TU Ilmenau, Fakultät für Informatik und Automatisierung
FG Komplexitätstheorie und Effiziente Algorithmen
Univ.-Prof. Dr. M. Dietzfelbinger, Philipp Schlag, Stefan Walzer, Alexander Jeuk
https://www.tu-ilmenau.de/iti/lehre/lehre-ss-2020/aud/

TECHNISCHE UNIVERSITÄT
ILMENAU

Algorithmen und Datenstrukturen SS 2020 – Praktikum 3

Zur Bearbeitung zwischen 22. Juni - 26. Juni (KW 26)

Ziel dieses Praktikums ist die Implementierung einer Hashtabelle mit Strings als Schlüsseln und leerem Datenteil. Dazu sollen lineare Hashfunktionen sowie geschlossenes Hashing mit linearem Sondieren verwendet werden. Wiederholen Sie daher sorgfältig Kapitel 5 der Vorlesung, insbesondere die Inhalte:

- *Lineare Hashfunktionen für Strings.* Druckfolien 27 und 28.
- *Lineares Sondieren.* Druckfolien 39 bis 43.
- *Behandlung von Löschungen.* Druckfolien 62 bis 66.
- *Überlaufbehandlung.* Druckfolien 67 und 68.

Hilfreiche Neuerungen

- Ab sofort gibt es eine vierte Debug-Konfiguration `DIFF *.in`. Wenn Ihr Programm auf `sample.in` bereits die richtige Ausgabe produziert, können Sie damit zusätzlich das Verhalten auf sämtlichen `in`-Dateien, die sich im Ordner Ihres Programmes befinden, untersuchen.
- Beim Einreichen eines Programms im DOMjudge-System erhalten Sie jetzt noch präziseres Feedback, wenn Sie den entsprechenden Eintrag unter `Home > Submissions` auswählen.

Aufgabe 1 (Geschlossenes Hashing)

Konzentrieren Sie sich zuerst auf die Ein-/Ausgabedateien `sample.(in/out)`. Danach können Sie Ihre Lösung noch mit allen anderen Ein-/Ausgabedateien testen und für das DOMjudge-Problem **p3-a1-hashing** einreichen. Die Eingabe startet mit einer Zahl (bei `sample.in` mit 10), die sich als Größe für die zu verwendende Hashtabelle eignet. Danach folgt ein Befehl pro Zeile, mit folgenden Bedeutungen:

h $\langle s \rangle$**.** Berechne den Hashwert des Strings s und gebe ihn aus.
Bemerkung: Die erste Sondierposition eines Schlüssels s ergibt sich als Hashwert von s modulo Tabellengröße. Im vorliegenden Fall mit Tabellengröße 10 ist das einfach die letzte Stelle des Hashwerts. Zum Beispiel ist die erste Sondierposition des Schlüssels „`der`“ Position 6.

i $\langle s \rangle$**.** Füge den Schlüssel s in die Hashtabelle unter Verwendung von linearem Sondieren ein. Einen Datenteil gibt es in dieser Aufgabe nicht. Ist s bereits in der Hashtabelle enthalten, soll einfach *nichts* passieren.

l $\langle s \rangle$**.** Suche den Schlüssel s in der Hashtabelle. Gib 1 aus, falls der Schlüssel enthalten ist, 0 sonst.

d $\langle s \rangle$**.** Lösche den Schlüssel s aus der Hashtabelle.

p $\langle s \rangle$**.** Gib den Inhalt der Hashtabelle in Form einer Folge der dort eingetragenen Schlüssel aus.
(Den Wert von s können Sie ignorieren, er dient nur dazu, dass alle möglichen Befehle dieselbe Form haben.)

(a) Implementieren Sie die Funktion `linHash`, wie in der Vorlesung auf Druckfolien 27 und 28 erklärt. Eine geeignete Primzahl `p` und zufällig gewählte Koeffizienten `a[0]`, ..., `a[19]` $\in \{0, \ldots, p-1\}$ finden Sie bereits in der Datei `hash.cpp`. Sie dürfen davon ausgehen, dass jeder String der Eingabe Länge höchstens 20 hat. Behandeln Sie Strings der Länge $\ell < 20$ so, als wären sie Strings der Länge 20, wobei die letzten $20 - \ell$ Zeichen implizit den numerischen Wert 0 haben.

