Δομές Δεδομένων

ΑΤΔ “Ουρά Προτεραιότητας”

Α. Συμβώνης
Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Εφαρμοσμένων Μαθηματικών και Φυσικών Επιστημών
Τομέας Μαθηματικών

Ουρά προτεραιότητας
[Priority Queue]

Ουρά προτεραιότητας [Priority Queue]

- Ο ΑΤΔ ουρά προτεραιότητας αποθηκεύει μία συλλογή από εγγραφές (αντικείμενα) τύπου `Entry`
- Εγγραφές τύπου `Entry` αποτελούνται από ένα ζεύγος `(key, value)`, οπου:
  - `key`: είναι το κλειδί της εγγραφής
  - `value`: η τιμή (τα δεδομένα) που αποθηκεύονται στην εγγραφή
- Βασικές μέθοδοι της ουράς προτεραιότητας:
  - `insert(k, v)`: εισάγει μία εγγραφή με κλειδί `k` και τιμή `v`
  - `removeMin()`: αφαιρεί και επιστρέφει την εγγραφή με το μικρότερο κλειδί, ή null εάν η ουρά προτεραιότητας είναι άδεια.
- Βοηθητικές μέθοδοι της ουράς προτεραιότητας:
  - `min()`: επιστρέφει, χωρίς να αφαιρεί, την εγγραφή με το μικρότερο κλειδί.
  - `int size()`: επιστρέφει το πλήθος των εγγραφών της ουράς προτεραιότητας
  - `boolean isEmpty()`: ελέγχει εάν η ουρά προτεραιότητας είναι άδεια ή όχι

Ουρά προτεραιότητας [Priority Queue]

- Εφαρμογές
  - Εξυπηρέτηση “πελατών”: όταν η πολιτική “First-come First-serve” δεν αρκεί.
    - Επιλογή επομένου αεροπλάνου για προσγείωση
    - Επιλογή επιβατών σε αναμονή
  - Αγορά/πώληση μετοχών στο χρηματιστήριο
  - Προσομοίωση διακριτών γεγονότων [discrete event simulation]
    - Το κλειδί μοντελοποιεί τις χρονικές στιγμές που πρέπει να συμβούν τα γεγονότα.
  - Βοηθητική χρήση σε πλήθος αλγορίθμων

Ουρά προτεραιότητας [Priority Queue]

- Η διεπαφή `PriorityQueue`

Στην βιβλιοθήκης της Java:
Κλάση java.util.PriorityQueue

```
1  public interface PriorityQueue<K,V> {
2
3    int size();
4    boolean isEmpty();
5
6    Entry<K,V> insert(K key, V value) throws IllegalArgumentException;
7
8    Entry<K,V> min();
9    Entry<K,V> removeMin();
10 }
```

- Η διεπαφή `Entry<K,V>`

```
1 public interface Entry<K,V> {
2   K getKey();
3   V getValue();
4 }
```

Ουρά προτεραιότητας [Priority Queue]

Παράδειγμα: Χρήση `PriorityQueue`

Method	Return value	Περιεχόμενα Priority Queue
insert(5,A)	(5,A)	{ (5,A) }
insert(9,C)	(9,C)	{ (5,A), (9,C) }
insert(3,B)	(3,B)	{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

Ουρά προτεραιότητας [Priority Queue]

- Η διεπαφή `java.util.Comparator`
  - Ένα αντικείμενο τύπου `Comparator<T>` υλοποιεί μηχανισμό σύγκρισης αναμεσά σε αντικείμενα τύπου `T` με βάση μία σχέση ολικής διάταξης
  - `int compare(T o1, T o2)`: Επιστρέφει ακέραιο `i`:
    - `i<0` εάν `o1 < o2`
    - `i=0` εάν `o1 == o2`
    - `i>0` εάν `o1 > o2`
  - Κάθε ουρά προτεραιότητας χρησιμοποιεί έναν Comparator για να συγκρίνει κλειδιά εγγραφών της

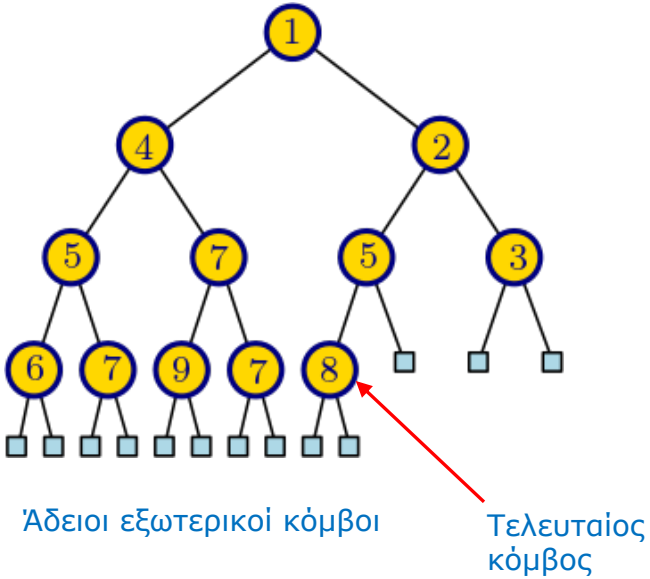
Ουρά προτεραιότητας: απλές υλοποιήσεις

	Υλοποίηση			
Priority Queue	Μη-ταξινομημένο διάνυσμα (Array)	Ταξινομημένο διάνυσμα (Array, **Circular Array**)	Μη-ταξινομημένη λίστα (PositionalList)	Ταξινομημένη λίστα (PositionalList)
`insert()`	O(1)	O(n)	O(1)	O(n)
`removeMin()`	O(n)	O(1)	O(n)	O(1)
`min()`	O(n)	O(1)	O(n)	O(1)
`size()`	O(1)			
`isEmpty()`				
`isFull()`				

Η Δομή Δεδομένων του Σωρού [Heap]

