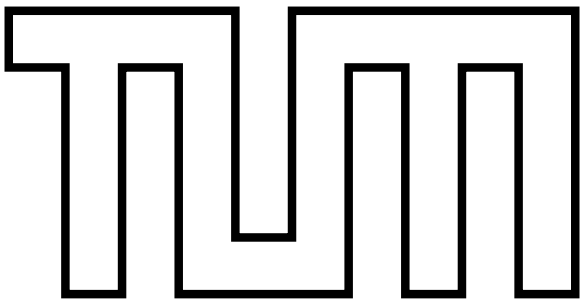


Technische Universität München
Institut für Informatik

Effiziente Algorithmen und Datenstrukturen WS 1997/98 - Kapitel 1

Stand: 10. März 1998

Dozent:

Prof. Dr. Ernst W. Mayr

Mitschrift:

Martin Stumpf
Thomas Schöpf
Martin Hans
Guido Wimmel

Inhaltsverzeichnis

1 Höhere Datenstrukturen **9**

1.1 Grundlegende Operationen . . . 9
1.2 Suchbäume, balancierte Bäume . . . 10
1.2.1 Höhenbalancierte Bäume . . . 10
1.2.1.1 (a,b)-Bäume . . . 10
1.2.1.2 Rot-Schwarz-Bäume . . . 12
1.2.1.3 AVL-Bäume . . . 14
1.2.2 Gewichtsbalancierte Bäume . . . 14
1.2.3 Hashing . . . 14
1.3 Priority Queues . . . 14
1.3.1 Wiederholung . . . 14
1.3.1.1 Binomialbäume . . . 14
1.3.1.2 Binomial Queues (binomial heaps) . . . 15
1.3.2 Fibonacci-Heaps . . . 16
1.3.2.1 Implementierung . . . 17
1.3.2.2 Amortisierte Kostenanalyse für Fibonacci-Heaps . . . 18
1.3.2.3 Eine $\mathcal{O}(\log\log N)$ Priority-Queue . . . 19
1.3.3 Sich selbst organisierende Datenstrukturen . . . 22
1.3.3.1 Sich selbst organisierende lineare Listen . . . 22
1.3.3.2 Splay-Trees als Suchbäume . . . 23
1.3.3.2.1 Einleitung . . . 23
1.3.3.2.2 Die Splaying-Operation . . . 24
1.3.3.2.3 Amortisierte Kostenanalyse . . . 25
1.3.3.2.4 Wörterbuchoperationen in Splay-Trees . . . 27
1.3.4 Weitere Arten von Datenstrukturen . . . 29

Kapitel 1

Höhere Datenstrukturen

1.1 Grundlegende Operationen

Menge S mit Ordnung $\leq$. Oft auch S als Schlüsselmenge. Ordnung auf den Schlüsseln $S \ni s \mapsto k(s)$.

1. Element(a, S): Ist $a \in S$?
2. Insert(a, S): $S := S \cup \{a\}$
3. Delete(a, S): $S := S - \{a\}$
   (a) Falls Ordung auf S (bzw. Schlüsseln für S):
   Delete_Min(S): Delete(a, S), wobei a minimal in S (bzw. $k(a)$ minimal)
4. Replace(a, a', S): $S := S - \{a\} \cup \{a'\}$
   (a) Decrease_Key(a, Δ, S): ersetze $a = \underbrace{(k, v)}_{\text{Schlüssel k}(v)}$ durch $a' = (k - \Delta, v)$
5. Union(S_1, S_2): $S_1 := S_1 \cup S_2$; falls $S_1 \cap S_2 = \emptyset$, dann Merge
6. Sei $S_i;\ i = 1, \ldots, r$ eine Ansammlung von disjunkten Mengen.
   Find(a): liefert $i \in \{1, \ldots, r\}$, falls $a \in S_i$, ansonsten undefiniert
7. Find_Min(S): minimales Element in S (falls S geordnet), bzw. Element in S mit minimalem Schüssel; undefiniert, falls $S = \emptyset$
8. Find_Max(S): analog
9. Split(S_i, a, S_j); if $a \in S_i$ then $S_j := \{b \in S_i;\ b > a\},\ S_i := \{b \in S_i;\ b \leq a\}$ (bzw. $k(b) > k(a),\ k(b) \leq k(a)$) else undefiniert fi
10. Concatenate(S_i, S_j): if Find_Max$(S_i) <$ Find_Min(S_j) then $S_i := S_i \cup S_j$ else undefiniert fi

Datenstrukturklasse	mindestens angebotene Funktionen	realisiert in
Wörterbuch (Dictionary)	Element(), Insert(), Delete()	Hashtable, balancierte Bäume
Vorrangwarteschlange (Priority Queue)	Find_Min(), Insert(), Delete(), [Element()]	balancierte, Leftist-, Binomialbäume
Mergeable heaps	Find_Min(), Insert(), Delete(), Merge()	2-3-Bäume, Binomialwälder, Leftist-Bäume
Concatenable queues	Find_Min(), Insert(), Delete(), Concatenate()	2-3-Bäume

1.2 Suchbäume, balancierte Bäume

1.2.1 Höhenbalancierte Bäume

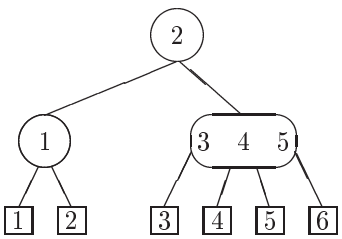
Suchbäume:

- extern: Elemente nur in den Blättern gespeichert
- intern: Elemente in allen Knoten

1.2.1.1 (a,b)-Bäume

Definition: Ein (a,b)-Baum ist ein externer Suchbaum, alle Blätter haben die gleiche Tiefe, die Anzahl der Kinder eines jeden internen Knoten ist $\leq$ b und $\geq$ a (Für die Wurzel ≥ 2), und es gilt $b \geq 2a-1$. Spezialfall: $b = 2a-1$ (B-Bäume (Bayer))

Beispiel:
Wir betrachten im folgenden (2,4)-Bäume.



Interne Knoten: -∞, k_1, ..., k_{d-1}, +∞

Lemma: Sei T ein (a,b)-Baum mit n Blättern und der Höhe h. Dann

$$(1) \quad 2a^{h-1} \leq n \leq b^h$$

$$(2) \quad \log_b n = \frac{\log n}{\log b} \leq h \leq 1 + \log_a \frac{n}{2}$$

Beweis:

(1) $n \leq b^h$: Maximal für vollständigen Baum-Verzweigungsgrad b. Minimal für Baum mit Grad der Wurzel = 2, alle anderen internen Knoten = a.

(2) aus (1).

