Algorithmen und Datenstrukturen
Hashverfahren

Matthias Teschner
Graphische Datenverarbeitung
Institut für Informatik
Universität Freiburg

SS 09

Überblick

- Prinzip
- Details
- Anwendungen

Motivation Hashverfahren (Hashtabellen)

- Datenstruktur
- effiziente Implementierung dynamischer Mengen, insbesondere Wörterbuchoperationen
  - `insert, search, delete`
- mittlere Laufzeit für Wörterbuchoperationen O (1)
  - schlechtester Fall O (n)

Grundlagen

- basiert auf einem Feld
  - schneller direkter Zugriff auf beliebige Position in O (1)
- Position eines Elements ergibt sich aus seinem Schlüssel
  - Schlüssel s
  - Hashfunktion h
  - Hashwert i = h(s)

- Für jeden Schlüssel s liefert die Hashfunktion h einen Hashwert (Hashindex) i = h(s), der die Position (Bucket) des Elements im Feld (in der Hashtabelle) bezeichnet.

Kleine Schlüsselmenge

- Schlüssel {4, 7, 8, 12}
- Hashfunktion h (s) = s (direkter Zugriff)
- Hashtabelle t [0..40] der Größe n=41

- `t.insert (4);`
  - h (4) = 4; t [4] = 4;
- `t.insert (7); t.insert (8); t.insert (12);`
  - h (7) = 7; t [7] = 7; h (8) = 8; t [8] = 8; h (12) = 12; t [12] = 12;
- `t.delete (7);`
  - h (7) = 7; t [7] = -1;
- `t.search (8);`
  - h (8) = 8; t [8] == 8 $\rightarrow$ true;

Große Schlüsselmenge

- Menge von möglichen Schlüsseln ist groß gegenüber der zu repräsentierenden Menge von Elementen
  - direkte Adressierung ($h(s) = s$) nicht praktikabel:
  - sehr große Hashtabellengröße m,
  - sehr geringer Belegungsfaktor
    (Zahl der Elemente n / Größe der Hashtabelle m)
- Wahl der Hashtabellengröße m in der Größenordnung der zu repräsentierenden Elemente
- Wahl der Hashfunktion, sodass alle Schlüssel auf einen gültigen Index der Hashtabelle abgebildet werden, beispielsweise $h(s) = s \bmod m$ ($0 \leq h(s) \leq m-1$)

Große Schlüsselmenge

- Schlüsselmenge - natürliche Zahlen
- Schlüssel {17, 235436, 143251348}
- Hashtabelle t [0..n-1] mit m=10
  - (m=10 Einträge für n=3 Elemente)
- Hashfunktion h (s) = s mod m

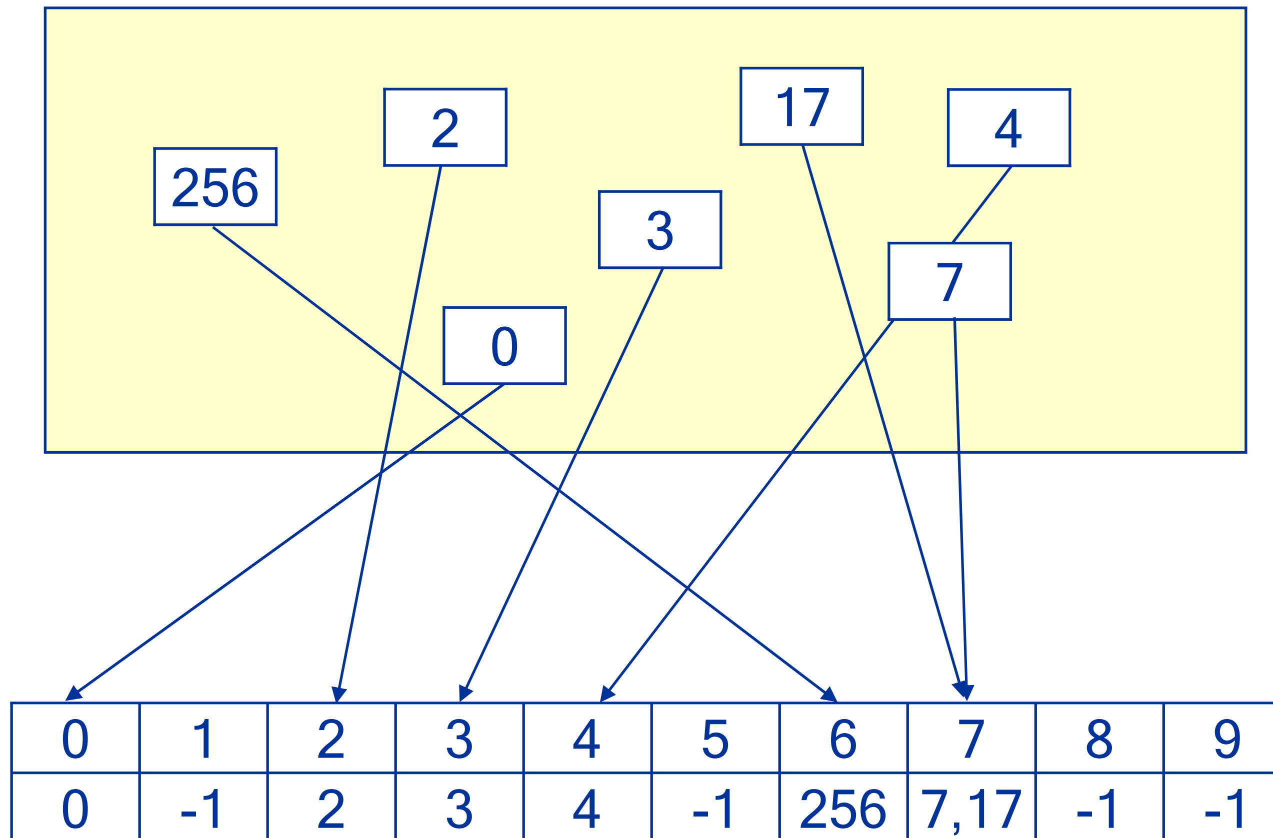
- `t.insert (17);`
  - h (17) = 7; t [7] = 17;
- `t.insert (235436); t.insert (143251348);`
  - h (235436) = 6; t [6] = 6; h (143251348) = 8; t [8] = 143251348;
- `t.search (143251348);`
  - h (143251348) = 8; t [8] == 143251348 $\rightarrow$ true;

