Algorithmen & Datenstrukturen

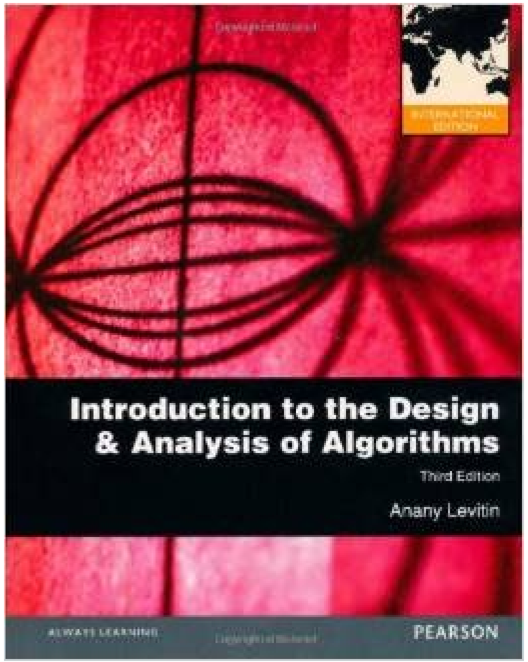
Space/Time-Tradeoff-Algorithmen

Uwe Neuhaus

Literaturangaben

Diese Lerneinheit basiert größtenteils auf dem Buch „The Design and Analysis of Algorithms“ von Anany Levitin.

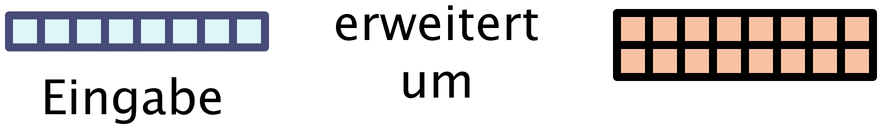
In dieser Einheit behandelte Kapitel:
7 Space and Time Tradeoffs
7.2 Input Enhancements in String Matching (nur Horspool-Algorithmus)
7.3 Hashing
7.4 B-Trees



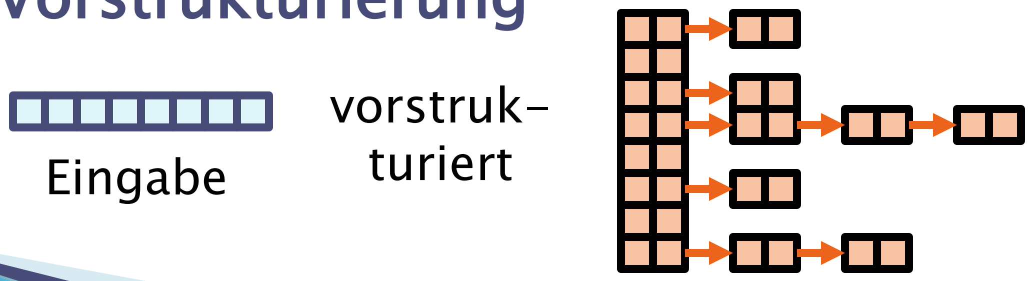
Space/Time Tradeoffs I

- Space/time tradeoff-Algorithmen:
  - Nutzung von **zusätzlichem Speicherplatz**
  - zur **Verbesserung der Laufzeit** des Algorithmus

- Zwei wichtige Varianten
  - **Eingabeerweiterung**



  - **Vorstrukturierung**



Space/Time Tradeoffs II

- **Eingabeerweiterung**
  - Vorverarbeitung der Eingabe, um bestimmte **Information** zu **ermitteln**, mit der das Problem später **effizienter gelöst** werden kann
  - **Beispiele**
    - Counting Sort
    - Zeichenkettensuche (z. B. Horspool-Algorithmus)
- **Vorstrukturierung**
  - Vorverarbeitung der Eingabe, um später **schneller** auf die Eingabeelemente **zugreifen** zu können
  - **Beispiele**
    - Hashing
    - Indexierung (z. B. mittels B-Bäumen)

Wiederholung: Zeichenkettensuche mit Brute Force

- **Muster**: Suchzeichenkette mit **m** Zeichen
- **Text**: (Lange) Zeichenkette mit **n** Zeichen, in der gesucht wird

