Datenstrukturen und Algorithmen

Vorlesung 9

Überblick

- Vorige Woche:
  - Binärer Heap
  - Binomial-Heap

- Heute betrachten wir:
  - Hashtabellen:
    - Intro
    - Unabhängige Verkettung
    - Verzahnte Verkettung

Beispiel

- Man braucht eine Anwendung welche Daten über die Buslinien aus einer Stadt speichert:
  - Die Nummer der Buslinie ist eine Zahl, die eindeutig ist
  - Die Nummer gehört zu einem relativ kleinen Intervall – in Cluj-Napoca ungefähr bis 100, plus die M-Linien
  - Wenn man eine Buslinienummer auswählt, dann soll der Fahrplan und die Linienkarte angezeigt werden
  - Man kann Buslinien einfügen, löschen oder ändern

- Wie kann man die Daten speichern, so dass die Anwendung effizient ist?

Direkt adressierbare Tabellen

- Formalisieren der Aufgabe:
  - Es gibt Daten, für welche jedes Element einen Schlüssel hat (eine natürliche Zahl)
  - Die Universalmenge der Schlüssel (alle mögliche Werte) ist relativ klein, $U = \{0, 1, ..., m - 1\}$
  - Es gibt keine zwei Elemente mit demselben Schlüssel
  - Man muss die Wörterbuchoperationen unterstützen: INSERT, DELETE, SEARCH

Direkt adressierbare Tabellen

- Lösung:
  - Die direkte Adressierung ist eine einfache Methode, die gut funktioniert, wenn die Universalmenge *U* der Schlüssel relativ klein ist.
  - Um die dynamische Menge darzustellen, verwenden wir ein Array, auch *direkt adressierbare Tabelle* genannt, das wir mit T[0 … m–1] bezeichnen und in dem jede Position einem Schlüssel der Universalmenge *U* entspricht
  - Die Daten für das Element mit Schlüssel *k* werden an der Position T[*k*] gespeichert
  - Wenn die Menge kein Element mit dem Schlüssel *k* enthält, dann gilt T[*k*] = NIL

Direkt adressierbare Tabellen

- In einer direkt adressierbaren Tabelle gibt es unterschiedliche Möglichkeiten die Elemente zu speichern:
  - Man kann in der Tabelle Zeiger auf die Elemente speichern
  - Man kann in der Tabelle die eigentlichen Elemente speichern (sowohl den Schlüssel als auch den entsprechenden Wert)
  - Man kann in der Tabelle nur den Wert speichern (der Schlüssel entspricht der Position) – man muss aber irgendwie wissen ob eine Position leer ist oder nicht

Direkt adressierbare Tabellen – Operationen

function search(T, k) **is**:
//pre: T ist ein Array (die direkt adressierbare Tabelle), k ist ein Schlüssel
search ← T[k]
end-function

subalgorithm insert(T, x) **is**:
//pre: T ist ein Array (die direkt adressierbare Tabelle), x ist ein Element
T[key(x)] ← x //key(x) gibt den Schlüssel des Elementes x zurück
end-subalgorithm

subalgorithm delete(T, x) **is**:
//pre: T ist ein Array (die direkt adressierbare Tabelle), x ist ein Element
T[key(x)] ← NIL
end-subalgorithm

Direkt adressierbare Tabellen

- Vorteile der direkten Adressierung:
  - Einfach
  - Effizient – alle Operationen haben $\Theta(1)$ Komplexität
- Nachteil der direkten Adressierung (Einschränkungen):
  - Die Schlüsseln müssen natürliche Zahlen sein
  - Wenn die Universalmenge *U* groß ist, kann die Speicherung einer Tabelle *T* der Größe $|U|$ unpraktikabel sein
  - Die Menge *K* der Schlüssel, die tatsächlich gespeichert werden, kann im Vergleich zu *U* so klein sein, dass der größte Teil des zugewiesenes Speicherplatzes verschwendet wäre

Hashtabellen

- Hashtabellen sind eine Verallgemeinerung der direkt adressierbaren Tabellen

- Eine Hashtabelle erfordert wesentlich weniger Speicherplatz als eine direkt adressierbare Tabelle

- Ein Element in einer Hashtabelle suchen braucht auch $\Theta(1)$ Zeit, aber im **durchschnittlicher Fall** und nicht im schlimmsten Fall

Hashtabellen

- Es gibt immer noch eine Tabelle T von Größe *m* (aber *m* ist nicht die Anzahl der möglichen Schlüsselwerte) - **Hashtabelle**
- Wir verwenden eine **Hashfunktion** *h,* um den Slot für den Schlüssel *k* zu berechnen
- Hierbei bildet *h* die Universalmenge *U* der Schlüssel in die Menge der Slots einer Hashtabelle T[0...m – 1] ab:

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

- Bemerkung:
  - Bei der direkten Adressierung wird ein Element mit dem Schlüssel *k* an der Position *k* gespeichert, also in *T*[*k*]
  - Beim Hashing wird dieses Element an der Position ***h(k)*** gespeichert, also in *T*[*h(k)*]

Hashtabellen

- Die Idee der Hashfunktion ist es, das Intervall der Indizes für das Array zu reduzieren ⇒ anstatt $|U|$ Werte gibt es nur m Werte für die Indexe

- Folglich, kann es passieren, dass zwei Schlüssel an derselben Position abgebildet werden ⇒ **Kollision**
- Es gibt unterschiedliche Verfahren zur Behandlung der durch die Kollisionen verursachten Konflikte

- Wichtige Diskussionspunkte für Hashtabellen sind:
  - Wie man die Hashfunktion definiert
  - Wie die Kollisionen behandelt werden

Hashfunktion

- Eine gute Hashfunktion:
  - Kann die Anzahl der **Kollisionen minimieren** (gar keine Kollisionen ist aber nicht möglich)
  - Ist **deterministisch**
  - Kann in $\boldsymbol{\Theta(1)}$ **Zeit ausgewertet** werden
  - Idealerweise Annahme des **einfachen, gleichmäßigen Hashens** erfüllt– d.h jeder Schlüssel ist ungefähr gleichermaßen wahrscheinlich auf eine der *m* Positionen gehashed, unabhängig davon wo andere Schlüssel gehashed werden:

$$P(h(k) = j) = \frac{1}{m}, \forall j = 0, \dots, m-1, \forall k \in U$$

Hashfunktion

- Beispiele für schlechte Hashfunktionen:
  - h(k) = konstante Zahl
  - h(k) = zufällige Zahl
  - h(k) = k mod 10 , wenn m > 10
  - usw.

Hashfunktion

- Die „**ideale**“ Eigenschaften einer Hashfunktion sind in der **Praxis nicht möglich**, da die Wahrscheinlichkeitsverteilung der auftretenden Schlüssel nicht im Voraus bekannt ist; ferner sind diese möglicherweise nicht unabhängig
- In der Praxis können wir häufig heuristische Verfahren (basierend auf der Grundmenge der Schlüssel) anwenden, um eine Hashfunktion mit guter Performanz zu erzeugen.
- Hashfunktionen sind für natürliche Zahlen definiert. Wenn die Schlüssel nicht natürliche Zahlen sind, müssen diese als natürliche Zahl interpretiert werden.

Hashfunktion

- Es gibt unterschiedliche Verfahren um Hashfunktionen zu definieren:
  - Die Divisionsmethode
  - Die Mittquadratmethode
  - Die Multiplikationsmethode
  - Universelles Hashing

Divisionsmethode

Divisionsmethode:

h(k) = k mod m, wobei *m* die Größe der Hashtabelle ist

- Zum Beispiel für m = 13:
  - k = 63 ⇒ h(k) = 11
  - k = 52 ⇒ h(k) = 0
  - k = 131 ⇒ h(k) = 1
- Auswertung schnell, da nur eine Division erforderlich
- Gute Wahl für *m:* Primzahl nicht allzu nahe an einer Zweierpotenz

Mittquadratmethode

- Nehmen wir an, dass die Größe der Tabelle 10^r ist, zum Beispiel $100 = 10^2$ (r = 2)
- Für den Hashwert berechnet man erstmal das Quadrat der Zahl und dann wählt man die mittleren r Ziffern.
- Zum Beispiel, h(4567) = die mittleren 2 Ziffern von 4567 * 4567 = die mittleren 2 Ziffern von 20857489 sind 57
- Man kann derselbe Algorithmus auf die binäre Darstellung der Zahlen anwenden, wenn $m = 2^r$.
- $m = 2^4$, h(1011) = die mittleren 4 Ziffern von 01111001 sind 1110

Multiplikationsmethode

Multiplikationsmethode :

$h(k) = \lfloor m(kA \bmod 1) \rfloor$, wobei *m* die Größe der Hashtabelle und A eine Konstante aus dem Intervall (0,1) ist

- Hier wird die Hashfunktion wie folgt konstruiert:
  1. Wahl einer Konstanten *A* im Intervall (0,1)
  2. Multiplikation des Schlüssels *k* mit *A*
  3. Bestimmung des gebrochenen (nichganzzahligen) Anteils von *kA*
  4. Multiplikation des gebrochenen Anteils mit *m*
  5. Ergebnis nach unten abrunden

