ΕΙΣΑΓΩΓΗ

Σε κάθε αλγόριθμο οι ασυμπτωτικές εκτιμήσεις χρόνου είναι πάντοτε βασισμένες στο μοντέλο υπολογισμού RAM (random access machine). Στο μοντέλο υπολογισμού RAM η προσπέλαση της μνήμης γίνεται μέσω υπολογισμού διευθύνσεων, δηλαδή υποστηρίζεται η χρήση πινάκων (arrays).

Σε αντιδιαστολή με τη RAM υπάρχει και η μηχανή δεικτών (pointer machine) όπου η προσπέλαση μνήμης είναι δυνατή μόνο μέσω δεικτών, δηλαδή οι διευθύνσεις είναι τιμές δεικτών. Η pointer machine εκπροσωπεί υλοποιήσεις αλγορίθμων με records και είναι ασθενέστερο μοντέλο απ' τη RAM.

ΔΙΑΤΑΞΗ ΣΤΟΙΧΕΙΩΝ (SORTING)

1) **Bubblesort**: κεντρική ιδέα: α) άλλαξε τα ζεύγη εκείνα που βρίσκονται εκτός διάταξης

β) επανέλαβε το προηγούμενο βήμα έως ότου διαταχθούν όλα τα ζεύγη.

Εμφανίζεται σε δύο μορφές. Στην πρώτη μορφή η επεξεργασία των στοιχείων λαμβάνει χώρα από αριστερά προς τα δεξιά και στη δεύτερη μορφή η επεξεργασία των στοιχείων γίνεται από αριστερά πρός τα δεξιά και απ' τα δεξιά πρός τα αριστερά εναλλάξ. (coctail-shakersort).

π.χ. input: 4 3 8 6 2 5 7

⇓

3 4 8 6 2 5 7

⇓

3 4 6 8 2 5 7

⇓

3 4 6 2 8 5 7

⇓

3 4 6 2 5 8 7

⇓

3 4 6 2 5 7 8

⇓

⋮

output: 2 3 4 5 6 7 8

Αρχικά, συγκρίνουμε το 1ο στοιχείο με το 2ο στοιχείο. Αν το 2ο είναι μικρότερο του 1ου τότε τα ανταλλάσσουμε και μετά συγκρίνουμε το τυχόν νέο 2ο στοιχείο με το 3ο κ.ο.κ. αν 2ο > 3ο τα αλλάζουμε, αλλιώς προχωράμε στο 4ο στοιχείο το οποίο συγκρίνουμε με το 5ο, κ.ο.κ. Μόλις θα έχουμε διατρέξει 1 φορά το input array, το μεγαλύτερο στοιχείο θα βρίσκεται στα δεξιότερα και το πρόβλημα θα έχει βελτιωθεί κατά 1. Μετά καλούμαστε να εφαρμόσουμε την ίδια διαδικασία με την προηγούμενη για τα υπόλοιπα n-1 στοιχεία (αν είναι n ο αρχικός αριθμός των στοιχείων του input).

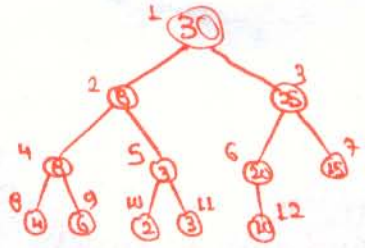
Πολυπλοκότητα: $n+(n-1)+(n-2)+\ldots+1 = \frac{n(n+1)}{2} = O(n^2)$ στη χειρότερη περίπτωση, γιατί έχουμε n βήματα με αριθμό συγκρίσεων $\leq n-1$ στο καθένα.

2) **Heapsort**:

- Μια δομή heap (=σωρός) είναι ένα δένδρο του οποίου οι κόμβοι φέρουν αριθμούς με την ιδιότητα ότι για κάθε κόμβο v ισχύει: αριθμός(p(v)) ≥ αριθμός(v). Σε ένα τέτοιο δένδρο ο κάθε πατέρας είναι μεγαλύτερος απ' τα παιδιά του και ο μεγαλύτερος αριθμός βρίσκεται στη ρίζα.
- Στην υλοποίηση του heapsort χρησιμοποιούνται ισοζυγισμένα και δυϊκά heaps, όπου δυϊκά σημαίνει ότι κάθε κόμβος έχει 0 ή 2 παιδιά με ενδεχόμενη εξαίρεση κάποιων κόμβων και ισοζυγισμένα σημαίνει ότι τα φύλλα έχουν βάθος k ή k+1 ή ότι βρίσκονται στο k ή k+1 επίπεδο. Τα φύλλα βάθους k+1 είναι συγκεντρωμένα πρός τα αριστερά.
- Το ισοζυγισμένο και δυϊκό heap μπορεί συνοπτικά να παρασταθεί με ένα array.

Π.Χ.

30	8	25	8	3	20	15	4	6	2	3	10
1	2	3	4	5	6	7	8	9	10	11	12



Η αρίθμηση των κόμβων γίνεται από τη ρίζα και κάτω:

$father(i) = \left\lfloor \frac{i}{2} \right\rfloor$

$left(i) = 2i$

$right(i) = 2i+1$

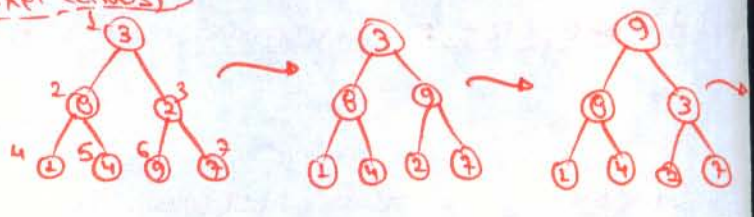
(μόνο για ισοζυγισμένο και δυαδικό heap)

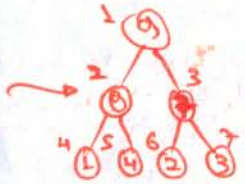
• Ο heapsort χωρίζεται σε δύο φάσεις.
Τη φάση δόμησης και τη φάση διαλογής.
Στη φάση δόμησης δημιουργείται το αντίστοιχο heap
ενώ στην φάση διαλογής χρησιμοποιείται η extractmax(),
η οποία όταν εφαρμόζεται σε ένα κόμβο βρίσκει το μεγαλύτερο παιδί του και το βάζει στην πατέρα, ενώ ο πατέρας βγαίνει από το heap. (διαλέγει και απομακρύνει το μεγαλύτερο στοιχείο, μετατρέπει τα υπόλοιπα στο αντίστοιχο heap και επαναλαμβάνει τη διαδικασία μέχρι τέλους)

Π.Χ.

3	8	2	1	4	9	7
1	2	3	4	5	6	7

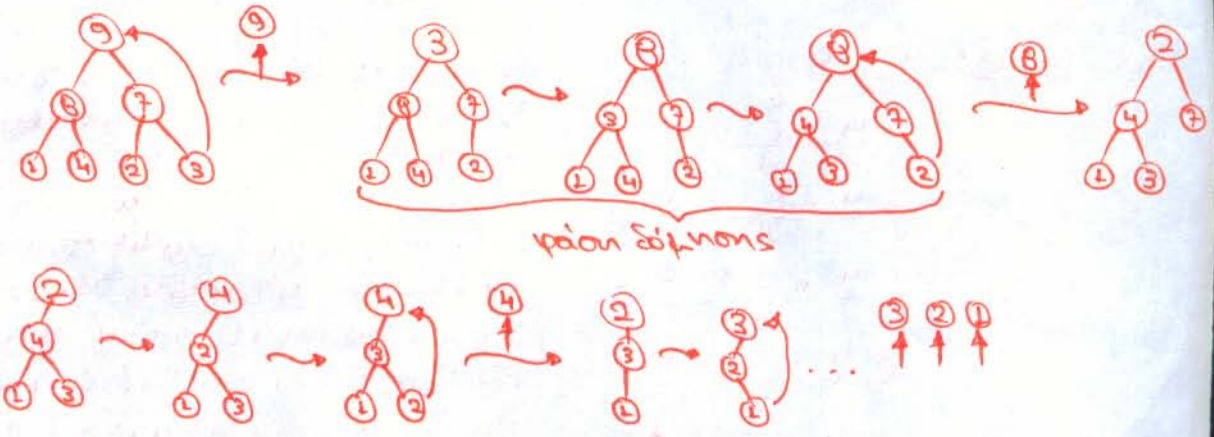
φάση δόμησης:





Κάθε φορά κάνουμε εναλλαγή των αριθμών για τους οποίους δεν ικανοποιείται η heap-property.

φάση διαλογής:



Στη φάση διαλογής παίρνουμε το μέγιστο στοιχείο (το οποίο είναι αποθηκευμένο στη ρίζα) και στη θέση του βάζουμε το πιο δεξί από τα παιδιά. Στη συνέχεια χρησιμοποιείται η extractmax η οποία εξάγει το μεγαλύτερο στοιχείο. Έπειτα εφαρμόζεται η φάση δόμησης ώστε να ικανοποιείται η heap-property και επαναλαμβάνεται πάλι η ίδια διαδικασία. Τελικά, όλα τα στοιχεία που θα εξαχθούν από το αρχικό heap θα είναι διατεταγμένα σε ένα πίνακα.

• Πολυπλοκότητα για τη φάση δόμησης: $\sum_{i=1}^{n}$ (κόστος πρόσθεσης του $f[i]$) $= \sum_{i=1}^{n} O(h(i)) =$

$= O\left(n + \sum_{k=0}^{\infty} k \cdot (\text{πλήθος κόμβων } i \text{ με } h(i)=k)\right) \underset{(*)}{=} O\left(n + \sum_{k=0}^{\infty} k \cdot \frac{n}{2^k}\right) = O\left(n \cdot \left(1 + \sum_{k=0}^{\infty} \frac{k}{2^k}\right)\right) = \underline{O(n)}$

(*) το πλήθος των κόμβων i με $h(i)=k$ είναι $\leq \frac{n}{2^k}$

• Πολυπλοκότητα για τη φάση διαλογής: το μέγιστο κάθε φορά στοιχείο απομακρύνεται από τη ρίζα του heap το πολύ n φορές. Μετά την απομάκρυνση κάθε μέγιστου, η μετατροπή σε heap κοστίζει $O(\log n)$. Οπότε το συνολικό κόστος είναι $\underline{O(n \log n)}$. (στη χειρότερη περίπτωση)

• Ο heapsort είναι ο καλύτερος αλγόριθμος για διάταξη η στοιχείων και έχει πολυπλοκότητα $\Theta(n \log n)$, είναι δηλαδή βέλτιστος (optimal). Ισχύει: $\Pi X \Pi$ (οποιαδήποτε αλγ. ταξ) $\geq$ ύψους δένδρου απόφασης

• Αν για n στοιχεία φτιάξουμε ένα δένδρο απόφασης, τότε ο αριθμός των εισόδων είναι $n!$ και άρα θα έχουμε $n!$ φύλλα. Ξέραμε, όμως, ότι ένα δυαδικό δένδρο ύψους h έχει 2^h φύλλα. Οπότε, για ένα διατακτικό αλγόριθμο που χρησιμοποιεί δυαδικό δένδρο, θα ισχύει:

$\boxed{2^h \geq n!} \Rightarrow h \geq \log n! \Rightarrow h \geq \log n^n \Rightarrow h \geq n \log n$ $\left(\log(n!) = \sum_{i=1}^{n} \log i \geq \int_1^n \log x \, dx = \Omega(n \log n)\right)$

3) Quicksort

• Βασίζεται στην αρχή divide-and-conquer: Για ένα πίνακα $A[p...r]$ εφαρμόζονται τα εξής:

Divide: Ο πίνακας $A[p...r]$ διαχωρίζεται σε δύο μη άδειους υποπίνακες $A[p...q]$ και $A[q+1...r]$ τέτοιους ώστε κάθε στοιχείο του $A[p...q]$ να είναι μικρότερο ή ίσο από κάθε στοιχείο του $A[q+1...r]$. Το στοιχείο q υπολογίζεται ως μέρος της συνάρτησης διαχωρισμού.

Conquer: Οι δύο υποπίνακες $A[p...q]$ και $A[q+1...r]$ διατάσσονται με αναδρομικές κλήσεις του quicksort.

Combine: Αφού ο κάθε πίνακας είναι διατεταγμένος τότε η συνένωσή τους θα μας δώσει τον ταξινομημένο πίνακα $A[p...r]$.

• Ο αλγόριθμος είναι ο εξής:

• Quicksort(A,p,r) ⇒ για πίνακα $A[p...r]$

```
if p<r
  then q ← partition(A,p,r)
       Quicksort(A,p,q)
       Quicksort(A,q+1,r)
```

Για ολόκληρο πίνακα η κλήση είναι Quicksort(A, 1, length[A])

• Partition(A,p,r)

```
x ← A[p]
i ← p-1
j ← r+1
while TRUE
  do repeat j ← j-1
       until A[j] ≤ x
     repeat i ← i+1
       until A[i] ≥ x
  if i<j
    then exchange A[i] ↔ A[j]
    else return j
```

3

π.χ. A[p...r]: 5 | 3 | 2 | 6 | 4 | 1 | 3 | 7 (i, j) ⇒ 5 | 3 | 2 | 6 | 4 | 1 | 3 | 7 (i, j) ⇒ 3 | 3 | 2 | 6 | 4 | 1 | 5 | 7 (i, j) ⇒ 3 | 3 | 2 | 6 | 4 | 1 | 5 | 7 (i, j) ⇒

⇒ A[p...q] | A[q+1...r]: 3 | 3 | 2 | 1 | 4 | 6 | 5 | 7 (return j, i)

Ο πίνακας έχει 8 στοιχεία, άρα $p=1$ και $r=8$ ($A[1...8]$). Μπαίνοντας στο Partition(A,1,8) έχουμε αρχικά ότι $x=A[1]=5$ και $i=0, j=9$.