- **Brute Force-Algorithmus**
  - **Schritt 1:** Muster am Textanfang ausrichten
  - **Schritt 2:** Muster Zeichen für Zeichen von links nach rechts vergleichen bis
    - alle Zeichen des Musters gefunden wurden (Erfolgsfall) oder
    - ein Zeichen nicht übereinstimmt
  - **Schritt 3:** Falls Muster nicht gefunden wurde und der Text noch nicht zu Ende ist: Muster ein Zeichen nach rechts verschieben und Schritt 2 wiederholen

Horspool-Algorithmus

- Verwendet **Eingabeerweiterung**:
  - Vorverarbeitung der **Suchzeichenkette**
  - Erzeugt eine **Shift-Tabelle**
  - Shift-Tabelle bestimmt **Größe der Verschiebung** bei Nichtübereinstimmung
  - Vergleiche Muster von **rechts nach links**
  - **Immer ausschlaggebend**: Textzeichen, das mit dem letzten Zeichen des Musters ausgerichtet ist

Wie weit darf man shiften?

Betrachte stets das **erste verglichene Zeichen**

- Das Zeichen kommt im Muster nicht vor

  `.....C.........` (`C` kommt nicht im Muster vor)

  `BAOBAB`

- Das Zeichen ist im Muster (aber nicht ganz rechts)

  `.....O.........` (`O` kommt einmal vor)

  `BAOBAB`

  `.....A.........` (`A` kommt mehrfach vor)

  `BAOBAB`

- Das Zeichen stimmt überein (spätere Zeichen aber nicht)

  `...CAB.........`

  `BAOBAB`

Shift-Tabelle

- Berechne **Shift-Länge** für alle Buchstaben c:
  - Falls c in ersten m-1 Zeichen des Musters enthalten: **shift(c)** = Abstand des am weitesten rechts stehenden c zum letzten Buchstaben des Musters
  - Sonst: **shift(c)** = Musterlänge m
- Shift-Längen **vor** Suche ermittelt und in **Shift-Tabelle** speichern
- Das **Alphabet** des Textes/Musters ist **Index** der Shift-Tabelle
- **Beispiel** für Muster „BAOBAB“

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

Shift-Tabellen-Algorithmus

ALGORITHM *ShiftTable*($P[0..m-1]$)

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: *Table*$[0..size-1]$ indexed by the alphabet's characters and
//   filled with shift sizes computed by formula (7.1)
initialize all the elements of *Table* with m
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Horspool-Algorithmus

- **Schritt 1**
  - Erstelle **Shift-Tabelle** für das gegebene Muster
- **Schritt 2**
  - Muster mit dem **Textanfang** ausrichten
- **Schritt 3**
  - Vergleiche die Zeichen des Musters **von rechts nach links**
  - Alle Zeichen stimmen überein → **Muster gefunden**
  - Sonst:
    - Ermittle Textzeichen c, das dem **letzten Zeichen** des Musters gegenüber steht
    - Bestimme **Shift-Länge für c**, verschiebe Muster entsprechend
    - **Wiederhole** bis Muster gefunden oder Textende erreicht

Horspool–Algorithmus (Pseudocode)

ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$
    //Implements Horspool's algorithm for string matching
    //Input: Pattern $P[0..m-1]$ and text $T[0..n-1]$
    //Output: The index of the left end of the first matching substring
    //   or -1 if there are no matches
    ShiftTable$(P[0..m-1])$ //generate *Table* of shifts
    $i \leftarrow m-1$ //position of the pattern's right end
    while $i \le n-1$ **do**
        $k \leftarrow 0$ //number of matched characters
        while $k \le m-1$ **and** $P[m-1-k] = T[i-k]$ **do**
            $k \leftarrow k+1$
        if $k = m$
            return $i - m + 1$
        else $i \leftarrow i + Table[T[i]]$
    return -1

Horspool-Algorithmus: Beispiel

Shift-Tabelle

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

Text: „BARD LOVED BANANAS"

Muster: „BAOBAB"

```
BARD LOVED BANANAS
BAOBAB
      BAOBAB
        BAOBAB
              BAOBAB
```

(Muster nicht gefunden)

