Sei T ein Splay-Tree mit n Knoten. Falls sich die Gewichte der Knoten nicht ändern, ist die Verringerung des Potenzials durch eine Folge von Splay-Operationen Splay(x_j, T), $j = 1, \ldots, n$, beschränkt durch

$$\sum_{i=1}^{n} (\log W - \log w_i) = \sum_{i=1}^{n} \log \frac{W}{w_i} \,,$$

wobei

$$W := \sum_{i=1}^{n} w_i \,,$$

$$w_i = \text{Gewicht von } x_i \,.$$

Satz 58

Die gesamten Kosten für die m Zugriffe im Splay-Tree sind

$$\mathcal{O}((m+n)\log n + m)\,.$$

Beweis:

Wähle $w_i = \frac{1}{n}$ für alle Knoten. Dann sind die amortisierten Kosten für einen Zugriff $\leq 1 + 3\log n$, da $W = \sum_{i=1}^{n} w_i = 1$.

Die Verringerung des Potenzials ist

$\leq \sum_{i=1}^{n} \log \frac{W}{w_i} = \sum_{i=1}^{n} \log n = n \log n.$

Damit sind die reellen Kosten $\leq m(1 + 3\log n) + n\log n.$ □

Satz 59

Sei $q(i)$ die Anzahl der Zugriffe auf das Element x_i (in einer Folge von m Zugriffen). Falls auf jedes Element zugegriffen wird (also $q(i) \geq 1$ für alle i), dann sind die (reellen) Gesamtkosten für die Zugriffe

$$\mathcal{O}\left(m + \sum_{i=1}^{n} q(i) \cdot \log\left(\frac{m}{q(i)}\right)\right).$$

Beweis:

Setze das Gewicht des i-ten Knotens gleich $\frac{q(i)}{m}$.

$$\Rightarrow \mathrm{W} = \sum_{i=1}^{n} \frac{q(i)}{m} = 1\,.$$

Der Rest folgt wie zuvor. □

Satz 60

Betrachte eine Folge von Zugriffsoperationen auf eine n-elementige Menge. Sei t die dafür nötige Anzahl von Vergleichen in einem optimalen statischen binären Suchbaum. Dann sind die Kosten in einem (anfangs beliebigen) Splay-Tree für die Operationenfolge $\mathcal{O}(t + n^2)$.

Beweis:

Sei U die Menge der Schlüssel, d die Tiefe eines (fest gewählten) optimalen statischen binären Suchbaumes. Für $x \in U$ sei weiter $d(x)$ die Tiefe von x in diesem Suchbaum. Setze

$$tw(x) := 3^{d-d(x)}\,.$$

Sei T ein beliebiger Splay-Tree für U, $|U| =: n$.

$$bal(T) \leq \sum_{x\in U} r(x) = \sum_{x\in U} \log(3^{d-d(x)}) = \sum_{x\in U} (\log 3)(d-d(x)) =$$

$$= (\log 3) \sum_{x\in U} (d-d(x)) = \mathcal{O}(n^2);$$

$$\sum_{x\in U} tw(x) = \sum_{x\in U} 3^{d-d(x)} \leq \sum_{i=0}^{d} 2^i 3^{d-i} \leq 3^d \frac{1}{1-\frac{2}{3}} = 3^{d+1}$$

$$\Rightarrow \log \frac{tw(T)}{tw(x)} \leq \log \frac{3^{d+1}}{3^{d-d(x)}} = \log 3^{d(x)+1}.$$

Beweis (Forts.):

Damit ergibt sich für die amortisierten Kosten von Splay(x, T)

$$\mathcal{O}(\log \frac{tw(T)}{tw(x)}) = \mathcal{O}(d(x) + 1)\,.$$

Die amortisierten Kosten sind damit

$\leq c \cdot$ Zugriffskosten (#Vergleiche) im optimalen Suchbaum

(wo sie $d(x) + 1$ sind).

