ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΑΝΑΛΥΣΗ ΑΛΓΟΡΙΘΜΩΝ

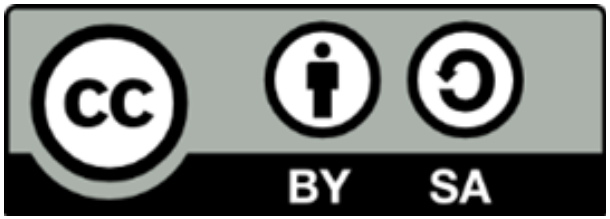
Ενότητα 10α: Αλγόριθμοι Σωρών- Σωρός Μεγίστων- Ταξινόμηση με Σωρό- Σωρός Ελαχίστων Μεγίστων- Διπλός Σωρός

Μαρία Σατρατζέμη
Τμήμα Εφαρμοσμένης Πληροφορικής

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.



Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Μακεδονίας» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ουρά Προτεραιότητας

Η **ουρά προτεραιότητας** (*priolity queue*) είναι μία σημαντική αφηρημένη δομή δεδομένων, που διακρίνεται για τις εξής δύο βασικές λειτουργίες

- ✓ εισαγωγή στοιχείου με τυχούσα προτεραιότητα, και
- ✓ εξαγωγή στοιχείου με τη μεγαλύτερη (ή τη μικρότερη) προτεραιότητα.

Οι ουρές προτεραιότητας είναι πολύ σημαντικές κατά τη δημιουργία προσομοιώσεων γιατί περιγράφουν πολλά φαινόμενα της καθημερινής ζωής.

Ο **σωρός** (*heap*) είναι μία δενδρική δομή για την υλοποίηση ουρών προτεραιότητας. Με άλλα λόγια, αν και ο σωρός είναι μία δενδρική δομή δεν έχει καμία σχέση με τη λειτουργία της Δυαδικού Δένδρου Αναζήτησης.

Στην έννοια του σωρού συμπεριλαμβάνεται μία μεγάλη οικογένεια δομών και αντίστοιχων αλγορίθμων. Οι δομές αυτές ανήκουν σε δύο βασικές κατηγορίες: τους σωρούς που υλοποιούνται με στατικές δομές (δηλαδή με πίνακες), και τους σωρούς που υλοποιούνται με δυναμικές δομές (δηλαδή με δείκτες). Στις επόμενες παραγράφους θα εξετάσουμε τη δομή του σωρού μεγίστων, του σωρού ελαχίστων και μεγίστων.

Δένδρο Μεγίστων - Ελαχίστων

Πριν ορισθεί η έννοια του **σωρού μεγίστων** (*max heap*) είναι απαραίτητο να δοθεί ο εξής ορισμός.

Ορισμός.

Δένδρο μεγίστων (*max tree*) είναι το δένδρο, όπου η τιμή του κλειδιού κάθε κόμβου δεν είναι μικρότερη από τις τιμές των κλειδιών των δύο παιδιών.

Αντίστοιχα, **δένδρο ελαχίστων** (*min tree*) είναι το δένδρο, όπου η τιμή του κλειδιού κάθε κόμβου δεν είναι μεγαλύτερη από τις τιμές των κλειδιών των δύο παιδιών.

Σχεδόν πλήρες δυαδικό δένδρο : συμπληρωμένα πλήρως όλα τα επίπεδα εκτός ενδεχόμενα του τελευταίου επιπέδου. Το τελευταίο επίπεδο συμπληρώνεται από αριστερά προς τα δεξιά.

Σωρός Μεγίστων - Ελαχίστων

Ορισμός.

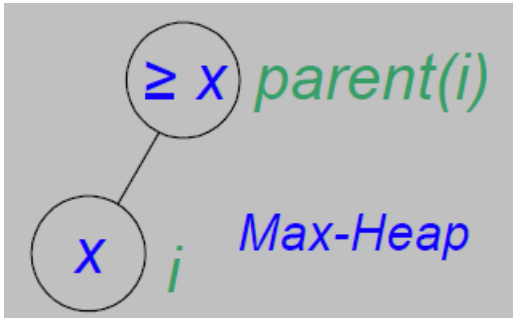
Σωρός μεγίστων *(max Heap*) το σχεδόν πλήρες δυαδικό δένδρο που είναι επίσης δένδρο μεγίστων.

Σωρός ελαχίστων (*min Heap*) είναι το σχεδόν πλήρες δυαδικό δένδρο που είναι επίσης δένδρο ελαχίστων.

Η δομή του σωρού μεγίστων συναντάται και με την ονομασία **φθίνων σωρός** (*descending heap*) ή ακόμη και ως **μερικά. *διατεταγμένο φθίνον δένδρο*** *(descending* partially ordered tree).

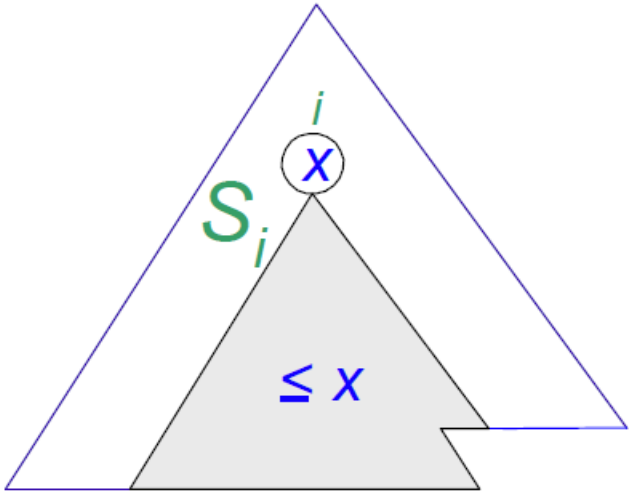
Επίσης ο σωρός ελαχίστων συναντάται και ως **αύξων σωρός** (*ascending heap*) ή **μερικά. διατεταγμένο αύξον δένδρο** (*ascending partially ordered tree*).

Ιδιότητα σωρού



- Για κάθε κόμβο *i* εκτός της ρίζας ισχύει:
  – Μέγιστο σωρό: A[parent(*i*)] ≥ A[*i*]
  – Ελάχιστο σωρό: A[parent(*i*)] ≤ A[*i*]

  Οπου το A[*i*] συμβολίζει το στοιχείο που είναι αποθηκευμένο στον κόμβο *i*
- Το μεγαλύτερο στοιχείο ενός υποδένδρου είναι η ρίζα του υποδένδρου



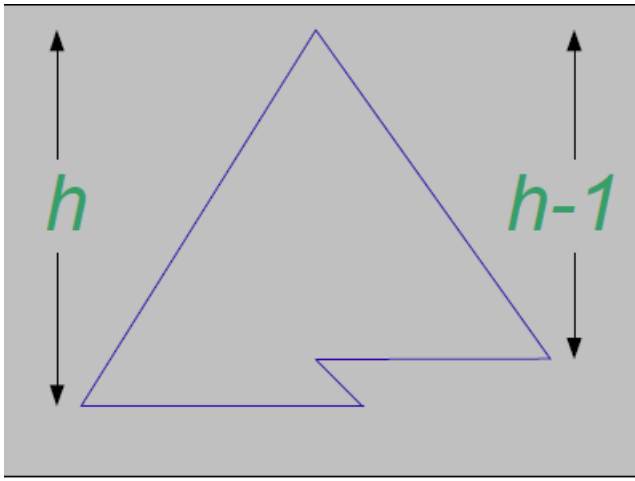
Σωρός ως Δομή δεδομένων

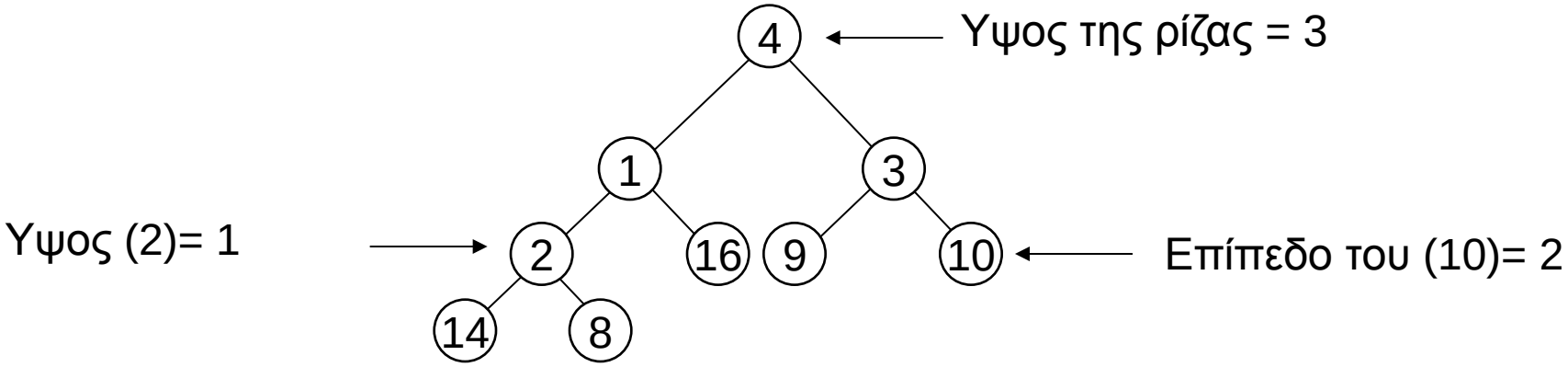
- Αποθήκευση του σωρού σε πίνακα με έμμεσους δείκτες
  - Αριστερό παιδί: left(i) = 2i
  - Δεξί παιδί: right(i) = 2i+1
- Πατέρας του i είναι: parent(i) = $\lfloor i/2 \rfloor$
- A[1]: στοιχείο αποθηκευμένο στη ρίζα
- Ο πίνακας έχει 2 χαρακτηριστικά
  - μήκος[A]: αριθμός στοιχείων πίνακα A
  - μέγεθος[A] = n: αριθμός στοιχείων του σωρού που έχουν αποθηκευτεί στον A

$$n \leq \text{μήκος[A]}$$

Ύψος κόμβου & σωρού

- Ορίζουμε **ύψος κόμβου του σωρού** ως το πλήθος των ακμών στη μακρύτερη απλή καθοδική διαδρομή από τον κόμβο μέχρι κάποιον τερματικό κόμβο (φύλλο).
- Ορίζουμε **ύψος του σωρού** ως το πλήθος των ακμών στη μακρύτερη απλή καθοδική διαδρομή από τη ρίζα προς έναν τερματικό κόμβο.





Χρήσιμες ιδιότητες -1-

Πρόταση 1.
*Σε ένα δυαδικό δένδρο στο επίπεδο d υπάρχουν **το πολύ** 2^d κόμβοι*
Πρόταση 2.
*Ένα δυαδικό δένδρο ύψους d έχει **το πολύ** $2^{d+1} - 1$ κόμβους*
Απόδειξη:
Το Δυαδικό δένδρο στο επίπεδο 0 έχει 2^0 το πολύ κόμβους
Το Δυαδικό δένδρο στο επίπεδο 1 έχει 2^1 το πολύ κόμβους
Το Δυαδικό δένδρο στο επίπεδο 2 έχει 2^2 το πολύ κόμβους
……
Το Δυαδικό δένδρο στο επίπεδο *d* έχει 2^d το πολύ κόμβους. Αρα οι κόμβοι *n* ενός δυαδικού δένδρου είναι το άθροισμα των κόμβων όλων των επιπέδων:

$$n \le \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$

Πρόταση 3.
Ένα δυαδικό δένδρο με n κόμβους το ύψος του είναι τουλάχιστον $h = \lfloor \log_2 n \rfloor$

Χρήσιμες ιδιότητες -2-

Πρόταση 4.
Σ' ένα πλήρες δυαδικό δένδρο αν m είναι πλήθος των φύλλων τότε m-1 είναι το πλήθος των εσωτερικών κόμβων και 2m-1 είναι το πλήθος όλων των κόμβων
Απόδειξη:
Το πλήρες Δυαδικό δένδρο στο επίπεδο 0 έχει 2^0 κόμβους
Το πλήρες Δυαδικό δένδρο στο επίπεδο 1 έχει 2^1 κόμβους
Το πλήρες Δυαδικό δένδρο στο επίπεδο 2 έχει 2^2 κόμβους
.....
Το πλήρες Δυαδικό δένδρο στο επίπεδο *d-1* έχει 2^{d-1} κόμβους.
Το πλήρες Δυαδικό δένδρο στο επίπεδο *d* έχει $\mathbf{2^d = m}$ κόμβους –φύλλα (1)
Αρα οι **εσωτερικοί κόμβοι** ενός πλήρους δυαδικού δένδρου ύψους d είναι το άθροισμα των κόμβων των επιπέδων 0 έως και *d-1*

$$\sum_{i=0}^{d-1} 2^i = \frac{2^{d-1+1}-1}{2-1} = 2^d - 1 = m - 1 \qquad (2)$$

Αρα το πλήθος των όλων των κόμβων = πλήθος φύλλων + πλήθος εσωτερικών κόμβων = (από τις 1 & 2) = $m + (m - 1) = 2m - 1$

Χρήσιμες ιδιότητες -3-

Πρόταση 5.

Σε ένα σωρό με n κόμβους και ύψος h ισχύει : $2^h \leq n \leq 2^{h+1}-1$

Απόδειξη: Με τον βάση τον ορισμό του σωρού, όλα τα επίπεδα του δέντρου είναι εντελώς συμπληρωμένα εκτός ίσως για το χαμηλότερο επίπεδο, το οποίο είναι συμπληρωμένο από τα αριστερά μέχρι ενός σημείου.

Είναι σαφές ότι ένας σωρός ύψους *h* έχει τον ελάχιστο αριθμό στοιχείων, όταν έχει μόνο έναν κόμβο στο χαμηλότερο επίπεδο. Τα επίπεδα πάνω από το χαμηλότερο επίπεδο σχηματίζουν ένα πλήρες δυαδικό δένδρο ύψους *h* -1 και $2^{(h-1)+1}$ -1 = 2^h-1 κόμβους.

Ο ελάχιστος αριθμός κόμβων σε ένα σωρό ύψους *h* είναι 2^h (προκύπτει: (πλήθος κόμβων σωρού ύψους *h*-1) + 1 κόμβο του χαμηλοτέρου επιπέδου *h* = (2^h-1) +1= 2^h).

Ενας σωρός ύψους *h*, έχει το μέγιστο αριθμό των στοιχείων, όταν το χαμηλότερο επίπεδο είναι τελείως γεμάτο. Στην περίπτωση αυτή, ο σωρός είναι ένα πλήρες δυαδικό δένδρο ύψους *h* και ως εκ τούτου έχει 2^{h+1} -1 κόμβους.

Ετσι για ένα σωρό με *n* κόμβους και ύψος *h* έχουμε: $2^h \leq n \leq 2^{h+1}-1$

Χρήσιμες ιδιότητες -4-

Πρόταση 6: *Το ύψος ενός ΔΔ με n κόμβους είναι $log_2 n$. Αρα το n-στοιχείο ενός σωρού έχει ύψος $\lfloor \log_2 n \rfloor$.*

Απόδειξη

Εστω ότι το *n*-στοιχείο του σωρού έχει ύψος *h*. Από τα όρια που γνωρίζουμε σχετικά με το ελάχιστο και το μέγιστο πλήθος στοιχείων που έχει ένας σωρός έχουμε:

$$2^h \le n \le 2^{h+1}-1 \text{ ισχύει και}$$

$$2^h \le n \le 2^{h+1}-1 < 2^{h+1}$$

Λογαριθμώντας με βάση το 2.

$$h \le \log_2 n < h+1$$

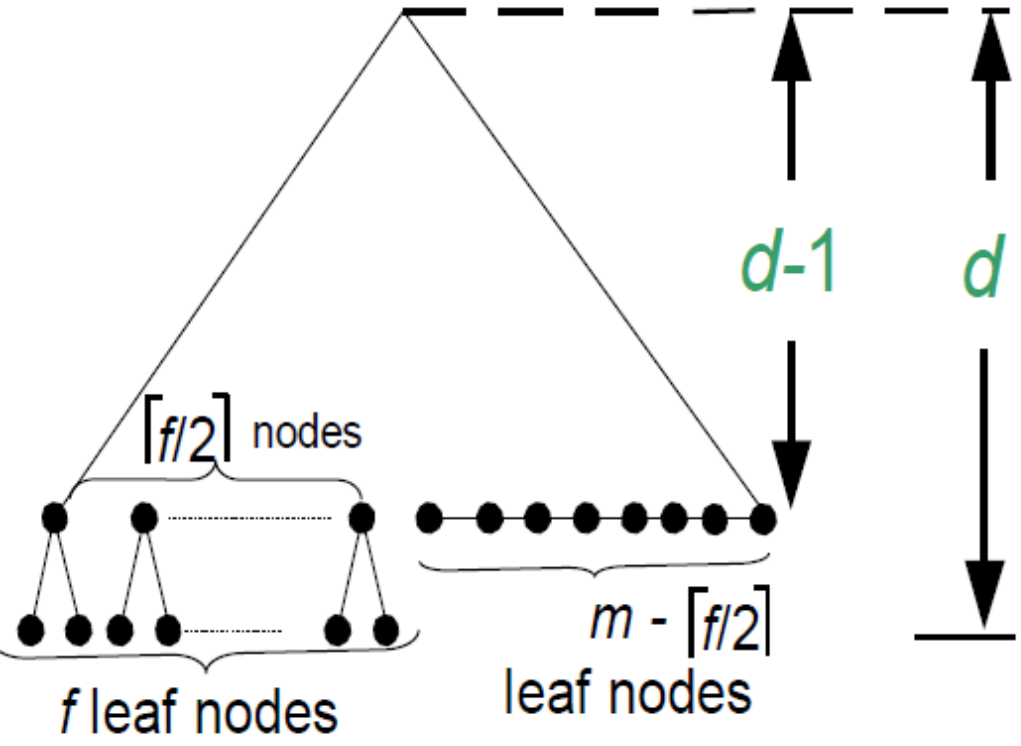
Συνεπάγεται: $h = \lfloor \log_2 n \rfloor$

Χρήσιμες ιδιότητες -5-

Πρόταση 7: *Οι τελευταίοι $\lceil n/2 \rceil$ κόμβοι ενός σωρού είναι όλοι φύλλα.*

Απόδειξη

$m = 2^{d-1}$: # κόμβων στο επίπεδο d – 1
f : # κόμβων στο επίπεδο d (τελευταίο επίπεδο)
#: πλήθος



Χρήσιμες ιδιότητες -6-

Απόδειξη

των φύλλων = $f + (m - \lceil f/2 \rceil) = m + \lfloor f/2 \rfloor$

$m + (m - 1) + f = n$

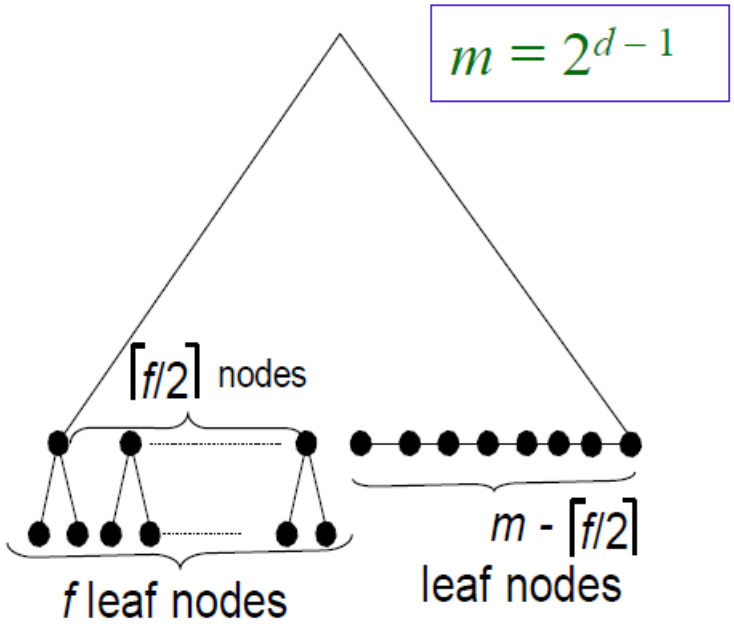
$2m + f = n + 1$

$$\left\lfloor \frac{1}{2}(2m + f) \right\rfloor = \left\lfloor \frac{1}{2}(n+1) \right\rfloor$$

$\lfloor m + f/2 \rfloor = \lceil n/2 \rceil$

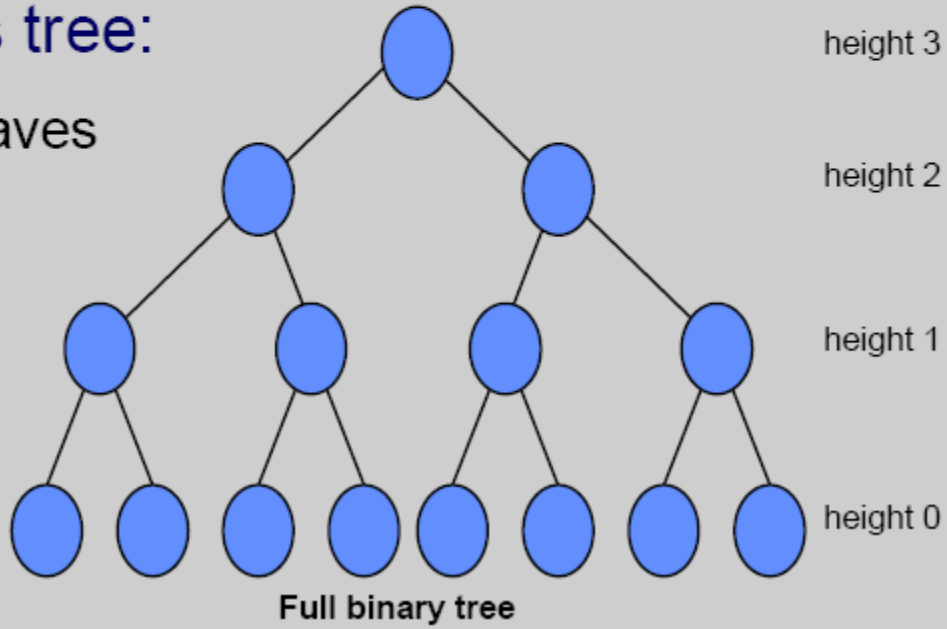
$m + \lfloor f/2 \rfloor = \lceil n/2 \rceil$

- # φύλλων = $\lceil n/2 \rceil$



Χρήσιμες ιδιότητες -7-

- Height of a node = longest distance from a leaf
- For a complete binary tree, there are $\lceil n/2^{h+1} \rceil$ nodes of height h
- Example: $n = 15$ gives tree:
  - $\lceil n/2^{0+1} \rceil = \lceil 15/2 \rceil = 8$ leaves
  - $\lceil n/2^{1+1} \rceil = \lceil 15/4 \rceil =$ 4 parents to leaves
  - ...
  - $\lceil n/2^{3+1} \rceil = \lceil 15/16 \rceil =$ 1 root node



Full binary tree

MRTC

Χρήσιμες ιδιότητες -8-

Πρόταση 8: *Σε έναν σωρό n στοιχείων υπάρχουν το πολύ κόμβοι* $\lceil n/2^{h+1} \rceil$ *ύψους h*

Απόδειξη (*με επαγωγή*)

1. Για h = 0. Ο αριθμός των φύλλων ύψους $h = 0$ είναι ο αριθμός των φύλλων ενός σωρού με n στοιχεία. Ο πατέρας του n-οστού στοιχείου (τελευταίου στοιχείου του σωρού) είναι ο $\lfloor n/2 \rfloor$ κόμβος που είναι και ο τελευταίος πατέρας του σωρού όλοι οι άλλοι κόμβοι μετά απ' αυτόν είναι φύλλα. Αρα ο αριθμός των φύλλων είναι $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil = \lceil n/2^1 \rceil = \lceil n/2^{0+1} \rceil = \lceil n/2^{h+1} \rceil$ άρα η σχέση ισχύει για $h = 0$.

