Κεφάλαιο 4

Δομές Δεδομένων

4.1 Εισαγωγή

Όπως είναι γνωστό, για να εκτελεστεί ένας αλγόριθμος με Η/Υ θα πρέπει να γραφτεί σε μία αυστηρά ορισμένη γλώσσα Η/Υ. Αυτή η υλοποίηση του αλγόριθμου σε γλώσσα προγραμματισμού Η/Υ, λέγεται **πρόγραμμα**. Ένα πρόγραμμα Η/Υ επενεργεί σε σύνολα δεδομένων που αποθηκεύονται στη μνήμη του Η/Υ. Είναι λοιπόν φανερό πως η επιλογή των δομών με τις οποίες θα οργανωθούν τα δεδομένα στη μνήμη του Η/Υ επηρεάζουν την απόδοση του προγράμματος, άρα την απόδοση γενικότερα, του αλγορίθμου. **Δομές δεδομένων**, ονομάζουμε τις διάφορες διμελείς σχέσεις μεταξύ των στοιχείων ενός συνόλου δεδομένων. Οι σχέσεις αυτές μπορεί να είναι γραμμικές ή μη γραμμικές.

Έστω ένα μη κενό σύνολο δεδομένων και μια διμελής σχέση που διατάσσει τα στοιχεία του έτσι ώστε ένα στοιχείο που ονομάζεται αρχή να έχει ένα επόμενο, ένα στοιχείο που ονομάζεται τέλος ένα προηγούμενο και κάθε άλλο στοιχείο να έχει ένα μόνο προηγούμενο και ένα μόνο επόμενο. Τότε λέμε ότι τα στοιχεία του συνόλου αυτού των δεδομένων, είναι **ολικώς ή γραμμικώς διατεταγμένα** (*totally or linearly ordered*) και η δομή που ορίζεται απ' αυτή τη σχέση ονομάζεται γραμμική δομή δεδομένων (*linear data structure*). Κάθε άλλη δομή δεδομένων που δεν είναι γραμμική, ονομάζεται **μη γραμμική δομή δεδομένων** (*non-linear*). Σημειωτέον ότι η διάταξη αυτή δεν έχει σχέση με τυχόν άλλη διάταξη των τιμών των κόμβων π.χ. λεξικογραφική ή αριθμητική.

Στις γραμμικές δομές δεδομένων, ανήκουν οι πίνακες, οι εγγραφές, τα σύνολα, τα αρχεία και οι γραμμικές λίστες. Από τις γραμμικές λίστες μπορούμε να ξεχωρίσουμε τις **στοίβες** (*stacks*) και τις **ουρές** (*queues*). Στις μη γραμμικές δομές δεδομένων ανήκουν οι **γράφοι** (ή **γραφήματα**) και τα δέντρα.

Στο κεφάλαιο αυτό θα συζητήσουμε τις πιο γνωστές δομές δεδομένων, καθώς και τους αλγορίθμους υλοποίησης των βασικών τους πράξεων. Το υλικό που παρουσιάζεται έχει εν μέρει στηριχθεί στα διδακτικά συγγράμματα [5, 2, 3, 4], όπου και μπορούν να αναζητηθούν περισσότερες λεπτομέρειες για τα θέματα που αναπτύσσονται εδώ.

4.2 Σωροί-Ουρά Προτεραιότητας-Heapsort

Πριν προχωρήσουμε στις ουρές προτεραιότητας, θα αναφέρουμε συνοπτικά τι είναι οι λεγόμενες αφηρημένες δομές δεδομένων (ADT: abstract data types). Οι δομές αυτές αποτελούν ουσιαστικά ένα μοντέλο που περιγράφει ένα σύνολο συγκεκριμένων δομών δεδομένων που έχουν παρόμοια συμπεριφορά. Με άλλα λόγια, δεν ορίζουμε με σαφή τρόπο τη δομή μας, αλλά ορίζουμε μόνο τις πράξεις που θέλουμε να εκτελούμε στα στοιχεία μας, και τι ιδιότητες πρέπει να έχουν αυτές οι πράξεις.

Ορίζουμε λοιπόν ένα σύνολο λειτουργιών (μεθόδους) επί των στοιχείων μας που χαρακτηρίζει την ADT, αλλά ο ακριβής τρόπος με τον οποίο θα αναπαραστήσουμε τα δεδομένα μας και θα υλοποιήσουμε τις πράξεις μας μπορεί να διαφέρει.

Η υλοποίηση μιας ADT από μια (συγκεκριμένη) δομή δεδομένων αποτελείται από

- Αναπαράσταση: οργάνωση στιγμιοτύπων και υλοποίηση λειτουργιών με κατάλληλους αλγόριθμους.
- Διατύπωση: ορισμός αναπαράστασης και περιγραφή υλοποίησης λειτουργιών (ψευδοκώδικας).
- Ανάλυση: προσδιορισμός απαιτήσεων σε χώρο αποθήκευσης και χρόνο εκτέλεσης για κάθε (βασική) λειτουργία.

Είμαστε πλέον έτοιμοι να προχωρήσουμε στον ορισμό μιας ADT που ονομάζεται ουρά προτεραιότητας (priority queue).

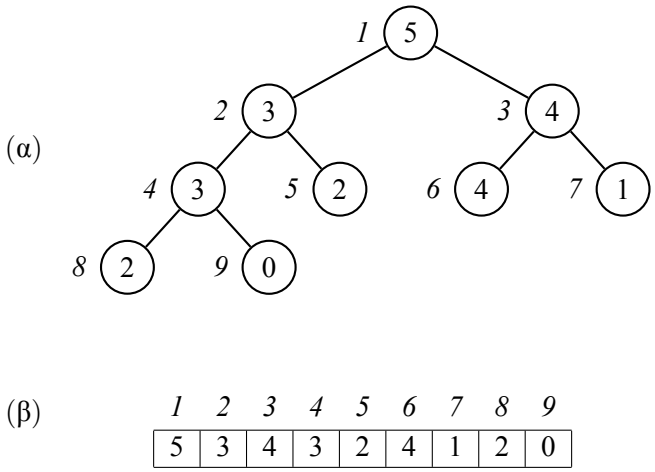
Ορισμός 4.1. Έστω ότι μας δίνονται κάποια στοιχεία για τα οποία ισχύει μια ολική σχέση διάταξης ($<$). Μια δομή δεδομένων η οποία μας επιτρέπει να κάνουμε με αποδοτικό τρόπο εισαγωγή ενός καινούργιου στοιχείου και εξαγωγή (δηλαδή εύρεση και διαγραφή) του μεγαλύτερου στοιχείου, λέγεται **ουρά με προτεραιότητα** (*priority queue*).

