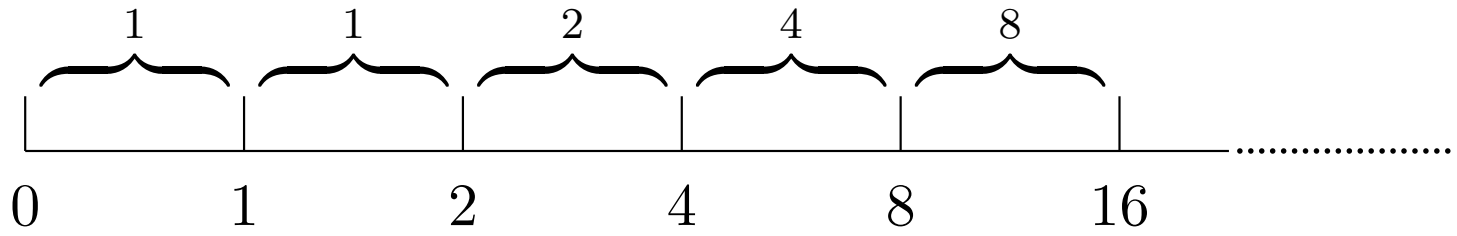
7.3 Radix-Heaps

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.



$B = \lceil \log(C+1) \rceil + 1$ Buckets

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 $b[0..B]$
- (untere) Schranken für Buckets $u[0..B+1]$
- Index $b\_no[x]$ des aktuellen Buckets für x

Invarianten:

i) $u[i] \leq$ Schlüssel in $b[i] < u[i+1]$

ii) $u[0] = 0$, $u[1] = u[0] + 1$, $u[B+1] = \infty$;
$0 \leq u[i+1] - u[i] \leq 2^{i-1}$; für $i = 1, \ldots,$ 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 **do**
    $u[i] := u[i-1] + 2^{i-2}$
od

Operationen:

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

   $i := B$
   while $u[i] > k(x)$ **do** $i := i - 1$ **od**
   füge x in $b[i]$ ein

Operationen:

3. *DecreaseKey*(x, k): 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 $b[i]$
   $k(x) := k$; **co** überprüfe Invarianten! **oc**
   while $(u[i] > k)$ **do** $i := i - 1$ **od**
   füge x in $b[i]$ ein

Operationen:

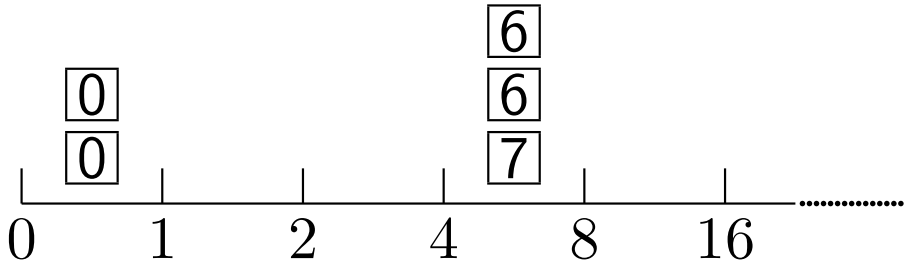
4. *ExtractMin*:
   Es wird ein Element aus $b[0]$ entfernt (und zurückgegeben). Falls $b[0]$ nun leer ist, wird nach dem ersten nicht leeren Bucket $b[i]$, $i > 0$, gesucht und der kleinste dort enthaltene Schlüssel k festgestellt. Es wird $u[0]$ auf k gesetzt, und die Bucketgrenzen werden gemäß der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

Operationen:

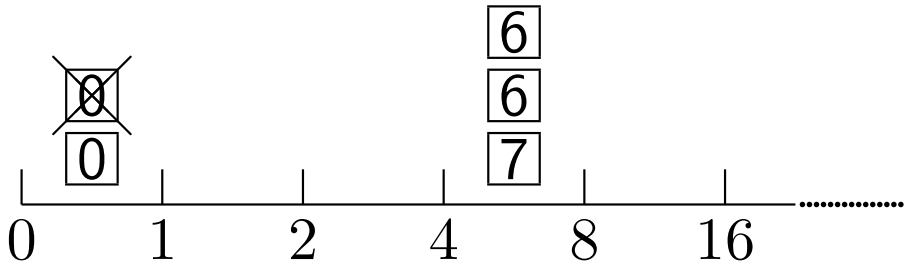
4. *ExtractMin*:

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**
$i := 0$
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**
$\quad u[j] = \min\{u[j-1] + 2^{j-2}, u[i+1]\}$
od
verteile alle Elemente aus $b[i]$ auf $b[0], b[1], \ldots, b[i-1]$