Hashing

- Sehr effiziente Methode zur Implementierung von **Dictionaries** mit den Operationen
  - find
  - insert
  - delete
- Basiert auf einer **Änderung der Datendarstellung** (→ voriges Kapitel) und einem **Space/Time-Tradeoff**
- Wichtige **Anwendungsbereiche**
  - Symbol-Tabellen (Compiler, Interpreter)
  - Datenbanken

Hashtabellen und Hashfunktionen

- **Grundidee:**
  Bilde die **n** Schlüssel einer **potenziell sehr großen Schlüsselmenge K** mittels einer definierten Funktion (**Hashfunktion**) auf eine **handhabbare** Tabelle (**Hashtabelle**) der Größe **m** ab
- Hashfunktion **h: K → Position (Zelle) in der Hashtabelle**
- **Beispiel:**
  Zugriff auf Studierendendaten über Matrikelnummer
  Hash-Funktion: $h(k) = k \bmod m$ (m typischerweise Primzahl)
- Allgemeine Anforderungen an Hashfunktionen:
  - **Leicht zu berechnen** (Ausnahme: in der Kryptologie)
  - Schlüssel sollen **gleichmäßig** über die Hashtabelle **verteilt** werden

Kollisionen

- **Definition**:
  Zwei Schlüssel werden auf dieselbe Position abgebildet

$$h(k_1) = h(k_2)$$

- Gute Hashfunktionen erzeugen **möglichst wenige** Kollisionen
- Kollisionen grundsätzlich aber **nicht vermeidbar** (Geburtstagsparadox)

Strategien zur Kollisionsbehandlung (I)

Open hashing

- Mehrere Schlüssel pro Zelle möglich
- Jede Zelle ist der Kopf einer verketteten Liste aller auf sie abgebildeter Schlüssel
- Andere Namen:
  - closed addressing (geschlossene Adressierung)
  - separate chaining (Verkettung)

Strategien zur Kollisionsbehandlung (II)

Closed hashing

- Nur ein Schlüssel pro Zelle möglich
- Bei einer Kollision wird eine andere Zelle gesucht:
  - Linear probing/lineares Sondieren:
    Finde nächste freie Zelle
  - Quadratic probing/quadratisches Sondieren:
    Suche nach freien Zellen in quadratisch wachsenden Abständen
  - Double hashing/doppeltes Hashing:
    Verwende eine zweite Hashing-Funktion um neue Zelle zu finden
- Anderer Name: open addressing (offene Adressierung)

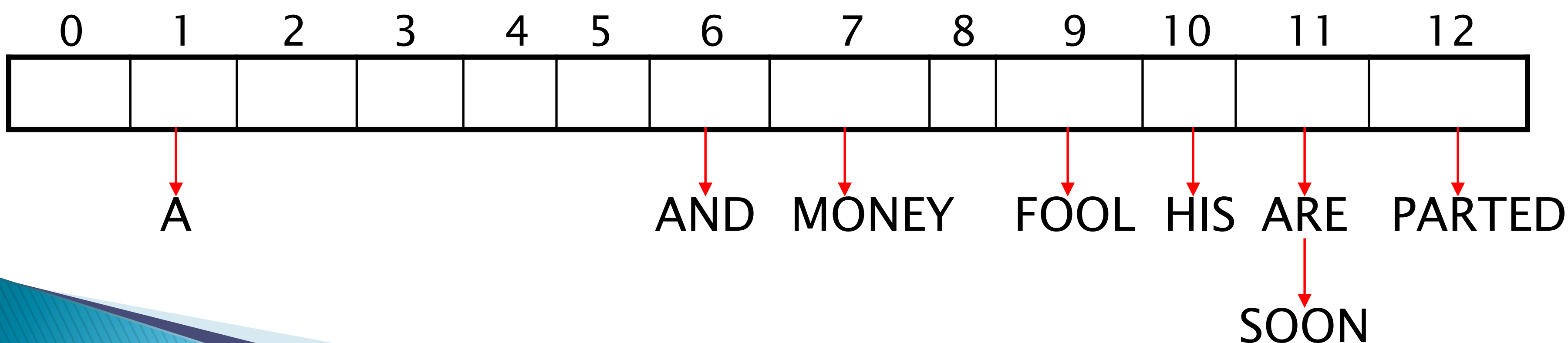
Open hashing: Beispiel

Schlüssel werden in verketteten Listen außerhalb der Hashtabelle gespeichert. Die Zellen der Tabelle sind die Kopfelemente der Listen.

