Δομές Δεδομένων και Αλγόριθμοι

Ταξινόμηση - αναζήτηση (V1.0)

Χρήστος Γκόγκος

Πανεπιστήμιο Ιωαννίνων, Τμήμα Πληροφορικής και Τηλεπικοινωνιών (2019-2020)

Ταξινόμηση

- Η (αύξουσα) ταξινόμηση μιας συλλογής δεδομένων οδηγεί στη διάταξη των στοιχείων της συλλογής έτσι ώστε $A[i] \leq A[j]$ για $i < j$. Αν υπάρχουν διπλότυπα στη συλλογή τότε στην ταξινομημένη συλλογή θα πρέπει τα στοιχεία αυτά να καταλαμβάνουν συνεχόμενες θέσεις.
- Οι ταξινομημένες συλλογές δεδομένων επιτρέπουν γρήγορες αναζητήσεις.
- Η εύρεση γρήγορων αλγορίθμων ταξινόμησης αποτέλεσε βασικό αντικείμενο έρευνας στα πρώτα χρόνια εμφάνισης των υπολογιστών, εστιάζοντας σε τρόπους ταξινόμησης συλλογών δεδομένων που ήταν πολύ μεγάλα για να χωρέσουν στην κύρια μνήμη των υπολογιστών της εποχής.
- Υπάρχουν δεκάδες αλγόριθμοι ταξινόμησης (π.χ. insertion-sort, quick-sort, merge-sort, heap-sort, counting-sort κ.α.).
- Σήμερα, δίνεται η δυνατότητα αποδοτικής ταξινόμησης δεδομένων της τάξης των εκατοντάδων TeraByte:
  - http://sortbenchmark.org/

Συγκρίσιμα στοιχεία (πρωτογενείς τύποι και λεκτικά)

- Τα στοιχεία μιας συλλογής θα πρέπει να είναι συγκρίσιμα μεταξύ τους προκειμένου η συλλογή να μπορεί να ταξινομηθεί. Δηλαδή για κάθε 2 στοιχεία p και q θα πρέπει ένα ακριβώς από τα ακόλουθα να ισχύει: $p = q$, $p < q$, $p > q$.
- Οι πρωτογενείς (primitive) τύποι δεδομένων όπως οι ακέραιοι αριθμοί, οι αριθμοί κινητής υποδιαστολής και οι χαρακτήρες διατάσσονται με βάση τον αναμενόμενο τρόπο (π.χ. $1 < 2$, $1.5 < 1.6$, $'a' <' b'$).
- Τα λεκτικά διατάσσονται λεξικογραφικά (σύγκριση χαρακτήρα προς χαρακτήρα από αριστερά προς τα δεξιά, οι πρώτοι διαφορετικοί χαρακτήρες καθορίζουν και ποιο λεκτικό θεωρείται μεγαλύτερο, π.χ. "ΑΘΗΝΑ" < "ΑΡΤΑ" διότι 'Θ' < 'Ρ').
  - Unicode collation algorithm (http://www.unicode.org/reports/tr10/)

Συγκρίσιμα στοιχεία (σύνθετοι τύποι)

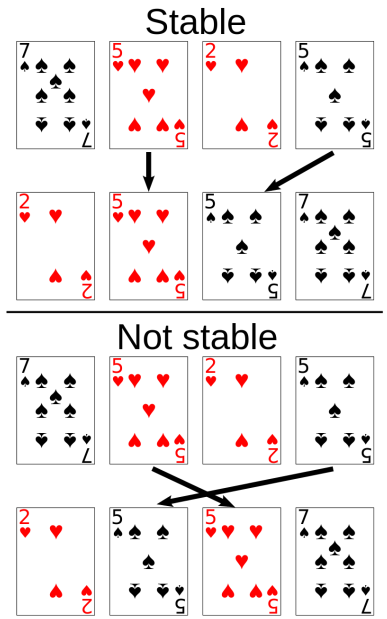
- Σύνθετα δεδομένα (π.χ. εγγραφές, αντικείμενα) διατάσσονται με τρόπο που αποφασίζεται κατά περίπτωση.
- Συνήθως ορίζεται ένα κλειδί (key) και η ταξινόμηση των σύνθετων δεδομένων γίνεται με βάση αυτό το κλειδί (π.χ. εγγραφές φοιτητών μπορεί να διατάσσονται με κλειδί το επώνυμο).

Συγκρίσιμα σύνθετα δεδομένα στη C++

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct student {
    string fname;
    string lname;
};
bool cmp(student &a, student &b) {
    return a.lname < b.lname;
}
int main() {
    vector<student> vs = {{"Maria", "Anagnostou"},
      {"Kostas", "Anagnostakis"}, {"Giannis", "Anagnostakis"}};
    sort(vs.begin(), vs.end(), cmp);
    for (auto &s : vs){
        cout << s.lname << " " << s.fname << endl;
    }
}
```

Σταθερή ταξινόμηση (stable sorting)

Όταν η συνάρτηση ταξινόμησης δύο σύνθετων αντικειμένων καθορίζει ότι δύο στοιχεία στην αρχική μη ταξινομημένη συλλογή είναι ίσα, μπορεί να είναι σημαντικό να διατηρήσουν τη μεταξύ τους σειρά όταν πλέον η συλλογή θα έχει ταξινομηθεί. Οι αλγόριθμοι ταξινόμησης που εγγυώνται ότι θα συμβεί αυτό ονομάζονται σταθεροί (stable).



Οποιοσδήποτε αλγόριθμος ταξινόμησης μπορεί να γίνει stable προσθέτοντας το δείκτη της θέσης του κάθε στοιχείου στο τέλος του κλειδιού.

In-place ταξινόμηση

Όταν η ταξινόμηση δεν απαιτεί επιπλέον δομές προσωρινής αποθήκευσης δεδομένων για να πραγματοποιηθεί και όταν τα δεδομένα επιστρέφονται ταξινομημένα στην αρχική συλλογή τότε η ταξινόμηση λέγεται επι τόπου (in-place, in-situ).

Online sort

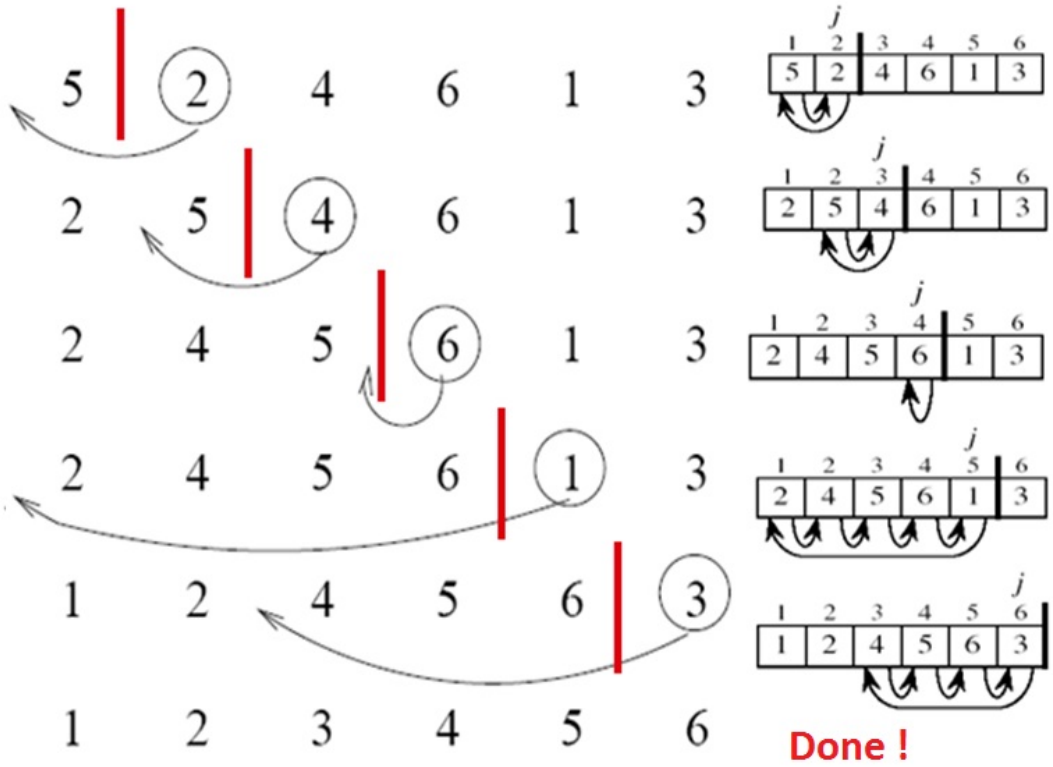
Πρόκειται για αλγορίθμους ταξινόμησης που ανήκουν στην κατηγορία των online αλγορίθμων, δηλαδή αλγορίθμων που επεξεργάζονται τα δεδομένα καθώς τους δίνονται σειριακά. Ένας online αλγόριθμος ταξινόμησης μπορεί να διαθέτει μια ταξινομημένη ακολουθία και να δέχεται μερικά ακόμα στοιχεία τα οποία να ενσωματώνει στην ταξινομημένη ακολουθία.

Ταξινόμηση με εισαγωγή (insertion-sort) (1/2)