Όταν το $j=7$ τότε το $A[j]=A[7]=3 \le 5$ και τότε το $i=1$, αφού $A[1]=5 \ge 5$. Οπότε τα 3 και 5 αντιμετατίθενται, αφού $i<j$ $(1<7)$. Η ίδια διαδικασία επαναλαμβάνεται έως ότου $j<i$ οπότε και επιστρέφουμε, καλώντας αναδρομικά τον αλγόριθμο για τα κομμάτια του πίνακα (υποπίνακες) $A[1...5]$ και $A[6...8]$.

• Πολυπλοκότητα χειρότερης περίπτωσης: Αν ορίσουμε σαν $QS(n)$ το μέγιστο πλήθος των συγκρίσεων που εκτελεί ο Quicksort σε έναν πίνακα μήκους n τότε ισχύει η παρακάτω ανισότητα:

$$QS(n) \le \underbrace{n+1}_{\text{για τον 1ο διαχωρισμό σε 2 ακολουθίες}} + \max_{1\le j\le n}\{QS(j-1)+QS(n-j)\}$$

, όπου ισχύει $QS(0)=QS(1)=0$ και επαγωγικά αποδεικνύεται ότι $\boxed{QS(n) \le \frac{(n+1)(n+2)}{2} - 3 = O(n^2)}$

• Πολυπλοκότητα μέσης περίπτωσης:

Έχουμε τις υποθέσεις:
• Όλα τα στοιχεία του πίνακα $A[1...n]$ είναι ανά δύο διαφορετικά
• Έχουμε ομοιόμορφη κατανομή, δηλαδή καθεμιά από τις $n!$ δυνατές μεταθέσεις των δεδομένων είναι ισοπίθανη.

(*) Έάν για τον πίνακα f έχουμε ότι το $f[1]=k$ τότε τα υποπροβλήματα μεγέθους $k-1$ και $n-k$ ικανοποιούνται πάλι τις προηγούμενες υποθέσεις. Για το υποπρόβλημα μεγέθους $k-1$ η απόδειξη είναι προφανής.

• Με τον 1ο διαμερισμό του quicksort παίρνουμε το σχηματισμό: $K = k\, I_1 \dots I_{k-1}\, J_{k+1} \dots J_n$ όπου $I_1 \dots I_{k-1}$ είναι μετάθεση των $1, \dots, k-1$ στοιχείων του πίνακα και $J_{k+1} \dots J_n$ είναι μετάθεση των $k+1, \dots, n$ αντίστοιχα. Αν είναι τ το πλήθος των εναλλάξεων στην προηγούμενη σειρά, δηλαδή μεταξύ των αριθμών $1, \dots, k-1$ και $k+1, \dots, n$, τότε εναλλάσσονται τ ζευγάρια. Οπότε, υπάρχουν:

$$\sum_{\tau \geq 0} \binom{k-1}{\tau} \cdot \binom{n-k}{\tau}$$ ακολουθίες που μπορούν να υπάρχουν στην ακολουθία K μετά το διαμερισμό.

Θεωρώντας, τώρα, ότι το μέσο πλήθος των συγκρίσεων που εκτελεί ο Quicksort σε ένα πίνακα μήκους n είναι $\overline{QS}(n)$, θα ισχύει $\overline{QS}(0) = \overline{QS}(1) = 0$, αφού απ' τις προηγούμενες υποθέσεις συμπεραίνουμε ότι $\text{prob}(S[1] = k) = 1/n$.

Επομένως, θα έχουμε:

$$\overline{QS}(n) = \underbrace{(n+1)}_{\text{για το διαμερισμό}} + \frac{1}{n} \cdot \sum_{s=1}^{n} \left[\overline{QS}(s-1) + \overline{QS}(n-s) \right] \Rightarrow \underline{\overline{QS}(n) = (n+1) + \frac{2}{n} \sum_{s=0}^{n-1} \overline{QS}(s)}$$

$$\begin{cases} n \cdot \overline{QS}(n) = n \cdot (n+1) + 2 \sum_{s=0}^{n-1} \overline{QS}(s) \\ (n+1) \cdot \overline{QS}(n+1) = (n+1)(n+2) + 2 \sum_{s=0}^{n} \overline{QS}(s) \end{cases} \xrightarrow{(-)} (n+1)\overline{QS}(n+1) - n\overline{QS}(n) = (n+1)(n+2) - n(n+1) + \underbrace{2\overline{QS}(n)}_{(\text{για } s=n)} \Rightarrow$$

$$\Rightarrow (n+1)\overline{QS}(n+1) = (n+2)\overline{QS}(n) + 2(n+1) \Rightarrow \overline{QS}(n+1) = 2 + \frac{n+2}{n+1} \cdot \overline{QS}(n) = 2 + \frac{n+2}{n+1}\left[2 + \frac{n+1}{n}\overline{QS}(n-1)\right] =$$

$$= 2 \cdot \left[1 + (n+2) \cdot \left(\frac{1}{n+1} + \frac{1}{n} + \dots + \frac{1}{2}\right)\right] \Rightarrow \overline{QS}(n) = 2 \cdot \left[1 + (n+1) \cdot \left(\frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{2}\right)\right] \Rightarrow$$

$$\Rightarrow \overline{QS}(n) = 2 + 2(n+1) \cdot \sum_{i=1}^{n} \frac{1}{i} - 2(n+1) = \underbrace{2(n+1) \cdot \sum_{i=1}^{n} \frac{1}{i} - 2n}$$

Όμως το $\sum_{i=1}^{n} \frac{1}{i}$ είναι αρμονικός αριθμός και ισχύει $\sum_{i=1}^{n} \frac{1}{i} \leq \int_1^n \frac{1}{x}\,dx = 1 + \ln(n)$

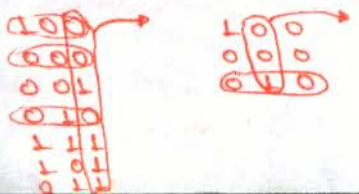
<u>Οπότε:</u> $\overline{QS}(n) = 2(n+1) + 2(n+1)\ln(n) - 2n \Rightarrow \underline{\overline{QS}(n) = 2(n+1)\log n + 2} \Rightarrow \boxed{\overline{QS}(n) = O(n \log n)}$

Άρα ο Quicksort διαταξινόμησε η στοιχεία σε χειρότερο χρόνο $O(n^2)$ και μέσο χρόνο $O(n \log n)$.

(*) Δίνεται ένα σύνολο αριθμών από 0 έως n, όπου το $n = 2^m - 1$, οι οποίοι μπορούν να αναπαρασταθούν με m bits. Οι αριθμοί αναπαριστάνται στο δυαδικό σύστημα αρίθμησης. Ένας απ' αυτούς λείπει. Να βρεθεί ο αριθμός σε γραμμικό χρόνο.

• Κάθε αριθμός του συνόλου αναπαριστάται με m bits: $x = \underset{MSB}{x_1} x_2 \dots \underset{LSB}{x_m}$, όπου το κάθε ψηφίο $x_i \in \{0,1\}$. Ο χρόνος που απαιτείται για να προσπελασθούν τα ψηφία είναι $O(m)$ και για όλους τους αριθμούς είναι $O(n \cdot m)$, ενώ αν έχουμε λογαριθμικό μέτρο θα είναι $\underline{O(n \cdot \log n)}$.

Μπορούμε να εφαρμόσουμε μια από έξυπνη ιδέα που συναντάται στην αρχή divide and conquer. Αν π.χ. θεωρήσουμε ότι $m = 3$ τότε το σύνολο των αριθμών είναι $n = 2^3 - 1 = 7$, άρα από 0...7. Απ' αυτούς λείπει ένας, τον οποίο και θα βρούμε:



Ξεκινάμε με τα LSB's και ομαδοποιούμε τους άσσους με τα μηδενικά. Βλέπουμε ότι λείπει ένα μηδενικό. Άρα στην επόμενη φάση παίρνουμε τους αριθμούς με 0 στο τέλος, ενώ ένα 0 θα προστεθεί στο τελευταίο bit του αριθμού που λείπει. Στη 2^n φάση βλέπουμε ότι λείπει ένας 1, οπότε συνεχίζοντας με τον ίδιο τρόπο έχουμε τον αριθμό (110)

Επομένως, η πολυπλοκότητα θα είναι $O(n)$ στο 1ο βήμα, $O(\frac{n}{2})$ στο 2ο, $O(\frac{n}{4})$ στο 3ο .... $O(1)$ στο τελευταίο (2^m-1-οστό)

External sorting

μέγεθος του block

- Θεωρούμε ότι διαθέτουμε ένα file μεγέθους N, μια κύρια μνήμη μεγέθους M και 2p βοηθητικές ταινίες (external devices). Τελικά, στην έξοδο θα έχουμε το file διατεταγμένο.
- Αρχικά, διαχωρίζουμε τις βοηθητικές ταινίες σε p-input και p-output ταινίες
- Στην κύρια μνήμη παράγονται διατεταγμένα υπο-files μεγέθους M.
- Αναμιγνύουμε τα διατεταγμένα υπο-files και αποθηκεύουμε το αποτέλεσμα εναλλάξ στις input και output ταινίες μέχρι να διατάξουμε το αρχικό file μεγέθους N.

π.χ. input: A SORTING AND MERGING EXAMPLE

Θεωρούμε ότι έχουμε 3 ταινίες εισόδου και εξόδου και τα υπο-files είναι μεγέθους M=3. (κρατούνται τοπολύ 3 γράμματα)

Αρχικά, χωρίζουμε το input-file σε 3-δες (M=3) στοιχείων και τα διατάσσουμε:

AOS	DMN	AEX	1η
IRT	EGR	LMP	2η
AGN	GIN	E	3η

1ο block

χρησιμοποιούν 3 ταινίες εισόδου (p=3 input devices) 1ο βήμα

Έπειτα, παίρνουμε τα πρώτα στοιχεία του 1ου block της κάθε ταινίας και τα διατάσσουμε. Αυτά τοποθετούνται ανα 9-δες (διατεταγμένες) σε 3-output ταινίες:

AAGINORST	1η	
DEGGINNNR	2η	
AEELMPX	3η	

χρησιμοποιούν 3 ταινίες εξόδου. 2ο βήμα

Τελικά, παίρνουμε μια ταινία στην οποία διατάσσουμε ξανά τα στοιχεία των 3 ταινιών εξόδου (από το μικρότερο στο μεγαλύτερο).

AAADEEEGGG...

- Στο 1ο βήμα έχουμε [redacted] $\lceil N/M \rceil$ blocks, όπου M είναι το μέγεθος της μνήμης και N είναι ο αριθμός των στοιχείων (γραμμάτων) σε κάθε block. (στο παράδειγμα έχουμε M=3 και N=9) → είναι το συνολικό $\lceil N/p \rceil$ (= $\lceil 25/3 \rceil = 9$)
- Έστω k το μέγιστο πλήθος των βημάτων ανάμιξης. Στο i-οστό βήμα παράγονται $p^{-i} \cdot (\frac{N}{M})$ blocks (όπου N είναι το συνολικό μέγεθος του file)
- Ο αλγόριθμος τελειώνει όταν έχουμε μόνο 1 block, δηλαδή $(\frac{N}{M}) \cdot \frac{1}{p^i} = 1 \overset{i=k}{\Rightarrow} \boxed{k = \log_p(\frac{N}{M})}$

(στο προηγούμενο παράδειγμα έχουμε $\underbrace{N=25}_{\text{συνολικό } N}$ και M=3, άρα $\lceil N/M \rceil = 9$ και τότε $k = \log_3 3^2 \Rightarrow k=2$ βήματα).

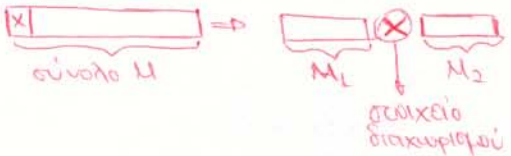
ΓΡΑΜΜΙΚΟΣ MEDIAN ΑΛΓΟΡΙΘΜΟΣ

- Δίνεται μια ακολουθία από ανά δύο διαφορετικά στοιχεία και ένας ακέραιος i με $1 \le i \le n$ (n είναι τα στοιχεία της ακολουθίας) και θέλουμε να βρούμε το i-οστό μικρότερο στοιχείο x_j τέτοιο ώστε να υπάρχουν i-1 στοιχεία x_k με $x_k < x_j$ και n-i στοιχεία x_λ με $x_\lambda > x_j$.
- Αυτό το $n/2$-οστό μικρότερο στοιχείο ονομάζεται median
- Μια πρώτη λύση είναι να διατάξουμε την ακολουθία και να απαριθμήσουμε έτσι ώστε να βρούμε το ζητούμενο στοιχείο. Κάτι τέτοιο όμως απαιτεί χρόνο $O(n\log n)$ στη χειρότερη και μέση περίπτωση.
- Ο αλγόριθμος find έχει γραμμικό χρόνο στη μέση περίπτωση ($O(n)$) και $O(n^2)$ στη χειρότερη
" " Select " " " " " " ($O(n)$) " $O(n)$ " "
- Η διαφορά του find με τον Quicksort στη μέση περίπτωση, αφού και οι δύο βασίζονται στην αρχή divide and conquer, είναι στο ότι ο Quicksort λύνει και τα 2 υποπροβλήματα, ενώ ο find

σε ένα από αυτά. Σαν χειρότερη, δηλ. περίπτωση και ο find δεν απορρίπτει τίποτε.