Beispiel: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

h(K) = Summe der Buchstaben von K modulo 13 (A=1, B=2, usw.)

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
h(k)	1	9	6	10	7	11	11	12



Open hashing: Effizienz

- Falls die Hashfunktion die Schlüssel gleichmäßig verteilt, beträgt die **durchschnittliche Listenlänge**:

$$\alpha = n/m$$

- Dieses Verhältnis wird **Load-Faktor** genannt
- **Durchschnittlicher Suchaufwand**
  - Erfolgreiche Suche: $\approx 1 + \alpha/2$
  - Erfolglose Suche: α
- Der Load-Faktor α wird **klein** gehalten (idealerweise in der Nähe von 1)
- Open hashing funktioniert selbst noch bei $n > m$

Closed hashing mit linearem Sondieren – Beispiel

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
h(K)	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Closed hashing – Analyse

- Funktioniert nicht falls $n > m$
- Keine zusätzlichen Pointer notwendig
- Löschung von Schlüsseln **nicht** trivial
- Anzahl der Sondierungen um Schlüssel zu finden/einzufügen/zu löschen abhängig vom **Load-Faktor** $\alpha = n/m$ und der **Strategie zur Kollisionsbehandlung**
- Beispiel: Lineares Sondieren
  - Erfolgreiche Suche: $(½)\ (1 + 1/(1-\alpha))$
  - Erfolglose Suche: $(½)\ (1 + 1/(1-\alpha)^2)$
- Mit steigendem Load-Faktor steigt der Aufwand dramatisch:

α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

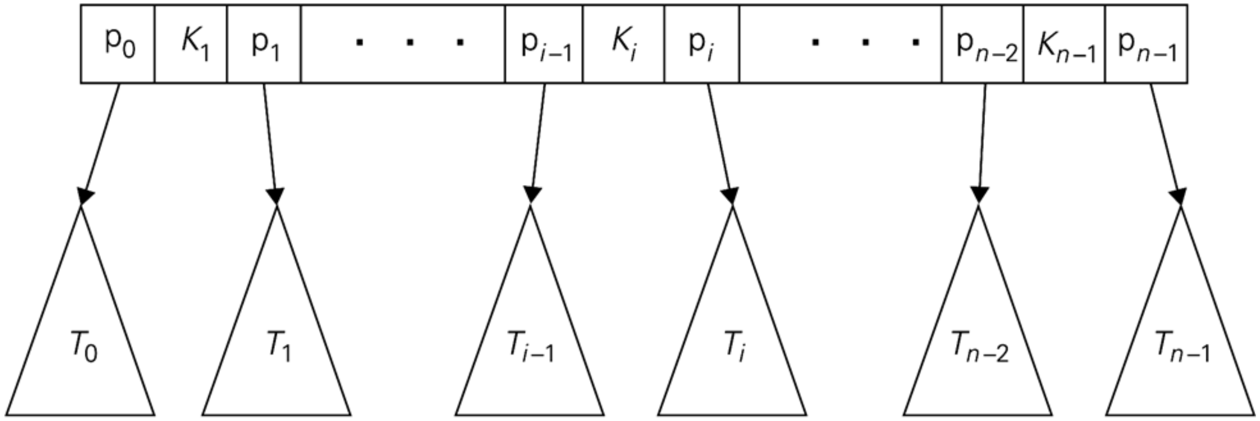
B-Bäume: Motivation

Merkmal	Hauptspeicher	Externer Speicher
Zugriffsgeschwindigkeit	sehr hoch	gering
Anschaffungspreis	sehr hoch	niedrig
Größe	klein – mittel	groß – sehr groß
Speicherung	volatil	persistent
Zugriff	datenwortweise	blockweise

- Große Mengen von Datensätzen müssen auf **externe Speicher** (Festplatten) ausgelagert werden
- Bisher behandelte Datenstrukturen für **schnellen, wahlfreien Zugriff** ausgelegt
- **Benötigt**: Effiziente Datenstruktur, die die besonderen Charakteristiken externer Speicher (**geringe Geschwindigkeit**, **blockweiser Zugriff**) berücksichtigt

