Kapitel 8: Externspeicheralgorithmen

8.3 Externe Array-Heaps

VO Algorithm Engineering

Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

23./24. VO 28.06./03.07.2007

Literatur für diese VO

Andreas Crauser: LEDA-SM: External Memory Algorithms and Data Structures in Theory and Praxis. Dissertation, Max-Planck-Institut für Informatik, Saarbrücken, 2001. Kapitel 4: Priority Queues; http://www.mpi-sb.mpg.de/~crauser/degrees.html

- hierzu gibt es auch ein ausgearbeitetes Skriptum auf unserer VO Web-Seite

Umsteiger-Info Bachelor/Master

- Prioritätswarteschlange

- Externe Array-Heaps
  - Idee - Einführung
  - Operationen Insert / Del_Min
  - Theoretische Analyse

Experimenteller Vergleich

Externspeicherdatenstruktur für Prioritätswarteschlangen

- Dynamische Datenstruktur für Elemente: Schlüssel + Information

- **Operationen:**
  - Get_Min: Ausgabe der Elemente mit kleinstem Schlüssel
  - Del_Min: Ausgabe und Entfernung des kleinsten Elements
  - Insert: Einfügen eines neuen Elements

Welche Datenstrukturen kennen Sie dafür?

Externe Array-Heaps

- Im internen Arbeitsspeicher: Heap

- Im externen Speicher: Menge von sortierten Feldern unterschiedlicher Länge

Die Anzahl der Plätze in einem Slot von L_{i+1} entspricht der Anzahl aller Plätze in L_i plus l_i

Lemma 1: $l_{i+1} = l_i\,(\mu+1)$

$l_1 = cM$

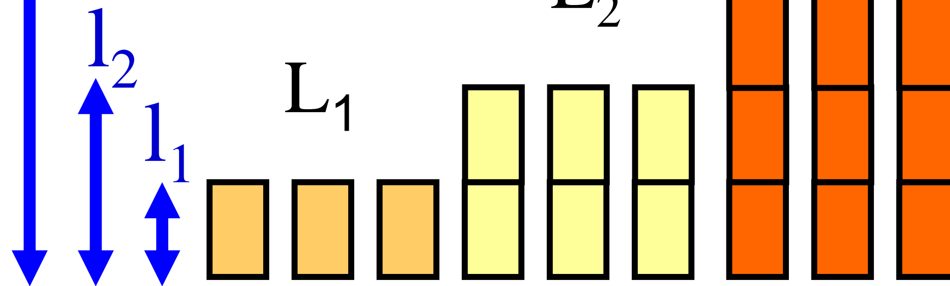
$l_2 = (cM)^2/B$

$= cM(\mu + 1)$

L Schichten L_i L_3

jeder Slot enthält sortierte Folge oder ist leer

$l_i = (cM)^i/B^{i-1}$



$c = 1/7$

$L <= 4$

$\mu = (cM/B) - 1$ μ μ

Operation Insert

- Fügt neue Elemente immer in den internen Heap H ein

- Falls kein Platz mehr in H ist, dann werden vorher $l_1=cM$ dieser Elemente in den Sekundärspeicher bewegt:
  - Falls freier Slot in L_1 existiert, dann werden diese Elemente in sortierter Folge dorthin bewegt
  - Sonst: Alle Elemente in L_1 werden mit den neuen Elementen aus H zu einer sortierten Liste gemischt, die dann in einen freien Slot von L_2 geschrieben werden.
  - Falls L_2 auch kein freier Slot, wiederhole L_3,...bis frei

Operation Del_Min

- **Invariante:** Das kleinste Element befindet sich immer in H

- Dazu: Heap wird in zwei Heaps geteilt: H_1 und H_2:
  - H_1 enthält immer die neu eingefügten Elemente, maximal $2cM$
  - H_2 speichert maximal die kleinsten B Elemente in jedem belegten Slot j in jeder Schicht L_i

- **Lemma 2:** Es befinden sich maximal $cM(2+L)$ Elemente im Hauptspeicher

$2cM+B\mu L < 2cM+B(cM/B)L$

- Zusätzlich wird $(\mu+1)B=cM$ gebraucht, um die μ Slots plus eine Overflow Folge zu mischen

Es muss gelten: $M \geq cM(3+L)$;
Daraus folgt: bei $c=1/7$ => $L \leq 4$

- Invariante: Das kleinste Element befindet sich immer in H

- Dazu: Heap wird in zwei Heaps geteilt: H_1 und H_2:
  - H_1 enthält immer die neu eingefügten Elemente, maximal $2cM$
  - H_2 speichert maximal die kleinsten B Elemente in jedem belegten Slot j in jeder Schicht L_i

- **Lemma 2:** Es befinden sich maximal $cM(2+L)$ Elemente im Hauptspeicher

- Zusätzlich wird $(\mu+1)B=cM$ gebraucht, um die μ Slots plus eine Overflow Folge zu mischen

Operationen (1)

- **Merge-Level (i,S,S´):**
  - produziert eine sortierte Folge S' durch das Mischen der sortierten Folge der µ Slots in L_i (inkl. der ersten Blocks in H_2) und der sortierten Sequenz S.
  - Analyse: $O(l_{i+1}/B)$ I/O´s

