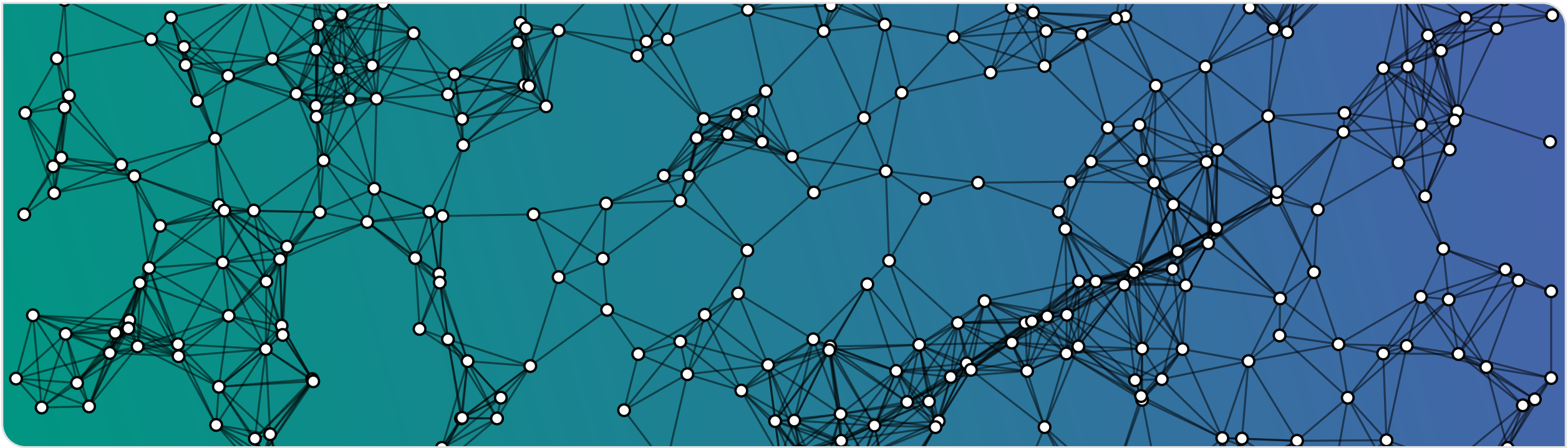
Karlsruher Institut für Technologie

Algorithmen 1

Dynamische Arrays & amortisierte Analyse



Datenstrukturen

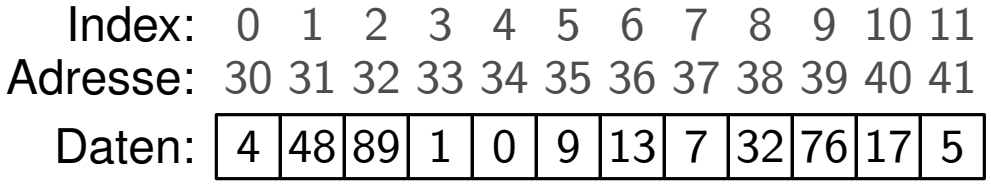
Was ist das überhaupt?

- Algorithmus: Abfolge von Schritten. Jeder Schritt . . .
  - liest (wenige) Daten im Speicher
  - verarbeitet die gelesenen Daten
  - schreibt Daten in den Speicher
- Zugriff auf den Speicher mittels Speicheradressen
- Datenstruktur: Abstraktionsebene über dem Speicher
  - versteckt, wie Daten genau im Speicher abgelegt sind
  - bietet Schnittstelle mit angenehmen Operationen

Zwei Arten der Datenablage

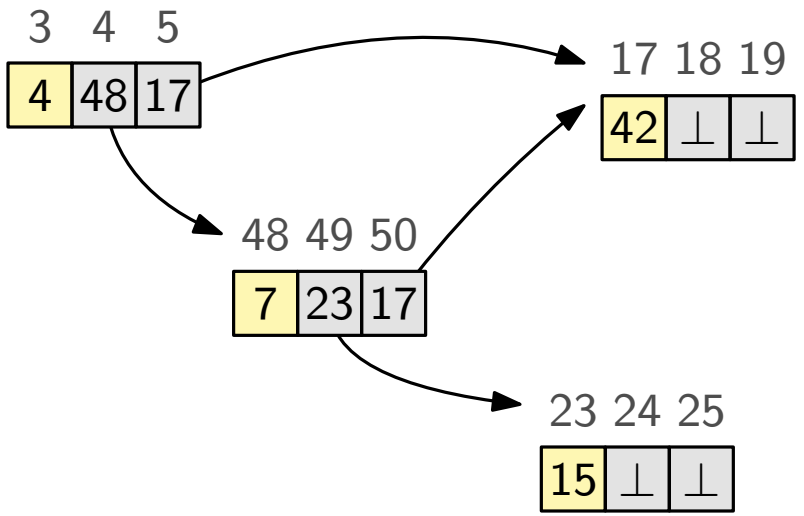
Felder (Arrays)