Hashkollision

- Enthält die Menge zwei verschiedene Schlüssel, welche den gleichen Hashwert liefern, tritt eine Hashkollision auf.
- Schlüssel {17, 235436, 143251348, 6}
- Hashfunktion h (s) = s mod 10
- h (235436) = h (6) = 6

Hashverfahren / Hashing

Menge mit n=7 Elementen



0	1	2	3	4	5	6	7	8	9
0	-1	2	3	4	-1	256	7,17	-1	-1

Hashtabelle t der Größe m=10

Hashfunktion:
h (s) = s mod 10

Hashwerte /
Hashindizes:
h(256) = 6
h(2) = 2
h(0) = 0
h(3) = 3
h(17) = 7
h(7) = 7
h(4) = 4

Hashkollision:
h(7) = h(17)

Implementierung von Hashtabellen

- Wahl der Hashtabellengröße m in Größenordnung der Anzahl der zu repräsentierenden Elemente n
  - Belegungsfaktor n / m < 1, aber nicht zu klein
- Wahl der Hashfunktion h
  - effiziente Berechnung, Vermeidung von Hashkollisionen
- Behandlung von Kollisionen (Kollisionsauflösung)
  - Strategie zur Behandlung von Fällen, in denen mehrere Elemente auf den gleichen Hashwert abgebildet werden

Überblick

- Prinzip
- Details
  - Hashtabellengröße
  - Hashfunktion
  - Behandlung von Hashkollisionen
- Anwendungen

Motivation

- großer Belegungsfaktor
  → möglichst kleine Tabelle
- Zahl der Elemente n / Hashtabellengröße m sollte kleiner eins, aber nicht zu klein sein (ungenutzter Speicher)
- Vermeidung von Hashkollisionen
  → möglichst große Tabelle
- große Tabellen verringern eventuell die Wahrscheinlichkeit von Hashkollisionen

Wahl der Hashtabellengröße

- schlechte Größen m
  - sind einfach zu finden
  - m gerade
  - $m = r^k \pm j$, r = {2, 10} Basis der Elementrepräsentation,
    k und j kleine natürliche Zahlen
  - Beispiele: 2, 8, 10, 16, 999, 10003, 65541 ($256*256+5 = 2^{16}+5$)
- nicht ganz so schlechte Größen m
  - große Primzahl, (die nicht Teiler von $r^k \pm j$ ist) [Knuth 1973]
  - Beispiele: 5657, 55931, 563359

Überblick

- Prinzip
- Details
  - Hashtabellengröße
  - Hashfunktion
  - Behandlung von Hashkollisionen
- Anwendungen

Hashfunktion

- h : `Schlüssel` → `Hashindex`
- `Schlüssel`
  Menge aller möglichen Schlüssel
- `Hashindex` ⊆ { 0, …, m-1 }
  Addressraum (Menge aller Positionen in der Hashtabelle)

Motivation

- effiziente Berechnung
- Zahl der Hashkollisionen für die zu erwartende Schlüsselmenge und die gewählte Hashtabellengröße möglichst klein (Gleichverteilung der Hashwerte)
- stark unterschiedliche Hashwerte für ähnliche Schlüssel (Chaos)
- alle Hashwerte sollten möglich sein (Surjektivität)
- injektive Hashfunktionen heissen perfekt, da Kollisionen vollständig vermieden werden

Hashkollisionen

- treten leider häufiger auf, als man intuitiv annimmt
- Geburtstagsproblem
  - Die Wahrscheinlichkeit, dass sich unter 23 Personen zwei Personen befinden, die am gleichen Tag (ohne Berücksichtigung des Jahres) Geburtstag haben, liegt bei über 50%. Bei 57 Personen liegt die Wahrscheinlichkeit bei 99%.
  - Nicht zu verwechseln mit der Wahrscheinlichkeit, dass jemand am gleichen Tag Geburtstag hat wie ich. Für eine Wahrscheinlichkeit von 50% sind hier 253 Personen nötig.

Hashkollisionen

- Konsequenz für Hashkollisionen
  - Hashtabellengröße 365
  - Hashfunktion liefert gleichverteilte Werte
  - 23 Elemente
  - → Wahrscheinlichkeit für das Auftreten einer Hashkollision bei 50%
  - 57 Elemente
  - → Wahrscheinlichkeit für das Auftreten einer Hashkollision bei 99%

Divisionsrestmethode

- $h(s) = s \bmod m$
- sehr effizient berechenbar
- für $m = 2^k$ liefert h (s) die k niederwertigsten Bits von s, Variationen der anderen Bits werden nicht berücksichtigt
- wird in der Regel in Kombination mit weiteren Funktionen eingesetzt in der Form $h(s) = f(s) \bmod m$, z. B. $h(s) = ((a \cdot s + b) \bmod p) \bmod m$

Multiplikative Methode

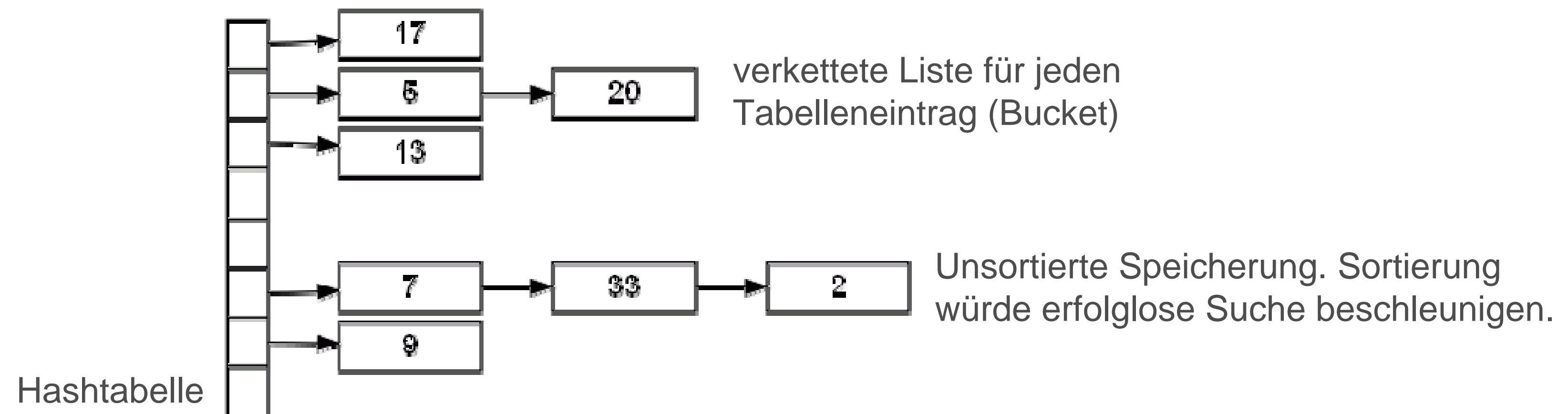
- $h(s) = \lfloor m(\alpha \cdot s - \lfloor \alpha \cdot s \rfloor) \rfloor$ $0 < \alpha < 1$
- möglichst irrationale Werte für α, z. B. $\alpha = \frac{\sqrt{5}-1}{2}$
- weniger sensitiv bezüglich m

