Aufgabe 5: von Mises' Geburtstagsparadoxon

Aufgabe 6: Sondierungstechniken (Kap. 4.5.4)

Aufgabe 7: Berechnung von Erwartungswerten (Kap. 4.5.4)

Aufgabe 8: Dreieckszahlen Hashing

Aufgabe 5 (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 6 (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 $c_1 = 1$, $c_2 = 3$

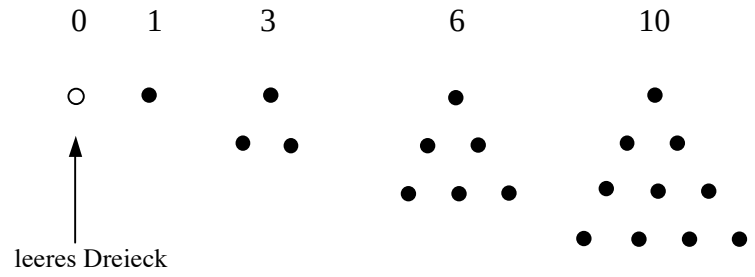
(c) Double Hashing mit **$h_2(k) = 1 + (k \bmod (m-1))$**

Aufgabe 7 (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 = 20%, a = 50 %, a = 70%, a = 90 %, a = 95 % und a = 100 %. Diskutieren Sie die Ergebnisse in der Dokumentation! (Hinweis: Berechnung und Darstellung der Tabellen mit einem kleinen Programm oder einem Excel-Sheet)

Aufgabe 8 (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 (Kap 4.5.4) haben wir folgende Performance-Ergebnisse für Secondary Clustering wie folgt angegeben:

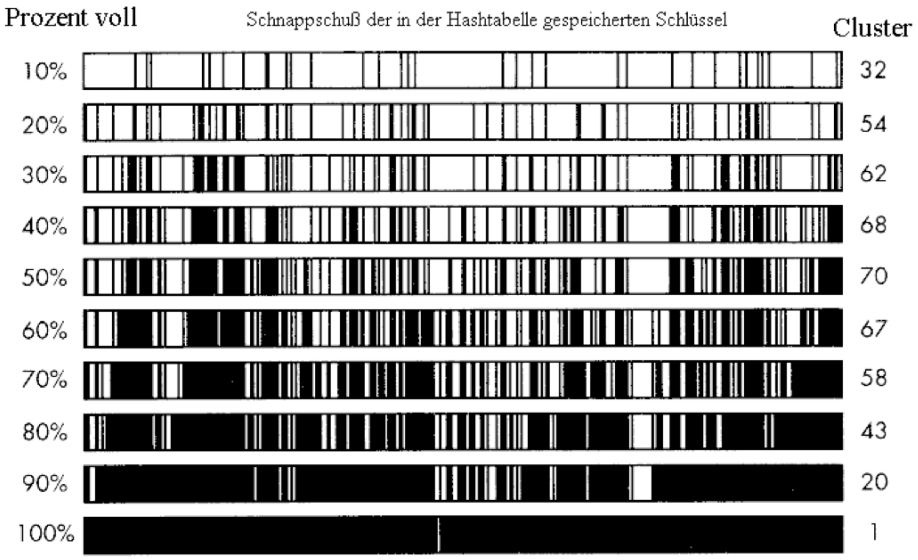
Erfolgreiche Suche: $$C_n \approx 1 + \ln\left(\frac{1}{1-a}\right) - \frac{a}{2}$$

Erfolglose Suche: $$C_{n'} \approx \frac{1}{1-a} - a + \ln\left(\frac{1}{1-a}\right)$$

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};

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