- Menge aufeinanderfolgender Speicherzellen
- Zugriff mit Adresse bzw. Index an beliebiger Stelle in $\Theta(1)$
- sehr nah an der Hardware
- heute: Komfort-Funktionen



Verzeigerte Strukturen

- viele kleine Stückchen Speicher (Knoten)
- ein Knoten speichert:
  - Daten, die uns tatsächlich interessieren
  - Speicheradressen anderer Knoten (Zeiger)
- Zugriff durch Navigation entlang Zeiger
- abstrahiert stärker von der Hardware
- nächstes Mal: Listen als einfaches Beispiel



Beschränkte und Unbeschränkte Arrays

Beschränkte Arrays

Index:	0	1	2	3	4	5	6	7	8	9	10	11
Adresse:	30	31	32	33	34	35	36	37	38	39	40	41
Daten:	4	48	89	1	0	9	13	7	32	76	17	5

- beim Erstellen des Arrays muss ich festlegen, wie viel Speicher ich brauche
- nachträgliches Vergrößern schwierig: nachfolgende Speicherzellen ggf. schon belegt
- Problem: benötigter Speicher ist ggf. durch äußere Faktoren bedingt (Nutzereingabe)

Ich wünsche mir: unbeschränkte Arrays

- Verhalten wie bei Arrays (direkter Zugriff mittels Index)
- zusätzliche Funktionen:
  - | 4 | 48 | 89 | 1 | 0 | .**pushBack**(9) → | 4 | 48 | 89 | 1 | 0 | 9 |
  - | 4 | 48 | 89 | 1 | 0 | .**popBack**() → | 4 | 48 | 89 | 1 |
  - | 4 | 48 | 89 | 1 | 0 | .**size**() → 5
- Wünschen kann man sich ja viel, aber wie setzen wir das um?

Unbeschränkte Arrays: naives Vorgehen

Umsetzung von pushBack(x)

- erstelle neues Array, das um 1 größer ist
- kopiere Daten von altem in neues Array
- schreibe x in die freie Zelle am Ende
- lösche altes Array

Kosten für das Einfügen von Elementen

- Einfügen von $i$tem Element: Kosten $\Theta(i)$
- Einfügen von n Elementen: $\sum_{i=1}^{n} \Theta(i) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$
- im Schnitt: Kosten $\Theta(n)$ pro Operation $\rightarrow$ viel zu teuer

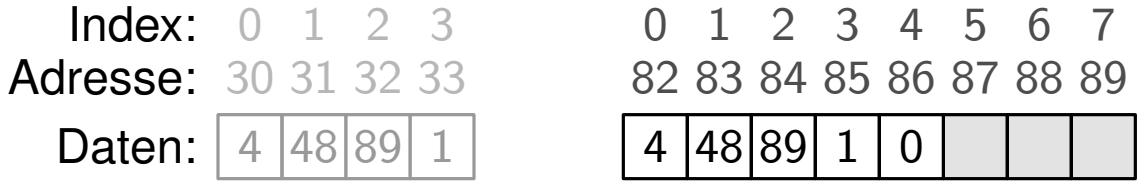
Müssen wir wirklich jedes Mal umkopieren?

