Universität Konstanz

Konzepte der Informatik

Informationscodierung und -speicherung II

Barbara Pampel

Universität Konstanz, WiSe 2023/2024

Datenstrukturen

Definition

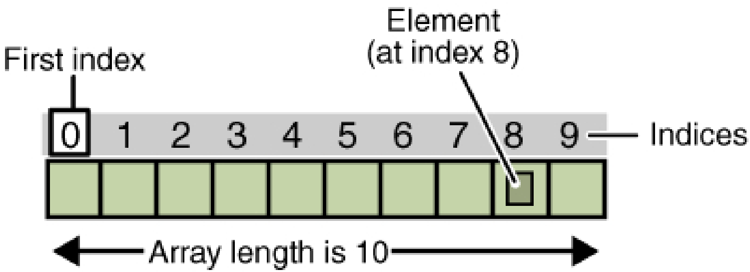
Datentyp zusammen mit Operationen auf diesen Daten, die Zugriff und Verwaltung ermöglichen und realisieren.

Hier: Datenstrukturen mit besonderer Relevanz für die Datenspeicherung

Arrays

Definition

Namentliche Zusammenfassung von gleichartigen Objekten eines Datentyps.



- Adressierung / Zugriff auf Elemente über Index
- in Java Indizes von 0 bis $n-1$ bei Länge n
- abstrakt auch oft von 1 bis n

Arrays

- Array mit Gleitkommezahlen in Java

```
double[] a = new double[12];
```

- in Java Indizes von 0 bis $n-1$ bei Länge n

```
int[] a = new int[5];
for (int i=1; i<=5; i++){
      a[i] = i;}
```

$\Rightarrow$ Exception zur Laufzeit, nicht zur Compilierzeit

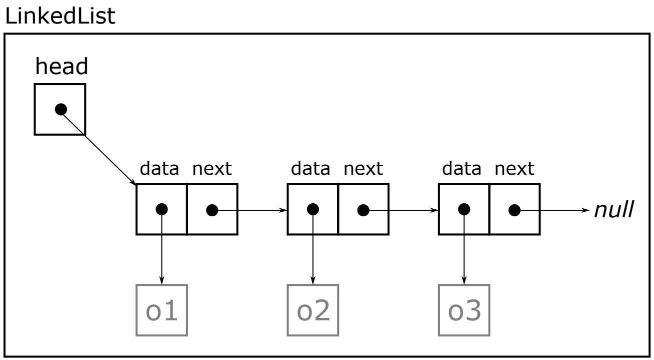
- Mehrdimensionale Arrays in Java, Achtung: ggf. riesigen Speicheranforderungen!

```
double[][][] a = new double[10][5][8];
```

Verkettete Listen

Für nicht aufeinanderfolgende Schlüssel

- Jedes Listenelement enthält einen Schlüssel UND
  einen Verweis auf den Nachfolger,
  bei *doppelter Verkettung* auch auf den Vorgänger



- mühsam zu durchlaufen
- dynamische veränderbar

Wörterbücher

- Verallgemeinerung von Schlüssel (auch Buchstaben, Wörter, Namen...)
- nicht notwendigerweise gleichmässig verteilt

Wörterbücher

- Verallgemeinerung von Schlüssel (auch Buchstaben, Wörter, Namen...)
- nicht notwendigerweise gleichmässig verteilt
- Nutzen eines kompletten Arrays wäre Speicherverschwendung

$\Rightarrow$ Arrays oder Listen mit Schlüssel als Einträgen

Wörterbücher

- Verallgemeinerung von Schlüssel (auch Buchstaben, Wörter, Namen...)
- nicht notwendigerweise gleichmässig verteilt
- Nutzen eines kompletten Arrays wäre Speicherverschwendung

$\Rightarrow$ Arrays oder Listen mit Schlüssel als Einträgen

- ohne weitere Bedingungenen: Sequenzielle Suche
- im schlimmsten Fall müssen ALLE n Einträge durchsucht werden

Geordnete Wörterbücher

Schlüssel suchen

- Schlüssel S suchen zwischen linker Grenze l und rechter Grenze r
- Betrachte Eintrag M in der Mitte $m = l + \lfloor \frac{r-l}{2} \rfloor$
- falls $M < S$: $l := m$
- falls $M = S$: ausgeben
- sonst $r := m$
- falls $l \neq r$: suche weiter zwischen l und r

- **Binäre Suche** im schlimmsten Fall müssen von n Einträgen
  Einträge betrachtet werden