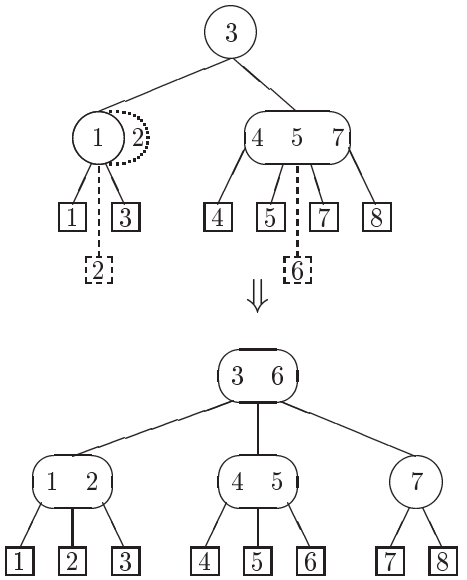
□

Operationen:

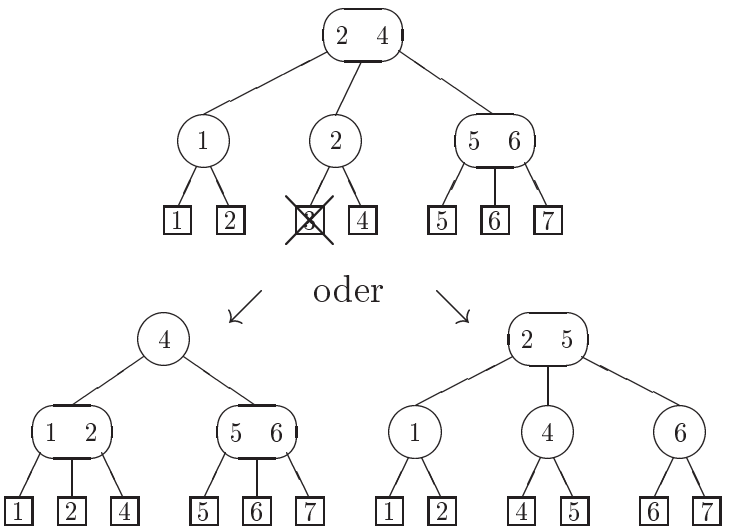
1. Element(x, S)
   $v :=$ Wurzel von T;
   <u>while</u> v ist nicht Blatt <u>do</u>
   $i := \min\{j;\ 1 \le j \le d(v),\ x \le k_j\}$;
   $v := i$-tes Kind von v;
   <u>od</u>;
   <u>if</u> $x = v$ <u>then</u> return (ja) <u>else</u> return (nein) <u>fi</u>;

   Zeitbedarf: $\mathcal{O}(\log n)$

2. Insert(x, S)
   Element, dann neues Blatt, rekursiv nach oben. Splitting von Knoten mit $> b$ Kindern.



3. Delete(x, S)



   Zeitbedarf: $\mathcal{O}(\log n)$

Bemerkungen:

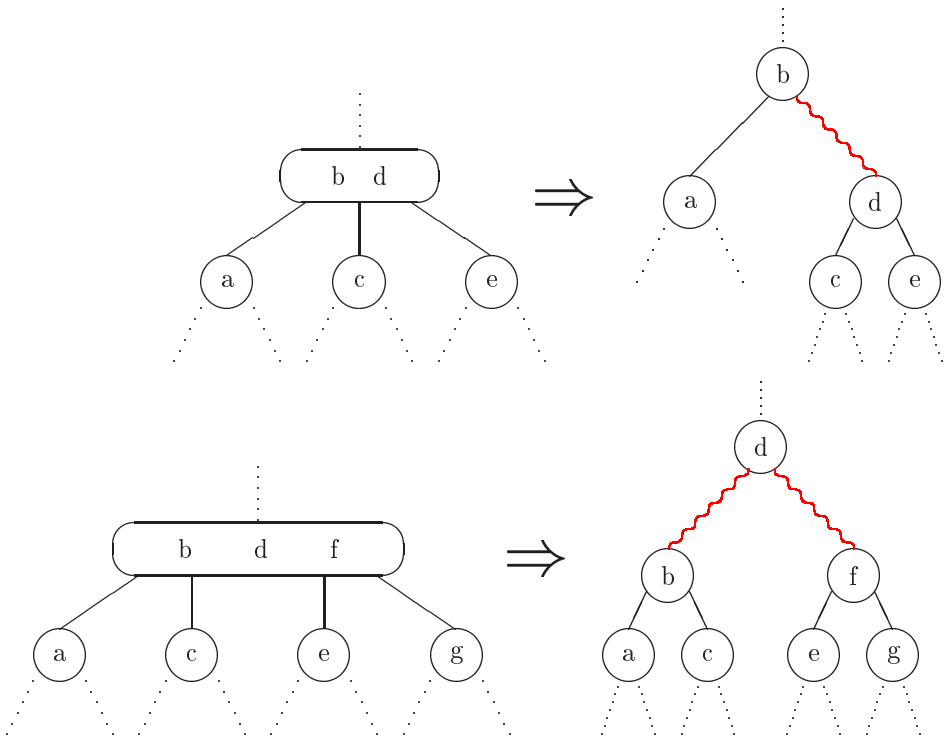
1. Zur Wahl von a:
   - Daten in RAM $\Rightarrow$ a klein, z.B. $a = 2$ oder $a = 3$.
   - Daten auf Platte $\Rightarrow$ a groß, z.B. $a = 100$.
2. Zur Wahl von b:
   $b \geq 2a$ liefert wesentlich bessere amortisierte Komplexität (ohne Beweis, siehe Mehlhorn).
3. Top-Down-Rebalancierung: $b \geq 2a$.
   Bei der Restrukturierung nach der Top-Down-Strategie, folgen wir wie gewohnt dem Pfad von der Wurzel zum gesuchten Blatt. Beim Einfügen stellen wir jetzt jedoch für jeden besuchten Knoten sicher, daß der Knoten weniger als b Kinder hat. Wenn der gerade betrachtete Knoten ein b-Knoten ist, dann spalten wir ihn sofort auf. Weil der Vater kein b-Knoten ist (das haben wir ja bereits sichergestellt) pflanzt sich der Aufspaltungsprozeß nicht nach oben hin fort. Insbesondere kann das neue Element ohne Probleme eingefügt werden, wenn die Suche das Blattniveau erreicht hat. Damit diese Strategie möglich ist, muß b mindestens $2a$ sein (und nicht $2a - 1$), da sonst nach der Spaltung nicht genügend Elemente für die beiden Teile vorhanden sind.
   Beim Löschen verfahren wir analog. Für jeden besuchten Knoten außer der Wurzel des (a, b)-Baumes stellen wir sicher, daß er mindestens $a + 1$ Kinder hat. Wenn der gerade betrachtete Knoten nur a Kinder hat, so versuchen wir zuerst ein Element des rechten oder linken Nachbarknoten zu stehlen. Sind beide Nachbarknoten vom Grad a, so verschmelzen wir unseren Knoten mit einem der beiden Nachbarn. Ist der Vater nicht die Wurzel, so hatte er aber vorher mindestens $a + 1$ Kinder, und dieser Verschmelzungsprozeß kann sich nicht nach oben fortsetzen. Andernfalls erniedrigt sich der Grad der Wurzel um 1, wenn die Wurzel Grad größer 2 hat, oder die alte Wurzel wird gelöscht und der soeben verschmolzene Knoten wird zur neuen Wurzel.
   Restrukturierung nach der Top-Down-Strategie sorgt nicht für eine bessere Laufzeit, sondern erleichtert die Synchronisation, wenn mehrere Prozesse gleichzeitig auf einen (a, b)-Baum zugreifen wollen. Bei herkömmlichen (a, b)-Bäumen schreiten die Such- und die Restrukturierungsoperationen in entgegengesetzten Richtungen fort, was dazu führt, daß sich oft Prozesse gegenseitig behindern (der eine will im Baum absteigen, der andere muß auf dem gleichen Pfad aufwärts restrukturieren). Bei der Top-Down-Strategie gibt es nur Operationsfolgen die den Baum hinabsteigen, mehrere Prozesse können so in einer Art Pipeline gleichzeitig einen Pfad mit kurzem Abstand begehen.

