Algorithm Engineering für grundlegende Datenstrukturen und Algorithmen

Peter Sanders

Was sind die schnellsten implementierten Algorithmen

für das 1×1 der Algorithmik:

Listen, Sortieren, Prioritätslisten, Sortierte Listen, Hashtabellen,

Graphenalgorithemen?

Nützliche Vorkenntnisse

- Informatik I/II
- Algorithmentechnik (gleichzeitig hören vermutlich OK)
- etwa Rechnerarchitektur (oder Ct lesen ;-)
- passive Kenntnisse von C/C++

Vertiefungsgebiet: Algorithmik

Material

- Folien
- wissenschaftliche Aufsätze. Siehe Vorlesungshomepage
- Basiskenntnisse: Algorithmenlehrbücher, z.B. Cormen Leiserson Rivest???, Mehlhorn, Sedgewick, sowie ein Manuskript zu einem neuen Lehrbuch
- Mehlhorn Näher: The LEDA Platform of Combinatorial and Geometric Computing. Gut für die fortgeschrittenen Papiere.

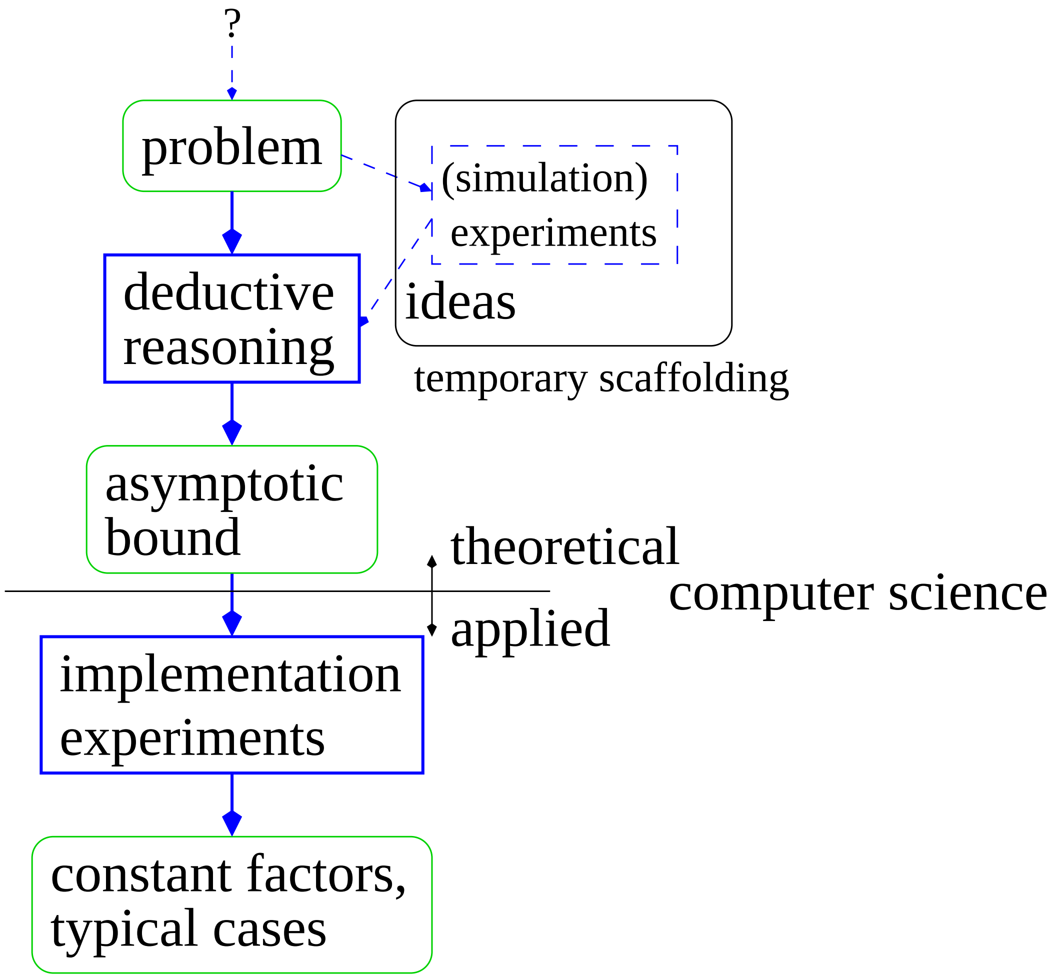
Überblick

- Was ist Algorithm Engineering, Modelle, ...
- Erste Schritte: Arrays, verkettete Listen, Stacks, FIFOs,...
- Sortieren rauf und runter
- Prioritätslisten
- Sortierte Listen
- Hashtabellen
- Minimale Spannbäume
- Kürzeste Wege
- Ausgewählte fortgeschrittene Algorithmen, z.B. maximale Flüsse

Methodik: in Exkursen

The Traditional Theoretical View?

A Waterfall Model of Algorithmics



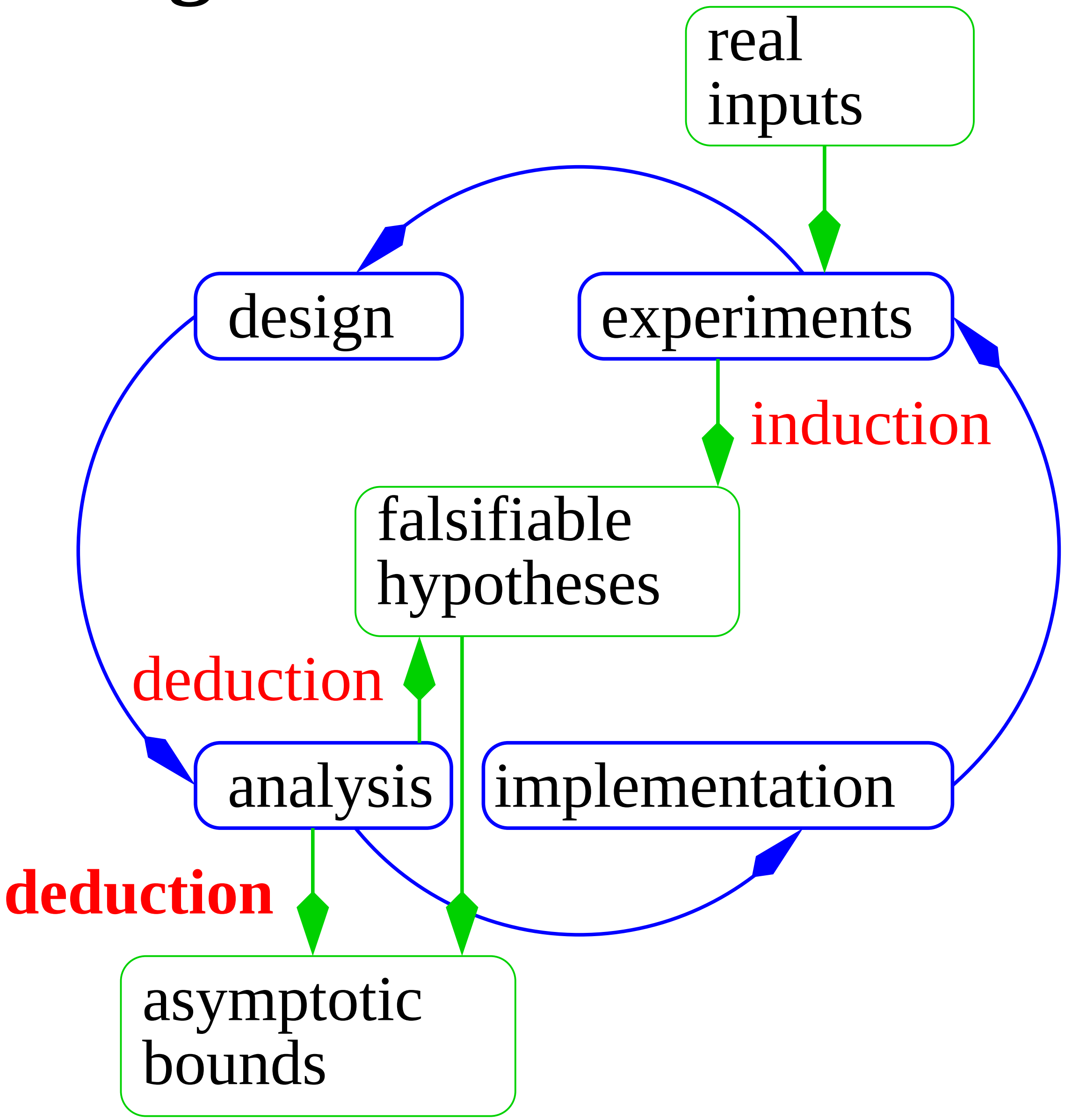
Algorithm Engineering

Scientific Method

The model

of natural science

[Popper 1934]

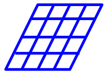
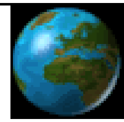
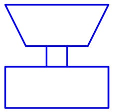
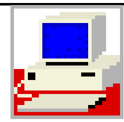
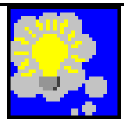
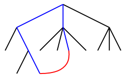
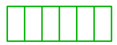
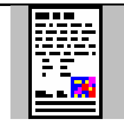


Goals

- Theory meets technology — machine models must cope with technologiscal developments

- Faster transfer of algorithmic result into applications
- Bridge existing gaps

Symptoms of Gaps

theory	$\longleftrightarrow$	**practice**
simple 	**problem model**	 complex
simple 	**machine model**	 real
complex 	**algorithms**	FOR simple
advanced 	**data structures**	 arrays,...
worst case max	**complexity measure**	 real inputs
asymptotical $O(\cdot)$	**efficiency**	42% constant factors

Why Bridge Gaps?

- With growing problems size asymptotics will eventually win
- Worst case bounds
  $\rightsquigarrow$ performance guarantees
  $\rightsquigarrow$ quality, real time properties
- Theory in the natural science means:
  Theory explains reality

Warum diese Vorlesung?

- Jeder Informatiker kennt einige Lehrbuchalgorithmen
  $\rightsquigarrow$ wir können gleich mit Algorithm Engineering loslegen
- Viele Anwendungen profitieren
- Es ist frappierend, dass es hier noch Neuland gibt
- Basis für Studien- Diplomarbeiten

Was diese Vorlesung nicht ist:

Keine wiedergekäute Algorithmentechnik o.Ä.

- Grundvorlesungen “vereinfachen” die Wahrheit oft
- z.T. fortgeschrittene Algorithmen
- steilere Lernkurve
- Implementierungsdetails
- Betonung von Messergebnissen

Was diese Vorlesung nicht ist:

Keine Theorievorlesung

- keine (wenig?) Beweise
- Reale Leistung vor Asymptotik

Was diese Vorlesung nicht ist:

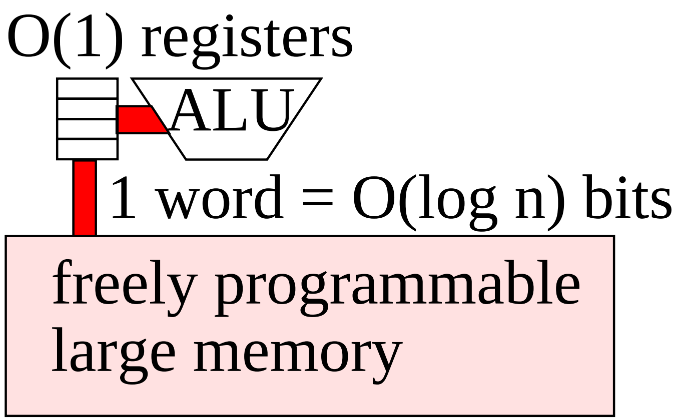
Keine Implementierungsvorlesung

- Etwas Algorithmenanalyse,. . .
- Wenig Software Engineering
- Keine Implementierungsübungen (aber, stay tuned für ein Praktikum)

Exkurs: Maschinenmodelle

RAM/von Neumann Modell

Analyse: zähle Maschinenbefehle — load, store, Arithmetik, Branch,. . .

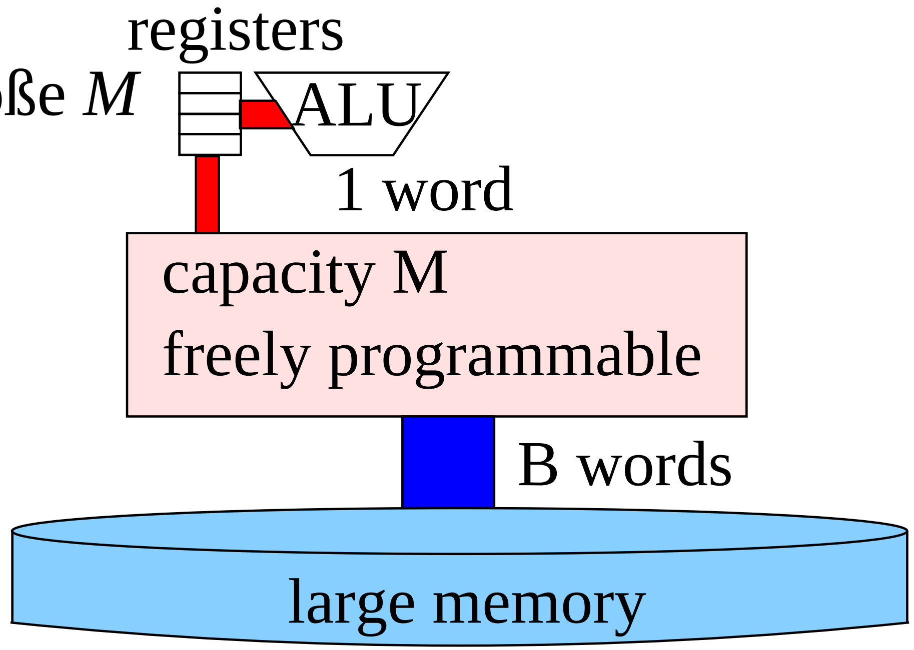


- Einfach
- Sehr erfolgreich
- zunehmrend unrealistisch
  weil reale Hardware
  immer komplexer wird

Das Sekundärspeichermodell

M: Schneller Speicher der Größe *M*

B: Blockgröße



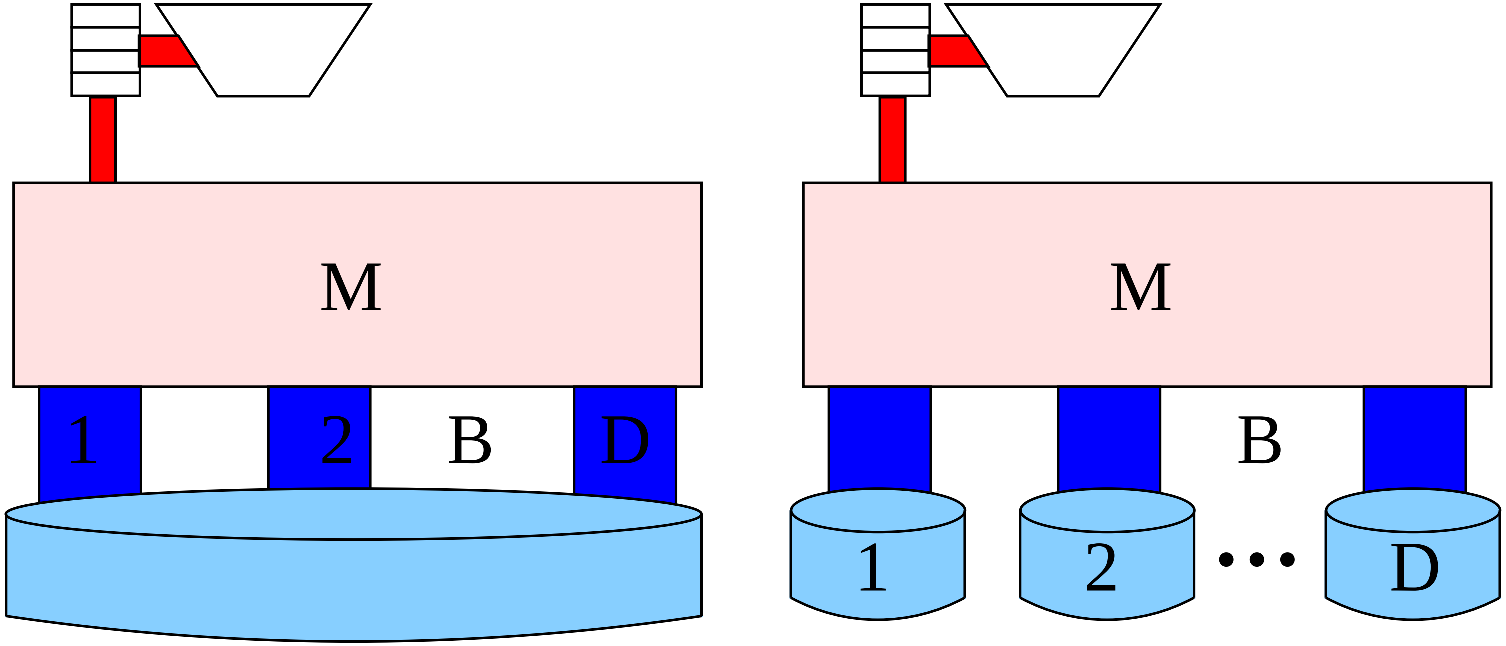
Analyse: zähle (nur?) Blockzugriffe (I/Os)

Interpretationen des Sekundärspeichermodelle

	Externspeicher	Caches
großer Speicher	Platte(n)	Hauptspeicher
M	Hauptspeicher	ein cache level
B	Plattenblock (MBytes!)	Cache Block (16–256) Byte

Ggf. auch zwei cache levels.

Parallele Platten



Mehrkopfmodell

[Aggarwal Vitter 88]

unabhängige Platten

[Vitter Shriver 94]

Mehr Modellaspekte

- Instruktionsparallelismus (Superscalar, VLIW, EPIC,SIMD,. . .)
- Pipelining
- Was kostet branch misprediction?
- Multilevel Caches (gegenwärtig 2–3 levels) $\rightsquigarrow$ “cache oblivious algorithms”
- Parallele Prozessoren, Multithreading
- Kommunikationsnetzwerke
- . . .

1 Arrays, Verkettete Listen und abgeleitete Datenstrukturen

Bounded Arrays

Eingebaute Datenstruktur.

Größe muss von Anfang an bekannt sein

Unbounded Array

z.B. `std::vector`

pushBack: Element anhängen

popBack: Letztes Element löschen

Idee: verdopple wenn der Platz ausgeht
halbiere wenn Platz verschwendet wird

Wenn man das richtig macht, brauchen
n pushBack/popBack Operationen Zeit $O(n)$
Algorithmese: pushBack/popBack haben konstante amortisierte
Komplexität Was kann man falsch machen?

Doppelt verkettete Listen

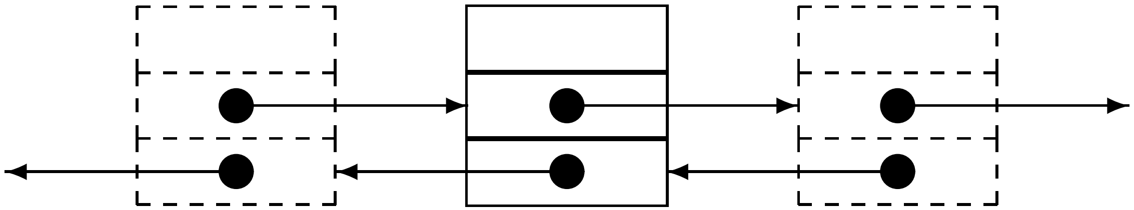
Class Item **of** Element // one link in a doubly linked list

e : Element

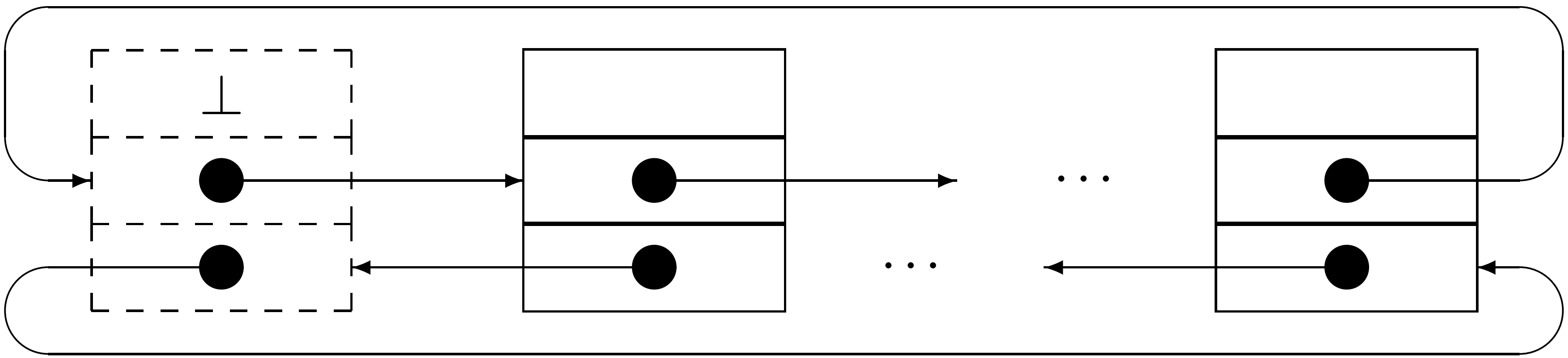
next : Handle //

prev : Handle

invariant next→prev=prev→next=**this**



Trick: Use a dummy header



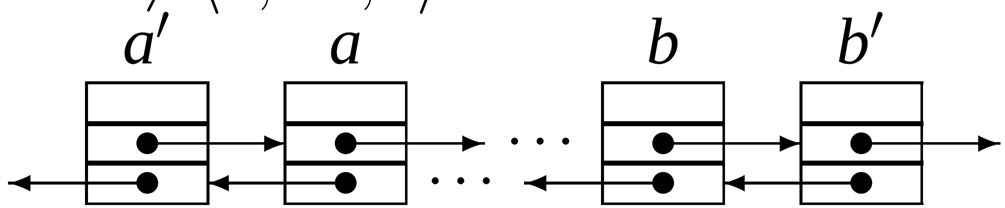
Procedure splice(a,b,t : Handle)

assert b is not before $a \wedge t \notin \langle a, \ldots, b \rangle$

// Cut out $\langle a, \ldots, b \rangle$

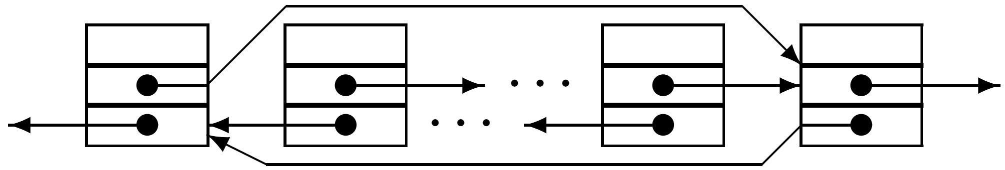
a' := a→prev

b' := b→next



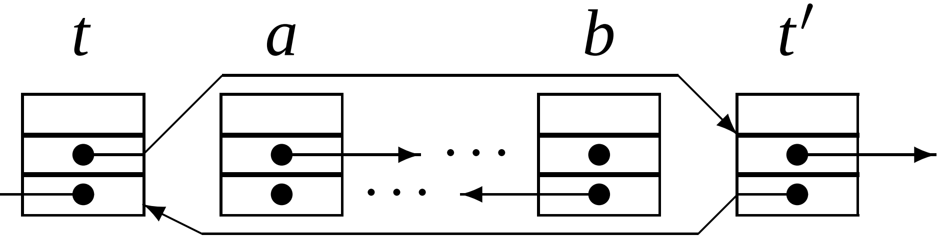
a' →next := b' //

b' →prev := a' //



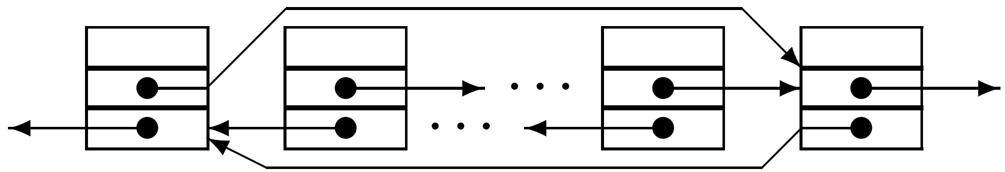
// insert $\langle a, \ldots, b \rangle$ after t

t' := t→next //



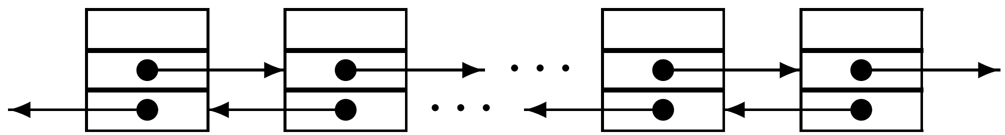
b→next := t’ //

a→prev := t //

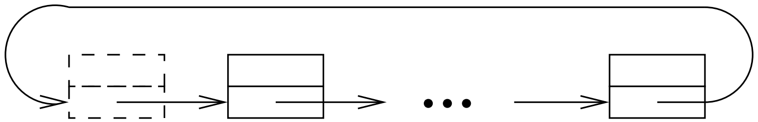


t →next := a //

t' →prev := b //



Einfach verkettete Listen



Vergleich mit doppelt verketteten Listen

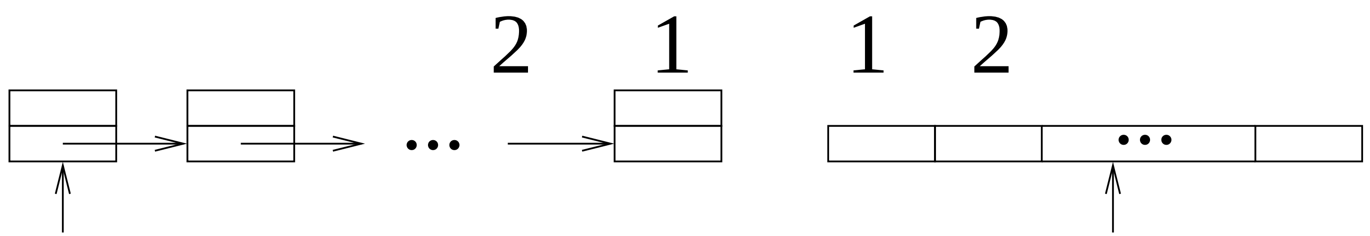
- Weniger Speicherplatz
- Platz ist oft auch Zeit
- Eingeschränkter z.B. kein delete
- Merkwürdige Benutzerschnittstelle, z.B. deleteAfter

Speicherverwaltung für Listen

- kann leicht 90 % der Zeit kosten!
- Lieber Elemente zwischen (Free)lists herschieben als echte `malloc`s
- Alloziere viele Items gleichzeitig
- Am Ende alles freigeben?
- Speichere „parasitär". z.B. Graphen:
  Knotenarray. Jeder Knoten speichert ein ListItem
  $\rightsquigarrow$ Partition der Knoten kann als verkettete Listen gespeichert werden
  $\rightsquigarrow$ MST, shortest Path

Challenge: garbage collection, viele Datentypen
$\rightsquigarrow$ auch ein Software Engineering Problem hier nicht

Beispiel: Stack

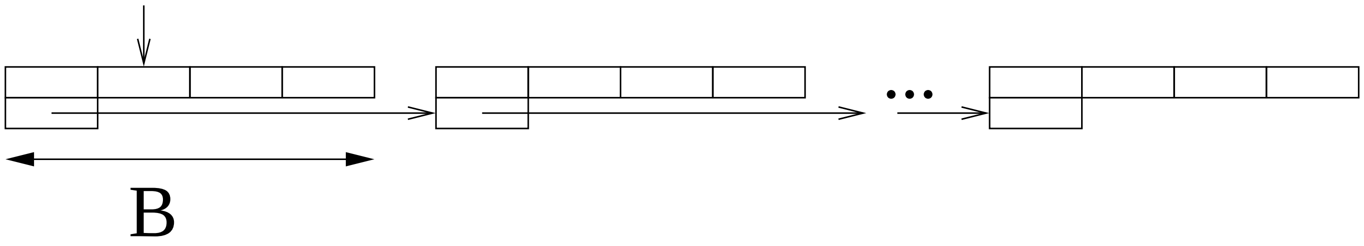


	SList	B-Array	U-Array
dynamisch	+	−	+
Platzverschwendung	pointer	zu gro/3?	zu gro/3?
	freigeben?		
Zeitverschwendung	cache miss	+	umkopieren
worst case time	(+)	+	−

Wars das?

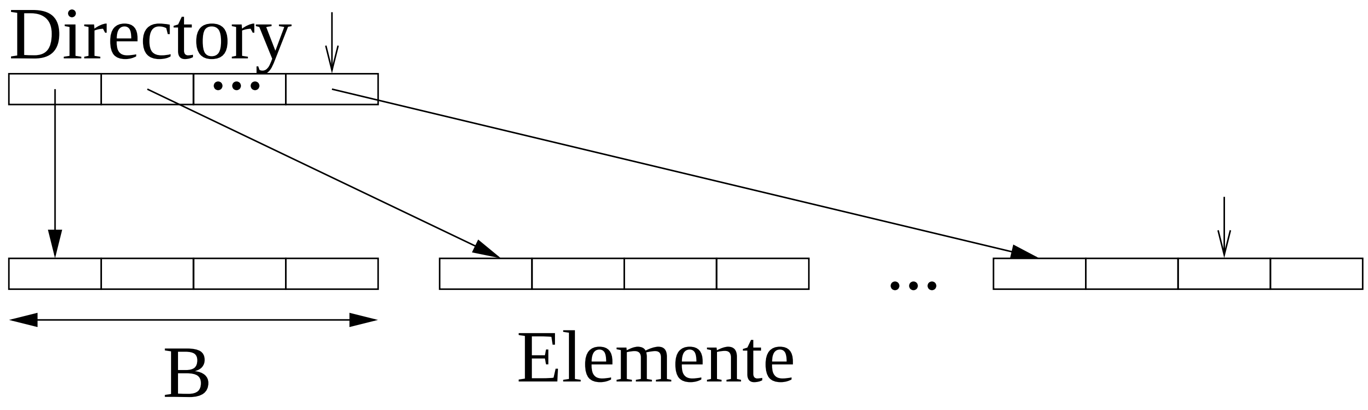
Hat jede Implementierung gravierende Schwächen?

The Best From Both Worlds



	hybrid
dynamisch	$+$
Platzverschwendung	$n/B+B$
Zeitverschwendung	$+$
worst case time	$+$

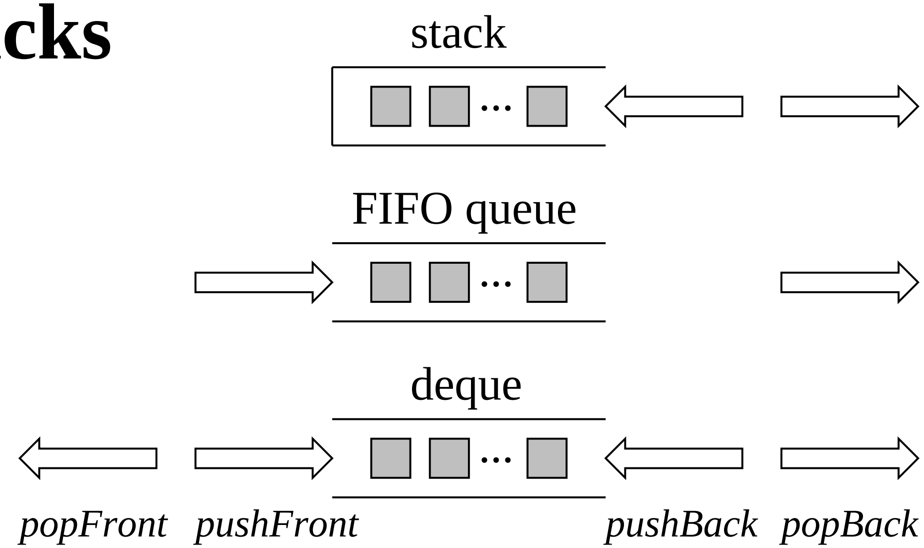
Eine Variante

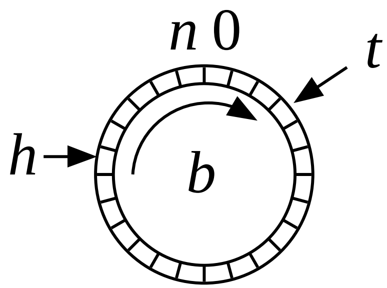


– Reallozierung im top level $\leadsto$ nicht worst case konstante Zeit

+ Indizierter Zugrif auf $S[i]$ in konstanter Zeit

Beyond Stacks





FIFO: BArray $\longrightarrow$ cyclic array

Aufgabe: Ein Array, das “[i]” in konstanter Zeit und einfügen/löschen von Elementen in Zeit $O(\sqrt{n})$ unterstützt

Aufgabe: Ein externer Stack, der n push/pop Operationen mit $O(n/B)$ I/Os unterstützt

Aufgabe: Tabelle für hybride Datenstrukturen vervollständigen

Operation	List	SList	UArray	CArray	explanation of '*'
$[\cdot]$	n	n	1	1	
$\lvert\cdot\rvert$	1^*	1^*	1	1	not with inter-list splice
first	1	1	1	1	
last	1	1	1	1	
insert	1	1^*	n	n	insertAfter only
remove	1	1^*	n	n	removeAfter only
pushBack	1	1	1^*	1^*	amortized
pushFront	1	1	n	1^*	amortized
popBack	1	n	1^*	1^*	amortized
popFront	1	1	n	1^*	amortized
concat	1	1	n	n	
splice	1	1	n	n	
findNext,...	n	n	n^*	n^*	cache efficient

Was fehlt?

Fakten Fakten Fakten

Messungen für

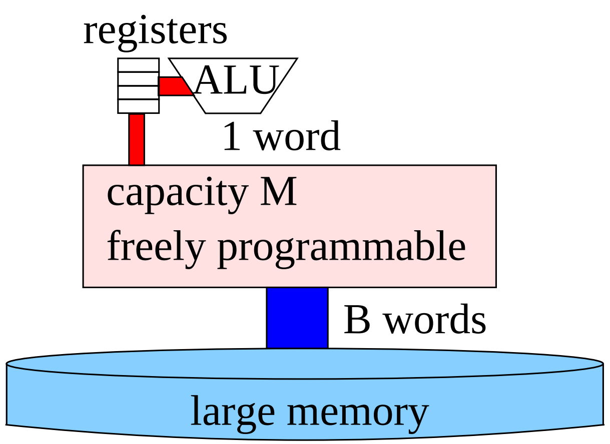
- Verschiedene Implementierungsvarianten
- Verschiedene Architekturen
- Verschiedene Eingabegrößen
- Auswirkungen auf reale Anwendungen
- Kurven dazu
- Interpretation, ggf. Theoriebildung

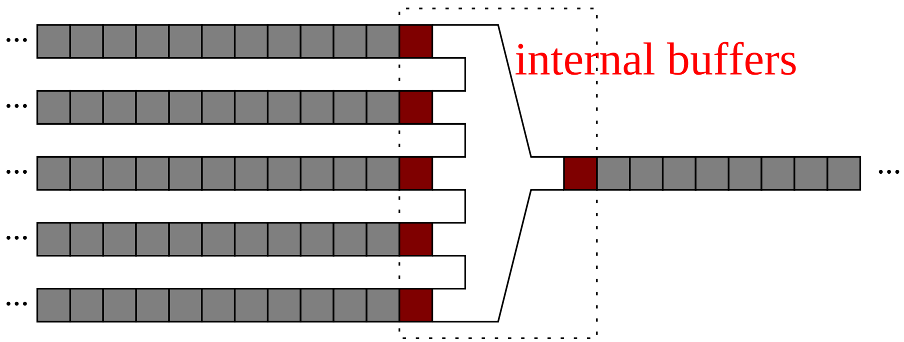
Aufgabe: Array durchlaufen versus zufällig allozierte verkettete Liste

Sorting by Multiway Merging

- Sort $\lceil n/M \rceil$ runs with M elements each $2n/B$ I/Os
- Merge M/B runs at a time $2n/B$ I/Os
- until only one run is left $\times \left\lceil \log_{M/B} \frac{n}{M} \right\rceil$ merge phases

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





Procedure twoPassSort($M : \mathbb{N}$; a : external **Array** $[0..n-1]$ **of** Element)

b : external **Array** $[0..n-1]$ **of** Element // auxiliary storage

formRuns(M, a, b)

mergeRuns(M, b, a)

// Sort runs of size M from f writing sorted runs to t

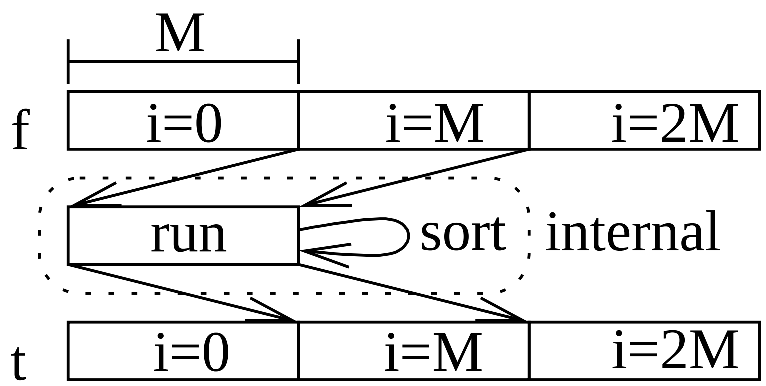
Procedure formRuns($M : \mathbb{N}$; f, t : external **Array** $[0..n-1]$ **of** Element)

for $i := 0$ **to** $n-1$ **step** M **do**

run := $f[i..i+M-1]$

sort(run)

$t[i..i+M-1]$:= run



//Merge n elements from f to t where f stores sorted runs of size L

Procedure mergeRuns($L : \mathbb{N}$; f, t : external **Array** $[0..n-1]$ **of** Element)

$k := \lceil n/L \rceil$ // Number of runs

next : PriorityQueue **of** Key $\times \mathbb{N}$

runBuffer := **Array** $[0..k-1][0..B-1]$ **of** Element

for $i := 0$ **to** $k-1$ **do**

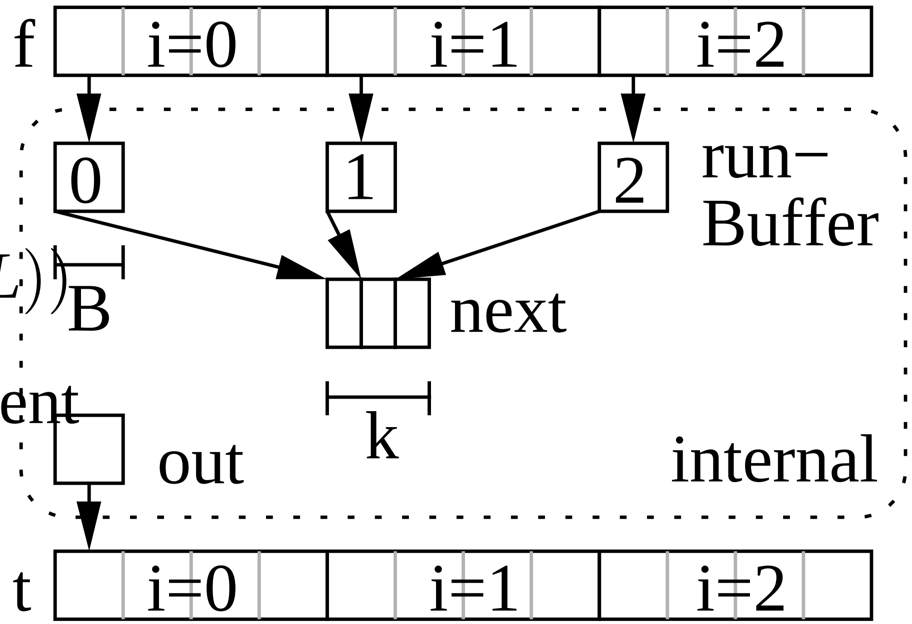
runBuffer$[i]$:=

$f[iM..iM+B-1]$

next.insert(

key(runBuffer$[i][0]$), iL))

out : **Array** $[0..B-1]$ **of** Element



// k-way merging

for $i := 0$ **to** $n-1$ **step** B **do**

  for $j := 0$ **to** $B-1$ **do**

    $(x, \ell) :=$ next.deleteMin

    out$[j] :=$ runBuffer$[\ell$ **div** $L][\ell$ **mod** $B]$

    $\ell{+}{+}$

    if ℓ **mod** $B = 0$ **then** // New input block

      if ℓ **mod** $L = 0 \vee \ell = n$ **then**

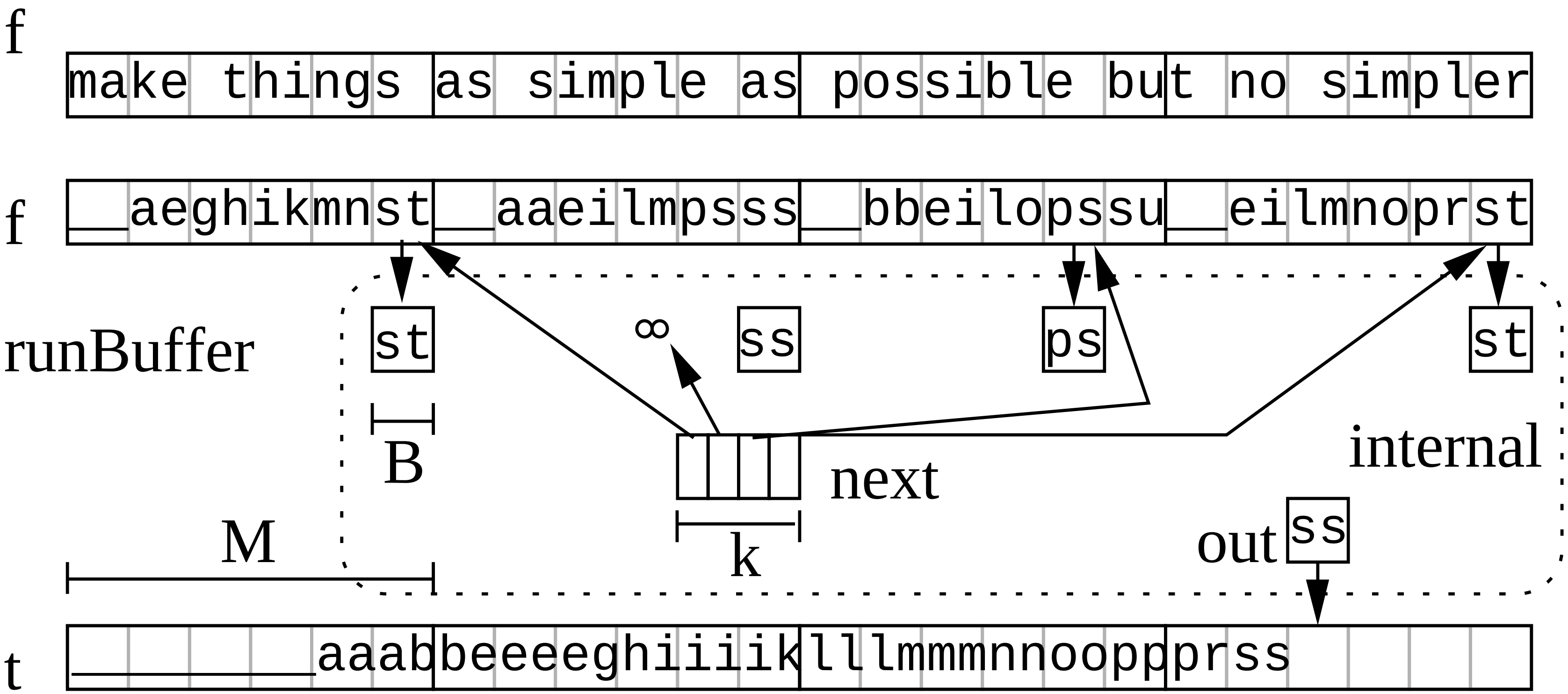
        runBuffer$[\ell$ **div** $L][0] := \infty$// sentinel for exhausted ru

      else runBuffer$[\ell$ **div** $L] := f[\ell..\ell+B-1]$// refill buffer

    next.insert$(($runBuffer$[\ell$ **div** $L][0], \ell))$

  write out to $t[i..i+B-1]$ // One output step

Example, $B = 2$, run size $= 6$



2 Super Scalar Sample Sort

Comparison Based Sorting

Large data sets (thousands to millions)

Theory: Lots of algorithms with $n\log n + O(n)$ comparisons

Practice: Quicksort

+ $n\log n + O(n)$ expected comparisons
  using sample based pivot seletction
+ sufficiently cache efficient
  $O\left(\frac{n}{B}\log n\right)$ cache misses on all levels
+ Simple
+ Compiler writers are aware of it

Can we nevertheless beat quicksort?

Quicksort

Function quickSort(s : Sequence **of** Element) : Sequence **of** Element
  if $|s| \leq 1$ **then return** s // base case
  pick $p \in s$ uniformly at random // pivot key
  a := $\langle e \in s : e < p \rangle$
  b := $\langle e \in s : e = p \rangle$
  c := $\langle e \in s : e > p \rangle$
  return concatenate$(\text{quickSort}(a), b, \text{quickSort}(c))$

$\leq 1.4 n \log n$ erwartete (3-Wege)-Vergleiche

$\longrightarrow n \log n$ wenn pivot = median eines $\omega(1)$ random sample

Engineering Quicksort

- array
- 2-Wege-Vergleiche
- sentinels für innere Schleife
- inplace swaps
- Rekursion auf kleinere Teilproblemgröße
  $\rightarrow O(\log n)$ zusätzlichen Platz
- Rekursionsabbruch für kleine (20–100) Eingaben, insertion sort
  (nicht ein großer insertion sort)

Procedure qSort(a : **Array of** Element; $\ell, r : \mathbb{N}$) // Sort $a[\ell..r]$
  while $r - \ell \geq n_0$ **do** // Use divide-and-conquer
    $j :=$ pickPivotPos(a, l, r)
    swap$(a[\ell], a[j])$ // Helps to establish the invariant
    $p := a[\ell]$
    $i := \ell$; $j := r$
    repeat // a: ℓ $i\rightarrow$ $\leftarrow j$ r
      while $a[i] < p$ **do** i++ // Scan over elements (A)
      while $a[j] > p$ **do** j−− // on the correct side (B)
      if $i \leq j$ **then** swap$(a[i], a[j])$; i++ ; j−−
    until $i > j$ // Done partitioning
    if $i < \frac{l+r}{2}$ **then** qSort(a,ℓ,j); $\ell := j + 1$
    else qSort(a,i,r) ; $r := i - 1$
  insertionSort$(a[l..r])$ // faster for small $r - l$

Previous Work

Integer Keys

+ Can be $2-3$ times faster than quicksort
– Naive ones are cache inefficient and slower than quicksort
– Simple ones are distribution dependent.

Cache efficient sorting

k-ary merge sort

[Nyberg et al. 94, Arge et al. 04, Ranade et al. 00, Brodal et al. 04]

+ Faktor $\log k$ less cache faults
– Only ≈ 20 % speedup, and only for laaarge inputs

Sample Sort

Function sampleSort($e = \langle e_1, \ldots, e_n \rangle, k$)

  if n/k is "small" **then return** smallSort(e)

  let $\langle S_1, \ldots, S_{ak-1} \rangle$ denote a random sample of e

  sort S

  $\langle s_0, s_1, s_2, \ldots, s_{k-1}, s_k \rangle :=$

  $\langle -\infty, S_a, S_{2a}, \ldots, S_{(k-1)a}, \infty \rangle$

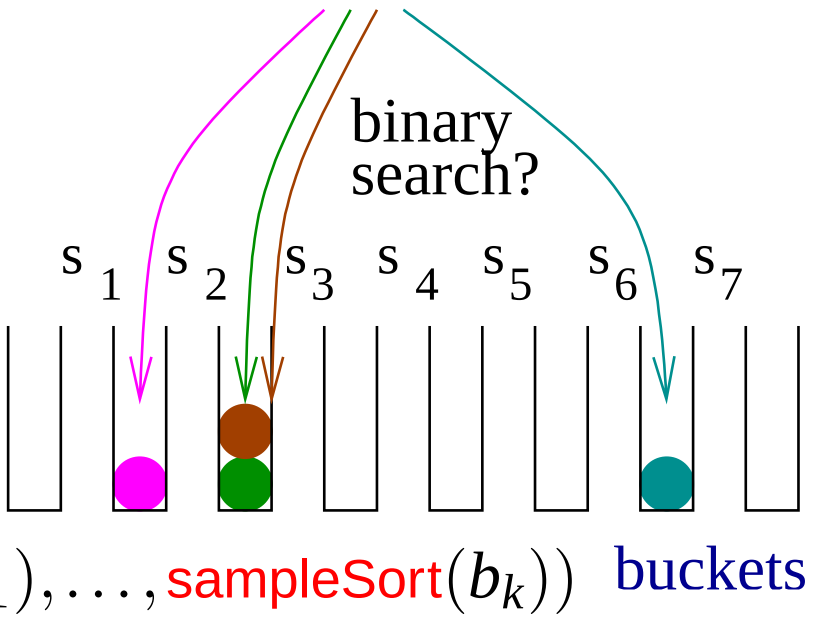
  for $i := 1$ **to** n **do**

    find $j \in \{1, \ldots, k\}$

      such that $s_{j-1} < e_i \leq s_j$

    place e_i in bucket b_j

  return concatenate(sampleSort(b_1), ..., sampleSort(b_k))



Why Sample Sort?

- □ traditionally: parallelizable on coarse grained machines
- + Cache efficient ≈ merge sort
- − Binary search not much faster than merging
- − complicated memory management

Super Scalar Sample Sort

- □ Binary search ⟶ implicit search tree
- □ Eliminate all conditional branches
- ⇝ Exploit instruction parallelism
- ⇝ Cache efficiency comes to bear
- □ "steal" memory management from radix sort

Classifying Elements

$t := \langle s_{k/2}, s_{k/4}, s_{3k/4}, s_{k/8}, s_{3k/8}, s_{5k/8}, s_{7k/8}, \ldots \rangle$

for $i := 1$ **to** n **do**

$j := 1$

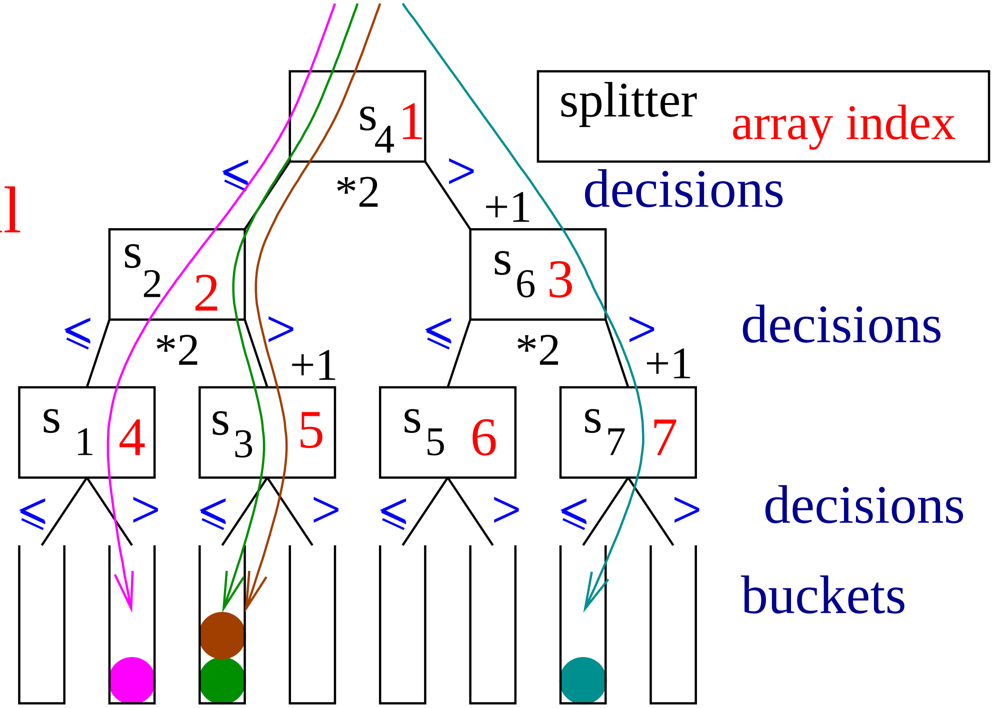
repeat $\log k$ times // unroll

$j := 2j + (a_i > t_j)$

$j := j - k + 1$

$|b_j|$++

o(i):= j // oracle



Now the compiler should:

- use predicated instructions
- interleave for loop iterations (unrolling $\vee$ software pipelining)

Predication

Hardware mechanism that allows instructions to be conditionally executed

- □ Boolean predicate registers (1–64) hold condition codes
- □ predicate registers p are additional inputs of predicated instructions I
- □ At runtime, I is executed if and only if p is true
- \+ Avoids branch misprediction penalty
- \+ More flexible instruction scheduling
- − Switched off instructions still take time
- − Longer opcodes
- − Complicated hardware design

Example (IA-64)

Translation of: if (r1 > 2) r3 := r3 + 4

With a conditional branch:

```
      cmp.gt p6,p7=r1,r2
(p7) br.cond .label
      add r3=4,r3
.label:
```

Via predication:

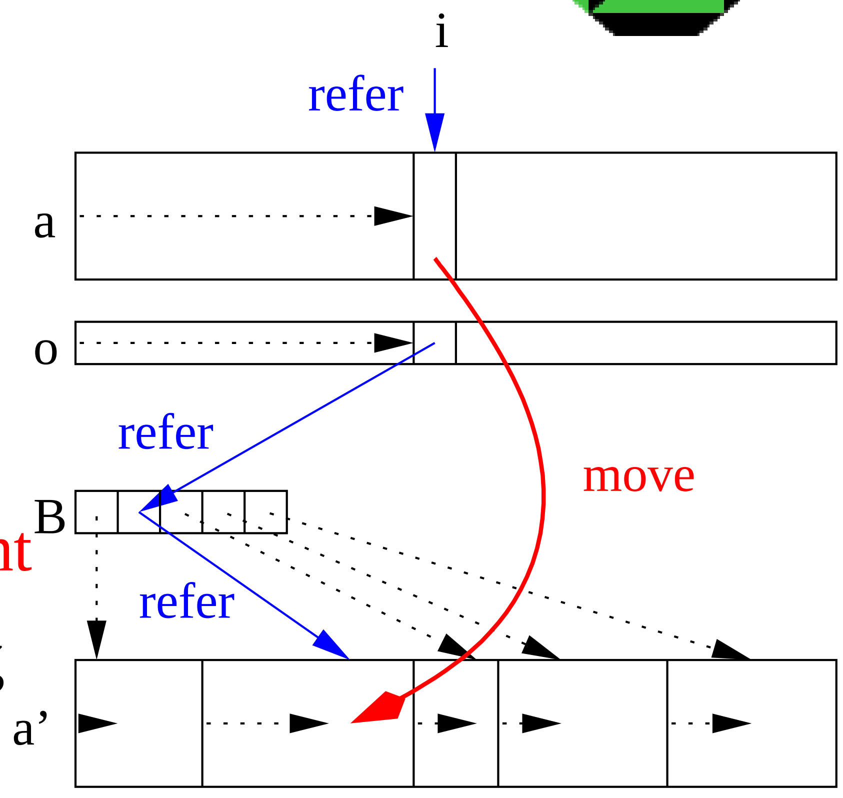
```
      cmp.gt p6,p7=r1,r2
(p6) add r3=4,r3
```

```
(...)
```

Other Current Architectures:

Conditional moves only

Distributing Elements



for $i := 1$ **to** n **do** $a'_{B[o(i)]++} := a_i$

Why Oracles?