2. Υποθέτουμε ότι ο αριθμός των κόμβων ύψους h-1 δίνεται από τη σχέση: $\lceil n/2^{h-1+1} \rceil = \lceil n/2^h \rceil$

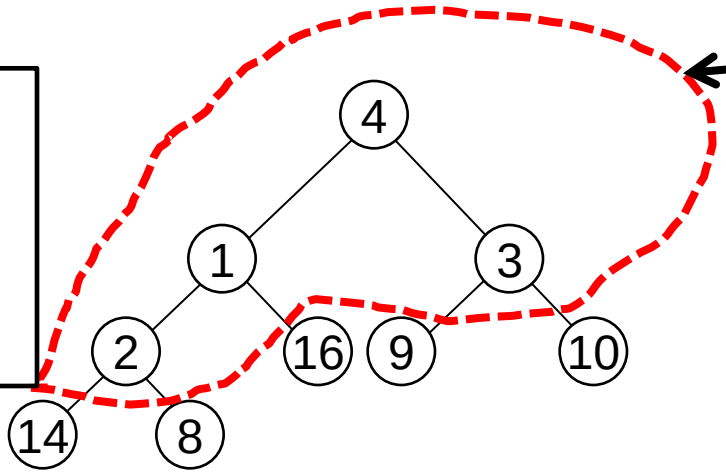
Χρήσιμες ιδιότητες -9-

1 κόμβος ο 4 ύψους 3

1 κόμβος ο 1 ύψους 2

2 κόμβοι οι 2, 3 ύψους 1

5 κόμβοι ύψους 0



Εσωτερικοί κόμβοι

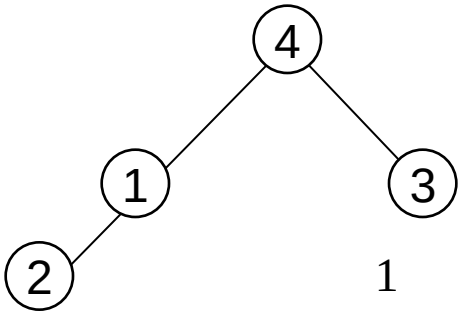
- $n = 9$
- Ο πατέρας του n-οστού στοιχείου (τελευταίου στοιχείου του σωρού) του 8 είναι ο $\lfloor n/2 \rfloor = \lfloor 9/2 \rfloor = 4$ος
- $\lfloor n/2 \rfloor = \lfloor 9/2 \rfloor = 4$ εσωτερικοί κόμβοι
- $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil = 5$ φύλλα

Ο αριθμός των κόμβων ύψους h στο παλιό σωρό είναι ο ίδιος με τον αριθμό των κόμβων ύψους h-1 στο νέο σωρό

1 κόμβος ο 4 ύψους 2

1 κόμβος ο 1 ύψους 1

2 κόμβοι οι 2, 3 ύψους 0



A:

1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8

Χρήσιμες ιδιότητες -10-

3. Θα δείξουμε ότι ισχύει για κόμβους ύψους h.

Αν αφαιρέσουμε όλα τα φύλλα του σωρού τότε οι κόμβοι που είχαν ύψος 1 (στον αρχικό σωρό) τώρα γίνονται φύλλα στο νέο σωρό, δηλαδή έχουν ύψος 0. Αντίστοιχα για όλους τους κόμβους του νέου σωρού το ύψος τους είναι κατά 1 μικρότερο απ' αυτό που είχαν στο παλιό σωρό.

Στο νέο σωρό (αυτόν που προέκυψε μετά την απομάκρυνση των φύλλων του σωρού ύψους *h*) το πλήθος των κόμβων του είναι $n - \lceil n/2 \rceil$.

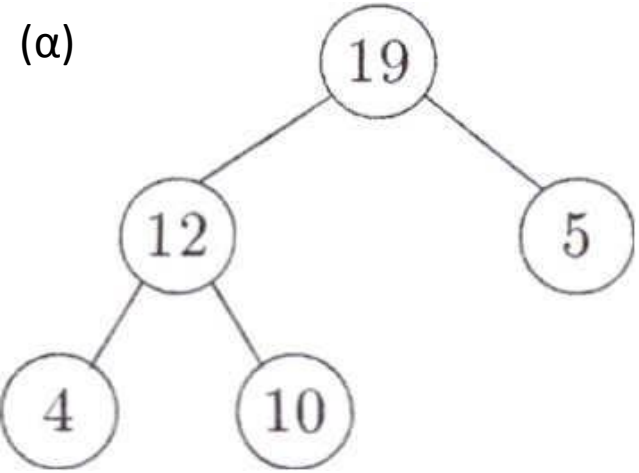
Ο αριθμός των κόμβων ύψους *h* στο παλιό σωρό είναι ο ίδιος με τον αριθμό των κόμβων ύψους *h*-1 στο νέο σωρό.

Αρα με βάση το βήμα 2 ισχύει $\lceil n - \lceil n/2 \rceil / 2^h \rceil = \lceil n/2^{h+1} \rceil$

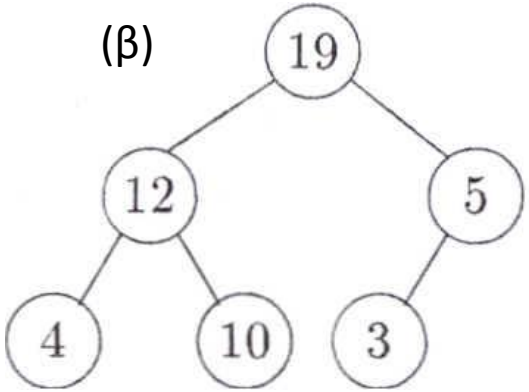
Πρόταση 8.

Σε ένα σωρό μεγίστων η εισαγωγή και η εξαγωγή απαιτούν χρόνο τάξης $O(\log_2 n)$

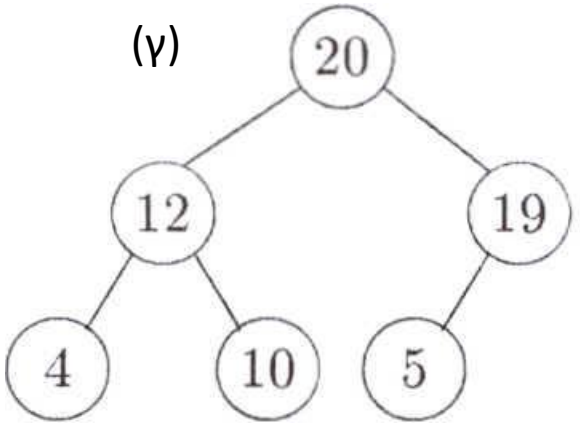
(α)



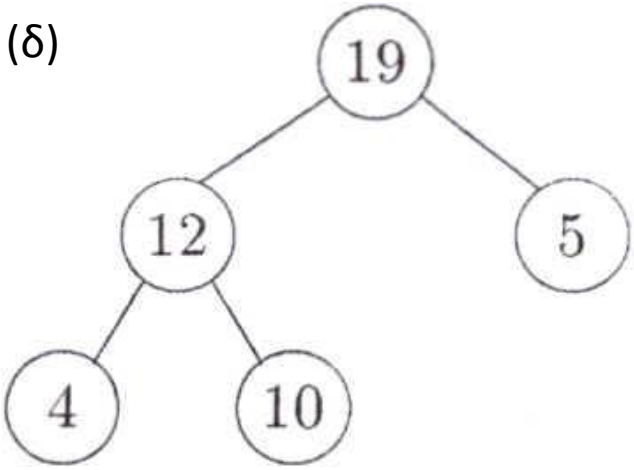
(β)



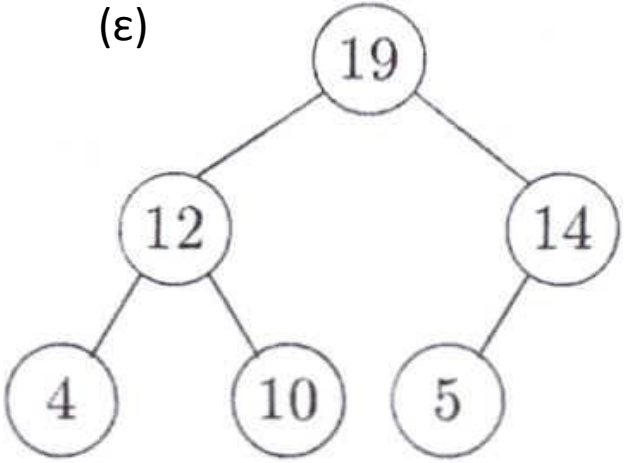
(γ)



(δ)



(ε)

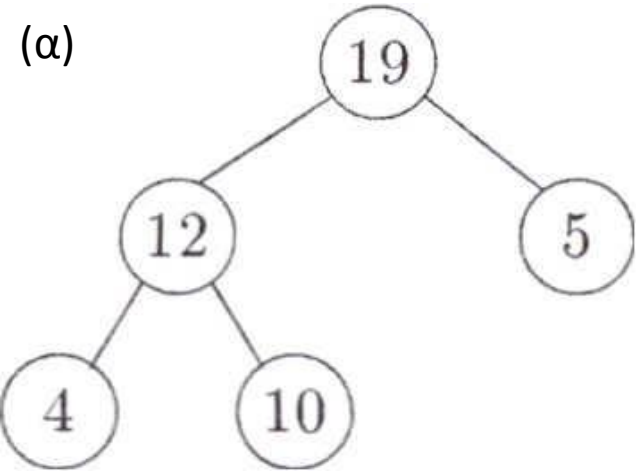


Πρόταση 7.

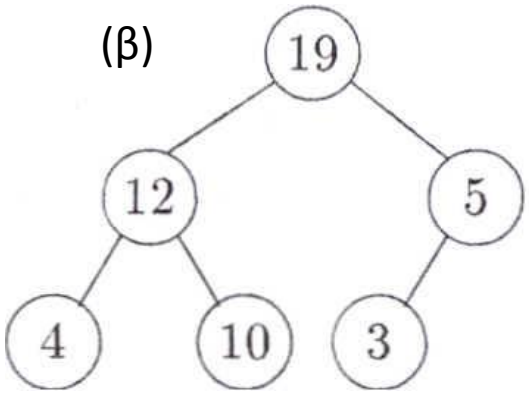
Σε ένα σωρό μεγίστων η εισαγωγή και η εξαγωγή απαιτούν χρόνο τάξης $O(\log_2 n)$

Η αλήθεια της πρότασης θα γίνει κατανοητή με τη βοήθεια ενός παραδείγματος. Έστω ότι στον πρώτο σωρό του σχήματος πρέπει να εισαχθεί ένα νέο στοιχείο. Η θέση του νέου στοιχείου επιβάλλεται να είναι αριστερό παιδί του κόμβου με το κλειδί 5, επειδή το δένδρο πρέπει να είναι σχεδόν πλήρες. Αν η τιμή του νέου κλειδιού είναι μεγαλύτερη του 5, τότε παραβιάζεται η ιδιότητα που θέλει το σωρό να είναι ένα δένδρο μμεγίστων. Έτσι αν η τιμή του νέου κλειδιού είναι 3, 14 ή 20, τότε προκύπτουν τα αντίστοιχα σχήματα.

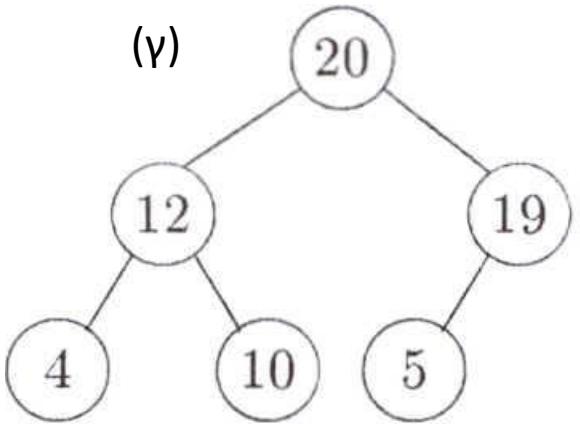
(α)



(β)



(γ)



Συνέχεια…

Εστω ότι από τον προτελευταίο σωρό του σχήματος αυτού πρέπει να εξαχθεί η ρίζα, δηλαδή το 20. Η δομή έχει πλέον πέντε στοιχεία και θα πρέπει να αποκατασταθούν οι ιδιότητες του σωρού.

Έτσι λαμβάνεται το τελευταίο στοιχείο της δομής, δηλαδή το 5, και τοποθετείται στην κενή ρίζα.

Η δομή αυτή είναι μεν ένα σχεδόν πλήρες δυαδικό δένδρο, δεν είναι όμως ένα δένδρο μεγίστων.

Για το λόγο αυτό το μεγαλύτερο παιδί της ρίζας, δηλαδή το 19, ανταλλάσσεται με το 5 όπως φαίνεται στο τελικό σχήμα.

Βέβαια είναι ευνόητο ότι η πορεία του κλειδιού που εγκαθίσταται στη ρίζα μπορεί να συνεχιστεί και να φθάσει μέχρι τα φύλλα, ανάλογα με τις τιμές των κλειδιών που θα βρεθούν στο σχετικό μονοπάτι.

Άρα κατά την εισαγωγή κλειδιού στη θέση *n* είναι ενδεχόμενο να διανυθεί το μονοπάτι προς τη ρίζα μήκους όσο το ύψος του *n*-οστού κόμβου, δηλαδή

$$h = \lfloor \log_2 n \rfloor$$

Εισαγωγή - Διαγραφή σε Σωρό Μεγίστων

Στη συνέχεια δίνονται οι διαδικασίες της εισαγωγής και της εξαγωγής σε ένα σωρό μμεγίστων.

Επειδή ο σωρός είναι ένα σχεδόν πλήρες δυαδικό δένδρο μπορεί εύκολα και αποτελεσματικά να υλοποιηθεί με ένα πίνακα που ονομάζουμε Heap.

Στη διαδικασία που ακολουθεί υποτίθεται ότι εισάγουμε το (n+1)-οστό στοιχείο στο Heap που περιέχει έχει αρκετά μεγάλο μέγεθος (ώστε να αποφύγουμε τις εντολές ελέγχου υπερχείλισης).

Αντίστοιχα στη διαδικασία της εξαγωγής υποθέτουμε ότι υπάρχει ένα τουλάχιστον στοιχείο προς εξαγωγή (ώστε να αποφύγουμε τις εντολές ελέγχου υποχείλισης).

Αλγόριθμος InsertMaxHeap(key);

1. *n* ← *n*+1; *hole* ← *n*; *done* ← false;
2. **while** not done **do**
3. **if** (*hole*==1) then *done* ← true
4. **else if** (key<=*Heap* [*hole* /2]) then *done* ← true
5. **else**
6. *Heap*[*hole*] ← *Heap* [*hole* /2]; *hole* ← *hole* /2
7. *Heap* [*hole*] ← *key*

Αλγόριθμος DeleteMaxHeap(key);

1. *key* ← *Heap*[1]; *temp* ← *Heap* [*n*]; *n* ← *n* -1;
2. *father* ← 1; *child* ← 2; *done* ← false;
3. **while** (*child* <=*n*) and (not *done*) do
4. **if** (*child* < *n*) then
   if (*Heap* [*child*]<*Heap* [*child* +1]) then
   child ← *child* +1;
5. **if** (*temp*>=*Heap* [*child*] then *done* ← true
6. **else**
7. *Heap*[*father*] ← *Heap* [*child*]; *father* ← *child* ;
   child ← 2* *child;*
8. *Heap*[*father*] ← *temp*

Κατασκευή Σωρού Μεγίστων

Ιδιαίτερα σημαντική είναι και η διαδικασία της αρχικής κατασκευής του σωρού.

Στη συνέχεια δίνεται η διαδικασία **MakeHeap** που κατασκευάζει ένα σωρό μεγίστων, δεδομένου ενός συνόλου *n* κλειδιών.

Αλγόριθμος MakeHeap;

1. **for** $i \leftarrow 2$ **to** *n* **do**
2. $child \leftarrow i$; $father \leftarrow i/2$;
3. **while** ($child \neq 1$) **and** (*Heap*[*father*]<=*Heap*[*child*]) **do**
4. swap (*Heap* [*father*], *Heap*[child]);
5. $child \leftarrow father$; $father \leftarrow child/2$;

Η προηγούμενη διαδικασία συνίσταται στη σταδιακή θεώρηση του αταξινόμητου πίνακα Heap και την εισαγωγή των κλειδιών στο σωρό μεγίστων, που σταδιακά μεγεθύνεται.

Θεωρούμε ότι δημιουργείται σωρός μεγέθους *i,* όταν το *i*-οστό κλειδί του πίνακα εισάγεται σε σωρό μεγέθους *i* - 1. Η τιμή του κλειδιού που πρόκειται να εισαχτεί, K_i συγκρίνεται με την τιμή του κλειδιού του κόμβου πατέρα K_j. Αν ισχύει $K_i > K_j$, τότε το περιεχόμενο των κόμβων ανταλλάσσεται.

Αυτή η διαδικασία συγκρίσεων και ανταλλαγών συνεχίζεται μέχρις ότου ή βρεθεί κάποιος πρόγονος με τιμή κλειδιού μεγαλύτερη από K_i ή ο νέος κόμβος γίνει η νέα ρίζα του σωρού. Στο σημείο αυτό τελειώνει η φάση της δημιουργίας του σωρού μεγέθους *i* επιτυγχάνοντας τη βέβαιη τοποθέτηση του μεγαλύτερου κλειδιού στη ρίζα του σωρού.

Πρόταση 9.

Στη χειρότερη περίπτωση ο αριθμός των συγκρίσεων και ανταλλαγών κλειδιών κατά τη δημιουργία του σωρού με τον αλγόριθμο Make-Heap είναι πολυπλοκότητας $O(n\log_2 n)$.

Απόδειξη.

Ο αριθμός των επαναλήψεων στον εξωτερικό βρόχο είναι *n* - 1. Κατά την εισαγωγή του *i*-οστού κλειδιού απαιτούνται συγκρίσεις και ανταλλαγές στη χειρότερη περίπτωση, που αντιστοιχεί στην περίπτωση όπου το θεωρούμενο κλειδί καταλήγει να πάρει τη θέση της ρίζας του σωρού, δηλαδή θα διανύσει το μονοπάτι προς τη ρίζα που αντιστοιχεί στο ύψος του κόμβου *i* που γνωρίζουμε ότι είναι $\lfloor \log_2 i \rfloor$. Επομένως συνολικά η χειρότερη τιμή του αριθμού των ανταλλαγών είναι:

$$\sum_{i=2}^{n} \lfloor \log_2 i \rfloor \leq \sum_{i=2}^{n} \log_2 i = \log_2 2 + \log_2 3 + \ldots \log_2 n =$$

$$\log_2 (2.3.\ldots.n) = \log_2 (1.2.3\ldots n) = \log_2 n! \leq \log_2 n^n = n \log_2 n$$

Με χρήση του προσεγγιστικού τύπου του Stirling για τον υπολογισμό του *n*! προκύπτει ότι η χειρότερη τιμή του αριθμού των συγκρίσεων και ανταλλαγών είναι $O(n\log_2 n)$

Αναδρομικός Αλγόριθμος κατασκευής σωρού

Max-Heapify(*A*, *i*):

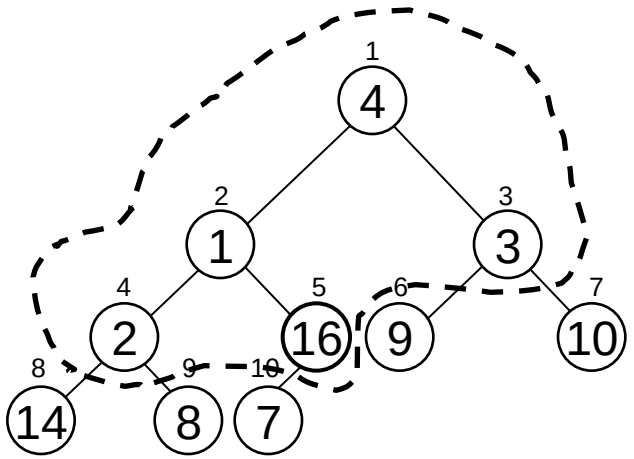
1. *left* ← 2*i;*
2. *right* ← 2*i* + 1
3. *largest* ← *i*
4. **if** *left* ≤ *heap_length*[*A*] **and** *A*[*left*] > A[*largest*] **then**
   largest ← *left*
5. **if** *right* ≤ *heap_length*[*A*] **and** *A*[*right*] > *A*[*largest*] **then**
   largest ← *right*
7. **if** *largest* ≠ *i* then
8. **swap** *A*[*i*] ↔ A[*largest*]
9. Max-Heapify(*A*, *largest*)

Build-Max-Heap(*A*):
heap_length[*A*] ← *length*[*A*]
for *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
Max-Heapify(*A*, *i*)

Κατασκευή Σωρού Μεγίστων

1. Μετατρέπουμε τον πίνακα A[1 … *n*] σε ένα max-heap (*n* = length[A])
2. Τα στοιχεία του υποπίνακα A[(⌊*n*/2⌋+1) .. *n*] είναι φύλλα
3. Εφαρμόζουμε τον MAX-HEAPIFY για τα στοιχεία μεταξύ των ⌊*n*/2⌋ και 1

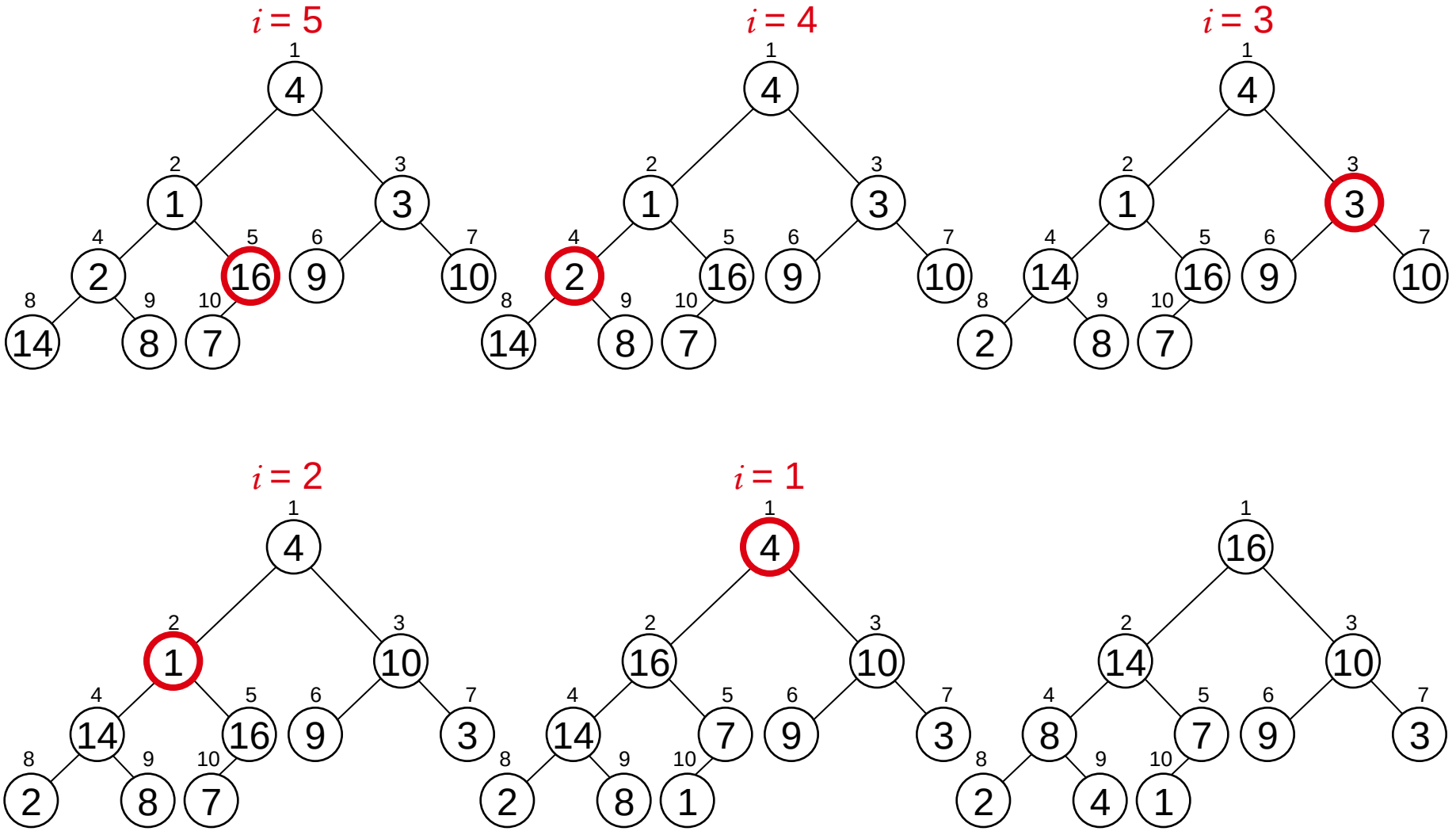
Build-Max-Heap(*A*):
heap_length[*A*] ← *length*[*A*]
for *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
    Max-Heapify(*A*, *i*)



