Ουρά Προτεραιότητας (priority queue)

Δομή δεδομένων που υποστηρίζει δύο βασικές λειτουργίες :

- Εισαγωγή στοιχείου με δεδομένο κλειδί.
- Επιστροφή ενός στοιχείου με μέγιστο (ή ελάχιστο) κλειδί και διαγραφή του από τη δομή.

Αποτελεί γενίκευση των δομών της στοίβας και της ουράς.

⇨ Εξαιρετικά χρήσιμη δομή δεδομένων με πολλές εφαρμογές.

Π.χ. ταξινόμηση με χρήση ουράς προτεραιότητας.

Ουρά Προτεραιότητας (priority queue)

Δομή δεδομένων που υποστηρίζει δύο βασικές λειτουργίες :

- Εισαγωγή στοιχείου με δεδομένο κλειδί.
- Επιστροφή ενός στοιχείου με μέγιστο (ή ελάχιστο) κλειδί και διαγραφή του από τη δομή.

Αποτελεί γενίκευση των δομών της στοίβας και της ουράς.

⇨ Εξαιρετικά χρήσιμη δομή δεδομένων με πολλές εφαρμογές.
Π.χ. ταξινόμηση με χρήση ουράς προτεραιότητας.

Σε ορισμένες εφαρμογές χρειαζόμαστε επιπλέον λειτουργίες, όπως αλλαγή κλειδιού ενός στοιχείου, διαγραφή στοιχείου, ένωση δύο ουρών προτεραιότητας σε μία νέα ουρά προτεραιότητας.

Ουρά Προτεραιότητας (priority queue)

Ουρά Προτεραιότητας Μέγιστου : υποστηρίζει τις ακόλουθες λειτουργίες

`MaxPQ()`	κατασκευή ουράς προτεραιότητας
`MaxPQ(int maxN)`	κατασκευή ουράς προτεραιότητας για `maxN` κλειδιά
`MaxPQ(Key[] a)`	κατασκευή ουράς προτεραιότητας από τα κλειδιά του πίνακα `a[]`
`void insert(Key v)`	εισαγωγή κλειδιού `v`
`Key max()`	επιστροφή του μέγιστου κλειδιού
`Key delMax()`	διαγραφή και επιστροφή του μέγιστου κλειδιού
`boolean isEmpty()`	έλεγχος αν η ουρά προτεραιότητας είναι κενή
`int size()`	αριθμός κλειδιών στην ουρά προτεραιότητας

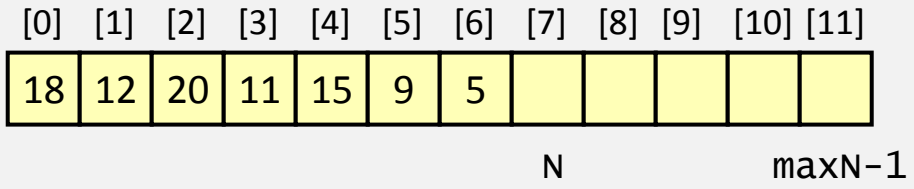
Ουρά Προτεραιότητας (priority queue)

Ουρά Προτεραιότητας Ελάχιστου : υποστηρίζει τις ακόλουθες λειτουργίες

`MinPQ()`	κατασκευή ουράς προτεραιότητας
`MinPQ(int maxN)`	κατασκευή ουράς προτεραιότητας για `maxN` κλειδιά
`MinPQ(Key[] a)`	κατασκευή ουράς προτεραιότητας από τα κλειδιά του πίνακα `a[]`
`void insert(Key v)`	εισαγωγή κλειδιού `v`
`Key min()`	επιστροφή του ελάχιστου κλειδιού
`Key delMin()`	διαγραφή και επιστροφή του ελάχιστου κλειδιού
`boolean isEmpty()`	έλεγχος αν η ουρά προτεραιότητας είναι κενή
`int size()`	αριθμός κλειδιών στην ουρά προτεραιότητας

Ουρά Προτεραιότητας (priority queue)

Στοιχειώδης υλοποίηση με μη διατεταγμένο πίνακα

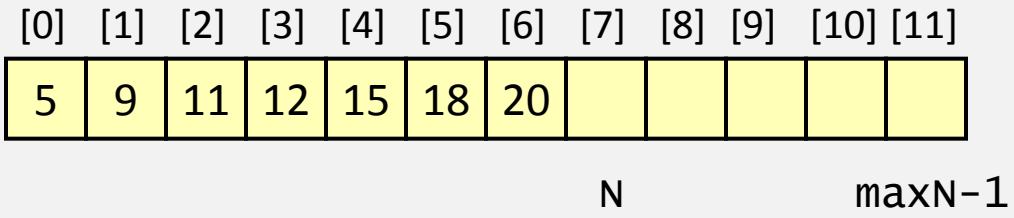



```
public class NaïveMaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N=0;
   private boolean less(int i, int j) { return pq[i].compareTo(pq[j]) < 0; }
   private void exch(int i, int j) { Key t=pq[i]; pq[i]=pq[j]; pq[j]=t; }
   NaïveMaxPQ(int maxN) { pq = (Key[]) new Comparable[maxN]; N = 0;}

   public int size() { return N; }
   public boolean isEmpty() { return N==0;}
   void insert(Key v) { pq[N++]=v; }

   Key delMax() { int j, max = 0;
     for (j = 1; j < N; j++)  if ( less(max,j) ) max = j;
     exch(pq,max,N-1); // αντταλλαγή pq[max] και pq[N-1]
     return pq[--N];
   }
}
```

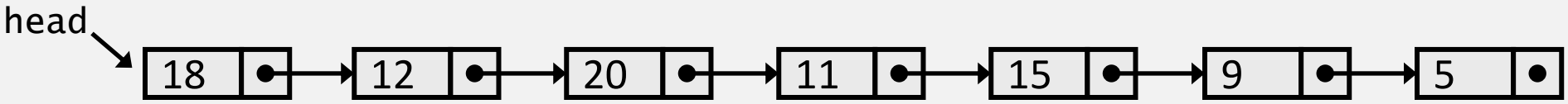
Ουρά Προτεραιότητας (priority queue)

Άλλες στοιχειώδεις υλοποιήσεις

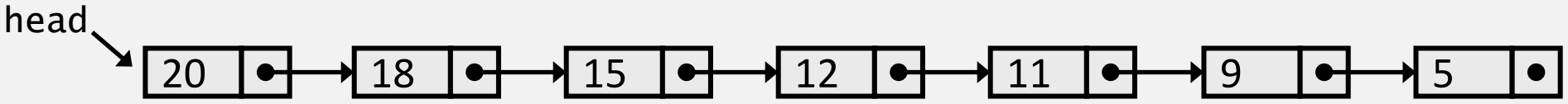
- Διατεταγμένος πίνακας: Το μέγιστο στοιχείο στην τελευταία μη κενή θέση



- Μη διατεταγμένη λίστα



- Διατεταγμένη λίστα: Το μέγιστο στοιχείο στην αρχή της λίστας



Ουρά Προτεραιότητας (priority queue)

	εισαγωγή	διαγραφή μέγιστου	διαγραφή(*)	εύρεση μέγιστου	αλλαγή προτεραιότητας	ένωση
διατεταγμένος πίνακας	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(N)$
διατεταγμένη λίστα	$O(N)$	$O(1)$	$O(1)$	$O(1)$	$O(N)$	$O(N)$
μη διατεταγμένος πίνακας	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(N)$
μη διατεταγμένη λίστα	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(1)$
σωρός	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(1)$	$O(\log N)$	$O(N)$
διωνυμική ουρά	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$
καλύτερος θεωρητικά	$O(1)$	$O(\log N)$	$O(\log N)$	$O(1)$	$O(1)$	$O(1)$

(*) Υποθέτει ότι γνωρίζουμε τη θέση του στοιχείου που διαγράφεται

Ουρά Προτεραιότητας (priority queue)

	εισαγωγή	διαγραφή μέγιστου	διαγραφή(*)	εύρεση μέγιστου	αλλαγή προτεραιότητας	ένωση
διατεταγμένος πίνακας	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(N)$
διατεταγμένη λίστα	$O(N)$	$O(1)$	$O(1)^{(**)}$	$O(1)$	$O(N)$	$O(N)$
μη διατεταγμένος πίνακας	$O(1)$	$O(N)$	$O(1)$	$O(N)$	$O(1)$	$O(N)$
μη διατεταγμένη λίστα	$O(1)$	$O(N)$	$O(1)^{(**)}$	$O(N)$	$O(1)$	$O(1)^{(**)}$
σωρός	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(1)$	$O(\log N)$	$O(N)$
διωνυμική ουρά	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$
καλύτερος θεωρητικά	$O(1)$	$O(\log N)$	$O(\log N)$	$O(1)$	$O(1)$	$O(1)$

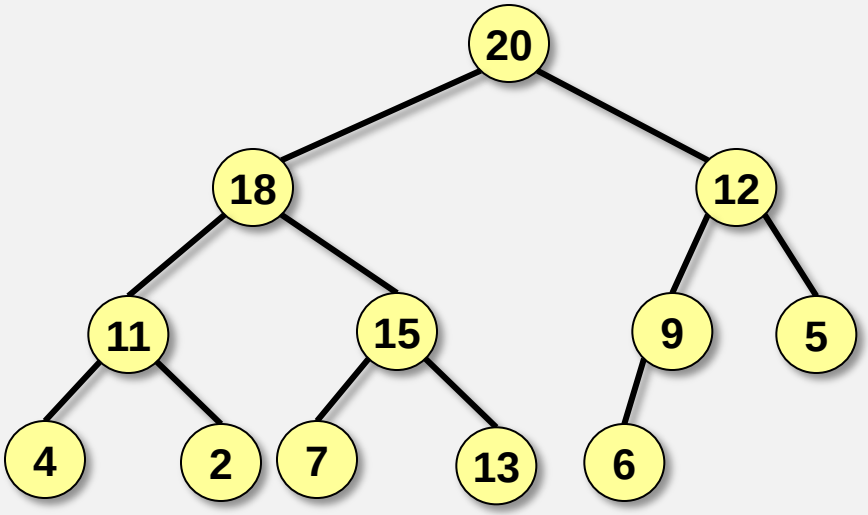
(*) Υποθέτει ότι γνωρίζουμε τη θέση του στοιχείου που διαγράφεται

(**) Απαιτεί πιο σύνθετη συνδεδεμένη λίστα, π.χ. διπλή λίστα

Δομή Δεδομένων Σωρού (heap)

Υποστηρίζει αποδοτικά τις λειτουργίες μιας ουράς προτεραιότητας.

Αναπαράσταση ως πλήρες δυαδικό δένδρο: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

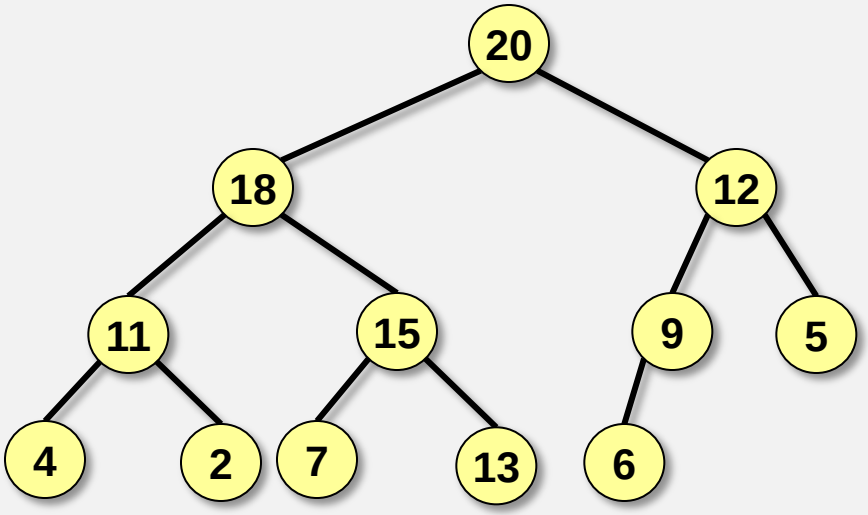


Δομή Δεδομένων Σωρού (heap)

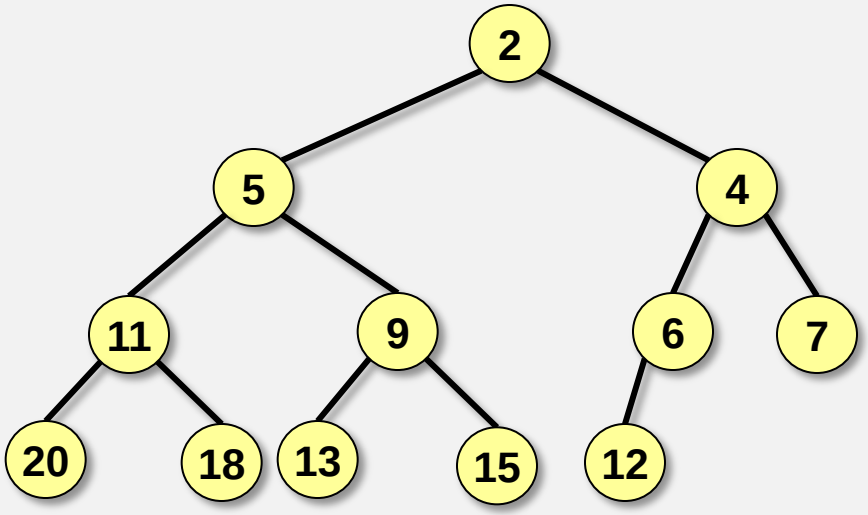
Υποστηρίζει αποδοτικά τις λειτουργίες μιας ουράς προτεραιότητας.

Σωρός μέγιστου: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

Σωρός ελάχιστου: κάθε κόμβος έχει κλειδί μεγαλύτερο ή ίσο με το κλειδί του γονέα του.



σωρός μέγιστου : γρήγορη εξαγωγή μέγιστου στοιχείου

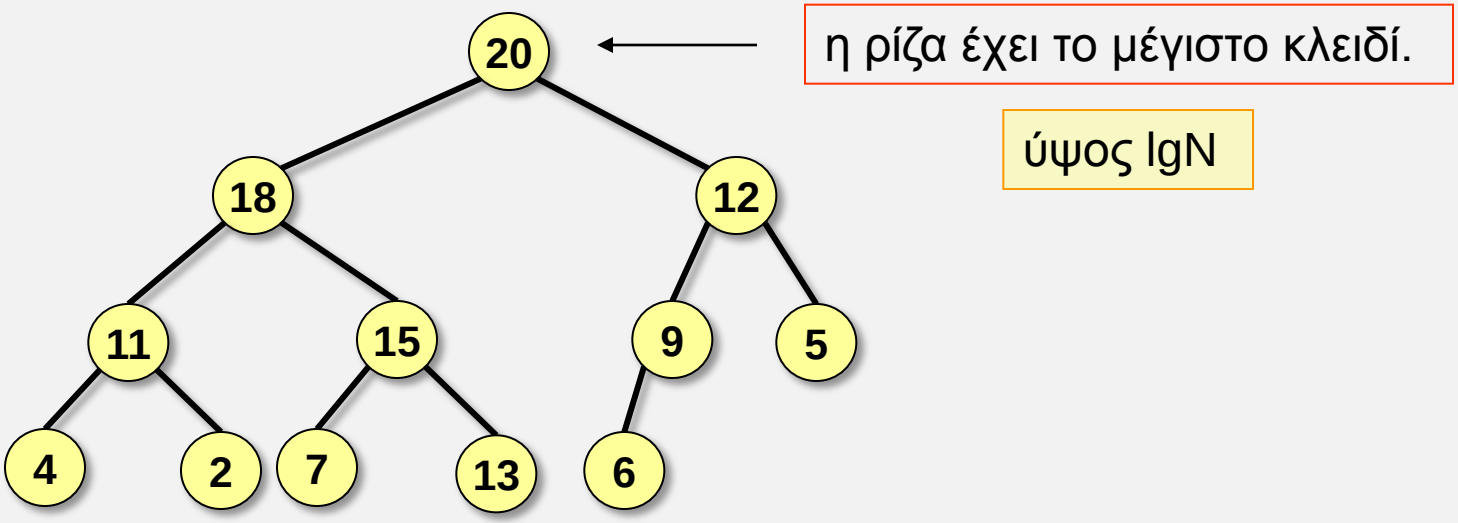


σωρός ελάχιστου : γρήγορη εξαγωγή ελάχιστου στοιχείου

Δομή Δεδομένων Σωρού (heap)

Υποστηρίζει αποδοτικά τις λειτουργίες μιας ουράς προτεραιότητας.

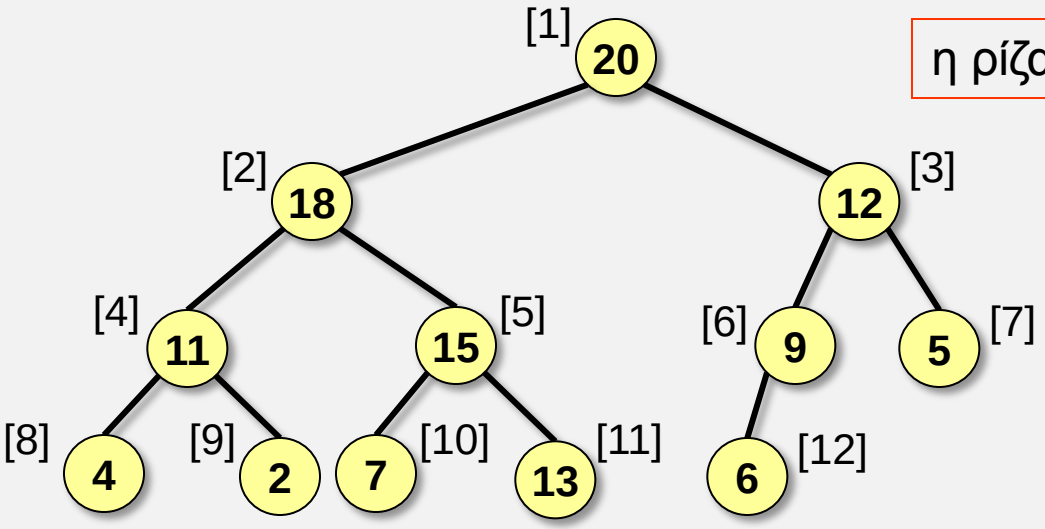
Αναπαράσταση ως πλήρες δυαδικό δένδρο: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.



Δομή Δεδομένων Σωρού (heap)

Υποστηρίζει αποδοτικά τις λειτουργίες μιας ουράς προτεραιότητας.

Αναπαράσταση ως πλήρες δυαδικό δένδρο: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.



η ρίζα έχει το μέγιστο κλειδί.

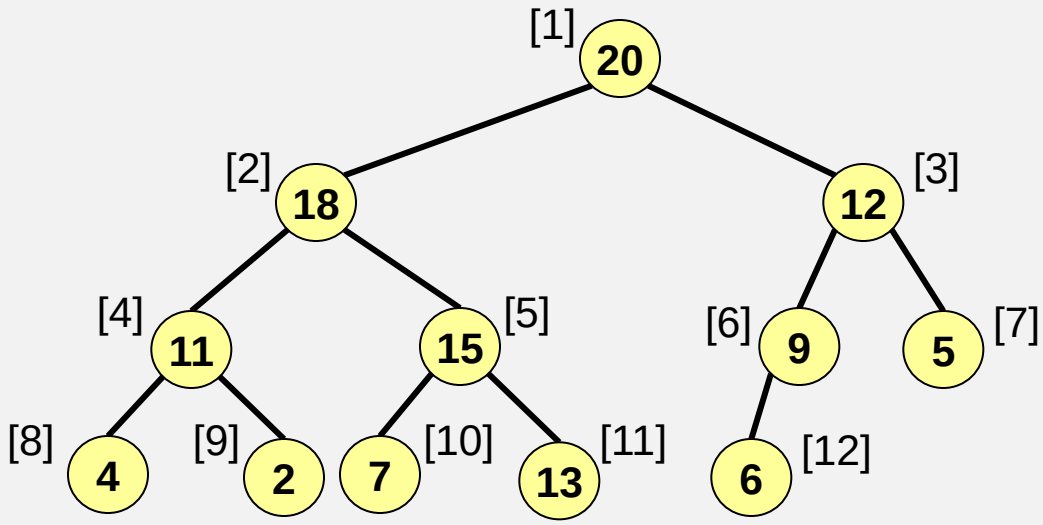
ύψος lgN

Υλοποίηση με πίνακα: το στοιχείο στη θέση i είναι ο γονέας των στοιχείων στις θέσεις 2i και 2i+1.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
20	18	12	11	15	9	5	4	2	7	13	6

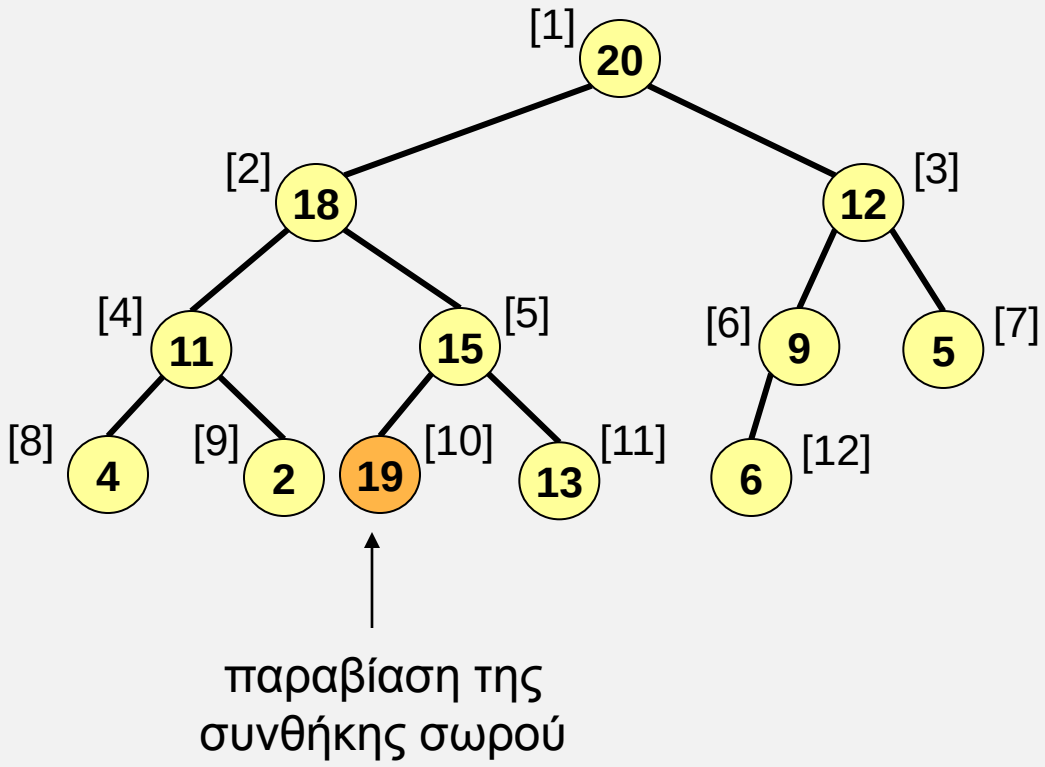
Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.