Με άλλα λόγια, η ουρά προτεραιότητας είναι μια ουρά όπου η σειρά διαγραφής καθορίζεται από μια προτεραιότητα (μεγαλύτερη – μικρότερη). Το κάθε στοιχείο λοιπόν έχει μια προτεραιότητα. Οι λειτουργίες που μας ενδιαφέρουν είναι:

- insert(x): εισαγωγή x
- deleteMax(): διαγραφή και επιστροφή στοιχείου μέγιστης προτεραιότητας
- max(): επιστροφή στοιχείου μέγιστης προτεραιότητας (χωρίς διαγραφή)
- changePriority(k): αλλαγή προτεραιότητας θέσης k
- isEmpty(), size(): βοηθητικές λειτουργίες.

Ένας τρόπος για να παραστήσουμε τις ουρές με προτεραιότητα είναι η υλοποίηση με σωρό.

Ορισμός 4.2. Σωρός (*heap tree*) είναι ένα πλήρες δυαδικό δέντρο στο οποίο η τιμή της ρίζας είναι μεγαλύτερη ή ίση (ή μικρότερη ή ίση) από τις τιμές των υπολοίπων κόμβων και αυτό ισχύει αναδρομικά για την ρίζα κάθε υποδέντρου (σχήμα 4.1).



Σχήμα 4.1: (α) Σωρός (Heap tree) και (β) ο πίνακας που αντιστοιχεί σε αυτόν

Έστω ότι έχουμε n ακεραίους αποθηκευμένους με τυχαία σειρά σε ένα μονοδιάστατο πίνακα $a[n]$. Θα περιγράψουμε δύο στρατηγικές με τις οποίες μπορούμε να διατάξουμε τα στοιχεία του πίνακα a σε σωρό.

Η πρώτη στρατηγική είναι η παρακάτω: Θεωρούμε το πρώτο στοιχείο του πίνακα ($a[1]$) σαν σωρό και εισάγουμε ένα προς ένα τα υπόλοιπα στοιχεία του πίνακα διατηρώντας κάθε φορά τη δομή του σωρού. Δηλαδή, όπως φαίνεται στον αλγόριθμο 4.1, κάθε στοιχείο ξεκινά από το τέλος του μέχρι στιγμής δέντρου και βρίσκει τη σωστή θέση του ανεβαίνοντας προς την ρίζα. Η χρονική πολυπλοκότητα του αλγορίθμου είναι $O(n \log n)$. Αυτό συμβαίνει διότι στη χειρότερη περίπτωση (δηλαδή όταν τα στοιχεία βρίσκονται αποθηκευμένα στον πίνακα με αύξουσα σειρά), κάθε στοιχείο πρέπει να διανύσει όλη την απόσταση ως τη ρίζα, συνεπώς αρκεί χρόνος $O(\log n)$.

Παρατήρηση 4.3. Η μέση χρονική πολυπλοκότητα είναι $O(n)$ (Άσκηση).

Αλγόριθμος 4.1 Κατασκευή σωρού (insert)

```
procedure insert (var a:array; n:integer);
var item:integer; k:integer;
begin
  item:=a[n]; k:=n div 2; (* εε  *)
  while ((k>0) and (a[k]<item)) do
    begin a[n]:=a[k]; n:=k; k:=k div 2; end;
  a[n]:=item
end

procedure ConstructHeapInsert (var a:array; n:integer);
var i:integer;
begin
  for i:=2 to n do insert(a,i)
end
```

Η δεύτερη στρατηγική κατασκευής ενός σωρού είναι η εξής: Θεωρούμε τον πίνακα a σαν δέντρο και εξετάζουμε ένα προς ένα όλα τα υποδέντρα του αρχίζοντας από το τέλος. Κάθε υποδέντρο το μετατρέπουμε σε σωρό και το ενώνουμε με την τελική δομή. Στη χειρότερη περίπτωση, ο χρόνος που χρειάζεται για την κατασκευή ενός σωρού με τη βοήθεια του αλγορίθμου 4.2, είναι $O(n)$ [1].

Όμως η χρήση της διαδικασίας *CostructHeapCombine* απαιτεί, όλα τα στοιχεία που θα φτιάξουν το σωρό να είναι διαθέσιμα από την αρχή της διαδικασίας, σε αντίθεση με τη χρήση της διαδικασίας *ConstructHeapInsert* όπου ένα καινούργιο στοιχείο μπορεί να εισαχθεί στο δέντρο οποιαδήποτε χρονική στιγμή.

Παρατήρηση 4.4. Η διαδικασία *combine* μπορεί να χρησιμοποιηθεί κι όταν θέλουμε να διαγράψουμε οποιοδήποτε στοιχείο ενός σωρού (όχι μόνο τη ρίζα) χωρίς να χαλάσουμε την ιδιότητα

[1] Αυτό προκύπτει ως εξής: Για κάθε i (του αλγορίθμου) ο αριθμός επαναλήψεων είναι $k - i$, όπου $k = \lceil \log n \rceil$. Οι κόμβοι που υπάρχουν για κάθε i είναι το πολύ 2^{i-1}. Συνεπώς :

$$\sum_{i=1}^{k} 2^{i-1}(k-i) \leq n \sum_{i} \frac{i}{2^i} = O(n)$$

Αλγόριθμος 4.2 Κατασκευή σωρού (combine)

```
procedure combine (var a:array; i,n:integer);
(* Μετατρέπει το υποδένδρο με ρίζα a[i] σε σωρό, υπό την προϋπόθεση
 ότι τα υποδένδρα με ρίζες τα παιδιά του a[i] είναι σωροί *)
var left, right, largest_child:integer;

begin
  while 2·i ≤ n do (* ο κόμβος i έχει τουλάχιστον ένα παιδί *)
  begin
    left:=2·i; (* ε *)
    right:=2·i+1; (* ε *)
    largest_child:=left;
    if right ≤ n and a[right]>a[left] then largest_child:=right;
    (* largest_child: η θέση του παιδιού με τη μεγαλύτερη τιμή *)
    if a[i]<a[largest_child] then
    begin
      swap(a[i],a[largest_child]);
      i:=largest_child
    end
    else i:= n div 2 + 1 (*exit while*)
  end
end

procedure CostructHeapCombine (var a:array; n:integer);
var i:integer;
begin
  for i:=n div 2 downto 1 do combine(a,i,n)
end
```

σωρού.

