Αλγόριθμοι Ταξινόμησης – Μέρος 3

Μανόλης Κουμπαράκης

Ταξινόμηση με Ουρά Προτεραιότητας

- Θα παρουσιάσουμε τώρα δύο αλγόριθμους ταξινόμησης που χρησιμοποιούν μια **ουρά προτεραιότητας** για την υλοποίηση τους.
- Η δομή της ουράς προτεραιότητας και οι υλοποιήσεις της έχουν μελετηθεί διεξοδικά σε προηγούμενες διαλέξεις.

Δηλώσεις σε C

```
#define n 100  /* n gives the number of items */
               /* in the array to be sorted */

typedef int KeyType;   /* we assume that the type */
                       /* of keys is int */
typedef KeyType SortingArray[n];

SortingArray A;
```

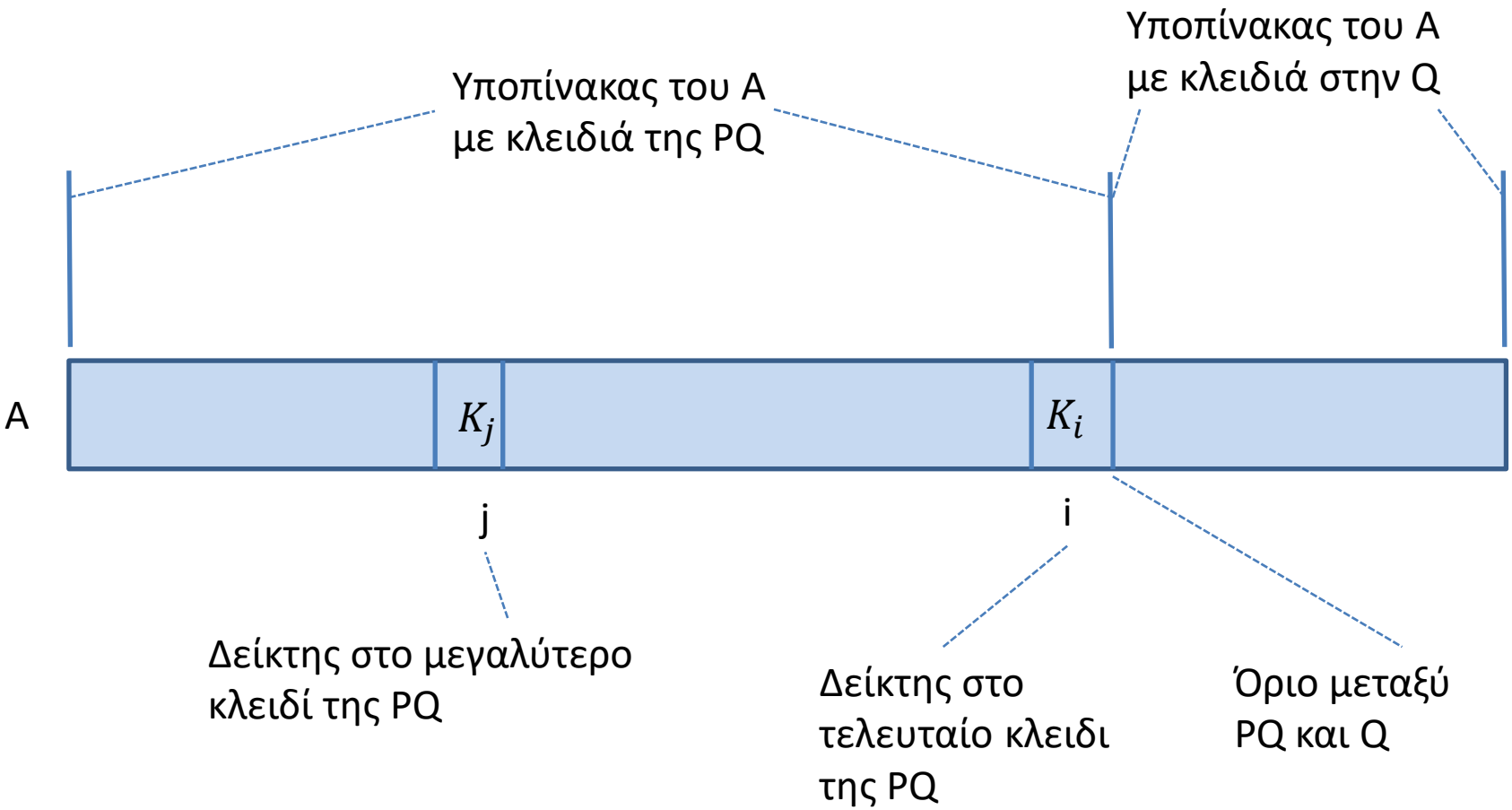
Ο Γενικός Αλγόριθμος Ταξινόμησης με Ουρά Προτεραιότητας

```
void PriorityQueueSort(SortingArray A)
{
   Έστω Q μία αρχικά κενή ουρά αναμονής.

   Keytype K;

   Οργάνωσε τα κλειδιά του A σε μια ουρά προτεραιότητας PQ.
   while (PQ δεν είναι κενή){
     Βγάλε το μεγαλύτερο κλειδί K από την PQ.
     Βάλε το κλειδί K στο τέλος της ουράς Q.
   }

   Μετακίνησε τα κλειδιά της Q στον πίνακα A σε αύξουσα
   σειρά.
}
```