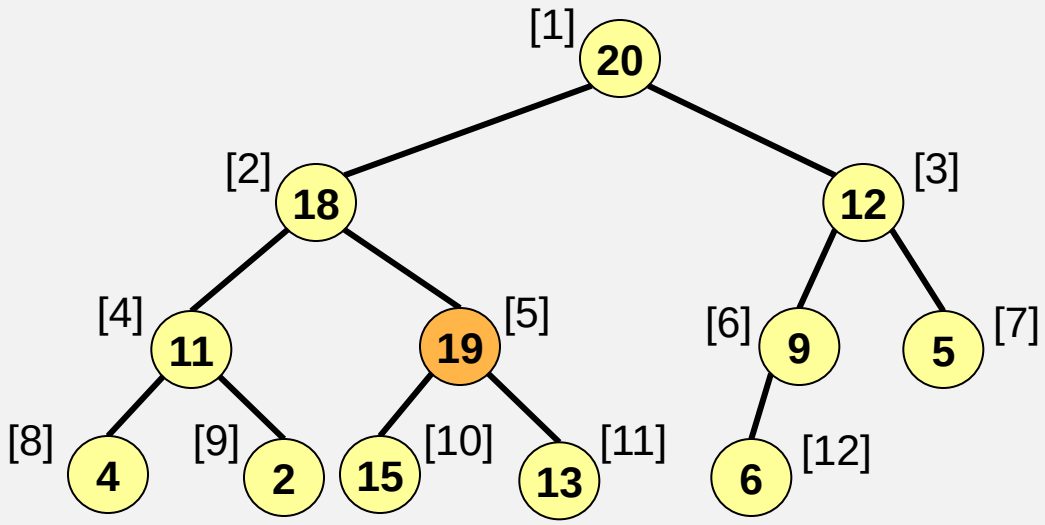
Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.



Αλγόριθμοι σε Σωρούς

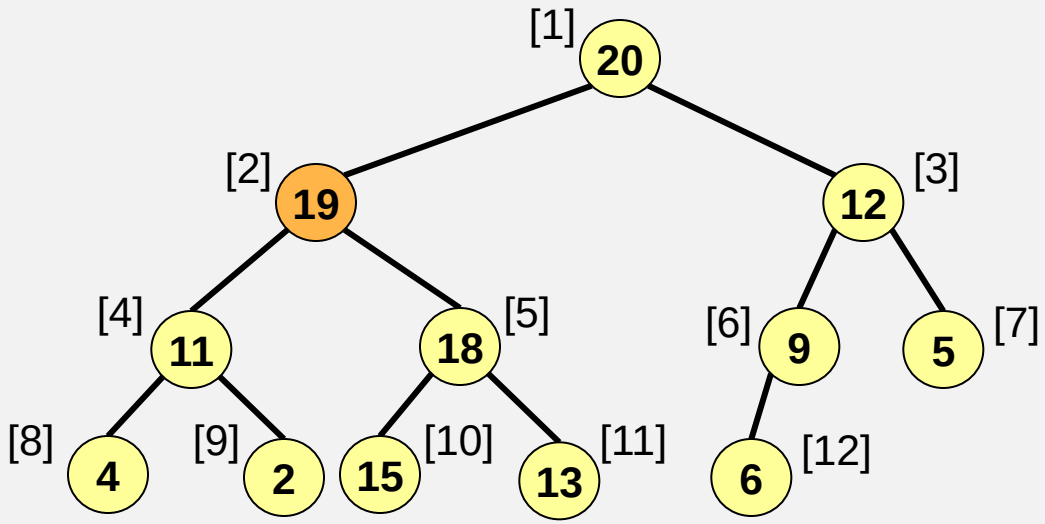
Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.



αντιμετάθεση με
το γονέα

Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

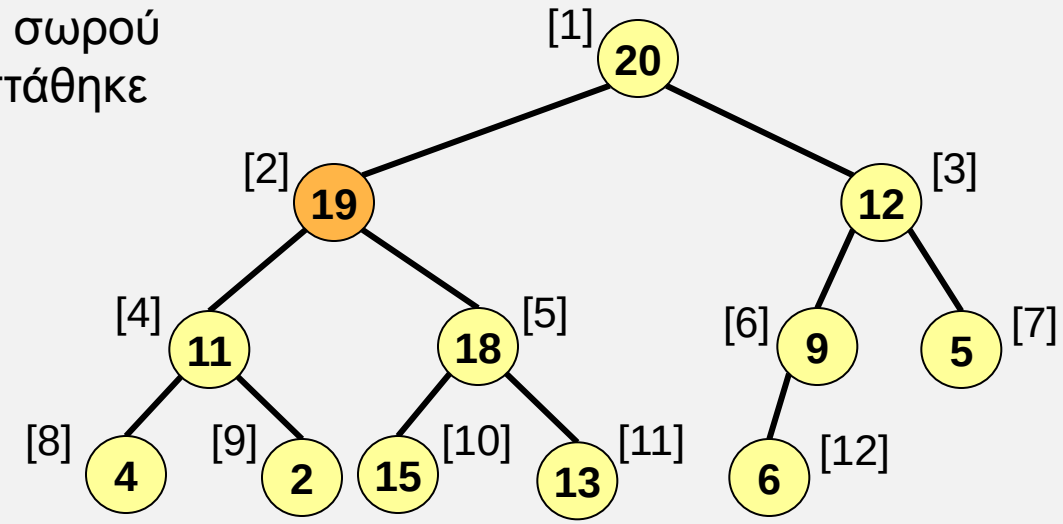


αντιμετάθεση με
το γονέα

Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

η συνθήκη σωρού αποκαταστάθηκε

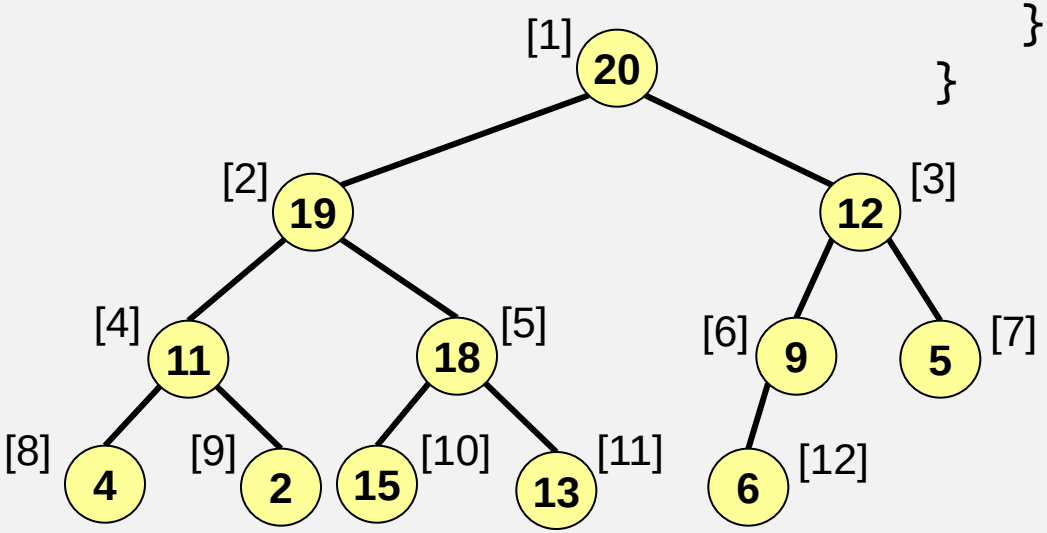


αντιμετάθεση με το γονέα

Αλγόριθμοι σε Σωρούς

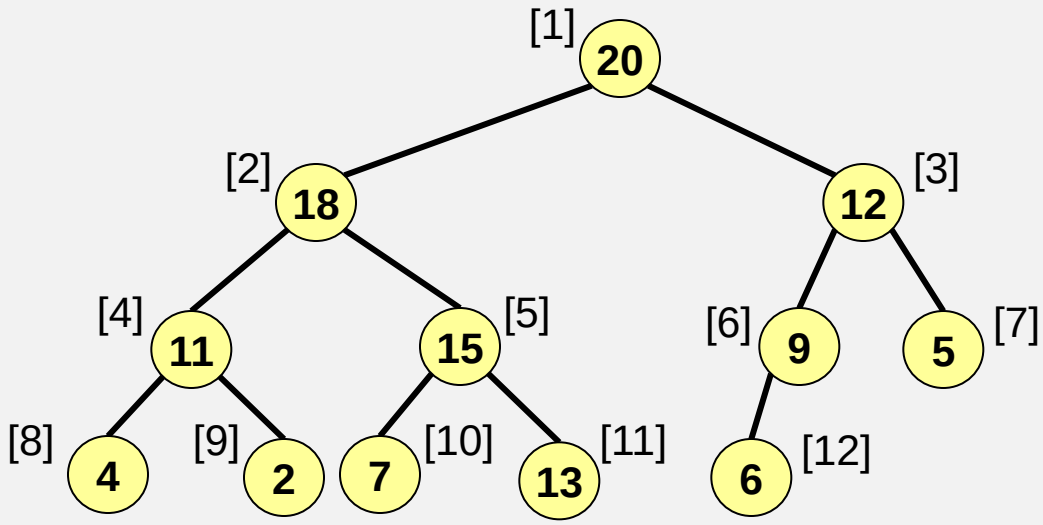
Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

```
private void fixUp(int k)
{
   while (k>1 && less(k/2, k)) {
      exch(k,k/2);
      k=k/2;
   }
}
```



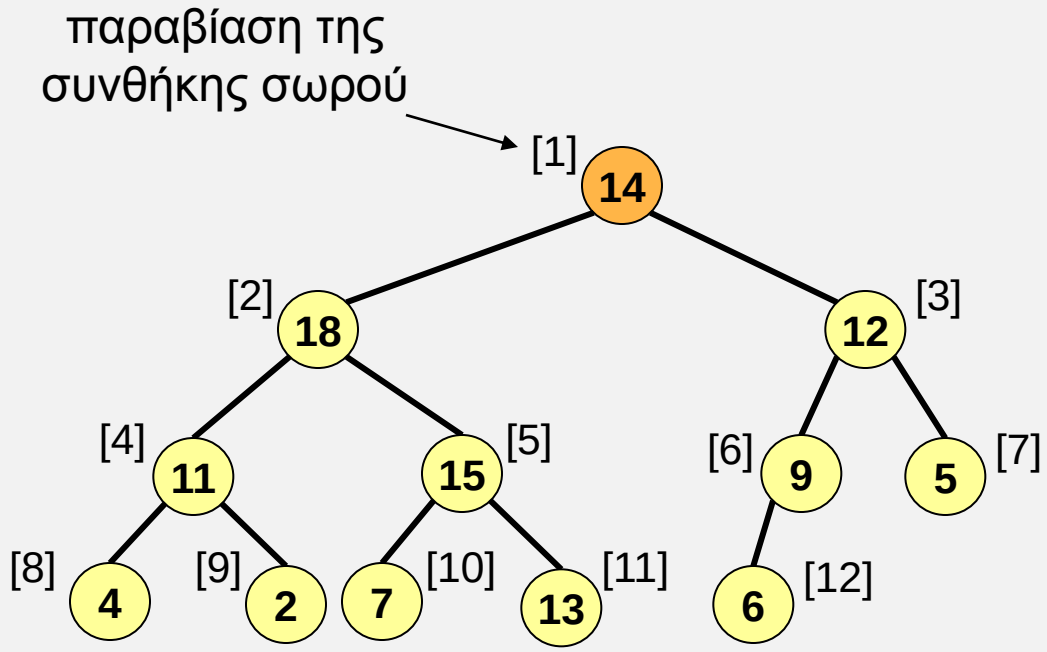
Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.



Αλγόριθμοι σε Σωρούς

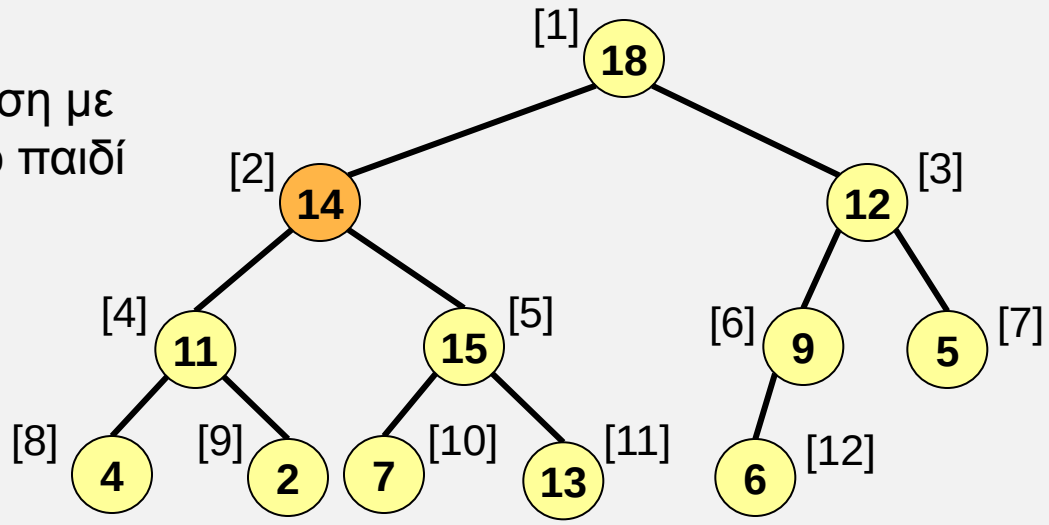
Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.



Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

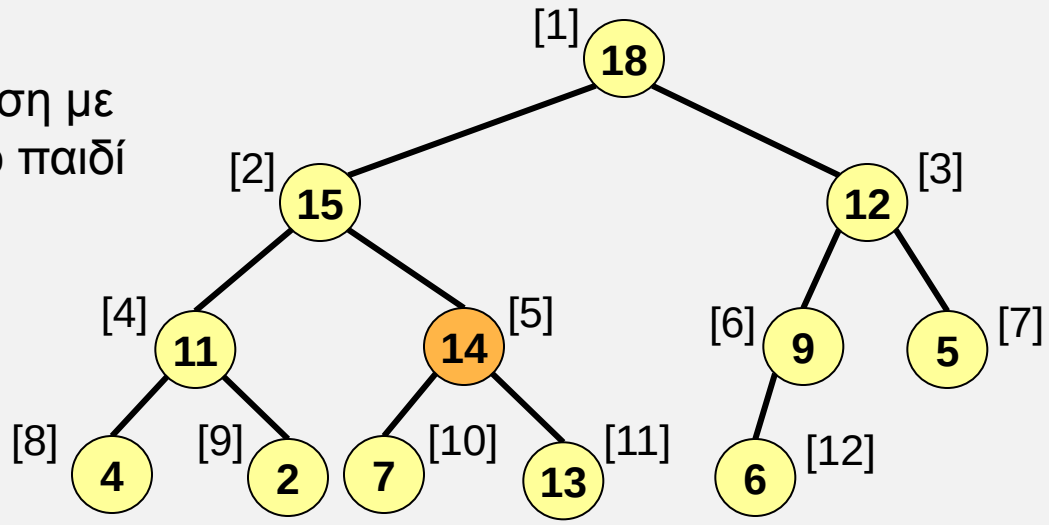
αντιμετάθεση με μεγαλύτερο παιδί



Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

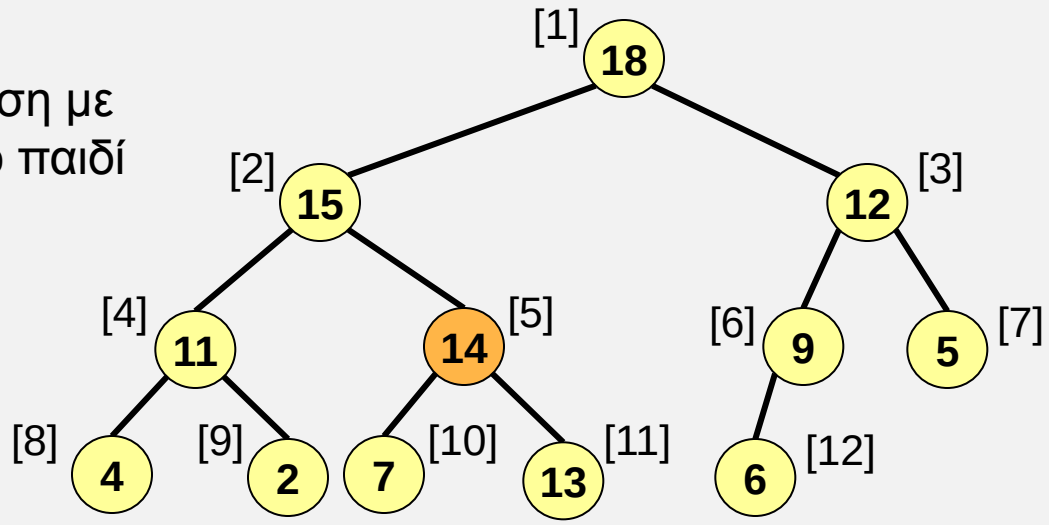
αντιμετάθεση με μεγαλύτερο παιδί



Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

αντιμετάθεση με μεγαλύτερο παιδί

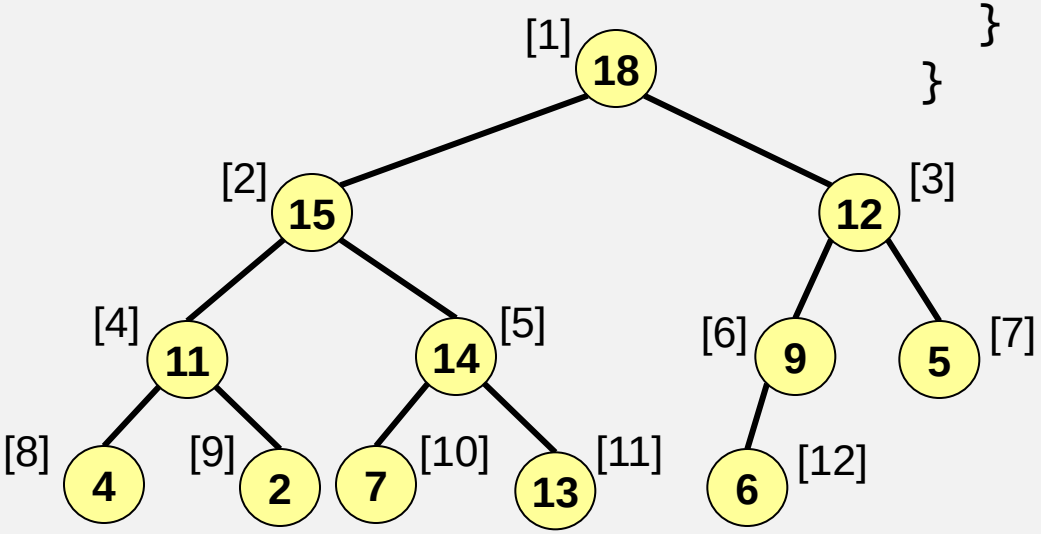


η συνθήκη σωρού αποκαταστάθηκε

Αλγόριθμοι σε Σωρούς

Συνθήκη σωρού: κάθε κόμβος έχει κλειδί μικρότερο ή ίσο με το κλειδί του γονέα του.