- Ο αλγόριθμος find διαχωρίζει το σύνολο M με βάση ένα οποιοδήποτε στοιχείο i του M σε δύο υποσύνολα, το M_1 και το M_2. Αν $M_1 < i$, τότε το i-οστό μικρότερο στοιχείο δεν είναι στο υποσύνολο M_1 και είναι στο M_2. Αν $M_1 = i$ τότε το στοιχείο x είναι το στοιχείο διαχωρισμού, δηλαδή το i-οστό μικρότερο στοιχείο. Αν $M_1 > i$ τότε το i-οστό μικρότερο στοιχείο θα είναι στο M_1.

π.χ.:



Έχω την ακολουθία 5, 6, 4, 3, 8, 7

ως στοιχείο διαχωρισμού έχω το $5 = x$ οπότε το M_1 θα αποτελείται απ' τα στοιχεία που είναι μικρότερα του 5 και το M_2 απ' τα μεγαλύτερα.

$$\underbrace{(4, 3)}_{M_1}\ (5)\ \underbrace{(6, 8, 7)}_{M_2}$$

Ψάχνουμε το i-οστό μικρότερο στοιχείο της ακολουθίας. Αν π.χ. $i=4$, το $4^{ο}$ μικρότερο στοιχείο της ακολουθίας είναι το 6 (στην παρούσα περίπτωση το $4^{ο}$ μικρότερο στοιχείο ανήκει στο υποσύνολο M_2). Οπότε, τα M_1 και x πετάγονται και ψάχνεται το M_2. Φυσικά, προχωρώντας στον αλγόριθμο και καλώντας τον αναδρομικά για το M_2, θα έχουμε ως νέο x το 6 και τώρα θα ψάχνουμε το $i-|M_1|-1$ στοιχείο, δηλαδή το $i-2-1 = 4-3 = 1^{ο}$ στοιχείο, το οποίο τυχαίνει να είναι ίδιο με το στοιχείο διαχωρισμού x. Οπότε το i-οστο ($4^{ο}$) στοιχείο που είναι μικρότερο στην ακολουθία είναι το 6.

- Η πολυπλοκότητα χειρότερης περίπτωσης για τον find είναι $O(n^2)$ όπως φαίνεται, αν πάρουμε $i=1$ και $|M_1| = |M|-1$.

- Πολυπλοκότητα μέσης περίπτωσης: υποθέτουμε ότι τα στοιχεία του M είναι ανά δύο διαφορετικά και κάθε με κάποια τύχη είναι ισοπίθανη. Θεωρώντας ότι το στοιχείο s που επιλέγεται ως στοιχείο διαχωρισμού επιλέγεται με πιθανότητα $p = \frac{1}{|M|}$ (όπου $|M| = n$)

Τότε: $T(n,i) \le c \cdot n + \frac{1}{n}\left[\sum_{k=0}^{i-2} \underbrace{T(n-k-1, i-k-1)}_{\text{μέσος χρόνος για find}(M_2, i-|M_1|-1) \text{ εφόσον } |M_1|<i-1} + \sum_{k=i}^{n-1} \underbrace{T(k,i)}_{\text{μέσος χρόνος για } |M_2|>i-1 \text{ δηλαδή για find}(M_1,i)}\right] \Rightarrow T(n,i) \le cn + \frac{1}{n}\max_i\left[\sum_{k=0}^{i-2} T(n-k-1) + \sum_{k=i}^{n-1} T(k)\right]$

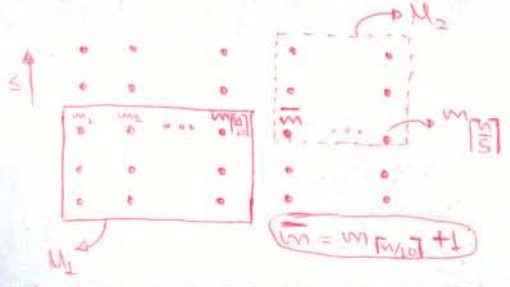
$(T(1) = c_1$ σταθ.$)$

$p = \frac{1}{|M|}$

αφού $T(n) = \max_i T(n,i)$

Επαγωγικά αποδεικνύεται ότι $T(n) \le 4cn$

- Για τον αλγόριθμο Select, αν έχουμε μεγάλο σύνολο M (με $n \ge 100$) τότε χωρίζουμε το input σε 5-άδες. Έπειτα, διατάσσουμε τις πενταάδες και βρίσκουμε το median κάθε πενταάδας. Τέλος, παραλαμβάνουμε όλους τους medians και βρίσκουμε το median των medians.



Διαχωρίζουμε το M σε $\lceil n/5 \rceil$ υποσύνολα με 5 στοιχεία το καθένα.

χωρίς βλάβη της γενικότητας θεωρούμε ότι $m_1, \dots, m_{\lceil n/10 \rceil} < \bar{m} \le m_{\lceil n/10 \rceil + 1}, \dots, m_{\lceil n/5 \rceil}$. Με βάση το $\bar{m}$ θα εφαρμόσουμε αναδρομικά τον select στο M_1 ή στο M_2.

Είναι: $|M_1|, |M_2| \le 8n/11$ και κάθε ορθογώνιο έχει $\frac{3n}{10}$ στοιχεία.

Αν $T(n)$ είναι ο μέγιστος χρόνος για τον Select τότε υπάρχουν σταθερές a, b οι οποίες:

$$\begin{cases} T(n) \le a \cdot n, & n \le 100 \\ T(n) \le T\left(\frac{24n}{100}\right) + T\left(\frac{8n}{11}\right) + bn, & n > 100 \end{cases}$$

Επαγωγικά αποδεικνύεται και στις 2 περιπτώσεις ότι $T(n) \le cn$

Άρα ο Select εκτελείται σε γραμμικό χρόνο.

ΣΥΝΟΠΤΙΚΕΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Οι συνοπτικές δομές δεδομένων αποθηκεύουν ένα σύνολο M, όπου $M \subseteq U$ και U σύμπαν με $M = \{x_1, \dots x_n\}$, και χρειάζονται χώρο $c_1 n + c_2$ όπου c_1, c_2 σταθερές. Το $c_2 = 1$ και το c_1 είναι μικρό.

• Αν το σύνολο M είναι διατεταγμένο και αποθηκευμένο σε έναν πίνακα $S = S[1 \dots n]$ με $S[i] = x_i$ και $x_1 < x_2 < \dots < x_i < \dots < x_n$, τότε ο παραδοσιακός αλγόριθμος υλοποιεί την πράξη search(x) κατά την οποία η πληροφορία των στοιχείων συλλέγεται μόνο με συγκρίσεις:

```
(left=1, right=n)
next = τυχαίος αριθμός από [left...right];
while x≠S[next] and right>left do
    if x<S[next] then right = next - 1
    else left = next+1
    next = τυχαίος αριθμός από [left...right];
if x=S[next] then επιτυχία
else αποτυχία
```

Αν ισχύει:
• next = left ⇒ έχουμε linear search
• $next = \left\lfloor \frac{right+left}{2} \right\rfloor$ ⇒ έχουμε binary search
• $next = \left\lfloor \frac{x - S[left]}{S[right] - S[left]} \cdot (right - left) \right\rfloor + left$ ⇒ έχουμε interpolation search

ισχύει για uniform κατανομή, δηλαδή όλα τα στοιχεία είναι ισοπίθανα για να εμφανισθούν

• Binary interpolation search: Για να βρούμε ένα στοιχείο a στο διατεταγμένο σύνολο $M = \{x_1, \dots x_n\}$ με $x_1 < x_2 < \dots < x_n$ κάνουμε τα εξής:
- Αρχικά, εφαρμόζουμε interpolation search για το a, κάνοντας ευαίσθησία στο πού κοντά μπορεί να βρίσκεται
- Ψάχνουμε σε $\lfloor \sqrt{n} \rfloor$ αποστάσεις κάνοντας άλματα μεγέθους $\sqrt{n}$ έτσι ώστε να φτάσουμε σε ένα στοιχείο $z = x_k > a$
- Εφαρμόζουμε την ίδια διαδικασία στο υπο-file μεγέθους $\lfloor \sqrt{n} \rfloor$ στο οποίο περιέχεται το a.

• Πολυπλοκότητα χειρότερης περίπτωσης: το μέγιστο πλήθος των συγκρίσεων είναι $\sum_{i=1}^{\infty} n^{1/2^i} = \sqrt{n} + \sqrt{\sqrt{n}} + \sqrt{\sqrt{\sqrt{n}}} + \dots = \boxed{O(\sqrt{n})}$

• Πολυπλοκότητα μέσης περίπτωσης: Θεωρούμε ότι τα στοιχεία x_i που επιλέγονται από το σύνολο υπακούν σε uniform κατανομή. Έστω τότε C ο μέσος αριθμός των συγκρίσεων που απαιτείται για να βρούμε το κατάλληλο υπο-file μεγέθους $\sqrt{n}$ και p_i η πιθανότητα του να χρειάζονται τουλάχιστον i συγκρίσεις για να βρεθεί αυτό το υπο-file. Τότε:

$$C = \sum_{i \geq 1} i(p_i - p_{i+1}) = \sum_{i \geq 1} p_i \quad \text{με } p_1 = p_2 = 1$$, αφού πάντα εκτελούνται τουλάχιστον 2 συγκρίσεις

Ισχύει: $p_i \leq Prob\{|\text{θέση του } a - next| \geq (i-2)\sqrt{n}\}$ για $i \geq 3$

Chebyschev inequality: $Prob(|X - \mu| \geq \varepsilon) \leq \sigma^2/\varepsilon^2$, όπου ε σταθερά, μ η μέση τιμή και σ η απόκλιση

Αν x είναι τυχαία μεταβλητή που παίρνει τιμές μικρότερες του a, τότε θα δείξω ότι x = θέση του a, $\mu = next$ και $\sigma^2/\varepsilon^2 = (i-2)\sqrt{n}$

Κάνω πειράματα μεγέθους $\sqrt{n}$ στο $[x_0, x_{n+1}]$. Αν κάνω j τέτοια πειράματα τότε η πιθανότητα γι' αυτό θα είναι $q_j = P(x = j)$. Η πιθανότητα να επιλέξω ένα στοιχείο μικρότερο του a είναι:

$p = \frac{a - x_0}{x_{n+1} - x_0}$. Τότε: $q_j = \binom{n}{j} \cdot p^j \cdot (1-p)^{n-j}$, δηλαδή έχω binomial κατανομή με $\mu = p \cdot n$ και $\sigma^2 = p(1-p)n \leq \frac{n}{4}$ διακυμανσή

Άρα $\mu = next$ και θα είναι:

$$p_i \leq Prob\{|x - \mu| \geq (i-2)\sqrt{n}\} \leq \frac{p(1-p)n}{((i-2)\sqrt{n})^2} = \frac{p(1-p)}{(i-2)^2} \leq \frac{1}{4(i-2)^2}$$

(για $p = \frac{1}{2} \Rightarrow \sigma^2 = \frac{n}{4}$)

Οπότε: $\boxed{C \leq \frac{1+1}{2} + \sum_{i \geq 3} \frac{1}{4(i-2)^2} \simeq 2{,}4}$

Αν, λοιπόν, είναι $\overline{T}(n)$ = μέσο πλήθος των συγκρίσεων τότε $\overline{T}(n) \leq C + \overline{T}(\sqrt{n})$, για $n \geq 3$ $\xrightarrow[\overline{T}(2) \leq 2]{\overline{T}(1) \leq 1}$ $\overline{T}(n) \leq 2 + 2{,}4 \log\log n$

$\Rightarrow \boxed{\overline{T}(n) = O(\log\log n)}$ Άρα ο μέσος χρόνος για το BTS είναι $O(\log\log n)$

• Για να διορθώσουμε την πολυπλοκότητα χειρότερης περίπτωσης του BTS εφαρμόσαμε το ευθύγραμμο και δυϊκό ψάξιμο. Αντί δηλαδή να εφαρμόσουμε γραμμικό ψάξιμο για να βρούμε ένα i τέτοιο ώστε $i \cdot \sqrt{n}$ βήματα να έχουμε βρεί το υπο-file που βρίσκεται το a, κάνουμε ευθύγραμμο ψάξιμο βήματα $2^0, 2^1, \dots, 2^i$ ώστε να βρούμε ένα j τέτοιο ώστε με $2^j \cdot \sqrt{n}$ βήματα να βρούμε το υπο-file με το a. (ισχύει $j < \log i$)

• Έστω ότι βρίσκουμε ένα υπο-file για το οποίο ισχύει $\underline{S[next + 2^{j-1}\sqrt{n}] < a \le S[next + 2^j \cdot \sqrt{n}]}$. Τότε σ'αυτό το υπο-file εφαρμόζουμε δυϊκό ψάξιμο. <u>Δηλαδή:</u>

Το υπο-file έχει μέγεθος $2^j\sqrt{n} - 2^{j-1}\sqrt{n} = \underline{2^{j-1}\sqrt{n}}$. Μπορώ, λοιπόν, να το χωρίσω σε $\sqrt{n}$ υπο-ομάδες και να έχω 2^{j-1} υπο-ομάδες. Αρά, δηλαδή, βρίσκω το διάστημα $2^{j-1}\sqrt{n}$ εφαρμόζω μέσα σε αυτό <u>το γραμμικό ψάξιμο</u> για να βρώ το $\sqrt{n}$ διάστημα στο οποίο περιέχεται το στοιχείο μου. Έπειτα, σ'αυτό το διάστημα μεγέθους $\sqrt{n}$ θα εφαρμόσω πάλι την ίδια μέθοδο, δηλαδή τα ευθύγραμμα βήματα και το γραμμικό ψάξιμο. Το δυϊκό ψάξιμο έγκειται στο γεγονός ότι μόλις βρεθεί το υπο-file τότε σ'αυτό ψάχνουμε δυϊκά, δηλαδή ψάχνουμε δυϊκά μεταξύ των υπο-πινάκων $2^{j-1}\sqrt{n}, (2^{j-1}+1)\sqrt{n}, \dots, 2^j\sqrt{n}$. Με άλλα λόγια τα διαστήματα που δεν χρειάζονται τα πετάω. (αν π.χ. έχω 8 υπο-ομάδες (j=4) τις χωρίζω σε 4 και 4 και κοιτάω που είναι το a...)

