Prioritätswarteschlangen

Linksbaum und binomialer Heap

Algorithm Engineering, Prof. Weicker

Matthias Jauernig

Fachbereich Informatik, Mathematik und Naturwissenschaften

19.11.2007

Rahmenbedingungen

- Zeitnahme über `ThreadMxBean`
- Durchschnittswerte über 100 Iterationen
- keine Benutzerinteraktionen mit dem Rechner
- Zugriffsdatenstruktur: Java `Hashtable` (`Map`-Interface)
- gespeicherte Objekte: Key und Value als Integer
- Prioritäten zufällig im Bereich 1..n*10 erzeugt (n = Elementanzahl)
- bei über 300000 Elementen Stack Overflow
  (damit maximal verwendete Queue-Größe)
- Linksbaum vs. Binomialheap interessant aufgrund gleicher Komplexitäten

Rahmenbedingungen

- Zeitnahme über `ThreadMxBean`
- Durchschnittswerte über 100 Iterationen
- keine Benutzerinteraktionen mit dem Rechner
- Zugriffsdatenstruktur: Java `Hashtable` (`Map`-Interface)
- gespeicherte Objekte: Key und Value als Integer
- Prioritäten zufällig im Bereich 1..n*10 erzeugt (n = Elementanzahl)
- bei über 300000 Elementen Stack Overflow (damit maximal verwendete Queue-Größe)
- Linksbaum vs. Binomialheap interessant aufgrund gleicher Komplexitäten

Rahmenbedingungen

- Zeitnahme über `ThreadMxBean`
- Durchschnittswerte über 100 Iterationen
- keine Benutzerinteraktionen mit dem Rechner
- Zugriffsdatenstruktur: Java `Hashtable` (`Map`-Interface)
- gespeicherte Objekte: Key und Value als Integer
- Prioritäten zufällig im Bereich 1..n*10 erzeugt (n = Elementanzahl)
- bei über 300000 Elementen Stack Overflow
  (damit maximal verwendete Queue-Größe)
- Linksbaum vs. Binomialheap interessant aufgrund gleicher Komplexitäten

Interface für Priority Queues

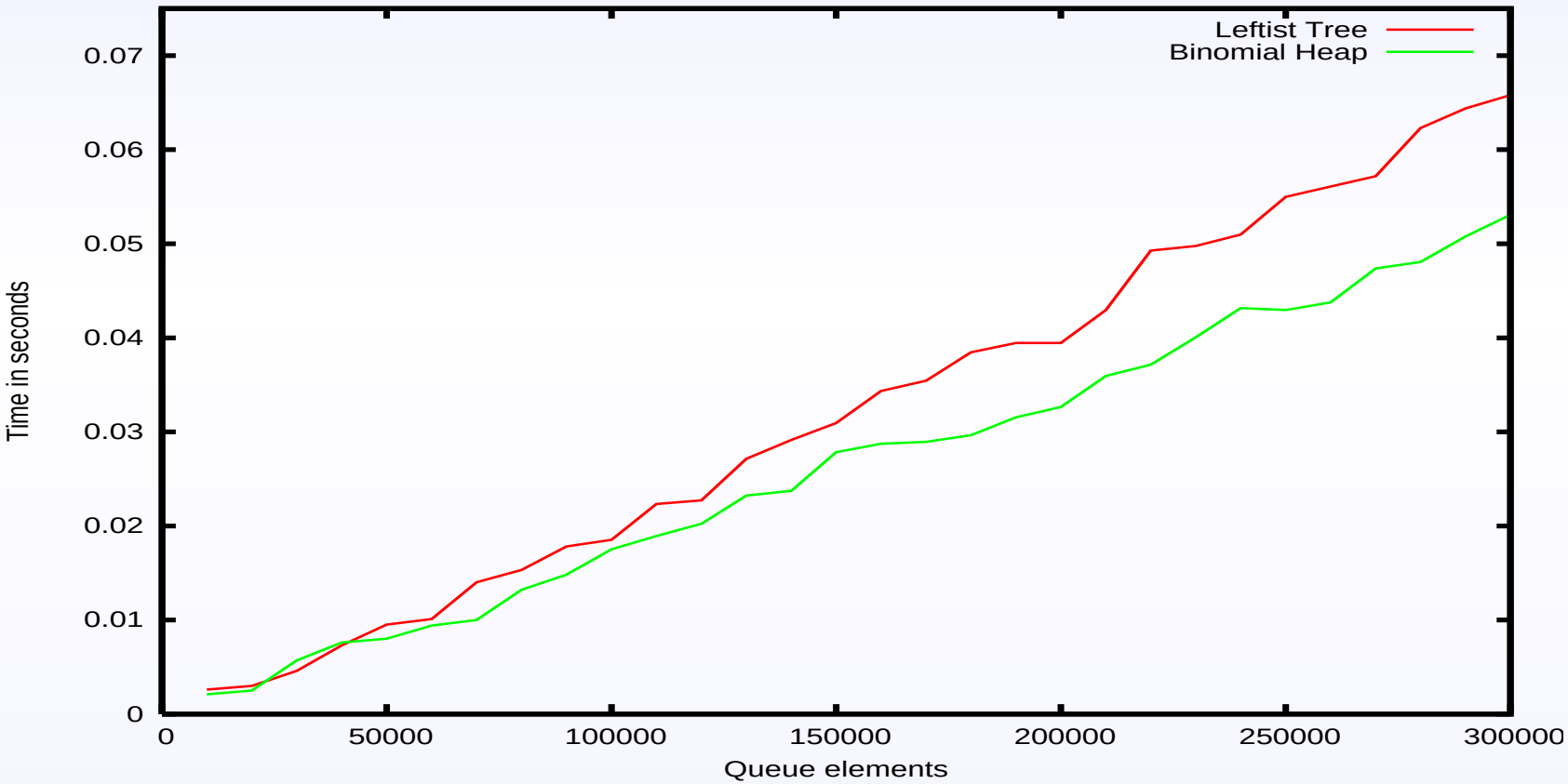
```
public interface PriorityQueue<K,V> {
  public void insert(PriorityQueueNode<K,V> x);
  public PriorityQueueNode<K,V> getMinimum();
  public PriorityQueueNode<K,V> deleteMinimum();
  public void merge(PriorityQueue<K,V> q2);
  public void decreasePriority(
    PriorityQueueNode<K,V> node, int newValue);
  public boolean delete(PriorityQueueNode<K,V> node);

  public PriorityQueueNode<K,V> getNode(K key);
  public Set<K> getKeySet();

  public boolean checkQueueIntegrity();
  public int size();
}
```

