Karlsruher Institut für Technologie

Zusatztutorium II
Datenstrukturen, Hashing, Union-Find

Algorithmen I, Zusatztutorium

Kilian Wurm | 5. August 2024

FAKULTÄT FÜR INFORMATIK

Roadmap

1. Wiederholung: Datenstrukturen
2. Wiederholung: Hashing
3. Wiederholung: Union-Find

Roadmap

Motivation

Was wollen wir?

Daten speichern!

Und was noch?

Motivation

Was wollen wir?

Daten speichern!

Und was noch?

- Einfügen (an beliebiger Stelle)
- Löschen (an beliebiger Stelle)
- Suchen
- Sortieren
- Abändern (an beliebiger Stelle)
- Wahlfreier Zugreifen
- ...

Operationen auf Datenstrukturen

Wie interagieren wir mit einer Datenstruktur?

- insert, pushBack, pushFront
- remove, popBack, popFront
- first, last
- size
- findNext
- concat, splice
- …

Listen

Listen bilden eine Aneinanderreihung einzelner Elemente ab.

Eigenschaften

Vorteile:

- Kapazität (prinzipiell) uneingeschränkt
- Zugriff auf Vorgänger- und Nachfolger-Element in $\Theta(1)$
- Erweiterung durch Einfügen am Ende in $\Theta(1)$

Nachteile:

- Zugriff auf ein beliebiges Listenelement in nur $\Theta(n)$
- zusätzlicher Speicheraufwand pro Element für Zeiger
- nicht cache-effizient, Elemente sind i.d.R. nicht benachbart

Doppelt verkettete Listen

Aufbau

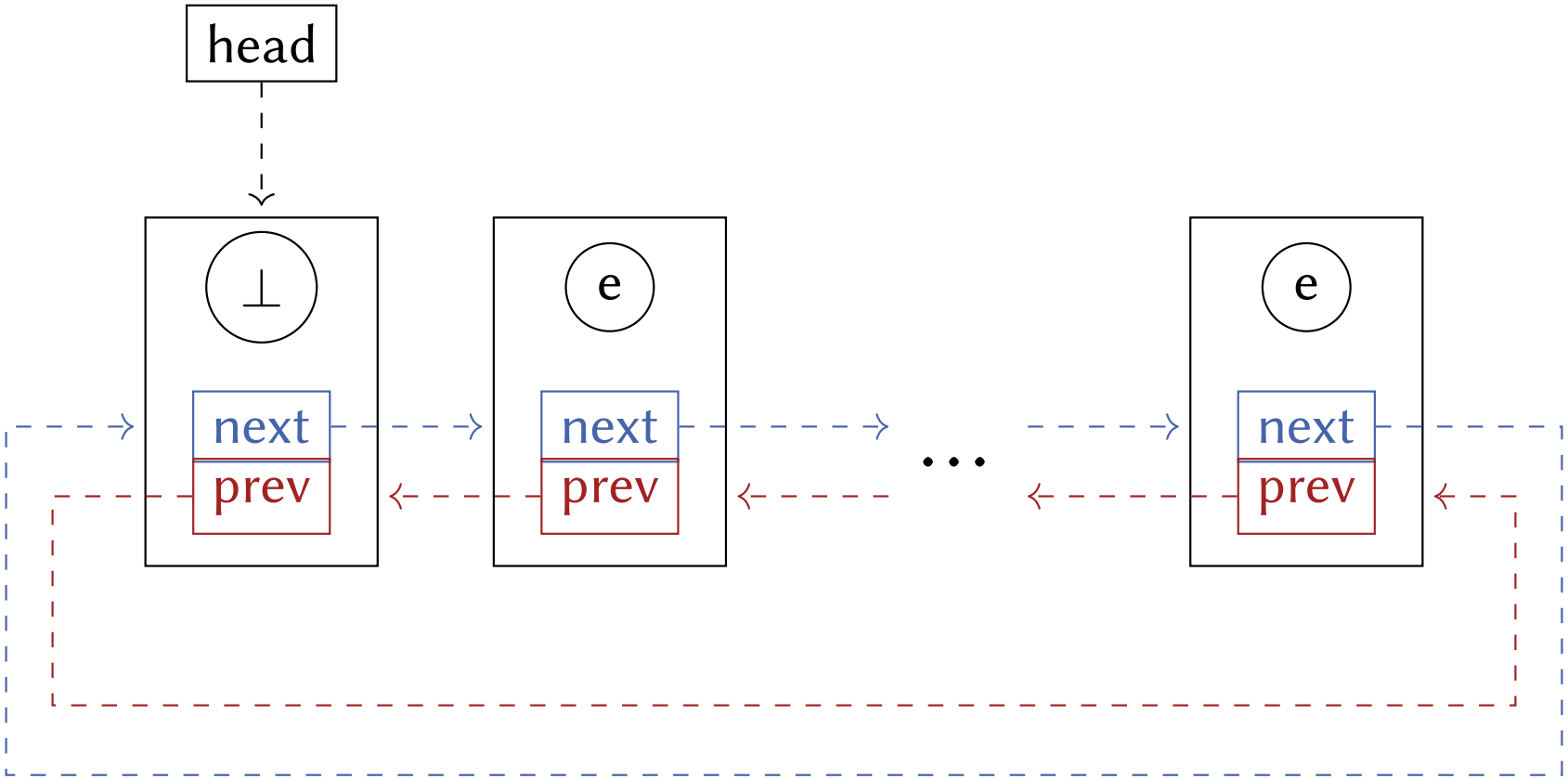
Class Item **of** ListElement
    e: ListElement
    next: Handle
    prev: Handle

