Effiziente Algorithmen

Master of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WiSe 2022/23

Inhalt

Einleitung

- Bewertung von Algorithmen
- Berechenbarkeitstheorie
- Komplexitätstheorie
- Exakte Algorithmen für schwere Probleme

Entwurfsmethoden

- divide and conquer
- dynamic programming
- greedy
- local search

Inhalt

Sortieren

- Quick-Sort – unterschiedliche Varianten
- Heap-Sort
- Untere Schranke
- Counting-/Radix-Sort
- Bucket-Sort

Auswahlproblem (Median-Berechnung)

- randomisiert
- worst-case

Inhalt

Graphalgorithmen

- Tiefen- und Breitensuche und deren Anwendungen
- Zusammenhangsprobleme
- Kürzeste Wege
- Minimaler Spannbaum
- Netzwerkfluss
- Matching

Spezielle Graphklassen

- Bäume und Co-Graphen
- Chordale Graphen
- Vergleichbarkeitsgraphen (comparability graph)
- Planare Graphen

Inhalt

Inhalt

Amortisierte Laufzeitanalysen

- dynamic tables
- Selbstanordnende lineare Listen
- Splay-Bäume
- Fibonacci-Heaps
- Union-Find-Datenstruktur

Algorithmen für moderne Hardware

- Lokalität der Daten
- Externe Datenstrukturen

Algorithmen für geometrische Probleme

- Scan-Line-Prinzip
- Konvexe Hülle

Literatur

- T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen.* Springer Spektrum
- T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms.* MIT Press
- Robert Sedgewick: *Algorithms.* Addison-Wesley
- Uwe Schöning: *Algorithmen - kurz gefasst.* Spektrum Akademischer Verlag
- Volker Heun: *Grundlegende Algorithmen.* Vieweg Verlag
- A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Datastructures and Algorithms.* Addison-Wesley
- Jon Kleinberg, Éva Tardos: *Algorithm Design.* Pearson-Addison Wesley
  `https://www.cs.princeton.edu/~wayne/kleinberg-tardos`

Literatur

- F. Gurski, I. Rothe, J. Rothe, E. Wanke: *Exakte Algorithmen für schwere Graphenprobleme.* Springer Verlag
- Shimon Even: *Graph Algorithms.* Computer Science Press
- Uwe Schöning: *Theoretische Informatik - kurz gefasst.* Spektrum Akademischer Verlag
- C.H. Papadimitriou: *Computational Complexity.* Addison-Wesley
- Juraj Hromkovič: *Randomisierte Algorithmen.* Vieweg + Teubner Verlag
- Rolf Wanka: *Approximationsalgorithmen.* Teubner B.G.
- I. Gerdes, F. Klawonn, R. Kruse: *Evolutionäre Algorithmen.* Vieweg Verlag

Übersicht

Übersicht

Amortisierte Laufzeitanalysen

- *dynamic tables*
- Selbstanordnende lineare Listen
- Fibonacci-Heaps
- Splay-Bäume
- Union-Find-Datenstruktur

Motivation

```
class Stack {
protected:
   int _next, _size, *_values;

   bool isFull() {
      return _size == _next;
   }

   void enlarge() {
      int *tmp = new int[2 * _size];

      for (int i = 0; i < _size; i++)
          tmp[i] = _values[i];

      delete[] _values;
      _values = tmp;
      _size *= 2;
   }
```

Motivation

```
public:
   Stack(int size = 8) {
      _size = size;
      _next = 0;
      _values = new int[size];
   }

   bool isEmpty() {
      return 0 == _next;
   }

   int top() {
      if (isEmpty())
         throw "empty stack exception";
      return _values[_next - 1];
   }
```

Motivation

```
//    void pop() {
//       if (isEmpty())
//          throw "empty stack exception";
//       _next -= 1;
//    }

   void push(int val) {
      if (isFull())     // !!!!!!!!
         enlarge();     // !!!!!!!!

      _values[_next] = val;
      _next += 1;
   }
};
```

Laufzeit von `push` im worst-case:

- Speicherbereich für $2n$ Elemente allokieren.
- Verschiebe n Elemente in größeren Speicherbereich.
- Kopiere neues Element in den Stack.

$\rightarrow$ Laufzeit: $\mathcal{O}(n)$

N Ausführungen von `push` dauern also:

$$\sum_{i=1}^{N} \mathcal{O}(i) \;=\; \mathcal{O}(N^2)$$

Aber: Ein `push` ist nur selten teuer!

Genauere Abschätzung für N Ausführungen von `push`:

$$\sum_{i=1}^{N} 1 + \sum_{i=1}^{\log_2(N)} 2^i \;\leq\; N + 2^{\log_2(N)+1} \;=\; 3 \cdot N \;\in\; \mathcal{O}(N)$$

Amortisierte (durchschnittliche) Laufzeit von `push`:

$$T(N) \;=\; \frac{\mathcal{O}(N)}{N} \;=\; \mathcal{O}(1)$$

amortisieren: ausgleichen, aufwiegen

Aggregat-Methode

auch *Ganzheitsmethode* genannt:

- Zeige für jedes n,
- dass jede beliebige Folge von Operationen
- im worst-case
- eine Laufzeit von $T(n)$ hat.

$\rightarrow$ amortisierte Kosten pro Operation: $\frac{T(n)}{n}$

Problem: Keine Differenzierung auf unterschiedliche Typen von Operationen wie `push`, `pop`, `top`, usw.

Account-Methode

auch *Bankkonto-Paradigma* genannt:

- Betrachte eine beliebige Folge von n Operationen.
- Jede Operation kostet Geld.
- t_i: Tatsächliche Kosten der i-ten Operation.
- a_i: Amortisierte Kosten der i-ten Operation.
- falls $t_i < a_i$, dann schreibe Überschuss auf Konto gut.
- falls $t_i > a_i$, dann zahle Differenz vom Konto.
- Kontostand darf niemals negativ sein!

Account-Methode

Pro `push` bezahle 3 elementare Einfüge-Operationen:

- einfügen des Elements
- verschieben des Elements beim Vergrößern
- verschieben eines anderen Elements beim Vergrößern

Der Kontostand wird niemals negativ! Nur bei der Vergrößerung des Speicherbereichs könnte der Kontostand negativ werden, aber:

- Zwischen den beiden Vergrößerungen auf 2^k und 2^{k+1} Elemente wurden genau 2^k Elemente eingefügt.
- Auf dem Konto ist also mindestens ein Guthaben von $2^k \cdot (3-1) = 2^{k+1}$.

$\rightarrow$ Entspricht genau der Anzahl Elemente, die verschoben werden müssen.

Potenzial-Methode

Anstelle von Kontoständen weise der Datenstruktur eine potenzielle Energie zu:

- Betrachte eine Folge von n Operationen.
- Ordne jedem Bearbeitungszustand ein nicht-negatives Potenzial zu.
- Ordne jeder Operation amortisierte Kosten zu.
- Φ_i: Potenzial nach Ausführung der i-ten Operation.
- Die amortisierten Kosten a_i der i-ten Operation sind die tatsächlichen Kosten t_i plus der Differenz der Potenziale.

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

Die gesamten amortisierten Kosten sind:

$$\begin{aligned}
\sum_{i=1}^{n} a_i &= a_1 + a_2 + a_3 + \ldots + a_n \\
&= t_1 + \Phi_1 - \Phi_0 \\
&+ t_2 + \Phi_2 - \Phi_1 \\
&+ t_3 + \Phi_3 - \Phi_2 \\
&+ \quad \ldots \\
&+ t_n + \Phi_n - \Phi_{n-1} \quad = \sum_{i=1}^{n} t_i + \Phi_n - \Phi_0
\end{aligned}$$

Falls $\Phi_n \geq \Phi_0$ ist, dann gilt:

$$\sum_{i=1}^{n} t_i = \sum_{i=1}^{n} a_i - (\Phi_n - \Phi_0) \leq \sum_{i=1}^{n} a_i$$

Also: Für $\Phi_n \geq \Phi_0$ ist die Summe der amortisierten Kosten eine obere Schranke für den tatsächlichen Aufwand.

Problem: Wir wissen nicht, wie viele Operationen ausgeführt werden.

Daher: Stelle $\Phi_i \geq \Phi_0$ für alle i sicher.

- In der Praxis: $\Phi_0 = 0$ und $\Phi_i \geq 0$, d.h. die Potenzial-Funktion wird niemals negativ, oder anders gesagt:
- Wir zahlen wie beim Bankkonto-Paradigma im vorraus.

Für den Stack S definiere die Potenzial-Funktion Φ als:

$$\Phi(S) := 2 \cdot num(S) - size(S)$$

Dabei ist

- *num* die Anzahl der gespeicherten Elemente und
- *size* die Größe des Arrays.

Der Stack ist immer mindestens halb voll, also gilt:

$$num(S) \geq \frac{size(S)}{2} \iff 2 \cdot num(S) \geq size(S)$$

$\rightarrow \Phi(S)$ wird niemals negativ!

Wir unterscheiden zwei Fälle beim Einfügen des i-ten Elements:

- *Keine Vergrößerung des Arrays* $\rightarrow size_i = size_{i-1}$

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i - 1) - size_i) \\ &= 3 \end{aligned}$$

- *Array-Vergrößerung* $\rightarrow size_i/2 = size_{i-1} = num_i - 1$

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 3 \cdot num_i - size_i - (2 \cdot (num_i - 1) - size_i/2)) \\ &= num_i + 2 - size_i/2 \\ &= num_i + 2 - (num_i - 1) \;=\; 3 \end{aligned}$$

Bei pop wollen wir ungenutzten Speicher wieder freigeben.

```
void Stack::reduce() {
    .....
}

void Stack::pop() {
    if (isEmpty())
        throw "empty stack exception";
    _next -= 1;
    if (isSlightlyFilled())   // ?????
        reduce();             // !!!!!
}
```

Übung 1.

- *Ab welchem Füllgrad soll die* reduce*-Funktion aufgerufen werden?*
- *Welche amortisierte Laufzeiten haben dann die Funktionen* push *und* pop*?*

Lösung ???

erste Idee:

- Speicherplatz verdoppeln, wenn das Array voll belegt ist und ein weiteres Element eingefügt werden soll, und
- Speicherplatz halbieren, wenn Array weniger als halb voll ist.

funktioniert leider nicht:

- Füge $n = 2^k$ Werte ein: $num(S) = size(S) = n = 2^k$
- Betrachte die Folge $i, d, d, i,\ \ i, d, d, i,\ \ i, d, d, i, \ldots$ aus insert- und delete-Operationen, bzw. push- und pop-Operationen.
  - i: $num(S) = n + 1 \quad \rightarrow$ vergrößert auf $size(S) = 2^{k+1}$
  - d: $num(S) = n$
  - d: $num(S) = n - 1 \quad \rightarrow$ verkleinert auf $size(S) = 2^k$
  - i: $num(S) = n$

$\rightarrow$ Bei jeder zweiten Operation ergibt sich ein Aufwand von $\Theta(n)$, so dass die amortisierte Zeit pro Operation $\Theta(n)$ beträgt.

zweite Idee:

- Speicherplatz verdoppeln, wenn das Array voll belegt ist, und
- halbieren, wenn das Array nur noch zu einem Viertel belegt ist.

Problem: Die bisherige Potenzialfunktion

$$\Phi(S) = 2 \cdot num(S) - size(S)$$

kann dann negativ werden, z.B. wenn der Stack nur zu einem Drittel gefüllt ist, also wenn $num(S) = \frac{1}{3} \cdot size(S)$.

neue Potenzialfunktion:

$$\Phi(S) := \begin{cases} 2 \cdot num(S) - size(S), & \text{falls } \alpha(S) \geq \frac{1}{2} \\ size(S)/2 - num(S), & \text{sonst} \end{cases}$$

wobei $\alpha(S) := num(S)/size(S)$ der *Füllgrad*, engl. *load factor*, ist.

$$\Phi(S) := \begin{cases} 2 \cdot num(S) - size(S), & \text{falls } \alpha(S) \geq \frac{1}{2} \\ size(S)/2 - num(S), & \text{sonst} \end{cases}$$

Intuitiv: Φ gibt an, wie weit der Stack vom Füllgrad $\alpha = 1/2$ entfernt ist. Wenn $\alpha = 1/2$ ist, dann gilt $2 \cdot num(S) = size(S)$.

- Wenn der Stack halb gefüllt ist, also für $\alpha = 1/2$, ist der Wert der Potenzialfunktion 0.
- Wenn der Stack komplett gefüllt ist, also für $\alpha = 1$, dann gilt $size(S) = num(S)$ und der Wert der Potenzialfunktion ist $num(S)$. Das Potenzial reicht also, um das Kopieren aller Elemente zu bezahlen.
- Wenn der Stack nur noch zu einem Viertel gefüllt ist, also wenn $\alpha = 1/4$ ist, dann gilt $size(S) = 4 \cdot num(S)$, und der Wert der Potenzialfunktion beträgt $num(S)$. Somit kann das Kopieren aller Elemente aus dem Potenzial bezahlt werden.

Für die *pop-Operation* gilt $num_i = num_{i-1} - 1$, und wir müssen drei Fälle unterscheiden. Betrachten wir zunächst $\alpha_{i-1} < \frac{1}{2}$.

- keine Verkleinerung: Dann gilt $size_i = size_{i-1}$ und

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2 \end{aligned}$$

- Verkleinerung: Dann gilt $size_i/2 = size_{i-1}/4 = num_i + 1$ und

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= (\cancel{num_i} + 1) + (size_i/2 - \cancel{num_i}) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (num_i + 1) - ((2 \cdot num_i + 2) - (num_i + 1)) \\ &= 1 \end{aligned}$$

Betrachten wir nun den Fall $\alpha_{i-1} \geq \frac{1}{2}$.

Dann wird das Array nicht verkleinert und es gilt $size_i = size_{i-1}$. Wir erhalten schließlich:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - \cancel{size_i}) - (2 \cdot (num_i + 1) - \cancel{size_i}) \\ &= 1 + 2 \cdot num_i - 2 \cdot (num_i + 1) \\ &= -1 \end{aligned}$$

Die amortisierten Kosten für eine pop-Operation sind also in allen Fällen durch eine Konstante nach oben beschränkt.

Für die *push-Operation* gilt $num_i = num_{i-1} + 1$.

Betrachten wir zunächst den Fall $\alpha_{i-1} \geq \frac{1}{2}$. Da für diesen Fall die alte und die neue Potenzialfunktion identisch sind, erhalten wir als Kosten für die push-Operation höchstens 3.

Für den Fall $\alpha_{i-1} < \frac{1}{2}$ kann keine Vergrößerung des Arrays notwendig werden, da eine Vergrößerung nur bei $\alpha_{i-1} = 1$ erfolgt. Also gilt $size_i = size_{i-1}$. Wir unterscheiden zwei Fälle.

- für $\alpha_i < \frac{1}{2}$ erhalten wir:

$$\begin{aligned} a_i &= t_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0 \end{aligned}$$

- für $\alpha_i \geq \frac{1}{2}$ erhalten wir:

$$\begin{aligned}
a_i &= t_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2 \cdot num_{i-1} + 2 - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 + 3 \cdot num_{i-1} - \frac{3}{2} \cdot size_{i-1} \\
&= 3 + 3 \cdot \alpha_{i-1} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} \\
&< 3 + \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} \\
&= 3
\end{aligned}$$

Auch für die push-Operation sind die amortisierten Kosten in allen Fällen durch eine Konstante nach oben beschränkt.

Account-Methode

Mittels Bankkonto-Methode ergeben sich folgende amortisierten Laufzeiten:

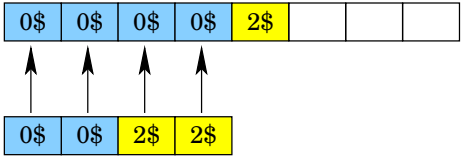
- push: $a_i = 3 \rightarrow 2\$$ sparen
- pop: $a_i = 2 \rightarrow 1\$$ sparen

Dann gilt für $size(S) = 2^{k+1}$ beim

- Vergrößern: Zwischen $num(S') = 2^k$ und $num(S) = 2^{k+1}$ gibt es 2^k push-Operationen mehr als pop-Operationen. Daher wurde ein Guthaben von mindestens $2 \cdot 2^k = 2^{k+1}$ angespart, wovon das Kopieren der 2^{k+1} Elemente in das nächst größere Array bezahlt werden kann.
- Verkleinern: Zwischen $num(S') = 2^{k-1}$ und $num(S) = 2^{k-2}$ gibt es 2^{k-2} pop-Operationen mehr als push-Operationen. Daher wurde ein Guthaben von mindestens 2^{k-2} angespart, wovon das Kopieren der 2^{k-2} Elemente in das kleinere Array bezahlt werden kann.

Account-Methode: push

Account-Methode: push

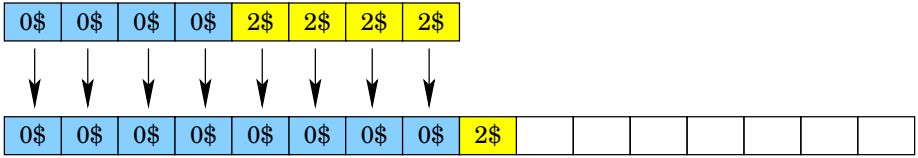


Account-Methode: push

Account-Methode: push

Account-Methode: push

Account-Methode: push



Account-Methode: pop

Account-Methode: pop

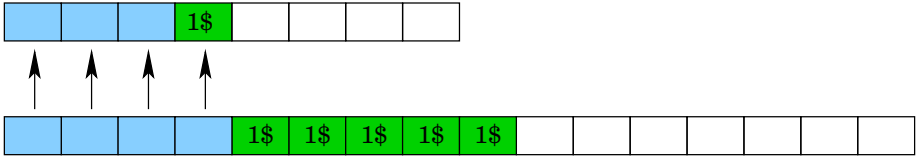
Account-Methode: pop

Account-Methode: pop

Account-Methode: pop

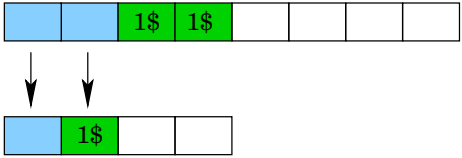
Account-Methode: pop

Account-Methode: pop



Account-Methode: pop

Account-Methode: pop



Übersicht

Amortisierte Laufzeitanalysen

- dynamic tables
- *Selbstanordnende lineare Listen*
- Fibonacci-Heaps
- Splay-Bäume
- Union-Find-Datenstruktur

Motivation: Bei Datenbanken erfolgen 80% der Zugriffe auf nur 20% der Daten.

$\rightarrow$ Wie kann man den Zugriff effizient gestalten?

Aufgabenstellung:

- Gegeben: Linear verkettete Liste (unsortiert)
- Gesucht: Effizientes Einfügen, Löschen und Suchen.

Idee: Plaziere die Elemente, auf die häufig zugegriffen wird, weit vorne in der Liste.

Problem: In der Regel sind die Zugriffshäufigkeiten nicht im voraus bekannt.

Lösung: Ändere nach jedem Zugriff auf ein Element die Liste so, dass zukünftige Anfragen nach diesem Element schneller sind.

Die letzte Suchanfrage habe auf das Element e zugegriffen.

- *Move-To-Front-Regel* (MF-Regel)
  Mache e zum ersten Element der Folge.
- *Transpose-Regel* (T-Regel)
  Vertausche e mit unmittelbar vorangehendem Element.

Die relative Anordnung der anderen Elemente bleibt bei den obigen Regeln unverändert.

- *Frequency-Count-Regel* (FC-Regel)
  - Ordne jedem Element einen Häufigkeitszähler zu und
  - ordne die Liste nach jedem Zugriff entsprechend neu.

Beispiel: Betrachte Zugriffe auf die Zahlenfolge mittels *MF*-Regel:

$$1, 2, 3, 4, 5, 6, 7$$

- Greife zehnmal hintereinander auf alle Zahlen $1 \ldots 7$ zu:

  $$1, 2, 3, 4, 5, 6, 7,\ 1, 2, 3, 4, 5, 6, 7,\ \ldots,\ 1, 2, 3, 4, 5, 6, 7$$

  $\rightarrow$ Kosten pro Zugriff: 6.7
- Greife jeweils zehnmal auf jedes einzelne Element zu:

  $$1, 1, 1, 1, 1, 1, 1, 1, 1, 1,\ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,\ \ldots$$

  $\rightarrow$ Kosten pro Zugriff: 1.3
- Ohne Selbstanordnung ergeben sich bei obigen Zugriffsfolgen jeweils die Kosten 4.

Gibt es eine optimale Strategie?

- Das Vorziehen eines Elements an den Listenanfang entsprechend der *MF*-Regel ist eine sehr drastische Änderung, die erst allmählich korrigiert werden kann, falls ein „seltenes“ Element „irrtümlich“ an den Listenanfang gesetzt wurde.
- Die *T*-Regel ist vorsichtiger und macht geringere Fehler, aber man kann leicht Zugriffsfolgen finden, so dass Zugriffe nach der *T*-Regel praktisch überhaupt nichts bringen.
- Die *FC*-Regel hat den Nachteil, dass man zusätzlichen Speicherplatz zur Aufnahme der Häufigkeitszähler bereitstellen muss.
- In der Literatur findet man weitere Permutationsregeln: *J.H. Hester und D.S. Hirschberg. Self-organizing linear search. ACM Computing Surveys, 17:295-311, 1985.*

- Wir müssen nicht nur unterschiedliche Zugriffshäufigkeiten berücksichtigen, sondern auch Clusterungen von Zugriffsfolgen, die sogenannte Lokalität.
- In der Literatur findet man meist nur asymptotische Aussagen über das erwartete Verhalten der Strategien, wobei die Zugriffsfolgen bestimmten Wahrscheinlichkeitsverteilungen genügen.
- Es gibt auch eine Reihe experimentell ermittelter Messergebnisse für reale Daten, bei denen man die verschiedenen Strategien relativ zueinander beurteilt.
- Slator und Tarjan gelang ein bemerkenswertes theoretisches Ergebnis, das das sehr gute, beobachtete Verhalten der *MF*-Regel untermauert.

D.D. Sleator and R.E. Tarjan: Amortized efficiency of list update and paging rules. Communications of the ACM, 28:202-208, 1985.

Selbstanordnende lineare Listen

Gegeben: Eine Liste mit N Elementen sowie eine Folge $s = s_1, \ldots, s_m$ von m Zugriffsoperationen.

Sei A ein beliebiger Algorithmus zur Selbstanordnung der Elemente, der

- das Element zuerst sucht und
- es dann durch Vertauschungen einige Positionen zum Anfang oder zum Ende der Liste bewegt.

Kosten:

- Zugriff auf Element an Position k kostet k Einheiten.
- Vertauschung in Richtung Listenanfang ist kostenfrei.
- Vertauschung in andere Richtung kostet eine Einheit.

Zu einem Algorithmus A und einer Zugriffsoperation s_i definieren wir:

$C_A(s_i)$ Schritte zur Ausführung der Zugriffsoperation s_i.
$F_A(s_i)$ Anzahl kostenfreier Vertauschungen.
$X_A(s_i)$ Anzahl kostenbehafteter Vertauschungen.

