Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung

Datenbanksysteme II

Sommersemester 2005

Kapitel 7: Algorithmen zur Ähnlichkeitssuche

Vorlesung: Christian Böhm
Übungen: Elke Achtert, Peter Kunath

http://www.dbs.informatik.uni-muenchen.de/Lehre/DBSII

Inhalt

1. Einführung

2. Multidimensionale Indexstrukturen

3. Anfragetypen

Einführung

- **Vorangegangenes Kapitel:**
  Wie kann man für verschiedene Applikationen Ähnlichkeit formal so definieren, dass der applikationsspezifische, intuitive Ähnlichkeitsbegriff am besten wiedergegeben wird?
  → Effektivität der Ähnlichkeitssuche.

- **Dieses und die folgenden Kapitel:**
  Wie kann man unter Berücksichtigung der vorgegebenen Ähnlichkeitsmaße schnell die zu einem Anfrageobjekt ähnlichen (bzw. ähnlichsten) Datenbankobjekte finden?
  → Effizienz der Ähnlichkeitssuche.

Für die Effizienz in Datenbanken ist der Einsatz geeigneter Indexstrukturen entscheidend. Die Entwicklung spezieller Indexstrukturen für einzelne Anwendungsgebiete oder Ähnlichkeitsmaße ist jedoch im allgemeinen zu aufwendig. Man versucht deshalb mit wenigen Arten von Indexstrukturen möglichst viele Anwendungsgebiete abzudecken.

Inhalt

1. Einführung

2. Multidimensionale Indexstrukturen

3. Anfragetypen

Multidimensionale Indexstrukturen (1)

([BBK 01] Böhm C., Berchtold S., Keim D.A.: Searching in High-dimensional Spaces: Index Structures for Improving the Performance of High-Dimensional Indexing, to appear in ACM Computing Surveys, 2001.)

Generelle Prinzipien vieler Indexstrukturen

- Jedem Knoten des Baums ist zugeordnet:
  - eine *Seite* des Hintergrundspeichers
  - eine *Region* des Datenraums
- Es gibt zwei Typen von Seiten (=Knoten):
  - Datenseiten sind Blattknoten und speichern Punktdaten
  - Directoryseiten sind innere Knoten und speichern *Directory-Einträge*, bestehen aus:
    - *Verweis* zu Kindseite (Adresse auf dem Hintergrundspeicher) und
    - Beschreibung der *Region* der Kindseite

Multidimensionale Indexstrukturen (2)

- Physische vs. logische Seiten:
  - Ursprüngliche Idee: Eine *physische* Seite des Hintergrundspeichers wird verwendet
  - Kleinste Informationseinheit, die zwischen Plattenspeicher und Arbeitsspeicher übertragen werden kann
  - physische Seiten meist zu klein:
    fasse aufeinander folgende physische Seiten zu einer *logischen Seite* zusammen
  - Die meisten Indexstrukturen verwenden aber eine *einheitliche* Seitengröße für alle Seiten eines Indexes, um Probleme wie Freispeicherverwaltung zu vermeiden
  - Aus der Seitengröße ergibt sich auch die *Kapazität* (maximale Anzahl Einträge) der Directory- und Datenseiten:

$$C_{Data} = \left\lfloor \frac{|\text{Seite}| - |\text{Verwaltungsoverhead}|}{|\text{Datensatz}|} \right\rfloor \qquad C_{Dir} = \left\lfloor \frac{|\text{Seite}| - |\text{Verwaltungsoverhead}|}{|\text{Directoryeintrag}|} \right\rfloor$$

Multidimensionale Indexstrukturen (3)

- Meist werden die Seiten nicht zu 100% gefüllt, damit Platz für neu eingefügte Datensätze bleibt. Meist wird eine minimale Speicherauslastung, z.B. 40% definiert.
- Der durchschnittliche Anteil besetzter Einträge wird als *Speicherauslastung* (*storage utilization*), die durchschnittliche Anzahl besetzter Einträge als *effektive Kapazität* bezeichnet mit: $C_{\text{eff,Data}} = su_{\text{Data}} \cdot C_{\text{Data}}$
- Jeder Punkt-Datensatz wird in genau einer Datenseite gespeichert (keine Duplikate)
- Durch die Regionen wird gewährleistet, daß räumlich benachbarte Punkte möglichst auf denselben Datenseiten oder in denselben Teilbäumen gespeichert werden.

