Priority Queue Benchmark

Knut Krause Thomas Siwczyk Stefan Tittel

Technische Universität Dortmund
Fakultät für Informatik

Algorithmen und Datenstrukturen
20. November 2008

Gliederung

Gliederung

Getestete Datenstrukturen

- Fibonacci Heap
  - Implementierung des JGraphT-Projekts[1]
- Binary Heap
  - `PriorityQueue`-Klasse der Java-API
- Pairing Heap
  - Implementierung von Mark Allen Weiss[2]

Anmerkung

- verwendete Datenstrukturen ohne Einschränkungen (wie Monotonie)
- beliebige Operationsfolgen möglich

[1] `http://www.jgrapht.org`
[2] `http://tinyurl.com/56pf9f`

Theoretische Laufzeiten

	Fibonacci Heap	Binary Heap	Pairing Heap
`insert`	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
`extractMin`	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
`decreaseKey`	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$2^{\mathcal{O}(\sqrt{\log \log n})}$
	amortisiert	Worst Case	amortisiert

Anmerkung

amortisierte Laufzeit bei `decreaseKey` im Pairing Heap ist mindestens $\Omega(\log \log n)$

Gliederung

Allgemeines Vorgehen

- Implementierung in Java
- Benutzung fremder Priority-Queue-Implementierungen
- verwendete Adapter mit exakt gleichem Overhead für alle Implementierungen
- Bildung des arithmetischen Mittels der Laufzeit über mehrere identische Durchläufe
- zufällige Durchführung von `insert`, `decreaseKey` und `extractMin` mit einstellbarer Wahrscheinlichkeit
- initiale Füllung durch `insert`-Operationen
- identische Operationen, Schlüsselwerte und Operationsfolgen für alle Priority Queues
- per Hand gesetzter Random Seed
- manuelle Garbage Collection, ansonsten Ergebnisse abhängig von Zeitpunkt der Priority-Queue-Ausführung

Optionen

- Anzahl der Wiederholungen pro Priority Queue
- Anzahl initialer `insert`-Operationen
- maximaler Schlüsselwert
- Anzahl durchzuführender Operationen
- Wahrscheinlichkeit von
  - „Operation ist `insert`“
  - „Operation ist `decreaseKey`“
  - „Operation ist `extractMin`“
  - „füge Knoten zu Kandidatenliste hinzu“ (dazu später mehr)
- Random Seed

Hilfsdatenstruktur für `decreaseKey` 1/4

- `decreaseKey` definiert für <Knoten, neuer Schlüsselwert>
- *Problem:* Zugriff auf Knoten
  - nur möglich mit Hilfsdatenstruktur, da kein wahlfreier Zugriff auf Knoten in Priority Queue
  - Hinzufügen eines Knotens ohne großen Overhead
  - *zufällige* Auswahl eines Knoten für `decreaseKey` ohne großen Overhead
  - nur Knoten von Interesse, die nicht schon vorher durch `extractMin` entfernt wurden

Hilfsdatenstruktur für `decreaseKey` 2/4

Lösung:

- verwende verlinkte Liste für Knoten, die für `decreaseKey` in Frage kommen (Kandidatenliste)
- verwende Hashmap <Schlüsselwert, Knoten> für bereits per `extractMin` entfernte Knoten
- füge bei `insert` zufällig mit einstellbarer Wahrscheinlichkeit Knoten der Kandidatenliste hinzu
- füge bei `extractMin` extrahierten Knoten und dessen Schlüssel der Hashmap hinzu

Hilfsdatenstruktur für `decreaseKey` 3/4

Führe `decreaseKey` wie folgt durch:

1. entferne ersten Knoten aus Kandidatenliste
2. prüfe, ob Schlüsselwert des Knotens in Hashmap
   - falls nein: Knoten ist gesund
   - falls ja: prüfe, ob zugehöriger Knoten identisch mit Knoten aus Kandidatenliste
     - falls ja: Knoten ist nicht gesund
     - falls nein: Knoten ist gesund
3. falls Knoten gesund: führe `decreaseKey` durch
4. falls Knoten nicht gesund: springe zurück zu 1.

