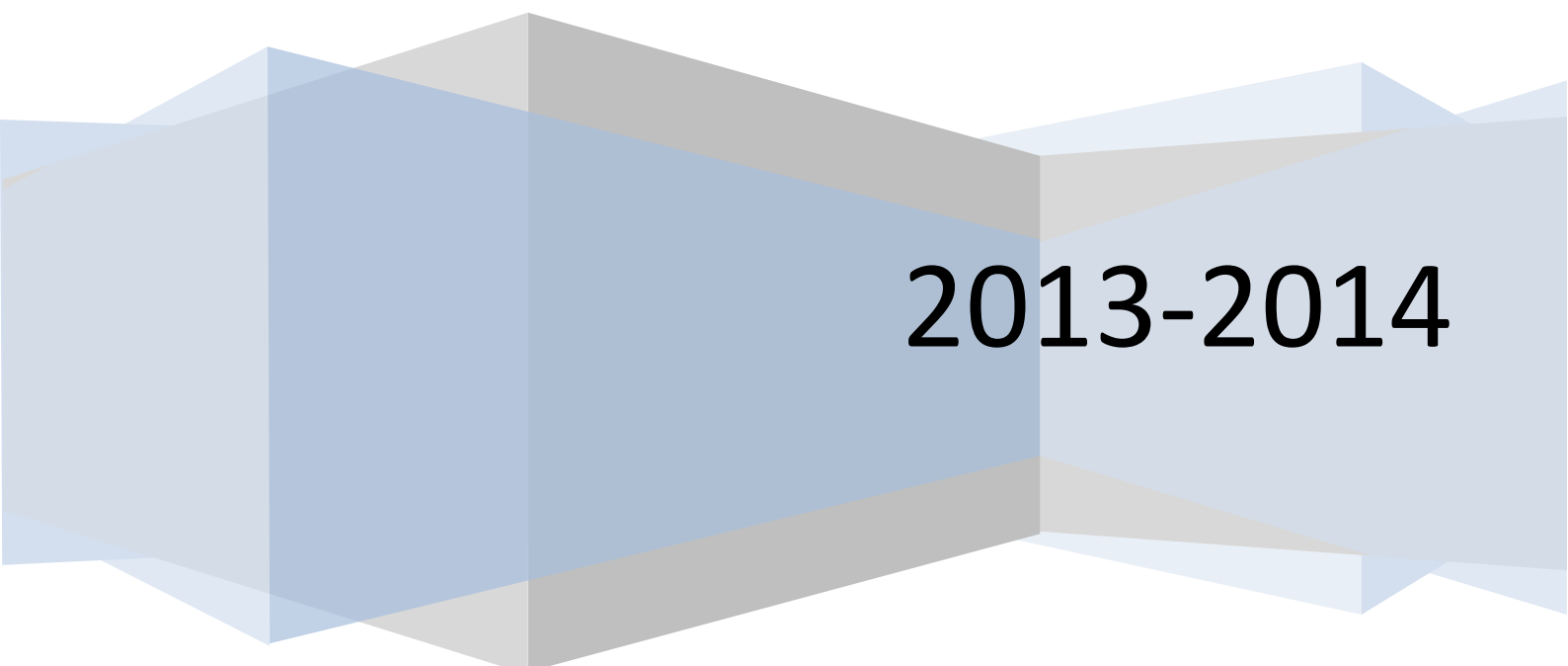


Πανεπιστήμιο Θεσσαλίας
Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών
Υπολογιστών

Μελέτη Βαθμοταιριασμένων Σωρών

Επιβλέπων καθηγητής Μποζάνης Παναγιώτης

Εμμανουηλίδης Κωνσταντίνος



2013-2014

Περιεχόμενα

Κεφάλαιο 1 : Εισαγωγή	2
Κεφάλαιο 2 : Οργάνωση	6
Κεφάλαιο 3 : Τουρνουά	6
Κεφάλαιο 4 : Lazy binomial queue	9
Κεφάλαιο 5 : Rank pairing heap	16
Κεφάλαιο 6 : Amortized efficiency των rank pairing heap.....	23
Κεφάλαιο 7 : One-tree rank pairing heap	30
Κεφάλαιο 8 : Μπορεί η μείωση κλειδιού να γίνει πιο εύκολη;.....	32
Κεφάλαιο 9 : Παρατηρήσεις	35
Κεφάλαιο 10 : Παραδείγματα πράξεων rank pairing heap	36
Παράρτημα : Κώδικας αλγορίθμου	56

Περίληψη

Παρουσιάζουμε τον αλγόριθμο Βαθμοταιριασμένων Σωρών στο εξής rank pairing heaps, όπως περιγράφεται από τους BERNHARD HAEUPLER, SIDDHARTHA SEN, AND ROBERT E. TARJAN, ο οποίος συνδυάζει την ασυμπτωτική αποτελεσματικότητα (asymptotic efficiency) του σωρού Fibonacci (Fibonacci heap) και την απλότητα του αλγορίθμου pairing heap. Άλλοι σωροί που έχουν ίδια όρια (bounds) με τα όρια του σωρού Fibonacci, το καταφέρνουν, διατηρώντας έναν κανόνα ισορροπίας για τα δένδρα που αντιπροσωπεύουν τον σωρό. Σε αντίθεση με αυτούς τους αλγορίθμους, αλλά στην ίδια λογική με τον pairing heap τα δένδρα που παρουσιάζονται μπορούν να αναπτυχθούν με παράξενη δομή. Να είναι δηλαδή μη ισορροπημένα. Ο αλγόριθμος απαιτεί το πολύ ένα κόψιμο (cut) στο δένδρο και καμία άλλη αναδιάρθρωση (restructuring), για κάθε μείωση κλειδιού (key decrease). Η μόνη αλλαγή η οποία μπορεί να επέλθει κατά τη διάρκεια της μείωσης κλειδιού είναι η αλλαγή στα ranks των κόμβων. Αν και η δομή είναι απλή, η ανάλυση της δεν είναι τόσο απλή.

1. Εισαγωγή

Ένας meldable σωρός (σωρό) είναι μια δομή δεδομένων που αποτελείται από ένα σύνολο στοιχείων, το καθένα με διαφορετικές τιμές κλειδιών, που υποστηρίζουν τις ακόλουθες πράξεις (λειτουργίες):

- Make – heap : επιστρέφει έναν καινούργιο , άδειο σωρό
- Insert (x, H) : εισάγει το στοιχείο x με προκαθορισμένο κλειδί
- Find-min(H) : Επιστρέφει το στοιχείο με το μικρότερο κλειδί στο σωρό
- Delete-min(H) : Αν ο σωρός δεν είναι άδειος, διαγράφει το στοιχείο με το μικρότερο κλειδί (key).
- Meld (H_1, H_2) : επιστρέφει έναν σωρό με τα στοιχεία ενωμένα από τους δυο σωρούς H_1 και H_2 , και μετέπειτα καταστρέφει τους H_1 και H_2 .

Μερικοί αλγόριθμοι χρειάζονται επιπλέον τις δυο επόμενες πράξεις:

- Decreasy-key(x, Δ, H) : Μείωση το κλειδί του στοιχείου x από τον σωρό H , στην νέα τιμή Δ , υποθέτοντας ότι ο σωρός H είναι αυτός που περιέχει το στοιχείο x .
- Delete(x, H) : Διαγραφή του στοιχείου x από τον σωρό H , υποθέτοντας ότι ο σωρός H είναι αυτός που περιέχει το στοιχείο x .

Σύμφωνα με τους BERNHARD HAEUPLER, SIDDHARTHA SEN, AND ROBERT E. TARJAN γίνονται μόνο δυαδικές συγκρίσεις κλειδιών, και εξετάζεται η amortized efficiency που έχουν οι λειτουργίες στους σωρούς. Για να δημιουργηθεί ένα όριο στην amortized efficiency, ανατίθεται σε κάθε δομή δεδομένων ένα μη μηδενικό δυναμικό, που είναι αρχικά μηδέν. Ορίζεται ως amortized time μιας λειτουργίας (πράξη) ως τον πραγματικό χρόνο , συν την αλλαγή του δυναμικού που αυτό προκαλεί. Έτσι σε κάθε αλληλουχία λειτουργιών το άθροισμα των πραγματικών χρόνων είναι το πολύ ίσο με το amortized time.

Εφόσον η αριθμοί μπορούν να ταξινομηθούν κάνοντας η εισαγωγές (insertions) σε έναν αρχικά άδειο σωρό, ακολουθούμενο από n -διαγραφές (n -deletions), το κλασσικό $\Omega(n \log(n))$ κάτω όριο (lower bound) στον αριθμό των δυαδικών συγκρίσεων που χρειάζονται για την ταξινόμηση συνεπάγεται ότι, είτε η εισαγωγή στοιχείου (insertion) είτε η διαγραφή ελαχίστου (delete-min) χρειάζονται $\Omega(\log(n))$ amortized time, όπου n είναι ο αριθμός των στοιχείων που υπάρχουν αυτή τη στιγμή στο σωρό, τα οποία για την απλότητα των εργασιών, έστω ότι είναι τουλάχιστον δυο. Εξετάζονται απλές δομές δεδομένων όπου η διαγραφή μικρότερου (minimum deletion) ή η διαγραφή ενός τυχαίου στοιχείου εφόσον ο αλγόριθμος το υποστηρίζει , χρειάζονται $O(\log n)$ amortized time, ενώ οι άλλες πράξεις που υποστηρίζει ο σωρός, χρειάζονται $O(1)$ amortized time. Αυτά τα όρια ταιριάζουν με τα κατώτερα όρια (lower bounds).

Πολλοί αλγόριθμοι σωρού έχουν παρουσιαστεί σε βάθος χρόνου. Αναφέρονται μόνο στους αλγόριθμους που έχουν σχέση αλγόριθμο που εξετάζουμε. Ο δυωνυμικός αλγόριθμος (binomial queue) του Vuillemin υποστηρίζει όλες τις πράξεις σε χρόνο χειρότερης περίπτωσης $O(\log(n))$ ανά πράξη. Ο Fredman και ο Tarzan ανακάλυψαν τον σωρό Fibonacci (Fibonacci Heap) ειδικά, για να υποστηρίξει την πράξη της μείωσης κλειδιού σε χρόνο $O(1)$, ο οποίος μας δίνει την δυνατότητα να έχουμε efficient implementation του αλγορίθμου Dijkstra για την λειτουργία του συντομότερου μονοπατιού (Dijkstra's shortest path). Επίσης ο σωρός Fibonacci βοήθα και στον αλγόριθμο της ελάχιστης διακλάδωσης του Edmond (Edmond minimum branching algorithm), αλλά και σε αρκετούς αλγορίθμους ελαχίστων γεννητικών δένδρων (minimum spanning tree algorithms)[15,17]. Ο σωρός Fibonacci υποστηρίζει την πράξη της διαγραφής του ελαχίστου στοιχείου και της διαγραφής οποιουδήποτε στοιχείου σε amortized χρόνο $O(\log n)$ και τις υπόλοιπες πράξεις σε amortized χρόνο $O(1)$.

Αρκετά χρόνια μετά την παρουσίαση του σωρού Fibonacci ο Fredman παρουσίασε έναν σχετικά παρόμοιο αλγόριθμο με αυτορυθμιζόμενη λογική τον οποίο ονόμασε pairing heap. Ο pairing heap υποστηρίζει όλες τις πράξεις του σωρού σε amortized χρόνο $O(\log n)$. Ο σωρός Fibonacci δεν λειτουργεί καλά σε πραγματικές συνθήκες, σε αντίθεση με τον αλγόριθμο pairing heap ο οποίος λειτουργεί καλά σε πραγματικές συνθήκες. Ο Fredman et. al. υπέθεσε ότι ο pairing heap έχει την ίδια amortized efficiency με τον αλγόριθμο Fibonacci, και συγκεκριμένα, όριο amortized χρόνου $O(1)$ για την πράξη της μείωσης κλειδιού (key decrease). Παρόλο που με εμπειρικά στοιχεία αποδείχθηκε σωστή η υπόθεση που έκανε ο Fredman et. al., ο ίδιος απέδειξε ότι τελικά δεν είναι σωστά τα amortized times του pairing heap. Απέδειξε ότι ο pairing heap και παρόμοιοι αλγόριθμοι που δεν αποθηκεύουν στοιχεία για το μέγεθος του υποδένδρου (size of subtree) χρειάζονται amortized χρόνο για την μείωση κλειδιού $O(\log \log n)$ εάν οι υπόλοιπες πράξεις χρειάζονται amortized χρόνο $O(\log n)$. Όταν ο pairing heap πιάνει αυτό το όριο είναι ανοικτός (open). Το καλύτερο άνω όριο (upper bound) για την μείωση κλειδιού είναι $O(2^{2\sqrt{\lg \lg n}})$ εάν οι άλλες πράξεις επιτρέπεται να γίνονται σε amortized χρόνο $O(\log n)$.

Αυτά τα αποτελέσματα έδωσαν κίνητρο για να βελτιωθούν οι αλγόριθμοι Fibonacci και pairing heap. Μερικές από τις προσπάθειες κατάφεραν να έχουν καλύτερα όρια, αλλά το κόστος που πληρώνουν είναι ότι η δομή δεδομένων έγινε πιο πολύπλοκη. Ειδικότερα τα όρια του σωρού Fibonacci μπορούν να γίνουν worst-case. Ο αλγόριθμος Run relaxed heap πιάνει τα όρια εκτός από την πράξη της συνένωσης (meld), η οποία χρειάζεται χρόνο $O(\log n)$. Οι αλγόριθμοι fat heaps και trinomial heap που είναι παρόμοιες δομές πού πιάνουν τα ίδια όρια με πιο εύκολο τρόπο. Ο αλγόριθμος two-tier relaxed heap έχει πιο πολύπλοκη δομή αλλά πιάνει τα όρια με δυο βελτιώσεις: Η πρώτη είναι, η διαγραφή μικρότερου (minimum deletion) η οποία χρειάζεται χρόνο $\lg n + O(\log \log n)$ για τις συγκρίσεις, έτσι πετυχαίνει το καλύτερο δυνατό σταθερό παράγοντα (factor) στις συγκρίσεις. Η δεύτερη βελτίωση είναι για την συνένωση (meld) που χρειάζεται χρόνο $O(\log n')$, όπου n' είναι ο αριθμός των στοιχείων του μικρού σωρού που είναι, να γίνει η συνένωση. Ο αλγόριθμος του Brodal, και ο αλγόριθμος του Brodal και Okasaki πιάνουν τα όρια του Fibonacci εκτός από την μείωση κλειδιού, η οποία παίρνει χρόνο $O(\log n)$ στην χειρότερη περίπτωση. Μια πολύ πολύπλοκη υλοποίηση του Brodal, επιτυγχάνει όλα τα όρια στην

χειρότερη περίπτωση. Εργαζόμενος προς άλλη κατεύθυνση, ο Elmasry πρότεινε έναν εναλλακτικό αλγόριθμο αντί του pairing heap, με λογική να μην αποθηκεύει το μέγεθος των υποδένδρων (size of subtree) αλλά θέλει amortized χρόνο $O(\log n)$ για την μείωση κλειδιού, πιάνοντας τα κάτω όρια του αλγόριθμου Fredman (τα όρια του Fredman στην πραγματικότητα δεν εφαρμόζονται στον αλγόριθμο του Elmasry επειδή η δομή του δεν πληροί ορισμένους τεχνικούς περιορισμούς στα όρια).

Δουλεύοντας επίσης προς άλλη κατεύθυνση, πολλοί επιστήμονες πρότειναν δομές δεδομένων με amortized efficiency ίδια με τον σωρό Fibonacci , αλλά στη λογική να είναι πιο απλές. Αυτός είναι και ο σκοπός του αλγορίθμου που παρουσιάζουμε. Ο Peterson πρότεινε έναν αλγόριθμο βασισμένο σε δένδρα AVL (AVL Trees). Ο Høyer δημιούργησε αρκετές δομές , συμπεριλαμβανομένης μιας δομής με δένδρα AVL, red-black δένδρα και a,b-δένδρα. Ο Takaoka έδωσε έναν αλγόριθμο βασισμένο σε 2,3-δένδρα . Η πιο απλή δομή του Høyer ονομάζεται one-step heap. Ο Kaplan and Tarjan γέμισαν ένα κενό στον one-step heap και έδωσαν έναν παρόμοιο αλγόριθμο , τον thin heap. Ο Elmasry [11] δημιούργησε τον αλγόριθμο violation heaps και ο Chan [5] τον quake heaps. Ο αλγόριθμος violation heaps είναι παρόμοιος με τον rank pairing heap στη λογική του, ότι δηλαδή αποφεύγουν τις διαδοχικές αποκοπές δένδρων που γίνονται στο σωρό Fibonacci , αλλά χρησιμοποιούν μια διαφορετική φόρμα αποκοπής που μετακινεί δύο υποδέντρα αντί ενός, και χρησιμοποιούν μια κάπως περίπλοκη μέθοδο για την ενημέρωση του rank (rank-update). Ο αλγόριθμός quake heaps κάνει μεγάλης κλίμακας ανακατασκευή που προκαλείται από ένα subtree και γίνεται αρκετά ασύμμετρο.

Εκτός από τους Violation heaps, όλες αυτές οι δομές έχουν το κοινό χαρακτηριστικό ότι τα δένδρα που εκπροσωπούν το σωρό έχουν κάποιου είδους μεταβλητή ισορροπίας την οποία οι πράξεις του σωρού πρέπει να διατηρούν. Ως αποτέλεσμα, μια μείωση κλειδιού μπορεί να προκαλέσει ένα αυθαίρετο ποσό ανακατασκευών . Η αντίληψη για τους rank-pairing heaps είναι ότι δεν χρειάζεται αυτή η μεταβλητή ισορροπίας. Το μόνο που χρειάζεται είναι ένας τρόπος να ελέγχονται τα μεγέθη των υποδένδρων που συνενώνονται. Η καινούργια δομή δεδομένων, ο αλγόριθμος rank pairing heap ή rp-heap, χρειάζεται το πολύ ένα κόψιμο (cut) και καμία άλλη ανακατασκευή για κάθε μείωση κλειδιού. Όπως και σε όλους τους αλγορίθμους προηγουμένως και για τον rank pairing heap, αποθηκεύεται μια μεταβλητή για κάθε κόμβο με όνομα rank. Τα ranks δίνουν κάτω όρια (όχι άνω όρια) του μεγέθους των υποδένδρων. Μόνο δένδρα των οποίων οι ρίζες έχουν ίδιο rank συνενώνονται. Μετά από μια μείωση κλειδιού , το rank αλλάζει (μειώνεται) και μπορεί να σκαρφαλώσει στο δένδρο. Αλλά δεδομένου ότι υπάρχει μόνο ένα cut για μια μείωση κλειδιού , κατάλληλες αλληλουχίες από μειώσεις κλειδιών μπορούν να προκαλέσουν στα δένδρα που αντιπροσωπεύουν τον σωρό να αναπτυχθούν και να έχουν αυθαίρετη δομή , ακόμα και ανισορροπία στη δομή τους. Ο αλγόριθμος rank pairing heap έχει την ίδια amortized efficiency με τον σωρό Fibonacci, και την πιο απλή δομή που έχει προταθεί μέχρι τώρα (Όπως σημειώνεται παραπάνω, ο αλγόριθμος violation heaps είναι παρόμοιος με τον rp-heap αλλά χρησιμοποιεί μια πιο πολύπλοκη μορφή για το cut και έναν πιο περίπλοκο κανόνα για την ενημέρωση του rank. Αν και ο αλγόριθμος rp-heap είναι απλός , η ανάλυση του δεν είναι τόσο απλή.

Ολοκληρώνοντας την ενότητα 1 παρουσιάζουμε σε έναν πίνακα συγκεντρωτικά τους amortized χρόνους που αναφέραμε παραπάνω.

Αλγόριθμος	Make-hip	Insert	Find-min	Delete-min	Meld	Decrease-key	delete
Fibonacci Heap Fredman, Tarzan	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(\log n)$
Binomial queue Vuillemin	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Pairing heap Fredman et. Al.	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log \log n)$	$O(\log n)$
Run Relaxed Heap	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
Fat heap Trinomial Heap	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n)$
Two tier relaxed heap	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n')$	$O(1)$	$O(\log n)$
Brodal okasaki	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Quake heaps							
Violation heaps							
Thin heaps							
Høyer's heaps							
Peterson's heaps							
Rank pairing heap							

2. Οργάνωση.

Η υπόλοιπη εργασία αποτελείται από επτά ενότητες. Στην αρχή βλέπουμε την κοινή βάση που υπάρχει για τους αλγορίθμους, όπως είναι ο δυωνυμικός σωρός (binomial queue), ο σωρός Fibonacci (Fibonacci heap), ο pairing heap, και άλλες σχετικές δομές. Κάθε μία από αυτές τις δομές μπορεί να θεωρηθεί ως ένα σύνολο τουρνουά μονού αποκλεισμού (single elimination tournaments).

Η ενότητα 3 εξετάζει τέτοια τουρνουά και τους τρόπους που αυτά μπορούν να παρουσιαστούν.

Η ενότητα 4 εξερευνά τρεις διαφορετικές εκδοχές της δυωνυμικής ουράς. Και οι τρεις εκδοχές χαρακτηρίζονται ως τεμπέλικες (lazy) και έχουν τα εξής ονόματα: multipass (παλαιά εκδοχή), one-pass (νέα εκδοχή), one-tree (νέα εκδοχή).

Η ενότητα 5 επεκτείνει την δυωνυμική ουρά multipass ώστε να υποστηρίζει μείωση κλειδιού και διαγραφή οποιουδήποτε στοιχείου, δίνοντας μας τον rank pairing heap. Το ίδιο ισχύει και για την δυωνυμική ουρά one-pass. Υπάρχουν δύο τύποι rank pairing heap, ο τύπου 1 και ο τύπου 2, οι οποίοι διαφέρουν μόνο ως προς τον κανόνα που τηρείται για τα ranks των κόμβων. Ο rank pairing heap τύπου 2 υπακούει σε έναν ασθενέστερο rank rule, τον οποίο τον καθιστά λίγο πιο περίπλοκο, αλλά απλοποιεί την ανάλυση του και παράγει μικρότερους σταθερούς συντελεστές.

Η ενότητα 6 αναλύει την amortized efficiency και των δύο τύπων του αλγορίθμου rank pairing heap.

Η ενότητα 7 παρουσιάζει την έκδοση one-tree για τον αλγόριθμο μας. Ο μετασχηματισμός ο οποίος μετασχηματίζει την δομή δεδομένων σε ένα μοναδικό δέντρο εφαρμόζεται είτε με τον αλγόριθμο rp-heap τύπου 1 είτε με τον αλγόριθμο rp-heap τύπου 2 και διατηρεί τα ιδανικά αποτελέσματα της ενότητας 6.

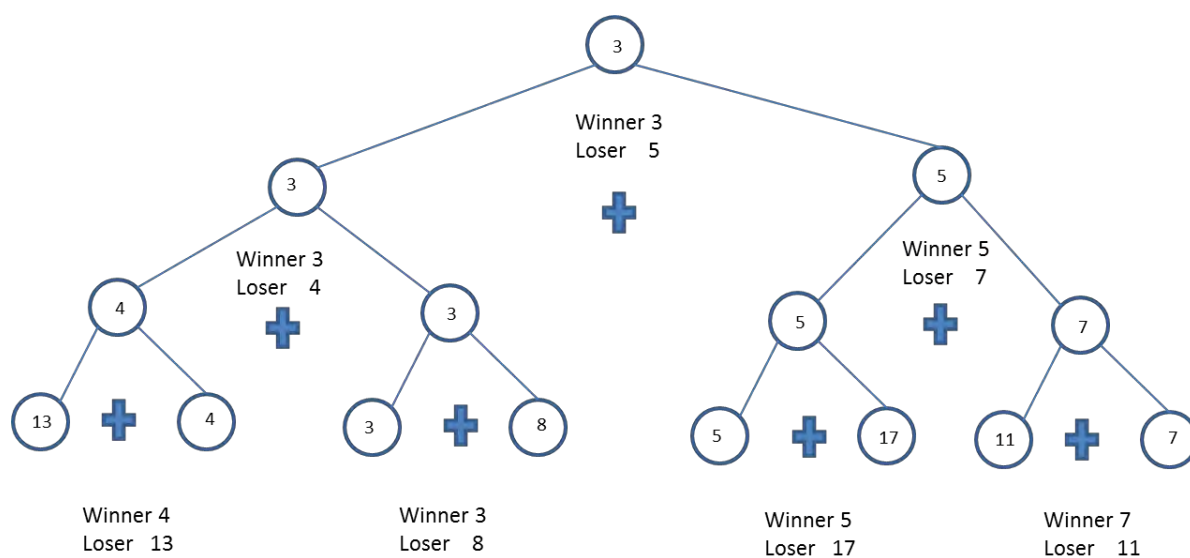
Η ενότητα 8 δείχνει ότι μερικοί ακόμα πιο απλουστευμένοι τρόποι για την μείωση κλειδιού δεν καταφέρνουν να πιάσουν τα όρια του σωρού Fibonacci.

Η ενότητα 9 αναφέρει τα συμπεράσματά μας και παρουσιάζει κάποια ανοιχτά προβλήματα.

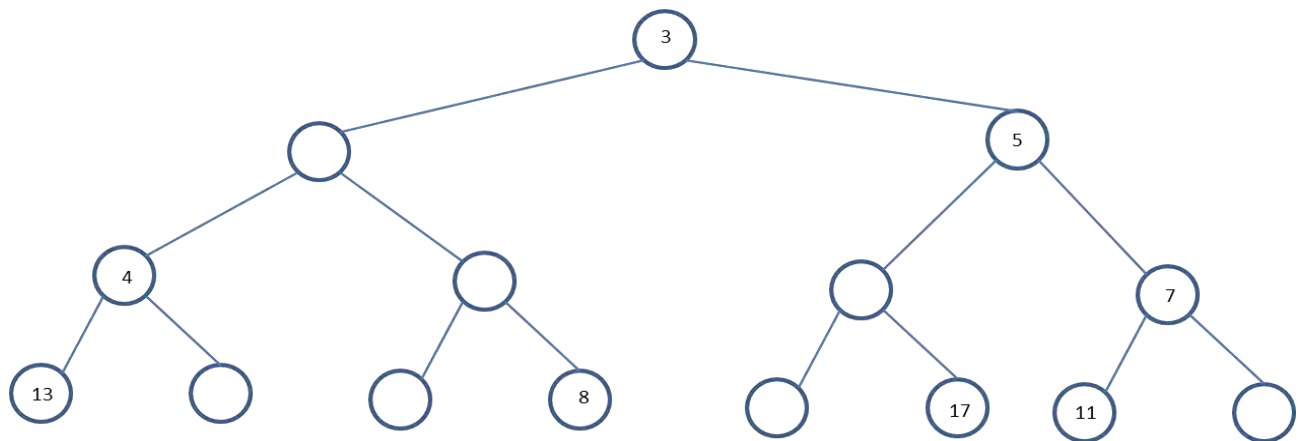
3. Τουρνουά

Η βάση για την δημιουργία πολλών από τους σωρούς που αναφέρονται στην εισαγωγή, όπως βέβαια και για τον αλγόριθμο rank-rairing heaps είναι το τουρνουά single-elimination. Ένα τουρνουά είναι είτε άδαιο, είτε αποτελείται από ένα στοιχείο, τον νικητή (winner), είτε είναι μια φόρμα από δυο στοιχεία που προήλθαν από συνένωση δυο διαφορετικών τουρνουά. Για να συνενωθούν δυο τουρνουά, συνδυάζεται το σύνολο των στοιχείων και των δυο τουρνουά και συγκρίνονται τα κλειδιά των δυο winners. Το στοιχείο με το μικρότερο κλειδί είναι ο νικητής της συνένωσης όποτε και ο νέος winner του νέου τουρνουά που κατασκευάστηκε. Το στοιχείο με το μεγαλύτερο κλειδί είναι ο χαμένος (loser) της συνένωσης. Για να κατασκευαστεί ένα τουρνουά με n στοιχεία, χρειάζονται $n-1$ συγκρίσεις. Ο νικητής του τουρνουά είναι το στοιχείο με το μικρότερο κλειδί.

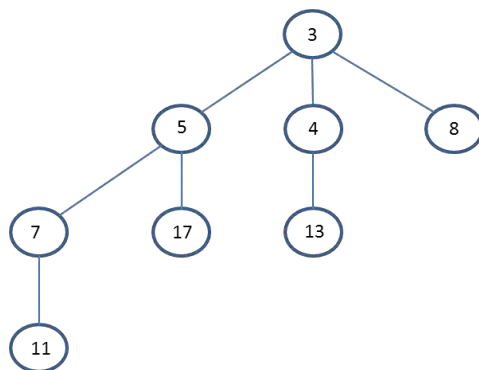
Υπάρχουν τουλάχιστον τέσσερις ισοδύναμοι τρόποι να παρουσιαστεί το ίδιο τουρνουά (σχήμα 3.1). Η Full-representation είναι ένα γεμάτο δυαδικό δένδρο (full binary tree) με ένα φύλλο ανά στοιχείο και ένα μη-φύλλο ανά συνένωση. Κάθε μη-φύλλο περιέχει τον winner της αντίστοιχης συνένωσης. Επιπλέον οι κόμβοι που περιέχουν ένα δεδομένο στοιχείο σχηματίζουν ένα μονοπάτι στο δένδρο, που αποτελείται από ένα φύλλο και τα μη-φύλλα αντιστοιχούν στις συνενώσεις στις οποίες το στοιχείο αυτό κέρδισε. Το δένδρο είναι heap ordered (ταξινομημένος σωρός): το στοιχείο σε έναν κόμβο έχει το ελάχιστο κλειδί μεταξύ όλων των στοιχείων των απογόνων του κόμβου.



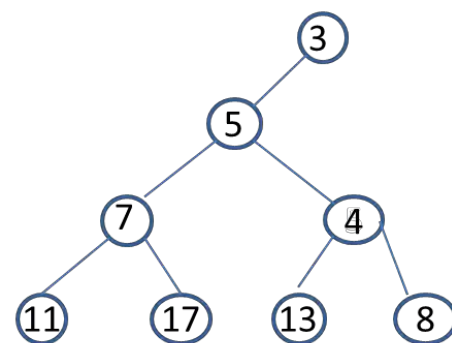
α) full representation tournament



β) half full representation tournament



γ)Heap ordered tournament



δ) Half ordered tournament

Σχήμα 3.1

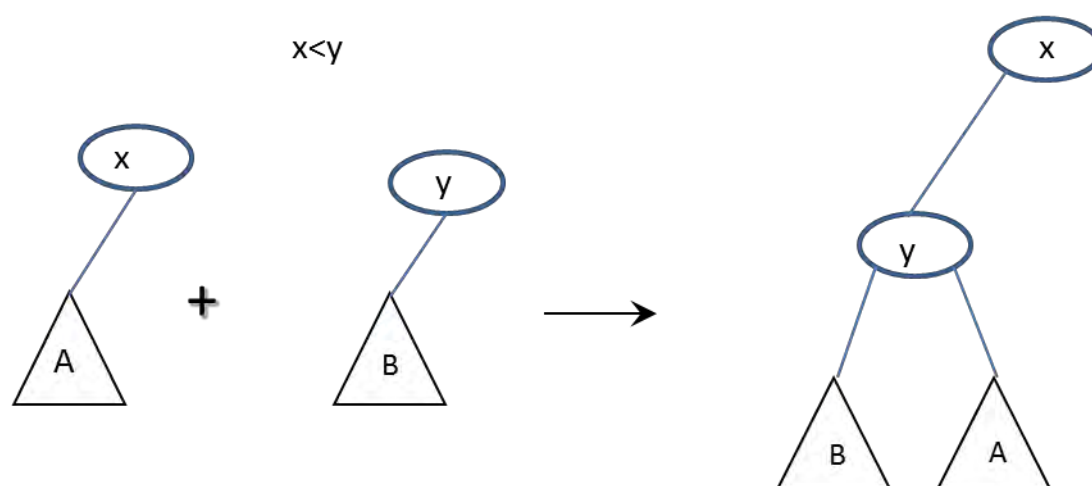
Παίρνουμε την half-full representation ενός τουρνουά από την full representation, αφαιρώντας κάθε στοιχείο το οποίο περιέχεται σε μεγαλύτερο κόμβο. Αυτή η παρουσίαση είναι ένα δυαδικό ταξινομημένος-σωρός δένδρο (binary heap-ordered tree) στο οποίο η ρίζα (root) είναι γεμάτη, κάθε πατέρας έχει ένα γεμάτο και ένα άδειο παιδί και κάθε στοιχείο το βλέπουμε μόνο σε έναν κόμβο.

