

Διπλωματική εργασία
του
Σεντελίδη Θεόδωρου
Α.Ε.Μ. : 614

Αξιολόγηση των απόλυτων σωρών Fibonacci
Evaluation of strict Fibonacci heaps

Επιβλέπων καθηγητής:
Μποζάνης Παναγιώτης

Βόλος, Σεπτέμβριος 2012

Περιεχόμενα

1. ΕΙΣΑΓΩΓΗ – ΒΑΣΙΚΕΣ ΕΝΝΟΙΕΣ.....	3
1.1 Βασικές έννοιες Αλγορίθμων.....	4
1.1.1 Δομές Δεδομένων.....	4
1.2 Ανάλυση Αλγορίθμων.....	4
1.2.1 Μοντέλα Υπολογισμού-Μοντέλο RAM.....	4
1.2.2 Ανάλυση Χειρότερης Περιπτώσεως.....	5
1.2.3 Ανάλυση Μέσης Περιπτώσεως.....	5
1.2.4 Ανάλυση Επιμερισμένης Περιπτώσεως.....	5
2. ΟΥΡΕΣ ΠΡΟΤΕΡΑΙΟΤΗΤΑΣ.....	7
2.1 Ουρά Προτεραιότητας Σωρού.....	7
2.2 Δένδρο Διατάξης Σωρού.....	8
2.3 Σωρός Fibonacci.....	9
3. ΑΠΟΛΥΤΗ ΣΩΡΟΣ FIBONACCI	12
3.1 Strict Fibonacci Heap.....	12
3.1.1 Περιγραφή Πράξεων.....	13
3.1.2 Ανάλυση Πλυπλοκότητας – Σταθερές και Θεωρήματα.....	21
3.1.3 Βασικοί Μετασχηματισμοί.....	23
3.1.4 Υλοποίηση.....	26
4. ΠΕΙΡΑΜΑΤΙΚΗ ΑΞΙΟΛΟΓΗΣΗ.....	61
5. ΒΙΒΛΙΟΓΡΑΦΙΑ.....	64

Εισαγωγή

Η διπλωματική αυτή εργασία ασχολείται με την αξιολόγηση μιας νέας δομής δεδομένων, η οποία ονομάζεται “Απόλυτη σωρός Fibonacci” (Strict Fibonacci heap). Η σωρός αυτή αναπτύχθηκε από τους Gerth Stølting Brodal, George Lagogiannis και Robert E. Tarjan και αποτελεί εξέλιξη της σωρού Fibonacci. Συγκεκριμένα, πετυχαίνει ίδια πολυπλοκότητα στη χειρότερη περίπτωση, χρησιμοποιώντας γραμμικό χώρο και επίσης είναι η πρώτη προσέγγιση που βασίζεται αποκλειστικά σε δείκτες.

Στο πρώτο κεφάλαιο, θα παρουσιάσουμε συνοπτικά τις βασικές έννοιες των δομών δεδομένων και της ανάλυσης αλγορίθμων ώστε να υπάρχει το κατάλληλο υπόβαθρο για τον αναγνώστη για να κατανοήσει καλύτερα την ανάλυση που θα ακολουθήσει.

Έπειτα, στο δεύτερο κεφάλαιο, θα αναλύσουμε αρχικά την γενική κατηγορία στην οποία ανήκει η δομή δεδομένων που πραγματεύεται η εργασία, δηλαδή τις ουρές προτεραιότητας. Ιδιαίτερη αναφορά και μια επιφανειακή ανάλυση θα γίνει στη σωρό Fibonacci.

Στο τρίτο κεφάλαιο, θα γίνει η παρουσίαση της κύριας δομής της εργασίας της Απόλυτης σωρού Fibonacci με λεπτομερή περιγραφή των επιμέρους διαδικασιών του αλγορίθμου, καθώς επίσης και παρουσίαση ψευδοκώδικα, αλλά και του πραγματικού κώδικα της σωρού. Η αξιολόγηση έχει γίνει κυρίως βάσει κώδικα σε C από το google code [15]. Τέλος, στο κεφάλαιο αυτό γίνεται και θεωρητική αναφορά στην πολυπλοκότητα των πράξεων για τις επιμέρους εργασίες του αλγορίθμου.

Τέλος, στο τέταρτο κεφάλαιο, υπάρχει η αξιολόγηση των εν λόγω δομών, μέσω κάποιων πειραμάτων, τα οποία έχουν ως σκοπό την επαλήθευση των θεωρημάτων της εργασίας τα οποία θα προσπαθήσουμε να αποδείξουμε πρακτικά.

Στο πέμπτο και τελευταίο κεφάλαιο βρίσκουμε τις βιβλιογραφικές αναφορές οι οποίες μας βοήθησαν στην συγγραφή, κατανόηση και ανάπτυξη της διπλωματικής εργασίας.

1.1 Βασικές έννοιες αλγορίθμων

Με τον όρο «αλγόριθμος» εννοούμε μια ακολουθία υπολογιστικών βημάτων, η οποία απεικονίζει την είσοδο (input) του προβλήματος, δηλαδή τα δεδομένα του, στην έξοδο (output), δηλαδή στη λύση του προβλήματος. Κάθε είσοδος που ικανοποιεί τις προδιαγραφές του προβλήματος καλείται νόμιμη (legal) και λέμε ότι ορίζει ένα συγκεκριμένο στιγμιότυπο (instance) του προβλήματος. Ένας αλγόριθμος επιλύει ένα πρόβλημα όταν για κάθε στιγμιότυπο του εν λόγω προβλήματος, τερματίζει μετά από πεπερασμένο χρόνο παράγοντας σωστή έξοδο.

1.1.1 Δομές Δεδομένων

Αντικείμενο των «Δομών Δεδομένων» είναι η επισταμένη μελέτη των υλοποιήσεων των συχνότερα εμφανιζόμενων Αφηρημένων Τύπων Δεδομένων (ΑΤΔ) (Abstract Data Types - ADT). Ως αφηρημένος τύπος δεδομένων ορίζεται ένα σύνολο, με μια συλλογή πράξεων επί των στοιχείων του συνόλου.

1.2 Ανάλυση Αλγορίθμων

Ανάλυση αλγορίθμου (algorithm analysis) καλείται η εύρεση των πόρων (resources) που απαιτούνται για να τρέξει. Με άλλα λόγια, ο χρόνος (time) περάτωσης του και ο αναγκαίος για τους υπολογισμούς χώρος (space), μετρούμενος σε αποθηκευτικές θέσεις (memory locations). Οι δύο αυτοί δείκτες μετρήσεως της αποτελεσματικότητας του εκάστοτε αλγορίθμου συνιστούν την πολυπλοκότητα χρόνου και χώρου του (time and space complexity).

1.2.1 Μοντέλα Υπολογισμού - Μοντέλο Μηχανής Τυχαίας Προσπελάσεως RAM (Random Access Machine Model)

Αναφέρεται σε συστήματα του ενός επεξεργαστή γενικού σκοπού δίχως τη δυνατότητα τελέσεως ταυτόχρονων ενεργειών. Δηλαδή, σε ένα σύστημα που: (α) διαθέτει τους αναγκαίους καταχωρητές (registers), έναν συσσωρευτή (accumulator) και μια ακολουθία αποθηκευτικών θέσεων με διευθύνσεις $0, 1, 2, \dots$ οι οποίες συνιστούν την κύρια μνήμη του και (β) είναι σε θέση να εκτελεί τις αριθμητικές πράξεις $\{+, -, *, /, \text{mod}\}$, να παίρνει αποφάσεις διακλάδωσης (τύπου if) βάσει των τελεστών $\{=, <, >, \leq, \geq, \neq\}$ και να διαβάζει και να γράφει από και προς τις θέσεις μνήμης.

Οι στοιχειώδεις πράξεις (primitive operations) που αναφέρονται στο (β) πρέπει να χρεωθούν κάποιο χρόνο. Υπάρχουν δύο θεωρήσεις, με την πρώτη να είναι η μέτρηση μοναδιαίου κόστους (unit cost measure) όπου σε κάθε πράξη χρεώνεται σταθερό (πεπερασμένο) κόστος, ανεξαρτήτως του μήκους της δυαδικής αναπαράστασης των τελεστών (operands). Η δεύτερη είναι η μέτρηση λογαριθμικού κόστους (logarithmic cost measure), η οποία θεωρεί πως η πράξη

παίρνει χρόνο ανάλογο με το μήκος της δυαδικής αναπαράστασης των τελεστών.

1.2.2 Ανάλυση Χειρότερης Περιπτώσεως

Με τον όρο ανάλυση χειρότερης περιπτώσεως (worst case analysis) ονομάζουμε την μέγιστη τιμή που μπορεί να πάρει ο χρόνος τρεξίματος ή ο χώρος ενός αλγορίθμου για οποιαδήποτε είσοδο με συγκεκριμένο μέγεθος n . Επομένως, η ανάλυση χειρότερης περίπτωσης βρίσκει το άνω όριο στην συμπεριφορά του αλγορίθμου, όταν του δοθεί μια νόμιμη είσοδος μεγέθους n .

1.2.3 Ανάλυση Μέσης Περιπτώσεως

Σε περίπτωση που είναι γνωστή η κατανομή πιθανότητας επί του συνόλου των στιγμιοτύπων του εν λόγω προβλήματος, τότε είναι δυνατή η ανάλυση μέσης ή αναμενόμενης περίπτωσης (average/expected case analysis). Αυτή μας δίνει τη μέση ή την αναμενόμενη συμπεριφορά του αλγορίθμου, όταν του δοθεί μια νόμιμη είσοδος με συγκεκριμένο μέγεθος n .

1.2.4 Ανάλυση Επιμερισμένης ή Κατανεμημένης Περιπτώσεως

Η ανάλυση της επιδόσεως μιας δομής ως μέσος όρος επιδόσεως επί ακολουθίας πράξεων, είναι γνωστός ως ανάλυση κατανεμημένης ή επιμερισμένης περιπτώσεως (amortized-case analysis). Έστω $T(n)$ ο μέγιστος χρόνος εκτελέσεως μιας οποιασδήποτε ακολουθίας n πράξεων επί μιας δομής. Ως επιμερισμένος ή κατανεμημένος χρόνος για μια πράξη ορίζεται το πηλίκο $T(n)/n$. Αυτό σημαίνει πως αν η επιμερισμένη επίδοση μιας δομής είναι $f(n)$, τότε μια οποιαδήποτε ακολουθία n πράξεων κοστίζει το πολύ $nf(n)$. Δύο ισοδύναμες τεχνικές επιμερισμένης αναλύσεως είναι η μέθοδος λογαριασμού τραπεζίτη (banker account method) και η μέθοδος συναρτήσεως δυναμικού (potential function method).

Μέθοδος Λογαριασμού Τραπεζίτη ή Λογιστική

Κατά την μέθοδο λογαριασμού τραπεζίτη (banker account method) ή λογιστική μέθοδο (accounting method), κάθε πράξη χρεώνεται ένα κατανεμημένο ή επιμερισμένο κόστος (amortized cost), το οποίο ενδεχομένως να είναι μεγαλύτερο ή μικρότερο από το αντίστοιχο πραγματικό. Η επιλογή του κατανεμημένου κόστους πρέπει να γίνει κατάλληλα έτσι ώστε να προσεγγίζει το μέσο κόστος της πράξεως σε μια οποιαδήποτε ακολουθία πράξεων και το επιμέρους κατανεμημένο κόστος όλων των πράξεων, αθροιζόμενο, να φράσσει από πάνω το πραγματικά παρατηρούμενο χειρότερο κόστος της ακολουθίας. Η διαφορά μεταξύ πραγματικού και κατανεμημένου κόστους χαρακτηρίζεται ως πίστωση (credit) και δηλώνει είτε το πλεόνασμα που κατατίθεται προς μελλοντική χρήση, κατά την εξυπηρέτηση των επόμενων πράξεων, είτε το δάνειο που λαμβάνεται από τα αποθεματικά, για την κάλυψη των τρεχουσών αναγκών μιας πράξεως. Η συγκεκριμένη πρακτική, χρεώνει τις «φθηνές» πράξεις κάτι παραπάνω ώστε να καλυφθεί το επιπλέον, από το μέσο

παρατηρούμενο, κόστος των «ακριβών» πράξεων.

Μέθοδος Δυναμικού

Η μέθοδος δυναμικού (potential method) στηρίζεται στην ιδέα της απεικόνισης της κατάστασης μιας δομής ή ενός αλγορίθμου A μέσω μιας συνάρτησης δυναμικού:

$$\Phi: A \rightarrow \mathbb{R}$$

Αρχικά αποδίδεται μια αρχική τιμή $\Phi(A_0)$. Μετά την i -στή πράξη o_i , πραγματικού κόστους c_i , έχουμε μετάβαση από την κατάσταση A_{i-1} στην A_i και μεταβολή του δυναμικού κατά:

$$\Delta\Phi_i = \Phi(A_i) - \Phi(A_{i-1})$$

Το κατανεμημένο κόστος c_i της o_i ορίζεται ως:

$$c'_i = c_i + \Delta\Phi_i$$

Δηλαδή, το πραγματικό κόστος συν τη μεταβολή που επήλθε στο δυναμικό εξ αιτίας της o_i . Κεντρικό ρόλο στην ανάλυση δυναμικού, παίζει η εκλογή της κατάλληλης συνάρτησης δυναμικού Φ . Χάρης στην τελευταία, κάποιες πράξεις χρεώνονται περισσότερο (όταν $\Delta\Phi_i > 0$) και κάποιες λιγότερο (όταν $\Delta\Phi_i < 0$), συνολικά όμως, επιτυγχάνεται ορθή ερμηνεία της πολυπλοκότητας της A .

2.1 Ουρά Προτεραιότητας Σωρού

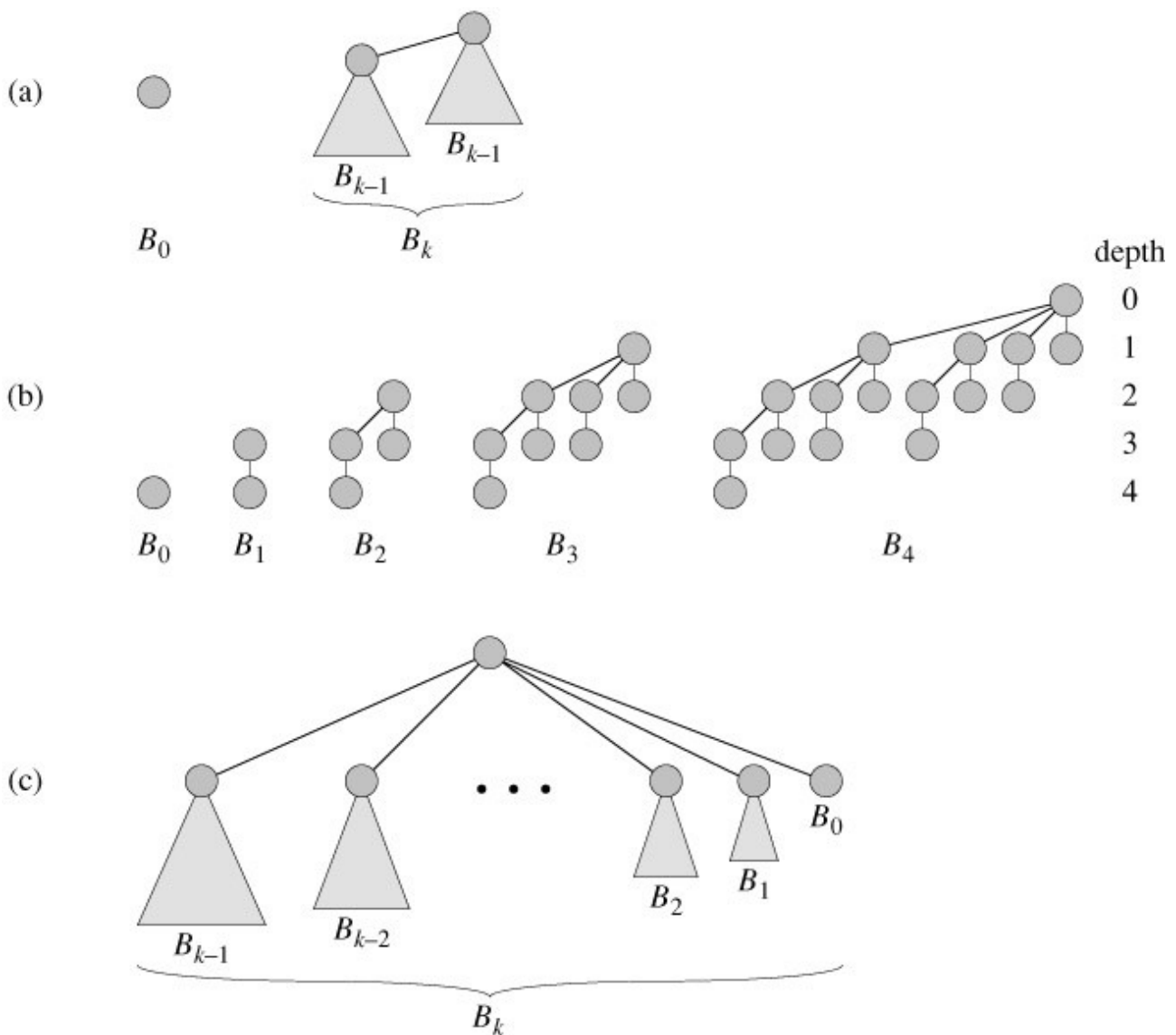
Με τον όρο **σωρός (heap)** ή **ουρά προτεραιότητας (priority queue)** περιγράφουμε έναν αφηρημένο τύπο δεδομένων (ΑΤΔ) που αποτελείται από ένα σύνολο αντικειμένων, το καθένα εκ των οποίων έχει έναν πραγματικό αριθμό ως τιμή κλειδιού-προτεραιότητας, και υποστηρίζει τις ακόλουθες πράξεις:

- **MakeHeap()**: Επιστρέφει μία νέα, άδεια σωρό.
- **Insert(H, i)**: Επιστρέφει τη σωρό που δημιουργείται αν στη σωρό H εισάγουμε το αντικείμενο i.
- **Meld(H1,H2)**: Επιστρέφει τη σωρό που δημιουργείται από την συνένωση των σωρών H1 και H2 (οι σωροί H1 και H2 δεν μπορούν να προσπελαστούν)α.
- **Find-min(H)**: Επιστρέφει τον κόμβο που περιέχει το ελάχιστο αντικείμενο στη σωρό H.
- **Delete-min(H)**: Επιστρέφει τη σωρό που δημιουργείται αν από τη σωρό H αφαιρέσουμε τον κόμβο με το ελάχιστο αντικείμενο.
- **Delete(H, e)**: Επιστρέφει τη σωρό που προκύπτει από την αφαίρεση του αντικειμένου e από τη σωρό H.
- **Decrease-key(H, e, k)**. Μειώνει την τιμή του αντικειμένου e, το οποίο βρίσκεται στη σωρό H, στη νέα τιμή k

Οι παραπάνω ορισμοί αφορούν την ουρά προτεραιότητας ελαχίστου. Σ' αυτήν την κατηγορία ανήκει και η απόλυτη σωρός Fibonacci, την οποία θα μελετήσουμε παρακάτω.

