

## 6.5 Weitere Arten wichtiger Datenstrukturen

### Dynamische Datenstrukturen:



Samuel W. Bent:

*Dynamic weighted data structures*

TR STAN-CS-82-916, Department of Computer Science,  
Stanford University, 1982

### Persistente Datenstrukturen:



J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan:

*Making data structures persistent*

Proceedings of the 18<sup>th</sup> Annual ACM Symposium on Theory  
of Computing (STOC), pp. 109–121 (1986)

### Probabilistische Datenstrukturen:



William Pugh:

*Skip Lists: A Probabilistic Alternative to Balanced Trees*

Commun. ACM, 33(6):668-676, 1990

## 7. Radix-basierte Priority Queues

### 7.1 Buckets

Eine relativ einfache Möglichkeit, Vorrangwarteschlangen zu implementieren, stellen **Buckets** dar. Diese Implementierung beinhaltet einen Array von Buckets, wobei der  $i$ -te Bucket alle Elemente  $x$  mit dem Schlüssel  $k(x) = i$  enthält. Sobald der Schlüssel eines Elements sich ändert, wird das Element vom alten Bucket entfernt und entsprechend dem neuen Schlüssel in dem neuen Bucket eingefügt.

Dazu müssen folgende Annahmen erfüllt sein:

- Schlüssel sind ganzzahlig
- Zu jedem Zeitpunkt gilt für die zu speichernden Elemente:

$$\text{größter Schlüssel} - \text{kleinster Schlüssel} \leq C$$

Diese Bedingungen sind zum Beispiel beim Algorithmus von Dijkstra erfüllt, falls die Kantengewichte natürliche Zahlen  $\leq C$  sind.

## 7.1.1 1-Level-Buckets

1-Level-Buckets bestehen aus:

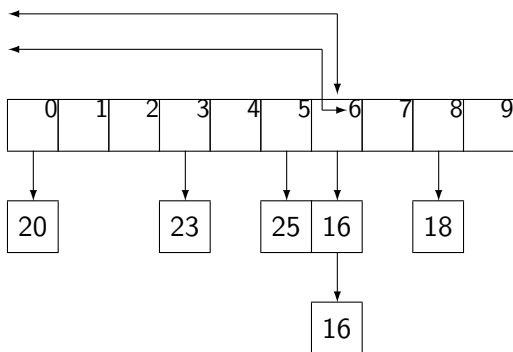
- einem Array  $b[0..C]$  zur Aufnahme der Buckets. Jedes  $b_i$  enthält einen Pointer auf die Liste der Elemente im Bucket  $i$ ;
- einer Zahl  $minvalue$ , die gleich dem kleinsten gespeicherten Schlüssel ist;
- einer Zahl  $0 \leq minpos \leq C$ , die den Index des Buckets mit dem kleinsten Schlüssel enthält, und
- der Zahl  $n$  der gespeicherten Elemente.

$C = 9$

$minvalue = 16$

$minpos = 6$

$n = 6$



Wie bei jeder Vorrangwarteschlange müssen drei Operationen unterstützt werden:

- *Insert(x)*: fügt das Element  $x$  in die Vorrangwarteschlange ein. Falls der Schlüssel des neuen Elements kleiner als der *minvalue* ist, werden *minpos* und *minvalue* aktualisiert.
- *ExtractMin*: liefert und löscht eines der kleinsten Elemente der Vorrangwarteschlange (falls das Element **das** kleinste ist, müssen *minpos* und *minvalue* noch aktualisiert werden).
- *DecreaseKey(x, k)*: verringert Schlüssel des Elements  $x$  auf den Wert  $k$  (falls nötig, werden *minpos* und *minvalue* aktualisiert).

Dazu kommt noch eine Initialisierung *Initialize*.

