ARISTOTLE UNIVERSITY OF THESSALONIKI

SCHOOL OF INFORMATICS

DATA&WEB SCIENCE LABORATORY

ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Ουρές Προτεραιότητας

Απόστολος Ν. Παπαδόπουλος
Αναπληρωτής Καθηγητής
Τμήμα Πληροφορικής Α.Π.Θ.

Βασικές Έννοιες

Η ουρά είναι μία δομή που χρησιμοποιεί τον κανόνα FIFO για την εισαγωγή και διαγραφή στοιχείων.

Η **ουρά προτεραιότητας** (priority queue) είναι ξεχωριστή δομή δεδομένων η οποία επιβάλλει έναν **κανόνα προτεραιότητας**.

Βασικές Έννοιες

Μπορούμε να υποθέσουμε ότι τα στοιχεία έχουν έναν αριθμό που ισοδυναμεί με την προτεραιότητα.

Ανάλογα με την εφαρμογή, ίσως οι μικρές τιμές έχουν μεγαλύτερη προτεραιότητα ή οι μεγάλες τιμές έχουν μεγαλύτερη προτεραιότητα.

Βασικές Έννοιες

Παράδειγμα κλάσης

```
class PriorityQueue {
    int k; // τρέχον πλήθος στοιχείων
    int size; // μέγιστος αριθμός στοιχείων
    …
    FindMin(); // εύρεση μικρότερου στοιχείου
    ExtractMin(); // εξαγωγή μικρότερου στοιχείου
    Insert(x); // εισαγωγή νέου στοιχείου
    BuildHeap(data[ ]); // κατασκευή
    HeapSort(); // ταξινόμηση με σωρό
}
```

Δένδρο-Σωρός

Είναι ένα (σχεδόν) **πλήρες δυαδικό δένδρο** που υποστηρίζει τις ακόλουθες βασικές λειτουργίες:

- Εύρεση ελαχίστου (ή μεγίστου) σε σταθερό χρόνο **O(1)**.
- Εισαγωγή στοιχείου σε χρόνο **O(log *n*)** όπου *n* το πλήθος των στοιχείων του σωρού.
- Διαγραφή ελαχίστου (ή μεγίστου) σε χρόνο **O(log *n*)**.

Να συγκρίνετε τους παραπάνω χρόνους με τους αντίστοιχους του ταξινομημένου πίνακα.

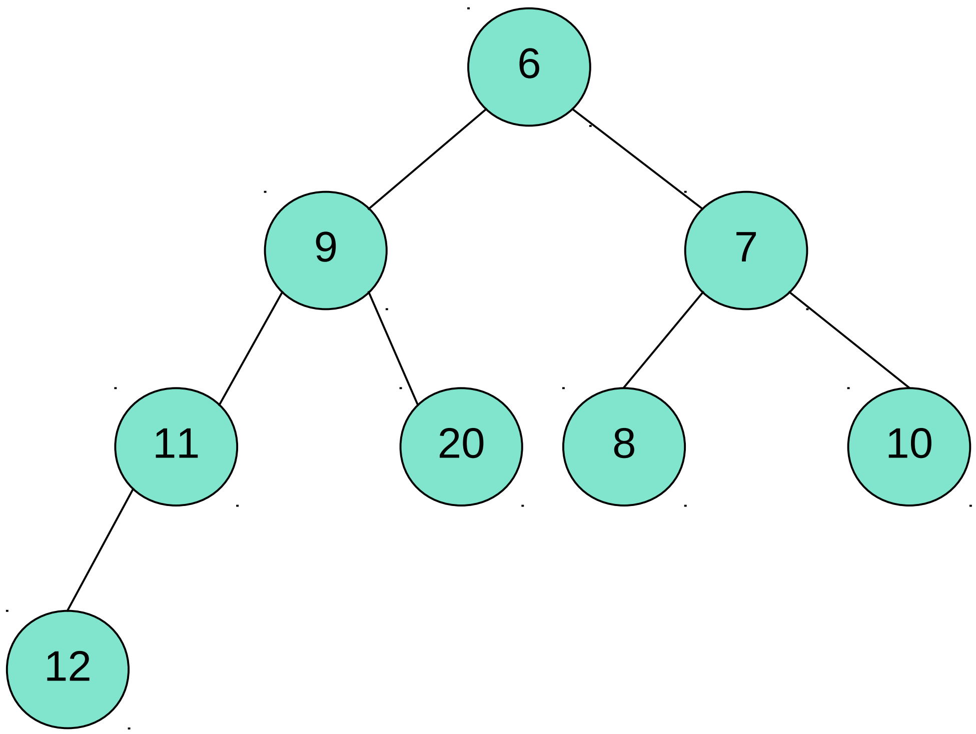
Δένδρο-Σωρός

Σωρός ελαχίστων (minHeap)

Το δένδρο-σωρός μεγαλώνει από πάνω προς τα κάτω και από αριστερά προς τα δεξιά. Μόνο το τελευταίο επίπεδο μπορεί να μην είναι πλήρως συμπληρωμένο.

Το στοιχείο ενός κόμβου είναι **μικρότερο** από τα στοιχεία των παιδιών του.

Το **ελάχιστο** στοιχείο βρίσκεται πάντα στη ρίζα του δένδρου.



Δένδρο-Σωρός

Διαδικασία εισαγωγής

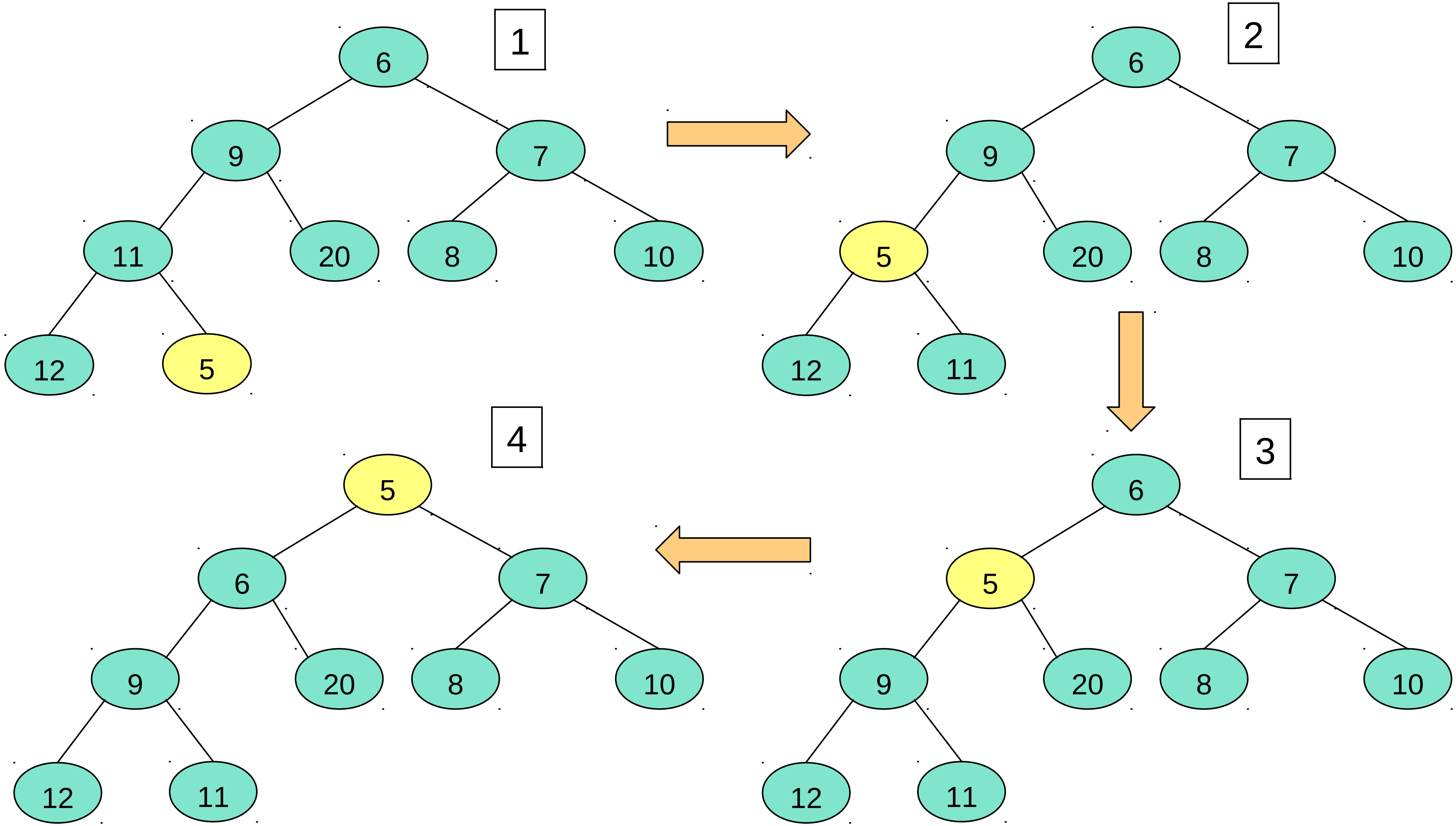
•Το νέο στοιχείο τοποθετείται σε έναν κόμβο στο τέλος του σωρού.

•Στη συνέχεια, λαμβάνουν χώρα αντιμεταθέσεις ώστε το νέο στοιχείο να τοποθετηθεί τελικά στο σωστό επίπεδο.

