Sortieren und Suchen

Vortrag im Hauptseminar Hallo Welt!

Johannes Schlumberger
spjsschl@cip.informatik.uni-erlangen.de

Friedrich-Alexander-Universität Erlangen/Nürnberg

1. Mai 2006

Teil I

Einführung

Das Sortierproblem

1. informell:
   - $5, 1, 4, 6, 9, 7, 8, 2, 0, 3$
   - $\downarrow$
   - $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$

Das Sortierproblem

2. formal:
   - gegeben: Liste von Datensätzen $[D_1, D_2, \ldots, D_n]$
     - jeder Datensatz D_i hat Schlüssel k_i
     - totale Ordnung auf der Menge aller möglichen Schlüssel ($\leq$)
   - gesucht: Permutation $\pi \in \sigma_n$ (σ_n = Menge der Permutationen von $\{1, 2, 3, \ldots, n\}$) mit
     - $k_{\pi^{-1}(1)} \leq \cdots \leq k_{\pi^{-1}(n)}$ oder
     - $k_{\pi^{-1}(1)} \geq \cdots \geq k_{\pi^{-1}(n)}$
   - hierbei bezeichne π^{-1} die Umkehrfunktion der Permutation π

Das Suchproblem

1. informell: Welcher Wert gehört zu 7?

Schlüssel	0	1	9	3	4	6	7	8	2	5
Satz	9	4	1	2	4	7	6	2	5	2

↓

6

Das Suchproblem

2. formal:
   - gegeben: Eine Menge *Schlüssel*, eine (endliche) (Multi-)Menge *Sätze*, ein $k \in \mathit{Schlüssel}$ sowie eine Funktion f, die aus der Menge der Schlüssel in die Menge der Sätze abbildet.
   - gesucht: $x \in \mathit{Sätze}$ mit $f(k) = x$ oder Fehlschlag

Vereinbarungen

- Vergleichsfunktion *compare* steht zur Verfügung

$$compare(x, y) = \begin{cases} 1 & falls\ x > y \\ -1 & falls\ x < y \\ 0 & falls\ x = y \end{cases}$$

  x, y Schlüssel
- wir arbeiten stets auf Feldern – wahlfreier Zugriff
- mit Zeigern (Referenzen) – kein Umkopieren von Datensätzen
- keine externen Speichern
- eindeutige Schlüssel
- Schlüssel – isomorph abbildbar auf die natürlichen Zahlen
- Suche nur Schlüssel; angehängte Daten sind dann leicht zu erreichen (z.B. Schlüssel i: Daten an Position $i + 1$, oder Position i in anderer Datei).

Komplexitätsgrößen

- die Anzahl der benötigten paarweisen Vergleiche
- die Anzahl der benötigten Zuweisungen
- der benötigte Speicherplatz

Kosten werden uniform gemessen.

Warum ist das interessant?

Computer manufacturers in the 1960s estimated that more than 25 percent of the running time of computer were spent sorting.[. . .] In fact, there were many installations in which tasks of sorting were responsible for more than half of the computing time.

Donald E. Knuth

Warum ist das wichtig?

Suchen und Sortieren taucht als

- Problem an sich

oder als

- Teilproblem eines übergeordneten Problems

beinahe überall auf, wo Programme oder Daten eine Rolle spielen.

- Test auf Eindeutigkeit (uniqueness)
- Eliminierung von Duplikaten
- Vergeben von Prioritäten
- Mediansuche/Elementauswahl(das k-te Element)
- Auftrittshäufigkeiten zählen
- Sortieren ermöglicht effizientes Suchen (die beiden Probleme hängen zusammen)

Teil II

Sorting I

Sortieren durch Vertauschen (Bubblesort)

Bubblesort – die Idee

Eine Liste ist sortiert wenn nie ein größeres Element vor einem kleineren steht.
Betrachte die Elemente paarweise und drehe alle Paare die »falsch« stehen um.
In jedem Schritt kommt so das nächstgrößte Element an seinen Platz.
Nach n Schritten ist die Liste sortiert.

Bubblesort – das Beispiel

3 5 6 1 4 2

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2
3	5	1	4	2	6

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2
3	5	1	4	2	6
3	1	5	4	2	6

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2
3	5	1	4	2	6
3	1	5	4	2	6
3	1	4	5	2	6

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2
3	5	1	4	2	6
3	1	5	4	2	6
3	1	4	5	2	6
3	1	4	2	5	6

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2
3	5	1	4	2	6
3	1	5	4	2	6
3	1	4	5	2	6
3	1	4	2	5	6
1	3	4	2	5	6

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2
3	5	1	4	2	6
3	1	5	4	2	6
3	1	4	5	2	6
3	1	4	2	5	6
1	3	4	2	5	6
1	3	2	4	5	6

Bubblesort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2
3	5	1	6	4	2
3	5	1	4	6	2
3	5	1	4	2	6
3	1	5	4	2	6
3	1	4	5	2	6
3	1	4	2	5	6
1	3	4	2	5	6
1	3	2	4	5	6
1	2	3	4	5	6

Große Elemente steigen wie Blasen (“bubbles”) im Wasser nach oben auf.

Bubblesort – der Algorithmus

```
void swap (int *a, int *b){
    int tmp = *a; *a = *b; *b = tmp;
}

void bubblesort(int *array, int length){
    for (int i = 0; i < length-1; i++){
        for (int j = 0; j < length-1-i; j++){
            if (array[j] > array[j+1])
                swap(&(array[j]), &(array[j+1]));
        }
    }
}
```

Bubblesort – die Komplexität

- In jedem Schritt wird die noch zu betrachtende Liste um ein Element kürzer; also sind im i-ten Schritt noch $n - i$ Elemente zu betrachten.
- Summe über alle i Schritte:
  $C_{Bubble} = \sum_{i=0}^{n-1}(n - i - 1) = \sum_{i=1}^{n}(n - i) = \sum_{i=0}^{n-1} i$
- Gaußsche Summe:
  $= \frac{n(n-1)}{2} = \binom{n}{2}$

$$\Rightarrow \Theta(n^2)$$

Bubblesort – Verbesserung

Brich ab wenn in einem inneren Schleifendurchlauf keine Vertauschungen mehr gemacht werden mussten.
Die Liste ist dann bereits sortiert.
Ob im aktuellen Durchlauf Vertauschungen gemacht wurden oder nicht kann man mit einem flag signalisieren.

Bubblesort – Varianten

- flag-Variante
  Abbruch nach einem Durchlauf in dem keine Vertauschung mehr gemacht wurde.
- Shakersort/Cocktailsort
  Die Richtung des Schleifendurchlaufes wird bei jeder Iteration geändert, in der Hoffnung dass die Blasen so schneller ihre Position erreichen.
  Theoretisch ist das kein bisschen besser, praktisch aber dafür komplizierter zu implementieren.
- Ripplesort
  ist genau Bubblesort.

Bubblesort – die Bewertung

- leicht zu implementieren
- sehr schnell bei (vor)sortierten Arrays (flag-Variante)
  $\sim (n)$ wenn die Liste komplett sortiert war
- $\Theta(n^2)$ für andere Eingaben ist nicht schön.

Sortieren durch Einfügen (Insertionsort)

Insertionsort – die Idee

Nimm ein beliebiges Element aus dem unsortierten Teil und füge es an die richtige Stelle im sortierten Teil.
Die richtige Stelle wird duch sukzessives Vergleichen des Einzufügenden Elementes mit den schon sortierten Elementen gefunden. (vgl. Spielkarten in der Hand sortieren)
Wiederhole das bis kein Element mehr im unsortierten Teil vorhanden ist.

