Δομές Δεδομένων

Μέρος 6ο: Ουρές Προτεραιότητας

Εξάμηνο Σπουδών: 6ο

Κωδικός Μαθήματος: 681

Τμήμα Μαθηματικών
Πανεπιστήμιο Ιωαννίνων

Μιχάλης Α. Μπέκος
bekos@uoi.gr

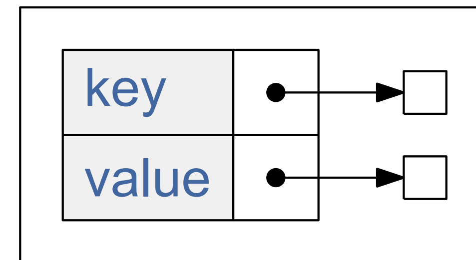
ΑΤΔ: Ουρά Προτεραιότητας

ADT: priority queue

- Ο ΑΤΔ ουρά προτεραιότητας μοντελοποιεί μια συλλογή συγκρίσιμων ανά δύο στοιχείων υποστηρίζοντας τις ακόλουθες λειτουργίες:

`empty()`	ελέγχει αν η δομή είναι κενή
`size()`	επιστρέφει το πλήθος των στοιχείων της δομής
`insert(e)`	ενθέτει το στοιχείο e στη δομή
`removeMin()`	διαγράφει το μικρότερο στοιχείο από τη δομή
`min()`	επιστρέφει (χωρίς να διαγράφει) το μικρότερο στοιχείο της δομής

- Τυπικά, ένα στοιχείο είναι ένα ζεύγος (κλειδί, τιμή), όπου το κλειδί ορίζει την προτεραιότητα
- Δύο στοιχεία σε μια ουρά προτεραιότητας μπορούν να έχουν το ίδιο κλειδί.



Ολικές Διατάξεις

Total orders

- Τα κλειδιά μιας ουράς προτεραιότητας είναι στοιχεία ενός ολικώς διατεταγμένου συνόλου, σ.σ., ενός συνόλου εφοδιασμένου με μια διμερή σχέση $\preceq$ με τις ακόλουθες ιδιότητες:
    - Ανακλαστική ιδιότητα: $x \preceq x$
    - Αντισυμμετρική ιδιότητα: $x \preceq y$ και $y \preceq x \Rightarrow x = y$
    - Μεταβατική ιδιότητα: $x \preceq y$ και $y \preceq z \Rightarrow x \preceq z$
- Οι συγκρίσεις γίνονται μέσω του ΑΤΔ συγκριτή, ο οποίος υποστηρίζει την παρακάτω μέθοδο:

  comparator

  `isLess(p,q)` επιστρέφει `true` αν το στοιχείο p είναι μικρότερο του q

- Σημείωση: Ο έλεγχος της ισότητας μεταξύ δύο στοιχείων p και q μπορεί να επιτευχθεί ελέγχοντας αν `isLess(p,q)` και `isLess(q,p)`

Υλοποίηση ενός Συγκριτή στην C++

A C++ implementation of a comparator

- Επιτυγχάνεται μέσω υπερφόρτωσης του τελεστή `()`
- Παράδειγμα: Σύγκριση αντικειμένων τύπου `Point2D`

```
class LeftRight { // A left-right comparator for points in 2D
public:
  bool operator()(const Point2D& p, const Point2D& q) const {
    return p.getX() < q.getX();
  }
};
template <typename E, typename C> // Element type and comparator
void printSmaller(const E& p, const E& q, const C& isLess) {
  std::cout << (isLess(p, q) ? p : q) << std::endl;
}
int main() {
  Point2D p(1.3, 5.7), q(2.5, 0.6); // Two points
  LeftRight comparator;              // A left-right comparator
  printSmaller(p, q, comparator);   // Outputs: (1.3, 5.7)
}
```

Ταξινόμηση με Ουρά Προτεραιότητας

Priority queue sorting

- Μπορούμε να ταξινομήσουμε συγκρίσιμα στοιχεία με μια ουρά προτεραιότητας ως εξής:

Είσοδος: Μια λίστα L και ένας συγκριτής C.
Εξοδος : Η λίστα L ταξινομημένη σε αύξουσα διάταξη.

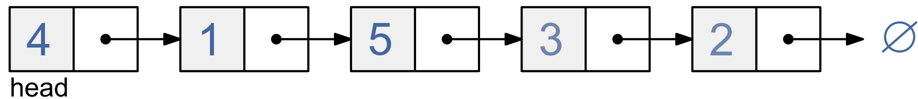
```
1  Algorithm PQSort(L, C)
2      PQ ← Priority queue with comparator C;
3      while !L.empty() do
4          e ← L.front();
5          L.pop_front();
6          PQ.insert(e);
7      while !PQ.empty() do
8          L.push_back(PQ.min());
9          PQ.removeMin();
10     return L;
```

- Σημείωση: Ο χρόνος εκτέλεσης αυτής της μεθόδου ταξινόμησης εξαρτάται από την υλοποίηση της ουράς προτεραιότητας

