Datenstrukturen und Algorithmen

Vorlesung 10

Überblick

- Vorige Woche:
  - Hashtabellen:
    - Intro
    - Unabhängige Verkettung
    - Verzahnte Verkettung

- Heute betrachten wir:
  - Hashtabellen:
    - Offene Adressierung
    - Perfektes Hashing
    - Cuckoo Hashing
    - Verkettete Hashtabellen

Hashtabellen - Kollisionen

- Es gibt unterschiedliche Strategien für Kollisionsauflösung:
  - **Unabhängige Verkettung** (separate chaining)
  - **Coalesced/Verzahnte Verkettung** (coalesced chaining)
  - **Offene Adressierung**
  - **und andere...**

Kollisionsauflösung: **offene Adressierung**

Offene Adressierung

- Im Falle der offenen Adressierung wird jedes Element direkt in die Hashtabelle gespeichert und es gibt keine Pointer oder *next* Felder zwischen den Elementen

- Um das Einfügen mithilfe offener Adressierung durchzuführen, müssen wir die Hashtabelle sukzessiv überprüfen oder ***sondieren***, bis wir einen leeren Slot finden, an dem wir den Schlüssel speichern können

Offene Adressierung

- Um die Slots zu bestimmen, die sondiert werden, erweitern wir die Hashfunktion um einen zusätzlichen Eingabeparameter, über den eine Sondierungszahl (mit 0 beginnend) übergeben wird
- Die Hashfunktion wird damit zu:

  $h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$

- Für ein Element wird der Reihe nach folgende Sequenz sondiert

  <h(k,0), h(k,1), ..., h(k,m-1)> - ***Sondierungssequenz***

- Für jeden Schlüssel *k* fordert man, dass die Sondierungssequenz eine **Permutation** von (0, 1, ..., m – 1) ist, sodass letztendlich jede Position der Hashtabelle als Slot für einen neuen Schlüssel berücksichtigt wird

Offene Adressierung – **lineares Sondieren**

- Eine Methode die Hashfunktion zu definieren ist **lineares Sondieren:**

$$h(k,i) = (h'(k) + i)\ mod\ m, \forall i = 0, \dots, m-1$$

- wobei h’(k) eine einfache Hashfunktion ist (z.B. h’(k) = k mod m)

- die Sondierungssequenz für lineares Sondieren ist:

<h’(k), h’(k)+1,h’(k)+2, ..., m-1, 0, 1, ..., h’(k)-1>

Offene Adressierung – lineares Sondieren Beispiel

- Sei eine Hashtabelle mit der Größe m = 16 mit offener Adressierung mit linearem Sondieren als Kollisionsbehandlung (h‘(k) ist eine Hashfunktion mit der Divisionsmethode)
- Wir wollen folgende Elemente in die Tabelle einfügen: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Schlüssel	76	12	109	43	22	18	55	81	91	27	13	16	39
h‘(Schlüssel)	12	12	13	11	6	2	7	1	11	11	13	0	7

Offene Adressierung – lineares Sondieren Beispiel

Schlüssel	76	12	109	43	22	18	55	81	91	27	13	16	39
h‘(Schlüssel)	12	12	13	11	6	2	7	1	11	11	13	0	7

T	27	81	18	13	16		22	55	39			43	76	12	109	91
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Offene Adressierung – lineares Sondieren

- Nachteil des linearen Sondierens:
  - Es gibt nur *m* unterschiedliche Sondierungssequenzen (nachdem man den Anfangspunkt kennt, kennt man schon die ganze Sondierungsequenz)
  - Das Problem der *primären Cluster* – es bilden sich lange Folgen besetzter Slots, wodurch sich die Suchzeit erhöht

- Vorteile des linearen Sondierens:
  - Sondierungssequenz ist garantiert immer eine Permutation
  - Kann von Caching profitieren

Offene Adressierung – lineares Sondieren - *primäres Clustern*

- Warum ist *primären Clustern* ein Problem?

Für eine Hashtabelle mit m Positionen, n Elemente und $\alpha = 0.5$:

- Im besten Fall: jede zweite Position ist leer
- Im schlechtesten Fall: alle n Elemente sind nacheinander gespeichert
- Welche ist die durchschnittliche Anzahl der überprüfte Positionen, um ein neues Element einzufügen:
  - im besten Fall?
  - im schlechtesten Fall?

Offene Adressierung – Quadratisches Sondieren

- Verwende eine Hashfunktion der Form:

$$\boldsymbol{h(k,i) = \left(h'(k) + c_1 * i + c_2 * i^2\right) mod\ m, \forall i = 0, \ldots, m-1}$$

- wobei h’(k) eine einfache Hashfunktion ist (z.B. h’(k) = k mod m) und c_1 und c_2 Konstante sind, $c_2 \neq 0$
- Wenn man ein vereinfachtes Beispiel betrachtet mit $c_1 = 0$ und $c_2 = 1$, dann ist die Sondierungssequenz:

<k, k+1, k+4, k+9, k+16, …>

Offene Adressierung – Quadratisches Sondieren

- Wie kann man aber **die Werte für m, c_1 und c_2 auswählen**, sodass die Sondierungssequenz eine Permutation ist?
- Falls *m eine Primzahl* ist, dann ist die erste Hälfte der Sondierungssequenz eindeutig, also, sobald die Hälfte der Tabelle voll ist, kann man nicht sicherstellen, dass eine leere Position gefunden wird
  - z.B. m = 17, c_1 = 3 , c_2 = 1 ($h(k,i) = (k\ mod\ 17 + 3 * i + 1 * i^2)\ mod\ 17$) und k = 13, Sondierungssequenz:

    <13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11>
  - z.B. m = 11, c_1 = 1 , c_2 = 1 und k = 27, Sondierungssequenz:

