθε αλγόριθμο παρουσιάζεται ο ψευδοκώδικας του, ένα αναλυτικό παράδειγμα και η ανάλυση πολυπλοκότητάς του.

Στο κεφάλαιο 6 παρουσιάζεται η υπολογιστική μελέτη που αναπτύχθηκε για τους δύο αυτούς αλγορίθμους, ενώ στο κεφάλαιο 7 ακολουθούν τα συμπεράσματα της εργασίας.

1.3 Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επιβλέπον καθηγητή μου κύριο Σαμαρά Νικόλαο καθηγητή του Τμήματος Εφαρμοσμένης Πληροφορικής του Πανεπιστημίου Μακεδονίας για τη βοήθεια , τη συμπαράσταση , την κατανόηση και την καθοδήγησή του όλη αυτή την περίοδο που ήταν πολύ δύσκολη για εμένα. Επίσης θα ήθελα να ευχαριστήσω την οικογένειά μου , τους φίλους μου και κυρίως τα παιδιά μου για την αμέριστη συμπαράσταση και υπομονή τους αλλά και έναν ακόμη άνθρωπο που με βοήθησε πολύ και δε βρίσκεται πλέον στη ζωή τον κύριο Παπαρρίζο Κωνσταντίνο , τον άνθρωπο που με έκανε να αγαπήσω την επιστήμη των αλγορίθμων.

2. Αλγόριθμοι Ταξινόμησης

2.1 Εισαγωγή

Υπάρχουν πολλοί αλγόριθμοι ταξινόμησης οι οποίοι μάλιστα διαιρούνται σε κατηγορίες. Οι πιο γνωστές κατηγορίες είναι οι:

1. Exchange sorts
 - a. Bubble sort
 - b. Quick sort
2. Selection sorts
 - a. Selection sort
 - b. Heap sort
3. Insertion sorts
 - a. Insertion sort
 - b. Shell sort
4. Merge sorts
 - a. Merge sort
5. Distribution sorts
 - a. Bucket sort
 - b. Counting sort
 - c. Radix sort

Οι αλγόριθμοι της πρώτης κατηγορίας βασίζονται στην ανταλλαγή δύο στοιχείων, όταν αυτό κριθεί απαραίτητο από τη ροή του προγράμματος, για να επέλθει τελικά η διάταξη. Οι αλγόριθμοι της δεύτερης κατηγορίας βασίζονται στην επιλογή κάθε φορά του μεγαλύτερου/μικρότερου στοιχείου από όσα έμειναν αταξινομήτα, ώστε να

τοποθετηθεί στη σωστή θέση μέχρι να ταξινομηθούν όλα. Οι αλγόριθμοι της τρίτης κατηγορίας, εισαγάγουν κάθε φορά ένα στοιχείο στη θέση που θα έχει στην τελική διάταξη. Οι αλγόριθμοι της τέταρτης κατηγορίας χρησιμοποιούν την τεχνική του διαίρει και βασίλευε δηλαδή, διαιρούν την αρχική λίστα στοιχείων σε μικρότερες, τις διατάσσουν και στη συνέχεια τις ενοποιούν. Τέλος, οι αλγόριθμοι της πέμπτης κατηγορίας αξιοποιούν πληροφορία σχετική με την κατανομή των τιμών που έχουν τα στοιχεία.

Περαιτέρω, οι αλγόριθμοι ταξινόμησης κατατάσσονται σε ευσταθείς και σε μη-ευσταθείς. Ένας αλγόριθμος ταξινόμησης χαρακτηρίζεται ευσταθής όταν στην τελική διατεταγμένη λίστα, στοιχεία τα οποία είχαν ίδια τιμή, διατηρούν πάντα τη σχετική τους διάταξη, ενώ μη-ευσταθής είναι ένας αλγόριθμος όταν η σχετική διάταξη ίδιων στοιχείων δεν διατηρείται κατ' ανάγκη. Οι περισσότεροι αλγόριθμοι, μπορούν να τροποποιηθούν ώστε να γίνουν ευσταθείς.

Ένας άλλος διαχωρισμός είναι η απαίτηση των αλγορίθμων σε μνήμη. Συνοψίζοντας όλα τα παραπάνω, μπορούμε να κατασκευάσουμε τον πίνακα 1 (Cormen et al., 1990):

Πίνακας 1: Συγκεντρωτικός Πίνακας Αλγορίθμων Ταξινόμησης

Αλγόριθμος	Πολυπλοκότητα καλύτερης περίπτωσης	Πολυπλοκότητα μέσης περίπτωσης	Πολυπλοκότητα χειρότερης περίπτωσης	Απαιτήσεις μνήμης	Ευσταθής	Κατηγορία
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ναι	Exchange
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	Εξαρτάται	Exchange
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Όχι	Selection
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Όχι	Selection
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Ναι	Insertion

Sort						
Shell Sort	$O(n)$	Εξαρτάται	$O(n \log^2 n)$	$O(n)$	Όχι	Insertion
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Ναι	Merge
Bucket Sort	-	$O(n+k)$	$O(n^2-k)$	$O(n-k)$	Ναι	Distribution
Counting Sort	$O(n+r)$	$O(n+r)$	$O(n+r)$	$O(n+r)$	Ναι	Distribution
Radix Sort	$O(nk/d)$	$O(nk/d)$	$O(nk/d)$	$O(n)$	Ναι	Distribution

Όπου:

- n είναι το πλήθος των στοιχείων προς ταξινόμηση,
- k είναι το μέγεθος των στοιχείων και
- r είναι το εύρος των στοιχείων.

Παρακάτω, γίνεται αναφορά στους πιο σημαντικούς αλγορίθμους ταξινόμησης. Ο ενδιαφερόμενος μπορεί να επισκεφθεί την σελίδα (Sorting Algorithms, 2012) για μια γραφική αναπαράσταση των περισσότερων από τους αλγορίθμους ταξινόμησης, ενώ για περισσότερες λεπτομέρειες σχετικά με τους αλγορίθμους ταξινόμησης μπορεί να συμβουλευθεί τα (Knuth, 1998; Παπαρρίζος, 2008).

2.2 Bubble Sort

Ταξινόμηση φυσαλίδας (bubble sort) είναι το όνομα ενός αλγόριθμου ταξινόμησης, ο οποίος λειτουργεί συγκρίνοντας βηματικά τα στοιχεία μιας λίστας και εναλλάσσοντας τα ώστε να βρεθούν σε σωστή σειρά. Τα βήματα επαναλαμβάνονται μέχρι να ταξινομηθεί ολόκληρη η λίστα. Το όνομα του αλγόριθμου προέρχεται από τον τρόπο ταξινόμησης: τα μεγαλύτερα στοιχεία κατευθύνονται προς το τέλος, όπως οι φυσαλίδες που αναδύονται στην επιφάνεια. Αν και απλός, ο αλγόριθμος φυσαλίδας είναι

πολύ αναποτελεσματικός. Η πολυπλοκότητα καλύτερης περίπτωσης του συγκεκριμένου αλγόριθμου είναι $O(n)$ και της χειρότερης περίπτωσης $O(n^2)$, όπου n είναι το μέγεθος της λίστας.

Η ταξινόμηση φυσαλίδας έχει τα εξής χαρακτηριστικά:

- απλή υλοποίηση
- αποτελεσματικός για μικρά σύνολα δεδομένων
- ευσταθής (stable)
- προσαρμοστική (adaptive)
- επιτόπιος (in-place)
- βελτιώνεται η απόδοση του όταν τα στοιχεία είναι ήδη ταξινομημένα
- γενική μέθοδος ανταλλαγής
- πολυπλοκότητα χειρότερης περίπτωσης $O(n^2)$

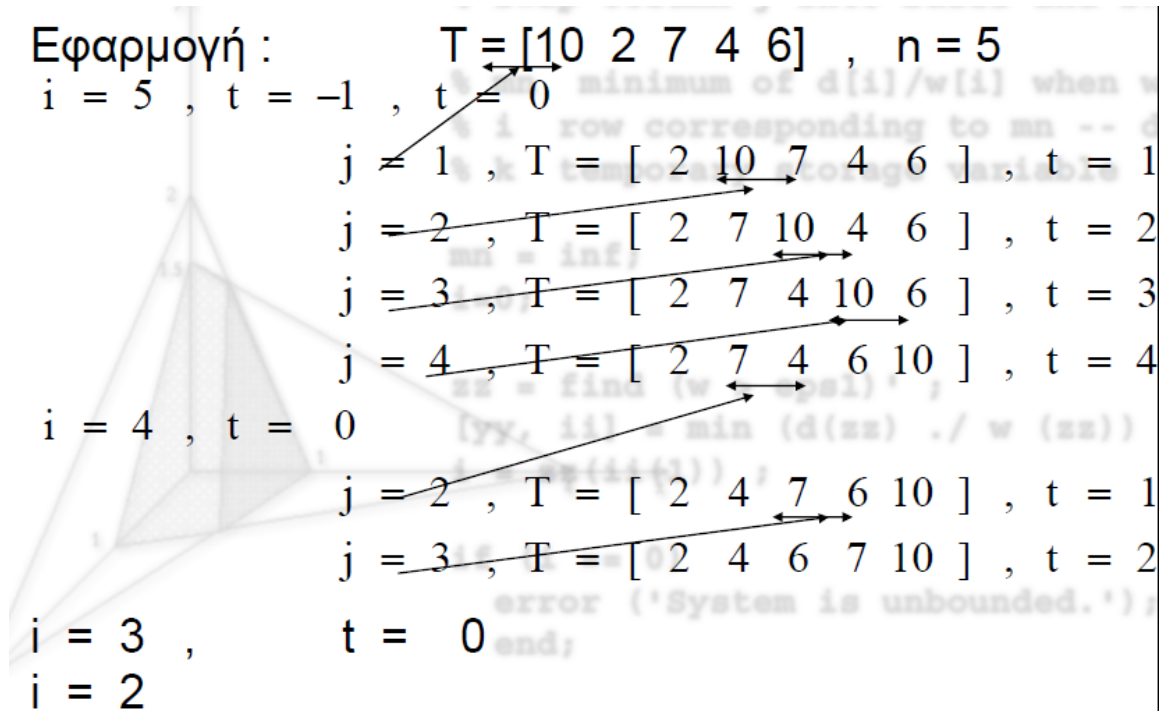
Στην παρακάτω εικόνα φαίνεται ο ψευδοκώδικας του αλγόριθμου:

Αλγόριθμος: bubblesort
Δεδομένα: T, n
Αποτελέσματα: T

```
1,  i ← n, t ← -1
2,  όσο i ≥ 2 και t ≠ 0
3,      t ← 0
4,      για j από 1 μέχρι i - 1
5,          αν T(j) > T(j+1)
6,              [T(j), T(j+1)] ← εναλλαγή(T(j), T(j+1))
7,              t ← t+1
8,      i ← i - 1
9,  
```

Εικόνα 1: Ψευδοκώδικας αλγόριθμου Bubble Sort

Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα ταξινόμησης ενός πίνακα 5 στοιχείων σε αύξουσα σειρά σύμφωνα με τα βήματα του αλγορίθμου που παρουσιάστηκαν.



Εικόνα 2: Παράδειγμα ταξινόμησης με τον αλγόριθμο Bubble Sort

2.3 Selection Sort

Ένας άλλος απλός αλγόριθμος ταξινόμησης είναι ο αλγόριθμος επιλογής (selection sort) που έχει βελτιωμένη απόδοση σε σχέση με τον αλγόριθμο φυσαλίδας. Η ταξινόμηση ευθείας επιλογής έχει πολυπλοκότητα $\Theta(n^2)$ γεγονός που την κάνει αναποτελεσματική σε μεγάλες λίστες. Ανήκει στην κατηγορία των αλγορίθμων ταξινόμησης που είναι in-place, δηλαδή για την ταξινόμηση ενός πίνακα δεν χρησιμοποιούν κάποιο βοηθητικό πίνακα ίδιου τύπου αλλά εκτελούν την ταξινόμηση κάνοντας χρήση των ίδιων θέσεων του πίνακα. Ο αλγόριθμος επιλογής βασίζεται στις ακόλουθες δυο αρχές:

- i) επιλογή του στοιχείου με το ελάχιστο κλειδί (για αύξουσα σειρά)

ii) ανταλλαγή αυτού του στοιχείου με το πρώτο στοιχείο του πίνακα.

Αυτές οι λειτουργίες επαναλαμβάνονται για τα υπόλοιπα $n-1$ στοιχεία μέχρι στο τέλος να απομείνει το μεγαλύτερο στοιχείο.

Η ταξινόμηση επιλογής έχει τα εξής χαρακτηριστικά:

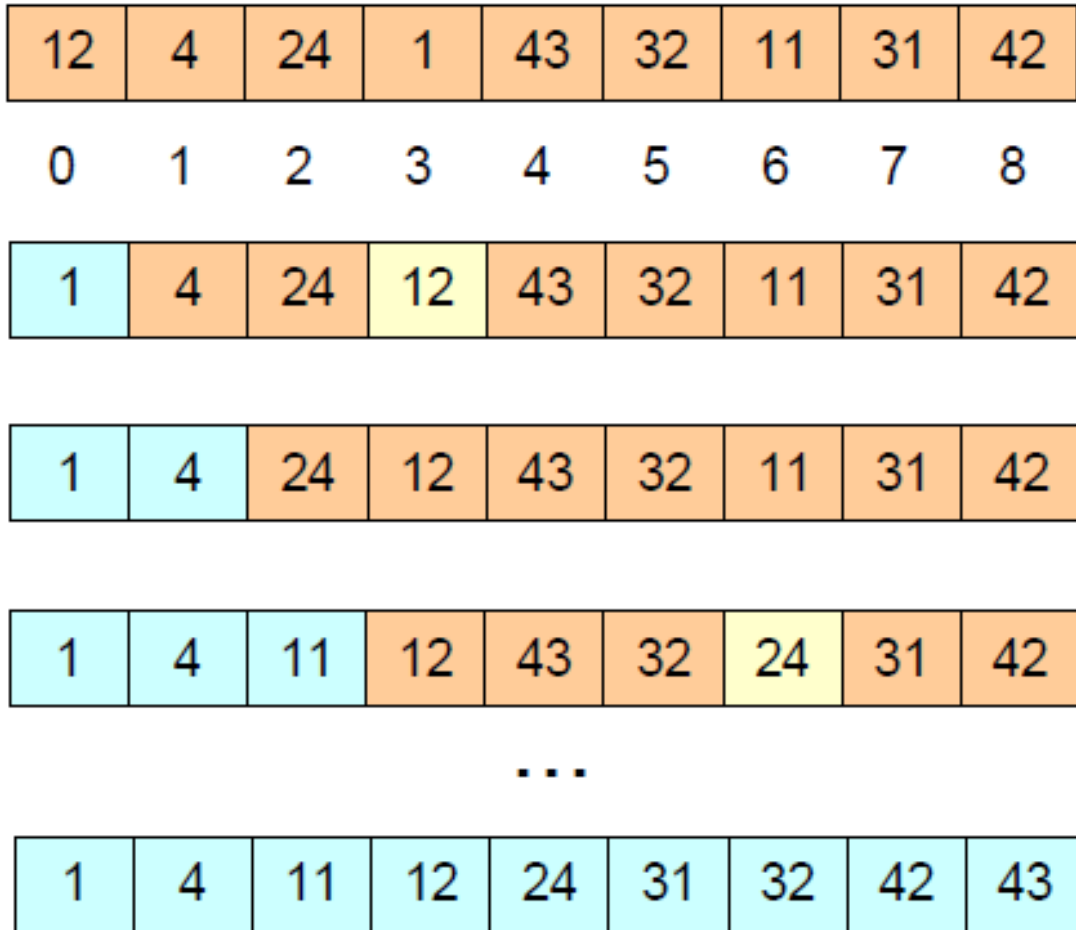
- απλή υλοποίηση
- αποτελεσματικός για μικρά σύνολα δεδομένων
- ασταθής (unstable)
- επιτόπιος
- γενική μέθοδος επιλογής

Στην παρακάτω εικόνα φαίνεται ο ψευδοκώδικας του αλγόριθμου:

	Αλγόριθμος: selectsort Δεδομένα: T, n Αποτελέσματα: T
1	για i από 1 μέχρι n-1
2	[min, index] ← min1(T(i:n))
3	T(i+index-1) ← T(i)
4	T(i) ← min

Εικόνα 3: Ψευδοκώδικας αλγόριθμου Selection Sort

Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα ταξινόμησης ενός πίνακα 9 στοιχείων σε αύξουσα σειρά σύμφωνα με τα βήματα του αλγορίθμου που παρουσιάστηκαν.



