Δομές Δεδομένων

12η Διάλεξη

Διάσχιση Δέντρων και Ουρές Προτεραιότητας

Ε. Μαρκάκης

Περίληψη

- Διάσχιση δέντρων
- Ουρές προτεραιότητας
- Στοιχειώδεις υλοποιήσεις
- Δομή δεδομένων σωρού
- Αλγόριθμοι σε σωρούς
- Ο αλγόριθμος heapsort

Διάσχιση δέντρου

- Διάσχιση δέντρου (tree traversal)
  - Έστω ότι θέλουμε να επεξεργαστούμε όλους τους κόμβους του δέντρου με συστηματικό τρόπο
  - 4 βασικές επιλογές διάσχισης
  - Προδιατεταγμένη (preorder): επεξεργαζόμαστε πρώτα έναν κόμβο, και μετά επεξεργαζόμαστε το αριστερό και δεξιό υποδέντρο
  - Ενδοδιατεταγμένη (inorder): επεξεργαζόμαστε πρώτα το αριστερό υποδέντρο, μετά τον κόμβο, και μετά το δεξί υποδέντρο
  - Μεταδιατεταγμένη (postorder): αριστερό υποδέντρο, δεξί υποδέντρο, και στο τέλος ο κόμβος
  - Διάσχιση ανά επίπεδο (level-order): πρώτα η ρίζα, μετά όλοι οι κόμβοι στο επίπεδο 1, κ.ο.κ. (ή με ανάποδη σειρά από κάτω προς τα πάνω)

Διάσχιση δέντρου

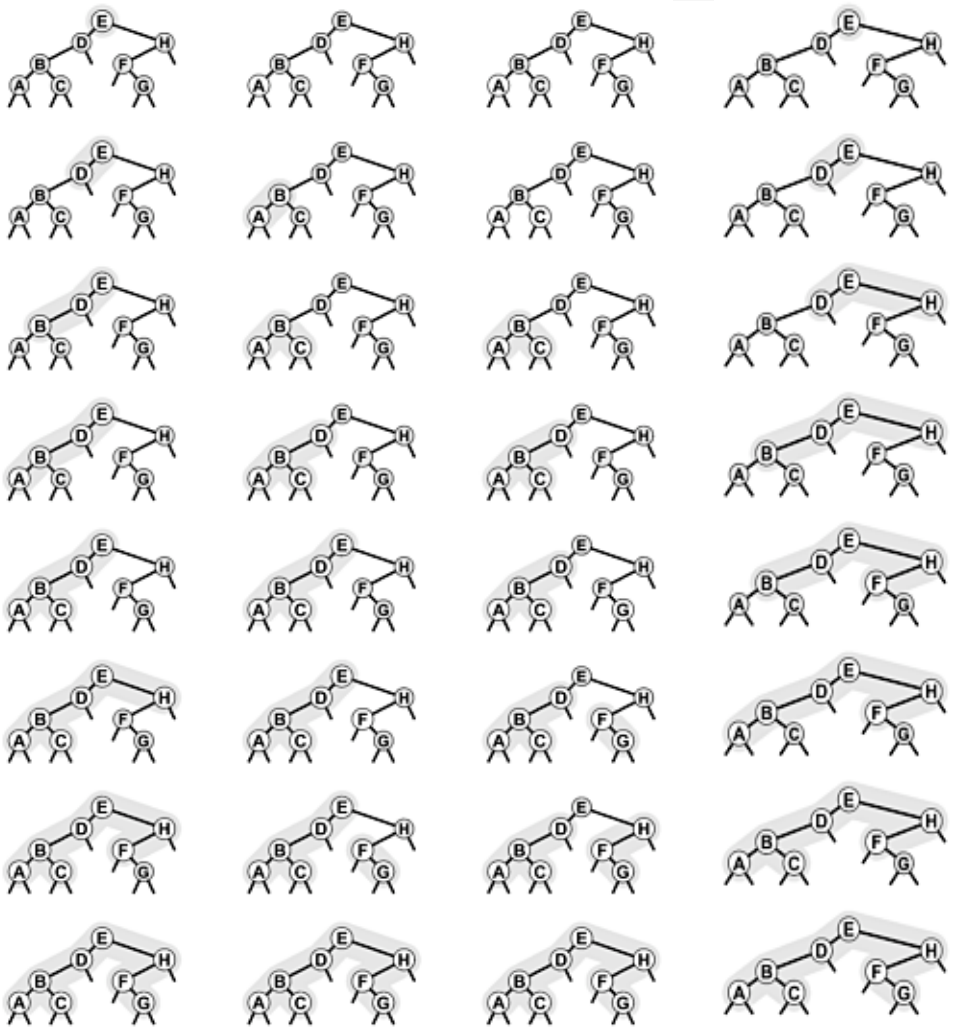
- Αναδρομική υλοποίηση για preorder, inorder, postorder
- Γενική μορφή κώδικα διάσχισης (preorder)

```
//έστω visit() μία μέθοδος που επεξεργάζεται έναν κόμβο
/*η διάσχιση παίρνει ως όρισμα τον δείκτη από όπου ξεκινάμε
  π.χ. τη ρίζα του δέντρου */
private void traverseR(Node h) {
  if (h == null) return;
  h.item.visit();
  traverseR(h.l);
  traverseR(h.r); }
void traverse() { traverseR(root); }
```