    <5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5>

Offene Adressierung – Quadratisches Sondieren

- Falls ***m eine Potenz von 2*** ist und $c_1 = c_2 = 0.5$, dann ist die Sondierungssequenz immer eine Permutation.
- Z.B. für m = 8 und k = 3:
  - $h(3,0) = (3 \% 8 + 0.5 * 0 + 0.5 * 0^2) \% 8 = 3$
  - $h(3,1) = (3 \% 8 + 0.5 * 1 + 0.5 * 1^2) \% 8 = 4$
  - $h(3,2) = (3 \% 8 + 0.5 * 2 + 0.5 * 2^2) \% 8 = 6$
  - $h(3,3) = (3 \% 8 + 0.5 * 3 + 0.5 * 3^2) \% 8 = 1$
  - $h(3,4) = (3 \% 8 + 0.5 * 4 + 0.5 * 4^2) \% 8 = 5$
  - $h(3,5) = (3 \% 8 + 0.5 * 5 + 0.5 * 5^2) \% 8 = 2$
  - $h(3,6) = (3 \% 8 + 0.5 * 6 + 0.5 * 6^2) \% 8 = 0$
  - $h(3,7) = (3 \% 8 + 0.5 * 7 + 0.5 * 7^2) \% 8 = 7$

Offene Adressierung – Quadratisches Sondieren

- Falls ***m eine Primzahl der Form 4 * k + 3*** ist, $c_1 = 0$ *und* $c_2 = (-1)^i$ (dann ist die Sondierungssequenz +0, -1, +4, -9, usw.) dann ist die Sondierungssequenz immer eine Permutation.
- Z.B. für m = 7 und k = 3:
  - $h(3,0) = (3 \,\%\, 7 + 0^2) \,\%\, 7 = 3$
  - $h(3,1) = (3 \,\%\, 7 - 1^2) \,\%\, 7 = 2$
  - $h(3,2) = (3 \,\%\, 7 + 2^2) \,\%\, 7 = 0$
  - $h(3,3) = (3 \,\%\, 7 - 3^2) \,\%\, 7 = 1$
  - $h(3,4) = (3 \,\%\, 7 + 4^2) \,\%\, 7 = 5$
  - $h(3,5) = (3 \,\%\, 7 - 5^2) \,\%\, 7 = 6$
  - $h(3,6) = (3 \,\%\, 7 + 6^2) \,\%\, 7 = 4$

Offene Adressierung – Quadratisches Sondieren Beispiel

- Sei eine Hashtabelle mit der Größe $m = 16$ mit offener Adressierung mit quadratischem Sondieren als Kollisionsbehandlung (h‘(k) ist eine Hashfunktion mit der Divisionsmethode), $c_1 = c_2 = 0.5$
- Wir wollen folgende Elemente in die Tabelle einfügen: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Schlüssel	76	12	109	43	22	18	55	81	91	27	13	16	39
h‘(Schlüssel)	12	12	13	11	6	2	7	1	11	11	13	0	7

Offene Adressierung – Quadratisches Sondieren

Schlüssel	76	12	109	43	22	18	55	81	91	27	13	16	39
h‘(Schlüssel)	12	12	13	11	6	2	7	1	11	11	13	0	7

T	13	81	18	16		91	22	55	39		27	43	76	12	109	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- $h(k,i) = (k\ mod\ 16 + 0.5 * i + 0.5 * i^2)\ mod\ 16$
- Nachteile des quadratischen Sondieren:
  - Die Leistung hängt von den Werten m, c_1 und c_2 ab
  - Das Problem des *sekundären Clustern* – wenn die Anfangsslots die gleichen sind, dann sind zwei Sondierungssequenzen gleich, denn aus $h(k_1,0) = h(k_2,0)$ folgt $h(k_1,i) = h(k_2,i)$
  - Wie beim linearen Sondieren legt die erste sondierte Position die gesamte Sondierungssequenz fest und so werden nur *m* verschiedene Sondierungssequenzen verwendet

Offene Adressierung – Doppeltes Hashing

- Verwendet eine Hashfunktion der Form:

$$\boldsymbol{h(k,i) = (h'(k) + i * h''(k))\ mod\ m, \forall i = 0, \ldots, m-1}$$

- wobei h’(k) und h’’(k) einfache Hashfunktionen sind, aber h’’(k) sollte nie den Wert 0 haben

- Für einen Schlüssel, k, wird die erste sondierte Position h’(k) sein; alle anderen Positionen werden mithilfe der zweiten Hashfunktion h’’(k) berechnet

Offene Adressierung – Doppeltes Hashing

- Ähnlich wie bei dem quadratischen Sondieren wird nicht jede Kombination vom *m* und h’’(k) eine Permutation als Sondierungssequenz generieren
- Um eine Permutation zu generieren, müssen ***m* und die Werte h’’(k) teilerfremd** (relatively primes) sein. Es gibt zwei Möglichkeiten das zu erreichen:
  - ***m* als Potenz von 2** auszuwählen und die Funktion **h’’** so zu konstruieren, dass sie **immer eine ungerade Zahl** erzeugt
  - ***m* als Primzahl** auszuwählen und die Funktion **h’’** so zu konstruieren, dass sie **immer eine positive ganze Zahl kleiner als m** ergibt

Offene Adressierung – Doppeltes Hashing

- *m* als Primzahl zu wählen und die Funktion h'' so zu konstruieren, dass sie immer eine positive ganze Zahl kleiner als m ergibt
- Zum Beispiel:

  h'(k) = k % m

  h''(k) = 1 + (k % (m-1))
- Für m = 11 und k = 36:

  h'(36) = 3

  h''(36) = 7
- Die Sondierungssequenz ist: <3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7>

  (Hashfunktion $h(k,i) = (h'(k) + i * h''(k)) \, mod \, m$)