- simplifies memory management
- no overflow tests or re-copying
- simplifies software pipelining
- separates computation and memory access aspects
- small (n bytes)
- sequential, predictable memory access
- can be hidden using prefetching / write buffering

Aufgabe

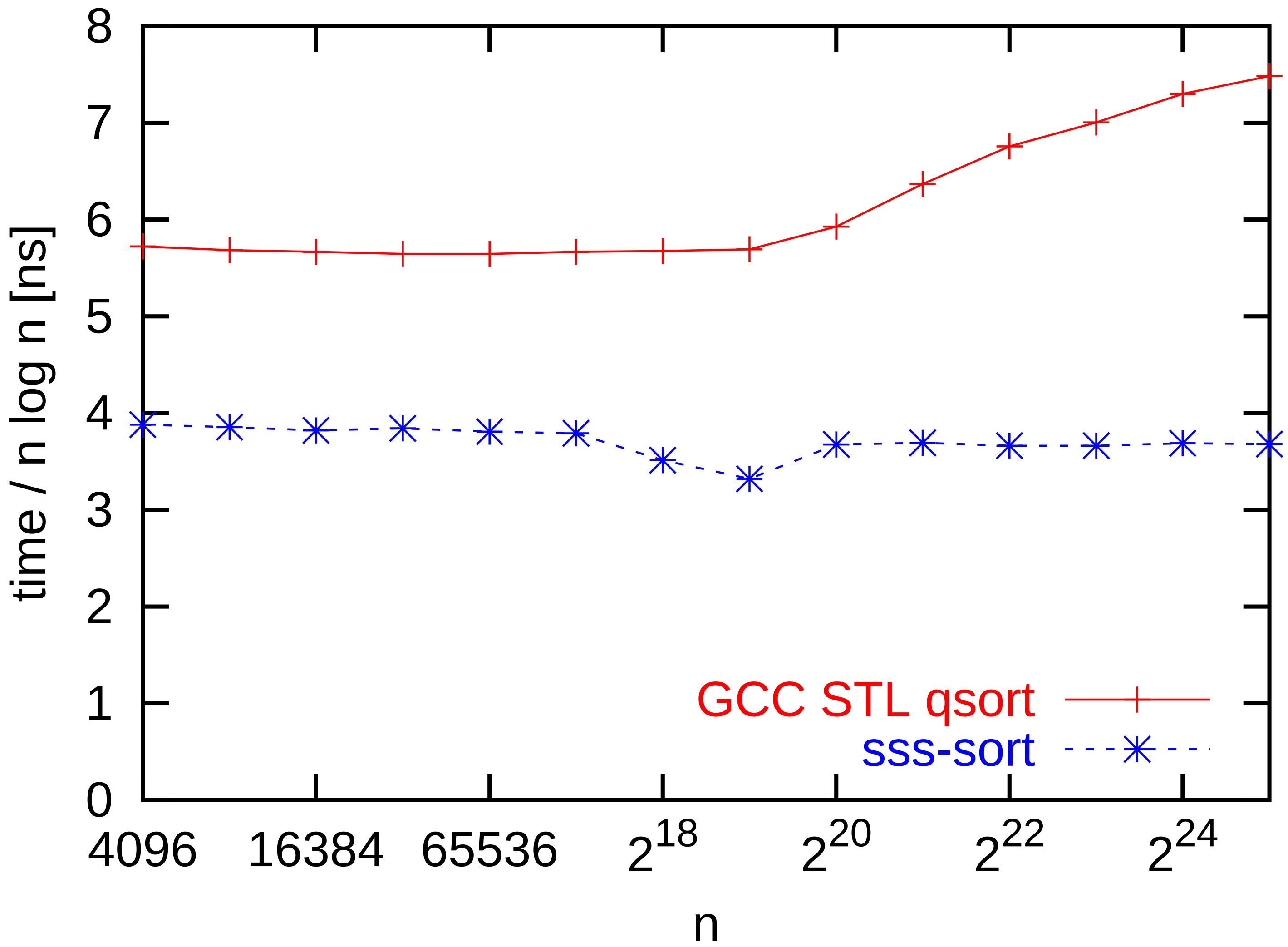
Wie implementiert man 4-Wege Mischen (8-Wege-Mischen) mit

- $2n$ Speicherzugriffen und
- $2n$ $(3n)$ Vergleichen
- auf einer Maschine mit 8 (16) Registern?

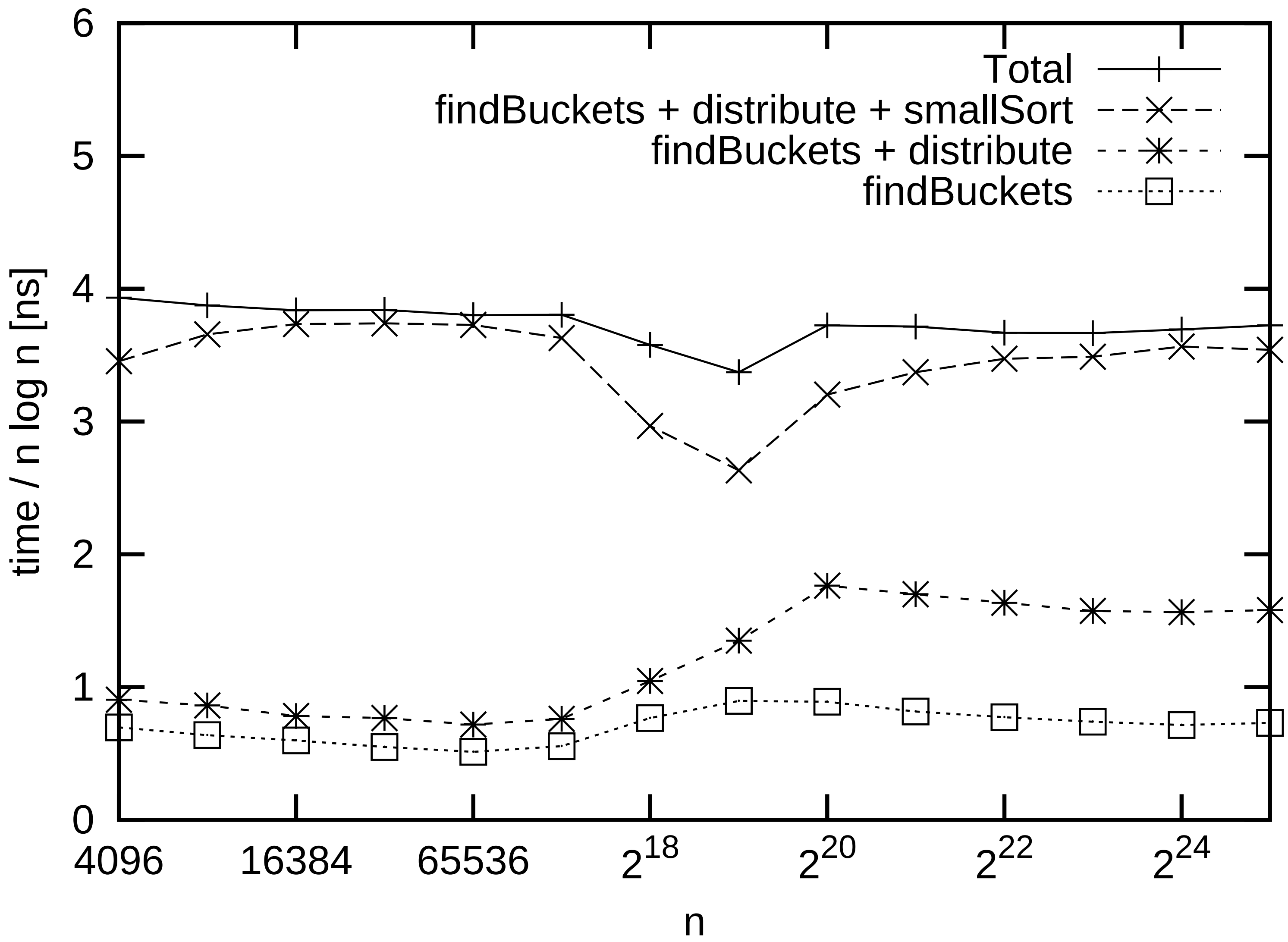
Experiments: 1.4 GHz Itanium 2

- `restrict` keyword from ANSI/ISO C99 to indicate nonaliasing
- Intel's C++ compiler v8.0 uses predicated instructions automatically
- Profiling gives 9% speedup
- $k = 256$ splitters
- Use `stl:sort` from g++ ($n \leq 1000$)!
- insertion sort for $n \leq 100$
- Random 32 bit integers in $[0, 10^9]$

Comparison with Quicksort



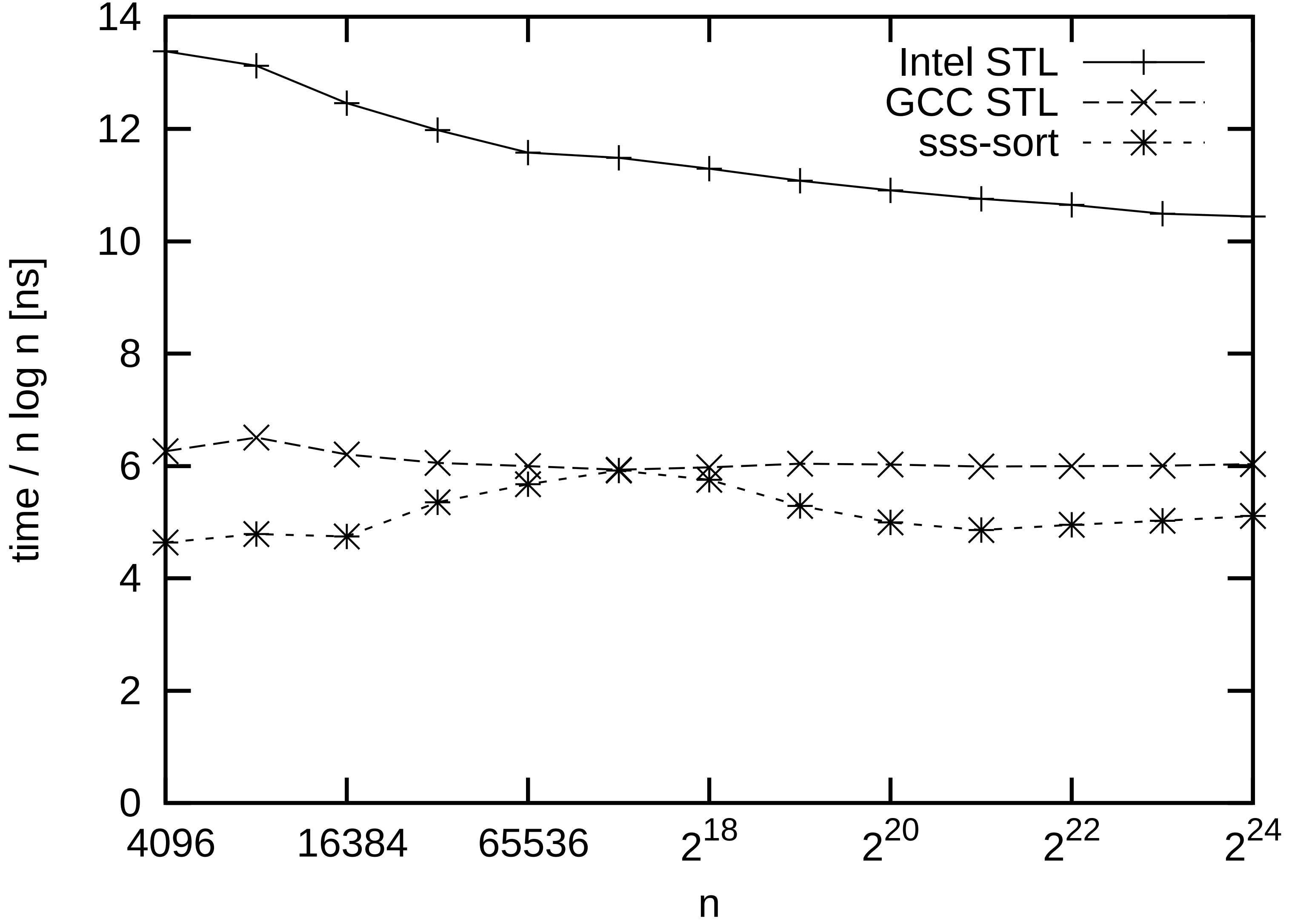
Breakdown of Execution Time



A More Detailed View

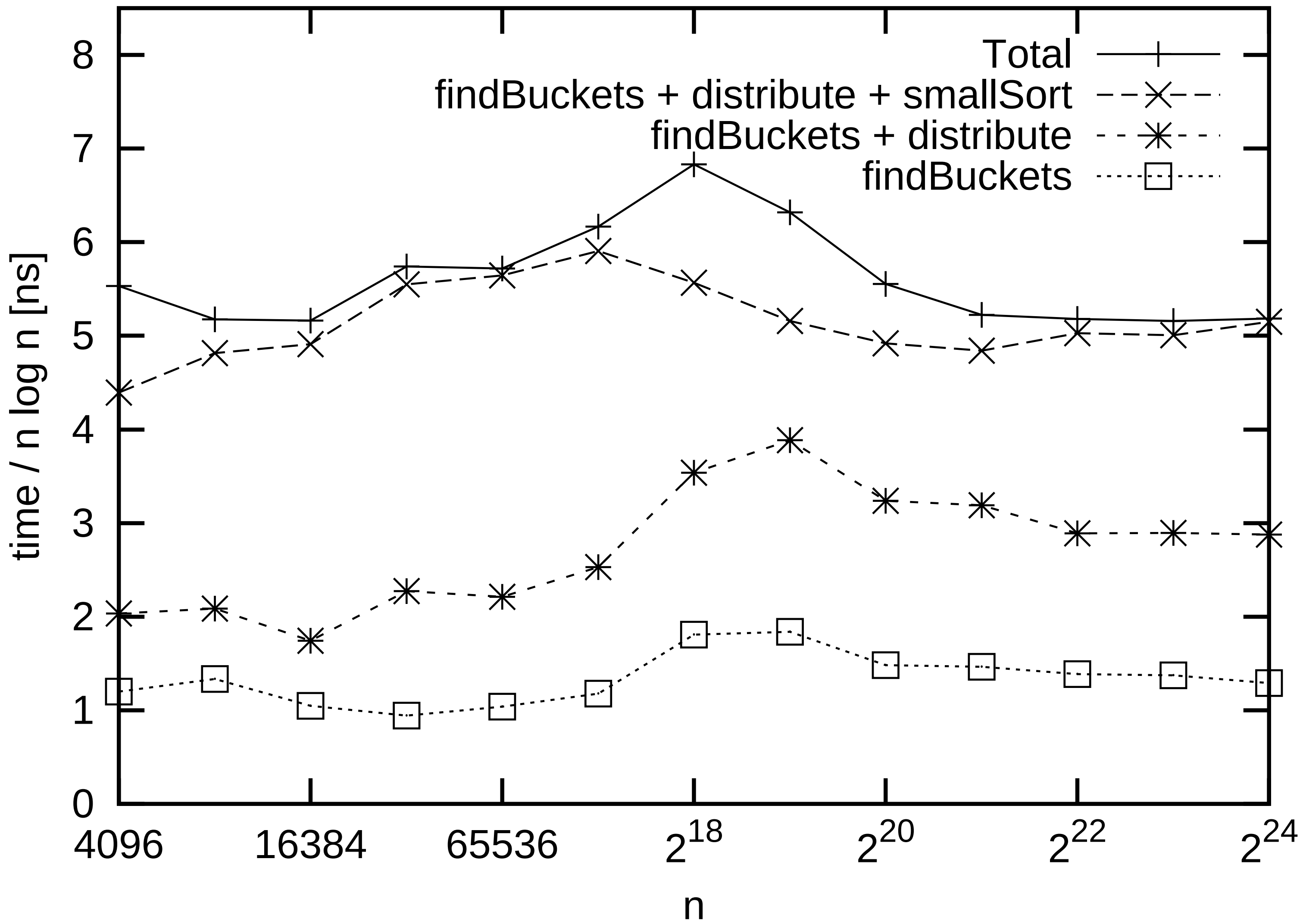
	instr.	cycles	dynamic IPC small n	dynamic IPC $n = 2^{25}$
findBuckets, $1\times$ outer loop	63	11	5.4	4.5
distribute, one element	14	4	3.5	0.8

Comparison with Quicksort Pentium 4



Problems: few registers, one condition code only, compiler needs “help”

Breakdown of Execution Time Pentium 4



Analysis

	mem. acc.	branches	data dep.	I/Os	registers	instructions
k-way distribution:						
sss-sort	$n\log k$	$O(1)$	$O(n)$	$3.5n/B$	$3\times$unroll	$O(\log k)$
quicksort $\log k$ lvls.	$2n\log k$	$n\log k$	$O(n\log k)$	$2\frac{n}{B}\log k$	4	$O(1)$
k-way merging:						
memory	$n\log k$	$n\log k$	$O(n\log k)$	$2n/B$	7	$O(\log k)$
register	$2n$	$n\log k$	$O(n\log k)$	$2n/B$	k	$O(k)$
funnel $k'^{\log_{k'} k}$	$2n\log_{k'} k$	$n\log k$	$O(n\log k)$	$2n/B$	$2k'+2$	$O(k')$

Conclusions

- sss-sort up to twice as fast as quicksort on Itanium
- comparisons $\neq$ conditional branches
- algorithm analysis is not just instructions and caches

Kritik I

Warum nur zufällige Schlüssel?

Antwort I

Sample sort ist kaum von der Verteilung der Eingabe abhängig

Kritik I´

Aber was wenn es viele gleiche Schlüssel gibt?

Die landen alle in einem Bucket

Antwort I´

Its not a bug its a feature:

Sei $s_i = s_{i+1} = \cdots = s_j$ Indikator für häufigen Schlüssel!

Setze $s_i := \min\{x \in Key : x < s_i\}$,

(optional: streiche $s_{i+2}, \ldots s_j$)

Nun muss bucket $i+1$ nicht mehr sortiert werden!

todo: implementieren

Kritik II

Quicksort ist inplace

Antwort II

Benutze hybride Listen-Array Repräsentation von Folgen
braucht $O\left(\sqrt{kn}\right)$ extra Platz für k-way sample sort

Kritik II´

Ich will aber Arrays für Ein- und Ausgabe

Antwort II´

Inplace Konvertierung

Eingabe: einfach

Ausgabe: tricky. Aufgabe: grobe Idee entwickeln. Hinweis: Permutation von Blöcken. Jede Permutation besteht aus einem Produkt zyklischer Permutationen. Inplace zyklische Permutation ist einfach

Future Work

- high level fine-tuning, e.g., clever choice of k
- other architectures, e.g., Opteron, PowerPC
- almost in-place implementation
- multilevel cache-aware or cache-oblivious generalization (oracles help)

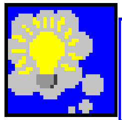
Studienarbeit, (Diplomarbeit), Hiwi-Job?

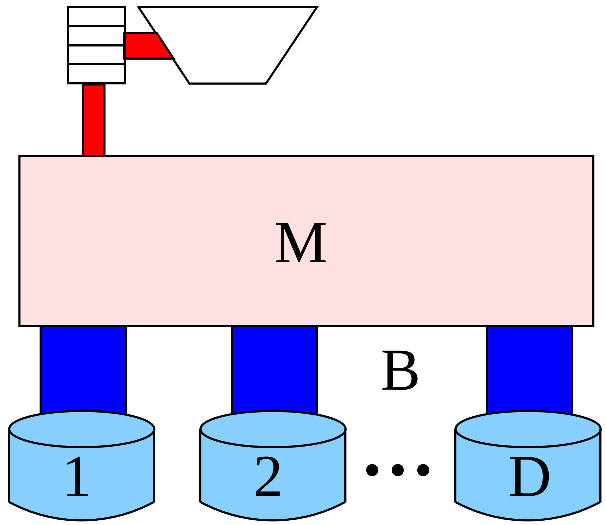
3 Sorting with Parallel Disks

I/O Step := Access to a single physical block per disk

Theory: Balance Sort [Nodine Vitter 93].

Deterministic, complex

asymptotically optimal  $O(\cdot)$



independent disks

[Vitter Shriver 94]

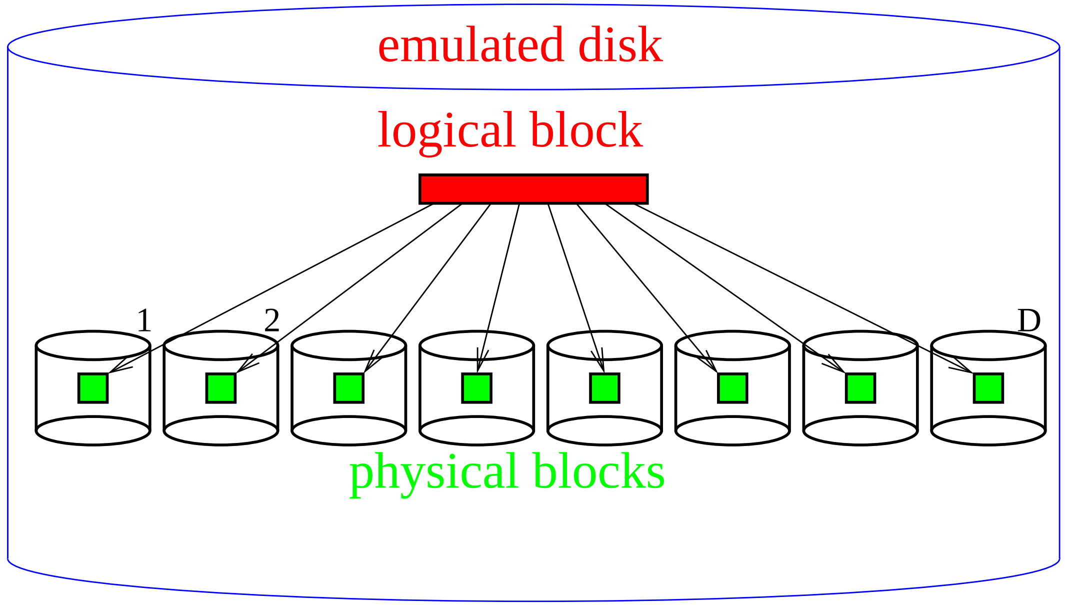
Multiway merging

"Usually" factor 10? less I/Os.

Not asymptotically optimal. 42%

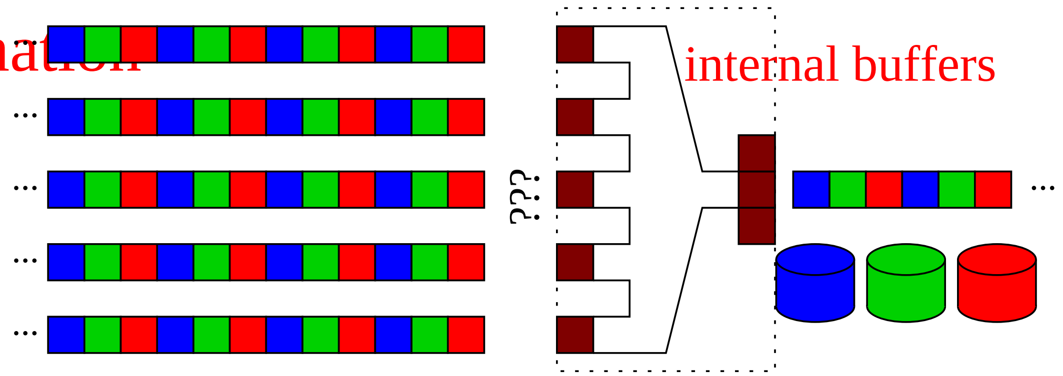
Basic Approach: Improve Multiway Merging

Striping



That takes care of run formation

and writing the output



But what about merging?

Prediction

[Folklore, Knuth]

Smallest Element

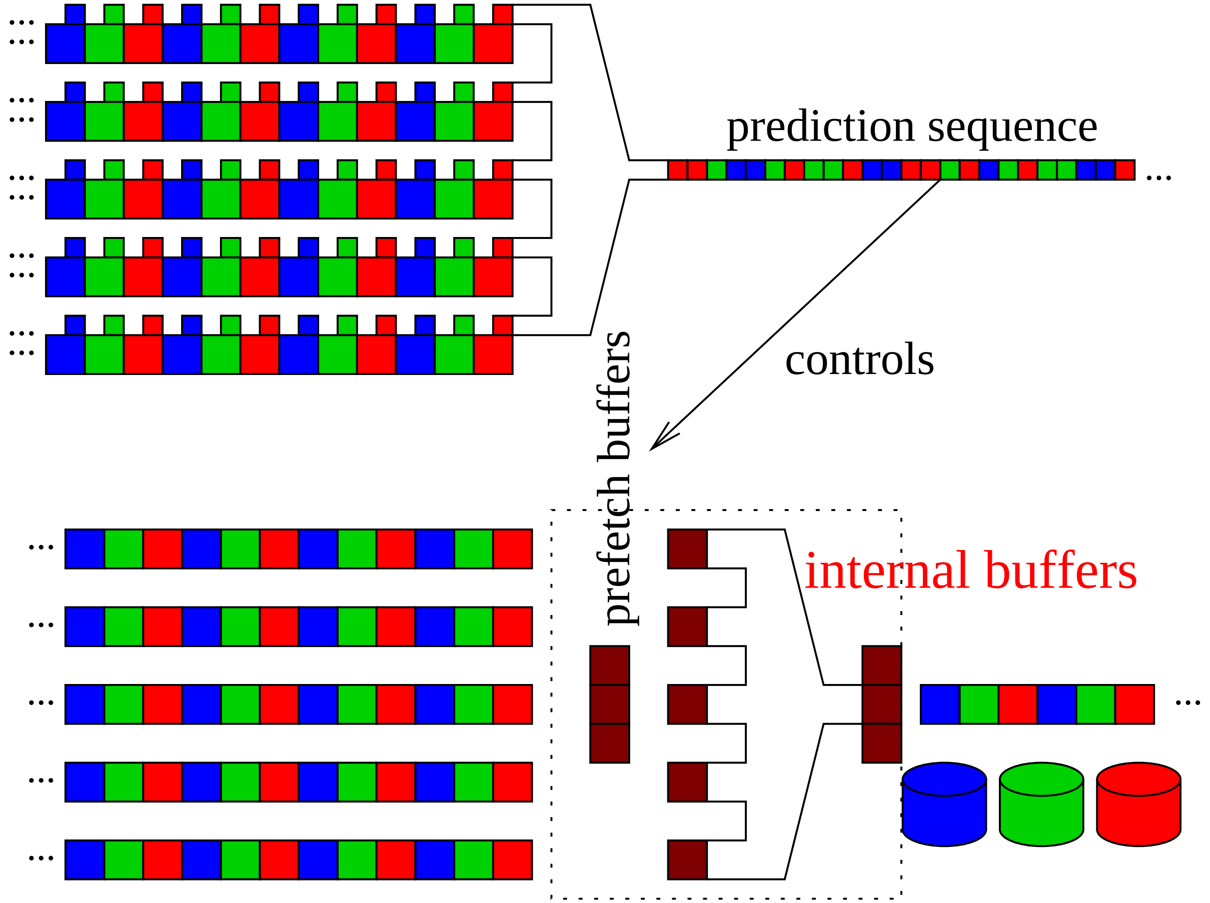
of each block

triggers fetch.

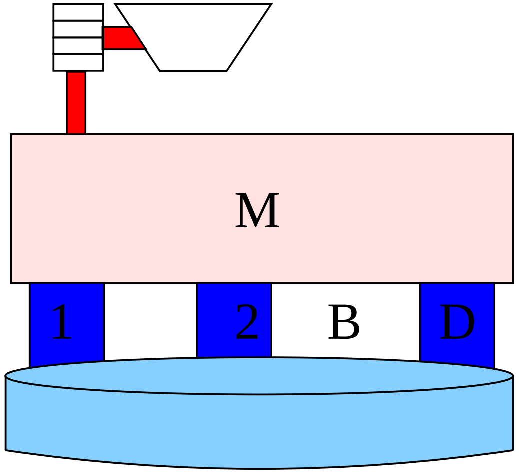
Prefetch buffers

allow parallel access

of next blocks

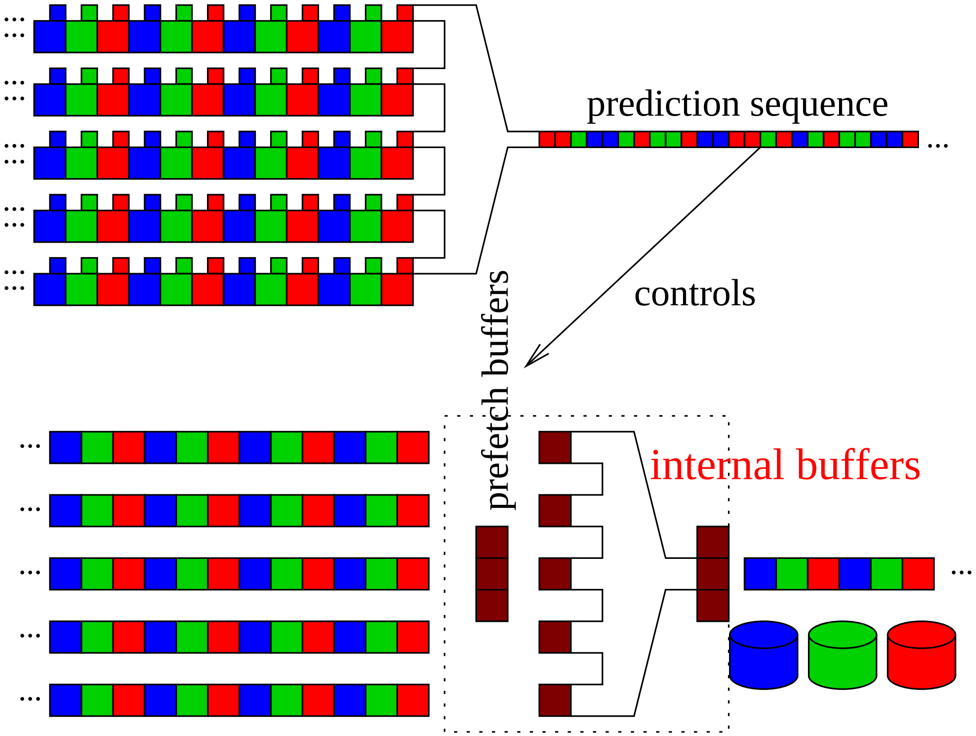


Warmup: Multihead Model



Multihead Model

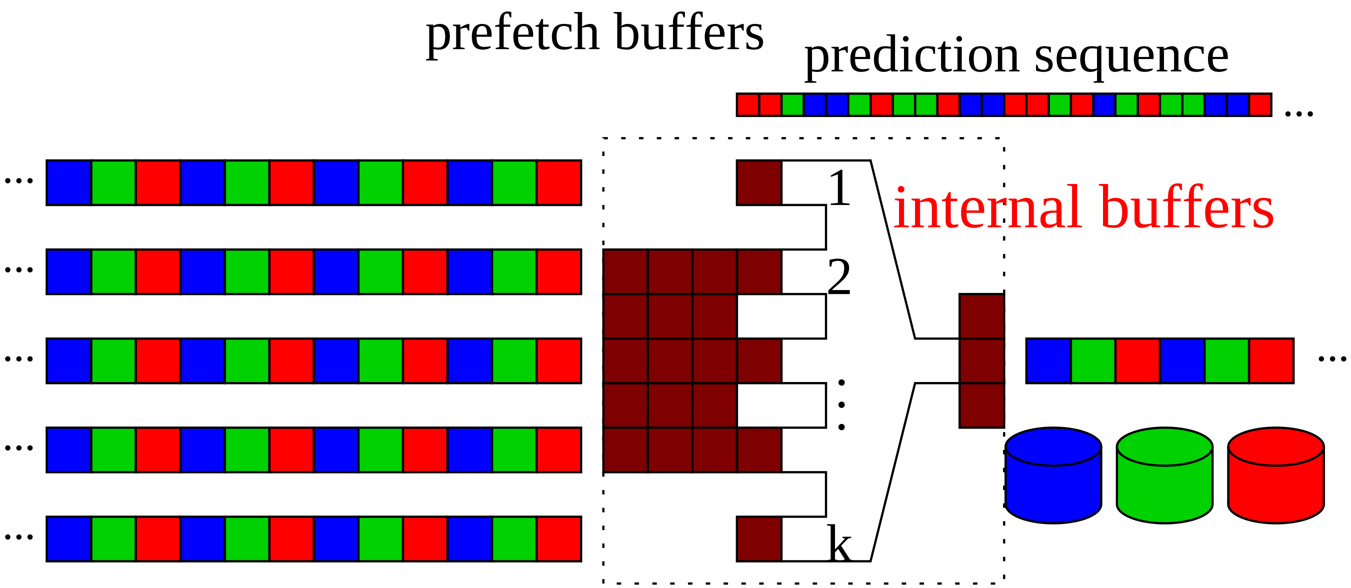
[Aggarwal Vitter 88]



D prefetch buffers yield an optimal algorithm

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

Bigger Prefetch Buffer



$Dk \rightsquigarrow$ good deterministic performance

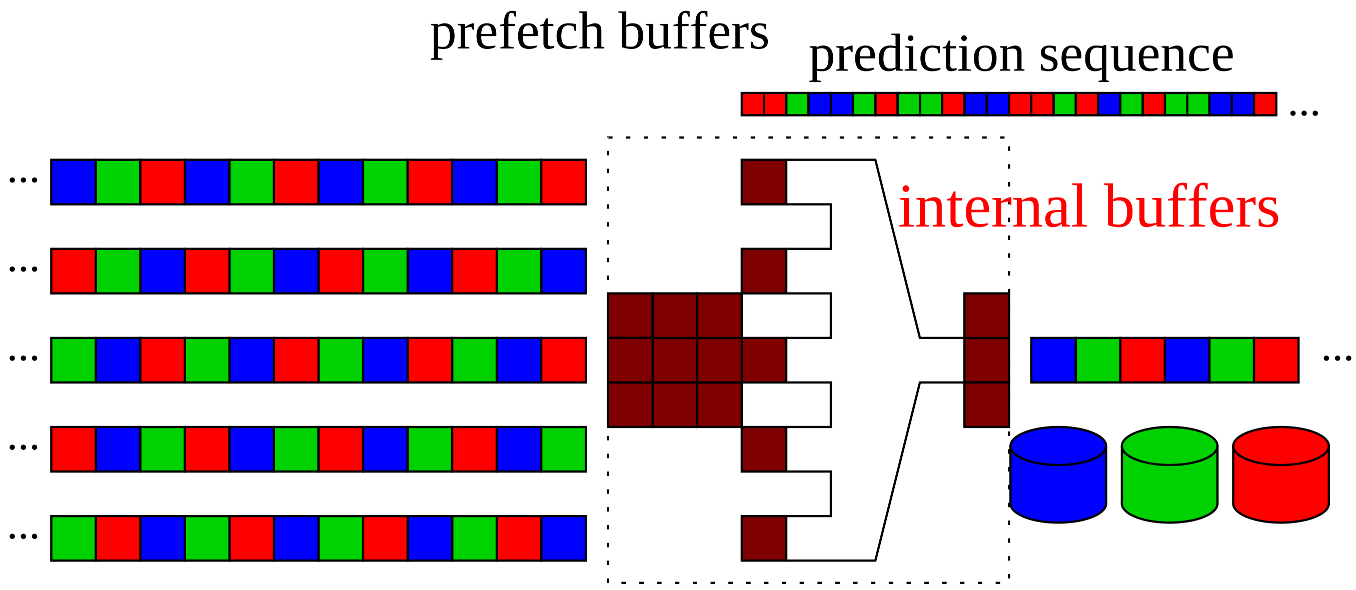
$O(D)$ would yield an optimal algoritihm.

Possible?

Randomized Cycling

[Vitter Hutchinson 01]

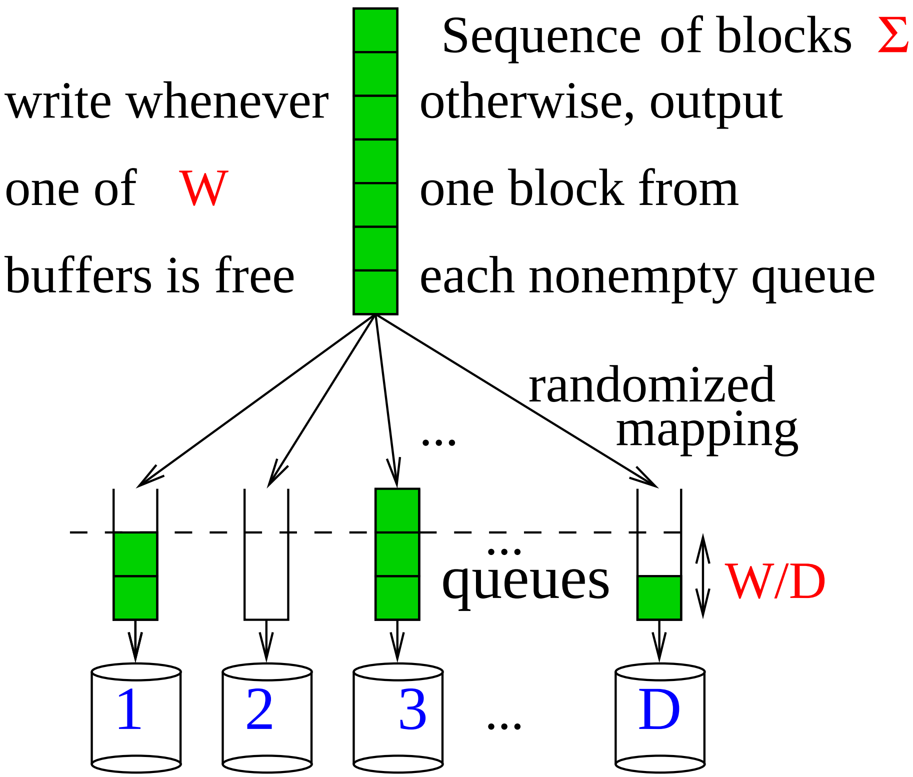
Block i of stripe j goes to disk $\pi_j(i)$ for a random permutation π_j



Good for naive prefetching and $\Omega(D \log D)$ buffers

Buffered Writing

[Sanders-Egner-Korst SODA00, Hutchinson Sanders Vitter ESA 01]



Theorem: Buffered Writing is optimal

...

But how good is optimal?

Theorem: Randomized cycling achieves efficiency $1 - O(D/W)$.

Analysis: negative association of random variables, application of queueing theory to a “throttled” Alg.

Optimal Offline Prefetching

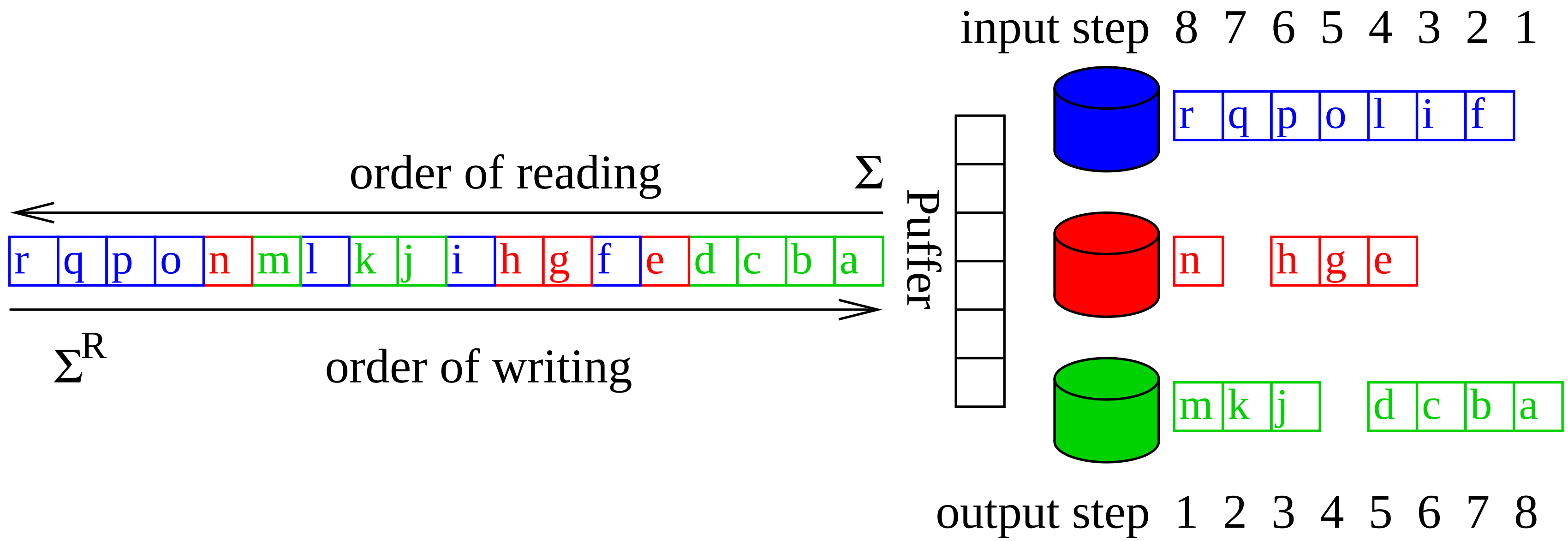
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

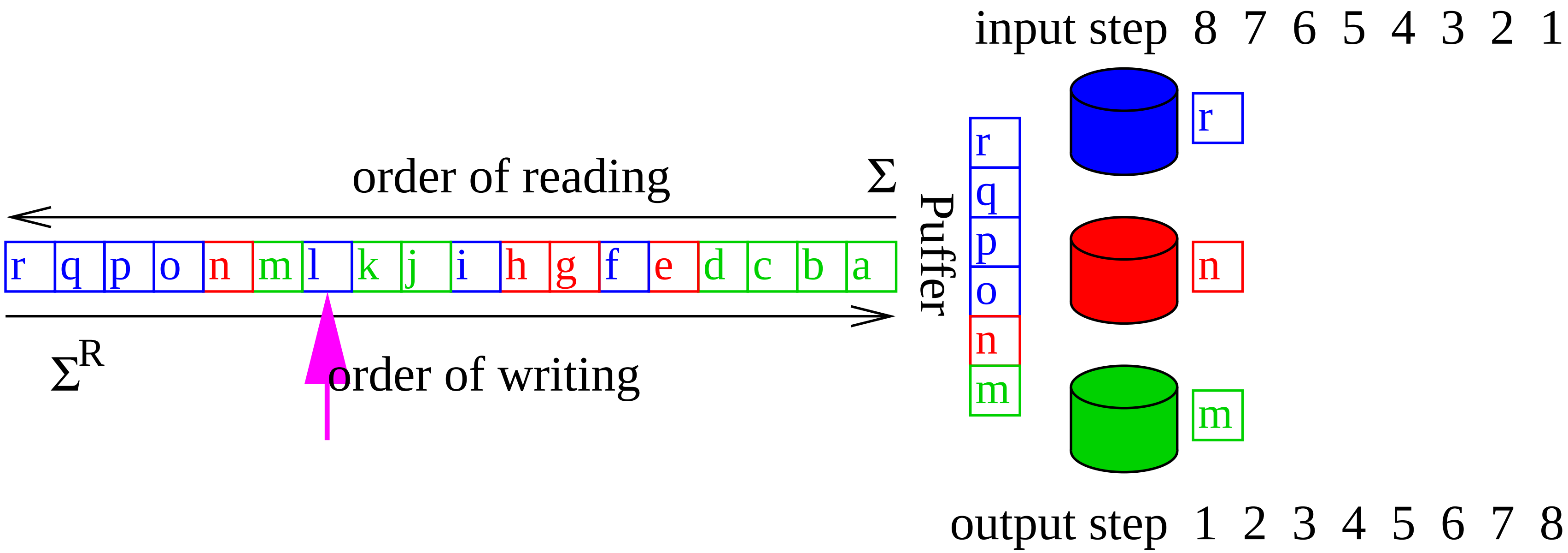
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

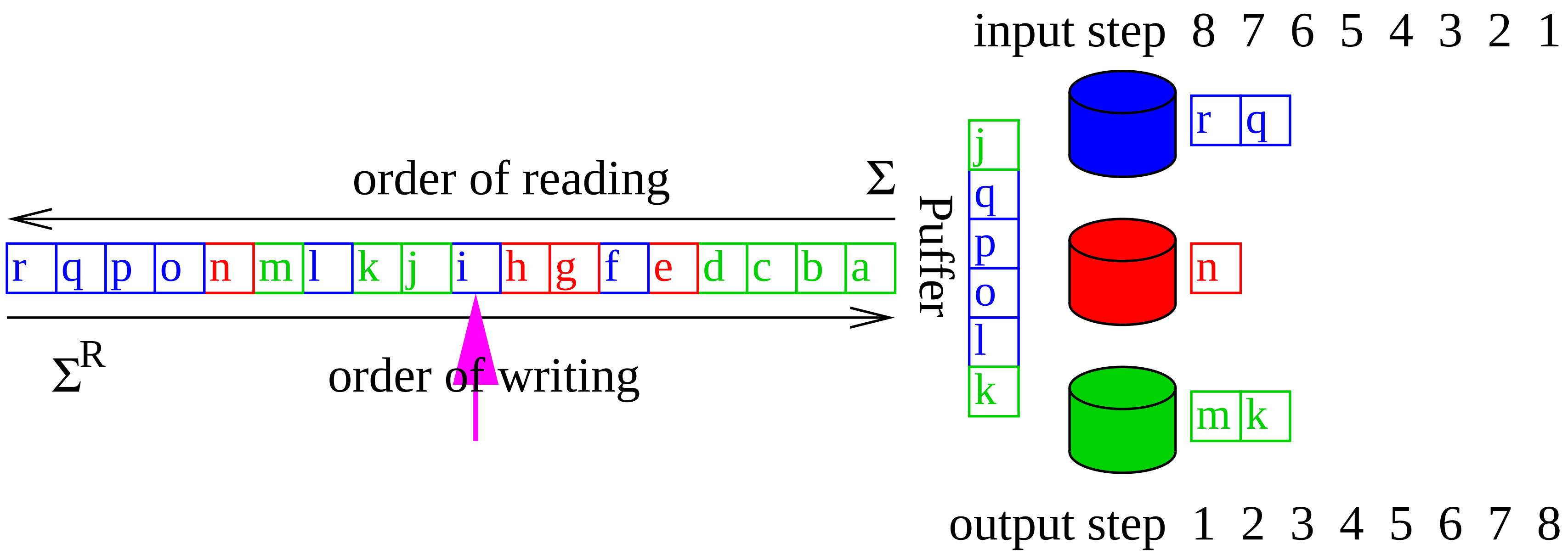
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

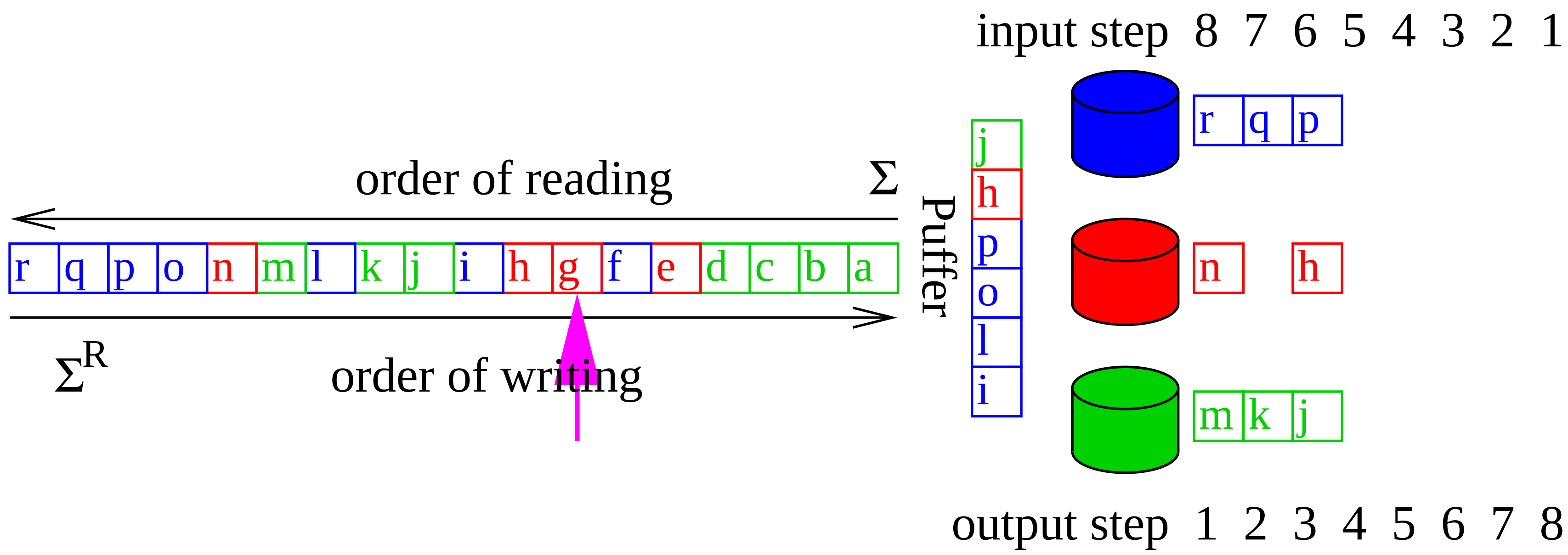
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

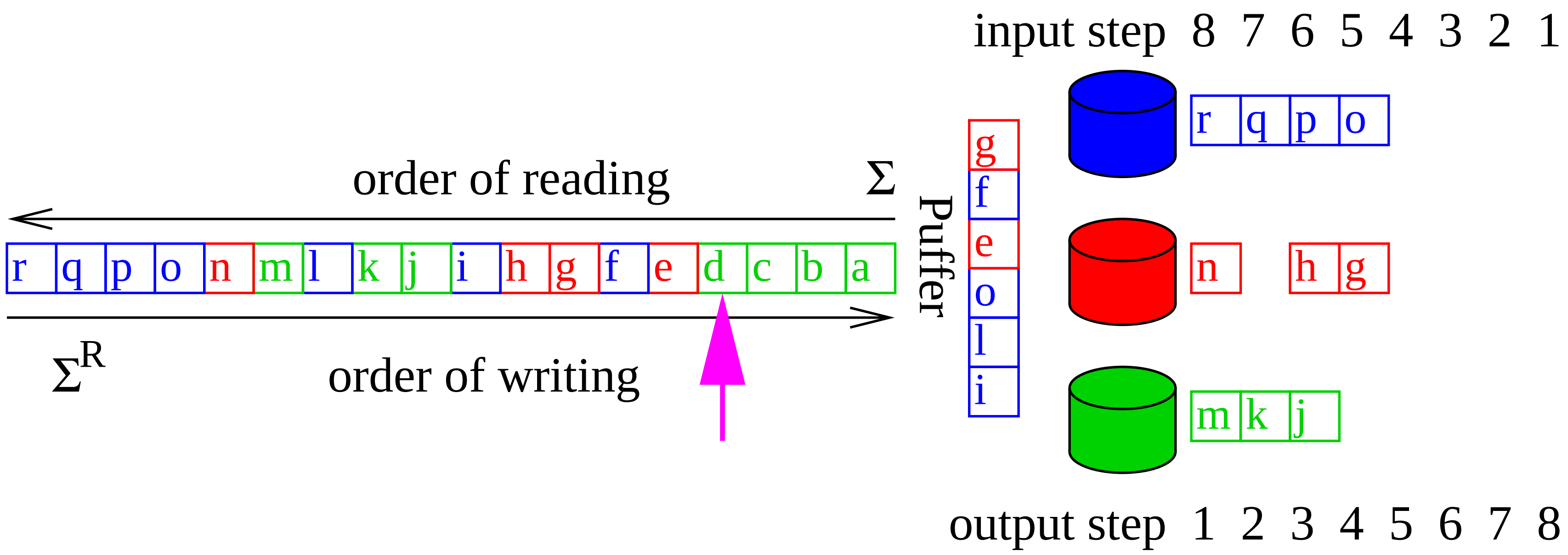
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

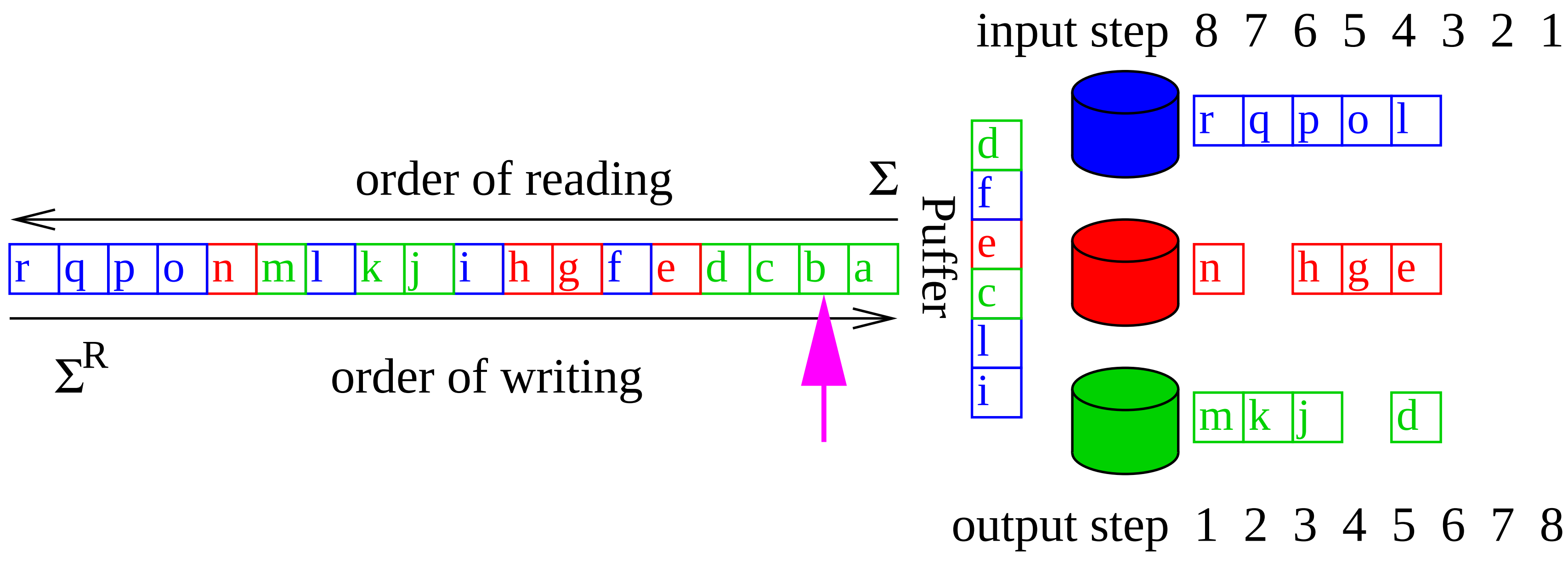
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

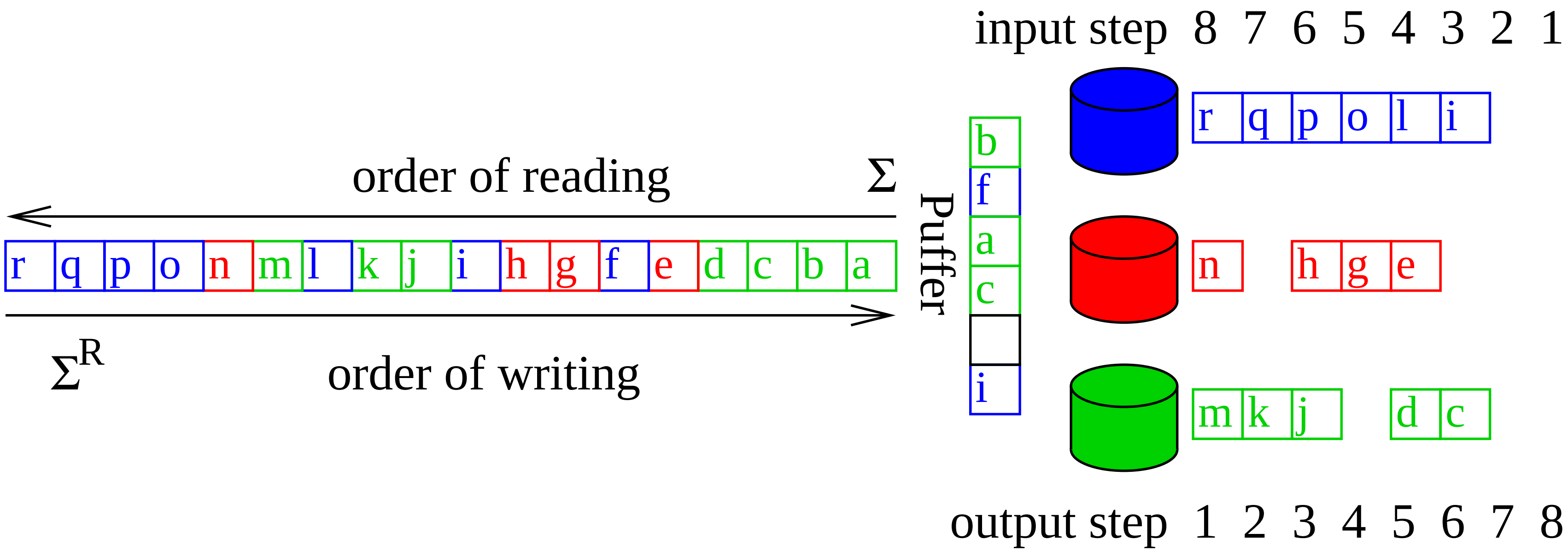
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

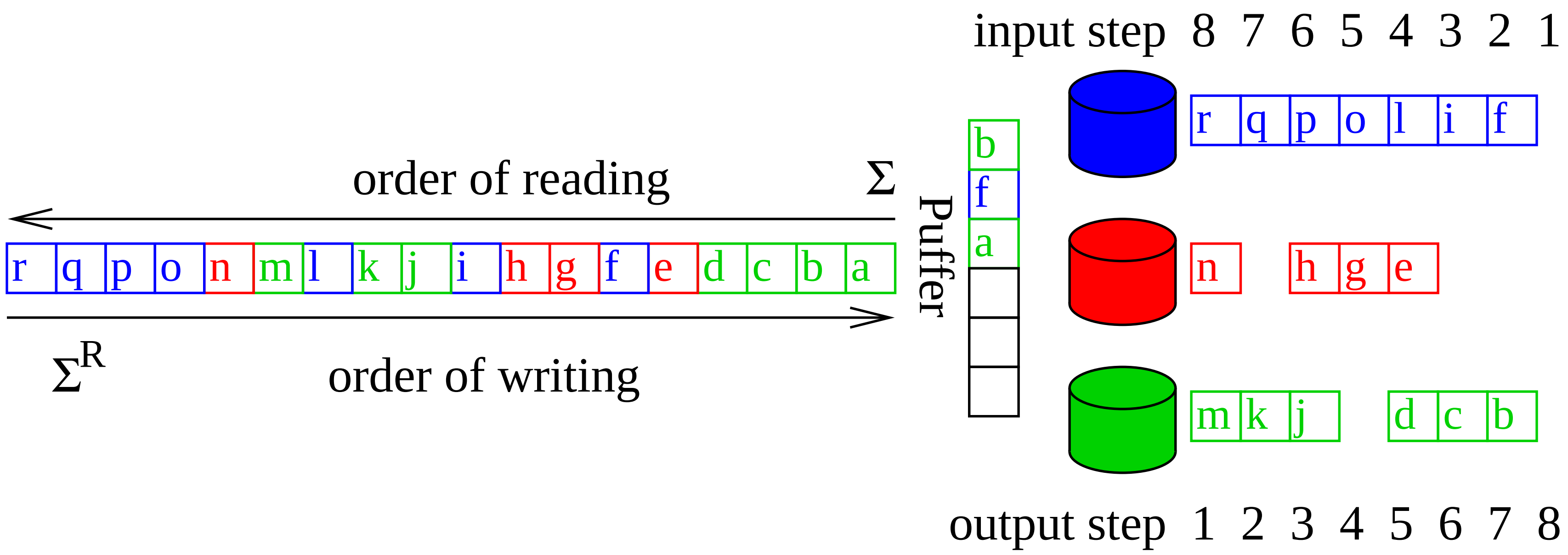
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Optimal Offline Prefetching

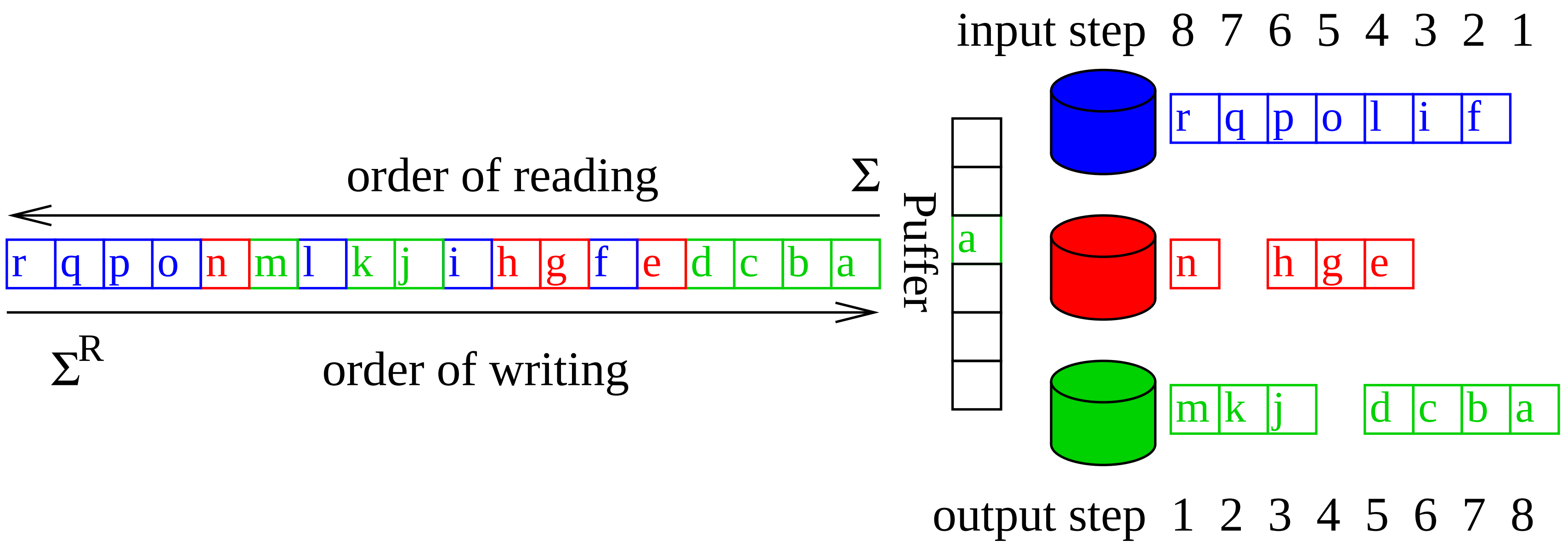
Theorem:

For buffer size W:

$\exists$ (offline) prefetching schedule for Σ with T input steps

$\Leftrightarrow$

$\exists$ (online) write schedule for Σ^R with T output steps



Synthesis

Multiway merging

+ prediction [60s Folklore]

+optimal (randomized) writing [Sanders Egner Korst SODA 2000]

+randomized cycling [Vitter Hutchinson 2001]

+optimal prefetching [Hutchinson Sanders Vitter ESA 2002]

$\rightsquigarrow (1+o(1))\cdot\mathrm{sort}(n)$ I/Os

$\rightsquigarrow$“answers” question in [Knuth 98]; difficulty 48 on a 1..50 scale.