- **Store(i;S):**
  - Annahme: L_i enthält einen leeren Slot und die Folge S besitzt Länge im Bereich $[l_i/2\ ,\ l_i]$
  - S wird in einen leeren Slot von L_i gespeichert und seine kleinsten B Elemente werden nach H bewegt.
  - Analyse: $O(l_i/B)$ I/O´s

Operationen (2)

- **Load (i,j):**
  - Holt die nächsten B kleinsten Elemente vom j-ten Slot aus L_i in den internen Heap H_2.
  - Analyse: $O(1)$ I/O´s

- **Compact(i):**
  - Annahme: es existieren mind. 2 Slots in L_i, mit Gesamtzahl an Elementen (inkl. H_2), höchstens l_i.
  - Diese beiden Slots werden gemischt, und in einen freien Slot von L_i eingetragen. Damit wird ein Slot in L_i frei.
  - Analyse: $O(l_i/B)$ I/O´s

Operation Insert

- Fügt neue Elemente immer in den internen Heap H ein

- Falls kein Platz mehr in H_1 ist, dann werden die größten $l_1=cM$ Elemente nach L_1 bewegt (und die kleinsten B davon nach H_2):
  - Falls freier Slot in L_1 existiert, dann wird $\mathrm{Store}(1,S)$ aufgerufen
  - Sonst: Alle Slots in L_1 (bis auf einen) enthalten mindestens $l_1/2$ Elemente: $\mathrm{Merge\text{-}Level}(1,S,S')$
  - Falls freier Slot in L_2 existiert, dann: $\mathrm{Store}(2,S')$
  - Sonst: wiederhole L_3,...bis frei.

Operation Del_Min

- Das kleinste Element wird vom internen Heap entfernt (H_1 oder H_2).

- Falls in H_2: dann korrespondiert dieses zu Slot j einer Schicht L_i.
- Falls es das letzte Element in H_2 ist, das zu j gehört, dann werden die nächsten B Elemente von Slot j nach H_2 mittels Load(i,j) bewegt.
- Nach jedem Load(i,j) wird Compact(i) bei Bedarf aufgerufen

Korrektheit

Lemma 3: Das kleinste Element ist immer in H (H_1 oder H_2).

Lemma 4: Bei der Ausführung von $Store(i,S)$ ist immer garantiert, dass S zwischen $l_i/2$ und l_i Elementen enthält.

Beweis: Betrachte Schicht i-1: Sei a_j die Anzahl der Elemente in Slot j. Wir wissen: $a_j+a_k>l_{i-1}$ für alle Paare j und k
→ Summe über alle Paare:

$$(\mu-1)\sum_{j=1}^{\mu} a_j = \sum_{\substack{j,k=1\\ j\neq k}}^{\mu} (a_j + a_k) > \frac{\mu(\mu-1)}{2} l_{i-1}$$

Hinzu kommt das vorige S mit mindestens $l_{i-1}/2$ Elementen.
Dies sind also zusammen mind. $(\mu+1)\, l_{i-1}/2 = l_i/2$ Elemente.

I/O Schranken

Annahme: cM>3B

Lemma 5: Nach N Operationen existieren höchstens $L \leq \log_{cM/B}(N/B)+1$ Schichten.

Beweis: Im Worst Case sind alle N Operationen Inserts. Dann wird bei jedem Überlauf die maximal mögliche Anzahl von Elementen in die nächste Schicht bewegt. Die Anzahl der Elemente in j Slot-Schichten ist also:

o.E. ohne Elem. in H

$$\sum_{i=1}^{j} l_i \mu = \sum_{i=1}^{j} \frac{(cM)^i}{B^{i-1}} \mu = \sum_{i=1}^{j} \frac{(cM)^i}{B^{i-1}} \left(\frac{cM}{B} - 1\right) =$$

$$= \sum_{i=1}^{j} \frac{(cM)^{i+1}}{B^i} - \sum_{i=1}^{j} \frac{(cM)^i}{B^{i-1}} = \frac{(cM)^{j+1}}{B^j} - cM < \frac{(cM)^{j+1}}{B^j}$$

Wir müssen kleinstes j wählen, so dass $cM(cM/B)^j \geq N$. Auflösen nach $j \Rightarrow j \geq \log_{cM/B}(N/cM)$. Damit ist $L \leq \log_{cM/B}(N/cM) + 1$. Es gilt $N/(cM) < N/(3B) < N/B$

$j = \lceil \log \ldots \rceil$

I/O Schranken

Annahme: $cM>3B$

Lemma 6: Store(i,S) benötigt höchstens $3l_i/B$ I/O´s.
Merge-Level(i,S,S´) und Compact(i+1) benötigen höchstens $3l_{i+1}/B$ I/O´s.

