Διάλεξη 17: Ο Αλγόριθμος Ταξινόμησης HeapSort

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Η διαδικασία PercolateDown, Δημιουργία Σωρού
- Ο Αλγόριθμος Ταξινόμησης HeapSort
- Υλοποίηση, Παραδείγματα
- Παραλλαγές Σωρών

Διαδικασία Καθόδου PercolateDown

- Έστω ένας πίνακας A[1..n] και μια τιμή i, θα ορίσουμε διαδικασία PercoladeDown(i), η οποία μετακινεί το στοιχείο A[i] μέσα στον σωρό προς τα κάτω όσο χρειάζεται.
- Έστω ότι A[i] = k.
- Θεωρούμε πως η i είναι άδεια θέση.
- Αν η άδεια θέση έχει παιδί που περιέχει στοιχείο μικρότερο του k και x είναι το μικρότερο τέτοιο παιδί, τότε μετακινούμε το στοιχείο του x στην κενή θέση και μετακινούμε την κενή θέση στο x.
- Επαναλαμβάνουμε την ίδια διαδικασία μέχρι τη στιγμή που η κενή θέση δεν έχει παιδιά με στοιχεία μικρότερα του k. Τότε αποθηκεύουμε το k στην θέση αυτή.
- Ο χρόνος εκτέλεσης είναι ανάλογος του ύψους του κόμβου που αντιστοιχεί στη θέση i του σωρού. Δηλαδή, στη χείριστη περίπτωση, όπου i=n, O(lg n).

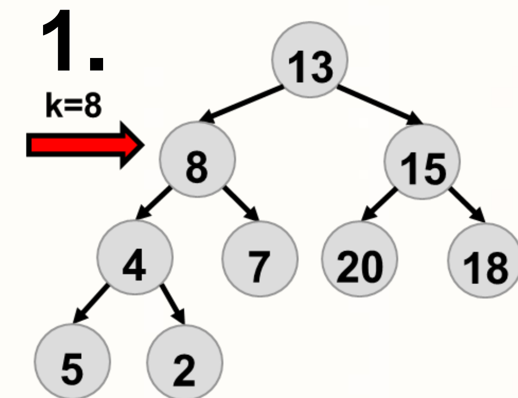
Διαδικασία Καθόδου PercolateDown (συν.)

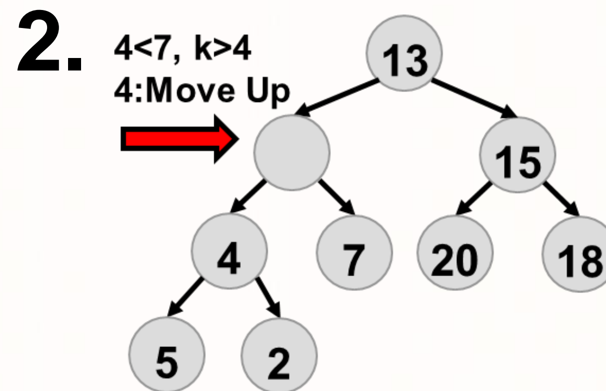
- Μη αναδρομική διαδικασία PercolateDown

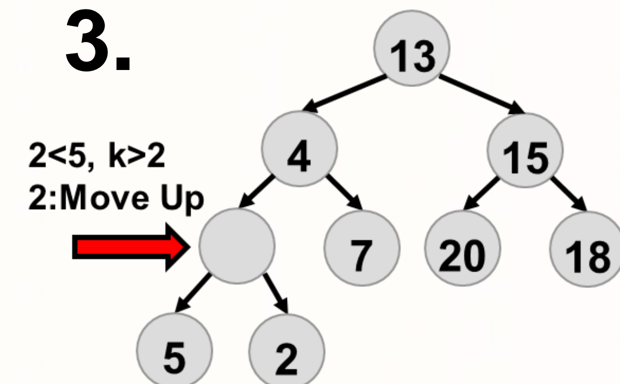
```
public static void PercolateDown(int A[], int n, int i) {
  int k = A[i];
  int j;

  while (2 * i <= n) {
    // Find min child
    j = 2 * i;
    if (j < n && A[j + 1] < A[j])
      j++;
    // Move object A[i] down if it violates the min-heap
    if (k > A[j]) {
      A[i] = A[j];
      i = j;
    }
    else break;  // No reason to move down anymore
  }
  A[i] = k;
}
```

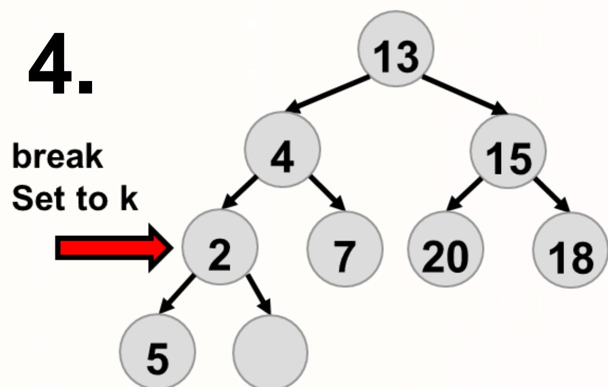
Παράδειγμα Εκτέλεσης PercolateDown

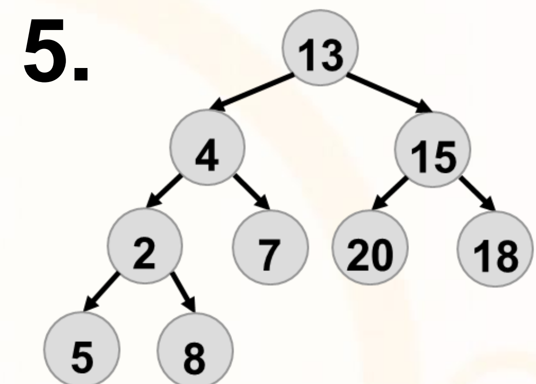
- `int A[]={-1 , 13, 8, 15, 4, 7, 20, 18, 5, 2};`
- `i=2, n=9, PercolateDown(A, 9, 2);` ➔ `k=8`
- Αν φανταστούμε τον πίνακα σαν δυαδικό δέντρο… (**προσοχή**: ο πίνακας δεν είναι heap… ακόμη!)