B-Bäume: Grundidee

- Erweiterung der 2-3-Bäume
  - **Große Anzahl** von Schlüsseln pro Knoten möglich
  - **Hohe Schlüsselzahl** pro Knoten:
    → **weniger Knoten**
    → **geringere Baumhöhe**
    → **weniger Speicherzugriffe**
- Wähle Schlüsselzahl so, dass ein Knoten **einen Block füllt**



B-Bäume: Definition

B-Baum der **Ordnung $m \geq 3$**

- Die **Wurzel** ist ein **Blatt** oder besitzt **zwischen 2 und m Kinder**
- Alle anderen Knoten besitzen **zwischen $\lceil m/2 \rceil$ und m Kinder** (entsprechend: **zwischen $\lceil m/2 \rceil - 1$ und $m-1$ Schlüssel**)
- Der Baum ist **vollständig balanciert** (d. h. alle Blätter befinden sich auf der selben Ebene)

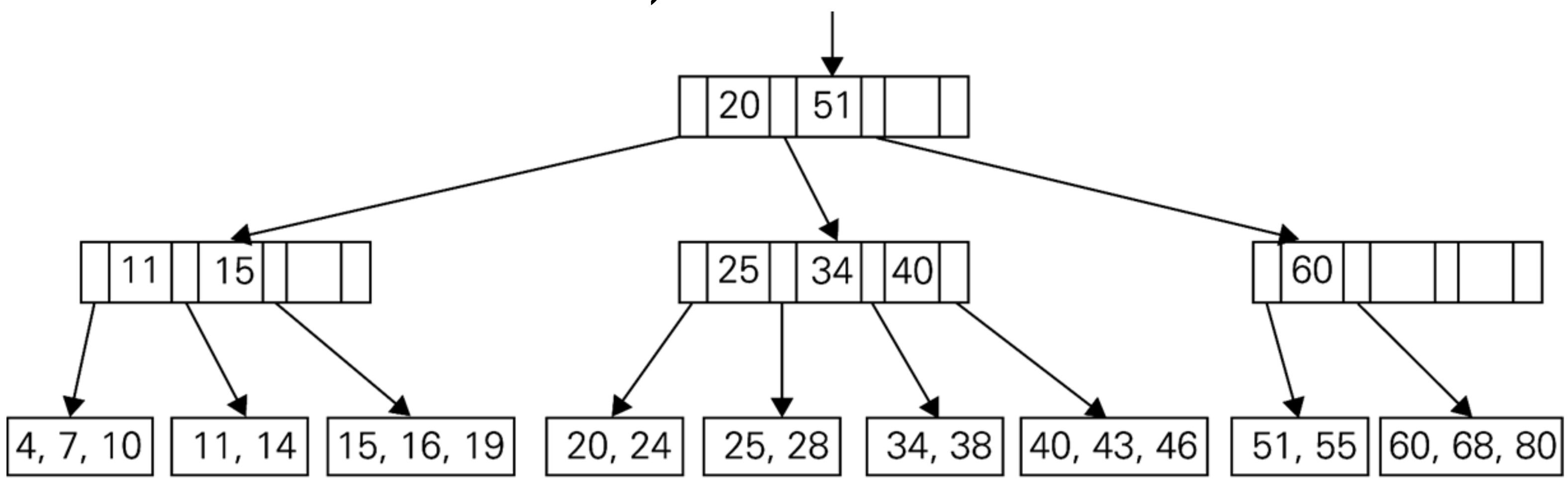


FIGURE 7.8 Example of a B-tree of order 4

B-Bäume: Operationen

- Suche
  - Ähnlich wie bei Binärbäumen
  - Zusätzlich Schlüsselsuche innerhalb eines Knotens
  - Effizienz bestimmt von der Anzahl der Knotenzugriffe (nicht der Schlüsselvergleiche)
  - Suche in O(log n) möglich
- Einfügen und Löschen
  - Durch strukturelle Anforderungen nicht trivial
  - Dennoch in O(log n) realisierbar