Schlüssel suchen

- Schlüssel S suchen zwischen linker Grenze l und rechter Grenze r
- Betrachte Eintrag M in der Mitte $m = l + \lfloor \frac{r-l}{2} \rfloor$
- falls $M < S$: $l := m$
- falls $M = S$: ausgeben
- sonst $r := m$
- falls $l \neq r$: suche weiter zwischen l und r

- **Binäre Suche** im schlimmsten Fall müssen von n Einträgen $\lceil \log_2 n \rceil$ Einträge betrachtet werden
- Wenn implementiert als Array: Verschieben bei Einfügen von Schlüsseln
- Wenn implementiert als Liste: Kein direkter Zugriff auf Mitte

Tagesmenü

Binäre Suchbäume

Definition (Binärbaum)

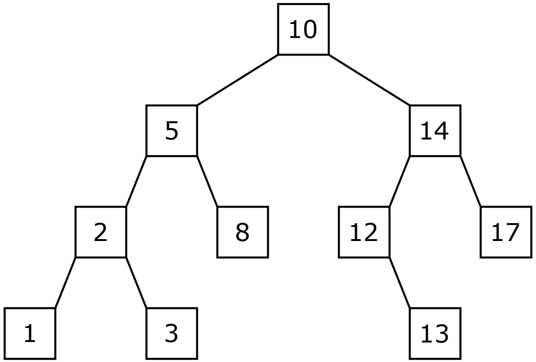
Ein *Binärbaum* ist entweder leer oder besteht aus einer Wurzel W mit linkem und rechten Kind K_l und K_r die jeweils selber Binärbäume sind.

Definition (Binärer Suchbaum)

Ein *binärer Suchbaum* ist ein Binärbaum mit *vergleichbaren* Schlüsseln. Dabei ist der Inhalt der Wurzel größer oder gleich allen Schlüsseln im linken Teilbaum und kleiner gleich allen Schlüsseln im rechten Suchbaum.

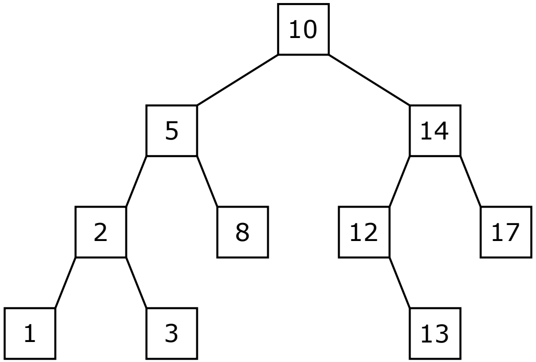
Annahme hier: Keine doppelten Schlüssel erlaubt!

Beispiel

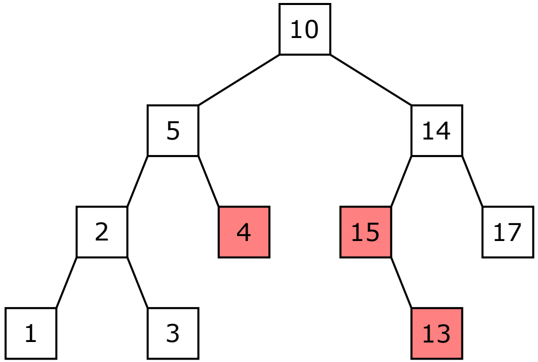


Binärer Suchbaum

Beispiel



Binärer Suchbaum

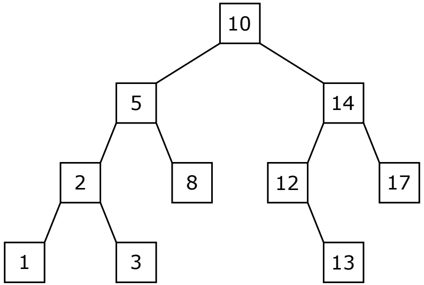


Kein binärer Suchbaum

Suche nach Schlüssel

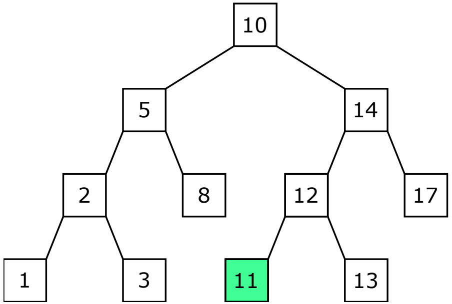
Suche nach Schlüssel

- Vergleich des gesuchten Schlüssel mit dem Wurzelknoten
- je nach Ergebnis Suche im linken oder rechten Teilbaum
  bis Schlüssel gefunden oder kein Kinder vorhanden
- Ausgabe des Wertes oder *key not found*



