Algorithm Engineering von Anfang an: Sortieren und grundlegende Datenstrukturen

Peter Sanders

Institut für Theoretische Informatik, Algorithmik II

mit

Rene Beier, Stefan Burkhardt, Roman Dementiev, Sebastian Egner,

Dimitris Fotakis, Stefan Funke, David Hutchinson, Juha Kärkkäinen,

Irit Katriel, Lutz Kettner, Domagoj Matijevic, Kurt Mehlhorn,

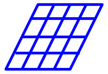
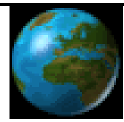
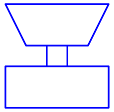
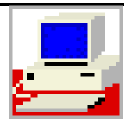
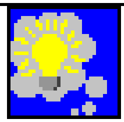
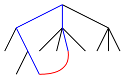
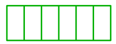
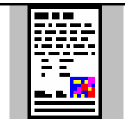
Jens Mehnert, Uli Meyer, Thomas Novak, Rasmus Pagh,

Dominik Schultes, Jop Sibeyn, Naveen Sivadasan, Paul Spirakis,

Jesper Träff, Jeff Vitter, Berthold Vöcking, Sebastian Winkel

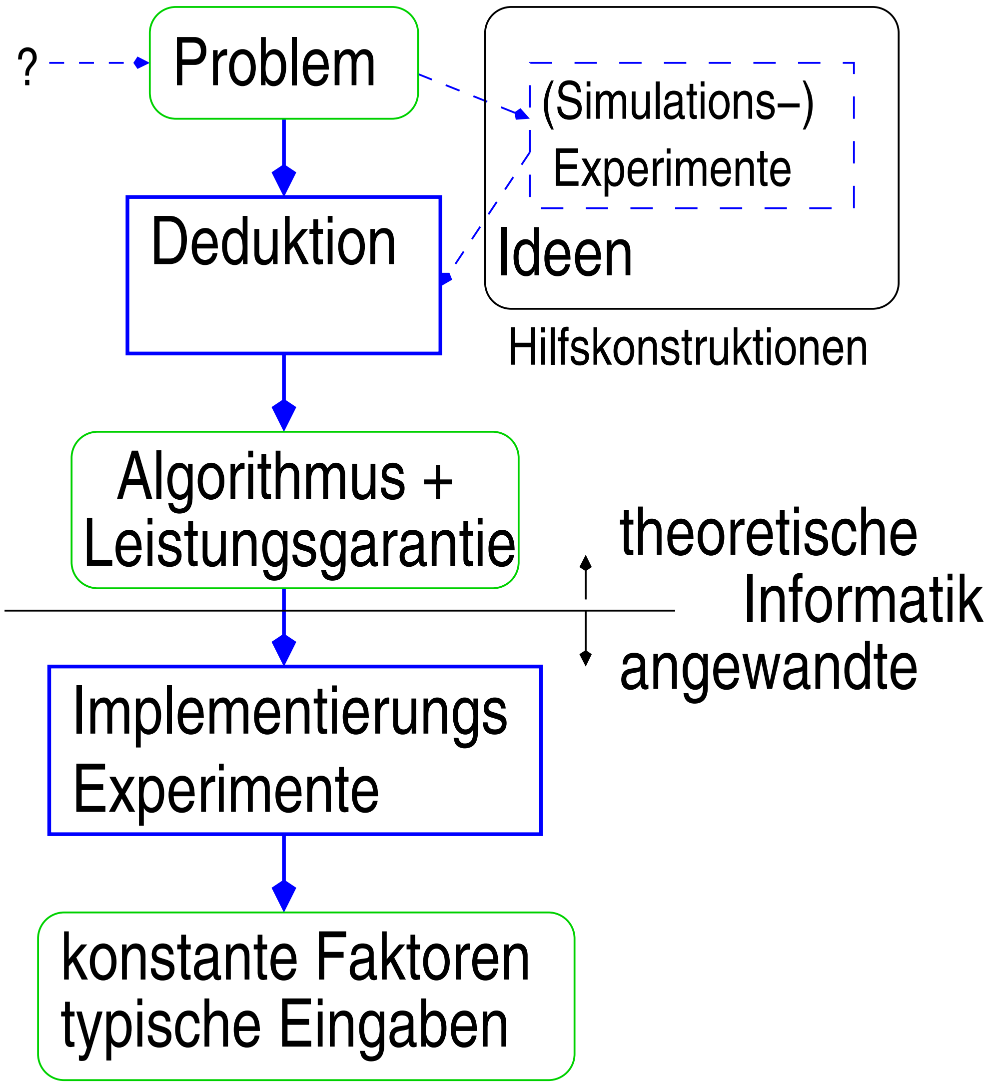
Langversion: Vorlesung WS 04/05

Theoretische $\neq$ Praktische Algorithmik?

Theorie	$\longleftrightarrow$	**Praxis**
einfach 	**Problemmodell**	 komplex
einfach 	**Maschinenmodell**	 real
komplex 	**Algorithmen**	FOR einfach
fortgeschr. 	**Datenstrukturen**	 Felder,...
worst case max	**Komplexitätsmaß**	 Eingaben
asympt. $\mathcal{O}(\cdot)$	**Effizienz**	42% Konstanten

Traditionelle Theoretikersicht?

Ein Wasserfallmodell der Algorithmik



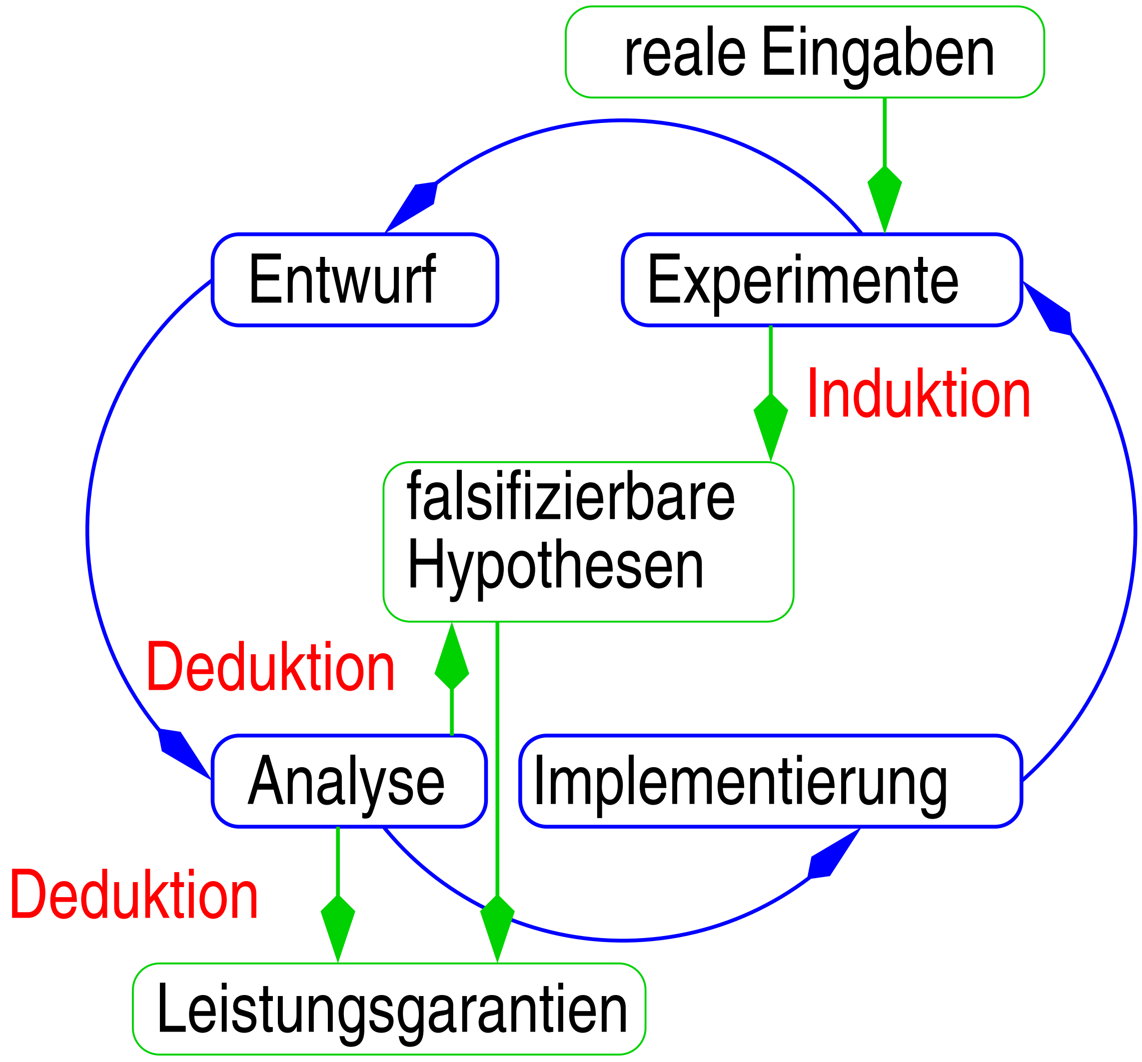
Algorithm Engineering

Scientific Method

Das Modell

der Naturwissenschaften

[Popper 1934]



Ziele

- Entstandene Lücken überbrücken
- Schneller Transfer algorithmischer Ergebnisse in Anwendungen
- Theorie trifft Technologie — Maschinenmodelle müssen der technologischen Entwicklung gerecht werden

Grundlegende Algorithmen

hier

bearbeitet

- Listen, Arrays, Stacks, Queues
- Sortieren
- Prioritätslisten
- Sortierte Listen/Suchbäume
- Hashtabellen
- Graphenalgorithmen
  - Graphtraversierung (DFS, BFS)
  - Kürzeste Wege
  - Minimale Spannbäume
  - Flussprobleme
- Strings

Sortieren (z.B. zur Indexkonstruktion)

Quicksort: [Hoare 61]

Teile in Elemente $\leq$ und $>$ Pivotelement; Rekursion.

Irgendwie immer noch der beste Algorithmus

$\boxed{\mathcal{O}(\cdot)}$

- $n \log n + \mathcal{O}(n)$ Vergleiche (erwartet, gute Pivots)

FOR	42%

- Einfach, wenig Instruktionen
- (einigermaßen) cache-effizient

Geht es trotzdem besser?

Scheinbar nicht: Vergleiche $\rightsquigarrow$ schwer vorhersagbare Sprünge

Cache ist zweitrangig oder gar irrelevant

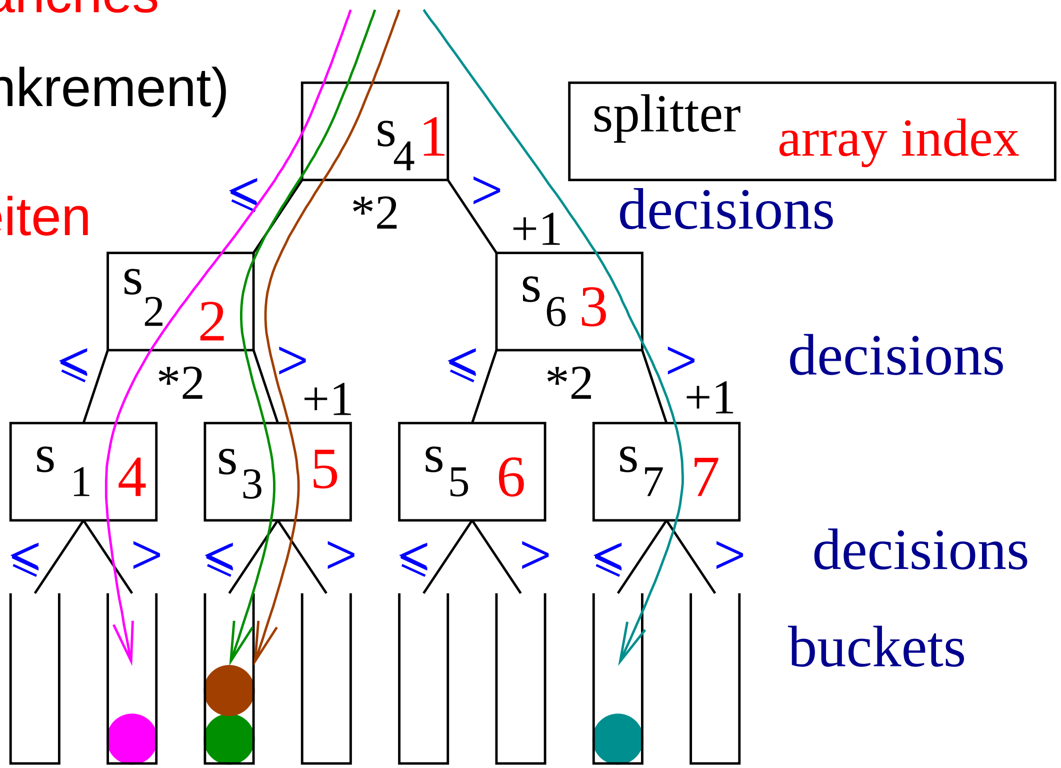
Super Scalar Sample Sort

[Sanders-Winkel 04]