•Το νέο στοιχείο μπορεί να φτάσει μέχρι τη ρίζα του σωρού.

Δένδρο-Σωρός

Παράδειγμα εισαγωγή του 5



Δένδρο-Σωρός

Διαδικασία διαγραφής ελαχίστου

•Παίρνουμε το τελευταίο στοιχείο του σωρού και το τοποθετούμε στη ρίζα.

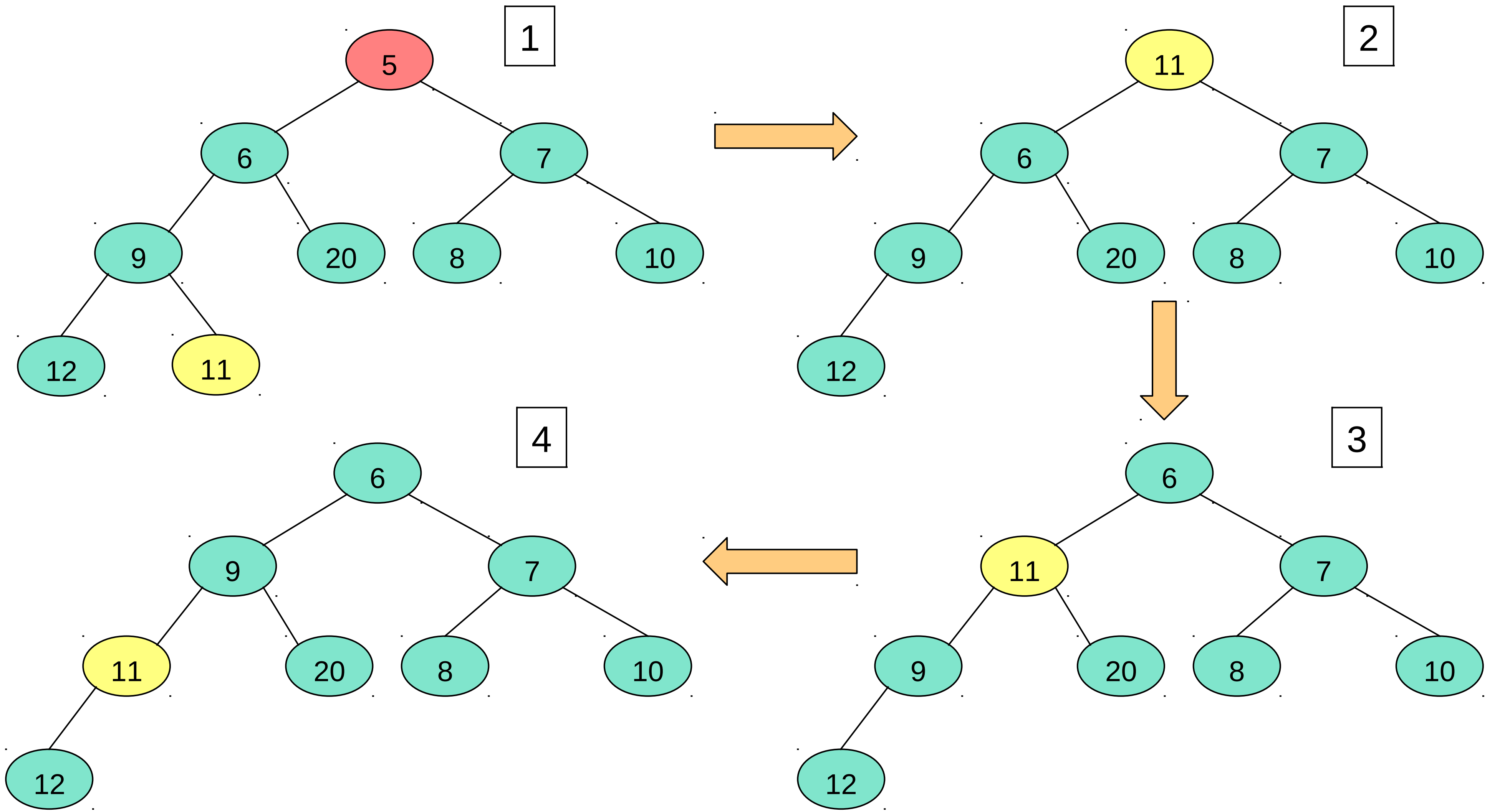
•Στη συνέχεια, λαμβάνουν χώρα αντιμεταθέσεις ώστε το στοιχείο που μπήκε στη ρίζα να τοποθετηθεί τελικά στο σωστό επίπεδο.

•Το στοιχείο που τοποθετήθηκε στη ρίζα μπορεί να φτάσει μέχρι το τελευταίο επίπεδο του σωρού.

•Το μέγεθος (=πλήθος στοιχείων) του σωρού μειώνεται κατά ένα.

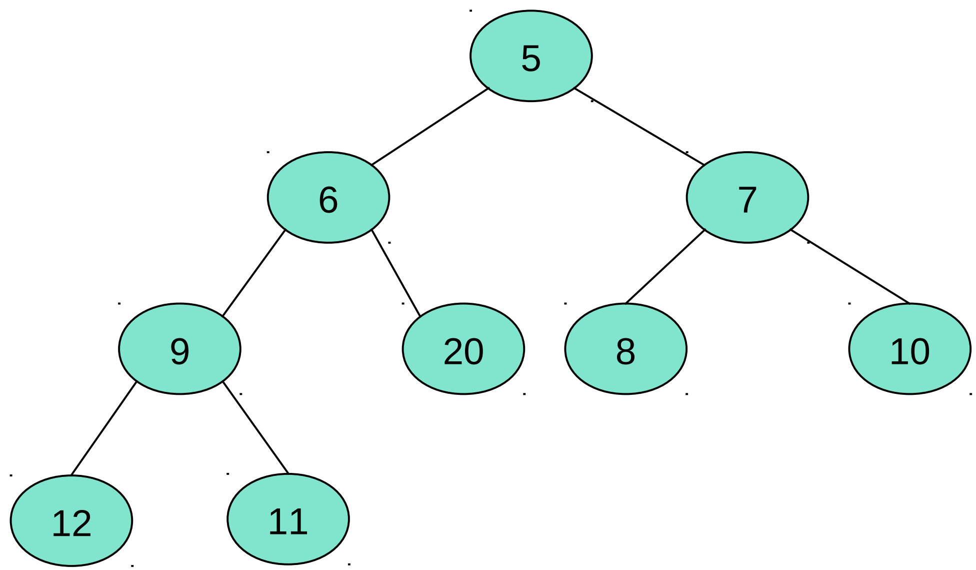
Δένδρο-Σωρός

Παράδειγμα διαγραφής ελαχίστου



Δένδρο-Σωρός

Αποθήκευση σωρού σε πίνακα



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	6	7	9	20	8	10	12	11						

Τα παιδιά του κόμβου που βρίσκεται στη θέση *i* του πίνακα βρίσκονται στις θέσεις 2*i*+1 και 2*i*+2. Ο πρόγονος του κόμβου που βρίσκεται στη θέση *i* του πίνακα βρίσκεται στη θέση (*i*-1)/2

Δένδρο-Σωρός

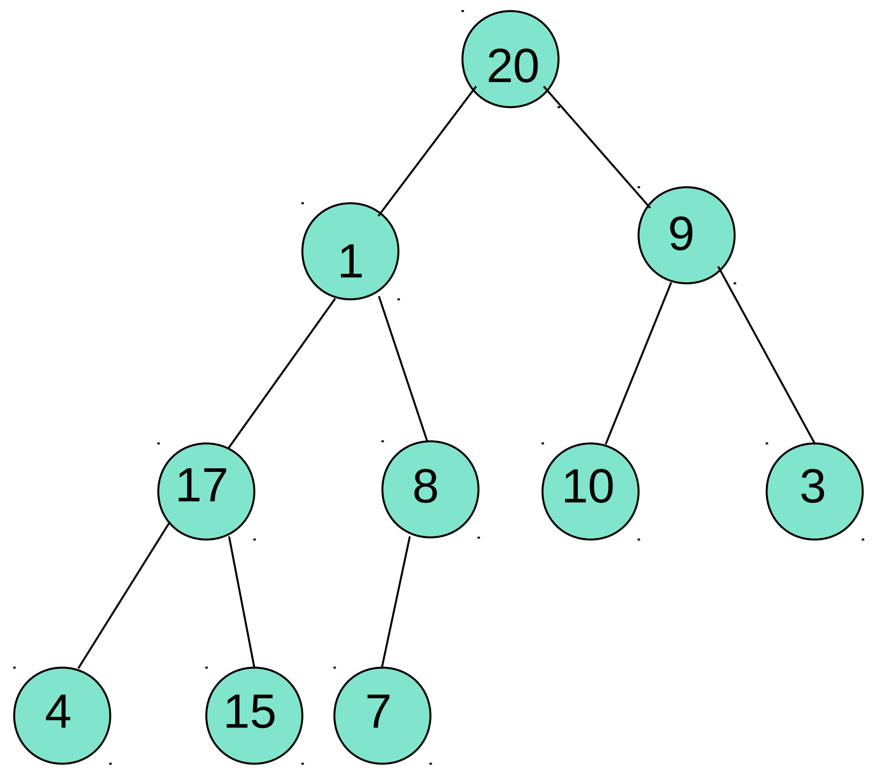
Κατασκευή

- Ο πιο προφανής τρόπος κατασκευής ενός δένδρου-σωρού είναι να εισάγουμε τα στοιχεία ένα προς ένα. Εύκολα αποδεικνύεται ότι το κόστος της μεθόδου σε αριθμό συγκρίσεων είναι $O(n \log n)$.