- Παρόμοιος κώδικας για `inorder, postorder`

Διάσχιση δέντρου

- Παράδειγμα διάσχισης
  - Α) προδιατεταγμένη
  - Β) ενδοδιατεταγμένη
  - Γ) μεταδιατεταγμένη
  - Δ) ανά επίπεδο



Διάσχιση δέντρου

- Μη αναδρομική διάσχιση δέντρου
  - Στην αναδρομή οι κλήσεις μπαίνουν στη στοίβα
  - Χρησιμοποιώντας τη δική μας στοίβα δεν χρειάζεται αναδρομή
  - Αντί για κλήση, ωθούμε τον κόμβο ή το υποδέντρο στη στοίβα

```
private void traverseS(Node h) {
  NodeStack s = new NodeStack(max);
  s.push(h);
  while (!s.empty()) {
       h = s.pop();
       h.item.visit();
       if (h.r != null) s.push(h.r);
       if (h.l != null) s.push(h.l); } }
void traverseS() { traverseS(root); }
```

Διάσχιση δέντρου

- Διάσχιση δέντρου ανά επίπεδο
  - Επισκεπτόμαστε τους κόμβους στο ίδιο επίπεδο με τη σειρά
  - Απλά αντικαθιστούμε τη στοίβα με ουρά
    - Επεξεργαζόμαστε τους κόμβους με τη σειρά που τους βλέπουμε
  - Δεν υπάρχει προφανής αναδρομική υλοποίηση

```
private void traverseQ(Node h) {
  NodeQueue q = new NodeQueue(max);
  q.put(h);
  while (!q.empty()) {
       h = q.get();
       h.item.visit();
       if (h.l != null) q.put(h.l);
       if (h.r != null) q.put(h.r); } }
void traverseQ() { traverseQ(root); }
```

Ουρές Προτεραιότητας

- Έστω N ασθενείς με νέα γρίπη στα επείγοντα ενός νοσοκομείου
- Κάθε ασθενής έχει μία προτεραιότητα ανάλογα με τη σοβαρότητα της κατάστασης (π.χ. περιστατικά όπου απειλείται άμεσα η ζωή του ασθενούς έχουν μεγαλύτερη προτεραιότητα)
- Ανά πάσα στιγμή μπορεί να εισέρχονται νέα περιστατικά (N++), ενδεχομένως με υψηλότερη προτεραιότητα
- Οι γιατροί μετά το τέλος μίας εξέτασης πρέπει να δέχονται ως επόμενο ασθενή τον ασθενή με την τρέχουσα υψηλότερη προτεραιότητα από αυτους που περιμένουν

Ουρές προτεραιότητας

- Λειτουργία ουράς προτεραιότητας (priority queue)
  - Δομή δεδομένων για στοιχεία με κλειδιά
  - Υποστηριζόμενες λειτουργίες:
  - Εισαγωγή νέου στοιχείου
  - Αφαίρεση στοιχείου με το *μεγαλύτερο* κλειδί (ή το μικρότερο: ανάποδος σωρός)

