Vorlesungsmitschrift

Algorithmen und Berechenbarkeit

Vorlesung 06

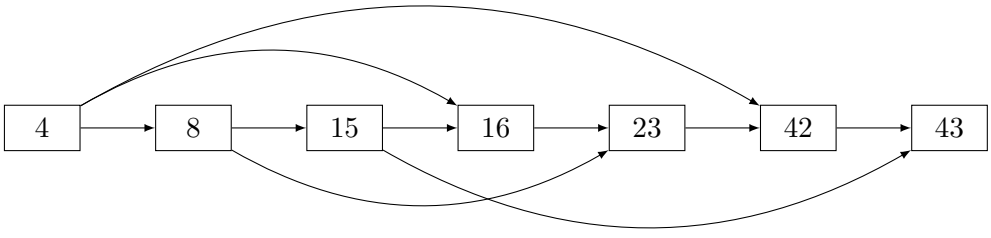
Letztes Update: 2017/12/01 - 16:15 Uhr

Randomisierte Skiplisten

Randomisierte Skipliste beschreibt eine Datenstruktur, die Ähnlichkeiten mit verketteten Listen aufweist. Die Elemente der Skipliste sind ebenfalls mit dem nächsten Element verkettet aber zusätzlich auch mit zufällig bestimmten anderen Elementen der Liste.
Die drei Hauptoperationen *Einfügen, Löschen* und *Finden* werden unterstützt, wobei sich Letzteres wie folgt verhält: Die Suche nach x liefert x, falls die Skipliste x enthält, ansonsten wird x' zurückgegeben, wobei x' das nächstkleinere enthaltene Element von x darstellt.
Alternative Datenstrukturen werden aus verschiedenen Gründen nicht verwendet:

- Binärer Suchbaum und sortierte verkettete Listen
  $\Rightarrow$ Suche in $\mathcal{O}(n)$ zu langsam
- Balancierter Suchbaum
  $\Rightarrow$ zu kompliziert
- Sortiertes Array
  $\Rightarrow$ *Einfügen* und *Löschen* mit $\mathcal{O}(n)$ zu langsam

Motivation



Durch zusätzliche Pointer wird ein schnelleres Bewegen durch die verkettete Liste ermöglicht.
$\Rightarrow$ Wie viele zusätzliche Pointer?
$\Rightarrow$ Wie weit die Sprünge?

Jedes Element hat einen *Turm* von Pointern, wobei die Höhe des Turmes zum Beispiel nach folgender Methode bestimmt werden kann:

$$
\begin{aligned}
P(\text{Höhe} = h) = \quad & 2^{-(h+1)} \\
\Rightarrow \quad P(h = 0) &= \frac{1}{2} \\
P(h = 1) &= \frac{1}{4} \\
P(h = 2) &= \frac{1}{8} \\
\ldots &
\end{aligned}
$$

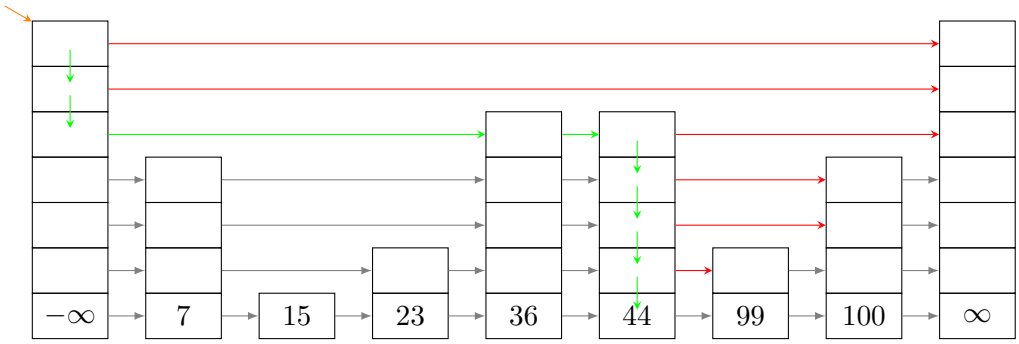
Jede Turmetage wird mit dem nächsten Element des nächsten Turmes verbunden, der mindestens genauso groß ist.

Search(x)
    $v \leftarrow -\infty$-Turm
    $h \leftarrow v$.height

    while $h > 0$ **do**
        while $x \leq v$.forward$[h] \rightarrow$ key **do**
            $v \leftarrow v$.forward$[h]$
        end while
        $h \leftarrow h - 1$
    end while

    return v

Beispiel: Finde 44



Laufzeit von Search(x)

$$\text{Beweis mit } X_{ik} \text{ möglich: } X_{ik} = \begin{cases} 1 & \text{Falls Turm } i \text{ auf Höhe } h \text{ besucht} \\ 0 & \text{sonst} \end{cases}$$

Besser wäre jedoch, eine Routine zu bauen, die dieselben Zellen besucht wie die Suche, jedoch einfacher zu analysieren ist.