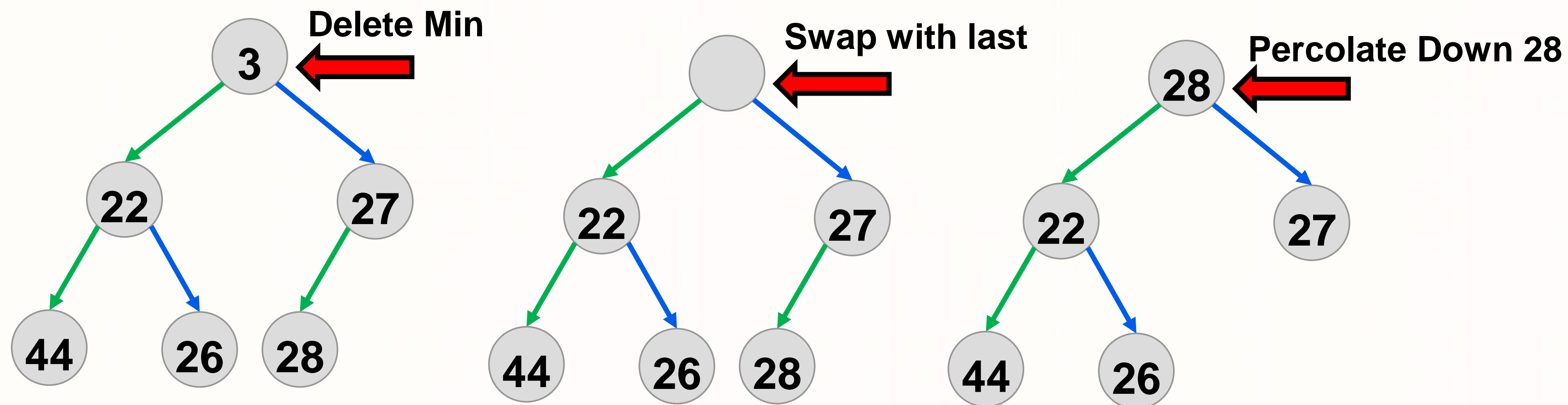
Διαδικασία DeleteMin (2) σε σωρό

- Αφαιρούμε το στοιχείο της ρίζας (είναι το μικρότερο κλειδί του σωρού).
- Μεταφέρουμε το τελευταίο κλειδί στη ρίζα, και εφαρμόζουμε τη διαδικασία PercoladeDown(A, n, 1):

```
public static int deleteMin(Heap h) {
  int min = 0;
  int swap = 0;
  if (!h.isEmpty()) {
    min = h.contents[1];
    // swap(contents[1],contents[size])
        swap = h.contents[1];
        h.contents[1] = h.contents[h.size];
        h.contents[h.size] = swap;
    h.size--;
    PercolateDown(h.contents, h.size, 1);
  }
return min;
}
```

- Χρόνος Εκτέλεσης: O(h) = O(log n)

Παράδειγμα DeleteMin(2)



22<27, 28>22
22: Move up

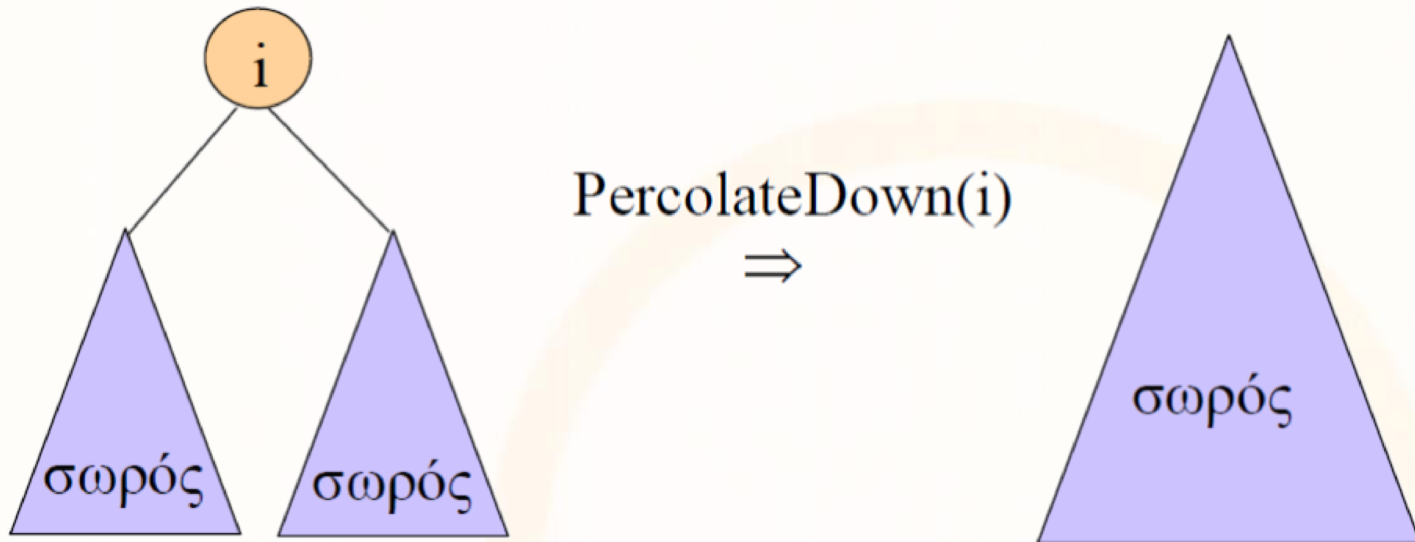
26<44, 28>26
26: Move up

break
Set to 28

0	1	2	3	4	5	6
0	22	26	27	44	28	…

Από πίνακες σε σωρούς

- Έστω πίνακας A[1..n].
- Μπορούμε να θεωρήσουμε τον πίνακα ως ένα πλήρες δυαδικό δένδρο με n κόμβους.
- Αν για μια τιμή i το αριστερό και το δεξί υπόδενδρο του i ικανοποιούν τις ιδιότητες ενός σωρού, τότε, αν καλέσουμε τη διαδικασία PercolateDown(A, n, i) θα έχουμε σαν αποτέλεσμα το υπόδενδρο που ριζώνει στη θέση i να ικανοποιεί τις ιδιότητες ενός σωρού.