$C_A(s) = C_A(s_1) + \ldots + C_A(s_m)$, $F_A(s)$ und $X_A(s)$ analog.

Unsere bisher betrachteten Algorithmen

- MF (Move-to-Front),
- T (Transpose) und
- FC (Frequently-Count)

führen keine kostenbehafteten Vertauschungen durch. Somit gilt:

$$X_{MF}(s) = X_T(s) = X_{FC}(s) = 0$$

Wird auf ein Element an Position k zugegriffen, kann man es anschließend maximal mit allen $(k-1)$ vorangehenden Elementen kostenfrei vertauschen.

$\rightarrow$ Die Anzahl kostenfreier Vertauschungen ist höchstens so groß wie die Kosten der Operation minus 1.

Daher gilt für jede Strategie A (mit m Operationen):

$$F_A(s) \leq C_A(s) - m$$

denn

$$\begin{aligned} F_A(s) &= F_A(s_1) + F_A(s_2) + \ldots + F_A(s_m) \\ &\leq C_A(s_1) - 1 + C_A(s_2) - 1 + \ldots + C_A(s_m) - 1 \\ &= C_A(s) - m \end{aligned}$$

Definition: Seien L_1, L_2 Listen, die dieselben Elemente enthalten.

> $inv(L_1, L_2)$: Anzahl der Elementpaare x_i, x_j, deren Anordnung in L_2 eine andere ist als in L_1 → *Inversionen*

Beispiel: Für $L_1 = 4, 3, 5, 1, 7, 2, 6$ und $L_2 = 3, 6, 2, 5, 1, 4, 7$ erhalten wir $inv(L_1, L_2) = 12$, denn in L_2 steht

> 3 vor 4, 6 vor 2, 6 vor 5, 6 vor 1, 6 vor 4, 6 vor 7,
> 2 vor 5, 2 vor 1, 2 vor 4, 2 vor 7, 5 vor 4, 1 vor 4.

Diese Elementpaare stehen jedoch in L_1 in umgekehrter Reihenfolge.

Satz: Für jeden Algorithmus A der obigen Art und für jede Folge s von m Zugriffsoperationen gilt:

$$C_{MF}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$

Dieser Satz besagt grob, dass die MF-Regel höchstens doppelt so schlecht ist, wie jede andere Regel zur Selbstanordnung von Listen.

Die MF-Regel ist also nicht wesentlich schlechter als die beste überhaupt denkbare Strategie.

Beweis:

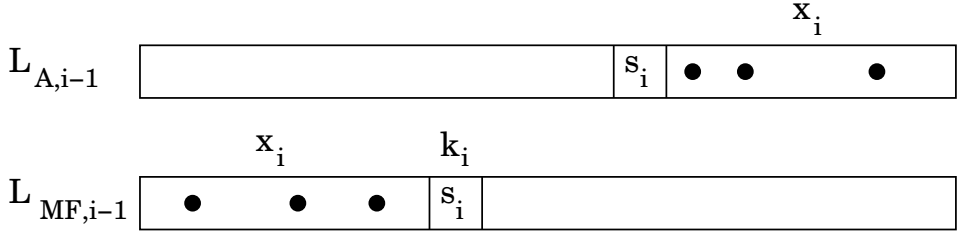
- Sei $L_{A,i}$ die von Algorithmus A geänderte Liste nach Zugriff i.
- Sei $L_{MF,i}$ die von der MF-Regel geänderte Liste nach Zugriff i.

Für die *Potenzialfunktion* $\Phi_i = inv(L_{A,i}, L_{MF,i})$ gilt:

- Initial ist $\Phi_0 = inv(L_{A,0}, L_{MF,0}) = 0$, da beide Algorithmen mit derselben Liste beginnen.
- Der Wert der Potenzialfunktion ist nie negativ!

Betrachte Zugriff auf das Element s_i:

- k_i: Position, an der Element s_i in $L_{MF,i-1}$ steht.
- x_i: Anzahl Elemente, die in $L_{MF,i-1}$ vor s_i aber in $L_{A,i-1}$ hinter s_i stehen.



Jedes dieser x_i Elemente ist mit s_i eine Inversion.

Es gilt:

$$inv(L_{A,i-1}, L_{MF,i}) \;=\; inv(L_{A,i-1}, L_{MF,i-1}) - x_i + (k_i - 1 - x_i)$$

denn: Durch das Vorziehen von s_i in $L_{MF,i-1}$

- verschwinden insgesamt x_i Inversionen und
- es entstehen $k_i - (1 + x_i)$ Inversionen.
  $k_i - (1 + x_i)$ ist die Anzahl der übrigen Elemente in $L_{MF,i-1}$ vor dem Element s_i.

Es gilt also:

$$inv(L_{A,i-1}, L_{MF,i}) \;=\; inv(L_{A,i-1}, L_{MF,i-1}) - x_i + (k_i - 1 - x_i)$$

Anschließend:

- Jedes Vorziehen von s_i in $L_{A,i-1}$ mit Algorithmus A erniedrigt die Anzahl der Inversionen.
- Jedes Vertauschen von s_i in $L_{A,i-1}$ zum Listenende mit Algorithmus A erhöht die Anzahl der Inversionen.

Somit erhalten wir:

$$inv(L_{A,i}, L_{MF,i}) \;=\; inv(L_{A,i-1}, L_{MF,i-1}) - x_i + (k_i - 1 - x_i) - F_A(s_i) + X_A(s_i)$$

- Tatsächliche Kosten für Zugriff auf s_i mit MF-Regel: k_i

$\rightarrow$ Amortisierte Kosten a_i der i-ten Operation:

$$\begin{aligned} a_i &= k_i + inv(L_{A,i}, L_{MF,i}) - inv(L_{A,i-1}, L_{MF,i-1}) \\ &= k_i - x_i + (k_i - 1 - x_i) - F_A(s_i) + X_A(s_i) \\ &= 2(k_i - x_i) - 1 - F_A(s_i) + X_A(s_i) \end{aligned}$$

Tatsächliche Kosten für Zugriff auf s_i in $L_{A,i-1}$ mit Algorithmus A: Mindestens $k_i - x_i$, denn $k_i - x_i - 1$ Elemente stehen in beiden Listen vor dem Element s_i, also auch in der Liste $L_{A,i-1}$ vor s_i.

Damit erhalten wir:

$$\sum_{i=1}^{m} a_i \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$

Übersicht

Amortisierte Laufzeitanalysen

- dynamic tables
- Selbstanordnende lineare Listen
- *Fibonacci-Heaps*
- Splay-Bäume
- Union-Find-Datenstruktur

Wir benötigen einige Vorarbeiten, um die Laufzeiten der einzelnen Operationen eines Fibonacci-Heaps abschätzen zu können.

Lemma 1: Sei x ein Knoten im Fibonacci-Heap, seien $y_1, \ldots, y_k$ die Kinder von x in der zeitlichen Reihenfolge, in der sie an x angehängt wurden. Dann gilt:

$$deg(y_1) \geq 0 \quad \text{und} \quad deg(y_i) \geq i - 2 \text{ für } i = 2, 3, \ldots, k$$

Induktionsanfang: $deg(y_1) \geq 0$ ist klar.

Induktionsschluss für $i \geq 2$:

- Als y_i an x angehängt wurde, waren $y_1, \ldots, y_{i-1}$ bereits Kinder von x, also gilt $deg(x) \geq i - 1$.
- Da y_i an x angehängt wird, gilt: $deg(x) = deg(y_i)$, also gilt $deg(y_i) \geq i - 1$.
- Seither hat Knoten y_i höchstens ein Kind verloren, sonst hätten wir y_i von x abgetrennt, also gilt: $deg(y_i) \geq i - 2$

Die Fibonacci-Zahlen F_i sind wie folgt rekursiv definiert:

$$F_0 = 0, \quad F_1 = 1, \quad F_i = F_{i-1} + F_{i-2} \text{ für } i \geq 2$$

Lemma 2:

$$F_{k+2} \;=\; 1 + \sum_{i=0}^{k} F_i$$

Beweis durch Induktion über k:
Induktionsanfang: $k = 0$

$$F_2 = 1 \;\stackrel{!}{=}\; 1 + \sum_{i=0}^{0} F_i = 1 + 0 = 1$$

Induktionsschluss: $k - 1 \to k$

$$F_{k+2} \;\stackrel{\text{Def.}}{=}\; F_k + F_{k+1} \;\stackrel{\text{I.V.}}{=}\; F_k + (1 + \sum_{i=0}^{k-1} F_i) \;=\; 1 + \sum_{i=0}^{k} F_i$$

oder anschaulich:

$$
\begin{aligned}
F_{k+2} &= F_k + F_{k+1} \\
&= F_k + F_{k-1} + F_k \\
&= F_k + F_{k-1} + F_{k-2} + F_{k-1} \\
&= F_k + F_{k-1} + F_{k-2} + F_{k-3} + F_{k-2} \\
&\vdots \\
&= F_k + F_{k-1} + F_{k-2} + F_{k-3} + \ldots + F_1 + F_2 \\
&= \underbrace{F_k + F_{k-1} + F_{k-2} + F_{k-3} + \ldots + F_1 + F_0}_{=\sum_{i=0}^{k} F_i} + \underbrace{F_1}_{=1} \\
&= \sum_{i=0}^{k} F_i + 1
\end{aligned}
$$

Lemma 3: Sei x ein Knoten in einem Fibonacci-Heap, sei $k = deg(x)$. Dann gilt:

$$size(x) \geq F_{k+2}$$

Beweis: Bezeichne s_k den minimalen Wert von $size(z)$ über alle Knoten z mit $deg(z) = k$. Dann gilt offensichtlich:

- $s_0 = 1, \; s_1 = 2, \; s_2 = 3$
- $s_k \leq size(x)$

Da x den Grad k hat, gilt für jedes Kind y_i, $i = 2, \ldots, k$ von x nach Lemma 1: $deg(y_i) \geq i - 2$. Außerdem müssen wir bei $size(x)$ den Knoten x und das Kind y_1 berücksichtigen. Daher erhalten wir:

$$size(x) \geq s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} = 2 + \sum_{i=0}^{k-2} s_i$$

Ferner können wir feststellen

$$\begin{aligned} s_0 = 1 &\geq F_2 = 1 \\ s_1 = 2 &\geq F_3 = 2 \\ s_2 = 3 &\geq F_4 = 3 \end{aligned}$$

und mittels vollständiger Induktion und Lemma 2 zeigen:

$$s_k \;\geq\; 2 + \sum_{i=0}^{k-2} s_i \;\geq\; 2 + \sum_{i=0}^{k-2} F_{i+2} \;=\; 1 + \sum_{i=0}^{k} F_i \;=\; F_{k+2}$$

Insgesamt erhalten wir also:

$$size(x) \;\geq\; s_k \;\geq\; F_{k+2}$$

Aus der Mathematik wissen wir:

$$F_k = \frac{1}{\sqrt{5}} \cdot \phi^k \quad \text{mit} \quad \phi = \frac{(1+\sqrt{5})}{2}$$

ϕ ist der goldene Schnitt mit $\phi \approx 1.61803$.

$\rightarrow$ *Die Fibonacci-Zahlen wachsen exponentiell!*

Korollar: Jeder Knoten x eines Fibonacci-Heaps der Größe n hat höchstens Grad $\mathcal{O}(\log(n))$.

Beweis: Sei $k = deg(x)$, dann gilt nach Lemma 3:

$$n \geq size(x) \geq \phi^k \iff \log_\phi(n) \geq k$$

Potenzial-Funktion: Definiere zu einem Fibonacci-Heap H

$$\Phi(H) \ := \ b(H) + 2 \cdot m(H),$$

wobei

- $b(H)$ die Anzahl der Bäume in der Wurzelliste und
- $m(H)$ die Anzahl der markierten Knoten bezeichnet.

Zur Erinnerung:
Die amortisierten Kosten a_i sind die tatsächlichen Kosten t_i plus der Differenz der Potenziale:

$$a_i = t_i + \Phi_i - \Phi_{i-1}$$

MakeHeap() $\rightarrow b(H) = 0,\ m(H) = 0$
amortisierte Kosten = aktuelle Kosten $\in \mathcal{O}(1)$

Minimum(H): $\rightarrow b(H)$ und $m(H)$ unverändert
amortisierte Kosten = aktuelle Kosten $\in \mathcal{O}(1)$

Union(H_1, H_2): $\rightarrow \Phi(H) = \Phi(H_1) + \Phi(H_2)$
amortisierte Kosten = aktuelle Kosten $\in \mathcal{O}(1)$

Insert(H, x): $\rightarrow b(H') = b(H) + 1,\ m(H)$ unverändert
amortisierte Kosten = aktuelle Kosten $+ 1 \in \mathcal{O}(1)$

ExtractMin(H):

- Der Knoten, der entfernt wird, hat maximal $\mathcal{O}(\log(n))$ Kinder, die in die Wurzelliste aufgenommen werden.
- Die Wurzelliste wird einmal durchlaufen, um ggf. Bäume gleichen Ranges zu verschmelzen.

$\rightarrow$ aktuelle Kosten: $t_i = b(H) + \mathcal{O}(\log(n))$

- Nach dem Verschmelzen können nur maximal $\mathcal{O}(\log(n))$ viele Bäume in der Wurzelliste stehen, da dann alle Bäume unterschiedlichen Rang haben.
- Es werden keine Knoten markiert.

$\rightarrow$ $\Phi_{i-1} = b(H) + 2m(H)$ und $\Phi_i \leq \mathcal{O}(\log(n)) + 2m(H)$

$\Rightarrow$ amortisierte Kosten: $t_i + \Phi_i - \Phi_{i-1} \in \mathcal{O}(\log(n))$

DecreaseKey(H, x, k): Bei insgesamt c Abtrennungen

- entsteht ein tatsächlicher Aufwand von c Einheiten,
- werden mindestens $c - 1$ Markierungen gelöscht und
- c viele Bäume werden in die Wurzelliste aufgenommen.
- Mit $\Phi_{i-1} = b(H) + 2m(H)$ gilt also:

  $\Phi_i = b(H) + c \ + \ 2(m(H) - (c - 1))$

$\Rightarrow$ amortisierte Kosten: $t_i + \Phi_i - \Phi_{i-1} \in \mathcal{O}(1)$

Delete(H, x):

- Laufzeit ergibt sich aus DecreaseKey und ExtractMin

$\Rightarrow$ amortisierte Kosten $\in \mathcal{O}(\log(n))$

Übersicht

Amortisierte Laufzeitanalysen

- dynamic tables
- Selbstanordnende lineare Listen
- Fibonacci-Heaps
- *Splay-Bäume*
- Union-Find-Datenstruktur

Bei Splay-Bäumen werden alle drei Operationen Suchen, Einfügen und Entfernen auf die Splay-Operation zurückgeführt.

Die Kosten einer Splay-Operation sei die Anzahl der ausgeführten Rotationen (plus 1, falls keine Operation ausgeführt wird).

- Jede zig-Operation zählt eine Rotation.
- Jede zig-zig- und zig-zag-Operation zählt zwei Rotationen.

Für einen Knoten p, sei $s(p)$ die Anzahl aller inneren Knoten (Schlüssel) im Teilbaum mit Wurzel p.

Sei $r(p)$ der Rang von p, definiert durch $r(p) = \log(s(p))$.

Für einen Baum T mit Wurzel p und für einen in p gespeicherten Schlüssel x sei $r(T)$ und $r(x)$ definiert als $r(p)$.

Die Potenzialfunktion $\Phi(T)$ für einen Suchbaum T sei definiert als die Summe aller Ränge der inneren Knoten von T.

Lemma: Der amortisierte Aufwand der Operation $splay(T, x)$ ist höchstens $3 \cdot (r(T) - r(x)) + 1$.

Beweis: Ist x in der Wurzel von T gespeichert, so wird nur auf x zugegriffen und keine weitere Operation ausgeführt. Der tatsächliche Aufwand 1 stimmt mit dem amortisierten Aufwand überein und das Lemma gilt.

Angenommen es wird wenigstens eine Rotation durchgeführt.

Für jede bei der Ausführung von $splay(T, x)$ durchgeführte Rotation, die einen Knoten p betrifft, betrachten wir

- die Größe $s_v(p)$ und den Rang $r_v(p)$ von p unmittelbar vor Ausführung der Rotation und
- die Größe $s_n(p)$ und den Rang $r_n(p)$ von p unmittelbar nach Ausführung der Rotation.

Wir werden zeigen, dass

- jede zig-zig- und zig-zag-Operation auf p mit amortisiertem Aufwand $3(r_n(p) - r_v(p))$ und
- jede zig-Operation mit amortisiertem Aufwand $3(r_n(p) - r_v(p)) + 1$ ausführbar ist.

Angenommen obige Aussage wäre bereits gezeigt.

Sei $r^{(i)}(x)$ der Rang von x nach Ausführung der i-ten von insgesamt k zig-zig-, zig-zag- oder zig-Operationen.

Beachte: Nur die letzte Operation kann eine zig-Operation sein.

Dann ergibt sich als amortisierter Aufwand von $splay(T, x)$:

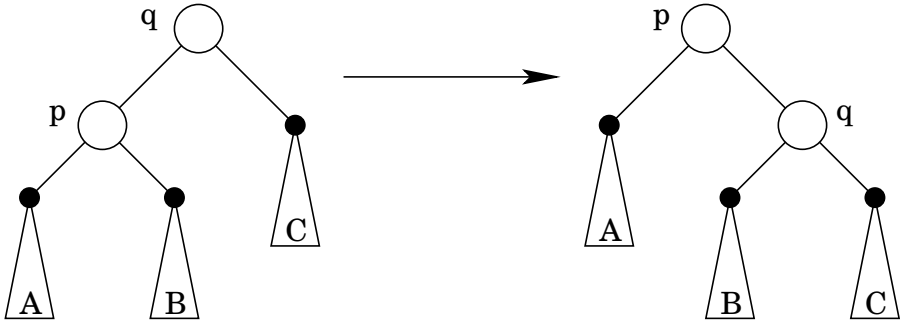
$$\begin{aligned} splay(T,x) &= 3(r^{(1)}(x) - r(x)) \\ &+ 3(r^{(2)}(x) - r^{(1)}(x)) \\ &\vdots \\ &+ 3(r^{(k)}(x) - r^{(k-1)}(x)) + 1 \\ &= 3(r^{(k)}(x) - r(x)) + 1 \end{aligned}$$

Da x durch die k Operationen zur Wurzel gewandert ist, gilt $r^{(k)}(x) = r(T)$ und damit das Lemma.

Noch zu zeigen:

- Jede zig-zig- und zig-zag-Operation auf p ist mit amortisiertem Aufwand $3(r_n(p) - r_v(p))$ und
- jede zig-Operation mit amortisiertem Aufwand $3(r_n(p) - r_v(p)) + 1$ ausführbar.

Fall 1: (zig-Operation)



Dann ist q die Wurzel. Es wird eine Rotation ausgeführt. Die tatsächlichen Kosten sind 1. Es können höchstens die Ränge von p und q geändert worden sein.

Die amortisierten Kosten der zig-Operation sind daher:

$$\begin{aligned}
a_{zig} &= 1 + \Phi_n(T) - \Phi_v(T) \\
&= 1 + (r_n(p) + r_n(q) + \ldots) - (r_v(p) + r_v(q) + \ldots) \\
&= 1 + (r_n(p) + r_n(q)) - (r_v(p) + r_v(q)) \\
&= 1 + r_n(q) - r_v(p), \quad \text{weil } r_n(p) = r_v(q) \\
&\leq 1 + r_n(p) - r_v(p), \quad \text{weil } r_n(p) \geq r_n(q) \\
&\leq 1 + 3(r_n(p) - r_v(p)), \quad \text{weil } r_n(p) \geq r_v(p)
\end{aligned}$$

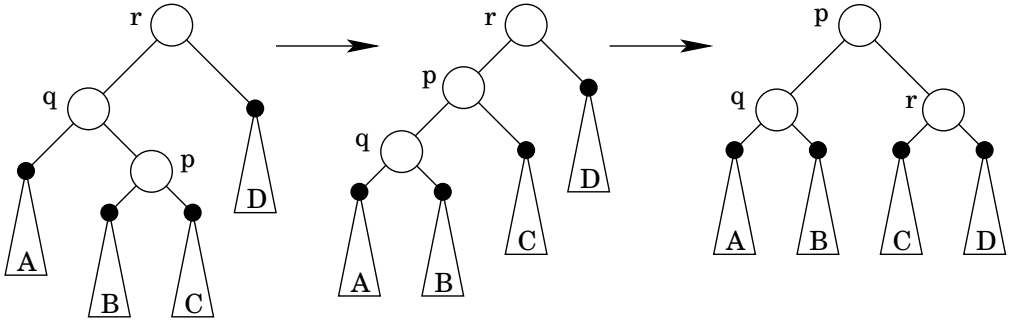
Für die nächsten beiden Fälle, formulieren wir folgende Hilfsaussage:

> Sind a und b positive Zahlen und gilt $a + b \leq c$, so folgt $\log(a) + \log(b) \leq 2\log(c) - 2$.

Da das geometrische Mittel zweier positiver Zahlen niemals größer ist als das arithmetische Mittel, gilt:

$$\begin{aligned}
\sqrt{ab} &\leq (a+b)/2 \\
\sqrt{ab} &\leq c/2 \\
ab &\leq (c/2)(c/2) \\
\log(ab) &\leq \log((c/2)(c/2)) \\
\log(a) + \log(b) &\leq 2\log(c/2) \\
\log(a) + \log(b) &\leq 2\log(c) - 2\log(2) \\
\log(a) + \log(b) &\leq 2\log(c) - 2
\end{aligned}$$

Fall 2: (zig-zag-Operation)



Die Operation hat tatsächliche Kosten von 2, weil zwei Rotationen durchgeführt werden.

Es können sich höchstens die Ränge von p, q und r ändern, ferner ist $r_n(p) = r_v(r)$.