Beispiel 70

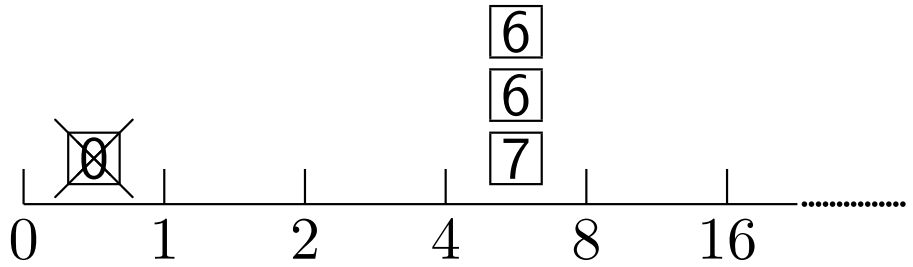


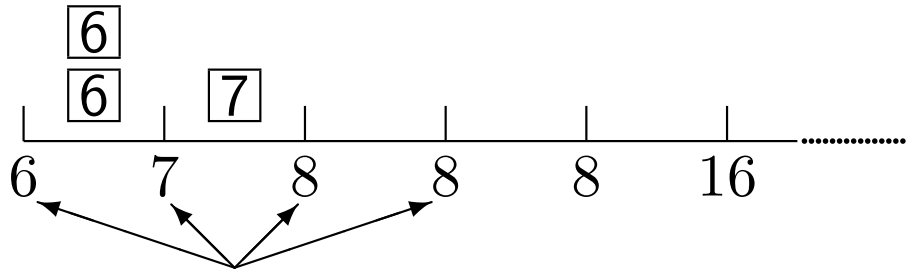
1. ExtractMin



Beispiel 70

2. ExtractMin





neu gesetzte Intervallgrenzen

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

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

(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
$b[0], b[1], \ldots, b[i-1]$.

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*:$\mathcal{O}(\Delta b\_no[x] - c \cdot \Delta b\_no[x] + 1) = \mathcal{O}(1)$

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 71

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

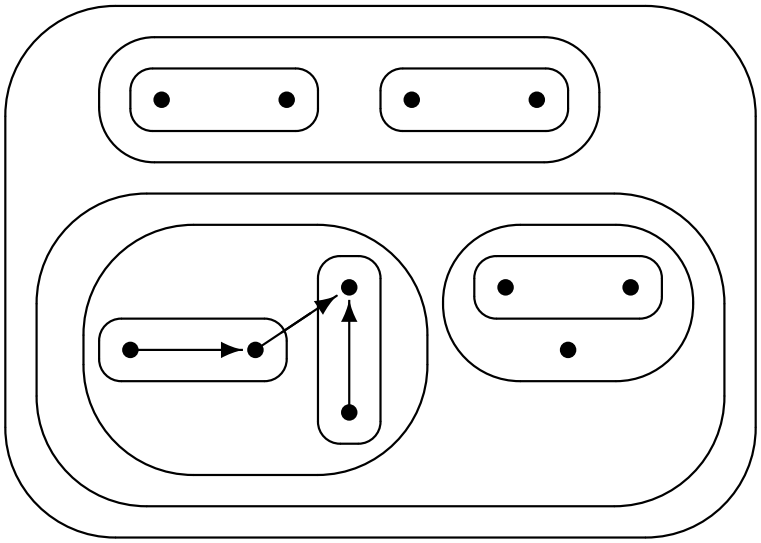
$$\mathcal{O}(k \log C + l)\,.$$

Beweis:

s.o. □

8. Union/Find-Datenstrukturen

8.1 Motivation

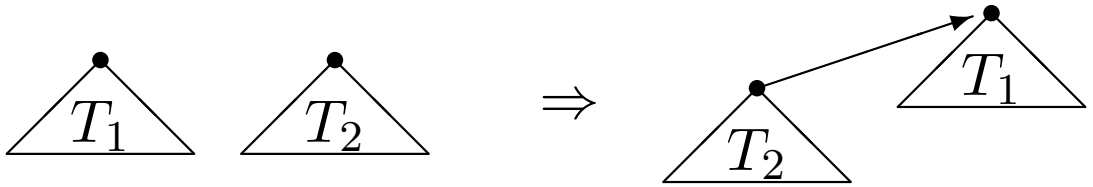


- *Union*(T_1, T_2): Vereinige T_1 und T_2
  $T_1 \cap T_2 = \emptyset$
- *Find*(x): Finde den Repräsentanten der (größten) Teilmenge, in der sich x gerade befindet.

8.2 Union-Find-Datenstruktur

8.2.1 Intrees

1. Initialisierung: $x \longrightarrow \bullet x$: Mache x zur Wurzel eines neuen (einelementigen) Baumes.
2. *Union*(T_1, T_2):



3. *Find*: Suche Wurzel des Baumes, in dem sich x befindet.

Bemerkung: Naive Implementation: worst-case-Tiefe $= n$

- Zeit für *Find* $= \Omega(n)$
- Zeit für *Union* $= \mathcal{O}(1)$

8.2.2 Gewichtete Union (erste Verbesserung)

Mache die Wurzel des kleineren Baumes zu einem Kind der Wurzel des größeren Baumes. Die Tiefe des Baumes ist dann $\mathcal{O}(\log n)$.

- Zeit für *Find* $= \mathcal{O}(\log n)$
- Zeit für *Union* $= \mathcal{O}(1)$

Es gilt auch: Tiefe des Baumes im worst-case:

$$\boxed{\Omega(\log n)}$$