Programmierung II - Übungsblatt 9
Übungsgruppe Do 14-16
Andreas Augustin

Jan Hendrik Dithmar (Matrikelnummer 2031259)
Dirk Heine (Matrikelnummer 2030941)
Andreas Glitsch (Matrikelnummer 2026130)

Aufgabe 1

Gegeben:

$$H^{fool} = \{h_{(a,b)} : a, b \in 0...m-1\} \text{ mit } h_{(a,b)}(x) = ax + b \mod m$$

N sei die Anzahl aller Schlüssel.
Menge M von $\left\lfloor \frac{N}{m} \right\rfloor$ Schlüsseln für die gilt:

$$\forall x, y \in M : \forall h_{(a,b)} \in H^{fool} : h_{(a,b)}(x) = h_{(a,b)}(y)$$

Beweis:

$$\begin{aligned}
h_{(a,b)}(x) = h_{(a,b)}(y) &\Leftrightarrow ax + b \mod m = ay + b \mod m \\
&\Leftrightarrow ax + b + tm = ay + b + t'm \\
&\Leftrightarrow ax + tm = ay + t'm \\
&\Leftrightarrow ax + tm - t'm = ay \\
&\Leftrightarrow ax + m(t - t') = ay \\
&\Leftrightarrow ax - ay + m(t - t') = 0 \\
&\Leftrightarrow a(x - y) + m(t - t') = 0 \\
&\Leftrightarrow a(x - y) \mod m = 0
\end{aligned}$$

Aufgabe 2

Problematisch bei allen Hashstrategien mit offener Adressierung ist das Löschen von Elementen. Die belegte Tabellenposition darf nicht ohne zusätzliche Vorkehrungen freigegeben werden! „Dahinterliegende“ Elemente, die aufgrund einer Kollision auf die folgenden Plätze eingefügt wurden, könnten dadurch nicht mehr erreicht werden.
Lösung bei linear probing:
Das zu löschende Element ans Ende der Hashtabelle „shiften“ und dann dort löschen, da dann kein Element dem zu löschenden nachfolgt. Bei linear probing wird beim Einfügen bei einer Kollision so lange gesucht, bis ein freies Feld gefunden wurde; Einfügen und Suchen eines Elementes sind durch das „Shiften“ nicht beeinflusst.

Aufgabe 3

Algorithmus zum Auswerten der Kosten von Teilnehmern aus einer Log-Datei:
Ich lese die Log-Datei ein und beginne mit der ersten Zeile. Die Kosten des

dortigen Teilnehmers werden zusammen mit seinem Namen in einer Zelle gespeichert. Den Namen verwende ich in einer Suchfunktion. Bevor die Suche beginnt wird die aktuelle Zeile noch gelöscht. Ich lasse die Suchfunktion laufen und bei jedem Treffer die dort gespeicherten Kosten in der Speicherzelle des Namens hinzuzählen. Dann wird die Zeile gelöscht und die Suche fährt fort. Dasselbe Verfahren wende ich bei allen anderen Treffern an, solange bis ich das EOF erreiche. Name und Kosten werden dann ausgegeben. Dann gehe ich an den Anfang zurück, an dem jetzt ein neuer Name steht. Bei diesem wende ich dieselbe Vorgehensweise an wie schon mit dem Vorherigen. Kosten aufsummieren, Zeile löschen, Suche fortführen bis zum EOF. Die ganze Prozedur läuft solange die erste Zeile kein EOF ist.

Aufgabe 4

(i)

Pseudo-Code für die insert-Funktion des Kuckuck-hashing:

```
void insert(item y)
{
  while(true)
  {
    item tmp;
    if(t_1[h_1(y)] != leer)
    {
      tmp = t_1[h_1(y)];
      t_1[h_1(y)] = y;
    }
    else
    {
      t_1[h_1(y)] = y;
      return;
    }
    if(t_2[h_2(tmp)] != leer)
    {
      y = t_2[h_2(tmp)];
      t_2[h_2(tmp)] = tmp;
    }
    else
    {
      t_2[h_2(tmp)] = tmp;
      return;
    }
```

```
  }
}
```

(ii)

Die insert-Funktion könnte endlos laufen, wenn keine freie Speicherzelle gefunden wird. Dies kann z.B. passieren, wenn beide Arrays t_1 und t_2 voll sind und darauf die insert-Funktion angewendet wird.

