Kapitel 3

Vektoren

3 Vektoren

Ein Vektor (Feld, array) verwaltet eine fix vorgegebene Anzahl von Elementen eines einheitlichen Typs.

- Zugriff auf ein Element über einen ganzzahligen Index (die Position im Vektor)
- Aufwand des Zugriffes für alle Elemente konstant

x_0	x_1	x_2	…	x_{n-1}	x_n

Methoden

- Hashing: Datenorganisationsform
- Sortieren: Datenreorganisationsmethoden

Ein *Dictionary* ist eine Datenstruktur, bei der die einzelnen Elemente jeweils aus einem *Schlüssel-* (key) und einem *Information*-Teil (info) bestehen

- Unterstützt nur die einfachen Operationen des Einfügens, Löschens und der Suche zur Verfügung stellt.
- Beispiele sind
  - Wörterbücher, Namenslisten, Bestandslisten, etc.
- Vektoren sind zur Realisierung von Dictionaries gut geeignet
- Struktur

key_0	key_1	key_2	…	key_{n-1}
$info_0$	$info_1$	$info_2$		$info_{n-1}$

Wörterbuch

(Ausschnitt)

Schlüssel	Information
computable	berechenbar
computation	Berechnung
compute	(be)rechnen
computer	Rechenautomat

Bestandsliste

Schlüssel	Information
1	CPU
4	Bildschirm
17	Tastatur
25	Maus

Operationen

Construct

- Erzeugen eines leeren Dictionary

IsEmpty

- Abfrage auf leeres Dictionary

Insert

- Einfügen eines Elementes

Delete

- Löschen eines Elementes

LookUp

- Abfrage eines Elementes über seinen Schlüssel und liefern der Information

Klasse Dictionary

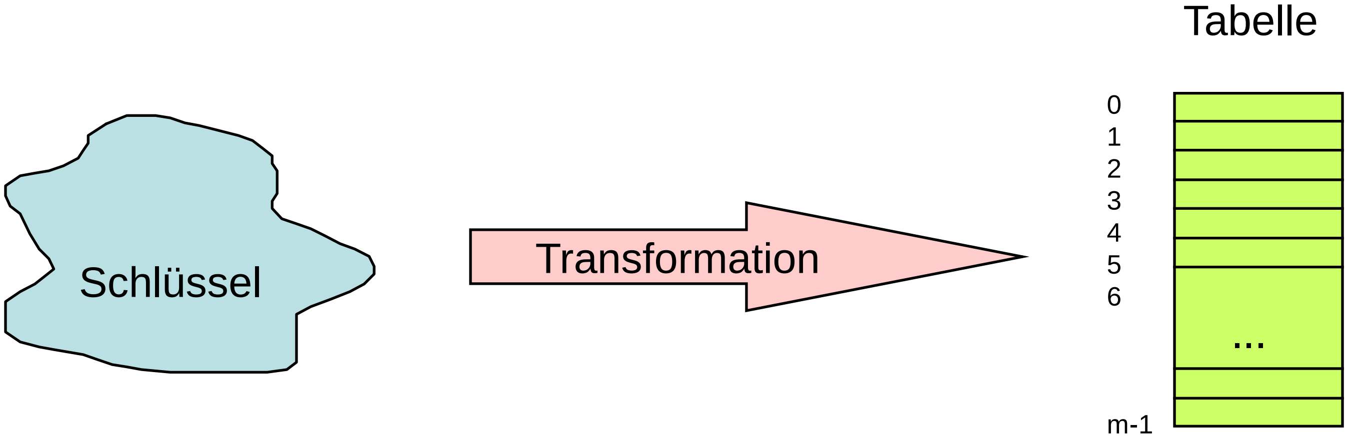
```
class Dictionary {
  private:
    node { KeyType Key; InfoType Info; };
    node *Dict;
    int NumberElements;
  public:
    Dictionary(int max) {
      Dict = new node[max];
      NumberElements = 0;
    }
    ~Dictionary() { delete[] Dict; }
    void Insert(KeyType k, InfoType I);
    void Delete(KeyType k);
    InfoType LookUp(KeyType k);
};
```

3.2 Hashing

Hashing ist eine Methode Elemente in einer Tabelle direkt zu adressieren, in dem ihre Schlüsselwerte durch arithmetische Transformationen direkt in Tabellenadressen (Indizes) übersetzt werden

Mit anderen Worten, der Index (die Position in der Tabelle) eines Elements wird aus dem Schlüsselwert des Elements selbst berechnet.

Schlüssel $\rightarrow$ Adresse (Index)



Hashing, allgemein

Ansatz

Menge K von n Schlüsseln $\{k_0, k_1, \ldots, k_{n-1}\}$

Hashtabelle T der Größe m

Randbedingung: n >> m

(Anzahl der möglichen Schlüsselwerte viel größer als Plätze in der Tabelle)

Transformation

Hash-Funktion h

$$h: K \rightarrow \{0, 1, \ldots m-1\}$$

Für jedes j soll der Schlüssel k_j an der Stelle $h(k_j)$ in der Tabelle gespeichert werden. Der Wert h(K) wird als Hash-Wert von K bezeichnet.

Wahl der Hashfunktion (eigentlich) beliebig!

Gewünschte Eigenschaften

- Einfach und schnell zu berechnen.
- Elemente werden gleichmäßig in der Tabelle verteilt.
- Alle Positionen in der Tabelle werden mit gleicher Wahrscheinlichkeit berechnet.
- Beispiele
  - Modulo Bildung, Schlüssel mod m
    - Man sollte darauf achten, daß m eine Primzahl ist, sonst kann es bei Schlüsseltransformationen (Strings) zu Clusterbildungen (Anhäufungen) durch gemeinsame Teiler kommen.
  - Bitstring-Interpretation
    - Teile aus der binären Repräsentation des Schlüsselwertes werden als Hashwert herangezogen.
  - Transformationstabellen

Beispiel Hashing

Bestandsliste

Größe der Liste 7

Hashfunktion h(k) = k mod 7

Datensätze

Schlüssel	Information
1	CPU
4	Bildschirm
17	Tastatur
25	Maus

1 mod 7 = 1 → 1

17 mod 7 = 3 → 3

4 mod 7 = 4 → 4

0		
1	1	CPU
2		
3	17	Tastatur
4	4	Bildschirm
5		
6		

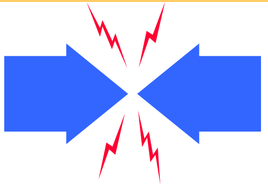
?

25 mod 7 = **4** → Position in der Tabelle schon besetzt

Kollision

Kollision

Eine *Kollision* tritt auf, wenn zwei oder mehrere Schlüsselwerte auf dieselbe Tabellenposition abgebildet (*gehasht)* werden.



Dies bedeutet, dass für 2 Schlüssel k_i, k_j, mit $k_i \neq k_j$, gilt $h(k_i) = h(k_j)$.

- Diese Situation ist aber zu erwarten, da es viel mehr mögliche Schlüssel als Tabellenplätze gibt ($n >> m$ als Randbedingung).
- Die Hashfunktion h ist i.A. nicht injektiv, d.h. aus $h(x) = h(y)$ folgt nicht notwendigerweise $x=y$.

Kollisionsbehandlung

Eine Kollision löst eine notwendige *Kollisionsbehandlung* aus.

- D.h., für den kollidierenden Schlüsselwert muss ein Ausweichsplatz gefunden werden.
  - Maßnahmen zur Kollisionsbehandlung sind meist recht aufwendig und beeinträchtigen die Effizienz von Hashverfahren.
- der Kollisionspfad ist definiert durch die Ausweichplätze aller Elemente, die auf den gleichen Platz "ge-hasht" wurden
- Wir betrachten nun 2 simple Kollisionsbehandlungen

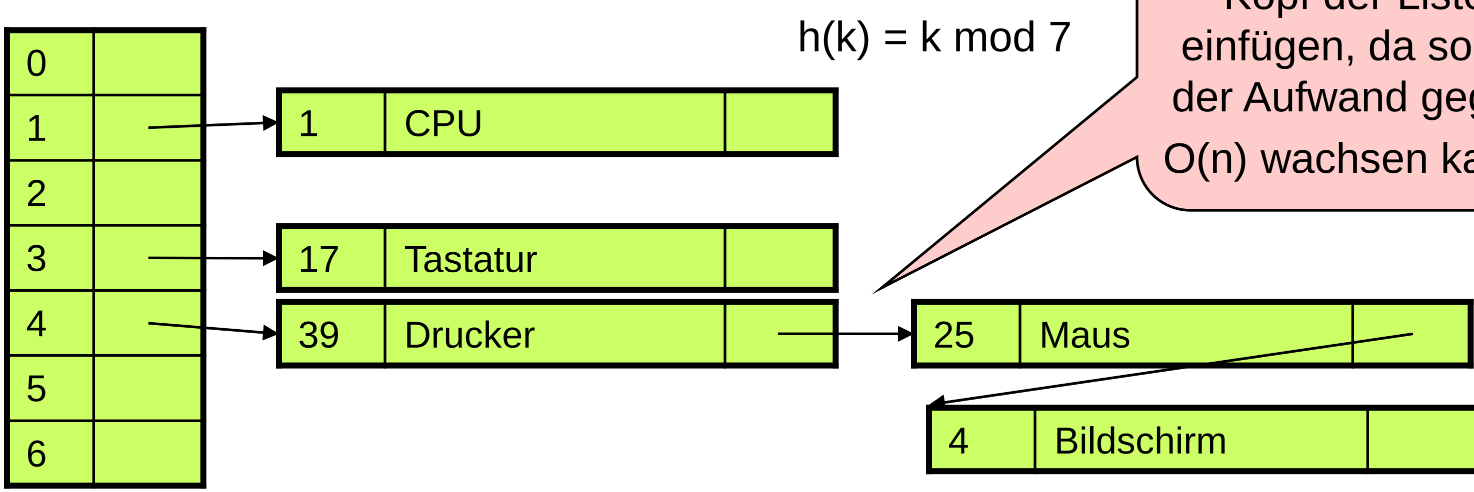
  Separate Chaining und *Double Hashing*

Beim Separate (Simple) Chaining wird die Kollisions-behandlung mit linearen Listen bewerkstelligt.

Kollidierende Schlüsselwerte werden in einer linearen Überlaufkette (Liste) ausgehend vom ursprünglich gehashten Tabellenplatz gespeichert.

Beispiel:

Wichtig: Element am Kopf der Liste einfügen, da sonst der Aufwand gegen O(n) wachsen kann!



Der Eintrag des Elementes (39, Drucker) führt zu einer Verlängerung der Überlaufkette.

3.2.2 Double Hashing

Beim *Double Hashing* wird bei Kollision eine zweite, von h unabhängige, Hashfunktion zur Bestimmung einer alternativen Position verwendet.

Überlauf auf Position $a_0 = h(k)$

Bestimmung einer alternativen Position mit Kollisionsfunktion $g : K \rightarrow \{1, \dots m-1\}$, z.B. $a_{i+1} = (a_i + g(k)) \bmod m$

Beispiel

K … Kollision

h(k) = k mod 7

g(k) = letzte Ziffer von k mal 3

$a_0 = 25 \bmod 7 = 4$ K
$a_1 = (4 + (5*3)) \bmod 7 = 5$

$a_0 = 39 \bmod 7 = 4$ K
$a_1 = (4 + (9*3)) \bmod 7 = 3$ K
$a_2 = (3 + (9*3)) \bmod 7 = 2$

0		
1	1	CPU
2	39	Drucker
3	17	Tastatur
4	4	Bildschirm
5	25	Maus
6		

Linear Probing

Spezialfall des Double Hashing

Kollisionsbehandlung: Man verwendet den nächsten freien Platz

Bei Erreichen des Tabellenendes sucht man am Tabellenanfang weiter

Hashfuktionen: $a_0 = h(k)$, $a_{i+1} = (a_i + g(k)) \bmod m$

g(k) = 1

Beispiel

25 mod 7 = 4 K
(4+1) mod 7 = 5

39 mod 7 = 4 K
(4+1) mod 7 = 5 K
(5+1) mod 7 = 6

0		
1	1	CPU
2		
3	17	Tastatur
4	4	Bildschirm
5	*25*	*Maus*
6	*39*	*Drucker*

Suchen und Löschen in Hashtabellen

Suchen

- Solange eine Zieladresse durch ein Element belegt ist, und der gesuchte Schlüsselwert noch nicht gefunden wurde, muss über den Kollisionspfad weitergesucht werden.
  - Separate Chaining: Verfolgen der linearen Liste
  - Double Hashing: Aufsuchen aller möglichen Plätze der Kollisionsfkt.

Löschen

- Separate Chaining: Entfernen des Listenelements
- Double Hashing: Positionen, an denen ein Element gelöscht wurde, müssen gegebenenfalls mit einem speziellen Wert (“wiederfrei”) markiert werden, damit der entsprechende Kollisionspfad für die restlichen Elemente nicht unterbrochen wird.

Eigenschaften

Generell für Hashverfahren

+ Aufwand beim Zugriff auf ein Element im besten Fall konstant, O(1), einfache Evaluation der Hashfunktion.
- Kollisionsbehandlung aufwendig, volle Hashtabelle führt zu vielen Kollisionen, daher (Faustregel) nie über 70% füllen.
- Kein Zugriff in Sortierreihenfolge möglich.

Separate Chaining

+ Dynamische Datenstruktur, beliebig viele Elemente.
- Speicherplatzaufwendig (zusätzliche Zeigervariable).

Double Hashing

+ Optimale Speicherplatzausnützung.
- Statische Datenstruktur, Tabellengröße vorgegeben.
- Kollisionbehandlung komplex, „wiederfrei“ Markierung beim Löschen.

