



Priority Queues

Maintain set M of Elements with Keys

Insert:

DeleteMin:

Applications (no additional operations):

- Multi-way Merging (small)
- Greedy Algorithms (e.g., machine scheduling) (small–medium)
- Discrete event simulation (medium–large)
- Branch-and-Bound (large)
- Building long runs (large)
- Time forward processing (huge)

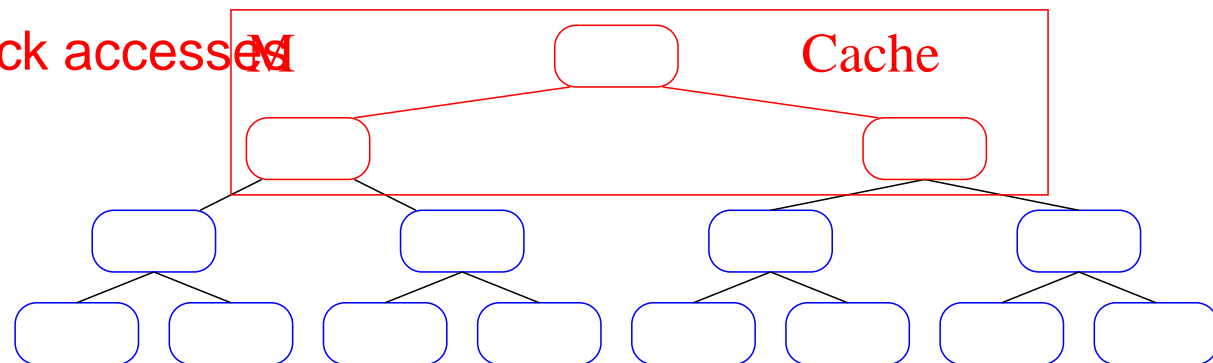


Binary Heaps

best comparison based “flat memory” algorithm

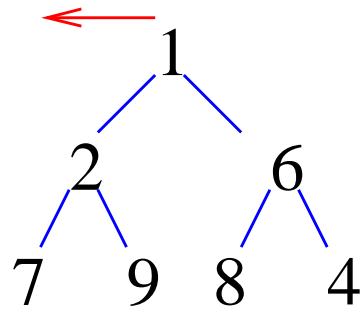
- + On average **constant** time for **insertion**
- + On average $\log n + o(1)$ key comparisons per delete-Min using the “bottom-up” heuristics [Wegener 93].

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





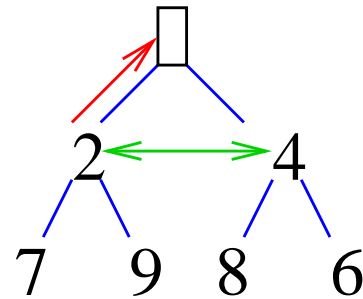
Bottom Up Heuristics



delete Min

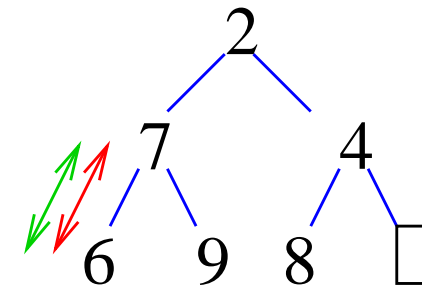
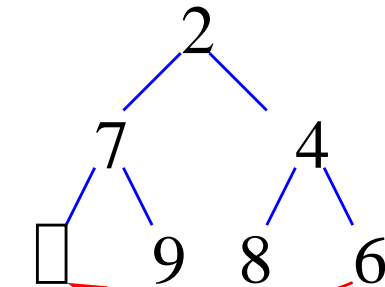
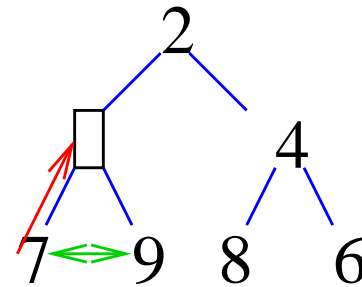
$O(1)$

compare \longleftrightarrow swap \longleftrightarrow move \longrightarrow



sift down hole

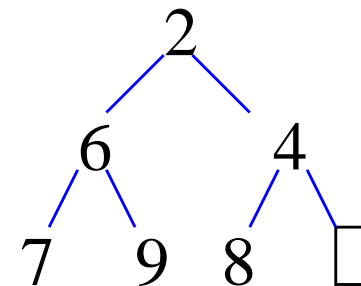
$\log(n)$



sift up

$O(1)$

average



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)

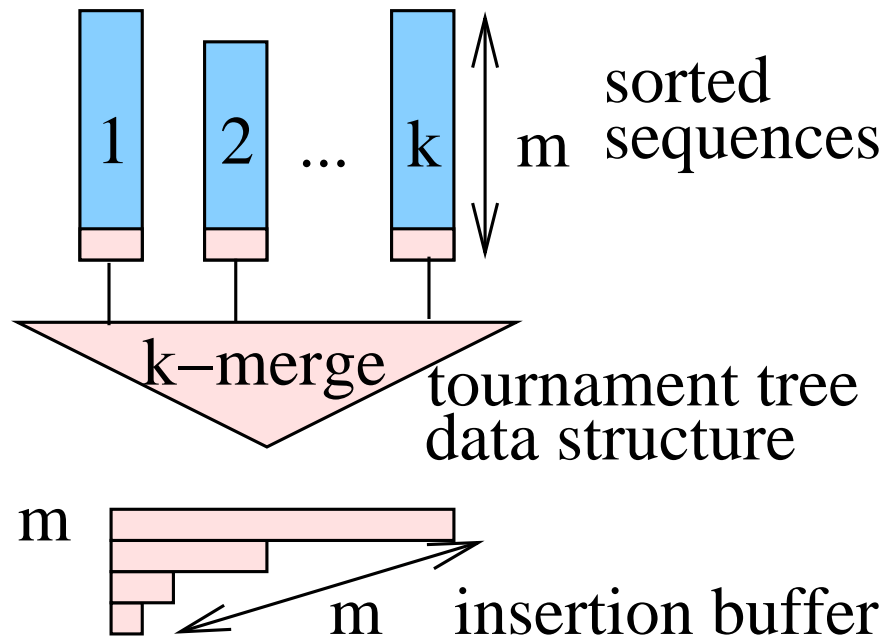
Rekursive Funktion für große Eingaben $2 \times$ schneller!

(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; move to external memory

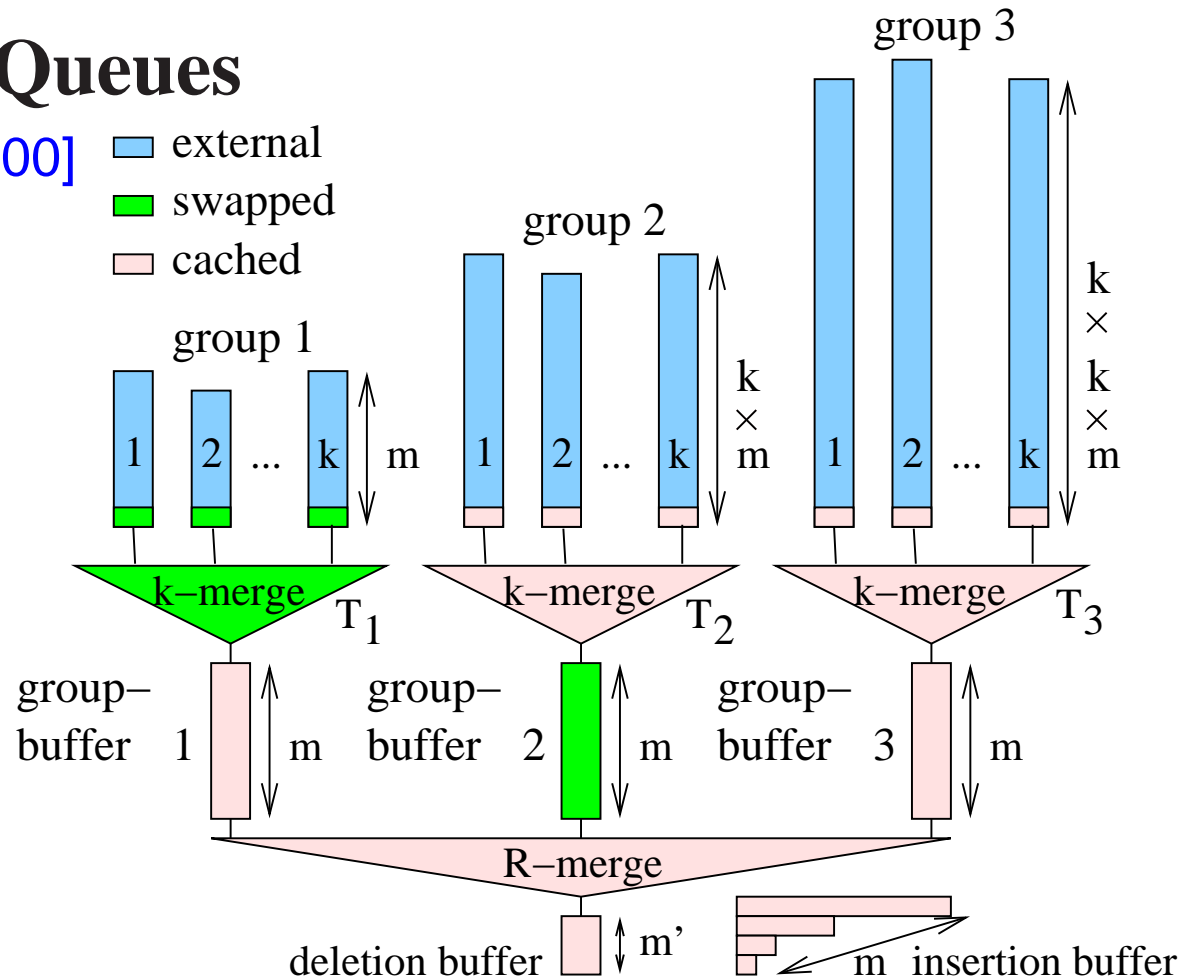
Delete-Min: Take minimum of insertion and merging queue.



Large Queues

[Sanders 00]

- external
- swapped
- cached



insert: group full \longrightarrow merge group; shift into next group.

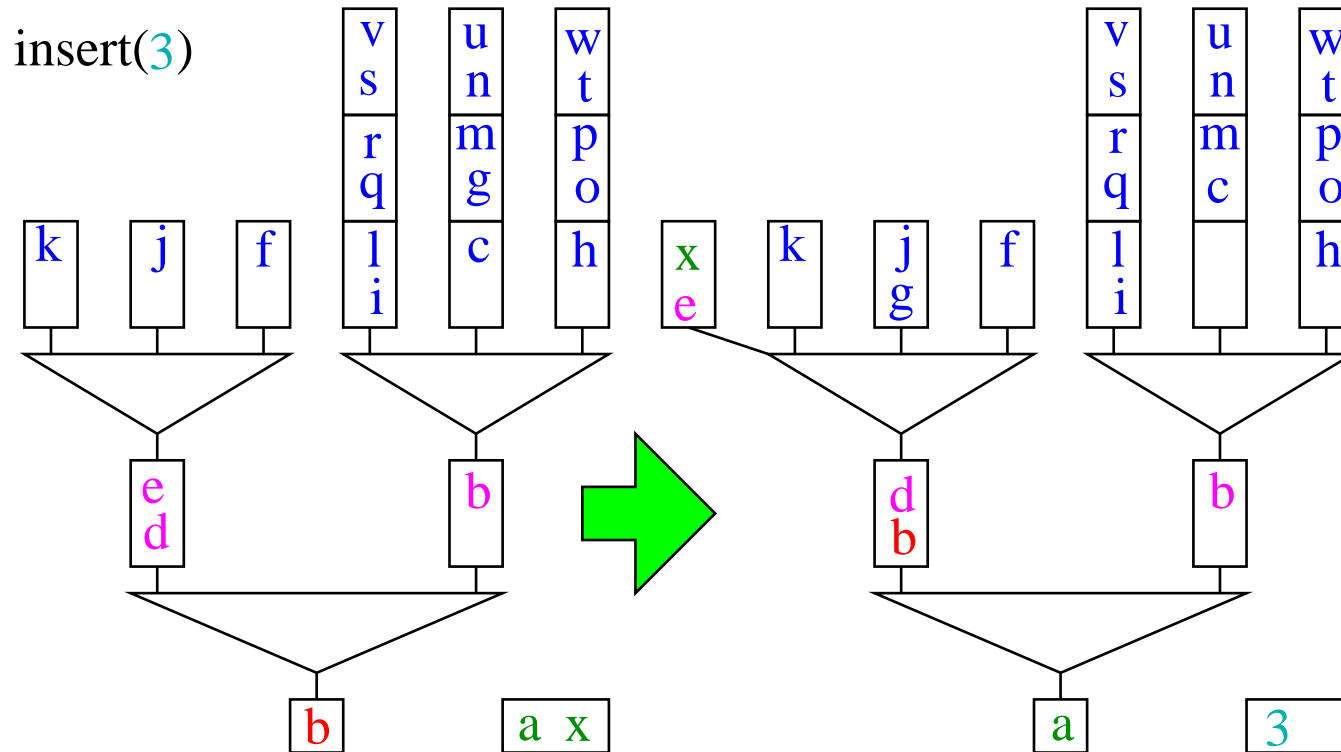
merge invalid group buffers and move them into group 1.

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



Example

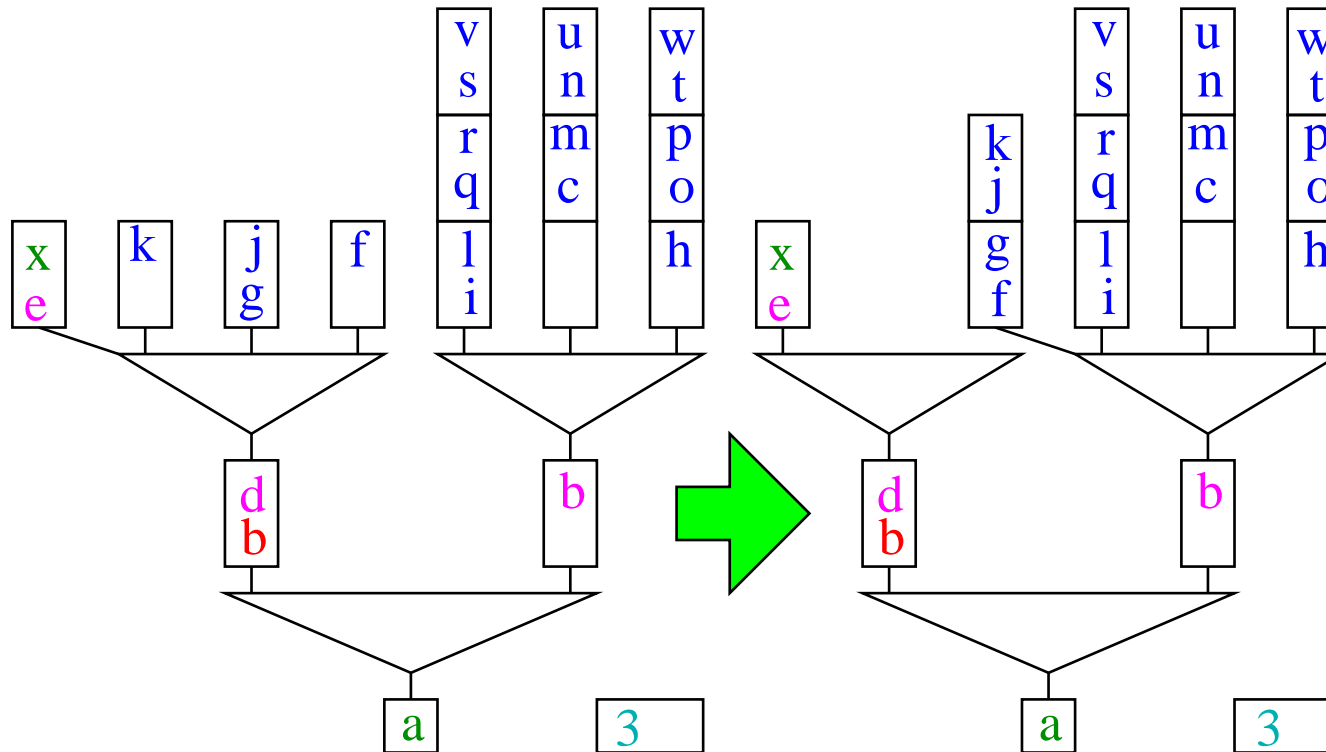
Merge insertion buffer, deletion buffer, and leftmost group buffer