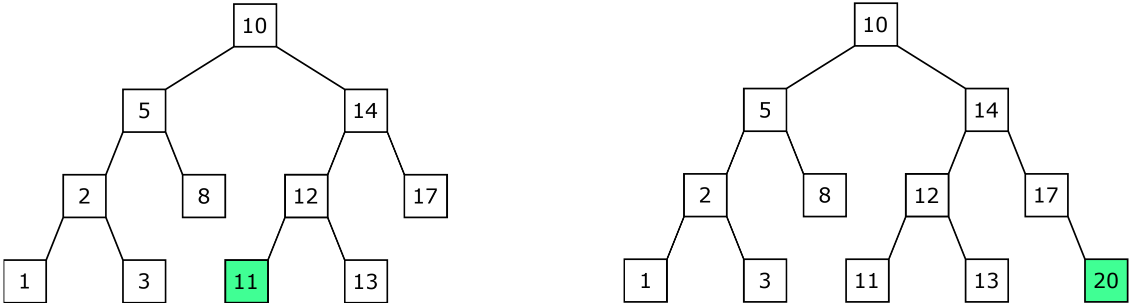
Schlüssel einfügen I

- Neue Schlüssel müssen an der richtigen Position eingefügt werden
- Einfacher Fall: Schlüssel ist noch nicht im Baum enthalten
  - nach Schlüssel suchen und an der erwarteten Position einfügen



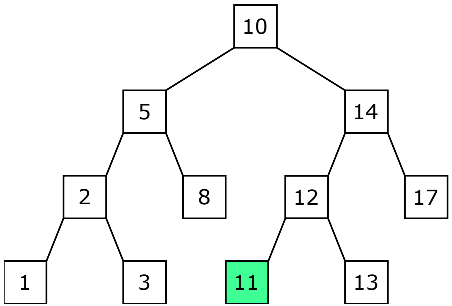
Schlüssel einfügen I

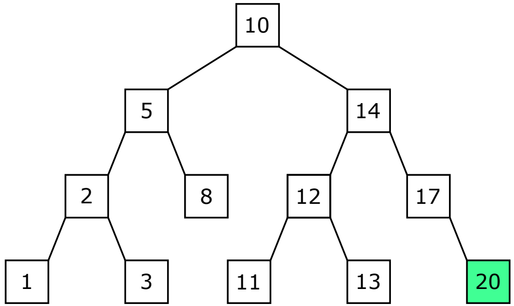
- Neue Schlüssel müssen an der richtigen Position eingefügt werden
- Einfacher Fall: Schlüssel ist noch nicht im Baum enthalten
  - nach Schlüssel suchen und an der erwarteten Position einfügen



- Spezialfall leerer Baum: Schlüssel wird Wurzel

Schlüssel einfügen II



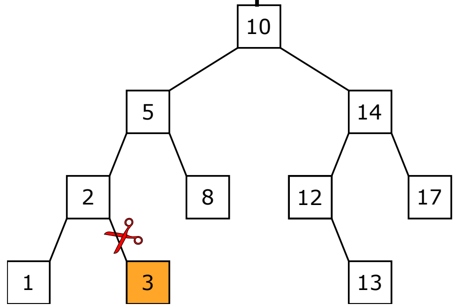


MERKE

- ein neuer Schlüssel wird auf diese Weise immer als BLATT eingefügt!
- links von ihm sind alle Schlüssel kleiner und rechts grösser!

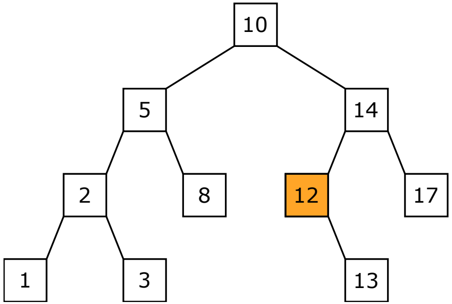
Schlüssel löschen I

- Drei Fälle
  - Knoten hat keine Kinder (Blattknoten)
  - Knoten hat ein Kind
  - Knoten hat zwei Kinder
- Blattknoten löschen ist einfach
  - Schlüssel suchen
  - Im Vorgänger Referenz auf das entsprechende Kind auf `null` setzen



