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 abzuhelfen:

Es gibt einen Array $btop$, bestehend aus $B := \lceil \sqrt{C+1} \rceil$ top-Buckets.
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 Interval $[iB, (i+1)B - 1]$ 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

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

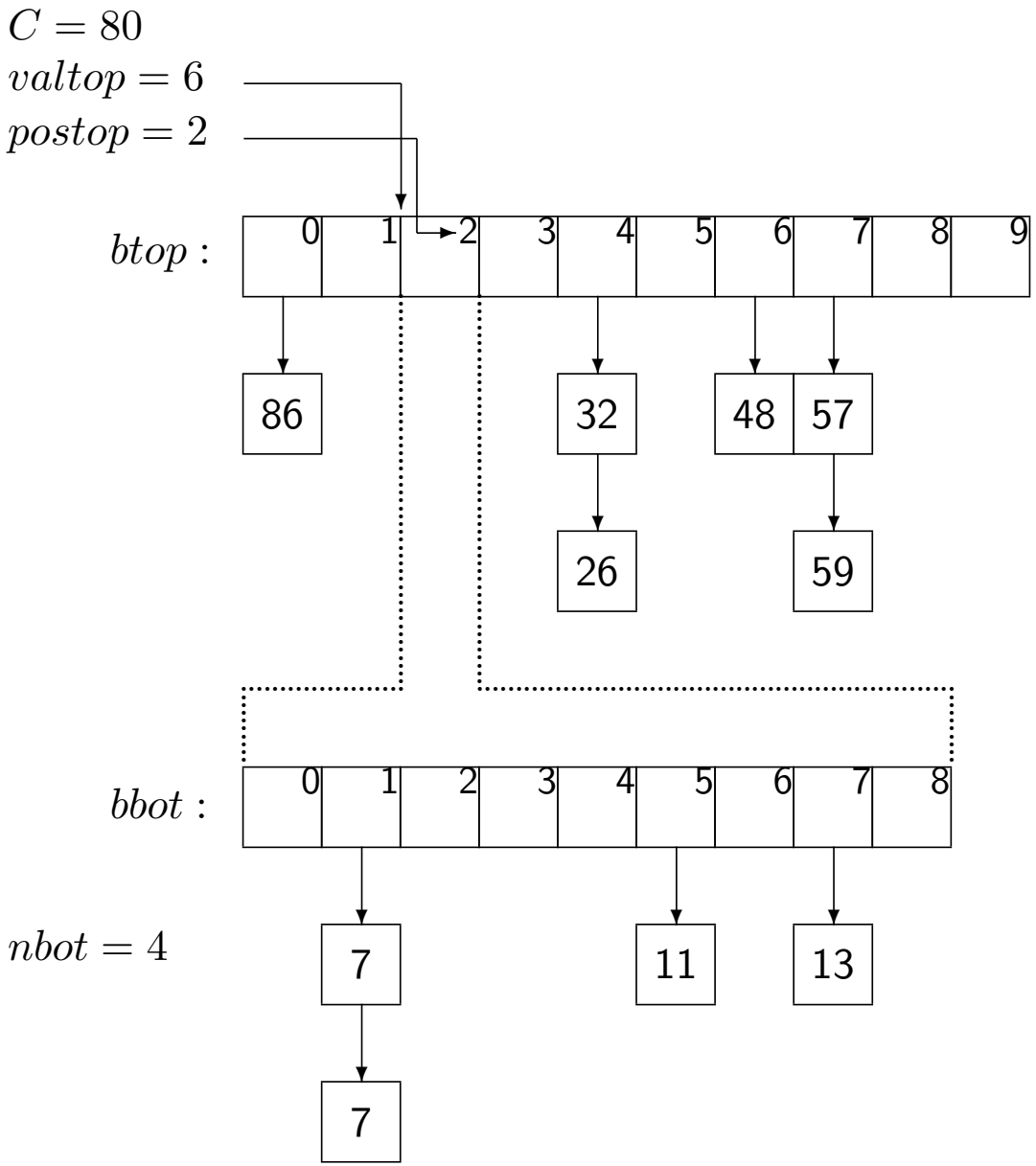
Buckets die meisten Elemente in grob vorsortierter Form, nur die Elemente mit den kleinsten Schlüsseln werden im Array für die Bottom-Buckets, der die Länge

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

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 Intervall der Länge B auf; die Schlüssel stammen aus allen Intervallen 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.
- $valtop$ ist die linke Intervallgrenze des niedrigsten Intervalls.
- $postop$ enthält den Index des Buckets in $btop$ für das niedrigste Intervall (das aber in $bbot$ gespeichert wird).
- $nbot$ ist die Anzahl der Elemente in $bbot$.
- n ist die Anzahl der Elemente in $bbot$ und $btop$.


$C = 80$
$valtop = 6$
$postop = 2$
$btop$:
0 1 2 3 4 5 6 7 8 9
86
32
48
57
26
59
$bbot$:
0 1 2 3 4 5 6 7 8
$nbot = 4$
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.):

größtmöglicher Schlüssel in $btop$

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

□

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;

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

if $i = postop$ **then**

füge x in $bbot[k(x) - valtop]$ ein

$nbot := nbot + 1$

else

füge x in $btop[i]$ ein

fi

$n := n + 1$

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*

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

□

Setze Potenzial := Anzahl der Elemente in Buckets in $btop$.

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*:

$$\mathcal{O}\left(\sqrt{C} + \#\text{Elemente}_{\substack{btop \\ \downarrow \\ bbot}}\right) - \#\text{Elemente}_{\substack{btop \\ \downarrow \\ bbot}} = \mathcal{O}(\sqrt{C})$$

□