

# Algorithmen I

## Tutorium 1 - 4. Sitzung

Dennis Felsing

`dennis.felsing@student.kit.edu`  
`www.stud.uni-karlsruhe.de/~ubcqr/algo`

2011-05-09



# Überblick

- 1 Verkettete Listen
- 2 Unbeschränkte Felder
- 3 Amortisierte Laufzeitanalyse
- 4 Hashing
- 5 Kreativaufgabe

# Verkettete Listen

- 1 Verkettete Listen**
  - Wiederholung
  - Wächterelement
  - Einfach verkettete Listen
- 2 Unbeschränkte Felder
- 3 Amortisierte Laufzeitanalyse
- 4 Hashing
- 5 Kreativaufgabe

# Verkettete Listen

## Funktionsweise

Objekte linear angeordnet, Zugriff mit Zeigern

## Varianten

- Einfach und doppelt verkettet
- Unsortiert und sortiert
- Nichtzyklisch und zyklisch

Wir betrachten doppelt verkettete, unsortierte, nichtzyklische Listen.

## Operationen

- SEARCH: Suche ein Element ( $O(n)$ )
- INSERT: Füge Element ein ( $O(1)$ )
- DELETE: Entferne Element ( $O(1)$ )

# Wächterelement

## Definition

Ein **Wächterelement** (englisch Sentinel) ist ein Dummy-Objekt, dass dazu dient die Sonderbehandlung in Listen zu vereinfachen.

## Nachteil

Erhöhter Speicherbedarf (bei kleinen Listen relevant)

# Wächterelement

## Listensuche mit Wächterelement

LIST-SEARCH'(L, k)

```
1  x = L.nil.next
2  while x ≠ L.nil and x.key ≠ k
3      x = x.next
4  return x
```

## Aufgabe

Die Listensuche mit Wächterelement (aus der Vorlesung) ist nicht einfacher als die ohne. Entwerfe einen Algorithmus LIST-SEARCH'', der im jeweils schlechtesten Fall  $n = |L|$  Operationen weniger benötigt.

# Einfach verkettete Listen

## Vorteile

- Geringerer Speicherbedarf
- Setzen der Vorgängerzeiger entfällt  $\Rightarrow$  Einfacherer Code

## Nachteile

- Löschen aufwendiger
- Vorgängerelement nicht einfach ermittelbar

## Aufgabe

Wie lässt sich das Löschen eines Elementes aus einer einfach verketteten Liste in konstanter Zeit bewerkstelligen?

# Unbeschränkte Felder

## Idee

Feldgröße dynamisch anpassen: Vergrößern wenn kein Platz mehr vorhanden, Verkleinern wenn viel freier Platz

## Operationen

- $\text{PUSH}(S, x)$ : Hinzufügen von Element  $x$  an Ende des Feldes. Falls Feld voll, Feldgröße verdoppeln
- $\text{POP}(S)$ : Entnahme von letztem Element. Falls Feld nur  $\frac{1}{4}$  belegt, Feldgröße halbieren

Amortisiert beides in  $O(1)$ .

Warum nicht einfach Listen benutzen? Felder haben wahlfreien Zugriff in  $O(1)$



# Amortisierte Laufzeitanalyse

## Idee

Betrachte gemittelten Aufwand für Folge von Operationen im Worst-Case.

## Account-Methode

- Jeder Operation werden Token (Zeiteinheiten) zugewiesen
- Nicht verbrauchte Token werden auf Guthabenkonto eingezahlt
- Teure Operationen nehmen sich Token von Guthabenkonto zum bezahlen

# Amortisierte Laufzeitanalyse

## Account-Methode

- Jeder Operation werden Token (Zeiteinheiten) zugewiesen
- Nicht verbrauchte Token werden auf Guthabenkonto eingezahlt
- Teure Operationen nehmen sich Token von Guthabenkonto zum bezahlen

## Beispiel: $k$ -Pop-Stack

Gegeben sei ein Stack  $S$  mit folgenden Operationen:

- $PUSH(S, x)$ : Hinzufügen von Element  $x$  auf  $S$
- $POP(S, k)$ :  $k$  Elemente vom  $S$  nehmen

Zeige: Eine beliebige Operationenfolge  $\sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_m \rangle$  von  $PUSH$ - und  $POP$ -Operationen auf  $S$  wird in  $O(m)$  ausgeführt. Jede einzelne Operation ist somit amortisiert in  $O(1)$ .

# Hashing

- 1 Verkettete Listen
- 2 Unbeschränkte Felder
- 3 Amortisierte Laufzeitanalyse
- 4 Hashing**
  - Adresstabellen mit direktem Zugriff
  - Hashtabellen
  - Kollisionen und deren Behandlung
  - Hashfunktionen
  - Universelles Hashing
- 5 Kreativaufgabe

# Adresstabellen mit direktem Zugriff

## Ziel

Entwicklung einer Datenstruktur, die die Wörterbuch-Operationen INSERT, SEARCH und DELETE performant unterstützt.

