Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 9b, Mittwoch, 28. Juni 2017
(Bucket Queues)

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

UNI FREIBURG

Blick über die Vorlesung heute

- Inhalt
  - Prioritätswürgeschlangen — Fortsetzung von gestern
  - Fibonacci Heaps — nur kurz zur Komplexität
  - Bucket Queues — effizienter als Heaps für einen wichtigen Spezialfall
  - ÜB9, Aufgabe 2: Implementierung einer BucketQueue

    Sie brauchen dazu u.a. eine LinkedList

    Für Python haben wir eine Implementierung auf dem Wiki bereitgestellt, für Java können Sie die "java.util.LinkedList" benutzen, für C++ die "std::list"

Evolution des Homo Sapiens

- Was war die entscheidende Mutation und wann?
  - Ein spannendes Thema!
  - Das möchte ich jetzt nicht hier in der Voraufzeichnung (ohne Publikum) abhandeln
  - Deswegen auf nächste Woche (Mittwoch) verschoben

Fibonacci Heaps

- 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

Bucket Queues 1/14

- Monotone ganzzahlige Prioritätswarteschlangen
  - Eine Folge von Operationen auf einer PW heißt **monoton ganzzahlig**, wenn gilt:

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

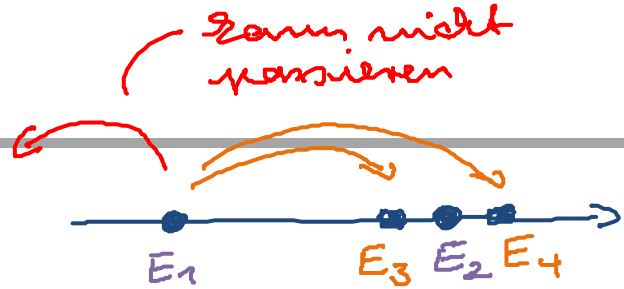
    Das Minimum der gespeicherten Elemente wird durch neue Operationen **nie kleiner**, höchstens größer

    Man beachte: die Argumente der Operationen müssen dazu nicht unbedingt monoton steigen

    MIN=7 MIN=7 MIN=7 MIN=12 MIN=12 ...
  - Positivbeispiel: ins(7), ins(18), ins(12), delMin(), ins(14), ...

    MIN=7 MIN=7 MIN=7 MIN=12 MIN=10
  - Negativbeispiel: ins(7), ins(18), ins(12), delMin(), ins(10), ...

Bucket Queues 2/14



- Anwendungen dafür
  - 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 (Zeit oder Entfernung)

    Mehr dazu nächste Woche ...

Bucket Queues 3/14

Keys

- 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"

    SORT ⟹ (0, "U"), (0, "O"), (1, "A"), ...

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

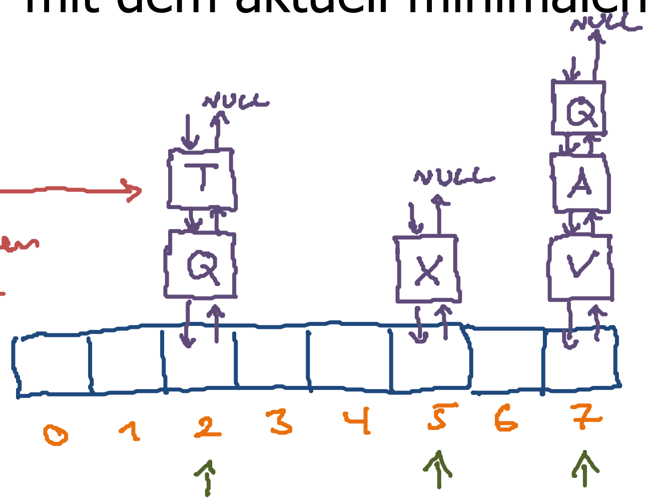
Bucket Queues 4/14

- Datenstruktur
  - Ein Feld von verketteten Listen bietet sich an, dann:

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

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

minItem
(wir könnten auch das Q nehmen)



In den verketteten Listen stehen nur die Values.
Der Key ist ja schon implizit in der Position der Liste

← KEYS

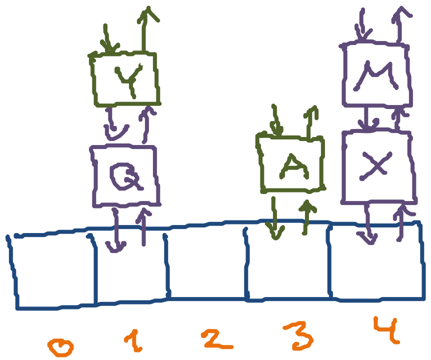
die haben alle Key 2 ... ...