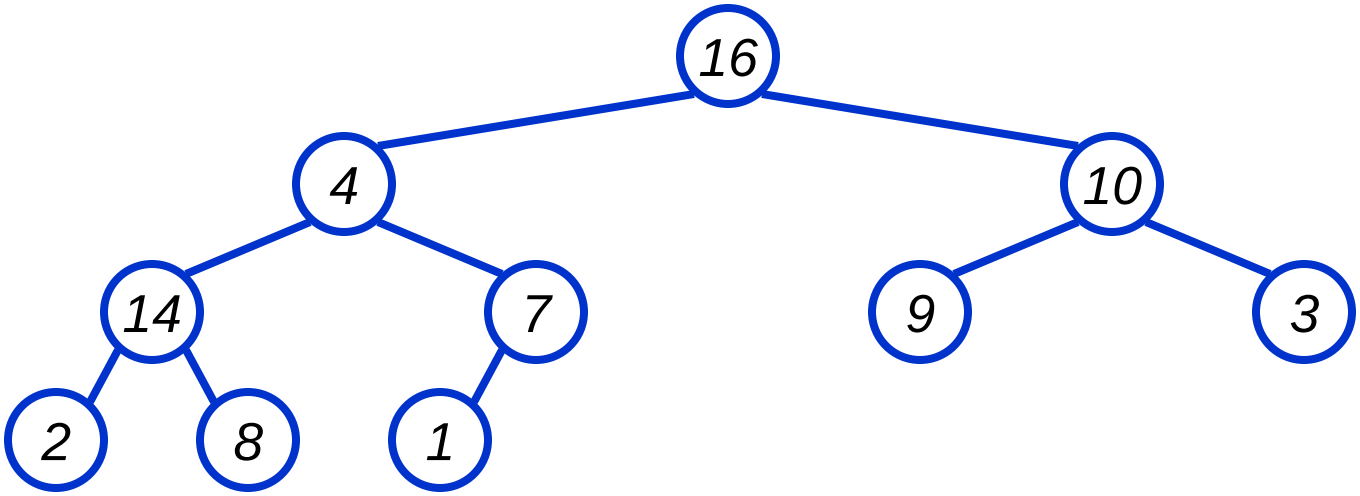
	1	2	3	4	5	6	7	8	9	10
A:	4	1	3	2	16	9	10	14	8	7

Example: A

4	1	3	2	16	9	10	14	8	7



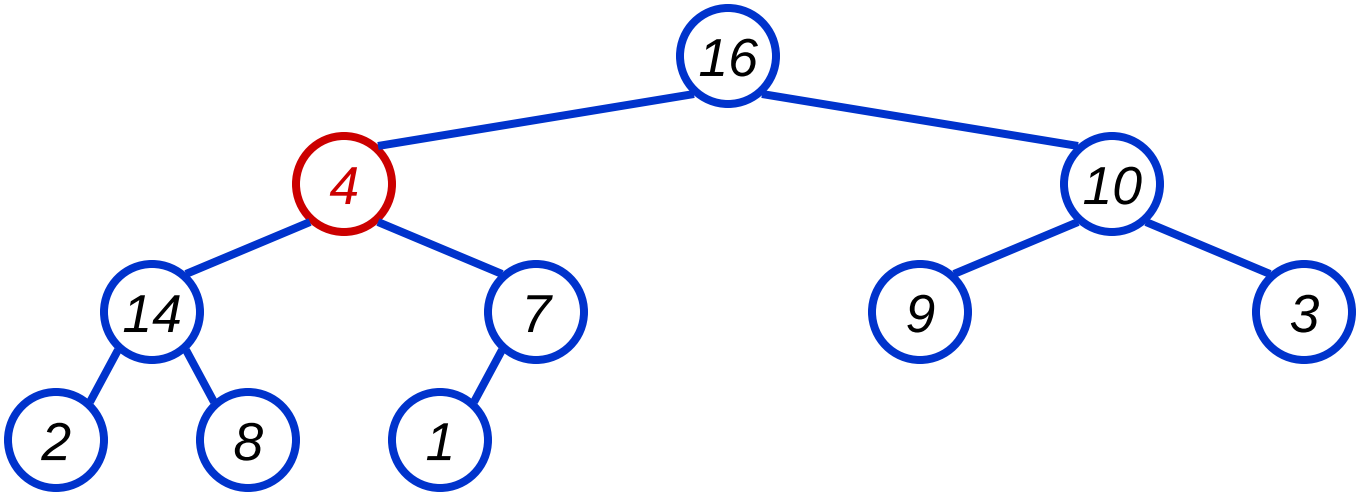
Max-Heapify() Example



$A =$

16	4	10	14	7	9	3	2	8	1

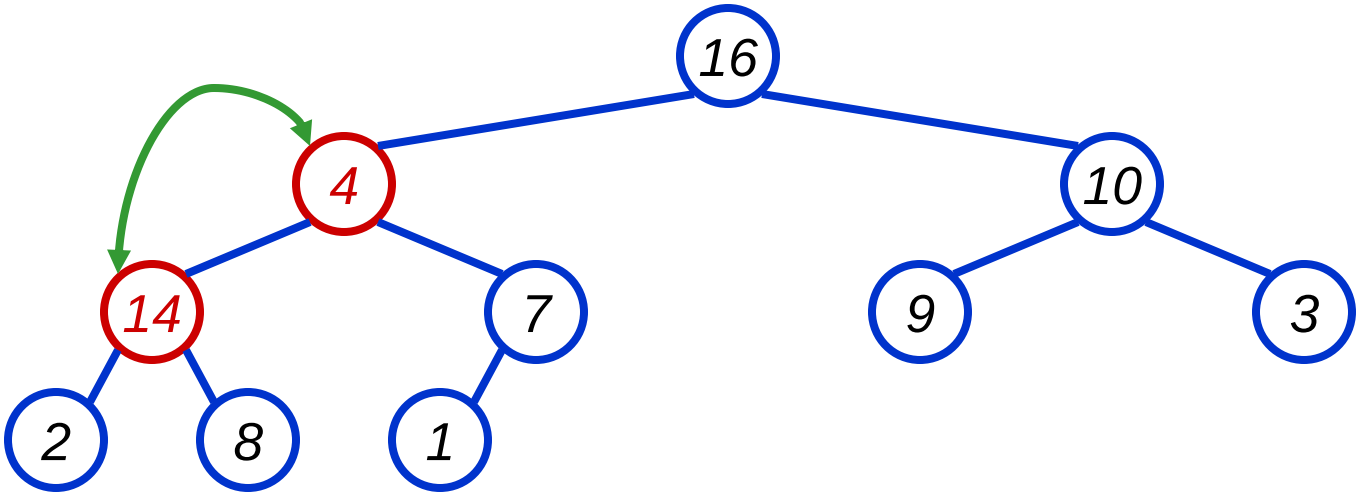
Max-Heapify() Example

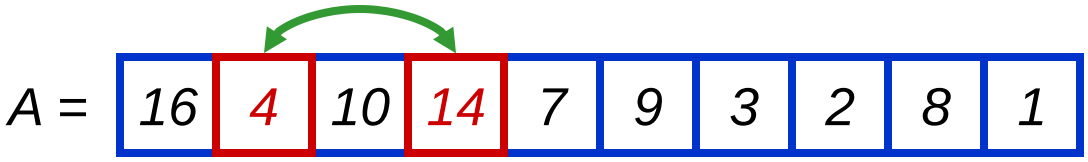


$A =$

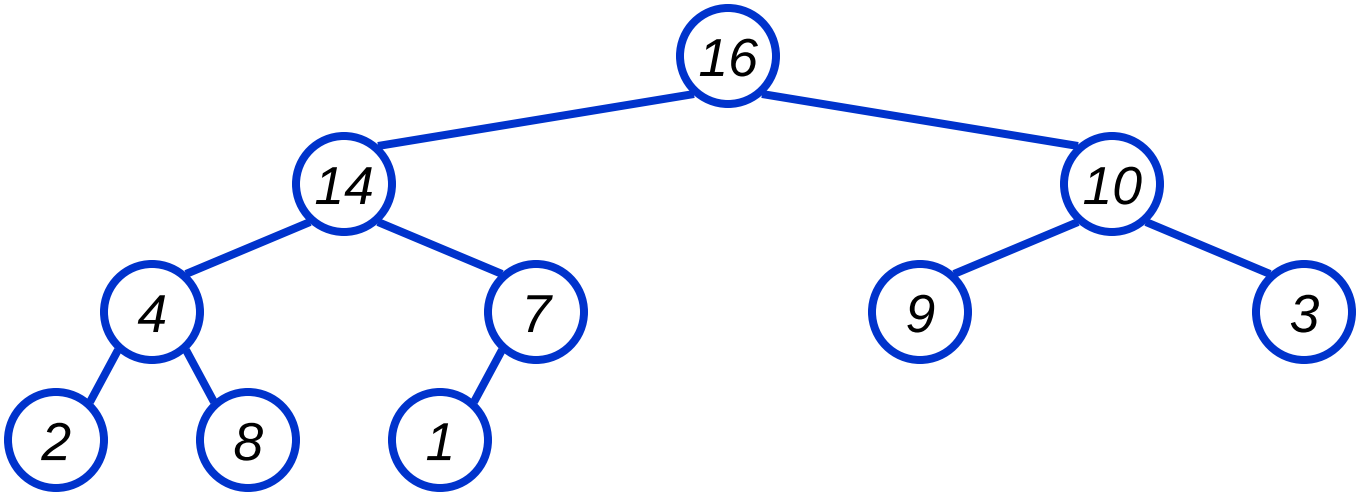
16	4	10	14	7	9	3	2	8	1

Max-Heapify() Example





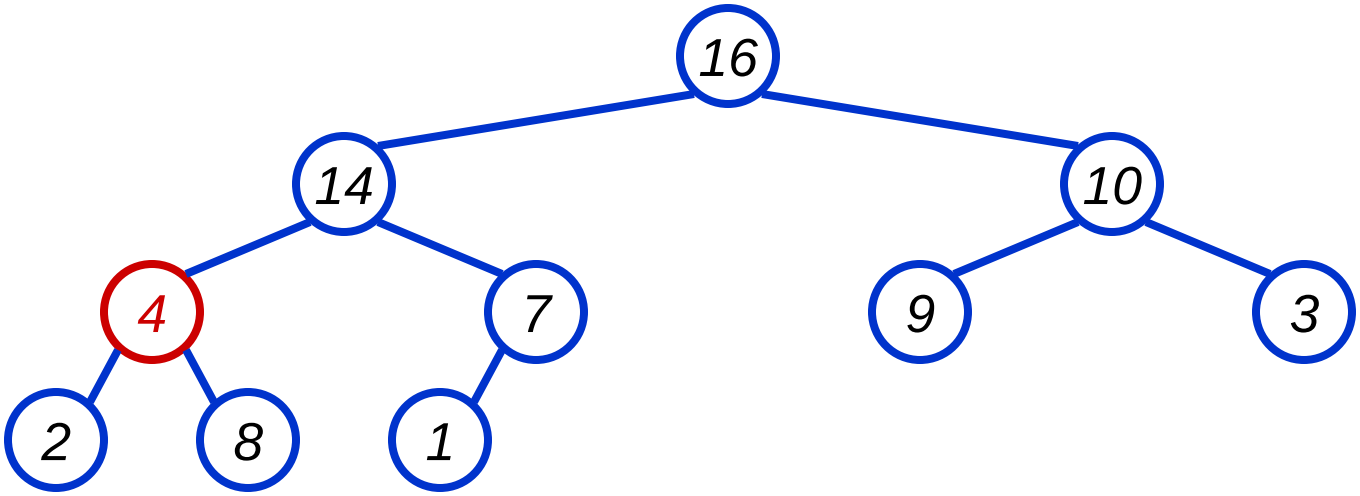
Max-Heapify() Example



$A =$

16	14	10	4	7	9	3	2	8	1

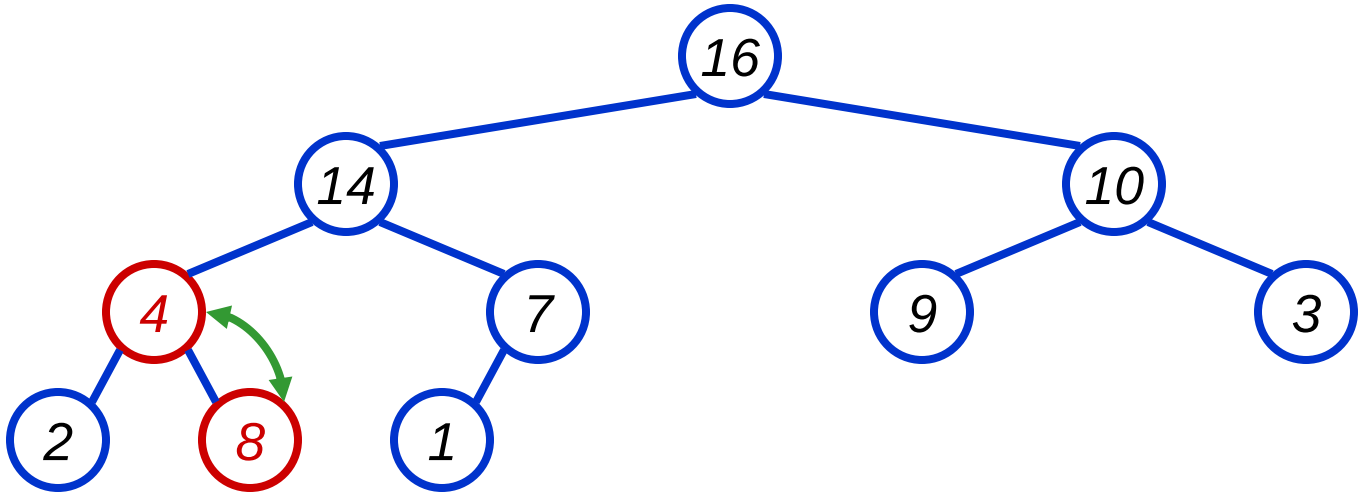
Max-Heapify() Example

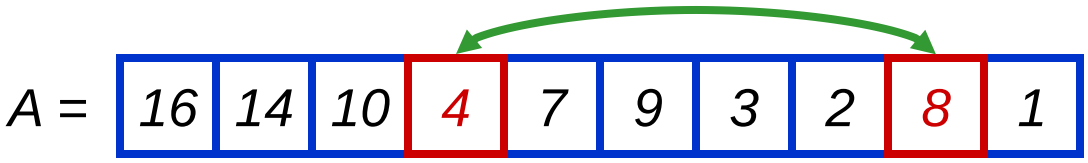


$A =$

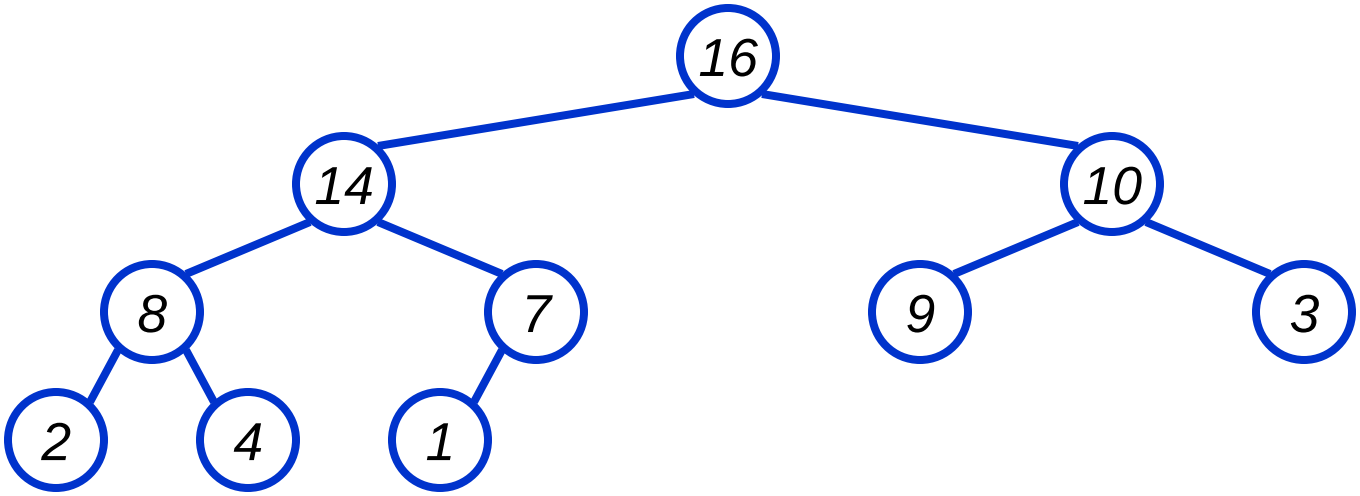
16	14	10	4	7	9	3	2	8	1

Max-Heapify() Example





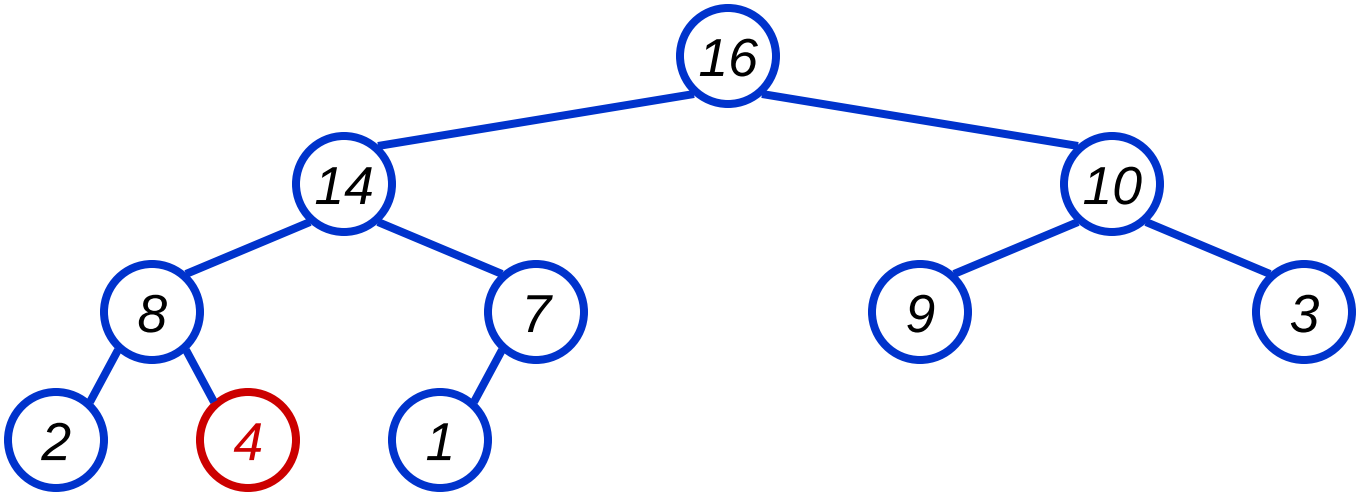
Max-Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

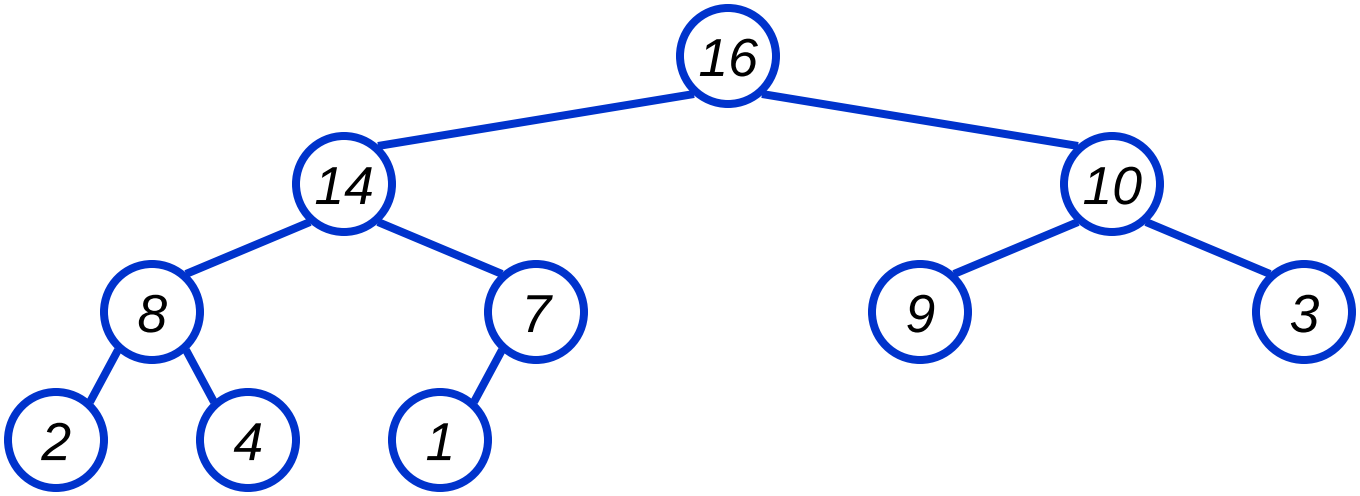
Max-Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

Max-Heapify() Example



$A =$

16	14	10	8	7	9	3	2	4	1

Διαισθητική ανάλυση του Max-Heapify

- Για την Μαχ-HEAPIFY(A, i)
  - έστω $h(i)$ το ύψος του κόμβου i
  - το πολύ $h(i)$ επίπεδα αναδρομής θα εκτελεστούν
- Σταθερός χρόνος για κάθε επίπεδο: Θ(1)