2.2 Δένδρο Διατάξης Σωρού

Οι περισσότερες σωροί στηρίζονται στα δέντρα διάταξης σωρού τα οποία δίνουν εφαρμογές σωρών με λογαριθμικό κόστος για όλες τις διεργασίες. Τα **δένδρα διατάξης σωρού (heap-ordered trees)** είναι δένδρα τα οποία τηρούν την αμετάβλητη συνθήκη σωρού ελαχίστου: Κανένα παιδί δεν έχει μικρότερη προτεραιότητα από αυτή του πατέρα του.



Σχήμα: (a) Αναδρομικός ορισμός δυωνυμικού δένδρου, (b) στιγμιότυπα διωνυμικών δένδρων B_0 - B_4 , (c) εναλλακτική θεώρηση δυωνυμικού δένδρου.

2.3 Σωρός Fibonacci

Αποτελεί μια παραλλαγή των δυωνυμικών ουρών. Δομικό στοιχείο τη σωρού Fibonacci είναι τα δένδρα διατάξεως σωρού, τα οποία τηρούν την αμετάβλητη συνθήκη σωρού ελαχίστου. Ένας σωρός Fibonacci ορίζεται ως ένα αδιάτακτο δάσος από ξένα, ως προς τα στοιχεία που φέρουν, μεταξύ τους δένδρα διατάξεως σωρού. Θεμελιώδης πράξη τους είναι η ένωση δύο δένδρων διατάξεως σωρού σε ένα, με σύγκριση των προτεραιοτήτων των ριζών και τοποθέτηση της ρίζας με την μεγαλύτερη προτεραιότητα ως παιδί της ρίζας με την μικρότερη προτεραιότητα. Το πλήθος των παιδιών ενός κόμβου καλείται τάξη του κόμβου και ένας κόμβος μπορεί να είναι είτε **σημαδεμένος (marked)** είτε **ασημάδευτος (unmarked)**.

2.3.1 Περιγραφή Πράξεων

Τα συστατικά δένδρα χρησιμοποιούν την αναπαράσταση πρώτου παιδιού-δεξιού αδελφού, με έναν επιπλέον, ανά κόμβο, δείκτη, προς τον αριστερό αδελφό έτσι ώστε τα παιδιά κάθε κόμβου να σχηματίζουν κυκλική λίστα σε κόμβο της οποίας δείχνει ο πατέρας, ενώ και οι ρίζες του δάσους συμμετέχουν σε κυκλική λίστα.

Έυρεση Ελαχίστου. Η πράξη της ευρέσεως του ελαχίστου στοιχείου υλοποιείται σε σταθερό χρόνο, μέσω ενός δείκτη προς την ρίζα με το ελάχιστο στοιχείο.

Ένθεση Στοιχείου. Για την ένθεση ενός στοιχείου x στον σωρό, πρώτον σχηματίζουμε το στοιχειώδες δένδρο για την στέγαση του x (κόστος $O(1)$), δεύτερον το συνενώνουμε με την κυκλική λίστα των ριζών του δάσους (κόστος $O(1)$) και τρίτον, βρίσκουμε το ελάχιστο με απλή σύγκριση σταθερής πολυπλοκότητας της προτεραιότητας του x με την ελάχιστη προτεραιότητα της δομής.

Συγχώνευση Σωρών. Υλοποιείται ως συνένωση των αντίστοιχων κυκλικών ουρών των ριζών. Κατόπιν, θέτουμε ως νέο ελάχιστο το ελάχιστο των δύο ελαχίστων στοιχείων, κοστίζει συνεπώς σταθερό χρόνο.

Απόσβεση Ελαχίστου. Αρχικά, το δένδρο με το ελάχιστο στοιχείο αποκόπτεται από το δάσος (από την λίστα των ριζών) και διαγράφουμε την ρίζα του δένδρου αποκόποντάς την από τα παιδιά της. Έπειτα, συνενώνουμε την λίστα των ορφανών παιδιών με τη λίστα του δάσους και εφαρμόζουμε διαδοχικές συνενώσεις δύο οποιονδήποτε δένδρων ίδιας τάξεως, έως ότου προκύψει δάσος με δένδρα διακριτής τάξεως. Τέλος, σαρώνουμε τα μέλη του δάσους ώστε να εντοπιστεί το νέο ελάχιστο.

Αλλαγή Προτεραιότητας. Αποκόπτουμε από τον πατέρα του τον κόμβο του οποίου την προτεραιότητα θα αλλάξουμε, μαζί και το υποδένδρο του, και καθώς ο πατέρας του χάνει ένα παιδί του μειώνουμε την τάξη. Ενσωματώνουμε το υποδένδρο στο δάσος, μέσω της εισαγωγής του κόμβου που αποκόψαμε στη λίστα των ριζών. Στην συνέχεια, εξετάζουμε τον πατέρα του κόμβου που αλλάξαμε την προτεραιότητα. Αν είναι ρίζα, η πράξη ολοκληρώθηκε. Διαφορετικά, εξετάζουμε το σημάδι του. Εάν δεν είναι σημαδεμένος, απλώς τον σημαδεύουμε. Αλλιώς, αφαιρούμε το σημάδι και τον αποκόπτουμε από τον πατέρα του, ενσωματώνοντάς τον στη λίστα των ριζών. Κατά συνέπεια, ο πατέρας του χάνει ένα παιδί και τον αντιμετωπίζουμε αναδρομικά. Δηλαδή, του μειώνουμε την τάξη και αν δεν είναι σημαδεμένος τον σημαδεύουμε, διαφορετικά, τον ενσωματώνουμε στη λίστα των ριζών αφαιρώντας το σημάδι του κ.ο.κ.

Διαγραφή Στοιχείου. Αποκόπτουμε από τον πατέρα του τον προς διαγραφή κόμβο, μαζί και το υποδένδρο του, και καθώς ο πατέρας του χάνει ένα παιδί του μειώνουμε την τάξη. Στην συνέχεια, ενσωματώνουμε τα υποδένδρα των παιδιών στο δάσος ενώνοντας την αντίστοιχη κυκλική τους λίστα με τη λίστα των ριζών και διαγράφοντας τον πατέρα τους. Έπειτα, εξετάζουμε τον πατέρα του κόμβου που διαγράψαμε. Αν είναι ρίζα, η πράξη ολοκληρώθηκε. Διαφορετικά, εξετάζουμε το σημάδι του. Εάν δεν είναι σημαδεμένος, απλώς τον σημαδεύουμε. Αλλιώς, αφαιρούμε το σημάδι και τον αποκόπτουμε από τον πατέρα του, ενσωματώνοντάς τον στη λίστα των ριζών. Κατά συνέπεια, ο πατέρας του χάνει ένα παιδί και τον αντιμετωπίζουμε αναδρομικά. Δηλαδή, του μειώνουμε την τάξη και αν δεν είναι σημαδεμένος τον σημαδεύουμε, διαφορετικά, τον ενσωματώνουμε στη λίστα των ριζών αφαιρώντας το σημάδι του κ.ο.κ.

2.5.2 Ανάλυση Πολυπλοκότητας

Λήμμα: Ένας κόμβος τάξεως k διαθέτει τουλάχιστον $F_{k+1} \geq \phi_k$ απογόνους, συμπεριλαμβανομένου του εαυτού του, όπου F_{k+2} είναι ο $(k+2)$ -στός αριθμός Fibonacci και $\phi = (1+\sqrt{5})/2$ η χρυσή αναλογία.

Θεώρημα: Έστω μια αναμεμιγμένη ακολουθία πράξεων επί μιας, αρχικώς άδειας, σωρού Fibonacci. Το επιμερισμένο κόστος μιας αποσβέσεως ελαχίστου ή αποσβέσεως στοιχείου είναι λογαριθμικό στο πλήθος των στοιχείων, ενώ κάθε άλλη πράξη επιδεικνύει σταθερό κόστους επιμερισμένη χρονική συμπεριφορά. Στην περίπτωση της αποσβέσεως ελαχίστου, αν η ρίζα με το ελάχιστο στοιχείο είναι τάξεως k , τότε:

$$\phi_k \leq n \Rightarrow k \leq \log_{\phi} n \leq 1.44 \log n$$

Συνεπώς, η απόσβεσή της αυξάνει το πλήθος των δένδρων, το πολύ, κατά $1.44 \log n$. Κάθε συνένωση κοστίζει $O(1)$ πραγματικό χρόνο, αλλά μειώνει το πλήθος των δένδρων, άρα και το δυναμικό, κατά ένα. Επομένως, το δυναμικό, συνολικά, αυξάνει το πολύ $O(\log n)$.

Μια μείωση προτεραιότητας έχει σταθερό επιμερισμένο κόστος, διότι μεταβάλλει το δυναμικό το πολύ κατά συν τρία, μείον το πλήθος των συνεχόμενων αποκοπών που προκαλεί, ενώ το πραγματικό κόστος ισούται με το πλήθος των συνεχόμενων αποκοπών. Η πρώτη αποκοπή αυξάνει το πλήθος των δένδρων, μετατρέποντας έναν, ενδεχομένως μη σημαδεμένο, κόμβο σε ρίζα (συν ένα για δυναμικό και πραγματικό κόστος). Οι υπόλοιπες αποκοπές, πλην της τελευταίας, μετατρέπουν έναν σημαδεμένο κόμβο σε μη σημαδεμένη ρίζα (συν ένα, μείον δύο για το δυναμικό, συν ένα για το πραγματικό κόστος), ενώ η τελευταία αποκοπή μπορεί να σημαδέψει έναν μη σημαδεμένο κόμβο (συν δύο για το δυναμικό, συν ένα για το πραγματικό κόστος).

Η πολυπλοκότητα για την απόσβεση στοιχείου προκύπτει άμεσα, καθώς κοστίζει σταθερό πραγματικό χρόνο, ενώ, ταυτόχρονα, αυξάνει το πλήθος των δένδρων το πολύ κατά $O(\log n)$. Η αύξηση της προτεραιότητας είναι λογαριθμική πράξη, καθώς ισοδυναμεί με μια απόσβεση και μια ένθεση. Τέλος, η ένθεση αυξάνει κατά ένα το πλήθος των δένδρων, ενώ οι υπόλοιπες πράξεις έχουν σταθερό πραγματικό κόστος.

Διεργασία	Κόστος χειρότερης περίπτωσης	Τελικό κόστος
Ένθεση στοιχείου	$O(1)$	$O(1)$
Απόσβεση ελαχίστου	$O(n)$	$O(\log n)$
Αλλαγή προτεραιότητας	$O(n)$	$O(1)$

3.1 Απόλυτη σωρός Fibonacci (Strict Fibonacci Heap)

Οι σωροί αναπτύχθηκαν σε θεωρητικό επίπεδο από τους Gerth Stølting Brodal, George Lagogiannis και Robert E. Tarjan στο 44^ο Συμπόσιο ACM στη Θεωρία του Προγραμματισμού το 2012.

Οι εφαρμογές που χρησιμοποιούνται οι σωροί της κατηγορίας των ουρών προτεραιότητας είναι η εξεύρεση συντομότερου μονοπατιού καθώς και η εξεύρεση του ελάχιστου Spanning Tree. Οι προηγούμενες εκδόσεις των ουρών ήταν αρκετά πολύπλοκες στην εφαρμογή και έτσι οι απόλυτες σωροί Fibonacci με τη χρήση δεικτών κατάφεραν να μειώσουν την πολυπλοκότητα.

Η απόλυτη σωρός Fibonacci είναι μια παραλλαγή της απλής σωρού Fibonacci και είναι η πρώτη εφαρμογή σωρού που βασίζεται αποκλειστικά σε κόμβους, με την πολυπλοκότητα των πράξεών της να είναι ίδια με αυτήν της απλής σωρού στη χειρότερη περίπτωση. Συγκεκριμένα, πετυχαίνει μια δομή δεδομένων γραμμικού χώρου, η οποία υποστηρίζει τις διεργασίες make-heap, insert, find-min, meld και decrease-key με πολυπλοκότητα χρόνου χειρότερης περίπτωσης $O(1)$ και τις delete και delete-min με πολυπλοκότητα χρόνου χειρότερης περίπτωσης $O(\log n)$.

Οι σωροί στην εργασία αυτή χρησιμοποιούν την αρχή του περιστρώνα. Οι δομικές παραβάσεις είναι υποδένδρα τα οποία «κόβονται» (και προσκολλούνται στην ρίζα) όπως και στις σωρούς Fibonacci. Η καινοτομία αυτής της εφαρμογής είναι, ότι όταν δύο σωροί συνενώνονται (meld) οι δομές που κρατούνται για το μικρότερο δένδρο παραμερίζονται καθώς όλοι του οι κόμβοι γίνονται **παθητικοί**. Μπορούμε να σημαδεύουμε όλους τους κόμβους ενός δένδρου παθητικούς σε χρόνο $O(1)$, χρησιμοποιώντας μια κοινή σημαία.

Κάποιες βασικές έννοιες που θα χρειαστούμε στη συνέχεια είναι οι εξής: Μια σωρός η αντικειμένων αντιπροσωπεύεται από ένα δένδρο διάταξης σωρού με n κόμβους. Κάθε κόμβος αποθηκεύει ένα στοιχείο. Το **μέγεθος** (size) ενός δέντρου είναι ο αριθμός των κόμβων που περιέχει. Ο **βαθμός** (degree) ενός κόμβου είναι ο αριθμός των παιδιών του κόμβου. Υποθέτουμε ότι όλα τα αντικείμενα είναι διαφορετικά και αν όχι, θα σπάσουμε τις ισοπαλίες με το αναγνωριστικό στοιχείο του κάθε αντικειμένου. Η αρχή της σωρού ικανοποιείται από όλους τους κόμβους. Συγκεκριμένα, έστω το $x.key$ που χαρακτηρίζει το βασικό της στοιχείο που είναι αποθηκευμένο στον κόμβο x . Αν το x είναι ένα παιδί του y τότε $x.key > y.key$. Συνεπάγεται ότι το στοιχείο με το ελάχιστο κλειδί αποθηκεύεται στη ρίζα. Η βασική ιδέα της κατασκευής είναι να διασφαλίσουμε ότι όλοι οι κόμβοι έχουν λογαριθμικό βαθμό, ότι μια λειτουργία συγχώνευσης (meld) κάνει τη ρίζα με το μεγαλύτερο κλειδί το παιδί της ρίζας με το μικρότερο κλειδί και μια λειτουργία μείωσης του κλειδιού (decrease-key) αποκόπτει το υποδένδρο του κόμβου και το τοποθετεί σαν υποδένδρο της ρίζας.

Χρησιμοποιούμε επίσης τις ακόλουθες έννοιες, για να γίνουν οι απαραίτητες αναδομήσεις. Κάθε κόμβος είναι σημαδεμένος είτε ως **ενεργός** (active) είτε ως **παθητικός** (passive). Ένας ενεργητικός κόμβος με παθητικό πατέρα καλείται

ενεργή ρίζα (active root). Η **τάξη** (rank) ενός ενεργού κόμβου είναι ο αριθμός των ενεργών παιδιών του, ενώ σε κάθε ενεργό κόμβο ανατίθεται ένας μη-αρνητικός ακέραιος **ζημία** (loss). Η **συνολική ζημία** (total loss) της σωρού είναι το άθροισμα των ζημιών όλων των ενεργών κόμβων. Ένας παθητικός κόμβος είναι **συνδέσιμος** (linkable) αν όλα τα παιδιά του είναι παθητικά. Για να κρατηθεί ο βαθμός των κόμβων λογαριθμικός, διατηρούμε όλους τους κόμβους εκτός της ρίζας, σε μια ουρά Q. Ένας κόμβος που δεν είναι ρίζα έχει **θέση** (position) p αν είναι ο p-οστός κόμβος στην ουρά Q.

3.1.1 Περιγραφή βασικών πράξεων

Αφού περιγράψαμε τις βασικές έννοιες, μπορούμε να συνεχίσουμε με την περιγραφή των βασικών πράξεων της σωρού.

Δημιουργία σωρού - MakeHeap() : Φτιάχνουμε μία νέα, άδεια σωρό και επιστρέφουμε ένα δείκτη σε αυτήν. Απαιτείται σταθερός χρόνος.

