Algorithmen und Datenstrukturen

Blatt 4

Uni Hamburg, Wintersemester 2019/20

Präsentation am 11.–13. Dezember 2019

Jede Teilaufgabe zählt als ein einzelnes Kreuzchen.

Übung 1.
Gegeben ist eine Hashtabelle T der Länge $m = 11$ mit Hashfunktion $h(k; i) = (k + i^2) \bmod m$. Tragen Sie die Schlüssel 56, 46, 31, 45,42, 65, 29, 44, 23 in der angegebenen Reihenfolge in die Hashtabelle ein. Verwalten Sie Kollisionen:

(a) durch Verkettung ($i = 0$),

(b) durch offene Adressierung.

(c) (i) Nehmen Sie an, dass die Hash-Funktion $h(k)$ Schlüssel zufällig gleichverteilt auf einen Wert in $\{0, 1\}^b$ abbildet, wobei b die Anzahl der Bits ist. Berechnen Sie die Wahrscheinlichkeit, dass keiner von m Schlüsseln einen beliebigen, festen Hash-Wert besitzt.

(ii) Ein System hat m Benutzer, die sich durch Eingabe eines Passwort authentifizieren. Nehmen Sie an, dass die m Hash-Werte der von den Benutzern gewählten Passwörtern zufällig und unabhängig voneinander sind. Nutzen Sie das vorherige Ergebnis, um zu ermitteln, wieviele Bits notwendig sind, damit ein Angreifer durch Eingabe eines zufälligen Passworts mit höchstens Wahrscheinlichkeit p Erfolg hat. *Hinweis: Nutzen Sie, dass* $(1+x)^n \approx e^{nx}$ *für* $n > 1, |x| \leq 1$.

Lösung 1.

(a) Berechnung der Hash-Werte für alle Schlüssel mit Hashfunktion $h(k; i = 0)$:

key k	56	46	31	45	42	65	29	44	23
$h(k, 0)$	1	2	9	1	9	10	7	0	1

Beim Einfügen eines Elements x in die einfach verketteten Liste $T[h(x.key)]$, wird x am Kopf der Liste eingefügt, sodass sich folgende Hashtabelle ergibt:

hash $h(k, 0)$	0	1	2	3	4	5	6	7	8	9	10
$T[h(k)]$	[44]	[23,45,56]	[46]	[]	[]	[]	[]	[29]	[]	[42,31]	[65]

(b) Wir berechnen die Hash-Werte für alle Schlüssel von links nach rechts und erhöhen i um 1, wenn eine Kollision festgestellt wird. Jeder Schlüssel wird sofort beim ersten Hash-Wert ohne Kollision an der entsprechenden Stelle eingefügt.

key k	56	46	31	45	42	65	29	44	23
$h(k, 0)$	1	2	9	1	9	10	7	0	1
$h(k, 1)$				2	10	0		1	2
$h(k, 2)$				5				4	5
$h(k, 3)$									10
$h(k, 4)$									6

Somit ergibt sich folgende Hashtabelle:

$h(k,i)$	0	1	2	3	4	5	6	7	8	9	10
$T[h(k,i)]$	65	56	46		44	45	23	29		31	42

(c) (i) Da h gleichverteilt ist, ist die Wahrscheinlichkeit, dass die Funktion einen beliebigen Wert x annimmt, der Kehrwert der Anzahl der möglichen Werte. Es gibt 2^b viele darstellbare Werte mit b Bits, daher:

$$\Pr[h(k) = x] = 2^{-b} \tag{1}$$

Da jeder der m Schlüssel verschieden von x sein soll, ist die gesuchte Wahrscheinlichkeit das m-fache Produkt der Gegenwahrscheinlichkeit.

$$\Pr[\text{alle } m \text{ Schlüssel sind von } x \text{ verschieden}] = (1 - 2^{-b})^m \approx e^{-m/2^b} \tag{2}$$

Wir nutzen hier bereits den Hinweis aus der folgenden Aufgabe.

(ii) Wir nehmen an, der Angreifer probiert ein Passwort, dessen Hashwert x ist. Die Wahrscheinlichkeit, dass die m Benutzer-Passwörter einen davon verschiedenen Wert besitzen, haben wir in der vorigen Aufgabe berechnet.

$$e^{-\frac{m}{2^b}} \geq 1 - p \tag{3}$$

$$\frac{m}{2^b} \leq \ln\left(\frac{1}{1-p}\right) \tag{4}$$

$$2^b \geq \frac{m}{\ln\left(\frac{1}{1-p}\right)} \tag{5}$$

$$b \geq \log_2(m) + \log_2\left(\frac{1}{\ln\left(\frac{1}{1-p}\right)}\right) \tag{6}$$

Die benötigte Länge wächst also abhängig von p und logarithmisch in m. *Anmerkung: Die übliche Länge sind 32 Bits. Bei mehreren Milliarden Nutzern stehen die Chancen demnach gut, irgendein Passwort zu erraten. Andererseits ist die Annahme zufällig gewählter Passwörter natürlich unhaltbar.*

Übung 2.