1.2.1.2 Rot-Schwarz-Bäume

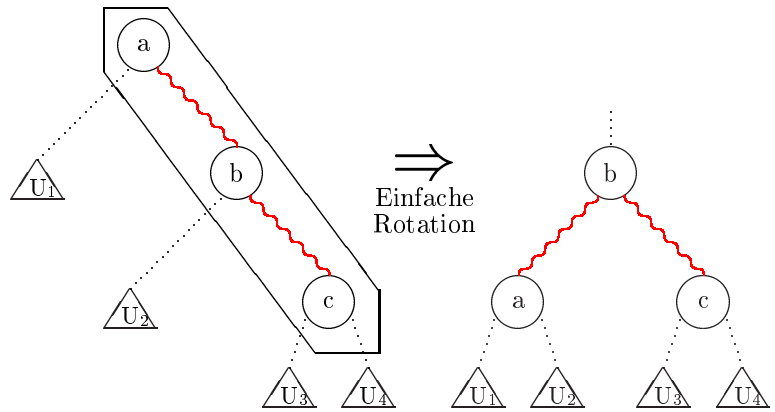
Definition: Rot-Schwarz-Bäume sind Binärbäume (jeder Knoten hat 0 oder 2 Kinder) mit roten und schwarzen Kanten, so daß gilt:

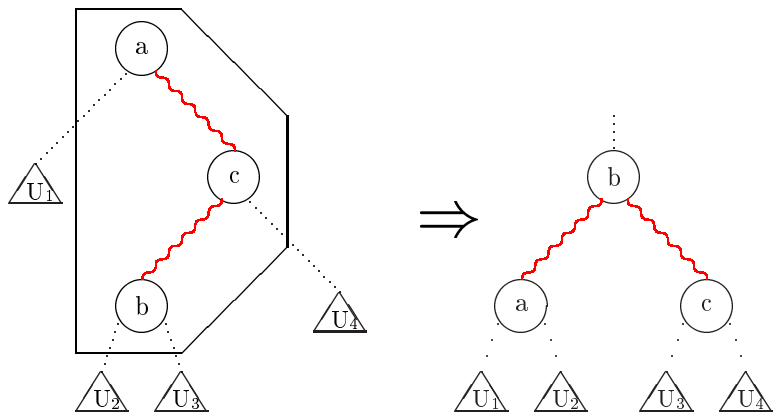
1. alle Blätter hängen an schwarzen Kanten (durchgezogene Linien)
2. alle Blätter haben die gleiche „Schwarztiefe“.
3. kein Pfad von der Wurzel zu einem Blatt enthält aufeinanderfolgende rote Kanten (gewellte Linien).

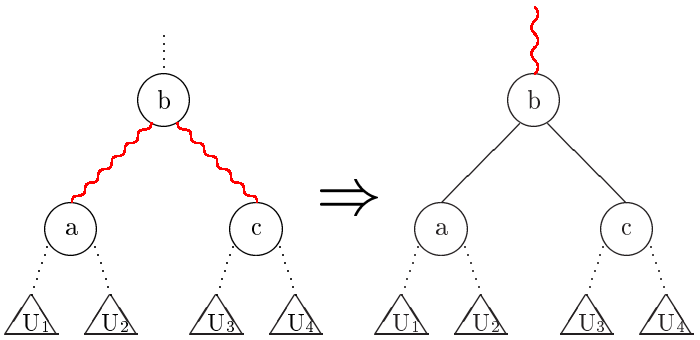
Rot-Schwarz-Bäume können zur Implementierung von (2,4)-Bäumen mittels Binärbäumen dienen.
Vorteil: Interne Knoten haben Verzweigungsgrad = 2



Rebalancierung:







Satz: Die Tiefe eines Rot-Schwarz-Baumes mit n Blättern ist $\mathcal{O}(\log n)$

Beweis: klar □

Korollar: Die Wörterbuch-Operationen benötigen auf Rot-Schwarz-Bäumen $\mathcal{O}(\log n)$ Zeit.

1.2.1.3 AVL-Bäume

Benannt nach Adelson, Velskii und Landis

Siehe

- Aho, Hopcroft, Ullmann
- Sedgewick/Flajolet
- Info IV Skript SS94, Prof. Mayr

In der Theorie: Sehr schöne Eigenschaften.
In der Praxis: Sehr aufwendig zu implementieren.

1.2.2 Gewichtsbalancierte Bäume

Siehe Mehlhorn, Bd. 1, S. 189 ff.

1.2.3 Hashing

Siehe Info IV Skript SS94, Prof. Mayr

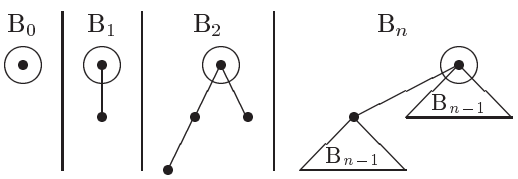
1.3 Priority Queues

Auch: Vorrangwarteschlangen, Heaps.

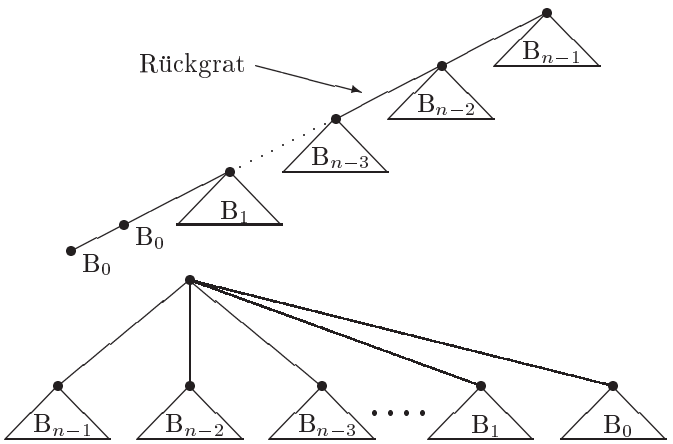
1.3.1 Wiederholung

Binomial-Warteschlangen.

1.3.1.1 Binomialbäume



Zwei (interessante) Dekompositionen:



Eigenschaften:

1. $|B_n| = 2^n \quad [= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1 + 1]$
2. Tiefe$(B_n) = n$
3. Anzahl der Knoten in B_n mit Tiefe $k = \binom{n}{k}$

1.3.1.2 Binomial Queues (binomial heaps)

- Daten(schlüssel) an allen Knoten, aus total geordnetem Universum.
- Heap-Bedingung: $\forall$ Knoten x mit Vater p(x) gilt: Das in p(x) gespeicherte Element ist $\leq$ dem in x gespeicherten (Min. heap).
- Größe n beliebig: Wald von $\leq \lceil \log(n+1) \rceil$ Binomialbäumen [ein Baum für jede 1 in der Binärdarstellung von n]. Doppelt verkettete Liste der Wurzeln, mit Zeiger auf Wurzel mit minimalem Element.
- Implementierung: Pointer für am weitesten links befindliches Kind, rechtes Geschwister und Vater

Operationen:

1. Verschmelzen (Mergen) zweier Binomial Heaps
   $B_i \cup B_i \Rightarrow B_{i+1}$, kleinere Wurzel wird Wurzel des B_{i+1}
   $B_i \cup B_j, i \neq j \Rightarrow$ Wald aus B_i und B_j.

   B_i, B'_j Binomialbäume der Ordnung i

   $F_1 = \{B_{i_1}, B_{i_2}, \ldots, B_{i_r}\} \cup \{B'_{j_1}, B'_{j_2}, \ldots, B'_{j_s}\} = F_2$
   $i_1 < i_2 < \ldots < i_r$
   $j_1 < j_2 < \ldots < j_s$

   Von kleinen zu großen Indizes, entsprechend der Schulmethode für die Addition von Binärzahlen (mit Übertrag), unter Benutzung der obigen Basisschritte.

   Zeitaufwand proportional zur Länge der beiden Listen und der Anzahl der neu geschaffenen Kanten. Die Anzahl der Kanten in einem Binomialwald mit n Knoten ist $n - \#_1(n)$ ($\#_1(n) \equiv$ Anzahl der Einsen in der Binärdarstellung von n).