Μια από τις εφαρμογές του σωρού είναι η ταξινόμηση (*sorting*). Στην ταξινόμηση η απλή στρατηγική επιβάλλει συνεχώς να διαλέγουμε από τα στοιχεία που απομένουν, το μεγαλύτερο (ή το μικρότερο). Ένας αλγόριθμος που θα χρησιμοποιούσε αυτή τη στρατηγική όπως είναι, χωρίς καμία βελτίωση της αρχικής σκέψης, θα απαιτούσε στη χειρότερη περίπτωση χρόνο $O(n^2)$ ($n-1$ συγκρίσεις για κάθε στοιχείο). Η χρήση σωρού επιτρέπει την εύρεση του μεγαλύτερου στοιχείου και τη διαγραφή του από τα υπόλοιπα σε χρόνο τάξης $O(\log n)$. Έτσι επιτυγχάνεται για όλη τη διαδικασία της ταξινόμησης χρόνος στη χειρότερη περίπτωση της τάξης $O(n \log n)$.

Η μέθοδος που κάνει ταξινόμηση είναι η παρακάτω (αλγόριθμος 4.3): Κατασκευάζουμε από τον δοσμένο πίνακα $a[1..n]$, ένα σωρό με κάποια από τις μεθόδους που ήδη περιγράφηκαν. Έπειτα ανταλλάσσουμε (*swap*) το πρώτο στοιχείο της δομής ($a[1]$) με το τελευταίο ($a[n]$). Έτσι το τελευταίο στοιχείο του πίνακα τώρα είναι το μεγαλύτερο. Στη συνέχεια κάνουμε σωρό τον πίνακα

$a[1..n-1]$, παίρνουμε πάλι το πρώτο στοιχείο και το βάζουμε στη θέση $a[n-1]$, κ.ο.κ. Τελικά, και μετά από χρόνο $O(n \log n)$, ο πίνακας a είναι ταξινομημένος σε αύξουσα σειρά (ascending order).

Αλγόριθμος 4.3 Ταξινόμηση HeapSort

```
procedure HeapSort (var a:array; n:integer);
var i:integer;
begin
  ConstructHeap(a,n);  (* Αλγόριθμος 1 ή Αλγόριθμος 2 *)
  for i:=n downto 2 do
    begin swap(a[1],a[i]); combine(a,1,i−1) end
end
```

Ένα εύλογο ερώτημα είναι το εξής: Υπάρχει αλγόριθμος ο οποίος να ταξινομεί με **συγκρίσεις** στοιχεία σε χρόνο μικρότερο από αυτόν της τάξης $O(n \log n)$ (μιλώντας πάντα για τη χειρότερη περίπτωση); Η απάντηση είναι πως δεν είναι δυνατόν να υπάρξει τέτοιος αλγόριθμος.

Παρατήρηση 4.5. Πολυπλοκότητα δέντρου απόφασης λέγεται το ύψος του, αφού το ύψος καθορίζει τον αριθμό συγκρίσεων που θα χρειαστεί στη χειρότερη περίπτωση ένας αλγόριθμος που υλοποιεί τις συγκρίσεις του δέντρου.

Θεώρημα 4.6. *Κάθε δέντρο απόφασης για ταξινόμηση n στοιχείων, έχει πολυπλοκότητα $\Omega(n \log n)$.*

Απόδειξη. Κάθε αλγόριθμος που ταξινομεί με συγκρίσεις, μπορεί να παρασταθεί με τη βοήθεια ενός **δέντρου απόφασης** (*decision tree*), δηλαδή ενός δυαδικού δέντρου του οποίου οι εσωτερικές κορυφές αντιπροσωπεύουν μία σύγκριση (μία απόφαση). Το δέντρο που θα χρησιμοποιηθεί για την ταξινόμηση n στοιχείων θα έχει οπωσδήποτε $n!$ φύλλα (όλες οι δυνατές μεταθέσεις των n στοιχείων). Το μήκος του δρόμου απ' τη ρίζα στο φύλλο ενός δέντρου απόφασης μας δίνει τον αριθμό των συγκρίσεων που χρειάστηκαν για την ταξινόμηση που παριστάνει το φύλλο. Το μήκος του μεγαλύτερου απ' τα μονοπάτια, δηλαδή το ύψος του δέντρου μας δίνει το αριθμό των συγκρίσεων στη χειρότερη περίπτωση. Μπορούμε να προσδιορίσουμε το ελάχιστο ύψος h ενός δέντρου απόφασης n στοιχείων το οποίο έχει $n!$ φύλλα ως εξής. Ακόμη και αν το δέντρο είναι εντελώς πλήρες δυαδικό θα έχει το πολύ 2^h φύλλα. Επομένως θα πρέπει να ισχύει:

$$2^h \geq n! \Leftrightarrow h \geq \log(n!) \\ n! \geq (\tfrac{n}{2})^{\frac{n}{2}} \Big\} \Rightarrow h \geq \frac{n}{2} \log(\frac{n}{2})$$

□

Σαν συνέπεια έχουμε το ακόλουθο:

Πόρισμα 4.7. *Η ταξινόμηση n αριθμών με συγκρίσεις έχει χρονική πολυπλοκότητα $\Theta(n \log n)$.*

Απόδειξη. Από το πιο πάνω θεώρημα είδαμε ότι χρειαζόμαστε για την ταξινόμηση n στοιχείων με συγκρίσεις, χρόνο $\Omega(n \log n)$. Όπως είδαμε ο αλγόριθμος 4.3 έχει πολυπλοκότητα $O(n \log n)$.

Συνεπώς το πρόβλημα της ταξινόμησης με συγκρίσεις λύνεται σε χρόνο $\Theta(n \log n)$, άρα ο αλγόριθμος 4.3 είναι *βέλτιστος*. □

4.3 Σύνολα - Συστήματα Εισαγωγής και Ανάκτησης Πληροφοριών - Πράξεις σε Σύνολα

4.3.1 Γενικά

Στην σχεδίαση αλγορίθμων τα σύνολα είναι η βάση πολλών σπουδαίων και χρήσιμων **γενικευμένων δομών δεδομένων** (*abstract data types*). Έχουν αναπτυχθεί πολλές τεχνικές υλοποίησης τέτοιων γενικευμένων δομών δεδομένων που βασίζονται σε σύνολα.

Η δομή του συνόλου είναι η βάση πολλών προβλημάτων στα οποία μας ενδιαφέρει η γρήγορη ανάκτηση πληροφοριών (*information retrieval*). Σε αυτά τα προβλήματα, συνήθως έχουμε ένα σύμπαν, καθολικό σύνολο (*universe*) από το οποίο μπορούν να πάρουν στοιχεία όλα τα σύνολα (*sets*) που χρησιμοποιούνται. Οι περιπτώσεις που συναντάμε είναι:

- | sets | $\sim$ | universe |, δηλαδή οι πληθικοί αριθμοί των συνόλων που χρησιμοποιούνται είναι της τάξης του πληθικού αριθμού (cardinality) του καθολικού συνόλου.
- | sets | $<<$ |universe|, # operations $\sim$ | universe |, δηλαδή οι πληθικοί αριθμοί των συνόλων που χρησιμοποιούνται είναι πολύ μικρότεροι από τον πληθικό αριθμό του καθολικού συνόλου και επιπλέον ο αριθμός των πράξεων ανάμεσα στα σύνολα είναι πολύ μεγάλος.
- | sets | $<<$ | universe |, # operations $<<$ |universe|, όμοια με την προηγούμενη περίπτωση, με τη διαφορά ότι ο αριθμός των πράξεων με τα σύνολα είναι μικρός.