Analyse Hashing

Datenverwaltung

- Einfügen und Löschen unterstützt

Datenmenge

- beschränkt
  - abhängig von der Größe der vorhandenen Hashtabelle

Modelle

- Hauptspeicherorientiert
- Unterstützung simpler Operationen
  - keine Bereichsabfragen, keine Sortierreihenfolge

Analyse Hashing (2)

Speicherplatz	O(1)
Konstruktor	O(1)
Zugriff	O(1)
Einfügen	O(1)
Löschen	O(1)

gültig nur für reines Hashverfahren ohne Kollisions-behandlung

Bitte beachten: Aufwand von Hashing stark abhängig vom Kollisionsverfahren, geht bei Kollisionen oft gegen O(n)

3.3 Dynamische Hash-Verfahren

Dynamische Hash-Verfahren versuchen im Falle von Kollisionen die ursprüngliche Hashtabelle zu erweitern

Anlegen von Überlaufbereichen (z.B. lineare Listen)

Vergrößerung des Primärbereichs (ursprüngliche Tabelle)

- Erfordert Modifikation der Hash-Funktion
- Ansatz: Familien von Hash-Funktionen
  - $h_1: K \rightarrow addr_1$
  - ...
  - $h_n: K \rightarrow addr_n$
  - Wobei $|addr_1| < |addr_2| < \ldots < |addr_n|$
- Ziel: Wechsel von $addr_i$ auf $addr_{i+1}$ erfordert minimale Reorganisation der Hash-Tabelle (d.h. Umspeichern von Daten minimieren)

Hash-Funktionen Schema

Simpler Ansatz

$|addr_{i+1}| = 2 * |addr_i|$

h(x) ist eine Hash-Funktion

$h_i(x)$ seien die i „least significant bits“ von h(x)

Somit gilt

$h_{i+1}(x) = h_i(x)$ oder

$h_{i+1}(x) = h_i(x) + 2^i$

Beispiel:

$addr_2 = 0 .. 3$,

$addr_3 = 0 .. 7$,

$h(x) = x$

4 Adresswechsel

Wert	h(x)	binär	$h_2(x)$	$h_3(x)$
7	7	00111	11=3	111=7
8	8	01000	00=0	000=0
9	9	01001	01=1	001=1
10	10	01010	10=2	010=2
13	13	01101	01=1	101=5
14	14	01110	10=2	110=6
23	23	10111	11=3	111=7
26	26	11010	10=2	010=2

Dynamische Hashverfahren

„Two-Disk-Access“ Prinzip

- Finden eines Schlüssel mit maximal 2 Platten Zugriffen

Hashverfahren für externe Speichermedien

Linear Hashing

- Verzeichnisloses Hashverfahren
- Primärbereiche, Überlaufbereiche

Extendible Hashing

- Verzeichnis mit binärer Expansion
- Primärbereiche

Bounded Index Size Extendible Hashing

- Verzeichnis mit beschränkter Größe
- Binäre Expansion der Blöcke

3.3.1 Linear Hashing

W. Litwin 1980

Organisation der Elemente in Blöcken (Buckets)

- Analog zu B-Bäumen
- Blockgröße b, dh in einer Adresse in der Tabelle können bis zu b Elemente gespeichert werden (=Block)

Idee

- Bei Überlauf eines Blocks (= b+1. Element von einem Block wird eingefügt) (Splitting) wird der Primär- und Überlaufbereich um jeweils einen Block erweitert
- Primärbereich durch Hinzufügen eines Blocks am Ende der Hash-Tabelle
- Überlaufbereich durch lineare Liste für einzelne Blocks
- Rundennummer d speichert die Anzahl der Splits pro Block ab, es gibt gleichzeitig Blocks die d Mal und Blocks die d+1 Mal gesplittet wurden

Splitting wird „Runden“-weise durchgeführt

- Eine Runde endet, wenn alle ursprünglichen N Blocks (für Runde R) gesplittet worden sind
  - Blocks 0 bis NextToSplit-1 sind schon gesplittet
  - Nächster Block zum Splitten ist definiert durch Index NextToSplit
- Aktuelle Rundennummer ist definiert durch d

Suche

- Suche nach Wert x
- **a =** h_d**(x);**
- **if(a < NextToSplit) a =** h_{d+1}**(x);**

d=2, NextToSplit=1, b=3

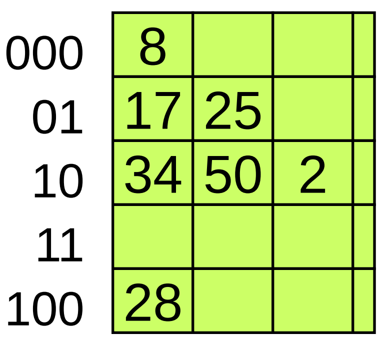
000	8			
01	17	25		
10	34	50	2	
11				
100	28			

2=000010
8=001000
17=010001
25=011001
28=011100
34=100010
50=110010

Einfügen

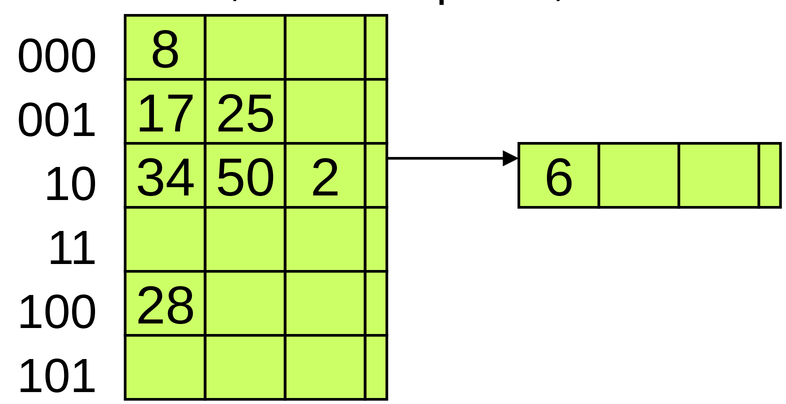
- Suche Block (h_d oder h_{d+1})
- Falls Block voll
  - Füge Element in einen Überlaufblock
    - Neues Element in der linearen Liste
  - Splitte Block NextToSplit und erhöhe NextToSplit um 1
- Beispiel: Einfügen von 6 (=000110)

d=2, NextToSplit=1, b=3



2=000010
8=001000
17=010001
25=011001
28=011100
34=100010
50=110010

d=2, NextToSplit=2, b=3



Beispiel: linear Hashing

Aktuelle $h_i(x) = h_3(x)$

d=3, NextToSplit=0, b=3

000	8		
001	17	25	
010	34	50	2
011			
100	28		
101	5		
110			
111	55		

2=000010 28=011100
5=000101 34=100010
8=001000 50=110010
17=010001 55=110111
25=011001

Einfügen: 16, 13, 21, 37

d=3, NextToSplit=0, b=3

000	8	16	
001	17	25	
010	34	50	2
011			
100	28		
101	5	13	21
110			
111	55		

Überlaufblock (zu 101):

37		

16=010000
13=001101
21=010101
37=100101

Überlaufblock

Split

d=3, NextToSplit=1, b=3

0000	16		
001	17	25	
010	34	50	2
011			
100	28		
101	5	13	21
110			
111	55		
1000	8		

Überlaufblock (zu 101):

37		

Bemerkungen

- Die Rundennummer d wird erhöht, wenn das Splitting einmal durch ist, d.h.
  - **`if(NextToSplit == 2`^{d}`) { d++; NextToSplit=0; }`**
- In der Praxis keine linearen Adressräume sondern Aufteilen des Primärbereiches auf mehrere Hash Dateien pro Platte
  - Zuordnungssystem über Verwaltungsblöcke

Hash-Verfahren mit Index

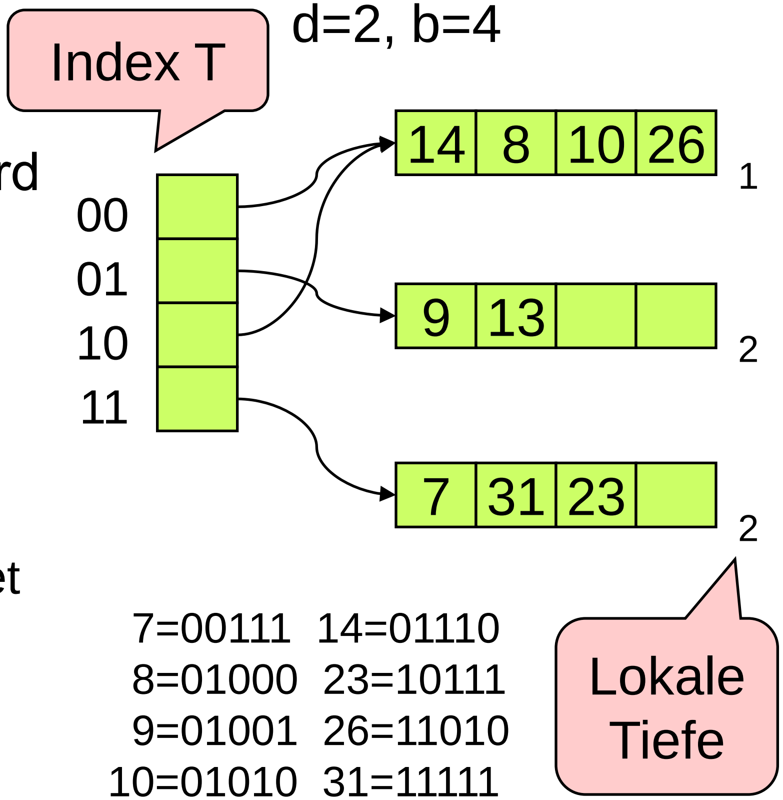
- R. Fagin, J. Nievergelt, N. Pippenger and H.R. Strong, 1979

Idee

- Wenn Primärbereich zu klein wird, Vergrößerung durch Verdopplung
- Primärbereich wird über Index T verwaltet
- Bei Überlauf eines Blocks Split dieses Blocks und Verwaltung des Blocks durch Verdoppeln des Index
  - Index ist viel kleiner als die Datei, daher Verdoppeln viel günstiger
- Keine Überlaufbereiche

Eigenschaften

- Jeder Indexeintrag referenziert genau einen Datenblock
- Jeder Datenblock hat ≤ b Einträge und wird von genau 2^k mit k aus N_0 Indexeinträgen referenziert
- Die Anzahl der Indexeinträge, die den Block referenzieren ist im Block lokal bekannt
  - Wird durch eine lokale Tiefe t ≤ d verwaltet (Gegensatz globale Tiefe d für die gesamte Hash-Tabelle)
  - Anzahl der den Block referenzierenden Indexeinträge entspricht 2^{d-t}



7=00111 14=01110
8=01000 23=10111
9=01001 26=11010
10=01010 31=11111
13=01101

Suche

- Element x im Block $T[h_d(x)]$

Einfügen

Zwei Fälle zu unterscheiden falls ein Block überläuft

- t < d: mehrere Indexeinträge referenzieren diesen Block
- t = d: ein Indexeintrag referenziert diesen Block

Fall 1: t < d

z. B. Einfügen von 6

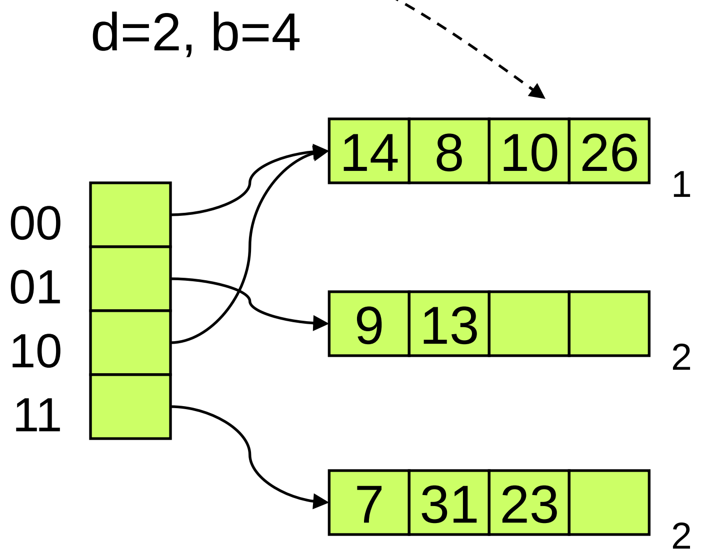
6=00110

Fall 2: t = d

z. B. Einfügen von 15, 19

15=01111

19=10011



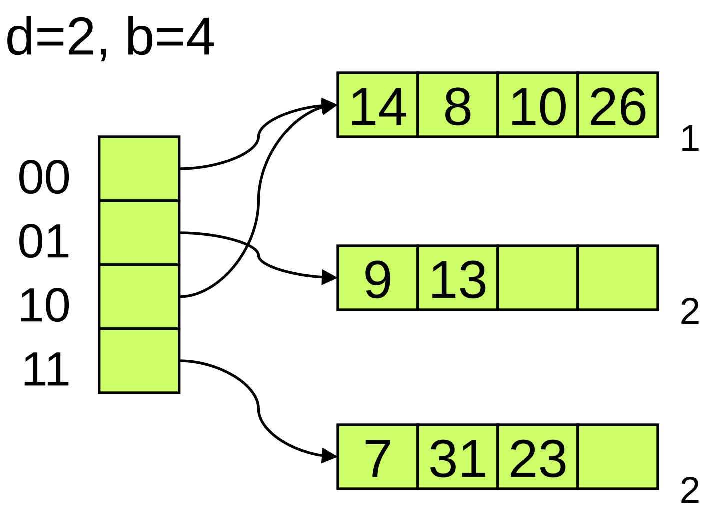
7=00111
8=01000
9=01001
10=01010
13=01101
14=01110
23=10111
26=11010
31=11111

