Πανεπιστήμιο Θεσσαλίας
Τμήμα Μηχανικών Η/Υ,Τηλεπικοινωνιών & Δικτύων

ΥΛΟΠΟΙΗΣΗ ΚΑΙ ΕΚΤΙΜΗΣΗ ΔΟΜΩΝ ΣΩΡΟΥ

ΚΑΡΠΑΤΣΗΣ ΓΕΩΡΓΙΟΣ

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ

ΜΠΟΖΑΝΗΣ ΠΑΝΑΓΙΩΤΗΣ

Περιεχόμενα

Εισαγωγή

Αντικείμενο της παρούσας διπλωματικής εργασίας είναι η υλοποίηση και η αξιολόγηση δύο σχετικά νέων δομών σωρού,των thin και thick heaps,οι οποίοι αποτελούν παραλλαγές της πλέον γνωστής δομής της συγκεκριμένης κατηγορίας,του σωρού Fibonacci.Οι δομές αυτές,υπόσχονται ίδια χρονική πολυπλοκότητα με αυτή του Fibonacci αλλά πιο αποδοτική διαχείρηση του χώρου,καθώς απαιτούν έναν δείκτη λιγότερο ανά κόμβο.

Στο πρώτο κεφάλαιο,παρουσιάζονται συνοπτικά οι βασικότερες έννοιες των δομών δεδομένων και της ανάλυσης αλγορίθμων,με έμφαση στις έννοιες που θα σηναντήσουμε στα επόμενα κεφάλαια της διπλωματικής.

Εν συνεχεία,στα πλαίσια του δευτέρου κεφαλαίου,αναλύονται οι βασικές έννοιες της κατηγορίας των δομών στην οποία ανήκουν και οι δύο σωροί που αποτελούν το αντικείμενο μελέτης μας.Η κατηγορία αυτή είναι οι ουρές προτεραιότητας.Έμφαση δίνεται στην ανάλυση του σωρού Fibonacci.

Ακολουθεί,στο τρίτο κεφάλαιο,η παρουσίαση των thin και thick heaps,με διεξοδική περιγραφή των πράξεων επί των συγκεκριμένων σωρών,με τη βοήθεια παραδειγμάτων.Παρουσιάζεται επίσης,ο κώδικας υλοποίησής τους σε Java ενώ στο κεφάλαιο αυτό γίνεται και η ανάλυση της πολυπλοκότητάς τους σε θεωρητικό επίπεδο.

Τέλος,στο τέταρτο κεφάλαιο,συναντάμε την πειραματική αξιόλογηση των εν λόγω δομών,μέσω πλήθους πειραμάτων από τα οποία εξάγουμε τα επιθυμητά αποτελέσματα και τα παρουσιάζουμε με τη μορφή γραφημάτων.

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 οτι 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 εφανιζόμενων ***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)*** περάτωσής 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,... οι οποίες συνιστούν την κύρια μνήμη του,και

(β) είναι σε θέση να εκτελεί τις αριθμητικές πράξεις {+, -, *, /, mod},να παίρνει αποφάσεις δικλάδωσης (τύπου **if**) βάσει των τελεστών {=, <, >, ≤, ≥, ≠} 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.Η πρώτη είναι η μέτρηση μοναδιαίου κόστους (unit cost measure) όπου σε κάθε πράξει χρεώνεται σταθερό (πεπερασμένο) κόστος,ανεξαρτήτους του μήκους της δυαδικής αναπαραστάσεως των τελεστέων (operands).Η δεύτερη,η μέτρηση λογαριθμικού κόστους (logarithmic cost

measure),θεωρεί πως η πράξη παίρνει χρόνο ανάλογο με το μήκος της δυαδικής αναπαραστάσεως των τελεστέων.

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:

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

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:

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

Το κατανεμημένο κόστος c'_i της o_i ορίζεται ως:

$$c'_i = c_i + \mathbf{\Delta\Phi_i}$$

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.**Χάρις στην τελευταία,κάποιες πράξεις χρεώνονται περισσότερο (όταν $\mathbf{\Delta\Phi_i} > 0$) και κάποιες λιγότερο (όταν $\mathbf{\Delta\Phi_i} < 0$),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:

- Create. Επιστρέφει έναν νέο,άδειο σωρό.
- Insert(x,k,h). Επιστρέφει τον σωρό που δημιουργείται αν στον σωρό h εισάγουμε το αντικείμενο x με προτεραιότητα k.
- Find-min(h). Επιστρέφει ,χωρίς να αφαιρεί,τον κόμβο με την ελάχιστη προτεραιότητα του σωρού h.
- Delete-min(h). Επιστρέφει τον σωρό που δημιουργείται αν από τον σωρό h αφαιρέσουμε τον κόμβο με την ελάχιστη προτεραιότητα.
- Meld(h1,h2). Επιστρέφει τον σωρό που δημιουργείται από την συνένωση των σωρών h1 και h2.
- Decrease-key(d,x,h). Μειώνει κατά d την προτεραιότητα του αντικειμένου x του σωρού h.
- Delete(x,h). Επιστρέφει τον σωρό που προκύπτει από την αφαίρεση του αντικειμένου x από τον σωρό h.

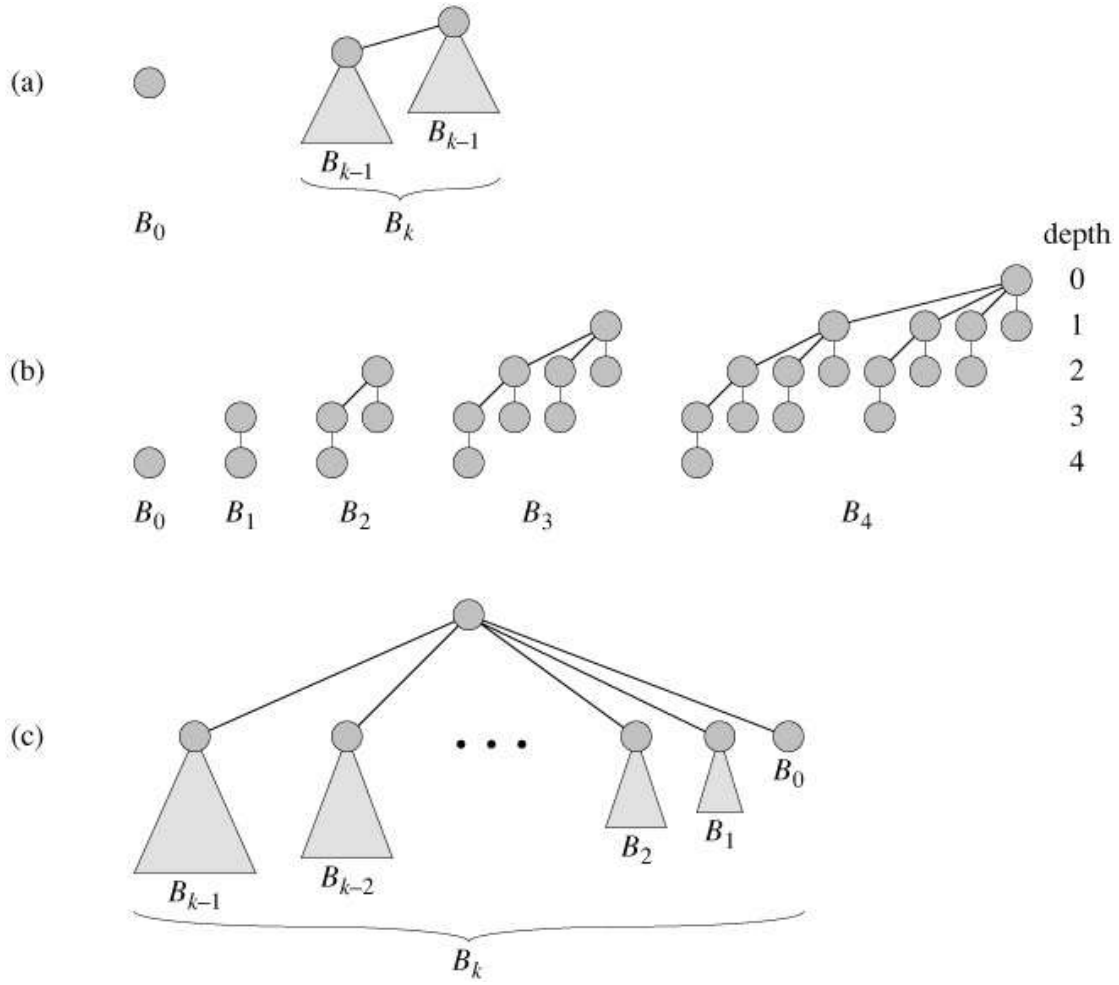
Οι παραπάνω ορισμοί αφορούν την ουρά προτεραιότητας ελαχίστου.Ανάλογοι ορισμοί ισχύουν και για την περίπτωση της ουράς προτεραιότητας μεγίστου.

2.2 Δυωνυμικό Δένδρο

Ένα ***δυωνυμικό δένδρο (binomial tree)*** B_k τάξης k,είναι είτε κόμβος,αν k = 0,είτε αποτελείται από δύο δυωνυμικά δένδρα B_{k-1} τάξης k-1,με την ρίζα του ενός να αποτελεί το αριστερό-πρώτο παιδί της ρίζας του άλλου (Σχήμα 2.1).Εάν «ξεδιπλώσουμε» την αναδρομή,τότε μπορούμε να δούμε πως,εναλλακτικά,ένα δυωνυμικό δένδρο B_k τάξης k αποτελείται από έναν κόμβο με k παιδιά-δυωνυμικά δένδρα B_j, $0 \leq j \leq k-1$ (Σχήμα 2.1 (c)). Ένα δυωνυμικό δένδρο τάξης k περιέχει 2^k κόμβους και η ρίζα του έχει ακριβώς k παιδιά.

2.3 Δένδρο Διατάξεως Σωρού

Τα ***δένδρα διατάξεως 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 2.1: (a) Αναδρομικός ορισμός δυωνυμικού δένδρου,(b)στιγμιότυπα διωνυμικών δένδρων B_0-B_4,(c)εναλλακτική θεώρηση δυωνιμικού δένδρου.

2.4 Δυωνυμική Ουρά

Η ***δυωνυμική ουρά (binomial queue)*** ανήκει στην κατηγορία των ουρών προτεραιότητας.Χρησιμοποιεί ως δομικό στοιχείο την συνδυαστική δομή δυωνυμικό δένδρο (βλέπε 2.2).Συγκεκριμένα,αποτελείται από ένα σύνολο αδιάτακτων δυωνυμικών δένδρων που τηρούν την αμετάβλητη συνθήκη σωρού ελαχίστου,είναι δηλαδή συνδυασμός των δομών δυωνυμικού δένδρου και δένδρου διατάξεως σωρού που περιγράψαμε στις παραγράφους 2.2 και 2.3.Η δομή που προκύπτει από τον συνδυασμό των παραπάνω δομών καλείται ***δυωνυμικός σωρός*** κι έτσι μπορούμε να πούμε οτι μια δυωνυμική ουρά επί ενός συνόλου S,n στοιχείων με προτεραιότητες,ορίζεται ως ένα δάσος F_n από δυωνυμικούς σωρούς.

Ως προς την απόδοση,μια δυωνυμική ουρά χαρακτηρίζεται από λογαριθμική συμπεριφορά πράξεων,ενώ η δόμηση μιας αρχικώς άδειας ουράς με διαδοχικές ενθέσεις,κοστίζει γραμμικό χρόνο.

2.5 Σωρός Fibonacci

Αποτελεί μια παραλλαγή των δυωνυμικών ουρών.Δομικό στοιχείο του 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 του 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).**

2.5.1 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 χ στον σωρό,πρώτον σχηματίζουμε το στοιχειώδες δένδρο για την στέγαση του χ (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 χ 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 αποκόπτοντάς την από τα παιδιά της.Έπειτα,συνενώνουμε την λίστα των ορφανών παιδιών με τη λίστα του δάσους και εφρμόζουμε διαδοχικές συνενώσεις δύο οποιονδήποτε δένδρων ιδίας 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.Κατα 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.Κατα 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

Για την ανάλυση,χρησιμοποιούμε την τεχνική του δυναμικού της αναλύσεως επιμερισμένου κόστους.Ως δυναμικό **Φ** της δομής ορίζεται το πλήθος των δένδρων διατάξεως σωρού συν δύο φορές το πλήθος των σημαδεμένων κόμβων.Ως κόστος επιμερισμού μιας πράξεως,λογίζεται ο πραγματικός χρόνος συν την μεταβολή του δυναμικού που προκαλεί.

Λήμμα 2.1. Ένας κόμβος τάξεως k διαθέτει τουλάχιστον $F_{k+1} \geq \varphi^k$ απογόνους,συμπεριλαμβανομένου του εαυτού του,όπου F_{k+2} είναι ο (k+2)-στός αριθμός Fibonacci και $\varphi = (1+\sqrt{5})/2$ η χρυσή αναλογία.

Θεώρημα 2.1. Έστω μια αναμεμιγμένη ακολουθία πράξεων επί ενός,αρχικώς άδειου,σωρού Fibonacci.Το επιμερισμένο κόστος μιας αποσβέσεως ελαχίστου ή αποσβέσεως στοιχείου είναι λογαριθμηκό στο πλήθος των στοιχείων,ενώ κάθε άλλη πράξη επιδεικνύει σταθερού κόστους επιμερισμένη χρονική συμπεριφορά.

Στην περίπτωση της αποσβέσεως ελαχίστου,αν η ρίζα με το ελάχιστο στοιχείο είναι τάξεως k,τότε:

$$\varphi^k \leq n \Rightarrow k \leq \log_\varphi n \leq 1.44 \log n$$

Συνεπώς,η απόσβεσή της αυξάνει το πλήθος των δένδρων,το πολύ,κατά 1.44 logn.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(logn).

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(logn).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.

3.1 Thin Heap

Αν και ο σωρός Fibonacci είναι θεωρητικά αρκετά αποδοτικός σε όρους επιμερισμένης ανάλυσης,δεν είναι τόσο αποδοτικός στην χειρότερη περίπτωση ούτε κάνει αποδοτική χρήση του χώρου,καθώς για την υλοποίησή του απαιτούνται τέσσερεις δείκτες ανά κόμβο.Οι δομές που θα εξετάσουμε στο παρόν κεφάλαιο,παρουσιάζουν καλύτερη διαχείριση του χώρου,ίδια θεωρητική επιμερισμένη ανάλυση ενώ υπόσχονται καλύτερες επιδόσεις στην πράξη.

Κοινός παρονομαστής σε όλες τις παραλλαγές του σωρού Fibonacci που υλοποιούμε,είναι η χρήση τριών δεικτών ανά κόμβο,συγχωνεύοντας τον δείκτη του αριστερού αδελφού και του πατέρα,σε έναν.Η πρώτη δομή που θα εξετάσουμε είναι ο thin heap.

Ένας thin heap είναι ένα αδιάτακτο σύνολο (δάσος) από,ξένα ως προς τα στοιχεία που φέρουν,thin trees τα οποία τηρούν την αμετάβλητη συνθήκη σωρού ελαχίστου.

Thin Tree. Ένα thin tree είναι ένα δυωνυμικό δένδρο στο οποίο οποιοσδήποτε κόμβος του μπορεί μπορεί να «χάσει» το πρώτο-αριστερότερό του παιδί και το αντίστοιχο υποδένδρο.Τυπικότερα,είναι ένα διατεταγμένο δένδρο που αποτελείται από κόμβους με μη αρνητικές ακέραιες τιμές τάξεων (ranks) και έχει τις εξής ιδιότητες:

(1) Τα παιδιά ενός κόμβου με k παιδιά,είναι τάξεως k-1,k-2,…,0 από αριστερά προς τα δεξιά.
(2) Έστω ένας κόμβος με k παιδιά.Αν έχει τάξη k τον αποκαλούμε normal,ενώ αν έχει τάξη k+1 καλείται thin.
(3) Η ρίζα ενός thin tree είναι πάντα normal.

Με το όρο «τάξη ενός thin tree» αναφερόμαστε στην τάξη της ρίζας του.Ένα thin tree του οποίου όλοι οι κόμβοι είναι normal,είναι ένα δυωνυμικό δένδρο.Όπως και στα δυωνυμικά δένδρα,αν συνδέσουμε δύο thin trees ίδιας τάξης κάνοντας τη ρίζα του ενός αριστερό παιδί της ρίζας του άλλου και αυξάνοντας την τάξη του κατά ένα,τότε το αποτέλεσμα είναι επίσης ένα thin tree.

Λήμμα 3.1. Ένας κόμβος με k παιδιά που ανήκει σε ένα thin tree,έχει τουλάχιστον $F_{k+1} \geq \varphi^k$ απογόνους,συμπεριλαμβανομένου του εαυτού του,όπου F_k είναι ο k-στός αριθμός Fibonacci και $\varphi = (1+\sqrt{5})/2$ η χρυσή αναλογία.

3.1.1 Περιγραφή Πράξεων

Για την υποστήριξη των πράξεων του σωρού,τα συστατικά δένδρα χρησιμοποιούν την αναπαράσταση πρώτου παιδιού-δεξιού αδελφού,με έναν επιπλέον,ανά κόμβο,δείκτη προς τον αριστερό αδελφό ή τον πατέρα αν ο κόμβος είναι πρώτο παιδί.Αναλυτικότερα,οι ρίζες του δάσους συμμετέχουν σε απλή κυκλική λίστα στην κορυφή της οποίας βρίσκεται πάντα ο κόμβος με την ελάχιστη

προτεραιότητα.Επίσης,ο κάθε κόμβος,σχηματίζει διπλή μη κυκλική λίστα με τα παιδιά του.Τέλος,η πληροφορία που κρατάμε για κάθε κόμβο είναι η τάξη και η προτεραιότητά του.

Δημιουργία Σωρού. Υλοποιείται σε σταθερό χρόνο,επιστρέφοντας έναν δείκτη null.

Ένθεση Στοιχείου. Η ένθεση ενός στοιχείου x ανάγεται πρώτον στην δημιουργία του δένδρου το οποίο θα αποτελείται από έναν κόμβο ο οποίος θα στεγάζει το εν λόγω στοιχείο (κόστος O(1)) ,δεύτερον στην συνένωση του x με την κυκλική λίστα των ριζών του δάσους (κόστος O(1)) και τρίτον στην εύρεση του νέου ελαχίστου,με απλή σύγκριση σταθερής πολυπλοκότητας,της προτεραιότητας του x με την ελάχιστη προτεραιότητα της δομής.Η ένθεση του στοιχείου γίνεται στην πρώτη ή την δεύτερη θέση ανάλογα με το αν η προτεραιότητά του είναι μικρότερη ή όχι από αυτή του τρέχοντος ελαχίστου στοιχείου.