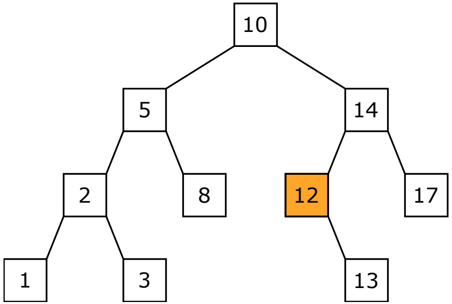
Schlüssel löschen II

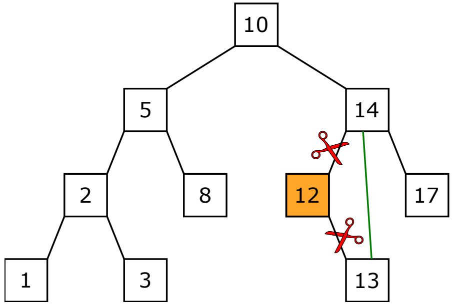
- Knoten mit einem Kind löschen
  - gelöschten Knoten durch das einzige Kind ersetzen



Schlüssel löschen II

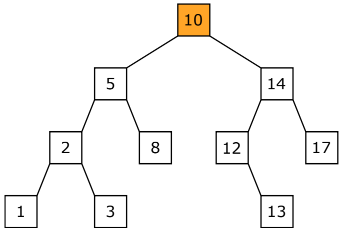
- Knoten mit einem Kind löschen
  - gelöschten Knoten durch das einzige Kind ersetzen





Schlüssel löschen III

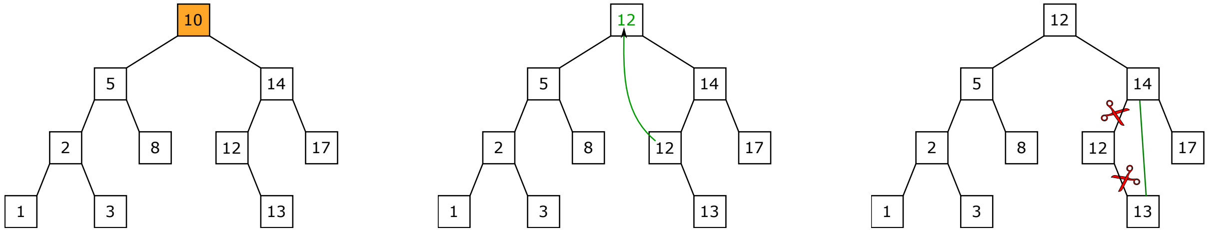
- Knoten mit zwei Kindern
    - Schlüssel ersetzen mit:



Schlüssel löschen III

- Knoten mit zwei Kindern
  - Schlüssel ersetzen mit:

    linkestem Kind im rechten Teilbaum oder
    rechtestem Konten im linken Teilbaum



Vorteile von binären Suchbäumen

- Suche deutlich schneller als in unsortierter Liste
    - im Idealfall bei balanciertem Binärbaum $\lceil \log_2 n \rceil$ Schritte bis zum Ziel
    - in sortierter und indizierter Liste mit binärer Suche auch $\lceil \log_2 n \rceil$

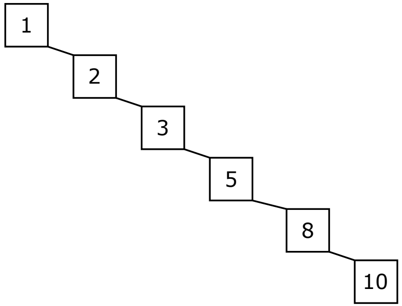
- Einfügen von Werten deutlich schneller als in sortierte Arrays
    - im Idealfall bei balanciertem Binärbaum $\lceil \log_2 n \rceil$ Schritte bis zum Ziel
    - im Array aber Verschieben der Werte „rechts" der Einfügeposition

Nachteile von binären Suchbäumen

- Binäre Suchbäume können entarten

Nachteile von binären Suchbäumen

- Binäre Suchbäume können entarten
  - ungeschickte Einfügereihenfolge
  - viele Lösch- und Einfügeoperationen



- in diesem Fall nicht besser als sequenzielle Suche

$\Rightarrow$ balancierte Bäume, B-Bäume... mehr dazu in AlgoDat!

Streuspeicherung

- 6-stellige Matrikelnummer, mit insgesamt $\approx 10^6$ möglichen Werten

Streuspeicherung

- 6-stellige Matrikelnummer, mit insgesamt $\approx 10^6$ möglichen Werten
- Allerdings werden nur etwa 10.000 gleichzeitig benutzt
  $\Rightarrow$ Abbildung aller möglichen Matrikelnummern auf ≈ 10.000 Plätze