Multiplikationsmethode

- Zum Beispiel:
  - m = 8, A = 0.6180339887
  - k = 3 ⇒ h(k) = $\lfloor 8((3 * A) \; mod \; 1)\rfloor = \lfloor 8((1.8541019661) \; mod \; 1)\rfloor = \lfloor 8 * 0.8541019661\rfloor = \lfloor 6.8328157288\rfloor = 6$
  - k = 1 ⇒ h(k) = $\lfloor 8((1 * A) \; mod \; 1)\rfloor = \lfloor 4.9442719096\rfloor = 4$
  - k = 63 ⇒ h(k) = $\lfloor 8((63 * A) \; mod \; 1)\rfloor = \lfloor 7.4891303048\rfloor = 7$
- Vorteile: Wahl von *m* nicht kritisch
- Nachteil: Auswertung langsamer als bei Divisionsmethode
- Manche Werte für A funktionieren besser; Knuth schlägt den Wert $\frac{\sqrt{5}-1}{2} = 0.6180339887$ vor

Universelles Hashing

- Wenn die Hashfunktion, die bei der Hashtabelle benutzt wird, bekannt ist, dann kann man immer eine Menge von Schlüsseln generieren, welche alle zu derselben Position gehasht werden (Kollision)
- Das kann die Leistung der Hashtabelle reduzieren
- Zum Beispiel:
  - m = 13
  - h(k) = k mod m
  - k = 11, 24, 37, 50, 63, 76, usw.

Universelles Hashing

- Anstatt einer einzigen Hashfunktion kann man eine Sammlung von Hashfunktionen *H* benutzen, welche die Universalmenge *U* der Schlüssel in die Menge {0, 1, ..., m – 1} abbildet
- *H* heißt **universell**, falls für jedes Schlüsselpaar $k, l \in G, k \neq l$, gilt:

$$|\{h \in H | h(k) = h(l)\}| \leq \frac{|H|}{m}$$

- *H* ist **universell**, falls für jede zufällig ausgewählte Hashfunktion $h \in H$ die **Kollisionswahrscheinlichkeit** für zwei verschiedene Schlüssel **höchstens 1/m** beträgt, also nicht größer ist als in dem Fall, dass die Hash-Werte für *k* und *l* gleichverteilt zufällig aus {0, 1, . . . ,m − 1} ausgewählt werden.

Universelles Hashing

Beispiel 1:

Sei p eine Primzahl > der maximale Wert für einen Schlüssel aus U

Für jede $a \in \{1, ..., p - 1\}$ und $b \in \{0, 1, ..., p - 1\}$ kann man eine Hashfunktion definieren:

$$h_{a,b}(k) = ((a * k + b) \bmod p) \bmod m$$

- Zum Beispiel:
  - $h_{3,7}(k) = ((3 * k + 7) \bmod p) \bmod m$
  - $h_{4,1}(k) = ((4 * k + 1) \bmod p) \bmod m$
  - $h_{8,0}(k) = ((8 * k) \bmod p) \bmod m$
- Es gibt $p * (p - 1)$ mögliche Hashfunktionen

Universelles Hashing

Beispiel 2:

Wenn der Schlüssel k ein Array ist $< k_1, k_2, ..., k_r >$, sodass $k_i < m$ (oder es kann in einem Array umgewandelt werden)

Sei $< x_1, x_2, ..., x_r >$ eine Sequenz von zufälligen Zahlen, sodass $x_i \in \{0, ..., m - 1\}$

$$h(k) = (\sum_{i=1}^{r} k_i * x_i) \, mod \, m$$

Universelles Hashing

Beispiel 3:

- Nehmen wir an, dass die Schlüssel *u* Bits enthalten und m = 2^b
- Man wählt eine zufällige $b \times u$ Matrix *h* mit den Werten 0 und 1.
- Man wählt $h(k) = h * k$, wo die Multiplikation mit Hilfe der Addition mod 2 berechnet wird:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

Schlüssel, die nicht natürliche Zahlen sind

- Die vorigen Hashfunktionen arbeiten mit Schlüsseln, die natürliche Zahlen sind
- Falls das nicht der Fall ist, dann gibt es zwei Möglichkeiten:
  - Man definiert eine spezielle Hashfunktion, die mit den Schlüsseln arbeitet (z.B. $h(k) = [k * m]$ für reele Zahlen aus dem Intervall $[0, 1)$)
  - Man benutzt eine Funktion, welche die Schlüssel in natürliche Zahlen umwandelt (dann kann man eine der vorigen Hashfunktionen auf das Ergebnis anwenden) – *hashCode* in Java, *hash* in Python

Schlüssel, die nicht natürliche Zahlen sind

- Falls der Schlüssel ein Wort ist, dann kann man:
  - Die ASCII-Codes für jede Buchstabe betrachten
  - Die Buchstaben nummerieren 1 für a, 2 für b, usw.
- Mögliche Implementierungen für Hashcode:
  - s[0] + s[1] + ... + s[n-1]
    - Anagramme haben dieselbe Summe (AMPEL und LAMPE)
    - DATES hat zum Beispiel dieselbe Summe mit CAUSE (D = C + 1, T = U – 1)
    - Nehmen wir an, dass die maximale Länge eines Wortes 10 ist. Dann nimmt der Hashcode Werte zwischen 1 und 260. Für ein Wörterbuch mit 50000 Wörter, hat man im Durchschnitt 192 Wörter pro Hashcode-Wert.