- Ένας πιο αποδοτικός τρόπος είναι να προχωρήσουμε στην κατασκευή «από κάτω προς τα πάνω» (bottom-up). Με τη μέθοδο αυτή, το κόστος πέφτει σε $O(n)$!

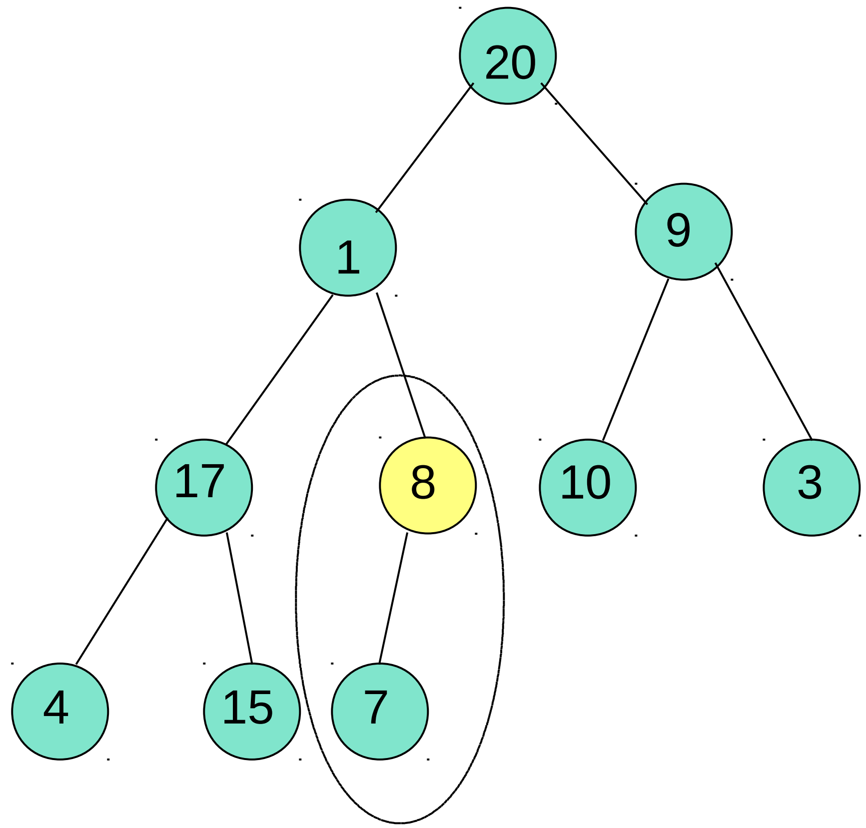
Κατασκευή Σωρού

Κατασκευή bottom-up για τα στοιχεία
20   1 9 17 8 10 3 4 15 7



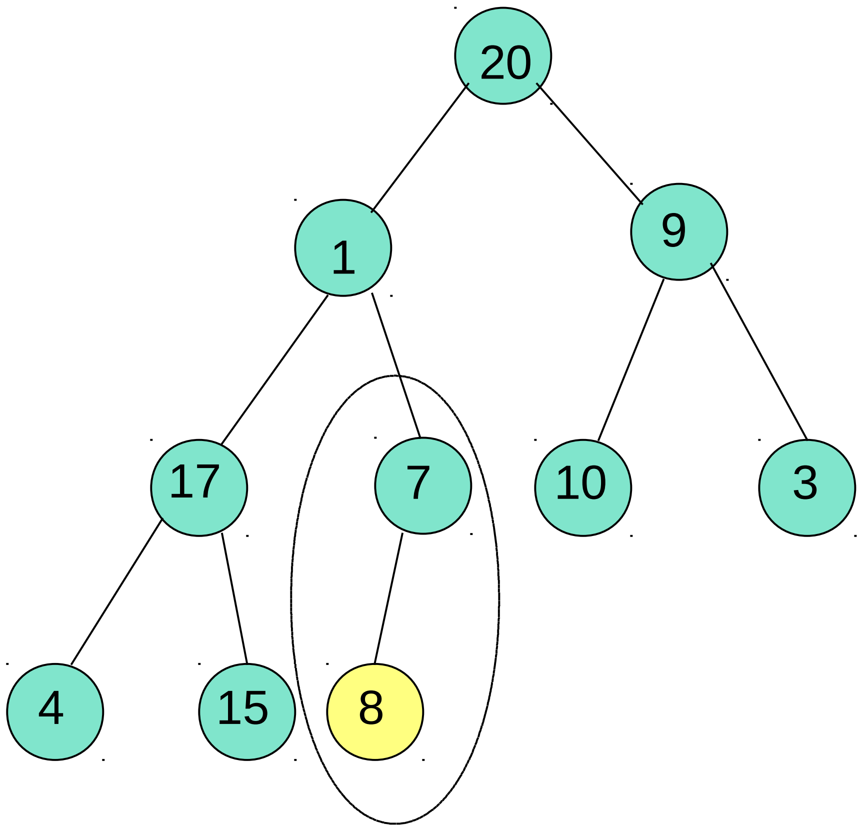
Κατασκευή Σωρού

κατασκευή bottom-up



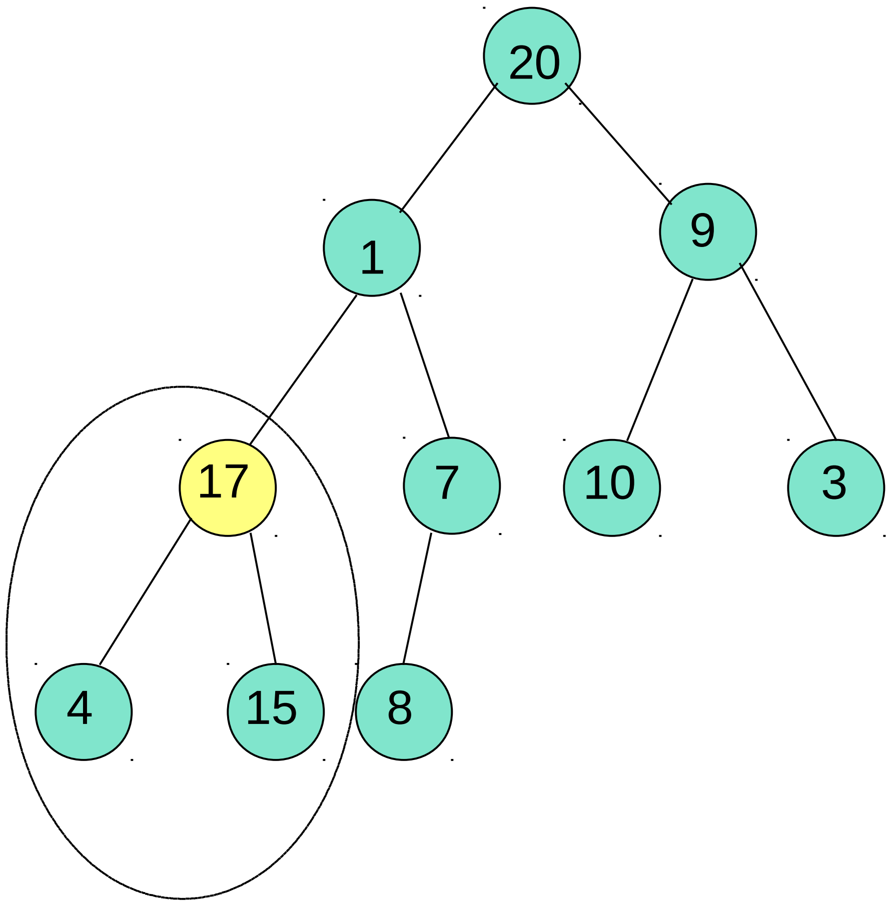
Κατασκευή Σωρού

κατασκευή bottom-up



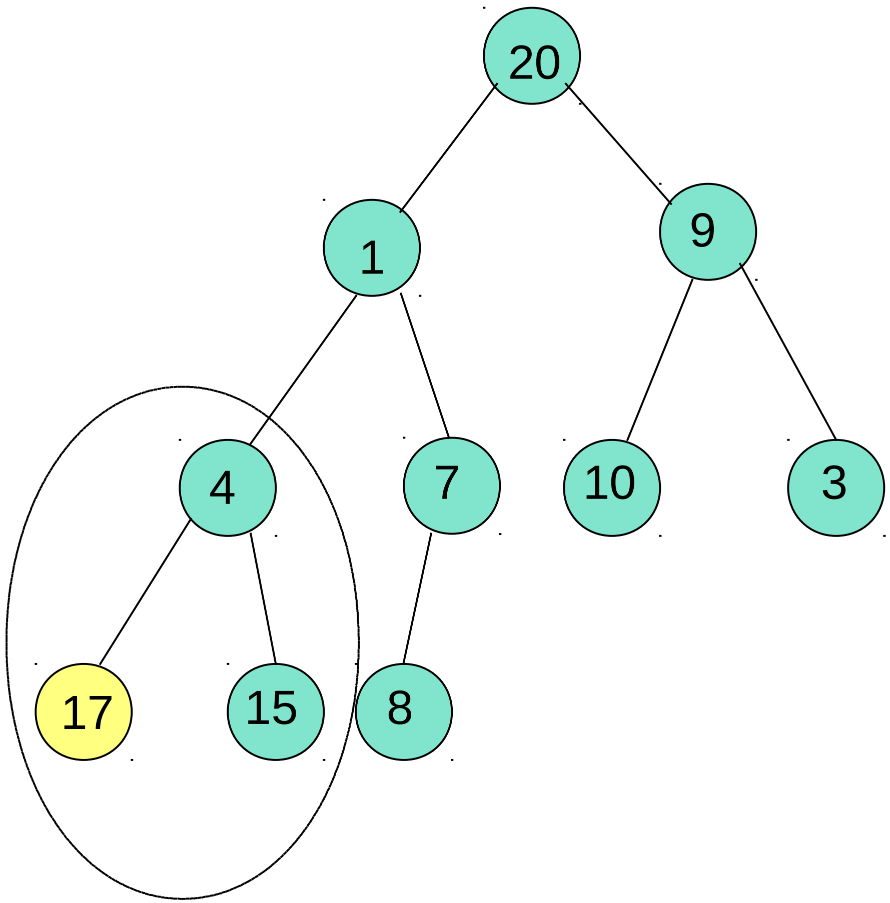
Κατασκευή Σωρού

κατασκευή bottom-up



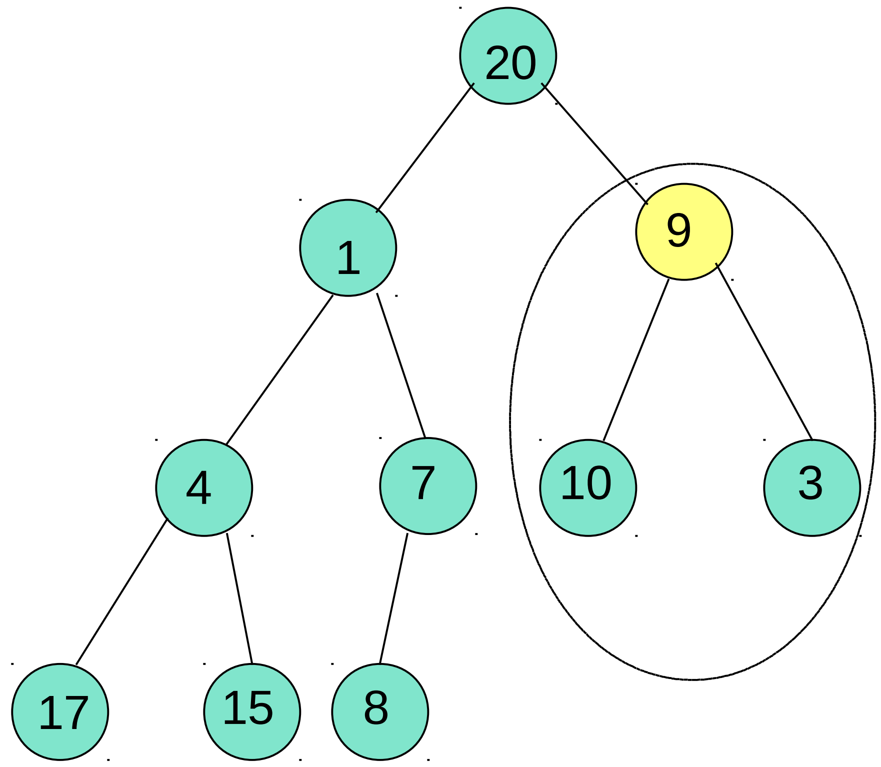
Κατασκευή Σωρού

κατασκευή bottom-up