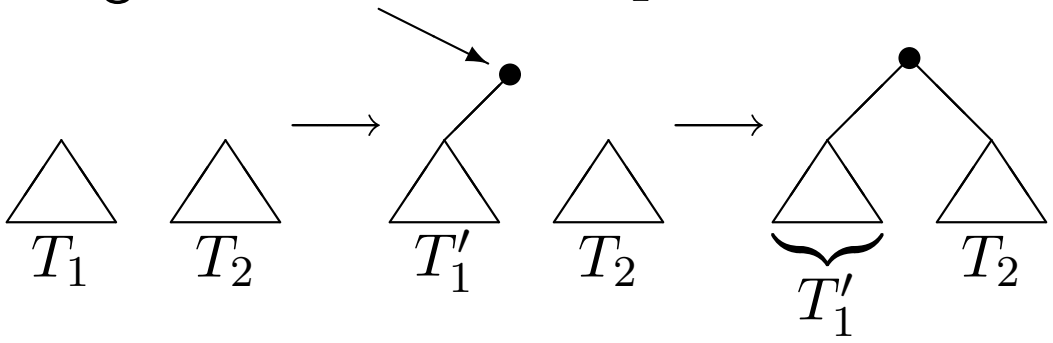
Die gesamten amortisierten Kosten für die Zugriffsfolge sind daher $\leq c \cdot t$.
Die reellen Kosten ergeben sich zu $\leq$ amort. Kosten + Verringerung des Potenzials, also $\mathcal{O}(t + n^2)$. □

6.4.3 Wörterbuchoperationen in Splay-Trees

Alle folgenden Operationen werden mit Hilfe von Splay implementiert.

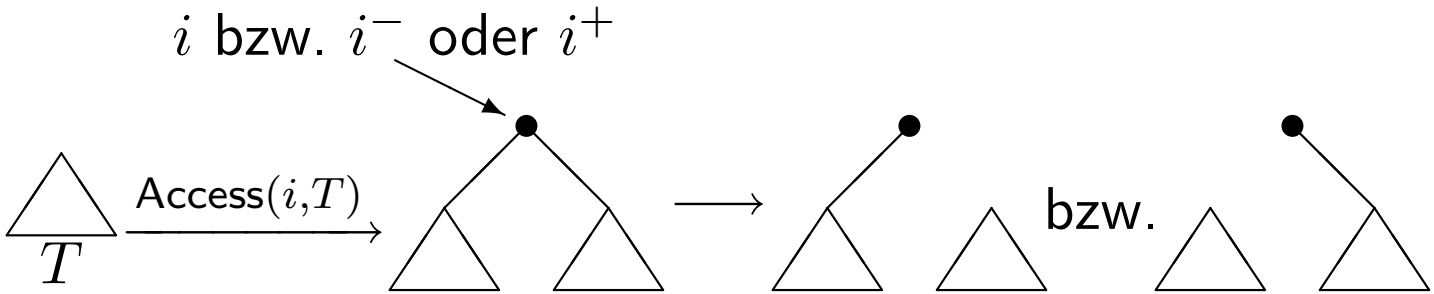
- *Access*(i, T): $\surd$ (siehe oben)
- *Join*(T_1, T_2):

  größtes Element in T_1

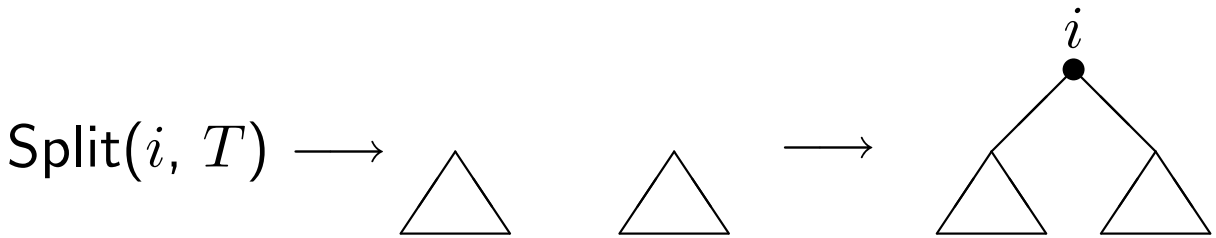


Beachte: Falls $x \in T_1$, $y \in T_2$, dann $x < y$.