- alloziere großzügig etwas mehr Speicher
- nur dann kopieren, wenn der allozierte Speicher voll ist

Unbeschränkte Arrays: besseres Vorgehen

Größe vs. Kapazität

- Kapazität: für das Array allozierter Speicher
- Größe: tatsächlich benutzte Speicherzellen
- nach außen Sichtbar: Größe (via **size**())



Umsetzung von pushBack(x)

Beispiel:
pushBack(89) **pushBack**(1) **pushBack**(0)

- falls Kapazität $>$ Größe: einfach einfügen
- sonst: erstelle neues Array mit doppelter Kapazität und verfahre wie bisher

Kosten für das Einfügen von Elementen

- Kosten für $i$tes Element, wenn $i = 2^j + 1$: $\Theta(i) = \Theta(2^j)$
- sonst: $\Theta(1)$
- Einfügen von n Elementen: $\sum_{i=1}^{n} \Theta(1) + \sum_{j=0}^{\log_2 n} \Theta(2^j) = \Theta(n) + \Theta(2^{\log_2 n}) = \Theta(n)$
- im Schnitt: Kosten $\Theta(1)$ pro Operation $\rightarrow$ besser geht's nicht

Unbeschränkte Arrays: Anmerkungen

Speicherverschwendung

- wir allozieren mehr Speicher als wir tatsächlich brauchen
- aber: höchstens einen Faktor 2 zu viel

Was machen wir bei popBack()?

- hier kann man verschiedene Strategien verfolgen
- z.B.: einfach die Größe des Arrays verringern aber Kapazität beibehalten $\rightarrow \Theta(1)$

Worst Case vs. Durchschnitt

- im Worst Case benötigt eine einzelne Operation $\Theta(n)$ Zeit ($n =$ Arraygröße)
- aber: im Schnitt benötigt jede Operation nur $\Theta(1)$ Zeit
- wir sagen: **pushBack**() hat **amortisiert** konstante Laufzeit

Abgrenzung zum Average Case

(werden wir später noch kennen lernen)

- Average Case: Durchschnitt über alle möglichen Eingaben/Ausführungen
- amortisiert: Durchschnitt über alle Operationen für beliebige (Worst Case) Eingabe

Unbeschränkte Arrays in der Wildnis

C++

`https://en.cppreference.com/w/cpp/container/vector`

std::vector

The complexity (efficiency) of common operations on vectors is as follows:

- Random access - constant $\mathcal{O}(1)$
- Insertion or removal of elements at the end - amortized constant $\mathcal{O}(1)$

operator[]	access specified element (public member function)
size	returns the number of elements (public member function)
capacity	returns the number of elements that can be held in currently allocated storage (public member function)
push_back	adds an element to the end (public member function)
pop_back	removes the last element (public member function)

Unbeschränkte Arrays in der Wildnis

Java

`https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html`

Class ArrayList<E>

Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.

Method Summary

Modifier and Type	Method	Description
`boolean`	`add(E e)`	Appends the specified element to the end of this list.
`E`	`get(int index)`	Returns the element at the specified position in this list.
`int`	`size()`	Returns the number of elements in this list.
`void`	`trimToSize()`	Trims the capacity of this `ArrayList` instance to be the list's current size.

Unbeschränkte Arrays: Zusammenfassung

Effiziente Operationen ((amortisiert) konstant)

- Zugriff an beliebiger Stelle
- hinten einfügen
- hinten löschen

Langsame Operationen (linear)

- einfügen an beliebiger Stelle
- löschen an beliebiger Stelle
- zwei Arrays konkatenieren
- Teilarray löschen

Ausflug: Amortisierte Analyse

Situation

- Sequenz von vielen Operationen
- manche sind teuer, die meisten sind günstig $\rightarrow$ günstig im Durchschnitt

Amortisierte Analyse

- Worst Case über alle möglichen Sequenzen von Operationen
- Garantie: Gesamtlaufzeit verhält sich, als wäre jede Operation günstig

Jetzt: verschiedene Techniken für die Analyse

- Aggregation
- Charging
- Konto
- Potential
- jeweils am Beispiel von n **pushBack** Operationen

Amortisierte Analyse: Aggregation

Allgemeines Vorgehen

- summiere die Kosten für alle Operationen
- teile Gesamtkosten durch Anzahl Operationen

Am Beispiel pushBack

- Kosten für das Einfügen von n Elementen: $\sum_{i=1}^{n} \Theta(1) + \sum_{j=0}^{\log_2 n} \Theta(2^j) = \Theta(n)$
- durchschnittliche Kosten pro Operation: $\Theta(1)$

Anmerkungen

- schön einfach und direkt
- manchmal unpraktikabel, insbesondere bei mehrere Operationen unterschiedlichen Typs