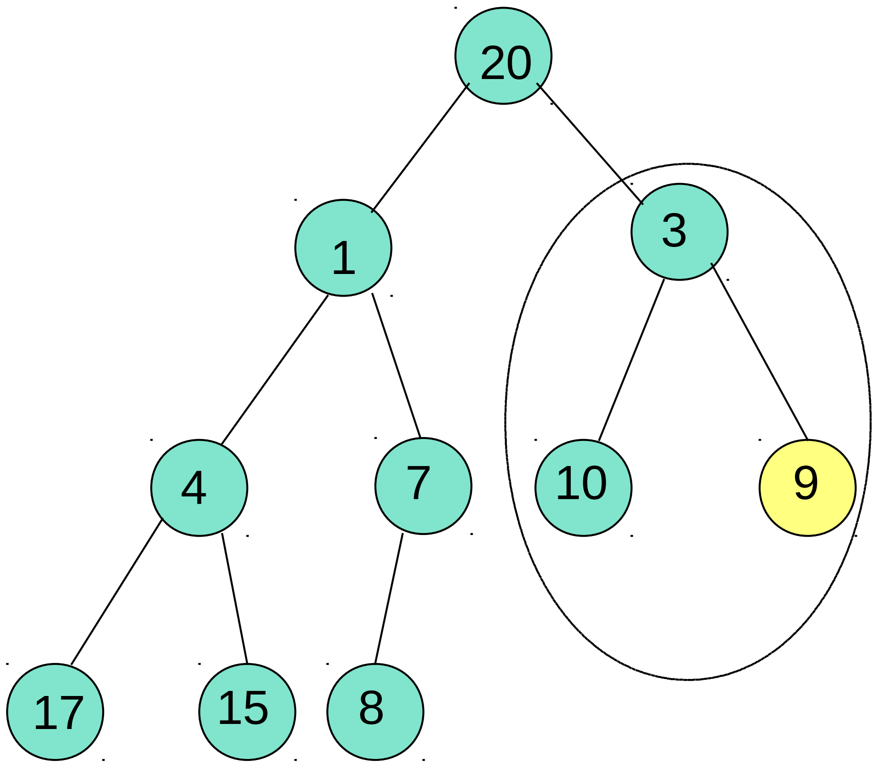
Κατασκευή Σωρού

κατασκευή bottom-up



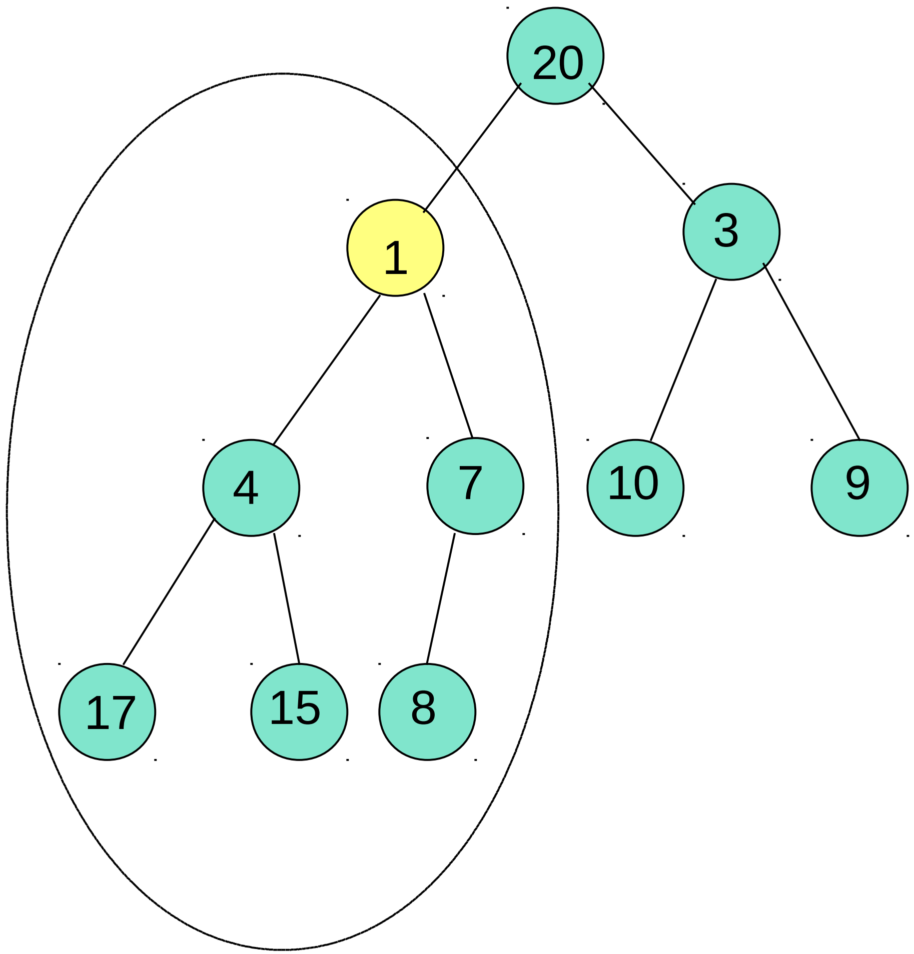
Κατασκευή Σωρού

κατασκευή bottom-up



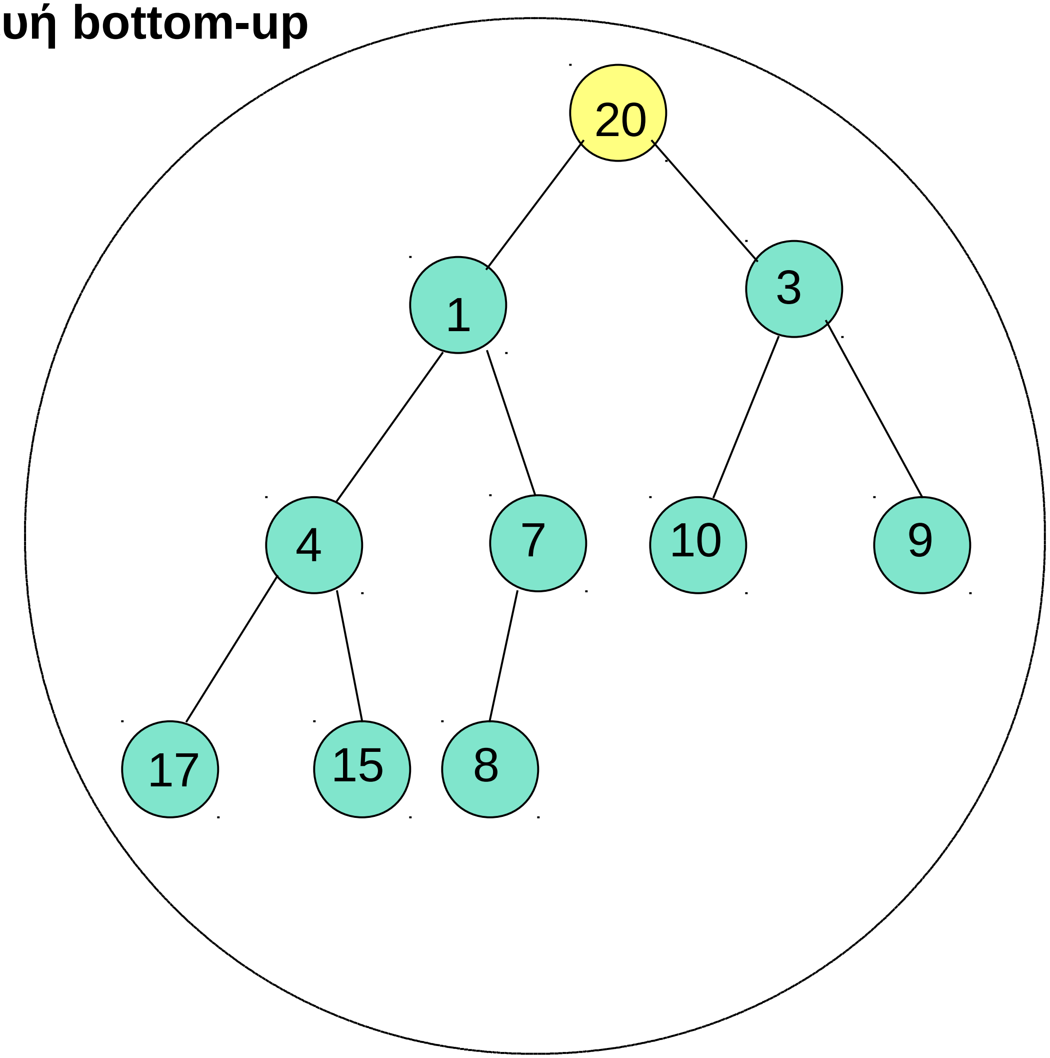
Κατασκευή Σωρού

κατασκευή bottom-up



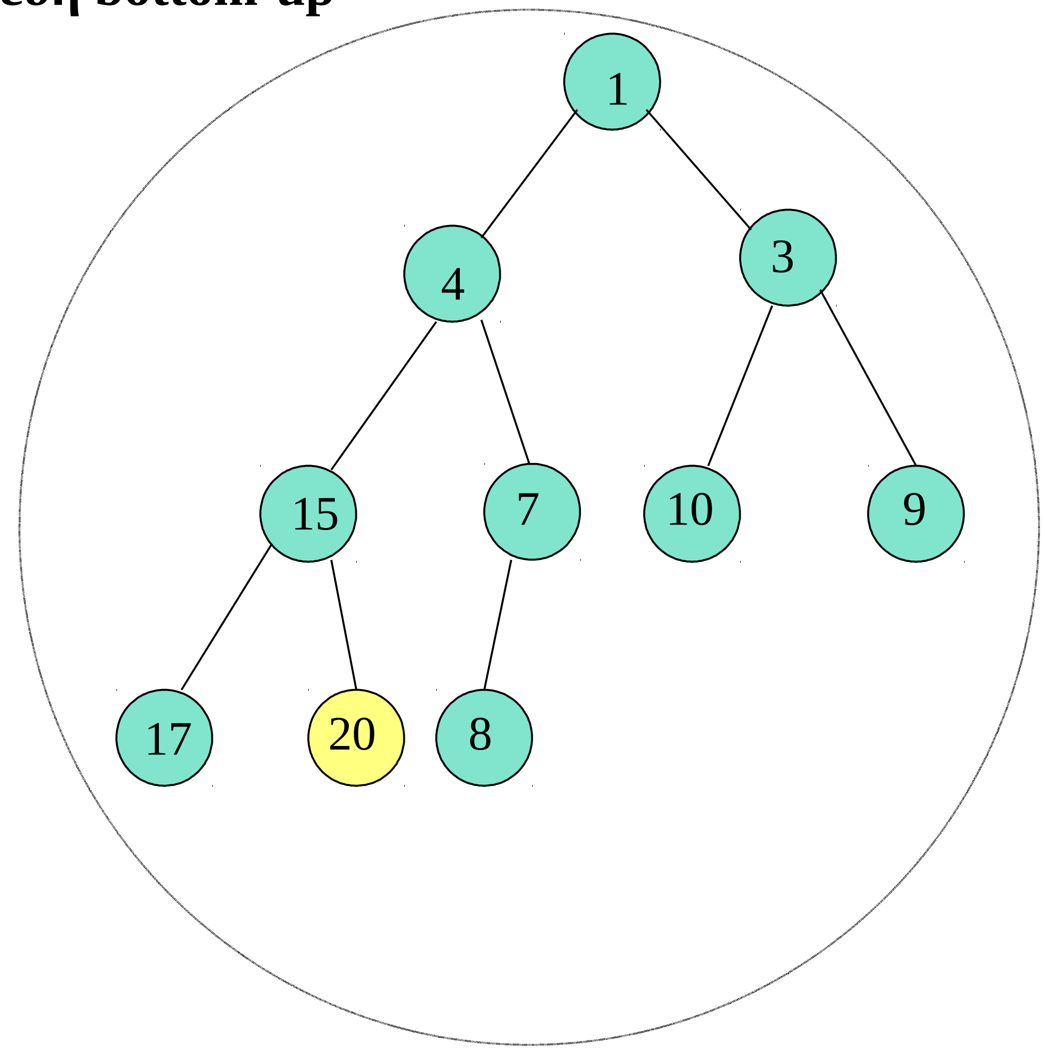
Κατασκευή Σωρού

κατασκευή bottom-up



Κατασκευή Σωρού

κατασκευή bottom-up



Κατασκευή Σωρού

Ανάλυση κατασκευής bottom-up

- Παρατηρούμε ότι αριθμώντας τα επίπεδα του δένδρου από το 1 και από από κάτω προς τα πάνω τότε στο επίπεδο k υπάρχουν το πολύ $n/2^k$ κόμβοι.
- Επόμένως κατά την κατασκευή έχουμε **το πολύ**