  Αρα $T(i) = O(h(i))$
- Ο σωρός σχεδόν πλήρες δυαδικό δένδρο

  $h(i) = O(\lg n)$

  $T(n) = O(\lg n)$

Ανάλυση του Max-Heapify

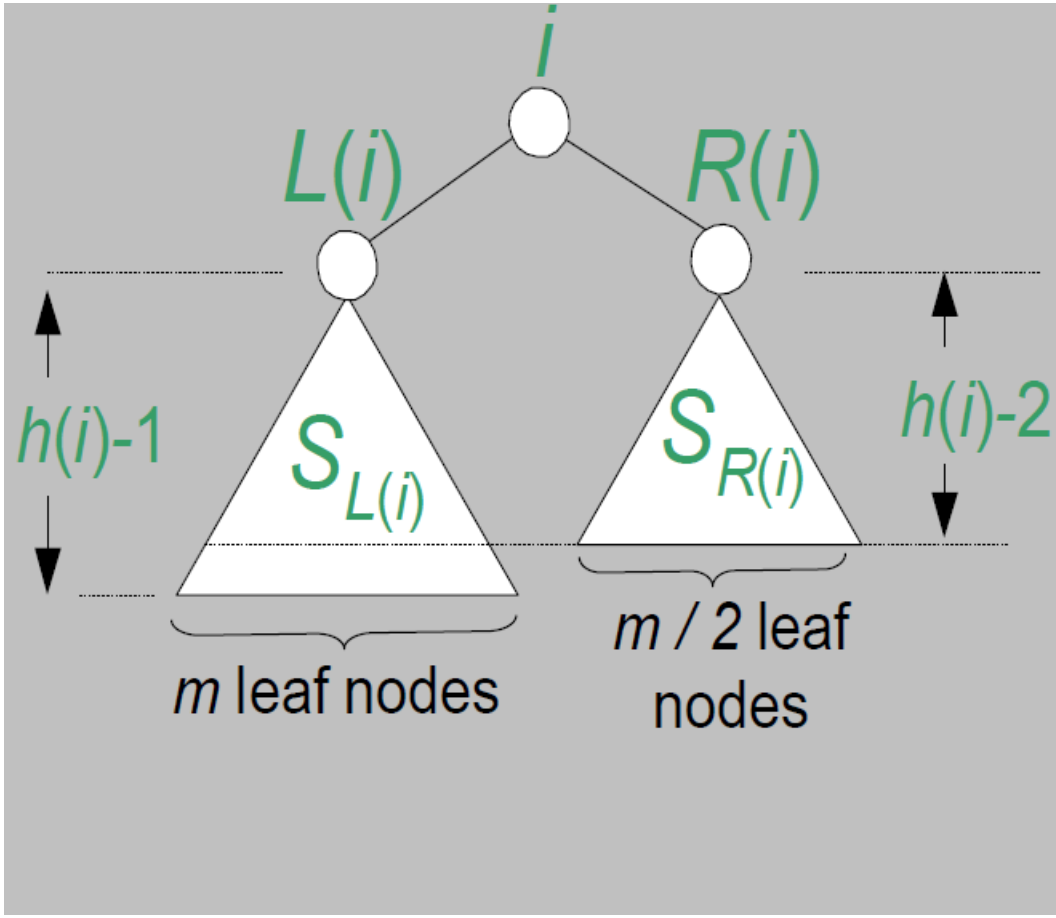
Ο χρόνος εκτέλεσης του Max-Heapify σε υποδένδρο μεγέθους n που εκφύεται από έναν δεδομένο κόμβο i αντιστοιχεί στο χρόνο $\Theta(1)$ που απαιτείται για να αποκατασταθούν οι σωστές σχέσεις μεταξύ των $A[i]$, $A[left]$ και $A[right]$ συν το χρόνο εκτέλεσης της Max-Heapify σε ένα υποδένδρο που εκφύεται από κάποιον από τους θυγατρικούς του κόμβου i. Το καθένα απ' αυτά τα υποδένδρα έχει μέγεθος μικρότερο ή ίσο από $2n/3$ – η χειρότερη περίπτωση προκύπτει όταν το τελευταίο επέιπεδο του δένδρου είναι ημισυμπληρωμένο – και επομένως ο χρόνος εκτέλεσης της Max-Heapify περιγράφεται από την αναδρομική σχέση

$$T(n) \leq T(2n/3) + \Theta(1)$$

Τυπική ανάλυση του Max-Heapify -1-

Απόδειξη:
Η χειρότερη περίπτωση συμβαίνει όταν η τελευταία γραμμή των κόμβων του υποδένδρου S_i που εκφύεται από τον κόμβο i είναι κατά το ήμισυ γεμάτο

$T(n) \leq T(|S_{L(i)}|) + \Theta(1)$



$S_{L(i)}$ και $S_{R(i)}$ είναι πλήρη δυαδικά δένδρα με ύψος $h(i) - 1$ and $h(i) - 2$, αντίστοιχα

Τυπική ανάλυση του Max-Heapify -2-

Εστω m ο αριθμός των φύλλων του $S_{L(i)}$

- $|S_{L(i)}| = \underbrace{m}_{\text{ext}} + \underbrace{(m-1)}_{\text{int}} = 2m-1$

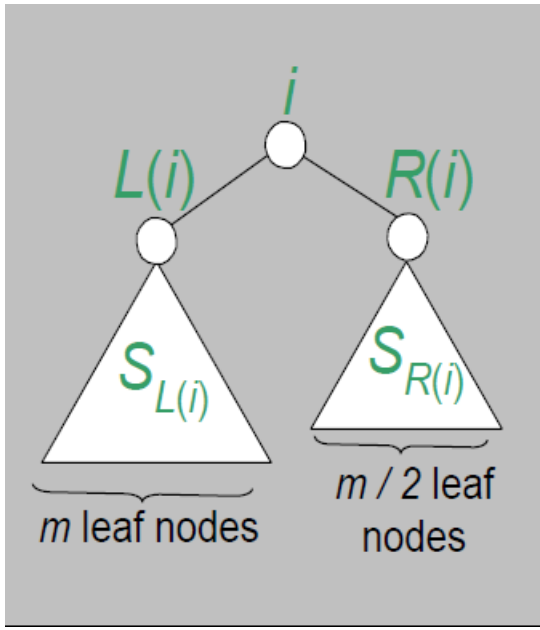
Πρόταση 4

- $|S_{R(i)}| = m/2 + (m/2 - 1) = m - 1$
- $|S_{L(i)}| + |S_{R(i)}| + 1 = n \Rightarrow$

$(2m-1) + (m-1) + 1 = n \Rightarrow m = (n+1)/3$

$|S_{L(i)}| = 2m - 1 = 2(n+1)/3 - 1 = (2n/3+2/3) - 1 =$
$= 2n/3 - 1/3 \leq 2n/3$

- $T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\lg n)$ από τη $2^{η}$ περίπτωση του κεντρικού θεωρήματος



Ανάλυση του Build-Max-Heap

Build-Max-Heap(*A*):

heap_length[*A*] ← *length*[*A*]

for *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do** — O(*n*)

  Max-Heapify(*A*, *i*) — O($\log_2 n$)

Μπορούμε να υπολογίσουμε ένα απλό άνω φράγμα για το χρόνο εκτέλεσης του Build-Max-Heap ως εξής: Σε κάθε κλήση της Max-Heapify (αποκατάσταση σωρού μεγίστου) απαιτεί χρόνο $O(\log_2 n)$ και εκτελούνται $O(n)$ τέτοιες κλήσεις. Επομένως ο χρόνος είναι $O(n\log_2 n)$.

Όμως το φράγμα αυτό δεν είναι ασυμπτωτικά αυστηρό. Μπορούμε να υπολογίσουμε ένα αυστηρότερο φράγμα. Παρατηρώντας ότι ο χρόνος εκτέλεσης της Max-heapify() εξαρτάται από το ύψος του κόμβου στο δένδρο.

Η βέλτιστη μέθοδος για τη κατασκευή του μέγιστου σωρού ξεκινά με την αυθαίρετη τοποθέτηση των στοιχείων σε ένα δυαδικό δέντρο, το δέντρο θα μπορούσε να αναπαρασταθεί με έναν πίνακα.

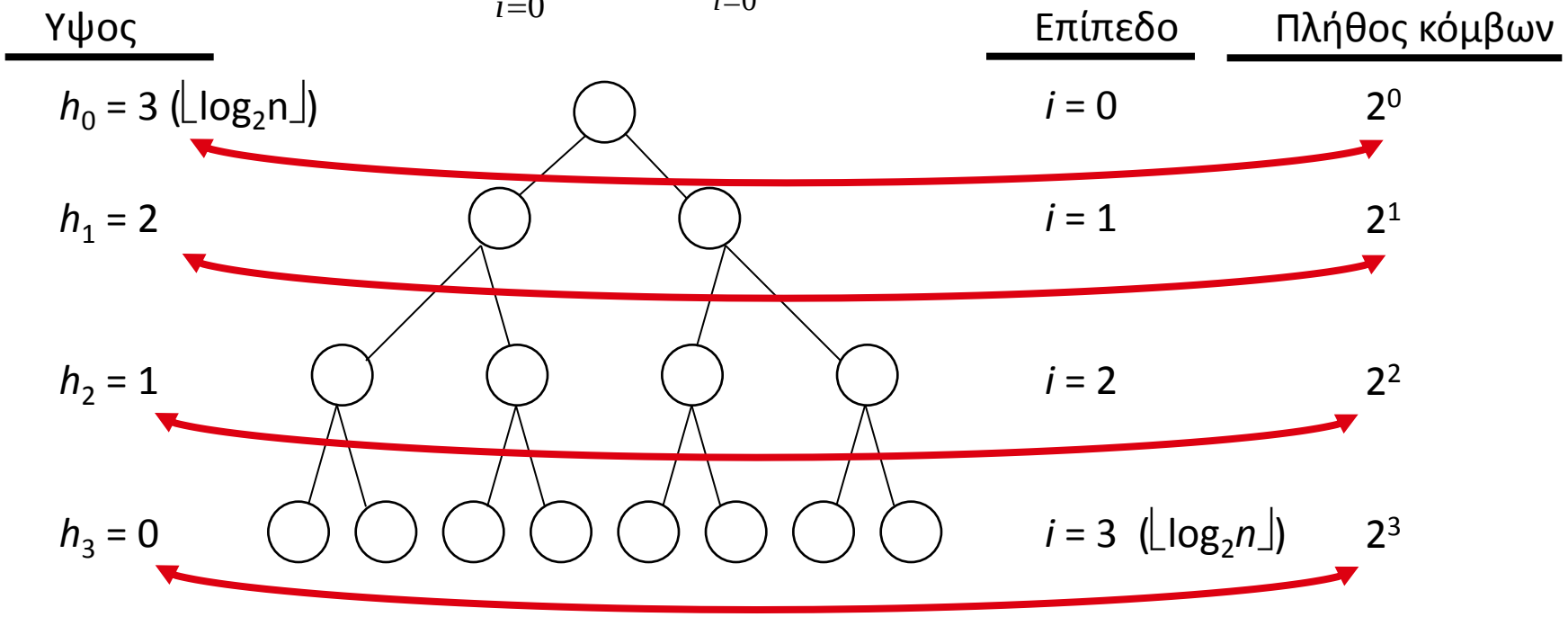
Στη συνέχεια, αρχής γενομένης από το χαμηλότερο επίπεδο και προχωρώντας προς τα πάνω, η ρίζα του κάθε υποδέντρου μετακινείται προς τα κάτω, όπως στον αλγόριθμο διαγραφής μέχρι η ιδιότητα του μέγιστου σωρού να αποκατασταθεί.

Πιο συγκεκριμένα, αν για όλα τα υποδένδρα που αρχίζουν από κάποιο ύψος *h* έχει ήδη αποκατασταθεί η ιδιότητα του μέγιστου σωρού ("heapified"), τότε για τα δέντρα σε ύψος *h*+1 μπορεί να αποκατασταθεί η ιδιότητα του μέγιστου σωρού στέλνοντας τη ρίζα τους προς τα κάτω, κατά μήκος της διαδρομής των παιδιών με τη μέγιστη αξία κατά την οικοδόμηση ενός μέγιστου σωρού.

Ανάλυση του Build-Max-Heap

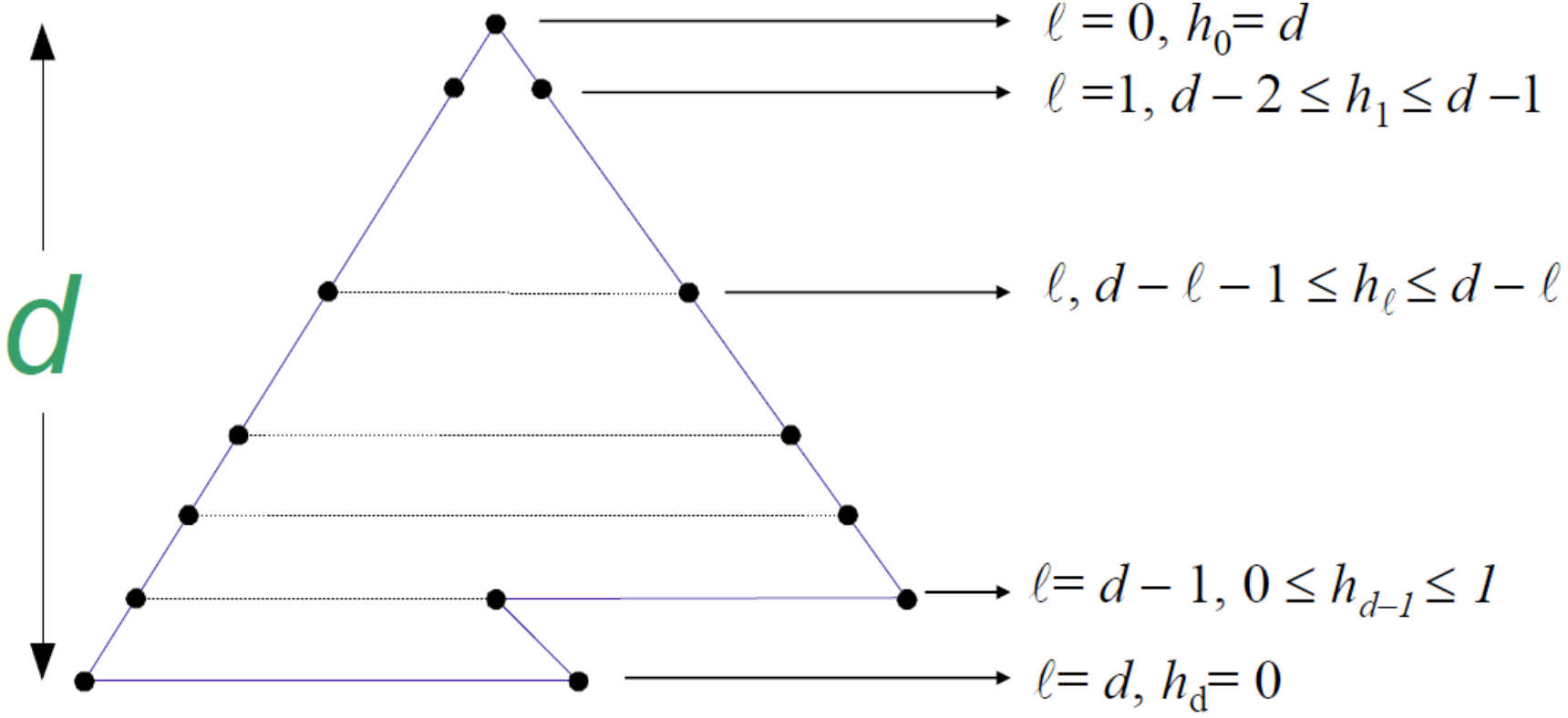
Πρόταση 10. Το κόστος για να αποκατασταθεί η ιδιότητα του μέγιστου σωρού για έναν κόμβο *i* είναι ανάλογο του ύψους του κόμβου *i*

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$



$h_i = h - i$ ύψος του σωρού με ρίζα στο επίπεδο *i*
$n_i = 2^i$ πλήθος των κόμβων στο επίπεδο *i*

Ανάλυση του Build-Max-Heap



Αν ο σωρός είναι πλήρες δυαδικό δένδρο τότε $h_l = d - l$

Αλλιώς οι κόμβοι δοσμένου επιπέδου δεν έχουν όλοι το ίδιο ύψος και ισχύει $d - l - 1 \le h \le d - l$

1η Απόδειξη

$$T(n) = \sum_{i=0}^{h} n_i h_i$$

Κόστος του Max-Heapify στο επίπεδο *i* ∗ πλήθος κόμβων στο επίπεδο *i*

$$= \sum_{i=0}^{h} 2^i (h-i)$$

Αντικαθιστούμε τις τιμές των n_i και h_i που υπολογίστηκαν πριν

$$= \sum_{i=0}^{h} \frac{h-i}{2^{h-i}} 2^h$$

Πολ/ζουμε με 2^h αριθμητή και παρονομαστή και γράφουμε 2^i ως $1/2^{-i}$

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$

Αντικαθιστούμε με: $k = h - i$

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

Το παραπάνω άθροισμα είναι μικρότερο από όλα τα στοιχεία για ∞ και $h = \log_2 n$

$$= O(n)$$

Το άθροισμα $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2} = \frac{1/2}{(1-1/2)^2} = 2$ είναι μικρότερο από 2.

2η Απόδειξη

Γνωρίζουμε ότι ένας σωρός *n* στοιχείων έχει ύψος $\lfloor \lg n \rfloor$ (πρόταση 3) και το πολύ $\lceil n/2^{h+1} \rceil$ κόμβους ύψους *h (Πρόταση 8)*. Ο χρόνος που απαιτεί η Max-Heapify όταν καλείται για έναν κόμβο ύψους *h* είναι O(*h*) και επομένως το συνολικό κόστος της Build-Max-Heap μπορεί να φραγεί άνω ως εξής:

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{h}{2^h} \right\rceil \right)$$

Για να υπολογίσουμε το τελευταίο άθροισμα αντικαθιστούμε στον τύπο $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$ όπου *x* =1/2, και έχουμε $\sum_{k=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

ο χρόνος εκτέλεσης του Build-Max-Heap μπορεί να φραγεί ως

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil \right) = O\left(n \sum_{h=0}^{\infty} \left\lceil \frac{h}{2^h} \right\rceil \right) = O(2n) = O(n)$$

Ταξινόμηση με Σωρό

- Η αρχική μέθοδος, όπως διατυπώθηκε από τους Williams και Floyd, είναι λιγότερο αποτελεσματική από τη γρήγορη ταξινόμηση. Ωστόσο, νεότερη παραλλαγή της **ταξινόμησης με σωρό** (*Heapsort*) είναι η καλύτερη μέθοδος που έχει εμφανισθεί στη βιβλιογραφία.
- Η ταξινόμηση με σωρό αποτελείται από δύο φάσεις:

  (α) τη φάση της δημιουργίας του σωρού, και

  (β) τη φάση της επεξεργασίας του σωρού.
- Η φάση α) έχει παρουσιαστεί εκτενώς παραπάνω.
- Κατά τη β) φάση της επεξεργασίας του σωρού το κλειδί της ρίζας ανταλλάσσεται με το τελευταίο κλειδί του σωρού και στο εξής δεν λαμβάνεται υπ' όψη. Όμως με την ανταλλαγή αυτή τα εναπομένοντα κλειδιά δεν αποτελούν πλέον σωρό. Έτσι ακολουθεί μία διαδικασία αποκατάστασης των ιδιοτήτων του σωρού.

Ταξινόμηση με Σωρό

- Πιο συγκεκριμένα, το νέο κλειδί που βρέθηκε στη ρίζα του σωρού συγκρίνεται με το μεγαλύτερο από τα κλειδιά των κόμβων που είναι παιδιά της ρίζας.
- Αν το νέο κλειδί είναι μικρότερο, τότε τα κλειδιά ανταλλάσσονται.
- Η διαδικασία της καθόδου του νέου κλειδιού από τη ρίζα προς κατώτερα επίπεδα συνεχίζεται μέχρι να βρεθεί μικρότερο κλειδί ή να φθάσει και πάλι στο επίπεδο των φύλλων.
- Στη συνέχεια και πάλι το κλειδί της νέας ρίζας ανταλλάσσεται με το κλειδί του (προ-)τελευταίου κόμβου του πίνακα.
- Έτσι η διαδικασία συνεχίζεται μέχρι να προκύψει σωρός μεγέθους 1.

Heapsort

- Στη συνέχεια δίνεται ο αλγόριθμος **HeapSort** που υλοποιεί την προηγούμενη ανάπτυξη χρησιμοποιώντας ένα σωρό μεγίστων.
- Η διαδικασία αποτελείται από δύο εντολές for.
- Η πρώτη εντολή for δημιουργεί τον αρχικό σωρό από τον αταξινόμητο πίνακα, ενώ η δεύτερη εντολή for επεξεργάζεται αυτόν το σωρό.
- Και στις δύο φάσεις καλείται ο αλγόριθμος **Restore**.

Heapsort

Αλγόριθμος Restore(*first*, *last*)

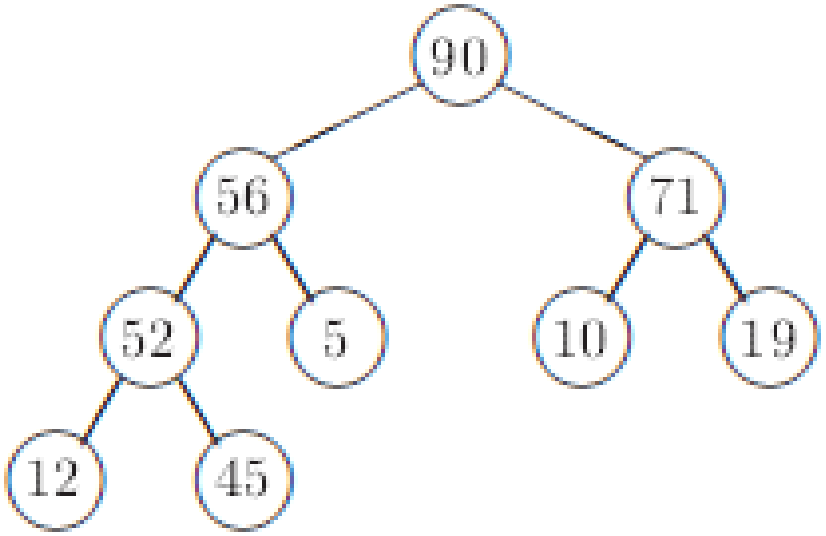
1. *father* ← *first*; *child* ← 2**father*;
2. **while** (*father* ≤ *last*) do
3. **if** (*child* < *last*) then
4. **if** (*A*[*child*] < *A*[*child*+1]) then *child* ← *child* + 1;
5. **if** *A*[*father*] < A[*child*] then
6. *A*[*child*/2] ← A[*child*]; *child* ← 2**child*;
7. **else** *father* ← *last* + 1;
8. *A*[*child*/2] <-- *A*[*father*] ;

Αλγόριθμος HeapSort

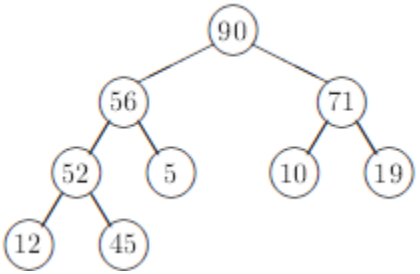
1. **for** *i* ← (*n*/2) downto 1 do Restore(*i*, *n*);
2. **for** *i* ← *n* downto 2 do
3. Swap(*A*[1], *A*[*i*]); Restore(1, *i*-1);

Heapsort Example

- Στο παρακάτω σχήμα φαίνεται η επεξεργασία του σωρού. Τα τετράγωνα πλαίσια δείχνουν ότι το κλειδί έχει καταλάβει την τελική του θέση και δεν υπόκειται σε άλλη επεξεργασία.
- Στη δεύτερη φάση της επεξεργασίας του σωρού η διαδικασία Restore καλείται οκτώ φορές μέσα από το δεύτερο for της διαδικασίας HeapSort. Έτσι, μετά την όγδοη κλήση τα κλειδιά είναι τοποθετημένα στον πίνακα κατά αύξουσα τάξη.

