Informatik II - SS 2018 (Algorithmen & Datenstrukturen)

Vorlesung 8 (14.5.2018)

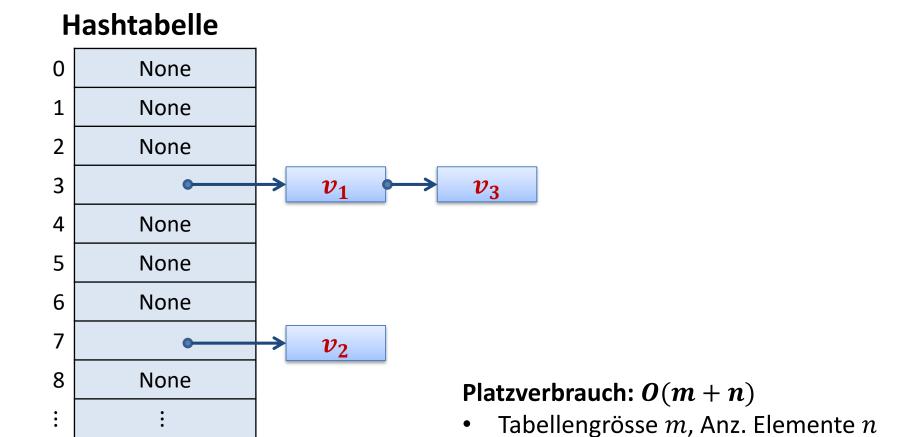
Hashtabellen III



Fabian Kuhn
Algorithmen und Komplexität

None

Jede Stelle in der Hashtabelle zeigt auf eine verkette Liste



Hashing mit offener Adressierung



Ziel:

- Speichere alles direkt in der Hashtabelle (im Array)
- offene Adressierung = geschlossenes Hashing
- keine Listen

Grundidee:

- Bei Kollisionen müssen alternative Einträge zur Verfügung stehen
- Erweitere Hashfunktion zu

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

- Für jedes $x \in S$ sollte h(x, i) durch alle m Werte gehen (für versch. i)
- Zugriff (schreiben/lesen) zu Element mit Schlüssel x:
 - Versuche der Reihe nach an den Positionen

$$h(x,0), h(x,1), h(x,2), ..., h(x,m-1)$$

Doppel-Hashing

UNI FREIBURG

Ziel: Verwende mehr als m verschiedene Abfolgen von Positionen

Idee: Benutze zwei Hashfunktionen

$$h(x,i) = (h_1(x) + i \cdot h_2(x)) \mod m$$

Vorteile:

- Sondierungsfunktion hängt in zwei Arten von x ab
- Vermeidet die Nachteile von linearem und quadr. Sondieren
- Wahrscheinlichkeit, dass zwei Schlüssel x und x' die gleiche Positionsfolge erzeugen:

$$h_1(x) = h_1(x') \land h_2(x) = h_2(x') \implies WSK = \frac{1}{m^2}$$

Funktioniert in der Praxis sehr gut!

Offene Adressierung:

- Alle Schlüssel/Werte werden direkt im Array gespeichert
- Keine Listen nötig
 - spart den dazugehörigen Overhead...
- Nur schnell, solange der Load

$$\alpha = \frac{n}{m}$$
Gosse Tabelle

nicht zu gross wird...

- dann ist's dafür besser als Chaining...
- $\alpha > 1$ ist nicht möglich!
 - da nur m Positionen zur Verfügung stehen

Rehash



Was tun, wenn die Hashtabelle zu voll wird?

- Offene Adressierung: $\alpha > 1$ nicht möglich, bei $\alpha \to 1$ sehr ineff.
- Chaining: Komplexität wächst linear mit α

Was tun, wenn die gewählte Hashfunktion schlecht ist?

