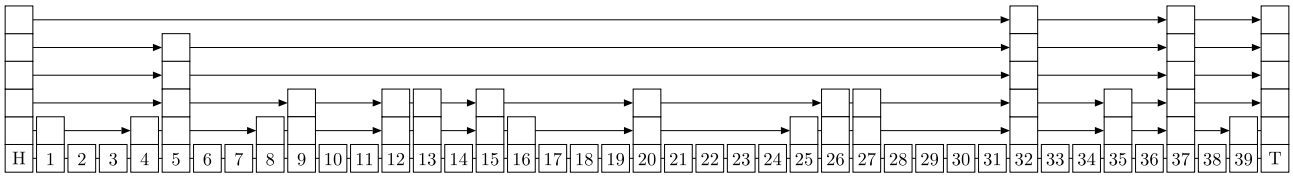
Datenstrukturen und Algorithmen, SS 2024
Prof. Dr. P. Rossmanith
D. Cloerkes, M. Gehnen, D. Mock, N. Speetzen

Abgabe: 10.05.2024

Übungsblatt mit Lösungen 04

Aufgabe T11

Geben Sie jeden Schritt im Detail an, wenn wir in folgender Skiplist nach 22, 40 und 0 suchen.



Lösungsvorschlag

Suche nach 22:

- Starte an H in oberster Liste (Ebene 5)
- $22 < 32 \Rightarrow$ Wechsel an H zur nächst tieferen Liste (Ebene 4)
- $22 > 5 \Rightarrow$ Wechsel zur 5 (Ebene 4)
- $22 < 32 \Rightarrow$ Wechsel an 5 zur nächst tieferen Liste (Ebene 3)
- $22 < 32 \Rightarrow$ Wechsel an 5 zur nächst tieferen Liste (Ebene 2)
- $22 > 9 \Rightarrow$ Wechsel zur 9 (Ebene 2)
- $22 > 12 \Rightarrow$ Wechsel zur 12 (Ebene 2)
- $22 > 13 \Rightarrow$ Wechsel zur 13 (Ebene 2)
- $22 > 15 \Rightarrow$ Wechsel zur 15 (Ebene 2)
- $22 > 20 \Rightarrow$ Wechsel zur 20 (Ebene 2)
- $22 < 26 \Rightarrow$ Wechsel an 20 zur nächst tieferen Liste (Ebene 1)
- $22 < 25 \Rightarrow$ Wechsel an 20 zur nächst tieferen Liste (Ebene 0)
- $22 > 21 \Rightarrow$ Wechsel zur 21 (Ebene 0)
- $22 = 22 \Rightarrow$ 22 gefunden nach 13 Vergleichen

Suche nach 40:

- Starte an H in oberster Liste (Ebene 5)
- $40 > 32 \Rightarrow$ Wechsel zur 32 (Ebene 5)
- $40 > 37 \Rightarrow$ Wechsel zur 37 (Ebene 5)
- Nächstes Element ist $T \Rightarrow$ Wechsel an 37 zur nächst tieferen Liste (Ebene 4)

- Nächstes Element ist $T \Rightarrow$ Wechsel an 37 zur nächst tieferen Liste (Ebene 3)
- Nächstes Element ist $T \Rightarrow$ Wechsel an 37 zur nächst tieferen Liste (Ebene 2)
- Nächstes Element ist $T \Rightarrow$ Wechsel an 37 zur nächst tieferen Liste (Ebene 1)
- $40 > 39 \Rightarrow$ Wechsel zur 39 (Ebene 1)
- Nächstes Element ist $T \Rightarrow$ Wechsel an 39 zur nächst tieferen Liste (Ebene 0)
- Nächstes Element ist $T \Rightarrow$ Suche endet erfolglos nach 9 Vergleichen

Suche nach 0:

- Starte an H in oberster Liste (Ebene 5)
- $0 < 32 \Rightarrow$ Wechsel an H zur nächst tieferen Liste (Ebene 4)
- $0 < 5 \Rightarrow$ Wechsel an H zur nächst tieferen Liste (Ebene 3)
- $0 < 5 \Rightarrow$ Wechsel an H zur nächst tieferen Liste (Ebene 2)
- $0 < 5 \Rightarrow$ Wechsel an H zur nächst tieferen Liste (Ebene 1)
- $0 < 1 \Rightarrow$ Wechsel an H zur nächst tieferen Liste (Ebene 0)
- $0 < 1 \Rightarrow$ Suche endet erfolglos nach 6 Vergleichen