Beweis:

Store(i,S): r+w: $2\lceil l_i/B\rceil \leq 2(l_i/B)+2 \leq 3l_i/B$ (wg. $li/B>3$ da $l_i \geq l_1 = cM > 3B$)

Merge-Level(i,S,S´): r+w: $2\lceil (|S|+\mu l_i)/B\rceil \leq 2\lceil l_{i+1}/B\rceil \leq 3l_{i+1}/B$

Compact(i): $\lceil a_j/B\rceil + \lceil a_k/B\rceil + \lceil l_i/B\rceil \leq (a_j+a_k)/B + l_i/B + 3 \leq 2l_i/B+3 \leq 3l_i/B$, wg. $l_i/B>3$

I/O Schranken

Theorem:

Annahme: $cM>3B$ und $0<c<1/3$ und $N \leq B(cM/B)^{(1/c)-3}$

In einer Folge von N Operationen vom Typ Insert und Del_Min benötigt

- Insert amortisiert $18/B (\log_{cM/B}(N/B)+1)$ I/O´s und
- Del_Min $7/B$ amortisierte I/O´s.

Die Schranke für N kommt aus Platzbeschränkungen her: der Heap benötigt $cM(3+\log_{cM/B}(N/B))$ internen Speicherplatz (s. folgendes Theorem).

Beweis: Amortisierte Analyse (1)

- Insert: $18/B(\log_{cM/B}(N/B)+1) \geq 18L/B$ amortisierte I/O´s
- Del_Min $7/B$ amortisierte I/O´s.

- Bankkonto-Methode:
  - Jedes Element erhält beim Einfügen ein Guthaben von $18L/B$
  - Wir zeigen: es werden höchstens $18/B$ benötigt um von einer zur anderen Schicht zu wandern
  - Beim Entfernen werden $7/B$ Einheiten im Heap belassen

Beweis: Amortisierte Analyse (2)

- Insert mit Overflow kostet $6l_{i+1}/B$, denn:
  - Merge_level(i,S;S´): kostet $3l_{i+1}/B$ und Store(i+1,S´): $3l_{i+1}/B$
- Wie können diese Kosten bezahlt werden?

- **Fall 1:** Overflow zur Schicht L_1:
- Jedes Element, das nun bewegt wird, gibt von seinem Bankkonto jeweils $12/B$ Einheiten dafür ab; da der Slot in Schicht L_1 mindestens zur Hälfte gefüllt ist, kommen so mind. $(12/B)\ (l_1/2)=6l_1 / B$ Einheiten zusammen.

- (Interpretation: stellen Sie sich vor, jedes Element erhält beim Einfügen $18L/B$ Einheiten; nun möchten die Elemente in die Schicht L_1 wechseln, das kostet aber insgesamt $6l_1/B$. Diese können dadurch aufgebracht werden, indem alle bewegten Elemente jeweils $12/B$ Einheiten von ihrem momentanen Bankkonto abgeben.)

Beweis: Amortisierte Analyse (2)

- Insert mit Overflow kostet $6l_{i+1}/B$, denn:
  - Merge_level(i,S;S´): kostet $3l_{i+1}/B$ und Store(i+1,S´): $3l_{i+1}/B$
- Wie können diese Kosten bezahlt werden?

- **Fall 2:** Overflow von Schicht L_i nach L_{i+1}:
- Jedes Element hatte anfangs $18L/B$ Einheiten zur Verfügung; das sind für Schicht i genau $18/B$ Einheiten, die das Element verbrauchen kann. Da fast alle Slots von Schicht L_i mind. zur Hälfte gefüllt sind, können $12/B$ Einheiten von den Bankkonten der bewegten Elemente genommen werden plus $6/B\ l_i$ Einheiten des evtl. leeren Slots (s. Deposits D_{ij}).

- **Beob.:** Damit hat jedes Element nach der Merge_level() und Store()-Operation noch $6/B$ Einheiten pro Schicht übrig.

Beweis: Amortisierte Analyse (2)

- **Invariante:** Zu jedem nicht-leeren Slot j der Schicht L_i gehört ein Deposit $D_{i,j}$ von $6x/B$, wobei x die Anzahl der freien Felder in j entspricht.

- Das heisst: Um die Invariante zu erfüllen, muss jedes durch den Store() Aufruf nach Schicht L_{i+1} bewegte Element $6/B$ Einheiten an $D_{i+1,j}$ abgeben

- Insgesamt: kostet also eine Merge_Level() und eine Store() Operation pro Overflow (Schicht) $18/B$ Einheiten per Element.

Beweis: Amortisierte Analyse (3)

- Beim Entfernen werden $7/B=(1+6)/B$ Einh. im Heap belassen

- Eine Load()-Operation wird durch das Nehmen von $B(1/B)=1$ Einheiten aus dem Heap bezahlt.
- Diese Einheiten kamen jeweils durch die letzten (aus Slot j) B entfernten Elemente zustande.
- Die restlichen $B(6/B)=6$ Einheiten dieser entfernten Elemente werden dem $D_{i,j}$ zugeordnet, auf dem Load() operiert hat (denn danach sind es dort B Einheiten weniger, es werden also $6*B/B=6$ Einheiten mehr in $D_{i,j}$ benötigt).