Überblick

- Prinzip
- Details
  - Hashtabellengröße
  - Hashfunktion
  - Behandlung von Hashkollisionen
- Anwendungen

Verkettung

- Hashtabelleneintrag ist Kopf einer verketteten Liste
- kollidierende Schlüssel werden gemeinsam in die Liste eines Eintrags (Bucket) sortiert oder am Ende eingefügt



- Operationen in O(1) bei sinnvoller Wahl von Tabellengröße und Hashfunktion (wenig Elemente pro Bucket)
- schlechtester Fall O(n), z. B. bei Tabellengröße 1
- dynamisch, variable Zahl von Elementen möglich

Offene Hashverfahren

- Für kollidierende Schlüssel (Überläufer) wird ein freier Eintrag in der Tabelle gesucht.
- statisch, Zahl der Elemente fest
- Sondierungsreihenfolge bestimmt für jeden Schlüssel, in welcher Reihenfolge alle Hashtabelleneinträge auf einen freien Platz durchsucht werden.
  - Ist ein Eintrag belegt, kann beispielsweise iterativ der nächstfolgende Tabelleneintrag geprüft werden. Wird ein freier Eintrag gefunden, wird das Element eingetragen.
  - Wird bei der Suche ein Element nicht am entsprechenden Tabelleneintrag gefunden, obwohl der Eintrag belegt ist, muss ebenfalls die definierte Sondierungsreihenfolge abgearbeitet werden, bis das Element oder ein freier Eintrag gefunden wurde.

Prinzip

- h (s) - Hashfunktion für Schlüssel s
- g (s, j) - Sondierungsfunktion für Schlüssel s mit $0 \leq j \leq m-1$
- (h (s) - g (s, j)) mod m - Sondierungsreihenfolge, d. h. Permutation von < 0, 1, ..., m-1 >

- Einfügen

```
j:=0;
while ( t [( h (s) - g (s, j) ) mod m] != frei ) j++;
t [( h (s) - g (s, j) ) mod m] := s;
```

- Suchen

```
j:=0;
while ( t [( h (s) - g (s, j) ) mod m] != frei &&
        t [( h (s) - g (s, j) ) mod m] != s ) j++;
if (t [( h (s) - g (s, j) ) mod m] == s) return true;
   else return false;
```

Lineares Sondieren

- g (s, j) = j → Hashfunktion (h (s) - j) mod m
- führt zu Sondierungsreihenfolge
  h(s), h(s)-1, h(s)-2, ..., 0, m-1, ..., h(s)+1
- einfach
- führt aber zu primärer Häufung (primary clustering)
- Behandlung einer Hashkollision erhöht die Wahrscheinlichkeit einer Hashkollision in benachbarten Tabelleneinträgen

Beispiel

- Schlüssel: {12, 53, 5, 15, 2, 19},
  Hashfunktion: h(s, j) = (s mod 7 - j) mod 7

- t.insert(12); h(12,0)=5;

0	1	2	3	4	5	6
					12	

- t.insert(53); h(53,0)=4;

0	1	2	3	4	5	6
				53	12	

- t.insert(5); h(5,0)=5;
  h(5,1)=4; h(5,2)=3;

0	1	2	3	4	5	6
			5	53	12	

- t.insert(15); h(15,0)=1;

0	1	2	3	4	5	6
	15		5	53	12	

- t.insert(2); h(2,0)=2;

0	1	2	3	4	5	6
	15	2	5	53	12	

- t.insert(19);
  h(19,0)=5; h(19,1)=4; h(19,2)=3;
  h(19,3)=2; h(19,4)=1; h(19,5)=0;

0	1	2	3	4	5	6
19	15	2	5	53	12	

Quadratisches Sondieren

- Motivation: Vermeidung von lokalen Häufungen
- $g(s, j) = (-1)^j \lceil j / 2 \rceil^2$
- führt zu Sondierungsreihenfolge
  h(s), h(s)+1, h(s)-1, h(s)+4, h(s)-4, h(s)+16, h(s)-16
- für Primzahl $m = 4 \cdot k+3$ ist die Sondierungsreihenfolge eine Permutation der Indizes der Hashtabelle
- alternativ auch $h(s, j) = (h(s) - c1 \cdot j - c2 \cdot j^2) \bmod m$

  c1, c2 – Konstanten
- Problem der sekundären Häufung
  - keine lokale Häufung mehr, allerdings durchlaufen Schlüssel mit gleichem Hashwert immer die gleiche Ausweichsequenz

Uniformes Sondieren

- Motivation: Funktion g (s, j) berücksichtigt beim linearen und quadratischen Sondieren lediglich den Schritt j. Die Sondierungsreihenfolge ist vom Schlüssel unabhängig.
- Uniformes Sondieren berechnet die Folge g (s, j) von Permutationen aller möglichen Hashwerte in Abhängigkeit vom Schlüssel s
- Vorteil: Häufung wird vermieden, da unterschiedliche Schlüssel mit gleichem Hashwert zu unterschiedlichen Sondierungsreihenfolgen führen
- Nachteil: schwierige praktische Realisierung

Double Hashing

- Motivation: Berücksichtung des Schlüssels s in der Sondierungsreihenfolge
- Verwendung zweier unabhängiger Hashfunktionen h_1, h_2
- $h(s, j) = (h_1(s) + j \cdot h_2(s)) \bmod m$
- funktioniert praktisch sehr gut
- Approximation des uniformen Sondierens