Laufzeitbewertung der Hilfsdatenstruktur

Overhead ist klein $\Rightarrow$ verfälscht Benchmark nicht

Hilfsdatenstruktur für `decreaseKey` 4/4

Implizite Nebenbedingung

E(gesunde Knoten in Kandidatenliste) >
E(#`decreaseKey`-Operationen)

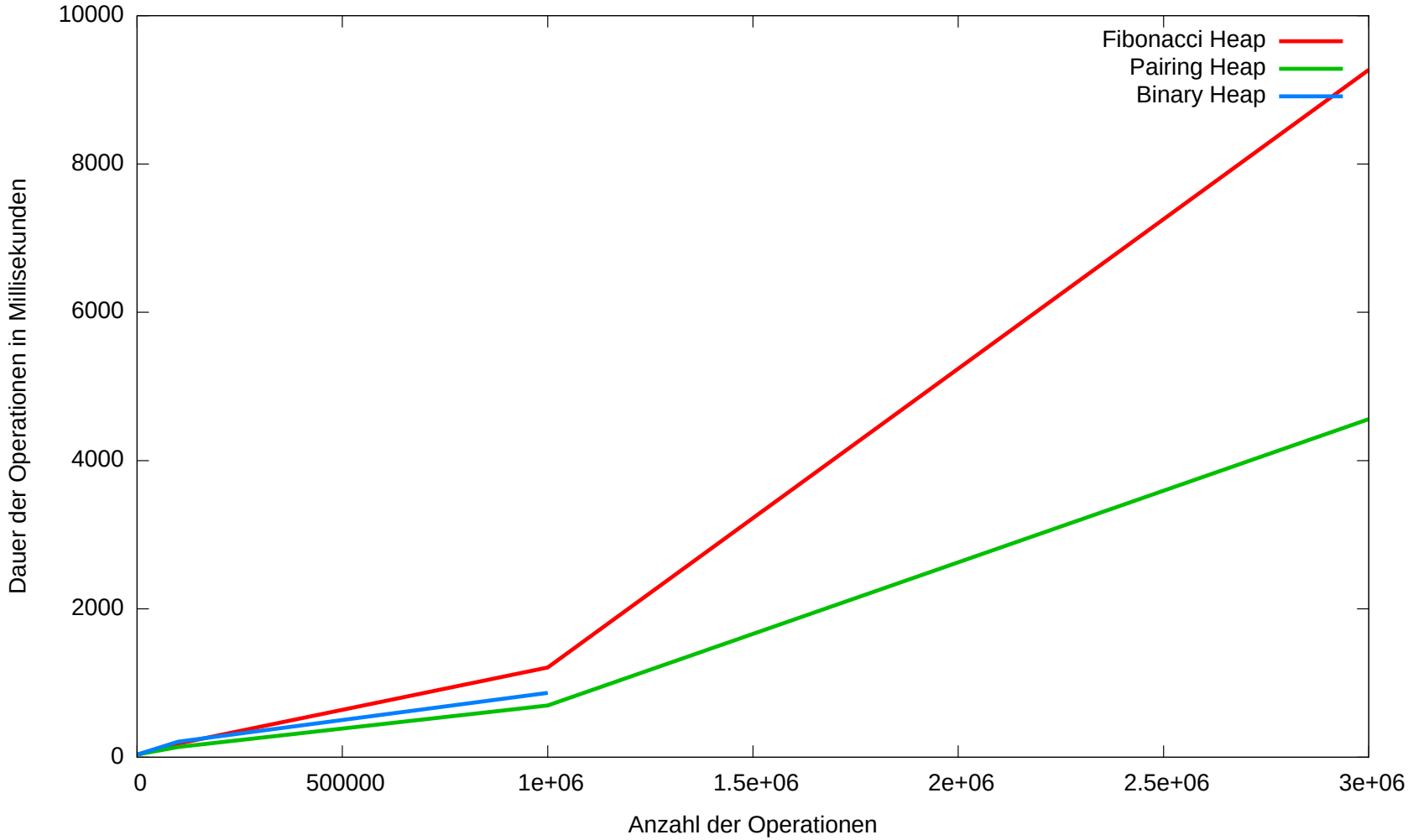
- praktisch kein Nachteil: erhöhte Wahrscheinlichkeit von „füge Knoten zu Kandidatenliste hinzu“ führt zu kaum messbarer Laufzeiterhöhung
- Grund deshalb nicht gleich alle Knoten der Kandidatenliste hinzuzufügen: Zufälligkeit der Knoten für `decreaseKey` nicht nur hinsichtlich der Schlüsselwerte, sondern auch hinsichtlich `insert`-Zeitpunkts