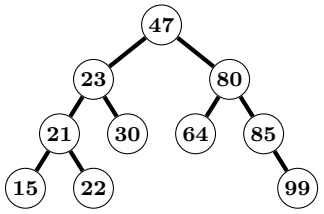
(a) Betrachten Sie den nebenstehenden binären Suchbaum. Geben Sie an, in welcher Reihenfolge bei Traversieren mit

(i) Preorder

(ii) Inorder

(iii) Postorder

die Knoten evaluiert werden.



(b) Nehmen Sie an, die Suche nach einem Schlüssel k in einem binären Suchbaum endet in einem Blatt. Dann gibt es drei Mengen: A, die Schlüssel links des Suchpfads; B, die Schlüssel auf dem Suchpfad; sowie C, die Schlüssel rechts des Suchpfads. Beweisen oder widerlegen Sie die folgende Behauptung: Für beliebige drei Schlüssel $a \in A$, $b \in B$ und $c \in C$ gilt $a \leq b \leq c$.

Lösung 2.

(a) (i) $47, 23, 21, 15, 22, 30, 80, 64, 85, 99$
Prinzip: Wurzel, linker Teilbaum, rechter Teilbaum

(ii) $15, 21, 22, 23, 30, 47, 64, 80, 85, 99$
Prinzip: Linker Teilbaum, Wurzel, rechter Teilbaum

(iii) $15, 22, 21, 30, 23, 64, 99, 85, 80, 47$
Prinzip: Linker Teilbaum, rechter Teilbaum, Wurzel

(b) Wir betrachten den Suchbaum aus Aufgabe 2.a). Sei $k = 22$, $a = 15$, $b = 47$ und $c = 30$. Da $b > c$, ist dies ein Gegenbeispiel.
Allgemein gilt durch die Suchbaumeigenschaft, dass im linken Teilbaum eines Knoten nur kleinere Werte zu finden sind. Damit ist die Wurzel eines Teilbaums insbesondere größer als jeder Knoten rechts eines Pfades in diesem Teilbaum. Umgekehrtes gilt für die rechte Seite.

Übung 3.

Erweitern Sie einen Baum um eine Funktion `getDepth`, die für einen beliebigen Knoten die Höhe des Teilsbaums (mit diesem Knoten als Wurzel) zurückliefert.

(a) Beschreiben Sie zwei Möglichkeiten zur Umsetzung der `getDepth` Methode:

(i) ohne zusätzlichen Speicher im Baum,

(ii) mit zusätzlichem Speicher im Baum.

Begründen Sie kurz deren Korrektheit.

(b) Welche Laufzeit haben die beiden `getDepth` Methoden und welcher zusätzliche Speicherbedarf wird jeweils benötigt?

(c) Welche Auswirkungen haben beide Alternativen auf die Laufzeiten von Einfügen und Löschen?

Lösung 3.

(a) (i) Bei einem Aufruf von `getDepth` auf einem beliebigen Knoten x muss die Höhe des Teilbaums neu berechnet werden. Ein rekursiver Algorithmus gibt immer das Maximum der Höhen seiner Kinder um 1 erhöht zurück. Wenn der rekursive Aufruf bei einem Kindknoten angelangt, wird konstant 0 zurückgeliefert.

$$getDepth(x) = \begin{cases} 0 & \text{falls } x \text{ Blatt} \\ \max(getDepth(x.l), getDepth(x.r)) + 1 & \text{sonst} \end{cases}$$

Die Methode ist korrekt, da alle Knoten des Teilbaums rekursiv durchlaufen werden und in jeder Ebene des Teilbaums die Höhe um 1 inkrementiert wird. Durch das Wählen des Maximums der Höhe von linkem bzw. rechtem Teilbaum wird sichergestellt, dass auch in unbalancierten Teilbäumen die korrekte Höhe (längster Pfad von Wurzel zu Blatt) gefunden wird.

(ii) Jeder Knoten speichert zusätzlich die Tiefe des Teilbaums mit ihm selbst als Wurzel. Die `getDepth` Methode muss beim Aufruf ausschließlich den gespeichert Wert zurückliefern. Dafür müssen beim Einfügen und Löschen die

gespeicherten Tiefen in betroffenen Knoten und ihren Eltern ggf. angepasst werden.

