Ενότητα 7
Ουρές Προτεραιότητας

Ουρές Προτεραιότητας

Θεωρούμε ένα χώρο κλειδιών U και έστω ότι με κάθε κλειδί K (τύπου Key) έχει συσχετισθεί κάποια πληροφορία I (τύπου Type). Έστω επίσης ότι έχει ορισθεί μια διάταξη στα στοιχεία του U έτσι ώστε για κάθε ζεύγος στοιχείων $x,y \in U$, να ισχύει είτε $x = y$ ή $x < y$ ή $x > y$.

Τα στοιχεία που είναι επιθυμητό να αποθηκευθούν στη δομή είναι ζεύγη της μορφής <K,I>.

Ορισμός
Μια **ουρά προτεραιότητας** είναι ένας αφηρημένος τύπος δεδομένων για ένα σύνολο με στοιχεία ζεύγη <K,I>, που υποστηρίζει τις ακόλουθες λειτουργίες:

- *MakeEmptySet():* επιστρέφει το κενό σύνολο ∅.
- *IsEmptySet(S):* Επιστρέφει true αν S = ∅, false διαφορετικά
- *Insert(K,I,S):* Εισάγει το ζεύγος <K,I> στο S.
- *FindMin(S):* Επιστρέφει το πεδίο I του ζεύγους <K,I>, όπου K είναι το μικρότερο κλειδί στο σύνολο.
- *DeleteMin(S):* Διαγράφει το ζεύγος <K,I>, όπου K είναι το μικρότερο κλειδί στο σύνολο, και επιστρέφει I.

Ουρές Προτεραιότητας

Υλοποίηση με Ισοζυγισμένα Δένδρα

Μπορούμε να υλοποιήσουμε μια ουρά προτεραιότητας με ισοζυγισμένα δένδρα;

AVL δένδρα, 2-3 δένδρα, κοκκινόμαυρα δένδρα: όλα παρέχουν αποτελεσματικές υλοποιήσεις ουρών προτεραιοτήτων.

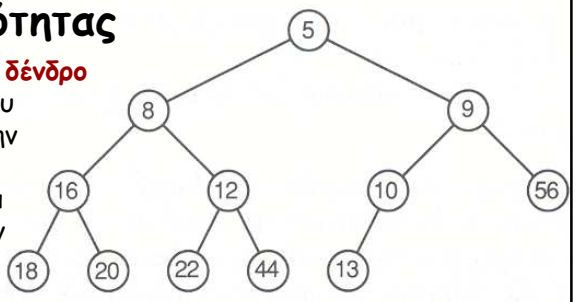
Δεδομένου ενός ισοζυγισμένου δένδρου, πως θα μπορούσαμε να υλοποιήσουμε μια ουρά προτεραιότητας;

Ποια είναι η χρονική πολυπλοκότητα για τις λειτουργίες Insert(), FindMin() και DeleteMin();
Υπάρχουν άλλες λειτουργίες που να υποστηρίζονται αποδοτικά;
(1) LookUp; (2) Delete; (3) FindMax – DeleteMax;

Μια ουρά προτεραιότητας που υποστηρίζει και τις λειτουργίες FindMax() και DeleteMax() ονομάζεται **διπλή ουρά προτεραιότητας** (ή **ουρά προτεραιότητας με δύο άκρα**).

Ουρές Προτεραιότητας

Ένα **μερικώς διατεταγμένο δένδρο** είναι ένα δυαδικό δένδρο του οποίου τα στοιχεία έχουν την εξής ιδιότητα:
Το κλειδί κάθε κόμβου είναι μικρότερο ή ίσο εκείνου των παιδιών του κόμβου.



Θεωρούμε ότι η προτεραιότητα ενός κόμβου καθορίζεται από το κλειδί του ως εξής: μικρό κλειδί -> μεγάλη προτεραιότητα και το αντίστροφο. Έτσι, η διάταξη των κόμβων του σχήματος από μεγαλύτερες προς μικρότερες προτεραιότητες είναι 5, 8, 9, 10, 12, 13, 16, 18, 20, 22, 44, 56.