Multidimensionale Indexstrukturen (4)

- Verschiedene Möglichkeiten für die Gestalt der Seitenregionen:
  - achsenparallele Rechtecke, die meist minimal um die Punktemenge gespannt werden (*minimum bounding rectangle, MBR*) am weitesten verbreitet (R-tree, X-tree ...)
  - Kugeln (SS-tree)
  - Zylinder (TV-tree)
  - Kombinationskörper (SR-tree)
- Die Seitenregion umfaßt immer geometrisch alle auf einer Datenseite bzw. in dem entsprechenden Teilbaum unter einer Directoryseite gespeicherten Punkte.
  Dies ist Voraussetzung um die Vollständigkeit des Anfrageergebnisses zu gewährleisten (*konservative Approximation* bzw. *lower bounding property*)
- Bäume sind meist *balanciert*: Abstand zwischen Wurzel und alle Blätter gleich groß

Multidimensionale Indexstrukturen (5)

- Indexstrukturen sind *dynamisch*:
  Insert- und Delete-Operationen sind effizient, möglichst in O(log *n*)

  Typische Vorgehensweise:
  - Auf einen Pfad beschränkte *Tiefensuche* nach einer geeigneten *Datenseite*: Nach bestimmten räumlichen Kriterien wird jeweils die beste Kindseite gewählt
  - Wenn auf der Datenseite noch Platz ist, wird der Punkt dort gespeichert
  - Manche Indexstrukturen: Restrukturierungsversuche ohne Anlegen neuer Seite
  - Sonst wird im Rahmen einer Überlaufbehandlung die Datenseite aufgeteilt (*Split*)
    - verschiedene Kriterien (*minimale Überlappung, toter Raum*)
    - verschiedene Algorithmen (*linear, quadratisch*)
    - neuer Knoten wird im Elternknoten eingetragen
    - durch Überlauf von Elternknoten kann Split im Extremfall bis zur Wurzel laufen

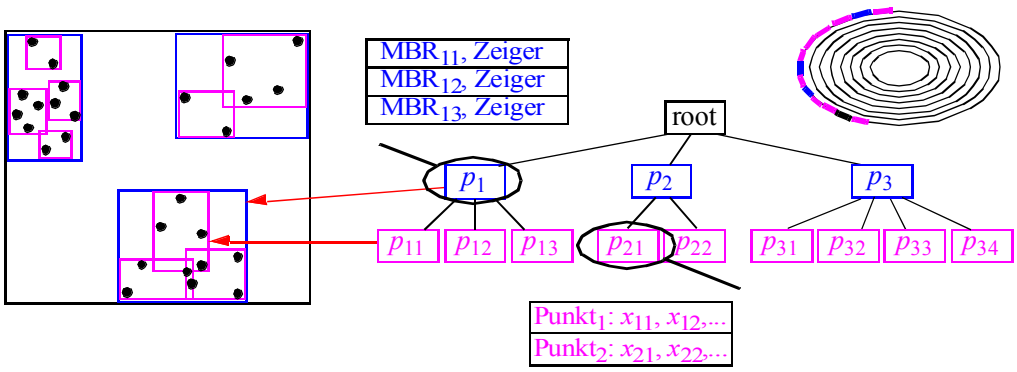
R*-Baum, Konzept

Der R*-Baum ist ein balancierter Baum, der zur Speicherung von 2D Rechteckdaten entworfen wurde. Er wird jedoch häufig auch zur Speicherung höherdimensionaler Punktdaten verwendet und diente als Grundlage für wichtige spezialisierte Indexstrukturen für hochdimensionale Daten wie z.B. den TV-tree oder den X-tree.