Υλοποίηση Ουράς Προτεραιότητας με Λίστα

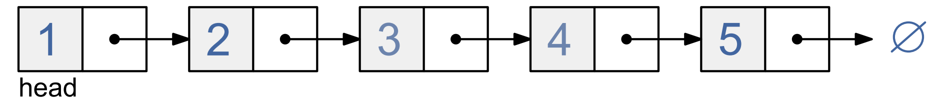
List-based priority queue

- Υλοποίηση με μη-ταξινομημένη λίστα



  - `insert(e)`: Υλοποιείται σε $\mathcal{O}(1)$ χρόνο καθώς κάθε στοιχείο εισάγεται στην αρχή της λίστας
  - `min()`, `removeMin()`: Υλοποιούνται σε $\mathcal{O}(n)$ χρόνο καθώς απαιτείται η διαπέραση όλης της λίστας για τον εντοπισμό του μικρότερου στοιχείου
  - `PQSort`: Υλοποιείται σε $\mathcal{O}(n^2)$ χρόνο
    όμοια της selection sort

- Υλοποίηση με ταξινομημένη λίστα



  - `insert(e)`: Υλοποιείται σε $\mathcal{O}(n)$ χρόνο καθώς απαιτείται ο εντοπισμός της θέσης εισαγωγής του στοιχείου
  - `min()`, `removeMin()`: Υλοποιούνται σε $\mathcal{O}(1)$ χρόνο καθώς το μικρότερο στοιχείο βρίσκεται πάντα στην κεφαλή της λίστας
  - `PQSort`: Υλοποιείται σε $\mathcal{O}(n^2)$ χρόνο
    όμοια της insertion sort

Σωροί

Heaps

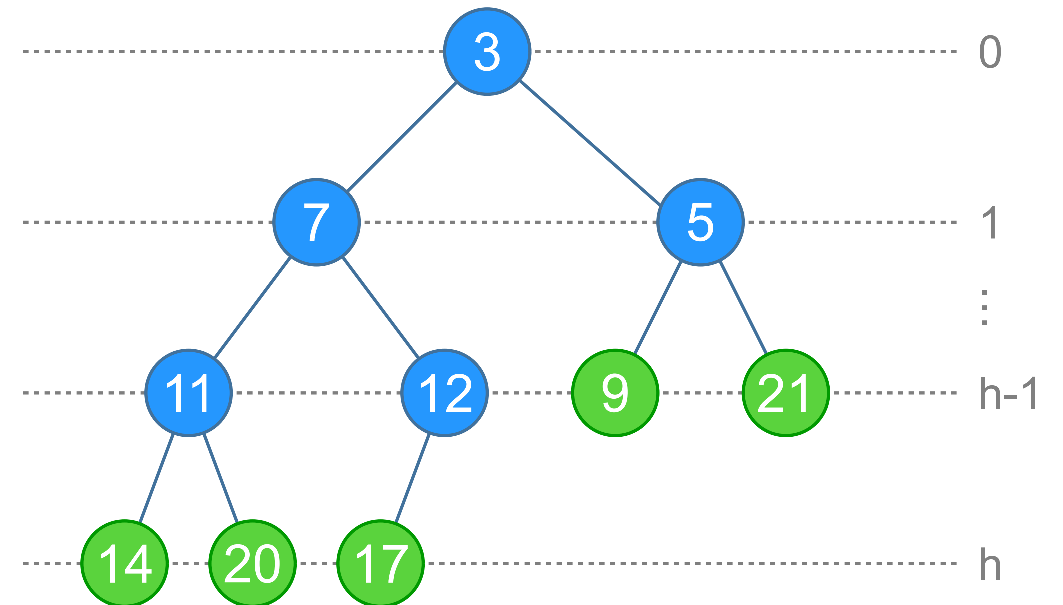
- Ένας σωρός είναι ένα δυαδικό δένδρο, το οποίο αποθηκεύει κλειδιά στους κόμβους του και έχει τις παρακάτω ιδιότητες:

I.1 Διάταξη σωρού: Για κάθε εσωτερικό κόμβο v, εκτός της ρίζας του δένδρου, ισχύει η παρακάτω ιδιότητα:

$$\texttt{key(v)} \geq \texttt{key(parent(v))}$$

I.2 Το δένδρο είναι "πλήρες δυαδικό", σ.σ., αν το ύψος του είναι h, τότε:

- Για κάθε $i = 0, 1, \ldots, h-1$, υπάρχουν 2^i κόμβοι βάθους i,
- Οι εσωτερικοί κόμβοι βάθους $h-1$ είναι στα αριστερά των αντίστοιχων εξωτερικών ιδίου βάθους.