Gliederung

Einstellungen mit Fokus auf `insert`

- Initiale `insert`-Operationen: 50 000
- P(`insert`) = 83,33 %
- P(`deleteMin`) = 16,66 %
- P(`decreaseKey`) = 0 %

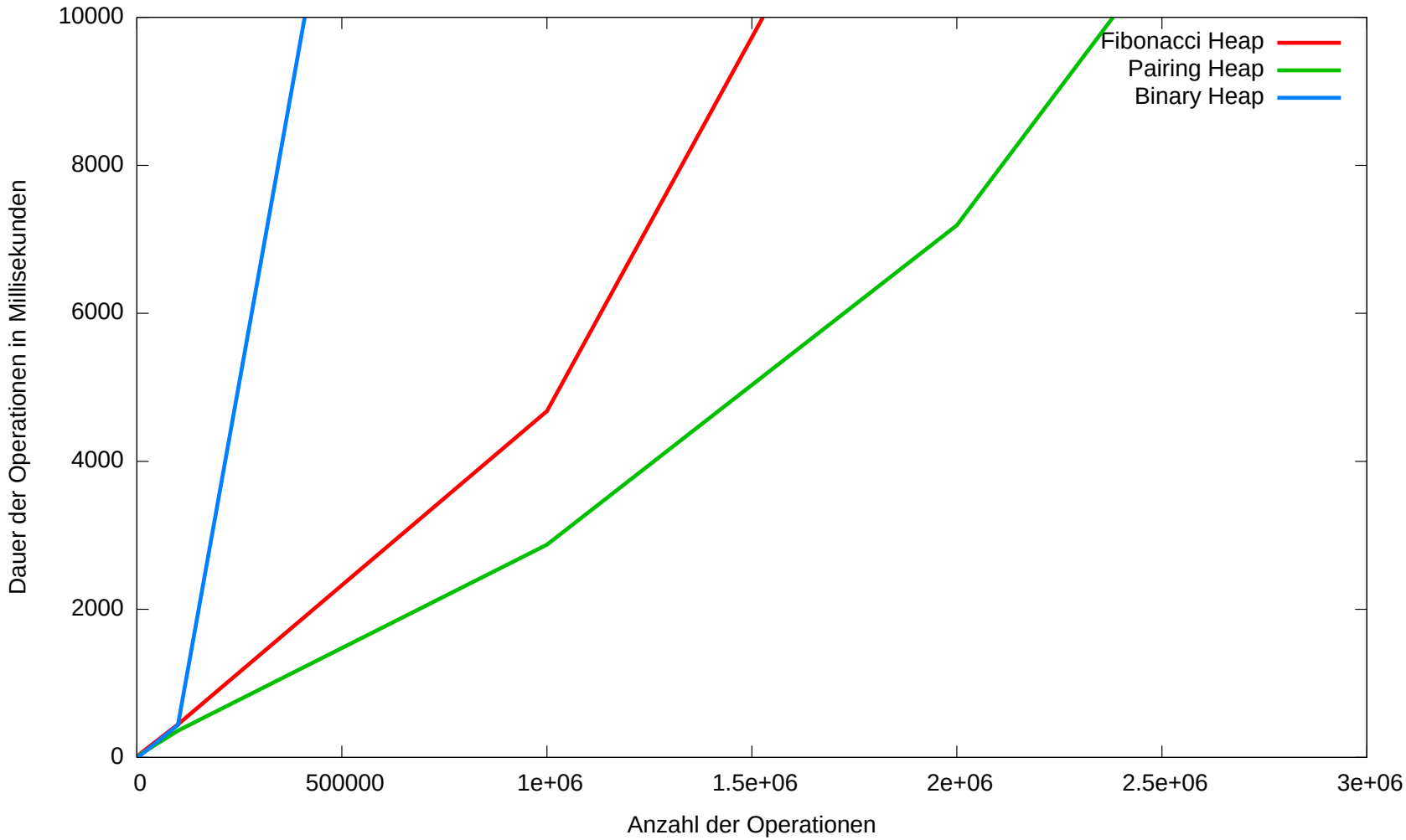
Fokus auf `insert`



Einstellungen mit Fokus auf `extractMin`

- Initiale `insert`-Operationen: 1 000–3 000 000
- P(`insert`) = 15,78 %
- P(`deleteMin`) = 78,94 %
- P(`decreaseKey`) = 5,26 %