Εικόνα 4: Παράδειγμα ταξινόμησης με τον αλγόριθμο Selection Sort

2.4 Insertion Sort

Η ταξινόμηση εισαγωγής (insertion sort) είναι ένας πολύ απλός αλγόριθμος ταξινόμησης της κατηγορίας των αλγορίθμων ταξινόμησης με συγκρίσεις. Σύμφωνα με την ταξινόμηση εισαγωγής τα στοιχεία χωρίζονται σχηματικά σε μια ακολουθία προορισμού $table[1], table[2], \dots, table[i-1]$ και σε μια ακολουθία πηγής $table[i], \dots, table[n]$. Σε κάθε βήμα αρχίζοντας με $i=2$ και αυξάνοντας διαδοχικά το i κατά 1, το στοιχείο με δείκτη i λαμβάνεται και μεταφέρεται στην κατάλληλη θέση της ακολουθίας προορισμού. Έτσι σε κάθε βήμα αυξάνεται η ακολουθία προορισμού κατά ένα και μειώνεται η ακολουθία πηγής κατά ένα.

Η λειτουργία της εισαγωγής είναι, όμως, «ακριβή» γιατί απαιτεί τη μετατόπιση των υπόλοιπων στοιχείων μία θέση αριστερά (shift left). Ο εν λόγω αλγόριθμος έχει πολυπλοκότητα καλύτερης περίπτωσης $O(n)$ και χειρότερης περίπτωσης $O(n^2)$.

Είναι λιγότερο αποτελεσματικός σε μεγάλους πίνακες από ότι άλλοι αλγόριθμοι ταξινόμησης όπως η γρήγορη ταξινόμηση, η ταξινόμηση σωρού, η ταξινόμηση συνένωσης αλλά η ταξινόμηση εισαγωγής έχει τα εξής πλεονεκτήματα:

- απλή υλοποίηση
- αποτελεσματικός για μικρά σύνολα δεδομένων
- προσαρμοστικός
- είναι ευσταθής (stable)
- επιτόπιος (in place)
- Γενική μέθοδος εισαγωγή

Στην παρακάτω εικόνα φαίνεται ο ψευδοκώδικας του αλγόριθμου:

Αλγόριθμος: insertsort	
Δεδομένα: T, n	
Αποτελέσματα: T	
1	για i από 2 μέχρι n
2	T(1:i) ← insert(T(1:i-1), T(i))

Εικόνα 5: Ψευδοκώδικας αλγόριθμου Insertion Sort

Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα ταξινόμησης ενός πίνακα 6 στοιχείων σε αύξουσα σειρά σύμφωνα με τα βήματα του αλγορίθμου που παρουσιάστηκαν.

Εφαρμογή. $T = [15 \ 10 \ 7 \ 2 \ 5 \ 4]$

$i = 2$	$T = [15 \ \underline{10} \ 7 \ 2 \ 5 \ 4]$
$i = 3$	$T = [10 \ 15 \ \underline{7} \ 2 \ 5 \ 4]$
$i = 4$	$T = [7 \ 10 \ 15 \ \underline{2} \ 5 \ 4]$
$i = 5$	$T = [2 \ 7 \ 10 \ 15 \ \underline{5} \ 4]$
$i = 6$	$T = [2 \ 5 \ 7 \ 10 \ 15 \ \underline{4}]$
	$T = [2 \ 4 \ 5 \ 7 \ 10 \ 15]$

Εικόνα 6: Παράδειγμα ταξινόμησης με τον αλγόριθμο Insertion Sort

2.5 Radix Sort

Η πιο γνωστή μη γενική μέθοδος ταξινόμησης. Συγκρίνει τα στοιχεία κομμάτι-κομμάτι και τα χωρίζει σε ομάδες. Στο τέλος ο χωρισμός έχει γίνει έτσι ώστε τα στοιχεία να είναι ταξινομημένα. Δηλαδή οι δυαδικοί αριθμοί είναι ακολουθίες από bit, τα αλφαριθμητικά είναι ακολουθίες χαρακτήρων και οι δεκαδικοί αριθμοί είναι ακολουθίες ψηφίων.

Υπάρχουν δύο παραλλαγές αυτού του τύπου του αλγορίθμου: η πρώτη παραλλαγή ταξινομεί τους αριθμούς αρχίζοντας από το πιο σημαντικό ψηφίο και η δεύτερη παραλλαγή από το λιγότερο σημαντικό ψηφίο. Ο αλγόριθμος ταξινόμησης βάσης έχει θεωρητική πολυπλοκότητα $O(n*k)$, όπου n είναι ο αριθμός των στοιχείων της λίστας και k το μέγεθος σε bit της αναπαράστασης των αριθμών.

Η ταξινόμηση βάσης έχει τα εξής χαρακτηριστικά:

- ευσταθής

- μνήμη $O(n)$
- πολυπλοκότητα $O(n*k)$

Στην παρακάτω εικόνα φαίνεται ο ψευδοκώδικας του αλγόριθμου:

Αλγόριθμος : radixsort	
Δεδομένα : T, n, d	
Αποτελέσματα : T	
1	για i από 1 μέχρι d
2	$x \leftarrow \lfloor \text{mod}(T, 10^i) / 10^{i-1} \rfloor$
3	$[x, \text{theseis}] \leftarrow \text{countsort2}(x+1, n, 10)$
4	$T \leftarrow T(\text{theseis})$

Εικόνα 7: Ψευδοκώδικας αλγόριθμου Radix Sort

2.6 Count Sort

Αυτή η μέθοδος βασίζεται στο εύρος των τιμών των στοιχείων που θέλουμε να ταξινομήσουμε. Μετράει πόσες φορές εμφανίζεται κάθε τιμή στον πίνακα και στο τέλος απλά τον δημιουργεί από την αρχή ταξινομημένο. Είναι πολύ βολική μέθοδος για πίνακες ακεραίων με μικρό εύρος τιμών. Ο αλγόριθμος που παρουσιάζεται εδώ αφορά ακέραιους και το k στην πολυπλοκότητα είναι η διαφορά μεταξύ μέγιστου και ελάχιστου στοιχείου.

Ο Count sort έχει τα εξής χαρακτηριστικά:

- πολύ βολική μέθοδος για πίνακες ακεραίων με μικρό εύρος τιμών
- ευσταθής
- μνήμη $O(k)$
- πολυπλοκότητα μέσης και χειρότερης περίπτωσης $O(n+k)$

Στην παρακάτω εικόνα φαίνεται ο ψευδοκώδικας του αλγόριθμου:

Αλγόριθμος : countsort1

Δεδομένα : S, n, m

Αποτελέσματα : T

1	για i από 1 μέχρι n
2	$C(S(i)) \leftarrow C(S(i)) + 1$
3	για i από 2 μέχρι m
4	$C(i) \leftarrow C(i) + C(i - 1)$
5	για i από n μέχρι 1 βήμα $- 1$
6	$T(C(S(i))) \leftarrow S(i)$
8	$C(S(i)) \leftarrow C(S(i)) - 1$

Εικόνα 8: Ψευδοκώδικας αλγόριθμου Count Sort

Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα ταξινόμησης ενός πίνακα 8 στοιχείων σε αύξουσα σειρά σύμφωνα με τα βήματα του αλγορίθμου που παρουσιάστηκαν.

Εφαρμογή: $S = [5 \ 1 \ 5 \ 2 \ 1 \ 2 \ 1 \ 5]$, $m = 5$, $C = [3 \ 2 \ 0 \ 0 \ 3]$ $C = [3 \ 5 \ 5 \ 5 \ 8]$

i	$S(i)$	$C(S(i))$	T	C
-	-	-	[0 0 0 0 0 0 0 0]	[3 5 5 5 8]
8	5	8	[0 0 0 0 0 0 0 5]	[3 5 5 5 7]
7	1	3	[0 0 1 0 0 0 0 5]	[2 5 5 5 7]
6	2	5	[0 0 1 0 2 0 0 5]	[2 4 5 5 7]
5	1	2	[0 1 1 0 2 0 0 5]	[1 4 5 5 7]
4	2	4	[0 1 1 2 2 0 0 5]	[1 3 5 5 7]
3	5	7	[0 1 1 2 2 0 6 5]	[1 3 5 5 6]
2	1	1	[1 1 1 2 2 0 6 5]	[0 3 5 5 6]
1	5	6	[1 1 1 2 2 5 5 5]	[0 3 5 5 5]

Εικόνα 9: Παράδειγμα ταξινόμησης με τον αλγόριθμο Count Sort

2.7 Bucket Sort

Ο αλγόριθμος bucket sort έχει κατώτερο μέσο χρόνο από το κάτω φράγμα του $\delta(n \log n)$ για ταξινόμηση βασιζόμενη σε συγκρίσεις. Αυτό οφείλεται στο ότι ο αλγόριθμος θεωρεί ότι τα n στοιχεία προς ταξινόμηση κατανέμονται ομοιόμορφα στο διάστημα $[a, b)$. Αυτός ο αλγόριθμος καλείται bucket sort και εκτελείται ως εξής: Το διάστημα $[a, b)$ διαιρείται σε m ίσα υπό-διαστήματα που καλούνται κάδοι (buckets). Κάθε στοιχείο τοποθετείται στο κατάλληλο κάδο. Επειδή τα n στοιχεία κατανέμονται ομοιόμορφα στο διάστημα $[a, b)$, το πλήθος των στοιχείων σε κάθε κάδο είναι περίπου n/m .

Ο Bucket sort έχει τα εξής χαρακτηριστικά:

- ευσταθής
- επιπλέον μνήμη $O(n \cdot k)$
- πολυπλοκότητα χειρότερης περίπτωσης $O(n^2 \cdot k)$

Στην παρακάτω εικόνα φαίνεται ο ψευδοκώδικας του αλγόριθμου:

Αλγόριθμος: bucketsort

Δεδομένα: A, n

Αποτελέσματα: T

- 1 για i από 1 μέχρι n
- 2 κάνε εισαγωγή του $A(i)$ στο μπουκέτο $B(\lfloor n \cdot A(i) \rfloor)$
- 3 για i από 1 μέχρι $\text{πλήθος}(B(i))$
- 4 ταξινόμησε το μπουκέτο $B(i)$ με εισαγωγή
- 5 Συνένωσε τα μπουκέτα σε ένα ταξινομημένο διάνυσμα T

Εικόνα 10: Ψευδοκώδικας αλγόριθμου Bucket Sort

3. Σωροί και Δομή WeakHeap

3.1 Δένδρα

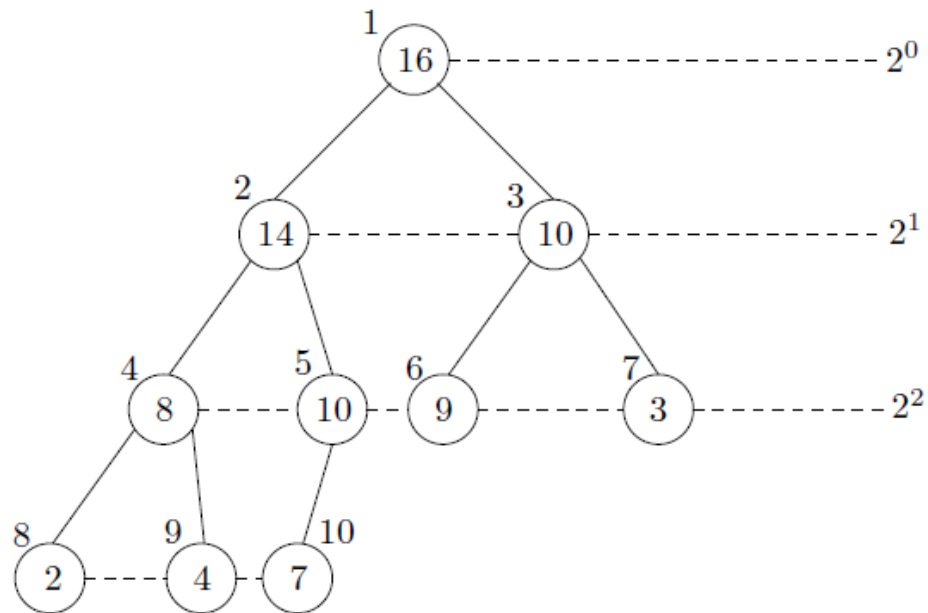
Ένα δένδρο είναι μια δομή δεδομένων η οποία μπορεί να είναι είτε ένα ατομικό δένδρο (ένα φύλλο), είτε ένας κόμβος και μια ακολουθία από υποδένδρα. Οι κόμβοι ενός δένδρου διακρίνονται σε εσωτερικούς κόμβους και φύλλα. Τα φύλλα είναι κόμβοι οι οποίοι δεν έχουν παιδιά. Επιπρόσθετα, το βάθος ενός κόμβου είναι το μήκος του μονοπατιού το οποίο ενώνει τον κόμβο αυτό με τη ρίζα. Κατά σύμβαση θεωρούμε πάντα ότι η ρίζα έχει βάθος 0. Το ύψος ενός κόμβου είναι το μήκος του μεγαλύτερου μονοπατιού που ενώνει τον κόμβο με τα φύλλα. Το ύψος του δένδρου είναι το ύψος της ρίζας.

Μια ειδική κατηγορία δένδρων είναι τα δυαδικά δένδρα στα οποία κάθε κόμβος έχει το πολύ δύο παιδιά. Αν διαιρέσουμε το σύνολο των κόμβων ενός τέτοιου δένδρου σε γραμμές ακολουθώντας τα βάθη τους, θα έχουμε μια γραμμή (τη γραμμή 0) η οποία περιέχει τη ρίζα (2^0), μια γραμμή (τη γραμμή 1) η οποία περιέχει το πολύ 2 κόμβους (2^1) κ.ο.κ. Γενικά η γραμμή i περιέχει το πολύ 2^i κόμβους. Σχεδόν πλήρες καλείται ένα δυαδικό δένδρο όταν όλες οι γραμμές, εκτός ίσως από την τελευταία, περιέχουν το μέγιστο αριθμό κόμβων (δηλαδή 2^i). Επιπλέον ισχύει μια σειρά από ιδιότητες όπως:

- Τα φύλλα της τελευταίας γραμμής είναι όλα αριστερά
- Τα φύλλα βρίσκονται όλα στην τελευταία και ενδεχομένως στην προτελευταία γραμμή
- Οι εσωτερικοί κόμβοι είναι όλοι δυαδικοί, εκτός από το δεξιότερο της προτελευταίας γραμμής, ο οποίος μπορεί να μην έχει δεξιό παιδί.