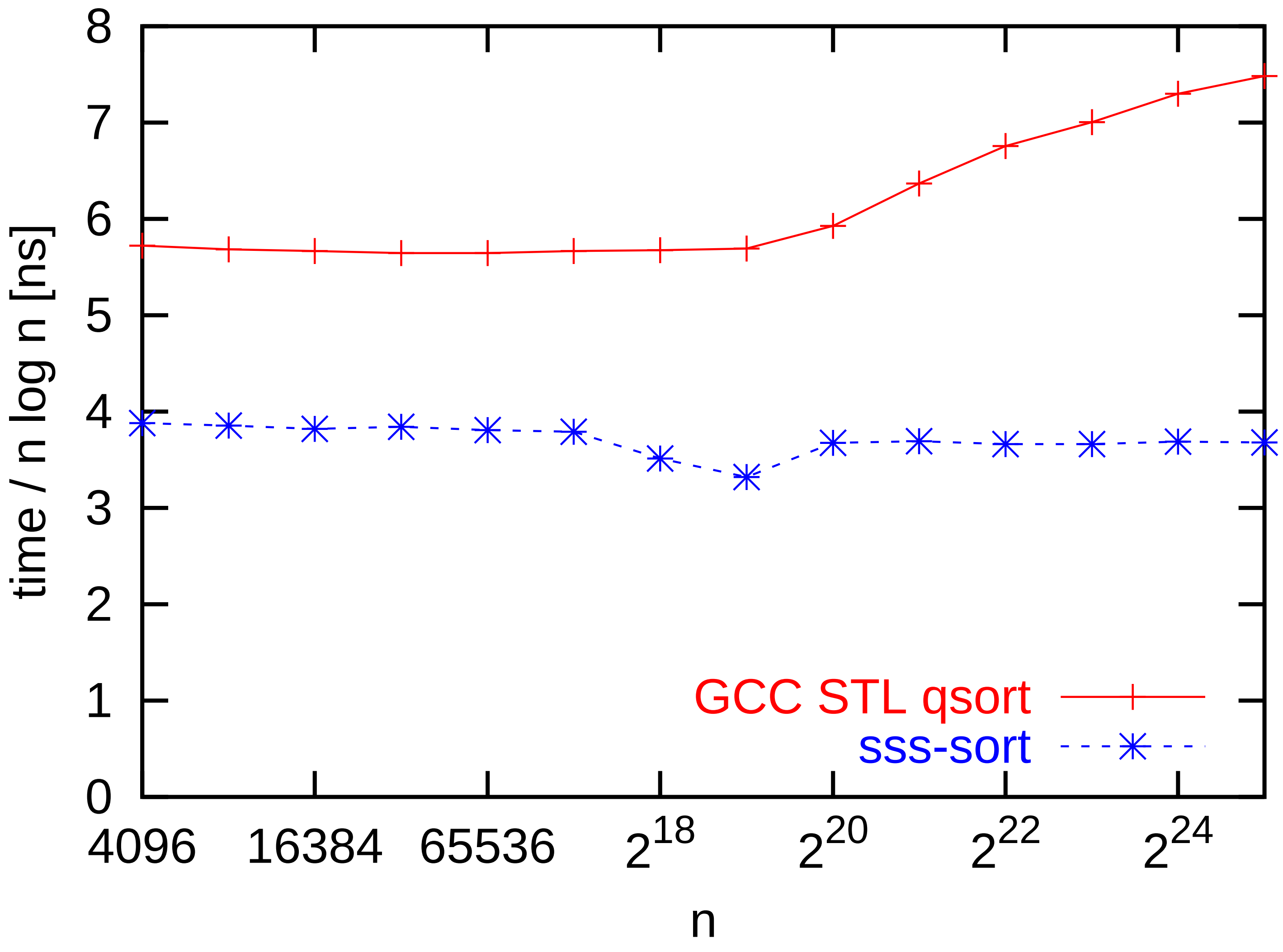
$\approx$ quicksort mit mehreren Splittern $s_1, \ldots, s_{k-1}$

Bucket i kriegt Elemente zwischen s_{i-1} und s_i; Rekursion

- Entkopple Vergleiche von Branches
  $\rightsquigarrow$ conditional execution (1 Inkrement)
- Entschärfe Datenabhängigkeiten
  $\rightsquigarrow$ Software Pipelining
- Cache-Effizienz
  $\rightsquigarrow$ multi-way distribution



Messungen Itanium 2



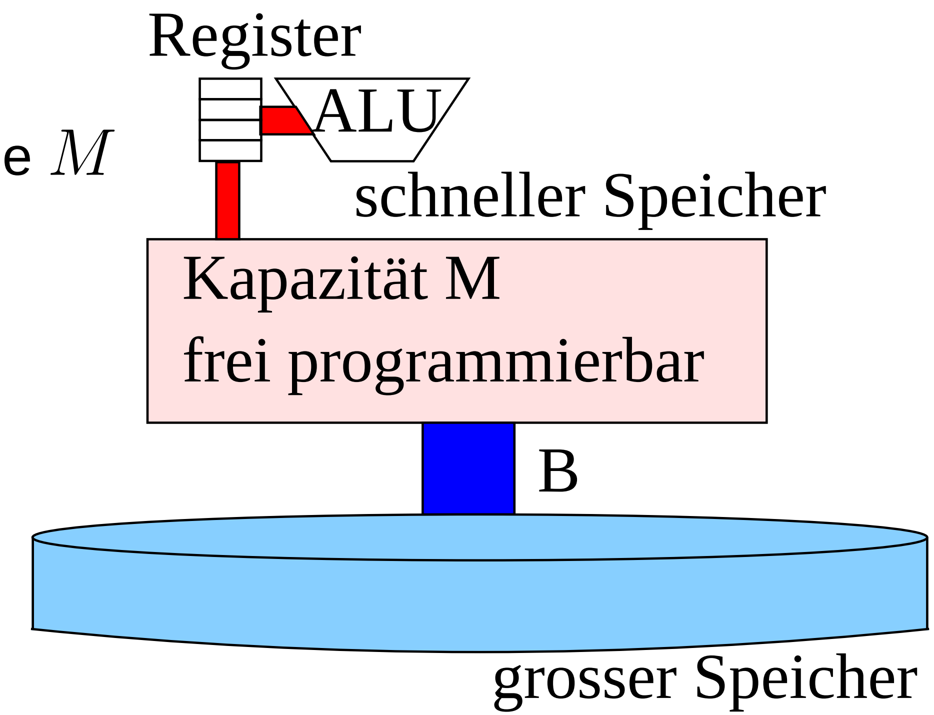
Bis zu $2\times$ schneller

Das Sekundärspeichermodell

M: Schneller Speicher der Größe M

B: Blockgröße

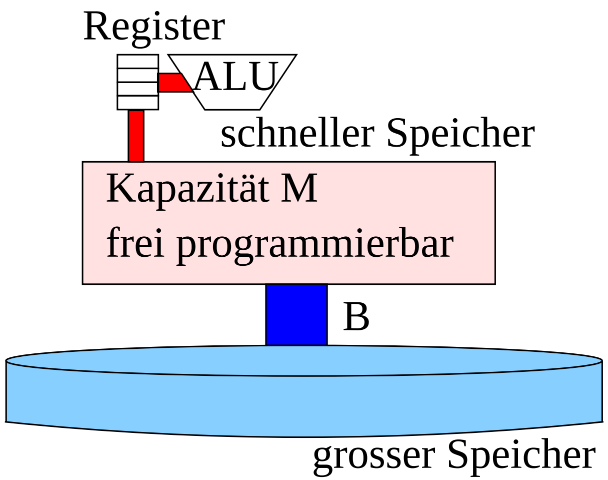
Analyse: Blockzugriffe zählen

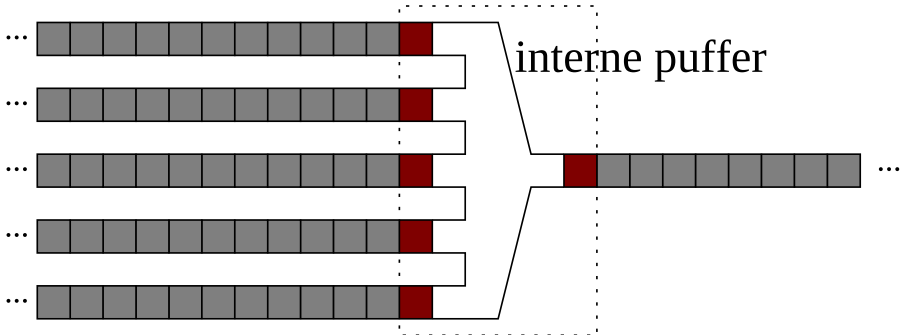


Sortieren durch Mehrwegemischen FOR

- Sortiere $\lceil n/M \rceil$ runs mit je M Elementen — $2n/B$ I/Os
- Mische je M/B runs — $2n/B$ I/Os
- bis nur ein run übrig — $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ Mischphasen

Insgesamt $$\mathrm{sort}(n) := \frac{2n}{B}\left(1 + \left\lceil \log_{M/B} \frac{n}{M} \right\rceil\right) \text{ I/Os}$$





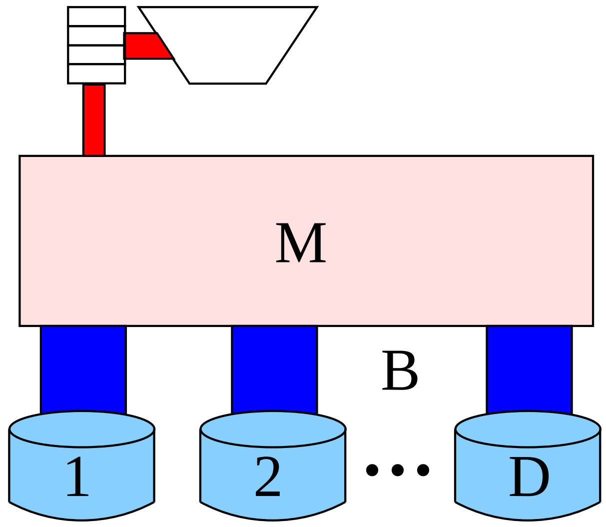
Sortieren mit parallelen Platten

I/O Schritt $:=$ Zugriff auf einen physikalischen Block pro Platte

Theorie: Balance Sort [Nodine-Vitter 93].

Deterministisch, asymptotisch optimal, $\boxed{\mathcal{O}(\cdot)}$

kompliziert.



unabhängige Platten

[Vitter Shriver 94]

Praxis: Mehr-Wege-Mischen $\boxed{\text{FOR}}$

“Meist” Faktor 10? weniger I/Os. $\boxed{42\%}$

Nicht asymptotisch optimal.