Offene Adressierung – Doppeltes Hashing Beispiel

- Sei eine Hashtabelle mit der Größe $m = 17$ mit offener Adressierung mit doppeltem Hashing als Kollisionsbehandlung mit $h'(k) = k \% m$ und $h''(k) = 1 + (k \% (m-1))$
- Wir wollen folgende Elemente in die Tabelle einfügen: 75, 12, 109, 43, 22, 18, 55, 81, 92, 27, 13, 16, 39

Schlüssel	75	12	109	43	22	18	55	81	92	27	13	16	39
h‘	7	12	7	9	5	1	4	13	7	10	13	16	5
h’’	12	13	14	12	7	3	8	2	13	12	14	1	8

Offene Adressierung – Doppeltes Hashing

Schlüssel	75	12	109	43	22	18	55	81	92	27	13	16	39
h‘	7	12	7	9	5	1	4	13	7	10	13	16	5
h’’	12	13	14	12	7	3	8	2	13	12	14	1	8

T	16	18		55	109	22		75		43	27	39	12	81		13	92
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- Hauptvorteil des doppelten Hashings: auch wenn $h(k_1,0) = h(k_2,0)$ werden die Sondierungssequenzen nicht gleich sein, falls $k_1 \neq k_2$
- Zum Beispiel:
  - Schlüssel 75: <7, 2, 14, 9, 4, 16, 11, 6, 1, 13, 8, 3, 15, 10, 5, 0, 12>
  - Schlüssel 109: <7, 4, 1, 15, 12, 9, 6, 3, 0, 14, 11, 8, 5, 2, 16, 13, 10>
- Da jedes mögliche Paar (h’(k), h’’(k)) eine andere Sondierungssequenzen liefert, werden bei der doppelten Hashing ungefähr m^2 unterschiedliche Permutationen verwendet

Offene Adressierung – Operationen

- Wir besprechen die Implementierung der Standardoperationen für Kollisionsbehandlung mit offener Adressierung

- Wir benutzen die Notation *h(k, i)* für eine Hashfunktion, ohne anzugeben ob diese lineares Sondieren, quadratisches Sondieren oder doppelte Hashing benutzt (die Implementierung der Operationen unterscheiden sich nicht, nur die Implementierung der Hashfunktion *h)*

Offene Adressierung – Repräsentierung

- Welche Felder braucht man um eine Hashtabelle mit mit offener Adressierung als Methode für Kollisionsbehandlung zu speichern?
- Für die Vereinfachung betrachten wir nur die Schlüssel

HashTable:

T: TKey[]

m: Integer

h: TFunction

Offene Adressierung – insert

```
subalgorithm insert (ht, e) is:
//pre: ht ist ein HashTable, e ist ein TKey
//post: e wurde in ht eingefügt
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 execute
        //-1 heißt leere Position
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i = ht.m then
        @resize and rehash
    else
        ht.T[pos] ← e
    end-if
end-subalgorithm
```

Offene Adressierung – search

```
function search (ht, e) is:
//pre: ht ist ein HashTable, e ist ein TKey
//post: überprüft ob e in ht enthalten ist
    i ← 0
    pos ← ht.h(e, i)
    while i < ht.m and ht.T[pos] ≠ -1 and ht.T.[pos] ≠ e execute
        //-1 means empty space
        i ← i + 1
        pos ← ht.h(e, i)
    end-while
    if i < ht.m and ht.T[pos] = e execute
        search ← True
    else
        search ← False
    end-if
end-function
```

Offene Adressierung – remove

- Wie kann man ein Element aus einer Hashtabelle löschen?
- Aus einer Hashtabelle mit offener Adressierung ist es nicht einfach ein Element zu löschen:
  - Man kann nicht einfach die Position als leer markieren – die *search* Funktion wird dann die anderen Elemente nicht mehr finden
  - Man kann die Elemente nicht einfach verschieben – die *search* Funktion wird dann die anderen Elemente nicht mehr finden
- Die Löschoperation wird meisten implementiert indem man die leere Position mit einem speziellen Wert, DELETED, markiert
- Wie ändert dieser spezielle Wert die *insert* und *search* Operationen?

Offene Adressierung – Effizienz

- In einer Hashtabelle mit offener Adressierung mit Belegungsfaktor $\alpha = n/m$ ($\alpha < 1$) ist die *durchschnittliche* Anzahl von Sondierungen:
  - Für *insert* und *erfolglose search:*

$$\frac{1}{1-\alpha}$$

  - Für *erfolgreiche search:*

$$\frac{1}{\alpha} * ln \frac{1}{1-\alpha}$$

- Falls α eine Konstante ist, dann ist die Komplexität $\Theta(1)$
- Im schlimmsten Fall ist die Komplexität $\Theta(n)$

Offene Adressierung – Effizienz

- In der Praxis funktioniert doppeltes Hashing gut bis $\alpha \approx 0.8$
- Bei linearem Sondieren ist es sehr wahrscheinlich dass manman schon ab $\alpha \approx 0.5$ lange Sequenzen durchsuchen muss
- Quadratisches Sondieren liegt meist zwischen den Beiden

- Im Allgemeinen: resize & rehash bei $\alpha = 0.75$

Kollisionsauflösung:
Perfektes Hashing

Perfektes Hashing

- Vermute, dass man alle Schlüssel von Anfang an kennt und, dass wir unabhängige Verkettung als Kollisionsbehandlung benutzen ⇒
  - um so mehrere Listen man bildet, um so kürzer sind die Listen (man reduziert die Anzahl der Kollisionen) ⇒
  - wenn man eine große Anzahl von Listen hat, dann wird jede Liste ein einziges Element enthalten (keine Kollisionen)
- Wie groß sollte die Hashtabelle sein, damit man sicherstellt, dass es keine Kollisionen gibt?
- Falls $M = N^2$, dann kann man zeigen, dass die Tabelle mit einer Wahrscheinlichkeit von 1/2 keine Kollisionen enthält
- Man fängt an eine Hashtabelle aufzubauen. Falls man eine Kollision entdeckt, dann wählt man eine neue Hashfunktion und man fängt neu an (man braucht höchstens 2 Versuche)