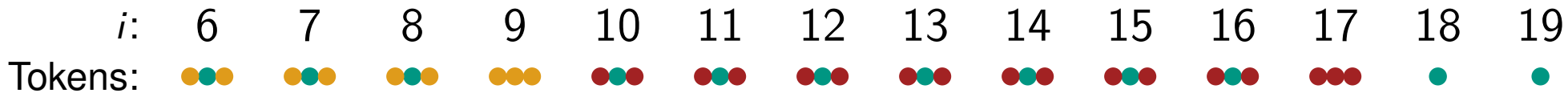
Amortisierte Analyse: Charging

Allgemeines Vorgehen

- verteile Kosten-Tokens von teuren zu günstigen Operationen (Charging)
- zeige: jede Operation hat am Ende nur wenige Tokens

Am Beispiel pushBack

- $i$te Operation hat i Tokens, falls $i = 2^j + 1$, sonst 1 Token



- verschiebe von $2^j + 1$ je zwei Tokens auf jede der $2^{j-1} - 1$ Operation in $(2^{j-1} + 1, 2^j]$
- falls $i = 2^j + 1$: Operation wird $2^j - 2 = i - 3$ Tokens los $\rightarrow$ 3 Tokens verbleiben
- sonst: Operation erhält höchstens zwei zusätzliche Tokens $\rightarrow$ maximal 3 Tokens

Anmerkung

- lokal: hohe Kosten fallen an $\rightarrow$ charge sie rückwirkend auf vorherige Operationen
- global: aufpassen, dass keine Operation zu viele Token bekommt

Amortisierte Analyse: Konto

Allgemeines Vorgehen

- günstige Operation: bezahlt mehr als sie tatsächlich Kostet $\rightarrow$ ins Konto einzahlen
- teure Operation: bezahlt tatsächlich Kosten zum Teil mit Guthaben aus dem Konto

Am Beispiel pushBack

- günstige Operation ($i \in (2^{j-1}+1, 2^j]$): Kosten 1, bezahle $3 \rightarrow$ zahle 2 ins Konto ein
- Guthaben nach Operation 2^j beträgt mindestens $2 \cdot (2^j - (2^{j-1}+1)) = 2^j - 2$
- teure Operation ($i = 2^j + 1$): Kosten i, bezahle $3 \rightarrow$ hebe $i - 3 = 2^j - 2$ vom Konto ab
- jede Operation bezahlt nur 3 und Konto ist nie negativ $\Rightarrow$ amortisiert $\Theta(1)$

Anmerkung

- lokal: günstige Operationen zahlen vorausschauend Kosten für spätere Operationen
- global: aufpassen, dass nie mehr abgehoben wird als schon eingezahlt wurde
- sehr ähnlich zum Charging, aber leicht andere Perspektive

Amortisierte Analyse: Potential

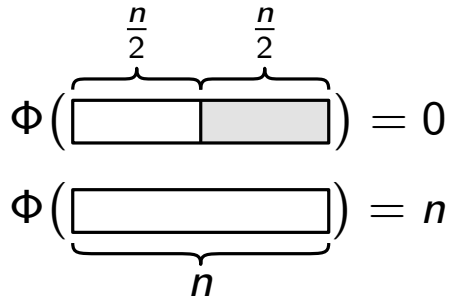
Allgemeines Vorgehen

- bei der Konto-Methode:
  - definiere wie viel jede Operation (amortisiert) bezahlt
  - daraus ergibt sich: Änderung des Kontostands
- bei der Potential-Methode:
  - definiere Kontostand abhängig vom Zustand der Datenstruktur $\rightarrow$ Potentialfunktion
  - daraus ergibt sich: die amortisierten Kosten jeder Operation
- genauer:
  - Potentialfunktion $\Phi(A)$: bildet den Zustand des Arrays A auf eine Zahl ≥ 0 ab
  - A_{vor} und A_{nach}: die Zustände vor bzw. nach einer Operation
  - amortisierte Kosten $=$ tatsächliche Kosten $+ \Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$