Die Anzahl der neu geschaffenen Kanten ist daher

$$n_1 + n_2 - \#_1(n_1 + n_2) - (n_1 - \#_1(n_1)) - (n_2 - \#_1(n_2))$$

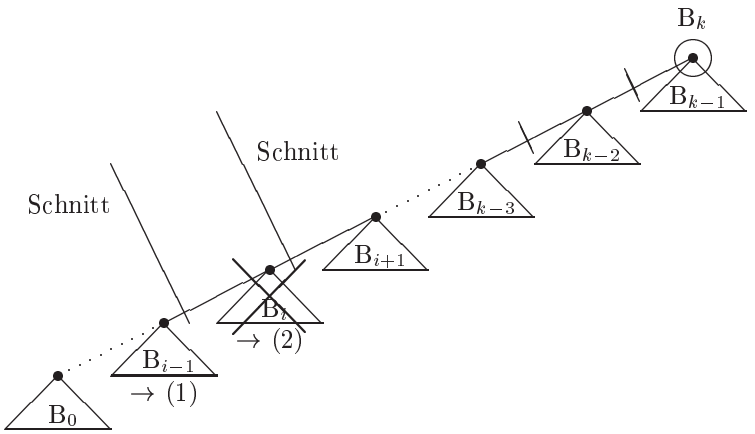
Zeitbedarf: $\mathcal{O}(\log(n_1 + n_2))$, wobei n_1 (bzw. n_2) die Anzahl der Knoten in F_1 (bzw. F_2) ist.

2. Insert(x): $\widehat{=}$ Verschmelzen mit B_0.
   Einfügen von n einzelnen Elementen der Reihe nach: Kosten $\mathcal{O}(n)$

3. Löschen:

   (a) Finde richtigen Baum

   (b)



   $\Rightarrow B_{k-1}, B_{k-2}, \ldots, B_{i+1}$
   Zu (1): $\Rightarrow B_i$
   Zu (2): $\Rightarrow$ Behandle dieses B_i rekursiv, bis das zu löschende Element auf dem Rückgrat liegt. In diesem Fall: lösche die Wurzel von $B_i (= x)$, Liste der Kinder $(B_{i-1}, B_{i-2}, \ldots, B_0)$. Merge all diese Binomialbäume (höchstens einer pro Index) mit dem Rest des Binomialwaldes.

1.3.2 Fibonacci-Heaps

$\equiv$ Priority Queue, (Fredman/Tarjan 1984)

Effiziente Datenstruktur für viele Probleme, z.B. Dijkstra-Algorithmus, minimale Spannbäume.

Operationen:	worst case	amortisiert
Insert	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Merge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Find_Min	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Decrease_Key	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Delete_Min	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$

Potentialmethode (Potential $\equiv$ Bankguthaben): Folgen von Operationen. Jede Operation hat

1. tatsächliche Kosten (Zeitbedarf)

2. amortisierte Kosten := tatsächliche Kosten $+\Delta$(Bankguthaben)

$\Rightarrow$ Ansparen für „schlechte“ Zeiten.

Datenstruktur:

- Wald von Bäumen, mit Heap-Bedingung, rang(v) eines Knoten v ist Anzahl seiner (direkten) Kinder.
- Wurzeln der Bäume doppelt zyklisch verkettet
- Zeiger auf das kleinste Element (Min_Pointer)
- Kinder eines jeden Knoten doppelt zyklisch verkettet
- jeder Knoten (außer den Wurzeln) hat ein Markierungsbit, das angibt, ob eines seiner Kinder bereits gelöscht wurde.

1.3.2.1 Implementierung

Prinzip ist Binomial Queue mit „Lazy Delete“

1. Insert: Neues Element in Liste der Wurzeln einfügen, dann Update Min_Pointer.
2. Merge: Konkateniere die beiden Wurzellisten, Update Min_Pointer.
3. Find_Min: ...
4. Delete:

   Falls x minimales Element dann Delete_Min, ansonsten

   $v := x$
   while true do
   if $v = x$ then
   lösche Knoten v
   hänge Liste der Kinder von v in die Wurzelliste
   des Fibonacci-Heaps ein
   else
   hänge den Unterbaum mit Wurzel v in die Wurzelliste ein;
   mache v unmarkiert
   fi
   if p(v) existiert then
   $v :=$ p(v)
   else
   return
   fi
   if v nicht markiert and v nicht Wurzel then
   markiere v
   return
   fi
   od

   reelle Kosten für Delete (falls nicht Delete_Min): $\mathcal{O}$(1+ Anzahl der kaskadierenden Schritte)

5. Decrease_Key:

   if an Wurzel then
   verringere Schlüssel an dieser Wurzel;
   update Min_Pointer (falls nötig)
   else
   führe Delete und Insert (mit dem neuen Schlüssel) aus;
   fi

   reelle Kosten: wie Delete

6. Delete_Min:

   lösche minimale Wurzel.
   füge Liste ihrer Kinder in die Wurzelliste ein.
   solange möglich, verbinde je zwei Bäume in der Wurzelliste, deren
   Wurzeln gleichen Rang haben, zu einem Baum (beachte Heap-Bedingung)
   update Min_Pointer.

   reelle Kosten: $\mathcal{O}$(max. Rang + Anzahl der Verbindungs-Operationen)

1.3.2.2 Amortisierte Kostenanalyse für Fibonacci-Heaps

Amortisierte Kosten für Operation := reelle Kosten für Operation + Δ Potential;
Potential := Anzahl der Bäume in der Wurzelliste + $2 \cdot$ Anzahl der markierten Knoten.
Damit ergeben sich folgende amortisierte Kosten:

1. Insert: $\mathcal{O}(1)$
2. Merge: $\mathcal{O}(1)$
3. Find_Min: $\mathcal{O}(1)$
4. Delete: $\mathcal{O}$(max. Rang)
5. Decrease_Key: $\mathcal{O}(1)$, da ein neuer Baum in der Wurzelliste + kaskadierende Schnitte.
6. Delete_Min: $\mathcal{O}$(max. Rang)

Lemma: Sei x ein Knoten im Fibonacci-Heap mit Rang$(x) = k$. Seien $y_1, y_2, \ldots, y_k$ die Kinder von x, in der Reihenfolge, in der sie hinzugefügt wurden. Für $i \geq 2$ gilt: Rang$(y_i) \geq i - 2$.

Beweis: Als y_i hinzugefügt wurde, war Rang$(x) \geq i-1$, also auch y_i. Danach kann es höchstens ein Kind verloren haben. (y_i ist nicht Wurzel). □

Satz: Sei x Wurzel eines Baums im Fibonacci-Heap, Rang$(x) = k$. Dann enthält der Baum mit Wurzel x mind. F_{k+2} Elemente.
($F_i = i$-te Fibonacci-Zahl mit $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$)

Beweis: Sei f_k die minimale Anzahl der Elemente im Baum, dessen Wurzel Rang k hat.
$\Rightarrow f_k \geq f_{k-2} + f_{k-3} + \ldots + f_0 + 1 + 1$
Setze: $g_k := g_{k-2} + g_{k-3} + \ldots + g_0 + 2$, d. h. $f_k \geq g_k$; $g_0 := 1$

(1)	g_k	$=$	g_{k-2}	$+$	g_{k-3}	$+$	$\cdots$			
(2)	g_{k-1}	$=$			g_{k-3}	$+$	g_{k-4}	$+$	$\cdots$	
(1) - (2)	$g_k - g_{k-1}$	$=$	g_{k-2}							

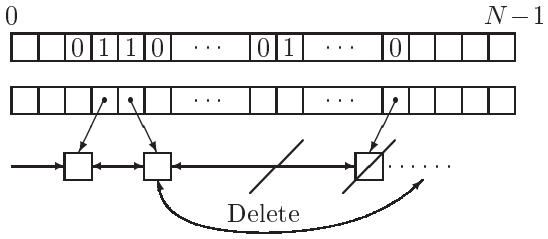
$\Rightarrow g_k = g_{k-1} + g_{k-2} \Rightarrow f_k \geq g_k = F_{k+2}$