$v \leftarrow x$
$h \leftarrow 0$

while $v \neq -\infty$ **and** $h \neq h_{max}$ **do**
    if v.height $> h$ **then**
        $h = h + 1$
    else
        $v = v$.backward$[h]$
    end if
end while

return v

Man muss sich also den **Search(x)**-Algorithmus rückwärts vorstellen. Damit lässt sich beobachten

- Die Wahrscheinlichkeit, dass sich über der aktuellen Zelle noch eine weitere Zelle befindet, ist $\frac{1}{2}$.
- Im `if-else` - Zweig entscheidet sich, ob hoch (WS $= \frac{1}{2}$) oder nach links (WS $= \frac{1}{2}$) weitergegangen wird.

$\rightarrow$ Erwartete Anzahl an Linksschritten = Erwartete Anzahl an Rechtsschritten

- Für die erwartete Maximalhöhe **eines** Turmes ergibt sich damit:

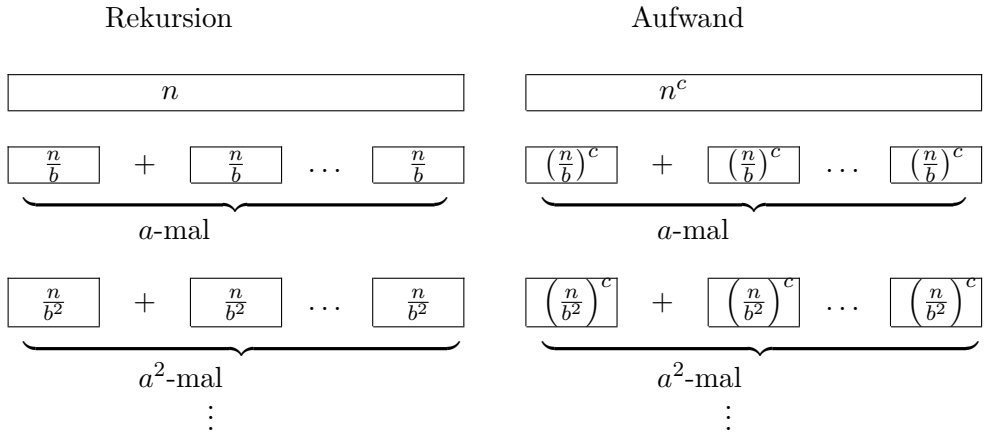
$$\begin{aligned} P(h_i \geq h) &= 2^{-h} \\ \hookrightarrow P(h_i \geq 2 \cdot \log(n) - 1) &\leq \frac{1}{n^2} \end{aligned}$$

- Und für die erwartete Maximalhöhe **aller** Türme somit:

$$\begin{aligned} & P(h_1 \geq 2 \cdot \log(n) \| h_2 \geq 2 \cdot \log(n) \| ... \| h_n \geq 2 \cdot \log(n)) \\ \leq & \frac{1}{n^2} + \frac{1}{n^2} + \ldots + \frac{1}{n^2} \\ = & n \cdot \frac{1}{n^2} = \frac{n}{n^2} = \frac{1}{n} \end{aligned}$$

$\Rightarrow$ Mit hoher Wahrscheinlichkeit ist **Search(x)** also $\mathcal{O}(\log(n))$.

Visualisierung der Laufzeit von Search(x) und Analyse mit Master-Theorem



$$\sum_{l=0}^{\log_b(n)} n^c \cdot \underbrace{\left(\frac{a}{b^c}\right)^l}_{q} = n^c \cdot \sum_{l=0}^{\log_b(n)} q^l \quad \text{(Geom. Reihe)}$$

$$\text{Also } \begin{cases} \leq & n^c \cdot \frac{1}{1-q} \in \Theta(n^c), \qquad q < 1 \Leftrightarrow \log_b(a) < c \\ = & n^c \cdot log_b(n) \in \Theta(n^c \cdot \log(n)), \qquad q = 1 \Leftrightarrow \log_b(a) = c \\ > & \in \Theta(n^{\log_b(a)} \cdot \log(n)), \qquad q > 1 \Leftrightarrow \log_b(a) > c \end{cases}$$

Quick-Sort mit $a = 2$, $b = 2$ und $c = 1$ entspricht also der zweiten Klasse: $\log_2(2) = 1 = c$

Anhang

Master-Theorem

Rekursionsgleichungen für *Divide* & *Conquer*-Algorithmen lassen sich mit dem Master-Theorem lösen:

$$T(n) = a \cdot T(\frac{n}{b}) + \Theta(n^c)$$

$a \tilde{=}$ Anzahl der Teilprobleme $\quad | \; a \geq 1$

$b \tilde{=}$ Verkleinerung des Teilproblems $\quad | \; b > 1$

$c \tilde{=}$ Aufwand für *Divide* & *Merge* $\quad | \; c \geq 1$