Perfektes Hashing

- Eine Tabelle von Größe N^2 ist nicht praktisch
- Lösung anstatt:
  - Benutze eine Hashtabelle von Größe N (primäre Hashtabelle)
  - Anstatt verkettete Listen für Kollisionen zu benutzen ist jedes Element der Hashtabelle eine Hashtabelle selber (sekundäre Hashtabelle)
  - Die sekundäre Hashtabelle wird die Größe n_j^2 haben, wobei n_j die Anzahl der Elemente aus dieser Hashtabelle ist (um garantieren zu können, dass es keine Kollisionen auf der zweiten Ebene gibt)
  - Jede sekundäre Hashtabelle wird mit einer unterschiedlichen Hashfunktion erstellt und wird neuerstellt, falls es eine Kollision gibt
- Das heißt **perfektes Hashing**
- Man kann zeigen, dass der gesamte Platz, der für die sekundäre Hashtabellen gebraucht wird, höchstens *2N* ist (falls man mehr Platz braucht, dann wählt man eine andere Hashfunktion)

Perfektes Hashing

- Perfektes Hashing benötigt mehrere Hashfunktionen, darum benutzen wir universelles Hashing
- Sei *p* eine Primzahl, größer als der größte Schlüsselwert
- Die universelle Hashfunktionen Familie *H* kann folgendermaßen definiert werden:

  $H = \{ H_{a,b}(x) = ((a * x + b) \% p) \% m \}$,

  wobei $1 \leq a \leq p-1$, $0 \leq b \leq p-1$
- *a* und *b* werden zufällig ausgewählt bei der Initialisierung der Hashfunktion

Perfektes Hashing – Beispiel

- Wir wollen folgende Elemente in die Tabelle einfügen: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39
- Da es $N = 13$ Elemente gibt, wählen wir $m = 13$
- p muss eine Primzahl größer als der maximale Schlüsselwert sein, also $p = 151$
- Die Hashfunktion ist:

$$H_{a,b}(x) = ((a * x + b) \% p) \% m$$

wobei $a = 3$ und $b = 2$ (zufällig ausgewählt)

Schlüssel	76	12	109	43	22	18	55	81	91	27	13	16	39
H(Schlüssel)	1	12	1	1	3	4	3	3	7	5	2	11	2

Perfektes Hashing – Beispiel

Schlüssel	76	12	109	43	22	18	55	81	91	27	13	16	39
H(Schlüssel)	1	12	1	1	3	4	3	3	7	5	2	11	2

- Die Elemente werden auf folgende Positionen gehasht:
  - Position 1– 76, 109, 43
  - Position 2 – 13, 39
  - Position 3 – 22, 55, 81
  - Position 4 – 18
  - Position 5 – 27
  - Position 7 – 91
  - Position 11 – 16
  - Position 12 – 12
- Summe der Größen der sekundären Hashtabellen: 9 + 4 + 9 + 1 + 1 + 1 + 1 + 1 = 27

Perfektes Hashing – Beispiel

- Für die Positionen wo es keine Kollisionen gibt, werden wir eine sekundäre Hashtabelle mit einem Element haben und mit der Hashfunktion $h(x) = 0$
- Für die Positionen mit zwei Elementen werden wir eine sekundäre Hashtabelle mit 4 Positionen haben und unterschiedliche Hashfunktionen, aus derselben Familie, aber mit unterschiedlichen Werten für a und b und mit $m = 4$
- Zum Beispiel für die Position 2 kann man a = 4 und b= 11 haben, dann kriegt man:

  h(13) = 3

  h(39) = 0

Perfektes Hashing – Beispiel

- Nehmen wir an, dass man die Werte a = 14 und b= 1 für die sekundäre Tabelle von der Position 1 auswählt
- Dann sind die Positionen der Elemente:

  h(76) = ((14 * 76 + 1) % 151) % 9 = 8

  h(109) = ((14 * 109 + 1) % 151) % 9 = 8

  h(43) = ((14 * 43 + 1) % 151) % 9 = 6
- Bei perfektem Hashing darf man keine Kollisionen in der sekundären Tabelle haben, also wählt man eine andere Hashfunktion, d.h. andere zufällige Werte für *a* und *b*. Zum Beispiel für a = 2 und b= 13 kriegt man h(76) = 5, h(109) = 8, h(43) = 0

Perfektes Hashing

- Bei perfektes Hashing muss man höchstens 2 Positionen für ein Element überprüfen (Position in der primären und in der sekundären Tabelle)

- Das heißt im schlimmsten Fall ist die Komplexität $\Theta(1)$

- Aber um perfektes Hashing zu benutzen braucht man statische Schlüssel: nachdem die Tabelle erstellt wurde, sollten keine neue Elemente eingefügt werden

Kollisionsauflösung:
Cuckoo Hashing

Cuckoo Hashing

- Bei Cuckoo Hashing gibt es zwei Hashtabellen derselben Größe, jede der beiden Hashtabellen hat mehr als die Hälfte leer und jede hat ihre eigene Hashfunktion
- Um jedes Element einzufügen kann man zwei Positionen berechnen: eine in der ersten Tabelle und eine in der zweiten Tabelle. Bei Cuckoo Hashing wird sichergestellt, dass sich ein Element auf einer dieser beiden Position befindet
- Die Suche ist einfach, da man nur auf zwei Positionen suchen muss
- Das Löschen ist einfach, da man die zwei Positionen betrachten muss und die Position, wo man das Element findet, als leer bezeichnen muss