Insertionsort – das Beispiel

3	5	6	1	4	2

Insertionsort – das Beispiel

3	5	6	1	4	2
3	5	6	1	4	2

Insertionsort – das Beispiel

3	5	6	1	4	2
|3	5	6	1	4	2
3|	5	6	1	4	2

Insertionsort – das Beispiel

3	5	6	1	4	2
|3	5	6	1	4	2
3|	5	6	1	4	2
3	5|	6	1	4	2

Insertionsort – das Beispiel

3	5	6	1	4	2
|3	5	6	1	4	2
3|	5	6	1	4	2
3	5|	6	1	4	2
3	5	6|	1	4	2

Insertionsort – das Beispiel

3	5	6	1	4	2
|3	5	6	1	4	2
3|	5	6	1	4	2
3	5|	6	1	4	2
3	5	6|	1	4	2
1	3	5	6|	4	2

Insertionsort – das Beispiel

3	5	6	1	4	2
|3	5	6	1	4	2
3|	5	6	1	4	2
3	5|	6	1	4	2
3	5	6|	1	4	2
1	3	5	6|	4	2
1	3	4	5	6|	2

Insertionsort – das Beispiel

3	5	6	1	4	2
|3	5	6	1	4	2
3|	5	6	1	4	2
3	5|	6	1	4	2
3	5	6|	1	4	2
1	3	5	6|	4	2
1	3	4	5	6|	2
1	2	3	4	5	6|
1	2	3	4	5	6

Insertionsort – der Algorithmus

```
void insertionsort(int *array, int length){
    int i, j, key;
    /*ein element alleine ist sortiert -> j=2*/
    for (j = 2; j < length; j++){
        key = array[j];
        i = j-1; /*rueckwaerts*/
        while (i > 0 && array[i] > key)
            array[i+1] = array[i--];/*umkopieren*/
        array[i+1] = key;
    }
}
```

Insertionsort – Variante

Eine Variante von Insertion Sort ist Selection Sort. Hierbei wird aus der unsortierten Teilliste jeweils das Maximum ausgewählt und an die Sortierte angehängt. Diese Variante hat Aufwand $\Theta(n^2)$

Insertionsort – die Komplexität

- Um in eine sortierte Liste aus m Elementen ein Element einzufügen genügen m Vergleiche (im günstigsten Fall 1).
- Um im i-ten Schritt ein Element einzufügen braucht man also maximal i Vergleiche.

Insertionsort – die Komplexität

- Summiert über alle Schritte:

$$C_{Insertion} \leq \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \binom{n}{2}$$

$$\Rightarrow O(n^2)$$

Insertionsort – die Bewertung

- einfache und schnelle Implementierung
- kein zusätzlicher Speicherplatzbedarf
- $O(n^2)$, d.h. mindestens so schnell wie Selectionsort

$\Rightarrow$ leicht und schnell bei kleinen n
$\Rightarrow$ »bessere« Sortieralgorithmen schalten oft beim Erreichen einer gewissen minimalen Listenlänge auf Insertionsort um.

Insertionsort – die Verbesserung?

Insertionsort kann in der Praxis verbessert werden indem man zum Einfügen binäre statt linearer Suche verwendet. (zu binärer und linearer Suche später mehr)
Es bleibt dabei aber in der Klasse $O(n^2)$ hinsichtlich der Anzahl der nötigen Verschiebeoperationen.
Bei den Vergleichsoperationen kommt es zu einer Verbesserung auf $O(n \log n)$.

Teil III

Exkurs: Binärbäume (binary trees)

8 Bäume allgemein
- Was ist ein Baum?
- Teile von Bäumen

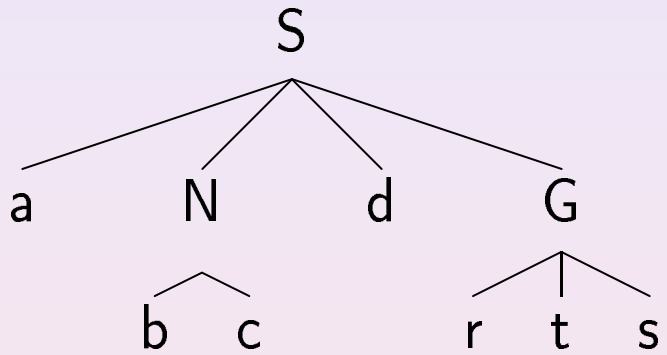
9 Binäre Bäume
- fast vollständige binäre Bäume
- vollständige binäre Bäume
- Parameter
- Beziehungen zwischen Parametern

Bäume allgemein

Wozu dieser Exkurs?

- der Binärbaum ist eine der wichtigsten Datenstrukturen in der Informatik.
- Zur Betrachtung von Algorithmen gehören immer auch die verwendeten Datenstrukturen. Diese geschickt zu wählen ist oft schon die halbe Arbeit.
- die im folgenden eingeführten Begriffe und Strukturen werden uns bei den etwas komplizierteren Algorithmen auch zur Analyse dienen.

Bäume



Bäume sind besondere Graphen:

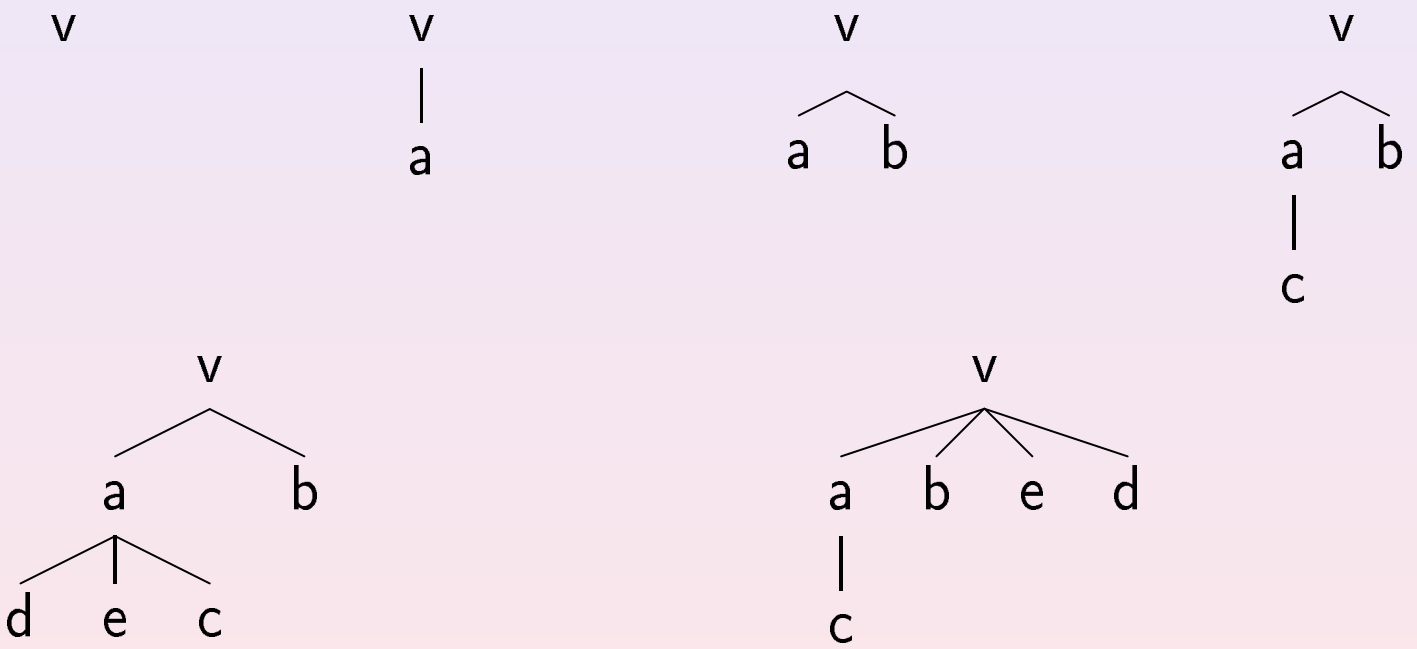
- stark zusammenhängend
- ungerichtet
- keine Zyklen

Bäume

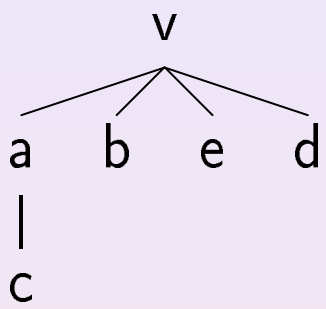
Rekursive Definition:

- Ein einzelner Knoten v ist ein Baum.
- An v können nun beliebig viele Bäume angehängt werden.

Beispiele für Bäume



Bezeichnungen an Bäumen



- Wurzel (v)
- Blatt, äußerer, externer Knoten $\rightarrow$ Knoten ohne Nachfolger (c, b, e, d)
- innerer, interner Knoten $\rightarrow$ Knoten der kein Blatt ist (a, v)
- Elter $\rightarrow$ der direkte Vorgänger eines Knotens (v ist Elter von a, b, e, d)
- Kinder $\rightarrow$ direkte Nachfolger eines Knotens (a, b, e, d sind Kinder von v)

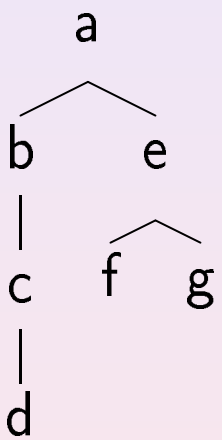
Bezeichnungen an Bäumen

- Nachfolger $\rightarrow$ v ist N. von v' falls
  - v Kind von v'
  - Elter von v ist Nachfolger von v'

  (a, b, e, d, c sind Nachfolger von v)
- Vorgänger (analog)
- Geschwister $\rightarrow$ Knoten die denselben Elter haben (a, b, e, d)
- Teilbaum $\rightarrow$ Jeder Baum der nach vorhergehender rekursiver Konstruktionsregel erschaffen wurde ist Teilbaum des Baumes mit Wurzel v
- Weg zwischen v und $v' \rightarrow$ v ist Vorgänger von v'

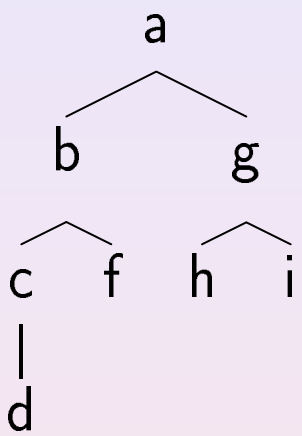
Binäre Bäume

Binäre Bäume



Ein Baum heißt binär wenn jeder Knoten des Baumes maximal zwei Kinder hat.