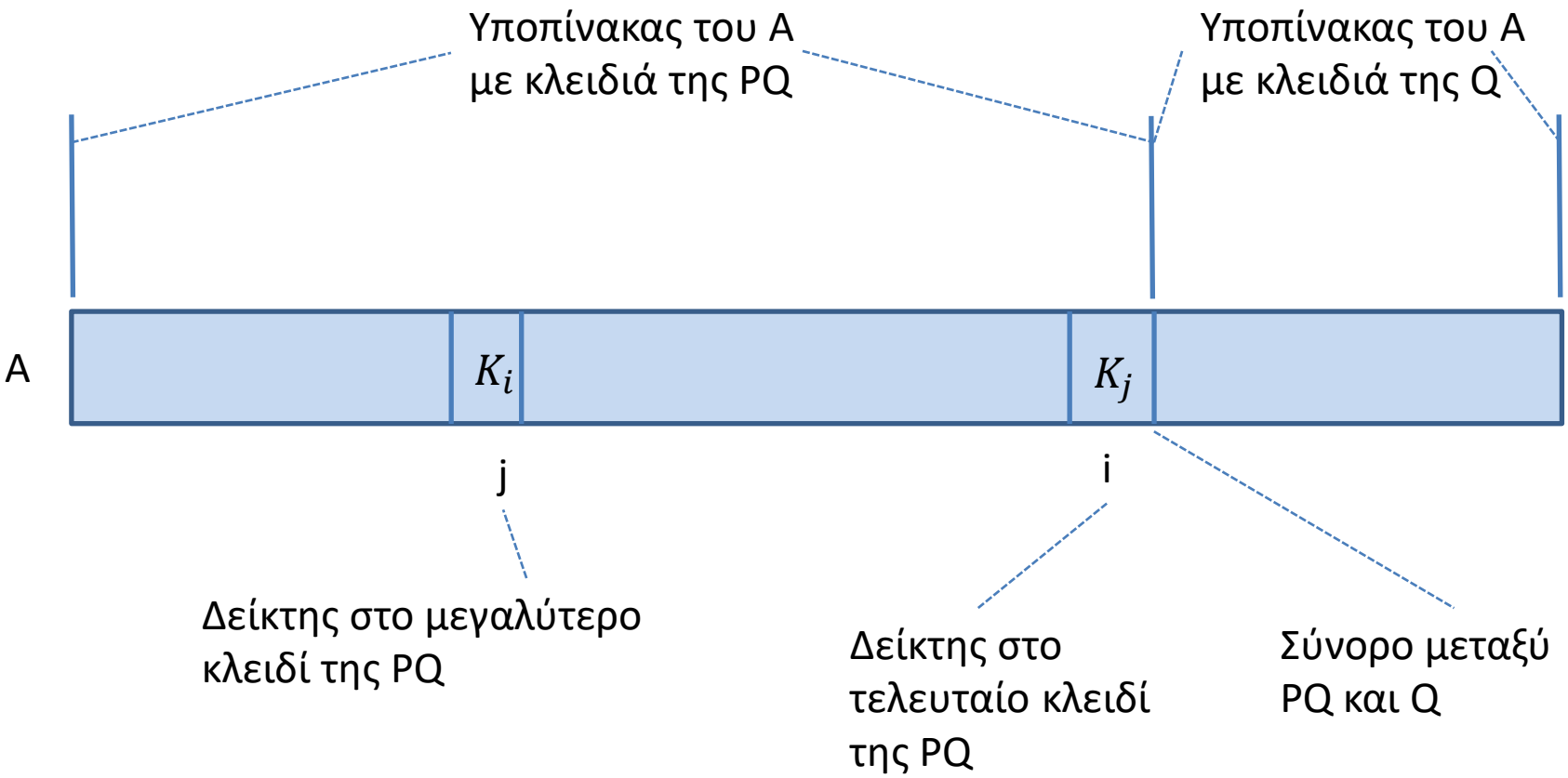
Ταξινόμηση με Ουρά Προτεραιότητας

- Θα παρουσιάσουμε τώρα δύο συγκεκριμένες υλοποιήσεις του γενικού αλγόριθμου, χρησιμοποιώντας τις δύο αναπαραστάσεις μιας ουράς προτεραιότητας που έχουμε μελετήσει:
  - Αναπαράσταση με αταξινόμητο πίνακα
  - Αναπαράσταση με σωρό (heap)
- Και οι δύο αναπαραστάσεις θα υλοποιηθούν **επιτόπου** στον πίνακα `A`.