Synthese

Mehrwegemischen $+$ Zugriffsvorhersage [60s Folklore]

$+$Randomisierter Schreibzugriff [Sanders Egner Korst 00]

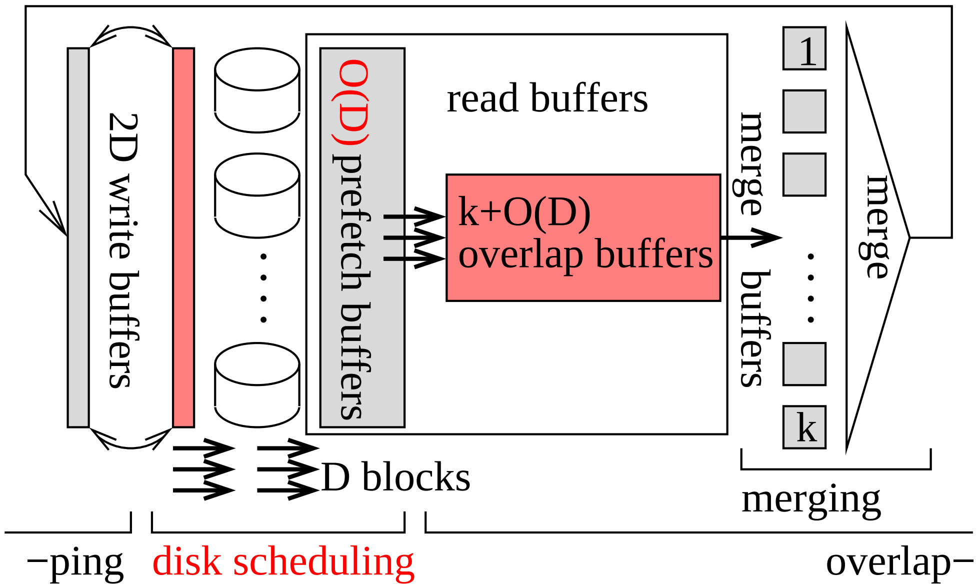
$+$Kompromiss aus Striping und Randomisierung [Vitter Hutchinson 01]

$+$Optimales Prefetching [Hutchinson Sanders Vitter 02]

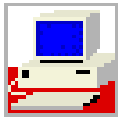
ergibt sich durch „Zeitumkehr“ aus trivialem Schreibalgorithmus

$\leadsto (1+o(1))\cdot$sort(n) I/Os

$+$Überlappung von I/O und Berechnung [Dementiev Sanders 03]



Engineering [PhD Dementiev 2002–]

Hardware (Mitte 2002) 

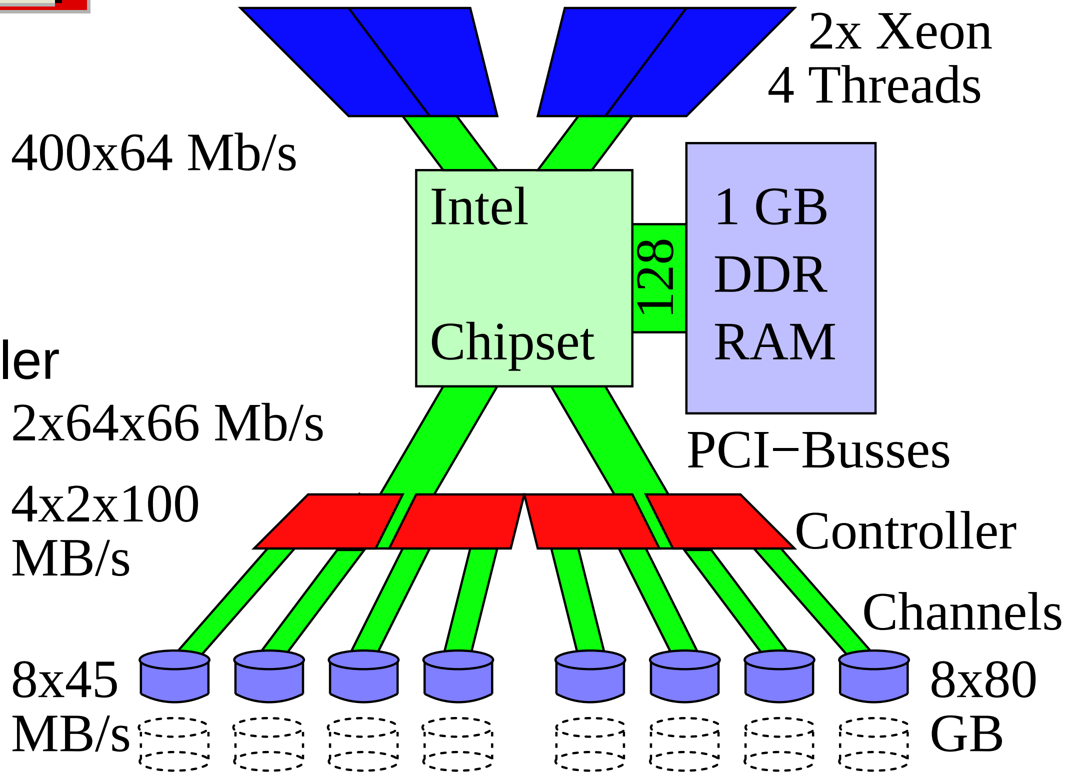
Linux Multiprozessorsystem

(2 $\times$ 2GHz Xeon $\times$ 2 Threads)

Mehrere 66 MHz PCI-Busse

Mehrere schnelle IDE Controller

Viele schnelle IDE Platten



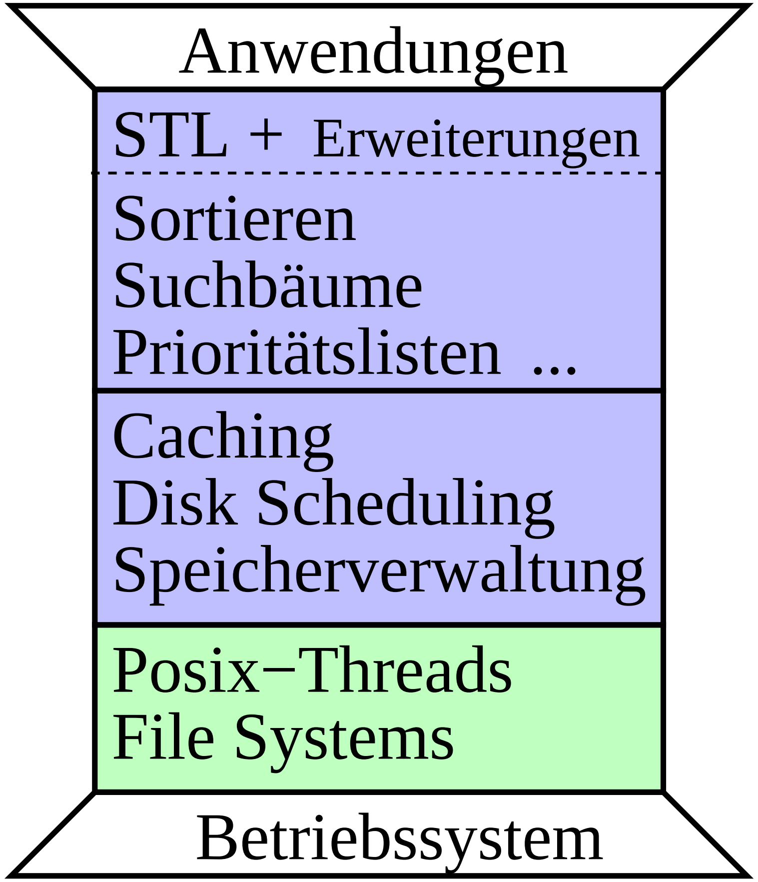
Preiswerte I/O-Bandbreite (reale 360 MB/s für ≈ 3000 €)

Software Schnittstelle

42% | FOR

Ziele: effizient + einfach + kompatibel

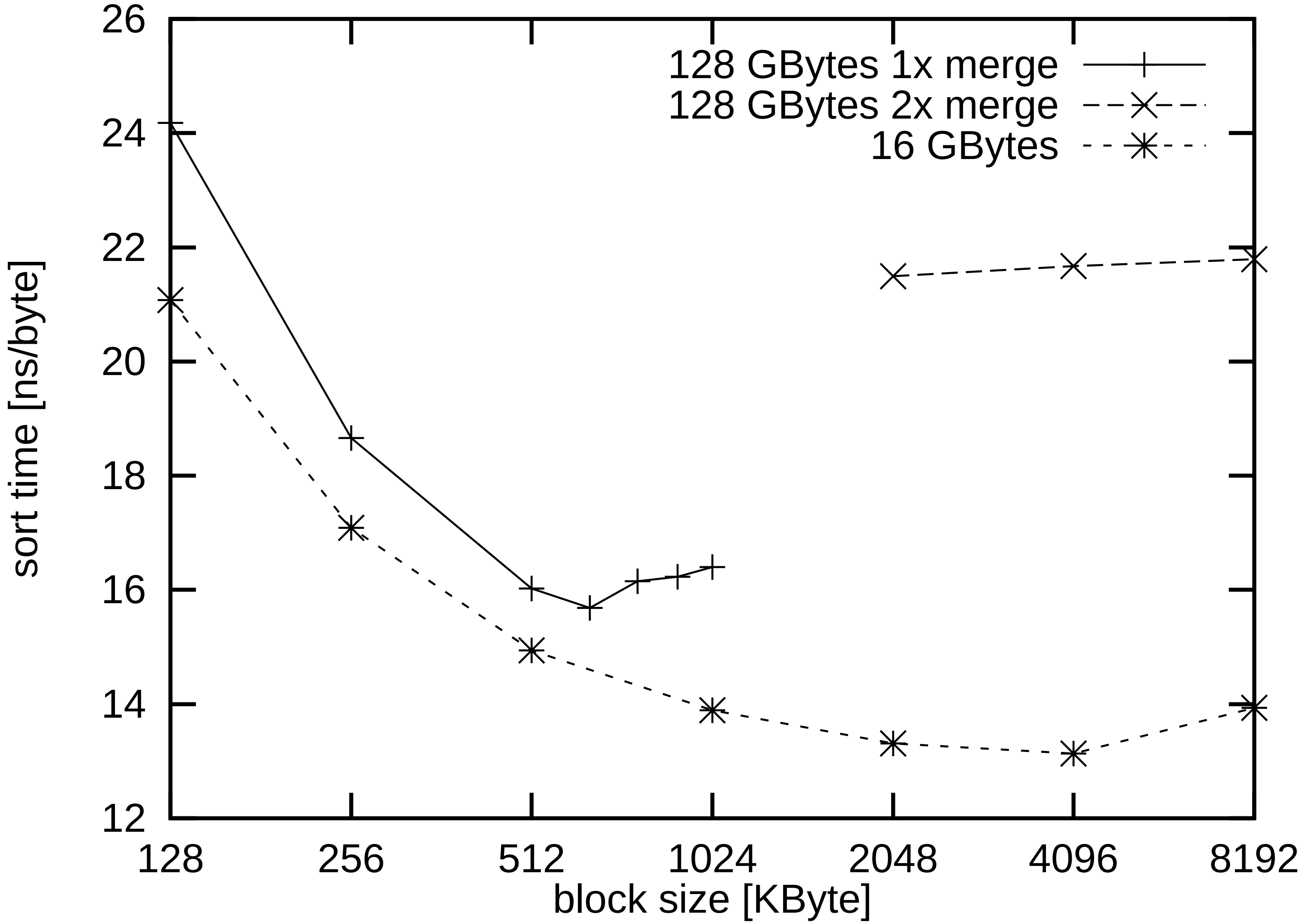
Basis: C++/STL (`iterator, vector, queue, deque, stack, map, set, stream, string, priority_queue, sort, find,`...)