- Balancierte Indexstruktur mit Daten- und Directoryseiten einheitlicher Größe
- Seitenregionen: · Minimal umgebende achsenparallele Rechtecke (MBR)
- Insert-Strategie: · Vergrößerung des Overlap
  · Vergrößerung des Volumen
  · Volumen
- Überlaufbehandlung (Re-Insert-Konzept):
  · Teil der Punkte wird gelöscht und in den Baum neu eingefügt
- Split-Kriterien: · Umfang/Oberfläche
  · Überlappungsvolumen
  · Toter Raum (nicht sinnvoll bei Datenseiten mit Punktdaten)
- Split-Algorithmus: erste Phase: Ermittlung der Dimension
  zweite Phase: Ermittlung der Splitebene
  (jeweils durch Sortieren)

R*-Baum, Beispiel

- Beispiel eines R*-Baums:



Inhalt

1. Einführung

2. Multidimensionale Indexstrukturen

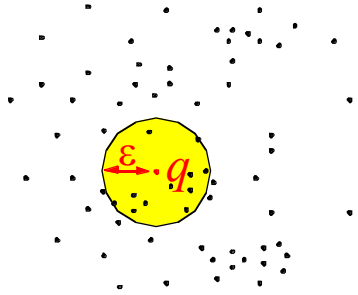
3. Anfragetypen

Bereichsanfragen (1)

Allgemeines

- Charakteristik:
  Benutzer gibt Anfrageobjekt *q* und maximale Distanz e vor;
  System ermittelt alle Datenbankobjekte, die von *q* höchstens die Distanz ε haben

- Formale Definition:
  $sim_q(\varepsilon) := \{o \in DB \mid d(q,o) \leq \varepsilon\}$

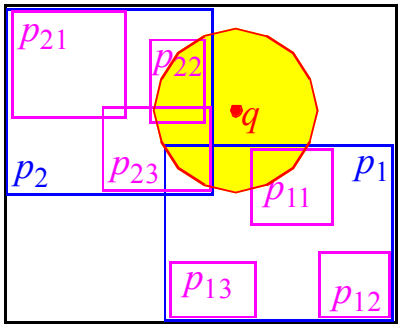


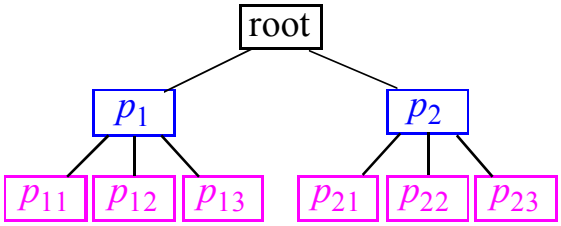
Bereichsanfragen, Algorithmus (1)

Algorithmus mit multidimensionalem Index: Tiefensuche (rekursiv)

```
function RangeQuery (q: Point; e: Real; pa: DiskAddress): Set of Point
    result := ∅ ;
    p := LoadPage (pa) ;
    if IsDataPage (p) then
        for i := 0 to p.num_objects do
            if distance (q, p.object [i]) ≤ ε then
                result := result ∪ p.object [i] ;
    else
        for i := 0 to p.num_objects do
            if MINDIST (q, p.region [i]) ≤ ε then
                result := result ∪ RangeQuery (q, p.childpage [i]) ;
    return result ;
```

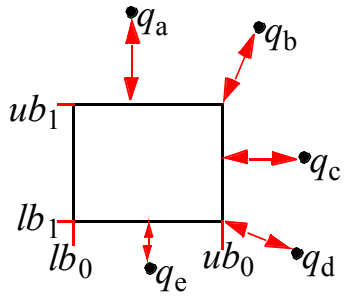
Bereichsanfragen, Algorithmus (2)





- Der Test, ob sich die Queryregion mit einer Seitenregion schneidet, wird am einfachsten mit Hilfe der Distanz zwischen dem Anfragepunkt und der Seitenregion entschieden. In diesem Fall wird die *minimale* Distanz zwischen dem Anfragpunkt und einem Punkt der Seitenregion benötigt (MINDIST).

Bereichsanfragen, Algorithmus (3)



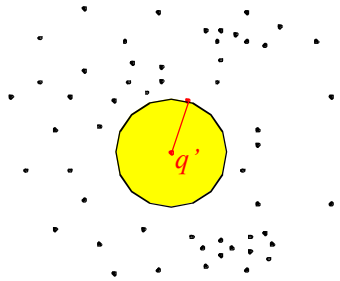
- Berechnung der MINDIST für den euklidischen Abstand:

$$\text{MINDIST}(p,q) = \sqrt{\sum_{0<i\le d} \begin{cases} (lb_i - q_i)^2 & \text{if } q_i \le lb_i \\ 0 & \text{if } lb_i \le q_i \le ub_i \\ (q_i - ub_i)^2 & \text{if } ub_i \le q_i \end{cases}}$$