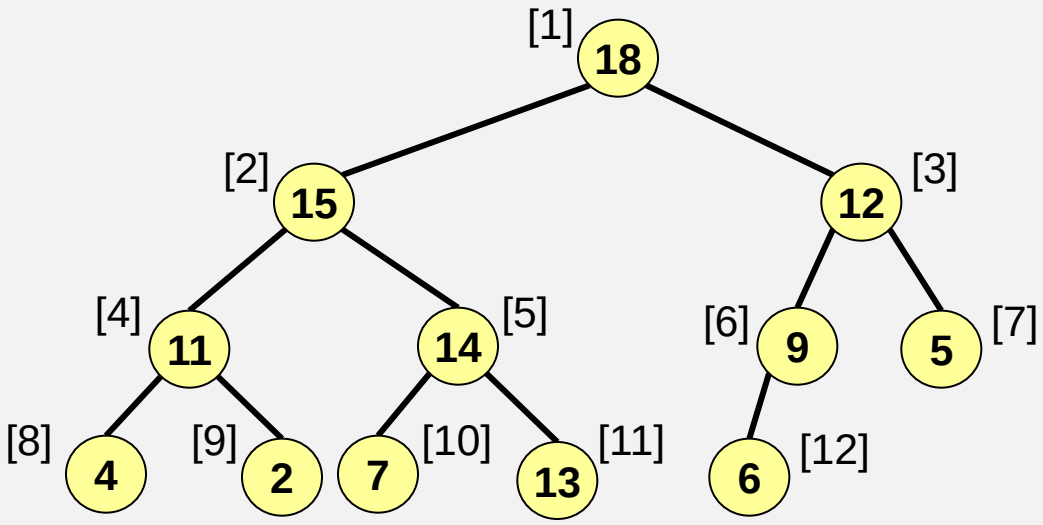
```
private void fixDown(int k)
{ int j;
  while (2*k <= N)
  { j=2*k;
    if (j<N && less(j,j+1)) j++;
    if (!less(k,j)) break;
    exch(k,j); k=j;
  }
}
```



Αλγόριθμοι σε Σωρούς

Ουρά προτεραιότητας βασισμένη σε σωρό

```
public class MaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N = 0;

   MaxPQ(int maxN)
   { pq = (Key[]) new Comparable[maxN+1]; }

   public void insert(Key v)
   { pq[++N]=v; fixUp(N); }

   public Key delMax()
   {
      Key max = pq[1];
      exch(1,N);
      pq[N--] = null;
      fixDown(1);
      return max;
   }
}
```

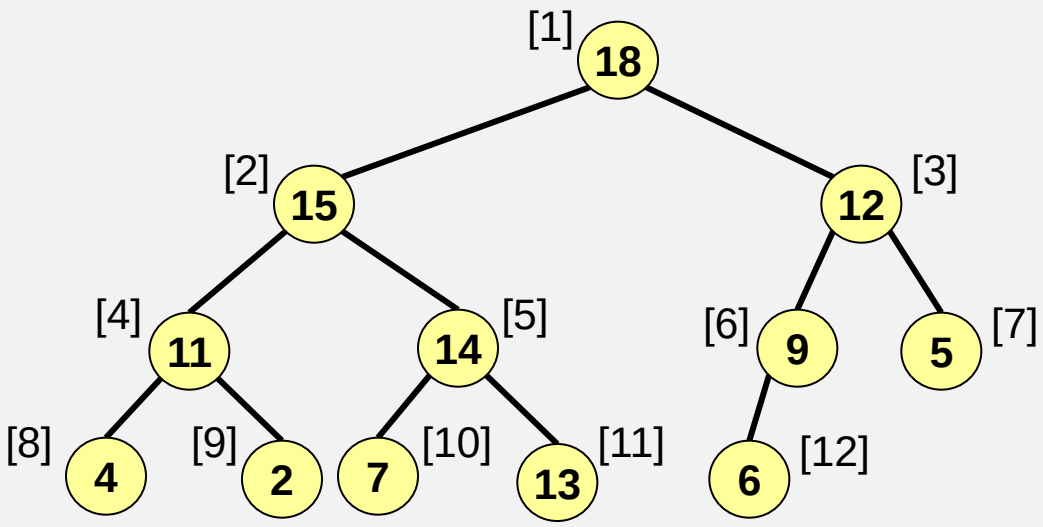


Αλγόριθμοι σε Σωρούς

Ουρά προτεραιότητας βασισμένη σε σωρό

```
public class MaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N = 0;

   MaxPQ(int maxN)
   { pq = (Key[]) new Comparable[maxN+1]; }

   public void insert(Key v)
   { pq[++N]=v; fixUp(N); }

   public Key delMax()
   {
      Key max = pq[1];
      exch(1,N);
      pq[N--] = null;
      fixDown(1);
      return max;
   }
}
```

διαγραφή μέγιστου

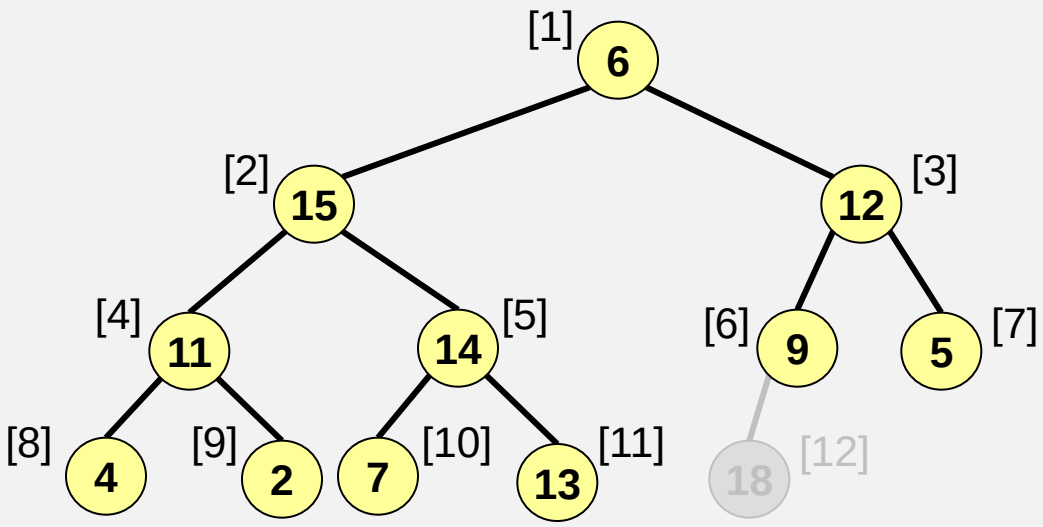


Αλγόριθμοι σε Σωρούς

Ουρά προτεραιότητας βασισμένη σε σωρό

```
public class MaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N = 0;

   MaxPQ(int maxN)
   { pq = (Key[]) new Comparable[maxN+1]; }

   public void insert(Key v)
   { pq[++N]=v; fixUp(N); }

   public Key delMax()
   {
      Key max = pq[1];
      exch(1,N);
      pq[N--] = null;
      fixDown(1);
      return max;
   }
}
```

διαγραφή μέγιστου

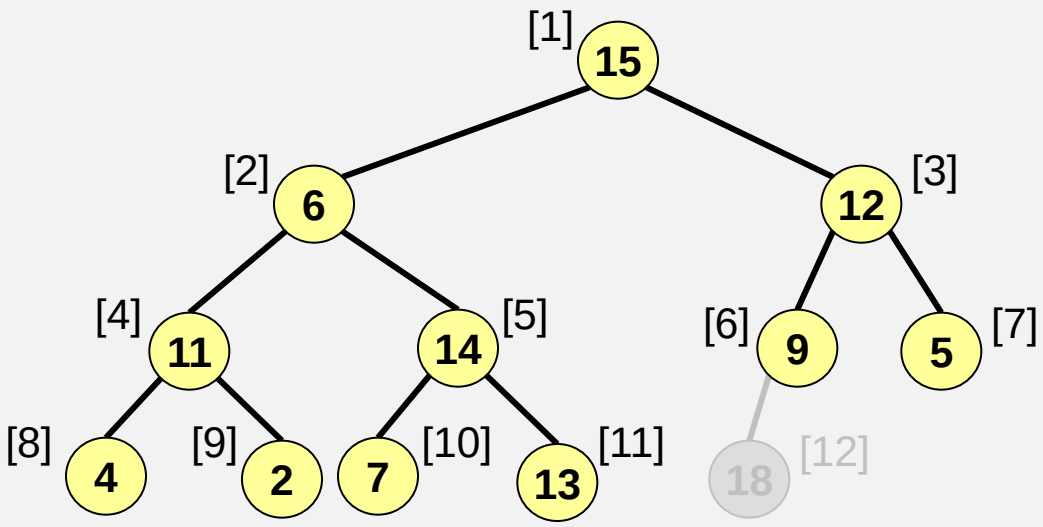


Αλγόριθμοι σε Σωρούς

Ουρά προτεραιότητας βασισμένη σε σωρό

```
public class MaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N = 0;

   MaxPQ(int maxN)
   { pq = (Key[]) new Comparable[maxN+1]; }

   public void insert(Key v)
   { pq[++N]=v; fixUp(N); }

   public Key delMax()
   {
      Key max = pq[1];
      exch(1,N);
      pq[N--] = null;
      fixDown(1);
      return max;
   }
}
```

διαγραφή μέγιστου

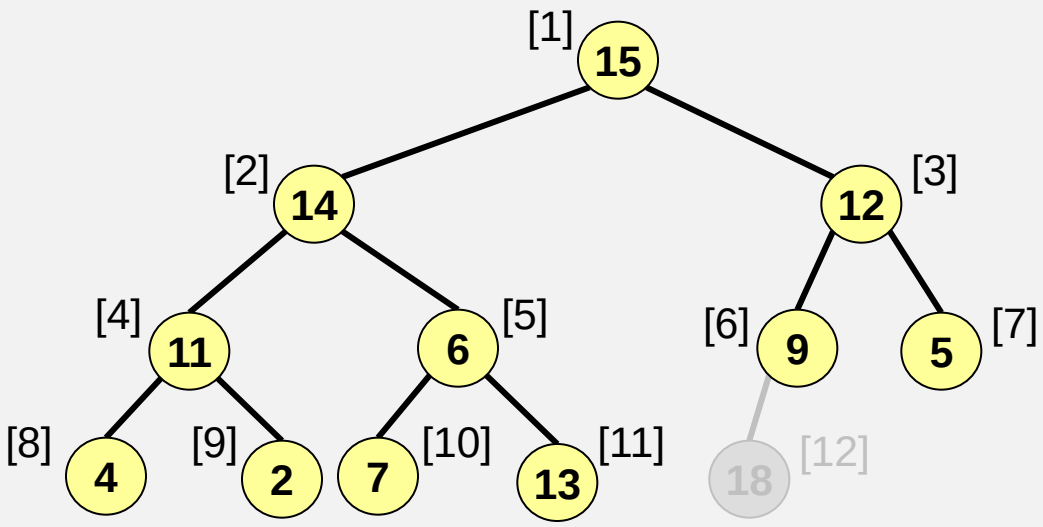


Αλγόριθμοι σε Σωρούς

Ουρά προτεραιότητας βασισμένη σε σωρό

```
public class MaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N = 0;

   MaxPQ(int maxN)
   { pq = (Key[]) new Comparable[maxN+1]; }

   public void insert(Key v)
   { pq[++N]=v; fixUp(N); }

   public Key delMax()
   {
      Key max = pq[1];
      exch(1,N);
      pq[N--] = null;
      fixDown(1);
      return max;
   }
}
```

διαγραφή μέγιστου

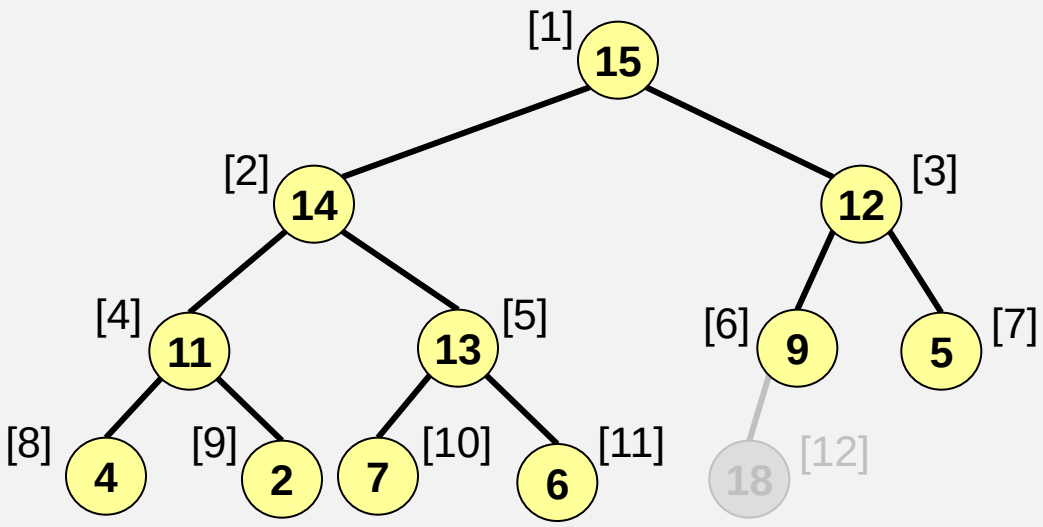


Αλγόριθμοι σε Σωρούς

Ουρά προτεραιότητας βασισμένη σε σωρό

```
public class MaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N = 0;

   MaxPQ(int maxN)
   { pq = (Key[]) new Comparable[maxN+1]; }

   public void insert(Key v)
   { pq[++N]=v; fixUp(N); }

   public Key delMax()
   {
      Key max = pq[1];
      exch(1,N);
      pq[N--] = null;
      fixDown(1);
      return max;
   }
}
```

διαγραφή μέγιστου

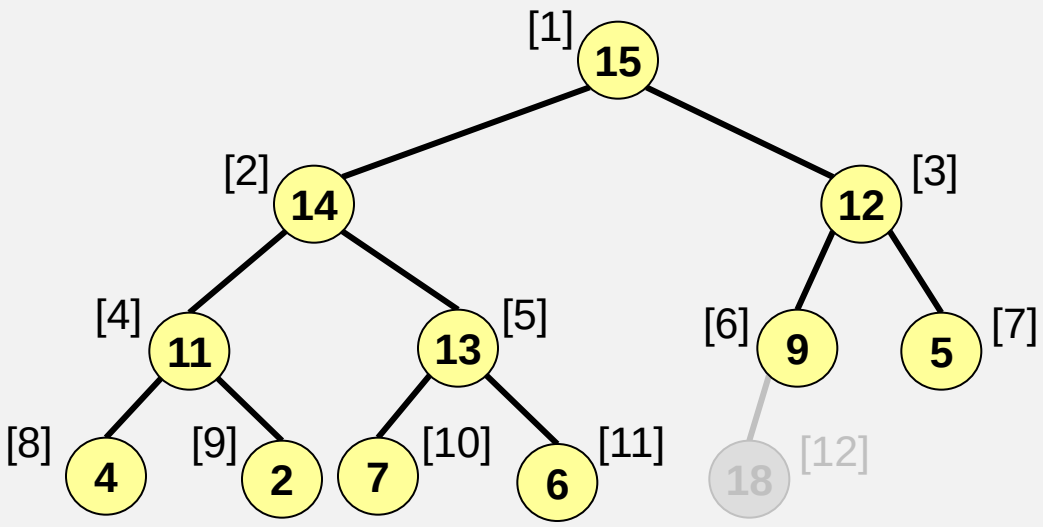


Αλγόριθμοι σε Σωρούς

Ουρά προτεραιότητας βασισμένη σε σωρό

```
public class MaxPQ<Key extends Comparable<Key>> {
   private Key[] pq; private int N = 0;

   MaxPQ(int maxN)
   { pq = (Key[]) new Comparable[maxN+1]; }

   public void insert(Key v)
   { pq[++N]=v; fixUp(N); }

   public Key delMax()
   {
      Key max = pq[1];
      exch(1,N);
      pq[N--] = null;
      fixDown(1);
      return max;
   }
}
```

διαγραφή μέγιστου



Αλγόριθμοι σε Σωρούς

Ταξινόμηση με ουρά προτεραιότητας

```
public static void PQsort(Comparable a[])
{ int N = a.length;
  MaxPQ pq = new MaxPQ(N);
  for (int k=0; k<N; k++) pq.insert(a[k]);
  for (int k=N-1; k>=0; k--) a[k]=pq.delMax();
}
```

Αλγόριθμοι σε Σωρούς

Ταξινόμηση με ουρά προτεραιότητας

```
public static void PQsort(Comparable a[])
{ int N = a.length;
  MaxPQ pq = new MaxPQ(N);
  for (int k=0; k<N; k++) pq.insert(a[k]);
  for (int k=N-1; k>=0; k--) a[k]=pq.delMax();
}
```

Διαδοχική εισαγωγή των στοιχείων στην ουρά
$\Rightarrow O(N \cdot \log N)$ χρόνος

Αλγόριθμοι σε Σωρούς

Ταξινόμηση με ουρά προτεραιότητας

```
public static void PQsort(Comparable a[])
{ int N = a.length;
  MaxPQ pq = new MaxPQ(N);
  for (int k=0; k<N; k++) pq.insert(a[k]);
  for (int k=N-1; k>=0; k--) a[k]=pq.delMax();
}
```

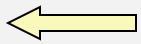
Διαδοχική εισαγωγή των στοιχείων στην ουρά
$\Rightarrow$ O(N·logN) χρόνος

Διαδοχική εξαγωγή μέγιστου στοιχείου
$\Rightarrow$ O(N·logN) χρόνος

Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```



Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) $\Rightarrow$ O(N) χρόνος

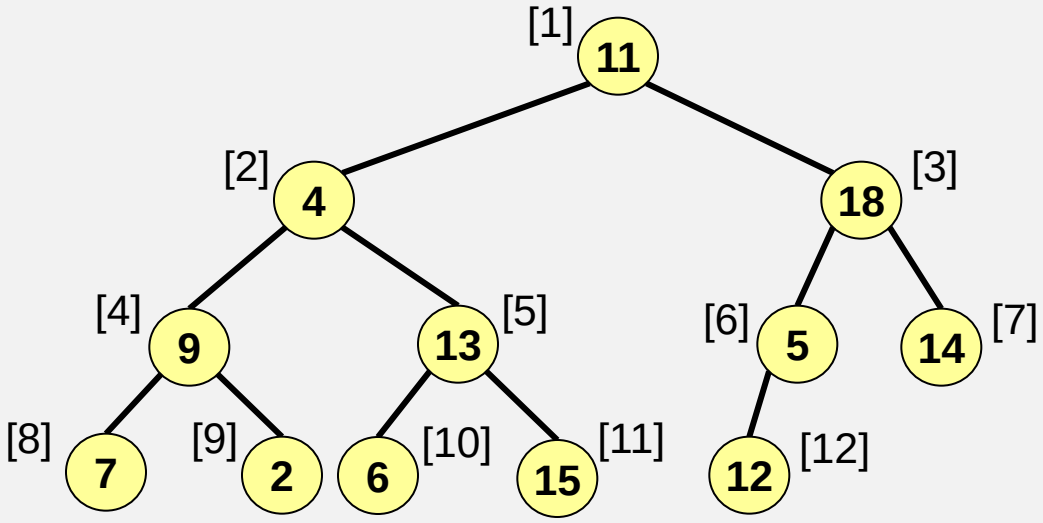
Ταξινομεί σε αύξουσα σειρά τα στοιχεία `a[1]` έως `a[N]`. Χρησιμοποιούμε παραλλαγές των μεθόδων `fixDown`, `fixUp` και `exch`, οι οποίες δέχονται τον πίνακα `a[]` και το `N` ως ορίσματα.

Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

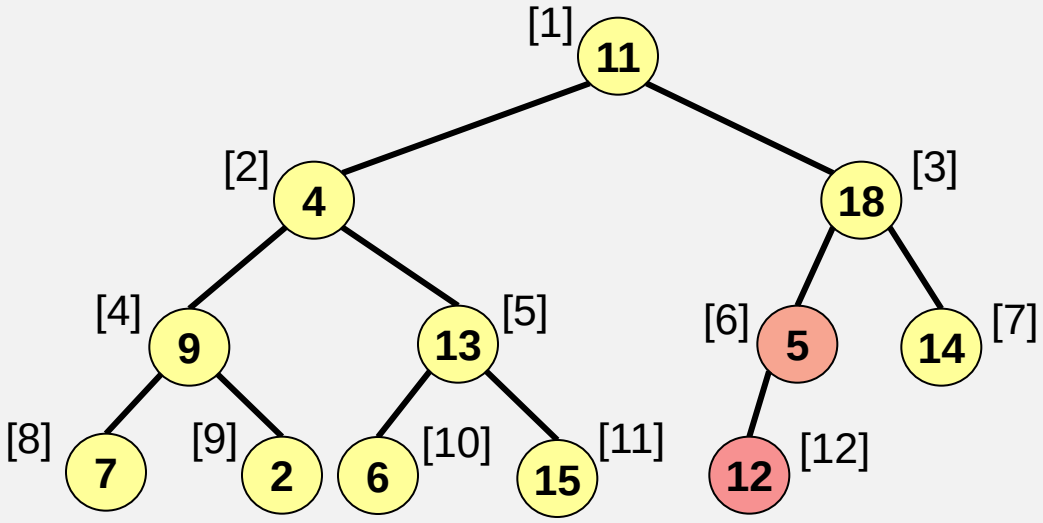


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

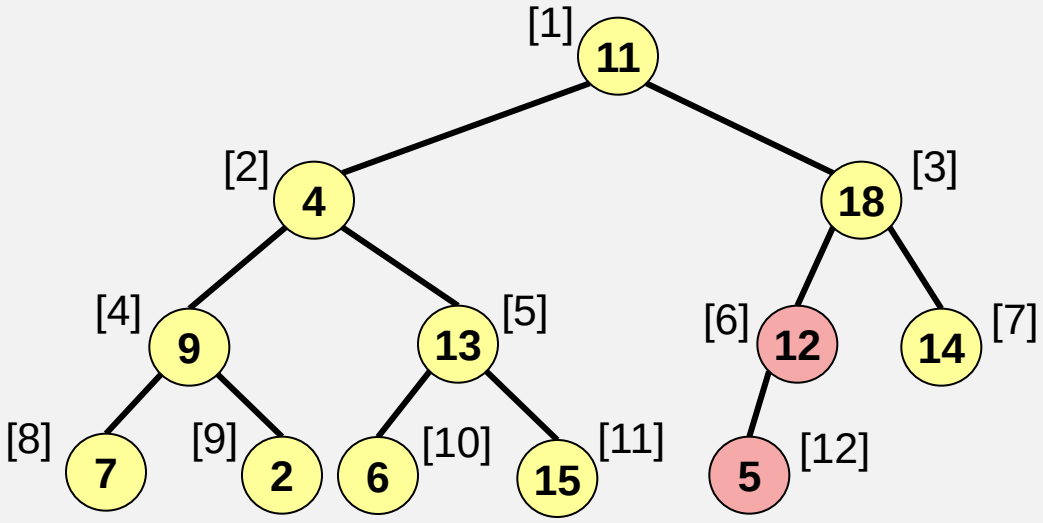


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

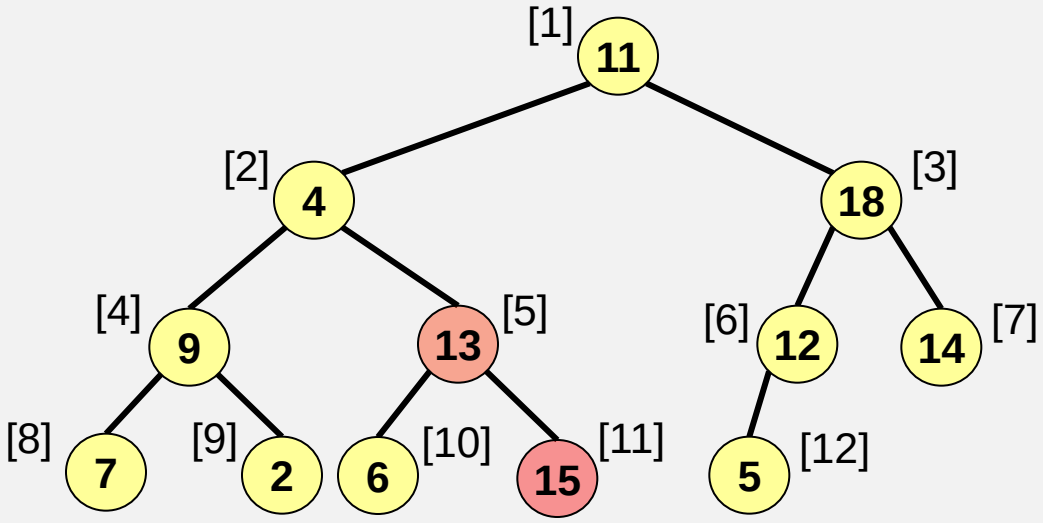


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

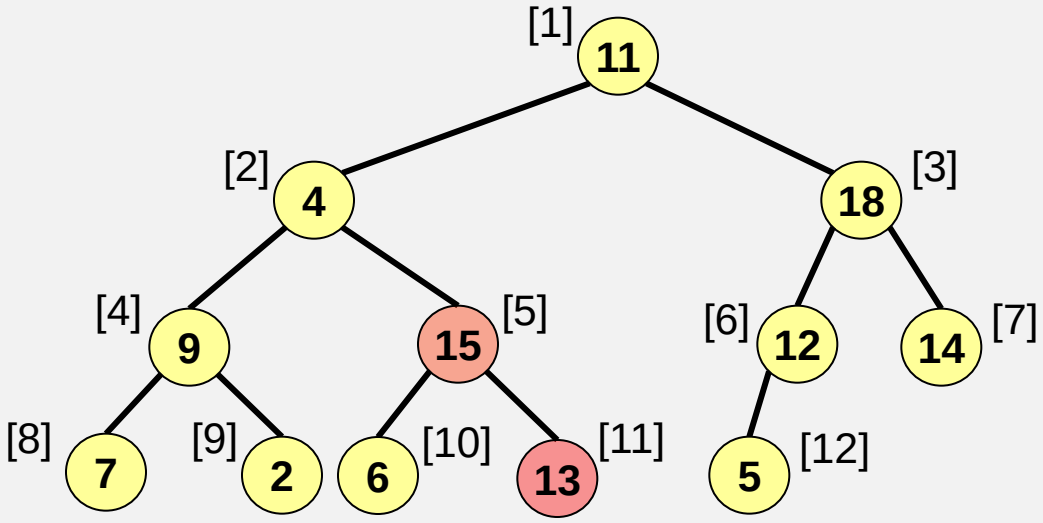


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

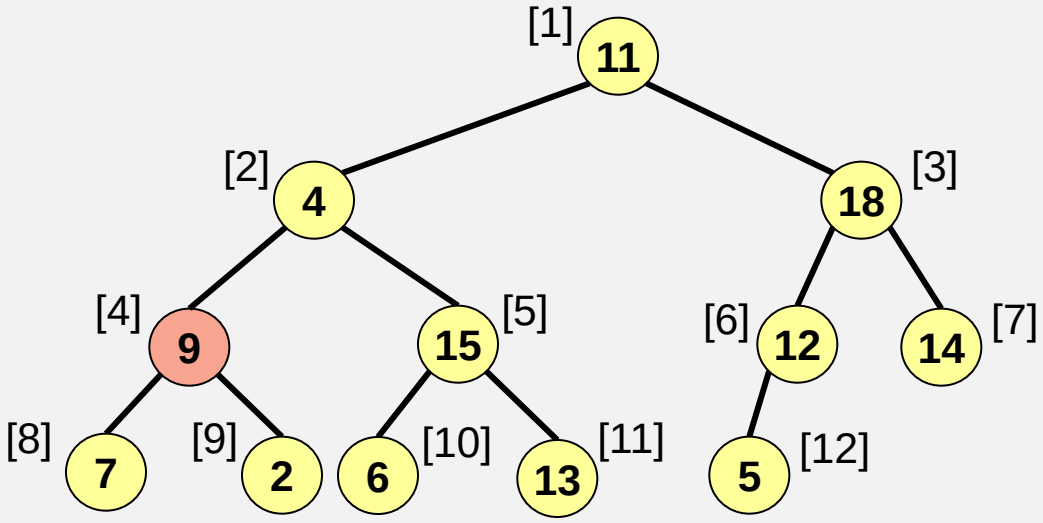


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

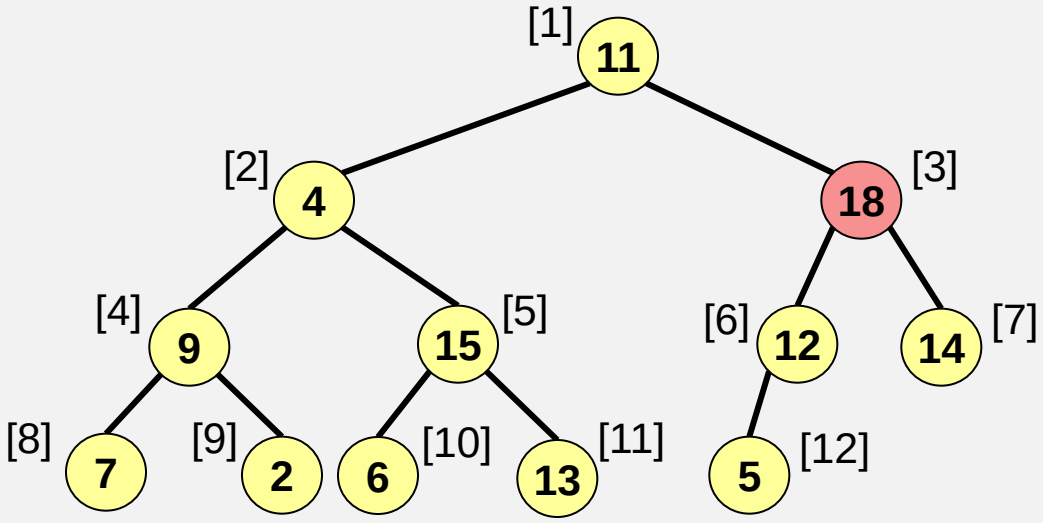


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

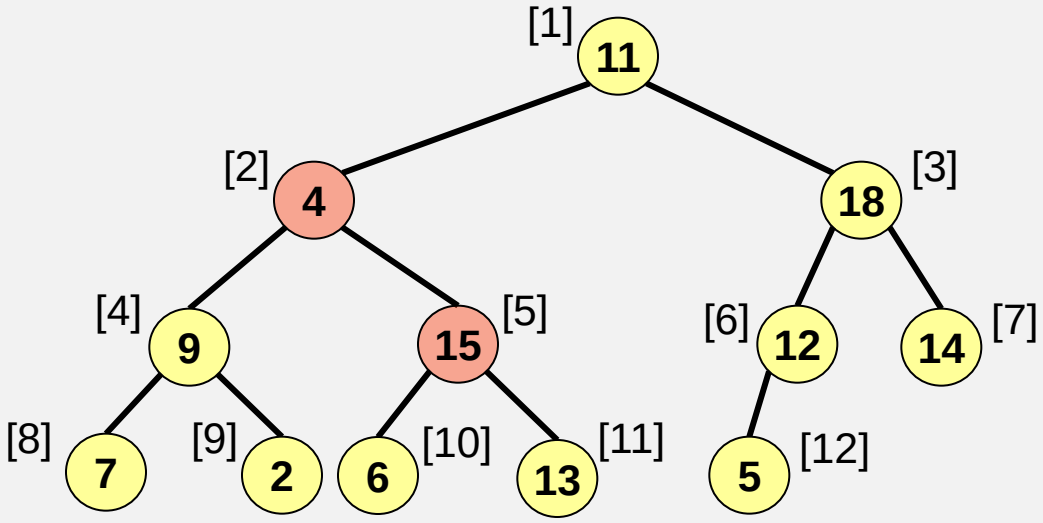


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

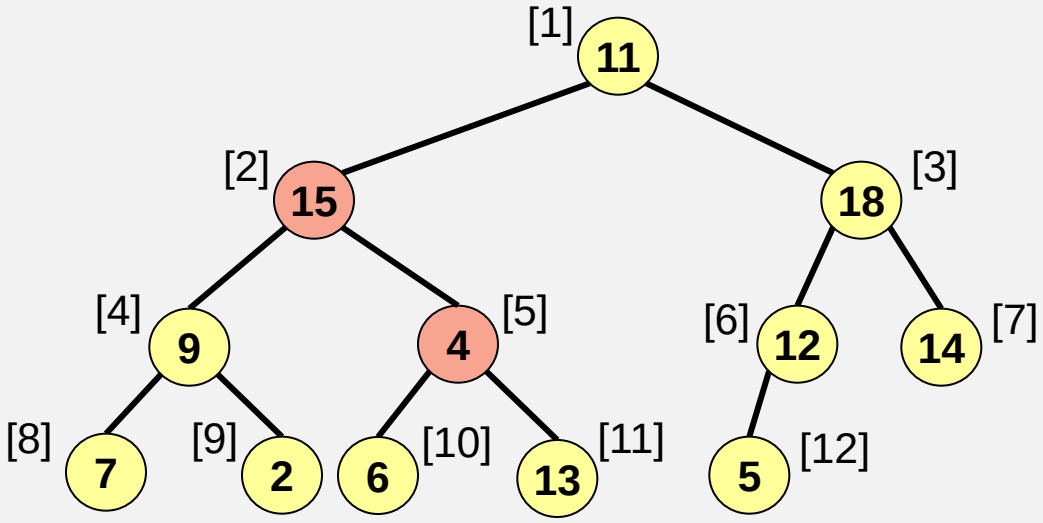


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

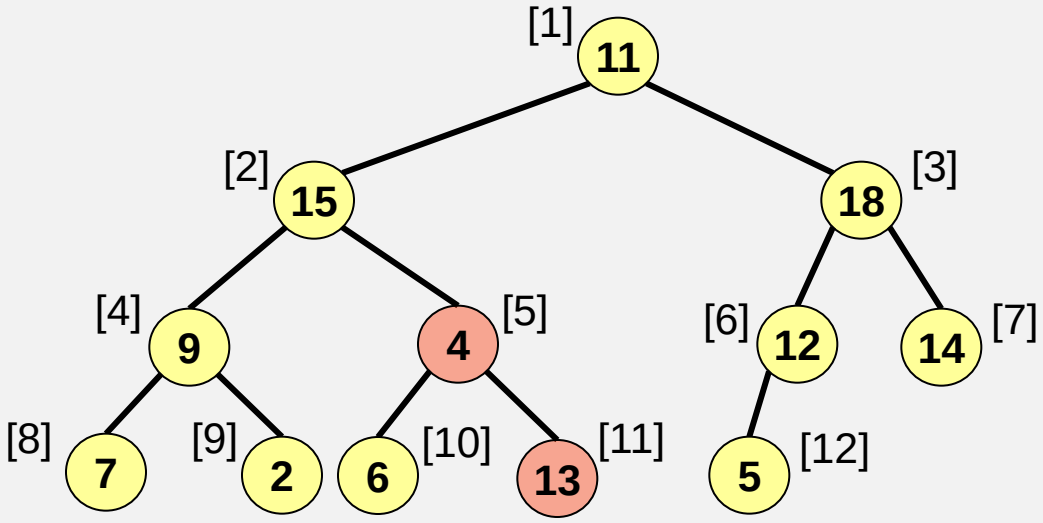


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

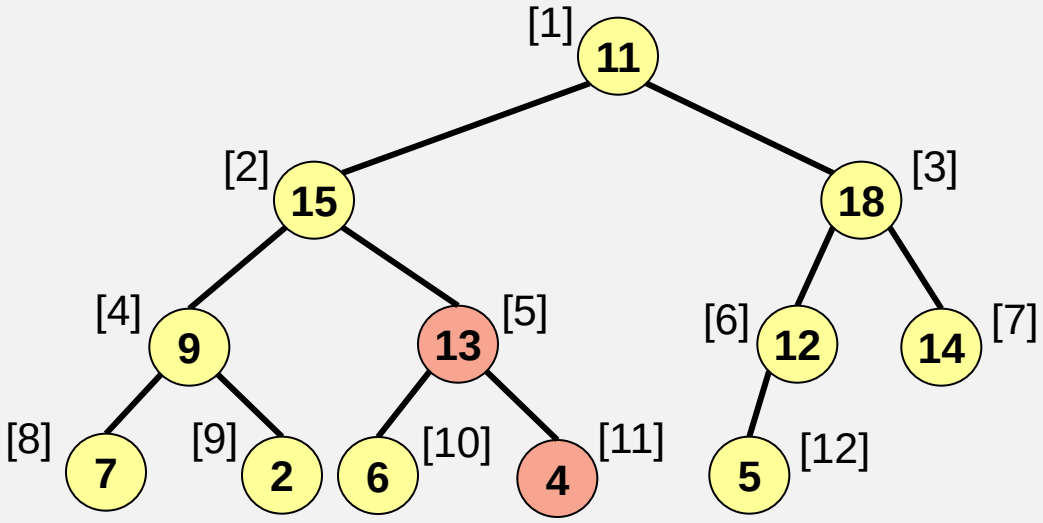


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

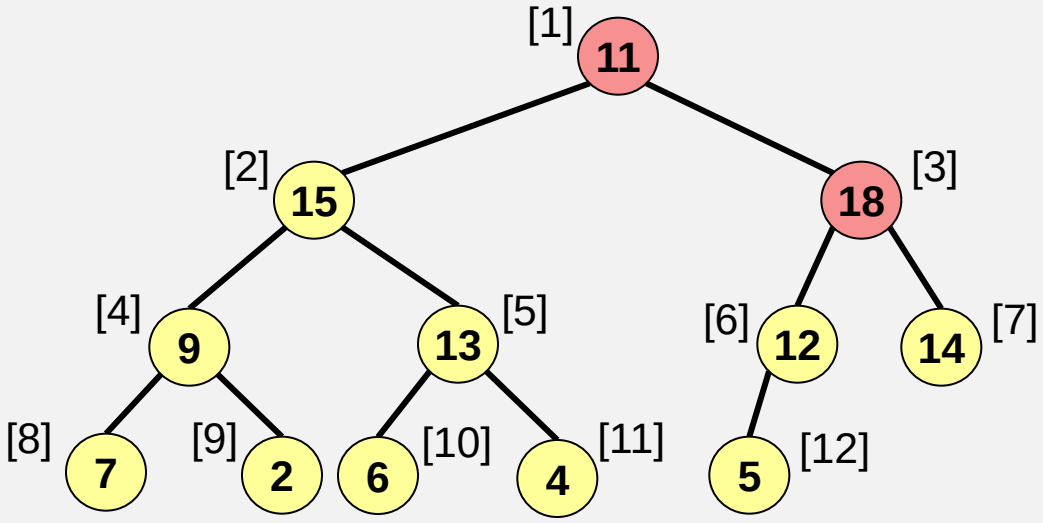


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

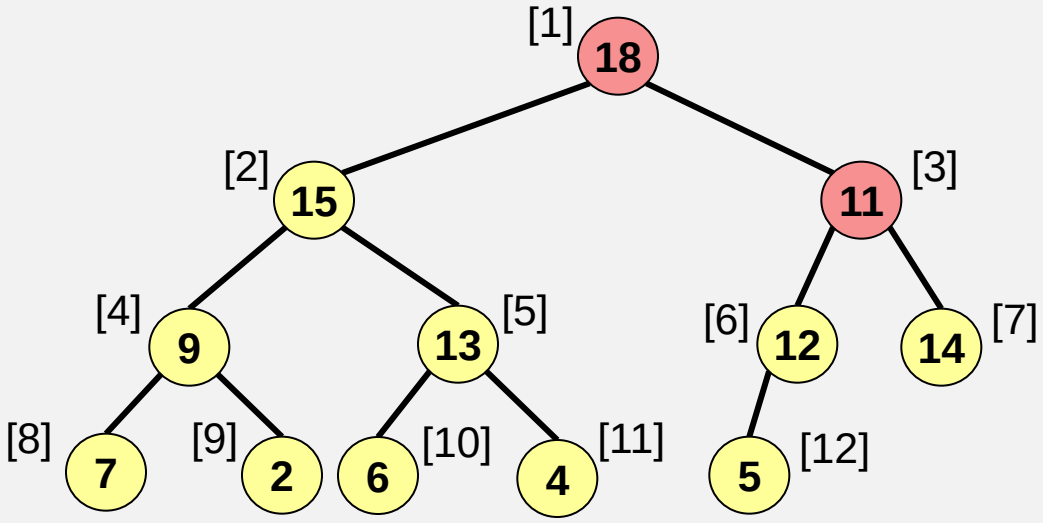


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

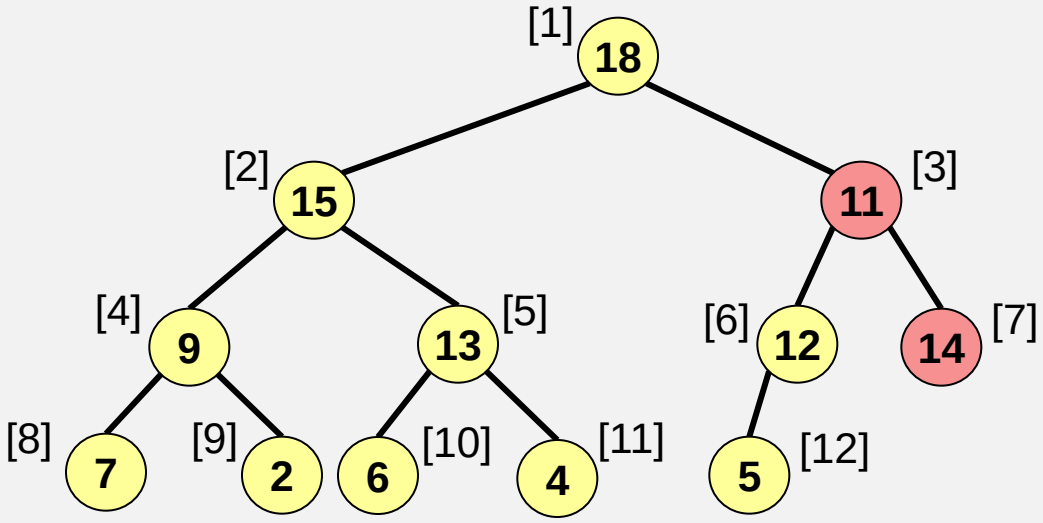


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

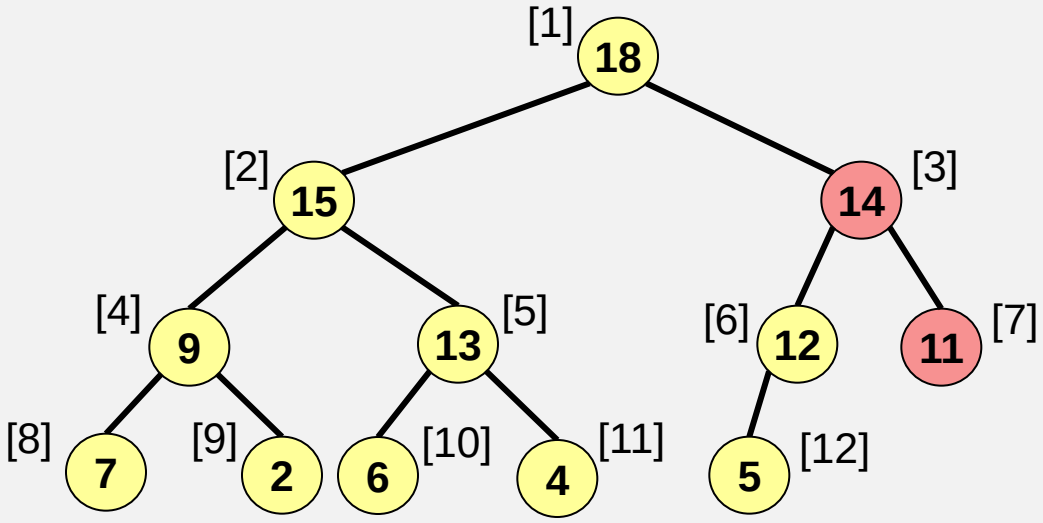


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος

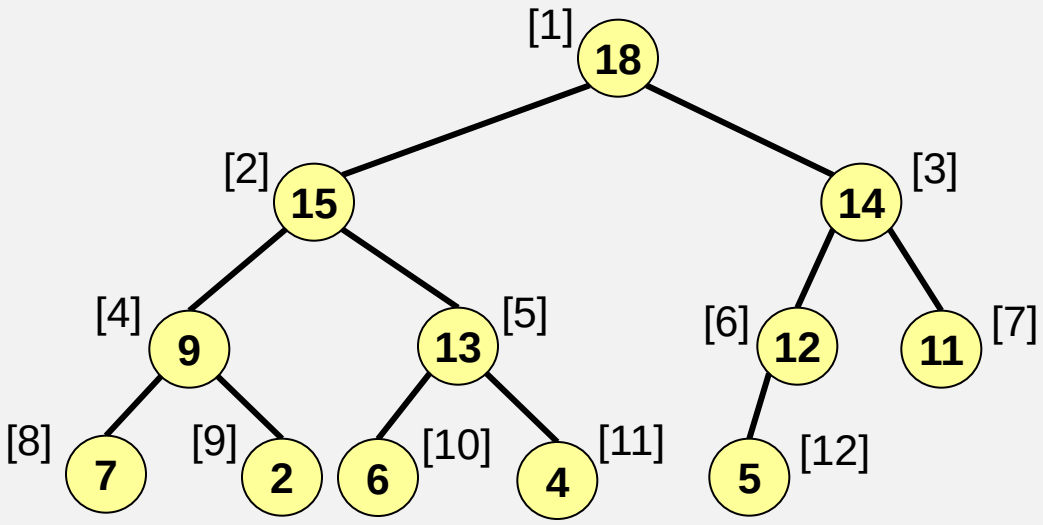


Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) ⇒ O(N) χρόνος