```
#include <iostream>
using namespace std;
void insertion_sort(int a[], int n) {
    for (int i = 1; i < n; i++) {
        int key = a[i];
        int j = i - 1;
        while ((j >= 0) && (key < a[j])) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}
int main() {
    int a[] = {4, 3, 2, 7, 5, 6, 1, 8, 10, 9};
    int n = sizeof(a) / sizeof(a[0]);
    insertion_sort(a, n);
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
}
```

Ταξινόμηση με εισαγωγή (insertion-sort) (2/2)

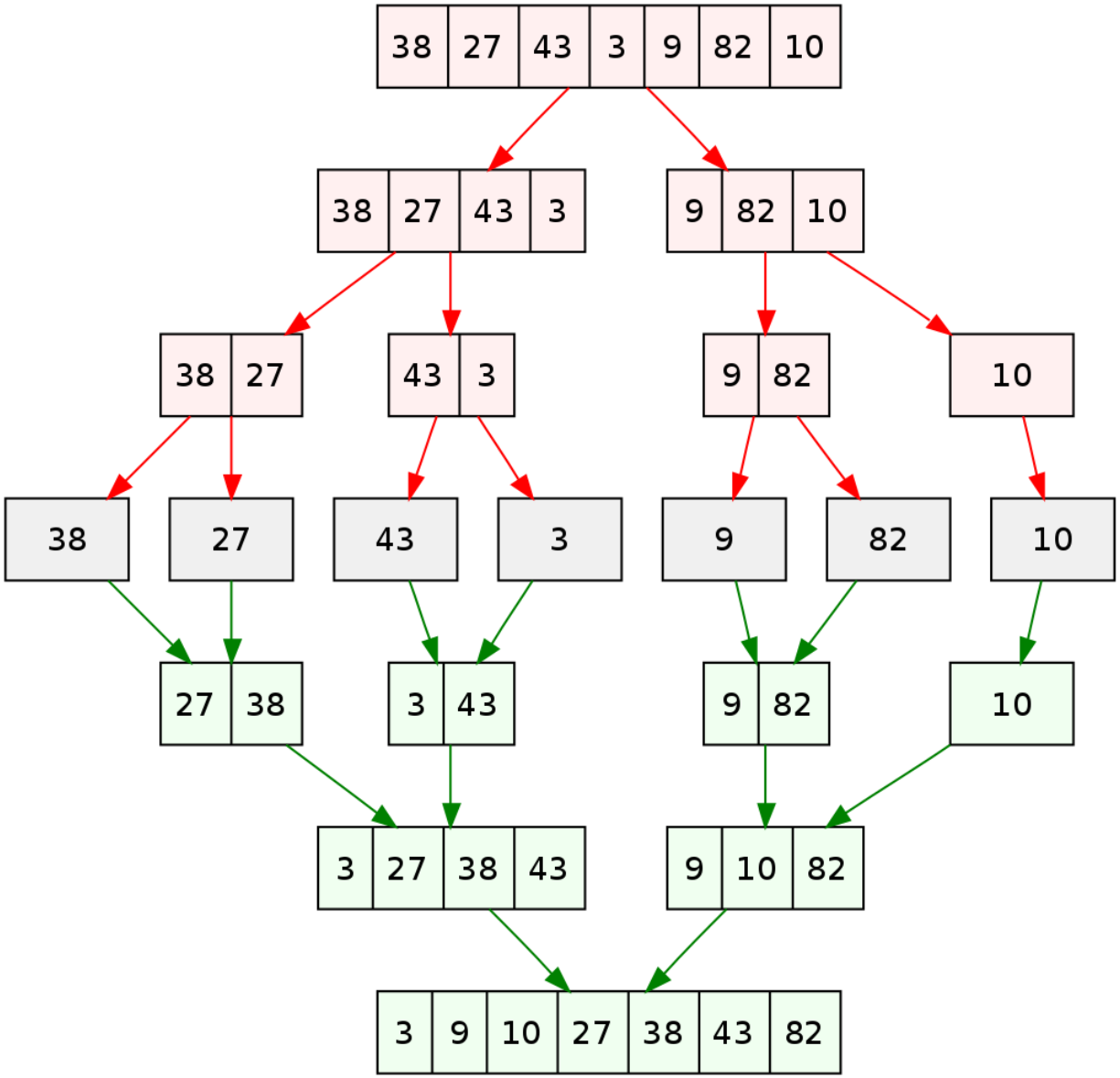
- Η ταξινόμηση με εισαγωγή έχει πολυπλοκότητα $\mathrm{O}(n^2)$
- Η ταξινόμηση με εισαγωγή είναι in-place.
- Μπορεί να χρησιμοποιηθεί όταν το πλήθος των στοιχείων είναι μικρό ή όταν τα στοιχεία στην αρχική συλλογή είναι σχεδόν ταξινομημένα.



Ταξινόμηση με συγχώνευση (merge-sort) (1/5)

- Ο αλγόριθμος merge-sort χρησιμοποιεί την τεχνική **διαίρει και βασίλευε** (divide and conquer)
  - Διαίρει: Διαιρεί τον πίνακα n στοιχείων σε δύο πίνακες μοιράζοντας τα στοιχεία.
  - Βασίλευε: Αναδρομική εκτέλεση του αλγορίθμου για κάθε υποπίνακα ή αν ο πίνακας αποτελείται από ένα μόνο στοιχείο επιστροφή του ίδιου του πίνακα.
  - Συνδύασε: Συγχώνευση ταξινομημένων πινάκων σε ενιαίο πίνακα.
- Βήματα αλγορίθμου:
  - Διαίρεση της ακολουθίας των n στοιχείων σε 2 υποακολουθίες με $\frac{n}{2}$ στοιχεία η καθεμία
  - Αναδρομική ταξινόμηση των υποακολουθιών.
  - Συγχώνευση των ταξινομημένων υποακολουθιών ώστε να σχηματιστεί η τελική ταξινομημένη ακολουθία
- Υπάρχει αναφορά στην ταξινόμηση με συγχώνευση από τον John Von Neuman το 1945!

Ταξινόμηση με συγχώνευση (αριθμητικό παράδειγμα) (2/5)



Ταξινόμηση με συγχώνευση (κώδικας) (3/5)

```
// merging sorted sequences:
// sequence from a[l] to a[m] with sequence from a[m+1] to a[r]
void merge(int a[], int l, int m, int r) {
...
}

void merge_sort(int a[], int l, int r) {
  if (l < r) {
    int m = (l + r) / 2;
    merge_sort(a, l, m); merge_sort(a, m + 1, r);
    merge(a, l, m, r);
  }
}
void merge_sort(int a[], int N) { merge_sort(a, 0, N - 1); }
int main(){
    int a[]={3,2,6,9,1,5};
    merge_sort(a, 6);
    ...
}
```

https://github.com/chgogos/ceteiep_dsa/blob/master/lab03/merge_sort.cpp

Ταξινόμηση με συγχώνευση (αναδρομικός τύπος - πολυπλοκότητα) (4/5)

Ο χρόνος εκτέλεσης του αλγορίθμου για μια συλλογή n στοιχείων δίνεται αναδρομικά από τον ακόλουθο τύπο:

$$T(n) = \begin{cases} \Theta(1) & \text{if} \quad n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if} \quad n > 1 \end{cases}$$

Από το master θεώρημα προκύπτει ότι η πολυπλοκότητα του αλγορίθμου είναι $\mathrm{O}(n \log n)$

Ταξινόμηση με συγχώνευση (ανάλυση) (5/5)

- Αν κατασκευαστεί το δένδρο αναδρομών (στο οποίο απεικονίζονται ως κλάδοι ενός δένδρου οι κλήσεις των αναδρομικών συναρτήσεων) μπορούν να διαπιστωθούν τα ακόλουθα:
  - Το ύψος του δένδρου είναι $\log n$
  - Ο αριθμός των επιπέδων του δένδρου είναι $\log n + 1$
  - Σε κάθε επίπεδο j υπάρχουν 2^j υποπροβλήματα και το καθένα από αυτά έχει μέγεθος $\frac{n}{2^j}$

Αν θεωρήσουμε ότι η συγχώνευση των ταξινομημένων πινάκων εκτελείται σε cn χρόνο τότε ο αλγόριθμος merge-sort εκτελείται σε $cn \log n + cn$ χρόνο. Αυτό συμβαίνει διότι ο αριθμός των λειτουργιών που εκτελούνται σε κάθε επίπεδο j είναι ίσος με: (αριθμό υποπροβλημάτων) $\times$ (μέγεθος κάθε υποπροβλήματος) $2^j c\frac{n}{2^j} = cn$. *Λόγω του ότι ο αριθμός των επιπέδων στο δένδρο είναι $\log n + 1$ προκύπτει ότι ο συνολικός αριθμός λειτουργιών είναι* $cn(\log n + 1) = cn \log n + cn$.

Γρήγορη ταξινόμηση (quick-sort) (1/7)

- Προτάθηκε από τον C.A.R. (Tony) Hoare το 1961.
- Επιλέγει ένα στοιχείο (το pivot) και διαμερίζει με βάση αυτό το στοιχείο την ακολουθία σε 2 υποακολουθίες, την αριστερή που περιέχει τιμές μικρότερες του pivot και τη δεξιά που περιέχει τιμές μεγαλύτερες του pivot. Ο διαμερισμός θα πρέπει να γίνεται σε χρόνο $O(n)$
- Η διαδικασία επαναλαμβάνεται αναδρομικά.
- Η επιλογή του pivot γίνεται χρησιμοποιώντας κάποια στρατηγική (πρώτο στοιχείο από αριστερά, μεσαίο στοιχείο, τυχαία κ.α.)

Η συνάρτηση διαμερισμού (partition) της γρήγορης ταξινόμησης (2/7)

- Ο διαμερισμός μπορεί να γίνει εύκολα με τη χρήση βοηθητικού πίνακα αλλά δημιουργείται το πρόβλημα της απαίτησης επιπλέον μνήμης.
- Ο ακόλουθος αλγόριθμος παρουσιάζει μια in-place έκδοση του διαμερισμού.