We are not done yet!

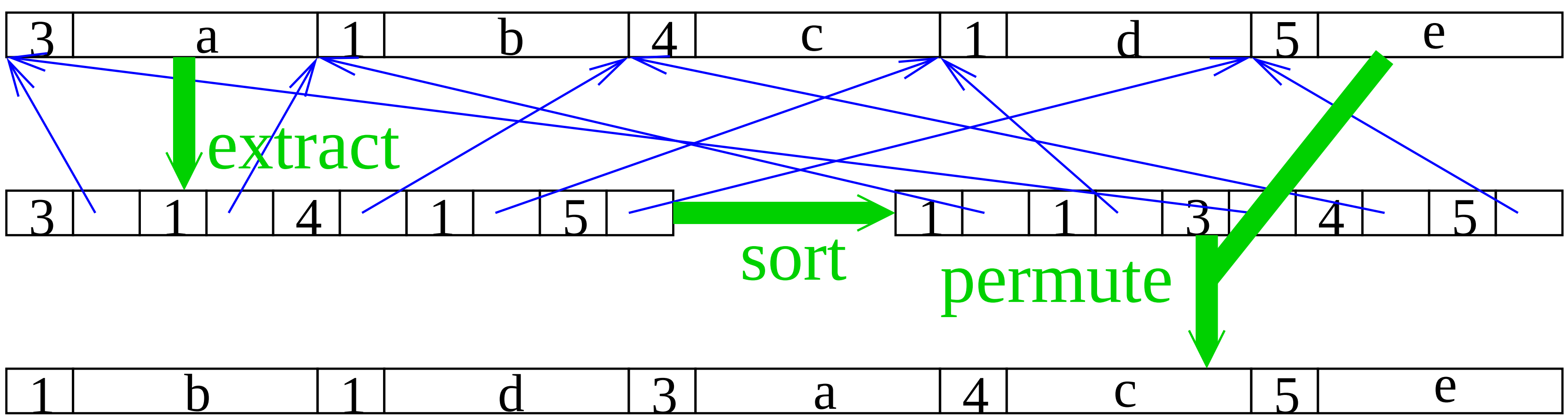
- Internal work
- Overlapping I/O and computation
- Reasonable hardware
- Interfacing with the Operating System
- Parameter Tuning
- Software engineering
- Pipelining

Key Sorting

The I/O bandwidth of our machine is about $1/3$ of its main memory bandwidth

$\rightsquigarrow$ If key size $\ll$ element size

sort key pointer pairs to save memory bandwidth during run formation



Tournament Trees for Multiway Merging

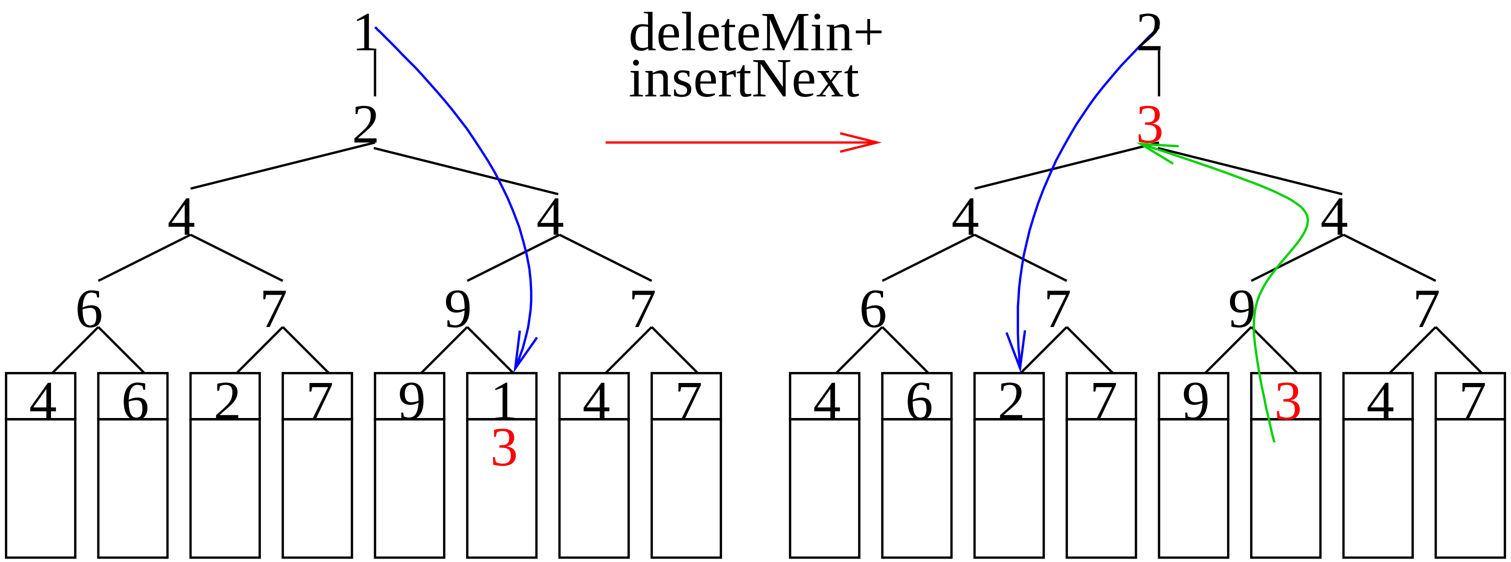
Assume $k = 2^K$ runs

K level complete binary tree

Leaves: smallest current element of each run

Internal nodes: loser of a competition for being smallest

Above root: global winner



Why Tournament Trees

- Exactly $\log k$ element comparisons
- Implicit layout in an array $\rightsquigarrow$ simple index arithmetics (shifts)
- Predictable load instructions and index computations

(Unrollable) inner loop:

```
for (int i=(winnerIndex+kReg)>>1; i>0; i>>=1){
  currentPos = entry + i;
  currentKey = currentPos->key;
  if (currentKey < winnerKey) {
    currentIndex      = currentPos->index;
    currentPos->key   = winnerKey;
    currentPos->index = winnerIndex;
    winnerKey         = currentKey;
    winnerIndex       = currentIndex;}}
```

Overlapping I/O and Computation

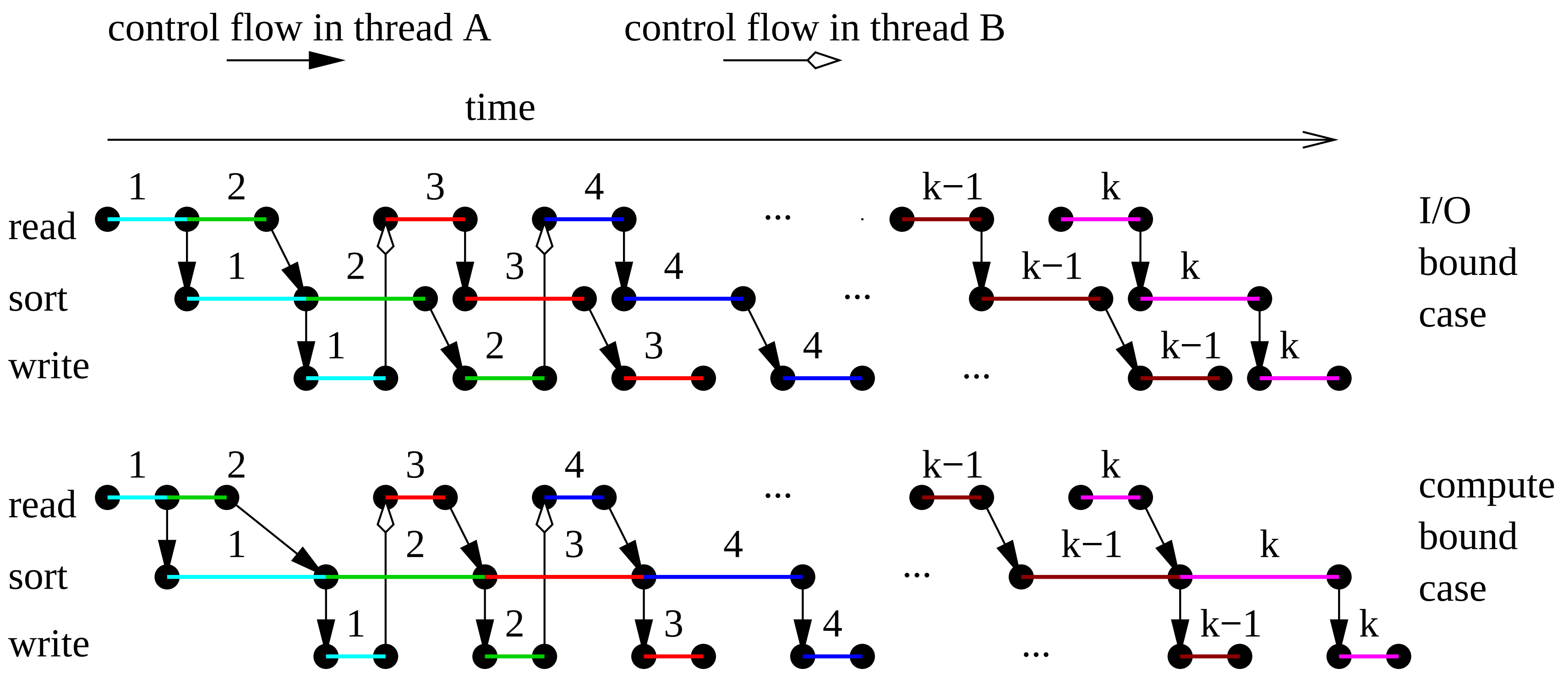
- One thread for each disk (or asynchronous I/O)
- Possibly additional threads
- Blocks filled with elements are passed by references between different buffers

Overlapping During Run Formation

First post read requests for runs 1 and 2

Thread A: Loop { wait-read i; sort i; post-write i};

Thread B: Loop { wait-write i; post-read $i+2$};



Overlapping During Merging

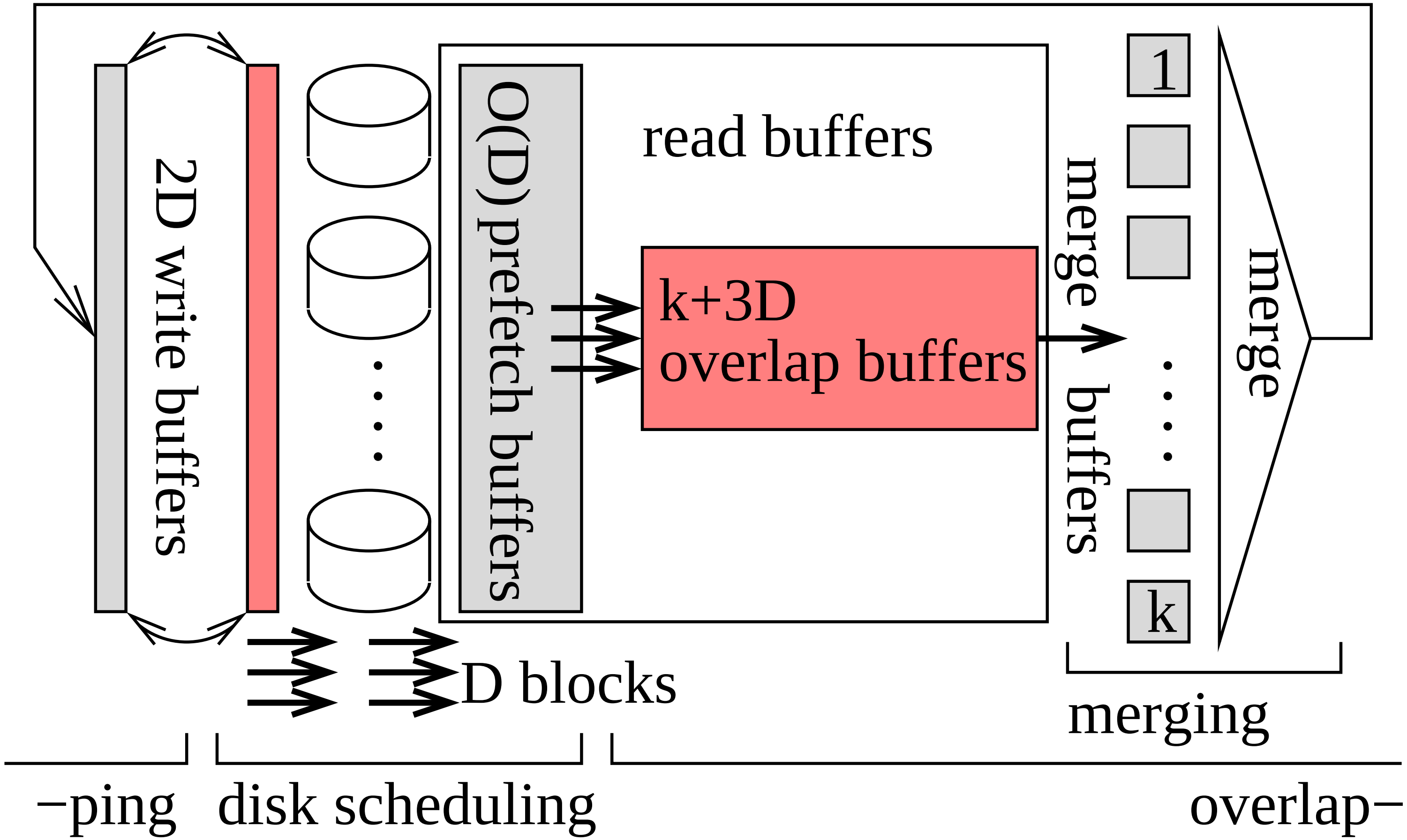
Bad example:

$$\boxed{1^{B-1}2}\boxed{3^{B-1}4}\boxed{5^{B-1}6}\cdots$$

$$\cdots$$

$$\boxed{1^{B-1}2}\boxed{3^{B-1}4}\boxed{5^{B-1}6}\cdots$$

Overlapping During Merging



I/O Threads: Writing has priority over reading

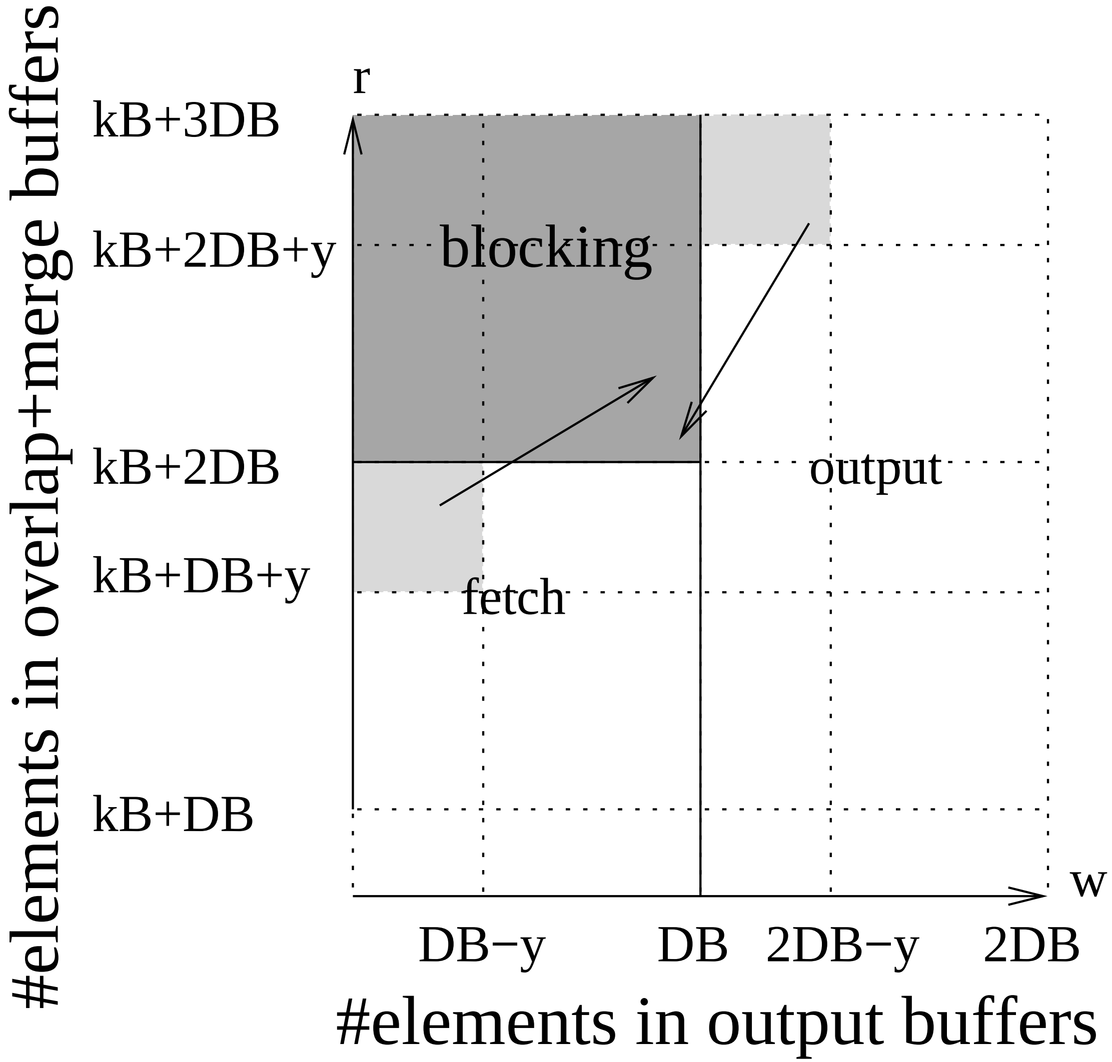
I/O bound case: The I/O thread never blocks

$y = \#$ of elements merged during one I/O step.

I/O bound $\leadsto$

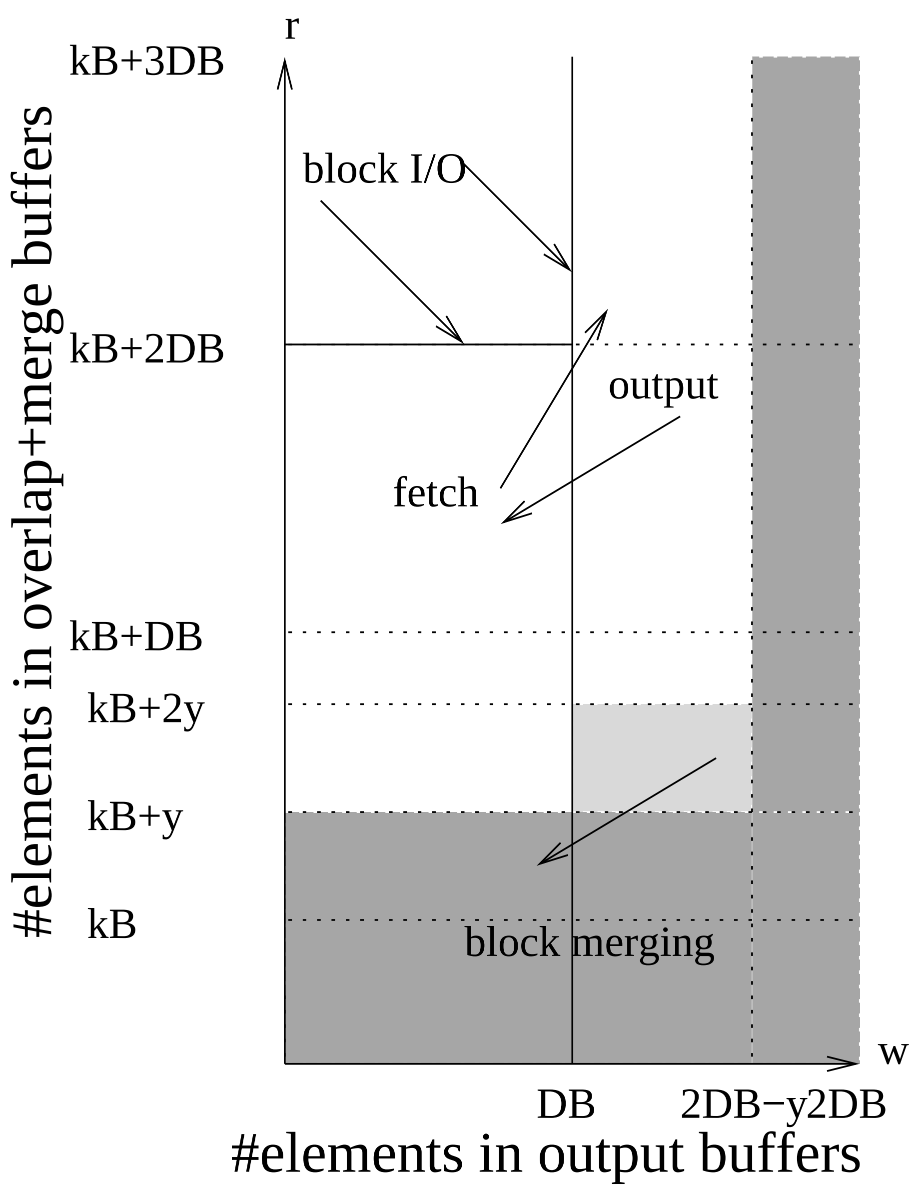
$y > \frac{DB}{2}$

$y \leq DB$



Compute bound case:

The merging thread never blocks



Hardware (mid 2002)

Linux

(2 × 2GHz Xeon × 2 Threads)

Several 66 MHz PCI-buses

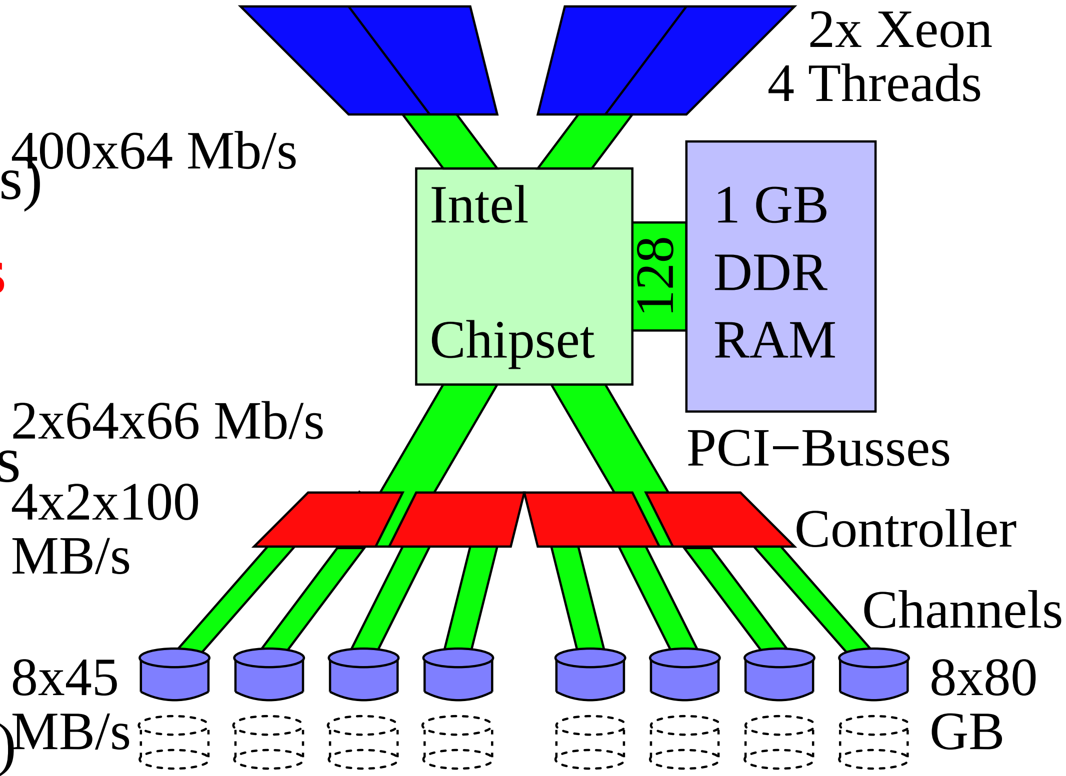
(SuperMicro P4DPE3)

Several fast IDE controllers

(4 × Promise Ultra100 TX2)

Many fast IDE disks

(8× IBM IC35L080AVVA07)

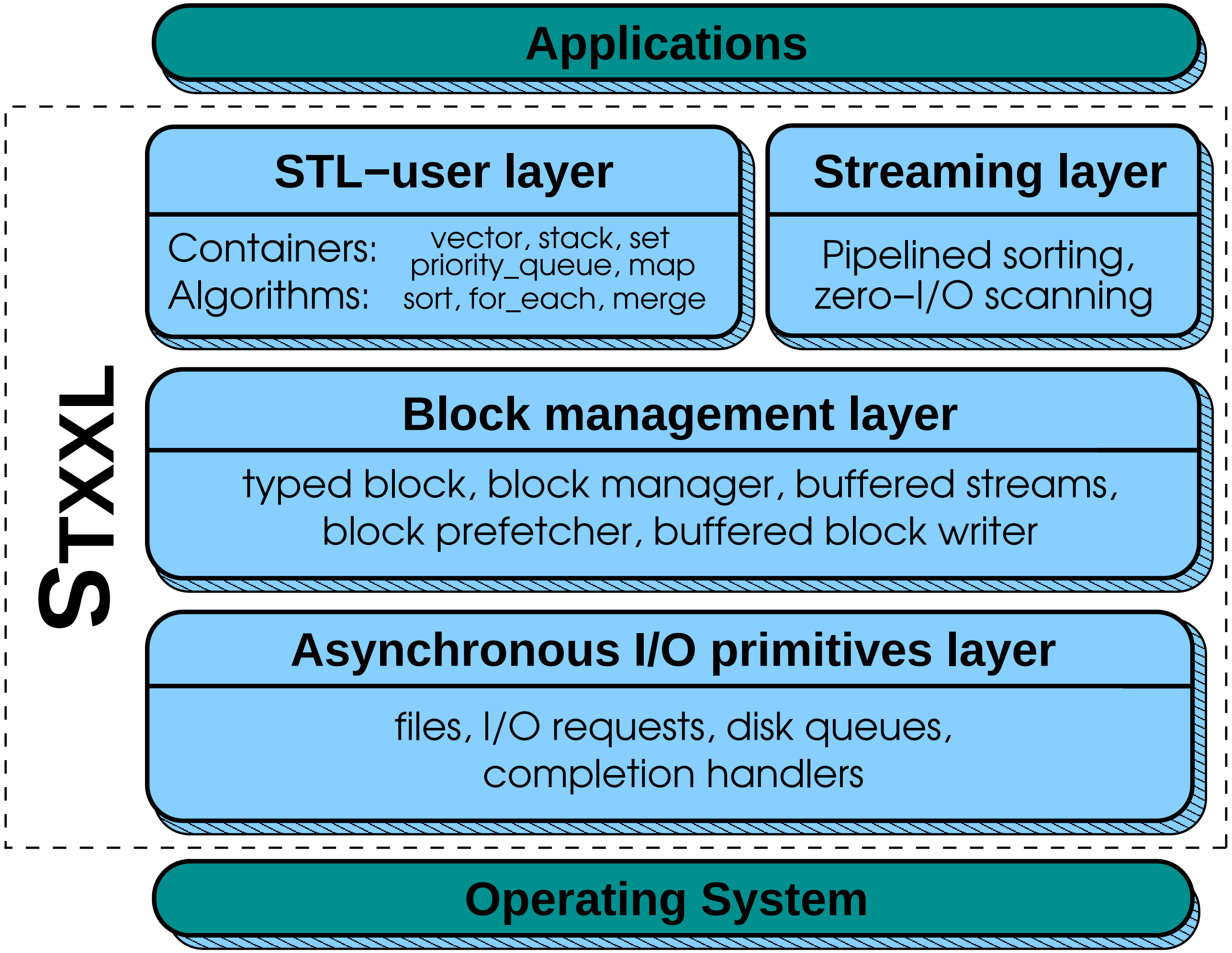


---

cost effective I/O-bandwidth (real 360 MB/s for ≈ 3000 €)

Software Interface

Goals: efficient + simple + compatible



Default Measurement Parameters

$t :=$ number of available buffer blocks

Input Size: 16 GByte

Element Size: 128 Byte

Keys: Random 32 bit integers

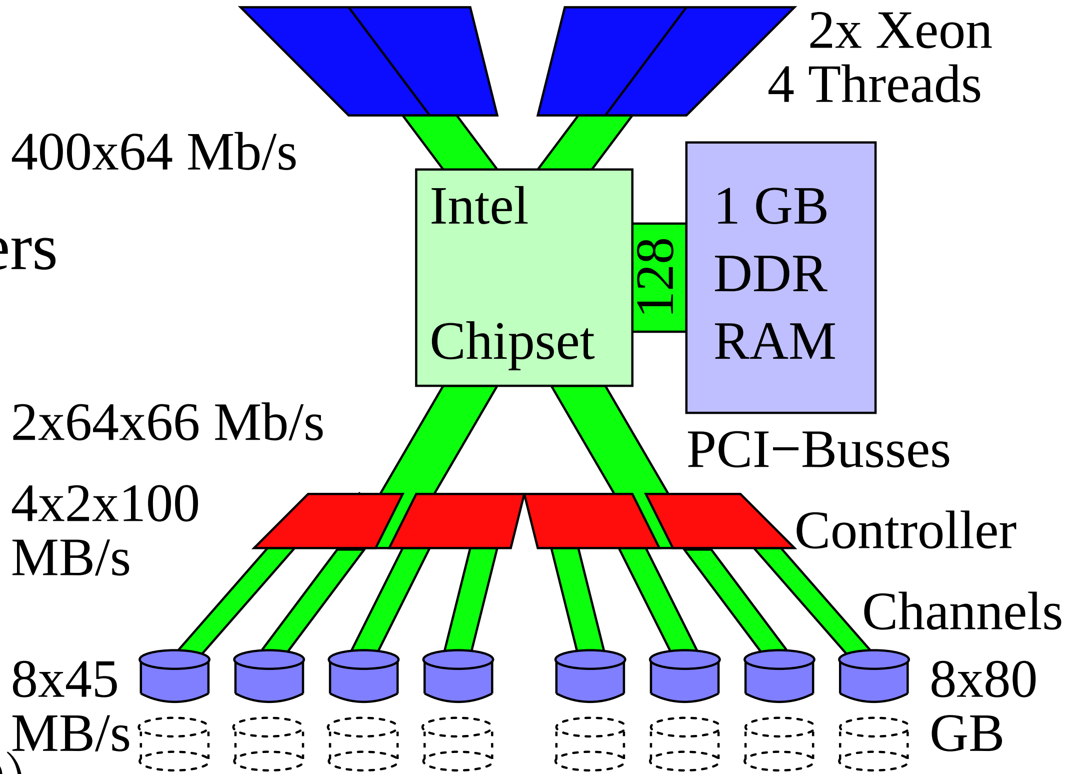
Run Size: 256 MByte

Block size B: 2 MByte

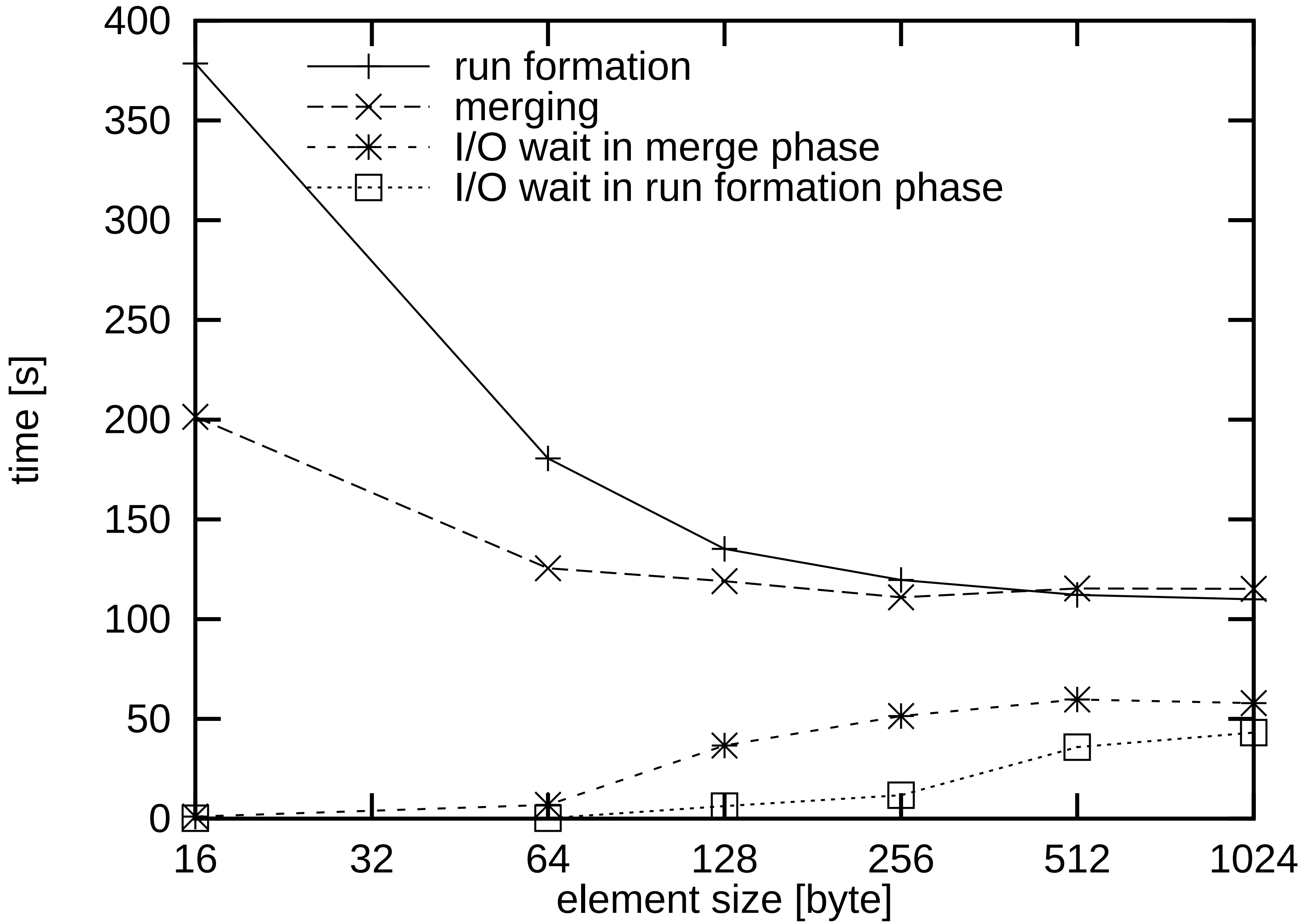
Compiler: g++ 3.2 -O6

Write Buffers: $\max(t/4, 2D)$

Prefetch Buffers: $2D + \frac{3}{10}(t - w - 2D)$



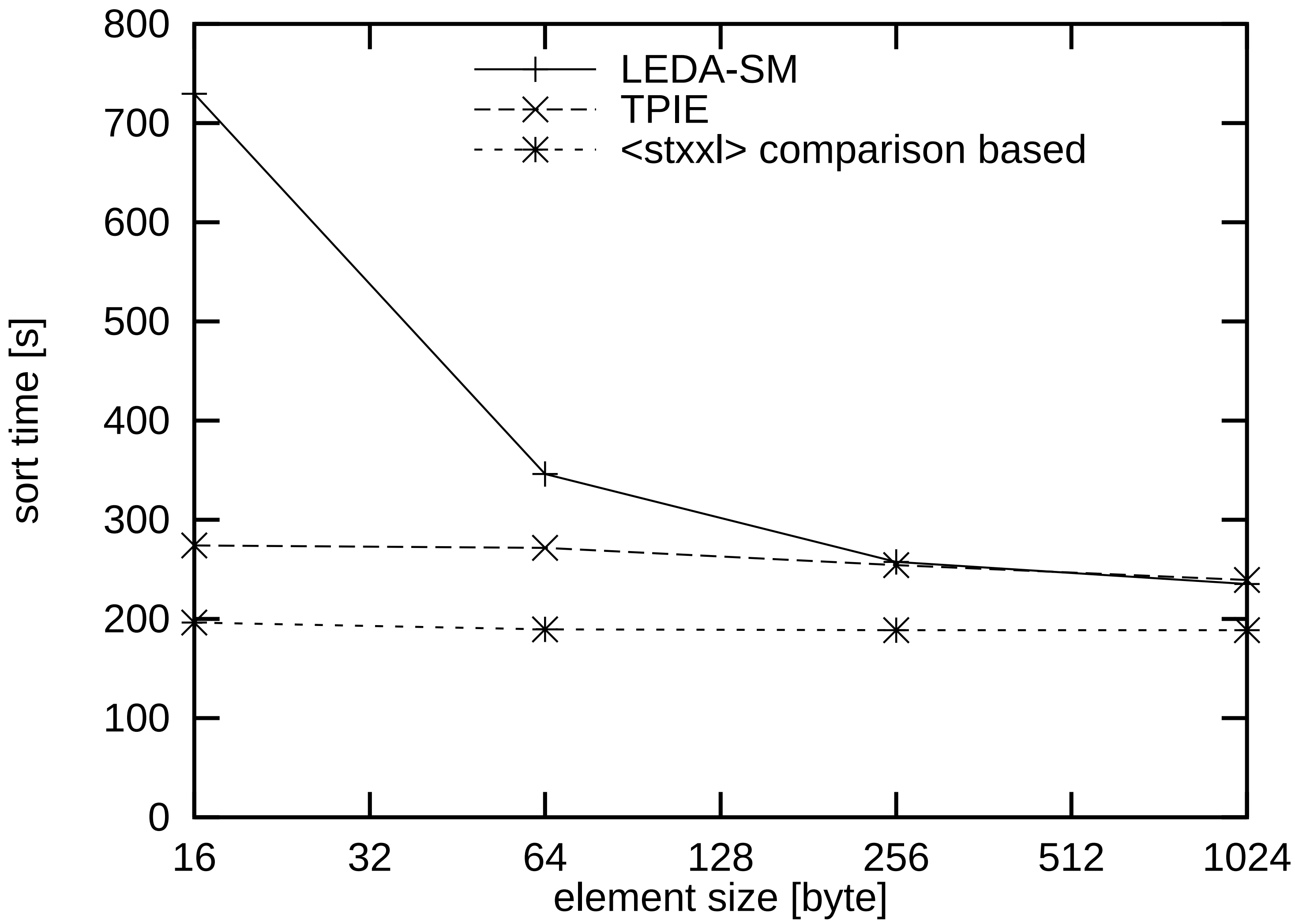
Element sizes (16 GByte, 8 disks)



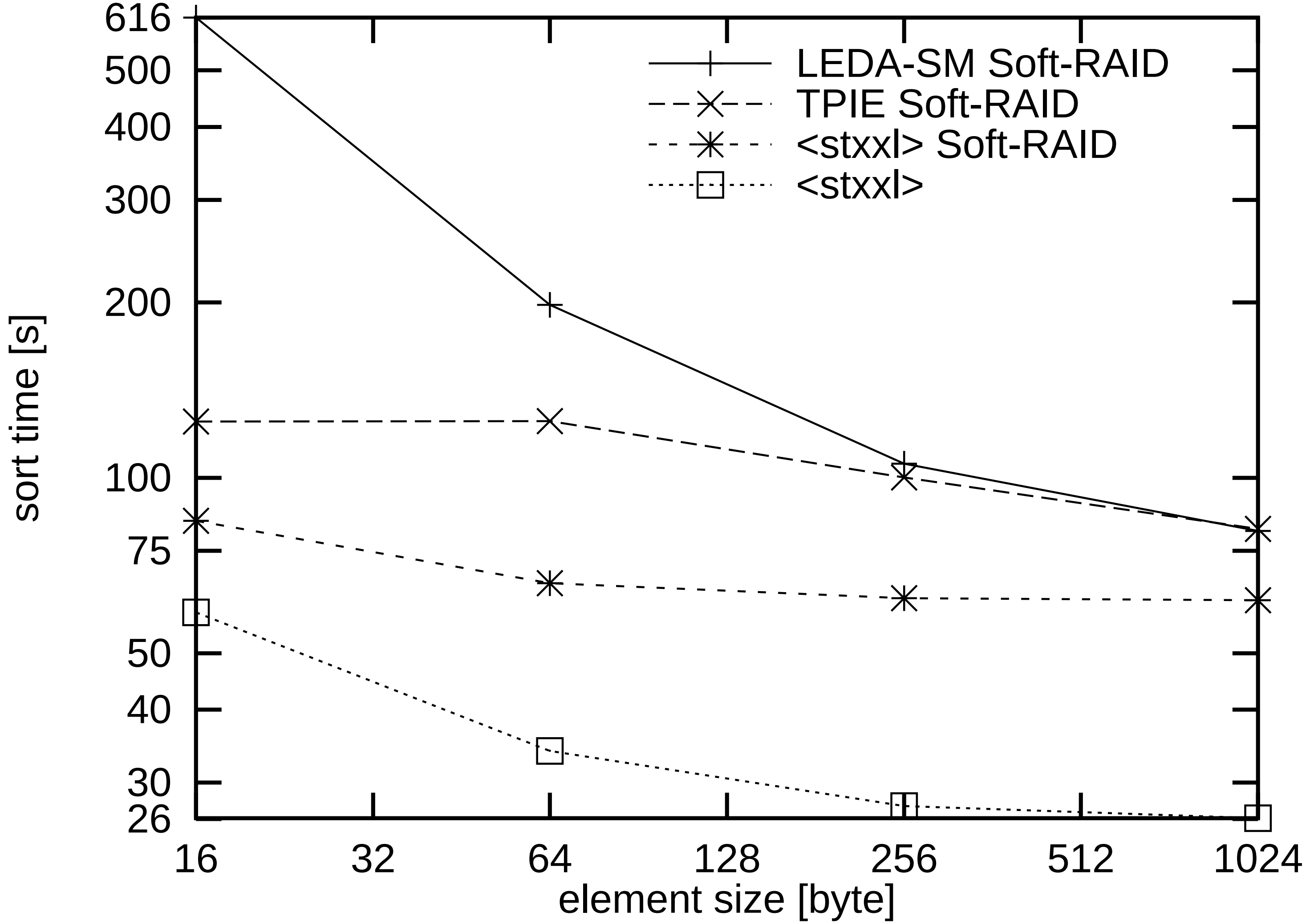
parallel disks ⇝ bandwidth “for free” ⇝ internal work, overlapping are relev

Earlier Academic Implementations

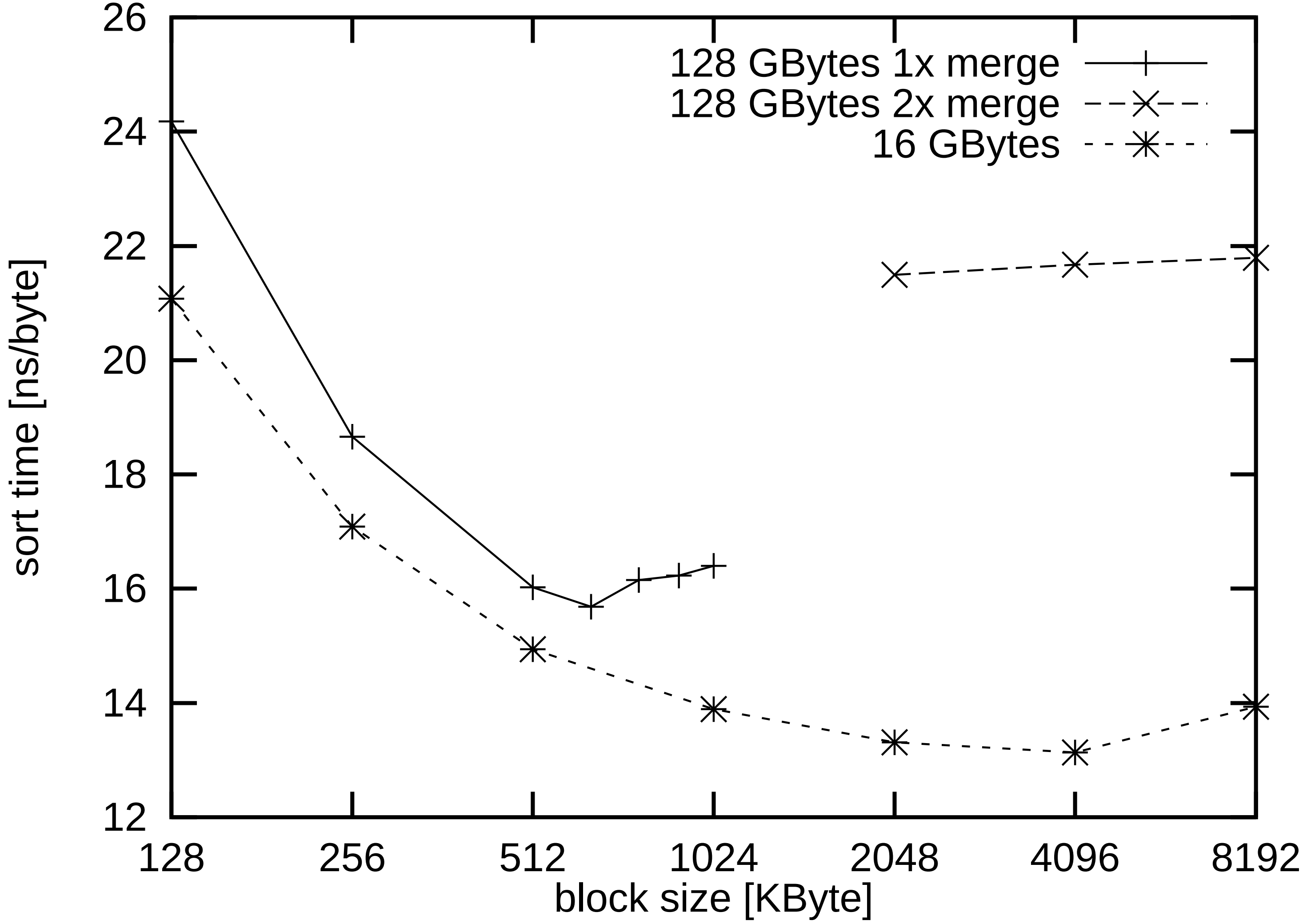
Single Disk, at most 2 GByte, old measurements use artificial M



Earlier Academic Implementations: Multiple Disks



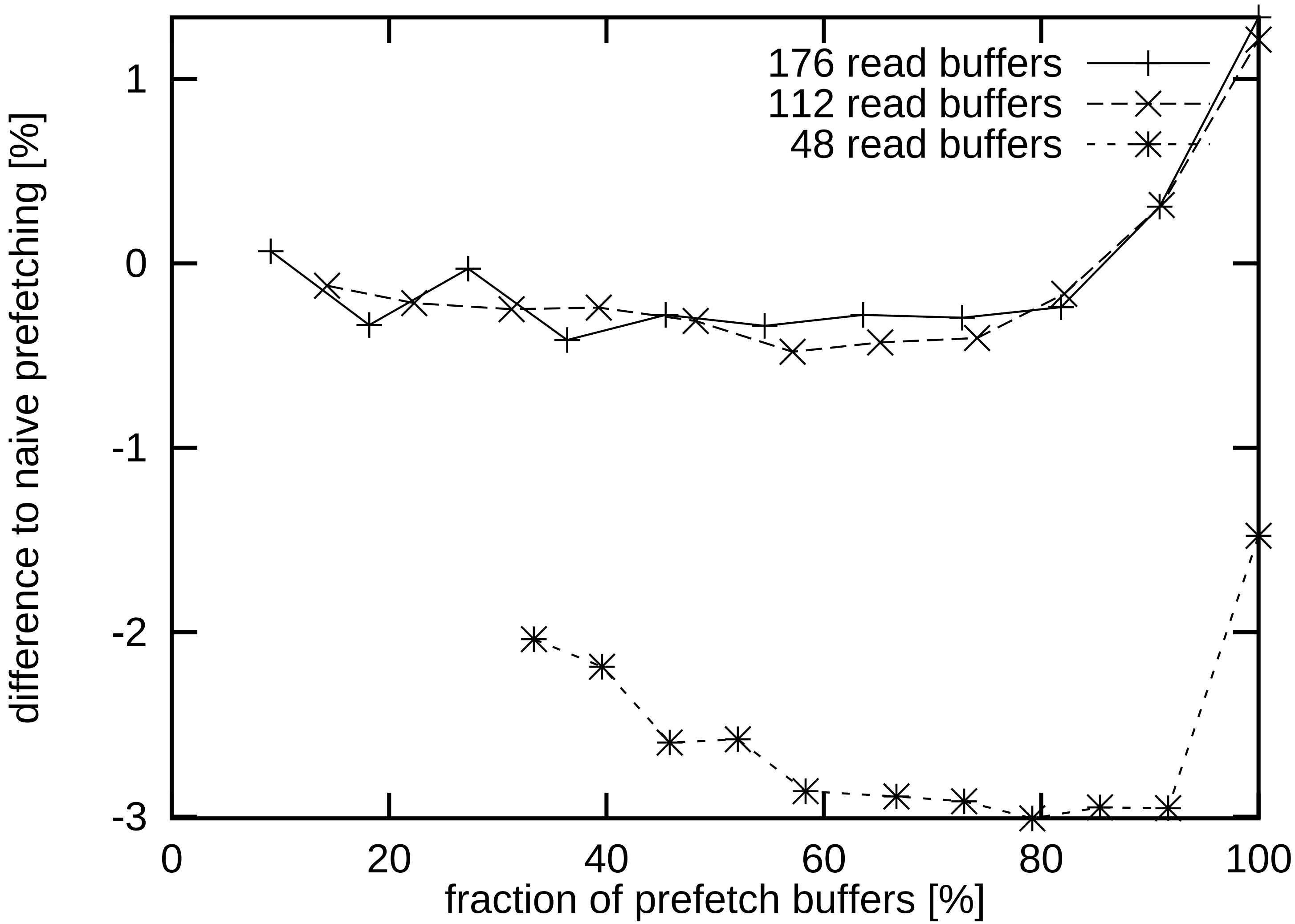
What are good block sizes (8 disks)?