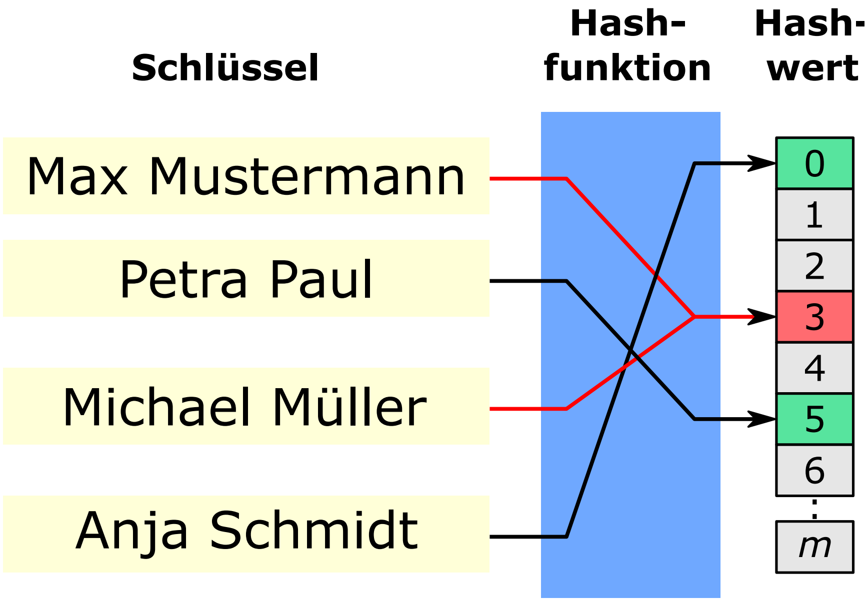
Streuspeicherung

- 6-stellige Matrikelnummer, mit insgesamt $\approx 10^6$ möglichen Werten
- Allerdings werden nur etwa 10.000 gleichzeitig benutzt
  $\Rightarrow$ Abbildung aller möglichen Matrikelnummern auf ≈ 10.000 Plätze
- Buchtitel in der Bibliothek mit ≈ 50 Zeichen $\Rightarrow 26^{50}$ mögliche Titel
- Die Bibliothek hat aber nur $\approx$ 2 Millionen Bücher
  $\Rightarrow$ Abbildung aller möglichen Titel auf $\approx$ 2 Millionen Plätze

Streuspeicherung

- Anwendung:
  - Assoziative Arrays
    z.B. in Python: dictionaries $(key, value)$ mit hash-tables
  - Datenbanken
  - Kryptographie
- Hashfunktion $h : \mathcal{K} \to \{0, 1, ..., m-1\}$
  - ordnet jedem Schlüssel k einen Index $0 \leq h(k) \leq m-1$ zu
- Problem von Hashkollisionen
  - da meist $n \gg m$ erhalten zwangsläufig unterschiedliche Schlüssel den gleichen Hashwert
  - erfordern ausgefeilte Sonderbehandlungen
  - abhängig vom Belegungsfaktor α – Verhältnis zwischen bereits belegten und vorhandenen Plätzen

Beispiele



Schlüssel
Hash-
funktion
Hash-
wert
Max Mustermann
Petra Paul
Michael Müller
Anja Schmidt
0
1
2
3
4
5
6
m

Hashfunktionen

- Zentrale Anforderungen
  - leicht und schnell berechenbar
  - möglichst gleichmäßige Aufteilung auf die vorhandenen Plätze zur Vermeidung von Kollisionen
  - deterministische Berechnung
  - (dynamisch) anpassbar an die Anzahl der freien Plätze
- Hashfunktionen basieren meistens auf positiven, ganzen Zahlen, also $\mathcal{K} \subseteq \mathbb{N}_0$
- Berechnung des Hashwertes von anderen Objekten (z.B. Zeichenketten) meistens direkt ohne Umweg in eine Zahlendarstellung

Die Divisions-Rest-Methode

- $h(k) = k \mod m$
  - ergibt automatisch Werte zwischen 0 und $m - 1$
- Ideale Hashfunktion, falls Werte aus $\mathcal{K}$ gleichverteilt
- Wahl eines passenden m ist wichtig zur Minimierung von Kollisionen bei ungleich verteilten Schlüsseln
  - beste Wahl sind Primzahlen
  - schlechte Wahl sind Potenzen der Basis der Zahlendarstellung
  - bei $m = 100$ bestimmen nur die untersten beiden Ziffern einer Zahl den Hashwert
  - bei $m = 32$ bestimmen nur die untersten fünf Bits einer Zahl den Hashwert

Beispiel

- Datumsangaben als Zahlen dargestellt, Tag zuletzt

Schlüssel k	k mod 97	k mod 100
20100101	52	1
20100227	81	27
20100427	87	27
20110527	2	27
20090811	74	11