Schlüssel, die nicht natürliche Zahlen sind

- Mögliche Implementierungen für Hashcode:
  - $s[0] * 26^{n-1} + s[1] * 26^{n-2} + ... + s[n-1]$
    - wobei n die Länge des Strings ist
    - Erzeugt ein viel größeres Intervall von Hashcode-Werten
    - Anstatt 26 (die Anzahl der Buchstaben) kann man eine Primzahl auswählen (z.B. Java benutzt 31)
- Vielleicht will man auch Großbuchstaben, Sonderzeichen, andere Schriftsysteme u.a. haben...

Kollisionen

- Wenn es zwei Schlüssel, *x* und *y,* gibt mit demselben Hashwert *h(x) = h(y)* dann gibt es eine Kollision
- Eine gute Hashfunktion kann die Anzahl der Kollisionen minimieren, aber nicht vollständig entfernen:
  - Versuche *m + 1* Schlüssel in einer Tabelle von Größe *m* zu verteilen
- Es gibt unterschiedliche Strategien für Kollisionsauflösung:
  - **Unabhängige Verkettung** (separate chaining)
  - **Verzahnte Verkettung** (coalesced chaining)
  - **Offene Adressierung**
  - usw

Geburtstagsparadox

- Wie viele zufällig ausgewählten Personen müssen sich in einem Raum befinden, damit die Wahrscheinlichkeit, dass zwei oder mehr dieser Personen am gleichen Tag (ohne Beachtung des Jahrganges) Geburtstag haben, größer als 50 % ist ?
- Offensichtlich, wenn sich in dem Raum 367 Personen befinden, dann gibt es bestimmt zwei, die am gleichen Tag Geburtstag haben
- Damit die Wahrscheinlichkeit 99.9% ist, braucht man 70 Personen
- Damit die Wahrscheinlichkeit 50% ist, genügen 23 Personen

Kollisionsauflösung: **unabhängige Verkettung**

Unabhängige Verkettung

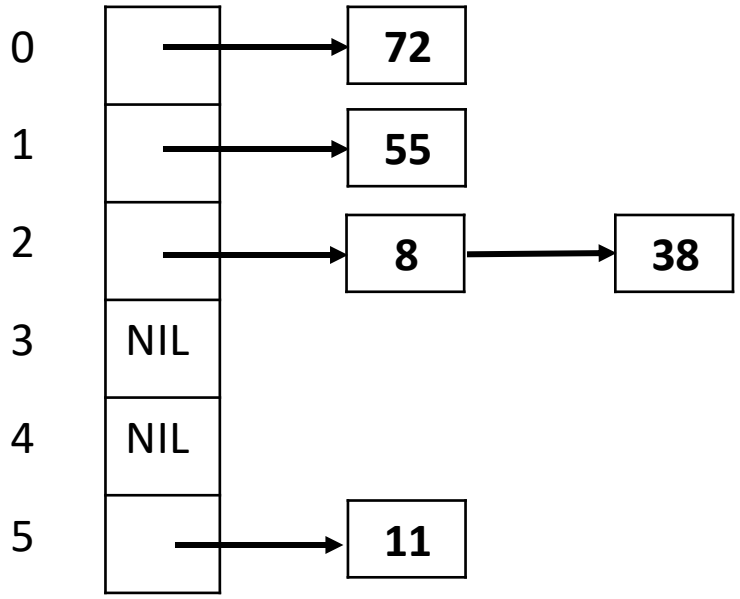
- Bei der Verkettung speichert man alle Elemente, die auf den gleichen Slot abgebildet werden, in einer verketteten Liste
- Slot *j* enthält einen Zeiger auf den Kopf der Liste aller gespeicherten Elemente, die auf *j* abgebildet werden, oder NIL falls es keine solche Elemente gibt
- Wörterbuchoperationen werden als Operationen auf der Liste implementiert:
  - *insert(T, x)* – Füge einen neuen Knoten am Anfang der Liste *T[h(key[x])]*
  - *search(T, k)* – Suche in der Liste *T[h(k)]* nach einem Element mit dem Schlüssel *k*
  - *delete(T, x)* – Entferne *x* aus der Liste *T[h(key[x])]*

Unabhängige Verkettung – Beispiel

- Sei eine Hashtabelle von Größe $m = 6$ mit unabhängiger Verkettung als Kollisionsbehandlung und eine Hashfunktion mit der Divisionsmethode
- Wie sieht die Hashfunktion aus?
- Füge in die Tabelle folgende Elemente ein: 38, 11, 8, 72, 55