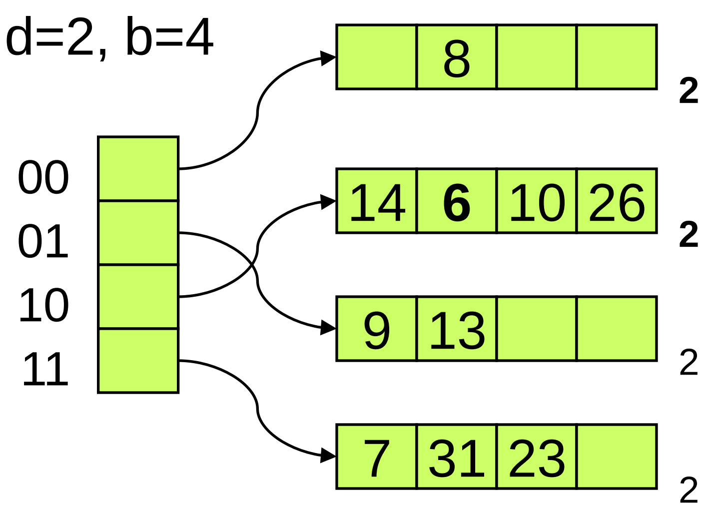
Einfügen Fall 1: t < d

Versuch den Split ohne Indexexpansion durchzuführen

- Neuen Datenblock anfordern
- Aufteilung der Daten des überlaufenden Blocks nach h_{t+1}
- t = t + 1 für alten und neuen Datenblock
- Falls Split wieder zu einem Überlauf führt, wiederholen

Beispiel: Einfügen von 6 (=00110)





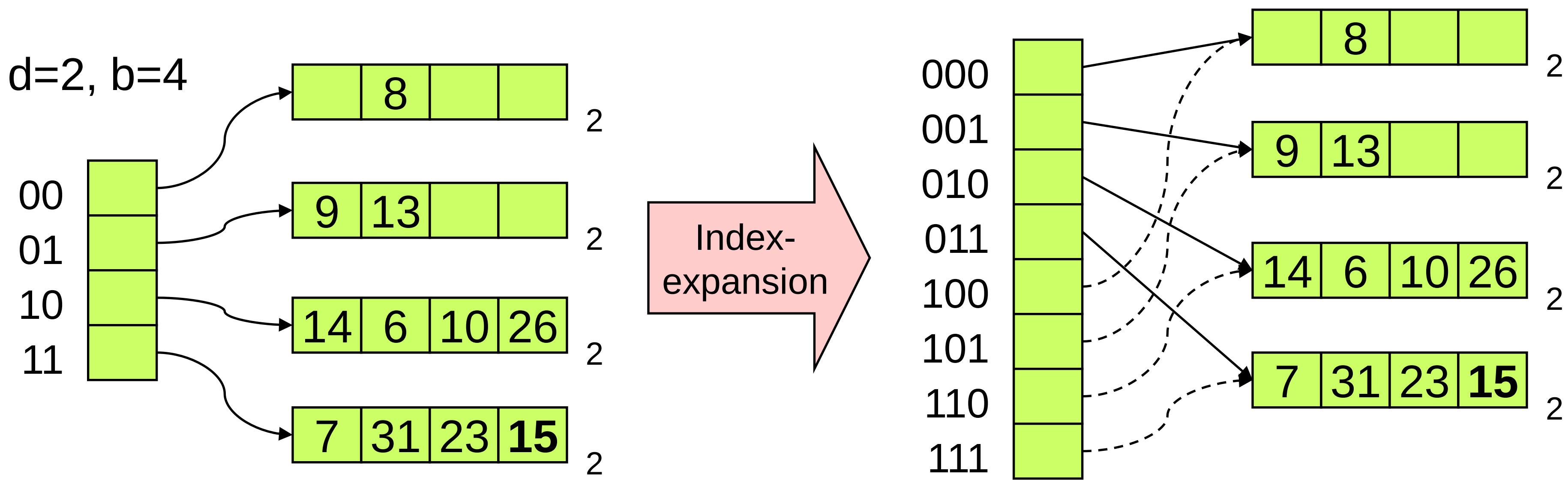
Einfügen Fall 2: t = d

Erfordert eine Indexexpansion (Verdoppelung des Indexes)

- Neuen Speicherbereich für 2^d zusätzliche Referenzen anfordern
- Für jedes T[x], für das gilt $x \geq 2^d$: Indexeintrag $T[x] = T[x-2^d]$
- d = d+1

Danach Rückführung auf Fall 1

Beispiel: Einfügen von 15 und 19

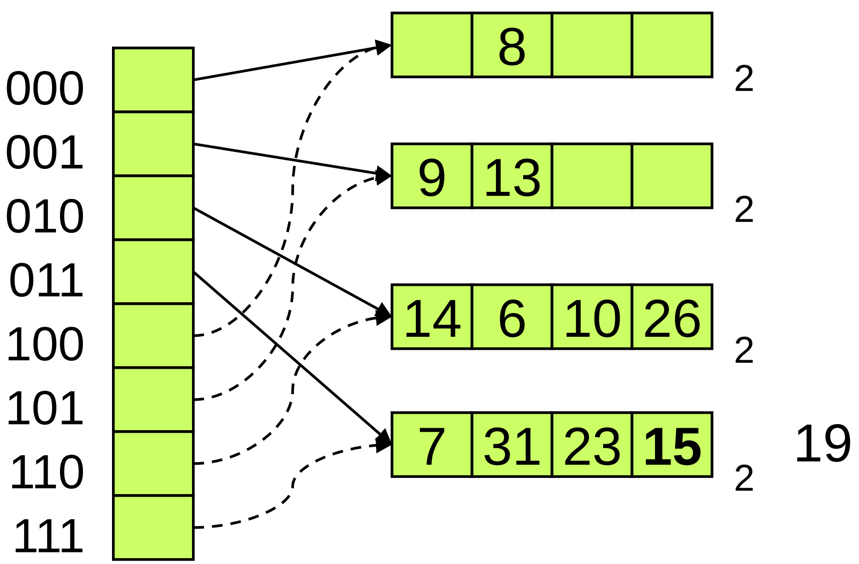


Einfügen Fall 2: t = d (2)

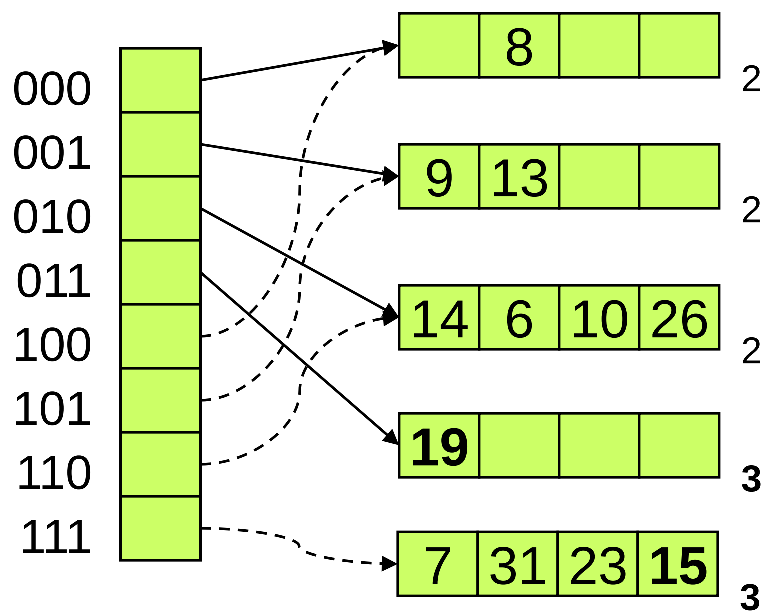
Jetzt Fall 1

7=00111
15=01111
19=10011
23=10111
31=11111

d=**3**, b=4



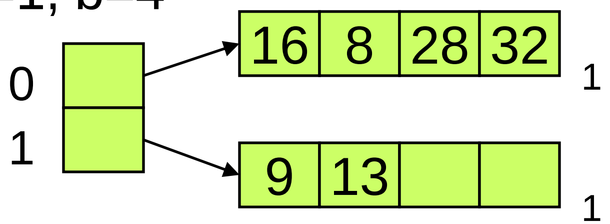
d=**3**, b=4



Problem des Indexwachstums

Indexexpansion kann scheitern, mehrfache Verdoppelungen

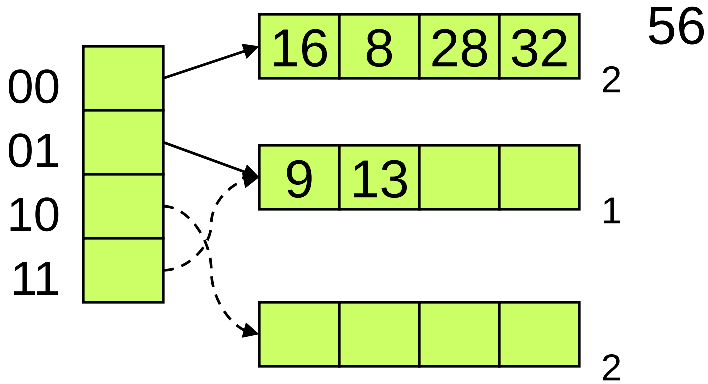
d=1, b=4



8=001000
9=001001
13=001101
16=010000
28=011100
32=100000
56=111000

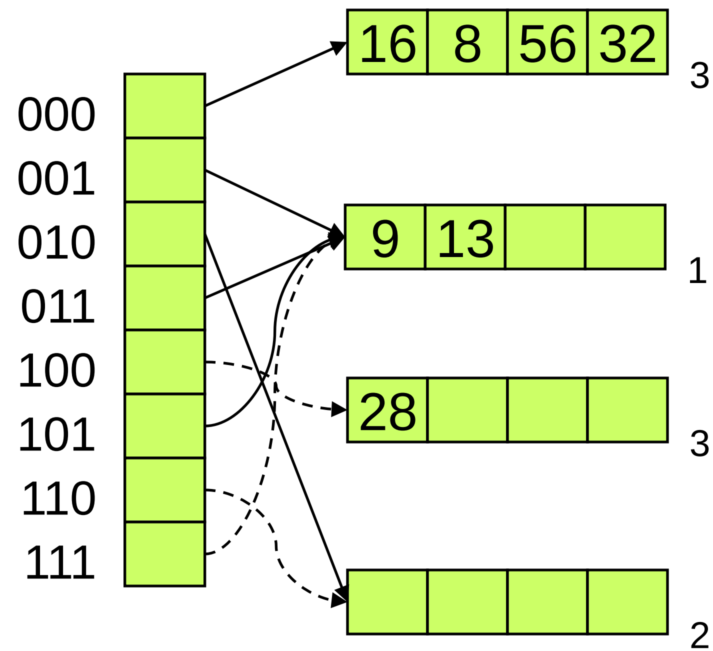
Einfügen 56

d=2, b=4



1. Indexexpansion

d=3, b=4



2. Indexexpansion

Kommentare

- Löschen leerer Blöcke nicht einfach
  - Verschmelzung mit ihren Split-Buddies („Split-Images“)
  - Analog für Indexverkleinerung
- Falls Index im Hauptspeicher gehalten werden kann, nur ein externer Zugriff notwendig
- Im worst case exponentielles Indexwachstum
  - Typisch bei clustered data (Daten sind nicht gleichverteilt)
  - Daher Speicherplatzbedarf für Index im worst case nicht akzeptierbar
    - Index normalerweise im Hauptspeicher
- Problem der Blockung des Index auf der Platte
- Keine garantierte Mindestauslastung

Analyse komplex

- Es lässt sich zeigen, dass Extendible Hashing zur Speicherung einer Datei mit n Datensätzen und einer Blockgröße von b Datensätzen ungefähr 1.44*(n/b) Blöcke benötigt
- Das Verzeichnis hat durchschnittlich für gleichverteilte Datensätze $n^{1+1/b}/b$ Einträge

Großer Vorteil falls das Verzeichnis in den Hauptspeicher passt, nur ein Plattenzugriff auf einen Datensatz notwendig

3.3.3 Bounded Index Size Extendible Hashing

Ansatz durch

- Primärbereich (analog zu Extendible Hashing)
- Index mit beschränkter Größe
- Binäre Expansion der Datenbereiche
  - (1 Block, 2, 4, ... Blöcke)

Versucht dem Unvermögen des Extendible Hashings, den Index auf der Platte unterzubringen, mit einer speziellen Indexstruktur zu begegnen

Struktur

Index besteht aus x Bereichen

Ein Bereich enthält y Einträge der Form <z, ptr> wobei

- ptr ist eine Adresse auf einen Plattenblock
- z gibt die Anzahl der Verdoppelungen der entspr. Datenbereiche
- d.h. ptr ist die Adresse eines Datenbereichs mit 2^z Plattenblöcken
- x und y sind Potenzen von 2

Hash-Werte werden gezont interpretiert, z.B.