Rehash:

- Erstelle neue, grössere Hashtabelle, wähle neue Hashfunktion h^\prime
- Füge alle Schlüssel/Werte neu ein

Beispiel: $X = \{5, 10, 11, 22\}, \ h(x) = x \mod 4, h'(x) = 3x - 1 \mod 8$

Kosten Rehash

UNI FREIBURG

Ein Rehash ist teuer!

W > M

Kosten (Zeit):

- $\Theta(m+n)$: linear in der Anzahl eingefügten Elemente und der Länge der alten Hashtabelle
 - typischerweise ist das einfach $\Theta(n)$
- Wenn man es richtig macht, ist ein Rehash selten nötig.
- richtig heisst:
 - gute Hashfunktion (z.B. aus einer universellen Klasse)
 - gute Wahl der Tabellengrössen: bei jedem **Rehash** sollte die **Tabellengrösse** etwa **verdoppelt** werden alte Grösse $m \implies$ neue Grösse $\approx 2m$
 - Verdoppeln ergibt immer noch durchschnittlich konstante Zeit pro Hashtabellen-Operation
 - → amortisierte Analyse (werden wir gleich anschauen...)

Kosten Rehash



Analyse Verdoppelungsstrategie

- Wir machen ein paar vereinfachende Annahmen:
 - Bis zu Load α_0 (z.B. $\alpha_0 = 1/2$) kosten alle Hashtabellen-Operationen $\leq c$
 - Bei Load α_0 wird die Tabellengrösse verdoppelt: Alte Grösse m, neue Grösse 2m, Kosten $\leq c \cdot m$
 - Am Anfang hat die Tabelle Grösse $m_0 \in O(1)$
 - Die Tabelle wird nie verkleinert...
- Wie gross sind die Kosten für das Rehashing, verglichen mit den Gesamtkosten für alle anderen Operationen?

übrige Kosten "auntisteren" die Rehash-Kosten

UNI

Gesamtkosten

- Wir nehmen an, dass die Tabellengrösse $\underline{m} = \underline{m_0 \cdot 2^k}$ für $k \geq 1$ ist
 - d.h., bis jetzt haben wir $k \ge 1$ Rehash-Schritte gemacht
 - Bemerkung: Bei k=0 sind die Rehash-Kosten 0.
- Die Gesamt-Rehash-Kosten sind dann

$$\leq \sum_{i=0}^{k-1} \underline{c \cdot m_0 \cdot 2^i} = c \cdot m_0 \cdot (2^k - 1) \leq \underline{c \cdot m}$$

Kosten Rehash

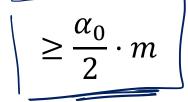


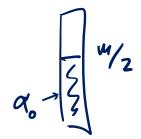
Gesamtkosten

- Wir nehmen an, dass die Tabellengrösse $m=m_0\cdot 2^k$ für $k\geq 1$ ist
 - d.h., bis jetzt haben wir $k \ge 1$ Rehash-Schritte gemacht
 - Bemerkung: Bei k=0 sind die Rehash-Kosten 0.
- Die Gesamt-Rehash-Kosten sind dann

$$\leq \sum_{i=0}^{k-1} c \cdot m_0 \cdot 2^i = c \cdot m_0 \cdot (2^k - 1) \leq c \cdot m$$

- Gesamt-Kosten für die übrigen Operationen
 - Beim Rehash von Grösse $\frac{m}{2}$ auf m waren $\geq \alpha_0 \cdot \frac{m}{2}$ Einträge in der Tabelle
 - Anzahl Hashtabellen-Operationen (ohne Rehash)





Kosten Rehash



Die Gesamt-Rehash-Kosten sind dann

$$\leq \sum_{i=0}^{k-1} c \cdot m_0 \cdot 2^i = c \cdot m_0 \cdot \left(2^k - 1\right) \leq \underline{c \cdot m}$$

Anzahl Hashtabellen-Operationen

$$\#\mathsf{OP} \ge \frac{\alpha_0}{2} \cdot m$$

Durchschnittskosten pro Operation Gesamblesten
$$\frac{\# OP \cdot c + Rehash_Kosten}{\# OP} \leq c + \frac{2c}{\alpha_0} \in O(1)$$