- Πολλές πρακτικές εφαρμογές
  - Συστήματα προσομοιώσεων: χρονολογική επεξεργασία γεγονότων
  - Λειτουργικά συστήματα: επιλογή διεργασιών (κλειδιά αντιστοιχούν σε προτεραιότητες που δείχνουν ποιοι χρήστες πρέπει να εξυπηρετηθούν πρώτοι)

Ουρές προτεραιότητας

- Αφηρημένος τύπος δεδομένων ουράς προτεραιότητας

```
class PQ {
  PQ(int)      // το μέγιστο μέγεθος της ουράς
  boolean empty()
  void insert(ITEM)
  ITEM getmax() };
```

- Άλλες πιθανές επιθυμητές λειτουργίες ουρών προτεραιότητας
  - Αλλαγή προτεραιότητας κάποιου στοιχείου
  - Αφαίρεση (τυχαίου) στοιχείου
  - Ένωση ουρών προτεραιότητας
  - Μπορεί να υλοποιούνται με απλούστερες λειτουργίες
  - Οι εξειδικευμένες υλοποιήσεις είναι όμως πιο γρήγορες

Ουρές προτεραιότητας

- Ουρές προτεραιότητας και ταξινόμηση
  - Μπορούν να χρησιμοποιηθούν για ταξινόμηση
    - Εισάγουμε όλα τα στοιχεία στην ουρά
    - Τα αφαιρούμε με σειρά μεγέθους
  - Δεν είναι όμως μόνο εργαλεία ταξινόμησης

- Γενίκευση άλλων δομών δεδομένων
  - Στοίβα: ουρά προτεραιότητας όπου το στοιχείο που ωθείται τελευταίο έχει τη μεγαλύτερη προτεραιότητα
  - Ουρά FIFO: ουρά προτεραιότητας όπου το στοιχείο που έχει παραμείνει τον περισσότερο χρόνο έχει τη μεγαλύτερη προτεραιότητα

Στοιχειώδεις υλοποιήσεις

- Μπορεί να υλοποιηθεί με λίστα ή πίνακα
- Ταξινομημένος ή αταξινόμητος πίνακας;
  - Αταξινόμητος πίνακας (πρόγραμμα)
    - Ράθυμη (lazy) προσέγγιση
    - Εισαγωγή: σταθερό κόστος
    - Αφαίρεση μεγίστου: γραμμικό κόστος (διατρέχουμε όλο τον πίνακα)
  - Ταξινομημένος πίνακας
    - Πρόθυμη (eager) προσέγγιση
    - Εισαγωγή: γραμμικό κόστος (πρέπει το στοιχείο να μπει στη σωστή θέση)
    - Αφαίρεση μεγίστου: σταθερό κόστος
- Σταθερό κόστος και για τις 2 λειτουργίες?
  - Πιο δύσκολο πρόβλημα!

Στοιχειώδεις υλοποιήσεις

```
class PQ {
  static boolean less(ITEM v, ITEM w) {
       return v.less(w); }
  static void exch(ITEM[] a, int i, int j) {
       ITEM t = a[i]; a[i] = a[j]; a[j] = t; }
  private ITEM[] pq;
  private int N;
  PQ(int maxN) { pq = new ITEM[maxN]; N = 0; }
  boolean empty() { return N == 0; }
  void insert(ITEM item) { pq[N++] = item; }
  ITEM getmax() {
       int max = 0; for (int j = 1; j < N; j++)
              if (less(pq[max], pq[j])) max = j;
       exch(pq, max, N-1);
       return pq[--N]; } };
```

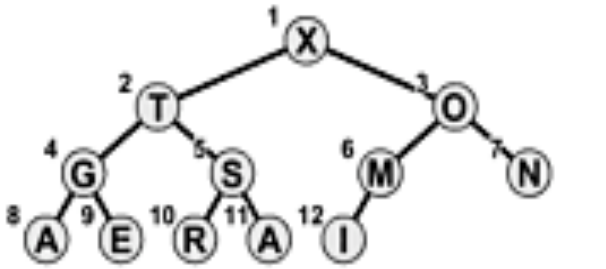
Στοιχειώδεις υλοποιήσεις

- Όλες οι άλλες πράξεις έχουν ανάλογα κόστη
  - Αφαίρεση, αλλαγή προτεραιότητας
  - Γραμμικό ή σταθερό ανάλογα με την υλοποίηση
- Μικρή ωφέλεια αν χρησιμοποιήσουμε λίστα
  - Αντισταθμίζεται από το κόστος των δεικτών
- Πολυπλοκότητα σε ένα πρόγραμμα πελάτη εξαρτάται και από τη συχνότητα εμφάνισης των λειτουργιών εισαγωγής ή αφαίρεσης