Cuckoo Hashing

- Wenn man ein Element einfügen will, dann berechnet man seine **Position in der ersten Hashtabelle**. Falls diese Position leer ist, dann fügt man das Element hier ein
- Falls die Position in der ersten Hashtabelle **nicht leer** ist, dann **ersetzt** man das Element, das sich an der Position befindet und man speichert das neue Element an dieser Position
- **Das Element das ersetzt wurde** wird an der Position **in der zweiten Hashtabelle** gespeichert. Falls diese Position besetzt ist, dann ersetzt man dieses Element mit dem neuen Element und das ersetzte Element wird in der ersten Hashtabelle eingefügt
- Man wiederholt diese Schritte bis man ein Element auf einer leeren Position einfügen muss
- Falls man zurück zu einer vorigen Position kommt, dann gibt es einen Zyklus, d.h. man kann das Element nicht einfügen ⇒ resize, rehash

Cuckoo Hashing – Beispiel

- Nehmen wir an, dass es zwei Hashtabelle gibt mit m = 11 Positionen und mit den folgenden Hashfunktionen:
  - h1(k) = k % 11
  - h2(k) = (k div 11) % 11
- Wir wollen folgende Elemente einfügen: 20, 50, 53, 75, 100, 67, 105, 3, 36, 39

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo Hashing – Beispiel

h1(k) = k % 11

h2(k) = (k div 11) % 11

- Füge 20 ein:
  - h1(20) = 9 – leere Position, wird in die erste Tabelle eingefügt
- Füge 50 ein:
  - h1(50) = 6 – leere Position, wird in die erste Tabelle eingefügt
- Wo sollte man 53 einfügen?

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			20	

Position	0	1	2	3	4	5	6	7	8	9	10
T											

Cuckoo Hashing – Beispiel

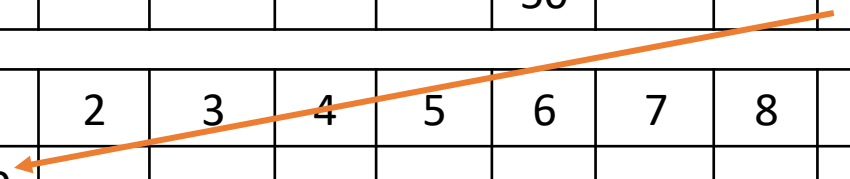
h1(k) = k % 11

h2(k) = (k div 11) % 11

- Füge 53 ein:
  - h1(53) = 9 – besetzte Position
  - 53 wird in die erste Tabelle eingefügt und 20 wird in die zweite Tabelle eingefügt an die Position h2(20) = 1
- Wo sollte man 75 einfügen?

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20									



Cuckoo Hashing – Beispiel

h1(k) = k % 11

h2(k) = (k div 11) % 11

- Füge 75 ein:
  - h1(75) = 9 – besetzte Position
  - 75 wird in die erste Tabelle eingefügt und 53 wird in die zweite Tabelle eingefügt an die Position h2(53) = 4
- Wo sollte man 100 einfügen?

Position	0	1	2	3	4	5	6	7	8	9	10
T							50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53						

Cuckoo Hashing – Beispiel

h1(k) = k % 11

h2(k) = (k div 11) % 11

- Füge 100 ein:
  - h1(100) = 1 – leere Position
- Füge 67 ein:
  - h1(67) = 1 – besetzte Position
  - 67 wird in die erste Tabelle eingefügt und 100 wird in die zweite Tabelle eingefügt an die Position h2(100) = 9
- Wo sollte man 105 einfügen?

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			53					100	

Cuckoo Hashing – Beispiel

h1(k) = k % 11

h2(k) = (k div 11) % 11

- Füge 105 ein:
  - h1(105) = 6 –besetzt
  - 105 wird in die erste Tabelle eingefügt und 50 wird in die zweite Tabelle eingefügt an die Position h2(50) = 4 – besetzt
  - 50 wird in die zweite Tabelle eingefügt und 53 wird in die erste Tabelle eingefügt an die Position h1(53) = 9 – besetzt
  - 53 wird in die erste Tabelle eingefügt und 75 wird in die zweite Tabelle eingefügt an die Position h2(75) = 6

Position	0	1	2	3	4	5	6	7	8	9	10
T		67					105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T		20			50		75			100	

Cuckoo Hashing – Beispiel

h1(k) = k % 11

h2(k) = (k div 11) % 11

- Füge 3 ein:
  - h1(3) = 3 – leere Position
- Füge 36 ein
  - h1(36) = 3 – besetzt
  - 36 wird in die erste Tabelle eingefügt und 3 wird in die zweite Tabelle eingefügt an der Position h2(3) = 0

Position	0	1	2	3	4	5	6	7	8	9	10
T		67		36			105			53	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20			50		75			100	

Cuckoo Hashing – Beispiel

h1(k) = k % 11

h2(k) = (k div 11) % 11

- Füge 39 ein:
  - h1(39) = 6 – besetzt
  - 39 wird in die erste Tabelle eingefügt und 105 wird in die zweite Tabelle eingefügt an die Position h2(105) = 9 – besetzt
  - 105 wird in die zweite Tabelle eingefügt und 100 wird in die erste Tabelle eingefügt an die Position h1(100) = 1 – besetzt
  - 100 wird in die erste Tabelle eingefügt und 67 wird in die zweite Tabelle eingefügt an die Position h2(67) = 6 – besetzt
  - 67 wird in die zweite Tabelle eingefügt und 75 wird in die erste Tabelle eingefügt an die Position h1(75) = 9 – besetzt
  - 75 wird in die erste Tabelle eingefügt und 53 wird in die zweite Tabelle eingefügt an die Position h2(53) = 4 – besetzt
  - 53 wird in die zweite Tabelle eingefügt und 50 wird in die erste Tabelle eingefügt an die Position h1(50) = 6 – besetzt
  - 50 wird in die erste Tabelle eingefügt und 39 wird in die zweite Tabelle eingefügt an die Position h2(39) = 3 – frei, endlich!