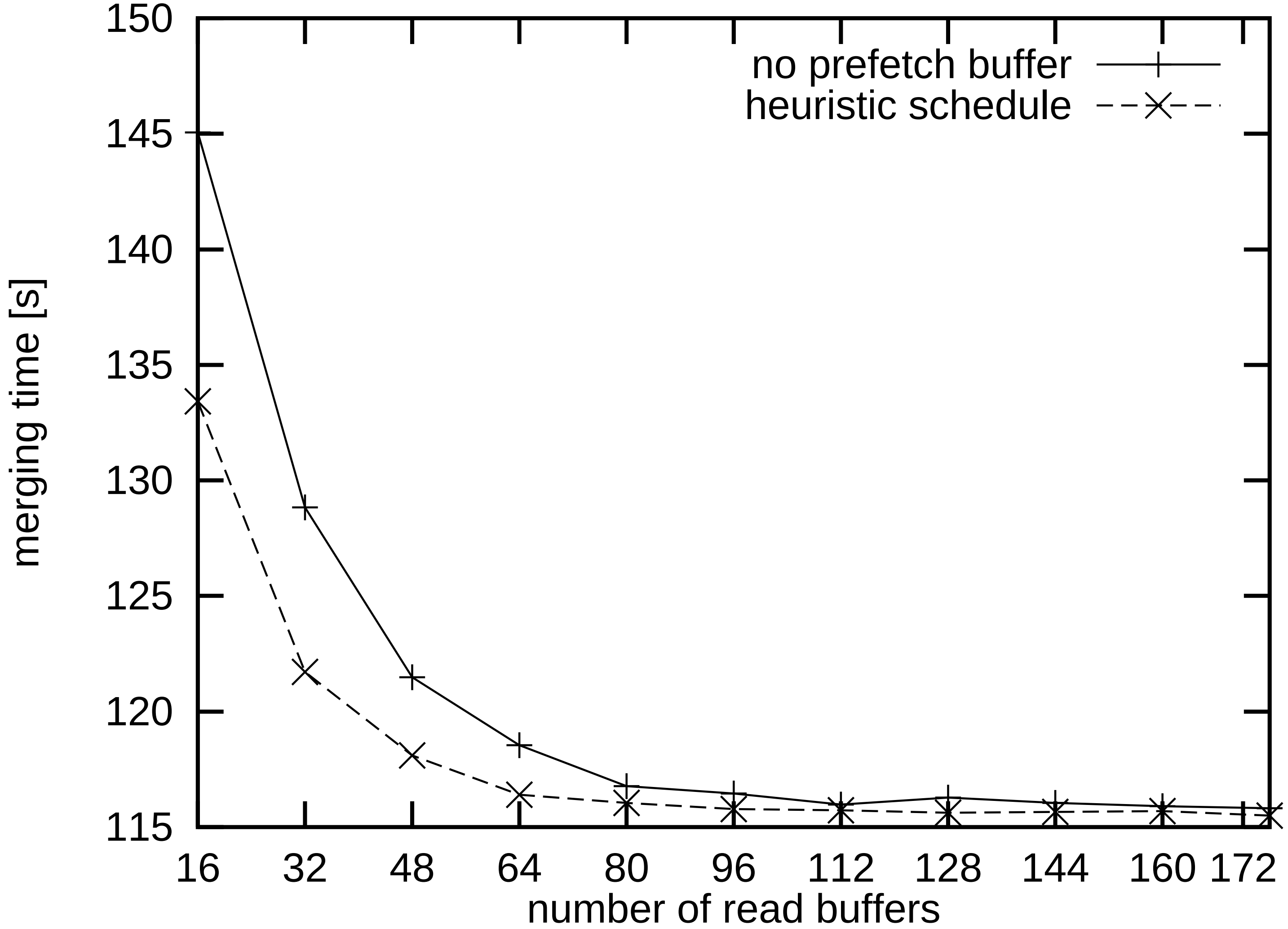
B is not a technology constant

Optimal Versus Naive Prefetching

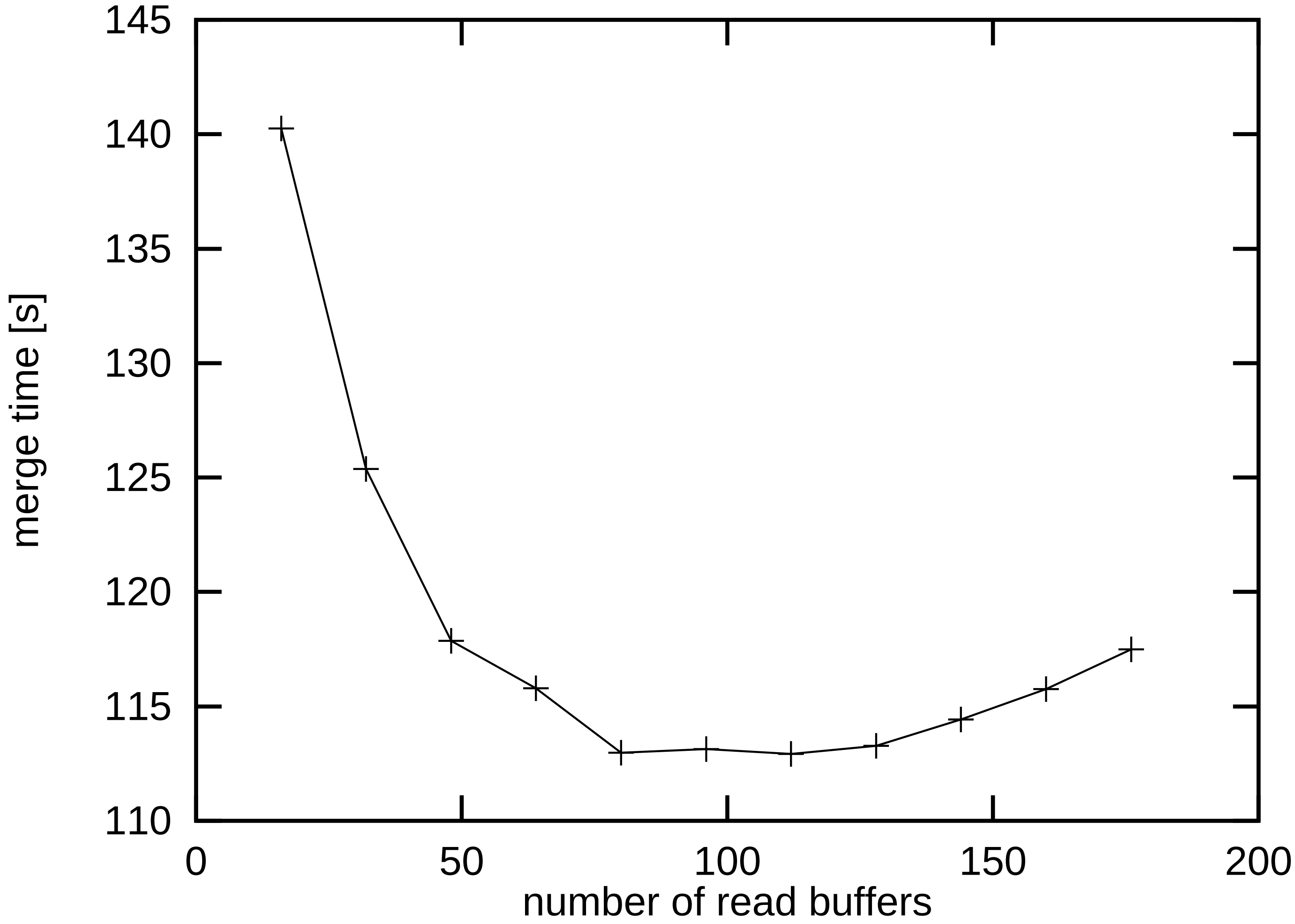
Total merge time



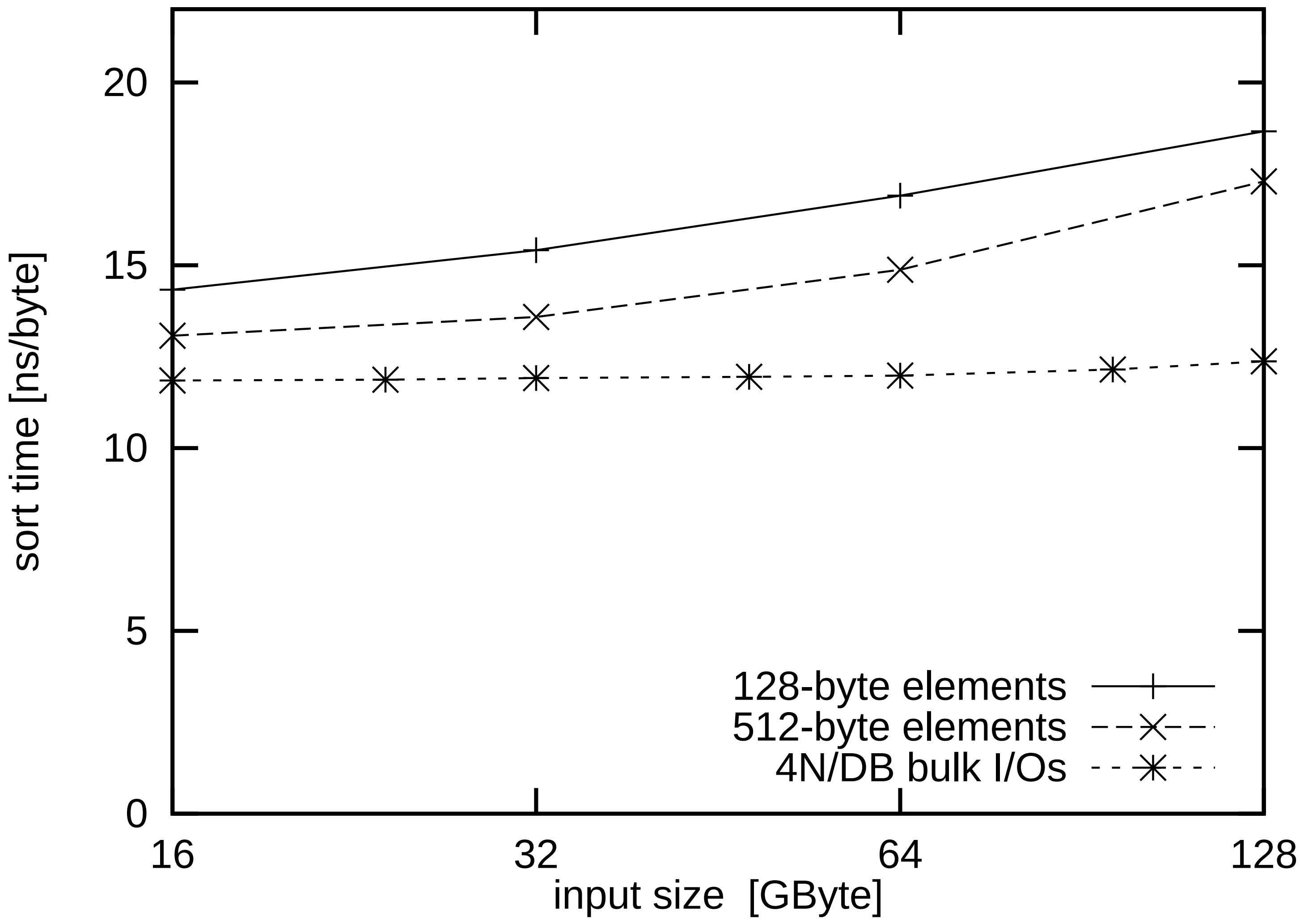
Impact of Prefetch and Overlap Buffers



Tradeoff: Write Buffer Size Versus Read Buffer Size



Scalability



Discussion

- Theory and practice harmonize
- No expensive server hardware necessary (SCSI,...)
- No need to work with artificial M
- No 2/4 GByte limits
- Faster than academic implementations
- (Must be) as fast as commercial implementations but with performance guarantees
- Blocks are much larger than often assumed. Not a technology constant
- Parallel disks $\rightsquigarrow$ bandwidth "for free" $\rightsquigarrow$ don't negliect internal costs

More Parallel Disk Sorting?

Pipelining: Input does not come from disk but from a logical input stream. Output goes to a logical output stream
$\rightsquigarrow$ only half the I/Os for sorting
$\rightsquigarrow$ often no I/Os for scanning todo: better overlapping

Parallelism: This is the only way to go for really many disks

Tuning and Special Cases: ssssort, permutations, balance work between merging and run formation?...

Longer Runs: not done with guaranteed overlapping, fast internal sorting !

Distribution Sorting: Better for seeks etc.?

Inplace Sorting: Could also be faster

Determinism: A practical and theoretically efficient algorithm?

Procedure formLongRuns

q, q' : PriorityQueue

for $i := 1$ **to** M **do** q.insert(readElement)

invariant $|q| + |q'| = M$

loop

while $q \neq \emptyset$

writeElement(e:= q.deleteMin)

if input exhausted **then** break outer loop

if e':= readElement $< e$ **then** q'.insert(e')

else q.insert(e')

q:= q'; q':= $\emptyset$

output q in sorted order; output q' in sorted order

Knuth: average run length $2M$

todo: cache-effiziente Implementierung

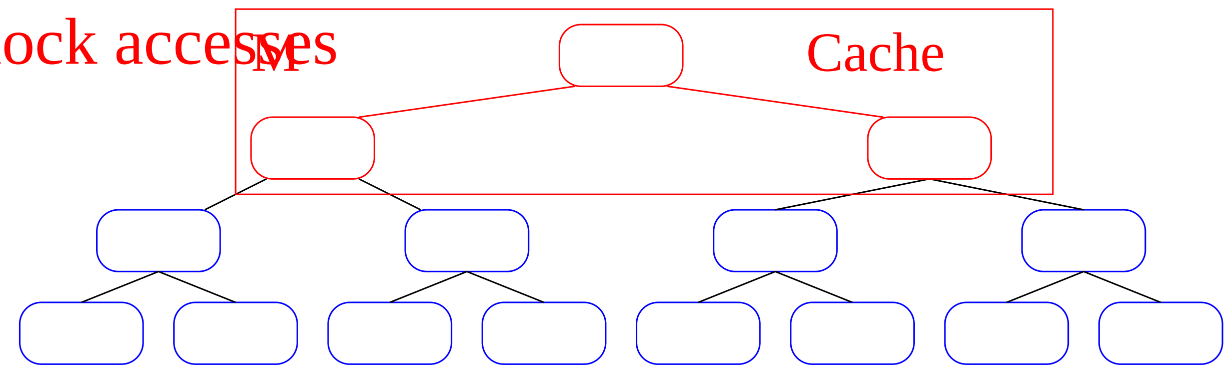
4 Priority Queues (insert, deleteMin)

Binary Heaps best comparison based “flat memory” algorithm

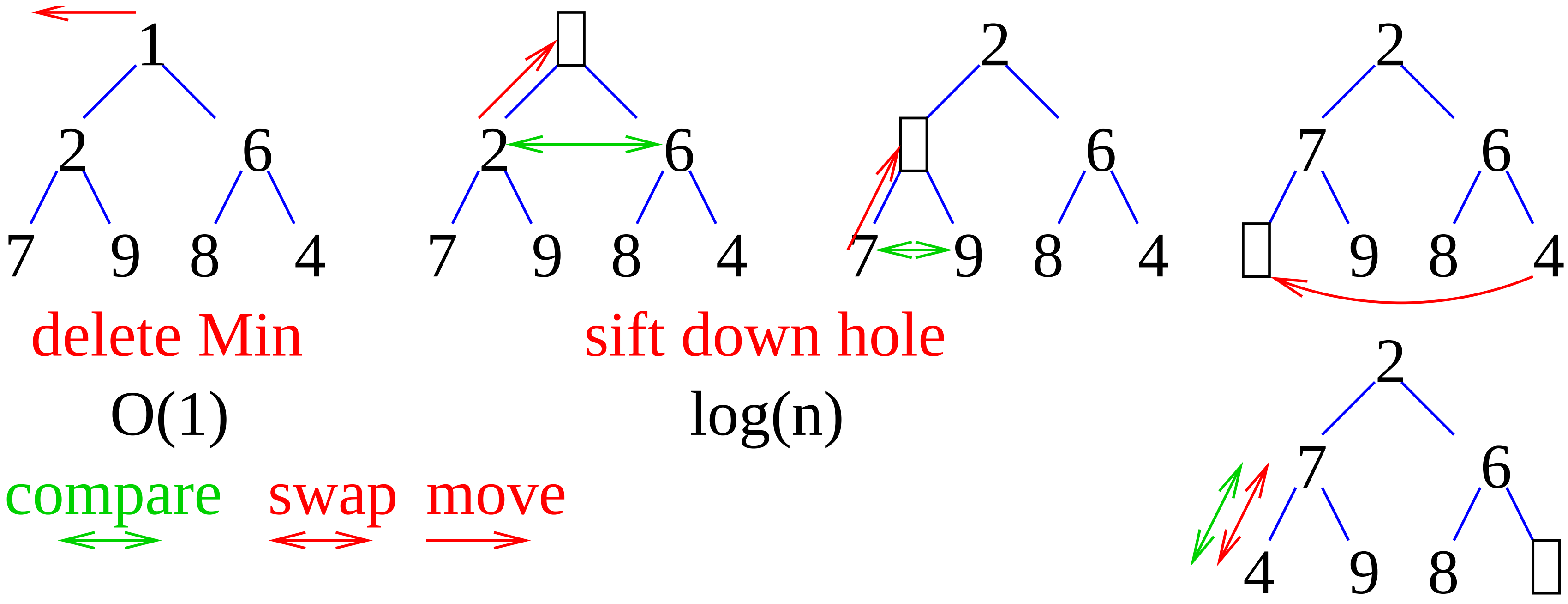
+ On average constant time for insertion

+ On average $\log n + O(1)$ key comparisons per delete-Min using the “bottom-up” heuristics [Wegener 93].

– $\approx 2\log(n/M)$ block accesses per delete-Min

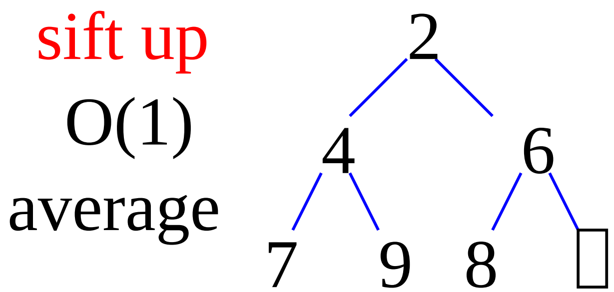


Bottom Up Heuristics



Factor two faster

than naive implementation



Der Wettbewerber fit gemacht:

```
int i=1, m=2, t = a[1];
m += (m != n  &&  a[m] > a[m + 1]);
if (t > a[m]) {
  do { a[i] = a[m];
       i    = m;
       m    = 2*i;
       if (m > n) break;
       m += (m != n  &&  a[m] > a[m + 1]);
  } while (t > a[m]);
  a[i] = t;}
```

Keine signifikanten Leistungsunterschiede auf meiner Maschine (heapsort von random integers)

Vergleich

Speicherzugriffe: $O(1)$ weniger als top down $O(\log n)$ worst case. bei effizienter Implementierung

Elementvergleiche: $\approx \log n$ weniger für bottom up (average case) aber die sind leicht vorhersagbar

Aufgabe: siftDown mit worst case $\log n + O(\log\log n)$ Elementvergleichen

Heapkonstruktion

Procedure buildHeapBackwards

for $i := \lfloor n/2 \rfloor$ **downto** 1 **do** siftDown(i)

Procedure buildHeapRecursive$(i : \mathbb{N})$

if $4i \leq n$ **then**

buildHeapRecursive$(2i)$

buildHeapRecursive$(2i+1)$

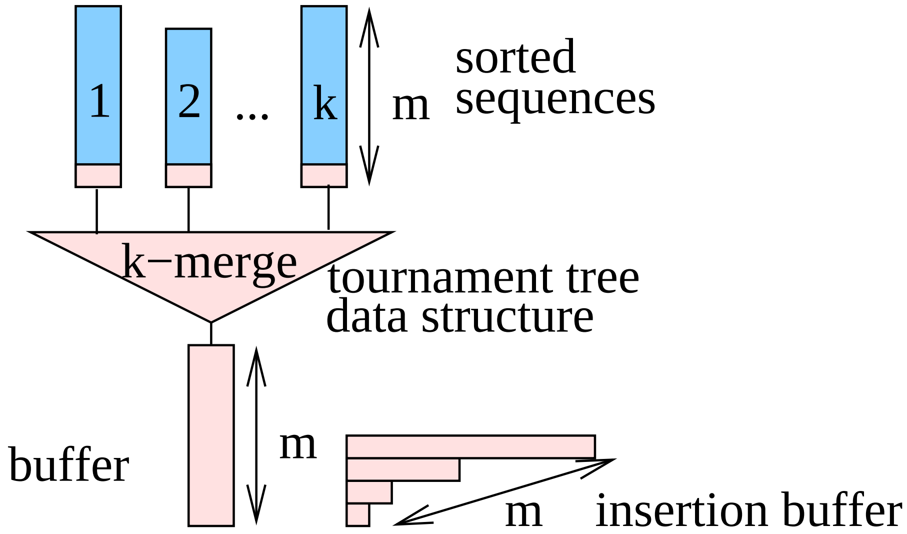
siftDown(i)

Rekursive Funktion für große Eingaben $2\times$ schneller!

(Rekursion abrollen für 2 unterste Ebenen)

Aufgabe: Erklärung

Medium Size Queues ($km \ll M^2/B$ Insertions)



Insert: Initially into insertion buffer.

Overflow $\longrightarrow$

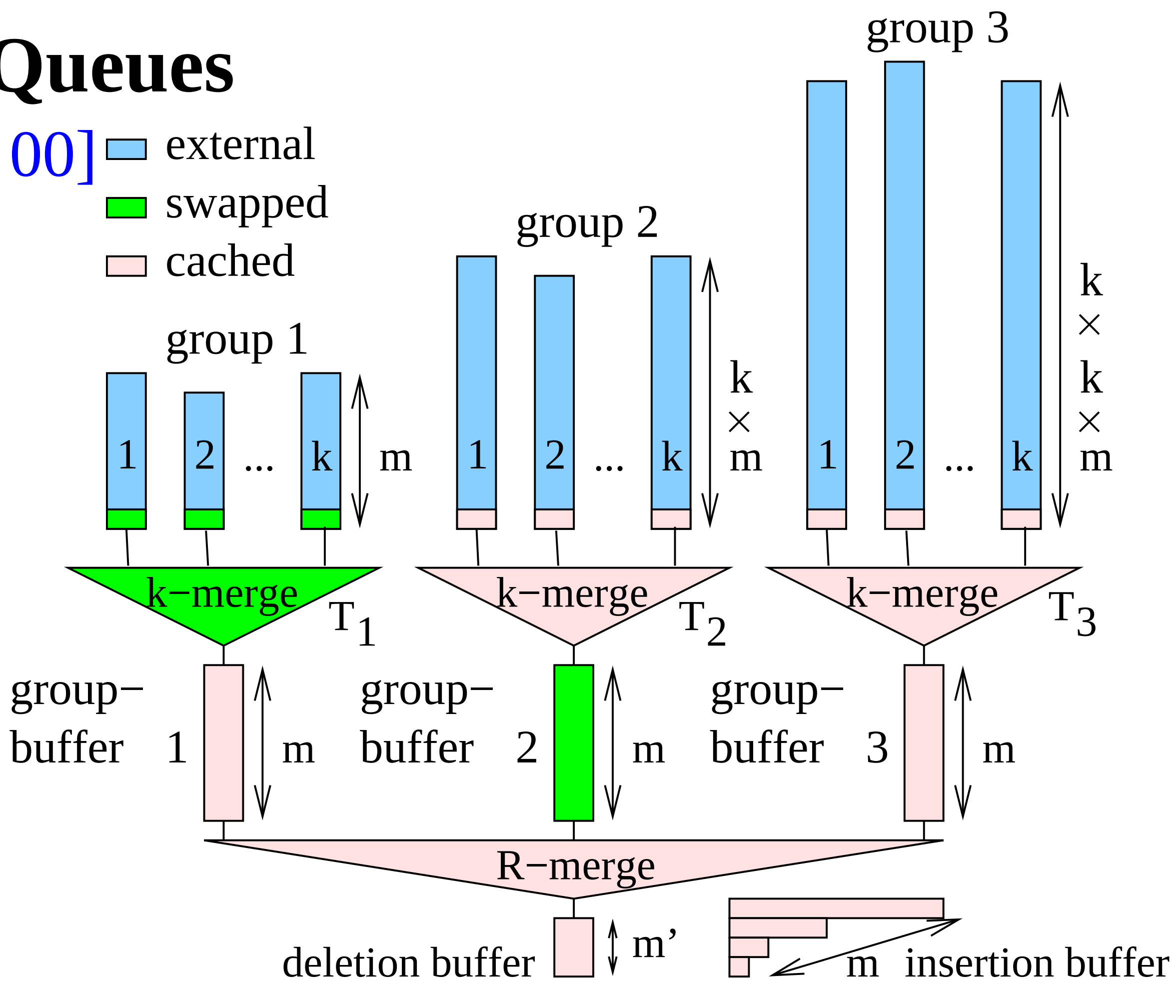
sort; merge with deletion buffer; write out largest elements.

Delete-Min: Take minimum of insertion and deletion buffer.

Refill deletion buffer if necessary.

Large Queues

[Sanders 00]



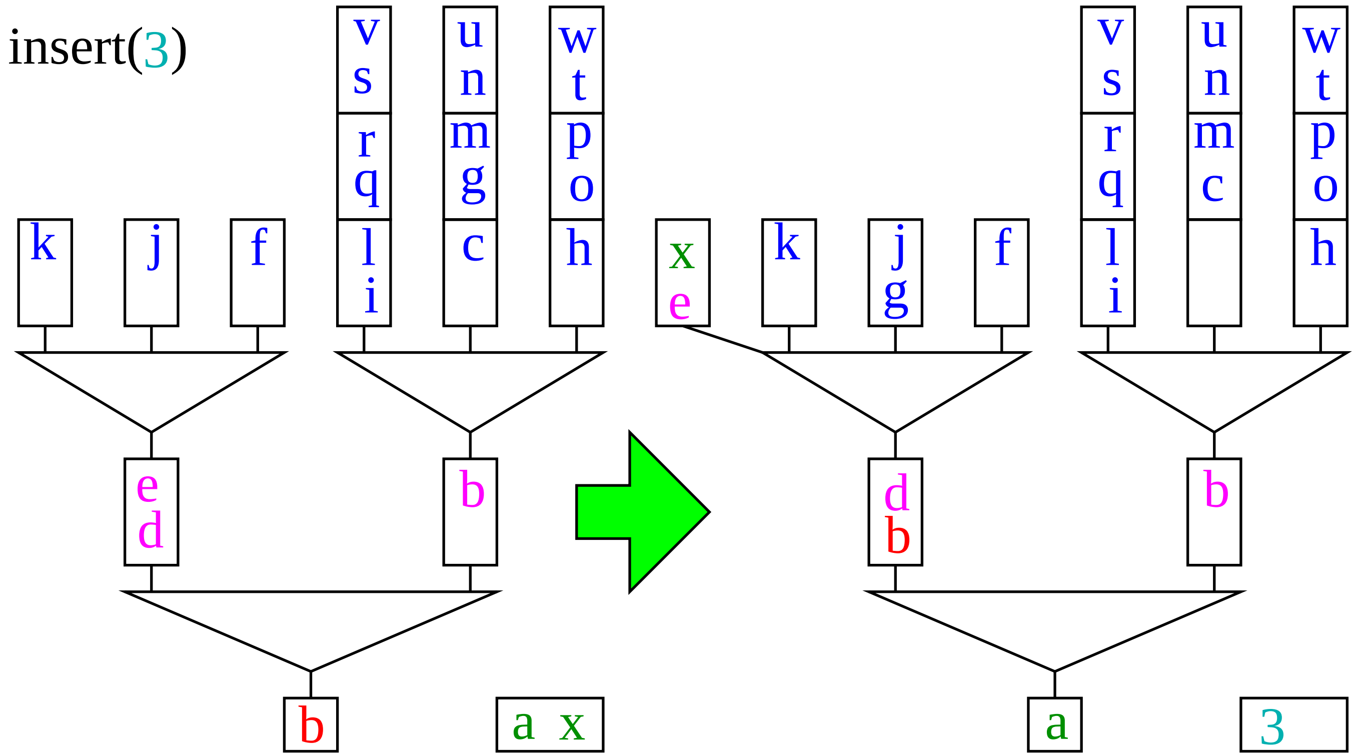
insert: group full ⟶ merge group; shift into next group.

merge invalid group buffers and move them into group 1.

Delete-Min: Refill. $m' \ll m$. nothing else

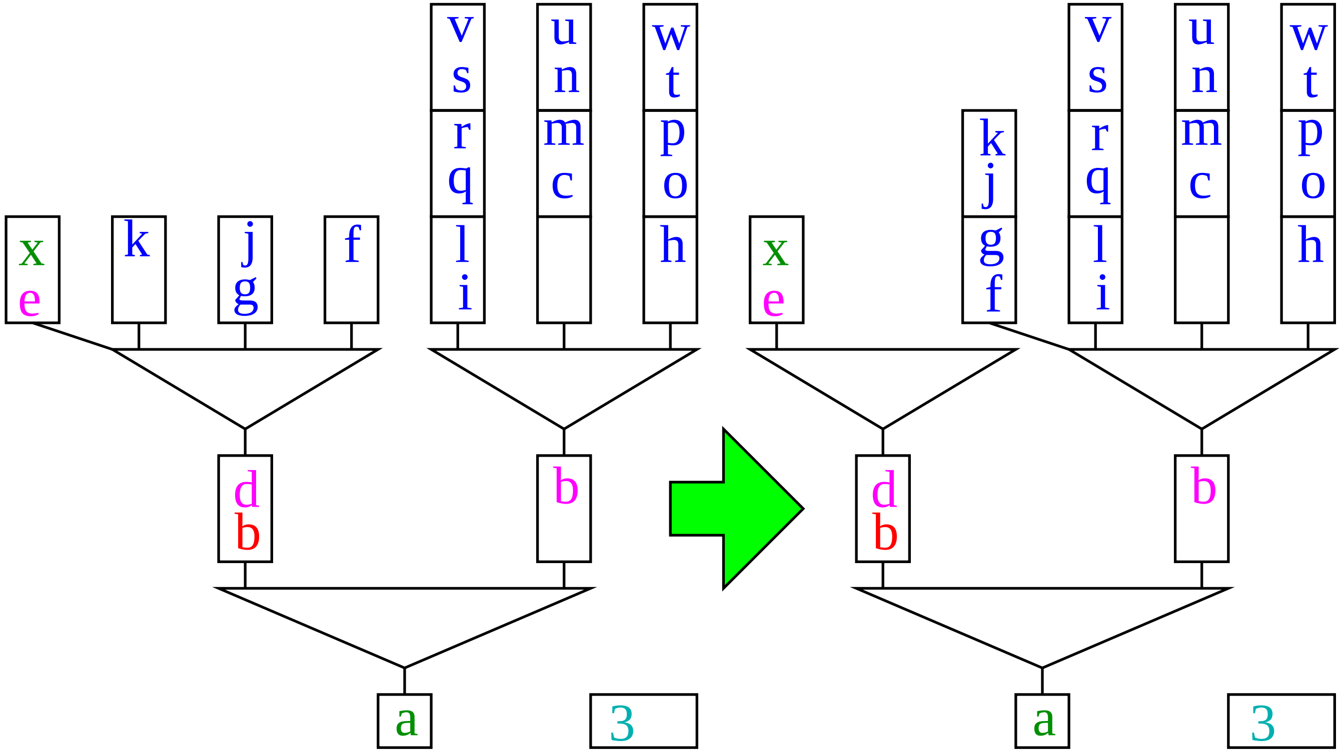
Example

Merge insertion buffer, deletion buffer, and leftmost group buffer



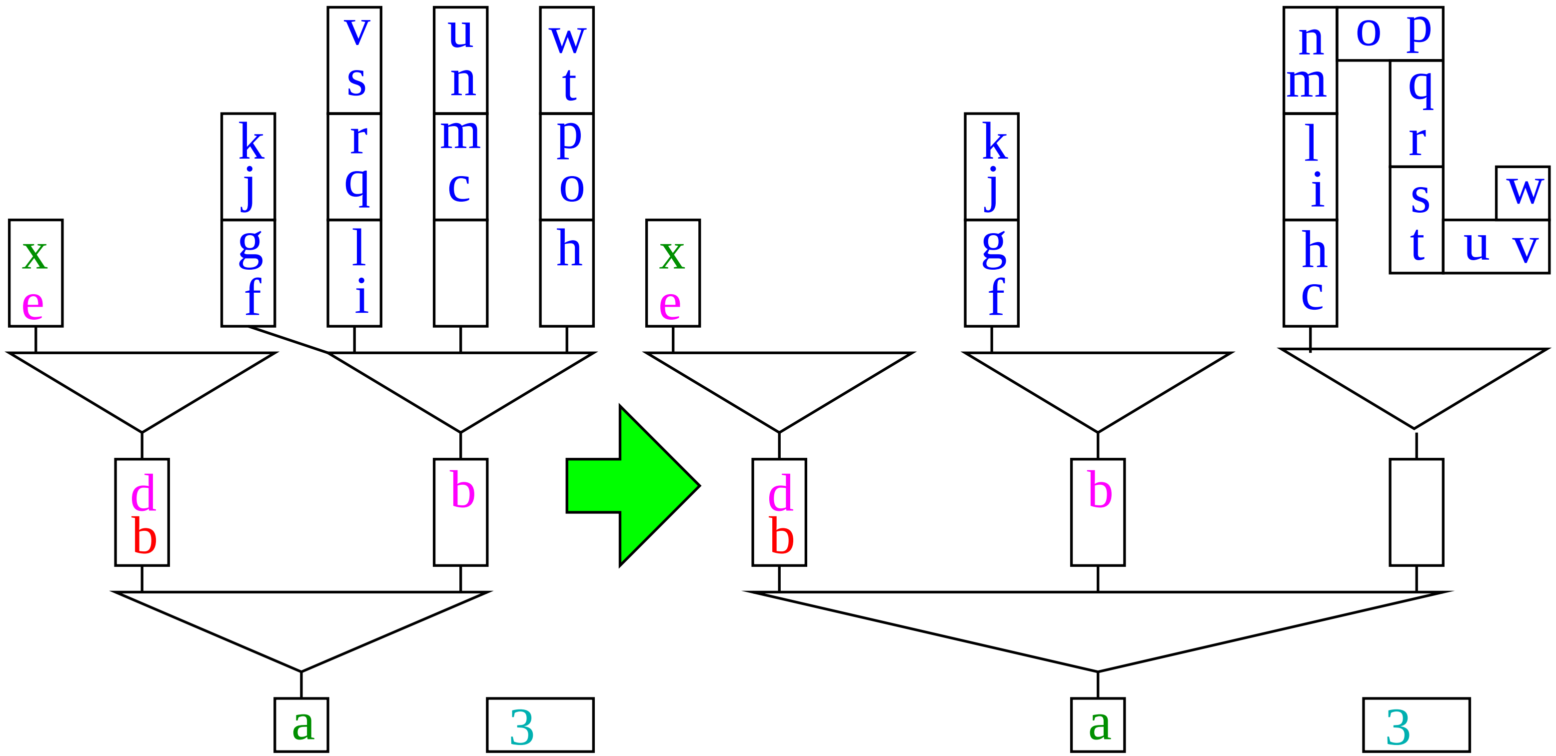
Example

Merge group 1



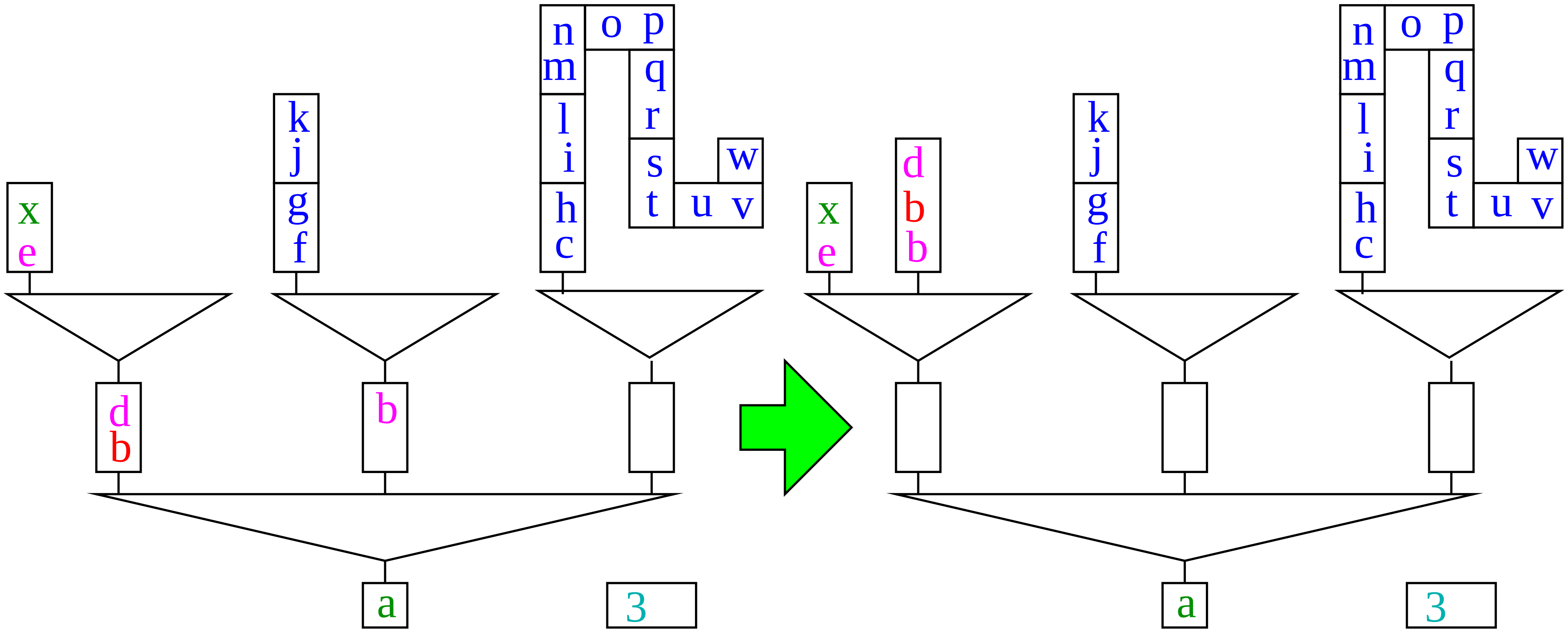
Example

Merge group 2



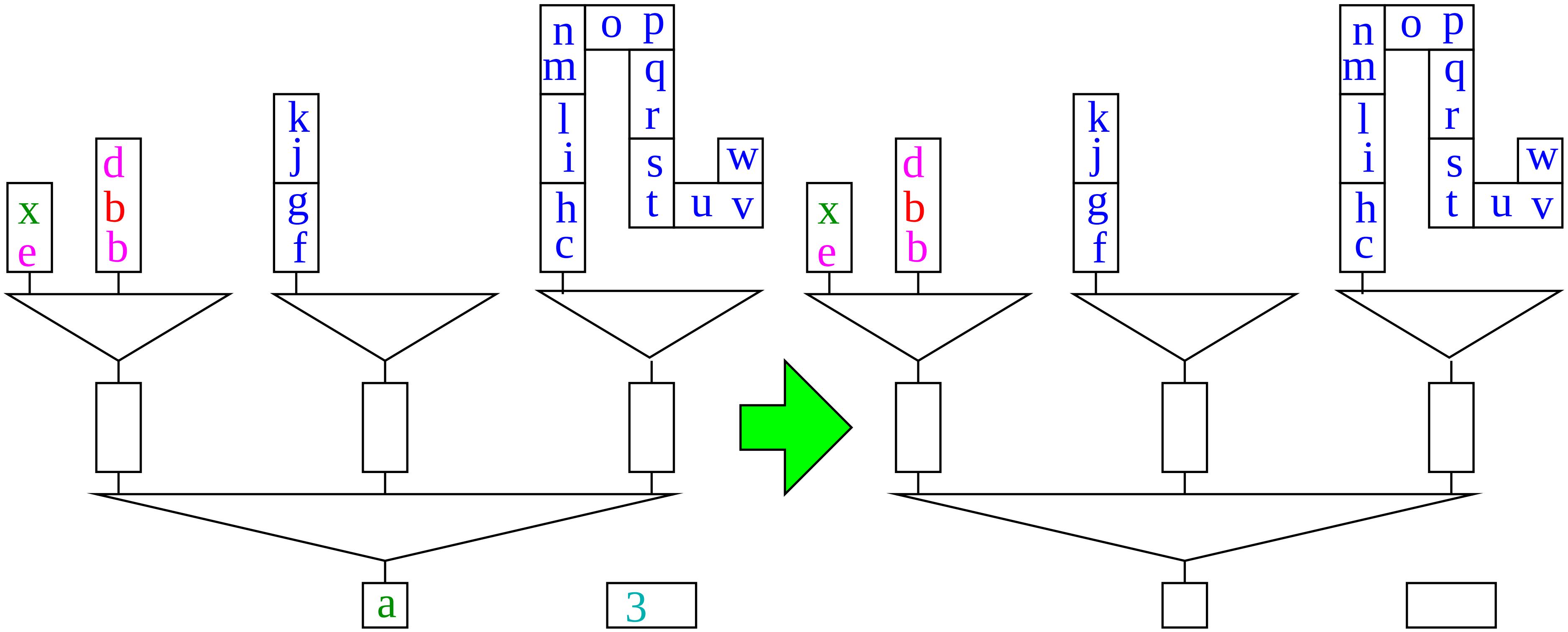
Example

Merge group buffers



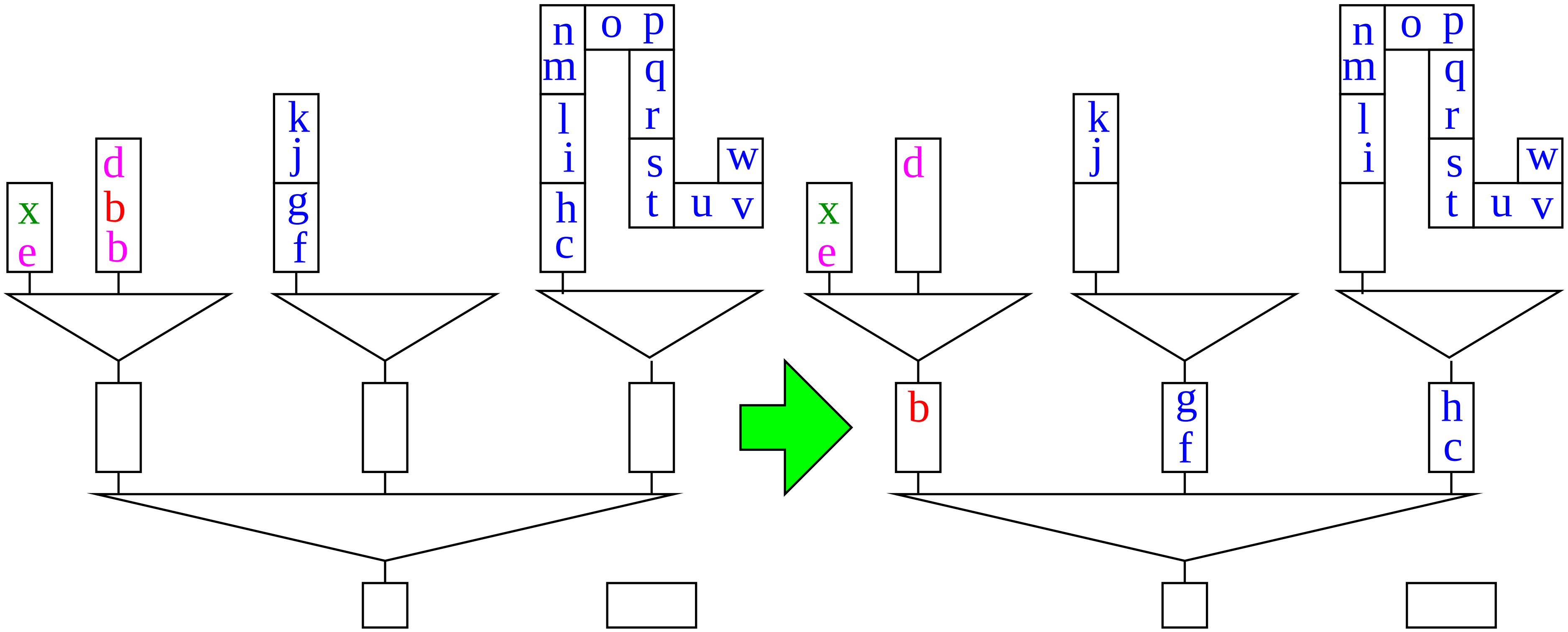
Example

DeleteMin ⇝ 3; DeleteMin ⇝ a;



Example

DeleteMin $\rightsquigarrow$ b



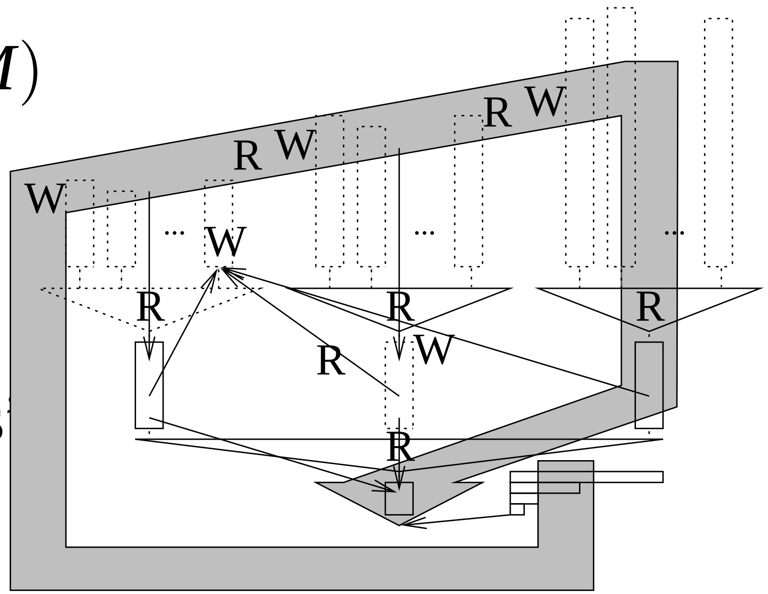
Analysis

- I insertions, buffer sizes $m = \Theta(M)$
- merging degree $k = \Theta(M/B)$

block accesses: $\mathrm{sort}(I)$+“small terms”

key comparisons: $I \log I$ + “small terms”

(on average)



Other (similar, earlier) [Arge 95, Brodal-Katajainen 98, Brengel et al. 99, Fadel et al. 97] data structures spend a factor ≥ 3 more I/Os to replace I by queue size.

Implementation Details

- Fast routines for 2–4 way merging keeping smallest elements in registers
- Use sentinels to avoid special case treatments (empty sequences, ...)
- Currently heap sort for sorting the insertion buffer
- $k \neq M/B$: multiple levels, limited associativity, TLB

Experiments

Keys: random 32 bit integers

Associated information: 32 dummy bits

Deletion buffer size: 32

Group buffer size: 256

Merging degree k: 128

Near optimal : performance on all machines tried!

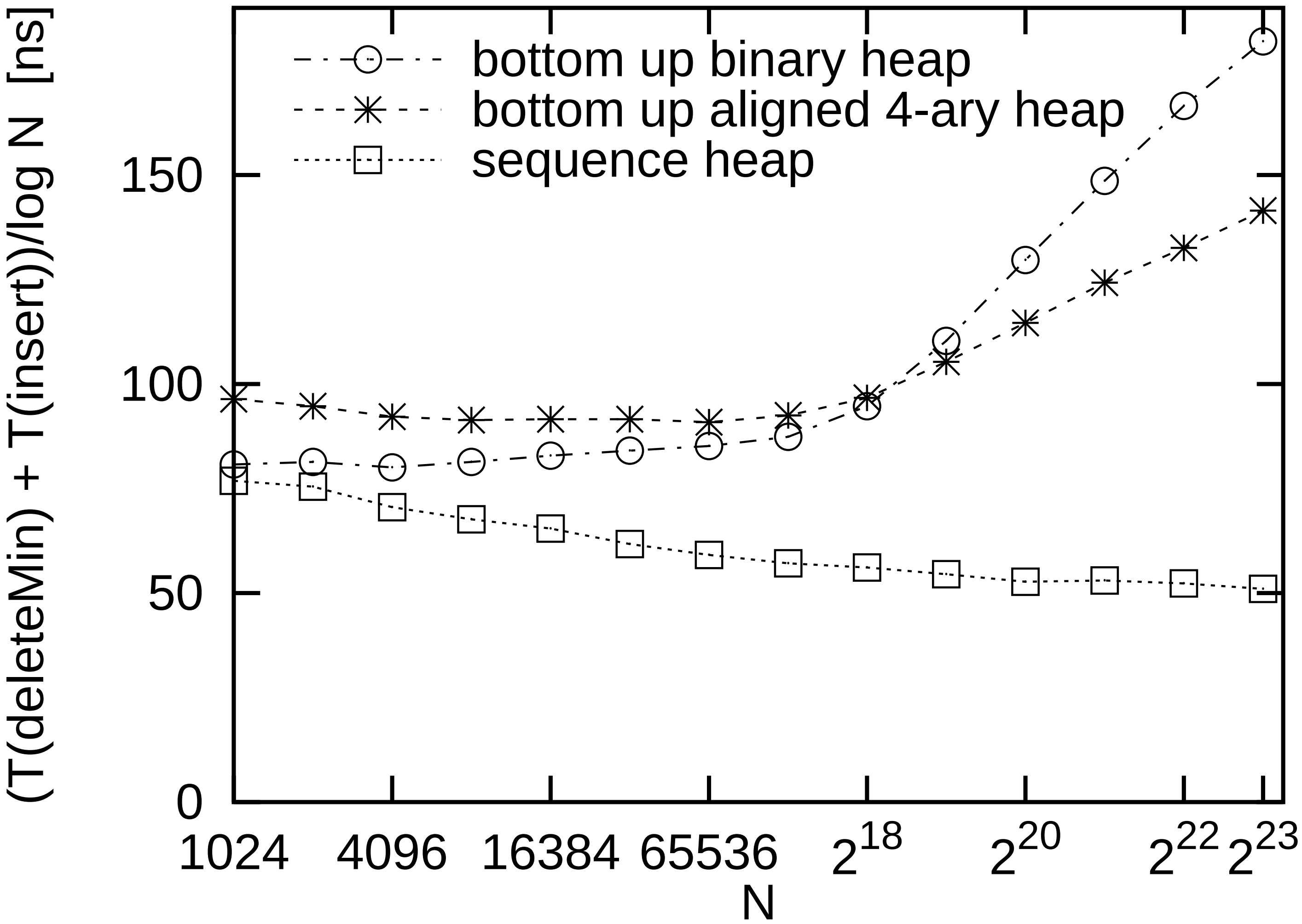
Compiler flags: Highly optimizing, nothing advanced

Operation Sequence:

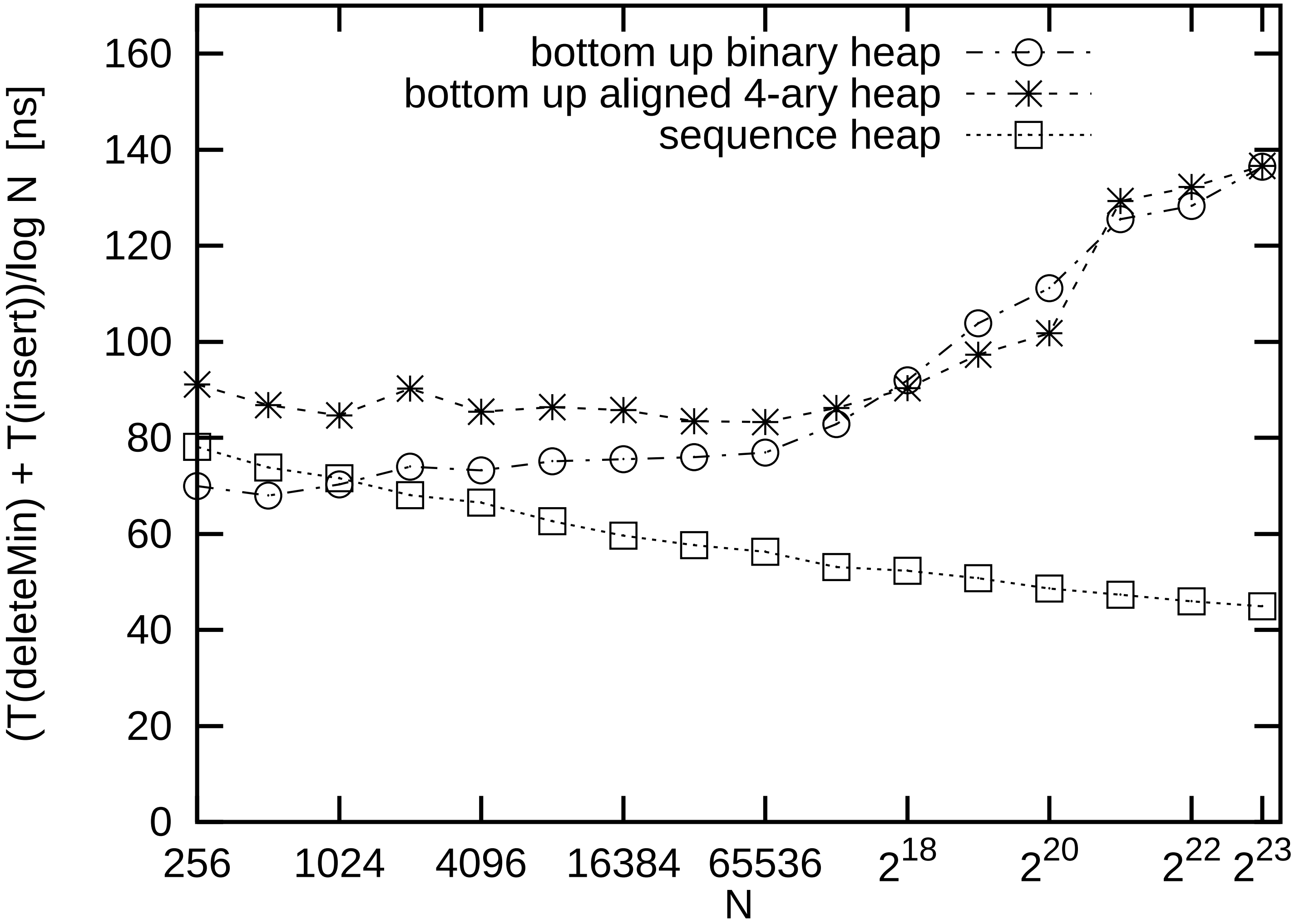
$$(\text{Insert-DeleteMin-Insert})^N(\text{DeleteMin-Insert-DeleteMin})^N$$

Near optimal performance on all machines tried!