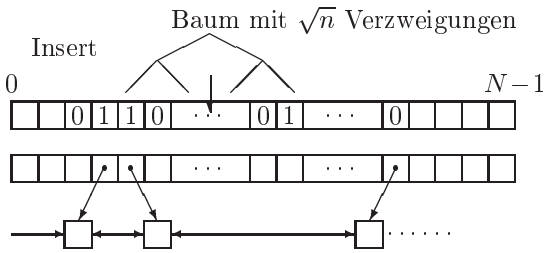
□

Variante: Lazy Delete

1.3.2.3 Eine $\mathcal{O}(\log\log N)$ Priority-Queue

(Peter van Emde Boas)
Universum U, $|U| = N$, $U \subseteq \mathbb{N}$
Hier: $U = \{0, 1, \ldots, N-1\}$
Bessere Priority-Queue für den Fall $n \geq \log N$



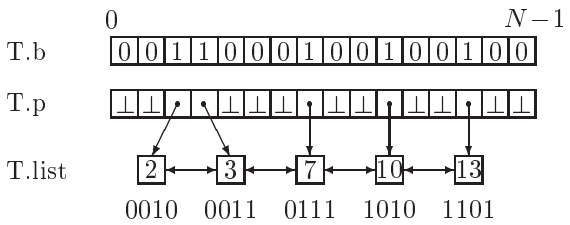


Sei $k \in \mathbb{N}$, $k \geq 2$; $k' = \lceil \frac{k}{2} \rceil$, $k'' = \lfloor \frac{k}{2} \rfloor$
$x \in [0, \ldots, 2^k - 1] \leq k$ Bits
$x' = \left\lfloor \frac{x}{2^{k''}} \right\rfloor$, $x'' := x \mod 2^{k''}$ (x' vordere Hälfte, x'' hintere Hälfte der Bits)
Sei $S = \{x_1, \ldots, x_m\} \subseteq [0, \ldots, 2^k - 1]$

Definition: Eine k-Struktur T für S besteht aus:

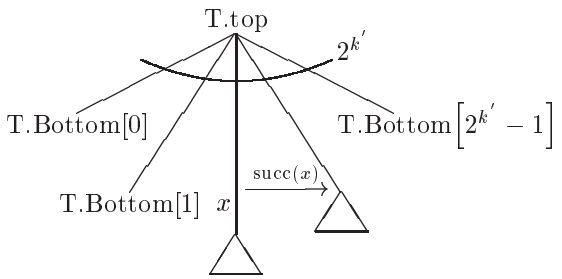
1. der Zahl T.size $= |\{x_1, \ldots, x_m\}| = |S| = m$
2. einer doppelt verketteten Liste T.list, die die Elemente von S in aufsteigender Reihenfolge enthält
3. einem Bitvektor T.b$[0 \ldots 2^k - 1]$ mit T.b$[i] = \begin{smallmatrix} 0 \\ 1 \end{smallmatrix}$ falls $\begin{smallmatrix} i \notin S \\ i \in S \end{smallmatrix}$ einem Zeiger (Pointer) Vektor T.p$[0 \ldots 2^k - 1]$. 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$[0 \ldots 2^{k'} - 1]$ 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 $\{x'_1, x'_2, \ldots, x'_m\}$ gegebene Menge, und für jedes $x \in [0 \ldots 2^{k'} - 1]$ ist T.bottom$[x]$ eine k''-Struktur für die Menge $\{x''_i;\ 1 \leq i \leq m;\ x = x'_i\}$

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



T.top ist 2-Struktur $\{0, 1, 2, 3\}$
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 $\min\{y \in \text{S};\ y > x\}$ in der k-Struktur T.

if $k = 1$ or T.Size ≤ 2 then
    naive Suche
else
    if T.size $= 0$ or $x \geq$ max in T.list then
        return(succ(x) gibt's nicht)
    else
        $x' := \left\lfloor \frac{x}{2^{k''}} \right\rfloor$
        $x'' := x \mod 2^{k''}$
        if T.top.b$[x'] = 1$ and $x'' < \max\{\text{T.bottom}[x']\}$ then
            return$(x' \cdot 2^{k''} + \text{succ}(x'', \text{T.bottom}[x']))$
        else
            $z' := \text{succ}(x', \text{T.top})$, return$(z' \cdot 2^{k''} + \min\{\text{T.bottom}[z']\})$
        fi
    fi
fi



Kosten: $\text{T}(k) \leq c + \text{T}(\lceil \frac{k}{2} \rceil) = \mathcal{O}(\log k)$

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

Beweis: $\surd$ □

Insert Operation:

- falls x bereits in Priority-Queue, dann fertig

- bestimme succ(x, T), füge x davor ein
- bestimme x' und x''
- behandle die entsprechende Unterstrukturen rekursiv:
  Insert$(x', \mathrm{T.top})$, Insert$(x'', \mathrm{T.bottom}[x'])$ (nur ein nicht trivialer rekursiver Aufruf)

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

Delete Operation:

Komplexität von Delete in k-Struktur: $\mathcal{O}(\log k)$
Kosten der Initialisierung: $\sim$ Größe der Datenstruktur
Platzbedarf für k-Struktur: Sei $\mathrm{S}(k)$ der Platzbedarf für eine k-Struktur.
Dann gilt:
$\mathrm{S}(1) = c$
$\mathrm{S}(k) = c2^k + \mathrm{S}(k') + 2^{k'}\mathrm{S}(k')$ für $k \geq 2$
Wir ersetzen zur Vereinfachung: $\mathrm{S}(k) = c2^k + \mathrm{S}(\frac{k}{2}) + 2^{\frac{k}{2}}\mathrm{S}(\frac{k}{2})$

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

Beweis:
Setze: $\mathrm{S}(k) := c' \cdot 2^k \log k$ (besser: Zeige $\mathrm{S}(k) := c'2^k \log k$ funktioniert)
Für $k = 1$ klar
Rekursionsschritt:

$$\begin{aligned}
\text{Platz für } k\text{-Struktur} &\leq c2^k + c'2^{\frac{k}{2}}(\log k - 1) + 2^{\frac{k}{2}}c'2^{\frac{k}{2}}(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(1 + 2^{\frac{k}{2}})(\log k - 1) \\
&= c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}}\log k + \log k - 2^{\frac{k}{2}} - 1) \\
&\leq c2^k + c'2^{\frac{k}{2}}(2^{\frac{k}{2}}\log k + \log k - 2^{\frac{k}{2}}) \\
&\leq c'2^k \log k \text{ falls} \\
&\quad c2^k + c'2^k \log k + c'2^{\frac{k}{2}}\log k - c'2^k \leq c'2^k \log k \\
&\quad \Leftrightarrow c'2^{\frac{k}{2}}\log k \leq (c' - c)2^k \\
&\quad \Leftrightarrow \frac{c' - c}{c'} \geq \frac{\log k}{2^k} \text{ (Gilt für } c' \text{ groß genug.)}
\end{aligned}$$

□

Satz: Sei $N = 2^k$, Universum $U = \{0, \ldots, N-1\}$. Wird eine Teilmenge $S \subseteq U$ durch eine k-Struktur dargestellt, dann benötigen die Operationen Insert, Delete und Find_Min 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:

1. Mehlhorn Bd. 1, S.290-296
2. van Emde Boas, P., R. Kaas, E. Zijlstra: Design and Implementation of an efficient priority queue, Math. Systems Theory 10 (1977), S. 99-127

1.3.3 Sich selbst organisierende Datenstrukturen

1.3.3.1 Sich selbst organisierende lineare Listen (als Wörterbücher)

Heuristiken, um häufig besuchte Elemente nahe des Listenanfangs zu halten.