Erweiterungen

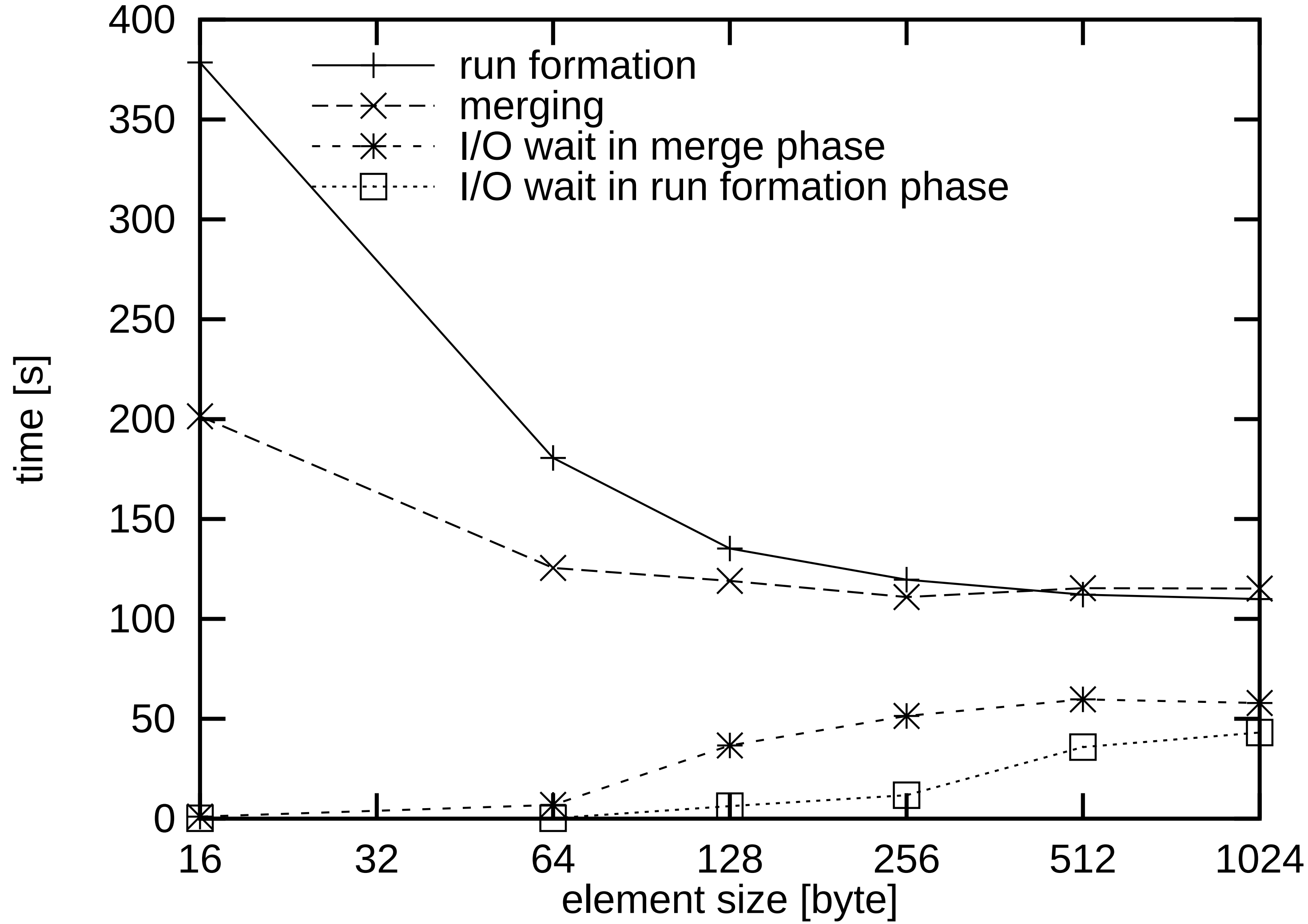
- Pipelining
- Batched updates
- Viele Kleinigkeiten, z.B., `key()` sorting

Blockgrößen (8 Platten)



Erheblich größer als vielfach angenommen. B ist keine Technologiekonstante

Elementgrößen (16 GB, 8 Platten) 42%

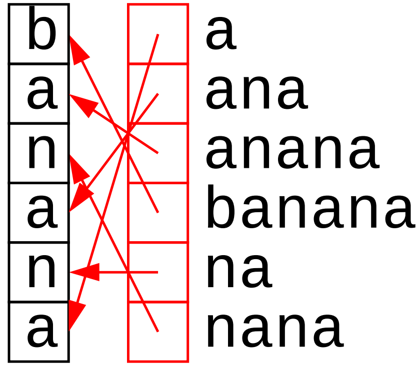


Parallele Platten ⇝ Bandbreite “for free” ⇝ interne Kosten nicht vernachlässigen

Suffixe Sortieren

[Kärkäinen-Sanders 03]

sortiere Suffixe $s_i \cdots s_n$ von string $S = s_1 \cdots s_n$, $s_i \in \{1..n\}$.



- Anwendungen: Volltextsuche,

  Burrows-Wheeler Textkompression, Bioinformatik,. . .

- Einfacher interner Linearzeitalgorithmus.

  Radix-Sort $+$ Scan $+$ lineare Rekursion

  $\rightsquigarrow$ triviale externe (und parallele) Adaptierung

- $?? \cdot 1000$ Zeilen $\longrightarrow$ 50 Zeilen $\boxed{\mathcal{O}(\cdot)} \longrightarrow$ $\boxed{\text{FOR}}\boxed{\mathcal{O}(\cdot)}\boxed{42\%}$

  [Farach, Ferragina, Muthukrishnan 96,97, 98, 00]

- Komplettiert Ersatz von Suffix-Trees insbesondere in der Lehre

```
inline bool leq(int a1, int a2,   int b1, int b2)
{ return(a1 < b1 || a1 == b1 && a2 <= b2); }
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3)
{ return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3)); }

// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];
  for (int i = 0;  i <= K;  i++) c[i] = 0;
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;
  for (int i = 0, sum = 0;  i <= K;  i++)
  {  int t = c[i];  c[i] = sum;  sum += t; }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];
  delete [] c;
}

// find the suffix array SA of s[0..n-1] in {1..K}^n
// require s[n]=s[n+1]=s[n+2]=0, n>=2
void suffixArray(int* s, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* s12  = new int[n02 + 3];  s12[n02]= s12[n02+1]= s12[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* s0   = new int[n0];
  int* SA0  = new int[n0];

  // generate positions of mod 1 and mod  2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) s12[j++] = i;
  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(s12 , SA12, s+2, n02, K);
  radixPass(SA12, s12 , s+1, n02, K);
  radixPass(s12 , SA12, s  , n02, K);

  // find lexicographic names of triples
  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0;  i < n02;  i++) {
    if (s[SA12[i]]!=c0||s[SA12[i]+1]!=c1||s[SA12[i]+2]!=c2)
    {  name++; c0=s[SA12[i]]; c1=s[SA12[i]+1]; c2=s[SA12[i]+2];}
    s12[SA12[i]/3 + n0*(SA12[i] % 3 == 2)] = name;
  }

  // recurse if names are not yet unique
  if (name < n02) {
    suffixArray(s12, SA12, n02, name);
    // store unique names in s12 using the suffix array
    for (int i = 0;  i < n02;  i++) s12[SA12[i]] = i + 1;
  } else // generate the suffix array of s12 directly
    for (int i = 0;  i < n02;  i++) SA12[s12[i] - 1] = i;

  // stably sort the mod 0 suffixes from SA12 by their first character
  for (int i=0, j=0;  i < n02;  i++)
    if (SA12[i] < n0) s0[j++] = 3*SA12[i];
  radixPass(s0, SA0, s, n0, K);

  // merge sorted SA0 suffixes and sorted SA12 suffixes
  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t]<n0?SA12[t]*3+1:(SA12[t]-n0)*3+2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    if (SA12[t] < n0 ?
        leq(s[i],       s12[SA12[t] + n0], s[j],       s12[j/3]) :
        leq(s[i],s[i+1],s12[SA12[t]-n0+1], s[j],s[j+1],s12[j/3+n0]))
    {        // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
    } else { // suffix from SA0  is smaller
      SA[k] = j;  p++;
      if (p == n0)  // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
    }
  }
  delete [] s12; delete [] SA12; delete [] SA0; delete [] s0;
}
```

Auch DER praktische Algorithmus?

Im Hauptspeicher

Nein! [Manzini Ferragina 02] mehrfach schneller für reale Eingaben.

Weniger Platz, einigermaßen cache-effizient,

string sorting in inner loops

Sekundärspeicherimplementierung

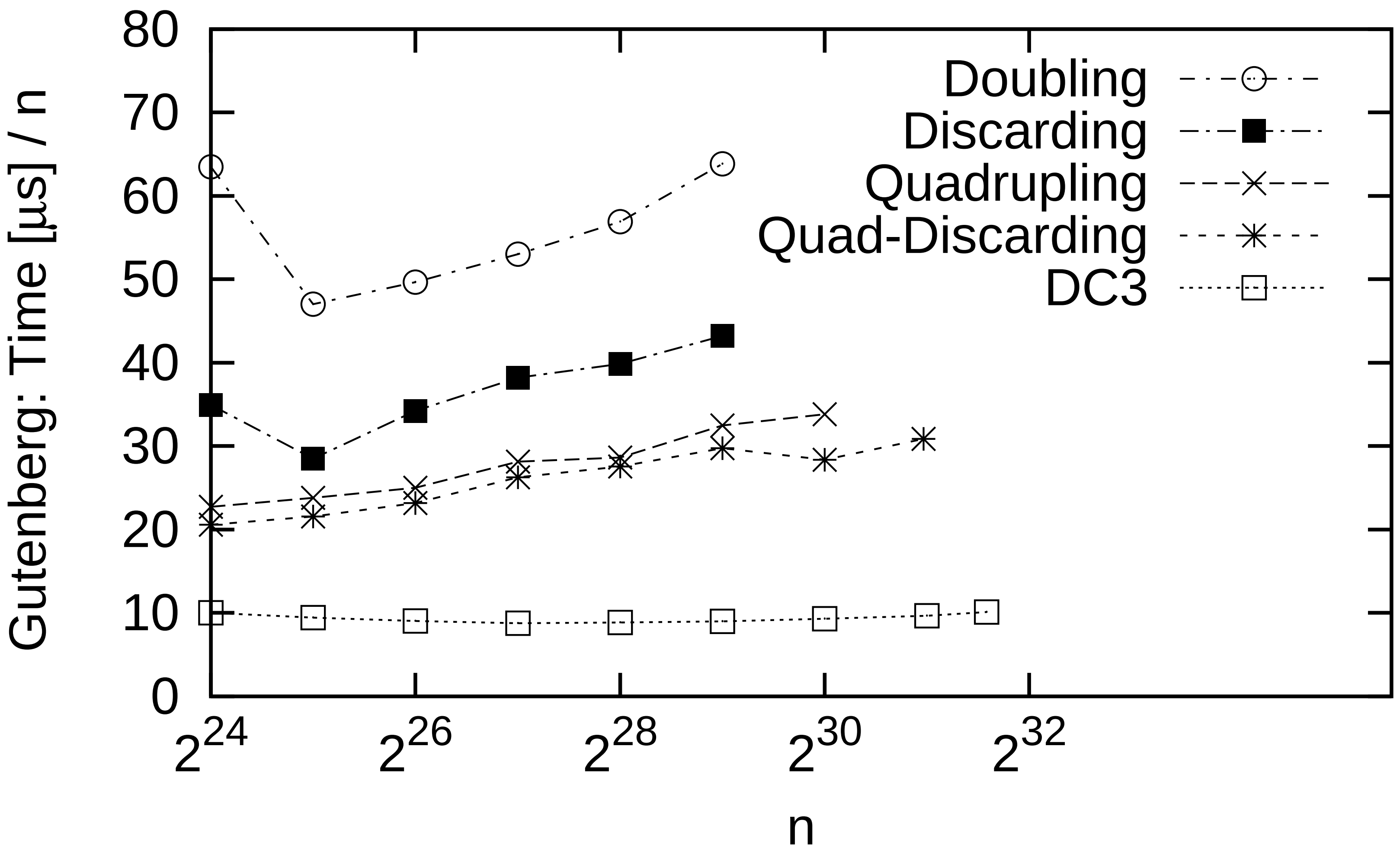
max

[Dementiev/Kärkkäinen/Mehnert/Sanders 05]

Ja!