Έστω ότι έχουμε ένα σύνολο αναφοράς U με n στοιχεία από το οποίο μπορούμε να κατασκευάσουμε άλλα σύνολα S, τα οποία είναι υποσύνολα του U.

Ένας τρόπος να παραστήσουμε τα σύνολα αυτά S, είναι με τη βοήθεια ενός διανύσματος μήκους n, $S[1..n]$ τέτοιου ώστε $S[i] = 1$ αν το i-οστό στοιχείο του U ανήκει στο S και $S[i] = 0$ σε άλλη περίπτωση.

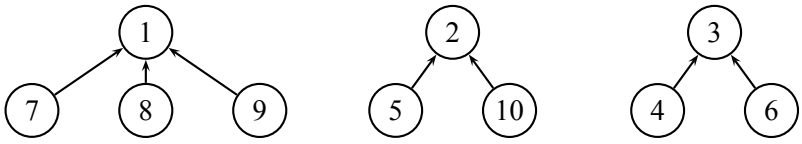
Άλλος τρόπος για να παραστήσουμε σύνολα S, ξένα μεταξύ τους, είναι με τη βοήθεια των δέντρων.

Παράδειγμα 4.8. Έστω

$$U = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\},$$
$$S_1 = \{1, 7, 8, 9\}, S_2 = \{2, 5, 10\}, S_3 = \{3, 4, 6\}.$$

Τα S_1, S_2, S_3 μπορούμε να τα παραστήσουμε όπως φαίνεται στο σχήμα 4.2. Δηλαδή κάθε κορυφή συνδέεται με τον γονέα της. Π.χ.

`Parent`[7]	$= 1,$	στοιχείο του ίδιου συνόλου
`Parent`[9]	$= 1,$	στοιχείο του ίδιου συνόλου
`Parent`[10]	$= 2,$	στοιχείο του ίδιου συνόλου
`Parent`[1]	$= 0,$	το 1 είναι ρίζα του δέντρου και χρησιμεύει σαν όνομα του συνόλου



$S_1 = \{1, 7, 8, 9\}$ $S_2 = \{2, 5, 10\}$ $S_3 = \{3, 4, 6\}$

Σχήμα 4.2: Παράσταση συνόλων ξένων μεταξύ τους με χρήση δένδρων

Έστω στοιχείο $a \in U$ (universe) και σύνολα $S, S_1, S_2 \subseteq U$. Οι πιο χαρακτηριστικές πράξεις μεταξύ συνόλων είναι οι παρακάτω:

- Member(a, S): Ελέγχει αν το στοιχείο a ανήκει στο σύνολο S και αν αυτό ισχύει επιστρέφει true αλλιώς false.
- Search(a, S): Επιστρέφει ένα δείκτη στο στοιχείο a αν $a \in S$ αλλιώς επιστρέφει nil.
- Insert(a, S): Εισάγει το στοιχείο a στο σύνολο S και επιστρέφει το σύνολο $S \bigcup \{a\}$.
- Delete(a, S): Διαγράφει το στοιχείο a από το σύνολο S και επιστρέφει το σύνολο $S \setminus \{a\}$.
- Union(S_1, S_2): Επιστρέφει το σύνολο $S_1 \bigcup S_2$. Υποθέτουμε ότι τα σύνολα S_1, S_2 είναι ξένα (disjoint) για να αποφύγουμε τον έλεγχο για διπλά στοιχεία.
- Find(a): Επιστρέφει το όνομα του συνόλου στο οποίο ανήκει το a. Υποθέτουμε ότι έχουμε διαμέριση σε ξένα σύνολα, άρα το στοιχείο a, θα ανήκει σε ένα ακριβώς σύνολο.
- Split(a, S): Χωρίζει το σύνολο S σε δύο σύνολα S_1, S_2 τέτοια ώστε:

  $$S_1 = \{b \mid b \leq a \bigwedge b \in S\} \text{ και } S_2 = \{b \mid b > a \bigwedge b \in S\}$$

  Εδώ υποθέτουμε ότι το σύνολο S είναι ένα σύνολο του οποίου τα στοιχεία έχουν μια γραμμική διάταξη ($\leq$).
- Max(S), Min(S): Επιστρέφει το μεγαλύτερο ή το μικρότερο των στοιχείων του S. Υποθέτουμε πάλι γραμμική διάταξη των στοιχείων του S.
- Successor(a, S): Επιστρέφει το μικρότερο από τα στοιχεία του S που είναι μεγαλύτερο του a (επόμενο στοιχείο). Predecessor(a, S): Ομοίως επιστρέφει το μεγαλύτερο από τα στοιχεία του S που είναι μικρότερο του a (προηγούμενο στοιχείο).

Ανάλογα λοιπόν με το ποιες λειτουργίες από τις παραπάνω χρησιμοποιούμε συχνά, φτιάχνουμε και τις κατάλληλες δομές, υλοποιώντας τα σύνολα με διάφορες τεχνικές.

4.3.2 Δομή λεξικού (Dictionary)

Ορισμός 4.9. Ας υποθέσουμε ότι έχουμε ένα σύνολο S και θέλουμε να εκτελούνται γρήγορα οι λειτουργίες της εισαγωγής καινούργιου στοιχείου, διαγραφής παλαιού στοιχείου και ελέγχου για την ύπαρξη κάποιου στοιχείου στο S. Θέλουμε δηλαδή να εκτελείται αποδοτικά μία ακολουθία από διαδικασίες *Insert, Delete* και *Member*. Αυτή τη γενικευμένη αφηρημένη δομή δεδομένων (ADT) την ονομάζουμε **λεξικό** (*Dictionary*).

Η υλοποίηση ενός λεξικού μπορεί να γίνει π.χ. με μια συνάρτηση

$$h\colon universe \to \{0, \ldots, m-1\},$$

που ονομάζεται **συνάρτηση κατακερματισμού** (*hashing function*). Φροντίζουμε ώστε η συνάρτηση h που διαλέγουμε να υπολογίζεται γρήγορα για οποιοδήποτε στοιχείο του καθολικού συνόλου, σε χρόνο δηλαδή $O(1)$. Θεωρούμε ένα πίνακα A (hash table). Κάθε στοιχείο $A[i]$ του πίνακα δείχνει σε μια λίστα η οποία περιέχει εκείνα τα στοιχεία a του καθολικού συνόλου για τα οποία ισχύει $h(a) = i$. Συνεπώς για να κάνουμε *Insert, Delete* και *Member* ένα στοιχείο a, αρκεί να ψάξουμε μόνο τη λίστα στην οποία δείχνει το στοιχείο $A[h(a)]$.