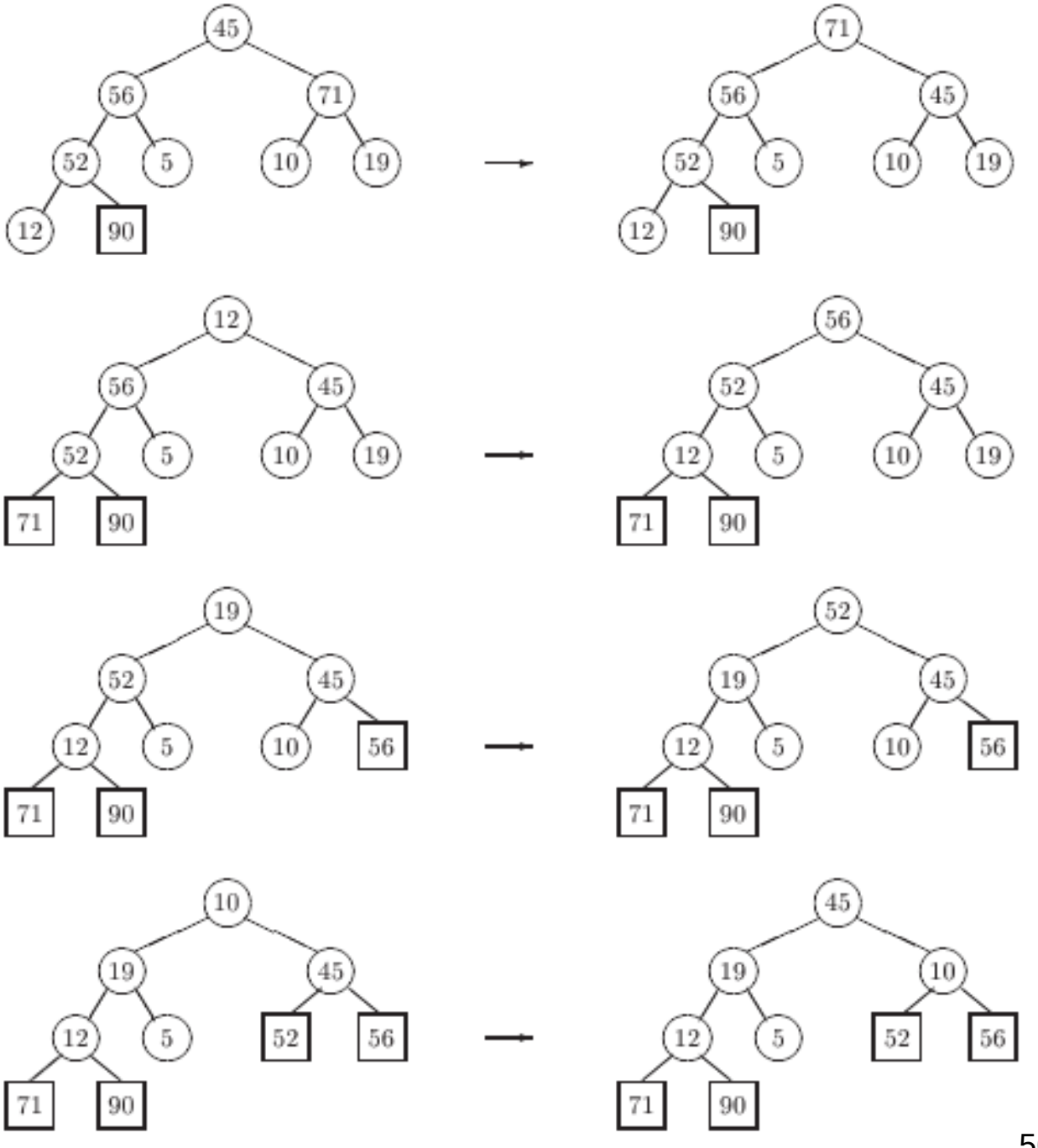


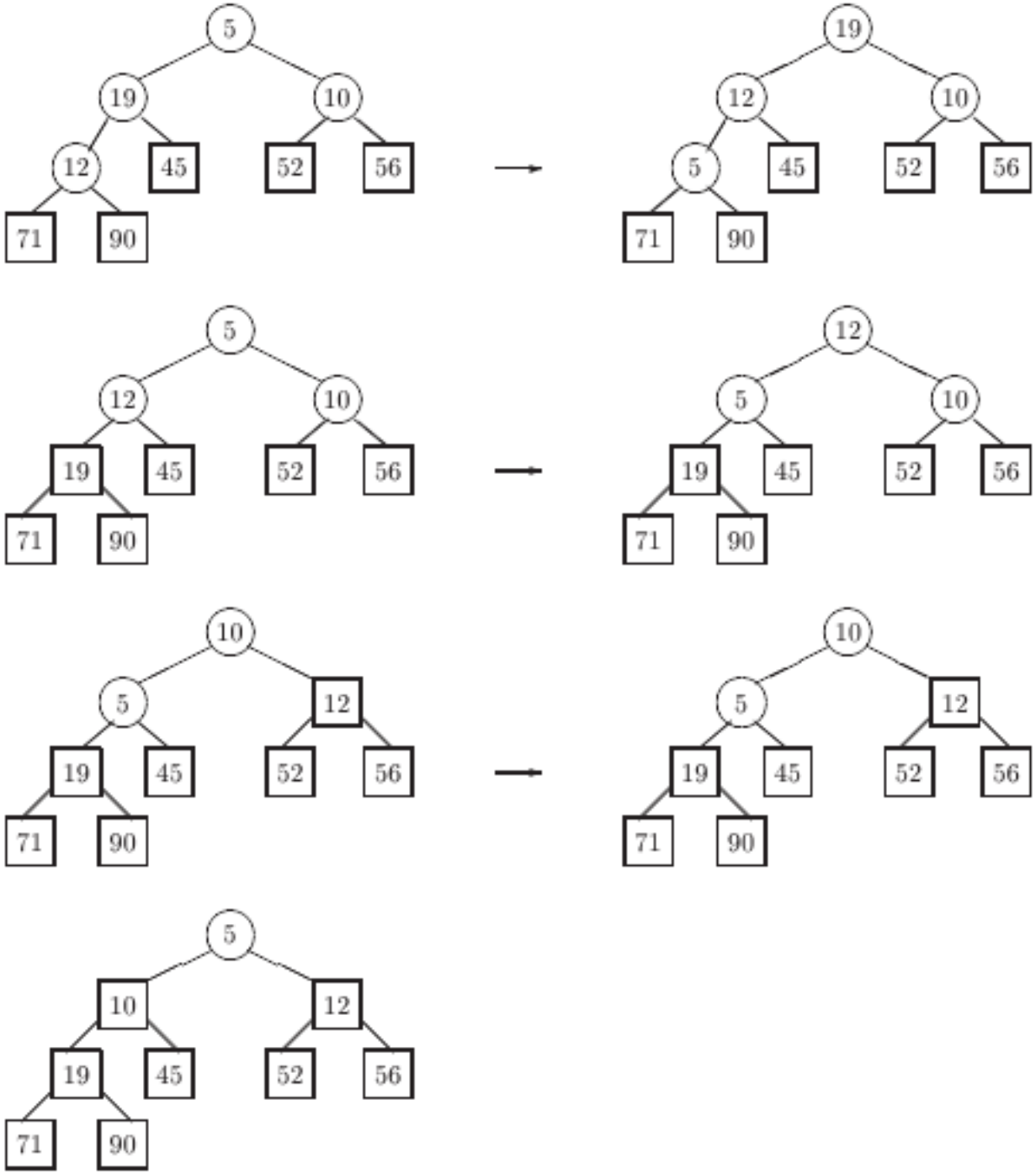
Έχουμε τον αρχικό σωρό



Και στο διπλανό Σχήμα φαίνεται η επεξεργασία του σωρού.

Τα τετράγωνα πλαίσια δείχνουν ότι το κλειδί έχει καταλάβει την τελική του θέση και δεν υπόκειται σε άλλη επεξεργασία.




5
19
10
12
45
52
56
71
90
19
12
10
5
45
52
56
71
90
5
12
10
19
45
52
56
71
90
12
5
10
19
45
52
56
71
90
10
5
12
19
45
52
56
71
90
10
5
12
19
45
52
56
71
90
5
10
12
19
45
52
56
71
90

