

7.1.3 k -Level-Buckets

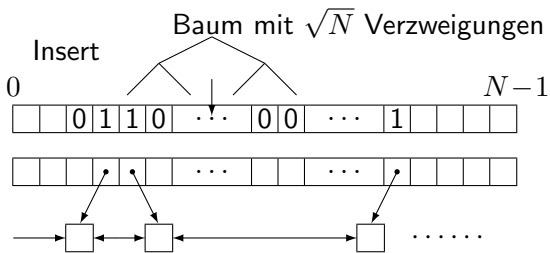
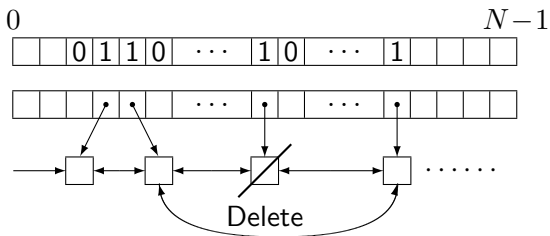
Die Verallgemeinerung von 2-Level-Buckets führt zu k -Level-Buckets. Diese bestehen dann aus k Arrays der Länge $\lceil \sqrt[k]{C} \rceil$. Dadurch lassen sich die Speicher- und Zeitkomplexität weiter verbessern, der Implementierungsaufwand steigt jedoch stark an.

7.2 van Emde Boas-Priority Queues

Universum U , $|U| = N$, $U \subset \mathbb{N}$.

Hier: $U = \{0, 1, \dots, N - 1\}$.

Wir untersuchen hier eine bessere Priority Queue für den Fall $m \geq \log N$.



Notation

Sei

$$k \in \mathbb{N}, k \geq 2$$

$$k' = \left\lceil \frac{k}{2} \right\rceil$$

$$k'' = \left\lfloor \frac{k}{2} \right\rfloor$$

$$x \in [0..2^k - 1]$$

(x hat $\leq k$ Bits)

$$x' = \left\lfloor \frac{x}{2^{k''}} \right\rfloor$$

(x' vordere Hälfte von x)

$$x'' = x \bmod 2^{k''}$$

(x'' hintere Hälfte von x)

Sei

$$S = \{x_1, \dots, x_m\} \subseteq [0..2^k - 1].$$

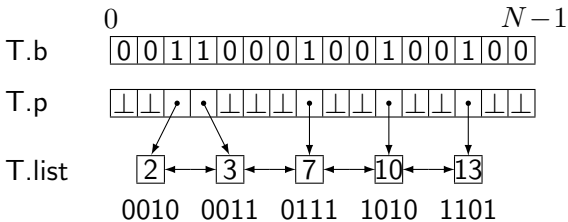
Definition 66

Eine k -Struktur T für S besteht aus:

- 1 der Zahl $T.size = |\{x_1, \dots, x_m\}| = |S| = m$;
- 2 einer doppelt verketteten Liste $T.list$, die die Elemente von S in aufsteigender Reihenfolge enthält;
- 3 einem Bitvektor $T.b[0..2^k - 1]$ mit $T.b[i] = \begin{matrix} 0 \\ 1 \end{matrix}$, falls $\begin{matrix} i \notin S \\ i \in S \end{matrix}$ einem Zeiger (Pointer) Vektor $T.p[0 \dots 2^k - 1]$. Falls $T.b[i] = 1$, dann zeigt $T.p[i]$ auf i in der Liste aus 2.
- 4 einer k' -Struktur $T.top$ und einem Feld $T.bottom[0 \dots 2^{k'} - 1]$ von k'' -Strukturen. Falls $m = 1$, dann $T.top$, $T.bottom$ und die zugehörigen k'' -Strukturen leer, $T.size = 1$. $T.list = \{x\}$, der Bitvektor wird nicht benötigt. Falls $m > 1$, dann ist $T.top$ eine k' -Struktur für die durch $\{x'_1, x'_2, \dots, x'_m\}$ gegebene Menge, und für jedes $y \in [0 \dots 2^{k'} - 1]$ ist $T.bottom[y]$ eine k'' -Struktur für die Menge $\{x''_i; 1 \leq i \leq m; y = x'_i\}$

Beispiel 67

$k = 4$, $S = \{2, 3, 7, 10, 13\}$, $T.size = 5$:



T.top ist 2-Struktur für $\{0, 1, 2, 3\}$

T.bottom[0] ist eine 2-Struktur für $\{2, 3\}$

T.bottom[1] ist eine 2-Struktur für $\{3\}$

T.bottom[2] ist eine 2-Struktur für $\{2\}$

T.bottom[3] ist eine 2-Struktur für $\{1\}$

Operation $Succ(x)$ findet $\min\{y \in S; y > x\}$ in der k -Struktur T .