Εύρεση ελαχίστου - Find_min() : Επιστρέφουμε τον κόμβο που βρίσκεται στη ρίζα του δένδρου. Απαιτείται σταθερός χρόνος.

Ένθεση κόμβου - Insert() : Για την πράξη της ένθεσης νέου κόμβου δημιουργούμε ένα νέο δένδρο ενός κόμβου και κατόπιν, το ενώνουμε με την ήδη υπάρχουσα σωρό. Στο τέλος, απαιτούνται ελάττωση ζημίας, ενεργών ριζών και του βαθμού της ρίζας (βλ. 3.1.2).

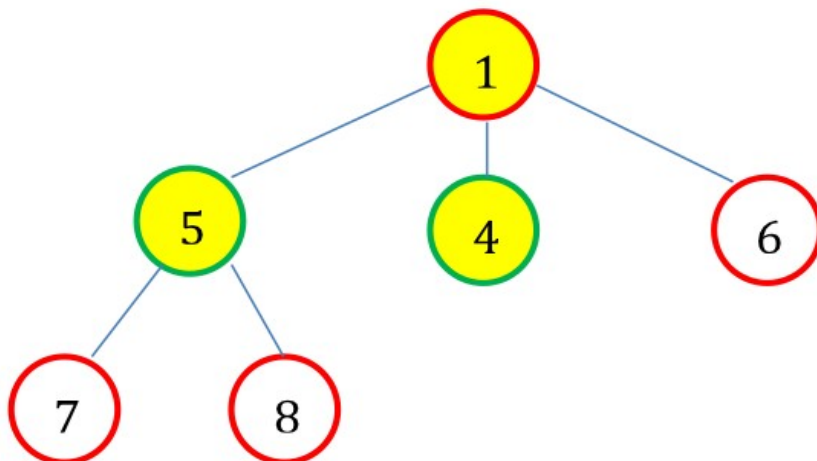
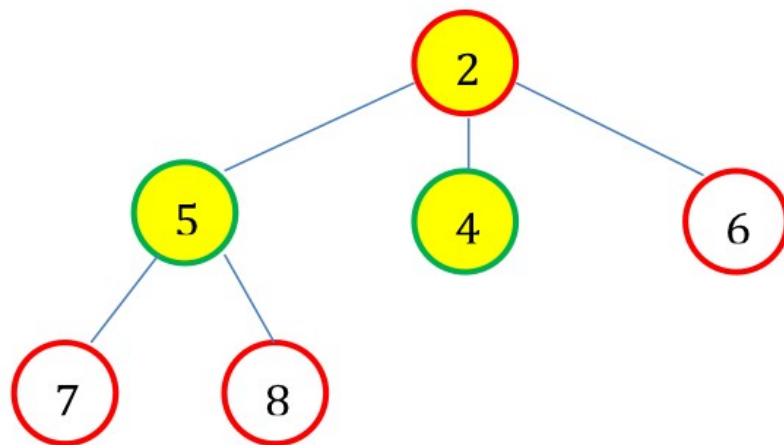
Διαγραφή κόμβου - Delete_node() : Για τη διαγραφή κόμβου μειώνουμε το κλειδί του κόμβου (μέσω της decrease-key) σε 0 και μετά κάνουμε μείωση ελαχίστου (μέσω της delete-min).

Ελάττωση κλειδιού - Decrease-key() : Για τη μείωση του κλειδιού (key) ενός κόμβου x σε μια σωρό με ρίζα z, ξεκινάμε μειώνοντας το κλειδί του αντικειμένου. Αν ο x είναι η ρίζα, έχουμε τελειώσει. Διαφορετικά, αν $x.key < z.key$, αντιμεταθέτουμε (swap) τα αντικείμενα x και z. Στην ουσία, όπως έχουμε κάνει την αναπαράσταση των κόμβων με δείκτες, ο κάθε κόμβος δείχνει σε ένα εξωτερικό struct τύπου αντικειμένου το οποίο έχει έναν δείκτη στον κόμβο που το περιέχει, οπότε αλλάζοντας απλά τους δείκτες έχουμε τελειώσει εύκολα.

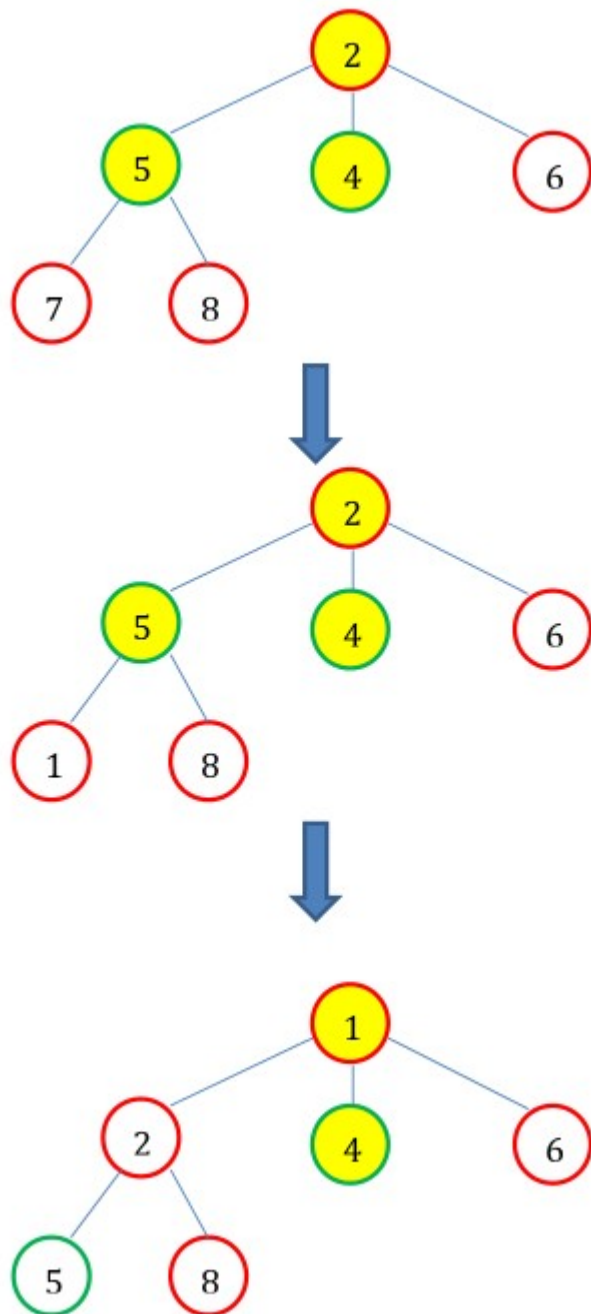
Ας θέσουμε ως y τον πατέρα του x και ας κάνουμε τον x παιδί της ρίζας. Αν ο x ήταν ενεργός κόμβος αλλά όχι και ενεργή ρίζα, τότε τον κάνουμε ενεργή ρίζα με ζημία 0, ενώ η τάξη του κόμβου μειώνεται κατά 1. Αν τώρα ο y ήταν ενεργός, αλλά όχι ενεργή ρίζα, τότε η ζημία του αυξάνεται κατά 1.

Αν είναι δυνατόν, επιχειρούμε ελάττωση ζημίας και μετά κάνουμε έξι ελαττώσεις ενεργών ριζών και τέσσερις ελαττώσεις βαθμού ρίζας (στο βαθμό που αυτό είναι δυνατό). Οι μετασχηματισμοί έχουν σταθερό κόστος.

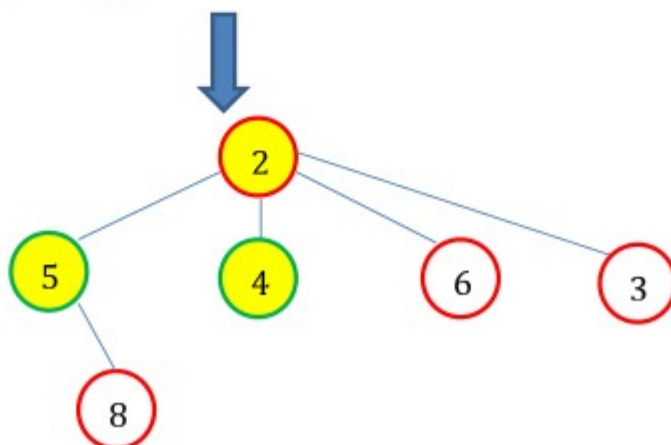
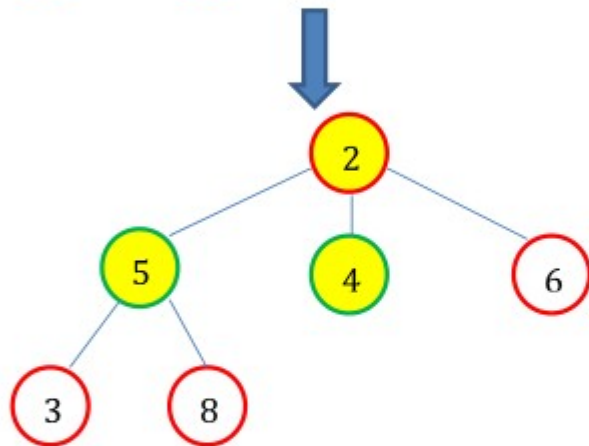
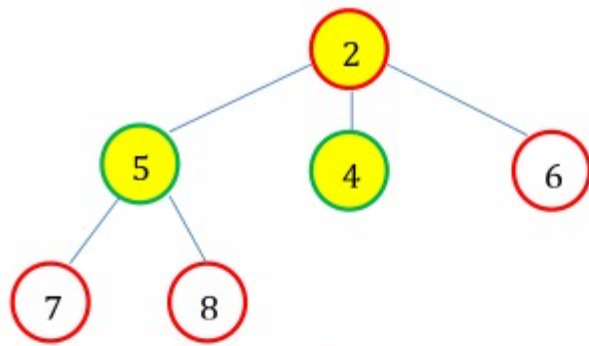
Σχήμα: Παράδειγμα της decrease-key



Περίπτωση κατά
την οποία κάνουμε
ελάττωση κλειδιού
της ρίζας



Περίπτωση κατά την οποία μειώνουμε το κλειδί του κόμβου 7. Γίνεται εναλλαγή (swap) αρχικά με τον κόμβο-πατέρα 5 και μετά με τον κόμβο πατέρα 2.



Περίπτωση κατά την οποία αλλάζουμε το κλειδί του κόμβου 7. Είναι μικρότερο από το κλειδί του πατέρα (5) αλλά μεγαλύτερο από της ρίζας (2). Έτσι γίνεται παιδί της ρίζας.

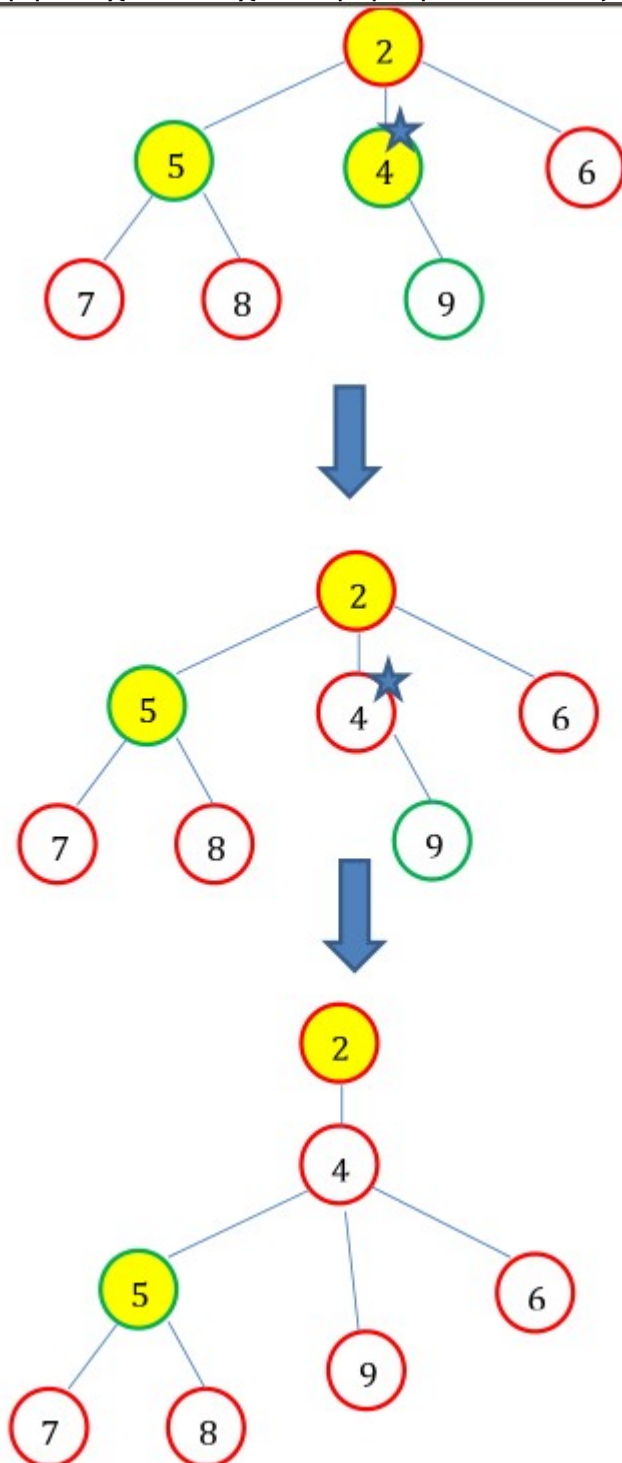
Διαγραφή ελαχίστου - Delete-min() : Για τη διαγραφή ελαχίστου σε ένα δένδρο με ρίζα z , βρίσκουμε πρώτα τον κόμβο x που περιέχει το ελάχιστο κλειδί μεταξύ των παιδιών της ρίζας. Αν ο x είναι ενεργός, τον κάνουμε παθητικό και όλα τα ενεργά παιδιά του, γίνονται ενεργές ρίζες. Κάθε άλλο παιδί του z γίνεται παιδί του x , ενώ τα παθητικά συνδέσιμα παιδιά του x γίνονται τα δεξιότερα παιδιά του. Αφαιρούμε τον x από την ουρά Q (αφού πλέον είναι ρίζα) και καταστρέφουμε τον z .

Στη συνέχεια, επαναλαμβάνουμε δύο φορές τα παρακάτω:

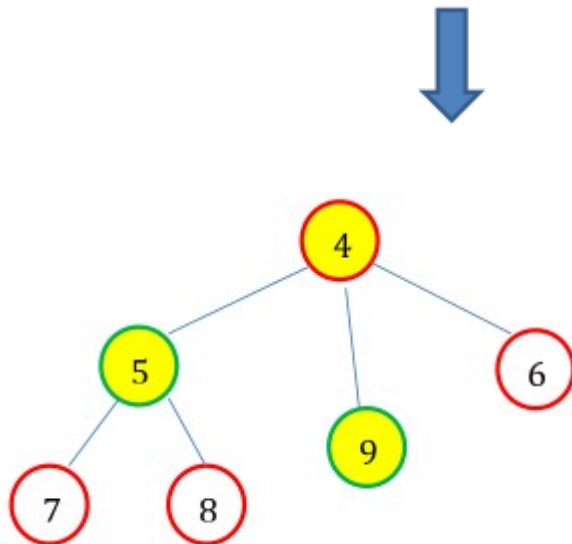
- Μετακινούμε τον μπροστινό κόμβο y της ουράς Q , στο τέλος
- Ενώνουμε τα δύο δεξιότερα παιδιά του y στον x , αν είναι παθητικά

- Κάνουμε μια ελάττωση ζημίας, αν είναι δυνατό
- Κάνουμε ελαττώσεις ενεργών ριζών και βαθμού ρίζας μέχρις ότου καμία από τις δύο ενέργειες να είναι δυνατή

Η σύνδεση έχει κόστος $O(\log n)$ όπως και οι μετασχηματισμοί, οπότε συνολικά η διαγραφή ελαχίστου έχει λογαριθμικό κόστος



Παράδειγμα της delete-min



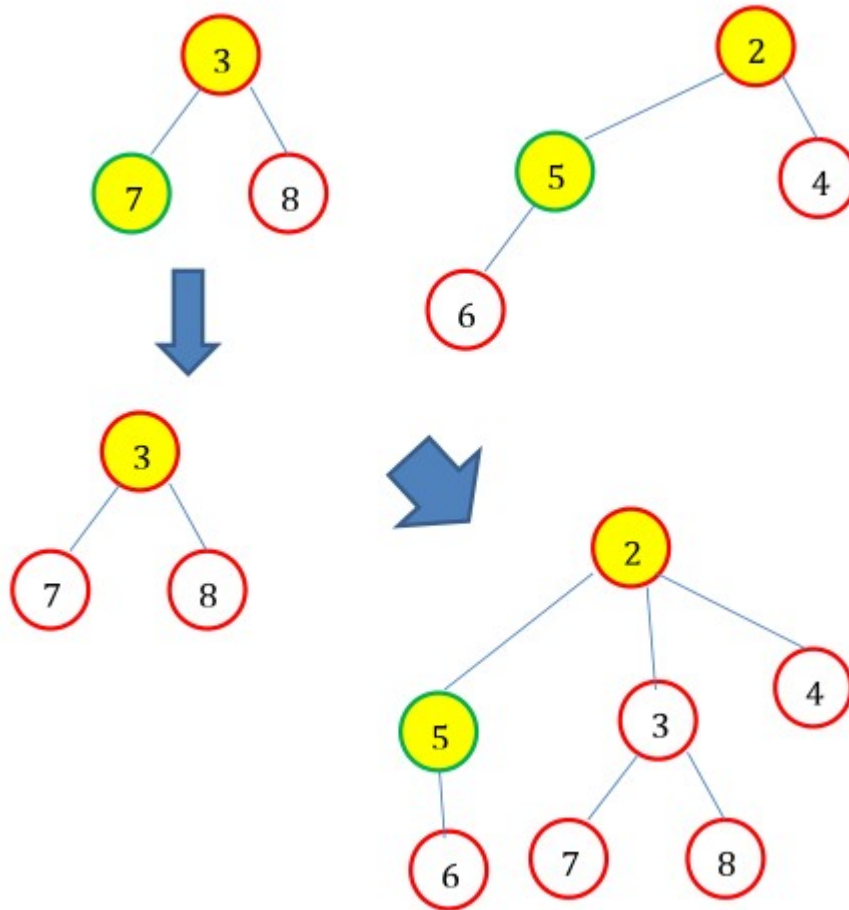
Συγχώνευση σωρών - Meld() : Για τη συγχώνευση δύο σωρών με ρίζες x και y αντίστοιχα, βρίσκουμε πρώτα την σωρό με το μεγαλύτερο μέγεθος. Έπειτα, δημιουργούμε μια νέα σωρό, με το μέγεθος της μεγαλύτερης από τις συγχωνευμένες σωρούς και κάνουμε τους κόμβους της σωρού με ρίζα x παθητικές (αυτό απαιτεί χρόνο $O(1)$ λόγω του δείκτη που έχουν όλοι οι ενεργοί κόμβοι, ο οποίος δείχνει σε ένα struct τύπου active record).