Aufgabe 5

Beweis:
Sei x Knoten des Baumes, dann gilt für jeden Knoten y

- im linken Unterbaum von x: $y.key \leq x.key$
- im rechten Unterbaum von x: $y.key \geq x.key$

D.h., dass alle Elemente, die kleiner gleich der Wurzel sind, links vom aktuellen Knoten eingefügt werden und alle Elemente großer gleich der Wurzel rechts davon.
Für $n = 0$ macht `tree_walk` garnichts.

Induktionsanfang: $n = 1$
$n = 1 \Leftrightarrow$ Baum besteht nur aus Wurzel.
Die Zeiger `left_child` und `right_child` zeigen auf `NULL`, d.h. die Aufrufe von `tree_walk` für linkes und rechtes Kind machen nichts (vgl. Fall $n = 0$). Somit wird der Baum korrekt sortiert ausgegeben.
Bemerkung: Dies ist der Fall, wenn ein (Unter-)Baum keine Kinder hat.

Induktionsannahme: Die Aussage gilt für alle natürlichen Zahlen n.

Induktionsschluss: $n \rightsquigarrow n + 1$
Einem Baum mit n Knoten wird 1 Element hinzugefügt. `tree_insert` aus der Vorlesung behält die Eigenschaften des Suchbaums beim Einfügen bei, d.h., der neue Knoten wurde an der richtigen Stelle eingefügt. Da `tree_walk` zuerst rekursiv den linken Teilbaum hinabsteigt, bekommt man hier das kleinste Element im Baum. Falls kein linker Teilbaum (mehr) vorhanden ist, gibt `tree_walk` den aktuellen Knoten aus (kleinstes Element im linken Teilbaum), steigt auf und geht rekursiv in den jeweils rechten Teilbaum. Ist in diesem ein linker Unterbaum vorhanden, so arbeitet die Prozedur wie oben bereits beschrieben. Der rekursive Abstieg im rechten Teilbaum liefert (am Ende) das größte Element im Baum. Ist kein rechter Teilbaum (mehr) vorhanden, gibt `tree_walk` das aktuelle Element aus (größtes Element im Baum).

Somit gilt für jeden ausgegebenen Knoten k:

$$k_i \leq k_j \quad \forall i < j$$

und der Baum ist sortiert ausgegeben. □

Aufgabe 6

Das Löschen im binären Suchbaum ist kommutativ.
Begründung:
Fallunterscheidung:

1. Fall: **Die zu löschenden Knoten x und y sind Blätter**
   Hier ist es offensichtlich, dass die Reihenfolge bei der Löschung keine Rolle spielt.

2. Fall: **Einer der Knoten ist ein Blatt, der andere ist Teil eines Teilbaums**
   Bei der Löschung von einem Knoten wird der Baum so umsortiert, dass vorherige Blätter danach immernoch Blätter sind. Somit spielt es keine Rolle, ob das Blatt zuerst oder an zweiter Stelle gelöscht wird. Auf den Teilbaum hat die Reihenfolge ebenfalls keine Auswirkung, da die Struktur des Teilbaums bis auf das Verschieben des kleinsten Elementes an die Stelle der Wurzel ebenfalls erhalten bleibt.

3. Fall: **Beide zu löschenden Knoten x und y sind Teil eines Teilbaums**
   Wie oben gesagt wird die Struktur des Teilbaums bis auf das Verschieben des kleinsten Elements an die Stelle der Wurzel nicht verändert. Ob dabei zuerst x und dann y gelöscht wird oder umgekehrt spielt keine Rolle, da bei jeder Löschung nur das kleinste Element verschoben wird und ggf. die Kinder dieses Elements „nachrücken“.