- Im Durschnitt sind die Kosten pro Operation konstant
 - auch für worst-case Eingaben (solange die Annahmen zutreffen)
 - **Durschnittskosten pro Operation = amortisierte Kosten der Operation**

Amortisierte Analyse



12

Algorithmenanalyse bisher:

worst case, best case, average case

Jetzt zusätzlich amortized worst case:

- n Operationen o_1, \ldots, o_n auf einer Datenstruktur, t_i : Kosten von o_i
- Kosten können sehr unterschiedlich sein (z.B. $t_i \in [1, c \cdot i]$)
- Amortisierte Kosten pro Operation

$$T$$
 wobei $T = \sum_{i=1}^{n} t_i$

- Amortisierte Kosten: Durchschnittskosten pro Operation bei einer worst-case Ausführung
 - amortized worst case ≠ average case!!
- Mehr dazu in der Algorithmentheorie-Vorlesung (und evtl. später)

- Falls man immer nur vergrössert und davon ausgeht, dass bei kleinem Load, Hashtabellenop. O(1) Kosten haben, sind die amortisierten Kosten pro Operation O(1).
- Analyse funktioniert auch bei zufälliger Hashfunktion aus universeller Familie (mit hoher Wahrscheinlichkeit)
 - dann haben Hashtabellen-Op. bei kleinem Load mit hoher Wahrscheinlichkeit amortisierte Kosten O(1)
- Die Analyse lässt sich auch auf Rehashs zum Verkleinern erweitern
- In einer ähnlichen Art kann man aus fixed-size Arrays dynamische Arrays bauen
 - Alle Arrayoperationen haben dann O(1) amortisierte Laufzeit
 - Vergrössern/verkleinern erlaubt der ADT nur in 1-Elem.-Schritten am Ende!
 - Werden wir vielleicht noch genauer anschauen...

Hashing Zusammenfassung:

- effiziente Dictionary-Datenstruktur
- Operationen brauchen im Erwartungswert (meistens) O(1) Zeit
- Bei Hashing mit Chaining hat insert immer O(1) Laufzeit
- Können wir auch bei find O(1) Laufzeit garantieren?
 - wenn gleichzeitig insert nur noch im Erwartungswert O(1) ist...

Cuckoo Hashing Idee:

- Offene Adressierung
 - an jeder Position der Tabelle hat es nur für ein Element Platz
- Zwei Hashfunktionen h_1 und h_2
- Ein Schlüssel x wird immer bei $h_1(x)$ oder $h_2(x)$ gespeichert
 - Falls beim Einfügen beide Stellen schon besetzt sind, müssen wir umorganisieren...

Cuckoo Hashing

FREIBURG

Enfügen eines Schlüssels x:

- x wird immer an der Stelle $h_1(x)$ eingefügt
- Falls schon ein anderer Schlüssel y an der Stelle $h_1(x)$ ist:
 - Werfe y da raus (daher der Name: Cuckoo Hashing)
 - -y muss an seiner alternativen Stelle eingefügt werden (falls es bei $h_1(y)$ war, an Stelle $h_2(y)$, sonst an Stelle $h_1(y)$)
 - falls da auch schon ein Element z ist, werfe z raus und platziere es an seiner
 Alternativposition
 - und so weiter...

Find / Delete:

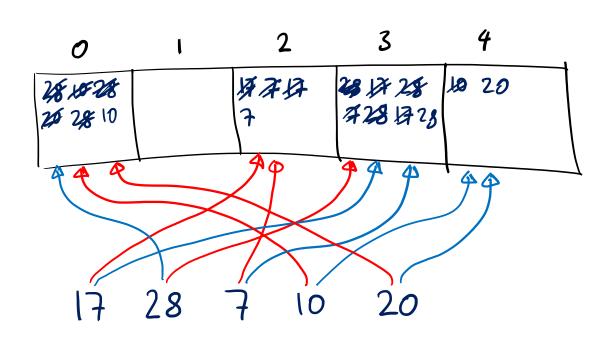
- Falls x in der Tabelle ist, ist's an Stelle $h_1(x)$ oder $h_2(x)$
- bei Delete: Markiere Zelle als leer!
- beide Operationen immer O(1) Zeit!