Για την αρίθμηση των κόμβων χρησιμοποιούμε τους παρακάτω κανόνες: Κάθε κόμβος έχει τον πατέρα του στη θέση $\lfloor i/2 \rfloor$, το αριστερό παιδί του κόμβου i , είναι ο κόμβος $2i$ και το δεξιό παιδί του κόμβου i είναι το $2i+1$. Επίσης σε ένα δυαδικό δένδρο με m κόμβους και ύψος n ισχύει:

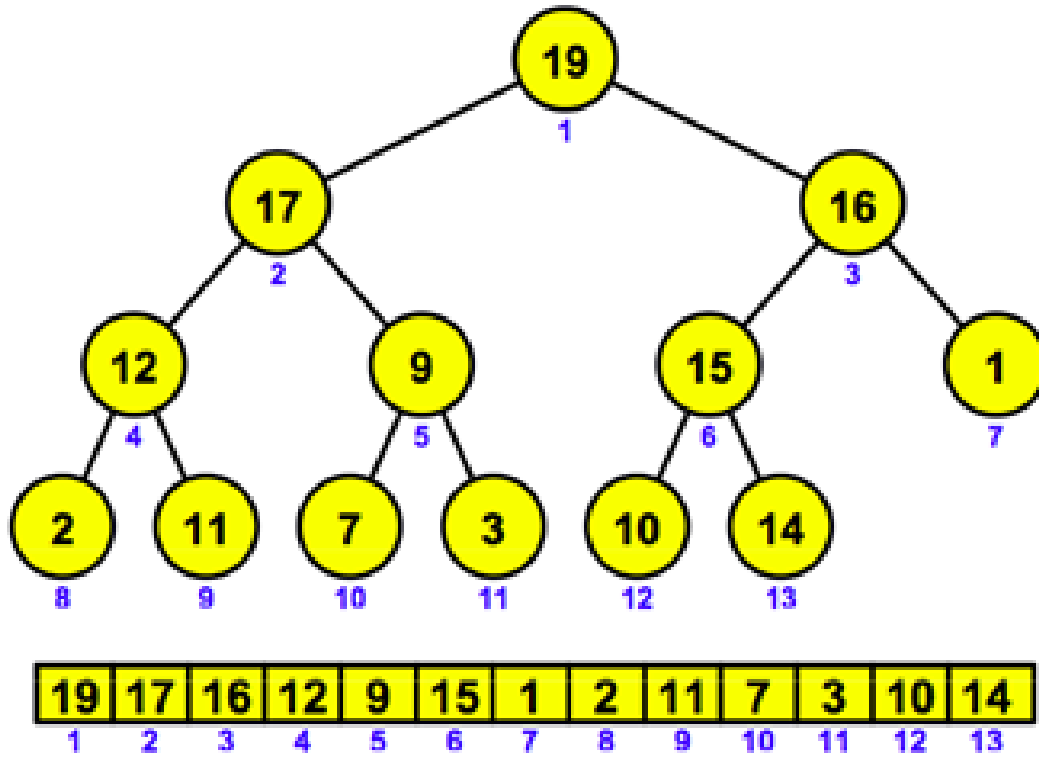
$$\log_2 m \leq n \leq m-1$$



Εικόνα 11: Σχεδόν πλήρες δυαδικό δέντρο

3.2 Σωροί

Ο σωρός είναι μια δενδρική δομή και χρησιμοποιείται για την δημιουργία ουρών προτεραιότητας (priority queues) (Atkinson et al., 1986). Η ρίζα του δέντρου περιλαμβάνει το μικρότερο-μεγαλύτερο στοιχείο του, αναλόγως αν έχουμε σωρό ελαχίστων ή μεγίστων. Τα επόμενα δύο στοιχεία του δέντρου είναι τα παιδιά του. Γενικότερα αν ο πατέρας είναι στη θέση i τα παιδιά του θα είναι στην θέση $2i$ (αριστερό παιδί) και $2i+1$ (δεξί παιδί) αντίστοιχα. Αν i η θέση ενός παιδιού $i/2$ είναι η θέση του πατέρα του. Κάθε σωρός με n στοιχεία έχει ύψος $\log_2 n$. Ο σωρός, όπως και τα δέντρα γενικότερα, μπορεί να υλοποιηθεί με πίνακα, στον οποίο εισάγονται τα κλειδιά του σωρού από αριστερά προς τα δεξιά και από πάνω προς τα κάτω. Στην εικόνα 1 φαίνεται ένας σωρός και η αντίστοιχη υλοποίησή του σε πίνακα:



Εικόνα 12: Υλοποίηση σωρού με πίνακα

Υπάρχουν δύο είδη σωρών:

- Οι σωροί μεγίστου (maxheap)

Σωρός μεγίστων είναι ένα δένδρο, το οποίο ικανοποιεί δύο συνθήκες:

- Η τιμή του κλειδιού κάθε κόμβου είναι μεγαλύτερη ή ίση από τις τιμές των κλειδιών των παιδιών του.
- Είναι ένα συμπληρωμένο δένδρο, που σημαίνει ότι μπορεί να προκύψει από ένα πλήρες δένδρο αφαιρώντας έναν αριθμό στοιχείων από το τέλος.

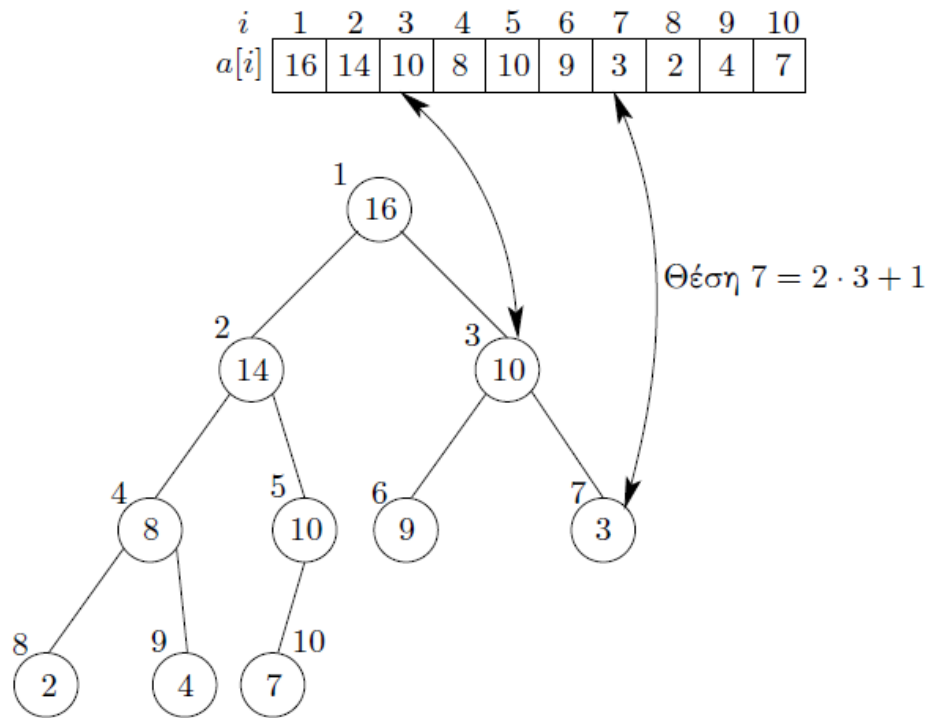
- Οι σωροί ελαχίστων (minheap)

Σωρός ελαχίστων είναι ένα δένδρο, το οποίο ικανοποιεί δύο συνθήκες:

- Η τιμή του κλειδιού κάθε κόμβου είναι μικρότερη ή ίση από τις τιμές των κλειδιών των παιδιών του.

- Είναι ένα συμπληρωμένο δέντρο, που σημαίνει ότι μπορεί να προκύψει από ένα πλήρες δέντρο αφαιρώντας έναν αριθμό στοιχείων από το τέλος.

Ένα παράδειγμα σωρού φαίνεται στην παρακάτω εικόνα.



Εικόνα 13: Ένας σωρός και η υλοποίησή του με πίνακα

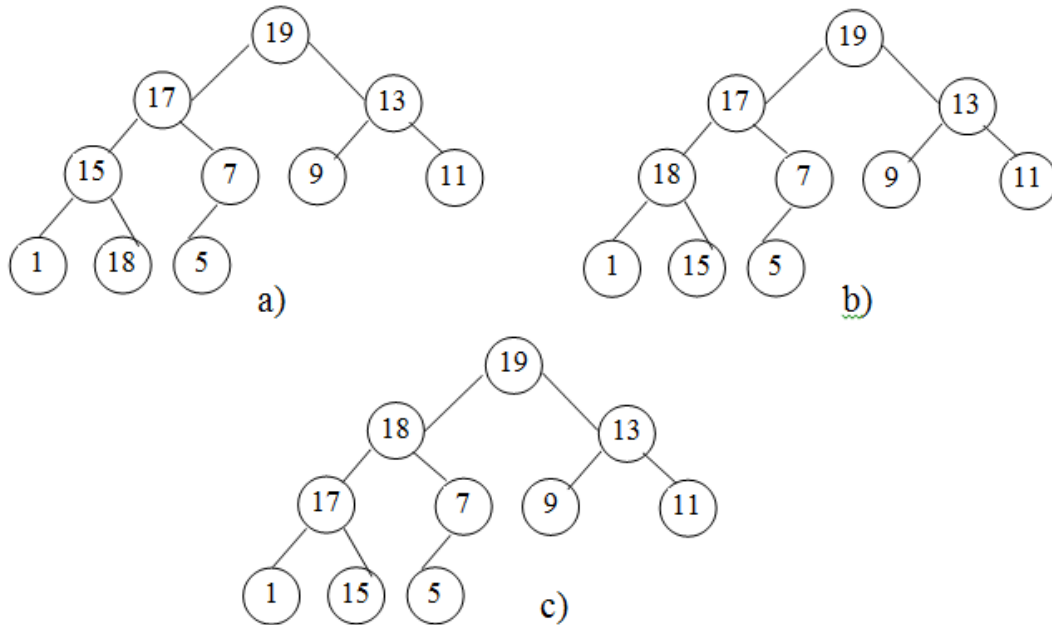
Η αναπαράσταση με σωρό μας επιτρέπει να κάνουμε τις πράξεις της αναζήτησης του μέγιστου στοιχείου, της εισαγωγής και της διαγραφής σε χρόνο $O(\log n)$.

Η αποκατάσταση της ιδιότητας της σωρού, όταν αυτή δεν ισχύει εξ αιτίας της τιμής κάποιου κλειδιού, μπορεί να γίνει με δυο βασικές λειτουργίες την άνοδο και την κάθοδο. Με τις λειτουργίες αυτές μπορούμε να περιγράψουμε πολλές άλλες πράξεις, όπως για παράδειγμα, την εισαγωγή νέου στοιχείου, τη διαγραφή στοιχείου, κ.λπ. Διευκρινίζουμε ότι όταν λέμε ότι η ιδιότητα της σωρού δεν ισχύει εξ αιτίας του κλειδιού $H(i)$ εννοούμε το εξής. Πριν ο κόμβος i πάρει το κλειδί $H(i)$ είχε ένα άλλο κλειδί το οποίο ικανοποιούσε την ιδιότητα της σωρού.

Πίνακας 2: Ψευδοκώδικας μεθόδου heapifyup

Αλγόριθμος : heapifyup	
Δεδομένα : H, i	
Αποτελέσματα : H	
1	$k \leftarrow \lfloor i/2 \rfloor$
2	όσο $k \geq 1$ και $H(k) < H(i)$
3	$[H(k), H(i)] \leftarrow \text{εναλλαγή}(H(k), H(i))$
4	$i \leftarrow k$
5	$k \leftarrow \lfloor i/2 \rfloor$

Η άνοδος εφαρμόζεται στην περίπτωση, που το κλειδί $H(i)$ του κόμβου i είναι μεγαλύτερο του κλειδιού $H(\lfloor i/2 \rfloor)$ του πατέρα του. Αν εναλλάξουμε τις θέσεις των κλειδιών $H(i)$ και $H(\lfloor i/2 \rfloor)$, τότε το υποδένδρο με ρίζα τον κόμβο $H(\lfloor i/2 \rfloor)$ είναι σωρός. Τώρα είναι δυνατόν να παραβιάζεται η ιδιότητα της σωρού εξαιτίας του κλειδιού του κόμβου $\lfloor i/2 \rfloor$, που είναι μεγαλύτερο από το κλειδί του πατέρα του $H(\lfloor \lfloor i/2 \rfloor / 2 \rfloor) = H(\lfloor i/4 \rfloor)$. Για να αποκατασταθεί η ιδιότητα της σωρού, πρέπει να εναλλαγεί το κλειδί του κόμβου $\lfloor i/2 \rfloor$ με το κλειδί του πατέρα του. Η λειτουργία της ανόδου είναι τώρα προφανής και περιγράφεται ως κώδικας, heapifyup. Για παράδειγμα, αντικαθιστώντας στην προηγούμενη σωρό $H H(9) = 18$ κατασκευάζεται η σωρός της παρακάτω εικόνας. Εφαρμόζοντας στη συνέχεια την πράξη heapifyup παράγονται διαδοχικά οι σωροί b) και c) της παρακάτω εικόνας. Βλέπουμε ότι το κλειδί $H(9) = 18$ ανέρχεται ένα ένα τους κόμβους του δρόμου που συνδέει τη ρίζα με τον κόμβο $i = 9$.



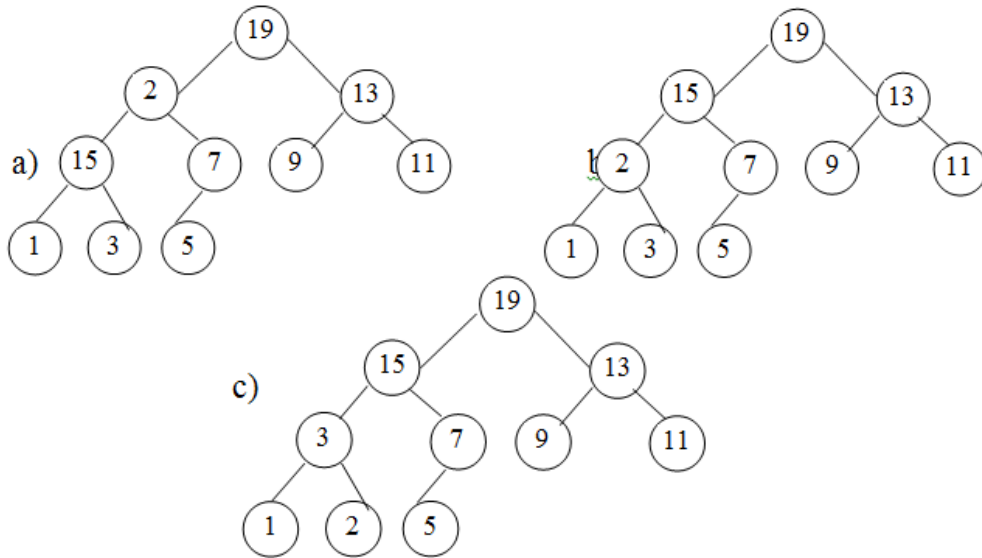
Εικόνα 14: Παράδειγμα της πράξης της ανόδου.

Έστω ότι η πράξη της ανόδου εφαρμόζεται στο κλειδί $H(i)$. Επειδή σε κάθε επανάληψη του βρόχου του όσο τίθεται $i \leftarrow \lfloor i/2 \rfloor$, το πλήθος επαναλήψεων του είναι $\Theta(1)$ στην καλύτερη και $\Theta(\log i)$ στη χειρότερη περίπτωση. Επειδή κάθε εκτέλεση του βρόχου παίρνει χρόνο $\Theta(1)$, η πολυπλοκότητα του αλγόριθμου είναι $\Theta(1, \log i)$. Στην περίπτωση $i = n$ η πολυπλοκότητα του αλγόριθμου είναι $\Theta(1, \log n)$. Στιγμιότυπα χειρότερης και καλλίτερης περίπτωσης υπολογίζονται εύκολα. Επομένως η άνοδος είναι ένας μη ομογενής αλγόριθμος.

Πίνακας 3: Ψευδοκώδικας μεθόδου heapifyup

Αλγόριθμος : heapifydown	
Δεδομένα : H, n, i	
Αποτελέσματα : H	
1	όσο $2i+1 \leq n$
2	$imax \leftarrow 2i$
3	αν $H(imax) < H(2i+1)$
4	$imax \leftarrow 2i+1$
5	αν $H(i) \geq H(imax)$
6	stop
7	αλλιώς
8	$[H(i), H(imax)] \leftarrow \text{εναλλαγή}(H(i), H(imax))$
9	$i \leftarrow imax$
10	αν $2i = n$ και $H(i) < H(n)$
11	$[H(i), H(2i)] \leftarrow \text{εναλλαγή}(H(i), H(2i))$

Η κάθοδος εφαρμόζεται στην περίπτωση, που το κλειδί κάποιου κόμβου i γίνει μικρότερο του μεγαλύτερου κλειδιού των παιδιών του. Φυσικά, τώρα το κλειδί του πατέρα εναλλάσσεται με το μεγαλύτερο κλειδί των παιδιών του. Η πράξη της καθόδου περιγράφεται στον κώδικα heapifydown. Στον κώδικα αυτό η μεταβλητή imax είναι ο δείκτης του παιδιού (του κόμβου i) με το μεγαλύτερο κλειδί. Εντοπίζεται πρώτα ο δείκτης imax του παιδιού με το μεγαλύτερο κλειδί στις γραμμές 2 - 4. Με την επόμενη εντολή αν, γραμμές 5 - 9, σταματούμε τους υπολογισμούς με την εντολή stop (γραμμή 6), αν η ιδιότητα της σωρού έχει αποκατασταθεί ή εναλλάσσουμε τα κλειδιά του πατέρα και του μεγαλύτερου παιδιού του. Η εντολή αν των γραμμών 10 - 11 είναι απαραίτητη γιατί καλύπτει την περίπτωση, που ο τελευταίος κόμβος πατέρας έχει ένα μόνο παιδί, τον κόμβο n . Μια εφαρμογή της καθόδου φαίνεται στην παρακάτω εικόνα με $i = 2$.



Εικόνα 15: Παράδειγμα της πράξεως της καθόδου.