Für 1-Level-Buckets sehen die Operationen wie folgt aus:

① *Initialize(x)*:

$n := 0$ ;  $minvalue := \infty$

reserviere Platz für  $b$  ( $C$  sei bekannt)

initialisiere  $b$

② *Insert*:

füge  $x$  in  $b[k(x) \bmod (C + 1)]$  ein

$n := n + 1$

**if**  $k(x) < minvalue$  **then**

**co**  $x$  ist jetzt das Element mit dem kleinsten Schlüssel **oc**

$minpos := k(x) \bmod (C + 1)$

$minvalue := k(x)$

**fi**

Für 1-Level-Buckets sehen die Operationen wie folgt aus:

- ③ *ExtractMin*: Entferne ein beliebiges Element aus  $b[\text{minpos}]$

**co** wir nehmen  $n > 0$  an **oc**

extrahiere beliebiges Element in  $b[\text{minpos}]$

$n := n - 1$

**if**  $n > 0$  **then**

**co** suche neues Element mit kleinstem Schlüssel **oc**

**while**  $b[\text{minpos}]$  ist leer **do**

$\text{minpos} := (\text{minpos} + 1) \bmod (C + 1)$

**od**

$\text{minvalue} :=$  Schlüssel eines Elements in  $b[\text{minpos}]$

**else**

$\text{minvalue} := \infty$

**fi**



Für 1-Level-Buckets sehen die Operationen wie folgt aus:

- ④  $DecreaseKey(x, k)$ : verringert Schlüssel des Elements  $x$  auf den Wert  $k$

entferne  $k(x)$  aus Bucket  $k(x) \bmod (C + 1)$

$k(x) := k$

füge  $x$  in  $b[k(x) \bmod (C + 1)]$  ein

**if**  $k(x) < minvalue$  **then**

$minpos := k(x) \bmod (C + 1)$

$minvalue := k(x)$

**fi**

Bei geeigneter Implementierung der Buckets, z.B. als doppelt verkettete Listen, gilt:

## Satz 62

Die worst-case (reellen) Kosten sind  $\mathcal{O}(1)$  für *Insert* und *DecreaseKey*, und sie sind  $\mathcal{O}(C)$  für *Initialize* und *ExtractMin*.

### Beweis:

Wird  $x$  am Anfang der Liste eingefügt, so müssen bei *Insert* nur einige Zeiger umgehängt sowie  $n$ ,  $minpos$  und  $minvalue$  angepasst werden, was wieder nur ein paar Zeigeroperationen sind. Die Aktualisierung von  $n$ ,  $minpos$  und  $minvalue$  benötigt auch nur konstante Zeit. Für das Finden des nächstkleinsten Elements müssen aber möglicherweise alle weiteren Buckets betrachtet werden, im schlimmsten Falle  $C$ . Da bei *DecreaseKey* das Element  $x$  direkt übergeben wird, sind neben dem Einfügen nur wenige Zeigeroperationen und je eine Zuweisung an  $n$  und  $k(x)$  nötig.  $\square$

## 7.1.2 2-Level-Buckets

Bei einem großen Wertebereich der zu speichernden Schlüssel, d.h. bei großem  $C$ , und einer geringen Anzahl tatsächlich abgelegter Datenelemente sind 1-Level-Buckets in zweifacher Hinsicht ungünstig:

- Das Feld  $b$  belegt statisch Speicherplatz der Größe  $\Theta(C)$ , obwohl nur ein kleiner Teil davon wirklich gebraucht wird.
- Der Zeitbedarf für ein *ExtractMin* nähert sich der worst-case-Komplexität  $\Theta(C)$ , da der nächste nicht-leere Bucket ziemlich weit entfernt sein kann.

2-Level-Buckets versuchen diesen Problemen mit folgender Idee abzuhelpfen:

Es gibt einen Array  $btop$ , bestehend aus  $B + 1$  **top**-Buckets, mit  $B := \lceil \sqrt{C + 1} \rceil$ .

Zu jedem Bucket  $i$  in  $btop$  gibt es noch einen weiteren Array  $bbot_i$ , der ebenfalls aus  $B$  **bottom**-Buckets besteht.

$bbot_i$  nimmt Elemente auf, deren Schlüssel im Intervall  $[iB, (i + 1)B - 1]$  liegen. Um ein Element in einen Bucket einzufügen, wird zuerst der passende Bucket in  $btop$  gesucht. Dann wird in dem dazugehörigen  $bbot_i$  das Element (wie bei 1-Level-Buckets) eingefügt.

