DHBW Lörrach
Duale Hochschule
Baden-Württemberg

Algorithmen und Komplexität
TIF 21A/B
Dr. Bruno Becker

7. Symbol- und Hashtabellen

Symbol- und Hashtabellen

- **Symboltabellen**
- Hashtabellen
- Hashverfahren

Symboltabellen

- Eine **Symboltabelle** ist eine Datenstruktur für Schlüssel-Wert-Paare, die folgende Operationen unterstützt:
  1. *Einfügen* eines neuen Paares in die Tabelle (*put*).
  2. *Suchen* nach dem Wert, der mit einem gegebenen Schlüssel verbunden ist (*get*).

  Die Schlüssel stammen aus einer (*großen) Menge möglicher Schlüssel*

Anwendungen für Symboltabellen

…sind sehr vielfältig!

Anwendung	Schlüssel	Wert
Wörterbuch	Wort	Übersetzung
Telefonbuch	Name	Telefonnummer
Web-Suche	Suchbegriff	Liste von URLs
Dateisystem	Dateiname	Speicherort
DNS	Domain-Name	IP-Adresse

Symboltabellen

- API

Public class SymbolTable <Key, Value >

SymbolTable() /* Symboltabelle anlegen

void put(Key key, Value value) /* *Lege Wert value unter Schlüssel key ab, überschreibe ggfs. vorh. Wert*

Value get (Key key). /* *Liefert den Wert zurück, der zu dem Schlüssel key gehört, sonst null*

void delete(Key key) /* *Entfernt den Schlüssel (und seinen Wert) aus der Tabelle*

boolean isEmpty() /* Ist Symboltabelle leer?

int size() /* Anzahl der Schlüssel-Werte Paare in der Tabelle

Unterschiede zu Arrays

- **Array**
  - Integer-Schlüssel 0,…,n-1
  - Feste Größe
  - Reserviert Speicherplatz für alle möglichen Schlüssel
- **Symboltabelle**
  - Schlüssel: Objekte aus beliebiger Klasse (evtl. ohne Totalordnung)
  - Variable Größe: Abhängig von Anzahl aktueller Einträge
  - Anzahl tatsächlicher Einträge i.a. << Anzahl möglicher Schlüssel
    - z.B. 2.000 Kunden mit je 10-stelliger Kundennummer

Listen-Implementierungen von Symboltabellen

- **Unsortierte Liste**
  - *get*
  - *put*
- **Sortierte Liste**
  - Voraussetzung Schlüssel mit Totalordnung
  - *get*
  - *put*

Suchbaum-Implementierungen von Symboltabellen

- **Einfache (*unbalancierte)* binäre Suchbäume**
  - Voraussetzung: Schlüssel mit Totalordnung
  - Keine O(log n) Garantie im worst case
  - Average Case-Analyse schwierig, wenn zahlreiche Löschungen
  - Speicher-Overhead für Baumstruktur
- **Balancierte binäre Suchbäume**
  - Voraussetzung: Schlüssel mit Totalordnung
  - *get, put* mit O (log n) Garantie
  - Komplizierte Implementierung

Implementierung von Symboltabellen

- **Ziele**
  - Datenstruktur mit wenig Speicher-Overhead
  - Einfache Implementierung
  - Totalordnung der Schlüssel nicht erforderlich
  - Worst case O(log n)
  - Average case O(1)

Idee: Nicht suchen – Speicherort aus Schlüsselwert *berechnen…*

Symbol- und Hashtabellen

- Symboltabellen
- **Hashtabellen**
- Hashverfahren

Hashtabellen

- **Hashtabelle**
  - Speicherung Schlüssel/Wert-Paare in Array *a* der Größe *m*
  - **Hashfunktion:** Schlüsselraum *K* → $\{0,\ldots,m-1\}$
  - Paar mit Schlüssel *k* wird unter Index *hash(k)* in Array *a* gespeichert

- **Fragen:**
  - Wahl der Array-Größe *m* ?
  - Welche Hash-Funktion ?
  - Was tun bei **Kollision**: 2 verschiedene Schlüssel *k* und *k`* mit *hash (k) = hash (k`)?*

Hashtabellen – Wahl der Tabellengröße

- **„Unbegrenzter Platz“**
  - Wähle $m = \| K \|$: Reserviert Speicher für *jeden möglichen* Schlüssel
  - Hashfunktion: *K* durchnummerieren, bijektive Abbildung
  - Kollisionsfrei
  - Zugriffszeit?
- **„Unbegrenzte Zeit“**
  - Wahl *m* = Anzahl der Einträge *n,* d.h. minimaler Speicher
  - Sequentielle Suche zur Kollisionsbehandlung, Hashfunktion unwichtig
  - Zugriffszeit?

Kompromiss zwischen Zeit- und Platz-Bedarf → Hashing

Hashfunktion

Wahl der Hashfunktion entscheidend für Performance vom Hashing

- **Anforderungen**
  - Hashfunktion schnell berechenbar
  - Alle Indexwerte erreichbar (d.h. *surjektive* Abbildung)
  - Schlüsselraum möglichst *gleichverteilt* auf Indexraum

Beispiele:

- Telefonnummern:
  - Führende Ziffern: Schlecht
  - Hintere Ziffern: Gut
- Datumswert:
  - Jahr oder Nummer vom Tag?