Σωροί

Heaps

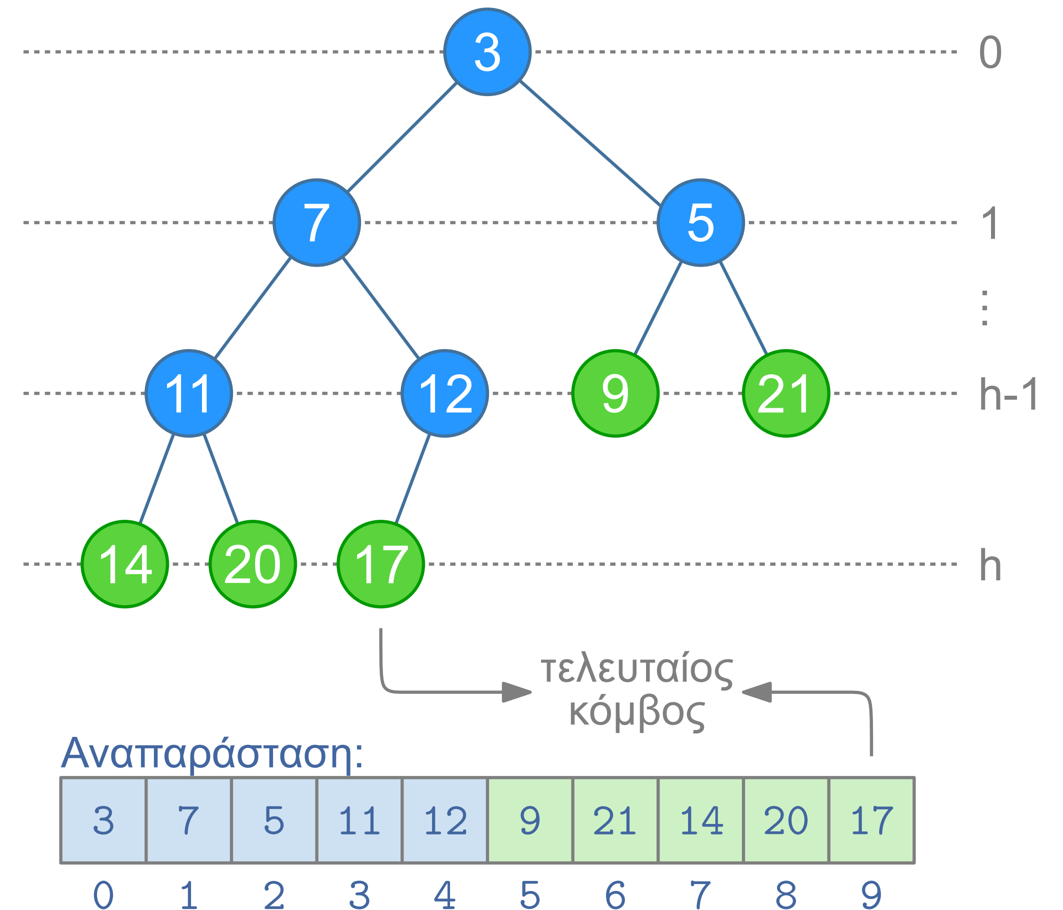
- Ένας σωρός είναι ένα δυαδικό δένδρο, το οποίο αποθηκεύει κλειδιά στους κόμβους του και έχει τις παρακάτω ιδιότητες:

I.1 Διάταξη σωρού: Για κάθε εσωτερικό κόμβο v, εκτός της ρίζας του δένδρου, ισχύει η παρακάτω ιδιότητα:

$$\texttt{key(v)} \geq \texttt{key(parent(v))}$$

I.2 Το δένδρο είναι "πλήρες δυαδικό", σ.σ., αν το ύψος του είναι h, τότε:

- Για κάθε $i = 0, 1, \ldots, h-1$, υπάρχουν 2^i κόμβοι βάθους i,
- Οι εσωτερικοί κόμβοι βάθους $h-1$ είναι στα αριστερά των αντίστοιχων εξωτερικών ιδίου βάθους.



Ύψος Σωρού

- Θεώρημα: Το ύψος ενός σωρού με n στοιχεία είναι $\mathcal{O}(\log n)$.

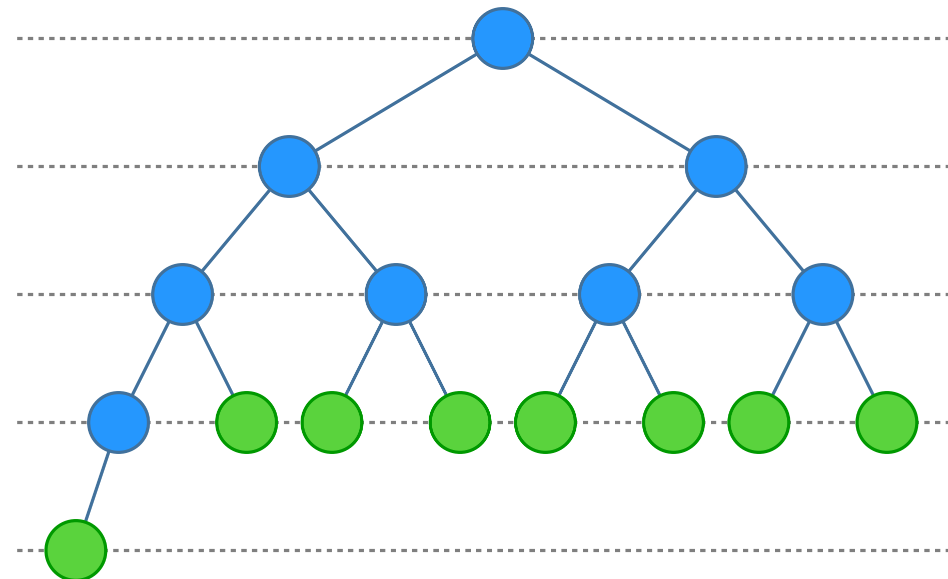
  Απόδειξη:

  - $h \leftarrow$ το ύψος του σωρού
  - Για κάθε $i = 0, 1, \ldots, h-1$, υπάρχουν 2^i κόμβοι βάθους i, και τουλάχιστον ένας βάθους h. Οπότε:

    $$n \geq 2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} + 1$$

    $$\Rightarrow n \geq \frac{2^h - 1}{2 - 1} + 1 \Rightarrow n \geq 2^h$$

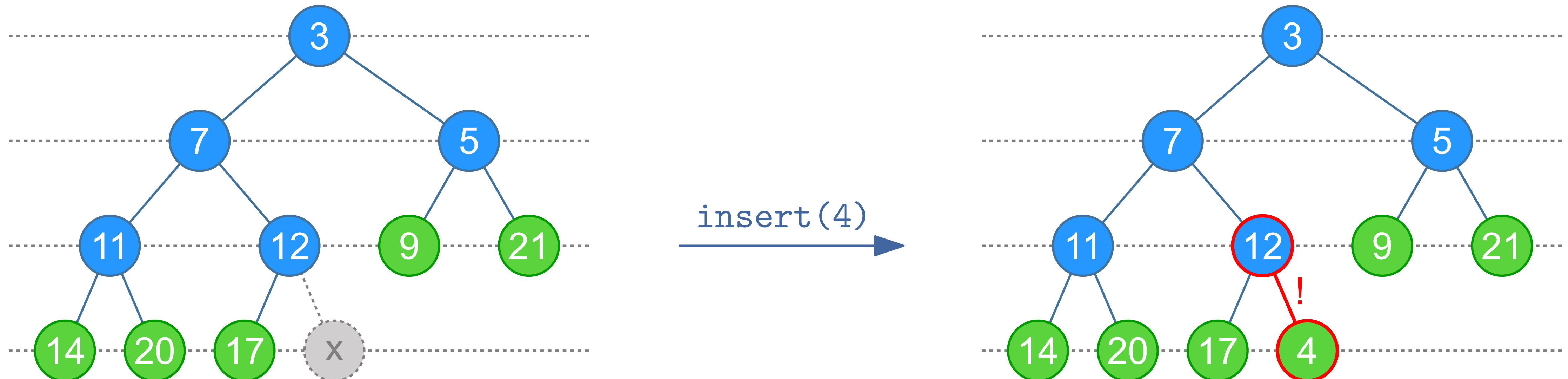
    $$\Rightarrow h \leq \log n$$



βάθος	στοιχεία
0	1
1	2
⋮	⋮
$h-1$	2^{h-1}
h	1

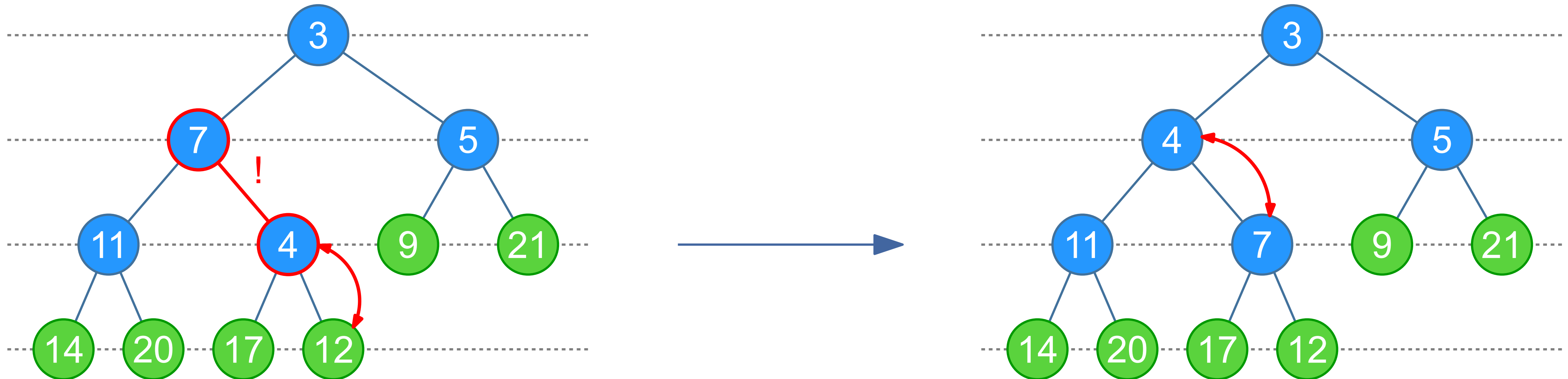
Εισαγωγή στοιχείων σε σωρό

- `insert(e)`: Η εισαγωγή ενός στοιχείου e σε ένα σωρό γίνεται ως εξής:
  - Αρχικά εντοπίζουμε τον νέο κόμβο εισαγωγής x στο σωρό
    (σ.σ., το επόμενο φύλλο μετά τον τελευταίο κόμβο)
  - Αποθηκεύουμε το στοιχείο e στον κόμβο x
  - Αποκαθιστούμε την ιδιότητα της διάταξης σωρού (αλγόριθμος upheap; επόμενο slide)