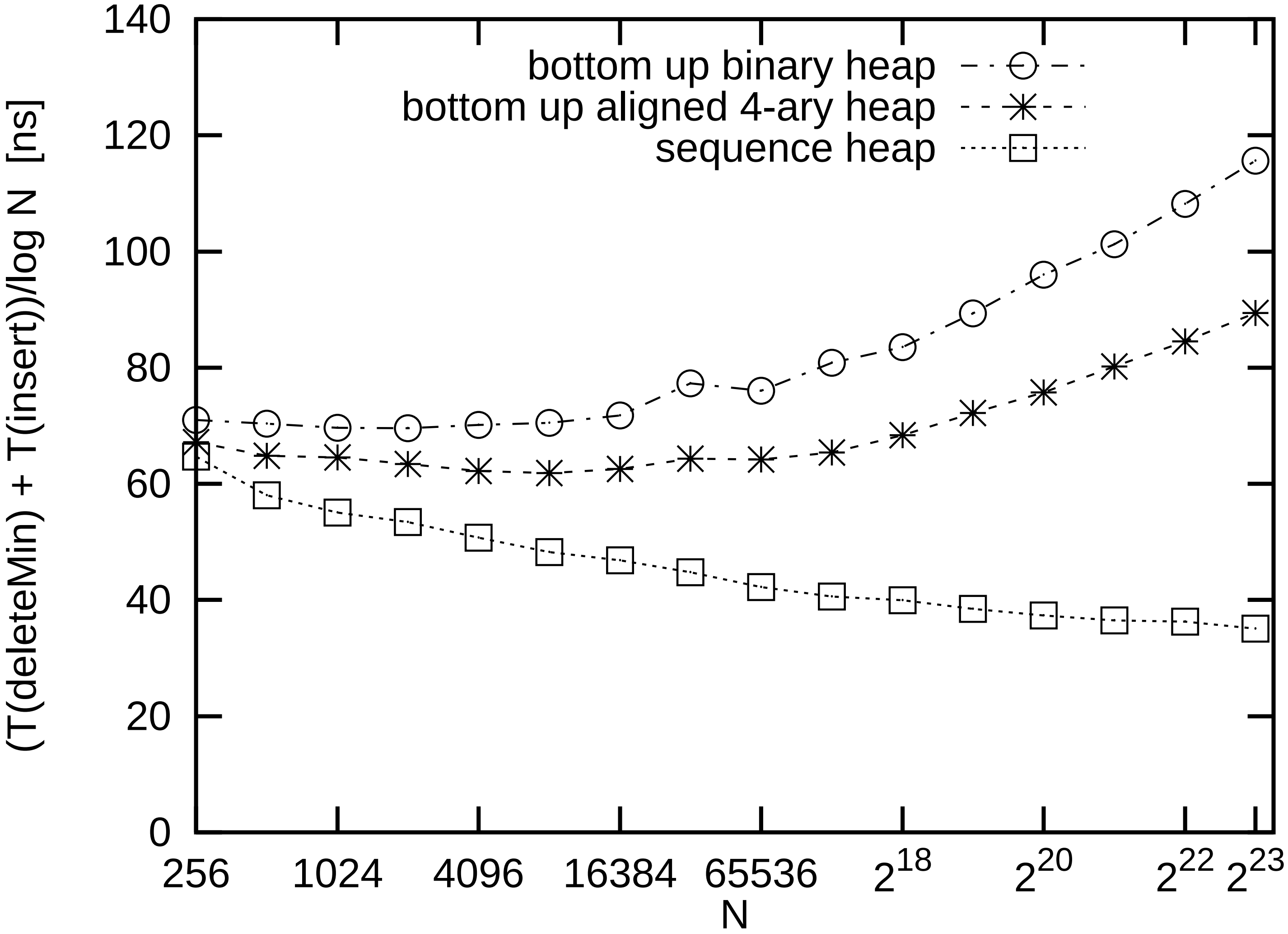
MIPS R10000, 180 MHz



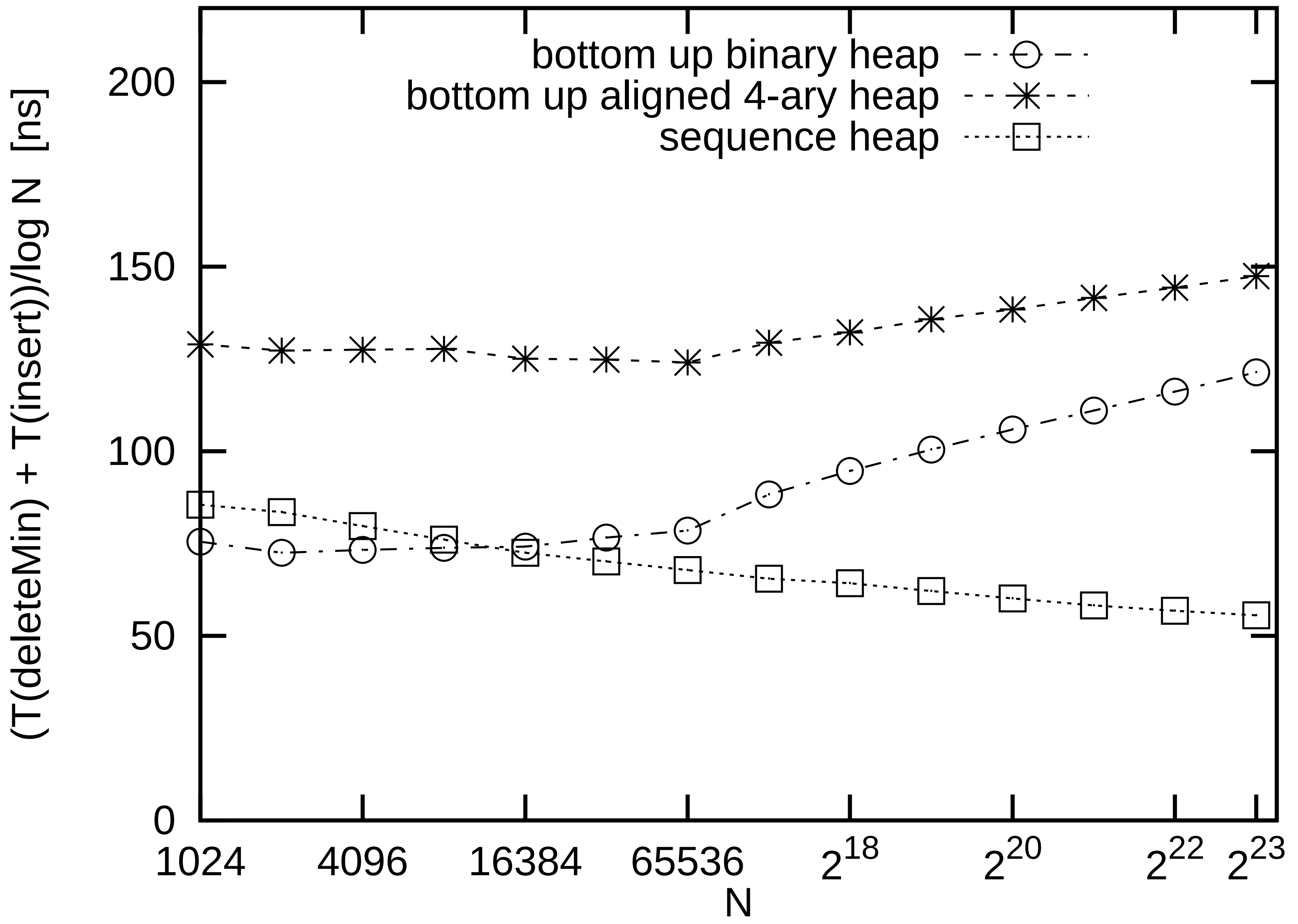
Ultra-SparcIIi, 300 MHz



Alpha-21164, 533 MHz

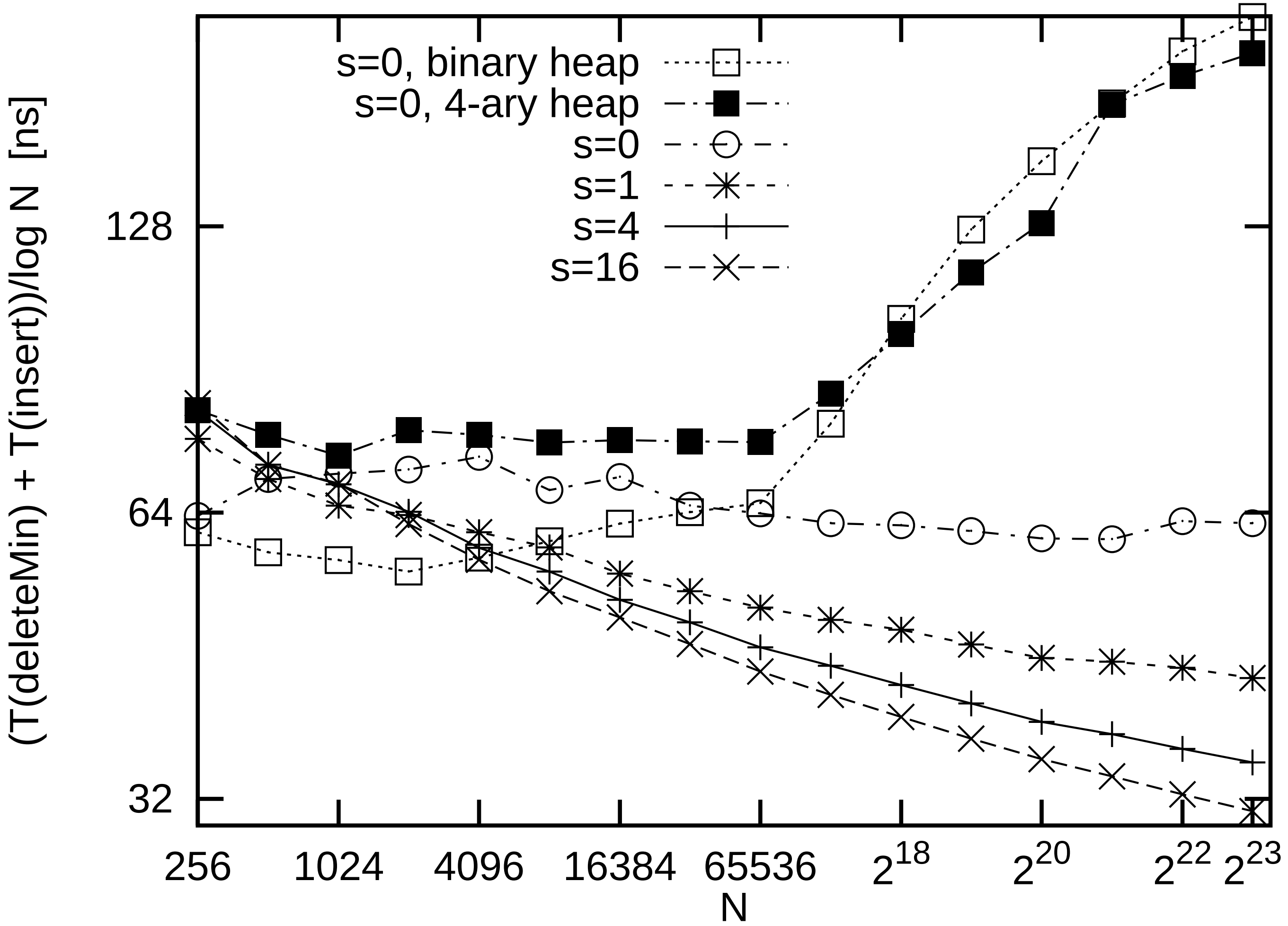


Pentium II, 300 MHz



$(\text{insert } (\text{deleteMin insert})^s)^N$

$(\text{deleteMin } (\text{insert deleteMin})^s)^N$



Methodological Lessons

If you want to compare small constant factors in execution time:

- Reproducibility demands publication of source codes (4-ary heaps, old study in Pascal)
- Highly tuned codes in particular for the competitors (binary heaps have factor 2 between good and naive implementation).
  How do you compare two mediocre implementations?
- Careful choice/description of inputs
- Use multiple different hardware platforms
- Augment with theory (e.g., comparisons, data dependencies, cache faults, locality effects . . .)

Open Problems

- Integrate into STL
- Dependence on size rather than number of insertions
- Parallel disks
- Space efficient implementation
- Multi-level cache aware or cache-oblivious variants

5 Adressable Priority Queues

Procedure build$(\{e_1,\ldots,e_n\})$ $M:=\{e_1,\ldots,e_n\}$

Function size **return** $|M|$

Procedure insert(e) $M:=M\cup\{e\}$

Function min **return** $\min M$

Function deleteMin $e:=\min M;$ $M:=M\setminus\{e\};$ **return** e

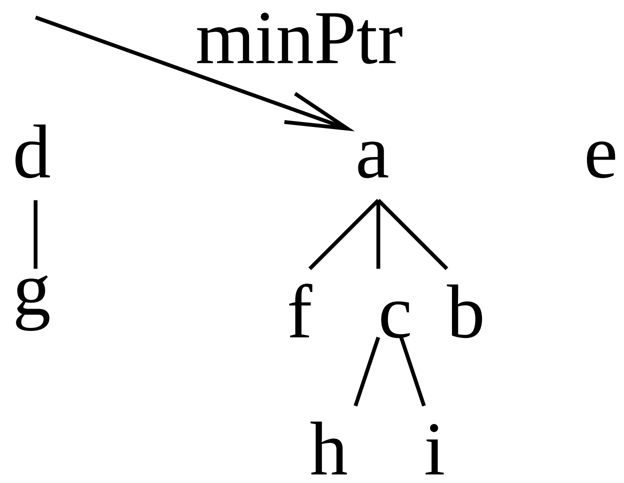
Function remove$(h:\text{Handle})$ $e:=h;$ $M:=M\setminus\{e\};$ **return** e

Procedure decreaseKey$(h:\text{Handle},k:\text{Key})$ **assert** $\text{key}(h)\geq k;$ $\text{key}(h):=k$

Procedure merge(M') $M:=M\cup M'$

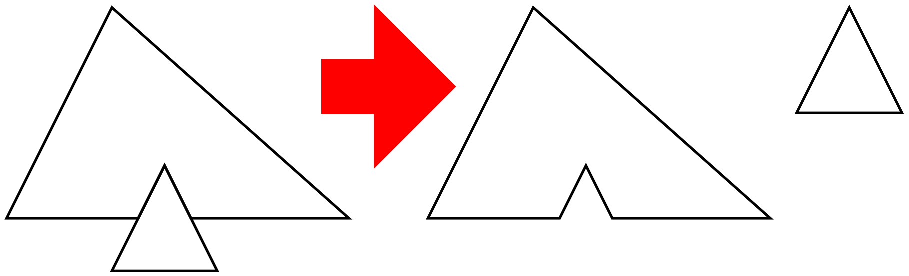
Basic Data Structure

A forest of heap-ordered trees

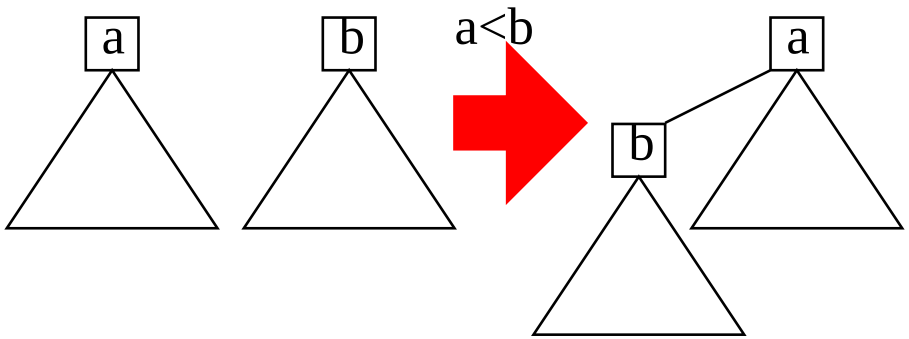


Manipulating Forests

Cut:



Link:



Pairing Heaps

Procedure insertItem$(h : \text{Handle})$

  newTree(h)

Procedure newTree$(h : \text{Handle})$

  forest:= forest $\cup \{i\}$

  if $e < \min$ **then** minPtr:= i

Pairing Heaps

Procedure decreaseKey$(h : \mathsf{Handle}, k : \mathsf{Key})$

  $\mathsf{key}(h) := k$

  if h is not a root **then** cut(h)

Pairing Heaps

Function deleteMin : Handle

m:= minPtr

forest:= forest $\setminus \{m\}$

foreach child h of m **do** newTree(h)

Perform a pairwise link of the tree roots in forest

return m

Pairing Heaps

Procedure merge(o : AdressablePQ)

  if minPtr $>$ o.minPtr **then** minPtr:= o.minPtr

  forest:= forest $\cup$ o.forest

  o.forest:= $\emptyset$

Fibonacci Heaps (A sample from the Zoo)

Ranks: initially zero, increases for root of a link

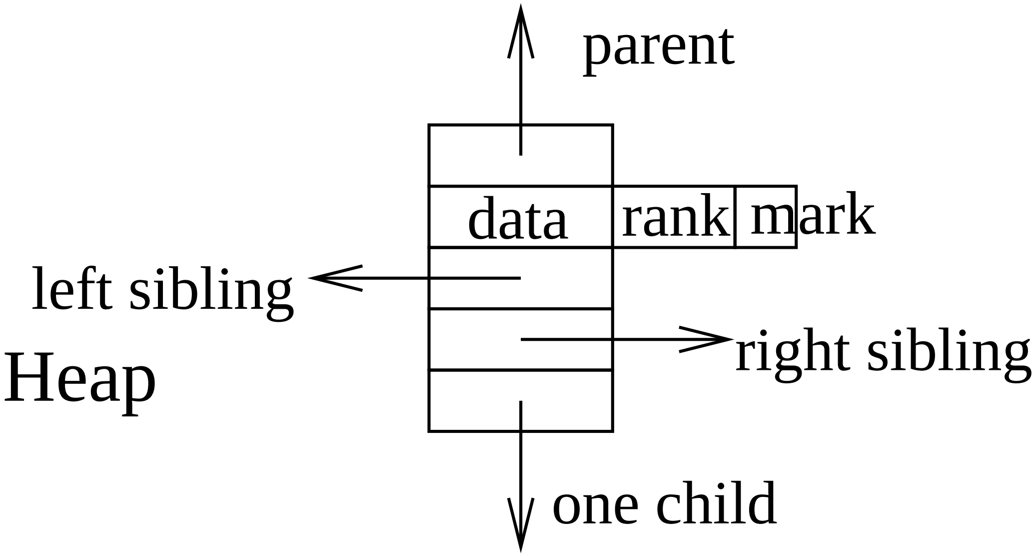
Union by rank: Only link roots of equal rank

Mark nodes that lost a child

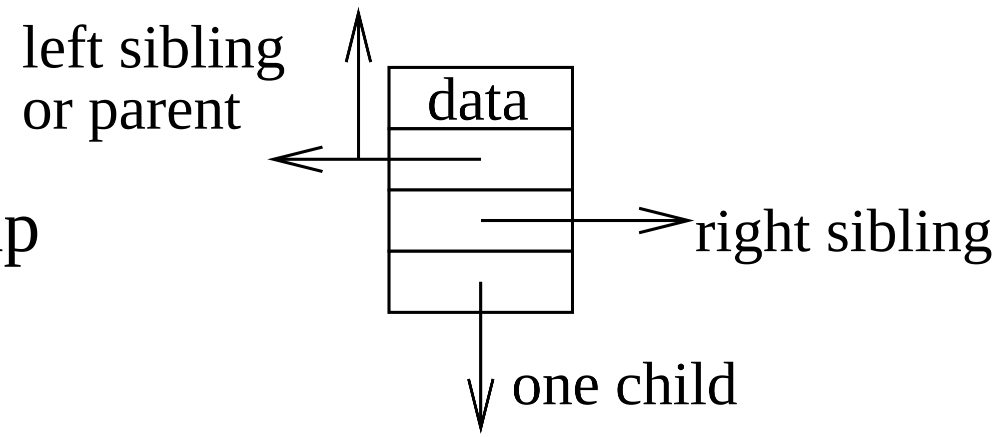
Cascading cuts: cut marked nodes (i.e., lost two childs)

Amortized complexity: $O(\log n)$ for deleteMin
$O(1)$ for all other operations

Representing the Tree Items



Fibonacci Heap

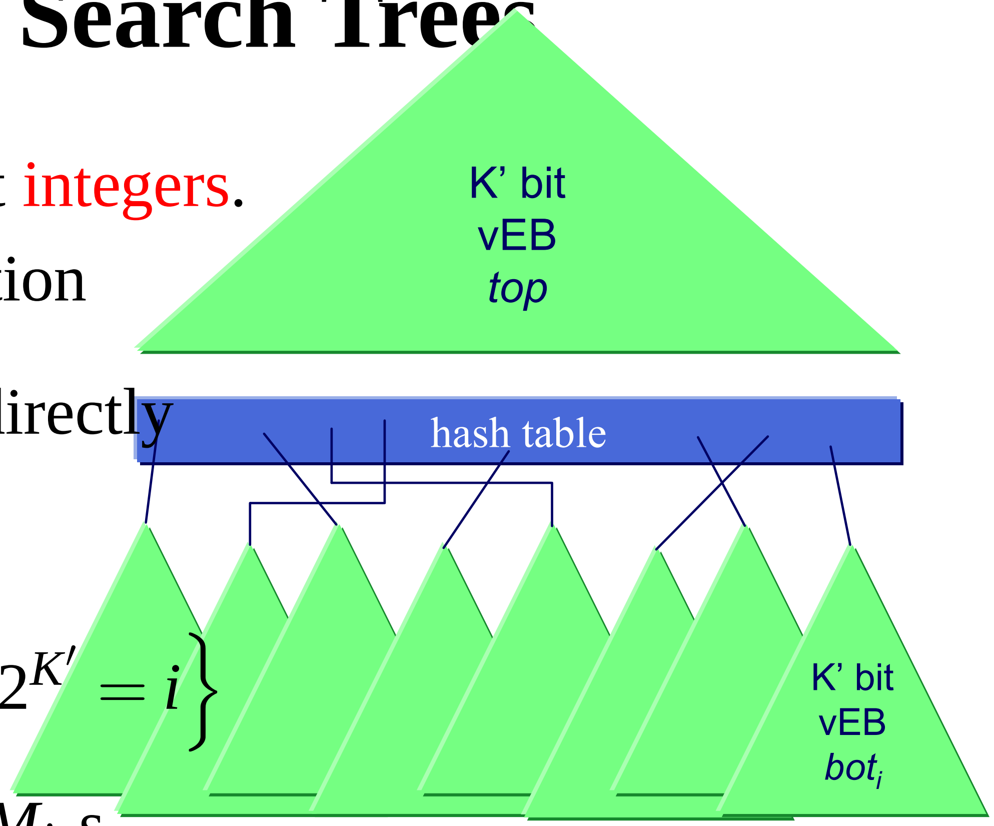


Pairing Heap

Addressable Priority Queues: Todo

- No recent comparison of efficient implementations
- No tight analysis of pairing heaps
- No implementation of compromises, e.g., thin heaps [Kaplan Tarjan 99] (three pointers, no mark, slightly more complicated balancing)
- No implementation of worst case efficient variants
- Study implementation tricks: two pointers per item? sentinels,...
- (Almost) mothing known for memory hierarchies

6 van Emde-Boas Search Trees



- Store set M of $K = 2^k$-bit integers.
  later: associated information
- $K = 1$ or $|M| = 1$: store directly
- $K' := K/2$
- $M_i := \left\{ x \bmod 2^{K'} : x \,\mathbf{div}\, 2^{K'} = i \right\}$
- root points to nonempty M_i-s
- top $t = \{i : M_i \neq \emptyset\}$
- insert, delete, search in $O(\log K)$ time

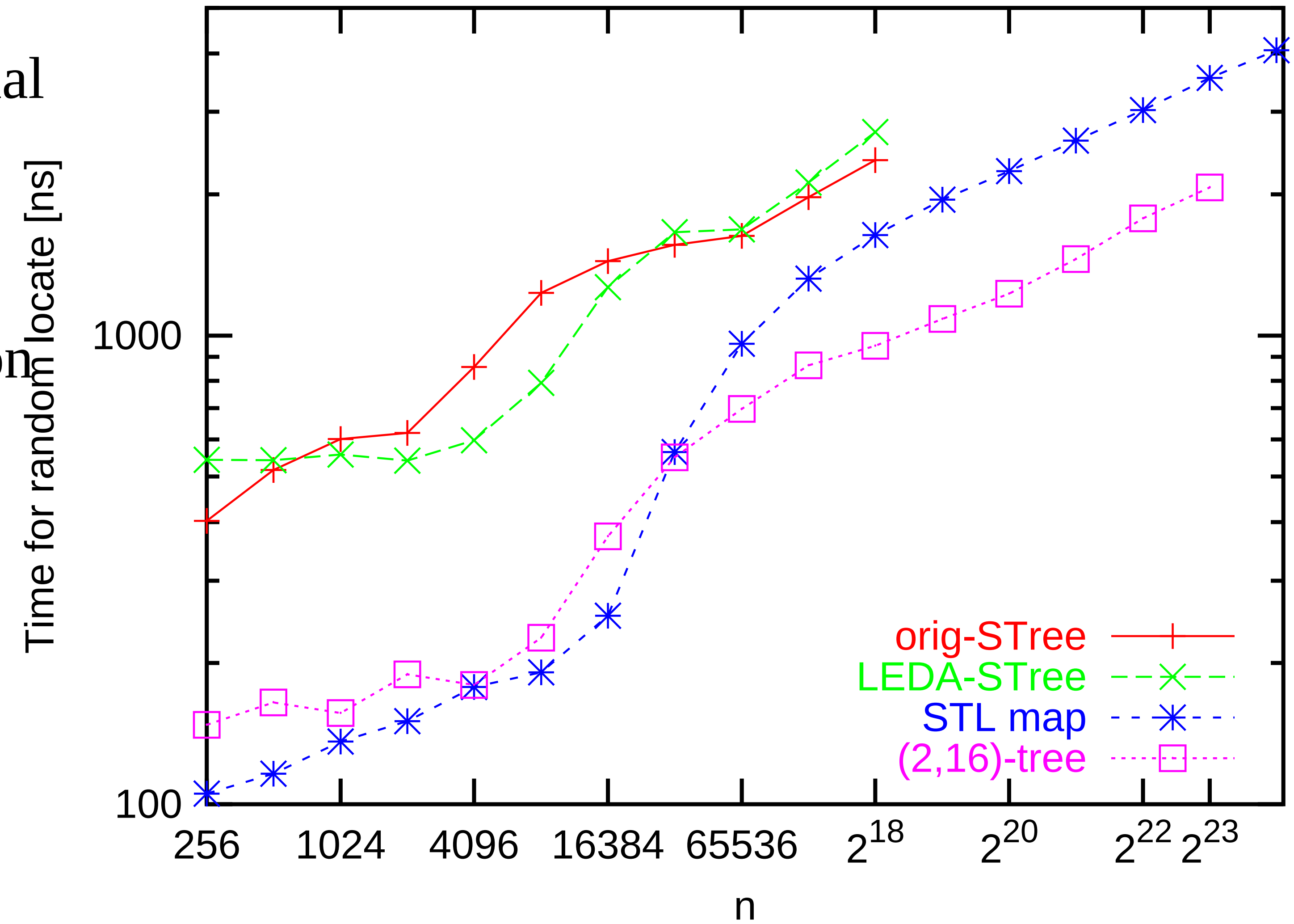
Comparison with Comparison Based Search Trees

Ideally: $\log n \rightsquigarrow \log\log n$

Problems:

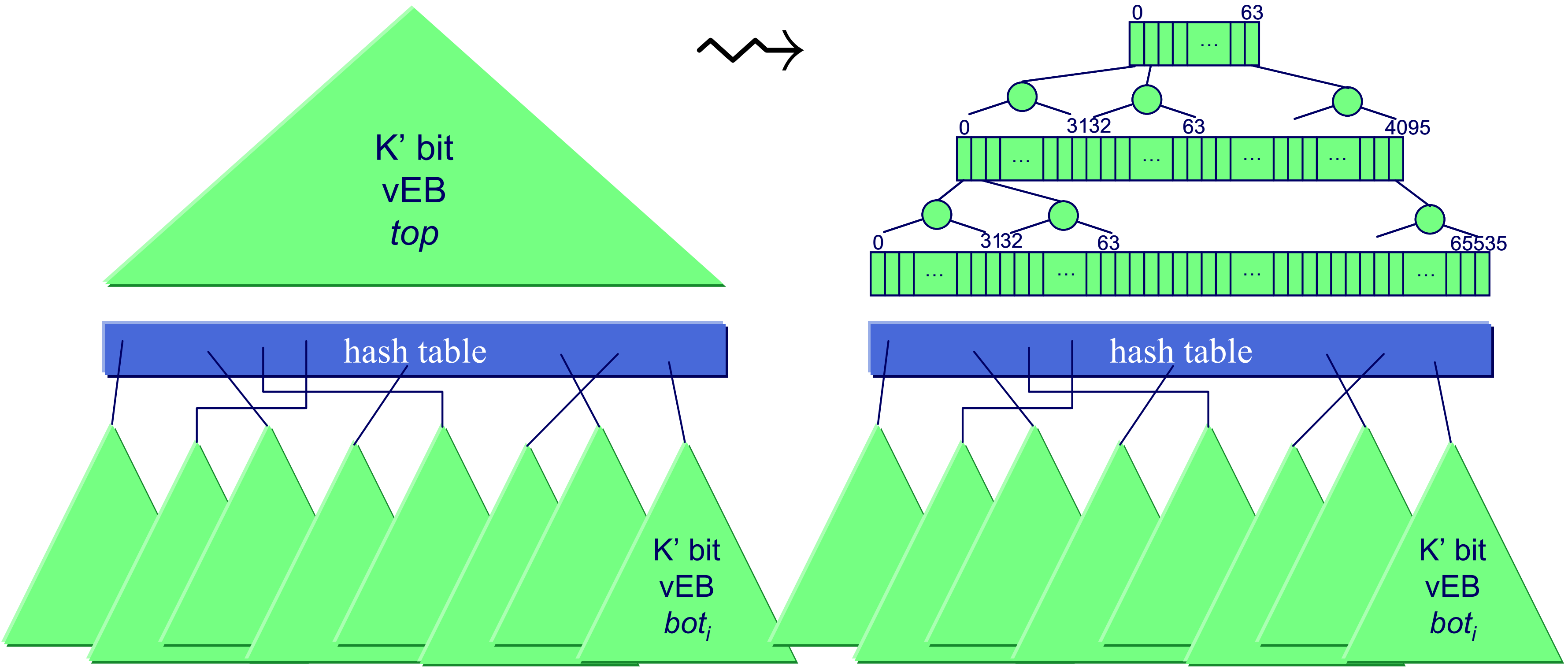
Many special case tests

High space consumption



Efficient 32 bit Implementation

top ⇝ 3 layers of bit arrays



Layers of Bit Arrays

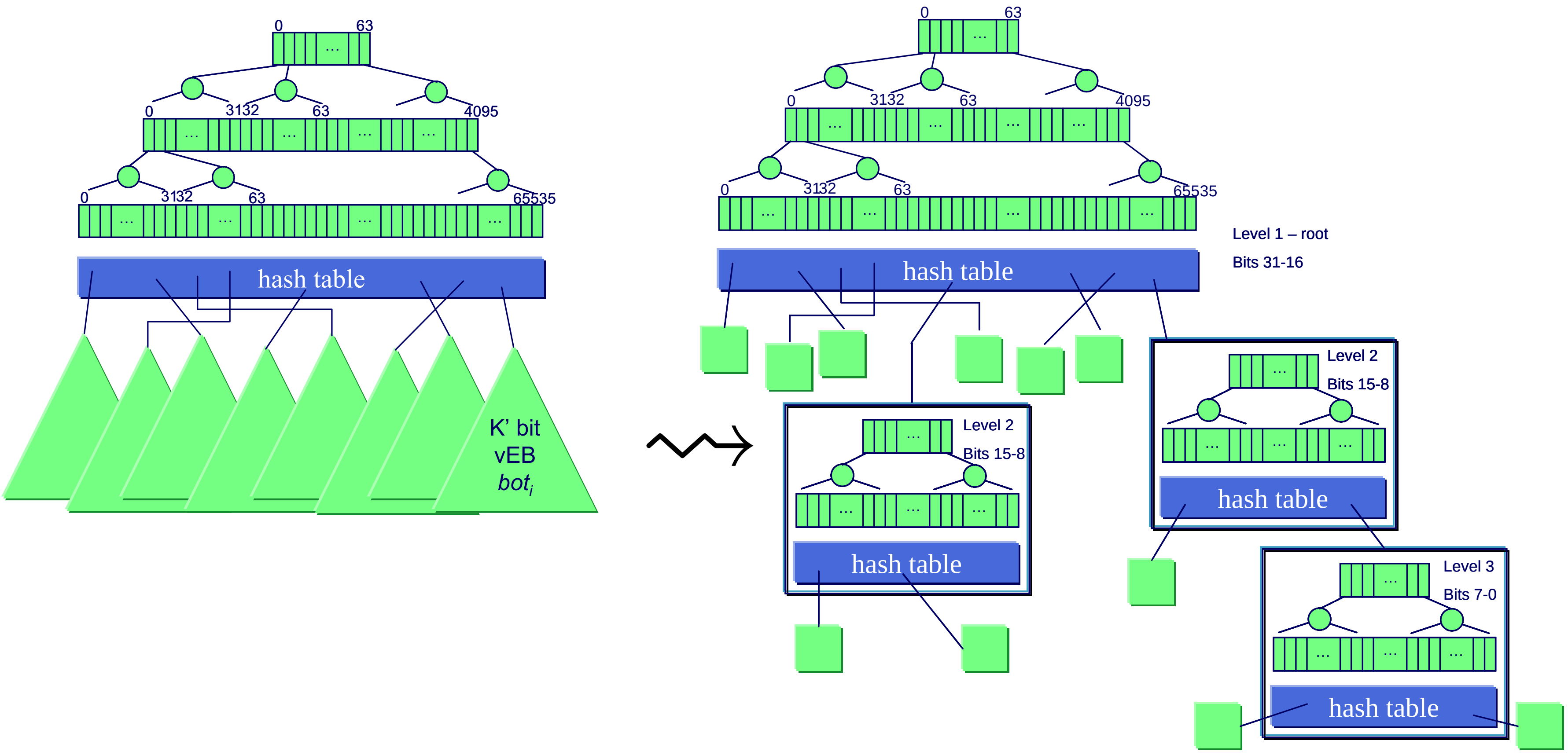
$t^1[i] = 1$ iff $M_i \neq \emptyset$

$t^2[i] = t^1[32i] \vee t^1[32i+1] \vee \cdots \vee t^1[32i+31]$

$t^3[i] = t^2[32i] \vee t^2[32i+1] \vee \cdots \vee t^2[32i+31]$

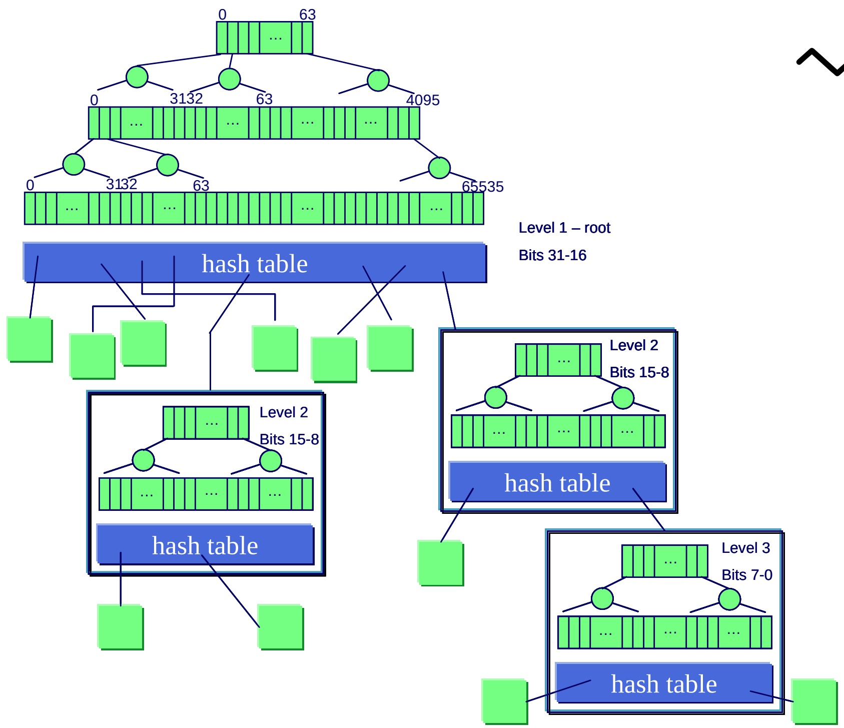
Effi cient 32 bit Implementation

Break recursion after 3 layers



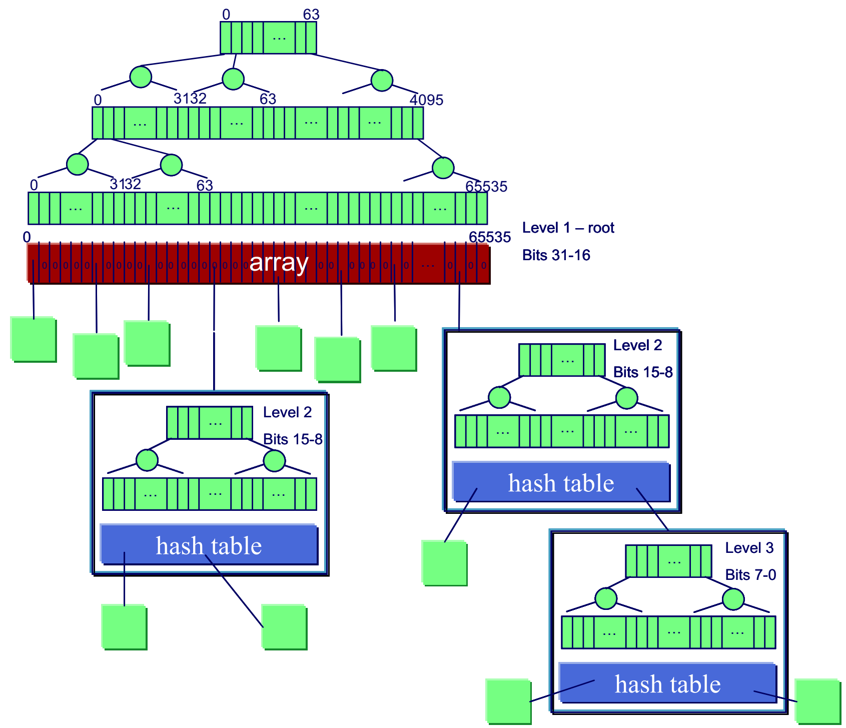
Efficient 32 bit Implementation

root hash table



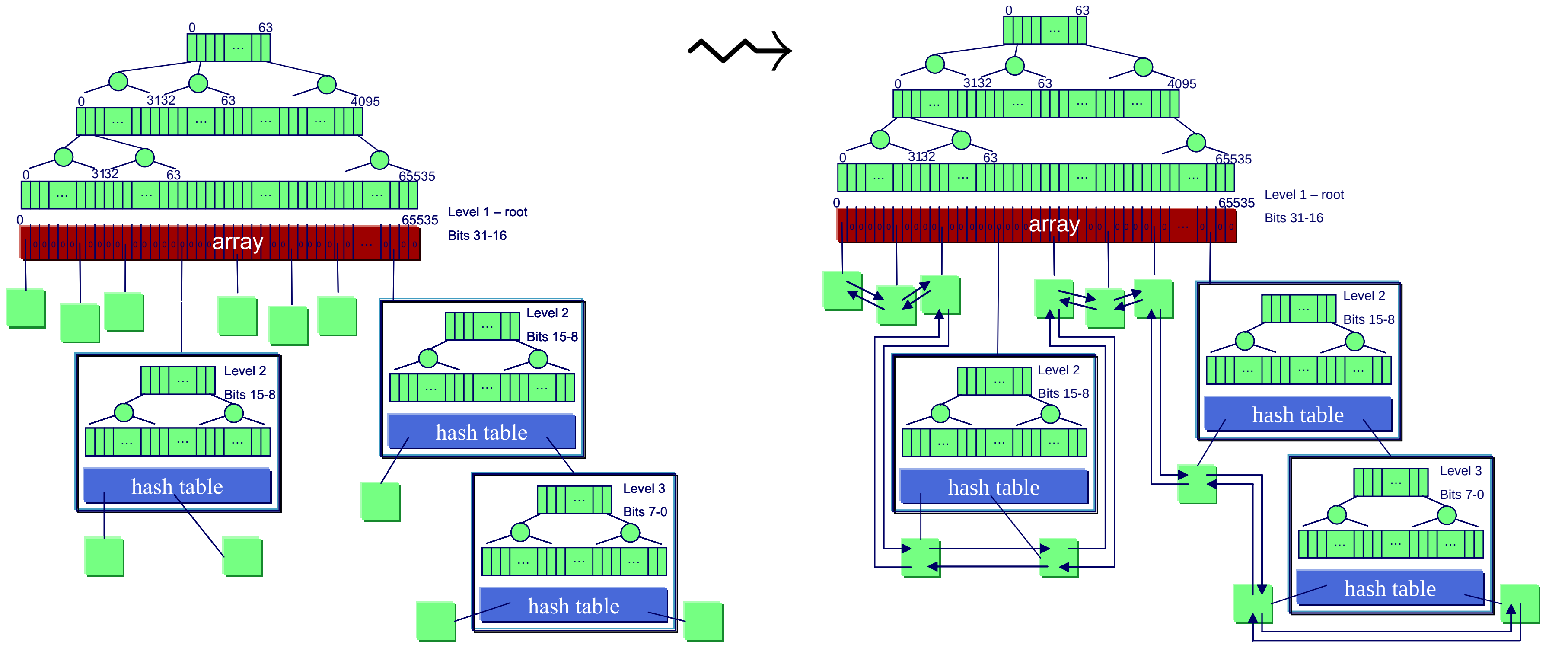
⇝

root array

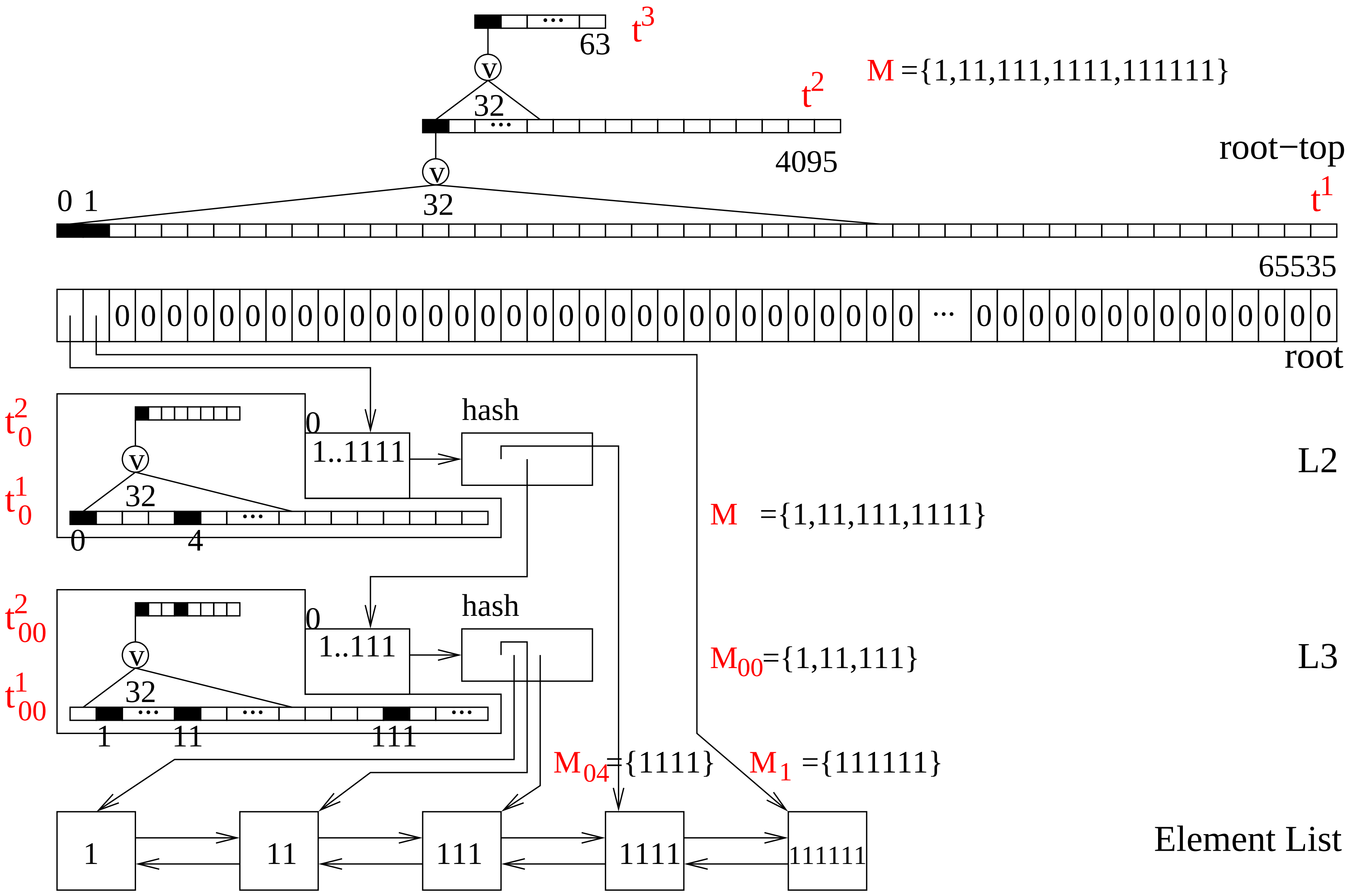


Efficient 32 bit Implementation

Sorted doubly linked lists for associated information and range queries



Example



63
t^3
$M = \{1,11,111,1111,111111\}$
32
t^2
4095
root−top
0 1
32
t^1
65535
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ··· 0 0 0 0 0 0 0 0 0 0 0 0 0 0
root
t^2_0
t^1_0
32
0 4
0
1..1111
hash
L2
$M = \{1,11,111,1111\}$
t^2_{00}
t^1_{00}
32
1 11 111
0
1..111
hash
$M_{00} = \{1,11,111\}$
L3
$M_{04} = \{1111\}$
$M_1 = \{111111\}$
1
11
111
1111
111111
Element List

Locate High Level

$//$return handle of $\min x \in M : y \leq x$

Function locate$(y : \mathbb{N})$: ElementHandle

if $y > \max M$ **then return** ∞

$i := y[16..31]$ // Level 1

if $r[i] = \mathsf{nil} \vee y > \max M_i$ **then return** $\min M_{t^1.\mathsf{locate}(i)}$

if $M_i = \{x\}$ **then return** x

$j := y[8..15]$ // Level 2

if $r_i[j] = \mathsf{nil} \vee y > \max M_{ij}$ **then return** $\min M_{i,t_i^1.\mathsf{locate}(j)}$

if $M_{ij} = \{x\}$ **then return** x

return $r_{ij}[t_{ij}^1.\mathsf{locate}(y[0..7])]$ // Level 3

Locate in Bit Arrays

// find the smallest $j \geq i$ such that $t^k[j] = 1$

Method locate(i) for a bit array t^k consisting of n bit words

  // $n = 32$ for $t^1, t^2, t_i^1, t_{ij}^1$; $n = 64$ for t^3; $n = 8$ for t_i^2, t_{ij}^2

  assert some bit in t^k to the right of i is nonzero

  $j := i$ **div** n // which word?

  $a := t^k[nj..nj + n - 1]$

  set $a[(i \bmod n) + 1..n - 1]$ to zero // $n - 1 \cdots i \bmod n \cdots 0$

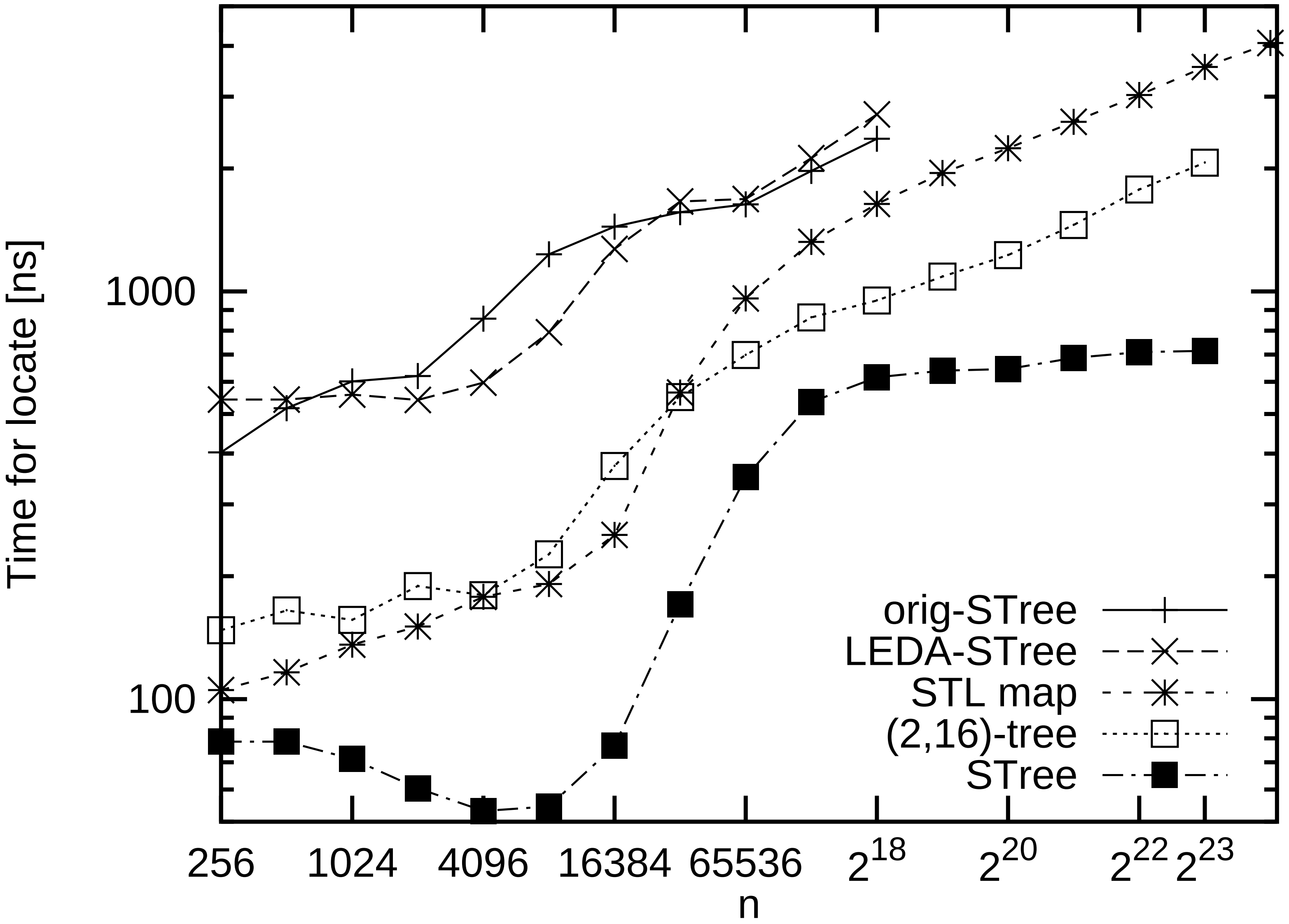
  if $a = 0$ **then**

    $j := t^{k+1}$.locate(j)

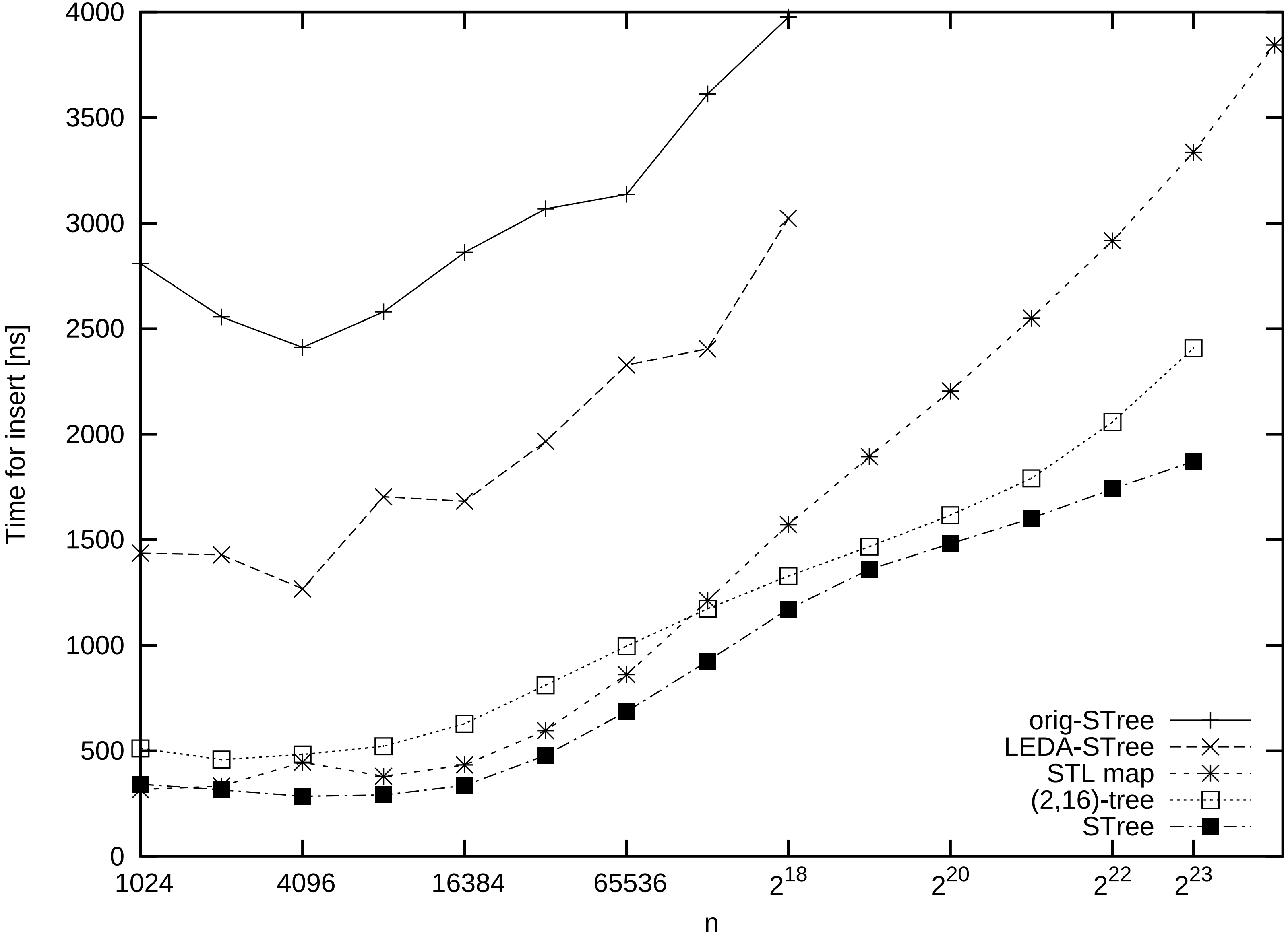
    $a := t^k[nj..nj + n - 1]$

  return $nj +$ msbPos(a) // e.g. floating point conversion

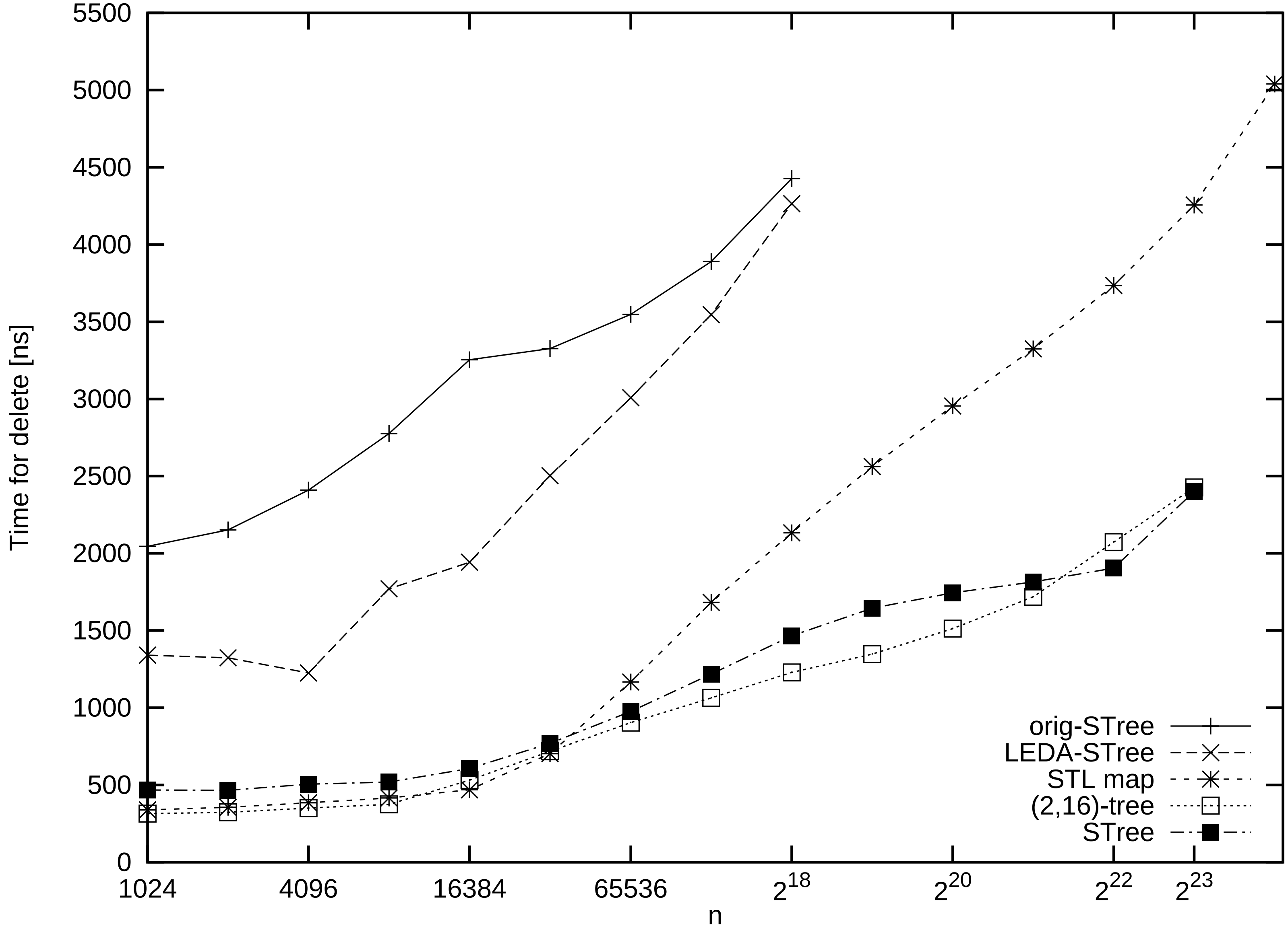
Random Locate



Random Insert



Delete Random Elements



Open Problems

- Measurement for “worst case” inputs
- Measure Performance for realistic inputs
  - IP lookup etc.
  - Best first heuristics like, e.g., bin packing
- More space efficient implementation
- (A few) more bits

7 Hashing

"to hash" ≈ "to bring into complete disorder"

paradoxically, this helps us to find things more easily!

store set $M \subseteq \text{Element}$.

$\text{key}(e)$ is unique for $e \in M$.

support dictionary operations in $O(1)$ time.

