Algorithmik SS 2006 – Praktikum 3: Hashtechniken

Aufgabe 9: von Mises' Geburtstagsparadoxon

Aufgabe 10: Sondierungstechniken

Aufgabe 11: Berechnung von Erwartungswerten

Aufgabe 12: Universelles Hashing

Aufgabe 13: Simulation mit Experimentierumgebung

Aufgabe 14: Dreieckszahlen Hashing

Name: ..... Matr-Nr: .....

Aufgabe 10 (von Mises' Geburtstagsparadoxon)

Kollisionen kommen häufiger vor, als man intuitiv vermutet. Bekannt ist dieses Fakt als das *von Mises' Geburtstagsparadoxon*: Befinden in einem Raum mindestens 23 Personen, dann ist die Wahrscheinlichkeit, dass zwei oder mehr Personen am selben Tag Geburtstag haben, größer als 50 %.

Lit.: R. von Mises. Über Aufteilungs- und Besetzungs-Wahrscheinlichkeiten. Revue de la Faculté des Sciences de l'Université d'Istanbul, N.S. 4. 1938-39, S. 145-63

Übertragen auf das Kollisionsproblem bedeutet dies:

> Bei einer Hashtabelle T mit 365 Slots und 23 zu speichernden Schlüsseln ist die Wahrscheinlichkeit einer Kollision größer als 50% .

Berechnen Sie die Wahrscheinlichkeit P(n) für ein oder mehrere Kollisionen, wenn man n Schlüssel in eine Hashtabelle T der Größe 365 einfügt.

Programmieren Sie die Funktion P(n), und geben Sie in einem Koordinatensystem die Ergebnisse (Wahrscheinlichkeiten) aus für n = 0, 5, 10, ..., 80 Schlüssel (x-Achse). Lesen Sie den Wert für n=23 aus der Kurve ab.

Lösungshinweis

> Sei Q(n) die Wahrscheinlichkeit, dass n Schlüssel zufällig in einer Hashtabelle T der Größe m=365 gespeichert werden und keine Kollision auftritt. Sei P(n) die Wahrscheinlichkeit für mindestens eine Kollision. Dann gilt:
>
> $$P(n) = 1-Q(n)$$
>
> da 1 minus die Wahrscheinlichkeit für keine Kollision gleich der Wahrscheinlichkeit für mindestens eine Kollision ist.
>
> Es gilt: Q(1) = 1. Überlegen Sie sich Q(2), Q(3), ... Sie erhalten eine Differenzgleichung, die Sie durch Entfaltung lösen können.

Aufgabe 11 (Sondierungstechniken)

Folgende Schlüssel 10, 22, 31, 4, 15, 28, 17, 88 und 59 sollen in eine Hashtabelle der Größe m = 11 mit offener Adressierung eingefügt werden. Benutzen Sie als primäre Hashfunktion **h(k) = k mod m**.

Wie lauten die Ergebnisse für

(a) Linear Probing

(b) Quadratic Probing mit $c_1 = 1$, $c_2 = 3$

(c) Double Hashing mit **$h_2(k) = 1 + (k \bmod (m-1))$**

Aufgabe 12 (Erwartungswerte)

Vergleichen Sie für die Hash-Techniken Linear Probing, Quadratic Probing und uniformes Hashing die Erwartungswerte für die Anzahl der Schlüsselvergleiche bei **erfolgreicher** und **nicht-erfolgreicher Suche**. Berechnen Sie dazu die Erwartungswerte für die Füllungsgrade a = 50 %, a = 90 %, a = 95 % und a = 100 %. Interpretieren Sie die Ergebnisse!

Aufgabe 13 (Universelles Hashing)

Implementieren Sie die Methode des universellen Hashing.

a) Entwerfen und implementieren Sie ein Programm, mit dem Zufallsschlüssel erzeugt werden können. Die Verteilung der Schlüssel muss beeinflussbar sein, zum Beispiel 10% der Schlüssel liegen im Intervall [2001 .. 4000], 40% liegen in [4001 .. 4500] usw. Stellen Sie die Verteilung der Schlüssel grafisch dar. Außerdem sollen Schlüssel erzeugt werden können, die durch eine vorgegebene Zahl teilbar sind.