Στα (α) και (β) του σχήματος 3.1 παρέχονται παραδείγματα ενθέσεως στοιχείων.Η πληροφορία που περιέχει κάθε κόμβος είναι η προτεραιότητα και η τάξη του.Παρατηρούμε οτι οι διαδοχικές ενθέσεις σε έναν αρχικώς άδειο σωρό, δημιουργούν ένα δάσος από ρίζες οι οποίες σχηματίζουν μια κυκλική λίστα.

Έυρεση Ελαχίστου. Για την πράξη της ευρέσεως ελαχίστου,απλώς επιστρέφουμε τον δείκτη που δείχνει στο ελάχιστο στοιχείο-ρίζα της δομής.

Συγχώνευση Σωρών. Η συγχώνευση δύο σωρών είναι πράξη σταθερού χρόνου και υλοποιείται ως συνένωση των αντίστοιχων κυκλικών λιστών των ριζών.Έπειτα,θέτουμε ως νέο ελάχιστο,το ελάχιστο των δύο ελαχίστων στοιχείων.Στο (γ) του σχήματος 3.1 φαίνεται το αποτέλεσμα της συγχώνευσης των σωρών του (α) και (β).

Απόσβεση Ελαχίστου. Αρχικά,αποκόπτουμε το δένδρο με το ελάχιστο στοιχείο από την λίστα των ριζών.Διαγράφουμε την ρίζα του δένδρου αυτού και στην συνέχεια συνενώνουμε την λίστα των,ορφανών πλέον,παιδιών της με την λίστα των ριζών του δάσους,κάνοντας προηγουμένως όλα τα παιδιά normal αν δεν είναι ήδη,με μείωση κατά ένα της τάξης τους.Κατόπιν,εφαρμόζουμε διαδοχικές συνενεώσεις δύο οποιονδήποτε δένδρων ίδιας τάξης,κάνοντας τη ρίζα του δένδρου με την μεγαλύτερη προτεραιότητα αριστερό παιδί της ρίζας του δένδρου με την μικρότερη,έως ότου προκύψει δάσος με δένδρα διαφορετικής τάξεως.Τέλος,σαρώνουμε τα μέλη του δάσους κι εντοπίζουμε το νέο ελάχιστο.

Στιγμιότυπο αποσβέσεως ελαχίστου αποτελεί το σχήμα 3.1 (δ).Το στοιχειώδες δένδρο με προτεραιότητα 1 αφαιρείται,δίχως να προκαλέσει ορφανά.Στη συνέχεια,έχουμε διαδοχικές συνενώσεις των δένδρων τάξεως 0,όπου δημιουργούνται επτά δένδρα τάξεως 1.Έπειτα,έξη από αυτά συγχνωνεύονται σε τρία δένδρα τάξεως 2 και τέλος,δύο από αυτά σχηματίζουν ένα δένδρο τάξεως 3.Το αποτέλεσμα της απόσβεσης ελαχίστου,είναι ένα δάσος που αποτελείται από τρία δένδρα τάξεως 3,1 και 2 αντίστοιχα.

Μείωση Προτεραιότητας. Η υλοποίηση της συγκεκριμένης πράξης διαφέρει αρκετά από την αντίστοιχη του σωρού Fibonacci.Αρχικά,θέτουμε στον κόμβο x τη νέα του προτεραιότητα.Αν ο x είναι πρώτο παιδί και η προτεραιότητά του παραμένει μικρότερη ή ίση από αυτήν του πατέρα του,τότε η πράξη έχει ολοκληρωθεί.Αν ο x είναι ρίζα,τότε συγκρίνουμε τη νέα του προτεραιότητα με το ελάχιστο στοιχείο της λίστας των ριζών και στην περίπτωση που αυτή είναι μικρότερη,θέτουμε το x ως νέο ελάχιστο.Σε περίπτωση που δεν ισχύει τίποτα από τα παραπάνω,θέτουμε y ίσο με τον αριστερό αδελφό του x ή τον πατέρα του,στην περίπτωση που ο x είναι πρώτο παιδί.Αποκόπτουμε τον x από την λίστα των παιδιών-και κατά συνέπεια και από τον πατέρα του-και τον ενθέτουμε στο δάσος σαν νέα ρίζα (μαζί με το υποδένδρο του),στην πρώτη ή δεύτερη θέση,ανάλογα με το αν η προτεραιότητά του είναι μικρότερη ή όχι από αυτή του τρέχοντος ελαχίστου.Πριν το ενθέσουμε,το κάνουμε normal αν δεν είναι.

Η αποκοπή του x,ενδέχεται να προκαλέσει στον y παραβίαση των ιδιοτήτων (1),(2) ή (3) που περιγράψαμε κατά τον ορισμό του thin tree.Επαναλαμβάνουμε το παρακάτω βήμα επιδιόρθωσης της δομής,το οποίο διορθώνει την παράβαση στον y αλλά ενδέχεται να προκληθεί νέα παράβαση στον αριστερό αδελφό ή τον πατέρα του y,μέχρις ότου δεν υπάρχει καμία παράβαση:

Βήμα Επιδιόρθωσης (Repair Step)

Περίπτωση 1. Παραβίαση του (1):Ο κόμβος y έχει τάξη κατά δύο μεγαλύτερη από αυτή του δεξιού αδελφού του,ή έχει τάξη 1 και κανέναν δεξί αδελφό.

1α. Ο κόμβος y είναι thin.Μειώνουμε την τάξη του κατά ένα, διορθόνωντας την παράβαση και κάνοντας τον normal.Αντικαθιστούμε τον y με τον αριστερό του αδελφό ή τον πατέρα του αν είναι πρώτο παιδί και ελέγχουμε για παράβαση στον νέο y.

1β. Ο κόμβος y είναι normal.Έστω w το πρώτο παιδί του y.Το αφαιρούμε από από την λίστα των παιδιών του y και το εισάγουμε ως δεξί αδελφό του y,στην λίστα των παιδιών στην οποία περιέχεται ο y.Αυτό κάνει τον κόμβο y thin,αλλά επιδιορθώνει την παράβαση χωρίς να προκαλεί νέα.

Το σχήμα 3.2 (ε) αποτελεί στιγμιότυπο της περίπτωσης 1β.Μειώνουμε την προτεραιότητα του 20 σε 2.Σύμφωνα με την παραπάνω περιγραφή,ο y είναι ο αριστερός αδελφός του 20,δηλαδή το 8 και ο w είναι το πρώτο παιδί του y δηλαδή ο 13.Πλέον,θέτοντας 20 = 2,ο συγκεκριμένος κόμβος έχει μικρότερη προτεραιότητα από αυτή του πατέρα του (4) και αυτό έχει ως συνέπεια να αποκόψουμε τον 2 και να τον ενθέσουμε στη λίστα των ριζών,θέτοντάς τον μάλιστα ως νέο ελάχιστο καθώς 2<3 που ήταν το προηγούμενο ελάχιστο.Στην θέση του 2 εισάγουμε τον w = 13,κάτι που έχει σαν συνέπεια ο y να γίνει thin καθώς χάνει ένα παιδί.Εδώ τελειώνει η πράξη της μείωσης προτεραιότητας,καθώς δεν παραβιάζεται κάποιος κανόνας.

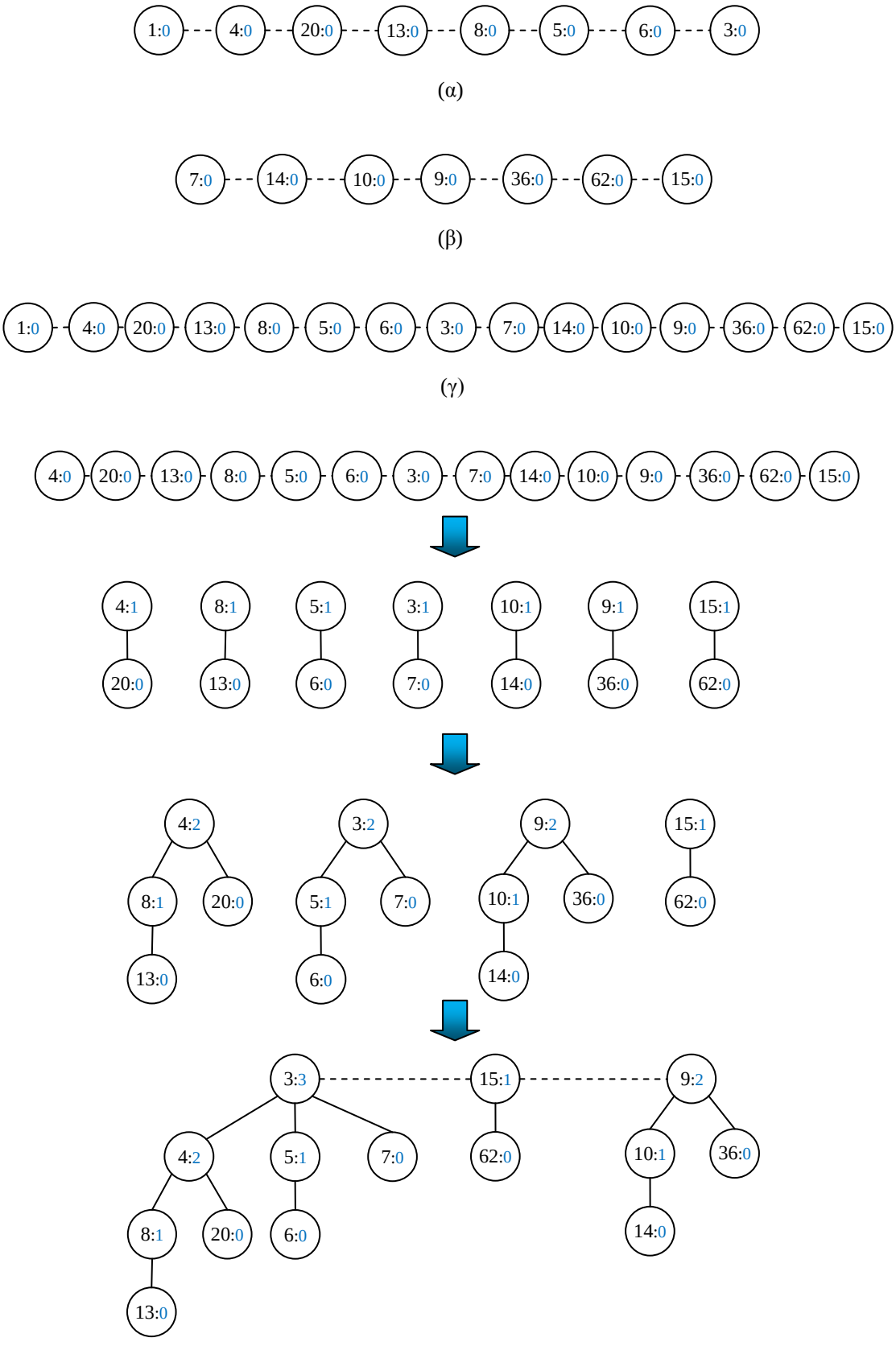
Στο σχήμα 3.2 (στ) παρουσιάζεται ένα στιγμιότυπο της περίπτωσης 1α.Μειώνουμε την προτεραιότητα του 13 σε 1 κάτι που σημαίνει οτι πλέον έχει μικρότερη προτεραιότητα από αυτήν του πατέρα του (4).Τον αποκόπτουμε από την λίστα παιδιών του 4 και τον ενθέτουμε στην λίστα των ριζών ως νέο ελάχιστο,καθώς 1<2.Στην συνέχεια εξετάζουμε τον αριστερό του αδελφό,τον 8,ο οποίος είναι thin,έχει τάξη 1 και κανέναν αδελφό.Μειώνουμε την τάξη του 8 κατά ένα,κάνοντας τον παράλληλα normal,διορθώνοντας έτσι την παράβαση.Στην συνέχεια εξετάζουμε τον πατέρα του 8,δηλαδή τον κόμβο με προτεραιότητα 4.Διαπιστώνουμε οτι δεν παραβιάζεται κάποια ιδιότητα των thin trees,επομένως η πράξη τελειώνει εδώ.

Περίπτωση 2. Παραβίαση του (2):Ο κόμβος y έχει τάξη κατά τρία μεγαλύτερη από αυτή του πρώτου παιδιού του,ή έχει τάξη δύο και κανένα παιδί.Μειώνουμε την τάξη του y κατά δύο,κάνοντάς τον normal κι επιδιορθώνοντας την παράβαση.Έστω z ο αριστερός αδελφός ή ο πατέρας του y,αν ο y είναι πρώτο παιδί.Αφαιρούμε τον y από την λίστα των παιδιών στην οποία περιέχεται,και τον ενθέτουμε στην δεύτερη θέση της λίστας των ριζών,δεξιά δηλαδή του ελαχίστου στοιχείου.Αντικαθιστούμε τον y με τον z (θέτουμε δηλαδή y = z) και ελέγχουμε για παράβαση στον νέο y.

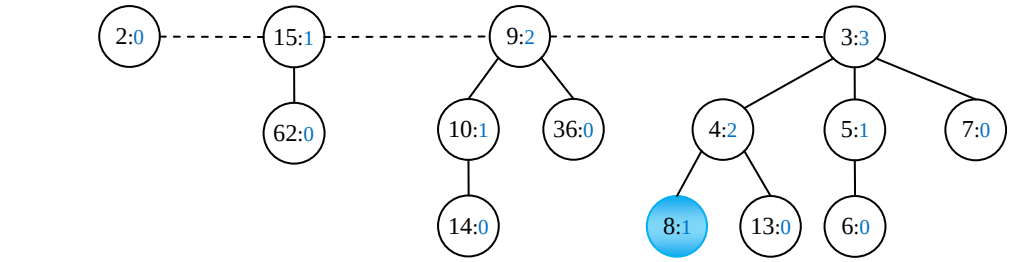
Περίπτωση 3. Παραβίαση του (3):Ο κόμβος y είναι thin ρίζα.Κάνουμε τον y normal,μειώνοντας την τάξη του κατά ένα.

Το σχήμα 3.2 (ζ) αποτελεί στιγμιότυπο των περιπτώσεων 2 και 3.Μειώνουμε την προτεραιότητα του 8 σε 3,αποκόπτοντάς τον έτσι από τον πατέρα του και ενθέτοντάς τον στο δάσος.Πλέον ο κόμβος 4 έχει τάξη 2 και κανένα παιδί.Μειώνουμε την τάξη του κατά δύο,κάνοντας τον normal,και τον αποκόπτουμε από τον πατέρα του,ο οποίος είναι ο κόμβος 3,ενθέτοντάς τον στο δάσος.Στην συνέχεια,ελέγχουμε τον πατέρα του.Παρατηρούμε ότι είναι thin αλλά είναι ρίζα.Διορθώνουμε την παράβαση μειώνοντας κατά ένα την τάξη του,κάνοντάς τον έτσι,normal.

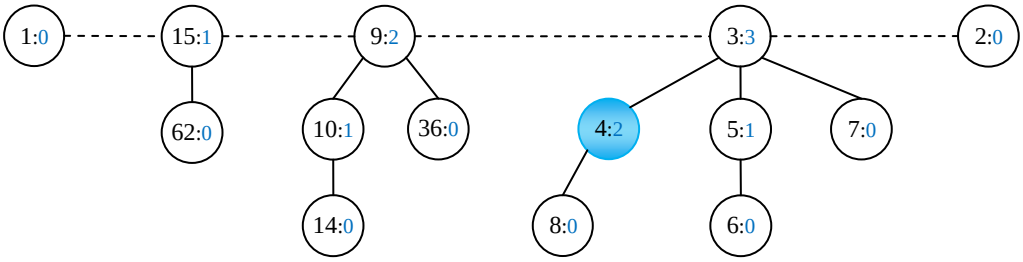
Διαγραφή Στοιχείου. Υλοποιείται ως deleteMin(decreaseKey(∞,x,h)).Μειώνουμε δηλαδή την προτεραιότητα του στοιχείου που θέλουμε να διαγράψουμε ώστε να γίνει ελάχιστη και στη συνέχεια εκτελούμε διαγραφή ελαχίστου.Το σχήμα 3.3 (η) αποτελεί στιγμιότυπο της διαγραφής του στοιχείου με προτεραιότητα 4.Αρχικά εκτελούμε μείωση προτεραιότητας του 4 σε 0 (θέτοντας 4 = τρέχων ελάχιστο-1 = 1-1 = 0) και στη συνέχεια διαγραφή ελαχίστου.Με όμοιο τρόπο όπως και στο σχήμα 3.1 (δ),τακτοποιείται η δομή μας και καταλήγουμε σε ένα δάσος με τρία δένδρα διακριτής τάξεως.