Δομή δεδομένων σωρού

- Δυαδικό δέντρο: κάθε κόμβος έχει 0 ή 1 ή 2 παιδιά
- Πλήρες δυαδικό δέντρο: όλα τα επίπεδα γεμάτα εκτός του τελευταίου, στο οποίο οι κόμβοι είναι όσο το δυνατόν αριστερότερα
    - Για N κόμβους έχουμε το πολύ logN επίπεδα
- Σωρός (Heap): πλήρες δυαδικό δέντρο όπου το κλειδί κάθε κόμβου είναι μεγαλύτερο ή ίσο με τα κλειδιά των παιδιών του
    - Η ρίζα έχει το μεγαλύτερο κλειδί του δέντρου
    - Λέμε επίσης ότι το δέντρο είναι διεταταγμένο σε σωρό (heap-sorted)

Δομή δεδομένων σωρού

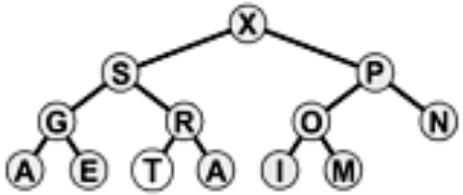


1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

- Αναπαράσταση με πίνακα
  - Η ρίζα (κόμβος 1) βρίσκεται στη θέση 1 (η θέση 0 είναι κενή)
  - Τα παιδιά του κόμβου i είναι στις θέσεις 2i και 2i+1
  - Ο γονέας του κόμβου i είναι στη θέση floor(i/2)
    - Βασιζόμαστε στο ότι το δέντρο είναι πλήρες
  - Εύκολη μετάβαση από γονείς σε παιδιά και αντίστροφα

Αλγόριθμοι σε σωρούς

- Όταν εισάγουμε ή εξάγουμε ένα στοιχείο στο σωρό ενδέχεται να παραβιαστεί η ιδιότητα του σωρού

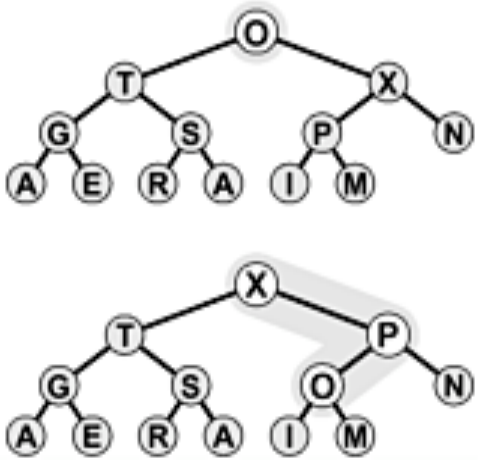


- Αποκατάσταση ιδιότητας σωρού: περίπτωση 1
  - Το κλειδί ενός κόμβου είναι μεγαλύτερο από αυτό του γονέα του (έστω ότι τα υπόλοιπα κλειδιά είναι ok)
  - Συμβαίνει όταν εισάγουμε νέους κόμβους
    - Συνήθως οι εισαγωγές γίνονται στο τέλος του σωρού
  - Ανάδυση: Ανεβάζουμε τον κόμβο προς τα επάνω (swim)
    - Αν είναι μεγαλύτερος από τον πατέρα, θα είναι και από τον αδελφό


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

Αλγόριθμοι σε σωρούς

- Αποκατάσταση ιδιότητας σωρού: περίπτωση 2
  - Το κλειδί ενός κόμβου είναι μικρότερο από αυτά των παιδιών του
  - Συμβαίνει όταν κάνουμε εξαγωγή και αντικατάσταση
  - Κατάδυση: κατεβάζουμε τον κόμβο προς τα κάτω (sink)
  - Ανεβάζουμε το μεγαλύτερο από τα παιδιά του