Ενώνουμε τις ουρές των δύο σωρών, αφού έχουμε κάνει παιδί της ρίζας με το μικρότερο κλειδί, τη ρίζα με το μεγαλύτερο κλειδί, έστω v και u αντίστοιχα. Η προκύπτουσα ουρά της νέας σωρού θα είναι της μορφής $Q = Q_x \& \{v\} \& Q_y$. Με το "&" υπονοούμε ένωση, ενώ τα Q_x και Q_y αναφέρονται στις ουρές των σωρών με ρίζα x και y αντίστοιχα.

Τέλος, κάνουμε ελαττώσεις ενεργών ριζών και βαθμού ρίζας στο βαθμό που αυτές είναι δυνατές.

Η μέθοδος της συγχώνευσης είναι και η κύρια καινοτομία αυτών των σωρών, αφού "ξεχνάει" τη δομή του μικρότερου δένδρου, κάτι που εξαλείφει την ανάγκη για συνδυασμό περίπλοκων δομών δεδομένων.

Το κόστος των μετασχηματισμών στο τέλος είναι σταθερό.



3.1.2 Ανάλυση Πολυπλοκότητας - Σταθερές και Θεωρήματα

Ας υποθέσουμε ότι το $R = 2\log n + 6$. Την τιμή του R την κρατάμε και χρειάζεται μόνο στην ανάλυση, δε χρησιμοποιείται στον αλγόριθμο. Οφείλουμε σε κάθε σημείο του αλγορίθμου να υπακούμε σε κάποιους περιορισμούς:

- I1 (Δομή) : Για όλους τους κόμβους τα ενεργά παιδιά είναι στα αριστερά των παθητικών παιδιών, η ρίζα είναι παθητική και τα παθητικά συνδέσιμα παιδιά της ρίζας είναι τα δεξιότερα παιδιά της. Για έναν ενεργό κόμβο, το i -οστό δεξιότερο ενεργό παιδί έχει τάξη+ζημία (rank+loss) τουλάχιστον $i-1$. Μια ενεργή ρίζα έχει μηδενική ζημία.
- I2 (Ενεργές ρίζες) : Ο συνολικός αριθμός των ενεργών ριζών είναι το πολύ $R+1$.
- I3 (Ζημία) : Η συνολική ζημία είναι το πολύ $R+1$
- I4 (Βαθμοί) : Ο μέγιστος βαθμός της ρίζας είναι $R+3$. Έστω x ένας κόμβος που δεν είναι ρίζα και p είναι η θέση του στη λίστα Q . Αν ο x είναι παθητικός ή ενεργός με θετική ζημία, τότε ο βαθμός του είναι το πολύ $2\log(2n-p) + 9$.
Αλλιώς, αν ο x είναι ενεργός με μηδενική ζημία, τότε μπορεί να έχει βαθμό κατά ένα μεγαλύτερο, δηλαδή το πολύ $2\log(2n-p) + 10$.

Με λίγα λόγια από το I4 συμπεραίνουμε ότι όλοι οι κόμβοι μπορούν να έχουν βαθμό το πολύ $2\log n + 12$. Έτσι, ο μέγιστος βαθμός των ενεργών κόμβων φράσσεται από πάνω και το παρακάτω λήμμα δείχνει ότι η μέγιστη τάξη εξαρτάται από την τιμή του R , για τυχαίες τιμές του R .

Λήμμα: Αν ικανοποιείται το I1 και η συνολική ζημία είναι L , τότε η μέγιστη τάξη είναι το πολύ $\log n + \sqrt{2L} + 2$

Απόδειξη: Αν υποθέσουμε ότι ο x είναι ενεργός κόμβος μέγιστης τάξης $r \geq k + 1 + \log n$, όπου k ο ελάχιστος ακέραιος για τον οποίο ισχύει $k(k+1)/2 \geq L$. Θα αποδείξουμε το αντίθετο ότι το υποδένδρο με ρίζα τον x περιέχει τουλάχιστον $n+1$ κόμβους. Αν T_x είναι το υποδένδρο, κόβουμε όλα τα υποδένδρα που έχουν ρίζες παθητικούς κόμβους. Αν y παιδί του x , z παιδί του y και υπάρχει ένας κόμβος με θετική ζημία στο υποδένδρο που έχει ρίζα τον z , τότε κόβουμε το υποδένδρο του z και αυξάνουμε τη ζημία του y κατά 1 (έτσι ώστε να ικανοποιείται το I1). Αυτό αφαιρεί την θετική ζημία που είχε το υποδένδρο του z και αυξάνει τη ζημία του y κατά μόλις 1, οπότε η συνολική ζημία συνεχίζει να φράσσεται από το L . Τώρα μόνο τα παιδιά του x μπορούν να έχουν θετική ζημία. Μειώνουμε το άθροισμα τάξη+ζημία του i -οστού δεξιότερου παιδιού του x σε $i-1$, μειώνοντας τη ζημία και πιθανώς κόβοντας εγγόνια. Τέλος, για όλους τους κόμβους v εκτός του x κόβουμε τα εγγόνια τους επανηλημένα έτσι ώστε το i -οστό δεξιότερο παιδί τους να έχει πάντα βαθμό $i-1$ ακριβώς. Τα εναπομείναντα υποδένδρα είναι δυωνυμικά δένδρα μεγέθους $2^{\text{βαθμός}(v)}$. Το ελάχιστο μέγεθος ενός τέτοιου υποδενδρου T_x επιτυγχάνεται επαναλαμβάνοντας L φορές το παρακάτω βήμα: κόβουμε ένα

εγγόνι της ρίζας με το μέγιστο βαθμό. Αφού ο μέγιστος βαθμός ενός εγγονιού σε ένα διωνυμικό δένδρο μεγέθους 2^r είναι $r-2$ και υπάρχουν j εγγόνια βαθμού $r-j-1$, υπάρχουν $\sum_{j=1}^k j = k(k+1)/2$ εγγόνια βαθμού $\geq r-k-1$.

Αφού, $k(k+1)/2 \geq L$ κανένα εγγόνι βαθμού $\leq r-k-2$ δεν κόβεται, δηλαδή το $(r-k)$ -οστό δεξιότερο παιδί w του x έχει βαθμό $r-k-1$ και μηδενική ζημία. Με την υπόθεση για το r , ο βαθμός του w είναι $\geq \log n$ και το T_w έχει μέγεθος $\geq n$. Συνεπάγεται ότι το T_x έχει μέγεθος τουλάχιστον $n+1$ το οποίο είναι άτοπο. Αυτό μας δίνει ότι $r < k+1+\log n \leq \log n + \sqrt{(2L)} + 2$, αφού $(k-1)k/2 < L$.

Από το I3 έχουμε ότι $L \leq R + 1$ και αφού $\log n + \sqrt{(2(R+1))} + 2 \leq R$ για $R = 2\log n + 6$ εξάγουμε το παρακάτω πόρισμα:

Πόρισμα : Όλοι οι κόμβοι έχουν τάξη $\leq R$

Το πόρισμα φράσσει τη μέγιστη τάξη πριν από μια διεργασία της σωρού. Αν παραβιάσουμε τα I2 ή I3 προσωρινά μέσα σε μια διεργασία, τότε η αρχή του περιστρώνα εγγυάται ότι μπορούμε να εφαρμόσουμε τους μετασχηματισμούς που θα περιγραφούν στο επόμενο κεφάλαιο. Αν η συνολική ζημία είναι $> R + 1$, τότε υπάρχει κόμβος με ζημία τουλάχιστον 2, ή υπάρχουν δύο κόμβοι με ζημία 1. Παρόμοια, αν υπάρχουν $> R + 1$ ενεργές ρίζες, τότε τουλάχιστον δύο ενεργές ρίζες έχουν την ίδια τάξη. Τέλος, αν ικανοποιούνται τα I1-I3 αλλά η ρίζα παραβιάζει το I4, τότε η ρίζα έχει τουλάχιστον τρία παθητικά συνδέσιμα παιδιά, αφού έχει το πολύ $R + 1$ παιδιά ή εγγόνια που μπορούν να είναι ενεργές ρίζες, δηλαδή το πολύ $R + 1$ παιδιά της ρίζας είναι ενεργές ρίζες ή παθητικοί, μη-συνδέσιμοι κόμβοι

Το μέγεθος του χώρου που απαιτείται για την υλοποίηση αυτή φράσσεται γραμμικά από τον αριθμό των αποθηκευμένων αντικειμένων. Συγκεκριμένα, για κάθε αντικείμενο έχουμε ένα node record και πιθανότατα ένα fix-list record. Ο αριθμός των active records φράσσεται από τον αριθμό των κόμβων, καθώς στη χειρότερη περίπτωση κάθε active record έχει ref-count (δηλαδή, αριθμό των κόμβων που δείχνουν σε αυτό) = 1.

Τέλος, για κάθε σωρό, έχουμε ένα record σωρού και έναν αριθμό από rank-list records ο οποίος φράσσεται από τη μέγιστη τάξη ενός κόμβου συν ένα $(r+1)$, δηλαδή είναι λογαριθμικός ως προς το μέγεθος της σωρού.

Συνολικά, λοιπόν, έχουμε ότι ο χώρος για μία σωρό είναι γραμμικός ως προς τον αριθμό των αποθηκευμένων αντικειμένων.

Η υλοποίηση αυτή επιτυγχάνει, όπως αναφέρθηκε και παραπάνω, χρόνους ίδιους με την υλοποίηση των απλών σωρών Fibonacci. Αυτό σημαίνει πρακτικά, ότι πετυχαίνει χρόνους:

- $O(1)$ για τις
 - ✓ make-heap
 - ✓ insert
 - ✓ find-min
 - ✓ meld
 - ✓ decrease-key
- και $O(\log n)$ για τις
 - ✓ delete και

✓ delete-min

Το κόστος των μετασχηματισμών στην περίπτωση της meld και της decrease-key είναι $O(1)$, ενώ στην delete-min το κόστος είναι $O(\log n)$.

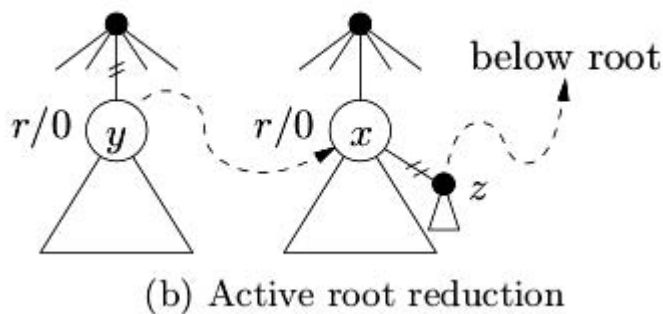
3.1.3 Βασικοί Μετασχηματισμοί

Ο βασικός μετασχηματισμός είναι η αποκοπή ενός υποδένδρου με ρίζα x και η σύνδεσή του ως παιδί ενός κόμβου y . Αν ο x είναι ενεργός τον κάνουμε το αριστερότερο παιδί του y , ενώ αν είναι παθητικός τον κάνουμε το δεξιότερο.

Οι μετασχηματισμοί ελάττωσης (reduction transformations) χρησιμοποιούνται ώστε να διατηρούμε αληθείς τις σταθερές 12-14 που είδαμε παραπάνω.

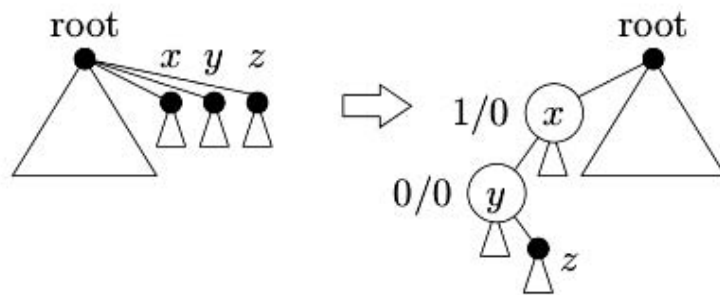
Ελάττωση ενεργών ριζών - Active root reduction : Έστω x και y ενεργές ρίζες της ίδιας τάξης r . Συγκρίνουμε τα κλειδιά τους και βρίσκουμε π.χ. $x.key < y.key$. Συνδέουμε το y και το x και αυξάνουμε την τάξη του x κατά 1. Αν το δεξιότερο παιδί του x , το z είναι παθητικό, τότε το κάνουμε παιδί της ρίζας. Σ' αυτόν το μετασχηματισμό ο αριθμός των ενεργών ριζών μειώνεται κατά 1 και ο βαθμός της ρίζας πιθανώς αυξάνεται κατά 1.

Σχήμα: Ελάττωση ενεργών ριζών



Ελάττωση βαθμού ρίζας - Root degree reduction : Έστω x, y, z τα τρία δεξιότερα παθητικά συνδέσιμα παιδιά της ρίζας. Χρησιμοποιώντας τρεις συγκρίσεις κατατάσσουμε τα κλειδιά και έστω π.χ. $x.key < y.key < z.key$. Σημαδεύουμε τους x και y ως ενεργούς. Συνδέουμε τον z στον y και τον y στον x και κάνουμε το x το αριστερότερο παιδί της ρίζας. Αναθέτουμε στους x και y μηδενική ζημία και τάξη 1 και 0 αντίστοιχα. Σ' αυτόν το μετασχηματισμό τόσο ο x όσο και ο y , αλλάζουν από παθητικοί σε ενεργητικοί με ζημία μηδέν και οι δύο έχουν από ένα περισσότερο παιδί και ο x γίνεται μια νέα ενεργή ρίζα. Ο βαθμός της ρίζας μειώνεται κατά 2 και ο αριθμός των ενεργών ριζών αυξάνεται κατά 1.

Σχήμα: Ελάττωση βαθμού ρίζας



(a) Root degree reduction

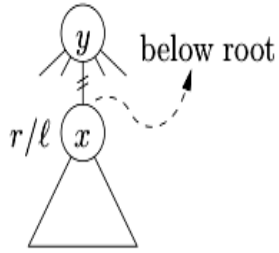
Ελάττωση ζημίας - Loss reduction : Για την ελάττωση της ζημίας έχουμε δύο διαφορετικούς μετασχηματισμούς. Την **ελάττωση ζημίας ενός κόμβου** η οποία λαμβάνει χώρα όταν ένας ενεργός κόμβος έχει ζημία ≥ 2 . Έστω y ο πατέρας του x . Σ' αυτήν την περίπτωση ο x συνδέεται με τη ρίζα και γίνεται ενεργή ρίζα με ζημία 0 και η τάξη του y μειώνεται κατά 1. Αν ο y δεν είναι ενεργή ρίζα, η ζημία του αυξάνεται κατά 1. Έτσι, συνολικά αφού η ζημία του x έχει ελαττωθεί κατά τουλάχιστον 2, η συνολική ζημία μειώνεται κατά τουλάχιστον 1.

Ο δεύτερος μετασχηματισμός είναι η **ελάττωση ζημίας δύο κόμβων**, και λαμβάνει χώρα όταν δύο ενεργοί κόμβοι x και y τάξης r έχουν ακριβώς ζημία 1. Συγκρίνουμε τα κλειδιά τους, και έχουμε π.χ. $x.key < y.key$. Έστω z ο πατέρας του y . Συνδέουμε τον y στον x , αυξάνουμε την τάξη του x και θέτουμε τη ζημία του x και του y σε 0. Ο βαθμός και η τάξη του z μειώνονται κατά 1 και αν δεν είναι ενεργή ρίζα τότε η ζημία του αυξάνεται κατά 1.

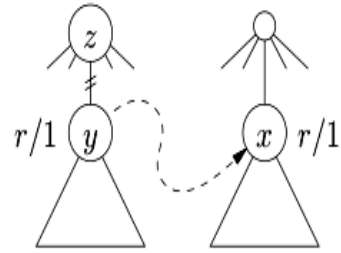
Ορισμένοι συνδυασμοί ελαττώσεων ενεργών ριζών και βαθμού ρίζας έχουν μόνο ευεργητικά αποτελέσματα. Πάντα κάνουμε τις ελαττώσεις στο βαθμό που αυτό είναι δυνατό, με οποιαδήποτε σειρά, σταματώντας μόνο όταν όλες οι ελαττώσεις έχουν πραγματοποιηθεί και δεν μπορεί να γίνει καμία άλλη.