Beim Einfügen von einem Blatt hat dieser neue Knoten immer eine Tiefe 0. Wir nehmen an, der Knoten zum Einfügen des Blattes wurde rekursiv in einer Methode `insert` gefunden. Wenn der Call-Stack von `insert` wieder abgearbeitet wird, muss in jedem Knoten x auf dem Pfad von dem neuen Blatt bis einschließlich zur Wurzel des Baums geprüft werden, ob die aktuelle gespeicherte Tiefe korrekt ist, d.h. $\max(getDepth(x.l), getDepth(x.r)) + 1$. Wenn der Wert unverändert ist, kann die Prüfung abgebrochen werden, andernfalls muss der Wert geändert werden. Beim Löschen von Knoten kann analog zum Einfügen verfahren werden.

Durch diese Anpassungen wird sichergestellt, dass die eingespeicherte Tiefe des Teilbaums jedes Knotens in dem Baum nach Einfüge/Lösch-Aktionen korrekt ist. Die Argumentation ist analog zu (i) nur werden in diesem Fall die Zwischenergebnisse gespeichert und falls nötig nach jeder verändernden Operation aktualisiert und propagiert.

(b) Sei n die Anzahl der Knoten in dem Teilbaum des gewählten Knoten x. Dann

(i) ist die Laufzeit $O(n)$, da alle Knoten des Teilbaums durchlaufen werden. Es wird kein zusätzlicher Speicherplatz benötigt.

(ii) ist die Laufzeit $O(1)$, da das Lesen eines gespeicherten Werts konstante Zeit benötigt. Es wird in jedem Knoten des Baums ein zusätzliches Feld benötigt, sodass sich ein zusätzlicher Speicherbedarf von $O(n)$ ergibt.

(c) (i) Die Laufzeit von Einfügen und Löschen bleibt unberührt, da die jeweiligen Methoden nicht angepasst wurden.

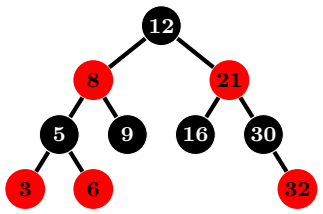
(ii) Die asymptotische Laufzeit von Einfügen und Löschen bleibt unverändert, da je Knoten entlang des Pfads nur konstanter zusätzlicher Aufwand entsteht (Zugriff auf gespeicherte Tiefe der Kinder, Maximum bestimmen, Addition).

Übung 4.
Betrachten Sie den folgenden Rot-Schwarz-Baum.

(a) Fügen Sie die Werte 13 und 42 ein. Löschen Sie anschließen den Wert 8. Geben Sie den Inhalt des Baums nach jeder Einfüge/Lösch-Aktion an.

(b) Zeichnen Sie einen validen Rot-Schwarz-Baum mit Schwarz-Höhe 4, der so unbalanciert wie möglich ist.

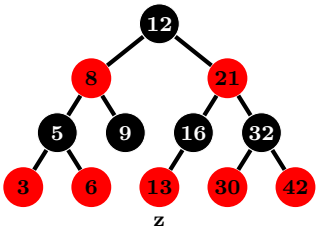
(c) Entwickeln Sie basierend auf der vorigen Aufgabe eine Formel für die minimale Anzahl der Knoten in einem maximal unbalancierten Rot-Schwarz-Baum der Höhe h.



Lösung 4.

(a) Wir verwenden den RB-Insert Algorithmus, der einen Wert einfügt und dann RB-Insert-Fixup aufruft, um sicherzustellen, dass die Rot-Schwarz-Baum Eigenschaften eingehalten werden.

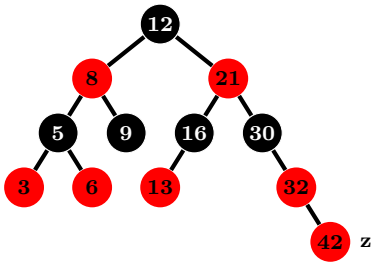
Einfügen von 13



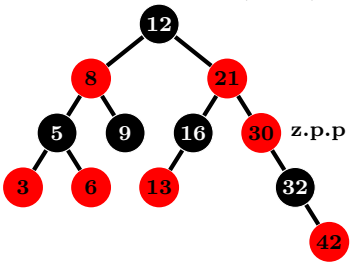
RB-Insert-Fixup(T,z) terminiert sofort, da $z.p.color \neq$ RED.

Einfügen von 42 führt zu Fall 3 in RB-Insert-Fixup.

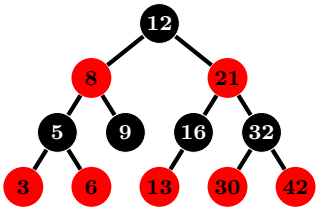
Vor RB-Insert-Fixup ⇒



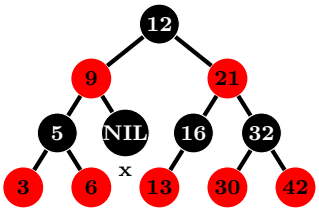
Vor Left-Rotate(z.p.p) ⇒



Final



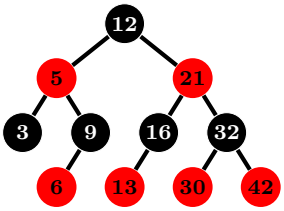
Löschen von 8 : RB-Delete führt zu dem Zweig wo y =Tree-Minimum(z.right) = 9. Knoten 8 wird durch Knoten 9 ersetzt.