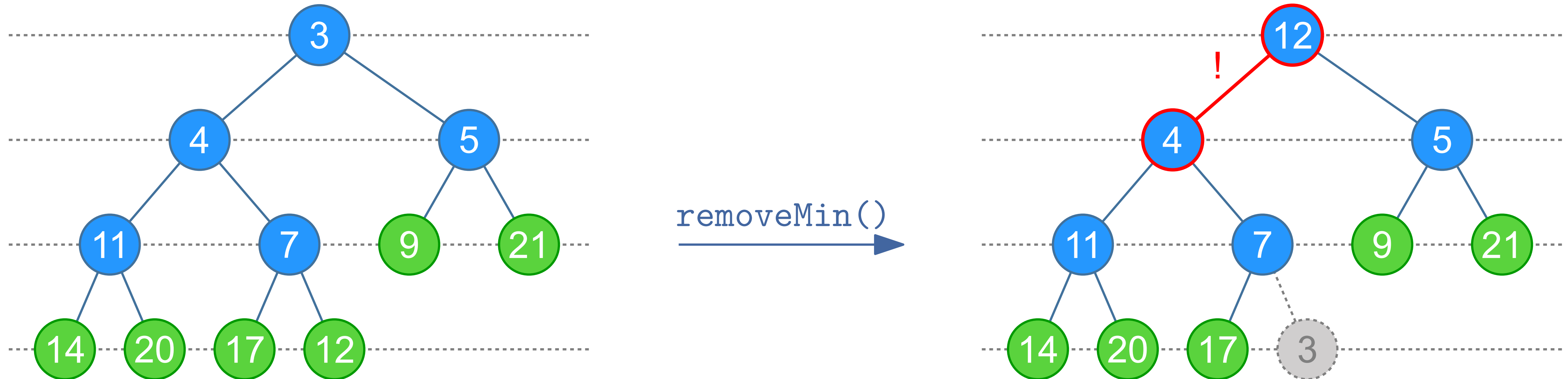
Upheap

- Ο αλγόριθμος upheap επαναφέρει την ιδιότητα της διάταξης σωρού, ανταλλάσοντας τη θέση του στοιχείου e με αυτή του γονέα του επαναληπτικά, έως ότου το στοιχείο e:
  - είτε βρεθεί στη ρίζα του δένδρου
  - είτε σε μια θέση στην οποία το κλειδί του γονέα είναι μικρότερο ή ίσο αυτό του στοιχείου e
- Δεδομένου ότι ένας σωρός έχει ύψος $\mathcal{O}(\log n)$, ο αλγόριθμος τερματίζει σε χρόνο $\mathcal{O}(\log n)$



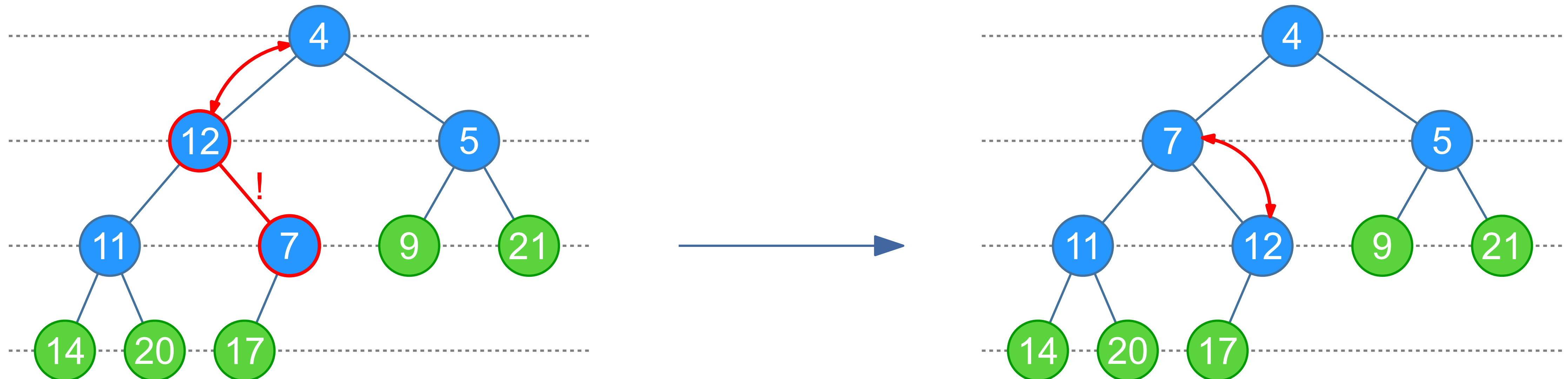
Διαγραφή ελαχίστου από σωρό

- `deleteMin()`: Η διαγραφή του ελάχιστου στοιχείου από ένα σωρό γίνεται ως εξής:
    - Αντικαθιστούμε το κλειδί της ρίζας του δένδρου με αυτό του τελευταίου κόμβου
    - Διαγράφουμε τον τελευταίο κόμβο του σωρού
    - Αποκαθιστούμε την ιδιότητα της διάταξης σωρού (αλγόριθμος downheap; επόμενο slide)



Downheap

- Ο αλγόριθμος downheap επαναφέρει την ιδιότητα της διάταξης σωρού, ανταλλάσοντας τη θέση του στοιχείου e με αυτή του μικρότερου παιδιού του επαναληπτικά, έως ότου:
  - είτε το στοιχείο e βρεθεί σε κάποιο φύλλο του δένδρου
  - είτε σε μια θέση στην οποία το κλειδί του γονέα είναι μικρότερο ή ίσο αυτό του στοιχείου e
- Δεδομένου ότι ένας σωρός έχει ύψος $\mathcal{O}(\log n)$, ο αλγόριθμος τερματίζει σε χρόνο $\mathcal{O}(\log n)$



ΑΤΔ: Ουρά Προτεραιότητας

	Unsorted list	Sorted list	Heap
`empty()`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`size()`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert(e)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`removeMin()`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
`min()`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

- Ερώτημα: Πως θα υλοποιούσατε μια ουρά προτεραιότητας με μια λίστα παράβλεψης?