Aufgabe T12

Wir verwenden jetzt folgende universelle Familie von Hashfunktionen:

$$\mathcal{H}_{\text{T12}} = \{\, h_{a,b} \mid 1 \leq a < 5,\ 0 \leq b < 5 \,\}$$

mit $$h_{a,b}(x) = ((ax + b) \bmod 5) \bmod 4.$$

Berechnen Sie die Wahrscheinlichkeit, dass $h(2) = h(3)$ ist, falls wir h zufällig aus obiger Familie von Funktionen wählen. Was müsste herauskommen, wenn man bedenkt, dass es sich um eine universelle Familie von Hashfunktionen handelt?
Da es sich um viele Funktionen handelt, sollte die Arbeit unter vielen Personen verteilt werden, damit es schneller geht. Wie lange brauchen Sie gemeinsam, um diese Wahrscheinlichkeit ohne Taschenrechner oder ähnlichem zu berechnen?

Lösungsvorschlag

Es gibt insgesamt vier Möglichkeiten für a und fünf Möglichkeiten für b, so dass unsere Familie aus 20 verschiedenen Funktionen $h_{a,b}$ besteht. Für jede dieser Funktionen müssen wir nun herausfinden, ob $h_{a,b}(2) = h_{a,b}(3)$ gilt. Wir berechnen also einfache beide benötigte Funktionswerte für alle 20 Funktionen:

a	b	$h_{a,b}(2)$	$h_{a,b}(3)$
1	0	2	3
1	1	3	0
1	2	0	0
1	3	0	1
1	4	1	2
2	0	0	1
2	1	0	2
2	2	1	3
2	3	2	0
2	4	3	0
3	0	1	0
3	1	2	0
3	2	3	1
3	3	0	2
3	4	0	3
4	0	3	2
4	1	0	3
4	2	0	0
4	3	1	0
4	4	2	1

Wir sehen, dass es genau zwei Kollisionen gibt. Da es 20 verschiedene Funktionen sind, ist die Wahrscheinlichkeit einer Kollision also genau 1/10. Aus der Theorie der universellen Hashfunktionen erwarten wir, dass die Kollisionswahrscheinlichkeit höchstens $1/m = 1/4$ beträgt. Mit 1/10 wird dieser garantierte Wert sogar noch unterboten.

Aufgabe T13

Wir betrachten die Hashfunktion $h\colon \Sigma^* \to \{0, \ldots, m-1\},\ c_1 c_2 \ldots c_k \mapsto \left(\sum_{i=1}^{k} c_i\right) \bmod m$, welche die Menge Σ^* aller Wörter (im ASCII-Alphabet) auf eine Zahl zwischen 0 und $m-1$ abbildet.

a) Ist h ein gute Hashfunktion? Überlegen Sie sich realistische Anwendungsbeispiele, für welche h sehr viele Kollisionen erzeugt.

b) Wie könnte eine vernünftige Hashfunktion für Zeichenketten aussehen, für welche Sie unter normalen Umständen ein gutes Verhalten erwarten würden? Wie gehen Sie mit dem Fall um, dass m sehr groß ist?

Lösungsvorschlag

a) Wenn wir Wörter hashen, die alle Permutationen voneinander sind, dann werden *alle* auf denselben Wert abgebildet – schlechter geht es wirklich nicht. Außerdem werden alle kurzen Wörter auf relativ kleine Zahlen abgebildet, so dass nur ein verschwindend kleiner Teil einer sehr großen Hashtabelle verwendet würde. Dort gäbe es dann sehr viele Kollisionen.

b) Eine einfache Möglichkeit ist eine Linearkombination der c_i mit geeigneten Koeffizienten, die mit steigendem i auch größer werden, zum Beispiel $\sum_{i=1}^{k} c_i \cdot p^i \bmod m$ für Primzahlen p. So werden auch die Hashwerte auch groß genug, um die Hashtabelle der Größe m aufzufüllen.

Aufgabe T14

Ist die folgende Behauptung wahr? Wenn nein, dann finden Sie ein Gegenbeispiel und den Fehler im angeblichen Beweis.