Beispiel

- $h_1 (s) = s \bmod 7$
- $h_2 (s) = 1 + (s \bmod 5)$
- $h(s, j) = (h_1 (s) + j \cdot h_2 (s)) \bmod 7$

s	10	19	31	22	14	16
$h_1 (s)$	3	5	3	1	0	2
$h_2 (s)$	1	5	2	3	5	2

Effizienz beruht auf
h1 (s) != h2 (s) → Sondierungs-
reihenfolge abhängig von s

Verbesserung des Double Hashing nach Brent

- Motivation: Da unterschiedliche Schlüssel unterschiedliche Sondierungsreihenfolgen haben, hat die Reihenfolge des Einfügens der Schlüssel Einfluss auf die Effizienz der erfolgreichen Suche
- Beispiel:
  - s1 wird an Position p1 = h (s1, 0) eingefügt.
  - s2 liefert ebenfalls p1 = h (s2, 0).
  - h (s2, 1..n) sind ebenfalls belegt.
  - s2 wird an Position h (s2, n+1) eingetragen, was bei der Suche recht ineffizient ist.
  - Brents Idee: Teste, ob h (s1, 1) frei ist.
  - Falls ja, wird s1 von Position h (s1,0) nach Position h (s1,1) verschoben und s2 an Position h (s1, 0) eingetragen.

Ordered Hashing

- Motivation: Kollidierende Elemente werden in sortierter Reihenfolge in der Hashtabelle abgelegt. Dadurch kann bei erfolgloser Suche von Elementen in Kombination mit lin. Sondierung oder bei double hashing früher abgebrochen werden, da hier einzelne Sondierungsschritte feste Länge haben.
- Realisierung:
  - Bei einer Kollision werden beide Schlüssel verglichen.
  - Der kleinere Schlüssel wird abgelegt.
  - Für den größeren Schlüssel wird eine neue Position gemäß Sondierungsreihenfolge gesucht.
- Beispiel:
  - 12 ist an Position p1 = h (12, 0) gespeichert.
  - 5 liefert p1 = h (5, 0). 5<12 → 5 wird an Position p1 eingetragen.
  - Für 12 werden die Positionen h (12, 1 …) weiter getestet.

Robin-Hood-Hashing

- Motivation: Angleichung der Länge der Sondierungsfolgen für alle Elemente. Gesamtkosten bleiben gleich, aber gerecht verteilt. Führt zu annähernd gleichen Suchzeiten für alle Elemente.
- Realisierung:
  - Bei einer Kollision zweier Schlüssel s1 und s2 mit p1 = h (s1, j1) = h (s2, j2) werden j1 und j2 verglichen.
  - Schlüssel mit größerer Länge der Sondierungsfolge wird an Position p1 gespeichert. Der andere Schlüssel erhält neue Position.
- Beispiel:
  - 12 ist an Position p1 = h (12, 7) gespeichert.
  - 5 liefert p1 = h (5, 0). $0<7 \rightarrow$ 12 bleibt an Position p1.
  - Für 5 werden die Positionen h (5, 1 …) weiter getestet.

Implementierung Einfügen / Entfernen

- Entfernen von Elementen kann problematisch sein.
  - Schlüssel s1 wird an Position p1 eingefügt.
  - Schlüssel s2 liefert den gleichen Hashwert, wird aber durch das Sondieren an Position p2 eingefügt.
  - Wird Schlüssel s1 entfernt, kann s2 nicht wiedergefunden werden.
- Lösung.
  - Entfernen: Elemente werden nicht entfernt, sondern als gelöscht markiert.
  - Einfügen: Als gelöscht markierte Einträge werden überschrieben.

Zusammenfassung Kollisionsbehandlung

- Verkettung (dynamisch, Zahl der Elemente variabel)
  - kollidierende Schlüssel werden in Liste gespeichert
- Offene Hashverfahren (statisch, Zahl der Elemente fest)
  - Bestimmung einer Ausweichsequenz (Sondierungsreihenfolge), Permutation aller Hashwerte (Indizes der Hashtabelle)
  - lineares, quadratisches Sondieren: einfach, führt zu Häufungen, da Sondierungsreihenfolge vom Schlüssel unabhängig
  - uniformes Sondieren, double hashing: unterschiedliche Sondierungsreihenfolgen für unterschiedliche Schlüssel, vermeidet Häufungen von Elementen
- Effizienzsteigerung der Suche durch Umsortieren von Elementen beim Einfügen (Brent, Ordered Hashing)

Zusammenfassung Hashverfahren

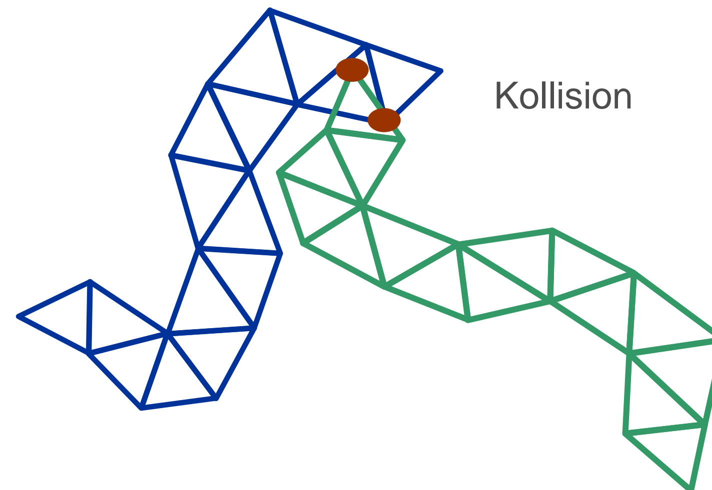
- effiziente Wörterbuchoperationen:
  Einfügen, Suchen, Entfernen
- direkter Zugriff auf Elemente einer Hashtabelle
- Berechnung der Position in der Hashtabelle durch Hashfunktion (Hashwert)
- gleiche Hashwerte für unterschiedliche Schlüssel führen zu Hashkollisionen
- Hashfunktion, Größe der Hashtabelle und Strategie zur Vermeidung von Hashkollisionen beeinflussen die Effizienz der Datenstruktur.