- Insgesamt: sind das $7/B$ Einheiten für die Load() Operation

Beweis: Amortisierte Analyse (4)

- Es bleibt: die Bezahlung für Compact(i): $3l_i/B$

- Dies wird durch die Deposits $D_{i,j}$ an den slots $j1$ und $j2$, die kompaktiert werden, bezahlt:
- Die Gesamtanzahl der leeren Plätze in den slots $j1$ und $j2$ ist mindestens l_i.
- Dafür gibt es in den Deposits $D_{i,j1}$ und $D_{i,j2}$ zusammen mindestens $6l_i/B$ Einheiten.
- Nach dem Mischen gibt es in $D_{i,j1}$ höchstens $l_i/2$ freie Slots, d.h. für das neue $D_{i,j1}$ werden nur $3l_i/B$ Einheiten benötigt
- Die anderen $3l_i/B$ Einheiten werden für Compact(i) ausgegeben.

Speicherplatzbedarf

Lemma 7: Jede Schicht enthält höchstens einen Slot, der nicht-leer und aus weniger als $l_i/2$ Elementen besitzt.

Speicherplatzbedarf

Theorem 2: Die Gesamtanzahl der benützten externen Blöcke ist höchstens $2(X/B)+L$, wobei X die Anzahl der Elemente in unserer Datenstruktur ist.

Der Gesamtspeicherplatz im MM beträgt $cM(3+L)$.

Beweis:

- In jedem Slot jeder Schicht existiert höchstens ein nur teilweise gefüllter Speicherplatz, nämlich der oberste.
- Pro Schicht existiert höchstens ein nicht-leerer Slot mit weniger als $l_i/2$ Elementen. Von diesen gibt es zusammen höchstens L.
- Für die anderen Slots gilt: für jeden halb gefüllten gibt es mindestens auch einen ganz gefüllten Block: $\leq 2X/B$

Experimentelles Setup

8 verschiedene PQ Implementierungen:

- **extern:** Externe Array-Heaps, externe Radix Heaps (Achtung: nur für monotone DEL-Folgen und integers einsetzbar), Buffer Trees, B-trees
- **intern:** Fibonacci Heaps, k-ary Heaps, Pairing Heaps, interne Radix Heaps

SPARC ULTRA 1/143:

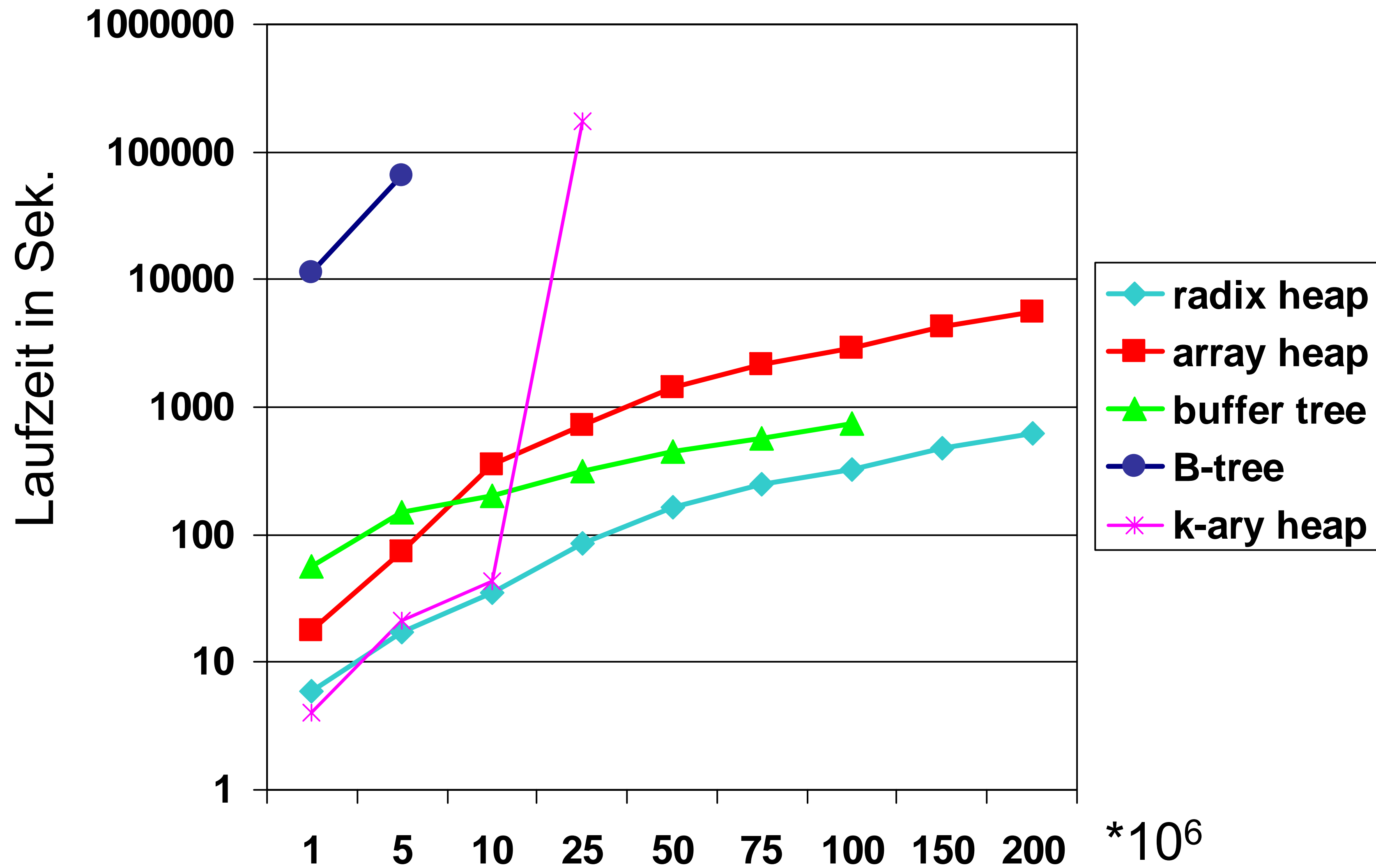
- MM: 256 Mbytes (nur $M=16$ Mbytes genutzt)
- lokale 9 GBytes fast-wide SCSI disk
- $B=32$ kbytes