Behauptung: Für alle $x \in U$, alle universelle Familien $\mathcal{H}$ von Hashfunktionen $U \to \{1, \ldots, m\}$ und alle $k \in \{1, \ldots, m\}$ gilt: $\Pr[h(x) = k] = 1/m$, falls h zufällig aus $\mathcal{H}$ gewählt wurde.

Beweis: Nehmen wir an, es gibt ein y mit $h(y) = k$. Dann setze $S = \{y\}$ und wende das letzte Theorem an:

$$E\Big(|\{\, y \in S \mid h(x) = h(y)\,\}|\Big) \leq \frac{|S|}{m}.$$

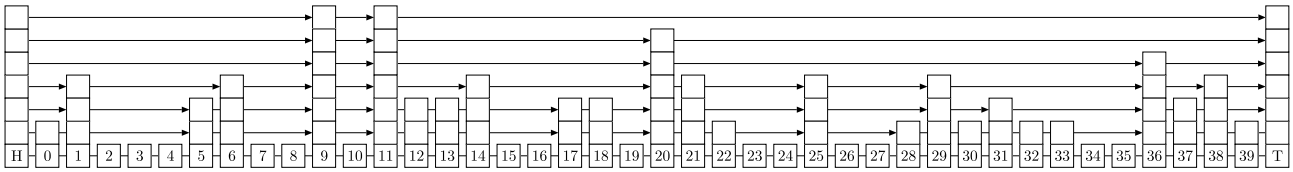
Hier folgt daraus $\Pr[h(x) = k] \leq 1/m$.
Wenn es kein y mit $h(y) = k$ gäbe, dann wäre $\Pr[h(x) = k] = 0$.
Das geht aber nicht, denn $\sum_{k=1}^{m} \Pr[h(x) = k] = 1$.

Lösungsvorschlag

Die Behauptung ist falsch. Der erste Satz im „Beweis“ ist bereits Unsinn. Es macht keinen Sinn zu fragen, ob es ein solches y gibt, denn $h(x)$ ist eine Zufallsvariable und hat daher keinen festen Wert. Es würde nur Sinn machen zu fragen, wie groß die Wahrscheinlichkeit ist, das $h(y) = k$.
Ein Gegenbeispiel: Wir betrachten die Familie $\mathcal{H} = \{\, x \mapsto ((ax + b) \bmod 3) \bmod 2 \mid 0 < a \leq 2, 0 \leq b \leq 2\,\}$ für das Universum $\mathcal{U} = \{0, 1, 2\}$ und die Hashtabellengröße $m = 2$. Laut Vorlesung ist $\mathcal{H}$ universell (mit $p = 3, m = 2$). Der Input $x = 1$ wird unter dieser Familie zweimal auf 1 abgebildet, aber viermal auf 0. D.h., dass $\Pr[h(1) = 0] = 4/6 \neq 1/2 = 1/m$.

a	b	$h_{a,b}(1)$
1	0	1
1	1	0
1	2	0
2	0	0
2	1	0
2	2	1

Aufgabe H12 (6 Punkte)
Geben Sie jeden Schritt im Detail an, wenn wir in folgender Skiplist nach 10, 19 und 93 suchen.



Lösungsvorschlag
Suche nach 10:

- Starte an H in oberster Liste (Ebene 6)
- $10 > 9 \Rightarrow$ Wechsel an 9.
- $10 < 11 \Rightarrow$ Wechsel an 9 zur nächst tieferen Liste (Ebene 5)
- ... (das bis auf Ebene 0)
- $10 == 10 \Rightarrow$ 10 gefunden nach 8 Vergleichen.

Suche nach 19:

- Starte an H in oberster Liste (Ebene 6)
- $19 > 9 \Rightarrow$ Wechsel an 9.
- $19 > 11 \Rightarrow$ Wechsel an 11.
- Nächstes Element ist $T \Rightarrow$ Wechsel an 11 zur nächst tieferen Liste (Ebene 5)
- $19 < 20 \Rightarrow$ Wechsel an 11 zur nächst tieferen Liste (Ebene 4)
- $19 < 20 \Rightarrow$ Wechsel an 11 zur nächst tieferen Liste (Ebene 3)
- $19 > 14 \Rightarrow$ Wechsel an 14.
- $19 < 20 \Rightarrow$ Wechsel an 11 zur nächst tieferen Liste (Ebene 2)
- $19 > 17 \Rightarrow$ Wechsel an 17.
- $19 > 18 \Rightarrow$ Wechsel an 18.
- $19 < 20 \Rightarrow$ Wechsel an 18 zur nächst tieferen Liste (Ebene 1)
- $19 < 20 \Rightarrow$ Wechsel an 18 zur nächst tieferen Liste (Ebene 0)
- $19 == 19 \Rightarrow$ 11 gefunden nach 11 Vergleichen.