Και οι δύο παρουσιάσεις τουρνουά (half-full representation, full representation) χρειάζονται $2n-1$ κόμβους για να παρουσιάσουν ένα τουρνουά n -στοιχείων. Ο τρίτος ισοδύναμος τρόπος παρουσίασης ενός τουρνουά είναι η heap-ordered representation. Ο οποίος χρειάζεται μόνο n κόμβους για να παρουσιάσουν ένα τουρνουά n -στοιχείων. Είναι ένα ταξινομημένο δένδρο στο οποίο τα στοιχεία είναι οι κόμβοι και τα παιδιά ενός στοιχείου είναι εκείνοι που έχασαν στις συγκρίσεις. Πρώτη μπαίνει η πιο τελευταία σύγκριση. Το δένδρο είναι heap-ordered αλλά δεν είναι δυαδικό. Πολλές από τις υλοποιήσεις σωρού που αναφέρθηκαν στην εισαγωγή, παρουσίασαν τον σωρό τους με τέτοια μορφή(heap-ordered).

Η 4η παρουσίαση ενός τουρνουά που την ονομάζεται *half-ordered representation*, κατασκευάζεται από την *heap-ordered representation* χρησιμοποιώντας την *binary tree representation* ενός δένδρου: το αριστερό παιδί ενός στοιχείου στην *half-ordered representation* είναι το πρώτο παιδί στην *heap-ordered representation*, και το δεξί παιδί στην *half-ordered representation* είναι ο επόμενος αδερφός (*next sibling*) στην *heap-ordered representation*. Το αποτέλεσμα είναι ένα μισό δένδρο (*half tree*). *Half tree* ονομάζεται ένα δένδρο που η ρίζα του δεν έχει δεξιό υποδένδρο. Το *Half tree* είναι και μισό ταξινομημένο (*half ordered*). Το κλειδί κάθε στοιχείου είναι μικρότερο από όλα τα στοιχεία που υπάρχουν στο αριστερό υποδένδρο. Σε ένα *half tree* που είναι *half ordered* ορίζεται ως ταξινομημένο πρόγονο ενός κόμβου x (ο x να μην είναι η ρίζα) τον πατέρα του πλησιέστερου προγόνου του x ο οποίος είναι αριστερό παιδί. Αυτός ακριβώς είναι ο πατέρας του x στο τουρνουά με παρουσίαση *heap-ordered*.

Πλέον όλα τα δέντρα μας είναι δυαδικά και *half ordered*. Η παρουσίαση *half-ordered* έχει δυο πλεονεκτήματα σε σχέση με την *heap-ordered*. Είναι πιο κοντά σε μια πραγματική εφαρμογή, και ενοποιεί τη πράξη της μείωσης κλειδιού. Όλες, και οι 4, παρουσιάσεις του τουρνουά είναι πλήρως συμβατές, και όλα τα αποτελέσματα μας μπορούν να παρουσιαστούν σε οποιαδήποτε από τις τέσσερις μορφές αν γίνει μια κατάλληλη χαρτογράφηση των δεικτών(*pointers*).

Κατασκευάζουμε ένα δυαδικό δένδρο αποθηκεύοντας σε κάθε κόμβο x , δείκτες να δείχνουν το αριστερό του παιδί και το δεξιό του παιδί, $left(x)$ και $right(x)$, αντίστοιχα. Η δεξιά ραχοκοκαλιά ενός κόμβου σε ένα δυαδικό δένδρο είναι το μονοπάτι από τον κόμβο διαμέσου δεξιών παιδιών μέχρι να φτάσουμε σε φύλλο. Η συνένωση παίρνει την παρακάτω φόρμα στα *half trees* (δες σχήμα 3.2). Συγκρίνουμε τα κλειδιά των δυο ριζών. Αν x, y είναι οι ρίζες με το μικρότερο και το μεγαλύτερο κλειδί αντίστοιχα (x :μικρότερο, y :μεγαλύτερο) αποσπάτε το αριστερό υποδένδρο της ρίζας x και κάντε το δεξιό υποδένδρο της ρίζας y . Έστερα γίνεται το δένδρο με την ρίζα y αριστερό παιδί της ρίζας x . Μια συνένωση χρειάζεται χρόνο $O(1)$.



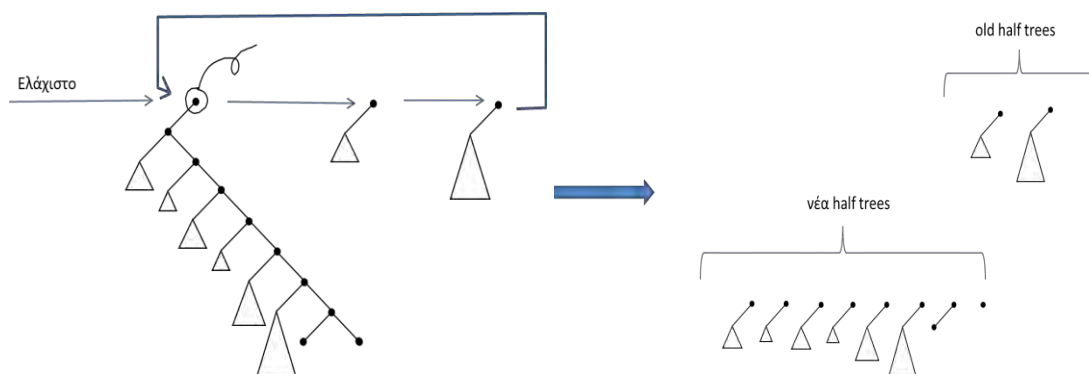
Σχήμα 3.2: συνένωση δυο *half trees* με ρίζες x και y , με x να είναι το μικρότερο κλειδί.

4. Lazy binomial queues

Κατά την μετάβαση στην παρουσίαση half-ordered , πολλές από τις υλοποιήσεις σωρών που αναφέρονται στην εισαγωγή όπως και η rank-pairing είναι επεκτάσεις της επόμενης γενικής υλοποίησης . Ένας σωρός αποτελείται από ένα σύνολο από δένδρα half trees , των οποίων οι κόμβοι είναι τα στοιχεία του σωρού και παρουσιάζονται σαν μια μονή κυκλική λίστα (single-linked circular list) των ριζών των δένδρων. Πρόσβαση στην λίστα των ριζών γίνεται από ένα δείκτη που δείχνει την ρίζα με το μικρότερο κλειδί, το οποίο καλούμε και min-root. Κάντε τις διάφορες πράξεις για τον σωρό, εκτός από μείωση κλειδιού και τη διαγραφή τυχαίου στοιχείου ως εξής:

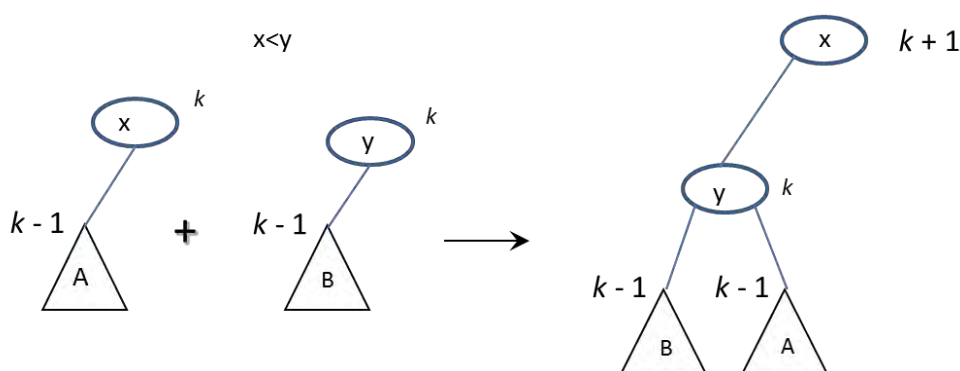
- Για να δημιουργήσετε έναν σωρό , φτιάξτε μια άδεια λίστα από ρίζες.
- Για να εισάγεται ένα στοιχείο στο σωρό , φτιάξτε ένα half tree ενός κόμβου , εισάγεται το στο σωρό μετά την ρίζα με το μικρότερο στοιχείο και κάντε το , min-root αν έχει το μικρότερο κλειδί από όλες τις ρίζες του σωρού.
- Για να συγχωνεύσετε δυο σωρούς , συνδέστε σε σειρά τις δυο λίστες των ριζών και φτιάξτε την παλιά min-root με το μικρότερο κλειδί την νέα min-root του καινούργιου σωρού.
- Για την διαγραφή του μικρότερου στοιχείου , αποσυναρμολογήστε το δένδρο που έχει ρίζα το min-root όπως αναφέρουμε παρακάτω :

Έστω y είναι το αριστερό παιδί του x . Διαγράψτε το x και κόψτε κάθε άκρη της δεξιάς ραχοκοκαλιάς του δένδρου. Αυτό έχει ως αποτέλεσμα κάθε κόμβος της δεξιάς ραχοκοκαλιάς του y να γίνεται ρίζα ενός νέου half tree, διατηρώντας τον εαυτό του και το αριστερό του υποδένδρο (σχήμα 4.1). Εισάγεται τα νέα half trees στα εναπομείναντα half trees του σωρού. Βρείτε την ρίζα με το μικρότερο κλειδί και κάντε το την νέα min-root. Επιπλέον, μετά από κάθε πράξη σωρού , κάντε μηδέν ή περισσότερες συνενώσεις half trees δένδρων για να μειωθεί ο αριθμός τους. Με αυτή την υλοποίηση οι κόμβοι που δεν συμμετείχαν σε συνενώσεις σε ένα δοσμένο κόμβο x είναι ακριβώς αυτές στην δεξιά ραχοκοκαλιά του αριστερού παιδιού του x .



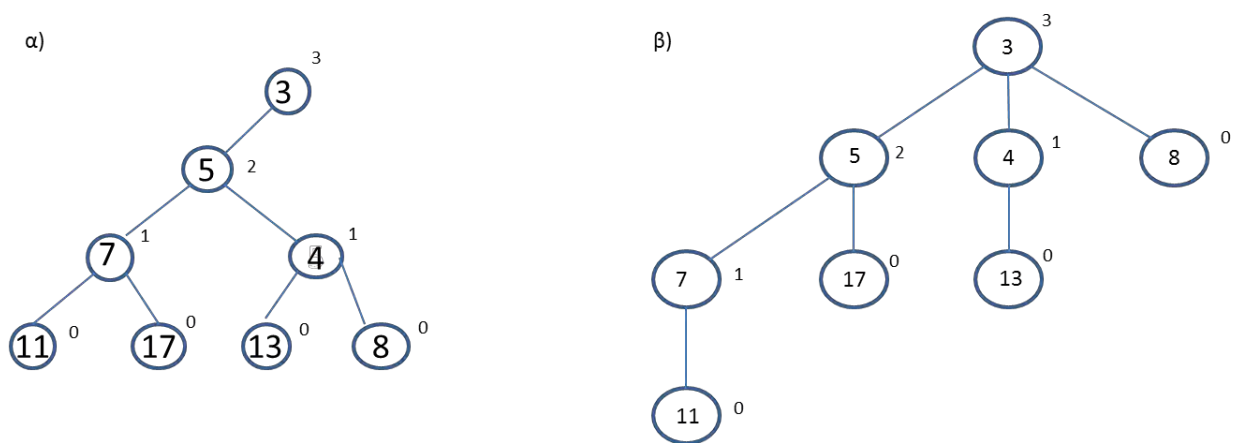
Σχήμα 4.1 : Η αποσυναρμολόγηση κατά τη διάρκεια της διαγραφής ελαχίστου. Κάθε κόμβος στην δεξιά ραχοκοκαλιά του αριστερού παιδιού της ρίζας γίνεται ρίζα στα νέα half trees.

Αυτές οι δομές δεδομένων είναι αποτελεσματικές αν οι συνενώσεις γίνονται προσεκτικά. Στον αλγόριθμο pairing heap, όπου υπάρχουν αρκετές φόρμες, όλες οι ενώσεις είναι από half trees των οποίων οι ρίζες είναι συνεχόμενες στη λίστα των ριζών. Αυτή η μέθοδος δεν είναι αποτελεσματική. Ο Fredman έδειξε ότι για να πιάσουμε τα όρια του σωρού Fibonacci, είναι αναγκαίο να κάνεις πολλές ενώσεις από δένδρα με παρόμοιο μέγεθος. Εκτός από τους pairing heaps όλοι οι άλλοι αλγόριθμοι χρησιμοποιούν μη αρνητικό rank για τους κόμβους ως βοήθεια για το μέγεθος των δένδρων. Ο απλούστερος τρόπος για να χρησιμοποιήσετε ranks είναι ο ακόλουθως. Έστω ότι rank του half tree είναι το rank της ρίζας. Δώστε σε ένα νεοεισερχόμενο στοιχείο, rank ίσο με μηδέν. Ενώστε δυο half trees μόνο αν έχουν το ίδιο rank. Μετά την ένωση αυξήστε το rank του winner κατά ένα. Μην αλλάξετε το rank του loser (σχήμα 4.2).



Σχήμα 4.2 : Συνένωση δυο half trees, με ρίζες x και y , ο x έχει μικρότερο κλειδί. Τα ranks βρίσκονται στα δεξιά κάθε κόμβου.

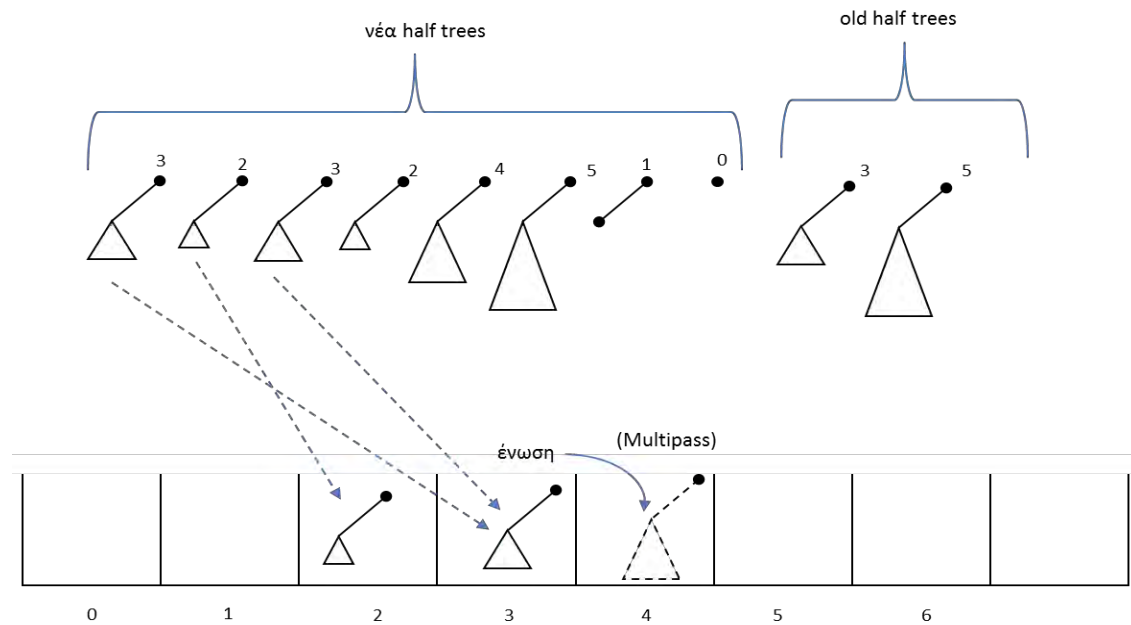
Αν όλες οι ενώσεις γίνουν με αυτόν τον τρόπο όλα τα half trees που είναι στο σωρό είναι τέλεια(perfect). Δηλαδή αποτελούνται από μια ρίζα που το αριστερό του υποδένδρο είναι ένα τέλειο δυαδικό δένδρο (perfect binary tree), κάθε παιδί έχει rank ένα λιγότερο από τον πατέρα του και το δένδρο περιέχει 2^k , όπου k το rank του δένδρου. Επιπλέον το maximum rank είναι $\lg n$. Το αποτέλεσμα αυτής της δομής δεδομένων είναι η δυωνυμική ουρά (binomial queue), έτσι καλείται επειδή στην heap-ordered representation ο αριθμός των κόμβων σε βάθος d σε ένα δένδρο με rank r είναι η δυωνυμικός συντελεστής $\binom{r}{d}$ (σχήμα 4.3)



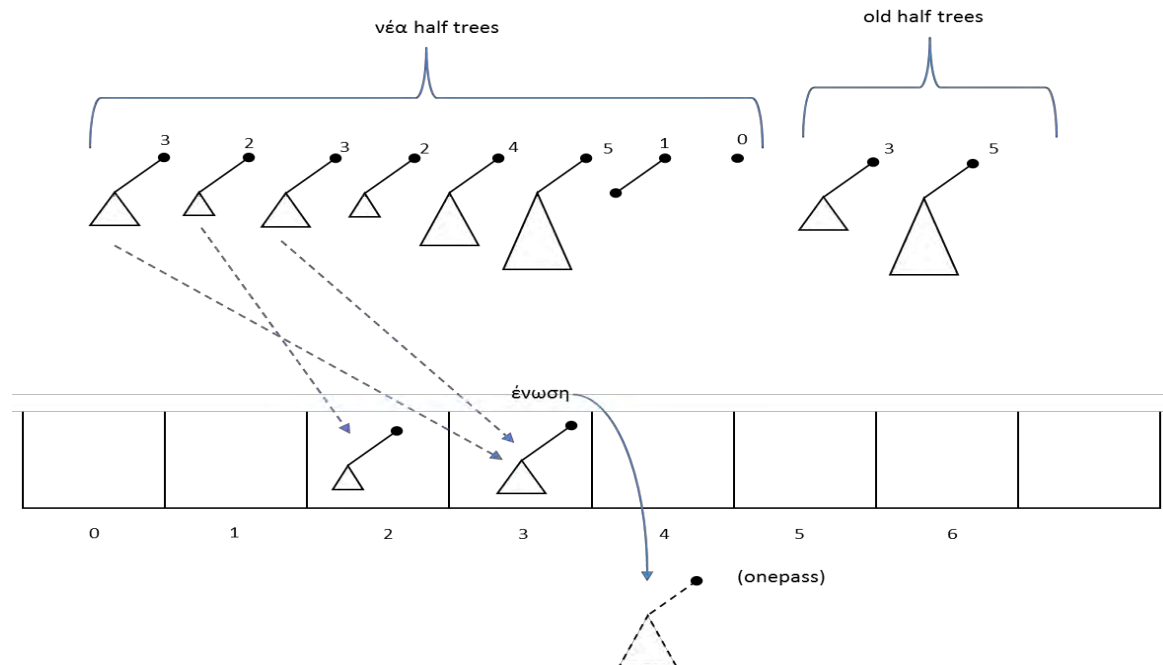
Σχήμα 4.3 : α) half ordered β) heap ordered παρουσιάσεις ενός δένδρου στην δυωνυμική ουρά.

Στην original εκδοχή των δυωνυμικών ουρών, οι ενώσεις γίνονται γρήγορα για να διατηρηθεί αμετάβλητο ότι ο σωρός περιέχει το πολύ μία ρίζα ανά βαθμό rank. Αυτό δίνει χρόνο χειρότερης περίπτωσης, όριο $O(\log n)$ για τις πράξεις εισαγωγή στοιχείου, συνένωση ουρών και διαγραφής ελαχίστου. Κάνοντας ενώσεις τεμπέλικα (lazy) και ειδικά μόνο όταν έχουμε διαγραφή ελαχίστου στοιχείου πετυχαίνουμε καλύτερη amortized efficiency. Μία μέθοδος, που χρησιμοποιείται στον σωρό Fibonacci και σε όλες τις άλλες δομές που είναι παρόμοιες, είναι να κάνουμε όσες ενώσεις είναι δυνατόν μετά από μια διαγραφή ελαχίστου, αφήνοντας μια ρίζα ανά βαθμό rank. Αυτή η μέθοδος που καλείται multipass linking δουλεύει αρκετά καλά. Μια εναλλακτική μέθοδος επίσης στη λογική τεμπέλικων ενώσεων η οποία ονομάζεται one-pass linking επίσης δουλεύει αρκετά καλά. Μετά από μια διαγραφή ελαχίστου βάζουμε σε μια φόρμα ένα maximum αριθμό από ζευγάρια half trees με ίδιο rank, και ενώνουμε αυτά τα ζευγάρια και τίποτα άλλο. Αυτή η μέθοδος ενώσεων μοιάζει με αυτή που χρησιμοποιείται στον αλγόριθμο pairing heap. Ονομάζεται μια δυωνυμική ουρά με multipass linking, multipass binomial queue και αυτή με την μέθοδο one-pass, την ονομάζουμε one-pass binomial queue.

Για την υλοποίηση είτε του one-pass linking είτε του multipass linking, διατηρήστε ένα σύνολο από κάδους (buckets), έναν για κάθε rank. Κατά τη διάρκεια μιας διαγραφής ελαχίστου, επεξεργαστείτε τα half trees, πρώτα αυτά που κατασκευάστηκαν από την αποσυναρμολόγηση και μετά τα υπόλοιπα δένδρα του σωρού. Η επεξεργασία ενός half tree γίνεται ως εξής: προσθέστε το στον κάδο με το κατάλληλο rank αν ο κάδος είναι άδειος. Αν δεν είναι άδειος κάντε συνένωση με το δένδρο που βρίσκεται στον κάδο και αφήστε τον κάδο άδειο. Το νέο half tree που δημιουργήθηκε από την συνένωση προσθέστε το στο νέο σωρό που δημιουργήθηκε αν εφαρμόζετε η μέθοδος one-pass linking, η προσθέστε το στον κάδο με αυξημένο rank κατά ένα αν εφαρμόζετε η μέθοδος multipass linking (σχήμα 4.4 α, β). Κατά τη διάρκεια της διαδικασίας παρακολουθείτε τους μη κενούς κάδους. Όταν όλα τα half trees έχουν περάσει το στάδιο της επεξεργασίας, όσα half trees έχουν μείνει στους κάδους προσθέστε τα, στη λίστα των δένδρων του νέου σωρού, αφήνοντας όλους τους κάδους άδειους.



Σχήμα 4.4(α): Ένωση κατά την διάρκεια minimum deletion χρησιμοποιώντας ένα σύνολο από buckets, ένα για κάθε rank. Τα νέα half trees από τη αποσυναρμολόγηση ξεκινούν πρώτα, και ακολουθούν τα υπόλοιπα. Μετά την ένωση το νέο half tree(από την ένωση) στην multipass διαδικασία μεταπηδά στο bucket με το αμέσως υψηλότερο rank.



Σχήμα 4.4(β): Ένωση κατά την διάρκεια minimum deletion χρησιμοποιώντας ένα σύνολο από buckets, ένα για κάθε rank. Τα νέα half trees από τη αποσυναρμολόγηση ξεκινούν πρώτα, και ακολουθούν τα υπόλοιπα. Μετά την ένωση το νέο half tree(από την ένωση) στην one pass διαδικασία προστίθεται στην λίστα εξόδου (output list) του νέου σωρού.

Παρόλο που η υλοποίηση της συνένωσης φαίνεται να χρειάζεται την χρησιμοποίηση array, είναι εύκολο να υλοποιηθεί με ένα μοντέλο με pointers, όπως παρατηρήθηκε από τον Fredman και τον Tarjan. Ο Fredman και ο Tarjan πρότειναν μια διπλά συνδεδεμένη λίστα (double-linked list) από κόμβους με ranks (rank nodes), έναν για κάθε πιθανό rank, από το μηδέν μέχρι το μεγαλύτερο rank. Για κάθε i , ο κόμβος με βαθμό rank i , δείχνει στους κόμβους με rank $i-1$ και $i+1$, και στον κάδο για βαθμό rank i .

Για να αναλυθεί η one-pass και η multipass binomial queue, ορίζετε το δυναμικό ενός σωρού να είναι διπλάσιο από τον αριθμό των half trees δένδρων.

Θεώρημα 4.1 : Ο amortized time για μια πράξη στην one-pass και στην multipass binomial queue είναι $O(1)$ για τις πράξεις δημιουργία σωρού, εύρεση μικρότερου, εισαγωγή ή συνένωση σωρών και $O(\log n)$ για την πράξη της διαγραφής ελαχίστου.

Απόδειξη: Η δημιουργία σωρού, εύρεση μικρότερου, εισαγωγή ή συνένωση σωρών χρειάζονται $O(1)$ πραγματικό χρόνο. Από αυτές τις λειτουργίες μόνο η εισαγωγή αυξάνει το δυναμικό, κατά δυο. Έτσι κάθε μια από αυτές τις λειτουργίες χρειάζεται $O(1)$ amortized time. Ας εξεταστεί η διαγραφή ελαχίστου. Τα ακόλουθα επιχειρήματα ισχύουν και για την one-pass και την multipass linking. Η αποσυναρμολόγηση του half tree του δένδρου που έχει ρίζα τον min-root, αυξάνει τον αριθμό των half trees το πολύ κατά $\lg n$ και επομένως το δυναμικό κατά $2\lg n$. Έστω h είναι ο αριθμός των half trees μετά την αποσυναρμολόγηση. Ολόκληρη η διαγραφή ελαχίστου παίρνει h συγκρίσεις κλειδιών και χρόνο $O(h+1)$. Κλιμακώστε αυτόν τον χρόνο να είναι το πολύ $h+O(1)$ (Αυτό είναι ισοδύναμο με τον πολλαπλασιασμό του δυναμικού κατά ένα σταθερό συντελεστή). Κάθε συνένωση μετά την αποσυναρμολόγηση, μειώνει το δυναμικό κατά δυο. Το πολύ $(\lg n + 1)$ half trees δεν συμμετέχουν σε μια συνένωση, άρα υπάρχουν το λιγότερο $(h - \lg n - 1)/2$ ενώσεις. Έτσι η διαγραφή ελαχίστου αυξάνει το δυναμικό το πολύ $\lg n - h + 1$, δίνοντας amortized time $O(\log n)$ και το πολύ $3\lg n$ amortized συγκρίσεις κλειδιών.

Έχει συμπεριληφτεί εδώ το one-pass linking, για να απεικονίσουν το φάσμα της αποτελεσματικότητας στην υλοποίηση των δυωνυμικών ουρών. Η μέθοδος one-pass linking είναι τεμπέλικη (lazier) αλλά παράγει μακρύτερες λίστες με ρίζες από ότι η μέθοδος multipass linking. Αυτό επιβραδύνει τη διαγραφή ελαχίστου. Μπορούμε να γίνει αυτή την παρατήρηση ποσοτική: Με μια συνάρτηση δυναμικού ίση με τον αριθμό των half trees, ένα όρισμα όπως αυτό στην απόδειξη του θεωρήματος 4.1 δείχνει ότι το multipass linking κάνει το πολύ $2\lg n$ amortized συγκρίσεις κλειδιών (key comparisons) ανά διαγραφή ελαχίστου, λιγότερες σε σχέση με το one-pass linking κατά ένα συντελεστή $3/2$.

Μια μέθοδος η οποία μπορεί να είναι καλύτερη και από την multipass linking και από την one-pass linking είναι να διατηρηθεί ο σωρός ως ένα half tree, αποφεύγοντας τον πονοκέφαλο της λίστας των ριζών μια και καλή. Μια τέτοια αναπαράσταση κωδικοποιεί όλες τις συγκρίσεις κλειδιών στη δομή δεδομένων ενώ σε μια αναπαράσταση με πολλά δένδρα η σύγκριση κλειδιών γίνεται για να διατηρήσουμε ότι η min-root δεν κωδικοποιείται στην δομή, και τα αποτελέσματά τους χάνονται μετά από κάθε διαγραφή ελαχίστου.

Διατηρώντας τον σωρό ως ένα μοναδικό half tree δένδρο απαιτείται συνένωση half trees με διαφορετικό rank. Ονομάζεται μια τέτοια ένωση unfair , ενώ ονομάζεται μια ένωση δυο half trees με ίδιο rank fair. Μετά από μια ένωση unfair, αφήστε τα ranks και του winner και του loser όπως έχουν. Η ενώσεις unfair δεν επηρεάζουν δυσμενώς την amortized efficiency της δομής δεδομένων , αν βεβαίως γίνουν όσο το δυνατό λιγότερες.

Η one-tree binomial queue αποτελείται από ένα half tree. Για να βρείτε το μικρότερο , επιστρέψτε την ρίζα. Για να δημιουργήσουμε έναν σωρό , δημιουργήστε ένα άδειο half tree. Για την εισαγωγή νέου στοιχείου , δημιουργήστε ένα half tree με έναν κόμβο , με rank μηδέν και ένωσε το με το υπάρχων half tree. Για την συγχώνευση δυο σωρών , απλά συνενώστε τα δυο half tree δένδρα τους. Για την διαγραφή ελαχίστου , διαγράψτε την ρίζα και αποσυναρμολογήστε το half tree και δημιουργήστε half tree με ρίζες κάθε κόμβο στην δεξιά ραχοκοκαλιά του παλιού αριστερού παιδιού της ρίζας που διαγράφηκε. Συνεχίζουμε με την συνένωση των νέων δένδρων με την μέθοδο του multipass linking. Όταν όλα τα half trees είναι σε κάδους (buckets) τα αφαιρούνται από τους κάδους και συνενώνονται με οποιαδήποτε σειρά (με fair και unfair links) έως ότου μείνει ένα half tree.

Λήμμα 4.2. Ένα one-tree binomial queue με rank k περιέχει τουλάχιστον 2^k κόμβους. Ως εκ τούτου, $k \leq \lg n$.

Απόδειξη: Αποδεικνύεται το λήμμα με επαγωγή στον αριθμό των ενώσεων και των αποσυναρμολογήσεων του half tree. Ένα νέο με έναν κόμβο half tree, έχει rank μηδέν όποτε ικανοποιείται το λήμμα. Μια fair ένωση , ενώνει δυο half trees με ίδιο rank, έστω k , σε ένα half tree με rank $k + 1$. Σύμφωνα με την επαγωγική υπόθεση κάθε συστατικό half tree περιέχει το πολύ 2^k κόμβους , άρα το ενωμένο δένδρο θα περιέχει το πολύ 2^{k+1} κόμβους, οπότε ικανοποιεί το λήμμα. Μια unfair ένωση , ενώνει δυο half trees με διαφορετικό rank , έστω j και k , με $j < k$, σε ένα half tree με rank το πολύ k . Σύμφωνα με την επαγωγική υπόθεση το half tree με rank k , περιέχει το πολύ 2^k κόμβους , άρα το ενωμένο δένδρο ικανοποιεί το λήμμα. Μια αποσυναρμολόγηση ενός half tree αναιρεί όλες τις ενώσεις που νίκησε η ρίζα και η οποία διαγράφηκε. Αφού τα προκύπτον half trees ικανοποιούν το λήμμα όταν δημιουργήθηκαν, θα ικανοποιούν το λήμμα και μετά την αποσυναρμολόγηση.

Για να αναλυθεί ο αλγόριθμος one-tree binomial queues ορίζουμε το δυναμικό ενός κόμβου να είναι μηδέν αν είναι ο loser μιας fair ένωσης ή ένα διαφορετικά (αυτό είναι μια ρίζα ή είναι ο loser μιας unfair ένωσης). Ορίζουμε το δυναμικό ενός σωρού να είναι το άθροισμα των δυναμικών των κόμβων του . Μια διαγραφή ελαχίστου αναιρεί όλες τις ενώσεις που νίκησε ο κόμβος ο οποίος διαγράφηκε. Ο loser μιας τέτοιας ένωσης γίνεται ρίζα και σταματά πλέον να είναι loser. Έτσι το δυναμικό του αυξάνεται κατά ένα , αν η ένωση είναι fair και δεν μεταβάλλεται αν η ένωση είναι unfair.

Θεώρημα 4.3 : Ο amortized time για μια πράξη στην one-tree binomial queue είναι $O(1)$ για τις πράξεις , δημιουργία σωρού , εύρεση μικρότερου , εισαγωγή η συνένωση σωρών και $O(\log n)$ για την πράξη της διαγραφής ελαχίστου.