Operation `init`

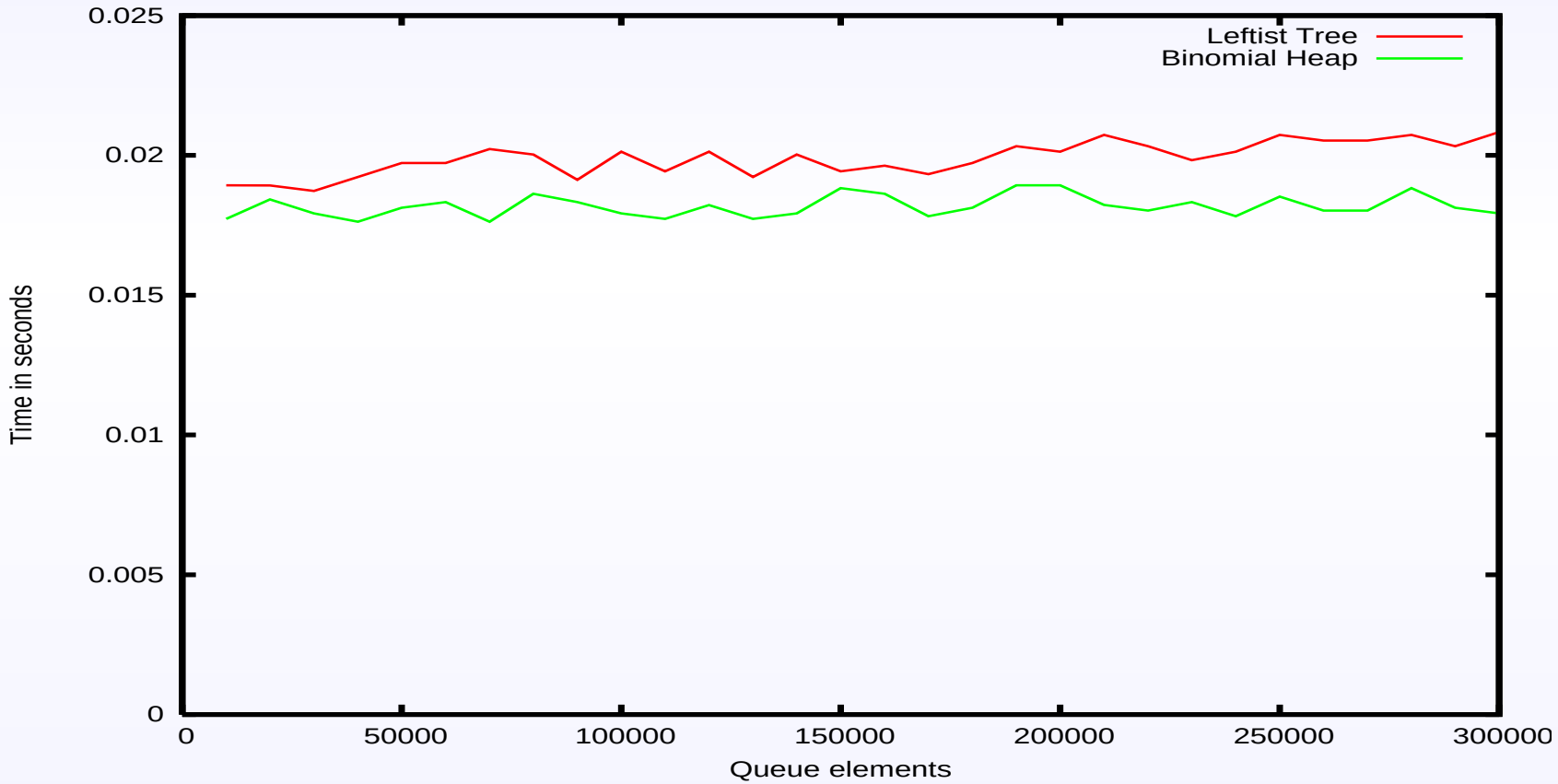
- Szenario: Initialisierung von max. 300000 Elementen
- Benötigte Zeit:



- Auswertung: Binomialheap besser

Operation `insert`

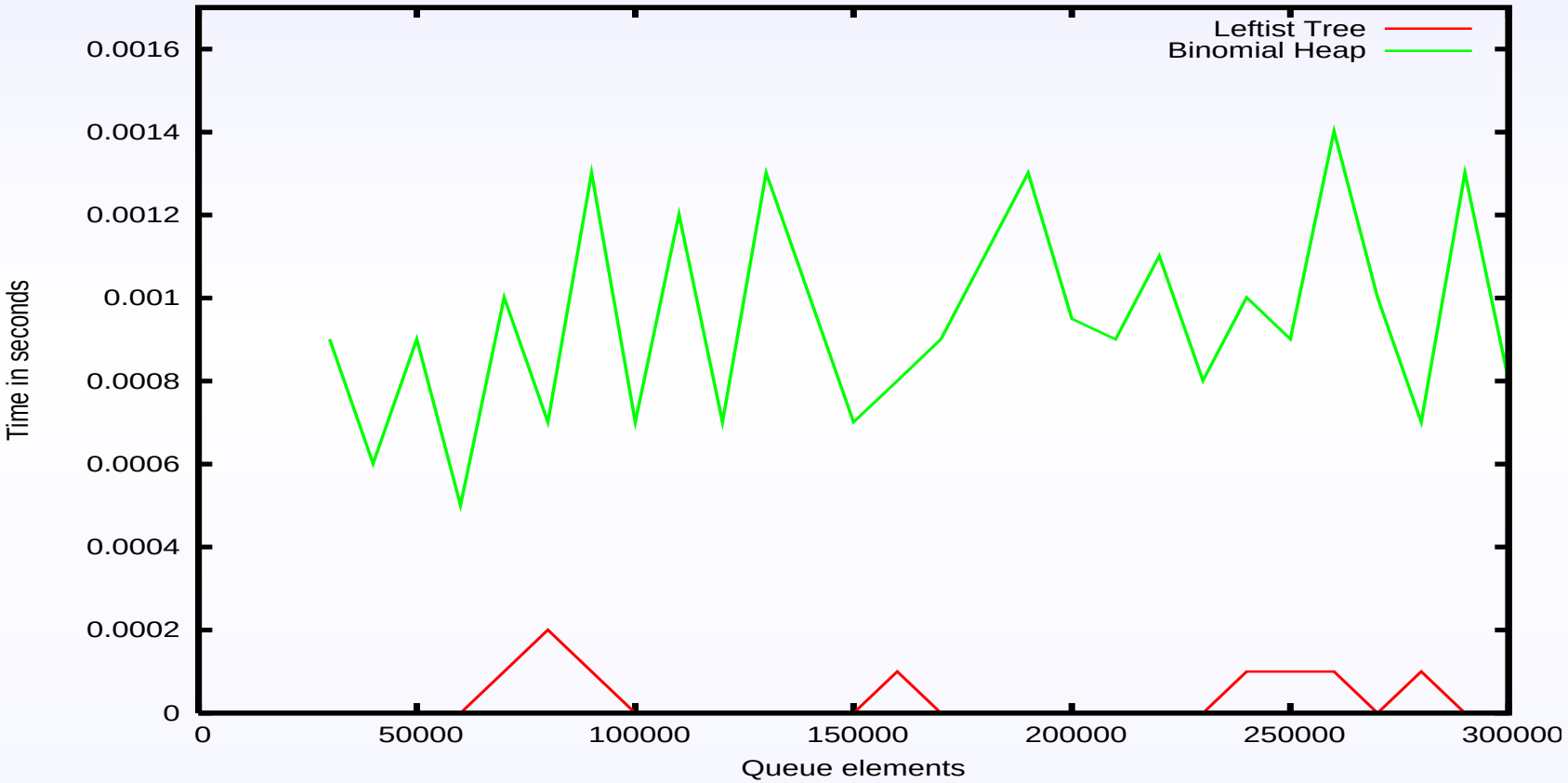
- Szenario: Queues bis 300000 Elemente, füge jeweils 30000 neue Elemente hinzu
- Benötigte Zeit:



- Auswertung: Binomialheap besser

Operation `getMinimum`

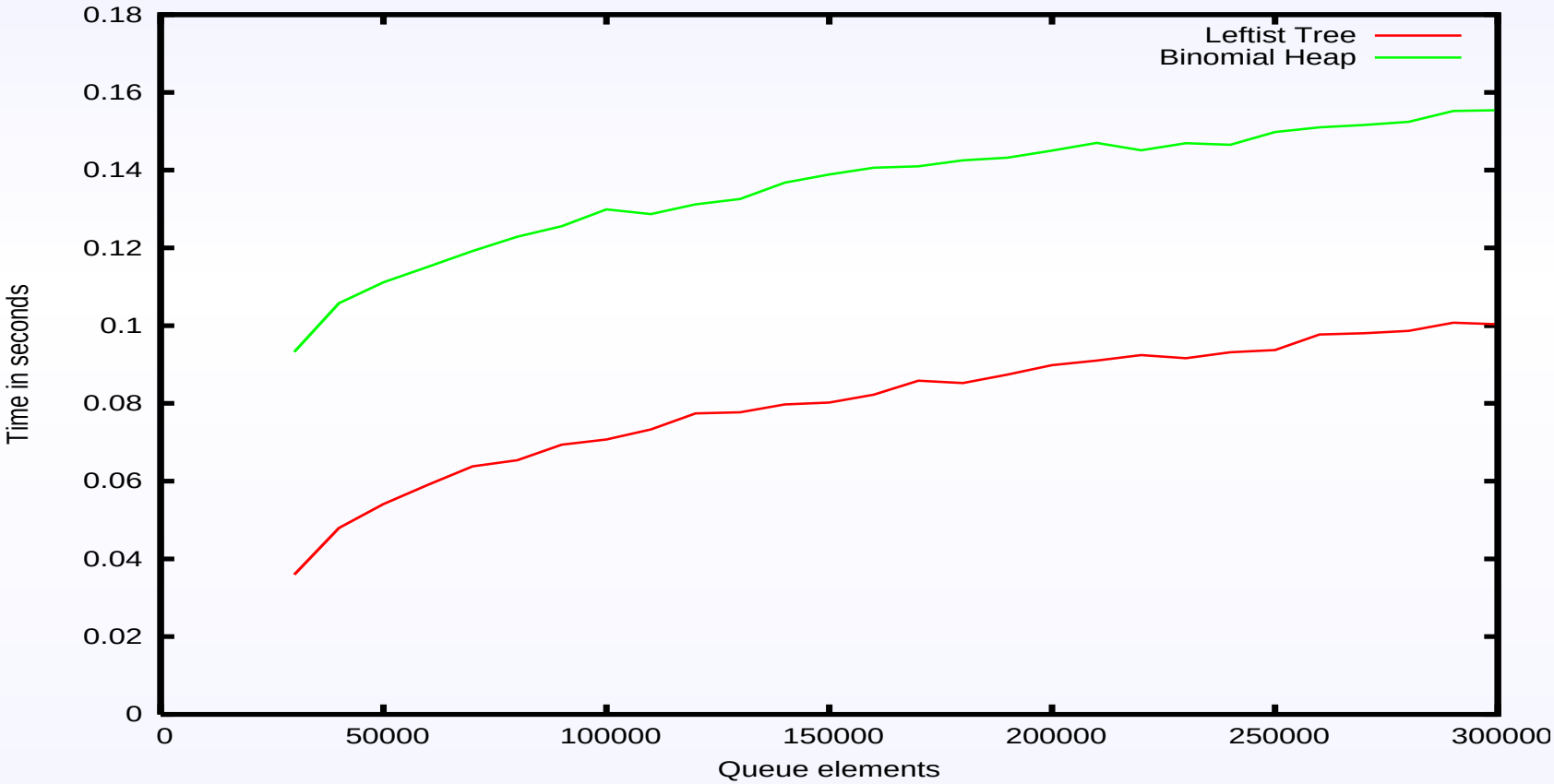
- Szenario: Queues bis 300000 Elemente, 30000 Minimum erhalten
- Benötigte Zeit:



- keine exakten Laufzeiten aufgrund sehr kleiner Zeit-Beträge
- Auswertung: Linksbaum besser (direkter Zugriff)

Operation `deleteMinimum`

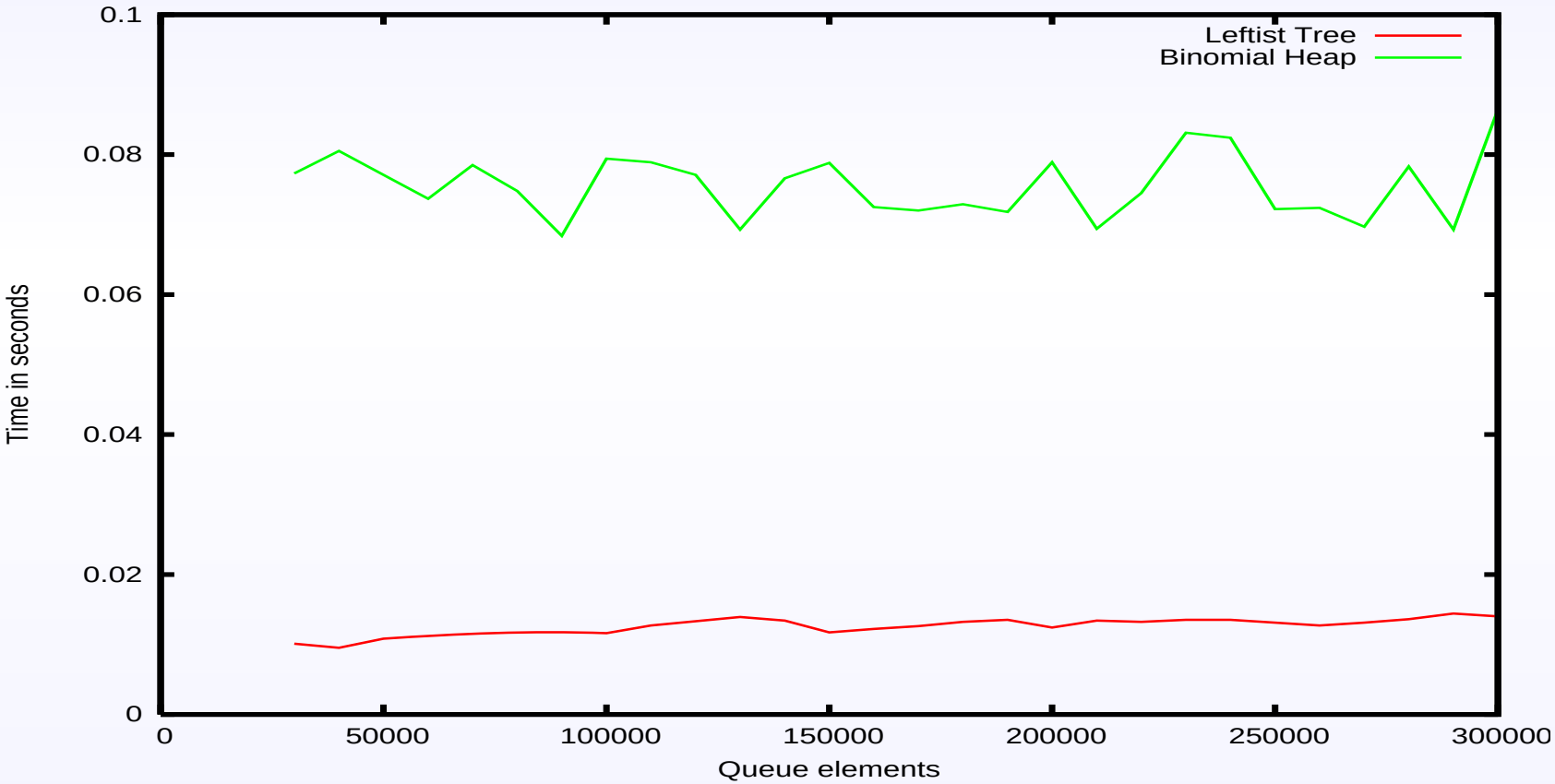
- Szenario: Queues bis 300000 Elemente, lösche jeweils 30000 mal das Minimum
- Benötigte Zeit:



- Logarithmischer Aufwand gut erkennbar
- Auswerung: Linksbaum besser

Operation `delete`

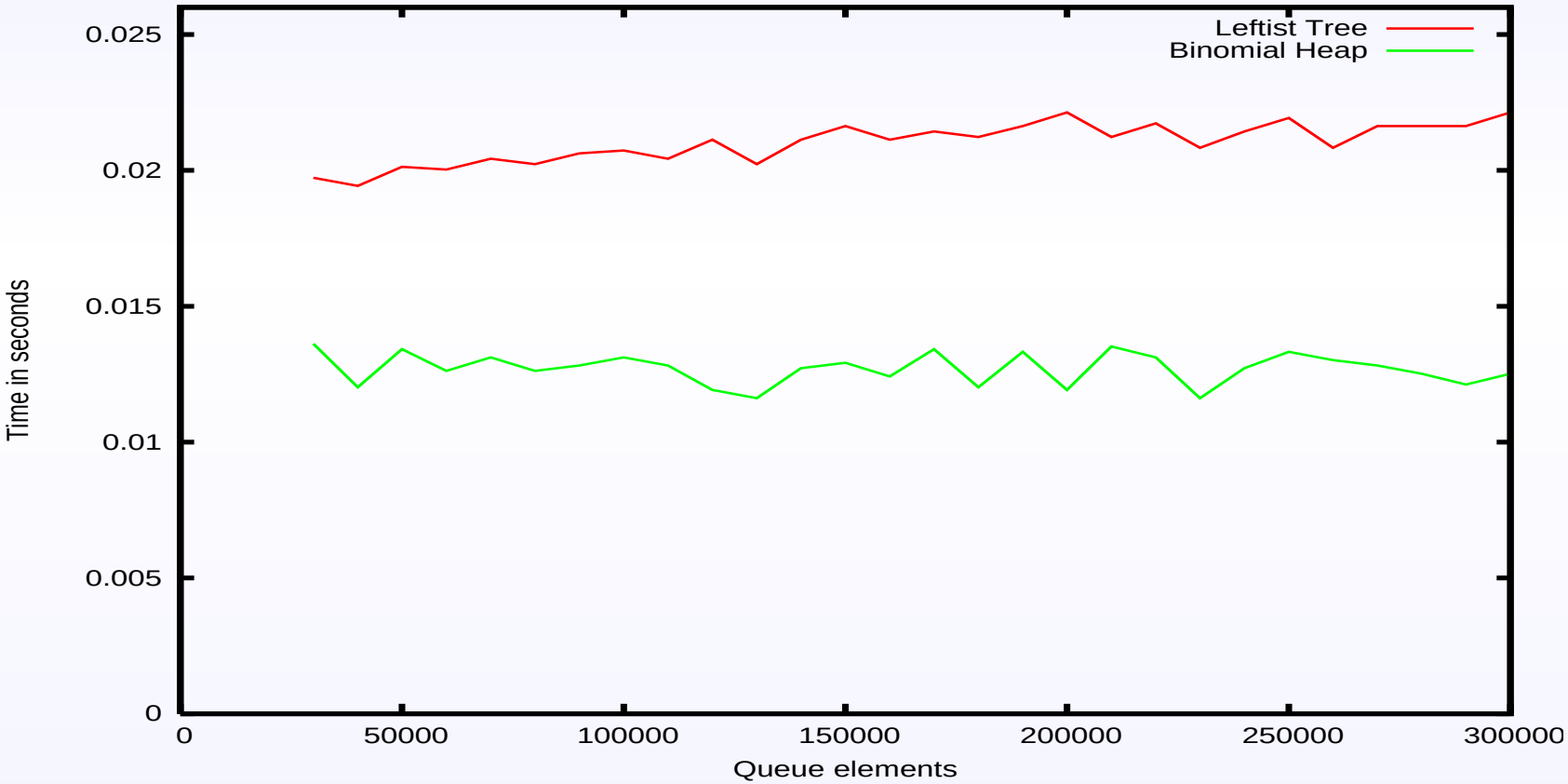
- Szenario: Queues bis 300000 Elemente, lösche jeweils 30000 Elemente (zufällige Priorität)
- Benötigte Zeit:



- Auswerung: Linksbaum besser

Operation `decreasePriority`

- Szenario: Queues bis 300000 Elemente, verringere bei 30000 Elementen (zufällige Priorität) die Priorität um einen zufälligen Wert
- Benötigte Zeit:



- Auswerung: Binomialheap besser

Operation `merge`

- Szenario: Erzeuge 2 Queues mit zusammen maximal 250000 Elementen und merge sie
- Zeit für merge zu gering um brauchbare Ergebnisse zu erhalten