Hashfunktion

- **Modulo-Funktion** ist naheliegende Hashfunktion
  - *hash (k) = k mod m*
- Ggfs. vorgeschaltet eine Funktion, die einen Schlüssel *k* in eine ganzzahlige Zahl überführt.

Wahl von *m* auch wichtig:

- Primzahl ist günstig (kein gemeinsamer Teiler mit Schlüssel)
- *m* nicht zu klein, sonst sind Kollisionen sehr wahrscheinlich
  - Kollisionen immer möglich, sobald $\| K \| > m$
  - Geburtstags-Paradox - Ab 23 Personen im Raum haben wahrscheinlich 2 Personen am selben Tag Geburtstag ($n \geq 1{,}2 * \sqrt{m}$)

Symbol- und Hashtabellen

- Symboltabellen
- Hashtabellen
- **Hashverfahren**

Hashverfahren

- Hashfunktion und Größe *m* gegeben
- **Annahme:**
  - *Uniform Hashing-Annahme:* Für jeden betrachteten Schlüssel ist jeder der *m* möglichen Hashwerte gleich wahrscheinlich, unabhängig von den anderen betrachteten Schlüsseln

  → *n* Schlüssel auf *m* Positionen hashen:
  Auf jede Position entfallen im Erwartungswert $\alpha = n/m$ Schlüssel
  (α ist der Belegungsfaktor einer Hashtabelle der Größe *m*, die gerade *n* Elemente speichert)
- Wie soll die Kollisionsbehandlung stattfinden?

Offene Hashverfahren: Kollisionen werden innerhalb der Hashtabelle gespeichert

- Lineares Sondieren, Quadratisches Sondieren

(Halb-)Dynamische Hashverfahren:

- Hashverfahren mit Verkettung der Überläufer (in der Vorlesung)
- Dynamische Hashverfahren (s. Ottmann/Widmayer Kap 4.4.)

Lineares Sondieren

- **Verwendet array *a* mit m > n Elementen**
  - *h (k) = k mod m*
  - Bei Einfügen (*put*) von k: Sei *i = h(k)*
    - Falls *a[i]* besetzt, dann versuche a[i-1], a[i-2], …
    - *nach a[0]* versuche *a[m-1]*

Beispiel: *K= {0,1,…500}, h(k) = k mod 7: 53-12-5-15-2-19: h(12)=5; h(53)=4*

0	1	2	3	4	5	6
				53	12	

Einfügen von 5: *h(5) = 5: a[5] besetzt, a[4] besetzt => speichere in a[3]*

Danach Einfügen von 15,2,19 ergibt

0	1	2	3	4	5	6
19	15	2	5	53	12	

Lineares Sondieren - Aufwand

- Beispiel zeigt, dass man evtl. lange braucht, um Schlüssel zu finden (19 wird an Position 0 gespeichert, Hashwert ist aber 5, d.h. *get* sucht an Position 5,4,3,2,1,0)

Analyse: Für die durchschnittliche der bei erfolgloser bzw. erfolgreicher Anzahl betrachteten Einträge $C`_n$ bzw C_n beim Linearen Sondieren gilt (α Belegungsfaktor):

$$C`_n \approx \frac{1}{2} * (1 + \frac{1}{(1-\alpha)^2})$$

$$C_n \approx \frac{1}{2} * (1 + \frac{1}{(1-\alpha)})$$

Belegungsfaktor α	Erfolgreiche Suche	Erfolglose Suche
0,5	1,5	2,5
0,9	5,5	50,5
0,95	10,5	200,5

Quadratisches Sondieren

- **Verwendet array *a* mit m > n Elementen**
  - *h (k) = k mod m, m* Primzahl der Form *4i +3*
  - Bei Einfügen (*put)* von k: Sei *i = h(k)*
    - Falls *a[i]* besetzt, dann versuche a[i-1], a[i+1], a [h(k) -4], a [h(k) + 4], a [h(k) -9],…
    - Plätze um h(k) herum wachsen mit quadratischem Abstand