Listenelement speichert:

- Inhalt
- Zeiger auf Nachfolger
- Zeiger auf Vorgänger

Als «Einstiegspunkt» gibt es einen head-Knoten, oft auch als Dummy-Header oder Wächter-Element bezeichnet.

Doppelt verkettete Listen



Der Dummy-Header

Vor- und Nachteile Dummy-Header

+ Invarianten immer erfüllt
+ Vermeidung von Sonderfällen
+ lesbarer in der Programmierung
+ lässt sich besser Testen
- verbraucht Speicherplatz

Invariante für Listenelement e

e.next.prev = e.prev.next = e

Einfach verkettete Listen

Aufbau

Class Item **of** ListElement

e: ListElement

next: Handle

Listenelement speichert:

- Inhalt
- Zeiger auf Nachfolger

Einfach verkettete Listen

Aufbau

Class Item **of** ListElement
    e: ListElement
    next: Handle

Listenelement speichert:

- Inhalt
- Zeiger auf Nachfolger

Vor- und Nachteile

+ verbraucht weniger Speicher
+ Speichergewinn oftmals auch Zeitgewinn
- Operationen, die auf DLLs einfach sind, können hier kompliziert (und langsam) werden
- hat zusätzlich die Nachteile einer DLL

Ein Dummy-Header hat auch hier Vorteile, zusätzlich erlaubt ein Zeiger auf das letzte Element pushBack-Operationen in $\Theta(1)$.

Beschränkte Felder (Arrays)

Arrays bilden Blöcke allozierten Speichers von fester Größe.

Eigenschaften

- Größe wird bei der Initialisierug festgelegt
- jedes Element ist eindeutig über Index adressierbar

Beschränkte Felder (Arrays)

Arrays bilden Blöcke allozierten Speichers von fester Größe.

Eigenschaften

- Größe wird bei der Initialisierug festgelegt
- jedes Element ist eindeutig über Index adressierbar

Vorteile:

- effizienter Zugriff auf jedes Feld in $\Theta(1)$
- cache-effizient, da zusammenhängender Speicherblock

Beschränkte Felder (Arrays)

Arrays bilden Blöcke allozierten Speichers von fester Größe.

Eigenschaften

- Größe wird bei der Initialisierug festgelegt
- jedes Element ist eindeutig über Index adressierbar

Vorteile:

- effizienter Zugriff auf jedes Feld in $\Theta(1)$
- cache-effizient, da zusammenhängender Speicherblock

Nachteile:

- beschränkte Kapazität
- kein effizientes Einfügen an einer beliebigen Position

Unbeschränkte Felder (Arrays)

Arrays, aber ohne die Nachteile von Arrays (?)

Eigenschaften

Gewünschte Operationen:

- `pushBack()`: fügt ein Element am Ende des Feldes hinzu
- `popBack()`: entfernt das letzte Element des Feldes

Unbeschränkte Felder (Arrays)

Arrays, aber ohne die Nachteile von Arrays (?)