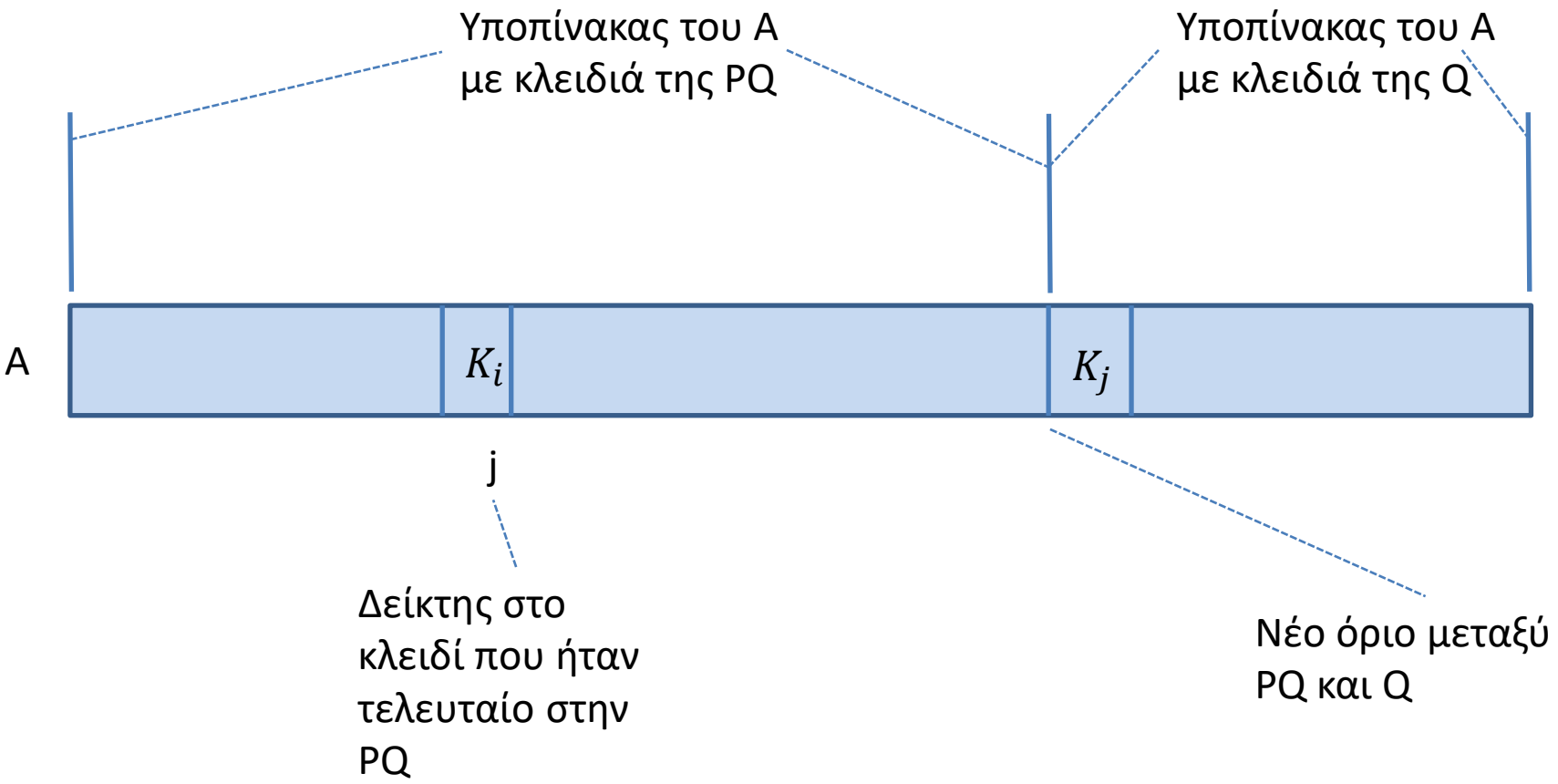
Η Γενική Μέθοδος



Αντιμετάθεση του `A[i]` και του `A[j]`



Μετακίνηση του Συνόρου Μεταξύ PQ και Q



Ο Αλγόριθμος `SelectionSort` (Ξανά!)

- Εάν υλοποιήσουμε τον γενικό αλγόριθμο ταξινόμησης με ουρά προτεραιότητας χρησιμοποιώντας την υλοποίηση της ουράς προτεραιότητας με αταξινόμητο πίνακα, τότε παίρνουμε το αλγόριθμο `SelectionSort` που έχουμε ήδη παρουσιάσει.

SelectionSort (cont'd)

```
void SelectionSort(SortingArray A)
{
   int i,j,k;
   KeyType Temp;


   /* Initially, Q is empty and PQ contains all keys in A, so the index, i, */
   /* of the last key in PQ is set to n-1, the index of the last key in A. */
   i=n-1;

   /* While PQ contains more than one key, */
   /* identify and move the largest key in PQ into Q */
   while (i>0){
      /* Let j initially point to the last key in PQ */
      j=i;

      /* Scan remaining positions in 0:i-1 to find largest key, A[j] */
      for (k=0; k<i; k++){
         if (A[k]>A[j]) j=k;
      }

      /* Swap the largest key, A[j], and the last key, A[i] */
      Temp=A[i]; A[i]=A[j]; A[j]=Temp;

      /* Move boundary between PQ and Q downward one position */
      i--;
   }
}
```

Σχόλια στον `SelectionSort`

- Σε κάθε βήμα, για να βρούμε τη θέση του μεγαλύτερου κλειδιού `j` στον υποπίνακα `PQ`, διασχίζουμε τον `PQ` ενθυμούμενοι την θέση `j` του μεγαλύτερου κλειδιού μέχρι εκείνη τη στιγμή.
- Μετά, αντιμεταθέτουμε το το μεγαλύτερο κλειδί `A[j]` της `PQ` με το τελευταίο κλειδί `A[i]` της `PQ` και μειώνουμε το `i` κατά 1 ώστε να μετακινήσουμε το σύνορο μεταξύ `PQ` και `Q`.
- Δεν χρειάζεται να κάνουμε κάτι για να οργανώσουμε τα κλειδιά του αταξινόμητου πίνακα `A` σε μια ουρά προτεραιότητας `PQ`, επειδή ο `A` χρησιμοποιείται κατευθείαν ως αναπαράσταση της `PQ`.
- Δεν χρειάζεται να κάνουμε κάτι για να μετακινήσουμε τα κλειδιά από την ουρά `Q` στον πίνακα `A` σε αύξουσα σειρά, επειδή η `Q` μεταβάλλεται από ένα άδειο υποπίνακα του `A` που είναι αρχικά, σε ολόκληρο τον `A` όταν ο αλγόριθμος τερματίζει.