- Ενας σωρός είναι ένα δυαδικό δένδρο το οποίο αποθηκεύει στον κάθε κόμβο του ένα κλειδί [key] και έχει τις παρακάτω ιδιότητες:

1. Ιδιότητα Διάταξης Σωρού [heap order]: για κάθε εσωτερικό κόμβο v του δένδρου, εκτός της ρίζας, ισχύει ότι:
   $$key(v) \geq key(parent(v))$$
2. Ιδιότητα Πλήρους Δυαδικού Δένδρου [complete binary tree]: Έστω h το ύψος του σωρού.
   - Στο επίπεδο i, $i = 0, \ldots, h-1$, υπάρχους ακριβώς 2^i κόμβοι [πλήρη επίπεδα]
   - Στο επίπεδο h, οι κόμβοι καταλαμβάνουν διαδοχικές θέσεις στο αριστερό τμήμα του επιπέδου [αριστερά-στοιχισμένο επίπεδο]

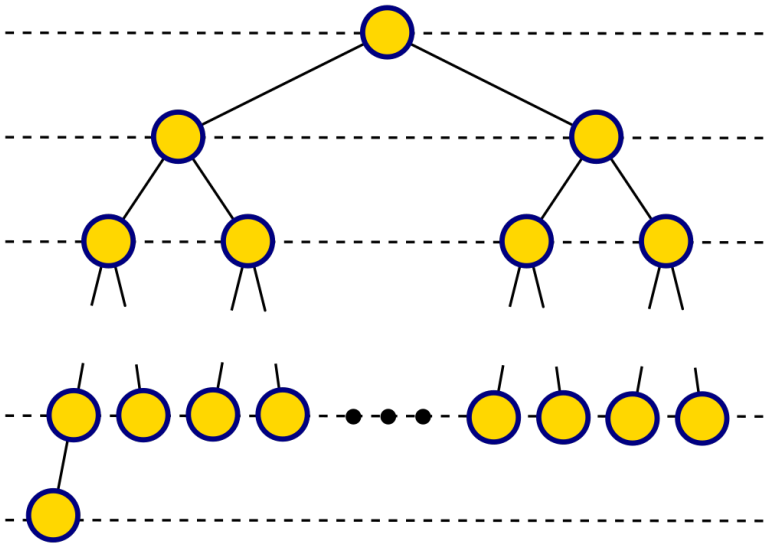


Η Δομή Δεδομένων του Σωρού [Heap]

Θεώρημα: Ένας σωρός που αποθηκεύει n στοιχεία έχει ύψος $O(\log n)$

Απόδειξη:

- Έστω h το ύψος του σωρού που έχει n στοιχεία.
- Ισχύει ότι: $n \geq 1 + 2 + 4 + \cdots + 2^{h-1} + 1$
  - Στα επίπεδα βάθους $i, i = 0, \ldots, h-1$ βρίσκονται ακριβώς στοιχεία.
  - Στο επίπεδο βάθους h βρίσκεται τουλάχιστον 1 στοιχείο.
- Συνεπώς: $n \geq 2^h \iff h \leq \log n$

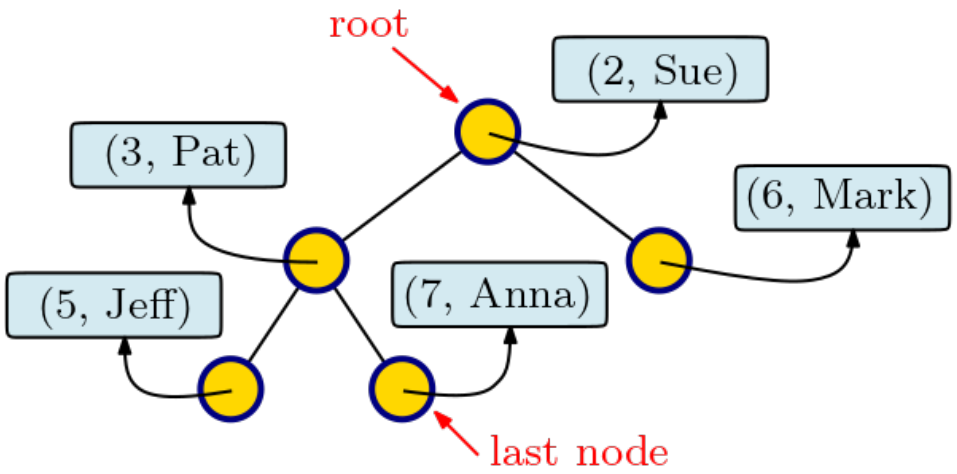


depth	keys
0	1
1	2
2	4
$h-1$	2^{h-1}
h	1

Υλοποίηση Ουράς Προτεραιότητας με Σωρό

Ουρά προτεραιότητας με δομή σωρού

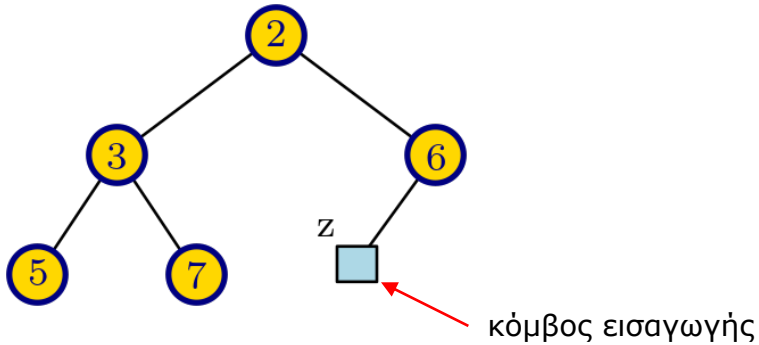
- Αποθηκεύουμε ένα ζεύγος `(key, element)` σε κάθε κόμβο
- Αποθηκεύουμε τη θέση της ρίζας και του τελευταίου κόμβου