Also gilt für die amortisierten Kosten:

$$\begin{aligned} a_{zig-zag} &= 2 + \Phi_n(T) - \Phi_v(T) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r) + \ldots) \\ &\quad - (r_v(p) + r_v(q) + r_v(r) + \ldots) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r)) - (r_v(p) + r_v(q) + r_v(r)) \\ &= 2 + r_n(q) + r_n(r) - r_v(p) - r_v(q), \quad \text{weil } r_n(p) = r_v(r) \end{aligned}$$

Weil p vor Ausführung der zig-zag-Operation Kind von q war, gilt $r_v(q) \geq r_v(p)$ und somit

$$a_{zig-zag} \leq 2 + r_n(q) + r_n(r) - 2 \cdot r_v(p)$$

Wir hatten zuletzt festgestellt:

$$a_{zig-zag} \leq 2 + r_n(q) + r_n(r) - 2 \cdot r_v(p)$$

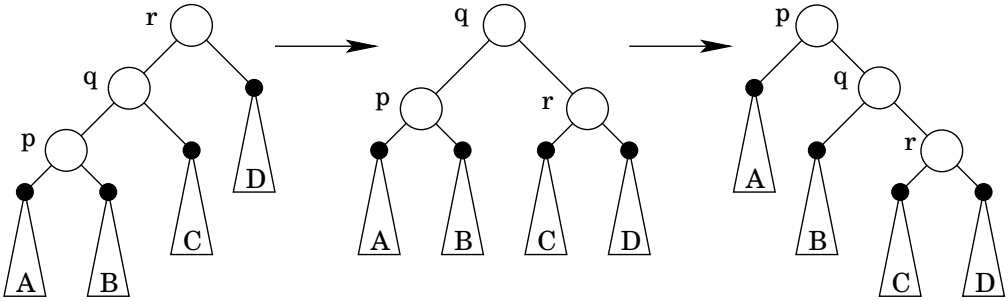
Ferner gilt $s_n(q) + s_n(r) \leq s_n(p)$ und damit aufgrund der Definition des Ranges und obiger Hilfsaussage:

$$\begin{aligned} &\log(s_n(q)) + \log(s_n(r)) \leq 2 \cdot \log(s_n(p)) - 2 \\ \iff \quad &r_n(q) + r_n(r) \leq 2 \cdot r_n(p) - 2 \end{aligned}$$

Damit erhalten wir schließlich:

$$\begin{aligned} a_{zig-zag} &\leq 2 + r_n(q) + r_n(r) - 2 \cdot r_v(p) \\ &\leq 2 + (2 \cdot r_n(p) - 2) - 2 \cdot r_v(p) = 2(r_n(p) - r_v(p)) \\ &\leq 3(r_n(p) - r_v(p)), \quad \text{weil } r_n(p) \geq r_v(p) \end{aligned}$$

Fall 3: (zig-zig-Operation)



Die Operation hat tatsächliche Kosten von 2, weil zwei Rotationen durchgeführt werden.

Auch hier können sich höchstens die Ränge von p, q und r ändern, und es gilt wie im vorherigen Fall $r_n(p) = r_v(r)$.

Für die amortisierten Kosten gilt:

$$\begin{aligned} a_{zig-zig} &= 2 + \Phi_n(T) - \Phi_v(T) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r) + \ldots) \\ &\quad - (r_v(p) + r_v(q) + r_v(r) + \ldots) \\ &= 2 + (r_n(p) + r_n(q) + r_n(r)) - (r_v(p) + r_v(q) + r_v(r)) \\ &= 2 + r_n(q) + r_n(r) - r_v(p) - r_v(q), \quad \text{weil } r_n(p) = r_v(r) \end{aligned}$$

Da vor Ausführung der zig-zig-Operation p Kind von q und nachher q Kind von p ist, folgt $r_v(p) \leq r_v(q)$ und $r_n(p) \geq r_n(q)$, und somit

$$a_{zig-zig} \leq 2 + r_n(p) + r_n(r) - 2 \cdot r_v(p)$$

Zuletzt hatten wir festgestellt:

$$a_{zig-zig} \leq 2 + r_n(p) + r_n(r) - 2 \cdot r_v(p)$$

Diese Summe ist genau dann kleiner oder gleich $3(r_n(p) - r_v(p))$, wenn $r_v(p) + r_n(r) \leq 2 \cdot r_n(p) - 2$ gilt.

$$\begin{aligned} 2 + \cancel{r_n(p)} + r_n(r) - \cancel{2r_v(p)} &\overset{!}{\leq} \cancel{r_n(p)} - r_v(p) + 2r_n(p) - \cancel{2r_v(p)} \\ 2 + r_n(r) + r_v(p) &\overset{!}{\leq} 2r_n(p) \\ r_n(r) + r_v(p) &\overset{!}{\leq} 2r_n(p) - 2 \end{aligned}$$

Aus der zig-zig-Operation folgt, dass $s_v(p) + s_n(r) \leq s_n(p)$. Mit obiger Hilfsaussage und der Definition der Ränge erhält man die obige Ungleichung.

Damit ist das Lemma bewiesen.

Satz: Das Ausführen einer beliebigen Folge von m Such-, Einfüge- und Entferne-Operationen, in der höchstens n-mal Einfügen vorkommt und die mit dem anfangs leeren Splay-Baum beginnt, benötigt höchstens $\mathcal{O}(m \cdot \log(n))$ Schritte.

Weil für jeden im Verlauf der Operationsfolge erzeugten Baum $s(T) \leq n$ gilt und jede Operation ein konstantes Vielfaches der Kosten der Splay-Operation verursacht, folgt die Behauptung aus obigem Lemma.

Übersicht

Amortisierte Laufzeitanalysen

- dynamic tables
- Selbstanordnende lineare Listen
- Fibonacci-Heaps
- Splay-Bäume
- *Union-Find-Datenstruktur*

Betrachten wir noch einmal die Struktur und die einzelnen Operationen der Union-Find-Datenstruktur:

```
struct Node {
    int rank;              // depth of tree
    Node *parent;          // reversed tree
};

void makeSet(Node *x) {
    x->parent = x;
    x->rank = 0;
}

Node *findSet(Node *x) {  // path compression
    if (x->parent != x)
        x->parent = findSet(x->parent);
    return x->parent;
}
```

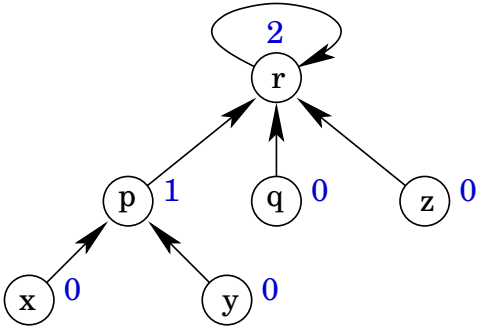
```
// union by rank
Node *mergeSets(Node *x, Node *y) {
    if (x->rank > y->rank) {
        y->parent = x;
        return x;
    }

    x->parent = y;
    if (x->rank == y->rank)
        y->rank += 1;
    return y;
}
```

Im Folgenden:

- Die Wurzel eines Union-Find-Baums nennen wir den Repräsentanten (leader) der Menge.
- Wir betrachten beliebige Folgen von m union-, find- und makeSet-Operationen über n Elementen.

Wir behalten die Sprechweise bzgl. Kinder und Vorgänger bei, obwohl die Bäume von den Blättern zur Wurzel gerichtet sind.



- r ist die Wurzel des Baums.
- p ist der Vorgänger von x und y.
- r ist der Repräsentant der Menge $\{p, q, r, x, y, z\}$.
- p ist die Wurzel des Teilbaums mit den Elementen $\{p, x, y\}$.

- Nur die Ränge von Repräsentanten werden geändert. Wenn ein Knoten kein Repräsentant einer Menge ist, dann wird sein Rang nicht mehr verändert.
  - Frage: Warum aktualisieren wir während der find-Operation nicht die Ränge der Knoten?
- Ist ein Knoten kein Repräsentant einer Menge, dann ist sein Rang echt kleiner als der Rang seines Vorgängers.
  - anders gesagt: Die Werte der Ränge auf einem Weg von einem Knoten zur Wurzel sind streng monoton steigend.
- Wir haben bereits gezeigt: Sei x ein Knoten mit Rang k. Dann besteht der Teilbaum mit Wurzel x aus mindestens 2^k Knoten.
- Es gibt höchstens $\frac{n}{2^k}$ Knoten mit Rang k.
  - Wenn einem Knoten der Rang k zugewiesen wird, markiere alle Knoten der Menge, das sind mindestens 2^k viele.
  - Es gibt nur n Knoten insgesamt, und jeder Knoten mit Rang k markiert mindestens 2^k Knoten.

Definition:

$$\log^{(i)}(n) := \begin{cases} n & \text{falls } i = 0 \\ \log(\log^{(i-1)}(n)) & \text{für } i \geq 1 \end{cases}$$

Und wir definieren $\log^\star(n) := \min\{i \geq 0 \mid \log^{(i)}(n) < 2\}$. Anders gesagt: Wie oft muss man die log-Taste des Taschenrechners nach Eingabe von n drücken, bis ein Wert kleiner als 2 angezeigt wird.

Beispiele:

- $\log^\star(2^1) = 1$
- $\log^\star(2^2) = 2$
- $\log^\star(2^{2^2}) = \log^\star(2^4) = 3$
- $\log^\star(2^{2^{2^2}}) = \log^\star(2^{16}) = 4$
- $\log^\star(2^{2^{2^{2^2}}}) = \log^\star(2^{65536}) = 5$

Definition:

$$tower(b) := \begin{cases} 1 & \text{falls } b = 0 \\ 2^{tower(b-1)} & \text{für } b \geq 1 \end{cases}$$

anschaulich:

$$tower(b) := \left. 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \right\} b$$

Zur Laufzeitabschätzung wollen wir den Knoten Blöcke zuordnen. Ein Knoten x wird dem Block b zugeordnet, wenn gilt:

$$tower(b-1) \;<\; rank(x) \;\leq\; tower(b)$$

Da jeder Knoten einen Rang zwischen 0 und $\log(n)$ hat, gibt es nur die Blocknummern 0 bis $\log^\star(\log(n)) = \log^\star(n) - 1$, also insgesamt höchstens $\log^\star(n)$ Blöcke.

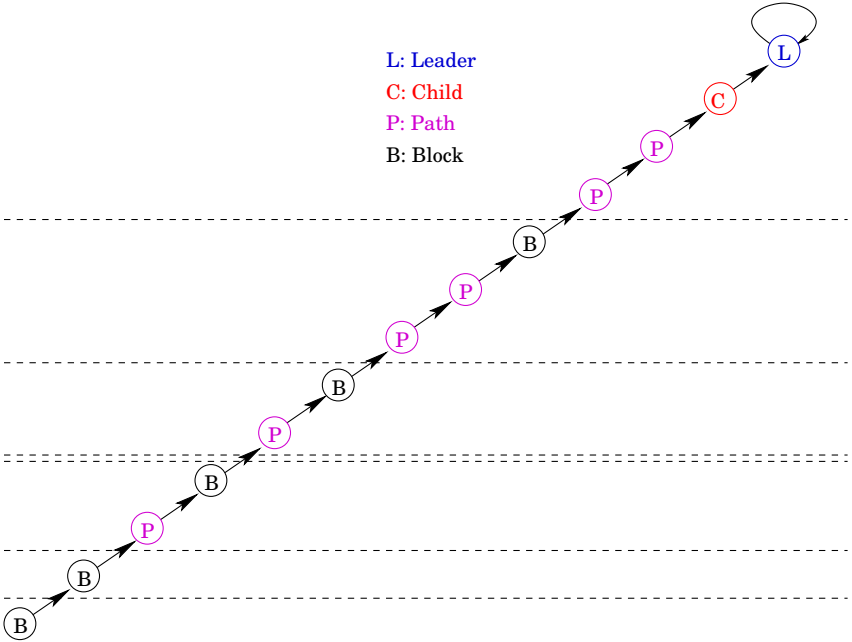
Da es höchstens $\frac{n}{2^r}$ Elemente vom Rang r gibt, ist die gesamte Anzahl der Knoten in Block b höchstens:

$$\begin{aligned}
\sum_{r=tower(b-1)+1}^{tower(b)} \frac{n}{2^r} &= n \cdot \sum_{r=tower(b-1)+1}^{tower(b)} \frac{1}{2^r} \\
&< n \cdot \sum_{r=tower(b-1)+1}^{\infty} \frac{1}{2^r} \\
&= n \cdot \left(\frac{1}{2^{tower(b-1)+1}} + \frac{1}{2^{tower(b-1)+2}} + \dots \right) \\
&\leq 2n \cdot \frac{1}{2^{tower(b-1)+1}} \\
&\leq n \cdot \frac{1}{2^{tower(b-1)}} = \frac{n}{tower(b)}
\end{aligned}$$

Obwohl wir im Folgenden immer von Konten sprechen, arbeiten wir nicht mit der Account- oder Bankkonto-Methode, sondern mit der Aggregat-Methode.

- Sei $x_1, x_2, x_3, \ldots, x_l$ der Weg von einem Knoten x_1 zum Repräsentanten x_l des Baums.
- Die Kosten von find(x_1) entsprechen der Länge l des Weges.
- Jeder Knoten auf dem Weg zahlt jeweils 1\$ in eines von mehreren Konten.
  - leader account: darin zahlen Repräsentanten ein
  - child account: darin zahlen Kinder der Repräsentanten ein
  - block accounts: darin zahlen Knoten x_k ein, die in einem anderen Block sind als deren Vorgänger x_{k+1}
  - path account: darin zahlen alle anderen Knoten ein
- Der Gesamtbetrag aus allen Konten gibt dann die Summe der Laufzeiten aller Operationen an.

Union-Find-Datenstruktur



Jede find-Operation zahlt

- 1\$ in das Repräsentanten-Konto (leader account),
- höchstens 1\$ in das Kinder-Konto (child account), und
- höchstens 1\$ in jedes Block-Konto (block account), wovon es nur $\log^\star(n)$ viele gibt.

Bei m Operationen über n Elementen wird insgesamt höchstens ein Wert von $2m + m \cdot \log^\star(n)$ in diese Konten eingezahlt.

Die Frage ist: Wie viel wird in das Pfad-Konto eingezahlt?

Betrachten wir einen Knoten x_i in Block b, der in das Pfad-Konto einzahlt:

- Der Knoten x_i wird nie zu einem Repräsentanten, also ändert sich sein Rang nicht und der Knoten bleibt auch nach einer Pfadkomprimierung in Block b.
- Der Vorgänger x_{i+1} von x_i ist ebenfalls kein Repräsentant, so dass durch eine Pfadkomprimierung x_i den neuen Vorgänger x_l bekommt, dessen Rang $rank(x_l)$ echt größer ist als der Rang $rank(x_{i+1})$ des alten Vorgängers x_{i+1}.
- Da die Ränge $rank(parent(x))$ streng monoton wachsen, liegt eventuell ein neuer Vorgänger von x_i in einem anderen Block als x_i selbst. x_i zahlt dann nie wieder in das Pfad-Konto ein.

$\rightarrow$ Der Knoten x_i zahlt höchstens einmal für jeden Rang in Block b ins Pfad-Konto, also höchstens $tower(b)$ oft.

- Da Block b weniger als $\frac{n}{tower(b)}$ Knoten enthält, und jeder dieser Knoten weniger als $tower(b)$ in das Pfad-Konto einzahlt, wird weniger als n in das Pfad-Konto eingezahlt.
- Da es nur $\log^\star(n)$ Blöcke gibt, wird also höchstens ein Wert von $n \cdot \log^\star(n)$ in das Pfad-Konto eingezahlt.

$\rightarrow$ In allen Konten zusammen wurde also höchstens der Wert $2m + m \cdot \log^\star(n) + n \cdot \log^\star(n) \in \mathcal{O}(m \cdot \log^\star(n))$ eingezahlt.

Damit haben wir das folgende Theorem bewiesen.

Theorem: Die find- und die union-Operation über n Elementen haben jeweils eine amortisierte Laufzeit von $\mathcal{O}(\log^\star(n))$.

Übersicht

Übersicht

Algorithmen für moderne Hardware

- *Motivation*
- Lokalität der Daten
- Externe Datenstrukturen

Motivation

Immer größere Datenmengen werden gesammelt und verarbeitet:

- geografische Informationssysteme
- Molekularbiologie (DNA-Sequenzen), Bioinformatik
- Suche im Web

Probleme:

- Geschwindigkeit der Prozessoren verbessert sich pro Jahr um etwa 30% - 50%, Geschwindigkeit des Speichers nur um 7% - 10% pro Jahr
- Betriebssysteme versuchen durch *caching* und *prefetching* I/O-Engpässe zu vermeiden (virtuelles Speichermanagement), aber: diese allgemeinen Algorithmen sind nicht speziell auf das jeweilige Problem optimiert
- Zugriff auf Hauptspeicher spricht kleine Blöcke an (cache-line 64 Byte bei Intel Core i7), Zugriff auf externen Speicher wie HDD/SSD spricht große Blöcke an

$\rightarrow$ greifen Algorithmen unstrukturiert auf Sekundärspeicher zu, ohne Lokalität der Zugriffe auszunutzen, erfolgen viel mehr Speicherzugriffe als nötig

Durchwandern eines Arrays

Initialisieren:

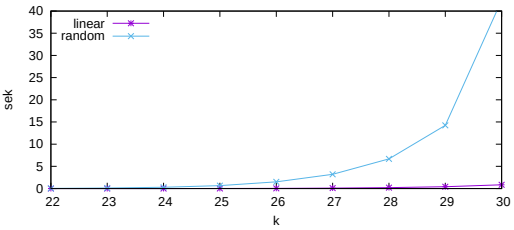
```
for (i = 0; i < N; i++)
     D[i] = i;
C = permute(D)
```

Lineares Durchlaufen:

```
for (i = 0; i < N; i++)
    A[D[i]] = A[D[i]] + 1;
```

Zufälliges Durchlaufen:

```
for (i = 0; i < N; i++)
    A[C[i]] = A[C[i]] + 1;
```



Laufzeiten für Arrays mit $N = 2^k$ vielen `int`-Elementen, $k = 22, \ldots, 30$ auf einem PC mit Intel Core i7, 2.80 GHz CPU, 16 GB RAM, Ubuntu 20.04.

Hierarchisches Speichermodell[1] für Intel Core i7:

- 1 cycle to read a CPU register
- 4 cycles to reach L1-Cache (32 kB + 32 kB, 64 byte per line)
- 11 cycles to reach L2-Cache (256 kB, 64 byte per line)
- 39 cycles to reach L3-Cache (2 MB per core (shared), 64 byte per line)
- ≈100 cycles to reach main memory (RAM)
- ≈10.000 of cycles to reach external memory

Wir müssen also versuchen unsere Programme so zu schreiben, dass die Daten möglichst im Cache vorhanden sind, wenn wir sie benötigen. Das funktioniert nicht immer, aber wir werden sehen, dass oft ein bisschen Nachdenken hilft, um die Programme signifikant zu beschleunigen.

Größen in einem 64-bit C-Programm: 64 byte per line = 16 `int` = 8 `long int`

[1] Ulrich Drepper: What every programmer should know about memory. `https://people.freebsd.org/~lstewart/articles/cpumemory.pdf`

Speichermodell

Speicherhierarchie ist sehr komplex und besitzt viele Abstufungen:

- Hardware Caches nutzen verschiedene, komplexe Ersetzungsstrategien
- Swap-Bereich: Mapping der virtuellen auf physikalische Speicheradressen erfolgt mittels Translation Lookaside Buffer.
- Vielzahl von Parametern resultiert in komplexen Modellen
- komplexe Modelle erschweren Analyse von Algorithmen und Datenstrukturen

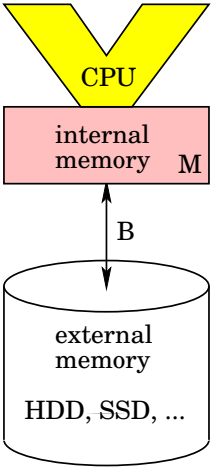
$\rightarrow$ externes Speichermodell als Kompromiss

externes Speichermodell

- starke Abstraktion der Speicherhierarchie auf nur 2 Ebenen
- wird oft verwendet für theoretische Analyse und Entwicklung von Algorithmen und Datenstrukturen
- Vitter[2]: Ergebnisse des Modells sind in die Praxis übertragbar

[2]Vitter, Jeffrey Scott: External memory algorithms and data structures: Dealing with Massive Data. ACM Computing Surveys, 33(2):209–271, 2001.

Externes Speichermodell



Parameter des Modells:

- M maximale Anzahl Datenelemente in dem internen Speicher
- B Anzahl Datenelemente, die mit einer I/O-Operation zwischen dem internen und dem externen Speicher ausgetauscht werden kann

Annahmen für die Performanceanalyse:

- Bandbreite ist unendlich und damit nicht der Flaschenhals der Auslagerung
- Datenzugriff auf internen Speicher kostet eine Zeiteinheit
- arithmetische Operationen kosten eine Zeiteinheit

$\rightarrow$ Anzahl der I/O-Operationen ist aussagekräftig

Externer Speicher kann auch RAM bezeichnen, dann ist der interne Speicher der Cache. Das Modell kann also auch genutzt werden, um Cache-Effekte zu untersuchen.

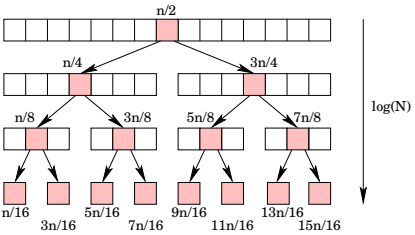
Übersicht

Algorithmen für moderne Hardware

- Motivation
- *Lokalität der Daten*
- Externe Datenstrukturen

Annahme: $n = 2^k - 1$ für ein $k \in \mathbb{N}$.

- zunächst müssen die Werte sortiert werden
- sortieren lohnt sich nur, wenn oft Werte gesucht werden
- bei jeder Suche wird zunächst auf das mittlere Element zugegriffen → Rang eins
- danach wird auf eines der mit Rang 2 markierten Elemente zugegriffen



Bei einem großen Array liegen das Rang-1-Element und die Rang-2-Elemente weit auseinander, also in verschiedenen Blöcken, sodass bei jedem Zugriff ein Cache-Miss auftritt.

Idee: platziere die Elemente, auf die oft zugegriffen wird, nah beieinander an den Anfang des Arrays

Rang:	4	3	4	2	4	3	4	1	4	3	4	2	4	3	4
Werte:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Rang:	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4
Werte:	8	4	12	2	6	10	14	1	3	5	7	9	11	13	15
Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Dazu muss aber eine völlig andere Sortierung der Daten vorgenommen werden und bei der binären Suche muss die Index-Berechnung des als nächstes zu untersuchenden Elements geändert werden.

Sortierung anpassen:

```
int i = 0;
for (int t = 2; t-1 <= N; t *= 2) {
    for (int p = 1; p < t; p += 2)
        r[i++] = s[p*N/t];
}
```

Binäre Suche

```
void *msearch(const void *key, const void *base, size_t nmemb,
        size_t size, int (*comp)(const void *, const void *)) {
    int p = 0;
    char *pos = (char *)base;

    while (p < nmemb) {
        int vgl = comp(key, pos);
        if (vgl == 0)
            return pos;
        if (vgl < 0) {
            pos += (p + 1) * size;
            p = 2*p + 1;
        } else {
            pos += (p + 2) * size;
            p = 2*p + 2;
        }
    }
    return nullptr;
}
```

Experiment:

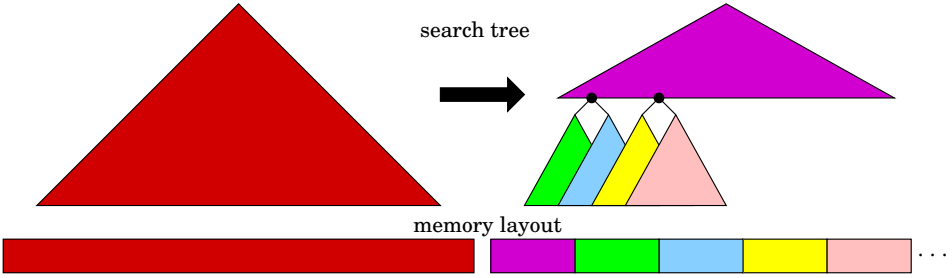
- erzeuge Array mit n Elementen vom Typ `int`
- `for (int i = 0; i < n; i++) A[i] = rand() % n`
- jedes Element von 0 bis n wird einmal gesucht und die Zeit in Sekunden gemessen
- vergleiche `msearch` mit `bsearch` aus der Standardbibliothek

n	qsort	bsearch	msearch
2^25 = 33554431	4	1	1
2^26 = 67108863	9	3	2
2^27 = 134217727	20	6	5
2^28 = 268435455	41	16	11
2^29 = 536870911	84	37	22
2^30 = 1073741823	198	86	46
2^31 = 2147483647	408	231	109

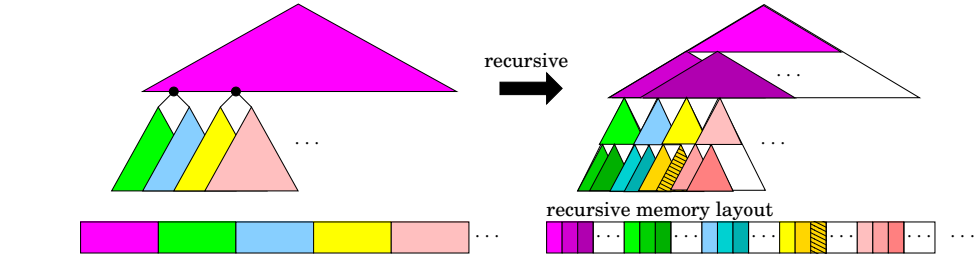
Problem: Die Lokalität der Daten wird nur für die ersten Zugriffe verbessert, weil nur die ersten Ebenen des „Suchbaums“ innerhalb eines Blocks liegen.