• Το ευθύγραμμο ψάξιμο χρειάζεται χρόνο $O(\log i)$ και το δυϊκό ψάξιμο χρειάζεται $\log 2^{j-1} = O(j-1) = \underline{O(\log i)}$. Άρα ο χειρότερος χρόνος είναι $O(\log \sqrt{n})$ διότι $i \le \sqrt{n}$. Ο συνολικός χειρότερος χρόνος είναι:

$$\log n^{1/2} + \log n^{1/4} + \dots = (1/2 + 1/4 + \dots) \cdot \log n = \underline{\underline{O(\log n)}}$$

ΕΚΤΕΝΕΙΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Οι εκτενείς δομές δεδομένων αποθηκεύουν ένα σύνολο M, με $|M|=n$ και χρειάζονται χώρο $c_1 \cdot n + c_2$ όπου c_1, c_2 σταθερές με $c_1 \neq 1$ και c_1, c_2 μικρές, ιμπρίως $c_1 \in \{3,4,5\}$

Διακρίνονται στις απλές που υλοποιούνται με χρήση δεικτών και στις υβριδικές που χρησιμοποιούν και αλλαγές.

• <u>Ισοζυγισμένα δέντρα</u>: Ένα δένδρο εύρεσης T για το σύνολο S με $|S|=n$, ονομάζεται ισοζυγισμένο όταν ισχύει: $h(T) = c_1 \log n + c_2$, c_1, c_2 σταθερές και $h(T)$ το ύψος του T.

⊛ Τα ισοζυγισμένα δένδρα ορίζονται βάσει ενός κριτηρίου ζύγισης και εκτελούνε το ψάξιμο, ένθεση, αφόρεση σε $O(\log n)$ χρόνο. (στη χ.π.).

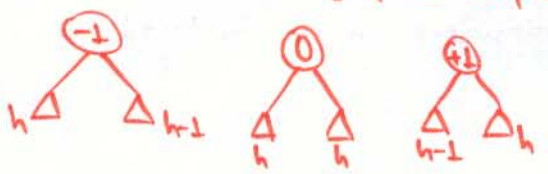
α) <u>Υποζυγισμένα δέντρα</u>: ισοζυγίζουμε βάσει του <u>ύψους</u> των υποδένδρων κάθε κόμβου.

1) AVL-δένδρο

• Έστω $hb(v)$ η <u>υποζύγιση</u> του κόμβου v, με $hb(v) = \text{ύψος}(\text{rson}(v)) - \text{ύψος}(\text{lson}(v))$ όπου rson ο δεξής γιος και lson ο αριστερός

• Ένα δυϊκό δένδρο T λέγεται AVL όταν για κάθε κόμβο $v \in T$ ισχύει $|hb(v)| \leq 1$

• Υπάρχουν 3 κατηγορίες κόμβων ανάλογα με την υποζύγιση:



Ο κόμβος με ισοζύγιση $\emptyset$ είναι ζυγισμένος και οι άλλοι μη ζυγισμένοι.

• Τα δένδρα-Fibonacci ορίζονται ως εξής ($F_0, F_1, F_2, \ldots$): $F_0 =$ (1 φύλλο)

Το F_h δένδρο έχει ύψος h

$F_1 =$

$F_h =$ F_{h-1} F_{h-2}

⊛ Το F_h περιέχει ακριβώς $\frac{a^{h+2}-b^{h+2}}{\sqrt{5}}$ φύλλα, με $a = \frac{1+\sqrt{5}}{2}$, $b = \frac{-1+\sqrt{5}}{2}$ και ο αριθμός φύλλων δίνεται απ' την ακολουθία $|F_h| = |F_{h-1}| + |F_{h-2}|$ με $F_0 = 1, F_1 = 2$.

⊛ Για AVL με n φύλλα και ύψος h ισχύει: $|F_h| \leq n \leq$ πλήρες δυϊκό με ύψος h (πλέον ισορροπημένο AVL) $\Rightarrow$

$\Rightarrow \frac{a^{h+2}-b^{h+2}}{\sqrt{5}} \leq n \leq 2^h$. Είναι: $n \leq 2^h \Rightarrow \underline{\log n \leq h}$

<u>Όμως</u>: $\frac{a^{h+2}-b^{h+2}}{\sqrt{5}} \leq n \xrightarrow{b<1} n \geq \frac{a^{h+2}-1}{\sqrt{5}} \Rightarrow a^{h+2} \leq \sqrt{5}n+1 \Rightarrow \underbrace{a^{h+2} \leq \sqrt{5}(n+1)}_{\text{(φράσσεται από λογάριθμο).}} \Rightarrow$

$\Rightarrow \ldots \Rightarrow \underline{h \leq 1{,}44 \log(n+1) - 0{,}3277}$

$\Rightarrow$ Το ψάξιμο στο AVL κοστίζει $O(\log n)$

⊛ Οι εξτρεμιστικές καταστάσεις για το AVL είναι το F_h και το πλήρες δυϊκό δένδρο (όλα με ύψος h)

• **Ένθεση - απόσβεση στο AVL:**

→ Έστω μονοπάτι $p = (v_0, v_1, \dots, v_k)$ από τη ρίζα v_0 έως το φύλλο v_k. Αν για τον κόμβο v_i ισχύει $hb(v_i) = hb(v_{i+1}) = \dots = hb(v_{k-1}) = 0$ τότε ο κόμβος v_{i-1} ονομάζεται κρίσιμος κόμβος ενθέσεως (ΚΚΕ).

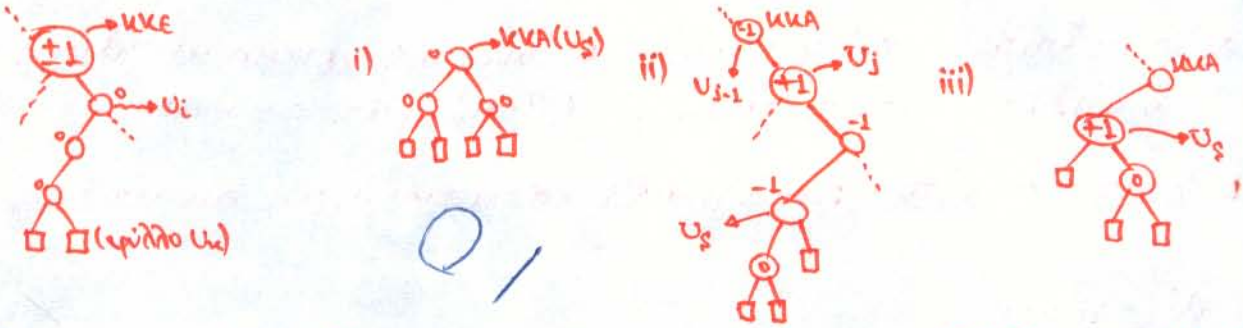
→ Έστω v_s ένας κόμβος στο μονοπάτι εύρεσης με $h(v_s) = 2$. Τότε ο ΚΚΑ είναι:

i) ο v_s όταν $hb(v_s) = 0$

ii) ο v_{j-1} όπου $j \le t < s$ για $hb(v_t) = \begin{cases} +1, & \text{όταν } rson(v_t) \in \text{στο μονοπάτι εύρεσης} \\ -1, & \text{.. } lson(v_t) \in \text{ .. .. ..} \end{cases}$

iii) ο $\pi(v_s)$ όταν δεν υπάρχει t που να ικανοποιεί την προηγούμενη σχέση.

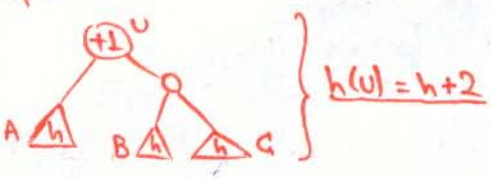
π.χ.



(*) Μετά από κάθε ένθεση ή απόσβεση το ύψος του υποδένδρου με ρίζα ΚΚΕ ή ΚΚΑ μεταβάλλεται κατά +1 ή -1 (απόσβεση). Πάνω στο κρίσιμο μονοπάτι $(v_i, \dots, v_k)$ εκτελούνται οι αλλαγές ύψους. Γενικά, οι ενθέσεις ή αποσβέσεις αλλάζουν το ύψος του υποδένδρου και έτσι μπορεί να παρουσιασθεί διαφορά +2 ή -2 και γι' αυτό εκτελούνται επανασυμμετρικές πράξεις.

→ **Ένθεση**

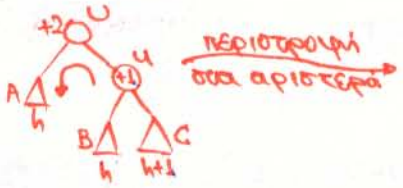
Έστω v ο ΚΚΕ και χωρίς βλάβη της γενικότητας είναι $hb(v) = +1$ και έχουμε το δένδρο:

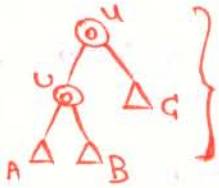


$h(v) = h+2$

• Κατόπιν ενθέσεως στο A έχουμε απορρόφηση (absorption): το ύψος του •A γίνεται h+1, οπότε θέτουμε $hb(v) = 0$

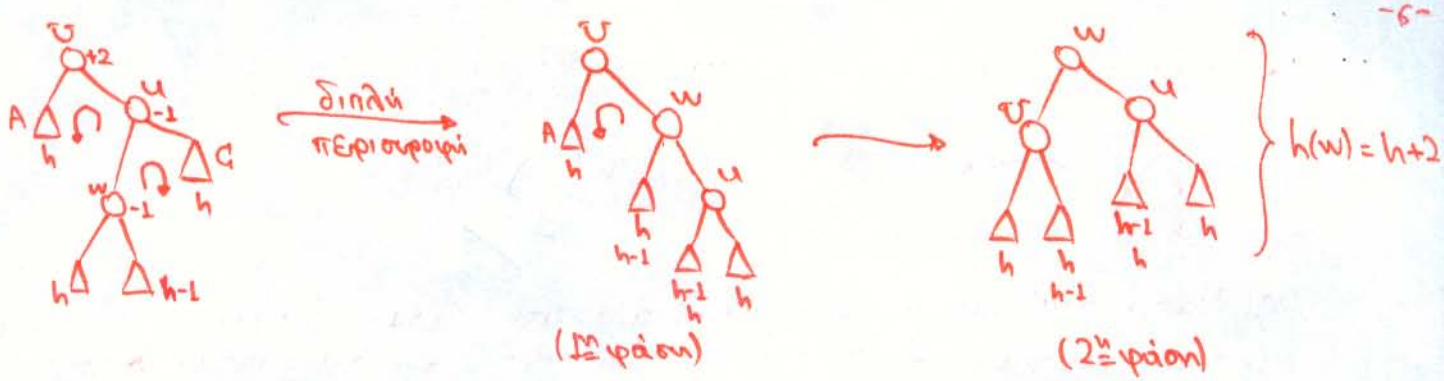
• Κατόπιν ενθέσεως στο C έχουμε απλή περιστροφή (rotation): το ύψος του C γίνεται h+1 οπότε ο v έχει βάρος +2 (+1+1) και ο u +1





$h(u) = h+2$

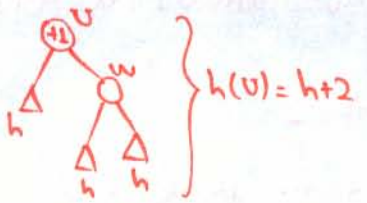
• Κατόπιν ενθέσεως στο B έχουμε διπλή περιστροφή (double rotation): Θεωρούμε ότι η ένθεση γίνεται στο αριστερό υποδένδρο του B, άρα $h(l[T_B]) = (h-1)+1 = h$ και το δεξί υποδένδρο του B έχει ύψος h-1. Οπότε:



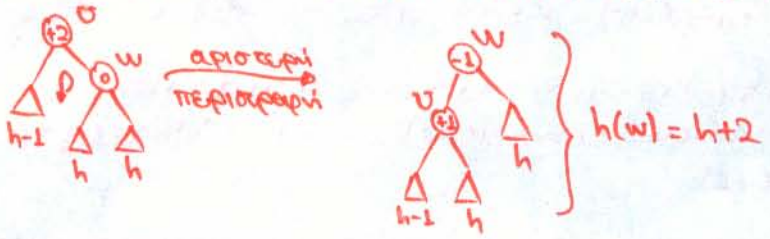
(*) Μετά από 1 ένθεση χρειάζονται O(1) rotations ή double rotations (το αλγεβρικό μήκος των υπόλοιπων μονοπατιών μιας ανοδικής η ενθέσεων είναι 2,618n => καταναλώνεται 2,618).