Αλγόριθμοι σε Σωρούς

Ταξινόμηση με σωρό

```
public static void heapsort(Comparable[] a)
{ int N = a.length;
  for (int k=N/2; k>=1; k--)
    fixDown(a,k,N);
  while (N>1)
  {
    exch(a,1,N--);
    fixDown(a,1,N);
  }
}
```

Τακτοποίηση σωρού
(αντικαθιστά τη διαδοχική εισαγωγή των στοιχείων στην ουρά) $\Rightarrow$ O(N) χρόνος

Απόδειξη για $N = 2^n - 1$

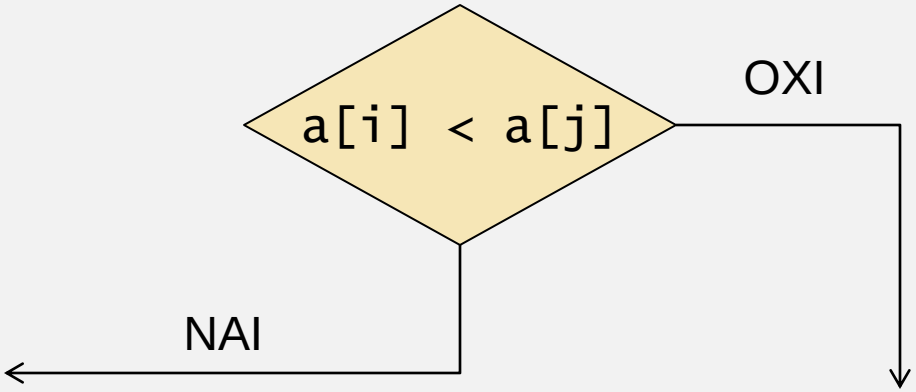
Ο αριθμός των αντιμεταθέσεων είναι το πολύ

$$\sum_{k=1}^{n} k2^{n-k-1} = 2^n - n - 1 < N$$

Αλγόριθμοι Ταξινόμησης

Κάτω φράγμα για ταξινόμηση που χρησιμοποιεί μόνο συγκρίσεις

Οι αλγόριθμοι ταξινόμησης που μελετάμε στο μάθημα βασίζονται σε συγκρίσεις ανά δύο των στοιχείων της ακολουθίας:



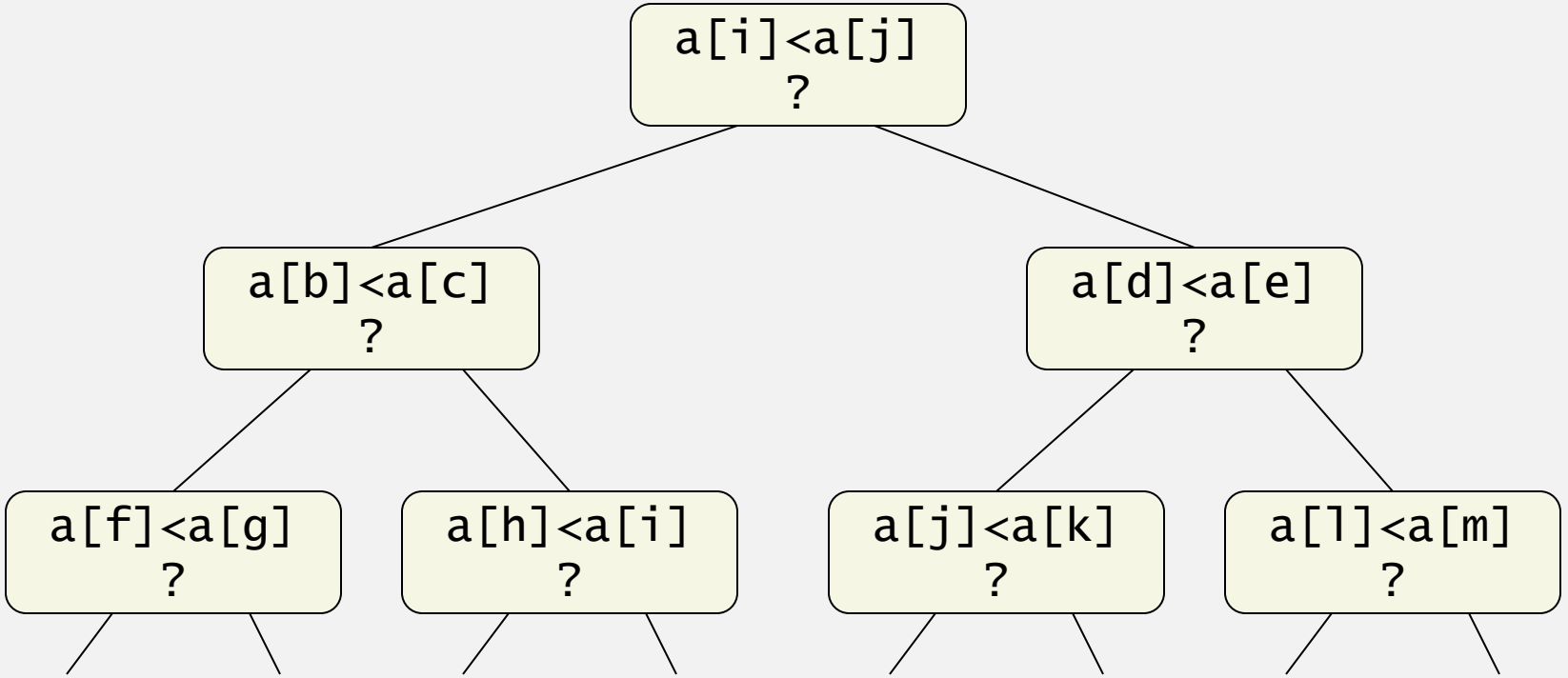
Οποιοσδήποτε τέτοιος αλγόριθμος μπορεί να αναπαρασταθεί με ένα **δένδρο απόφασης**

Θα δείξουμε ότι ο χρόνος εκτέλεσης στη χειρότερη περίπτωση είναι $\Omega(n \log n)$

Αλγόριθμοι Ταξινόμησης

Κάτω φράγμα για ταξινόμηση που χρησιμοποιεί μόνο συγκρίσεις

Δένδρο απόφασης

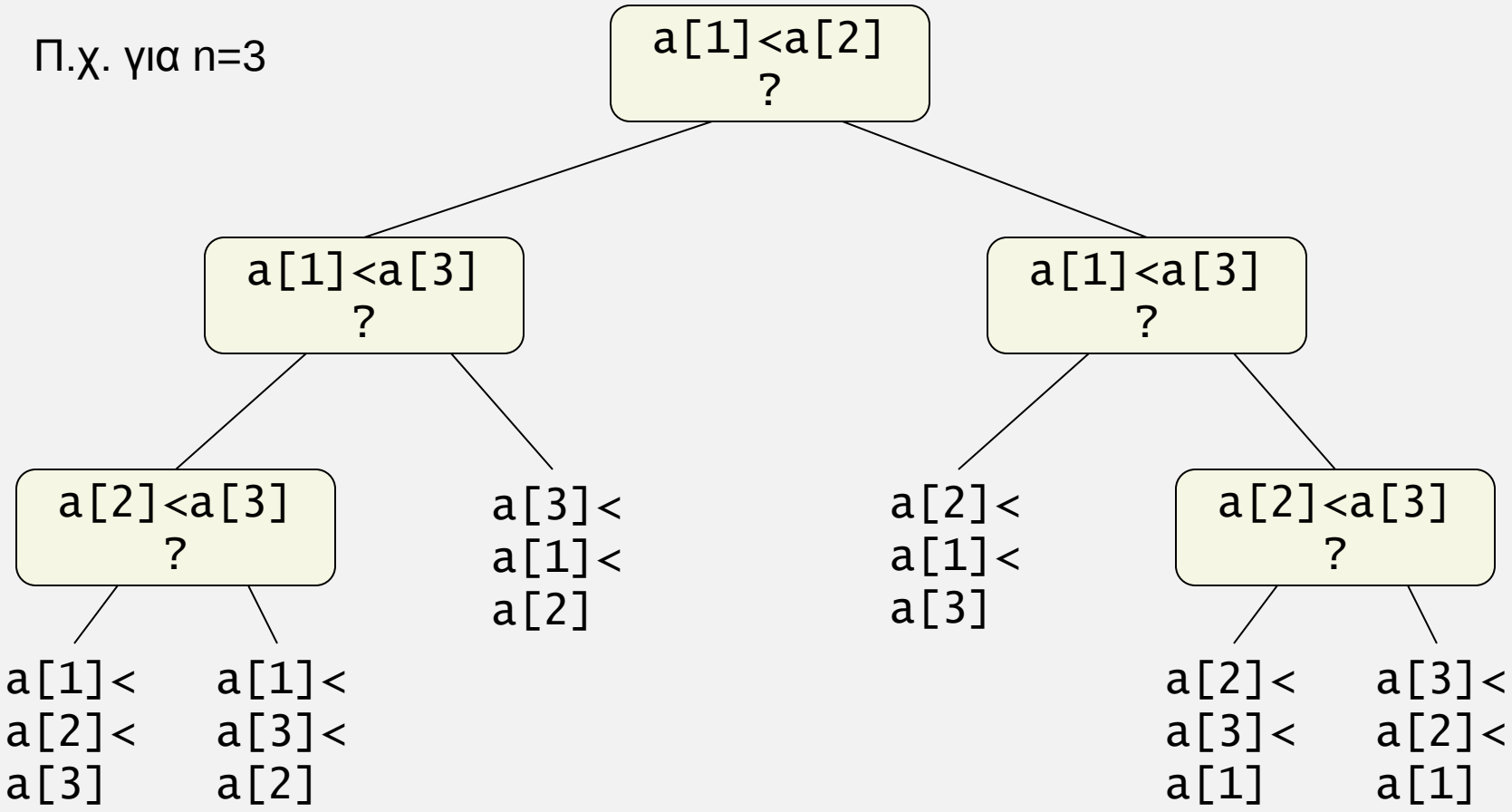


Αλγόριθμοι Ταξινόμησης

Κάτω φράγμα για ταξινόμηση που χρησιμοποιεί μόνο συγκρίσεις

Δένδρο απόφασης

Π.χ. για n=3



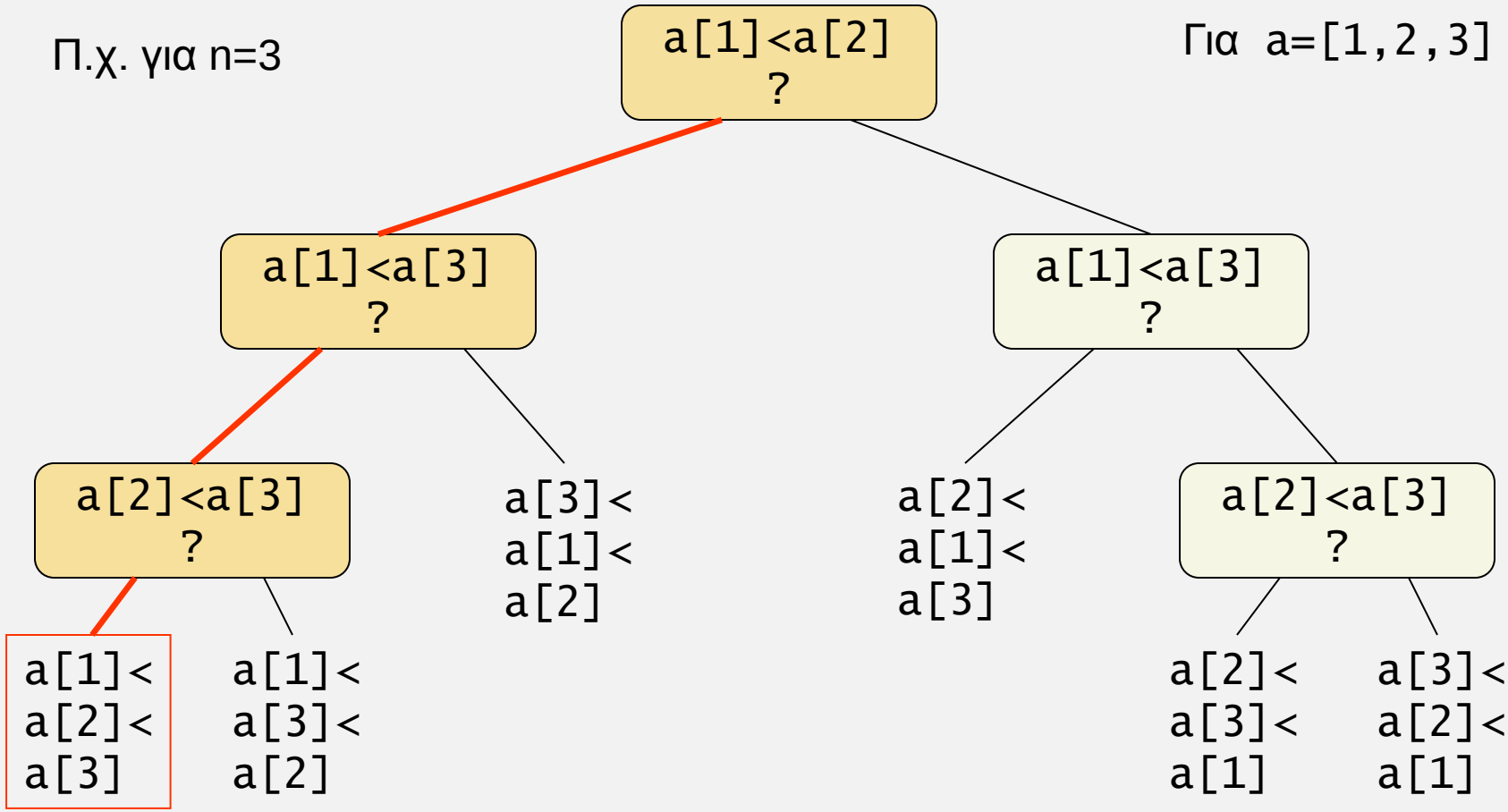
Η εκτέλεση του αλγόριθμου ακολουθεί ένα μονοπάτι από τη ρίζα προς κάποιο φύλλο

Αλγόριθμοι Ταξινόμησης

Κάτω φράγμα για ταξινόμηση που χρησιμοποιεί μόνο συγκρίσεις

Δένδρο απόφασης

Π.χ. για n=3



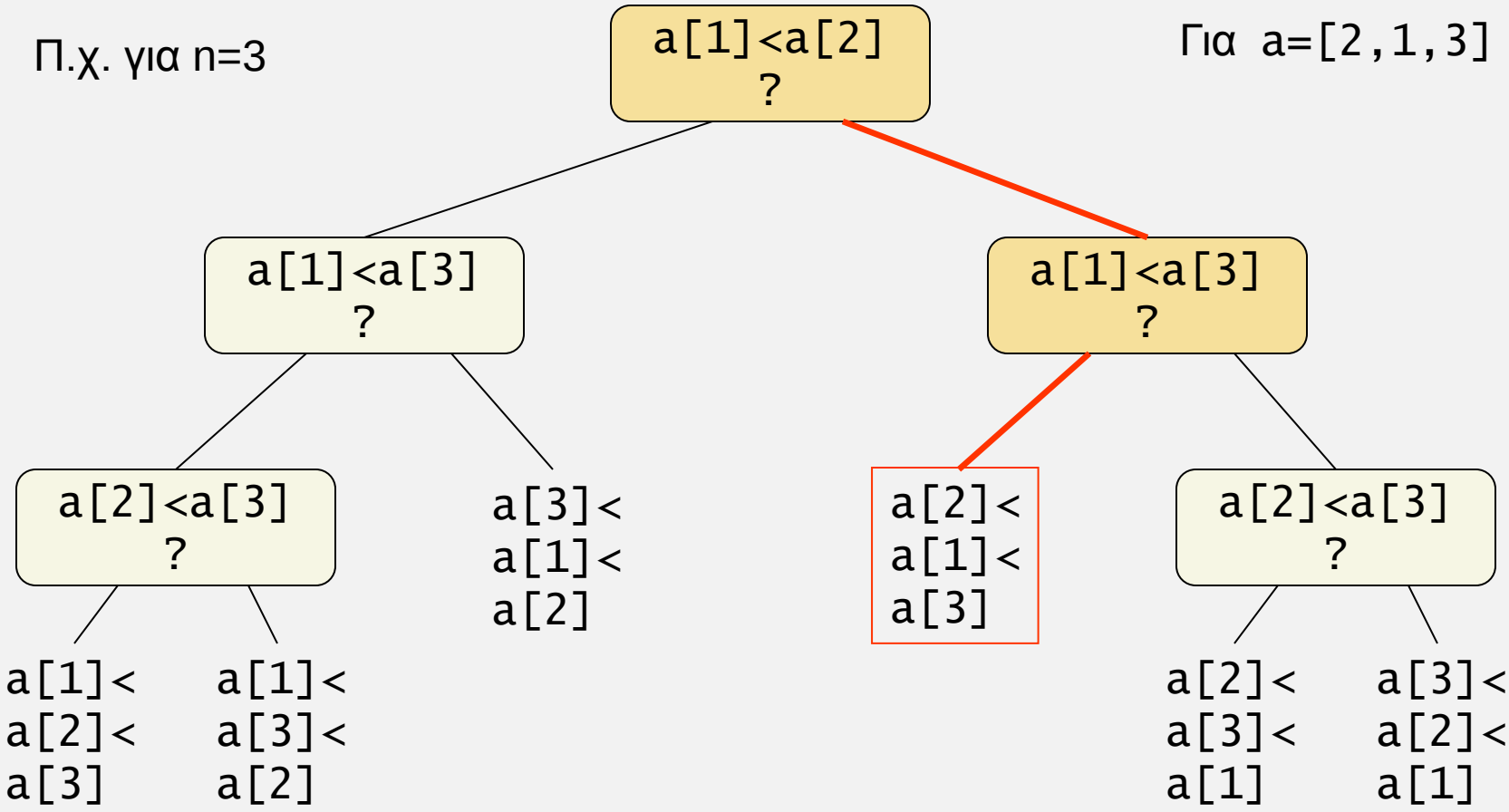
Η εκτέλεση του αλγόριθμου ακολουθεί ένα μονοπάτι από τη ρίζα προς κάποιο φύλλο