```
#include <iostream>
using namespace std;
int partition(int a[], int left, int right) {
    int p = left, i = left + 1;
    for (int j = left + 1; j <= right; j++)
        if (a[j] < a[p]) { swap(a[j], a[i]); i++; }
    }
    swap(a[p], a[i - 1]);
    return i - 1;
}
int main() {
    int a[] = {5, 8, 2, 6, 7, 3}; int n = sizeof(a) / sizeof(a[0]);
    int pos = partition(a, 0, n - 1);
    cout << "Pivot new pos: " << pos << endl;
    for (int i = 0; i < n; i++) cout << a[i] << " ";
}
```

Γρήγορη ταξινόμηση (κώδικας) (3/7)

```
#include <iostream>
using namespace std;
int partition(int a[], int left, int right) {
    int p = left, i = left + 1;
    for (int j = left + 1; j <= right; j++)
        if (a[j] < a[p]) { swap(a[j], a[i]); i++; }
    }
    swap(a[p], a[i - 1]);
    return i - 1;
}
void quick_sort(int a[], int l, int r) {
    if (l >= r) return;
    else {
        int p = partition(a, l, r);
        quick_sort(a, l, p - 1); quick_sort(a, p + 1, r);
    }
}
int main() {
    int a[] = {5, 8, 2, 6, 7, 3}; int n = sizeof(a) / sizeof(a[0]);
    quick_sort(a, 0, n - 1);
    for (int i = 0; i < n; i++) cout << a[i] << " ";
}
```

Γρήγορη ταξινόμηση (απόδοση) (4/7)

- Η μέση απόδοση του αλγορίθμου είναι $\mathrm{O}(n \log n)$ ενώ η χειρότερη απόδοση του αλγορίθμου είναι $\mathrm{O}(n^2)$.
- Η $\mathrm{O}(n^2)$ συμπεριφορά του quick-sort παρατηρείται όταν η διαμέριση σε κάθε αναδρομικό βήμα χωρίζει τα n στοιχεία σε ένα άδειο σύνολο και σε ένα σύνολο με όλα τα υπόλοιπα στοιχεία πλην του pivot (δηλαδή $n-1$ στοιχεία).
- Η επιλογή του pivot με τυχαίο τρόπο επιτρέπει στον quick-sort συχνά να είναι γρηγορότερος από ανταγωνιστικούς αλγορίθμους.

Γρήγορη ταξινόμηση (ανάλυση) (5/7)

- Στην ιδανική περίπτωση η συνάρτηση διαμέρισης χωρίζει την ακολουθία τιμών στη μέση. Αν αυτό μπορούσε να συμβεί σε κάθε αναδρομή τότε ο χρόνος εκτέλεσης της quick-sort θα ήταν $T(n) = 2T(\frac{n}{2}) + \mathrm{O}(n)$ όπου $\mathrm{O}(n)$ είναι ο χρόνος που απαιτεί η διαμέριση. Εφαρμόζοντας το master θεώρημα προκύπτει ότι σε αυτή την περίπτωση η πολυπλοκότητα της quick-sort είναι $\mathrm{O}(n \log n)$.
- Εναλλακτικά, και ξεκινώντας πάλι από τον τύπο $T(n) = 2T(\frac{n}{2}) + \mathrm{O}(n)$ και πραγματοποιώντας αντικαταστάσεις προκύπτει: $\rightarrow 2\left[2(T(\frac{n}{4}) + \mathrm{O}(\frac{n}{2})\right] + \mathrm{O}(n) \rightarrow \cdots \rightarrow 2^k T(\frac{n}{2^k}) + \mathrm{O}(k * n)$. Λόγω του ότι το ανωτέρω ανάπτυγμα τελειώνει όταν $2^k = n \rightarrow (k = \log n)$ και επειδή ο χρόνος επίλυσης με μέγεθος προβλήματος 1 μπορεί να θεωρηθεί σταθερός c προκύπτει ότι $T(n) = n * c + \mathrm{O}(n \log n)$ και καθώς το $\mathrm{O}(n \log n)$ είναι ασυμπτωτικά μεγαλύτερο από το cn προκύπτει ότι ο χρόνος εκτέλεσης μπορεί να γραφεί ως $\mathrm{O}(n \log n)$

Παραλλαγές της γρήγορης ταξινόμησης (6/7)

- Χρήση της ταξινόμησης με εισαγωγή όταν το μέγεθος μιας διαμέρισης είναι μικρό (π.χ. ≤ 10).
- Επιλογή του pivot χρησιμοποιώντας τη διάμεσο των τριών (δηλαδή τη διάμεσο ανάμεσα στο πρώτο, στο τελευταίο και στο μεσαίο στοιχείο της ακολουθίας).
- Κλήση της αναδρομικής συνάρτησης πρώτα για τη μικρότερη ακολουθία τιμών από τις δύο που δημιουργεί η διαμέριση και μετά για τη μεγαλύτερη ακολουθία τιμών. Με αυτό τον τρόπο επιτυγχάνεται μείωση μεγέθους που απαιτείται στο stack.

Introspective sort (7/7)

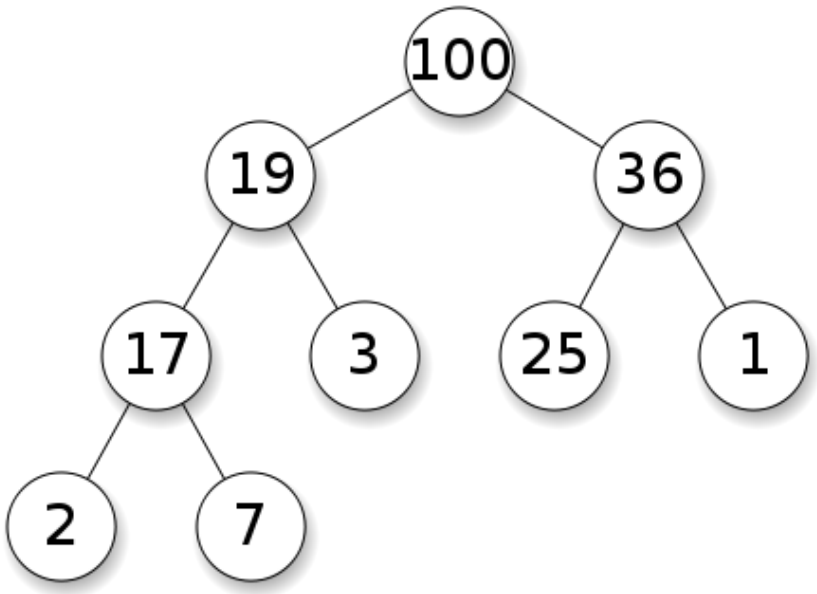
Πρόκειται για τον αλγόριθμο που χρησιμοποιείται από την STL της C++ από το 2000. Εκτιμά το πλήθος αναδρομών που θα απαιτηθεί από την quick-sort και αν ξεπερνά ένα όριο τότε η ταξινόμηση γίνεται με την heap-sort. Αλλιώς ενεργοποιείται η quick-sort με επιλογή pivot με την μέθοδο της διαμέσου των τριών (median of three) και εφαρμόζοντας την ταξινόμηση με εισαγωγή όταν προκύπτουν μικρές ακολουθίες τιμών. Προτάθηκε από τον D. Musser το 1997.

Σωροί (heaps) και ταξινόμηση σωρού (heap-sort)

- Ο σωρός είναι μια **μερικά** ταξινομημένη δομή δεδομένων που υποστηρίζει τις ακόλουθες λειτουργίες:
  - εύρεση του στοιχείου με τη μεγαλύτερη τιμή κλειδιού.
  - διαγραφή του στοιχείου με τη μεγαλύτερη τιμή κλειδιού.