Gutenberg Time

42%



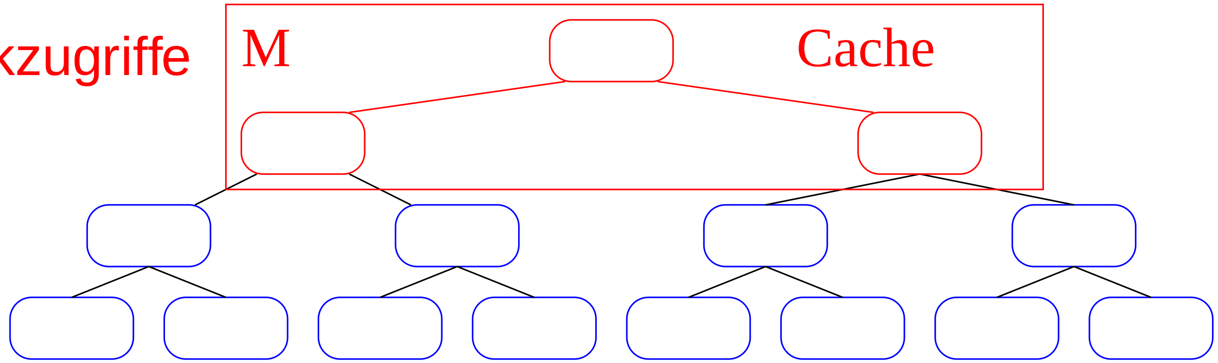
Prioritätslisten (insert, deleteMin)

Binäre Heaps bester vergleichsbasierter „Flachspeicher“-Algorithmus

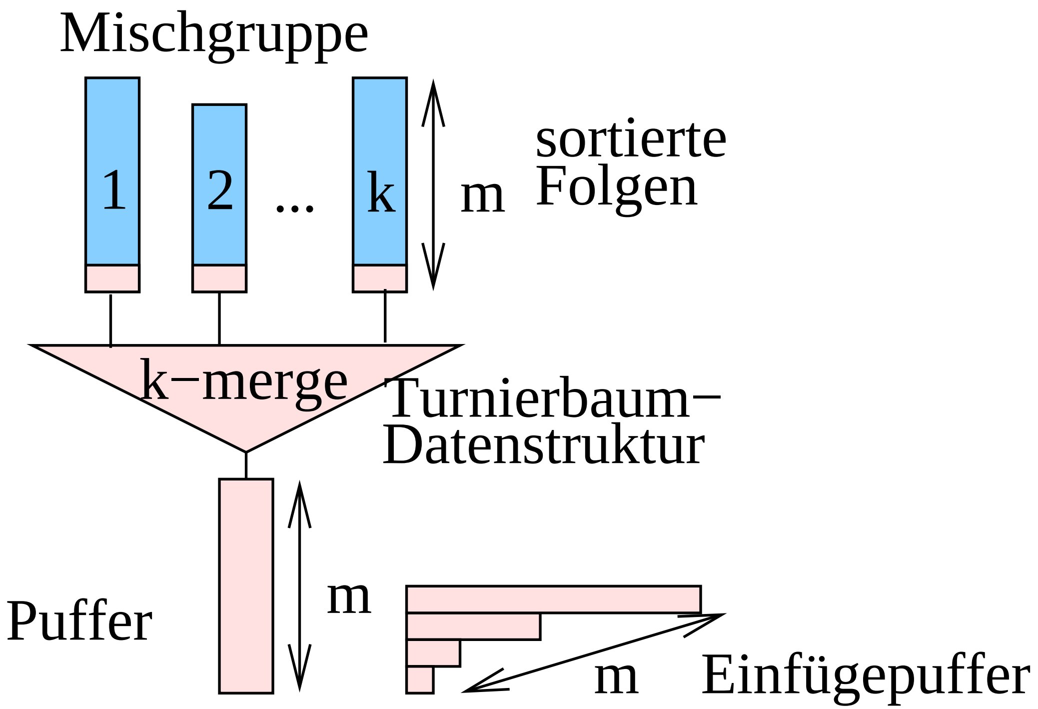
+ Im Mittel $\log n + \mathcal{O}(1)$ Schlüsselvergleiche pro Entferne-Minimum bei Einsatz der „bottom-up“ Heuristik [Wegener 93].

  (≈ # schwer vorhersagbarer Verzweigungen)

– $\approx 2\log(n/M)$ Blockzugriffe für Entferne-Minimum



Mittelgroße Queues ($km \ll M^2/B$ Einfügungen)



Einfügen: Zunächst in Einfügepuffer. Überlauf $\longrightarrow$

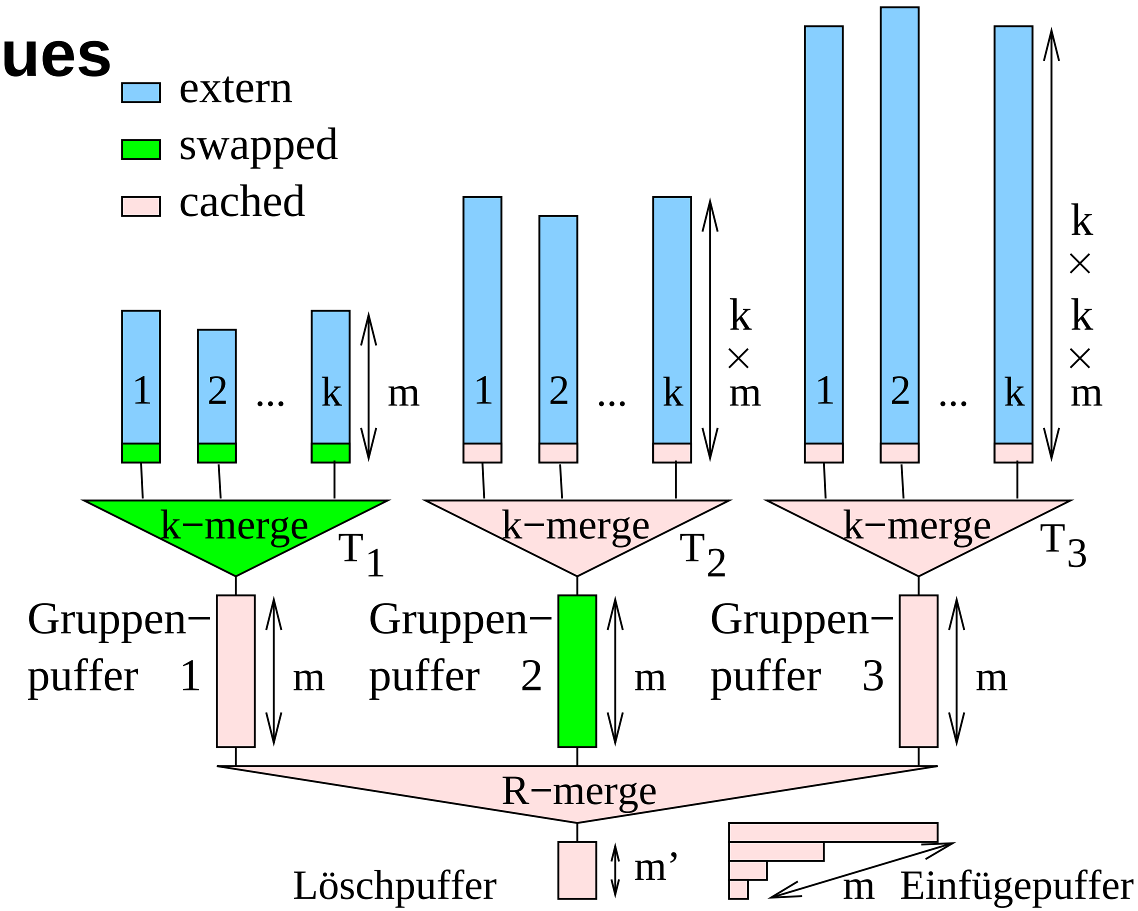
sortieren, mit Löschpuffer mischen, größte Elemente auslagern.

Entferne-Minimum: Minimum von Einfügepuffer und Löschpuffer.

Löschpuffer ggf. auffüllen.

Große Queues

[Sanders 00]



Einfügen: Gruppenüberlauf $\longrightarrow$ Gruppe mischen; in nächste Gruppe.

Ungültige Gruppenpuffer nach Gruppe 1 “abschieben”.

Entferne-Minimum: Auffüllen. $m' \ll m$. sonst nichts

Analyse

42% $\mathcal{O}(\cdot)$

I Einfügeoperationen

#Blockzugriffe und #Schlüsselvergleiche wie

sortieren von I Elementen plus “kleine Terme”

Mindestens Faktor 3 besser als andere externe Algorithmen

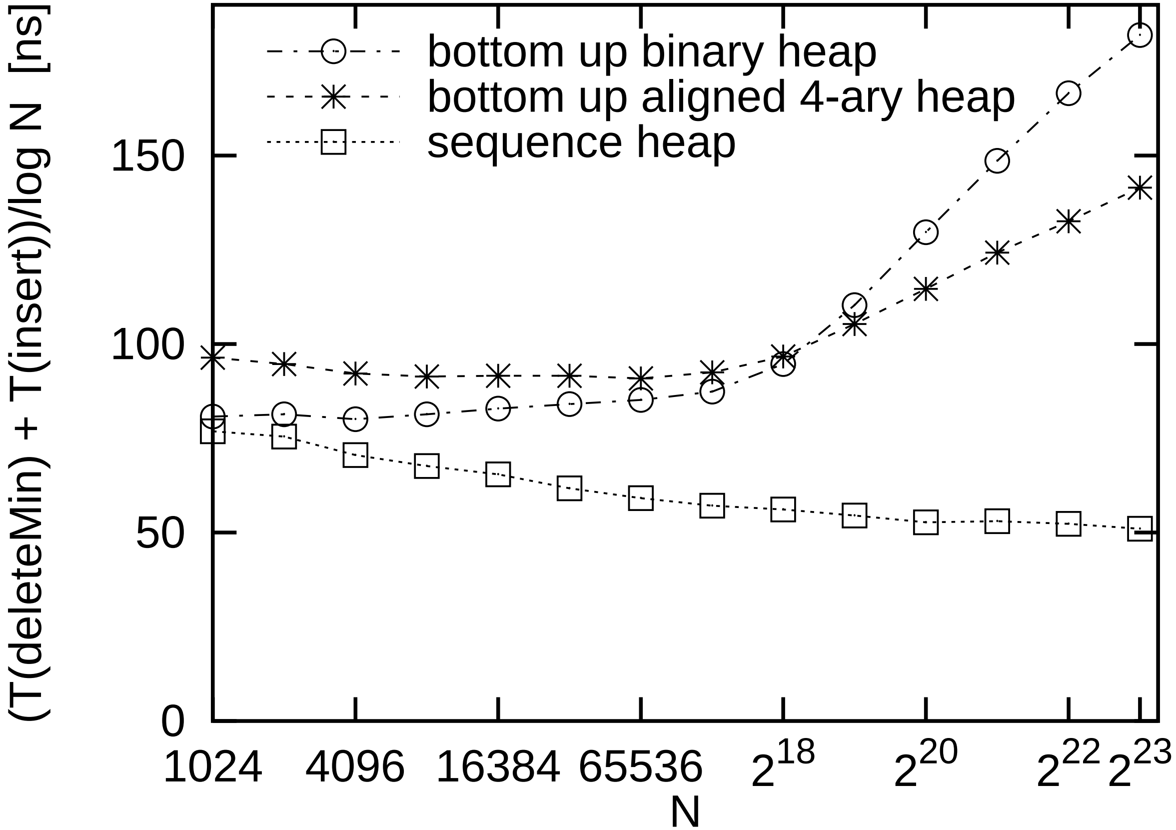
[Arge 95, Brodal-Katajainen 98, Brengel et al. 99, Fadel et al. 97]

Implementierung für Hardware-Caches

42%

MIPS R10000, zufällige Schlüssel

$\left(\text{Einf.-Entf.-Einf.}\right)^N \left(\text{Entf.-Einf.-Entf.}\right)^N$

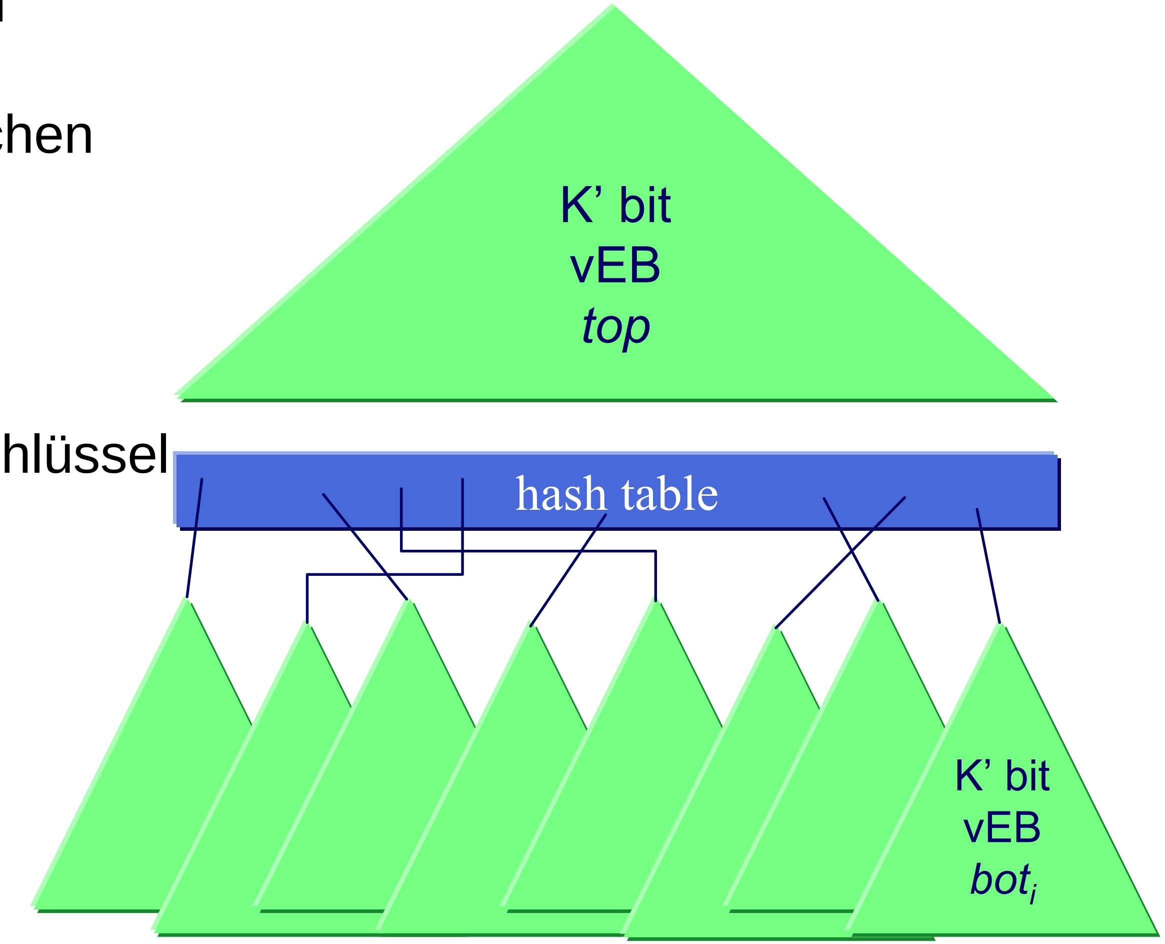


Constant factors matter $\rightsquigarrow$ alle Algorithmen effizient implementieren