$M.\text{insert}(e : \text{Element})$: $M := M \cup \{e\}$

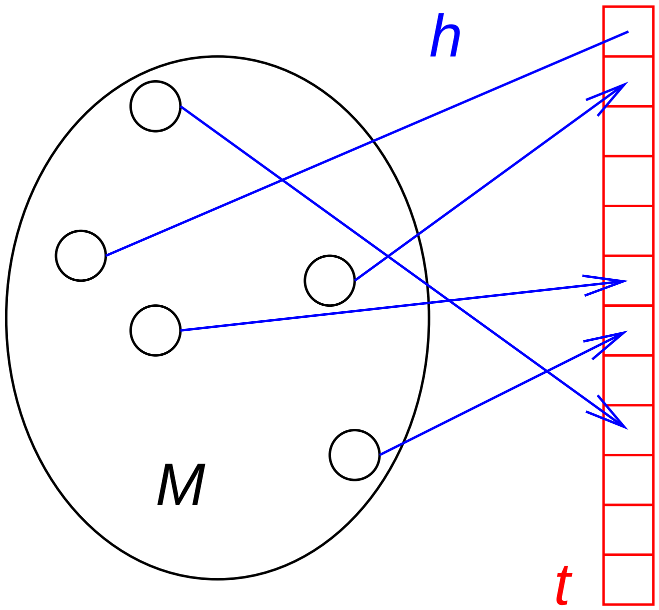
$M.\text{remove}(k : \text{Key})$: $M := M \setminus \{e\}, e = k$

$M.\text{find}(k : \text{Key})$: return $e \in M$ with $e = k$; $\perp$ if none present

(Convention: key is *implicit*), e.g. $e = k$ iff $\text{key}(e) = k$)

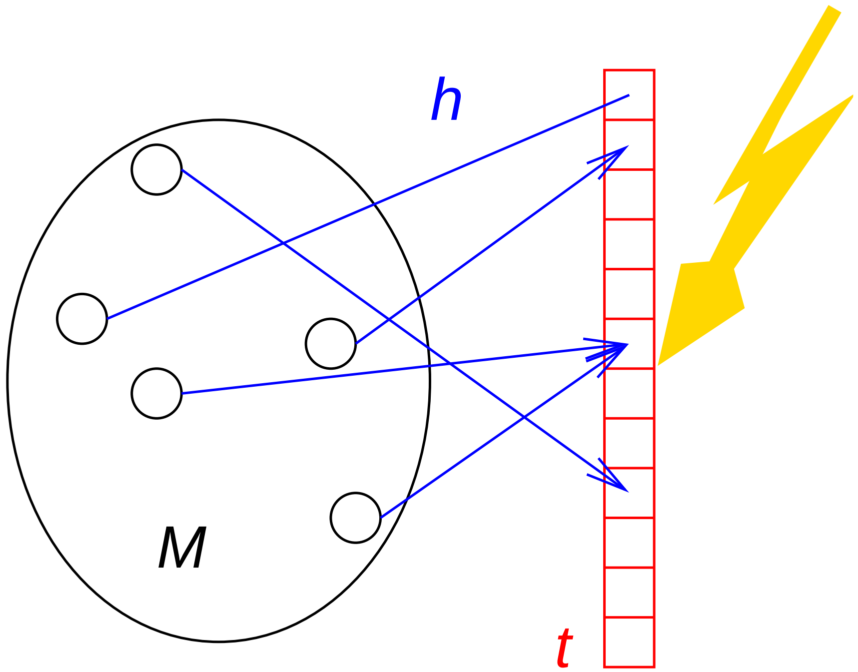
An (Over)optimistic approach

A (perfect) hash function h
maps elements of M to
unique entries of table $t[0..m-1]$, i.e.,
$t[h(\mathsf{key}(e))] = e$



Collisions

perfect hash functions are difficult to obtain

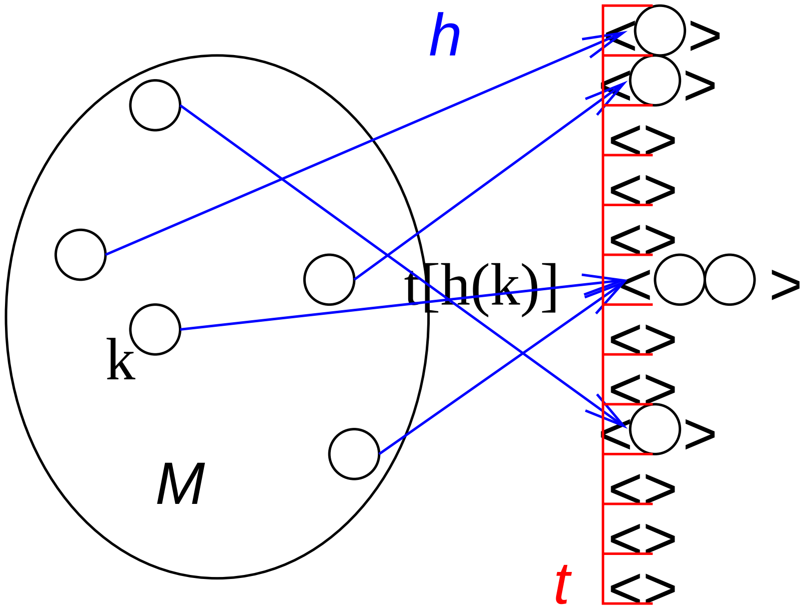


Example: Birthday Paradoxon

Collision Resolution

for example by closed hashing

entries: elements ⇝ sequences of elements



Hashing with Chaining

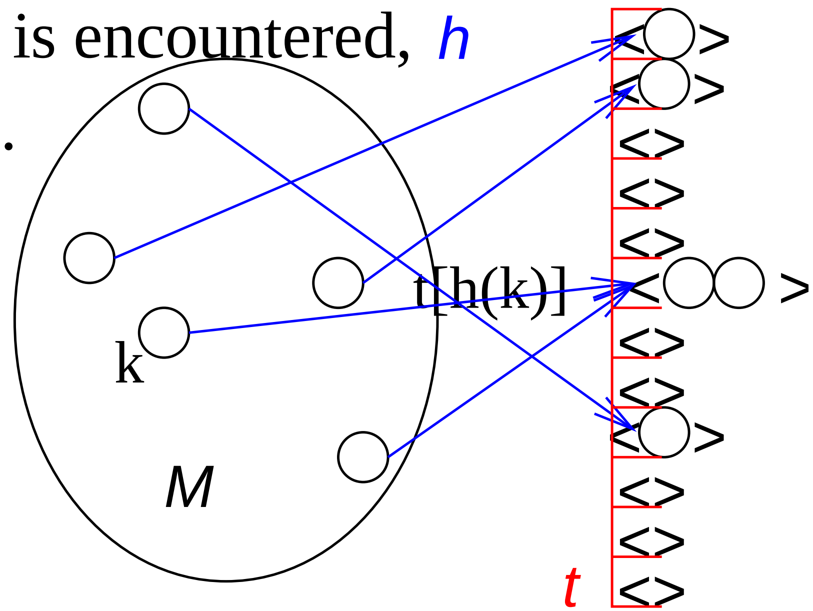
Implement sequences in closed hashing by singly linked lists

insert(e): Insert e at the beginning of $t[h(e)]$. constant time

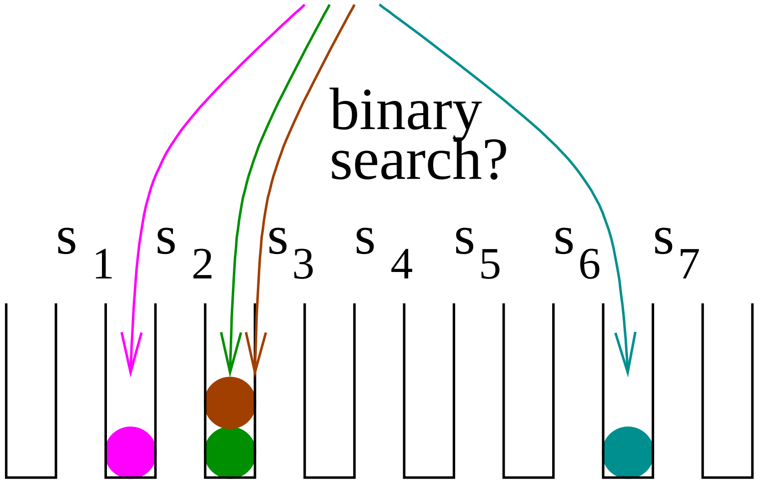
remove(k): Scan through $t[h(k)]$. If an element e with $h(e) = k$ is encountered, remove it and return.

find(k) : Scan through $t[h(k)]$.
If an element e with $h(e) = k$ is encountered, return it. Otherwise, return $\perp$.

$O(|M|)$ worst case time for remove and find



A Review of Probability



Example from hashing

sample space Ω — random hash functions$\{0..m-1\}^{\mathrm{Key}}$

events: subsets of Ω — $\mathcal{E}_{42} = \{h \in \Omega : h(4) = h(2)\}$

p_x =probability of $x \in \Omega$ — uniform distr. $p_h = m^{-|\mathrm{Key}|}$

$\mathbb{P}[\mathcal{E}] = \sum_{x \in E} p_x$ — $\mathbb{P}[\mathcal{E}_{42}] = \frac{1}{m}$

random variable $X_0 : \Omega \to \mathbb{R}$ — $X = |\{e \in M : h(e) = 0\}|$.

expectation $E[X_0] = \sum_{y\in\Omega} p_y X(y)$ $\qquad E[X] = \frac{|M|}{m}$ (*)

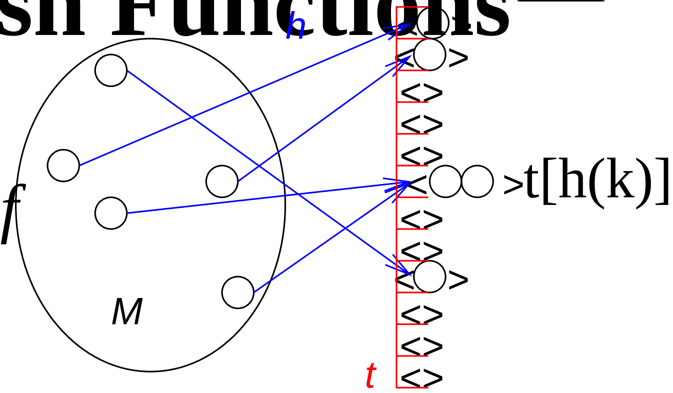
Linearity of Expectation: $E[X+Y] = E[X] + E[Y]$

Proof of (*):

Consider the 0-1 RV $X_e = 1$ if $h(e) = 0$ for $e \in M$ and $X_e = 0$ else.

$$E[X_0] = E[\sum_{e\in M} X_e] = \sum_{e\in M} E[X_e] = \sum_{e\in M} \mathbb{P}[X_e = 1] = |M| \cdot \frac{1}{m}$$

Analysis for Random Hash Functions



Satz 1. *The expected execution time of* remove(k) *and* find(k) *is* $O(1)$ *if* $|M| = O(m)$.

Proof. Constant time plus the time for scanning $t[h(k)]$.

$X := |t[h(k)]| = |\{e \in M : h(e) = h(k)\}|$.

Consider the 0-1 RV $X_e = 1$ if $h(e) = h(k)$ for $e \in M$ and $X_e = 0$ else.

$$\mathrm{E}[X] = \mathrm{E}[\sum_{e \in M} X_e] = \sum_{e \in M} \mathrm{E}[X_e] = \sum_{e \in M} \mathbb{P}[X_e = 1] = \frac{|M|}{m}$$
$$= O(1)$$

This is independent of the input set M. □

Universal Hashing

Idea: use only certain “easy” hash functions

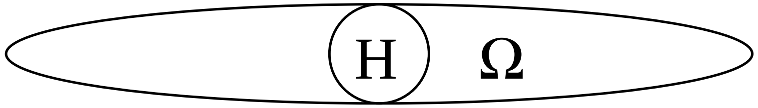
Definition:

$\mathcal{U} \subseteq \{0..m-1\}^{\mathsf{Key}}$ is *universal*
if for all x, y in Key with $x \neq y$ and random $h \in \mathcal{U}$,

$$\mathbb{P}[h(x) = h(y)] = \frac{1}{m} \quad .$$

Satz 2. *Theorem 1 also applies to universal families of hash functions.*

Proof. For $\Omega = \mathcal{U}$ we still have $\mathbb{P}[X_e = 1] = \frac{1}{m}$.
The rest is as before. □


H
Ω

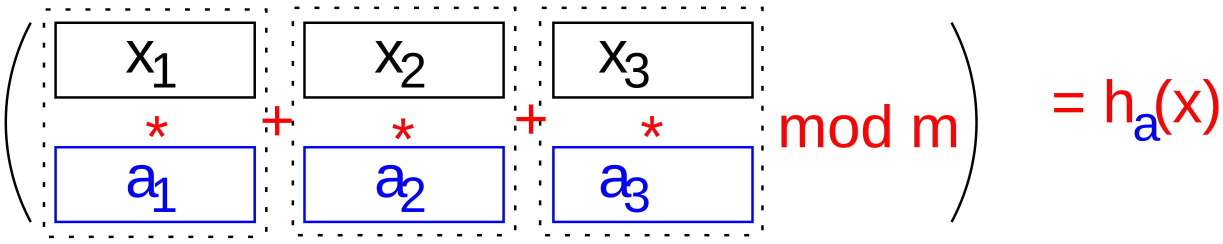
A Simple Universal Family

Assume m is prime, Key $\subseteq \{0,\ldots,m-1\}^k$

Satz 3. *For* $\mathbf{a} = (a_1,\ldots,a_k) \in \{0,\ldots,m-1\}^k$ *define*

$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a}\cdot\mathbf{x} \bmod m$, $H^{\cdot} = \left\{h_{\mathbf{a}} : \mathbf{a} \in \{0..m-1\}^k\right\}$.

$H^{\cdot}$ *is a universal family of hash functions*



Proof. Consider $\mathbf{x} = (x_1, \ldots, x_k)$, $\mathbf{y} = (y_1, \ldots, y_k)$ with $x_j \neq y_j$
count **a**-s with $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.
For each choice of a_is, $i \neq j$, $\exists$ exactly one a_j with
$h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$:

$$\sum_{1 \le i \le k} a_i x_i \equiv \sum_{1 \le i \le k} a_i y_i (\mathbf{\ mod\ } m)$$

$$\Leftrightarrow a_j(x_j - y_j) \equiv \sum_{i \neq j, 1 \le i \le k} a_i(y_i - x_i) (\mathbf{\ mod\ } m)$$

$$\Leftrightarrow a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j, 1 \le i \le k} a_i(y_i - x_i) (\mathbf{\ mod\ } m)$$

m^{k-1} ways to choose the a_i with $i \neq j$.
m^k is total number of **a**s, i.e.,

$$\mathbb{P}\left[h_{\mathbf{a}}(x) = h_{\mathbf{a}}(\mathbf{y})\right] = \frac{m^{k-1}}{m^k} = \frac{1}{m}.$$

□

Bit Based Universal Families

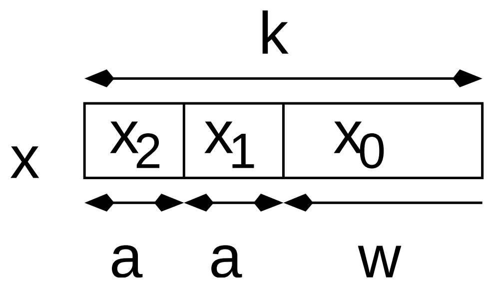
Let $m = 2^w$, Key $= \{0,1\}^k$

Bit-Matrix Multiplication: $H^{\oplus} = \left\{ h_{\mathbf{M}} : \mathbf{M} \in \{0,1\}^{w \times k} \right\}$

where $h_{\mathbf{M}}(\mathbf{x}) = \mathbf{M}x$ (arithmetics mod 2, i.e., xor, and)

Table Lookup: $H^{\oplus[]} = \left\{ h^{\oplus[]}_{(t_1,\ldots,t_b)} : t_i \in \{0..m-1\}^{\{0..w-1\}} \right\}$

where $h^{\oplus[]}_{(t_1,\ldots,t_b)}((x_0, x_1, \ldots, x_b)) = x_0 \oplus \bigoplus_{i=1}^{b} t_i[x_i]$



Hashing with Linear Probing

Open hashing: go back to original idea.

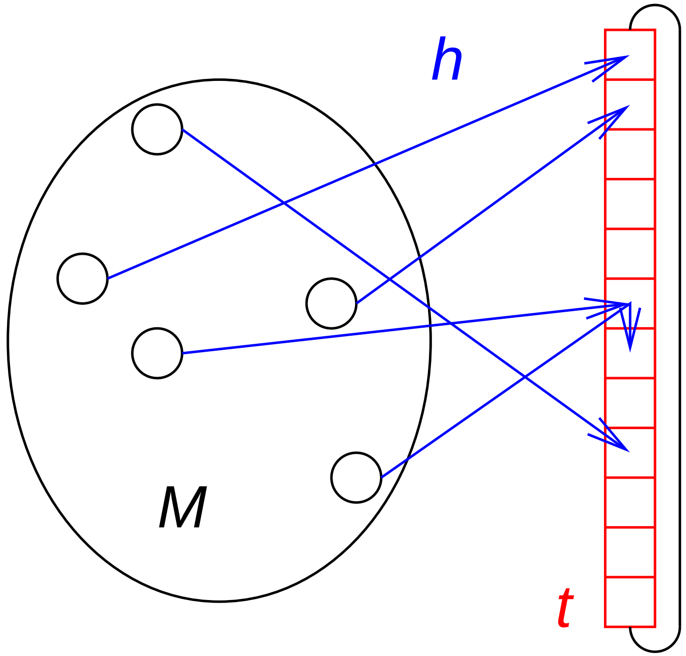
Elements are directly stored in the table.

Collisions are resolved by finding other entries.

linear probing: search for next free place by scanning the table.

Wrap around at the end.

- simple
- space efficient
- cache efficient



The Easy Part

Class BoundedLinearProbing$(m, m' : \mathbb{N}; h : \mathsf{Key} \to 0..m-1)$

$t{=}[\perp, \ldots, \perp]$: **Array** $[0..m+m'-1]$ **of** Element

invariant $\forall i : t[i] \neq \perp : \forall j \in \{h(t[i])..i-1\} : t[i] \neq \perp$

Procedure insert$(e : \mathsf{Element})$

for $i := h(e)$ **to** ∞ **while** $t[i] \neq \perp$ **do** ;

assert $i < m + m' - 1$

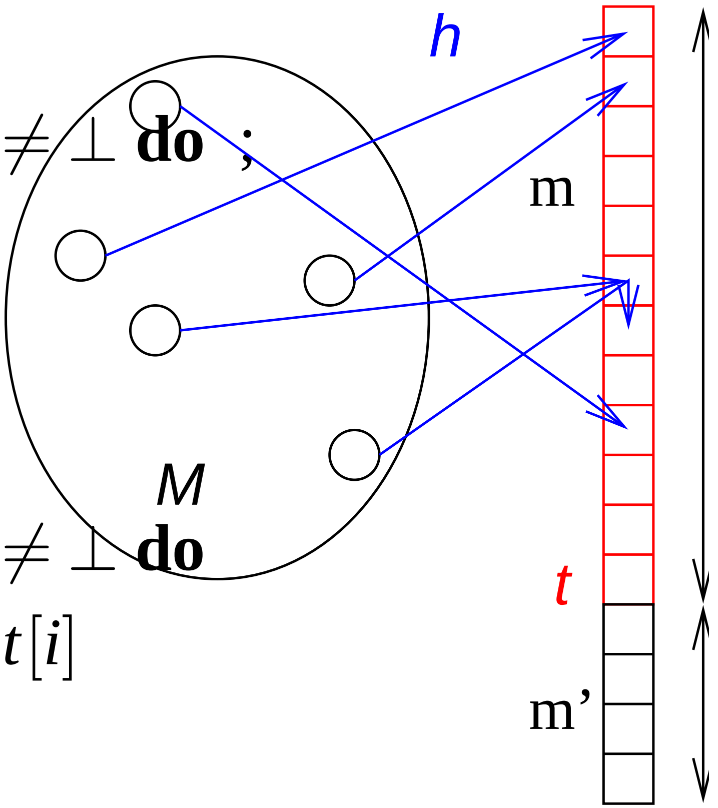
$t[i] := e$

Function find$(k : \mathsf{Key})$: Element

for $i := h(e)$ **to** ∞ **while** $t[i] \neq \perp$ **do**

if $t[i] = k$ **then return** $t[i]$

return $\perp$



Remove

example: $t = [\ldots, \underset{h(z)}{x}, y, z, \ldots]$, remove$(x)$

invariant $\forall i : t[i] \neq \bot : \forall j \in \{h(t[i])..i-1\} : t[i] \neq \bot$

Procedure remove$(k : \text{Key})$

    for $i := h(k)$ **to** ∞ **while** $k \neq t[i]$ **do** // search k

        if $t[i] = \bot$ **then return** // nothing to do

    //we plan for a hole at i.

    for $j := i+1$ **to** ∞ **while** $t[j] \neq \bot$ **do**

        //Establish invariant for $t[j]$.

        if $h(t[j]) \leq i$ **then**

            $t[i] := t[j]$ // Overwrite removed element

            $i := j$ // move planned hole

    $t[i] := \bot$ // erase freed entry

More Hashing Issues

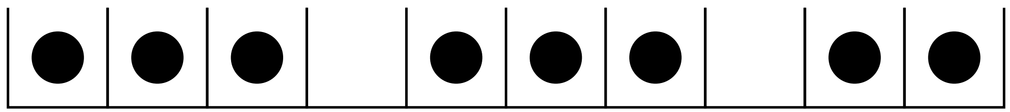
- High probability and worst case guarantees
  $\rightsquigarrow$ more requirements on the hash functions
- Hashing as a means of load balancing in parallel systems, e.g., storage servers
  – Different disk sizes and speeds
  – Adding disks / replacing failed disks without much copying

Space Efficient Hashing with Worst Case Constant Access Time

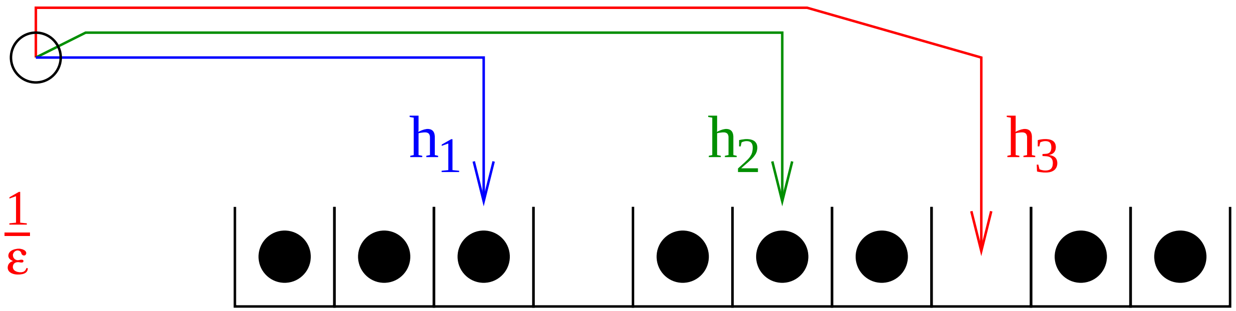
Represent a set of n elements (with associated information) using space $(1+\varepsilon)n$.

Support operations insert, delete, lookup, (doall) efficiently.

Assume a truly random hash function h

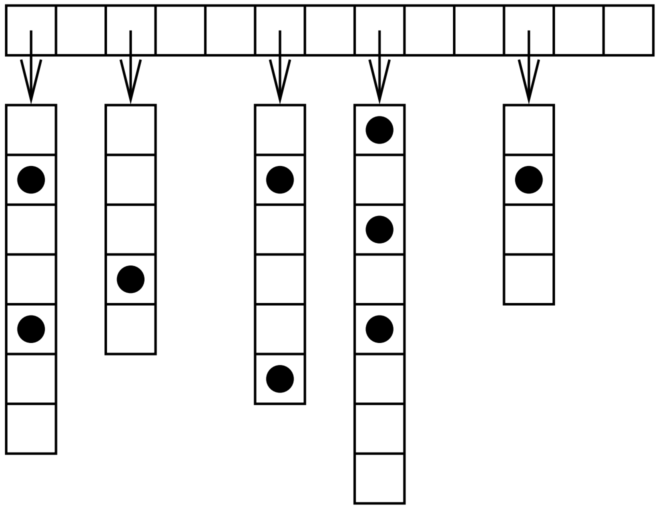


Related Work



Uniform hashing:

Expected time $\approx \frac{1}{\varepsilon}$



Dynamic Perfect Hashing,

[Dietzfelbinger et al. 94]

Worst case constant time

for lookup but ε is not small.

Approaching the Information Theoretic Lower Bound:

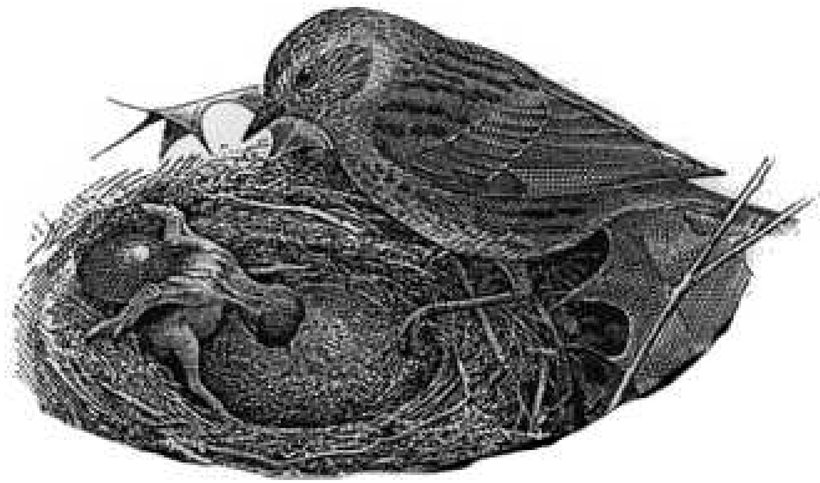
[Brodnik Munro 99,Raman Rao 02]

Space $(1+o(1))\times$lower bound without associated information

[Pagh 01] static case.

Cuckoo Hashing

[Pagh Rodler 01] Table of size 2



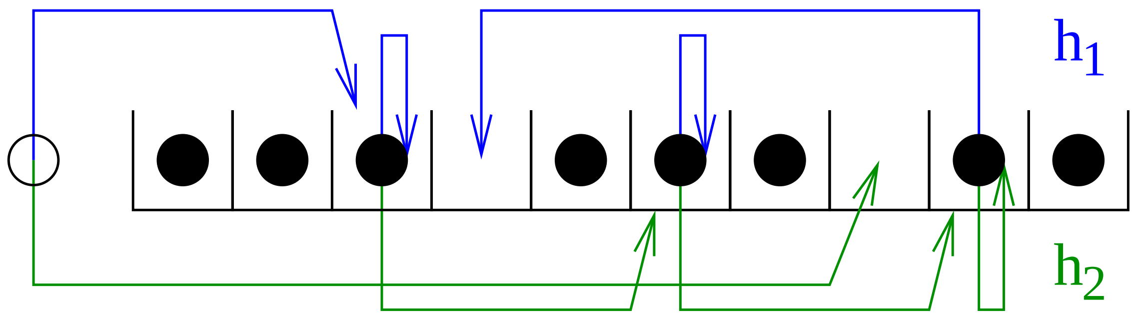
Two choices for each element.

Insert moves elements;

rebuild if necessary.

Very fast lookup and insert.

Expected constant insertion time.

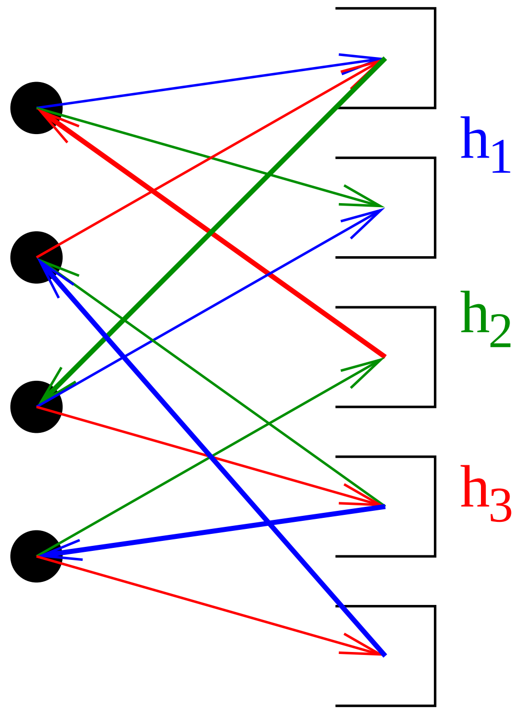


d-ary Cuckoo Hashing

d choices for each element.

Worst case d probes for delete and lookup.

Task: maintain perfect matching
in the bipartite graph
$(L = \text{Elements}, R = \text{Cells}, E = \text{Choices})$,
e.g., insert by BFS of random walk.

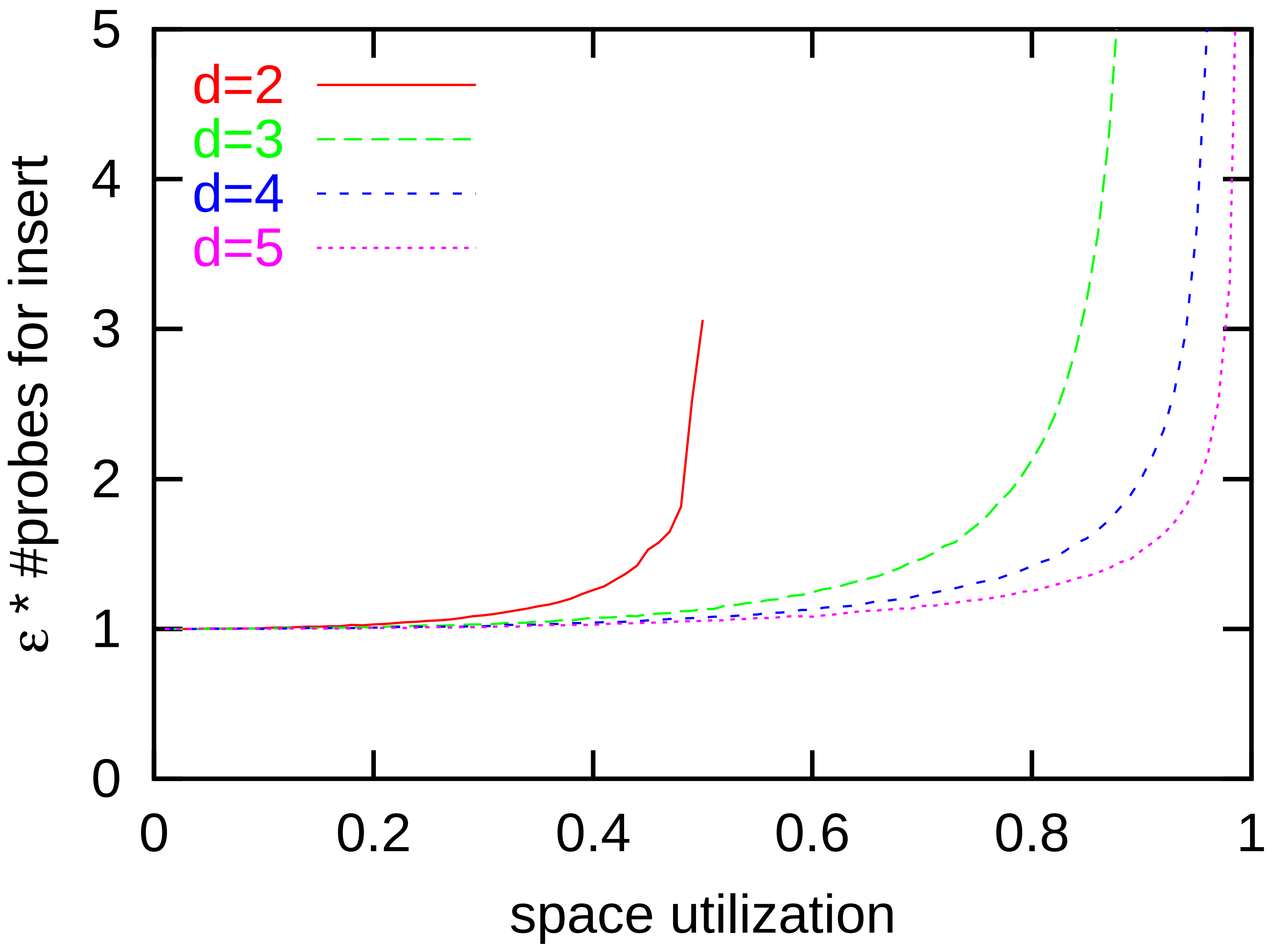


Tradeoff: Space $\leftrightarrow$ Lookup/Deletion Time

Lookup and Delete: $d = O\left(\log \frac{1}{\varepsilon}\right)$ probes

Insert: $\left(\frac{1}{\varepsilon}\right)^{O(\log(1/\varepsilon))}$, (experiments) $\longrightarrow O(1/\varepsilon)$?

Experiments



Open Questions and Further Results

- Tight analysis of insertion
- Two choices with d slots each [Dietzfelbinger et al.]
  $\rightsquigarrow$ cache efficiency

Good Implementation?

- Automatic rehash
- Always correct

8 Minimum Spanning Trees

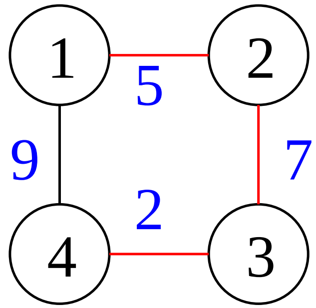
undirected Graph $G = (V, E)$.

nodes V, $n = |V|$, e.g., $V = \{1, \ldots, n\}$

edges $e \in E$, $m = |E|$, two-element subsets of V.

edge weight $c(e)$, $c(e) \in \mathbb{R}_+$.

G is connected, i.e., $\exists$ path between any two nodes.



Find a tree (V, T) with minimum weight $\sum_{e \in T} c(e)$ that connects all nodes.

MST: Overview

- Basics: Edge property and cycle property
- Jarník-Prim Algorithm
- Kruskals Algorithm
- Some tricks and comparison
- Advanced algorithms using the cycle property
- External MST

Applications: Clustering; subroutine in combinatorial optimization, e.g., Held-Karp lower bound for TSP. Challenging real world instances???

Anyway: almost ideal “fruit fly” problem

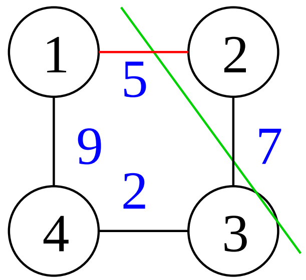
Selecting and Discarding MST Edges

The Cut Property

For any $S \subset V$ consider the cut edges

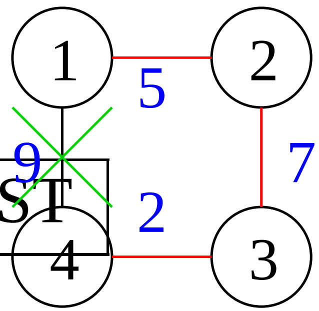
$C = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$

The lightest edge in a C can be used in an MST.



The Cycle Property

The heaviest edge on a cycleis not needed for an MST



The Jarník-Prim Algorithm [Jarník 1930, Prim 1957]

Idea: grow a tree

$T := \emptyset$

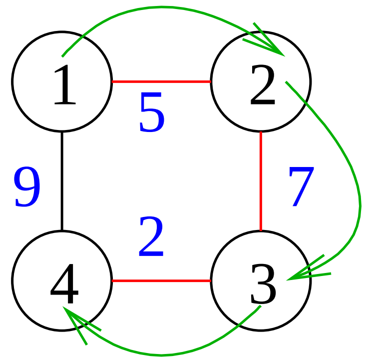
$S := \{s\}$ for arbitrary start node s

repeat $n-1$ times

find (u,v) fulfilling the cut property for S

$S := S \cup \{v\}$

$T := T \cup \{(u,v)\}$



Implementation Using Priority Queues

Function jpMST(V, E, w) : Set **of** Edge

dist$=[\infty,\ldots,\infty]$: **Array** $[1..n]$// dist$[v]$ is distance of v from the tree

pred : **Array of** Edge// pred$[v]$ is shortest edge between S and v

q : PriorityQueue **of** Node with dist$[\cdot]$ as priority

dist$[s]$:= 0; q.insert(s) for any $s \in V$

for i := 1 **to** $n-1$ **do** **do**

u := q.deleteMin() // new node for S

dist$[u]$:= 0

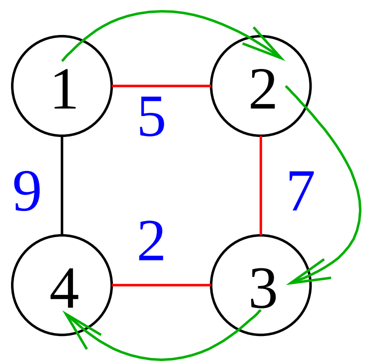
foreach $(u,v) \in E$ **do**

if $c((u,v)) <$ dist$[v]$ **then**

dist$[v]$:= $c((u,v))$; pred$[v]$:= (u,v)

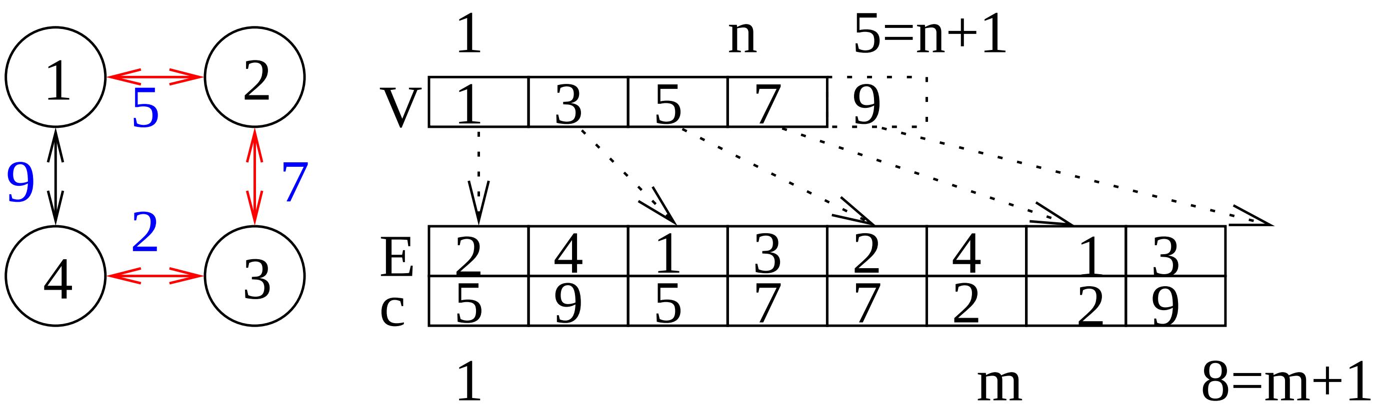
if $v \in q$ **then** q.decreaseKey(v) **else** q.insert(v)

return $\{$pred$[v] : v \in V \setminus \{s\}\}$



Graph Representation for Jarník-Prim

We need node $\longrightarrow$ incident edges



+ fast (cache efficient)

+ more compact than linked lists

– difficult to change

– Edges are stored twice

Analysis

- $O(m+n)$ time outside priority queue
- n deleteMin (time $O(n\log n)$)
- $O(m)$ decreaseKey (time $O(1)$ amortized)

$\rightsquigarrow$ $O(m+n\log n)$ using Fibonacci Heaps

practical implementation using simpler pairing heaps.

But analysis is still partly open!

Kruskal's Algorithm [1956]

$T := \emptyset$ // subforest of the MST

foreach $(u, v) \in E$ in ascending order of weight **do**

  if u and v are in different subtrees of T **then**

    $T := T \cup \{(u, v)\}$ // Join two subtrees

return T

The Union-Find Data Structure

Class UnionFind$(n : \mathbb{N})$ // Maintain a partition of $1..n$

parent$=[n+1,\ldots,n+1]$: **Array** $[1..n]$ **of** $1..n + \lceil \log n \rceil$

Function find$(i : 1..n) : 1..n$

if parent$[i] > n$ **then return** i

else $i' :=$ find(parent$[i]$)

parent$[i] := i'$

return i'

Procedure link$(i, j : 1..n)$

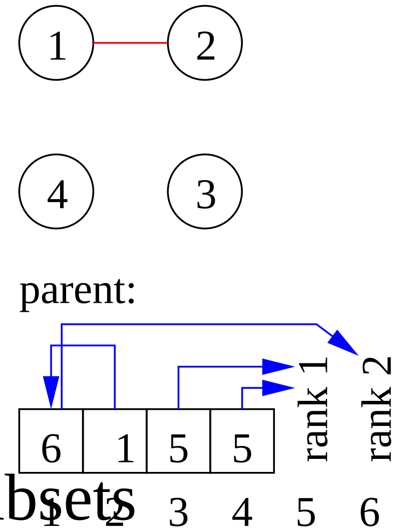
assert i and j are leaders of different subsets

if parent$[i] <$ parent$[j]$ **then** parent$[i] := j$

else if parent$[i] >$ parent$[j]$ **then** parent$[j] := i$

else parent$[j] := i$; parent$[i]$++ // next generation

Procedure union(i, j) **if** find$(i) \neq$ find(j) **then** link(find(i), find(j))



Kruskal Using Union Find

T : UnionFind(n)

sort E in ascending order of weight

kruskal(E)

Procedure kruskal(E)

 foreach $(u,v) \in E$ **do**

  $u' := T.\mathsf{find}(u)$

  $v' := T.\mathsf{find}(v)$

  if $u' \neq v'$ **then**

   output (u,v)

   $T.\mathsf{link}(u',v')$

Graph Representation for Kruskal

Just an edge sequence (array) !

+ very fast (cache efficient)
+ Edges are stored only once

$\rightsquigarrow$ more compact than adjacency array

Analysis

$\mathcal{O}(\mathrm{sort}(m) + m\alpha(m,n)) = \mathcal{O}(m \log m)$ where α is the inverse Ackermann function

Kruskal versus Jarník-Prim I

- Kruskal wins for very sparse graphs
- Prim seems to win for denser graphs
- Switching point is unclear
  - How is the input represented?
  - How many decreaseKeys are performed by JP? (average case: $n \log \frac{m}{n}$ [Noshita 85])
  - Experimental studies are quite old [**?**], use slow graph representation for both algs, and artificial inputs

Better Version For Dense Graphs ?

Procedure quickKruskal(E : Sequence **of** Edge)
  if $m \leq \beta n$ **then** kruskal(E) // for some constant β
  else
    pick a pivot $p \in E$
    $E_{\leq} := \langle e \in E : e \leq E \rangle$ // partitioning a la
    $E_{>} := \langle e \in E : e > E \rangle$ // quicksort
    quickKruskal($E_{\leq}$)
    $E'_{>} :=$ filter($E_{>}$)
    quickKruskal($E'_{>}$)

Function filter(E)
  make sure that leader[i] gives the leader of node i // $O(n)$!
  return $\langle (u,v) \in E : \text{leader}[u] \neq \text{leader}[v] \rangle$

8.1 Attempted Average-Case Analysis

Assume different random edge weights, arbitrary graphs

Assume pivot p has median weight

Let $T(m)$ denote the expected execution time for m edges

$m \leq \beta n$: $O(n \log n)$

Partitioning, Filtering: $O(m+n)$

$m > \beta n$: $T(m) = \Omega(m) +$T(m/2)+T(2n) [Chan 98]

Solves to $O\left(m + n\log(n) \cdot \log \frac{m}{n}\right) \leq O(m + n\log(n) \cdot \log\log n)$

Open Problem: I know of no graph family with

$T(n) = \omega(m + n\log(n))$

Kruskal versus Jarník-Prim II

Things are even less clear.

Kruskal may be better even for dense graphs

Experiments would be interesting.
Even for artificial graphs.

8.2 Filtering by Sampling Rather Than Sorting

R:= random sample of r edges from E

F :=MST(R) // Wlog assume that F spans V

L:= $\emptyset$ // "light edges" with respect to R

foreach $e \in E$ **do** // Filter

  C := the unique cycle in $\{e\} \cup F$

  if e is not heaviest in C **then**

    L:= $L \cup \{e\}$

return MST$((L \cup F))$

8.2.1 Analysis

[Chan 98, KKK 95]

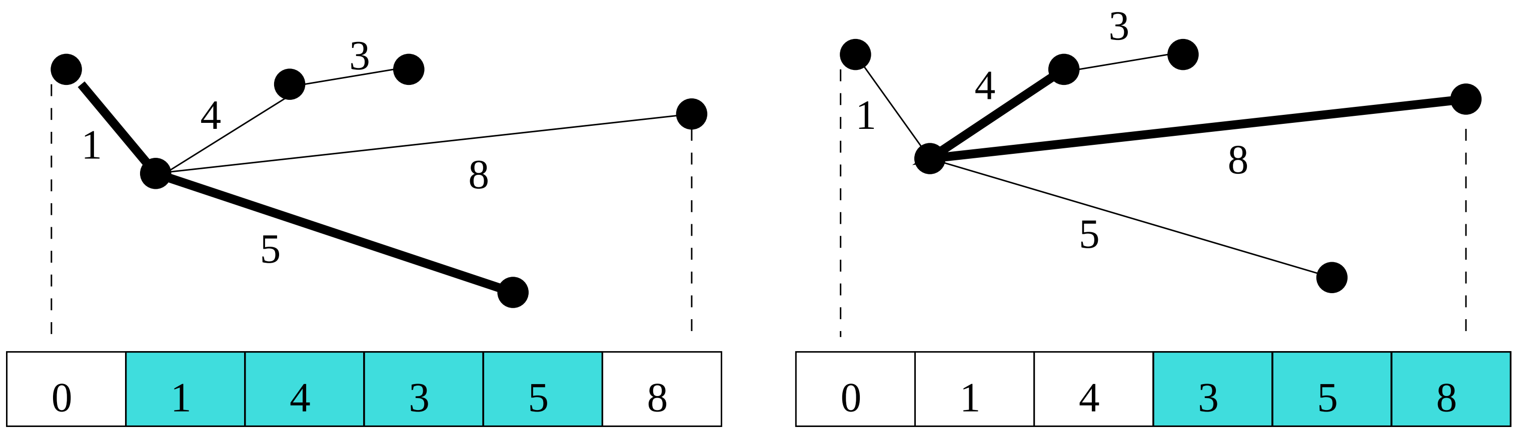
Observation: $e \in L$ only if $e \in \mathrm{MST}(R \cup \{e\})$.

(Otherwise e could replace some heavier edge in F).

Lemma 4. $\mathrm{E}[|L \cup F|] \leq \frac{mn}{r}$

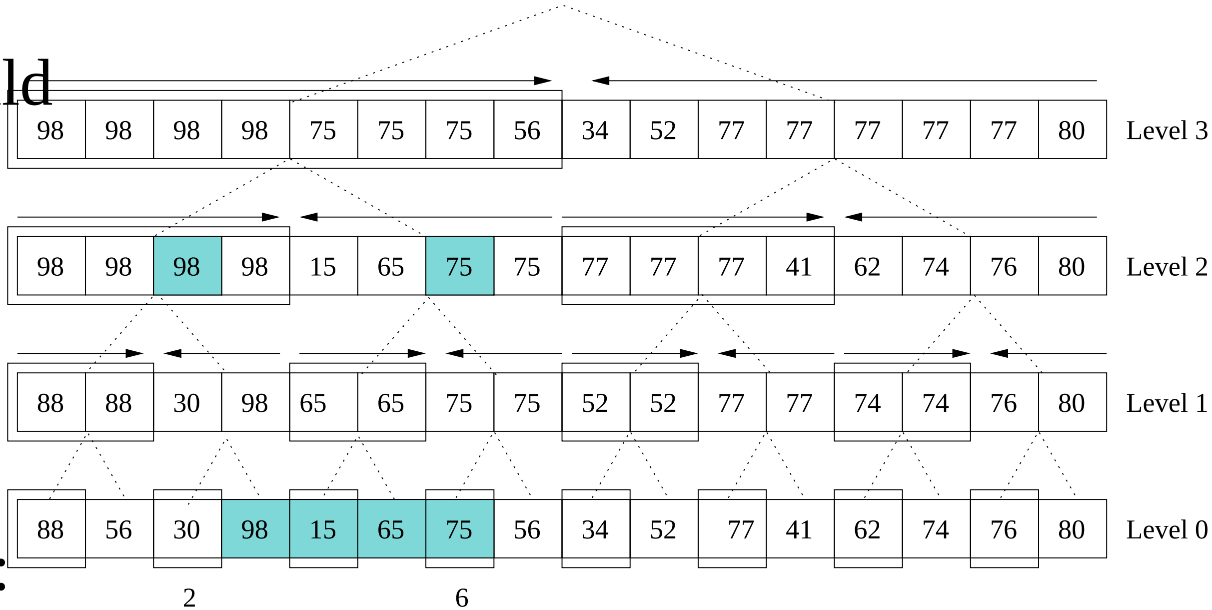
MST Verification by Interval Maxima

- Number the nodes by the order they were added to the MST by Prim's algorithm.
- $w_i =$ weight of the edge that inserted node i.
- Largest weight on path(u,v) $= \max\{w_j \| u < j \leq v\}$.



Interval Maxima

Preprocessing: build $n \log n$ size array PreSuf.



To find $\max a[i..j]$:

- Find the level of the LCA: $\ell = \lfloor \log_2(i \oplus j) \rfloor$.
- Return $\max(\mathrm{PreSuf}[\ell][i], \mathrm{PreSuf}[\ell][j])$.
- Example: $2 \oplus 6 = 010 \oplus 110 = 100$: $\ell = 2$

A Simple Filter Based Algorithm

Choose $r = \sqrt{mn}$.

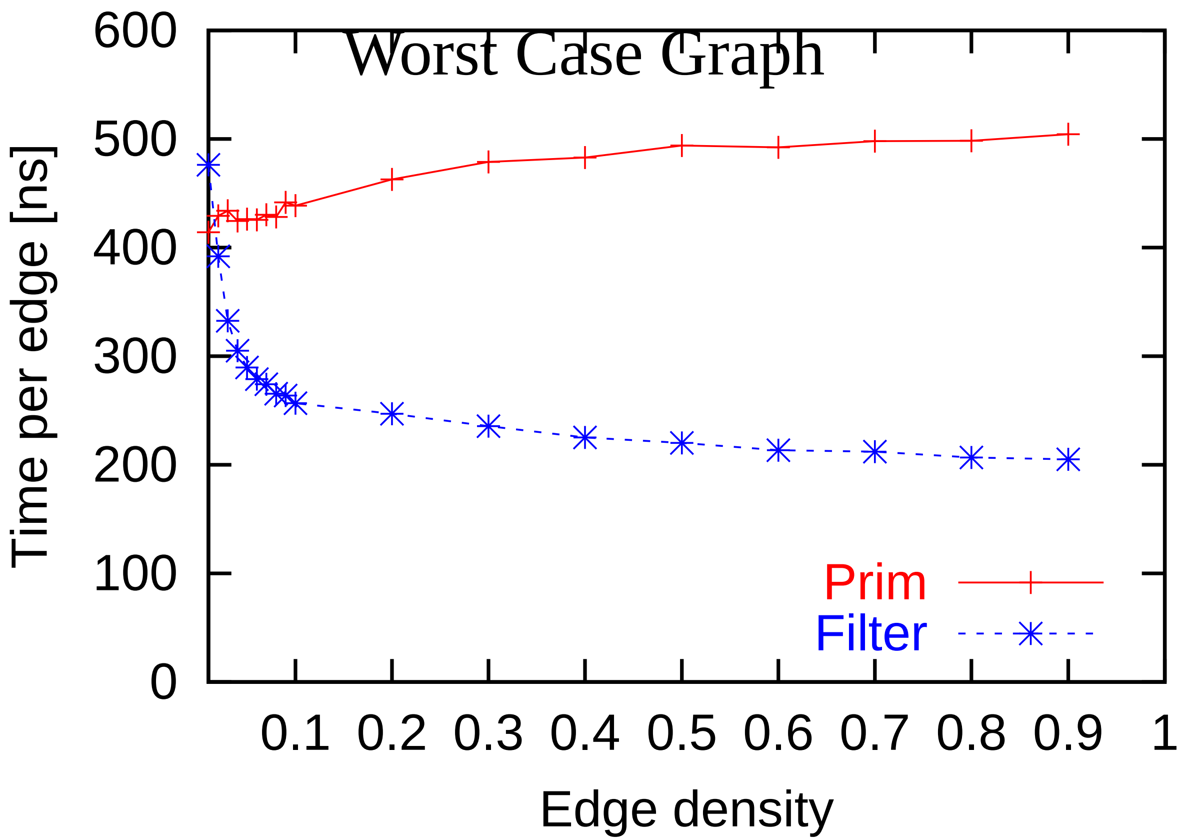
We get expected time

$$T_{\text{Prim}}(\sqrt{mn}) + O(n\log n + m) + T_{\text{Prim}}(\frac{mn}{\sqrt{mn}}) = O(n\log n + m)$$

The constant factor in front of the m is very small.

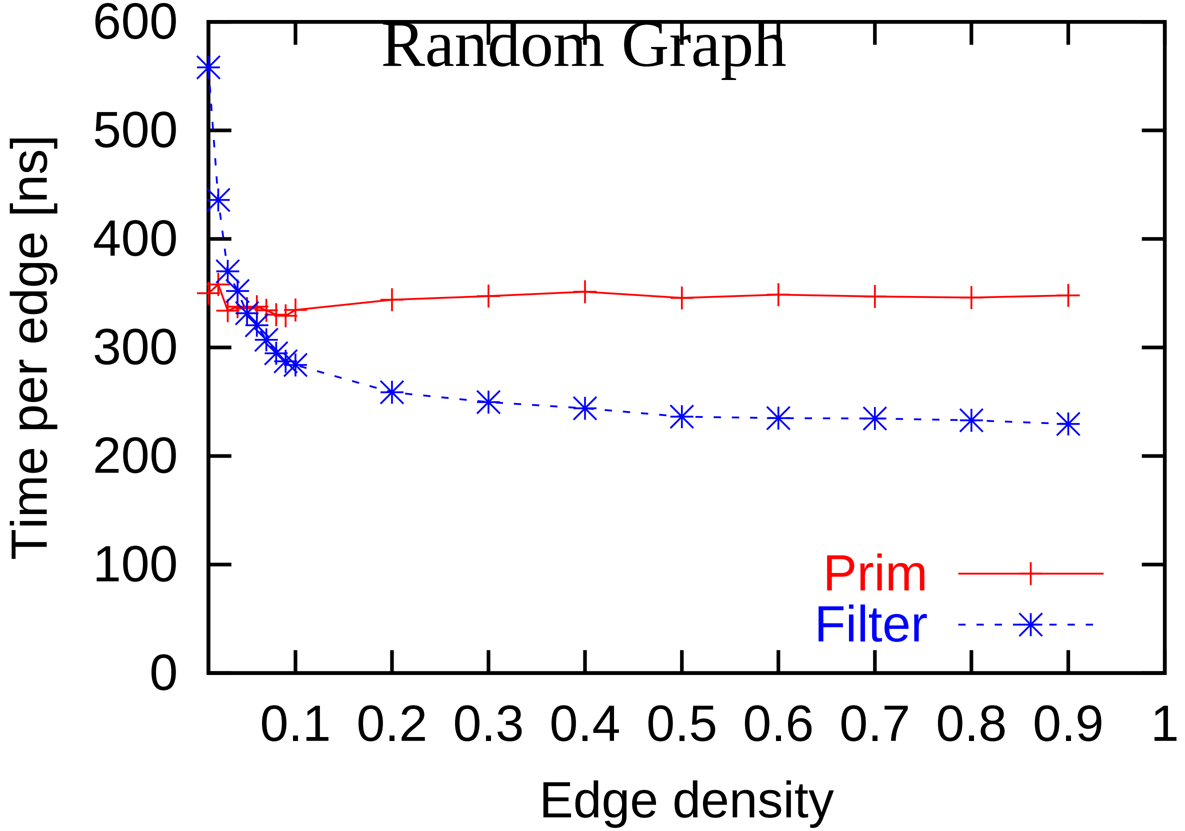
Results

10 000 nodes, SUN-Fire-15000, 900 MHz UltraSPARC-III+

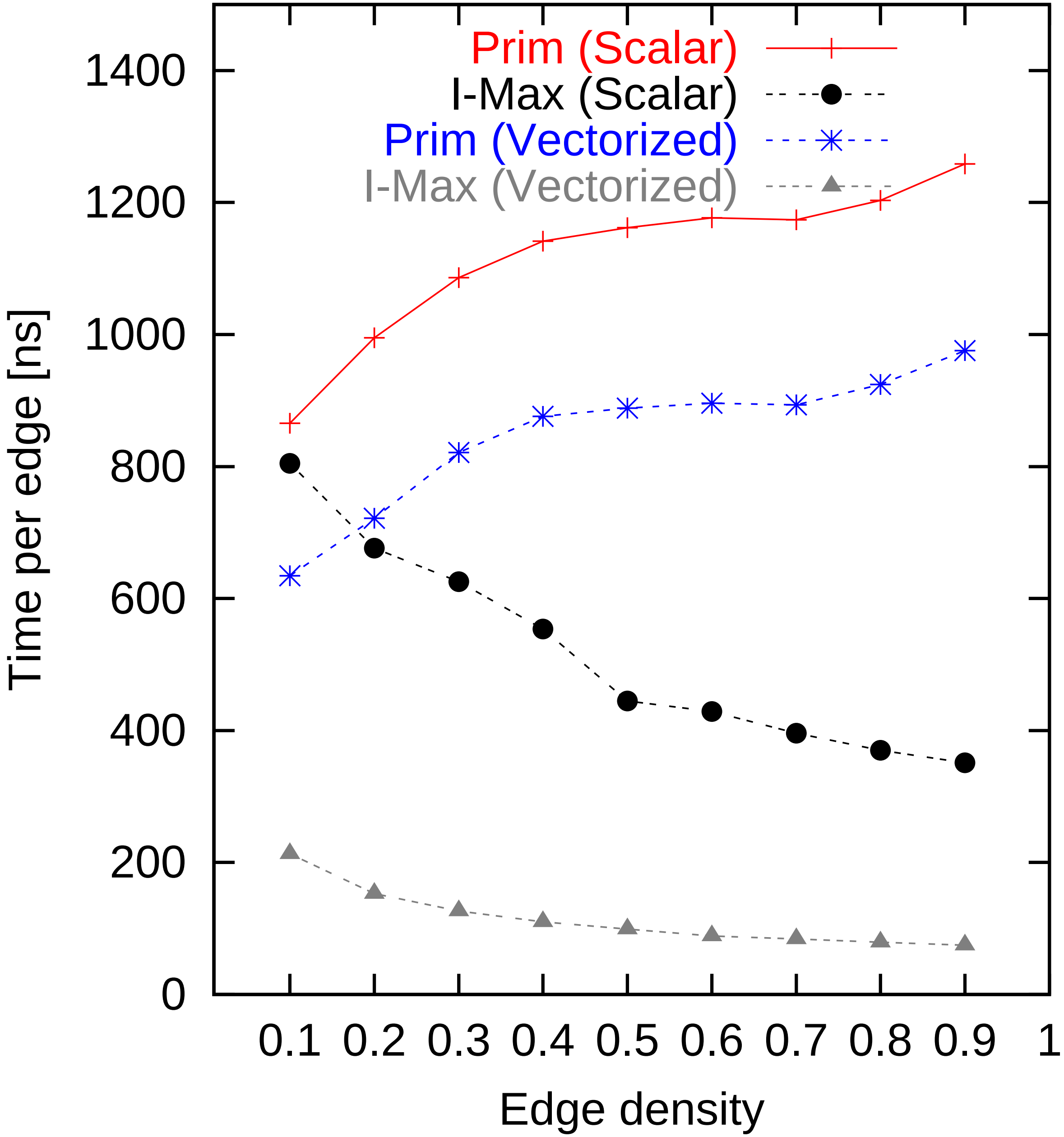


Results

10 000 nodes, SUN-Fire-15000, 900 MHz UltraSPARC-III+



Results



10 000 nodes,

NEC SX-5

Vector Machine

“worst case”

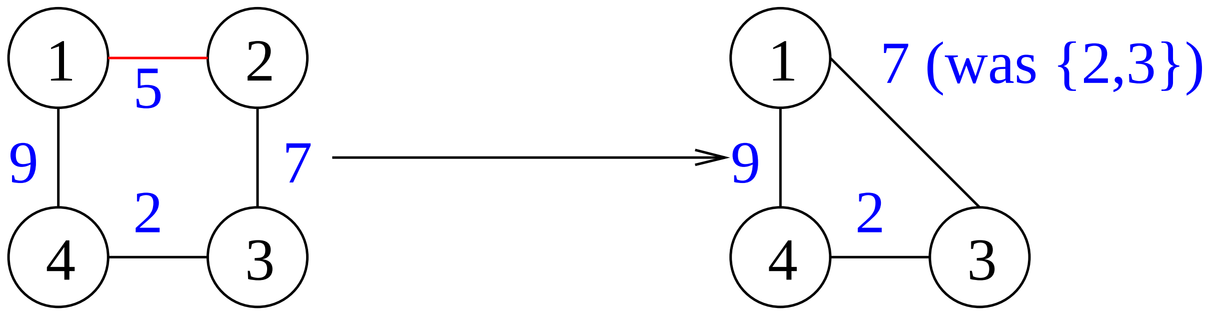
Edge Contraction

Let $\{u, v\}$ denote an MST edge.