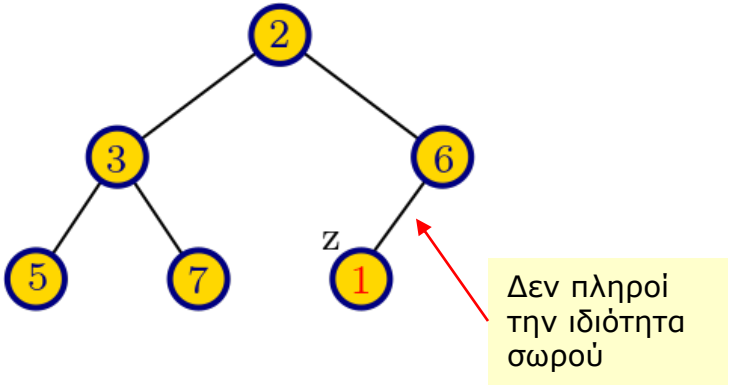


Υλοποίηση Ουράς Προτεραιότητας με Σωρό

Εισαγωγή στοιχείου στο σωρό

- Η μέθοδος `insert(k,v)` της ουράς προτεραιότητας υλοποιείται μέσω της εισαγωγής του κλειδιού `k` στο σωρό
- Αλγόριθμος εισαγωγής [addLeaf()]
  1. Βρες τον κόμβο εισαγωγής `z` (φύλλο, αμέσως μετά τον τελευταίο κόμβο)
  2. Αποθήκευσε το κλειδί `k` στον φύλλο `z`
  3. Επανάφερε σε ισχύ την ιδιότητα διάταξης-σωρού



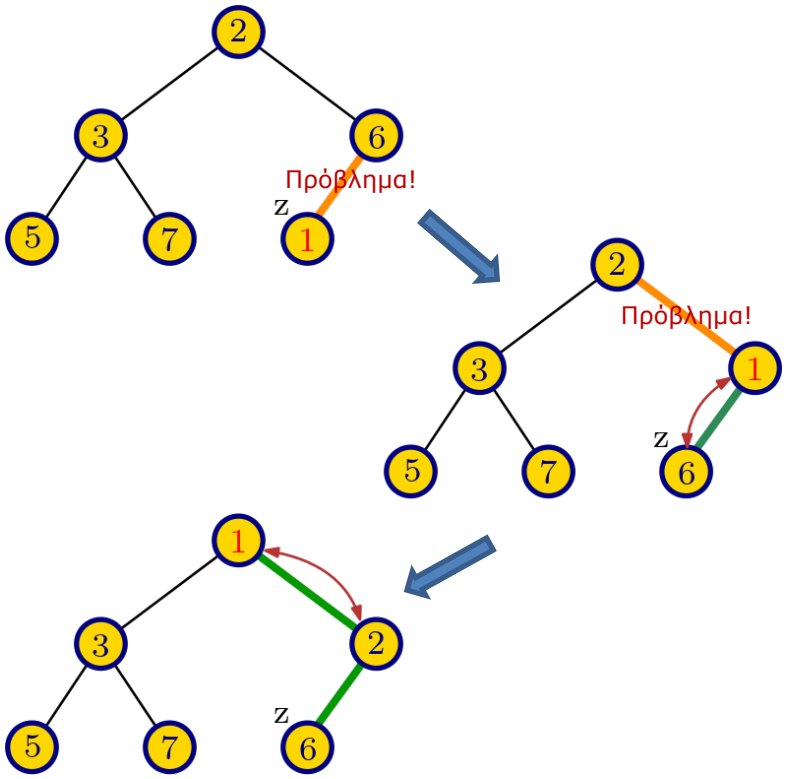


Υλοποίηση Ουράς Προτεραιότητας με Σωρό

Αλγόριθμος addLeaf()

- Ξεκινώντας από το φύλλο, και κινούμενο πάνω στο μοναδικό μονοπάτι που οδηγεί στη ρίζα, το κλειδί μεσω διαδοχικών συγκρίσεων και ανταλλαγών (swaps) καταλήγει:
  - Είτε στη ρίζα του σωρού, ή
  - σε κόμβο x τέτοιον ώστε να ισχύει ότι $getKey(x) \leq getKey(parent(x))$
- Η διαδρομή στην οποία κινείται το κλειδί έχει μήκος το πολύ $\log n$.

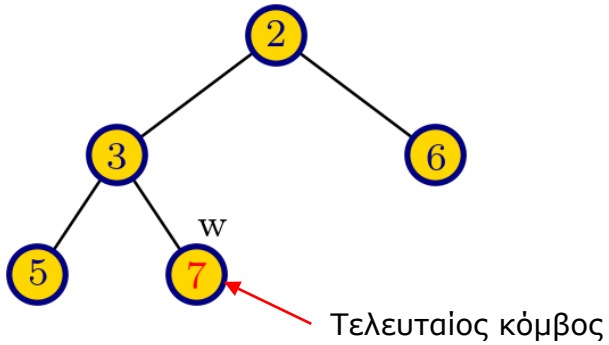
➢ Ο αλγόριθμος addLeaf() τερματίζει σε χρόνο $O(\log n)$.