  - $n/2^2$ στοιχεία που μετακινούνται **το πολύ** 1 θέση κάτω
  - $n/2^3$ στοιχεία που μετακινούνται **το πολύ** 2 θέσεις κάτω
  - $n/2^4$ στοιχεία που μετακινούνται **το πολύ** 3 θέσεις κάτω
  - … …
- $\text{cost}(n) = O(1*n/2^2 + 2*n/2^3 + 3*n/2^4 + \ldots)$

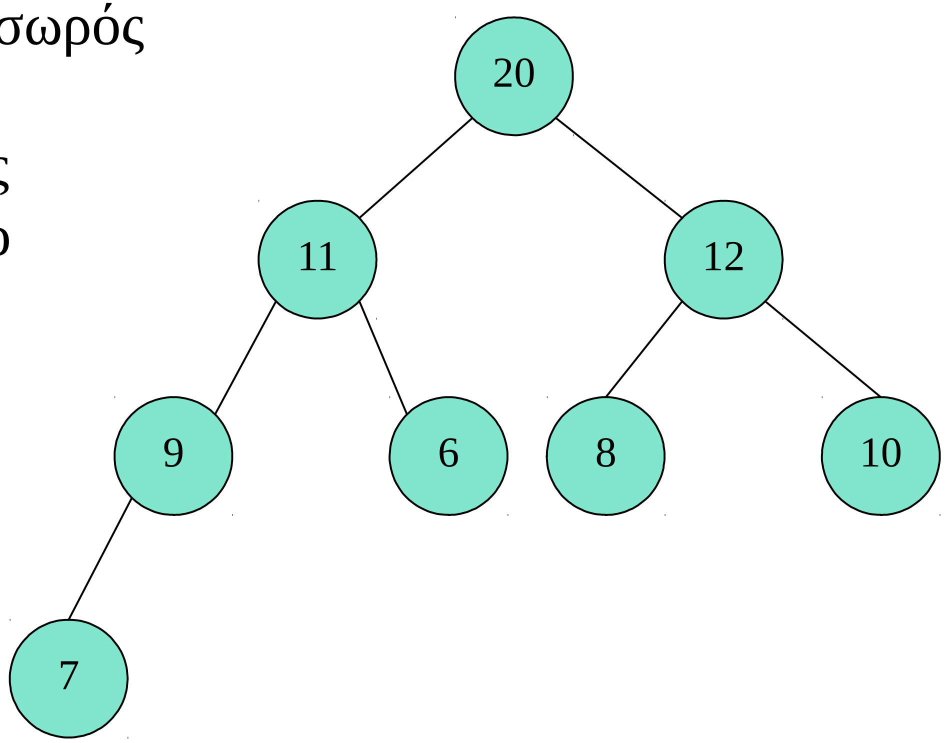
  $= O((n/2)(1/2 + 2/2^2 + 3/2^3 + 4/2^4 + \ldots)$

  $= \mathbf{O(}\boldsymbol{n}\mathbf{)}$

Σωρός Μεγίστων

Σωρός μεγίστων (maxHeap)

Όπως και στο **minHeap**, το δένδρο-σωρός **maxHeap** μεγαλώνει από πάνω προς τα κάτω και από αριστερά προς τα δεξιά. Μόνο το τελευταίο επίπεδο μπορεί να μην είναι πλήρως συμπληρωμένο.