⟹ Απόθεση

Έστω v ο κΚΑ και χωρίς βλάβη της γενικότητας είναι hb(v) = +1. Θα μελετήσουμε την απόθεση στο αριστερό υποδένδρο με αλλαγή ύψους αυτού του υποδένδρου.

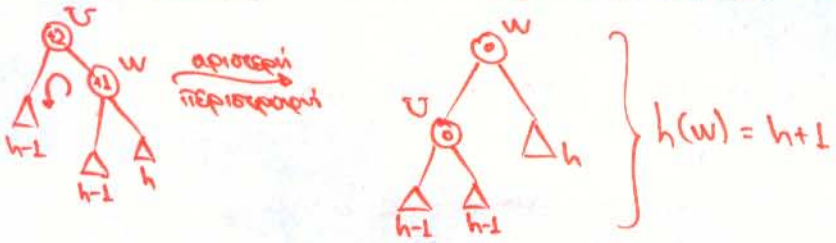


• Τελική περιστροφή (terminated rotation): αφαιρώντας από το αριστερό υποδένδρο, το ύψος του γίνεται h-1 οπότε hb(v) = +2 και hb(w) = 0:

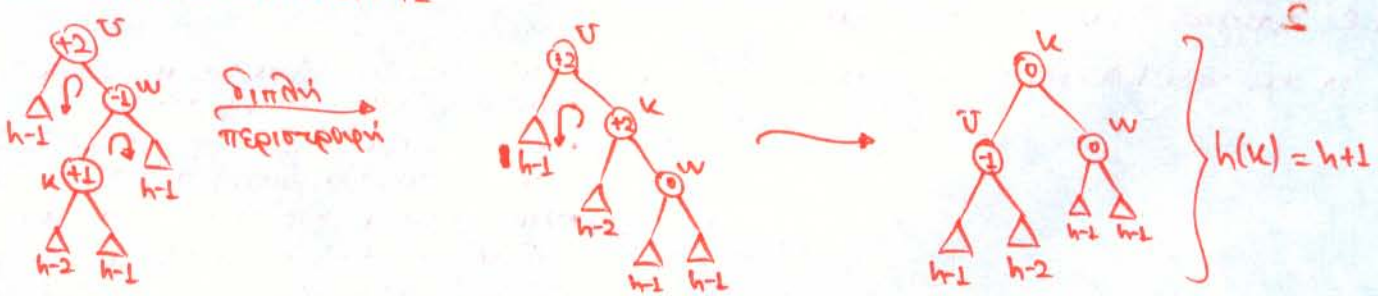


• Συνεχιζόμενη περιστροφή (propagated rotation): αφαιρώντας από το αριστερό υποδένδρο του w, το ύψος του γίνεται h-1 και τότε hb(v) = +2 και hb(w) = +1:

(h+1-(h-1))



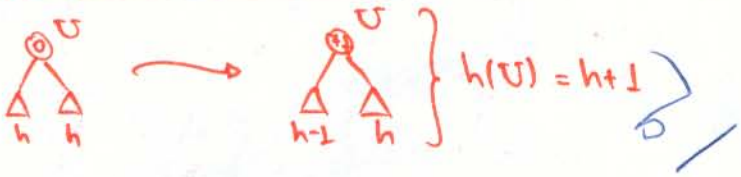
• Συνεχιζόμενη διπλοπεριστροφή (propagated double rotation): αφαιρώντας από το αριστερό υποδένδρο του k, το ύψος του γίνεται h-2 και τότε hb(k) = +1, hb(w) = -1 και hb(v) = (h+1)-(h-1) = +2



(*) Για hb(k) = 0 τότε στο τέλος έχω hb(k) = 0, hb(v) = 0, hb(w) = 0

Για hb(k) = -1 " " " " hb(k) = 0, hb(v) = 0, hb(w) = +1

• Απορρόφηση (absorption): όταν το $hb(U)=0$, δηλαδή:



⊛ Μετά από μια απόσβεση χρειάζονται $O(\log n)$ rotations ή double rotations. (ο αναμενόμενο κόστος για 1 απόσβεση σε μια ακολουθία n αποσβέσεων είναι 1,618)

2) Red-black δένδρο ή BB-δένδρο

- κάθε κόμβος του δένδρου έχει ακριβώς 2 παιδιά (πλήρες δυϊκό δένδρο ⇒ αδύνατες οι αλυσίδες κόμβων)
- τα στοιχεία αποθηκεύονται στα φύλλα
- σε κάθε κόμβο υ αντιστοιχεί ένας ακέραιος βαθμός (rank) που συμβολίζεται με $r(U)$
- αν το υ είναι φύλλο τότε $r(U)=0$ και $r(p(U))=1$
- ισχύει $r(U) \le r(p(U)) \le r(U)+1$ (το $r(U)$ αυξάνεται κατά 1 ή μένει ανέπαφο από πάνω προς τα κάτω)
- ισχύει $r(U) < r(p(p(U)))$

⊛ Αν έχουμε $rank(p(U)) - rank(U) = bit(U)$ τότε το $bit(U) \in \{0,1\}$

Στο red-black δένδρο έχουμε τις απλές πράξεις επανατάξεις στις οποίες έχουμε αλλαγή χρώματος (promotions, demotions) και τις σύνθετες πράξεις ή δομικές (rotations ή d-rotations)

- Κάθε ένθεση ή απόσβεση απαιτεί $O(\log n)$ απλές πράξεις (promotions-demotions) και $O(1)$ δομικές (rotations)
- Μια ακολουθία από $O(n)$ ενθέσεις ή αποσβέσεις σε ένα αρχικά άδειο BB δένδρο απαιτεί $O(n)$ promotions ή demotions.

⊙ Ένας κόμβος σε ύψος t θα πάθει μέρος σε μια επανατάξη με πιθανότητα $(\frac{1}{2})^t$

⊙ Είναι δυνατή η top-down επανατάξη

- Ένα bit είναι αναγκαίο για την αποθήκευση της τάξης.

→ είναι πολύ βασικές ιδιότητες σε εφαρμογές παραλληλισμού και πολυδιάστατων δομών δεδομένων

Όσον αφορά το χρώμα των ακμών, αυτές θα είναι κόκκινες (τιμή 0) αν οι διαφορές των ranks των 2 κόμβων της ακμής είναι μηδενικές και μαύρες (τιμή 1) αν οι διαφορές των ranks είναι $\neq 0 (=1)$. (αφού $r(U) < r(p(p(U)))$ δεν μπορούμε να έχουμε **2** ακμές κόκκινες στη σειρά)

Για το χρωματισμό των κόμβων έχουμε:
- κάθε κόμβος είναι κόκκινος ή μαύρος
- κάθε φύλλο είναι μαύρο
- αν ένας κόμβος είναι κόκκινος τότε και τα 2 παιδιά του θα είναι μαύρα.
- κάθε απλό μονοπάτι από ένα κόμβο προς ένα φύλλο που είναι απόγονός του περιέχει τον ίδιο αριθμό από μαύρους κόμβους

β) Βαροζυγισμένα δένδρα: ισοζυγίζουμε βάσει του πλήθους των κόμβων των υποδένδρων ενός κόμβου.

1) BB[a]-δένδρο

- Έστω T ένα δυϊκό δένδρο με αριστερό υπόδενδρο T_L και δεξί υπόδενδρο T_R. Τότε, ορίζουμε ως $\rho(T) = \frac{|T_L|}{|T|}$ τη βαροζύγιση της ρίζας του T, όπου $|T|$ είναι ο αριθμός των φύλλων του δένδρου.
- Ένα δένδρο T έχει βαροζύγιση a όταν ∀ υπόδενδρο T' του T ισχύει: $a \le \rho(T') \le 1-a$
- Ένα BB[a]-δένδρο είναι το σύνολο όλων των δένδρων T με περιορισμένης βαροζύγισης a
- Όταν ισχύει $a \le \rho(T')$ έχουμε πολλά στοιχεία στο αριστερό (άρα βαδίζουμε δεξιά)
  " " $\rho(T') \le 1-a$ " " " " δεξιά (" " αριστερά)

(*) Έστω δένδρο T το οποίο είναι BB[a]-δένδρο. Τότε: $h(T) \le 1 + \frac{\log(n+1)-1}{\log\frac{1}{1-a}} = O(\log n)$

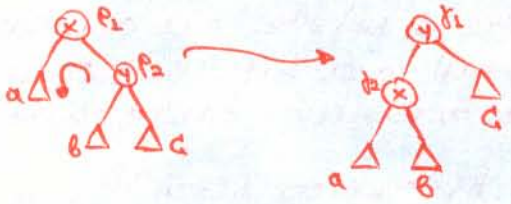
(το ύψος του T φράσσεται από λογάριθμο)

13

Όταν οι ενθέσεις ή οι αποβέσεις επηρεάζουν τις βαροζυγίσεις των κόμβων και τις βγάλουν έξω από το διάστημα $[a, 1-a]$ τότε μπορούμε να επαναφέρουμε τη ζύγιση στο επιτρεπτό διάστημα με απλές ή διπλές περιστροφές.

→ απλή περιστροφή:

$\rho_1, \rho_2, \gamma_1, \gamma_2$ είναι οι βαροζυγίσεις των αντίστοιχων κόμβων

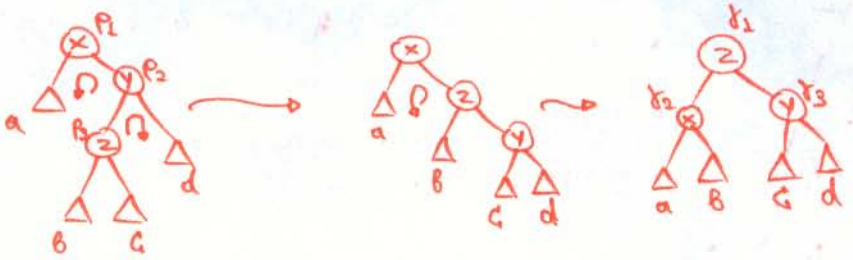


όπου $\gamma_1 = \rho_1 + (1-\rho_1)\rho_2$

$\gamma_2 = \frac{\rho_1}{\rho_1 + (1-\rho_1)\rho_2} = \frac{\rho_1}{\gamma_1}$

→ διπλή περιστροφή:

$\rho_1, \rho_2, \rho_3, \gamma_1, \gamma_2, \gamma_3$ είναι οι βαροζυγίσεις των αντίστοιχων κόμβων



όπου $\gamma_1 = \rho_1 + (1-\rho_1)\rho_2\rho_3$, $\gamma_2 = \frac{\rho_1}{\rho_1 + (1-\rho_1)\rho_2\rho_3} = \frac{\rho_1}{\gamma_1}$, $\gamma_3 = \frac{\rho_2(1-\rho_3)}{1-\rho_2\rho_3}$

(*) Για όλα τα $a \in \left(\frac{1}{4}, 1-\frac{\sqrt{2}}{2}\right]$ υπάρχουν σταθερές $d \in [a, 1-a]$ και $\delta \ge 0$ τέτοιες ώστε για ένα δυϊκό δένδρο T με υπόδενδρα T_L και T_R (τα οποία είναι BB[a]-δένδρα) να έχουμε $\rho(T) = \frac{|T_L|}{|T|}$ και:

$\rho(T) = \frac{|T_L|}{|T|} < a$ και

$\frac{|T_L|}{|T|-1} \ge a$ (δηλαδή έγινε ένθεση στο δεξί υπόδενδρο)

$\frac{|T_L|+1}{|T|+1} \ge a$ (" " αποβεση στο αριστερό ")

$\xrightarrow[\text{ισχύει}]{\text{τότε θα}}$

i) αν $\rho(T_L) \le d$ τότε μια απλή περιστροφή επαναφέρει τη ζύγιση και $\gamma_1, \gamma_2 \in [(1+\delta)a, 1-(1+\delta)a]$

ii) αν $\rho(T_L) > d$ τότε μια διπλή περιστροφή επαναφέρει τη ζύγιση και $\gamma_1, \gamma_2, \gamma_3 \in [(1+\delta)a, 1-(1+\delta)a]$

(ανάλογα ισχύουν και για $\rho(T) > 1-a$)

Γενικά, μετά την ένθεση ή την αποβεση εφαρμόζουμε επαναφορτωτικές πράξεις στο μονοπάτι από το φύλλο προς τη ρίζα οι οποίες στη χειρότερη περίπτωση είναι $O(\log n)$. Άρα, στη χ.π. έχουμε για ψάξιμο, ένθεση και αποβεση $O(\log n)$ πολυπλοκότητα.