Experimentelles Setup

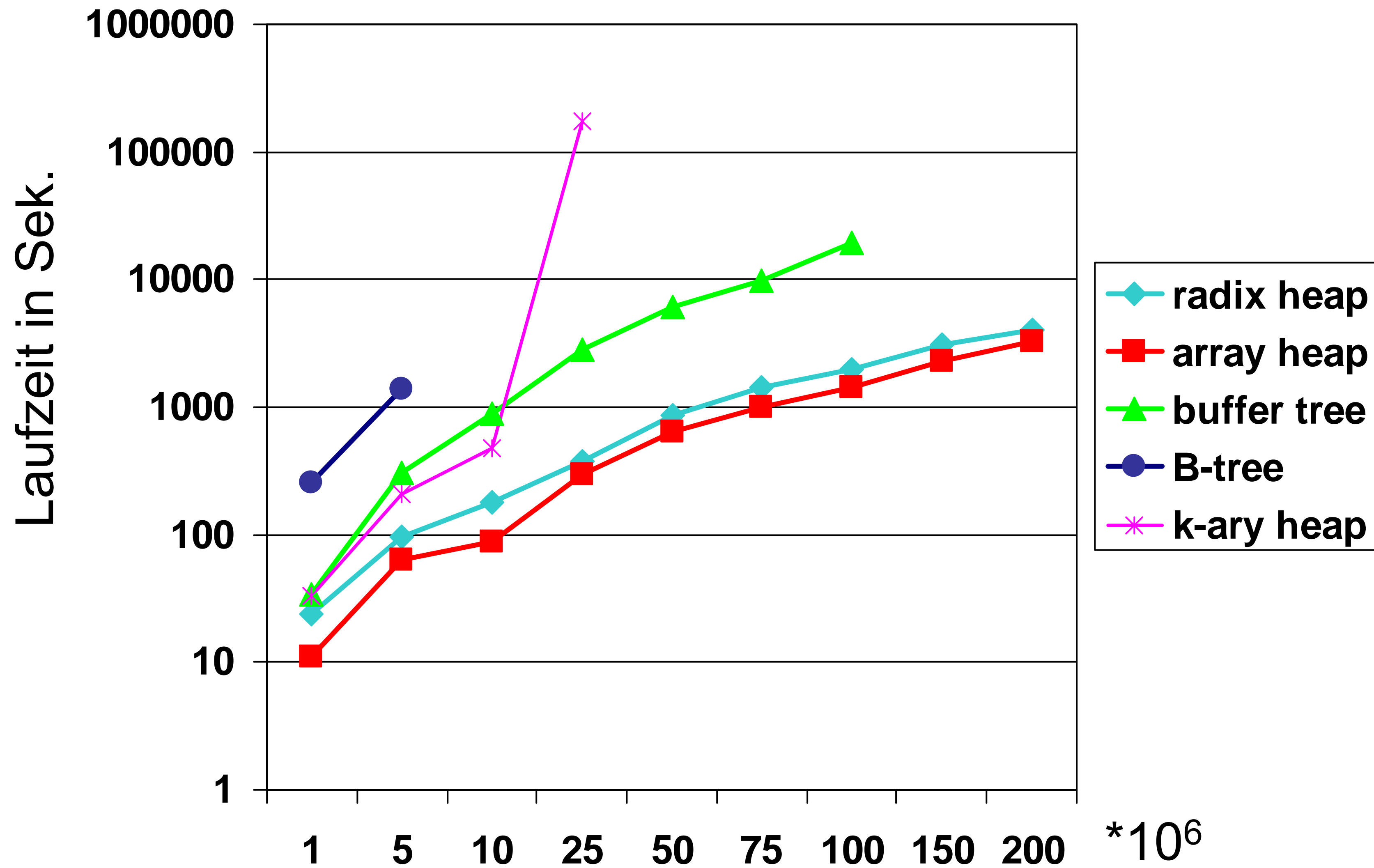
Experimentreihen:

1. **Insert-All-Delete-All:** zunächst N Insert, dann N Del_Min
2. **Intermixed:** zunächst N=20 Mio Insert, dann gemischte Insert mit prob=1/3 und Del_Min Operationen mit prob=2/3
3. **Dijkstra´s shortest-path**: simuliere Dijkstra in MM für große Graphen und teste die hierbei produzierte Sequenz von Insert und Del_Mins.