Ταξινόμηση σωρού

Heap sort

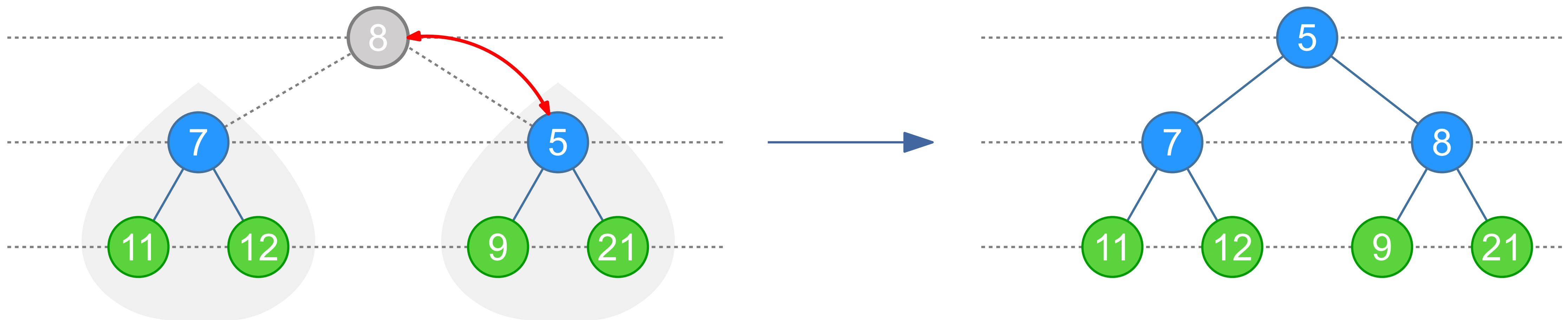
- Χρησιμοποιώντας την υλοποίηση μιας ουράς προτεραιότητας ως σωρό, μπορούμε να ταξινομήσουμε n στοιχεία σε χρόνο $\mathcal{O}(n \log n)$
  - n εισαγωγές $\rightarrow \mathcal{O}(n \log n)$ ← στη συνέχεια δείχνουμε πως το βήμα αυτό μπορεί να υλοποιηθεί σε $\mathcal{O}(n)$ χρόνο
  - n διαγραφές ελαχίστου $\rightarrow \mathcal{O}(n \log n)$

- Η ταξινόμηση σωρού είναι πολύ πιο γρήγορη μέθοδος ταξινόμησης από αντίστοιχες τετραγωνικές μεθόδους, όπως η insertion-sort ή η selection-sort

Συγχώνευση σωρών

Merging two heaps

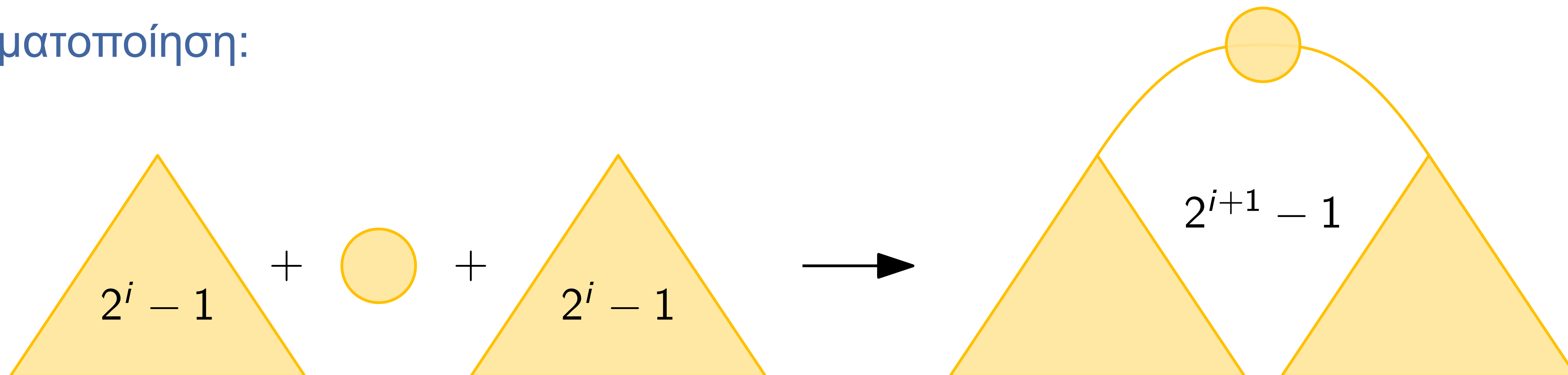
- Συγχώνευση σωρών: Δοθέντων δύο σωρών και ενός στοιχείου e:
    - δημιουργούμε μια νέα σωρό με το στοιχείο e στη ρίζα, και
    - τους δύο σωρούς ως υποδένδρα του e
- Αποκαθιστούμε την ιδιότητα της διάταξης σωρού χρησιμοποιώντας τον αλγόριθμο downheap