Βέβαια στη χειρότερη περίπτωση, είναι πιθανόν μετά από n εισαγωγές στοιχείων, να έχουμε μια λίστα μήκους σχεδόν n (δηλαδή σχεδόν όλα τα στοιχεία να έχουν πάει στην ίδια λίστα). Σε αυτή την περίπτωση, αν χρειαστεί να εκτελέσουμε n φορές τις διαδικασίες *Delete* ή *Member*, ο χρόνος που απαιτείται είναι $O(n^2)$. Αν όμως φροντίσουμε ώστε η εκλογή της συνάρτησης h να εξασφαλίζει μια όσο το δυνατό ομοιόμορφη κατανομή των στοιχείων στις λίστες, έτσι ώστε να μην υπάρχει συσσώρευση στοιχείων σε μια λίστα, τότε ο χρόνος αναζήτησης μπορεί να καλυτερεύσει σημαντικά. Για παράδειγμα, αν πρόκειται να εισαχθούν περίπου $n \in O(m)$ στοιχεία, τότε τη στιγμή που εισάγεται το i-οστό στοιχείο, η λίστα στην οποία θα πρέπει να μπει θα έχει αναμενόμενο μήκος $\frac{i-1}{m} < 1$ συνεπώς η κάθε διαδικασία που θα πρέπει να διατρέξει κάποια λίστα θα χρειάζεται περίπου σταθερό χρόνο $O(1)$ και έτσι n διαδικασίες θα χρειάζονται $O(n)$ χρόνο περίπου.

Είναι συνηθισμένο να μην γνωρίζουμε από πριν τον πληθικό αριθμό που μπορεί να έχει το σύνολό μας. Στην περίπτωση αυτή διαλέγουμε μια τιμή m για τον πίνακα (*hash table, bucket table*) και όταν ο αριθμός των στοιχείων γίνει μεγαλύτερος από m, δημιουργούμε ένα καινούργιο πίνακα-στήλη μεγέθους $2m$ και με rehashing (δηλαδή ορίζοντας μια νέα συνάρτηση με πεδίο τιμών αυτή τη φορά το $[0, 2m-1]$), βάζουμε τα στοιχεία στον καινούργιο πίνακα, καταστρέφοντας τον παλιό. Όταν τα στοιχεία γίνουν περισσότερα από $2m$, δημιουργούμε ένα άλλο πίνακα μεγέθους $4m$ κ.ο.κ. Είναι σαφές ότι κάθε φορά η κατάλληλη επιλογή της συνάρτησης κατακερματισμού παίζει σπουδαίο ρόλο προκειμένου να διατηρήσουμε τους χρόνους προσπέλασης μικρούς.

Παράδειγμα 4.10. Έστω ότι το σύνολό μας αποτελείται από ακεραίους που μπορούν να πάρουν τιμές στο διάστημα $[0, r], r > n$. Τότε αν χρησιμοποιήσουμε τη συνάρτηση $h(a) = a \bmod m$, όπου m το μέγεθος του τρέχοντα πίνακα-στήλη έχουμε τα παρακάτω: Έστω ότι εισάγουμε τους αριθμούς $1, 5, 8, 3, 9, 6$. Αρχικά επιλέγουμε $m = 2$ και έχουμε :

$$\begin{array}{ll} 0: & \\ 1: & 1, 5 \end{array}$$

Σε αυτό το σημείο επιλέγουμε $m = 4$:

$$\begin{array}{ll} 0: & 8 \\ 1: & 1, 5 \\ 2: & \\ 3: & 3 \end{array}$$

Τέλος με $m = 8$ προκύπτει το παρακάτω:

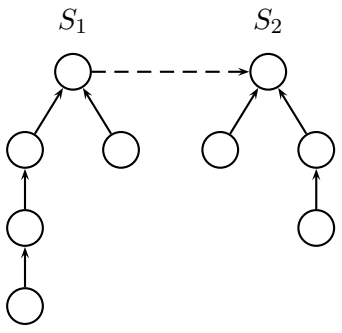
$$\begin{array}{ll} 0: & 8 \\ 1: & 1, 9 \\ 2: & \\ 3: & 3 \\ 4: & \\ 5: & 5 \\ 6: & 6 \\ 7: & \end{array}$$

4.3.3 Δομή Union–Find

Ορισμός 4.11. Έστω ότι έχουμε διάφορα ξένα μεταξύ τους σύνολα και μας ενδιαφέρει η αποδοτική υλοποίηση μιας **ακολουθίας** από διαδικασίες ένωσης (*Union*) συνόλων και εύρεσης του συνόλου στο οποίο ανήκει κάποιο στοιχείο (*Find*). Μια τέτοια δομή δεδομένων ονομάζεται δομή Union-Find.

Η αναπαράσταση των συνόλων μπορεί να γίνει π.χ. με δέντρα, όπου κάθε κόμβος περιέχει ένα στοιχείο και έχει ένα pointer προς τον γονέα του. Κατά σύμβαση το όνομα του συνόλου είναι το στοιχείο που τυχαίνει να βρίσκεται στη ρίζα.

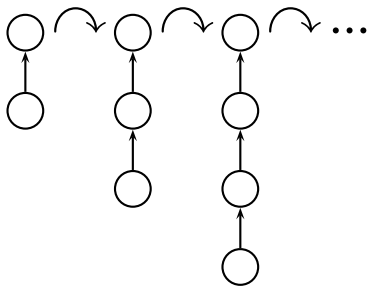
Η array Parent[i], μας δίνει τον γονέα του κόμβου i και θέτουμε Parent[root]:=0. Για να βρούμε την ένωση δύο συνόλων S_1, S_2 αρκεί να μεταβάλλουμε την εγγραφή της Parent σε μια απ' τις δύο ρίζες των S_1, S_2, έτσι ώστε αντί να δείχνει 0, να δείχνει στη ρίζα του άλλου δέντρου (σχήμα 4.3). Η υλοποίηση της συνάρτησης Union φαίνεται στον αλγόριθμο 4.4.



Σχήμα 4.3: Η ένωση των συνόλων S_1 και S_2

Η εκτέλεση της συνάρτησης Union γίνεται σε σταθερό χρόνο $O(1)$, ενώ για την εύρεση του συνόλου S στο οποίο ανήκει το στοιχείο i, αρκεί να βρούμε τη ρίζα του δέντρου που αναπαριστά το σύνολο S. Η υλοποίηση της συνάρτησης Find φαίνεται στον αλγόριθμο 4.5.