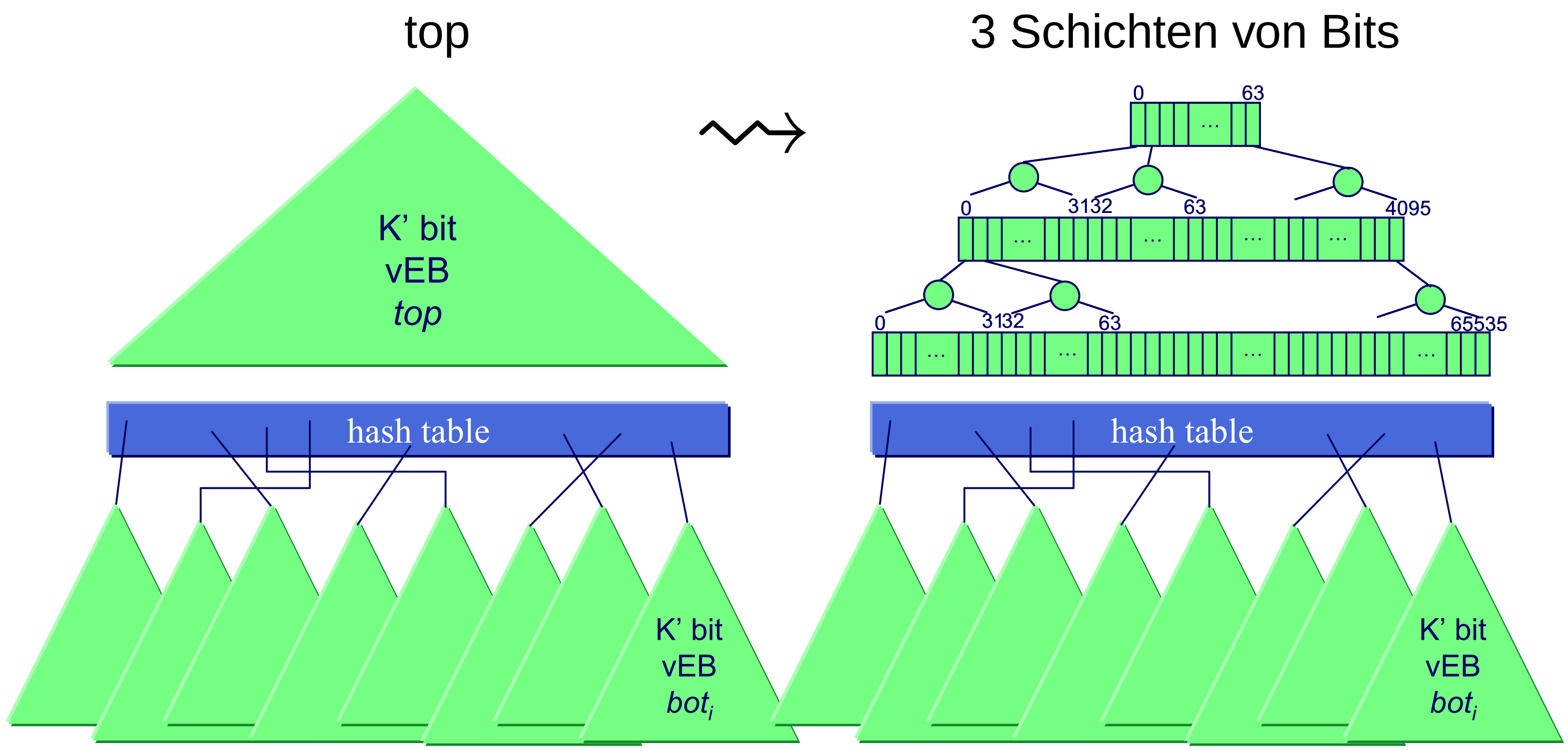
Quellen veröffentlichen

van Emde-Boas Suchbäume

- $K = 2K'$ bit Schlüssel
- Einfügen, Löschen, Suchen in $\mathcal{O}(\log K)$
- Wurzel verweist auf Teilbäume für K'-bit Schlüssel für jedes vorkommende Muster der high bits
- top speichert auftretende high bits