Überblick

- Prinzip
- Details
- Anwendungen
  - Kollisionsdetektion
  - Nachbarschaft eines Punktes

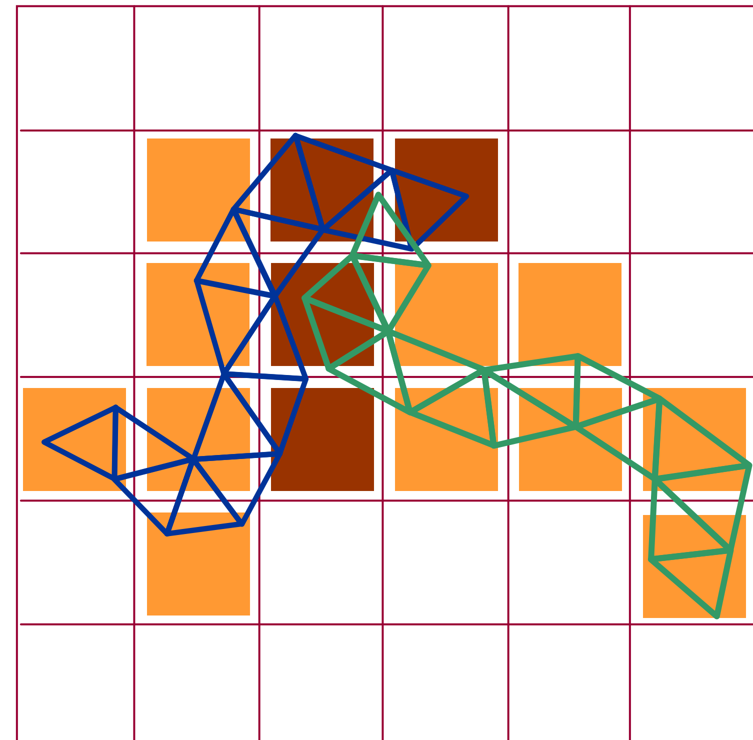
Kollisionsdetektion

- Objekte sind durch Dreiecke (2D) bzw. Tetraeder (3D) beschrieben
- Kollision liegt vor, wenn Punkt (x, y) eines Dreiecks innerhalb eines Dreiecks eines anderen Objektes liegt
- naive Lösung
  - teste alle Dreiecke gegeneinander
  - n Dreiecke $\rightarrow$ O(n^2) Tests



Beschleunigung durch Raumunterteilung

- Raum wird in kleine Zellen unterteilt
- Dreiecke werden den Zellen zugeordnet
- Dreiecke in der gleichen Zelle werden auf Kollision getestet
- Paare von Dreiecken, die nicht die gleiche Raumzelle teilen, werden nicht getestet (trivial reject)

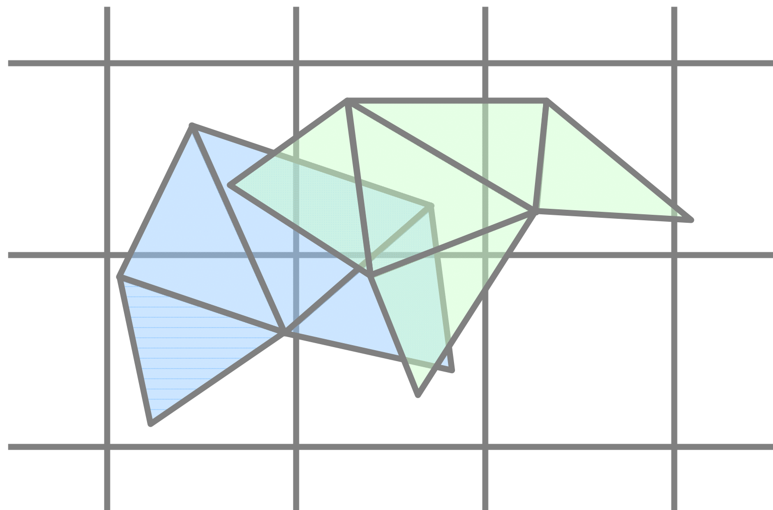


Zuordnung von Dreiecken zu Raumzellen

- erfolgt implizit
- Raumzelle eines Dreiecks wird als Schlüssel betrachtet
- Repräsentation der Dreiecke in einer Hashtabelle
- Motivation: gleiche Raumzelle $\rightarrow$ gleicher Hashwert
- Alle Dreiecke einer Raumzelle werden der gleichen Position in der Hashtabelle zugeordnet.
- Kollisionstest nur für Dreiecke innerhalb der gleichen Raumzelle
- Dreieckspaare mit unterschiedlichen Hashwerten können nicht kollidieren.

Realisierung

unbegrenzt großes
räumliches Gitter



Hashfunktion:

H (Raumzelle) → Hashwert

H (x, y) bzw. H (x, y, z)

Schlüssel besteht aus zwei (2D) oder drei (3D) reelwertigen Zahlen

Hashtabelle

(räumliche Datenstruktur)

(Repräsentation)

Implementierung – Schritt 1

- Punkte werden Raumzellen zugeordnet und entsprechend ihrem Hashwert in die Hashtabelle eingetragen

Zellgröße h

Behandlung von Hashkollisionen durch Listen

Punkt p: (Px, Py)