- 16 Indexbereiche (x=16), 32 Einträge pro Indexbereich (y=32), 4 Blöcke in jedem Datenbereich
  - Hash-Signatur für den 0. Block der vom 20. Eintrag im Indexbereich 10 referenziert wird

0110 00 10100 1010

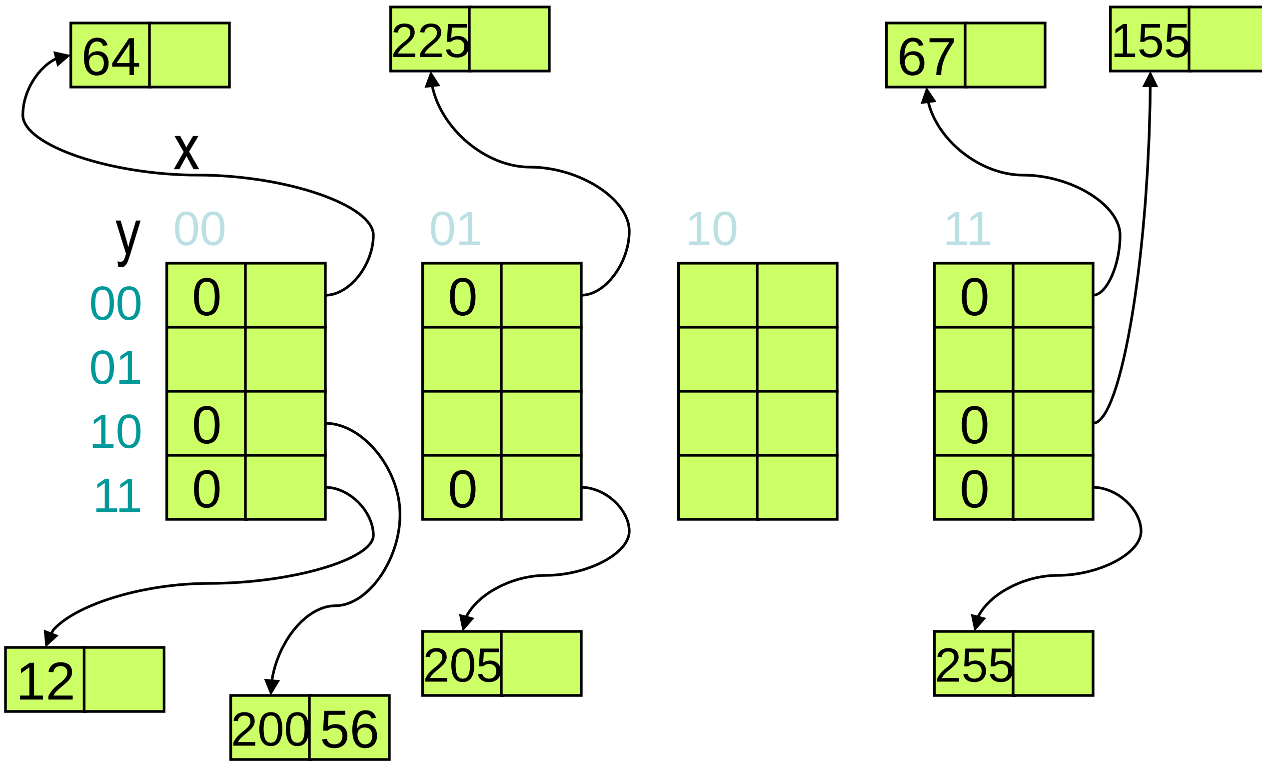
10. Indexbereich

0. Plattenblock

20. Eintrag im IB 10

Beispiel BISEH

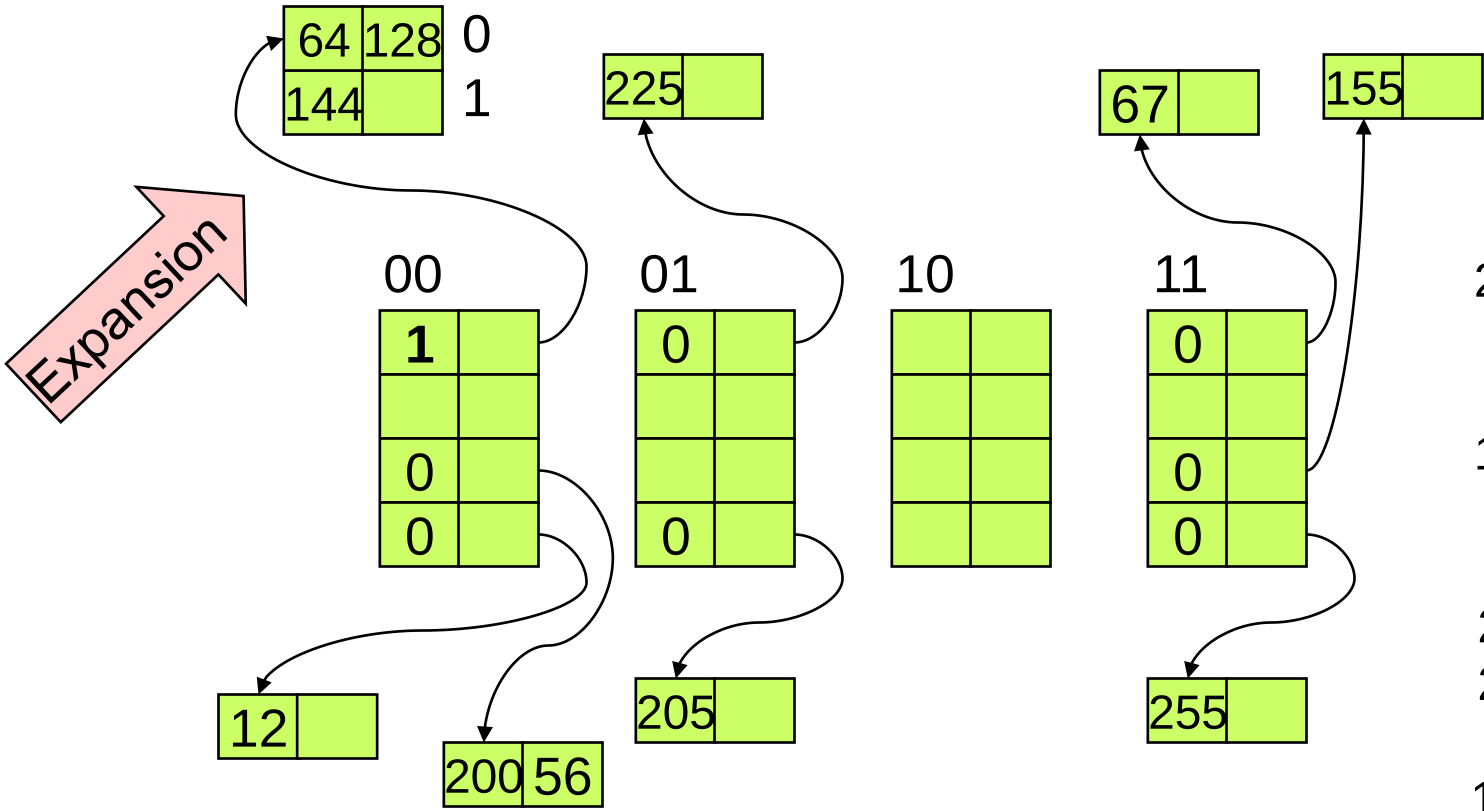
b=2 (Blockgröße), x=4 (Indexbereiche), y=4 (Einträge pro IB)



255 = 1111 11 11
200 = 1100 10 00
56 = 0011 10 00
64 = 0100 00 00
155 = 1001 10 11
67 = 0100 00 11
12 = 0000 11 00
205 = 1100 11 01
225 = 1110 00 01

Beispiel BISEH (2)

Einfügen von 128 und 144, im ersten Verdopplungsversuch erfolgreich

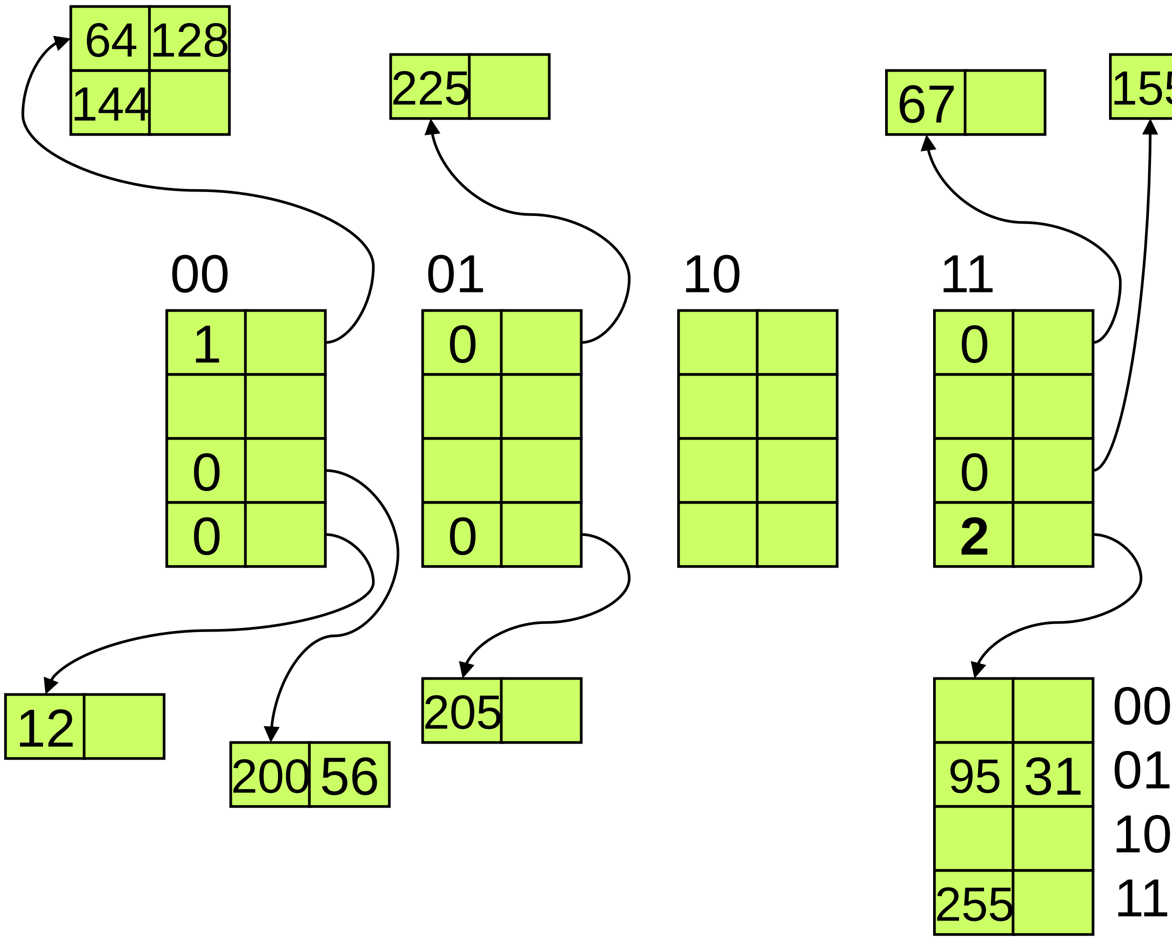


255 = 1111 11 11
200 = 1100 10 00
56 = 0011 10 00
64 = 010<u>0</u> 00 00
155 = 1001 10 11
67 = 0100 00 11
12 = 0000 11 00
205 = 1100 11 01
225 = 1110 00 01

128 = 100<u>0</u> 00 00
144 = 100<u>1</u> 00 00

Beispiel BISEH (3)

Einfügen von 95 und 31, erst im zweiten Verdopplungsversuch erfolgreich



255 = 1111 11 11
200 = 1100 10 00
56 = 0011 10 00
64 = 0100 00 00
155 = 1001 10 11
67 = 0100 00 11
12 = 0000 11 00
205 = 1100 11 01
225 = 1110 00 01
128 = 1000 00 00
144 = 1001 00 00

95 = 0101 11 11
31 = 0001 11 11

Allgemeiner Vergleich von Hash- zu Index-Verfahren

Punkte, die zu beachten sind:

Kosten der periodischen Reorganisation

Häufigkeit von Einfügen und Löschen

Average Case versus Worst Case Aufwand

Erwartete Abfrage Typen

- Hashing ist generell besser bei exakten Schlüsselabfragen
- Indexstrukturen (z.B. Suchbäume) sind bei Bereichsabfragen vorzuziehen

3.4 Sortierverfahren

- Das *Sortieren* von Elementen (Werten, Datensätzen, etc.) stellt in der Praxis der EDV eines der wichtigsten und aufwendigsten Probleme dar.
  - Es existieren hunderte Sortieralgorithmen in den verschiedensten Varianten für unterschiedlichste Anwendungsfälle.
    - Hauptspeicher - Externspeicher
    - Sequentiell - Parallel
    - Insitu - Exsitu
    - Stable - Unstable
    - ...
  - Sortierverfahren gehören zu den am umfangreichsten analysierten und verfeinerten Algorithmen in der Informatik.

Kriterien für die Auswahl von Sortierverfahren

Hauptspeicher - Externspeicher

Kann der Algorithmus nur Daten im Hauptspeicher oder auch Files (Dateien) oder Bänder auf dem Externspeicher (Platte, Bandstation) sortieren?

Insitu - Exsitu

Kommt das Sortierverfahren mit ursprünglichen Datenbereich aus (insitu) oder braucht es zusätzlichen Speicherplatz (exsitu)?

Worst Case - Average Case

Ist das (asymptotische) Laufzeitverhalten des Sortierverfahrens immer gleich oder kann es bei speziellen Fällen “entarten” (schneller oder langsamer werden)?

Stable - Unstable

Wenn mehrere Datensätze denselben Schlüsselwert haben, bleibt dann die ursprüngliche Reihenfolge nach dem Sortieren erhalten?

Aufgabe

- Ein Vektor der Größe n der Form V[0], V[1], …, V[n-1] ist zu sortieren.
  - Lexikographische Ordnung auf den Elementen.
- Nach dem Sortieren soll gelten: $V[0] \leq V[1] \leq \ldots \leq V[n-1]$.

Elementare (Simple) Verfahren (Auswahl)

- Selection Sort
- Bubble Sort

Verfeinerte („Intelligentere“) Verfahren (Auswahl)

- Quicksort
- Mergesort
- Heapsort

Klasse Vector

```
class Vector {
  int *a, size;
public:
  Vector(int max) { a = new int[max]; size = max; }
  ~Vector() { delete[] a; }
  void Selectionsort();
  void Bubblesort();
  void Quicksort();
  void Mergesort();
  int Length();
private:
  void quicksort(int, int);
  void mergesort(int, int, int*);
  void swap(int&, int&);
};
```

Selection Sort oder Minimumsuche Algorithmus

Finde das kleinste Element im Vektor und vertausche es mit dem Element an der ersten Stelle. Wiederhole diesen Vorgang für alle noch nicht sortierten Elemente.