Example

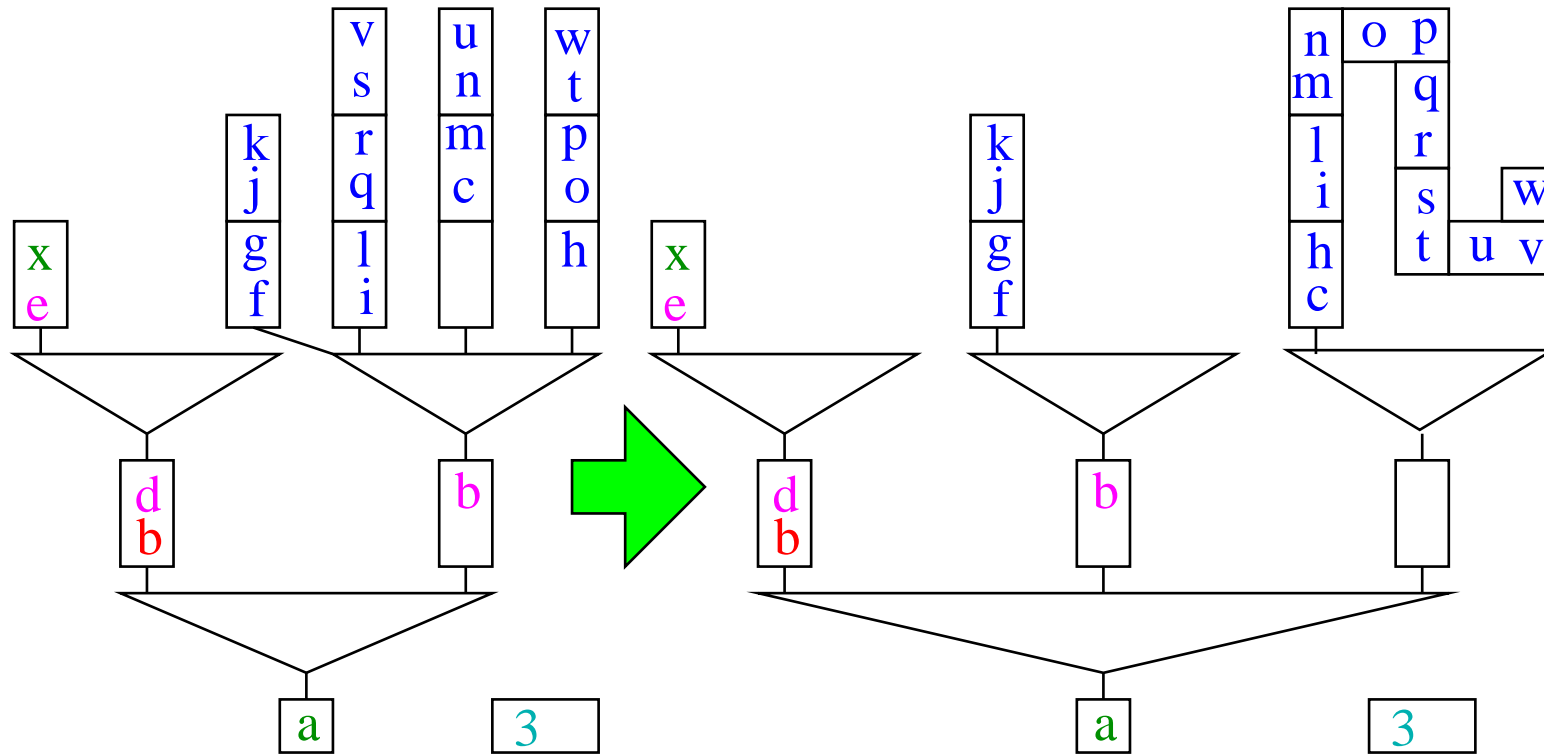
Merge group 1





Example

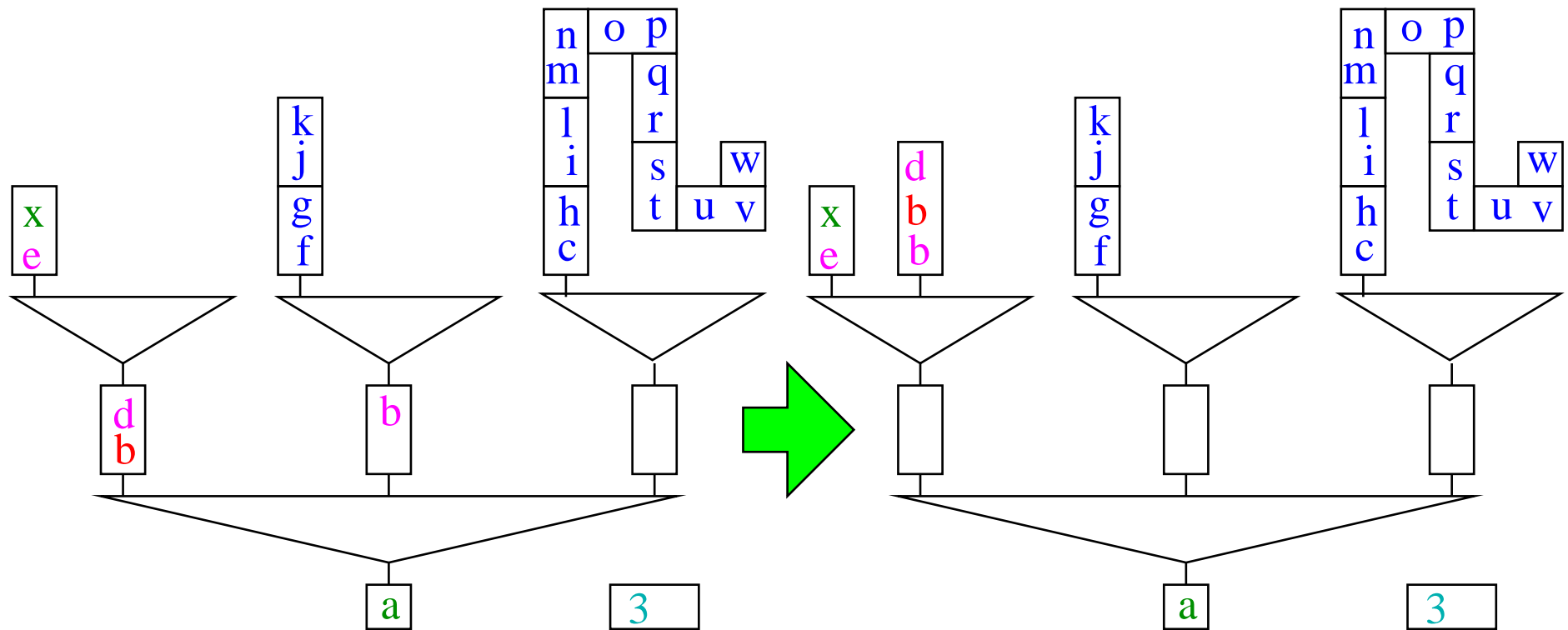
Merge group 2





Example

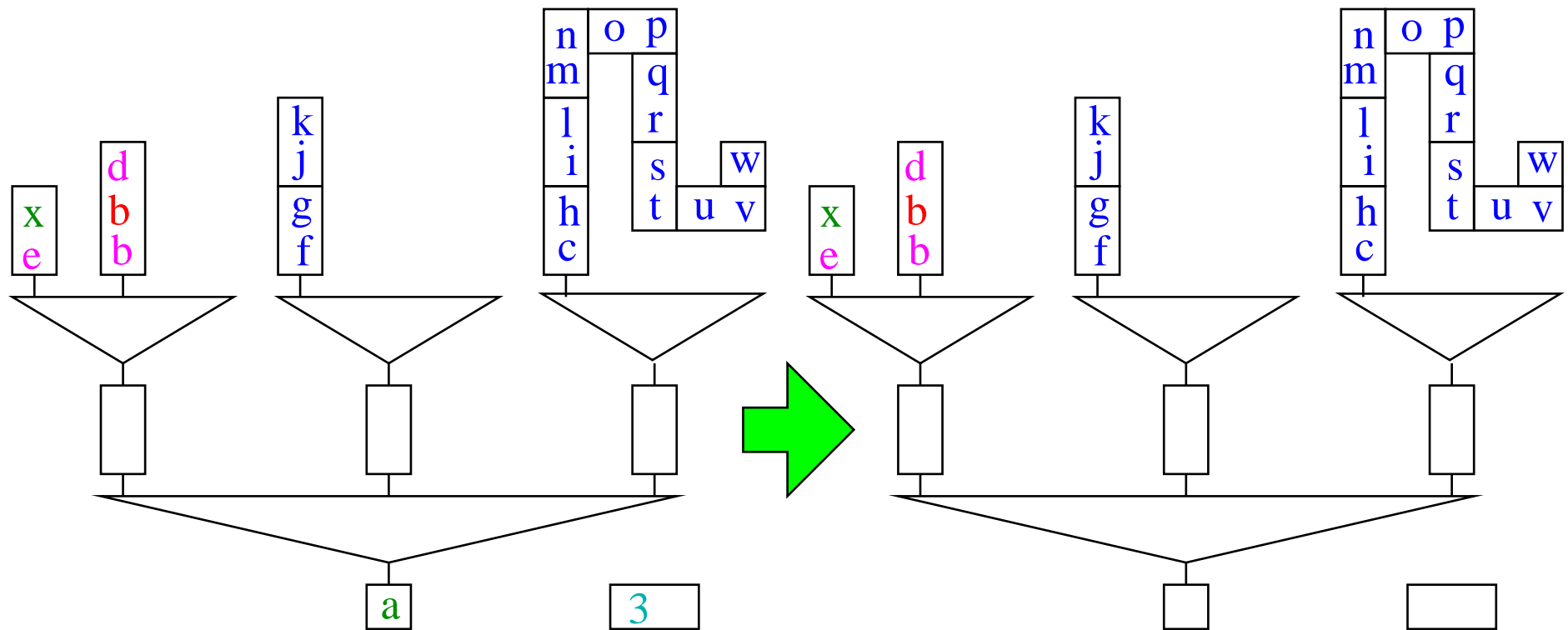
Merge group buffers





Example

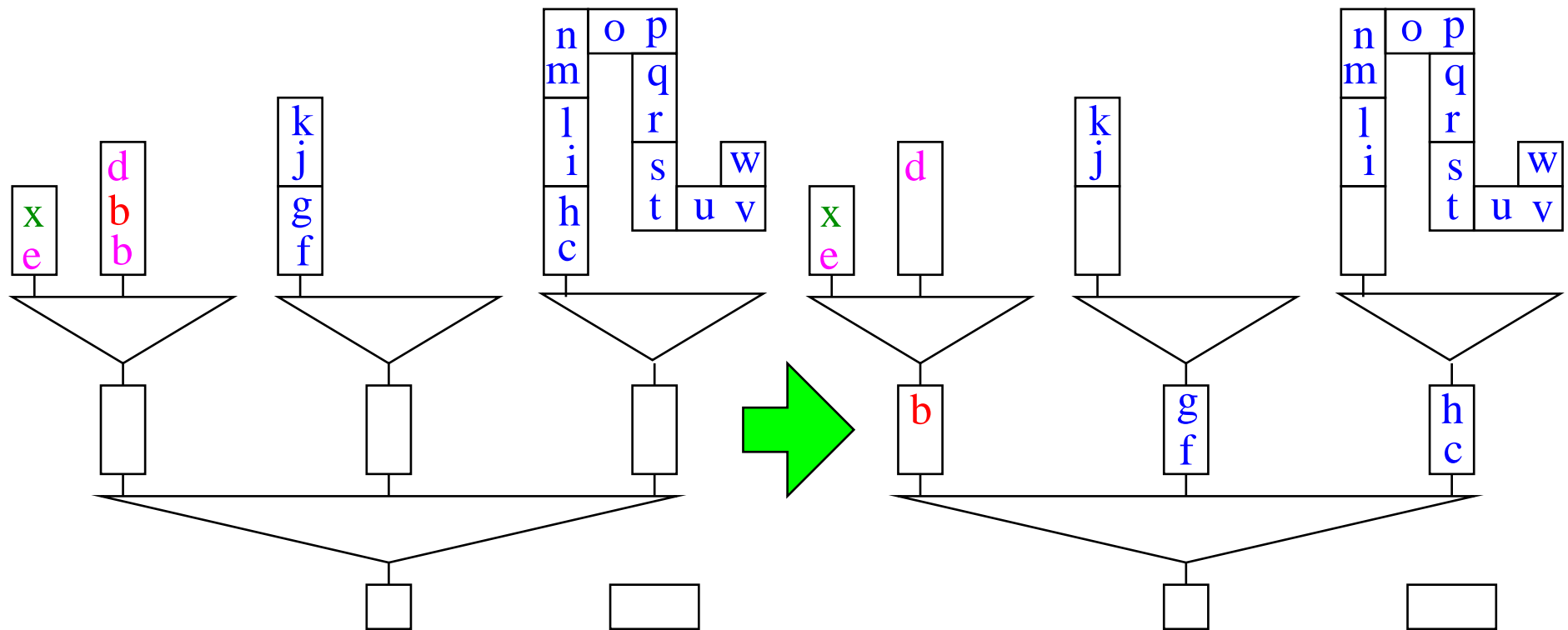
DeleteMin \rightsquigarrow 3; DeleteMin \rightsquigarrow a;





Example

DeleteMin \rightsquigarrow b





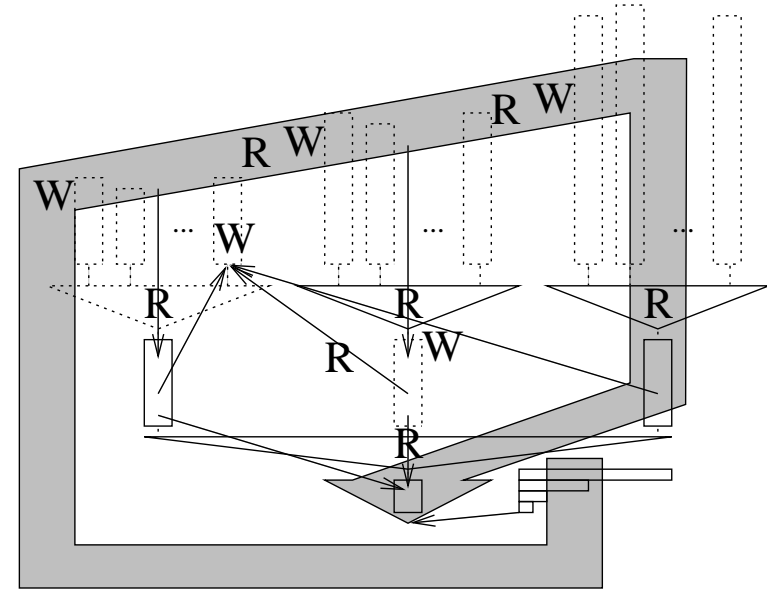
Analysis

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

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

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

(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

Near optimal

Group buffer size: 256

: performance on

Merging degree k : 128

all machines tried!