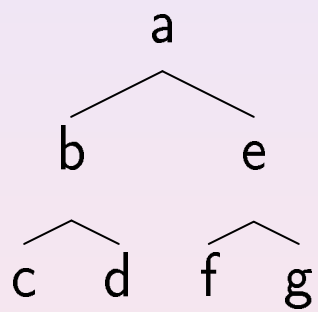
fast vollständige binäre Bäume



Ein binärer Baum heißt fast vollständig wenn:

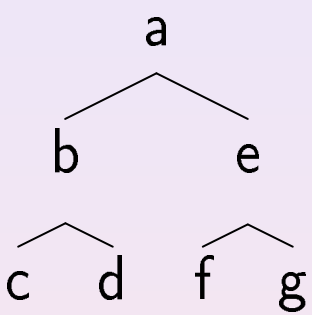
1. alle inneren Knoten bis auf maximal einen haben genau zwei Kinder.
2. Alle Knoten mit weniger als zwei Kindern befinden sich auf den beiden untersten Leveln.
3. Die Blätter im größten Level sind von links nach rechts aufgefüllt.

vollständige binäre Bäume



Ein fast vollständiger binärer Baum heißt vollständiger binärer Baum wenn sein tiefster Level voll besetzt ist.

vollständige binäre Bäume



Rekursive Konstruktionsvorschrift für vollständige binäre Bäume: □ bezeichnet ein Blatt, ◯ einen innere Knoten.

- □ ist ein vollständiger binärer Baum
- sind t_l, t_r vollständige binäre Bäume gleicher Höhe, so ist auch $\{\bigcirc, t_l, t_r\}$ ein vollständiger binärer Baum
- alle anderen Bäume sind keine vollständigen binären Bäume

Parameter

Sei t ein binärer Baum so ist
$i(t) =$ Anzahl der inneren Knoten von t:

$$i(t) = \begin{cases} 0 & \text{falls } t = \square \\ 1 + i(t_l) + i(t_r) & \text{falls } t = (\bigcirc, t_l, t_r) \end{cases}$$

$e(t) =$ Anzahl der äußeren Knoten von t:

$$e(t) = \begin{cases} 1 & \text{falls } t = \square \\ e(t_l) + e(t_r) & \text{falls } t = (\bigcirc, t_l, t_r) \end{cases}$$

$s(t) =$ Größe von t:

$$s(t) = \begin{cases} 1 & \text{falls } t = \square \\ 1 + s(t_l) + s(t_r) & \text{falls } t = (\bigcirc, t_l, t_r) \end{cases}$$

Höhe und Tiefe von Binärbäumen

Die Höhe eines Binärbaumes ist gleich seiner Tiefe.
Oder auch nicht.

V. Strehl

$h(t) =$ Höhe/Tiefe von t:

$$h(t) = \begin{cases} 0 & \text{falls } t = \square \\ 1 + max\{h(t_l), h(t_r)\} & \text{falls } t = (\bigcirc, t_l, t_r) \end{cases}$$

Höhe von Knoten

Die Höhe des Knoten a ist rekursiv definiert

$$h(\square, \square) = 0$$

$$h(a, (\bigcirc, t_l, t_r)) = \begin{cases} 0 & \text{falls } a = \bigcirc \\ h(a, t_l) + 1 & \text{falls } a \in t_l \\ h(a, t_r) + 1 & \text{falls } a \in t_r \end{cases}$$

Beziehungen zwischen Parametern

Hier werden vollständige binäre Bäume betrachtet, die Parameter gelten mit kleinen Korrekturen auch für die anderen Arten von binären Bäumen.

Beziehungen zwischen Parametern

Für binäre Bäume t gelten folgende Aussagen:

- $e(t) = i(t) + 1 \Rightarrow s(t) = 2i(t) + 1 = 2e(t) - 1$
- $h(t) \leq i(t)$
- $e(t) \leq 2^{h(t)} \Rightarrow \log e(t) \leq h(t)$

(Genauer und mit Beweisen nachzulesen im Skript zu Theoretischer Informatik 3 von V. Strehl)

Teil IV

Sorting II

Haldensortierung (Heapsort)

Heapsort – der Heap

Ein Heap ist ein fast vollständiger Binärbaum

- in dem jedem Knoten ein Schlüssel zugeordnet ist und
- bei dem jeder Knoten die Heap-Eigenschaft hat.

Die Heap-Eigenschaft: Der in einem Knoten abgespeicherte Schlüssel ist nicht kleiner als die in seinen Kindern abgespeicherten Schlüssel.

Heapsort – die Idee

In einem Heap steht das größte Element stets an der Spitze.
Entferne dieses Element und stelle danach den Heap wieder her.
Sukzessives Wiederholen liefert die Elemente des Heaps in sortierter Folge.

Heapsort – die Datenstruktur

Zwei Heaps mit den Elementen $[1 \ldots 6]$:

Heaps sind also im Allgemeinen nicht eindeutig.

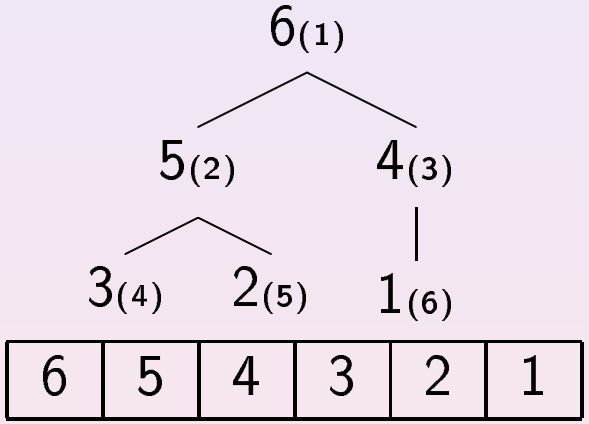
Heapsort – die Datenstruktur

auf einem Heap sind die folgenden Operationen definiert:

- `createheap(array)` $\longrightarrow$ `heap`
  erstellt einen heap aus einer Reihe von Schlüsseln.
- `delete_max(heap)` $\longrightarrow$ `key`
  liefert das erste Element aus dem Heap zurück und löscht dieses.
- `reheap(corrupted heap)` $\longrightarrow$ `heap`
  stellt in einem zerstörten Heap die Heap-Eigenschaft wieder her.

Heapsort – die Datenstruktur

Wir realisieren einen Heap als Feld indem wir ihn durchnummerieren:



Jetzt gilt für jeden Knoten mit Nummer i:

- sein linkes Kind hat Nummer $2i$
- sein rechtes Kind hat Nummer $2i + 1$
- sein Elter hat Nummer $\lfloor \frac{i}{2} \rfloor$

Heapsort – die Datenstruktur

Felder sind ab null indiziert $\longrightarrow$ Indexverschiebung
Einen Heap mit n Elementen stellen wir in einem Feld mit Indexmenge$[0 \ldots n-1]$ dar, so daß für einen Knoten mit Nummer i gilt:

- sein linkes Kind hat Nummer $2i+1$
- sein rechtes Kind hat Nummer $2i+2$
- sein Elter hat Nummer $\lfloor \frac{i-1}{2} \rfloor$

Heapsort – zur Implementierung

Splitte das Feld: vorne liegt der restliche Heap, hinten die schon herausgenommenen, sortierten Elemente.
$\Rightarrow$ ein Position links von der Bruchstelle liegt ein Blatt.

`reheap`: Ersetze die Wurzel durch eben dieses Blatt. Vertausche es dann so lange nach unten bis der Heap repariert ist.

konkret: Sei v die Wurzel des Heaps; Solange v die Heap-Eigenschaft nicht hat vertausche die Schlüssel in v und seinem größeren Kind (v') und setze $v = v'$.