Beispiele:

1. Move_to_front
2. Transpose: vertausche das besuchte Element mit seinem Vorgänger in der Liste.

Wir wollen zeigen, daß Move_to_front (MF) bis auf einen konstanten Faktor (im amortisierten Sinn) optimal ist. Für die Algorithmen nehmen wir an:

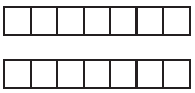
1. Kosten für access(x_i) und Delete(x_i) sind gleich der Position von x_i in der Liste.
2. Kosten für Insert(x) (x nicht in Liste) sind die Länge der Liste nach der Insert-Operation.
3. Ein Algorithmus kann die Liste jederzeit umorganisieren.
   (a) kostenlose Vertauschung: nach einer Access(x)- oder Insert(x)-Operation kann x beliebig näher an den Anfang der Liste verschoben werden.
   (b) bezahlte Vertauschung: jede andere Vertauschung benachbarter Listenelemente kostet eine Einheit.

Bemerkung: MF (und auch Transpose) benutzen nur kostenlose Vertauschungen. 3a ist pessimistisch für MF.

Satz: Sei s eine beliebige Folge von Access, Insert und Delete Operationen, beginnend mit einer leeren Liste. Sei A ein beliebiger (optimaler) Wörterbuch-Algorithmus mit obigen Einschränkungen. Sei $C_A(s)$ der Zeitaufwand von A auf s, ohne bezahlte Vertauschungen. Sei $X_A(s)$ die Anzahl der bezahlten Vertauschungen. Sei $C_{MF}(s)$ der Gesamtaufwand der Move_to_front-Heuristik auf s. Sei $F_A(s)$ die Anzahl der kostenlosen Vertauschungen, die A auf s ausführt. Sei $m := |s|$ die Länge von s. Dann

$$C_{MF}(s) \leq 2 \cdot C_A(s) + X_A(s) - F_A(s) - m$$

Beweis: Induktion über $|s|$. Potentialmethode für amortisierte Kosten von MF. □



Potential := $|\{\{x, y\};\ x \neq y;\ x, y$ kommen in beiden Listen vor, jedoch in verschiedener Reihenfolge$\}|$

Betrachte Insert(x) als access auf die (letzte + 1) Position in der Liste. Eine Access(x) oder Delete(x) Operation kostet also MF reell Zeit t, wenn t die Position von x in MF's Liste ist. i sei die Position von x in A's Liste.

Behauptung:

1. Die amortisierten Kosten von Access- (bzw. Insert-) Operationen sind $\leq 2i-1$ (i Index des Elements in der Liste).

2. Die amortisierten Kosten für eine kostenlose Vertauschung durch Algorithmus A sind ≤ -1.

3. Die amortisierten Kosten für eine bezahlte Vertauschung durch Algorithmus A sind $\leq +1$.

Beweis:

1. Liste L_A des Algorithmus A: $x_1\ x_2 \ldots x_i \ldots x_h \ldots x_n$
   Liste L_{MF} des Algorithmus MF: $\ldots x_h \ldots\ldots x_i \ldots$ $(\mathrm{Pos}(x_i) = t)$

   Sei k die Anzahl der Elemente (wie x_h), die in L_{MF} vor und in L_A nach x_i vorkommen.
   $\Rightarrow t \leq i + k \Rightarrow k \geq t - i$
   $\Delta_{\text{balance}} \leq -k + (i-1)$, also: $\Delta_{\text{balance}} \leq i - t + i - 1 \leq 2i - 1 - t$
   amortisierte Kosten = tatsächliche Kosten + $\Delta_{\text{balance}} \leq t + 2i - 1 - t = 2i - 1$
   Bei Delete Operation: $\Delta_{\text{balance}} = -k \leq i - t \Rightarrow$ amortisierte Kosten $\leq i$

2. L_A : $x_1 x_2 \ldots x_{i-1} x_i x_{i+1} \ldots x_n$ $\Delta_{\text{balance}} \leq -1$
   L_{MF} : $x_i \ldots\ldots$

3. L_A : $x_1 x_2 \ldots x_{i-1} x_i x_{i+1} \ldots x_n$ $\Delta_{\text{balance}} \leq 1$
   L_{MF} : $\ldots\ldots x_i \ldots\ldots$

   tatsächliche Kosten C_{MF} = amortisierte Kosten $- \underbrace{\Delta_{\text{balance}}}_{\geq 0}$
   $\leq$ amortisierte Kosten
   $\leq 2C_A(s) + X_A(s) - F_A(s) - m$

□

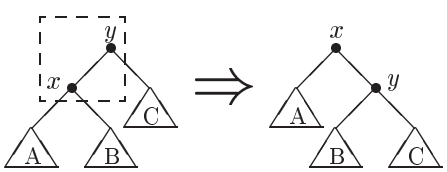
Anmerkung: Wenn wir nicht mit einer leeren Liste (oder zwei identischen Listen, d.h. $L_A = L_{MF}$) starten, ergibt sich:
$C_{MF} \leq 2C_A(s) + X_A(s) - F_A(s) - m + n^2$

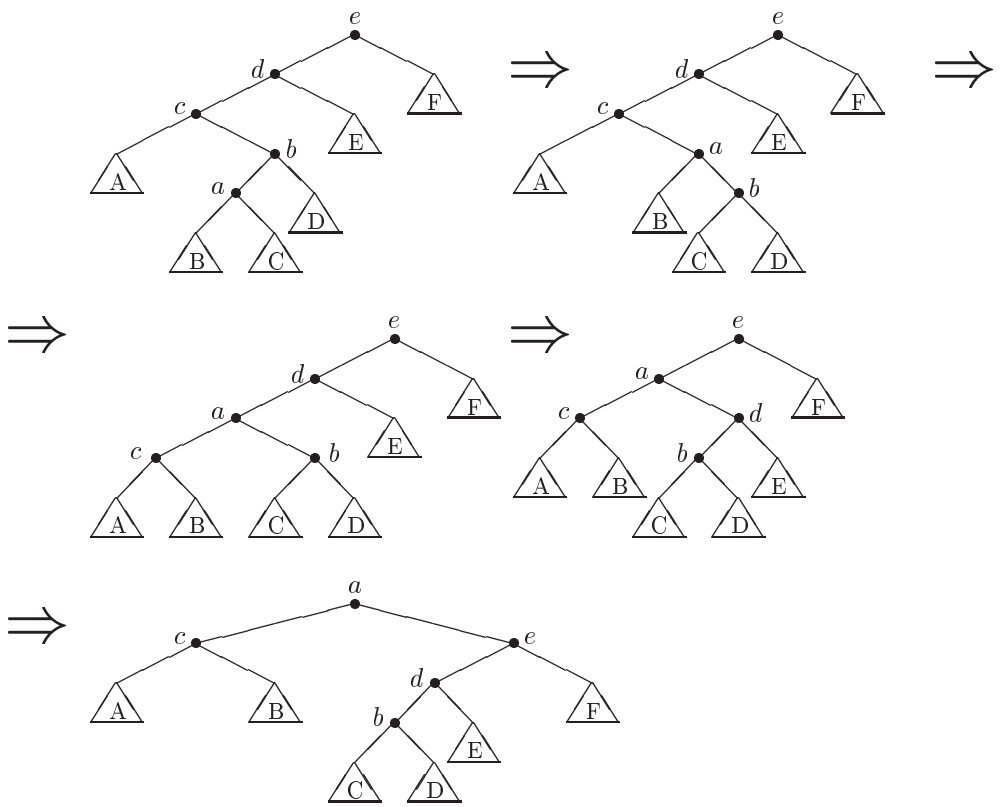
Literatur: Sleator, D.R., Tarjan: Amortized efficiency of list update and paging rules. Comm. ACM 28(2), p. 202-208 (1985)