Compiler flags: Highly optimizing, nothing advanced

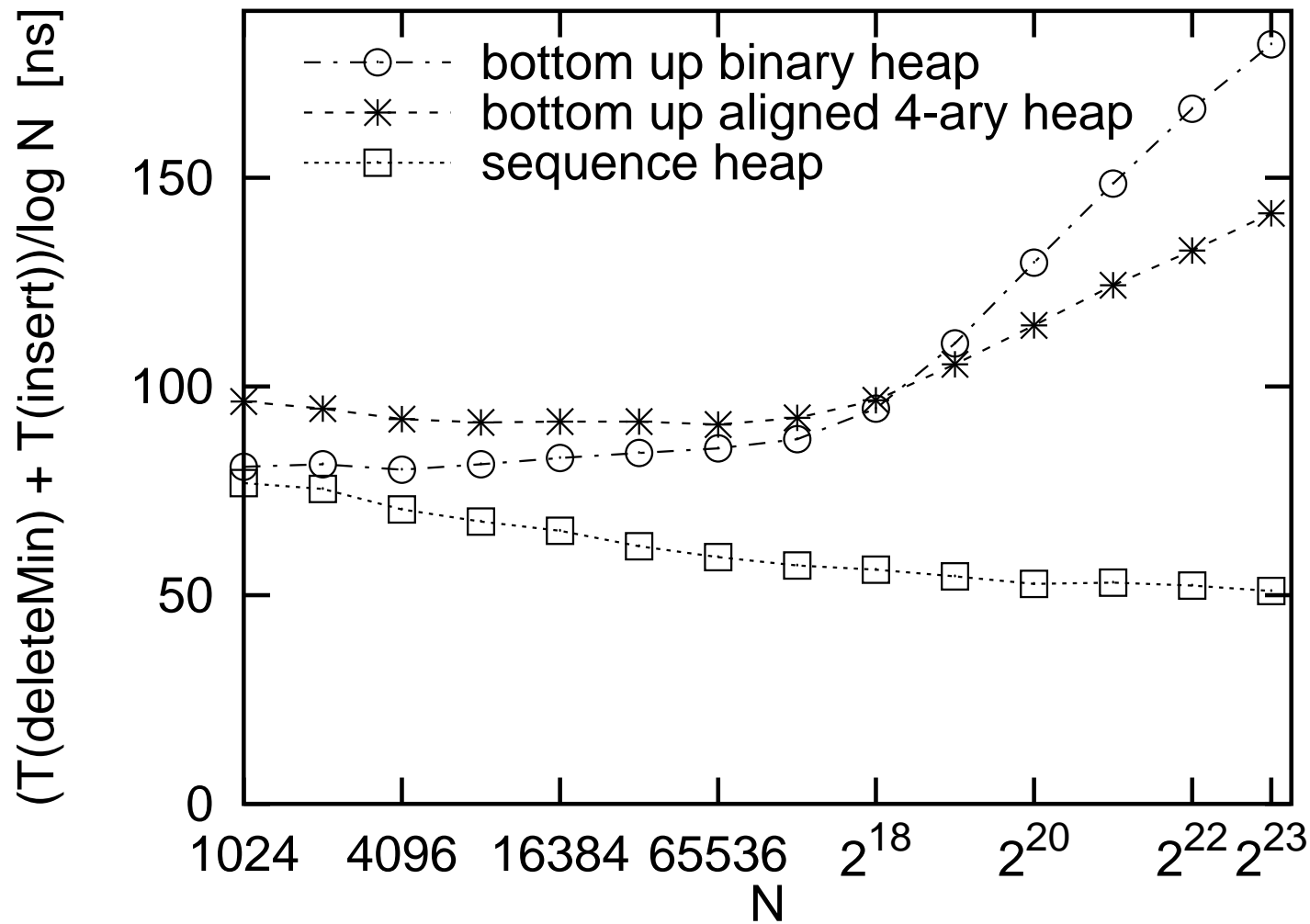
Operation Sequence:

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

Near optimal performance on all machines tried!