if $k = 1$ **or** $T.Size \leq 2$ **then**

naive Suche

elif $x \geq \max$ in $T.list$ **then**

return $Succ(x)$ gibt's nicht

else

$$x' := \left\lfloor \frac{x}{2^{k''}} \right\rfloor;$$

$$x'' := x \bmod 2^{k''};$$

if $T.top.b[x'] = 1$ **and** $x'' < \max\{T.bottom[x']\}$ **then**

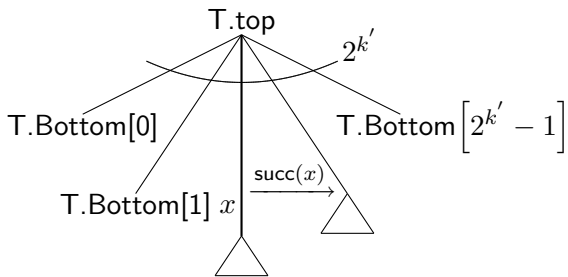
return $x' \cdot 2^{k''} + Succ(x'', T.bottom[x'])$

else

$z' := Succ(x', T.top)$; **return** $z' \cdot 2^{k''} + \min\{T.bottom[z']\}$

fi

fi



Kosten:

$$T(k) \leq c + T\left(\left\lceil \frac{k}{2} \right\rceil\right) = \mathcal{O}(\log k).$$

Lemma 68

Die Succ-Operation hat Zeitbedarf $\mathcal{O}(\log k)$.

Beweis:

✓



Insert Operation:

- falls x bereits in Priority Queue, dann fertig
- bestimme $Succ(x, T)$, füge x davor ein
- bestimme x' und x''
- behandle die entsprechenden Unterstrukturen rekursiv:
 $Insert(x', T.top)$, $Insert(x'', T.bottom[x'])$ (nur ein nicht trivialer rekursiver Aufruf)

Zeitbedarf: naiv $\mathcal{O}(\log^2 k)$, Optimierung: das oberste $Succ$ tut alles $\Rightarrow \mathcal{O}(\log k)$.

Delete Operation:

Komplexität von *Delete* in k -Struktur: $\mathcal{O}(\log k)$

Kosten der Initialisierung: \sim Größe der Datenstruktur

Platzbedarf für k -Struktur: Sei $S(k)$ der Platzbedarf für eine k -Struktur. Dann gilt:

$$S(1) = c$$

$$S(k) = c2^k + S(k') + 2^{k'} S(k'') \text{ für } k \geq 2$$

Wir ersetzen zur Vereinfachung:

$$S(k) = c2^k + S\left(\frac{k}{2}\right) + 2^{\frac{k}{2}} S\left(\frac{k}{2}\right)$$

Lemma 69

$$S(k) = \mathcal{O}(2^k \cdot \log k)$$

Beweis:

Zeige: $S(k) := c' 2^k \log k$ funktioniert.

Für $k = 1$ ist das klar.

Beweis (Forts.):

Rekursionsschritt:

$$\begin{aligned}\text{Platz für } k\text{-Struktur} &\leq c2^k + c'2^{\frac{k}{2}}(\log k - 1) + 2^{\frac{k}{2}}c'2^{\frac{k}{2}}(\log k - 1) \\ &= c2^k + c'2^{\frac{k}{2}}(1 + 2^{\frac{k}{2}})(\log k - 1) \\ &= c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}} \log k + \log k - 2^{\frac{k}{2}} - 1) \\ &\leq c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}} \log k + \log k - 2^{\frac{k}{2}}) \\ &\leq c'2^k \log k, \text{ falls}\end{aligned}$$

$$c2^k + c'2^k \log k + c'2^{\frac{k}{2}} \log k - c'2^k \leq c'2^k \log k$$

$$\Leftrightarrow c'2^{\frac{k}{2}} \log k \leq (c' - c)2^k$$

$$\Leftrightarrow \frac{c' - c}{c'} \geq \frac{\log k}{\frac{k}{2}} \quad (\text{gilt für } c' \text{ groß genug!})$$



Satz 70

Sei $N = 2^k$, Universum $U = \{0, \dots, N - 1\}$. Wird eine Teilmenge $S \subseteq U$ durch eine k -Struktur dargestellt, dann benötigen die Operationen *Insert*, *Delete* und *FindMin* jeweils Zeit $\mathcal{O}(\log \log N)$, die Initialisierung Zeit $\mathcal{O}(N \log \log N)$. Der Platzbedarf ist ebenfalls $\mathcal{O}(N \log \log N)$.