Eigenschaften

Gewünschte Operationen:

- `pushBack()`: fügt ein Element am Ende des Feldes hinzu
- `popBack()`: entfernt das letzte Element des Feldes

Umsetzung

- verwende beschränktes Feld, reallokiere ggf.
- Kein Platz?
  Kopiere alle Einträge in ein neues Feld um, welches um einen konstanten Faktor **größer** ist
- Ku viel ungenutzter Platz?
  Kopiere alle Einträge in ein neues Feld um, welches um einen konstanten Faktor **kleiner** ist

Unbeschränkte Felder (Arrays)

1: **struct** *Array* {
2: `capacity`: $\mathbb{N}_0$
3: `size`: $\mathbb{N}_0$
4: `data`: [Element; `capacity`]

Unbeschränkte Felder (Arrays)

```
1:  struct Array {
2:      capacity: ℕ₀
3:      size: ℕ₀
4:      data: [Element; capacity]

5:  pushBack(e: Element)
6:      if size = capacity then
7:          new_data : [Element; 2 · capacity]
8:          for i ∈ {0, ..., size − 1} do
9:              new_data[i] := data[i]
10:         capacity := 2 · capacity
11:         data := new_data
12:     data[size] := e
13:     size := size + 1
14: }
```

Unbeschränkte Felder (Arrays)

```
1:  struct Array {
2:      capacity: ℕ_0
3:      size: ℕ_0
4:      data: [Element; capacity]

5:  popBack( )                                  // Gibt kein Element zurück!
6:      size := size − 1
7:      if size ≤ ⌊capacity/4⌋ then
8:          new_data : [Element; ⌊capacity/2⌋]
9:          for i ∈ {0, ..., size − 1} do
10:             new_data[i] := data[i]
11:         capacity := ⌊capacity/2⌋
12:         data := new_data
13: }
```

Unbeschränkte Felder (Arrays)

Was haben wir damit gewonnen?

Laufzeiten

Sowohl für `pushBack` als auch für `popBack` gilt:

- Liegt `size` im tolerierten Bereich[1]?
  $\Rightarrow$ konstante Laufzeit
- Ansonsten: lineare Laufzeit durch Umkopieren

Aber: Sowohl `pushBack` als auch `popBack` haben amortisiert konstante Laufzeit.

[1] in Abhängigkeit von `capacity` und den gewählten Faktoren

Vergleich: Listen vs Felder

Vorteile von Listen

- flexibel
- kein ungenutzter Speicherplatz
- effizientes Einfügen von Elementen an eine beliebige Position

Vergleich: Listen vs Felder

Vorteile von Listen

- flexibel
- kein ungenutzter Speicherplatz
- effizientes Einfügen von Elementen an eine beliebige Position

Vorteile von Feldern

- effizienter Zugriff auf beliebige Elemente
- kein Overhead für Zeiger
- Cache-effizientes iterieren
- simpel in Handhabung und Implementierung

Vergleich: Listen vs Felder

Nachteile von Listen

- Zeiger benötigen zusätzlichen Speicher
- kein effizienter beliebiger Zugriff
- komplizierter in Handhabung und Impelementierung

Vergleich: Listen vs Felder

Nachteile von Listen

- Zeiger benötigen zusätzlichen Speicher
- kein effizienter beliebiger Zugriff
- komplizierter in Handhabung und Impelementierung

Nachteile von Feldern

- kein effizientes Einfügen von Elemente an eine beliebiger Position
- ungenutzter Speicherplatz
- einzelne Operationen haben nur amortisiert gute Laufzeit
- nicht sehr flexibel

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?
- Iteration über alle Elemente?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?
- Iteration über alle Elemente?
- Vereinen zweier Folgen?

Vergleich

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?
- Iteration über alle Elemente?
- Vereinen zweier Folgen?
- Direkter Zugriff auf einzelne Elemente?

Kosten der Operationen

Operation	Liste	einfach verkettete Liste	Array	Erklärung ($*$)
first	1	1	1	
last	1	1	1	
insert	1	1*	n	nur insertAfter
remove	1	1*	n	nur removeAfter
pushBack	1	1	1*	amortisiert
pushFront	1	1	n	
popBack	1	n	1*	amortisiert
popFront	1	1	n	
concat	1	1	n	
splice	1	1	n	
findNext	n	n	n	

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?
- Iteration über alle Elemente?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?
- Iteration über alle Elemente?
- Vereinen zweier Folgen?