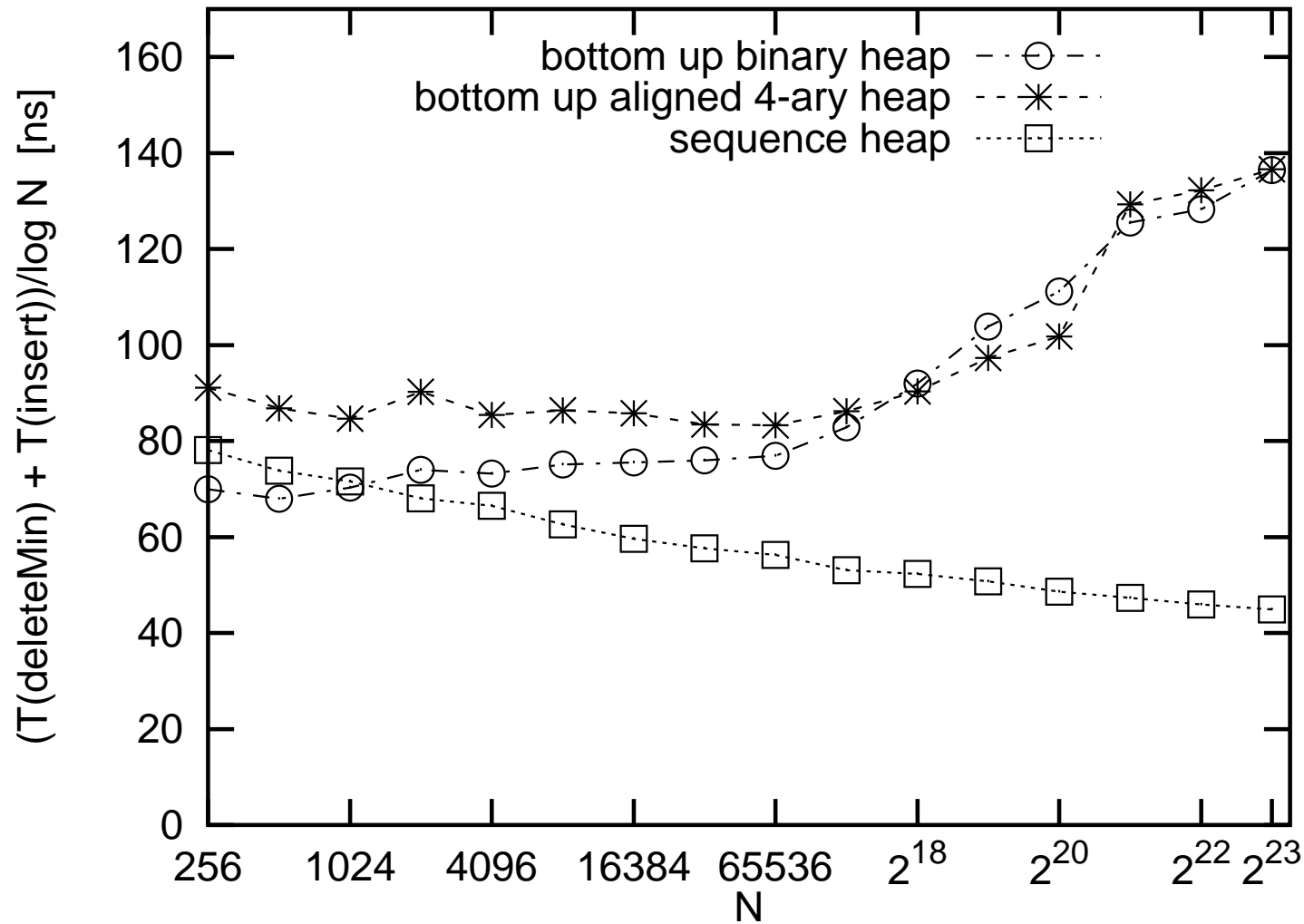


MIPS R10000, 180 MHz



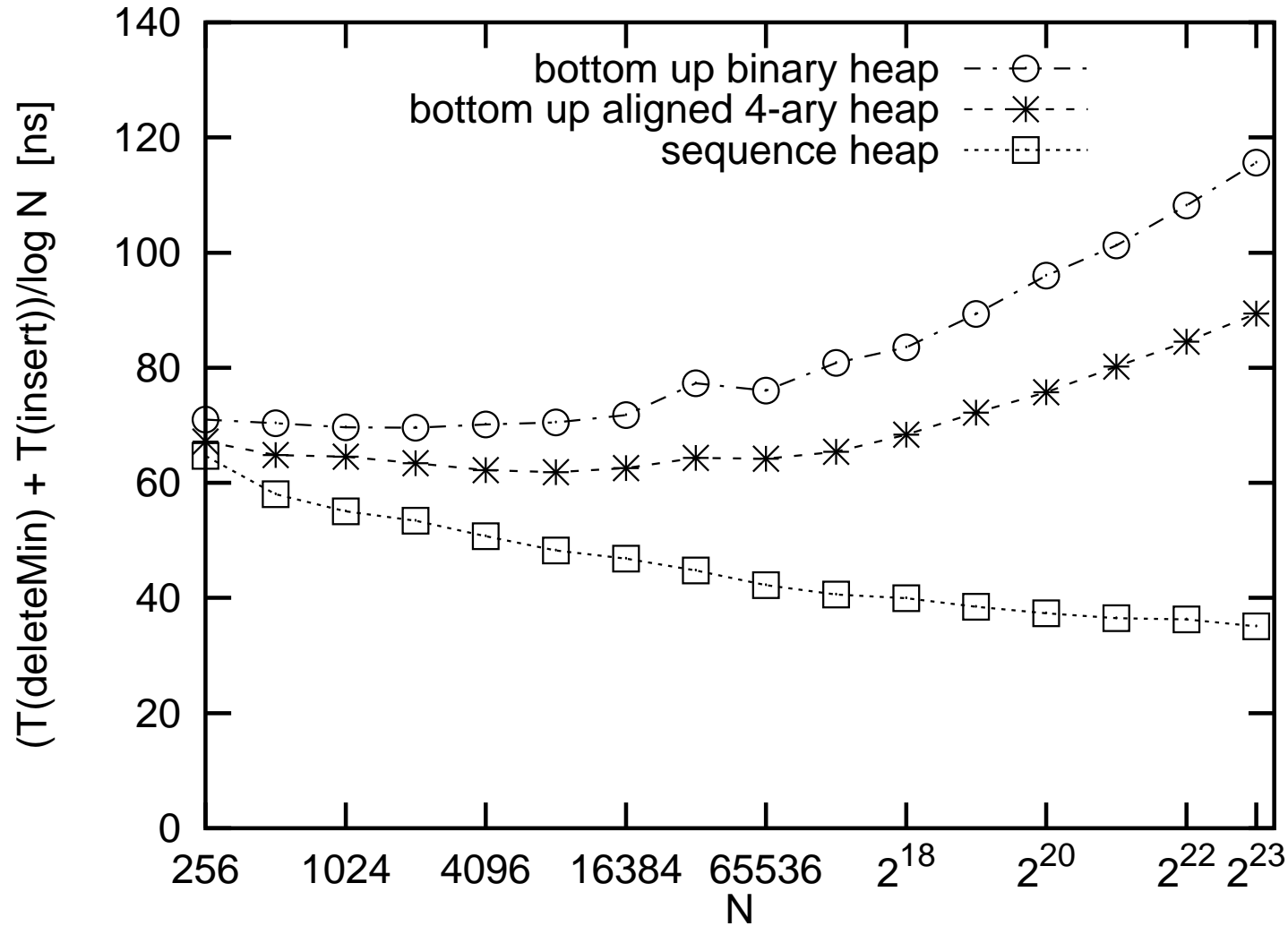


Ultra-SparcIIi, 300 MHz



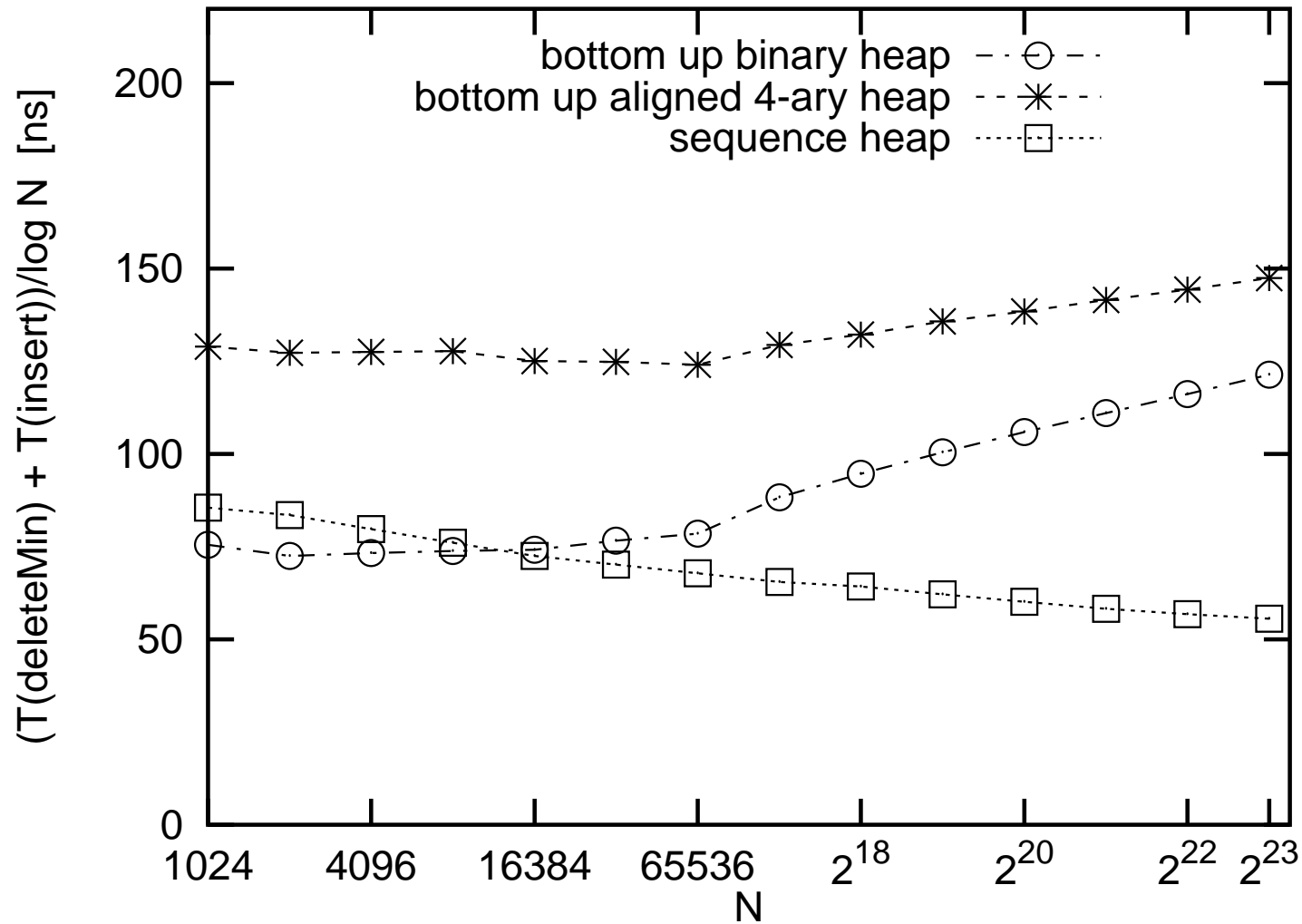


Alpha-21164, 533 MHz





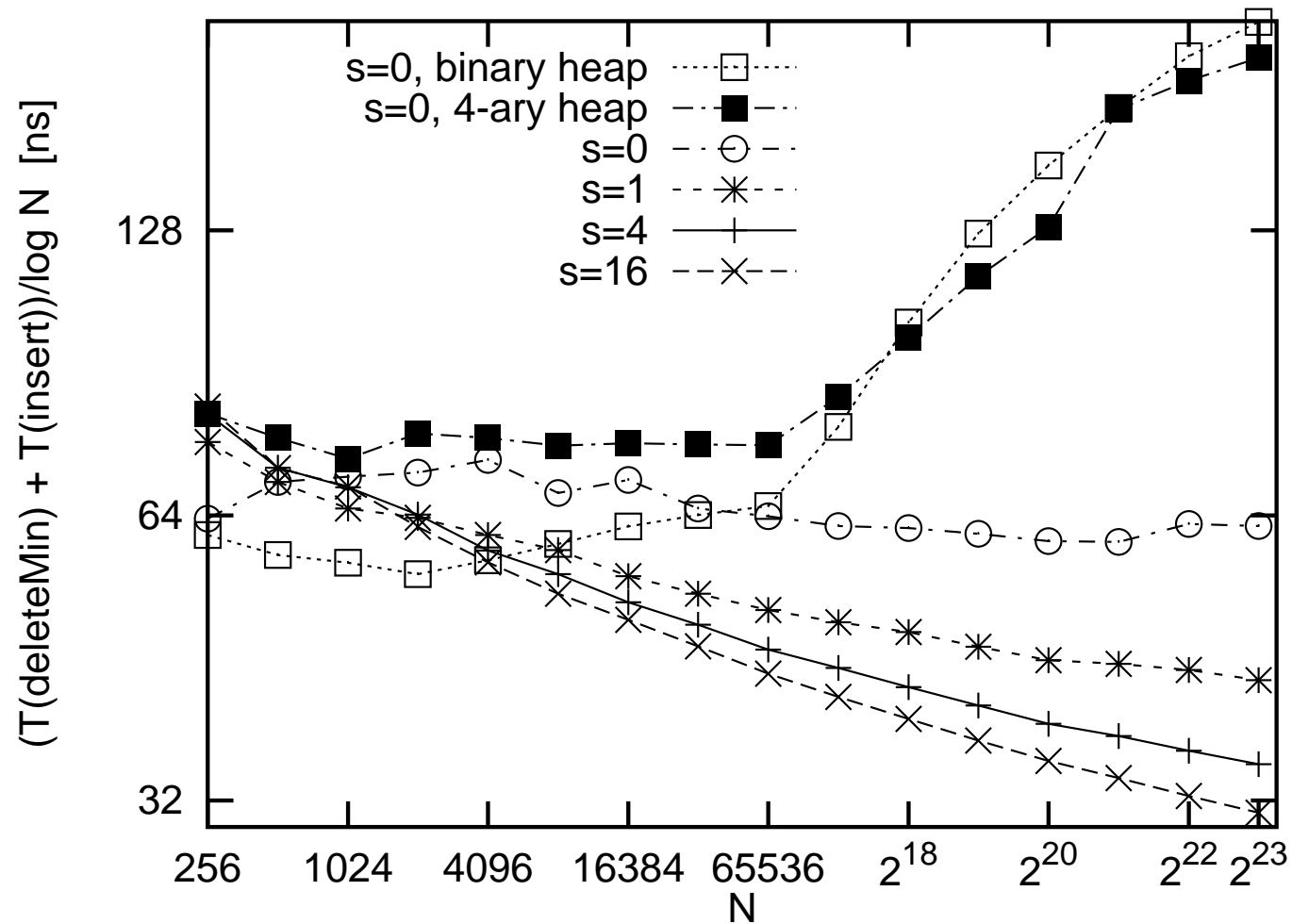
Pentium II, 300 MHz





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

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





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



Addressable Priority Queues

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

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') $M := M \cup M'$



Quick-Heaps

Idea:

Partition M into subsets M_1, \dots, M_k

Invariant:

$$\forall 1 \leq i < j \leq k, e_i \in M_i, e_j \in M_j : e_i \leq e_j$$

$|M_i|$ is “approximately” **growing geometrically** with i



Quick-Heaps – Insert

Procedure insert(e)

for $i := k$ **downto** 1 **while** $e < \min M_i$ **do** ;// define $\min M_0 = -\infty$

if $i = 0$ **then** $i := 1$; $\min M_1 := e$

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

decreaseKey: similar



Quick-Heaps – deleteMin

Procedure deleteMin

$e := \min M_1$

$M_1 := M_1 \setminus \{e\}$

if $M_1 = \emptyset$ **then**

(recursively) partition M_2 and renumber
to reestablish invariant

return e



Quick-Heaps – Analysis

Paredes: average case complexity $O(\log n)$

Experiments: with very favorable constant factors.

Open Problem:

Make fit for worst case inputs.

$O(1)$ expected cost for decreaseKey?

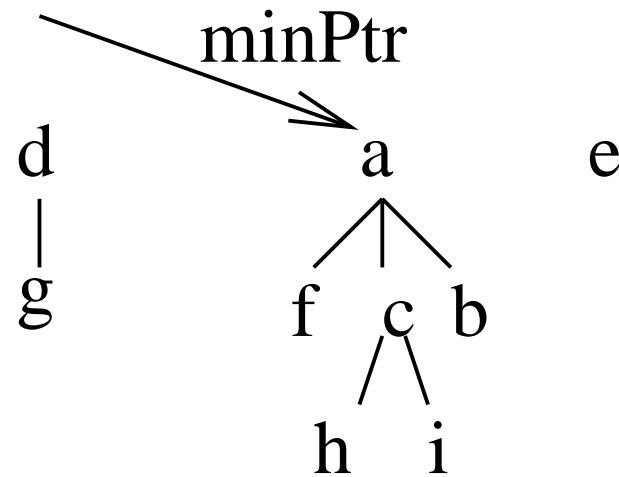
Idea: active rebalancing



Heap Ordered Forest

Basic Data Structure

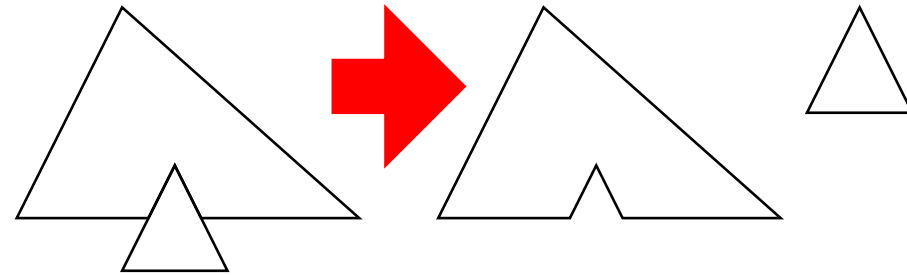
A forest of heap-ordered trees



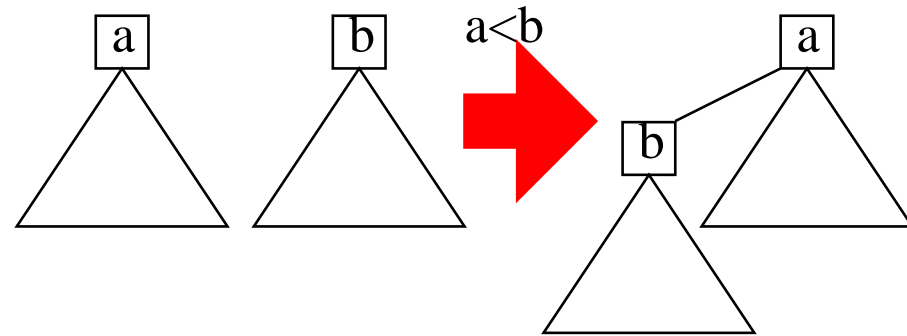


Manipulating Forests

Cut:



Link:





Pairing Heaps

Procedure insertItem(h : Handle)

newTree(h)

Procedure newTree(h : Handle)

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

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



Pairing Heaps

Procedure decreaseKey(h : Handle, k : Key)

key(h) := k

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



Pairing Heaps

Function deleteMin : Handle

$m := \text{minPtr}$

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 : AddressablePQ)

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

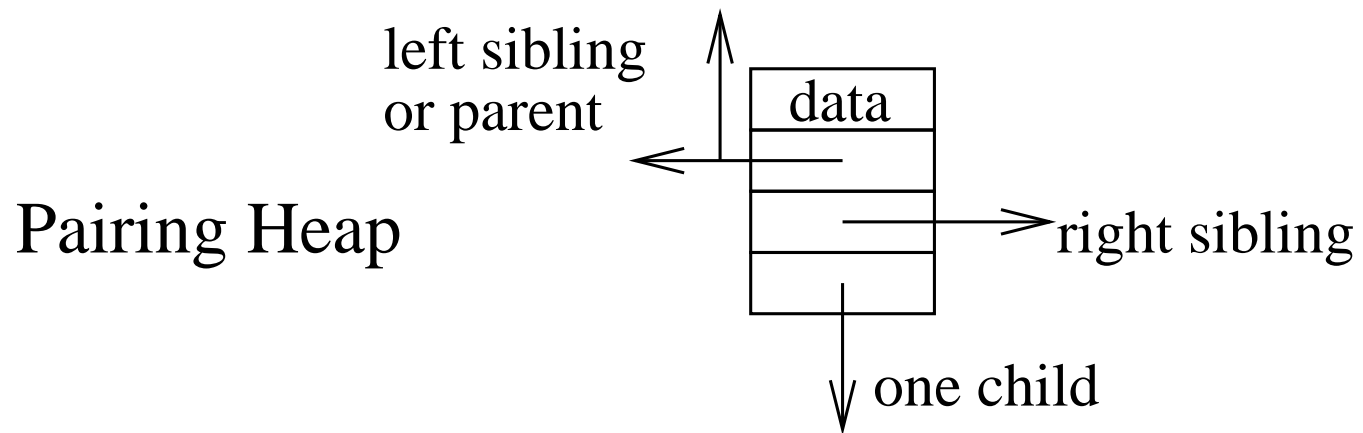
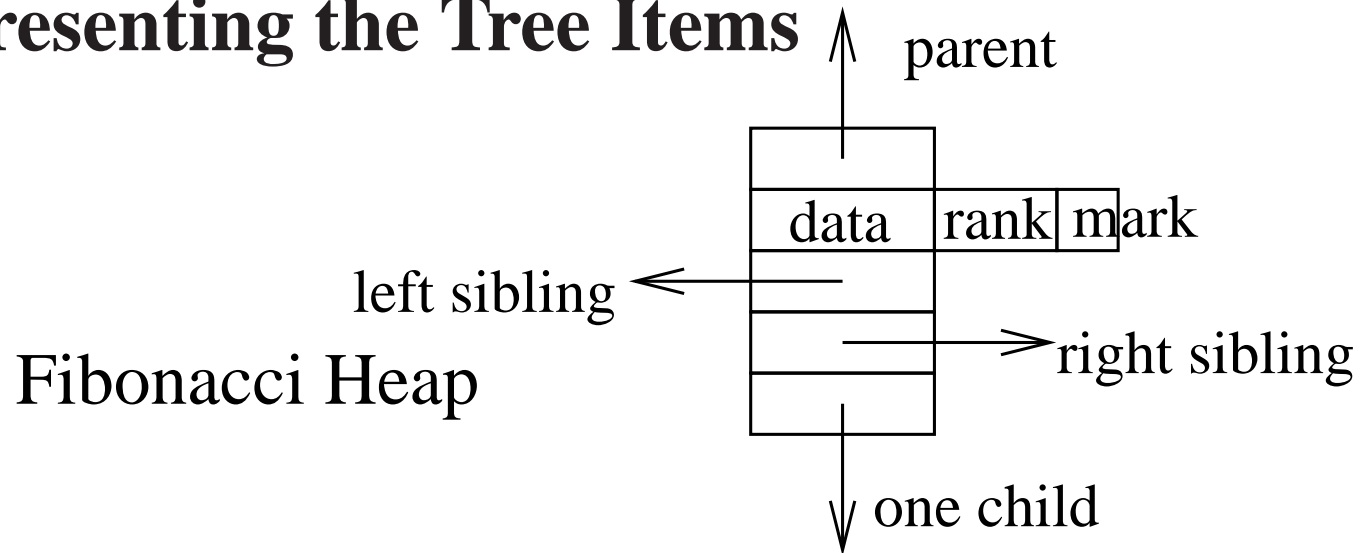
Cascading cuts: cut marked nodes (i.e., lost two childs)

Amortized complexity: $O(\log n)$ for deleteMin

$O(1)$ for all other operations



Representing the Tree Items





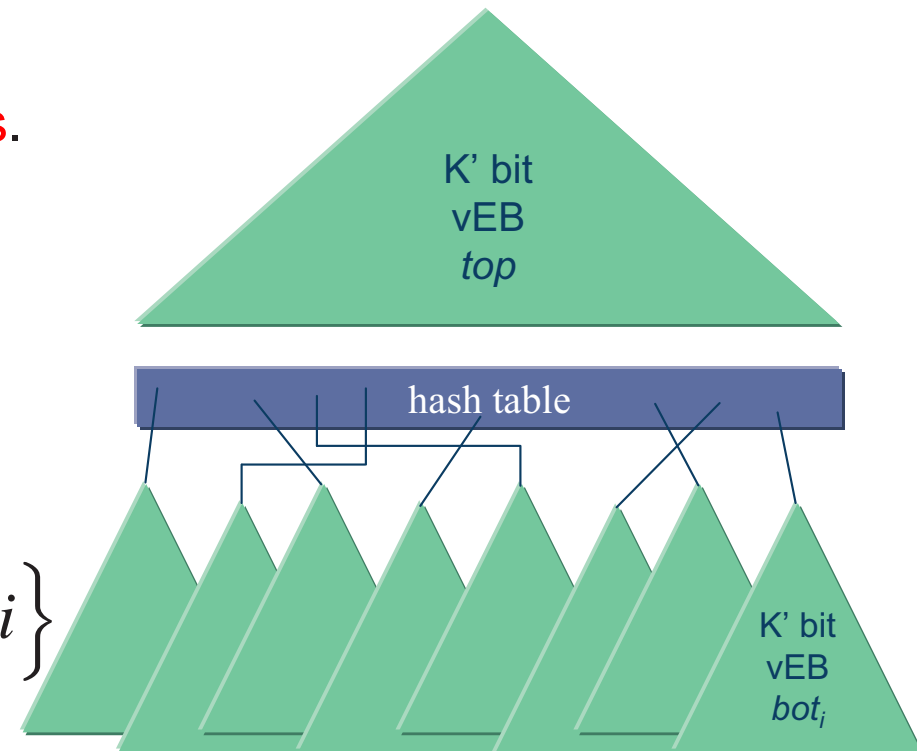
Addressable Priority Queues: Todo

- 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) nothing known for memory hierarchies



van Emde-Boas Search Trees

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





```

Function locate( $i, K$ )
    if  $i > \max M$  then return  $\infty$ 
    if  $|M| = \{x\}$  then return  $x$ 
    if  $K = 1$  then return  $1$ 
     $h := i \text{ div } 2^{(K/2)}$ 
     $\ell := i \text{ mod } 2^{(K/2)}$ 
    if  $M_h = \emptyset$  then
         $j := t.\text{locate}(h, K/2)$ 
        return  $j2^{(K/2)} + \min M_j$ 
    else
        if  $i > \max M_h$  then return  $(h + 1)2^{(K/2)} + \min M_{h+1}$ 
        return  $h2^{(K/2)} + M_h.\text{locate}(\ell, K/2)$ 

```

// return $\min \{j \in M : j \geq i\}$
 // store max explicitly
 // $M = \{0, 1\}$
 // store min directly