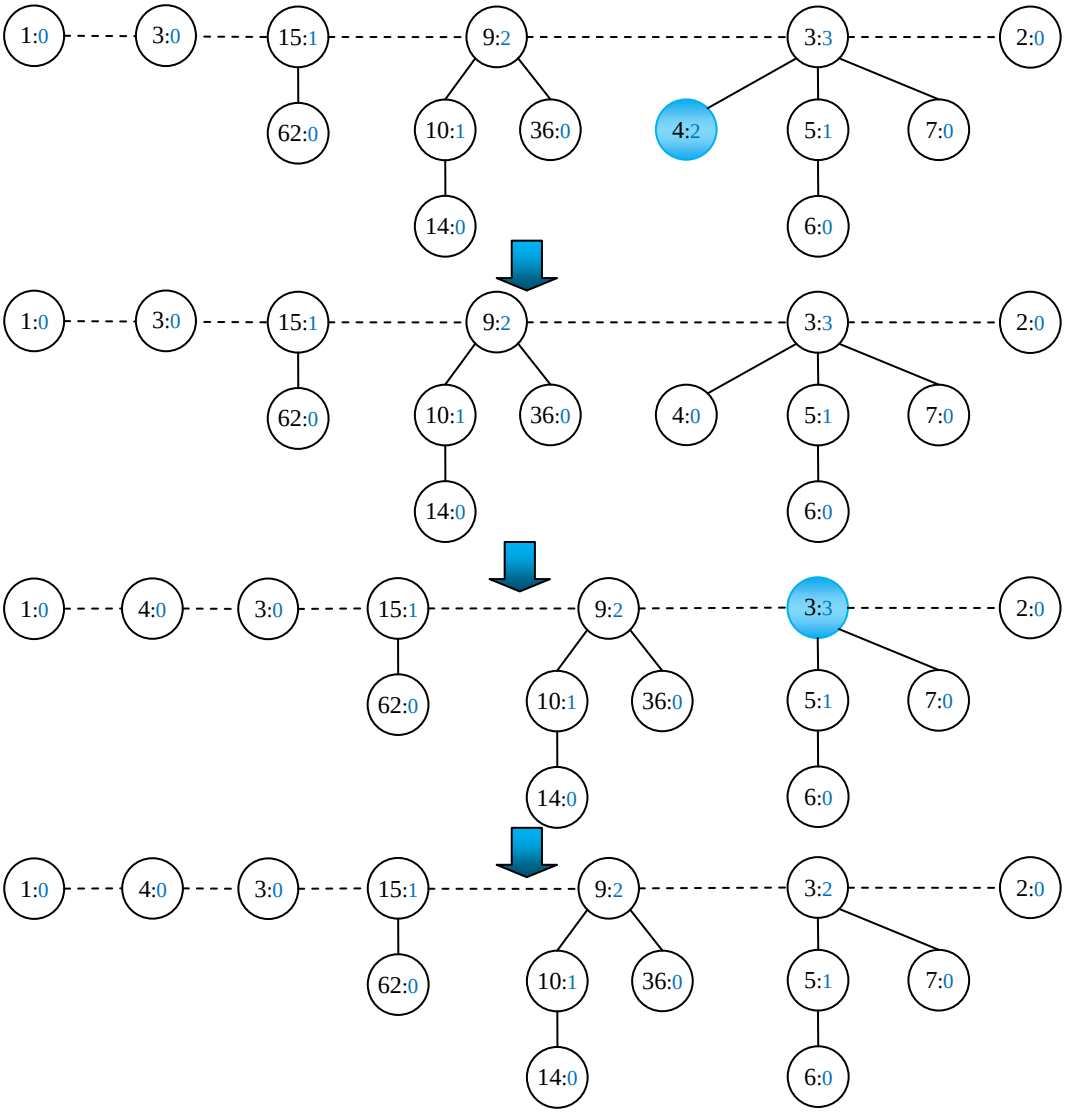
Σχήμα 3.1: Στιγμιότυπα πράξεων thin heap: (α)-(β) δύο σωροί,ως αποτέλεσμα διαδοχικών ενθέσεων,(γ) η συνένωσή τους,(δ) απόσβεση ελαχίστου.


2:0
15:1
9:2
3:3
62:0
10:1
36:0
4:2
5:1
7:0
14:0
8:1
13:0
6:0

(ε)

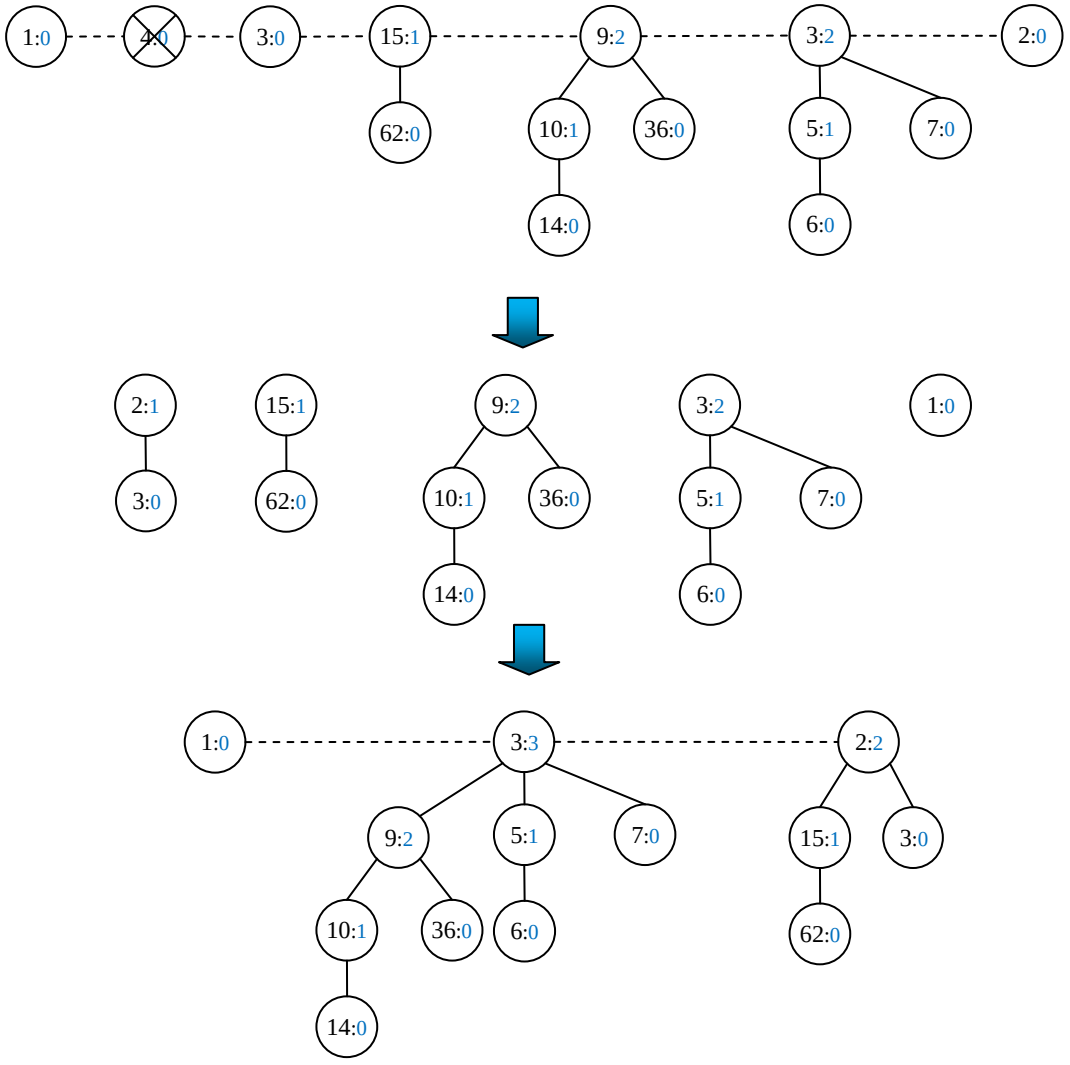

1:0
15:1
9:2
3:3
2:0
62:0
10:1
36:0
4:2
5:1
7:0
14:0
8:0
6:0

(στ)


1:0
3:0
15:1
9:2
3:3
2:0
62:0
10:1
36:0
4:2
5:1
7:0
14:0
6:0
4:0
1:0
4:0
3:0
15:1
9:2
3:3
2:0
62:0
10:1
36:0
5:1
7:0
14:0
6:0
1:0
4:0
3:0
15:1
9:2
3:2
2:0
62:0
10:1
36:0
5:1
7:0
14:0
6:0

(ζ)

Σχήμα 3.2: (συνέχεια),με γαλάζιο συμβολίζονται οι thin κόμβοι,(ε) μείωση της προτεραιότητας του 20 σε 2, (στ) μείωση της προτεραιότητας του 13 σε 1, (ζ) μείωση της προτεραιότητας του 8 σε 3.



(η)

Σχήμα 3.3: (συνέχεια) (η) διαγραφή του κόμβου με προτεραιότητα 4.

3.1.2 Υλοποίηση

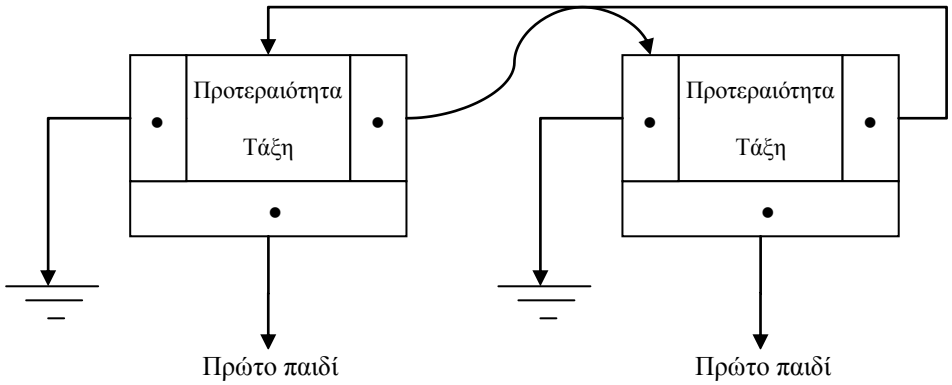
Δομικό στοιχείο του thin heap αποτελεί η κλάση κόμβος tqNode,η οποία περιλαμβάνει πληροφορία για την τάξη ενός κόμβου,την προτεραιότητά του,το αν είναι thin ή όχι καθώς επίσης και μεθόδους για τον χειρισμό των τριών δεικτών(σε πρώτο παιδί,δεξί αδελφό και αριστερό αδελφό ή πατέρα).

Κώδικας 3.1: Κόμβος Thin Heap

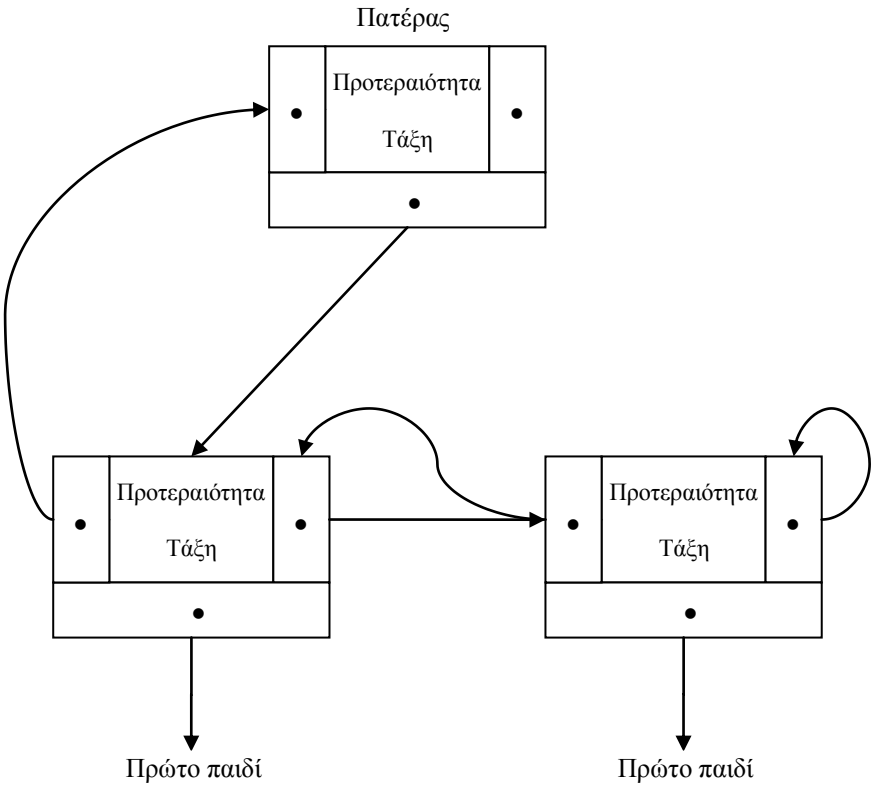
```
public class tqNode {
   private int rank,key;                  //Ο δείκτης left δείχνει στο πρώτο παιδί
   private tqNode left,previous,right;    //Ο previous στον αριστερό αδελφό ή στον
   private boolean thin;                  //πατέρα αν κόμβος μας είναι πρώτο παιδί
                                          //Ο right στον δεξί αδελφό ή στην επόμενη ρίζα
   public tqNode() {}                     //Η μεταβλητή thin είναι αληθής αν ο
                                          //κόμβος μας είναι thin
   public tqNode(int k) {
      setKey(k);
      setLeft(null);
      setRight(this);
      setPrevious(this);
      rank = 0;
      thin = false;
   }

   public int getKey(){                   //επιστρέφει την προτεραιότητα
      return key;
   }

   public void setKey(int k){             //θέτει την προτεραιότητα
      key = k;
   }

   public int getRank(){                  //επιστρέφει την τάξη
      return rank;
   }

   public void setRank(int i){            //θέτει την τάξη
      rank = i;
   }

   public void incRank(){                 //αυξάνει την τάξη κατά ένα
      rank++;
   }
```

```
    public void decRank(){              //μειώνει την τάξη κατά ένα
        rank--;
    }

    public boolean isThin(){            //είναι thin;
        return thin;
    }

    public void setThin(){              //κάνει τον κόμβο thin
        thin = true;
    }

    public void unThin(){               //κάνει τον κόμβο normal
        thin = false;
    }

    public tqNode getPrevious(){        //επιστρέφει τον προηγούμενο
        return previous;
    }

    public void setPrevious(tqNode p){          //θέτει τον προηγούμενο
        previous = p;
    }

    public tqNode getLeft(){            //επιστρέφει το πρώτο παιδί
        return left;
    }

    public void setLeft(tqNode l){      //θέτει το πρώτο παιδί
        left = l;
    }

    public tqNode getRight(){           //επιστρέφει τον επόμενο
        return right;
    }

    public void setRight(tqNode r){     //θέτει τον επόμενο
        right = r;
    }

}
```

Σύμφωνα με την συζήτηση που προηγήθηκε,η δομή μας αποτελείται από κόμβους με τα παραπάνω χαρακτηριστικά οι οποίοι συμμετέχουν σε δύο ειδών λίστες.Μια απλή κυκλική στην οποία συμμετέχουν οι ρίζες των δένδρων και μία ή περισσότερες διπλές μη κυκλικές που αποτελούνται από τον εκάστοτε πατέρα και τα παιδιά του.



(α)



(β)

Σχήμα 3.4: (α) στιγμιότυπο δάσους που αποτελείται από δύο δένδρα-ρίζες συνδεδεμένα σε απλή κυκλική λίστα,(β) δένδρο με έναν πατέρα και δύο παιδιά.

Στο σχήμα 3.4 παρατηρούμε ότι ο δείκτης previous των κόμβων των ριζών είναι πάντα γειωμένος.Στην υλοποίηση που ακολουθεί,αυτός είναι ο έλεγχος που γίνεται για να διαπιστώσουμε αν ένας κόμβος είναι ρίζα ή όχι.Επίσης,παρατηρούμε οτι το τελευταίο παιδί της λίστας πατέρα-παιδιών,δείχνει στον εαυτό του.