$(x \leq Px < x+h) \&\& (y \leq Py < y+h) \rightarrow t [h (x, y)] = p$

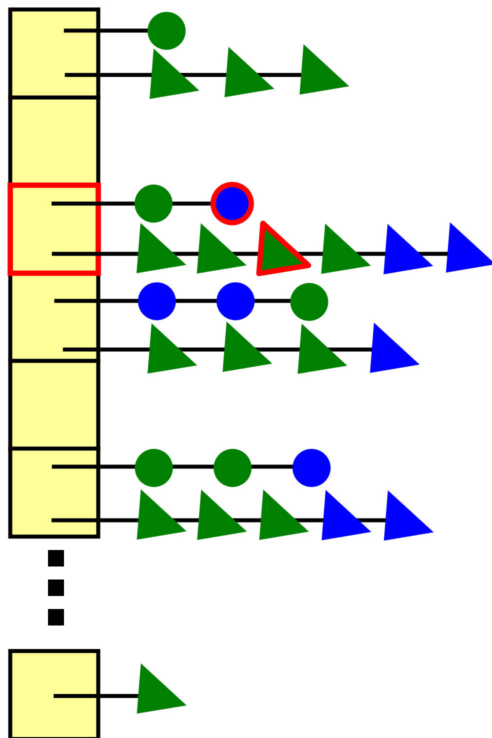
Hashtabelle t

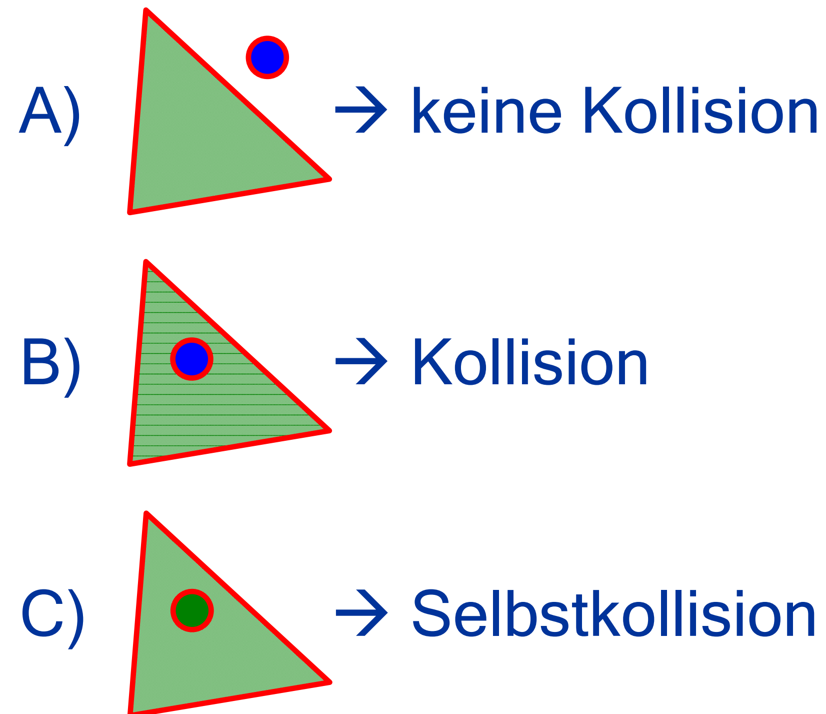
Implementierung – Schritt 2

- Begrenzungsbox für jedes Dreieck
- Dreieck wird allen Raumzellen zugeordnet, die von der Begrenzungsbox überlagert sind

Implementierung – Schritt 3

- Punkte und Dreiecke in der gleichen Hashtabellen-Position werden paarweise auf Kollision getestet





→ keine Kollision

→ Kollision

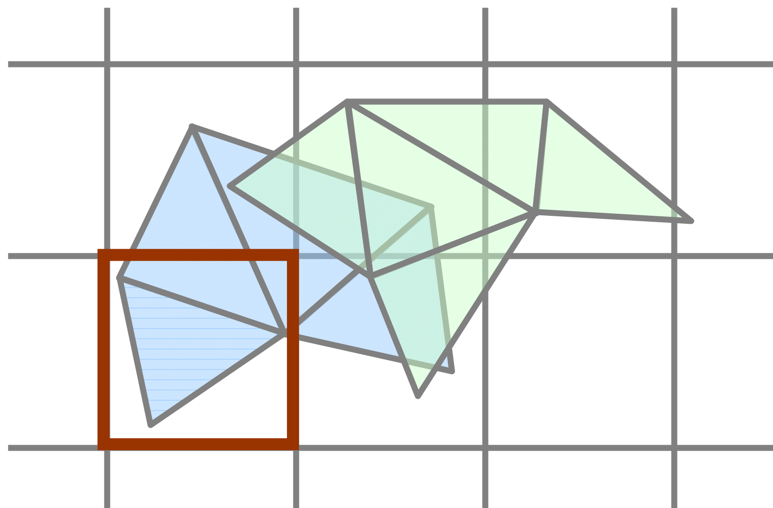
→ Selbstkollision

Implementierung

- trage Punkte in Hashtabelle ein
- berechne Hashwerte für Begrenzungbox eines Dreiecks
- speichere die Dreiecke nicht in der Hashtabelle, sondern teste, ob einer der dort gespeicherten Punkte im Dreieck liegt
- Parameter
  - Größe einer Gitterzelle
  - Größe der Hashtabelle
  - Hashfunktion