Swap

Das Vertauschen (*Swap*) zweier Elemente (**`int`**) stellt einen zentralen Vorgang beim Sortieren dar.

```
void swap(int &x, int &y) {
  int help = x;
  x = y;
  y = help;
}
```

Selection Sort, Beispiel

Beispiel

n=6

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	
P	E	V	A	K	S	
P	E	V	**A**	K	S	swap(v[0], v[3])
A	**E**	V	P	K	S	---
A	E	V	P	**K**	S	swap(v[2], v[4])
A	E	K	**P**	V	S	---
A	E	K	P	V	**S**	swap(v[4], v[5])
A	E	K	P	S	V	

Algorithmus

```
void Vector::Selectionsort() {
  int n = Length();
  int i, j, min;
  for(i = 0; i < n; i++) {
    min = i;
    for(j = i+1; j < n; j++)
      if(a[j] < a[min]) min = j;
    swap(a[min], a[i]);
  }
}
```

Selection Sort, Eigenschaften

universität wien

Eigenschaften

- Hauptspeicheralgorithmus
- Insitu Algorithmus
  - sortiert im eigenen Datenbereich, braucht nur eine temporäre Variable, konstanter Speicherplatzverbrauch
- Aufwand
  - generell $O(n^2)$

Bubble Sort Algorithmus

- Wiederholtes Durchlaufen des Vektors.
- Wenn 2 benachbarte Elemente aus der Ordnung sind (größeres vor kleinerem), vertausche (swap) sie.

- Dadurch wird beim ersten Durchlauf das größte Element an die letzte Stelle gesetzt, beim 2. Durchlauf das 2.größte an die vorletzte Stelle, usw. Die Elemente steigen wie Blasen (bubbles) auf.

Beispiel

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	
n=6	P	E	V	A	K	S	
							swaps:
	P	E	V	A	K	S	P-E, V-A, V-K, V-S
	E	P	A	K	S	V	P-A, P-K
	E	A	K	P	S	V	E-A
	A	E	K	P	S	V	
	A	E	K	P	S	V	
	A	E	K	P	S	V	

Algorithmus

```
void Vector::Bubblesort() {
  int n = Length();
  int i, j;
  for(i = n-1; i >= 1; i--)
    for(j = 1; j <= i; j++)
      if(a[j-1] > a[j])
        swap(a[j-1], a[j]);
}
```

Bubble Sort, Eigenschaften

Eigenschaften

- Hauptspeicheralgorithmus
- Insitu Algorithmus
  - sortiert im eigenen Datenbereich, braucht nur eine temporäre Variable, konstanter Speicherplatzverbrauch
- Aufwand
  - generell $O(n^2)$

Quicksort (Hoare 1962) Algorithmus

- Der Vektor wird im Bezug auf ein frei wählbares Pivotelement durch Umordnen der Vektorelemente in 2 Teile geteilt, sodaß alle Elemente links vom Pivotelement kleiner und alle Elemente rechts größer sind.
- Die beiden Teilvektoren werden unabhängig voneinander wieder mit quicksort (rekursiv) sortiert.

divide-and-conquer Paradigma

  - Das Pivotelement steht dadurch auf seinem endgültigen Platz. Es bleiben daher nur mehr n-1 Element (in den beiden Teile) zu sortieren (Verringerung der Problemgröße)
- Beruht auf dem “divide-and-conquer” Ansatz

Pivotelement umordnen

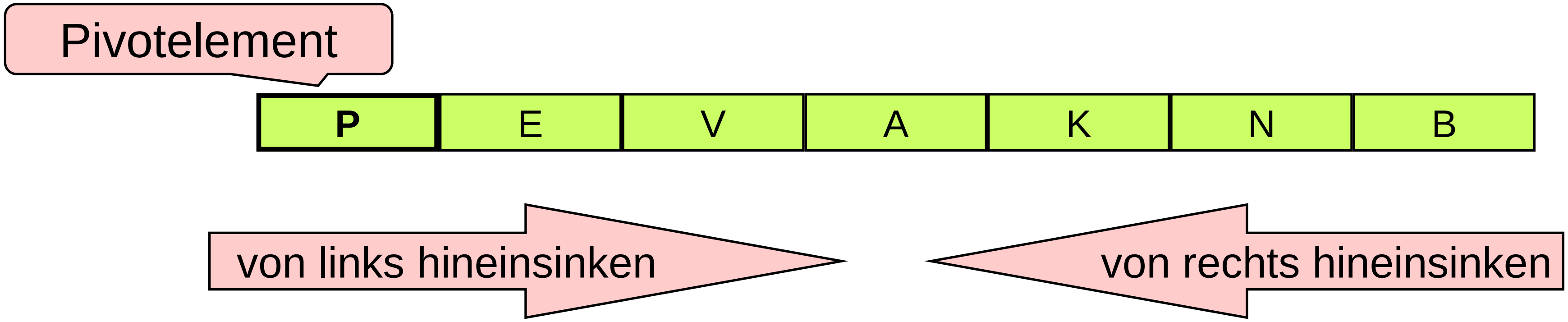
Beliebiges Element als Pivotelement wählen

(im u.a. Bsp. linkes Element im Vektor, auch zufällige Wahl oder Median aus 3 zufälligen üblich)

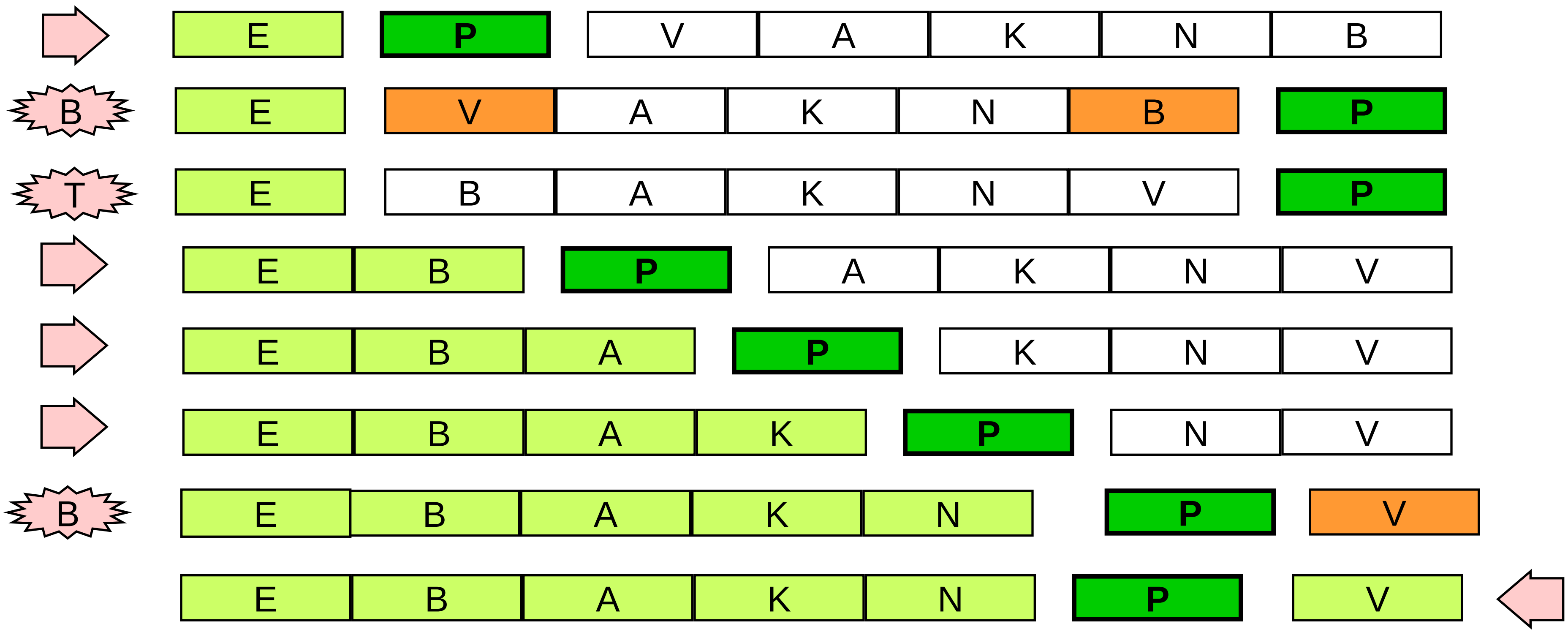
Pivotelement von links bzw. rechts in den Vektor ‘hineinsinken’ lassen,

d.h. jeweils sequentiell von links mit allen kleineren Elementen, von rechts mit allen größeren vertauschen.

Bei Blockierung, d.h. links kleineres und rechts größeres Element, diese beiden Elemente vertauschen



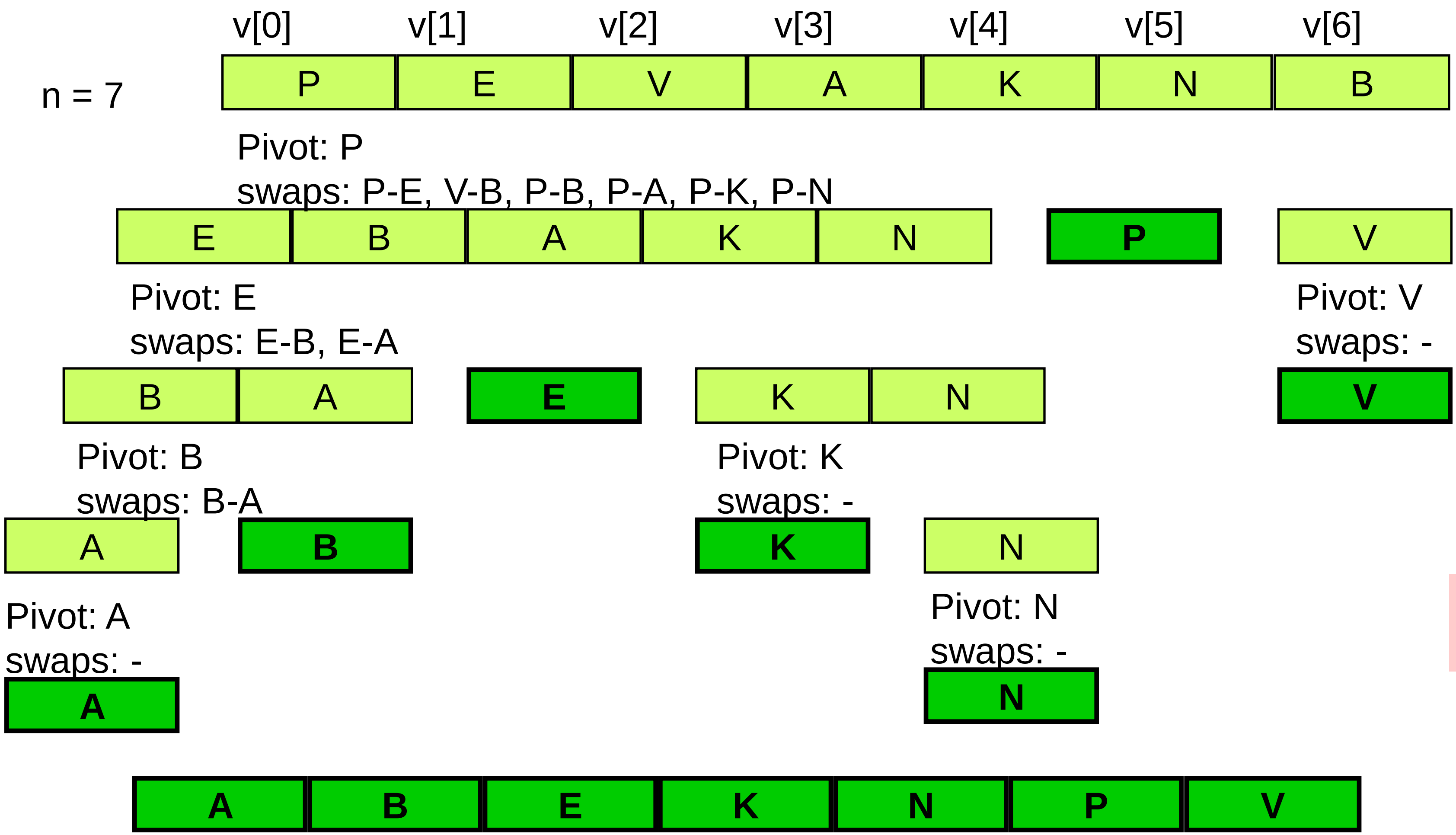
Pivotelement umordnen (2)



von links hineinsinken — E — umgeordnete Elemente

von rechts hineinsinken — B — blockierende Elemente

B,T Blockierung u. Tausch — P — Pivotelement



v[0] v[1] v[2] v[3] v[4] v[5] v[6]
n = 7
P E V A K N B
Pivot: P
swaps: P-E, V-B, P-B, P-A, P-K, P-N
E B A K N
P
V
Pivot: E
swaps: E-B, E-A
Pivot: V
swaps: -
B A
E
K N
V
Pivot: B
swaps: B-A
Pivot: K
swaps: -
A
B
K
N
Pivot: A
swaps: -
Pivot: N
swaps: -
A
N
A B E K N P V

Quicksort, Algorithmus

```
void Vector::Quicksort() {
  quicksort(0, Length()-1);
}
void Vector::quicksort(int l, int r) {
  int i, j; int pivot;
  if(r > l) {
    pivot = a[r]; i = l-1; j = r;
    for(;;) {
      while(a[++i] < pivot);
      while(a[--j] > pivot) if (j == l) break;
      if(i >= j) break;
      swap(a[i], a[j]);
    }
    swap(a[i], a[r]);
    quicksort(l, i-1);
    quicksort(i+1, r);
  }
}
```

Quicksort, Eigenschaften

Hauptspeicheralgorithmus

Insitu Algorithmus

- sortiert im eigenen Datenbereich, braucht nur temporäre Hilfsvariable (Stack), konstanter Speicherplatzverbrauch