Unabhängige Verkettung – Beispiel

- h(38) = 2 (Belegungsfaktor $\alpha = 1/6$)
- h(11) = 5 (Belegungsfaktor $\alpha = 2/6$)
- h(8) = 2 (Belegungsfaktor $\alpha = 3/6$)
- h(72) = 0 (Belegungsfaktor $\alpha = 4/6$)
- h(55) = 1 (Belegungsfaktor $\alpha = 5/6 = 0.83$)



Hashtabellen mit unabhängiger Verkettung - Repräsentierung

- Als Vereinfachung speichern wir nur die Schlüssel in den Knoten
- Eine Hashtabelle mit unabhängiger Verkettung kann folgendermaßen repräsentiert werden:

Node:

key: TKey

next: ↑ Node

HashTable:

T: ↑ Node[] //ein Array von Pointers zu den Knoten

m: Integer

h: TFunction //die Hashfunktion

Hashtabellen mit unabhängiger Verkettung – insert

```
subalgorithm insert(ht, k) is:
//pre: ht ist ein HashTable, k ist ein TKey
//post: k wurde in ht eingefügt
    position ← ht.h(k)      //man berechnet den Hashwert
    allocate(newNode)
    [newNode].next ← NIL
    [newNode].key ← k
    if ht.T[position] = NIL then
        ht.T[position] ← newNode
    else
        [newNode].next ← ht.T[position]
        ht.T[position] ← newNode
    end-if
end-subalgorithm
```

Hashtabellen mit unabhängiger Verkettung – search

```
function search(ht, k) is:
//pre: ht ist ein HashTable, k ist ein TKey
//post: gibt True zurück falls k in ht enthalten ist, False ansonsten
    position ← ht.h(k)
    currentN ← ht.T[position]
    while currentN ≠ NIL and [currentN].key ≠ k execute
        currentN ← [currentN].next
    end-while
    if currentN ≠ NIL then
        search ← True
    else
        search ← False
    end-if
end-function
```

- Normalerweise sollte die Suchfunktion den Wert entsprechend zu dem Schlüssel zurückgeben anstatt wahr oder falsch, aber jetzt arbeiten wir nur mit den Schlüsseln

Analyse des Hashings mit Verkettung

- Die durchschnittliche Laufzeit des Hashings hängt davon ab, wie gut die Hashfunktion *h* die Menge der zu speichernden Schlüssel auf die *m* Slots verteilt
- **Einfaches gleichmäßiges Hashing** Annahme: jedes beliebige gegebene Element wird mit gleicher Wahrscheinlichkeit und unabhängig von den anderen Elementen auf einer der *m* Slots abgebildet
- Zu einer gegebenen Hashtabelle *T* mit *m* Slots, die *n* Elemente speichert, definieren wir den **Belegungsfaktor** α als ***n/m***, d.h. als die durchschnittliche Anzahl der Elemente, die in einer Kette der Hashtabelle gespeichert sind
- Der Belegungsfaktor für Hashing mit unabhängige Verkettung kann kleiner, gleich oder größer als 1 sein

Analyse des Hashings mit Verkettung – *insert*

- Der Slot, wo man das Element einfügen muss, kann:
  - leer sein – dann erstellt man einen neuen Knoten und man fügt ihn zu dem leeren Slot ein
  - besetzt – dann erstellt man einen neuen Knoten und man fügt ihn am Anfang der Liste ein
- Die Laufzeit für das Einfügen ist für beide Fälle im schlechtesten Fall $\Theta(1)$
- Wenn man überprüfen muss, ob das Element schon in der Tabelle existiert, dann kommen noch die Kosten der Suchoperation dazu

Analyse des Hashings mit Verkettung – *search*

- Es gibt zwei Fälle:
  - Erfolglose Suche
  - Erfolgreiche Suche

- Wir setzen voraus:
  - der Hashwert kann in $\Theta(1)$ berechnet werden
  - die Zeit, die für das Suchen eines Elementes mit dem Schlüssel *k* benötigt wird, hängt linear von der Länge der Liste *T[h(k)]* ab

Analyse des Hashings mit Verkettung – *search*

- **Theorem**: In einer Hashtabelle, in der Kollision durch Verkettung aufgelöst wird, benötigt eine **erfolglose Suche** unter der Annahme des einfachen gleichmäßigen Hashings eine Laufzeit im Durchschnitt von $\boldsymbol{\Theta(1+\alpha)}$**.**

- **Theorem**: In einer Hashtabelle, in der Kollision durch Verkettung aufgelöst wird, benötigt eine **erfolgreiche Suche** unter der Annahme des einfachen gleichmäßigen Hashings eine Laufzeit im Durchschnitt von $\boldsymbol{\Theta(1+\alpha)}$**.**