Cuckoo Hashing Beispiel

Tabellengrösse m=5

Hashfunktionen $h_1(x) = x \mod 5$, $h_2(x) \equiv 2x - 1 \mod 5$

Füge Schlüssel 17, 28, 7, 10, 20 ein:



Cuckoo Hashing: Zyklen



17

- Beim Einfügen kann es zu einem Zyklus kommen
 - -x wirft y_1 raus
 - $-y_1$ wirft y_2 raus
 - $-y_2$ wirft y_3 raus
 - **–** ...
 - $-y_{\ell-1}$ wirft y_{ℓ} raus
 - $-y_{\ell}$ wirft x raus
- Dann wird noch der alternative Platz f
 ür x ausprobiert, aber da kann das Gleiche auch wieder passieren...
- Tritt insbesondere auf, falls $h_1(y_i) = h_2(y_i)$
- In dem Fall wählt man neue Hash-Funktionen und macht einen Rehash (normalerweise mit grösserer Tabelle)

Cuckoo Hashing: Hashfunktionen



Wie wählt man die zwei Hashfunktionen?

- Sie sollten möglichst "unabhängig" sein...
- Wenige Schlüssel x, für welche $h_1(x) = h_2(x)$
- Eine gute Möglichkeit:

Zwei unabhängige, zufällige Funktionen einer universellen Menge

- Dann kann man zeigen, dass Zyklen nur sehr selten vorkommen, solange $n \leq m/2$
- Sobald die Tabelle halbvoll ist $(n \ge m/2)$ sollte man daher einen Rehash machen und zu einer doppelt so grossen Tabelle wechseln

Find / Delete:

- Hat immer Laufzeit O(1)
- Man muss nur die zwei Stellen $h_1(x)$ und $h_2(x)$ anschauen
- Das ist der grosse Vorteil von Cuckoo Hashing

Insert:

- Man kann zeigen, dass das im Durchschnitt auch Zeit O(1) braucht
- Falls man die Tabelle nicht mehr als zur Hälfte füllt
- Verdoppeln der Tabellengrösse bei Rehash ergibt konstante durchschnittliche Laufzeit für alle Operationen!

Hashing in Python

Hashtabellen (Dictionary):

https://docs.python.org/2/library/stdtypes.html#mapping-types-dict

```
neue Tabelle generieren: table = {}
```

• (key,value)-Paar einfügen: table.update({key: value})

• Suchen nach key: key in table $\rightarrow \neg / \mp$

table.get(key)

table.get(key, default_value)

• Löschen von *key*: del *table*[*key*]

table.pop(key, default_value)

Hashing in Java



21

Java-Klasse HashMap:

- Neue Hashtab. erzeugen (Schlüssel vom Typ K, Werte vom Typ V)
 HashMap<K,V> table = new HashMap<K,V>();
- Einfügen von (key,value)-Paar (key vom Typ K, value vom Typ V)
 table.put(key, value)
- Suchen nach key
 table.get(key)
 table.containsKey(key)
- Löschen von key table.remove(key)
- Ähnliche Klasse HashSet: verwaltet nur Menge von Schlüsseln

Es gibt nicht eine Standard-Klasse

hash_map:

Sollte bei fast allen C++-Compilern vorhanden sein

http://www.sgi.com/tech/stl/hash_map.html

unordered_map:

Seit C++11 in Standard STL

http://www.cplusplus.com/reference/unordered_map/unordered_map/

Hashing in C++



C++-Klassen hash_map / unordered_ map:

- Neue Hashtab. erzeugen (Schlüssel vom Typ K, Werte vom Typ V)
 unordered_map<K,V> table;
- Einfügen von (key,value)-Paar (key vom Typ K, value vom Typ V)
 table.insert(key, value)
- Suchen nach key
 table[key] oder table.at(key)
 table.count(key) > 0
- Löschen von key table.erase(key)

Hashing in C++



Achtung

- Man kann eine hash_map / unordered_map in C++ wie ein Array benutzen
 - die Array-Elemente sind die Schlüssel

Aber:

T[key] fügt den Schlüssel key ein, falls er noch nicht drin ist

T.at(key) wirft eine Exception falls key nicht in der Map ist

Ziel: Ein verteilter Dictionary

- Verwalte (*key, value*)-Paare in einem Netzwerk
 - z.B. auf vielen Rechnern im Internet
- Jeder Rechner soll einen Teil der Daten speichern
- insert / find
- Daten sollen schnell zugreifbar sein (übl. Dictionary-Operationen)
- Da die Anzahl Rechner gross sein kann, soll jeder Rechner im Netzwerk nur wenige andere "kennen" müssen...
 - Eine Tabelle mit allen Rechnern ist nicht machbar
 - Einen zentralen Server mit allen Informationen wollen wir auch nicht...
- Typische Anwendung: Peer-to-Peer Netzwerke
- Wir schauen uns eine von vielen ähnlichen Lösungen an...
 - im Wesentlichen Chord...

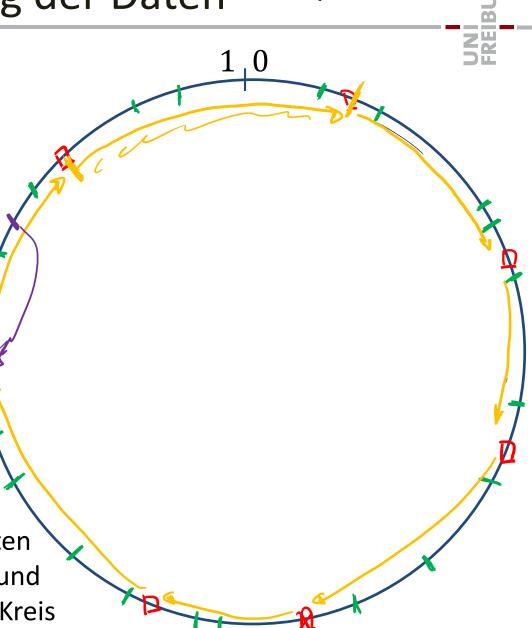
Verteilte Speicherung der Daten

9/ ... , m-/

Hashfunktion:

- $h: S \rightarrow [0, 1]$
 - verstehe Intervall [0,1]als Einheitskreis
- Jeder Schlüssel wird auf den Einheitskreis gemappt

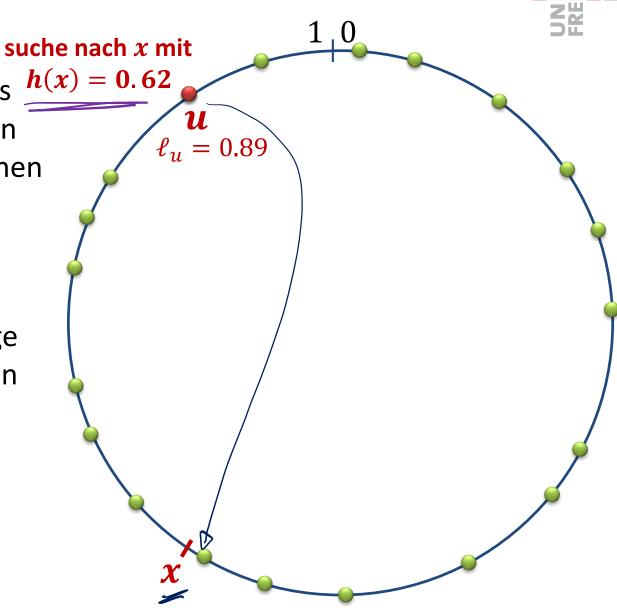
- Jeder Knoten u wählt einen zufälligen Wert $\ell_u \in [0, 1]$ (einen zufälligen Pkt. auf dem Einheitskreis)
- Ein Knoten u speichert die Daten zu den Schlüsseln zwischen u und seinem Nachfolger v auf dem Kreis