Heapsort – der Algorithmus

```
void reheap(int *array, int len, int r){
    int i = r;
    int j = 2*r+1; /*j ist kind von i*/
    while (j < len){/*suche groesseres kind*/
        if ((j+1 < len) && (array[j+1] > array[j]))
            j++;
        /*heap-eigenschaft ist verletzt*/
        if (array[j] > array[i]){
            swap(&(array[i]), &(array[j]));
            i = j;
            j = 2*j+1;
        } else break;
    }
}
```

Heapsort – der Algorithmus

```
void heapsort(int *array, int len){
    for (int i = len-1; i >= 0; i--) /*create heap*/
        reheap(array, len, i);
    for (int l = len-1;l >= 1; l--){/*delete_max*/
        swap(&(array[0]), &(array[l]));
        reheap(array, l, 0);
    }
}
```

Heapsort – die Komplexität

Kosten von `reheap` auf einem Teilbaum mit Wurzel i und Knoten mit Nummern kleiner n:

`reheap` beginnt auf dem Level $\log(i+1)$ und steigt maximal bis zum Level $\lfloor \log n \rfloor$ hinab. (Hier sind die Blätter)

Heapsort – die Komplexität

Dabei sind bei jedem Levelwechsel zwei Vergleiche nötig:

- Kind mit dem größeren Schlüssel bestimmen
- Überprüfen der Heap-Eigenschaft

Heapsort – die Komplexität

Man erhält also für die maximale Anzahl an Vergleichen beim Aufruf von reheap am Knoten i in einem Heap mit n Elementen:

$$C_{reheap}(n, i) = 2(\lfloor \log n \rfloor - \lfloor \log(i+1) \rfloor)$$

Heapsort – die Komplexität

Anzahl der Vergleiche für createheap auf einem Feld mit n Elementen:

$$C_{create}(n) \leq \sum_{i=0}^{n-1} C_{reheap}(n,i) \leq \cdots \leq 5n$$

Berücksichtigt man zusätzlich das reheap nur für $i < \lfloor \frac{n}{2} \rfloor$ etwas tut kommt man sogar auf $\frac{7}{2}n$

Heapsort – die Komplexität

Es bleiben noch die Kosten für den eigentlichen Sortiervorgang:

$$
\begin{aligned}
C_{heap}(n) &\leq \sum_{l=1}^{n-1} C_{reheap}(l, 0) \\
&\leq \sum_{l=1}^{n-1} (\lfloor \log(l+1) \rfloor - \lfloor \log(1) \rfloor) \\
&\leq 2 \sum_{l=2}^{n} \lfloor \log(l) \rfloor \\
&\leq 2 \sum_{l=2}^{n} \log(n) \\
&\leq 2n \log(n)
\end{aligned}
$$

Heapsort – die Komplexität

Insgesamt ergibt sich also als Komplexität für Heapsort:
$C_{create} + C_{heap} \sim 2n \log n + \frac{7}{2} n$

$$\Rightarrow O(n \log n)$$

Heapsort – die Bewertung

- etwas aufwändiger zu implementieren
- echtes in-situ Verfahren, sehr guter Platzaufwand
- $O(n \log n)$ mit nur kleinen linearen Faktoren ist ziemlich gut
- $\Rightarrow$ einfachstes »schnelles« Sortierverfahren, das keinen zusätzlichen Speicherplatzverbrauch hat

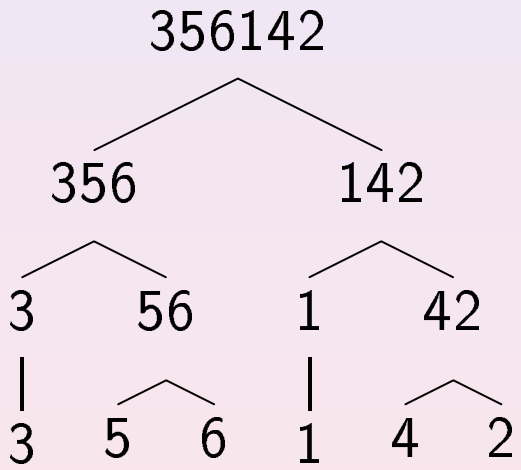
Sortieren durch Mischen (Mergesort)

Mergesort – die Idee

Eine Liste mit nur einem Element ist trivialerweise sortiert.
Zwei sortierte Listen zu einer neuen sortierten Liste zusammenzufügen ist ebenfalls einfach. Man schaut sich jeweils die beiden ersten Elemente der Listen an und nimmt das kleinere in die neue Liste auf.
Man sortiert jetzt indem man erst die Liste in Listen mit nur einem Element zerlegt und diese dann nach obigem Muster wieder zusammenbaut.

Mergesort – Splitting

Zerlegung in Probleme der Größe (1):



Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2

Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2
3	5	6	1	4	2

Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	2	4

Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	2	4
3	5	6	1	2	4

Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	2	4
3	5	6	1	2	4
3	5	6	1	2	4

Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	2	4
3	5	6	1	2	4
3	5	6	1	2	4
3	5	6	1	2	4

Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	2	4
3	5	6	1	2	4
3	5	6	1	2	4
3	5	6	1	2	4
1	2	3	4	5	6

Mergesort – das Beispiel

Mischen:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	2	4
3	5	6	1	2	4
3	5	6	1	2	4
3	5	6	1	2	4
1	2	3	4	5	6
1	2	3	4	5	6

Mergesort – der Algorithmus

```
void merge(int *a, int *b, int l, int m, int r){
    for (int i=l, int j=m, int k=l; k < r; k++){
        if ((j >= r) || (i < m) && (a[i] >= b[j]))
            b[k] = a[i++];
        else
            b[k] = a[j++];
    }
}

void copy(int *a, int *b, int l, int r){
    while(l < r) a[l] = b[l++];
}
```

Mergesort – der Algorithmus

```
void mergesort(int *array, int *help, int l, int r){
    if (l < r){
        int m = (l + r)/2;
        mergesort(array, help, l, m - 1);
        mergesort(array, help, m, r);
        merge(array, help, l, m, r);
        copy(help, array, l, r);
    }
}
```

Mergesort – die Komplexität

Um eine Folge mit n und eine Folge mit m Elementen zu mischen werden maximal $n + m - 1$ Vergleiche benötigt.
In jedem Schritt werden die beiden kleinsten Elemente miteinander verglichen.
Die Anzahl der Restelemente nimmt dabei in jedem Schritt um eins ab.
Ist nur noch ein Element übrig entfällt der letzte Vergleich.

Mergesort – die Komplexität

Man erhält also folgende Rekursionsgleichung:

$$
\begin{aligned}
C_{merge}(1) &= 0 \\
C_{merge}(n) &= C_{merge}(\lceil \frac{n}{2} \rceil) + C_{merge}(\lfloor \frac{n}{2} \rfloor) + n - 1 \quad , n >= 2 \\
&= \ldots \\
&= (n-1)\lceil \log(n) \rceil \\
&\Rightarrow \Theta(n \log(n))
\end{aligned}
$$

Mergesort – die Bewertung

- einfacher Ansatz, aber nicht ganz leicht zu implementieren
- verbraucht n mehr Speicherplatz als z.B. heapsort
- sehr geringe Anzahl an Vergleichen
- $\Theta(n \log(n))$ ist eine gute Komplexität

Shellsort

Shellsort – die Idee

Eine vollkommen unsortierte Matrix ist schlechter als eine spaltenweise sortierte Matrix.
Wenn wir eine Matrix mit n Spalten spaltenweise sortiert haben und in eine Matrix mit $n - k$ Spalten umformen so ist die neue Matrix bereits vorsortiert und jetzt leichter wieder spaltenweise zu sortieren.
Setzt man das fort bis zu einer Matrix mit nur noch einer Spalte so ist diese Spalte sortiert.

Shellsort – das Beispiel

$3, 7, 9, 0, 5, 1, 6, 8, 4, 2, 0, 6, 1, 5, 7, 3, 4, 9, 8, 2$ in einer Matrix mit 7 Spalten:

$$\begin{array}{ccccccc} 3 & 7 & 9 & 0 & 5 & 1 & 6 \\ 8 & 4 & 2 & 0 & 6 & 1 & 5 \\ 7 & 3 & 4 & 9 & 8 & 2 & \end{array} \quad \rightarrow \quad \begin{array}{ccccccc} 3 & 3 & 2 & 0 & 5 & 1 & 5 \\ 7 & 4 & 4 & 0 & 6 & 1 & 6 \\ 8 & 7 & 9 & 9 & 8 & 2 & \end{array}$$

Shellsort – das Beispiel

Umformen in eine Matrix mit 3 Spalten:

$$\begin{array}{ccc} 3 & 3 & 2 \\ 0 & 5 & 1 \\ 5 & 7 & 4 \\ 4 & 0 & 6 \\ 1 & 6 & 8 \\ 7 & 9 & 9 \\ 8 & 2 & \end{array} \quad \longrightarrow \quad \begin{array}{ccc} 0 & 0 & 1 \\ 1 & 2 & 2 \\ 3 & 3 & 4 \\ 4 & 5 & 6 \\ 5 & 6 & 8 \\ 7 & 7 & 9 \\ 8 & 9 & \end{array}$$

Shellsort – das Beispiel

Umformen in eine Matrix mit 1 Spalte:

0	0	1	1	2	2	3	3	4	4	5	6	5	6	8	7	7	9	8	9	T
									↕										
0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	T

Shellsort – der Algorithmus

- die Daten sind nicht in einem zweidimensionalen Feld, sondern in einem Eindimensionalen, das entsprechend indiziert wird.
- in den einzelnen Spalten wird Insertionsort benutzt, da es schnell auf vorsortierten Eingaben arbeitet.