Die multiplikative Methode

- Multiplikation des ganzzahligen Schlüssels mit einer irrationalen Zahl $\ominus$
- Abschneiden des ganzzahligen Teils
- Multiplikation mit m
- $h(k) = \lfloor m(k \cdot \ominus - \lfloor k \cdot \ominus \rfloor) \rfloor$
- Beispiel mit $\ominus = \pi$ und $m = 100$
  - $h(20) = \lfloor 100(20\pi - \lfloor 20\pi \rfloor) \rfloor = \lfloor 100(62,831853... - 62) \rfloor = 83$

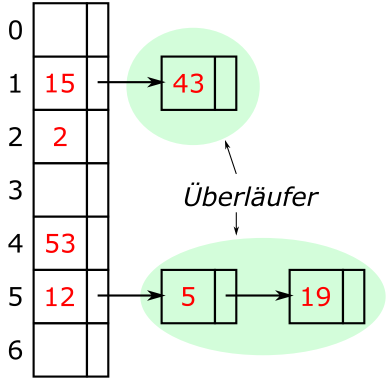
Schlüssel k	$h(k)$
20100101	63
20100227	47
20100427	79
20110527	88
20090811	24

- Beste Ergebnisse mit dem goldenen Schnitt $\phi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0,6180339887...$

Kollisionsbehandlung

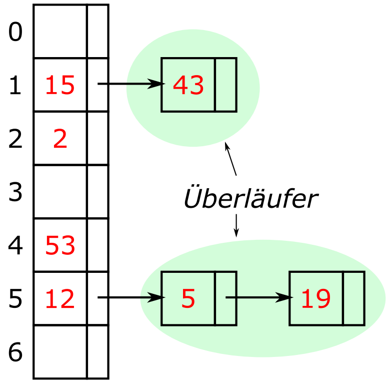
Kollisionsbehandlung durch Verkettung

- Eintrag in der Hashtabelle ist eine verkettete Liste
- Kollidierende Einträge werden an die Liste gehängt
- Beispiel: $m = 7$, $h(k) = k \mod 7$



Kollisionsbehandlung durch Verkettung

- Eintrag in der Hashtabelle ist eine verkettete Liste
- Kollidierende Einträge werden an die Liste gehängt
- Beispiel: $m = 7$, $h(k) = k \mod 7$



- Vorteil der Verkettung: Hashtabelle an sich muss nicht vergrößert werden

Operationen

- Suchen
  - Hashfunktion $h(k)$ berechnen
  - zugehörige Liste der Hashtabelle von vorne nach hinten durchsuchen
- Einfügen
  - Hashfunktion $h(k)$ berechnen
  - zugehörige Liste der Hashtabelle von vorne nach hinten durchsuchen und bei Bedarf neues Element hinten an Liste anhängen
- Löschen
  - Hashfunktion $h(k)$ berechnen
  - zugehörige Liste der Hashtabelle von vorne nach hinten durchsuchen und bei Bedarf Element aus Liste entfernen
- Aufwand

Operationen

- Suchen
  - Hashfunktion $h(k)$ berechnen
  - zugehörige Liste der Hashtabelle von vorne nach hinten durchsuchen
- Einfügen
  - Hashfunktion $h(k)$ berechnen
  - zugehörige Liste der Hashtabelle von vorne nach hinten durchsuchen und bei Bedarf neues Element hinten an Liste anhängen
- Löschen
  - Hashfunktion $h(k)$ berechnen
  - zugehörige Liste der Hashtabelle von vorne nach hinten durchsuchen und bei Bedarf Element aus Liste entfernen
- Aufwand
  - im günstigsten Fall eine Operation
  - im schlimmsten Fall n Operationen

Offene Hashverfahren

- Speichern von kollidierenden Einträgen in freien Plätzen in der Tabelle
- Knackpunkt ist das Vorgehen zum Finden eines freien Platzes

Generelles Vorgehen

- $h(k)$ sei eine gegebene Hashfunktion und $s(j, k)$ definiert eine Sondierungsfolge
- Dann ist

$$(h(k) + s(j, k)) \mod m$$

für $j = 0, 1, \ldots, m - 1$ eine offene Hashfunktion

Offene Hashverfahren

- Speichern von kollidierenden Einträgen in freien Plätzen in der Tabelle
- Knackpunkt ist das Vorgehen zum Finden eines freien Platzes

Generelles Vorgehen

- $h(k)$ sei eine gegebene Hashfunktion und $s(j,k)$ definiert eine Sondierungsfolge
- Dann ist
$$(h(k) + s(j,k)) \mod m$$
für $j = 0, 1, ..., m-1$ eine offene Hashfunktion

Sondierungfolgen z.B.