Aufwand

- Im Durchschnitt O(n*log(n))
- Kann zu $O(n^2)$ entarten
  - Beispiel: Vektor ist vorsortiert und als Pivotelement wird immer das erste oder letzte Vektorelement gewählt

Empfindlich auf unvorsichtige Wahl des Pivotelements

Stabilität nicht einfach realisierbar

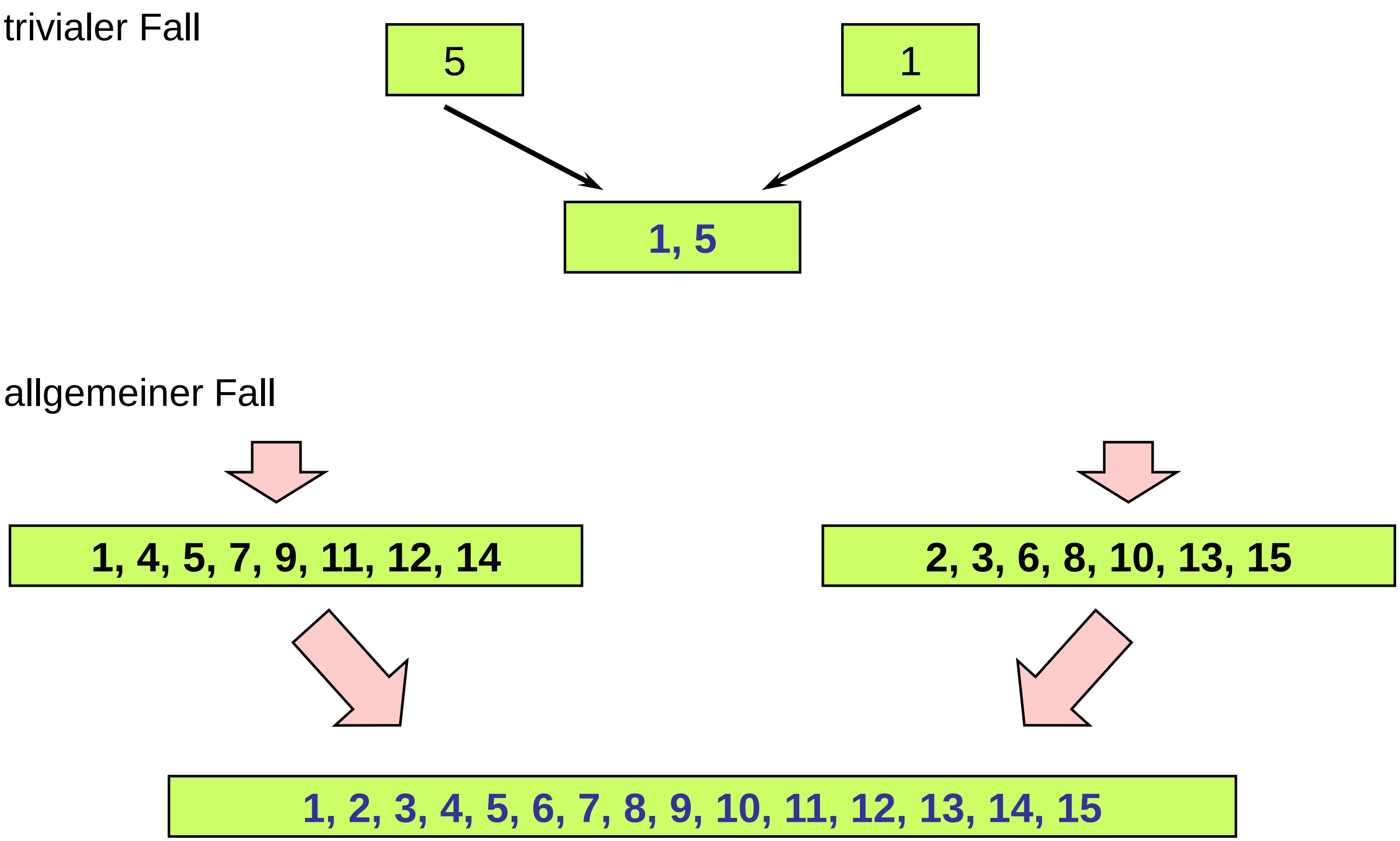
Rekursiver Algorithmus

- Komplex zu realisieren, wenn Rekursion nicht zur Verfügung steht

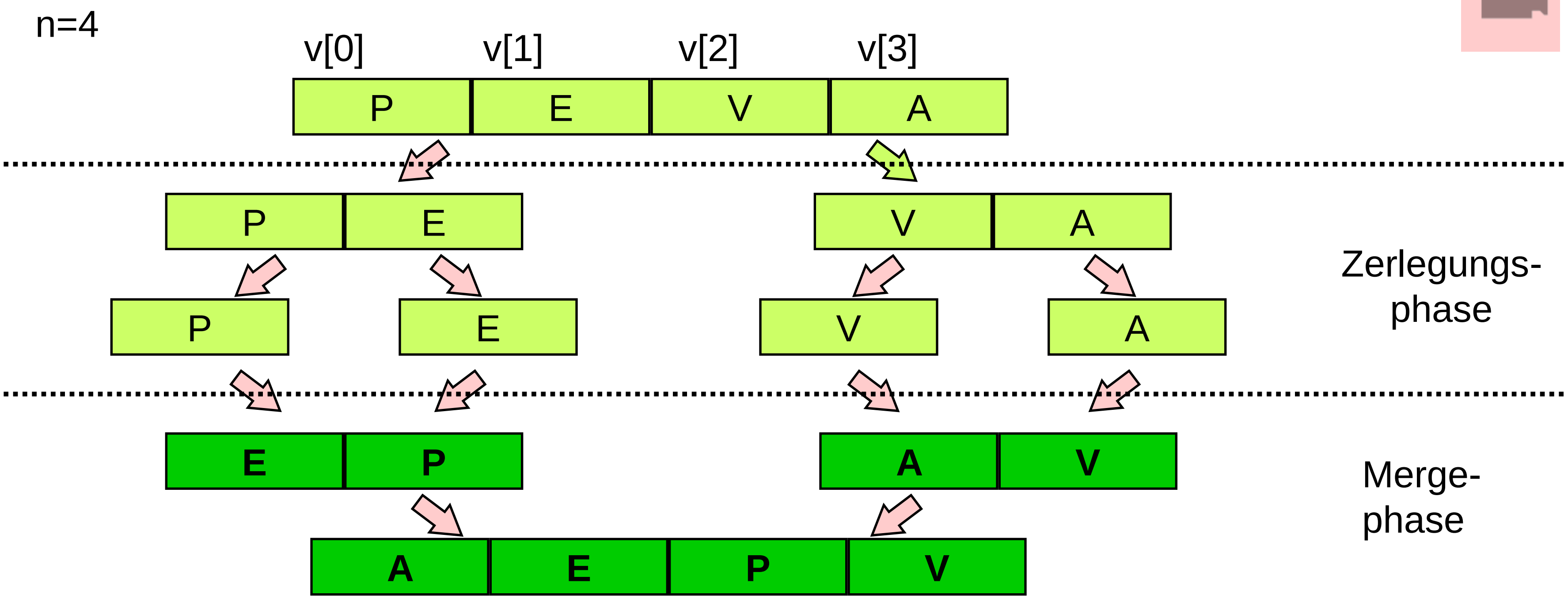
Mergesort (???, 1938) Algorithmus

- Der zu sortierende Vektor wird rekursiv in Teilvektoren halber Länge geteilt, bis die (trivialen, skalaren) Vektoren aus einem einzigen Element bestehen
- Danach werden jeweils 2 Teilvektoren zu einem doppelt so großen Vektor gemerged (sortiert)
  - Beim Mergen werden sukkzessive die ersten beiden Element der Vektoren verglichen und das kleinere in den neuen Vektor übertragen, bis alle Elemente im neuen Vektor sortiert gespeichert sind
  - Das Mergen wird solange wiederholt, bis in der letzten Phase 2 Vektoren der Länge n/2 zu einem sortierten Vektor der Länge n verschmelzen
- “divide-and-conquer” Ansatz

Merging zweier Vektoren



Mergesort, Rekursion



Mergesort, Algorithmus (naiv)

```
void Vector::Mergesort() {
 if (size>1) {
    int m=size/2;
    Vector left(m);
    Vector right(size-m);
    for (int i=0; i<m; ++i) left.a[i]=a[i];
    for (int i=0; i<m; ++i) right.a[i]=a[i+m];

    left.Mergesort();
    right.Mergesort();

    int li=0;
    int ri=0;
    for (int i=0; i<size; ++i)
      if (li>=left.size) a[i]=right.a[ri++];
      else if (ri>=right.size) a[i]=left.a[li++];
      else a[i]=(right.a[ri]<left.a[li])?
                                   right.a[ri++]:left.a[li++];
  }
}
```

korrekt?

Mergesort, Algorithmus (optimiert)

```
void Vector::Mergesort() {
  int *buf=new int[Length()];
  mergesort(0, Length()-1, buf);
  delete[] buf;
}

void Vector::mergesort(int left, int right, int* buf) {
  if(right > left) {
    int i, j, k, m;
    m = (right+left)/2;
    mergesort(left, m, buf);
    mergesort(m+1, right, buf);

    for(i=m+1; i>left; --i) buf[i-1] = a[i-1];
    for(j=m; j<right; ++j) buf[right+m-j] = a[j+1];
    for(k=left; k<=right; ++k)
      a[k]=(buf[i]<buf[j])?buf[i++]:buf[j--];
  }
}
```

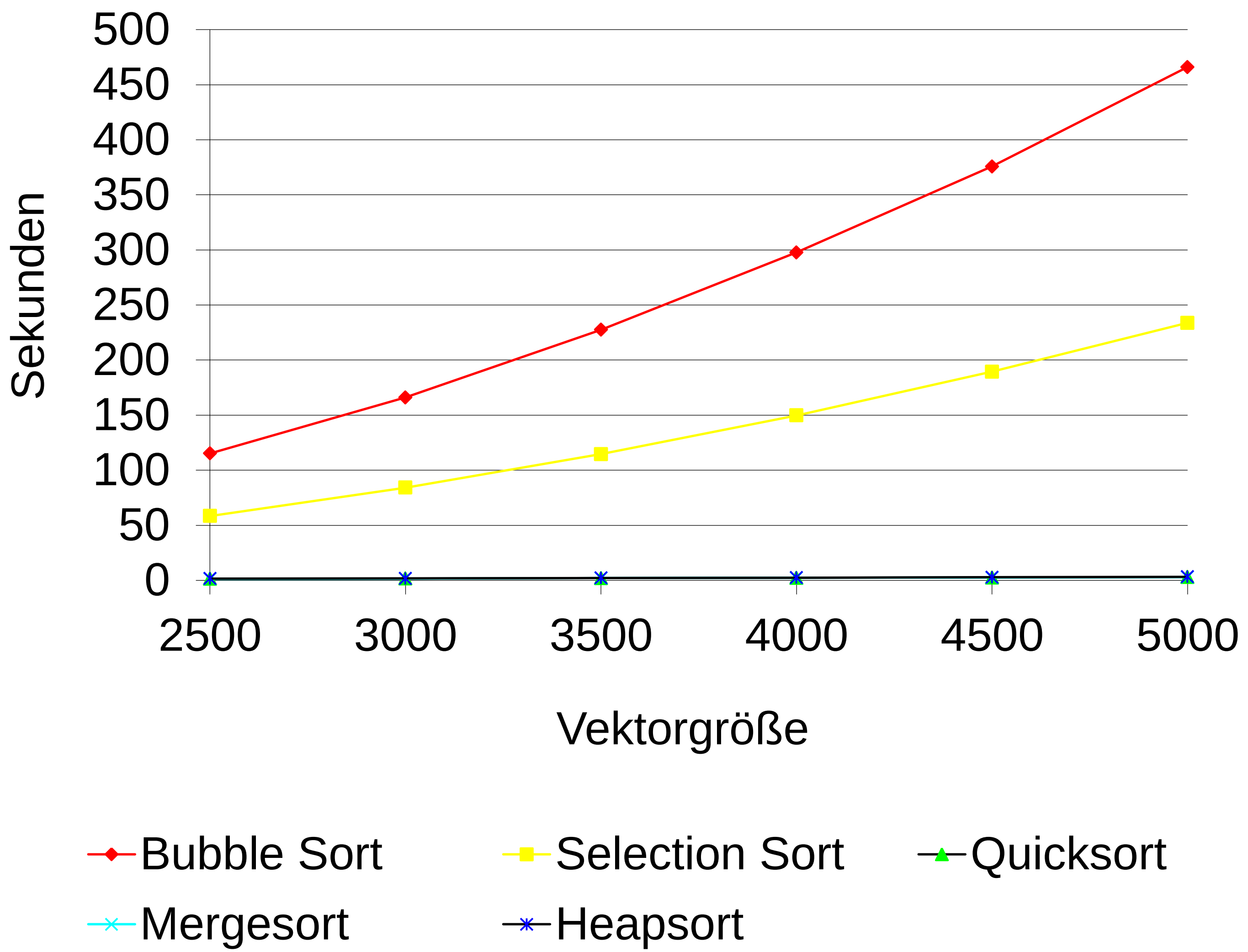
korrekt?

Mergesort, Eigenschaften

Eigenschaften

- Hauptspeicheralgorithmus, aber auch als Externspeicheralgorithmus verwendbar.
  - Extern Sortieren: statt Teilphase wird oft vorgegebene Datenaufteilung genommen, oder nur soweit geteilt, bis sich Teilvektor (Datei) im Hauptspeicher sortieren lässt.
- Stabiles Sortierverfahren
  - Mergen muss korrekt implementiert sein
- Exsitu Algorithmus
  - braucht einen zweiten ebenso großen Hilfsvektor zum Umspeichern
- Aufwand
  - Generell O(n*log(n))
  - Braucht doppelten Speicherplatz
- Rekursiver Algorithmus
  - Komplex zu realisieren, wenn Rekursion nicht zur Verfügung steht.

