Informatik II: Algorithmen und Datenstrukturen SS 2013

Vorlesung 9b, Mittwoch, 19. Juni 2013

(Prioritätswarteschlangen, alternative Implementierungen)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Alternative Implementierungen für PWs
  - Fibonacci Heaps

    Erweiterung des binären Heaps, asymptotisch schneller

    Wir besprechen nur kurz die Laufzeit
  - Bucket Queues

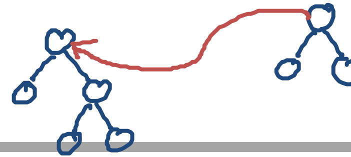
    Für ganzzahlige monotone Operationsfolgen

    Verfahren + Laufzeit ... Ü9, Aufgabe 2: Erweiterung dazu
  - Radix Heaps

    Verbesserung von Bucket Queues, asymptotisch schneller

    Wir besprechen nur kurz die Laufzeit

Fibonacci Heaps



UNI FREIBURG

- Grundidee
  - Ein "Wald" von (nicht mehr unbedingt vollständigen) binären Bäumen, die im Verlauf ineinander gehängt werden
- Laufzeit

  getMin in Zeit O(1) ... wie beim binary heap

  insert in Zeit O(1) ... binary heap O(log n)

  decreaseKey in amortisierter Zeit O(1) ... bin. heap O(log n)

  deleteMin in amortisierter Zeit O(log n) ... bin. heap O(log n)

  - In der Praxis ist der binäre Heap aufgrund seiner Einfachheit und guten Lokalität (Feld) aber schwer zu schlagen

  Selbst für $n = 2^{20} \approx 1.000.000$ ist $\log_2 n$ ja nur 20

  für $n = 2^{10} \approx 1.000$ ist $\log_2 n = 10$

Monotone ganzzahlige PWs 1/2

UNI FREIBURG

- Definition
  - Eine Folge von Operationen auf einer PW heißt monoton ganzzahlig, wenn gilt:

    Die Keys sind ganze Zahlen aus dem Bereich 0 .. M-1

    Für eine beliebige Folge von Operationen, gilt für die Folge x_1, x_2, x_3, … der von den deleteMin Aufrufen gelöschten Minima, dass $x_1 \leq x_2 \leq x_3 \leq \ldots$

  Gegenbeispiel: insert(4), deleteMin(), insert(2), deleteMin(), …

  =4 =2

Monotone ganzzahlige PWs 2/2

- Typische Anwendungen

  - Simulationen in der Zeit

    Es gibt Events zu diskreten (ganzzahligen) Zeitpunkten

    Ein Event kann neue Events (in der Zukunft) generieren

    Die Events will man chronologisch abarbeiten

  - Dijkstra's Algorithmus (zur Berechnung kürzester Wege)

    Man beginnt an einem Startort A

    Man besucht von dort alle anderen Orte in der Reihenfolge aufsteigender Kosten (etwa Zeit oder Entfernung)

    Mehr dazu nächste Woche ...

Bucket Queues 1/9

- Grundidee
  - Ähnlich wie beim ganzzahligen Sortieren mit vielen gleichen Schlüsseln aus einem kleinen Bereich 0..M – 1, zum Beispiel

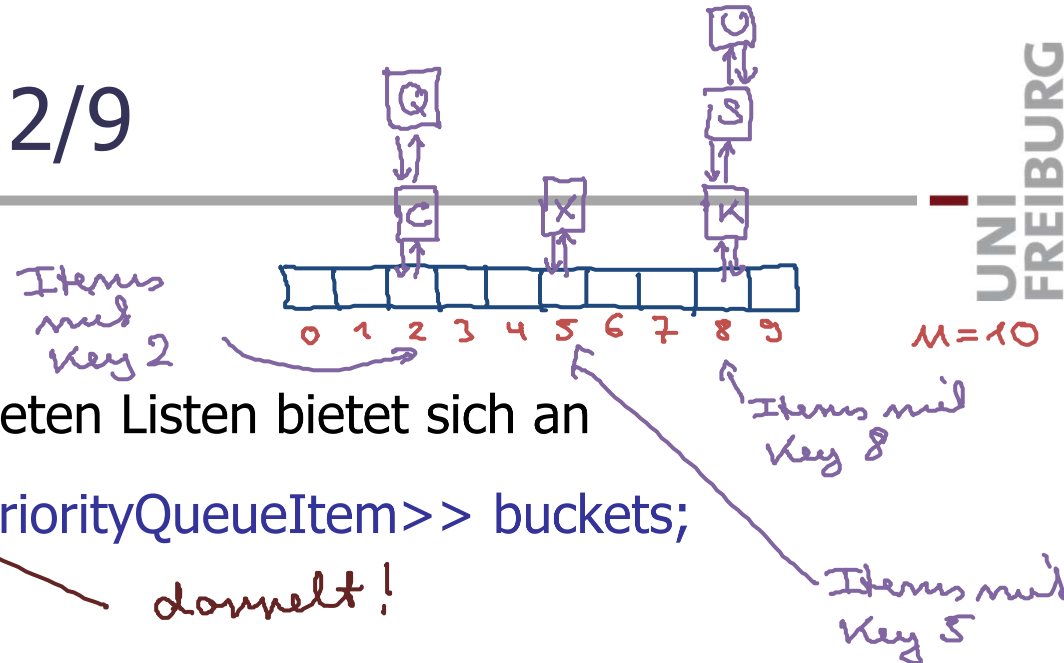
    (2, "S") (1, "A") (0, "U") (2, "D") (0, "O") (2, "F")

