Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dipl. Inform. Jan Küntzer

Programmierung II, SS 2004
Musterlösung zur 9. Übung

Aufgabe 1: Hashfunktion (10 Punkte)
Gegeben sei die Menge von Hashfunktionen

$$H^{fool} = \{h_{(a,b)} : a, b \in 0 \dots p-1\}$$

mit $h_{(a,b)}(x) = ax + b \mod m$.
Zeigen Sie, daß es eine Menge M von $\lfloor \frac{N}{m} \rfloor$ Schlüssel gibt, so daß gilt:

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

auch wenn m prim ist.

Beweis. Es muss gelten: $\forall x, y \in M : \forall h_{(a,b)} \in H^{fool}$:

$$\begin{aligned} h_{(a,b)}(x) &= h_{(a,b)}(y) \\ ax + b \mod m &= ay + b \mod m \\ x \mod m &= y \mod m \end{aligned}$$

Da wir N Schlüssel zur Verfügung haben und $m < N$ Restklassen, gilt nach dem Pigeon-Hole-Prinzip, dass es mindestens eine Restklasse geben muss, die mindestens zwei Schlüssel hat. Da die Restklassen gleichverteilt sind, haben wir somit genau $\lfloor \frac{N}{m} \rfloor$ Schlüssel pro Restklasse. Damit haben wir ein M mit der geforderten Eigenschaft gefunden. ■

Aufgabe 2: Löschen eines Elements (5 Punkte)

Beim Löschen eines Eintrags in einer Hashtabelle mit offener Adressierung gibt es folgende Problematik:
Betrachten wir das Beispiel

$$t = [\dots, \underset{h(z)}{x}, y, z, \dots]$$

und linear Probing. Würden wir naiv das Element y löschen und danach versuchen z zu finden, würden wir über das Loch stolpern, das beim Löschen von y entstanden ist, und denken z existiert nicht.

Das Problem kann man auf verschiedene Weise umgehen. Eine Möglichkeit wurde bereits in der Vorlesung vorgestellt. Eine andere Methode ist es, das Element zu löschen. Das dadurch entstandene Loch wird wie folgt gestopft: Zuerst ueberpruefe man wo die erste freie Stelle nach links zu finden ist. Nun laufe man vom entstandenen Loch bis zur nächsten freien Stelle nach rechts und schiebe das erste Element, welches einen Hashwert hat, der größer gleich der linken freien Stelle und kleiner gleich der aktuellen Lücke. Damit entsteht eine neue Lücke die genauso geschlossen wird. Dies geschieht so lange, bis man an der nächsten freien Stelle rechts angelangt ist.

Aufgabe 3: Anwendung Hashfunktion (5 Punkte)

Die effektivste Lösung des Problems verwendet Hashtabellen.
Man nehme eine Hashtabelle und verwende den Namen des Kunden als Schlüssel. Dabei ist zu beachten, dass der Name ein string ist und eindeutig in ein Integer umgewandelt werden muss. Nun liest man alle Log-Dateien der Reihe nach ein und fügt den Betrag unter diesem Schlüssel ein. Gibt es schon einen Eintrag, wird der Betrag dazuaddiert.
Will man die Rechnung erstellen, läuft man über alle Schlüssel und gibt den dazugehörigen Eintrag aus. Dies ist der Preis, den der Kunde bezahlen muss.

Aufgabe 4: Kuckuck-Hashing (10 Punkte)

i)
```
void insert(Element x1)

    Element x2;
    for (int i=0; i<MaxRepeat; i++)

       x2 = t1[h1(x1)];           // was ist im Weg?
       t1[h1(x1)] = x1;           // spielen wir Kuckuck
       if (x2 == empty) return;   // war nix im Weg
       x1 = t2[h2(x)];            // was ist im Weg?
       t2[h2(x2)] = x2;           // spielen wir Kuckuck
       if (x1 == empty) return;   // war nix im Weg

    throw TableOverflow();
```

ii) Der „Kuckuck-Prozess“ könnte nicht terminieren! Ein Beispiel gibt Abb.1. Allerdings kann man zeigen, dass diese Situation nur sehr selten auftritt. Sollte es trotzdem zum Schlimmsten kommen, dann nehmen wir uns eine neue Hashfunktion und reorganisieren die komplette Hashtabelle.

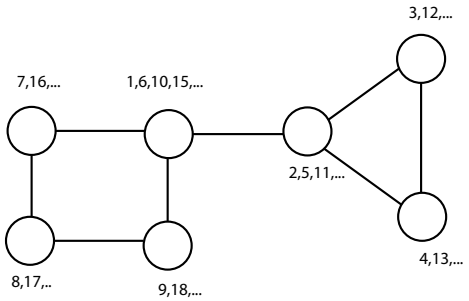

Abbildung 1: Beispiel bei dem Kuckuck-Hashing in eine Endlosschleife läuft.

Aufgabe 5: Binärer Suchbaum (5 Punkte)
Beweisen Sie induktiv, dass die aus der Vorlesung bekannte Funktion `tree_walk(TreeNode *x)` korrekt ist, also, dass die Funktion auf die Wurzel angewendet den Baum korrekt sortiert ausgibt.

Beweis. Wir führen eine Induktion über die Höhe n des Baumes.
Induktionsanfang: n=0: Wir haben nur einen Knoten, der keine Kinder hat. Also gibt `tree_walk` diesen Knoten aus. Damit ist die Ausgabe korrekt.
$n \rightarrow n+1$*:* Die Ausgabe von `tree_walk` für einen Baum der Höhe $\leq n$ sei korrekt.
Wir betrachten nun einen Baum der Höhe n+1. Da in einem binären Suchbaum der linke Unterbaum und rechte Unterbaum die Höhe n hat, gibt `tree_walk` diese nach IV korrekt aus. Da wir wissen, dass der linke Unterbaum nur kleine Elemente als die Wurzel und der rechte Unterbaum nur grössere Elemente als die Wurzel enthält, erhalten wir für unseren Baum eine korrekt sortierte Ausgabe. ■

Aufgabe 6: Kommutativität (5 Punkte)
Das Löschen im binären Suchbaum ist nicht kommutativ (siehe Abb.2)

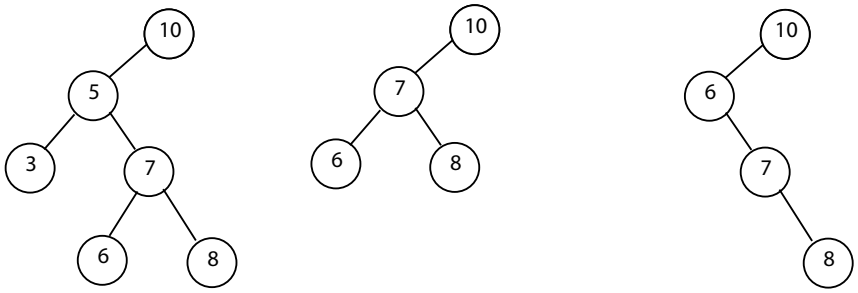

Abbildung 2: Wir betrachten den linken Graph. Den mittleren Graph erhält man, indem man zuerst 3 und dann 5 löscht. Den rechten Graph erhält man, indem man zuerst 5 und dann 3 löscht.