Απόδειξη : Η ανάλυση των πράξεων εύρεση ελαχίστου, δημιουργία σωρού, εισαγωγή στοιχείου , συγχώνευση σωρών είναι ίδιες με την ανάλυση που έγινε προηγουμένως για τον multipass binomial queue, εκτός ότι σε κάθε εισαγωγή στοιχείου και σε κάθε συγχώνευση σωρών γίνεται ένωση δένδρων. Θεωρήστε μια διαγραφή ελαχίστου. Η αποσυναρμολόγηση του half tree αναιρεί όλες τις ενώσεις στις οποίες νίκησε η ρίζα. Αυτό αυξάνει το δυναμικό κατά ένα για κάθε ένωση fair που νίκησε η ρίζα. Κάθε τέτοια ένωση αυξάνει το rank της ρίζας όταν αυτή συμβαίνει. Σύμφωνα με το λήμμα 4.2 υπήρχαν το πολύ $\lg n$ τέτοιες ενώσεις , έτσι η αποσυναρμολόγηση αυξάνει το δυναμικό το πολύ κατά $\lg n$. Έστω h είναι ο αριθμός των δένδρων μετά την αποσυναρμολόγηση. Ολόκληρη η διαγραφή ελαχίστου χρειάζεται $h-1$ συγκρίσεις κλειδιών και παίρνει χρόνο $O(h + 1)$. Κλιμακώστε αυτόν τον χρόνο να γίνει το πολύ $h+O(1)$. Κάθε fair ένωση μετά την αποσυναρμολόγηση μειώνει το δυναμικό κατά ένα , κάθε unfair ένωση δεν αλλάζει το δυναμικό. Υπάρχουν το πολύ $\lg n$ unfair ενώσεις , οπότε υπάρχουν το λιγότερο $(h - \lg n - 1)$ fair ενώσεις. Έτσι η διαγραφή ελαχίστου αυξάνει το δυναμικό κατά το πολύ κατά $2\lg n - h + 1$, δίνοντας έναν amortized χρόνο του μεγέθους $O(\log n)$ και το πολύ $2\lg n$ amortized συγκρίσεις κλειδιών.

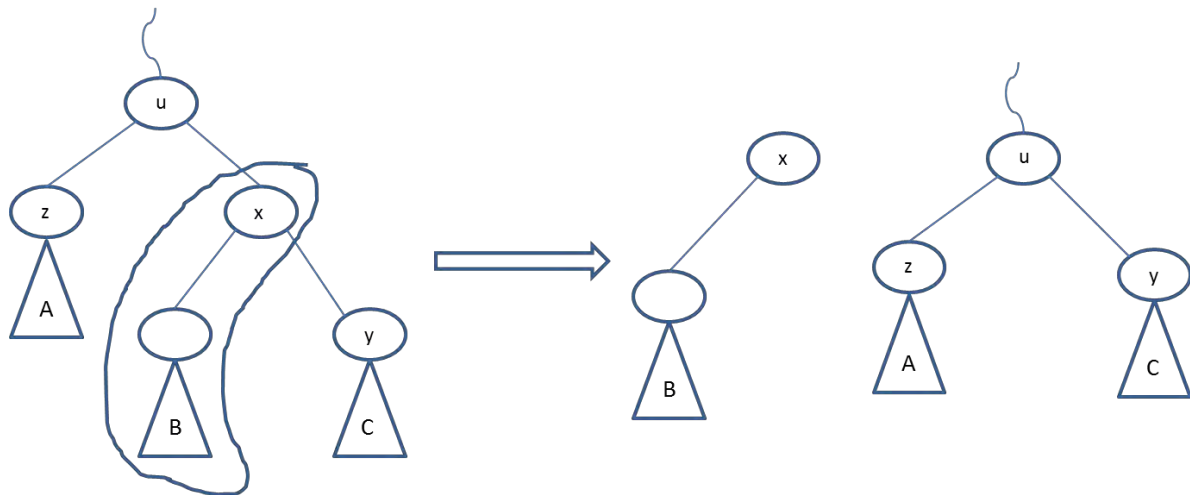
5. Rank-pairing heaps

Ο βασικός μας στόχος είναι να υλοποιηθεί η μείωση κλειδιού ώστε αυτή να θέλει amortized χρόνο $O(1)$. Όταν η μείωση κλειδιού θα υποστηρίζεται από τον αλγόριθμο θα μπορεί εύκολα να υποστηρίζεται και η πράξη της διαγραφής τυχαίου κόμβου, με την λογική να γίνει μείωση κλειδιού με νέα τιμή για το στοιχείο $-\infty$, να γίνει αυτό το στοιχείο min-root και μετά να γίνει η διαγραφή ελαχίστου. Μια παράμετρος και για τις δυο αυτές πράξεις είναι ότι πρέπει ο σωρός να έχει αυτό το στοιχείο. Αν η εφαρμογή δεν μας παρέχει αυτή την πληροφορία και γίνει η πράξη της συγχώνευσης (meld) τότε χρειάζεται ένα ξεχωριστό σύνολο από disjoint δομών δεδομένων, για να διατηρηθεί το partition των στοιχείων στους σωρούς. Με μια τέτοια δομή δεδομένων ο χρόνος για να βρούμε αν ο σωρός περιέχει ένα δεδομένο στοιχείο είναι μικρή αλλά όχι $O(1)$.

Θα επεκτείνουμε τον multipass binomial queues να υποστηρίζει μείωση κλειδιού. Αυτή η επέκταση επίσης δουλεύει και με τον one-pass binomial queues. Ονομάζουμε το αποτέλεσμα της δομής δεδομένων rank pairing heap. Αναπτύσσουμε δυο τύπος rp-heaps: ο type-1 είναι πιο απλός αλλά πιο δύσκολος στην ανάλυση και έχει μεγαλύτερη σταθερά κλάσματος στα χρονικά όρια και ο type-2 που είναι μια version πιο χαλαρή, η οποία είναι πιο εύκολη στην ανάλυση και έχει μικρότερη σταθερά κλάσματος στα χρονικά όρια.

Για να υλοποιηθεί η μείωση κλειδιού, προσθέτουμε δείκτη για τον πατέρα (parent pointer) στα half trees, ώστε να έχουμε 3 δείκτες ανά κόμβο αντί για δυο. Όπως παρατηρήθηκε από Fredman et al., δυο δείκτες ανά κόμβο είναι αρκετοί: κάθε κόμβος δείχνει στο αριστερό του παιδί ή στο δεξιό του παιδί αν δεν έχει αριστερό παιδί, και στο δεξιό αδερφό, ή στο πατέρα αν δεν έχει δεξιό αδερφό. Αυτή η εναλλακτική διαπραγματεύεται χρόνο με χώρο. Αν ένας κόμβος έχει μόνο ένα παιδί, το βλέπουμε ως αριστερό παιδί αν έχει μεγαλύτερο κλειδί ή ως δεξιό παιδί αν έχει μικρότερο κλειδί. Έτσι αποφεύγετε η ανάγκη για ένα extra bit για την αποσαφήνιση αυτών των δυνατοτήτων. Αυτό μπορεί να μετατρέψει το αριστερό παιδί σε δεξιό παιδί, αλλά δεν επηρεάζει ούτε την ορθότητα ούτε την ανάλυση της δομής δεδομένων. Η ίδια ιδέα εφαρμόζεται και στον pairing heap.

Μόλις η δομή δεδομένων υποστηρίζει τη γονική πρόσβαση (parental access) είναι δυνατό να μειωθεί το κλειδί του στοιχείου x στον σωρό H ως ακολούθως (σχήμα 5.1): Μειώστε το κλειδί του x . Αν το x δεν είναι ρίζα, τότε το x μπορεί να παραβιάζει την half order. Για να ανακτηθείτο half order, δημιουργήστε ένα νέο half tree με ρίζα το x , αποσπώντας το subtree με ρίζα το x , και για το $y = \text{right}(x)$ γίνεται επανατοποθέτηση του subtree με ρίζα το y στο σημείο που βρισκόταν πριν το υποδένδρο με ρίζα το x . Το x με το υποδένδρο του τοποθετείται στο σωρό, δηλαδή στη λίστα με τις ρίζες. Αυτή τη διαδικασία ονομάζετε cut στο x . Είτε ο x ήταν αρχικά ρίζα είτε όχι, κάντε το min-root αν το κλειδί του είναι το μικρότερο στο σωρό.



Σχήμα 5.1: Restructuring κατά την μείωση κλειδιού.

Παρατήρηση. Αν ο x δεν είναι αρχικά ρίζα, δεν υπάρχει κανένας τρόπος στην παρουσίαση μας να τεστάρουμε σε χρόνο $O(1)$ εάν μειώνοντας το κλειδί του x έχουμε παραβίαση στο half tree : μια τέτοια δοκιμή απαιτεί πρόσβαση στο διατεταγμένο πρόγονο (ordered ancestor) του x . Έτσι γίνεται το x ρίζα, είτε υπάρχει παραβίαση είτε δεν υπάρχει.

Αυτή η υλοποίηση είναι σωστή, αλλά καταστρέφει την efficiency της δομής δεδομένων: υπάρχουν αυθαίρετα μακριές ακολουθίες πράξεων που χρειάζονται χρόνο $\Omega(n)$ ανά πράξη. Το πρόβλημα είναι ότι μια μείωση κλειδιού μπορεί να αφαιρέσει ένα αυθαίρετο half tree, και μία ακολουθία τέτοιων αφαιρέσεων μπορεί να παράγει ένα half tree του οποίου το rank είναι $\omega(\log n)$. Για να διατηρηθεί η efficiency χρειαζόμαστε έναν τρόπο να διασφαλίσουμε ότι τα ranks παραμένουν $O(\log n)$, δηλαδή χρειάζεται $O(1)$ amortized time ανά μείωση κλειδιού.

Στο σωρό Fibonacci, η λύση είναι να κάνεις μια ακολουθία από cuts μετά από μια μείωση κλειδιού, αλλά μόνο $O(1)$ amortized χρόνο ανά μείωση. Αυτά τα cuts συμβαίνουν κατά μήκος μιας διαδρομής προγόνων στην heap-ordered representation. Εφόσον ο πατέρας του κόμβου x στην heap-ordered representation είναι ο διατεταγμένος πρόγονος (ordered ancestor) στην παρουσίαση half-ordered, η υλοποίηση αυτής της πράξης απαιτεί ένα extra σετ δεικτών για να διατάξουμε τους προγόνους. Αυτός είναι ένας λόγος γιατί ο σωρός Fibonacci δεν αποδίδει καλά στην πράξη.

Υπάρχουν πολλοί άλλοι τρόποι για να επιτευχθεί ο ίδιος στόχος: Χρησιμοποιείται μια γνωστή ισορροπημένη δομή δέντρου όπως τα AVL trees, ή τα red-black trees. Επινοήστε ένα νέο κανόνα ισορροπίας, όπως ο Høyer στον αλγόριθμό του τον one-step heaps (που ονομάζεται και thick heaps) ή στον αλγόριθμο thin heaps [24]. Άλλος τρόπος είναι να γίνουν περισσότερες global ανακατασκευές όπως στον αλγόριθμο quake heaps. Μία άλλη προσέγγιση, που χρησιμοποιούνται στους relaxed heaps, είναι να επιτρέπουμε την παραβίαση του half order. Έτσι η μείωση κλειδιού δεν χρειάζεται άμεσα να κάνουμε κάτι, απλά παράγει ακόμα μια παραβίαση (violation). Για να διατηρήσουμε την efficiency, το σύνολο των παραβιάσεων πρέπει να ελέγχεται κατά κάποιο τρόπο, οπότε απαιτείται η περιοδική αναδιάρθρωση ώστε να μειωθεί το σύνολο των παραβιάσεων.

Όλες αυτές οι μέθοδοι έχουν ένα πράγμα κοινό: κάνουν επιπλέον αναδιάρθρωση ώστε να διατηρούν κάποια κατάσταση ισορροπίας. Παραδόξως, δεν χρειάζεται να υπάρχει μια τέτοια κατάσταση ισορροπίας. Δεν είναι απαραίτητη: αρκεί μόνο η ενημέρωση στα ranks, συγκεκριμένα να μειωθούν τα ranks κάποιων προγόνων του κόμβου x που το κλειδί υπέστη μείωση. Η μόνη αναδιάρθρωση είναι το cut στο x . Μια ακολουθία από μειώσεις κλειδιών μπορεί να δημιουργήσει half trees με αυθαίρετη δομή, αλλά τα ranks παραμένουν λογαριθμικά όποτε διατηρούμε την αποτελεσματικότητα (efficiency) του αλγορίθμου.

Συμβολίζετε με $p(x)$ τον πατέρα του x και με $r(x)$ το rank του κόμβου x . Υιοθετείτε η σύμβαση ότι το rank ενός παιδιού που λείπει είναι -1 . Αν ο x είναι παιδί, η rank difference είναι $\Delta r(x) = r(p(x)) - r(x)$. Ένα παιδί με rank difference i είναι ένα i -child. Μια ρίζα της οποίας το αριστερό παιδί είναι ένα i -child, είναι μια i -node. Μια μη-ρίζα της οποίας τα παιδιά είναι ένα i -child και ένα j -child είναι ένας i,j -node. Αυτοί οι ορισμοί εφαρμόζονται ακόμη και εάν το αριστερό παιδί της ρίζας, ή ένα ή και τα δυο παιδιά ενός κόμβου (όχι ρίζα) λείπουν. Ο ορισμός για έναν i,j -node δεν κάνει διάκριση μεταξύ αριστερού και δεξιού παιδιού. Σε έναν δυωνυμικό σωρό, εκτός της εκδοχής one-tree κάθε ρίζα είναι 1-node κάθε μη-ρίζα είναι ένα 1,1-node. Θα χαλαρώσει το δεύτερο μισό αυτής της συνθήκης.

Η καίρια παρατήρηση είναι ότι κάθε κόμβος έχει έναν αριθμό από απογόνους τουλάχιστον εκθετικό στο rank τους, ακόμη και αν επιτραπούν 0,i-nodes για αυθαίρετα i εκτός από τους 1,1-nodes. Με αυτό κατά νου, εισάγετε τον type-1 rank rule: κάθε ρίζα είναι 1-node και κάθε παιδί είναι ένας 1,1-node ή ένας 0,i-node για κάποιο $i > 0$ (ενδεχομένως διαφορετικό για κάθε κόμβο). Ο type-1 rp-heap είναι ένα σετ από heap-ordered half trees των οποίων οι κόμβοι έχουν ranks τα οποία υπακούουν στον type-1 rank rule. Τα ranks δίνουν ένα εκθετικό κάτω φράγμα (αλλά όχι άνω φράγμα) στα μεγέθη των subtrees.

Λήμμα 5.1. Σε σωρό type-1 rp-heap κάθε κόμβος με rank k έχει τουλάχιστον 2^k απογόνους συμπεριλαμβανομένου του ιδίου, τουλάχιστον $2^{k+1} - 1$ ένα είναι παιδί. Ως εκ τούτου, $k \leq \lg n$.

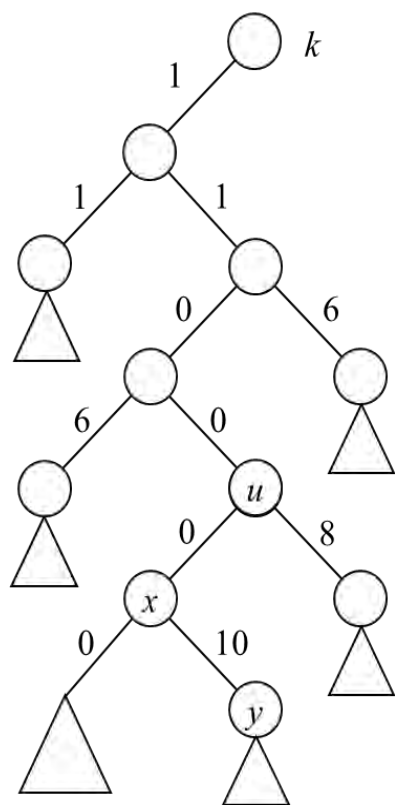
Απόδειξη: Το δεύτερο μέρος του λήμματος αν αποδειχθεί, αποδεικνύει αυτόματα το πρώτο και το τρίτο μέρος. Αποδεικνύετε το δεύτερο μέρος με επαγωγή στο ύψος ενός κόμβου. Ένα φύλλο έχει μηδενικό rank και πληροί το δεύτερο μέρος. Έστω x είναι ένα παιδί με rank k του οποίου τα παιδιά πληρούν το δεύτερο μέρος. Αν x είναι ένας 0,i-node, το παιδί 0-child έχει $2^{k+1} - 1$ απογόνους σύμφωνα με την επαγωγική υπόθεση. Έτσι κάνει και το x . Αν ο x είναι ένας 1,1-node, ο x έχει $2(2^k - 1) + 1 = 2^{k+1} - 1$ απογόνους σύμφωνα με την επαγωγική υπόθεση.

Οι πράξεις εύρεση ελαχίστου, δημιουργία σωρού, εισαγωγή στοιχείου, συγχώνευση σωρών και διαγραφή ελαχίστου είναι ίδιες στον type-1 rp-heaps όπως είναι στον multipass binomial queue, εκτός από ένα αλλαγή στην διαγραφή ελαχίστου: κατά τη διάρκεια της αποσυναρμολόγησης του half tree, δώσε σε κάθε νέα ρίζα rank που είναι κατά ένα μεγαλύτερο από το rank του αριστερού παιδιού της. Δεδομένου ότι κάθε ένωση είναι fair, κάθε ένωση διατηρεί τον rank rule: ο loser γίνεται ένας 1,1-node.

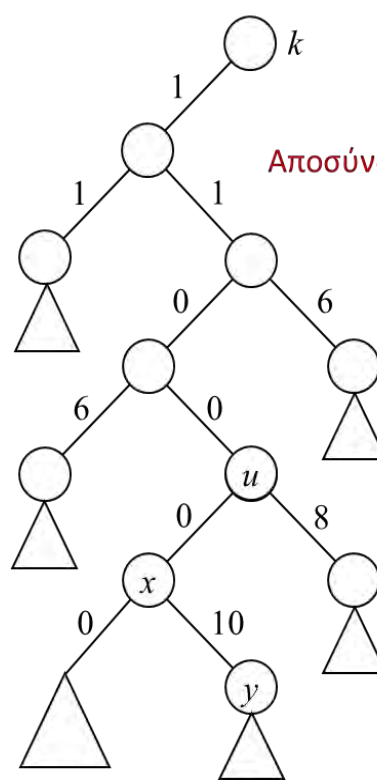
Για την μείωση κλειδιού ενός κόμβου x , προχωρήστε ως εξής (βλέπε σχήμα 5.2): Μειώστε το κλειδί του x . Αν ο x είναι ρίζα, κάντε την min-root αν το κλειδί της είναι πλέον το μικρότερο στο σωρό. Αν ο x δεν είναι ρίζα, αποσυνδέστε τα subtrees που έχουν ως ρίζα το x και το δεξιό του παιδί, το y , επανασυνδέστε το subtree με ρίζα το y στη θέση που κάποτε ήταν το x , και το x βάλτε το στη λίστα των ριζών του σωρού, κάνοντας το min-root αν έχει το πιο μικρό κλειδί. Ολοκληρώνετε με την αποκατάσταση του rank rule: κάντε το rank του x ένα μεγαλύτερο από το rank του αριστερού του παιδιού, και, ξεκινώντας από τον νέο πατέρα του y προχωρήστε δια μέσου των προγόνων μειώνοντας τα ranks του ώστε να υπακούει τον rank rule, μέχρι να φτάσει στη ρίζα ή να φτάσει σε έναν κόμβο στον οποίο δεν χρειάζονται μειώσεις στα ranks. Για να γίνει την μείωση ranks έστω $u = p(y)$ και επαναλάβετε το επόμενο βήμα μέχρι να τελειώσει :

- Type-1 rank-reduction step: Αν u είναι ρίζα, θέσε $r(u) = r(\text{left}(u)) + 1$ και σταμάτα. Διαφορετικά, έστω v και w είναι τα παιδιά του u . Έστω $k = \max\{r(v), r(w)\}$ αν $r(v) \neq r(w)$, $k = r(v) + 1$ αν $r(v) = r(w)$. Εάν $k \geq r(u)$, σταματάμε. Διαφορετικά, το $r(u) = k$ και το $u = p(u)$.

Παρατήρηση: Σε έναν multipass rp-heap, το $k \leq r(u)$ σε κάθε βήμα μείωσης του rank. Αυτό δεν ισχύει στην περίπτωση του one-tree rp-heaps.

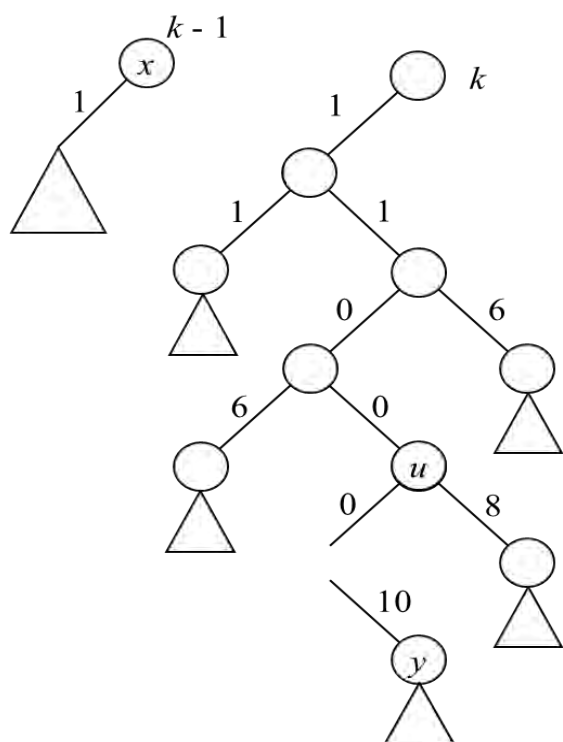


α)

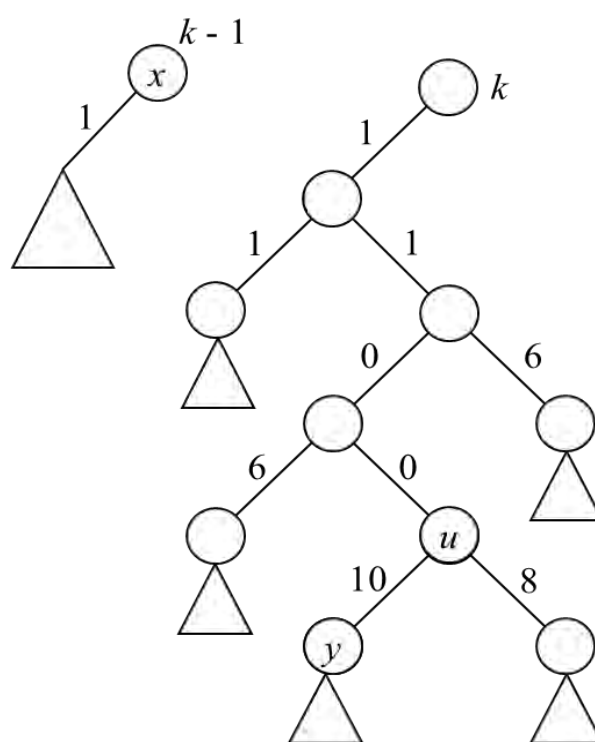


β)

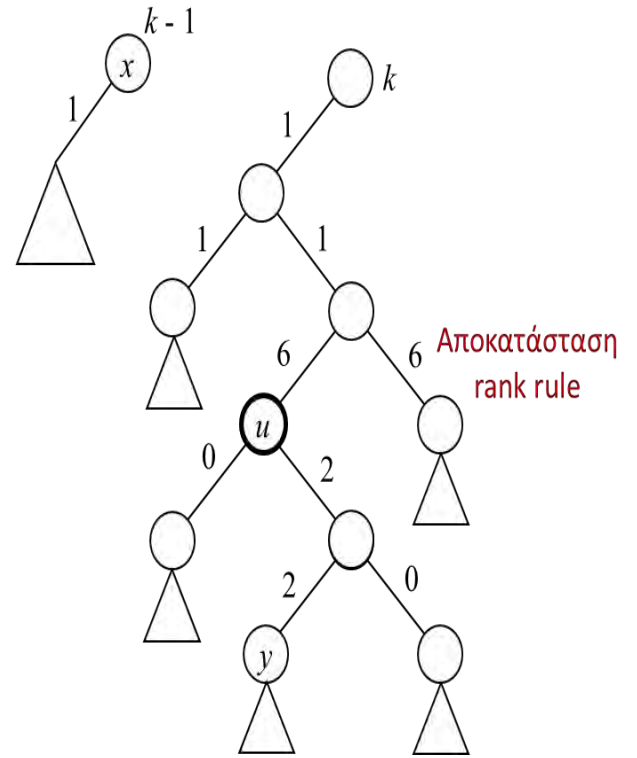
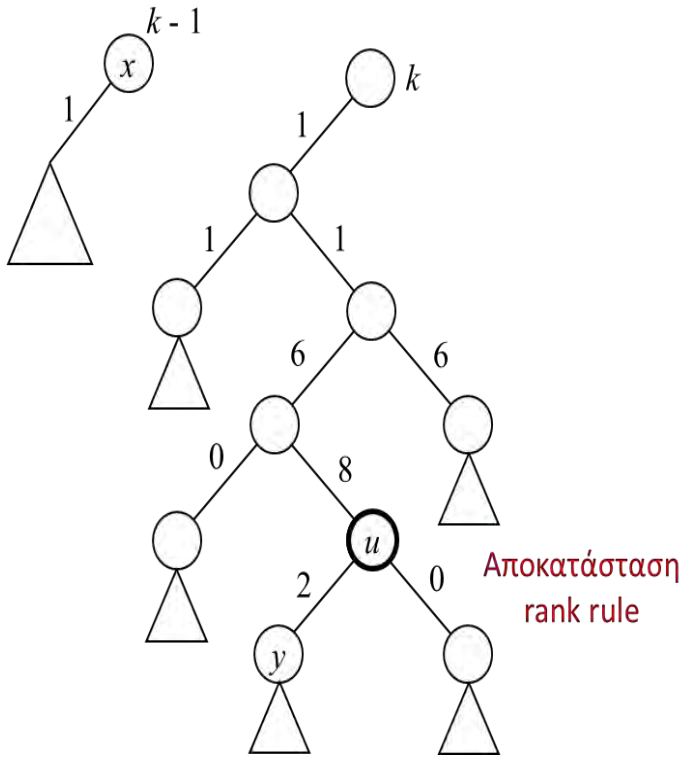
Αποσύνδεση half tree



γ)

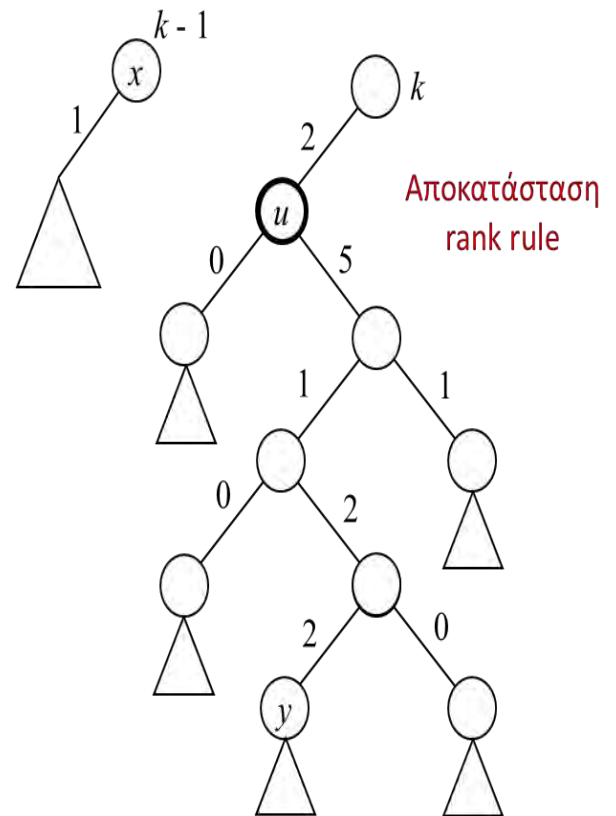
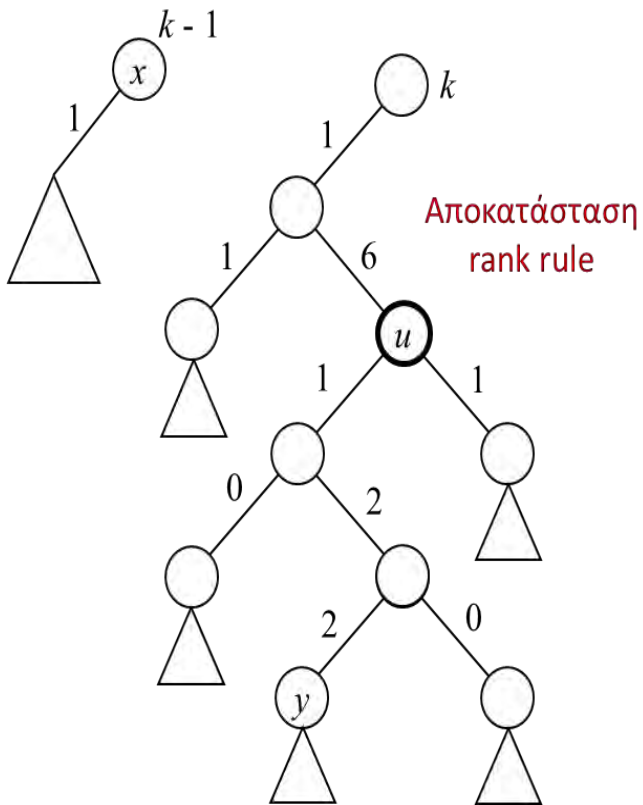


δ)



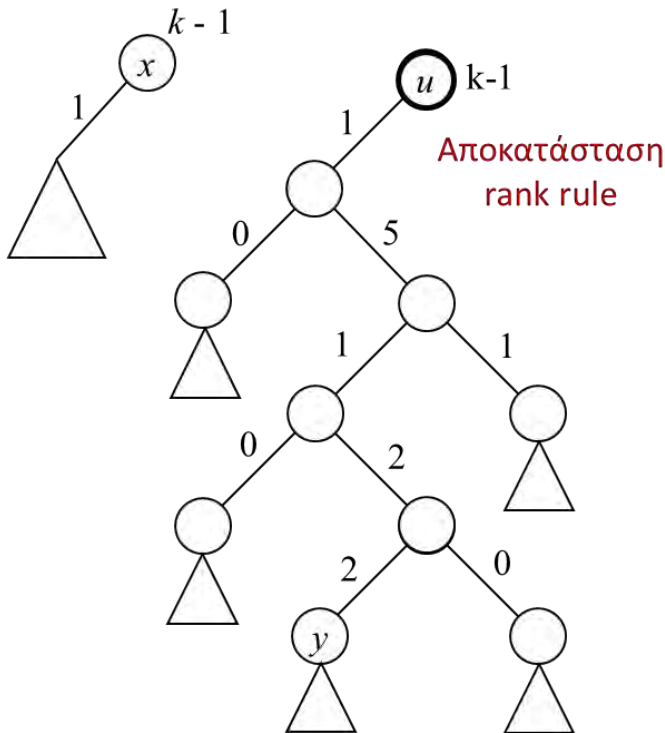
ε)

ζ)



η)

θ)



ι)

Σχήμα 5.2 : Μείωση κλειδιού στον type 1 rp heap. Οι αριθμοί στις ακτίνες είναι τα rank difference. Όταν ο x γίνεται ρίζα το rank του γίνεται όσο του αριστερού του παιδιού +1 . Κάθε κόμβος στο μονοπάτι από το u στο y μειώνεται το rank του.

Λήμμα 5.2. Η διαδικασία κατάταξης μείωσης του rank (rank-reduction process) αποκαθιστά τον rank rule.

Απόδειξη : Εφόσον όλα τα rank differences είναι μη αρνητικά , $r(y) \leq r(x)$ πριν το κλειδί μειωθεί . Εάν $r(y) < r(x)$, αντικαθίσταται το x με το y , μπορεί να προκαλέσει $r(y)$, αλλά μόνο $r(y)$, να παραβιάσει τον rank rule. Αν ο u παραβιάζει τον rank rule πριν από το βήμα μειώσεως κλειδιού, το βήμα μειώνει το rank του για να υπακούει στο κανόνα. Αυτό μπορεί να προκαλέσει $r(u)$, αλλά μόνο το $r(u)$, να παραβιάσει τον κανόνα. Επαγωγικά ο αριθμός των βημάτων δίνει το λήμμα.

Πριν παρουσιάσουμε την ανάλυση του type-1 rp-heaps, θα παρουσιάσουμε μια πιο χαλαρή έκδοση που θα υπακούει στον κανόνα type-2 rank rule. Κάθε ρίζα είναι 1-node και κάθε παιδί είναι 1,1-node, 1,2-node, ή 0,i-node για κάποιο $i > 1$ (ενδεχομένως διαφορετικό για κάθε κόμβο). Ο type-2 rp-heap είναι ένα σετ από heap-ordered half trees των οποίων οι κόμβοι έχουν ranks τα οποία υπακούουν στον type-2 rank rule.