Comparison with Comparison Based Search Trees

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

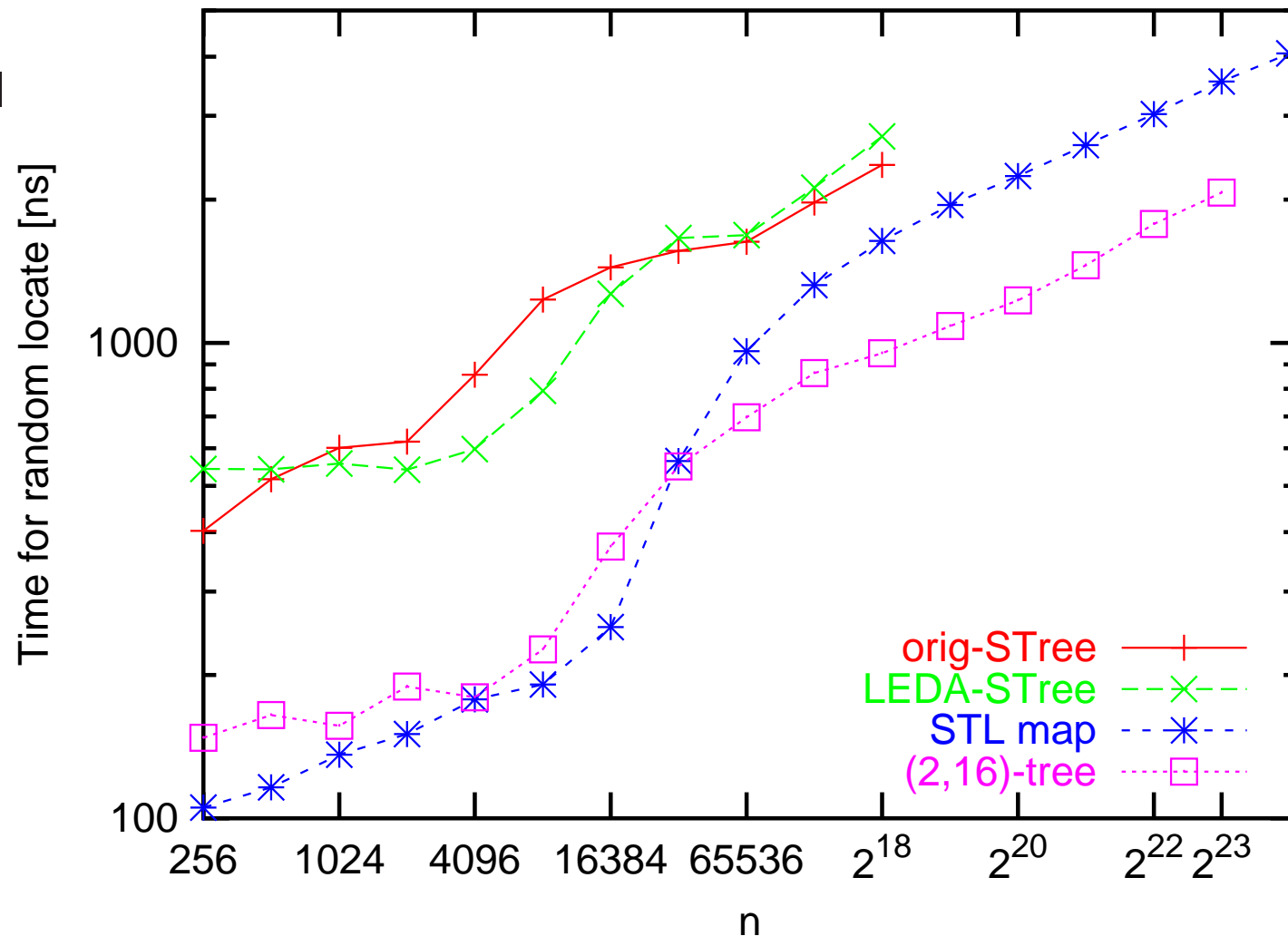
Problems:

Many special

case tests

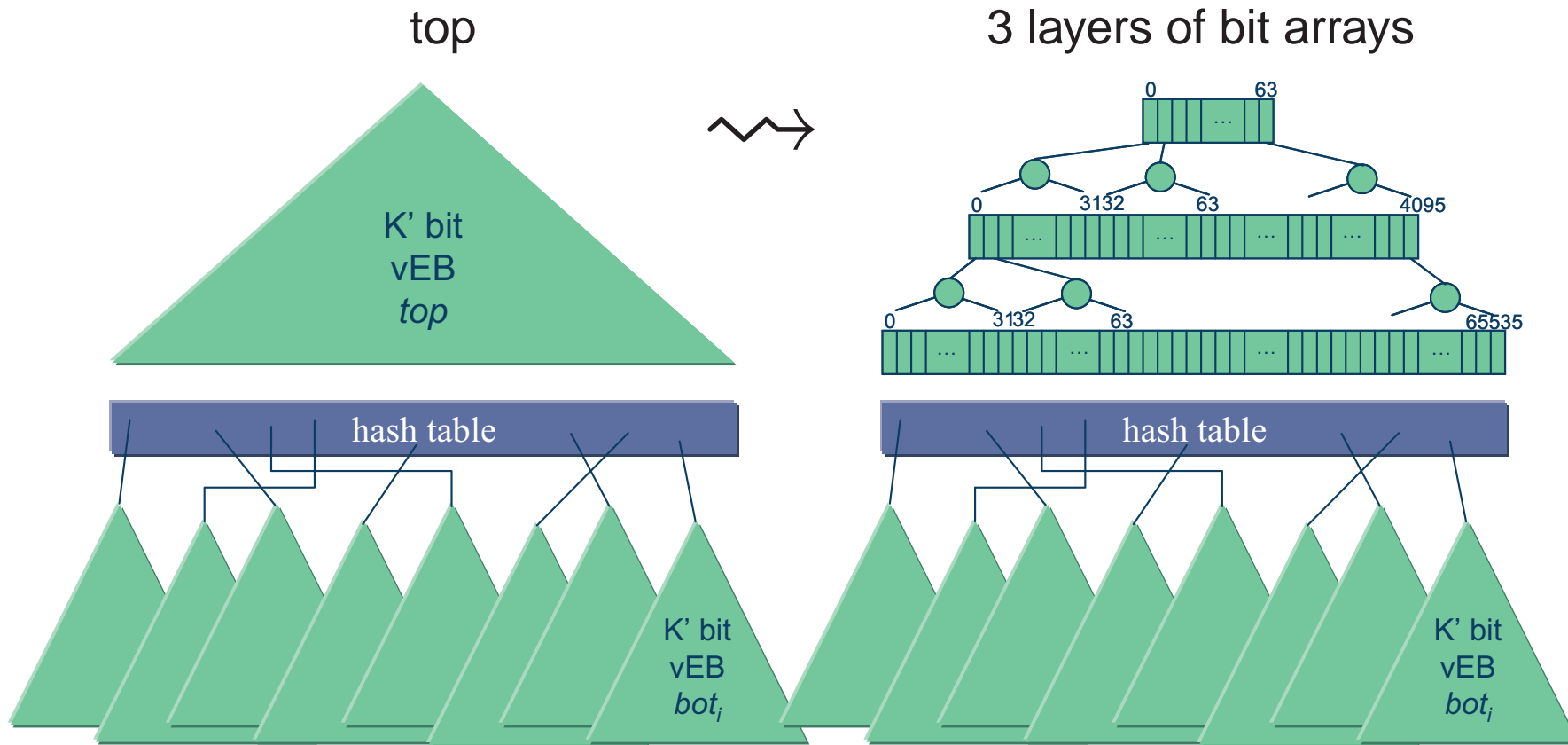
High space

consumption





Efficient 32 bit Implementation





Layers of Bit Arrays

$$t^1[i] = 1 \text{ iff } M_i \neq 0$$

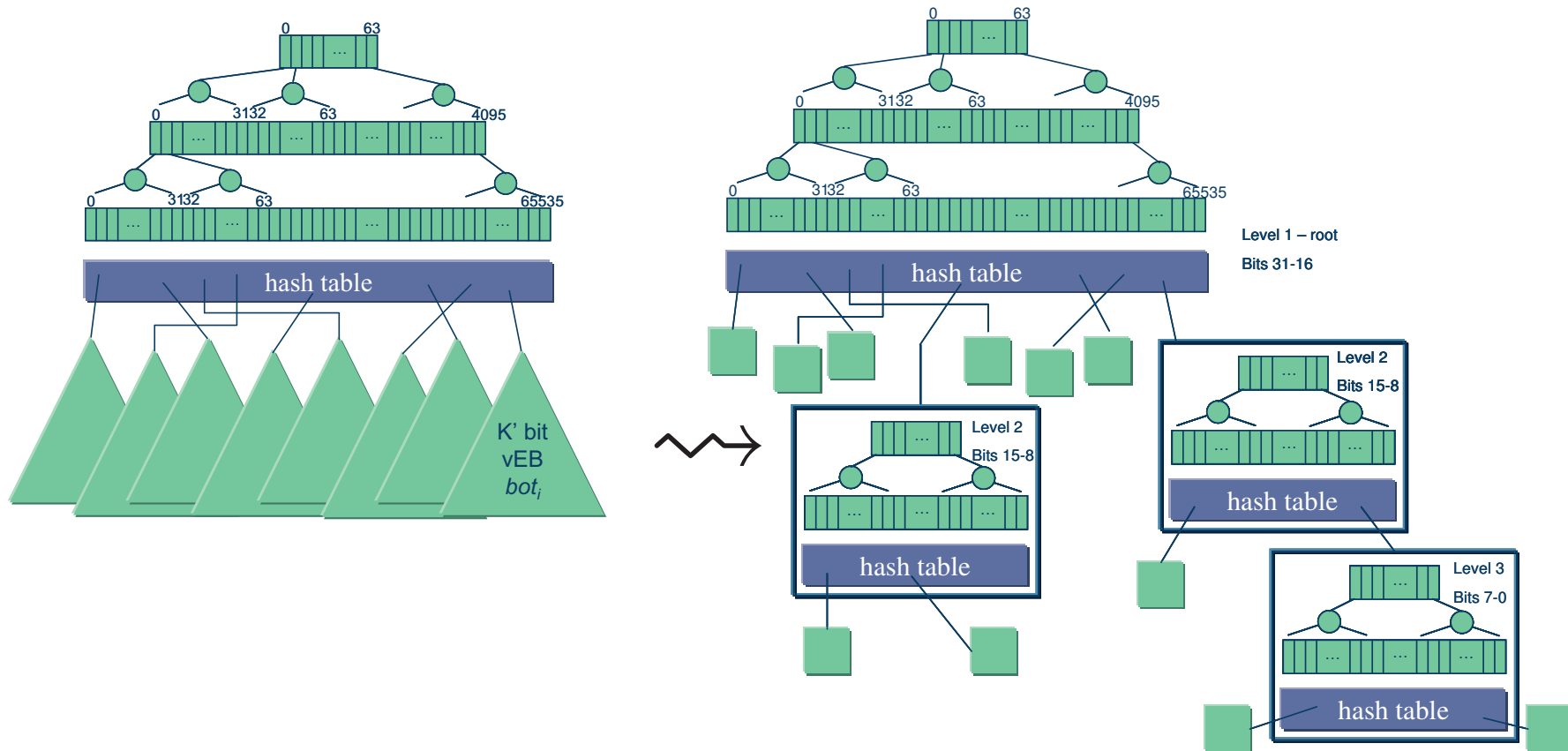
$$t^2[i] = t^1[32i] \vee t^1[32i + 1] \vee \dots \vee t^1[32i + 31]$$