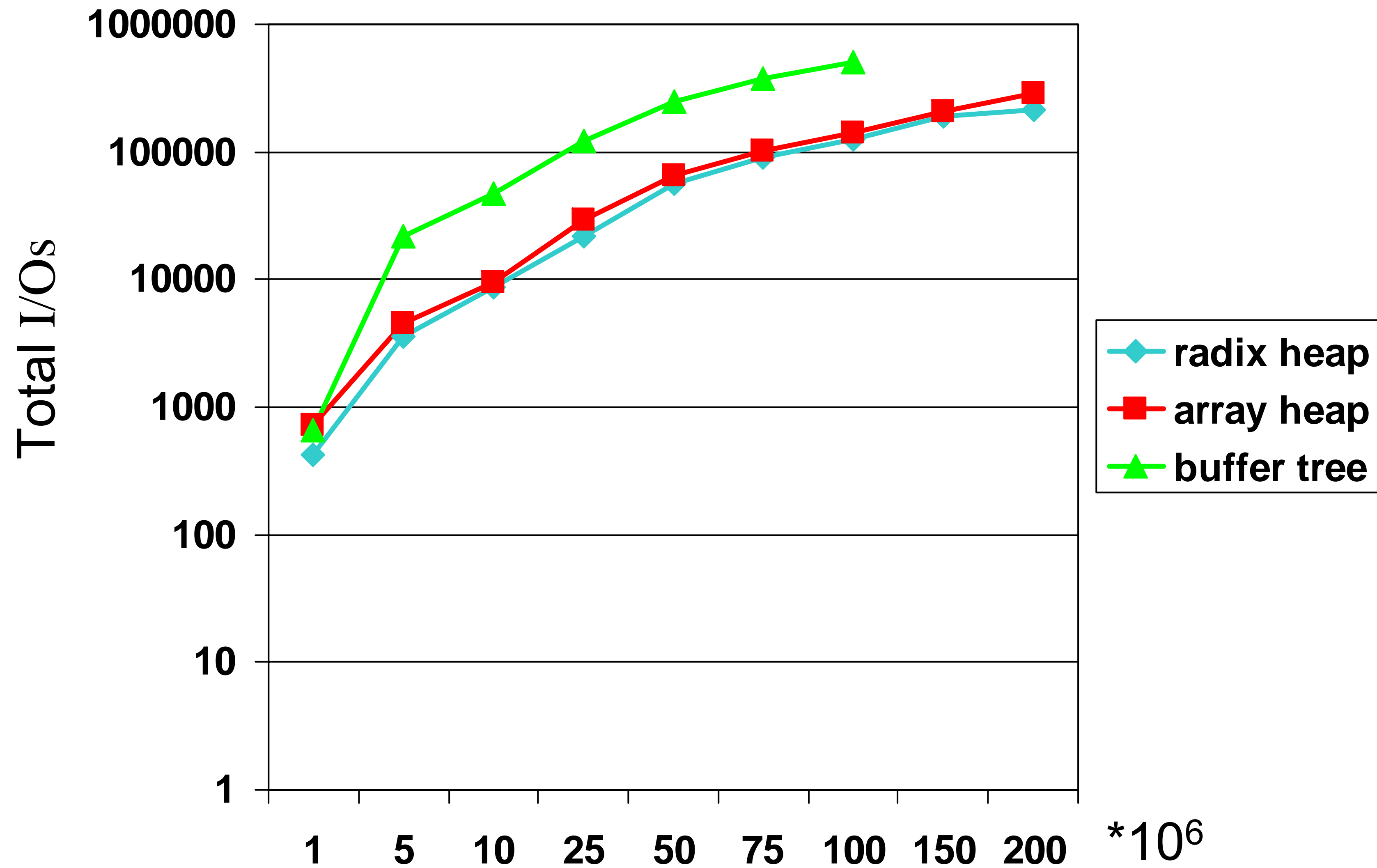
Laufzeit Insert-All-Delete-All: Insert



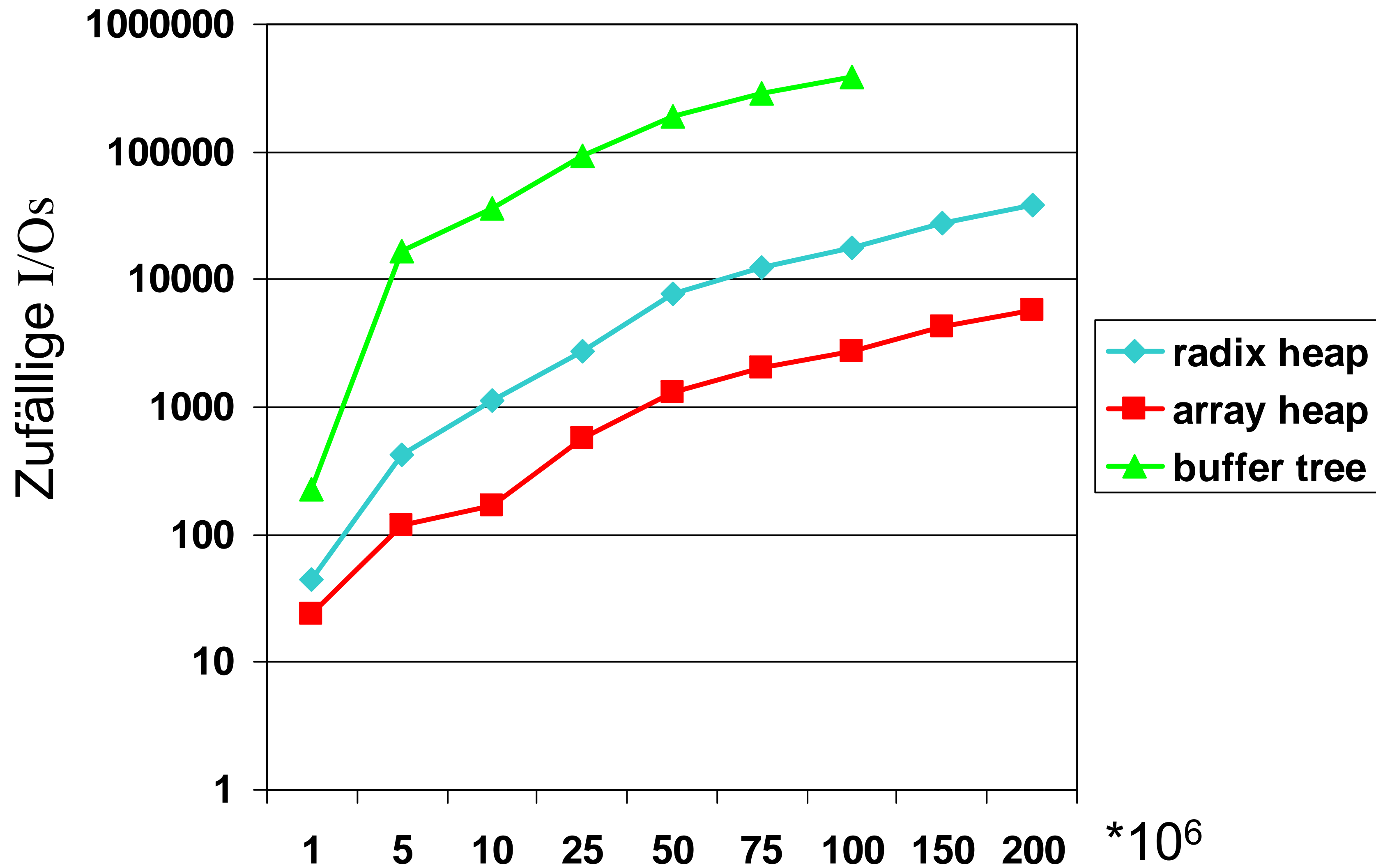