Besser: Layout nach van Emde Boas

- zunächst: erstelle vollständig balancierten Suchbaum
- teile den Baum in der Mitte bzgl. der Höhe



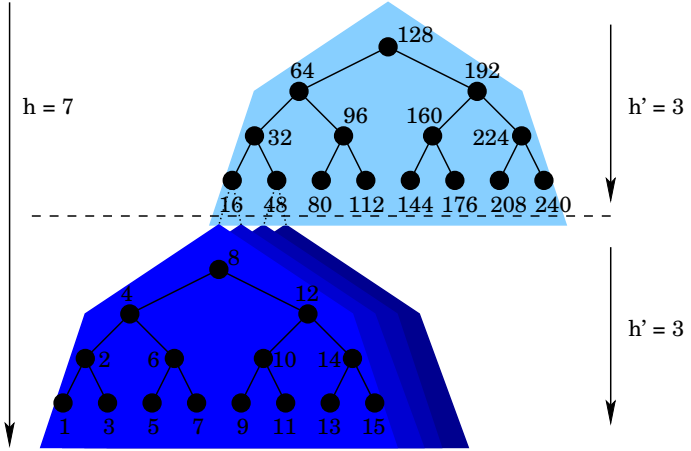
- platziere Teilbäume rekursiv im Speicher



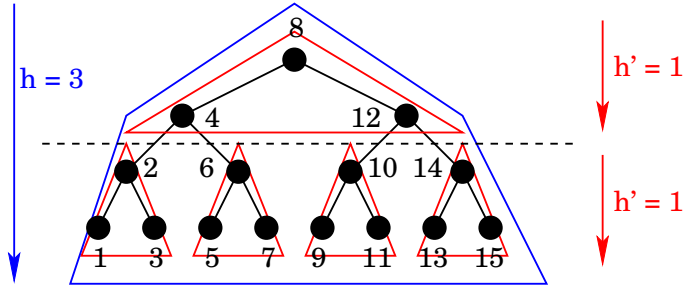
Ein Suchbaum der Höhe x hat 2^x viele Blätter und $2^{x+1} - 1$ viele Knoten.

- Suchbaum T habe Höhe $h = 2k + 1$, also $N = 2^{2k+2} - 1$ Knoten.
- Teilen wir T in der Mitte bzgl. der Höhe auf, erhalten wir oben einen Teilbaum der Höhe k und unten 2^{k+1} viele Teilbäume der Höhe k. Jeder dieser Teilbäume hat $2^{k+1} - 1$ Knoten.
- Insgesamt erhalten wir also $2^{k+1} - 1 + 2^{k+1} \cdot (2^{k+1} - 1) = 2^{2k+2} - 1$ viele Knoten.
- Anmerkung: Für $N = 2^x$ gilt $\sqrt{N} = 2^{x/2}$, nach der ersten Rekursion sind in den obigen Teilbäumen also etwa $\sqrt{N}$ viele Knoten enthalten.

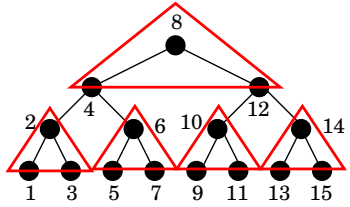
Beispiel: Für $N = 255$ erhalten wir einen Baum der Höhe $h = 7$. Teilen wir den Baum in der Mitte bzgl. der Höhe, so erhalten wir oben einen Baum der Höhe 3 mit 15 Knoten und unten 16 Bäume mit jeweils 15 Knoten, insgesamt also $17 \cdot 15 = 255$ Knoten.

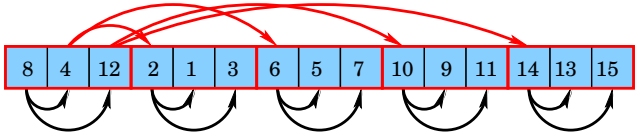


Für eine Blöckgröße von 3 Elementen müssen die Bäume der Höhe $h = 3$ aufgeteilt werden und es entstehen Bäume der Höhe 1, die jeweils 3 Knoten haben, insgesamt also $5 \cdot 3 = 15$ Knoten.

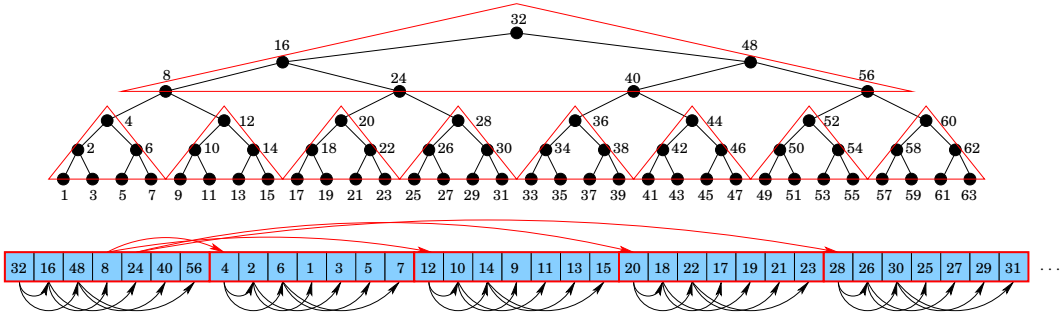


Für die Blockgröße 3 erhalten wir dann bspw. das folgende Layout:





Beispiel: Für eine Blockgröße 7 und $N = 63$ Elemente müssen wir den Baum nur einmal in der Mitte bzgl. der Höhe aufteilen und erhalten das folgende Layout:



Analyse: Betrachten wir einen Suchpfad von der Wurzel zu einem Blatt.

- die Länge des Pfades ist $\log_2(N)$
- bei einer Blockgröße B hat ein Teilbaum mit B Knoten die Höhe $\log_2(B)$
- beim Durchlaufen des Suchpfads müssen $\log_2(N)/\log_2(B)$ Blöcke gelesen werden
- es müssen $\log_B(N) = \log_2(N)/\log_2(B)$ viele I/Os durchgeführt werden

$\rightarrow$ Reduktion von $\log_2(N)$ auf $\log_B(N)$

Anwendung beim Rechnen mit Matrizen, z.B. bei der schnellen Fourier-Transformation.

Die i-te Zeile der transponierten Matrix A^T entspricht der i-ten Spalte der Matrix A, die transponierte Matrix A^T entsteht also durch Spiegelung der Matrix A an ihrer Hauptdiagonalen.

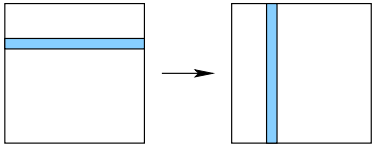
$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \end{pmatrix} \quad \rightarrow \quad A^T = \begin{pmatrix} 1 & 5 & 9 & 13 & 17 & 21 & 25 & 29 \\ 2 & 6 & 10 & 14 & 18 & 22 & 26 & 30 \\ 3 & 7 & 11 & 15 & 19 & 23 & 27 & 31 \\ 4 & 8 & 12 & 16 & 20 & 24 & 28 & 32 \end{pmatrix}$$

Wir wollen uns hier ansehen, wie man durch Ausnutzen von Cache-Effekten das Tranponieren einer Matrix effizient implementieren kann.

Vereinfachung: Wir betrachten nur quadratische Matrizen, also $n = m$.

Im einfachsten Fall können wir das Transponieren einer Matrix A mittels zweier verschachtelter Schleifen realisieren:

```
for (int r = 0; r < n; r++)
    for (int c = 0; c < n; c++)
        AT[c][r] = A[r][c];
```



Annahme:

- Die Größe eines Blocks beträgt B Elemente, es werden also bei einer I/O-Operation B Elemente der Matrix gelesen oder geschrieben. (1)
- Nicht alle n Elemente einer Zeile passen in den Cache. (2)

Analyse:

- eine Zeile aus A lesen $\rightarrow {}^{n}/_{B}$ viele I/O-Operationen wg. (1) und (2)
- eine Spalte nach A^T schreiben $\rightarrow n$ viele I/O-Operationen wg. (2)

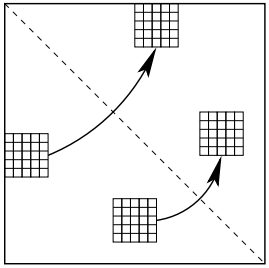
$\rightarrow$ insgesamt also $n \cdot ({}^{n}/_{B} + n) = n^2 + {}^{n^2}/_{B} \in \mathcal{O}(n^2)$ viele I/O-Operationen

Transponieren einer Matrix

Wir gehen daher anders vor: Zur Cache-Größe C definieren wir eine Blockgröße B, sodass $2B^2 \leq C$ gilt und daher beide Teilmatrizen in den Cache passen.

- Durchlaufe die Teilmatrizen der Größe $B \times B$ von A und transponiere diese Teilmatrizen.
- 1 Block von A lesen $\rightarrow B$ viele I/Os
- 1 Block nach A^T schreiben $\rightarrow B$ viele I/Os
- insgesamt gibt es $n/B \cdot n/B$ viele Teilmatrizen

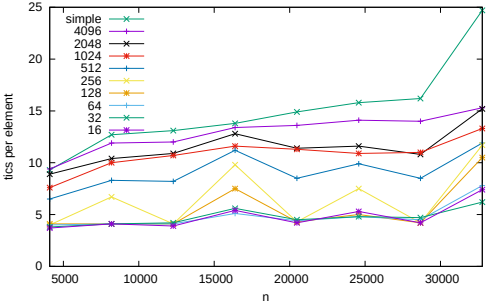
$\rightarrow$ $n^2/B^2 \cdot 2B = 2n^2/B \in \mathcal{O}(n^2/B)$ viele I/Os

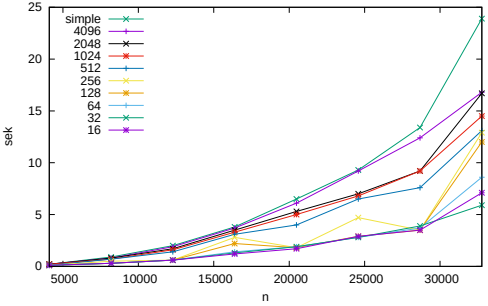


```
for (int i = 0; i < n; i += blocksize)
    for (int j = 0; j < n; j += blocksize)
        // transpose the block beginning at [i,j]
        for (int k = i; k < i + blocksize; ++k)
            for (int l = j; l < j + blocksize; ++l)
                AT[l][k] = A[k][l];
```

Welche Blockgrößen passen in welchen Cache?

- 32 kB großer L1-Cache: 64×64 `int` entspricht 16 kB (beste Resultate)
- 256 kB großer L2-Cache: 256×256 `int` entspricht 256 kB
- 8 MB großer L3-Cache: 1024×1024 `int` entspricht 4 MB





Wenn die Teilmatrizen nicht mehr in den Cache passen, dann ist die Performanz bei beiden Verfahren annähernd gleich schlecht.

Zur Vereinfachung beschränken wir uns wieder auf quadratische $n \times n$ Matrizen.

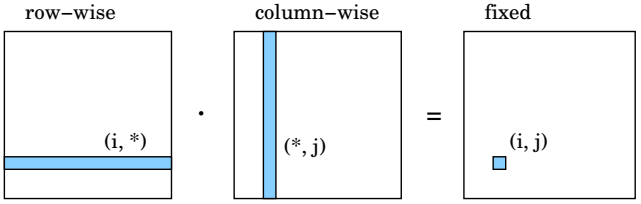
$$\begin{pmatrix} z_{11} & \cdots & z_{1n} \\ z_{21} & \cdots & z_{2n} \\ \vdots & \ddots & \vdots \\ z_{n1} & \cdots & z_{nn} \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ x_{21} & \cdots & x_{2n} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nn} \end{pmatrix} \cdot \begin{pmatrix} y_{11} & \cdots & y_{1n} \\ y_{21} & \cdots & y_{2n} \\ \vdots & \ddots & \vdots \\ y_{n1} & \cdots & y_{nn} \end{pmatrix} \text{ mit } z_{ij} = \sum_{k=1}^{n} x_{ik} \cdot y_{kj}$$

Da insgesamt n^2 viele Elemente z_{ij} der Ergebnismatrix zu berechnen sind und jede einzelne dieser Berechnungen einen Aufwand $\mathcal{O}(n)$ hat, ergibt sich insgesamt ein Aufwand von $\mathcal{O}(n^3)$ für $n \times n$ Matrizen bei Nutzung des naiven ijk-Verfahrens:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            Z[i][j] += X[i][k] * Y[k][j];
```

Problem der obigen, einfachen Matrix-Multiplikation:

- Wir nehmen wieder an, dass eine Zeile der Matrix nicht in den Cache passt.



- Für ein festes Paar (i, j), also bei der Berechnung eines Wertes z_{ij} gilt:
  - eine Zeile von X lesen $\rightarrow {}^{n}/_{B}$ viele I/O-Operationen
  - eine Spalte von Y lesen $\rightarrow n$ viele I/O-Operationen

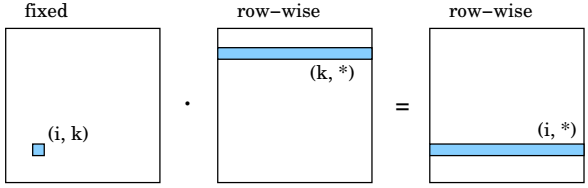
  $\rightarrow$ $n + {}^{n}/_{B}$ viele I/O-Operationen, um einen Wert z_{ij} zu berechnen
- Um n^2 Werte der Matrix Z zu berechnen werden also $n^2 \cdot (n + {}^{n}/_{B}) = n^3 + {}^{n^3}/_{B} \in \mathcal{O}(n^3)$ viele I/O-Operationen benötigt.

Durch einen einfachen Trick können wir die Anzahl der I/O-Operationen um etwa den Faktor $B/2$ reduzieren: Tausche die Zeilen 2 und 3 im obigen Code. → ikj-Verfahren

```
for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
        for (int j = 0; j < n; j++)
            Z[i][j] += X[i][k] * Y[k][j];
```

Was erreichen wir dadurch?

- Für ein festes (i, k), also für den Durchlauf der inneren Schleife, muss jeweils auf eine Zeile aus Matrix Y und Z zugegriffen werden. → $2 \cdot n/B$ viele I/O-Operationen



- es gibt insgesamt n^2 viele Paare (i, k), also $n^2 \cdot 2n/B = 2n^3/B \in \mathcal{O}(n^3/B)$ viele I/Os

Der Wert der Matrix X, der in der innersten Schleife nicht geändert wird, kann in einer Variablen gespeichert werden, die der Compiler vermutlich in einem Register ablegt.

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < n; k++) {
        register int r = X[i][k];      // !!!!!

        for (int j = 0; j < n; j++)
            Z[i][j] += r * Y[k][j];
    }
}
```

Dadurch werden sehr teure Index-Zugriffe vermieden.

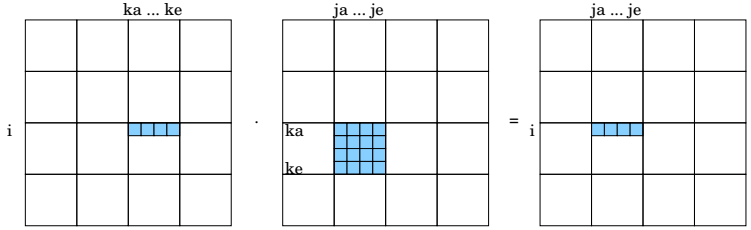
Es geht aber noch besser mit der gleichen Idee, die wir schon beim Transponieren einer Matrix kennen gelernt haben. Wir teilen die Matrizen in Teilmatrizen auf, sodass die Teilmatrizen in den Cache passen. Um die passende Blockgröße zu bestimmen, haben wir im folgenden Programm mit verschiedenen Blockgrößen experimentiert.

```
for (int ka = 0; ka < n; ka += blk) {
    int ke = min(ka + blk, n);
    for (int ja = 0; ja < n; ja += blk) {
        int je = min(ja + blk, n);
        for (int i = 0; i < n; i++) {
            for (int k = ka; k < ke; k++) {
                register int r = X[i][k];

                for (int j = ja; j < je; j++)
                    Z[i][j] += r * Y[k][j];
            }
        }
    }
}
```

Annahme: eine $B \times B$ Teilmatrix passt in den Cache

- Für ein festes Tupel (ka, ja, i) ergeben sich $B + 2$ viele I/O-Operationen:
  - um eine Blockzeile von X zu lesen $\rightarrow$ 1 I/O
  - um einen ganzen Block von Y zu lesen $\rightarrow$ B I/O
  - um eine Blockzeile von Z zu schreiben $\rightarrow$ 1 I/O



- Wird der Wert i geändert, ergibt sich ein Cache-Miss, weil eine Zeile nicht in den Cache passt. Für ein festes (ka, ja) ergeben sich also $n \cdot (B + 2)$ viele I/Os.
- Weil es $n/B \cdot n/B$ viele Paare (ka, ja) gibt, erhalten wir also insgesamt $n^2/B^2 \cdot n \cdot (B + 2) = n^3/B + 2n^3/B^2 \in \mathcal{O}(n^3/B)$ viele I/Os. (etwa Faktor 2 besser)

An den Laufzeiten sehen wir sehr schön das kubische Wachstum der Funktion:

$$T(n) \approx n^3 \quad \text{und} \quad T(2n) \approx (2n)^3 \;=\; 2^3 \cdot n^3 \;=\; 8n^3$$

n	ikjf	ikj	32	64	128	256	512	1024	ijk
512	0.1	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.2
1024	0.6	0.8	0.5	0.4	0.4	0.3	0.3	0.3	3.3
2048	5.8	6.8	4.4	3.7	3.4	2.8	2.4	2.6	65.3
4096	46.2	54.5	41.0	31.9	27.0	22.6	19.7	20.6	602.1
8192	371.9	441.0	381.0	262.0	212.9	232.8	211.4	188.4	6671.4

Bewertung:

- Das ijk-Verfahren sollte nach unserer Analyse etwa um den Faktor $B/2 = 16/2 = 8$ langsamer sein als das ikj-Verfahren.
- Das geblockte Verfahren sollte etwa um Faktor 2 schneller sein als ikj-Verfahren

Übersicht

Algorithmen für moderne Hardware

- Motivation
- Lokalität der Daten
- *Externe Datenstrukturen*[3]

[3] basiert auf einer Vorlesung von Petra Mutzel, TU Dortmund

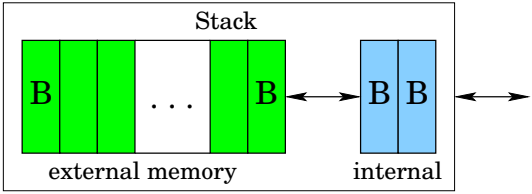
Große Datenmengen können nicht im Hauptspeicher verwaltet werden, daher müssen Teile der Daten auf einen sekundären oder externen Speicher wie HDD, SSD oder Flash-Medien ausgelagert werden.

Die Seitenersetzungsverfahren wie LRU der Betriebssysteme sowie das Caching und Prefetching sind allgemeine Verfahren, die nicht speziell auf die jeweiligen Probleme abgestimmt sind.

Hier: Spezielle (einfache) Datenstrukturen, um große Datenmengen mit wenigen I/O-Zugriffen verwalten zu können.

- Stack
- Queue
- Lineare Liste
- B-Baum
- externes Hashing
- Array-Heap

Nutze kombinierten Input/Output-Buffer der Größe $2 \cdot B$:



push
- füge neues Element in den Buffer ein
- falls Buffer voll ist: schreibe unteren Block auf die Festplatte

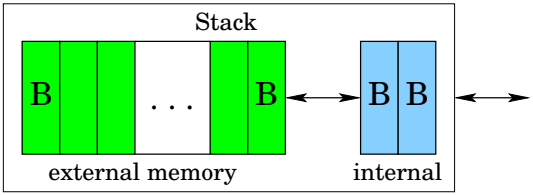
$\rightarrow$ $1/B$ viele I/Os amortisiert

top/pop
- lese/entferne Element aus dem Buffer
- falls Buffer leer ist: lese Block von der Festplatte in den Buffer ein

$\rightarrow$ $1/B$ viele I/Os amortisiert

Frage: Warum hat der Buffer eine Größe von $2 \cdot B$?

Nutze kombinierten Input/Output-Buffer der Größe $2 \cdot B$:



push
- füge neues Element in den Buffer ein
- falls Buffer voll ist: schreibe unteren Block auf die Festplatte

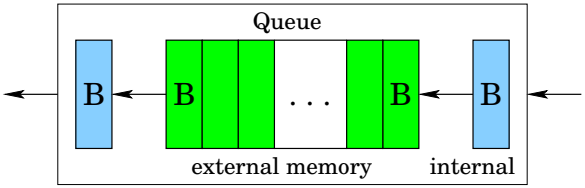
$\rightarrow$ $1/B$ viele I/Os amortisiert

top/pop
- lese/entferne Element aus dem Buffer
- falls Buffer leer ist: lese Block von der Festplatte in den Buffer ein

$\rightarrow$ $1/B$ viele I/Os amortisiert

Hätte Buffer nur Größe B: Zunächst $B - 1$ mal push ausführen, dann abwechselnd 2 x push und 2 x pop: Bei jedem zweiten Zugriff findet eine I/O-Operation statt.