  - προσθήκη νέου κλειδιού στο σωρό.
- Ένας σωρός μπορεί να έχει στη κορυφή το μεγαλύτερο στοιχείο (Max-Heap) ή αντίστοιχα να έχει στη κορυφή το μικρότερο στοιχείο (Min-Heap). Στη δεύτερη περίπτωση οι λειτουργίες της εύρεσης και της διαγραφής αφορούν τη μικρότερη τιμή κλειδιού.

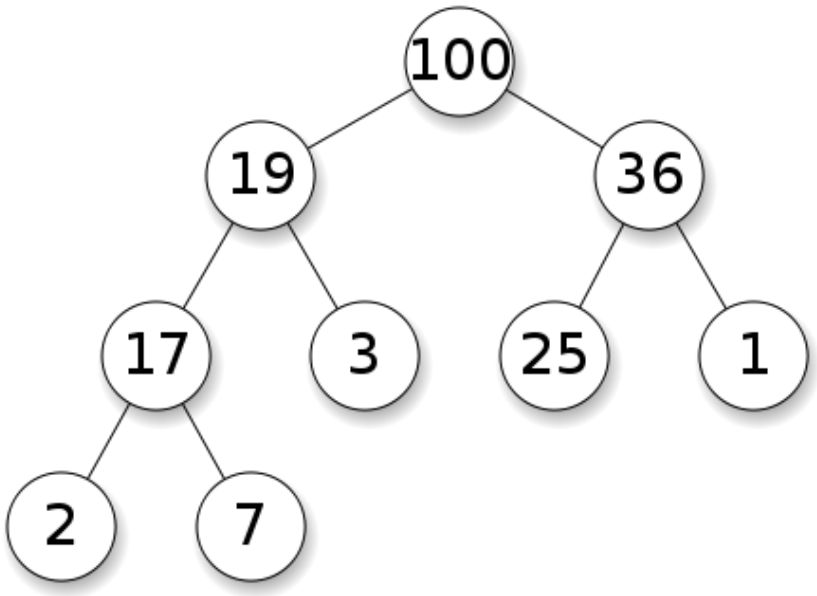
Σωροί

- Ένας σωρός μπορεί να θεωρηθεί ως δυαδικό δένδρο για το οποίο ισχύουν οι ακόλουθοι 2 περιορισμοί:
  1. **πληρότητα**: το δυαδικό δένδρο είναι συμπληρωμένο, δηλαδή όλα τα επίπεδά του είναι πλήρη εκτός πιθανά από το τελευταίο επίπεδο στο οποίο μπορούν να λείπουν μόνο τα δεξιότερα φύλλα.
  2. **κυριαρχία γονέα**: το κλειδί σε κάθε κορυφή είναι μεγαλύτερο ή ίσο από τα κλειδιά των παιδιών (σε Max-Heap).



Ιδιότητες σωρών

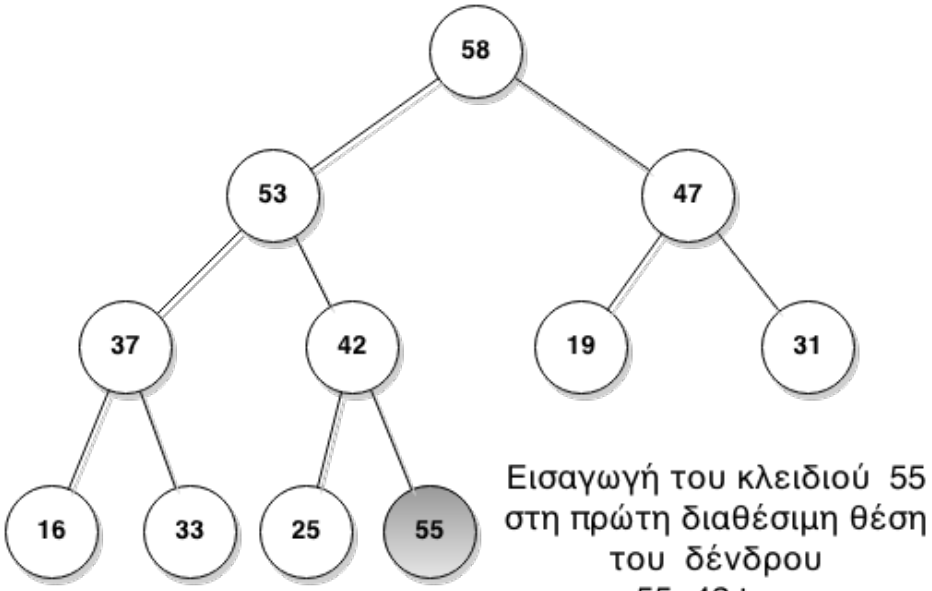
- Η ρίζα του σωρού περιέχει το μεγαλύτερο στοιχείο.
- Κάθε κόμβος του σωρού μαζί με όλους τους απογόνους του είναι και αυτός σωρός.
- Τα κλειδιά είναι ταξινομημένα από πάνω προς τα κάτω δηλαδή τα κλειδιά σε κάθε διαδρομή από την κορυφή προς τα φύλλα είτε μειώνονται είτε παραμένουν στην ίδια τιμή.
- Τα κλειδιά σε κάθε επίπεδο του δένδρου δεν είναι ταξινομημένα.
- Τα παιδιά κάθε κόμβου δεν είναι ταξινομημένα.



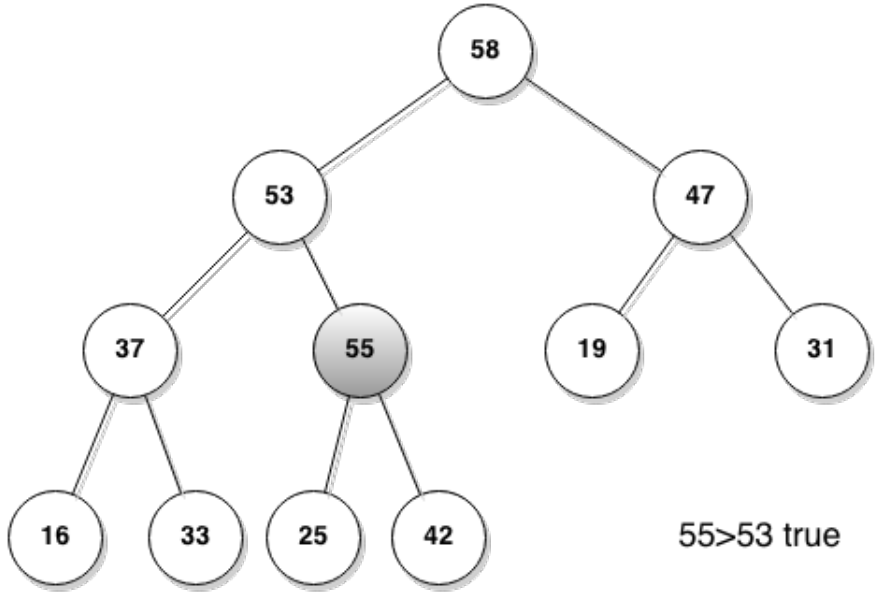
Εισαγωγή νέου στοιχείου στο σωρό (1/2)

- Το νέο κλειδί εισάγεται ως φύλλο στο δένδρο όσο πιο αριστερά γίνεται.
- Το νέο κλειδί επαναληπτικά συγκρίνεται με τον γονέα του και αν είναι μεγαλύτερο τον αντικαθιστά.

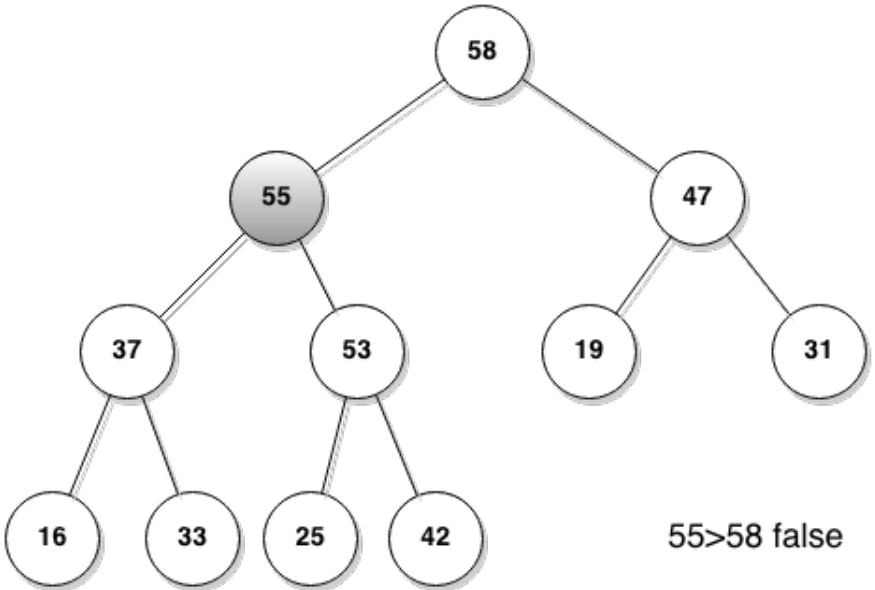
Εισαγωγή νέου στοιχείου στο σωρό (2/2)



Εισαγωγή του κλειδιού 55 στη πρώτη διαθέσιμη θέση του δένδρου
55>42 true



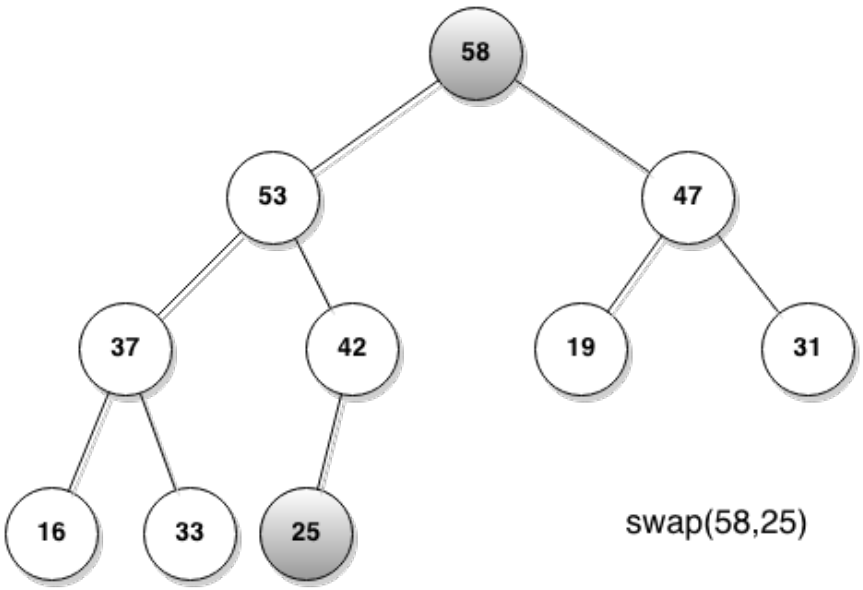
55>53 true



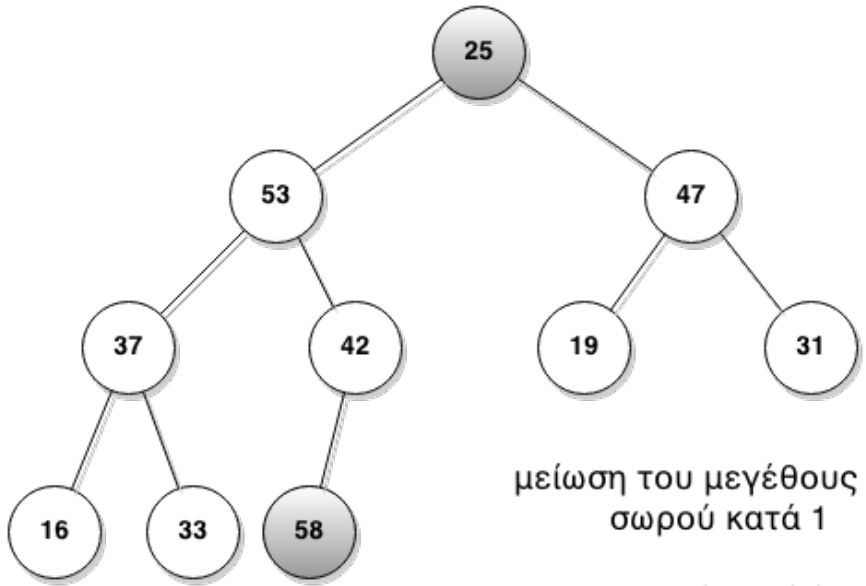
55>58 false

Διαγραφή ρίζας σωρού (1/2)

- Η ρίζα αντιμετατίθεται με το στοιχείο που βρίσκεται στην τελευταία σειρά στην πλέον δεξιά θέση.
- Το κλειδί που αντιμετατέθηκε με τη ρίζα συγκρίνεται με τα παιδιά του και σε περίπτωση που είναι μικρότερο αντιμετατίθεται με το μεγαλύτερο από τα παιδιά του.
- Η διαδικασία επαναλαμβάνεται μέχρι το κλειδί που αντιμετατέθηκε με τη ρίζα να είναι μεγαλύτερο ή ίσο από τα παιδιά του.



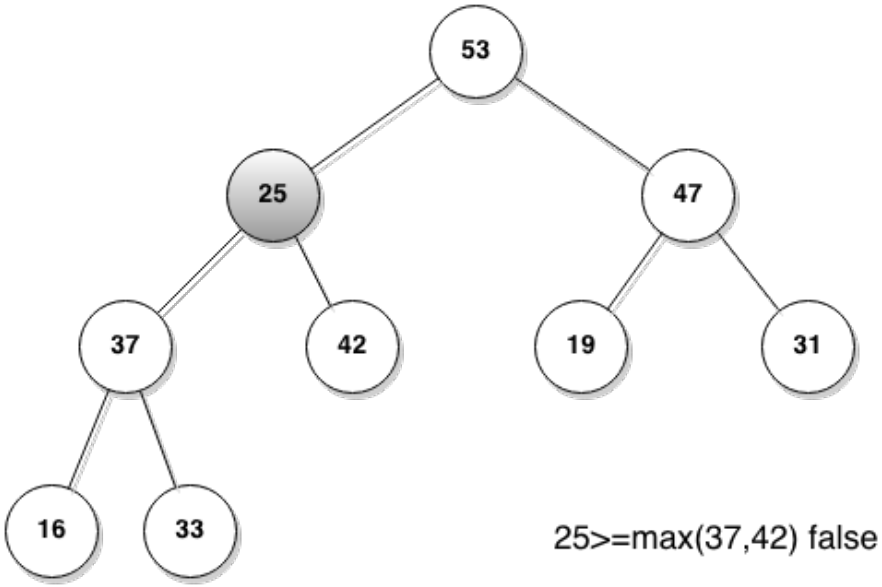
swap(58,25)

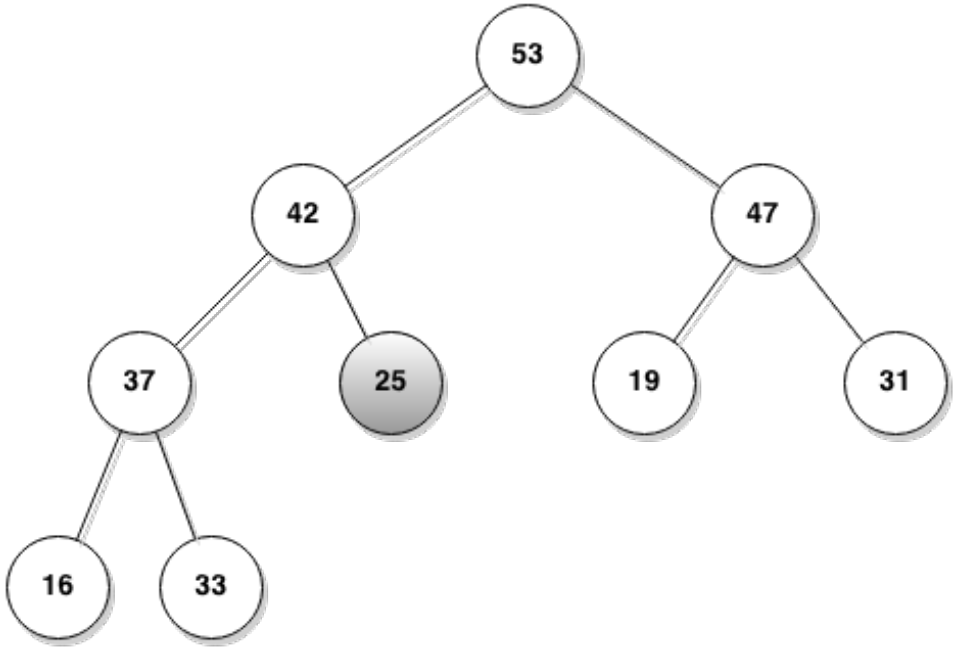


μείωση του μεγέθους του σωρού κατά 1

25>=max(53,47) false

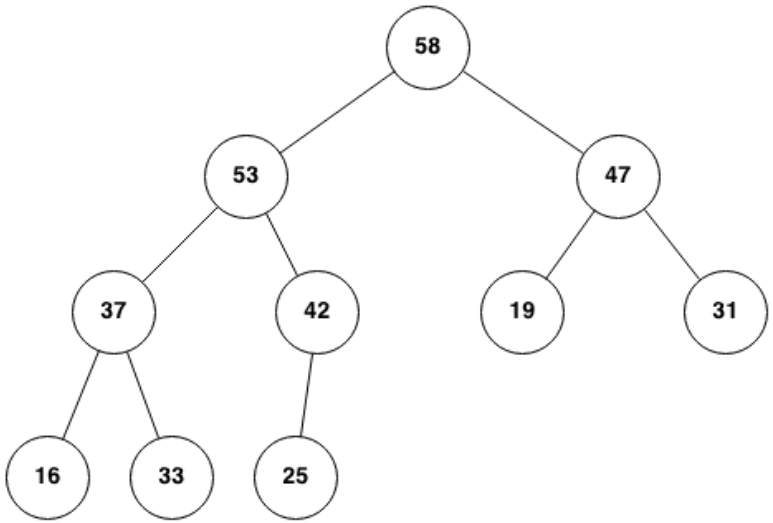
Διαγραφή ρίζας σωρού (2/2)





Η δενδρική δομή του σωρού ως πίνακας

- Ένας σωρός μπορεί να υλοποιηθεί με ένα πίνακα καταγράφοντας τα στοιχεία από πάνω προς τα κάτω και από αριστερά προς τα δεξιά:
  - Τα κελιά γονείς βρίσκονται στις πρώτες $\lfloor \frac{n}{2} \rfloor$ ενώ τα φύλλα καταλαμβάνουν τις τελευταίες $\lceil \frac{n}{2} \rceil$ θέσεις.
  - Τα παιδιά για κάθε κλειδί στις θέσεις i από 1 μέχρι και $\lfloor \frac{n}{2} \rfloor$ βρίσκονται στις θέσεις $2i$ και $2i+1$.
  - Ο γονέας για κάθε κλειδί στις θέσεις i από 2 μέχρι και n βρίσκεται στη θέση $\lfloor \frac{i}{2} \rfloor$.

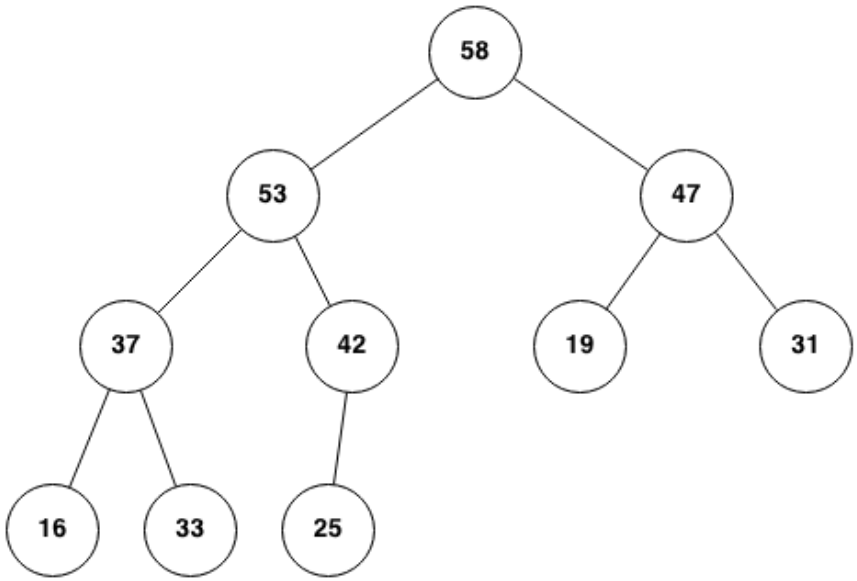


NULL	58	53	47	37	42	19	31	16	33	25
0	1	2	3	4	5	6	7	8	9	10

Σχέσεις στοιχείων πίνακα

Ένας σωρός (Max-Heap) με n στοιχεία μπορεί να οριστεί ως ένας πίνακας $H\left[1 \cdots n\right]$ για τον οποίο ισχύει ότι κάθε στοιχείο στη θέση i στο πρώτο μισό του πίνακα είναι μεγαλύτερο ή ίσο από τα στοιχεία στις θέσεις $2i$ και $2i+1$.

$$H[i] \geq max\left\{H\left[2i\right], H\left[2i+1\right]\right\} \forall i = 1, \cdots, \lfloor \frac{n}{2} \rfloor$$



NULL	58	53	47	37	42	19	31	16	33	25
0	1	2	3	4	5	6	7	8	9	10

Δημιουργία σωρού από κάτω προς τα πάνω (bottom up) 1/2

Ξεκινώντας από τον τελευταίο στη σειρά κόμβο γονέα ο αλγόριθμος ελέγχει αν ικανοποιείται η κυριαρχία γονέα για τη τιμή κλειδιού K που διαθέτει. Αν δεν ισχύει τότε η τιμή κλειδιού K ανταλλάσσεται με την τιμή κλειδιού του μεγαλύτερου από τα κλειδιά των παιδιών και ελέγχεται αν η κυριαρχία γονέα ισχύει για το K στη νέα του θέση. Η διαδικασία συνεχίζεται μέχρι να ικανοποιηθεί για το K η κυριαρχία γονέα. Στη συνέχεια η ίδια διαδικασία ακολουθείται για τον επόμενο κόμβο γονέα και μέχρι να έρθει η σειρά της ρίζας.

Δημιουργία σωρού από κάτω προς τα πάνω (bottom up) (2/2)