Alles klar soweit?

Aufgabe

Welche der gerade behandelten Datenstrukturen (einfach-, doppeltverkettete Listen, Felder) sollte man wählen (und weshalb) bei:

- Zugriff auf das erste Element?
- Austauschen einzelner Elemente?
- Entfernen einzelner Elemente?
- Suchen eines Elements?
- Suchen eines Elements in einer sortierten Folge?
- Speicherverbrauch?
- Iteration über alle Elemente?
- Vereinen zweier Folgen?
- Direkter Zugriff auf einzelne Elemente?

Roadmap

Was war nochmal Hashing?

Was war nochmal Hashing?

Und wozu war das nochmal gut?

Dr. Metas Problem

Der ebenso geniale wie misstrauische Superbösewicht Dr. Meta hat einen schweren Schlag erlitten: Nach dem letzten Coup, den er gegen seinen Erzfeind Theorie-Man startete, ging seine Flotte der superschnellen Meta-Mobile unter ungeklärten Umständen in Flammen auf. Da dies nicht das erste Mal war, beschließen sich einige seiner Investoren, ihm die Geldmittel zu streichen.
Dr. Meta muss sparen!

Er beschließt kurzerhand, Personalkosten zu sparen und den Teil seiner Mitarbeiter den Krokodilen vorzuwerfen, die seiner Einschätzung nach im letzten Jahr zu viel Urlaub nehmen.

Dr. Metas Problem

Leider hat Dr. Meta für diese Aufgabe nur eine Aufzählung der Form (Mitarbeiter-ID, Tag), wobei eine ID eindeutig und ein Tag eine natürliche Zahl aus $\{0, \ldots, 364\}$.

Natürlich kennt Dr. Meta auch noch die Anzahl seiner Mitarbeiter von seiner letzten Säuberungsaktion.

Er sucht nun einen Algorithmus, der in erwartet linearer Laufzeit die IDs der Mitarbeiter ausgibt, die mehr als zwei Wochen Urlaub genommen haben. Jede ID darf dabei nicht mehr als einmal ausgegeben werden.

Dr. Metas Problem

Leider hat Dr. Meta für diese Aufgabe nur eine Aufzählung der Form (Mitarbeiter-ID, Tag), wobei eine ID eindeutig und ein Tag eine natürliche Zahl aus $\{0, \ldots, 364\}$.

Natürlich kennt Dr. Meta auch noch die Anzahl seiner Mitarbeiter von seiner letzten Säuberungsaktion.

Er sucht nun einen Algorithmus, der in erwartet linearer Laufzeit die IDs der Mitarbeiter ausgibt, die mehr als zwei Wochen Urlaub genommen haben. Jede ID darf dabei nicht mehr als einmal ausgegeben werden.

Aber wie?

Dr. Metas Problem

Leider hat Dr. Meta für diese Aufgabe nur eine Aufzählung der Form (Mitarbeiter-ID, Tag), wobei eine ID eindeutig und ein Tag eine natürliche Zahl aus $\{0, \ldots, 364\}$.

Natürlich kennt Dr. Meta auch noch die Anzahl seiner Mitarbeiter von seiner letzten Säuberungsaktion.

Er sucht nun einen Algorithmus, der in erwartet linearer Laufzeit die IDs der Mitarbeiter ausgibt, die mehr als zwei Wochen Urlaub genommen haben. Jede ID darf dabei nicht mehr als einmal ausgegeben werden.

Aber wie?

Lösungsansätze

Was können wir nutzen?

Lösungsansätze

Was können wir nutzen?

- Daten pro Tag speichern?
  - nicht zielführend, wir wollen Daten pro Mitarbeiter
- Mitarbeiter-IDs nutzen?
  - Problem: IDs sind unabhängig von der Anzahl Mitarbeiter

  $\Rightarrow$ wir können die maximale ID in Linearzeit bestimmen, doch das bringt uns nichts

Optimal wäre: für jeden Mitarbeiter einen Eintrag, auf den man in $\Theta(1)$ zugreifen kann.
Das ist i.d.R. nicht möglich, da wir zu wenig über die IDs wissen. Aber wir können dem nahe kommen…

Motivation I

«hashing» zu deutsch: «zerhacken»

Wir zerhacken die Gesamtheit unseres Inputs in Untermengen, die jeweils über einen Hashwert adressierbar sind.