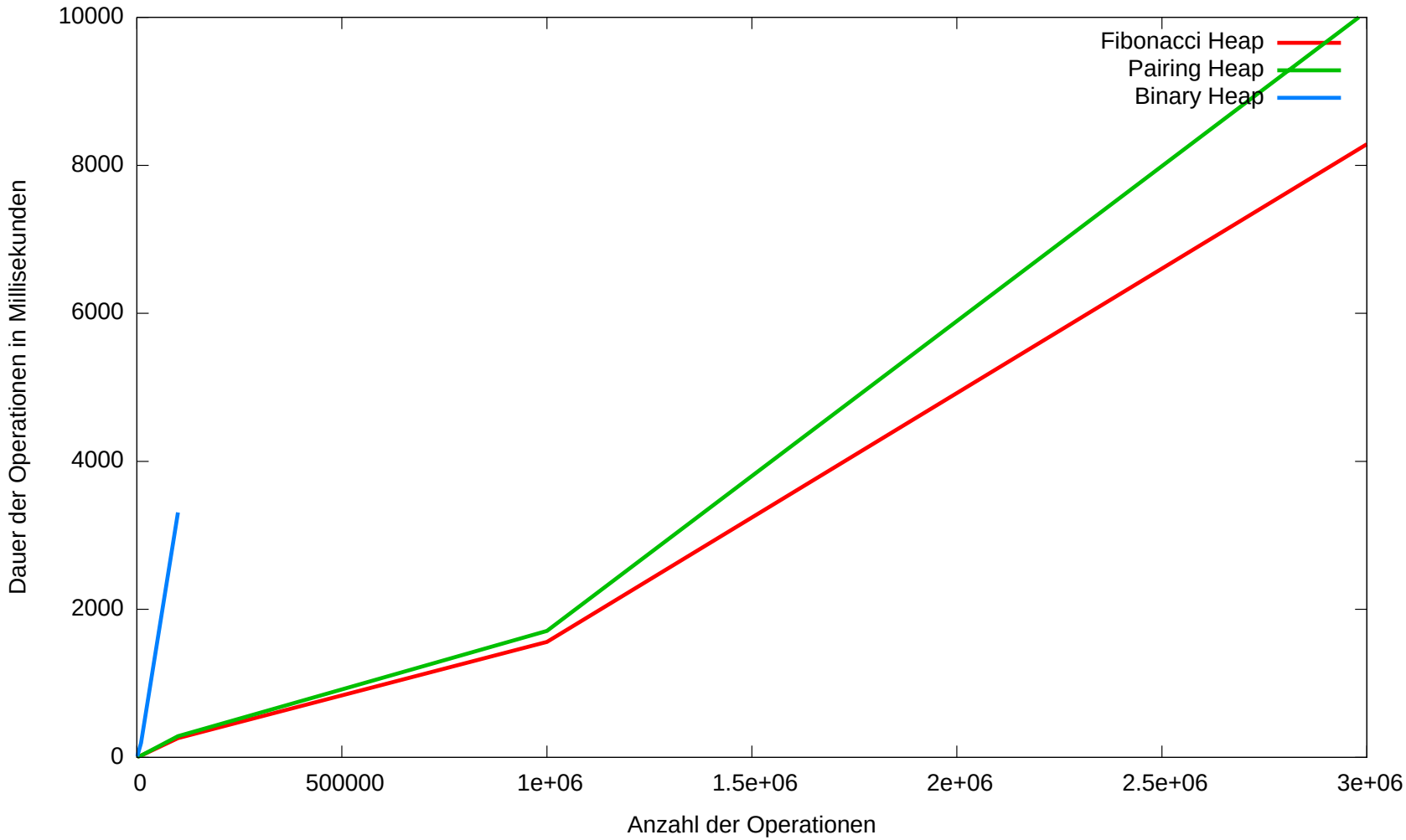
Fokus auf `extractMin`



Einstellungen mit Fokus auf `decreaseKey`

- Initiale `insert`-Operationen: 1 000–5 000 000
- P(`insert`) = 15,78 %
- P(`deleteMin`) = 5,26 %
- P(`decreaseKey`) = 78,94 %