```
const int MAX_HEAP_SIZE = 1000; int heap[MAX_HEAP_SIZE + 1];
int heap_size = 0;
void heapify(int k) {
    int v = heap[k]; bool flag = false;
    while (!flag && 2 * k <= heap_size) {
        int j = 2 * k;
        if (j < heap_size && heap[j] < heap[j + 1])
            j++;
        if (v >= heap[j])
            flag = true;
        else {
            heap[k] = heap[j]; k = j;
        }
    }
    heap[k] = v;
}
void heap_bottom_up(int *a, int N) {
    heap_size = N;
    for (int i = 0; i < N; i++)
        heap[i + 1] = a[i];
    for (int i = heap_size / 2; i >= 1; i--)
        heapify(i);
}
```

Εισαγωγή νέου κλειδιού στο σωρό

Αρχικά ο κόμβος προσαρτάται στο τέλος του σωρού (δηλαδή μετά το τελευταίο φύλλο του δένδρου). Στη συνέχεια το κλειδί ανεβαίνει στη κατάλληλη θέση ως εξής: Το K συγκρίνεται με το κλειδί του γονέα του και αν το κλειδί του γονέα είναι μεγαλύτερο η διαδικασία τερματίζει αλλιώς τα 2 κλειδιά αντιμετατίθενται και το κλειδί K συγκρίνεται με το κλειδί του νέου γονέα του. Η διαδικασία συνεχίζεται μέχρι το K να είναι μικρότερο ή ίσο με τον γονέα του ή να έχει φτάσει στη ρίζα του δένδρου (δηλαδή στο στοιχείο 1 του πίνακα).