1.3.3.2 Splay-Trees als Suchbäume

1.3.3.2.1 Einleitung

Rotationen, um Knoten an die Wurzel zu bewegen.
Beispiele:

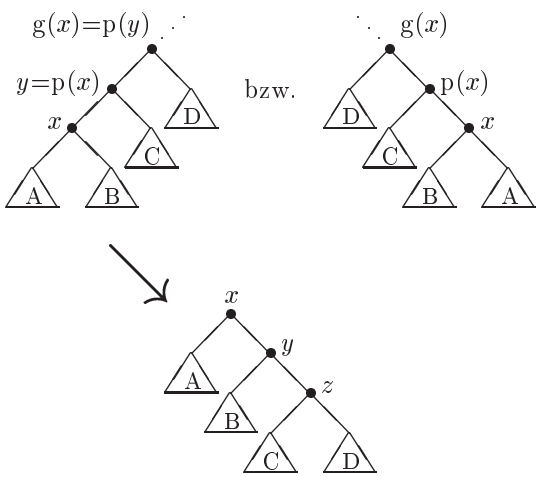




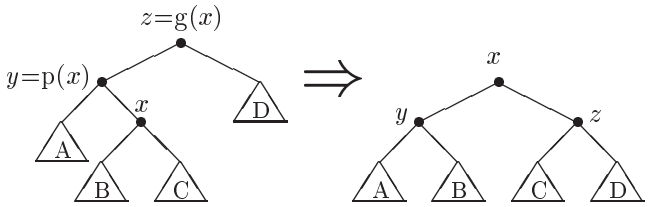
1.3.3.2.2 Die Splaying-Operation

Splay(x, T) bewegt mittels (einfacher + doppelter) Rotationen x zur Wurzel.
Fälle:

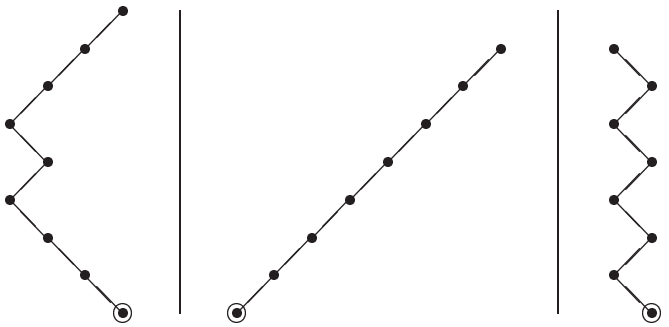
1. (zig): x ist Kind der Wurzel von T: einfache Rotation
2. (zig-zig): x hat Großvater g(x) (und Vater p(x)) und x und p(x) sind jeweils linke (bzw. rechte) Kinder ihres Vaters.



3. (zig-zag): x hat Großvater g(x) und Vater p(x), x ist linkes (rechtes) Kind von p(x), p(x) ist rechtes (linkes) Kind von g(x).



Übung: (Splaying-Operation mit dem eingekreisten Element durchführen)



1.3.3.2.3 Amortisierte Kostenanalyse der Splay-Operation

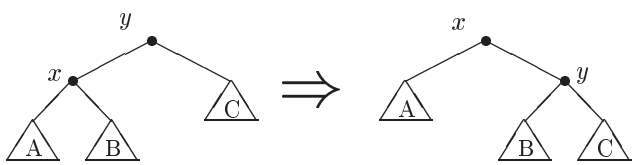
Jeder Knoten habe Gewicht $\mathrm{w}(x) > 0$. Gewicht $\mathrm{tw}(x)$ des Unterbaums mit Wurzel x ist $\sum_{\substack{y \text{ im} \\ \text{Unterbaum}}} \mathrm{w}(y)$.

Rang $\mathrm{r}(x) = \log(\mathrm{tw}(x))$
Potential eines Baumes T: $\sum_{x \in T} \mathrm{r}(x)$

Lemma: Sei T ein Splay-Tree mit Wurzel u, x ein Knoten in T. Die amortisierten Kosten für Splay(x, T) sind $\leq 1 + 3(\mathrm{r}(u) - \mathrm{r}(x)) = \mathcal{O}\left(\log \frac{\mathrm{tw}(u)}{\mathrm{tw}(x)}\right)$.

Beweis:
Berechne r und r′, tw und tw′, die Rang- bzw. Gewichtsfunktion vor und nach dem Rotationsschritt. Wir zeigen, daß die amortisierten Kosten im Fall 1 (zig) $\leq 1+3(\mathrm{r}'(x)-\mathrm{r}(x))$ und in den Fällen 2 und 3 (zig-zig bzw. zig-zag) $\leq 3(\mathrm{r}'(x)-\mathrm{r}(x))$ sind. y sei der Vater von x, z der Großvater (falls er existiert).

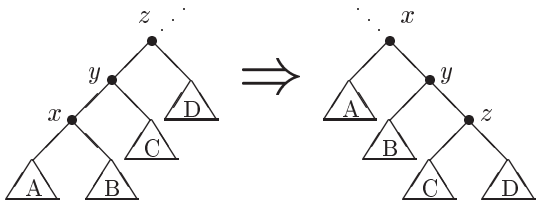
1. Fall:



amortisierte Kosten:

$$\leq 1 + \mathrm{r}'(x) + \mathrm{r}'(y) - \mathrm{r}(x) - \mathrm{r}(y) \leq 1 + \mathrm{r}'(x) - \mathrm{r}(x), \quad \text{da } \mathrm{r}'(y) \leq \mathrm{r}(y)$$
$$\leq 1 + 3(\mathrm{r}'(x) - \mathrm{r}(x)), \quad \text{da } \mathrm{r}'(x) \geq \mathrm{r}(x)$$

2. Fall:



amortisierte Kosten:

$$\begin{aligned}
&\leq 2 + \mathrm{r}'(x) + \mathrm{r}'(y) + \mathrm{r}'(z) - \mathrm{r}(x) - \mathrm{r}(y) - \mathrm{r}(z) \\
&= 2 + \mathrm{r}'(y) + \mathrm{r}'(z) - \mathrm{r}(x) - \mathrm{r}(y), \quad \text{da } \mathrm{r}'(x) = \mathrm{r}(z) \\
&\leq 2 + \mathrm{r}'(x) + \mathrm{r}'(z) - 2\mathrm{r}(x), \quad \text{da } \mathrm{r}'(x) \geq \mathrm{r}'(y) \text{ und } \mathrm{r}(y) \geq \mathrm{r}(x)
\end{aligned}$$

Es gilt, daß

$$2 + \mathrm{r}'(x) + \mathrm{r}'(z) - 2\mathrm{r}(x) \leq 3(\mathrm{r}'(x) - \mathrm{r}(x))$$

d.h.

$$2\mathrm{r}'(x) - \mathrm{r}(x) - \mathrm{r}'(z) \geq 2$$

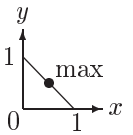
Betrachte dazu die Funktion

$$\mathrm{f}(x, y) = \log(x) + \log(y)$$

auf dem Bereich

$$x, y > 0, \ x + y \leq 1$$

d.h. $\mathrm{f}(x, y)$ nimmt ihr eindeutiges Maximum im Punkt $\left(\frac{1}{2}, \frac{1}{2}\right)$ an.