  - Da können wir mit einem Feld der Größe M alle Elemente mit demselben Schlüssel zusammen gruppieren

    0: "U", "O"
    1: "A"
    2: "S", "D", F"

  - Genau so fast auch die Bucket Queue alle PQItems mit demselben Key zusammen, in sogenannten Buckets

Bucket Queues 2/9



- Datenstruktur
  - Ein Feld von verketteten Listen bietet sich an

    Array<LinkedList<PriorityQueueItem>> buckets;

    Damit haben wir:

    Zugriff auf Bucket für Schlüssel x in Zeit O(1)

    Einfügen in diesen Bucket / Löschen daraus in O(1) Zeit
  - Außerdem merken wir uns immer ein Item mit dem aktuell minimalen Key (es kann mehrere geben)

    PriorityQueueItem minItem;

im Beispiel oben wäre das C oder Q

Bucket Queues 3/9

UNI FREIBURG

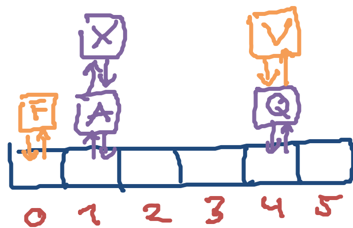
- Insert(newItem)
  - Einfach an die entsprechende Liste anhängen

    `buckets[newItem.key].append(newItem)`
  - Mit einer verketteten Liste pro Eintrag in Zeit O(1)
  - Bei Bedarf minItem neu setzen, geht auch in Zeit O(1)

    `if (newItem.key < minItem.key) { minItem = newItem; }`

insert (4, "V")

insert (0, "F") ⇒ minItem = (0, "F")



m = 6

Bucket Queues 4/9

UNI FREIBURG

- changeKey(item, newKey)
  - Aus der alten Liste löschen und in die neue einfügen

```
item.removeFromBucket();
item.key = newKey;
buckets[newItem.key].append(newItem)
```

  - Geht mit einer verketteten Liste ebenfalls in O(1) Zeit
  - Bei Bedarf minItem neu setzen, geht auch in Zeit O(1)

```
if (item.key < minItem.key) { minItem = item; }
```

Bucket Queues 5/9

UNI FREIBURG

- getMin
  - Wir geben einfach das minItem zurück, das merken wir uns ja zu jedem Zeitpunkt explizit
  - Geht offensichtlich in $O(1)$ Zeit

Bucket Queues 6/9

UNI FREIBURG

- deleteMin()
  - Das minItem aus seinem Bucket löschen

    ```
    minItem.removeFromBucket()
    ```

  - Falls dieser Bucket jetzt leer, so weit im Feld nach rechts gehen, bis man die nächste nicht-leere Liste findet, und dort ein neues minItem wählen

    + Abbruchbedingung

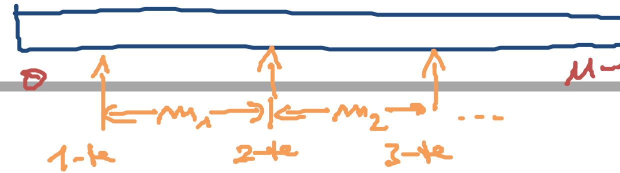
    ```
    minKey = minItem.key;
    while(buckets[minKey].empty()) { minKey++; }
    minItem = buckets[minKey].first();  // or any item there.
    ```

deleteMin ... löscht (1,"X")
deleteMin ... löscht (1,"S")

neues minItem (oder (4,"A") ginge auch)

M=6

Bucket Queues 7/9



- Laufzeitanalyse deleteMin
  - Ein einzelnes deleteMin kann bis zu $\Theta(M)$ dauern
  - Aber sei m_i die Anzahl Schleifendurchläufe für das i-te deleteMin, dann ist $\Sigma\, m_i = O(M)$

    Man geht immer nur weiter nach rechts in dem Feld, nie nach links, und das Feld ist nur M groß

Ohne Monotonie im worst case $\Theta(n \cdot M)$, z.B. bei

insert(0)
insert(M-1)
deleteMin ... $\Theta(M)$ Zeit
insert(0)
deleteMin ... $\Theta(M)$ Zeit
insert(0)
usw.