Αλγόριθμοι Ταξινόμησης

Κάτω φράγμα για ταξινόμηση που χρησιμοποιεί μόνο συγκρίσεις

Δένδρο απόφασης

Π.χ. για n=3

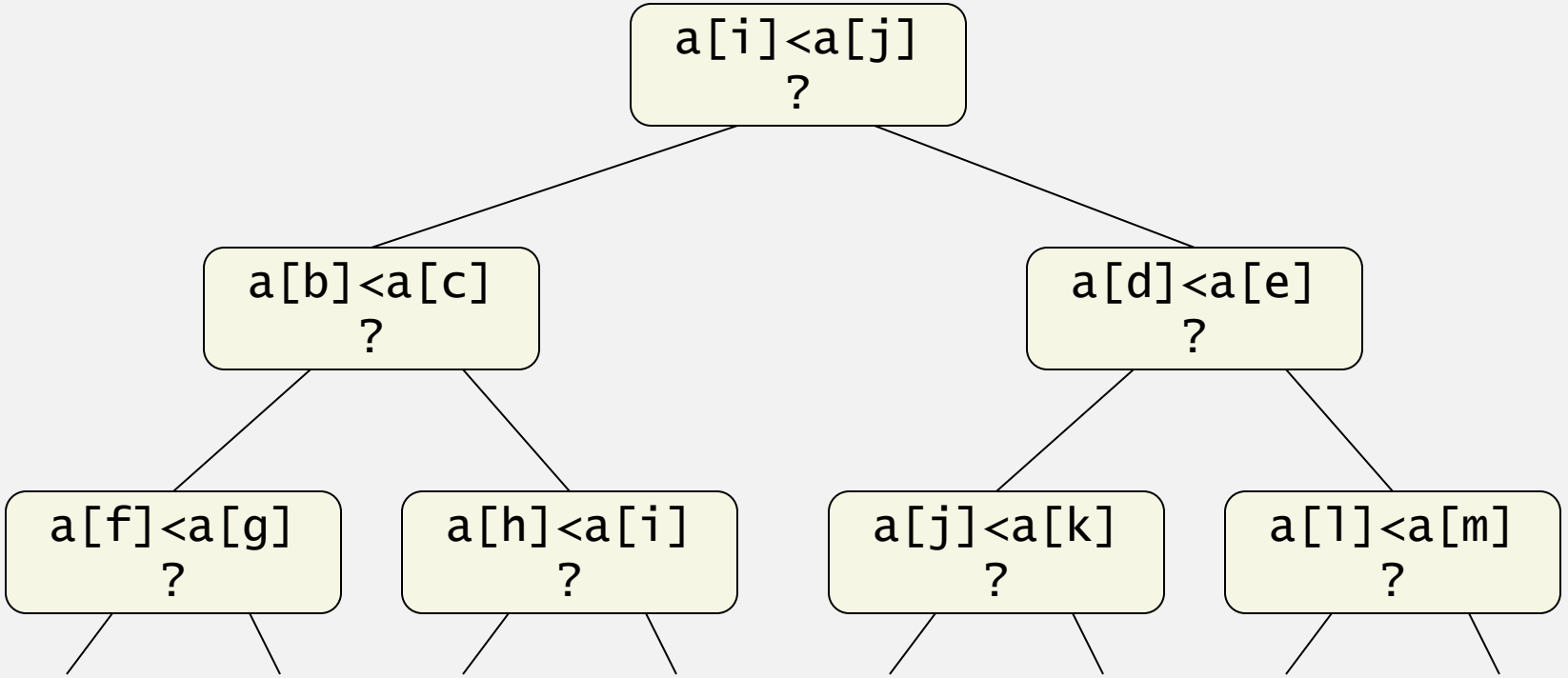


Η εκτέλεση του αλγόριθμου ακολουθεί ένα μονοπάτι από τη ρίζα προς κάποιο φύλλο

Αλγόριθμοι Ταξινόμησης

Κάτω φράγμα για ταξινόμηση που χρησιμοποιεί μόνο συγκρίσεις

Δένδρο απόφασης

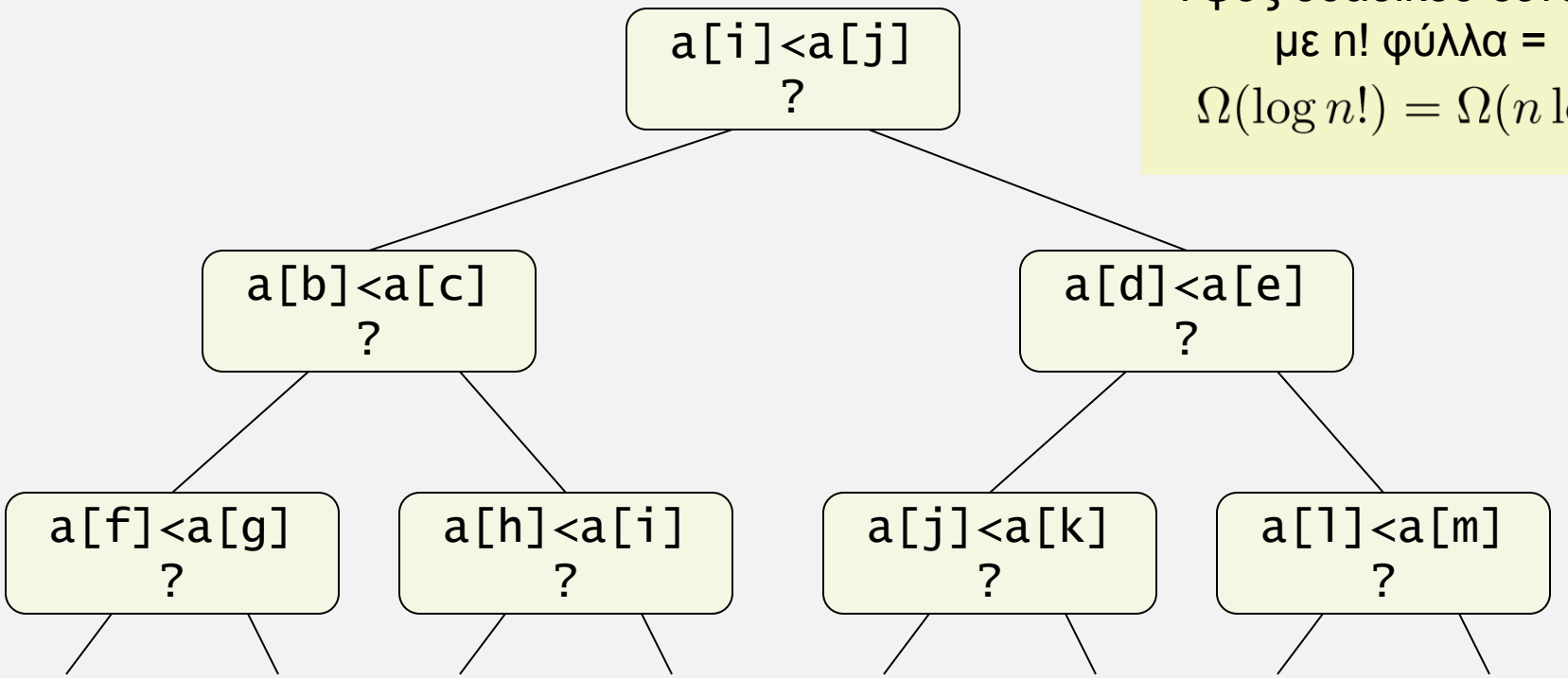


- Κάθε μετάθεση της ακολουθίας εισόδου αντιστοιχεί σε διαφορετικό φύλλο $\Rightarrow$ υπάρχουν n! φύλλα
- Ο αριθμός των συγκρίσεων που πραγματοποιεί μια εκτέλεση του αλγόριθμου στη χειρότερη περίπτωση είναι ίσος με το ύψος του δένδρου

Αλγόριθμοι Ταξινόμησης

Κάτω φράγμα για ταξινόμηση που χρησιμοποιεί μόνο συγκρίσεις

Δένδρο απόφασης



Ύψος δυαδικού δένδρου με n! φύλλα =

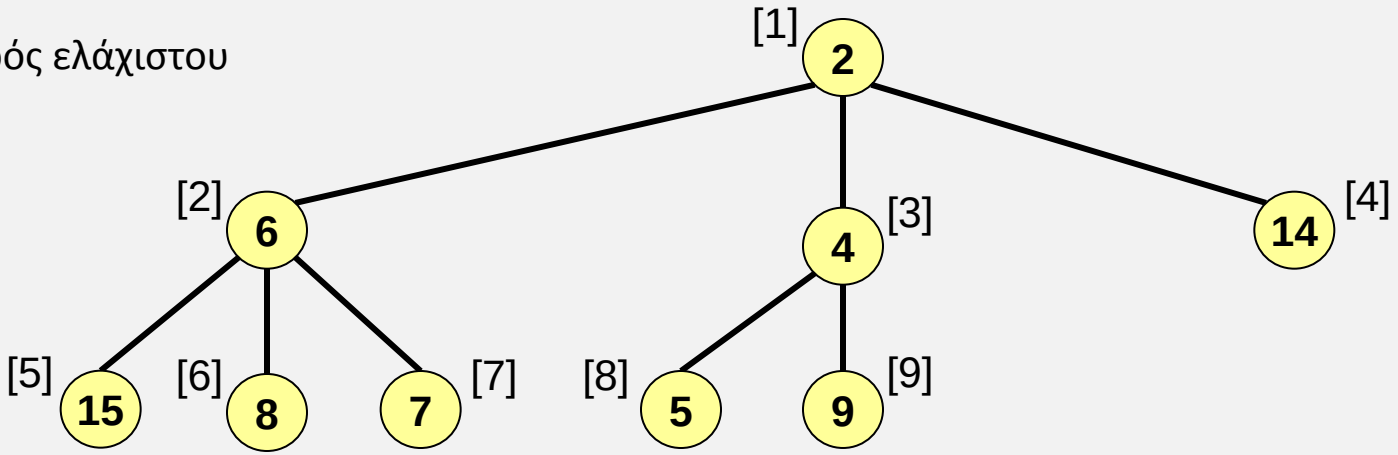
$$\Omega(\log n!) = \Omega(n \log n)$$

- Κάθε μετάθεση της ακολουθίας εισόδου αντιστοιχεί σε διαφορετικό φύλλο ⇒ υπάρχουν n! φύλλα
- Ο αριθμός των συγκρίσεων που πραγματοποιεί μια εκτέλεση του αλγόριθμου στη χειρότερη περίπτωση είναι ίσος με το ύψος του δένδρου

δ-Σωρός

Αποτελεί άμεση γενίκευση του δυαδικού σωρού : Κάθε κόμβος έχει το πολύ δ παιδιά και οι νέοι κόμβοι προστίθενται στο τελευταίο επίπεδο από τα αριστερά προς τα δεξιά

3-σωρός ελάχιστου



Εισαγωγή : $O(\log_\delta n)$ χρόνος

Διαγραφή : $O(\delta \log_\delta n)$ χρόνος

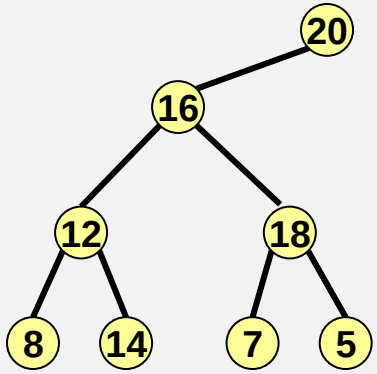
Διωνυμικές ουρές (binomial queues)

Δυαδικά δένδρα αριστερά διατεταγμένα σε σωρό

Το κλειδί κάθε κόμβου είναι μεγαλύτερο ή ίσο από όλα τα κλειδιά του αριστερού υποδένδρου αυτού του κόμβου.

Σωρός δύναμης του 2

Δένδρο αριστερά διατεταγμένο σε σωρό, στο οποίο το δεξί υποδένδρο της ρίζας είναι κενό και το αριστερό υποδένδρο είναι πλήρες.



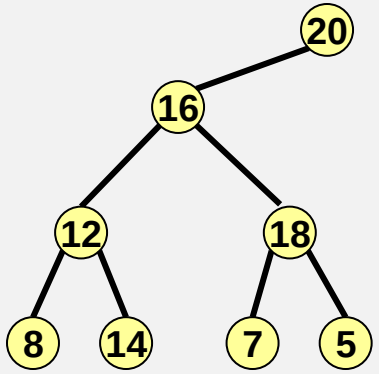
Διωνυμικές ουρές (binomial queues)

Δυαδικά δένδρα αριστερά διατεταγμένα σε σωρό

Το κλειδί κάθε κόμβου είναι μεγαλύτερο ή ίσο από όλα τα κλειδιά του αριστερού υποδένδρου αυτού του κόμβου.

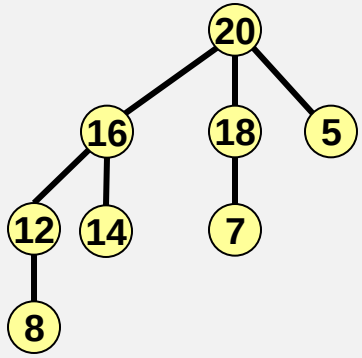
Σωρός δύναμης του 2

Δένδρο αριστερά διατεταγμένο σε σωρό, στο οποίο το δεξί υποδένδρο της ρίζας είναι κενό και το αριστερό υποδένδρο είναι πλήρες.



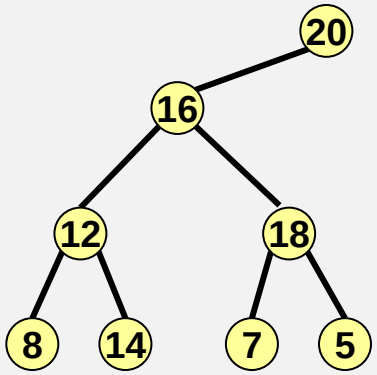
Διωνυμικό δένδρο

Δένδρο που με την αντιστοίχηση αριστερού παιδιού και δεξιού αδελφού δίνει σωρό δύναμης του 2.

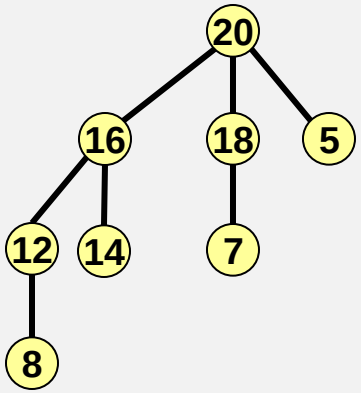


Διωνυμικές ουρές (binomial queues)

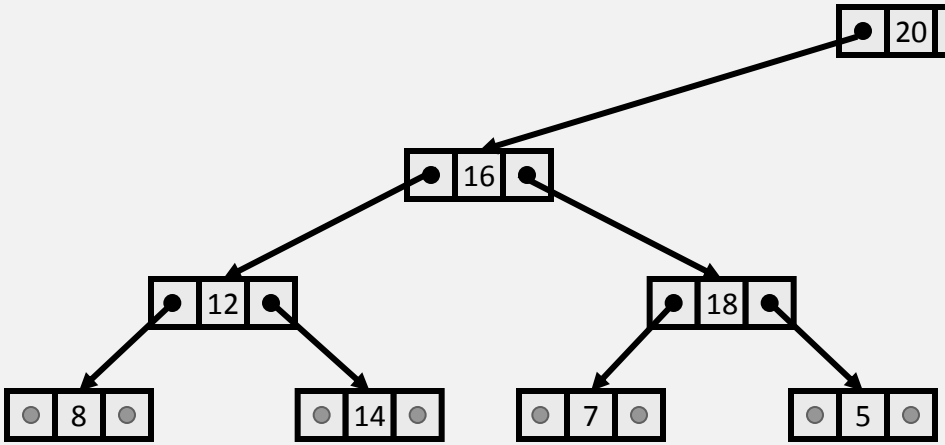
Υλοποίηση



Δυαδικό δένδρο

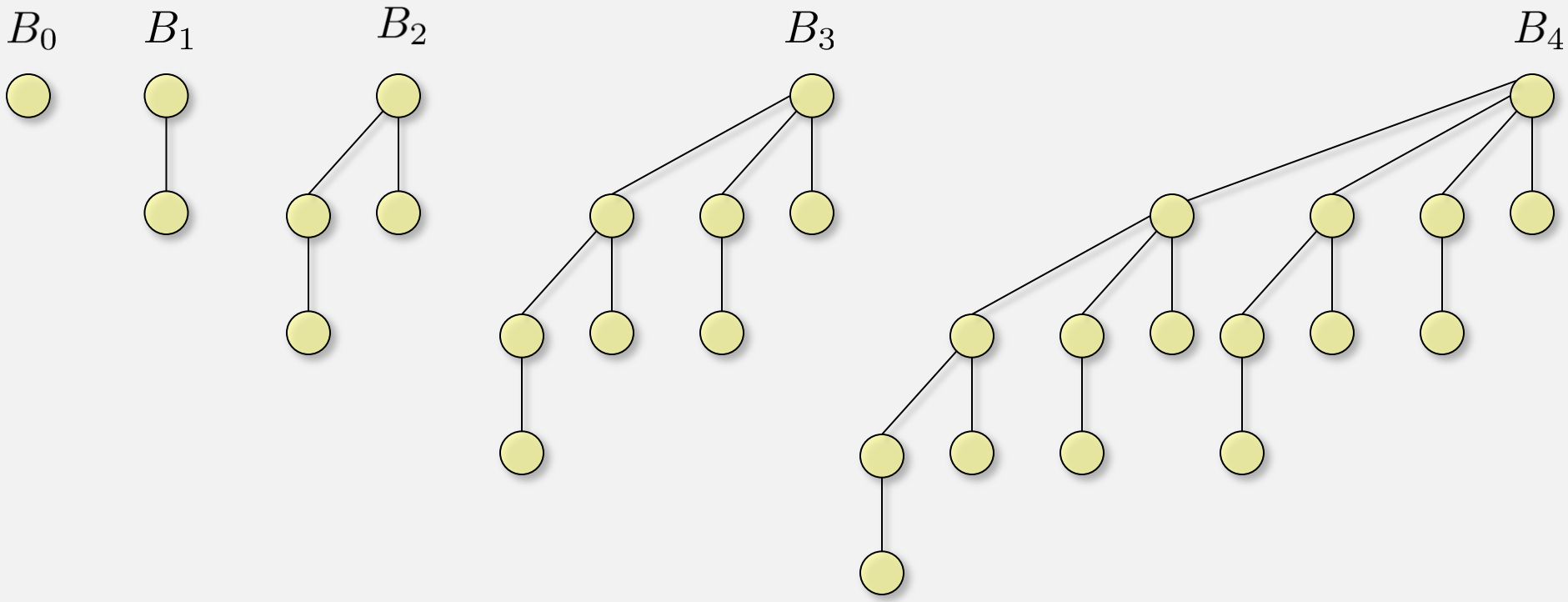


Νοητή αναπαράσταση:
Διωνυμικό δένδρο διατεταγμένο σε σωρό



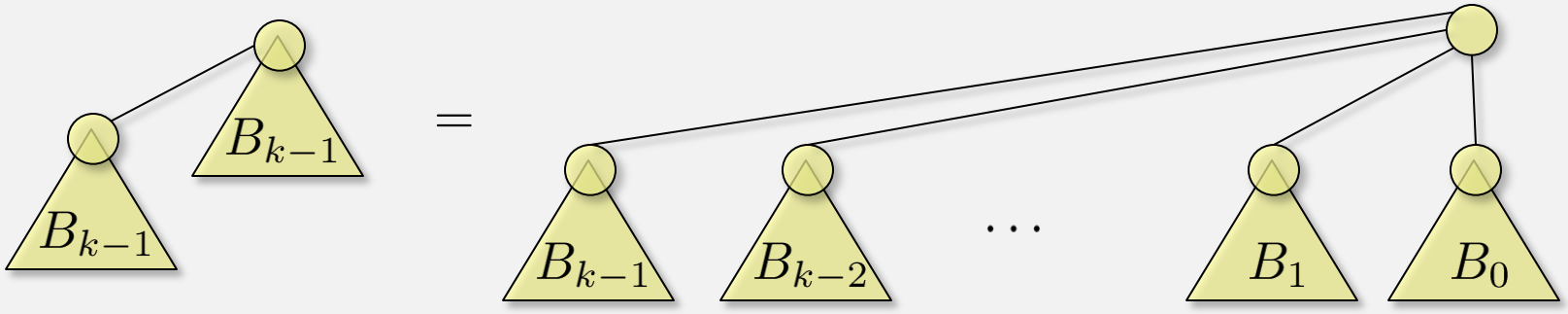
Διωνυμικές ουρές (binomial queues)

Διωνυμικά δένδρα



Διωνυμικές ουρές (binomial queues)

Διωνυμικά δένδρα

$B_k =$ 

Το διωνυμικό δένδρο B_k έχει 2^k κόμβους : $\binom{k}{j}$ κόμβους στο επίπεδο j

$$\sum_{j=0}^{k} \binom{k}{j} = 2^k$$

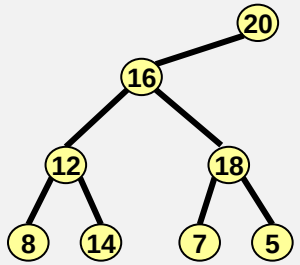
Διωνυμικές ουρές (binomial queues)

Δυαδικά δένδρα αριστερά διατεταγμένα σε σωρό

Το κλειδί κάθε κόμβου είναι μεγαλύτερο ή ίσο από όλα τα κλειδιά του αριστερού υποδένδρου αυτού του κόμβου.