Laufzeit Insert-All-Delete-All: Del_Min



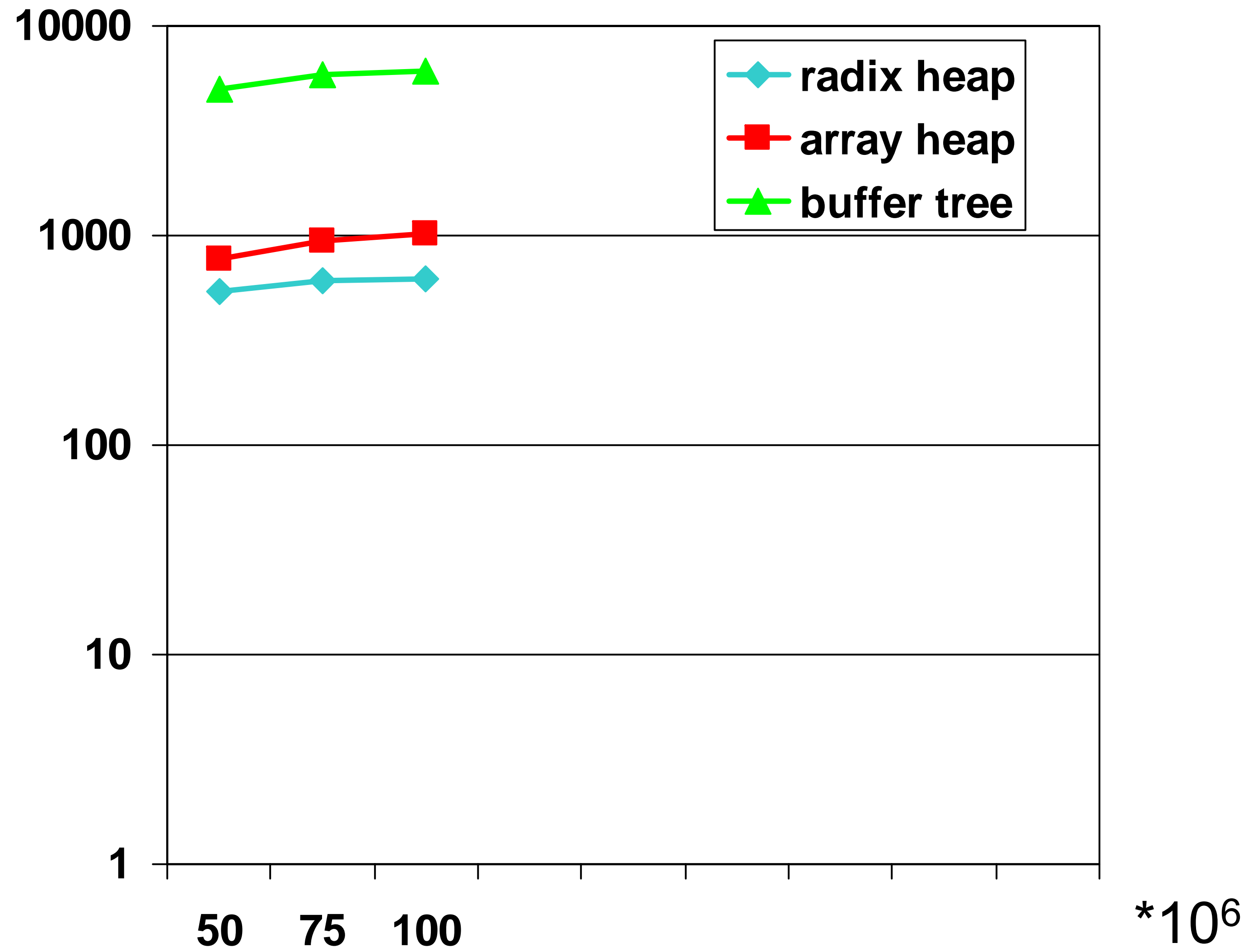
I/Os Insert-All-Delete-All: Total I/Os