Heapsort

- Με βάση το **Build-Max-Heap()** μια άλλη έκφραση του Heapsort μπορεί να δοθεί:
  - Μέγιστο στοιχείο είναι το $A[1]$
  - Το απομακρύνουμε και το ανταλλάσοντας με $A[n]$
    - Μειώνουμε το μέγεθος του heap (heap_size[A])
    - Το $A[n]$ έχει πλέον σωστή τιμή
  - Αποκαθιστούμε την ιδιότητα heap για το $A[1]$ καλώντας τη **Max-Heapify()**
  - Επαναλαμβάνουμε ανταλλάσοντας το $A[1]$ με το A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
        Build-Max-Heap (A);                          O(n)
        for (i = length(A) downto 2)                 } n-1 times
        {
                Swap(A[1], A[i]);                    }
                heap_size(A) -= 1;                   }
                Max-Heapify(A, 1);      O(log2n)     }
        }
}
```

$O(n\log_2 n)$ μπορεί να αποδειχθεί $\Theta(n\log_2 n)$

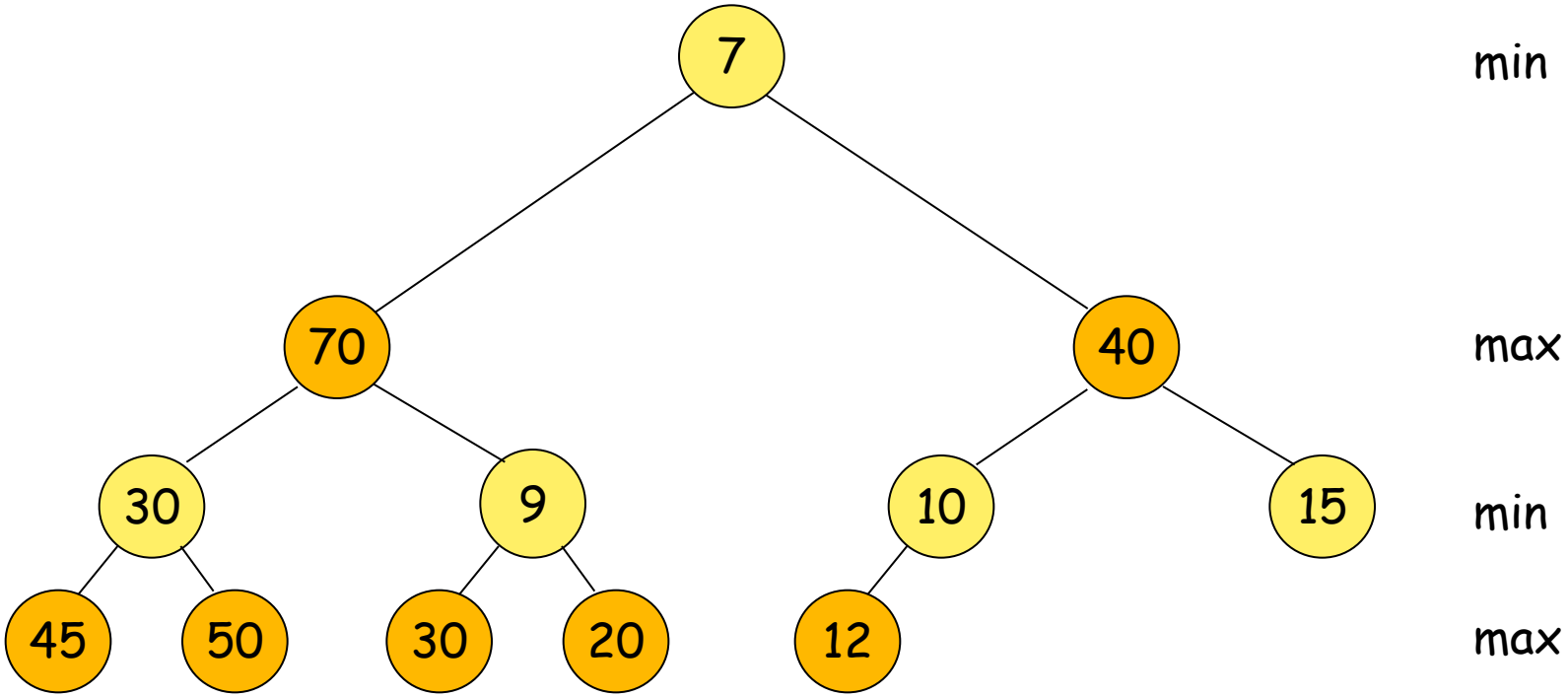
Σωρός Ελαχίστων και Μεγίστων

- Η αφηρημένη δομή δεδομένων **ουρά προτεραιότητας με δύο άκρα** (*doubleended priority queue*) διακρίνεται για τις εξής τρεις βασικές λειτουργίες:
  - εισαγωγή στοιχείου με οποιοδήποτε κλειδί,
  - εξαγωγή στοιχείου με το μικρότερο κλειδί, και
  - εξαγωγή στοιχείου με το μεγαλύτερο κλειδί.
- Ο **σωρός ελαχίστων και μεγίστων** (*min-max heap*), που προτάθηκε από τους Atkinson et al. (1986), είναι η δομή που υλοποιεί μία ουρά προτεραιότητας με δύο άκρα.
- **Ορισμός**. Ο **σωρός ελαχίστων και μεγίστων** είναι ένα σχεδόν πλήρες δυαδικό δένδρο. Τα κλειδιά των κόμβων που βρίσκονται στα περιττά (άρτια) επίπεδα είναι μικρότερα (αντίστοιχα, μεγαλύτερα) από τα κλειδιά όλων των κόμβων του αντίστοιχου υποδένδρου.

Σωρός Ελαχίστων και Μεγίστων

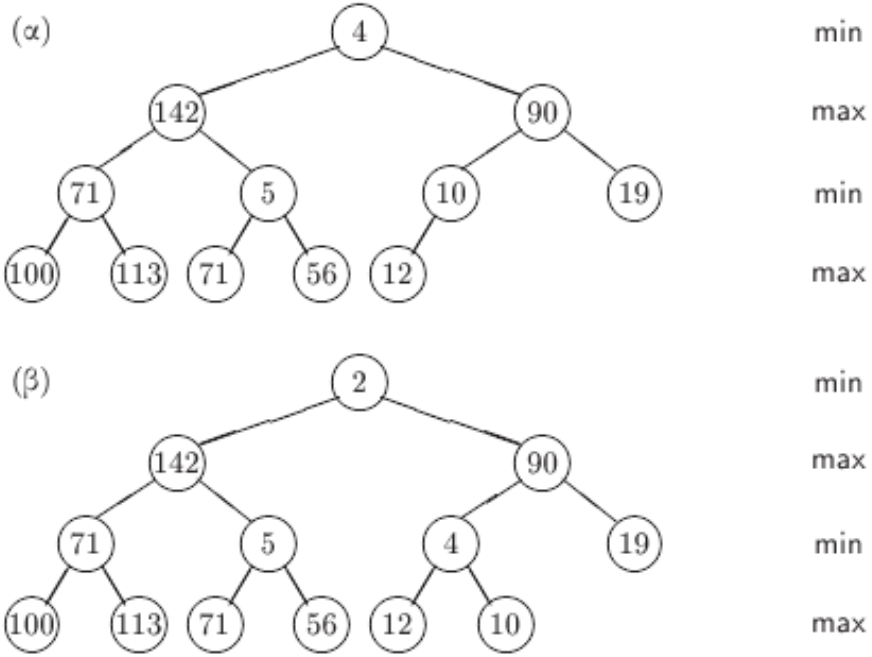
- Η ρίζα βρίσκεται πάντα στο επίπεδο των ελαχίστων.
- Έστω *x* οποιοσδήποτε κόμβος του min-max heap. Αν ο *x* βρίσκεται στο επίπεδο των ελαχίστων τότε ο κόμβος *x* έχει τη μικρότερη τιμή μεταξύ όλων των κόμβων του υποδένδρου με ρίζα τον *x*.
- Εστω *x* οποιοσδήποτε κόμβος του min-max heap. Αν ο *x* βρίσκεται στο επίπεδο των μεγίστων τότε ο κόμβος *x* έχει τη μεγαλύτερη τιμή μεταξύ όλων των κόμβων του υποδένδρου με ρίζα τον *x*.

Σωρός Ελαχίστων και Μεγίστων



Εισαγωγή στοιχείου στο min-max heap

- Συγκρίνεται το κλειδί με τον πατέρα του αν είναι μικρότερο από τον πατέρα του τότε είναι μικρότερο από όλα τα κλειδιά των κόμβων στο μονοπάτι από τον κόμβο αυτό μέχρι τη ρίζα. Χρειάζεται να ελεγχτούν μόνο τα κλειδιά στα επίπεδα των ελαχίστων. Αν το κλειδί είναι στη σωστή θέση σταματάμε αλλιώς το ανταλλάσουμε με τον πατέρα του.
- Συγκρίνεται το κλειδί με τον πατέρα του αν είναι μεγαλύτερο από τον πατέρα του τότε είναι μεγαλύτερο από όλα τα κλειδιά των κόμβων στο μονοπάτι από τον κόμβο αυτό μέχρι τη ρίζα. Χρειάζεται να ελεγχτούν μόνο τα κλειδιά στα επίπεδα των μεγίστων. Αν το κλειδί είναι στη σωστή θέση σταματάμε αλλιώς το ανταλλάσουμε με τον πατέρα του.



Στο σχήμα παρουσιάζεται η εναλλαγή του ρόλου των επιπέδων σε μία τέτοια δομή με 12 στοιχεία. Η ρίζα είναι στο πρώτο επίπεδο και συνεπώς περιέχει το μικρότερο κλειδί της δομής.

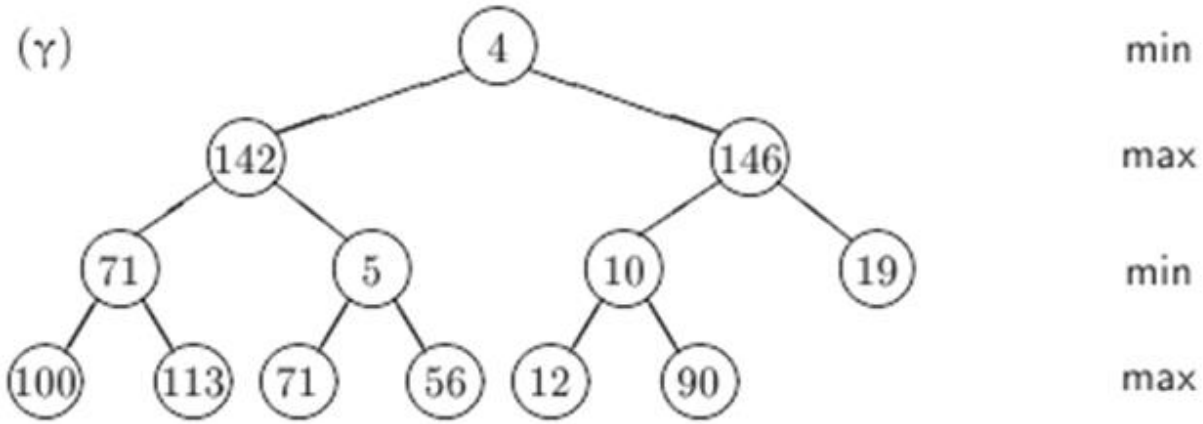
Εισαγωγή στοιχείου

Αν πρέπει στη δομή αυτή να εισαχθεί ένα στοιχείο, τότε αυτό θα καταλάβει τη θέση του δεξιού παιδιού του κόμβου με κλειδί 10, έτσι ώστε το δένδρο να είναι σχεδόν πλήρες.

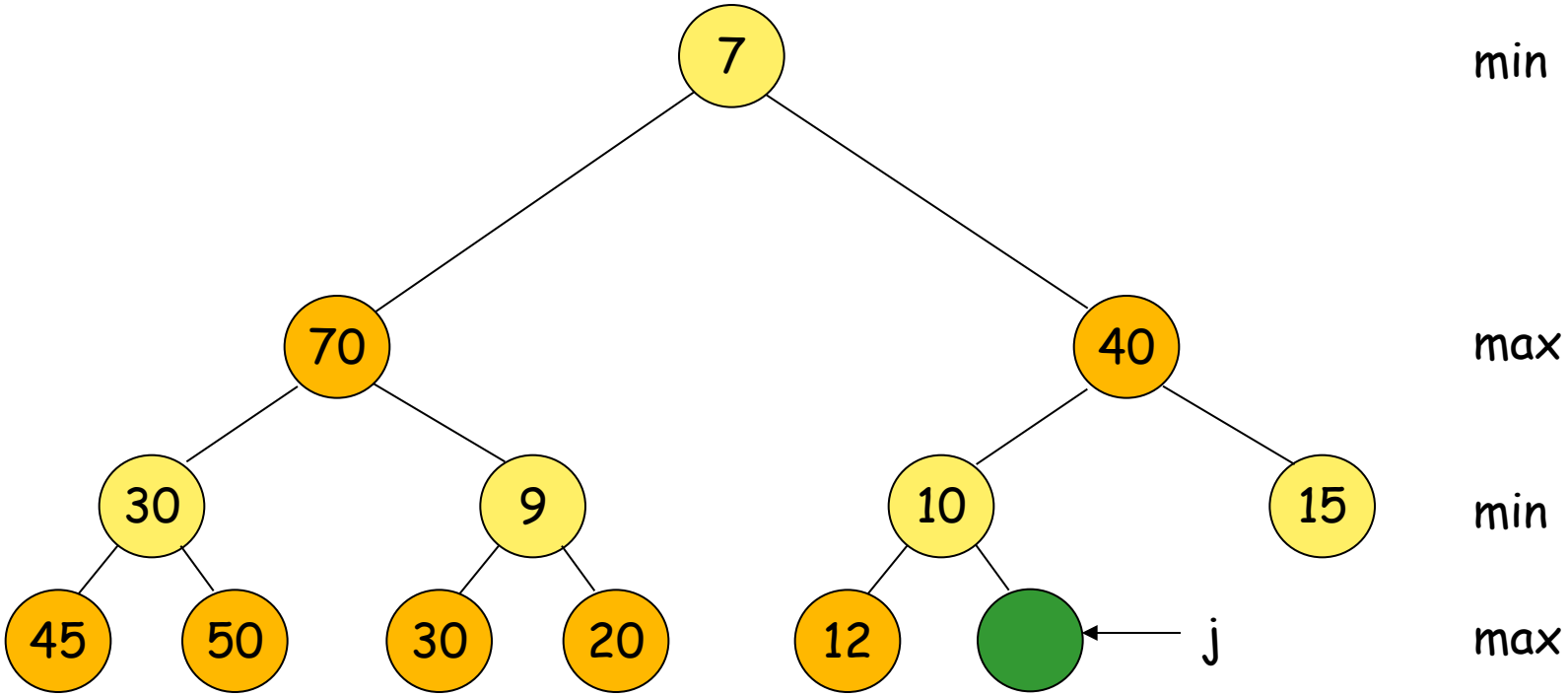
Το κλειδί του εισαγόμενου στοιχείου συγκρίνεται με το κλειδί του κόμβου πατέρα, ώστε να υπάρχει η σωστή σχέση διάταξης ανάλογα με το ρόλο των δύο διαδοχικών επιπέδων. Έτσι **αν το εισαγόμενο κλειδί έχει τιμή 2**, τότε διαπιστώνεται ότι έχει τιμή μικρότερη από τον κόμβο πατέρα που βρίσκεται σε επίπεδο ελαχίστων. Άρα αρχικά είναι απαραίτητο να γίνει μία ανταλλαγή μεταξύ των κλειδιών 2 και 10. Κατόπιν γίνεται έλεγχος στα επίπεδα των ελαχίστων στο μονοπάτι προς τη ρίζα, οπότε το κλειδί με τιμή 2 ανταλλάσσεται και με τη ρίζα που έχει τιμή 4, καταλήγοντας στη δομή του σχήματος (β).

Αν στην ίδια θέση έπρεπε να **εισαχθεί κλειδί με τιμή 146**, τότε θα χρειαζόταν μόνο ανταλλαγή με το κλειδί 90 του αμέσως ανωτέρου επιπέδου μεγίστων και θα προέκυπτε η αμέσως επόμενη δομή (σχήμα (γ)).

Εισαγωγή κλειδιού με τιμή 146 (σχήμα γ).

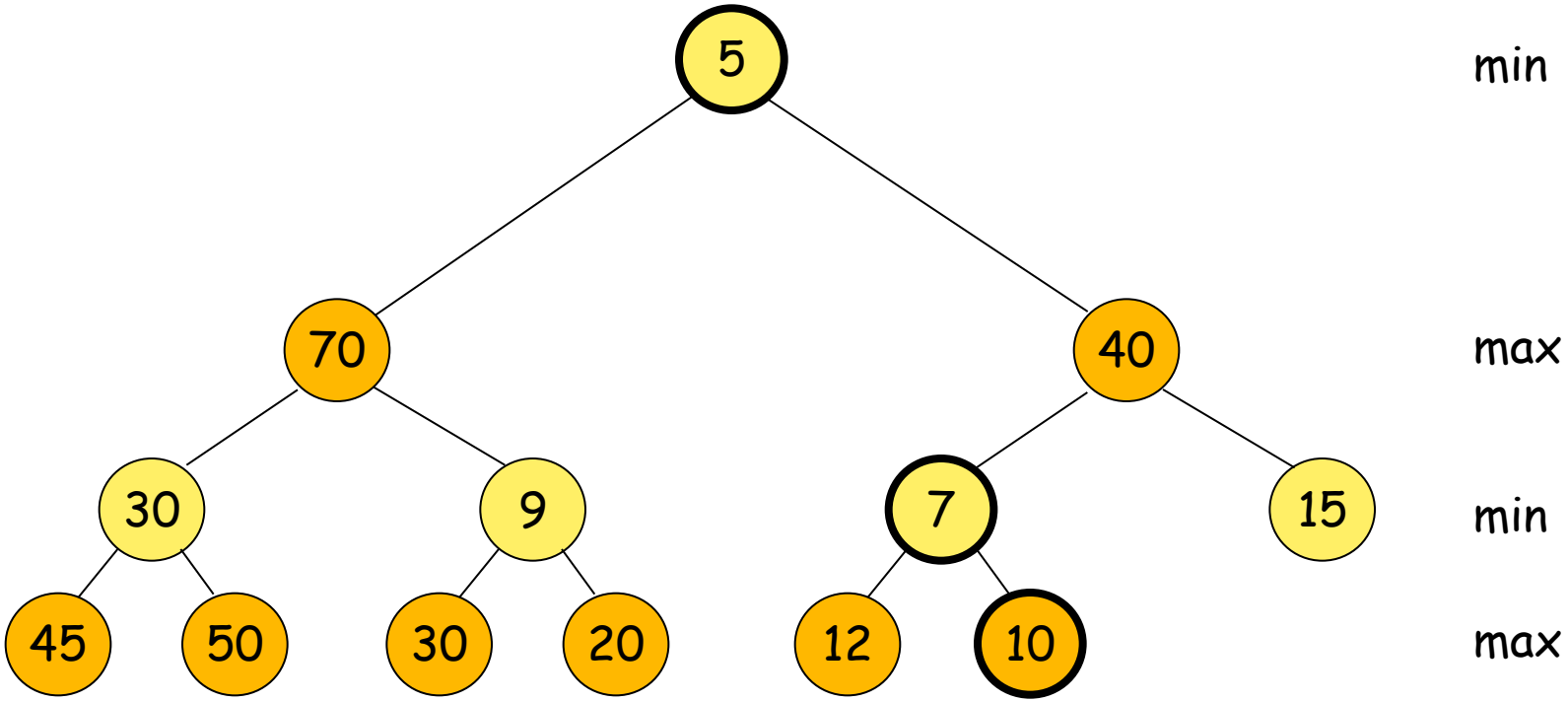


Εισαγωγή στο Min-Max Heap

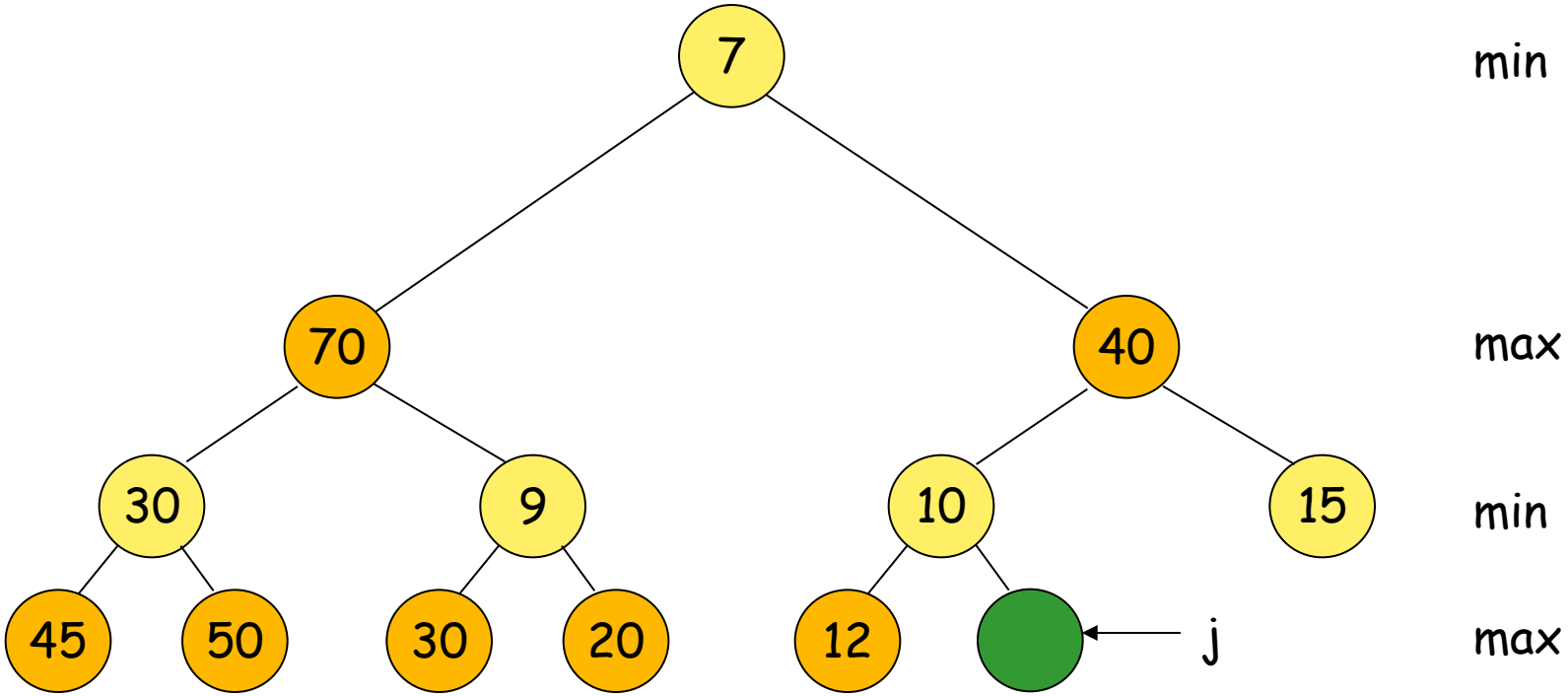


- Εισαγωγή του 5 στη θέση j. Επειδή 5 < 10 (είναι ο πατέρας του j), το 5 εγγυάται ότι είναι μικρότερο από όλα τα κλειδιά που βρίσκονται στα επίπεδα max στο μονοπάτι προς τη ρίζα. Οπότε χρειάζεται να ελεγχθεί μόνο με τα κλειδιά των min επιπέδων.

Min-Max Heap μετά την εισαγωγή του 5

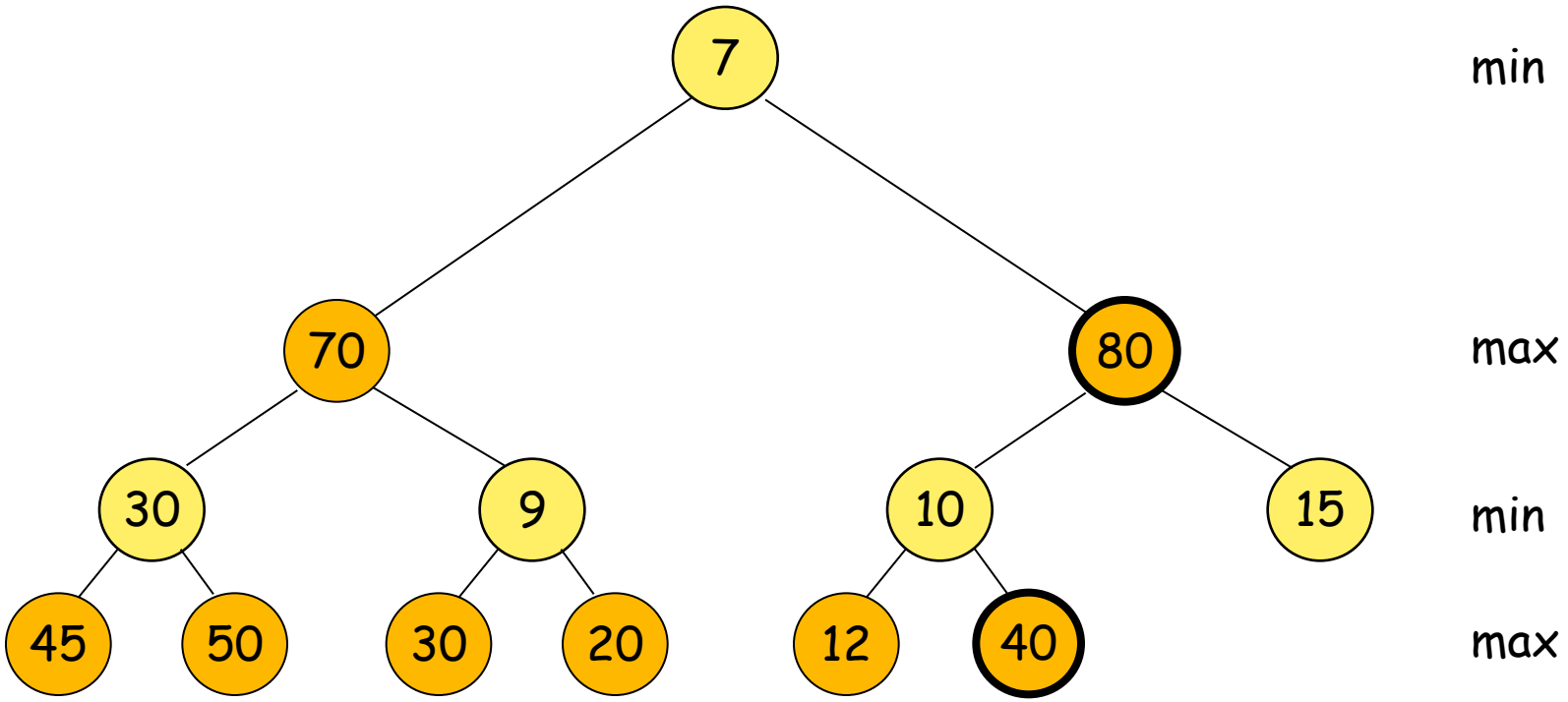


Εισαγωγή στο Min-Max Heap



Όταν εισάγεται το 80 στο min-max heap, επειδή 80 > 10, και το 10 είναι στο min επίπεδο, είμαστε βέβαιοι ότι το 80 είναι μεγαλύτερο από όλα τα κλειδιά που είναι και στα επίπεδα min στο μονοπάτι προς τη ρίζα. Χρειάζεται μόνο να ελεγχθεί με τα κλειδιά των max επιπέδων.

Min-Max Heap μετά την εισαγωγή του 80



Όταν εισάγεται το 80 στο min-max heap, επειδή 80 > 10, και το 10 είναι στο min επίπεδο, είμαστε βέβαιοι ότι το 80 είναι μεγαλύτερο από όλα τα κλειδιά που είναι και στα επίπεδα min. Χρειάζεται μόνο να ελέγχουν τα κλειδιά των max επιπέδων.

Διαγραφή στοιχείου

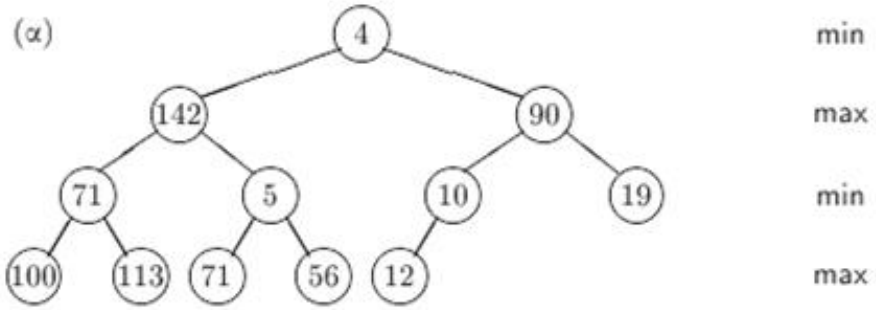
Από ένα σωρό ελαχίστων και μεγίστων πρέπει να εξαχθεί το μικρότερο στοιχείο που βρίσκεται στη ρίζα. Η θέση του θα αναπληρωθεί από το τελευταίο στοιχείο του σωρού, ώστε το δένδρο να είναι σχεδόν πλήρες. Ωστόσο, η δομή δεν είναι πλέον σωρός και επομένως χρειάζεται κάποια επεξεργασία, που έχει την αντίστροφη λογική από τη διαδικασία εισαγωγής.

i. Αν η ρίζα δεν έχει παιδιά, τότε η διαδικασία τελειώνει.

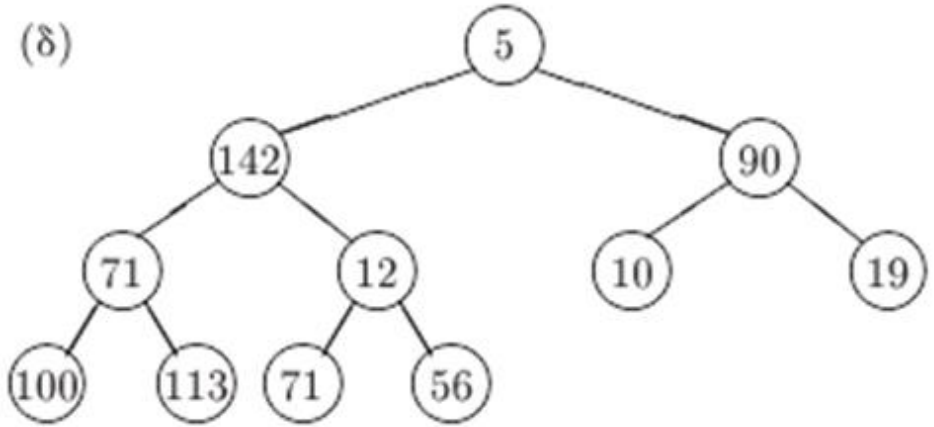
ii. Αν όμως η ρίζα έχει ένα τουλάχιστον παιδί, τότε πρέπει να ελεγχθούν τα δύο κατώτερα επίπεδα με σκοπό την εύρεση του μικρότερου κλειδιού που είναι μικρότερου από τη νέα ρίζα. Διακρίνονται, λοιπόν, τρεις περιπτώσεις:

   1. Αν δεν βρεθεί μικρότερο κλειδί από το κλειδί της ρίζας, τότε και πάλι η διαδικασία τελειώνει,
   2. Αν το μικρότερο κλειδί βρεθεί στο δεύτερο επίπεδο, που είναι επίπεδο μεγίστων, τότε αρκεί μία ανταλλαγή μεταξύ των δύο κλειδιών επειδή δεν υπάρχει σε όλη τη δομή άλλο κλειδί μικρότερο, ενώ
   3. Αν το μικρότερο κλειδί ανήκει στο τρίτο επίπεδο, που είναι επίπεδο ελαχίστων, τότε στη διαδικασία συμμετέχουν τα τρία κλειδιά του μονοπατιού και γίνονται οι εξής ενέργειες:
      a. το μικρότερο κλειδί αποθηκεύεται στη ρίζα,
      b. από τα άλλα δύο το μεγαλύτερο αποθηκεύεται στο δεύτερο επίπεδο, και
      c. στο τρίτο επίπεδο εισάγεται το απονέμον κλειδί. Έτσι μπορεί να εφαρμοσθεί και πάλι η ίδια διαδικασία αναδρομικά, γιατί ο κόμβος του τρίτου επιπέδου είναι η ρίζα ενός υποσωρού ελαχίστων και μεγίστων.

Διαγραφή στοιχείου – εξαγωγή ρίζας

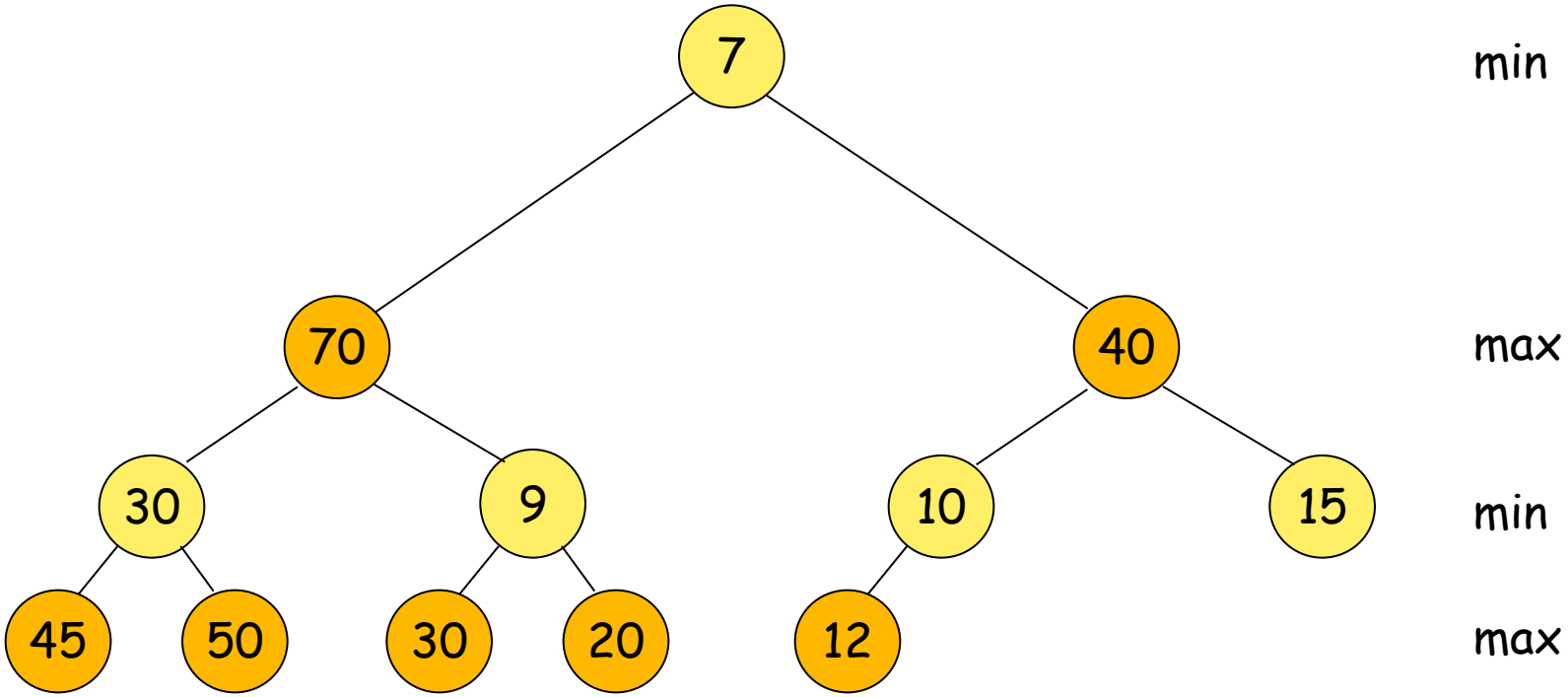


Σχήμα (δ). Εστω ότι **εξάγεται η ρίζα**, το κλειδί 4 (αρχικός σωρός σχήμα α). Στη θέση της έρχεται το τελευταίο κλειδί, δηλαδή το 12. Το μικρότερο κλειδί από το δεύτερο ή το τρίτο επίπεδο είναι το κλειδί 5. Συνεπώς στη διαδικασία εμπλέκονται τρία κλειδιά, το 12, το 142 και το 5.

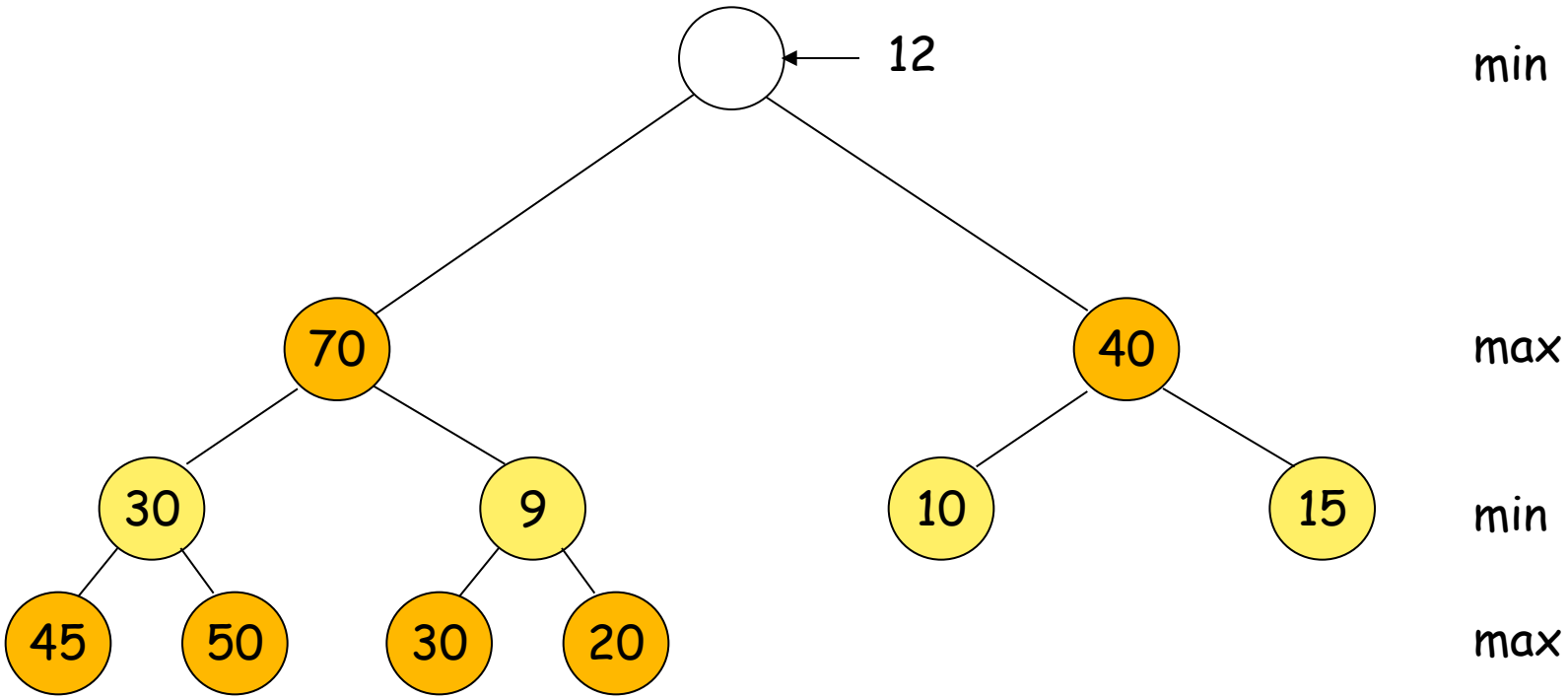


Το κλειδί 5 πρέπει να αποθηκευθεί στη ρίζα, το κλειδί 142 πρέπει να αποθηκευθεί στο δεύτερο επίπεδο, ενώ το κλειδί 12 πρέπει να εισαχθεί στον υποσωρό.

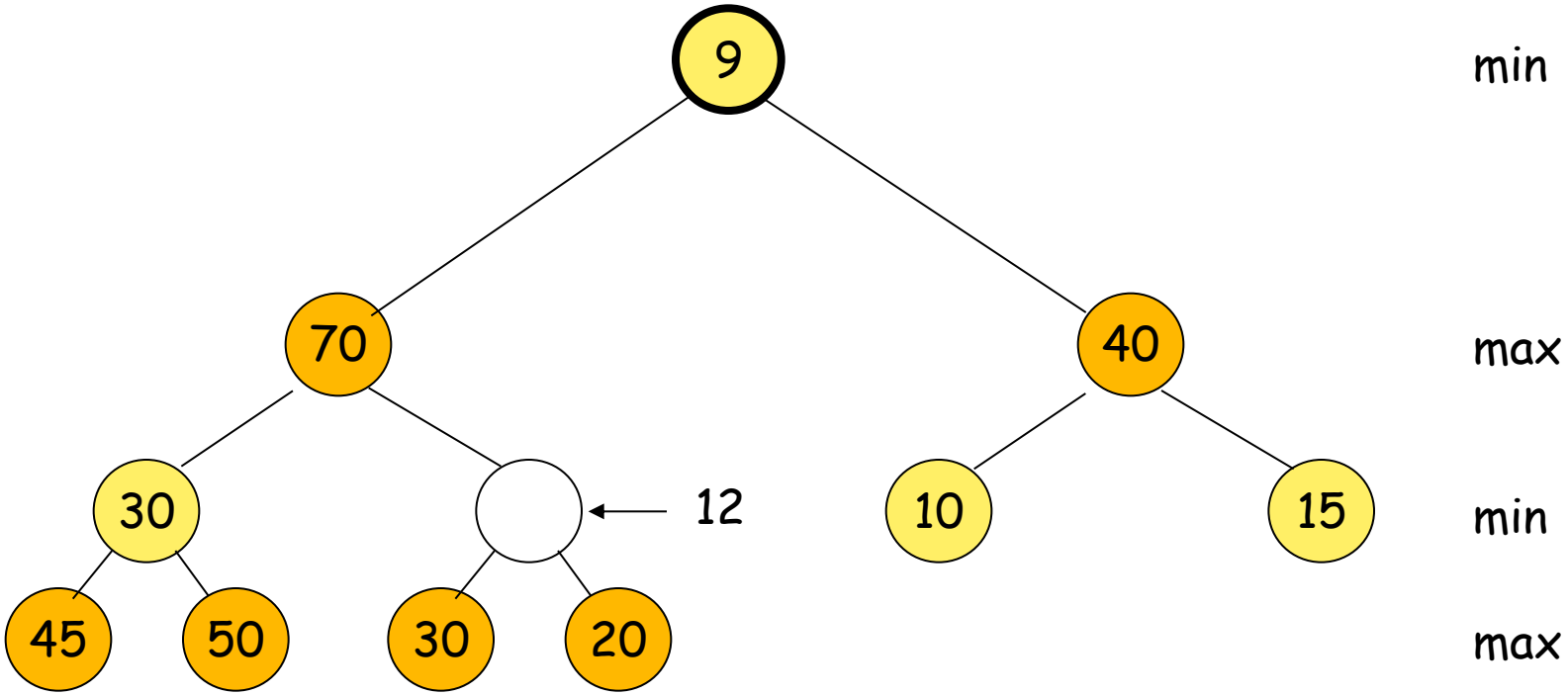
Διαγραφή στοιχείου - ρίζας



Διαγραφή στοιχείου - ρίζας



Min-Max Heap μετά τη διαγραφή του Min στοιχείου



Αλγόριθμοι Εισαγωγής - Διαγραφής σε Σωρό Ελαχίστων-Μεγίστων