Κτίσιμο σωρού από ένα πίνακα

- Μπορούμε να μετατρέψουμε ένα πίνακα A[1..n] σε ένα σωρό με διαδοχική εφαρμογή της διαδικασίας PercoladeDown() από κάτω προς τα πάνω.
- **Παρατήρηση:** οι θέσεις > n/2 αντιστοιχούν σε φύλλα.

```
public static void BuildHeap(int A[], int n) {
  for (int i = n / 2; i > 0; i--)
    PercolateDown(A, n, i);
}
```

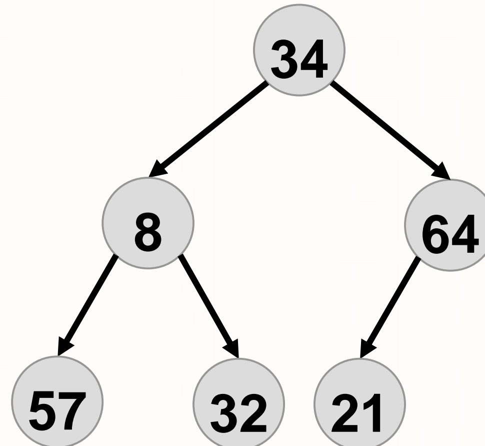
- **Ορθότητα (αποδεικνύεται με τη μέθοδο της επαγωγής):** μετά από την εφαρμογή της διαδικασίας PercoladeDown(A,n,i), τα υπόδενδρα που ριζώνουν στις θέσεις i, ..., n, ικανοποιούν τις ιδιότητες σωρού.
- **Ανάλυση του Χρόνου Εκτέλεσης:** Ο ολικός χρόνος εκτέλεσης είναι ανάλογος του αθροίσματος των υψών όλων των εσωτερικών κόμβων, το οποίο είναι O(n).

Τι κάνει ο πιο κάτω αλγόριθμος;

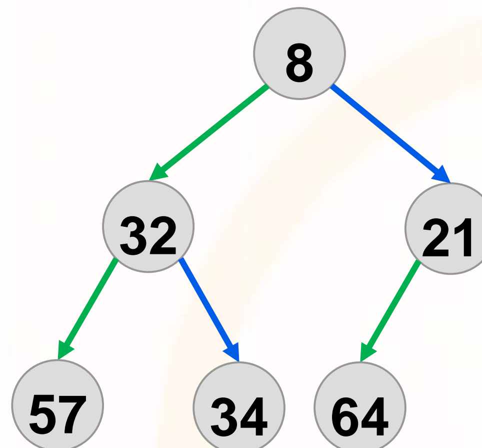
```
public static void mystery(int A[], int n) {
  BuildHeap(A, n);
  int swap;
  for (int i = n; i > 1; i--) {
    // swap (A[1], A[i]);
    swap = A[1];
    A[1] = A[i];
    A[i] = swap;
    PercolateDown(A, i - 1, 1);
  }
}
```