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



Efficient 32 bit Implementation

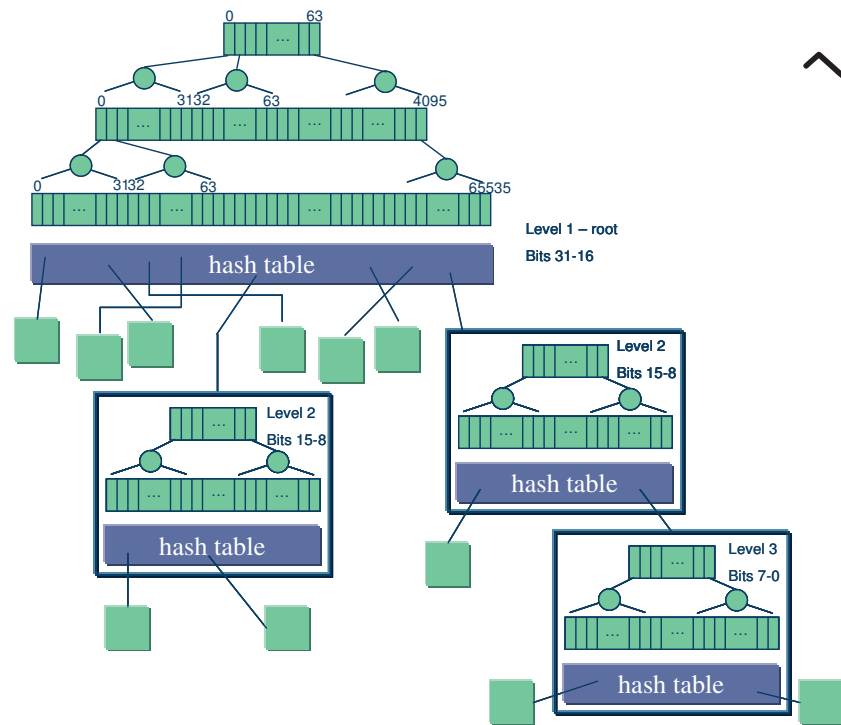
Break recursion after 3 layers



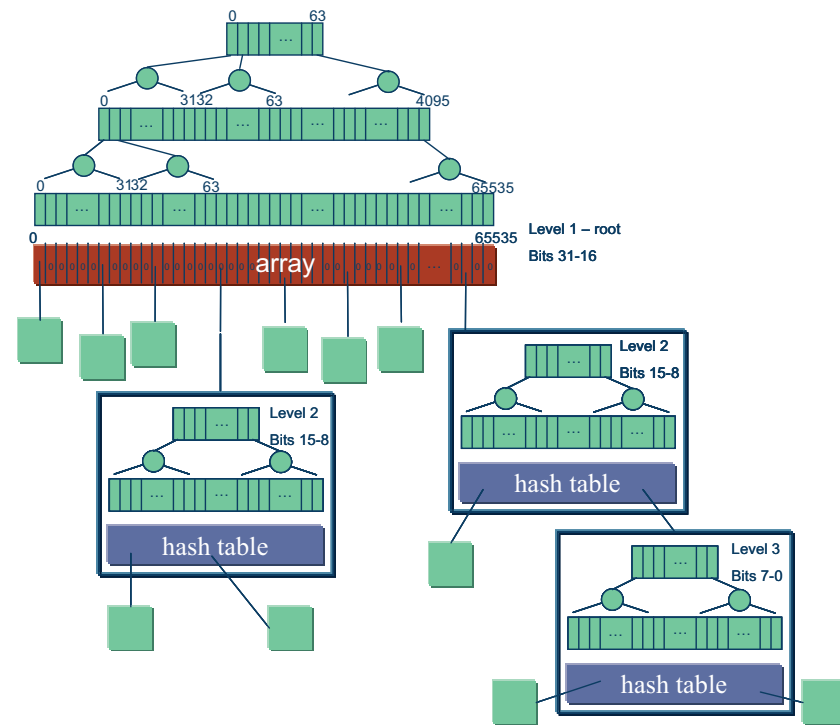


Efficient 32 bit Implementation

root hash table



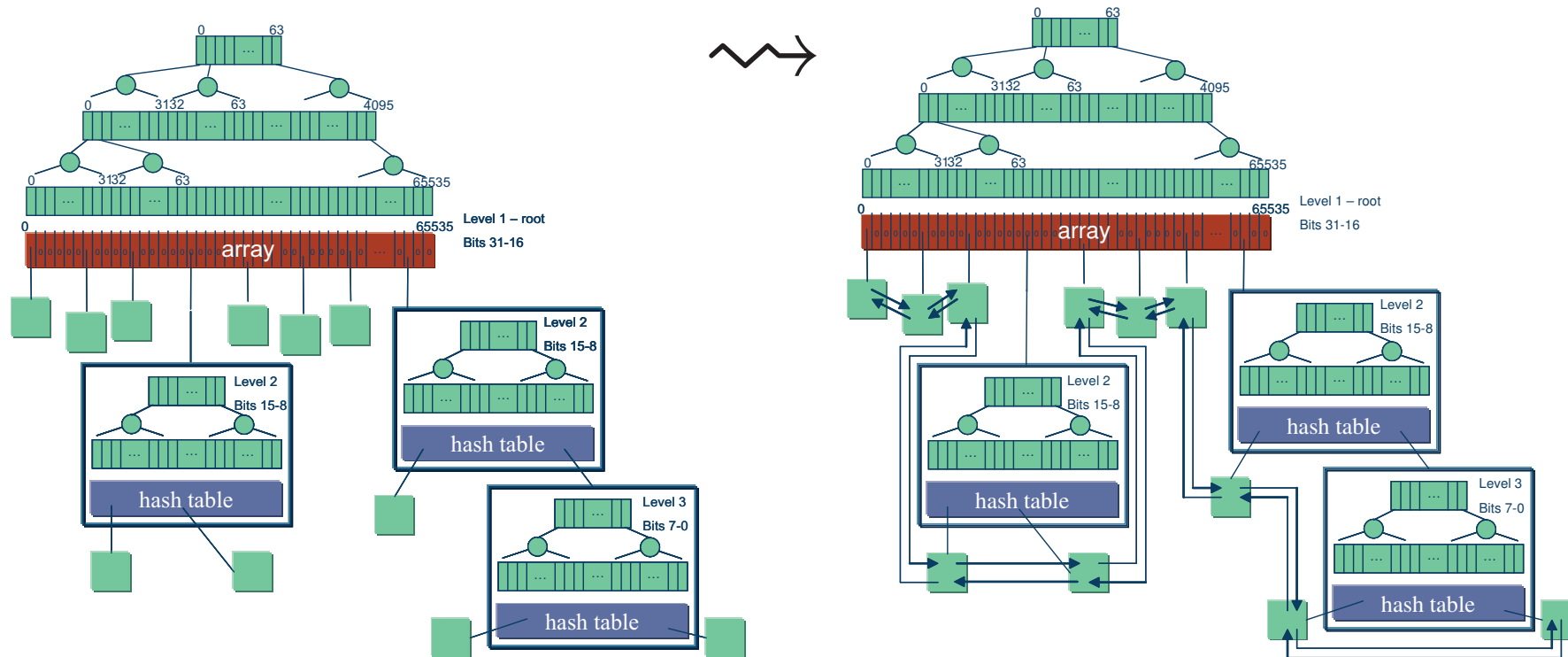
root array





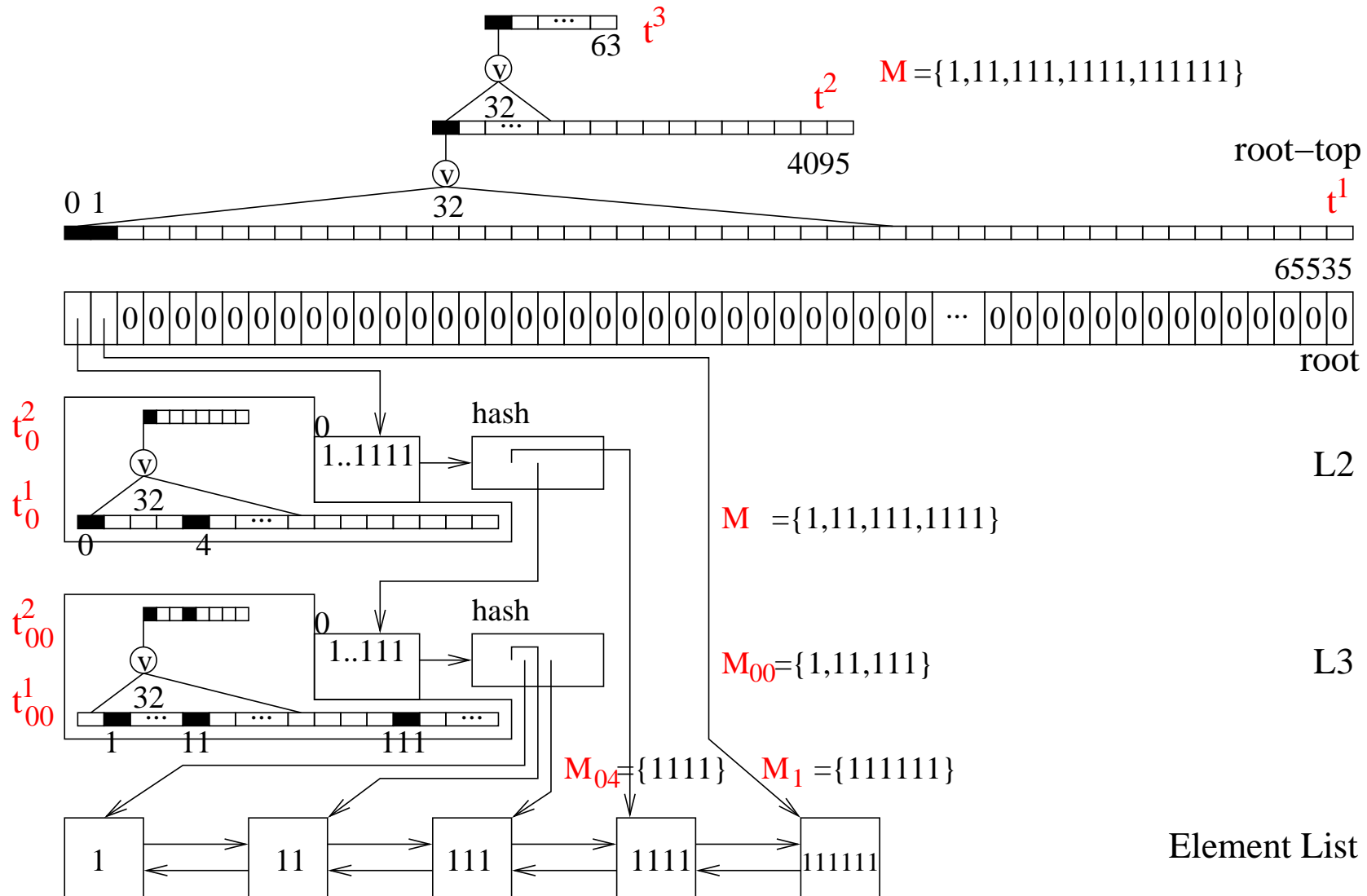
Efficient 32 bit Implementation

Sorted doubly linked lists for **associated information** and **range queries**





Example





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

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

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

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

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

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

$j := i \text{ div } n$ // which word?

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

set $a[(i \text{ mod } n) + 1..n - 1]$ to zero // $n - 1 \dots i \text{ mod } n \dots 0$