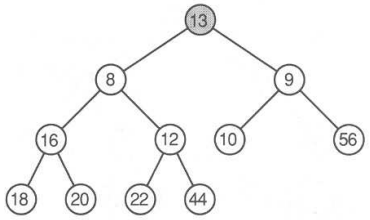
Ποιος είναι ο κόμβος με τη μεγαλύτερη προτεραιότητα σε ένα μερικώς διατεταγμένο δένδρο;

- Σε κάθε μονοπάτι από τη ρίζα προς οποιοδήποτε κόμβο, οι κόμβοι που διατρέχουμε είναι φθίνουσας προτεραιότητας.

Πως υλοποιούμε την FindMin() σε μερικώς διατεταγμένο δένδρο;

Ουρές Προτεραιότητας

Προκειμένου οι λειτουργίες που υποστηρίζονται από μια ουρά προτεραιότητας να υλοποιηθούν αποδοτικά, θα ήταν επιθυμητό το ύψος του δένδρου να είναι $O(\log n)$.

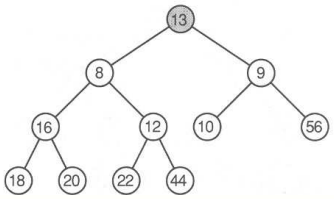


DeleteMin()
Δεν διαγράφουμε τη ρίζα, αλλά το δεξιότερο φύλλο του τελευταίου επιπέδου του δένδρου, αφού πρώτα αντιγράψουμε τα δεδομένα του στη ρίζα.

Πρόβλημα που μπορεί να προκύψει
Το προκύπτον δένδρο μπορεί να μην είναι μερικώς διατεταγμένο δένδρο.

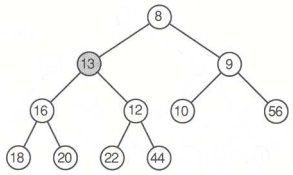
Παρατήρηση
Η μερική διάταξη καταστρέφεται μόνο στη ρίζα.

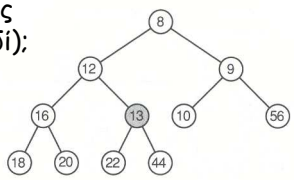
Μερικώς Διατεταγμένα Δένδρα Διαγραφή



DeleteMin() - Επανόρθωση διάταξης

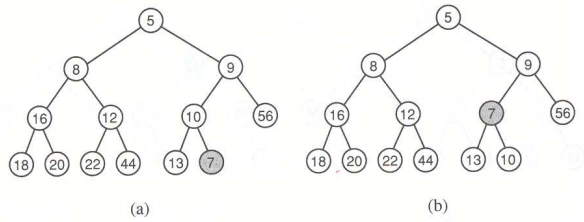
1. Έστω u η ρίζα του δένδρου και w ο θυγατρικός κόμβος του u με το μικρότερο κλειδί.
2. Αν (w == nill OR κλειδί του u < κλειδί του w) ο αλγόριθμος τερματίζει;
3. Ανταλλάσσουμε τα δεδομένα <Κ,Ι> του u με τα δεδομένα <Κ',Ι'> του w;
4. Θέτουμε (u = w) και (w = θυγατρικός κόμβος του u με το μικρότερο κλειδί);
5. Επιστρέφουμε στο βήμα 2;





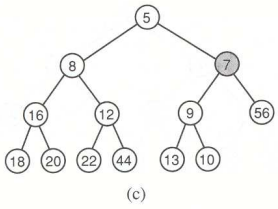
Ποια είναι πολυπλοκότητα της DeleteMin(); O(ύψος δένδρου)

Μερικώς Διατεταγμένα Δένδρα



Insert()

- Εισάγουμε το νέο στοιχείο ως δεξιότερο φύλλο του δένδρου.
- Η ιδιότητα της μερικής διάταξης ίσως καταστρέφεται άλλα μόνο για τον πατέρα του φύλλου αυτού.