Shellsort – der Algorithmus

```
void shellsort (int *array, int len){
    int i, j, k, h, v;
    int cols[] = {1391376,463792,198768,33936,13776,
                  4592,1968,861,336,112,48,21,7,3,1};
    for (k = 0; k < 15; k++){/*spalten*/
        h = cols[k];/*aktuelle spaltenzahl*/
        for (i = h;  i < len; i++){/*spaltenelemente*/
            v = a[i];
            j = i;
            while (j >= h && a[j-h] > v){/*insertion sort*/
                a[j] = a[j-h];
                j = j-h;
            }
            a[j] = v;
        }
    }
}
```

Shellsort – die Komplexität

Die Analyse ist kompliziert. Sie hängt vor allem von der Folge h ab
wählt man zum Beispiel $h(k) = 2^{k-1}$ so erhält man $O(n\sqrt{n})$
Mit der Folge von Pratt $h(p, q) = 2^p 3^q$ erhält man $O(n \log(n)^2)$
das ist asymptotisch besser, die Folge hat aber sehr viele Elemente, was sich nachteilig bei vorsortierten Eingaben auswirkt.

Shellsort – die Komplexität

Es gibt noch andere gute Folgen, aber mit keiner erreicht man bisher das gewünschte $O(n \log n)$ im worst case.
für den average case ist es ebenfalls nicht klar ob man $O(n \log n)$ erreichen kann. Bisher gilt also für Shellsort:

$$\Rightarrow O(n \log(n)^2)$$

Shellsort – die Bewertung

- noch nicht vollständig untersuchter Algorithmus
- in der Praxis ziemlich schnell
- gut einstellbar auf bestimmte Regelmäßigkeiten im Input mittels der Folge h
- kaum zusätzlicher Speicherplatzbedarf
- mit $O(n \log(n)^2)$ nicht langsam, aber bei ungünstigen Eingaben eventuell Einbrüche.

Quicksort

Quicksort – der beliebteste Sortieralgorithmus

- klassisches, sehr beliebtes – weil im Mittel sehr effizientes – Sortierverfahren
- trotz seiner schlechten Effizienz im worst case $\longrightarrow O(n^2)$
- bei *UNIX* das Standardverfahren
- basiert auf dynamischem divide-and-conquer
- wenig Speicherplatzbedarf $\longrightarrow n + O(\log n)$

Quicksort – die Idee

Ein Splitter in einer Liste $L[1 \ldots n]$ ist ein Index $j (0 \leq j \leq n-1)$ mit $\forall i < j : L[i] < L[j]$ und $\forall k > j : L[j] < L[k]$.
Ein Splitter j steht bereits an seiner richtigen Position, es genügt also $L[0 \ldots j-1]$ und $L[j+1 \ldots n-1]$ zu sortieren.
Da die Wahrscheinlichkeit für einen zufälligen Splitter sehr gering ist muss man sich die Splitter durch Umordnung ("partition") selbst beschaffen.

Quicksort – die Idee

- wähle eine Element der Liste als Pivot p (z.B. $L[n-1]$)
- durchlaufe die Liste einmal komplett und erzeuge durch Vergleiche mit p die beiden Teillisten
  - $L'[0 \ldots j-1]$ – enthält Elemente aus $L[0 \ldots n-1]$ die $< p$ sind
  - $L'[j+1 \ldots n-1]$ – enthält Elemente aus $L[0 \ldots n-1]$ die $> p$ sind
- setze jetzt $L'[j] = p$ so ist p ein Splitter für $L'[0 \ldots n-1]$
- setze dies rekursiv auf den Teillisten fort bis diese sortiert sind und setze sie wieder zusammen.

Quicksort – das Beispiel

Bezeichnungen im Beispiel:

- Das Pivotelement
- Zwei **Zeiger** die die Liste von links nach rechts bzw. umgekehrt durchlaufen
- Ein Element das seinen Platz gleich tauschen wird

Quicksort – das Beispiel

3 5 6 1 4 2

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	6	3	4	5

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	|6	3	4	5
1	2	|**6**	3	**4**	5

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	|6	3	4	5
1	2	|**6**	3	**4**	5
1	2	|4	**3**	**6**	5

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	|6	3	4	5
1	2	|**6**	3	**4**	5
1	2	|4	**3**	**6**	5
1	2	|4	3	5	|6

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	|6	3	4	5
1	2	|**6**	3	**4**	5
1	2	|4	**3**	**6**	5
1	2	|4	3	5	|6
1	2	|**4**	**3**	5	|6

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	|6	3	4	5
1	2	|**6**	3	**4**	5
1	2	|4	**3**	**6**	5
1	2	|4	3	5	|6
1	2	|**4**	**3**	5	|6
1	2	|3	4	**5**	|6

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	|6	3	4	5
1	2	|**6**	3	**4**	5
1	2	|4	**3**	**6**	5
1	2	|4	3	5	|6
1	2	|**4**	**3**	5	|6
1	2	|3	4	**5**	|6
1	2	|3	4	5	|6

Quicksort – das Beispiel

3	5	6	1	4	2
3	5	6	1	**4**	2
3	5	6	**1**	4	2
1	**5**	6	3	4	2
1	2	|6	3	4	5
1	2	|**6**	3	**4**	5
1	2	|4	**3**	**6**	5
1	2	|4	3	5	|6
1	2	|**4**	**3**	5	|6
1	2	|3	4	**5**	|6
1	2	|3	4	5	|6
1	2	3	4	5	6

Quicksort – der Algorithmus

```
int partition(int *array, int l, int r, int p){
    int i = l - 1, j = r; /*zeiger*/
    swap(&(array[p]), &(array[r]));
    p = r; /*pivot nach rechtsaussen*/
    while (i < j){
        do i++; while((i<j) && (array[i]<array[p]));
        do j--; while((j>i) && (array[j]>array[p]));
        if (i >= j)/*swap pivot*/
            swap(&(array[i]), &(array[p]));
        else /*tausche normal*/
            swap(&(array[i]), &(array[j]));
    }
    return i;/*naechstes pivot*/
}
```

Quicksort – der Algorithmus

```
void quicksort(int *array, int l, int r){
    if (l < r ){/*terminiere rekursion*/
        int p = partition(array, l, r, r);
        if (p - l < r - p){
            quicksort(array, l, p-1);
            quicksort(array, p+1, r);
        } else {
            quicksort(array, p+1, r);
            quicksort(array, l, p-1);
        }
    }
}
```

Aus Speicherplatzgründen wird immer die kleinere Folge zuerst sortiert. Beachtet man das nicht, kann der zusätzliche Speicherplatzbedarf bis auf $\Theta(n)$ anwachsen.

Quicksort – Varianten

It is tempting to try to develop ways to improve Quicksort: a faster sorting algorithm is computer science's better "mousetrap".
Many ideas have been tried and analyzed, but it is easy to be deceived, because the algorithm is so well balanced that the effects of improvements in one part of the program can be more than offset by the effects of bad performance in another part.

R. Sedgewick

Quicksort – die Komplexität

Ein Element in einer Folge hat den Rang k wenn es in der sortierten Folge an $k$ter Stelle steht.

- gearbeitet wird eigentlich nur in `partition`:

  $L[0 \ldots n-1] \rightarrow \langle L'[1 \ldots j-1], L'[j+1 \ldots] \rangle$
  falls $j =$ Rang von $L[n]$ in L ist.

- Daher kann man ansetzen:

  $C_{quicksort}(L) =$
  $n - 1 + C_{quicksort}(L'[1 \ldots j-1]) + C_{quicksort}(L'[j+1 \ldots n]$
  falls $j =$ Rang von $L[n]$ in L.