Η ιδιότητα βάρους εξυπηρετεί για ισοδυναμοποιημένη καθυστέρηση των αναβλητών επαναζυγιστικών πράξεων και έτσι μπορεί να γίνει καταμερισμός (amortization)

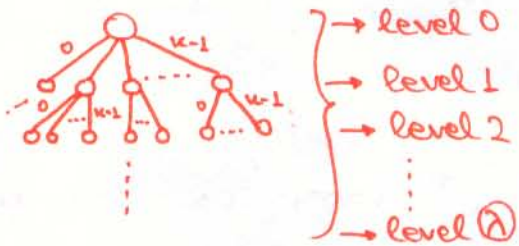
• Υβριδικές δομές δεδομένων

1) **Tries (ψηφιακά δένδρα)**: προτάθηκε ως ένας απλός τρόπος δόμησης ενός file χρησιμοποιώντας την ψηφιακή αναπαράσταση των στοιχείων του.

Ορίζουμε ένα σύμπαν $U=\{0,\dots,k-1\}^{\lambda}$ το οποίο αποτελείται από strings μήκους λ ψηφίων τα οποία δομούνται πάνω σε ένα αλφάβητο με k στοιχεία. ($k \geq 2$)

- Ένα σύνολο $S \subseteq U$ αναπαρίσταται ως ένα k-ικό δένδρο που περιέχει όλα τα προθέματα (prefixes) του S (των στοιχείων του).
- Μια πρακτική υλοποίηση ενός trie είναι να χρησιμοποιήσουμε για κάθε εσωτερικό κόμβο του δένδρου ένα array μήκους k.

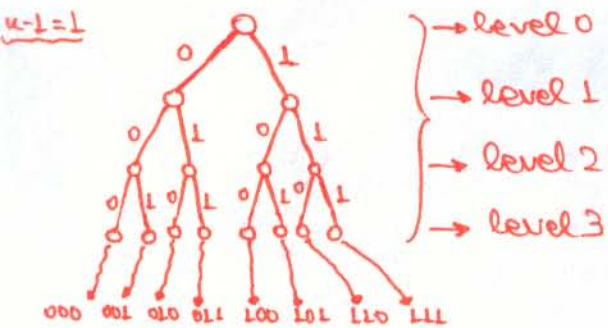
Αναπαράσταση του συνόλου $S \subseteq U$:



Στην περίπτωση της αναπαράστασης του trie που αποθηκεύει όλα τα στοιχεία του σύμπαντος U θα είναι: $\lambda = \log_k |U|$ όπου $|U|$ είναι το μέγεθος του σύμπαντος.

- Θεωρούμε το σύμπαν $U=\{0,1\}^3$, όπου δηλαδή $k=2$ στοιχεία (0,1) και $\lambda=3$, δηλαδή έχουμε strings των 3 ψηφίων. Τότε το μέγεθος του σύμπαντος U είναι $|U|=8$, αφού σ'αυτό αποθηκεύονται (περιέχονται) τα strings 000, 001, 010, 011, 100, 101, 110, 111. Θα αναπαραστήσουμε το πλήρες trie που αποθηκεύει όλα τα στοιχεία του U. Είναι:

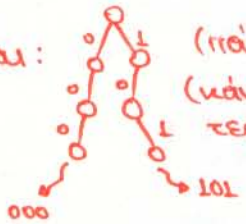
$k-1=1$



Βλέπουμε ότι $h(\text{trie})=3$ και το τελευταίο επίπεδο (level λ) είναι το 3. Οπότε: $\lambda = \log_2|U| = \log_2|8| = 3$ (ουσιαστικά).

Ο αριθμός φύλλων του trie είναι όσο και το μέγεθος του U. ($k^h = k^{\lambda} = 2^3 = 8$)

Για το σύνολο $S=\{000,101\} \subseteq U$ το νέο trie είναι:



(πάμε από τη ρίζα προς τα κάτω)
(κάναμε compression του εξωτερικού βαθμού των κόμβων)

Παρατηρούμε ότι θέλουμε χρόνο $O(1)$ για ψάξιμο.

Γενικά, ο χώρος που χρησιμοποιεί το trie είναι υπερβολικός: $O(n \cdot \lambda \cdot k)$, όπου λ είναι το ύψος του trie, k είναι τα στοιχεία του αλφαβήτου του σύμπαντος και n είναι το μέγεθος του συνόλου S ($|S|=n$). Οπότε, έχουμε:

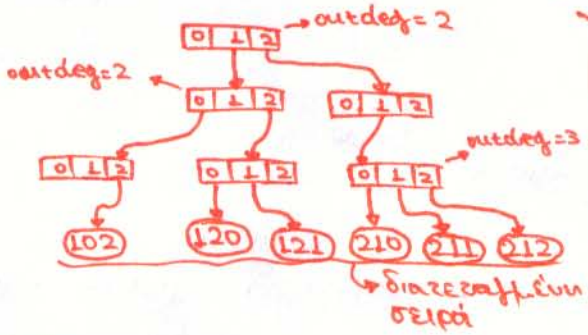
$$\text{χώρος} = n \cdot \lambda \cdot k = n \cdot \log_k |U| \cdot k \Rightarrow \boxed{\text{χώρος} = nk \cdot \frac{\log|U|}{\log k}}$$

$$\text{χρόνος} = O(\log_k |U|) = O\left(\frac{\log|U|}{\log k}\right) \Rightarrow \boxed{\text{χρόνος} = O\left(\frac{\log|U|}{\log k}\right)}$$

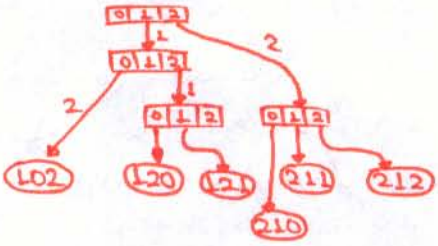
k αυξάνεται όταν το [redacted] τότε έχουμε ελάχιστο χρόνο και μέγιστο χώρο. (όταν $k \nearrow$ αυξάνεται)

(*) ο χρόνος δεν εξαρτάται από το $n=|S|$ αλλά από το $|U|$, στη μέση περίπτωση όμως αυτός είναι $O(\log_k n)$

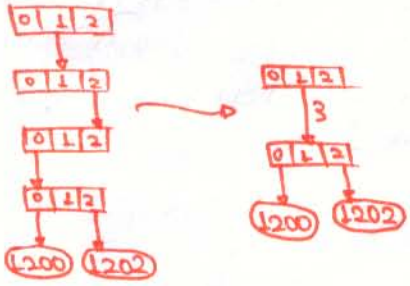
⊛ Για το σύνολο $S = \{121, 102, 211, 120, 210, 212\}$ χρησιμοποιούμε ένα array 3 θέσεων σε κάθε εσωτερικό κόμβο και παίρνουμε το δένδρο:



Για να ελαττώσουμε το χώρο σε $O(n \cdot k)$ αποθηκεύουμε μόνο τους εσωτερικούς κόμβους που έχουν outdegree ≥ 2. Τότε θα έχουμε το επόμενο σχήμα:



π.χ. ⟹



15

2) Interpolation search tree

- Ένα IST είναι ένα πολυδιαμελισμένο δένδρο όπου ο βαθμός διαμελίσεως κάθε κόμβου εξαρτάται από το πλήθος των στοιχείων που αποθηκεύει στο υπόδενδρό του. Ακριβέστερα, ο βαθμός της ρίζας είναι $\Theta(\sqrt{n})$. Η ρίζα διαιρεί το file σε $\Theta(\sqrt{n})$ υπο-files διαμορφούμενα μεταξύ τους μήκους.
- Ένα ιδανικό IST είναι ένα IST στο οποίο όλα τα υπο-files έχουν περίπου ίσο μέγεθος και επομένως οι γιοί της ρίζας έχουν βαθμό $\Theta(\sqrt{n})$. Τα ιδανικά IST έχουν βάθος $O(\log\log n)$
- Ο αλγόριθμος ψαξίματος για το IST χρησιμοποιεί interpolation search σε κάθε κόμβο του IST για να βρει το υπο-file όπου θα συνεχίσει το ψάξιμο. Γι' αυτό οι κόμβοι υλοποιούνται ως arrays. Σε κάθε array προσαρτούμε και μια προσέγγιση της αντίστοιχης συνάρτησης κατανομής, η οποία μας επιτρέπει να κάνουμε τέλειο interpolate σε ένα δείγμα μεγέθους n^a, όπου $1/2 \leq a < 1$ είναι μια παράμετρος.
- Για να αποθηκευθούν n στοιχεία ο χώρος που χρησιμοποιείται είναι γραμμικός: $\boxed{O(n)}$
- Για ψάξιμο σε ένα IST μεγέθους n, προφανώς χρειάζονται $O(\sqrt{n})$ μονάδες χρόνου για το ψάξιμο στο array της ρίζας. Το επόμενο υπο-file που θα ψαχθεί θα έχει μέγεθος το πολύ $n/2$ και συνεπώς θα ισχύει:

  $T(n) \leq O(\sqrt{n}) + T(n/2) \Longrightarrow \boxed{T(n) = O(\sqrt{n})}$, όπου $T(n)$ ο χειρότερος χρόνος.

  Αυτός ο χρόνος βελτιώνεται σε $\boxed{T(n) = O(\log^2 n)}$ εφαρμόζοντας ενθετικό και δυϊκό ψάξιμο.
- Για n ενθέσεις ή αποβέσεις ο χρόνος χειρότερης περίπτωσης είναι $O(n \log n)$ και το αντίστοιχο κατανεμημένο κόστος (μέσος όρος) είναι $O\left(\frac{n \log n}{n}\right) = \boxed{O(\log n)}$

  Ο χρόνος μέσης περίπτωσης με μορφή κατανεμημένου κόστους είναι $\boxed{O(\log\log n)}$
- Σε κάθε κόμβο v του IST προσδίδουμε έναν αριθμητή $c(v)$. Αυτός απαριθμεί το πλήθος των ενθέσεων ή διαγραφών που γίνονται στο υπόδενδρο T_v του v. Αν ανακατασκευάσουμε το T_v του κόμβου v τότε δίνουμε αρχική τιμή 0 σε όλους τους μετρητές των κόμβων του T_v. Ένας αριθμητής $c(v)$ υπερχειλίζει (overflows) όταν η τιμή του ξεπερνάει το $w/4$, όπου w είναι το μέγεθος (size) του κόμβου v τη στιγμή που ανακατασκευάσθηκε το δένδρο (δηλ. όταν $c(v)=0$ την τελευταία φορά). Όταν γίνει overflow τότε γίνεται και η αντίστοιχη ανακατασκευή.
- Ορίζουμε ως $w(v)$ το βάρος ενός κόμβου που είναι το πλήθος των στοιχείων που είναι αποθηκευμένα στο T_v. Το μέγεθος του v είναι το πλήθος των στοιχείων του T_v τα οποία είναι μη-μαρκαρισμένα (μαρκαρισμένο λέγεται ένα στοιχείο όταν αυτό πρόκειται να αποβεβοθεί. Δεν απομακρύνεται όμως παράλληλο τότε που θα γίνει η ανακατασκευή του υπόδενδρου που το περιέχει).

• Μια ένθεση στο IST γίνεται ως εξής: βρίσκουμε τη θέση που θα έπρεπε να βρίσκεται το x και το εισάγουμε σ' αυτή. Επειτα, αυξάνουμε τους μετρητές όλων των κόμβων τ που βρίσκονται στο μονοπάτι από το x στη ρίζα. Βρίσκουμε τον υψηλότερο τέτοιο κόμβο, ώστε ο μετρητής του να υπερχειλίζει, κι αν υπάρχει αυτός αναδιαρθρωνόμαστε το υπόδενδρό του.

• Μια απόθεση στο IST γίνεται ως εξής: βρίσκουμε το x και το διαγράφουμε ως φύλλο και αυξάνουμε κατά 1 όλους τους κόμβους που βρίσκονται στο μονοπάτι από το x στη ρίζα. Επειτα, βρίσκουμε τον υψηλότερο κόμβο του οποίου ο μετρητής έχει υπερχειλιοθεί και αναδιαρθρωνόμαστε το υπόδενδρό του.

• Ένα IST με όρια a και b για ένα σύνολο $S = \{x_1 < x_2 < \dots < x_n\} \subseteq [a,b]$ με n στοιχεία αποτελείται από δύο πίνακες: τον $REP[1 \dots k]$ με $\sqrt{n}/2 \le k \le 2\sqrt{n}$, ο οποίος περιέχει ένα δείγμα του file S (στα δυναμικά IST απαιτούμε το δείγμα να είναι ισοκατανεμημένο) και τον $ID[1 \dots m]$ με $m \le k$, στον οποίο αποθηκεύουμε την αντίστοιχη κατανομή των παιδιών (επί τεύξη τέλειου interpolate)

• Στην αναζήτηση του ευθύγραμμου δίκτυου φάσματος συγκρίνουμε το στοιχείο x που ψάχνουμε με το $REP[j], REP[j+2^0], REP[j+2^1], REP[j+2^2], \dots$ μέχρι να βρεθεί λ τέτοιο ώστε:

$REP[j+2^{\lambda-1}] < x \le REP[j+2^{\lambda}]$. Με δυαδικό φάσμα, ο χειρότερος χρόνος αναζ. γίνεται $O(\log n)$ και συνολικά στο IST λαμβάνουμε $O(\log^2 n)$

• Για το φάσμα, το IST παρουσιάζει κακή χειρότερη συμπεριφορά αν τα βάρη κατά μήκος κάθε μονοπατιού από τη ρίζα στα φύλλα έχουν γεωμετρική πτώση, διότι έτσι προκύπτει λογαριθμικό βάθος. Για ενθέσεις ή αποθέσεις έχουμε κακή χειρότερη συμπεριφορά αν οι κόμβοι μεγέθους n μένουν ανενεργοί για πολύ χρόνο. Και οι δύο συμπεριφορές επιτυγχάνονται μέσω των αριθμητών των κόμβων.