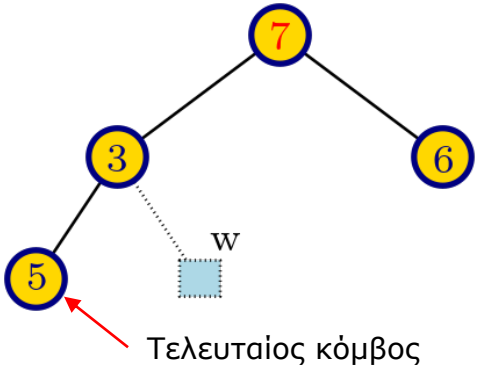


Υλοποίηση Ουράς Προτεραιότητας με Σωρό

Διαγραφή ελαχίστου από σωρό

- Η μέθοδος `removeMin()` της ουράς προτεραιότητας υλοποιείται μέσω της διαγραφής του κλειδιού της ρίζας του σωρού.
- Αλγόριθμος διαγραφής [addRoot()]
  1. Αντικατέστησε το κλειδί της ρίζας με το κλειδί του τελευταίου κόμβου `w`
  2. Διέγραψε τον κόμβο `w`
  3. Επανάφερε σε ισχύ την ιδιότητα διάταξης-σωρού



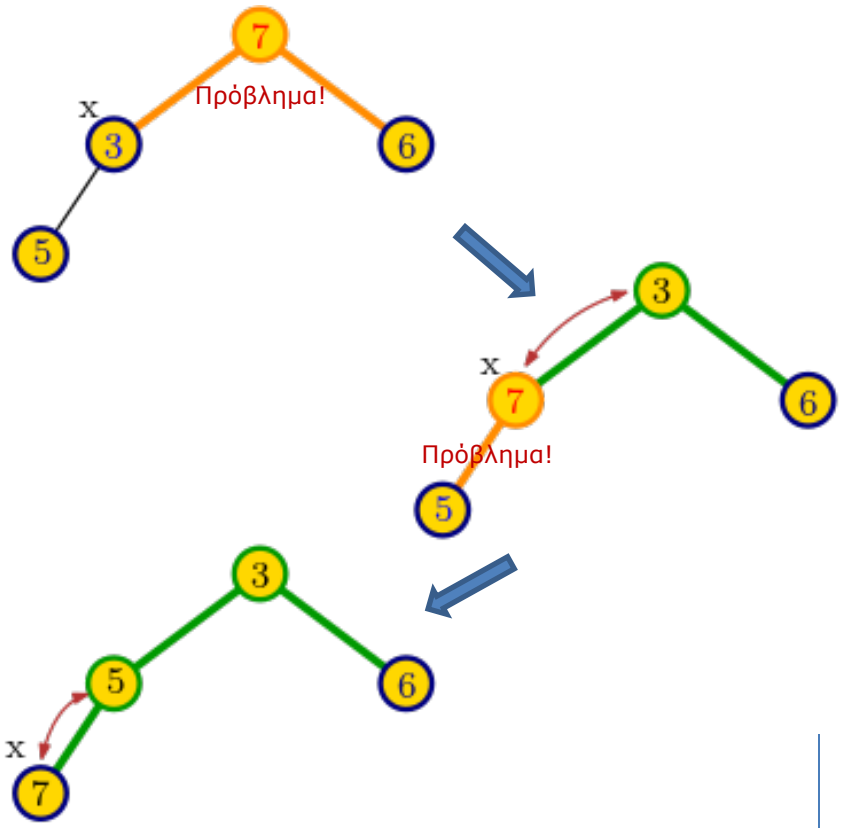


Υλοποίηση Ουράς Προτεραιότητας με Σωρό

Αλγόριθμος addRoot(r)

- Αν το νέο κλειδί της ρίζας παραβιάζει τη διάταξη σωρού μεταξύ της ρίζας και των παιδιών της, τότε:
  - Έστω `x` το παιδί της ρίζας με το μικρότερο κλειδί.
  - Αντάλλαξε το κλειδί της ρίζας με το κλειδί του κόμβου `x`
  - Εκτέλεσε αναδρομικά τον `addRoot(x)`
- Η διαδρομή στην οποία κινείται το κλειδί που τοποθετήθηκε στη ρίζα έχει μήκος το πολύ $\log n$.

➢ Ο αλγόριθμος addRoot() τερματίζει σε χρόνο $O(\log n)$.



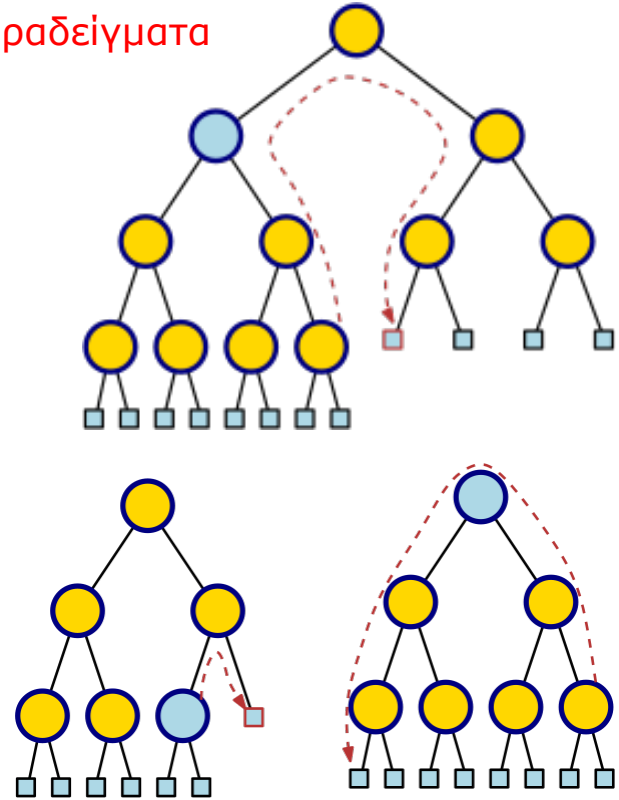
Υλοποίηση Ουράς Προτεραιότητας με Σωρό

Εύρεση του νέου "τελευταίου" κόμβου

- Ο κόμβος εισαγωγής μπορεί να υπολογιστεί διασχίζοντας ένα μονοπάτι μήκους $O(\log n)$.
  - Ξεκινώντας από τον τελευταίο κόμβο, κινήσου προς τα πάνω μέχρι να φτάσεις είτε σε ένα αριστερό παιδί ή τη ρίζα του σωρού.
  - Εάν έχεις φτάσει σε ένα αριστερό παιδί, πήγαινε στο δεξί αδέλφι του.
  - Κινήσου προς τα κάτω, πηγαίνοντας πάντα προς το αριστερό παιδί, έως ότου φτάσεις σε ένα φύλλο.
- Συμμετρικός αλγόριθμος χρησιμοποιείται για τον υπολογισμό του τελευταίου κόμβου μετά από διαγραφή.