Quicksort – die Komplexität

worst case: Pivot ist Minimum oder Maximum:

- $\Rightarrow$ eine der Listen L' ist leer, d.h. $j = 1$ oder $j = n$

$$C^{max}_{quicksort}(1) = 0$$
$$C^{max}_{quicksort}(n) = n - 1 + C^{max}_{quicksort}(n-1) \quad (n > 1)$$

- also (Rekursionsgleichung lösen)

$$C^{max}_{quicksort}(n) = \frac{n(n-1)}{2}$$

$$\Rightarrow O(n^2)$$

Quicksort – die Komplexität

- best case: Pivot ist mittleres Element:
  $\Rightarrow$ die Listen L' sind ziemlich genau gleich lang, d.h.
  $j = \lceil \frac{n+1}{2} \rceil$ oder $j = \lfloor \frac{n+1}{2} \rfloor$

  $$C^{min}_{quicksort}(1) = 0$$
  $$C^{min}_{quicksort}(n) =$$
  $$n - 1 + C^{min}_{quicksort}(\lceil \tfrac{n+1}{2} \rceil) + C^{min}_{quicksort}(\lfloor \tfrac{n+1}{2} \rfloor) \quad (n > 1)$$

- also (Rekursionsgleichung aufstellen und lösen)

  $$C^{min}_{quicksort} = n \log n + O(n)$$

Quicksort – die Komplexität

average case:

- Annahme: jedes der Elemente hat die gleiche Wahrscheinlichkeit als Pivot zu dienen ($\frac{1}{n}$)
- Diese Gleichverteilung vererbt sich auf die Teilprobleme (Chu-Vandermonde-Identität)

Quicksort – die Komplexität

- Man erhält folgende Rekursion:

$$C^{avg}_{quicksort}(1) = 0$$
$$C^{avg}_{quicksort}(n) = \sum_{k=1}^{n} \frac{1}{n}[(n-1) + C^{avg}_{quicksort}(k-1) + C^{avg}_{quicksort}(n-k)] \quad (n < 2)$$

- vereinfachen zu:

$$C^{avg}_{quicksort}(n) = n - 1 + \frac{2}{n} \sum_{k=1}^{n-1} + C^{avg}_{quicksort}(k)$$

Quicksort – die Komplexität

- weitere Umformungen:

$$\frac{C^{avg}_{quicksort}(n)}{n+1} = \sum_{j=3}^{n+1} j/4 - \sum_{j=2}^{n} 2/j$$

- hier versteckt sich die binäre Entropiefunktion:

$$H(p) = -\sum_{i=1}^{n} p_i \log p_i \qquad p(p_1 \dots p_n) \quad 0 \leq p_i \leq 1$$

- und man erhält:

$$2H_n + \frac{4}{n+1} - 4 = 2\ln n + O(1)$$

$$\Rightarrow \sim 1.386 n \log n$$

Quicksort – Varianten

Die Wahl des Pivotelementes ist entscheidend, dementsprechend gibt es zahlreiche Varianten.

- Wähle p als $\frac{\mathit{arraylänge}}{2}$
- Random-Quicksort: zufällige Pivotwahl verhindert schlechte Instanzen, $O(n^2)$ im worst case bleibt aber (versteckt) erhalten
- Median-of-three-Quicksort: Mittleres von 3 Elementen an festen Positionen als Pivot wählen
- Randomized-median-of-three-Quicksort

Quicksort – die Bewertung

- trickreiche Implementierung
- kein in-situ Verfahren, da für jeden noch zu sortierenden Teilbereich die Indexgrenzen gespeichert werden müssen
- schlechter worst case (paradoxerweise sortierte Liste bei dieser Implementierung)
- aber sehr schnell im best case und
- average case $O(n \log n)$ mit Vorfaktor ~ 1.3

Quicksort – die Bewertung

A carefully tuned version of Quicksort is likely to run significantly faster on most computers than any other sorting method.

R. Sedgewick

Eine allgemeine untere Schranke für das Sortieren

allgemeine untere Schranke

Geht es überhaupt noch schneller?

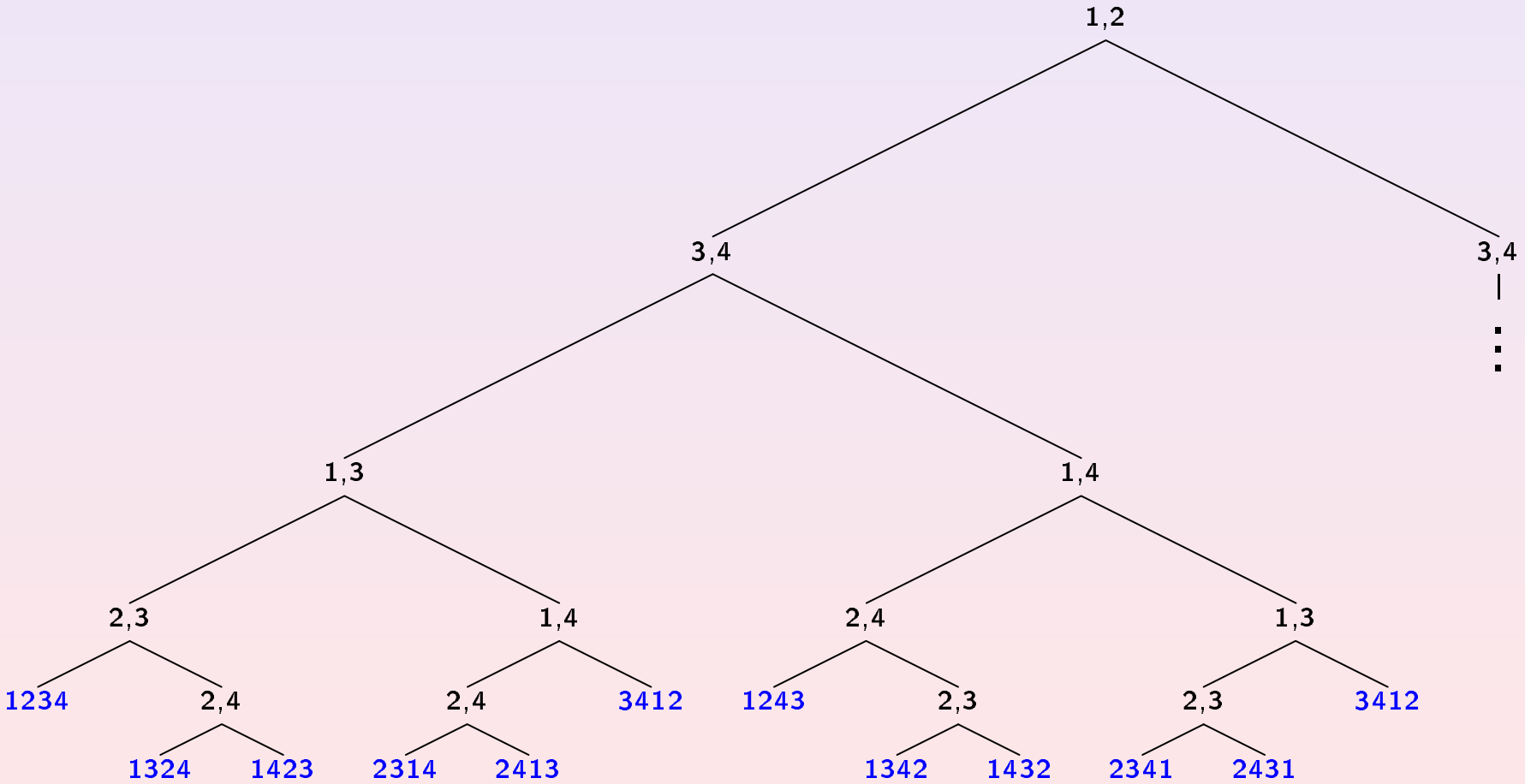
binäre Entscheidungsbäume

- Betrachte beliebigen vergleichsorientierten Algorithmus A auf einer Liste mit Länge n
- stelle den Algorithmus als Baum dar
- innere Knoten werden mit den ausgeführten Vergleichen beschriftet
- gehe links wenn der Vergleich im Knoten i, j ergeben hat dass $i < j$, sonst rechts (Schlüssel sind eindeutig)
- wir erhalten einen binären Baum

binäre Entscheidungsbäume

- Ein Weg im Baum von der Wurzel bis zu einem Blatt stellt einen möglichen Ablauf des Algorithmus dar.
- notiere die zum Ablauf gehörige Permutation an den Blättern
- einen solchen Baum nennt man binären Entscheidungsbaum $T(A)$.

Entscheidungsbaum für Mergesort mit $n = 4$



Wir Wissen...