Sie sollten Ihr Programm bereits jetzt mit `sample.in` testen um herauszufinden, ob die ersten acht Zeilen der Ausgabe korrekt sind.

Vorsicht: Je nachdem, wie Sie die Funktion implementieren, könnten Zwischenergebnisse entstehen, die nicht in den 32-bit Datentyp `int` passen. Sie können in dem Fall den 64-bit Datentyp `int64_t` verwenden.

(b) Wenn Sie `hash.cpp` als Starthilfe verwenden, müssen Sie sich nicht mehr um die Umsetzung des **p**-Befehls kümmern. Andernfalls sollten Sie diese Funktionalität jetzt implementieren.

(c) Implementieren Sie die Funktionen `insert` und `lookup`, wie geschlossenes Hashing mit linearem Sondieren dies vorsieht (Druckfolien 39 bis 43). Die leeren Tabellenzellen können Sie einfach daran erkennen, dass sie den leeren String `EMPTY` enthalten.[1]

(d) Implementieren Sie die Funktion `_delete`, wie in Druckfolien 62 bis 66 der Vorlesung beschrieben. Beachten Sie, dass durch das Löschen von Einträgen auch Anpassungen in `insert` und `lookup` nötig werden. Um den Status *gelöscht* einer Tabellenzelle zu repräsentieren, weisen Sie ihr einfach den String `DELETED` zu.[2]

Listing 1: `sample.in`

```
10
h der
h die
h das
h zu
h und
h nein
h sei
h dem
p 0
i der
i die
i das
i zu
i und
p 1
l der
l zu
l und
l nein
d zu
l und
l zu
d nein
i sei
i dem
l dem
d dem
p 2
```

Listing 2: `sample.ans`

```
h(der) = 5893556
h(die) = 626111
h(das) = 14264785
h(zu) = 2084325
h(und) = 16336476
h(nein) = 3379809
h(sei) = 12869179
h(dem) = 12779219
(,,,,,,,,,)
(,die,,,,das,der,zu,und,)
1
1
1
0
1
0
1
(-,die,,,,das,der,-,und,sei)
```

[1] Das ist nicht ganz sauber, denn der leere String $\varepsilon =$ „“ ist ja eigentlich ein ganz gewöhnlicher String. Wir versprechen aber, dass der leere String in dieser Aufgabe nicht als Schlüssel vorkommt.

[2] Wie bei `EMPTY == ""` ist das nicht ganz sauber, denn der String `DELETED == "-"` ist auch ein ganz gewöhnlicher String. Auch hier versprechen wir, dass er in dieser Aufgabe nicht als Schlüssel vorkommt.

Wenn alles übrige funktioniert, versuchen Sie sich an folgender **Zusatzaufgabe**.

Aufgabe 2 (Überlaufbehandlung) optional

Die Eingabedateien des DOMjudge-Problems **p3-a2-hashing-overflow** enthalten keine empfohlene Tabellengröße, stattdessen geht es direkt mit dem ersten Befehl los. Sie befinden sich also in dem in der Praxis sehr häufigen Fall, dass Sie nicht wissen, wie viele verschiedene Schlüssel es geben wird und wie groß die Hashtabelle entsprechend sein sollte. Erzeugen Sie deshalb zu Beginn eine Tabelle konstanter Größe, zum Beispiel Größe 2.
Vergrößern Sie Ihre Hashtabelle nun *bei Bedarf*, nach Vorbild von Druckfolien 67 und 68 der Vorlesung. Sie müssen dazu die zusätzlichen Variablen `inUse` und `load` mitführen und nach jeder `insert` Operation überprüfen, ob der `inUse` Wert einen kritischen Wert überschreitet. Wir empfehlen $\alpha_0 = 0.7$, $\alpha_1 = 0.8 \cdot \alpha_0$ und den Test `inUse + 1` $> \alpha_0 \cdot$ `table.size()`. Reagieren Sie dann durch Neuaufbau der Tabelle entweder mit unveränderter oder verdoppelter Größe, wie in der Vorlesung erklärt.
Bemerkung: Da die Ausgabe des **p**-Befehls auch von der konkreten Wahl von α_0 und α_1 abhängt, enthalten die Eingabedateien diesen nicht mehr. Dadurch steht es Ihnen frei, Ihre Implementierung für beliebig gewählte Werte von α_0 und α_1 zu testen.