Gewinner-Überblick

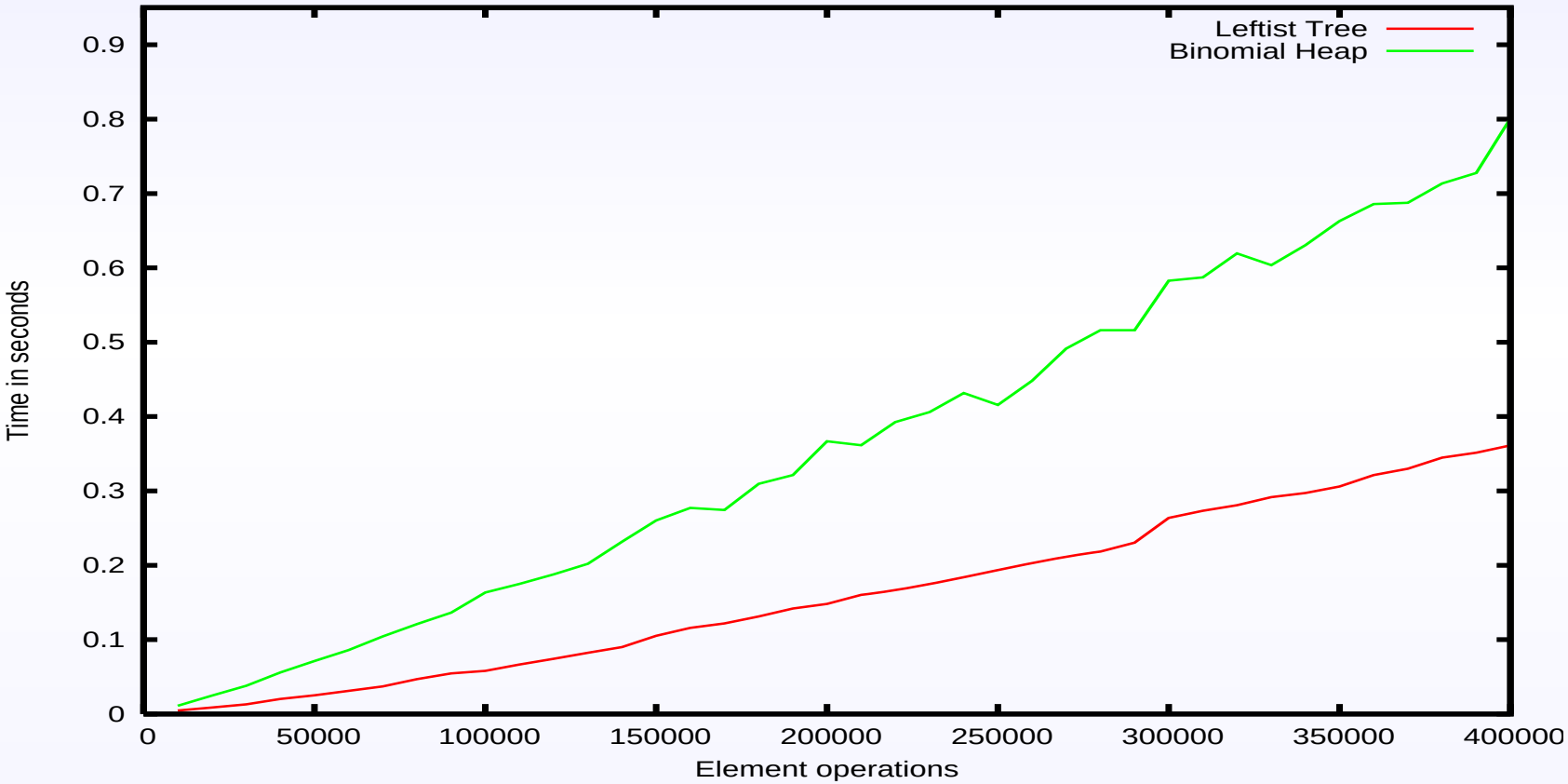
- `init`: Binomialheap
- `insert`: Binomialheap
- `getMinimum`: Linksbaum
- `deleteMinimum`: Linksbaum
- `delete`: Linksbaum
- `decreasePriority`: Binomialheap
- `merge`: ?? auf Papier Binomialheap
- ziemlich ausgeglichen, Wahl hängt von Anwendungsfall ab

Involvierte Konstanten

- `init`: siehe `merge`
- `insert`: siehe `merge`
- `getMinimum`: *Linksbaum* – direkt, *Binomialheap* – Länge der Topliste
- `deleteMinimum`: *Linksbaum* – siehe `merge`,
  Binomialheap – siehe `merge` + Länge der Topliste
- `delete`: *Linksbaum* – siehe `merge` + Baumhöhe,
  Binomialheap – `decreasePriority` + `deleteMinimum`
- `decreasePriority`: *Linksbaum* – `delete` + `insert`,
  Binomialheap – Baumhöhe
- `merge`: *Linksbaum* – Baumhöhe x2 (rekursiver Abstieg in rechte Seite und merge mit Vertauschungen nach oben),
  Binomialheap – Länge der Toplisten (Vereinigung dieser + Zusammenlegen von Teilbäumen mit gleichem Grad)