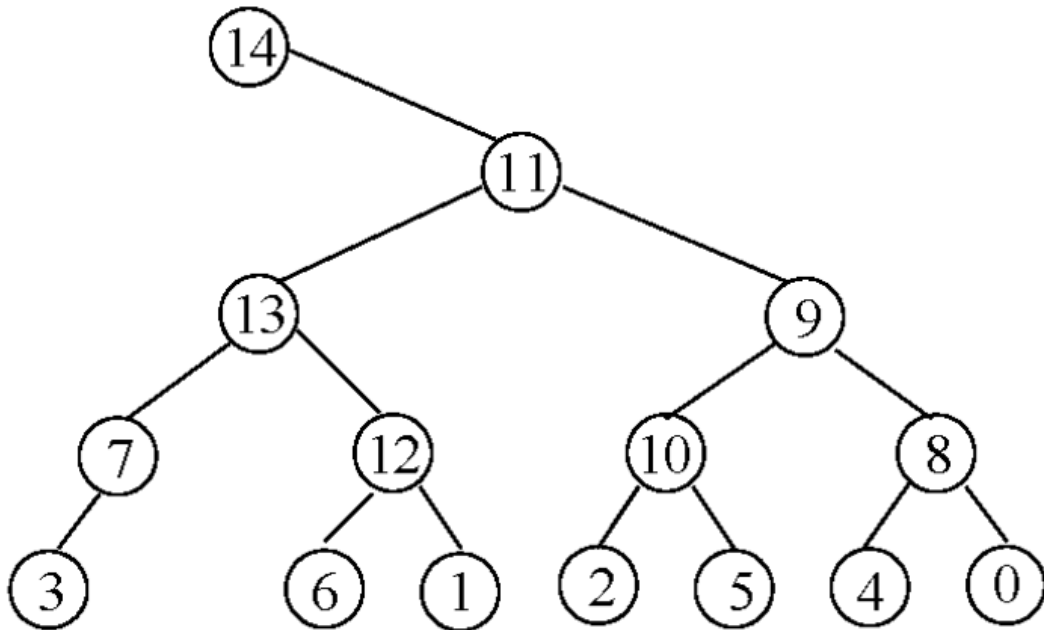
Έστω τώρα ότι ο κόμβος $H(i)$ βρίσκεται σε ύψος k . Επειδή σε κάθε επανάληψη του βρόχου του όσο ο κόμβος i κατέρχεται ένα επίπεδο, η πολυπλοκότητα του αλγορίθμου είναι $\Theta(1, k)$, αφού στην καλύτερη περίπτωση θα κατέλθει ένα και στη χειρότερη $\Theta(k)$ επίπεδα. Προφανώς, όταν $i = 1$, η πολυπλοκότητα του αλγορίθμου είναι $\Theta(1, h) = \Theta(1, \log n)$. Στιγμιότυπα χειρότερης και καλλίτερης περίπτωσης που αποδεικνύουν ότι η κάθοδος είναι ένας μη ομογενής αλγόριθμος υπολογίζονται εύκολα.

3.3 Δομή weakheap

Η δομή Weak-Hear (Dutton, 1992) δημιουργείται χαλαρώνοντας τον όρο των σωρών ως εξής:

- Κάθε κλειδί στο δεξί υποδένδρο κάθε κόμβου είναι μικρότερο ή ίσο με το κλειδί του κόμβου του.
- Η ρίζα δεν έχει κανένα αριστερό παιδί.
- Τα φύλλα βρίσκονται στα τελευταία δύο επίπεδα του δέντρου μόνο

Στην εικόνα 16 φαίνεται η δομή μιας Weak-heap.



Εικόνα 16: Δομή weakheap

Οι διάδοχοι από έναν κόμβο στον αριστερό κλάδο του δεξιού υποδένδρου ονομάζονται grandchildren. Στην προηγούμενη εικόνα τα grandchildren από τη ρίζα είναι το 11, 13, 7, 3.

Η λειτουργία αντιστροφής $Gparent(x)$ ορίζεται ως $Gparent(Parent(x))$ στην περίπτωση που x είναι ένα αριστερό παιδί και $Parent(x)$ αν x είναι ένα δεξί παιδί.

4. Αλγόριθμος Heapsort

4.1 Ψευδοκώδικας Αλγορίθμου

4.1.1 Οι αλγόριθμοι *buildheapup* και *buildheapdown*

Μια πράξη λίγο πιο σύνθετη είναι η κατασκευή της αρχικής σωρού, δηλαδή, η πράξη της μετατροπής ενός οποιουδήποτε διάνυσματος H σε σωρό. Η πράξη αυτή μπορεί να γίνει με δυο τρόπους. Στον πρώτο και πιο απλοϊκό τρόπο ξεκινούμε με το στοιχείο $H(1)$, το οποίο μόνο του είναι μια σωρός. Στη συνέχεια κάνουμε διαδοχικά ανόδους των στοιχείων $H(i)$, όπου $i = 2, 3, \dots, n$ έτσι ώστε κάθε φορά το διάνυσμα $H(1 : i)$ να μετασχηματίζεται σε σωρό. Ο κώδικας αυτού του αλγόριθμου είναι ο *buildheapup*.

Πίνακας 4: Ψευδοκώδικας αλγόριθμου *buildheapup*

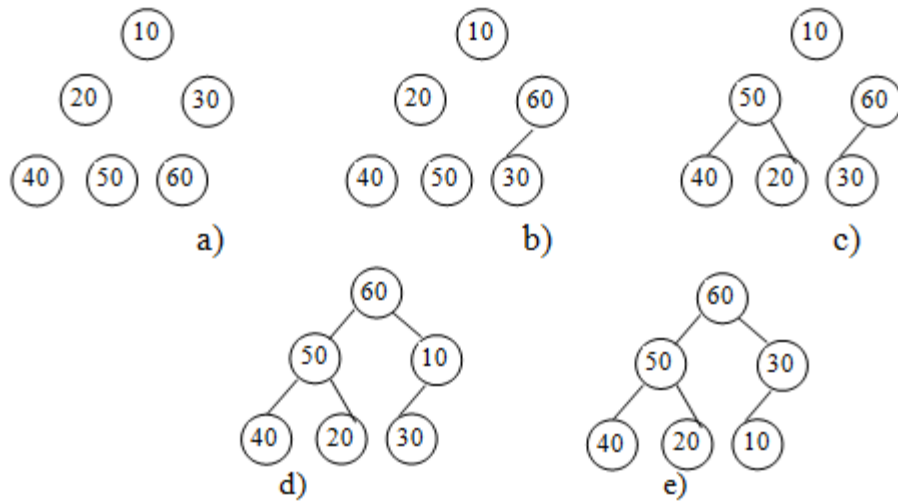
Αλγόριθμος : <i>buildheapup</i>	
Δεδομένα : H, n	
Αποτελέσματα : H	
1	για i από 2 μέχρι n
2	$H(1:i) \leftarrow \text{heapifyup}(H(1:i-1), i)$

Στο δεύτερο πιο προχωρημένο αλλά και πιο αποτελεσματικό τρόπο κάνουμε καθόδους στοιχείων. Απλά κάνουμε διαδοχικά καθόδους των στοιχείων

$$H(\lfloor n/2 \rfloor), H(\lfloor n/2 \rfloor - 1), H(\lfloor n/2 \rfloor - 2), \dots, H(1)$$

Πίνακας 5: Ψευδοκώδικας αλγόριθμου *buildheapdown*

Αλγόριθμος : <i>buildheapdown</i>	
Δεδομένα : H, n	
Αποτελέσματα : H	
1	για i από $\lfloor n/2 \rfloor$ μέχρι 1 βήμα -1
2	$H \leftarrow \text{heapifydown}(H, i)$



Εικόνα 17: Παράδειγμα αλγόριθμου κατασκευής αρχικής σωρού με καθόδους.

Εργαζόμενος επαγωγικά είναι εύκολο να δει κάποιος ότι, πριν γίνει κάθοδος του στοιχείου $H(i)$, το δυαδικό υποδέντρο, που έχει ρίζα το κλειδί $H(i)$ ικανοποιεί την ιδιότητα της σωρού εκτός πιθανόν από το κλειδί της ρίζας $H(i)$. Η κάθοδος του κλειδιού $H(i)$ αποκαθιστά αμέσως την ιδιότητα της σωρού στο υποδέντρο. Επομένως, όταν γίνει $i = 1$, το διάνυσμα H θα μετατραπεί σε σωρό. Ο κώδικας του αλγόριθμου αυτού μπορεί να γραφεί, όπως φαίνεται στον ψευδοκώδικα `buildheapdown`.

Στην Εικόνα 17 φαίνεται παραστατικά η λειτουργία του αλγόριθμου `buildheapdown`. Το αρχικό διάνυσμα $H = [10\ 20\ 30\ 40\ 50\ 60]$ απεικονίζεται στην Εικόνα 17a. Είναι $n = 6$. Αρχίζοντας από $i = \lfloor 6/2 \rfloor = 3$ κάνουμε διαδοχικά καθόδους των στοιχείων $H(3)$, $H(2)$ και $H(1)$. Στις εικόνες 17b-e φαίνονται τα υποδέντρα, που σχηματίζονται κάθε φορά, που γίνεται μια εναλλαγή δυο κλειδιών.

4.1.2 Ο αλγόριθμος *Heap Sort*

Ένας γνωστός αλγόριθμος ταξινόμησης είναι ο αλγόριθμος ταξινόμησης σωρού, που αποτελεί μια βελτιωμένη παραλλαγή του αλγόριθμου επιλογής, που περιγράφηκε σε προηγούμενη ενότητα. Όπως και ο αλγόριθμος επιλογής, έτσι και αυτός ο αλγόριθμος

δουλεύει ψάχνοντας το μεγαλύτερο (ή μικρότερο) στοιχείο της λίστας, τοποθετώντας το στο τέλος (ή στην αρχή) και συνεχίζει με το υπόλοιπο της λίστας. Η διαφορά είναι, όμως, ότι ο αλγόριθμος ταξινόμησης σωρού εκτελεί αυτή τη διαδικασία πιο αποτελεσματικά χρησιμοποιώντας ένα τύπο δεδομένων που ονομάζεται σωρός, που ουσιαστικά είναι ένας ειδικός τύπος δυαδικού δέντρου.

Μόλις τα στοιχεία της λίστας σχηματίσουν το σωρό, η ρίζα του δέντρου είναι το μεγαλύτερο στοιχείο. Τότε αφαιρείται και τοποθετείται στο τέλος της λίστας και σχηματίζει ξανά το σωρό με αποτέλεσμα η ρίζα του δέντρου να είναι πάλι το μεγαλύτερο στοιχείο. Χρησιμοποιώντας το σωρό για να βρεθεί το μεγαλύτερο στοιχείο της λίστας απαιτείται $O(\log n)$ χρόνος αντί για $O(n)$ που χρειάζεται για μια σειριακή σάρωση στον απλό αλγόριθμο επιλογής. Αυτό επιτρέπει στην ταξινόμηση σωρού να εκτελείται σε χρόνο $O(n \log n)$.

Η ταξινόμηση σωρού έχει τα εξής χαρακτηριστικά:

- παραλλαγή της ταξινόμησης επιλογής
- είναι ασταθής (unstable)
- επιτόπιος (με πολυπλοκότητα χειρότερης περίπτωσης $O(n \log n)$)
- γενική μέθοδος επιλογή

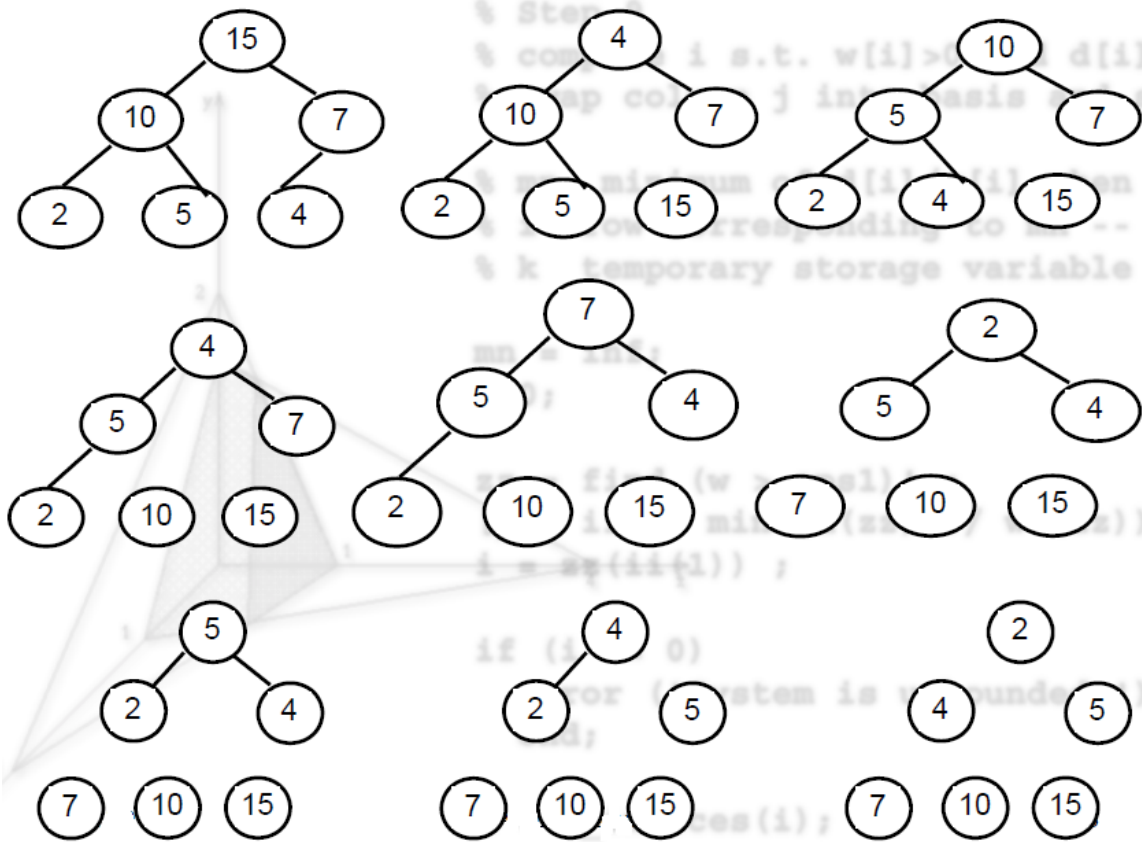
Στην παρακάτω εικόνα φαίνεται ο ψευδοκώδικας του αλγόριθμου:

Πίνακας 6: Ψευδοκώδικας αλγόριθμου Heap Sort

Αλγόριθμος : heapsort	
Δεδομένα : T, n	
Αποτελέσματα : H	
1	H ← buildheapdown(T)
2	για i από n μέχρι 2 βήμα -1
3	[H(1), H(i)] ← εναλλαγή(H(1), H(i))
4	H(1:i-1) ← heapifydown((H(i-1), 1)

4.2 Παράδειγμα

Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα ταξινόμησης ενός πίνακα 6 στοιχείων σε αύξουσα σειρά σύμφωνα με τα βήματα του αλγορίθμου που παρουσιάστηκαν.



Εικόνα 18: Παράδειγμα ταξινόμησης με τον αλγόριθμο Heap Sort

4.3 Ανάλυση πολυπλοκότητας

Δεν γνωρίζουμε προκαταβολικά αν ο αλγόριθμος είναι ομογενής ή όχι. Επομένως θα κάνουμε ανάλυση χειρότερης και καλύτερης περίπτωσης. Προς το σκοπό αυτό παρατηρούμε ότι η κλήση της συνάρτησης `buildheapdown` στη γραμμή 1 έχει

πολυπλοκότητα $\Theta(n)$ και η εντολή καταχώρησης της γραμμής 3 έχει πολυπλοκότητα $\Theta(1)$ σε κάθε επανάληψη του βρόχου του για.