Κώδικας 3.2: Thin Heap

```
public class thinHeap{

   protected tqNode minItemNode;
   protected comparator c;
   protected int nofItems;
   protected int nofTrees;

   public thinHeap(){}

   public thinHeap(comparator com){        //μέθοδος κατασκευής
      nofItems = 0;
      nofTrees = 0;
      minItemNode = null;
      c = com;
   }


   protected int size(){                  //επιστροφή μεγέθους σωρού
      return nofItems;
   }


   protected int forestSize(){            //επιστροφή αριθμού ριζών-δένδρων
      return nofTrees;
   }


   public void emptyTH(){                  //κένωση σωρού
      nofItems = 0;
      nofTrees = 0;
      minItemNode = null;
      c = null;
   }


   protected void insertTree(tqNode t){    //ένθεση κόμβου στα δεξιά του min
      t.setRight(minItemNode.getRight()); //H root list είναι
```

```
        minItemNode.setRight(t);          //singly-linked circular
        t.setPrevious(null);              //ο δείκτης right δείχνει στον επόμενο
        nofTrees++;                       //ο previous είναι γειωμένος
    }


    protected tqNode mergeTrees(tqNode v,tqNode w){ //συνένωση δένδρων με ρίζες
v και w
        if(v == null && w == null)
            return null;

        else if(v == null)
            return w;

        else if(w == null)
            return v;

        else if(c.less(w.getKey(),v.getKey())){   //w.key < v.key
            if(w.getLeft() == null){              //αν ο w δεν έχει παιδιά
                w.setLeft(v);
                v.setPrevious(w);                 //κάνε τον v πρώτο παιδί του w
                v.setRight(v);
            }
            else{
                v.setPrevious(w);          //αν ο w έχει παιδιά,κάνουμε τον v πρώτο του παιδί
                v.setRight(w.getLeft());          //εισαγωγή v στη λίστα παιδιών του w
                w.getLeft().setPrevious(v);       //η οποία είναι doubly-linked
                w.setLeft(v);                     //non-circular list
            }

            w.incRank();
            return w;
        }

        else{
            if(v.getLeft() == null){
                v.setLeft(w);
                w.setPrevious(v);
                w.setRight(w);
            }
            else{
                w.setPrevious(v);
                w.setRight(v.getLeft());
                v.getLeft().setPrevious(w);
```

```
            v.setLeft(w);
        }

        v.incRank();
        return v;
    }

} //end of mergeTrees


public void cutRoot(tqNode v){          //αποκοπή κόμβου v από την root list
    tqNode p = minItemNode,t = minItemNode.getRight();

    while(t != v){                  //σαρώνουμε την root list μέχρι
        p = t;                      //τον κόμβο που θα αποκοπεί
        t = t.getRight();
    }                               //μετά το τέλος της while,
                                    //ο p κρατάει τον previous του v
    p.setRight(v.getRight());
    return;
}


public tqNode insert(int i){            //εισαγωγή νέου κόμβου στον σωρό
    tqNode t = new tqNode(i);

    if(nofItems == 0){
        minItemNode = t;
        t.setPrevious(null);
        nofItems++;
        nofTrees++;
        return t;
    }

    else
        insertTree(t);
    nofItems++;

    if(c.less(t.getKey(),minItemNode.getKey()))
        minItemNode = t;
    return t;
}
```

```
public tqNode joinCircular(tqNode x,tqNode y){
   if(x == null){
      tqNode t = y;                  //μετατρέπει την διπλά συνδεδεμένη
      while(t.getRight() != t)     //μη κυκλική των παιδιών
         t = t.getRight();           //σε απλά συνδεδεμένη κυκλική
      t.setRight(y);                //και την συνενώνει με την λίστα των ριζών
      return y;
   }

   else if(y == null)
      return x;

   tqNode xend = x,yend = y;

   while(xend.getRight() != x)
      xend = xend.getRight();


   while(yend.getRight() != yend)
      yend = yend.getRight();



   xend.setRight(y);
   yend.setRight(x);

   return x;
}


public void meldTH(thinHeap T){    //συνένωση δύο thin σωρών
   if(T.size() == 0)
      return;

   tqNode Tstart = T.getMinNode();
   nofTrees += T.forestSize();
   nofItems += T.size();
   tqNode xend = minItemNode,yend = Tstart;
   while(xend.getRight() != minItemNode)
      xend = xend.getRight();
   while(yend.getRight() != Tstart)
      yend = yend.getRight();
   xend.setRight(Tstart);
   yend.setRight(minItemNode);
```

```
    if(c.less(Tstart.getKey(),minItemNode.getKey()))
        minItemNode = Tstart;
}


public int Min(){                   //επιστρέφει την τιμή του ελαχίστου key
    if(nofItems == 0)
        return -1;
    return minItemNode.getKey();
}


public tqNode getMinNode(){         //επιστρέφει τον κόμβο με το ελάχιστο key
    return minItemNode;
}


public tqNode removeMin(){          //αποκοπή κόμβου ελαχίστου

    if(nofItems == 0)
        return null;

    tqNode i = minItemNode;

    if(--nofItems == 0){
        minItemNode = null;
        nofTrees--;
        return i;
    }

    tqNode cursor,sibling = minItemNode.getRight();
    tqNode child = minItemNode.getLeft();
    tqNode[] A = new tqNode[((int)(Math.log(nofItems)/Math.log(2.0)))+1];
    int j;

    if(sibling != minItemNode){         //υπάρχει και άλλος κόμβος
        cutRoot(minItemNode);
    }
    else
        sibling = null;
    minItemNode.setRight(null);
    minItemNode.setLeft(null);
    minItemNode.setPrevious(null);
```

```
if(child != null){                //αν ο minNode έχει παιδιά
   child.setPrevious(null);       //αποκοπή του πρώτου παιδιού από τον πατέρα
   if(child.isThin()){            //κάνε το παιδί normal αν είναι thin
      child.decRank();
      child.unThin();             //set thin = false
   }
   cursor = child.getRight();

   while(cursor.getRight() != cursor){ //Η λίστα των παιδιών είναι
      if(cursor.isThin()){         //διπλή μη κυκλική και ο τελευταίος
         cursor.decRank();         //κόμβος δείχνει δεξιά στον εαυτό του
         cursor.unThin();          //κάνουμε τα παιδιά normal αν είναι thin
      }
      cursor.setPrevious(null);
      cursor = cursor.getRight();
   }

   if(cursor.isThin()){
         cursor.decRank();
         cursor.unThin();
   }
   cursor.setPrevious(null);


   sibling = joinCircular(sibling,child);  //συνένωση λίστας παιδιών με ριζών
}

cursor = sibling.getRight();
sibling.setRight(sibling);
A[sibling.getRank()] = sibling;

while(cursor != sibling){
   int r = cursor.getRank();
   tqNode next = cursor.getRight();
   cursor.setRight(cursor);
   while(A[r] != null){
      cursor = mergeTrees(A[r],cursor);
      cursor.setRight(cursor);
      A[r++] = null;
   }
   A[r] = cursor;
   cursor = next;
}
```

```
        for(j=0;A[j]==null;j++);
            minItemNode = A[j++];
        nofTrees = 1;
        for(;j<A.length;j++)
            if(A[j] != null){
                insertTree(A[j]);
                if(c.less(A[j].getKey(), minItemNode.getKey()))
                    minItemNode = A[j];
                A[j] = null;
            }
        return i;
    }


    public tqNode decreaseKey(tqNode v,int p){   //μείωση του key του v κατά p
        if(p == 0)
            return v;

        v.setKey(v.getKey() - p);
        tqNode y = v.getPrevious();            //y left sibling or parent of v

        if(v.getPrevious() != null && v.getPrevious().getLeft() == v){   //έλεγχος αν v
== first child
            if(c.less(v.getPrevious().getKey(),v.getKey()) ||
              c.equal(v.getPrevious().getKey(),v.getKey()))           //με key>=parent's key
                return v;
        }

        else if(v.getPrevious() == null){           //έλεγχος αν v == root
            if(c.less(v.getKey(), minItemNode.getKey())){
                minItemNode = v;
                return v;
            }
            return v;
        }

        //αν v δεν είναι root ή πρώτο παιδί με key>=parent's key
        if(v.getPrevious().getLeft() == v){     //αν v == first child
            y.setThin();                    //κάνουμε τον πατερα thin
            if(v.getRight() == v)              //και αν v είναι το μοναδικό παιδί
                y.setLeft(null);              //αποκοπή v από τον πατέρα του
            else{
                y.setLeft(v.getRight());       //αν δεν είναι το μοναδικό παιδί
```

```
            v.getRight().setPrevious(y);    //αποκοπή v από τον πατέρα του
        }
    }

    else{                           //αλλιώς αν v != first child
        if(v.getRight() == v)           //αν είναι τελευταίο παιδί
            y.setRight(y);              //κάνουμε το previous του τελευταίο
        else{
            y.setRight(v.getRight());       //αποκοπή από τη λίστα παιδιών
            v.getRight().setPrevious(y);
        }
    }

    v.setPrevious(null);
    if(v.isThin()){                 //και αν είναι thin το κανουμε normal
        v.decRank();
        v.unThin();
    }
    insertTree(v);
    if(c.less(v.getKey(), minItemNode.getKey()))
        minItemNode = v;
    repairStep(y);              //ελέγχουμε για νέα παράβαση στον y = v.getPrevious()
    return v;
}//end of decreaseKey



public void repairStep(tqNode y){       //διόρθωση της δομής
  //case1,αν y έχει rank +2 από του next sibling του
  //ή έχει rank 1 και κανένα sibling(και δεν είναι και root)
  if(((y.getRank() - y.getRight().getRank() == 2) ||
        (y.getRank() == 1 && y.getRight() == y)) && y.getPrevious() != null){
      //case 1a,y is thin
      if(y.isThin()){
          y.decRank();
          y.unThin();
          y = y.getPrevious();
          repairStep(y);
      }

      //case 1b,y is normal
      else{
          tqNode w = y.getLeft();         //w πρώτο παιδί του y
```

```
            if(w.getRight() != w){          //αν ο y έχει κι άλλα παιδιά
               y.setLeft(w.getRight());
               w.getRight().setPrevious(y);
            }
            else
               y.setLeft(null);            //αν w ήταν το μοναδικό παιδί του y

            if(y.getRight() == y)           //αν y δεν έχει δεξί αδελφό
               w.setRight(w);
            else{
               w.setRight(y.getRight()); //αποκόπτουμε το w από τη λίστα παιδιών του y
               y.getRight().setPrevious(w); //και το κάνουμε right sibling του y
            }
            w.setPrevious(y);
            y.setRight(w);
            y.setThin();

         }
      }

      //case 2,αν y έχει rank +3 από του πρώτου παιδιού του
      //ή έχει rank 2 και κανένα παιδί
      else if(((y.getLeft() != null) && (y.getRank() - y.getLeft().getRank() == 3)) ||
            (y.getRank() == 2 && y.getLeft() == null)){

         y.setRank(y.getRank() - 2);        //μειωνουμε το rank του y κατα 2 ώστε να
γίνει normal
         y.unThin();
         tqNode z = y.getPrevious();

         if(z.getLeft() == y){              //αν z πατέρας του y
            z.setThin();
            if(y.getRight() != y){          //αν y δεν είναι το μοναδικό παιδί του z
               z.setLeft(y.getRight());     //αποκοπή του y
               y.getRight().setPrevious(z);
            }
            else                        //αν y είναι μοναδικό παιδί του z
               z.setLeft(null);
         }

         else{                          //αλλιώς z left sibling του y
            if(y.getRight() != y){          //αν y δεν είναι τελευταίο παιδί
               z.setRight(y.getRight());   //αποκοπή y
               y.getRight().setPrevious(z);
```

```
            }
            else                        //αλλιώς,αν είναι τελευταιο παιδί
               z.setRight(z);
         }

         y.setPrevious(null);
         y.setRight(null);

         insertTree(y);                 //εισαγωγή στη 2η θέση της root list
         y = z;
         repairStep(y);                 //έλεγχος για πιθανή νέα παράβαση
      }

      //case 3,αν ο y είναι thin root τον κάνουμε normal
      else if(y.isThin() && y.getPrevious() == null){
         y.decRank();
         y.unThin();
      }

      return;

   }//end of repairStep


   public void delete(tqNode v){          //διαγραφή κόμβου v
      decreaseKey(v, v.getKey() - (Min()-1));      //αναγωγή σε διαγραφή ελαχίστου
      removeMin();                        //αφού μηδενίσουμε το key του
      return;                             //προς διαγραφή κόμβου
   }

}
```

3.1.3 Ανάλυση Πολυπλοκότητας

Η ανάλυση επιμερισμένου κόστους των thin heaps είναι παρεμφερής με αυτή του σωρού Fibonnaci,όπου στην θέση των σημαδεμένων κόμβων έχουμε τους thin κόμβους.Χρησιμοποιούμε την τεχνική του δυναμικού.Ως δυναμικό **Φ** της δομής ορίζεται το πλήθος των δένδρων διατάξεως σωρού συν δύο φορές το πλήθος των thin κόμβων.Ως κόστος επιμερισμού μιας πράξεως,λογίζεται ο πραγματικός χρόνος συν

την μεταβολή του δυναμικού που προκαλεί.Ο συνολικός πραγματικός χρόνος μια αναμεμειγμένης ακολουθίας πράξεων επί ενός αρχικά άδειου σωρού ισούται το πολύ με το άθροισμα των επιμερισμένων χρόνων των πράξεων.

Θεώρημα 3.1 Έστω μια αναμεμειγμένη ακολουθία πράξεων επί ενός αρχικώς άδειου thin heap.Αν χρεώσουμε κάθε διαγραφή και διαγραφή ελαχίστου σε έναν σωρό με n στοιχεία O(logn) επιμερισμένο χρόνο και κάθε άλλη πράξη με O(1) επιμερισμένο χρόνο,τότε ο συνολικός χρόνος ισούται,το πολύ,με τον συνολικό επιμερισμένο χρόνο.

Το πραγματικό και το επιμερισμένο κόστος της εύρεσης ελαχίστου,ενθέσεως στοιχείου και συνένωσης σωρών είναι O(1):Κάθε ένθεση αυξάνει τον αριθμό των δένδρων,άρα και το δυναμικό,κατά ένα.Οι άλλες δύο πράξεις δεν μεταβάλλουν το δυναμικό.Αν χρεώσουμε μια μονάδα χρόνου σε κάθε βήμα συνένωσης,ο επιμερισμένος χρόνος μιας διαγραφής ελαχίστου ισούται το πολύ με την μέγιστη τάξη κόμβου επί μια σταθερά,διότι κάθε βήμα συνένωσης μειώνει το δυναμικό κατά ένα και ο υπολοιπος χρόνος που ξοδεύεται σε μια διαγραφή ελαχίστου είναι φραγμένος από την μέγιστη τάξη πολλαπλασιαζόμενη με μια σταθερά.Σύμφωνα με το Λήμμα 3.1,η μέγιστη τάξη σε έναν thin heap n κόμβων είναι το πολύ $\log_{\varphi} n$,συνεπώς το επιμερισμένο κόστος μιας διαγραφής ελαχίστου είναι O(logn).

Για την ανάλυση της μείωσης προτεραιότητας,χρεώνουμε μια μονάδα χρόνου για κάθε βήμα επιδιόρθωσης.Κάθε τέτοιο βήμα (εκτός από το τελευταίο) κάνει έναν thin κόμβο normal και μπορεί ενδεχομένως να δημιουργήσει ένα νέο δένδρο,μειώνοντας το δυναμικό κατά ένα.Η αρχική αποκοπή και το τελευταίο βήμα επιδιόρθωσης κοστίζουν O(1) και αυξάνουν το δυναμικό το πολύ κατά τρία (μέγιστο ένα για την αποκοπή και δύο για το βήμα),συνεπώς η μείωση προτεραιότητας έχει σταθερό επιμερισμένο κόστος. Η ανάλυση χειρότερης περιπτώσεως για την πράξη της μείωσης προτεραιότητας είναι O(logn) διότι κάθε παράβαση συμβαίνει σε κόμβο υψηλότερης τάξεως.

Τέλος,το επιμερισμένο κόστος μιας διαγραφής στοιχείου είναι O(logn) καθώς πρόκειται στην ουσία για μια μείωση προτεραιότητας ακολουθούμενη από μια διαγραφή ελαχίστου.

3.2 Thick Heap

Αποτελεί μια ακόμα παραλλαγή του σωρού Fibonacci.Σε σχέση με τον thin heap,μειώνουμε τον αριθμό των παιδιών ανά κόμβο.Αυτό επιτυγχάνεται με την χρησιμοποποίηση μιας νέας δομής ως δομικό στοιχείο,του thick tree. Ένας thick heap είναι ένα αδιάτακτο σύνολο (δάσος) από,ξένα ως προς τα στοιχεία που φέρουν,thick trees τα οποία τηρούν την αμετάβλητη συνθήκη σωρού ελαχίστου.

Thick Tree. Ο ορισμός ενός thick tree διαφέρει από αυτόν του thin tree μόνο στην ιδιότητα (2).Συγκεκριμένα,ένα thick tree είναι ένα δυωνυμικό δένδρο στο οποίο οποιοσδήποτε κόμβος του μπορεί να «χάσει» το πρώτο-αριστερότερό του παιδί και το αντίστοιχο υποδένδρο.Έχει τις εξής ιδιότητες:

(1) Τα παιδιά ενός κόμβου με k παιδιά,είναι τάξεως k-1,k-2,…,0 από αριστερά προς τα δεξιά.
(2) Έστω ένας κόμβος με k παιδιά.Αν έχει τάξη k τον αποκαλούμε normal,ενώ αν έχει τάξη k-1 καλείται thick.
(3) Η ρίζα ενός thick tree είναι πάντα normal.

Λήμμα 3.2. Ένας κόμβος με k παιδιά σε ένα thick tree έχει τουλάχιστον 2^k απογόνους,συμπεριλαμβανομένου και του εαυτού του.

3.2.1 Περιγραφή Πράξεων

Όλες οι πράξεις εκτός από την μείωση προτεραιότας υλοποιούνται όπως ακριβώς στην thin heap,επομένως θα αρκεστούμε στην περιγραφή της πράξης αυτής.Οι διαφορές εντοπίζονται στο βήμα επιδιόρθωσης.

Μείωση Προτεραιότητας. Αρχικά,θέτουμε στον κόμβο x τη νέα του προτεραιότητα.Αν ο x είναι πρώτο παιδί και η προτεραιότητά του παραμένει μικρότερη ή ίση από αυτήν του πατέρα του,τότε η πράξη έχει ολοκληρωθεί.Αν ο x είναι ρίζα,τότε συγκρίνουμε τη νέα του προτεραιότητα με το ελάχιστο στοιχείο της λίστας των ριζών και στην περίπτωση που αυτή είναι μικρότερη θέτουμε το x ως νέο ελάχιστο.Σε περίπτωση που δεν ισχύει τίποτα από τα παραπάνω,θέτουμε y ίσο με τον αριστερό αδελφό του x ή τον πατέρα του,στην περίπτωση που ο x είναι πρώτο παιδί.Αποκόπτουμε τον x από την λίστα των παιδιών-και κατά συνέπεια και από τον πατέρα του-και τον ενθέτουμε στο δάσος σαν νέα ρίζα (μαζί με το υποδένδρο του),στην πρώτη ή δεύτερη θέση,ανάλογα με το αν η προτεραιότητά του είναι μικρότερη ή όχι από αυτή του τρέχοντος ελαχίστου.Πριν το ενθέσουμε,το κάνουμε normal αν δεν είναι.

Η αποκοπή του x,ενδέχεται να προκαλέσει στον y παραβίαση των ιδιοτήτων (1),(2) ή (3) που περιγράψαμε κατά τον ορισμό του thick tree.Επαναλαμβάνουμε το παρακάτω βήμα επιδιόρθωσης της δομής,το οποίο διορθώνει την παράβαση στον y αλλά ενδέχεται να προκληθεί νέα παράβαση στον αριστερό αδελφό ή τον πατέρα του y,μέχρις ότου δεν υπάρχει καμία παράβαση:

Βήμα Επιδιόρθωσης (Repair Step)

Περίπτωση 1. Παραβίαση του (1):Ο κόμβος y έχει τάξη κατά δύο μεγαλύτερη από αυτή του δεξιού αδελφού του,ή έχει τάξη 1 και κανέναν δεξί αδελφό.

1α. Ο κόμβος y είναι normal.Μειώνουμε την τάξη του κατά ένα, διορθώνοντας την παράβαση και κάνοντας τον thick.Αντικαθιστούμε τον y με τον αριστερό του αδελφό ή τον πατέρα του αν είναι πρώτο παιδί και ελέγχουμε για παράβαση στον νέο y.

1β. Ο κόμβος y είναι thick.Έστω w το πρώτο παιδί του y.Το αποκόπτουμε από τον y και το εισάγουμε ως αριστερό αδελφό του y,μειώνοντας παράλληλα την τάξη του y κατά ένα.Αυτό αφήνει τον κόμβο y thick,αλλά επιδιορθώνει την παράβαση χωρίς να προκαλεί νέα.

Περίπτωση 2. Παραβίαση του (2):Ο κόμβος y έχει τάξη κατά δύο μεγαλύτερη από αυτή του πρώτου παιδιού του,ή έχει τάξη ένα και κανένα παιδί.Μειώνουμε την τάξη του y κατά ένα,κάνοντάς τον normal.Αν είναι ρίζα,η πράξη έχει ολοκληρωθεί. Διαφορετικά,έστω w ο δεξιός αδελφός του y.Εφαρμόζουμε μια εκ των παρακάτω δύο περιπτώσεων:

2α. Ο κόμβος w είναι thick.Αυξάνουμε την τάξη του κατά ένα και ανταλλάσουμε τους κόμβους y και w στις λίστες στις οποίες περιέχονται.Αυτό έχει ως συνέπεια να γίνουν τόσο ο y όσο και ο w normal και να διορθωθεί η παράβαση χωρίς να προκαλείται νέα.