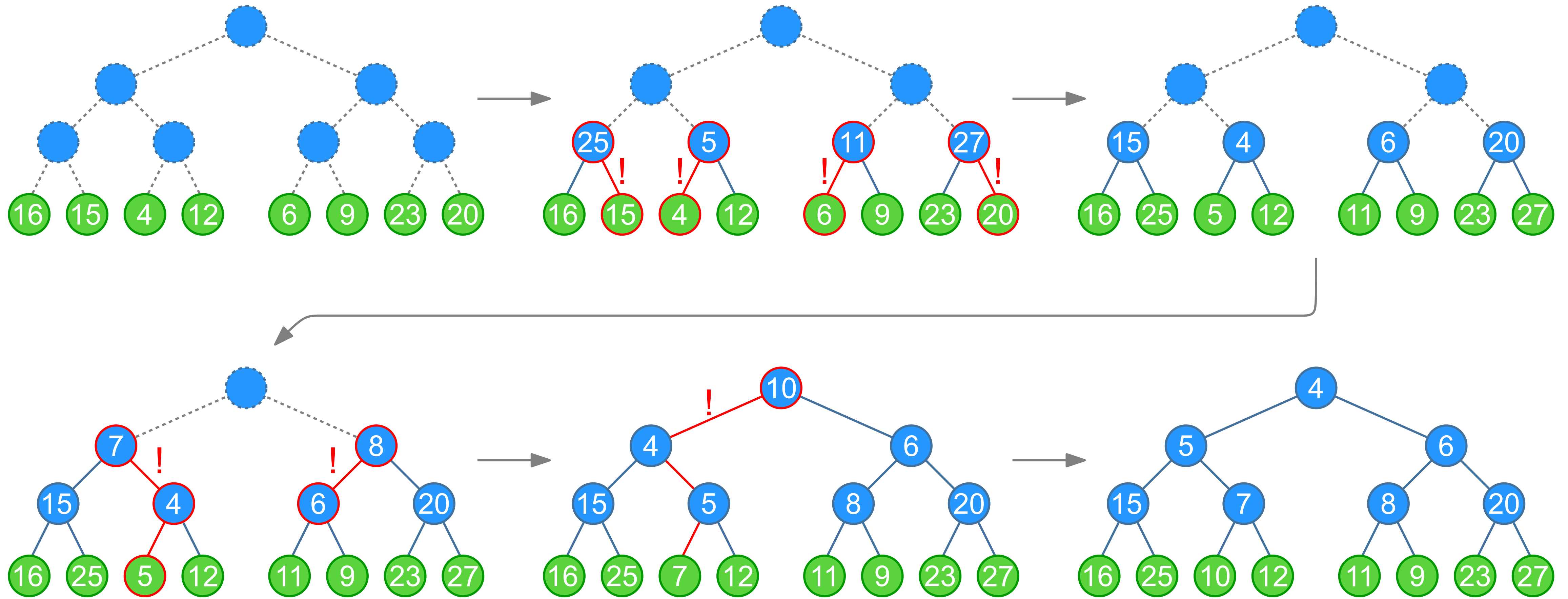
Κατασκευή σωρού από κάτω προς τα πάνω

Bottom-up heap construction

- Μπορούμε να κατασκευάσουμε έναν σωρό που αποθηκεύει n στοιχεία χρησιμοποιώντας μια κατασκευή από κάτω προς τα πάνω.
  - η κατασκευή γίνεται σε $\mathcal{O}(\log n)$ φάσεις
  - στην i-οστή φάση, δύο σωροί $2^i - 1$ στοιχείων συγχωνεύονται σε ένα σωρό με $2^{i+1} - 1$ στοιχεία
- Σχηματοποίηση:



Παράδειγμα



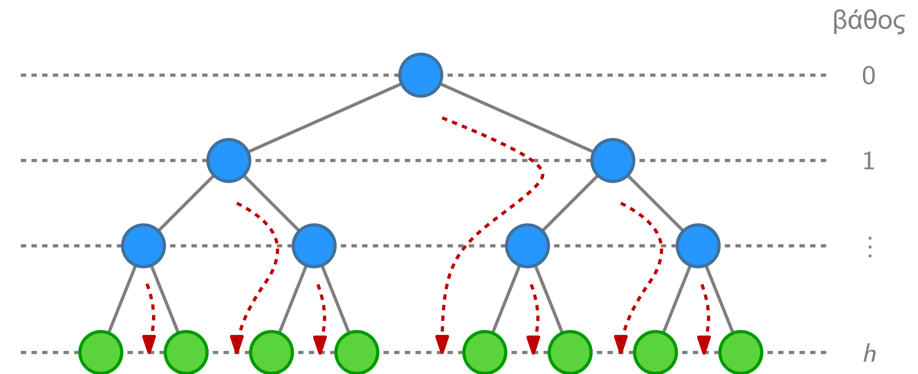
Ανάλυση

- Θεώρημα: Η κατασκευή ενός σωρού από κάτω προς τα πάνω υλοποιείται σε $\mathcal{O}(n)$ χρόνο.

  Απόδειξη:

  - Στην χειρότερη περίπτωση, ένας κόμβος βάθους d ακολουθεί ένα μονοπάτι μήκους $h - d$ προς ένα φύλλο του δένδρου.

    Κόστος: $\mathcal{O}(h - d)$.



  - Δεδομένου ότι υπάρχουν 2^d κόμβοι βάθους d, ο χρόνος κατασκευής του σωρού είναι:

$$\begin{aligned}
\sum_{d=0}^{h}(h-d)\cdot 2^d &\overset{j=h-d}{=} \sum_{j=h}^{0} j\cdot 2^{h-j} = \sum_{j=0}^{h} j\cdot 2^{h-j} \\
&= 2^h\cdot\sum_{j=0}^{h}\frac{j}{2^j} \leq 2^h\cdot\sum_{j=0}^{\infty}\frac{j}{2^j} = 2^h\cdot\left(2\cdot\sum_{j=0}^{\infty}\frac{j}{2^j} - \sum_{j=0}^{\infty}\frac{j}{2^j}\right) \\
&= 2^h\cdot\left(1+\frac{1}{2^1}+\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}\dots\right) = 2^h\cdot\frac{1}{1-\frac{1}{2}} \overset{h=\mathcal{O}(\log n)}{=} \mathcal{O}(n)
\end{aligned}$$