if $a = 0$ **then**

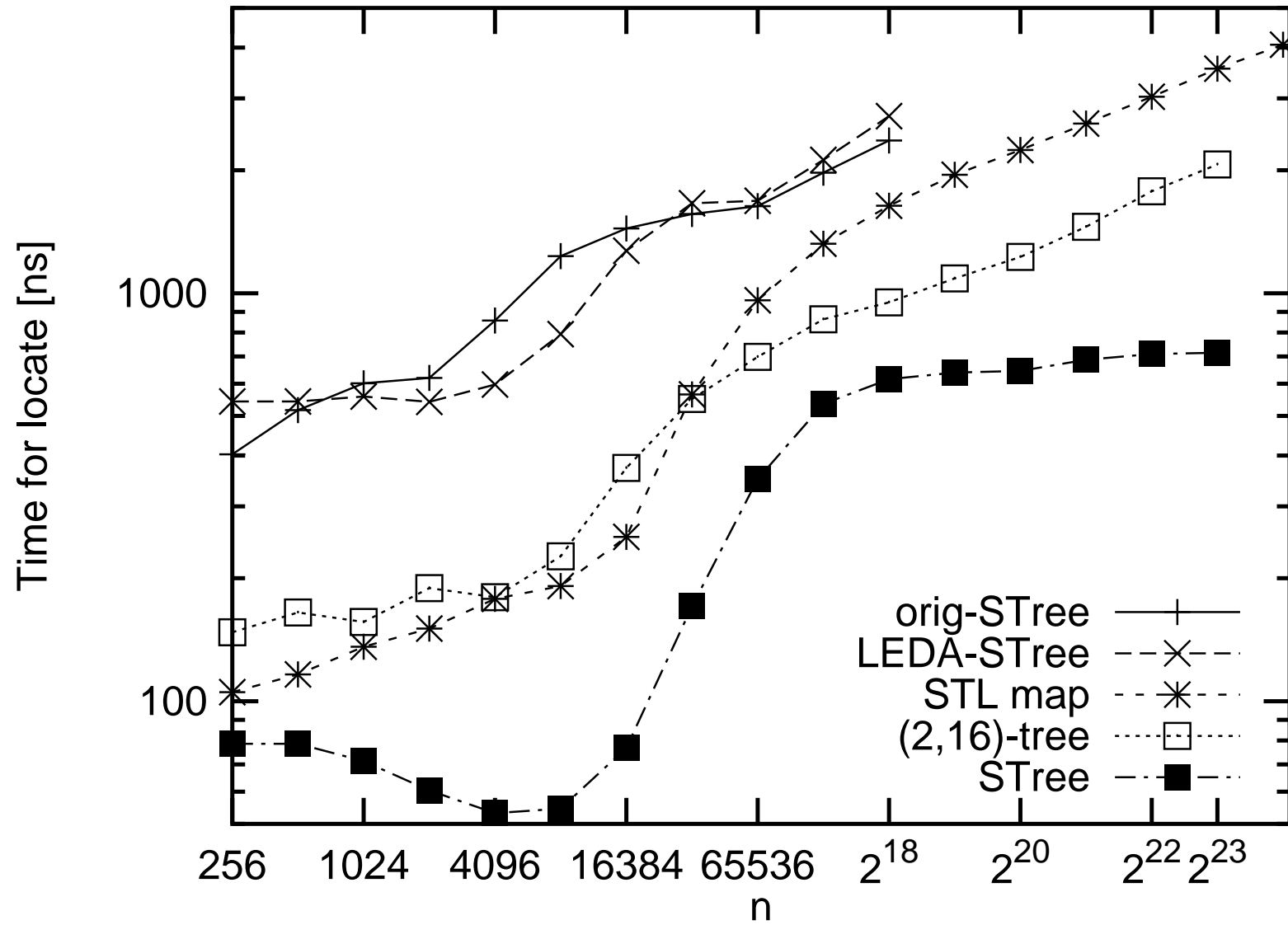
$j := t^{k+1}.\text{locate}(j)$

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

return $nj + \text{msbPos}(a)$ // e.g. floating point conversion

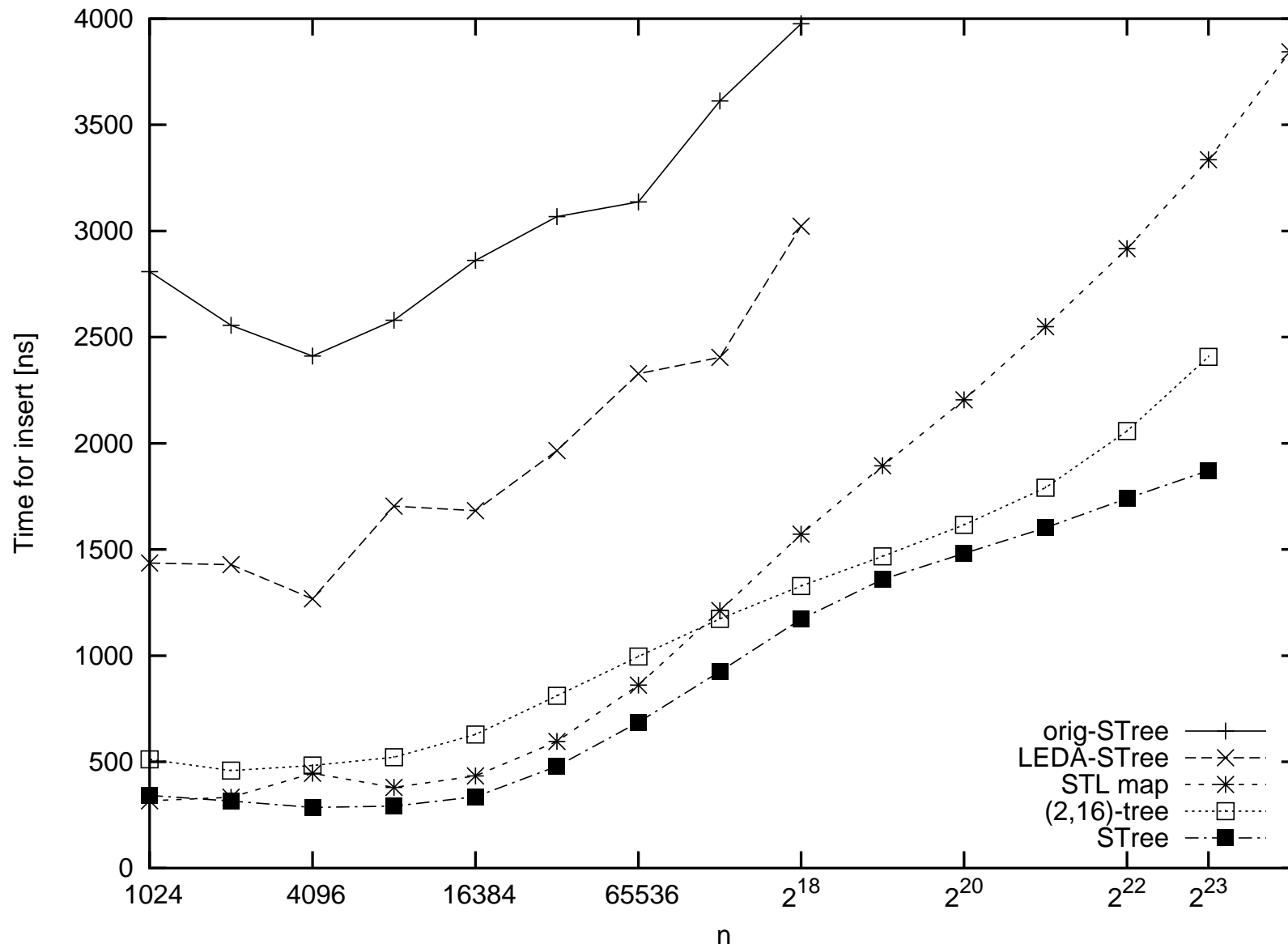


Random Locate



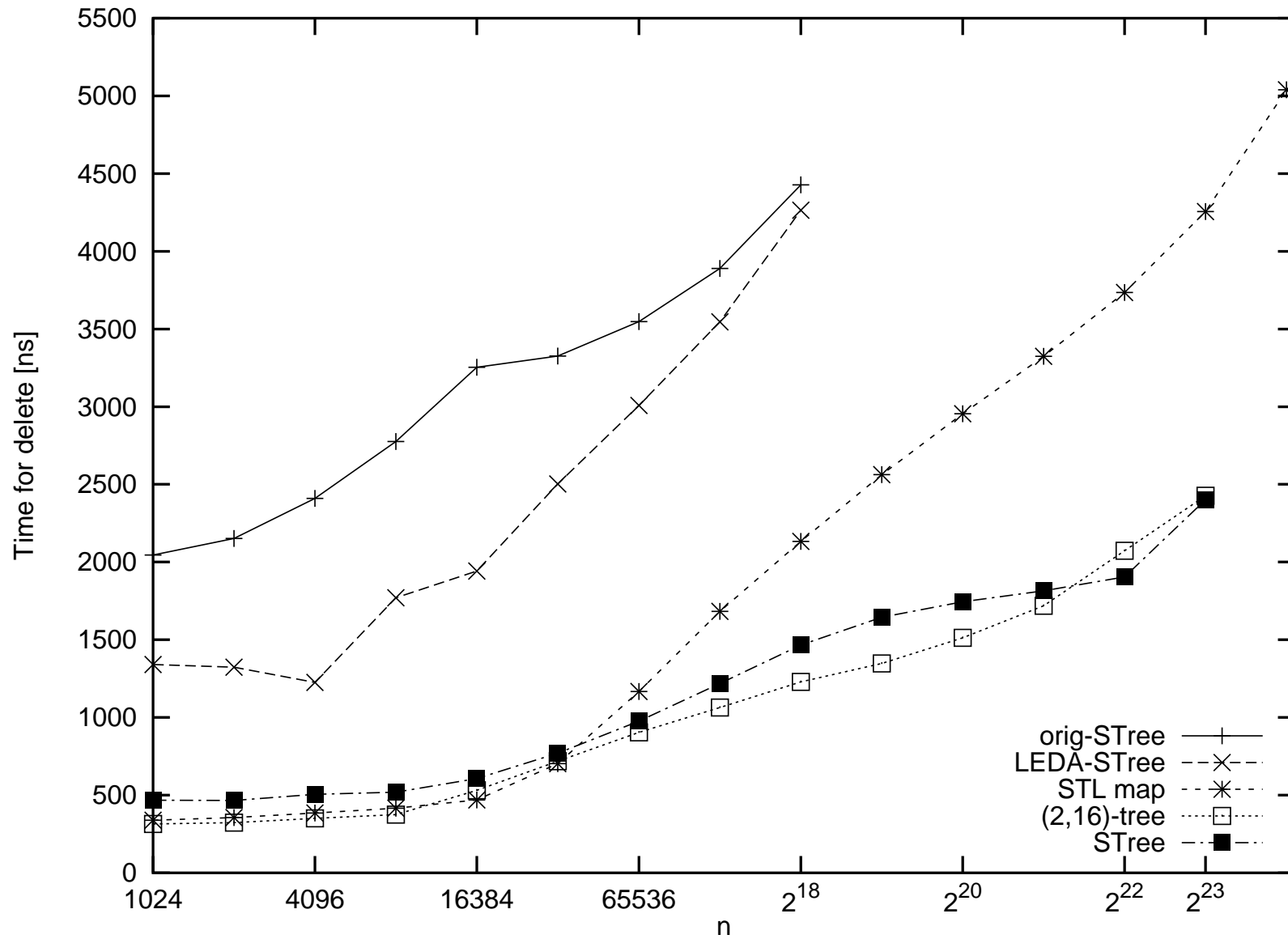


Random Insert





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**



1 Hashing

“to **hash**” \approx “to bring into complete **disorder**”

paradoxically, this helps us to find things more **easily**!

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

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

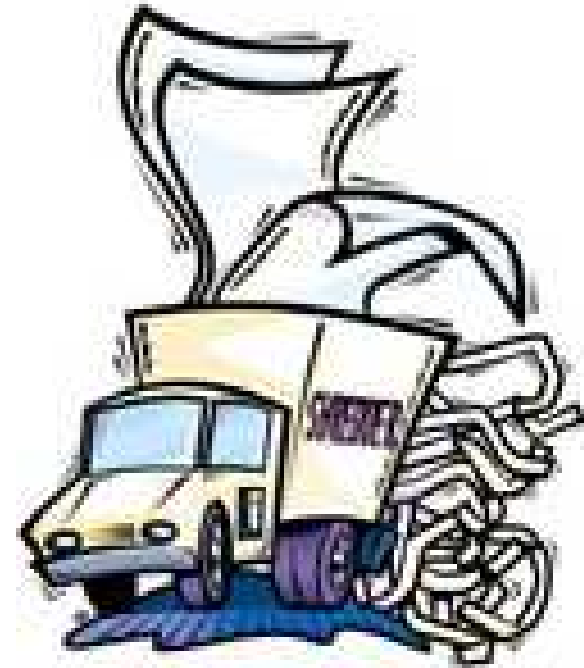
support **dictionary** operations in $O(1)$ time:

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

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

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

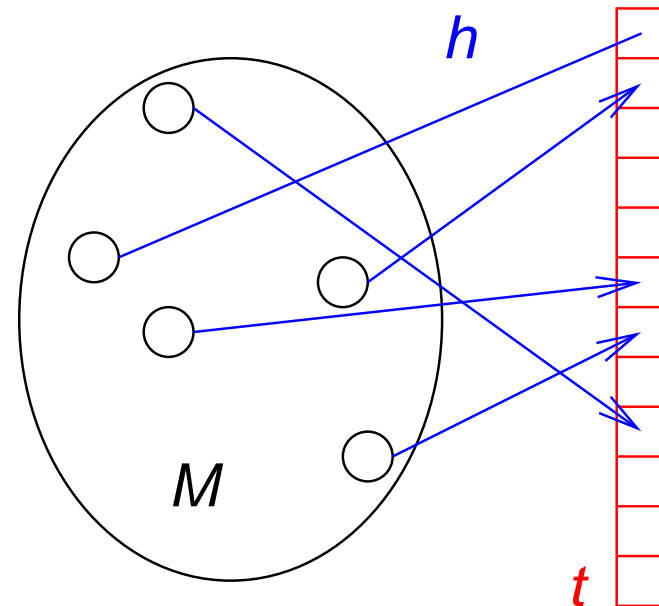
(Convention: key is *implicit*), e.g. $e = k$ iff $\text{key}(e) = k$)





An (Over)optimistic approach

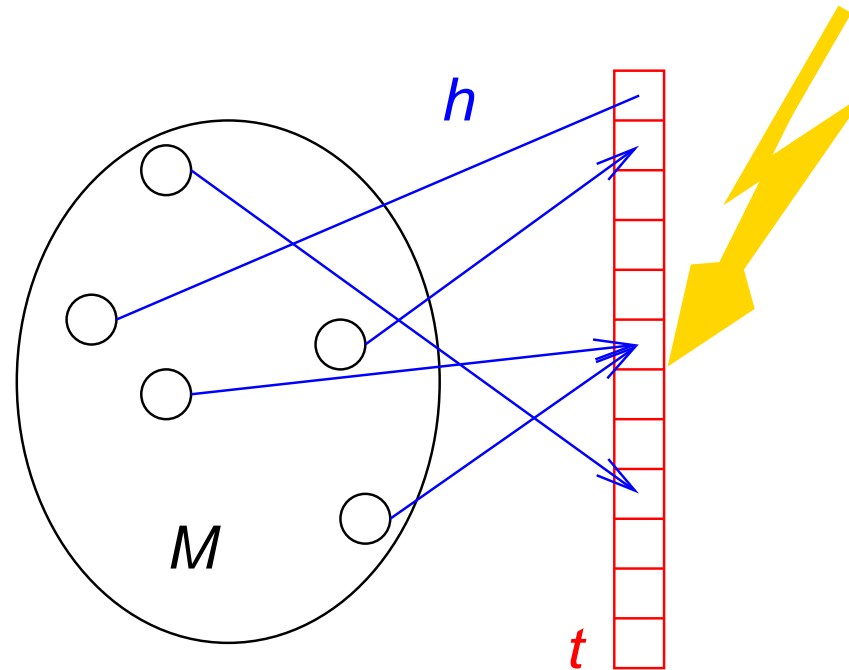
A (perfect) **hash function** h
maps elements of M to
unique entries of **table** $t[0..m-1]$, i.e.,
 $t[h(\text{key}(e))] = e$





Collisions

perfect hash functions are difficult to obtain



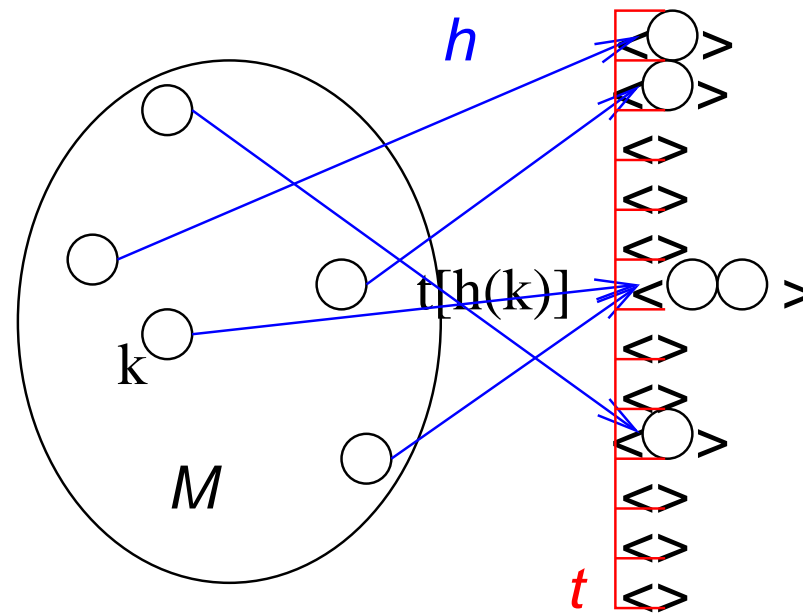
Example: Birthday Paradoxon



Collision Resolution

for example by **closed hashing**

entries: elements \rightsquigarrow **sequences** of elements





Hashing with Chaining

Implement sequences in closed hashing by singly linked lists

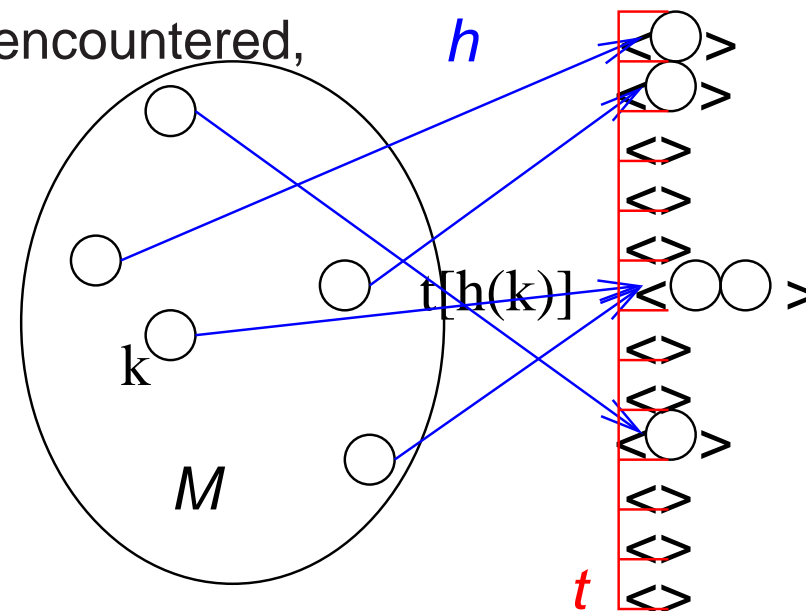
$\text{insert}(e)$: Insert e at the beginning of $t[h(e)]$. **constant time**

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

$\text{find}(k)$: Scan through $t[h(k)]$.

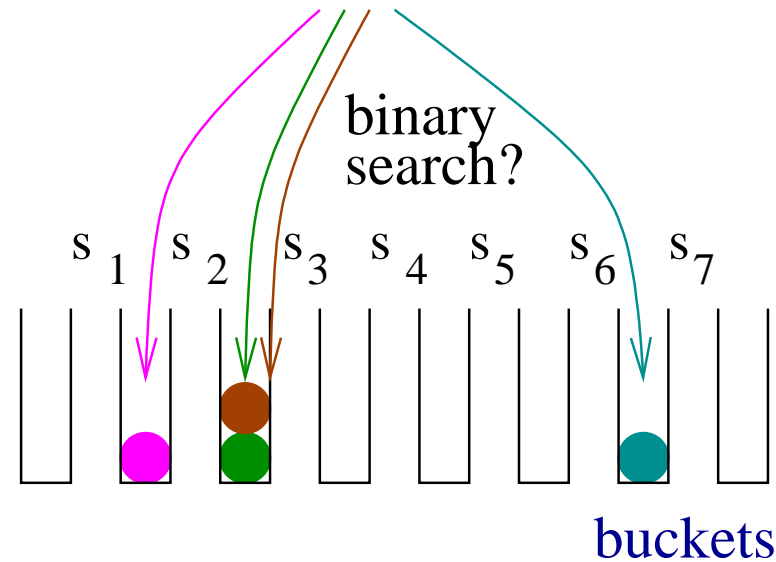
If an element e with $h(e) = k$ is encountered, return it. Otherwise, return \perp .

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





A Review of Probability



Example from hashing

sample space Ω

random hash functions $\{0..m-1\}^{\text{Key}}$

events: subsets of Ω

$$\mathcal{E}_{42} = \{h \in \Omega : h(4) = h(2)\}$$

p_x = probability of $x \in \Omega$

uniform distr. $p_h = m^{-|\text{Key}|}$

$$\mathbb{P}[\mathcal{E}] = \sum_{x \in \mathcal{E}} p_x$$

$$\mathbb{P}[\mathcal{E}_{42}] = \frac{1}{m}$$

random variable $X_0 : \Omega \rightarrow \mathbb{R}$

$$X = |\{e \in M : h(e) = 0\}|.$$



expectation $E[X_0] = \sum_{y \in \Omega} p_y X(y)$

$$E[X] = \frac{|M|}{m} (*)$$

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

Proof of (*):

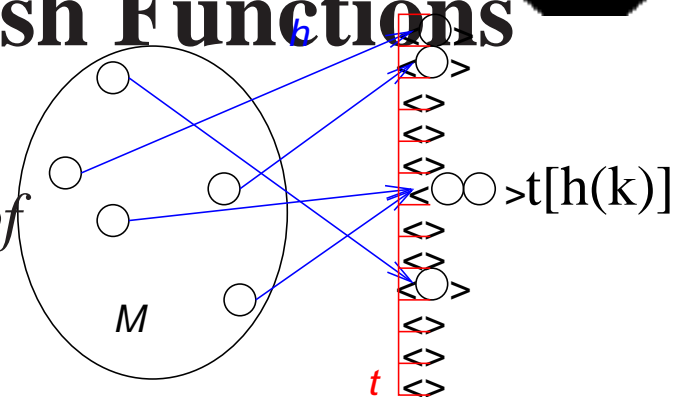
Consider the 0-1 RV $X_e = 1$ if $h(e) = 0$ for $e \in M$ and $X_e = 0$ else.

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



Analysis for Random Hash Functions

Satz 1. The expected *execution time* of $\text{remove}(k)$ and $\text{find}(k)$ 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.

$$\begin{aligned} \mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in M} X_e\right] = \sum_{e \in M} \mathbb{E}[X_e] = \sum_{e \in M} \mathbb{P}[X_e = 1] = \frac{|M|}{m} \\ &= O(1) \end{aligned}$$

This is *independent of the input set* M .





Universal Hashing

Idea: use only certain “easy” hash functions

Definition:

$\mathcal{U} \subseteq \{0..m-1\}^{\text{Key}}$ is *universal*

if for all x, y in Key with $x \neq y$ and random $h \in \mathcal{U}$,

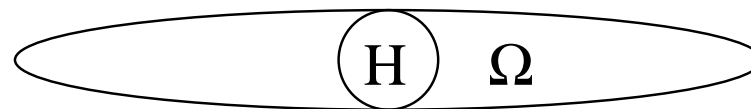
$$\mathbb{P}[h(x) = h(y)] = \frac{1}{m}.$$

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.







A Simple Universal Family

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

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

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

H is a universal family of hash functions

$$\left(\begin{array}{|c|c|c|} \hline x_1 & x_2 & x_3 \\ \hline * & * & * \\ \hline a_1 & a_2 & a_3 \\ \hline \end{array} \right) \text{mod } m = h_{\mathbf{a}}(\mathbf{x})$$



Proof. Consider $\mathbf{x} = (x_1, \dots, x_k)$, $\mathbf{y} = (y_1, \dots, y_k)$ with $x_j \neq y_j$
count \mathbf{a} -s with $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$.