```
void push(int key) {
    heap_size++;
    heap[heap_size] = key;
    int pos = heap_size;
    while (pos != 1 && heap[pos / 2] < heap[pos]) {
        swap(heap[pos / 2], heap[pos]);
        pos /= 2;
    }
}
```

Διαγραφή μέγιστου κλειδιού από το σωρό

Το κλειδί που βρίσκεται στη ρίζα αντιμετατίθεται με το τελευταίο στοιχείο του σωρού. Το μέγεθος του σωρού μειώνεται κατά 1 και εκτελείται η διαδικασία heapify για το νέο πρώτο στοιχείο του σωρού.

```
void pop() {
    swap(heap[1], heap[heap_size]);
    heap_size--;
    heapify(1);
}
```

Ταξινόμηση σωρού (heap-sort)

- Ο αλγόριθμος heap-sort προτάθηκε από τον J.W.J.Williams το 1964 και αποτελείται από 2 στάδια:
  - Δημιουργία σωρού με τα n στοιχεία ενός πίνακα τα στοιχεία του οποίου ζητείται να ταξινομηθούν.
  - Εφαρμογή της διαγραφής της ρίζας $n-1$ φορές.

```
void heapsort() {
    while (!empty()) pop();
}
int main() {
    int a[] = {42, 37, 31, 16, 53, 19, 47, 58, 52, 44};
    heap_bottom_up(a, 10);
    heapsort();
    for (int i = 1; i <= 10; i++)
        cout << heap[i] << " ";
    cout << endl;
}
```

Τα στοιχεία αφαιρούνται από το σωρό σε φθίνουσα σειρά. Καθώς κατά την αφαίρεσή του κάθε στοιχείο τοποθετείται στο τέλος του σωρού τελικά ο σωρός περιέχει τα αρχικά δεδομένα σε αύξουσα σειρά

Απόδοση του αλγορίθμου heap-sort

Το στάδιο δημιουργίας του σωρού (heap_bottom_up) έχει πολυπλοκότητα $O(n)$ ενώ το στάδιο των διαδοχικών διαγραφών της ρίζας (maximum_key_deletion) έχει πολυπλοκότητα $O(n \log n)$. Συνεπώς η πολυπλοκότητά του Heapsort είναι $O(n) + O(n \log n) = O(n \log n)$.

- Ο αλγόριθμος heap-sort είναι in place.
- Στην πράξη έχει συγκρίσιμη ταχύτητα εκτέλεσης με το merge-sort αλλά είναι λιγότερο γρήγορος από το quick-sort.

Άλλοι αλγόριθμοι ταξινόμησης

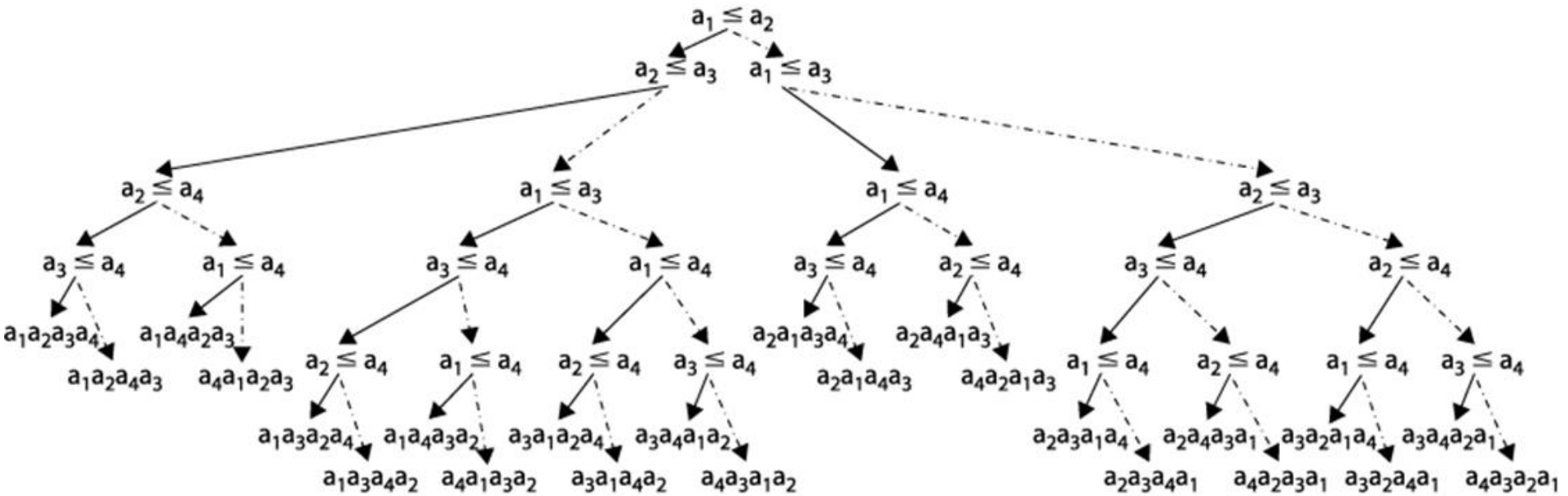
- Ταξινόμηση φυσαλίδας (bubble sort)
- Ταξινόμηση με επιλογή (selection sort)
- Ταξινόμηση με κατάταξη (rank sort)
- TimSort

Όριο απόδοσης αλγορίθμων ταξινόμησης (με σύγκριση στοιχείων) (1/3)

Ένα βασικό συμπέρασμα της επιστήμης υπολογιστών είναι ότι δεν υπάρχει αλγόριθμος που να ταξινομεί μια συλλογή n στοιχείων συγκρίνοντάς τα στοιχεία μεταξύ τους και ο οποίος να έχει απόδοση καλύτερη από $\mathrm{O}(n \log n)$ στη χειρότερη περίπτωση.
Εναλλακτικά, το ίδιο συμπέρασμα μπορεί να διατυπωθεί ως εξής: Όλοι οι αλγόριθμοι ταξινόμησης με σύγκριση στοιχείων είναι $\Omega(n \log n)$.

Όριο απόδοσης αλγορίθμων ταξινόμησης (με σύγκριση στοιχείων) (2/3)

- Έστω ένα δυαδικό δένδρο:
  - Η ρίζα του δένδρου αντιστοιχεί στην αρχική κατάσταση των δεδομένων (δηλαδή μη ταξινομημένα δεδομένα) ενώ στα φύλλα του δένδρου υπάρχουν οι καταστάσεις που αντιστοιχούν στην ταξινομημένη ακολουθία (τα φύλλα του δένδρου είναι $n!$).
  - Κάθε εσωτερικός κόμβος αναπαριστά τη σύγκριση δύο στοιχείων ($a_i \leq a_j$) οδηγώντας σε δύο νέους κόμβους.
  - Ο χρόνος χειρότερης περίπτωσης αντιστοιχεί στο ύψος του δένδρου.



Όριο απόδοσης αλγορίθμων ταξινόμησης (με σύγκριση στοιχείων) (3/3)

Ένας αλγόριθμος ταξινόμησης δημιουργεί μια μετάθεση (αλλαγή των θέσεων) των δεδομένων εισόδου του. Το δένδρο απόφασης που αντιστοιχεί στον αλγόριθμο ταξινόμησης θα πρέπει να έχει ένα φύλλο για κάθε μια από τις μεταθέσεις των n στοιχείων, άρα $n!$ φύλλα. Το ελάχιστο ύψος ενός δυαδικού δένδρου με $n!$ φύλλα είναι $\log n! \approx n \log n - n$ (όπως προκύπτει από τον τύπο του Stirling $\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}(\frac{n}{e})^n} = 1$). Συνεπώς η απόδοση ενός αλγορίθμου ταξινόμησης που χρησιμοποιεί συγκρίσεις στοιχείων δεν μπορεί να είναι καλύτερη από $O(n \log n)$.

Ωστόσο, αν τα δεδομένα έχουν κάποια ιδιαίτερα χαρακτηριστικά τότε η ταξινόμηση μπορεί να γίνει ταχύτερα. Για παράδειγμα, η ταξινόμηση n ακεραίων τιμών που κυμαίνονται σε εύρος τιμών $[0, k)$ με k πολύ μικρότερο του n μπορεί να γίνει σε χρόνο $O(n)$ με τον αλγόριθμο counting-sort.

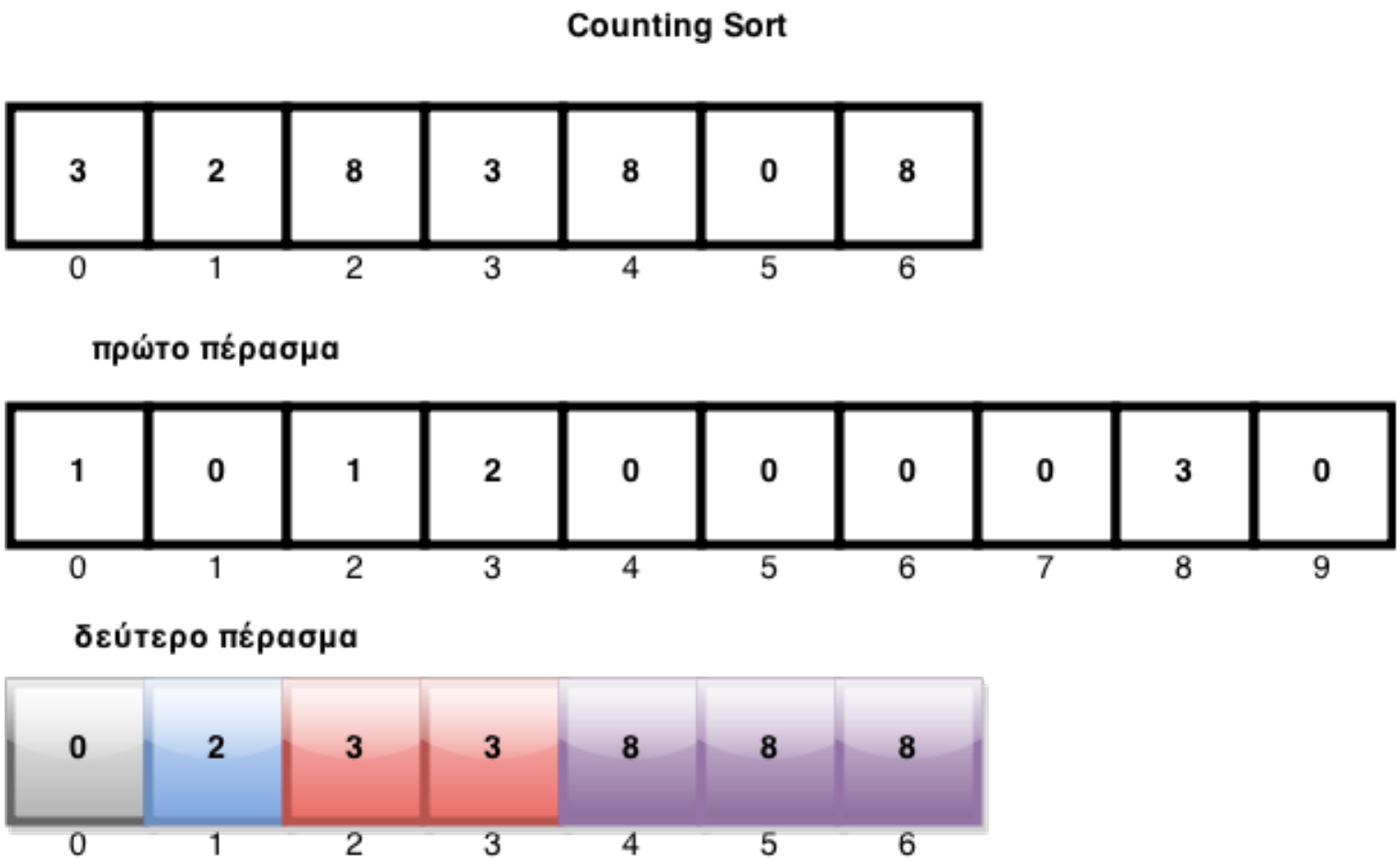
Αλγόριθμοι ταξινόμησης χωρίς σύγκριση στοιχείων

- Ταξινόμηση με καταμέτρηση (counting sort)
- Ταξινόμηση ρίζας (radix sort)
- Ταξινόμηση κάδου (bucket sort)

Ταξινόμηση με καταμέτρηση (counting sort) 1/3

- Ο αλγόριθμος Counting Sort προτάθηκε από τον Harold Seward το 1954. Δεν στηρίζεται σε σύγκριση αλλά σε απαρίθμηση των τιμών.
- Ο αλγόριθμος counting-sort μετρά πόσες φορές παρατηρείται η κάθε διακριτή τιμή της ακολουθίας εισόδου (δημιουργεί έναν πίνακα συχνοτήτων). Στη συνέχεια, από το μικρότερο προς το μεγαλύτερο δείκτη του πίνακα συχνοτήτων καταγράφει στον αρχικό πίνακα την τιμή του δείκτη i τόσες φορές όσες η αριθμητική τιμή που περιέχεται στον πίνακα συχνοτήτων στη θέση i.
- Έχει πολυπλοκότητα $O(n + k)$ όπου k είναι το εύρος των διακριτών τιμών και n είναι το μέγεθος της εισόδου.
- Για να μπορεί να εφαρμοστεί αποδοτικά θα πρέπει οι τιμές εισόδου να κυμαίνονται σε ένα γνωστό (και μικρό) εύρος τιμών και ο πίνακας συχνοτήτων των τιμών να μπορεί να υπολογιστεί γρήγορα.

Ταξινόμηση με καταμέτρηση (counting sort) 2/3



Ταξινόμηση με καταμέτρηση (counting sort) 3/3