Ανάλυση καλύτερης περίπτωσης. Στα στιγμιότυπα ελάχιστου χρόνου η πράξη `heapifydown` εκτελείται σε χρόνο $\Theta(1)$. Επομένως ο βρόχος του για σε μια εκτέλεση παίρνει χρόνο $\Theta(1) + \Theta(1) = \Theta(1)$. Επειδή ο βρόχος του για εκτελείται $n - 1$ φορές, η πολυπλοκότητα της καλύτερης περίπτωσης είναι

$$\Theta(n) + (n - 1)\Theta(1) = \Theta(n) + \Theta(n - 1) = \Theta(2n - 1) = \Theta(n).$$

Ανάλυση χειρότερης περίπτωσης. Στην i επανάληψη του βρόχου του για, η σωρός έχει i στοιχεία. Στην επανάληψη αυτή, στα στιγμιότυπα μέγιστου χρόνου η συνάρτηση `heapifydown` παίρνει χρόνο $\Theta(\lambda_{οδi})$. Επομένως η πολυπλοκότητα χειρότερης περίπτωσης του βρόχου είναι

$$\begin{aligned} \sum_{i=2}^n [\Theta(1) + \Theta(\lambda_{οδi})] &= \sum_{i=2}^n \Theta(1) + \sum_{i=2}^n \Theta(\lambda_{οδi}) \\ &= \Theta(n - 1) + \Theta\left(\sum_{i=2}^n \lambda_{οδi}\right) \\ &= \Theta(n) + \Theta(\lambda_{οδ}((n)!)) \\ &= \Theta(n) + \Theta(n \lambda_{οδ}(n)) \\ &= \Theta(n \lambda_{οδ}n) \end{aligned}$$

Επομένως η πολυπλοκότητα χειρότερης περίπτωσης είναι

$$\Theta(n) + \Theta(n \lambda_{οδ}n) = \Theta(n \lambda_{οδ}n)$$

5. Αλγόριθμος WeakHeapSort

5.1 Ψευδοκώδικας Αλγορίθμου

5.1.1 Ο αλγόριθμος Gparent

Ο αλγόριθμος Gparent (Dutton, 1993) ενεργεί ως λειτουργία που επιστρέφει τον Grandparent. Ο ψευδοκώδικας που υλοποιεί τον αλγόριθμο αυτό παρατίθεται στον παρακάτω πίνακα.

Πίνακας 7: Αλγόριθμος Gparent

Αλγόριθμος : <u>Gparent</u>	
Δεδομένα : j, Reverse	
Αποτελέσματα : Grandparent	
1	όσο odd(j) == Reverse(j/2)
2	j = $\lceil j/2 \rceil$
3	Grandparent = $\lceil j/2 \rceil$

5.1.1 Ο αλγόριθμος Merge

Στον αλγόριθμο Merge (Dutton, 1993), οι weak-heaps που βρίσκονται στο i και το j συγχωνεύεται σε μία weak-heap που βρίσκεται στο i. Ο ψευδοκώδικας που υλοποιεί τον αλγόριθμο αυτό παρατίθεται στον παρακάτω πίνακα.

Πίνακας 8: Αλγόριθμος Merge

Αλγόριθμος : Merge	
Δεδομένα : h, Reverse	
Αποτελέσματα : Reverse	
1	αν $h(i) < h(j)$
2	swap(i, j)
3	Reverse(j) = not Reverse(j)

5.1.3 Ο αλγόριθμος MergeForest

Η διαδικασία MergeForest (Dutton, 1993) επικαλείται τις τιμές του πίνακα $h[1... m]$ αναπαριστώντας ένα δάσος από weak-heaps. Η δήλωση επανέλαβε-μέχρις ότου προχωράει το x στον πιο αριστερό κόμβο. Κάθε κόμβος που μετράται, εκτός από τον κόμβο 1, είναι ο (weak-heap) αριστερός γιος του πατέρα του. Ο κόμβος m συγχωνεύεται αρχικά με τον κόμβο x ώστε το ελλοχεύον δυαδικό δέντρο να παραμείνει ισορροπημένο. Αυτό επίσης εγγυάται ότι τα ζευγάρια των weak-heaps που συγχωνεύονται, καθώς περπατάμε πίσω μέχρι τη ρίζα, είναι συμβατοί σωροί weak-heaps.

Όπως περιγράφεται ως εδώ η διαδικασία, η MergeForest θα άφηνε την παραγόμενη weak-heap σε έναν πίνακα $h[0... m - 1]$. Τότε ο WeakHeapSort θα πρέπει να κινήσει την τιμή από το $h[0]$ στο $h[m]$ πριν την επόμενη κλήση της MergeForest. Για να αποβάλει αυτή την περιττή μεταφορά δεδομένων, η MergeForest απλά εξετάζει τον κόμβο m ως τη ρίζα της weak-heap που κατασκευάζεται από την δήλωση while. Ο ψευδοκώδικας που υλοποιεί τον αλγόριθμο αυτό παρατίθεται στον παρακάτω πίνακα.

Πίνακας 9: Αλγόριθμος MergeForest

Αλγόριθμος : MergeForest	
Δεδομένα : Reverse, m	
Αποτελέσματα : x	
1	x = 1
2	Reverse($\lceil m / 2 \rceil$) = false
3	αν m > 3
4	επανάλαβε
5	x = 2 * x + Reverse(x)
6	μέχρις ότου 2*x + Reverse(x) > m
7	όσο x > 0
8	Merge(m, x)
9	x = $\lceil x / 2 \rceil$

5.1.4 Ο αλγόριθμος Weak-heapsort

Ο αλγόριθμος Weak-heapsort (Dutton, 1993) χρησιμοποιεί τις συναρτήσεις που παρουσιάστηκαν στα προηγούμενα υποκεφάλαια. Ο ψευδοκώδικας που υλοποιεί τον αλγόριθμο αυτό παρατίθεται στον παρακάτω πίνακα.

Πίνακας 10: Αλγόριθμος MergeForest

Αλγόριθμος : WeakHeapSort	
Δεδομένα : n, h	
Αποτελέσματα : h	
1	Για $i = n - 1:1$
2	Merge(Gparent(i),i)
3	h(n) = h(0)
4	Για $i = n - 1:2$
5	MergeForest(i)

5.2 Παράδειγμα

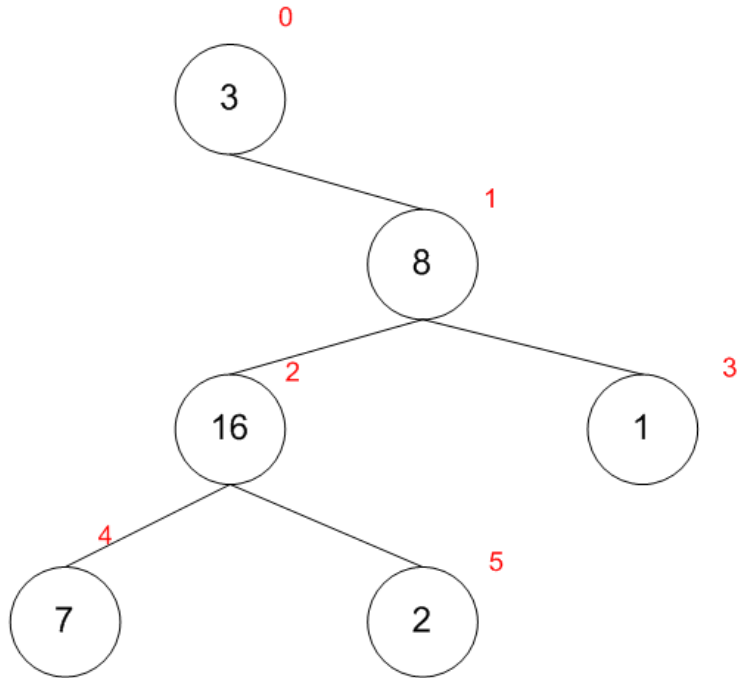
Έστω ότι έχουμε τον πίνακα A και θέλουμε να τον ταξινομήσουμε με τον αλγόριθμο Weakheapsort:

$$A=[3, 8, 16, 1, 7, 2]$$

Ορίζουμε τις τιμές του πίνακα αντιστροφής R ως false:

$$R=[0, 0, 0, 0, 0, 0]$$

Στην παρακάτω εικόνα παρουσιάζεται η απεικόνιση του πίνακα A με τη βοήθεια της δομής δεδομένων weak-heap.



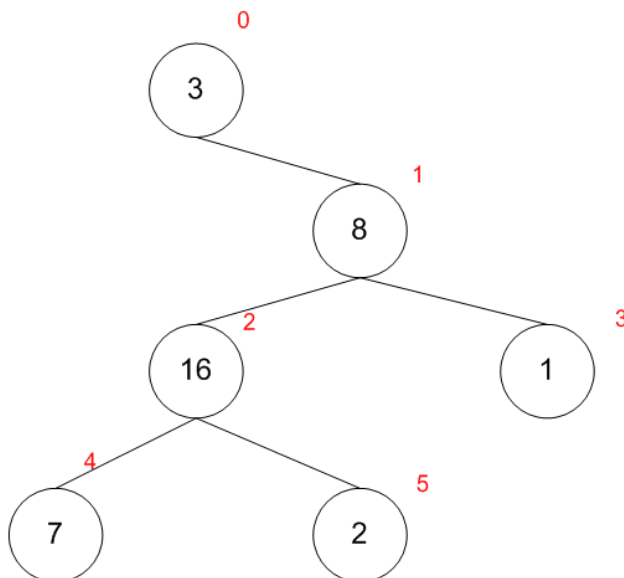
Εικόνα 19: Αρχικό δέντρο παραδείγματος

Πρώτα εκτελείται η συνάρτηση Weakhearify για i από 5 εώς 1:

Για $i=5$, έχουμε $Gparent(5)=2$.

Άρα η $Merge(2, 5)$ δίνει:

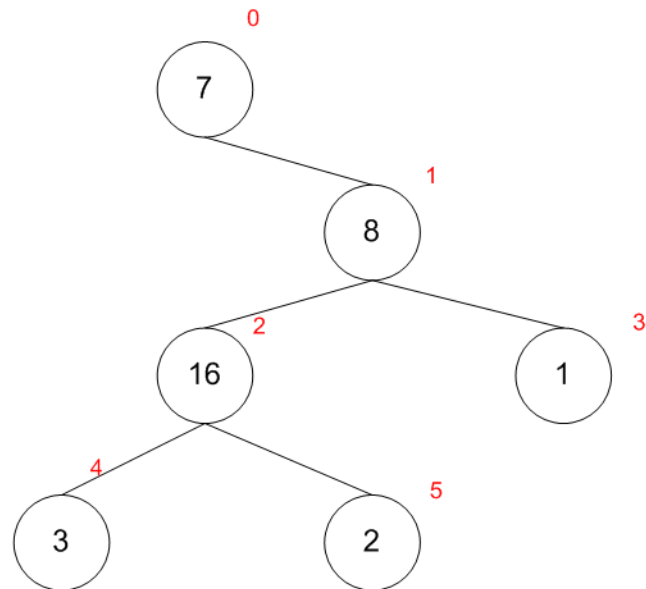
$$A=[3, 8, 16, 1, 7, 2] \text{ και } R=[0, 0, 0, 0, 0, 0]$$



Για $i=4$, έχουμε $Gparent(4)=0$.

Άρα η $Merge(0,4)$ δίνει:

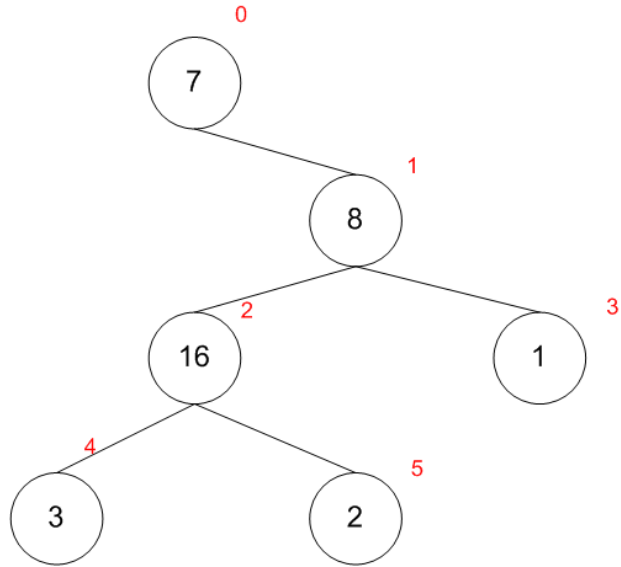
$A=[7, 8, 16, 1, 3, 2]$ και $R=[0, 0, 0, 0, 1, 0]$



Για $i=3$, $Gparent(3)=1$

Άρα η $Merge(1,3)$ δίνει:

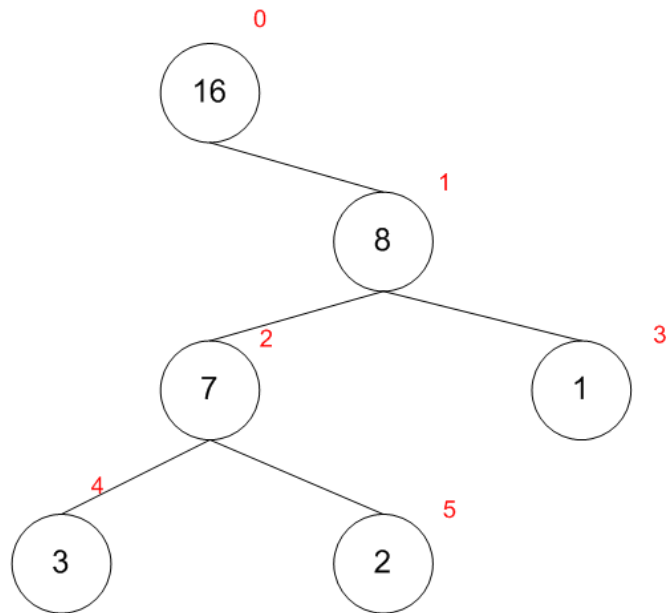
$A=[7, 8, 16, 1, 3, 2]$ και $R=[0, 0, 0, 0, 1, 0]$



Για $i=2$, $Gparent(2)=0$.

Άρα η $Merge(0,2)$ δίνει:

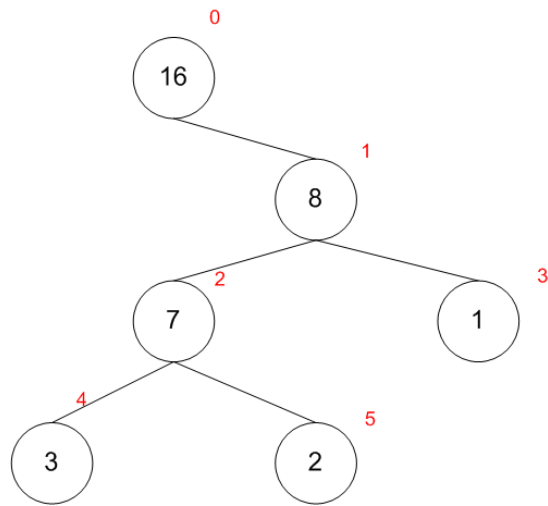
$$A=[16, 8, 7, 1, 3, 2] \text{ και } R=[0, 0, 1, 0, 1, 0]$$



Για $i=1$, $Gparent(1)=0$.

Άρα η $Merge(0,1)$ δίνει:

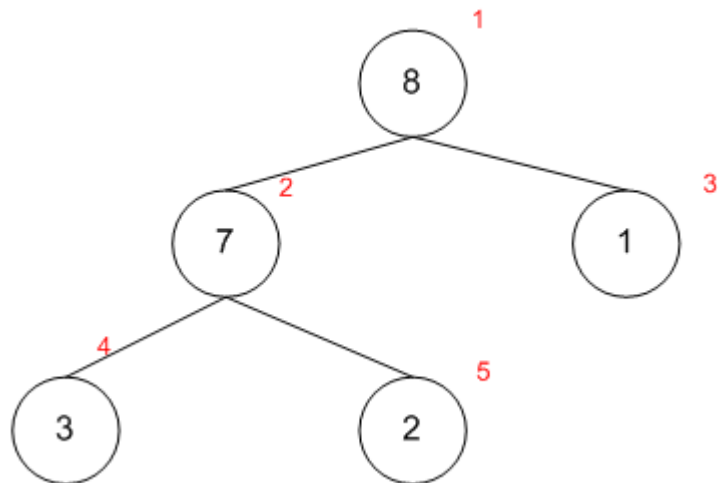
$$A=[16, 8, 7, 1, 3, 2] \text{ και } R=[0, 0, 1, 0, 1, 0]$$



$A[0]=A[n]$

$A[n]=A[6]=16$

Το $A[6]$ φεύγει από τον σωρό



Στη συνέχεια οι τιμές αυτές αναδιανέμονται σε έναν πίνακα $A[1 \dots n]$:

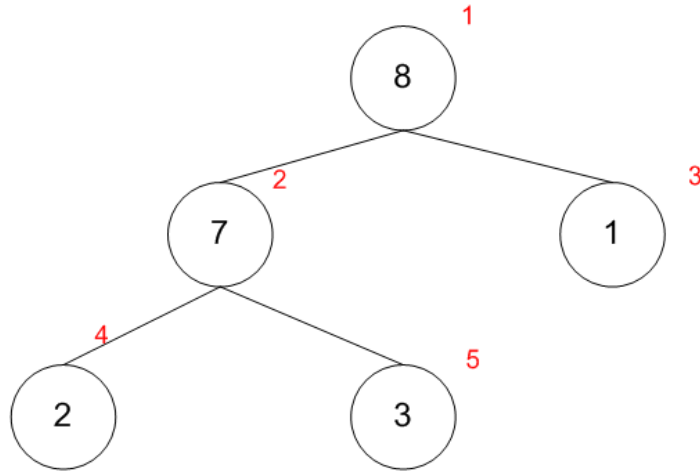
$A=[8, 7, 1, 3, 2, 16]$ και $R=[0, 1, 0, 1, 0, 0]$

Στη συνέχεια για i από 5 μέχρι 2 εκτελείται η συνάρτηση MergeForest.