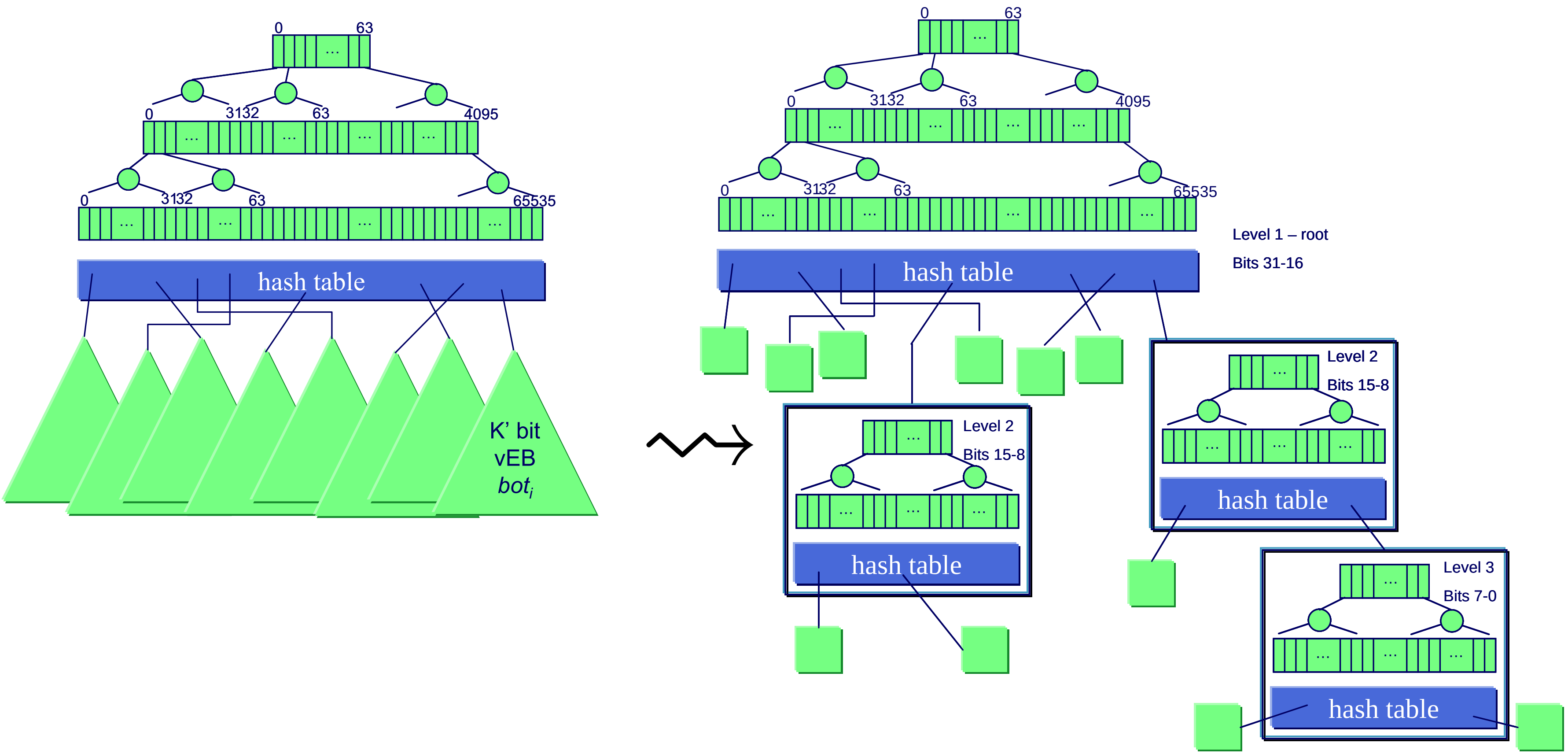


Effiziente 32 Bit-Implementierung



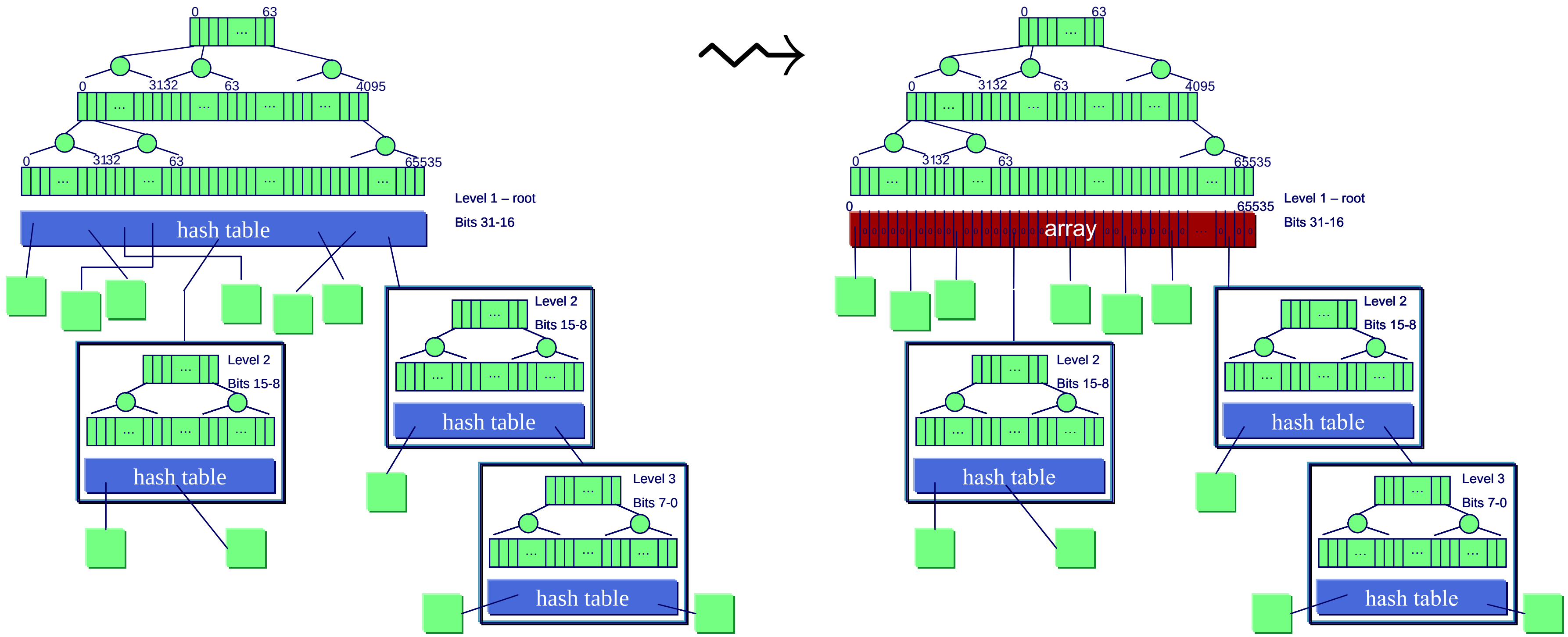
Effiziente 32 Bit-Implementierung

Rekursion nach 3 Schichten abbrechen



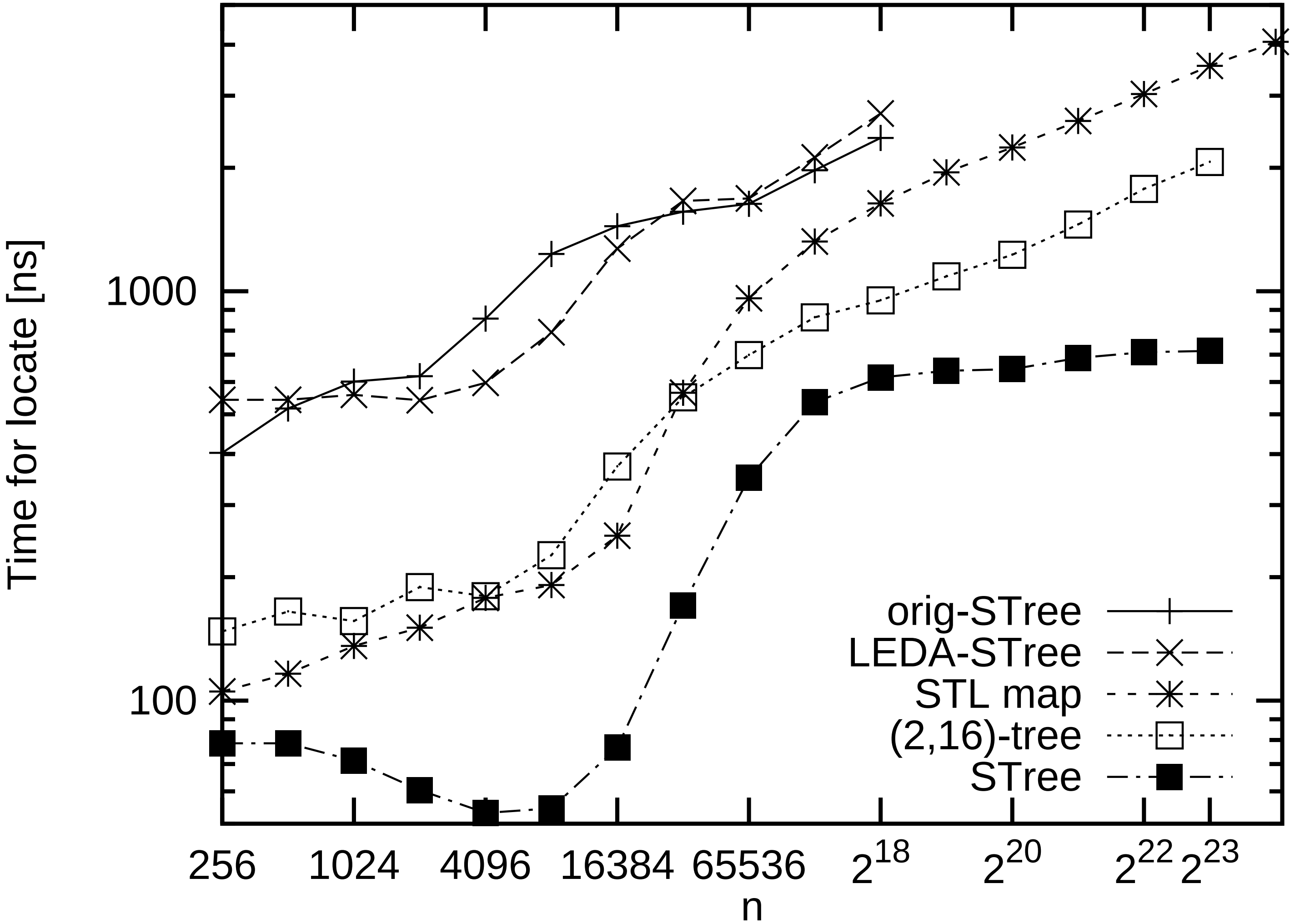
Effiziente 32 Bit-Implementierung

Wurzel-Hashtabelle $\leadsto$ Wurzel-Array



Messung: zufälliges Locate

42%

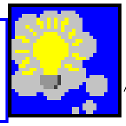
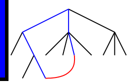


Echtzeit Hash-Tabellen

Einfügen, löschen in konstanter erwarteter Zeit

Suchen in konstanter Zeit, worst case

Theorie: [Dietzfelbinger/Karlin/Mehlhorn/MadH/Rohnert/Tarjan 94]

$\boxed{\mathcal{O}(\cdot)}$  

langsam, kompliziert, platzaufwendig

Cuckoo Hashing 42% $\mathcal{O}(\cdot)$

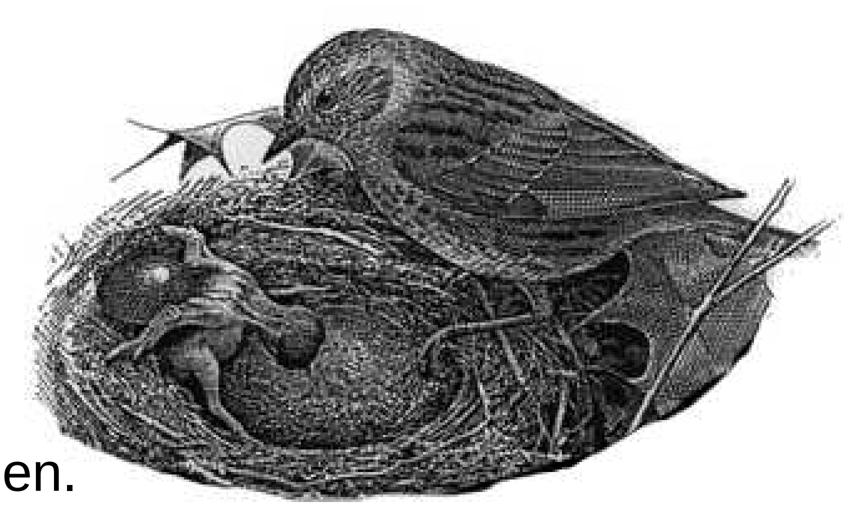
[Pagh Rodler 01]

Tabelle der Größe $(2+\epsilon)n$.

Zwei Optionen für jedes Element.

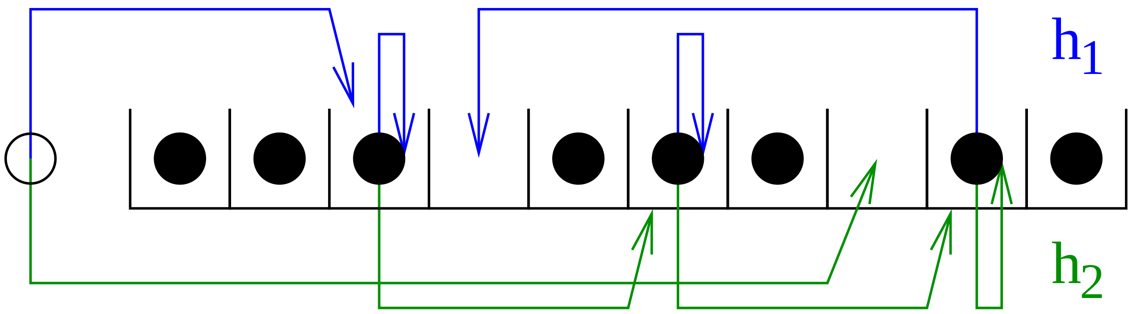
Einfügen verschiebt Elemente;

Neuaufbau falls nötig.

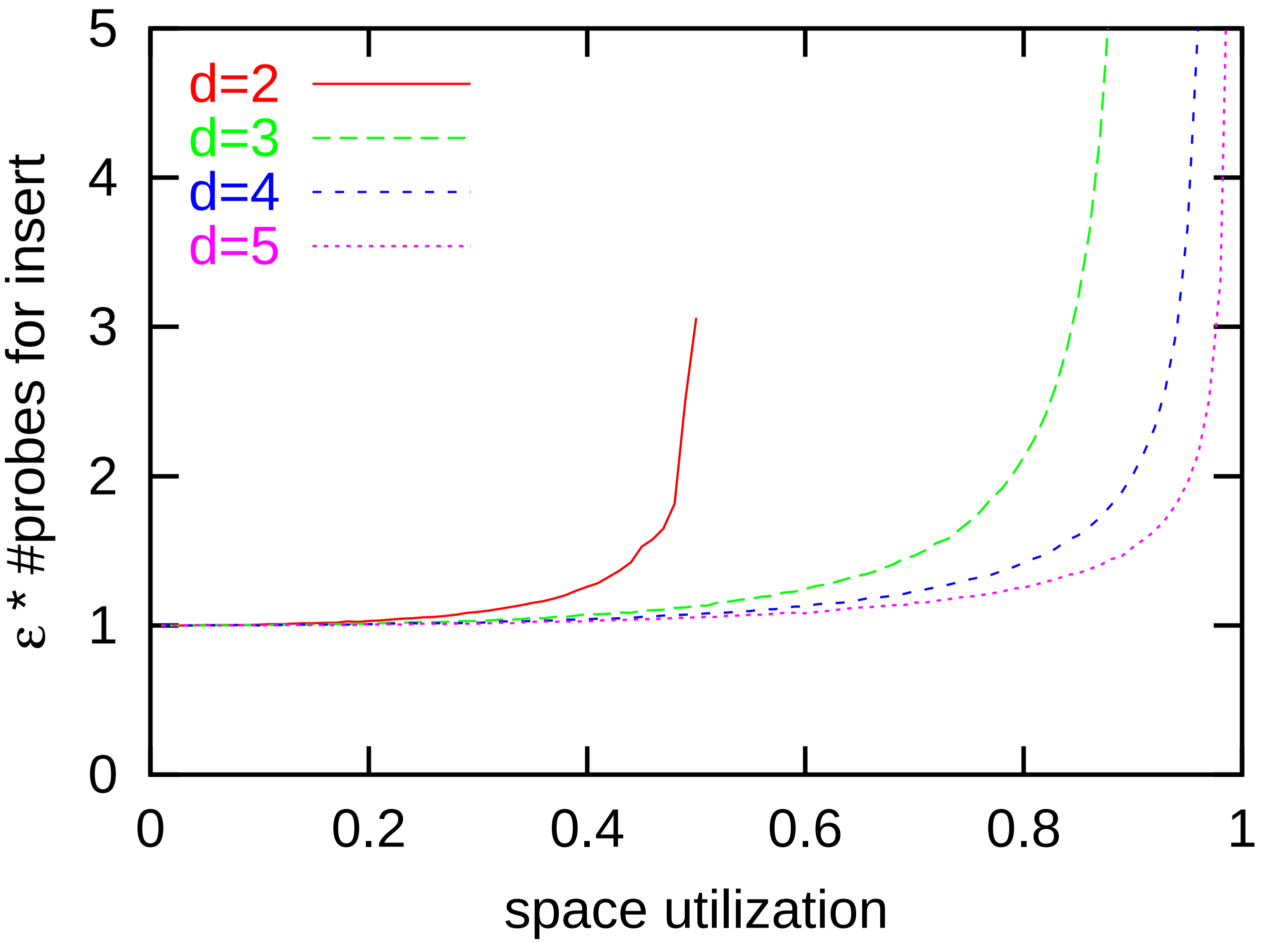


Sehr schnelles Einfügen und Löschen.