Οι πράξεις σωρού του type-2 rp-heaps, είναι ίδιες με αυτές του type-1 rp-heaps, εκτός από την λειτουργία της μείωσης του rank (rank reduction process) , που αποκαθιστά τον type-2, χρησιμοποιώντας το ακόλουθο βήμα:

- Type-2 rank-reduction step: Αν ο u είναι ρίζα, τότε $r(u) = r(\text{left}(u))+1$ και σταματάμε. Διαφορετικά, έστω v και w είναι τα παιδιά του u . Έστω $k = \max\{r(v), r(w)\}$ εάν $|r(v) - r(w)| > 1$, $k = \max\{r(v), r(w)\} + 1$ εάν $|r(v) - r(w)| \leq 1$. Αν $k \geq r(u)$, σταματάμε. Διαφορετικά, $r(u) = k$ και $u = p(u)$.

Το λήμμα 5.2 ισχύει και για τον type-2 μείωση rank με την ίδια απόδειξη. Και στους δυο τύπους rank reduction , κάθε πετυχημένη μείωση rank είναι ποσοτικά ίδια ή μικρότερη. Στον multipass rp-heap, $k \leq r(u)$ σε κάθε βήμα μείωσης rank, αν και αυτό δεν είναι αλήθεια στην παραλλαγή που αναπτύσσουμε στο ενότητα 7.

Το όριο για το rank είναι μεγαλύτερο για τον type-2 rp-heaps κατά ένα σταθερό κλάσμα σε σχέση με τον type-1 rp heaps, αλλά είναι ίδιο με τον σωρό Fibonacci. Συμβολίζουμε με F_k τον k -ιστό αριθμό Fibonacci, που ορίζεται από την επανάληψη $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$ για $k > 1$. Συμβολίζουμε με ϕ τη χρυσή αναλογία (golden ratio), $(1 + \sqrt{5}) / 2$.

Λήμμα 5.3. Σε έναν type-2 rp-heap κάθε κόμβος με rank k , έχει τουλάχιστον $F_{k+2} \geq \phi^k$ απογόνους συμπεριλαμβανομένου και του ιδίου , τουλάχιστον $F_{k+3}-1$ αν είναι παιδί. Οπότε $k \leq \log_{\phi} n$.

Απόδειξη: Το δεύτερο μέρος του λήμματος αν αποδειχθεί , αποδεικνύει αυτόματα το πρώτο και το τρίτο μέρος, δίνοντας την γνωστή [25, p. 18] ανισότητα $F_{k+2} \geq \phi^k$. Αποδεικνύουμε το δεύτερο μέρος με επαγωγή στο ύψος ενός κόμβου. Ένας κόμβος που λείπει ικανοποιεί το δεύτερο μέρος, το ίδιο κάνει και ένα φύλλο. Έστω x είναι ένα παιδί με rank k του οποίου τα παιδιά πληρούν το δεύτερο μέρος. Εάν ο x είναι ένας 0,i-node, τότε το 0-child του x έχει τουλάχιστον $F_{k+3}-1$ απογόνους από την επαγωγική υπόθεση. Το ίδιο και ο x . Αν ο x είναι 1,1- ή 1,2-node, τότε το x έχει τουλάχιστον $F_{k+1}-1 + F_{k+2}-1 + 1 = F_{k+3}-1$ απογόνους από την επαγωγική υπόθεση , δεδομένου ότι $F_{k+1} \leq F_{k+2}$.

6. Η Amortized efficiency των rank-pairing heaps.

Στην ενότητα αυτή αναλύετε η αποτελεσματικότητα (efficiency) των rp -heaps. Θα παρουσιαστεί η ανάλυση του $type-2$ heaps, η οποία είναι πιο εύκολο από ό,τι η ανάλυση του $type-1$ heaps. Χρησιμοποιείτε ένα όρισμα μεθόδου δυναμικού. Η επιλογή μιας μεθόδου δυναμικού είναι περισσότερο τέχνη παρά επιστήμη. Η επιλογή προέκυψε από μια διαδικασία δοκιμής και λάθους, βελτίωσης και απλούστευσης: Ακόμα και για τον $type-2$ rp -heaps, η επιλογή είναι λίγο λεπτή. Εκ των υστέρων, μπορεί να υπάρχει κάποια διαίσθηση για την επιλογή μας. Από την ενότητα 5 ότι ένα παιδί που λείπει έχει $rank -1$ από σύμβαση. Το μη υπάρχων δεξιό παιδί της ρίζας δεν θεωρείται πως λείπει. Θα πρέπει να γίνει $amortize$ δύο είδη βήματος χρόνου $O(1)$: τις ενώσεις, καθεμία από τις οποίες μετατρέπει μια ρίζα σε ένα $1,1$ -node, και τα βήματα του $rank$ -reduction. Είναι φυσικό να δοκιμαστεί μια μέθοδος δυναμικού που είναι το άθροισμα των δυναμικών των κόμβων. Αν αναθέσουμε μία μονάδα δυναμικού σε κάθε ρίζα και μηδέν σε κάθε $1,1$ -node, τότε κάθε ένωση μειώνει το δυναμικό κατά ένα. Αυτό ισχύει για τις ενώσεις. Παρατηρούμε ότι, αν το $rank$ ενός κόμβου μη-ρίζας μειώνεται κατά k , η $rank$ difference του κάθε ένα από τα δυο του παιδιά μειώνεται κατά k , και το δικό του $rank$ difference αυξάνεται κατά k , έτσι το άθροισμα από αυτά τα τρία $rank$ difference μειώνεται κατά k . Έτσι, αν έχουμε εκχωρήσει ένα δυναμικό $i + j$, για κάθε i, j -child, τότε κάθε βήμα $rank$ -reduction μειώνει το δυναμικό τουλάχιστον κατά ένα. Αυτό εξακολουθεί να ισχύει, αν δώσουμε ένα δυναμικό $i + j + c$ για κάθε i, j -child, όπου c είναι οποιοσδήποτε σταθερά. Επιλέγοντας $c = -2$ δίνει σε κάθε $1,1$ -node μηδενικό δυναμικό, σταθερά με την ανάθεση που χρειάζεται για να κάνουμε ενώσεις. Αυτό ισχύει για τα βήματα του $rank$ -reduction.

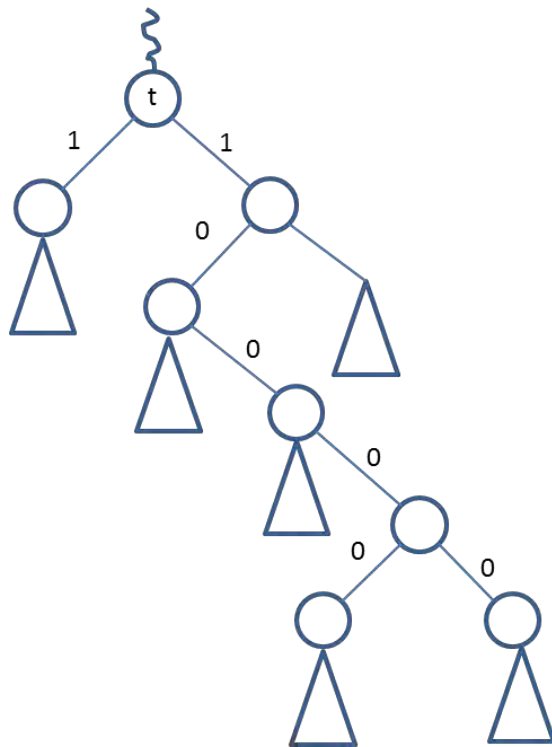
Δυστυχώς υπάρχει ένας ακόμα περιορισμός: Η αποσυναρμολόγηση του $half$ -tree που ξεκίνησε από την διαγραφή ελαχίστου μπορεί να μετατρέψει έναν αυθαίρετα μεγάλο αριθμό από $0,2$ -nodes, σε ρίζες ο καθένας από τους οποίους χρειάζεται μια μονάδα δυναμικού. Για κάθε τέτοιο κόμβο να έχει το αναγκαίο δυναμικό πριν την αποσυναρμολόγηση, θα πρέπει να επιλέξουμε $c \geq -1$, το οποίο παράγει φαινομενική μη επιλύσιμη κυκλικότητα. Ευτυχώς υπάρχει τρόπος να σπάσει αυτή η κυκλικότητα. Όπως αποδεικνύετε παρακάτω, το πολύ ένας $1,1$ -node μπορεί να μειωθεί το $rank$ του κατά την διάρκεια κάθε μείωσης κλειδιού. Αυτό μας δίνει την δυνατότητα να μειώσουμε το δυναμικό κάθε $1,1$ -node κατά ένα, δεδομένου ότι η πρόσθετη μονάδα απαιτείται όταν αλλάζει η κατάσταση και μπορεί να χρεωθεί στον αντίστοιχο κλειδί που μειώνεται. Έτσι επιλέγουμε $c = -1$, αλλά δίνουμε σε κάθε $1,1$ -node δυναμικό μηδέν.

Απομένει να αποδειχτεί ότι αυτό δουλεύει. Αναθέτουμε ένα δυναμικό σε κάθε σωρό ίσο με το άθροισμα των δυναμικών κάθε κόμβου. Το δυναμικό ενός κόμβου είναι το άθροισμα των $rank$ differences των παιδιών του, μείον ένα αν είναι παιδί, αλλά όχι ένας $1,1$ -node, ή μείον δυο αν είναι $1,1$ -node. Αυτός ο ορισμός εφαρμόζεται όχι μόνο σε κόμβους που υπακούουν το $rank$ rule, αλλά επίσης και στους κόμβους που παραβιάζουν το $rank$ rule στη διάρκεια της μείωσης κλειδιού. Ήτοι, το δυναμικό μιας ρίζας με ένα i -child είναι i , το δυναμικό ενός $1,1$ -node είναι 0 , και το δυναμικό ενός i,j -node εκτός από τον $1,1$ -node είναι $i + j - 1$.

Θεώρημα 6.1. Ο amortized time για πράξεις του αλγόριθμου type-2 rp-heap είναι $O(1)$ για τις πράξεις δημιουργία σωρού, συγχώνευση σωρών, εύρεση ελαχίστου, εισαγωγή στοιχείου, μείωση κλειδιού και $O(\log n)$ για διαγραφή ελαχίστου

Απόδειξη: Οι πράξεις δημιουργία σωρού, συγχώνευση σωρών, εύρεση ελαχίστου χρειάζονται $O(1)$ πραγματικό χρόνο και δεν αλλάζουν το δυναμικό: μια εισαγωγή χρειάζεται $O(1)$ χρόνο και αυξάνει το δυναμικό κατά ένα. Οπότε κάθε μια από αυτές τις πράξεις παίρνει χρόνο $O(1)$. Θεωρήστε μια διαγραφή ελαχίστου. Κάθε νέα ρίζα που δημιουργείται από την αποσυναρμολόγηση έχει το δυναμικό που χρειάζεται (μια μονάδα), εκτός αν στο παρελθόν ήταν ένας 1,1-node. Το πολύ ένας 1,1-node μπορεί να γίνει νέα ρίζα για κάθε rank μικρότερο από αυτό που είχε η ρίζα που διαγράφηκε. Σύμφωνα με το λήμμα 5.3 υπάρχουν το πολύ $\log_\phi n$ τέτοιοι κόμβοι 1,1-node. Έτσι, η αποσυναρμολόγηση αυξάνει το δυναμικό το πολύ κατά $\log_\phi n$. Έστω h ο αριθμός των half trees μετά την αποσυναρμολόγηση. Ολόκληρη η διαγραφή ελαχίστου χρειάζεται $h-1$ συγκρίσεις κλειδιών και χρόνο $O(h+1)$. Κλιμακώστε αυτόν τον χρόνο να γίνει το πολύ $h+O(1)$. Κάθε ένωση μετά την αποσυναρμολόγηση μετατρέπει μια ρίζα σε 1,1-node, το οποίο μειώνει το δυναμικό κατά ένα. Το πολύ $\log_\phi n + 1$ half trees παραμένουν μετά από όλες τις ενώσεις, οπότε υπάρχουν το λιγότερο $h - \log_\phi n - 1$ ενώσεις. Ο amortized time της διαγραφής ελαχίστου οπότε είναι το πολύ $\log_\phi n + h + O(1) - h + \log_\phi n + 1 = 2\log_\phi n + O(1)$ και ο amortized αριθμός συγκρίσεως κλειδιών είναι το πολύ $2\log_\phi n$.

Το καινοτόμο στοιχείο της ανάλυσης είναι η μείωση του κλειδιού. Εξετάζετε το ενδεχόμενο μείωσης του κλειδιού του κόμβου x . Αν το x είναι ρίζα, η μείωση κλειδιού παίρνει πραγματικό χρόνο $O(1)$ και δεν αλλάζει το δυναμικό. Ας υποθέσουμε ότι το x δεν είναι ρίζα. Έστω y το δεξί παιδί του x και u ο πατέρας του x . Θεωρήστε το half tree που περιέχει τον κόμβο αποσπάται το x μαζί με το υποδένδρο του. Έστω ότι z είναι ο τελευταίος (το ανώτερο) όχι-ρίζα κόμβος του οποίου το rank μειώνεται ως αποτέλεσμα της μείωσης του κλειδιού. Μπορεί να υπάρχουν το πολύ ένας 1,1-node στο μονοπάτι από το x στο z συμπεριλαμβανομένου του x και του z . Πράγματι, έστω t είναι ο χαμηλότερος 1,1-node σε αυτό το μονοπάτι. Αν $t = x$, τότε ο αντικαταστάτης του x ως παιδί του u από τον y δίνει στον u ένα νέο παιδί με rank difference κατά ένα μεγαλύτερο σε σχέση με το παλαιό του παιδί. Αν από την άλλη πλευρά $t \neq x$, ο t μπορεί να μειωθεί στο rank του το πολύ κατά ένα. Σε κάθε περίπτωση, κάθε επακόλουθη μείωση rank (μετά την αντικατάσταση του x από τον y ή μετά την μείωση του rank του t) είναι ακριβώς ένα. Εάν ένα παιδί ενός κόμβου 1,1-node μειώνεται το rank του κατά ένα, τότε ο κόμβος γίνεται 1,2-node, και η διαδικασία του rank reduction σταματά. Έτσι, η διαδρομή από το x στο z δεν μπορεί να περιέχει ένα δεύτερο 1,1-node. (Σχήμα 6.1.)



Σχήμα 6.1: Τα rank reduction κατά τη διάρκεια μείωσης κλειδιού για τον type-2 rp-heaps. Ο κόμβος y μπορεί να έχει αυθαίρετα μεγάλη rank difference. Οι κόμβοι στο μονοπάτι από το u στο t μπορούν να μειωθούν σε rank με αυθαίρετα μεγάλο ποσό, αλλά ο t και κάθε κόμβος πάνω από τον t μπορούν να μειώσουν το rank τους μόνο κατά 1. Εξ ου και ο επόμενος 1,1-node πάνω από τον t δεν μπορεί να μειωθεί το rank του, αλλά μόνο να γίνει ένας 1,2-node.

Προσθέστε μία μονάδα δυναμικού προς τον 1,1-node, εάν υπάρχει, στο μονοπάτι από x έως Z , και προσθέστε δύο μονάδες στο δυναμικό του x . Τώρα, κάθε (όχι ρίζα) κόμβος στο μονοπάτι από το x έως το z έχει δυναμικό ίσο με το άθροισμα του rank difference των παιδιών του μείον ένα, και ο x έχει δύο επιπλέον μονάδες δυναμικού. Αφαιρέστε το half tree με ρίζα το x , αντικαταστήστε το με το υποδένδρο με ρίζα το y , και δώστε τα όλα στο u εκτός από μια μονάδα του δυναμικού του x . Τώρα το x είναι ρίζα, έχει τη μία μονάδα δυναμικού που χρειάζεται (το rank του είναι τώρα μεγαλύτερο κατά ένα από το rank του αριστερού παιδιού του), και κάθε κόμβος στο μονοπάτι από το u στο z έχει δυναμικό ίσο με το άθροισμα των rank difference των παιδιών τους, μείον ένα. Τέλος, κάντε τις διαδοχικές μειώσεις του rank. Η μείωση του rank ενός κόμβου, non-root, κατά k μειώνει το δυναμικό του κατά τουλάχιστον $2k$ ($2k + 1$, εάν γίνει 1,1-node) και αυξάνει το δυναμικό του πατέρα κατά k , εκτός εάν ο γονέας είναι ένα 1,1-node που γίνεται 1,2-node. Στην περίπτωση αυτή το δυναμικό του κόμβου μειώνεται κατά δυο, το δυναμικό του πατέρα αυξάνεται κατά δύο, και αυτή είναι η τελευταία μείωση rank. Μείωση του rank της ρίζας κατά k μειώνει το δυναμικό του κατά k χωρίς να επηρεάζει το δυναμικό οποιουδήποτε άλλου κόμβου. Έτσι, αν χρεωθεί μία μονάδα χρόνου ανά rank reduction, ο amortized time για ένα rank reduction, εκτός του τελευταίου είναι το πολύ μηδέν, και του τελευταίου είναι το πολύ ένα. Ο amortized time για όλα τα rank reduction είναι το πολύ τέσσερα (τρεις μονάδες που

προσθέσαμε δυναμικό συν το πολύ ένα για την τελευταία μείωση του rank). Καταλήγουμε στο συμπέρασμα ότι η μείωση κλειδιού παίρνει amortized χρόνο $O(1)$.

Τώρα θα αναλύσουμε τον type-1 rp-heaps. Σε αντίθεση με την ανάλυση μας για τον type-2 rp-heaps, η ανάλυση του type-1 rp-heaps απαιτεί περιορισμό στις ενώσεις που γίνονται κατά τη διάρκεια της διαγραφής ελαχίστου. Κατά τη διάρκεια μιας τέτοιας διαγραφής ελαχίστου, τμηματοποιούμε τα half trees σε δύο είδη, νέα και παλιά: εκείνα που σχηματίζονται από την αποσυναρμολόγηση του half tree με ρίζα στην min-root είναι τα νέα, τα υπόλοιπα στο σωρό είναι παλιά. Κάνουμε συνεχόμενες ενώσεις των νέων half trees, κάνοντας τα νέα half trees που προκύπτουν από μια τέτοια ένωση παλιά. Όταν δεν υπάρχουν πλέον δύο νέα half trees με το ίδιο rank, κάνουμε αυθαίρετες ενώσεις έως ότου δεν υπάρχουν δύο half trees με το ίδιο rank. Αυτό το ονομάζουμε restricted multipass linking. Ένας απλός τρόπος για την εφαρμογή της εν λόγω ενώσεις είναι να κάνουμε one-pass linking τα νέα half trees και στη συνέχεια να κάνουν (standard) multipass linking όλων των υπόλοιπων half trees. Δεν ξέρουμε αν η ανάλυσή μας μπορεί να επεκταθεί σε αυθαίρετα multipass linking.

Χρειαζόμαστε μια πιο πολύπλοκη συνάρτηση δυναμικού από αυτή που είχαμε για τον type-2 rp-heaps. Μια απλή μείωση κλειδιού μπορεί να μετατρέψει έναν αυθαίρετα μεγάλο αριθμό 1,1-nodes σε 0,1-nodes. Αν αναθέσουμε ένα δυναμικό σε κάθε κόμβο, βασιζόμενη μόνο στα rank differences των παιδιών τους, χρειαζόμαστε το δυναμικό των ριζών να υπερβαίνει εκείνο του 1,1-node, το δυναμικό των 1,1-rank να υπερβαίνει εκείνο του 0,1-rank, και το δυναμικό των 0,1-node να είναι όχι μικρότερο από εκείνη των ριζών, πράγμα αδύνατο. Παρόλα αυτά, η μείωση κλειδιού έχει περιορισμένη επίδραση στους 1,1-nodes και στους 0,1-nodes, μια επίδραση που εξαρτάται από τα rank difference των εγγονών τους. Θεωρήστε έναν 1,1-node x με δύο 1,1-child. Μια μείωση στο rank του x μετατρέπει τον x σε 0,1-node του οποίου το 0-child είναι ένας 1,1-node. Ένας 0,1-node του οποίου το 0-child είναι ένα 1,1-node δεν μπορεί να μειωθεί σε rank ως αποτέλεσμα της μείωσης του rank difference ένα από τα παιδιά της: ένας τέτοιος κόμβος σταματά τον καταρράκτη των μειώσεων του rank difference που προκλήθηκαν από μια μείωση κλειδιού. Αυτό υποδηλώνει μια συνάρτηση δυναμικού που μεταχειρίζεται τέτοιους κόμβους ως ειδικές περιπτώσεις. Για να γίνει αυτήν την ιδέα να δουλέψει, θα πρέπει να διακρίνουμε τις ενώσεις σε αυτές που δημιουργούν 1,1-nodes με δύο 1,1-children και σε αυτές που δεν δημιουργούν. Εμείς χωρίζουμε τους κόμβους σε δύο είδη, καλούς (good) και κακούς (bad). Ένας κόμβος είναι καλός και αν είναι ρίζα με rank μηδέν (με ένα χαμένο αριστερό παιδί) ή αν το αριστερό παιδί του είναι ένας 1,1-node, ή είναι ένας 1,1-node με rank μηδέν (χωρίς παιδιά) ή τα παιδιά του είναι 1,1-nodes, ή είναι ένας 0,1-node του οποίου το 0-child είναι ένας 1,1-node. Όλοι οι άλλοι κόμβοι είναι κακοί. Με τον ορισμό αυτό, ο νικητής (winner) μιας ένωσης είναι μια καλή ρίζα, ο ηττημένος (loser) μιας ένωσης είναι ένας 1,1-node που είναι καλός, αν οι δύο ρίζες που ενώθηκαν ήταν και δυο καλές ή κακός αν τουλάχιστον μία από τις ρίζες που ενώθηκαν ήταν κακή.

Όπως δείχνουμε παρακάτω, μια μείωση κλειδιού μπορεί να κάνει το πολύ έναν καλό node κακό. Αυτό μας επιτρέπει να δώσουμε στους καλούς κόμβους αφύσικα χαμηλό δυναμικό, αποφεύγοντας έτσι το πρόβλημα της κυκλικότητας που αναφέρθηκε παραπάνω. Υπάρχει μια τελική δυσκολία, ωστόσο. Η αποσυναρμολόγηση του half tree που προκλήθηκε από

διαγραφή ελαχίστου μπορεί να παράγει νέες κακές ρίζες των οποίων τα δυναμικά είναι πολύ χαμηλά. Τέτοιες ρίζες έχουν αρκετό δυναμικό για να δώσουν(pay) για να γίνουν ενώσεις μεταξύ τους , αλλά όχι για ενώσεις μεταξύ ενός εξ αυτών και μιας καλής ρίζας. Αυτός είναι ο λόγος για τον περιορισμό των συνενώσεων.

Μας μένει να επεξεργαστούμε τις λεπτομέρειες αυτής της ιδέας. Ορίζουμε το δυναμικό της σωρού να είναι το άθροισμα των δυναμικών των κόμβων. Ορίζουμε το δυναμικό ενός κόμβου να είναι το άθροισμα των rank differences των παιδιών του, συν δύο αν είναι ρίζα, πλην ένα, εάν είναι καλή, ή συν τρία, αν είναι κακή. Ο ορισμός αυτός δεν ισχύει μόνο για τους κόμβους που υπακούουν στον rank rule , αλλά και στους κόμβους που παραβιάζουν τον rank rule στη διάρκεια μείωσης κλειδιού . Δηλαδή, το δυναμικό της ρίζας με ένα i-child είναι $i + 1$, εάν η ρίζα είναι καλή, $i+5$ εάν η ρίζα είναι κακή. Το δυναμικό ενός 1,1-node είναι 1 αν είναι καλό, 5 αν είναι κακό. Το δυναμικό ενός 0,1-node είναι 0 αν είναι καλό, 4, εάν ο κόμβος είναι κακός. Το δυναμικό του i, j-node ο οποίος δεν είναι 1,1-node ή 0,1-node είναι $i + j + 3$: όλοι αυτοί οι κόμβοι είναι κακοί.

Θεώρημα 6.2. Ο amortized time για τις λειτουργίες στον αλγόριθμο type-1 rp-heap με restricted multipass linking είναι $O(1)$ για δημιουργία σωρού, εύρεση ελαχίστου, ένθεση στοιχείου, συγχώνευση ουρών, ή μείωση κλειδιού και $O(\log n)$ για την λειτουργία διαγραφή ελαχίστου.

Οι πράξεις δημιουργία σωρού , συγχώνευση σωρών, εύρεση ελαχίστου χρειάζονται $O(1)$ πραγματικό χρόνο και δεν αλλάζουν το δυναμικό: μια εισαγωγή χρειάζεται $O(1)$ χρόνο και αυξάνει το δυναμικό κατά δύο , επειδή η νέα ρίζα είναι καλή. Οπότε κάθε μια από αυτές τις πράξεις χρειάζεται amortized time $O(1)$.

Θεωρείστε μια διαγραφή ελαχίστου. Κατά τη διάρκεια της αποσυναρμολόγησης και τις συνενώσεις από ζεύγη, νέων half trees (εκείνα τα οποία σχηματίζονται από την αποσυναρμολόγηση), δίνετε σε κάθε ρίζα ενός νέου half tree ένα προσωρινό δυναμικό , τέσσερα αν είναι καλό ή κακό, αντί του σωστού δυναμικό ,2 αν είναι καλό, και 6 αν είναι κακό. Ισχυριζόμαστε ότι αν το κάνουμε αυτό, η αύξηση του δυναμικού που προκαλείται από την αποσυναρμολόγηση είναι το πολύ $4\lg n + 4$. Οι μόνοι κόμβοι των οποίων τα δυναμικά μπορούν να αυξηθούν είναι αυτά των νέων ριζών. Κάθε κακός κόμβος έχει τουλάχιστον τέσσερις μονάδες δυναμικό, έτσι ώστε αν γίνει ρίζα να έχει τις τέσσερις μονάδες του προσωρινού δυναμικού που χρειάζεται. Εξετάστε ένα καλό κόμβο x που γίνεται ρίζα. Υπάρχουν δύο περιπτώσεις. Αν ο x είναι ένας 1,1-node, ή ο x είναι ένας 0,1-node του οποίου το δεξιό του παιδί είναι ένα 1-child, τότε χρεώνουμε τις τέσσερις ή λιγότερες μονάδες προσωρινού δυναμικού που χρειάζονται από το x ως μια νέα ρίζα για το $r(x)$. Αν ο x είναι ένας 0,1-node των οποίων το δεξιό του παιδί είναι ένα 0-child, τότε χρεώνουμε τις τέσσερις μονάδες προσωρινού δυναμικού που χρειάζονται από το x ως μια νέα ρίζα για το $r(y)$, όπου y είναι ο υψηλότερος κακός κόμβος στη δεξιά ραχοκοκαλιά του x . Δείτε το Σχήμα 6.2. Εάν δε υπάρχει τέτοιο y , τότε χρεώνουμε τις τέσσερις μονάδες που χρειάζονται από το x στην διαγραφή ελαχίστου. Αν ο y υπάρχει, τότε κάθε κόμβος στο μονοπάτι από το x στο y είναι ένας καλός 1,1-node εκτός από το x και το y . Δεδομένου ότι το y είναι κακό, το $r(y)$ μπορεί να χρεωθεί μόνο μία φορά για έναν τέτοιο 0,1-node, και δεν μπορεί να χρεώνεται για έναν καλό 1,1-node ή 0,1-node του οποίου το δεξιό παιδί είναι ένα

Τώρα θεωρείστε τις ενώσεις ζευγών από νέα half trees. Κάθε τέτοια ένωση μετατρέπει μια ρίζα σε 1,1-node και κάνει την άλλη ρίζα καλή ρίζα. Πριν από την ένωση, οι εν λόγω κόμβοι έχουν δυναμικό οκτώ (τέσσερα συν τέσσερα). Δίνουμε στον root που απομένει μετά την ένωση τους , το σωστό δυναμικό των δύο. Μετά την ένωση η εναπομείναντα ρίζα και το νέο 1,1-node έχουν δυναμικό το πολύ επτά (δύο συν πέντε), έτσι ώστε η ένωση να μειώσει το δυναμικό κατά τουλάχιστον ένα.

32

τουλάχιστον μία από τις ρίζες που συνενώνονται είναι κακή, τότε οι δύο ρίζες έχουν τουλάχιστον $6 + 2 = 8$ μονάδες δυναμικό πριν από την ένωση και το πολύ 2 (ο νικητής) $+ 5$ (ο χαμένος) $= 7$, μετά την ένωση. Αν και οι δύο ρίζες είναι καλές, έχουν $2 + 2 = 4$ μονάδες πριν από τη σύνδεση και $2 + 1 = 3$ μετά, εφόσον ο χαμένος είναι ένας καλός $1,1$ -node.

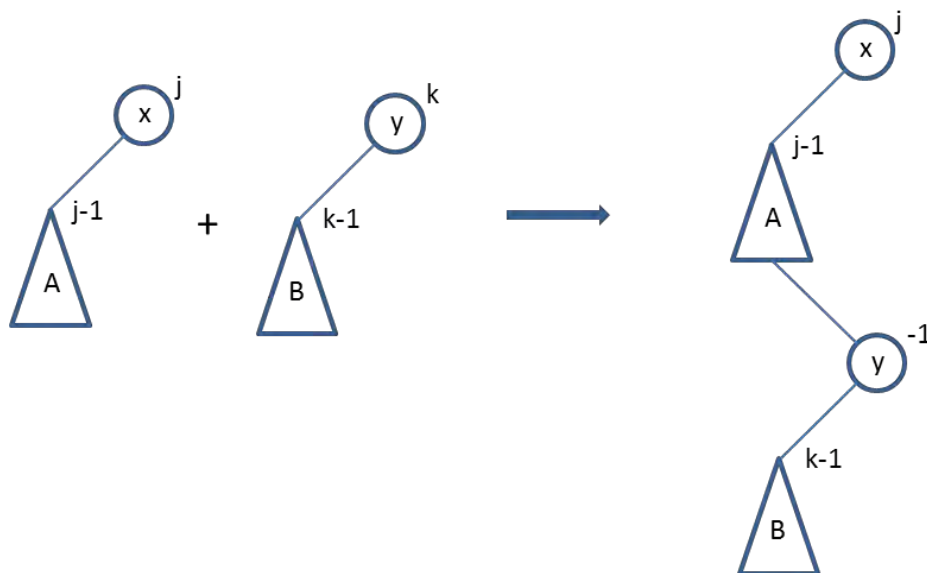
Καταλήγουμε στο συμπέρασμα, ότι η καθαρή αύξηση του δυναμικού που προκαλείται από την συνολική διαγραφή ελαχίστου είναι το πολύ $6\lg n + 4$, μείον ένα για κάθε ένωση. Έστω h είναι ο αριθμός των half trees μετά την αποσυναρμολόγηση. Ολόκληρη η διαγραφή ελαχίστου παίρνει $h-1$ συγκρίσεις κλειδιών και $O(h + 1)$ χρόνο. Κλιμακώστε αυτόν τον χρόνο να γίνει το πολύ $h + O(1)$. Το πολύ $\lg n + 1$ half trees θα παραμείνουν μετά από όλες τις συνενώσεις που θα γίνουν, έτσι ώστε να υπάρχουν τουλάχιστον $h - \lg n - 1$ ενώσεις. Η διαγραφή ελαχίστου αυξάνει έτσι το δυναμικό το πολύ σε $7\lg n - h + 1$, δίνοντας έναν amortized time $O(\lg n)$ και μέγιστο αριθμό συγκρίσεων κλειδιών $7\lg n + 5$.