Suche nach 93:

- Starte an H in oberster Liste (Ebene 6)
- $93 > 9 \Rightarrow$ Wechsel zur 9
- $93 > 11 \Rightarrow$ Wechsel zur 11
- Nächstes Element ist $T \Rightarrow$ Wechsel an 11 zur nächst tieferen Liste (Ebene 5)

- $93 > 20 \Rightarrow$ Wechsel zur 20
- Nächstes Element ist $T \Rightarrow$ Wechsel an 20 zur nächst tieferen Liste (Ebene 4)
- $93 > 36 \Rightarrow$ Wechsel zur 36
- Nächstes Element ist $T \Rightarrow$ Wechsel an 36 zur nächst tieferen Liste (Ebene 3)
- $93 > 38 \Rightarrow$ Wechsel zur 38
- Nächstes Element ist $T \Rightarrow$ Wechsel an 38 zur nächst tieferen Liste (Ebene 2)
- $93 > 39 \Rightarrow$ Wechsel zur 39
- Nächstes Element ist $T \Rightarrow$ Wechsel an 39 zur nächst tieferen Liste (Ebene 1)
- Nächstes Element ist $T \Rightarrow$ Suche endet erfolglos nach 9 Vergleichen

Aufgabe H13 (4+1+3 Punkte)

a) Erstellen Sie mithilfe eines Programms eine Tabelle, welche die Wahrscheinlichkeiten von $h(x) = h(y)$ für alle $0 \leq x, y < 5$ enthält, falls h wieder zufällig aus der Familie $\mathcal{H}_{\text{T12}}$ von T12 gezogen wird.

b) Ist das Ergebnis das, was Sie erwarten? Warum?

c) Wiederholen Sie das Experiment für $0 \leq x, y < 6$, $1 \leq a < 6$, $0 \leq b < 6$ und $h_{a,b}(x) = ((ax + b) \bmod 6) \bmod 4$. Kommentieren Sie das Ergebnis. Nehmen Sie insbesondere dazu Stellung, ob es sich auch jetzt um eine universelle Familie von Hashfunktionen handelt.

Lösungsvorschlag
Das Programm in Abbildung 1 erledigt die Arbeit für $p = 5$ und $p = 6$, wobei es nicht die Wahrscheinlichkeiten, sondern die Gesamtzahl der Kollisionen ausgibt. Um Wahrscheinlichkeiten zu erhalten, muss man noch durch $p(p-1)$ teilen, was aber hässliche Brüche ergibt.

Die Ausgabe des Programms ist:

```
H13a
20 2 2 2 2
2 20 2 2 2
2 2 20 2 2
2 2 2 20 2
2 2 2 2 20
H13b
30 4 14 12 14 4
4 30 4 14 12 14
14 4 30 4 14 12
12 14 4 30 4 14
14 12 14 4 30 4
4 14 12 14 4 30
```

Natürlich sind die Kollisionswahrscheinlichkeiten stets 1, wenn $x = y$. Ansonsten sind sie für $p = 5$ stets $2/20$. Wir erwarten natürlich, dass sie höchstens $1/4$ sind. Für $p = 6$ sieht es bitterer aus, da hier Kollisionswahrscheinlichkeiten von bis zu $14/30$ auftreten, was deutlich höher als $1/m = 1/4$ ist. Allerdings ist das Theorem aus der Vorlesung dadurch nicht widerlegt, weil ja 6 keine Primzahl ist. Ganz im Gegenteil: Dieses Beispiel demonstriert, wie wichtig es ist, eine Primzahl zu wählen.

```
#include ⟨stdio.h⟩

int p;
int h(int a, int b, int x) {
  return((a * x + b)%p)%4;
}
int countcollisions(int x, int y) {
  int a, b, count = 0;
  for(a = 1; a < p; a++) {
    for(b = 0; b < p; b++) {
      if(h(a, b, x) == h(a, b, y)) {
        count++;
      }
    }
  }
  return count;
}
void printtable() {
  int x, y;
  for(x = 0; x < p; x++) {
    for(y = 0; y < p; y++) {
      printf("%d ", countcollisions(x, y));
    }
    printf("\n");
  }
}
int main() {
  p = 5;
  printf("H13a\n");
  printtable();
  p = 6;
  printf("H13b\n");
  printtable();
}
```