Το στοιχείο ενός κόμβου είναι μεγαλύτερο από τα στοιχεία των παιδιών του.

Το μέγιστο στοιχείο βρίσκεται πάντα στη ρίζα του δένδρου.

Εφαρμογές

Εκτέλεση εργασιών με προτεραιότητες

Χρήση σε άλλους αλγορίθμους (Dijkstra, Prim, A*)

Ταξινόμηση με σωρό (HeapSort)

Κωδικοποίηση Huffman

Εύρεση top-k στοιχείων

Θα δούμε και άλλες εφαρμογές σε επόμενες διαλέξεις.

HeapSort

Βασικά βήματα HeapSort

(1) Κατασκευάζουμε **δένδρο-σωρό** από τα στοιχεία του πίνακα με διαδοχικές εισαγωγές των στοιχείων του. Ο σωρός που δημιουργείται σταδιακά ενσωματώνεται στον πίνακα από την αρχή προς το τέλος του.

(2) Εκτελούμε διαδοχικές **διαγραφές** της ρίζας του σωρού που κατασκευάστηκε στο βήμα (1) μέχρι να καταλήξουμε στο κενό δέντρο, αποθηκεύοντας τα διαγραφόμενα στοιχεία από το τέλος του πίνακα προς τη αρχή του.

(3) Ο τελικός πίνακας είναι **διατεταγμένος**, σε αύξουσα σειρά από την αρχή προς το τέλος του.

HeapSort

Ψευδοκώδικας της HeapSort

```
HeapSort(A) {

    BuildHeap(A);

    for (i = length(A) downto 2) {
        Swap(A[1], A[i]);
        heap_size(A) -= 1;
        Heapify(A, 1);
    }

}
```

HeapSort

BuildHeap() απαιτεί O(n) κόστος

Κάθε μία από τις n-1 κλήσεις της **Heapify()** απαιτεί O(log n) χρόνο

Συνολικά έχουμε για την **HeapSort()**
= O(n) + (n - 1) O(log n)
= O(n) + O(n log n)
= O(n log n)

Παράδειγμα HeapSort

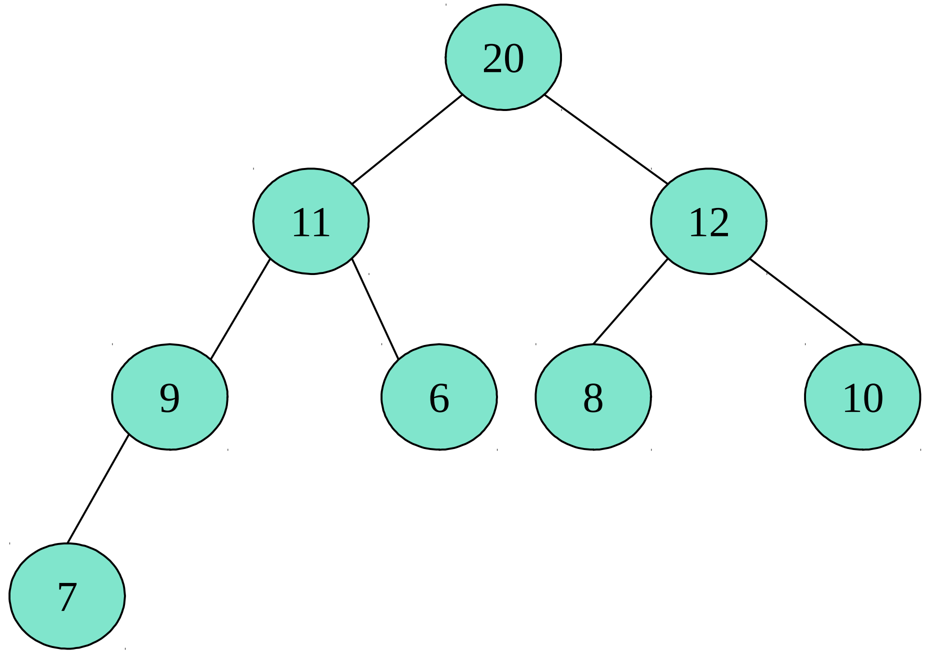
Έστω ο πίνακας ακεραίων:

20 11 12 9 6 8 10 7

Θέλουμε να ταξινομήσουμε τον πίνακα σε αύξουσα διάταξη.

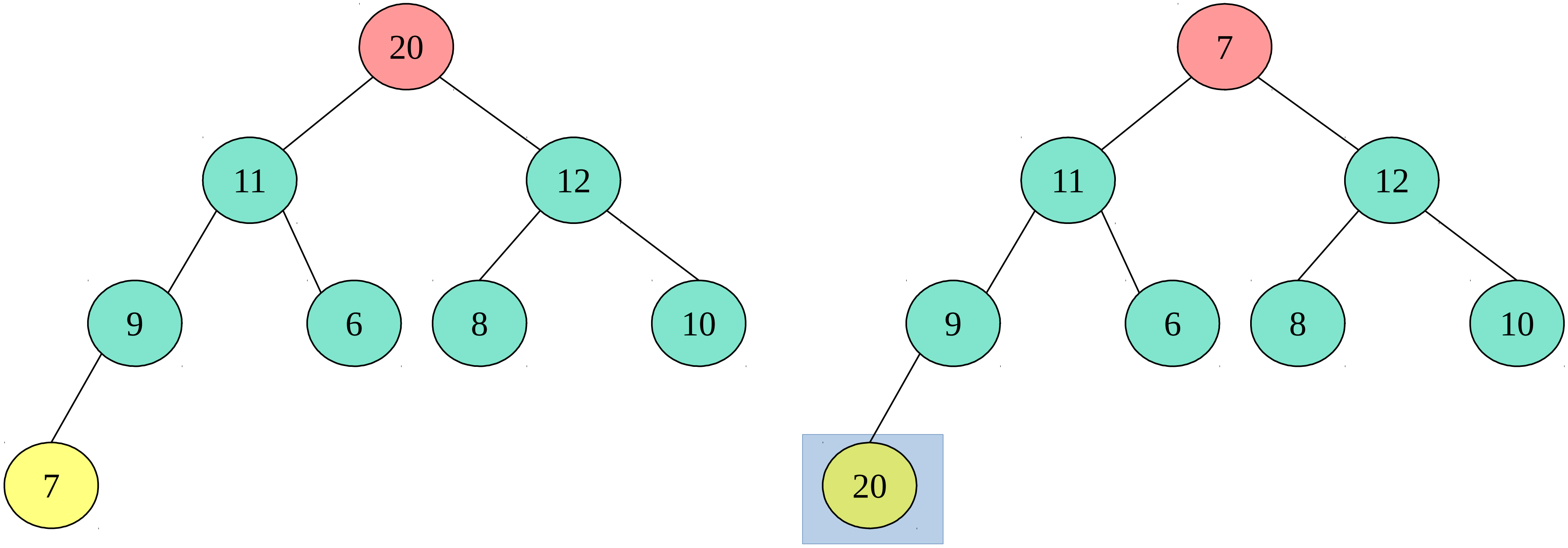
Παράδειγμα HeapSort

Εκτελούμε τη συνάρτηση **BuildHeap** για να κατασκευάσουμε ένα σωρό μεγίστων (το μεγαλύτερο στοιχείο είναι στην κορυφή του σωρού).