Nutze getrennte Input- und Output-Buffer jeweils der Größe B:



put
- füge neues Element am Ende des Input-Buffers ein
- falls Input-Buffer voll ist: schreibe Buffer auf die Festplatte

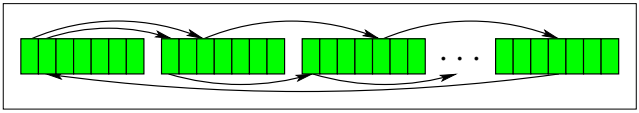
$\rightarrow$ $1/B$ viele I/Os amortisiert

head/get
- lese/entferne erstes Element aus dem Output-Buffer
- falls Output-Buffer leer ist:
  - lese Block von der Festplatte oder,
  - falls leer, aus dem Input-Buffer

$\rightarrow$ $1/B$ viele I/Os amortisiert

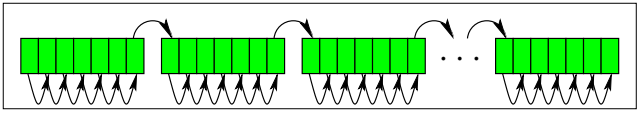
Versuch 1:

- einfach oder doppelt verkettete Elemente können so im Speicher verteilt liegen, dass aufeinanderfolgende Elemente in unterschiedlichen Blöcken liegen
- direkte Implementierung liefert pro Zugriff eine I/O-Operation



Versuch 2:

- speichere B konsekutive Listenelemente in einem Block und verkette die Blöcke

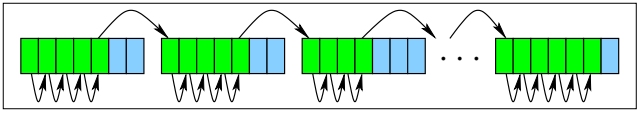


- ein Scan über alle n Elemente benötigt n/B viele I/O-Operationen ☺
- einfügen oder löschen eines Elements benötigt noch einmal n/B viele I/Os ☹

besser: Blöcke müssen nicht komplett gefüllt sein

- stelle sicher: *jedes Paar konsekutiver Blöcke* enthält mindestens $2/3B$ viele Elemente

→ maximal $3n/B \in \mathcal{O}(n/B)$ viele Blöcke werden zur Speicherung der n Werte benötigt



insert

- traversiere die Liste bis zur richtigen Stelle $\rightarrow \mathcal{O}(n/B)$ viele I/Os
- das eigentliche Einfügen kostet meist nur 1 I/O, außer wenn Block voll ist, dann:
  - falls benachbarter Block noch Platz hat: tausche ein Element aus $\rightarrow \mathcal{O}(1)$ viele I/Os
  - falls beide Nachbarblöcke voll sind:
    - teile den Block in 2 Teile der Größe $\approx B/2 \rightarrow \mathcal{O}(1)$ viele I/Os
    - danach sind mindestens $B/6$ Lösch-Operationen nötig, um Invariante zu verletzen

→ $\mathcal{O}(1 + n/B)$ viele I/Os

erase

- traversiere die Liste bis zur richtigen Stelle $\rightarrow \mathcal{O}(n/B)$ viele I/Os
- das eigentliche Löschen kostet meist nur 1 I/O, außer wenn der Block danach mit einem der beiden benachbarten Blöcke weniger als $2/3B$ Elemente besitzt
- dann: verschmelze die beiden Blöcke $\rightarrow \mathcal{O}(1)$ viele I/Os

$\rightarrow \mathcal{O}(1 + n/B)$ viele I/Os

Frage: Warum nicht einfach fordern, dass jeder Block mindestens zu einem Drittel gefüllt sein muss? Warum die „seltsame“ Regel mit Paaren von konsekutiven Blöcken?

erase

- traversiere die Liste bis zur richtigen Stelle $\rightarrow \mathcal{O}(n/B)$ viele I/Os
- das eigentliche Löschen kostet meist nur 1 I/O, außer wenn der Block danach mit einem der beiden benachbarten Blöcke weniger als $2/3B$ Elemente besitzt
- dann: verschmelze die beiden Blöcke $\rightarrow \mathcal{O}(1)$ viele I/Os

$\rightarrow \mathcal{O}(1 + n/B)$ viele I/Os

Beim Löschen von Elementen müsste der Block, aus dem das Element gelöscht wurde, mit einem benachbarten Block verschmolzen werden, sobald nicht mehr genug Elemente im Block vorhanden sind.

Das Zusammenfassen geht nicht, falls in den Nachbarblöcken zu viele Elemente enthalten sind. Dann müsste wieder gesplittet werden usw. Daher muss der Füllgrad nur für benachbarte Paare von Blöcken gelten.

B-Bäume und B*-Bäume sind externe Datenstrukturen. Sei b die Anzahl der Schlüssel, die in einem Knoten abgelegt werden können. Für $B = 4096$ Bytes lassen sich inklusive der zugehörigen Zeiger etwa

- $b = 256$ Elemente vom Typ `long int` oder
- $b = 340$ Elemente vom Typ `int` speichern.

unter Linux:

- `sudo tune2fs -l /dev/sda1` liefert Blockgröße des Dateisystems
- `sudo fdisk -l /dev/sda` liefert Größe der Sektoren

C++

- `sizeof(uintptr_t)` 8 Byte
- `sizeof(size_t)` 8 Byte
- `sizeof(int)` 4 Byte
- `sizeof(long int)` 8 Byte

Bei obigen Werten b und B können in einem B-Baum der Höhe h also etwa 256^{h+1} viele `long int`-Werte abgespeichert werden, wobei maximal h viele I/Os nötig sind.

Höhe	ungefähre Anzahl Elemente	
3	$16384 \cdot 2^{10}$	(16384 Kilo oder 16 Mega)
4	$4096 \cdot 2^{20}$	(4096 Mega oder 4 Giga)
5	$1024 \cdot 2^{30}$	(1024 Giga oder 1 Tera)
6	$256 \cdot 2^{40}$	(256 Tera)
7	$65536 \cdot 2^{40}$	(65536 Tera oder 64 Peta)
8	$16384 \cdot 2^{50}$	(16384 Peta oder 16 Exa)

Suchen, Einfügen und Löschen benötigen $\mathcal{O}(\log_b(N))$ viele I/Os.

Bereichsanfragen lassen sich mit B*-Bäumen aufgrund der verketteten Blätter sehr effizient realisieren.

Hashing = Schlüsselsuche durch Berechnung der Array-Indizes!

Idee:

- Notation: $[m] := \{0, 1, \ldots, m-1\}$
- bei einer Menge K von Werten aus einem Universum $U = [m]$, also $K \subseteq U$,
- verwende ein Array $A[0 \ldots m-1]$ und speichere die Schlüssel wie folgt:

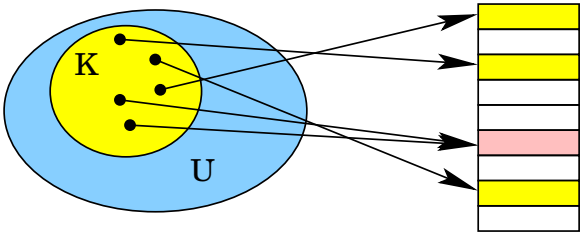
$$A[k] = \begin{cases} x & \text{falls } x.key \in K \text{ und } x.key = k \\ nil & \text{sonst} \end{cases}$$

$\rightarrow$ suchen, einfügen und löschen in $\mathcal{O}(1)$ Schritten

Probleme: Der Wertebereich $[m]$ kann sehr groß sein. Für 8-stellige Namen ergeben sich ungefähr $26^8 \approx 208 \cdot 10^9$ viele mögliche Werte. Außerdem wird viel Speicherplatz verschwendet, weil in der Regel nur wenige Plätze im Array genutzt werden.

[4] Wiederholung aus ALD

Lösung: Verwende Hash-Funktion h, um Wertebereich U und insbesondere die Menge der Schlüsselwerte $K \subseteq U$ auf Zahlen in $[n]$, $n \lll m$, abzubilden, also $h : U \to [n]$.



Anmerkungen:

- Die Hash-Funktion ist im Allg. nicht injektiv, d.h. verschiedene Schlüssel werden auf dieselbe Hashadresse abgebildet. $\to$ Adress-Kollision
  - Schlüssel, die auf die gleiche Adresse abgebildet werden, heißen *Synonyme*.
  - Die Menge der Synonyme bezüglich einer Adresse heißt *Kollisionsklasse*.
- Da die Hash-Funktion zum Platzieren und Suchen verwendet wird, muss sie einfach bzw. effizient zu berechnen sein.

Ein Hashverfahren muss zwei Anforderungen genügen:

1. es sollen möglichst wenige Adress-Kollisionen auftreten
   - Hash-Funktion soll die zu speichernden Datensätze möglichst gleichmäßig auf den Speicherbereich verteilen, um Adress-Kollisionen zu vermeiden.
   - Häufungen in der Schlüsselverteilung sollen sich nicht auf die Verteilung der Adressen auswirken.
   - Es gilt: Wenn eine Hash-Funktion $\sqrt{\pi n/2}$ Schlüssel auf eine Tabelle der Größe n abbildet, dann gibt es fast sicher eine Kollision. (für $n = 365$ ist $\sqrt{\pi n/2} \approx 23$)
2. die Adress-Kollisionen müssen effizient aufgelöst werden

Division-Rest-Methode: Der Schlüssel wird ganzzahlig durch die Länge der Hashtabelle dividiert. Der Rest wird als Index verwendet: $h(k) = k \bmod m$

- Der Wert m soll keinen kleinen Teiler haben und soll keine Potenz der Basis des Zahlensystems sein!
  - Beispiel: für $m = 2^r$ hängt der Hash-Wert nur von den letzten r Bit ab

$\rightarrow$ Wähle m als Primzahl, die nicht nah an Potenz der Basis des Zahlensystems liegt.

- Beispiel: $m = 761$, aber nicht $m = 509$ (nah an 2^9) oder $m = 997$ (nah an 10^3)

Multiplikative Methode: Sei $m = 2^r$ eine Zweierpotenz. Bei einer Wortgröße w wähle eine Zahl a so, dass $2^{w-1} < a < 2^w$ ist.

$$h(k) = (k \cdot a \bmod 2^w) \textbf{ div } 2^{w-r}$$

Anmerkungen:

- Wähle a nicht zu nah an 2^w
- Modulo-Operation und Rechts-Shift (Integer-Division) ist schnell
- gute Ergebnisse für $a \approx \frac{\sqrt{5}-1}{2} \cdot 2^w$ (goldener Schnitt)

Beispiel: Für $w = 8$, $r = 3$, $a = 191$ und $k = 23$ erhalten wir $h(k) = 1$.

$$\begin{array}{rrr} & 1\,0\,1\,1\,1\,1\,1\,1 & 191 \\ * & 0\,0\,0\,1\,0\,1\,1\,1 & 23 \\ \hline 1\,0\,0\,0\,1\,0\,0\,1\,0\,1\,0\,0\,1 & & 4393 \end{array}$$

Bewertung: Güte der Hash-Funktion h hängt von der gewählten Schlüsselmenge K ab.

- Güte ist nur unzureichend analysierbar
- zu h lässt sich immer ein K finden mit besonders vielen Kollisionen
- keine Hash-Funktion ist immer besser als alle anderen

Probleme treten auf

- beim Einfügen, wenn die berechnete Hashadresse nicht leer ist
- bei der Suche, wenn der berechnete Platz ein anderes Element enthält

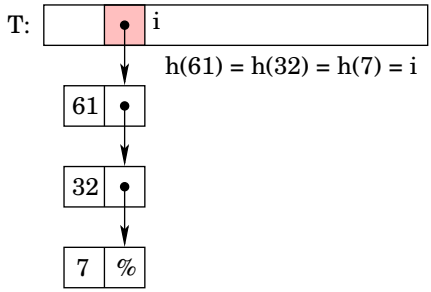
Kollisionsbehandlung für k_p falls $h(k_q) = h(k_p)$ für ein bereits gespeichertes k_q gilt:

- füge k_p in einem separaten Überlaufbereich außerhalb der Hashtabelle ein
  $\rightarrow$ *Verkettung der Überläufer*
- füge k_p an einem freien Platz innerhalb der Hashtabelle ein
  $\rightarrow$ *offenes Hashing*

Hashing: Verkettung der Überläufer

Kollisionsauflösung:

- Überläufer werden in linearer Liste verkettet
- Liste wird an Hashtabelleneintrag angehängt
- Verkettung der Synonyme (Überläufer) pro Kollisionsklasse
- Suchen, Einfügen und Löschen sind auf eine Kollisionsklasse beschränkt



Durchschnittliche Anzahl der Einträge in $h(k)$ ist N/M, wenn N Einträge durch die Hash-Funktion gleichmäßig auf M Listen verteilt werden. *Belegungsfaktor:* $\alpha = N/M$

Kosten von Zugriffen sind abhängig vom Belegungsfaktor.

- $S(\alpha)$: Kosten für erfolgreiche Suche und Löschen (finde Schlüssel: Search)
- $U(\alpha)$: Kosten für erfolglose Suche (entspricht Einfügekosten: Update)

Mittlere Laufzeit bei erfolgreicher Suche:

- beim Einfügen des j-ten Schlüssels ist die durchschnittliche Listenlänge $j-1/M$
- spätere Suche nach j-tem Schlüssel betrachtet im Schnitt $1 + j-1/M$ Einträge, wenn stets am Listenende eingefügt wird und keine Elemente gelöscht wurden

$$S(\alpha) = \frac{1}{N}\sum_{j=1}^{N}\left(1+\frac{j-1}{M}\right) = 1 + \frac{1}{NM}\cdot\frac{(N-1)N}{2} = 1 + \frac{N-1}{2M} \approx 1 + \frac{N}{2M} = 1 + \frac{\alpha}{2}$$

Mittlere Laufzeit bei erfolgloser Suche: $U(\alpha) = 1 + N/M = 1 + \alpha$

In der Regel gilt $N \in \mathcal{O}(M)$: Die Größe der Hashtabelle ist ungefähr so groß wie die Anzahl der Datensätze. $\rightarrow \alpha = N/M = \mathcal{O}(M)/M \in \mathcal{O}(1)$

Zum Vergleich: Binäre Suche hat Laufzeit $\mathcal{O}(\log(N))$ plus Aufwand $\mathcal{O}(N \cdot \log(N))$ zum Sortieren der Datensätze.

Problem: dynamische Speicheranforderung ist teuer

Hashing: offene Hashverfahren

Idee: Speichere Überläufer in der Hashtabelle, nicht in zusätzlichen Listen.

- Ist Hashadresse $h(k)$ belegt, suche systematisch eine Ausweichposition.
- *Sondierungsfolge:* Folge der zu betrachtenden Speicherplätze für einen Schlüssel.
- Die Hash-Funktion hängt nun vom Schlüssel und von der Anzahl durchgeführter Platzierungsversuche ab:

$$h : U \times [m] \to [m]$$

- *wichtig:* Die Sondierungsfolge muss eine Permutation der Zahlen $0, \ldots, m-1$ sein, damit alle Einträge der Hashtabelle genutzt werden können.

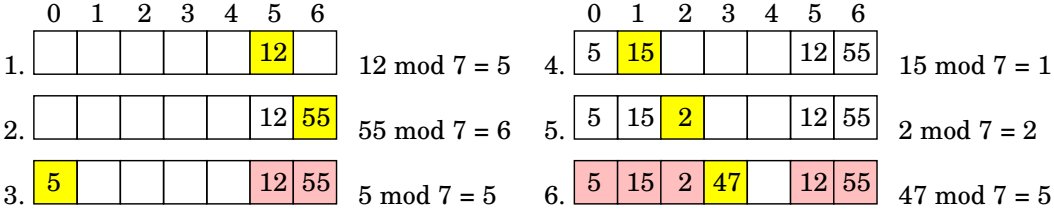
Anmerkungen:

- Beim Einfügen und Suchen wird dieselbe Sondierungsfolge durchlaufen.
- Beim Löschen wird der Datensatz nicht gelöscht, sondern nur als gelöscht markiert: Der Wert wird ggf. bei einem späteren Einfügen überschrieben.
- Je voller die Tabelle wird, umso schwieriger wird das Einfügen neuer Schlüssel.

Zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k,i) = (h'(k) + i) \bmod M.$$

Beispiel: Betrachte das Einfügen der Schlüssel $12, 55, 5, 15, 2, 47$ für $M = 7$ und $h'(k) = k \bmod 7$.

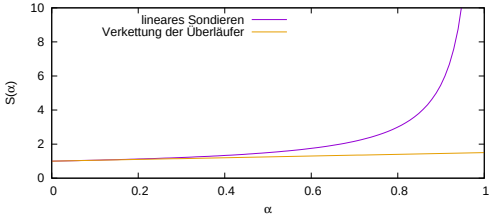


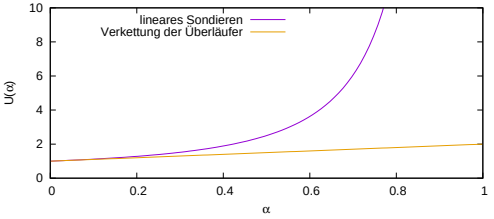
Die Sondierungsfolge $1, 2, 3, \ldots, m$ (modulo m) ist offensichtlich eine Permutation der Hash-Adressen, aber führt zu *primärer Häufung*: Behandlung einer Kollision erhöht Wahrscheinlichkeit einer Kollision in benachbarten Tabelleneinträgen.

Analyse nach Knuth[5] für $0 \leq \alpha < 1$:

$$S(\alpha) \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right), \qquad U(\alpha) \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Vergleich *Verkettung der Überläufer* mit *linearem Sondieren*:



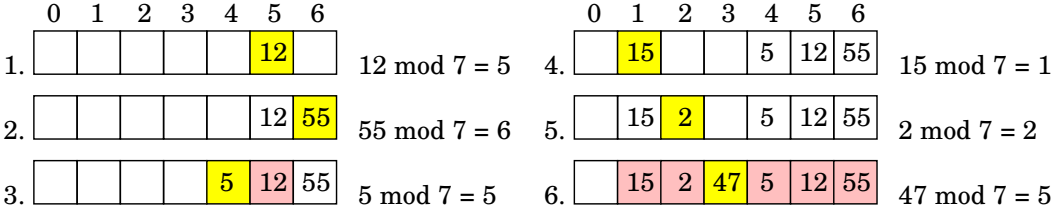


[5] Knuth, Donald E.: The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, 1973.

Zu einer gegebenen Hash-Funktion $h'(k)$ sei

$$h(k,i) = (h'(k) + (-1)^i \cdot \lceil i/2 \rceil^2) \bmod M.$$

Beispiel: Betrachte das Einfügen der Schlüssel $12, 55, 5, 15, 2, 47$ für $M = 7$ und $h'(k) = k \bmod 7$.

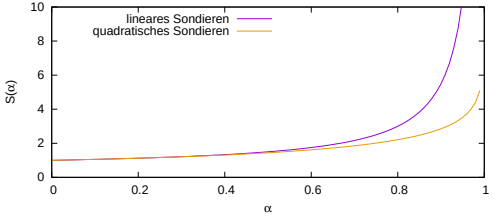


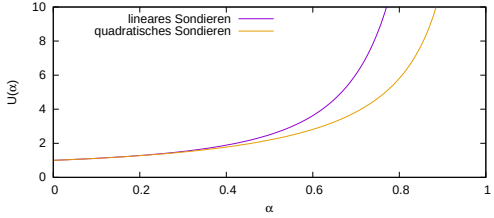
Für Primzahl $M = 4 \cdot j + 3 = 7, 11, 19, 23, 31, 43, \ldots$ ist die Sondierungsfolge eine Permutation der Hash-Adressen; keine lokale Häufung mehr, aber *sekundäre Häufung*: Schlüssel durchlaufen bei gleichem Hash-Wert immer die gleiche Ausweichsequenz!

Analyse nach Knuth für $0 \leq \alpha < 1$:

$$S(\alpha) \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}, \qquad U(\alpha) \approx \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right)$$

Vergleich *lineares Sondieren* mit *quadratischem Sondieren*:





Motivation: Funktion $h(k, i)$ berücksichtigt beim linearen und quadratischen Sondieren lediglich den Schritt i. Die Sondierungsfolge ist vom Schlüssel k unabhängig und damit für alle Synonyme die gleiche.

Schlüssel behindern sich weniger, wenn die Sondierungsfolge auch für Synonyme variiert. Uniformes Sondieren berechnet daher die Folge $h(k, 1), \ldots, h(k, M)$ als Permutation aller möglichen Hash-Werte in Abhängigkeit vom Schlüssel k.

- Vorteil: Häufung wird vermieden, da unterschiedliche Schlüssel mit gleichem Hash-Wert zu unterschiedlichen Sondierungsfolgen führen.
- Nachteil: schwierige praktische Realisierung

Approximation des uniformen Sondierens:

- zufälliges Sondieren: Wähle zufällige Hashadresse für $h(k, i)$, die abhängig von k ist. Ein belegter Platz wird evtl. noch einmal betrachtet, da es keine Permutation aller Hashadressen ist, aber dies ist selten.
- double hashing: später

Annahme: Sondierungsfolgen zu verschiedenen Schlüsseln sind zufällig und jede der $M!$ möglichen Permutationen wird mit gleicher Wahrscheinlichkeit $1/M!$ gewählt.

- *Ereignis* A_i: Beim i-ten Sondierungsversuch erfolgt Zugriff auf belegten Slot.
- *Zufallsvariable* X: Anzahl der sondierten Slots bei nicht erfolgreicher Suche.
- Dann gilt für $i \geq 2$: $\quad P(X \geq i) = P(A_1 \cap \ldots \cap A_{i-1})$
- verallgemeinerter Multiplikationssatz der Wahrscheinlichkeitsrechnung:

$$\begin{aligned} & P(A_1 \cap \ldots \cap A_{i-1}) \\ = \;& P(A_1) \cdot P(A_2 \mid A_1) \cdot P(A_3 \mid A_1 \cap A_2) \cdot \ldots \cdot P(A_i \mid A_1 \cap \ldots \cap A_{i-2}) \end{aligned}$$

Aufgrund unserer Annahme gilt:

$$P(A_1) = \frac{N}{M} \quad \text{und} \quad P(A_j \mid A_1 \cap \ldots \cap A_{j-1}) = \frac{N-(j-1)}{M-(j-1)}$$

Suche in $M-(j-1)$ Plätzen, von denen $N-(j-1)$ belegt sind.