Motivation I

«hashing» zu deutsch: «zerhacken»

Wir zerhacken die Gesamtheit unseres Inputs in Untermengen, die jeweils über einen Hashwert adressierbar sind.

Wozu braucht man das?

Hashtabellen sind eine Möglichkeit, assoziative Arrays zu implementieren.

Intuition: Ein Array mit potentiell unendlicher / sehr großer Indexmenge, bei dem nur sehr wenige Indizes tatsächlich benutzt werden

(Zum Beispiel Mitarbeiter-IDs)

Etwas formaler …

Motivation II

Wir wollen eine Datenstruktur aufbauen, die ein assoziatives Array implementiert. Sie soll die Operationen

- GET(`key`)
- SET(`key`, `value`)
- DELETE(`key`)

in erwartet konstanter Zeit ausführen können.

Dabei machen wir uns Schlüssel zu nutze, um Elemente in der Datenetruktur zu finden.

Vorhang auf für …

Hashtabellen

Eine Hashtabelle ist ein Array von m Buckets zusammen mit einer Hashfunktion $h : U \to \{0, \ldots, m-1\}$.

Hashtabellen

Eine Hashtabelle ist ein Array von m Buckets zusammen mit einer Hashfunktion $h : U \to \{0, \ldots, m-1\}$.

Wollen wir ein Element e in eine Hashtabelle eintragen, müssen wir Folgendes leisten:

Hashtabellen

Eine Hashtabelle ist ein Array von m Buckets zusammen mit einer Hashfunktion $h : U \to \{0, \ldots, m-1\}$.

Wollen wir ein Element e in eine Hashtabelle eintragen, müssen wir Folgendes leisten:

1. den (eindeutigen) Schlüssel $\texttt{key}(e)$ bestimmen
2. den Hashwert $h(\texttt{key}(e))$ berechnen
3. e am Index $h(\texttt{key}(e))$ in A einfügen

Hashing

Def.: Hash-Funktion

Sei M eine Menge von beliebigen Elementen. Jedes Element $e \in M$ hat dabei einen eindeutigen Schlüssel $\texttt{key}(e)$, d.h. es gilt

$$\forall e_1, e_2 \in M : \texttt{key}(e_0) = \texttt{key}(e_1) \Rightarrow e_0 = e_1$$

Die Menge aller möglichen Schlüssel nennen wir das Universum U.
Eine Hash-Funktion **h** bildet Schlüssel auf natürliche Zahlen aus $\{0, \ldots, m-1\}$ für ein $m \in \mathbb{N}$ ab.

Elemente vs. Schlüssel vs. Hashwert

$$\text{Element} \xrightarrow{\texttt{key}} \text{Schlüssel} \xrightarrow{h} \text{Hashwert}$$

Aber: Elemente können ihr eigener Schlüssel sein (z.B. numerische Werte)

Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

- **Surjektivität**: alle Hashwerte können genutzt werden
- **Gleichverteilung**: alle Hashwerte werden mit der gleichen Wahrscheinlichkeit getroffen
- Kollisionen sind möglichst selten

Was, wenn es beim Einfügen eines Elements zur Kollision kommt?

Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

- **Surjektivität**: alle Hashwerte können genutzt werden
- **Gleichverteilung**: alle Hashwerte werden mit der gleichen Wahrscheinlichkeit getroffen
- Kollisionen sind möglichst selten

Was, wenn es beim Einfügen eines Elements zur Kollision kommt?
Kollisionsauflösung!

Hashing

Welche Kriterien erfüllt eine «gute» Hashfunktion?

- **Surjektivität**: alle Hashwerte können genutzt werden
- **Gleichverteilung**: alle Hashwerte werden mit der gleichen Wahrscheinlichkeit getroffen
- Kollisionen sind möglichst selten

Was, wenn es beim Einfügen eines Elements zur Kollision kommt?
Kollisionsauflösung!

Sind Kollisionen Anzeichen einer schlechten Hashfunktion?

Nein! Allein schon deswegen, weil wir die Anzahl der Elemente, die wir in eine Hashtabelle einfügen, nicht beschränken, sind Kollisionen unvermeidbar.