Για $i=5$, έχουμε $R=[0, 0, 0, 1, 0, 0]$ και $X=4$.

Η Merge(5,4) δίνει:

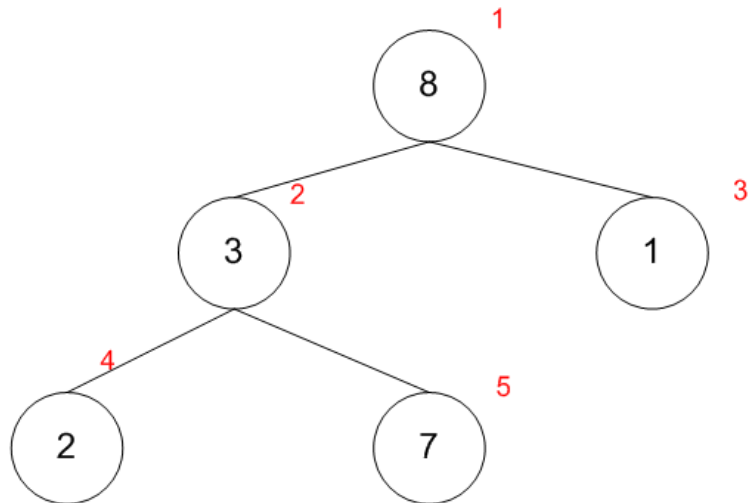
$A=[8, 7, 1, 2, 3, 16]$ και $R=[0, 0, 0, 0, 0, 0]$



$X=2$

Η Merge(5,2) δίνει:

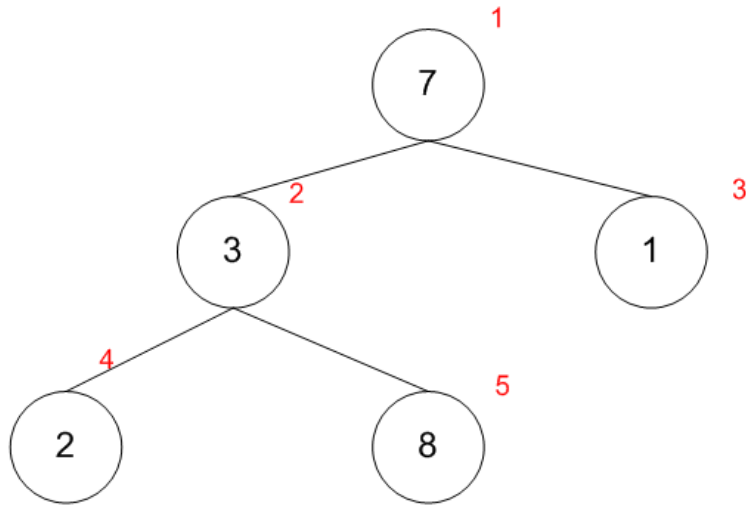
$A=[8, 3, 1, 2, 7, 16]$ και $R=[0, 1, 0, 0, 0, 0]$



$X= \text{Div}(x/2)=1$

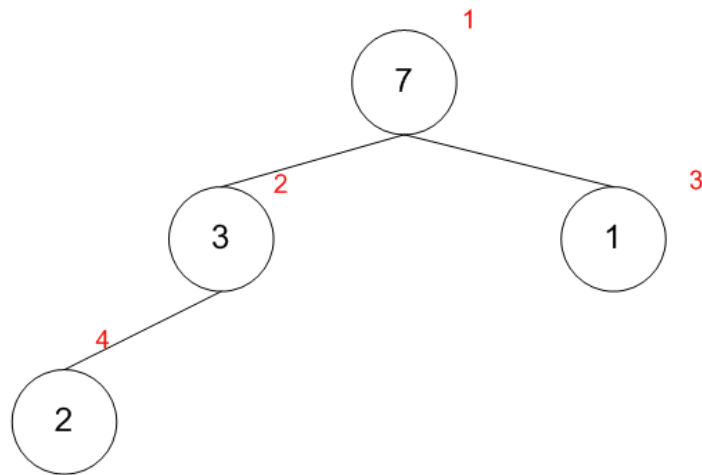
Η Merge(5,1) δίνει:

$A=[7, 3, 1, 2, 8, 16]$ και $R=[1, 1, 0, 0, 0, 0]$



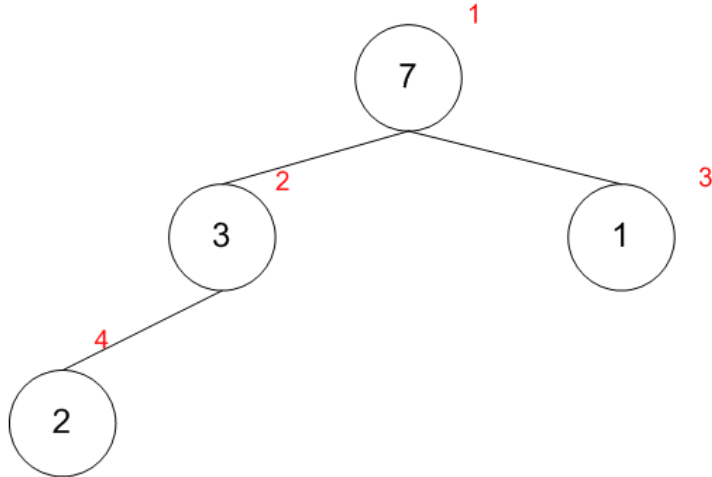
Το $A[5]$ αφαιρείται από τον σωρό και έχουμε:

$A=[7, 3, 1, 2, 8, 16]$ και $R=[1, 1, 0, 0, 0, 0]$



Για $i=4$, έχουμε:

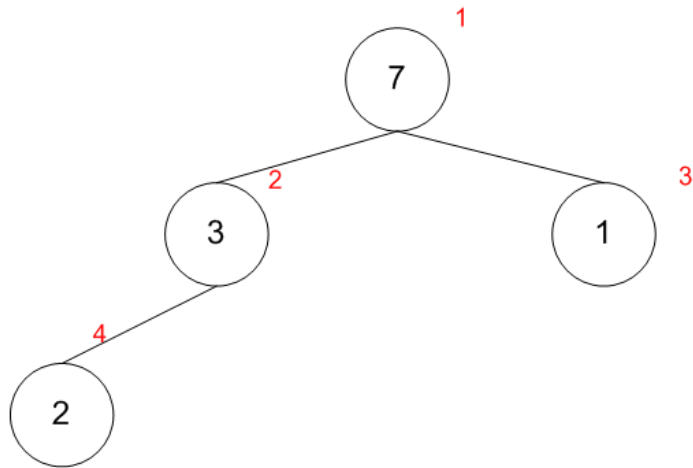
$R=[1, 0, 0, 0, 0, 0]$



$X=3$

Η Merge(4,3) δίνει:

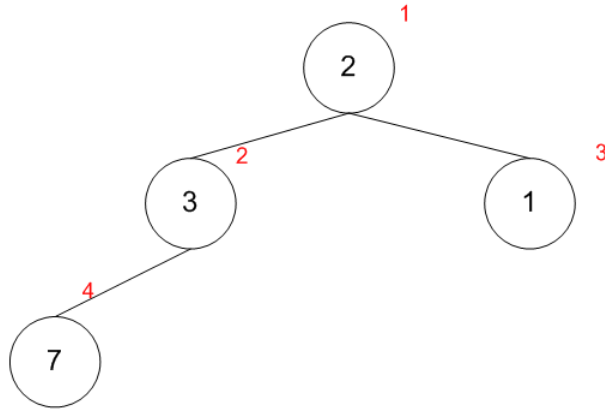
$A=[7, 3, 1, 2, 8, 16]$ και $R=[1, 0, 0, 0, 0, 0]$



$X= \text{Div}(x/2)=1$

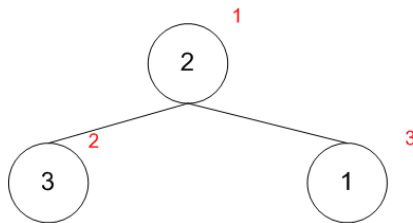
Η Merge(4,1) δίνει:

$A=[2, 3, 1, 7, 8, 16]$ και $R=[0, 0, 0, 0, 0, 0]$



Το A[4] φεύγει από τον σωρό, άρα έχουμε:

$$A=[2, 3, 1, 7, 8, 16] \text{ και } R=[0, 0, 0, 0, 0, 0]$$

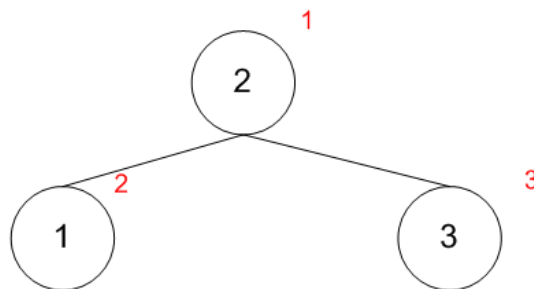


Για $i=3$, έχουμε:

$$R=[0, 0, 0, 0, 0, 0] \text{ και } X=2$$

Η Merge(3,2) δίνει:

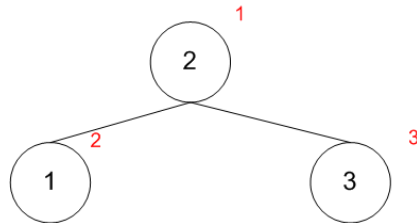
$$A=[2, 1, 3, 7, 8, 16] \text{ και } R=[0, 1, 0, 0, 0, 0]$$



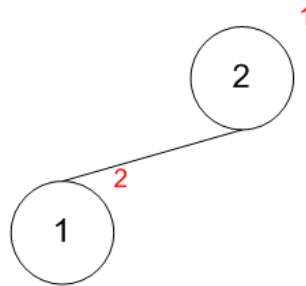
$$X = \text{Div}(x/2) = 1$$

Η Merge(3,1) δίνει:

$$A = [2, 1, 3, 7, 8, 16] \text{ και } R = [0, 1, 0, 0, 0, 0]$$



Το A[3] φεύγει από τον σωρό.



$$A = [2, 1, 3, 7, 8, 16]$$

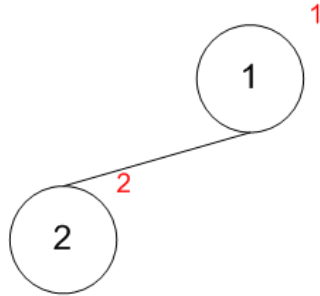
$$R = [0, 1, 0, 0, 0, 0]$$

Για $i=2$

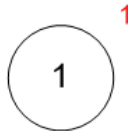
$$X = 1$$

Η Merge(2,1) δίνει:

$$A = [1, 2, 3, 7, 8, 16] \text{ και } R = [1, 1, 0, 0, 0, 0]$$



Το $A[2]$ φεύγει από τον σωρό.



$A=[1, 2, 3, 7, 8, 16]$

Το $A[1]$ φεύγει από τον σωρό:

$A=[1, 2, 3, 7, 8, 16]$

5.3 Ανάλυση πολυπλοκότητας

Το μέτρο της πολυπλοκότητας θα είναι ο αριθμός συγκρίσεων στοιχείων με δύο τρόπους. Αγνοούμε τις διαιρέσεις και τους πολλαπλασιασμούς ακέραιων αριθμών δια δύο που απαιτούνται για να περπατήσει πάνω-κάτω ένα δυαδικό δέντρο. Εκείνες οι μερίδες από το πρόγραμμα θα μπορούσαν να κωδικοποιηθούν σε μια γλώσσα χαμηλότερου επιπέδου, έτσι ώστε οδηγίες καταλόγων μετατόπισης να χρησιμοποιηθούν για να γίνουν αυτές οι διαδικασίες μικρότερες κατά έναν παράγοντα στο χρόνο εκτέλεσης. Επίσης, όταν τα στοιχεία των δεδομένων είναι πιο σύνθετα, αυτές οι διαδικασίες γίνονται λιγότερο σημαντικές από τους χειρισμούς στοιχείων.

Θεώρημα 1: Ο μέγιστος αριθμός συγκρίσεων στοιχείων που απαιτείται από τον WeakHeapSort για οποιοδήποτε αριθμό $n > 1$ τιμών είναι λιγότερος από $(n - 1) \log(n) + 0.086013n$ (Atkinson, 1986).

Απόδειξη: Παράγουμε αρχικά ένα ανώτερο όριο στον ελάχιστο αριθμό συγκρίσεων που απαιτούνται. Κατόπιν υποστηρίζουμε ότι ο μέγιστος αριθμός συγκρίσεων είναι το πολύ $n - \log(n) - 1$ περισσότερος από αυτό το αποτέλεσμα. Καμία σύγκριση δεν εμφανίζεται όταν $n=1$. Έτσι μπορούμε να υποθέσουμε ότι $2^k - 1 < n < 2^{k+1}$, για θετικό ακέραιο αριθμό k . Κατόπιν, $k = \log(n)$ είναι ο αριθμός επιπέδων των κόμβων στο κατώτατο σημείο του δέντρου με τη ρίζα στο επίπεδο 0, τον δεξιό γιο του στο επίπεδο 1 κλπ.

Καθώς οι κόμβοι ρίζας αφαιρούνται, επικαλείται η MergeForest με τις εναπομείναντες τιμές να καλούνται διαδοχικά $n - 1, n - 2, \dots, 2$. Καθώς το i μειώνεται από $n - 1$ ως το 2, ο ελάχιστος συνολικός αριθμός συγκρίσεων, αποκλειστικά της WeakHeapify, είναι (Edelkamp et al., 2001):

$$\sum_{i=2}^{n-1} (\lceil \log(i+1) \rceil - 1) = nk - 2^k - n + 2.$$

Από τη στιγμή που η WeakHeapify χρησιμοποιεί ακριβώς $n - 1$ συγκρίσεις δεδομένων, ο γενικός αριθμός συγκρίσεων καλύτερης περίπτωσης είναι $nk - 2^k + 1$.

Από τα προηγούμενα σχόλια, ο συνολικός αριθμός συγκρίσεων χειρότερης περίπτωσης είναι ο αριθμός καλύτερης περίπτωσης συν $n - 1 - \lceil \log n \rceil$ δηλαδή ένα σύνολο από $nk - 2^k + n - k$. Για οποιοσδήποτε $n > 1$, υπάρχουν μερικοί πραγματικοί αριθμοί x , όπου $0 < x < 1$, έτσι ώστε $k = \log n + x$.

Κατόπιν, $nk - 2^k + n - k = n \log n + nx - n2^x + n - \log n - x = (n - 1) \log n + n(x - 2^x + 1) - x$. Η λειτουργία $x - 2^x + 1$ φράσσεται άνω από το 0.086013 και είναι μηδέν όταν $x = 0$. Κατά συνέπεια, ο συνολικός αριθμός συγκρίσεων είναι αυστηρά λιγότερος από $(n - 1) \log n + 0.086013n$.

6. Υπολογιστική Μελέτη

6.1 Εισαγωγή

Πολλές εργασίες υπάρχουν στη βιβλιογραφία που περιέχουν οδηγίες για τη διεξαγωγή πειραμάτων και υπολογιστικών μελετών (Hooker, 1998; Rardin and Uzsoy, 2001). Για τη σύγκριση διαφορετικών αλγορίθμων πρέπει να εκτελέσουμε τους αλγόριθμους σε κοινά στιγμιότυπα δεδομένων. Τα δεδομένα αυτά πρέπει να είναι μεγάλης διάστασης και να δημιουργούνται από γεννήτριες τυχαίων αριθμών. Για κάθε μία διάσταση οι αλγόριθμοι πρέπει να εκτελούν 10 ή και περισσότερα στιγμιότυπα.

Στο κεφάλαιο αυτό θα παρουσιαστεί η υπολογιστική μελέτη που διενεργήθηκε στα πλαίσια της εργασίας αυτής μεταξύ των δύο αλγορίθμων ταξινόμησης. Η υπολογιστική μελέτη διενεργήθηκε σε έναν Intel Core i7 2.2 GHz και 6 GB μνήμης RAM με λειτουργικό σύστημα Microsoft Windows 7 64 bit. Οι δύο αλγόριθμοι εκτελέστηκαν σε τυχαίους αριθμούς διαστάσεων 1000000 – 10000000. Όλοι οι αριθμοί δημιουργήθηκαν με τη γεννήτρια ψευδοτυχαίων αριθμών της Java. Για κάθε διάσταση εκτελέστηκαν 10 στιγμιότυπα και υπολογίστηκε ο μέσος όρος.