Idee:

• Suche ist einfach, falls u eine Tabelle mit den Adressen und Bereichen von allen Knoten zur Verfügung hat

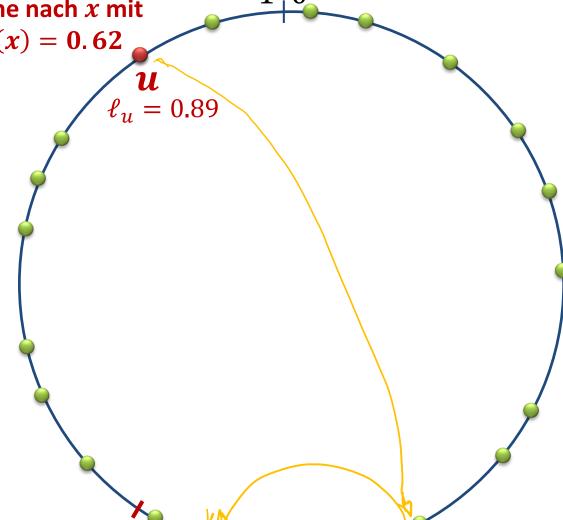
 u will aber nur wenige Adressen von anderen Knoten verwalten



Idee:

benutze binäre Suche! h(x) = 0.62





Topologie

BURG

• Jeder Knoten u hat eine direkte Verbindung zu den direkten Nachfolgern

 $l + \frac{1}{2}$

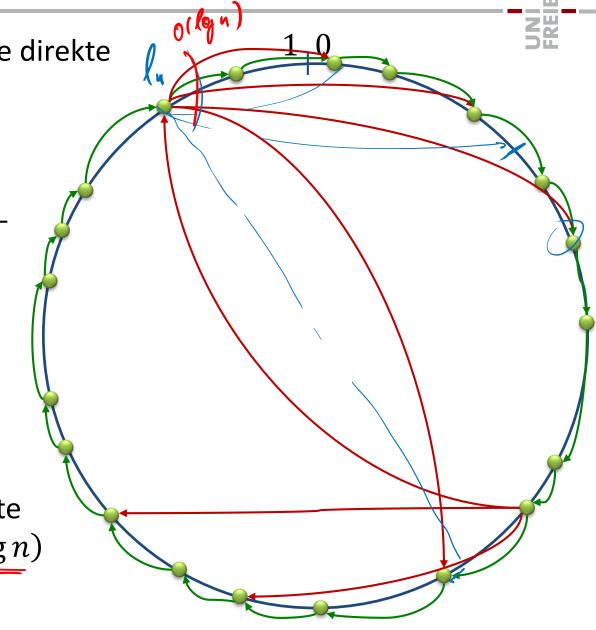
 Und zu den Nachfolger-Knoten der Werte

$$\ell_u + 2^{-i}$$

$$(i = 1, \dots, \log n)$$

- n: Anz. Knoten

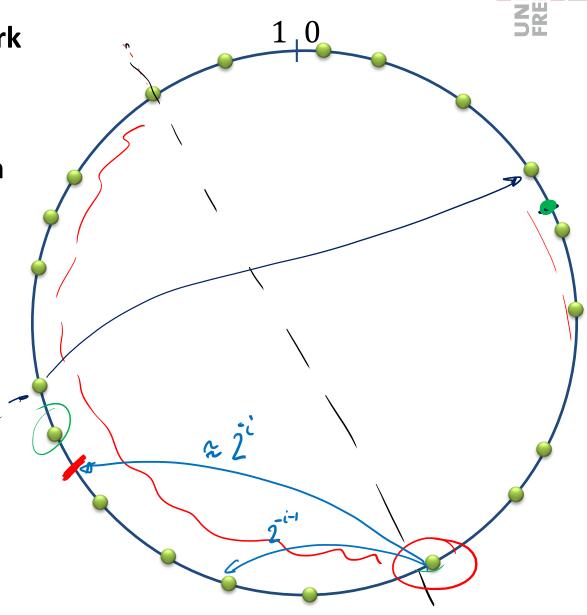
• Jeder Knoten hat direkte Verbindungen zu $O(\log n)$ anderen Knoten