Στη συνέχεια δίνεται οι αλγόριθμοι **InsertMinMax** και **DeleteMin** που εκτελούν τις αντίστοιχες λειτουργίες στον πίνακα MinMax που περιέχει ακέραια κλειδιά.

Η InsertMin καλεί τις διαδικασίες **VerifyMin** και **VerifyMax**, που ελέγχουν τα αντίστοιχα μονοπάτια προς τη ρίζα, και τη συνάρτηση **LevelMin** που επιστρέφει την τιμή true (false) αν η εισαγωγή γίνεται σε επίπεδο ελαχίστων (αντίστοιχα, μεγίστων).

Η διαδικασία **DeleteMin** καλεί τη διαδικασία **MinChildGrandChild**, που βρίσκει το μικρότερο κλειδί μεταξύ των κόμβων του δευτέρου και του τρίτου επιπέδου. Σε περίπτωση ισότητας κλειδιών από το δεύτερο και το τρίτο επίπεδο πρέπει να επιλεχθεί το κλειδί του δευτέρου επιπέδου για λόγους αποτελεσματικότητας.

Αλγόριθμος VerifyMax(*i*, *key*);

1. *GrandParent* ← *i*/4;
2. **while** (*GrandParent* ≠ 0) **do**
3. **if** (*key* > *MinMax*[*GrandParent*]) then
4. *MinMax*[*i*] ← *MinMax*[*GrandParent*];
5. *i* ← *GrandParent*;
6. *GrandParent* ← *GrandParent*/4
7. **else** *GrandParent* ← 0;
8. *MinMax*[*i*] ← *key*

Αλγόριθμος InsertMinMax(key);

1. $n \leftarrow n+1$; $p \leftarrow n/2$;
2. **if** ($p = 0$) then $MinMax[1] \leftarrow key$
3. **else**
4. **switch** *LevelMin*(p)
5. **case** true: ***//πατέρας σε επίπεδο ελαχίστων***
6. **if** ($key < MinMax[p]$) then
7. $MinMax[n] \leftarrow MinMax[p]$;
8. *VerifyMin*(n, key);
9. **else** *VerifyMax*(n, key);
10. **case** false: ***//πατέρας σε επίπεδο μεγίστων***
11. **if** ($key > MinMax[p]$) then
12. $MinMax[n] \leftarrow MinMax[p]$;
13. *VerifyMax*(n, key);
14. **else** *VerifyMin*(n, key)

Αλγόριθμος DeleteMin(key);

1. *key* ← *MinMax*[1]; *temp* ← *MinMax*[*n*];
2. *n* ← *n*-1; *i* ← 1; *j* ← *n*/2; *done* ← false;
3. **while** (*i* ≤ *j*) and (not *done*) **do**
4. *k* ← *MinChildGrandChild*(*i*);
5. **if** (*temp* ≤ *MinMax*[*k*]) then *done* ← true;
6. **else**
7. *MinMax*[*i*] ← *MinMax*[*k*];
8. **if** (*k* ≤ 2**i*+1) then *done* ← true;
9. **else**
10. *l* ← *k*/2;
11. **if** (*temp* > *MinMax*[*l*]) then
12. *Swap*(*temp*, *MinMax*[*l*]);
13. *i* ← *k;*
14. *MinMax*[*i*] ← *temp*

$k \le 2*i < 2*i+1 \Rightarrow$ k= άρτιος

k θέση του μικρότερου στοιχείου από η νέα ρίζα (μεταξύ αυτών που βρίσκονται στο 2ο και 3ο επίπεδο)

ii.1 Αν δεν βρεθεί μικρότερο κλειδί από το κλειδί της ρίζας, τότε και πάλι η διαδικασία τελειώνει

ii.2 Αν το μικρότερο κλειδί βρεθεί στο δεύτερο επίπεδο, που είναι επίπεδο μεγίστων, τότε αρκεί μία ανταλλαγή μεταξύ των δύο κλειδιών επειδή δεν υπάρχει σε όλη τη δομή άλλο κλειδί μικρότερο

l πατέρας του k

iii.3 Αν το μικρότερο κλειδί ανήκει στο τρίτο επίπεδο, που είναι επίπεδο ελαχίστων, τότε στη διαδικασία συμμετέχουν τα τρία κλειδιά του μονοπατιού και γίνονται οι εξής ενέργειες:

a. το μικρότερο κλειδί αποθηκεύεται στη ρίζα,
b. από τα άλλα δύο το μεγαλύτερο αποθηκεύεται στο δεύτερο επίπεδο, και
c. στο τρίτο επίπεδο εισάγεται το απονέμον κλειδί. Έτσι μπορεί να εφαρμοσθεί και πάλι η ίδια διαδικασία αναδρομικά, γιατί ο κόμβος του τρίτου επιπέδου είναι η ρίζα ενός υποσωρού ελαχίστων και μεγίστων.

Πρόταση.

Σε ένα σωρό ελαχίστων και μεγίστων η εισαγωγή τυχόντος κλειδιού, και η εξαγωγή του μικρότερου κλειδιού απαιτούν χρόνο τάξης $O(\log_2 n)$.

TABLE I. Worst-Case Complexities for Min-Heaps and Min-Max Heaps

	min-heap	min-max heap
Create	$2n$	$7n/3$
Insert	$\log(n+1)$	$0.5 \log(n+1)$
DeleteMin	$2 \log(n)$	$2.5 \log(n)$
DeleteMax	$0.5\, n + \log(n)$	$2.5 \log(n)$

Atkinson et al. Min-Max Heaps and Generalized Priority Queues

Διπλός Σωρός

- Ο **διπλός σωρός** (*double-ended heap, deap*) προτάθηκε από τον Carlsson (1987) και είναι μία δομή που εκτελεί τις ίδιες ακριβώς λειτουργίες που εκτελεί και ο σωρός ελαχίστων και μεγίστων και μάλιστα με την ίδια ποιοτική επίδοση.
- Η διαφορά τους είναι ότι η διαχείριση του διπλού σωρού είναι απλούστερη και επομένως η επίδοση είναι οριακά καλύτερη.

Διπλός Σωρός

Ορισμός.

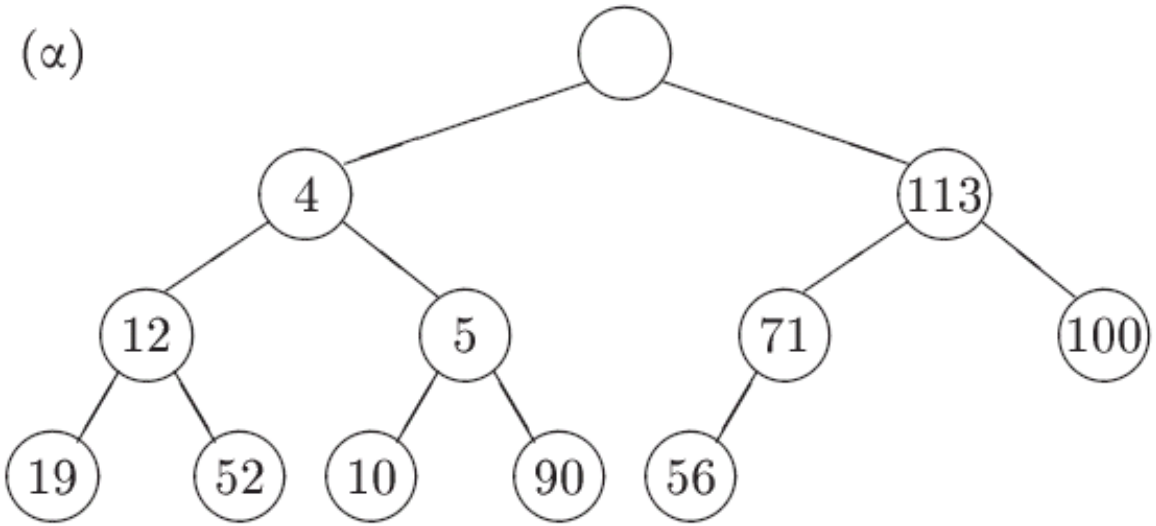
Ο διπλός σωρός είναι ένα σχεδόν πλήρες δυαδικό δένδρο με τα εξής χαρακτηριστικά:

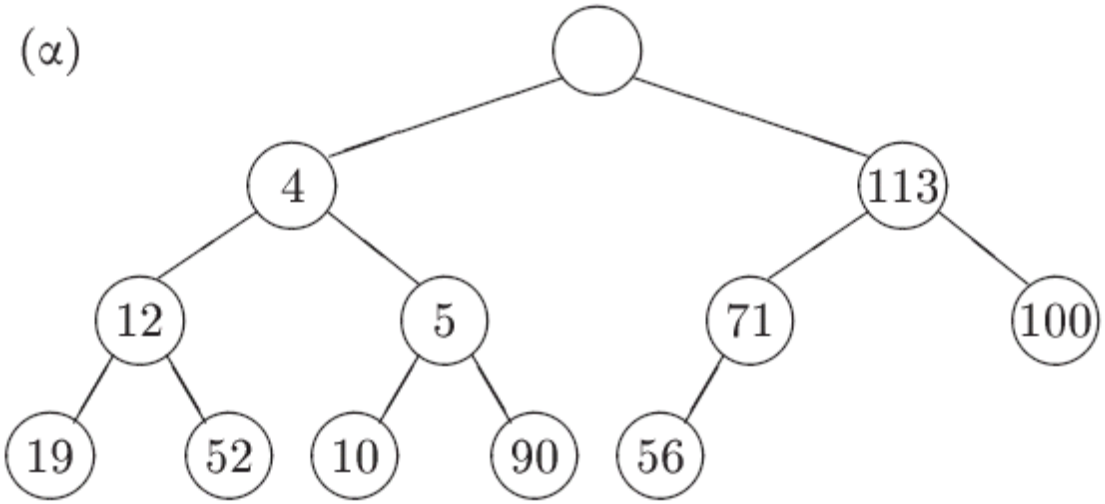
1. η ρίζα είναι κενή,
2. το αριστερό υποδένδρο είναι σωρός ελαχίστων,
3. το δεξιό υποδένδρο είναι σωρός μεγίστων, και
4. αν το δεξιό υποδένδρο δεν είναι κενό, *i* είναι ένας κόμβος του αριστερού υποδένδρου και *j* είναι ο συμμετρικός κόμβος του στο δεξιό υποδένδρο, τότε το κλειδί του κόμβου *i* είναι μικρότερο ή ίσο από το κλειδί του κόμβου *j*. Αν ο κόμβος *i* δεν έχει συμμετρικό κόμβο στο δεξιό υποδένδρο, τότε ως *j* λαμβάνεται ο συμμετρικός του πατέρα του κόμβου *i*.

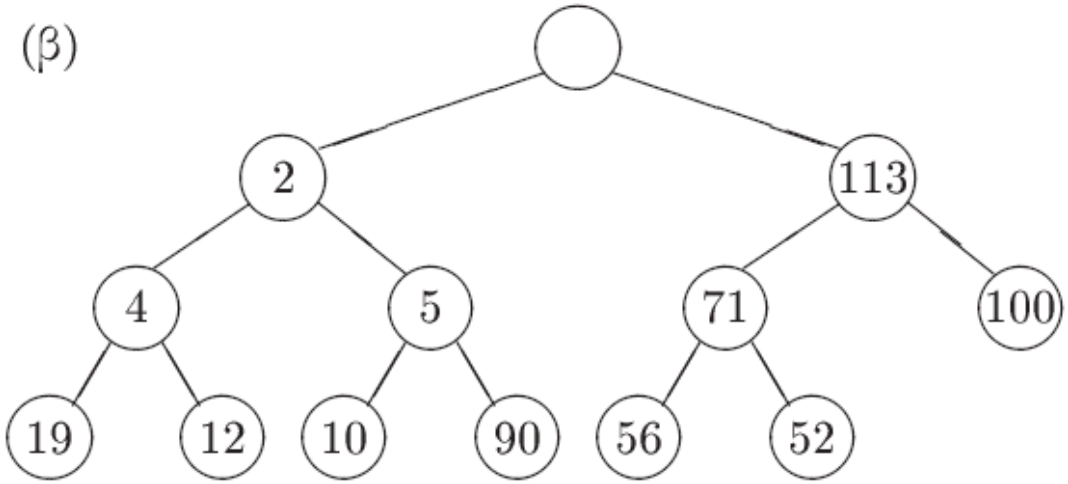
Στο επόμενο σχήμα δίνεται ένας διπλός σωρός με 12 κόμβους, δηλαδή με 11 κλειδιά.

Στο διπλό σωρό η ρίζα μένει κενή εκτός αν ο σωρός περιέχει ένα μόνο στοιχείο. Με βάση το τέταρτο χαρακτηριστικό του ορισμού απαιτείται η συσχέτιση των στοιχείων *i* και *j*. Ισχύει, λοιπόν:

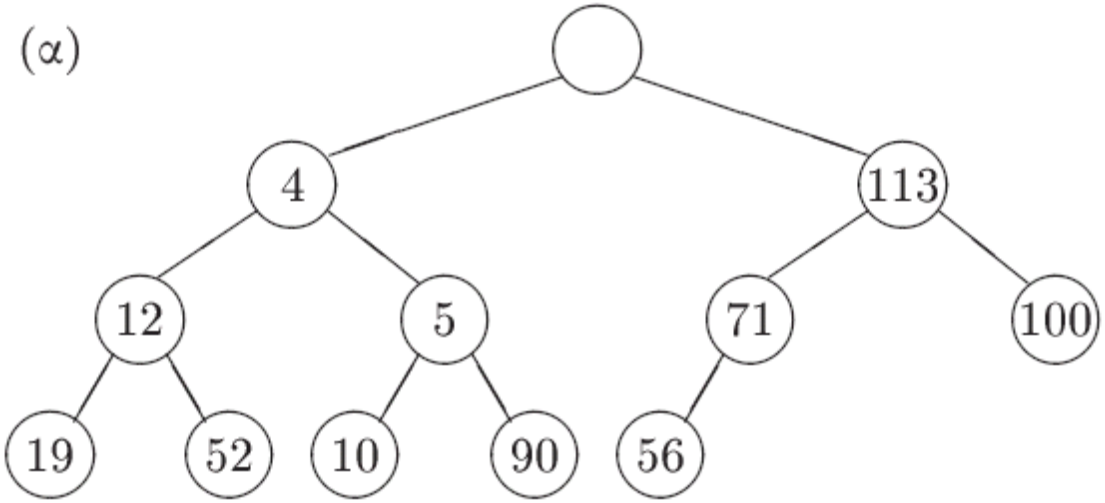
$$j = \begin{cases} i + 2^{\lfloor \log i \rfloor - 1} & \alpha\nu\ j \le n \\ \left\lfloor \frac{1}{2}\left(i + 2^{\lfloor \log i \rfloor - 1}\right) \right\rfloor & \alpha\nu\ j > n \end{cases}$$

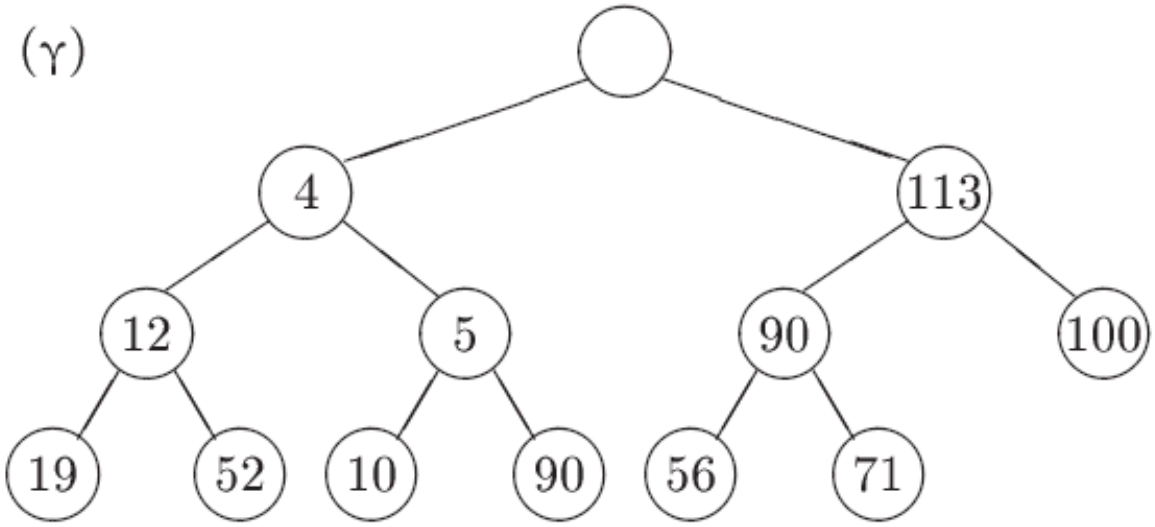






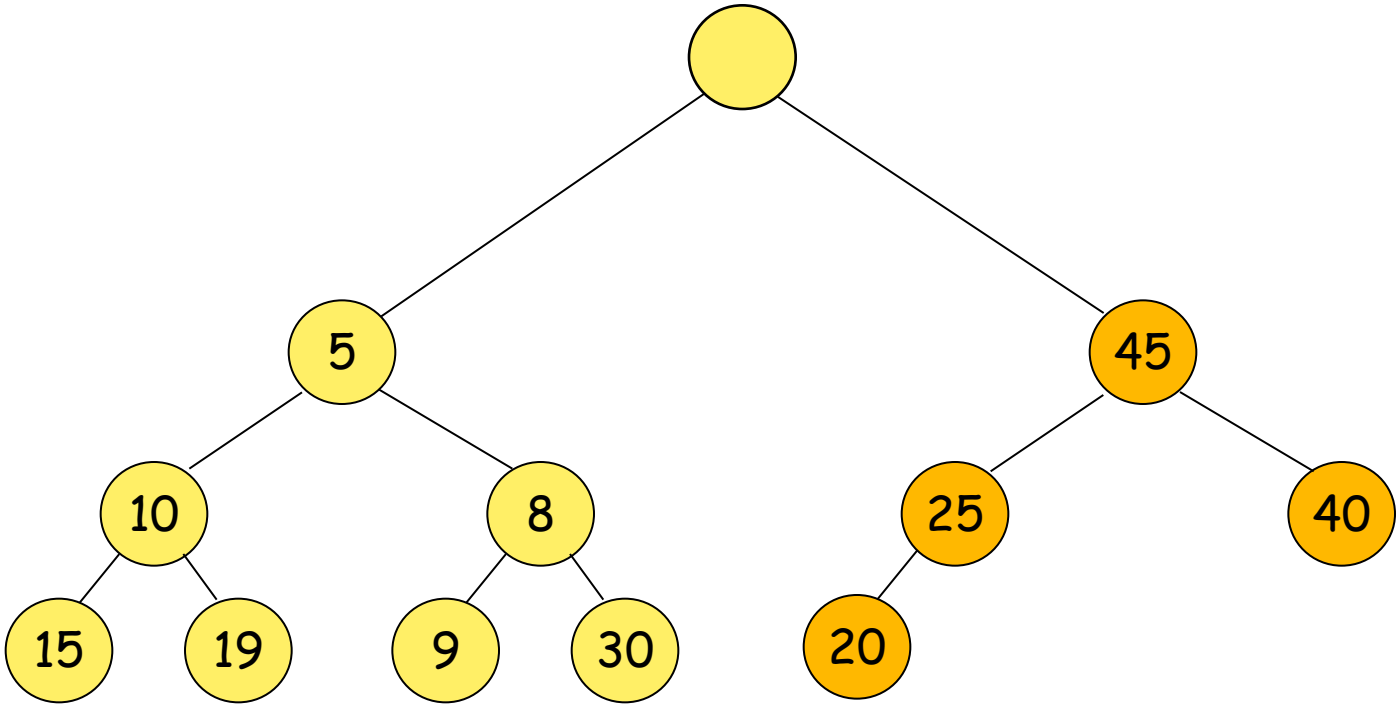
- Για παράδειγμα, έστω ότι πρέπει να **εισαχθεί ένα νέο στοιχείο με κλειδί 2** στην πρώτη δομή του σχήματος. Η αρχική θέση του είναι ως δεξιό παιδί του κόμβου με κλειδί το 71.
- Ο πρώτος έλεγχος γίνεται σε σχέση με το κλειδί του συμμετρικού κόμβου, δηλαδή το 52. Επειδή δεν ισχύει το τέταρτο χαρακτηριστικό του ορισμού, πρέπει να γίνει ανταλλαγή των δύο στοιχείων.
- Στο δεύτερο στάδιο ελέγχεται ο σωρός ελαχίστων, οπότε το κλειδί 2 ανέρχεται μέχρι το δεύτερο επίπεδο.
- Τέλος, στο τρίτο στάδιο ελέγχεται ο σωρός μεγίστων και εκτελούνται οι σχετικές ανταλλαγές αν χρειάζεται. Στη δεύτερη δομή του σχήματος φαίνεται η τελική μορφή της δομής.



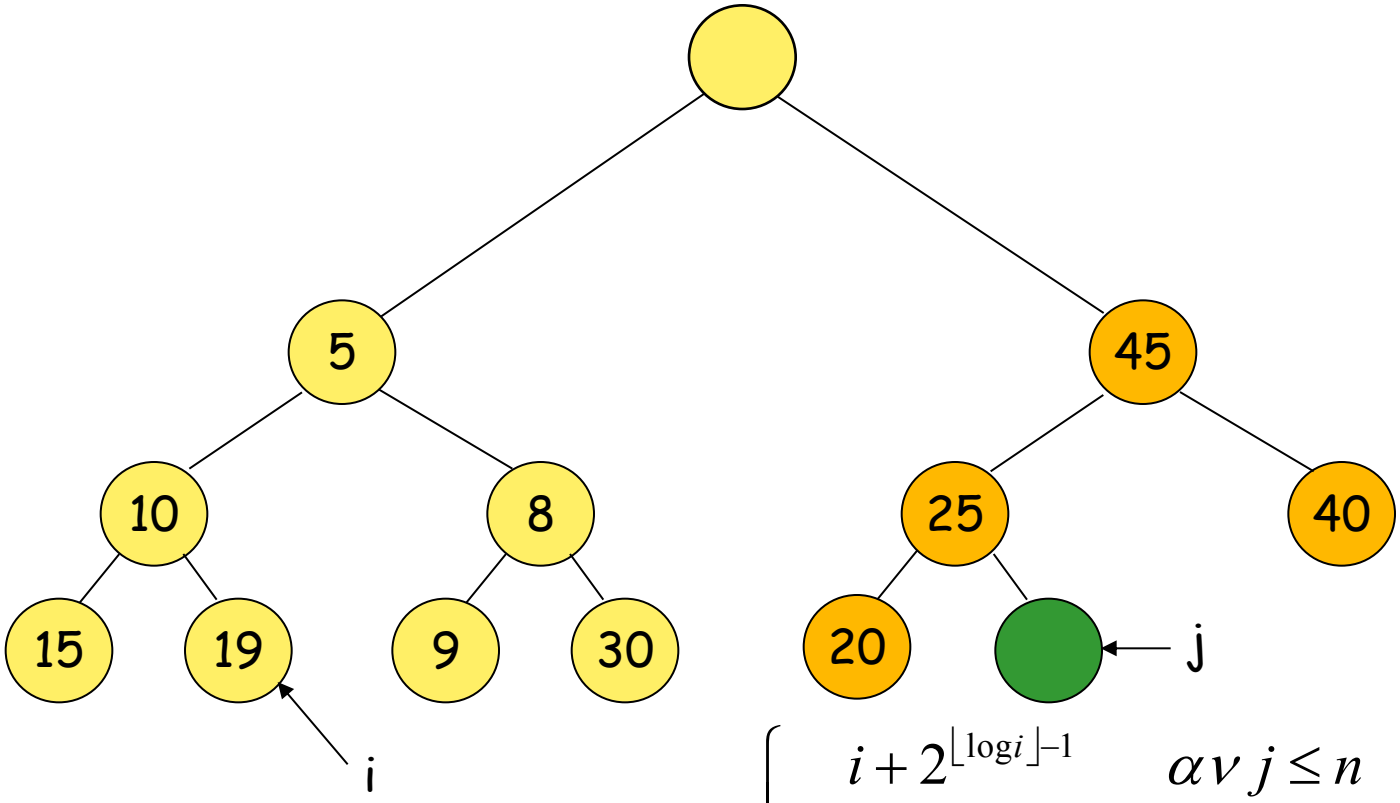


Αν στην πρώτη δομή του σχήματος (α) **εισαχθεί ένα νέο κλειδί με τιμή 90**, τότε θα προκύψει ο τελικός σωρός του σχήματος (γ).

A Deap Example

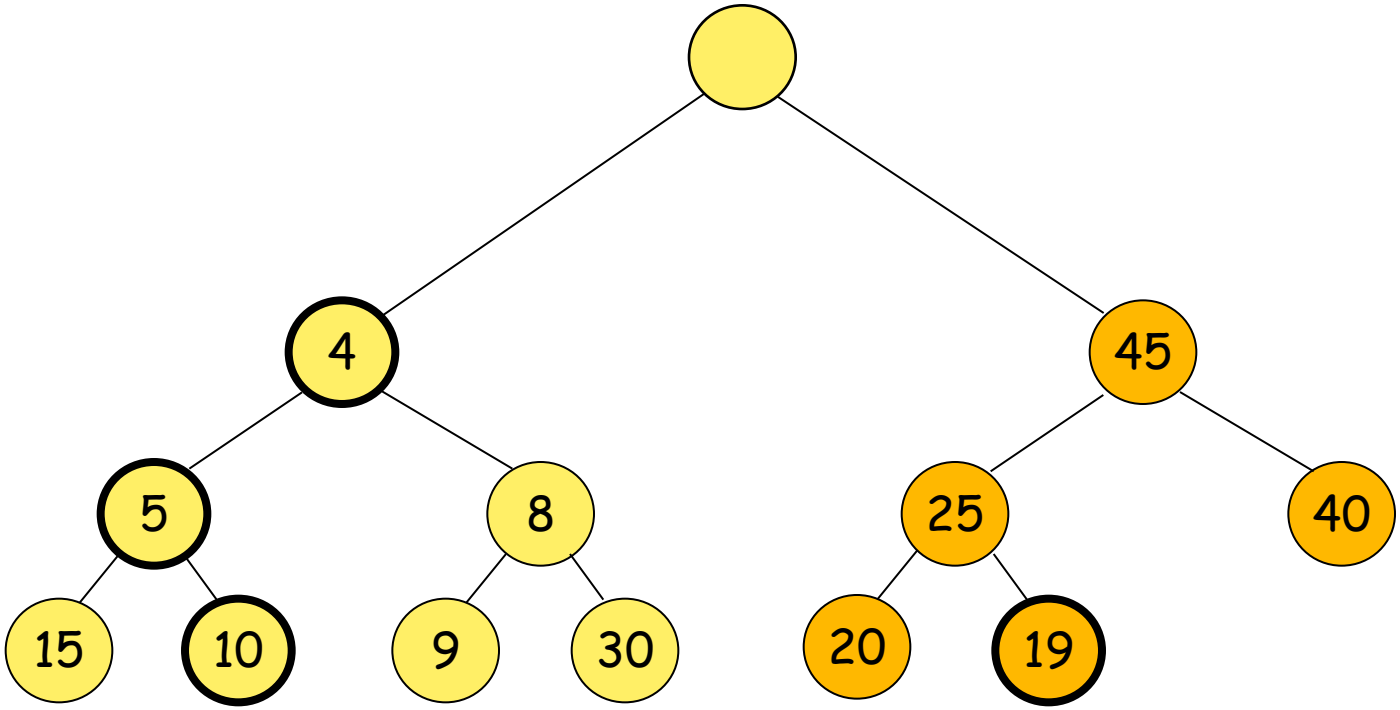


Εισαγωγή σε Deap του 4

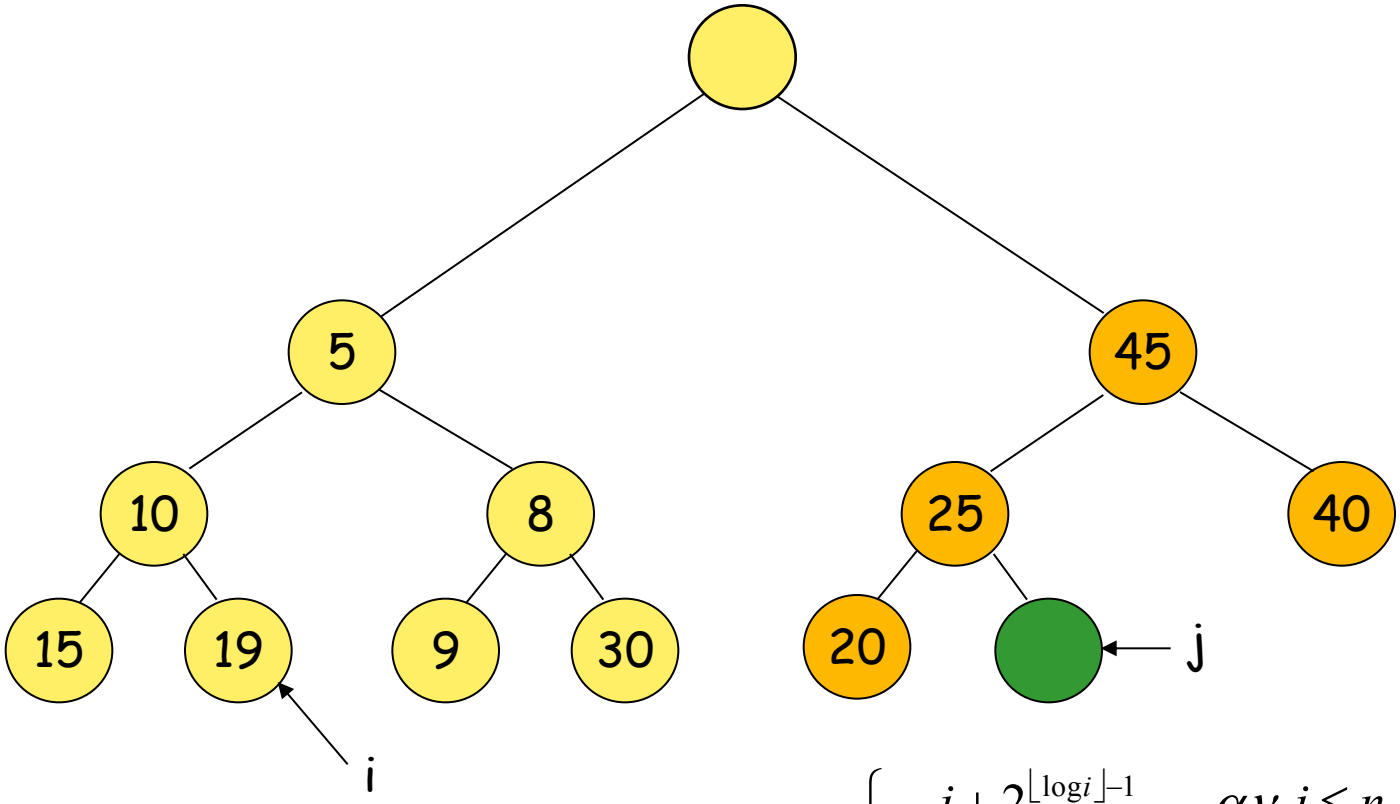


$$j = \begin{cases} i + 2^{\lfloor \log i \rfloor - 1} & \alpha \nu \; j \le n \\ \left\lfloor \frac{1}{2}\left(i + 2^{\lfloor \log i \rfloor - 1}\right) \right\rfloor & \alpha \nu \; j > n \end{cases}$$

Το Deap μετά την εισαγωγή του 4

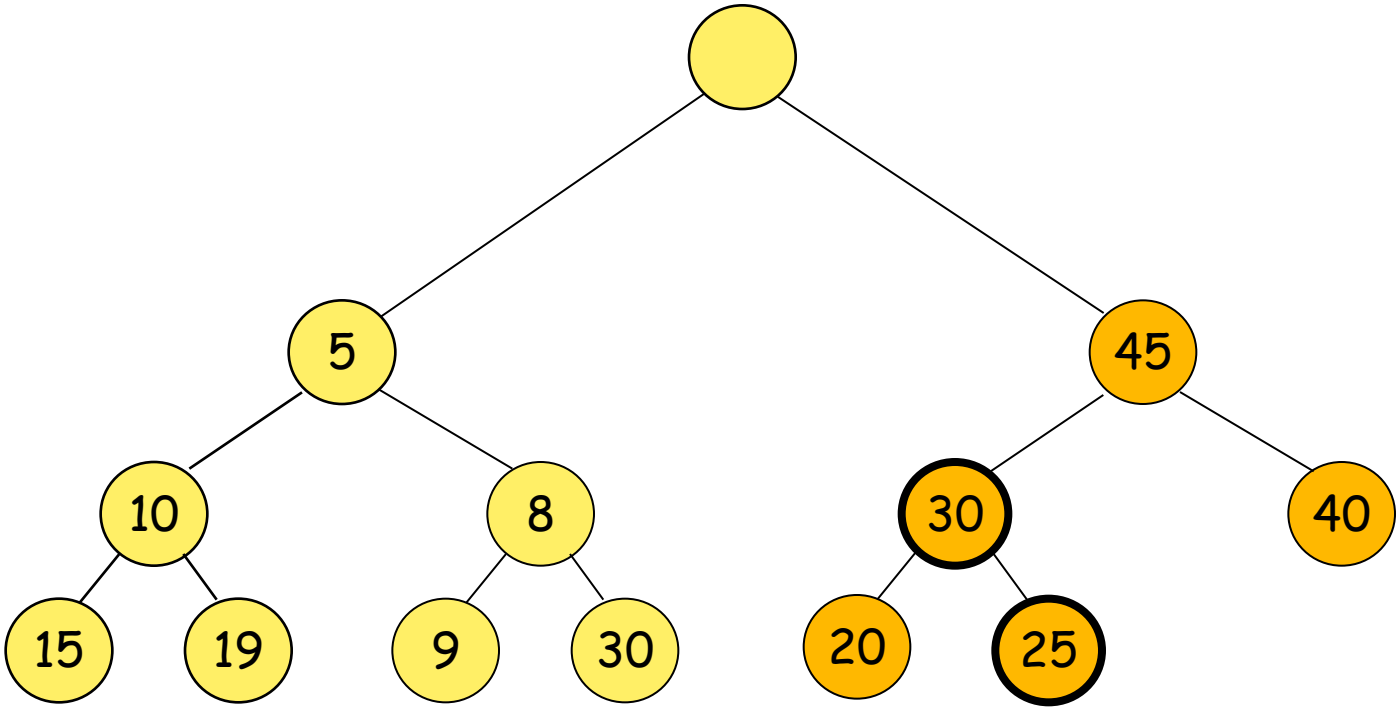


Εισαγωγή σε Deap του 30



$$j = \begin{cases} i + 2^{\lfloor \log i \rfloor - 1} & \alpha \nu \; j \le n \\ \left\lfloor \frac{1}{2}\left(i + 2^{\lfloor \log i \rfloor - 1}\right) \right\rfloor & \alpha \nu \; j > n \end{cases}$$

Το Deap μετά την εισαγωγή του 30



Αλγόριθμος Εισαγωγής σε Deap

- Στη συνέχεια δίνεται η διαδικασία **InsertDeap** που εκτελεί τη σχετική λειτουργία στον πίνακα Deap με *n* ακεραίους. Η διαδικασία αυτή καλεί τα εξής υποπρογράμματα:
- τη συνάρτηση **MaxHeap** που επιστρέφει την τιμή true (false) αν η *n*-οστή θέση βρίσκεται στο σωρό μεγίστων (αντίστοιχα, ελαχίστων), η συνάρτηση **MinPartner** (**MaxPartner**) που υπολογίζει στο σωρό ελαχίστων (αντίστοιχα, μεγίστων) τη θέση του συμμετρικού κόμβου (αντίστοιχα, του πατέρα) του η-οστού κόμβου,
- τις διαδικασίες **MinInsert** και **MaxInsert** που εκτελούν αντίστοιχα την εισαγωγή στο σωρό ελαχίστων και στο σωρό μεγίστων.
- Σημειώνεται ότι οι δυο αυτοί σωροί έχουν τη ρίζα τους στη δεύτερη και στην τρίτη θέση αντίστοιχα

τις διαδικασίες ***MinInsert*** και ***MaxInsert*** που εκτελούν αντίστοιχα την εισαγωγή στο σωρό ελαχίστων και στο σωρό μεγίστων

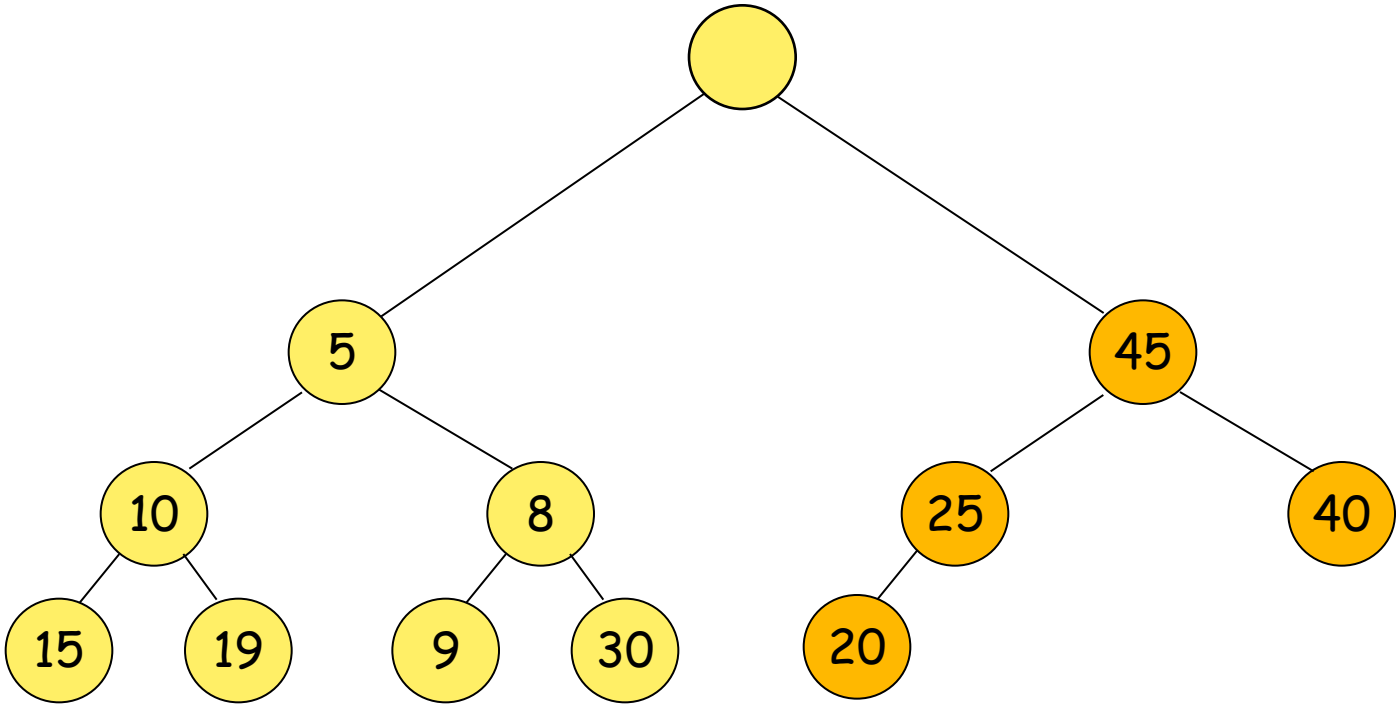
Αλγόριθμος InsertDeap(key);