```
#include <iostream>
using namespace std;
void count_sort(int a[], int n, int k) {
    int i; int *b = new int[k]{0};
    for (i = 0; i < n; i++)
        b[a[i]]++;
    int j = 0;
    for (i = 0; i < k; i++)
        while (b[i] > 0) {
            b[i]--;
            a[j] = i;
            j++;
        }
    delete[] b;
}
int main() {
    int a[] = {3, 2, 8, 3, 8, 0, 8};
    int n = sizeof(a) / sizeof(a[0]);
    count_sort(a, n, 9);
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
}
```

Ταξινόμηση κάδου (bucket sort) (1/3)

- Ο αλγόριθμος bucket-sort είναι χρήσιμος όταν τα δεδομένα είναι σχετικά ομοιόμορφα κατανεμημένα σε ένα εύρος τιμών.
  - Δημιουργεί n αριθμημένους κάδους (buckets ή bins) και τοποθετεί κάθε στοιχείο σε έναν από τους κάδους ανάλογα με την τιμή του (για κάθε 2 κάδους A και B θα πρέπει να ισχύει ότι όλα τα στοιχεία του κάδου A είναι μικρότερα από όλα τα στοιχεία του κάδου B ή το αντίστροφο)
  - Ταξινομεί τα στοιχεία σε κάθε κάδο ξεχωριστά χρησιμοποιώντας κάποιον αλγόριθμο ταξινόμησης (π.χ. insertion-sort)
  - Συνενώνει τα περιεχόμενα των κάδων έτσι ώστε να δημιουργηθεί η τελική ταξινομημένη ακολουθία.
- Υπάρχει και έκδοση του bucket sort που λειτουργεί αναδρομικά δηλαδή τα στοιχεία κατανέμονται αρχικά σε κάδους και σε κάθε κάδο κατανέμονται εκ νέου τα στοιχεία του σε άλλους κάδους μέχρι το μέγεθος των κάδων να γίνει μικρό.

Ταξινόμηση κάδου (bucket-sort) (2/3)

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void bucket_sort(float arr[], int n, int bins) {
    vector<float> b[bins];
    for (int i = 0; i < n; i++) {
        int bi = bins * arr[i];
        b[bi].push_back(arr[i]);
    }
    for (int i = 0; i < bins; i++)
        sort(b[i].begin(), b[i].end());
    int index = 0;
    for (int i = 0; i < bins; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}
int main() {
    int bins=3;
    float a[] = {0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434};
    int n = sizeof(a) / sizeof(a[0]);
    bucket_sort(a, n, bins);
    for (int i = 0; i < n; i++)  cout << a[i] << " ";
}
```

Ταξινόμηση κάδου (bucket-sort) (3/3)

Η πολυπλοκότητα του bucket-sort είναι $\mathrm{O}(mC(\frac{n}{m}))$ όπου n είναι το πλήθος των στοιχείων εισόδου, m είναι ο αριθμός των κάδων και $C(x)$ είναι η πολυπλοκότητα του αλγορίθμου ταξινόμησης που θα χρησιμοποιηθεί για την ταξινόμηση των στοιχείων κάθε κάδου.

Ταξινόμηση ρίζας (radix sort) 1/3

- Η ταξινόμηση radix sort είναι ταξινόμηση ψηφίο προς ψηφίο ξεκινώντας από τα λιγότερο σημαντικά ψηφία και συνεχίζοντας προς τα περισσότερο σημαντικά. Χρησιμοποιεί μια άλλη μέθοδο ταξινόμησης (η οποία θα πρέπει να είναι stable) ως υπορουτίνα.
- Παράδειγμα ταξινόμησης radix sort. Έστω η ακολουθία εισόδου: 341, 167, 43, 99, 777, 51, 92, 80, 60, 101
  - Ταξινόμηση με το ψηφίο των μονάδων: 80, 60, 341, 51, 101, 92, 43, 167, 777, 99
  - Ταξινόμηση με το ψηφίο των δεκάδων: 101, 341, 43, 51, 60, 167, 777, 80, 92, 99
  - Ταξινόμηση με το ψηφίο των εκατοντάδων: 43, 51, 60, 80, 92, 99, 101, 167, 341, 777

Ταξινόμηση ρίζας (radix sort) 2/3

```
#include <iostream>
using namespace std;
void count_sort_digit_by_digit(int a[], int n, int d) {
    int b[10] = {0}; int temp[n];
    for (int i = 0; i < n; i++)
        b[(a[i] / d) % 10]++;
    for (int i = 1; i < 10; i++)
        b[i] += b[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        int x = (a[i] / d) % 10;
        temp[b[x] - 1] = a[i]; b[x]--;
    }
    for (int i = 0; i < n; i++)
        a[i] = temp[i];
}
void radix_sort(int a[], int n, int m) {
    int d = 1;
    for (int i = 0; i < m; i++)
        count_sort_digit_by_digit(a, n, d); d *= 10;
}
int main() {
    int a[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(a) / sizeof(a[0]);  radix_sort(a, n, 3);
    for (int i = 0; i < n; i++) cout << a[i] << " ";
}
```

Ταξινόμηση ρίζας (radix sort) 3/3

Η απόδοση του αλγορίθμου εξαρτάται από το πλήθος των ψηφίων που απαιτούνται για την αναπαράσταση των τιμών εισόδου. Η αλγοριθμική πολυπλοκότητα του Radix Sort είναι $O(kn)$, όπου k είναι το πλήθος των ψηφίων και n είναι το μέγεθος της εισόδου.