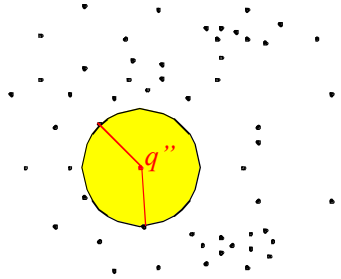
Nächste-Nachbarn-Anfragen (1)

Allgemeines

- Benutzer gibt Anfrageobjekt q vor;
  System ermittelt das Datenbankobjekt, das von q den geringsten Abstand hat
- *Mehrdeutigkeiten* können auftreten und müssen sinnvoll behandelt werden:
  entweder *mehrere* nächste Nachbarn, oder das Ergebnis ist nichtdeterministisch.
- Formale Definition:
  $NN_q := \{o \in DB \mid \forall\, o' \in DB\ d(q,o) \leq d(q,o')\}$



q‘ hat eindeutigen nächsten Nachbarn



q‘‘ hat zwei nächste Nachbarn

- Nichtdeterministische Variante:
  $NN_q := \text{SOME } o \in DB \mid \forall\, o' \in DB\ d(q,o) \leq d(q,o')$

Nächste-Nachbarn-Anfragen (2)

Einfacher Tiefensuche-Algorithmus

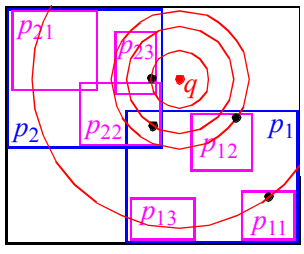
- Unterschied zu bisherigen Anfragealgorithmen:
  Da der nächste Nachbar prinzipiell beliebig weit vom Anfragepunkt entfernt sein kann, ist die Gestalt der Query zunächst unbekannt, d.h. es ist nicht wie bei der Range-Query leicht anhand der Seitenregion zu entscheiden ob eine Seite gebraucht wird. Ob eine Seite gebraucht wird, hängt *auch* davon ab was auf den anderen Seiten gespeichert ist.
- Wäre der Abstand zum nächsten Nachbarn (bzw. eine obere Schranke hierfür) bekannt, würde Range-Query ausreichen.
  - Kennt man *einen beliebigen* Punkt der Datenbank, dann kann man dessen Abstand als obere Schranke für die Nearest-Neighbor-Distance nutzen.
  - Kennt man *mehrere* Punkte der Datenbank, dann kann man den *geringsten* Abstand als obere Schranke für die Nearest-Neighbor-Distance nutzen.
- Der Algorithmus für Range-Queries wird deshalb so umformuliert, daß ε durch die Distanz zum besten, bisher gefundenen Nachbarn (*resultdist*) ersetzt wird. Die globalen Variablen *result* und *resultdist* werden wie oben mit $\perp$ bzw. $+\infty$ initialisiert

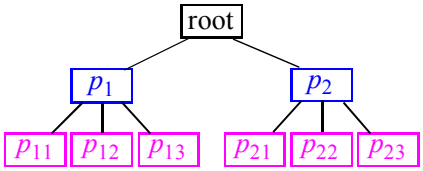
Nächste-Nachbarn-Anfragen (3)


```
procedure SimpleNNQuery (q: Point; pa: DiskAddress)
    p := LoadPage (pa) ;
    if IsDataPage (p) then
        for i := 0 to p.num_objects do
            if distance (q, p.object [i]) ≤ resultdist then
                result := p.object [i] ;
                resultdist := distance (q, p.object [i]) ;
    else
        for i := 0 to p.num_objects do
            if MINDIST (q, p.region [i]) ≤ resultdist then
                SimpleNNQuery (q, p.childpage [i]) ;
```


Nächste-Nachbarn-Anfragen (4)

- Nachteil des einfachen Tiefensuche-Algorithmus:
  - Fängt mit *resultdist* = +∞ an
  - Hierdurch: Start mit einem beliebigen Pfad, nicht mit einem Pfad, der möglichst nahe am Anfragepunkt liegt
  - Dadurch sind auch die ersten gefundenen Punkte sehr weit vom Anfragepunkt weg
  - D.h. das Verfahren schränkt seinen Suchraum nur sehr langsam ein. Viele Pfade werden unnötigerweise verfolgt.