Abb. 1: Programm für Aufgabe H13

Aufgabe H14 (7+7 Punkte)

Im April 2020 mussten das Studium plötzlich komplett von zuhause aus gemacht werden. So auch FoSAP, welches wir damals gehalten haben; Prof. Rossmanith hielt die Vorlesung über Zoom und Daniel hat den Chat betreut. Dies war auch zwingend nötig, da sich einige Studis in ihrer Pseudonymität schnell zu wohl gefühlt haben. Da die Moderationsmöglichkeiten in Zoom ziemlich beschränkt waren, schlägt Daniel vor, so einen Chatraum selber zu realisieren.

In einem Chatraum gibt es eine feste Menge von Studis S, die alle durch eine eindeutige IntegerID s identifiziert sind. Jeder Studi s kann Nachrichten in den Chatraum schicken. Jeder Studi kann die letzten k Nachrichten lesen, wobei k ein selbstgewählter Wert ist. Sollte Daniel das Verhalten eines Studis nicht gefallen, so kann er diesen bannen; d.h. es werden alle Nachrichten von diesem Studi gelöscht und er kann keine neuen senden.[1] Genauer gesagt, gibt es folgende vier Operationen, die es umzusetzen gilt:

- *erstellen*(S): Initialisieren des Chatraumes mit $n = |S|$ Studis.
- *senden*(s, m): Sende Nachricht m zum Chat von Studi s (außer er ist gebannt).
- *letzte*(k): Gib die letzten k (oder alle, falls es weniger als k Nachrichten gibt) nichtgelöschten Nachrichten aus.
- *ban*(s): Banne Studi s und lösche alle seine Nachrichten.

Random-Access ist hier nicht nötig; der Chat ist lediglich ein Strom von affektiven Reaktionen ohne Gedächtnis.

Geben Sie für die folgendenen Teilaufgaben die Datenstrukturen an und wie die einzelnen Operationen mittels der Datenstrukturen realisiert werden. Erläutern Sie, wie die Laufzeiten zustande kommen.

(a) Entwickeln Sie Datenstrukturen, die die obigen Operationen unterstützt, mit folgenden worst-case Laufzeiten:

- *erstellen*(S) in $O(n \log n)$ Zeit,
- *senden*(s, m) in $O(\log n)$ Zeit,
- *letzte*(k) in $O(k)$ Zeit,
- *ban*(s) in $O(n_s + \log n)$ Zeit, wobei n_s die Anzahl der Nachrichten von Studi s ist.

(b) Ihnen fällt auf, dass die Vorlesungen FoSAP bzw. DSAL so beliebt sind, dass die obigen Laufzeiten zu langsam sind. Entwerfen Sie Datenstrukturen mit folgenden *amortisierten worst-case Laufzeiten im Erwartungswert*:

- *erstellen*(S) in $O(1)$ Zeit,
- *senden*(s, m) in $O(1)$ Zeit,
- *letzte*(k) in $O(k)$ Zeit,
- *ban*(s) in $O(1)$ Zeit.

Hinweis: Das „im Erwartungswert“ bezieht sich auf die zufällige Wahl der Funktion aus einer universellen Familie von Hashfunktionen; worst-case bezieht sich auf den Input.

Hinweis: Kombinieren Sie aus der Vorlesung bekannte Datenstrukturen auf geschickte Weise. Außerdem ist explizit nach keinem Programm oder Pseudocode gefragt, sondern nach einer prosaischen Beschreibung.

[1] Inspiriert durch eine Aufgabe aus der Vorlesung 6.006 „Intro to Algorithms“ vom MIT.