Kumulierte Wahrscheinlichkeit, dass *erfolglose Suche* mindestens i Schritte braucht:

$$\begin{aligned} P(X \geq i) &= P(A_1 \cap \ldots \cap A_{i-1}) \\ &= P(A_1) \cdot P(A_2 \mid A_1) \cdot P(A_3 \mid A_1 \cap A_2) \cdot \ldots \cdot P(A_{i-1} \mid A_1 \cap \ldots \cap A_{i-2}) \\ &= \frac{N}{M} \cdot \frac{N-1}{M-1} \cdot \ldots \cdot \frac{N-i+2}{M-i+2} \leq \left(\frac{N}{M}\right)^{i-1} = \alpha^{i-1} \end{aligned}$$

Der Erwartungswert ist die Summe der kumulierten Wahrscheinlichkeiten:

$$U(\alpha) = \sum_{i=1}^{N} P(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

Bei einer *erfolgreichen Suche* wird dieselbe Sondierungsfolge wie beim Einfügen des Schlüssels durchlaufen. Für den als $(i+1)$-tes eingefügten Schlüssel entspricht dies einer erfolglosen Suche in einer Hashtabelle der Länge M mit i Schlüsseln:

$$q_i \;=\; \frac{1}{1-i/M} \;=\; \frac{M}{M-i}$$

Mittelwert über alle N Schlüssel:

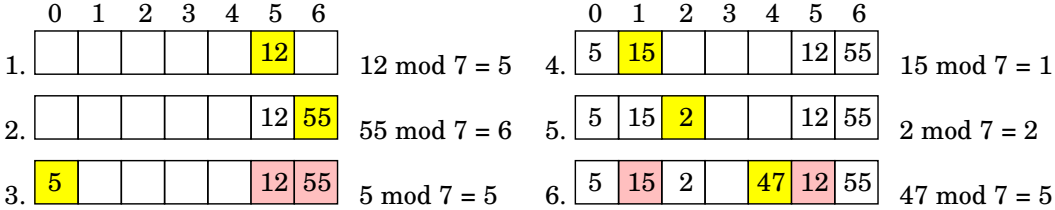
$$\begin{aligned} S(\alpha) \;&=\; \frac{1}{N}\cdot\sum_{i=0}^{N-1} q_i \;=\; \frac{M}{N}\cdot\sum_{i=0}^{N-1}\frac{1}{M-i} \;=\; \frac{1}{\alpha}\cdot\sum_{i=M-N+1}^{M}\frac{1}{i} \\ &\leq\; \frac{1}{\alpha}\cdot\int_{M-N}^{M}\frac{1}{x}dx \;=\; \frac{1}{\alpha}\big[\ln(M)-\ln(M-N)\big] \;=\; \frac{1}{\alpha}\ln\left(\frac{M}{M-N}\right) \\ &=\; \frac{1}{\alpha}\ln\left(\frac{1}{1-\frac{N}{M}}\right) \;=\; \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right) \end{aligned}$$

Für jede monoton fallende Funktion f gilt: $\sum_{i=m}^{n} f(i) \leq \int_{m-1}^{n} f(x)dx$

Zu zwei gegebenen Hash-Funktionen $h_1(k)$ und $h_2(k)$ sei

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod M.$$

Beispiel: Betrachte das Einfügen der Schlüssel $12, 55, 5, 15, 2, 47$ für $M = 7$, $h_1(k) = k \bmod 7$ und $h_2 = 1 + k \bmod 5$.



Knuth: We want to minimize the number of accesses to the file, and this has two major effects on the choice of algorithms:

- It is reasonable to spend more time computing the hash function, since the penalty for bad hashing is much greater than the cost of the extra time needed to do a careful job.
- The records are usually grouped into pages or buckets, so that several records are fetched from the external memory each time.

The file is divided into M buckets containing B records each. Collisions now cause no problem unless more than B keys have the same hash address.

$\rightarrow$ Hashing mit Verkettung der Überläufer oder offenes Hashing mit linearem Sondieren weisen im Erwartungsfall auch eine gute Performance im externen Speicher auf.

Übersicht

- Einleitung
- Entwurfsmethoden
- Sortieren
- Auswahlproblem
- Graphalgorithmen
- Spezielle Graphklassen
- Vorrangwarteschlangen
- Suchbäume
- Amortisierte Laufzeitanalyse
- Algorithmen für moderne Hardware
- *Algorithmen für geometrische Probleme*

Übersicht

Algorithmen für geometrische Probleme

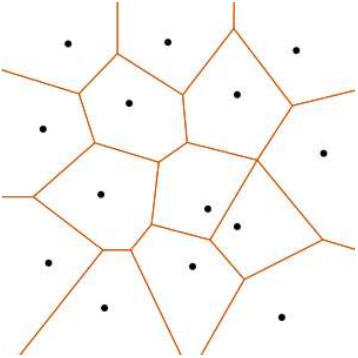
- *Einleitung*
- Scan-Line-Prinzip
- Konvexe Hülle

Motivation

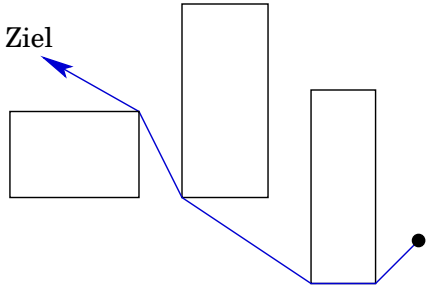
Beispiel: Angenommen, wir kennen die Standorte aller Briefkästen in der Stadt. Welcher Briefkasten ist unserem aktuellen Standort am nächsten?

Um diese Frage beantworten zu können, unterteilen wir die Ebene in sogenannte Voronoi-Regionen:

- Jede Region besitzt ein Zentrum: der Punkt der Punktemenge, der diese Region definiert.
- Eine Region umfasst alle Punkte der Ebene, die näher an dem Zentrum dieser Region liegen, als an irgend einem anderen Zentrum.
- Voronoi-Region: Menge aller Anfragepunkte, für die die obige Antwort dieselbe ist.



Beispiel: Ein Roboter soll einen Brief für uns einwerfen. Wie findet der Roboter sein Ziel, ohne gegen Hindernisse zu laufen?



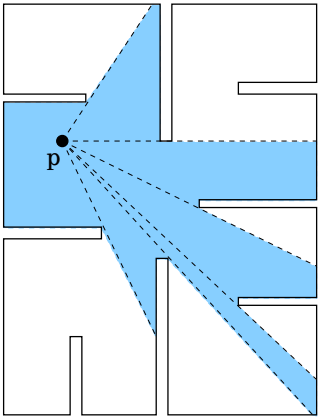
Bewegungsplanung in der Robotik:

- Gegeben: eine Menge von Hindernissen, ein Start- und ein Zielpunkt
- Aufgabe: Finde einen kollisionsfreien kürzesten Weg zum Ziel.

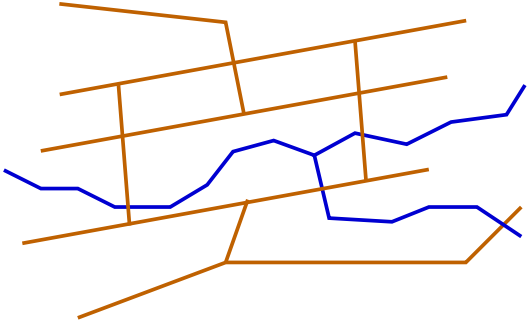
Beispiel: Das Sichtbarkeitspolygon $vis(p)$ eines Punktes p ist ein Objekt des $\mathbb{R}^2$ und ist der Teil eines einfachen Polygons P, der vom Punkt p aus sichtbar ist.

Anwendung zum Beispiel beim Museumsproblem bzw. Art-Gallery-Problem:

Ein Museum, das dargestellt wird durch das Polygon P, soll mit Kameras überwacht werden.
Wie viele Kameras werden benötigt, und wo müssen die Kameras platziert werden, damit jeder Punkt im Inneren von P gesehen wird?



Beispiel: In Geo-Informationssystemen werden Straßen, Flüsse, Kanäle usw. durch Mengen von einzelnen Strecken modelliert und in verschiedenen Ebenen gespeichert. Durch Überlagerung kann man bestimmen, wo Brücken gebaut werden müssen.



Alle Kantenpaare zu testen ist aber viel zu langsam. Wie sehen effiziente Verfahren zur Schnittpunktberechnung aus?

Beispiel: Wie finden wir zu einem gegebenen Anfragepunkt q das Land, in welchem sich der Punkt q befindet?

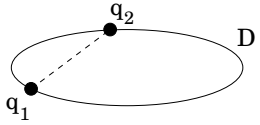
Beispiel: Navigationssysteme müssen zu einem gegebenen Kartenausschnitt effizient alle benötigten Daten bestimmen.

Jedes einzelne Kartenobjekt zu prüfen wäre zu zeitintensiv. Wir benötigen daher eine Datenstruktur zur schnellen Beantwortung von Bereichsanfragen.

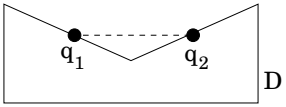
- Ein d-Tupel $(x_1, \ldots, x_d) \in \mathbb{R}^d$ ist ein *Punkt*.
- Für zwei Punkte $p = (p_1, \ldots, p_d)$ und $q = (q_1, \ldots, q_d)$ ist der *euklidische Abstand* definiert als:

$$|pq| = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}$$

- Für $q_1, q_2 \in \mathbb{R}^d$ und $\alpha \in \mathbb{R}$
  - ist $q_1 + \alpha \cdot (q_2 - q_1)$ eine *Linie*, und
  - für $0 \leq \alpha \leq 1$ ist $\overline{q_1 q_2}$ das *Liniensegment* $q_1 + \alpha \cdot (q_2 - q_1)$.
- Eine Menge $D \subseteq \mathbb{R}^d$ heißt *konvex*, falls für alle Punktpaare $q_1, q_2 \in D$ gilt: $\overline{q_1 q_2} \subseteq D$



konvex



nicht konvex

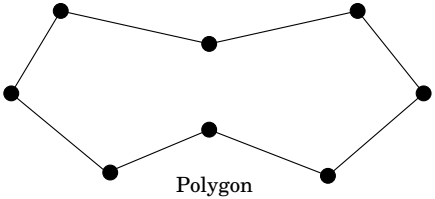
- Eine endliche Menge von Liniensegmenten

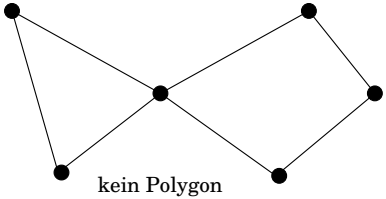
$$P = \{\overline{q_1q_2},\ \overline{q_2q_3},\ \overline{q_3q_4}, \ldots,\ \overline{q_{n-1}q_n}\} \text{ mit } q_i \in \mathbb{R}^2$$

heißt *Polygon*, falls

$$\overline{q_iq_{i+1}} \cap \overline{q_jq_{j+1}} \subseteq \{q_i, q_{i+1}\}, 1 \leq i, j \leq n-1$$

gilt und sich in jedem Punkt maximal 2 Liniensegmente schneiden.





Frage: Wie berechnet man für zwei Liniensegmente $l = (p_1, p_2)$ und $g = (p_3, p_4)$, die jeweils durch ihre Endpunkte $p_i = (x_i, y_i)$ definiert sind, den Schnittpunkt der Liniensegmente?

Antwort: Es gilt $l = p_1 + \alpha \cdot (p_2 - p_1)$ und $g = p_3 + \beta \cdot (p_4 - p_3)$, sodass wir für den Schnittpunkt $l = g$ erhalten:

$$\begin{aligned} p_1.x + \alpha \cdot (p_2.x - p_1.x) &= p_3.x + \beta \cdot (p_4.x - p_3.x) \quad (*) \\ p_1.y + \alpha \cdot (p_2.y - p_1.y) &= p_3.y + \beta \cdot (p_4.y - p_3.y) \quad (**) \end{aligned}$$

Durch Umformen von $(**)$ ergibt sich:

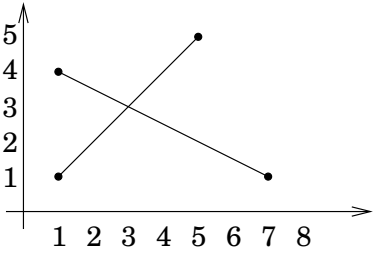
$$\alpha = \frac{(p_3.y - p_1.y) + \beta \cdot (p_4.y - p_3.y)}{p_2.y - p_1.y}$$

Durch Einsetzen in $(*)$ ergibt sich:

$$\beta = \frac{(p_1.x - p_3.x)(p_2.y - p_1.y) + (p_3.y - p_1.y)(p_2.x - p_1.x)}{(p_4.x - p_3.x)(p_2.y - p_1.y) - (p_4.y - p_3.y)(p_2.x - p_1.x)}$$

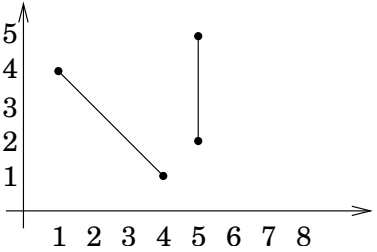
Beispiele zum Segmentschnitt:

$p_1 = (1, 1)$
$p_2 = (5, 5)$
$p_3 = (1, 4)$
$p_4 = (7, 1)$



$\Rightarrow \beta = \frac{1}{3}$

$p_1 = (5, 2)$
$p_2 = (5, 5)$
$p_3 = (1, 4)$
$p_4 = (4, 1)$



$\Rightarrow \beta = \frac{4}{3} > 1$

Dichtestes Punktepaar in der Ebene

gegeben: n Punkte in der Ebene $x_1, x_2, \ldots, x_n \in \mathbb{R}^2$.

gesucht: Ein Punktepaar $x_i, x_j, i \neq j$, dass von allen Paaren den kleinsten Abstand $d(x_i, x_j) = |x_i x_j|$ hat.

Einfacher Algorithmus: Bestimme unter allen $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Paaren das dichteste Punktepaar.

$p_1 := x_1$
$p_2 := x_2$
$min := d(x_1, x_2)$

for $i := 1$ to $n - 1$ do
  for $j := i + 1$ to n do
    if $d(x_i, x_j) < min$
      $min := d(x_i, x_j)$
      $p_1 := x_i$
      $p_2 := x_j$

$\rightarrow$ Laufzeit: $\Theta(n^2)$

Geht es schneller?

Dichtestes Paar einer Zahlenfolge

gegeben: n reelle Zahlen $x_1, x_2, \ldots, x_n \in \mathbb{R}$.

gesucht: Ein Zahlenpaar $x_i, x_j, i \neq j$, dass von allen Paaren den kleinsten Abstand $d(x_i, x_j) = |x_i - x_j|$ hat.

Einfacher Algorithmus: Bestimme unter allen $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Paaren das dichteste Zahlenpaar (wie oben).

$p_1 := x_1$
$p_2 := x_2$
$min := d(x_1, x_2)$

for $i := 1$ to $n - 1$ do
    for $j := i + 1$ to n do
        if $d(x_i, x_j) < min$
            $min := d(x_i, x_j)$
            $p_1 := x_i$
            $p_2 := x_j$

$\rightarrow$ Laufzeit: $\Theta(n^2)$

Geht es schneller?

Dichtestes Paar einer Zahlenfolge

Scan-Line-Verfahren:

$sort(x_1, \ldots, x_n)$
$p_1 := x_1$
$p_2 := x_2$
$min := d(x_1, x_2)$

for $i := 2$ to $n - 1$ do
    if $d(x_i, x_{i+1}) < min$
        $min := d(x_i, x_{i+1})$
        $p_1 := x_i$
        $p_2 := x_{i+1}$

$\rightarrow$ Laufzeit: $\Theta(n \cdot \log(n))$

Kann man diese Idee auch im zweidimensionalen Raum nutzen?

Übersicht

Algorithmen für geometrische Probleme

- Einleitung
- *Scan-Line-Prinzip*
- Konvexe Hülle

Scan-Line-Prinzip

- Lasse eine vertikale Linie, eine sogenannte Scan-Line, von links nach rechts über eine gegebene Menge von Objekten in der Ebene laufen.
  - $\rightarrow$ Damit zerlegt man ein zweidimensionales geometrisches Problem in eine Folge eindimensionaler Probleme.
- Während man die Scan-Line über die Eingabemenge schwenkt, hält man eine Vertikalstruktur L aufrecht, in der man sich alle für das jeweils zu lösende Problem benötigte Daten merkt.
- Die Scan-Line teilt zu jedem Zeitpunkt die gegebene Menge von Objekten in drei disjunkte Teilmengen:
  - die *toten Objekte*, die bereits vollständig von der Scan-Line überstrichen wurden
  - die gerade *aktiven Objekte*, die gegenwärtig von der Scan-Line geschnitten werden
  - die noch *inaktiven Objekte*, die erst künftig von der Scan-Line geschnitten werden

Übersicht

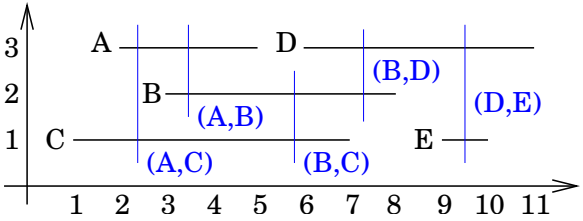
Scan-Line-Prinzip:

- *Sichtbarkeitsproblem*
- Schnittproblem für iso-orientierte Liniensegmente
- Allgemeines Liniensegment-Schnittproblem
- Dichtestes Punktepaar in der Ebene

Motivation: Bei der Kompaktierung höchst-integrierter Schaltkreise müssen Abstandsbedingungen zwischen den relevanten Paaren von Schaltelementen beachtet werden.

- Die Schaltelemente werden abstrahiert durch horizontale Liniensegmente.
- Die relevanten Paare müssen zunächst bestimmt werden.
  → Menge aller Paare, die sich gegenseitig sehen können.

Beispiel:



Sichtbarkeitsproblem

Definition: Zwei Liniensegmente s und s' sind *gegenseitig sichtbar*, wenn es eine vertikale Gerade gibt, die s und s', aber kein anderes Liniensegment zwischen s und s' schneidet.

vorläufig: Alle x-Koordinaten der Anfangs-/Endpunkte sind paarweise verschieden. Später werden wir sehen, wie wir uns von diesen Annahmen frei machen.

Einfacher Algorithmus: Stelle für alle $\binom{n}{2} = \frac{n \cdot (n-1)}{2}$ Paare von Liniensegmenten fest, ob sie gegenseitig sichtbar sind.

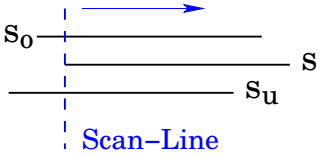
$\rightarrow$ Laufzeit $\Theta(n^2)$, falls der Test auf gegenseitige Sichtbarkeit zweier Liniensegmente in konstanter Zeit erfolgt.

- *Frage:* Wie testet man effizient, ob zwei Liniensegmente gegenseitig sichtbar sind?

Scan-Line-Algorithmus:

$Q :=$ Anfangs-/Endpunkte in aufsteigender x-Reihenfolge
$L := \emptyset$
while $Q \neq \emptyset$ do

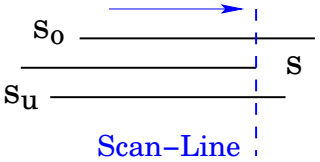
- $p :=$ nächster Punkt aus Q
- falls p ist linker Endpunkt von Segment s



  - füge s in L ein
  - bestimme oberen Nachbarn s_o von s in L
  - bestimme unteren Nachbarn s_u von s in L
  - gib (s, s_o) und (s, s_u) als gegenseitig sichtbar aus

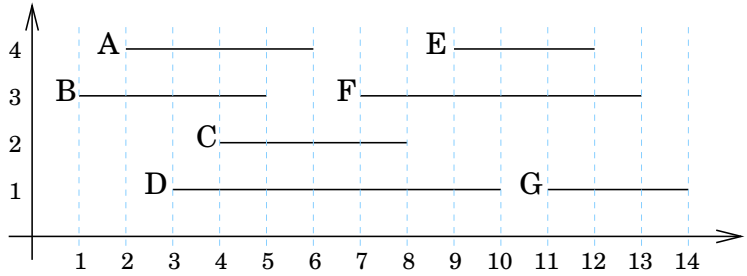
Scan-Line-Algorithmus: (Fortsetzung)

∗ falls p ist rechter Endpunkt von Segment s



· bestimme oberen Nachbarn s_o von s in L
· bestimme unteren Nachbarn s_u von s in L
· entferne s aus L
· gib (s_o, s_u) als gegenseitig sichtbar aus

Gegeben seien folgende horizontale Liniensegmente:



Übung 2.

- *Bestimmen Sie mittels des Scan-Line-Verfahrens die gegenseitig sichtbaren Paare von Liniensegmenten.*
- *Geben Sie an jedem Haltepunkt der Scan-Line die Datenstruktur L und die dortige Ausgabe an.*
- *Kann es vorkommen, dass Paare doppelt ausgegeben werden?*

Implementierung:

- Die Vertikalstruktur L wird als balancierter Suchbaum implementiert, z.B. als Rot/Schwarz- oder AVL-Baum.
- Die Operationen Einfügen, Entfernen und Bestimmen von Nachbarn werden gut unterstützt. $\rightarrow$ logarithmische Zeit
- Frage: Wie viele Paare gegenseitig sichtbarer Segmente werden höchstens ausgegeben?

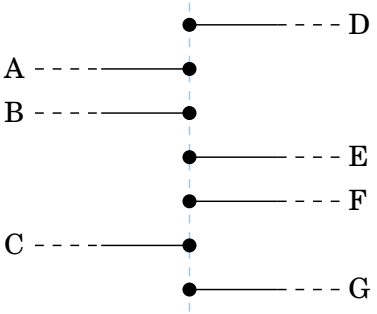
Bemerkung: Es gibt nur $3n - 6$ gegenseitig sichtbare Segmente.

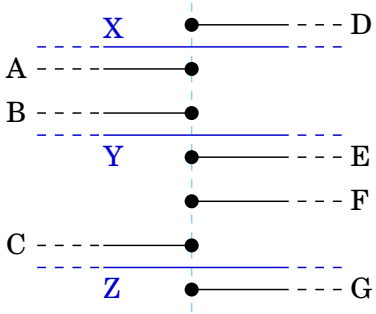
- Stelle die Relation „ist gegenseitig sichtbar“ als planaren Graphen $G = (V, E)$ dar.
  - Jeder Knoten aus V repräsentiert ein Liniensegment,
  - jede Kante aus E stellt ein gegenseitig sichtbares Paar dar.
- Für einen planaren Graphen $G = (V, E)$ gilt: $\mathcal{E} \leq 3 \cdot \mathcal{V} - 6$

$\rightarrow$ Laufzeit $\mathcal{O}(n \cdot \log(n))$, Platz $\mathcal{O}(n)$.

Übung 3. *Welche Änderungen am Algorithmus sind nötig, wenn wir uns von der Annahme befreien wollen, dass alle Anfangs- und Endpunkte verschiedene x-Koordinaten haben müssen?*

Es wären also folgende Situationen möglich:





Ändert sich durch die Änderungen die asymptotische Laufzeit?

Übersicht

Scan-Line-Prinzip:

- Sichtbarkeitsproblem
- *Schnittproblem für iso-orientierte Liniensegmente*
- Allgemeines Liniensegment-Schnittproblem
- Dichtestes Punktepaar in der Ebene

Schnitt iso-orientierter Liniensegmente

Schnittproblem für iso-orientierte Liniensegmente:

gegeben: Eine Menge von n vertikalen und horizontalen Liniensegmenten in der Ebene.

gesucht: Alle Paare sich schneidender Segmente.

vorläufig: Alle x-Koordinaten der Anfangs-/Endpunkte sowie der vertikalen Segmente sind paarweise verschieden.

Idee: Wenn man sich in der Vertikalstruktur L stets die gerade aktiven horizontalen Segmente merkt, und man dann auf ein vertikales Segment s trifft, dann kann s nur mit den gerade aktiven Elementen Schnittpunkte haben.