For each choice of a_i s, $i \neq j$, \exists exactly one a_j with
 $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$:

$$\begin{aligned} \sum_{1 \leq i \leq k} a_i x_i &\equiv \sum_{1 \leq i \leq k} a_i y_i \pmod{m} \\ \Leftrightarrow a_j (x_j - y_j) &\equiv \sum_{i \neq j, 1 \leq i \leq k} a_i (y_i - x_i) \pmod{m} \\ \Leftrightarrow a_j &\equiv (x_j - y_j)^{-1} \sum_{i \neq j, 1 \leq i \leq k} a_i (y_i - x_i) \pmod{m} \end{aligned}$$

m^{k-1} ways to choose the a_i with $i \neq j$.

m^k is total number of \mathbf{a} s, i.e.,



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





Bit Based Universal Families

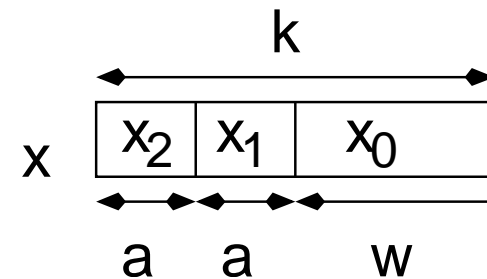
Let $m = 2^w$, Key = $\{0, 1\}^k$

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

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

Table Lookup: $H^{\oplus \square} = \{h_{(t_1, \dots, t_b)}^\oplus : t_i \in \{0..m-1\}^{\{0..w-1\}}\}$

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





Hashing with Linear Probing

Open hashing: go back to original idea.

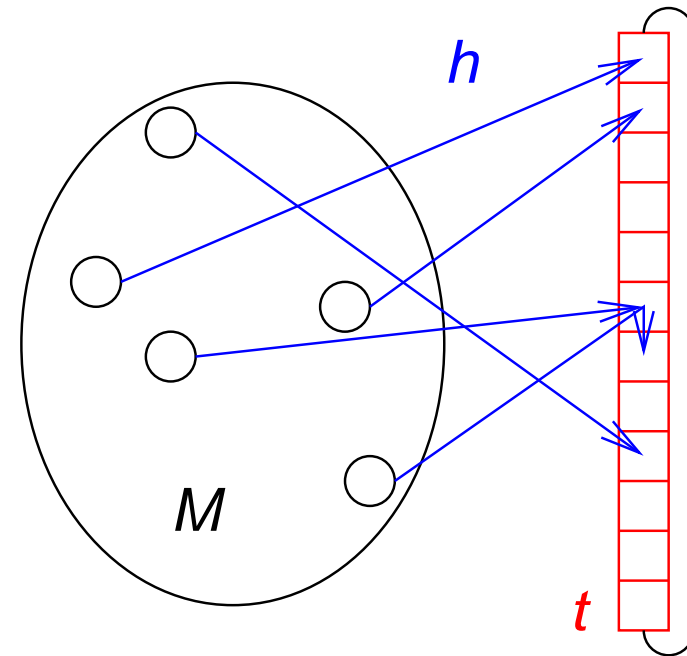
Elements are directly stored in the table.

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($m, m' : \mathbb{N}; h : \text{Key} \rightarrow 0..m - 1$)

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

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

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

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

assert $i < m + m' - 1$

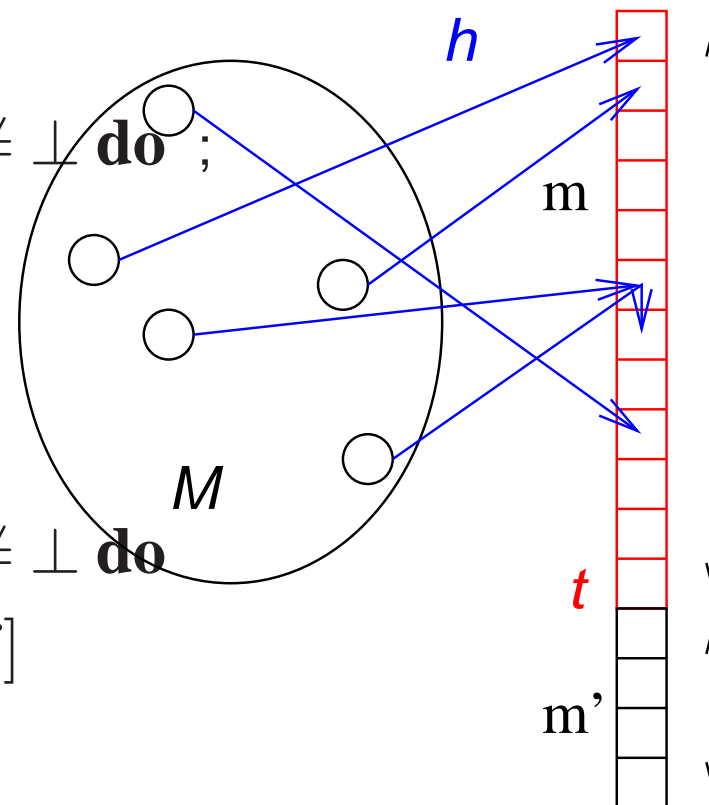
$t[i] := e$

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

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

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

return \perp





Remove

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

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

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

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

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

// we plan for a **hole at i** .

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

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



More Hashing Issues

- High probability and **worst case** guarantees
 - ↪ 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

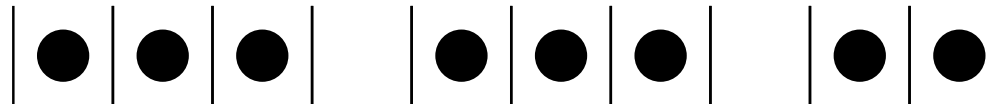


Space Efficient Hashing with Worst Case Constant Access Time

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

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

Assume a truly random hash function h

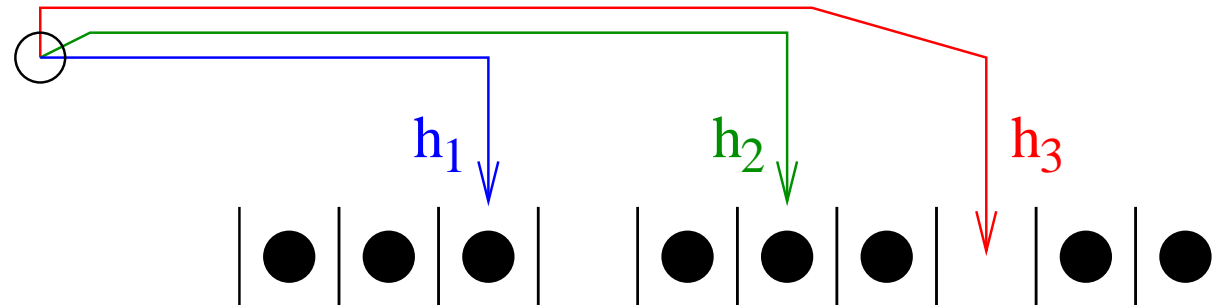




Related Work

Uniform hashing:

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

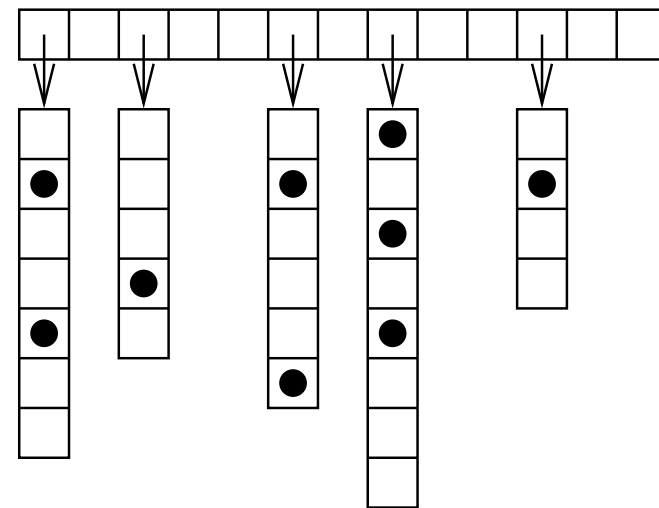


Dynamic Perfect Hashing,

[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.

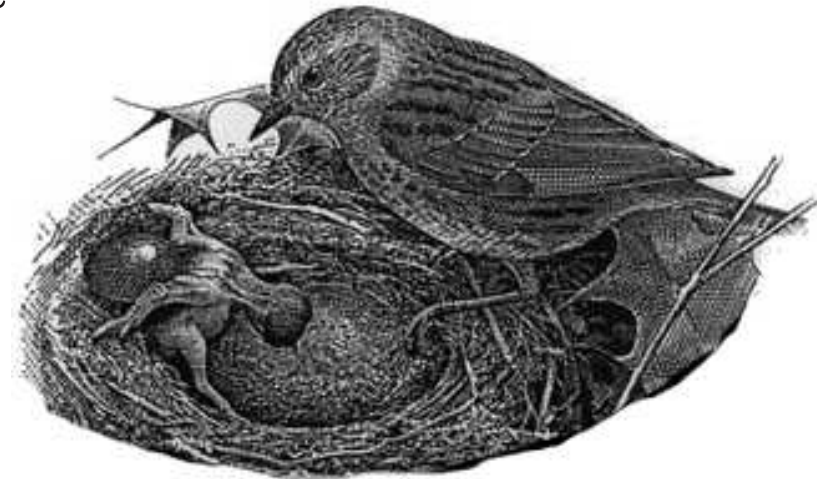


Cuckoo Hashing

[Pagh Rodler 01] Table of size $2 + \epsilon$

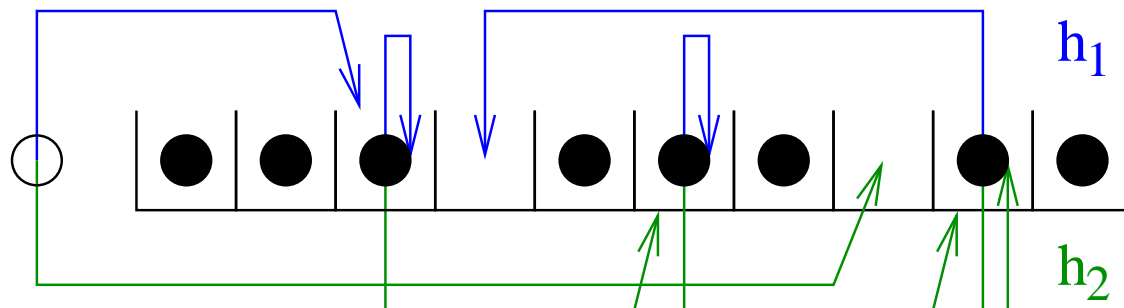
Two choices for each element.

Insert moves elements;
rebuild if necessary.



Very fast lookup and insert.

Expected constant insertion time.





d -ary Cuckoo Hashing

d choices for each element.

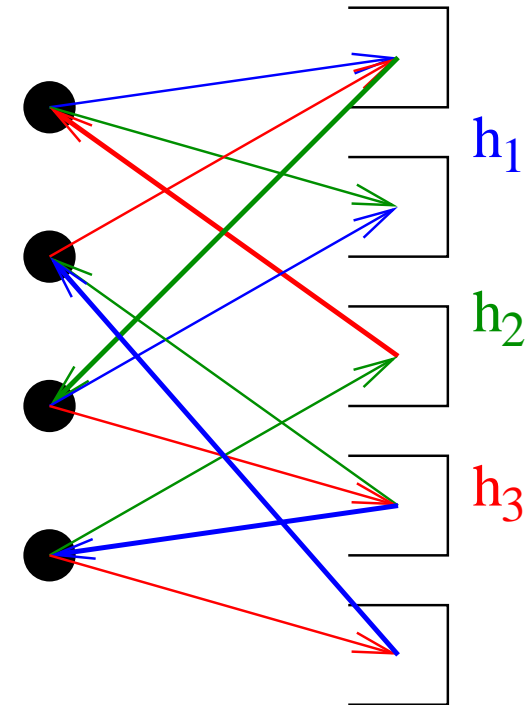
Worst case d probes for **delete** and **lookup**.

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**.





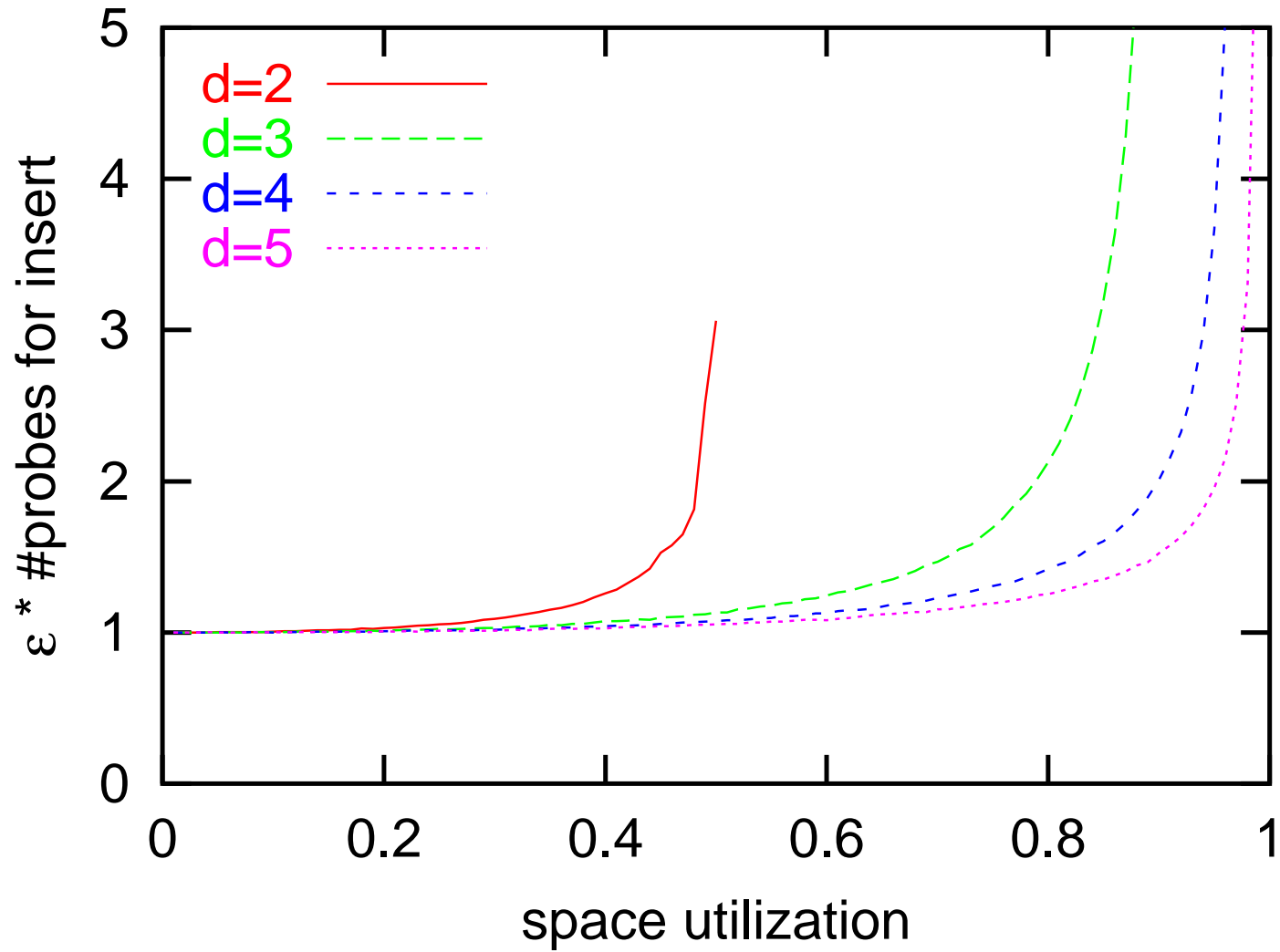
Tradeoff: Space \leftrightarrow Lookup/Deletion Time

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**