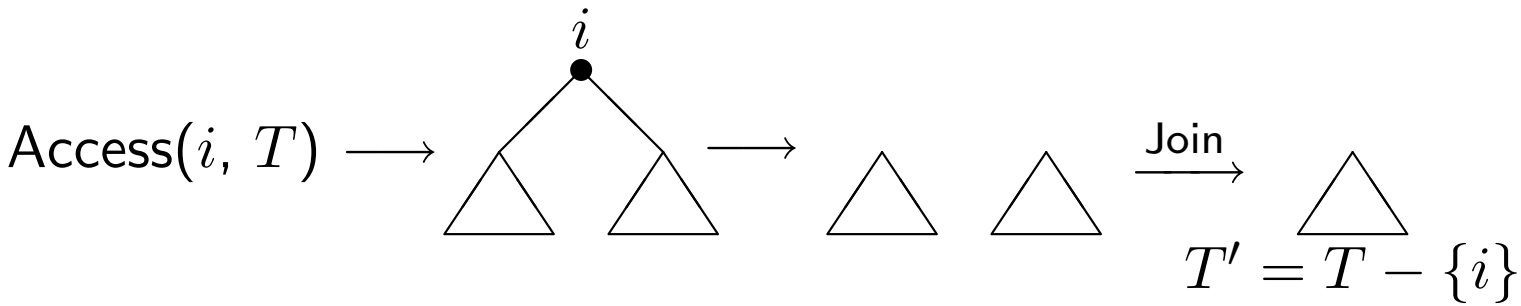
- *Split*(i, T):



- *Insert*(i, T):



- *Delete*(i, T):



Sei $i \in U$, T ein Splay-Tree. Dann bezeichnen wir mit i^- (bzw. i^+) den Vorgänger (bzw. den Nachfolger) von i in U (falls diese existieren). U ist ja total geordnet. Falls i^- bzw. i^+ undefiniert sind, so setzen wir $w(i^-) = \infty$ bzw. $w(i^+) = \infty$.

Weiterhin sei W das Gesamtgewicht aller an einer Wörterbuch-Operation beteiligten Knoten.

Satz 61

Für die amortisierten Kosten der Wörterbuch-Operationen in Splay-Trees gelten die folgenden oberen Schranken ($T,\ T_1,\ T_2 \neq \emptyset$):

$$\mathit{Access}(i,T): \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + \mathcal{O}(1), & \text{falls } i \in T \\ 3\log\left(\frac{W}{\min\{w(i^-),\ w(i^+)\}}\right) + \mathcal{O}(1), & \text{falls } i \notin T \end{cases}$$

$$\mathit{Split}(i,T): \begin{cases} 3\log\left(\frac{W}{w(i)}\right) + \mathcal{O}(1), & \text{falls } i \in T \\ 3\log\left(\frac{W}{\min\{w(i^-),\ w(i^+)\}}\right) + \mathcal{O}(1), & \text{falls } i \notin T \end{cases}$$

Satz 61

Für die amortisierten Kosten der Wörterbuch-Operationen in Splay-Trees gelten die folgenden oberen Schranken ($T,\ T_1,\ T_2 \neq \emptyset$):

$$\textit{Join}(T_1, T_2) : 3\log\left(\frac{W}{w(i)}\right) + \mathcal{O}(1),\ i \text{ maximal in } T_1$$

$$\underset{i \notin T}{\textit{Insert}(i, T)} : 3\log\left(\frac{W - w(i)}{\min\{w(i^-), w(i^+)\}}\right) + \log\left(\frac{W}{w(i)}\right) + \mathcal{O}(1)$$

$$\underset{i \in T}{\textit{Delete}(i, T)} : 3\log\left(\frac{W}{w(i)}\right) + 3\log\left(\frac{W - w(i)}{w(i^-)}\right) + \mathcal{O}(1),$$

falls i nicht minimal in T

Literatur zu Splay-Trees:

Daniel D. Sleator, Robert E. Tarjan:
Self-adjusting binary search trees
Journal of the ACM **32**(3), pp. 652–686 (1985)

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 18th 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:

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

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:

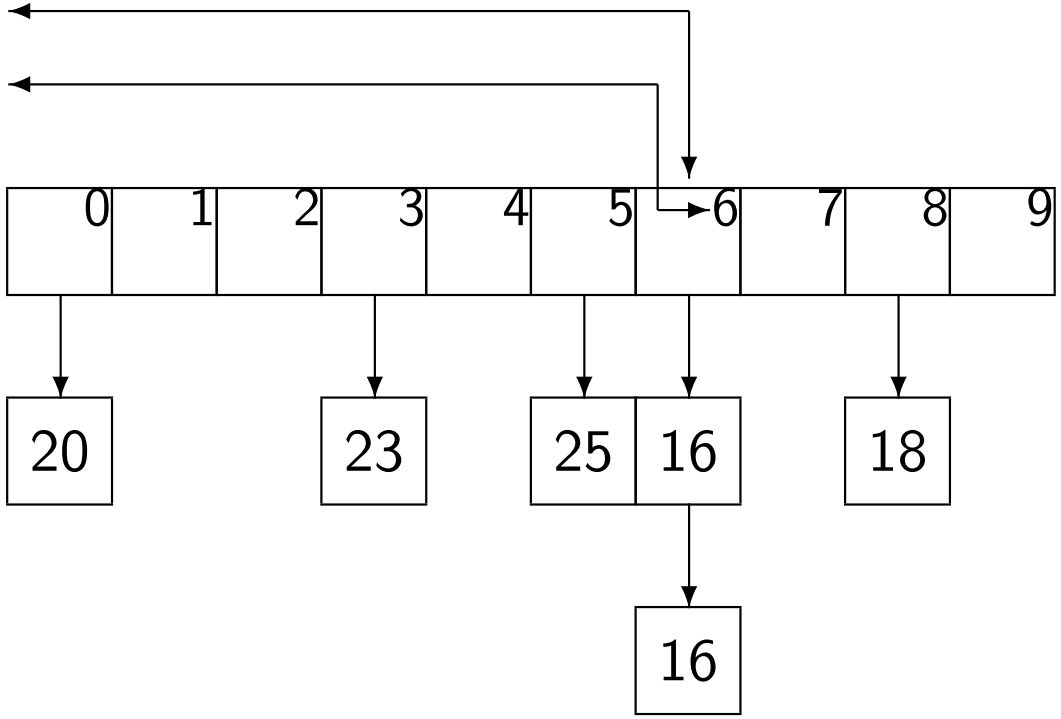
- einem Array $b[0..C]$ zur Aufnahme der Buckets. Jedes b_i enthält einen Pointer auf die Liste der Elemente im Bucket i;
- einer Zahl $minvalue$, die gleich dem kleinsten gespeicherten Schlüssel ist;
- einer Zahl $0 \leq minpos \leq C$, die den Index des Buckets mit dem kleinsten Schlüssel enthält,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

co wir nehmen $n > 0$ an **oc**
extrahiere beliebiges Element in $b[minpos]$
$n := n - 1$
if $n > 0$ **then**
    co suche neues Element mit kleinstem Schlüssel **oc**
    while $b[minpos]$ ist leer **do**
        $minpos := (minpos + 1) \bmod (C + 1)$
    od
    $minvalue :=$ Schlüssel eines Elements in $b[minpos]$
else
    $minvalue := \infty$
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. □