Cuckoo Hashing – Beispiel

Position	0	1	2	3	4	5	6	7	8	9	10
T		100		36			50			75	

Position	0	1	2	3	4	5	6	7	8	9	10
T	3	20		39	53		67			105	

Cuckoo Hashing

- Es kann sein, dass man ein Element nicht einfügen kann, falls es eine Endlosschleife gibt. In diesem Fall muss man die Größe der zwei Hashtabellen erhöhen und dann werden zufällig zwei neue Hashfunktionen ausgewählt und alle Elemente werden rehasht (versuche, in die vorherige Hashtabelle eine 6 einzufügen!)

- Obwohl in manchen Fällen die Einfügeoperation viele Elemente verschiebt, kann man zeigen, dass die Wahrscheinlichkeit einer Endlosschleife klein ist, wenn der Belegungsfaktor weniger als 0.5 beträgt, und dass es unwahrscheinlich ist, dass mehr als $O(log_2 n)$ Elemente verschoben werden

Cuckoo Hashing

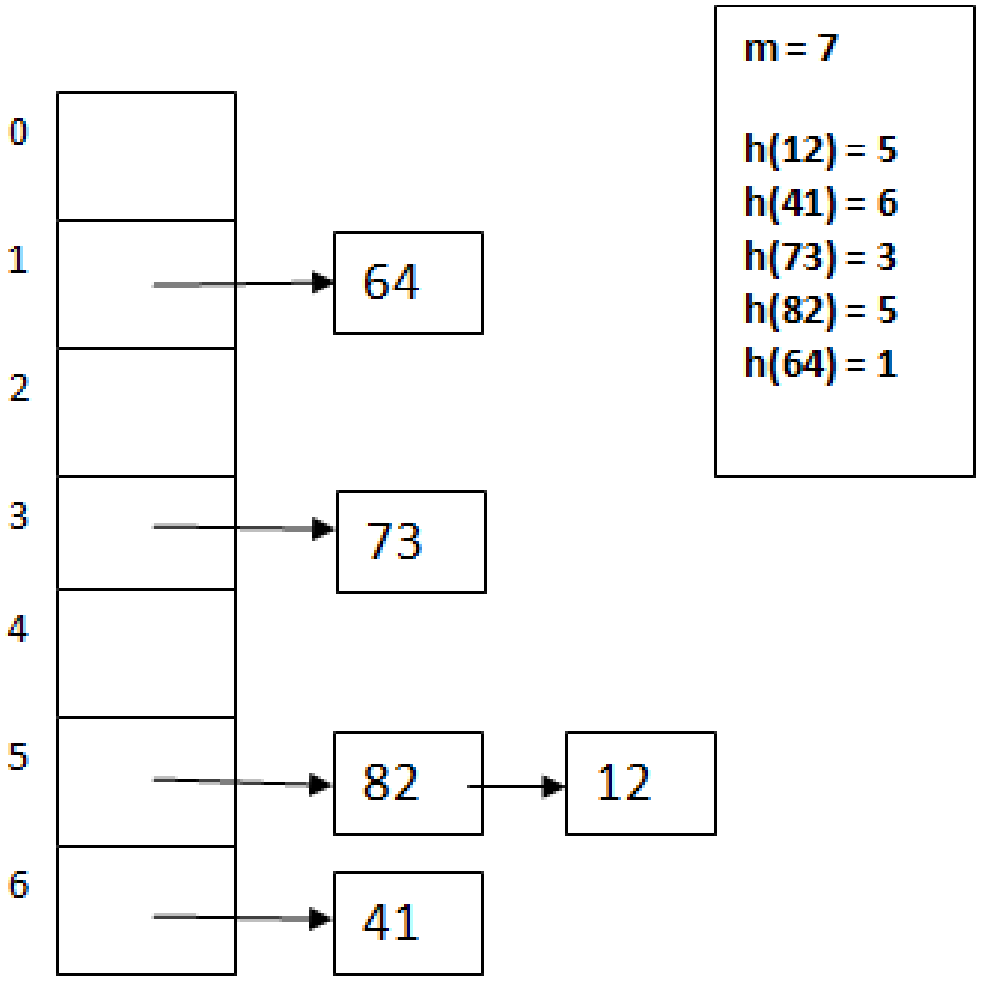
- Wenn man zwei Hashtabellen benutzt und jede Position aus der Hashtabelle höschstens ein Element enthält, dann müssen die Tabellen einen Belegungsfaktor kleiner als 0.5 haben, damit es gut funktioniert
- Wenn man drei Tabellen benutzt, dann können diese den Belegungsfaktor bis 0.91 haben
- Wenn man vier Tabellen benutzt, dann können diese den Belegungsfaktor bis 0.97 haben