Entwerfen Sie das Programm so, dass es problemlos in die Experimentierumgebung für das universelle Hashing integriert werden kann.

b) Entwickeln Sie eine **Experimentierumgebung**, mit der die folgenden Parameter des universellen Hashing modifiziert werden können:

p : Dekomposition eines Schlüssels in p+1 Bytes
m : Größe der Hashtabelle T = {0, ..., m-1}
n : Anzahl der Schlüssel im Universum U. n = |U|

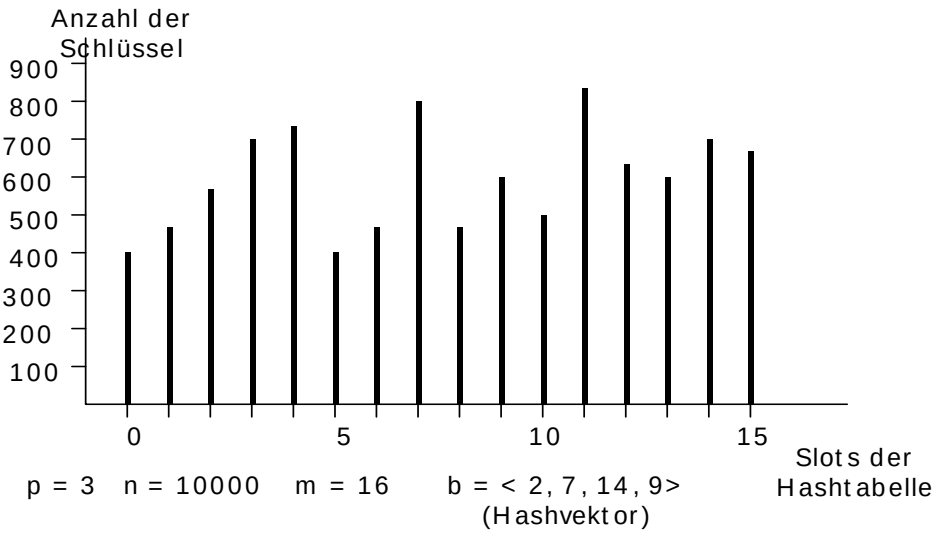
Es soll möglich sein, den Hashvektor b = $<b_0, ..., b_p>$ interaktiv einzugeben und auch per Zufallsgenerator zu erzeugen.

c) Wenden Sie universelles Hashing auf Zufallsschlüssel an. Führen Sie **Experimente** durch, indem Sie die obigen Parameter variieren. Führen Sie für jedes Experiment ein Protokoll, damit es reproduzierbar ist.

Hinweise für Experimente:

- **p** vergrößern und verkleinern. Worauf hat dies Einfluss?
- **m** vergrößern und verkleinern.
- Untersuchen Sie die Fälle **n ≤ m** und **n** >> **m**.
- **m** als Primzahl und als 2-er Potenz
- **Variation der Schlüssel**:
  - durch 2 teilbar
  - durch 3 teilbar
  - durch 5 teilbar
  - ungerade Schlüssel
  - gleich verteilt
  - mit bestimmten Häufigkeiten in verschiedenen Intervallen (siehe a))

Stellen Sie die **Ergebnisse** der Experimente in geeigneter Form grafisch dar. Zum Beispiel lässt sich die Verteilung der Schlüssel auf die Slots einfach **visualisieren**:



- **Planen** Sie Ihre Experimente, d.h. überlegen Sie sich genau, was Sie in einem Experiment testen bzw. herausfinden wollen.
- Dokumentieren Sie Ihre **Vorgehensweise** und **Testergebnisse**.
- Versuchen Sie **Erklärungen** für beobachtete Testergebnisse zu finden.
- Überlegen Sie sich eine geeignete **Darstellungsform** für die Testergebnisse.
- Erstellen Sie exakte **schriftliche Unterlagen** zu allen Punkten (Testprotokoll).