Laufzeitvergleich der Verfahren



3.4.5 Comparison Sort Verfahren

Man nennt diese bekannten Verfahren *Vergleichende Sortierverfahren* (Comparison sort)

- Die Reihenfolge der Elemente wird dadurch bestimmt, dass die Elemente miteinander verglichen werden
  - Nur Vergleiche der Form $x_i < x_j$, $x_i \leq x_j$, ... angewendet
- Diese Algorithmen zeigen als günstigste Laufzeit eine Ordnung von $O(n \log n)$, d.h. wir konnten bis jetzt als untere Grenze $\Omega(n \log n)$ feststellen

Vermutung: Das Problem des Sortierens durch Vergleichen der Elemente lässt sich mit $\Omega(n \log n)$ beschreiben

Falls gültig $\Rightarrow$ Algorithmische Lücke geschlossen, da $\Omega(P) = O(A)$

Entscheidungsbaum

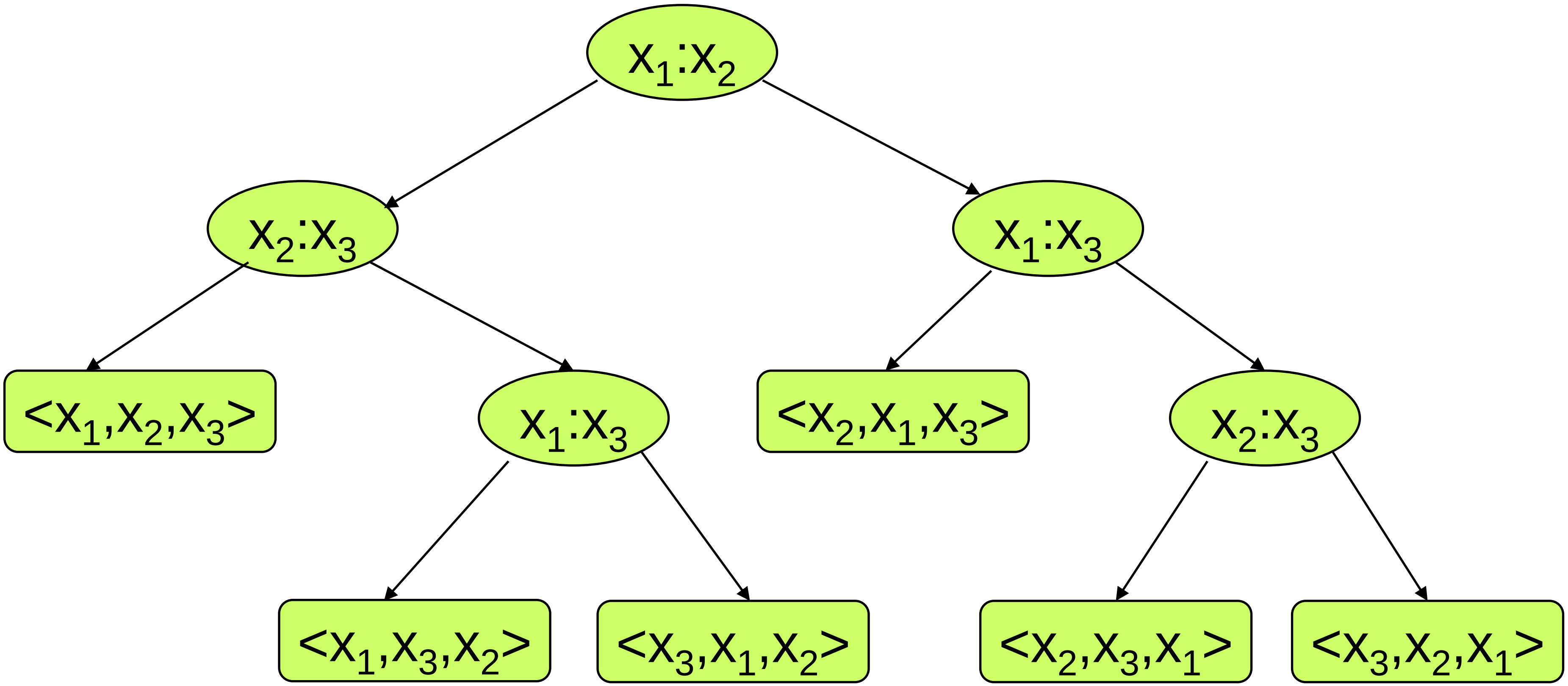
Betrachtung der Vergleichenden Sortierverfahren durch Entscheidungsbäume

- Der Entscheidungsbaum enthält alle möglichen Vergleiche eines Sortierverfahrens um eine beliebige Sequenz einer vorgegebenen Länge zu sortieren
  - Die Blattknoten repräsentieren alle möglichen Permutationen der Eingabe Sequenz
- Die Wege von der Wurzel zu den Blättern definieren die notwendigen Vergleiche um die entsprechenden Permutationen zu erreichen

Entscheidungsbaum für 3 Elemente

d.h. 3! = 6 mögliche Permutationen der Eingabe Elemente < x_1, x_2, x_3 >

Linker Nachfolger Relation x:y erfüllt, rechter Nachfolger nicht erfüllt



Beweis: Untere Grenze für den worst case

Wir ignorieren Swappen, Administrationsoperationen, etc.

Die Anzahl der Blätter im Entscheidungsbaum ist n!

Die Anzahl der Blätter in einem binären Baum ist $\leq 2^h$

daher

$$2^h \geq n!$$

$$h \geq \lg(n!)$$

Stirlingsche Näherung $\quad n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + \theta(1/n)) > \left(\frac{n}{e}\right)^n$

$$h > \lg\left(\frac{n}{e}\right)^n = n \lg n - n \lg e$$

$$h = \Omega(n \lg n)$$

d.h. Algorithmische Lücke für Vergleichende Sortierverfahren geschlossen

3.5 Sortieren in linearer Zeit

Sortierverfahren, die NICHT auf dem Vergleich zweier Werte beruhen, können eine Eingabe Sequenz in linearer Zeit sortieren

- Erreichen die Ordnung O(n)

Beispiele

- Counting Sort
- Radix Sort
- Bucket Sort

3.5.1 Counting Sort

1954 Erstmals verwendet von Harold H.Seward MIT Thesis „Information Sorting in the Application of Electronic Digital Computers to Bussiness Operations“ als Grundlage für Radix Sort.

1956 Erstmals publiziert von E.H. Fried

1960 unter dem Namen Mathsort von W. Feurzeig.

Bedingung

- Die zu sortierenden n Elemente sind Integer Werte im Bereich 0 bis k
  - Es gilt üblicherweise k>n
- Falls **k von der Ordnung n** ist ($k = O(n)$) ist der Aufwand für das Sortierverfahren Counting Sort $O(n)$

Ansatz

- Gegeben sei eine exakte Permutation
- Jedes Element wird einfach auf die Position seines Wertes in einem Zielvektor gestellt

5	8	4	2	7	1	6	3

Counting Sort (2)

Erweiterung

- Die Elemente stammen aus dem Bereich [1..k] und es gibt keine doppelten Werte
- Frage: wie können wir den ersten Ansatz erweitern?

Allgemeiner Fall

- Die Elemente sind aus dem Bereich [1..k], Doppelte sind erlaubt

Idee des Counting Sort

- Bestimme für jedes Element x die Anzahl der Elemente kleiner als x im Vektor, verwende diese Information um das Element x auf seine Position im Ergebnisvektor zu platzieren

Counting Sort Algorithmus

Eingangsvektor **A**, Ergebnisvektor **B**, Hilfsvektor **C**

```
for(i=1; i<=k; i++)
  C[i] = 0;
for(j=1; j<=length(A); j++)
  C[A[j]] = C[A[j]] + 1;
for(i=2; i<=k; i++)
  C[i] = C[i] + C[i-1];
for (j=length(A); j>=1; j--) {
  B[C[A[j]]] = A[j];
  C[A[j]] = C[A[j]] - 1;
}
```

1: C[i] enthält die Anzahl der Elemente gleich i, i = 1,2,...,k

2: C[i] enthält die Anzahl der Elemente kleiner oder gleich i

3: Jedes Element wird an seine korrekte Position platziert

Falls alle Element A[j] unterschiedlich sind ist die korrekte Position in C[A[j]], da Elemente gleich sein können wird der Wert C[A[j]] um 1 verringert

Counting Sort Beispiel

1:

A

3	6	4	1	3	4	1	4

C

2	0	2	3	0	1

2:

C

2	2	4	7	7	8

3:
1. Durchlauf

B

						4	

C

2	2	4	**6**	7	8

3:
2. D.

B

	1					4	

C

1	2	4	6	7	8

3:
3. D.

B

	1				4	4	

C

1	2	4	**5**	7	8

3:
Ende

B

1	1	3	3	4	4	4	6

Counting Sort Analyse

```
for(i=1; i<=k; i++)                          O(k)
  C[i] = 0;
------------------------------------------------
for(j=1; j<=length(A); j++)                  O(n)
  C[A[j]] = C[A[j]] + 1;
------------------------------------------------
for(i=2; i<=k; i++)
  C[i] = C[i] + C[i-1];                      O(k)
------------------------------------------------
for (j=length(A); j>=1; j--) {
  B[C[A[j]]] = A[j];                         O(n)
  C[A[j]] = C[A[j]] - 1;
}
```

- Daraus folgt, dass der Gesamtaufwand O(n + k) ist
- In der Praxis haben wir sehr oft k = O(n), wodurch wir einen realen Aufwand von O(n) erreichen
  - Man beachte, dass kein Vergleich zwischen Werten stattgefunden hat, sondern die Werte der Elemente zur Platzierung verwendet wurden (vergl. Hashing)
- Counting Sort ist ein stabiles Sortierverfahren
  - Garantiert durch letzte Schleife, die von hinten nach vorne arbeitet
- Kein insitu Verfahren, 3 Vektoren der Größe O(n) notwendig

3.5.2 Radix Sort

Radixsort betrachtet die Struktur der Schlüsselwerte

- Die Schlüsselwerte der Länge b werden in einem Zahlensystem der Basis M dargestellt (M ist der Radix)
- z.B. M=2, die Schlüssel werden binär dargestellt, b = 4

	8	4	2	1	Gewicht
9 =	1	0	0	1	b = 4
	3	2	1	0	Stelle #

Es werden Schlüssel sortiert, indem ihre einzelnen Bits an derselben Bit Position miteinander verglichen werden

- Das Prinzip lässt sich auch auf alphanumerische Strings erweitern

Allgemeiner Radix Sort Algorithmus

```
for(Index i läuft über alle b Stellen) {
  sortiere Schlüsselwerte mit einem (stabilen)
    Sortierverfahren bezüglich Stelle i
}
```

Richtung des Indexlaufs, Stabilität

Von links nach rechts

d.h. **`for(int i=b-1; i>=0; i--) { ... }`**

Führt zum *Binären Quicksort (Radix Exchange Sort)*

Von rechts nach links und stabiles Sortierverfahren

d.h. **`for(int i=0; i<b; i++) { ... }`**

Führt zum *LSD-Radixsort (Straight Radix Sort)*

Binärer Quicksort

Sehr altes Verfahren

1929 erstmals für maschinelles Sortieren von Lochkarten.

1954 H.H. Seward [MIT R-232 (1954), p25-28]

eigenständig und ausführlich: P.Hildebrandt , H. Isbitz, H. Rising,
J. Schwartz
[JACM 6 (1959), 156-163]

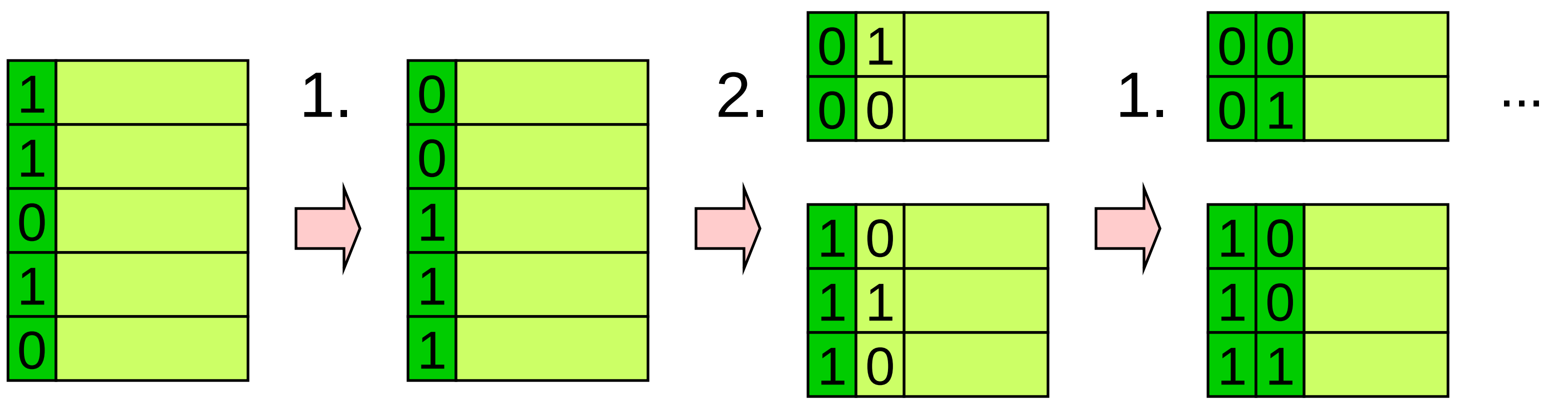
1 Jahr vor Quicksort

Binärer Quicksort Algorithmus

Betrachte Stellen von links nach rechts

1. Sortiere Datensätze bezogen auf das betrachtete Bit
2. Teile die Datensätze in M unabhängige, der Größe nach geordnete, Gruppen und sortiere rekursiv die M Gruppen wobei die schon betrachteten Bits ignoriert werden