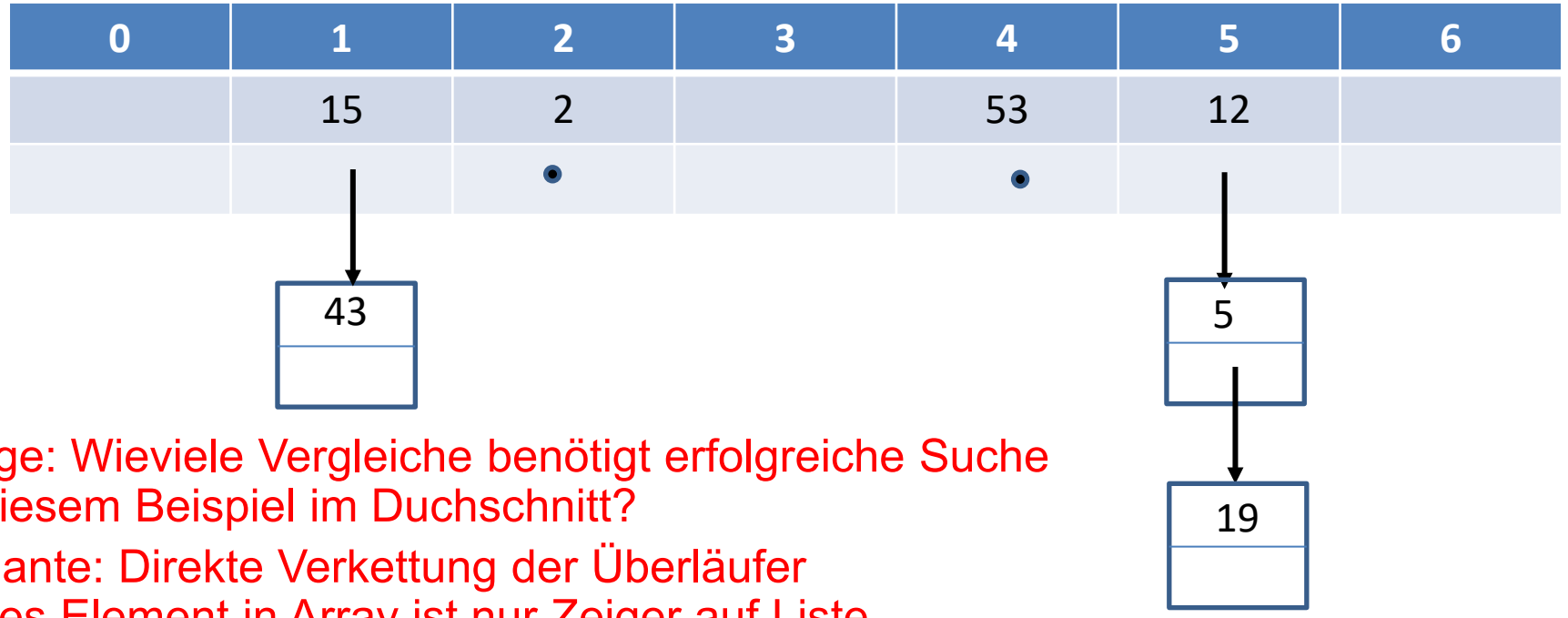
Analyse: Für $C`_n$ bzw C_n beim Quadratischem Sondieren gilt

$$C`_n \approx \frac{1}{(1-\alpha)} - \alpha + \ln\left(\frac{1}{(1-\alpha)}\right) \qquad C_n \approx 1 + \ln\left(\frac{1}{(1-\alpha)}\right) - \frac{\alpha}{2}$$

Belegungsfaktor α	Erfolgreiche Suche	Erfolglose Suche
0,5	1,44	2,5
0,9	2,85	11,4
0,95	3,52	22,05

Hashing mit Verkettung der Überläufer

- **Verwendet Array *a* mit m < n Elementen (separate Verkettung Überläufer)**
  - *put:* Eintrag am Anfang von Liste *i,* wenn noch nicht drin
  - *Get:* Lineare Suche in Liste *i*



- Frage: Wieviele Vergleiche benötigt erfolgreiche Suche in diesem Beispiel im Duchschnitt?
- Variante: Direkte Verkettung der Überläufer
  Jedes Element in Array ist nur Zeiger auf Liste

Hashing mit Verkettung Überläufer - Aufwand

Analyse: Für die durchschnittliche der bei erfolgloser bzw. erfolgreicher Anzahl betrachteten Einträge $C`_n$ bzw C_n bei separater Verkettung der Überläufer gilt (α Belegungsfaktor):

$$C`_n \approx \alpha + e^{-\alpha} \qquad C_n \approx 1 + \frac{\alpha}{2}$$

Belegungsfaktor α	Erfolgreiche Suche	Erfolglose Suche
0,5	1,25	1,11
0,9	1,45	1,31
0,95	1,48	1,34
1	1,5	1,37

- Typische Wahl in der Praxis $m \approx n/5$

Zusammenfassung

- Symboltabellen für Speicherung von Datensätzen mit Schlüsseln aus großem Datenraum
- Anstatt Suche nach Schlüssel *Berechnung* der Adresse aus Schlüsselwert
- Hash-Verfahren als Kompromiss zwischen Platz- und Zeitbedarf
- Offene Hash-Verfahren: Behandlung innerhalb Hash-Tabelle
  - Cluster vermeiden, Hash-Tabellen nicht zu voll, sonst Suche aufwändig
  - Löschen mit Clustern auch problematisch → ggfs. viele Adressen ändern
- „Halbdynamisch“: Hash-Verfahren mit Verkettung Überläufer
  - Stabiler bei größerem Füllgrad als offene Hashverfahren
  - Benötigt zusätzlichen Platz für Überläufer