Um sowohl Platz als auch Zeit zu sparen, kann man durch leichte Modifizierung mit einem einzigen Array von Bottom-Buckets auskommen:

Wir verwenden zwei Arrays (Top-Buckets und Bottom-Buckets). Dabei enthält der Array der Top-Buckets in

$$B = \left\lceil \sqrt{C + 1} \right\rceil$$

Buckets die meisten Elemente in grob vorsortierter Form (ein Top-Bucket wird nicht benützt!), nur die Elemente mit den kleinsten Schlüsseln werden im Array für die Bottom-Buckets, der die Länge

$$B = \left\lceil \sqrt{C + 1} \right\rceil$$

hat, dann endgültig sortiert vorgehalten.

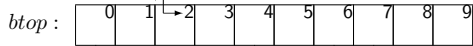
2-Level-Buckets können mit folgenden Elementen aufgebaut werden:

- Der Array  $btop[0..B]$  nimmt in jedem Bucket Elemente mit Schlüsseln aus einem Teilintervall der Länge  $B$  auf; die Schlüssel stammen aus allen Teilintervallen außer dem niedrigsten.
- Der Array  $bbot[0..B - 1]$  nimmt in jedem Bucket Elemente mit genau einem Schlüssel auf; die Schlüssel stammen nur aus dem niedrigsten Teilintervall.
- $valtop$  ist die linke Intervallgrenze des niedrigsten Teilintervalls.
- $postop$  enthält den Index des Buckets in  $btop$  für das niedrigste Teilintervall (das aber in  $bbot$  gespeichert wird).
- $nbot$  ist die Anzahl der Elemente in  $bbot$ .
- $n$  ist die Anzahl der Elemente in  $bbot$  und  $btop$ .

$C = 80 \quad (\Rightarrow B = 9)!$

$valtop = 6$

$postop = 2$



86

32

48

57

26

59



$nbot = 4$

7

11

13

7

Dabei gilt:

- $btop[(postop + i) \bmod B]$  enthält alle Elemente  $x$  mit

$$valtop + iB \leq k(x) < valtop + (i + 1)B, \text{ wobei } 1 \leq i \leq B;$$

- $bbot[i]$  enthält alle Elemente  $x$  mit

$$k(x) = valtop + i, \text{ wobei } 0 \leq i \leq B - 1.$$

Dazu ist allerdings folgende Annahme zu machen:

Hat ein *ExtractMin* ein Element mit Schlüssel  $k$  zurückgeliefert, werden bis zum nächsten Aufruf von *ExtractMin* nur Elemente mit Schlüsseln  $\geq valtop$  eingefügt. Dies stellt sicher, dass ein Element aus *bbot* nie nach *btop* wandert, und ist z.B. in Dijkstra's Algorithmus für kürzeste Pfade erfüllt.



## Lemma 63

*Zu jedem Zeitpunkt haben alle Schlüssel Platz in der Datenstruktur.*

### Beweis:

Am meisten Platz wird benötigt, wenn der kleinste Schlüssel ganz rechts in *bbot* gespeichert oder dort entfernt worden ist. Er hat dann den Wert  $valtop + B - 1$ . Der größte Schlüssel, der nun vorkommen kann und also Platz finden muss, ist  $valtop + B - 1 + C$ .