- $n!$ Permutationen auf einer Liste mit n Elementen $\longrightarrow$ binärer Entscheidungsbaum hat mindestens $n!$ Blätter.
- $\log e(t) \leq h(t) \longrightarrow$ Jeder binäre Baum mit n Blättern hat mindestens Höhe $\lceil \log n \rceil$.

der Trick

- man erhält also sofort folgende Abschätzung:

  $$C_A(n) \geq h(T(A(n))) \geq \lceil \log(n!) \rceil$$

- die man mit der Stirlingformel bearbeiten kann:

  $$n! = \frac{n^n}{e^n}\sqrt{2\pi n}\left(1 + O(\tfrac{1}{n})\right)$$

- um zu erkennen, dass für die maximale Anzahl an Vergleichen gilt:

untere Schranke

Jeder vergleichsbasierte Sortieralgorithmus benötigt auf Folgen der Länge n mindestens $n \log(n) - 1,44n$ Vergleiche.

Genauere Analyse zeigt, dass das auch für die mittlere Anzahl an Vergleichen gilt.

die Folgen

- Heapsort, Shellsort und Mergesort sind asymptotisch optimal
- Quicksort nur im average case
- die naiven Sortierverfahren sind asymptotisch nicht optimal

Teil V

Searching

Vorbemerkung

Suchen ist sehr eng mit Sortieren verwandt.
In sortierten Strukturen lässt sich besonders gut Suchen.
Suchen ist Teilproblem des Sortierens.
Sortiere eine Liste so, dass das gesuchte Element vorne steht (eine geeignete Ordnungsfunktion kann dies leisten).
Das Element ist dann beim Sortieren gleichzeitig gesucht worden.

Sequentielle Suche

sequentielle Suche – die Idee

Fange am Anfang an und vergleiche jedes Element mit dem Schlüssel.
Wenn das aktuelle Element gleich dem Schlüssel ist, ist es gefunden; terminiere dann.
Wenn das Ende der Liste erreicht ist, ist das Element nicht in der Liste.

sequentielle Suche – das Beispiel

Suche Wert 6

3	5	6	1	4	2

sequentielle Suche – das Beispiel

Suche Wert 6

3	5	6	1	4	2
3	5	6	1	4	2

sequentielle Suche – das Beispiel

Suche Wert 6

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

sequentielle Suche – das Beispiel

Suche Wert 6

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

sequentielle Suche – das Beispiel

Suche Wert 6

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

$\Rightarrow$ Erfolg

Suche den Wert 7 führt dagegen zum Mißerfolg.

sequentielle Suche – der Algorithmus

```
int sequentialsearch(int *array, int len, int key){
    for (int i = 0; i < len; i++){
        if (array[i] ==  key)
            return 1;
    }
    return 0;
}
```

sequentielle Suche – die Komplexität

trivialerweise:

$$\Rightarrow \Theta(n)$$

sequentielle Suche – Varianten

An der Komplexität $\Theta(n)$ kann nicht gerüttelt werden, aber einige Varianten machen die Implementierung schneller:

- Quicksequentialsearch

`array[len]` bekommt den Schlüssel zugewiesen. Im Erfolgsfall wird ein Mißerfolg gemeldet falls `i == len`
Das spart den Vergleich in der Abbruchbedingung der for-Schleife ein.

- sequentielle Suche auf sortierter Liste

Das Feld wird erst sortiert bevor gesucht wird. Man kann dann die Abbruchbedingung der for-Schleife auf `i < len && array[i] < key` ändern. So muss im Mißerfolgsfall nicht die ganze Liste durchsucht werden. Lohnt sich erst ab gewisser Feldgröße.

sequentielle Suche – die Bewertung

- sehr leichte Implementierung
- nicht gerade geistreich
- für kleine n trotzdem keine schlechte Wahl

Minimum- und Maximumsuche

Minimum- und Maximumsuche – die Idee

Gehe die Liste sequentiell durch und merke jeweils das größte/kleinste bisher gelesene Element.
Am Ende der Liste gilt: das gemerkte Element ist Maximum/Minimum.

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – das Beispiel

Suche Maximum:

3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2
3	5	6	1	4	2

Minimum- und Maximumsuche – der Algorithmus

```
int maxfind(int *array, int len){
    int max = array[0];
    for (int i = 1; i < len; i++){
        if (array[i] > max)/* < fuer minfind*/
            max = array[i];
    }
    return max;
}
```

Minimum- und Maximumsuche – die Komplexität

- Anzahl der Vergleichsoperationen:

$$\sim n - 1$$

- Anzahl der Wertzuweisungen:
  abhängig von der Anordnung (relative Größe) der Listenelemente

  $C^{max}_{maxfind} = n - 1$ (aufsteigend sortierte Liste)
  $C^{min}_{maxfind} = 1$ (absteigend sortierte Liste)

- max/minfind haben auf sortierten Listen $O(1)$

Minimum- und Maximumsuche – die Komplexität

Recht kompliziert ist die Analyse für den average case; der Weg führt über die Permutationen der Eingabelisten und die Anzahl der in ihnen vorkommenden links-rechts-Maxima (also neuer lokaler Maxima).

Minimum- und Maximumsuche – Komplexität

- Zwischen n und $n-1$ Zuweisungen sind nötig.
- Genau $n-1$ Vergleiche sind nötig.

Quickselect

Quickselect – die Idee

Um das Element mit Rang k in einer Liste zu finden kann man vorgehen wie bei Quicksort.
Im Rekursionsschritt muss jedoch nur der Teil weiter betrachtet werden der das gesuchte Element enthält.

Quickselect – der Algorithmus

```
int quickselect(int *array, int len, int k){
    /*gibt index des elements mit rang k zurueck*/
    int pivot = len-1;
    pivot = partition(array, 0, len-1, pivot);
    int rank = pivot+1;/*rang des pivot*/
    if (k == rank)
        return pivot + 1;
    if (k < rank)
        return quickselect(array, rank-1, k);
    return pivot + 1 +
       quickselect(&(array[pivot-1]),
                   len-rank, k-rank);
}
```

Quickselect – die Komplexität

best case:
trivial:

$$\Rightarrow O(1)$$

worst case:
Wenn wir als Pivot immer nur das Maximum oder Minimum erwischen verkommt Quickselect zu einem worst-case-Quicksort. Es geht in jedem Schritt nur ein Element verloren; das Teilfeld das nicht mehr betrachtet werden muss hat minimale Länge.

$$\Rightarrow O(n^2)$$

Quickselect – die Komplexität

average case:

- Wir recyclen den Quicksort Ansatz:

$$C_{qs}^{avg}(n,k) = 0 \quad \text{falls } n \in [0 \ldots 1]$$
$$C_{qs}^{avg}(n,k) = (n-1) + \frac{1}{n}\left[\sum_{r=1}^{k-1} C_{qs}^{avg}(n-r,k-r) + \sum_{r=k+1}^{n} C_{qs}^{avg}(r-1,k)\right] \quad \text{falls } n \geq 2$$

- Abschätzung nach oben, unabhängig von k:

$$C_{qs}^{avg}(n) = 0 \quad \text{falls } n \in [0 \ldots 1]$$
$$C_{qs}^{avg}(n) \leq (n-1) + \frac{1}{n}\sum_{i=1}^{n} \max\{C_{qs}^{avg}(i-1), C_{qs}^{avg}(n-i)\} \quad \text{falls } n \geq 2$$

Quickselect – die Komplexität

Hier schließt sich ein länglicher Beweis durch Induktion an der zeigt, dass

$$C_{qs}^{avg}(n) \leq 4n$$

gilt.

$$\Rightarrow O(n)$$

Quickselect – die Bewertung

- Abfallprodukt von Quicksort
- bei günstigen Eingaben geeignet um Element mit Rang k schnell zu finden
- bei mehreren Anfragen nach einem Element bestimmten Ranges ist sortieren aber besser
- ungünstiger worst case
- im average case aber besser als andere Selektionsalgorithmen

binäre Suche (binary search)

binary search – die Idee

In einer sortierten Liste kann man Suchen, indem man zuerst ihr mittleres Element betrachtet und sich dann, davon ausgehend, der Rechten oder der linken Teilliste zuwendet.
Hier wiederholt man den Vorgang solange, bis die zu betrachtende Teilliste nur noch ein Element enthält.
Ist dieses Element das Gesuchte ist es gefunden, wenn nicht war es nicht in der Liste. (vgl. Telefonbuch)

binary search – das Beispiel

Suche 5:

1 2 3 4 5 6

binary search – das Beispiel

Suche 5:

1	2	3	4	5	6
1	2	3	4	5	6

binary search – das Beispiel

Suche 5:

1	2	3	4	5	6
1	2	3	4	5	6
			4	5	6

binary search – das Beispiel

Suche 5:

1	2	3	4	5	6
1	2	3	4	5	6
			4	5	6
			4	5	6

binary search – das Beispiel

Suche 5:

1	2	3	4	5	6	
1	2	3	4	5	6	
			4	5	6	
			4	5	6	
				5		

binary search – das Beispiel

Suche 5:

1	2	3	4	5	6	
1	2	3	4	5	6	
			4	5	6	
			4	5	6	
				5		
				5		

binary search – der Algorithmus

```
int binarysearch(int *array, int len, int key){
    int l = 0, r = len-1;
    for (int pos = (l+r)/2; l < r; pos = (l+r)/2){
        if (key >= array[pos])
            l = pos+1; /*rechtes teilarray*/
        else
            r = pos;/*linkes teilarray*/
    }
    return (l==key);
}
```

binary search – die Komplexität

Sei $n \in [2^{k-1} \dots 2^k - 1]$:

- nach dem Teilen der Liste müssen wir maximal noch eine Liste mit $\lceil \frac{(2^k-1)-1}{2} \rceil$ Elementen durchsuchen.
- Ist die Liste nur noch einelementig brauchen wir nur noch einen weiteren Vergleich.
- Da zu Beginn gilt:$k = \lceil \log(n+1) \rceil$ benötigen wir insgesamt maximal $\lceil \log(n+1) \rceil$ Vergleiche.