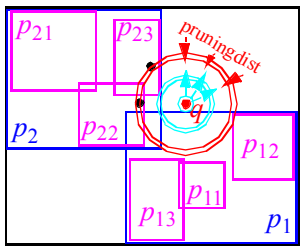
Prioritätssuche nach Hjaltason und Samet (1)

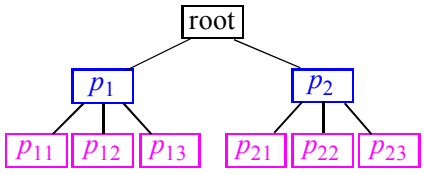
([HS 95] Hjaltason G. R., Samet H.: *Ranking in Spatial Databases*, Int. Symp. on Large Spatial Databases (SSD), 1995.)

- Statt eines rekursiven Durchlaufs durch den Index hält der Algorithmus explizit eine *Liste der aktiven Seiten* (*active page list APL*).
- Definition: Eine Seite p ist *aktiv* genau dann wenn folgende Bedingungen erfüllt sind:
  - p wurde noch nicht geladen
  - die Elternseite von p wurde bereits geladen
  - die Distanz (MINDIST) zwischen der Region *p.region* und dem Anfragepunkt übersteigt nicht die Pruningdistanz (Distanz des bisherigen nächsten Nachbarn)
- Anfangs wird *APL* mit der Wurzel des Index initialisiert.
- Der Algorithmus entnimmt in jedem Schritt die Seite aus der APL, die die höchste Priorität (d.h. die geringste MINDIST vom Anfragepunkt q) hat.
- Die entnommene Seite wird geladen und verarbeitet:
  - Datenseiten werden wie bisher verarbeitet
  - Directoryseiten: Kindseiten mit MINDIST ≤ *pruningdist* werden in *APL* eingefügt
  - Ändert sich *pruningdist*, dann werden die betroffenen Seiten aus *APL* gelöscht (Alternativ können diese Seiten auch nur als "gelöscht" markiert bzw. auch ohne explizite Markierung später ignoriert werden)

Prioritätssuche nach Hjaltason und Samet (2)

- Beispiel:





APL:

1. root
2. p_1 p_2
3. p_2 p_{12} p_{13} p_{11}
4. p_{23} p_{22} p_{12} p_{13} p_{11} p_{21}
5. p_{22} p_{12} p_{13}

- Beobachtung:
  - Seiten werden nach aufsteigendem Abstand (blaue Kreise) geordnet zugegriffen
  - *pruningdist* (rote Kreise) wird kleiner, sobald nähergelegener Datenpunkt gefunden
  - die Anfragebearbeitung stoppt, wenn sich beide Kreise treffen

Prioritätssuche nach Hjaltason und Samet (4)

Algorithmus:

apl: **list of** (*dist*: Real, *pa*: DiskAdr) **ordered by** *dist* **ascending** := ⟨(0.0, root)⟩ ;
while not empty (*apl*) **and** *apl* [0].*dist* ≤ *resultdist* **do**
    p := LoadPage (*apl* [0].*pa*) ;
    delete_first (*apl*) ;
    if (IsDataPage (*p*)) **then**
        (* wie üblich *)
    else
        for *i* := 0 **to** *p.num_objects* **do**
            h := MINDIST (*q*, *p.region* [*i*]) ;
            if *h* ≤ *resultdist* **then**
                insert ((*h*, *p.childpage* [*i*]), *apl*) ;

k-nächste-Nachbar-Anfragen (1)

Allgemeines

- Der Benutzer spezifiziert einen Anfragepunkt *q* und eine Anzahl *k*. Das System ermittelt die *k* nächsten Nachbarn von *q*.
- formale Definition:
  kleinste Menge $NN_q(k) \subseteq DB$ mit mindestens *k* Objekten, so dass
  - $\forall\, o' \in NN_q \;\; \forall\, o' \in DB \setminus NNq$: $d(q,o) < d(q,o')$
- nichtdeterministische Variante:
  Menge $NN_q(k) \subseteq DB$ mit exakt *k* Objekten, so dass
  - $\forall\, o' \in NN_q \;\; \forall\, o' \in DB \setminus NN_q$: $d(q,o) \leq d(q,o')$

k-nächste-Nachbar-Anfragen (2)