Η ανάλυση της μείωσης κλειδιού σε έναν κόμβο x είναι ακριβώς έτσι και για τον type-2 heaps, εκτός από το ότι πρέπει ναδειχτεί ότι η μείωση κλειδιού μπορεί να κάνει μόνο $O(1)$ κόμβους κακούς. Ένας καλός $1,1$ -node δεν μπορεί να γίνει κακός, μπορεί να γίνει μόνο ένας καλός $0,1$ -node. Σε έναν καλός $0,1$ -rank δεν μπορεί να γίνει μείωση στο rank του, οπότε αν γίνεται κακός θα είναι ο τελευταίος κόμβος στον οποίο θα εφαρμοστεί ένα rank reduction. Αν ο x γίνει ρίζα, τότε μπορεί να γίνει μόνο κακός αν προηγουμένως ήταν ένας καλή $0,1$ -node με δεξιό παιδί 0 -child, στην οποία περίπτωση κανένα rank δεν αλλάζει και ο x είναι ο μόνος κόμβος που αλλάζει και γίνεται κακός. Για την ρίζα του παλιού half tree που περιέχει το x ο οποίος γίνεται κακός, το αριστερό παιδί του πρέπει να είναι $1,1$ -node, και η παλιά ρίζα είναι ο μόνος κόμβος που γίνεται κακός. Το συμπέρασμα είναι ότι η μείωση κλειδιού μπορεί να κάνει μόνο ένα κόμβο κακό. Αν δοθεί στον κόμβο που γίνεται κακός, τέσσερις μονάδες δυναμικού πριν από την διαδικασία rank reduction, τότε κάθε μείωση του rank του κόμβου μειώνει το δυναμικό τουλάχιστον κατά ένα, πληρώνοντας για τη μείωση. Η μείωση κλειδιού μπορεί επίσης να δημιουργήσει μια νέα ρίζα, για μια αύξηση δυναμικού δύο. Έτσι, η μείωση κλειδιού παίρνει amortized χρόνο $O(1)$.

Ο χρόνος χειρότερης περίπτωσης για μείωση κλειδιού στον αλγόριθμο rank pairing heap και των δυο τύπων $\Theta(n)$, όπως και για την σωρό Fibonacci. Μπορεί να το μειωθεί αυτό σε $O(1)$, καθυστερώντας κάθε πράξη μείωσης κλειδιού την επόμενη διαγραφή ελαχίστου. Αυτό απαιτεί τη διατήρηση του συνόλου των κόμβων που θα μπορούσαν να έχουν το ελάχιστο κλειδί, δηλαδή, όλες τις ρίζες και όλους τους κόμβους των οποίων τα κλειδιά έχουν μειωθεί από την τελευταία διαγραφή ελαχίστου.

7. One-tree rank-pairing heaps.

Εξετάζεται αν υπάρχει η one-tree version των rank pairing heaps. Μια ιδανική λύση θα ήταν να χειριζόμαστε τις unfair συνενώσεις με τον ίδιο τρόπο όπως οι fair συνενώσεις, όπως στην ενότητα 4: Ο loser μιας fair ένωσης γίνεται το αριστερό παιδί του winner. Δεν έγινε δυνατό να αποκτηθεί μια αποδεδειγμένα αποτελεσματική έκδοση που το κάνει αυτό. Αντ' αυτού, παρουσιάζεται εδώ μια one-tree version από rp-heaps που χειρίζεται τις unfair και τις συνενώσεις διαφορετικά. Αν ο x είναι ένας κόμβος, τότε η δεξιά ραχοκοκαλιά του $\text{left}(x)$ περιέχει όλους τους losers των ενώσεων, με το νεότερο loser να είναι ο πιο ρηχός. Δηλαδή, υπάρχει μια στοίβα από τους losers του x . Κάντε κάθε τέτοια ραχοκοκαλιά μια deque (double-ended ουρά) αντί για μια στοίβα. Αν ο y χάσει μια δίκαιη συνένωση με το x , προωθήστε το y στην στοίβα, καθιστώντας το, νέο αριστερό παιδί του x . Αν ο y χάσει μια unfair συνένωση με το x , βάλτε τον y στο κάτω μέρος της στοίβας, καθιστώντας τον, τον νέο βαθύτερο κόμβο στην δεξιά ραχοκοκαλιά του $\text{left}(x)$. (Βλ. Σχήμα 7.1.) . Επιπλέον, όταν y χάνει μια δίκαιη συνένωση από το x , βάλτε το rank του -1 . Μην αλλάζεται το rank του x . Το -1 είναι απλώς μια σημαία που εμποδίζει την μείωση του rank όταν γίνονται συνεχόμενες fair και unfair ενώσεις. Όταν ο χαμένος μιας άδικης ένωσης γίνει και πάλι ρίζα, είτε επειδή το κλειδί του έχει μειωθεί ή ως αποτέλεσμα της αποσυναρμολόγησης, θέστε το rank του ίσο με $1 + \text{το rank του αριστερού του παιδιού}$.



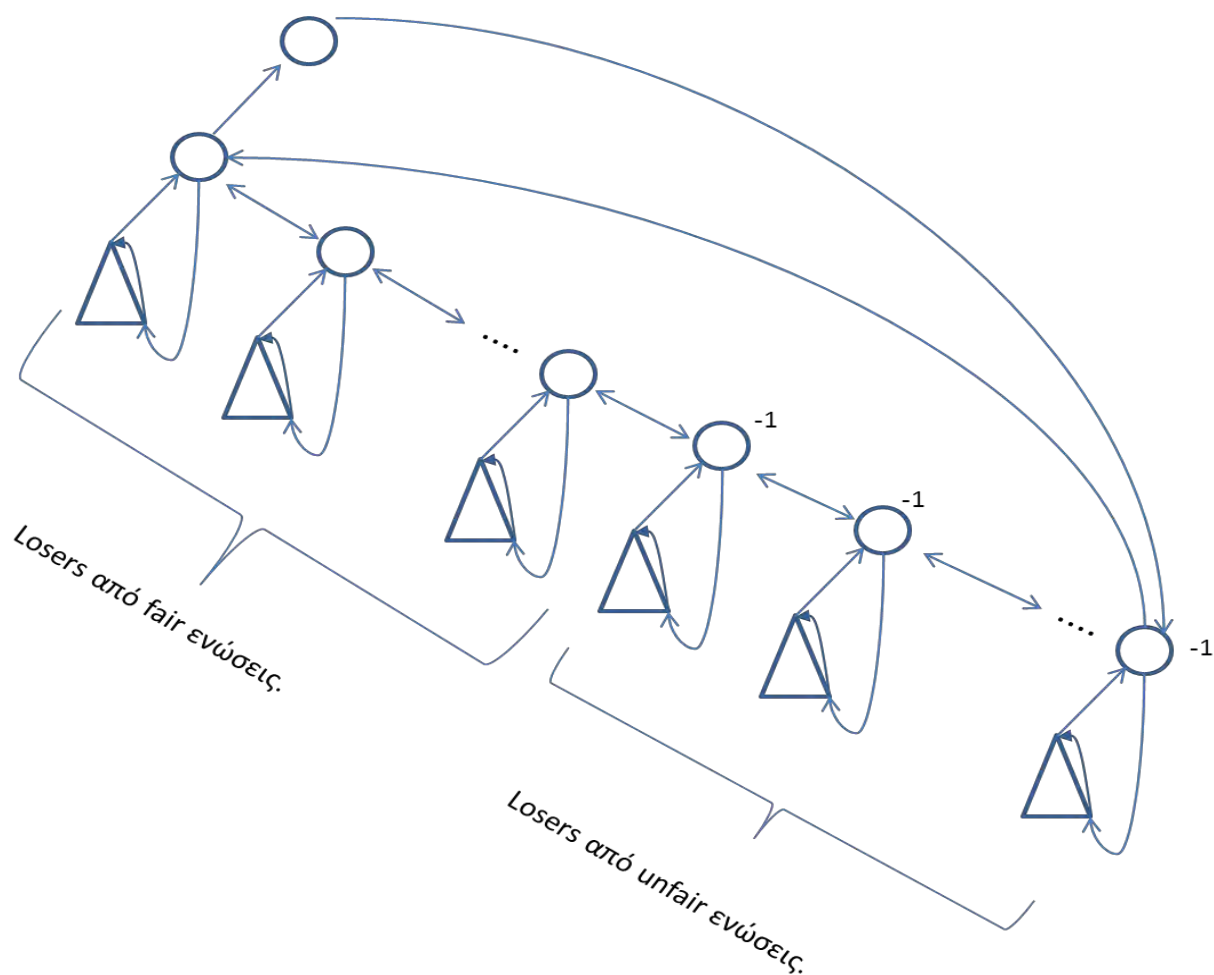
Σχήμα 7.1: Μια unfair ένωση

Παρουσιάζετε ένα σωρό από ένα μόνο half tree. Για να εισάγετε ένα αντικείμενο σε έναν σωρό, δημιουργήστε ένα one-node half tree και συνενώστε αυτό το half tree με το υπάρχον half tree μέσα από μια fair ή unfair ένωση ανάλογα με την περίπτωση. Για να συγχωνευτούν δύο σωροί, συνενώστε τα half trees τους, μέσα από μια fair ή unfair ένωση. Για να κάνετε διαγραφή ελαχίστου, αποσυναρμολογείτε το half tree, συνενώστε τα half trees που δημιουργήθηκαν από την αποσυναρμολόγηση με fair συνενώσεις έως ότου δεν υπάρχουν δύο εναπομείναντα δέντρα ίδιου rank, και στη συνέχεια συνενώστε τα υπόλοιπα half trees

με unfair συνενώσεις. Αν ο σωρός είναι τύπου 1, κάντε τις fair συνενώσεις με την restricted multipass method (ενότητα 6). Κάνετε μια μείωση κλειδιού, όπως στην ενότητα 5, μόλις σταματήσει η διαδικασία του rank reduction, αν υπάρχουν δύο δένδρα συνενώνοντε, με μια fair η unfair ένωση.

Για την αποτελεσματική εφαρμογή αυτής της μεθόδου, θα πρέπει να αλλάξει η δομή του δείκτη ώστε οι unfair ενώσεις να γίνονται σε $O(1)$ χρόνο. Ένας τρόπος να γίνει αυτό είναι να γίνει κάθε κόμβος με αριστερό παιδί να δείχνει όχι στο αριστερό του παιδί, αλλά στον κατώτατο (bottommost) κόμβο επί της δεξιάς ραχοκοκαλιάς από το αριστερό παιδί του, και να γίνει αυτό το κατώτατο κόμβο (το οποίο δεν έχει κανένα δεξιό παιδί) να δείχνει στο αριστερό παιδί. (Βλ. Σχήμα 7.2). Και άλλες παρουσιάσεις του one-tree half tree είναι δυνατές. Ποιά είναι η καλύτερη είναι ένα ερώτημα.

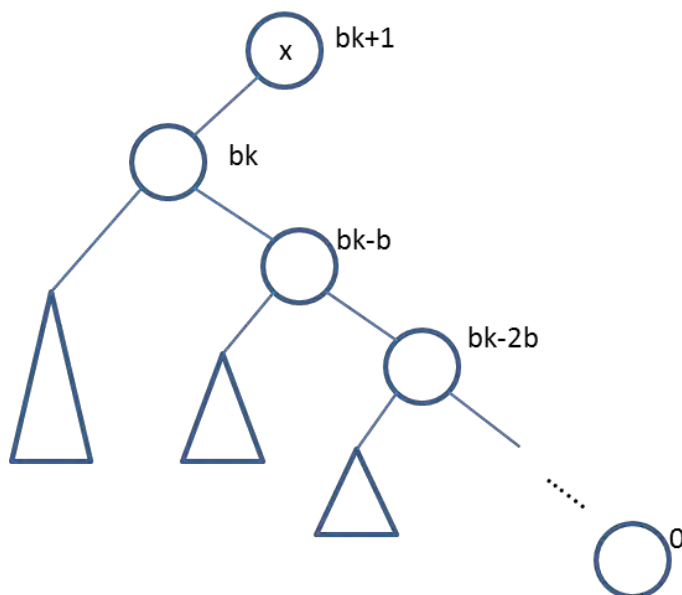
Η ανάλυση του αλγορίθμου one-tree rp-heaps είναι όπως της one-pass rp-heaps εκτός από το ότι πρέπει να λογαριάζονται και οι unfair ενώσεις. Υπάρχει μια unfair ένωση εισαγωγή στοιχείου, συγχώνευσης, και μειώσεις κλειδιού, και $O(\log n)$ ανά διαγραφή ελαχίστου. Δίνονται στους χαμένους των unfair ενώσεων το ίδιο δυναμικό με τις ρίζες: στην πραγματικότητα, είναι ρίζες. Στη συνέχεια είναι εύκολο να επεκτείνουμε τις αποδείξεις των Θεωρημάτων 6.1 και 6.2 σε έναν one-tree rp-heaps.



Σχήμα 7.2: Η παρουσίαση one-tree rp-hear που χρησιμοποιεί τρεις δείκτες ανά κόμβο .Οι unfair losers παίρνουν ένα προσωρινό rank -1 για να αποτρέψουν ένα rank reduction από την διάδοση μέσω αυτών.

8. Μπορεί η μείωση κλειδιού να γίνει πιο εύκολη;

Είναι φυσικό να αναρωτιέται κανείς αν υπάρχει ακόμα απλούστερος τρόπος για να γίνει μείωση κλειδιού, διατηρώντας ταυτόχρονα την amortized efficiency του σωρού Fibonacci. Δίνονται δύο απαντήσεις: "όχι" και "ίσως". "Όχι" δείχνοντας ότι δύο πιθανές μέθοδοι αποτυγχάνουν. Η πρώτη μέθοδος επιτρέπει αυθαίρετα αρνητικά rank differences αλλά με θετικά όρια στο rank difference. Με ένα τέτοιο rank rule, η διαδικασία rank reduction ακολουθεί μια μείωσης κλειδιού χρειάζεται να εξετάσει μόνο τους προγόνους του κόμβου των οποίων το κλειδί μειώνεται, όχι τα αδέρφια του. Μια τέτοια μέθοδος μπορεί να πάρει χρόνο $\Omega(\log n)$ ανά μείωση κλειδιού, όπως δείχνει το ακόλουθο παράδειγμα. Έστω b είναι η μέγιστη επιτρεπόμενη διαφορά rank. Επιλέξτε k αυθαίρετα. Μέσω μιας κατάλληλης αλληλουχίας ενθέσεων και διαγραφής ελαχίστων, δημιουργείτε ένας σωρός που περιέχει ένα τέλειο half tree με κάθε rank από το 0 έως $bk + 1$. Έστω x η ρίζα του half tree με rank $bk + 1$. Εξετάστε την δεξιά ραχοκοκαλιά $\text{left}(x)$. Μειώστε το κλειδί κάθε κόμβου σε αυτό το μονοπάτι του οποίου το rank δεν είναι διαιρετό από το b . Κάθε τέτοια μείωση κλειδιού παίρνει χρόνο $O(1)$ χρόνο και δεν παραβιάζει τον rank rule, έτσι κανένα rank δεν αλλάζει. Τώρα, η διαδρομή αποτελείται από $k + 1$ κόμβους, το καθένα με διαφορά rank b εκτός από το ανώτερο. (Βλ. Σχήμα 8.1). Μειώστε τα κλειδιά από αυτούς τους κόμβους, από το μικρότερο rank προς το μεγαλύτερο. Κάθε τέτοια μείωση κλειδιού θα προκαλέσει μια αλυσίδα μειώσεων σε όλο το μονοπάτι μέχρι τον κορυφαίο κόμβο του μονοπατιού. Ο συνολικός χρόνος για αυτές τις $k+1$ μειώσεις κλειδιών είναι $\Omega(k^2)$. Μετά από όλες τις μειώσεις κλειδιών, ο σωρός περιέχει τρία τέλεια half trees με rank μηδέν και δύο για κάθε rank από το 1 έως το bk . Η διαγραφή ελαχίστου (μιας από τις ρίζες με rank μηδέν) που ακολουθείται από μια ένθεση καθιστά το σωρό και πάλι από ένα σύνολο από perfect half trees, ένα για κάθε rank από το 0 έως το $bk + 1$. Κάθε εκτέλεση αυτού του κύκλου κάνει $O(\log n)$ μειώσεις κλειδιών, μια διαγραφή ελαχίστου, και μια εισαγωγή, και χρειάζεται $\Omega(\log^2 n)$ χρόνο.



Σχήμα 8.1 : Ένα half tree με rank $bk+1$ στο αντιπαράδειγμα για την μέθοδο της μείωσης κλειδιού η οποία επιτρέπει αυθαίρετα αρνητικά rank difference αλλά θετικά rank difference

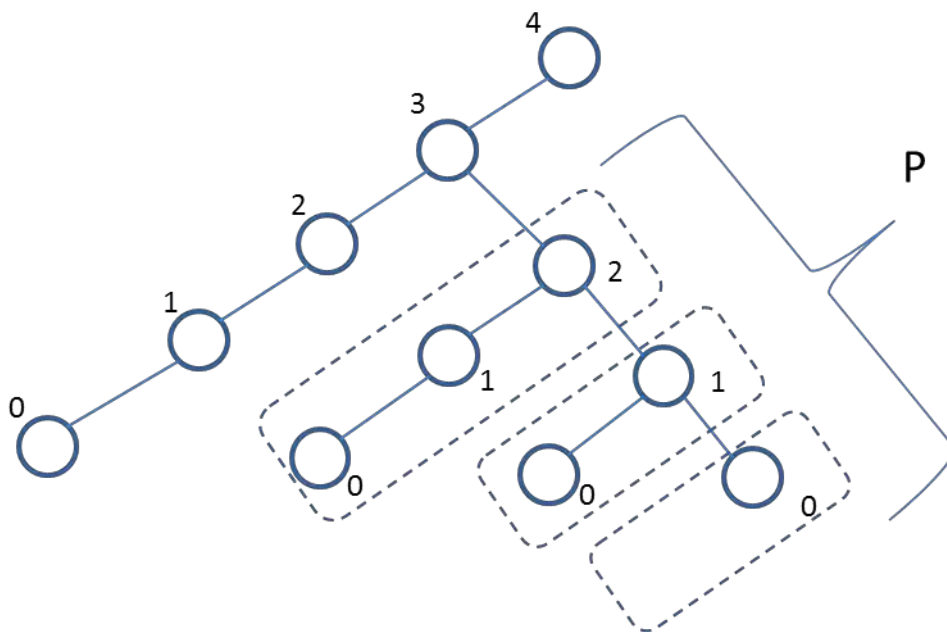
με φράγμα το b . Μια ακολουθία από $k+1$ μειώσεις κλειδιών στη δεξιά ραχοκοκαλιά του $\text{left}(x)$, από το μικρότερο rank στο μεγαλύτερο χρειάζεται συνολικό χρόνο $\Omega(k^2)$.

Η δεύτερη, ακόμη πιο απλούστερη μέθοδος ξοδεύει μόνο $O(1)$ χρόνο χειρότερης περίπτωσης (worst-case) σε κάθε μείωση κλειδιού, αποφεύγοντας έτσι την αυθαίρετη υπερχειλίση. Στην περίπτωση αυτή, κάνοντας αρκετές λειτουργίες μπορεί κάποιος να κατασκευάσει ένα half tree κάθε δυνατού rank , έως ένα rank που είναι $\omega(\log n)$. Μόλις αυτό ολοκληρωθεί, κατ'επανάληψη κάνουμε εισαγωγή που ακολουθείται από διαγραφή ελαχίστου (του στοιχείου που μόλις εισήχθη), αυτό θα έχει ως αποτέλεσμα σε κάθε διαγραφή ελαχίστου να χρειάζεται χρόνος $\omega(\log n)$. Παρακάτω παρουσιάζονται οι λεπτομέρειες. Έστω ότι κάθε μείωση κλειδιού αλλάζει τα ranks των κόμβων το πολύ d pointers μακριά από τον κόμβο του οποίου το κλειδί μειώνεται, όπου d είναι σταθερό. Επιλέξτε το k αυθαίρετα. Μέσω μιας κατάλληλης αλληλουχίας ενθέσεων και διαγραφής ελαχίστων, κατασκευάστε ένα σωρό που περιέχει ένα perfect half tree με όλα τα rank από 0 έως k . Σε κάθε κόμβο με απόσταση $d+2$ ή μεγαλύτερη από τη ρίζα, κατά φθίνουσα σειρά ανάλογα με την απόσταση, κάντε μια μείωση κλειδιού με $\Delta = \infty$ ακολουθούμενη από μια διαγραφή ελαχίστου. Καμία ρίζα δεν μπορεί να επηρεαστεί από τις πράξεις αυτές, οπότε ο σωρός εξακολουθεί να αποτελείται από half trees για κάθε rank του, αλλά κάθε half tree περιέχει το πολύ 2^{d+1} κόμβους, οπότε υπάρχουν τουλάχιστον $n/2^{d+1}$ half trees. Τώρα επαναλάβετε τον κύκλο μιας εισαγωγής που ακολουθείται από μια διαγραφή ελαχίστου. Κάθε τέτοιος κύκλος παίρνει $\Omega(n/2^{d+1})$ χρόνο. Η επιλογή του " $d+2$ " σε αυτή την κατασκευή εγγυάται ότι καμία μείωση κλειδιού μπορεί να φτάσει στο παιδί της ρίζας, και ως εκ τούτου δεν μπορεί να αλλάξει το rank της ρίζας (εκτός από τον κόμβο του οποίου μειώνεται το κλειδί).

Η κατασκευή αυτή λειτουργεί ακόμα και αν προστεθεί επιπλέον δείκτες στα half trees, όπως και στον σωρό Fibonacci. Έστω ότι έχουν προστεθεί διατεταγμένοι δείκτες που δείχνουν τους προγόνους στο half tree. Ακόμη και για μια τέτοια επαυξημένη δομή, η τελευταία αυτή κατασκευή δίνει ένα κακό παράδειγμα, εκτός από ότι το μέγεθος του κατασκευασμένου half tree με $\text{rank } k$ είναι $O(k^{d+1})$ αντί για $O(2^{d+1})$, και κάθε κύκλος μιας εισαγωγής που ακολουθείται από μια διαγραφή ελαχίστου λαμβάνει $\Omega(n^{1/(d+2)})$ χρόνο.

Ένας περιορισμός αυτής της κατασκευής είναι ότι η δημιουργία του αρχικού συνόλου των half trees, χρειάζεται έναν αριθμό πράξεων εκθετικό στο μέγεθος του σωρού στον οποίο η επαναλαμβανόμενη εισαγωγές και διαγραφές ελαχίστων έχουν γίνει. Έτσι, δεν είναι ένα αντιπαράδειγμα στο ακόλουθο ερώτημα: υπάρχει ένα σταθερό d έτσι ώστε αν σε κάθε μείωση κλειδιού ακολουθείται από το πολύ d βήματα rank-reduction (π.χ. τύπου 1), τότε ο amortized χρόνος είναι $O(1)$ ανά εισαγωγή, συγχώνευση και μείωση-κλειδιού, και $O(\log m)$ ανά delete-min, όπου m είναι ο συνολικός αριθμός εισαγωγών. Ένα παρόμοιο ερώτημα είναι αν ο σωρός Fibonacci χωρίς cuts τύπου καταρράκτη (συνεχόμενα) έχουν αυτά τα όρια. Έστω ότι η απάντηση είναι ναι, για κάποιο θετικό d , ίσως ακόμη και για $d = 1$. Το ακόλουθο αντιπαράδειγμα δείχνει ότι η απάντηση είναι όχι, για $d = 0$. Δηλαδή, η απάντηση είναι όχι για τη μέθοδο στην οποία μια μείωση κλειδιού δεν αλλάζει κανένα rank εκτός από τα ranks των ριζών. Για αυθαίρετο k , κατασκευάστε ένα half tree για κάθε rank από 0 έως k ,

το καθένα αποτελείται από ρίζα και ένα μονοπάτι του αριστερού παιδιού, προβαίνοντας επαγωγικά ως ακολούθως. Δίνοντας τέτοια half trees με ranks από 0 έως $k - 1$, εισάγεται ένα στοιχείο στο σωρό το οποίο είναι μικρότερο από όλα τα στοιχεία του σωρού και μετά κάντε k κύκλους, οι οποίοι ο καθένας αποτελείται από μια εισαγωγή που ακολουθείται από μια διαγραφή ελαχίστου, η οποία διαγράφει το στοιχείο που μόλις εισήλθε. Το αποτέλεσμα είναι ένα one half tree με rank k , που αποτελείται από την ρίζα, από ένα μονοπάτι από το αριστερό παιδί κατεβαίνοντας από την ρίζα, από το μονοπάτι P του δεξιού παιδιού κατεβαίνοντας από το αριστερό παιδί της ρίζας και ένα μονοπάτι από το αριστερό παιδί κατεβαίνοντας από κάθε κόμβο του P . Κάθε παιδί έχει rank difference 1 (Δείτε σχήμα 8.2). Κάντε ένα rank reduction για κάθε κόμβο του P . Αυτό έχει ως αποτέλεσμα την δημιουργία half trees με rank από 0 έως k , εκτός από το $k-1$ για κάθε κόμβο. Επαναλάβετε αυτή τη διαδικασία για το σύνολο των half trees μέχρι το rank $k-2$, με αποτέλεσμα την δημιουργία half trees με rank από 0 έως k , με το $k-2$ να λείπει. Συνεχίστε με αυτό τον τρόπο μέχρι να μείνει μόνο το rank 0 να λείπει, και τότε κάντε μια απλή εισαγωγή. Τώρα υπάρχουν half trees για κάθε rank, από 0 έως k . Ο συνολικός αριθμός των πράξεων σωρού που χρειάζονται για να αυξήσουν το maximum rank από $k - 1$ σε k είναι $O(k^2)$. Έτσι σε m πράξεις σωρού μπορεί κανείς να δημιουργήσει ένα σύνολο από half trees με κάθε δυνατό rank μέχρι το rank που είναι $O(m^{1/3})$. Στο σωρό που αντιπροσωπεύεται από αυτό το σύνολο των half trees, μια εισαγωγή που ακολουθείται από μια διαγραφή ελαχίστου χρειάζεται $O(m^{1/3})$ χρόνο, και αυτό το ζεύγος των πράξεων μπορεί να επαναληφθεί όσες φορές χρειάζεται.



Σχήμα 8.2 : Ένα half tree με rank $k=4$ κατασκευασμένο σε $O(k^3)$ πράξεις εάν η μείωση κλειδιού δεν αλλάζει τα ranks. Η μείωση κλειδιού στους κόμβους του P αποσπάει τα κυκλικά subtrees.

9. Παρατηρήσεις.

Παρουσιάστηκε μια νέα δομή δεδομένων, τον rank pairing heap, που συνδυάζει τις αποδόσεις του σωρού Fibonacci με την προσεγγιστική απλότητα του pairing heap. Όπως στον pairing heap, τα δέντρα στον rank pairing heap μπορούν να έχουν αυθαίρετη και ασύμμετρη δομή. Σε αντίθεση με τον pairing heap, ο rank pairing heap χρησιμοποιεί ranks για να εξασφάλιση την αποδοτικότητα (efficiency) του. Όπως ο Fredman [13] έδειξε, με την επιφύλαξη ορισμένων τεχνικών περιορισμών, μερικές πληροφορίες, όπως τα ranks πρέπει να αποθηκεύονται για τη διασφάλιση της αποτελεσματικότητας, η δομή δεδομένων μας υπακούει στους περιορισμούς για τα όρια του Fredman. Τα αποτελέσματά μας βασίζονται σε προηγούμενες εργασίες των Peterson, Høyer, Karlan και Tarjan, και άλλων, και μπορεί να είναι η φυσική κατάληξη αυτής της εργασίας: παρουσιάστηκε ότι οι απλούστερες μέθοδοι για να γίνει μείωση κλειδιού δεν έχουν την επιθυμητή απόδοση.

Ο Type-1 rp-heaps, παρόλο που είναι απλός, δεν είναι εύκολος στην ανάλυση. Πραγματικά ήταν έκπληξη όταν ανακαλύφτηκε ότι ο αλγόριθμος έχει την ίδια αποτελεσματικότητα με τον σωρό Fibonacci. Για τον type-2 rp-heaps, ήταν πιο αναμενόμενο. Κάποιος μπορεί να κάνει την ανάλυση ακόμη απλούστερη με τη χαλάρωση της δομής των δεδομένων, ακόμη περισσότερο, και συγκεκριμένα απαγορεύοντας 0,2-nodes, αλλά επιτρέποντας 1,3-nodes. Αυτό δίνει τον type-3 rp heaps: κάθε μη-ρίζα είναι ένας 1,1-node, ή ένας 1,2-node, ή ένας 1,3-node, ή ένας 0,i-node για κάποιο $i > 2$. Για αυτή τη δομή δεδομένων, δίνοντας σε κάθε ρίζα δυναμικό ένα και κάθε μη ρίζα δυναμικό ίσον με μείον δυο συν το άθροισμα των rank difference των παιδιών του. Ο συμβιβασμός είναι ότι το όριο του μεγέθους του subtree είναι χειρότερο για τον τύπο-3 από τον τύπο-2. Ο τύπος-2 φαίνεται να είναι ο πιο εύκολος, απλός και κατανοητός.

Τα προκαταρκτικά πειράματα μας δείχνουν ότι ο rank pairing heap είναι ανταγωνιστικός στην πράξη με τον pairing heap. Πολύ πιο εμπεριστατωμένα και προσεκτικά πειράματα πρέπει να γίνουν ακόμα για να συγκρίνουμε αυτές τις δομές με άλλες.

10. Παραδείγματα Πράξεων rank pairing heap.

Σε αυτήν την ενότητα σας παρουσιάζονται αναλυτικά παραδείγματα με όλες τις πράξεις οι οποίες μπορούν να εκτελεστούν με τον αλγόριθμο μας.

Πράξεις του αλγόριθμου rank pairing heap.

1) Insert(x , H): Ένθεση Στοιχείου x στο σωρό H

Θεωρία : Δημιουργείται ένα half tree με το στοιχείο που θέλουμε να ενθέσουμε, με rank 0, και το εισάγουμε στον σωρό.

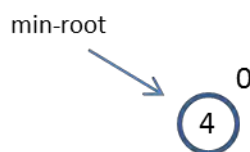
- Αν η ρίζα του είναι η μικρότερη από όλες τις ρίζες που υπάρχουν στο σωρό τότε το ενθέτουμε πρώτο και ενημερώνουμε τον δείκτη της λίστας για τη μικρότερη ρίζα.
- Αν δεν είναι τελικά η min-root την βάζουμε αμέσως μετά την min-root (δεξιά της.)