Bucket Queues 8/9

UNI FREIBURG

$M = 1000$
zu Zeitpunkt X:
Keys: 507, 507, 509, 512, 512
$\Rightarrow \Delta = 6$
(der range [507, 512] enthält $\Delta = 6$ ganze Zahlen, nicht 5)

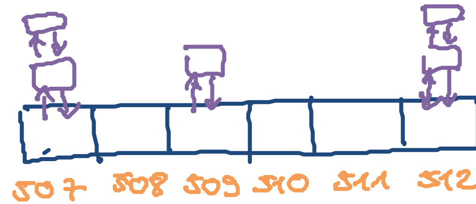
- Laufzeit insgesamt
  - Mit einer Bucket Queue lässt sich also eine beliebige Folge von n Operationen in Zeit $O(n + M)$ bearbeiten

    Für $M = O(n)$ ist das durchschnittlich $O(1)$ pro Operation

- Platzverbrauch
  - Die beschriebene Variante braucht $\Theta(n + M)$ Platz
  - Oft ist es in Anwendungen so, dass zu jedem Zeitpunkt die in der PW gespeicherten Schlüssel um maximal $\Delta << M$ auseinander liegen, also in einem Intervall

    [minItem.key .. minItem.key + $\Delta - 1$]
  - Dann geht es auch mit $\Theta(n + \Delta)$ Platz ... Ü9, Aufgabe 2 !

Bucket Queues 9/9



UNI FREIBURG

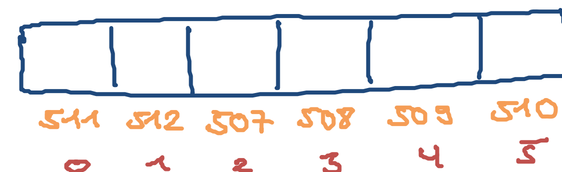
$\Delta = 6$

Indexe für $x \in [507, 512]$:
$x - 507$

- Hilfestellung Ü9, Aufgabe 2
  - Keys über den Gesamtverlauf aus dem Bereich 0 .. M – 1
  - Zu jedem Zeitpunkt muss man aber nur Keys aus dem Bereich [minItem.key .. minItem.key + Δ – 1] speichern

    Dafür reicht eigentlich ein Feld der Größe Δ
  - **Problem:** minItem.key verändert sich dabei, es kann allerdings immer nur größer werden
  - **Idee:** man muss das erste Element aus dem Bereich ja nicht unbedingt an Position 0 abspeichern, sondern kann an einer beliebigen Position i anfangen ... und am Ende des Feldes fängt man einfach vorne wieder an

Indexe für $x \in [507, 512]$?
$(x - 507 + 2) \bmod 6$



Radix Heaps 1/2

UNI FREIBURG

- Motivation + Grundidee
  - Was wenn $M >> n$, zum Beispiel $M = \Theta(n^2)$
  - Dann ist die Laufzeit der Bucket Queue $\Theta(n^2)$, also schlechter als der gewöhnliche binäre Heap
  - **Problem:** man hat dann ein großes Feld buckets der Größe $M >> n$ und die meisten Einträge sind leer

    Es kann ja maximal n nicht-leere Einträge geben
  - **Idee:** viele Einträge zu einem zusammenfassen, so dass man lange Folgen von nicht-leeren Einträgen einfach überspringen kann

Radix Heaps 2/2

UNI FREIBURG

- Laufzeit (im Vergleich zu Bucket Queues)
  - Falls die Keys aus dem Bereich $[0 .. M - 1]$ sind, dann:

    Bucket Queues: Zeit $O(n + M)$

    Radix Heaps: Zeit $O(n \cdot \log M)$
  - Falls die gespeicherten Keys zu jedem Zeitpunkt in $[\text{minItem.key} .. \text{minItem.key} + \Delta - 1]$ liegen, dann:

    Bucket Queues: Zeit $O(n \cdot \Delta)$

    Radix Heaps: Zeit $O(n \cdot \log \Delta)$

Literatur / Links

UNI FREIBURG

- Monotone ganzzahlige Prioritätswarteschlangen
  - In Mehlhorn/Sanders:

    10.5 Monotone Integer Priority Queues

    10.5.1 Bucket Queues