ΑΤΔ: Προσαρμόσιμη Ουρά Προτεραιότητας

ADT:Adaptable priority queue

- Ο ΑΤΔ προσαρμόσιμη ουρά προτεραιότητας επεκτείνει τον ΑΤΔ ουρά προτεραιότητας υποστηρίζοντας επιπρόσθετα τις ακόλουθες λειτουργίες:

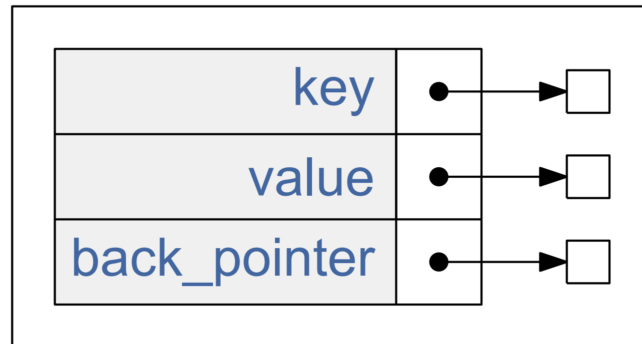
`remove(p)`	διαγράφει από τη δομή το στοιχείο που βρίσκεται στη θέση p
`replace(p,e)`	αντικαθιστά το στοιχείο της θέσης p με το δοθέν στοιχείο e

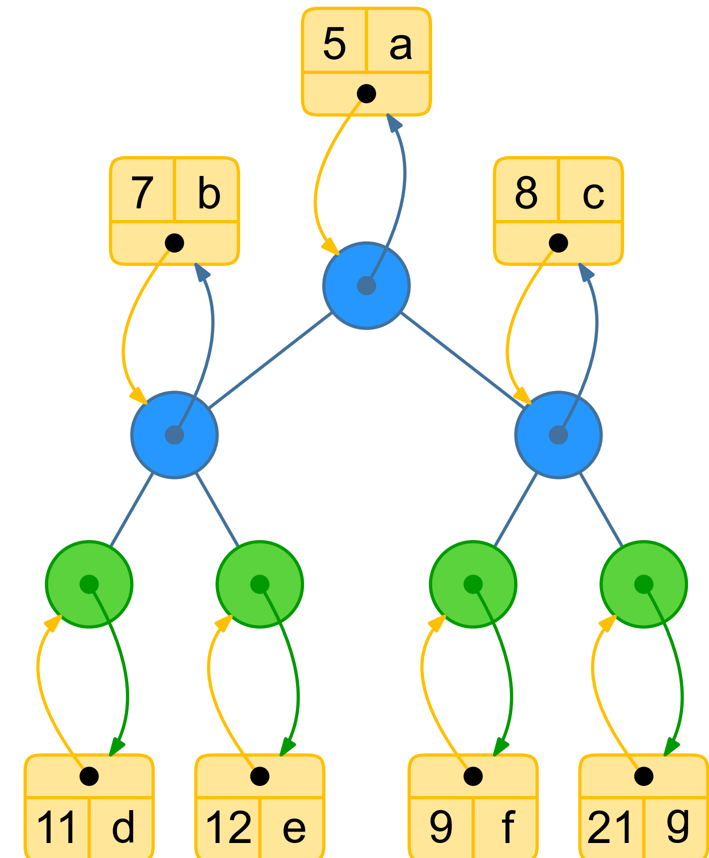
- Παρατήρηση: Για την αποδοτική υλοποίηση των παραπάνω λειτουργιών απαιτείται γρήγορος εντοπισμός της θέσης p ενός στοιχείου στην ουρά προτεραιότητας

Υλοποίηση μέσω στοιχείων που γνωρίζουν τη θέση τους

Implementation through location-aware entries

- Κάθε στοιχείο, το οποίο δημιουργείται από τη δομή, κατά την εισαγωγή του σε αυτη, γνωρίζει τη θέση του μέσα στην δομή (η οποία είναι πάντα επικαιροποιημένη)





- Με άλλα λόγια, η ουρά προτεραιότητας αποθηκεύει σε κάθε στοιχείο πληροφορίες σχετικές με τη θέση του στην υλοποίηση

- Με τον τρόπο αυτό, διευκολύνονται μελλοντικές ενημερώσεις, αφού η θέση κάθε στοιχείου στην δομή εντοπίζεται σε $\mathcal{O}(1)$ χρόνο.

ΑΤΔ: Προσαρμόσιμη Ουρά Προτεραιότητας

	Unsorted list	Sorted list	Heap
`empty()`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`size()`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`insert(e)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
`removeMin()`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
`min()`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove(p)`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
`replace(p,e)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

STD Priority Queues

```
#include <iostream>
#include <queue>

int main ()
{
  std::priority_queue<int> pq;

  pq.push(30);
  pq.push(100);
  pq.push(25);

  std::cout << "Popping out elements...";
  while (!pq.empty())
  {
     std::cout << ' ' << pq.top();
     pq.pop();
  }
  std::cout << std::endl;
}
```

https://www.cplusplus.com/reference/queue/priority_queue/pop/

Επιπλέον υλικό

- Κεφάλαιο 5:
  Michael T. Goodrich, Roberto Tamassia, Αλγόριθμοι σχεδίαση και εφαρμογές
  ISBN: 9789605126971, εκδόσεις Γκιούρδα, 2016.