Eliminate v:

forall $(w, v) \in E$ **do**

$E := E \setminus (w, v) \cup \{(w, u)\}$ // but remember orignal terminals

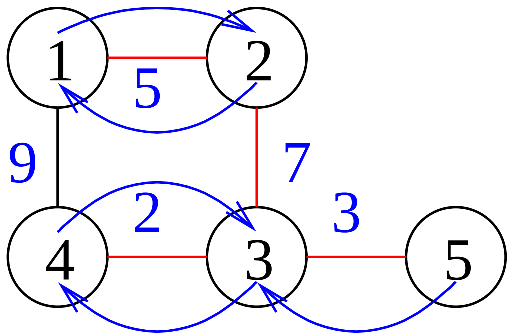


Boruvka's Node Reduction Algorithm

For each edge find the lightest incident edge.
Include them into the MST (cut property)
contract these edges,

Time $O(m)$

At least halves the number of remaining nodes



8.3 Simpler and Faster Node Reduction

for $i := n$ **downto** $n' + 1$ **do**

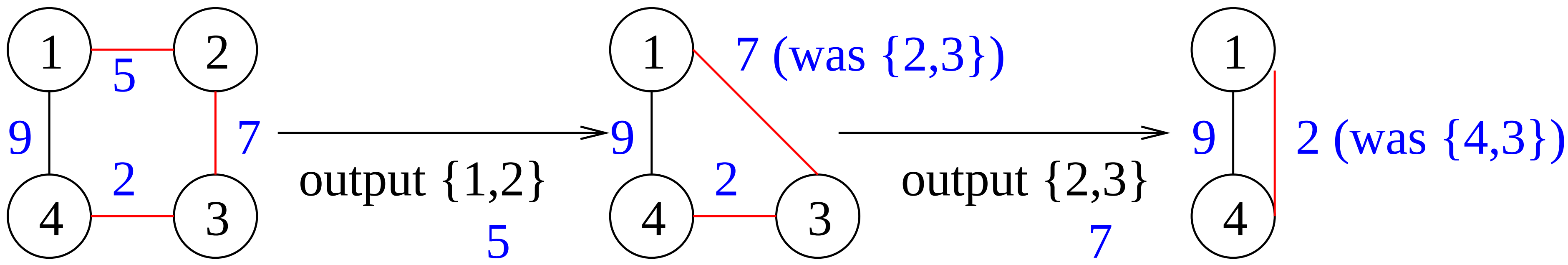
pick a random node v

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

contract (u, v)

$$\mathrm{E}[\text{degree}(v)] \le 2m/i$$

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



8.4 Randomized Linear Time Algorithm

1. Factor 8 node reduction ($3\times$ Boruvka or sweep algorithm) $\mathcal{O}(m+n)$.
2. $R \Leftarrow m/2$ random edges. $\mathcal{O}(m+n)$.
3. $F \Leftarrow MST(R)$ [Recursively].
4. Find light edges L (edge reduction). $\mathcal{O}(m+n)$
   $\mathrm{E}[|L|] \leq \frac{mn/8}{m/2} = n/4$.
5. $T \Leftarrow MST(L \cup F)$ [Recursively].

$T(n,m) \leq T(n/8, m/2) + T(n/8, n/4) + c(n+m)$

$T(n,m) \leq 2c(n+m)$ fulfills this recurrence.

8.5 External MSTs

Semiexternal Algorithms

Assume $n \le M - 2B$:

run Kruskal's algorithm using external sorting

Streaming MSTs

If M is yet a bit larger we can even do it with m/B I/Os:

$T := \emptyset$ // current approximation of MST

while there are any unprocessed edges **do**

    load any $\Theta(M)$ unprocessed edges E'

    $T := \mathsf{MST}(T \cup E')$ // for any internal MST alg.

Corollary: we can do it with linear expected internal work

Disadvantages to Kruskal:

Slower in practice

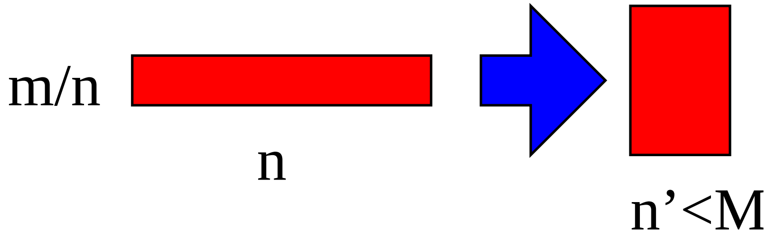
Smaller max. n

General External MST

while $n > M - 2B$ **do**

perform some node reduction

use semi-external Kruskal



Theory: $O(\text{sort}(m))$ expected I/Os by externalizing the linear time algorithm.

(i.e., node reduction + edge reduction)

External Implementation I: Sweeping

π : random permutation $V \rightarrow V$

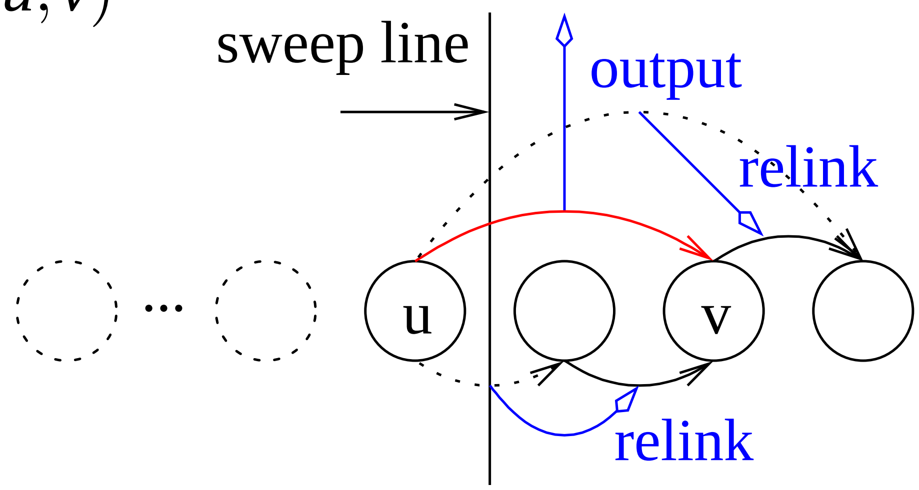
sort edges (u, v) by $\min(\pi(u), \pi(v))$

for $i := n$ **downto** $n' + 1$ **do**

    pick the node v with $\pi(v) = i$

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

    contract (u, v)



Problem: how to implement relinking?

Relinking Using Priority Queues

Q: priority queue // Order: max node, then min edge weight

foreach $(\{u,v\},c) \in E$ **do** Q.insert$((\{\pi(u),\pi(v)\},c,\{u,v\}))$

current := $n+1$

loop

  $(\{u,v\},c,\{u_0,v_0\})$:= Q.deleteMin()

  if current$\neq \max\{u,v\}$ **then**

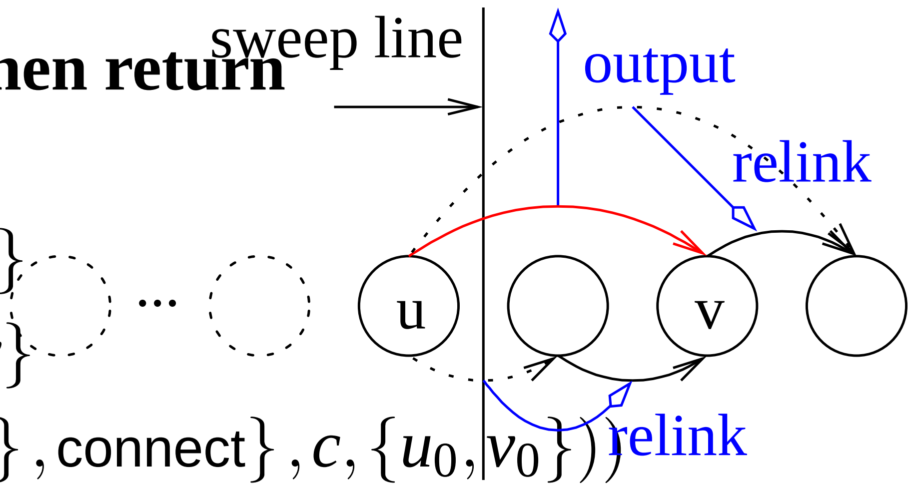
    if current$= M+1$ **then return**

    output $\{u_0,v_0\},c$

    current := $\max\{u,v\}$

    connect := $\min\{u,v\}$

  else Q.insert$((\{\min\{u,v\},\text{connect}\},c,\{u_0,v_0\}))$

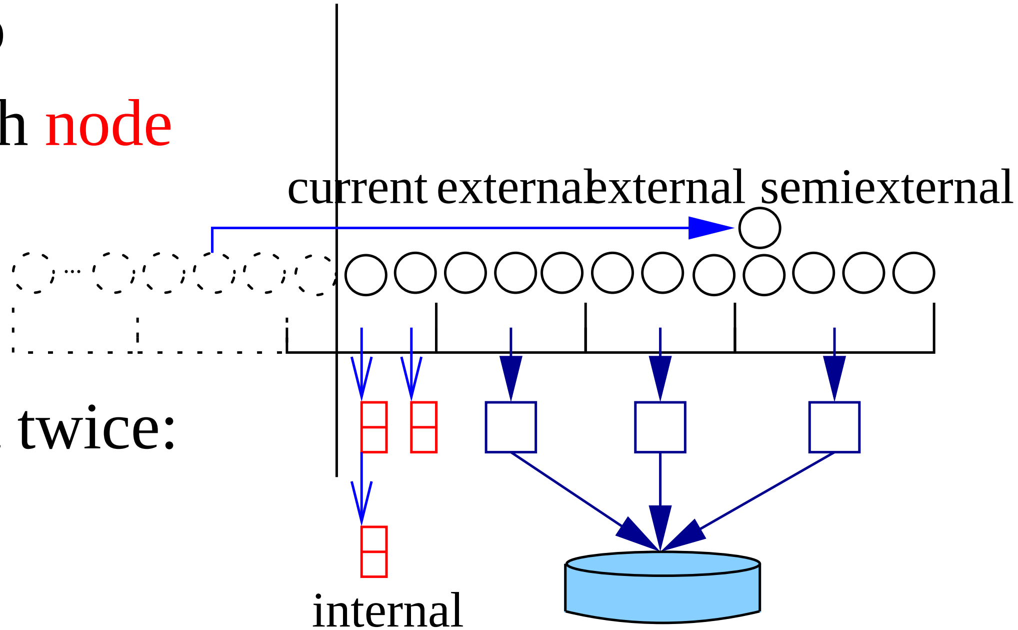


$\approx$ sort$(10m\ln\frac{n}{M})$ I/Os with opt. priority queues [Sanders 00]

Problem: Compute bound

Sweeping with linear internal work

- Assume $m = o\left(M^2/B\right)$
- $k = \Theta(M/B)$ external buckets with n/k nodes each
- M nodes for last "semiexternal" bucket
- split current bucket into internal buckets for each node



Sweeping:

Scan current internal bucket twice:

1. Find minimum
2. Relink

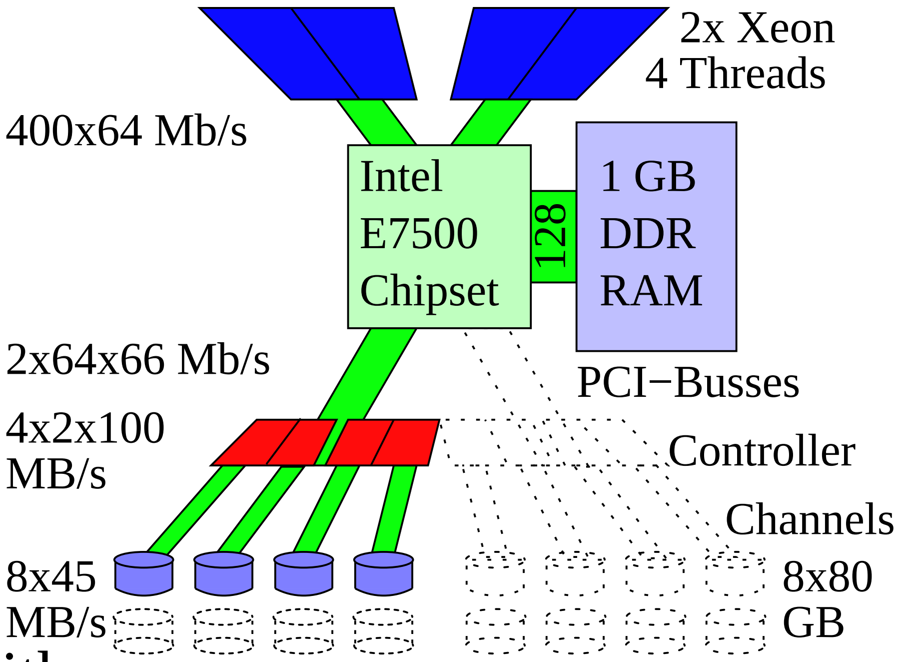
New external bucket: scan and put in internal buckets

Large degree nodes: move to semiexternal bucket

Experiments

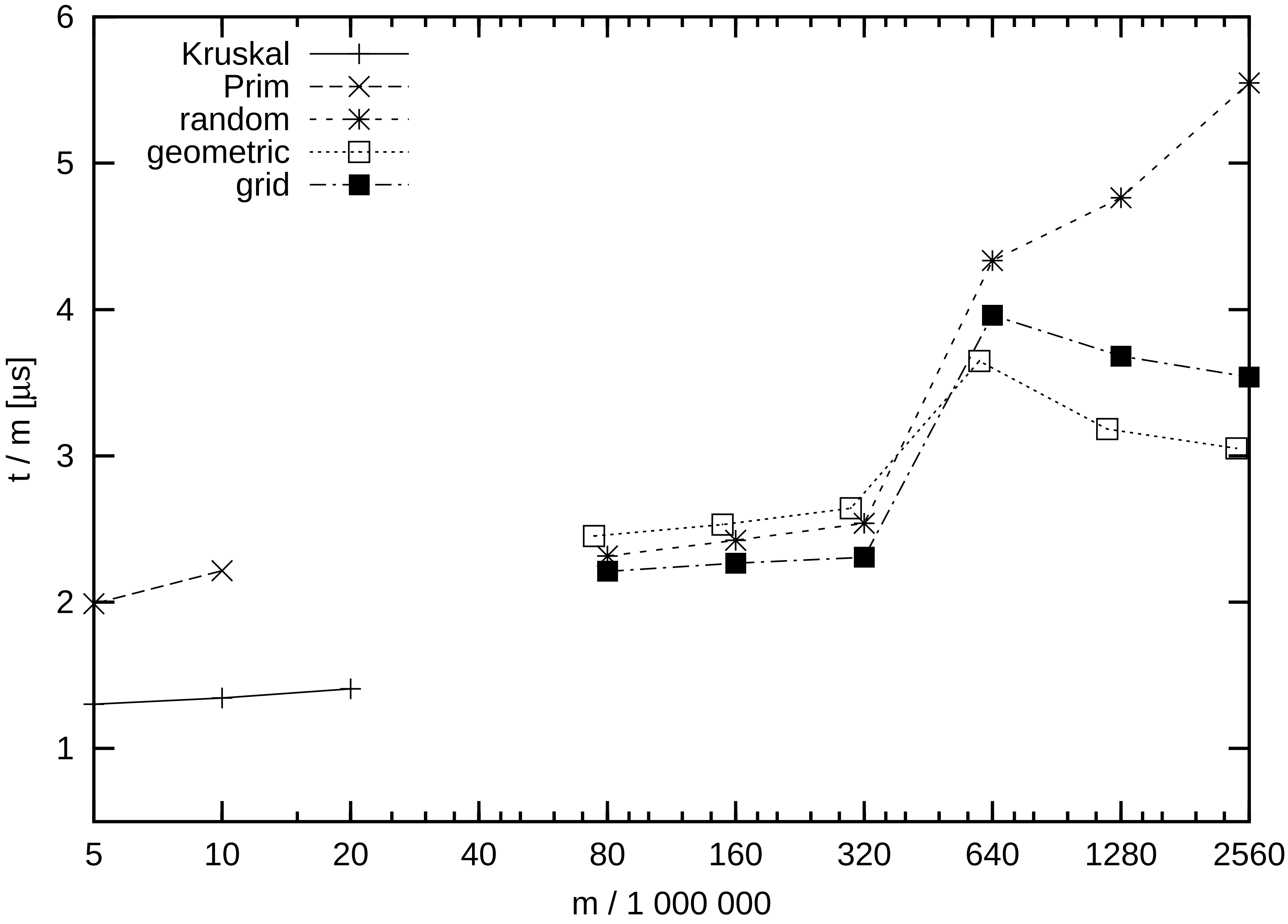
Instances from “classical” MST study [Moret Shapiro 1994]

- sparse random graphs
- random geometric graphs
- grids
  $O(\mathrm{sort}(m))$ I/Os
  for planar graphs by
  removing parallel edges!



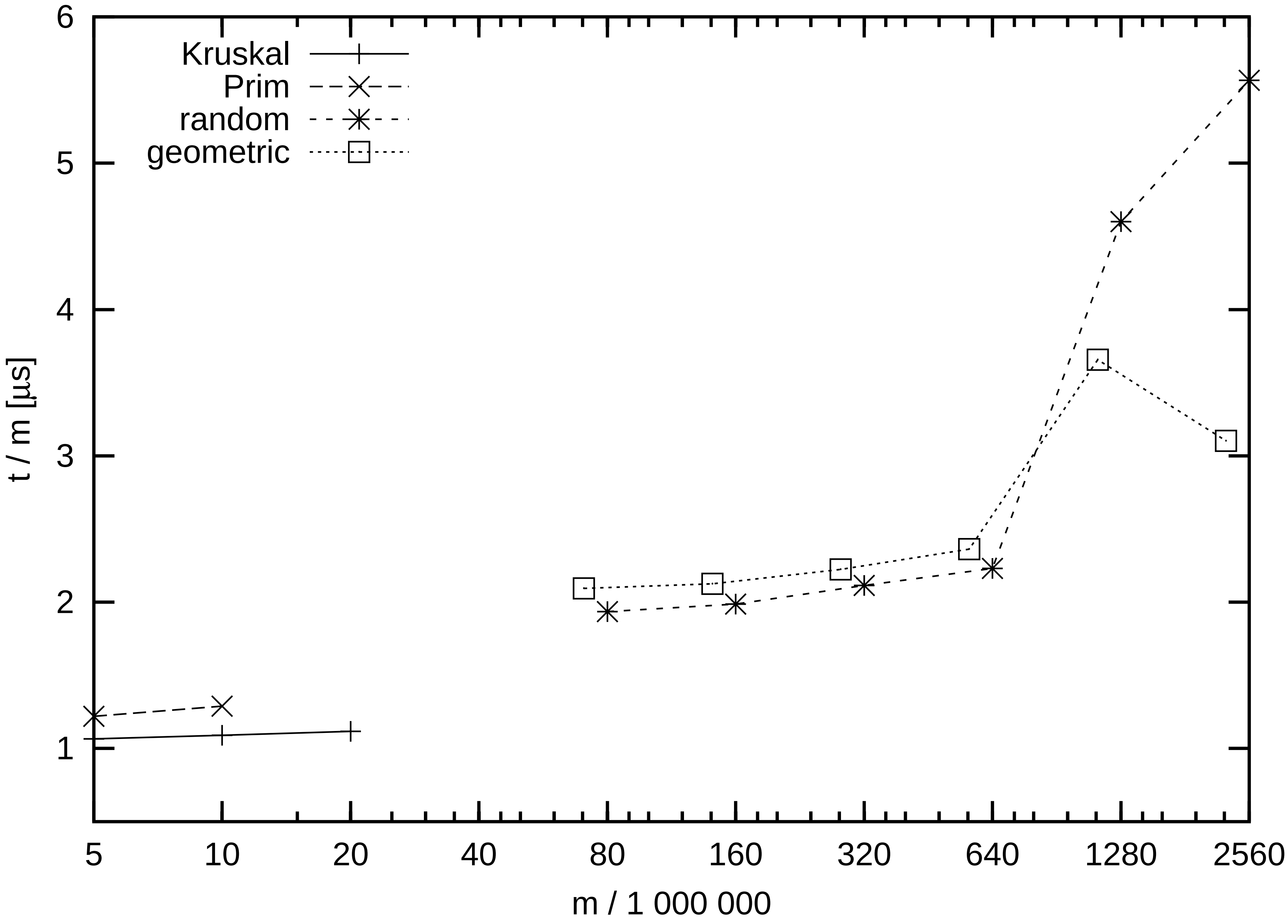
Other instances are rather dense or designed to fool specific algorithms.

$m \approx 2n$



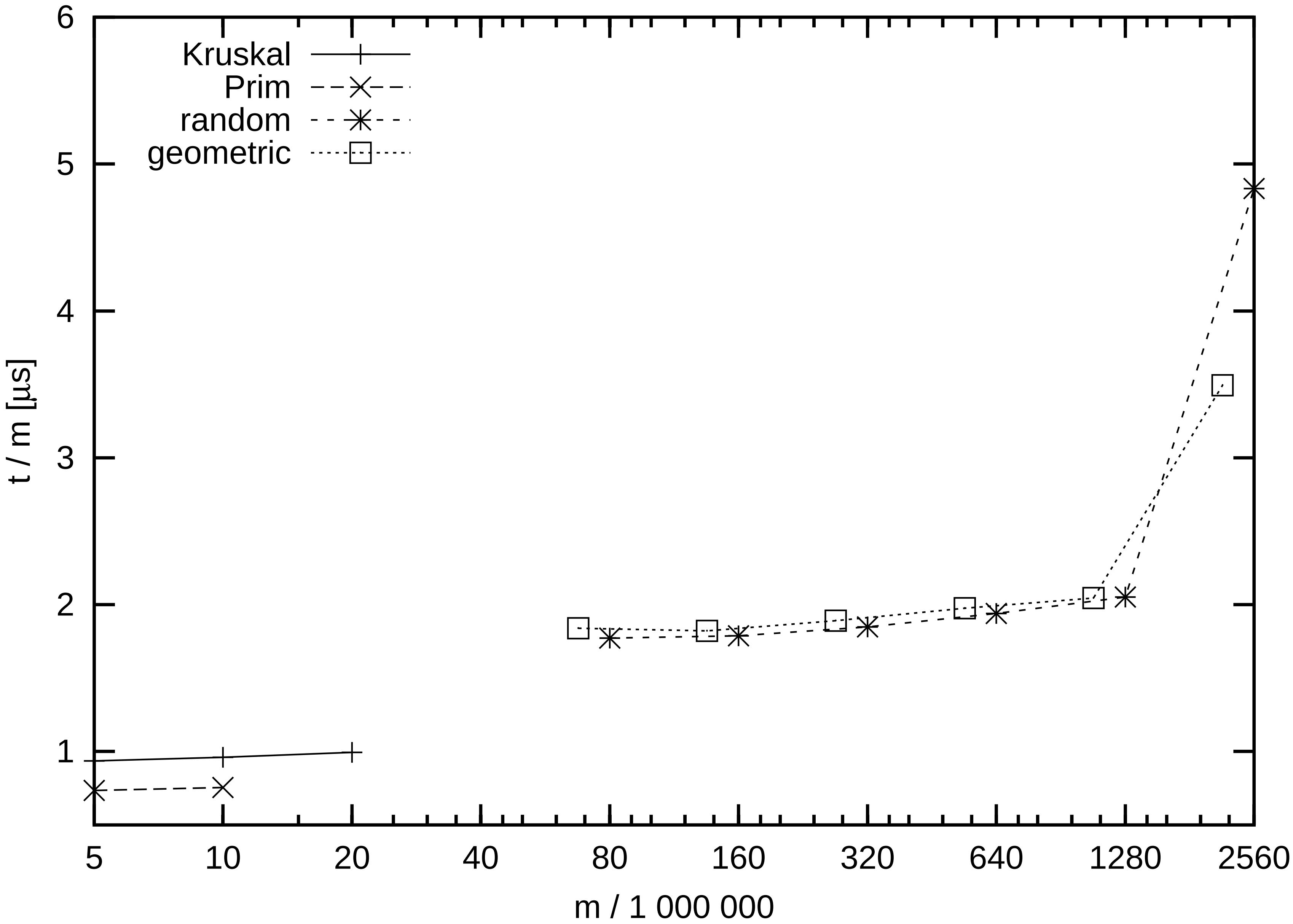
Kruskal
Prim
random
geometric
grid
6
5
4
3
2
1
t / m [µs]
5
10
20
40
80
160
320
640
1280
2560
m / 1 000 000

$m \approx 4n$



Kruskal
Prim
random
geometric
t / m [µs]
m / 1 000 000
6
5
4
3
2
1
5
10
20
40
80
160
320
640
1280
2560

$m \approx 8n$


6
5
4
3
2
1
Kruskal
Prim
random
geometric
t / m [µs]
5
10
20
40
80
160
320
640
1280
2560
m / 1 000 000

MST Summary

- Edge reduction helps for very dense, “hard” graphs
- A fast and simple node reduction algorithm
  $\rightsquigarrow$ $4\times$ less I/Os than previous algorithms
- Refined semiexternal MST, use as base case
- Simple pseudo random permutations (no I/Os)
- A fast implementation
- Experiments with huge graphs (up to $n = 4 \cdot 10^9$ nodes)

External MST is feasible

Open Problems

- New experiments for (improved) Kruskal versus Jarník-Prim
- Realistic (huge) inputs
- Parallel external algorithms
- Implementations for other graph problems

Conclusions

- Even fundamental, “simple” algorithmic problems still raise interesting questions
- Implementation and experiments are important and were neglected by parts of the algorithms community
- Theory an (at least) equally important, essential component of the algorithm design process

More Algorithm Engineering on Graphs

- Count triangles in very large graphs. Interesting as a measure of clusteredness. (Cooperation with AG Wagner)
- External BFS (Master thesis Deepak Ajwani)
- Maximum flows: Is the theoretical best algorithm any good? (Jein)
- Approximate max. weighted matching (Studienarbeit Jens Maue)

Maximal Flows

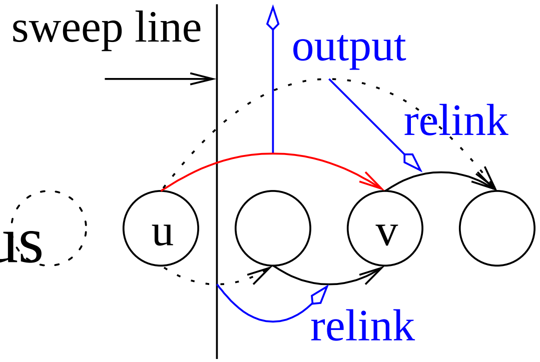
Theory: $O\left(m\Lambda\log(n^2/m)\log U\right)$ binary blocking flow-algorithm mit $\Lambda = \min\{m^{1/2}, n^{2/3}\}$ [Goldberg-Rao-97].

Problem: best case $\approx$ worst case

[Hagerup Sanders Träff WAE 98]:

- Implementable generalization
- best case $\ll$ worst case
- best algorithms for some “difficult” instances

Ergebnis



- Einfach extern implementierbar
- $n' = M \rightsquigarrow$ semiexterner Kruskal Algorithmus
- Insgesamt $O\left(\mathrm{sort}(m \ln \frac{n}{m})\right)$ erwartete I/Os
- Für realistische Eingaben mindestens $4\times$ bisher als bisher bekannte Algorithmen
- Implementierung in `<stxxl>` mit bis zu 96 GByte gro/3en Graphen läuft „über Nacht“

Presenting Data from Experiments in Algorithmics

Restrictions

- black and white $\leadsto$ easy and cheap printing
- 2D (stay tuned)
- no animation
- no realism desired

Not here

- ensuring reproducibility
- describing the setup
- finding sources of measurement errors
- reducing measuremnt errors (averaging, median,unloaded machine. . .)
- measurements in the creative phase of experimental algorithmics.

The Starting Point

- (Several) Algorithm(s)
- A few quantities to be measured: time, space, solution quality, comparisons, cache faults,... There may also be measurement errors.
- An unlimited number of potential inputs. $\rightsquigarrow$ condense to a few characteristic ones (size, $|V|$, $|E|$, ... or problem instances from applications)

Usually there is not a lack but an abundance of data $\neq$ many other sciences

The Process

Waterfall model?

1. Design
2. Measurement
3. Interpretation

Perhaps the paper should at least look like that.

The Process

- Eventually stop asking questions (Advisors/Referees listen !)
- build measurement tools
- automate (re)measurements
- Choice of Experments driven by risk and opportunity
- Distinguish mode

explorative: many different parameter settings, interactive, short turnaround times

consolidating: many large instances, standardized measurement conditions, batch mode, many machines

Of Risks and Opportunities

Example: Hypothesis $=$ my algorithm is the best

big risk: untried main competitor

small risk: tuning of a subroutine that takes 20 % of the time.

big opportunity: use algorithm for a new application

$\leadsto$ new input instances

Basic Principles

- Minimize nondata ink
  (form follows function, not a beauty contest,...)
- Letter size ≈ surrounding text
- Avoid clutter and overwhelming complexity
- Avoid boredom (too little data per m^2.
- Make the conclusions evident

Tables

+ easy
− easy $\rightsquigarrow$ overuse
+ accurate values ($\neq$ 3D)
+ more compact than bar chart
+ good for unrelated instances (e.g. solution quality)
− boring
− no visual processing

rule of thumb that "tables usually outperform a graph for small data sets of 20 numbers or less" [Tufte 83]

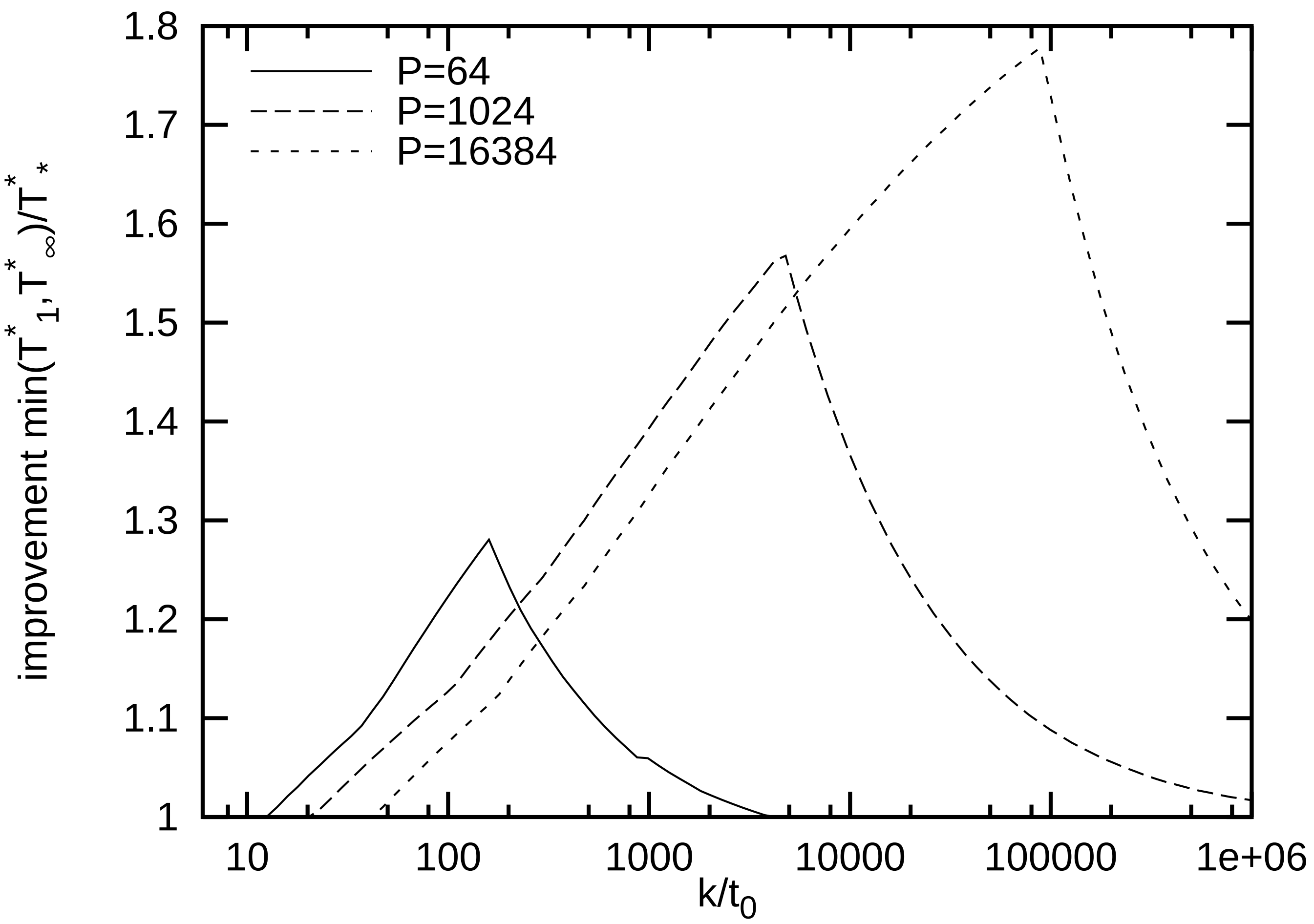
Curves in main paper, tables in appendix?

2D Figures

default: $x =$ input size, $y = f(\text{execution time})$

x Axis

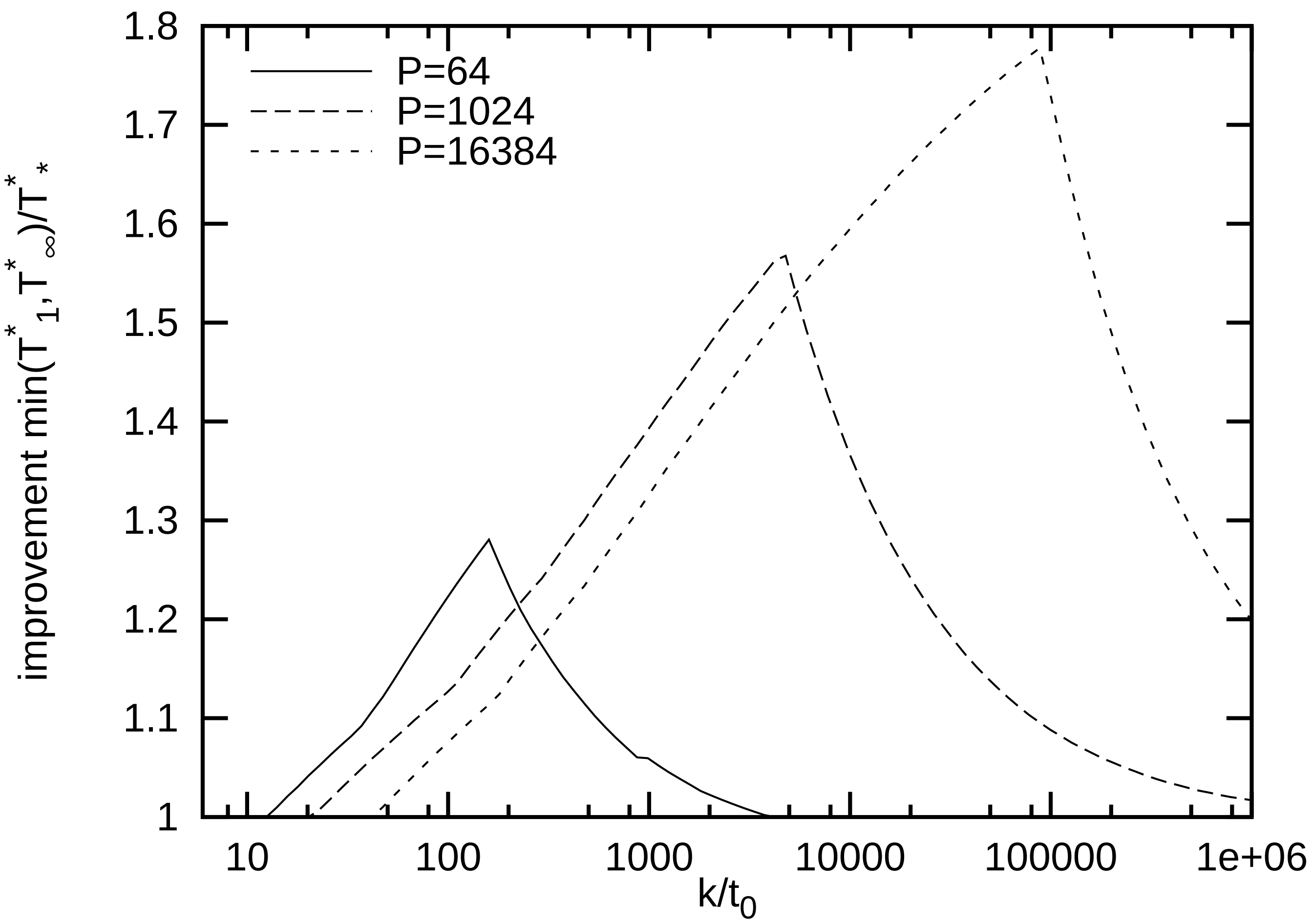
Choose unit to eliminate a parameter?



length k fractional tree broadcasting. latency $t_0 + k$

x Axis

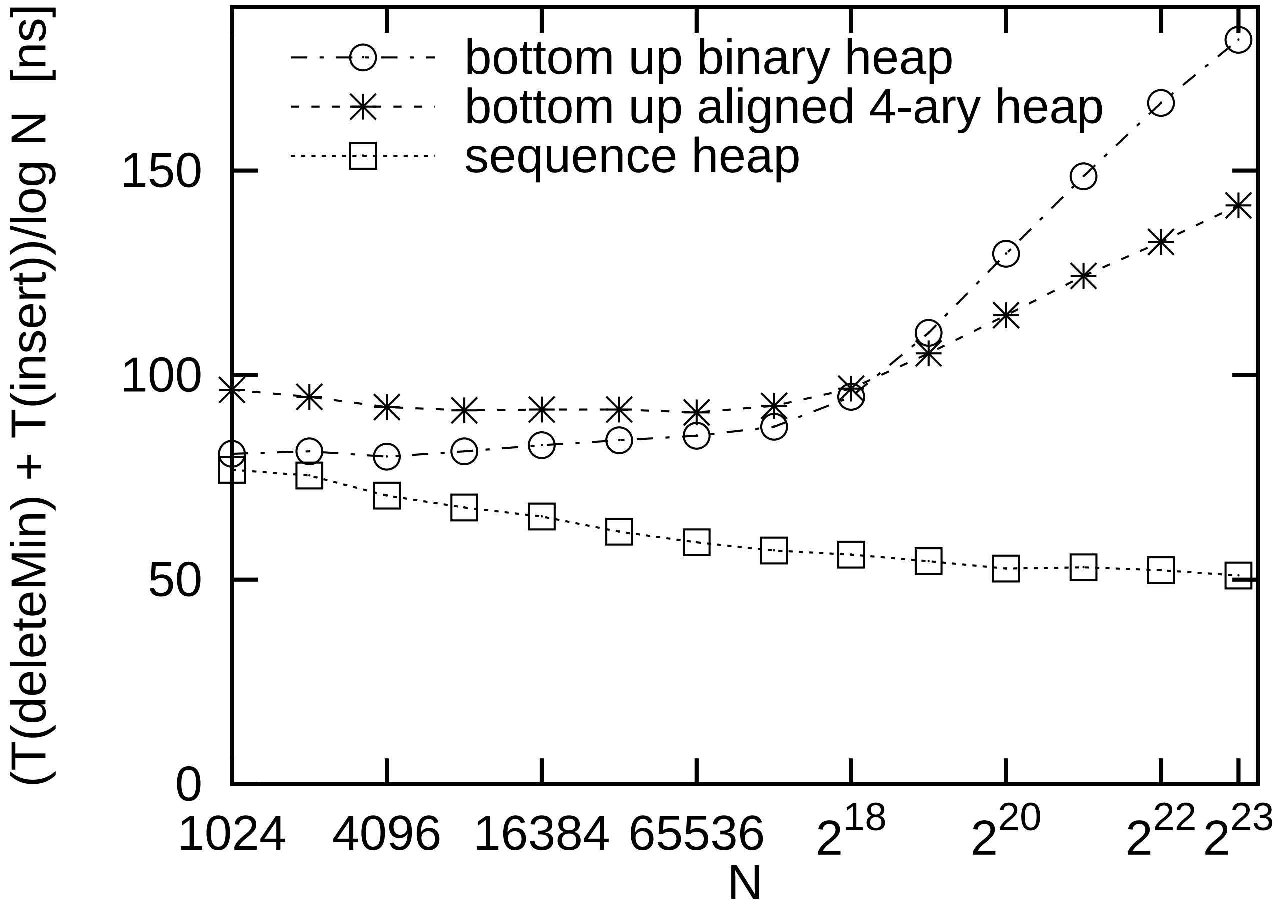
logarithmic scale?



yes if *x* range is wide

x Axis

logarithmic scale, powers of two (or $\sqrt{2}$)



with tic marks, (plus a few small ones)

gnuplot

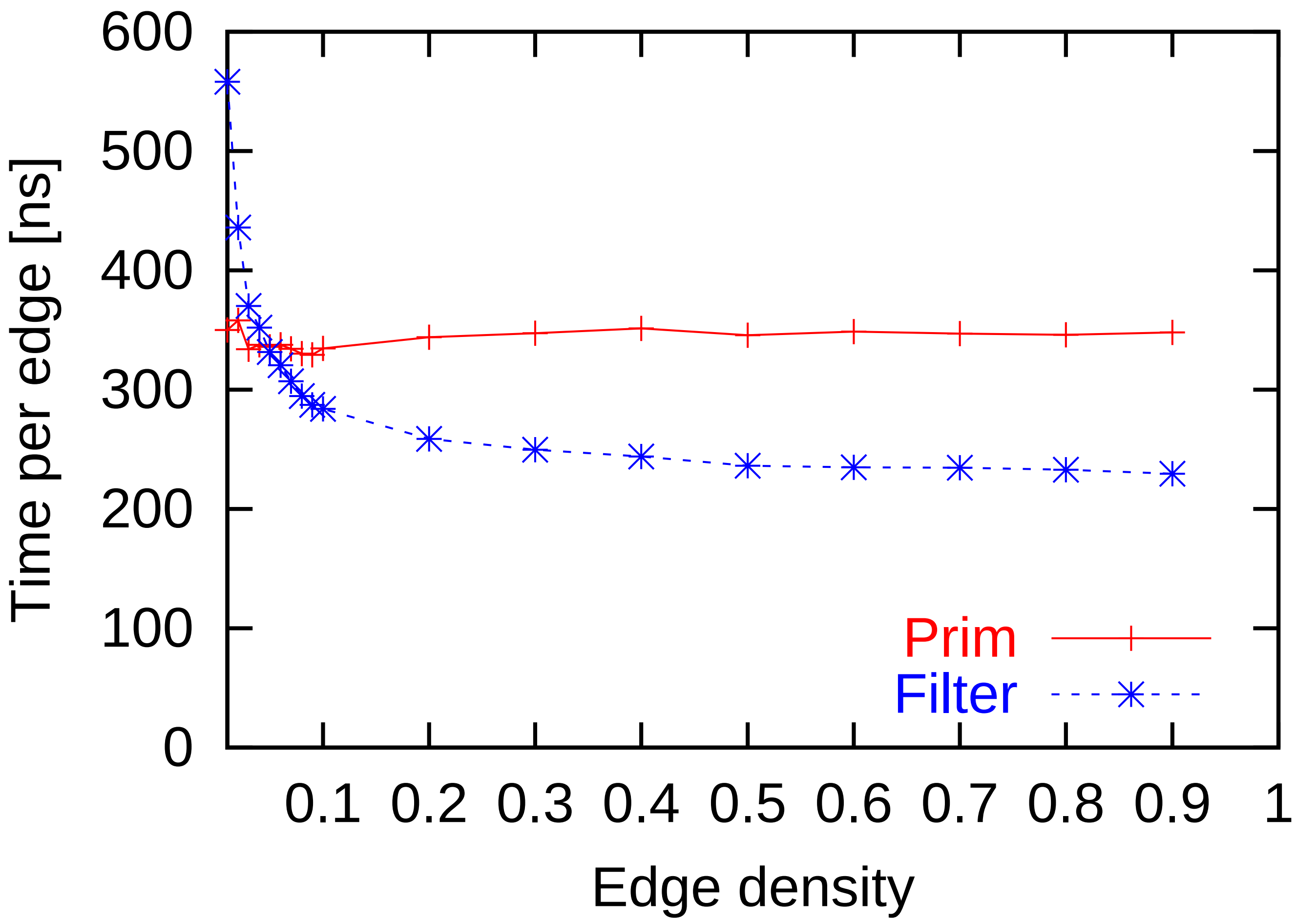
```
set xlabel "N"
set ylabel "(time per operation)/log N  [ns]"
set xtics (256, 1024, 4096, 16384, 65536, "2^{18}" 262144,
set size 0.66, 0.33
set logscale x 2
set data style linespoints
set key left
set terminal postscript portrait enhanced 10
set output "r10000timenew.eps"
plot [1024:10000000][0:220]\
 "h2r10000new.log" using 1:3 title "bottom up binary heap"
 "h4r10000new.log" using 1:3 title "bottom up aligned 4-ar
 "knr10000new.log" using 1:3 title "sequence heap" with li
```

Data File

```
256 703.125 87.8906
512 729.167 81.0185
1024 768.229 76.8229
2048 830.078 75.4616
4096 846.354 70.5295
8192 878.906 67.6082
16384 915.527 65.3948
32768 925.7 61.7133
65536 946.045 59.1278
131072 971.476 57.1457
262144 1009.62 56.0902
524288 1035.69 54.51
1048576 1055.08 52.7541
2097152 1113.73 53.0349
4194304 1150.29 52.2859
8388608 1172.62 50.9836
```

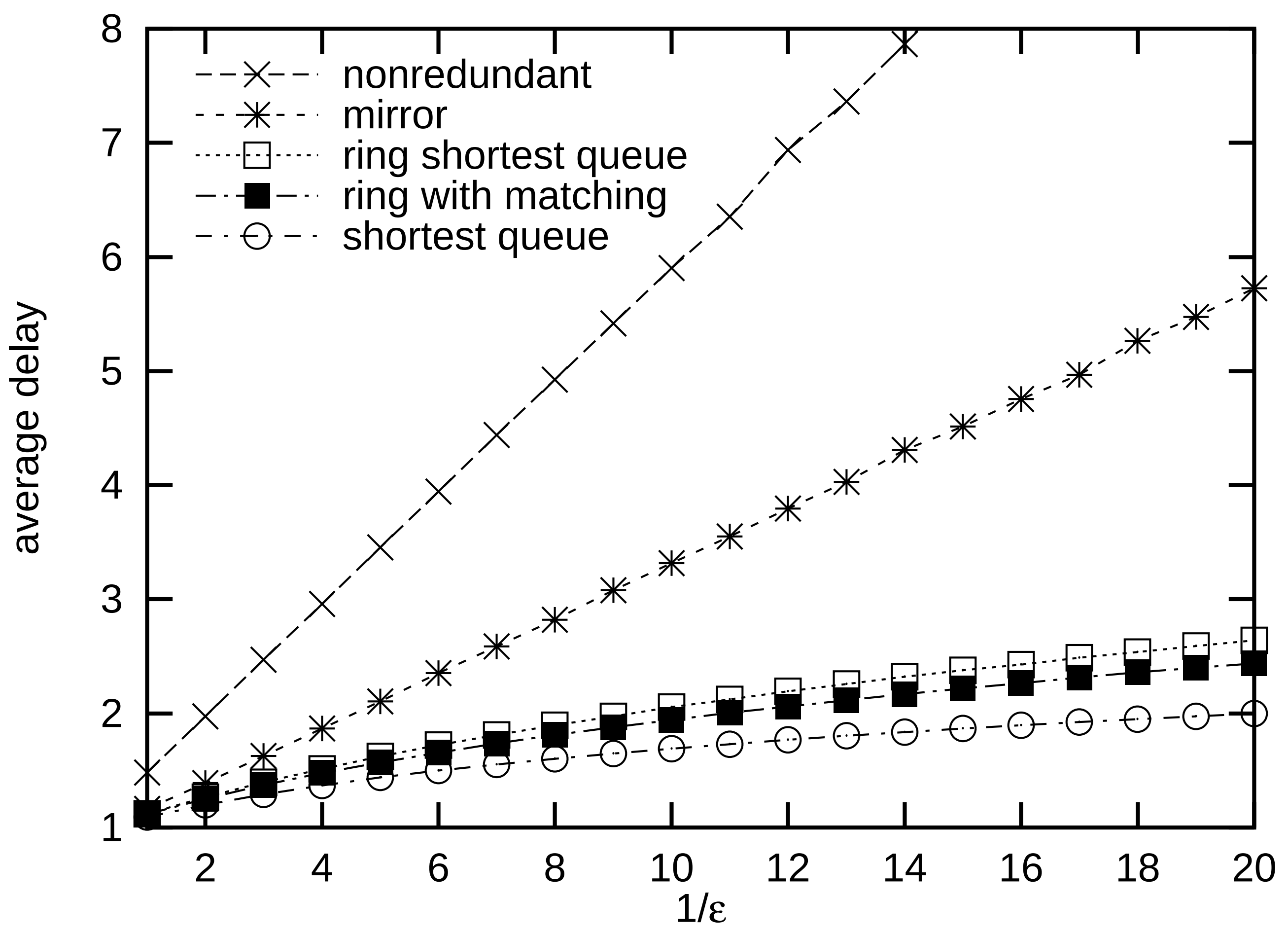
x Axis

linear scale for ratios or small ranges (#processor,…)