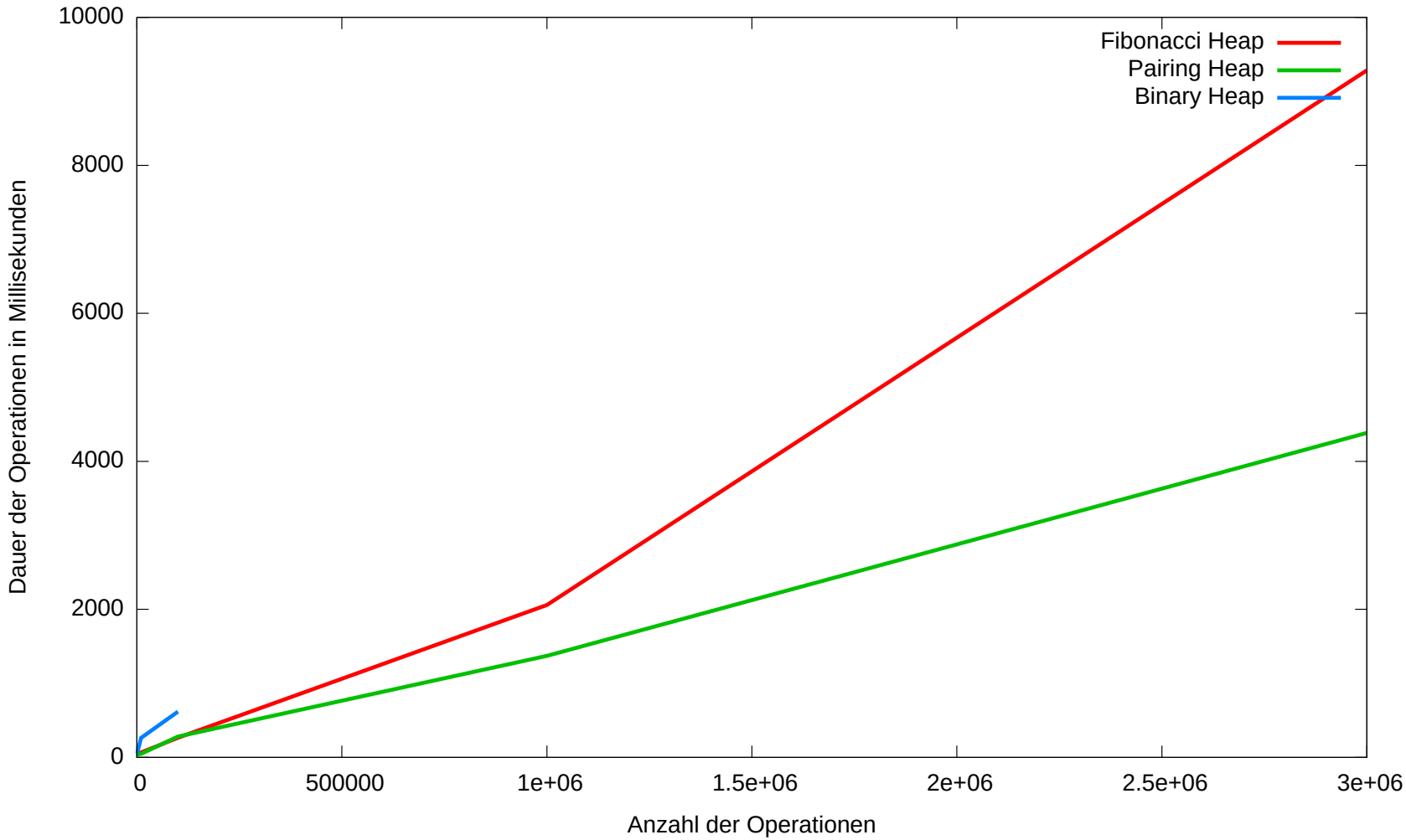
Fokus auf decreaseKey



Einstellungen mit Fokus auf gemischter Anwendung

- Initiale `insert`-Operationen: 50 000
- P(`insert`) = 55,55 %
- P(`deleteMin`) = 25,92 %
- P(`decreaseKey`) = 18,51 %

Fokus auf gemischter Anwendung



Ergebnis

Die Tests zeigen, dass...

- ...Pairing Heaps bei `insert`- und `extractMin`-Operationen am besten abschneiden.
- ...die asymptotische untere Schranke von Pairing Heaps auch in der Praxis eine Rolle spielt.
- ...Fibonacci Heaps bei vielen `decreaseKey`-Operationen die besten Resultate erzielen.
- ...Binary Heaps nur bei wenigen Daten konkurrenzfähige Resultate erzielen.
- ...Binary Heaps bei wenigen Daten teilweise sogar am besten abschneiden.