## Beweis (Forts.):

$$\begin{aligned} & \text{größtmöglicher Schlüssel in } btop \\ &= valtop + (B + 1) \cdot B - 1 \\ &= valtop + B + B \cdot B - 1 \\ &= valtop + B + \lceil \sqrt{C + 1} \rceil \lceil \sqrt{C + 1} \rceil - 1 \\ &\geq valtop + B + C + 1 - 1 \\ &> valtop + B - 1 + C \\ &= \text{größtmöglicher erlaubter Schlüssel} \end{aligned}$$



**Annahme:** Vor dem ersten *ExtractMin* werden nur Elemente  $x$  mit  $0 \leq k(x) \leq C$  eingefügt.

i) *Initialize:*

$valtop := postop := nbot := n := 0$   
initialisiere  $btop, bbot$

ii) *Insert(x)*:

überprüfe Invarianten;

$i := \left( \left\lfloor \frac{k(x) - \text{valtop}}{B} \right\rfloor + \text{postop} \right) \bmod (B + 1)$

**if**  $i = \text{postop}$  **then**

füge  $x$  in  $\text{bbot}[k(x) - \text{valtop}]$  ein

$\text{nbot} := \text{nbot} + 1$

**else**

füge  $x$  in  $\text{btop}[i]$  ein

**fi**

$n := n + 1$

### iii) *ExtractMin*

**co** suche kleinstes Element in *bbot* **oc**

$j := 0$ ; **while**  $bbot[j] = \emptyset$  **do**  $j := j + 1$  **od**

entferne ein (beliebiges) Element aus  $bbot[j]$  und gib es zurück

$nbot := nbot - 1$ ;  $n := n - 1$

**if**  $n = 0$  **then return fi**

**if**  $nbot = 0$  **then**

**while**  $btop[postop] = \emptyset$  **do**

$postop := (postop + 1) \bmod (B + 1)$

$valtop := valtop + B$

**od**

**while**  $btop[postop] \neq \emptyset$  **do**

        entferne beliebiges Element  $x$  aus  $btop[postop]$

        füge  $x$  im  $bbot[k(x) - valtop]$  ein

$nbot := nbot + 1$

**od**

**fi**

iv) *DecreaseKey*( $x, k$ )

entferne  $x$  aus seinem momentanen Bucket

falls nötig, aktualisiere  $nbot$

$k(x) := k$

überprüfe Invarianten;

$i := \left( \left\lfloor \frac{k(x) - valtop}{B} \right\rfloor + postop \right) \bmod (B + 1)$

**if**  $i = postop$  **then**

füge  $x$  in  $bbot[k(x) - valtop]$  ein

$nbot := nbot + 1$

**else**

füge  $x$  in  $btop[i]$  ein

**fi**

## Lemma 64

Die worst-case (reelle) Laufzeit bei 2-Level-Buckets ist für Insert und DecreaseKey  $\mathcal{O}(1)$ , für ExtractMin  $\mathcal{O}(n + \sqrt{C})$  und für Initialize  $\mathcal{O}(\sqrt{C})$ .

### Beweis:

- Insert:  $\mathcal{O}(1) + 1 = \mathcal{O}(1)$
- DecreaseKey:  $\mathcal{O}(1)$
- ExtractMin

$$\mathcal{O} \left( \begin{array}{l} \sqrt{C} + \sqrt{C} + \#Elemente \\ btop \\ \downarrow \\ bbot \end{array} \right)$$

□

Setze Potenzial := Anzahl der Elemente in Buckets in  $b_{top}$ .

## Satz 65

Die amortisierten Kosten für 2-Level-Buckets sind  $\mathcal{O}(1)$  bei Insert und DecreaseKey und  $\mathcal{O}(\sqrt{C})$  bei ExtractMin.

### Beweis:

- Insert: s.o.
- DecreaseKey: s.o.
- ExtractMin:

$$\mathcal{O} \left( \begin{array}{c} \sqrt{C} + \#Elemente \\ \downarrow \\ b_{top} \\ \downarrow \\ b_{bot} \end{array} \right) - \#Elemente \begin{array}{c} b_{top} \\ \downarrow \\ b_{bot} \end{array} = \mathcal{O}(\sqrt{C})$$

□



### 7.1.3 $k$ -Level-Buckets

Die Verallgemeinerung von 2-Level-Buckets führt zu  $k$ -Level-Buckets. Diese bestehen dann aus  $k$  Arrays der Länge  $\lceil \sqrt[k]{C} \rceil$ . Dadurch lassen sich die Speicher- und Zeitkomplexität weiter verbessern, der Implementierungsaufwand steigt jedoch stark an.