Ιδιότητες του `SelectionSort`

- Επειδή ο αλγόριθμος `SelectionSort` αντιμεταθέτει το `A[i]` με το `A[j]` για κάθε `i` από το `n-1` μέχρι το `1`, εκτελεί $n-1$ αντιμεταθέσεις.
- Ο αριθμός των συγκρίσεων που κάνει ο αλγόριθμος `SelectionSort` δίνεται από το άθροισμα

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}.$$

- Άρα η χρονική πολυπλοκότητα του `SelectionSort` είναι $O(n^2)$.

HeapSort

- Θα παρουσιάσουμε τώρα μια δεύτερη εξειδίκευση του γενικού αλγόριθμου ταξινόμησης με ουρά προτεραιότητας.
- Θα παραστήσουμε την ουρά προτεραιότητας με ένα **σωρό μεγίστων** (**max heap**).
- Θα χρησιμοποιήσουμε την ακολουθιακή αναπαράσταση ενός σωρού ως υποπίνακα `A[1:n]` ενός πίνακα `A[0:n]` που ορίζεται στη C με την παρακάτω δήλωση:

```
KeyType A[n+1];
```

HeapSort

- Όταν ξεκινάει ο αλγόριθμος, η ουρά προτεραιότητας `PQ` καταλαμβάνει τον πίνακα `A[1:n]`. Οργανώνουμε τα στοιχεία της `PQ` σε σωρό χρησιμοποιώντας τον αλγόριθμο που έχουμε παρουσιάσει σε προηγούμενες διαλέξεις.
- Στο τέλος της διαδικασίας μετατροπής του πίνακα `A[1:n]` σε σωρό, το μεγαλύτερο στοιχείο του σωρού βρίσκεται στη θέση `A[1]`.
- Τώρα αντιμεταθέτουμε το πρώτο και το τελευταίο κλειδί της `PQ` δηλαδή το `A[1]` και το `A[n]`.
- Μετά, μετακινούμε προς τα αριστερά κατά μία θέση το σύνορο της `PQ` με την `Q`, ώστε η `Q` τώρα να περιέχει στη τελευταία της θέση το κλειδί που ήταν το μεγαλύτερο κλειδί της `PQ` νωρίτερα, και η `PQ` να περιέχει ένα κλειδί λιγότερο.

HeapSort

- Αναδιοργανώνουμε τα στοιχεία της PQ σε σωρό γιατί η εισαγωγή του νέου κλειδιού στη ρίζα μπορεί να έκανε την PQ να μην είναι πλέον σωρός.
- Όταν τελειώσει η αναδιοργάνωση, το μεγαλύτερο από τα εναπομείναντα κλειδιά της PQ θα έχει μετακινηθεί στη θέση A[1].
- Μετά, επανειλημμένα αντιμεταθέτουμε το πρώτο και το τελευταίο κλειδί της PQ, μετακινούμε το σύνορο μεταξύ PQ και Q, και αναδιοργανώνουμε την PQ σε σωρό μέχρι η PQ να περιέχει ένα μόνο στοιχείο.
- Σ' αυτό το σημείο, ο πίνακας A είναι ταξινομημένος.