Παράδειγμα Εκτέλεσης Διαδικασίας mystery

- Είσοδος, A={ -(6)-, 34, 8, 64, 57, 32, 21}

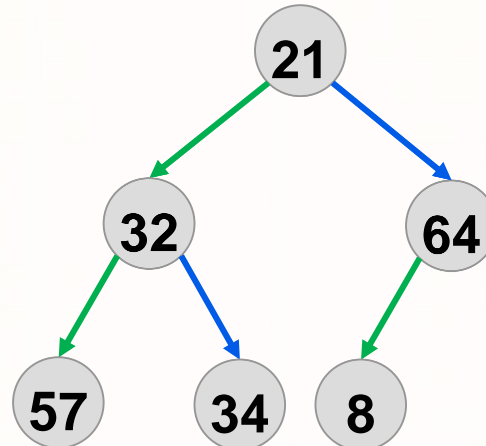


- Μετά την εκτέλεση της γραμμής BuildHeap

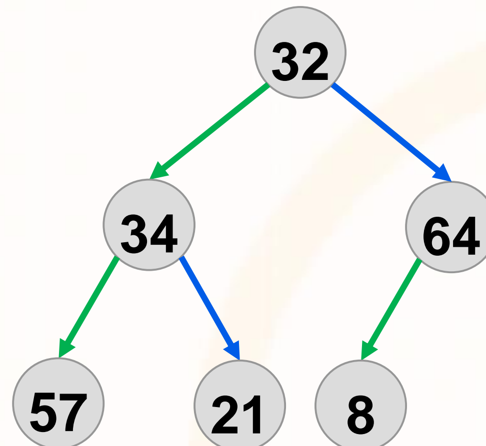


Παράδειγμα Εκτέλεσης Διαδικασίας mystery

- Μετά από την πρώτη επανάληψη του for

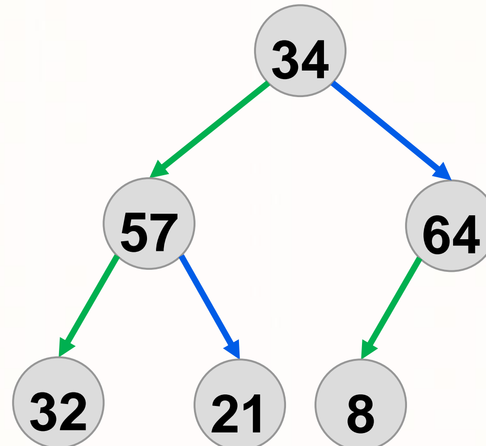


- Μετά από την δεύτερη επανάληψη του for

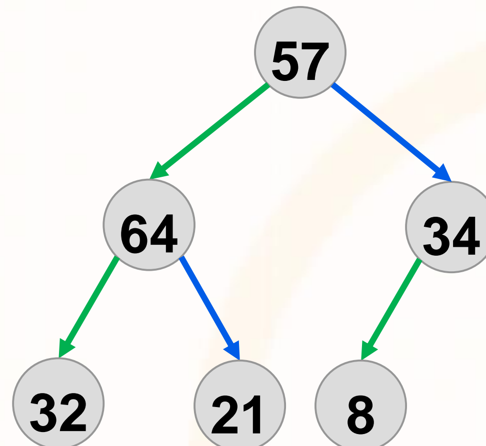


Παράδειγμα Εκτέλεσης Διαδικασίας mystery

- Μετά από την τρίτη επανάληψη του for

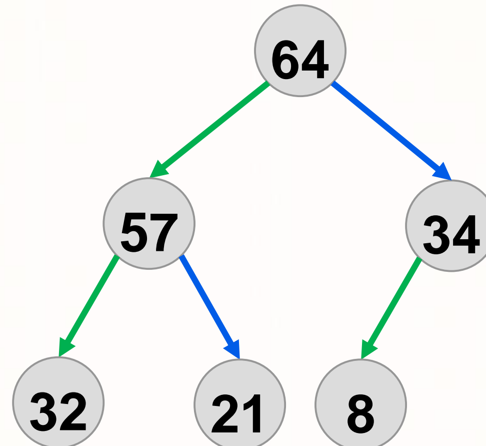


- Μετά από την τέταρτη επανάληψη του for



Παράδειγμα Εκτέλεσης Διαδικασίας mystery

- Μετά από την πέμπτη επανάληψη του for



- Σε επίπεδο πίνακα

-	64	57	34	32	21	8

Ο αλγόριθμος ταξινόμησης HeapSort

- Η διαδικασία mystery ταξινομεί ένα πίνακα σε φθίνουσα σειρά.
- Αρχικά δημιουργεί ένα σωρό σε χρόνο O(n).
- Στη συνέχεια επαναλαμβάνει το εξής: αφαιρεί το μικρότερο στοιχείο (της ρίζας του σωρού) και το μετακινεί στο τέλος (εκτελεί την PercolateDown). Κάθε εκτέλεση της PercoladeDown χρειάζεται χρόνο της τάξης O(log n).
- Ολικός Χρόνος Εκτέλεσης: O(n·log n)

- **Ο αλγόριθμος ονομάζεται Heapsort**

- Μπορούμε εύκολα να αλλάξουμε τον κώδικα ώστε να επιστρέφεται η λίστα σε αύξουσα σειρά.

Άλλες διαδικασίες σε σωρούς

- Παρόλο που εύρεση του ελάχιστου κλειδιού σε ένα σωρό μπορεί να πραγματοποιηθεί σε σταθερό χρόνο, η εύρεση τυχαίου στοιχείου στη χειρότερη περίπτωση επιβάλλει διερεύνηση ολόκληρης της δομής (δηλαδή, είναι της τάξης O(n)).
- Αν όμως γνωρίζουμε τη θέση στοιχείων με κάποιο άλλο τρόπο, διαδικασίες σε σωρούς πραγματοποιούνται εύκολα, π.χ. οι πιο κάτω εκτελούνται σε χρόνο λογαριθμικό.
- Increase_Key(P,Δ), αυξάνει την προτεραιότητα του κλειδιού P, κατά Δ. Χρησιμοποιείται από χειριστές λειτουργικών συστημάτων για αύξηση της προτεραιότητας σημαντικών διεργασιών. Η συμμετρική διαδικασία Decrease_Key(P,Δ) συχνά εκτελείται αυτόματα σε λειτουργικά συστήματα σε περίπτωση που κάποια δουλειά χρησιμοποιεί υπερβολικά μεγάλη ποσότητα χρόνου του CPU.
- Remove(I), αφαιρεί τον κόμβο της θέσης I (χρήσιμη σε περίπτωση τερματισμού διαδικασίας).

Συγχώνευση Σωρών (Merge Heap)

- Υποθέστε την ύπαρξη των δυαδικών σωρών h1 και h2. Πως μπορούν να **συγχωνευτούν** σε ένα καινούριο σωρό h;

- **Προσπάθεια Α:** Πρόσθεσε όλα τα στοιχεία του σωρού h1 στον h2.

  Χρόνος Εκτέλεσης;

- **Προσπάθεια Β:** Βρες τον πιο μικρό από τους δύο σωρούς (έστω h2) και πρόσθεσε όλα τα στοιχεία του σωρού h2 στον h1.

  Χρόνος Εκτέλεσης;

- **Προσπάθεια Γ:** Συνένωσε τους δύο σωρούς (δηλ., δημιούργησε ένα καινούριο πίνακα και αποθήκευσε τα στοιχεία του σωρού h1 πρώτα και μετά τα στοιχεία του σωρού h2) και μετά τρέξε τη διαδικασία BuildHeap.

  Χρόνος Εκτέλεσης;

Προσπάθεια Δ: Leftist Heaps

- Ιδέα: Η συντήρηση της σωρού να γίνεται σε ένα μικρό μέρος της σωρού
  - Τα περισσότερα στοιχεία βρίσκονται στα **αριστερά**
  - Η διαδικασία της συγχώνευσης γίνεται στα **δεξιά**

- Πως το επιτυγχάνουμε;

- Ορισμός **Null Path Length**
  Null Path Length (npl) κάποιου κόμβου ***u***: ο αριθμός των κόμβων μεταξύ του ***u*** και μίας κενής αναφοράς (null) σε κάποιο από τα υποδέντρα του

- Εναλλακτικός ορισμός: Ελάχιστη απόσταση ενός κόμβου από κάποιο απόγονο που έχει 0 ή 1 παιδιά

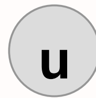
Ιδιότητες Leftist Heap

- **Ιδιότητα Σειράς (Σωρού)**
  - Η τιμή κάποιου κόμβου είναι μικρότερη από αυτή των παιδιών του
  - Αποτέλεσμα: ο μικρότερος κόμβος είναι η ρίζα

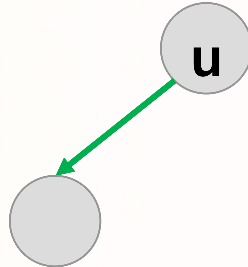
- **Ιδιότητα Leftist**
  - Για κάθε κόμβο **u** npl(**u**.left) ≥ npl(**u**.right)
  - Αποτέλεσμα: για κάθε κόμβο το αριστερό του υποδέντρο είναι τουλάχιστον τόσο βαρύ όσο το δεξί του υποδέντρο

Υπολογισμός Null Path Length (npl)