Konstante erwartete Einfügezeit.



d-ary Cuckoo Hashing [Fotakis/Pagh/Sanders/Spirakis 03]

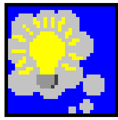


42%

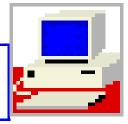
[Dietzfelbinger/Weidling 04] cache-effiziente Variante

Mehr Algorithm Engineering

Maximum Flows: Praktikable(re) Variante des theoretisch besten Algorithmus

 $\boxed{\mathcal{O}(\cdot)}(\boxed{42\%})$

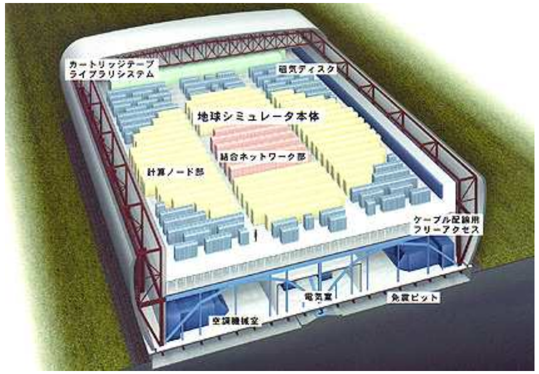
Minimale Spannbäume: Extern $+$ Praktikable(re) Variante des theoretisch besten Algorithmus

$\boxed{\text{FOR}}\boxed{\mathcal{O}(\cdot)}\boxed{42\%}$ 

Kürzeste Wege: Average case Linearzeitalgorithmus. Preprocessing für schnelle Pfadanfragen

Scheduling von parallelen Platten: Algorithmen für skalierbare Storage-Server

Konkrete Anwendungen



- Massiv parallele relativistische Optik
- Kollektive Kommunikation in Parallelrechnern
- Airline Crew Scheduling (Lufthansa, AF, BA, KLM, SAS, SA,...)

  (parallele, cache-effiziente, Active-Set Set-Covering-Heuristik)

  $\ln n$ nichtapproximierbar $\rightsquigarrow$ wenige ‰ Fehler oder ... max

- Lastverteilung für Klima- und Wettersimulation 42% ($\mathcal{O}(\cdot)$)

Zusammenfassung und Ausblick

- Selbst fundamentale, „einfache“ algorithmische Probleme werfen noch interessante Fragen auf
- Implementierung und Experiment sind wichtig und wurden von Teilen der Algorithmik-Community vernachlässigt

$+$ Theorie ist ein (mindestens) ebensowichtiger, integraler Bestandteil des Algorithmenentwurfsprozesses

$=$ Algorithm Engineering

Graphenalgorithmen

Kürzeste Wege

- Linearzeit für zufällige Kantengewichte

  [Meyer Sanders ESA 98, Meyer 01, Goldberg 01. . .]

- Geometrisches Modell für ernergieeffizientes Routing in Radionetzen.

  Exakt: „nahezu“ Linearzeit (statt quadratisch!)

  [Beier Sanders Sivadasan ICALP 02]

  $(1+\epsilon)$ Näherung: $\mathcal{O}(n \log n)$ preprocessing, $\mathcal{O}(n)$ Platz,

  $\mathcal{O}(1)$ Anfragen [Funke Matijevic Sanders ESA 03]

Maximale Flüsse

Theorie: $\mathcal{O}(m\Lambda\log(n^2/m)\log U)$

Binary-Blocking-Flow-Algorithmus mit $\Lambda = \min\{m^{1/2}, n^{2/3}\}$

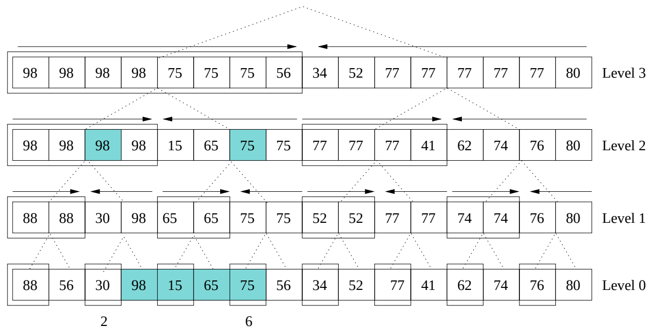
[Goldberg-Rao-97].

Problem: Bester Fall $\approx$ schlechtester Fall

[Hagerup Sanders Träff WAE 98]:

- Praktikable Verallgemeinerung und Implementierung
- Best Case $\ll$ Worst Case
- Bester Algorithmus für einige „schwere“ Instanzen

Minimale Spannende Bäume



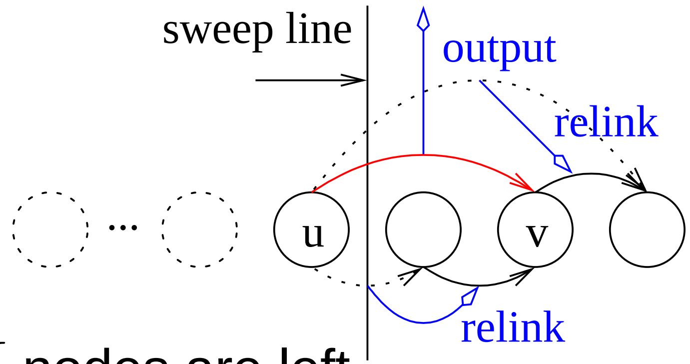
Ein praktikabler, vektorisierbarer Algorithmus,

der die cycle property nutzt

[Katriel Sanders Träff ESA 03]

Schlüssel: Eine Datenstruktur für $\mathcal{O}(1)$ Intervallminimum-Anfragen

Ein einfacher externer Algorithmus:



foreach node v in random order **do**

    contract the lightest edge out of v

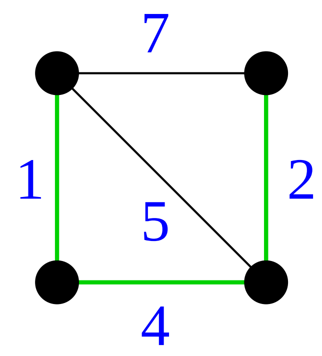
    switch to Kruskal's algorithm when only M nodes are left

$\mathcal{O}\left(\text{sort}(m \ln \frac{n}{M})\right)$ erwartete I/Os, Schlüssel $\approx$ externe Prioritätslisten

Für realistische Eingaben $4\times$ besser als bisherige Algorithmen

[Sanders Schultes Sibeyn 04]

Minimale Spannende Bäume



- Zusammenhängender Graph:

  $G = (V, E)$, $|V| = n$, $|E| = m$

- Positive Kantengewichte: $c(e) > 0$

- Finde Kantenmenge T die alle Knoten in V verbindet und dabei

  $\sum_{e \in T} c(e)$ minimiert

Theorie:

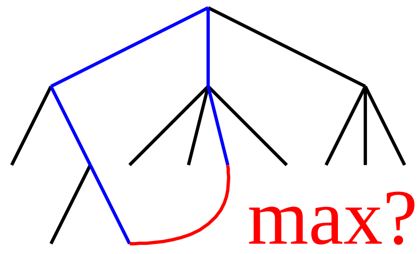
$\mathcal{O}(m)$ erwartete Zeit.

Randomisierter RAM-Algorithmus.

Kandidaten für T ausschließen mittels

komplizierter Filter-Datenstruktur

[KKT95,King 97].



Praxis:

Algorithmen, die Kanten festlegen

m klein: $\mathcal{O}(m \log m)$ [Kruskal 56]

(züchte einen Wald) $+$ Heuristiken [Mehlhorn Näher 99]

m groß: $\mathcal{O}(n \log n + m \log\log n)$ [Jarník 30, Prim 57]

(lasse Baum wachsen) mit Pairing-Heap-Prioritätslisten

[Shapiro-Moret 91].

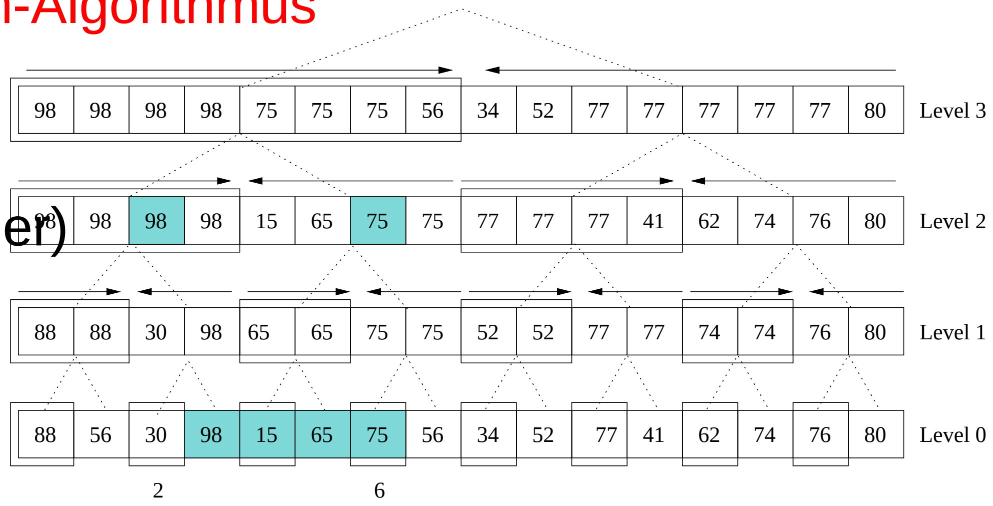
Praktikables Kantenfiltern

[Katriel Sanders Träff ESA 03]

- Einfache Reduktion: Cycle test $\rightsquigarrow$ range minima,
  Gewichte in Einfügereihenfolge des Jarník-Prim-Algorithmus
- Opfere $\log n$ Faktor im Preprocessing für konstanten Faktor pro Kante

$\rightsquigarrow$ $\mathcal{O}(m + n\log n)$ mit verbesserter Konstante vor dem m verglichen mit Jarník-Prim-Algorithmus

- Speedup bis zu $2.\cdots$,
  (bis zu 13 für Vektorrechner)



Externe MSTs

[Bachelor-Arbeit Dominik Schultes]

for $i := n$ **to** n' **do**

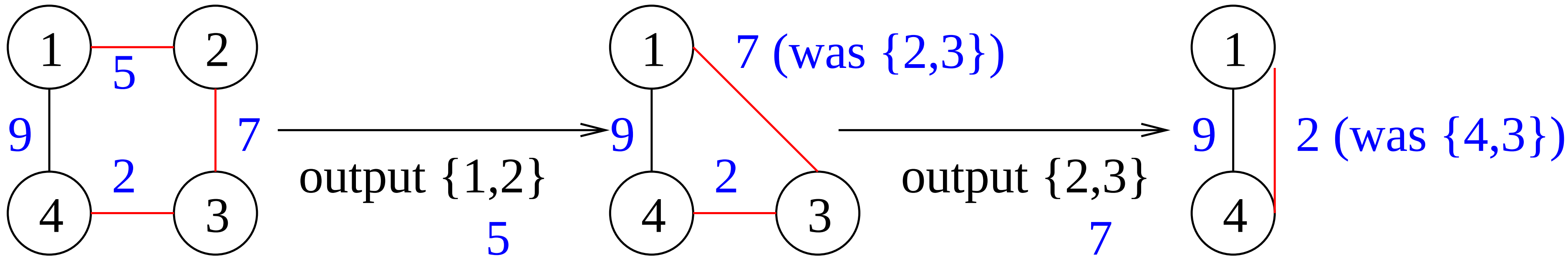
pick a random node v

find the lightest edge (u,v) out of v and output it

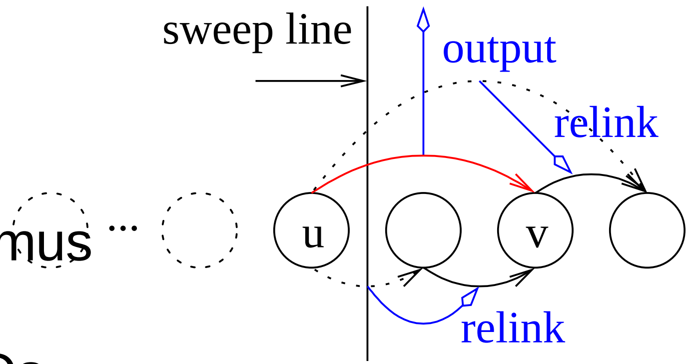
contract (u,v)

$$\mathbb{E}[degree(v)] \leq 2m/i$$

$$\sum_{n'<i\leq n} \frac{2m}{i} = 2m\left(\sum_{0<i\leq n}\frac{1}{i} - \sum_{0<i\leq n'}\frac{1}{i}\right) \approx 2m(\ln n - \ln n') = 2m \ln \frac{n}{n'}$$



Ergebnis



- Einfach extern implementierbar
- $n' = M \leadsto$ semiexterner Kruskal Algorithmus
- Insgesamt $\mathcal{O}\left(\mathrm{sort}(m \ln \frac{n}{m})\right)$ erwartete I/Os
- Für realistische Eingaben mindestens $4\times$ bisher als bisher bekannte Algorithmen
- Implementierung in <stxxl> mit bis zu 96 GByte grossen Graphen läuft „über Nacht“

Mehr zu Graphen

Kürzeste Wege

Maximale Flüsse