HeapSort

```
void HeapSort(SortingArray A)
{
   int i;
   KeyType Temp;

   /* Heapify in reverse level order all subtrees except the */
   /* subtree containing the root */
   for (i=(n/2); i>1; --i)
       SiftUp(A,i,n);

   /* Reheapify starting at the root, remove root, put it on */
   /* output queue, and replace root with the last leaf in level */
   /* order, until heap contains one key */
   for (i=n; i>1; --i){
      SiftUp(A,1,i);
      Temp=A[1]; A[1]=A[i]; A[i]=Temp;
   }
}
```

HeapSort

```
void SiftUp(SortingArray A, int i, int n)
{
   /* Let i point to the root and let n point to the last leaf in level order */

   int j;
   KeyType RootKey;
   Boolean NotFinished;

   /* Let RootKey be the key at the root */
   RootKey=A[i];

   /* Let j point to the left child of i */
   j=2*i;
   NotFinished=(j<=n); /* SiftUp is not finished if j exists in the tree */

   /* Move any larger child that is bigger than the root key upward one */
   /* level in the tree */
   while (NotFinished){
      if (j<n)   /* if a right child of i also exists in the tree */
        if (A[j+1]>A[j]) j++; /* set j to point to the larger child */
      if (A[j]<=RootKey)      /* if the larger child is not bigger than the root key, */
        NotFinished=false;    /* no more keys sift up */
      else {
        A[i]=A[j];            /* move larger child up one level in the tree */
        i=j;                  /* let i point to the larger child j */
        j=2*i;                /* and let j point to the new left child of i */
        NotFinished=(j<=n);   /* SiftUp is not finished iff j exists in the tree */
      }
   }

   /* Final placement of the root key */
   A[i]=RootKey;
}
```