Auswahl einer geeigneten Hashfunktion

Problem: In der Praxis können wir in der Regel keine Aussagen über die Menge an Schlüsseln machen, die wir der Hashtabelle übergeben. Entscheiden wir uns für eine feste Hashfunktion, ist es allein abhängig von der Schlüsselmenge, wie (un-)gleichmäßig die Elemente verteilt werden.

Frage: Wie können wir trotzdem eine möglichst günstige Hashfunktion wählen?

Idee der zufällig gewählten Hashfunktion

Lösung: Wir wählen unsere Hashfunktion aus einer Familie (geeigneter) Hashfunktionen zufällig.

Natürlich kann diese Funktion immer noch ungünstig für unsere späteren Eingaben sein. Der Vorteil ist jedoch, dass die Wahrscheinlichkeit dafür sinkt.

Universelles Hashing

Definition: Universelle Hashfamilie

Sei $\mathcal{H}$ eine Menge von Hashfunktionen, welche von U auf $\{0, .., \ m-1\}$ abbilden.
$\mathcal{H}$ heißt **universell**, wenn für ein zufällig gewähltes $h \in \mathcal{H}$ gilt:

$$\forall k, \ l \in U, \ k \neq l : \ \Pr[h(k) = h(l)] = \frac{1}{m}$$

Anders formuliert: Wählen wir eine Hashfunktion $h \in \mathcal{H}$ zufällig, ist die Kollisionswahrscheinlichkeit für zwei beliebige Schlüssel $\frac{1}{m}$.

Universelles Hashing

Definition: Universelle Hashfamilie

Sei $\mathcal{H}$ eine Menge von Hashfunktionen, welche von U auf $\{0, .., \ m-1\}$ abbilden.
$\mathcal{H}$ heißt **universell**, wenn für ein zufällig gewähltes $h \in \mathcal{H}$ gilt:

$$\forall k, \ l \in U, \ k \neq l : \ \Pr[h(k) = h(l)] = \frac{1}{m}$$

Anders formuliert: Wählen wir eine Hashfunktion $h \in \mathcal{H}$ zufällig, ist die Kollisionswahrscheinlichkeit für zwei beliebige Schlüssel $\frac{1}{m}$.

Beispiel einer universellen Hashfamilie

Wenn $U \subseteq \mathbb{N}$:
$h_{a,b}(k) = ((ak + b) \ mod \ p)$ mit $a, b \in \mathbb{N}_0, \ a \neq 0$ und p ist eine Primzahl.

Roadmap

Union-Find Datenstruktur

Wir brauchen eine effiziente Methode, um herauszufinden, ob eine Kante verschiedene Teilbäume verbindet.

Union-Find Datenstruktur

Wir brauchen eine effiziente Methode, um herauszufinden, ob eine Kante verschiedene Teilbäume verbindet.

Eine endliche Menge $V = \{1 \dots n\}$ sei in disjunkte Klassen M_i partitioniert:

$$V = M_1 \cup \dots \cup M_k \text{ mit } M_i \cap M_j = \emptyset \; \forall i \neq j$$

Union-Find Datenstruktur

Wir brauchen eine effiziente Methode, um herauszufinden, ob eine Kante verschiedene Teilbäume verbindet.

Eine endliche Menge $V = \{1 \dots n\}$ sei in disjunkte Klassen M_i partitioniert:

$$V = M_1 \cup \dots \cup M_k \text{ mit } M_i \cap M_j = \emptyset \; \forall i \neq j$$

Jede Klasse M_i hat einen Repräsentanten $r_i \in M_i$.
Die Union-Find-Datenstruktur unterstützt

- **procedure** union$(i, j : V)$
  - vereinigt die beiden Klassen, zu denen i und j gehören.
- **function** find$(i : V) : V$
  - bestimmt zu i den Repräsentanten, zu dessen Klasse i gehört.

Am Anfang bildet jedes $v \in V$ eine eigene Klasse mit v als Repräsentant.