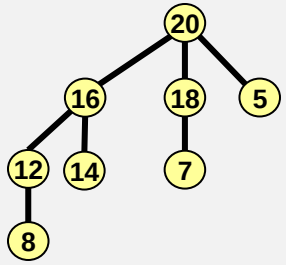
Σωρός δύναμης του 2

Δένδρο αριστερά διατεταγμένο σε σωρό, στο οποίο το δεξί υποδένδρο της ρίζας είναι κενό και το αριστερό υποδένδρο είναι πλήρες.



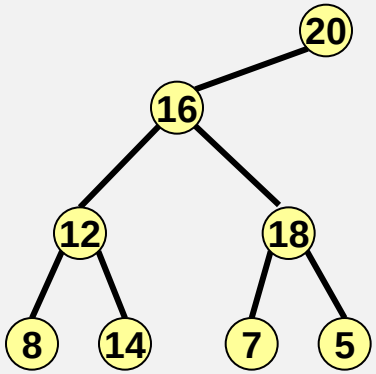
Διωνυμικό δένδρο

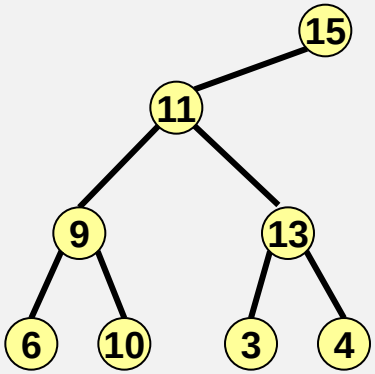
Δένδρο που με την αντιστοίχηση αριστερού παιδιού και δεξιού αδελφού δίνει σωρό δύναμης του 2.

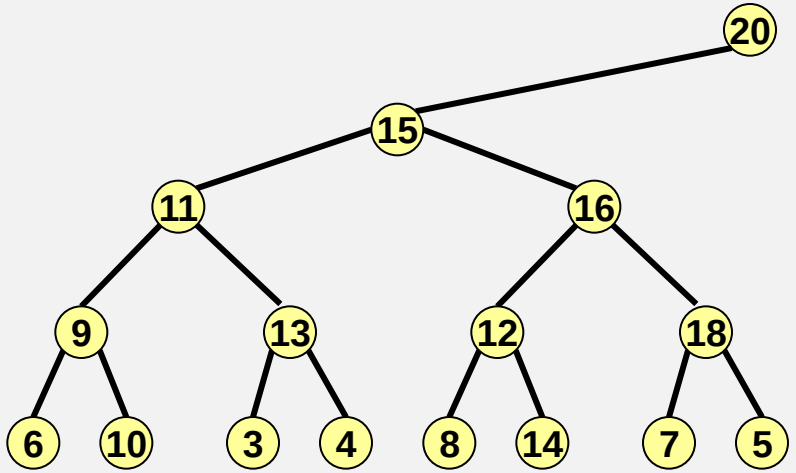


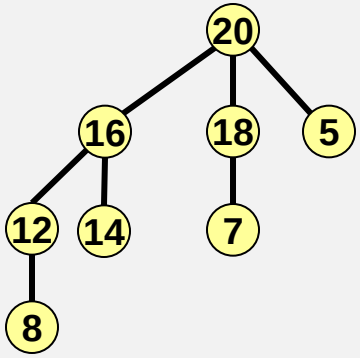
- Το πλήθος των κόμβων σε ένα σωρό δύναμης του 2 είναι δύναμη του 2
- Κανένας κόμβος δεν έχει κλειδί μεγαλύτερο από το κλειδί της ρίζας
- Τα διωνυμικά δέντρα είναι διατεταγμένα σε σωρό

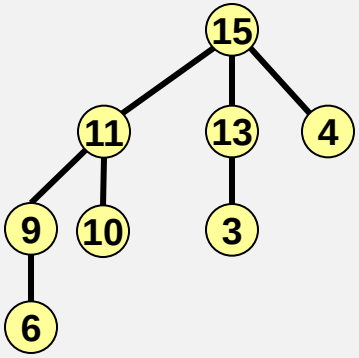
Διωνυμικές ουρές (binomial queues)

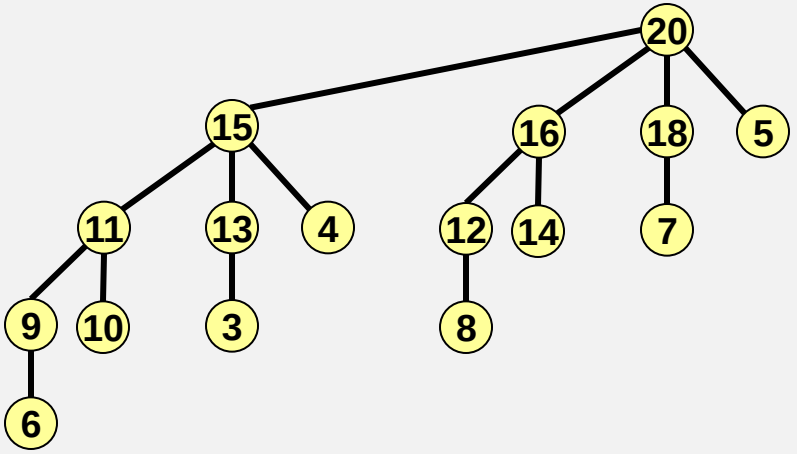










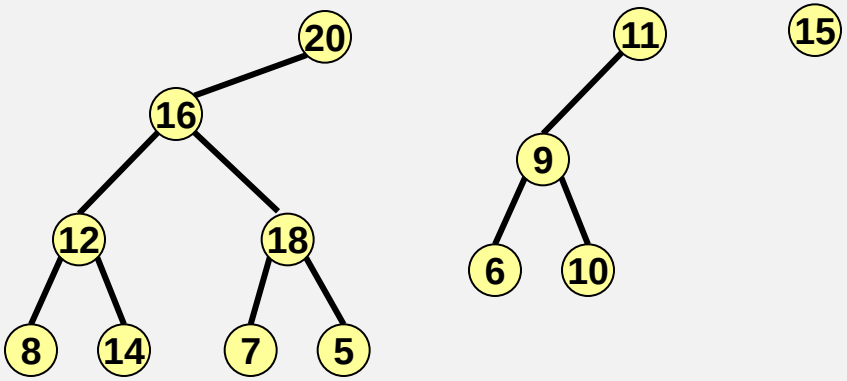


Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Παράδειγμα: διωνυμική ουρά μεγέθους 13 = $(1101)_2$



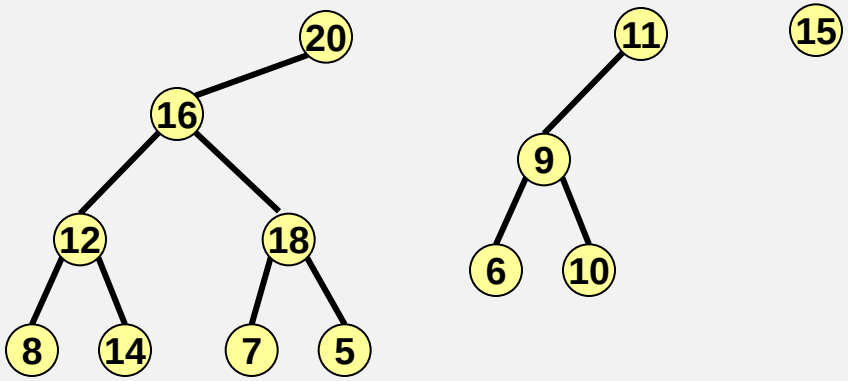
Μια διωνυμική ουρά με N κλειδιά αποτελείται από το πολύ $\lfloor \lg N \rfloor + 1$ σωρούς δύναμης του 2

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου

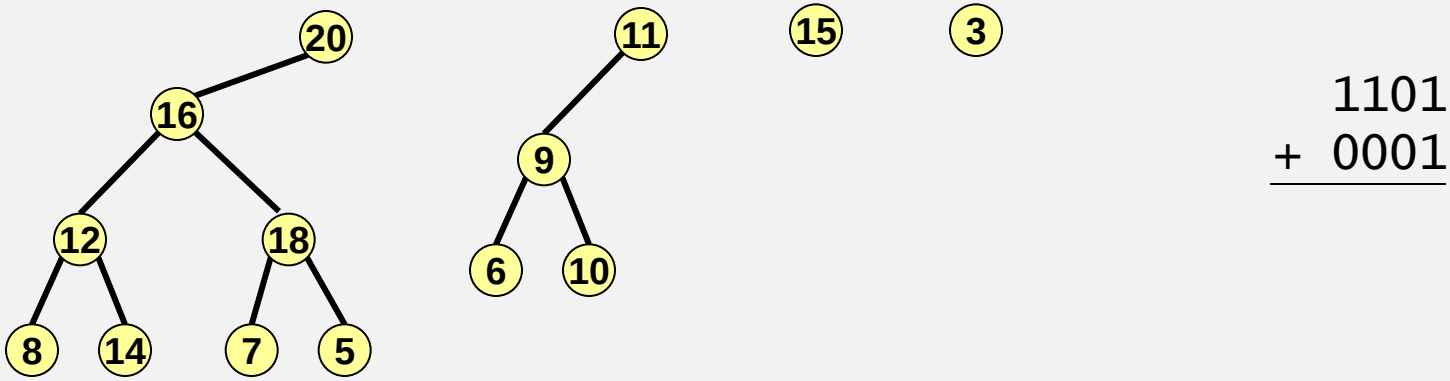


Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου



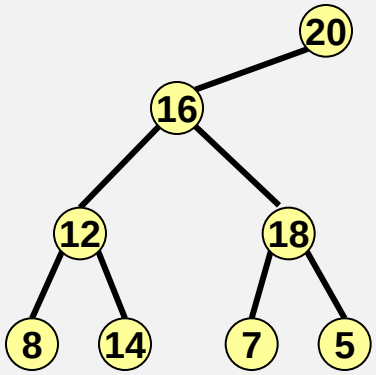
$$\begin{array}{r} 1101 \\ +\ 0001 \\ \hline \end{array}$$

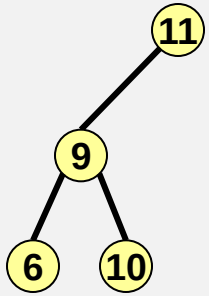
Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου






```
  1101
+ 0001
------
     0
```

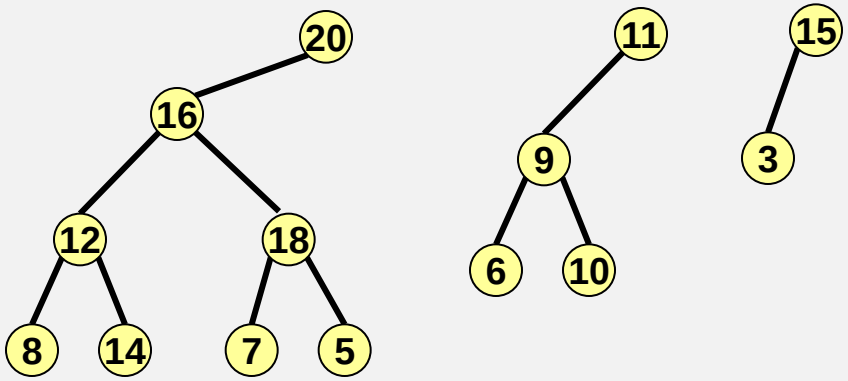
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου




```
  1101
+ 0001
------
  1110
```

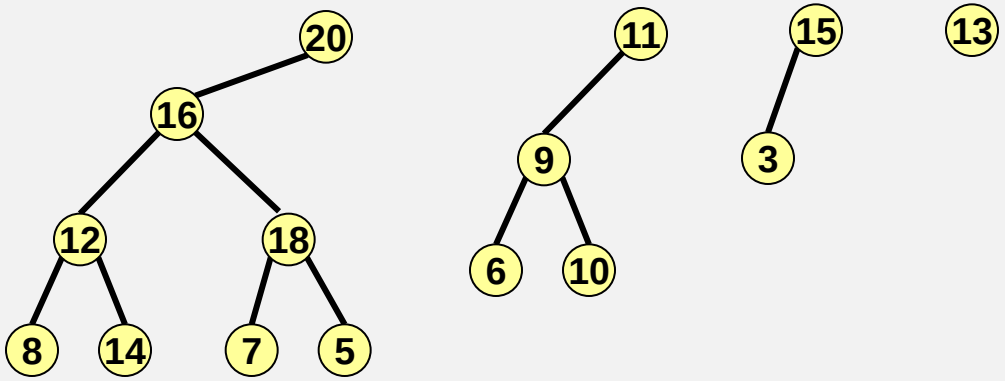
κρατούμενο 0

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου



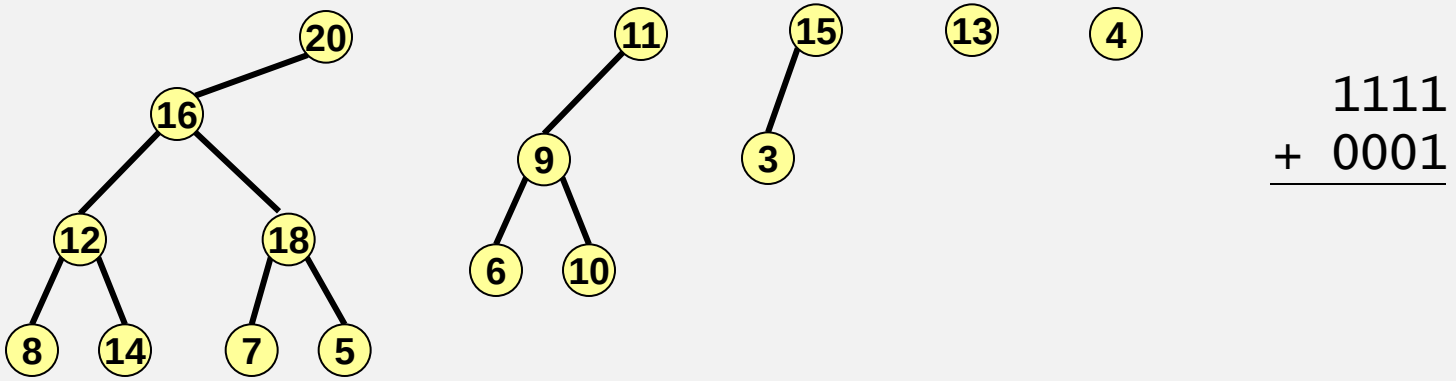
$$\begin{array}{r} 1110 \\ +\ 0001 \\ \hline 1111 \end{array}$$

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου

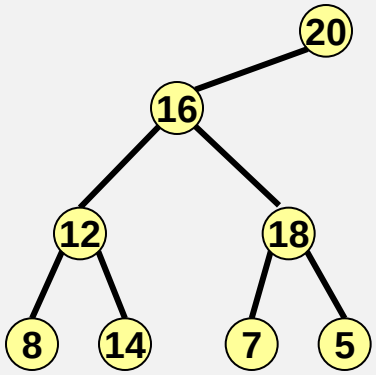


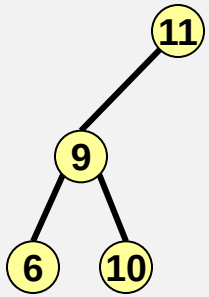
Διωνυμικές ουρές (binomial queues)

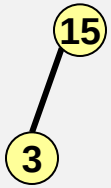
Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου








```
  1111
+ 0001
------
  0
```

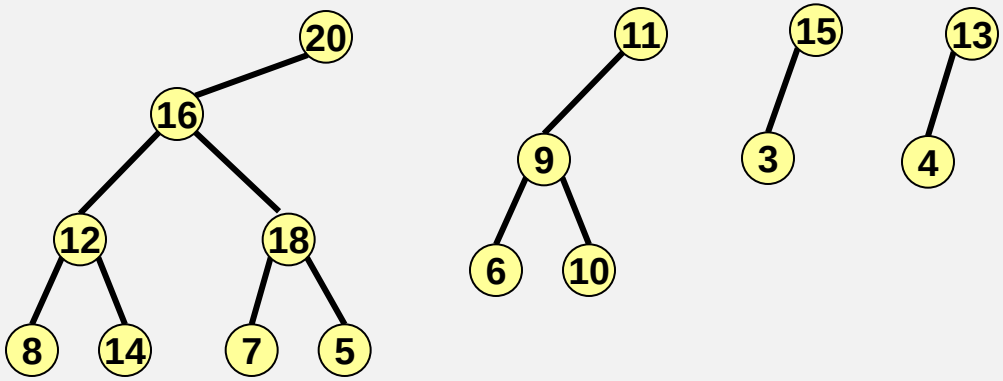
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου




```
  1111
+ 0001
------
     0
```

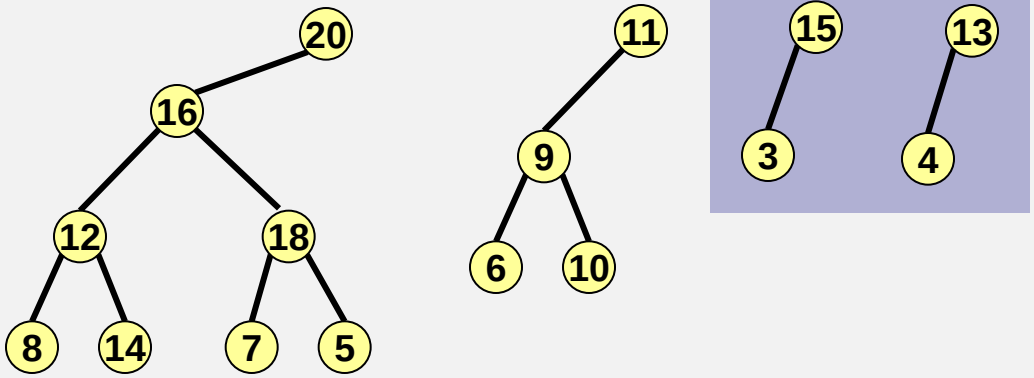
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου




```
  1111
+ 0001
------
  0000
```

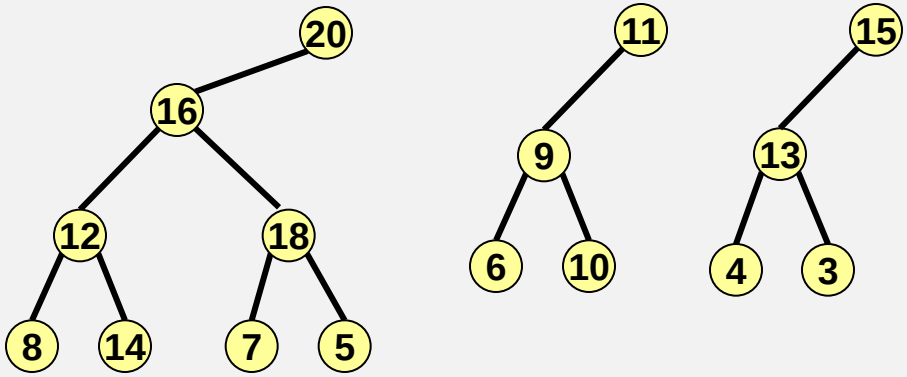
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου




```
  1111
+ 0001
------
  0000
```

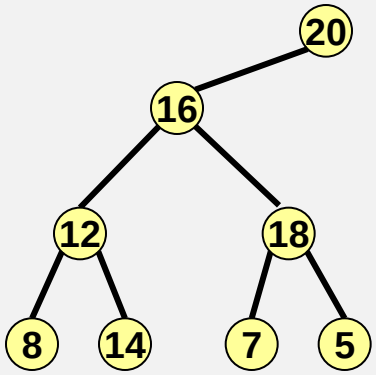
κρατούμενο 1

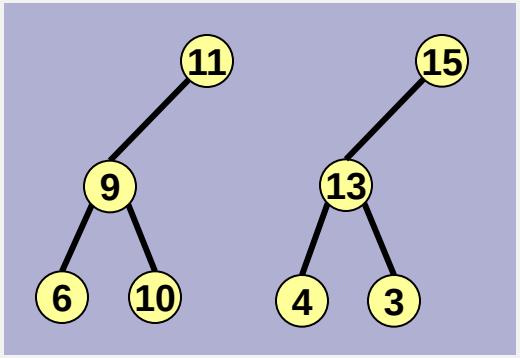
Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου






```
  1111
+ 0001
------
  0000
```

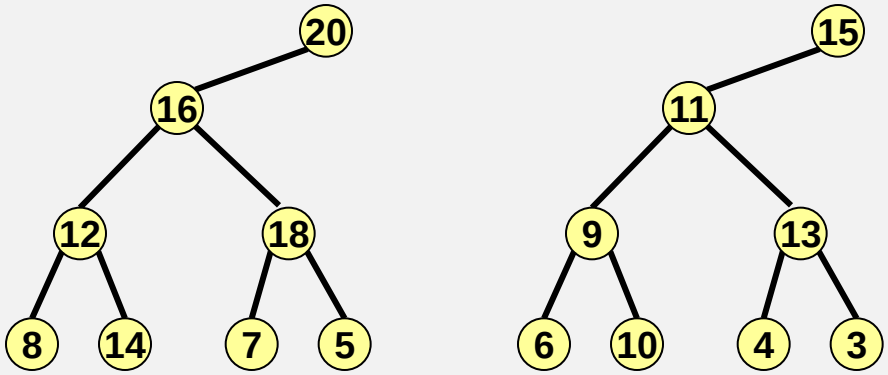
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου




```
  1111
+ 0001
------
  000
```