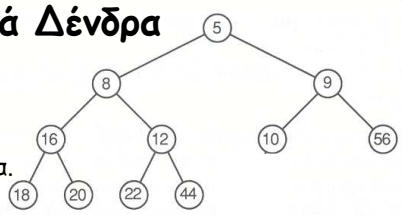
Insert() - Επανόρθωση διάταξης

1. Έστω u ο κόμβος φύλλο στον οποίο γίνεται η εισαγωγή και w ο πατρικός κόμβος του u.
2. Αν (w == nill OR κλειδί του w < κλειδί του u) ο αλγόριθμος τερματίζει;
3. Ανταλλάσσουμε τα δεδομένα <K,I> του u με τα δεδομένα (<K',I'>) του w;
4. Θέτω (u = w) και (w = πατρικός κόμβος του u);
5. Επιστρέφουμε στο βήμα 2;

Ποια είναι η χρονική πολυπλοκότητα της Insert();
O(ύψος δένδρου)

Μερικώς Διατεταγμένα Δένδρα
Υλοποίηση ως Πλήρη Δυαδικά Δένδρα

- Υλοποιούμε το μερικώς διατεταγμένο δένδρο χρησιμοποιώντας πλήρη δυαδικά δένδρα.



- Το δεξιότερο φύλλο με μέγιστο βάθος στο δένδρο είναι το τελευταίο στοιχείο του πίνακα.
- Η θέση του πίνακα που περιέχει αυτό τον κόμβο μπορεί να καθοριστεί αν γνωρίζουμε το πλήθος των στοιχείων και τη διεύθυνση του πρώτου στοιχείου του πίνακα.
- Η διαγραφή του φύλλου αυτού δεν επηρεάζει την μερική διάταξη του δένδρου, ενώ το δένδρο εξακολουθεί να είναι πλήρες.

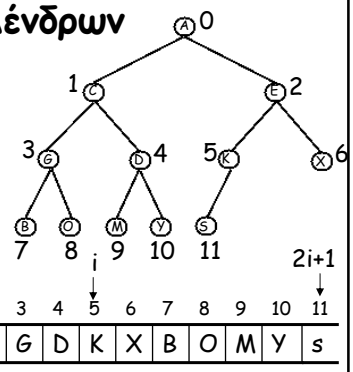
0	1	2	3	4	5	6	7	8	9	10	11
5	8	9	16	12	10	56	18	20	22	44	

Ένα μερικώς διατεταγμένο δένδρο υλοποιημένο με στατικό τρόπο (δηλαδή με πίνακα), ονομάζεται **σωρός**.

Το ύψος οποιουδήποτε πλήρους δυαδικού δένδρου είναι $O(\log n) \Rightarrow$ Πολυπλοκότητα HeapInsert() και HeapDeleteMin() = $O(\log n)$. Ο σωρός είναι μια εξαιρετικά αποδοτική δομή για την υλοποίηση ουρών προτεραιότητας.

Υλοποίηση Πλήρων Δυαδικών Δένδρων

Υπάρχει μόνο ένα πλήρες δυαδικό δένδρο με n κόμβους και το υλοποιούμε με ένα πίνακα Ν στοιχείων.
Αριθμούμε τους κόμβους με αριθμούς στο διάστημα {0,...,n-1} και αποθηκεύουμε τον κόμβο i στο στοιχείο Τ[i] του πίνακα.
Θέλουμε να κάνουμε την αρίθμηση με τέτοιο τρόπο ώστε να πετύχουμε την εκτέλεση χρήσιμων λειτουργιών στο δένδρο σε σταθερό χρόνο.



Αρίθμηση

Η ρίζα είναι ο κόμβος 0.
Το αριστερό παιδί του κόμβου i αριθμείται ως κόμβος 2i+1, ενώ το δεξί παιδί του ως κόμβος 2i+2.

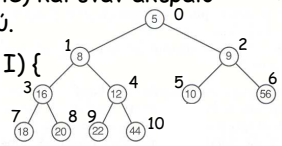
Υλοποίηση Λειτουργιών

- IsLeaf(i): return (2i+1≥n);
- LeftChild(i): if (2i+1 < n); return (2i+1) else return nil;
- RightChild(i): if (2i+2<n) return(2i+2); else return nil;
- LeftSibling(i): if (i != 0 and i not odd) return (i-1);
- RightSibling(i): if (i != n-1 and i not even) return(i+1);
- Parent(i): if (i != 0) return(⌊(i-1)/2⌋);