Parameter

unbegrenztes
räumliches Gitter



Zellgröße

Hashfunktion

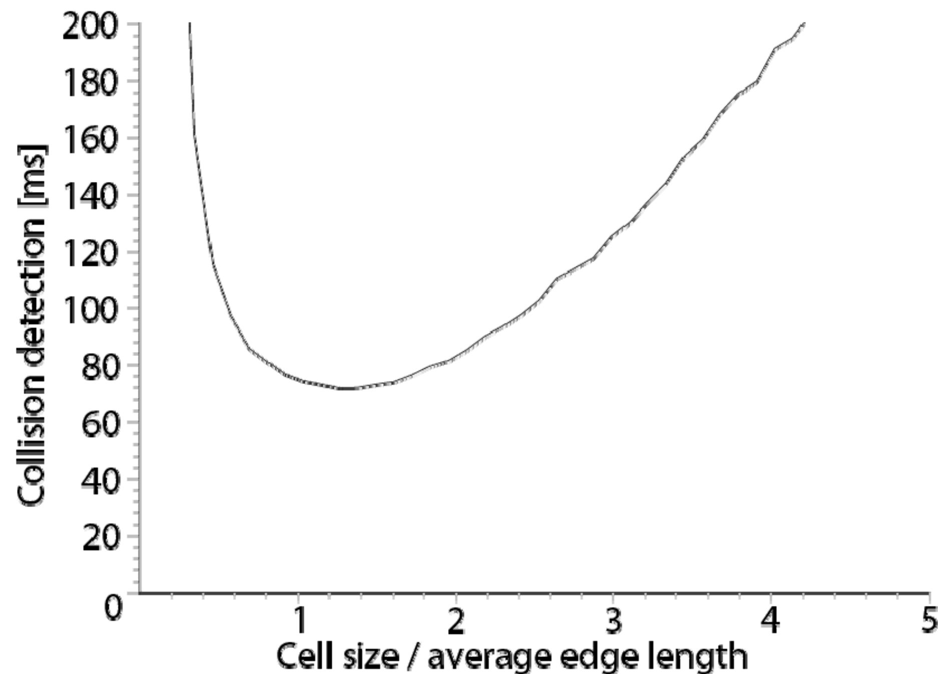
H(Raumzelle) → Hashwert

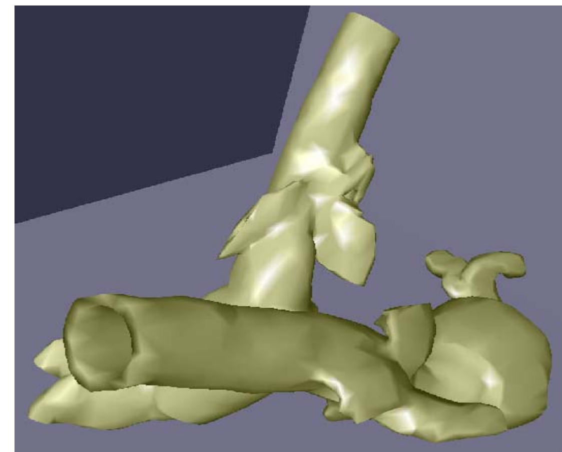
Hashtabelle

Größe der Hashtabelle

Effizienz
Größe einer Gitterzelle

- Größe einer Zelle sollte mit der Größe eines Dreiecks übereinstimmen [Bentley 1977]
- [Teschner, Heidelberger et al. 2003]

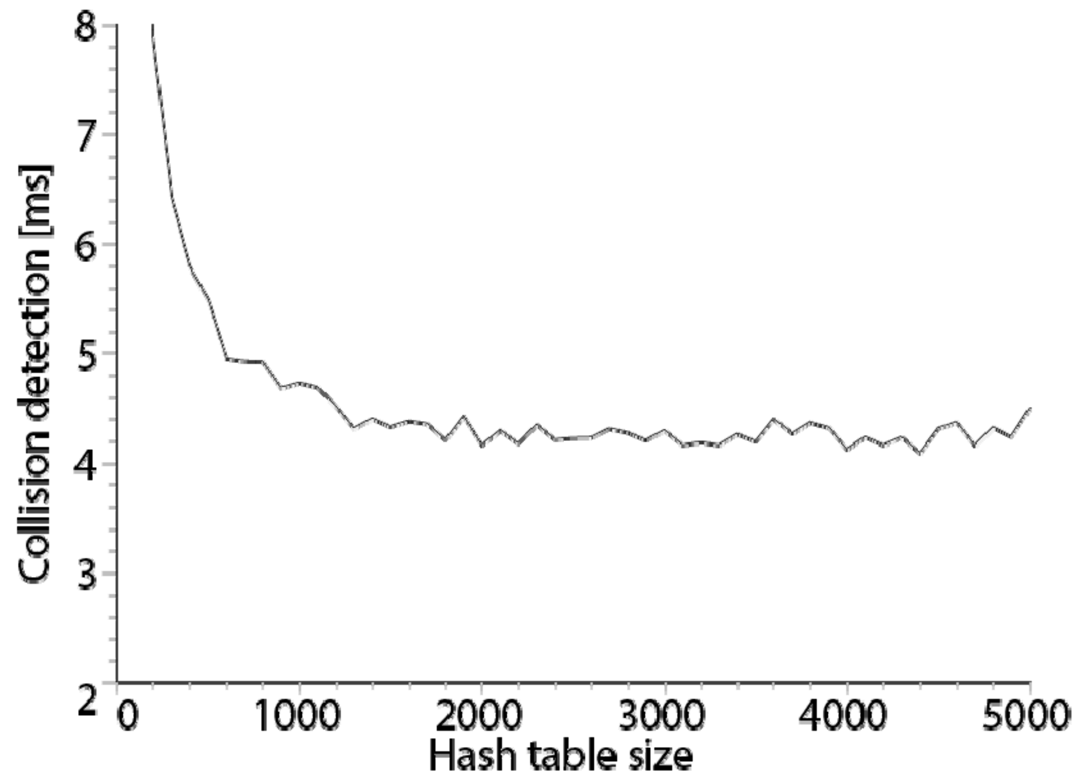


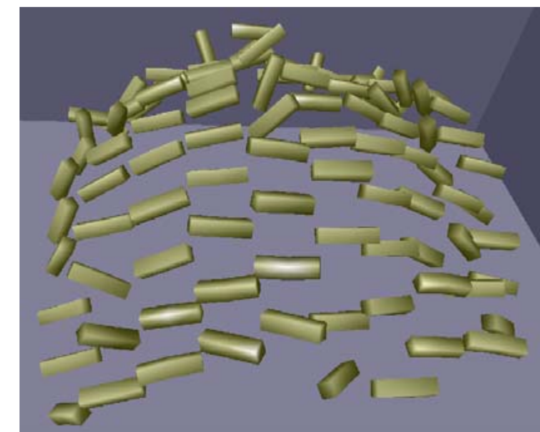


Testszene (3D, Tetraeder statt Dreiecke)

Effizienz
Größe der Hashtabelle

- Hashkollisionen verringern die Effizienz
- größere Hashtabellen verringern eventuell die Zahl der Hashkollisionen





Testszene (3D, Tetraeder statt Dreiecke)

Hashfunktion

- sollte Wahrscheinlichkeit von Hashkollisionen minimieren
- solle einfach zu berechnen sein
  (muss für alle Elemente berechnet werden)

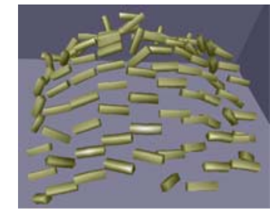
$$H(x, y, z) = (p_1 \cdot x \ \ \text{xor} \ \ p_2 \cdot y \ \ \text{xor} \ \ p_3 \cdot z) \ \ \text{mod} \ \ n$$

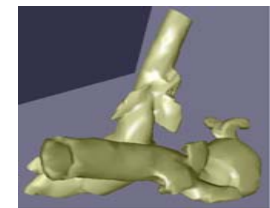
- Koordinaten einer Raumzelle (3D): x, y, z
- große Primzahlen: p_1, p_2, p_3
- Größe der Hashtabelle n