Bucket Queues 5/14

- Die Operation insert
  - Einfach neues Element an entsprechende Liste anhängen

    Geht mit einer verketteten Liste pro Eintrag in Zeit O(1)
  - minItem muss nicht neu gesetzt werden, weil monoton

    Außer natürlich beim Einfügen des allerersten Elementes



insert (3, A)

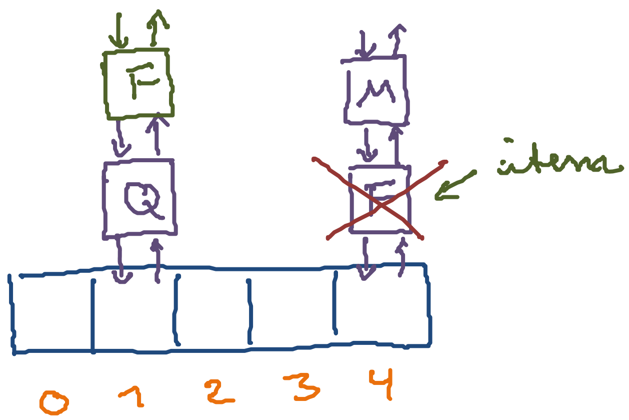
insert (1, Y)

...

Bucket Queues 6/14

- Die Operation changeKey
  - Aus der alten Liste löschen und in die neue einfügen

    Geht mit einer verketteten Liste ebenfalls in O(1) Zeit
  - minItem muss nicht neu gesetzt werden, weil monoton



changeKey (item, 1)

. . .

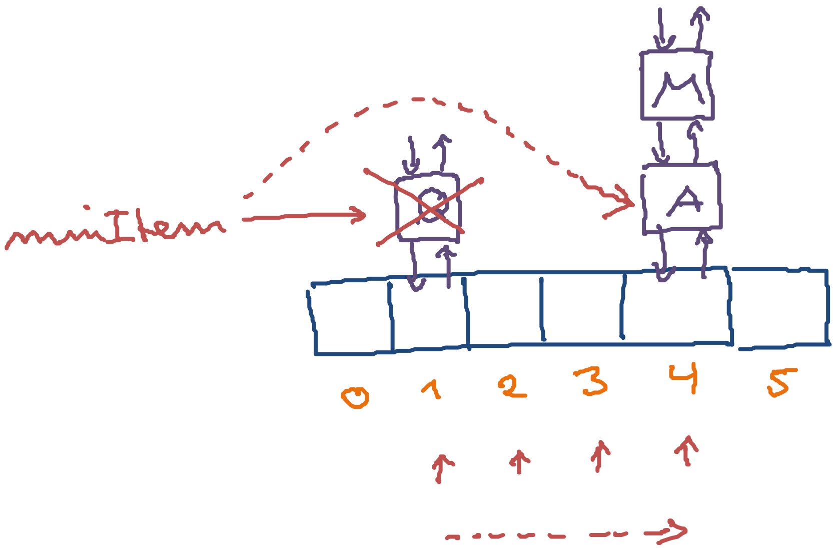
Bucket Queues 7/14

- Die Operation getMin
  - Wir geben einfach das minItem zurück, das merken wir uns ja zu jedem Zeitpunkt explizit

    Geht offensichtlich in O(1) Zeit

Bucket Queues 8/14

- Die Operation deleteMin
  - Das minItem aus seinem Bucket löschen
  - 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

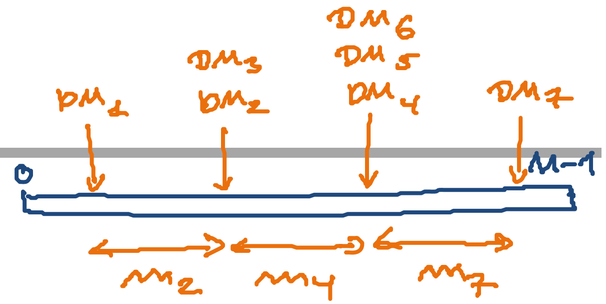


deleteMin()

...