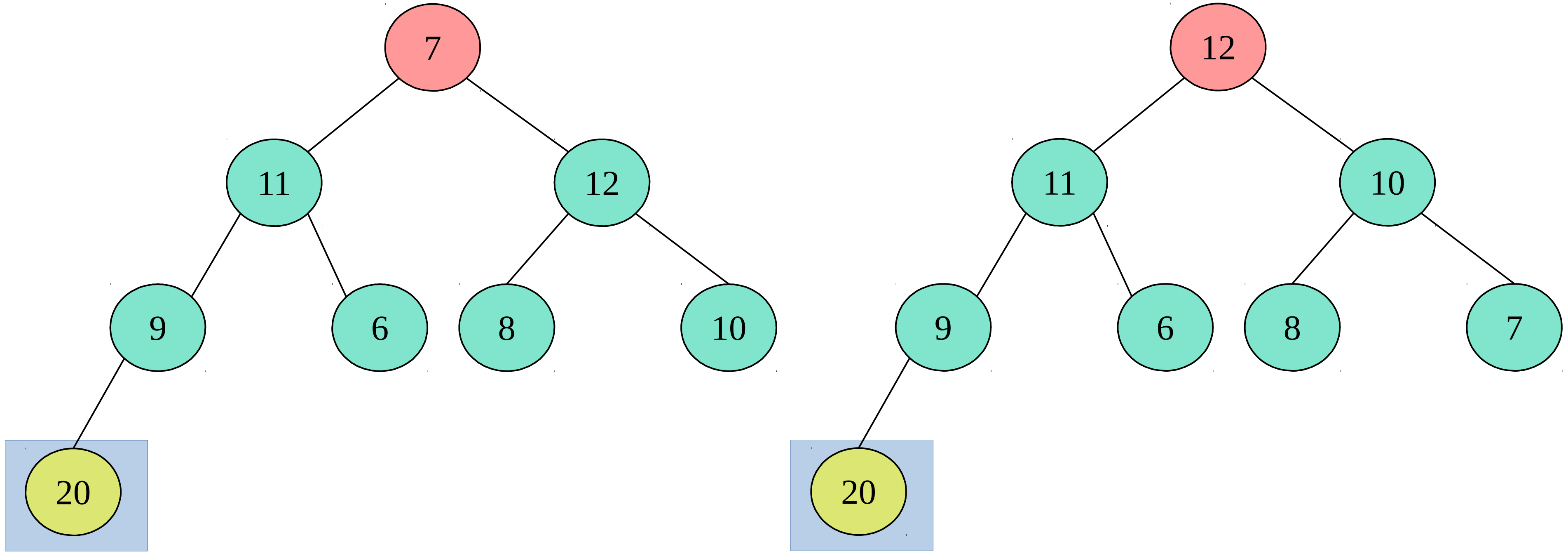
Παράδειγμα HeapSort

Στη συνέχεια εκτελείται η **Heapify** αφού αντιμεταθέσουμε το στοιχείο της κορυφής με το τελευταίο στοιχείο του σωρού.



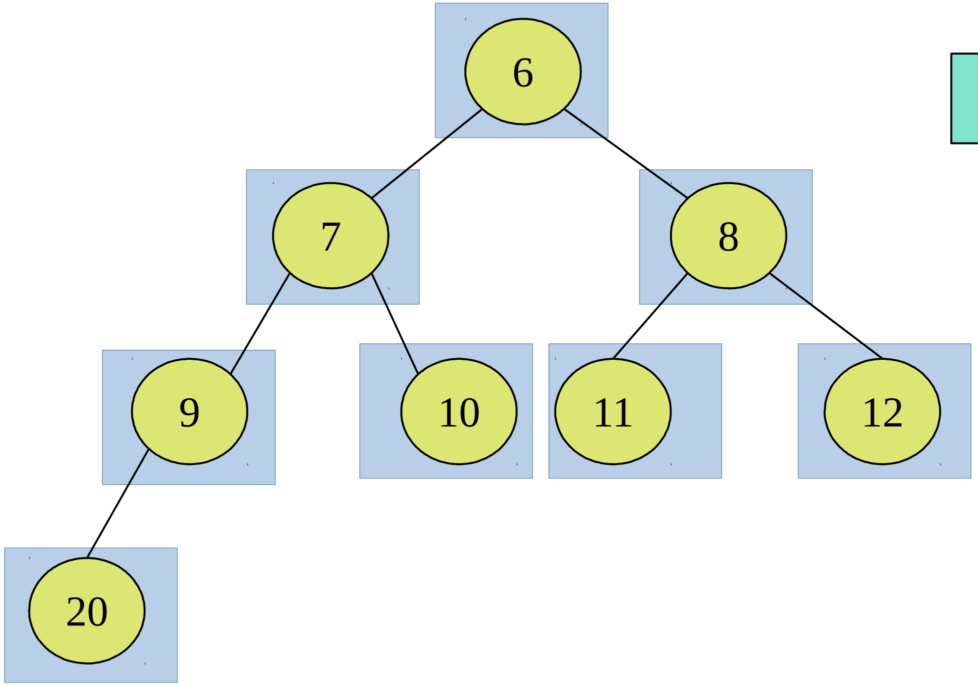
Παράδειγμα HeapSort

Μετά την εκτέλεση της **Heapify** έχουμε:



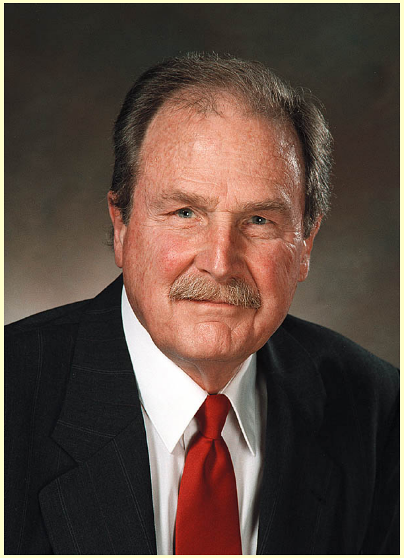
Παράδειγμα HeapSort

Αν ακολουθήσουμε την ίδια διαδικασία τότε στον πίνακα που αποθηκεύεται ο σωρός τα στοιχεία θα είναι σε **αύξουσα** διάταξη.



0	1	2	3	4	5	6	7	8
6	7	8	9	10	11	12	20	

Κωδικοποίηση Huffman



David Albert Huffman was an American **pioneer** in **computer science**, known for his **Huffman coding**. He was also one of the pioneers in the field of **mathematical origami**.

- Born: August 9, 1925, Ohio, United States
- Died: October 7, 1999, Santa Cruz, California, United States
- Thesis: The Synthesis of Sequential Switching Circuits (1953)
- Known for: Huffman coding
- Doctoral advisor: Samuel H. Caldwell
- Awards: IEEE Richard W. Hamming Medal, IEEE Computer Society Awards - W. Wallace McDowell Award

(*Πηγή: www*)

Κωδικοποίηση Huffman

Τι είναι;

Μέθοδος συμπίεσης (κωδικοποίησης) συμβόλων με βάση τη συχνότητα εμφάνισης.

Για παράδειγμα χρησιμοποιείται στο τελικό βήμα για τη συμπίεση των εικόνων τύπου JPG.

Κωδικοποίηση Huffman

Βασικά Σημεία:

Ο κύριος στόχος ενός κωδικοποιητή είναι η αντιστοίχιση **μικρών κωδικών σε συχνά εμφανιζόμενα σύμβολα** και μεγάλων κωδικών σε σπάνια εμφανιζόμενα σύμβολα.

Ο χρόνος κωδικοποίησης και αποκωδικοποίησης είναι σημαντικός. Μερικές φορές προτιμούμε να έχουμε μικρότερο λόγο συμπίεσης προκειμένου να κερδίσουμε σε χρόνο.

Κωδικοποίηση Huffman

Έστω τα σύμβολα A,B,C,D με τους εξής κωδικούς:

Code('A') = 0
Code('B') = 000
Code('C') = 11
Code('D') = 1

DDDAAA

DCB

CDAAA

DDDB

Ο κωδικός 111000 σε ποια σειρά χαρακτήρων αντιστοιχεί;

Κωδικοποίηση Huffman

Βασική προϋπόθεση:

Μετά τη φάση της κωδικοποίησης κανένας κωδικός δεν πρέπει να αποτελεί **πρόθεμα** (prefix) άλλου κωδικού.