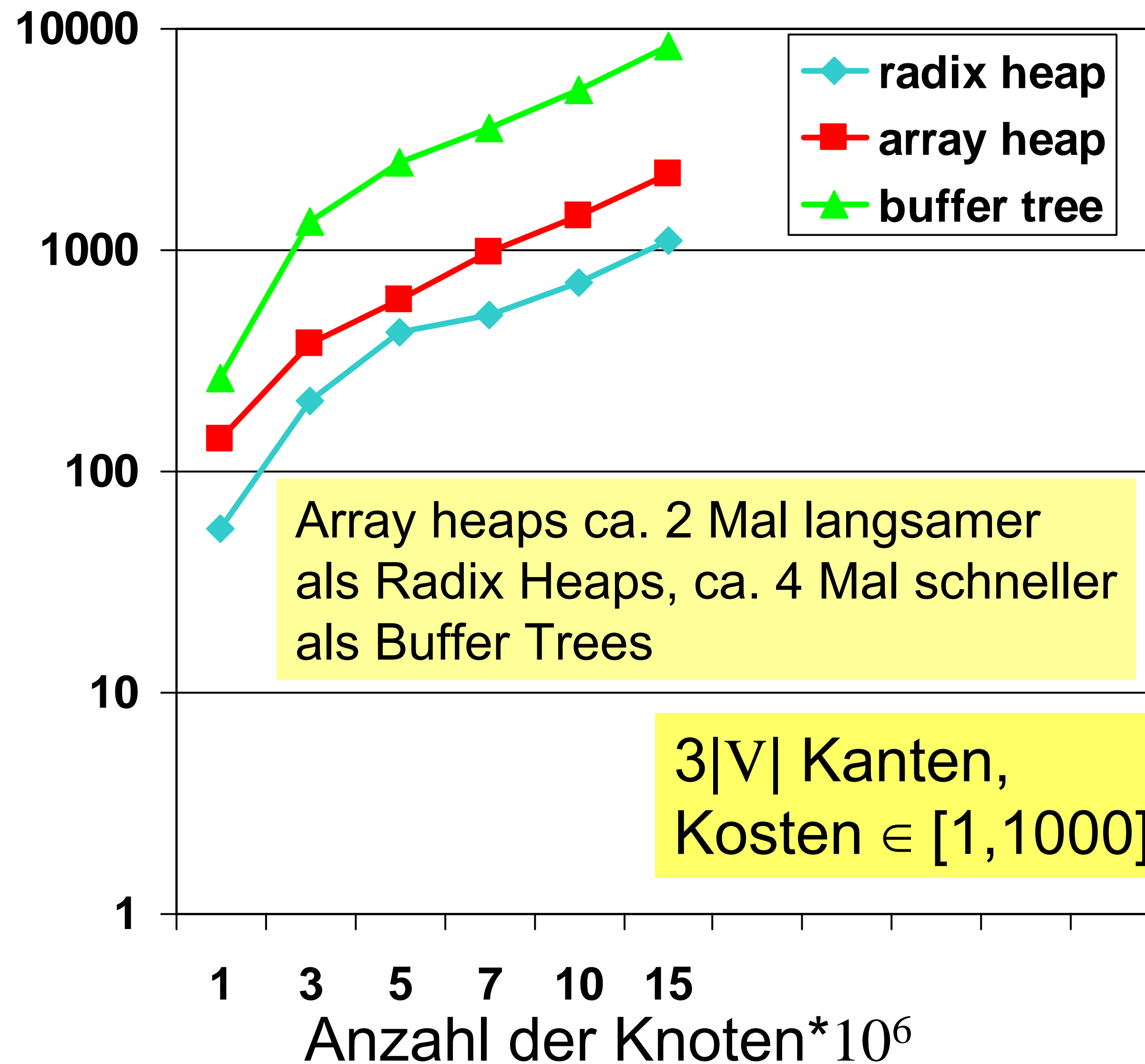
I/Os Insert-All-Delete-All: Random I/Os



Laufzeit Intermixed



Laufzeit Dijkstra



ENDE