Union-Find - Erste Version

Grundidee: Baumdarstellung mit Repräsentant als Wurzel

Union-Find - Erste Version

Grundidee: Baumdarstellung mit Repräsentant als Wurzel

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V \ : \ v.\text{parent} := v$

Union-Find - Erste Version

Grundidee: Baumdarstellung mit Repräsentant als Wurzel

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V \ : \ v.\text{parent} := v$

FIND($i : V$) : V

  if $i.\text{parent} == i$ **then return** i

  else return FIND($i.\text{parent}$)

Union-Find - Erste Version

Grundidee: Baumdarstellung mit Repräsentant als Wurzel

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V$: v.parent $:= v$

FIND($i : V$) : V

  if i.parent $==$ i **then return** i

  else return FIND(i.parent)

procedure UNION($i, j : V$)

  $r_i :=$FIND(i)

  $r_j :=$FIND(j)

  if $r_i \neq r_j$ **then** r_i.parent $:= r_j$

Union-Find - Erste Version

Grundidee: Baumdarstellung mit Repräsentant als Wurzel

Class UnionFind($V = \{1 \dots n\}$)

$\forall v \in V \ : \ v.\text{parent} := v$

FIND($i : V$) : V

  if i.parent $==$ i **then return** i

  else return FIND(i.parent)

procedure UNION($i, j : V$)

  $r_i :=$FIND(i)

  $r_j :=$FIND(j)

  if $r_i \neq r_j$ **then** r_i.parent $:= r_j$

- find im schlechtesten Fall in $\Theta(n)$, weil der Baum unbalanciert wird (siehe Binärbäume)

Union-Find - Pfadkompresssion

Class UnionFind($V = \{1 \dots n\}$)

$\vdots$

FIND($i : V$) : V

if i.parent $==$ i **then return** i

else

$i' :=$ FIND(i.parent)

i.parent $:= i'$

return i'

$\vdots$

- find jetzt amortisiert in $\mathcal{O}(\log n)$

Union-Find - Union by Rank

- Grundidee: Bei der Operation union(i, j) wird der Baum, der kleiner ist, unter den größeren Baum gehängt.
- Damit verhindert man, dass einzelne Teilbäume zu Listen entarten können
- Die Tiefe eines Teilbaums T kann damit nicht größer als $\log |T|$ werden
- Das verringert die Worst-Case-Laufzeit von find auf $\mathcal{O}(\log n)$

Fragen?

Vielen Dank für Eure Aufmerksamkeit!
Bis nächste Woche :)

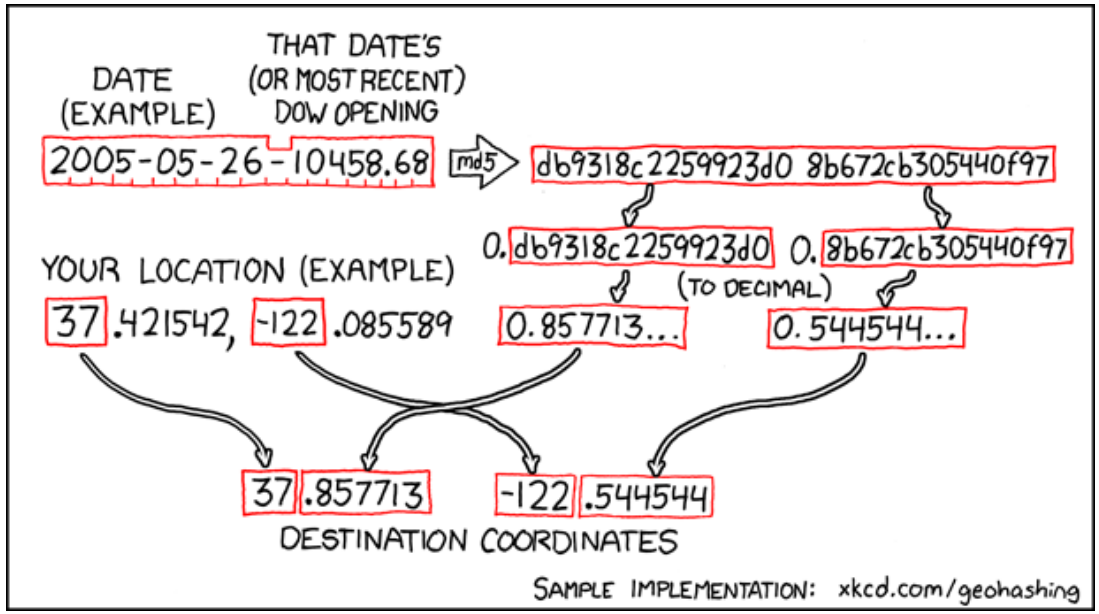


Abbildung: https://xkcd.com/426/