Beispiel

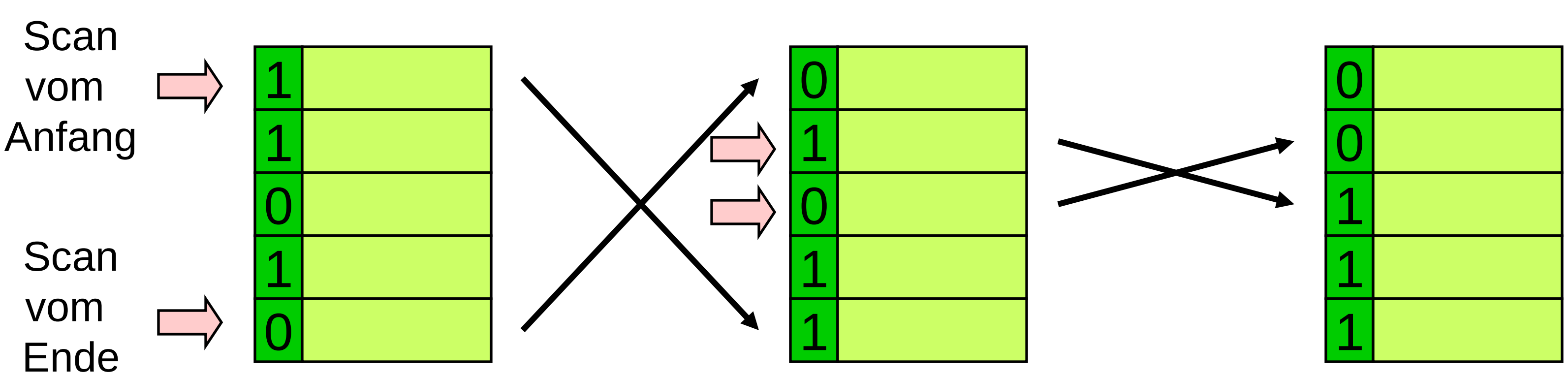


Binärer Quicksort Partition

Aufteilung der Datensätze durch insitu Ansatz ähnlich des Partitionierens beim Quicksort

```
do {
   scan top-down to find key starting with 1;
   scan bottom-up to find key starting with 0;
   exchange keys;
} while (not all indices visited)
```

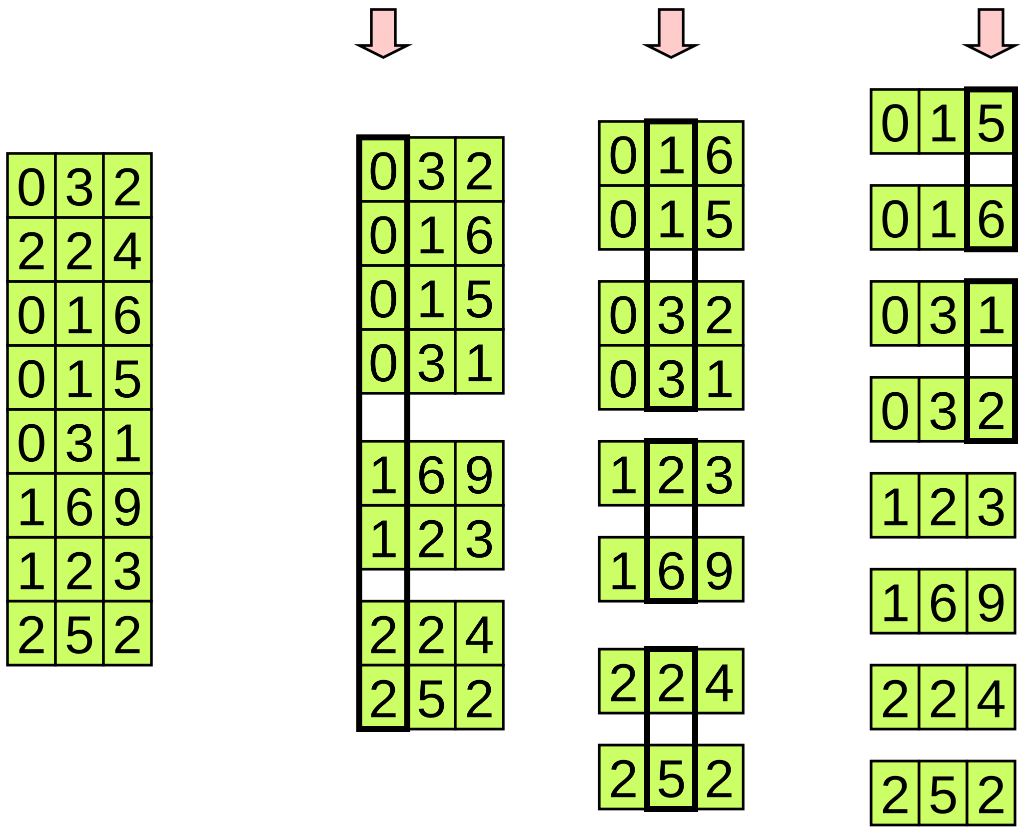
Beispiel



Binärer Quicksort für Dezimalzahlen

Anzahl der Gruppen M ist 10

Zahlenwerte 0 bis 9



Binärer Quicksort Eigenschaften

Binärer Quicksort verwendet im Durchschnitt ungefähr N log N Bitvergleiche

- log N (Kodierung des Zahlenwerts) = konstanter Faktor
- N = Unsicherheit in der Zahlendarstellung

Gut geeignet für kleine Zahlen

Benötigt relativ weniger Speicher als andere Lineare Sortierverfahren

Es ist ein Instabiles Sortierverfahren

Ähnlich dem Quicksort nur für positive ganze Zahlen

LSD-Radixsort Algorithmus

- Betrachte Bits von rechts nach links
  - LSD … „least significant digit“
- Sortiere Datensätze stabil (!) bezogen auf das betrachtete Bit

Bei Binärem Quicksort von links nach rechts!

Zu sortierende Bits pro Durchlauf

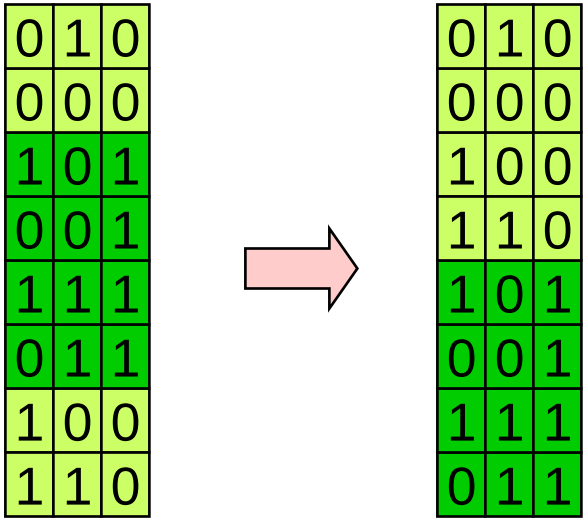
0 1 0	0 1 0	0 0 0	0 0 0
0 0 0	0 0 0	1 0 0	0 0 1
1 0 1	1 0 0	1 0 1	0 1 0
0 0 1	1 1 0	0 0 1	0 1 1
1 1 1	1 0 1	0 1 0	1 0 0
0 1 1	0 0 1	1 1 0	1 0 1
1 0 0	1 1 1	1 1 1	1 1 0
1 1 0	0 1 1	0 1 1	1 1 1

Was bedeutet „stabil“

Bei einem stabilen Sortierverfahren wird die relative Reihenfolge von gleichen Schlüsseln nicht verändert

Beispiel

- Erster Durchgang von letztem Beispiel
- Die relative Reihenfolge der Schlüssel die mit 0 enden bleibt unverändert, dasselbe gilt für die Schlüssel, die mit 1 enden



Stabilität garantiert Korrektheit des Algorithmus

- Mögliches Verfahren: Counting Sort

LSD-Radixsort für Dezimalzahlen

Anzahl der Gruppen M ist 10

Ziffern 0 bis 9

032	031	015	015
224	032	016	016
016	252	123	031
015	123	224	032
031	224	031	123
169	015	032	169
123	016	252	224
252	169	169	252

Analyse von Radix Sort

Sortieren von n Zahlen, Schlüssellänge b

```
for(Index i läuft über alle b Stellen) {
  sortiere n Schlüsselwerte mit einem (stabilen)
    Sortierverfahren bezüglich Stelle i
}
```

Bei der Anwendung eines Sortierverfahrens mit der Laufzeit O(n) (wie z.B. Partitionierung bei Radix Exchange Sort, Counting Sort bei Straight Radix Sort) ist daher die Ordnung von Radix Sort O(bn)

Wenn man b als Konstante betrachtet kommt man auf O(n)

- Bei Zahlenbereich von m Werten Darstellung am Computer mit b = log(m) Stellen, da aber Wertebereich am Computer begrenzt, Ansatz b konstant akzeptierbar

Kein insitu Verfahren, doppelter Speicherplatz notwendig

- führt in der Praxis zum Einsatz von O(n log n) Verfahren

3.5.3 Bucket Sort

Annahme über die n Eingabe Elemente, dass sie gleichmäßig über das Intervall [0,1) verteilt sind

Bucket Sort Algorithmus

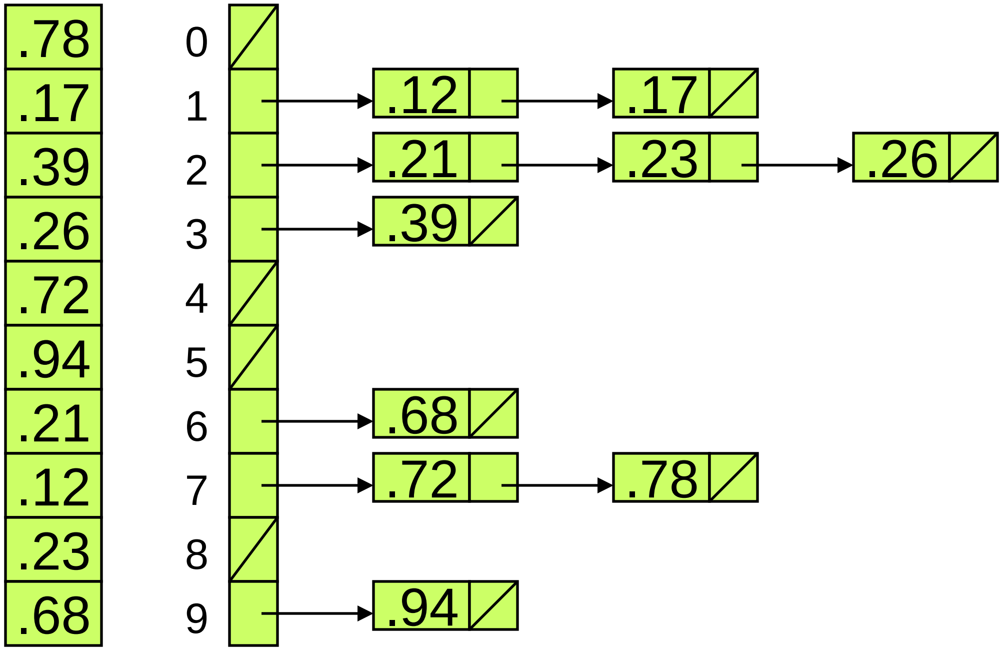
- Teile das Intervall [0,1) in n gleichgroße Teil-Intervalle (Buckets) und verteile die n Eingabe Elemente auf die Buckets
  - Da die Elemente gleichmäßig verteilt sind, fallen nur wenige Elemente in das gleiche Bucket
- Sortiere die Elemente pro Bucket mit Selection Sort
- Besuche die Buckets entlang des Intervalls und gib die sortierten Elemente aus

Bucket Sort Beispiel

Eingabe Vektor A der Größe 10

Bucket Vektor B verwaltet Elemente über lineare Listen

Bucket B[i] enhält die Werte des Intervalls [i/10, (i+1)/10)



Bucket Sort Algorithmus

```
n = length(A);
for(i=1; i<=n; i++)
   insert A[i] into list B[⌊n*A[i]⌋];
for(i=0; i<n; i++)
   sort list B[i] with insertion sort;
Concatenate the lists B[0], B[1], ..., B[n-1]
   together in order
```

Alle Anweisungen außer Sortieren sind von der Ordnung O(n)

Analyse des Sortierens

- n_i bezeichnet die Zufallszahl der Elemente pro Bucket
- Selection Sort läuft in $O(n^2)$ Zeit, d.h. die erwartete Zeit zum Sortieren eines Buckets ist $E[O(n^2)] = O(E[n^2])$, da alle Buckets zu sortieren sind

$$\sum_{i=0}^{n-1} O(E[n_i^2]) = O(\sum_{i=0}^{n-1} E[n_i^2]) \Rightarrow O(n)$$

- Um den Ausdruck zu berechnen
  Verteilung der Zufallsvariable n_i bestimmen
- Es gibt n Elemente mit n Buckets, d.h. die Wahrscheinlichkeit eines Elements in ein Bucket zu fallen ist $p = 1/n$
- n_i folgt der Binomial-Verteilung
  - $E[n_i] = n*p = 1$ und $Var[n_i] = n*p*(1-p) = 1 - 1/n$
- $E[n_i^2] = Var[n_i] + E^2[n_i] = 1 - 1/n + 1^2 = 2 - 1/n = \Theta(1)$

Daraus folgt, dass der Aufwand für Bucket Sort O(n) ist

Was nehmen wir mit?

Dictionary

Hashing

- Statische und dynamische Verfahren

Sortieren

- Klassische Verfahren $O(n^2)$ und $O(n \log n)$
- Lineare Verfahren $O(n)$