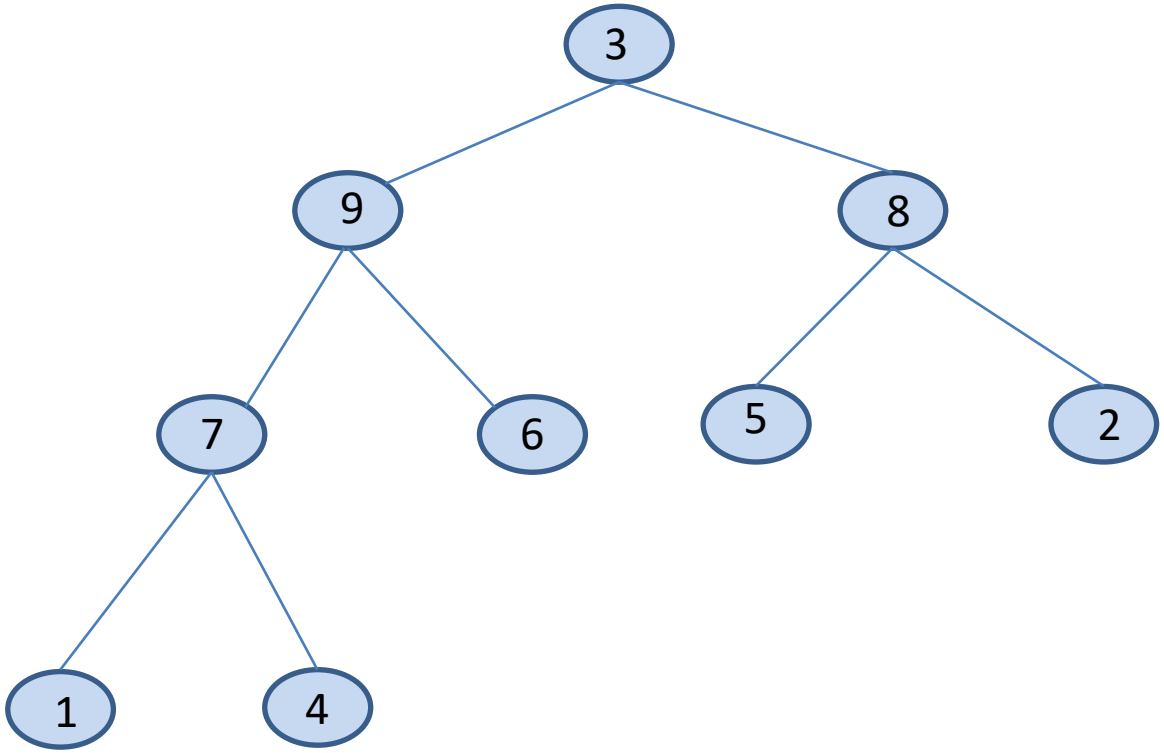
Σχόλια

- Όταν ξεκινάει ο αλγόριθμος `HeapSort`, όλα τα κλειδιά στον υποπίνακα `A[1:n]` οργανώνονται σε σωρό.
- Κατά την αρχική οργάνωση αυτών των κλειδιών σε σωρό στον πρώτο βρόχο `for` του αλγόριθμου, τα υποδέντρα της `PQ` επισκέπτονται κατά την αντίστροφη σειρά επιπέδου (reverse level order).
- Για την ακρίβεια, οργανώνονται σε σωρό μόνο τα μη τετριμμένα υποδέντρα της `PQ` (δηλαδή αυτά που δεν είναι φύλλα).
- Το υποδέντρο που αποτελείται από ολόκληρο το δέντρο δεν οργανώνεται σε σωρό στον πρώτο βρόχο `for` αλλά στο δεύτερο.
- Ο δεύτερος βρόχος `for` επίσης αντιμεταθέτει το πρώτο και το τελευταίο κλειδί της `PQ`, και μετακινεί το σύνορο μεταξύ `PQ` και `Q`.