Man kann in dem Netzwerk in $O(\log n)$ Zeit suchen:

Zeit = #besuchte Knoten

Man kann in jedem
 Schritt im Wesentlichen
 in die Mitte zwischen
 der aktuellen Position
 und dem Schlüssel x
 springen!



Zusammenfassung: Verteilte Hashtabelle



- Man geht davon aus, dass prinzipiell jeder mit jedem kommunizieren kann
 - Ist im Internet der Fall, IP-Adresse genügt, um Nachricht zu schicken
- Der durch die direkten Verbindungen induzierte Graph heisst auch Overlay Netzwerk
- Im Overlay Netzwerk hat jeder $O(\log n)$ Nachbarn
- Man kann den Algorithmus so implementieren, dass alle wichtigen Operationen $O(\log n)$ Laufzeit haben
 - Einfügen / löschen / suchen eines Schlüssels
 (Operation wird jeweils von irgend einem Knoten ausgeführt)
 - Einfügen / löschen eines Knotens

Zusätzliche Dictionary Operationen



Dictionary:

Zusätzliche mögliche Operationen:

D.minimum() : gibt kleinsten key in der Datenstruktur zurück

D.maximum() : gibt grössten key in der Datenstruktur zurück

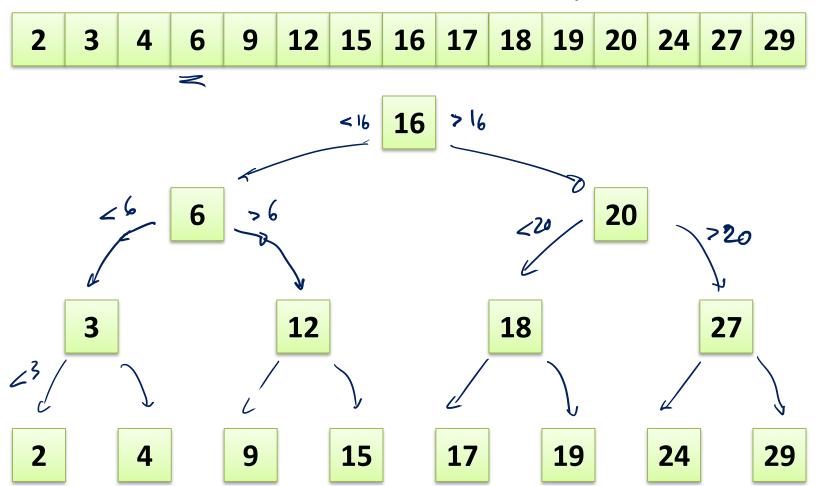
• D.successor(key) : gibt nächstgrösseren key zurück