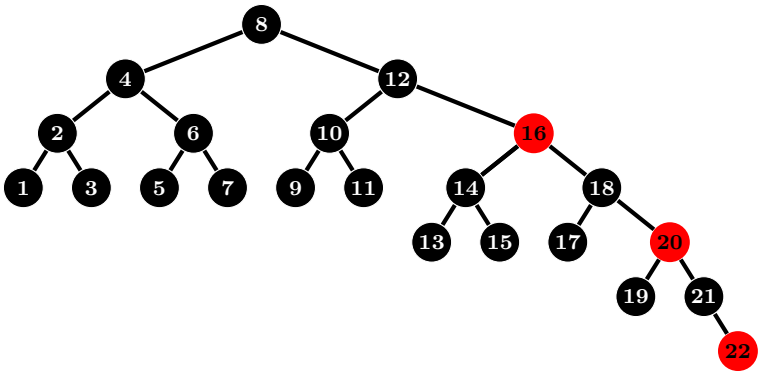
Anschließend wird RB-Delete-Fixup(T,x) aufgerufen. Jetzt ist x = x.p.right, sodass wir in Fall 4 sind: x's Geschwisterknoten w =x.p.left=5 ist schwarz und w's linker Kindknoten 3 ist rot. Es müssen folgende Operationen ausgeführt werden:

```
1:  w.color = x.p.color
2:  x.p.color = BLACK
3:  w.left.color = BLACK
4:  RIGHT-ROTATE(T, x.p )
5:  x = T.root
```

Danach terminiert RB-Delete-Fixup und der finale Baum steht:



(b) Valider, möglichst unbalancierter Rot-Schwarz-Baum mit Schwarz-Höhe 4:



Der angegebene Baum ist so unbalanciert wie möglich für Höhe 6, d.h. die Anzahl der Knoten ist minimal. Dafür existiert genau ein Pfad von der Wurzel bis zu einem Blatt der Länge 6, alle anderen Pfade sind so kurz wie möglich. Diese anderen Pfade sind minimal, da sie keine roten Knoten beinhalten. Da keine zwei roten Knoten hintereinander vorkommen dürfen, wechseln sich rote und schwarze Knoten auf dem langen Pfad ab. Um die Anzahl der Knoten der schwarzen Teilbäume zu minimieren, ist es von Vorteil, dass sich die schwarzen Knoten möglichst dicht an der Wurzel befinden. Daher ist Knoten 12 schwarz.

(c) Folgendes lässt sich bei einem maximal unbalancierten Rot-Schwarz-Baum T mit Schwarz-Höhe b feststellen:

(i) Der Baum besteht aus einem Pfad von der Wurzel bis zu einem roten Blatt mit $2b$ Knoten. Dabei wechseln sich bis auf eine Ausnahme rote und schwarze Knoten ab. Falls h gerade ist, befinden sich ganz oben zwei schwarze Knoten hintereinander (die Wurzel und ein Kind).

(ii) An jedem Knoten dieses Pfades hängt außerdem ein vollständiger Baum aus schwarzen Knoten.

(iii) Es kommen alle Bäume jeder Höhe von 1 bis $b-2$ doppelt vor.

(iv) Falls die Höhe von T gerade ist, gibt es einen Baum der Höhe $b-1$, sonst zwei.

Sei $N(b)$ die Anzahl der Knoten des maximal unbalancierten Rot-Schwarz-Baums in Abhängigkeit der Schwarzhöhe b.

$$N(b) = 2b - 1 + 2^{b-1} - 1 + 2\sum_{x=1}^{b-2}(2^x - 1) = 2^b - 2^{b-1} - 2 \quad \text{falls } h \text{ gerade} \tag{7}$$

$$N(b) = 2b + 2\sum_{x=1}^{b-1}(2^x - 1) = 2^{b+1} - 2 \quad \text{sonst} \tag{8}$$

Schließlich stellen wir fest, dass $b = \frac{h}{2} + 1$ falls h gerade und $b = \frac{h+1}{2}$ sonst. Damit drücken wir N in Abhängigkeit von h aus.

$$N(h) = \begin{cases} 2^{\frac{h+1}{2}+1} - 2 & \text{falls } h \text{ ungerade} \\ 2^{\frac{h}{2}+1} + 2^{\frac{h}{2}} - 2 & \text{sonst} \end{cases} \tag{9}$$