➢ Η ευρεση του νέου τελευταιου κομβου ολοκληρωνεται σε χρόνο $O(\log n)$.

Παραδείγματα



Ουρά προτεραιότητας: Ταξινόμηση

- Ταξινόμηση Σωρου [Heap-sort]

Με χρήση μιας ουράς προτεραιότητας μπορούμε να ταξινομήσουμε μία λίστα από n "comparable" στοιχεία

1. Εισάγουμε ένα-ένα τα στοιχεία στην ουρά προτεραιότητας [n εντολές `insert()`]
2. Αφαιρούμε ένα-ένα τα στοιχεία σε αύξουσα σειρά [n εντολές `removeMin()`]

➢ Εάν η ουρά προτεραιότητας υλοποιηθεί ως σωρός, τότε η ταξινόμηση n στοιχείων ολοκληρώνεται σε βέλτιστο χρόνο $O(n \log n)$

Αλγόριθμος PQ-Sort(s, c)

Input `List s, Comparator c`

Output `List s`, ταξινομημένη σε αύξουσα σειρά σύμφωνα με τον `c`

```
p = new HeapPriorityQueue()
while !s.isEmpty()
      e = s.remove(s.first())
      p.insert(e,null)
while !p.isEmpty()
      e=p.removeMin().getKey()
      s.addLast(e)
```

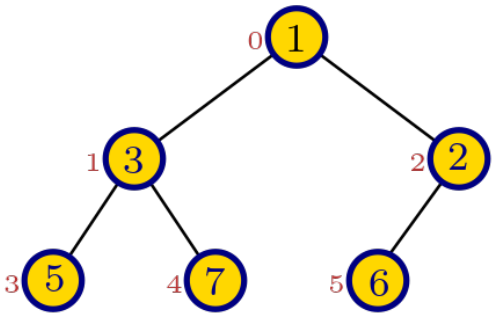
Υλοποίηση Ουράς Προτεραιότητας με “Διάνυσμα-Σωρό”

Υλοποίηση Σωρού με διάνυσμα

- Έστω `rank[v]` η αρίθμηση του κόμβου `v` στο δυαδικό δένδρο όταν αυτή γίνεται ανά διαδοχικά επίπεδα, ξεκινώντας από τη ρίζα, και σε κάθε επίπεδο από αριστερά προς δεξιά.
- Τοποθέτησε το κλειδί του κόμβου v στη θέση $rank(v)$:

```
heapArray[rank(v)]=v.getKey()
```

- Για τον κόμβο με v ισχύει:
  - Ο πατέρας είναι στη θέση: $\lfloor (rank(v) - 1)/2 \rfloor$
  - Το αριστερό παιδί είναι στη θέση: $2 \cdot rank(v) + 1$
  - Το δεξί παιδί είναι στη θέση: $2 \cdot rank(v) + 2$
- Εισαγωγή στοιχείου σε σωρό n στοιχείων γίνεται στη θέση n.
- Η διαγραφή του ελάχιστου στοιχείου αντιστοιχεί στην διαγραφή του στοιχείου στη θέση $n - 1$ [το τελευταίο στοιχείο του σωρού]

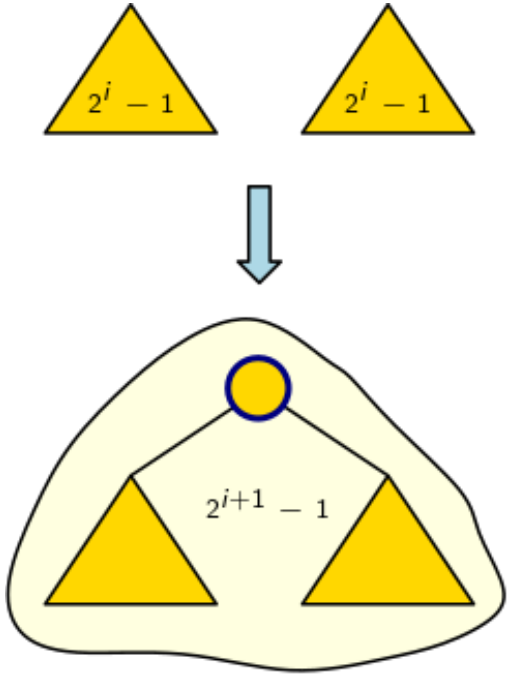


0	1	2	3	4	5
1	3	2	5	7	6

Υλοποίηση Ουράς Προτεραιότητας με “Διάνυσμα-Σωρό”

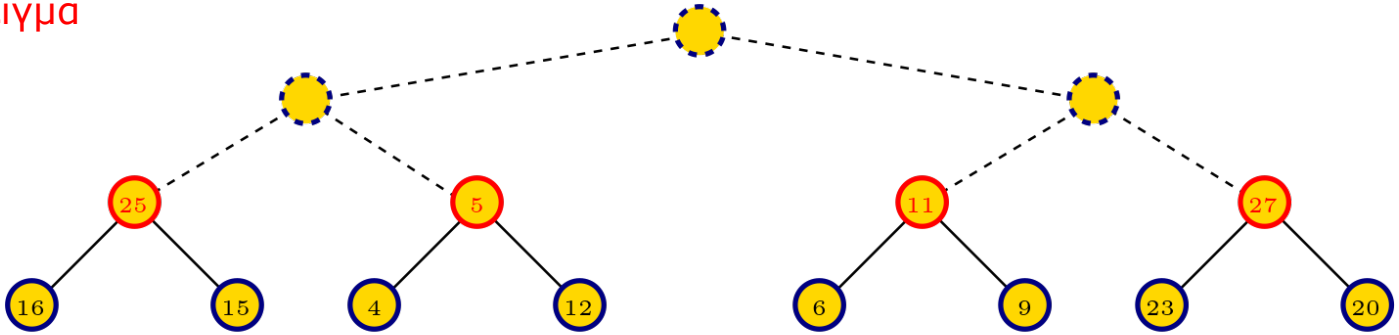
- Γρήγορος σχηματισμούς Σωρού
  - Σε ένα δέντρο για το οποίο ισχύει η ιδιότητα του πλήρους διάδικου δένδρου μπορούμε να μετακινήσουμε τα στοιχεία του έτσι ώστε να αποκτήσει και την ιδιότητα της διάταξης σωρού.