## Idee

- Elemente haben Schlüssel aus Universum  $U = \{0, 1, \dots, m - 1\}$
- Schlüssel kommen nicht doppelt vor (kennzeichnen Element)
- Element in Tabelle  $T[0..m - 1]$  an Slot des Schlüssels ablegen (Vorsicht: hier ab 0)

## Laufzeiten

Laufzeit für INSERT, SEARCH und DELETE ist  $O(1)$  im Worst-Case.

# Hashtabellen

## Probleme von Adrestabellen mit direktem Zugriff

- Großes Universum  $U \Rightarrow$  Tabelle  $T$  groß
- Viel Platz verschwendet

## Idee

- Wir haben ein großes Universum  $U$  und eine Tabelle  $T[0..m-1]$
- Sei  $h$  eine Hashfunktion  $h : U \rightarrow \{0, 1, \dots, m-1\}$
- Element  $x$  mit Schlüssel  $k$  an Slot  $h(k)$  in Tabelle ablegen

## Laufzeit

- Average-Case: Alles  $O(1)$
- Worst-Case für SEARCH:  $\Theta(n)$

# Kollisionen und deren Behandlung

## Situation

Zwei verschiedene Schlüssel  $k_1$  und  $k_2$  werden auf den gleichen Hashwert abgebildet, also  $h(k_1) = h(k_2)$ . In Tabelle ist aber nur Platz für ein Element.

## Kollisionsauflösung durch Verkettung

Zeiger auf Verkettete Liste an jeder Stelle in Tabelle

- Beliebig viele Einträge

## Lineares Sondieren (Offene Adressierung)

Bei Kollision bei INSERT in nächsten freien Slot eintragen

- Beschränkte Größe
- Löschen kompliziert

# Hashfunktionen

Sei  $h$  eine Hashfunktion  $h : U \rightarrow \{0, 1, \dots, m - 1\}$ .

## Beispiel Hashfunktion

$U = \mathbb{Z}_0, m = 10, h : \mathbb{Z}_0 \rightarrow \{0, \dots, 9\}, h(k) = k \bmod 10$   
 $h(4) = 4, h(19) = 9, h(34123) = 3, h(230) = 0$

## Wünschenswerte Eigenschaften

- Einfach und schnell berechenbar (deterministisch)
- Gleichmäßige Verteilung

# Aufgabe

Führe für je eine Hashtabelle

- mit Kollisionsauflösung durch Verkettung und  $h(x) = x \bmod 10$
- mit Offener Adressierung und  $h(x) = x \operatorname{div} 10$

folgende Operationen durch:

- INSERT(13)
- INSERT(35)
- INSERT(47)
- INSERT(12)
- INSERT(25)
- DELETE(25)
- INSERT(14)
- DELETE(13)
- SEARCH(14)



# Universelles Hashing

Situation: Angreifer kennt Hashfunktion und wählt  $n$  Schlüssel, die auf selben Slot abbilden  $\Rightarrow$  Laufzeit  $\Theta(n)$  im Average-Case

## Universelle Hashfunktion

- Wähle zufällige Hashfunktion aus Familie von universellen Hashfunktionen aus
- Universelle Hashfunktion verursacht mit Wahrscheinlichkeit  $\frac{1}{m}$  eine Kollision

$\Rightarrow$  Erwartete Laufzeit  $O(1)$  für alle Operationen

# Entwurf Algorithmus

Gegeben sei ein Array  $A = A[1], \dots, A[n]$  mit  $n$  Zahlen in beliebiger Reihenfolge. Für eine gegebene Zahl  $x$  soll ein Paar  $(A[i], A[j])$ ,  $1 \leq i, j \leq n$  gefunden werden, für das gilt:  $A[i] + A[j] = x$ .

- 1 Gib eine Lösung für  $x = 33$  und  $A = (7, 15, 21, 14, 18, 3, 9)$  an.
- 2 Gib einen Algorithmus an, der das Problem löst und bei Erfolg ein Paar  $(A[i], A[j])$  ausgibt, ansonsten *nil*. Versuche den Zeitaufwand zu minimieren

# Übersicht

- 1 Verkettete Listen**
  - Wiederholung
  - Wächterelement
  - Einfach verkettete Listen
- 2 Unbeschränkte Felder**
- 3 Amortisierte Laufzeitanalyse**
- 4 Hashing**
  - Adresstabellen mit direktem Zugriff
  - Hashtabellen
  - Kollisionen und deren Behandlung
  - Hashfunktionen
  - Universelles Hashing
- 5 Kreativaufgabe**

(AN UNMATCHED LEFT PARENTHESIS  
CREATES AN UNRESOLVED TENSION  
THAT WILL STAY WITH YOU ALL DAY.