Οι αλγόριθμοι προγραμματίστηκαν σε γλώσσα Java και χρησιμοποιήθηκε το Netbeans IDE. Στο παράρτημα υπάρχουν οι πηγαίοι κώδικες των δύο αλγορίθμων. Η υπολογιστική αυτή μελέτη καταγράφει το χρόνο εκτέλεσης (σε msec), τον αριθμό των συγκρίσεων και των αναθέσεων.

6.2 Εκτέλεση πειραμάτων

Για την διενέργεια της υπολογιστικής μελέτης δημιουργήθηκαν δέκα στιγμιότυπα για κάθε διάσταση (από 1000000 μέχρι 10000000 με βήμα 1000000). Τα στιγμιότυπα που δημιουργήθηκαν ακολουθούν την κανονική κατανομή και δημιουργήθηκαν με τη γεννήτρια των ψευδοτυχαίων αριθμών της Java.

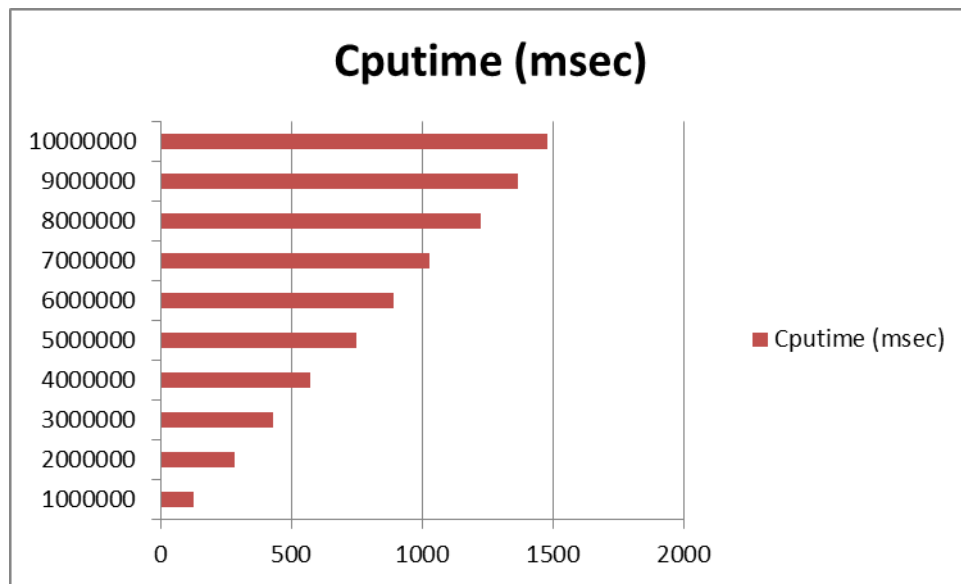
Στον παρακάτω πίνακα φαίνεται ο χρόνος εκτέλεσης, ο αριθμός των συγκρίσεων και ο αριθμός των αναθέσεων του αλγορίθμου HeapSort σε κάθε διάσταση (ως χρόνος λαμβάνεται ο μέσος χρόνος από την εκτέλεση και των δέκα στιγμιότυπων):

Πίνακας 11: Αποτελέσματα αλγορίθμου HeapSort

Elements	1000000	2000000	3000000	4000000	5000000	6000000	7000000	8000000	9000000	10000000
Cputime (msec)	126	282	429	574	751	893	1027	1226	1366	1480
Assignments	56500223	118957247	183762935	250126307	318168395	386713529	456145883	525804257	595447925	665737283
Comparisons	17533820	37042516	57562723	78350063	99610627	121680410	142991586	165555996	187701576	209648372

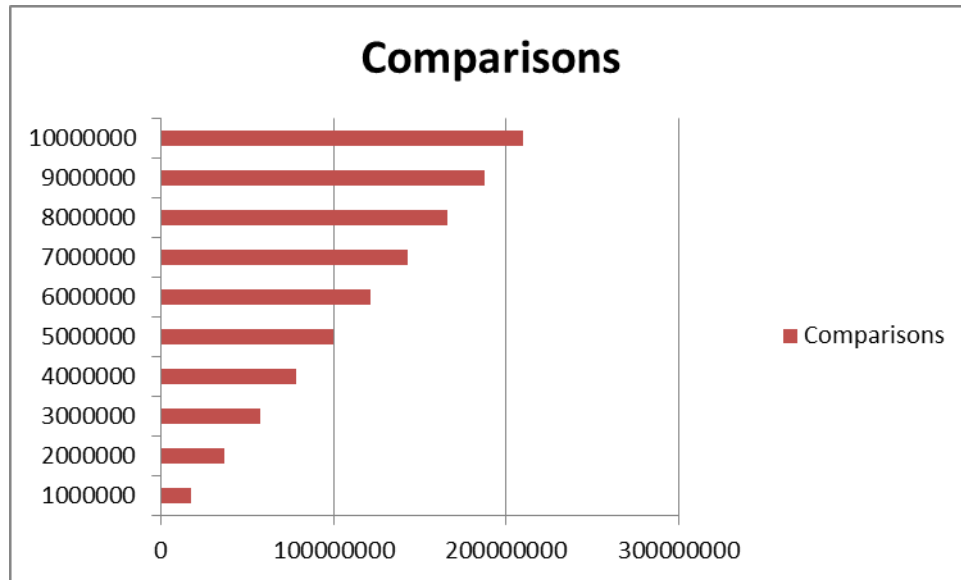
Παρατηρούμε, όπως ήταν αναμενόμενο, μια πολυωνυμική αύξηση του χρόνου, του αριθμού των συγκρίσεων και των αναθέσεων.

Στο παρακάτω γράφημα αποτυπώνεται ο χρόνος εκτέλεσης του αλγορίθμου HeapSort.



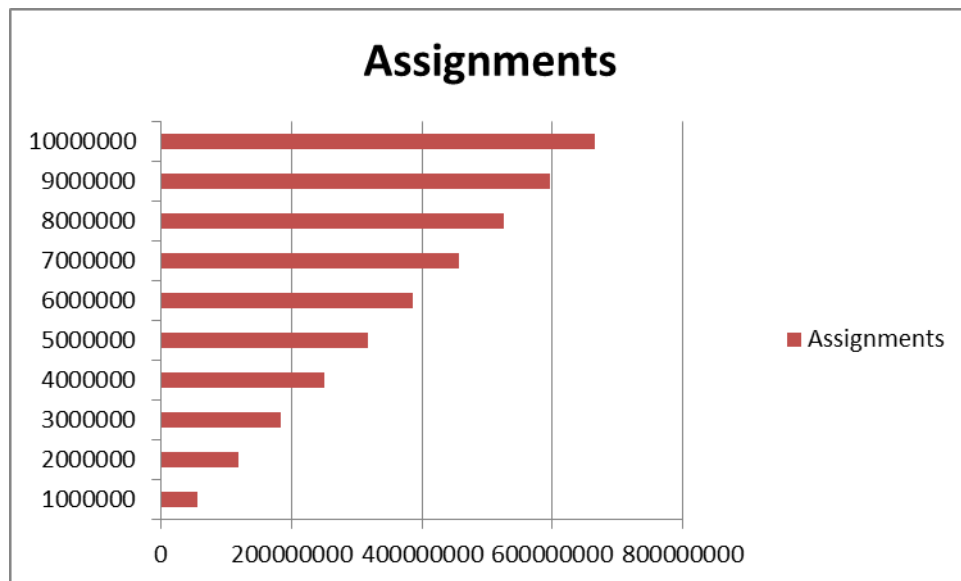
Εικόνα 20: Χρόνος εκτέλεσης αλγορίθμου HeapSort

Στο παρακάτω γράφημα αποτυπώνεται ο αριθμός συγκρίσεων του αλγορίθμου HeapSort.



Εικόνα 21: Αριθμός συγκρίσεων αλγόριθμου HeapSort

Στο παρακάτω γράφημα αποτυπώνεται ο αριθμός των αναθέσεων του αλγορίθμου HeapSort.



Εικόνα 22: Αριθμός αναθέσεων αλγόριθμου HeapSort

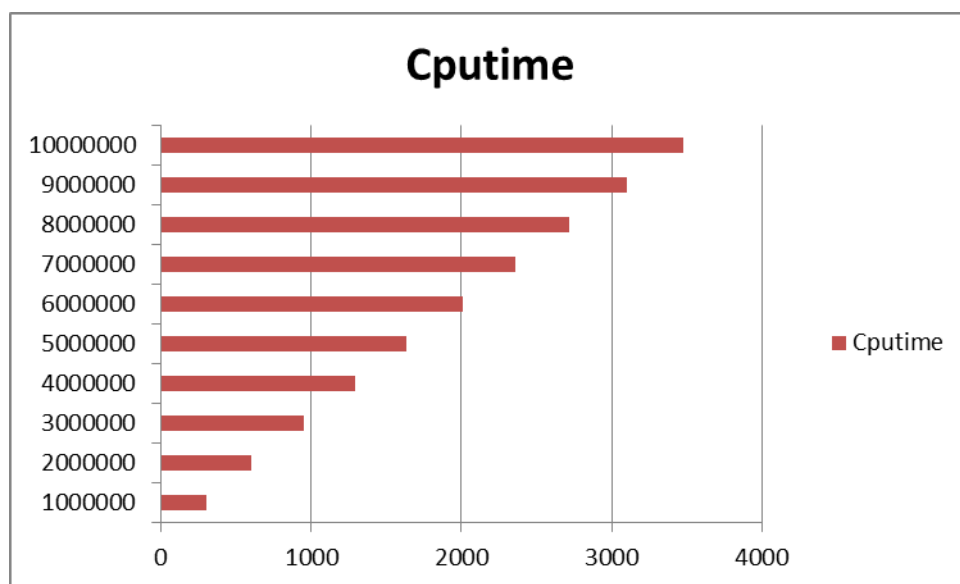
Στον παρακάτω πίνακα φαίνεται ο χρόνος εκτέλεσης, ο αριθμός των συγκρίσεων και ο αριθμός των αναθέσεων του αλγορίθμου WeakHeapSort σε κάθε διάσταση (ως χρόνος λαμβάνεται ο μέσος χρόνος από την εκτέλεση και των δέκα στιγμιότυπων):

Πίνακας 12: Αποτελέσματα αλγορίθμου WeakHeapSort

Elements	1000000	2000000	3000000	4000000	5000000	6000000	7000000	8000000	9000000	10000000
Cputime (msec)	305	605	948	1293	1635	2010	2357	2718	3100	3476
Assignments	34474341	70967431	107894279	145906923	183825891	221786056	261003426	299762005	338248698	377696640
Comparisons	40446763	84897858	130626850	177787425	225367800	273251457	322350214	371562520	420916593	470743753

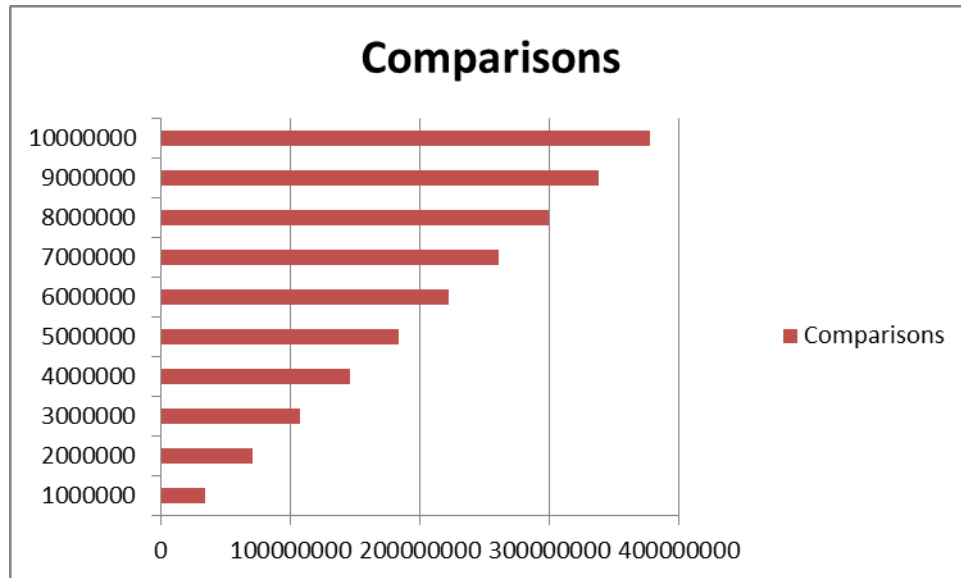
Παρατηρούμε πάλι, όπως ήταν αναμενόμενο, μια πολυωνυμική αύξηση του χρόνου, του αριθμού των συγκρίσεων και των αναθέσεων.

Στο παρακάτω γράφημα αποτυπώνεται ο χρόνος εκτέλεσης του αλγορίθμου WeakHeapSort.



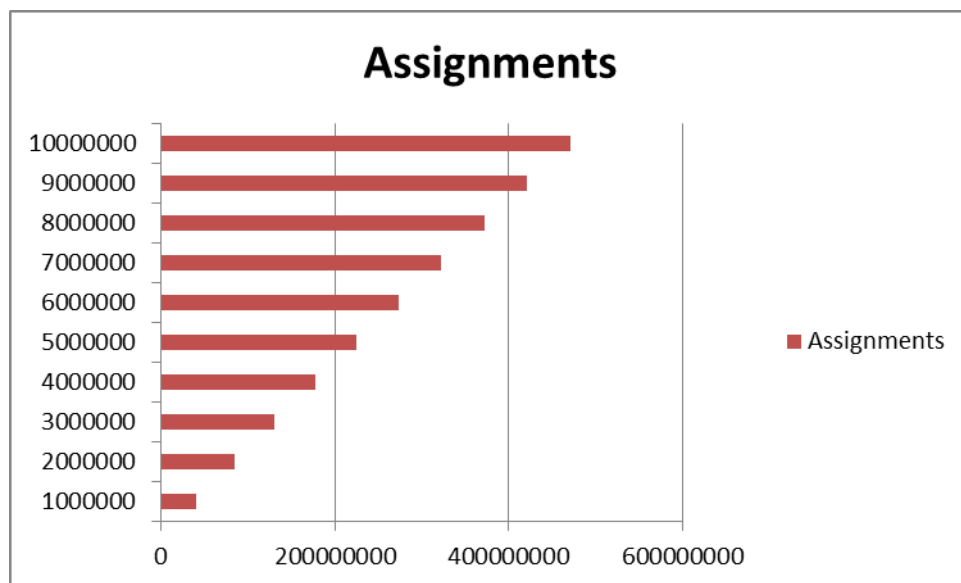
Εικόνα 23: Χρόνος εκτέλεσης αλγορίθμου WeakHeapSort

Στο παρακάτω γράφημα αποτυπώνεται ο αριθμός συγκρίσεων του αλγορίθμου WeakHeapSort.



Εικόνα 24: Αριθμός συγκρίσεων αλγόριθμου WeakHeapSort

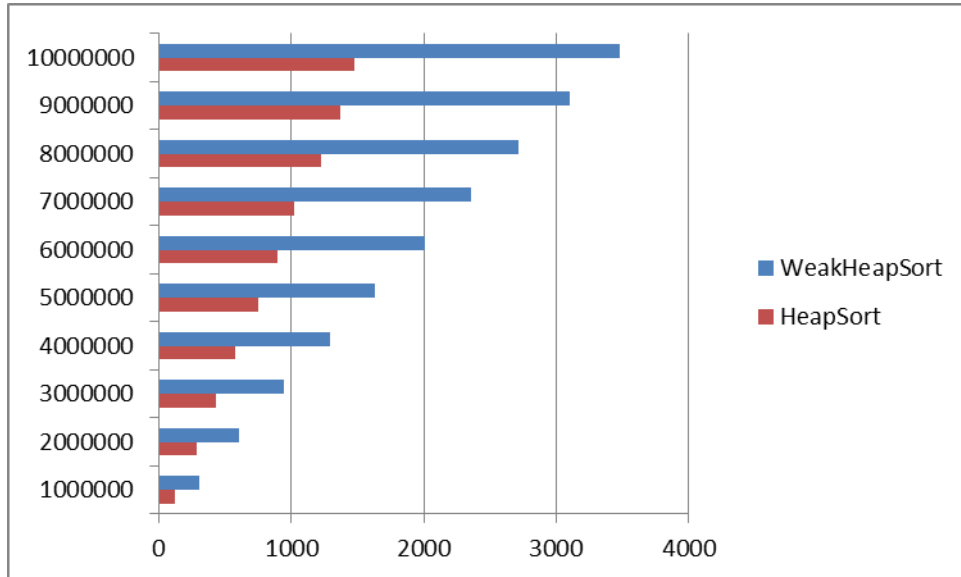
Στο παρακάτω γράφημα αποτυπώνεται ο αριθμός των αναθέσεων του αλγορίθμου WeakHeapSort.