- lineares Sondieren:
$s(j) = j * c$ für Konstante $c \neq 0$
- quadratisches Sondieren:
$s(j) = j^2 * c_1 + j * c_2$ für Konstanten $c_1 \neq 0 und c_2$
- Doppelhashing:
$s(j,k) = j * h'(k)$ für zweite Hashfunktion $h'(k) : \mathcal{U} \to \{1, \ldots, m-1\}$

Operationen

- Einfügen
  - Hashfunktion $h(k)$ berechnen
  - an Stelle suchen und falls nicht frei:
    Sondierungsfolge durchlaufen bis freie Stelle gefunden
- Suchen
  - Hashfunktion $h(k)$ berechnen
  - an Stelle suchen und falls nicht der richtige Schlüssel gefunden:
    Sondierungsfolge durchlaufen bis Schlüssel gefunden
- Löschen
  - Schlüssel suchen und löschen, wenn gefunden
  - Löschmarkierung einfügen
- Aufwand

Operationen

- Einfügen
  - Hashfunktion $h(k)$ berechnen
  - an Stelle suchen und falls nicht frei:
    Sondierungsfolge durchlaufen bis freie Stelle gefunden
- Suchen
  - Hashfunktion $h(k)$ berechnen
  - an Stelle suchen und falls nicht der richtige Schlüssel gefunden:
    Sondierungsfolge durchlaufen bis Schlüssel gefunden
- Löschen
  - Schlüssel suchen und löschen, wenn gefunden
  - Löschmarkierung einfügen
- Aufwand
  - im günstigsten Fall eine Operation
  - im schlimmsten Fall m Operationen

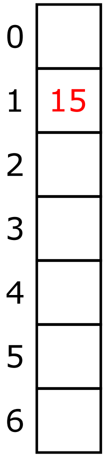
Löschmarkierungen

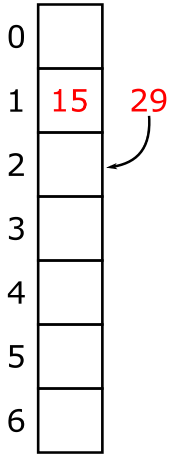
- Erzeugen eines speziellen Objekts zur Kennzeichnung von gelöschten Einträgen

0	
1	15
2	
3	
4	
5	
6	

Löschmarkierungen

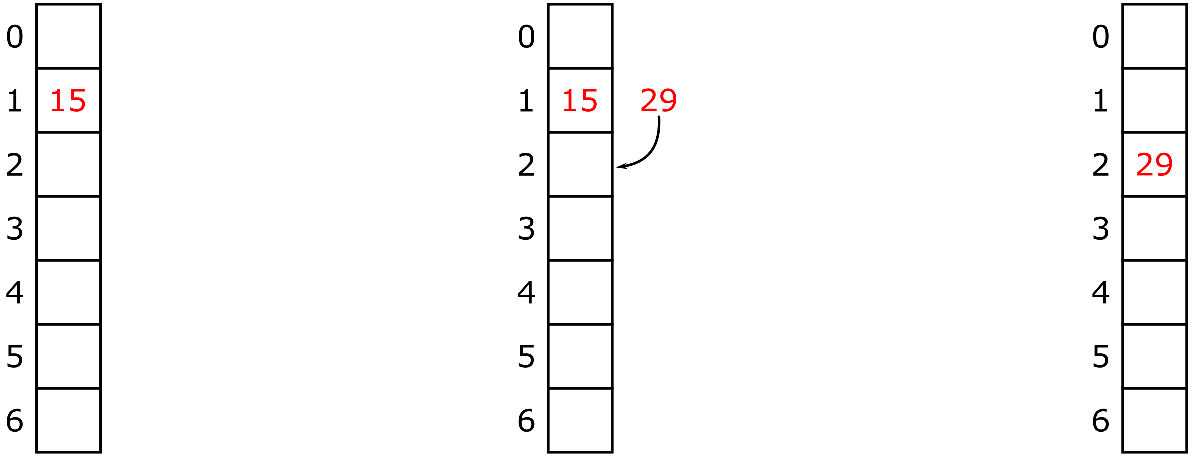
- Erzeugen eines speziellen Objekts zur Kennzeichnung von gelöschten Einträgen