Παράδειγμα 1:

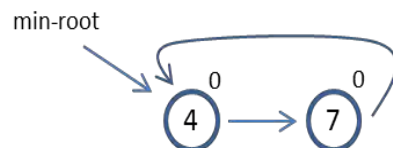
Ενθέτουμε διαδοχικά σε άδειο σωρό τα εξής στοιχεία:

4 , 7 , 2 , 9 , 5 , 18 , 23 , 12 , 10 , 1

Ένθεση του 4:

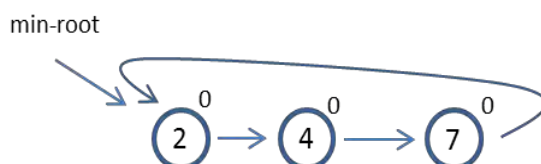


Ένθεση του 7:



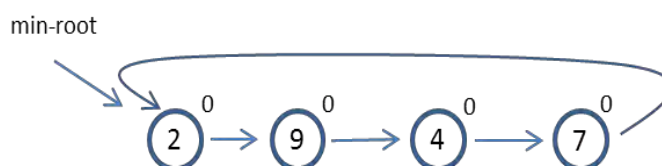
Ενημέρωση του min-root

Ένθεση του 2:

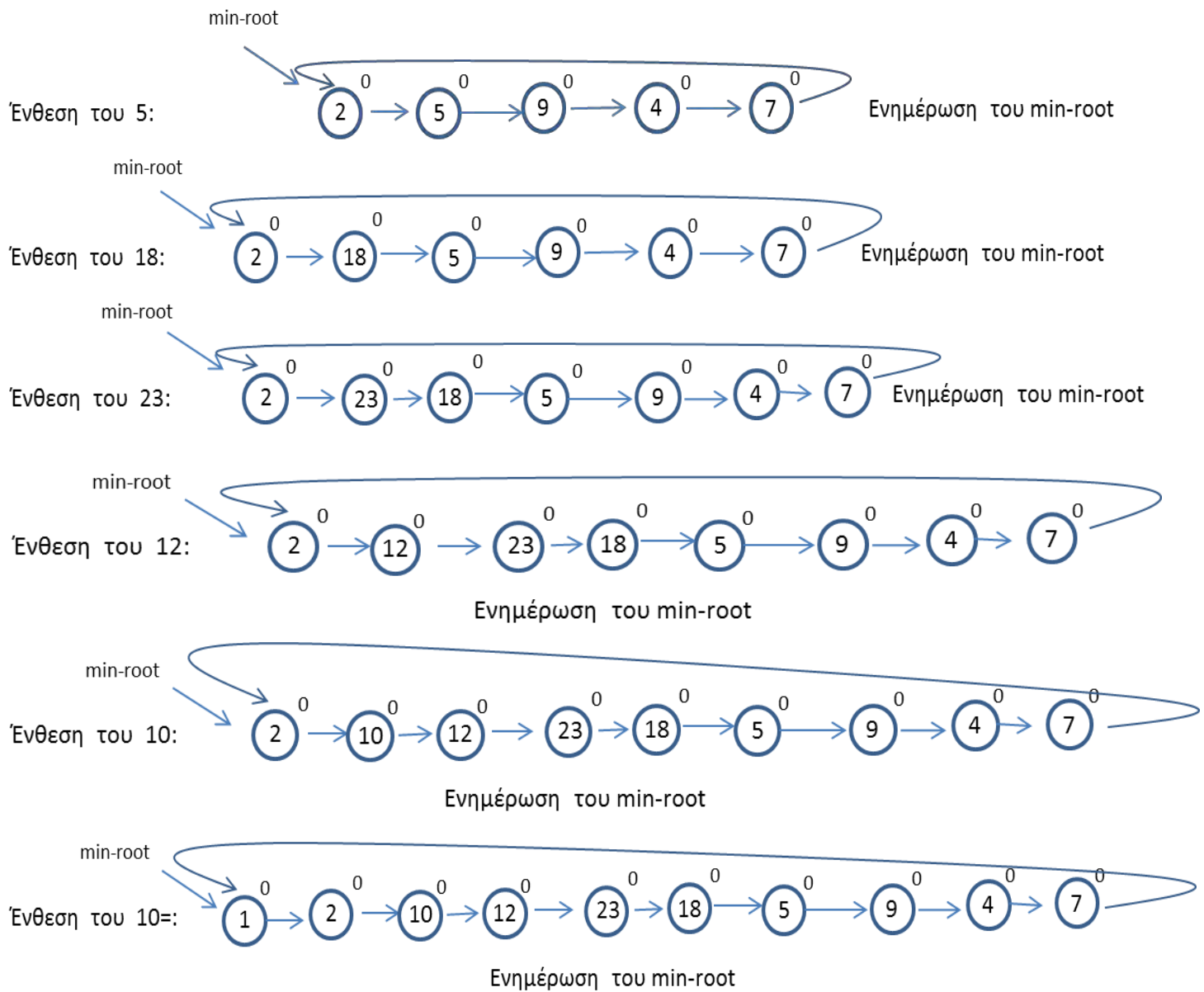


Ενημέρωση του min-root

Ένθεση του 9:

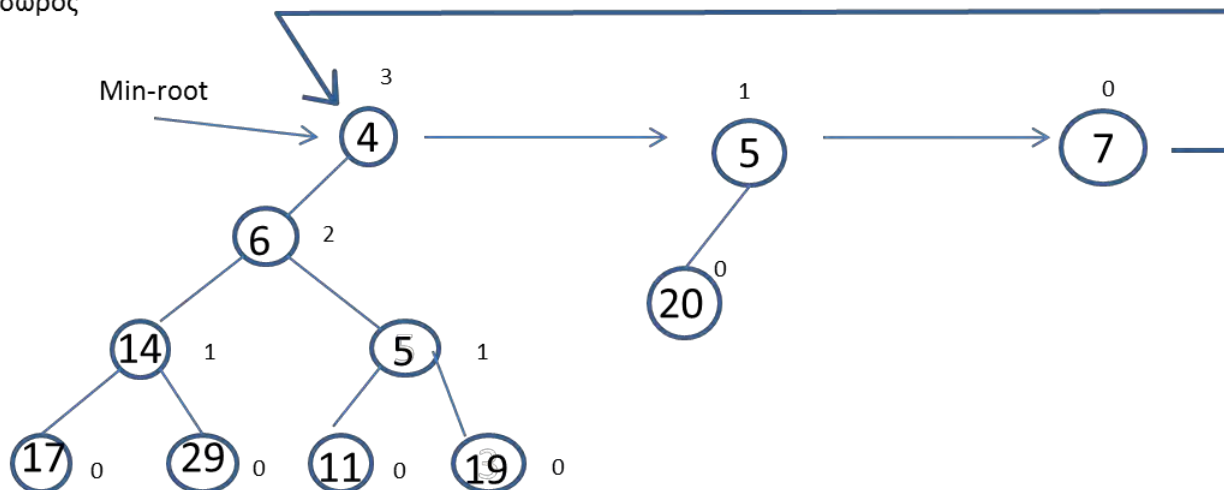


Ενημέρωση του min-root

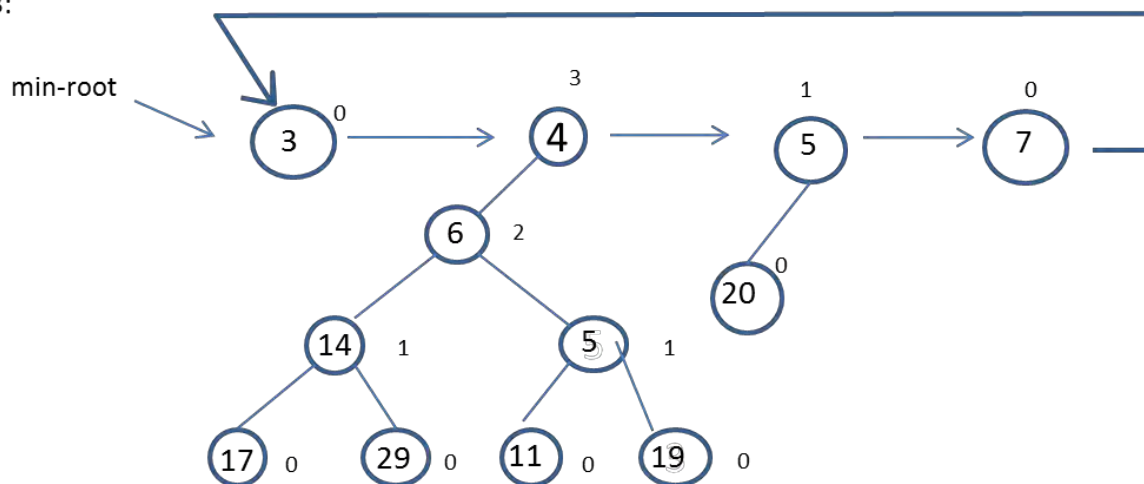


Παράδειγμα 2: Έστω ότι έχουμε τον παρακάτω σωρό και θέλουμε να ενθέσουμε διαδοχικά τα στοιχεία 3, 10, 44

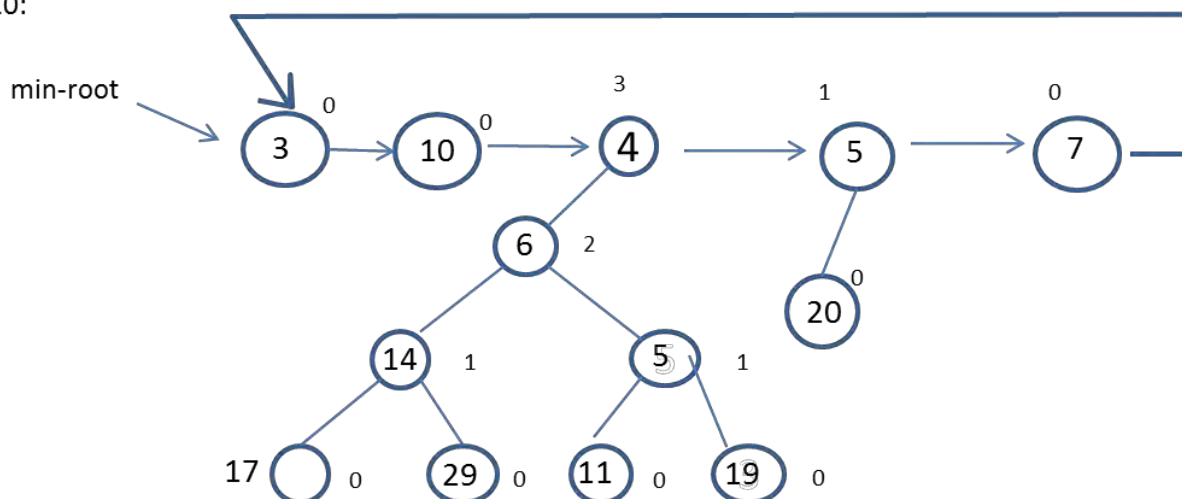
Αρχικός σωρός



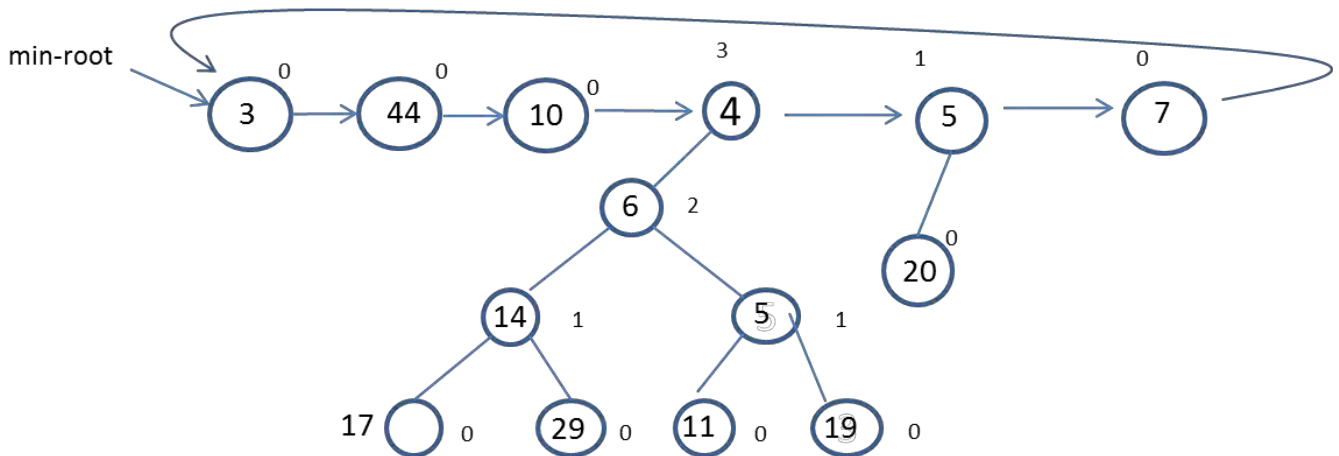
Ένθεση του 3:



Ένθεση του 10:



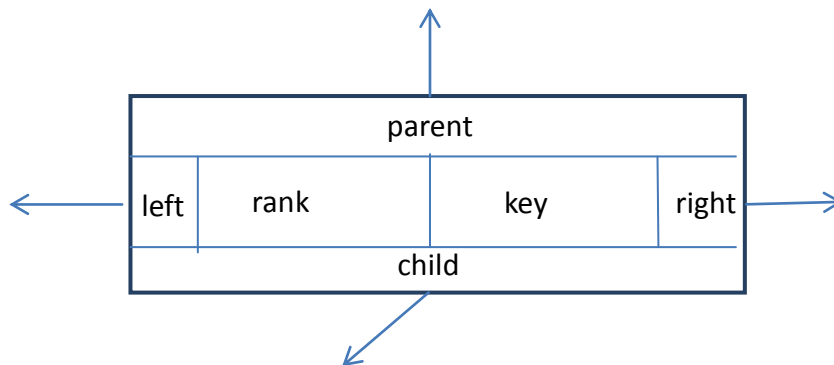
Ένθεση του 44:



2) Make heap : Επιστρέφει έναν νέο , άδειο σωρό

Για να δημιουργήσουμε έναν νέο άδειο σωρό, απλά δημιουργούμε μία άδεια λίστα με ρίζες.

Η ρίζα του σωρού σε αναπαράσταση ως κόμβος λίστας

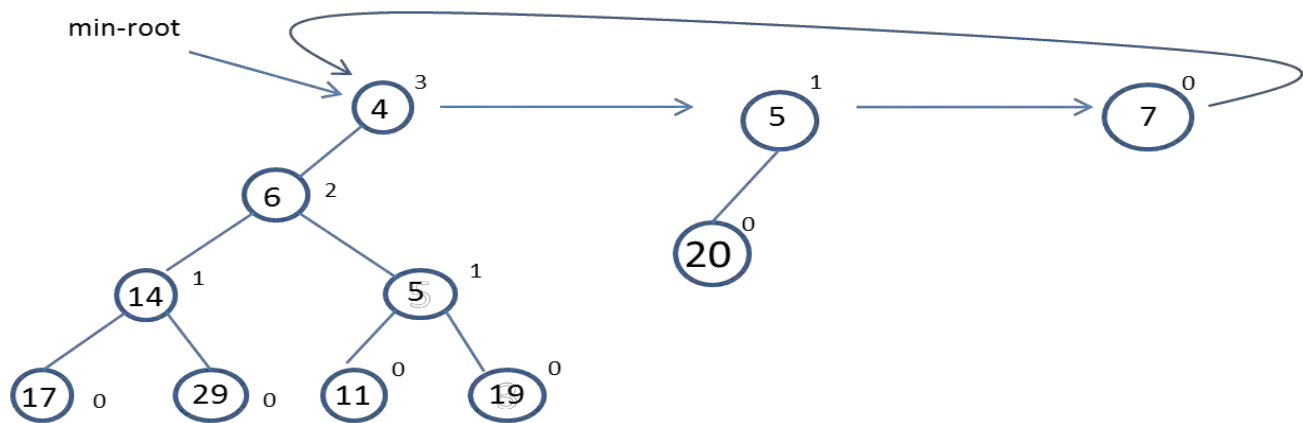


Οπότε για να δημιουργήσουμε έναν κενό σωρό φτιάχνουμε μια λίστα με έναν κόμβο τον οποίο θέτουμε με NULL.

3)Find-min(H) : Εύρεση μικρότερου στοιχείου στο σωρό H

Το πρώτο στοιχείο της λίστας δείχνει το μικρότερο στοιχείο της σωρού με δένδρα half tree.

Παράδειγμα:



Η πράξη $\text{find-min}(H)$ επιστρέφει τον κόμβο 4.

4) $\text{Delete-min}(H)$: Διαγραφή μικρότερου στοιχείου στο σωρό H

Άλλοι χρησιμοποιούν τον αλγόριθμο one-pass binomial queue και άλλοι τον αλγόριθμο multipass binomial queue

Εμείς θα χρησιμοποιήσουμε τον αλγόριθμο multipass binomial queue.

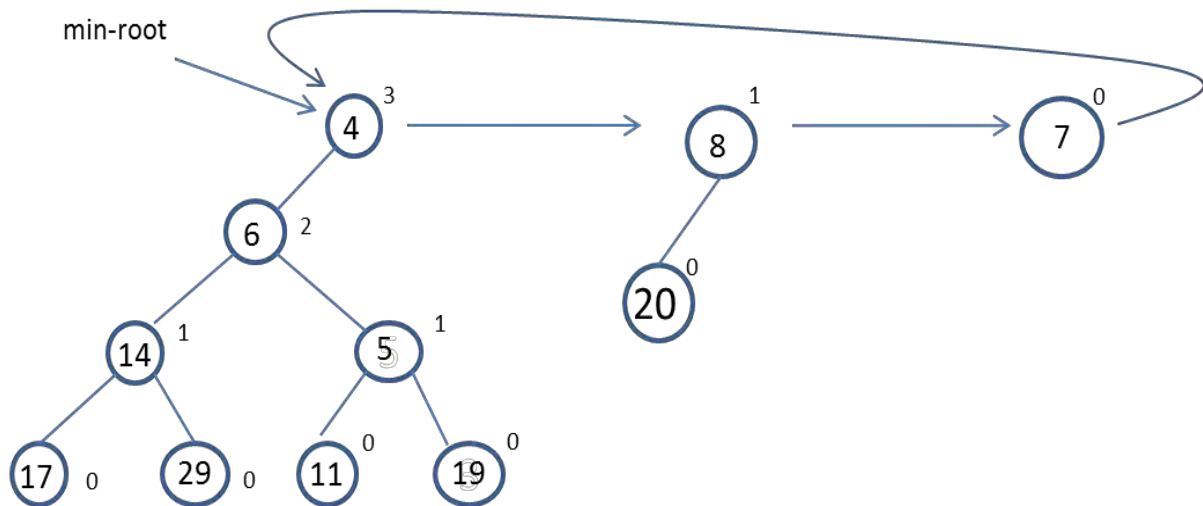
Υπάρχει μια μικρή αλλαγή στην υλοποίηση:

Το μόνο που αλλάζει είναι όταν στην αρχή κάνουμε αποσυναρμολόγηση του δένδρου εκεί βάζουμε σε κάθε νέα ρίζα τον εξής rank:

$\text{rank} = \text{rank}(\text{από το αριστερό του παιδί}) + 1$

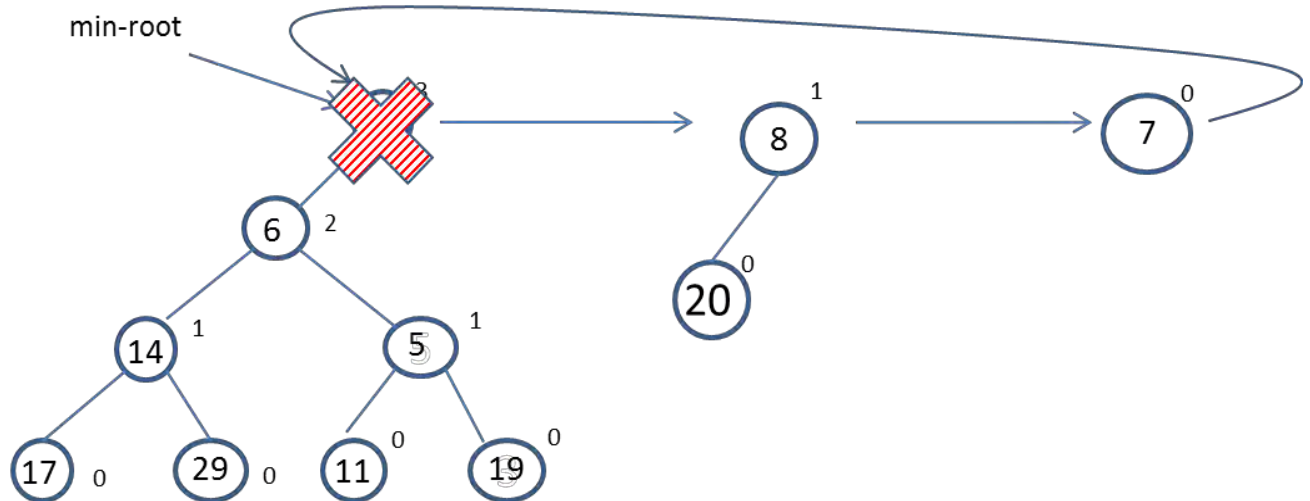
Παράδειγμα :Έστω το παρακάτω δένδρο:

Διαγραφή ελάχιστου στοιχείου



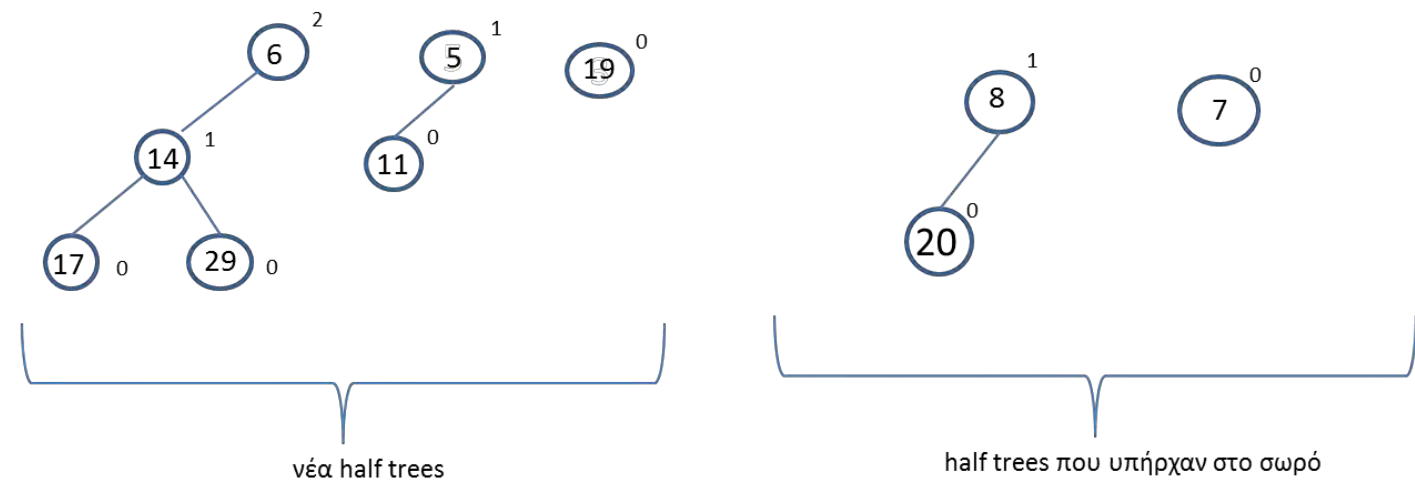
α)

Διαγραφή του 4.

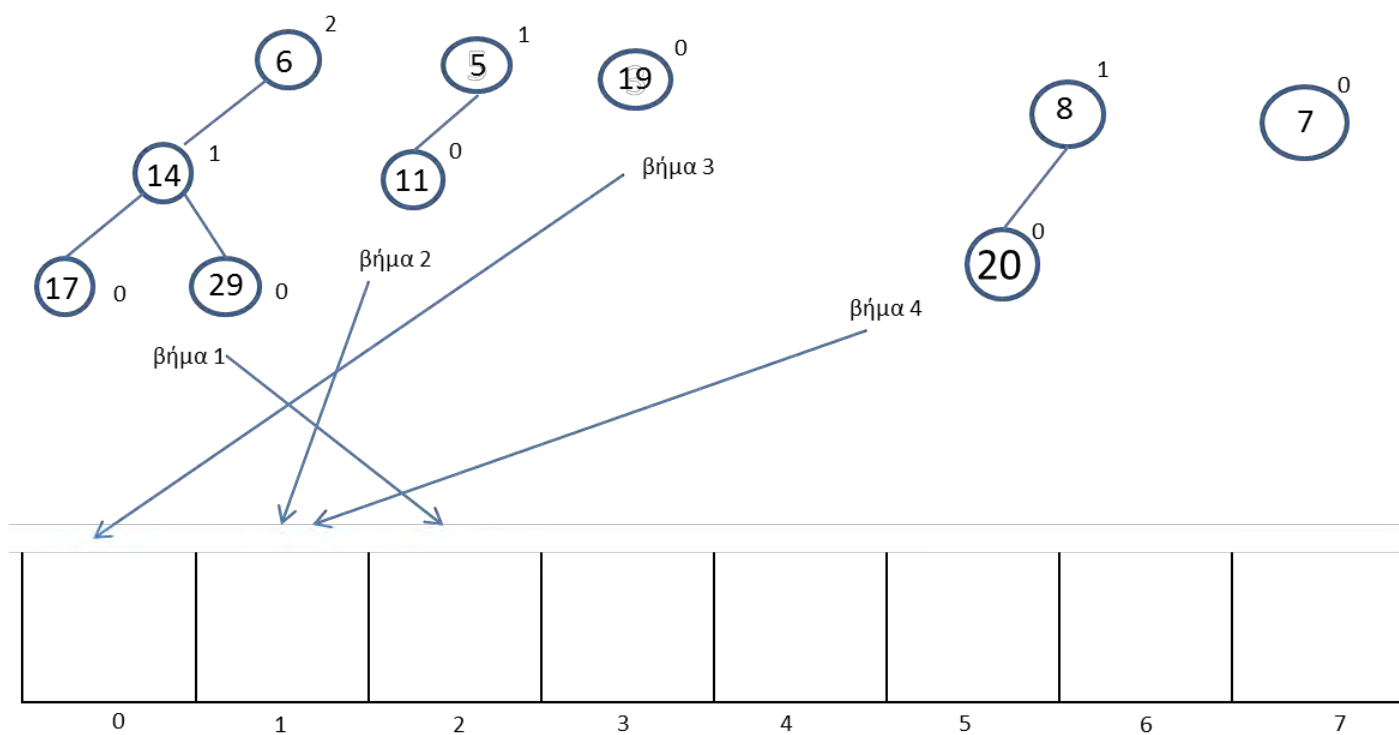


β)

Αποσυναρμολόγηση του δένδρου min-root , με κάθε κόμβο της δεξιάς ραχοκοκαλιάς του αριστερού παιδιού της ρίζας να γίνεται νέα ρίζα στα νέα half trees

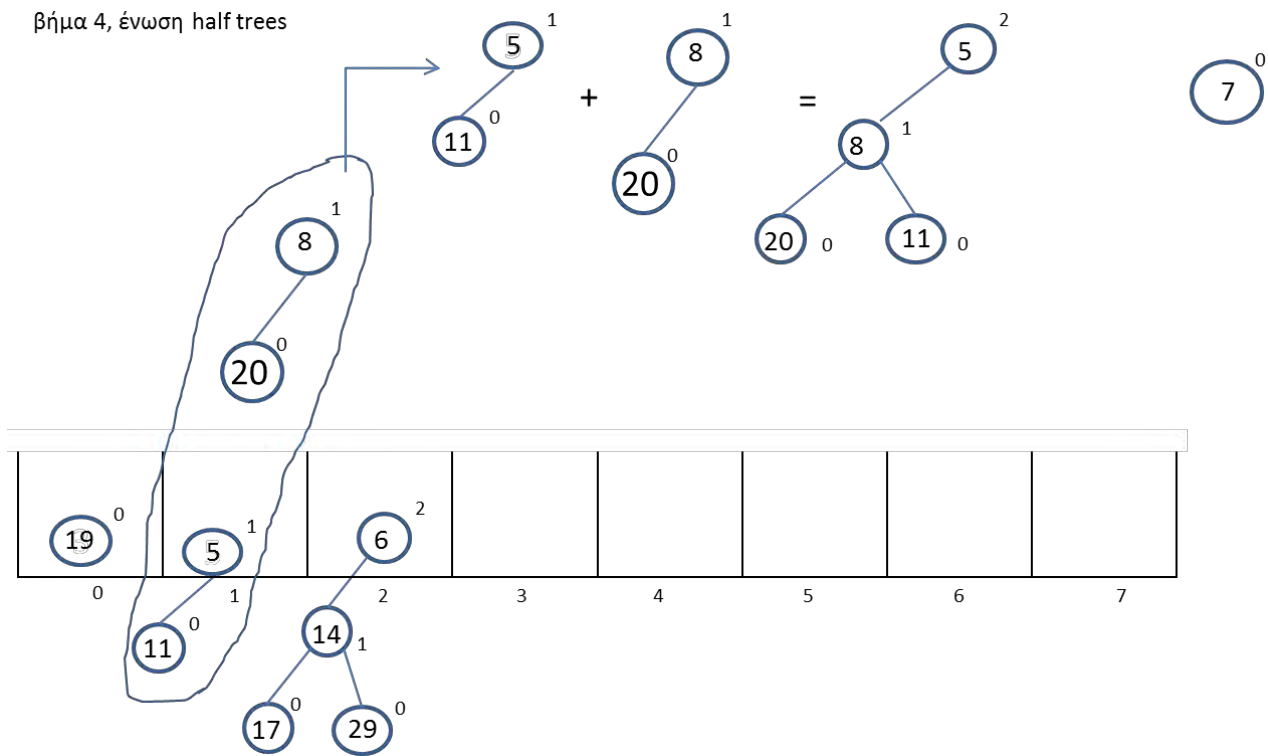


γ)



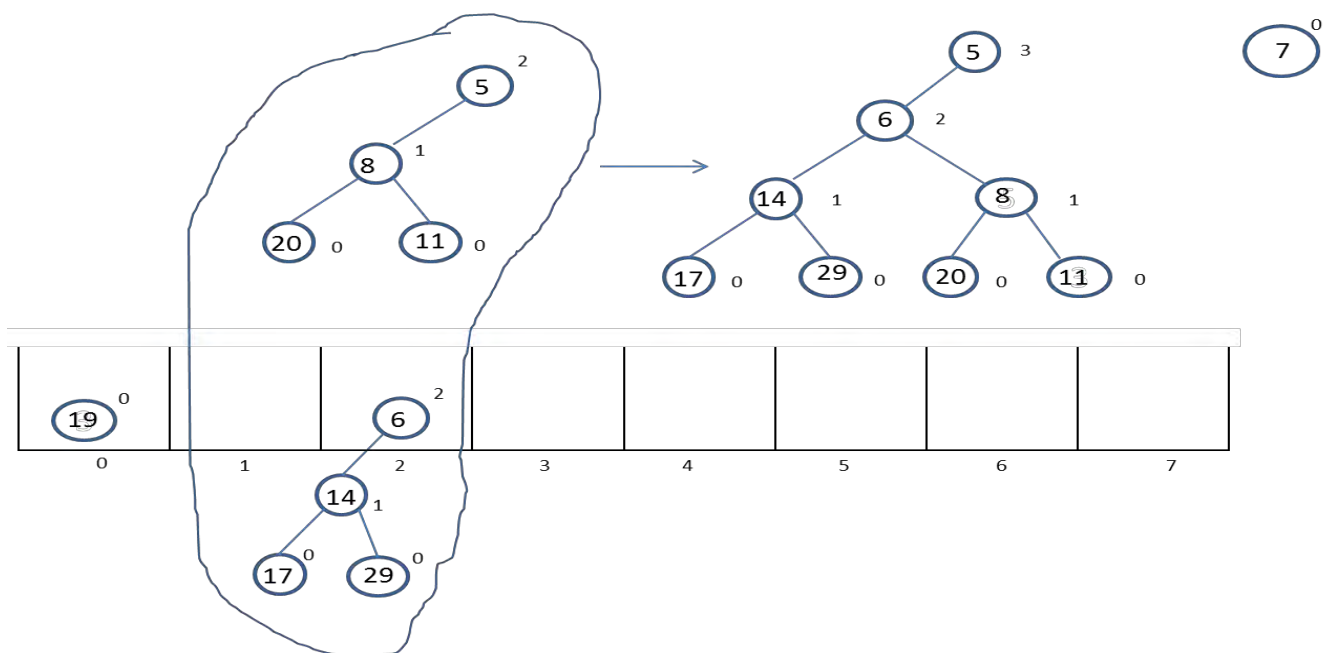
δ)

βήμα 4, ένωση half trees



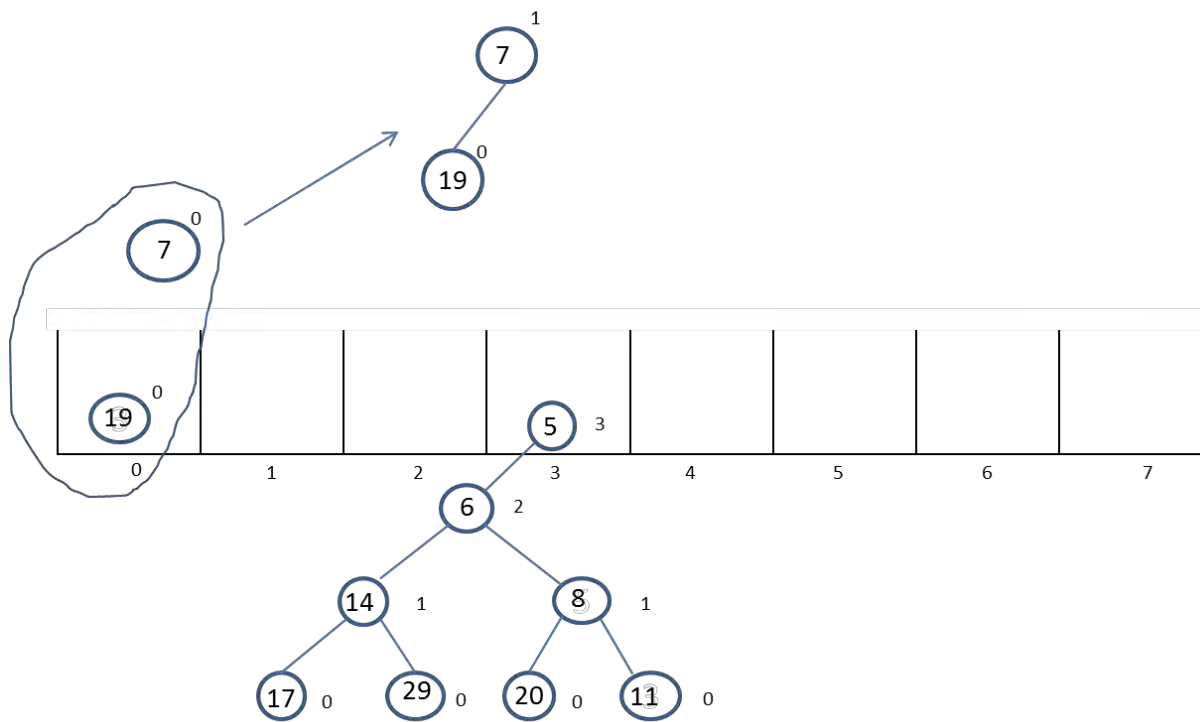
ε)