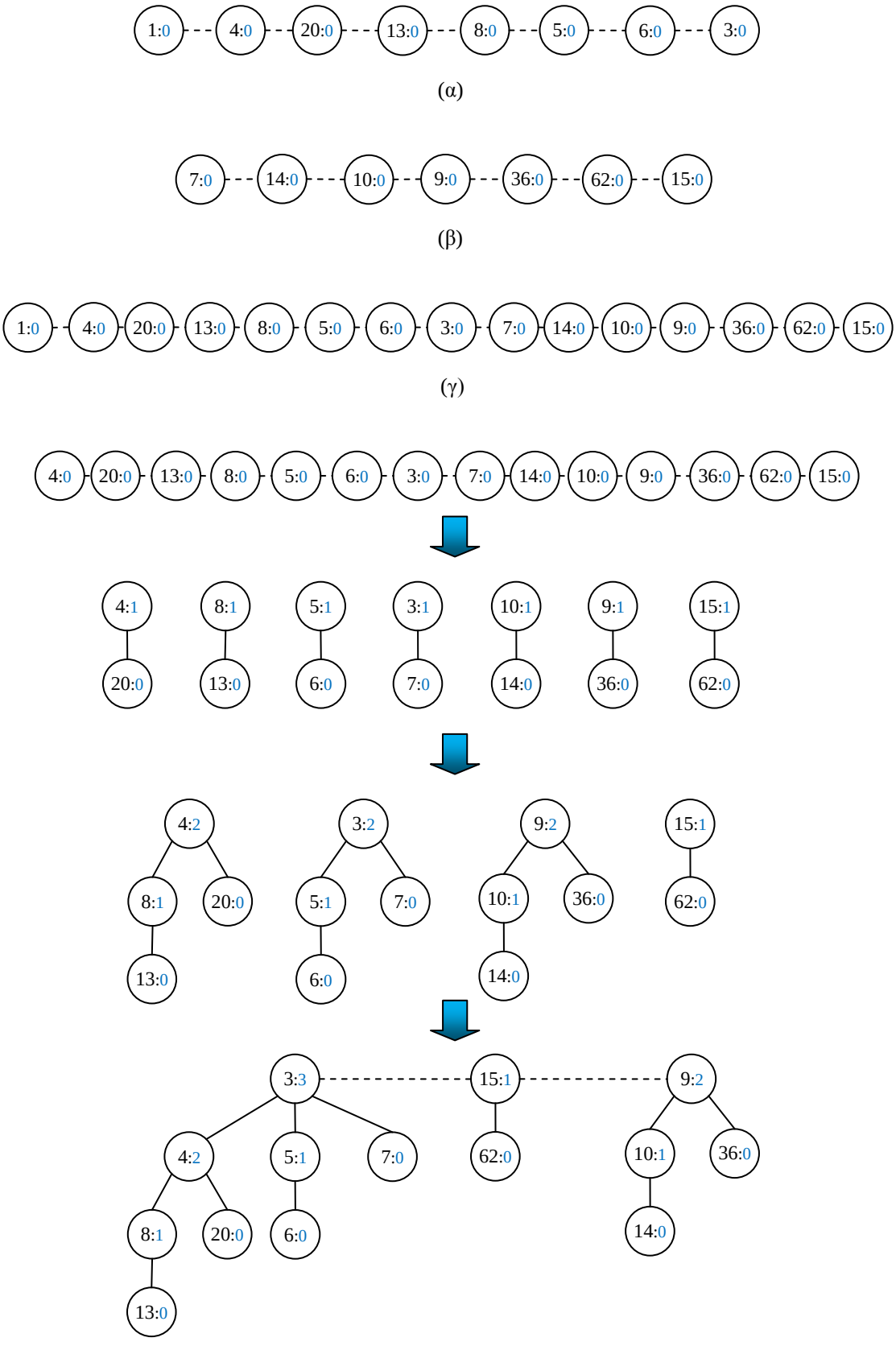
2β. Ο κόμβος w είναι normal.Έστω z ο αριστερός αδελφός του y ή ο πατέρας του,αν ο y είναι πρώτο παιδί.Αν ο y έχει προτεραιότητα μεγαλύτερη ή ίση από αυτή του w,αποκόπτουμε τον y από την λίστα των παιδιών στην οποία βρίσκεται και τον κάνουμε πρώτο παιδί του w (που μέχρι τώρα ήταν ο δεξιός αδελφός του).Ο κόμβος w πλέον γίνεται thick.Διαφορετικά,αν ο y έχει προτεραιότητα μικρότερη από αυτή του w,αποκόπτουμε τον w από την λίστα των παιδιών στην οποία περιέχεται και τον κάνουμε πρώτο παιδί του y.Ο κόμβος y πλέον γίνεται thick.Τέλος,θέτουμε y = z και ελέγχουμε για παράβαση στο νέο y,τον κόμβο δηλαδή που βρισκόταν στα αριστερά του προηγούμενου y.

Περίπτωση 3. Παραβίαση του (3):Ο κόμβος y είναι thick ρίζα.Κάνουμε τον y normal,αυξάνοντας την τάξη του κατά ένα.

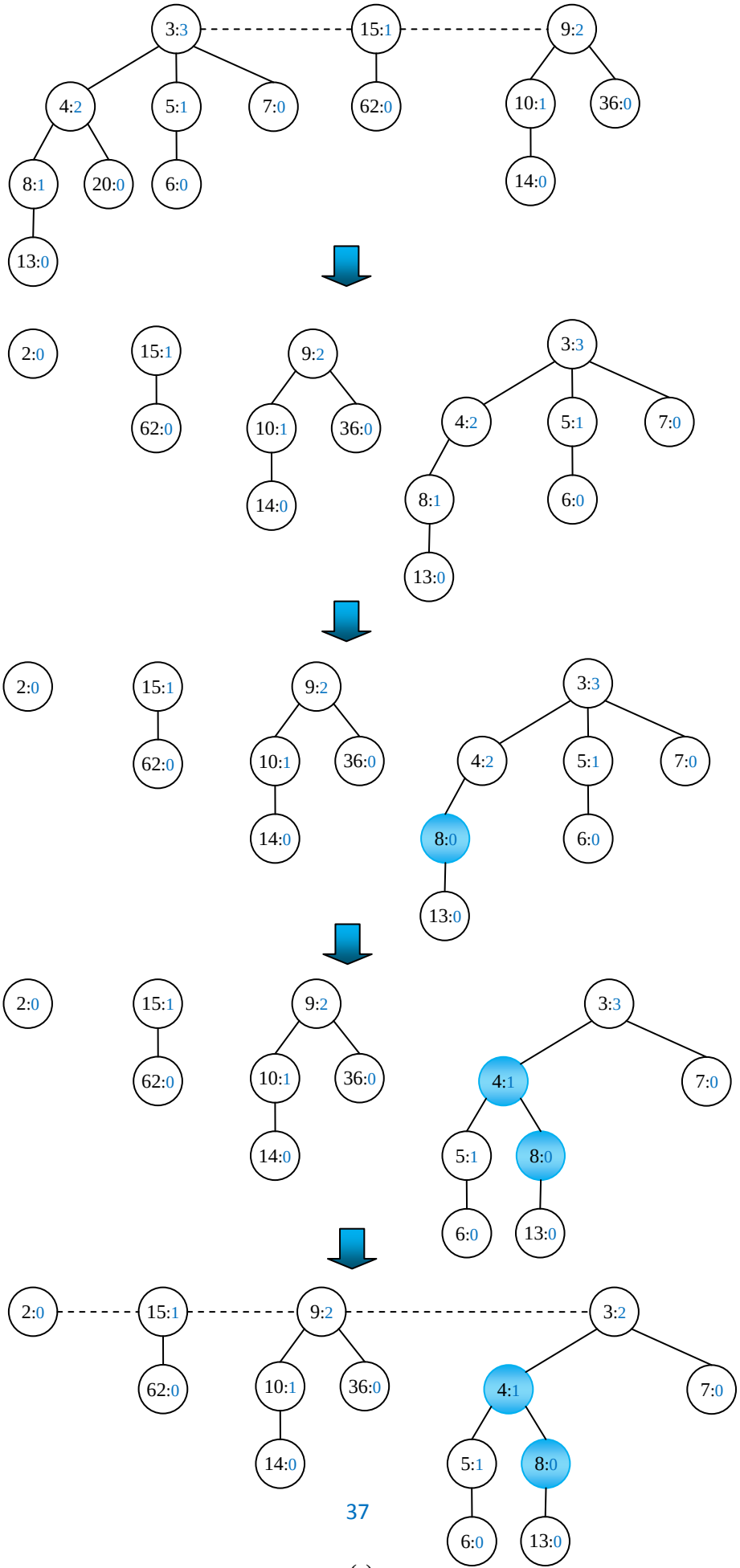
Όπως μπορούμε να παρατηρήσουμε στο σχήμα 3.5,η ένθεση στοιχείων και η απόσβεση ελαχίστου σε έναν thick heap λειτουργεί ακριβώς όπως στους thin heaps και τον σωρό Fibonacci.Στο σχήμα 3.6 (ε) παρουσιάζεται ένα στιγμιότυπο της περίπτωσης 1α.Μειώνουμε την προτεραιότητα του 20 σε 2,αυτό έχει σαν αποτέλεσμα την αποκοπή του 2 και την ένθεσή του στο δάσος σαν νέο ελάχιστο. Κατόπιν, εξετάζουμε τον αριστερό αδελφό του 2 που είναι το 8.Παρατηρούμε ότι έχει τάξη ένα και κανέναν δεξί αδελφό και καθώς είναι και normal έχουμε παραβίαση του 1α.Μειώνουμε την τάξη του κατά ένα κι έτσι λοιπόν γίνεται thick καθώς έχει τάξη 0

κι ένα παιδί.Έπειτα,ελέγχουμε τον προηγούμενό του,που είναι ο κόμβος με προτεραιότητα 4,για ενδεχόμενη παράβαση.Παρατηρούμε οτι ο 4 έχει τάξη κατά δύο μεγαλύτερη από αυτή του πρώτου παιδιού του,που είναι ο 8,άρα έχουμε παραβίαση του (2).Μειώνουμε την τάξη του 4 κατά ένα κι εξετάζουμε τον δεξί αδελφό του,που είναι ο 5 ο οποίος είναι normal,άρα αναγόμαστε στην περίπτωση 2β.Συγκρίνουμε τις προτεραιότητες του 4 και του δεξιού αδελφού του ο οποίος είναι ο 5. Αφού 4<5 αποκόπτουμε τον w=5 και τον κάνουμε πρώτο παιδί του y=4,επομένως ο 4 γίνεται thick.Στην συνέχεια,εξετάζουμε για νέα παράβαση τον κόμβο που βρίσκεται στα αριστερά του 4 και είναι ο πατέρας του με προτεραιότητα 3.Παρατηρούμε οτι έχει τάξη κατά 2 μεγαλύτερη από αυτή του πρώτου του παιδιού.Μειώνουμε την τάξη του κατά ένα και καθώς είναι ρίζα η πράξη σ'αυτό το σημείο ολοκληρώνεται.

Στην συνέχεια,στο σχήμα 3.7 (στ),μειώνουμε την προτεραιότητα του 7 σε 1.Αυτό έχει σαν αποτέλεσμα την αποκοπή του 1 και την ένθεσή του στο δάσος ως νέο ελάχιστο.Κατόπιν,εξετάζουμε τον κόμβο στα αριστερά του,ο οποίος είναι ο 4.Παρατηρούμε οτι έχει τάξη 1,κανέναν δεξί αδελφό και είναι thick,άρα βρισκόμαστε στην περίπτωση 1β.Αποκόπτουμε το πρώτο παιδί του 4,δηλαδή τον κόμβο 5,και τον κάνουμε αριστερό αδελφό του 4 μειώνοντας παράλληλα την τάξη του 4 κατά ένα,ο ποίος παραμένει thick αλλά διορθώνεται η παράβαση χωρίς να δημιουργείται νέα.

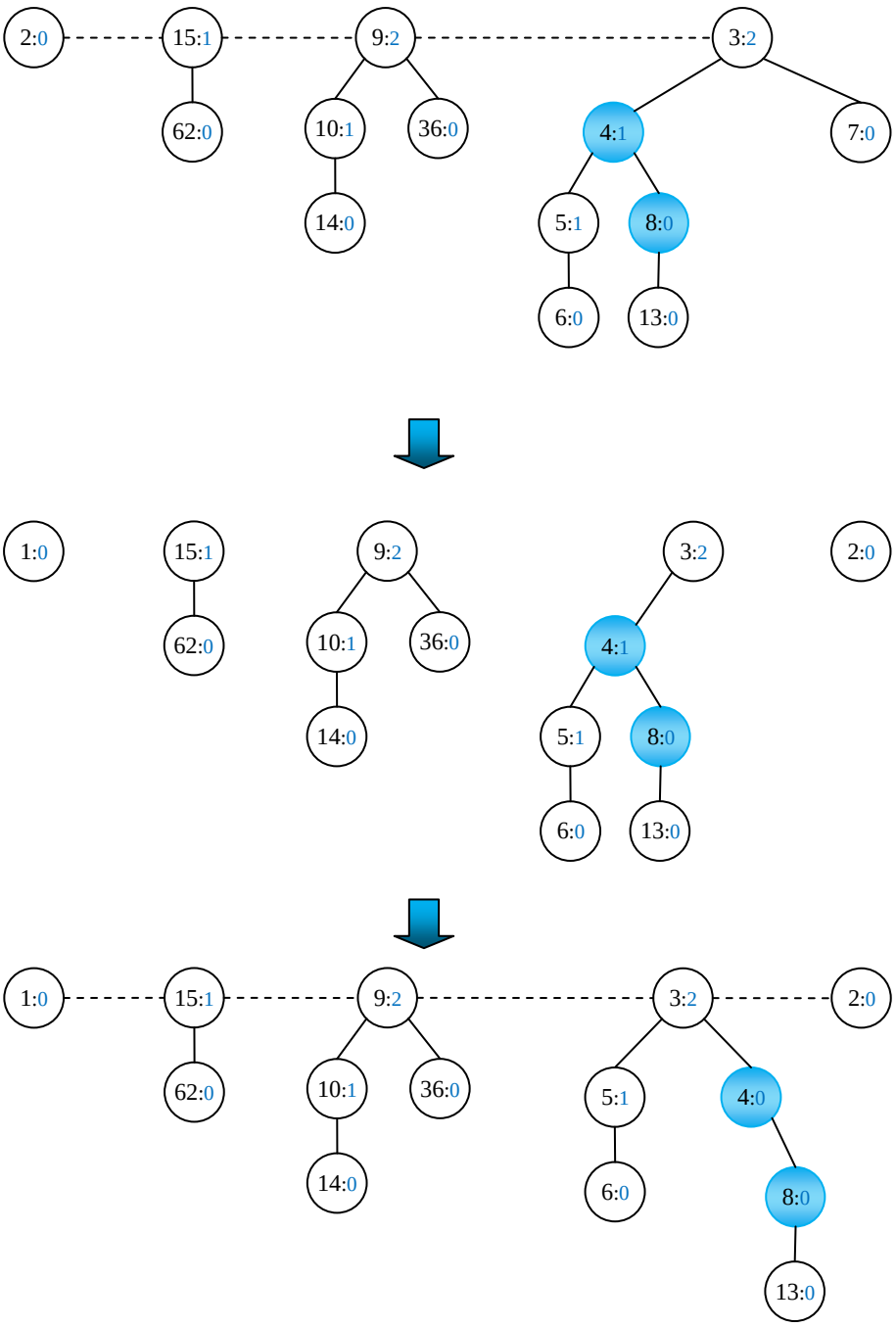


Σχήμα 3.5: Στιγμιότυπα πράξεων thick heap: (α)-(β) δύο σωροί,ως αποτέλεσμα διαδοχικών ενθέσεων,(γ) η συνένωσή τους,(δ) απόσβεση ελαχίστου.


3:3
15:1
9:2
4:2
5:1
7:0
62:0
10:1
36:0
8:1
20:0
6:0
14:0
13:0
2:0
15:1
9:2
3:3
62:0
10:1
36:0
4:2
5:1
7:0
14:0
8:1
6:0
13:0
2:0
15:1
9:2
3:3
62:0
10:1
36:0
4:2
5:1
7:0
14:0
8:0
6:0
13:0
2:0
15:1
9:2
3:3
62:0
10:1
36:0
4:1
7:0
14:0
5:1
8:0
6:0
13:0
2:0
15:1
9:2
3:2
62:0
10:1
36:0
4:1
7:0
14:0
5:1
8:0
6:0
13:0

(ε)

Σχήμα 3.6. (συνέχεια) (ε) μείωση προτεραιότητας του 20 σε 2.



(στ)

Σχήμα 3.7. (συνέχεια) (στ) μείωση προτεραιότητας του 7 σε 1.

3.2.2 Υλοποίηση

Δομικό στοιχείο του thick heap αποτελεί η κλάση κόμβος thickNode,η οποία περιλαμβάνει πληροφορία για την τάξη ενός κόμβου,την προτεραιότητά του,το αν είναι thick ή όχι καθώς επίσης και μεθόδους για τον χειρισμό των τριών δεικτών(σε πρώτο παιδί,δεξί αδελφό και αριστερό αδελφό ή πατέρα).

Κώδικας 3.3: Κόμβος Thick Heap

```
public class thickNode {

   private int rank,key;                          //Ο δείκτης left δείχνει στο πρώτο παιδί
   private thickNode left,previous,right;     //Ο previous στον αριστερό αδερφό ή στον
   private boolean thick;                          //πατέρα αν κόμβος μας είναι πρώτο παιδί
                                                  //Ο right στον δεξί αδερφό ή στην επόμενη ρίζα
   public thickNode() {}                      //Η μεταβλητή thick είναι αληθής αν ο
                                                  //κόμβος μας είναι thick
   public thickNode(int k) {
      setKey(k);
      setLeft(null);
      setRight(this);
      setPrevious(this);
      rank = 0;
      thick = false;
   }

   public int getKey(){
      return key;
   }

   public void setKey(int k){
      key = k;
   }

   public int getRank(){
      return rank;
   }

   public void setRank(int i){
      rank = i;
   }

   public void incRank(){
      rank++;
   }
```

```
    public void decRank(){
        rank--;
    }

    public boolean isThick(){
        return thick;
    }

    public void setThick(){
        thick = true;
    }

    public void unThick(){
        thick = false;
    }

    public thickNode getPrevious(){
        return previous;
    }

    public void setPrevious(thickNode p){
        previous = p;
    }

    public thickNode getLeft(){
        return left;
    }

    public void setLeft(thickNode l){
        left = l;
    }

    public thickNode getRight(){
        return right;
    }

    public void setRight(thickNode r){
        right = r;
    }

}
```

Για την υλοποίηση του thick heap ακολουθούμε ακριβώς τις ίδιες αρχές με τον thin heap.Οι διαφορές εντοπίζονται στην χρήση thick κόμβων και στην διαφορετική υλοποίηση της μεθόδου decreaseKey.