• D.predecessor(key): gibt nächstkleineren key zurück

• D.getRange(k1, k2): gibt alle Einträge mit Schlüsseln im Intervall

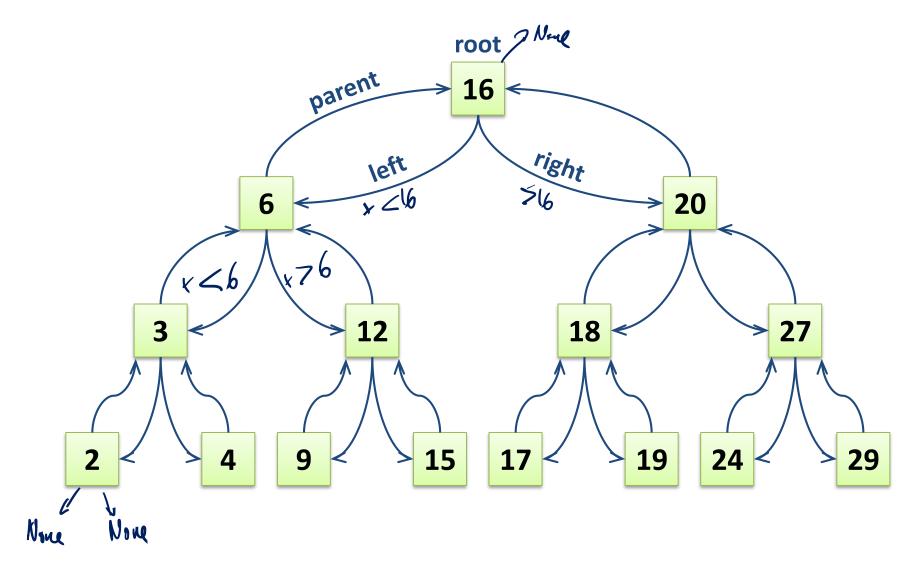
[k1,k2] zurück abc

• Binäre Suche nach x in einem sortierten Array...

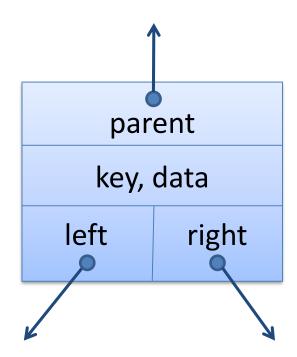


34

Benutze den Suchbaum der binären Suche als Datenstruktur



TreeElement:

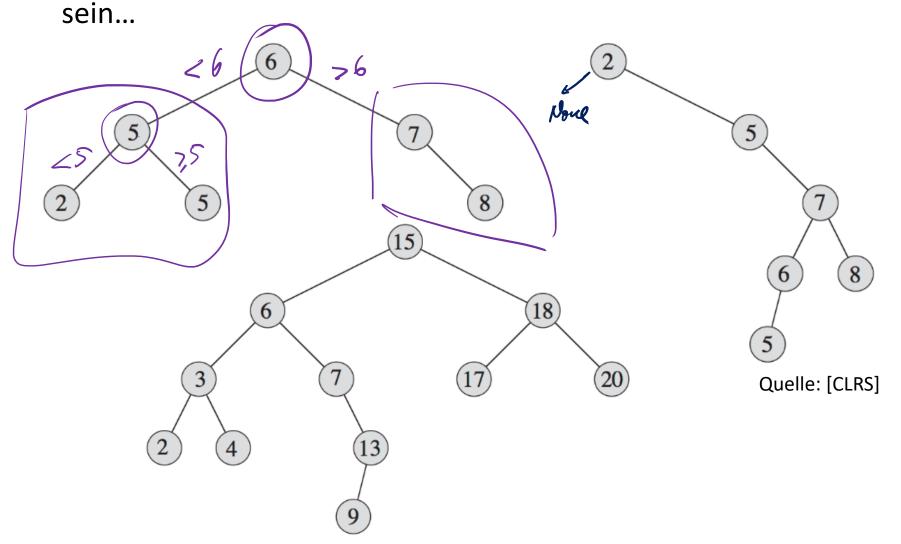


Implementierung: gleich wie bei den Listen-Elementen

Binäre Suchbäume



Binäre Suchbäume müssen nicht immer so schön symmetrisch



BURG

Suche nach Schlüssel x

 Benutze binäre Suche (darum heisst's binärer Suchbaum...)

```
3
7
117
20
20
9
```

```
current = root
```

while current is not None and current.key != x:

if current.key > x:

current = current.left

else:

current = current.right

current = None: x is will im Baum

current=x

Laufzeit der Suche in einem binären Suchbaum