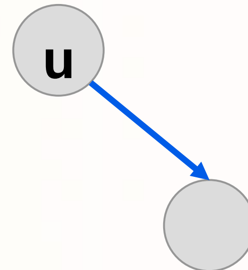
- npl(null) = -1



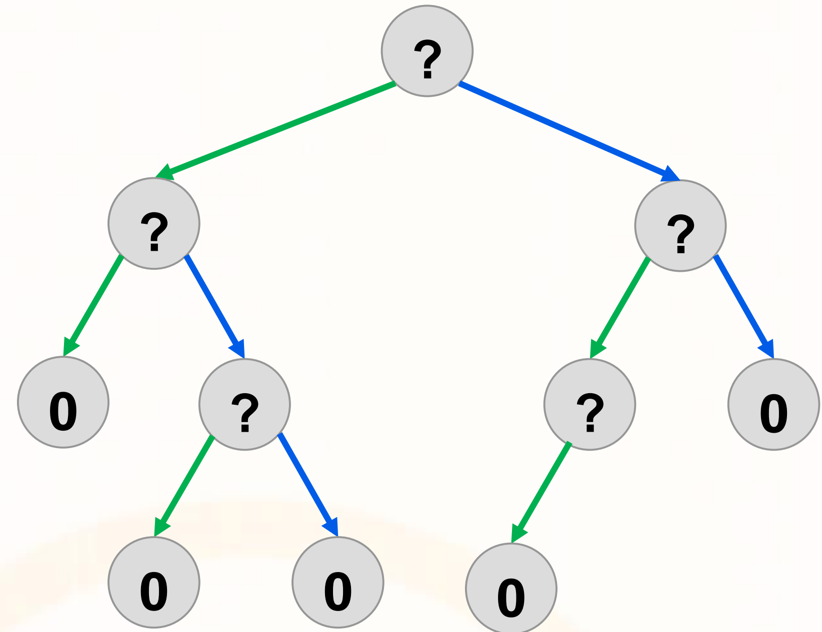
- npl(u) = ?



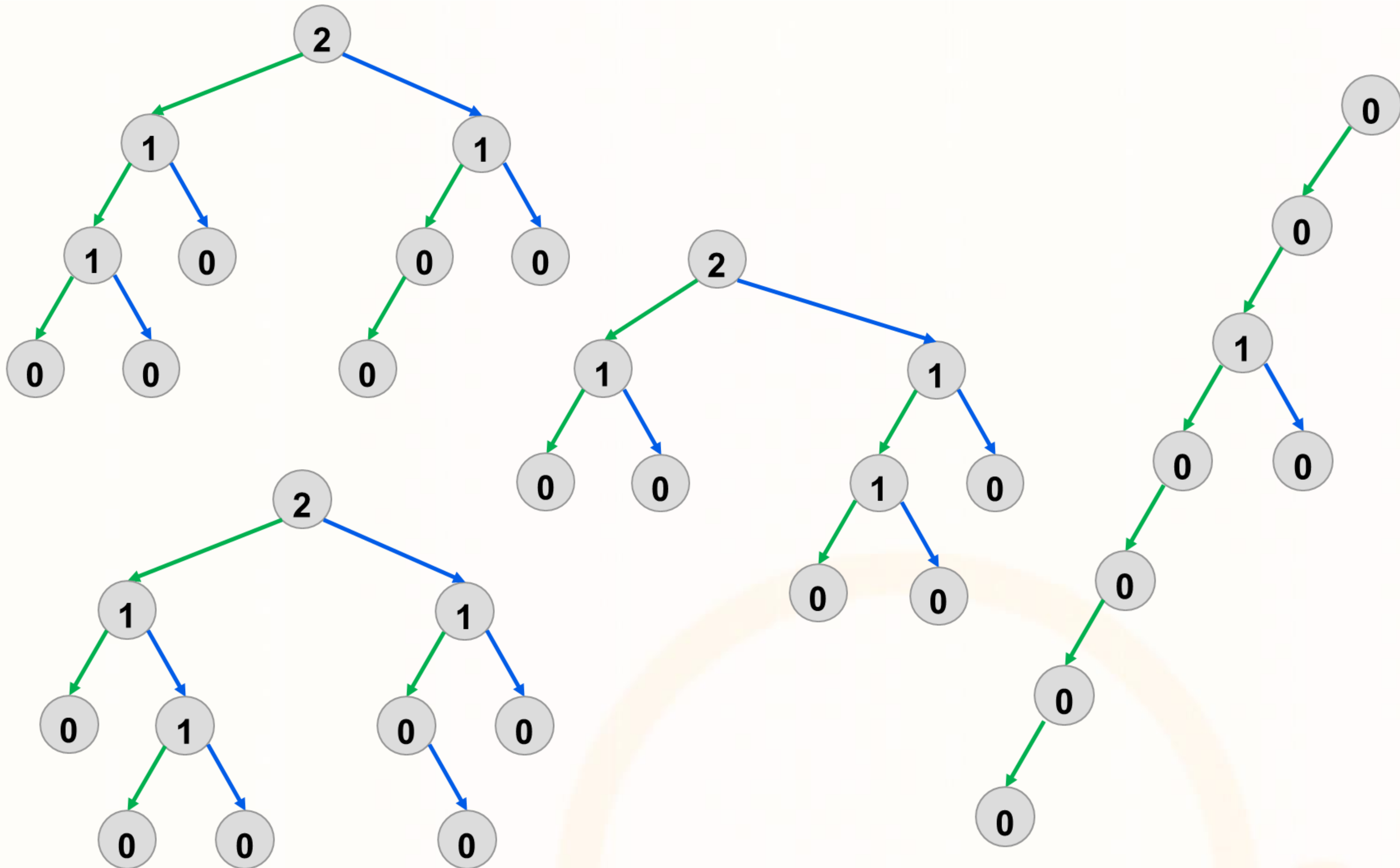
- npl(u) = ?



- npl(u) = ?