Beweis:

s.o.



Literatur zu van Emde Boas-Priority Queue:



Kurt Mehlhorn:

Data structures and algorithms 1: Sorting and searching,
pp. 290–296,

EATCS Monographs on Theoretical Computer Science,
Springer Verlag: Berlin-Heidelberg-New York-Tokyo, 1984



P. van Emde Boas, R. Kaas, E. Zijlstra:

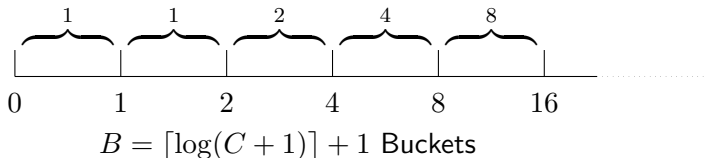
Design and implementation of an efficient priority queue

Math. Systems Theory 10 (1976), pp. 99–127

7.3 Radix-Heaps

Radix-Heaps stellen eine Möglichkeit zur effizienten Realisierung von Priority Queues dar, wobei ähnliche Randbedingungen wie bei den 2-Level-Buckets vorausgesetzt werden. Dabei wird die amortisierte Laufzeit der langsamsten Zugriffsfunktion im Vergleich zu diesen verbessert, nämlich von $\mathcal{O}(\sqrt{C})$ auf $\mathcal{O}(\log C)$. C bezeichne wie bei den 2-Level-Buckets die maximale Differenz zwischen zwei Schlüsseln im Heap.

Die Grundidee besteht darin, anstelle einer Hierarchie von Buckets konstanter Größe solche mit exponentiell zunehmender Größe zu verwenden. Somit sind nur noch $\mathcal{O}(\log C)$ Buckets zur Verwaltung der Elemente im Heap nötig. Wie bei 2-Level-Buckets hängt die Laufzeit der "teueren" Operationen direkt von der Anzahl der Buckets ab.



Randbedingungen:

- Schlüssel $\in \mathbb{N}_0$
- max. Schlüssel – min. Schlüssel stets $\leq C$
- Monotonie von *ExtractMin*

Implementierung:

- $B := \lceil \log(C + 1) \rceil + 1$
- Buckets $b[0..B - 1]$
- (untere) Schranken für Buckets $u[0..B]$
- Index $b_no[x]$ des aktuellen Buckets für x

Invarianten:

- $u[i] \leq \text{Schlüssel in } b[i] < u[i + 1]$
- $u[0] = 0, u[1] = u[0] + 1, u[B] = \infty;$
 $0 \leq u[i + 1] - u[i] \leq 2^{i-1}; \text{ für } i = 1, \dots, B - 1$

Operationen:

- 1 *Initialize*: Leere Buckets erzeugen und die Bucketgrenzen $u[i]$ entsprechend der Invariante ii) setzen:

for $i := 0$ **to** B **do** $b[i] := \emptyset$ **od**

$u[0] := 0$; $u[1] = 1$

for $i := 2$ **to** $B - 1$ **do**

$u[i] := u[i - 1] + 2^{i-2}$

od

$u[B] := \infty$

Operationen:

- ② $Insert(x)$: Um ein neues Element einzufügen, wird linear nach dem richtigen Bucket gesucht, beginnend beim letzten Bucket:

$i := B - 1$

while $u[i] > k(x)$ **do** $i := i - 1$ **od**

füge x in $b[i]$ ein

Operationen:

- ③ *DecreaseKey*(x, δ): Hierbei wird analog zur Prozedur *Insert* linear nach dem Bucket gesucht, in den das Element mit dem veränderten Schlüssel eingefügt werden muss. Der einzige Unterschied besteht darin, dass die Suche in diesem Fall beim aktuellen Bucket des Elements beginnen kann, da der Schlüssel erniedrigt wird und das Element deshalb nie in einen größeren Bucket wandern kann. Aus diesem Grund ist es jedoch notwendig, zu jedem Element x die dazugehörige aktuelle Bucketnummer in $b_no[x]$ zu speichern:

$i := b_no[x]$

entferne x aus $b[i]$

$k(x) := k := k(x) - \delta$; **co** überprüfe Invarianten! **oc**

while ($u[i] > k$) **do** $i := i - 1$ **od**

füge x in $b[i]$ ein

Operationen:

④ *ExtractMin:*

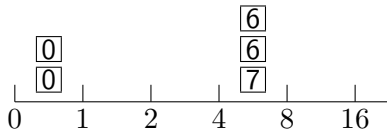
Es wird ein Element aus $b[0]$ entfernt (und zurückgegeben). Falls $b[0]$ nun leer ist, wird nach dem ersten nicht leeren Bucket $b[i]$, $i > 0$, gesucht und der kleinste dort enthaltene Schlüssel k festgestellt. Es wird $u[0]$ auf k gesetzt, und die Bucketgrenzen werden gemäß der Invarianten neu gesetzt. Danach werden die Elemente in $b[i]$ auf die davorliegenden Buckets verteilt:

Operationen:

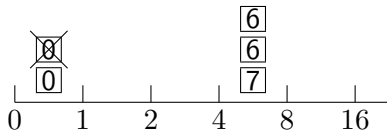
④ *ExtractMin*:

entferne (und gib zurück) ein beliebiges Element in $b[0]$
if #Elemente in Radix-Heap = 0 **then return**
if $b[0]$ nicht leer **then return**
 $i := 1$
while $b[i] = \emptyset$ **do** $i := i + 1$ **od**
 $k :=$ kleinster Schlüssel in $b[i]$
 $u[0] := k$
 $u[1] := k + 1$
for $j := 2$ **to** i **do**
 $u[j] = \min\{u[j - 1] + 2^{j-2}, u[i + 1]\}$
od
verteile alle Elemente aus $b[i]$ auf $b[0], b[1], \dots, b[i - 1]$

Beispiel 71

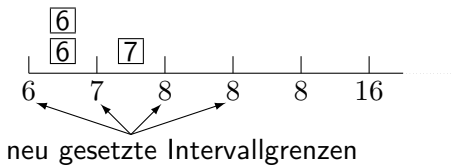
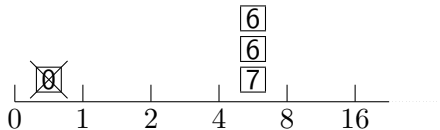


1. ExtractMin



Beispiel 71

2. ExtractMin



Korrektheit von ExtractMin:

Es gilt: $b[i] \neq \emptyset \Rightarrow$ Intervallgröße $\leq 2^{i-1}$.

Unterhalb $b[i]$ stehen i Buckets zur Verfügung mit Intervallen

$$[k, k + 1[, [k + 1, k + 2[, [k + 2, k + 4[, \dots, [k + 2^{i-2}, k + 2^{i-1}[$$

(wobei alle Intervallgrenzen jedoch höchstens $u[i + 1]$ sein können).

Da alle Schlüssel in $b[i]$ aus dem Intervall

$[k, \min\{k + 2^{i-1} - 1, u[i + 1] - 1\}]$ sind, passen sie alle in $b[0], b[1], \dots, b[i - 1]$.

Analyse der amortisierten Kosten:

$$\text{Potenzial: } c \cdot \sum_{x \in \text{Radix-Heap}} b_{no}[x],$$

für ein geeignetes $c > 0$.

Amortisierte Komplexität:

- i) *Initialize*: $\mathcal{O}(B)$
- ii) *Insert*: $\mathcal{O}(B)$
- iii) *DecreaseKey*: $1 + \mathcal{O}(\Delta b_{no}[x] - c \cdot \Delta b_{no}[x]) = \mathcal{O}(1)$
- iv) *ExtractMin*:
 $\mathcal{O}(i + |b[i]| + \sum_{x \in b[i]} \Delta b_{no}[x] - c \cdot \sum_{x \in b[i]} \Delta b_{no}[x]) = \mathcal{O}(1)$

Satz 72

Ausgehend von einem leeren Heap beträgt die worst-case (reelle) Laufzeit für k Insert-, l DecreaseKey und l ExtractMin-Operationen bei Radix-Heaps

$$\mathcal{O}(k \log C + l).$$

Beweis:

s.o.



7.3.1 Literatur:



R.K. Ahuja, K. Mehlhorn, J.B. Orlin, R.E. Tarjan:
Faster Algorithms for the Shortest Path Problem
J.ACM **37**, pp. 213–223 (1990)



B.V. Cherkassky, A.V. Goldberg, T. Radzig:
Shortest Path Algorithms: Theory and Experimental Evaluation
Math. Prog. **73**, pp. 129–174 (1996)



B.V. Cherkassky, A.V. Goldberg, C. Silverstein:
Buckets, Heaps, Lists and Monotone Priority Queues
Proc. 8th SODA, ACM, pp. 83–92 (1997)