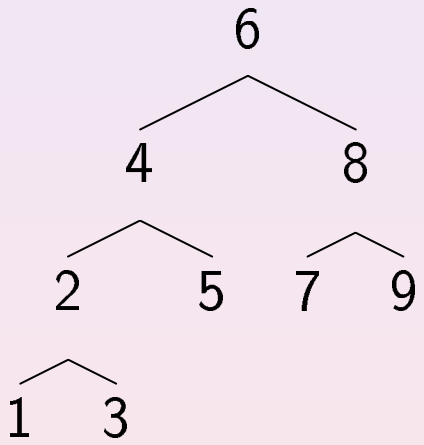
$$\Rightarrow O(\log(n))$$

binary search – binäre Suchbäume

Ein binärer Suchbaum ist ein binärer Baum in dem jeder Knoten die Suchbaumeigenschaft erfüllt:
Der in einem Knoten x abgespeicherte Schlüssel ist größer bzw. kleiner als jeder Schlüssel, der in einem Knoten des linken bzw. rechten Teilbaumes von x abgespeichert ist.

binary search – binäre Suchbäume

ein binärer Suchbaum aus der Liste $[1 \ldots 9]$



binary search – binäre Suchbäume

Suchen:
Vergleiche den gesuchten Schlüssel mit dem Schlüssel im Knoten.
Ist der Schlüssel gleich, terminiere erfolgreich.
Ist er größer, suche im rechten Teilbaum weiter, ist er kleiner im Linken.
Gibt es keinen Teilbaum mehr, terminiere erfolglos.

binary search – binäre Suchbäume

Einfügen:
Suche nach dem Schlüssel im Baum. Ist die Suche erfolgreich, terminiere.
Wenn nicht ist man jetzt in einem Blatt. Mache aus dem Blatt einen inneren Knoten und füge den Schlüssel abhängig vom Wert des Knotens als linkes oder rechtes Kind an.

binary search – binäre Suchbäume

Die Form eines Suchbaumes hängt von der Einfügereihenfolge ab. war die Liste sortiert, verkommt der Baum zum Beispiel zu einer Liste.
Es gibt bessere Datenstrukturen, die das verhindern, z.B. AVL-Bäume.

binary search – die Bewertung

- braucht eine sortierte Datenstruktur
- dann aber mit maximal $\log(n)$ Vergleichen sehr schnell
- ist das Standardverfahren für Suchen

Teil VI

Bibliotheksfunktionen

C

qsort

```
#include <stdlib.h>

void qsort(void *base, size_t nel,
           size_t width,
           int (*compare) (const void *,
                           const void *)
           );
```

qsort sortiert die ersten *nel* Elemente eines Arrays *base* aus Elementen der Größe *width* bytes mittels der Funktion *compare*.

$$compare(x, y) = \begin{cases} 1 & falls\ x > y \\ -1 & falls\ x < y \\ 0 & falls\ x = y \end{cases}$$

bsearch

```
#include <stdlib.h>

void bsearch(const void *key, const void *base,
             size_t nmemb, size_t width,
             int (*compare) (const void *,
                             const void *)
            );
```

bsearch sucht den Schlüssel *key* in den ersten *nmemb* Elementen eines Arrays *base* aus Elementen der Größe *width* bytes mittels der Funktion *compare*.

C Beispiel

Beispiel für eine Funktion, die Integer sortiert oder danach sucht:

```
int intcompare_cast(const void *i, const void *j){
    if (*(int *)i > *(int *)j) return 1;
    if (*(int *)i < *(int *)j) return -1;
    return 0;
}

int intcompare(int *i, int *j){
    if (*i > *j) return 1;
    if (*i < *j) return -1;
    return 0;
}
```

C Beispiel

Aufruf:

```
#define MAXNUMBER 500
#define KEY 42

int main(int *argc, int **argv){
    int array[MAXNUMBER]; /*...*/
    qsort((char *) array,
          MAXNUMBER, sizeof(int),
          intcompare_cast);
    bsearch(KEY, (char *) array,
            MAXNUMBER, sizeof(int),
            (int (*) (const void *, const void *))
                                    intcompare);
}
```

C++

(stable_)sort

```
#include <algorithm>

void sort(RandomAccessIterator begin,
          RandomAccessIterator end);
void sort(RandomAccessIterator begin,
          RandomAccessIterator end,
          BinaryPredicate op);
void stable_sort(RandomAccessIterator begin,
                 RandomAccessIterator end);
void stable_sort(RandomAccessIterator begin,
                 RandomAccessIterator end,
                 BinaryPredicate op);
```

(stable_)sort

Die Methoden sortieren die Elemente zwischen *begin* und *end* mittels des Operators $<$ der Klasse oder mittels einer speziellen Vergleichsfunktion *op*.
Die stable_sort-Varianten sortieren stabil (d.h. gleiche Elemente behalten ihre ursprüngliche Reihenfolge bei).

C++

```
#include <algorithm>

using namespace std;

bool operator<(const Student& a, const Student& b){
    return a.score < b.score;
}

int main(){
    student a[MAXNUMBER];
    sort(a, a+MAXNUMBER);
}
```

C++

```
#include <algorithm>

template<class InputIterator,
         class EqualityComparable>
    InputIterator find(InputIterator first,
                       InputIterator last,
            const EqualityComparable& value);
```

find liefert den ersten *iterator* zwischen *first* und *last*, dessen Wert *value* entspricht;

STL

Die STL hält auch noch einige andere nützliche Funktionen bereit, die in den Kontext passen:

```
#include <algorithm>

template<class RandomAccessIterator> inline
    void nth_element(RandomAccessIterator First,
                     RandomAccessIterator Nth,
                     RandomAccessIterator Last);
//liefert das n-groesste Element aus einem Container

template <class ForwardIterator>
     ForwardIterator unique(ForwardIterator first,
                            ForwardIterator last);
//entfernt aufeinander folgende gleiche Elemente
```

Java

sort

```
import java.util.Arrays;

static void sort(Object[] a);
static void sort(Object[] a, Comparator c);
```

Sortiert das Array *a* aufsteigend.
Wahlweise nach natürlicher Ordnung oder mittels Comparator *c*.

binarySearch

```
import java.util.Arrays;

static void binarysearch(Object[] a, Object key);
static void binarysearch(Object[] a, Object key,
                         Comparator c);
```

Sucht im Array *a* nach dem Objekt *key*.
Wahlweise nach natürlicher Ordnung oder mittels Comparator *c*.

Ausblick

Sorting is a problem of which we can prove a nontrivial lower bound. Our best upper bounds match the lower bound asymptotically, and so we know that our sorting algorithms are asymptotically optimal. An attribute that is very rarely found.

Cormen, Leiserson, Rivest, Stein

Ausblick

Even if sorting was almost useless, there would be plenty of rewarding reasons for studying it anyway! The ingenious algorithms that have been discovered show that sorting is an extremely interesting topic to explore in its own right. Many fascinating unsolved problems remain in this area, as well as quite a few solved ones.

Donald E. Knuth

Literatur

- Donald E. Knuth: The Art of Computer Programming. 2nd Edition. Vol. 3 Sorting and Searching, Addison-Wesley 1998
- Steven S. Skiena: Programming Challenges. The Programming Contest Training Manual. Springer, 2003
- Thomas H. Cormen e.a.: Introduction to Algorithms. 2nd Edition. MIT Press, 2001
- Volker Heun: Grundlegende Algorithmen. Einführung in den Entwurf und die Analyse effizienter Algorithmen. 2. Auflage. Vieweg, 2003
- Volker Strehl: Theoretische Informatik 3. (Vorlesungsskript im WS 2005/06 in gleichnamiger Veranstaltung an der FAU Erlangen/Nürnberg), 2005/06

Danke...

für eure Aufmerksamkeit!

Alles klar?

Gibts Fragen?