- Μία ελάττωση ενεργών ριζών και μια ελάττωση βαθμού ρίζας μειώνουν το βαθμό της ρίζας τουλάχιστον κατά 1.
- Δύο ελαττώσεις ενεργών ριζών και μια ελάττωση βαθμού ρίζας μειώνει τον αριθμό των ενεργών ριζών κατά 1 χωρίς να αυξάνει το βαθμό της ρίζας.
- Τρεις ελαττώσεις ενεργών ριζών και δύο ελαττώσεις βαθμού ρίζας μειώνουν τόσο τον αριθμό των ενεργών ριζών, όσο και το βαθμό της ρίζας τουλάχιστον κατά 1.

Σε καμία περίπτωση ελάττωσης δεν δύναται να δημιουργηθεί κύκλος μέσα στο δένδρο, λόγω της ιδιότητας της σωρού και της διαφορετικότητας των κλειδιών.



(c) One-node loss reduction ($\ell \geq 2$)



(d) Two-node loss reduction

Σχήμα: Ελάττωση ζημίας ενός και δύο κόμβων

Μετασχηματισμός ή συνδυασμός μετασχηματισμών	Βαθμός ρίζας	Συνολική ζημία	Ενεργές ρίζες	Συγκρίσεις κλειδιών
Ελάττωση ενεργών ριζών (EP)	≤ 1	0	-1	1
Ελάττωση βαθμού ρίζας (BP)	-2	0	1	3
Ελάττωση ζημίας	≤ 1	≤ -1	≤ 1	≤ 1
- ενός κόμβου	1	≤ -1	1	0
- δύο κόμβων	0	-1	0	1
EP + BP	≤ -1	0	0	4
2x (EP) + BP	≤ 0	0	-1	5
3x(EP) + 2x(BP)	≤ -1	0	-1	9

Πίνακας: Τα αποτελέσματα των διάφορων μετασχηματισμών

	Βαθμός ρίζας	Συνολική ζημία	Ενεργές ρίζες
Ελάττωση κλειδιού	$\leq 1+1+6-8$	$\leq 1-1+0+0$	$\leq 1+1-6+4$
Συγχώνευση	$\leq 1+0+1-2$	$\leq 0+0+0+0$	$\leq 0+0-1+1$
Διαγραφή ελαχίστου	$\leq (2\log n + 12 + 4) + 1$	$\leq 0-1$	$\leq R+1$

Πίνακας: Οι αλλαγές που επιφέρουν οι διάφορες διαδικασίες της σωρού

Στον παραπάνω πίνακα βλέπουμε τις διαφορές που επιφέρουν οι διαδικασίες decrease-key, meld και delete-min της σωρού στον αριθμό των ενεργών ριζών, στη συνολική ζημία και στο βαθμό της ρίζας. Κάθε πεδίο είναι ένα άθροισμα τεσσάρων όρων που αφορούν στην αλλαγή που επιφέρουν οι μετασχηματισμοί που συμβαίνουν σε κάθε διεργασία, οι ελαττώσεις ζημίας, ενεργών ριζών και βαθμού ρίζας αντίστοιχα. Για παράδειγμα, για τη συγχώνευση ο βαθμός της ρίζας είναι η αλλαγή που γίνεται στην παλιά ρίζα που γίνεται η νέα ρίζα, ενώ η συνολική ζημία και ο αριθμός των ενεργών ριζών συγκρίνεται με τη σωρό που δε

γίνεται παθητική.

Κατά τη διάρκεια τόσο της ελάττωσης-κλειδιού όσο και της συγχώνευσης όλα τα αθροίσματα είναι 0 και το R αυξάνεται κατά τη συγχώνευση και έτσι οι σταθερές I2 και I3 διατηρούνται αληθείς. Η διαγραφή ελαχίστου ελαττώνει το μέγεθος της σωρού κατά 1, μειώνοντας το R κατά το πολύ 1, αν οι κόμβοι είναι περισσότεροι ή ίσοι με 4 πριν τη διαδικασία. Η μείωση στη ζημία επαληθεύει το I3 μετά από τη διαγραφή ελαχίστου.

Μετά το σταμάτημα των ελαττώσεων ενεργών ριζών και βαθμού ρίζας έχουμε το μέτρο: $2 \times (\text{βαθμός ρίζας}) + 3 \times (\text{αριθμός ενεργών ριζών})$ μειωμένο κατά τουλάχιστον 1. Η αρχική του τιμή είναι $O(\log n)$ κάτι που συνεπάγεται αυτόματα τα χρονικά όρια της σωρού.

3.1.4 Υλοποίηση

Η ανάπτυξη του κώδικα έχει σε περιβάλλον Ubuntu, με χρήση της γλώσσας C. Κατά τη συγγραφή ήταν πολλές οι πηγές μέσω διαδικτύου οι οποίες βοήθησαν στο να ξεπεραστούν αρκετά προβλήματα υλοποίησης. Κυρίως όμως χρησιμοποιήθηκε κώδικας από την ιστοσελίδα <http://code.google.com> και ειδικά από το

<http://code.google.com/p/priority-queue-testing/source/browse/trunk/queues/>

Ξεκινώντας περιγράφουμε τις δομές που χρειάζονται για την υλοποίηση της σωρού.

Αρχικά, υπάρχει το **node record**:

```
/**
```

```
* Το vasiko domiko stoixeio ths heap. Kathe komvos apothikeuei ena zeugari antikeimeno-kleidi (item-key)
```

```
* kai perietetai se mia dipla syndedemenh, kyklikh lista me ta aderfia tou.
```

```
* Epipleon, kathe komvos exei ena deikth pros ton patera tou kai to aristerotero paidi tou.
```

```
* O komvos exei deiktes pros ton epomeno kai ton prohgoumeno komvo sthn heap Q kathws epishs
```

```
* sta records ths rank kai to active (an einai energos). Apothikeuoume ton teleutaio typo komvou
```

```
* etsi wste kathe allagh typou na ginetai antilhpth kai apofeugetai etsi axreiasth anadomhsh
```

```
* Telos, oi active nodes mporei na exoun thetikh loss kai ena deikth pros enan komvo sthn fix list.
```

```
*/
```

```
struct strict_fibonacci_node_t {
```

```
    item_type item;
```

```
    key_type key;
```

```
    struct strict_fibonacci_node_t *parent;
```

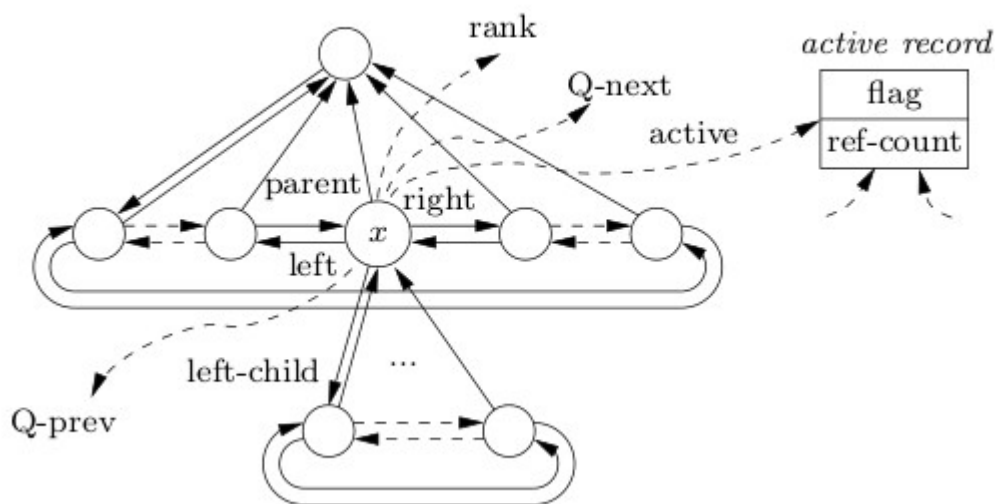
```

struct strict_fibonacci_node_t *left;
struct strict_fibonacci_node_t *right;
struct strict_fibonacci_node_t *left_child;

struct strict_fibonacci_node_t *q_prev;
struct strict_fibonacci_node_t *q_next;

uint32_t type;
active_record *active;
rank_record *rank;
fix_node *fix;
uint32_t loss;
}

```



Σχήμα: Η αναπαράσταση του node record και active record. Φαίνονται επίσης οι λίστες Q και αυτή των αδερφών.

Για το σημάδεμα των κόμβων σε παθητικούς και ενεργούς έχουμε ένα struct τύπου active record το οποίο περιέχει μια σημαία για την αλλαγή όλων των κόμβων που “δείχνουν” στο record αυτό σε χρόνο $O(1)$.

```

struct active_record_t {

    uint32_t flag;        //!< 1 if active, 0 allws

    uint32_t ref_count;   //!< Ο αριθμος twν komvwn pou deixnoun s'auto to record.
                        //!< If 0, free record.
}

```

Το struct της σωρού είναι επίσης βασικό στοιχείο και η περιγραφή του έχει ως εξής:

```

struct strict_fibonacci_heap_t {

```

```

mem_map *map;
uint32_t size;           //to megethos ths heap

strict_fibonacci_node *root; //deikths sth riza
strict_fibonacci_node *q_head; //deikths ston prwto komvo ths listas Q

active_record *active;           //deikths sto active record ths heap
rank_record *rank_list;         //deikths sto deksiotero record ths rank list
fix_node *fix_list[2];          //deikths sto deksiotero record ths fix list kai
                                //sto deksiotero record me thetikh zhmia

fix_node *garbage_fix;
}

```

Σημαντικές δομές της σωρού και κύριες καινοτομίες της, είναι οι rank και fix lists. Μια τάξη λέγεται active-root transformable αν υπάρχουν τουλάχιστον δύο ενεργές ρίζες αυτής της τάξης, ενώ λέγεται loss-transformable αν η συνολική ζημία των κόμβων αυτής της τάξης είναι τουλάχιστον 2.

Για κάθε τάξη κρατάμε έναν κόμβο και όλοι αυτοί οι κόμβοι ανήκουν σε μια λίστα (rank list). Ο δεξιότερος κόμβος αντιστοιχεί στη μηδενική τάξη και ο κόμβος που αντιστοιχεί στην k-οστή τάξη είναι το αριστερό αδέρφι αυτού που αντιστοιχεί στην k-1-οστή.

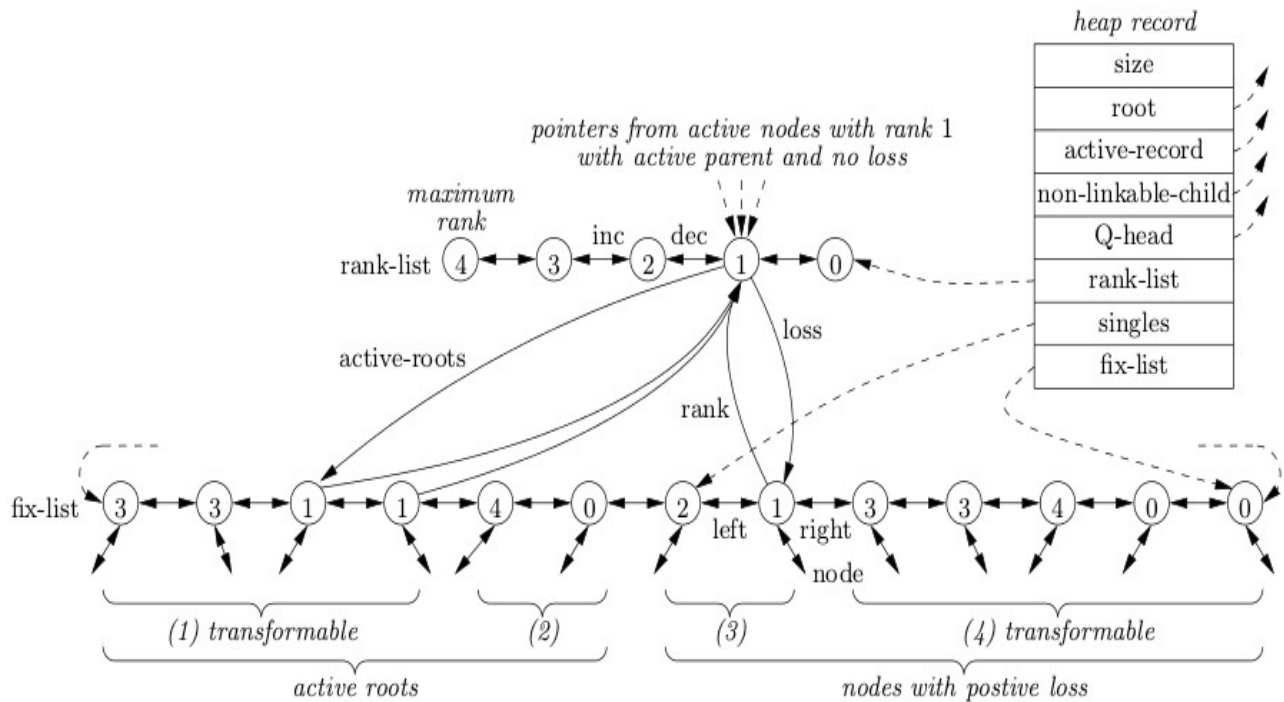
Όλοι οι ενεργοί κόμβοι που μπορούν μελλοντικά να συμμετάσχουν σε ένα μετασχηματισμό (ελαττώσεις), δηλαδή ενεργές ρίζες και ενεργοί κόμβοι με θετική ζημία, κρατούνται σε μια λίστα που ονομάζεται fix list. Κάθε κόμβος τάξης k στη λίστα αυτή δείχνει σε ένα κόμβο αυτής της τάξης στην rank list. Η fix list χωρίζεται σε τέσσερα κομμάτια, από τα αριστερά προς τα δεξιά. Στα κομμάτια 1-2 περιέχονται οι ενεργές ρίζες και στα 3-4 οι ενεργοί κόμβοι θετικής ζημίας. Πιο συγκεκριμένα:

- Το κομμάτι 1 περιέχει τις ενεργές ρίζες των active root transformable τάξεων, εκ των οποίων ένας περιέχει δείκτη από το πεδίο της τάξης στην rank list,
- Το κομμάτι 2 περιέχει τις εναπομείνασες ενεργές ρίζες,
- Το κομμάτι 3 περιέχει ενεργούς κόμβους με ζημία 1 και τάξη η οποία δεν είναι loss transformable και οι κόμβοι αυτοί περιέχουν δείκτη από την αντίστοιχη τάξη στην rank list και τέλος,
- Το κομμάτι 4 περιέχει όλους τους ενεργούς κόμβους τάξεων που είναι loss-transformable. Ένας κόμβος περιέχει δείκτη από το αντίστοιχο πεδίο της rank list.

Μετά τα παραπάνω μπορούμε να συμπεράνουμε ότι μπορούμε πάντα να κάνουμε μια ελάττωση ενεργών ριζών όσο το κομμάτι 1 της fix-list δεν είναι άδειο και επίσης μπορούμε να πραγματοποιούμε ελάττωση ζημίας εφόσον το κομμάτι 4 περιέχει στοιχεία. Στη δομή της σωρού διατηρούμε δείκτες στα όρια των κομματιών 2 και 3.

Για την ελάττωση ζημίας πηγαίνουμε στο δεξί άκρο της fix list και κάνουμε ελάττωση ενός κόμβου (αν έχουμε πρόσβαση σε κόμβο θετικής ζημίας) ή δύο

κόμβων (αν οι δύο δεξιότεροι κόμβοι της λίστας έχουν από ζημία 1). Μετά από μια ελάττωση ζημίας σε μια τάξη k , ίσως πρέπει να μεταφέρουμε έναν κόμβο τάξης k στο κομμάτι 3. Όποτε, η ζημία ενός κόμβου, με τάξη που είναι loss-transformable, αυξάνεται, το εισάγουμε στο κατάλληλο γκρουπ του κομματιού 4. Αν η τάξη του δεν είναι loss transformable το εισάγουμε στο κομμάτι 3, εκτός κι αν υπάρχει ένας κόμβος με ίδια τάξη, περίπτωση στην οποία μεταφέρουμε και τους δύο κόμβους στο κομμάτι 4 στο δεξί άκρο της fix list.



Σχήμα: Το record της σωρού και η αναπαράσταση των rank και fix list

```
static int reduce_loss( strict_fibonacci_heap *queue ) {

    int reduction = 2;
    fix_node *head = queue->fix_list[STRICT_FIX_LOSS];
    if( head == NULL )
        return 0;

    rank_record *rank = head->rank;
    if( head == rank->tail[STRICT_FIX_LOSS] && head->node->loss < 2 )
        return 0;

    fix_node *single;
    fix_node *next = head->right;
    strict_fibonacci_node *child, *parent, *old_parent;
    if( head->node->loss > 1 ) {
        reduction = 1;
        single = head;
    }
    else if( next->node->loss > 1 ) {
```

```

    reduction = 1;
    single = next;
}

if( reduction == 1 ) {
    child = single->node;
    parent = queue->root;
    old_parent = child->parent;

    if( child != parent )
        link( queue, parent, child );

    decrease_loss( queue, child );

    if( !is_active( queue, parent ) )
        convert_active_to_root( queue, child );
}
else {

    choose_order_pair( head->node, next->node, &parent, &child );
    old_parent = child->parent;

    link( queue, parent, child );

    decrease_loss( queue, child );
    decrease_loss( queue, parent );
    increase_rank( queue, parent );
}

if( old_parent != NULL && is_active( queue, old_parent ) ) {

    decrease_rank( queue, old_parent );
    if( is_active( queue, old_parent ) &&
        old_parent->type != STRICT_TYPE_ROOT )
        increase_loss( queue, old_parent );
}

return 1;
}

```

Για την διενέργεια ελάττωσης ενεργών ριζών, πηγαίνουμε στο αριστερό άκρο της fix list και συνδέουμε τις δύο αριστερότερες ενεργές ρίζες, εφόσον έχουν την ίδια τάξη (διαφορετικά το κομμάτι 1 θα είναι άδειο). Αν μετά την ελάττωση οι δύο αριστερότεροι κόμβοι της λίστας είναι μη ενεργές ρίζες ίσου βαθμού, μεταφέρουμε τον αριστερότερο στο κομμάτι 2.