Beweis: Da die Funktion $\mathrm{z} \to \log(z)$ streng monoton wachsend ist, kann sich das Maximum der Funktion $\mathrm{f}(x, y) = \log(x) + \log(y)$ nur auf dem Geradensegment $x + y = 1$, $x, y > 0$ befinden. Dadurch erhalten wir ein neues Maximierungsproblem für die Funktion $\mathrm{g}(x) = \log(x) + \log(1 - x)$ auf diesem Geradensegment. Da $\mathrm{g}(x)$ an den Rändern gegen $-\infty$ strebt, muß es sich um ein lokales Maximum handeln. Die einzige Nullstelle der Ableitung

$$\mathrm{g}'(x) = \frac{1}{\ln a}\left(\frac{1}{x} - \frac{1}{1-x}\right),$$

wenn $\log = \log_a$, ist $x = 0.5$ (unabhängig von a). Weiter ist

$$\mathrm{g}''(x) = -\frac{1}{\ln a}\left(\frac{1}{x^2} + \frac{1}{(1-x)^2}\right).$$

Da $\mathrm{g}''(0.5) < 0$ ist nimmt $\mathrm{g}(x)$ in $x = 0.5$ sein globales Maximum an. Insgesamt folgt, daß die Funktion $\mathrm{f}(x, y) = \log(x) + \log(y)$ ihr globales Maximum im Bereich $x, y > 0$, $x + y \leq 1$ an der Stelle $(0.5, 0.5)$ annimmt. □

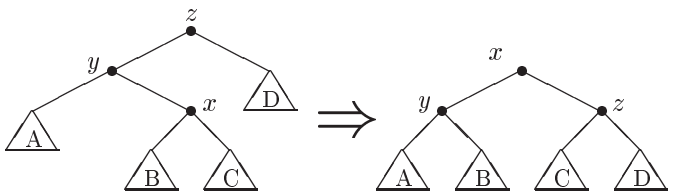
Damit gilt:

$$\mathrm{r}(x) + \mathrm{r}'(z) - 2\mathrm{r}'(x) = \log\left(\frac{\mathrm{tw}(x)}{\mathrm{tw}'(x)}\right) + \log\left(\frac{\mathrm{tw}'(z)}{\mathrm{tw}'(x)}\right) \leq -2$$

da

$$\mathrm{tw}(x) + \mathrm{tw}'(z) \leq \mathrm{tw}'(x)$$

3. Fall:



amortisierte Kosten:

$$\begin{aligned}
&\leq 2 + \mathrm{r}'(x) + \mathrm{r}'(y) + \mathrm{r}'(z) - \mathrm{r}(x) - \mathrm{r}(y) - \mathrm{r}(z) \\
&\leq 2 + \mathrm{r}'(y) + \mathrm{r}'(z) - 2\mathrm{r}(x), \quad \text{da } \mathrm{r}'(x) = \mathrm{r}(z) \text{ und } \mathrm{r}(x) \leq \mathrm{r}(y) \\
&\leq 2(\mathrm{r}'(x) - \mathrm{r}(x)), \quad \text{da } 2\mathrm{r}'(x) - \mathrm{r}'(y) - \mathrm{r}'(z) \geq 2
\end{aligned}$$

Letzteres folgt aus der Behauptung über $\mathrm{f}(x, y)$ wie im Fall 2.

Die Gesamtbehauptung des Lemmas folgt dann durch Aufaddieren der amortisierten Kosten für die einzelnen Schritte (Beachte: Teleskop-Summe). □

Betrachte Folge von m Zugriffen in einem Splay-Tree mit n Knoten. Die Verringerung des Potentials ist, falls sich die Gewichte nicht ändern, beschränkt durch

$$\sum_{i=1}^{n} \log \frac{W}{w_i}; \qquad W := \sum_{i=1}^{n} w_i.$$

Satz: die gesamten Kosten für die m Zugriffe sind $\mathcal{O}((m+n)\log n + m)$

Beweis:
amortisierte Kosten = reelle + Δ_{balance}; reelle Kosten = amortisierte - Δ_{balance}.
Wähle $w_i = \frac{1}{n}$ für alle Knoten. $\Rightarrow$ amortisierte Kosten für einen Zugriff $\leq 1 + 3\log n$, da $W = \sum\limits_{i=1}^{n} w_i = 1$. Die Verringerung im Potential ist $\leq \sum\limits_{i=1}^{n} \log \frac{W}{w_i} = \sum\limits_{i=1}^{n} \log n = n \log n$. $\Rightarrow$ reelle Kosten $\leq m(1 + 3\log n) + n \log n$. □

Satz: Sei $\mathrm{q}(i)$ die Anzahl der Zugriffe auf das Element i. Falls auf jedes Element zugegriffen wird (also $\mathrm{q}(i) > 0$ für alle i), dann sind die Gesamtkosten für die Zugriffe

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

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

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

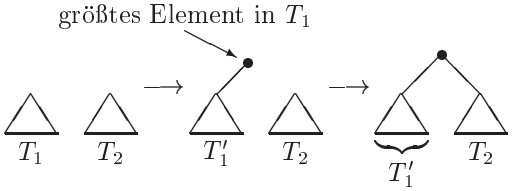
□

1.3.3.2.4 Wörterbuchoperationen in Splay-Trees

Betrachte (Alle diese Operationen werden mit Hilfe von Splay implementiert):

- Access$(i,\ T)$: $\surd$ (Haben wir bereits)
- Join$(T_1,\ T_2)$:

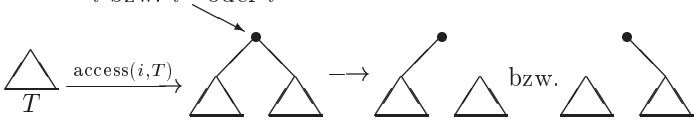
größtes Element in T_1



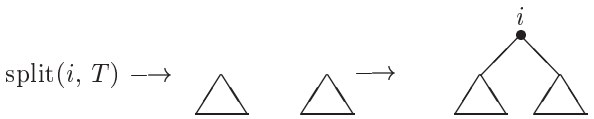
beachte $x \in T_1,\ y \in T_2$, dann gilt $x < y$

- Split$(i,\ T)$:

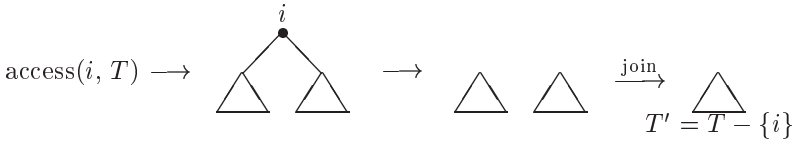
i bzw. i^- oder i^+



- Insert$(i,\ T)$:



- Delete$(i,\ T)$:



Definition: Sei $i \in U$; T ein Splay-Tree. Dann bezeichnen wir mit $i-$ (bzw. $i+$) den Vorgänger (bzw. den Nachfolger) von i in der gegebenen Ordnung in T (falls diese existieren). Falls $i-$ bzw. $i+$ undefiniert sind, so setzen wir $\mathrm{w}(i-) = \infty$ bzw. $\mathrm{w}(i+) = \infty$. Weiterhin sei W das Gesamtgewicht aller an einer Wörterbuch-Operation beteiligten Knoten.

Satz: Für die amortisierten Kosten der Wörterbuch-Operationen in Splay-Trees gelten die folgenden oberen Schranken: $(T,\ T_1,\ T_2 \neq 0)$

$$\text{access}(i,T): \begin{cases} 3\log\left(\frac{W}{\mathrm{w}(i)}\right)+1, \text{ falls } i \in T \\ 3\log\left(\frac{W}{\min\{\mathrm{w}(i-),\ \mathrm{w}(i+)\}}\right)+1, \text{ falls } i \notin T \end{cases}$$

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

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

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

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

falls i nicht minimal in T

1.3.4 Weitere Arten von Datenstrukturen

- Dynamische Datenstrukturen:
  Referenz: Bent, S. W.: Dynamic weighted data structures, TR STAN-CS-82-916, Department of Computer Science, Stanford University, 1982

- Persistente Datenstrukturen:
  Referenz: Driscoll, J. R., Sarnak, N., Sleater, D. D., Tarjan, R. E., Making data structures persistent, Proceedings of the 18^{th} Annual ACM Symposium on Theory of Computing (STOC), 1986. p.109–121.