Στη χειρότερη περίπτωση, ένα εκφυλισμένο (*degenarate*) δέντρο μπορεί να προκύψει από την ένωση πολλών συνόλων όπως στο σχήμα 4.4. Έτσι λοιπόν ο χρόνος που χρειάζεται η Find για να διανύσει ένα μονοπάτι του συνόλου-δέντρου με n-στοιχεία-κορυφές είναι στην χειρότερη περίπτωση $O(n)$. Συνεπώς n εκτελέσεις της Find χρειάζονται χρόνο $O(n^2)$.



Σχήμα 4.4: Εκφυλισμένο δένδρο μετά από την ένωση πολλών συνόλων

Μπορούμε να βελτιώσουμε αυτό το χρόνο αν λάβουμε υπόψη μας τον αριθμό των στοιχείων που έχει κάθε σύνολο και συνδέουμε κάθε φορά το σύνολο-δέντρο με τα λιγότερα στοιχεία-κόμβους σε εκείνο με τα περισσότερα, αλλάζοντας τη συνάρτηση Union (Balancing).

Θέτουμε την πληροφορία του πληθικού αριθμού κάθε συνόλου σαν τιμή του γονέα της ρίζας του ($Parent[root] := -\#elements$). Ο αρνητικός αριθμός ξεχωρίζει την τιμή του γονέα της ρίζας από τα υπόλοιπα στοιχεία του συνόλου. Η νέα υλοποίηση της συνάρτησης Union φαίνεται στον αλγόριθμο 4.6.

Λήμμα 4.12. *Ένα δέντρο που κατασκευάστηκε με τη βοήθεια του αλγόριθμου 4.6 έχει ύψος μικρότερο από* $\lfloor \log n \rfloor + 1$.

Απόδειξη. Επαγωγική βάση: $n = 1$. Προφανές.

Επαγωγικό βήμα: Έστω αληθές για όλα τα δέντρα με αριθμό κόμβων $\leq n - 1$. Έστω ότι η τελευταία εφαρμογή της διαδικασίας ήταν $union(k, j)$ και ότι το δέντρο j είχε m κόμβους. Χωρίς βλάβη της γενικότητας $1 \leq m \leq \frac{n}{2}$.

Περίπτωση 1: Νέο ύψος = ύψος του δέντρου k:

$$\text{ύψος} \leq \lfloor \log(n - m) \rfloor + 1 \leq \lfloor \log n \rfloor + 1.$$

Αλγόριθμος 4.4 Διαδικασία ένωσης (Union)

```
function Union (i, j: integer (*set*)): integer (*set*);
begin
  Parent[i]:=j; return j
end
```

Αλγόριθμος 4.5 Διαδικασία εύρεσης (Find)

```
function Find (i: integer (*element*)): integer (*set*);
begin
  while Parent[i]>0 do i := Parent[i];
  return i
end
```

Αλγόριθμος 4.6 Βελτιωμένη διαδικασία ένωσης (Balancing)

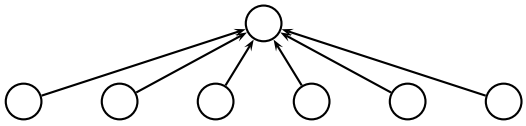
```
function Union (i, j: integer (*set*)): integer (*set*);
var x: integer; (*αρνητικός αριθμός στοιχείων του νέου συνόλου*)
begin
  x:=Parent[i]+Parent[j];
  if |Parent[i]| < |Parent[j]| then
    begin Parent[i]:=j; Parent[j]:=x; return j end
  else begin Parent[j]:=i; Parent[i]:=x; return i end
end
```

Περίπτωση 2: Νέο ύψος = ύψος του δέντρου $j+1$:

$$\text{ύψος} \leq \lfloor \log m \rfloor + 2 \leq \lfloor \log \frac{n}{2} \rfloor + 2 \leq \lfloor \log n \rfloor + 1.$$

□

Σαν αποτέλεσμα έχουμε ότι ο χρόνος που χρειάζεται μια εκτέλεση της συνάρτησης *Find* είναι $O(\log n)$, δηλαδή n εκτελέσεις χρειάζονται $O(n \log n)$ χρόνο. Μπορούμε να πετύχουμε μια ακόμη καλυτέρευση του χρόνου, αλλάζοντας αυτή τη φορά τη συνάρτηση *Find* με προσθήκη της διαδικασίας *Path Compression* : ένα προς ένα τα στοιχεία-κορυφές που συναντά ο αλγόριθμος στο δρόμο για τη ρίζα, τα «ξεκρεμά» από τη θέση τους και τα «κρεμάει» από τη ρίζα, με αποτέλεσμα το σύνολο-δέντρο να τείνει προοδευτικά να μετασχηματιστεί όπως στο σχήμα 4.5. Η νέα υλοποίηση της συνάρτησης *Find* φαίνεται στον αλγόριθμο 4.7.



Σχήμα 4.5: Path compression

Είναι προφανές ότι αυτός ο τρόπος συμφέρει όταν πρόκειται να εκτελεστούν πολύ περισσότερα του ενός *Find*. Αποδεικνύεται μάλιστα ότι n εκτελέσεις της συνάρτησης *Find* χρειάζονται χρόνο $O(n\alpha(n))$, όπου $\alpha(n)$ είναι ψευδοαντίστροφη της συνάρτησης Ackermann(σχεδόν σταθερά). Η ιδέα αυτή είναι παράδειγμα **αποσβεστικής** αποδοτικότητας (*amortization*). Ένα βήμα της *Path*

Compression κοστίζει αλλά έτσι εξοικονομείται χρόνος για κατοπινές εφαρμογές της *Find*. Η βελτίωση αυτή οφείλεται στον Tarjan [**?**].