Κώδικας 3.4: Thick Heap

```
public class thickHeap {

   protected thickNode minItemNode;
   protected comparator c;
   protected int nofItems;
   protected int nofTrees;

   public thickHeap(){}                  //απλός constructor

   public thickHeap(comparator com){       //μέθοδος κατασκευής
      nofItems = 0;
      nofTrees = 0;
      minItemNode = null;
      c = com;
   }


   protected int size(){                 //επιστροφή μεγέθους σωρού
      return nofItems;
   }


   protected int forestSize(){           //επιστροφή αριθμού ριζών-δένδρων
      return nofTrees;
   }


   public void emptyTH(){                //κένωση σωρού
      nofItems = 0;
      nofTrees = 0;
      minItemNode = null;
      c = null;
   }

   protected void insertTree(thickNode t){    //ένθεση κόμβου στα δεξιά του min
      t.setRight(minItemNode.getRight());    //H root list είναι
      minItemNode.setRight(t);               //singly-linked circular
      t.setPrevious(null);                   //ο δείκτης right δείχνει στον επόμενο κόμβο
```

```
        nofTrees++;                         //o previous είναι γειωμένος
    }

    protected thickNode mergeTrees(thickNode v,thickNode w){ //συνένωση δένδρων
με ρίζες v και w
        if(v == null && w == null)
            return null;

        else if(v == null)
            return w;

        else if(w == null)
            return v;

        else if(c.less(w.getKey(),v.getKey())){   //w.key < v.key
            if(w.getLeft() == null){              //αν ο w δεν έχει παιδιά
                w.setLeft(v);
                v.setPrevious(w);                 //κάνε τον v πρώτο παιδί του w
                v.setRight(v);
            }
            else{
                v.setPrevious(w);           //αν ο w έχει παιδιά,κάνουμε τον v πρώτο του παιδί
                v.setRight(w.getLeft());          //εισαγωγή v στη λίστα παιδιών του w
                w.getLeft().setPrevious(v);       //η οποία είναι doubly-linked
                w.setLeft(v);                     //non-circular list
            }

            w.incRank();
            return w;
        }

        else{
            if(v.getLeft() == null){
                v.setLeft(w);
                w.setPrevious(v);
                w.setRight(w);
            }
            else{
                w.setPrevious(v);
                w.setRight(v.getLeft());
                v.getLeft().setPrevious(w);
                v.setLeft(w);
            }
```

```
            v.incRank();
            return v;
        }

    } //end of mergeTrees


    public void cutRoot(thickNode v){        //αποκοπή κόμβου v από την root list
        thickNode p = minItemNode,t = minItemNode.getRight();

        while(t != v){                    //σαρώνουμε την root list μέχρι
            p = t;                        //τον κόμβο που θα αποκοπεί
            t = t.getRight();             //μετά το τέλος της while,
        }                                 //ο p κρατάει τον previous του v

        p.setRight(v.getRight());
        return;
    }


    public thickNode insert(int i){          //εισαγωγή νέου κόμβου στον σωρό
        thickNode t = new thickNode(i);

        if(nofItems == 0){
            minItemNode = t;
            t.setPrevious(null);
            nofItems++;
            nofTrees++;
            return t;
        }

        else
            insertTree(t);
        nofItems++;

        if(c.less(t.getKey(),minItemNode.getKey()))
            minItemNode = t;
        return t;
    }


    public thickNode joinCircular(thickNode x,thickNode y){    //x = root list
minNode, y = first child of children's circular
        if(x == null){
```

```
        thickNode t = y;              //μετατρέπει την διπλά συνδεδεμένη
        while(t.getRight() != t)      //μη κυκλική των παιδιών
            t = t.getRight();         //σε απλά συνδεδεμένη κυκλική
        t.setRight(y);                //και την συνενώνει με την λίστα των ριζών
        return y;
    }

    else if(y == null)
        return x;

    thickNode xend = x,yend = y;

    while(xend.getRight() != x)
        xend = xend.getRight();

    while(yend.getRight() != yend)
        yend = yend.getRight();

    xend.setRight(y);
    yend.setRight(x);

    return x;
}


public void meldTH(thickHeap T){      //συνένωση δύο thick σωρών
    if(T.size() == 0)
        return;

    thickNode Tstart = T.getMinNode();
    nofTrees += T.forestSize();
    nofItems += T.size();
    thickNode xend = minItemNode,yend = Tstart;
    while(xend.getRight() != minItemNode)
        xend = xend.getRight();
    while(yend.getRight() != Tstart)
        yend = yend.getRight();
    xend.setRight(Tstart);
    yend.setRight(minItemNode);
    if(c.less(Tstart.getKey(),minItemNode.getKey()))
        minItemNode = Tstart;
}
```

```
public int Min(){                  //επιστρέφει την τιμή του ελαχίστου key
   if(nofItems == 0)
      return -1;
   return minItemNode.getKey();
}


public thickNode getMinNode(){      //επιστρέφει τον κόμβο με το ελάχιστο key
   return minItemNode;
}


public thickNode removeMin(){       //αποκοπή κόμβου ελαχίστου

   if(nofItems == 0)
      return null;

   thickNode i = minItemNode;

   if(--nofItems == 0){
      minItemNode = null;
      nofTrees--;
      return i;
   }

   thickNode cursor,sibling = minItemNode.getRight();
   thickNode child = minItemNode.getLeft();
   thickNode[] A = new thickNode[((int)(Math.log(nofItems)/Math.log(2.0)))+1];
   int j;

   if(sibling != minItemNode){         //υπάρχει και άλλος κόμβος
      cutRoot(minItemNode);
   }
   else
      sibling = null;
   minItemNode.setRight(null);
   minItemNode.setLeft(null);
   minItemNode.setPrevious(null);

   if(child != null){                 //αν ο minNode έχει παιδιά
      child.setPrevious(null);        //αποκοπή του πρώτου παιδιού από τον πατέρα
      if(child.isThick()){            //κάνε το παιδί normal αν είναι thick
         child.incRank();
         child.unThick();             //set thick = false
```

```
        }
        cursor = child.getRight();

        while(cursor.getRight() != cursor){  //Η λίστα των παιδιών είναι
            if(cursor.isThick()){           //διπλή μη κυκλική και ο τελευταίος
                cursor.incRank();           //κόμβος δείχνει δεξιά στον εαυτό του
                cursor.unThick();           //κάνουμε τα παιδιά normal αν είναι thick
            }
            cursor.setPrevious(null);       //θέτουμε previous=null πριν τα εισάγουμε
στην λίστα ριζών
            cursor = cursor.getRight();
        }

        if(cursor.isThick()){
                cursor.incRank();
                cursor.unThick();
        }
        cursor.setPrevious(null);

        sibling = joinCircular(sibling,child);  //συνένωση λίστας παιδιών με ριζών
    }

    cursor = sibling.getRight();
    sibling.setRight(sibling);
    A[sibling.getRank()] = sibling;

    while(cursor != sibling){
        int r = cursor.getRank();
        thickNode next = cursor.getRight();
        cursor.setRight(cursor);
        while(A[r] != null){
            cursor = mergeTrees(A[r],cursor);
            cursor.setRight(cursor);
            A[r++] = null;
        }
        A[r] = cursor;
        cursor = next;
    }

    for(j=0;A[j]==null;j++);
        minItemNode = A[j++];
    nofTrees = 1;
    for(;j<A.length;j++)
        if(A[j] != null){
```

```
            insertTree(A[j]);
            if(c.less(A[j].getKey(), minItemNode.getKey()))
               minItemNode = A[j];
            A[j] = null;
         }
      return i;
   }


   public thickNode decreaseKey(thickNode v,int p){
      if(p == 0)
         return v;

      v.setKey(v.getKey() - p);
      thickNode y = v.getPrevious();            //y left sibling or parent of v

      if(v.getPrevious() != null && v.getPrevious().getLeft() == v){  //έλεγχος αν v ==
first child
         if(c.less(v.getPrevious().getKey(),v.getKey()) ||
           c.equal(v.getPrevious().getKey(),v.getKey()))          //με key>=parent's key
            return v;
      }

      else if(v.getPrevious() == null){                 //έλεγχος αν v == root
         if(c.less(v.getKey(), minItemNode.getKey())){
            minItemNode = v;
            return v;
         }
         return v;
      }

      //αν v δεν είναι root ή πρώτο παιδί με key>=parent's key
      if(v.getPrevious().getLeft() == v){   //αν v == first child
         if(v.getRight() == v)            //και αν v είναι το μοναδικό παιδί
            y.setLeft(null);              //αποκοπή v από τον πατέρα του
         else{
            y.setLeft(v.getRight());      //αν δεν είναι το μοναδικό παιδί
            v.getRight().setPrevious(y);  //αποκοπή v από τον πατέρα του
         }
      }

      else{                           //αλλιώς αν v != first child
         if(v.getRight() == v)            //αν είναι τελευταίο παιδί
            y.setRight(y);                //κάνουμε το previous του τελευταίο
```

```
            else{
                y.setRight(v.getRight());     //αποκοπή από τη λίστα παιδιών
                v.getRight().setPrevious(y);
            }
        }

        v.setPrevious(null);                  //πριν το εισάγουμε στη λιστα ριζών
        v.setRight(null);                     //το γειωνουμε
        if(v.isThick()){                      //και αν είναι thick το κανουμε normal
            v.incRank();
            v.unThick();
        }
        insertTree(v);
        if(c.less(v.getKey(), minItemNode.getKey()))
            minItemNode = v;
        repairStep(y);                        //ελέγχουμε για νέα παράβαση στον y =
v.getPrevious()
        return v;
    }


    public void repairStep(thickNode y){       //διόρθωση της δομής
        //case1,αν y έχει rank +2 από του next sibling του
        //ή έχει rank 1 και κανένα sibling(και δεν είναι και root)
        if(((y.getRank() - y.getRight().getRank() == 2) ||
                (y.getRank() == 1 && y.getRight() == y)) && y.getPrevious() != null){
            //case 1a,y is normal
            if(y.isThick() == false){
                y.decRank();                  //κάνουμε τον y thick
                y.setThick();
                y = y.getPrevious();
                repairStep(y);                //έλεγχος για παράβαση στο νέο y
            }

            //case 1b,y is thick
            else{
                thickNode w = y.getLeft();          //w πρώτο παιδί του y
                if(w.getRight() != w){              //αν ο y έχει κι άλλα παιδιά
                    y.setLeft(w.getRight());        //αποκοπή του w από τον y
                    w.getRight().setPrevious(y);
                }
                else
                    y.setLeft(null);                //αν w ήταν το μοναδικό παιδί του y
                y.decRank();
```

```
            if(y.getPrevious() != null && y.getPrevious().getLeft() == y )   //αν y είναι πρώτο παιδί
                y.getPrevious().setLeft(w);     //ο πατέρας του y κάνει τον w πρώτο παιδί
            else
                y.getPrevious().setRight(w);    //αν ο y δεν είναι πρωτο παιδί,ενημέρωση του προηγούμενου left sibling του

            w.setPrevious(y.getPrevious());     //εισαγωγή του w σαν left sibling του y
            w.setRight(y);
        }

    }

    //case 2,αν y έχει rank +2 από του πρώτου παιδιού του
    //ή έχει rank 1 και κανένα παιδί
    if(((y.getLeft() != null) && (y.getRank() - y.getLeft().getRank() == 2)) ||
        (y.getRank() == 1 && y.getLeft() == null)){
       y.decRank();
       y.unThick();
       if(y.getPrevious() == null)          //έλεγχος αν y είναι ρίζα
          return;
       else{
          thickNode w = y.getRight();
          //case 2a
          if(w.isThick()){
             w.incRank();

             if(w.getRight() == w)              //swap(y,w)
                y.setRight(y);
             else{
                y.setRight(w.getRight());
                w.getRight().setPrevious(y);
             }

             w.setPrevious(y.getPrevious());

             if(y.getPrevious().getLeft() == y)
                y.getPrevious().setLeft(w);
             else
                y.getPrevious().setRight(w);

             w.setRight(y);
             y.setPrevious(w);
             w.unThick();
```

```
    y.unThick();
    return;
}

//case 2b
if(w.isThick() == false){
    thickNode z = y.getPrevious();
    if(c.less(w.getKey(),y.getKey())){              //αν y.key >= w.key
        if(z.getLeft() != null && z.getLeft() == y)  //αν z parent του y
            z.setLeft(w);
        else                                    //αν z left sibling του y
            z.setRight(w);
        w.setPrevious(z);

        if(w.getLeft() == null){                    //αν ο w δεν είχε παιδιά
            w.setLeft(y);
            y.setPrevious(w);
            y.setRight(y);
        }
        else{                                   //αλλιώς,αν ο w είχε παιδιά
            y.setRight(w.getLeft());
            y.setPrevious(w);
            w.getLeft().setPrevious(y);
            w.setLeft(y);
        }
        w.setThick();
    }
    else if(c.less(y.getKey(),w.getKey())){         //αν y.key < w.key
        if(w.getRight() == w)                       //κάνουμε το w πρώτο παιδί του y
            y.setRight(y);
        else{
            y.setRight(w.getRight());
            w.getRight().setPrevious(y);
        }
        if(y.getLeft() == null){
            y.setLeft(w);
            w.setPrevious(y);
            w.setRight(w);
        }
        else{
            w.setRight(y.getLeft());
            w.setPrevious(y);
            y.getLeft().setPrevious(w);
            y.setLeft(w);
```

```
                }
                y.setThick();
            }

            y = z;
            repairStep(y);
        }
    }
}


//case 3,αν ο y είναι thin root τον κάνουμε normal
else if(y.isThick() && y.getPrevious() == null){
    y.incRank();
    y.unThick();
}

return;

}//end of repairStep

public void delete(thickNode v){          //διαγραφή κόμβου v
    decreaseKey(v, v.getKey() - (Min()-1));      //αναγωγή σε διαγραφή ελαχίστου
    removeMin();                          //αφού μηδενίσουμε το key του
    return;                               //προς διαγραφή κόμβου
}

}
```

3.2.3 Ανάλυση Πολυπλοκότητας

Χρησιμοποιούμε και εδώ την τεχνική του δυναμικού της αναλύσεως επιμερισμένου κόστους.Ως δυναμικό **Φ** της δομής ορίζεται δύο φορές το πλήθος των δένδρων διατάξεως σωρού συν τον αριθμό των normal κόμβων που δεν είναι ρίζες.Με αυτόν τον ορισμό του δυναμικού,το επιμερισμένο κόστος μιας συνένωσης(σε μια πράξη remove-min) είναι μηδέν αν την χρεώσουμε με μια μονάδα χρόνου.Η συγχώνευση μειώνει τον αριθμό των δένδρων κατά ένα (μειώνοντας συνεπώς το δυναμικό κατά δύο) αλλά αυξάνει τον αριθμό των normal κόμβων που δεν είναι ρίζες κατά ένα (αυξάνοντας και το δυναμικό κατά ένα).Κάθε μη τερματική περίπτωση της decrease – key (1α και 2β) μετατρέπει έναν κόμβο από normal σε thick,άρα μειώνει και το δυναμικό κατά ένα.Συνεπώς,το επιμερισμένο κόστος για κάθε τέτοια περίπτωση είναι μηδέν αν την χρεώσουμε με μια μονάδα χρόνου.Το υπόλοιπο κομμάτι της ανάλυσης

πολυπλοκότητας είναι ακριβώς ίδιο με αυτό της thin heaps,επομένως το θεώρημα 3.1 ισχύει και για την thick heaps.Όπως και στην thin heaps,η ανάλυση χειρότερης περιπτώσεως της decrease-key σε έναν thick heap που περιέχει n κόμβους είναι O(logn) διότι κάθε εκτέλεση του repair step γίνεται σε κόμβο μεγαλύτερης τάξης κάθε φορά.

4. Πειραματική Αξιολόγηση

Η επαλήθευση της απόδοσης των δομών που παρουσιάσαμε γίνεται με την εκτέλεση διαφόρων πειραμάτων,ώστε να επαληθευτεί η θεωρητική απόδοση των πράξεων των δομών και τα αντίστοιχα θεωρήματα.Υπολογίζουμε το κόστος κάθε πράξης σε πλήθος στοιχειωδών πράξεων.Συγκεκριμένα,χρεώνουμε μια μονάδα κόστους για κάθε δημιουργία και διαγραφή κόμβου όπως επίσης και για κάθε αλλαγή ή ανάγνωση δείκτη.Για την μέτρηση του κόστους κάθε πράξης εισάγουμε στον κώδικά μας μεταβλητές.Σε κάθε εκτέλεση πράξης αντιστοιχεί και ένας κόμβος μιας απλής λίστας όπου αποθηκεύουμε τον αύξων αριθμό της πράξης(αν είναι π.χ η $1^{η}$,$2^{η}$ ή η $3^{η}$ διαγραφή στοιχείου κ.ο.κ.),το επιμέρους κόστος της,το μέχρι εκείνη την στιγμή συνολικό κόστος καθώς και το επιμερισμένο,το οποίου ισούται με το συνολικό δια το μέχρι στιγμής πλήθος των πράξεων.