Όταν ένας ενεργός κόμβος τάξης k γίνεται ενεργή ρίζα και δεν υπάρχει άλλη ενεργή ρίζα ίδιας τάξης, τον βάζουμε στο κομμάτι 2, αλλιώς τον εισάγουμε στη συνομοταξία των ενεργών ριζών της ίδιας τάξης στο κομμάτι 1, εκτός κι αν

υπάρχει μόνο μια ενεργή ρίζα αυτής της τάξης (και βρίσκεται στο κομμάτι 2), στην οποία περίπτωση τους μεταφέρουμε στο κομμάτι 1. Όταν η τάξη ενός κόμβου που ανήκει στη fix list αλλάξει, μπορούμε εύκολα να κάνουμε τις κατάλληλες μετατροπές στη λίστα, ώστε όλα τα κομμάτια να είναι όπως περιγράψαμε παραπάνω.

```
static int reduce_active_roots( strict_fibonacci_heap *queue ) {

    fix_node *head = queue->fix_list[STRICT_FIX_ROOT];
    if( head == NULL )
        return 0;

    rank_record *rank = head->rank;
    if( head == rank->tail[STRICT_FIX_ROOT] )
        return 0;

    fix_node *next = head->right;
    strict_fibonacci_node *parent, *child;
    choose_order_pair( head->node, next->node, &parent, &child );

    link( queue, parent, child );
    convert_root_to_active( queue, child );
    increase_rank( queue, parent );

    strict_fibonacci_node *extra = parent->left_child->left;
    if( !is_active( queue, extra ) )
        link( queue, queue->root, extra );

    return 1;
}

/**
 * Meiwsh tou vathmou ths rizas syndeontas tous treis deksioteraus pathitikous
 * komvous.
 *
 * @param queue H heap sthn opoia tha ginei i diadikasia
 */
static int reduce_root_degree( strict_fibonacci_heap *queue ) {

    if( queue->root == NULL || queue->root->left_child == NULL )
        return 0;

    strict_fibonacci_node *x = queue->root->left_child->left;
    if( x == queue->root->left_child || is_active( queue, x ) )
        return 0;

    strict_fibonacci_node *y = x->left;
    if( y == queue->root->left_child || is_active( queue, y ) )
```

```
return 0;
```

```
strict_fibonacci_node *z = y->left;  
if( z == queue->root->left_child || is_active( queue, z ) )  
    return 0;
```

```
strict_fibonacci_node *grand, *parent, *child;  
choose_order_triple( x, y, z, &grand, &parent, &child );
```

```
convert_passive_to_active( queue, parent );  
convert_passive_to_active( queue, grand );  
increase_rank( queue, grand );  
if( !is_active( queue, queue->root ) )  
    convert_active_to_root( queue, grand );
```

```
link( queue, parent, child );  
link( queue, grand, parent );
```

```
return 1;  
}
```

Τα struct και οι διάφορες ενέργειες των rank και fix list

```
struct rank_record_t {  
  
    uint32_t rank;  
  
    struct rank_record_t *inc;    // epomeno rank an yparxei  
  
    struct rank_record_t *dec;    // prohgomeno rank an yparxei  
  
    int transformable[2];        // flags gia to teleutaio transformability status  
  
    struct fix_node_t *head[2];    // deiktes stous nodes ths fix ths sygkekrimenhs  
                                   //rank  
    struct fix_node_t *tail[2];  
  
    uint32_t ref_count;           // o ari8mos twn nodes pou deixnoun sto  
                                   // sygkekrimeno rank record, free record if 0  
}  
  
struct fix_node_t {  
  
    struct strict_fibonacci_node_t *node;    //deikths ston komvo ths heap  
    struct fix_node_t *left;                //deiktes sto epomeno kai  
    prohgomeno record ths fix list
```

```

    struct fix_node_t *right;
    rank_record *rank;                                     //deikths sto record ths rank list pou
                                                            //antistoixei sthn taksh tou komvou
}

/**
 * Auksanei thn taksh tou kathorismenou komvou kata 1. Kanei allocate ena neo
rank record
 * an xreiastei. Katallhlh anadomhsh ths fix list.
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros allagh
 */
static void increase_rank( strict_fibonacci_heap *queue, strict_fibonacci_node
*node ) {

    rank_record *new_rank = node->rank->inc;
    uint32_t target_rank = node->rank->rank + 1;
    if( new_rank->rank != target_rank )
        new_rank = create_rank_record( queue, target_rank, new_rank );

    switch_node_rank( queue, node, new_rank );
}

/**
 * Meiwnei thn taksh tou kathorismenou komvou kata 1. Kanei allocate ena neo
rank record
 * an xreiastei. Katallhlh anadomhsh ths fix list.
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros allagh
 */
static void decrease_rank( strict_fibonacci_heap *queue, strict_fibonacci_node
*node ) {

    rank_record *new_rank = node->rank->dec;
    uint32_t target_rank = node->rank->rank - 1;
    if( new_rank->rank != target_rank )
        new_rank = create_rank_record( queue, target_rank, node->rank );

    switch_node_rank( queue, node, new_rank );
}

```



```

/**
 * Auksanei thn zhmia tou kathorismenou komvou kata 1. Katallhlh anadomhsh
ths fix list.
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros allagh
 */
static void increase_loss( strict_fibonacci_heap *queue, strict_fibonacci_node
*node ) {

    node->loss++;
    if( node->loss == 1 )
        convert_active_to_loss( queue, node );
}

/**
 * Meiwnei thn zhmia tou kathorismenou komvou se 0. Katallhlh anadomhsh ths
fix list.
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros allagh
 */
static void decrease_loss( strict_fibonacci_heap *queue, strict_fibonacci_node
*node ) {
    node->loss = 0;
    convert_loss_to_active( queue, node );
}

/**
 * Allazei thn rank enos komvou kai kanei katallhlh anadomhsh ths fix list.
 * Apeleutherwnei to deikth pros thn palia rank, prokalwntas isws garbage
collection.
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros allagh
 * @param new_rank Nea rank gia ton komvo
 */
static void switch_node_rank( strict_fibonacci_heap *queue, strict_fibonacci_node
*node, rank_record *new_rank ) {

    int type = ( node->type == STRICT_TYPE_ROOT ) ? STRICT_FIX_ROOT :
STRICT_FIX_LOSS;

    fix_node *fix = node->fix;

```

```

if( fix != NULL )
    remove_fix_node( queue, fix, type );

release_rank_record( queue, node );
node->rank = new_rank;
new_rank->ref_count++;

if( fix != NULL ) {
    fix->rank = new_rank;
    insert_fix_node( queue, fix, type );
}
}

/**
 * Enthetei ena node typou fix list sthn kataallhli lista.
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param fix   Fix node pros enthesh
 * @param type  Poia lista tha allaxthei
 */
static void insert_fix_node( strict_fibonacci_heap *queue, fix_node *fix, int type )
{
    rank_record *rank = fix->rank;

    if( rank->head[type] == NULL ) {

        rank->head[type] = fix;
        rank->tail[type] = fix;

        if( queue->fix_list[type] == NULL ) {

            fix->right = fix;
            fix->left = fix;
            queue->fix_list[type] = fix;
            return;
        }
        else {
            fix->right = queue->fix_list[type];
            fix->left = fix->right->left;
            fix->right->left = fix;
            fix->left->right = fix;
        }
    }
    else {
        fix->right = rank->head[type];
        fix->left = fix->right->left;
        fix->right->left = fix;
    }
}

```

```

    fix->left->right = fix;

    if( queue->fix_list[type] == rank->head[type] )
        queue->fix_list[type] = fix;
    rank->head[type] = fix;
}

check_rank( queue, rank, type );
}

/**
 * Διαγραφη ενος καθορισμένου node τύπου fix list απο την καταλληλή lista.
 *
 * @param queue Heap στην οποία θα γίνει η διαδικασία
 * @param fix   Fix node προς διαγραφη
 * @param type  Ποια lista θα αλλαχθεί
 */
static void remove_fix_node( strict_fibonacci_heap *queue, fix_node *fix, int
type ) {

    rank_record *rank = fix->rank;

    if( queue->fix_list[type] == fix ) {

        if( fix->right == fix )
            queue->fix_list[type] = NULL;
        else
            queue->fix_list[type] = fix->right;
    }

    if( rank->head[type] == fix ) {
        if( rank->tail[type] == fix ) {
            rank->head[type] = NULL;
            rank->tail[type] = NULL;
        }
        else
            rank->head[type] = fix->right;
    }
    else if( rank->tail[type] == fix )
        rank->tail[type] = fix->left;

    fix_node *next = fix->right;
    fix_node *prev = fix->left;

    next->left = prev;
    prev->right = next;

    check_rank( queue, rank, type );
}

```

```
}
```

```
/**
```

```
 * Elegxos mias kathorismenhs rank gia na doume an xreiazetai na prowthithei h'
na opisthoxwrhsei sth fix list.
```

```
 *
```

```
 * @param queue H heap sthn opoia tha ginei h diadikasia
```

```
 * @param rank Rank pros elegxo
```

```
 * @param type Poia lista tha allaxthei
```

```
 */
```

```
static void check_rank( strict_fibonacci_heap *queue, rank_record *rank, int type )
{
```

```
    if( rank->head[type] == NULL )
        return;
```

```
    int status = ( rank->head[type] != rank->tail[type] ||
        rank->head[type]->node->loss > 1 );
```

```
    if( rank->transformable[type] && !status )
        move_rank( queue, rank, type, STRICT_DIR_DEMOTE );
    else if( !rank->transformable[type] && status )
        move_rank( queue, rank, type, STRICT_DIR_PROMOTE );
```

```
}
```

```
/**
```

```
 * Metakinsh mia kathorismenhs rank mesa sthn kathorismenh fix list
```

```
 *
```

```
 * @param queue H heap sthn opoia tha ginei h diadikasia
```

```
 * @param rank Rank pros metakinsh
```

```
 * @param type Poia lista tha allaxthei
```

```
 * @param direction STRICT_DIR_PROMOTE or STRICT_DIR_DEMOTE
```

```
 */
```

```
static void move_rank( strict_fibonacci_heap *queue, rank_record *rank, int type,
int direction ) {
```

```
    fix_node *head = rank->head[type];
    fix_node *tail = rank->tail[type];
    fix_node *pred = head->left;
    fix_node *succ = tail->right;
```

```
    rank->transformable[type] = direction;
```

```
    if( pred == tail )
        return;
```

```

if( queue->fix_list[type] == head && direction == STRICT_DIR_PROMOTE )
    return;

if( queue->fix_list[type] == succ && direction == STRICT_DIR_DEMOTE )
    return;

if( queue->fix_list[type] == succ && direction == STRICT_DIR_PROMOTE ) {
    queue->fix_list[type] = head;
    return;
}

if( queue->fix_list[type] == head )
    queue->fix_list[type] = succ;

pred->right = succ;
succ->left = pred;

succ = queue->fix_list[type];
pred = succ->left;

succ->left = tail;
pred->right = head;
head->left = pred;
tail->right = succ;

if( direction == STRICT_DIR_PROMOTE )
    queue->fix_list[type] = rank->head[type];
}

```

Οι βασικές διεργασίες της απόλυτης σωρού Fibonacci, οι οποίες έχουν αναλυθεί και σε προηγούμενο κεφάλαιο, μαζί με κάποιες μικρότερης σημασίας συναρτήσεις:

```

/**
 * Dhmiourgei mia nea, adeia heap.
 *
 * @param map   Memory map gia xrhsh sthn allocation tou node
 * @return      Deikth pros th nea heap
 */
strict_fibonacci_heap* makeHeap( mem_map *map ) {

    strict_fibonacci_heap *queue = (strict_fibonacci_heap*) calloc( 1,
sizeof( strict_fibonacci_heap ) );
    queue->map = map;

    return queue;
}

```

```
}
```

```
/**
 * Kanei Free olh th mnhmh pou exei xrhsimopoihthei apo thn heap.
 *
 * @param queue H heap pros katastrofh
 */
void destroyHeap( strict_fibonacci_heap *queue ) {

    clearHeap( queue );
    free( queue );
}
```

```
/**
 * Diagrafh olwn twv komvwn pou yparxoun se mia heap.
 *
 * @param queue H heap pros ekatharish
 */
void clearHeap( strict_fibonacci_heap *queue ) {

    mm_clear( queue->map );
    queue->size = 0;

    queue->root = NULL;
    queue->q_head = NULL;

    queue->active = NULL;
    queue->rank_list = NULL;
    queue->fix_list[0] = NULL;
    queue->fix_list[1] = NULL;
}
```

```
/**
 * Epistrefei to kleidi tou komvou pou zhth8hke.
 *
 * @param queue H heap sthn opoia anhkei o komvos
 * @param node Komvos pou zhteitai
 * @return to key tou node
 */
key_type get_key( strict_fibonacci_heap *queue, strict_fibonacci_node *node ) {

    return node->key;
}
```

```

/**
 * Epistrefei to antikeimeno tou komvou pou zhth8hke.
 *
 * @param queue H heap sthn opoia anhkei o komvos
 * @param node Komvos pou zhteitai
 * @return to item tou node
 */
item_type* get_item( strict_fibonacci_heap *queue, strict_fibonacci_node *node )
{
    return (item_type*) &(node->item);
}

/**
 * Epistrefei to trexon megethos ths ouras.
 *
 * @param queue H heap pou zhteitai
 * @return To megethos ths ouras
 */
uint32_t get_size( strict_fibonacci_heap *queue ) {
    return queue->size;
}

/**
 * Epistrefei to elaxisto antikeimeno ths ouras xwris na kanei allages.
 *
 * @param queue H heap pou tha psaksoume
 * @return O node me to minimum key
 */
strict_fibonacci_node* find_min( strict_fibonacci_heap *queue ) {
    if ( isEmpty( queue ) )
        return NULL;
    return queue->root;
}

```

```

/**
 * Pairnei ws parametro ena zeugos antikeimenou-kleidiau gia na to eisagei sthn
 heap kai dhmiourgei
 * ena neo node. Vazei to node san nea riza.
 *
 * @param queue H heap sthn opoia tha ginei h enthesh
 * @param item To antikeimeno pou tha enthethei
 * @param key To kleidi pou tha xrhsimopoihthei gia thn proteraiothta tou
 komvou
 * @return Deikth pros ton komvo
 */
strict_fibonacci_node* insert( strict_fibonacci_heap *queue, item_type item,
key_type key ) {

    strict_fibonacci_node* wrapper = alloc_node( queue->map,
    STRICT_NODE_FIB );
    ITEM_ASSIGN( wrapper->item, item ); //anathetei to item ston komvo
    wrapper->key = key;
    wrapper->right = wrapper;
    wrapper->left = wrapper;
    wrapper->q_next = wrapper;
    wrapper->q_prev = wrapper;

    strict_fibonacci_node *parent, *child;
    if( queue->root == NULL )
        queue->root = wrapper;
    else {
        choose_order_pair( wrapper, queue->root, &parent, &child ); //Vazoume sthn
        // katallhli thesh tous komvous analoga me to key
        link( queue, parent, child ); //Enwnoume to neo komvo me thn yparxousa
        //heap
        queue->root = parent; //Root ginetai o parent
        enqueue_node( queue, child ); //Vazoume to child sth heap

        post_meld_reduction( queue );
    }

    queue->size++;
    garbage_collection( queue );

    return wrapper;
}

```



```

/**
 * Afairei ena tyxaio antikeimeno apo ti lista. Apaitei h thesh tou antistoixou
 komvou
 * na einai gnwsth. Meta thn afaresh tou komvou, kanei ta paidia tou nees rizes
 ths ouras
 * Anadromika kanei enwnei ta dendra idias rank etsi wste na mhn yparxoun dyo
 dendra idias rank.
 *
 * @param queue H heap sthn opoia anhkei o node
 * @param node Deikths pros ton node pou antistoixei sto antikeimeno pros
 diagrafh
 * @return To kleidi tou antikeimenou pou diegrafh
 */
key_type delete_node( strict_fibonacci_heap *queue, strict_fibonacci_node
 *node ) {

    key_type key = node->key;

    decrease_key( queue, node, 0 );
    delete_min( queue );

    return key;
}

/**
 * Auth h diadikasia tha epanadomhsei thn heap gia na diorthwthoun tyxon
 * domikes paravaseis. Apokoptei to node apo ton patera tou kai to kanei nea riza
 *
 * @param queue H heap sthn opoia anhkei o komvos
 * @param node O node pros allagh
 * @param new_key To neo key pou tha xrhsimopoihthei
 */
void decrease_key( strict_fibonacci_heap *queue, strict_fibonacci_node *node,
 key_type new_key ) {

    strict_fibonacci_node *old_parent = node->parent;

    node->key = new_key;

    if( old_parent == NULL || node->key > old_parent->key ) //An einai h riza h' o
 pateras exei mikrotero kleidi teleiwsame
        return;

    strict_fibonacci_node *parent, *child;