Είναι τα πιο κάτω δέντρα Leftist;



Γιατί χρειαζόμαστε την ιδιότητα Leftist

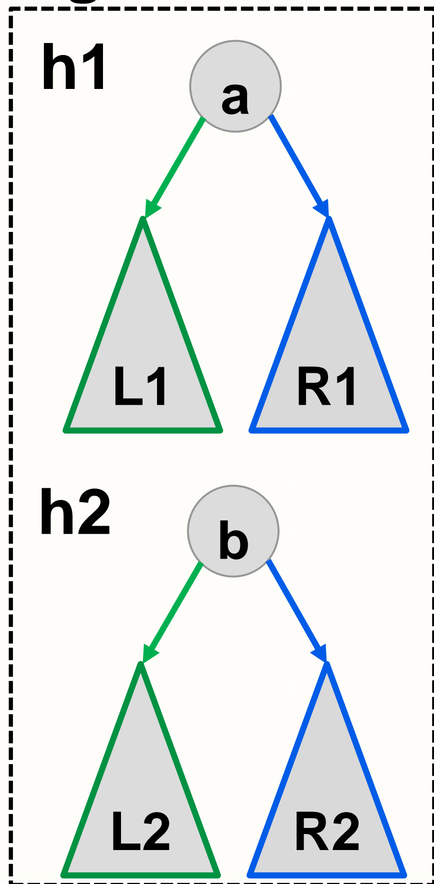
- Επειδή εγγυάται τα ακόλουθα:
  - Το δεξί μονοπάτι είναι «κοντό» σε σχέση με τον αριθμό των κόμβων στο δέντρο
  - Ένας σωρός leftist με N κόμβους έχει δεξί μονοπάτι με το πολύ log (N+1) κόμβους
- Αποτέλεσμα: κάνε τις αλλαγές στο δεξί «κοντό» υποδέντρο

- Πως θα εκτελεστεί η Συγχώνευση δύο σωρών h1 και h2;
- Βασική Ιδεά:
  - Το πιο μικρό στοιχείο από h1 και h2 πρέπει να είναι η καινούρια ρίζα (έστω του h1)
  - Το αριστερό υποδέντρο του h1 παραμένει αριστερά
  - Αναδρομικά συγχώνευσε το δεξί υποδέντρο με το h2

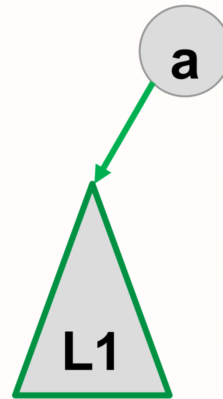
Συγχώνευση δύο Leftist Σωρών

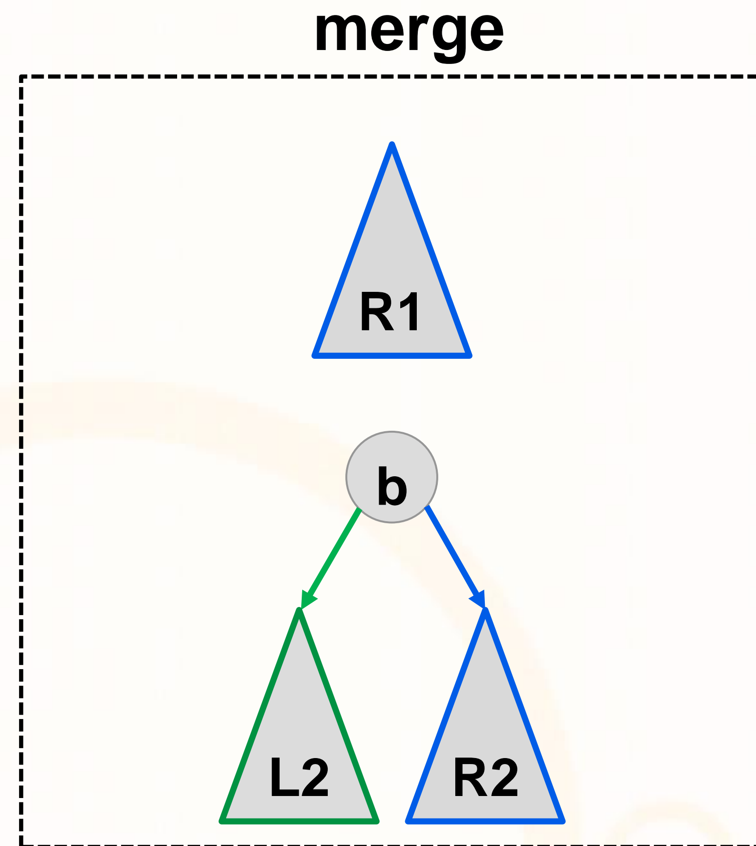
- **MergeHeaps(h1, h2):** επιστρέφει ένα leftist σωρό που περιέχει τα στοιχεία (διακριτά στοιχεία) των leftist σωρών h1 και h2.

merge

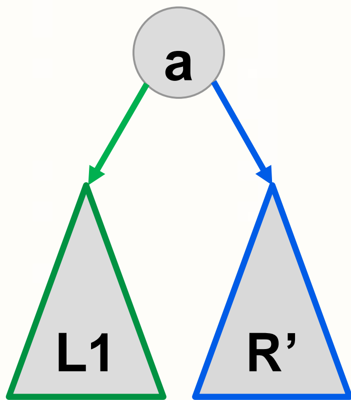


a < b



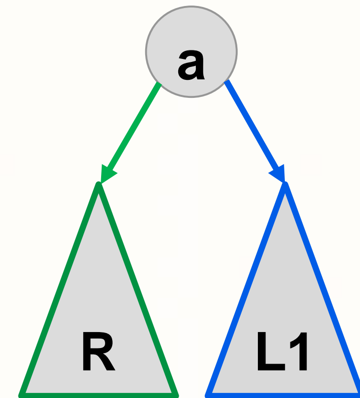


Συγχώνευση δύο Leftist Σωρών (συν.)