Για παράδειγμα,επεκτείνουμε τον κώδικα υλοποίησης της thin heaps για την πράξη της εισαγωγής ως εξής:

```
public tqNode insert(int i){
      tqNode t = new tqNode(i);
      insCost++;

      if(nofItems == 0){
         minItemNode = t;
         t.setPrevious(null);
         t.setRight(t);
         nofItems++;
         nofTrees++;
         insCost += 2;
         nofIns++;
         insCounter cnt = new insCounter();
         if(insHead == null)
            insTail = insHead = cnt;
         cnt.i = nofIns;
         cnt.cost = insCost;
         insTail.next = cnt;
         insTail = cnt;
         insTotal += insCost;
         cnt.total = insTotal;
         cnt.avg = insTotal/nofIns;
         insCost = 0;
         return t;
      }

      else{
         insertTree(t);
         insCost += 4; //η insertTree πάντα αυξάνει σταθερά το κόστος κατά 4
      }
```

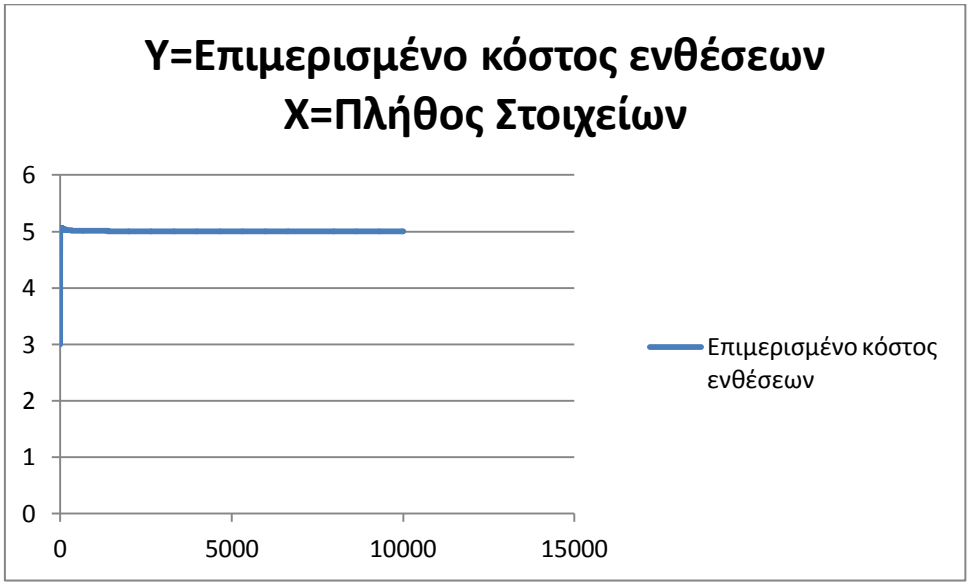
```
        nofItems++;

        if(c.less(t.getKey(),minItemNode.getKey())){
           minItemNode = t;
           insCost++;
        }

        nofIns++;
        insCounter cnt = new insCounter();
        if(insHead == null)
           insTail = insHead = cnt;
        cnt.i = nofIns;
        cnt.cost = insCost;
        insTail.next = cnt;
        insTail = cnt;
        insTotal += insCost;
        cnt.total = insTotal;
        cnt.avg = insTotal/nofIns;
        insCost = 0;
        return t;
    }
```

Στον παραπάνω κώδικα η μεταβλητή **insCost** αυξάνει κατά ένα με κάθε στοιχειώδη πράξη και μηδενίζεται πριν τον τερματισμό της εκάστοτε εισαγωγής στοιχείου.Ο κόμβος cnt κρατάει όλα τα στοιχεία της εκάστοτε εισαγωγής(μεταβλητή cost για επιμέρους κόστος,avg για επιμερισμένο και total για συνολικό) και αποτελεί κόμβο της απλής λίστας(αντικείμενα τύπου insCounter) που προαναφέραμε.Παρατηρούμε οτι δημιουργούμε ένα νέο τέτοιο αντικείμενο ακριβώς πριν τον τερματισμό της εισαγωγής.Η μεταβλητή **nofIns** αυξάνεται σταθερά κατά ένα με κάθε εισαγωγή.Με αυτές τις μεταβλητές και τις δομές είμαστε σε θέση να κρατάμε όλο το ιστορικό των ενθέσεων.Χρησιμοποιούμε εντελώς ανάλογες δομές και μεταβλητές και για την μέτρηση των υπολοίπων πράξεων.

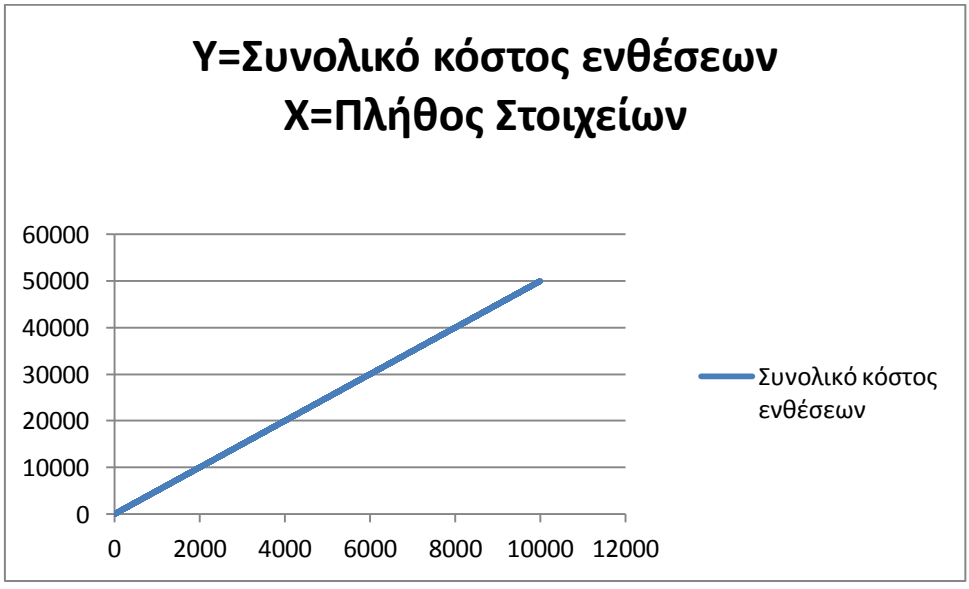
Ένθεση στοιχείου

Αφού εισάγουμε 10.000 στοιχεία (τυχαίες εισαγωγές με εύρος κλειδιού 0-1000) στην thin heap,επεξεργαζόμαστε τις μεταβλητές σε ένα φύλλο excel και παράγουμε τις γραφικές παραστάσεις του επιμερισμένου και του συνολικού κόστους ως προς το πλήθος των στοιχείων της δομής,όπου φαίνεται οτι η πράξη της ενθέσεως στοιχείου κοστίζει σταθερό χρόνο O(1).

Αριθμός Στοιχείων	Επιμερισμένο κόστος ενθέσεων	Συνολικό κόστος ενθέσεων
1	3	3
2	4,5	9
3	4,67	14
4	4,75	19
5	5	25
6	5	30
7	5	35
8	5	40
9	5	45



Επιμερισμένο κόστος ενθέσεων (Υ) ως προς πλήθος στοιχείων της δομής (Χ)



Συνολικό κόστος ενθέσεων (Υ) ως προς πλήθος στοιχείων της δομής (Χ)

Εκτελούμε το ίδιο πείραμα και στην thick heaps με τα ανάλογα αποτελέσματα.

Πλήθος Στοιχείων	Επιμερισμένο κόστος ενθέσεων	Συνολικό κόστος ενθέσεων	
1	3	3	
2	4	8	
3	4,33	13	
4	4,5	18	
5	4,6	23	
6	4,67	28	
7	4,71	33	
8	4,75	38	
9	4,78	43	

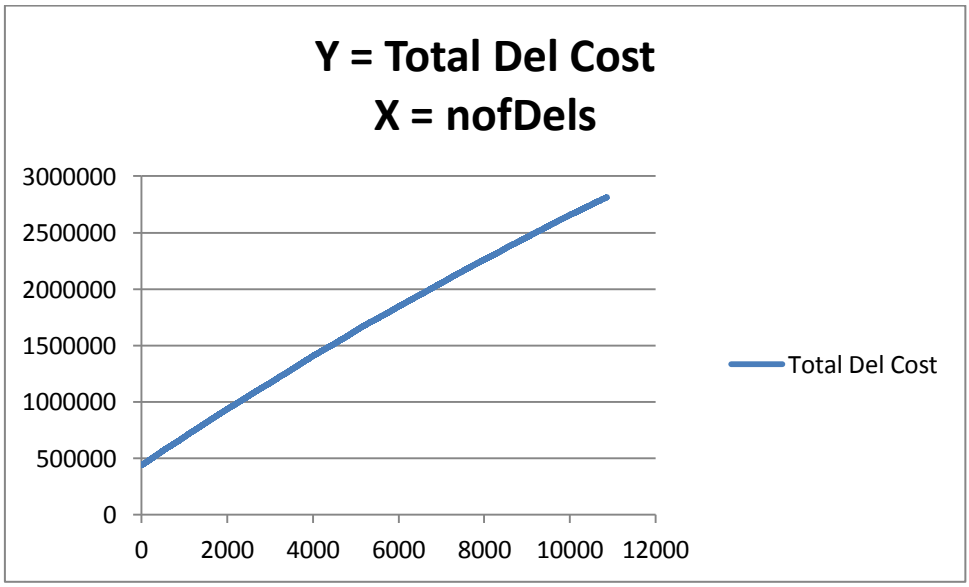
Επιμερισμένο κόστος ενθέσεων (Y) ως προς πλήθος στοιχείων της δομής (X)

Συνολικό κόστος ενθέσεων (Y) ως προς πλήθος στοιχείων της δομής (X)

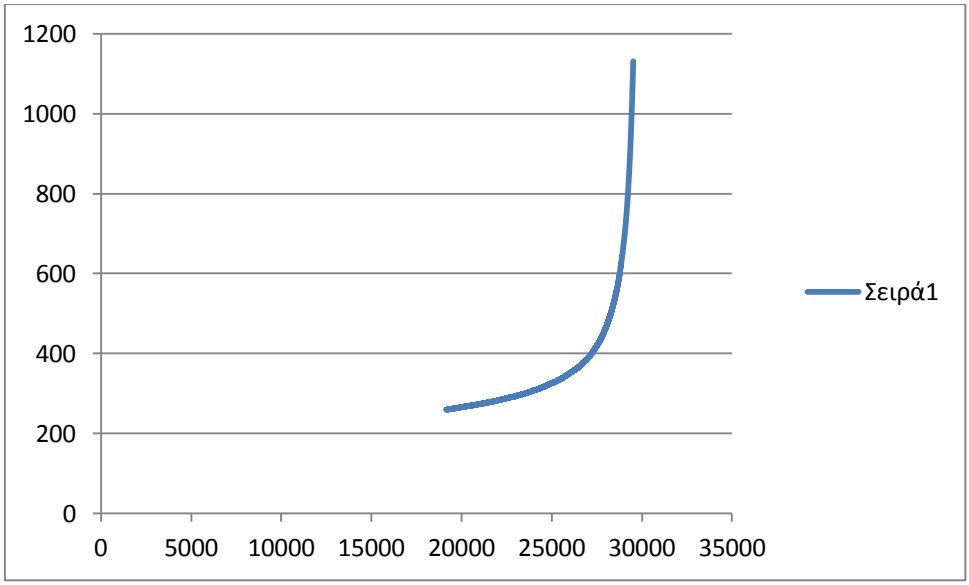
Διαγραφή Στοιχείου

Σε έναν thin heap πλήθους 30000 στοιχείων πραγματοποιούμε 10860 τυχαίες διαγραφές.Παρατηρούμε οτι το κόστος είναι λογαριθμηκό στο πλήθος των στοιχείων.

nofItems	nofDeletions	Total Del Cost	amortized Del Cost
29999	1	434936	434936
29998	2	435313	217656,5
29997	3	435521	145173,67
29996	4	435879	108969,75
29995	5	436055	87211
29994	6	436189	72698,17
29993	7	436472	62353,14
29992	8	436793	54599,13
29991	9	437163	48573,67
29990	10	437283	43728,3
29989	11	437479	39770,82
29988	12	437860	36488,33
29987	13	438036	33695,08
29986	14	438172	31298
29985	15	438573	29238,2
29984	16	438747	27421,69
29983	17	438945	25820,29



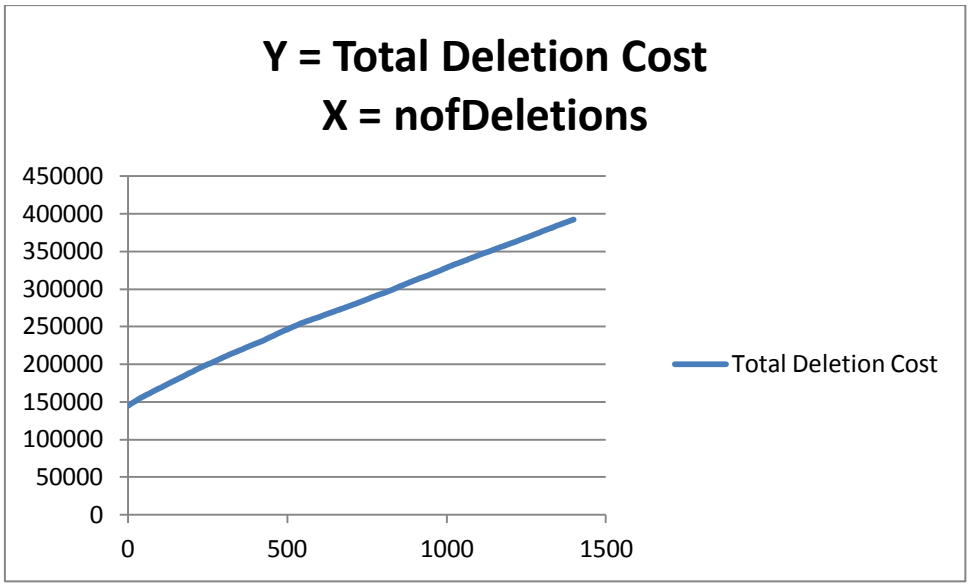
Συνολικό κόστος διαγραφών (Y) ως προς το πλήθος πράξεων διαγραφής (X)



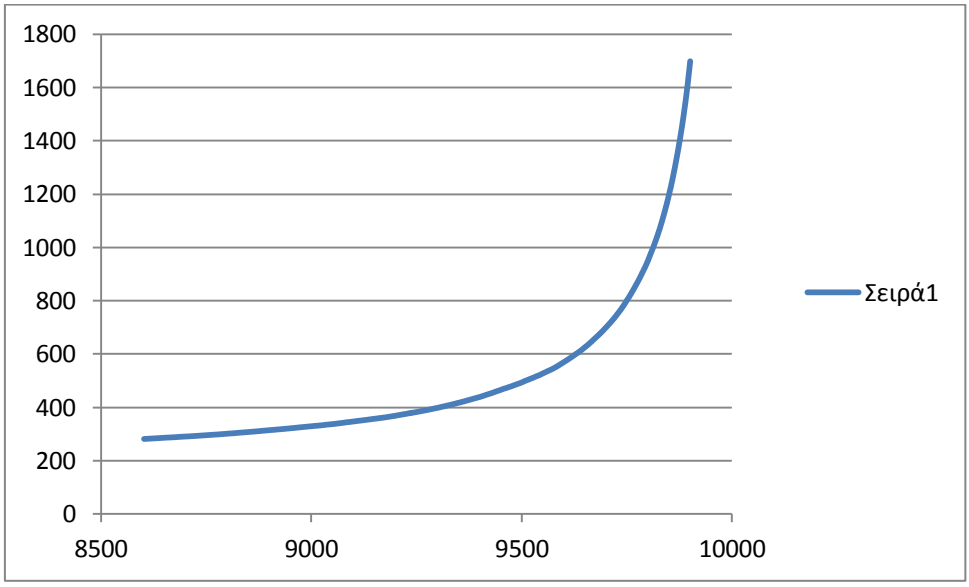
Επιμερισμένο κόστος διαγραφής (Y) ως προς πλήθος των στοιχείων της δομής (X)

Ομοίως και στην περίπτωση του thick heap,για 1398 τυχαίες διαγραφές σε μέγεθος δομής 10.000 κόμβων:

nofItems	Επιμερισμένο Deletion Cost	nofDeletions	Total Deletion Cost
9999	144953	1	144953
9998	72698,5	2	145397
9997	48554,33	3	145663
9996	36471	4	145884
9995	29240,2	5	146201
9994	24415,17	6	146491
9993	20953,57	7	146675
9992	18373,25	8	146986
9991	16365,67	9	147291
9990	14753,6	10	147536
9989	13443,73	11	147881
9988	12344,25	12	148131
9987	11412,23	13	148359
9986	10618	14	148652
9985	9930,33	15	148955
9984	9326	16	149216
9983	8786,65	17	149373
9982	8313,89	18	149650
9981	7889,26	19	149896



Συνολικό κόστος διαγραφών (Y) ως προς το πλήθος πράξεων διαγραφής (X)

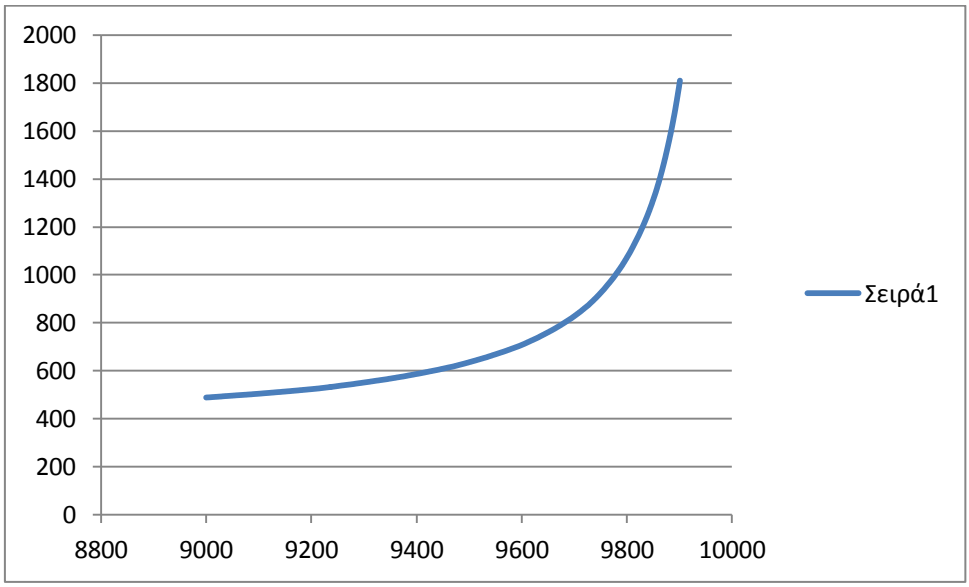


Επιμερισμένο κόστος διαγραφής (Y) ως προς πλήθος των στοιχείων της δομής (X)