Szenario 1

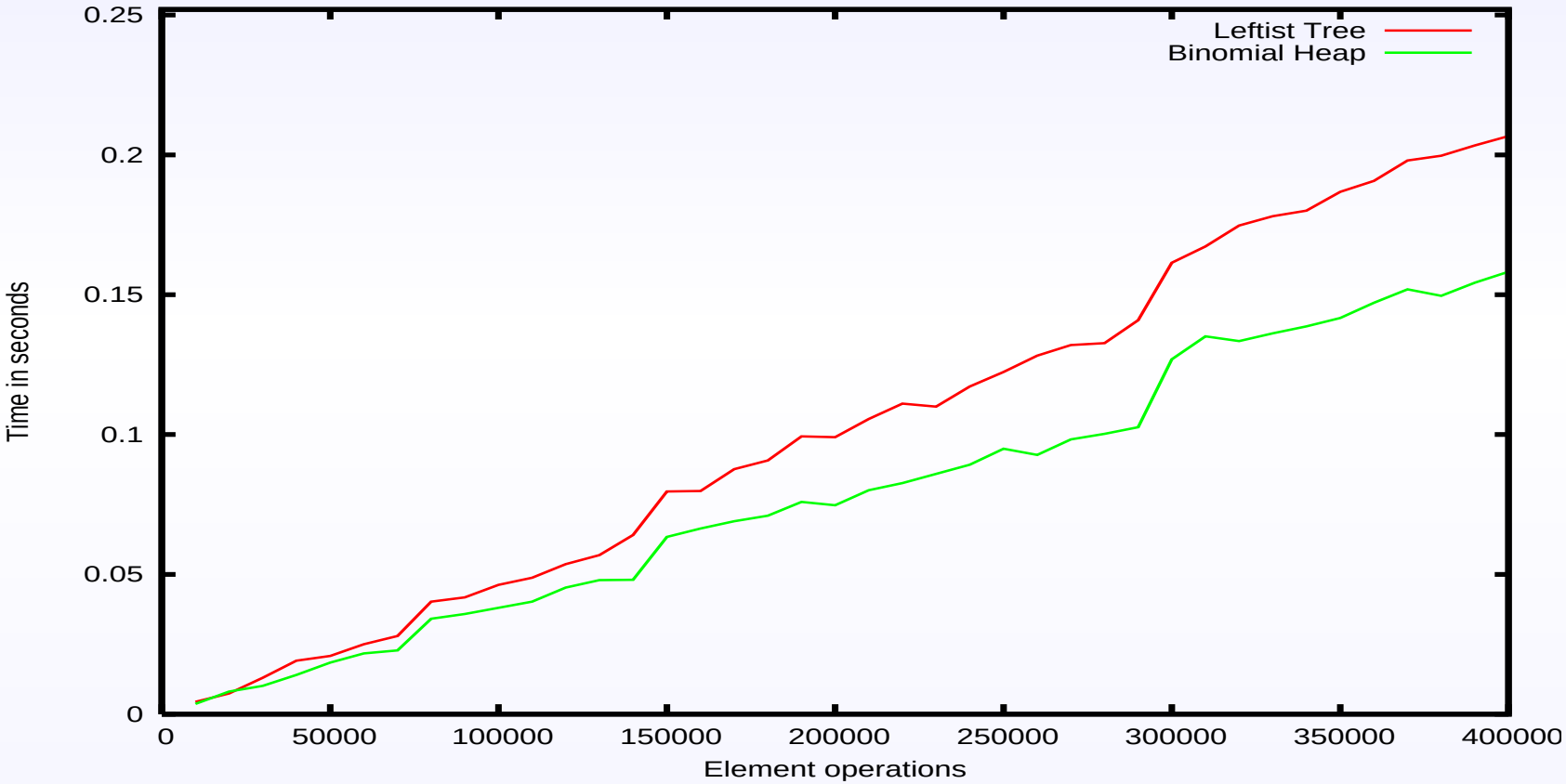
- Operationen: `init`: 50%, `deleteMinimum`: 25%, `delete`: 25%
- Behauptung: Linksbaum ist deutlich besser



- Behauptung bestätigt, Linksbaum bei deleteMinimum und delete überlegen

Szenario 2

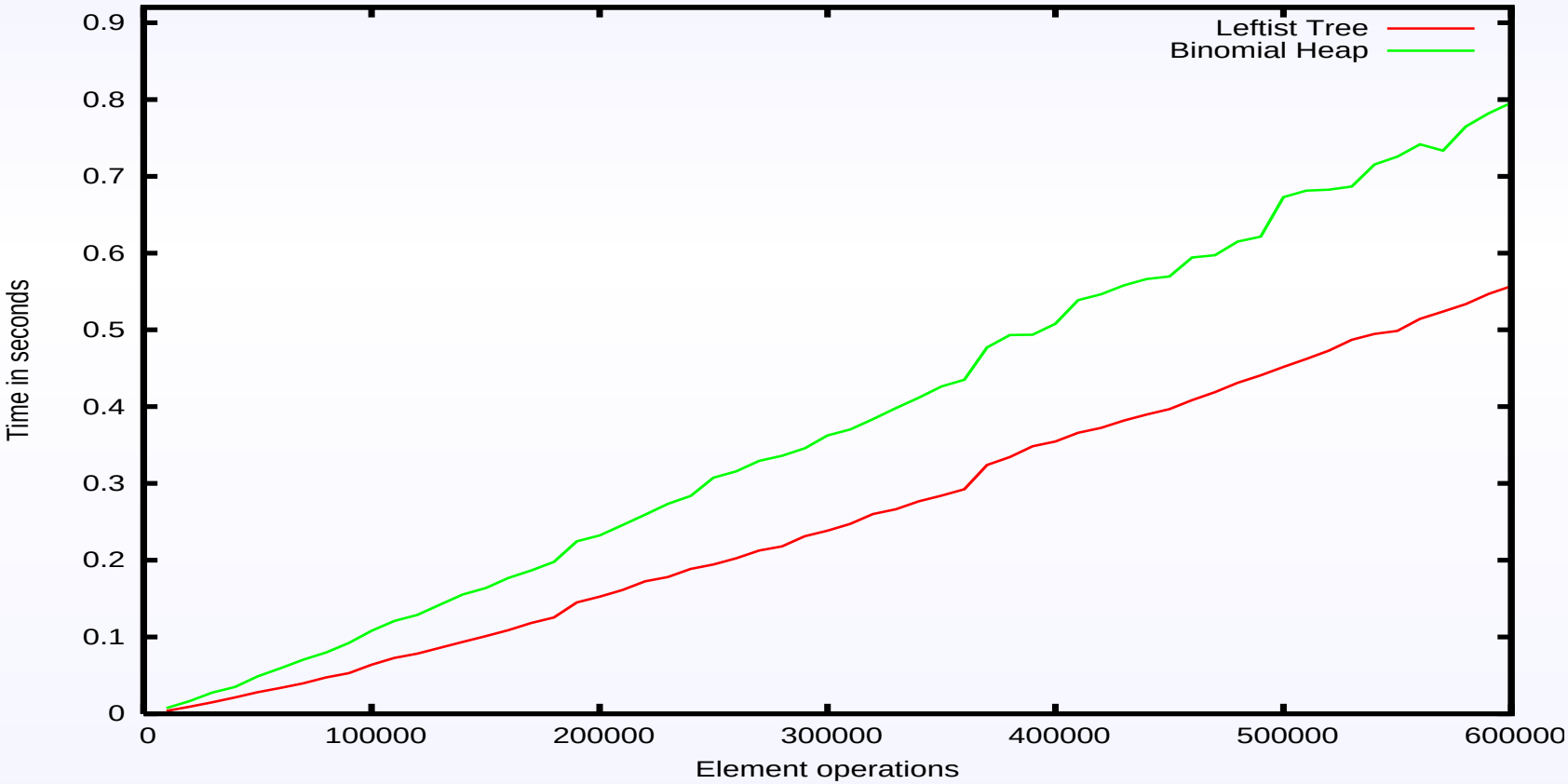
- Operationen: `init`: 50%, `decreasePriority`: 25%, `insert`: 25%
- Behauptung: Binomialheap ist deutlich besser