HASHING

• Έστω σύμπαν $U=\{0,\dots,m-1\}$ και ένα hashtable (hash πίνακας) $T[0,\dots,m-1]$. Τότε ορίζουμε μια hash συνάρτηση $h: U \to [0,\dots,m-1]$ τέτοια ώστε να αποθηκεύει ένα στοιχείο $x \in U$ στο $T[h(x)]$.

• Για παράδειγμα, έχουμε το $T=[0,1,2,3,4]$ $(m=5)$ και θέλουμε μέσα σ' αυτό να αποθηκεύσουμε το σύνολο $S=\{3,15,22,24\}$ που είναι υποσύνολο του σύμπαντος $U=[0,\dots,99]$. Χρησιμοποιούμε τη συνάρτηση $h(x)=x \bmod m \Rightarrow \underline{h(x)=x \bmod 5}$.

0	15
1	
2	22.
3	3.
4	24.

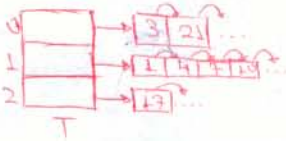
T

Είναι: $h(3)=3 \bmod 5=3$
$h(15)=15 \bmod 5=0$
$h(22)=22 \bmod 5=2$
$h(24)=24 \bmod 5=4$

Παρατηρούμε ότι δεν έχουμε καμία σύγκρουση (collision). Αλλά, αν είχαμε να εισάγουμε και το $x=20$, τότε $h(20)=20 \bmod 5=0$. Στη θέση 0 όμως είναι αποθηκευμένο το 15. Άρα έχουμε πρόβλημα.

• Το παραπάνω πρόβλημα λύνεται με hashing with chaining (με αλυσίδες) και hashing without addressing (ανοικτής διεύθυνσης).

• <u>Ανάλυση του hashing with chaining:</u> Όλα τα κελιά του T είναι κεφαλές γραμμικών λιστών. Επομένως, η i-οστή λίστα θα περιέχει όλα εκείνα τα στοιχεία ενός συνόλου S για τα οποία $h(x)=i$. Για παράδειγμα, θεωρούμε τη hashταμπέλλα $T=[0,1,2]$ $(m=3)$ και το σύνολο $S=\{1,3,4,7,10,17,21\}$. Η hash συνάρτηση είναι η $h(x)=x \bmod m \Rightarrow \underline{h(x)=x \bmod 3}$.



$h(1)=1 \bmod 3=1$ $\quad h(7)=7 \bmod 3=1$ $\quad h(21)=21 \bmod 3=0$
$h(3)=3 \bmod 3=0$ $\quad h(10)=10 \bmod 3=1$
$h(4)=4 \bmod 3=1$ $\quad h(17)=17 \bmod 3=2$

⊛ Στη μέση περίπτωση, η πολυπλοκότητα για ψάξιμο ενός στοιχείου $x \in S$, δηλαδή το να βρούμε την $h(x)$ και μετά να ψάξουμε το x στην αντίστοιχη λίστα $T[h(x)]$, απαιτεί σταθερό χρόνο, έστω και το μήκος της συγκεκριμένης λίστας. Το ίδιο συμβαίνει και για τις εισάγεις και διαγραφές, αφού αυτές σε μια λίστα εκτελούνται σε $O(1)$ χρόνο.

Στη χειρότερη περίπτωση, η πολυπλοκότητα για ψάξιμο, έτσι και αλλιώς είναι $O(|S|)$. Αυτό εμφανίζεται όταν ολόκληρο το σύνολο S αποθηκεύεται σε μία και μόνο λίστα.

• Μελετώντας μια ακολουθία από n εισάγεις, διαγραφές ή ψαξίματα, θα κάνουμε τις εξής παραδοχές:
- Η hash συνάρτηση h κατανέμει ισομερώς τα στοιχεία του σύμπαντος U πάνω στο διάστημα του $T[0,\dots,m-1]$, δηλαδή για όλα τα $i,j \in [0,\dots,m-1]$ ισχύει: $|h^{-1}(i)|=|h^{-1}(j)|$.
(αν π.χ. έχω σύμπαν 9 στοιχείων και τη $h=x \bmod 3$, τότε 3 στοιχεία θα πάνε στο 0, 3 στο 1, 3 στο 2)
- Έχουμε ομοιόμορφη κατανομή στην είσοδο, δηλαδή όλα τα στοιχεία του U εμφανίζονται ισοπίθανα στην ακολουθία μας πράξεις, με πιθανότητα δηλαδή $1/|U|$.

⊛ Ορίζουμε ως $b=\frac{n}{m}$ τον παράγοντα κατανομής. Τότε, αν πάρουμε το μέσο πλήθος των στοιχείων της λίστας, θα πρέπει $\underline{m \cdot (\text{μέσο πλήθος})=n}$, και το $b=\frac{n}{m}$ θα είναι αυτό το μέσο πλήθος.

• Πολυπλοκότητα μέσης περίπτωσης για n πράξεις είναι $O[(1+\frac{b}{2})n]$, όπου $b=\frac{n}{m}$ είναι ο μέγιστος παράγοντας κατανομής. Το κατανεμημένο κόστος (μέσο) είναι τότε $O(n)$

• Το μέσο μήκος της μεγαλύτερης αλυσίδας είναι $O(\frac{\log n}{\log\log n})$ όταν το $b=\frac{n}{m} \le 1$.

• <u>Ανάλυση του hashing with open addressing:</u> Εδώ όλα τα στοιχεία του σύμπαντος U αποθηκεύονται στο T. Δηλαδή, όλα τα κελιά του T θα περιέχουν ένα στοιχείο ή θα είναι κενά. Κάτι τέτοιο γίνεται δυνατό με το να θεωρήσουμε ότι κάθε στοιχείο $x \in U$ ορίζει μια ακολουθία από hash συναρτήσεις $h(0,x), h(1,x), \dots$ οι οποίες αποτελούν θέσεις στο T. Με άλλα λόγια, εισάγουμε ένα στοιχείο x ακολουθώντας αυτή την ακολουθία μέχρι να βρεθεί μια ελεύθερη θέση για το x. Ακριβώς με τον ίδιο τρόπο λειτουργεί και το ψάξιμο.

• Έχουμε τη hash συνάρτηση $h(i,x) = [h_1(x) + i\, h_2(x)] \bmod m$, όπου $h_1(x) = x \bmod 7$, $h_2(x) = 1 + x \bmod 4$. Διαθέτουμε μια hash-πίνακα $T = [0,1,2,3,4,5,6]$ $(m=7)$ και ακολουθώντας αυτή τη συνάρτηση, θέλουμε να αποθηκεύσουμε στο T το σύνολο $S = \{3,17,6,9,15,13,10\}$. Έχουμε:

0	
1	15
2	9
3	3
4	
5	17
6	6

T

$h(0,3) = h_1(3) \bmod 7 = [3 \bmod 7] \bmod 7 = 3$

$h(0,17) = h_1(17) \bmod 7 = [17 \bmod 7] \bmod 7 = 3 \bmod 7 = 3 \Rightarrow$ έχουμε σύγκρουση, άρα ακολουθούμε την ακολουθία για $i=1$.

$h(1,17) = [h_1(17) + h_2(17)] \bmod 7 = [17 \bmod 7 + 1 + 17 \bmod 4] \bmod 7 = (3+1+1) \bmod 7 = 5$

$h(0,6) = h_1(6) \bmod 7 = (6 \bmod 7) \bmod 7 = 6$

$h(0,9) = h_1(9) \bmod 7 = 2$

$h(0,15) = h_1(15) \bmod 7 = 1$

$h(0,13) = (13 \bmod 7) \bmod 7 = 6$ (collision)

$h(1,13) = [h_1(13) + h_2(13)] \bmod 7 = \ldots$ collision ....

$h(4,13) = [13 \bmod 7 + 4 + 4 \cdot 13 \bmod 4] \bmod 7 =$

$= (6 + 4 + 52 \bmod 4) \bmod 7 =$ collision..

$h(0,10) = \ldots =$ collision ....

• Δεν χρησιμοποιείται και επιπλέον χώρος για αποθήκευση δεικτών. Στη διαγραφή όμως έχουμε πρόβλημα με το ακριβέστερο σημείο ενός στοιχείου, γιατί τότε χαλάει η δομή και δε γίνεται σωστά η ένθεση και το ψάξιμο. Γι' αυτό ένα κόλπο θεωρούμε ότι έχει 3 καταστάσεις: ελεύθερο, κατειλημμένο ή σημειωμένο. Δηλαδή, στη διαγραφή <u>χαρακτηρίζουμε</u> ένα στοιχείο ως σημειωμένο (lazy deletion).

• Πολυπλοκότητα μέσης περίπτωσης για ένθεση $= \boxed{O\left(\frac{1}{1-\beta}\right)}$ όπου $\beta < 1 \Rightarrow \frac{n}{m} < 1 \Rightarrow n < m$

" " " " ψάξιμο $= \boxed{O\left(\frac{1}{\beta} \ln \frac{1}{1-\beta}\right)}$, όπου $n<m$

Στη χειρότερη περίπτωση και στις δύο περιπτώσεις έχουμε πολυπλοκότητα $\boxed{O(m)}$

• Συνήθως, το open addressing απαιτεί να επιλέξουμε το m μεγάλο αλλά όμως μπορεί να συνεπάγεται κατάχρηση χώρου. Γι' αυτό μια συνήθης τακτική είναι να επιλέξουμε αρχικά μικρό m, δηλαδή μικρό μέγεθος του T και να το διπλασιάσουμε όταν το n γίνει αρκετά μεγάλο.

Για να αντιμετωπίσουμε κάτι τέτοιο διατηρούμε μια ακολουθία από hash-πίνακες $T_0, T_1, T_2, \ldots$ με αντίστοιχες hash συναρτήσεις $h_0, h_1, h_2, \ldots$ όπου $|T_0| = m$ και $|T_i| = 2^i m$. Οπότε, για τον παράγοντα κορεσμού β_i έχουμε τα εξής:

$$T_i = \begin{cases} T_{i+1}, & \beta_i = 1 \ (A) \\ T_{i-1}, & \beta_i = \frac{1}{4} \ (B) \end{cases}$$

Όταν ο β γίνει ίσος με 1, δηλαδή $n=m$ και ο T_i έχει γεμίσει με $2^i m$ στοιχεία τότε δημιουργούμε τον $T_{i+1} = 2^{i+1} m$ και ενθέτουμε τα προηγούμενα στοιχεία.

• Τα έξοδα μετάβασης για να πάμε από τον $T_i \rightsquigarrow T_{i+1}$ είναι $\underline{O(2^i \cdot m)}$. Τότε προκύπτει ότι $\beta_{i+1} = \frac{1}{2}$ και πρέπει να εκτελέσουμε <u>τουλάχιστον</u> $\frac{1}{2} \cdot (2^i m) = \frac{1}{4} \cdot (2^{i+1} \cdot m)$ πράξεις στον T_{i+1} μέχρι να γίνει ο παράγοντάς του $\beta_{i+1} = \frac{1}{4}$ και να παραχθούν νέα έξοδα μετάβασης. Με κατανομή αυτών των εξόδων έχουμε κατανεμημένο κόστος $\underbrace{\frac{2^i \cdot m}{2^{i-1} \cdot m} = 2 = O(1)}$

• Τα έξοδα μετάβασης για να πάμε από τον $T_i \rightsquigarrow T_{i-1}$ είναι $O\left(\frac{1}{4} \cdot 2^i m\right)$. Τότε προκύπτει ότι $\beta_{i-1} = \frac{1}{2}$ και πρέπει να εκτελέσουμε <u>τουλάχιστον</u> $\frac{1}{8}(2^i \cdot m) = \frac{1}{4} \cdot (2^{i-1} \cdot m)$ πράξεις στο T_{i-1} μέχρι να γίνει ο παράγοντάς του $\beta_{i-1} = \frac{1}{4}$. Ανάλογα έχουμε κατανεμημένο κόστος $O(1) = 2$.

• **Extendible hashing**: χρησιμοποιείται όταν η κύρια μνήμη δεν είναι αρκετά μεγάλη για να αποθηκεύσει ολόκληρο το hash πίνακα. Γι' αυτό μεταφέρουμε το hashing στη δευτερεύουσα μνήμη (secondary memory).

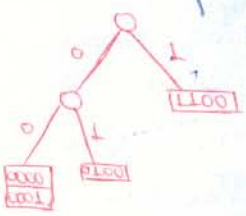
Θεωρούμε ότι η δευτερεύουσα μνήμη είναι χωρισμένη σε κάδους (buckets) ορισμένου μεγέθους A στους οποίους αποθηκεύουμε δεδομένα μεγέθους B. Αν είναι $S \subseteq U$ το υποσύνολο-κλειδί στο U που θέλουμε να αποθηκεύσουμε και αν είναι $d(S)$ το βάθος του S σε σχέση με το b, όπου b είναι το μέγεθος ενός bucket, τότε αποθηκεύουμε όλα τα στοιχεία σε buckets τα οποία αποτελούν φύλλα ενός δυαδικού TRIE βάθους $d(S)$. Τότε, το ψάξιμο στα φύλλα επιτυγχάνεται σε $O(1)$ χρόνο. (το μέσο πλήθος για n buckets είναι $O(\frac{n}{b \log n})$)

• Το extendible hashing χρησιμοποιεί μια hash ταμπέλλα $T=[0,\dots,2^{d(S)}-1]$ που ονομάζεται directory και μερικά buckets για να αποθηκεύσει τα στοιχεία του συνόλου S.