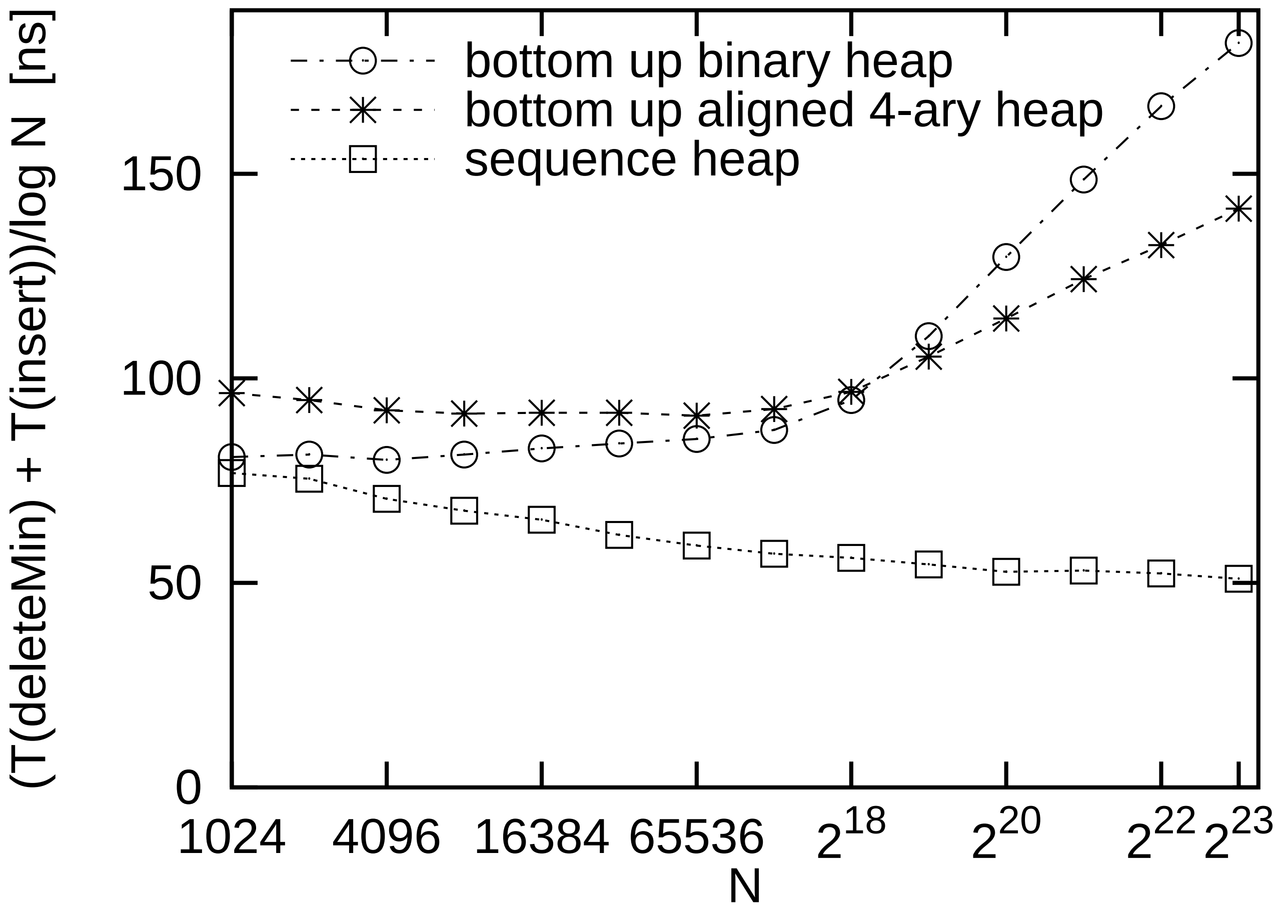
x Axis

An exotic scale: arrival rate $1-\varepsilon$ of saturation point



y Axis

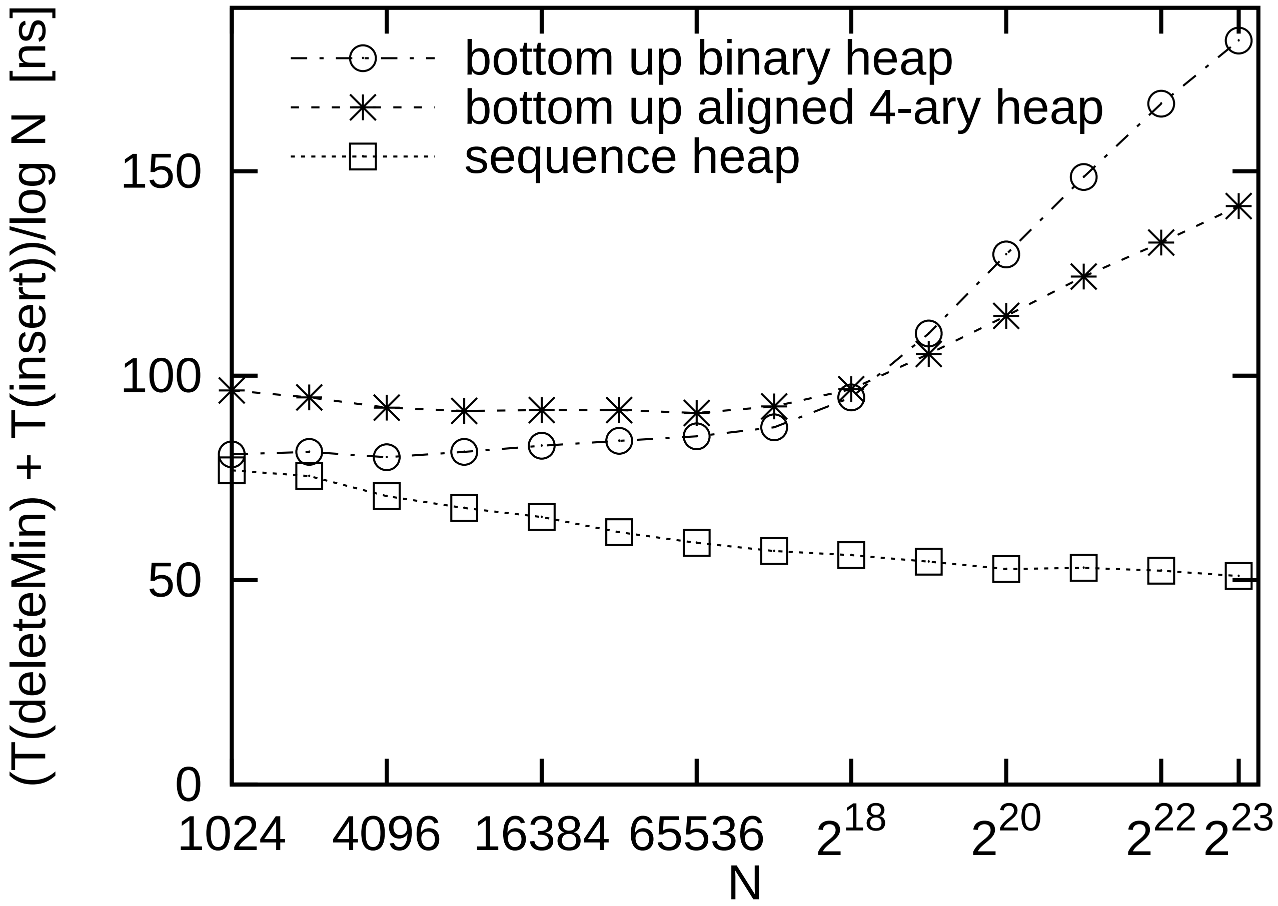
Avoid log scale ! scale such that theory gives ≈ horizontal lines



but give easy interpretation of the scaling function

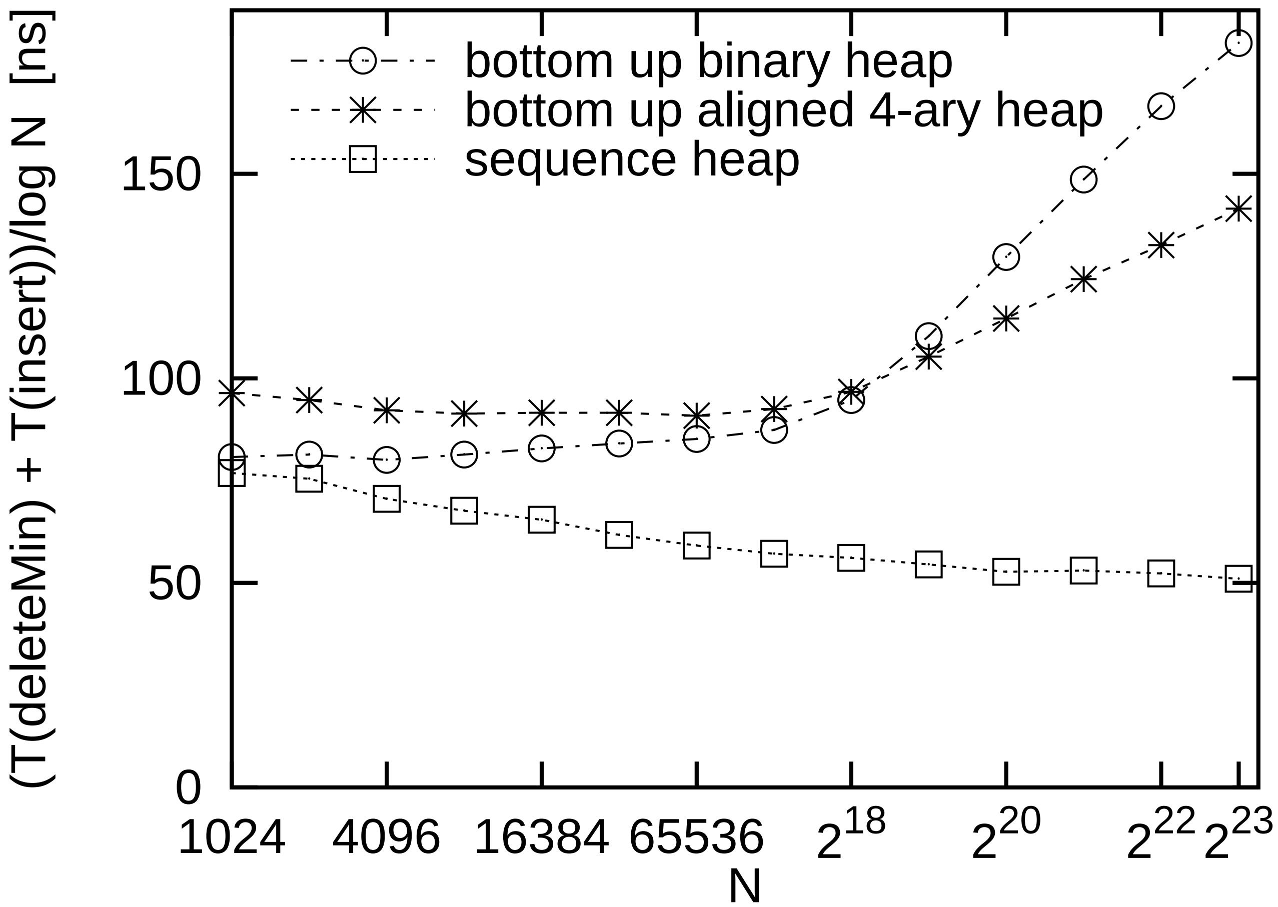
y Axis

give units



y Axis

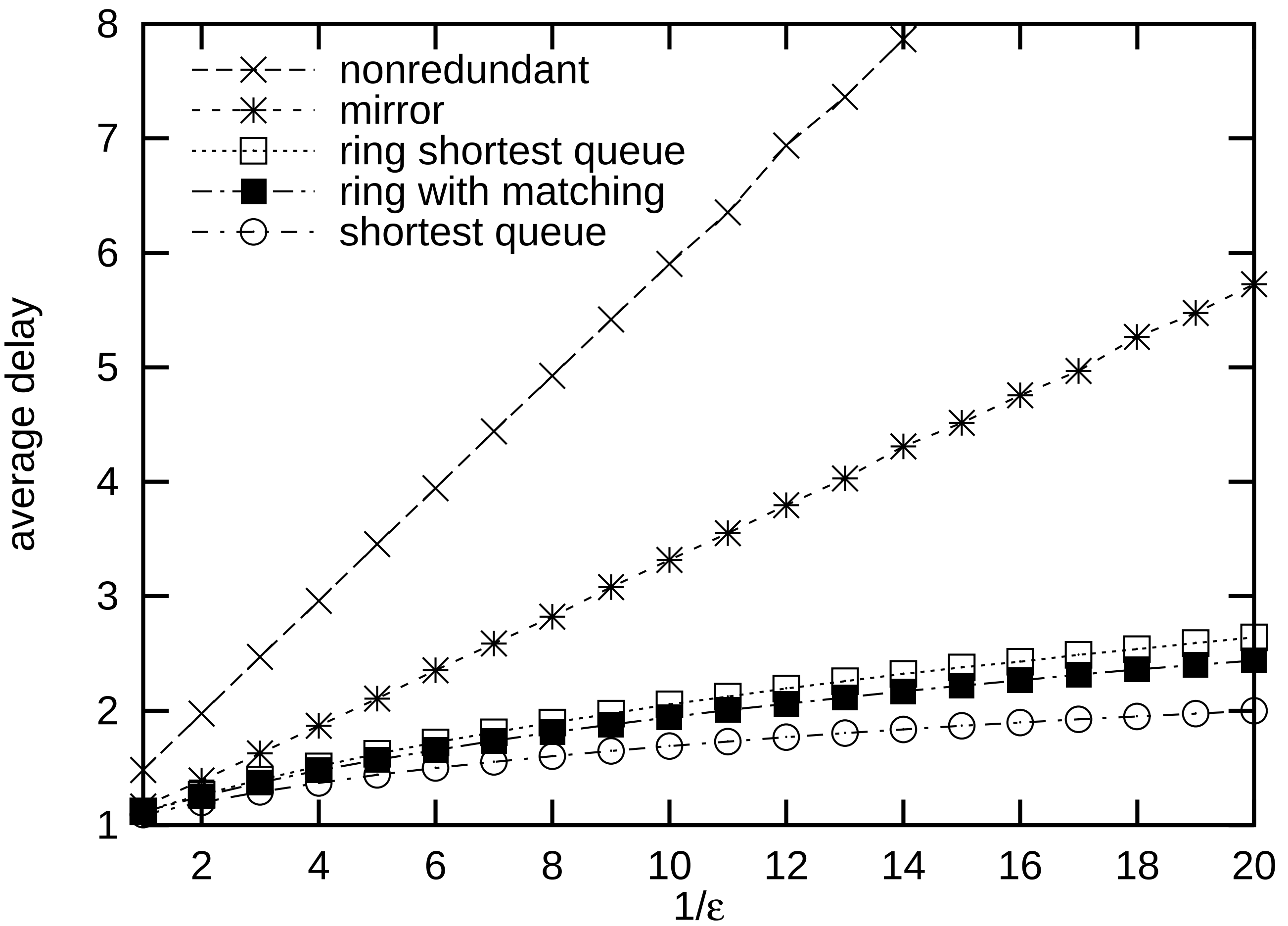
start from 0 if this does not waste too much space



you may assume readers to be out of Kindergarten

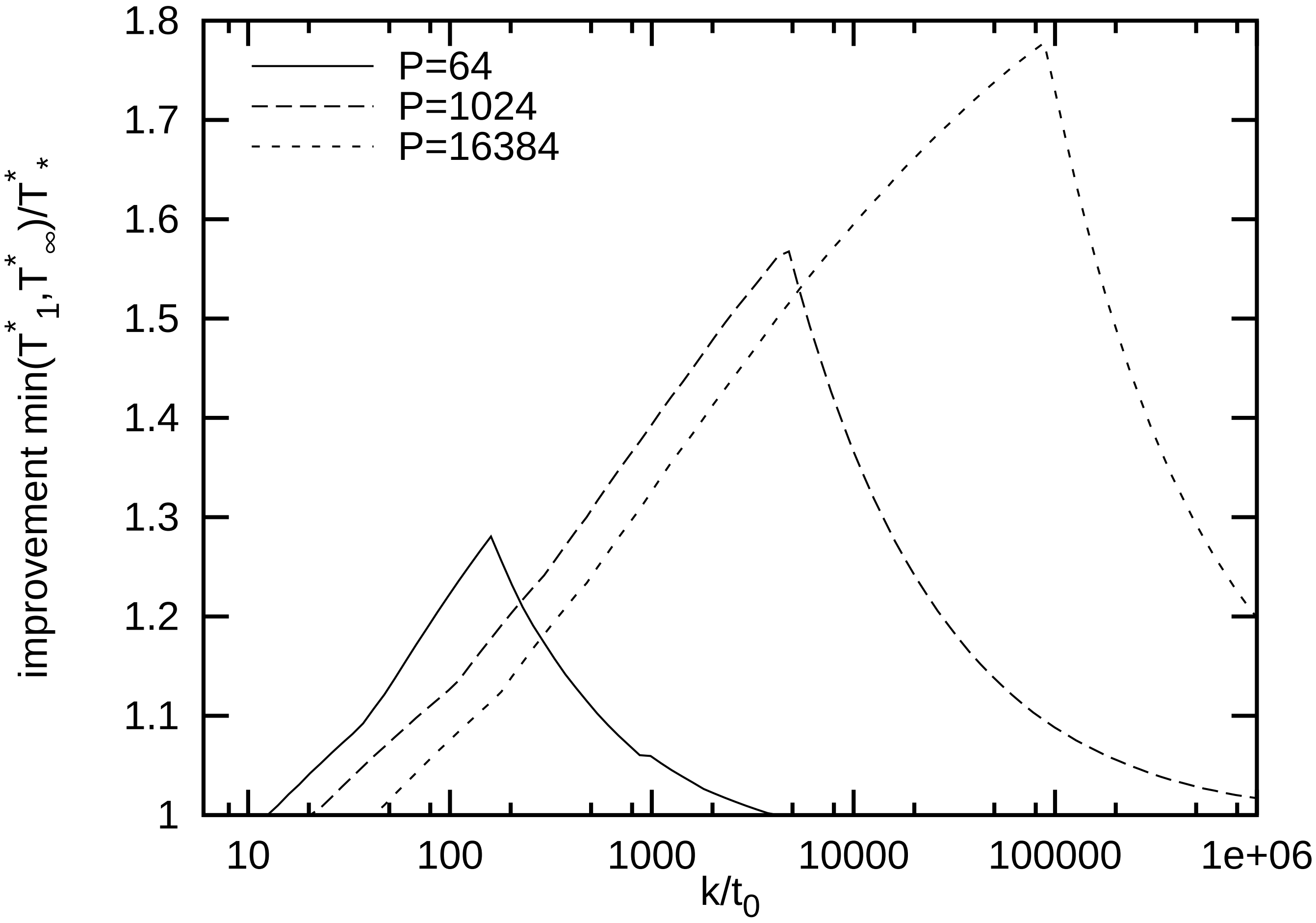
y Axis

clip outclassed algorithms



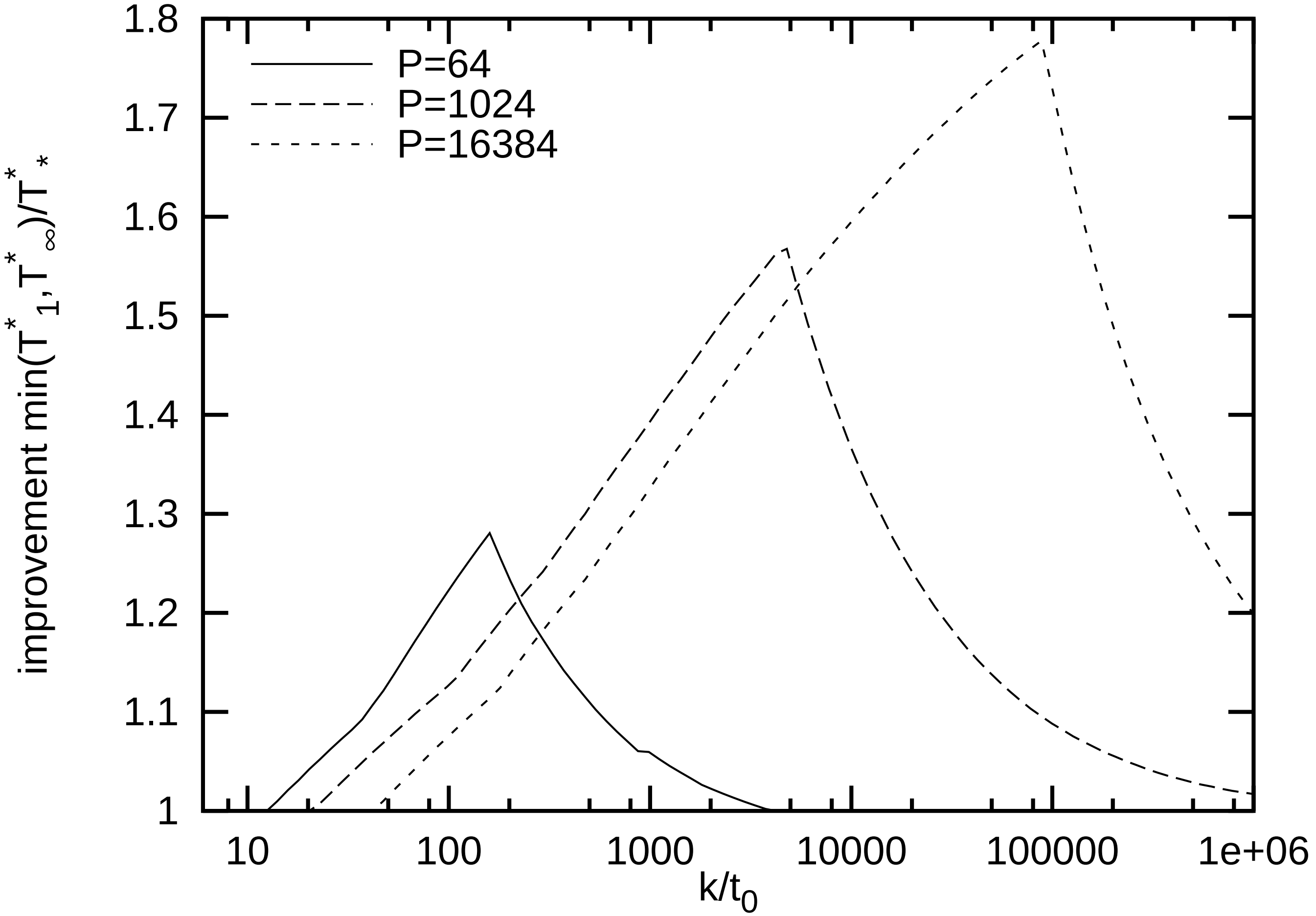
y Axis

vertical size: weighted average of the slants of the line segments in the figure should be about 45°[Cleveland 94]



y Axis

graph a bit wider than high, e.g., golden ratio [Tufte 83]



Multiple Curves

$+$ high information density

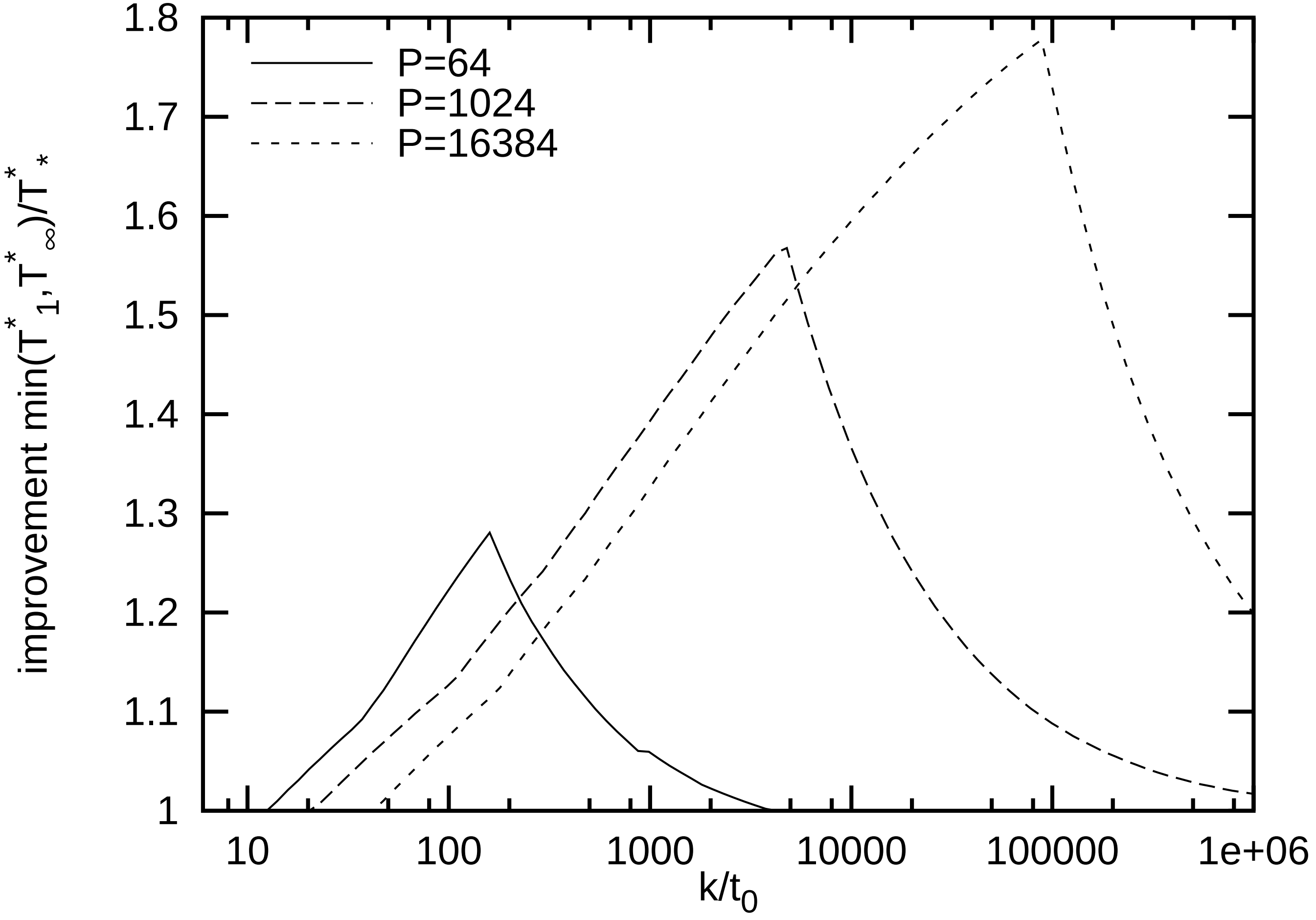
$+$ better than 3D (reading off values)

$-$ Easily overdone

≤ 7 smooth curves

Reducing the Number of Curves

use ratios



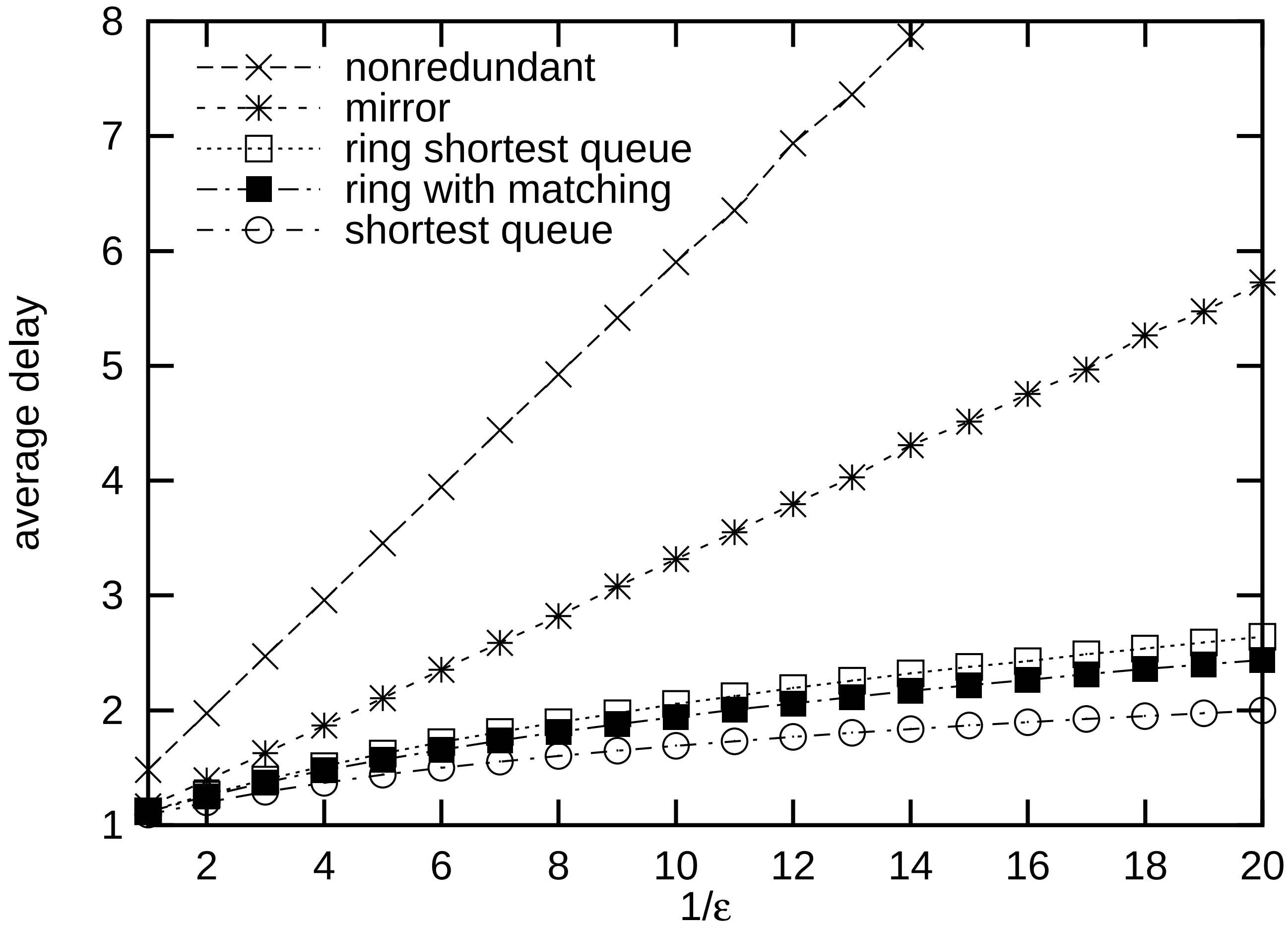
Reducing the Number of Curves

omit curves

- outclassed algorithms (for case shown)
- equivalent algorithms (for case shown)

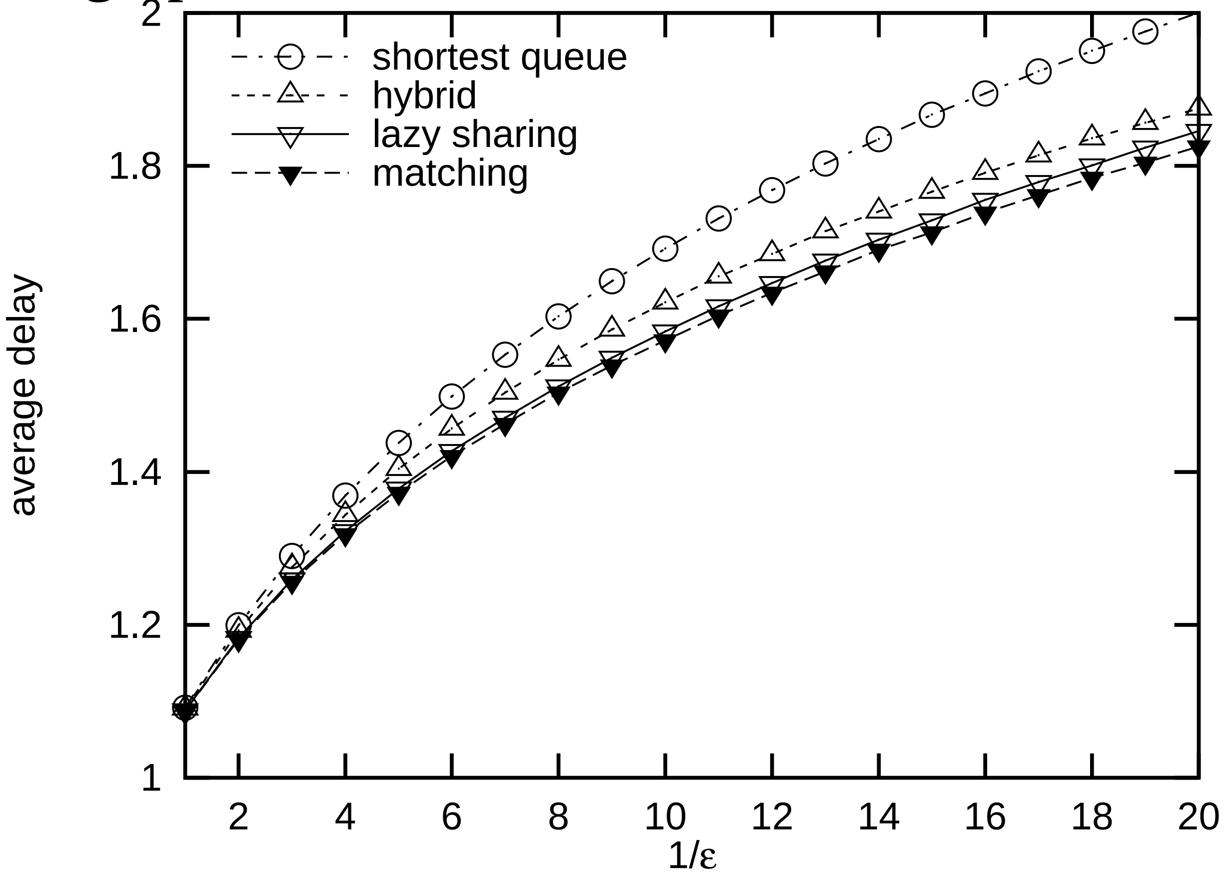
Reducing the Number of Curves

split into two graphs

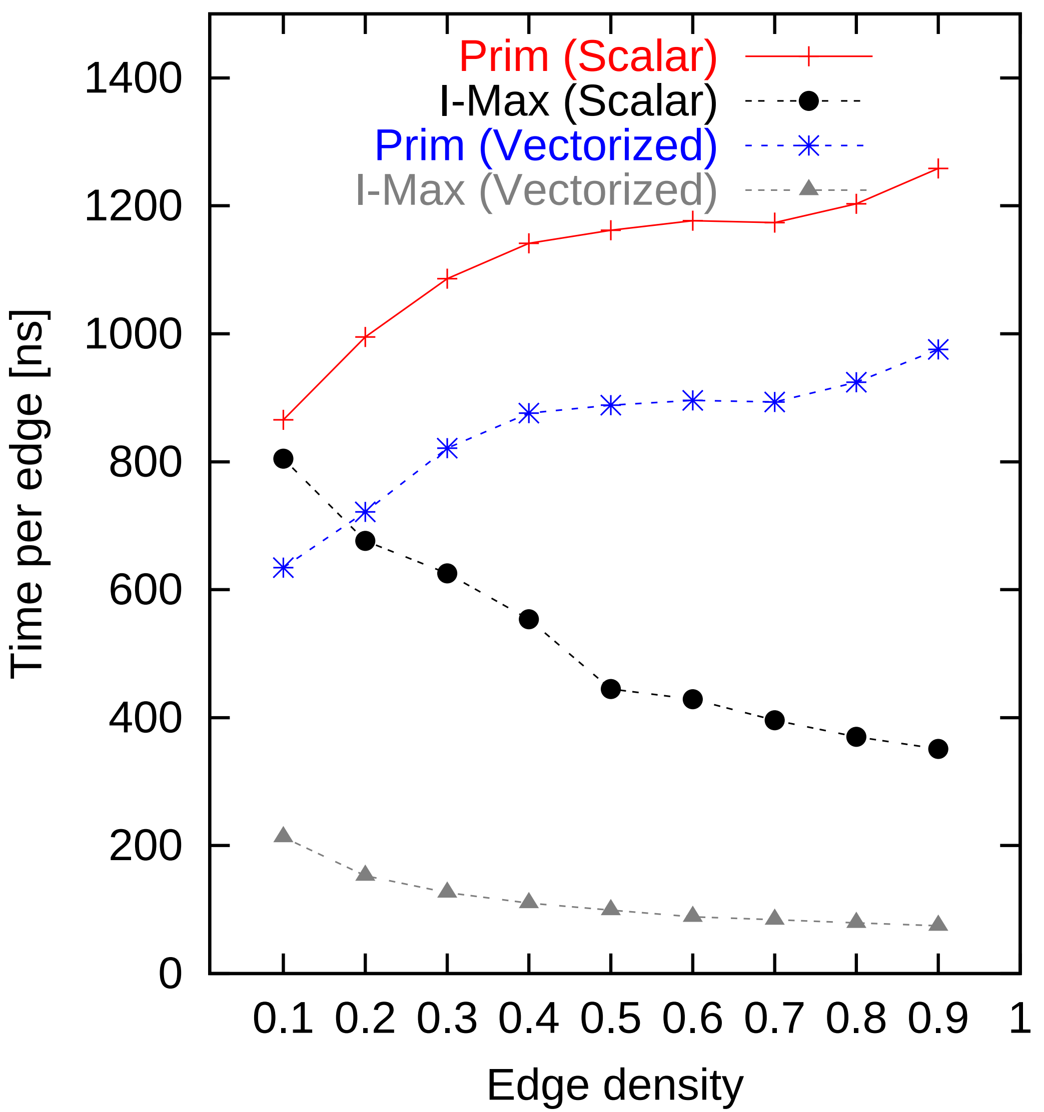


Reducing the Number of Curves

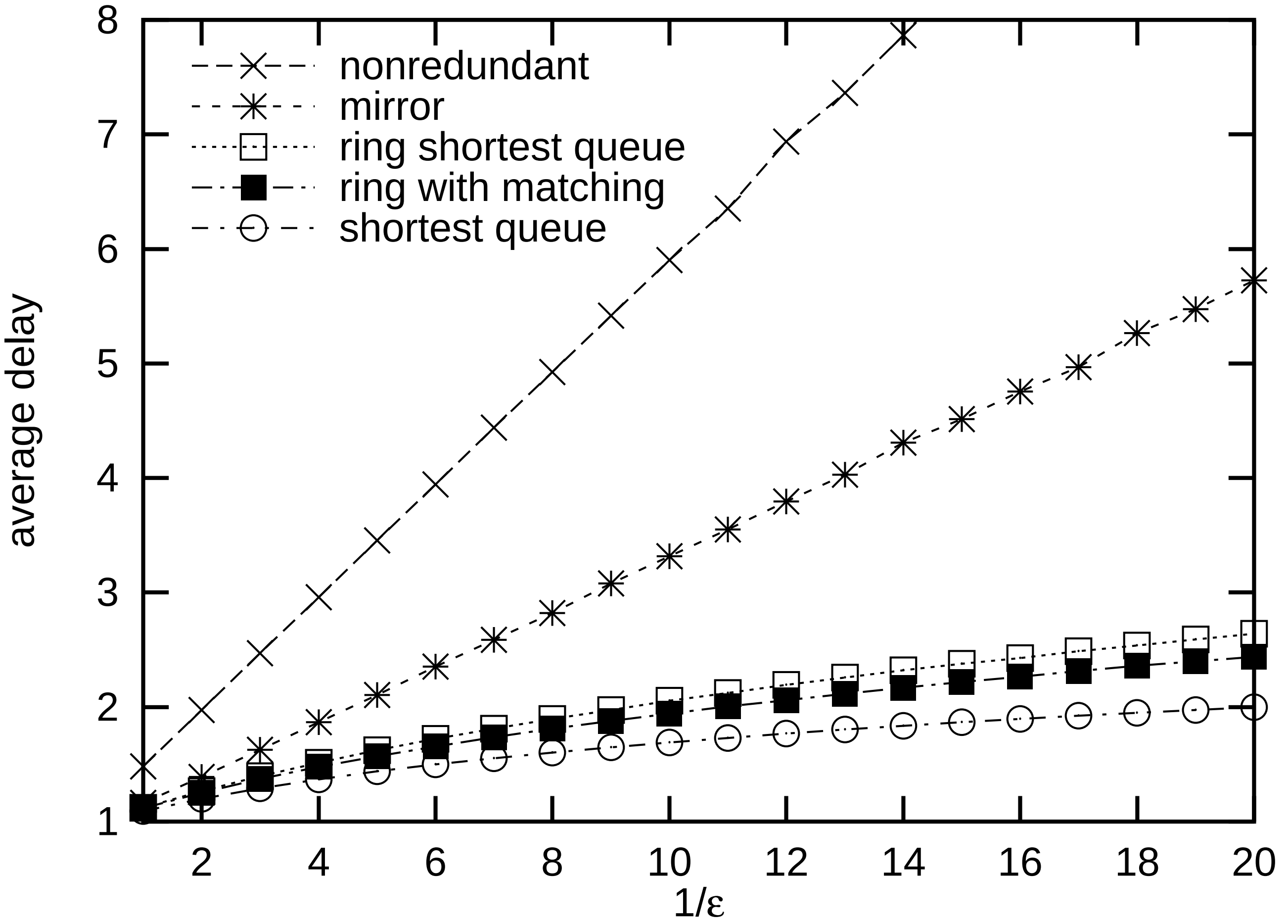
split into two graphs



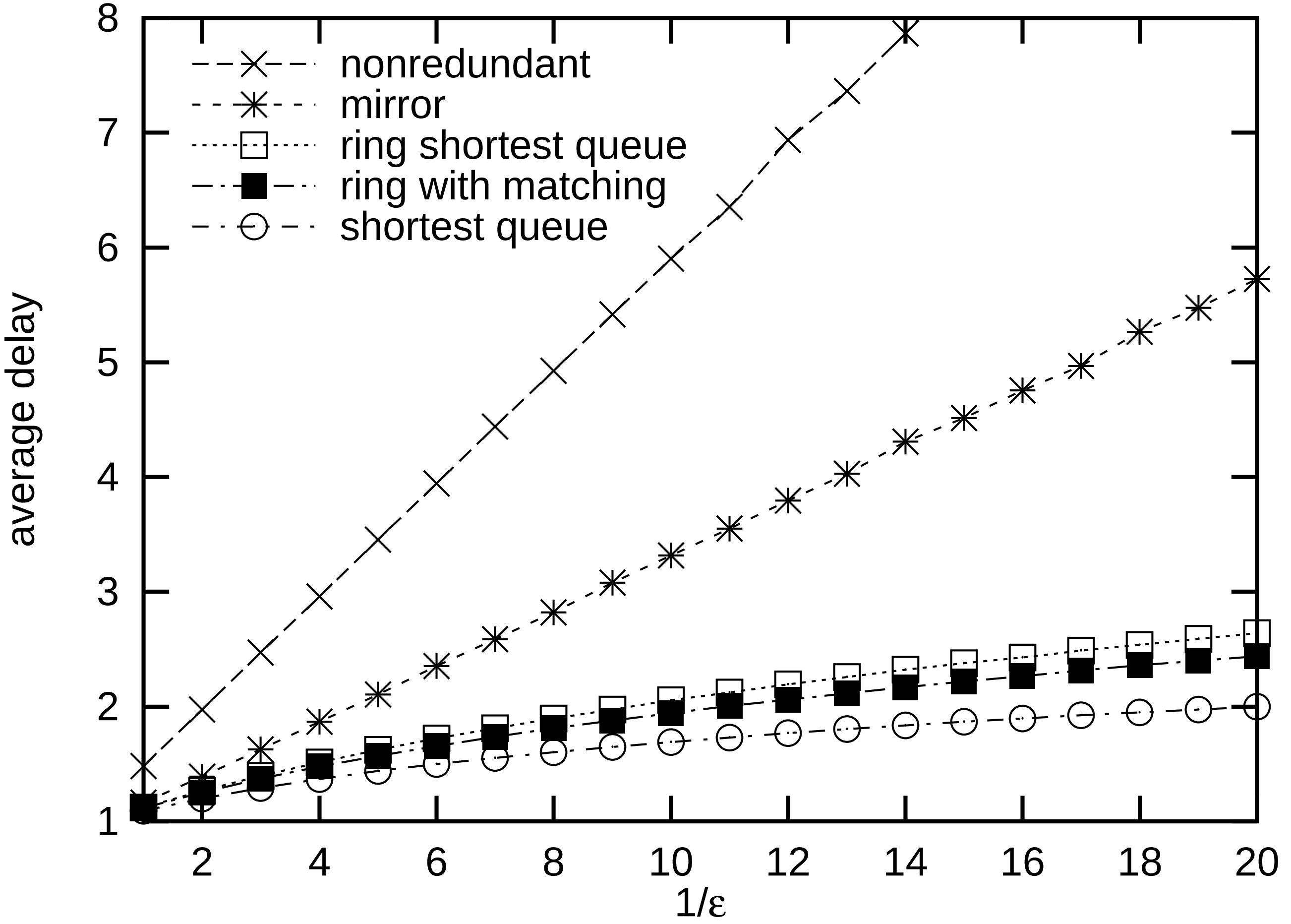
Keeping Curves apart: log y scale



Keeping Curves apart: smoothing



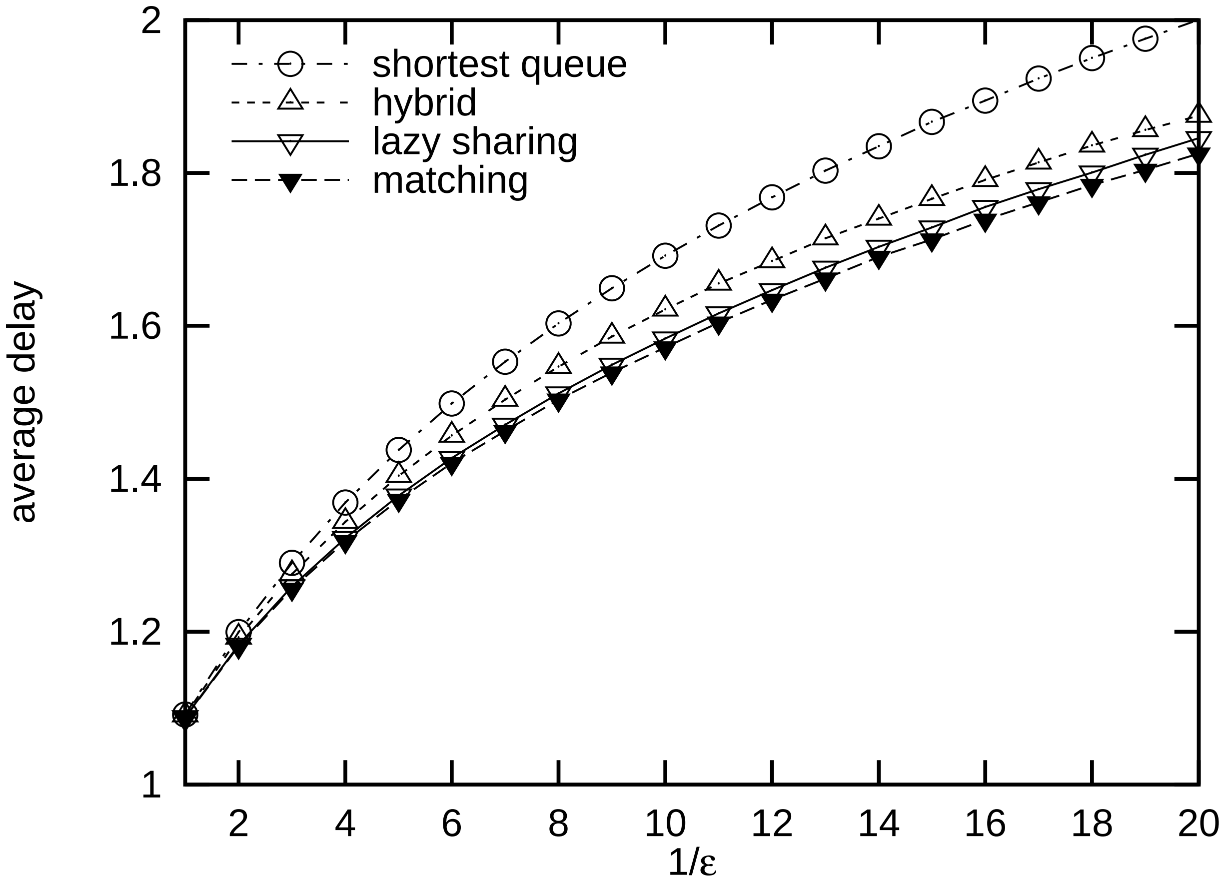
Keys



same order as curves

Keys

place in white space



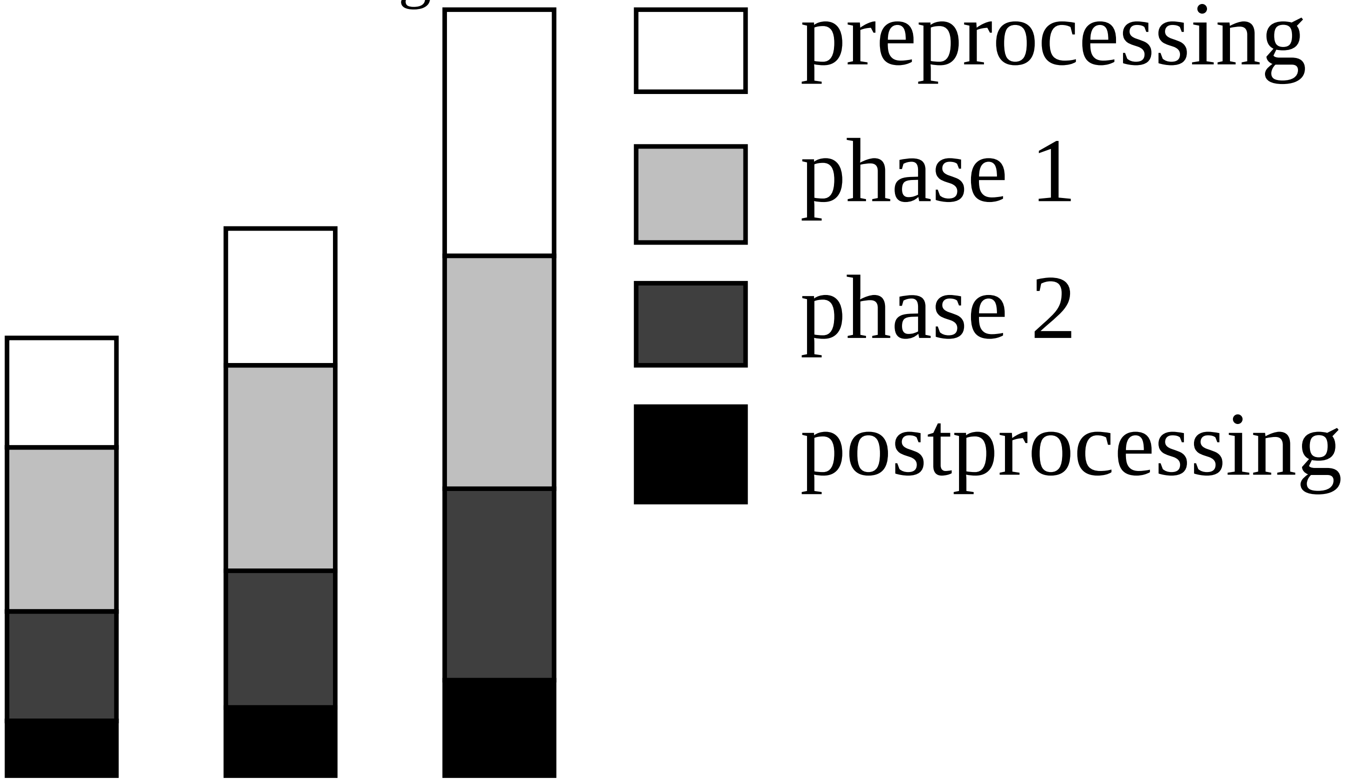
consistent in different figures

Todsünden

1. forget explaining the axes
2. connecting unrelated points by lines
3. mindless use/overinterpretation of double-log plot
4. cryptic abbreviations
5. microscopic lettering
6. excessive complexity
7. pie charts

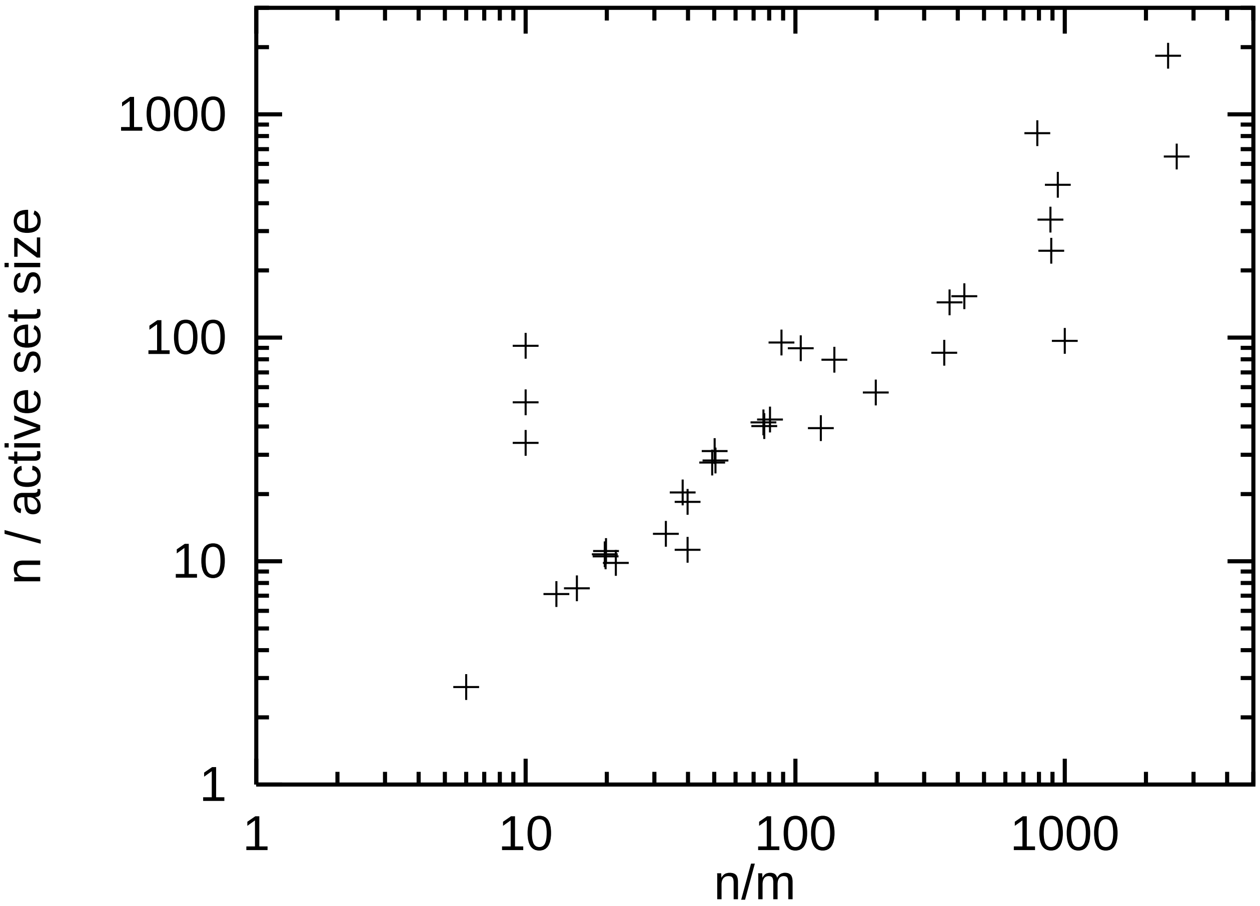
Arranging Instances

- bar charts
- stack components of execution time
- careful with shading



Arranging Instances

scatter plots



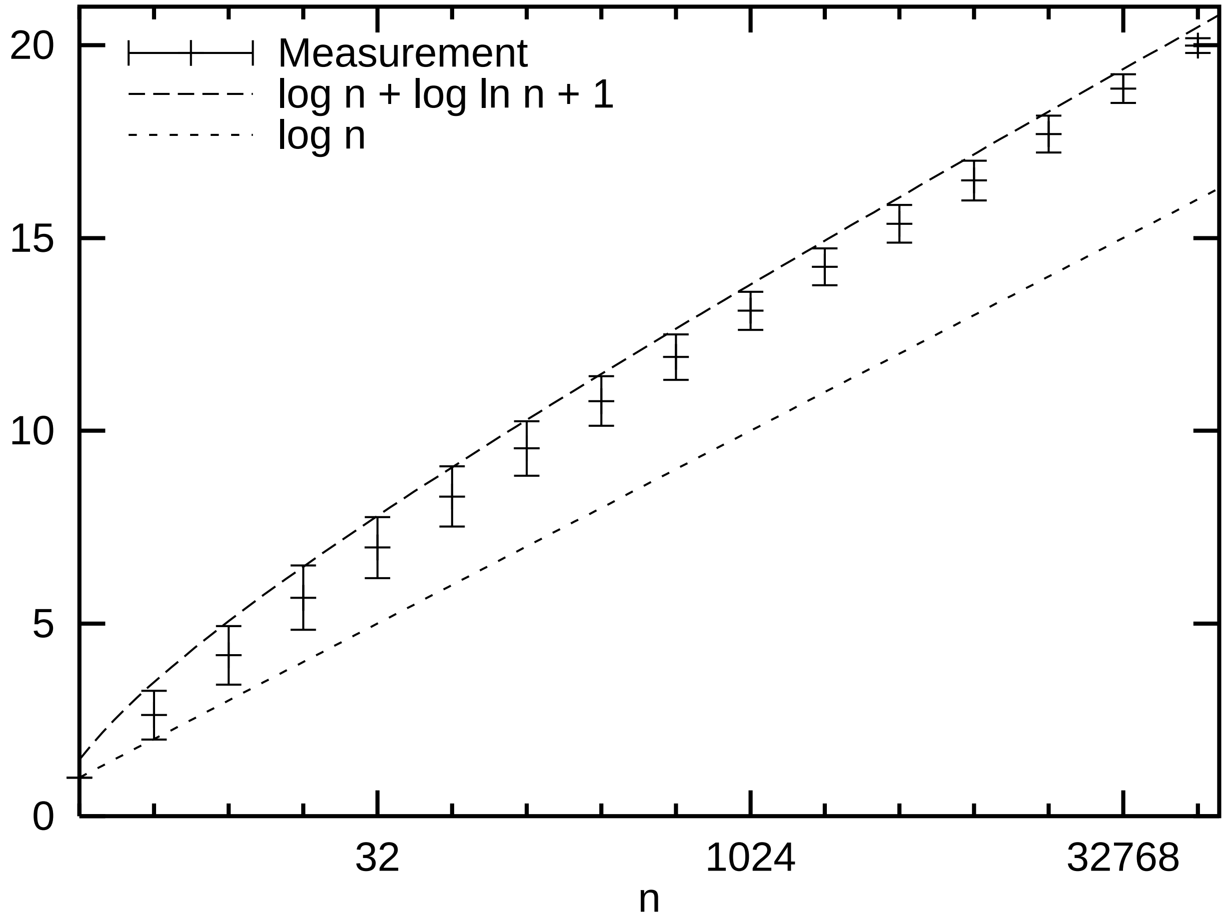
Measurements and Connections

- straight line between points do not imply claim of linear interpolation
- different with higher order curves
- no points imply an even stronger claim. Good for very dense smooth measurements.

Grids and Ticks

- Avoid grids or make it light gray
- usually round numbers for tic marks!
- sometimes plot important values on the axis

usually avoidable for randomized algorithms. median $\neq$ average,...



errors may not be of statistical nature!

3D

- – you cannot read off absolute values
- – interesting parts may be hidden
- – only one surface
- + good impression of shape

Caption

what is displayed

how has the date been obtained

surrounding text has more.

Check List

- ☐ Should the experimental setup from the exploratory phase be redesigned to increase conciseness or accuracy?
- ☐ What parameters should be varied? What variables should be measured? How are parameters chosen that cannot be varied?
- ☐ Can tables be converted into curves, bar charts, scatter plots or any other useful graphics?
- ☐ Should tables be added in an appendix or on a web page?
- ☐ Should a 3D-plot be replaced by collections of 2D-curves?
- ☐ Can we reduce the number of curves to be displayed?
- ☐ How many figures are needed?

- Scale the x-axis to make y-values independent of some parameters?
- Should the x-axis have a logarithmic scale? If so, do the x-values used for measuring have the same basis as the tick marks?
- Should the x-axis be transformed to magnify interesting subranges?
- Is the range of x-values adequate?
- Do we have measurements for the right x-values, i.e., nowhere too dense or too sparse?
- Should the y-axis be transformed to make the interesting part of the data more visible?
- Should the y-axis have a logarithmic scale?

- Is it be misleading to start the y-range at the smallest measured value?
- Clip the range of y-values to exclude useless parts of curves?
- Can we use banking to 45°?
- Are all curves sufficiently well separated?
- Can noise be reduced using more accurate measurements?
- Are error bars needed? If so, what should they indicate? Remember that measurement errors are usually *not* random variables.
- Use points to indicate for which x-values actual data is available.
- Connect points belonging to the same curve.

- Only use splines for connecting points if interpolation is sensible.
- Do not connect points belonging to unrelated problem instances.
- Use different point and line styles for different curves.
- Use the same styles for corresponding curves in different graphs.
- Place labels defining point and line styles in the right order and without concealing the curves.
- Captions should make figures self contained.
- Give enough information to make experiments reproducible.