- Behauptung großteils bestätigt, Binomialheap bei allen Operationen besser, aber nicht *überlegen*

Szenario 3.1

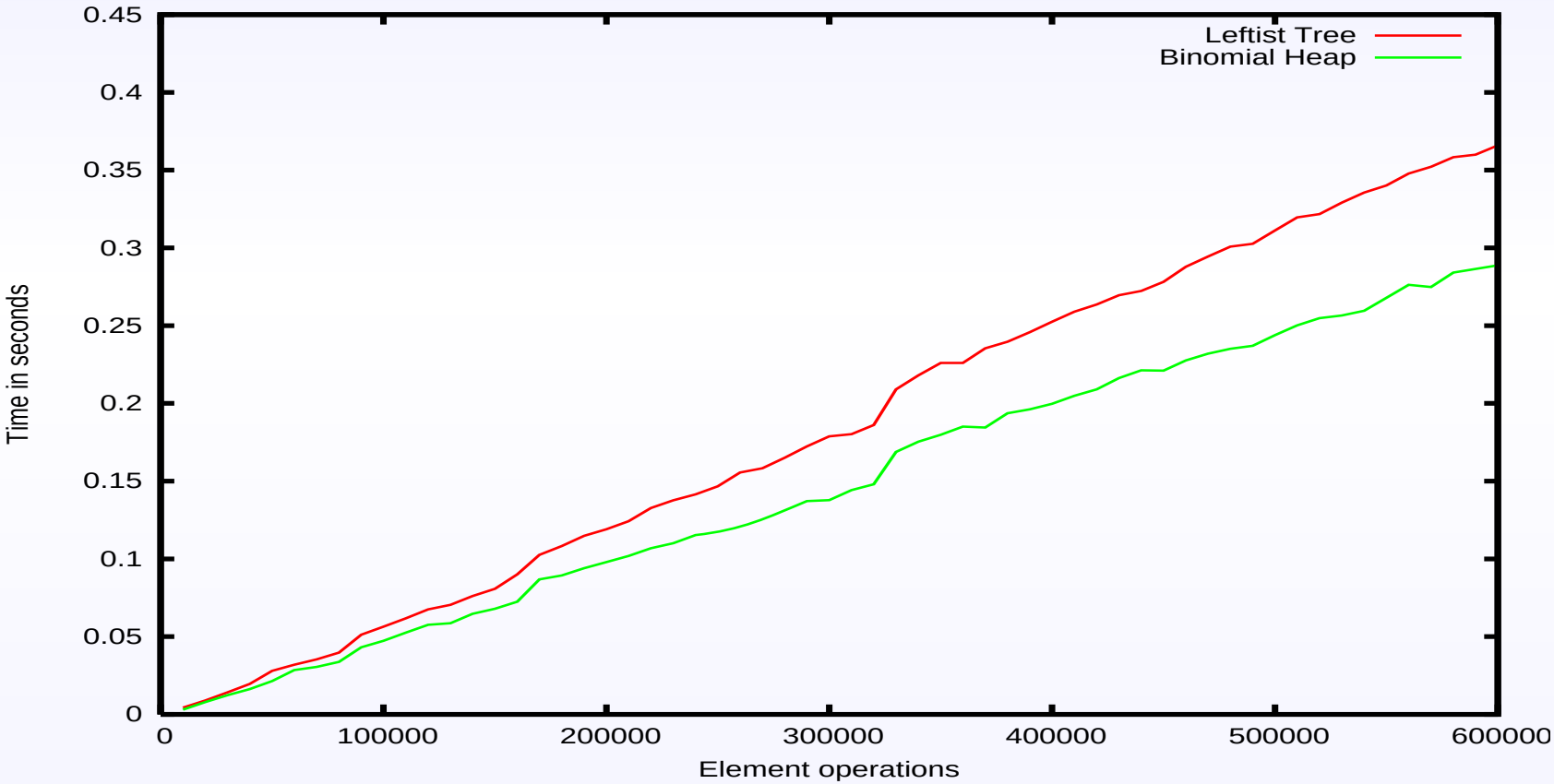
- Operationen: `init`: 40%, `delete`: 25%, `insert`: 10%, `decreasePriority`: 10%, `deleteMinimum`: 15%
- Behauptung: Linksbaum ist deutlich besser



- Behauptung bestätigt, Linksbaum ist überlegen

Szenario 3.2

- Operationen: `init`: 45%, `delete`: 1%, `insert`: 8%, `decreasePriority`: 45%, `deleteMinimum`: 1%
- Behauptung: Binomialheap ist deutlich besser



- Behauptung i.Allg. bestätigt, allerdings trotz stark differierender Prozentsätze kein *überlegener* Vorteil des Binomialheaps