```
1.   n ← n+1;
2.   if (n=2) Deap[2] ← key;
3.   else
         switch MaxHeap(n)
4.           case true:    //η n-οστή θέση βρίσκεται στο σωρό μεγίστων
5.                   i ← MinPartner(n);
6.                   if (key < Deap[i])
7.                           Deap[n] ← Deap[i]; MinInsert(i,key)
8.                   else MaxInsert(n,key)
9.           case false:
10.                  i ← MaxPartner(n);
11.                  if (key > Deap[i])
12.                          Deap[n] ← Deap[i]; MaxInsert(i,key)
13.                  else MinInsert(n,key)
```

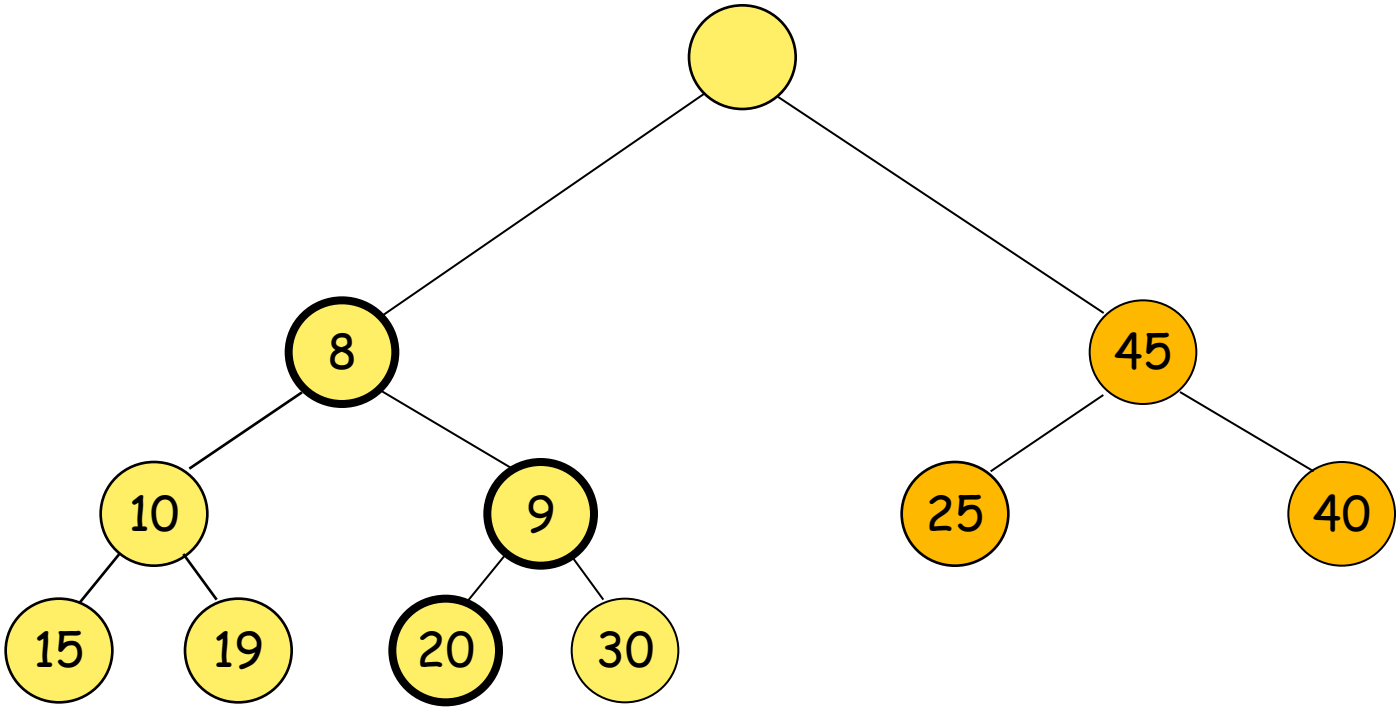
Διαγραφή του Min από το Deap

- Διαγραφή του ελάχιστου στοιχείου από το Deap.
  - Θέτουμε το τελευταίο στοιχείο του deap σε μια προσωρινή θέση t.
  - Η κενή θέση που δημιουργείται από τη διαγραφή του ελάχιστου στοιχείου είναι η θέση 2.
  - Το μικρότερο από τα παιδιά του κάθε φορά τρέχοντος κόμβου μετακινείται προς τα επάνω.
  - Στη συνέχεια μετακινείται στοιχείο στη θέση που μέχρι τώρα κατείχε το μετακινούμενο στοιχείο.
  - Επαναλαμβάνουμε τη διαδικασία μέχρι να προκύψει κενή θέση σε φύλλο.
  - Συγκρίνουμε το κλειδί που τέθηκε στην προσωρινή θέση t με το max (maxpartener) «συμμετρικό κόμβο (αντίστοιχα πατέρα».
  - If <, δεν χρειάζεται ανταλλαγή. Το στοιχείο της προσωρινής θέσης αποθηκεύεται στην κενή θέση – φύλλο.
  - If >, τα ανταλλάσουμε.

Διαγραφή του Min από το Deap



Το Deap μετά τη διαγραφή του Min



Τέλος Ενότητας