```

```
choose_order_pair( node, queue->root, &parent, &child ); //Dhmiourgia one-
node heap
```

```
link( queue, parent, child ); //synenwsh
queue->root = parent;
queue->root->parent = NULL;
```

```
if( parent == node ) {
    dequeue_node( queue, parent );
    enqueue_node( queue, child );
}
```

```
if( is_active( queue, node ) ) {
    if( is_active( queue, old_parent ) )
        decrease_rank( queue, old_parent ); //meiwsh ths rank an
                                              //htan energos alla oxi active root
    if( node->type != STRICT_TYPE_ROOT )
        convert_active_to_root( queue, node ); //ton metatrepoume se active
                                              //root
}
```

```
if( is_active( queue, old_parent ) && old_parent->type != STRICT_TYPE_ROOT )
    increase_loss( queue, old_parent ); //aukshsh tou loss tou patera
```

```
post_decrease_key_reduction( queue ); //aparaithtes reductions
garbage_collection( queue );
}
```

```
/**
```

```
* Afairei to elaxisto antikeimeno apo thn heap kai to epistrefei.
* @ref <delete> Eksartatai apo thn delete gia th diagrafh tou komvou.
*
* @param queue H heap pou tha psaksoume
* @return To minimum key, pou antistoixe sto antikeimeno pou diegrafh
*/
```

```
key_type delete_min( strict_fibonacci_heap *queue ) {
```

```
    if( isEmpty( queue ) )
        return 0;
```

```
    key_type key = queue->root->key;
    strict_fibonacci_node *current, *new_root, *old_root;
    int i, j;
```

```
    old_root = queue->root;
```

```
    if( old_root->left_child == NULL ) {
        old_root = queue->root;
        if( is_active( queue, old_root ) ) // An o komvos einai energhtikos ton
```

```

//kanoume pathitiko
convert_to_passive( queue, old_root );
queue->root = NULL;
}
else {
    new_root = select_new_root( queue );    //Vriskoume to neo elaxisto
    remove_from_siblings( queue, new_root ); //Vgazoume th nea riza apo th
                                              //lista

    dequeue_node( queue, new_root );
    queue->root = new_root;

    if( is_active( queue, new_root ) )
        convert_to_passive( queue, new_root );
    if( is_active( queue, old_root ) )
        convert_to_passive( queue, old_root );

    while( old_root->left_child != NULL )    //Ta paidia ths palias rizas
                                              //ginontai paidia ths neas
        link( queue, new_root, old_root->left_child );

    for( i = 0; i < 2; i++ ) {

        current = consume_node( queue );    //Kanoume ton prwto komvo ths
                                              //listas, teleutaio

        if( current != NULL ) {
            for( j = 0; j < 2; j++ ) {
                //Enwnoume ta dyo deksiotera
                //paidia tou komvou pou metakinhsame
                //ston x, efson einai pathitika
                if( current->left_child != NULL && !is_active( queue,
                    current->left_child->left ) )
                    link( queue, new_root, current->left_child->left );
                else
                    break;
            }
        }
    }
}

pq_free_node( queue->map, STRICT_NODE_FIB, old_root );

post_delete_min_reduction( queue );    //Oi aparaithtes reductions
garbage_collection( queue );

queue->size--;

return key;
}

```

```

/**
 * Synenwnei dyo diaforetikou megethous heaps.
 * Epistrefei enan pointer sthn prokypousa heap.
 *
 * @param a H 1h heap
 * @param b H 2h heap
 * @return H prokypousa synenwmenh heap
 */
strict_fibonacci_heap* meld( strict_fibonacci_heap *a, strict_fibonacci_heap *b ) {

    strict_fibonacci_heap *new_heap = makeHeap( a->map );           //Ftiaxnoume
                                                                    //th nea heap

    strict_fibonacci_heap *big, *small;

    strict_fibonacci_node *big_head, *big_tail, *small_head, *small_tail;
    strict_fibonacci_node *parent, *child;

    // Dialegoume poia heap tha krathsoume (th megalyterh)
    if( a->size < b->size ) {
        big = b;
        small = a;
    }
    else {
        big = a;
        small = b;
    }

    // thetουμε ta pedia ths neas heap (syndyasmos tw n paliwn)
    new_heap->size = big->size + small->size;
    new_heap->q_head = big->q_head;
    new_heap->active = big->active;
    new_heap->rank_list = big->rank_list;
    new_heap->fix_list[0] = big->fix_list[0];
    new_heap->fix_list[1] = big->fix_list[1];

    if( small->active != NULL )           //Kanoume pathitikous olous tous komvous
                                        //ths mikrhs heaps

        small->active->flag = 0;

    // sygxwneush tw n ourwn Qx kai Qy
    big_head = big->q_head;
    big_tail = big_head->q_prev;
    small_head = small->q_head;
    small_tail = small_head->q_prev;

```

```

big_head->q_prev = small_tail;
small_tail->q_next = big_head;
small_head->q_prev = big_tail;
big_tail->q_next = small_head;

// Kanoume th mikroterh paidi ths rizas ths megalyterhs
choose_order_pair( big->root, small->root, &parent, &child );
link( new_heap, parent, child );
new_heap->root = parent;
enqueue_node( new_heap, child );

release_to_garbage_collector( new_heap, small );
free( small );
free( big );
garbage_collection( new_heap );

return new_heap;
}

/**
 * Kathorizei an h oura einai adeia h' oxi.
 *
 * @param queue H heap pou theloume
 * @return True an h heap einai adeia, alliws false
 */
bool isEmpty( strict_fibonacci_heap *queue ) {

    return ( queue->size == 0 );
}

```

Μετά από τις βασικές συναρτήσεις τις σωρού πολλές φορές είναι απαραίτητες διορθωτικές ελαττώσεις ώστε να ικανοποιούνται οι σταθερές της υλοποίησης. Αυτές γίνονται μετά από συναρτήσεις όπως, η `delete_min`, η `meld` και η `decrease-key`. Οι συναρτήσεις αυτές, καλούν τις συναρτήσεις που θα πραγματοποιήσουν τις ελαττώσεις και έχουν περιγραφεί παραπάνω (π.χ. `reduce_loss`).

```

/**
 * Mia akolouthia apo reductions pou apaitountai meta apo synenwseis kai
 * entheseis.
 * Epixeirei mia ellatwsh ths zhmiias (loss reduction) kai sth synexeia elattwsh
 * energwn rizwn kai
 * tou vathmou ths rizas sto dynato vathmo
 *
 * @param queue H swros sthn opoia tha ginei h diadikasia elattwshs
 */
static void post_meld_reduction( strict_fibonacci_heap *queue ) {

```

```

int count_root = 0;
int count_degree = 0;

reduce_loss( queue );

while( count_root < 1 && count_degree < 1 ) {

    if( count_root < 1 && reduce_active_roots( queue ) )
        count_root++;
    else if( count_degree < 1 && reduce_root_degree( queue ) )
        count_degree++;
    else
        break;
}
}

/**
 * Mia akolouthia ellatwsewn pou prepei na ginoun meta apo diagrafes.
 * Meiwnei tis energies rizas kai to vathmo ths rizas sto vathmo pou auto einai
dynato.
 *
 * @param queue H heap sthn opoia tha ginei i diadikasia
 */
static void post_delete_min_reduction( strict_fibonacci_heap *queue ) {

    while( 1 ) {

        if( reduce_active_roots( queue ) )
            continue;
        else if( reduce_root_degree( queue ) )
            continue;
        else
            break;
    }
}

/**
 * Mia akolouthia ellatwsewn pou prepei na ginoun meta apo mia ellatwsh
kleidiou. Attempts
 * Prospathei na meiwsei thn zhmia, meta kanei 6 ellatwseis energwn rizwn kai 4
ellatwseis
 * vathmou rizas sto vathmo pou auto einai dynato.
 *
 * @param queue H heap sthn opoia tha ginei i diadikasia
 */
static void post_decrease_key_reduction( strict_fibonacci_heap *queue ) {

    int count_root = 0;

```

```

int count_degree = 0;

reduce_loss( queue );

while( count_root < 6 && count_degree < 4 ) {

    if( count_root < 6 && reduce_active_roots( queue ) )
        count_root++;
    else if( count_degree < 4 && reduce_root_degree( queue ) )
        count_degree++;
    else
        break;
}
}

```

Στη συνέχεια παραθέτουμε βοηθητικές συναρτήσεις που χειρίζονται τους κόμβους (τους συνδέουν, διαλέγουν νέα ρίζα, ελέγχουν αν είναι ενεργοί κτλ.) καθώς και συναρτήσεις που χειρίζονται τη λίστα Q της σωρού.

```

/**
 * Kathorizei an enas komvos einai energos. An enas pathitikos komvos htan
 * profata energos
 * apeleutherwnoume olous tous deiktes pros auton ton komvo
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros elegxo
 * @return 1 if active, 0 alliws
 */
static inline int is_active( strict_fibonacci_heap *queue, strict_fibonacci_node
*node ) {

    if( node->active == NULL )
        return 0;

    if( !node->active->flag ) {
        release_active_record( queue, node ); //Apeleutherwnei ton komvo
        release_rank_record( queue, node ); //kai to deikth apo to rank record
        if( node->fix != NULL )
            node->fix = NULL;
        node->type = STRICT_TYPE_PASSIVE;

        return 0;
    }

    return 1;
}

```

```

/**
 * Pairnei ws eisodo dyo komvous kai afou sygkrinei ta key tous,
 * grafei tis katalliles times stous deiktes tou patera kai tou paidiou.
 *
 * @param a      O prwtos komvos pros sygkrish
 * @param b      O deuterios komvos pros sygkrish
 * @param parent O deikths ston opoio tha deixnei o komvos me to mikrotero
key
 * @param child  O deikths ston opoio tha deixnei o komvos me to megalytero
key
 */
static inline void choose_order_pair( strict_fibonacci_node *a,
strict_fibonacci_node *b, strict_fibonacci_node **parent,
strict_fibonacci_node **child ) {

    if( a->key <= b->key ) {
        *parent = a;
        *child = b;
    }
    else {
        *parent = b;
        *child = a;
    }
}
/**
 * Pairnei ws eisodo treis komvous kai afou sygkrinei ta key tous,
 * grafei tis katalliles times stous deiktes tou pappou, tou patera kai tou paidiou.
 *
 * @param a      O prwtos komvos pros sygkrish
 * @param b      O deuterios komvos pros sygkrish
 * @param c      O tritos komvos pros sygkrish
 * @param grand  O deikths ston opoio tha deixnei o komvos me to mikrotero
key
 * @param parent O deikths ston opoio tha deixnei o komvos me to mesaio key
 * @param child  O deikths ston opoio tha deixnei o komvos me to megalytero
key
 */

static inline void choose_order_triple( strict_fibonacci_node *a,
strict_fibonacci_node *b, strict_fibonacci_node *c,
strict_fibonacci_node **grand, strict_fibonacci_node **parent,
strict_fibonacci_node **child ) {

    if( a->key < b->key ) {

```



```

    if( b->key < c->key ) {
        *grand = a;
        *parent = b;
        *child = c;
    }
    else if( a->key < c->key ) {
        *grand = a;
        *parent = c;
        *child = b;
    }
    else {
        *grand = c;
        *parent = a;
        *child = b;
    }
}
else {
    if( a->key < c->key ) {
        *grand = b;
        *parent = a;
        *child = c;
    }
    else if( b->key < c->key ) {
        *grand = b;
        *parent = c;
        *child = a;
    }
    else {
        *grand = c;
        *parent = b;
        *child = a;
    }
}
}

/**
 * Afairei enan komvo apo ti lista tw n aderfwn.
 *
 * @param queue H heap pou tha ginei h diadikasia
 * @param node Node pros diagrafh apo th lista
 */
static inline void remove_from_siblings( strict_fibonacci_heap *queue,
strict_fibonacci_node *node ) {

    if( node->parent == NULL )        //An o komvos einai h riza den kanoume tpt
        return;

    strict_fibonacci_node *next = node->right;

```

```

strict_fibonacci_node *prev;

if( next == node ) {          //An o deksios komvos einai o idios o komvos
    node->parent->left_child = NULL;
}
else {
    prev = node->left;
    next->left = prev;
    prev->right = next;
    if( node->parent->left_child == node )
        node->parent->left_child = next;
}

node->right = node;
node->left = node;
node->parent = NULL;
}

/**
 * Syndeai dyo dendra. O komvos pateras exei mikrotero key apo to paidi.
 * To child ginetai to aristerotero paidi tou parent.
 *
 * @param queue    H heap sthn opoia tha ginei h diadikasia
 * @param parent   Parent node
 * @param child    New child node
 */
static void link( strict_fibonacci_heap *queue, strict_fibonacci_node *parent,
strict_fibonacci_node *child ) {

    if( parent == child->parent )    // An to child exei patera ton parent, den
                                    // kanoume tpt
        return;

    if( child == queue->root )        // An to child einai h riza tote diagrafoume
                                    // ton patera apo th lista tw n aderfwn
        remove_from_siblings( queue, parent );
    else
        remove_from_siblings( queue, child ); //Alliws diagrafoume ton idio ton
                                                //komvo apo th lista tw n aderfwn

    strict_fibonacci_node *next = parent->left_child;
    strict_fibonacci_node *prev;

    if( parent->left_child == NULL ) //An o pateras den exei aristerotero paidi,
                                    //vazoume to child ws aristerotero
        parent->left_child = child;
    else {
        prev = next->left;

```

```

    child->right = next;
    child->left = prev;
    prev->right = child;
    next->left = child;

    if( is_active( queue, child ) )
        parent->left_child = child;
}

child->parent = parent;
}

/**
 * Psaxnei anamesa sta paidia ths palias rizas gia na vrei to neo elaxisto.
 *
 * @param queue H heap sthn opoia tha ginei h diadikasia
 * @return Deikths sth nea riza ths heap
 */
static strict_fibonacci_node* select_new_root( strict_fibonacci_heap *queue ) {

    strict_fibonacci_node *old_root = queue->root;
    strict_fibonacci_node *new_root = old_root->left_child;

    strict_fibonacci_node *current = new_root->right;
    while( current != old_root->left_child ) {
        if( current->key < new_root->key )
            new_root = current;
        current = current->right;
    }

    return new_root;
}

/**
 * Enthetei enan komvo sthn oura ths heap.
 *
 * @param queue H heap sthn opoia tha ginei h enthesh
 * @param node Node pros enthesh
 */
static void enqueue_node( strict_fibonacci_heap *queue, strict_fibonacci_node
*node ) {

    strict_fibonacci_node *next, *prev;

    if( queue->q_head != NULL ) { //Enthetoume ton komvo sthn oura ths heap
        next = queue->q_head;
        prev = next->q_prev;
    }

```

```

        node->q_next = next;
        node->q_prev = prev;
        next->q_prev = node;

    prev->q_next = node;
}

queue->q_head = node->q_next;
}

/**
 * Vgazoume apo th lista ths swrou enan komvo
 *
 * @param queue H heap sthn opoia tha ginei h diadikasia
 * @param node Node pros diagrafh apo th lista
 */
static void dequeue_node( strict_fibonacci_heap *queue, strict_fibonacci_node
*node ) {

    strict_fibonacci_node *prev;
    strict_fibonacci_node *next = node->q_next;
    if( next == node )
        queue->q_head = NULL;
    else {
        prev = node->q_prev;

        next->q_prev = prev;
        prev->q_next = next;

        node->q_next = node;
        node->q_prev = node;

        queue->q_head = next;
    }
}

/**
 * Metakinoume ton prwto komvo mias listas sthn oura ths.
 *
 * @param queue H heap sthn opoia tha ginei h diadikasia
 */
static strict_fibonacci_node* consume_node( strict_fibonacci_heap *queue ) {

    if( queue->q_head == NULL )
        return NULL;

    strict_fibonacci_node *target = queue->q_head;
    queue->q_head = target->q_next;

```

```

    return target;
}

```

Παρακάτω περιγράφονται οι συναρτήσεις οι οποίες αλλάζουν τον τύπο των κόμβων, π.χ. από ενεργό σε ενεργή ρίζα κτλ. Παράλληλα, γίνονται και οι κατάλληλες διεργασίες στις λίστες (fix, rank).

```

/**
 * Metatroph enos energou komvou se energh riza. Tou anathetoume enan fix
 node kai ton enthetoume antistoixa
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros metatroph
 */
static void convert_active_to_root( strict_fibonacci_heap *queue,
strict_fibonacci_node *node ) {

    if( is_active( queue, node ) && node->type == STRICT_TYPE_LOSS )
        convert_loss_to_active( queue, node );

    fix_node *fix = alloc_node( queue->map, STRICT_NODE_FIX );
    fix->node = node;
    fix->rank = node->rank;
    node->fix = fix;
    node->type = STRICT_TYPE_ROOT;

    insert_fix_node( queue, fix, STRICT_FIX_ROOT );
}

/**
 * Metrathroph enos energou komvou se komvou zhmiias loss node. Tou
 anathetoume enan fix node kai ton enthetoume antistoixa
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros metatroph
 */
static void convert_active_to_loss( strict_fibonacci_heap *queue,
strict_fibonacci_node *node ) {

    fix_node *fix = alloc_node( queue->map, STRICT_NODE_FIX );
    fix->node = node;
    fix->rank = node->rank;
    node->fix = fix;
    node->type = STRICT_TYPE_LOSS;

    insert_fix_node( queue, fix, STRICT_FIX_LOSS );
}