- Beweis Idee: $\Theta(1)$ wird gebraucht um den Wert der Hashfunktion zu berechnen und α ist die Durchschnittszeit, die gebraucht wird, um eine der m Listen zu durchsuchen

Analyse des Hashings mit Verkettung – *search*

- Falls *n = O(m)* (die Anzahl der Slots in der Hashtabelle ist proportional zur Anzahl der Elemente in der Tabelle)
  - $\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$
  - Die Suche benötigt im Durchschnitt konstante Zeit

- Die Komplexität im schlimmsten Fall ist Θ(n)
  - Alle *n* Schlüssel könnten auf denselben Slot abgebildet werden, wodurch eine Liste der Länge *n* erzeugt werden würde (dann durchsucht man die ganze Liste)
  - Im Praxis sind Hashtabellen effizient

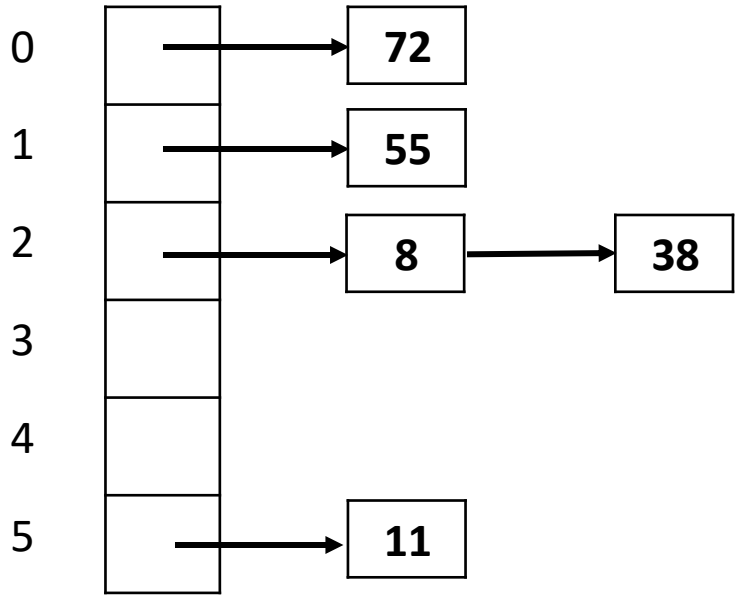
Analyse des Hashings mit Verkettung – *delete*

- Löschen:
  - Falls die Liste doppelt verkettet ist und man kennt die Adresse des Knotens: $\Theta(1)$
  - Falls die Liste einfach verkettet ist: Zeit ist linear mit der Länge der Liste

- **Alle Wörterbuchoperationen können im Durchschnitt in Zeit $\boldsymbol{\Theta(1)}$ ausgeführt werden**
- Theoretisch kann man eine beliebige Anzahl von Elementen in einer Hashtabelle mit unabhängige Verkettung speichern, aber die Komplexität ist proportional zu α. Falls α zu groß ist $\Rightarrow$ resize und rehash