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

Αλγόριθμοι σε σωρούς

- Υλοποίηση ουράς προτεραιότητας με σωρό
  - Εισαγωγή: βάζουμε το στοιχείο στο τέλος του σωρού και κάνουμε swim μέχρι να αποκατασταθεί η ιδιότητα του σωρού
  - Αφαίρεση μεγίστου: ανταλλαγή ρίζας με τελευταίο στοιχείο, επιστροφή τελευταίου στοιχείου και sink της νέας ρίζας
- Κόστος πράξεων
  - Ύψος πλήρους δυαδικού δέντρου με N στοιχεία: logN
  - Σε κάθε λειτουργία διατρέχουμε ένα μονοπάτι από τη ρίζα ως κάποιο φύλλο
  - Εισαγωγή: το πολύ logN συγκρίσεις
    - Στη χειρότερη περίπτωση μία σύγκριση ανά επίπεδο
  - Εξαγωγή μέγιστου: το πολύ 2logN συγκρίσεις
    - Δύο συγκρίσεις (με τα δύο παιδιά) ανά επίπεδο
  - O(logN) και για τις 2 λειτουργίες

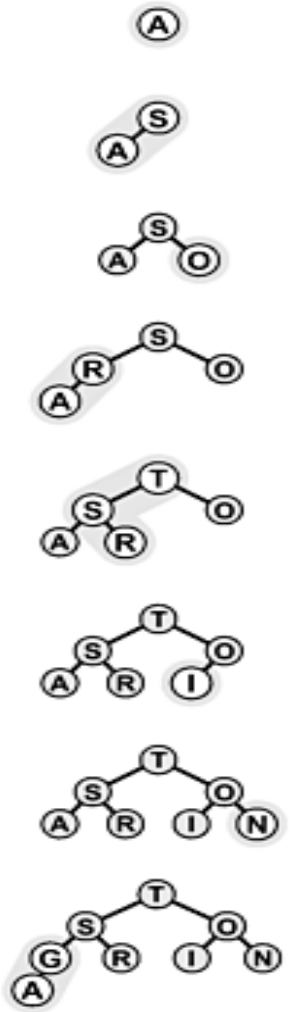
Αλγόριθμοι σε σωρούς

- Υλοποίηση ουράς προτεραιότητας με σωρό

```
class PQ {
  private boolean less(int i, int j) {
       return pq[i].less(pq[j]); }
  private void exch(int i, int j) {
       ITEM t = pq[i]; pq[i] = pq[j]; pq[j] = t; }
  private void swim(int k)
  private void sink(int k, int N)
  private ITEM[] pq; private int N;
  PQ(int maxN) { pq = new ITEM[maxN+1]; N = 0; }
  boolean empty() { return N == 0; }
  void insert(ITEM v) { pq[++N] = v; swim(N); }
  ITEM getmax() {
       exch(1, N); sink(1, N-1); return pq[N--]; } };
```

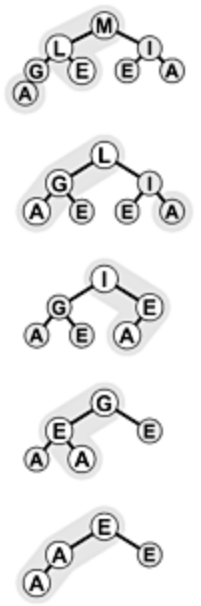
Αλγόριθμοι σε σωρούς

- Κατασκευή σωρού από τυχαία στοιχεία
  - Διαδοχικές εισαγωγές: εισαγωγή στο τέλος και swim
  - Όταν ο σωρός έχει ήδη i στοιχεία, κόστος εισαγωγής = log i
  - Συνολικό κόστος για κατασκευή σωρού με N στοιχεία: log 1 + log 2+…+logN ≤ NlogN
  - Στη μέση περίπτωση απαιτεί γραμμικό χρόνο
    - Θα βελτιώσουμε τη χειρότερη περίπτωση σε λίγο
- Άλλες λειτουργίες ουράς προτεραιότητας:
  - Οι λειτουργίες αλλαγής προτεραιότητας και αφαίρεσης μπορούν να υλοποιηθούν σε χρόνο O(logN)