Εικόνα 25: Αριθμός αναθέσεων αλγόριθμου WeakHeapSort

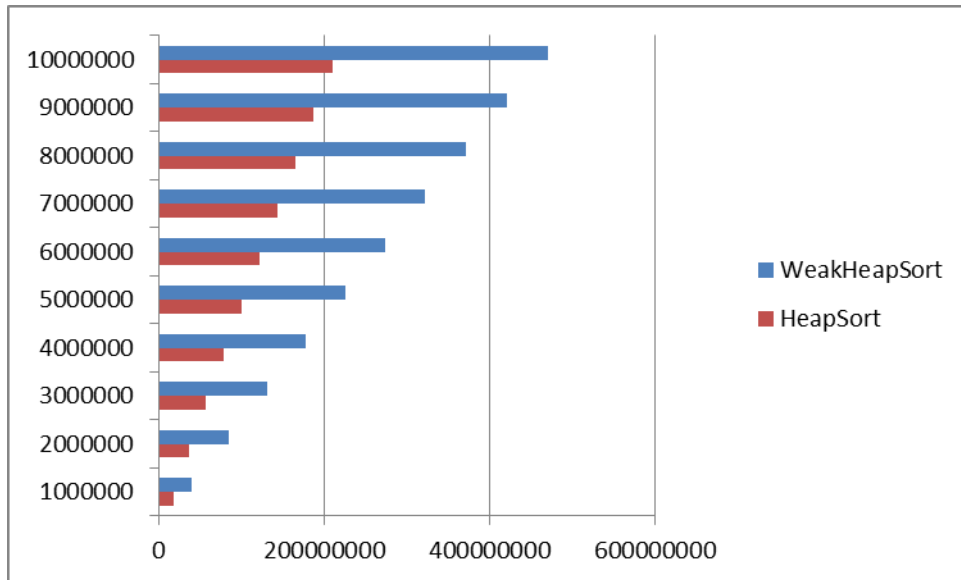
Στη συνέχεια συγκρίνονται οι δύο αλγόριθμοι μεταξύ τους. Ο αλγόριθμος HeapSort είναι καθαρά πιο γρήγορος από τον WeakHeapSort. Μάλιστα παρατηρείται ότι

ο αλγόριθμος HeapSort είναι περίπου κατά μέσο όρο 1,79 γρηγορότερος του WeakHeapSort.



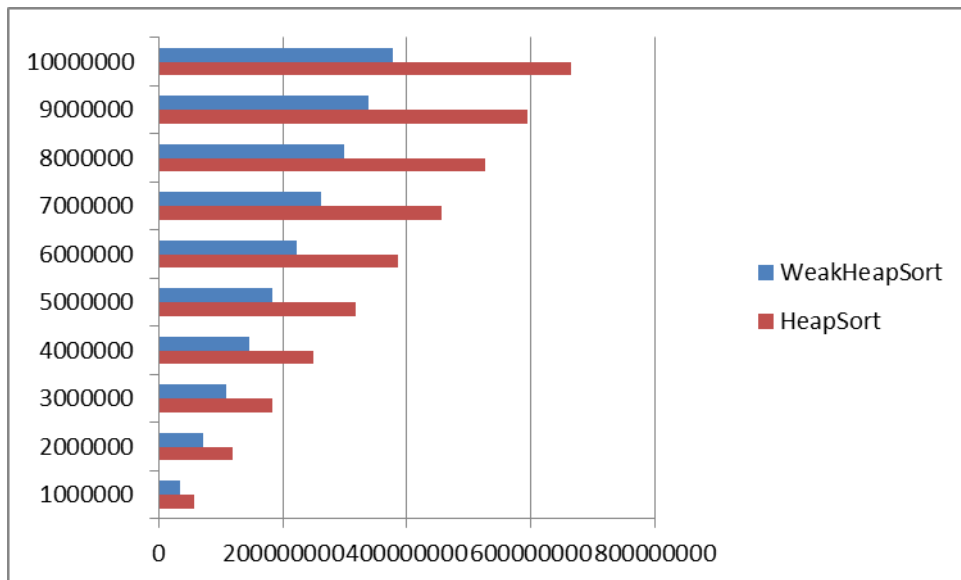
Εικόνα 26: Σύγκριση χρόνου εκτέλεσης αλγορίθμων HeapSort και WeakHeapSort

Ο αλγόριθμος HeapSort εκτελεί πολύ λιγότερες συγκρίσεις από τον WeakHeapSort. Μάλιστα παρατηρείται ότι ο αλγόριθμος HeapSort εκτελεί περίπου κατά μέσο όρο 1,79 λιγότερες συγκρίσεις του WeakHeapSort.



Εικόνα 27: Σύγκριση αριθμού συγκρίσεων αλγόριθμων HeapSort και WeakHeapSort

Ο αλγόριθμος WeakHeapSort εκτελεί πολύ λιγότερες αναθέσεις από τον HeapSort. Μάλιστα παρατηρείται ότι ο αλγόριθμος WeakHeapSort εκτελεί περίπου κατά μέσο όρο 1,38 λιγότερες αναθέσεις του WeakHeapSort.



Εικόνα 28: Σύγκριση αριθμού αναθέσεων αλγόριθμων HeapSort και WeakHeapSort

7. Συμπεράσματα

Στη συγκεκριμένη εργασία παρουσιάστηκαν και αναλύθηκαν διεξοδικά οι αλγόριθμοι HeapSort και WeakHeapSort. Όσον αφορά τις δομές που χρησιμοποιούν οι δύο αλγόριθμοι, ο αλγόριθμος HeapSort χρησιμοποιεί σωρούς, ενώ ο WeakHeapSort χρησιμοποιεί τους weak σωρούς που διαφέρουν από τους πρώτους στα εξής σημεία:

- Κάθε κλειδί στο δεξί υποδένδρο κάθε κόμβου είναι μικρότερο ή ίσο με το κλειδί του κόμβου του.
- Η ρίζα δεν έχει κανένα αριστερό παιδί.
- Τα φύλλα βρίσκονται στα τελευταία δύο επίπεδα του δέντρου μόνο

Ο αλγόριθμος HeapSort δουλεύει ψάχνοντας το μεγαλύτερο (ή μικρότερο) στοιχείο της λίστας, τοποθετώντας το στο τέλος (ή στην αρχή) και συνεχίζει με το υπόλοιπο της λίστας. Μόλις τα στοιχεία της λίστας σχηματίσουν το σωρό, η ρίζα του δέντρου είναι το μεγαλύτερο στοιχείο. Τότε αφαιρείται και τοποθετείται στο τέλος της λίστας και σχηματίζει ξανά το σωρό με αποτέλεσμα η ρίζα του δέντρου να είναι πάλι το μεγαλύτερο στοιχείο. Χρησιμοποιώντας το σωρό για να βρεθεί το μεγαλύτερο στοιχείο της λίστας απαιτείται $O(\log n)$ χρόνος. Αυτό επιτρέπει στην ταξινόμηση σωρού να εκτελείται σε χρόνο $O(n \log n)$.

Ο αλγόριθμος WeakHeapSort, που προτάθηκε από τον Dutton το 1993 αποτελεί μια παραλλαγή του αλγόριθμου HeapSort και χρησιμοποιεί τη δομή δεδομένων weak heap, όπως αναφέρθηκε ήδη. Οι σωροί αυτοί μπορούν να υλοποιηθούν με $n - 1$ συγκρίσεις. Ο αριθμός συγκρίσεων χειρότερης περίπτωσης του αλγορίθμου είναι $n \log n - 2 \log n + n - \log n < n \log n + 0.1n$.

Επίσης, στην εργασία αυτή υλοποιήθηκε μια υπολογιστική μελέτη των δύο αλγορίθμων σε τυχαία στιγμιότυπα καθορισμένων διαστάσεων και βρέθηκε ότι ο αλγόριθμος HeapSort είναι αρκετά γρηγορότερος του αλγορίθμου WeakHeapSort (κατά 1,79 φορές κατά μέσο όρο) και εκτελεί λιγότερες συγκρίσεις από τον αλγόριθμο WeakHeapSort (κατά 1,79 φορές κατά μέσο όρο). Αντίθετα, ο αλγόριθμος

WeakHeapSort εκτελεί λιγότερες αναθέσεις από τον αλγόριθμο HeapSort (κατά 1,38 φορές κατά μέσο όρο).

Βιβλιογραφία

Atkinson M. D., Sack J. R., Santoro N., and Strothotte T. Min-max heaps and generalized priority queues, *Commun. ACM* 29, pp. 996 – 1000, 1986.

Coffin M., and Saltzman M. J. Statistical analysis of computational tests of algorithms and heuristics. *INFORMS Journal on Computing*, vol. 12, no. 1, pp. 24-44, 2000.

Cormen T. H., Leiserson C. E., Rivest R. L., and Stein C. *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03293-7, 2000.

Crowder H. P., Dembo R. S., and Mulvey J. M. On reporting Computational Results in Mathematical Programming, *Mathematical Programming* 15, pp. 316-329, 1978.

Dutton R. D. Weak-heap sort, *BIT*, vol. 33, pp. 372 – 381, 1993.

Dutton R. D. The weak-heap data structure, Technical report, University of Central Florida, Orlando, FL 32816, 1992.

Edelkamp S., and Stiegeler P. Pushing the Limits in Sequential Sorting, In *Algorithm Engineering*, *Lecture Notes in Computer Science*, Springer, vol. 1982, pp. 39 – 50, 2001.

Edelkamp S., and Wegener I. On the performance of WEAK-HEAPSORT, STACS 2000, LNCS 1770, pp. 254 – 266, 2000.

Hall N. G., and Posner M. E. Generating Experimental Data for Computational Testing with Machine Scheduling Application, Operations Research, vol. 49, pp. 854-865, 2001.

Hooker J. N. Needed: an empirical science of algorithms, Operations Research, vol. 42(2), pp. 201-212, 1994.

Knuth D. E. The Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition). Addison-Wesley, 1998.

Rardin R. L., and Uzsoy R. Experimental evaluation of heuristic optimization algorithms: A tutorial, Journal of heuristics, vol. 7, pp. 261-304, 2001.

Sedgewick R., and Flajolet P. An Introduction to the Analysis of Algorithms. AddisonWesley, Reading, Mass, 1996.

Παπαρρίζος Κ. Ανάλυση και Σχεδίαση Αλγορίθμων, Εκδόσεις Ζυγός, Θεσσαλονίκη, 2008.

Παράρτημα: Κώδικας εφαρμογής

Π.1 Κλάση HeapSort

```
package heapweakheap;

// η κλάση HeapSort
public class HeapSort {

    private int array[]; // πίνακας προς ταξινόμηση
    private int comparisons; // συγκρίσεις
    private int assignments; // αναθέσεις

    // κατασκευαστής
    public HeapSort(int arr[]){
        this.array = arr;
        this.comparisons = 0;
        this.assignments = 0;
    }

    // βοηθητική μέθοδος
    public void HeapAdjust(int s, int m)
    {
```

```

int temp = array[s];
increaseAssignments();
for(int j = 2 * s + 1; j < m; j = j * 2 + 1)
{
    if(j + 1 < m && array[j] > array[j+1]){
        j++;
        increaseComparisons();
        increaseComparisons();
    }

    if(temp < array[j]){
        increaseComparisons();
        break;
    }

    array[s] = array[j];
    increaseAssignments();

    s = j;
    increaseAssignments();
    array[s] = temp;
    increaseAssignments();
}
}

// μέθοδος ταξινόμησης

```

```

public void sort()
{
    int temp;

    for(int i = (array.length / 2 - 1); i >= 0; --i)
    {
        HeapAdjust(i, array.length);
    }

    for(int j = array.length - 1; j > 0; --j)
    {
        temp = array[0];
        increaseAssignments();
        array[0] = array[j];
        increaseAssignments();
        array[j] = temp;
        increaseAssignments();
        HeapAdjust(0,j);
    }
}

// αύξηση συγκρίσεων κατά 1
public void increaseComparisons(){
    this.comparisons++;
}

```

```
// αύξηση αναθέσεων κατά 1
public void increaseAssignments(){
    this.assignments++;
}

// επιστροφή αναθέσεων
public int getAssignments() {
    return assignments;
}

// επιστροφή συγκρίσεων
public int getComparisons() {
    return comparisons;
}
}
```

Π.2 Κλάση WeakHeapSort

```
package heapweakheap;

public class WeakHeapSort {
```

```
private int r[]; // πίνακας αταξινόμητος
private int a[]; // πίνακας ταξινομημένος
private int comparisons; // συγκρίσεις
private int assignments; // αναθέσεις
```

```
// κατασκευαστής
```

```
public WeakHeapSort(int arr[]) {
    a = arr;
    r = new int[arr.length];
    this.comparisons = 0;
    this.assignments = 0;
}
```

```
// μέθοδος gparent
```

```
public int gparent(int j) {
    while ((j & 1) == r[j / 2]){
        j /= 2;
        increaseComparisons();
    }
    return (j / 2);
}
```

```
// μέθοδος merge
```

```

public void merge(int i, int j) {
    int temp;
    if (a[j] > a[i]) {
        increaseComparisons();
        r[j] = 1 - r[j];
        increaseAssignments();
        temp = a[i];
        increaseAssignments();
        a[i] = a[j];
        increaseAssignments();
        a[j] = temp;
        increaseAssignments();
    }
}

// μέθοδος mergeForest
public void mergeForest(int m) {
    int x=1;
    while (2* x + r[x] < m) {
        increaseComparisons();
        x = 2*x + r[x];
        increaseAssignments();
    }
    while (x > 0) {

```

```

    increaseComparisons();

    merge(0, x);

    x /= 2;

}

}

// μέθοδος heapify
public void heapify() {
    for (int i = a.length - 1; i >= 1; i--)
        merge(gparent(i), i);
}

// μέθοδος ταξινόμησης
public void sort() {
    int temp;

    heapify();

    for (int i = a.length - 1; i >= 2; i--) {
        temp = a[0];
        increaseAssignments();
        a[0] = a[i];
        increaseAssignments();
        a[i] = temp;
        increaseAssignments();
        mergeForest(i);
    }
}

```

```

    }
    if (a.length > 1){
        increaseComparisons();
        temp = a[0];
        increaseAssignments();
        a[0] = a[1];
        increaseAssignments();
        a[1] = temp;
        increaseAssignments();
    }
}

// αύξηση συγκρίσεων κατά 1
public void increaseComparisons(){
    this.comparisons++;
}

// αύξηση αναθέσεων κατά 1
public void increaseAssignments(){
    this.assignments++;
}

// επιστροφή αναθέσεων
public int getAssignments() {

```

```
    return assignments;
}

// επιστροφή συγκρίσεων
public int getComparisons() {
    return comparisons;
}
}
```

Π.3 Κλάση HeapWeakHeap

```
package heapweakheap;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Collections;

public class HeapWeakHeap {

    public static void main(String[] args) throws IOException {
```

```

long start, stop;

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

int minbound = 1;

int maxbound = 1000000000;

for(int n = 1000000; n <= 10000000; n = n + 1000000){

    System.out.println("-----");

    System.out.println("Number of elements: " + n);

    int array[] = new int[n];

    for(int i = 0; i < n; i++){

        array[i] = (int) (maxbound * Math.random());

    }

    Collections.shuffle(Arrays.asList(array));

    start = System.currentTimeMillis();

    HeapSort hsort=new HeapSort(array);

    hsort.sort();

    stop = System.currentTimeMillis();

    System.out.println("Cputime: " + (stop-start));

    System.out.println("Assignments: " + hsort.getAssignments());

    System.out.println("Comparisons: " + hsort.getComparisons());

    start = System.currentTimeMillis();

    WeakHeapSort w = new WeakHeapSort(array);

```

```
w.sort();

stop = System.currentTimeMillis();

System.out.println("Cputime: " + (stop-start));

System.out.println("Assignments: " + w.getAssignments());

System.out.println("Comparisons: " + w.getComparisons());

System.out.println("-----");
}
}
}
```