Κωδικοποίηση Huffman

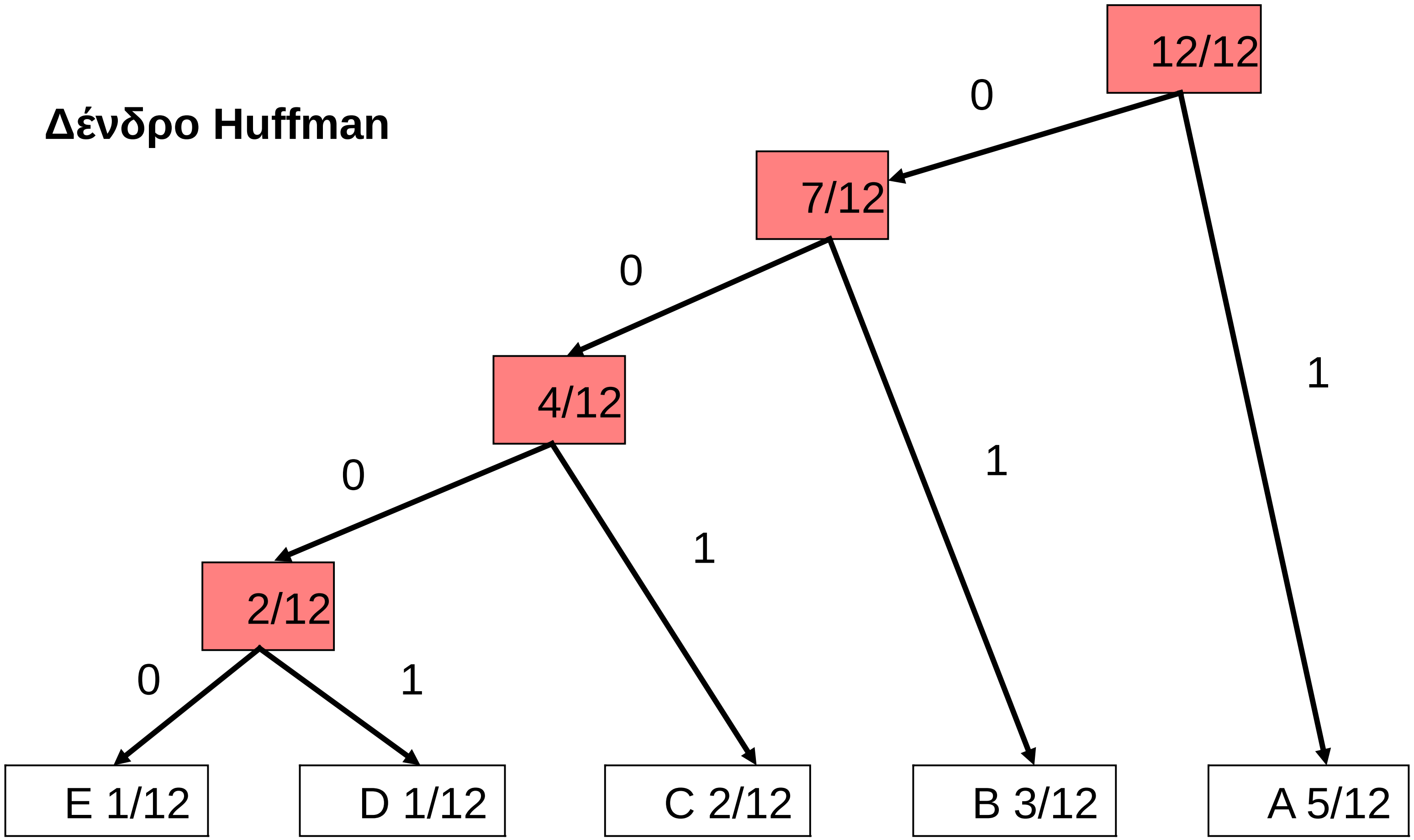
Έστω το ακόλουθο κείμενο:
A B C A B A A A B C D E

A: 5/12
B: 3/12
C: 2/12
D: 1/12
E: 1/12

Συχνότητες εμφάνισης

Κωδικοποίηση Huffman

Δένδρο Huffman



Κωδικοποίηση Huffman

Πως κατασκευάζεται το δένδρο Huffman;

Κάθε φορά συνδυάζουμε τις **<u>δύο μικρότερες</u>** συχνότητες εμφάνισης, δημιουργώντας έναν κόμβο με δύο κλάδους. Βάζουμε 0 στον αριστερό κλάδο και 1 στον δεξιό (σύμβαση, μπορούμε να ανάποδα).

Κωδικοποίηση Huffman

Ο κώδικας ενός συμβόλου παράγεται διασχίζοντας το δένδρο από τη ρίζα μέχρι να βρούμε το σύμβολο και κατραγράφοντας τα bits που συναντούμε στην πορεία.

code(E): 0000

code(D): 0001

code(C): 001

code(B): 01

code(A): 1

Τι παρατηρούμε;

Κανένας κωδικός δεν είναι πρόθεμα άλλου.

Κωδικοποίηση Huffman

Τι συμπίεση επιτυγχάνουμε για το παράδειγμα;

Απαιτούνται 12*8 = 96 bits για το αρχικό κείμενο (χωρίς τους κενούς χαρακτήρες).

Απαιτούνται 25 bits για το συμπιεσμένο κείμενο.

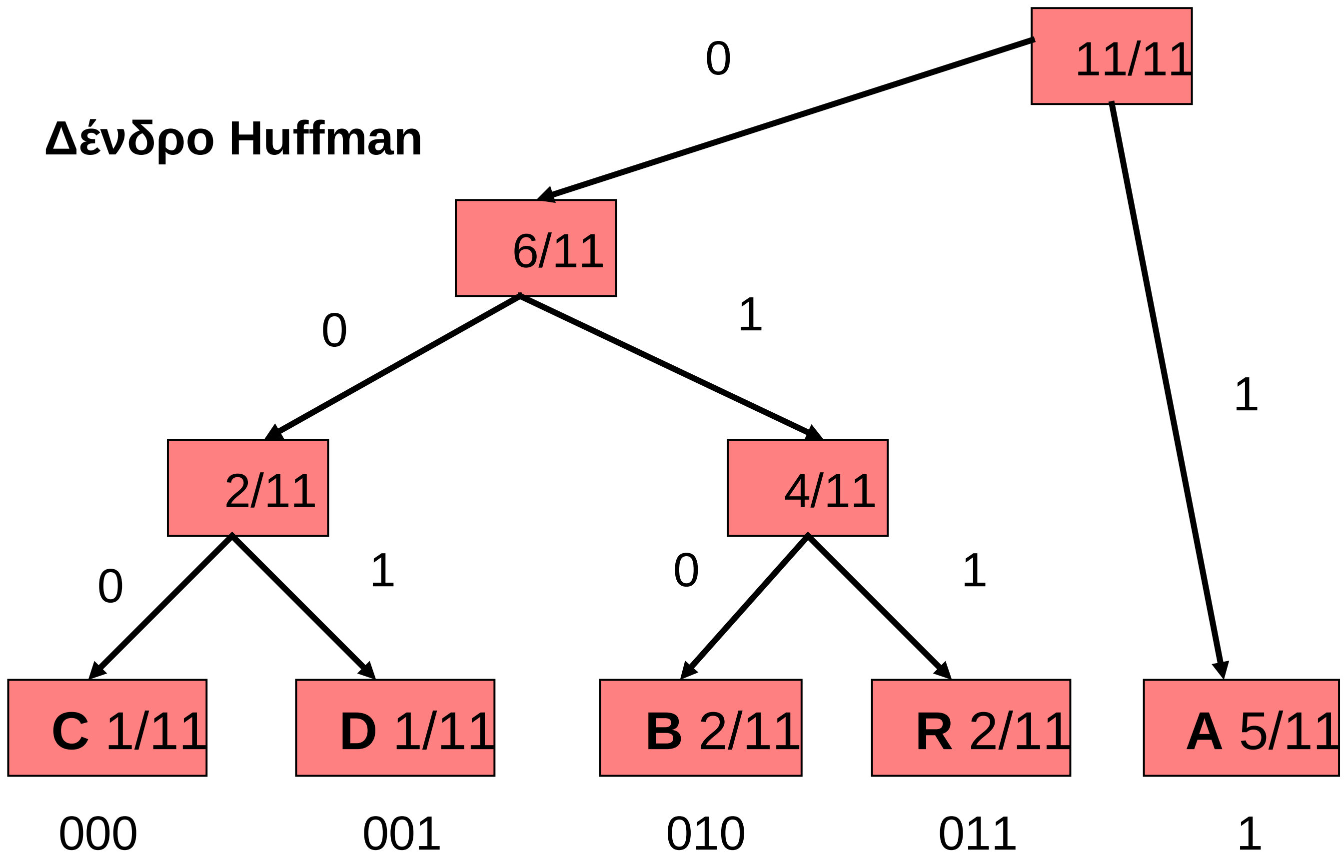
Κωδικοποίηση Huffman

Έστω το ακόλουθο κείμενο

ABRACADABRA

A	5/11
B	2/11
C	1/11
D	1/11
R	2/11

Κωδικοποίηση Huffman



Κωδικοποίηση Huffman

Η ουρά προτεραιότητας μας βοηθάει να βρούμε τα δύο σύμβολα με τη μικρότερη συχνότητα εμφάνισης.

Αν τα στοιχεία είναι οργανωμένα σε ουρά προτεραιότητας (σωρό ελαχίστων συγκεκριμένα) τότε με δύο διαδοχικές λειτουργίες **RemoveTop** παίρνουμε τα δύο μικρότερα στοιχεία.

Εύρεση Top-*k* Στοιχείων

Δίνεται μία συνεχόμενη ροή από δεδομένα, π.χ., μετρήσεις θερμοκρασίας. Θέλουμε να έχουμε πάντοτε τις *k* μικρότερες θερμοκρασίες.

Παράδειγμα

20 10 12 5 25 25 30 2 3

Μετά το τέλος της ροής, το αποτέλεσμα είναι για *k*=3, [2, 3, 5]

Εύρεση Top-*k* Στοιχείων

Λύση

Δημιουργούμε έναν πίνακα με *k* θέσεις, και κάθε φορά που έρχεται ένα στοιχείο *x* ελέγχουμε αν το *x* είναι μικρότερο από το max στοιχείο του πίνακα.

- Αν ο πίνακας είναι αταξινόμητος, η εύρεση του max κοστίζει στη ΧΠ Ο(*k*) συγκρίσεις. Αν η ροή έχει n στοιχεία, συνολικό κόστος **O(*n*k*)**.
- Αν κρατάμε τον πίνακα ταξινομημένο θα πληρώνουμε το κόστος σε μετακινήσεις στοιχείων. Το πολύ Ο(*k*) κάθε φορά, άρα συνολικά πάλι **O(*n*k*)**.
- Με χρήση σωρού μεγίστων το κόστος γίνεται **O(*n* * log*k*)**. Άρα για μεγάλες τιμές του *k* συμφέρει η χρήση του σωρού.


THANKS FOR LISTENING
ANY QUESTIONS?
makeameme.org