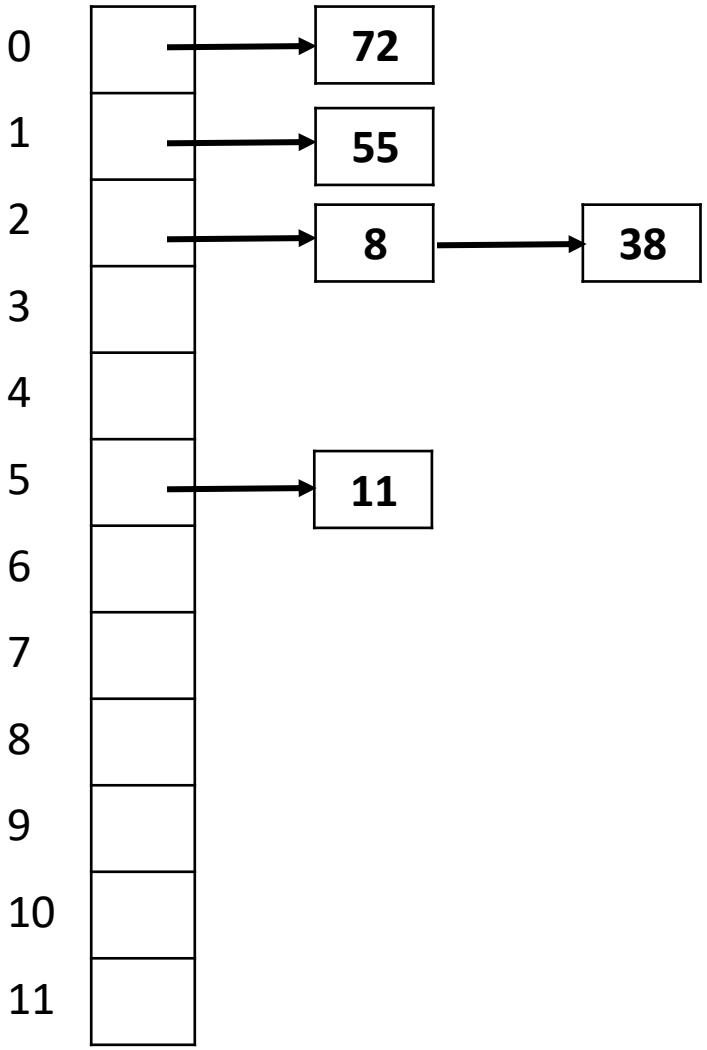
Unabhängige Verkettung – Beispiel

- h(38) = 2 (Belegungsfaktor $\alpha = 1/6$)
- h(11) = 5 (Belegungsfaktor $\alpha = 2/6$)
- h(8) = 2 (Belegungsfaktor $\alpha = 3/6$)
- h(72) = 0 (Belegungsfaktor $\alpha = 4/6$)
- h(55) = 1 (Belegungsfaktor $\alpha = 5/6 = 0.83$)



Unabhängige Verkettung – Beispiel

- Ist es in Ordnung wenn die Hashtabelle nach der resize Operation wie folgt aussieht?



Unabhängige Verkettung

- Was muss man bei der resize Operation tun?
  - Die Hashfunktion hängt von der Größe der Hashtabelle ab! Wenn die Größe geändert wird, dann ändern sich auch die Hashwerte der Elemente
  - Nach der resize Operationen müssen alle Elemente neu eingefügt werden → **rehash**

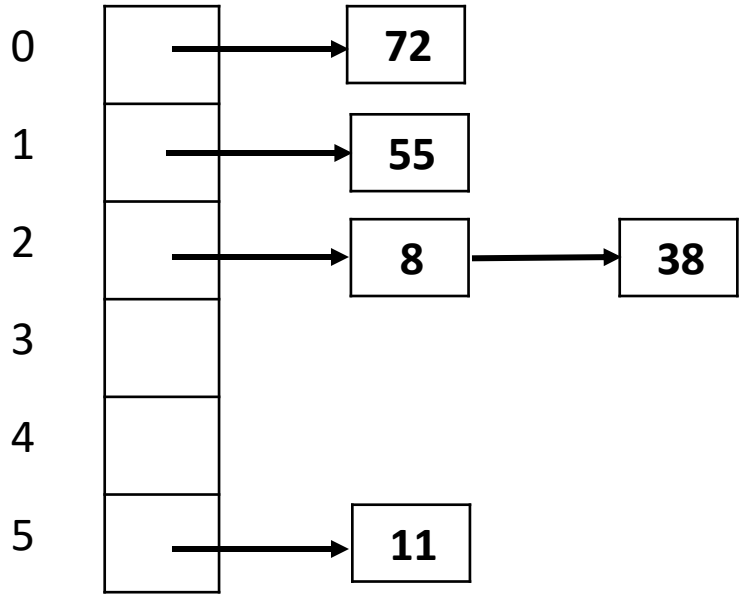
Unabhängige Verkettung

- Welche Containers können nicht auf einer Hashtabelle repräsentiert werden?

- Wie kann man einen sortierten Container auf einer Hashtabelle implementieren?
  - Hashtabellen sind nicht sehr geeignet für sortierte Containers
  - Wenn man aber einen sortierten Container auf Hashtabellen repräsentiert, dann kann man die Listen sortiert behalten – siehe Seminar

Iterator

- Wie kann man einen Iterator für eine Hashtabelle mit unabhängiger Verkettung definieren?
- Da die Reihenfolge der Elemente nicht wichtig ist, kann der Iterator die Elemente in einer beliebigen Reihenfolge iterieren
- Für die Hashtabelle aus dem vorigen Beispiel ist die einfachste Reihenfolge für das Iterieren: 72, 55, 8, 38, 11



Iterator

- Der Iterator für eine Hashtabelle mit unabhängiger Verkettung ist eine Mischung zwischen dem Iterator für ein Array und dem Iterator für eine verkettete Liste
- Man braucht eine aktuelle Position für die Hashtabelle und einen aktuellen Knoten für die verkettete Liste

IteratorHT:

h: HashTable

currentPos: Integer

currentNode: ↑ Node

Iterator - init

```
subalgorithm init(ith, ht) is:
//pre: ith ist ein IteratorHT, ht ist ein HashTable
    ith.ht ← ht
    ith.currentPos ← 0
    while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute
        ith.currentPos ← ith.currentPos + 1
    end-while
    if ith.currentPos < ht.m then
        ith.currentNode ← ht.T[ith.currentPos]
    else //die Hashtabelle ist leer
        ith.currentNode ← NIL
    end-if
end-subalgorithm
```

- Komplexität: $O(m)$

Iterator

- Wie kann man die *getCurrent* Operation implementieren?
- Wie kann man die *next* Operation implementieren?
- Wie kann man die *valid* Operation implementieren?