βήμα 5: εισαγωγή της ένωσης στον επόμενο bucket, η οποία προκαλεί εκ νέου ένωση (ένωση half trees με rank 2 και δημιουργία half tree με rank 3)



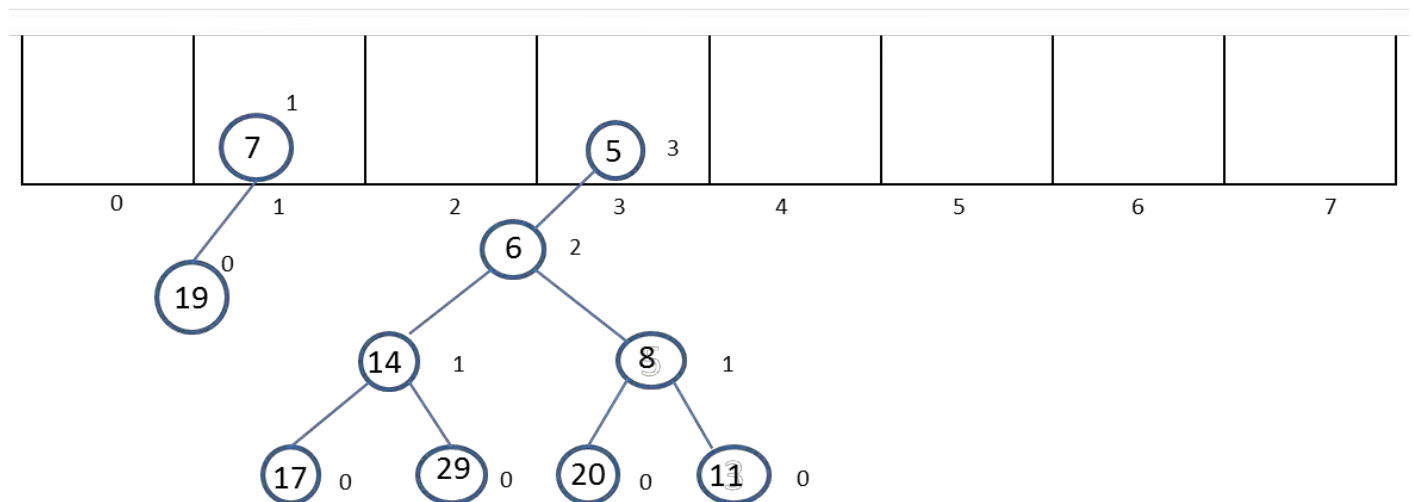
ζ)

βήμα 6: ένωση δυο half trees με rank 0 και δημιουργία νέου half tree με rank 1



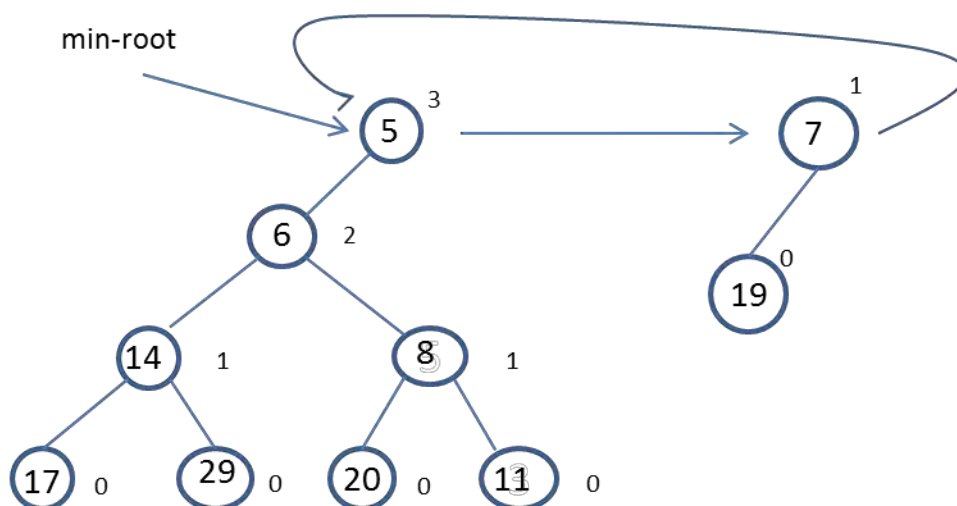
η)

Δεν έχουμε άλλα δένδρα να εισάγουμε στα buckets



θ)

Τελικό αποτέλεσμα (ενημερώνουμε και τον δείκτη για την min-root)

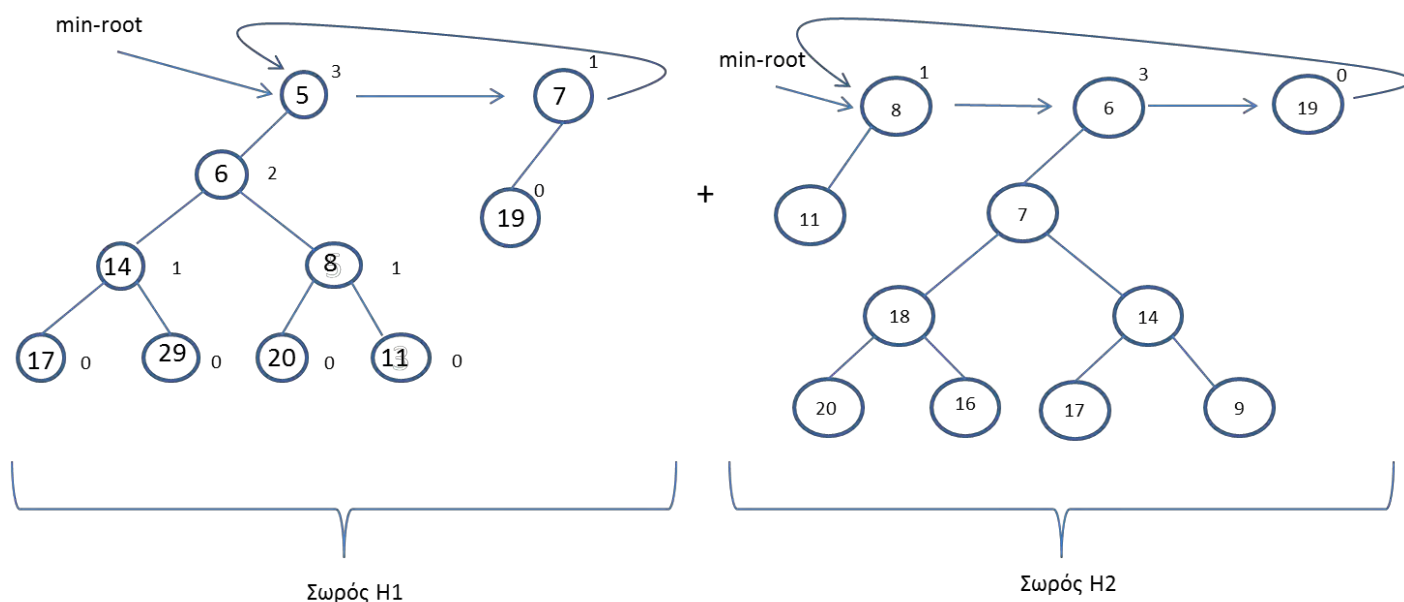


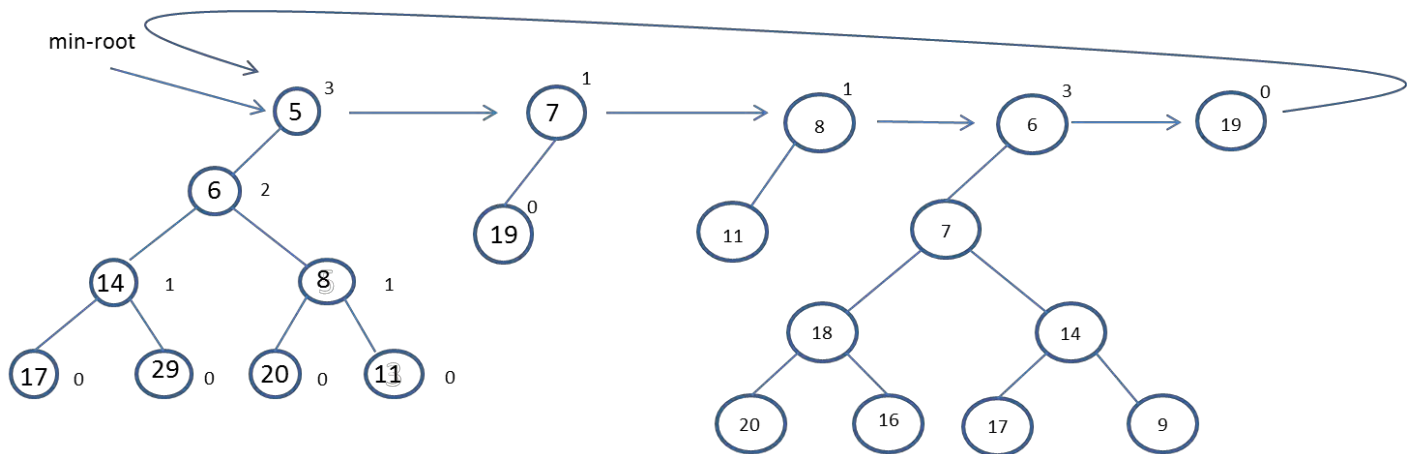
η)

5) Meld(H1,H2) : Συνένωση δυο σωρών

Έχουμε δυο λίστες με δυο δείκτες που δείχνουν τις ρίζες με το μικρότερο στοιχείο. Συγκρίνουμε αυτές τις ρίζες, βάζουμε την μικρότερη στην αρχή της νέας λίστας και συνενώνουμε τις λίστες όπως στο παράδειγμα.

Παράδειγμα:





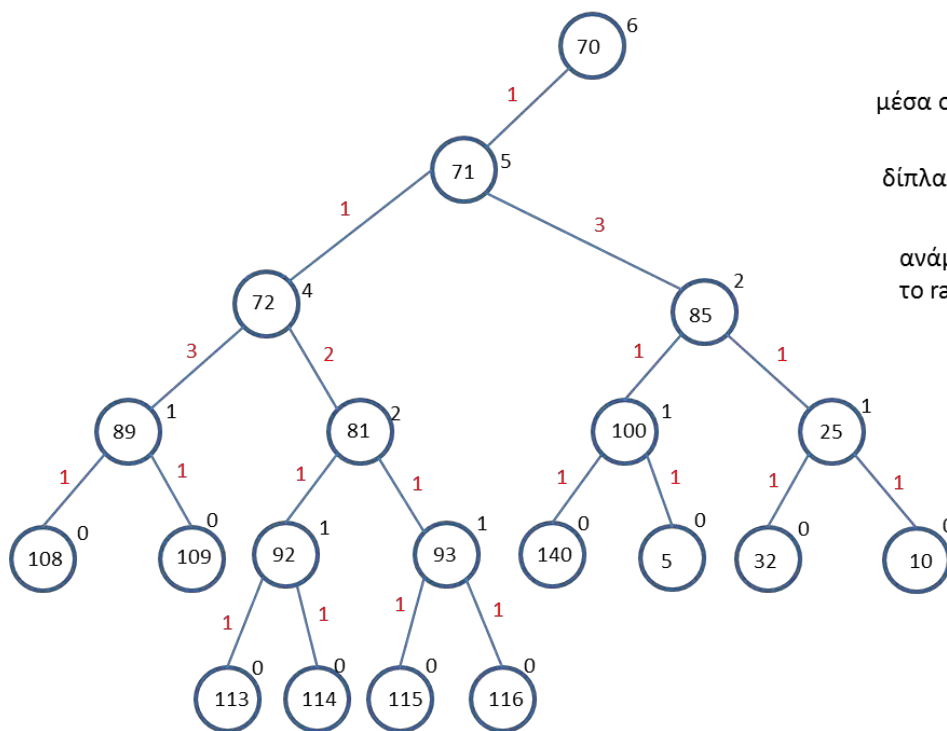
Νέος σωρός μετά την συγχώνευση δυο σωρών

6) Decrease(x, Δ, H) : Μείωση του κλειδιού του στοιχείου x κατά Δ στο σωρό H

Decrease(x, Δ, H) :

- Αποκόπτουμε τον κόμβο x και το αριστερό sub tree του(δημιουργείτε ένα νέο half tree στο σωρό)
- Αντικαθιστούμε τον κόμβο x με το δεξί παιδί του.
- Βάλε στο x τη νέα τιμή κλειδιού
- Βάλε το νέο half tree με ρίζα το x (με την νέα τιμή κλειδιού) στο σωρό με τις ρίζες
- Το νέο half tree με ρίζα το x έχει νέο rank = το rank του αριστερού του παιδιού +1
- Ενημέρωση του σωρού για το μικρότερη ρίζα

Αφού γίνει όλο αυτό ελέγχουμε στο δένδρο που έγινε η αποκοπή τους rank rules



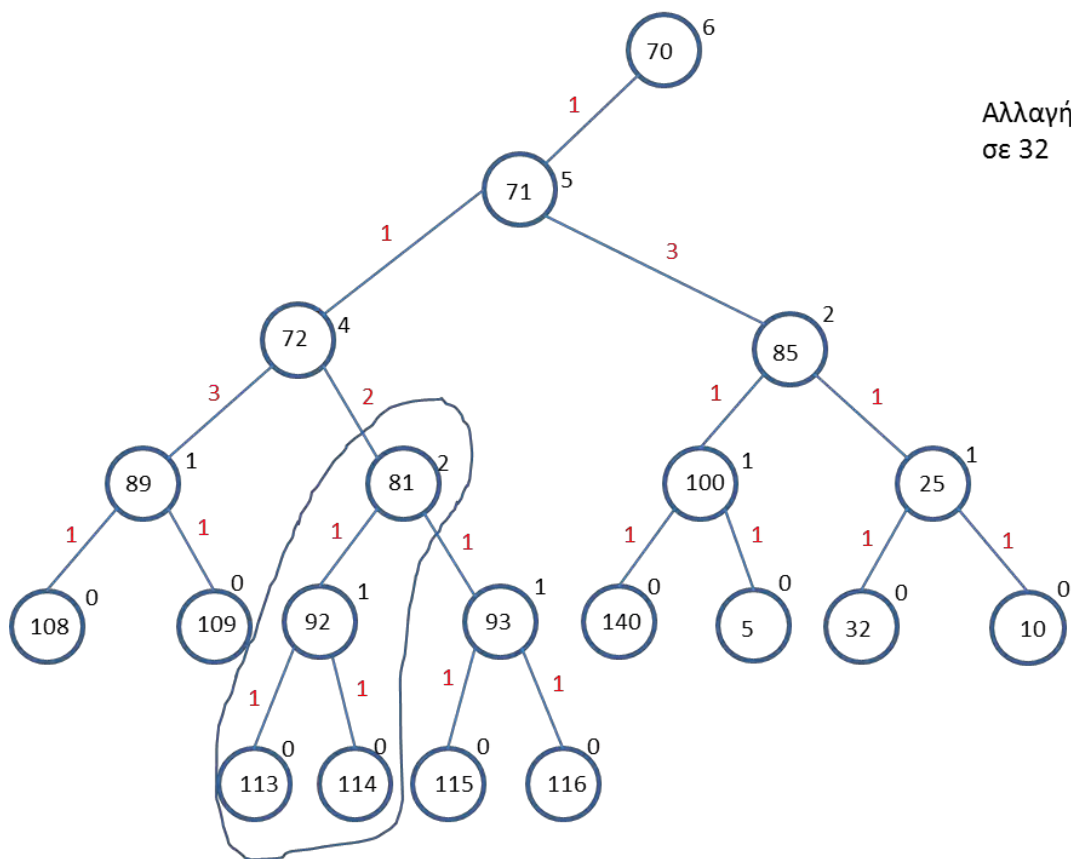
half tree

μέσα στον κόμβο: το κλειδί του κόμβου

δίπλα στον κόμβο: το rank του κόμβου

ανάμεσα στους κόμβους (κόκκινο) :
το rank difference πατέρα - παιδιού

α)

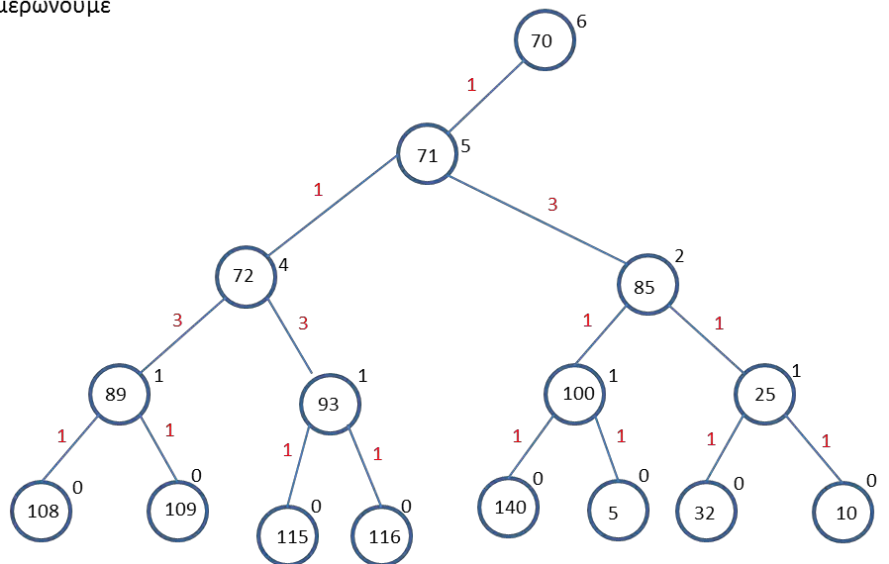
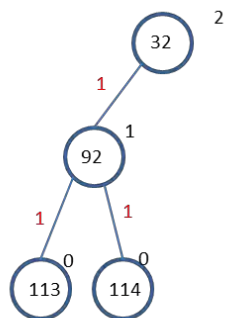


half tree

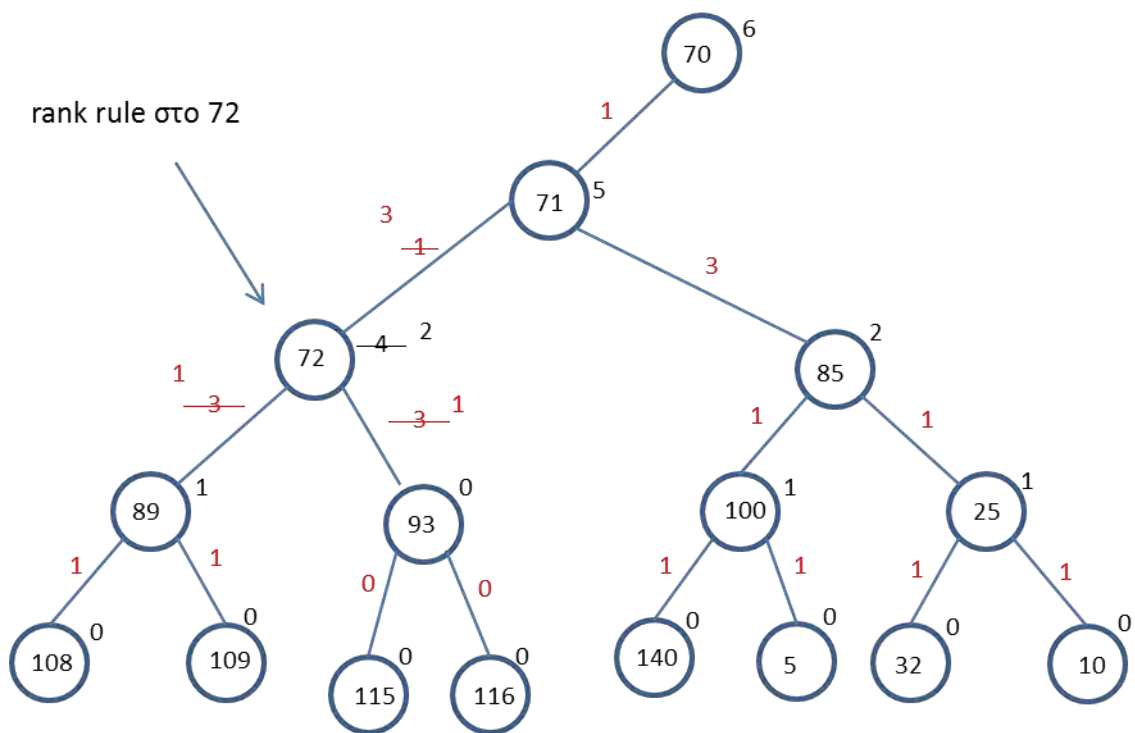
Αλλαγή κλειδιού του κόμβου 81
σε 32

β)

Ο κόμβος 32 μπαίνει στο σωρό ως ρίζα και ενημερώνουμε
Το rank του: όσο το παιδί του + 1

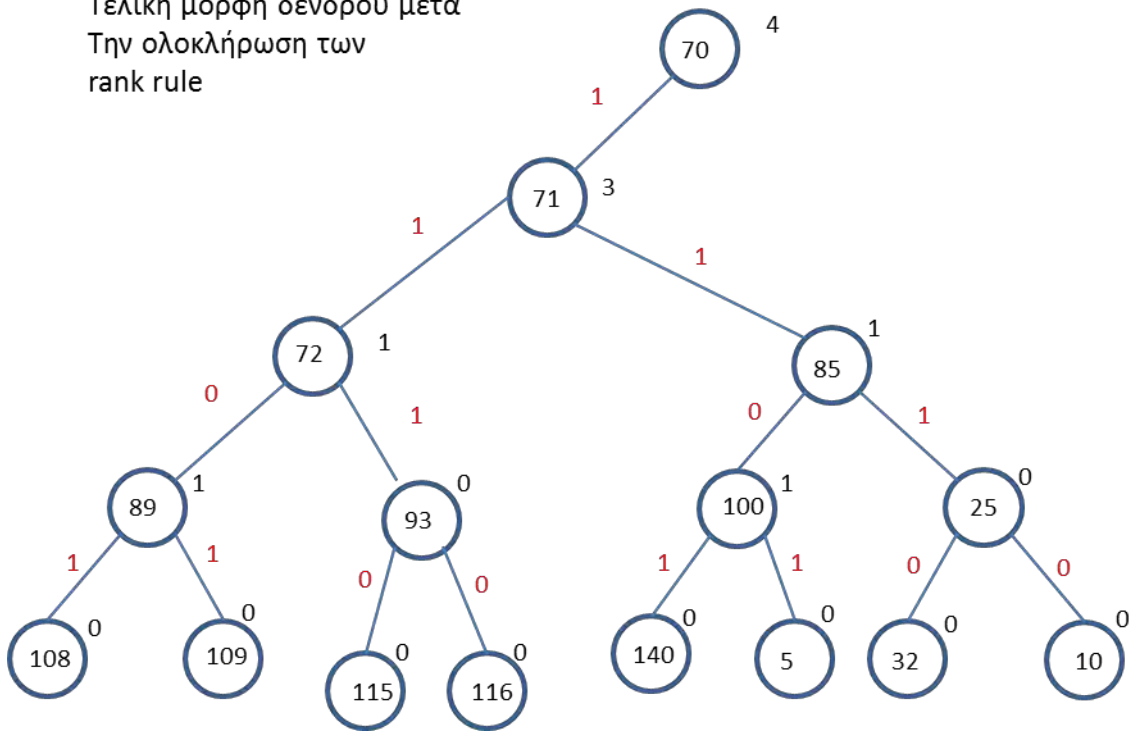


γ)



δ)

Τελική μορφή δένδρου μετά
Την ολοκλήρωση των
rank rule



η)

7) Delete(x, H) : Διαγραφή του στοιχείου x από των σωρό H

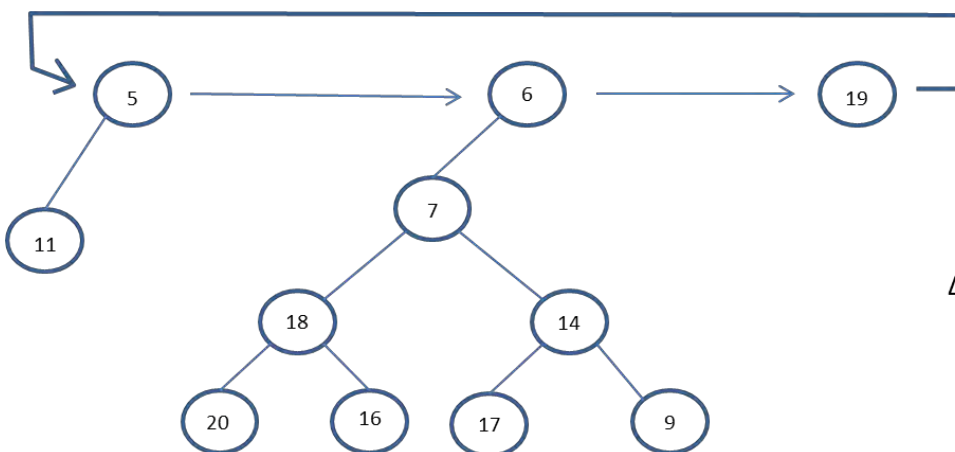
Η delete(x, H) πράξη γίνεται σε 2 στάδια:

- Πρώτα κάνουμε το κλειδί του x $-\infty$, ώστε να γίνει ρίζα στο δένδρο στο οποίο ανήκει
- Δεύτερον αφού έγινε ρίζα με την μικρότερη τιμή στο σωρό κάνουμε την πράξη

delete-min

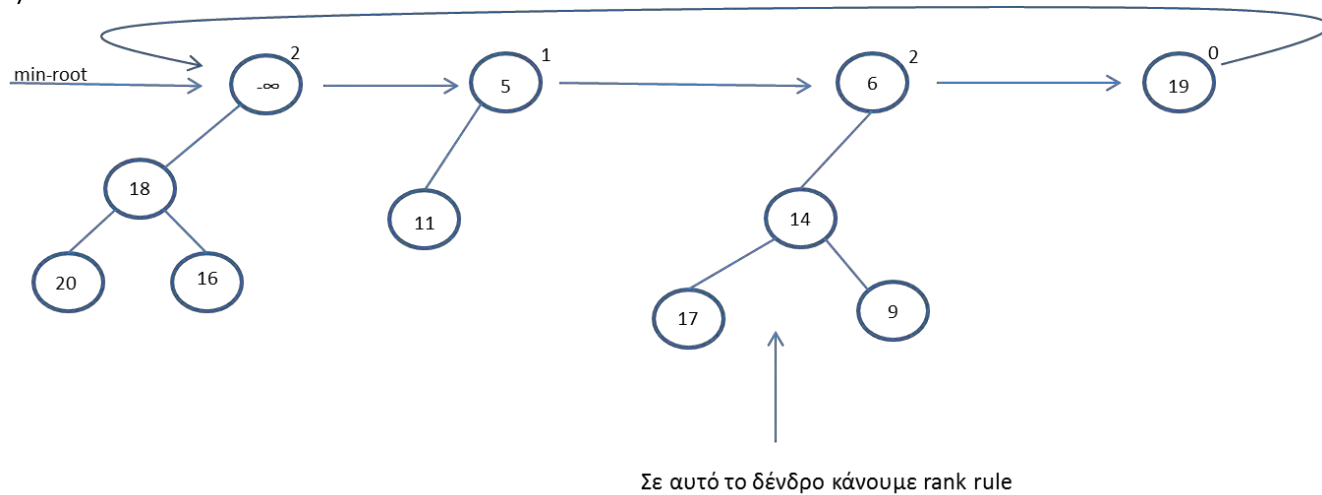
delete(x, H): decrease-key($x, -\infty, H$);

delete-min(H)



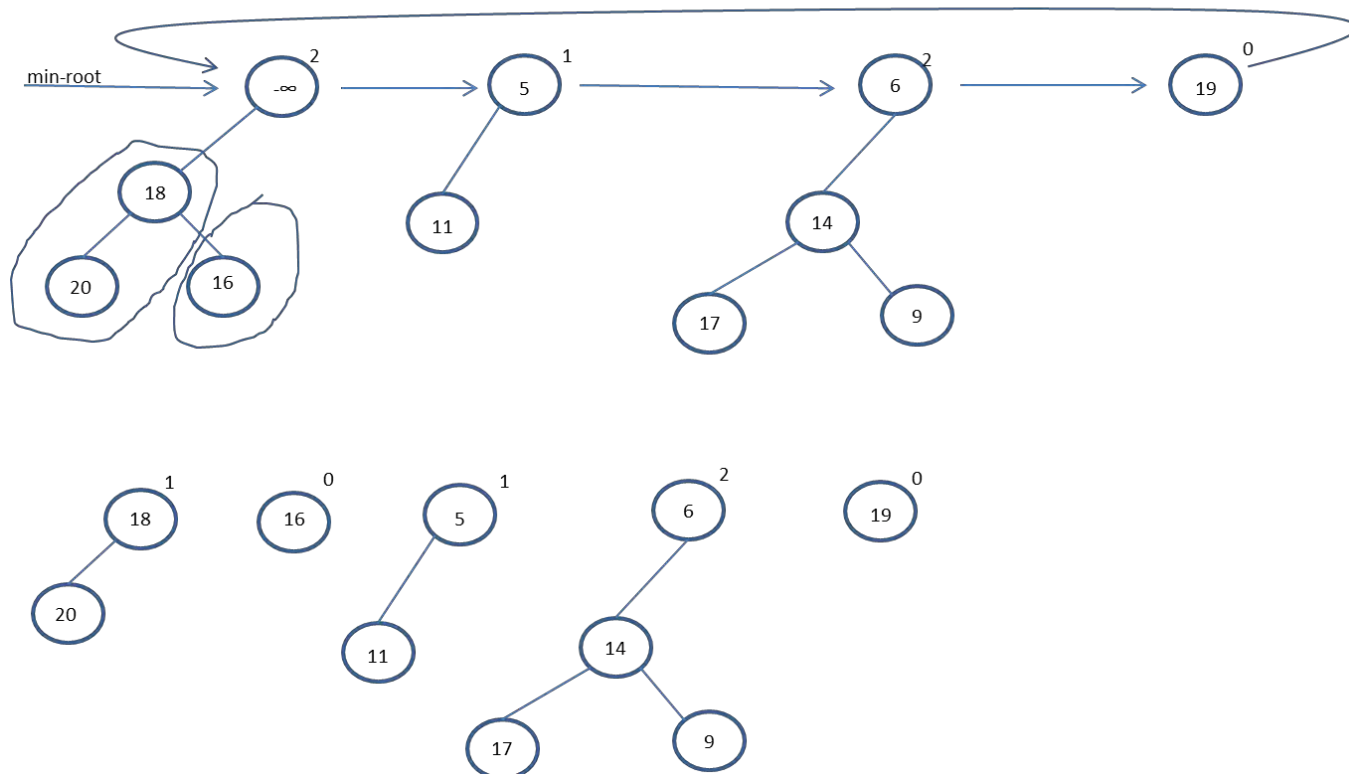
Διαγραφή του κόμβου με το κλειδί 7

α)



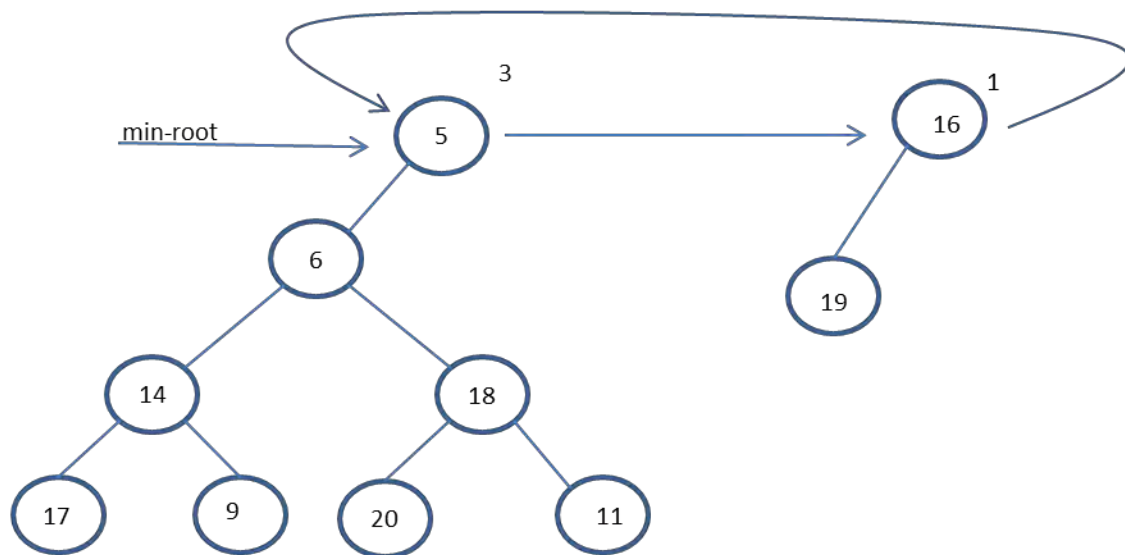
β)

Τώρα έχουμε decrease key



γ)

Με την λογική των buckets έχουμε τελικό αποτέλεσμα



δ)

Αναφορές

- G. M. Adel'son-Vel'skii and E. M. Landis, An algorithm for the organization of information, *Sov. Math. Dokl.*, 3 (1962), pp. 1259–1262.
- G. S. Brodal, Worst-case efficient priority queues, in *Proceedings of the Seventh ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1996, pp. 52–58.
- G. S. Brodal and C. Okasaki, Optimal purely functional priority queues, *J. Funct. Programming*, 6 (1996), pp. 839–857.
- M. R. Brown, Implementation and analysis of binomial queue algorithms, *SIAM J. Comput.*, 7 (1978), pp. 298–319.
- T. M. Chan, Quake Heaps: A Simple Alternative to Fibonacci Heaps, 2009; available online at <http://www.cs.uwaterloo.ca/~tmchan/heap.ps>.
- E. W. Dijkstra, A note on two problems in connexion with graphs, *Numer. Math.*, 1 (1959), pp. 269–271.
- J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, *Comm. ACM*, 31 (1988), pp. 1343–1354.
- R. D. Dutton, The Weak-Heap Data Structure, Technical report CS-TR-92-09, University of Central Florida, Orlando, FL, 1992.
- J. Edmonds, Optimum branchings, *J. Res. Nat. Bur. Standards*, B71 (1967), pp. 233–240.
- A. Elmasry, Pairing heaps with $O(\log \log n)$ decrease cost, in *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2009, pp. 471–476.
- A. Elmasry, The violation heap: A relaxed Fibonacci-like heap, in *Proceedings of the 16th International Conference on Computing and Combinatorics (COCOON)*, 2010, pp. 479–488.
- A. Elmasry, C. Jensen, and J. Katajainen, Two-tier relaxed heaps, *Acta Inform.*, 45 (2008), pp. 193–210.
- M. L. Fredman, On the efficiency of pairing heaps and related data structures, *J. ACM*, 6 (1999), pp. 473–501.
- M. L. Fredman, R. Sedgwick, D. D. Sleator, and R. E. Tarjan, The pairing heap: A new form of self-adjusting heap, *Algorithmica*, 1 (1986), pp. 111–129.
- M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM*, 34 (1987), pp. 596–615.
- M. L. Fredman and D. E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *J. Comput. System Sci.*, 48 (1994), pp. 533–551.