Amortisierte Analyse: Potential

Am Beispiel pushBack

- Potentialfunktion: $\Phi(A) = 2 \cdot A.\mathbf{size}() - A.\mathbf{capacity}()$
- beachte: $\Phi(A) \geq 0$
- günstige Operation:
  - Größe wächst um 1; Kapazität bleibt gleich
  - $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}}) = 2$
  - amortisierte Kosten $=$ tatsächliche Kosten $+ 2 \in \Theta(1)$
- teure Operation ($i = 2^j + 1$):
  - Größe wächst um 1; Kapazität wächst um 2^j
  - $\Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}}) = 2 - 2^j$
  - amortisierte Kosten $=$ tatsächliche Kosten $+ 2 - 2^j \in \Theta(1)$



Erinnerung: amortisierte Kosten $=$ tatsächliche Kosten $+ \Phi(A_{\text{nach}}) - \Phi(A_{\text{vor}})$

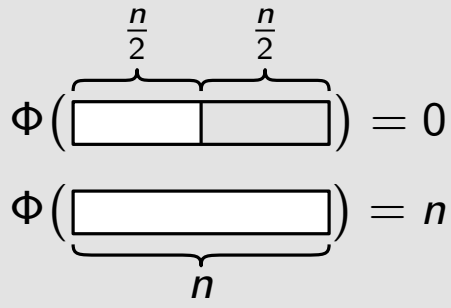
Amortisierte Analyse: Potential – Anmerkungen

Wie kommt man auf die Potentialfunktion?

- das ist der Knackpunkt; wenn man die erstmal hat ist der Rest einfach
- Interpretation 1: Maß dafür, wie nah man an einer teuren Operation ist
  - niedrig: kommende Operationen sind günstig
  - hoch: bald kommt vermutlich eine teure Operation
- Interpretation 2: Maß für die Unordnung der DS
  - niedrig: aufgeräumt; nahe dem Idealzustand
  - hoch: unordentlich; weit vom Idealzustand entfernt

Erinnerung: Φ für **pushBack**()

$\Phi(A) = 2 \cdot A.\textbf{size}() - A.\textbf{capacity}()$



Vor- und Nachteile

- Beweis oft recht kompakt
- Potentialfunktion manchmal intuitiv schwer nachvollziehbar
  (man kann das Ergebnis zwar nachrechnen, weiß aber gar nicht so genau, was man da rechnet)

Amortisierte Analyse: Zusammenfassung

Wann ist eine amortisierte Betrachtungsweise angebracht?

- bei einer Folge unterschiedlich teurer Operationen
- vor jeder teuren Operation kommen immer ausreichend viele günstige Operationen

Was garantiert eine amortisierte Laufzeitschranke?

- gute Gesamtlaufzeit (Worst Case) über alle Operationen hinweg
- keine Garantie für einzelne Operation (relevant z.B. in Echtzeitsystemen oder bei Parallelisierung)

Techniken für die amortisierte Analyse

- Aggregation: summiere Kosten über alle Operationen
- Charging: charge Kosten von teuren Operationen auf vorherige günstige Operationen
- Konto: lass günstige Operationen später folgende teure Operationen vorab bezahlen
- Potential: definiere Unordnungskonto, das misst wie unschön der aktuelle Zustand ist

Zusammenfassung

Beschränkte Arrays

- ein Stück Speicher (konsekutive Speicherzellen) auf das man mittels Adresse zugreift
- andere Arten Daten zu speichern basieren immer auf diesem Grundbaustein

Unbeschränkte Arrays

- Interface für dynamisch wachsendes Array $\rightarrow$ abstrahiert von einem Stück Speicher
- effiziente Implementierung mittels sinnvoller Größenänderungs-Strategie

Drei Sichtweisen auf eine Datenstruktur

- **Mathe** abstraktes Objekt (hier: Folge von Zahlen $\langle 4, 48, 89, 1, 0, 9, 13, 7, 32, 76, 17, 5 \rangle$)
- **Softwaretechnik** Funktionalität (hier: Zugriff mit Index, **pushBack**, **popBack**, **size**)
- **Algorithmik** Repräsentation und effiziente Umsetzung

Amortisierte Analyse

- wichtige Technik für die Analyse von Algorithmen

Hochschulgruppen

Was ist das?

- studentisch organisierte Gruppen
- große thematische Bandbreite: von sozialem Engagement bis hin zum Segelfliegen

Mitgliederwerbung

- viele Gruppen suchen neue Mitglieder
- einige Infoveranstaltungen in den nächsten Wochen
- siehe auch: Link auf der Homepage