Verkettete Hashtabellen

Verkettete Hashtabellen

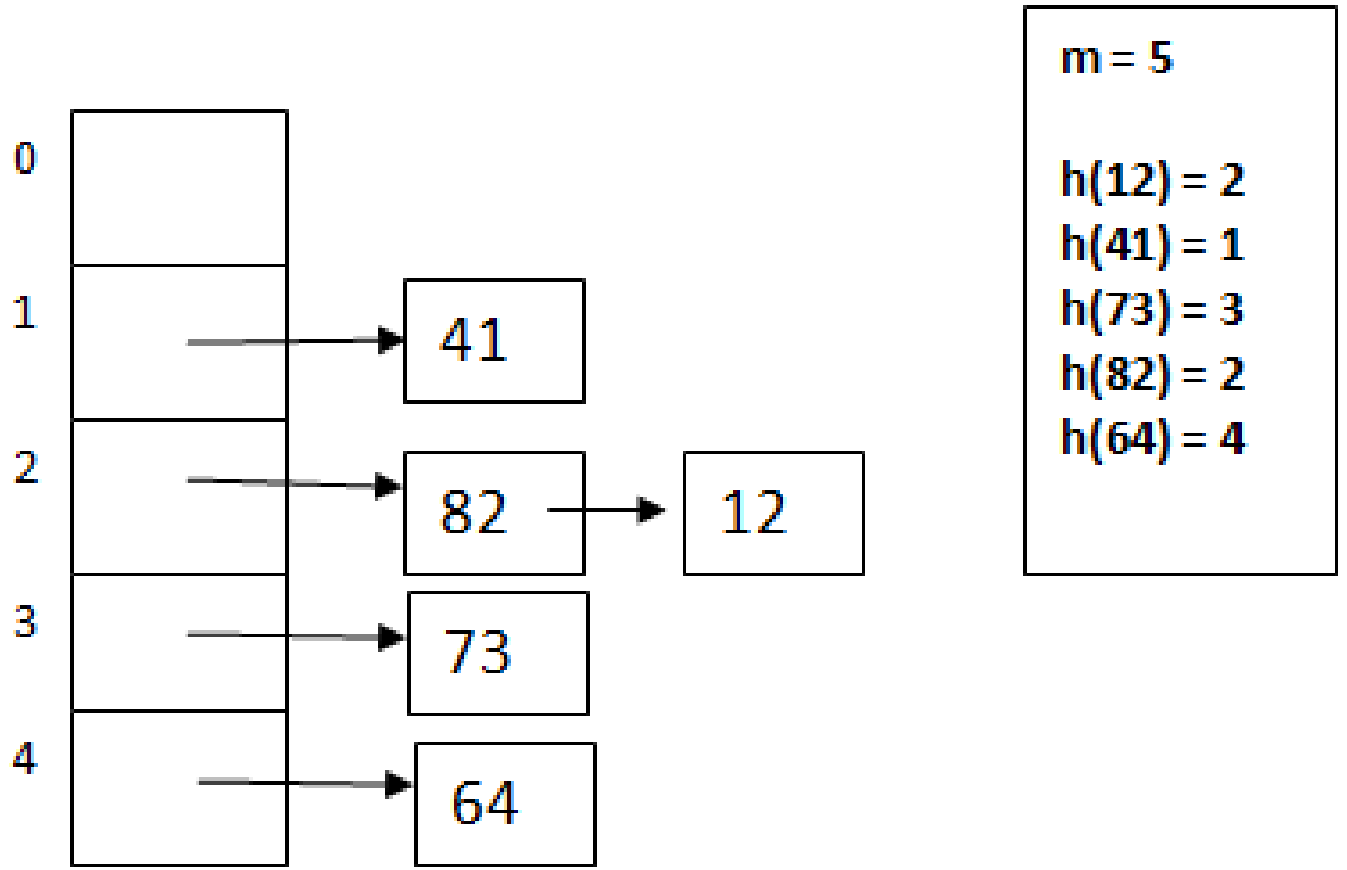
- Vermute, dass man eine Hashtabelle mit unabhängiger Verkettung als Kollisionsauflösungsmethode erstellt
- Wenn man durch die Elemente der Hashtabelle iteriert (mithilfe eines Iterators), dann ist die Reihenfolge nicht bekannt
- Zum Beispiel:
  - Nehmen wir an, dass die Hashtabelle am Anfang leer ist (wir kennen die Implementierung nicht)
  - Wir fügen folgende Elemente der Reihe nach ein: 12, 41, 73, 82, 64
  - In welcher Reihenfolge wird jetzt der Iterator die Elemente durchlaufen?

Beispiel



- Reihenfolge beim Iterieren: 64, 73, 82, 12, 41

Beispiel



- Reihenfolge beim Iterieren: 41, 82, 12, 73, 64

Verkettete Hashtabellen

- Eine **verkettete Hashtabelle** ist eine Datenstruktur, für welche die **Reihenfolge beim Iterieren bekannt** ist: diese kann genau die Reihenfolge sein in welcher die Elemente eingefügt wurden, oder sie kann durch eine Relation bestimmt werden

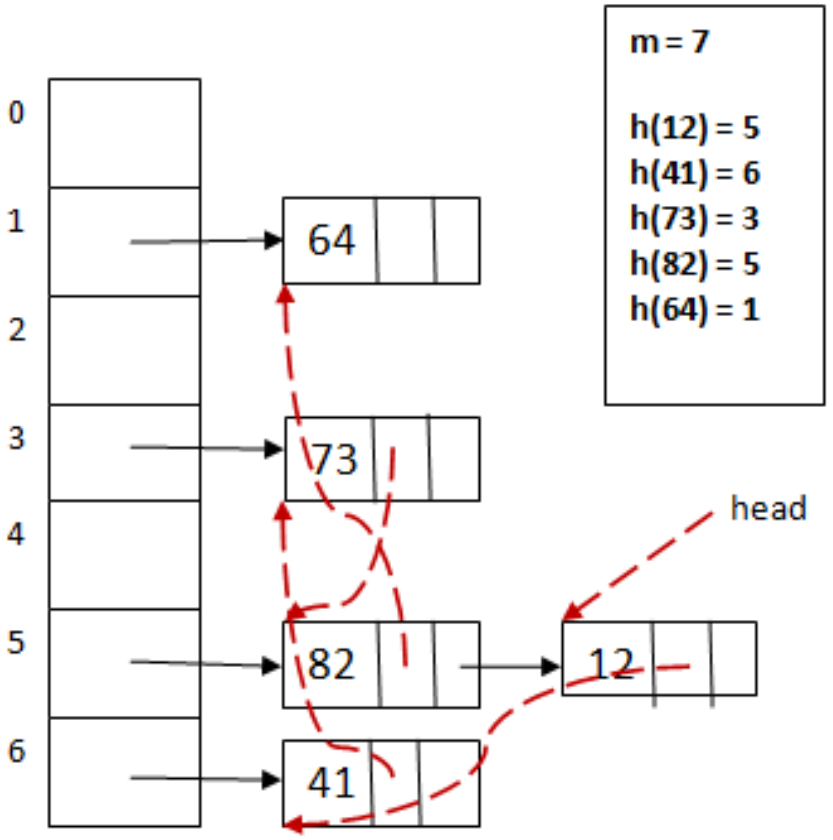
- Z.B. wenn die Elemente 12, 41, 73, 82, 64 in diese Reihenfolge in einer verketteten Hashtabelle eingefügt wurden, dann werden diese mithilfe des Iterators genau in dieser Reihenfolge iteriert

- Wie kann man eine verkettete Hashtabelle implementieren?