```

```
}
```

```
/**  
 * Metatroph mias energhs rizas se energo komvo. Ton afairoume apo th fix list  
 * kai kanoume free ton fix node.  
 *  
 * @param queue Heap sthn opoia tha ginei h diadikasia  
 * @param node Node pros metatroph  
 */
```

```
static void convert_root_to_active( strict_fibonacci_heap *queue,  
strict_fibonacci_node *node ) {
```

```
    remove_fix_node( queue, node->fix, STRICT_FIX_ROOT );  
    pq_free_node( queue->map, STRICT_NODE_FIX, node->fix );  
    node->fix = NULL;  
    node->type = STRICT_TYPE_ACTIVE;
```

```
}
```

```
/**  
 * Metatroph enos loss node se energo komvo. Ton afairoume apo th fix list  
 * kai kanoume free ton fix node.  
 *  
 * @param queue Heap sthn opoia tha ginei h diadikasia  
 * @param node Node pros metatroph  
 */
```

```
static void convert_loss_to_active( strict_fibonacci_heap *queue,  
strict_fibonacci_node *node ) {
```

```
    remove_fix_node( queue, node->fix, STRICT_FIX_LOSS );  
    pq_free_node( queue->map, STRICT_NODE_FIX, node->fix );  
    node->fix = NULL;  
    node->type = STRICT_TYPE_ACTIVE;
```

```
}
```

```
/**  
 * Metatroph enos pathitikou komvou se energo. Tou anathetoume to katallhlo  
active record  
 * kai rank record. To metakinoume pros to aristero akro ths listas me ta aderfia  
tou.  
 *  
 * @param queue Heap sthn opoia tha ginei h diadikasia  
 * @param node Node pros metatroph  
 */
```

```
static void convert_passive_to_active( strict_fibonacci_heap *queue,  
strict_fibonacci_node *node ) {
```

```

if( queue->active == NULL ) {

    queue->active = alloc_node( queue->map, STRICT_NODE_ACTIVE );

    queue->active->flag = 1;
}
node->active = queue->active;
queue->active->ref_count++;

rank_record *rank = queue->rank_list;
if( rank == NULL || rank->rank != 0 )
    rank = create_rank_record( queue, 0, queue->rank_list );
node->rank = rank;
rank->ref_count++;
node->type = STRICT_TYPE_ACTIVE;

node->parent->left_child = node;
}

/**
 * Metatroph enos energou komvou se pathitiko. Apeleutherwnoume ta fix nodes,
 * active kai
 * rank records.
 *
 * @param queue Heap sthn opoia tha ginei h diadikasia
 * @param node Node pros metatroph
 */
static void convert_to_passive( strict_fibonacci_heap *queue,
strict_fibonacci_node *node ) {

    if( node->fix != NULL ) {
        remove_fix_node( queue, node->fix, ( node->type == STRICT_TYPE_ROOT )
? STRICT_FIX_ROOT : STRICT_FIX_LOSS );
        pq_free_node( queue->map, STRICT_NODE_FIX, node->fix );
        node->fix = NULL;
    }

    release_rank_record( queue, node );
    release_active_record( queue, node );
    node->type = STRICT_TYPE_PASSIVE;

    if( node->parent != NULL )
        link( queue, node->parent, node );

    if( node->left_child == NULL )
        return;

    strict_fibonacci_node *current = node->left_child;

```

```

if( is_active( queue, current ) && current->type == STRICT_TYPE_ACTIVE )
    convert_active_to_root( queue, current );
current = current->right;
while( current != node->left_child ) {

    if( is_active( queue, current ) && current->type == STRICT_TYPE_ACTIVE )
        convert_active_to_root( queue, current );
    current = current->right;
}
}

```

Τέλος, έχουμε κάποιες συναρτήσεις οι οποίες πραγματοποιούν τη λεγόμενη **συλλογή σκουπιδιών (garbage collection)**.

Κατά τη διάρκεια μιας συγχώνευσης δύο σωρών, όλοι οι κόμβοι της μίας γίνονται παθητικοί αλλάζοντας τη σημαία στο active record που υπάρχει στο struct της σωρού. Τα records της σωρού, της rank και της fix list, αυτής της σωρού απελευθερώνονται για τη συλλογή σκουπιδιών.

```

/**
 * Dhmiourgei ena kainourio rank record me thn kathorismenh rank.
 * To topothetei sth rank list prin ton kathorismeno akoloutho.
 *
 * @param queue H heap sthn opoia tha ginei i diadikasia
 * @param rank H rank tou neou record
 * @param succ H epomenh rank sth lista (megalyterh)
 */
static rank_record* create_rank_record( strict_fibonacci_heap *queue, uint32_t
rank, rank_record *succ ) {

    rank_record *pred;
    rank_record *new_rank = alloc_node( queue->map, STRICT_NODE_RANK );
    new_rank->rank = rank;
    new_rank->inc = new_rank;
    new_rank->dec = new_rank;

    if( succ == NULL )
        queue->rank_list = new_rank;
    else {
        pred = succ->dec;

        new_rank->inc = succ;
        new_rank->dec = pred;

        succ->dec = new_rank;
        pred->inc = new_rank;
    }
    return new_rank;
}

```



```

/**
 * Apeleutherwnei mia anafora se ena active record kai to apeleutherwnei gia th
 syllogh skoupidiwn
 * an htan h teleutaia anafora.
 *
 * @param queue H heap sthn opoia tha ginei i diadikasia
 * @param node O node pou periexei thn anafora sto active record
 */
static void release_active_record( strict_fibonacci_heap *queue,
strict_fibonacci_node *node ) {

    node->active->ref_count--;
    if( node->active->ref_count == 0 ) {

        if( node->active == queue->active )
            queue->active = NULL;
        pq_free_node( queue->map, STRICT_NODE_ACTIVE, node->active );
    }
    node->active = NULL;
}

/**
 * Apeleutherwnei mia anafora se ena rank record kai to apeleutherwnei gia th
 syllogh skoupidiwn
 * an htan h teleutaia anafora.
 *
 * @param queue H heap sthn opoia tha ginei i diadikasia
 * @param node O node pou periexei thn anafora sto rank record
 */
static void release_rank_record( strict_fibonacci_heap *queue,
strict_fibonacci_node *node ) {

    rank_record *rank = node->rank;
    rank->ref_count--;
    if( rank->ref_count == 0 ) {
        if( rank->inc == rank ) {
            if( queue->rank_list == rank )
                queue->rank_list = NULL;
        }
        else {
            if( queue->rank_list == rank )
                queue->rank_list = rank->inc;

            rank->inc->dec = rank->dec;
            rank->dec->inc = rank->inc;
        }
    }
}

```

```

pq_free_node( queue->map, STRICT_NODE_RANK, rank );
}
node->rank = NULL;
}

/**
 * Apeleutherwnei mia heap gia ton syllekth skoupidiwn.
 * Episynaptei tis fix kai rank lists sthn allh queue.
 *
 * @param queue      H heap h opoia de tha diagrafei
 * @param garbage_queue H heap h opoia tha apeleutherwthei gia th syllogh
skoupidiwn
 */
static void release_to_garbage_collector( strict_fibonacci_heap *queue,
strict_fibonacci_heap *garbage_queue ) {

    int i;
    fix_node *tail, *head, *g_tail, *g_head;

    for( i = 0; i < 2; i++ ) {

        if( garbage_queue->fix_list[i] != NULL ) {

            if( queue->garbage_fix == NULL )
                queue->garbage_fix = garbage_queue->fix_list[i];
            else {
                head = queue->garbage_fix;
                tail = head->left;
                g_head = garbage_queue->fix_list[i];
                g_tail = g_head->left;

                head->left = g_tail;
                tail->right = g_head;
                g_tail->right = head;
                g_head->left = tail;
            }
        }
    }
}

```

```

/**
 * Afairei enan prohgomewws apeleutherwmeno komvo fix kai/'h rank list.
 *
 * @param queue Queue in which to operate
 */
static void garbage_collection( strict_fibonacci_heap *queue ) {

    fix_node *fix;

    if( queue->garbage_fix != NULL ) {
        fix = queue->garbage_fix;
        if( fix->right == fix )
            queue->garbage_fix = NULL;
        else {
            queue->garbage_fix = fix->right;
            fix->right->left = fix->left;
            fix->left->right = fix->right;
        }
    }
}

```

Για τη διαχείριση της μνήμης χρησιμοποιήθηκε κώδικας ο οποίος αποτέλεσε συρραφή κώδικα από το διαδίκτυο.

4. Πειραματική ανάλυση

Για την πειραματική ανάλυση της σωρού χρησιμοποιήσαμε κάποιες μεταβλητές για να μετρήσουμε το κόστος των επιμέρους πράξεων της σωρού. Συγκεκριμένα, χρησιμοποιήθηκε μια επιπλέον μεταβλητή-πεδίο στο struct του κόμβου ώστε να μετράει το κόστος της ένθεσης και δύο global μεταβλητές `delmin_cost` και `deckey_cost` για τη μέτρηση του κόστους της διαγραφής ελαχίστου και της ελάττωσης κλειδιού αντίστοιχα. Μετρήσαμε για κάθε συνάρτηση τις πράξεις που επιφέρουν κόστος. Αναλυτικά, προσφέρουν στο κόστος κατά μία μονάδα οι πράξεις της δημιουργίας κόμβου, της αλλαγής και ανάγνωσης δείκτη.

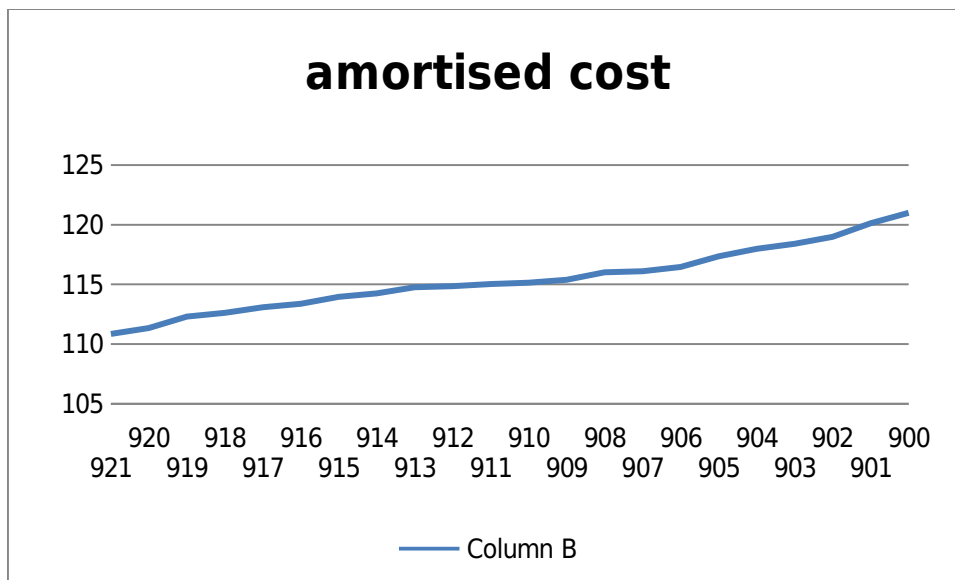
Με αυτά τα δεδομένα και κάνοντας 1000 ενθέσεις, 100 μειώσεις κλειδιού και 100 διαγραφές ελαχίστου παρατηρούμε τα εξής:

Το κόστος της ένθεσης στοιχείου είναι σταθερό ανεξάρτητα από το πλήθος των στοιχείων που βρίσκονται στη σωρό. Εξαιρέση αποτελεί το πρώτο στοιχείο του οποίου η ένθεση χρειάζεται λιγότερο χρόνο αφού δεν πραγματοποιούνται συγκρίσεις, συνδέσεις κτλ.

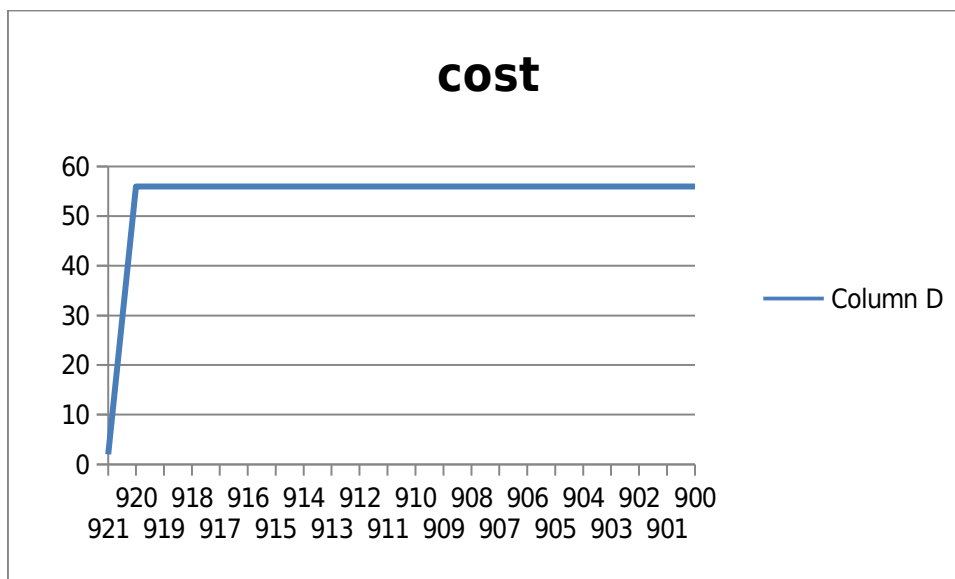
Το κόστος της διαγραφής ελαχίστου στοιχείου παρατηρούμε ότι είναι λογαριθμικό ως προς το πλήθος των στοιχείων, ενώ το κόστος της ελάττωσης κλειδιού είναι επίσης σταθερό.

Τέλος, σύμφωνα με το πόρισμα της εργασίας που μελετήθηκε και εισήγαγε την υλοποίηση της σωρού, όλοι οι κόμβοι έχουν τάξη το πολύ ίση με $2 \cdot \log n + 6$, κάτι που επαληθεύεται από τον κώδικα.

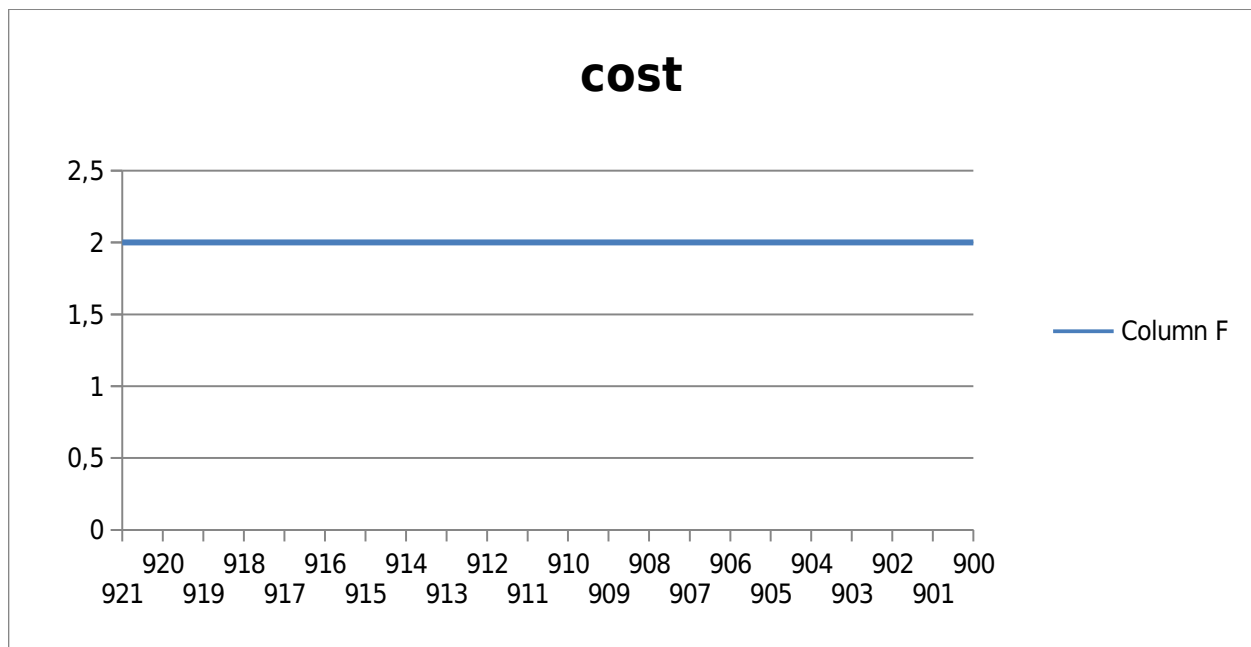
Ακολουθούν κάποια διαγράμματα που αφορούν το επιμερισμένο κόστος των τριών κύριων διεργασιών της σωρού ως προς το πλήθος των στοιχείων της:



Διάγραμμα: Το επιμερισμένο κόστος της delete-min ως προς το πλήθος των κόμβων της σωρού



Το κόστος της insert είναι σταθερό ως προς το πλήθος των κόμβων



Το κόστος της decrease-key είναι επίσης σταθερό

5. Βιβλιογραφία

1. <http://www.cs.au.dk/~gerth/papers/stoc12.pdf>
2. Παναγιώτης Δ. Μποζάνης. Δομές Δεδομένων (2006)
3. <http://www.cs.au.dk/~gerth/pub/stoc12.html>
4. <http://www.cforcoding.com/2009/07/plain-english-explanation-of-big-o.html>
5. <http://www.cs.princeton.edu/~wayne/cs423/fibonacci/FibonacciHeapAlgorithm.html>
6. <http://www.cs.princeton.edu/~wayne/cs423/fibonacci/FibonacciHeapAnimation.html>
7. [http://en.wikipedia.org/wiki/Heap_\(data_structure\)](http://en.wikipedia.org/wiki/Heap_(data_structure))
8. <http://www.cs.princeton.edu/~wayne/cs423/lectures/fibonacci-4up.pdf>
9. http://en.wikipedia.org/wiki/Fibonacci_heap
10. http://en.wikipedia.org/wiki/Big_O_notation
11. http://en.wikipedia.org/wiki/Priority_queue
12. <http://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf>
13. <http://courses.cms.caltech.edu/cs38/fibheap.pdf>
14. <http://code.google.com>
15. <http://code.google.com/p/priority-queue-testing/source/browse/trunk/queues/>