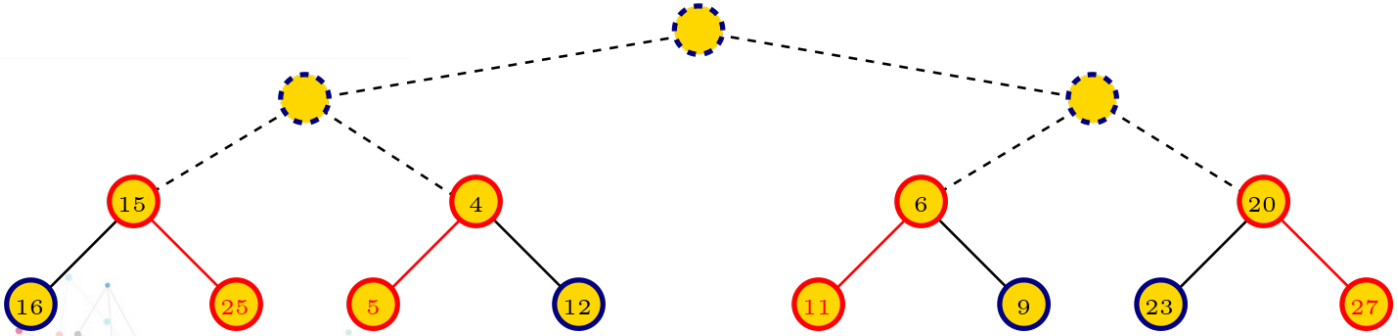
- Συγχώνευση 2 σωρών [Heap merging]
  - Έστω δυο σωροί με $2^i - 1$ στοιχεία ο καθένας και ένα νέο κλειδί k
  - Δημιουργούμε ένα νέο σωρό ο οποίος έχει $2^{i+1} + 1$ στοιχεία και ύψος $i + 1$
  - Χρήση της addRoot() [χρησιμοποιήθηκε στον `removeMin()`]
  - Ο αλγόριθμος γρήγορου σχηματισμού σωρού που βασίζεται στην συγχώνευση σωρών είναι γνωστός ως “heapify”



Υλοποίηση Ουράς Προτεραιότητας με “Διάνυσμα-Σωρό”

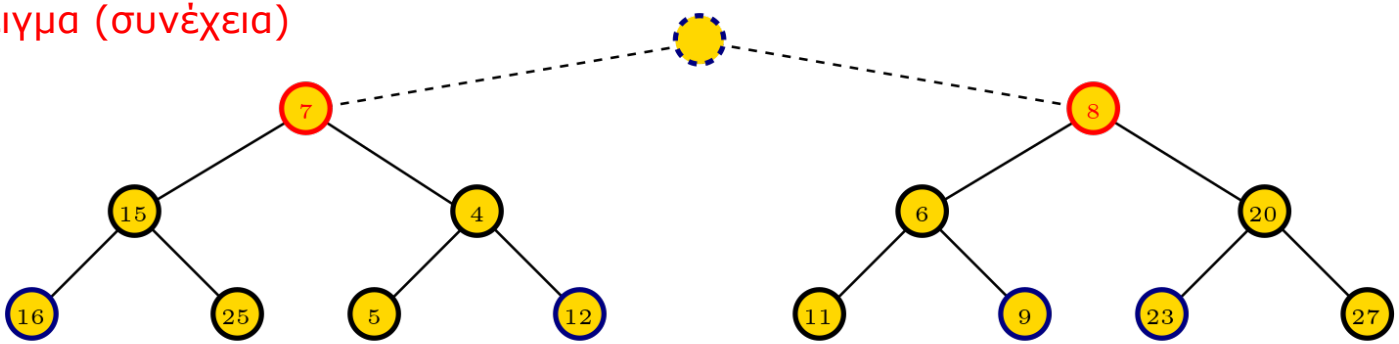
Παράδειγμα

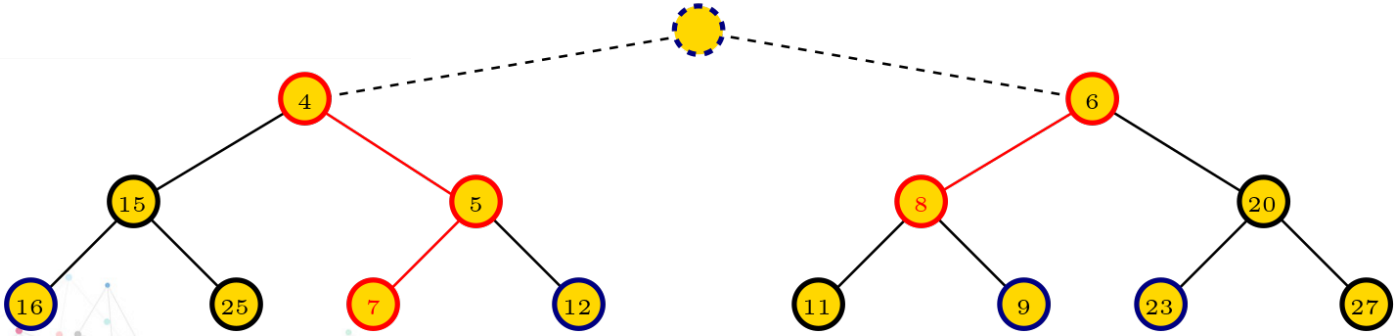




Υλοποίηση Ουράς Προτεραιότητας με “Διάνυσμα-Σωρό”