- H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, Efficient algorithms for finding minimum spanning trees in undirected and directed graphs, *Combinatorica*, 6 (1986), pp. 109–122.
- L. J. Guibas and R. Sedgwick, A dichromatic framework for balanced trees, in *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1978, pp. 8–21.
- B. Haeupler, S. Sen, and R. E. Tarjan, Rank-pairing heaps, in *Proceedings of the 17th European Symposium on Algorithms (ESA)*, 2009, pp. 659–670.
- Y. Han and M. Thorup, Integer sorting in $O(n \log \log n)$ expected time and linear space, in *Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002, pp. 135–144.
- P. Høyer, A general technique for implementation of efficient priority queues, in *Proceedings of the Third Israeli Symposium on the Theory of Computing and Systems (ISTCS)*, 1995, pp. 57–66.
- H. Kaplan, N. Shafrir, and R. E. Tarjan, Meldable heaps and boolean union-find, in *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC)*, 2002, pp. 573–582.
- H. Kaplan and R. E. Tarjan, New Heap Data Structures, Technical report TR-597-99, Department of Computer Science, Princeton University, Princeton, NJ, 1999.
- H. Kaplan and R. E. Tarjan, Thin heaps, thick heaps, *ACM Trans. Algorithms*, 4 (2008), pp. 1–14.
- D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, Upper Saddle River, NJ, 1973.
- D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
- [27] A. M. Liao, Three priority queue applications revisited, *Algorithmica*, 7 (1992), pp. 415–427.
- B. M. E. Moret and H. D. Shapiro, An empirical analysis of algorithms for constructing a minimum spanning tree, in *Proceedings of the 2nd Workshop on Algorithms and Data Structures (WADS)*, 1991, pp. 400–411.
- G. L. Peterson, A Balanced Tree Scheme for Meldable Heaps with Updates, Technical report GIT-ICS-87-23, School of Informatics and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.
- S. Pettie, Towards a final analysis of pairing heaps, in *Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2005, pp. 174–183.

J. T. Stasko and J. S. Vitter, Pairing heaps: Experiments and analysis, *Comm. ACM*, 30 (1987), pp. 234–249.

T. Takaoka, Theory of trinomial heaps, in *Proceedings of the Sixth International Conference on Computing and Combinatorics (COCOON)*, 2000, pp. 362–372.

T. Takaoka, Theory of 2-3 heaps, *Discrete Appl. Math.*, 126 (2003), pp. 115–128.

R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Disc. Meth.*, 6 (1985), pp. 306–318.

M. Thorup, Integer priority queues with decrease key in constant time and the single source shortest paths problem, *J. Comput. System Sci.*, 69 (2004), pp. 330–353.

M. Thorup, Equivalence between priority queues and sorting, *J. ACM*, 54 (2007), 28.

J. Vuillemin, A data structure for manipulating priority queues, *Comm. ACM*, 21 (1978), pp. 309–315.

```

#include<stdio.h>

#include<conio.h>

#include<math.h>

#include<malloc.h>

#define NIL 0

#define PLIN_APEIRO -100


int main();

void create_heap();

void insert(struct rp_heap_node * tree_root, int val) ;

void display_heap(struct rp_heap_node *min1) ;

void create_left_tree(int k0,int k1,int k2,int k3);

void create_left_tree2(int i0,int i1,int i2,int i3,int i4, int i5, int i6 , int i7);

void create_left_tree(int s0, int s1);

struct rp_heap_node * addtree(struct rp_heap_node *a,struct rp_heap_node *b);

void findmin() ;

int deletemin();

struct rp_heap_node * find_key(struct rp_heap_node *H,int findkey );

void decrease_key(struct rp_heap_node *min,struct rp_heap_node *n1,int newkey);

void cut(struct rp_heap_node *n2,int newkey2);

void rank_rule(struct rp_heap_node * gg);

void delete_node(struct rp_heap_node *delnode);


/* i domi tou kombou */


struct rp_heap_node

```

```

{

    int key;                // to kleidi tou kombou

    int rank;               // to rank()


    struct rp_heap_node *parent;    // deiktis deixnei ston patera tou kombou

    struct rp_heap_node *left;      // deiktis deixnei aritera

    struct rp_heap_node *right;     // deiktis deixnei deksia

    struct rp_heap_node *child;     // deiktis deixnei to paidi tou kombou


};

struct rp_heap_node *minNode,*min1Node;


int nofItems=0;    // o arithmos ton kombon pou uparxoun sinolika

int nofTrees=0;    // o arithmos ton dendron pou uparxoun sto soro


int main ()

{

    int option,minimum,n,m;

    int a0;

    int s0,s1;

    int i0,i1,i2,i3,i4,i5,i6,i7,i8;

    int k0,k1,k2,k3;

    int delkey,changekey,newkey,deletekey;

    char ch;

    struct rp_heap_node *eisagogi_kombou,*nodefind,*nodefind2;

    eisagogi_kombou = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));

```

```

eisagogi_kombou->rank=0;

eisagogi_kombou->key = NIL;

eisagogi_kombou->parent = NIL;

eisagogi_kombou->left =NIL;

eisagogi_kombou->right =NIL;

eisagogi_kombou->child =NIL;


minNode = NIL;  // thetoume arxika tin mikroteri riza tou sorou isi me miden


create_heap();

while(1)

{

    printf("\n*****\n");

    printf("      MENOY\n");

    printf("1:DIMIOYRGIA SOROY\n");

    printf("2:EISAGOGI RIZAS STO SORO\n");

    printf("3:EISAGOGI LEFT TREE DENDROU STO SORO\n");

    printf("4:EURESI MIKROTERIS RIZAS STO SORO \n");

    printf("5:DIAGRAFI MIKROTEROU STOIXEIOY\n");

    printf("6:MEIOSI KLEIDIOY SE STOIXEIO(KOMBOS)\n");

    printf("7:DIAGRAFI OPOIOUDIPOTE STOIXEIOU-KOMBOY\n");

    printf("8:DEITE TO SORO\n");

    printf("9:EKSODOS ");

    printf("\n*****\n");

    scanf ("%d",&option);

    switch(option)

```

```

{
    case 1 :
        create_heap();

        break;

    case 2:

        printf("\nEISAGETAI KLEIDI RIZAS(KOMBOY)= ");

        scanf("%d",&n);

        insert(eisagogi_kombou,n);

        break;

    case 3:

        printf("\nTI RANK THELETE NA EXEI TO LEFT TREE POU THA \nEISAGOUME STO
        SORO(EISAGETE APO 0 EOS 3)\nrank=");

        fflush(stdin);

        scanf("%d",&m);

        switch(m)
        {
            case 0:

                printf("\nEISAGETE TIN RIZA TOY DENDROY=");

                scanf("%d",&a0);

                insert(eisagogi_kombou,a0);

                break;

            case 1:

                printf("\nEISAGETE TIN RIZA TOY DENDROY=");

                scanf("%d",&s0);

                printf("\nEISAGETE TO ARISTERO PAIDI TIS RIZAS=");

                fflush(stdin);

                scanf("%d",&s1);

                if(i2<i1)

```



```

    {

        printf("\nTO PAIDI PREPEI NA EXEI MEGALITERO KLEIDI APO TIN RIZA\n");

        printf("PROSPATHISTE APO TIN ARXI\n\n");

    }

    else

    {

        create_left_tree(s0,s1);

    }

    break;

case 2:

    printf("\nEISAGETE TIN RIZA TOY DENDROY=");

    scanf("%d",&k0);

    printf("\nEISAGETE TO ARISTERO PAIDI TIS RIZAS=");

    fflush(stdin);

    scanf("%d",&k1);

    printf("\nEISAGETE TO ARISTERO PAIDI TOU PROIGOUMENOU KOMBOU=");

    fflush(stdin);

    scanf("%d",&k2);

    printf("\nEISAGETE TO DEKSIO PAIDI TOU PROIGOUMENOU KOMBOU=");

    fflush(stdin);

    scanf("%d",&k3);

    printf("\n\n");

    if(k1<k0 || k2<k1)

    {

        printf("\nTO PAIDI PREPEI NA EXEI MEGALITERO KLEIDI APO TIN RIZA\n");

        printf("\nH TO ARISTERO PAIDI PREPEI NA EXEI MEGALITERO KLEIDI APO

TON PATERA\n");

        printf("PROSPATHISTE APO TIN ARXI\n\n");

```

```

    }

    else

    {

        create_left_tree(k0,k1,k2,k3);

    }

    break;

case 3:

    printf("\nEISAGETE TIN RIZA TOY DENDROY=");

    scanf("%d",&i0);

    printf("\nEISAGETE TO ARISTERO PAIDI TIS RIZAS=");

    fflush(stdin);

    scanf("%d",&i1);

    printf("\nEISAGETE TO ARISTERO PAIDI TOU PROIGOUMENOU KOMBOU=");

    fflush(stdin);

    scanf("%d",&i2);

    printf("\nEISAGETE TO DEKSIO PAIDI TOU PROIGOUMENOU KOMBOU=");

    fflush(stdin);

    scanf("%d",&i3);

    printf("\nEISAGETE TO ARISTERO PAIDI TOU ARISTEROU PROIGOUMENOU
KOMBOU=");

    fflush(stdin);

    scanf("%d",&i4);

    printf("\nEISAGETE TO DEKSIO PAIDI TOU ARISTEROU PROIGOUMENOU
KOMBOU=");

    fflush(stdin);

    scanf("%d",&i5);

    printf("\nEISAGETE TO ARISTERO PAIDI TOU DEKSIOU PROIGOUMENOU
KOMBOU=");

```

```

        fflush(stdin);

        scanf("%d",&i6);

        printf("\nEISAGETE TO DEKSIO PAIDI TOU DEKSIOU PROIGOUMENOU
KOMBOU=");

        fflush(stdin);

        scanf("%d",&i7);

        if(i1<i0 || i2<i1)

        {

            printf("\nTO PAIDI PREPEI NA EXEI MEGALITERO KLEIDI APO TIN RIZA\n");

            printf("\nH TO ARISTERO PAIDI PREPEI NA EXEI MEGALITERO KLEIDI APO
TON PATERA\n");

            printf("PROSPATHISTE APO TIN ARXI\n\n");

        }

        else

        {

            create_left_tree2(i0,i1,i2,i3,i4,i5,i6,i7);

        }

        break;

        default: printf("\nLATHOS EPILOGI...KSANAPROSPATHISTE \n ");

    }

    break;

case 4:

    findmin();

    break;

case 5:

    minimum=deletemin();

    printf("\n\nDIAGRAFIKE I MIKROTERI RIZA ME KLEIDI:%d\n\n",minimum);

    break;

```

```

case 6:

    printf("\nTHELETE NA MEIOSETE TO KLEIDI TOY KOMBOY:");

    scanf("%d",&changekey);

    fflush(stdin);

    printf("\nKAI NA TOY DOSETE NEO KLEIDI:");

    scanf("%d",&newkey);

    nodefind = find_key(minNode,changekey);

    decrease_key(minNode,nodefind,newkey);

    break;

case 7:

    printf("\nTHELETE NA DIAGRAPSETE TO KLEIDI TOY KOMBOY:");

    scanf("%d",&deletekey);

    fflush(stdin);

    nodefind2 = find_key(minNode,deletekey);

    delete_node(nodefind2);

    break;

case 8:

    display_heap(min1Node);

    break;

case 9:

    exit(1);

    break;

default: printf("\nLATHOS EPILOGI...KSANAPROSPATHISTE \n ");

}

}

return 1;

```

```
}
```

```
void delete_node(struct rp_heap_node *delnode)
```

```
{
```

```
    decrease_key(minNode,delnode,PLIN_APEIRO);
```

```
    deletemin();
```

```
    nofTrees++;
```

```
}
```

```
int deletemin()
```

```
{
```

```
    if(nofItems==0)
```

```
    {
```

```
        minNode=NULL;
```

```
        return NULL;
```

```
    }
```

```
int delMinRoot;
```

```
delMinRoot = minNode->key;
```

```
struct rp_heap_node *delForest[20],*addition_array[20],*teliko_array[20] ;
```

```
struct rp_heap_node *cursor,*dummy,*dummy2,*dummy3,*dummy4,*dummy5;
```

```
for(int i=0 ; i <= 20 ; i++)
```

```

{
    addition_array[i] = NIL;

    delForest[i]=NIL;

}

int mm; //ti rank exei to dendro pou tha aferethei

cursor = minNode;

mm=minNode->rank;

dummy2 = minNode->child;

dummy = minNode->child;


int counter=1;

int i,j,rest,test;


while(dummy2!= NIL && dummy2->child != NIL)
{
    counter++;

    dummy2 = dummy2->child->right ;    // metrame tin deksia diadromi tou
dendrou(prosoxi oxi tou sorou)

}

printf("MERTISAME OTI TO DENDRO TIS RIZAS EXEI DEKSIO VATHOS
counter=%d\n\n",counter);


//vgazoume tin mikroteri riza apo to soro

```

```

minNode->left->right=minNode->right;

minNode->right->left=minNode->left;

minNode = cursor->right; // epomeno minNode o deksios aderfos tis rizas-diladi i epomeni
riza tou sorou

rest=test=0;

//spame to dendro tis mikroteris rizas
int gg = mm-1;
for( i = 0 ; i<= mm-1 ;i++)
{

    delForest[i]= dummy;
    if(dummy->child != NIL)
    {
        dummy = dummy->child->right;
    }

    delForest[i]->right= NIL;
    delForest[i]->left= NIL;
    delForest[i]->parent =NIL;

    rest++;
}

nofTrees-- ;

struct rp_heap_node * paok;

```

```

paok = minNode;

//edo vazoume ta enapomeinanta dendra tou sorou stis keneseis tou pinaka
delForest

for(j=0 ; j<nofTrees ; j++)
{

    delForest[rest] = paok; //ston pinaka delForest ksekiname apo tin thesi rest

    if(paok->right != paok)
    {
        paok = paok->right;
    }

    delForest[rest]->parent=NULL;
    delForest[rest]->right=NULL;
    delForest[rest]->left=NULL;

    test++;
    rest++;
}

rest=rest-1;

int thesi=0;

//edo pername ta dendra tou delforest ston pinaka

//addition array

//os eksis: analoga to rank ston delforest ton vazoume stin antistoixei thesi tou addition
array

//an ston addition array exoume 2 dendra idiou rank kaloume sinartisi i opoia

```



```

//kanei tin prosthesi ton duo dendron kai epistrefei neo dendro
for(int i=0;i < rest;i++)
{
    thesi = delForest[i]->rank;

    if(addition_array[thesi] == NIL)
    {
        addition_array[thesi] = delForest[i];
        delForest[i]=NIL;
    }
    else
    {
        dummy3 = addtree(addition_array[thesi],delForest[i]);
        if(addition_array[thesi+1]==NIL)
        {
            addition_array[thesi+1]=dummy3;
            addition_array[thesi]= NIL;
            delForest[i]=NIL;
        }
        else
        {
            dummy4=addtree(dummy3,addition_array[thesi+1]);
            if(addition_array[thesi+2]==NIL)
            {
                addition_array[thesi+2]=dummy4;
            }
        }
    }
}

```

```

        addition_array[thesi+1]= NIL;

        addition_array[thesi]= NIL;
    }
    else
    {
        dummy5=addtree(dummy4,addition_array[thesi+2]);

        if(addition_array[thesi+3]==NIL)
        {
            printf("\nMPIKAME POLI MESA\n");

        }
    }
}

}

}

}

}

}

//dimiuourgoume keno soro , kai kanoume eisagogi(meso for) osa dendra emeinan
//ston pinaka addition_array

create_heap();

for(int kk=0;kk<20;kk++)
{

    if(addition_array[kk]!=NIL)
    {
        insert(addition_array[kk],addition_array[kk]->key);
    }
}

```

```

    }

    return delMinRoot;

}

struct rp_heap_node* addtree(struct rp_heap_node *a, struct rp_heap_node *b)
{
    struct rp_heap_node *neodendro;

    neodendro = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));

    int neo_rank;

    neo_rank = (1+ a->rank);

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

        neodendro->key = a->key;

        neodendro->child = b;

        if(neo_rank >1)
        {

            neodendro->child->child->right = a->child;

        }

        neodendro->rank = neo_rank;
    }
}

```

```

    }

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

        neodendro->key = b->key;

        neodendro->child = a;

        if(neo_rank>1)
        {
            neodendro->child->child->right = b->child;
        }

        neodendro->rank = neo_rank;
    }

    else if(a->key == b->key)
    {

    }

    return neodendro;
}

void create_heap()
{
    struct rp_heap_node *minHelp;

    minHelp = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));

    minHelp->parent=NULL;

    minHelp->left=NULL;

```

```

minHelp->right=NULL;

minHelp->child=NULL;


minHelp->key=NULL;

minHelp->rank=NULL;


nofItems=0;

nofTrees=0;


minNode = minHelp;


printf("\nDIMIOYRGISAME ENAN KENO SORO\n");
}


void insert(struct rp_heap_node * tree_root, int val)
{

    struct rp_heap_node *insert_heap;

    insert_heap = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));

    insert_heap->key  = val;

    insert_heap->parent = tree_root->parent ;

    insert_heap->left  = tree_root->left;

    insert_heap->right = tree_root->right;

    insert_heap->child = tree_root->child;

    insert_heap->rank  = tree_root->rank;

```

```

if(minNode->key != NIL)    //exoume idi riza sto soro
{

    printf("EISAGOGI RIZAS SE SORO POU EXEI IDI RIZES,\n");

    if(val<=minNode->key)    // i nea riza tha ginei i riza me to mikrotero kleidi
    {
        printf("KAI I NEA RIZA EINAI I MIKROTERI RIZA TOU SOROU\n\n");
        insert_heap->left = minNode->left;
        insert_heap->left->right= insert_heap;
        insert_heap->right = minNode;
        insert_heap->right->left = insert_heap;
        minNode=insert_heap;
    }
    else
    {
        insert_heap->right = minNode->right;
        insert_heap->left = minNode->right->left;
        minNode->right = insert_heap;
        insert_heap->right->left = insert_heap;
        printf("KAI I NEA RIZA POY EISAGOUME\nDEN EINAI I MIKROTERI\n");
    }
}
else
{

```

```

minNode=insert_heap;

minNode->left=minNode;

minNode->right=minNode;

printf("O PROTOS ,ARA KAI O MIKROTEROS KOMBOS TOU SOROU\n");

printf("ME KLEIDI:%d\n",minNode->key);

}

nofItems++;

nofTrees++;

}

```

```

void display_heap(struct rp_heap_node *min1)
{
    struct rp_heap_node *q,*u,*chil;

    if(minNode == NIL)
    {
        printf("\n O SOROS EINAI ADEIOS");

        return;
    }

    else
    {

        q = minNode;

        printf("\n O SOROS RP-HEAPS EXEI OS EKSIS:\n\n");

        do

```

```

{
    printf("|%d(%d)|",q->key,q->rank);

    if(q->child != NIL)
    {
        printf("|%d(%d)|",q->child->key,q->child->rank);
        if(q->child->child != NIL)
        {
            printf("|%d(%d)|",q->child->child->key,q->child->child->rank);
            if(q->child->child->right != NIL)
            {
                printf("|%d(%d)|",q->child->child->right->key,q->child->child->right->rank);
            }
            if(q->child->child->child != NIL)
            {
                printf("|%d(%d)|",q->child->child->child->key,q->child->child->child->rank);
                if(q->child->child->child->right != NIL)
                {
                    printf("|%d(%d)|",q->child->child->child->right->key,q->child->child->child->right->rank);
                }

                if(q->child->child->right->child != NIL)
                {
                    printf("|%d(%d)|",q->child->child->right->child->key,q->child->child->right->child->rank);
                    if(q->child->child->right->child->right != NIL)

```



```

        {
            printf("|%d(%d)|",q->child->child->right->child->right->key,q->child->child-
>right->child->right->rank);

        }

    }

}

}

}

else

{

}

if(q->right!=NIL)

{

    printf("-->");

    q=q->right;

}

} while(q!= minNode);

printf("\n");

}

}

void findmin()

{

    printf("\nTO MIKROTERO KLEIDI RIZAS EINAI TO %d: ",minNode->key);

    printf("\n");

```

```
}
```

```
void decrease_key(struct rp_heap_node *min,struct rp_heap_node *n1,int newkey)
```

```
{
```

```
    struct rp_heap_node *n2,*n3,*dummy1,*dummy2,*neos_aderfos ;
```

```
    dummy1 = min;
```

```
    int true_false=1;
```

```
    int true_false2=1;
```

```
    int rank_diff1 ;
```

```
    rank_diff1 = n1->parent->rank - n1->rank;
```

```
    int rank_diff2;
```

```
    if(n1->child!=NIL && n1->child->right!=NIL)
```

```
    {
```

```
        rank_diff2 = n1->rank - n1->child->right->rank;
```

```
    }
```

```
    else
```

```
    {
```

```
        rank_diff2=0;
```

```
    }
```

```
    int rank_diff = rank_diff1+rank_diff2;
```

```
    n2=n1;
```

```
    n3=n1->parent;
```

```
    if(n1->parent->child->key == n1->key)
```

```
    {
```

```

if(n1->child != NIL)
{

    neos_aderfos=n1->right;
    n1->child->right->parent=n1->parent;
    n1=n1->child->right;
    n1->parent->child =n1;
    n1->right = neos_aderfos ;

    rank_rule(n1->parent);
    while(n1->parent != NIL)
    {
        n1=n1->parent;
        rank_rule(n1);

        while(true_false && n1->parent == NIL)
        {
            if(dummy1->right->key == n1->key)
            {
                true_false=0;
            }
            dummy1=dummy1->right;
        }
    }

    insert (n1,n1->key);

```

```

dummy1->left->right = dummy1->right;

dummy1->right->left = dummy1->left;

}

else

{

    // auto to else otan kanoume meiosi kleidiou se leaf

    neos_aderfos=n1->right;

    n1=neos_aderfos;

    n1->parent->child =n1;

    n1->right = NIL;

    //rank_rule(n1->parent);

    while(n1->parent != NIL)

    {

        n1=n1->parent;

        //rank_rule(n1);

        while( true_false2 && n1->parent == NIL)

        {

            if(dummy1->right->key == n1->key)

            {

                true_false2=0;

            }

            dummy1=dummy1->right;

```

```

    }
}

insert (n1,n1->key);

dummy1->left->right = dummy1->right;
dummy1->right->left = dummy1->left;
}

}
else
{
    if(n1->child != NIL) //oxi leaf
    {
        n1->parent->child->right = n1->child->right;
        rank_rule(n1->parent);
        while(n1->parent != NIL)
        {
            n1=n1->parent;
            rank_rule(n1);

            while(true_false && n1->parent == NIL)
            {
                if(dummy1->right->key == n1->key)
                {

```

```

        true_false=0;
    }
    dummy1=dummy1->right;
}
}

insert (n1,n1->key);

dummy1->left->right = dummy1->right;
dummy1->right->left = dummy1->left;
}
else
{
    neos_aderfos=n1->right;
    n1=neos_aderfos;
    n1->parent->child->right =n1;
    n1->right = NIL;

    while(n1->parent != NIL)
    {
        n1=n1->parent;

        while( true_false2 && n1->parent == NIL)
        {

```

```

        if(dummy1->right->key == n1->key)
        {
            true_false2=0;
        }
        dummy1=dummy1->right;
    }
}

insert (n1,n1->key);

dummy1->left->right = dummy1->right;
dummy1->right->left = dummy1->left;
}
}

cut(n2,newkey);
}

void rank_rule(struct rp_heap_node * gg)
{

    int rank_difference_left, rank_difference_right,compare,reduceaboveone;

    if(gg->parent == NULL) // ean vriskomaste stin riza
    {

```

```

gg->rank = gg->child->rank+1;

}

else

{

    int rankdiff_left = gg->rank - gg->child->rank;

    int rankdiff_right = gg->rank - gg->child->right->rank;

    if(rankdiff_left == rankdiff_right && rankdiff_left!=1)

    {

        int reducetoone=0;

        while(rankdiff_left >1)

        {

            rankdiff_left--;

            reducetoone++;

        }

        gg->rank = gg->rank - reducetoone;

    }

    if(rankdiff_left != rankdiff_right )

    {

        if(rankdiff_left > rankdiff_right)

        {

            reduceaboveone = rankdiff_right;

            gg->rank = gg->rank - reduceaboveone;

        }

        if(rankdiff_left < rankdiff_right)

        {

```



```

        reduceaboveone = rankdiff_left;

        gg->rank = gg->rank - reduceaboveone;
    }
}

}
}

```

```

void cut(struct rp_heap_node *n2,int newkey2)
{
    n2->key = newkey2;
    n2->parent =NIL;
    if(n2->child != NIL)
    {
        n2->child->right = NIL;
    }
    n2->right=NIL;
    n2->left = NIL;
    insert(n2,n2->key);
}

```

```

struct rp_heap_node *find_key(struct rp_heap_node *H,int findkey)
{

    struct rp_heap_node *x = H ;

```

```

struct rp_heap_node * p=NULL;

if(x==NULL)return NULL;

do {

        if(x->key==findkey)

        {

                return x;

        }

        p = find_key(x->child,findkey);

        if(p)

        {

                return p;

        }

        if(x->right!=NULL)

        {

                x=x->right;

        }

        }while(x!=H );

return NULL;

}

```

```

void create_left_tree(int k0,int k1,int k2,int k3)
{
    struct rp_heap_node *tree_root, *tree_node1, *tree_leaf1, *tree_leaf2;
    tree_root = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_node1 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_leaf1 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_leaf2 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));

    tree_root->key = k0;
    tree_root->rank = 2;
    tree_root->parent = NIL;
    tree_root->left=NIL;
    tree_root->right=NIL;
    tree_root->child = tree_node1;

    tree_node1->key = k1;
    tree_node1->rank = 1;
    tree_node1->parent = tree_root;
    tree_node1->left = NIL;
    tree_node1->right = NIL;
    tree_node1->child = tree_leaf1;

    tree_leaf1->key = k2;
    tree_leaf1->rank = 0;
    tree_leaf1->parent = tree_node1;
    tree_leaf1->left = tree_leaf2;

```

```

tree_leaf1->right = tree_leaf2;

tree_leaf1->child = NIL;


tree_leaf2->key = k3;

tree_leaf2->rank = 0;

tree_leaf2->parent = tree_node1;

tree_leaf2->left = tree_leaf1;

tree_leaf2->right = tree_leaf1;

tree_leaf2->child = NIL;


insert(tree_root,k0);

nofItems = nofItems + 3 ; // epeidi vazoume 3 kombous pou den fenontai sto soro
}

```

```

void create_left_tree2(int i0,int i1,int i2,int i3,int i4, int i5, int i6 , int i7)
{

    struct rp_heap_node *tree_root, *tree_node1,*tree_node2,*tree_node3,
        *tree_leaf1, *tree_leaf2,*tree_leaf3,*tree_leaf4;


    tree_root = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_node1 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_node2 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_node3 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_leaf1 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_leaf2 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));

```

```
tree_leaf3 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));  
tree_leaf4 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
```

```
tree_root->key = i0;  
tree_root->rank = 3;  
tree_root->parent = NIL;  
tree_root->left=NIL;  
tree_root->right=NIL;  
tree_root->child = tree_node1;
```

```
tree_node1->key = i1;  
tree_node1->rank = 2;  
tree_node1->parent = tree_root;  
tree_node1->left = NIL;  
tree_node1->right = NIL;  
tree_node1->child = tree_node2;
```

```
tree_node2->key = i2;  
tree_node2->rank = 1;  
tree_node2->parent = tree_node1;  
tree_node2->left = tree_node3;  
tree_node2->right = tree_node3;  
tree_node2->child = tree_leaf1;
```

```
tree_node3->key = i3;  
tree_node3->rank = 1;  
tree_node3->parent = tree_node1;
```

```
tree_node3->left = tree_node2;  
tree_node3->right = tree_node2;  
tree_node3->child = tree_leaf3;
```

```
tree_leaf1->key = i4;  
tree_leaf1->rank = 0;  
tree_leaf1->parent = tree_node2;  
tree_leaf1->left = tree_leaf2;  
tree_leaf1->right = tree_leaf2;  
tree_leaf1->child = NIL;
```

```
tree_leaf2->key = i5;  
tree_leaf2->rank = 0;  
tree_leaf2->parent = tree_node2;  
tree_leaf2->left = tree_leaf1;  
tree_leaf2->right = tree_leaf1;  
tree_leaf2->child = NIL;
```

```
tree_leaf3->key = i6;  
tree_leaf3->rank = 0;  
tree_leaf3->parent = tree_node3;  
tree_leaf3->left = tree_leaf4;  
tree_leaf3->right = tree_leaf4;  
tree_leaf3->child = NIL;
```

```
tree_leaf4->key = i7;  
tree_leaf4->rank = 0;
```

```

tree_leaf4->parent = tree_node3;

tree_leaf4->left = tree_leaf3;

tree_leaf4->right = tree_leaf3;

tree_leaf4->child = NIL;


insert(tree_root,i0);

nofItems = nofItems + 7 ; // epeidi vazoume 7 kombous pou den fenontai sto soro
}

```

```

void create_left_tree(int s0, int s1)
{
    struct rp_heap_node *tree_root, *tree_node1;


    tree_root = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));
    tree_node1 = (struct rp_heap_node *)malloc(sizeof(struct rp_heap_node));


    tree_root->key = s0;

    tree_root->rank = 1;

    tree_root->parent = NIL;

    tree_root->left=NIL;

    tree_root->right=NIL;

    tree_root->child = tree_node1;


    tree_node1->key = s1;

    tree_node1->rank = 0;

    tree_node1->parent = tree_root;

    tree_node1->left = NIL;

```

```
tree_node1->right = NIL;

tree_node1->child = NIL;


insert(tree_root,s0);

nofItems = nofItems + 1 ; // epeidi vazoume 1 kombo pou den fenontai sto soro


}
```