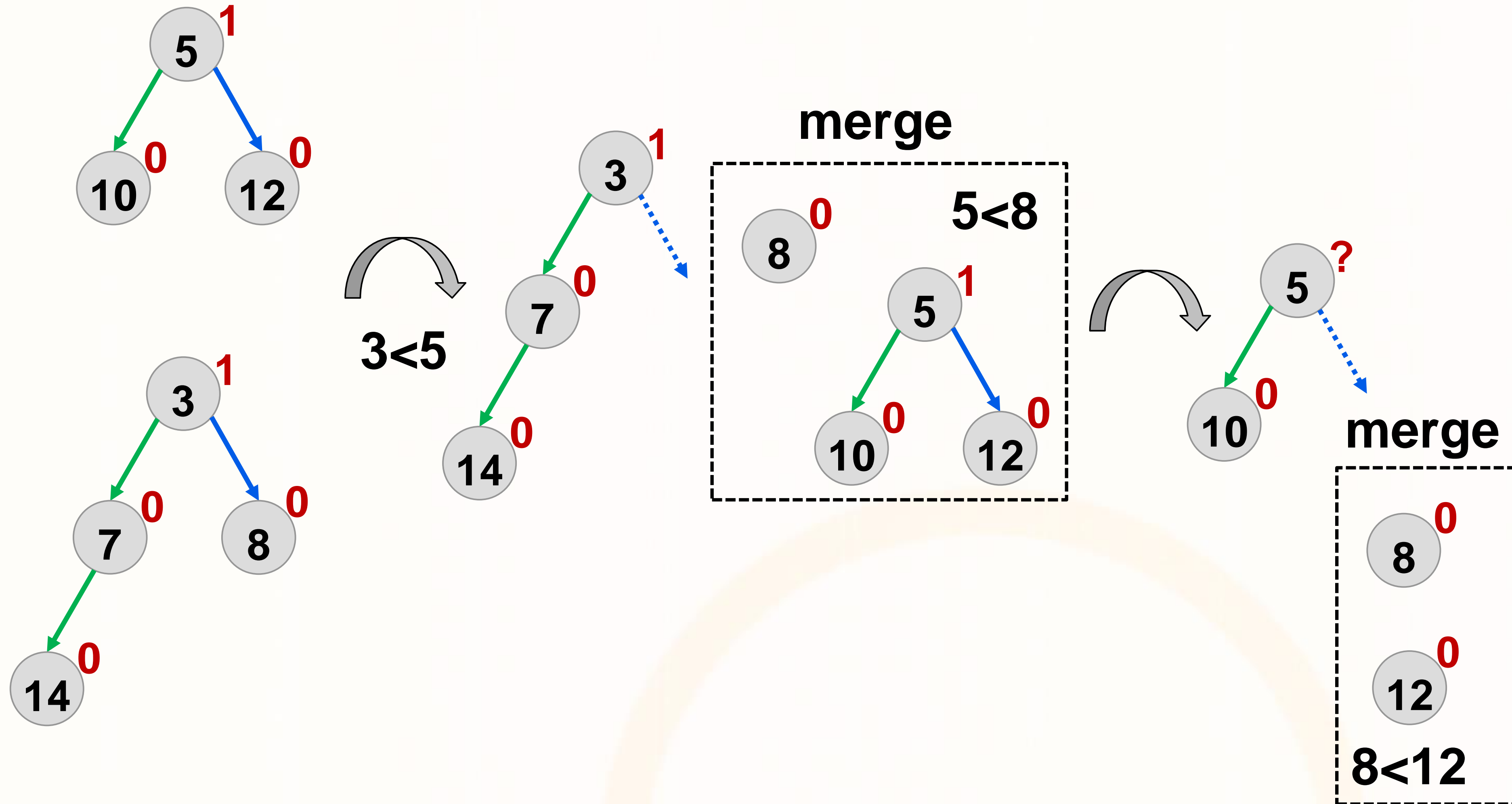
npl(L1) < npl(R’)

swap

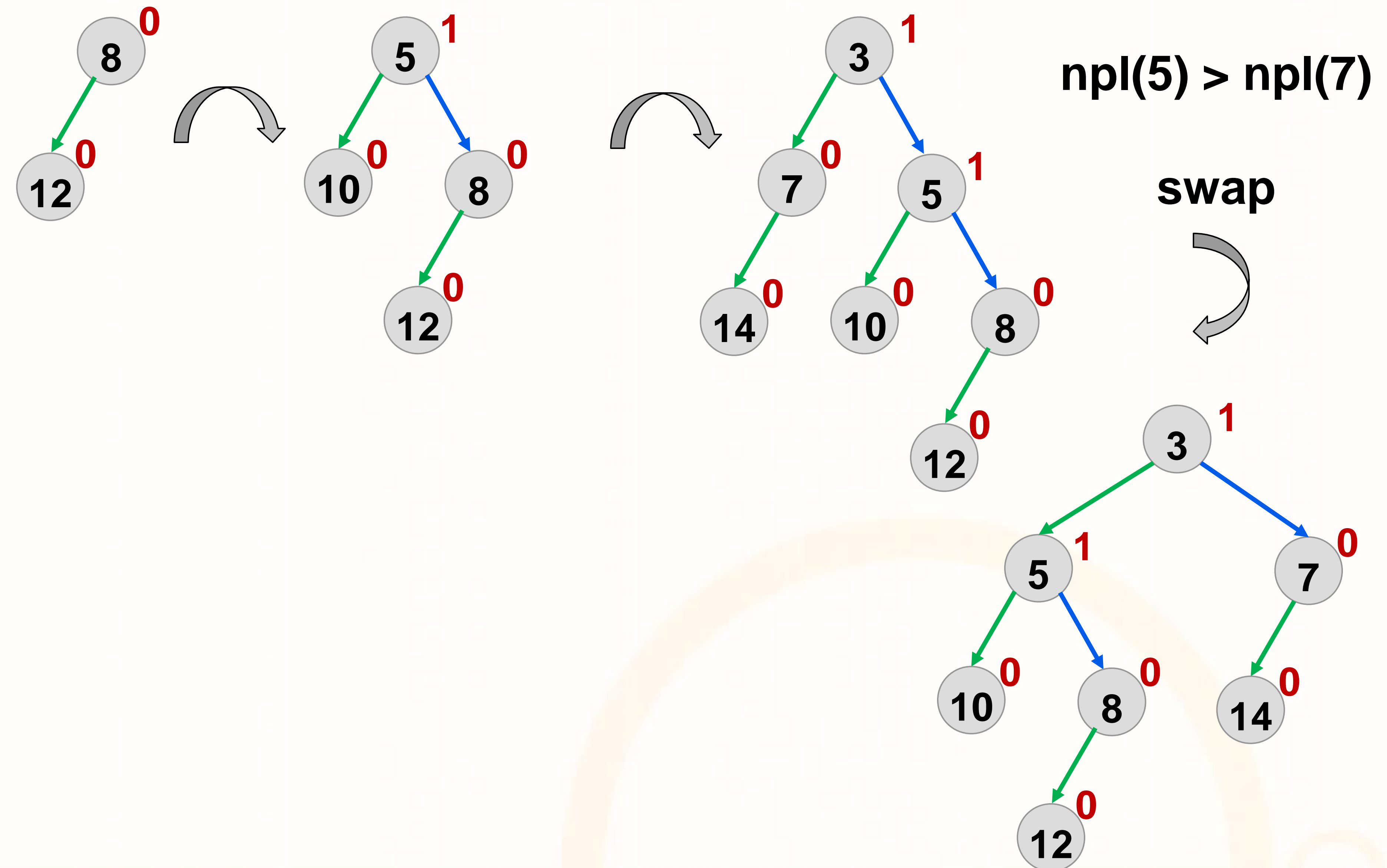


Χρονική Πολυπλοκότητα Συγχώνευσης;

Παράδειγμα MergeHeaps 1



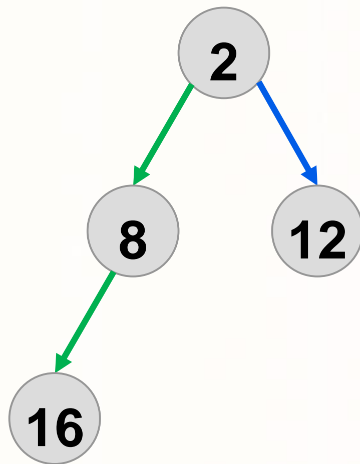
Παράδειγμα MergeHeaps 1 (συν.)