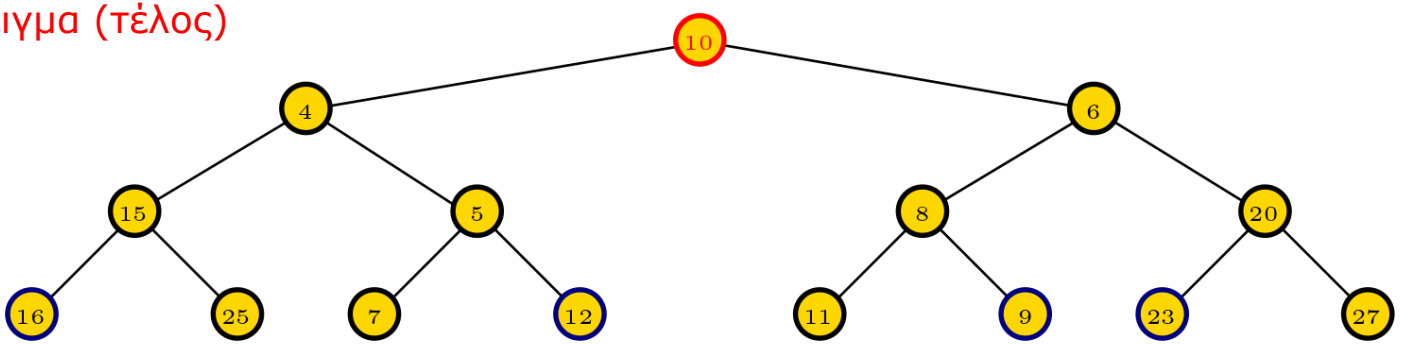
Παράδειγμα (συνέχεια)

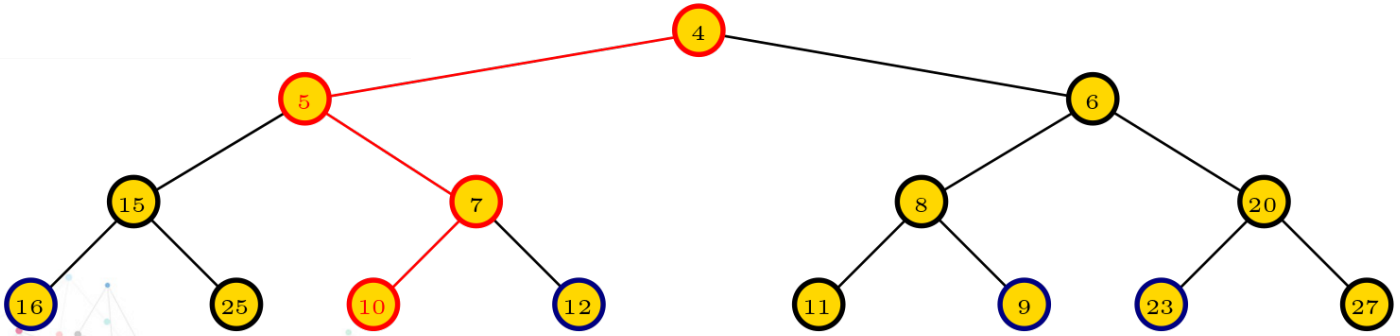




Υλοποίηση Ουράς Προτεραιότητας με “Διάνυσμα-Σωρό”

Παράδειγμα (τέλος)



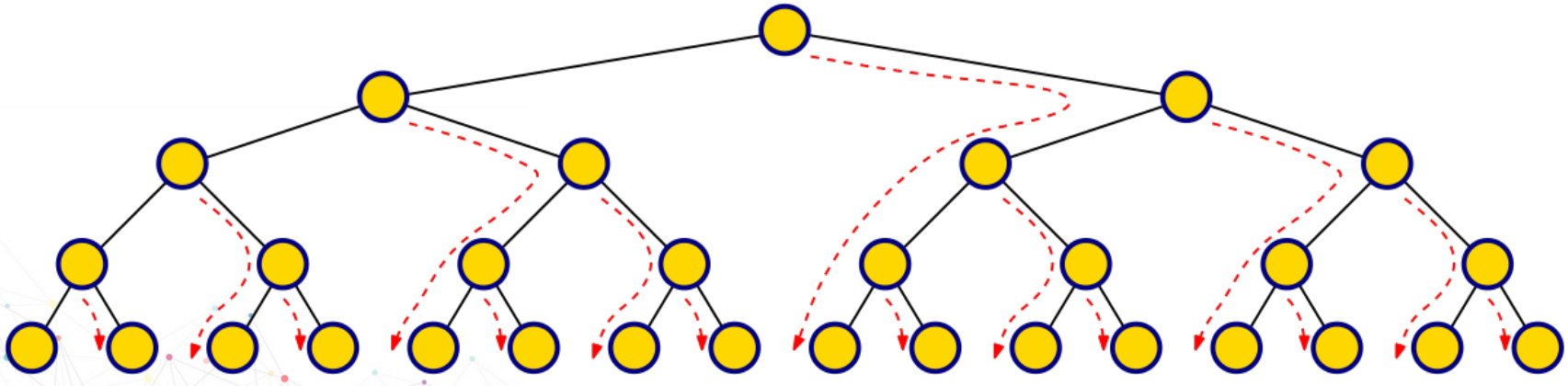


Υλοποίηση Ουράς Προτεραιότητας με “Διάνυσμα-Σωρό”

Ανάλυση:

- Οπτικοποιούμε το κόστος χειρότερης περίπτωσης μίας εντολής addRoot() με ένα μονοπάτι που ξεκινά από τον κόμβο, πηγαίνει ένα βήμα δεξιά και μετά συνέχεια αριστερά έως ότου φτάσει στα φύλλα.
- Η πραγματική κίνηση που γίνεται μπορεί να είναι διαφορετική, δεν έχει όμως μεγαλύτερο μήκος.
- Από κάθε κόμβο περνούν δυο μονοπάτια: αυτό που ξεκινά από τον ίδιο και αυτό που ήρθε από τον γονέα του.