Verkettete Hashtabellen

- Eine verkettete Hashtabelle ist eine Mischung zwischen Hashtabelle und verkettete Listen.
- Jedes Element wird **in der Hashtabelle gespeichert** und **zusätzlich** ist das Element auch **Teil einer verketteten Liste**, in der die Elemente in der Einfüge-Reihenfolge eingefügt werden
- Da es um eine Hashtabelle geht, will man im Durchschnitt $\Theta(1)$ für die Einfüge-, Lösch- und Suchoperationen brauchen
- Diese Operationen werden genau wie vorher implementiert und die verkettete Liste wird nur für das Iterieren benutzt

Verkettete Hashtabellen



- Die roten Pfeile zeigen, wie die Elemente verkettet sind (in der Reihenfolge, in der diese in die Hashtabelle eingefügt wurden)
- Head der Liste ist das Element 12, das als erstes eingefügt wurde

Verkettete Hashtabellen

- Braucht man eine doppelt verkette Liste oder ist eine einfach verkettete Liste genug?

- Die einzige Operation, die nicht effizient auf eine einfach verkettete Liste implementiert werden kann ist die *Löschoperation.*
- Wenn man ein Element aus einer einfach verketteten Liste löschen will, dann braucht man das Element davor, aber um dieses Element zu suchen braucht man *O(n)* Zeit

Verkettete Hashtabellen - Implementierung

- Welche Datenstrukturen braucht man, um verkettete Hashtabellen zu implementieren?

Node:
info:TKey
nextH: ↑ Node //Pointer zu dem nächsten Knoten in Kollision
nextL: ↑ Node //Pointer zu dem nächsten Knoten in der Einfüge-Reihenfolge Liste
prevL: ↑ Node //Pointer zu dem vorigen Knoten in der Einfüge-Reihenfolge Liste

LinkedHT:
m: Integer
T: (↑ Node)[]
h: TFunction
head: ↑ Node
tail: ↑ Node

Verkettete Hashtabellen - Insert

- Wie kann man die Einfügeoperation implementieren?

```
subalgorithm insert(lht, k) is:
//pre: lht ist eine LinkedHT, k ist ein Key
//post: k wird in lht eingefügt
    allocate(newNode)
    [newNode].info ← k
    @setze alle Pointers von newNode auf NIL
    pos ← lht.h(k)
    //erstmal wird newNode in die Hashtabelle eingefügt
    if lht.T[pos] = NIL then
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
//Fortsetzung auf der nächsten Folie
```

Verkettete Hashtabellen - Insert

```
    //jetzt wird newNode am Ende der Einfüge-Reihenfolge Liste eingefügt
    if lht.head = NIL then
        lht.head ← newNode
        lht.tail ← newNode
    else
        [newNode].prevL ← lht.tail
        [lht.tail].nextL ← newNode
        lht.tail ← newNode
    end-if
end-subalgorithm
```

Verkettete Hashtabellen - Remove

- Wie kann man die Löschoperation implementieren?

```
subalgorithm remove(lht, k) is:
//pre: lht ist eine LinkedHT, k ist ein Key
//post: k wird aus lht gelöscht
    pos ← lht.h(k)
    current ← lht.T[pos]
    nodeToBeRemoved ← NIL
    //man sucht nach k in der Kollisionsliste und man löscht das Element, falls es gefunden wird
    if current ≠ NIL and [current].info = k then
        nodeToBeRemoved ← current
        lht.T[pos] ← [current].nextH
    else
        prevNode ← NIL
        while current ≠ NIL and [current].info ≠ k execute
            prevNode ← current
            current ← [current].nextH
        end-while
//Fortsetzung auf der nächsten Folie
```

Verkettete Hashtabellen - Löschen

```
        if current ≠ NIL then
                nodeToBeRemoved ← current
                [prevNode].nextH ← [current].nextH
        else
                @k ist nicht in lht
        end-if
end-if
//falls k in lht gefunden wurde, dann enthält nodeToBeRemoved die Adresse des Knotens, der den Wert k
//enthält und der Knoten wurde schon aus der Kollisionsliste gelöscht
//man muss den Knoten auch aus dem Einfüge-Reihenfolge Liste löschen
if nodeToBeRemoved ≠ NIL then
        if nodeToBeRemoved = lht.head then
                if nodeToBeRemoved = lht.tail then
                        lht.head ← NIL
                        lht.tail ←  NIL
                else
                        lht.head ← [lht.head].nextL
                        [lht.head].prevL ← NIL
                end-if
//Fortsetzung auf der nächsten Folie
```

Verkettete Hashtabellen - Löschen

```
        else if nodeToBeRemoved = lht.tail then
            lht.tail ← [lht.tail].prevL
            [lht.tail].nextL ← NIL
        else
            [[nodeToBeRemoved].nextL].prevL ← [nodeToBeRemoved].prevL
            [[nodeToBeRemoved].prevL].nextL ← [nodeToBeRemoved].nextL
        end-if
        deallocate(nodeToBeRemoved)
    end-if
end-subalgorithm
```