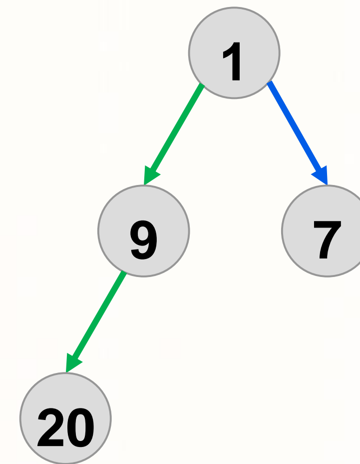
Παρατηρήσεις

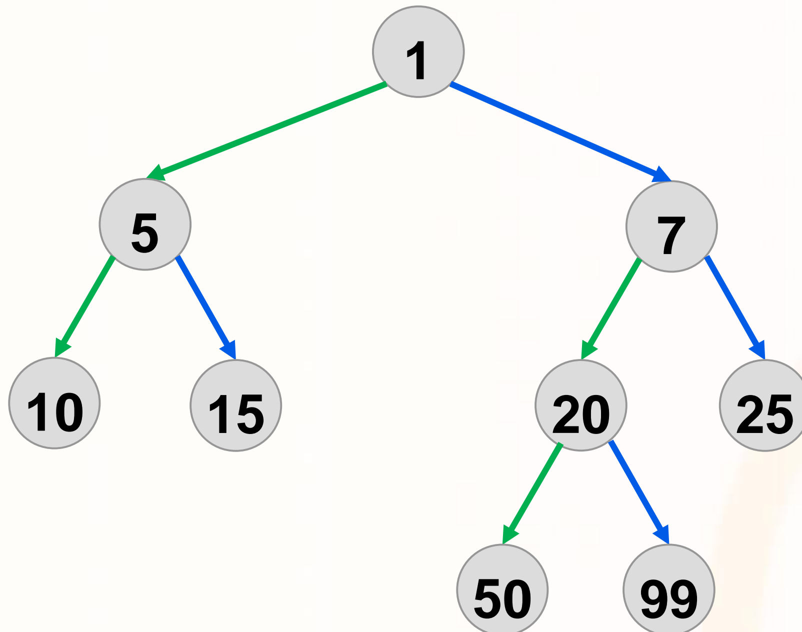
- Η χρονική πολυπλοκότητα της **MergeHeaps** είναι **O(log n)**
- Η χρονική πολυπλοκότητα της **Insert** σε leftist σωρό με n στοιχεία;
  - Χρήση της merge με h1=τον σωρό και τον καινούριο κόμβο σαν σωρό h2
- Η χρονική πολυπλοκότητα της **Delete** σε leftist σωρό με n στοιχεία;
  - Αφαίρεση της ρίζας και συγχώνευση του αριστερού και δεξιού υποδέντρου με χρήση της merge
- Προβλήματα leftist σωρών:
  - Μνήμη;
  - Πολυπλοκότητα;
  - Το δεξιό υποδέντρο είναι συνήθως πιο «βαρετό» και χρειάζεται αλλαγή. Τι σημαίνει αυτό σε επίπεδο πίνακα;

Άσκηση 1: Εκτέλεση MergeHeaps

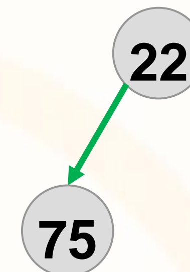


merge with





merge with



Άσκηση 2: Διαδικασία MergeHeaps

- Γράψετε με ψευδοκώδικα την διαδικασία MergeHeaps

Άσκηση 2: Διαδικασία MergeHeaps

- Γράψετε με ψευδοκώδικα την διαδικασία MergeHeaps

```
/**
 * Internal method to merge two roots.
 * Deals with deviant cases and calls recursive merge1.
 */
private Node<AnyType> merge( Node<AnyType> h1, Node<AnyType> h2 )
{
    if( h1 == null )
        return h2;
    if( h2 == null )
        return h1;
    if( h1.element.compareTo( h2.element ) < 0 )
        return merge1( h1, h2 );
    else
        return merge1( h2, h1 );
}
```

Άσκηση 2: Διαδικασία MergeHeaps

```
/**
 * Internal method to merge two roots.
 * Assumes trees are not empty, and h1's root contains smallest item.
 */
private Node<AnyType> merge1( Node<AnyType> h1, Node<AnyType> h2 )
{
    if( h1.left == null )   // Single node
        h1.left = h2;       // Other fields in h1 already accurate
    else
    {
        h1.right = merge( h1.right, h2 );
        if( h1.left.npl < h1.right.npl )
            swapChildren( h1 );
        h1.npl = h1.right.npl + 1;
    }
    return h1;
}
```

DEFINITION: A ***nearly complete binary tree*** of height h is a binary tree of height h in which

a) There are 2^d nodes at depth d for $d = 1, 2, \ldots, h-1$,

b) The nodes at depth h are as far left as possible.

- Condition (b) can be stated more rigorously, like this:

  If a node p at depth $h-1$ has a left child, then every node at depth $h-1$ to the left of p has 2 children. If a node at depth $h-1$ has a right child, then it also has a left child.

- The relationship between the height and number of nodes in a nearly complete binary tree is given by

$$2^h \le n \le 2^{h+1} - 1, \text{ or } \quad h = \lfloor \lg(n) \rfloor.$$

(This depends only on condition (a) in the definition.)

http://homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/nearly_complete.pdf