Iterator - next

```
subalgorithm next(ith) is:
    if [ith.currentNode].next ≠ NIL then
        ith.currentNode ← [iht.currentNode].next
    else
        ith.currentPos ← ith.currentPos + 1
        while ith.currentPos < ith.ht.m and ith.ht.T[ith.currentPos]=NIL execute
            ith.currentPos ← ith.currentPos + 1
        end-while
        if ith.currentPos ≠ NIL then
            ith.currentNode ← ith.ht.T[ith.currentPos]
        else
            ith.currentNode ← NIL
        end-if
    end-if
end-subalgorithm
```

- Komplexität: *O(m)*

Kollisionsauflösung:
verzahnte Verkettung

Verzahnte Verkettung (coalesced chaining)

- Sollte beim Einfügen eine Kollision auftreten, wird der Wert in **eine freie Position** eingetragen und **die Position in die Liste**, die von der ursprünglich berechneten Position ausgeht, eingehängt (in dem *next* Feld)
- Wenn ein neues Element eingefügt wird und die Position, an die es gespeichert werden soll, besetzt ist, setzt man es an eine beliebige leere Position und man setzt den *next* Link, damit das Element bei einer Suche gefunden werden kann.
- Belegungsfaktor kann höchstens 1 sein.

Coalesced Verkettung - Beispiel

- Sei eine Hashtabelle mit der Größe $m = 16$ mit coalesced Verkettung als Kollisionsbehandlung und eine Hashfunktion mit der Divisionsmethode
- Wir wollen folgende Werte in die Tabelle einfügen: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13

Schlüssel	76	12	109	43	22	18	55	81	91	27	13
Hashwert	12	12	13	11	6	2	7	1	11	11	13

Schlüssel	76	12	109	43	22	18	55	81	91	27	13
Hashwert	12	12	13	11	6	2	7	1	11	11	13

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T													76			
next	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

firstEmpty = 0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	12												76			
next	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 1

Schlüssel	76	12	109	43	22	18	55	81	91	27	13
Hashwert	12	12	13	11	6	2	7	1	11	11	13

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	12		18				22	55				43	76	109		
next	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 1

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

firstEmpty = 3

Schlüssel	76	12	109	43	22	18	55	81	91	27	13
Hashwert	12	12	13	11	6	2	7	1	11	11	13

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

firstEmpty = 4

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

firstEmpty = 8

Coalesced Verkettung – Repräsentierung

- Welche Felder benötigt man, um eine Hashtabelle mit coalesced Verkettung darzustellen?
- Als Vereinfachung speichern wir nur die Schlüssel in den Knoten

HashTable:

T: TKey[]
next: Integer[]
m: Integer
firstEmpty: Integer
h: TFunction //die Hashfunktion

Coalesced Verkettung – insert

```
subalgorithm insert(ht, k) is:
//pre: ht ist ein HashTable, k ist ein Tkey
//post: k wurde in ht eingefügt
    pos ← ht.h(k)
    if ht.T[pos] = -1 then //die Position ist leer
        ht.T[pos] ← k
        ht.next[pos] ← -1
        if ht.firstEmpy = pos then
            changeFirstEmpty(ht)
        end-if
    else
        if ht.firstEmpty = ht.m  then
            @resize and rehash
        end-if
```

Coalesced Verkettung – insert

```
            current ← pos
            while ht.next[current] ≠ -1 execute
                  current ← ht.next[current]
            end-while
            ht.T[ht.firstEmpy] ← k
            ht.next[ht.firstEmpty] ←-1
            ht.next[current] ← ht.firstEmpty
            changeFirstEmpty(ht)
      end-if
end-subalgorithm
```

- Komplexität:

$\Theta(1)$ in dem durchscnittlichen Fall, $\Theta(n)$ in dem schlimmsten Fall

Coalesced Verkettung – ChangeFirstEmpty

```
subalgorithm changeFirstEmpty(ht) is:
//pre: ht ist ein HashTable
//post: der Wert von ht.firstEmpty wird auf die nächste freie Position gesetzt
    ht.firstEmpty ← ht.firstEmpty + 1
    while ht.firstEmpty < m and ht.T[firstEmpty] ≠ -1  execute
        ht.firstEmpty ← ht.firstEmpty + 1
    end-while
end-subalgorithm
```

- Komplexität: $\mathrm{O}(m)$
- Denke nach: sollten wir die freien Positionen in einer Liste verketten so wie bei den verketteten Listen auf Arrays?

Coalesced Verkettung

- *remove* und *search* Operationen werden beim Seminar besprochen

- Wie kann man einen Iterator für Hashtabelle mit coalesced Verkettung definieren? Wie kann man die Iterator Operationen implementieren?
  - init
  - getCurrent
  - next
  - valid