➢ Το συνολικό κόστος επεξεργασίας της Heapify είναι $O(n)$, όπου n το πλήθος των κόμβων



Προσαρμοζόμενη Ουρά Προτεραιότητας [Adaptable PQ]

- Ο ΑΤΔ προσαρμοζόμενη ουρά προτεραιότητας επεκτείνει τον ΑΤΔ PriorityQueue
- Συλλογή από εγγραφές (αντικείμενα) τύπου `Entry`
- Επιπλέον μέθοδοι από την ουράς προτεραιότητας:
  - `remove(e)`: Διαγράφει από την συλλογή την εγγραφή `e`.
  - `replaceKey(e,k)`: Αντικαθιστά στην εγγραφή `e` το παλιό κλειδί με το `k`
  - `replaceValue(e,v)`: Αντικαθιστά στην εγγραφή `e` την παλιά τιμή(δεδομένα) με την τιμή `v`

Προσαρμοζόμενη Ουρά Προτεραιότητας [Adaptable PQ]

- Η διεπαφή `AdaptablePriorityQueue`

```
public interface AdaptablePriorityQueue<K,V> extends PriorityQueue<K,V> {

  void remove(Entry<K,V> entry) throws IllegalArgumentException;
  void replaceKey(Entry<K,V> entry, K key) throws IllegalArgumentException;
  void replaceValue(Entry<K,V> entry, V value) throws IllegalArgumentException;
}
```

- Η διεπαφή `Entry<K,V>`

```
public interface Entry<K,V> {
  K getKey();
  V getValue();
}
```

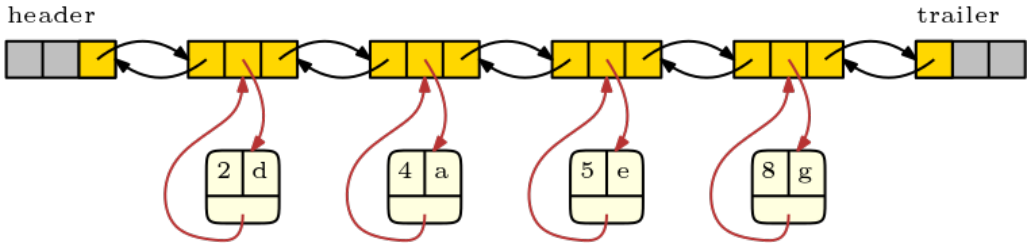
Προσαρμοζόμενη Ουρά Προτεραιότητας [Adaptable PQ]

- Εύρεση εγγραφών στην ουρά Προτεραιότητας
  - Η γρήγορη υλοποίηση των μεθόδων `remove(e)`, `replaceKey(e,k)` και `replaceValue(e,v)` απαιτεί τον γρήγορο προσδιορισμό της θέσης της εγγραφής `e` στην ουρά προτεραιότητας.
    - Μπορούμε να εντοπίσουμε τη θέση της εγγραφής `e` στην ουρά διασχίζοντας την τρέχουσα υλοποίηση (λίστα ή σωρός) αλλά αυτό απαιτεί χρόνο $O(n)$

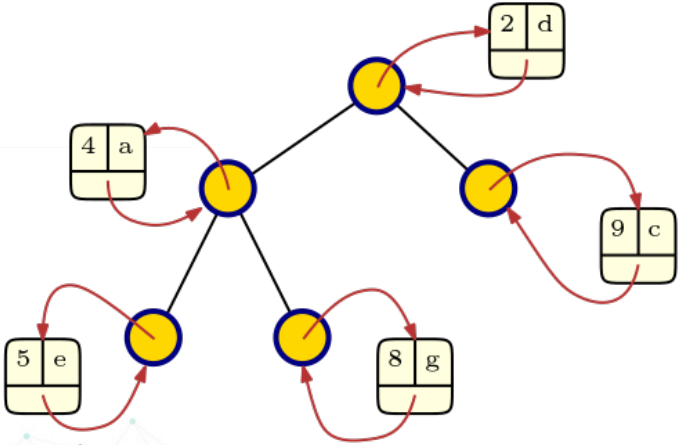
- Χρήση εγγράφων που γνωρίζουν τη θέση τους [location-aware Entries]
  - Κάθε εγγραφή γνωρίζει τη θέση της μέσα στην δομή και την διατηρεί πάντα ενήμερη.
    - `key, value`
    - `back_pointer`
  - Αντικείμενα τύπου `Entry` δημιουργούνται από την ουρά προτεραιότητας κατά την εισαγωγή εγγραφών. Συνεπώς, η ουρά προτεραιότητας μπορεί να αποθηκεύει σε αυτά πληροφορίες για την θέση τους στην τρέχουσα υλοποίηση.

Προσαρμοζόμενη Ουρά Προτεραιότητας [Adaptable PQ]

- Υλοποίηση "λίστας"



- Υλοποίηση "σωρού"



Ουρά προτεραιότητας: Σύνοψη

- Υλοποιήσεις/Απόδοση

	Υλοποίηση		
Adaptable Priority Queue	Μη-ταξινομημένη λίστα (PositionalList)	Ταξινομημένη λίστα (PositionalList)	**Σωρός (Heap)**
`insert()`	O(1)	O(n)	O(logn)
`removeMin()`	O(n)	O(1)	O(logn)
`min()`	O(n)	O(1)	O(1)
`remove()`	O(1)	O(1)	O(logn)
`replaceKey()` [decreaseKey()]	O(1)	O(n)	O(logn)
`replaceValue()`	O(1)	O(1)	O(1)
`size()`	O(1)		
`isEmpty(), isFull`			