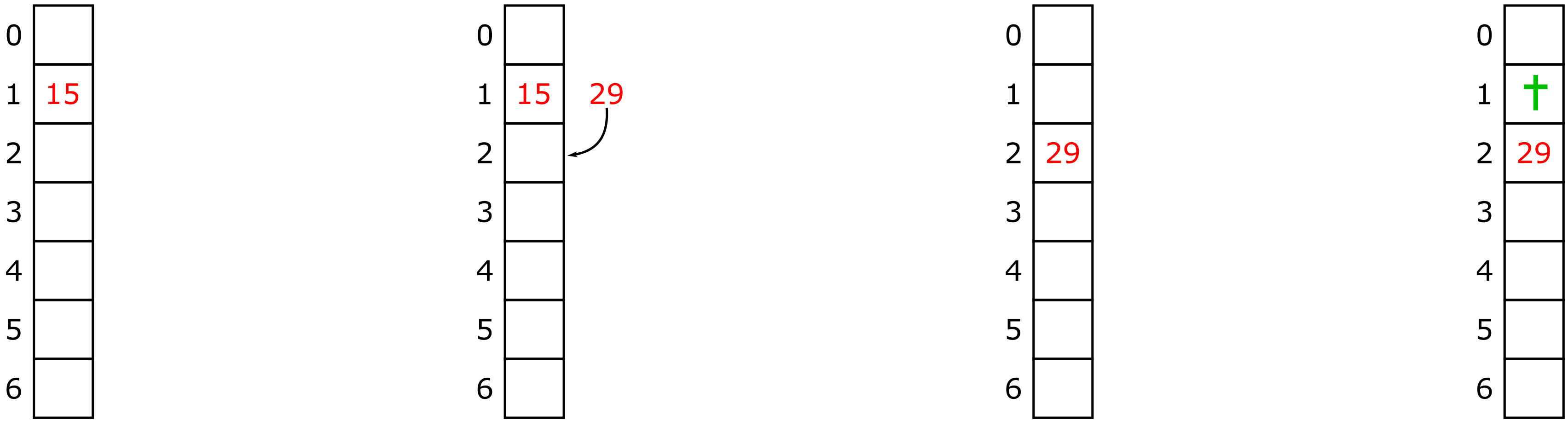
Löschmarkierungen

- Erzeugen eines speziellen Objekts zur Kennzeichnung von gelöschten Einträgen



Löschmarkierungen

- Erzeugen eines speziellen Objekts zur Kennzeichnung von gelöschten Einträgen



Generelle Probleme

- Verkettung
  - Überlauf möglich, aber Effizienz nimmt ab
- Offene Adressierung
  - Clusterbildung
  - es können nur maximal m Schlüssel in der Tabelle gespeichert werden
  - bei Vergrößerung ist komplette Neuberechnung der Tabelle notwendig
- Alternative: Besserer Hashfunktionen

Universelle Hashfunktionen - als Ausblick

- Idee: Menge von Hashfunktionen $\mathcal{H}$, aus der zufällig eine ausgewählt wird
  - Wahrscheinlichkeit, dass die „schlechteste” für die aktuellen Daten ausgewählt wurde ist gering

Definition (Universelle Hashfunktionen)

Eine endlich Menge $\mathcal{H}$ von Hashfunktionen heißt *universell*, wenn für je zwei verschiedene Schlüssel x, $y \in \mathcal{K}$ gilt

$$\frac{|\{h \in \mathcal{H} : h(x) = h(y)\}|}{|\mathcal{H}|} \leq \frac{1}{m}$$

Das wichtigste in Kürze

- Arrays für ganzzahlige Schlüssel
- Listen aus verketteten Objekten
- (geordnete) Wörterbücher für vergleichbare Schlüssel
- binäre Suchbäume
- Zugriff auf Schlüssel direkt durch Hashfunktion
    - Hashfunktion muss möglichst kollisionsfrei sein
    - Kollisionsbehandlung:
        - durch Verkettung
        - durch offene Adressierung (linear, quadratisch, Doppelhashing)
    - universelle Hashfunktionen

Ausblick

mehr Datenstrukturen im Algorithmik-Kapitel:

- Daten zweckdienlich angeordnet, kodiert und miteinander verknüpft
- Verwaltung von bzw. dem Zugriff auf die Daten in geeigneter Weise

Literatur

U. Brandes
Algorithmen und Datenstrukturen
Skript zur Vorlesung im WS 14/15

T. Ottmann und P. Widmayer.
Algorithmen und Datenstrukturen — Kapitel 4.1, 4.2, 4.3.
Spektrum Akademischer Verlag, 4. Ausgabe, 2002, ISBN 978-3-8274-1029-0.

Robert Sedgewick.
Algorithms in Java – Parts 1-4 – Kapitel 14.
Addison-Wesley Longman, Amsterdam, 3. Auflage, 2003, ISBN 0-210-36120-5.

Bildquellen

Foto Russischer Großbrief:
`https://unicodebook.readthedocs.io/_images/Letter_to_Russia_with_krakozyabry.jpg`
zuletzt geöffnet am 10. November 2020