Schnitt iso-orientierter Liniensegmente

Scan-Line-Algorithmus:

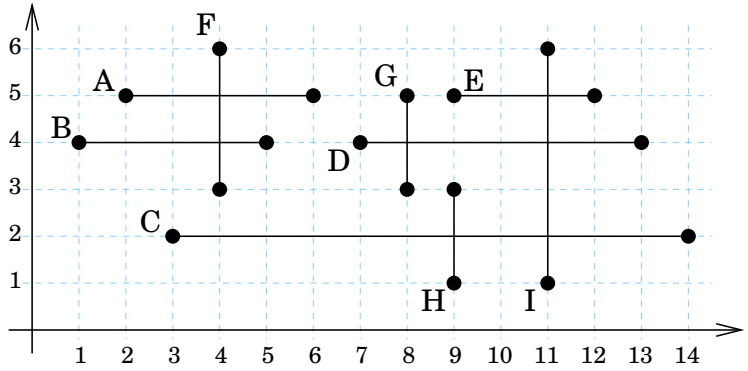
$Q :=$ Anfangs-/Endpunkte horizontaler Segemente und vertikale Segmente in aufsteigender x-Reihenfolge

$L := \emptyset$

while $Q \neq \emptyset$ do

- $p :=$ nächster Punkt aus Q
- falls p ist linker Endpunkt des horizontalen Segments s
  - füge s in L ein
- falls p ist rechter Endpunkt des horizontalen Segments s
  - entferne s aus L
- falls p ist x-Wert eines vertikalen Segments s mit unterem Endpunkt y_u und oberem Endpunkt y_o
  - bestimme alle horizontalen Segmente $t \in L$ mit $y_u \leq y(t) \leq y_o$ und gib (s, t) aus

Übung 4. *Gegeben seien folgende iso-orientierte Liniensegmente:*



- *Bestimmen Sie mittels des Scan-Line-Verfahrens die Schnittpunkte der Liniensegmente.*
- *Geben Sie an jedem Haltepunkt der Scan-Line die Datenstruktur L und die dortige Ausgabe an.*

Implementierung:

- Die Vertikalstruktur L wird als balancierter Blattsuchbaum implementiert, z.B. als Rot/Schwarz- oder AVL-Baum, dessen Blätter verkettet sind.
- Einfügen und Entfernen ist in Zeit $\mathcal{O}(\log(n))$ möglich.
- Bereichsanfragen können in Zeit $\mathcal{O}(\log(n) + r)$ beantwortet werden, wobei r die Anzahl der Elemente in dem angefragten Bereich ist.

$\rightarrow$ Laufzeit: $\mathcal{O}(n \cdot \log(n) + k)$, wobei k die Anzahl der Schnittpunkte insgesamt ist.

Platzbedarf: $\mathcal{O}(n)$

Übung 5. *Welche Änderungen am Algorithmus sind nötig, wenn wir uns von der Annahme befreien wollen, dass alle Anfangs- und Endpunkte sowie die vertikalen Segmente verschiedene x-Koordinaten haben müssen?*

Ändert sich dadurch die asymptotische Laufzeit?

Übersicht

Scan-Line-Prinzip:

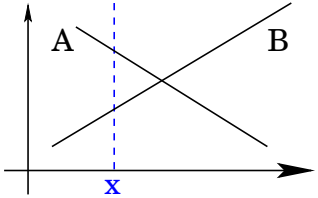
- Sichtbarkeitsproblem
- Schnittproblem für iso-orientierte Liniensegmente
- *Allgemeines Liniensegment-Schnittproblem*
- Dichtestes Punktepaar in der Ebene

Allgemeines Liniensegment-Schnittproblem

gegeben: Eine Menge von n Liniensegmenten.

gesucht: Alle Paare sich schneidender Segmente.

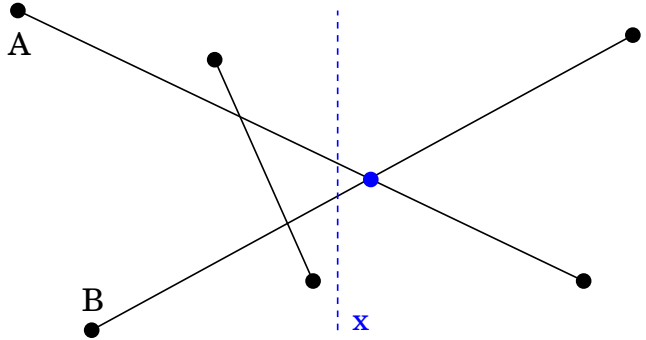
Vereinbarung: *A liegt x-oberhalb von B*, wenn die vertikale Gerade durch x sowohl A als auch B schneidet und der Schnittpunkt von x mit A oberhalb des Schnittpunktes von x mit B liegt. Schreibweise: $A \uparrow_x B$



vorläufig:

- Es gibt keine vertikalen Liniensegmente.
- In jedem Punkt schneiden sich höchstens zwei Segmente.
- Alle Anfangs-/Endpunkte haben paarweise verschiedene x-Koordinaten.

Beobachtung: Wenn sich zwei Liniensegmente A und B schneiden, dann gibt es eine Stelle x links vom Schnittpunkt, sodass A und B in der Ordnung $\uparrow_x$ unmittelbar aufeinanderfolgen.



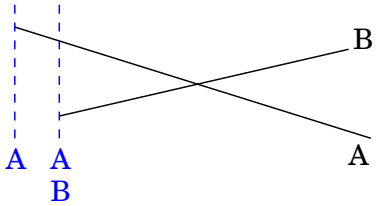
Hier geht die Voraussetzung ein, dass sich in jedem Punkt höchstens zwei Segmente schneiden.

Allgemeines Liniensegment-Schnittproblem

Scan-Line-Algorithmus:

$Q :=$ Anfangs-/Endpunkte in aufsteigender x-Reihenfolge
$L := \emptyset$
while $Q \neq \emptyset$ do

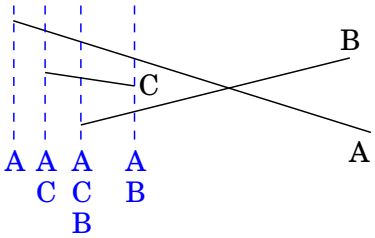
- $p :=$ nächster Punkt aus Q
- falls p ist linker Endpunkt von Segment s



  - füge s entsprechend $\uparrow_{p.x}$ in L ein
  - bestimme oberen Nachbarn s_o von s in L
  - bestimme unteren Nachbarn s_u von s in L
  - füge ggf. $s \cap s_o$ bzw. $s \cap s_u$ in Q ein

Scan-Line-Algorithmus: (Fortsetzung)

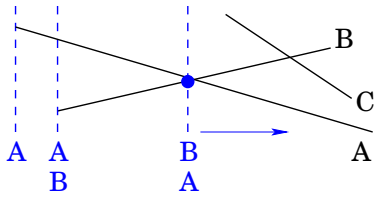
∗ falls p ist rechter Endpunkt von Segment s



· bestimme oberen Nachbarn s_o von s in L
· bestimme unteren Nachbarn s_u von s in L
· entferne s aus L
· füge ggf. Schnittpunkt $s_o \cap s_u$ in Q ein

Allgemeines Liniensegment-Schnittproblem

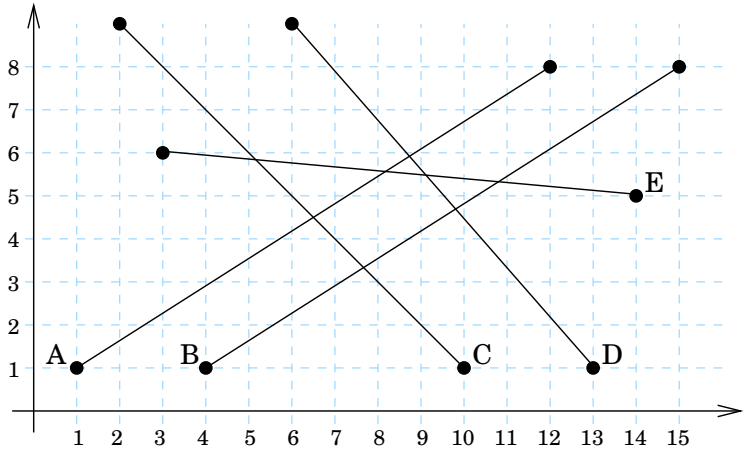
Wenn die Scan-Line einen Schnittpunkt zweier Segmente A und B passiert, dann wechseln A und B ihren Platz in L, damit die lokale Ordnung der Vertikalstruktur L korrekt bleibt.



Scan-Line-Algorithmus: (Fortsetzung)

- falls p Schnittpunkt von s und t ist mit $s \uparrow_{p.x} t$
  - gib (s, t) mit Schnittpunkt p aus
  - vertausche s und t in L
  - bestimme oberen Nachbarn t_o von t in L
  - füge ggf. Schnittpunkt $t \cap t_o$ in Q ein
  - bestimme unteren Nachbarn s_u von s in L
  - füge ggf. Schnittpunkt $s \cap s_u$ in Q ein

Übung 6. *Gegeben seien folgende Liniensegmente:*



- *Bestimmen Sie mittels des Scan-Line-Verfahrens alle Schnittpunkte der Segmente.*
- *Geben Sie an den Haltepunkten der Scan-Line die Struktur L und die Ausgabe an.*

Implementierung:

- Q wird als balancierter Suchbaum implementiert:
  - Füge Schnittpunkt S nur ein, falls noch nicht vorhanden.
  - Suchen, Einfügen, Entfernen und Bestimmen des kleinsten Wertes werden gut unterstützt. $\rightarrow$ logarithmische Laufzeit
- L wird ebenfalls als balancierter Suchbaum implementiert:
  - Einfügen, Entfernen und Bestimmen von direkten Nachbarn werden gut unterstützt. $\rightarrow$ logarithmische Laufzeit
- *Platz:* $\mathcal{O}(n+k)$, wobei k die Anzahl der Schnittpunkte ist.
- Die Schleife wird für k Schnittpunkte höchstens $(2n+k)$-mal durchlaufen und wir erhalten als *Laufzeit:* $\mathcal{O}((n+k)\cdot\log(n))$

Frage: Wenn der Platzbedarf $\mathcal{O}(n+k)$ ist, müsste die Laufzeit dann nicht $\mathcal{O}((n+k)\cdot\log(n+k))$ sein?

Allgemeines Liniensegment-Schnittproblem

Bemerkung:

- Die Komplexitäten sind nur akzeptabel für kleine Wert von k.
- Es gibt Verfahren[6] mit Laufzeit $\mathcal{O}(n \cdot \log(n) + k)$.

Platzersparnis: Speichere nicht alle Schnittpunkte.

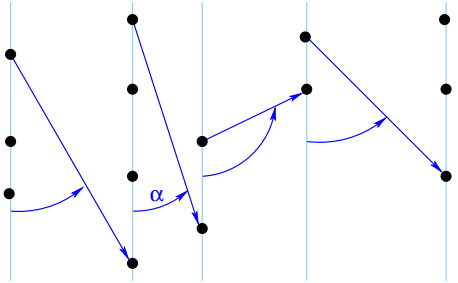
- Wir nehmen für jedes aktive Liniensegment s nur den am weitesten links liegenden Schnittpunkt in Q auf.
- Wird ein weiter links liegender Schnittpunkt mit Segment s gefunden, so wird dieser eingefügt und der alte entfernt.
- Um für jedes aktive Segment s leicht feststellen zu können, ob schon ein Schnittpunkt in Q enthalten ist, an dem s beteiligt ist, vermerke zu s einen Zeiger auf den Schnittpunkt.

$\rightarrow$ *Platz:* $\mathcal{O}(n)$

[6] Chazelle, Edelsbrunner: An optimal algorithm for intersecting line segments in the plane. J ACM, 1992.

Um die vereinfachenden Annahmen fallen lassen zu können, sind bspw. folgende Änderungen am Algorithmus nötig:

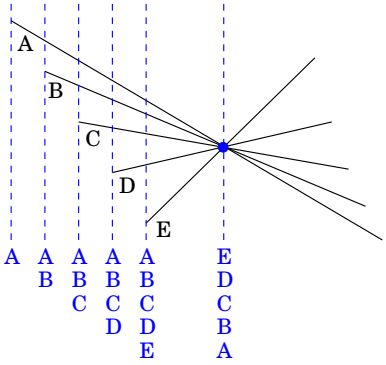
1. Anfangs- und Endpunkte haben gleiche x-Koordinaten.
   - Transformation des Koordinatensystems: Einteilung der Punkte in Gruppen mit gleicher x-Koordinate.



   - Jeweils vom oberen Punkt einer Gruppe zum unteren Punkt der benachbarten rechten Gruppe eine Gerade legen. Sei α der kleinste Winkel einer solchen Geraden mit der y-Achse.
   - Drehe Koordinatensystem um $\alpha/2$ entgegen den Uhrzeigersinn.

2. Schnittpunkte können aber noch mit Anfangs- oder Endpunkten zusammen fallen:
   - Bearbeite zunächst die linken Endpunkte, dann die Schnittpunkte und danach die rechten Endpunkte.

3. In einem Punkt schneiden sich mehr als zwei Segmente.
   - Verfahre wie bisher und teste jedes neu entdeckte Segment auf Schnitt mit seinen beiden Nachbarn.
   - Ordne die Schnittereignisse in Q lexikographisch an.
   - Berichte den mehrfachen Schnitt und invertiere die Reihenfolge der beteiligten Segmente en bloc.



Ändert sich durch diese Änderungen die asymptotische Laufzeit?

Übersicht

Scan-Line-Prinzip:

- Sichtbarkeitsproblem
- Schnittproblem für iso-orientierte Liniensegmente
- Allgemeines Liniensegment-Schnittproblem
- *Dichtestes Punktepaar in der Ebene*

Dichtestes Punktepaar in der Ebene

Kommen wir zurück zu unserem anfänglichen Problem.

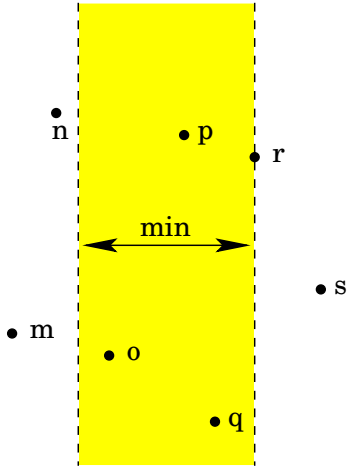
Wenn r mit einem Punkt p links von r ein Paar bilden soll, dann muss p im Innern des senkrechten Streifens der Breite *min* liegen.

Wir merken uns in L die Punkte innerhalb des Streifens.

Die Datenstruktur L muss aktualisiert werden, wenn

- der linke Streifenrand über einen Punkt hinweg wandert.

$\rightarrow$ Der betroffene Punkt wird aus L entfernt.



- der rechte Streifenrand (Scan-Line) auf einen neuen Punkt stößt.

→ Der neue Punkt muss in L aufgenommen werden und es ist zu testen, ob der neue Punkt mit einem der Punkte innerhalb des Streifens einen Abstand kleiner als *min* hat.

falls ja: *min* und die Streifenbreite müssen auf den neuen Wert verringert werden. Das wiederum kann eine Folge von Ereignissen des ersten Typs bedeuten.

oben: Der Streifen schrumpft auf die Breite $d(p, r)$, wenn die Scan-Line Punkt r erreicht. Der linke Streifenrand springt dann über Punkt o hinweg.

Punkte betreten und verlassen den Streifen in der Reihenfolge von links nach rechts:

- Wir sortieren daher die Punkte nach aufsteigenden x-Koordinaten und speichern sie in einem Array.
- Wir testen dabei, ob bei Punkten mit gleichem x-Wert auch die y-Koordinate übereinstimmt. (falls ja $\rightarrow$ terminiere)

Wir arbeiten mit zwei Positionen l und r:

- $Q[l]$: der am weitesten links liegende Punkt im senkrechten Streifen
- $Q[r]$: nächster Punkt, auf den die Scan-Line treffen wird

$\rightarrow$ $Q[l]$ verlässt den Streifen, wenn $Q[l].x + min \leq Q[r].x$ gilt.

Dichtestes Punktepaar in der Ebene

Scan-Line-Algorithmus:

$Q :=$ Folge der n Punkte in aufsteigender x-Reihenfolge
$L := \{p_1, p_2\}$
$min := d(p_1, p_2)$
$l := 1$
$r := 3$

while $r \leq n$ do
- if $Q[l].x + min \leq Q[r].x$ then
  - entferne $Q[l]$ aus L
  - $l := l + 1$
- else
  - $min := MinDist(L, Q[r], min)$
  - füge $Q[r]$ in L ein
  - $r := r + 1$

Dichtestes Punktepaar in der Ebene

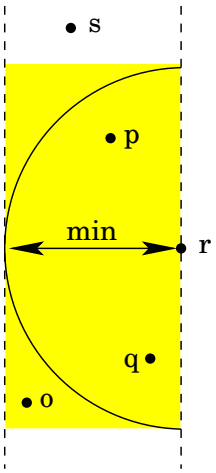
MinDist(L, r, min) liefert die minimale Distanz vom neuen Punkt r auf der Scan-Line zu allen übrigen Punkten im senkrechten Streifen L, oder ggf. den bisherigen Wert *min*.

Inspiziere nur diejenigen Punkte im Streifen, deren y-Koordinate um höchstens *min* oberhalb oder unterhalb von $r.y$ liegen.

L muss Einfügen und Entfernen von Punkten sowie Bereichsanfragen unterstützen:

→ nach y-Koordinaten geordneter, balancierter Suchbaum mit doppelt verketteten Blättern

Platz: $\mathcal{O}(n)$ (es werden nur Punkte gespeichert)
Zeit: ???



Zeit:

$$\mathcal{O}(n \cdot \log(n) + \sum_{i=1}^{n} k_i)$$

denn:

- Die while-Schleife wird n-mal durchlaufen.
- Einfügen in bzw. entfernen aus L erfolgt in Zeit $\mathcal{O}(\log(n))$.
- Der i-te Aufruf von *MinDist* kostet Zeit $\mathcal{O}(\log(n) + k_i)$, dabei bezeichnet k_i die Größe der Antwort der i-ten Bereichsanfrage.

Abschätzung:

- naiv: Für $k_i \leq i$ ergibt sich als Laufzeit $\mathcal{O}(n^2)$.
- bessere Abschätzung ???

Dichtestes Punktepaar in der Ebene

Idee für bessere Abschätzung:

- Alle Punkte links der Scan-Line, insbesondere die im Rechteck, haben einen Abstand von mindestens *min* voneinander.

$\rightarrow$ Es passen nicht allzuviele Punkte ins Rechteck hinein!

Lemma: Sei $M > 0$ und P eine Menge von Punkten in der Ebene, von denen je zwei mindestens den Abstand M voneinander haben. Dann enthält ein Rechteck mit Kantenlängen M und $2M$ höchstens 10 Punkte aus P.

Damit gilt $k_i \leq 10$ für alle i und wir erhalten als Laufzeit des Verfahrens: $\mathcal{O}(n \cdot \log(n))$

Dichtestes Punktepaar in der Ebene

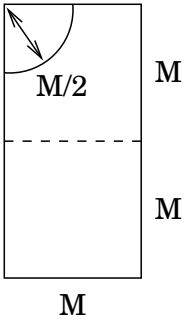
Beweis: Nach Voraussetzung gilt: Die Kreisumgebungen $U_{M/2}(p)$ der Punkte $p \in P$ sind paarweise disjunkt.

Liegt der Punkt p im Rechteck R, so ist mindestens $\frac{1}{4}$ der offenen Kreisscheibe in R enthalten.

Durch Flächenberechnung ergibt sich, dass das Rechteck R höchstens

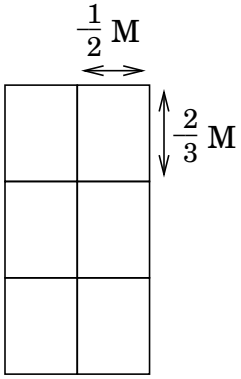
$$\frac{\text{Fläche}(R)}{\text{Fläche Viertelkreis}} = \frac{2M^2}{\frac{1}{4}\pi(\frac{M}{2})^2} = \frac{32}{\pi} < 11$$

viele Punkte aus P enthalten kann!



Bemerkung: Obiges Lemma ist auch für 6 Punkte korrekt!

Teile das Rechteck R in sechs gleich große Rechtecke mit Kantenlänge $\frac{2}{3}M$ und $\frac{1}{2}M$ ein.



Zwei Punkte in einem der Rechtecke können höchstens so weit voneinander entfernt sein, wie seine Diagonale lang ist, also höchstens

$$\sqrt{\left(\frac{2}{3}M\right)^2 + \left(\frac{1}{2}M\right)^2} \;=\; \frac{5}{6}M \;<\; M.$$

Also kann in jedem Rechteck höchstens ein Kreismittelpunkt sein.

Übersicht

Algorithmen für geometrische Probleme

- Einleitung
- Scan-Line-Prinzip
- *Konvexe Hülle*

Definition:

- Eine Menge $S \subseteq \mathbb{R}^2$ heißt konvex, wenn für je zwei Punkte $p, q \in S$ auch $\overline{pq} \subseteq S$ gilt.
- Die konvexe Hülle $CH(S)$ von S ist die kleinste konvexe Menge, die S enthält.

Anschaulich:

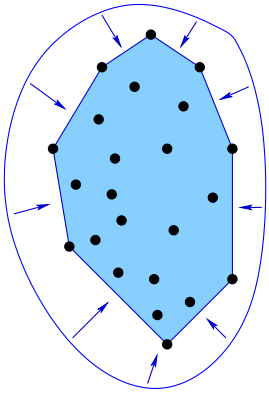
- Lege ein großes Gummiband um alle Punkte und lass es los!

$\rightarrow$ hilft algorithmisch leider gar nicht

In der Mathematik:

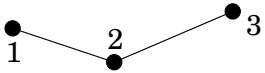
- Definiere $CH(S) := \bigcap_{C \supseteq S : C \text{ konvex}} C$.

$\rightarrow$ hilft leider auch nicht

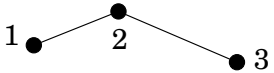


Motivation: Die konvexe Hülle hat deutlich weniger Punkte als das Original, wodurch sich Rechenzeit sparen lässt.

- Kollisionstest bei Computerspielen:
  - Zunächst mit den konvexen Hüllen der Objekte auf Kollision testen.
  - Wenn zwei Hüllen sich nicht berühren, dann berühren sich auch die Ausgangsformen nicht.
- Sichtbare Flächen beim Rendern:
  - Zunächst die Umgebung darstellen, danach die konvexen Hüllen der Figuren.
  - Wenn die Hülle einer Person nicht sichtbar ist, rendern wir die komplexe Figur nicht.



Linksdrehung



Rechtsdrehung

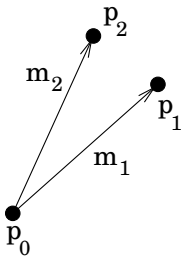
Algorithmus:

- Suche einen Punkt p mit kleinster y-Koordinate. Falls es mehrere gibt, wähle den mit kleinster x-Koordinate.
- Sortiere alle Punkte aufsteigend nach Winkeln zu einer gedachten Horizontalen mit Anfang p.
- $v := p$
- solange $next(v) \neq p$ tue
  falls $[v, next(v), next(next(v))]$ ist Linksdrehung dann
    - $v := next(v)$

  sonst
    - lösche $next(v)$
    - $v := pred(v)$

gegeben: Drei Punkte $p_0, p_1, p_2 \in \mathbb{R}^2$.