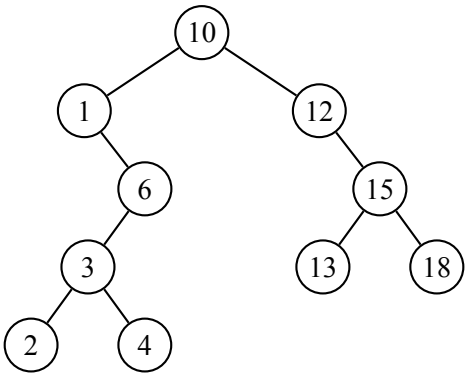
Αλγόριθμος 4.7 Βελτιωμένη διαδικασία εύρεσης (Path compression)

```
function Find(i: integer (*element*)): integer (*set*);
var j, t: integer (*element*);
begin
  j := i;
  (*Εύρεση της ρίζας*)
  while Parent[j] > 0 do j := Parent[j];
  (*Ξεκρέμασμα των φύλλων και κρέμασμα από τη ρίζα*)
  while (i ≠ j) do
    begin t := Parent[i]; Parent[i] := j; i := t end;
  return j
end
```

4.3.4 Δυαδικά δέντρα αναζήτησης (binary search trees)

Ένας άλλος τρόπος αναπαράστασης συνόλων με μια γραμμική διάταξη, είναι με **δυαδικά δέντρα**. Αυτή η δομή είναι χρήσιμη όταν έχουμε μεγάλα σύνολα και είναι ασύμφορη η χρησιμοποίηση των προηγούμενων μεθόδων. Ένα **δυαδικό δέντρο αναζήτησης** (*binary search tree*) μπορεί να υποστηρίξει αποδοτικά πράξεις όπως *Insert, Delete, Member* και *Min* απαιτώντας κατά μέσο όρο $O(\log n)$ χρόνο για κάθε διαδικασία (όπου n ο πληθάριθμος του συνόλου).

Τα στοιχεία του συνόλου διατάσσονται στο δυαδικό δέντρο ως εξής: Όλα τα στοιχεία-κορυφές που βρίσκονται στο αριστερό υποδέντρο μιας κορυφής x, είναι μικρότερα από το στοιχείο που βρίσκεται στην κορυφή x ενώ τα στοιχεία του δεξιού υποδέντρου είναι μεγαλύτερα. Αυτή η συνθήκη (*binary search tree property*) ισχύει για όλες τις κορυφές του δέντρου (σχήμα 4.6).



Σχήμα 4.6: Δυαδικό δένδρο αναζήτησης

Ο έλεγχος για το αν ένα στοιχείο x ανήκει στο σύνολο γίνεται όπως παρακάτω:

Συγκρίνουμε το στοιχείο x με τη ρίζα του δέντρου

- Αν τα στοιχεία είναι ίσα η διαδικασία τελειώνει.
- Αν το x είναι μικρότερο προχωράμε στο αριστερό υποδέντρο του x και επαναλαμβάνουμε τη διαδικασία.
- Αν το x είναι μεγαλύτερο προχωράμε στο δεξιό υποδέντρο του x και επαναλαμβάνουμε τη διαδικασία.

Η συνάρτηση Member φαίνεται στο αλγόριθμο 4.8. Κάθε κόμβος περιέχει κάποιο στοιχείο του συνόλου και δύο δείκτες, ένα στο αριστερό παιδί και ένα στο δεξιό. Οι διαδικασίες Insert, RetrieveMin και Delete είναι εντελώς ανάλογες και φαίνονται στους αλγόριθμους 4.9, 4.10 και 4.11.

Αλγόριθμος 4.8 Συνάρτηση Member σε δυαδικό δένδρο αναζήτησης

```
function Member(x:elementtype; A:^node):boolean;
begin
  if A=Nil then return(false)
  else if x=A^.element then return true
  else if x<A^.element then return Member(x,A^.leftchild)
  else return Member(x,A^.rightchild)
end
```

Αλγόριθμος 4.9 Συνάρτηση Insert σε δυαδικό δένδρο αναζήτησης

```
procedure Insert(x:elementtype; var A:^node);
begin
  if A=Nil then
  begin
    new(A); A^.element:=x;
    A^.leftchild:=Nil; A^.rightchild:=Nil
  end
  else if x<A^.element then Insert(x,A^.leftchild)
  else if x>A^.element then Insert(x,A^.rightchild)
  (* Αν x=A^.element, τότε ήδη x είναι στο A *)
end
```

4.3.5 Ισοζυγισμένα Δέντρα (Balanced Trees)

Όπως είδαμε στην προηγούμενη παράγραφο, χρησιμοποιώντας ένα δυαδικό δέντρο αναζήτησης σαν αναπαράσταση ενός συνόλου, έχουμε έναν αναμενόμενο χρόνο (*average-case complexity*)

Αλγόριθμος 4.10 Συνάρτηση RetrieveMin σε δυαδικό δένδρο αναζήτησης

```
function RetrieveMin(var A:^node):elementttype; (* Επιστρέφει,
διαγράφοντας από το σύνολο A το μικρότερο στοιχείο του *)
begin
  if A^.leftchild=Nil then
    (∗ Το A δείχνει στο μικρότερο στοιχείο ∗)
    begin return A^.element; A:=A^.rightchild end
  else return RetrieveMin(A^.leftchild)
end
```

Αλγόριθμος 4.11 Συνάρτηση Delete σε δυαδικό δένδρο αναζήτησης

```
procedure Delete(x:elementtype; var A:^node);
begin
  if A<>Nil then
    if x<A^.element then Delete(x, A^.leftchild)
    else if x>A^.element then Delete(x,A^.rightchild)
    else (* x=A^.element *)
          if A^.leftchild=Nil then A:=A^.rightchild
          else if A^.rightchild=Nil then A:=A^.leftchild
          else (∗ ο κόμβος με το x έχει 2 παιδιά ∗)
              A^.element := RetrieveMin(A^.rightchild)
end
```

$O(\log n)$ για κάθε προσπέλαση. Στη χειρότερη περίπτωση όμως, αν εισάγουμε συνεχώς στοιχεία στο σύνολό μας, μπορεί να καταλήξουμε σε εκφυλισμένο δέντρο του οποίου το ύψος προσεγγίζει τον αριθμό των κορυφών του. Έτσι μια προσπέλαση σε αυτό το δέντρο χρειάζεται χρόνο $O(n)$.

Έχουν αναπτυχθεί διάφορες τεχνικές που φροντίζουν να διατηρούν το δέντρο **ισοζυγισμένο** (*balanced*) κατά τη διάρκεια διαφόρων διαδικασιών που το μεταβάλλουν (Insert, Delete κ.α.). Δύο από αυτές τις τεχνικές είναι τα 2-3 trees και τα AVL trees[2].

Ορισμός 4.13. Το 2-3 tree είναι ένα δέντρο στο οποίο κάθε κορυφή που δεν είναι φύλλο έχει δύο ή τρία παιδιά και κάθε μονοπάτι από τη ρίζα σε ένα φύλλο έχει το ίδιο μήκος.

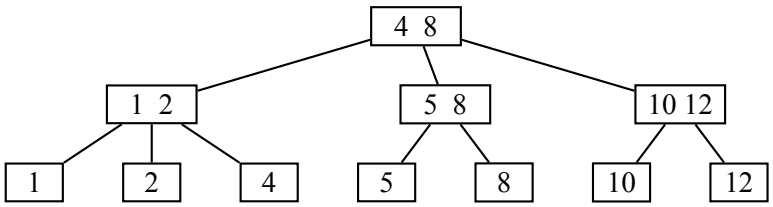
Τα στοιχεία εισάγονται σε ένα 2-3 tree συνήθως ως εξής: Κάθε εσωτερική κορυφή έχει δύο θέσεις.