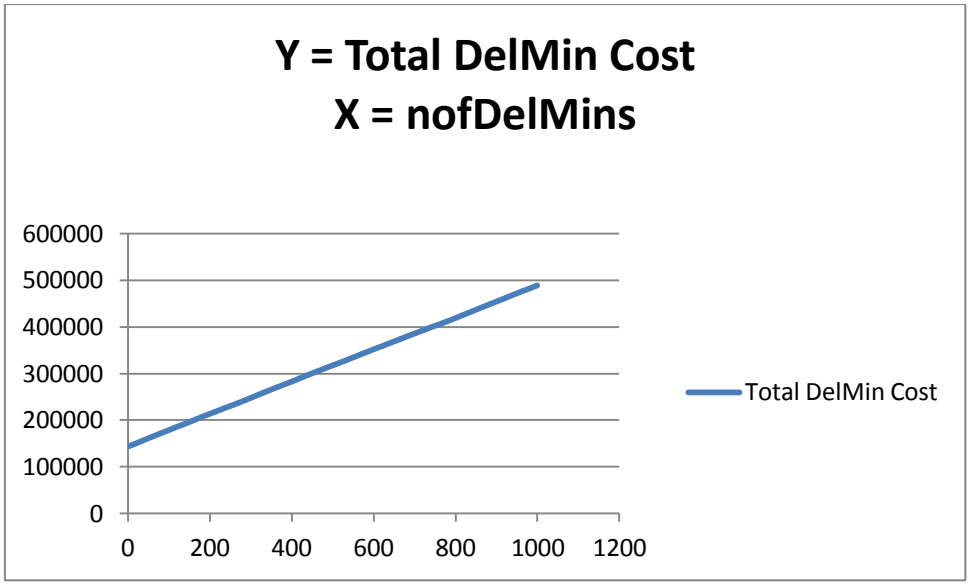
Διαγραφή Ελαχίστου

Σε έναν thin heap πλήθους 10.000 στοιχείων πραγματοποιούμε 1000 διαγραφές ελαχίστου.Παρατηρούμε οτι το κόστος είναι λογαριθμηκό στο πλήθος των στοιχείων.

nofItems	nofDelMins	Επιμερισμένο DelMin Cost	Total DelMin Cost
9999	1	144942	144942
9998	2	72657,5	145315
9997	3	48471,67	145415
9996	4	36429,25	145717
9995	5	29211,8	146059
9994	6	24404,17	146425
9993	7	20968,14	146777
9992	8	18391,75	147134
9991	9	16384,22	147458
9990	10	14782,4	147824
9989	11	13470,45	148175
9988	12	12377,67	148532
9987	13	11451,15	148865
9986	14	10658,57	149220
9985	15	9966,27	149494
9984	16	9364,88	149838
9983	17	8826	150042
9982	18	8358,39	150451
9981	19	7939,16	150844



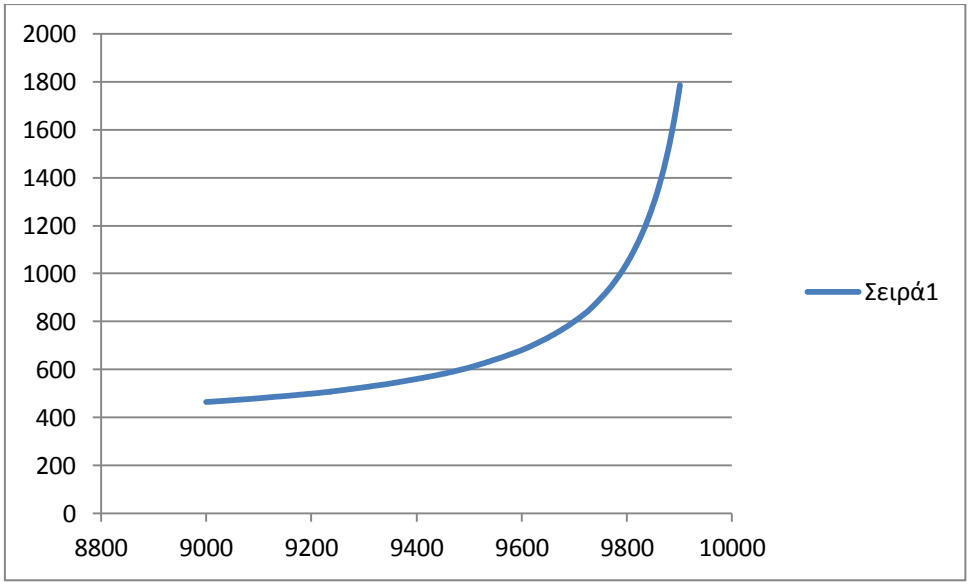
Επιμερισμένο κόστος removeMin (Y) ως προς πλήθος των στοιχείων της δομής (X)



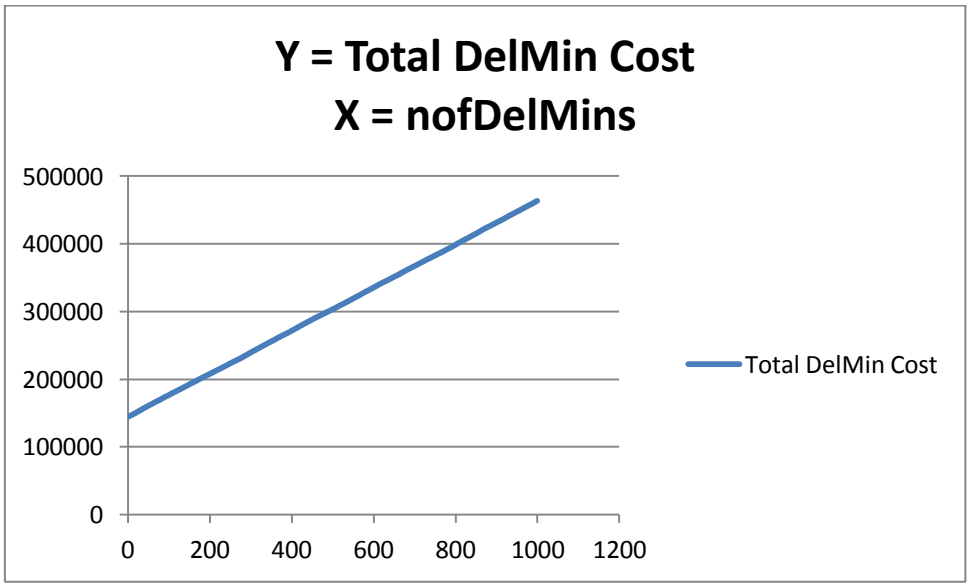
Συνολικό κόστος removeMin (Y) ως προς πλήθος πράξεων διαγραφής ελαχίστου (X)

Ομοίως και στην περίπτωση του thick heap:

nofItems	Επιμερισμένο DelMin Cost	nofDelMins	Total DelMin Cost
9999	144942	1	144942
9998	72625	2	145250
9997	48536,67	3	145610
9996	36494	4	145976
9995	29263,8	5	146319
9994	24436,83	6	146621
9993	20995,71	7	146970
9992	18401,88	8	147215
9991	16383,56	9	147452
9990	14781,8	10	147818
9989	13463,91	11	148103
9988	12371,5	12	148458
9987	11445,31	13	148789
9986	10653,21	14	149145
9985	9965,6	15	149484
9984	9358,81	16	149741
9983	8820,24	17	149944
9982	8352,89	18	150352
9981	7933,95	19	150745



Επιμερισμένο κόστος removeMin (Y) ως προς πλήθος των στοιχείων της δομής (X)

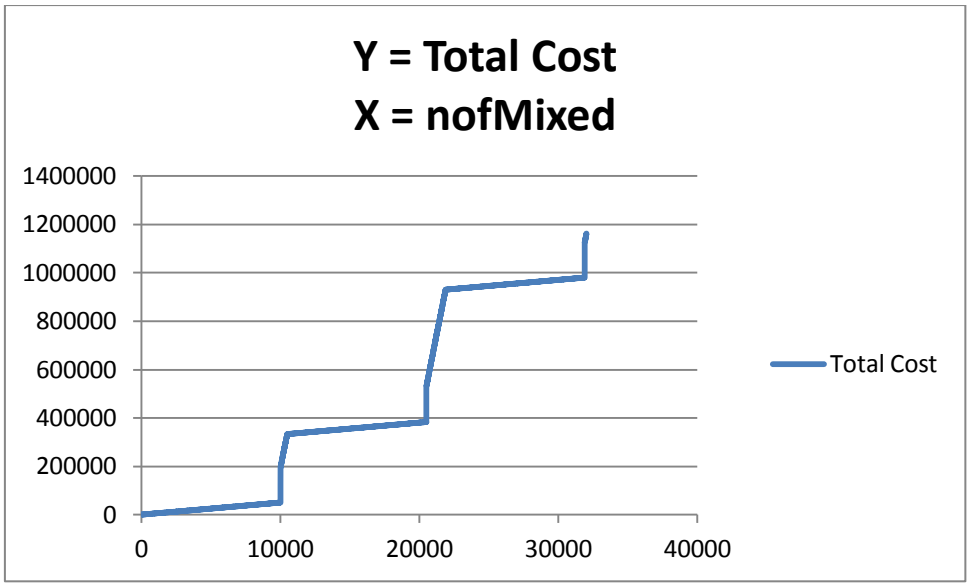


Συνολικό κόστος (Y) ως προς πλήθος πράξεων διαγραφής ελαχίστου (X)

Μεικτή ακολουθία πράξεων

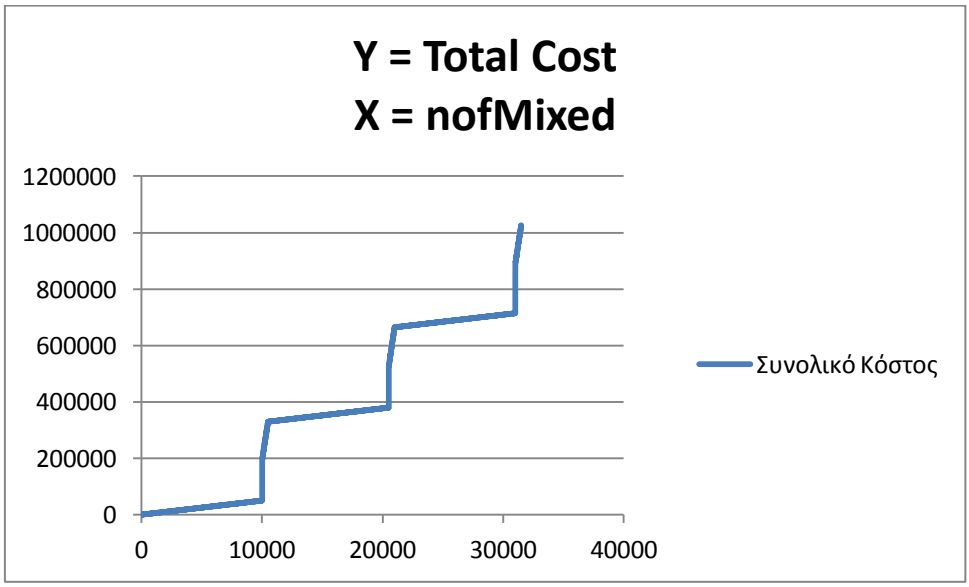
Αφού ενθέσουμε 10000 στοιχεία,προχωράμε σε τυχαία ανάμιξη περίπου 500 διαγραφών και διαγραφών ελαχίστου.Στη συνέχεια ενθέτουμε άλλα 10000 στοιχεία και ακολουθούν ξανά μικτές διαγραφές/διαγραφές ελαχίστου.
Τέλος,επαναλαμβάνουμε και μια τρίτη φορά την παραπάνω ανάμιξη.

Το αποτέλεσμα στην thin heap είναι το εξής:



Συνολικό κόστος (Y) ως προς το πλήθος των αναμεμιγμένων πράξεων (X)

Το ίδιο πείραμα στην thick heaps έχει το εξής αποτέλεσμα:

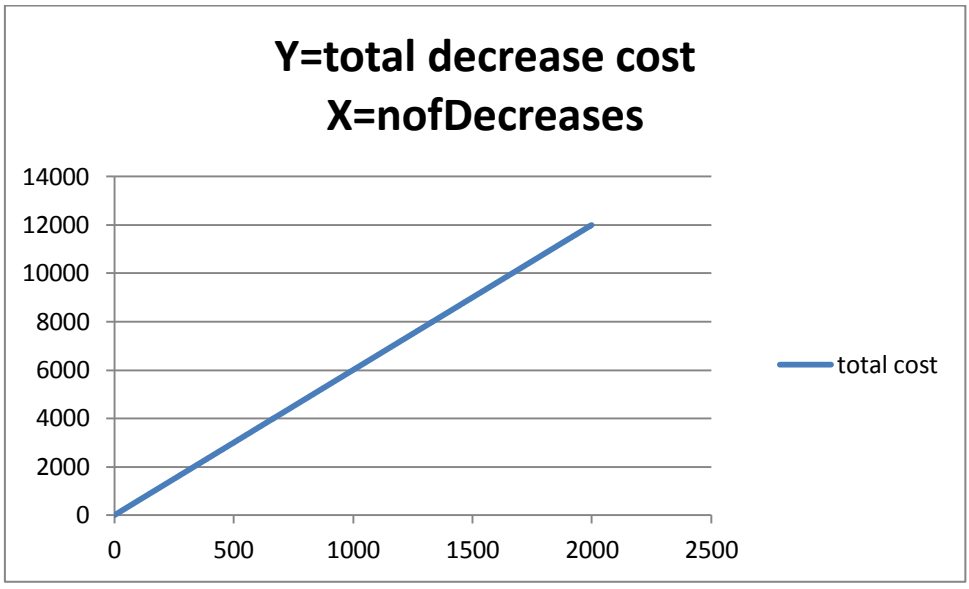


Συνολικό κόστος (Y) ως προς το πλήθος των αναμεμιγμένων πράξεων (X)

Μείωση προτεραιότητας

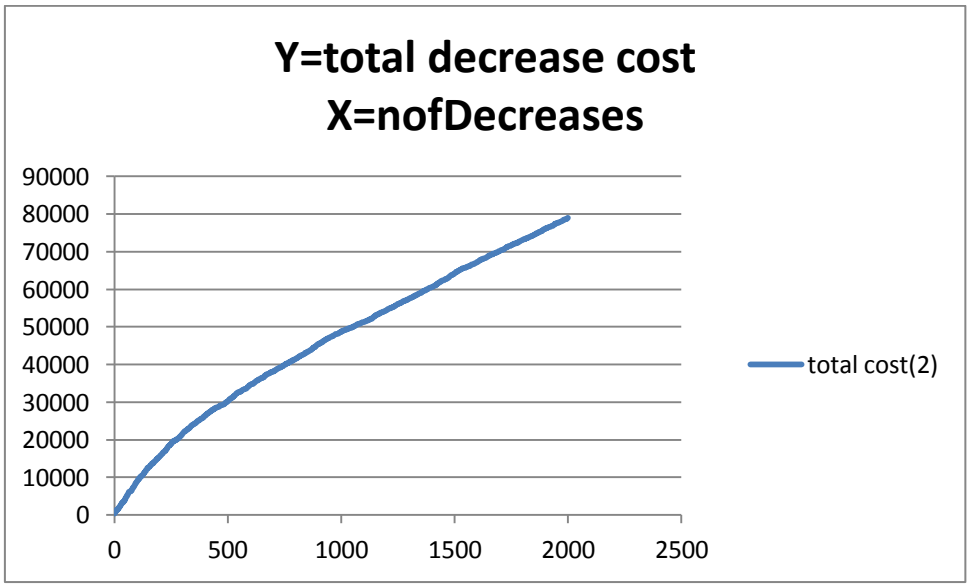
Σε έναν άδειο σωρό,ενθέτουμε 10000 στοιχεία.Στην συνέχεια εκτελούμε δύο πειράματα.Στην πρώτη περίπτωση κάνουμε 2000 τυχαίες μειώσεις προτεραιότητας,ενώ στην δεύτερη κάνουμε το ίδιο αφού όμως πρώτα εκτελέσουμε μια διαγραφή ελαχίστου η οποία αναδιατάσει - «τακτοποιεί» το δάσος.Παρατηρούμε οτι στην πρώτη περίπτωση το επιμερισμένο κόστος είναι σταθερό και το συνολικό μεταβάλλεται ανάλογα ως προς το πλήθος των μειώσεων πρετεραιότητας ενώ στη δεύτερη μεταβάλλεται λογαριθμηκά,καθώς αρκετές από τις πράξεις είναι πλέον πιο απαιτητικές σε κόστος λόγω εκτελέσεων του repair step περισσότερες φορές και σε μεγαλύτερο βάθος.

nofDecreases	amortized cost	total cost	
1	6	6	
2	6	12	
3	6	18	
4	6	24	
5	6	30	
6	6	36	
7	6	42	
8	6	48	
9	6	54	
10	6	60	
11	6	66	
12	6	72	
13	6	78	
14	6	84	
15	6	90	
16	6	96	
17	6	102	
18	6	108	



Συνολικό κόστος decreaseKey (Y) ως προς πλήθος μειώσεων προτεραιότητας (X)

nofDecreases	amortized cost(2)	total cost(2)	
1	298	298	
2	232	464	
3	214,33	643	
4	161,75	647	
5	130,2	651	
6	145	870	
7	151,14	1058	
8	156,88	1255	
9	139,89	1259	
10	126,3	1263	
11	115,18	1267	
12	105,92	1271	
13	100,62	1308	
14	93,71	1312	
15	91,67	1375	



Συνολικό κόστος decreaseKey (Y) ως προς πλήθος μειώσεων προτεραιότητας (X)

Ανάλυση Χωρικής Πολυπλοκότητας

Σε έναν αρχικώς άδειο σωρό,ενθέτουμε 30.000 στοιχεία-κόμβους σε κάθε έναν από του Fibonacci,Thin και Thick Heap.Με τη χρήση του NetBeans profiler,εξάγουμε το συνολικό χώρο σε Bytes που καταλαμβάνουν αυτοί οι κόμβοι σε κάθε δομή.Το αποτέλεσμα συνοψίζεται στον παρακάτω πίνακα και είναι το απολύτως αναμενόμενο καθότι οι Thin και Thick Heaps χρησιμοποιούν έναν δείκτη λιγότερο ανά κόμβο σε σχέση με τον Fibonacci Heap (τρεις έναντι τεσσάρων δεικτών).

Είδος Δομής	Σύνολο Κόμβων	Συνολικός Χώρος σε bytes	bytes/κόμβο
Fibonacci Heap	30.000	126.560	4,218666667
Thin Heap	30.000	101.088	3,3696
Thick Heap	30.000	101.504	3,383466667

5. Βιβλιογραφία

1. Haim Kaplan, Robert Endre Tarjan. Thin heaps, thick heaps. ACM Transactions on Algorithms (2008)
2. Παναγιώτης Δ. Μποζάνης. Δομές Δεδομένων (2006)
3. Fredman, M. L., and Tarjan, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. (1987)
4. Kaplan, H., and Tarjan, R. E. New heap data structures. Tech. Rep. TR-597-99, Princeton University. (1999)
5. Vuillemin, J. A data structure for manipulating priority queues. *Commun. ACM 21*, 309–314. (1978)
6. Tarjan,R. E. Amortized computational complexity. *SIAM J. Algebr. Discrete Meth. 6*, 2, 306–318. (1985)