Σειριακή αναζήτηση (linear search) 1/2

- Η σειριακή ή γραμμική αναζήτηση είναι ο απλούστερος αλγόριθμος αναζήτησης. Τα στοιχεία του πίνακα εξετάζονται στην σειρά μέχρι να εντοπιστεί το στοιχείο που αναζητείται.
- Μπορεί να εφαρμοστεί σε μη ταξινομημένους πίνακες
- Η σειριακή αναζήτηση έχει πολυπλοκότητα $O(n)$

Σειριακή αναζήτηση (linear search) 2/2

```
#include <iostream>
using namespace std;
int linear_search(int a[], int n, int key) {
    for (int i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
int main(){
    int a[]={4,6,2,17,9,11,16};
    int n = sizeof(a)/sizeof(a[0]);
    int key = 11; cout << linear_search(a,n,key) << endl;
    key = 13; cout << linear_search(a,n,key) << endl;
}
```

Αναζήτηση με αναπήδηση (jump search) 1/2

- Η αναζήτηση με αναπηδήσεις μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα.
- Επιλέγεται ένα βήμα k και γίνεται σταδιακός έλεγχος των στοιχείων a[0], a[k-1], a[2k-1], ...
- Όταν βρεθεί ότι η τιμή που αναζητείται είναι ανάμεσα σε 2 από τις παραπάνω τιμές τότε γίνεται σειριακή αναζήτηση στο διάστημα αυτό.
- Αν επιλεγεί ως βήμα η τιμή $\sqrt{n}$ όπου n το πλήθος των στοιχείων της ακολουθίας τότε η πολυπλοκότητα χειρότερης περίπτωσης για τον αλγόριθμο είναι $O(\sqrt{n})$

Αναζήτηση με αναπήδηση (jump search) 2/2

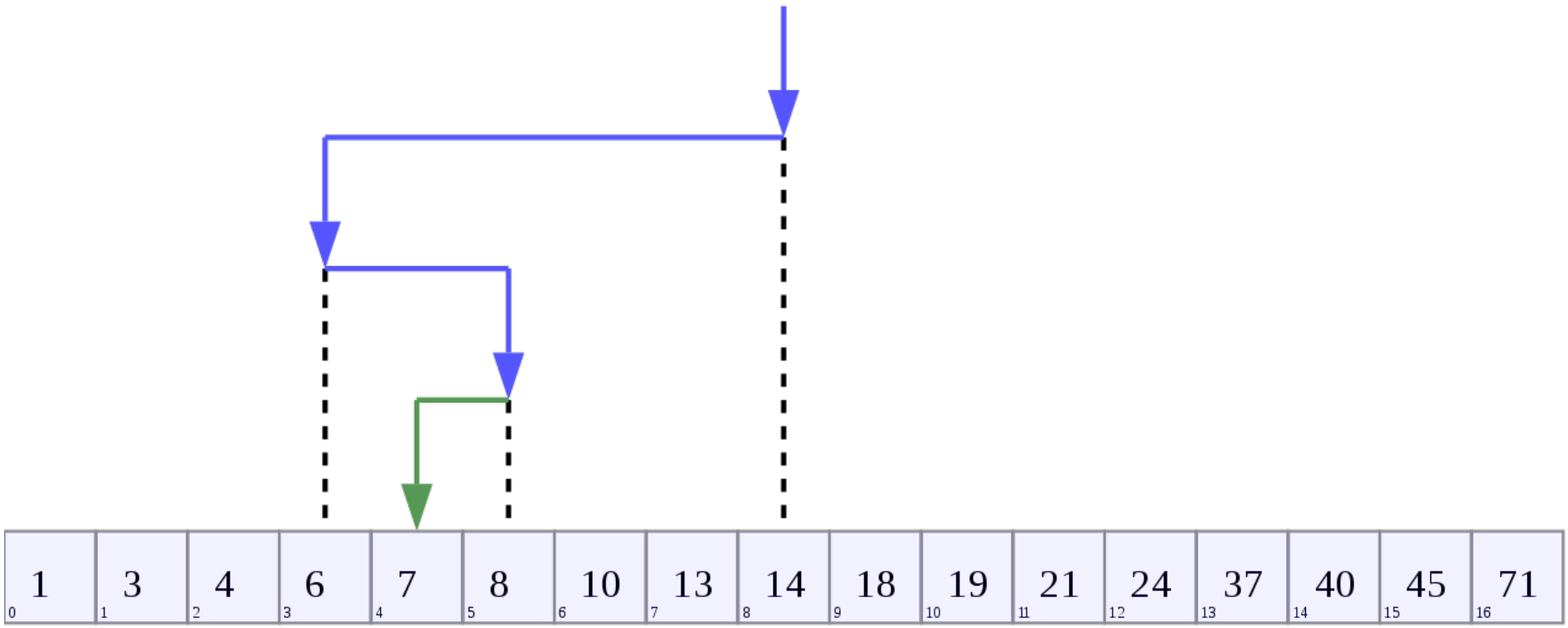
```
#include <iostream>
#include <cmath>
using namespace std;
int jump_search(int a[], int n, int key) {
    int step = (int)sqrt(n);
    int p1 = 0; int p2 = step;
    while (a[p2] < key) {
        if (a[p1] >= key)
            break;
        p1 = p2 + 1; p2 += step;
        if (p2 > n - 1)
            p2 = n - 1;
    }
    for (int i = p1; i <= p2; i++)
        if (a[i] == key)
            return i;
    return -1;
}
int main() {
    int a[] = {1,3,4,6,7,8,10,13,14,18,19,21,24,37,40,45,71};
    int n = sizeof(a) / sizeof(a[0]);
    int key = 7; cout << jump_search(a, n, key) << endl;
    key=15; cout << jump_search(a, n, key) << endl;
}
```

Δυαδική αναζήτηση (1/3)

- Η δυαδική αναζήτηση μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα.
- Διαιρεί επαναληπτικά την ακολουθία σε 2 υποακολουθίες και εξετάζοντας το στοιχείο στο οποίο γίνεται η διαίρεση της ακολουθίας απορρίπτει την υποακολουθία στην οποία λογικά συμπεραίνεται ότι δεν μπορεί να βρεθεί το στοιχείο που ζητείται.
- Έχει πολυπλοκότητα χειρότερης περίπτωσης $\mathrm{O}(\log n)$.

Δυαδική αναζήτηση (2/3)

Για την ακολουθία ταξινομημένων τιμών $\{1,3,4,6,7,8,10,13,14,18,19,21,24,37,40,45,71\}$ *ζητείται η εύρεση της θέσης της τιμής 7.*



Δυαδική αναζήτηση - αναδρομική λύση (3/3)

```
#include <iostream>
using namespace std;
int binary_search(int a[], int l, int r, int key) {
    int m = (l + r) / 2;
    if (l > r)
        return -1;
    else if (a[m] == key)
        return m;
    else if (key < a[m])
        return binary_search(a, l, m - 1, key);
    else
        return binary_search(a, m + 1, r, key);
}
int binary_search(int a[], int n, int key) {
    return binary_search(a, 0, n - 1, key);
}
int main() {
    int a[] = {1,3,4,6,7,8,10,13,14,18,19,21,24,37,40,45,71};
    int n = sizeof(a) / sizeof(a[0]);
    int key = 7; cout << binary_search(a, n, key) << endl;
    key = 15; cout << binary_search(a, n, key) << endl;
}
```

Αναζήτηση με παρεμβολή (interpolation search) 1/2

- Η αναζήτηση με παρεμβολή μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα.
- Αντί να χρησιμοποιηθεί η τιμή $\frac{1}{2}$ για να μοιραστεί η ακολουθία δεδομένων σε δύο υποακολουθίες όπως στη δυαδική αναζήτηση, υπολογίζεται μια τιμή η οποία εκτιμάται ότι οδηγεί πλησιέστερα στο στοιχείο που αναζητείται.
- Αν l είναι ο δείκτης του αριστερότερου στοιχείου της ακολουθίας και r ο δείκτης του δεξιότερου στοιχείου της ακολουθίας τότε υπολογίζεται ο συντελεστής $c = \frac{key - a[l]}{a[r] - a[l]}$ όπου key είναι το στοιχείο προς αναζήτηση και a είναι η ακολουθία στοιχείων στην οποία αναζητείται.
- Έχει πολυπλοκότητα χειρότερης περίπτωσης $O(n)$ αλλά **αν τα δεδομένα της ακολουθίας είναι ομοιόμορφα κατανεμημένα σε ένα εύρος τιμών** τότε η πολυπλοκότητα του αλγορίθμου γίνεται $O(\log \log n)$

Αναζήτηση με παρεμβολή (interpolation search) 2/2

```
#include <iostream>
using namespace std;
int interpolation_search(int a[], int l, int r, int key) {
    int m;
    if (l > r) return -1;
    else if (l == r) m = l;
    else {
        double c = (double)(key - a[l]) / (double)(a[r] - a[l]);
        if ((c < 0) || (c > 1)) return -1;
        m = (int)(l + (r - l) * c);
    }
    if (a[m] == key) return m;
    else if (key < a[m])
        return interpolation_search(a, l, m - 1, key);
    else return interpolation_search(a, m + 1, r, key);
}
int interpolation_search(int a[], int n, int key) {
    return interpolation_search(a, 0, n - 1, key);
}
int main() {
    int a[] = {1,3,4,6,7,8,10,13,14,18,19,21,24,37,40,45,71};
    int n = sizeof(a) / sizeof(a[0]);
    int key = 7; cout << interpolation_search(a, n, key) << endl;
    key = 15; cout << interpolation_search(a, n, key) << endl;
}
```