Αλγόριθμοι σε σωρούς

- Απλή ταξινόμηση με σωρό
  - Προσοχή: ο σωρός ΔΕΝ είναι ταξινομημένος
    - Η ιδιότητα του σωρού ισχύει μόνο μεταξύ γονέα-παιδιού
  - 1ο βήμα: κατασκευάζουμε το σωρό
  - 2ο βήμα: αφαιρούμε τα στοιχεία με τη σειρά
  - Απαιτεί χρήση δεύτερου πίνακα (όπως η mergesort)
  - Τρέχει σε O(NlogN)



```
class Sort {
  static void sort(ITEM[] a, int p, int r) {
       int k; PQ pq = new PQ(r-p+1);
       for (k = p; k <= r; k++) pq.insert(a[k])
       for (k = r; k >= p; k--) a[k] = pq.getmax(); }
}
```

Ο αλγόριθμος heapsort

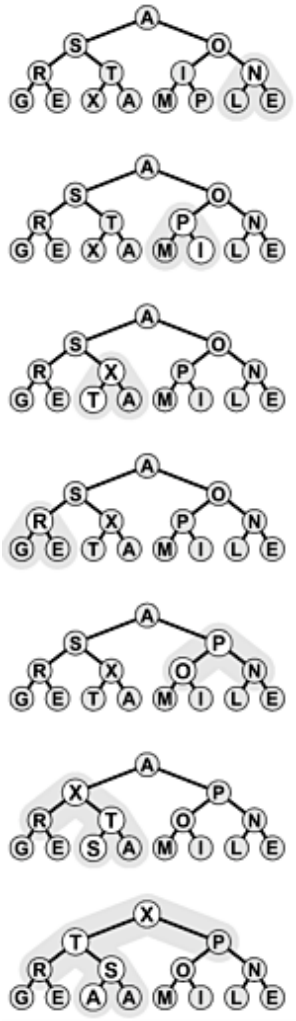
- Ταξινόμηση με σωρό χωρίς πρόσθετο πίνακα
  - Κατασκευή του σωρού στον ίδιο τον πίνακα a
    - Θυσιάζουμε τη γενική υλοποίηση για λόγους απόδοσης
  - Δημιουργία υποσωρών από κάτω προς τα πάνω
  - Θεωρούμε τον a ως αναπαράσταση πλήρους δυαδικού δέντρου
    - Αρχικά όλα τα φύλλα είναι σωροί ενός στοιχείου
    - Αποκαθιστούμε την ιδιότητα του σωρού σε κάθε υποδέντρο καλώντας τη sink από κάτω προς τα πάνω
    - Όταν φτάσουμε στη ρίζα έχει μείνει ένας μόνο σωρός

Ο αλγόριθμος heapsort

- Υλοποίηση heapsort
  - 1ο βήμα: κατασκευή του σωρού από κάτω προς τα πάνω στον πίνακα a
  - 2ο βήμα: εξαγωγή των στοιχείων και τοποθέτηση στο τέλος
  - Υποθέτουμε τα στοιχεία είναι στις θέσεις 1 ως N, εύκολα προσαρμόζεται αν ήταν από p ως r

```
for (int k = N/2; k >= 1; k--)
  sink(k, N); //μετατροπή του a σε σωρό
while (N > 1) {
    exch(1, N);
    sink(1, --N); }
```

  - Οι `sink` και `exch` εκτελούνται πάνω στον πίνακα `a` και όχι σε ΑΤΔ ουράς προτεραιότητας



Ο αλγόριθμος heapsort

- Ανάλυση κόστους
- 1ο βήμα: O(N)
  - Έστω ότι έχουμε ένα γεμάτο δέντρο (παρόμοια ανάλυση για άλλα δέντρα)
  - Π.χ. N = 127 ή πιο γενικά N = 2^n - 1 για κάποιο n
  - Θα έχουμε 64 φύλλα
  - 32 (= 2^{n-2}) σωρούς στο προτελευταίο επίπεδο
  - 16 (=2^{n-3}) σωρούς στο προπροτελευταίο επίπεδο
  - ...
  - Σε κάθε υποσωρό το κόστος της sink είναι όσο και το ύψος του υποσωρού
  - Συνολικό κόστος κατασκευής: $\sum_{k=1}^{n} k2^{n-k-1} = 2^n - n - 1 < N$

Ο αλγόριθμος heapsort

- 2ο βήμα:
  - Κάθε εξαγωγή απαιτεί χρόνο O(logN), άρα συνολικά O(NlogN)
- Συνολικά: O(N) + O(NlogN) = O(NlogN)