Grundalgorithmus ohne Index (nichtdeterministisch)

```
result: list of (dist: Real, P: Point) ordered by dist descending := ⟨⟩ ;
for i := 1 to k do
    insert ((distance (q, database [i]), database [i]), result) ;
for i := k + 1 to n do
    if distance (q, database [i]) ≤ result [0].dist then
      delete_first (result) ;
      insert ((distance (q, database [i]), database [i]), result) ;
```

Anmerkung: Die Ergebnisliste *result* ist hier *absteigend* geordnet, obwohl dies der Intuition aus Anwendersicht widerspricht (Intuition: erstes Element = bestes Element = erster NN). Der Grund ist, dass bei absteigender Ordnung Datenstrukturen verwendet werden können, die einen besonders effizienten Zugriff auf das *erste* Element erlauben.

k-nächste-Nachbar-Anfragen (3)

Algorithmen mit Index

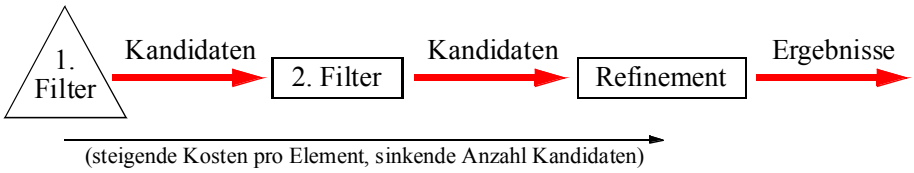
Grundsätzlich lassen sich sowohl die Tiefensuche-Algorithmen als auch der Prioritätsalgorithmen erweitern so dass sie *k* nächste Nachbarn suchen. Als Pruning-Distanz dient grundsätzlich die Distanz des am weitesten entfernten Eintrags in der Ergebnisliste *result*[0].*dist*. (Liste ist hier absteigend sortiert)

NN-Suche mit mehrstufiger Anfragebearbeitung

- Häufig wird bei der Anfragebearbeitung eine Multistep-Architektur eingesetzt, die aus einem oder mehreren Filterschritten sowie einem Verfeinerungsschritt besteht
- Um die Vollständigkeit des Anfrageergebnisses garantieren zu können, ist die Lower-Bounding-Property nachzuweisen, d.h. die im Filter ermittelte Distanz ist höchstens so groß wie die Distanz des Verfeinerungsschrittes
- Analog gilt bei mehreren Filtern $dist_{Filter1} \leq dist_{Filter2} \leq ...$

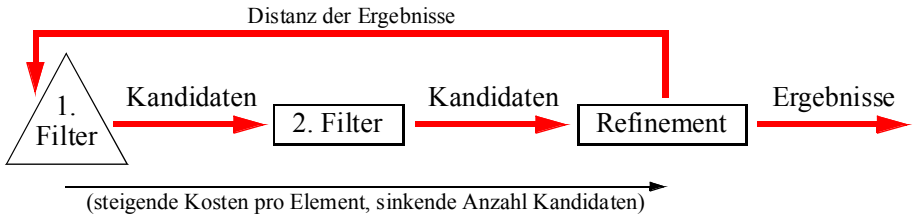
k-nächste-Nachbar-Anfragen (4)

- *Bereichsanfragen* können dann durch einfache kaskadierte Anwendung (ohne weitere Modifikation) der Filterschritte und des Verfeinerungsschrittes ausgewertet werden:



- Für Nearest-Neighbor-Queries gilt dies nicht, denn der im (1.) Filterschritt ermittelte nächste Nachbar ist nicht notwendig auch das Element mit minimaler $dist_{Refinement}$.
- Bei einer geeigneten Filterfunktion ist es aber *wahrscheinlich*, dass das Element mit minimaler $dist_{Refinement}$ auch unter *den ersten* Elementen des Filterschrittes ist.
- Man benötigt eine Art "Rückmeldung" der im Verfeinerungsschritt ermittelten Distanz an den Filterschritt.

k-nächste-Nachbar-Anfragen (5)



- Analog zu den NN-Algorithmen ohne Multistep-Architektur gibt es verschiedene Auswertungsstrategien, die sich in Speicherplatz- und Zeitkomplexität unterscheiden.