Σχόλια

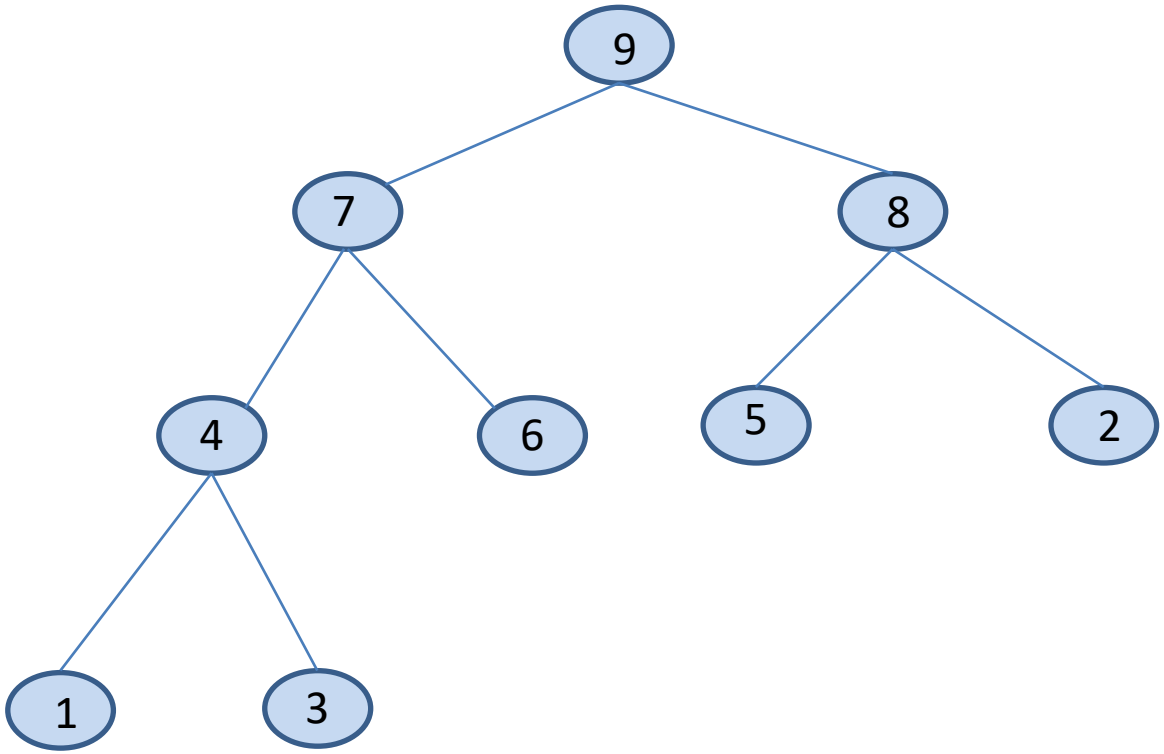
- Η συνάρτηση `SiftUp(A,i,n)` είναι μια βοηθητική ρουτίνα η οποία μετατρέπει ένα δέντρο που είναι σχεδόν σωρός σε σωρό, χρησιμοποιώντας μια **κυκλική μετάθεση (cyclic shift) κλειδιών** ως εξής.
- Ξεκινάμε από τη ρίζα και μετακινούμαστε προς τα κάτω ακολουθώντας ένα μονοπάτι κατά μήκος του οποίου τα κλειδιά μεγαλύτερα από την ρίζα μετακινούνται προς τα πάνω, ώστε να δημιουργηθεί μια άδεια θέση για την τελική μετακίνηση του κλειδιού της ρίζας.
- Τα δέντρα στα οποία εφαρμόζεται η `SiftUp` δεν έχουν την ιδιότητα του σωρού μόνο στη ρίζα τους (ή είναι ήδη σωροί).