Για παράδειγμα, θεωρούμε buckets μεγέθους $b=2$, μια συνάρτηση $h: U \to \{0,1\}^k$, $k \in N$ και το σύνολο S στο οποίο θα αποθηκεύσουμε τα στοιχεία, και $h(S)=\{0000, 0001, 0100, 1100\}$

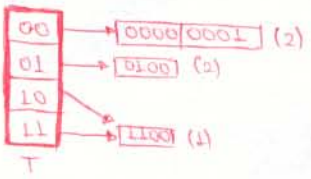
Φτιάχνουμε το αντίστοιχο trie:

Βλέπουμε ότι ένα bucket δε χρησιμοποιείται καθόλου, οπότε θα συμπιέσουμε το trie (compression) →



(σπάσαμε τον κλάδο χωρίς παιδί και έπειτα τον αντίστοιχο κόμβο).

Παρατηρούμε ότι το ύψος του trie είναι 2, άρα $\boxed{d(S)=2}$. Οπότε, σχεδιάζουμε τη δομή:



Τα αντίστοιχα βάθη των buckets, που φαίνονται στις παρενθέσεις, είναι τα τοπικά βάθη των buckets στο trie που είχαμε compressed. Γι' αυτά θα ισχύει, ότι αν είναι $2^{d(S)-r}$ ο αριθμός των στοιχείων του T που δείχνουν σε ένα bucket, τότε το r είναι το τοπικό βάθος του bucket.

(π.χ. για το τελευταίο bucket είναι: $2^{d(S)-r}=2 \Rightarrow d(S)-r=1 \Rightarrow 2-r=1 \Rightarrow r=1$)

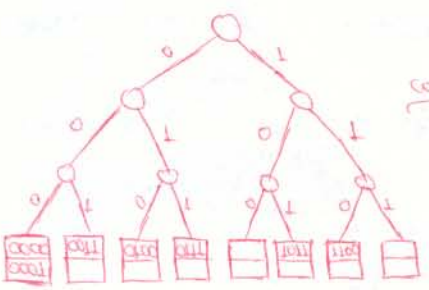
• Για να ενθέσουμε ένα στοιχείο κοιτάμε αν αυτό χωράει σε ένα από τα υπάρχοντα buckets. Αν δε χωράει τότε διπλασιάζουμε το μέγεθος του T και κατασκευάζουμε το trie από την αρχή.

→ Στο προηγούμενο παράδειγμα, αν θέλουμε να ενθέσουμε το στοιχείο 0111 βλέπουμε ότι το σχετικό bucket [0100] έχει χώρο, οπότε το ενθέτουμε χωρίς πρόβλημα.

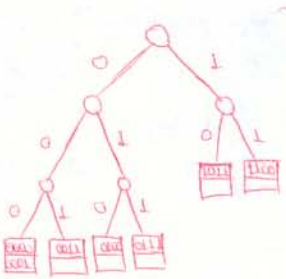
→ Αν, τώρα, θέλουμε να ενθέσουμε το στοιχείο 1011 θα πρέπει να κάνουμε το trie όπως ήταν πριν διπλασιάσει decompressed:

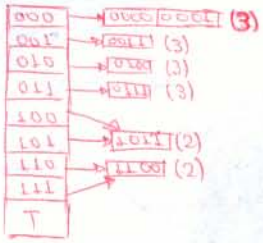
οπότε ο T γίνεται:

→ Αν, όμως, θέλουμε να ενθέσουμε το στοιχείο 0011 (δεν χωράει) θα πρέπει να κάνουμε νέο trie και να διπλασιάσουμε το T:



compression →





• Για να διαγράψουμε ένα στοιχείο, κοιτάμε αν αρχικά ο κάδος είναι γεμάτος. Αν είναι, αφαιρούμε το στοιχείο (διαγραφή). Αν δεν είναι γεμάτος και έχει 1 μόνο στοιχείο τότε μειώνουμε το βάθος του δένδρου στο μισό και κάνουμε τις απαραίτητες διευθετήσεις στον T (που θα δείχνουν πλέον τα στοιχεία μετά τη μείωση)

UNION-FIND

Θεωρούμε ένα σύμπαν $U = \{0,1,\dots,N-1\}$ και το διαχωρίζουμε σε υποσύνολα ξένα μεταξύ τους, δηλαδή που να μην έχουν κοινά στοιχεία το ένα με το άλλο. Τότε ορίζουμε τις πράξεις find(x), που επιστρέφει το υποσύνολο εκείνο στο οποίο βρίσκεται το x και union(A,B,C) η οποία επιστρέφει ένα σύνολο $C \subseteq U$ το οποίο είναι συνένωση των A και B.

- Χρησιμοποιούμε έναν πίνακα IS-in[0,…,N-1], όπου IS-in[i] είναι το όνομα του υποσυνόλου που περιέχει το i. Τότε η find(x) = IS-in[x] εκτελείται σε O(1) χρόνο (constant) αλλά η union(A,B,C) σε χρόνο O(N), αφού θα πρέπει να διασχίσουμε όλο τον πίνακα και να αντικαταστήσουμε τα A,B.

  Κάτι τέτοιο μπορεί να αποφευχθεί αν κάθε σύνολο διαφέρει για λίστα των στοιχείων που περιέχει. Τότε για union(A,B,C) θα πρέπει να διατρέξουμε μόνο τις λίστες των A και B και όχι όλο τον πίνακα. Σ'αυτή την περίπτωση έχουμε χρόνο $O(|A| \cup |B|)$. Ο χρόνος αυτός βελτιώνεται σε $O(\min\{|A|,|B|\})$ με την εφαρμογή του weighted union rule, κατά το οποίο σε κάθε λίστα ενός υποσυνόλου αποθηκεύω και το μέγεθός της, οπότε διαγράφουμε μόνο τη μικρότερη λίστα και την ενώνουμε στην άλλη.

  Στην προηγούμενη λίστα υπάρχει ένα πρόβλημα. Αντί η συνένωση των υποσυνόλων να ονομάζεται C για union(A,B,C) αυτή ονομάζεται A (αν πχ. |A|>|B|).

- Σ'αυτή την περίπτωση χρησιμοποιούμε 2 αντιστοιχίσεις πινάκων, τους MAPIN και MAPOUT. Θα είναι

  MAPIN: εξωτερικό όνομα → εσωτερικό όνομα
  MAPOUT: εσωτερικό όνομα → εξωτερικό όνομα

  } δηλαδή για το παραπάνω θα είναι MAPOUT[A] = C

  εξωτερικό όνομα υποσυνόλου με εσωτερικό όνομα A.

  Οπότε, η find καλείται ως find(x) = MAPOUT[IS-in[x]]

  εσωτερικό όνομα του υποσυνόλου που περιέχει το x

- Για την πράξη union(A,B,C), αρχικά ελέγχουμε το μέγεθος των λιστών A και B (χρόνος O(1)), έπειτα αλλάζουμε το εσωτερικό όνομα του μικρότερου υποσυνόλου και συνδέουμε τις λίστες (χρόνος O(1)) και τέλος ενημερώνουμε τα MAPOUT, MAPIN και SIZE (χρόνος O(1)). Ο SIZE[A] δείχνει το μέγεθος του υποσυνόλου με εξωτερικό όνομα A.

- Μια ακολουθία από m Finds και (n-1) Unions έχει συνολικό κόστος $O(m+n\log n)$.
  - Ένα find κοστίζει O(1) γι'αυτό έχουμε O(m) για m Finds.
  - Ένα union(A,B,C) κοστίζει $O(\min\{|A|,|B|\})$ ή $O(\min\{size[A], size[B]\}) \le c \cdot \min\{size[A], size[B]\}$, για μια σταθερά c.

    Για n-1 Unions θα ισχύει ότι $|A| \le n$ και το σύνολο C θα έχει τουλάχιστον διπλάσιο μήκος απ' τα A και B. Συμφώνως: $2 \cdot 2 \cdot 2 \cdot \ldots \cdot 2 \le n \Rightarrow 2^i \le n \Rightarrow i \le \log n$, και αν το στοιχείο $x \in A$ τότε αυτό λαμβάνει μέρος σε εναλλαγές κατά $n/2^i$ φορές. Στη χειρότερη περίπτωση θα είναι $2^i = n$ και $n/2^i = 1 \Rightarrow i = \log n$. Δηλαδή ένα Union κοστίζει $O(c \log n)$. Για n-1 Unions μπορούν να λάβουν μέρος το πολύ 2·(n-1) στοιχεία. Άρα: $2(n-1) \cdot c\log n \Longrightarrow (n-1)$ Unions κοστίζουν στη χειρότερη περίπτωση $O(n\log n)$. Επί το κατανεμημένο κόστος για ένα Union θα είναι $O(\log n)$.

- Το union-find πρόβλημα μπορεί να λυθεί και διαφορετικά. Μπορούμε να παρατάσσουμε κάθε υποσύνολο σαν ένα δένδρο και ως όνομα στο κάθε υποσύνολο να χρησιμοποιούμε τη ρίζα του. (υλοποίηση με 3 πίνακες: Father, Name[0,…,N-1], Root[ ])

  Σ'αυτή την περίπτωση το union κοστίζει O(1) και το find O(n) χρόνο. Μια πρώτη βελτίωση είναι να εφαρμόσουμε weighted union rule έτσι ώστε οι ρίζες των δένδρων με λιγότερους κόμβους να δείχνουν στις ρίζες των δένδρων με τους περισσότερους (δηλαδή να γίνονται παιδιά τους)

  πχ.: Union(1,2,A) Union(3,4,B) Union(A,B,C) Union(5,6,D) Union(7,8,E) Union(A,E,F) Union(C,F,G)

αυτή την περίπτωση για ακολουθία από n-1 Unions και m finds κοστίζει $O(n+m\log n)$

- Αν είναι F το δάσος που παίρνουμε μετά από n-1 Unions τότε ορίζουμε ως rank(x) το ύψος του x μέσα στο F και weight(x) τον αριθμό των κόμβων στο υπόδενδρο που ορίζει το x.
- Θέλουμε να δείξουμε ότι $weight(x) \geq 2^{rank(x)}$ και $rank(x) \leq \log n$
- $weight(x) \geq 2^{rank(x)}$: Αποδεικνύουμε επαγωγικά ως εξής: $rank(x)=0 \Rightarrow 2^{rank(x)}=1 \leq weight(x)$.

Για $rank(x)>0$, αν είναι y ο γιός του x με $rank(y)=rank(x)-1$, τότε όταν ο y έγινε γιος του x τότε ο x είχε τουλάχιστον τόσους απογόνους όσους είχε και ο y (weighted union rule).

- π.χ.

$weight(x')=10$, $weight(y)=9$ $\Big\}$ $weight(x)=19$

21

Οπότε $weight(x) \geq weight(x')+weight(y) \Rightarrow (weight(y) \leq weight(x')) \Rightarrow weight(x) \geq 2\,weight(y) \Rightarrow weight(x) \geq 2\cdot 2^{rank(y)}$

$\Rightarrow weight(x) \geq 2\cdot 2^{rank(y)}$ (επαγωγική υπόθεση)

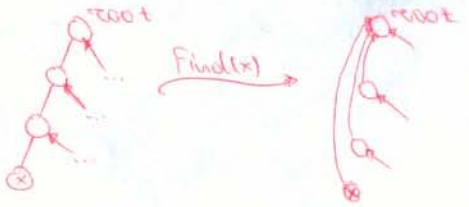
Τότε: $weight(x) \geq 2^{rank(y)+1} \Rightarrow \boxed{weight(x) \geq 2^{rank(x)}}$

- $rank(x) \leq \log n$: Με n-1 Unions μπορούν να παραχθούν σύνολα μεγέθους το πολύ ίσο με n ($\leq n$), οπότε $weight(x) \leq n \Rightarrow 2^{rank(x)} \leq weight(x) \leq n \Rightarrow \boxed{rank(x) \leq \log n}$

ή $weight(root) \geq 2^{rank(root)} \Rightarrow n \geq 2^{rank(root)} \Rightarrow rank(root) \leq \log n$

- Εξαιτίας του $rank(x) \leq \log n$ ένα find κοστίζει το πολύ $\log n$ μονάδες. Ένα Union στοιχίζει $O(1)$
- Μια δεύτερη βελτίωση είναι το path compression: κατά την εκτέλεση ενός find(x) στο μονοπάτι από το x στη ρίζα παίρνουμε τους δείκτες και τους βάζουμε να δείχνουν στη ρίζα. Κάτι τέτοιο κάνει τα επόμενα finds γρηγορότερα.

π.χ.



- Αν στο Union-Find πρόβλημα εφαρμόσουμε weighted union rule και path compression μαζί τότε το συνολικό κόστος για μια ακολουθία από n-1 Unions και m finds είναι $O(m \cdot a(m,n))$ όπου $a(m,n)=\min\{z \geq 1;\ A(z, 4\lceil m/n \rceil) > \log n\}$ είναι μια παραλλαγή της συνάρτησης Ackermann.