Lösungsvorschlag

(a) Wir implementieren die Aufgabe mittels einer doppelt-verketteten Liste L, die chronologisch Nachrichten m mit Absender s enthält, und einem AVL-Baum. Die Schlüssel für den AVL-Baum sind die Studis S und der Wert für einen Studi s ist eine verkettete Liste L_s, die für jede von s gesendete Nachricht eine Referenz auf die Position der Nachricht in L enthält (andere balancierte Bäume wie Splaybaum oder Rot-Schwarz-Baum gingen auch).

- *erstellen*(S): Initialisiere L und S je in konstanter worst-case Zeit (jeweils leer).
- *senden*(s, m): Finde s in S in $O(\log n)$ Zeit. Falls s als gebannt markiert ist, tue nichts. Sonst, hänge (s, m) hinten in L an und hänge den Pointer an L_s an, der auf das Listenelement von (s, m) in L zeigt, in $O(1)$ worst-case Zeit.
- *letzte*(k): Gebe die letzten k Elemente aus L aus.
- *ban*(s): Finde s in S, in Zeit $O(\log n)$. Mittels der Referenzen in dem Knoten von s in S, lösche dessen Nachrichten in L (mittels Umbiegen der Pointer benachbarten Listenelemente je in $O(1)$ Zeit) in insg Zeit $O(n_s)$. Markiere s in B als gebannt.

(b) Wir implementieren die Aufgabe mittels einer doppelt-verketteten Liste L, die chronologisch Nachrichten m mit Absender s enthält, und einer Menge B von gebannten Nutzern (mittels eines Hashsets). Diese wächst dynamisch mit und behält einen konstanten Lastfaktor (wie in der VL). Die Zeiten für hinten einfügen, iterieren und (mit Referenz) löschen sind bei L konstant. Das Hinzufügen und Testen auf Mitgliedschaft in B ist auch konstant (im amortisierten erwarteten worst-case).

- *erstellen(S)*: Initialisiere L als leere Liste und B als leere Menge je in konstanter worst-case Zeit.
- *senden(s, m)*: Hänge (s, m) hinten in L an, in konstanter worst-case Zeit.
- *ban(s)*: Füge s in B hinzu, in konstanter expected worst-case Zeit.
- *letzte(k)*: Iteriere L von hinten. Für den aktuell betrachteten Eintrag (s, m): Ist $s \in B$, lösche (s, m) aus L in konstanster worst-case Zeit, indem die benachbarten Zeiger umgebogen werden. Sonst gib (s, m) aus und fahre fort, bis k Nachrichten ausgegeben wurden (oder die Liste durchiteriert ist).

  Zur Laufzeitanalyse: Im *(expected) worst-case* dauern die Operationen alle Operationen konstante Zeit, nur die *letzte*-Operation braucht länger als das. Eine Abschätzung wäre $O(|L|)$. Man beachte, dass das *erstellen* nicht konstante Zeit braucht, wenn man es als leere Menge mit Kapazität n initialisiert!

  Für die amortisierte Laufzeitanalyse wählen wir als Potenzialfunktion $\Phi(i) = cl_i$, wobei l_i die Anzahl der Elemente in L zum Zeitpunkt i ist und c eine passend gewählte Konstante (z.B. das Maximum der Konstanten $c_1, ...$). Diese ist natürlich zu jedem Zeitpunkt nicht-negativ und anfangs 0.

  Die amortisierte Analyse für die drei ersten Operationen ist simpel: Für den Fall *senden* ist die reele Laufzeit t_i durch eine Konstante c_2 abzuschätzen, die Zugriff auf L ausdrückt.

  Der interessante Falle *letzte*: Nennen wir die Anzahl der betrachteten Elemente in L, bis man k nicht-gebannten Nachrichten gefunden hat, d_i. Damit kann die echte Laufzeit t_i dieser Operation mit $c_1 d_i$ für eine konstante c abgeschätzt werden, die die Zeit für je einen Zugriff auf das Hashset B und einen Iterationsschritt in der Liste L ausdrückt. Man beachte, dass man diese d_i Nachrichten bis auf die

$\leq k$ nicht-gebannten Nachrichten aus L entfernt. Damit sinkt die Potentialfunktion um mindestens $(d_i - k)c$. Damit kann man nun die amortisierten Kosten für *letzte* abschätzen:

$$t_i + \Phi(i) - \Phi(i-1) \leq c_1 d_i - c(d_i - k) \leq ck$$

. Damit ist die amortisierte Laufzeit dieser Operation $O(k)$.