Aufgabe 14 (Simulation)

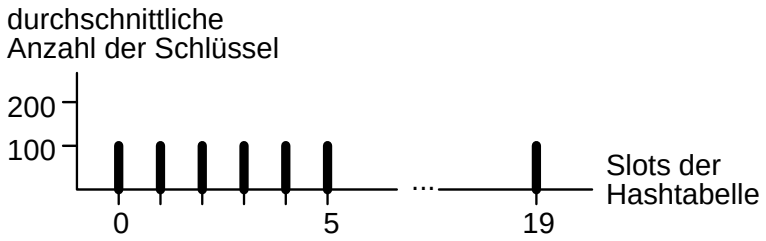
Führen Sie eine Simulation für eine universelle Klasse von Hashfunktionen durch.

- Wählen Sie zunächst sinnvolle Werte für die Parameter p, n und m.
  Wie groß ist für die gewählten Parameterwerte die universelle Menge H von Hashfunktionen?
- Legen Sie interaktiv fest, wie viele Hashvektoren zufällig erzeugt werden sollen. Die Zahl sollte wesentlich größer sein als |H|.
- Wenden Sie alle durch die Hashvektoren bestimmten Hashfunktionen wiederholt auf eine feste Schlüsselmenge an.
- Führen Sie die Simulation des letzten Punktes für $n \leq m$ und für $n >> m$ durch. Welche Ergebnisse erwarten Sie nach Theorem 1 und Theorem 2 der Vorlesung? Stellen Ihre Ergebnisse grafisch dar, und vergleichen Sie diese mit den theoretischen Ergebnissen (Erklärung).

Lösungshinweis zu Aufgabe 14

Die Anzahl der Hashfunktionen in H ist m^{p+1}. Sie Skript Seite 23.

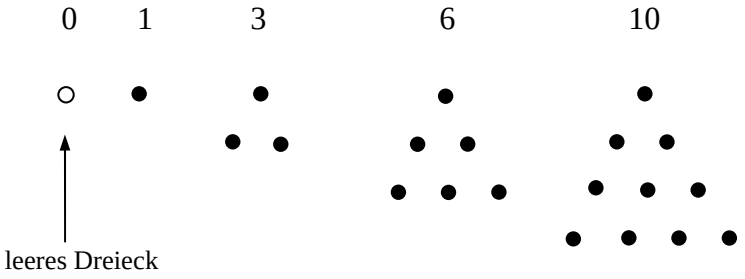
Grafische Darstellung für die durchschnittliche Anzahl von Schlüsseln in den Slots einer Hashtabelle. Die Grafik zeigt die optimale Verteilung, wenn die Hashfunktionen aus der Menge H zufällig gewählt werden.



Größter Index der Dekomposition eines Schlüssels: p = 3
Größe der Hashtabelle: m = 20
Anzahl der Schlüssel: n = 2000
Anzahl der zufällig gewählten Hashfunktionen: 5000

Aufgabe 15 (Dreieckszahlen-Hashing)

Eine Variante des Quadratic Probing ist das so genannte Dreieckszahlen-Hashing. Bei dieser Methode wird die Sondierungsfolge durch die Zahlenfolge 0, 1, 3, 6, 10, ... gebildet, wobei die Sondierungsposition 0 die erste Hashadresse h(k) der Sondierungsfolge ist. Diese Zahlen werden Dreieckszahlen genannt, weil sie die Anzahl der Punkte in einem Dreieck wiedergeben:



Alle Schlüssel mit derselben Start-Hashadresse unterliegen dem Secondary Clustering. In der Vorlesung haben wir folgende Performance-Ergebnisse für Secondary Clustering wie folgt angegeben:

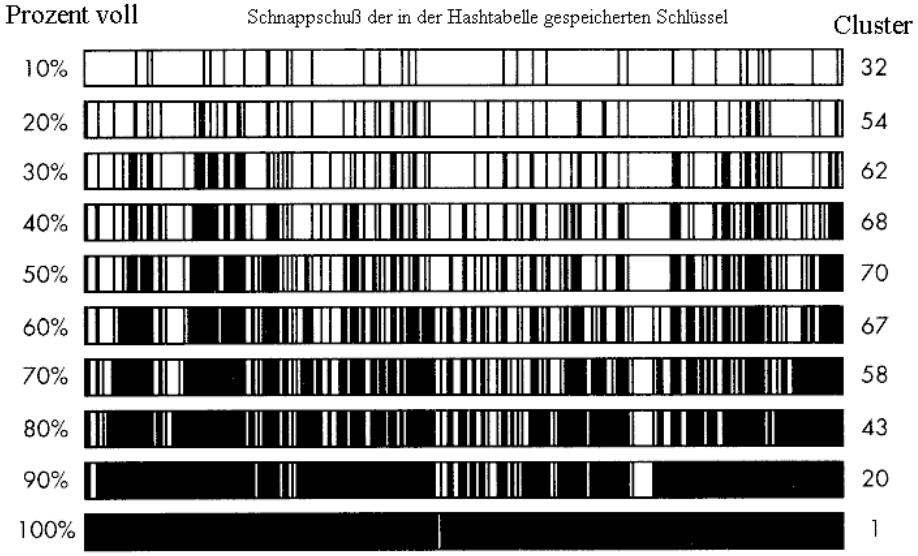
Erfolgreiche Suche: $C_n \approx 1 + \ln(\frac{1}{1-a}) - \frac{a}{2}$

Erfolglose Suche: $C_n' \approx \frac{1}{1-a} - a + \ln(\frac{1}{1-a})$

a) Implementieren Sie den folgenden Algorithmus für das Dreieckszahlen-Hashing und messen Sie die Performance für Hashtabellen der Größe m, wobei m eine 2er-Potenz ist, z.B. m = 1024. Vergleichen Sie Ihre gemessene Performance mit den theoretischen Werten für C_n und C_n'. Führen Sie dazu eine geeignete Anzahl von Experimenten durch.

Programmieren Sie eine grafische Darstellung, so dass die beim Einfügen entstehenden Cluster, ähnlich wie im folgenden Bild für Linear Probing gezeigt, am Bildschirm verfolgt werden können.

b) Was muss bei der Konstruktion einer Sondierungsfolge beachtet werden?



```
    procedure HashInsert(K:KeyType);
    |
    | var
    |     i, j           : Integer;
 5  |     ProbeKey       : KeyType;
    |
    begin
    |
    |     {initializations}
10  |         i := h(K);                                          {first hash}
    |         j := 0;                  {initialize counter for triangle number hashing}
    |         ProbeKey := T[i].Key;
    |
    |     {find first empty slot}
15  |         while ProbeKey <> EmptyKey do begin
    |         |     j := j + 1;          {increment triangle number hashing counter}
    |         |     i := i - j;                        {compute next probe location}
    |         |     if i < 0 then i := i + M;                {wrap around if needed}
    |         |     ProbeKey := T[i].Key
20  |         end{while};
    |
    |     {insert new key K in table T}
    |         T[i].Key := K
    |
    end{HashInsert};

{-------------------------------------------------------------------------------------}


{-------------------------------------------------------------------------------------}

    function HashSearch(K:KeyType):Integer;
    |
    | var
    |     i, j           : Integer;
 5  |     ProbeKey       : KeyType;
    |
    begin
    |
    |     {initializations}
10  |         i := h(K);                                          {first hash}
    |         j := 0;                  {initialize counter for triangle number hashing}
    |         ProbeKey := T[i].Key;
    |


    |     {find either an entry with key, K, or the first empty entry}
15  |         while (K <> ProbeKey) and (ProbeKey <> EmptyKey) do begin
    |         |     j := j + 1;          {increment triangle number hashing counter}
    |         |     i := i - j;                {decrement probe location by amount j}
    |         |     if i < 0 then i := i + M;                {wrap around if needed}
    |         |     ProbeKey := T[i].Key
20  |         end{while};
    |
    |     {return the position of key K in table T, or return -1 if K not in T}
    |         if ProbeKey = EmptyKey then
    |         |     HashSearch := -1              {return -1 to signify K was not found}
25  |         else begin
    |         |     HashSearch := i               {return location, i, of key K in table T}
    |         end{if}
    |
    end{HashSearch};

{-------------------------------------------------------------------------------------}
```