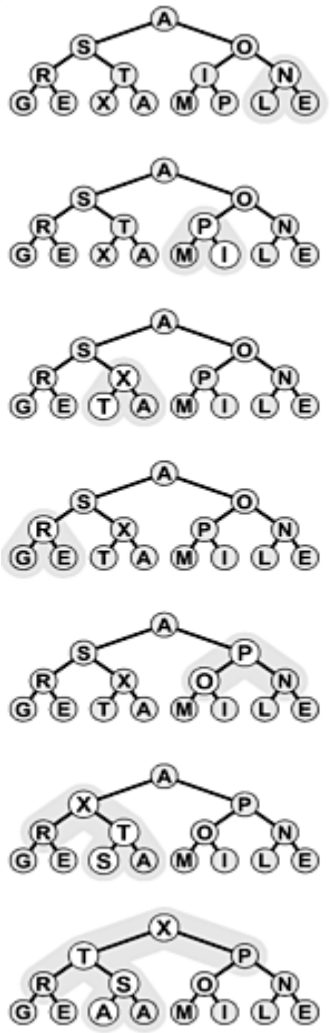
- Στην πράξη η Quicksort παραμένει η πιο γρήγορη μέθοδος
- Η heapsort τρέχει σε χρόνο παρόμοιο με τη mergesort
- Δεν χρειάζεται όμως πρόσθετο χώρο (αντίθετα με τη mergesort)

Ο αλγόριθμος heapsort

- Παράδειγμα ταξινόμησης με σωρό
  - Δημιουργία σωρού (σε μορφή δέντρου)
    - Τα στοιχεία ταξινομούνται μερικώς
  - Ταξινόμηση με βάση το σωρό (σε μορφή πίνακα)
    - Ουσιαστικά ταξινόμηση με επιλογή
    - Επιταχύνεται όμως η επιλογή!

A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S	O	R	T	P	N	G	E	X	A	M	I	L	E
A	S	O	R	X	P	N	G	E	T	A	M	I	L	E
A	S	O	R	X	P	N	G	E	T	A	M	I	L	E
A	S	P	R	X	O	N	G	E	T	A	M	I	L	E
A	X	P	R	T	O	N	G	E	S	A	M	I	L	E
X	T	P	R	S	O	N	G	E	A	A	M	I	L	E

T	S	P	R	E	O	N	G	E	A	A	M	I	L	X
S	R	P	L	E	O	N	G	E	A	A	M	I	T	X
R	L	P	I	E	O	N	G	E	A	A	M	S	T	X
P	L	O	I	E	M	N	G	E	A	A	R	S	T	X
O	L	N	I	E	M	A	G	E	A	P	R	S	T	X
N	L	M	I	E	A	A	G	E	O	P	R	S	T	X
M	L	E	I	E	A	A	G	N	O	P	R	S	T	X
L	I	E	G	E	A	A	M	N	O	P	R	S	T	X
I	G	E	A	E	A	L	M	N	O	P	R	S	T	X
G	E	E	A	A	I	L	M	N	O	P	R	S	T	X
E	A	E	A	G	I	L	M	N	O	P	R	S	T	X
E	A	A	E	G	I	L	M	N	O	P	R	S	T	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X



Ο αλγόριθμος heapsort

- Χρήση σωρού για επιλογή
  - Εντοπισμός των k μεγαλύτερων από τα N στοιχεία
  - Η ταξινόμηση λύνει το πρόβλημα σε χρόνο NlogN
  - Ο σωρός λύνει το πρόβλημα σε γραμμικό χρόνο για μικρό k
- 1η μέθοδος (προφανής)
  - Κατασκευάζουμε το σωρό σε χρόνο O(N)
  - Αφαιρούμε k στοιχεία σε χρόνο 2klogN
- 2η μέθοδος (όχι τόσο προφανής)
  - Κατασκευάζουμε έναν ανάποδο σωρό με k στοιχεία
    - Η εξαγωγή αφαιρεί το μικρότερο στοιχείο
  - Κάνουμε N-k εισαγωγές ακολουθούμενες από εξαγωγές του ελάχιστου
    - Κάθε φορά βγάζουμε το μικρότερο από k+1 στοιχεία
  - Συνολικό κόστος 2k + 2(N-k)logk