- Στην αριστερή θέση εισάγεται το μεγαλύτερο από τα στοιχεία του υποδέντρου που έχει ρίζα το αριστερό παιδί της κορυφής αυτής, στοιχείο που (πρέπει να) είναι και μικρότερο ή ίσο από τα στοιχεία των υπόλοιπων παιδιών.

[2] Το όνομα AVL προέρχεται από τα αρχικά των δημιουργών του Adelson-Velskii-Landis (1962)

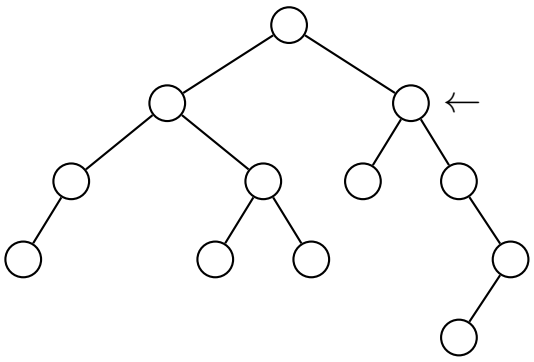
- Στη δεξιά θέση εισάγεται το μεγαλύτερο από τα στοιχεία του μεσαίου υποδέντρου, που (πρέπει να) είναι και μικρότερο ή ίσο από τα στοιχεία δεξιού υποδέντρου.

Τα φύλλα του δέντρου περιέχουν τα στοιχεία του συνόλου (σχήμα 4.7).



Σχήμα 4.7: 2-3 δένδρο

Ορισμός 4.14. Το **AVL tree** είναι ένα δυαδικό δέντρο αναζήτησης με την εξής ιδιότητα, το ύψος του αριστερού και του δεξιού υποδέντρου κάθε κορυφής, διαφέρουν το πολύ κατά 1.



Σχήμα 4.8: Το δένδρο δεν είναι AVL εξαιτίας της σημειωμένης κορυφής

Παράδειγμα 4.15. Το δέντρο στο σχήμα 4.8 δεν είναι AVL tree διότι η σημειωμένη κορυφή έχει ένα αριστερό υποδέντρο ύψους 0 και ένα δεξιό υποδέντρο ύψους 2. Παρ' όλα αυτά η ιδιότητα του AVL tree ισχύει σε κάθε άλλη κορυφή.

Οι διαδικασίες αναπτύσσονται έτσι ώστε να διατηρούν τη μία ή την άλλη δομή, με συνέπεια η προσπέλαση στα δέντρα να γίνεται στη χειρότερη περίπτωση (*worst-case complexity*) σε χρόνο $O(\log n)$.

4.4 Διαδραστικό υλικό - Σύνδεσμοι

- Στην σελίδα http://visualgo.net/ θα βρείτε οπτικοποιήσεις δομών δεδομένων με γραφικά.
- Στην σελίδα http://www.algolist.net/ υπάρχουν υλοποιήσεις δομών δεδομένων σε C++ και Java.

- Η σελίδα http://www.graphtheory.com/ είναι κεντρικός κόμβος για πληροφορίες, νέα και βιβλιογραφία στην Θεωρία Γραφημάτων.
- Εδώ μπορείτε να βρείτε υλοποιήσεις σε Python αναπαραστάσεων και βασικών λειτουργιών γραφημάτων.

4.5 Ασκήσεις

1. Για το παρακάτω σύνολο αριθμών, εκτελέστε τα στάδια κατασκευής σωρού για τους δύο αλγορίθμους heap combine και heap insert:

   $$\{x, y, z, w, 14, 5, 6, 15, 8, 7, 6, 1, 3\}$$

   όπου x, y, z, w είναι τα τέσσερα τελευταία ψηφία του αριθμού ταυτότητάς σας. Δείξτε τα στάδια εκτέλεσης με σχήματα.

2. Πόσα βήματα χρειάζεται ο αλγόριθμος heapsort για να ταξινομήσει ένα διάνυσμα με:

   - ήδη ταξινομημένα στοιχεία κατά αύξουσα σειρά;
   - ήδη ταξινομημένα στοιχεία κατα φθίνουσα σειρά;

3. Δείξτε ότι στη χειρότερη περίπτωση ο αλγόριθμος heapsort έχει πολυπλοκότητα $\Omega(n \log n)$.

4. Σε έναν κατευθυνόμενο γράφο, ένας κόμβος με indegree μηδέν λέγεται πηγή.

   (α) Αποδείξτε ότι σε κάθε ακυκλικό κατευθυνόμενο γράφο υπάρχει τουλάχιστον μία πηγή.

   (β) Δείξτε ότι ένας κατευθυνόμενος γράφος με n κόμβους είναι ακυκλικός αν και μόνο αν μπορούμε να τοποθετήσουμε ετικέτες $1, 2, \ldots, n$ στους κόμβους ώστε όλες οι ακμές να κατευθύνονται από κόμβο με μικρότερη ετικέτα σε κόμβο με μεγαλύτερη ετικέτα.

   (γ) Περιγράψτε πολυωνυμικό αλγόριθμο που να αποφαίνεται αν ένας κατευθυνόμενος γράφος είναι ακυκλικός.

5. Σχεδιάστε συνάρτηση που με είσοδο ένα δυαδικό δέντρο αναζήτησης και αριθμό x, επιστρέφει δείκτη στον μεγαλύτερο αριθμό στο δέντρο που είναι μικρότερος του x ή nil αν δεν υπάρχει τέτοιος αριθμός.

6. Σχεδιάστε συνάρτηση που να συνδυάζει σωστά δύο δυαδικά δέντρα αναζήτησης.

7. Σχεδιάστε τις συναρτήσεις Insert και RetrieveMin για 2-3 tree.

8. Σχεδιάστε τη συνάρτηση Delete για AVL tree.

Βιβλιογραφία

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest and Cliff Stein, “Introduction to Algorithms”, 3rd edition, MIT Press, 2009.

[2] C.L. Liu. Στοιχεία Διακριτών Μαθηματικών (απόδοση στα Ελληνικά: Κ. Μπους και Δ. Γραμμένος). Πανεπιστημιακές Εκδόσεις Κρήτης, 2003.

[3] Γεώργιος Γεωργακόπουλος, Δομές Δεδομένων, Πανεπιστημιακές Εκδόσεις Κρήτης, 2002, 960-524-125-0

[4] Παναγιώτης Μποζάνης, Δομές δεδομένων, 960-418-010-X, Τζιόλας, 2003