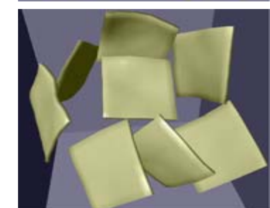
Laufzeit - Kollisionsdetektion

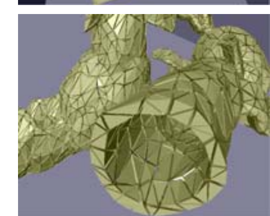
- Laufzeit linear in der Zahl der Dreiecke
- unabhängig von der Zahl der Objekte

Objekte	Drei-ecke	Punkte	Zeit [ms]
100	1000	1200	6
8	4000	1936	15
20	10000	4840	34
2	20514	5898	72
100	50000	24200	174









Testszenen

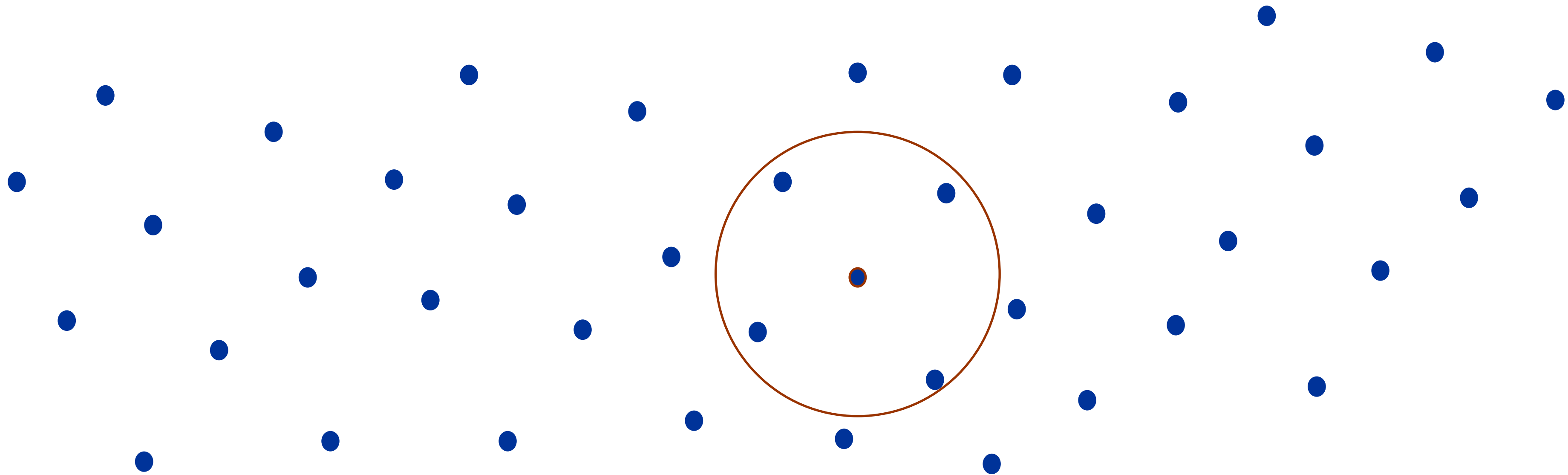
Pentium 4, 1.8GHz

Überblick

- Prinzip
- Details
- Anwendungen
    - Kollisionsdetektion
    - Nachbarschaft eines Punktes

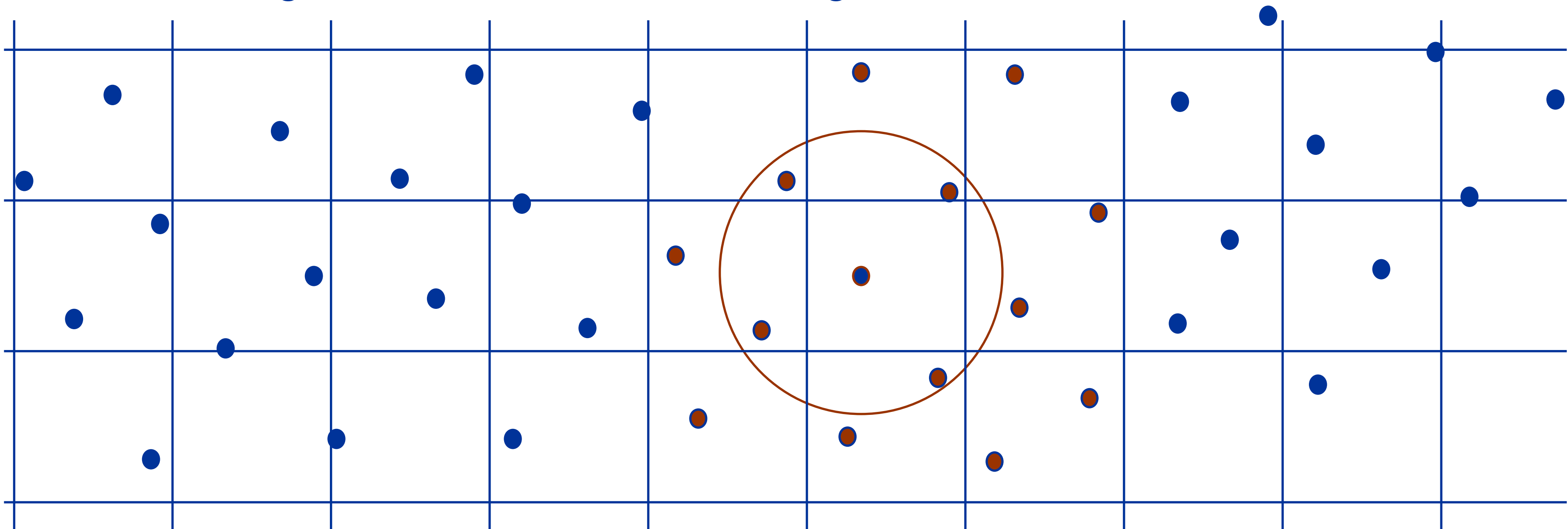
Nachbarschaft eines Punktes

- Problem
  - Finde alle Punkte in der Umgebung eines Punkte
  - z. B. zur Berechnung von Kräften in Fluid-Animationen



Nachbarschaft eines Punktes

- Lösungsansatz
  - Raumunterteilung
  - Partikel werden Raumzellen zugeordnet und entsprechend einer Hashfunktion in Hashtabelle gespeichert
  - Gesuchte Partikel können nur in der gleichen oder in angrenzenden Raumzellen liegen



Nächstes Thema

- Algorithmen / Datenstrukturen
  - Bäume