Frage: Ist der Weg p_0, p_1, p_2 eine Linksdrehung?



$$m_1 = \frac{dy_1}{dx_1} \qquad m_2 = \frac{dy_2}{dx_2}$$

$$\begin{aligned} dx_1 &= p_1.x - p_0.x & \qquad dx_2 &= p_2.x - p_0.x \\ dy_1 &= p_1.y - p_0.y & \qquad dy_2 &= p_2.y - p_0.y \end{aligned}$$

$\longrightarrow$ Linksdrehung, wenn $m_1 < m_2$

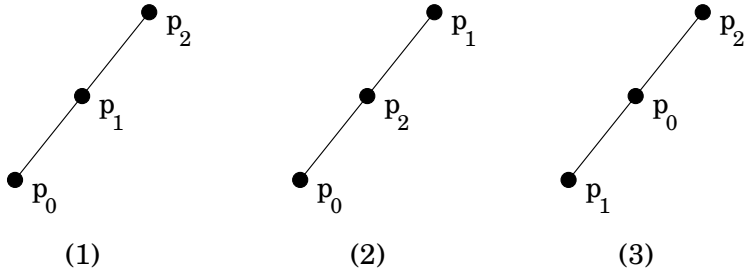
Problem, falls $dx_1 = 0$ oder $dx_2 = 0$ ist!
Multipliziere beide Seiten mit $dx_1 \cdot dx_2$:

$$m_1 < m_2 \iff dy_1 \cdot dx_2 < dy_2 \cdot dx_1$$

oder anders gesagt:

$$\begin{vmatrix} p_0.x & p_0.y & 1 \\ p_1.x & p_1.y & 1 \\ p_2.x & p_2.y & 1 \end{vmatrix} > 0$$

Was ist, wenn die drei Punkte auf einer Geraden liegen, also $m_1 = m_2$?

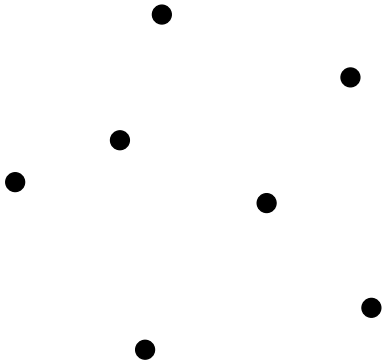


(1) o.k. für Graham-Scan

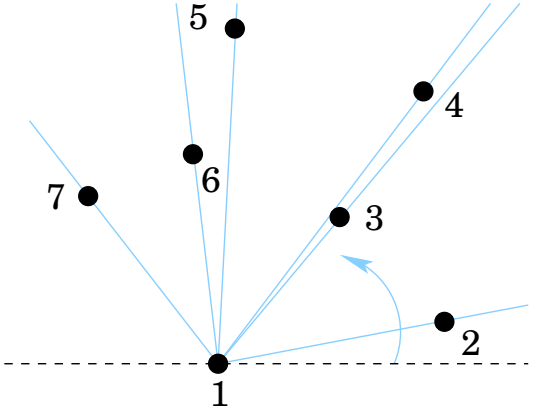
(2) $|p_0p_1| \geq |p_0p_2| \iff dx_1^2 + dy_1^2 \geq dx_2^2 + dy_2^2$

(3) $dx_1 \cdot dx_2 < 0$ oder $dy_1 \cdot dy_2 < 0$

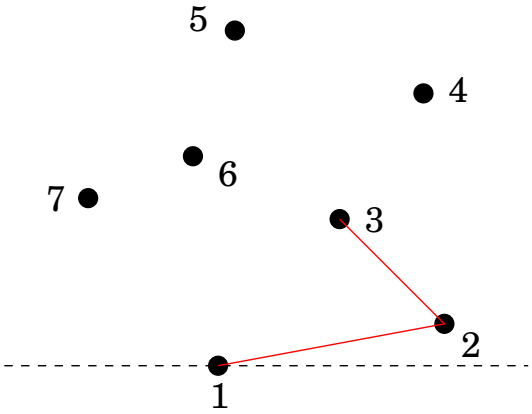
Graham Scan



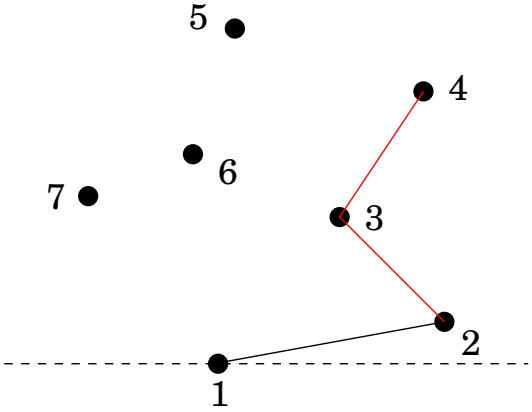
Graham Scan



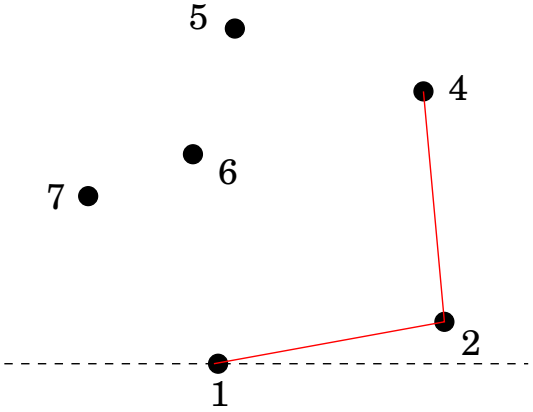
Graham Scan



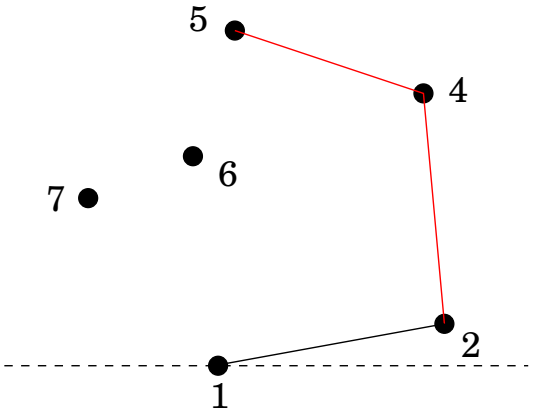
Graham Scan



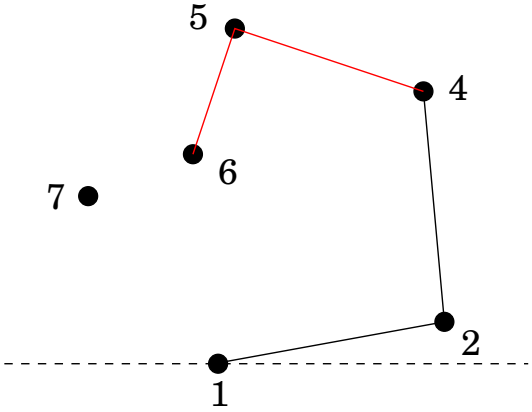
Graham Scan



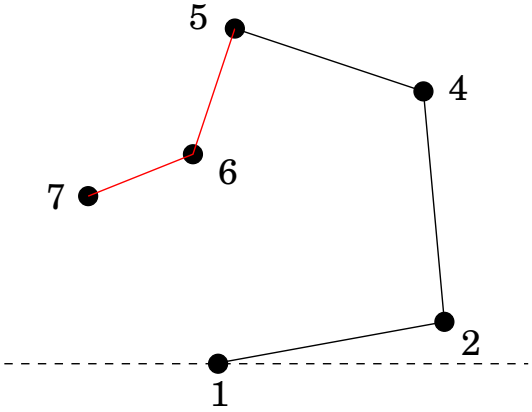
Graham Scan



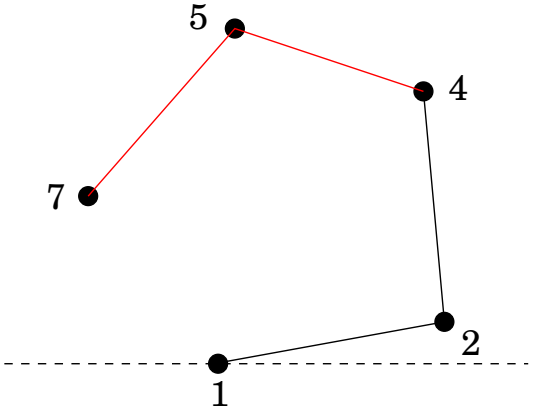
Graham Scan



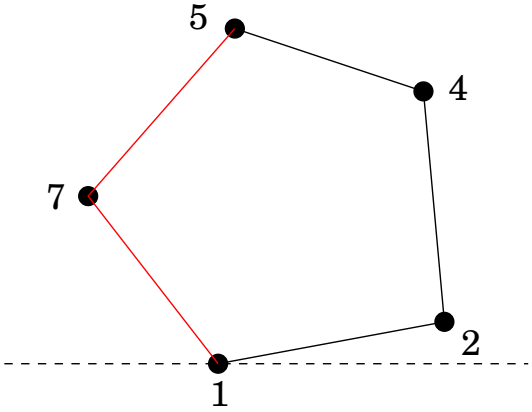
Graham Scan



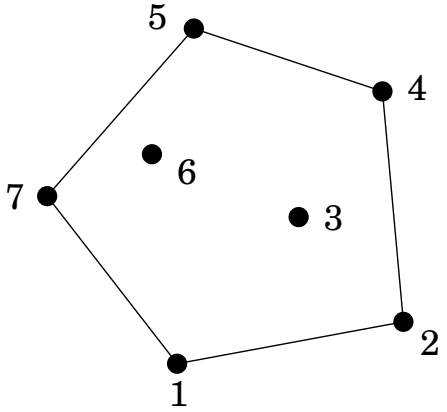
Graham Scan



Graham Scan



Graham Scan



Graham Scan

Implementierung:

- Die Punkte werden in einer doppelt verketteten Liste mit *next*- und *pred*-Zeigern gespeichert.

Komplexität:

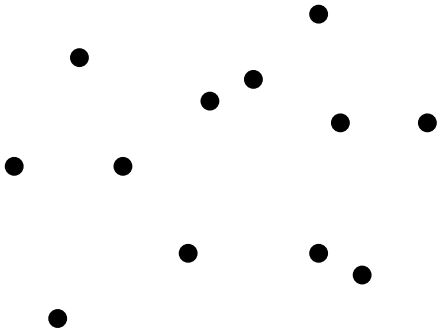
- Laufzeit: $\mathcal{O}(n \cdot \log(n))$
- Platz: $\mathcal{O}(n)$

Package Wrapping

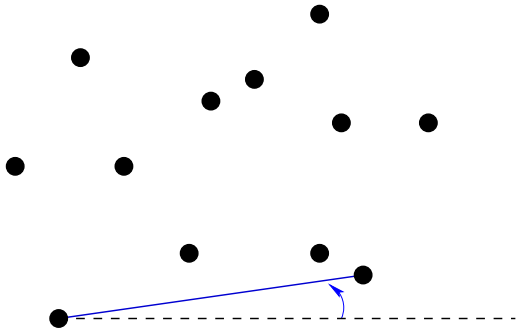
auch Jarvis' March genannt:

* Suche einen Punkt p mit kleinster y-Koordinate. Falls mehrere existieren: Wähle den mit kleinster x-Koordinate.
* $p' := p$
* solange (p' ist kein Punkt mit größter y-Koordinate) tue
  - Suche p'', sodass $\overline{p'p''}$ den kleinsten Winkel zu $\overline{p'q}$ hat, wobei $y(q) = y(p')$ und $x(q) > x(p')$ gilt.
  - füge p'' in $CH(S)$ ein
  - $p' := p''$
* solange ($p' \neq p$) tue
  - Suche p'', sodass $\overline{p'p''}$ den kleinsten Winkel zu $\overline{p'q}$ hat, wobei $y(q) = y(p')$ und $x(q) < x(p')$ gilt.
  - füge p'' in $CH(S)$ ein
  - $p' := p''$

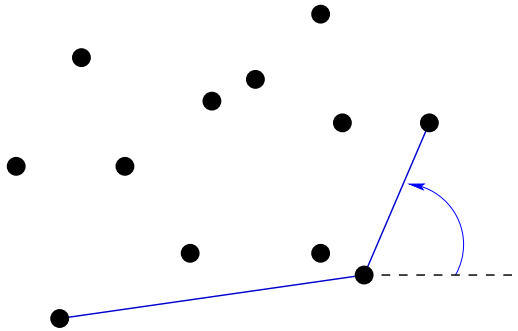
Package Wrapping



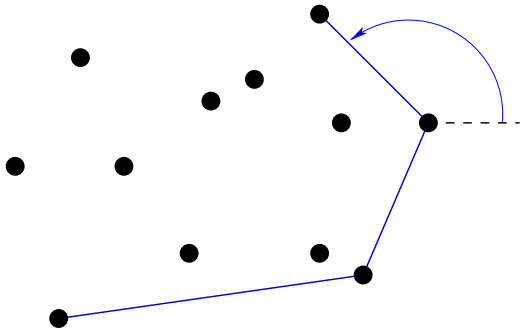
Package Wrapping



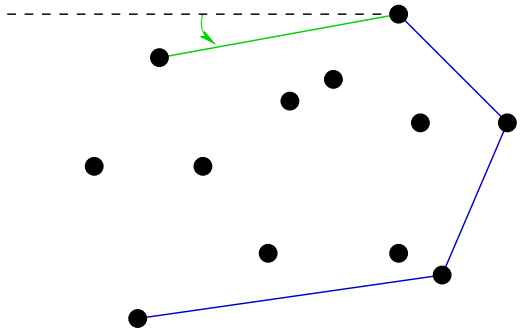
Package Wrapping



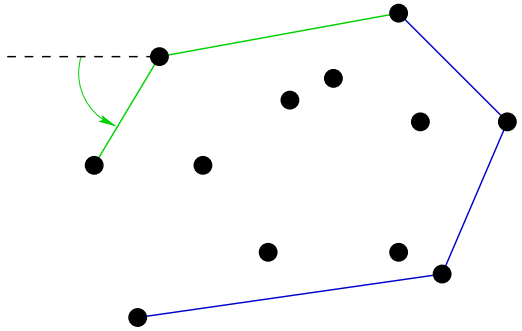
Package Wrapping



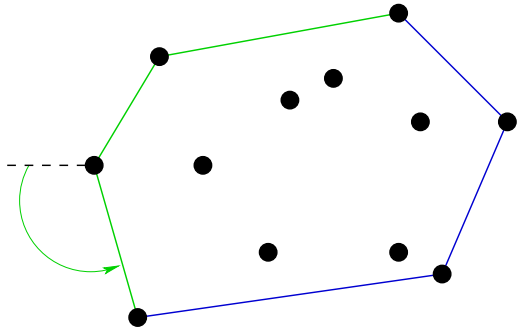
Package Wrapping



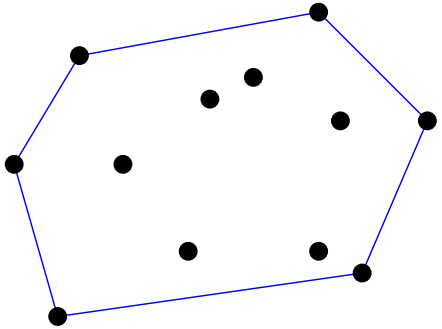
Package Wrapping



Package Wrapping



Package Wrapping



Implementierung:

- Durch die getrennte Berechnung des linken und rechten Teils der konvexen Hülle ist es möglich, die kleinsten Winkel zu berechnen, ohne trigonometrische Funktionen zu verwenden. Dabei wird auf die Idee der Linksdrehung zurück gegriffen.

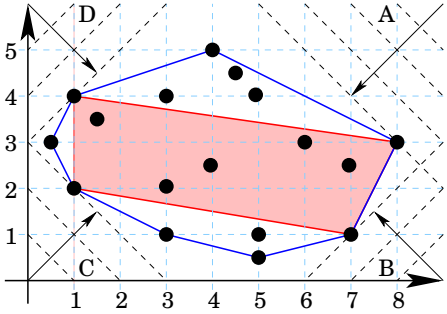
Komplexität:

- Laufzeit: $\mathcal{O}(|CH| \cdot n)$
  $\rightarrow$ Wenn alle Punkte auf der konvexen Hülle liegen: $\mathcal{O}(n^2)$
- Platz: $\mathcal{O}(n)$

Interior Elimination

Package Wrapping bzw. Jarvis' March kann beschleunigt werden, wenn zuerst einige der Punkte, die sicher nicht zur konvexen Hülle gehören, aus der Punktemenge P entfernt werden.

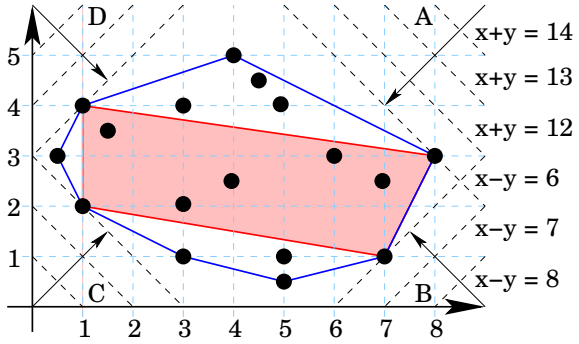
- Bei gleichverteilten x,y-Werten bleiben im Mittel nur $\mathcal{O}(\sqrt{n})$ Punkte erhalten.
- Im worst-case ergibt sich keine Verbesserung.



Fragen:

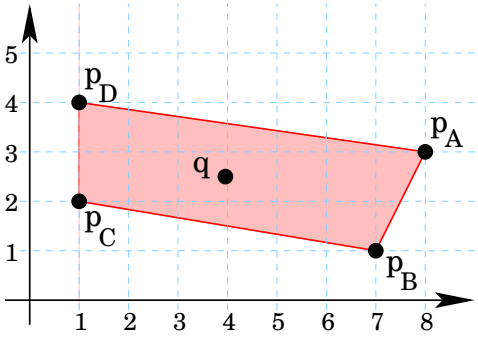
- Wie findet man effizient die 4 Punkte auf der konvexen Hülle?
- Wann liegt ein Punkt innerhalb des Vierecks?
- Wie entfernt man effizient die Punkte innerhalb des Vierecks aus der Punktemenge P?

Wie findet man effizient die 4 Punkte auf der konvexen Hülle?



- für p_A gilt: $p_A.x + p_A.y \geq q.x + q.y \ \forall q \in P$
- für p_B gilt: $p_B.x - p_B.y \geq q.x - q.y \ \forall q \in P$
- für p_C gilt: $p_C.x + p_C.y \leq q.x + q.y \ \forall q \in P$
- für p_D gilt: $p_D.x - p_D.y \leq q.x - q.y \ \forall q \in P$

Wann liegt ein Punkt q innerhalb des Vierecks?



- p_A, p_B, q ist Rechtsdrehung
- p_B, p_C, q ist Rechtsdrehung
- p_C, p_D, q ist Rechtsdrehung
- p_D, p_A, q ist Rechtsdrehung

Interior Elimination

Wie entfernt man effizient die Punkte innerhalb des Vierecks aus der Punktemenge P?

- Wir gehen davon aus, dass die Punkte in einem Array gespeichert sind.
- Algorithmus:

  $i := 1$
  $last := n$
  solange $i < last$ tue
  falls p_i innerhalb des Rechtecks ist, dann
  · swap(p_i, p_{last})
  · $last := last - 1$
  sonst
  · $i := i + 1$

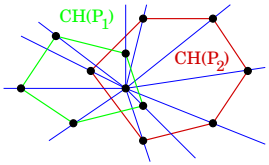
- *Divide:* Teile $P = \{p_1, \ldots, p_n\}$ auf in zwei etwa gleich große Mengen $P_1 = \{p_1, \ldots, p_{\lceil n/2 \rceil}\}$ und $P_2 = \{p_{\lceil n/2 \rceil + 1}, \ldots, p_n\}$.
- *Conquer:* Berechne $CH(P_1)$ und berechne $CH(P_2)$ rekursiv.
- *Merge:* Finde einen Punkt p innerhalb von $CH(P_1)$.
  - falls p innerhalb von $CH(P_2)$ liegt:

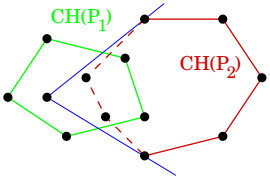
    Mische $CH(P_1)$ und $CH(P_2)$ zu einer Sortierung nach Winkeln bzgl. eines Segments mit Endpunkt p.

  - sonst liegt p außerhalb von $CH(P_2)$:

    Bestimme den Keil von p und q_1, q_2 aus $CH(P_2)$, sodass alle Punkte aus P_2 in diesem Keil liegen. Mische $CH(P_1)$ und die äußere Kontur von $CH(P_2)$.

  - Wende auf die gemischte Liste Graham Scan an.

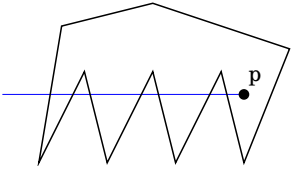




Anmerkungen:

- Mischen der beiden konvexen Hüllen in Zeit $\mathcal{O}(|CH(P_1)| + |CH(P_2)|) = \mathcal{O}(n)$.
- Anwendung von Graham Scan in Zeit $\mathcal{O}(n)$, da Sortieren nach Winkeln entfällt.
- Die Entscheidung, ob ein Punkt innerhalb oder außerhalb eines Polygons liegt, kann in Zeit $\mathcal{O}(n)$ getroffen werden:
  - Anzahl Schnittpunkte ungerade: p liegt innerhalb
  - sonst: p liegt außerhalb
  - es sind einige Sonderfälle zu beachten:
    - Linie schneidet Eckpunkt oder
    - eine Kante liegt auf der Linie

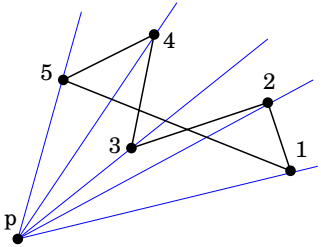
  $\rightarrow$ Punkt-in-Polygon-Test nach Jordan



- Komplexität:
  - Laufzeit: $T(n) = 2 \cdot T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n \cdot \log(n))$
  - Platz: $\mathcal{O}(n)$

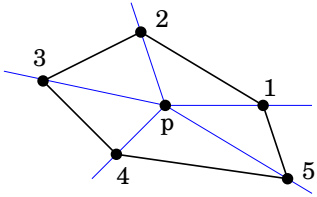
`https://de.wikipedia.org/wiki/Punkt-in-Polygon-Test_nach_Jordan`

Es ist wichtig, dass der Punkt p, mit dessen Hilfe die Sortierung der Winkel erfolgt, innerhalb, oder zumindest auf der zu bestimmenden konvexen Hülle liegt.



p liegt außerhalb der zu berechnenden konvexen Hülle:

Es ergibt sich ein geschlossener Weg, bei dem sich einige Kanten schneiden. Ungeeignet für Graham Scan.



p liegt innerhalb der zu berechnenden konvexen Hülle:

Es ergibt sich ein geschlossener Weg, bei dem sich keine Kanten schneiden. Geeignet für Graham Scan.