Σχόλια



Το παραπάνω δυαδικό δέντρο **δεν** είναι σωρός. Η ιδιότητα του σωρού δεν ισχύει για το κλειδί της ρίζας 3.

Σχόλια



Η συνάρτηση `SiftUp` μετακινεί τα κλειδιά 9, 7 και 4 προς τα πάνω ώστε να δημιουργηθεί μια άδεια θέση για το κλειδί της ρίζας 3.

Ιδιότητες του `HeapSort`

- Ας χρησιμοποιήσουμε ό,τι ξέρουμε ήδη για σωρούς για να υπολογίσουμε την πολυπλοκότητα χρόνου του `HeapSort`.
- Έχουμε αποδείξει ότι η διαδικασία οργάνωσης των κλειδιών του πίνακα `A` σε σωρό παίρνει χρόνο $O(n)$.
- Έχουμε επίσης αποδείξει ότι η διαδικασία διαγραφής της ρίζας ενός σωρού, η αντικατάσταση της με το τελευταίο κλειδί κατά σειρά επιπέδου, και η αναδιοργάνωση του δέντρου που προκύπτει σε σωρό με i κλειδιά χρειάζεται το πολύ $\lfloor \log_2 i \rfloor$ αντιμεταθέσεις κλειδιών.
- Ομοίως, η συνάρτηση `SiftUp` χρειάζεται $\lfloor \log_2 i \rfloor + 2$ μεταθέσεις κλειδιών από ένα κόμβο ή μια προσωρινή μεταβλητή σε ένα άλλο κόμβο ή μεταβλητή.
- Άρα για να αφαιρέσουμε όλα τα κλειδιά εκτός το τελευταίο από την `PQ` και να τα μετακινήσουμε στο τέλος της `Q`, χρειάζεται τουλάχιστον

$$\sum_{i=2}^{n} (\lfloor \log_2 i \rfloor + 2)$$

χρόνο.

Ιδιότητες του `HeapSort`

- Ο συνολικός αριθμός συγκρίσεων κλειδιών είναι ένα παρόμοιο άθροισμα φραγμένο από πάνω από την παράσταση $\sum_{i=2}^{n}(2\lfloor \log_2 i \rfloor + 1)$.
- Γνωρίζουμε ότι

$$\sum_{i=1}^{n} \lfloor \log_2 i \rfloor = (n+1)q - 2^{(q+1)} + 2$$

όπου $q = \lfloor \log_2(n+1) \rfloor$.

- Με λίγες πράξεις ανισοτήτων μπορεί να δειχτεί ότι το παραπάνω άθροισμα είναι $O(n \log n)$.
- Μια και αυτός είναι ο υπερισχύων χρόνος, ο αλγόριθμος `HeapSort` τρέχει σε χρόνο $O(n \log n)$.

Μελέτη

- Thomas A. Standish. *Data Structures, Algorithms and Software Principles in C*. Addison-Wesley.
  - Section 13.3
- R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος.
  - Κεφ. 9