KEYS

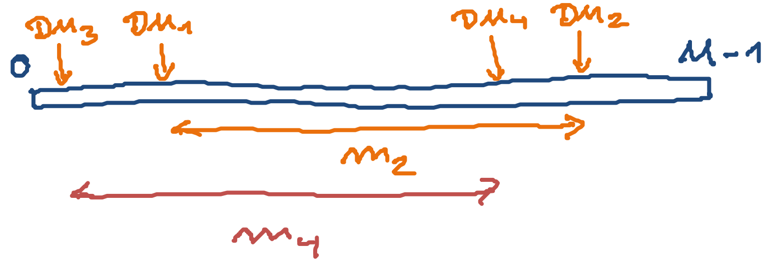
Bucket Queues 9/14



- Laufzeitanalyse deleteMin
  - Ein einzelnes deleteMin kann bis zu Θ(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 die Monotonie müsste man bei jedem deleteMin im worst case das ganze Feld durchgehen, um das neue Minimum zu finden

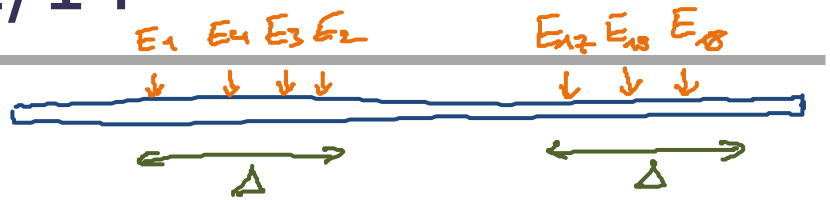


Bucket Queues 10/14

- 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

Bucket Queues 11/14



- 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 … und $O(n \cdot \Delta)$ Zeit

    Das ist die **Zusatzaufgabe** zu Ü9.2

    Das geht sehr elegant mit wenig zusätzlichem Code, aber etwas tricky … Hilfestellung dazu auf der nächsten Folie

Bucket Queues 12/14

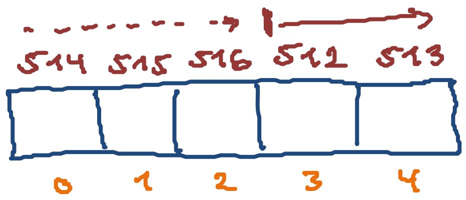
Δ ist gegeben

- Hilfestellung zur Zusatzaufgabe
  - 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 … allerdings verändert sich minItem.key im Laufe der Zeit

  - **Idee:** man muss das erste Element ja nicht unbedingt an Position 0 abspeichern, sondern kann an einer beliebigen Position i anfangen

    Am Ende des Feldes dann wieder vorne anfangen



Möchte abspeichern:
512 – 516

Δ = 5

Bucket Queues 13/14

- Verbesserung für $M \gg n$ zum Beispiel $M = \Theta(n^2)$
  - Dann ist die Laufzeit der Bucket Queue $\Theta(n^2)$

    Also schlechter als der gewöhnliche binäre Heap
  - Problem dabei: man hat dann ein großes Feld buckets der Größe $M \gg n$ 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

    Die resultierende Datenstruktur nenn man **Radix Heaps**

Bucket Queues 14/14

- Laufzeit von Radix Heaps
  - Falls die Keys aus dem Bereich $[0 .. M-1]$ sind, dann:

    Bucket Queues: $O(n + M)$

    Radix Heaps: $O(n \cdot \log M)$

  - Falls die gespeicherten Keys zu jedem Zeitpunkt in $[\text{minItem.key} .. \text{minItem.key} + \Delta - 1]$ liegen, dann:

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

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

Literatur / Links

- Monotone ganzzahlige Prioritätswarteschlangen
  - In Mehlhorn/Sanders:

    10.5 Monotone Integer Priority Queues

    10.5.1 Bucket Queues