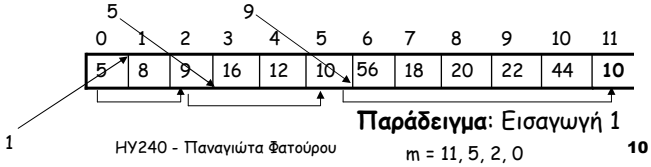
Χρονική πολυπλοκότητα κάθε λειτουργίας: Θ(1)

Σωροί – Ψευδοκώδικας για HeapInsert()

- Τα στοιχεία του σωρού είναι structs με δύο πεδία, ένα κλειδί Κ τύπου Key και τα δεδομένα data τύπου Type που έχουν συσχετισθεί με το κλειδί.
- Ο σωρός υλοποιείται από έναν πίνακα Α (τύπου HeapTable) και έναν ακέραιο size που υποδηλώνει το πλήθος των στοιχείων του σωρού.



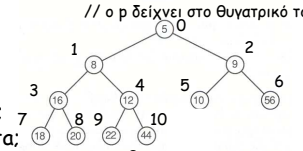
```
procedure HeapInsert(HeapTable A, int size, Key K, Type I) {
/* Εισαγωγή του ζεύγους <K,I> στο σωρό <A,size> */
   if (size == N) then error;   /* Heap is full */
   m = size;   // ο m είναι ακέραιος που χρησιμοποιείται ως δείκτης στους κόμβους ενός
               // μονοπατιού του δένδρου. Αρχικά, δείχνει στη θέση που θα εισαχθεί το νέο στοιχείο
   while (m > 0 and K < A[⌊(m-1)/2⌋]->Key) {   // όσο δεν έχω φθάσει στη ρίζα και
                                               // ο πατρικός κόμβος του m έχει μικρότερο κλειδί από το K
      A[m]->key  = A[⌊(m-1)/2⌋]->Key;   // το κλειδί και τα δεδομένα του πατρικού κόμβου
      A[m]->data = A[⌊(m-1)/2⌋]->data;  // του m αντιγράφονται στον m
      m = ⌊(m-1)/2⌋;                   // η διαδικασία επαναλαμβάνεται με m = πατρικός κόμβος του m
   }
   A[m]->Key = K;
   A[m]->data = I;
   size++;
}
```

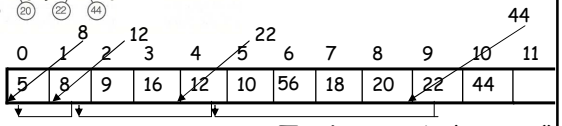


Παράδειγμα: Εισαγωγή 1
m = 11, 5, 2, 0

Σωροί – Ψευδοκώδικας για HeapDeleteMin()

```
Type HeapDeleteMin(HeapTable A, int size) {
// διαγραφή του στοιχείου με τη μεγαλύτερη προτεραιότητα και επιστροφή της τιμής του
   if (size == 0) then error;            // Κενός σωρός
   I = A[0]->data;                       // Στοιχείο που θα επιστραφεί
   K = A[size-1]->Key;                   // τιμή κλειδιού που θα μετακινηθεί προς τη ρίζα
   size--;                               // νέο μέγεθος σωρού
   if (size == 1) then return;           // ο σωρός περιέχει μόνο τη ρίζα μετά τη διαγραφή
   m = 0;                    /* ο m είναι ακέραιος που δεικτοδοτεί τους κόμβους ενός μονοπατιού
    while ((2m+1 < size AND K > A[2m+1]->Key) OR  // αν υπάρχει αριστερό παιδί και έχει μικρότερο κλειδί
           (2m+2 < size  AND K > A[2m+2]->Key)) {  // ή αν υπάρχει δεξιό παιδί και έχει μικρότερο κλειδί
         if (2m +2 < size) {                      // ο m έχει δύο θυγατρικούς κόμβους
            if (A[2m+1]->Key < A[2m+2]->Key)
                  p = 2m+1;                 // ο p δείχνει στο θυγατρικό του m με το μεγαλύτερο κλειδί
             else p = 2m+2;
         }
         else p = size-1;
         A[m]->Key = A[p]->Key;
         A[m]->data = A[p]->data;
         m = p;
   }
   A[m]->Key = K;
   A[m]->data = A[size]->data;
   return I;
}
```





Παράδειγμα: DeleteMin()

m = 0, 1, 4, 9