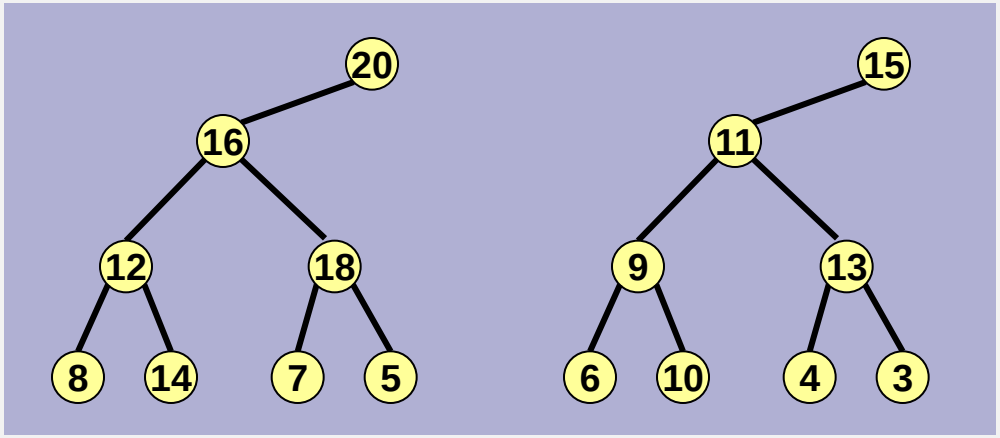
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου




```
   1111
+  0001
------
   0000
```

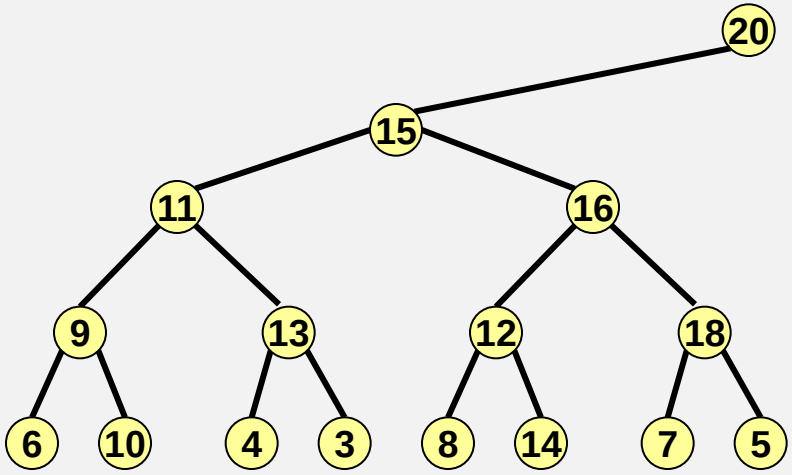
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Εισαγωγή στοιχείου




```
  1111
+ 0001
 10000
```

κρατούμενο 0

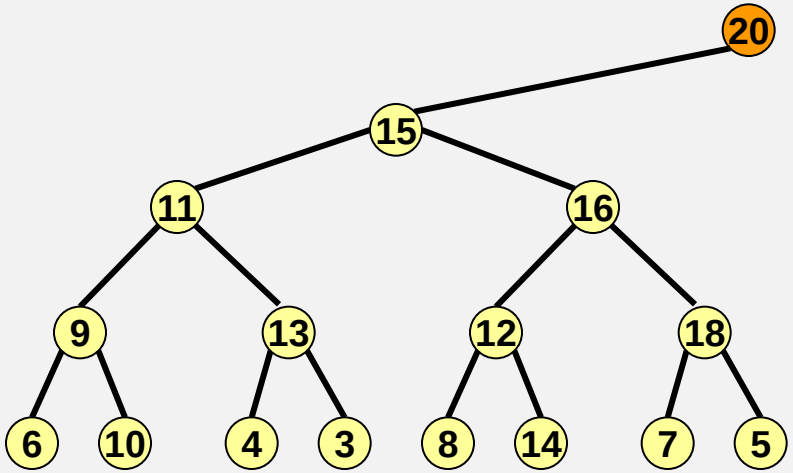
Χρόνος = $O(\log N)$

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Διαγραφή μέγιστου από σωρό δύναμης του 2

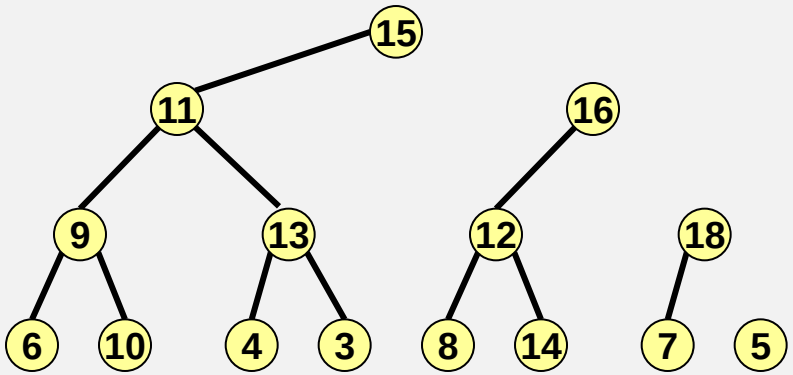


Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Διαγραφή μέγιστου από σωρό δύναμης του 2

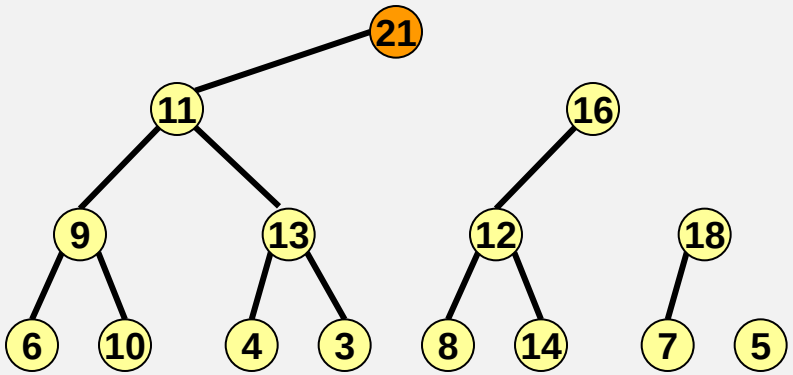


Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Διαγραφή μέγιστου

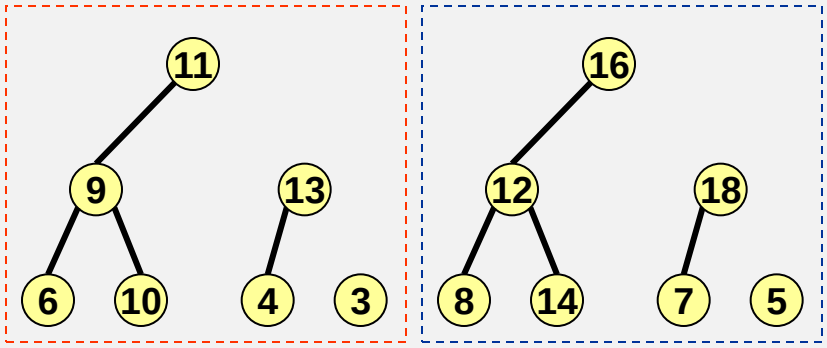


Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Διαγραφή μέγιστου



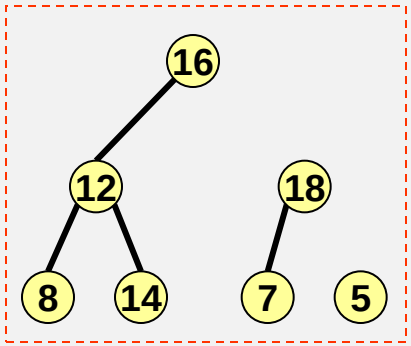
Πρέπει να **ενώσουμε** δύο ουρές

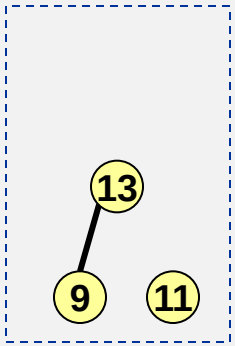
Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

΄Ενωση δύο διωνυμικών ουρών





$$\begin{array}{rr} & 111 \\ + & 011 \\ \hline \end{array}$$

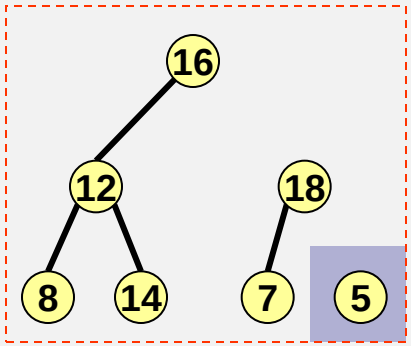
κρατούμενο 0

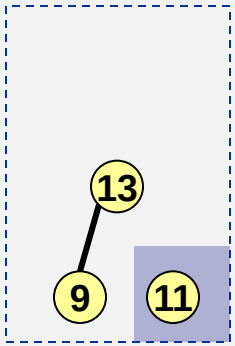
Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

΄Ενωση δύο διωνυμικών ουρών






```
   111
+  011
------
     0
```

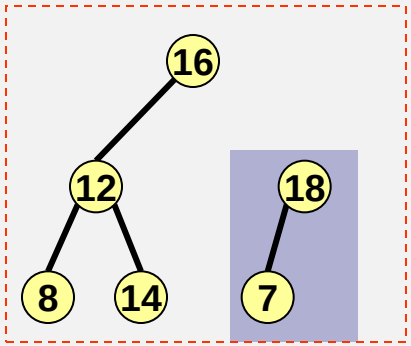
κρατούμενο 1

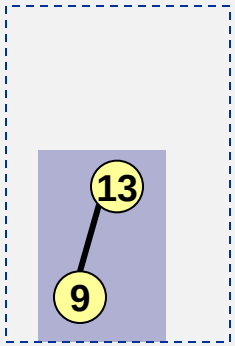
Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

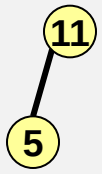
΄Ενωση δύο διωνυμικών ουρών






```
   111
+  011
  ----
   10
```

κρατούμενο 1

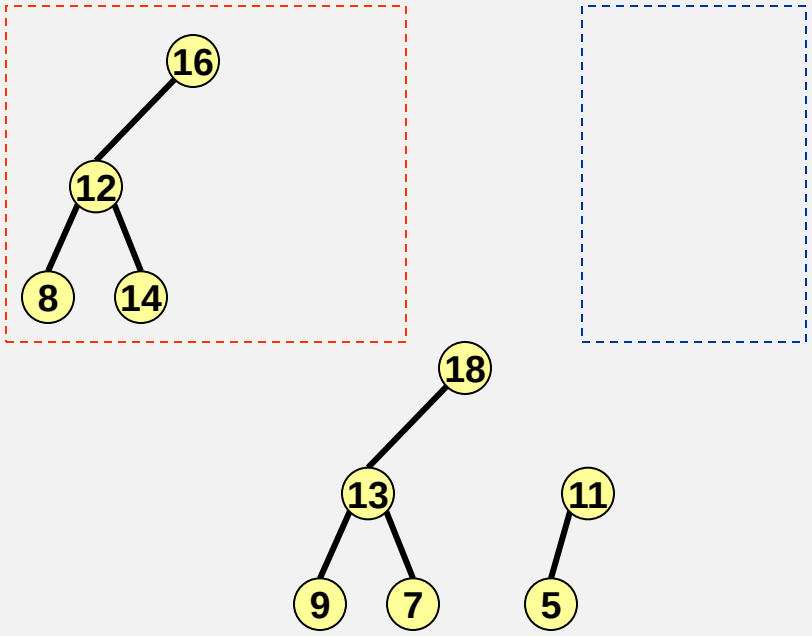


Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Ένωση δύο διωνυμικών ουρών




```
   111
+  011
------
    10
```

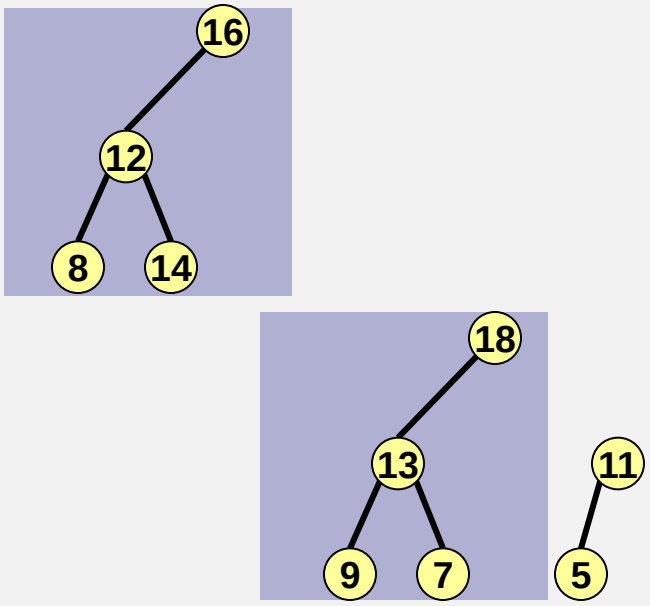
κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

΄Ενωση δύο διωνυμικών ουρών




```
   111
+  011
-------
   010
```

κρατούμενο 1

Διωνυμικές ουρές (binomial queues)

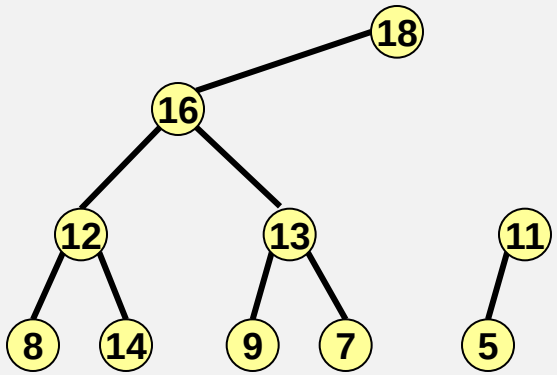
Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

΄Ενωση δύο διωνυμικών ουρών

```
  111
+ 011
 1010
```

κρατούμενο 0



Χρόνος = $O(\log N)$

Διωνυμικές ουρές (binomial queues)

Διωνυμική ουρά

Σύνολο σωρών δύναμης του 2 οι οποίοι δεν έχουν το ίδιο μέγεθος. Η δομή της καθορίζεται από τη δυαδική αναπαράσταση του αριθμού των κόμβων της.

Κατασκευή διωνυμικής ουράς με N κλειδιά

Η κατασκευή μιας διωνυμικής ουράς με N διαδοχικές εισαγωγές σε αρχικά κενή ουρά απαιτεί χρόνο $O(N)$

Για κάθε εισαγωγή έχουμε μια πράξη ένωσης σωρών δύναμης του δύο για κάθε bit που αλλάζει από 1 σε 0 στη δυαδική αναπαράσταση του αριθμού των κόμβων της ουράς

Διωνυμικές ουρές (binomial queues)

Επαύξηση δυαδικού μετρητή

Έστω ένας μετρητής C με k bits : μια πράξη επαύξησης θέτει

$$C \leftarrow C + 1 \pmod{2^k}$$

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Διωνυμικές ουρές (binomial queues)

Επαύξηση δυαδικού μετρητή

Έστω ένας μετρητής C με k bits : μια πράξη επαύξησης θέτει

$$C \leftarrow C + 1 \pmod{2^k}$$

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

1ο ψηφίο από το τέλος:

αλλάζει με κάθε επαύξηση

Διωνυμικές ουρές (binomial queues)

Επαύξηση δυαδικού μετρητή

Έστω ένας μετρητής C με k bits : μια πράξη επαύξησης θέτει

$$C \leftarrow C + 1 \pmod{2^k}$$

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

2ο ψηφίο από το τέλος:

αλλάζει με κάθε δεύτερη επαύξηση

Διωνυμικές ουρές (binomial queues)

Επαύξηση δυαδικού μετρητή

Έστω ένας μετρητής C με k bits : μια πράξη επαύξησης θέτει

$$C \leftarrow C + 1 \pmod{2^k}$$

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

3ο ψηφίο από το τέλος:

αλλάζει με κάθε τέταρτη επαύξηση

Διωνυμικές ουρές (binomial queues)

Επαύξηση δυαδικού μετρητή

Έστω ένας μετρητής C με k bits : μια πράξη επαύξησης θέτει

$$C \leftarrow C + 1 \pmod{2^k}$$

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

4ο ψηφίο από το τέλος:

αλλάζει με κάθε όγδοη επαύξηση

Διωνυμικές ουρές (binomial queues)

Επαύξηση δυαδικού μετρητή

Έστω ένας μετρητής C με k bits : μια πράξη επαύξησης θέτει

$$C \leftarrow C + 1 \pmod{2^k}$$

Γενικά το i-οστό ψηφίο από το τέλος αλλάζει μετά από 2^{i-1} επαυξήσεις.

$\Rightarrow$ Σε ακολουθία N πράξεων το i-οστό ψηφίο από το τέλος αλλάζει συνολικά $\left\lfloor \frac{N}{2^{i-1}} \right\rfloor$ φορές

$\Rightarrow$ Σύνολο αλλαγών για όλα τα ψηφία = $\sum_{i=1}^{k} \left\lfloor \frac{N}{2^{i-1}} \right\rfloor$

Διωνυμικές ουρές (binomial queues)

Επαύξηση δυαδικού μετρητή

Έστω ένας μετρητής C με k bits : μια πράξη επαύξησης θέτει

$$C \leftarrow C + 1 \pmod{2^k}$$

Γενικά το i-οστό ψηφίο από το τέλος αλλάζει μετά από 2^{i-1} επαυξήσεις.

$\Rightarrow$ Σε ακολουθία N πράξεων το i-οστό ψηφίο από το τέλος αλλάζει συνολικά $\left\lfloor \frac{N}{2^{i-1}} \right\rfloor$ φορές

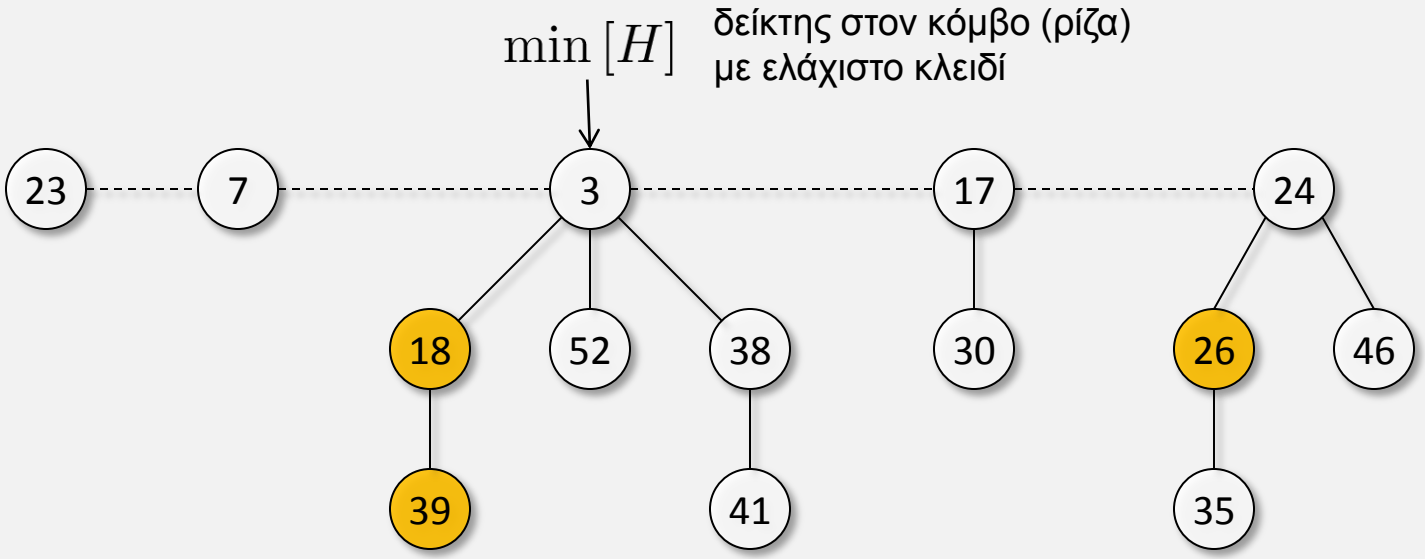
$\Rightarrow$ Σύνολο αλλαγών για όλα τα ψηφία =

$$\sum_{i=1}^{k} \left\lfloor \frac{N}{2^{i-1}} \right\rfloor \leq \sum_{i=1}^{k} \frac{N}{2^{i-1}} = N \sum_{j=0}^{k-1} 2^{-j}$$

$$< N \sum_{j=0}^{\infty} 2^{-j} = 2N$$

Σωρός Fibonacci

Βασίζεται στο διωνυμικό σωρό (δηλαδή αποτελεί μια συλλογή από δένδρα) αλλά έχει πιο χαλαρή δομή



Αντισταθμιστικοί χρόνοι εκτέλεσης

εισαγωγή, ένωση, εύρεση ελάχιστου, μείωση κλειδιού	$O(1)$
διαγραφή, εξαγωγή ελάχιστου	$O(\log n)$