Anmerkungen: Man kann auch unseren Lösungsansatz aus Teil (a) anpassen, indem man die Bäume durch Hashmaps ersetzt und eine sehr ähnliche amortisierte Analyse macht.

Aufgabe H15 (3+6+3 Punkte)

Am Ende dieser Aufgabe und in Moodle finden Sie das Programm `WordCount13.java`, das eine Datei lesen kann, welche je Zeile ein Wort enthalten soll. Das Programm zählt, wie viele verschiedene Wörter es in der Datei gibt, wobei es eine Hashtabelle verwendet.

a) Wenn Sie das Programm auf einer typischen Datei laufen lassen, wie lang ist die Laufzeit auf Ihrem Computer typischerweise für eine Datei mit n Bytes?

b) Erzeugen Sie eine Eingabedatei, deren Größe 1MB nicht überschreitet und deren Bearbeitung länger als 5 Sekunden dauert. Laden Sie diese Datei mit in Moodle hoch. Erklären Sie, wie Sie die Datei erstellt haben.

c) Könnten Sie solch eine furchtbare Datei auch dann *leicht* erstellen, wenn im Programm h nicht mit x sondern mit y multipliziert würde? (Eine sehr kurze Antwort reicht hier.)

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.*;

public class WordCount13 {
    static int x = 13;
    static int y = Math.abs((new Random()).nextInt());

    public static int hash(String w) {
        int h = 0;
        for(int i = 0; i < w.length(); i++) {
            h = h * x + w.charAt(i);
        }
        return Math.abs(h);
    }

    public static void main(String args[]) throws IOException {
        String filename = args[0];
        List<String> words = Files.readAllLines(Paths.get(filename));
        int m = 100000;
        long count = 0;
        ArrayList<LinkedList<String>> slot = new ArrayList<>();
        for(int i = 0; i < m; i++) {
            slot.add(new LinkedList<String>());
        }
        for(String w : words) {
            LinkedList<String> list = slot.get(hash(w)%m);
            if(!list.contains(w)) {
                count = count + 1;
                list.add(w);
            }
        }
        System.out.println(count + " different words in " + filename + ".");
    }
}
```

Lösungsvorschlag

a) Falls wir das Programm auf zufällig generierten Eingaben laufen lassen, können wir beobachten, dass es sehr schnell ist: Wir haben uns dafür einige Listen verschiedener Längen mit zufällig generierten, 10 Zeichen langen Worten erstellt. Wir bermerken, dass wir etwa 0.1 s/Mb auf unserem Rechner benötigen. Offensichtlich benötigt das Programm lineare Laufzeit in der Anzahl der Wörter einer Datei, wir können jedoch feststellen, dass die Einfügeoperationen in etwa konstante Zeit benötigen. Wenn wir die Anzahl der eingefügten Wörter gegen unendlich laufen lassen, wird die Zeit für das Einfügen in die konstant große Hashtabelle aufgrund der vielen Kollisionen jedoch gegen $O(n^2)$ konvergieren.

b) Um eine lange Laufzeit zu erzwingen, wollen wir viele Wörter mit dem gleichen Hash einfügen, welche dann alle in einer einfachen Liste gespeichert werden. Dies führt zu einer quadratischen Laufzeit.

   Folgendes Programm zählt Dezimalzahlen auf und speichert alle, die einen Hashwert von 115 haben.

```
import java.util.*;

public class BadWords13 {

  public static int hash(String w) {
    int h = 0;
    for(int i = 0; i < w.length(); i++) {
      h = h * 13 + w.charAt(i);
    }
    return Math.abs(h);
  }

  public static void main(String args[]) {
    Random gen = new Random();
    long l = 0;
    int len = 0;
    while(len < 900000) {
      String w = "" + l;
      int h = hash(w)%100000;
      l = l + 1;
      if(h ≠ 115) continue;
      System.out.println(w);
      len += w.length() + 1;
    }
  }
}
```

Das Erstellen dauert 3,5 Minuten auf einem Intel i7-9700.

Dieser Computer braucht für die von obigem Programm erzeugte Datei immerhin 11,1 Sekunden.

c) Nein können wir nicht, da wir dann keine Chance haben vorher zu wissen auf welchen Hash ein Wort abgebildet wird.